André Vinícius Mulho da Costa

Implementação do C-MAC para o Epos Mote

Florianópolis – SC

Outubro de 2009

André Vinícius Mulho da Costa

Implementação do C-MAC para o Epos Mote

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau Bacharel em Ciências da Computação.

Orientador:
Prof. Dr. Antônio Augusto Medeiros Fröhlich

Co-orientador:
Giovani Gracioli

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis – SC

Outubro de 2009

“Apenas o desconhecido amedronta o homem. Mas, uma vez que o homem enfrenta o desconhecido, esse medo se torna conhecido”

Antoine de Saint-Exupéry

Resumo

Nas últimas décadas, houve um grande avanço tecnológico nas áreas de sensores, circuitos integrados e comunicação sem fio, que possibilitou a criação de dispositivos de sensoriamento sem fio. Estes dispositivos trabalham com recursos extremamente limitados, e sua comunicação quando feita deve ser eficiente, para que não ocorram desperdícios.

A comunicação pelos nodos é feita através do protocolo de comunicação MAC(*Media Access Control*), que funciona como um controlador para o meio de comunicação, indicando o momento de receber e transmitir. Analisando a grande gama de protocolos de comunicação para redes de sensores sem fios, é possível verificar que cada uma delas possui a solução para problemas específicos, assim como cada uma delas possui características em comum.

Com o intuito de criar um protocolo que pode assumir as principais características observadas nos demais protocolos, foi criado o C-MAC, que originalmente foi implementado para o Atmega128 e para o rádio CC1000. A primeira versão do C-MAC conta com uma máquina de estados onde a transmissão e a recepção são disparados por um temporizador dedicado de tempo fixo, que é configurado em tempo de compilação. Este trabalho apresenta o projeto e implementação do C-MAC para o Atmega1281 e o rádio AT86RF230, foi utilizado a mesma máquina de estados do seu predecessor com a adição de novos estados e possibilidades, assim como um novo temporizador dedicado que pode ser configurado em tempo de execução.

A fim de otimizar a transmissão no Epos Mote, aproveitando o melhor que cada protocolo tem a oferecer para um cenário específico, o C-MAC foi reimplementado para o rádio do Zigbit podendo se configurar diversas características, para cada combinação de características configurada uma nova máquina de estados do protocolo é gerada em tempo de compilação. Cada uma dessas máquinas de estados serão exibidas no trabalho a seguir, assim como seus testes de consumo de memória e desempenho.

Abstract

In recent decades, there has been a major technological breakthrough in the areas of sensors, integrated circuits and wireless communication, which enabled the creation of wireless sensing devices. These devices work with extremely limited resources, and their communication when happen must be very efficient.

The communication between nodes are done through the communication protocol (*MAC - Media Access Control*) , which acts as a driver for the medium, indicating the time to receive and transmit. Looking at the wide range of communication protocols for networks of wireless sensors, we can see that each has the solution to specific problems, and each has points in common.

In order to create a protocol that can assume the main features observed in other protocols, has created the C-MAC, which was originally implemented for the Atmega128 and on CC1000 radio. The first version of C-MAC has a state machine where the transmission and reception are triggered by a dedicated fixed timer, which is configured at compile time. This presents the design and implementation of C-MAC for the ATmega1281 and AT86RF230 radio, we used the same state machine of its predecessor with the addition of new states and possibilities, as well as a new dedicated timer that can be configured in execution time.

In order to optimize the transmission in Epos Mote, integrating the best that each protocol has to offer for a particular scenario, the C-MAC has been redeployed to the radio Zigbit able to configure several features for each combination of features set a new machine state of the protocol is generated at compile time. Each of these state machines are shown in the following work, as well as testing their memory consumption and performance.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

O constante avanço na eletrônica permitiu o surgimento de “uma nova categoria de aplicações para o conceito fundamental de computador, na forma de micro-sensores sem fios de baixo consumo de energia”(Wanner 2006). Os sensores possuem um módulo de sensoriamento (e.g., sensor de temperatura, acústico), um processador digital, um módulo de comunicação sem fio e um módulo de energia.

Cada sensor é capaz de obter dados sobre um local específico. Ao se monitorar um ambiente com nodos sensores, informações em tempo real podem ser coletadas daquela região e transferidas para um computador, onde as mesmas podem ser analisadas a fim de haver respostas mais rápidas a eventos ocorridos naquele ambiente. Isso envolve monitoramento de florestas de maneira a prevenir incêndios, ou até mesmo regular a umidade do solo em grandes plantações para aumento de produção.

A capacidade computacional dos nodos faz com que o sistema opere com recursos extremamente limitados, além disso, a grande variabilidade de arquiteturas faz com que não seja possível garantir a funcionalidade de uma aplicação em diferentes arquiteturas. Para que esses problemas fossem solucionados, foram criados os sistemas operacionais para sistemas embarcados. Outro problema é que a comunicação entre os nodos são feitas através de rádios que concorrem entre si para estabelecer comunicação, se esses rádios tentarem se comunicar de maneira desordenada um grande desperdício de recursos ocorrerá (em termos de consumo de energia), para evitar esse desperdício e garantir a comunicação surgem os protocolos de comunicação.

A maioria dos protocolos de comunicação foram feitos para links de satélite e redes wireless locais, onde os pontos fortes devem ser velocidade de comunicação alta e atraso baixo. Em protocolos para redes de sensores sem fios, os pontos principais devem estar relacionados a eficiência na comunicação, já que essa é extremamente custosa para um nodo, não existindo importância fundamental na velocidade de comunicação.

O C-MAC foi implementado no sistema operacional EPOS, ele é um protocolo configurável

de acesso ao meio para redes de sensores sem fios de baixa potência. A versão original do C-MAC foi implementada no Atmega128 utilizando o rádio CC1000, ela possui uma máquina de estados onde os processos de transmissão e recepção são disparados por um temporizador dedicado de tempo fixo que é configurado em tempo de compilação.

Com a criação da plataforma do Epos Mote que utiliza um ZigBit (esse por sua vez contêm um Atmega1281 e um rádio AT86RF230) deu-se necessária a criação de um novo C-MAC que teria as mesmas características do seu predecessor, além de algumas novas possibilidades de configuração. A idéia foi utilizar a mesma máquina de estados com as adições de possibilidade de acks, backoff feito por CSMA-CA[(IEEE 2007)]. Outra melhoria é a utilização de um temporizador dedicado de disparo único (*Single Shot Timer*), que poderá assumir novos valores em tempo de execução, de maneira a adaptar o tempo de ciclo ativo.

1.1 Escopo do Trabalho

O trabalho se localiza no escopo de Redes de Sensores Sem Fios, sendo assim uma breve introdução sobre o assunto será abordada nesta sessão.

Uma rede de sensores sem fios é constituida de dispositivos autônomos, distribuidos por uma região, onde cada um monitora uma parcela de um ambiente, contribuindo para a monitoração completa de uma região. A Figura 1.1 mostra as principais partes de um nodo sensor sem fio.

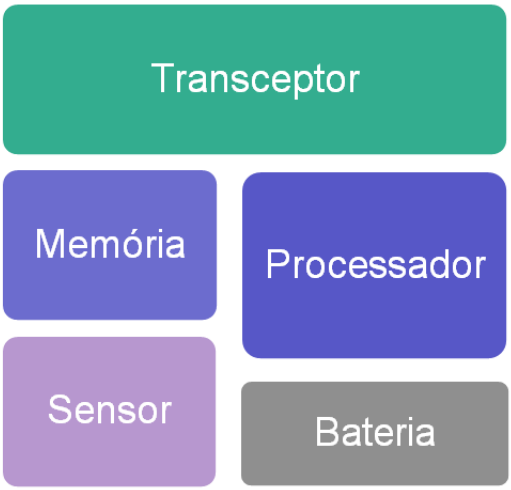

Figura 1.1: Componentes básicos de um nodo sensor sem fio.

Cada nodo sensor é dotado de um transceptor de rádio, uma memória, um microprocessador, um ou mais sensores e uma Bateria. A memória e o processador normalmente são inte-

grados em um microprocessador. O nodo é equipado normalmente com um ou mais sensores, que servem para monitorar as condições ambientais (e.g. som, luminosidade, pressão, temperatura, umidade) transformando-as em informação. Uma vez que essa informação é obtida o sensor pode utilizar o transceptor de rádio para transmití-la para o meio externo, normalmente o transceptor de rádio é um rádio de baixa potência.

A autonomia de um nodo sensor é possível graças a utilização de bateria e hardwares voltados ao baixo consumo de energia. Além da utilização de hardware especializado, o software também deve estar voltado a maximização da bateria. O software é parte importante no consumo de energia, já que a diferença entre o consumo energético de um sleep (rádio desligado) e uma transmissão (rádio ligado) pode chegar a até 2830 vezes utilizando a plataforma de desenvolvimento Zigbit, como podemos observar na Tabela 2.1. Devido a essa grande diferença de consumo, o tempo em que o rádio fica ligado deve ser severamente controlado pelo software.

Com base em pesquisas e aplicações atuais [(Pottie e Kaiser) (Barr John C. Bicket e Sirer.)], é possivel definir que os principais requisitos para os nodos sensores sem fios são:

1. Ter dimensões físicas reduzidas - Para poderem ser instalados de maneira a não atrapalhar o ambiente destino os nodos devem ter dimensões reduzidas. Dado o constante avanço na miniaturização do hardware, o nodo tende a diminuir mais, porém ficará limitado ao tamanho de sua bateria.

2. Ser capaz de operar por um longo período com quantidade limitada de energia - Tendo em vista que o nodo deve operar durante um longo periodo de tempo com recursos energéticos limitados, o projeto do hardware de um nodo sensor deve priorizar o consumo de energia reduzido.

3. Permitir a mais ampla configuração possível no canal de transmissão de dados - O transceptor de rádio é normalmente o componente no nodo sensor que mais consume energia [(Langendoen e Halkes)], desta maneira é importante que esse fique a maior parte do tempo desligado. Com a configuração do canal de dados, a aplicação poderá se beneficiar de melhorias no consumo energético sem que o rádio necessite ficar desligado muito tempo, assim não comprometendo a comunicação.

Com base nos requisitos apresentados acima, foram criadas as seguintes plataformas de hardware para redes de sensores sem fios: Mica2 [(Crossbow)], Telos [(Moteiv. 2005)], BTnode [(ETH 2005)] e o Epos Mote.

Redes de sensores sem fios possuem uma grande gama de aplicações que envolvem monitoramento de ambientes, automação industrial e residencial. Ao se monitorar um ambiente com nodos sensores, informações em tempo real podem ser coletadas daquela região e transferidas para um computador, onde as mesmas podem ser analisadas a fim de haver respostas mais rápidas a eventos ocorridos naquele ambiente. Isso envolve monitoramento de florestas de maneira a prevenir incêndios, ou até mesmo regular a umidade do solo em grandes plantações para aumento de produção.

Em de Redes de sensores sem fios, a área abordada no trabalho é a da comunicação feita entre os nodos sensores através do rádio onde o nodo utilizado foi o Epos Mote e o protocolo MAC de comunicação o C-MAC.

1.2 Objetivos gerais

Integrar o C-MAC a Árvore do Epos Mote de maneira a adicionar novas possibilidades de configuração.

A fim de alcançar o objetivo principal, são estabelecidos os seguintes objetivos específicos:

1. Estudo dos principais protocolos de comunicação para RSSF.
2. Entender o C-MAC, comparando-o a outros protocolos de comunicação, a fim de obter um melhor embasamento teórico.
3. Adaptar o atual C-MAC para que esse utilize um novo mediador de rádio (AT86RF230).
4. Utilizar o Temporizador de disparo único no C-MAC para melhorar a precisão do temporizador e torná-lo reconfigurável.
5. Implementar interrupções no AT86RF230 e integração no C-MAC.

1.3 Organização do Texto

O texto deste trabalho está organizado da seguinte maneira:

Capítulo 2 descreve conceitos e tecnologias utilizadas Projeto do Epos Mote, assim como o sistema operacional e o hardware utilizados.

Capítulo 3 descreve o funcionamento das comunicações em Redes de Sensores sem Fios, mostrando os principais protocolos e o C-MAC.

Capítulo 4 descreve os principais passos da nova implementação do C-MAC.

Capítulo 5 mostra os resultados obtidos.

Capítulo 6 conclui a dissertação e apresenta os trabalhos futuros.

2 *Projeto EPOS Mote*

No Lisha foi desenvolvido a placa para uso em campo que possui um chip Zigbit. Ela foi feita com base na placa de desenvolvimento MeshBeans 2 que possui diversos componentes além do ZigBit, a maior parte deles para fazer o interfaceamento do kit com um PC (USB ou RS232). Tendo em vista que grande parte dos componentes presentes na MeshBeans não são necessários para uma placa de uso em campo, vários desses foram removidos. Com a remoção de grande parte dos componentes em relação a MeshBeans foi possível obter uma placa final com tamanho bem reduzido como é possível verificar na Figura 2.1.

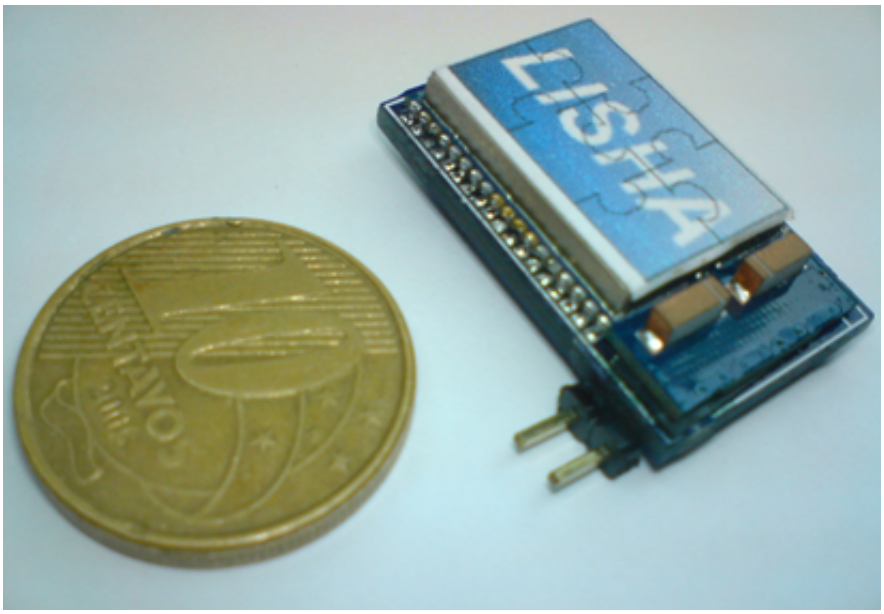

Figura 2.1: Placa do Epos Mote desenvolvida no Lisha.

A plataforma também conta com o sensor de umidade e temperatura SHT11. O sensor se comunica com o Atmega1281 através de dois pinos, sendo um para receber o clock e outro para dados, sendo que o protocolo de comunicação foi todo implementado em software.

Para que fosse possível a utilização do Epos nesse novo Hardware, foi necessário portá-lo para o Atmega1281. Esse porte tomou como base um já existente para o Atmega128, onde apenas foi necessário a mudança do endereço de alguns registradores, e algumas adaptações

para que os temporizadores funcionassem corretamente. As informações referentes ao porte do sistema fogem do escopo do trabalho.

2.1 Plataforma de Hardware

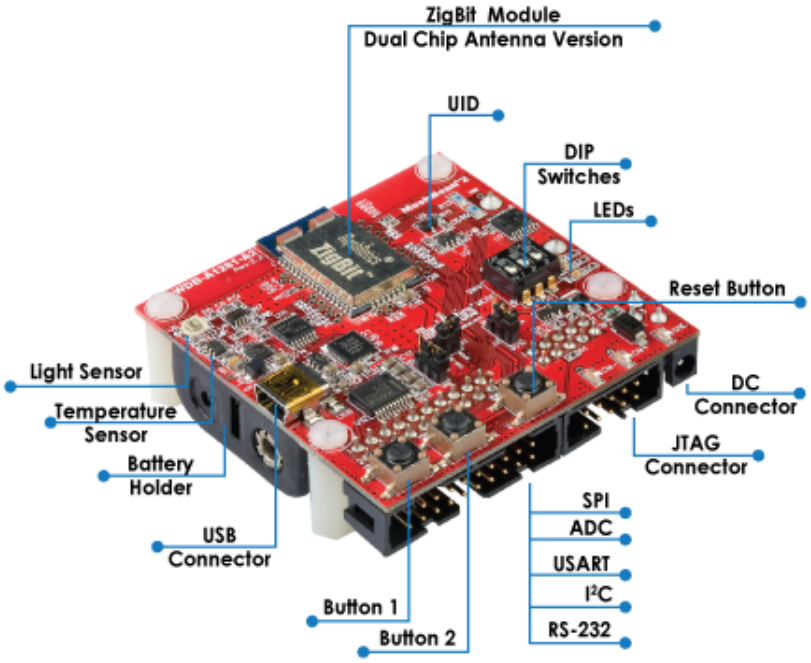

Figura 2.2: Placa de desenvolvimento Zigbit.

A empresa MeshNetics produziu uma família de placas de desenvolvimento baseadas no chip Zigbit, que pode ser visualizada na Figura 2.2. O chip possui um microcontrolador Atmel Atmega1281V que roda a 8Mhz, com 8KB de Ram e 128KB de memória Flash, além de 4KB de EEPROM.

O rádio utilizado no chip é o AT86RF230 capaz de operar entre 2400-2483MHz. A alimentação pode tanto ser feita através de um par de pilhas AA, fonte AC/DC ou ainda USB. A placa de desenvolvimento também possui sensores de luminosidade e temperatura e 3 leds. São várias as interfaces de comunicação: USB, RS-232, USART, I2C, SPI e GPIO. A Tabela 2.1 apresenta o consumo energético do chip Zigbit em diferentes atuações, os valores da tabela foram medidos em testes no laboratório, removendo o jumper J1, e conectando um amperímetro nos terminais CM- e CM+.

Tabela 2.1: Consumo do módulo Zigbit.

Estado	**Consumo (mA)**
Sleeping	0,0053
Rádio Enviando	14,9
Rádio Recebendo	16,6

2.1.1 Rádio AT86RF230

O AT86RF230 é um rádio de baixo consumo que trabalha na faixa de 2,4 GHz e é focado em aplicações que utilizam os protocolos ZigBee e IEE802.15.4. O AT86RF230 usa o barramento SPI para se conectar com o microcontrolador.

O rádio fornece algumas funções em seu hardware como o processo de CCA (*Clear Channel Assessment*), fornecimento automático de Ack, filtro de endereços, escolha automática da frequência utilizada entre outras funções.

O rádio conta com uma máquina de estados interna que pode ser visualizada na Figura 2.3. Ela fornece as funções básicas como receber e transmitir dados, ligar e desligar o rádio. Os estados dessa máquina são controlados por dois pinos de sinal e o registrador TRX-STATE, a mudança de um estado pode ser confirmada pela leitura do registrador TRX-STATUS.

2.2 EPOS

Em redes de sensores sem fios, devido ao grande número de requisistos específicos de hardware nas aplicações um grande número de plataformas é gerado. E para que seja possível utilizar uma aplicação feita para determinada plataforma de hardware em outra o sistema operacional deve oferecer mecanismos para abstrair diversas plataformas de hardware, assim como deve garantir que qualquer suporte feito em tempo de execução não consuma recursos excessivos. De acordo com (Wanner 2006), os requisitos para um sistema operacional de redes de sensores sem fios são:

1. Fornecer a funcionalidade básica de um sistema operacional.
2. Fornecer mecanismos para a gerência do consumo de energia para os nodos.
3. Prover mecanismos para reprogramação em campo.
4. Abstrair o hardware de sensoriamento heterogêneo de maneira uniforme.
5. Fornecer uma pilha de protocolos de comunicação configurável.

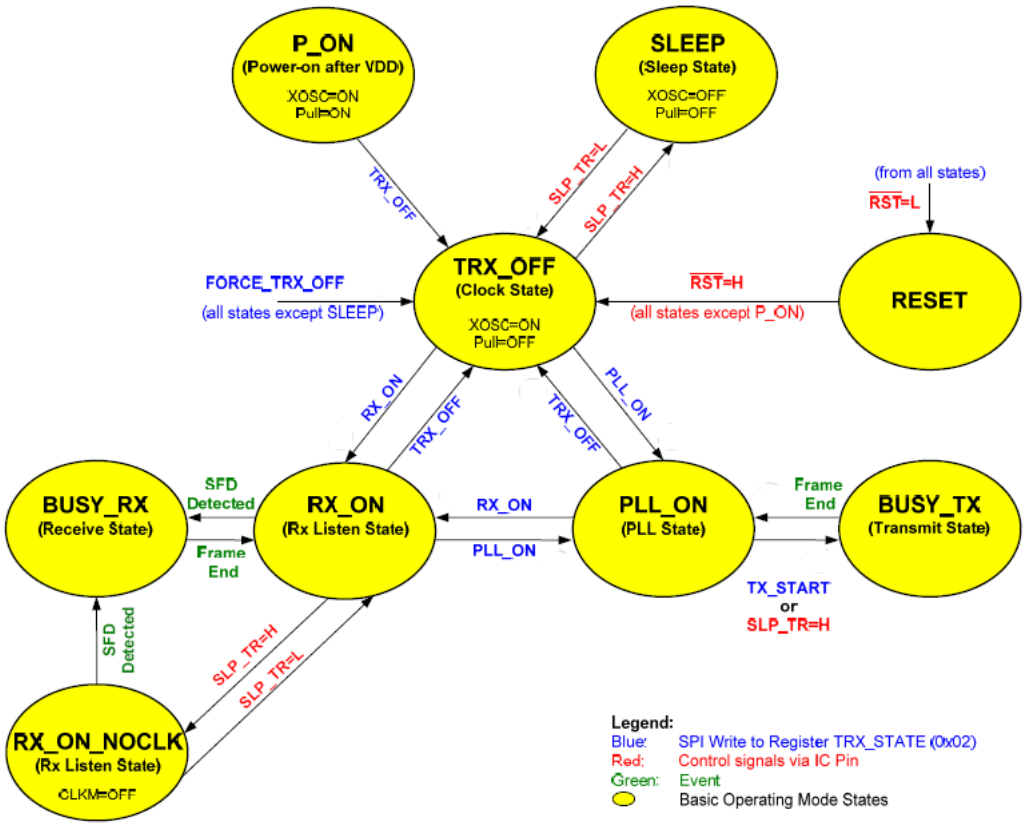

Figura 2.3: Máquina de estados do AT86RF230.

6. Operar com recursos limitados.

Tendo em vista os requisitos para um sistema operacional de redes de sensores, foi escolhido o EPOS [(Fröhlich 2001), (Polpeta et al. 2004) (Polpeta et al. 2004)] (*Embedded Parallel Operating System*), que é um framework baseado em componentes que geram sistemas de suporte de execução a aplicações. O EPOS foi desenvolvido e implementado seguindo a metodologia de Projeto de Sistemas Embarcados Orientados à Aplicação (ADESD - *Application Driven Embedded System Design*)[(Fröhlich 2001)], que mantém a portabilidade em sistemas embarcados.

O sistema suporta várias arquiteturas computacionais como: MIPS, IA32, PowerPC, H8, Sparc e AVR. No contexto deste trabalho, a árvore utilizada será uma para o AVR, que possui o microcontrolador Atmel Atmega1281.

3 *Controle de Acesso ao Meio em Rede de Sensores Sem Fios*

Durante a comunicação dos nodos sensores, a "transmissão de um único byte de dados pelo rádio consome mais energia do que a execução de dezenas de instruções no microcontrolador principal"(Wanner, Oliveira e Fröhlich 2007), por isso a comunicação, ou a tentativa dela deve ser otimizada.

Devido ao grande consumo energético da comunicação em redes de sensores sem fios ela é feita através de um protocolo de acesso ao meio (*Media Access Control - MAC*) que controla como e quando um nodo deve transmitir, receber ou escutar o meio.

De acordo com (Polastre, Hill e Culler 2004), os objetivos que devem ser alcançados na implementação de um protocolo MAC para redes de sensores sem fios são:

1. Operação com baixo custo energético
2. Sistema para evitar colisão de dados
3. Implementação simples , de código pequeno e pouco custoso para a RAM
4. Utilização eficiente do canal de comunicação
5. Tolerante a variações de rádio frequência
6. Escalável para um grande número de nodos

Dentre os principais problemas encontrados nos protocolos de acesso ao meio podemos citar os seguintes:

1. Escuta sem tráfego - Tempo que o nodo fica escutando sem que efetivamente uma mensagem seja recebida.
2. Colisões de Dados - Quando dois dados são enviados ao mesmo tempo, ambos são perdidos, assim inutilizando a transmissão.

3. Escuta desnecessária - Ocorre quando um nodo recebe um pacote, e só depois verifica que a mensagem não é para ele, assim se torna necessário o descarte do pacote.

4. Flutuações no Tráfego - Ocorre quando um fenômeno é detectado por vários nodos ao mesmo tempo (ex: variação brusca na medida de um valor), então vários nodos concorrem pelo meio de transmissão.

Cada um dos problemas citados acima possuem soluções simples, que são implementadas nos principais protocolos para Redes de Sensores Sem Fios. As principais soluções são mostradas nas próximas sessões.

Os protocolos de acesso ao meio para redes de sensores sem fios podem ser classificados em três categorias (Klues et al. 2008): os que escutam continuamente (*Channel Polling*), escuta programada (*Scheduled Contention*) e tempo dividido entre os nodos (*Time Division Multiple Access - TDMA*), além dos protocolos híbridos.

3.1 Protocolos de Escuta Contínua

Protocolos de escuta contínua (*Channel Polling*), são normalmente os mais simples, já que esses passam todo o seu tempo em um ciclo que varia entre dormir, acordar e escutar o rádio a procura de alguma atividade. Se alguma atividade é encontrada o dispositivo se mantém ligado a fim de finalizar a comunicação, senão ele volta a dormir. Para que ocorra a sincronização entre o nodo que está enviando um pacote e o nodo que receberá o pacote, existe o tempo de transmissão de dados - previamente conhecido - chamado Preâmbulo.

Protocolos baseados em escuta contínua são robustos em relação a flutuações de tráfego. É possível obter boa eficiência energética aumentando o preâmbulo, assim o canal será verificado com uma periodicidade menor (Wanner 2006). Exemplos de protocolos de Escuta Contínua são o B-MAC[(Polastre, Hill e Culler 2004)] e o X-MAC[(Buettner et al. 2006)], descritos abaixo.

3.1.1 B-MAC

De acordo com a Figura 3.1 comunicação no B-MAC (*Berkeley MAC*) ocorre quando o nodo emissor após verificar o canal com o processo de CCA inicia a transmissão do preâmbulo. Se no momento em que o preâmbulo é transmitido existe um nodo receptor em estado de escuta (LPL - *Low Power Listening*) a comunicação ocorre.

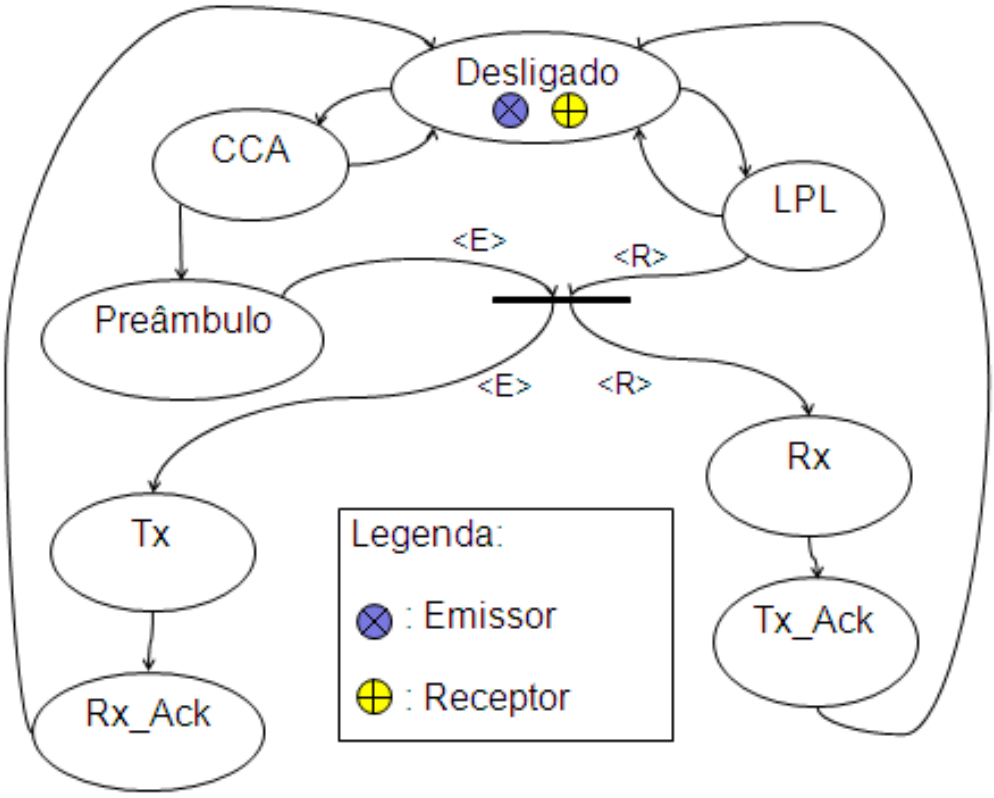

Figura 3.1: Protocolo de comunicação B-MAC.

3.1.2 X-MAC

O X-MAC é a evolução do protocolo B-MAC já que ele aproveita a transmissão do preâmbulo para embutir o endereço do nodo receptor. Com essa melhoria o problema da escuta desnecessária é resolvido, já que um nodo que está em estado de LPL que começa a receber um preâmbulo consegue verificar nesse momento se ele é o destinatário da mensagem, se não for, esse desliga o rádio.

3.2 Protocolos de Escuta Programada

Protocolos de Escuta Programada (*Scheduled Contention*), programam o tempo em que todos os nodos devem acordar, e os que tem dados pendentes a trasmitir passam pelo processo de CSMA-CA. O processo de CSMA-CA faz com que colisões sejam evitadas, aumentando a eficiência do protocolo.

Os protocolos de Escuta Programada reduzem o consumo energético através do limite da transmissão a períodos bem delimitados. Exemplos de protocolos de Escuta Programada são o S-MAC [(Ye, Heidemann e Estrin 2002)] e o T-MAC[(Dam e Langendoen 2003)].

3.2.1 S-MAC

De acordo com a Figura 3.2 comunicação no S-MAC (*Sensor MAC*) ocorre quando após um intervalo de tempo pré-definido, os nodos emissor e receptor acordam e entram em estado de sincronização, e após isso a comunicação ocorre normalmente.

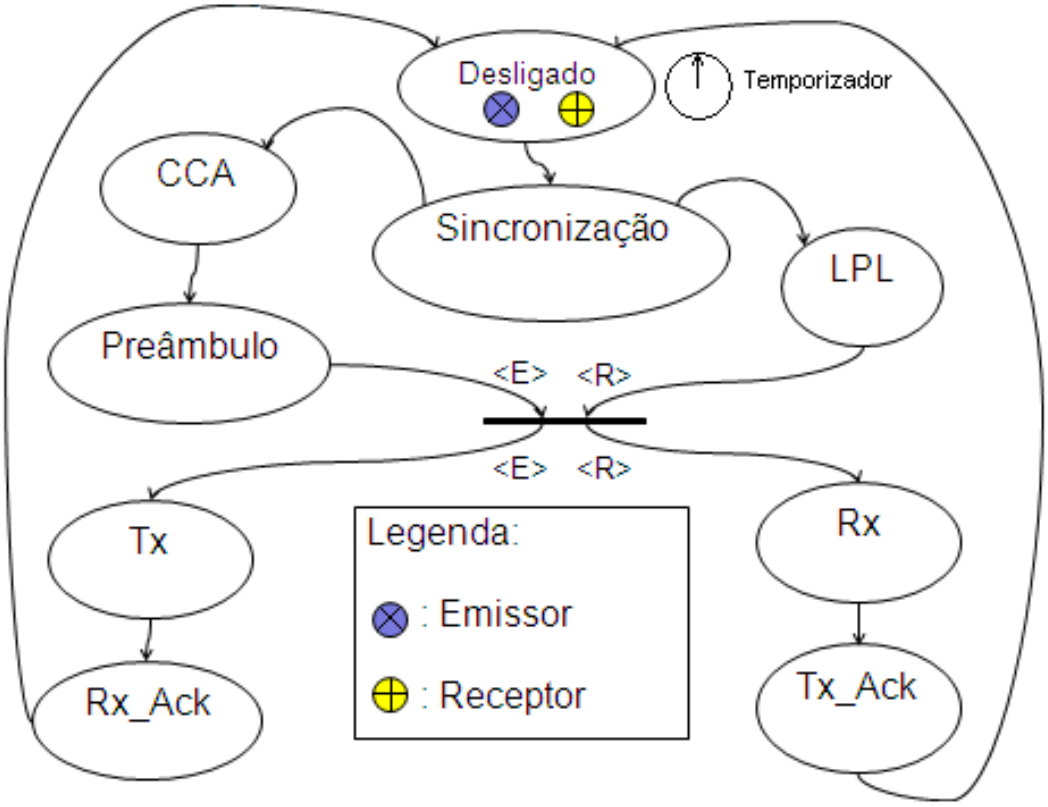

Figura 3.2: Protocolo de comunicação S-MAC.

3.2.2 T-MAC

O T-MAC (*Timeout MAC*) é a evolução do protocolo S-MAC, já que nele foi adicionado a possibilidade de modificar o período ativo em tempo de execução, dessa maneira o protocolo estende seu ciclo ativo em momentos de pico e diminui o ciclo ativo em momentos de menos tráfego. Essa é a solução para o problema da flutuação de tráfego.

3.3 Protocolos de Tempo Dividido

Protocolos de Tempo Dividido, dividem o tempo em slots, onde cada nodo terá seu tempo para transmitir, isso faz com que colisões de dados não ocorram. Esse tipo de protocolo não

favorece a velocidade de transmissão, já que cada nodo tem seu slot especifico, e nem sempre esse slot estará sendo usado. Um exemplo de protocolo de Tempo Dividido é o TDMA.

3.3.1 TDMA

De acordo com a Figura 3.3 o protocolo TDMA (*Time Division Multiple Access*) divide o tempo em slots, e relaciona cada um desses slots a um nodo. Se o nodo deseja transmitir, ele irá aguardar até o seu slot e então transmitirá.

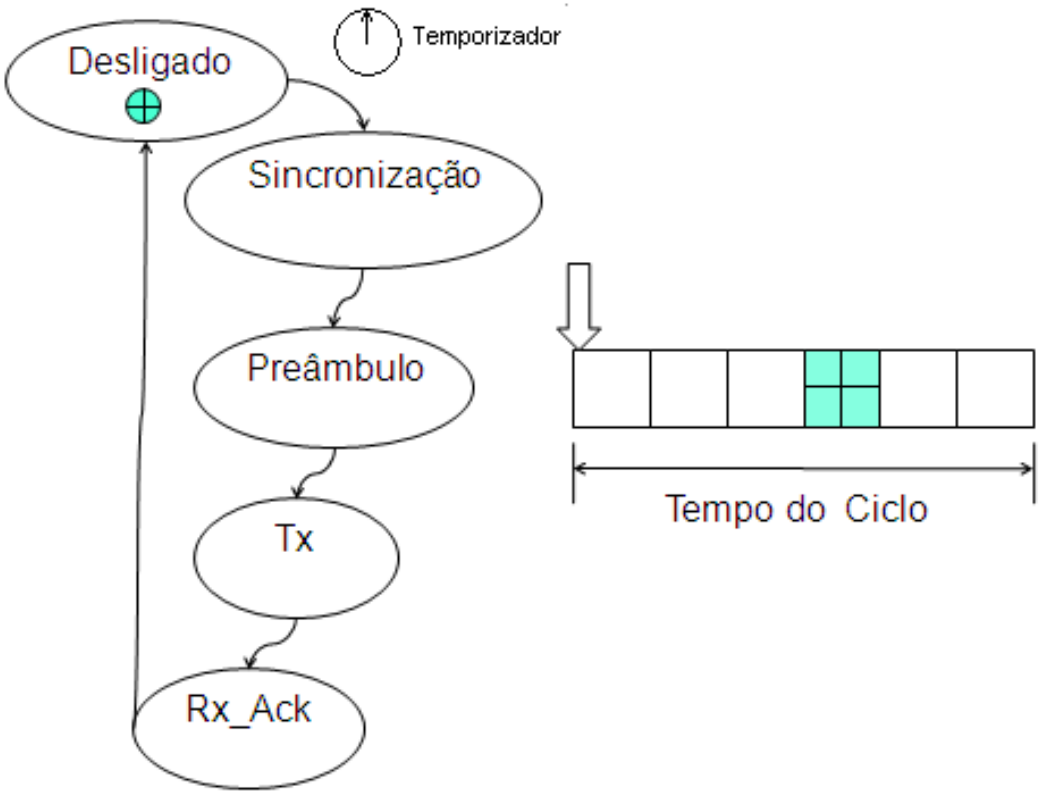

Figura 3.3: Protocolo de comunicação TDMA.

Podemos constatar que os protocolos descritos possuem regiões distintas em suas máquinas de estado como Backoff, necessidade de Ack ou não, configuração do ciclo ativo e sincronização com algum intervalo de tempo (como TDMA). Devido a similaridade entre os protocolos existentes, é possível construir uma máquina de estados configurável de um protocolo, que pode se transformar em qualquer um deles de maneira simples, essa idéia é explicada a seguir.

3.4 Protocolo de comunicação C-MAC

Com a grande variedade de aplicações pare redes de sensores sem fios hoje existentes no mercado, é necessário a existência de uma gama de protocolos de comunicação já que cada um visa atender especificamente um nicho de aplicações. Alguns protocolos são especializados em permitir um grande fluxo de dados, já outros visam a economia de energia ou até mesmo a garantia de entrega de mensagens.

Em 2005 foi desenvolvido [(Wanner, Oliveira e Fröhlich 2007)] um sistema de composição de protocolos leves e apresentaram várias vantagens em se usar meta-programação para configurar, selecionar e combinar protocolos de comunicação. Com esse sistema é possível adaptar um protocolo de acordo com as necessidades da aplicação.

Utilizando as premissas citadas acima, deu-se origem ao C-MAC que é um framework de estratégias de controle de acesso ao meio que é configurado de maneira transparente. As características do C-MAC permitem que sejam configurados parâmetros do protocolo tais como: sistema de contenção (CSMA-CA), necessidade de acknowledgments, tempo de ciclo ativo e envio e recebimento de dados. Tais parâmetros podem ser escolhidos pelo programador através do Trait do protocolo. O Trait é uma classe parametrizada cujos membros são constantes estáticos que descrevem as propriedades do protocolo. Quando uma determinada propriedade é escolhida, a funcionalidade que ela descreve é incluida no protocolo, como exibido na Figura 3.4. A configuração final de um protocolo é feita em tempo de compilação, de forma em que o sobrecusto em se ter várias opções de configuração é eliminado graças a meta-programação estática.

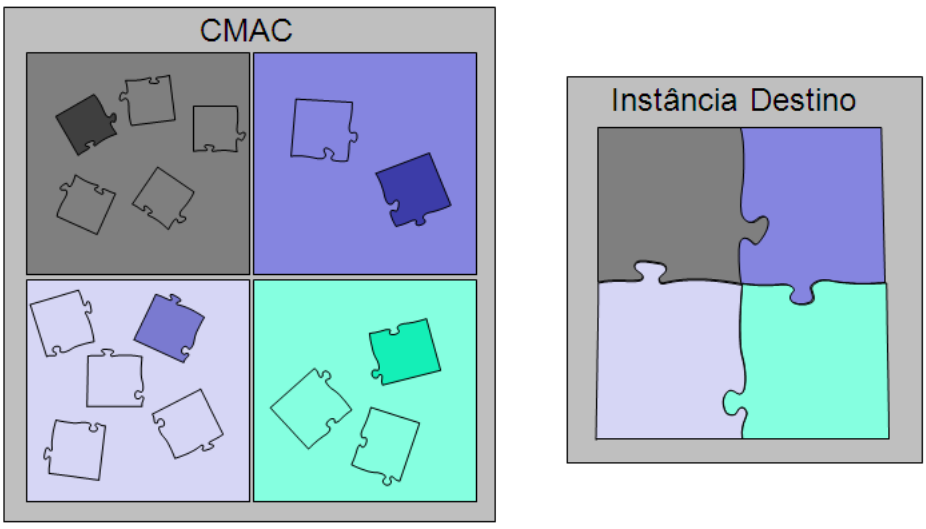

Figura 3.4: Protocolo de comunicação C-MAC.

O C-MAC foi implementado através de uma máquina de estados que contém de maneira generalizada todas as possibilidades de estados presentes em um protocolo de comunicação para uma rede de sensor sem fio (Wanner, Oliveira e Fröhlich 2007). A máquina de estados é controlada através de interrupções de um temporizador dedicado que será melhor descrito no Capítulo 4.

A Figura 3.5 mostra a máquina de estados da primeira versão do CMAC. É possível verificar que no estado Idle ocorre a busca do preâmbulo, e também existe um estado referente a sincronização do receptor com o recebimento de um pacote, tanto a sincronização, quanto o envio e recebimento do preâmbulo nessa versão são responsabilidades do C-MAC, assim como os estados referentes ao Ack e Contenção não tem a possibilidade de serem removidos nessa versão do C-MAC.

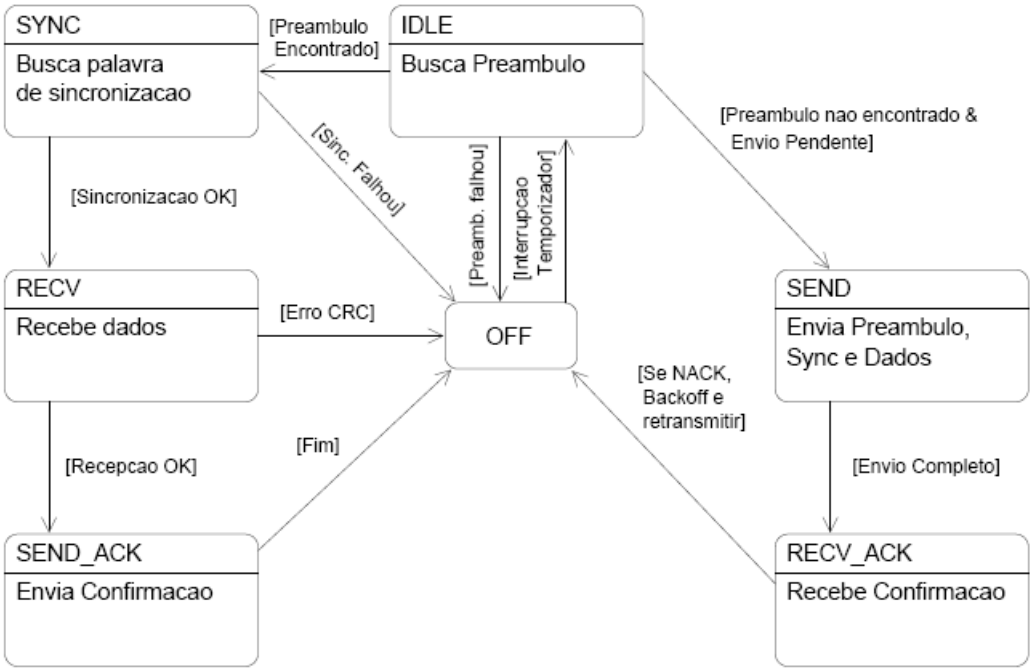

Figura 3.5: Máquina de estados da primeira versão do C-MAC.

4 Implementação

Com o intuito de utilizar a já disponível implementação do C-MAC para o novo hardware, algumas funções foram reimplementadas e outras criadas. As principais diferenças podem ser observadas na máquina de estados e no sistema de interrupções do temporizador. Neste capítulo serão apresentados detalhes do projeto e implementação do protocolo C-MAC para o Epos Mote.

4.1 Máquina de Estados

A nova máquina de estados do C-MAC possui algumas diferenças como podemos observar na Figura 4.1. Foram adicionadas a possibilidade habilitar ou desabilitar pacotes de Ack na comunicação, assim como habilitar ou desabilitar o sistema de contenção feito por CSMA-CA, já descritos nas Sessões 4.1.4 e 4.1.2 respectivamente. Algumas responsabilidades do C-MAC foram removidas pois passaram a ser feitas pelo hardware do rádio AT86RF230.

Com a utilização do rádio AT86RF230 alguns processos que eram feitos antes no C-MAC passaram a ser feitos pelo hardware do próprio rádio, sendo eles a transmissão do preâmbulo, sincronização e processo de CCA.

No envio, o processo de preâmbulo e sincronização são feitos pelo hardware do rádio basicamente enviando uma sequência de 32 zeros binários. Já para o recebimento, quando um dispositivo está ouvindo o canal, ao detectar essa sequência de 32 zeros binários ele fica sincronizado, com isso pode receber o pacote.

O processo de CCA tem a funcionalidade de verificar a disponibilidade ou não do canal no momento da transmissão, ele é feito pelo hardware do AT86RF230 verificando se o nível de sinal do canal está acima ou abaixo do limite especificado, se estiver acima o canal é dado como ocupado, senão é dado como livre. A resposta para o CCA é acessível 140us após o pedido [(ATMEL 2009)].

As subsessões a seguir estão organizadas na ordem cronológica de uma comunicação uti-

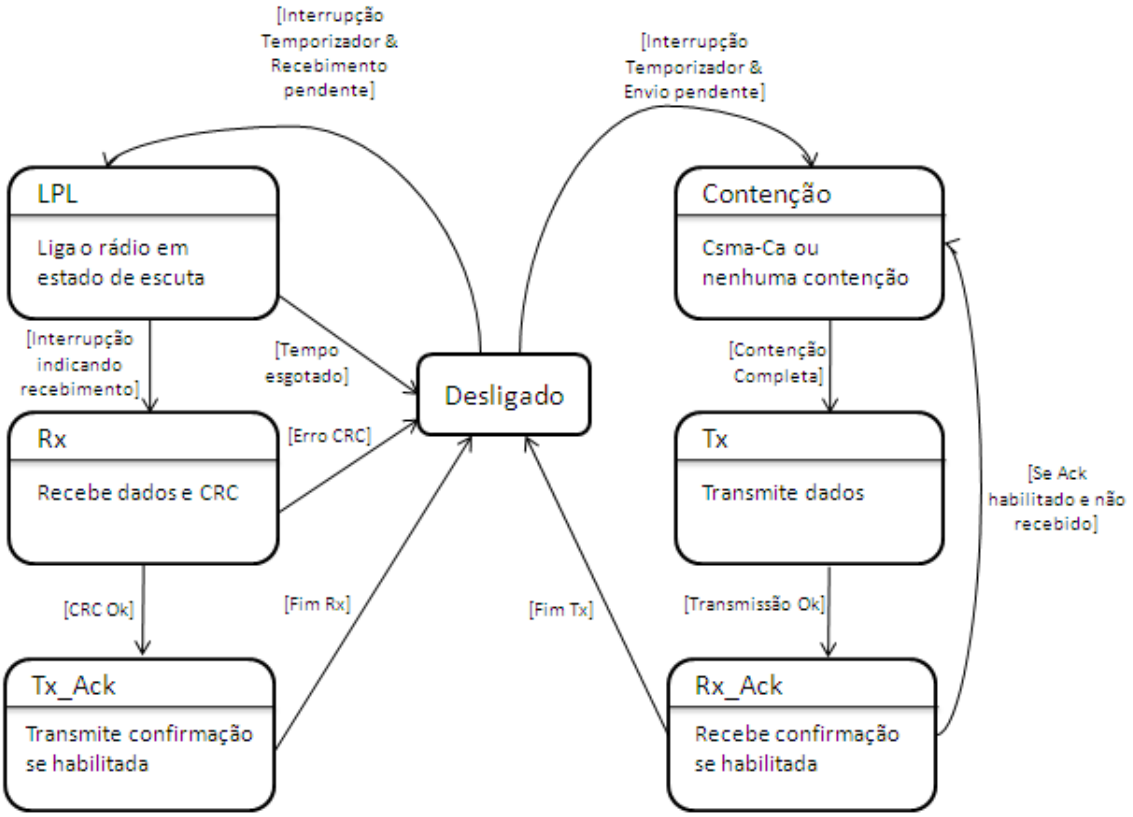

Figura 4.1: Máquina de estados do novo C-MAC.

lizando a máquina de estados do novo C-MAC.

4.1.1 Interrupções do temporizador

A máquina de estados é acionada por um temporizador dedicado que dispara uma interrupção de acordo com o tempo que foi a ele dado. Ao contrário da primeira versão do C-MAC, esse temporizador agora pode ser configurado em tempo de execução. Ao se disparar uma interrupção do temporizador, se houver recebimento e envio pendentes, a decisão de implementação foi priorizar o envio ao invés do recebimento.

Com a configuração em tempo de execução é possível adaptar o ciclo ativo do protocolo, assim como o tempo inativo de acordo com a necessidade de maior ou menor fluxo de dados, como descrito no problema de flutuações no tráfego, ou seja, há a possibilidade do C-MAC assumir características do protocolo T-MAC se implementado um algoritmo de controle do temporizador, sendo que a implementação deste algoritmo foge do escopo do trabalho.

Temporizador de Disparo Único

A possibilidade de se reconfigurar o temporizador em tempo de execução se dá devido a inclusão da implementação do Temporizador de disparo único [(Gracioli et al. 2008)] na árvore do EPOS. O Temporizador de disparo único ao contrário do temporizador periódico não utiliza ticks para contabilizar o tempo. A cada requisição ao temporizador, ele é reconfigurado para que dispare uma interrupção exatamente no momento desejado, com isso é possível obter melhoria na precisão de acordo com (Gracioli et al. 2008).

O temporizador de disparo único exclusivo do C-MAC utilizou o Atmega1281_Timer_3, que é um temporizador de 16bits. A utilização desse temporizador garante maior precisão se comparado ao Atmega1281_Timer_2, que possui apenas 8bits onde foi utilizado como temporizador do Alarme.

4.1.2 Algoritmo CSMA-CA

O CSMA-CA (*Carrier Sense Multiple Access With Collision Avoidance*) é uma regra utilizada para a transmissão no protocolo 802.15.4[(IEEE 2007)], onde o principal objetivo é evitar a colisão de dados, sua máquina de estados pode ser visualizada na Figura 4.2.

O método CSMA-CA implementado foi o não Slotted, e ao utilizá-lo em uma transmissão, ocorre um atraso de acordo com as variáveis especificadas no protocolo e com uma fórmula com um tempo de espera randômico onde o primeiro limite superior de tempo é baixo. No caso do processo de CCA encontrar o canal ocupado, uma nova tentativa será feita e a medida que o número de tentativas de transmissão aumenta o limite superior do tempo randômico de espera também aumenta. A transmissão ocorrerá quando o processo de CCA encontrar o rádio livre, e ela será cancelada se o numero de tentativas exceder o número máximo de tentativas permitidas.

4.1.3 Interrupções de rádio

O rádio AT86RF230 é capaz de fornecer alguns tipos de interrupções para o Atmega1281 no chip ZigBit. A Tabela 4.1 descreve essas interrupções.

A interrupção implementada no C-MAC foi a TRX_END, que é utilizada para indicar o momento em que o rádio termina de receber um pacote de dados. Com isso o C-MAC escuta o rádio no estado de LPL (*Low Power Listening*) sem haver a necessidade de verificar um registrador do rádio de tempos em tempos para determinar a chegada de uma mensagem, ou seja, quando uma mensagem é recebida, o rádio dispara uma interrupção que é tratada dentro

Tabela 4.1: Interrupções do rádio AT86RF230.

Nome IRQ	**Descrição**
IRQ_7: BAT_LOW	Indica que a bateria está abaixo do limite programado.
IRQ_6: TRX_UR	Indica uma violação de acesso no frame buffer.
IRQ_3: TRX_END	RX: Indica a finalização do recebimento de um frame. TX: Indica a finalização da transmissão de um frame.
IRQ_2: RX_START	Indica a detecção de um frame. Com isso o estado interno do rádio é trocado para Busy_RX.
IRQ_1: PLL_UNLOCK	Indica o destravamento do PLL. Implica na impossibilidade de transitir e receber.
IRQ_0: PLL_LOCK	Indica o travamento do PLL. Implica na possibilidade de transmitir e receber.

do C-MAC. As demais interrupções foram implementadas no mediador do rádio e fogem do escopo do trabalho.

No tratador de interrupção do C-MAC, o rádio é desligado voltando para o estado interno TRX_OFF, então o fluxo de processamento volta a máquina de estados de RX do C-MAC e copia os dados recebidos que estão no frame buffer do rádio, depois é feita a verificação da validade dos dados através do CRC (*Cyclic Redundancy Check*).

4.1.4 Pacote de confirmação ACK

Ao contrário da versão antiga do C-MAC, o pacote de confirmação ACK é implementado com estados exclusivos na máquina de estados do novo C-MAC. Com isso é possível habilitar ou não o envio de Acks pelo protocolo modificando uma constante no traits. Ao se desabilitar essa funcionalidade, devido ao uso da meta-programação estática, nenhum sobrecusto relativo a ela é adicionado a instância final do protocolo.

Após o envio de um pacote, o emissor entra em estado de escuta para receber o Ack de confirmação. Ao receber um Ack, esse compara o id do nodo que enviou o pacote Ack com o id que antes era o destinatário do processo de envio, se o valor for igual o Ack recebido é valido e o processo de envio é finalizado.

No caso do receptor, após o recebimento de um pacote, o C-MAC entrará no estado referente ao Ack que obtem o id do nodo que enviou o pacote, e adiciona esse id como destinatário no pacote Ack que então é enviado, assim o processo de recebimento é finalizado.

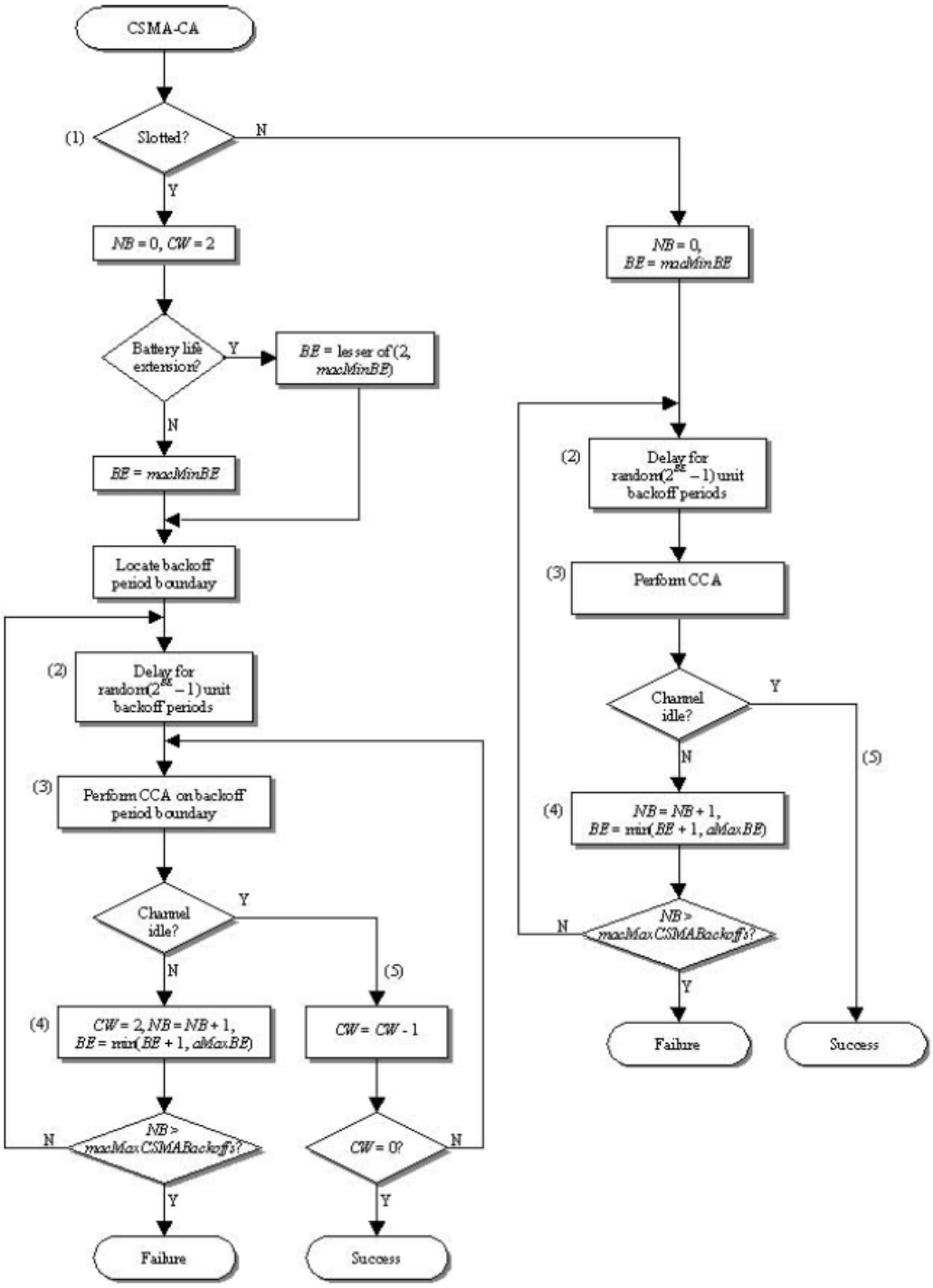

Figura 4.2: Máquina de estados do algoritmo CSMA-CA[(IEEE 2007)].

5 *Resultados*

Com o C-MAC é possível adaptarmos um protocolo de acesso ao meio de acordo com a necessidade de uma aplicação específica. Alguns dos resultados gerados pelas possibilidades de configuração são mostrados nessa sessão, assim como sua máquina de estados que foi gerada após a compilação do protocolo. Em cada uma das possibilidades de configuração destino foram feitos testes a fim de verificar a quantidade de memória de código e dados consumidos pelo (*footprint*) C-MAC e também verificar o desempenho onde é medido o tempo necessário para que a comunicação entre nodos seja concluída.

Nas medidas do tamanho de memória de código e dados do C-MAC foram verificados os valores do tamanho do código, dados e bss em bytes utilizando a ferramenta GNU Size, essas medidas foram comparadas e a diferença entre elas foi explicada. As medidas de desempenho foram feitas utilizando o temporizador dedicado do Cronômetro do EPOS, onde esse foi disparado no início do processo da transmissão e desligado após a conclusão do mesmo. Os valores de tempo obtidos em cada um dos protocolos finais são comparados e explicados. O compilador utilizado para todos os testes foi o GNU GCC para o AVR, versão 4.0.2 com flag de otimização -O2.

5.1 Configurações do Protocolo

Os resultados finais obtidos a partir das configurações possíveis para o protocolo C-MAC são apresentados abaixo. Esses resultados são as possíveis máquinas de estado que o protocolo C-MAC é capaz de gerar habilitando ou desabilitando uma opção de configuração.

5.1.1 CSMA-CA e Ack Habilitados

Ao se habilitar o CSMA-CA e Ack a máquina de estados gerada pode ser observada na Figura 5.1. Neste caso o protocolo com tamanho máximo é gerado, todo o código referente ao CSMA-CA e o Ack é adicionado no protocolo destino, então esse fica com o maior consumo

de memória de código e dados se comparado as outras configurações. Pelo fato desse protocolo destino possuir sistema de contenção e Acks a comunicação se torna mais lenta do que nos nas outras configurações, mas ganha evitando colisões e garantindo a entrega dos pacotes.

Essa configuração pode ser utilizada em ambientes onde há grande número de nodos concorrendo pelo meio e onde a troca de mensagem deve ser garantida já que cada nodo transmitirá poucas vezes a fim de poupar energia.

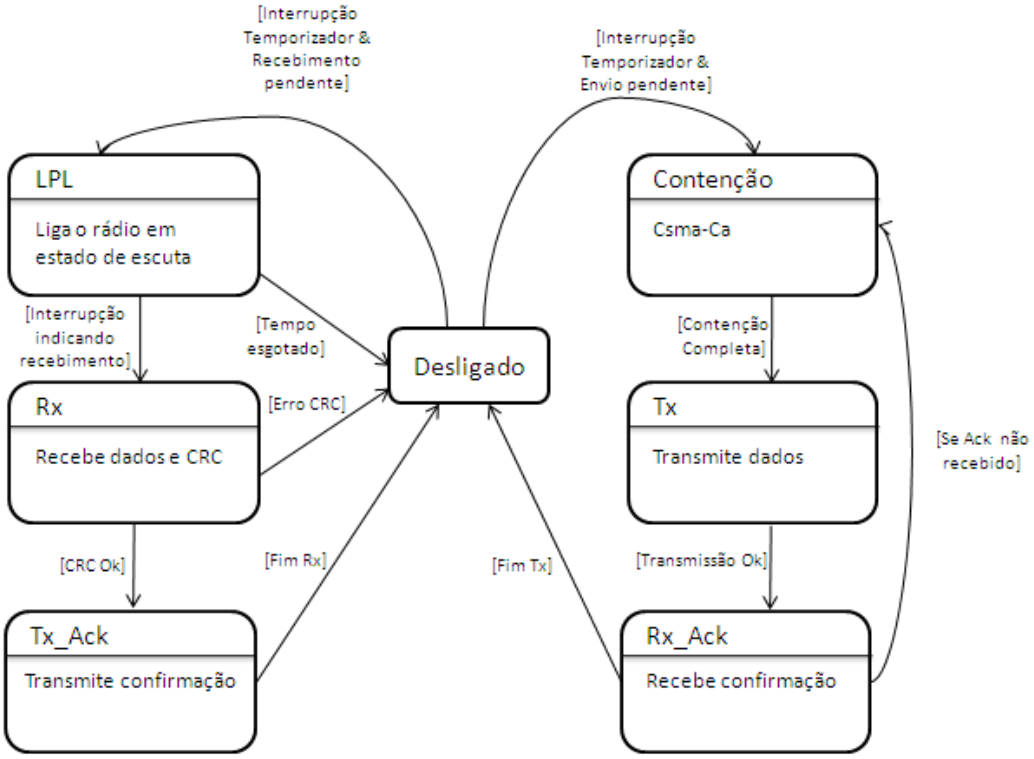

Figura 5.1: Máquina de estados para a configuração com CSMA-CA e Ack habilitados.

5.1.2 CSMA-CA Desabilitado e Ack Habilitado

Ao se desabilitar o sistema de contenção feito pelo CSMA-CA e manter o Ack habilitado foi gerado um protocolo com as características do B-MAC que pode ser visualizado na Figura 5.2. Se esse protocolo destino for comparado ao anterior, observa-se uma diminuição no consumo de memória de código e dados no C-MAC, e também a diminuição no tempo necessário para iniciar a transmissão. Essa configuração pode até ser mais vantajosa em redes de baixo tráfego de pacotes, mas se for utilizada em redes onde existe maior tráfego de dados acontecerão muitas colisões impossibilitando a comunicação.

A configuração utilizando CSMA-CA desabilitado e Ack habilitado é propícia para ambientes onde não exista grande concorrência pelo meio, ou seja, distribuição com baixa densidade

de nodos na região. Onde a aplicação que roda em cada nodo transmitire poucas vezes a fim de minimizar o consumo energético, porém em cada uma dessas transmissões há necessidade de garantia de entrega do pacote.

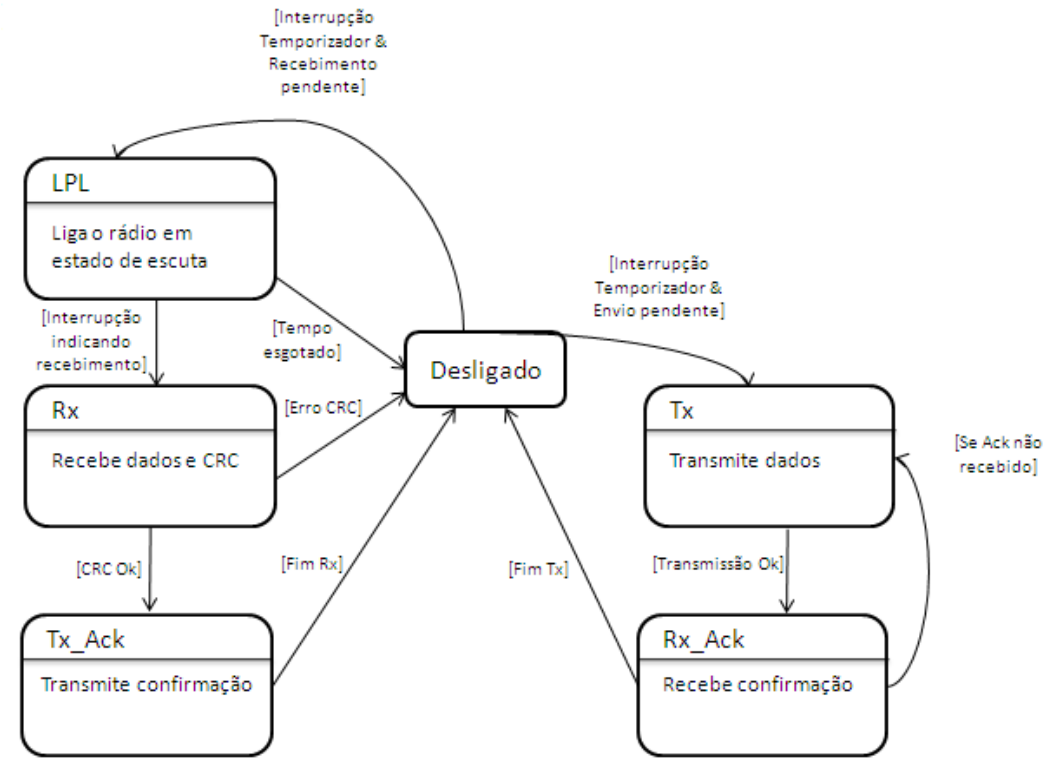

Figura 5.2: Máquina de estados para a configuração com CSMA-CA desabilitado e Ack habilitado.

5.1.3 CSMA-CA Habilitado e Ack Desabilitado

Ao se habilitar o CSMA-CA e desabilitar o Ack, a máquina de estados obtida pode ser visualizada na Figura 5.3. Utilizando esse protocolo destino, o sistema de contenção é melhorado pelo motivo de que o canal estará ocupado durante menos tempo do que se os Acks estivessem habilitados, então será mais fácil obter uma resposta positiva (canal livre) no processo de CCA feito dentro do CSMA-CA. Porém perde-se em confiabilidade na entrega dos pacotes, já que o emissor não recebe a confirmação de que o nodo destino recebeu o pacote.

A configuração utilizando CSMA-CA habilitado e Ack desabilitado é propícia para ambientes onde há grande número de nodos e cada um deles transmite com bastante frequência, não existindo grande importância em se garantir a entrega de cada mensagem individual com o sistema de Ack.

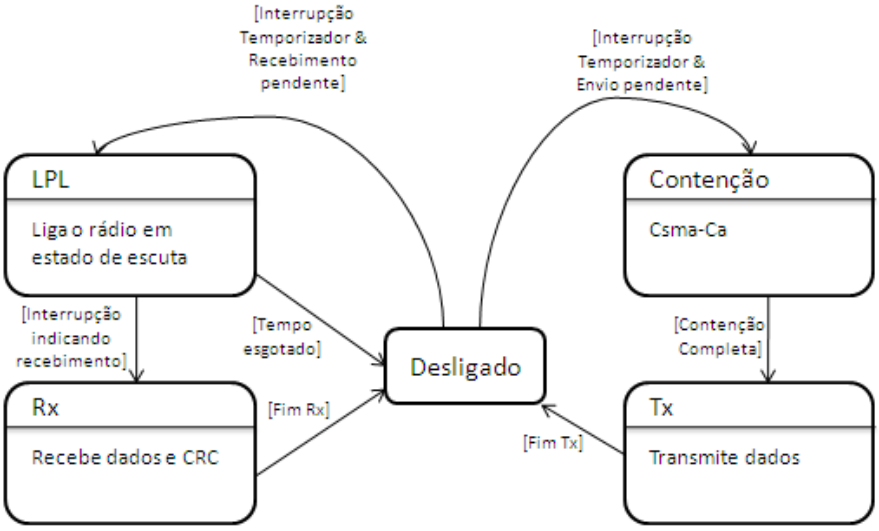

Figura 5.3: Máquina de estados para a configuração com CSMA-CA habilitado e Ack desabilitado.

5.1.4 CSMA-CA e Ack Desabilitados

Ao se desabilitar o CSMA-CA e Ack a máquina de estados obtida pode ser visualizada na Figura 5.4. A máquina de estados obtida é a mais simples que o C-MAC pode gerar. Basicamente a comunicação acontece sem nenhuma garantia de entrega de pacotes, e sem nenhuma organização que visa evitar colisão de pacotes.

A configuração utilizando CSMA-CA e Ack desabilitados é propicia para uma rede onde existem poucos nodos e esses transmitam com grande frequência, não existindo grande importância em se garantir a entrega de cada mensagem individual com o sistema de Ack.

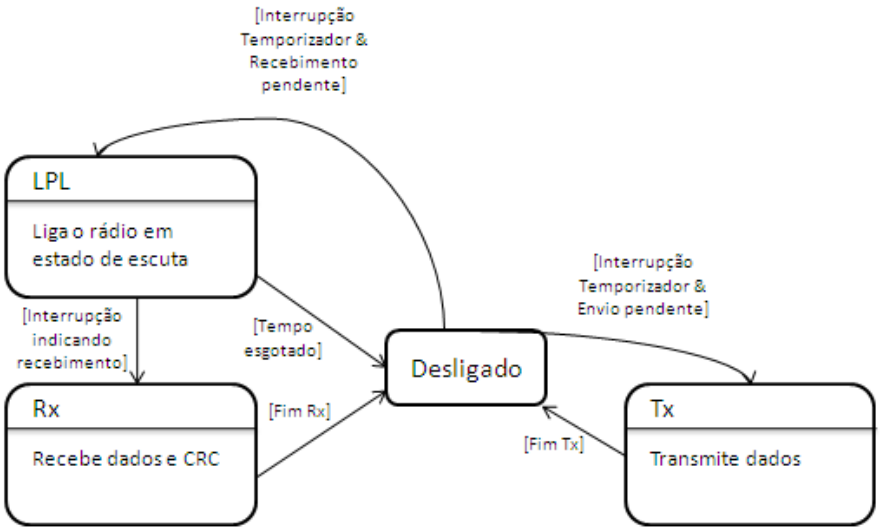

Figura 5.4: Máquina de estados para a configuração com CSMA-CA e Ack desabilitado.

Tabela 5.1: Consumo de memória nas diferentes configurações do protocolo.

CSMA-CA	**ACK**	**Código (bytes)**	**Dados (bytes)**	**Bss (bytes)**
Habilitado	Habilitado	9194	50	282
Habilitado	Desabilitado	9008	50	282
Desabilitado	Habilitado	4078	38	282
Desabilitado	Desabilitado	3892	38	282

5.2 Consumo de Memória

A Tabela 5.1 apresenta o tamanho do código e dados para cada uma das possíveis configurações do protocolo. Os valores foram obtidos utilizando a ferramenta GNU Size para AVR (*avr-size*) versão 2.19.

Os valores apresentados na tabela 5.1 são a soma de todos os arquivos referentes a comunicação de dados, sendo eles o Cmac, Cmac_init e o mediador de rádio AT86RF230.

Pode-se notar que ao habilitar o CSMA-CA houve um aumento no tamanho do código e dados devido a utilização de bibliotecas necessárias para os cálculos (e.g números randômicos e exponenciação) e a necessidade de mais atributos para controle dessa funcionalidade, respectivamente. Pode-se observar também que o valor bss não variou, pelo fato de que nenhuma das configurações possui variáveis não inicializadas. Com a habilitação do Ack não houve grande diferença no tamanho do código e dados, já que o envio ou recebimento de Ack é uma operação relativamente simples (em tamanho de código).

5.3 Tempo de comunicação

O teste de tempo de comunicação foi feito a fim de verificar o tempo necessário para que o processo de comunicação fosse finalizado, para envio em todas as opções de configuração.

Os valores referentes ao tempo de comunicação são exibidos na Tabela 5.2. Esses valores foram medidos utilizando o Cronômetro do EPOS, de maneira que a medição se iniciava no momento em que foi requisitado o envio, e terminava no momento em que o processo de envio acabasse.

Ao se utilizar um Temporizador para disparar o processo de envio conforme descrito na sessão 4.1.1, o tempo medido pelo cronômetro inclui o tempo em que o temporizador manteve o protocolo inativo, tendo em vista que esse valor não é interessante para as medições, ele foi subtraido para cada um dos casos.

Tabela 5.2: Tempo necessário para a comunicação nos protocolos destino.

CSMA-CA	**ACK**	**Tempo (ms)**
Habilitado	Habilitado	240
Habilitado	Desabilitado	133
Desabilitado	Habilitado	150
Desabilitado	Desabilitado	75

Ao enviarmos um pacote com a configuração utilizando Ack habilitado, o tempo de recebimento é adicionado no tempo total, já que o nodo receptor receberá o pacote, e enviará um Ack confirmando o recebimento, ou seja, essa medição é referente a dois envios e dois recebimentos.

Como esperado, o valor de tempo necessário para que a comunicação se finalizasse nas configurações onde o Ack está habilitado é muito superior as outras configurações, isso ocorre devido ao fato de que essa medição mede todo o tempo necessário para o envio por parte do nodo emissor, o recebimento por parte do nodo receptor, o envio do ack por parte do nodo receptor e o recebimento do Ack pelo nodo emissor.

No caso do processo de CSMA-CA habilitado pode-se verificar que o tempo necessário para a comunicação ocorrer foi maior se comparado ao CSMA-CA desabilitado. Isso ocorre devido ao fato de que o CSMA-CA gera um atraso para iniciar a transmissão (*Backoff*). No caso de existir outros nodos concorrendo pelo canal, o tempo necessário para executar a comunicação seria ainda maior, já que esse atraso é aumentado a cada tentativa de transmissão.

6 *Conclusão e Trabalhos Futuros*

Este trabalho apresentou um estudo sobre os principais protocolos para redes de sensores sem fios, assim como um estudo da primeira versão do C-MAC, com isso foi feita a implementação do protocolo C-MAC para uma nova plataforma de hardware. Com a implementação da nova versão do C-MAC foi possível verificar que de fato um protocolo configurável de maneira simples é possível e pode gerar bons resultados.

A principal contribuição deste trabalho foi a identificação de pontos de configuração para protocolos em redes de sensores sem fios, analisando tanto a implementação do antigo C-MAC como os principais protocolos em redes de sensores sem fios e implementando um protocolo facilmente configurável de acordo com as necessidades de cada aplicação. O desenvolvimento do C-MAC incluiu novas possibilidades de configuração, gerando assim novas máquinas de estado de comunicação onde cada uma delas pode ser utilizada em situações específicas, melhorando assim o desempenho de uma rede de sensores sem fios. Testes do C-MAC apresentaram as possíveis máquinas de estado de comunicação assim como a memória de código e dados utilizados pelo sistema e testes de desempenho onde são mostrados os tempos necessários para finalizar um processo de comunicação.

As próximas etapas de desenvolvimento do CMAC deverão permitir ainda mais possibilidades de configuração, principalmente em se criar um estado específico para que o protocolo possa se sincronizar com a rede de maneira a saber exatamente o momento de acordar, ouvir e transmitir no canal, essa possibilidade traria grande economia de energia e evitaria completamente colisões se as características do TDMA fossem incorporadas no C-MAC.

Apesar do sistema do Temporizador de Disparo Único ter sido incorporado no C-MAC, nenhum algoritmo específico para que o tempo do ciclo ativo se adaptasse as flutuações do trafego foi implementado. Com a inclusão de um algoritmo que controla o período ativo, o C-MAC poderá incorporar características do protocolo T-MAC efetivamente.

Ainda inúmeras possibilidades no rádio AT86RF230 se mantém inexploradas no atual trabalho, por isso outra possibilidade de trabalho futuro seria implementar todos os recursos que o

rádio oferece nativamente no mediador de rádio e integra-las ao C-MAC, com isso pode-se obter uma menor carga no microprocessador já que essas operações poderão ser feitas diretamente no hardware do rádio.

Referências Bibliográficas

ATMEL 2009 ATMEL. At86rf230 datasheet hardware reference. In: . [S.l.: s.n.], 2009.

Barr John C. Bicket e Sirer. BARR JOHN C. BICKET, D. S. D. B. D. T. D. K. B. Z. R.; SIRER., E. G. On the need for system-level support for ad hoc and sensor networks. In: . [S.l.: s.n.].

Buettner et al. 2006 BUETTNER, M. et al. X-mac: A short preamble mac protocol for duty-cycledwireless networks. In: . Boulder, CO: [s.n.], 2006. p. 307–320. Disponível em: <http://dx.doi.org/10.1145/1182807.1182838>.

Crossbow CROSSBOW. Mpr/mib mote hardware user's manual. In: . [S.l.: s.n.].

Dam e Langendoen 2003 DAM, T. v.; LANGENDOEN, K. An adaptive energy-efficient MAC protocol for wireless sensor networks. In: . Los Angeles, CA: [s.n.], 2003. p. 171–180. Disponível em: <http://dx.doi.org/10.1145/958491.958512>.

ETH 2005 ETH. Btnode rev3 hardware reference. In: . [S.l.: s.n.], 2005.

Fröhlich 2001 FRöHLICH, A. A. Application-Oriented Operating Systems. *GMD - Forschungszentrum Informationstechnik, Sankt Augustin*, 2001.

Gracioli et al. 2008 GRACIOLI, G. et al. One-shot time management analysis in epos. In: *SCCC*. [S.l.: s.n.], 2008. p. 92–99.

IEEE 2007 IEEE. Ieee standard 802.15.4a-2007. In: . [S.l.: s.n.], 2007.

Klues et al. 2008 KLUES, K. et al. A Component-Based Architecture for Power-Efficient Media Access Control in Wireless Sensor Networks. *ACM*, 2008.

Langendoen e Halkes LANGENDOEN, K.; HALKES, G. Embedded systems handbook. In: . [S.l.: s.n.].

Moteiv. 2005 MOTEIV. Tmote sky datasheet. In: . [S.l.: s.n.], 2005.

Polastre, Hill e Culler 2004 POLASTRE, J.; HILL, J.; CULLER, D. Versatile Low Power Media Access for Wireless Sensor Networks . *ACM*, 2004.

Polpeta et al. 2004 POLPETA, F. V. et al. Hardware Mediators: a Portability Artifact for Component-Based Systems. *In International Conference on Embedded and Ubiquitous Computing*, 2004.

Polpeta et al. 2004 POLPETA, F. V. et al. Portabilidade em Sistemas Operacionais Baseados em Componentes de Software. *In First Brazilian Workshop on Operating System*, 2004.

Pottie e Kaiser POTTIE, G.; KAISER, W. Wireless integrated sensor network. In: . [S.l.: s.n.].

Wanner 2006 WANNER, L. F. Um Ambiente de Suporte a Execução de Alicações em Redes de Sensores sem Fios. In: . [S.l.: s.n.], 2006. p. 7–8.

Wanner, Oliveira e Fröhlich 2007 WANNER, L. F.; OLIVEIRA, A. B. de; FRöHLICH, A. A. Configurable Medium Access Control for Wireless Sensor Networks. In: *International Embedded System Symposium*. [S.l.: s.n.], 2007. p. 401–410.

Ye, Heidemann e Estrin 2002 YE, W.; HEIDEMANN, J.; ESTRIN, D. An energy-efficient mac protocol for wireless sensor networks. 2002. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.8362>.

APÊNDICE A – Código Fonte

```
#ifndef __cmac_h
#define __cmac_h

#include "../common/at86rf230/at86rf230.h"
#include <nic.h>
#include <cpu.h>
#include <utility/crc.h>

__BEGIN_SYS

class CMAC: public Low_Power_Radio {

private:

    enum CMAC_STATE { IDLE, RX, TX };

    enum TX_STATE { TX_PENDING, TX_BACKOFF, TX_CSMA_CA, TX_DATA, TX_HANDLE, TX_RECE

    enum RX_STATE { RX_IDLE, RX_DATA, RX_HANDLE, RX_SEND_ACK, RX_DONE };

    static const int FREQUENCY = Traits<CMAC>::FREQUENCY;
    static const int POWER = Traits<CMAC>::POWER;

    static const unsigned int PREAMBLE_LENGTH = 80;
    static const unsigned int PREAMBLE_KEY = 0XCC33;
    static const unsigned int MAX_RX_SYNC_TRIES = 5;
    static const unsigned int MAX_FRAME_FULL_SIZE = 127;
```

```
public:

    CMAC(int unit = 0) {
_addr = Traits<CMAC>::ID;
    }

    int send(const Address & dst, const Protocol & prot,
     const void *data, unsigned int size);

    int receive(Address * src, Protocol * prot,
void * data, unsigned int size);

    const Address & address() {
return _addr;
    }

    const Statistics & statistics() {
return (Statistics&)_stats;
    }

    void reset() {
    }

    void config(int frequency, int power){
    }

    unsigned int mtu() const { return MTU; }

    static void init(unsigned int n);

    static void interruptHandler(unsigned int);

    static void timer_handler(unsigned int unit);
```

```
private:



static int _send(const Address & dst, const Protocol & prot,
     const void *data, unsigned int size, const char type);

    static void tx_state_machine();
    static bool tx_backoff();
    static bool tx_csma_ca();
    static bool tx_data();
    static void tx_handle();
    static bool tx_receive_ack();

    static int _receive(Address * src, Protocol * prot,
     void * data, unsigned int size, const char type);

    static void rx_state_machine();
    static bool rx_preamble();
    static bool rx_sync();
    static bool rx_data();
    static int  rx_handle();
    static void rx_giveup();
    static bool rx_send_ack();

private:


static Chronometer chrono;
static Chronometer chrono2;
static bool chrono2FirstTime;

    static AT86RF230 radio;

    static unsigned char buffer[MAX_FRAME_FULL_SIZE];
```

```
    static Semaphore interrupt_semaphore;
    static Semaphore receive_semaphore;
    static Semaphore send_semaphore;


  //  static CC1000                  _cc1000;
    static Address                 _addr;
    static volatile Statistics     _stats;
    static bool _dataReceived;
    static bool isReceiving;
    static bool isTransmiting;

    static volatile CMAC_STATE     _state;

    static volatile TX_STATE       _tx_state;
    static volatile Frame          _tx_frame;
    static volatile unsigned char * _tx_frame_ptr;
    static volatile int            _tx_frame_size;
    static volatile bool           _tx_available;
    static volatile unsigned char  _tx_data_count;

    static volatile RX_STATE       _rx_state;
    static volatile Frame          _rx_frame;
    static volatile unsigned char * _rx_frame_ptr;
    static int             _rx_frame_size;
    static volatile bool           _rx_available;
    static volatile unsigned char  _rx_preamble_count;
    static volatile unsigned char  _rx_sync_tries;
    static volatile unsigned char  _rx_sync_count;
    static volatile unsigned char  _rx_bit_offset;
    static volatile unsigned char  _rx_data_byte;
    static volatile union data_word {
struct {
    unsigned char lsb;
    unsigned char msb;
```

```
};
unsigned int word;
    }  _rx_data_word;
    static volatile unsigned char   _rx_data_count;
    static volatile unsigned char   _rx_tries;

    //Variáveis relacionadas ao CSMA-CA
    static int aUnitBackoffPeriod;
    static int NB;
    static int BE;
    static int macMinBE;
    static int aMaxBE;
    static int macMaxCSMABackoffs;//FIXME confirm this value
    static unsigned char macDSN;//This is inittiated with a random value

    static ATMega1281_Timer_3       _timer;

};

__END_SYS
#endif



// EPOS-- CMAC Implementation
#include <mach/common/at86rf230/at86rf230.h>
#include <machine.h>
#include <nic.h>
#include <mach/avr_common/cmac.h>
#include <utility/ostream.h>
#include <display.h>
#include <alarm.h>
#include <semaphore.h>
#include <math.h> //Csma-CA
#include <stdlib.h> //Csma-CA
```

```
#include <chronometer.h>

__BEGIN_SYS

OStream print;

int CMAC::send(const Address & dst, const Protocol & prot,
const void *data, unsigned int size) {
return _send(dst, prot, data, size, '0');
}

int CMAC::_send(const Address & dst, const Protocol & prot,
const void *data, unsigned int size, const char type) {
if(_tx_available) {
chrono.start();
_state = TX;
_tx_available = false;
Frame * _frame = new((void *)&_tx_frame)
Frame(dst,_addr,prot,data,size, 0, type);
_tx_frame_ptr= (unsigned char *) _frame;
_tx_data_count = 0;
_tx_frame_size = size + 6;//Alinha os bytes
send_semaphore.p();
return size;
}
return 0;
}

int CMAC::receive(Address * src, Protocol * prot,
void * data, unsigned int size) {
return _receive(src, prot, data, size, '0');
}

int CMAC::_receive(Address * src, Protocol * prot,
void * data, unsigned int size, const char type) {
```

```
if(!_rx_available) {
return 0;
}
chrono.start();
_rx_available = false;
receive_semaphore.p();
return size;

}

/* -------------- TX -------------- */

void CMAC::tx_state_machine() {
switch (_tx_state) {
case TX_BACKOFF:
if (Traits<CMAC>::STATE_BACKOFF_ON) {
for(unsigned int i = 0; i < 0xff; i++);
_tx_state = TX_DATA;
} else {
_tx_state = TX_CSMA_CA;
}
tx_state_machine();
break;

case TX_CSMA_CA:
if (Traits<CMAC>::CSMA_CA_ON) {
if (tx_csma_ca()) {
_tx_state = TX_DATA;
} else {
//print << "-CSMA-CA-FALHOU-";
_tx_state = TX_DONE;
}
} else {
_tx_state = TX_DATA;
}
```

```
tx_state_machine();
break;

case TX_DATA:
if(tx_data()) {
_tx_state = TX_HANDLE;
}

case TX_HANDLE:
tx_handle();
_tx_state = TX_RECEIVE_ACK;
tx_state_machine();
break;

case TX_RECEIVE_ACK:
if (tx_receive_ack()) {
_tx_state = TX_DONE;
}
break;
}

}

bool CMAC::tx_backoff() {
}

bool CMAC::tx_csma_ca() {
if (Traits<CMAC>::CSMA_CA_ON) {
print << "CMAC::CSMACA\n";
NB = 0;
BE = macMinBE;
while (NB < macMaxCSMABackoffs) {
srand(_addr);//Uma semente para cada nodo
int delay = static_cast<int> (fmod(rand(), (pow(2, BE) - 1))
* aUnitBackoffPeriod);
```

```
delay = (delay < 1000) ? 1000 : delay;
print << "delay: " << delay << "\n";
Alarm::delay(delay);//delay(random(2^BE -1)*Period)
//Clear Channel Assesment(CCA)
bool aux = false;
//print << "CCA:" << radio.CCA_measurement(aux);
if (aux) {
//print << "CCA-OK\n";
return true;
}
NB = NB + 1;
BE = BE + 1;
if (BE > aMaxBE)
BE = aMaxBE;
}
}
return false;
}

bool CMAC::tx_data() {
memcpy( (void*)&buffer, &_tx_frame_ptr+0x02, _tx_frame_size);//Essa soma arruma um
unsigned short crc =
CRC::crc16(reinterpret_cast<char*> (buffer), _tx_frame_size);
unsigned char *framePtr2 = reinterpret_cast<unsigned char*> (&crc);
_tx_frame_size = _tx_frame_size + 2;
buffer[_tx_frame_size - 2] = framePtr2[0];
buffer[_tx_frame_size - 1] = framePtr2[1];
print << "CMAC::tx_data(5) \n";
if (_tx_frame_size < CMAC::MAX_FRAME_FULL_SIZE) {
radio.send(buffer, _tx_frame_size);
radio.forceValidState();
//print << "send -> Enviado. \n";
return true;
}
return false;
```

```
}

void CMAC::tx_handle() {
_stats.tx_bytes += sizeof(Frame);
_stats.tx_packets++;
_tx_available = true;
}

bool CMAC::tx_receive_ack() {
NIC::Address src, dst;
NIC::Protocol prot;
char data[1];
if (Traits<CMAC>::ACK_ON & _tx_frame._rss != 'A') {
_rx_frame_ptr= (unsigned char *) &_rx_frame;
_rx_data_count = 0;
_state = RX;
_rx_state = RX_DATA;
radio.rx_on();
AT86RF230::result_t result;
while (!radio.dataReceived()) {}//Não usa interrupt
radio.rx_off();
result = radio.receive(buffer, _rx_frame_size);
memcpy( (void*)&_rx_frame, buffer, _rx_frame_size);
_tx_state = TX_DONE;
} else {
//print << "Ack Desabilitado";
_tx_state = TX_DONE;
}
chrono.stop();
print << "Measured time: " << chrono.read() / 1000 << "ms\n";
isTransmiting = false;
send_semaphore.v();
return false;
}
```

```
/* -------------- RX -------------- */

void CMAC::rx_state_machine() {
switch (_rx_state) {
case RX_DATA:
if(rx_data()) {
_rx_state = RX_HANDLE;
rx_state_machine();
}
break;

case RX_HANDLE:
if(rx_handle()) {
_rx_state = RX_SEND_ACK;
rx_state_machine();
}
break;

case RX_SEND_ACK:
if(rx_send_ack()) {
_rx_state = RX_DONE;
}
break;
}

}

bool CMAC::rx_data() {
AT86RF230::result_t result;
radio.rx_off();
result = radio.receive(buffer, _rx_frame_size);
memcpy( (void*)&_rx_frame, buffer, _rx_frame_size);
if (result == AT86RF230::SUCESS) {
_rx_available = true;
//print << "Recebido\n";
```

```
return true;
} else {
//print << "Não Recebido\n";
return false;
}
}

int CMAC::rx_handle() {
unsigned short crc = CRC::crc16((reinterpret_cast<char*> (buffer)),
_rx_frame_size - 2);
unsigned char *framePtr = reinterpret_cast<unsigned char*> (&crc);
if (buffer[_rx_frame_size - 2] == framePtr[0]
&& buffer[_rx_frame_size - 1] == framePtr[1]) {
//print << "CRC Ok\n";
_stats.rx_packets++;
_stats.rx_bytes += _rx_frame_size;
return true;
} else {
rx_giveup();
_stats.dropped_packets++;
return false;
}
}

bool CMAC::rx_send_ack() {
if (Traits<CMAC>::ACK_ON & _rx_frame._rss != 'A') {
//print << "Enviando Ack\n";
Frame * _frame = new((void *)&_tx_frame)
Frame(static_cast<const unsigned char>(_rx_frame._src),_addr,static_cast<const unsi
_tx_frame_ptr= (unsigned char *) _frame;
_tx_data_count = 0;
_tx_frame_size = 6;
memcpy( (void*)&buffer, &_tx_frame_ptr+0x02, _tx_frame_size);//A soma arruma um pro
unsigned short crc =
CRC::crc16(reinterpret_cast<char*> (buffer), _tx_frame_size);
```

```
unsigned char *framePtr2 = reinterpret_cast<unsigned char*> (&crc);
_tx_frame_size = _tx_frame_size + 2;
buffer[_tx_frame_size - 2] = framePtr2[0];
buffer[_tx_frame_size - 1] = framePtr2[1];
radio.send(buffer, _tx_frame_size);
radio.forceValidState();
//print << "Ack Enviado\n";
_rx_state = RX_DONE;
} else {
//print << "Ack Desabilitado";
_rx_state = RX_DONE;
}
chrono.stop();
print << "Measured time: " << chrono.read() / 1000 << "ms\n";
return false;

}

void CMAC::interruptHandler(unsigned int) {
unsigned char val = AT86RF230_HAL::readRegister(AT86RF230_HAL::IRQ_STATUS_REG);
print << "CMAC::interruptHandler:"<< val <<"-\n";
AT86RF230_HAL::writeRegister(AT86RF230_HAL::IRQ_STATUS_REG, val);

if (_state == RX) {
if (val & 0x08) {//IRQ_STATUS_TRX_END
print << "CMAC::interruptHandler: RX_END_INTERRUPT -\n";
AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_STATE_REG, 0x08 );//TRX_CMD_TRX_OFF
CMAC::_dataReceived = true;
isReceiving = false;
_rx_state=RX_DATA;
rx_state_machine();
receive_semaphore.v();
}
}
}
```

```
void CMAC::rx_giveup() {
static volatile bool _dataReceived = false;
_rx_state = RX_DATA;
}

/* ----------- Handlers ----------- */

void CMAC::timer_handler(unsigned int unit) {
//Disparado depois do sleep
_timer.reset();

if (!chrono2FirstTime) {
chrono2.start();
chrono2FirstTime = true;
} else {
chrono2.stop();
print << "Measured time2: " << chrono2.read() / 1000 << "ms\n";
}

if(!_tx_available) {
if (!isTransmiting) {
_timer.disable();
isTransmiting = true;
_state = TX;
if (Traits<CMAC>::CSMA_CA_ON) {
_tx_state = TX_CSMA_CA;
} else {
_tx_state = TX_BACKOFF;
}
tx_state_machine();
_timer.enable();
}
} else if (!_rx_available) {
if (!isReceiving) {
```

```
_timer.disable();
isReceiving = true;
_rx_frame_ptr= (unsigned char *) &_rx_frame;
_rx_data_count = 0;
_state = RX;
_rx_state = RX_DATA;
radio.rx_on();
_timer.enable();
}
}
}

//Cronometros apenas necessários para as medições de desempenho
Chronometer CMAC::chrono;
Chronometer CMAC::chrono2;
bool CMAC::chrono2FirstTime = false;

AT86RF230 CMAC::radio;
unsigned char CMAC::buffer[CMAC::MAX_FRAME_FULL_SIZE];
bool CMAC::_dataReceived = false;
bool CMAC::isReceiving = false;
bool CMAC::isTransmiting = false;
Semaphore CMAC::interrupt_semaphore(0);
Semaphore CMAC::receive_semaphore(0);
Semaphore CMAC::send_semaphore(0);

CMAC::Address CMAC::_addr;
volatile CMAC::Statistics CMAC::_stats;

volatile CMAC::CMAC_STATE CMAC::_state;

volatile CMAC::TX_STATE CMAC::_tx_state;
volatile CMAC::Frame CMAC::_tx_frame;
volatile int CMAC::_tx_frame_size;
volatile unsigned char * CMAC::_tx_frame_ptr;
```

```
volatile bool CMAC::_tx_available = true;
volatile unsigned char CMAC::_tx_data_count = 0;

volatile CMAC::RX_STATE CMAC::_rx_state;
volatile CMAC::Frame CMAC::_rx_frame;
int CMAC::_rx_frame_size;
volatile unsigned char * CMAC::_rx_frame_ptr;
volatile bool CMAC::_rx_available = true;//default era false
volatile unsigned char CMAC::_rx_preamble_count = 0;
volatile unsigned char CMAC::_rx_sync_count = 0;
volatile unsigned char CMAC::_rx_sync_tries = 0;
volatile unsigned char CMAC::_rx_bit_offset;
volatile unsigned char CMAC::_rx_data_byte;
volatile CMAC::data_word CMAC::_rx_data_word;
volatile unsigned char CMAC::_rx_data_count = 0;
volatile unsigned char CMAC::_rx_tries = 0;

//Variáveis relacionadas ao CSMA-CA
int CMAC::aUnitBackoffPeriod = 1;//20000
int CMAC::NB = 0;
int CMAC::BE = 0;
int CMAC::macMinBE = 3;
int CMAC::aMaxBE = 5;
int CMAC::macMaxCSMABackoffs = 8;//FIXME confirm this value
unsigned char CMAC::macDSN;//This is inittiated with a random value


ATMega1281_Timer_3 CMAC::_timer;

__END_SYS



// EPOS-- ATMega1281 Timer Mediator Declarations
```

```
#ifndef __atmega1281_timer_h
#define __atmega1281_timer_h

#include "../avr_common/timer.h"
#include <rtc.h>
#include <ic.h>
#include <display.h>

__BEGIN_SYS

class ATMega1281_Timer: public Timer_Common, private AVR_Timer
{
private:
    static const unsigned int CLOCK = Traits<Machine>::CLOCK >> 10;

public:
    // Register Settings
    enum {
     TIMER_PRESCALE_1    = CSn0,
TIMER_PRESCALE_8    = CSn1,
TIMER_PRESCALE_32   = CSn1 | CSn0,
TIMER_PRESCALE_64   = CSn2,
TIMER_PRESCALE_128  = CSn2 | CSn0,
TIMER_PRESCALE_256  = CSn2 | CSn1,
TIMER_PRESCALE_1024 = CSn2 | CSn1 | CSn0
    };

public:
    ATMega1281_Timer() {}

    ATMega1281_Timer(const Hertz & f) {
db<PC_Timer>(TRC) << "ATMega1281_Timer(f=" << f << ")\n";
frequency(f);
    }
```

```
    Hertz frequency() const { return count2freq(ocr2a()); }
    void frequency(const Hertz & f) {
ocr2a(freq2count(f));
tccr2a(WGM21);
tccr2b(TIMER_PRESCALE_1024);
    };

    void reset() { tcnt2(0); }

    void enable(){ timsk2(timsk2() | OCIE2A); }
    void disable(){ timsk2(timsk2() & ~OCIE2A); }

    Tick read() { return tcnt2(); }

protected:
    static Hertz count2freq(const Count & c) { return CLOCK / c; }
    static Count freq2count(const Hertz & f) { return CLOCK / f; }
};


class ATMega1281_Timer_2: public Timer_Common, private AVR_Timer
{
public:

    static const Hertz MACHINE_CLOCK = Traits<Machine>::CLOCK;

    // Traits
    static const Hertz CLOCK = Traits<Machine>::CLOCK >> 8;
    static const RTC::Microsecond MAX_INTERVAL = 500000;
    static const Count MIN_COUNT = (Count)(MAX_INTERVAL / (unsigned long)0xFFFF) +

    typedef RTC::Microsecond Microsecond;
    typedef unsigned int Ticks;

public:
```

```
ATMega1281_Timer_2() {
clock(CLOCK);
    }

    ATMega1281_Timer_2(const Hertz & f) {
clock(CLOCK);
frequency(f);
    }

    const void clock(const Hertz clock) {
tccr0b(TIMER_PRESCALE_1024|WGMn2);
    }

    const int irq() {//TODO CONFERIR
     if(software_interrupt())
     return IC::IRQ_TIMER2_OVF;
     else
     return IC::IRQ_TIMER2_COMPA;
        }

    void enable(){ timsk0(timsk0() | OCIE0A); }
    void disable(){ timsk0(timsk0() & ~OCIE0A); }

    Hertz frequency() {
if(software_interrupt()){
return _target * count2freq(MIN_COUNT);
} else {
return count2freq(ocr0a());
}
    }
    void frequency(const Hertz & f) {
if(software_interrupt()){
ocr0a(MIN_COUNT);
_target = f/count2freq(MIN_COUNT) + 1;
```

```
} else {
ocr0a(freq2count(f));
}
    }
//    Hertz frequency() const { return count2freq(ocr0a()); }
//    void frequency(const Hertz & f) {
// ocr0a(freq2count(f));
// tccr0a(WGM01);
// tccr0b(TIMER_PRESCALE_1024);
//    };

    void period(Microsecond t) {
     OStream osperiod;
     osperiod << "timer_2:period!!! \n";
  if(software_interrupt()){
  ocr0a(MIN_COUNT);
      _target = t/(min_period() * MIN_COUNT) + 1;
  } else {
      reset();
      unsigned long precise_period = 100000000/CLOCK; // = 800
      if(t > precise_period) {
  t *= 100;
  ocr0a(t/precise_period);
      } else if (t > (precise_period/=10)) {
  t *= 10;
  ocr0a(t/precise_period);
      }
      else
       ocr0a(t/min_period() + 1);//se t = 0, conta 1 tick em hardware
  }
    };

    void reset() {
  tcnt0(0);
  if(software_interrupt()){
```

```
       _elapsed = 0;
  }
     }

     Tick read() {
  return tcnt0();
     }




//    void reset() { tcnt0(0); }

     Microsecond elapsed_time() {
      if(software_interrupt()){
      return (1000000 * read() / CLOCK)+(_elapsed*min_period()*MIN_COUNT);
      } else {
      return 1000000 * read() / CLOCK;
      }
         }

         static const Microsecond min_period() {
      return (1000000/CLOCK);
         }

         static void init(unsigned int i);


//    Tick read() { return tcnt0(); }

//protected:
//    static Hertz count2freq(const Count & c) { return CLOCK / c; }
//    static Count freq2count(const Hertz & f) { return CLOCK / f; }
protected:
```

```
    static const bool software_interrupt() {
if(MAX_INTERVAL > (min_period() * 0xFFFE))
    return true;
else
    return false;
    }

    static void internal_handler(unsigned int);

    static Hertz count2freq(const Count & c) {
return CLOCK / c;
    }
    static Count freq2count(const Hertz & f) {
return CLOCK / f;
    }

protected:

    static Count _elapsed;
    static Count _target;


};

#define max(x,y) ((x > y) ? x : y)

class ATMega1281_Timer_3: public Timer_Common, public AVR_Timer
{
public:
    static const Hertz MACHINE_CLOCK = Traits<Machine>::CLOCK;

    // Traits
    static const Hertz CLOCK = Traits<Machine>::CLOCK >> 8;
    static const RTC::Microsecond MAX_INTERVAL = 1000000;
```

```
    static const Count MIN_COUNT = (Count)(MAX_INTERVAL / (unsigned long)0xFFFF) +

    typedef RTC::Microsecond Microsecond;
    typedef unsigned int Ticks;


public:

    ATMega1281_Timer_3() {
clock(CLOCK);
    }

    ATMega1281_Timer_3(const Hertz & f) {
clock(CLOCK);
frequency(f);
    }

    const void clock(const Hertz clock) {
// if(clock == MACHINE_CLOCK)
// tccr3b(TIMER_PRESCALE_1|WGMn2);
// else if (clock == (MACHINE_CLOCK >> 3))
// tccr3b(TIMER_PRESCALE_8|WGMn2);
// else if (clock == (MACHINE_CLOCK >> 6))
// tccr3b(TIMER_PRESCALE_64|WGMn2);
// else if (clock == (MACHINE_CLOCK >> 8))
// tccr3b(TIMER_PRESCALE_256|WGMn2);
// else
tccr3b(TIMER_PRESCALE_1024|WGMn2);
    }

    const int irq() {
if(software_interrupt())
return IC::IRQ_TIMER3_OVF;
else
return IC::IRQ_TIMER3_COMPA;
```

```
    }

    void enable(){
     timsk3(timsk3() | OCIE3A);
    }

    void disable(){timsk3(timsk3() & ~OCIE3A);}

    Hertz frequency() {
if(software_interrupt()){
return _target * count2freq(MIN_COUNT);
} else {
return count2freq(ocr3a());
}
    }
    void frequency(const Hertz & f) {
if(software_interrupt()){
ocr3a(MIN_COUNT);
_target = f/count2freq(MIN_COUNT) + 1;
} else {
ocr3a(freq2count(f));
}
    }

    void period(Microsecond t) {
if(software_interrupt()){
    ocr3a(MIN_COUNT);
    _target = t/(min_period() * MIN_COUNT) + 1;
} else {
    reset();
    unsigned long precise_period = 100000000/CLOCK; // = 800
    if(t > precise_period) {
t *= 100;
ocr3a(t/precise_period);
    } else if (t > (precise_period/=10)) {
```

```
t *= 10;
ocr3a(t/precise_period);
    }
    else
ocr3a(t/min_period() + 1);//se t = 0, conta 1 tick em hardware
}
OStream printo3;
printo3 << "ocr3a(" << ocr3a() << ")--";
    };

    void reset() {
tcnt3(0);
if(software_interrupt()){
     _elapsed = 0;
}
OStream printo4;
printo4 << "reset()";
    }

    Tick read() {
return tcnt3();
    }

    Microsecond elapsed_time() {
if(software_interrupt()){
return (1000000 * read() / CLOCK)+(_elapsed*min_period()*MIN_COUNT);
} else {
return 1000000 * read() / CLOCK;
}
    }

    static const Microsecond min_period() {
return (1000000/CLOCK);
    }
```

```
    static void init(unsigned int i);



protected:


    static const bool software_interrupt() {
if(MAX_INTERVAL > (min_period() * 0xFFFE))
    return true;
else
    return false;
    }

    static void internal_handler(unsigned int);

    static Hertz count2freq(const Count & c) {
return CLOCK / c;
    }
    static Count freq2count(const Hertz & f) {
return CLOCK / f;
    }

protected:

    static Count _elapsed;
    static Count _target;

};


__END_SYS

#endif
```

```
// EPOS-- PLASMA Timer Mediator Implementation

#include <mach/plasma/timer.h>

__BEGIN_SYS

PLASMA_Timer::Count PLASMA_Timer::_count = (1<<18);  //fixed by hardware

void PLASMA_Timer::int_handler(unsigned int interrupt) {
//will be overrided by alarm
db<PLASMA_Timer>(TRC) << "<Timer::int_handler>";
}

__END_SYS




#ifndef AT86RF230_H_
#define AT86RF230_H_

#include <mach/common/at86rf230/at86rf230_hal.h>
#include <nic.h>

namespace System{

class AT86RF230 {

public:
typedef unsigned int microseconds_t;

typedef enum{
```

```
SUCESS,
BUSY,
FAILED,
FAILED_OPERATING_MODE,
TIME_OUT,
}result_t;

typedef enum{
BASIC_OPERATING_MODE,
EXTENDED_OPERATING_MODE
}operating_mode_t;

typedef enum
{
CCA_DONE_IN_PROGRESS = 0x00,
CCA_DONE_FINISHED  = 0x01,

CCA_STATUS_BUSY = 0x00,
CCA_STATUS_IDLE = 0x01,

TRX_STATUS_P_ON                = 0x00,
TRX_STATUS_BUSY_RX             = 0x01,
TRX_STATUS_BUSY_TX             = 0x02,
TRX_STATUS_RX_ON               = 0x06,
TRX_STATUS_TRX_OFF             = 0x08,
TRX_STATUS_PLL_ON              = 0x09,
TRX_STATUS_SLEEP               = 0x0F,
TRX_STATUS_BUSY_RX_AACK        = 0x11,
TRX_STATUS_BUSY_TX_ARET        = 0x12,
TRX_STATUS_RX_AACK_ON          = 0x16,
TRX_STATUS_TX_ARET_ON          = 0x19,
TRX_STATUS_RX_ON_NOCLK         = 0x1C,
TRX_STATUS_RX_AACK_ON_NOCLK    = 0x1D,
TRX_STATUS_BUSY_RX_AACK_NOCLK  = 0x1E,
TRX_STATUS_STATE_TRANSITION_IN_PROGRESS = 0x1F,
```

```
}state_t;

typedef enum{
ENERGY_ABOVE_THRESHOLD = 0x01,
CARRIER_SENSE_ONLY = 0x02,
CARRIER_SENSE_WITH_ENERGY_ABOVE_THRESHOLD = 0x03
}CCA_mode_t;

public:
AT86RF230();

result_t forceValidState();

void intEnable();

void intDisable();

state_t getState();

operating_mode_t getOperatingMode();

result_t setOperatingMode();

result_t send(unsigned char *buffer, int size);

result_t rx_on();

result_t rx_off();

result_t receive(unsigned char *buffer, int &size);

bool dataReceived();

result_t CCA_measurement(bool &result, CCA_mode_t mode = ENERGY_ABOVE_THRESHOLD,
unsigned char ED_threshold = 0);
```

```
private:

static void intHandler(unsigned int);

static operating_mode_t operatting_mode;

};

}

#endif /* AT86RF230_H_ */
```

```
#include <mach/common/at86rf230/at86rf230.h>
#include <mach/common/at86rf230/at86rf230_hal.h>
#include <semaphore.h>
//#include <alarm_ss.h>
#include <alarm.h>
//#include <display.h>

namespace System{


//MASKS
enum RegisterMask{
TRX_STATUS_MASK_TRX_STATUS = 0x1F,
TRX_STATUS_MASK_CCA_DONE = 0x80,
TRX_STATUS_MASK_CCA_STATUS = 0x40,

PHI_CC_CCA_MASK_CCA_MODE = 0x60,
PHI_CC_CCA_MASK_CCA_REQUEST = 0x80,
```

```
};

enum TRX_STATE_REG {
TRAC_STATUS_SUCCESS                = 0x00,
TRAC_STATUS_CHANNEL_ACCESS_FAILURE = 0x03,
TRAC_STATUS_NO_ACK                 = 0x05,

TRX_CMD_NOP            = 0x00,
TRX_CMD_TX_START       = 0x02,
TRX_CMD_FORCE_TRX_OFF  = 0x03,
TRX_CMD_RX_ON          = 0x06,
TRX_CMD_TRX_OFF        = 0x08,
TRX_CMD_PLL_ON         = 0x09,
TRX_CMD_RX_AACK_ON     = 0x16,
TRX_CMD_TX_ARET_ON     = 0x19,
};

//Interrupt status and masks
enum Interrupt{
IRQ_STATUS_BAT_LOW = 0x80,
IRQ_STATUS_TRX_UR = 0x40,
IRQ_STATUS_TRX_END = 0x08,
IRQ_STATUS_RX_START = 0x04,
IRQ_STATUS_PLL_UNLOCK = 0x02,
IRQ_STATUS_PLL_LOCK = 0x01,

IRQ_STATUS_ALL = 0xCF,
IRQ_STATUS_UNKNOWN = 0x00,
};

//initialize static members
AT86RF230::operating_mode_t AT86RF230::operatting_mode = AT86RF230::BASIC_OPERATING

//variables
//volatile bool pll_locked = false;
```

```
//volatile bool sendingData = false;

//static volatile bool _dataReceived = false;

Semaphore int_trx_end(0);


AT86RF230::AT86RF230(){
AT86RF230_HAL::init();
// AT86RF230_HAL::setInterruptHandler(&intHandler);
// AT86RF230_HAL::initInterrupts();
forceValidState();
//TODO check when extended mode is implemented
operatting_mode = AT86RF230::BASIC_OPERATING_MODE;
}

AT86RF230::result_t AT86RF230::forceValidState(){
AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_STATE_REG, TRX_CMD_FORCE_TRX_OFF);
//TODO check when extended mode is implemented
operatting_mode = AT86RF230::BASIC_OPERATING_MODE;
if(getState() != TRX_STATUS_TRX_OFF)
return FAILED;
else
return SUCESS;
}

void AT86RF230::intEnable(){
AT86RF230_HAL::interruptEnable();

}

void AT86RF230::intDisable(){
AT86RF230_HAL::interruptDisable();
}
```

```
AT86RF230::state_t AT86RF230::getState(){
unsigned char value = AT86RF230_HAL::readRegister(AT86RF230_HAL::TRX_STATUS_REG);
value &= TRX_STATUS_MASK_TRX_STATUS;
return static_cast<AT86RF230::state_t>(value);
}

AT86RF230::operating_mode_t AT86RF230::getOperatingMode(){
//TODO implement
return BASIC_OPERATING_MODE;
}

AT86RF230::result_t AT86RF230::setOperatingMode(){
//TODO implement
return FAILED;
}


unsigned char ckp_count = 0;
void checkpoint(){
ckp_count = (ckp_count > 7)?0:(ckp_count+1);

CPU::out8(Machine::IO::PORTB, CPU::in8(Machine::IO::PORTB) & ~(0xE0));
CPU::out8(Machine::IO::PORTB, CPU::in8(Machine::IO::PORTB) | (ckp_count << 5));
}

AT86RF230::result_t AT86RF230::send(unsigned char *buffer, int size){
OStream print2;
ckp_count = 0;
checkpoint();
print2 << "send[0]";
if(getState() != AT86RF230::TRX_STATUS_TRX_OFF){
// print2 << "BUSY["<< getState() <<"]";
return BUSY;
}
```

```
print2 << "send[1]";
if(operatting_mode == AT86RF230::BASIC_OPERATING_MODE){
checkpoint();
//sendingData = true;
print2 << "send[2]";
AT86RF230_HAL::Frame frame;
frame.data = buffer;
frame.frame_length = size;
checkpoint();
print2 << "send[3]";
//write date to the frame buffer
AT86RF230_HAL::writeFrameBuffer(&frame);
checkpoint();
//go to PLL_ON state and wait for pll to lock
AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_STATE_REG, TRX_CMD_PLL_ON);
checkpoint();
/*while(true){
mutex_pll.p();
if(pll_locked){
mutex_pll.v();
break;
}
mutex_pll.v();
}*/
while(getState() != TRX_STATUS_PLL_ON);

checkpoint();

//starting TX

AT86RF230_HAL::setSLP_TRhigh();
AT86RF230_HAL::setSLP_TRlow();

/*if(getState() != TRX_STATUS_BUSY_TX)
return FAILED;*/
```

```
print2 << "send[4]";
// Alarm::delay(5000); //5000 deve funcionar


print2 << "send[5]";
//should be back to PLL_ON state when TX end
while(getState() != TRX_STATUS_PLL_ON){
print2 << "!send[getState]"<< getState() << "--!";
}
print2 << "send[6]";

checkpoint();
print2 << "send[7]";

//return to TRX_OFF
AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_STATE_REG, TRX_CMD_TRX_OFF);
checkpoint();
if(getState() != TRX_STATUS_TRX_OFF){
//sendingData = false;
print2 << "send[8]";
return FAILED;
}
print2 << "send[9]";
//sendingData = false;
return SUCESS;
}
else{ //extended operating mode
//TODO implement
return FAILED;
}
}


AT86RF230::result_t AT86RF230::rx_on(){
AT86RF230_HAL::setFrequency();
```

```
OStream print3;

// print3 << "AT86RF230::rx_on(" << getState() << ")\n";
if(getState() != TRX_STATUS_TRX_OFF){
print3 << "AT86RF230::rx_on(BUSY)\n";
return BUSY;
}

//_dataReceived = false;

ckp_count = 0;

if(operatting_mode == AT86RF230::BASIC_OPERATING_MODE){

checkpoint();

//go to RX_ON and wait pll to lock
AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_STATE_REG, TRX_CMD_RX_ON);

checkpoint();

/*while(true){
mutex_pll.p();
if(pll_locked){
mutex_pll.v();
break;
}
mutex_pll.v();
}*/
//while(!pll_locked);
while(getState() != TRX_STATUS_RX_ON);

checkpoint();

if(getState() != TRX_STATUS_RX_ON)
```

```
return FAILED;

//checkpoint();
print3 << "AT86RF230::rx_on(SUCESS)\n";
return SUCESS;
}
else{ //extended operating mode
//TODO implement
return FAILED;
}
}

AT86RF230::result_t AT86RF230::rx_off(){

state_t state = getState();

if(state == TRX_STATUS_BUSY_RX)
return BUSY;
else{
return forceValidState();
}
}

AT86RF230::result_t AT86RF230::receive(unsigned char *buffer, int &size){

if(getState() != TRX_STATUS_TRX_OFF)
return BUSY;

//if(!dataReceived()){
// size = 0;
// return SUCESS;
//}
//else{

AT86RF230_HAL::Frame frame;
```

```
frame.data = buffer;

//read the received data
AT86RF230_HAL::readFrameBuffer(&frame);

size = frame.frame_length;

//_dataReceived = false;

return SUCESS;
//}
}

bool AT86RF230::dataReceived(){
// return _dataReceived;
bool aux = AT86RF230_HAL::readRegister(AT86RF230_HAL::IRQ_STATUS_REG) & IRQ_STATUS_
// Alarm::delay(100000);
return aux;
}

//OStream os2;

AT86RF230::result_t AT86RF230::CCA_measurement(bool &result, CCA_mode_t mode, unsig

if(getState() != TRX_STATUS_TRX_OFF)
return BUSY;

//set the energy threshold
//????

//set cca mode and request
unsigned char regVal = AT86RF230_HAL::readRegister(AT86RF230_HAL::PHY_CC_CCA_REG);

//os2 << "PHY_CC_CCA: " << static_cast<unsigned int>(regVal) << "\n";
```

```
regVal &= ~PHI_CC_CCA_MASK_CCA_MODE;
regVal |= (mode << 5);
regVal |= PHI_CC_CCA_MASK_CCA_REQUEST;

//os2 << "PHY_CC_CCA: " << static_cast<unsigned int>(regVal) << "\n";

//start the measurement
result_t aux = rx_on();
if(aux != SUCESS)
return FAILED;

AT86RF230_HAL::writeRegister(AT86RF230_HAL::PHY_CC_CCA_REG, regVal);


//poll the radio to check if the measurement is finished
while(true){
regVal = AT86RF230_HAL::readRegister(AT86RF230_HAL::TRX_STATUS_REG);
// os2 << "TRX_STATUS: " << static_cast<unsigned int>(regVal) << "\n";
if(regVal & TRX_STATUS_MASK_CCA_DONE)
break;
}

forceValidState();

//check the result
result = regVal & TRX_STATUS_MASK_CCA_STATUS;

return SUCESS;
}

void AT86RF230::intHandler(unsigned int){

// OStream printHandler;
// unsigned char val = AT86RF230_HAL::readRegister(AT86RF230_HAL::IRQ_STATUS_REG);
// printHandler << "Interupt Handler:\n";
```

```
/*unsigned char val = AT86RF230_HAL::readRegister(AT86RF230_HAL::IRQ_STATUS_REG);

//os2 << "#INT=" << val << "#";

if (val & IRQ_STATUS_RX_START) {

}
else if (val & IRQ_STATUS_TRX_END){
if(sendingData)
int_trx_end.v();
else{
//go back to TRX_OFF
AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_STATE_REG, TRX_CMD_TRX_OFF);

_dataReceived = true;
}
}
else if (val & IRQ_STATUS_PLL_LOCK){
//mutex_pll.p();
pll_locked = true;
//mutex_pll.v();
}
else if (val & IRQ_STATUS_PLL_UNLOCK){
//mutex_pll.p();
pll_locked = false;
//mutex_pll.v();
}
else if(val & IRQ_STATUS_TRX_UR){
//does'n handle frame buffer access violation
}
else if(val & IRQ_STATUS_BAT_LOW){
//Disable BAT_LOW interrupt to prevent interrupt storm. The interrupt
//will continously be signaled when the supply voltage is less than the
//user defined voltage threshold.
```

```
unsigned char trx_isr_mask = AT86RF230_HAL::readRegister(AT86RF230_HAL::IRQ_MASK_RE
trx_isr_mask &= ~IRQ_STATUS_BAT_LOW;
AT86RF230_HAL::writeRegister(AT86RF230_HAL::IRQ_MASK_REG, trx_isr_mask);
}*/


}




}
```

```
#ifndef AT86RF230_HAL_H_
#define AT86RF230_HAL_H_

#include <ic.h>
#include <machine.h>

namespace System {

class AT86RF230_HAL {

public:

/*Registers' addresses*/
enum Register {

TRX_STATUS_REG   = 0x01,
TRX_STATE_REG    = 0x02,
TRX_CTRL_0_REG   = 0x03,
PHY_TX_PWR_REG   = 0x05,
```

```
PHY_RSSI_REG     = 0x06,
PHY_ED_LEVEL_REG = 0x07,
PHY_CC_CCA_REG   = 0x08,
CCA_THRES  = 0x09,
IRQ_MASK_REG     = 0x0E,
IRQ_STATUS_REG   = 0x0F,
VREG_CTRL_REG    = 0x10,
BATMON_REG       = 0x11,
XOSC_CTRL_REG    = 0x12,
FTN_CTRL_REG     = 0x18,
VERSION_NUM_REG  = 0x1D,
MAN_ID_0_REG     = 0x1E,
MAN_ID_1_REG     = 0x1F,
SHORT_ADDR_0_REG = 0x20,
SHORT_ADDR_1_REG = 0x21,
PAN_ID_0_REG     = 0x22,
PAN_ID_1_REG     = 0x23,
IEEE_ADDR_0_REG  = 0x24,
IEEE_ADDR_1_REG  = 0x25,
IEEE_ADDR_2_REG  = 0x26,
IEEE_ADDR_3_REG  = 0x27,
IEEE_ADDR_4_REG  = 0x28,
IEEE_ADDR_5_REG  = 0x29,
IEEE_ADDR_6_REG  = 0x2A,
IEEE_ADDR_7_REG  = 0x2B,
XAH_CTRL_REG     = 0x2C,
CSMA_SEED_0_REG  = 0x2D,
CSMA_SEED_1_REG  = 0x2E,
};

/*Constants*/
enum{
MAX_FRAME_LENGTH = 127
};
```

```
public:

typedef struct{
unsigned char *data;
unsigned char frame_length;
unsigned char lqi;
} Frame;


public:

static void init();
static unsigned char readRegister(Register reg);
static void writeRegister(Register reg, unsigned char value);
static void readFrameBuffer(Frame *frame);
static void writeFrameBuffer(Frame *frame);
static void setRSTlow();
static void setRSThigh();
static void setSLP_TRlow();
static void setSLP_TRhigh();
static void initInterrupts();
static void interruptEnable();
static void interruptDisable();
static void setInterruptHandler(Machine::int_handler handler);

};

}

#endif /* AT86RF230_HAL_H_ */




#include <mach/common/at86rf230/at86rf230_hal.h>
```

```
#include <ic.h>
#include <machine.h>
#include <cpu.h>
#include <traits.h>
#include <alarm.h>

namespace System {

typedef IO_Map<Machine> IO;

/*
Radio pins

Mega1281 AT86RF230
PA7 -> RST
PB0 -> SEL
PB4 -> SLP_TR
PD6 -> CLKM
PE5 -> IRQ
 */
enum {
SS_PIN       = 0,
SCK_PIN      = 1,
MOSI_PIN     = 2,
MISO_PIN     = 3,
SLP_TR  = 4,
RST  = 7,
};

/*
SPI Registers
*/
enum {
// SPCR
SPIE     = 7,
```

```
SPE      = 6,
DORD     = 5,
MSTR     = 4,
CPOL     = 3,
CPHA     = 2,
SPR1     = 1,
SPR0     = 0,
// SPSR
SPIF     = 7,
WCOL     = 6,
SPI2X    = 0,
};
/*
Commands
*/
enum {
REG_READ = 0x80,
REG_WRITE = 0xC0,
FB_READ = 0x20,
FB_WRITE = 0x60,
};


void SPI_SS_high(){
AVR8::out8(IO::PORTB, AVR8::in8(IO::PORTB) | (1<<SS_PIN));
}

void SPI_SS_low(){
AVR8::out8(IO::PORTB, AVR8::in8(IO::PORTB) & ~(1<<SS_PIN));
}

unsigned char SPI_dummy = 0;

void SPI_transmit(unsigned char toSlave, unsigned char &fromSlave){
AVR8::out8(IO::SPDR, toSlave);
```

```
while(!(AVR8::in8(IO::SPSR) & (1<<SPIF)));
fromSlave = AVR8::in8(IO::SPDR);
}

bool AT86RF230_init_ok = false;

void AT86RF230_HAL::init(){
if(AT86RF230_init_ok)
return;

AT86RF230_init_ok = true;

Alarm::delay(500000); //time to enter state P_ON

/*IO Specific Initialization.*/
AVR8::out8(IO::DDRB, AVR8::in8(IO::DDRB) | (1 << SLP_TR)); //Enable SLP_TR as outpu
AVR8::out8(IO::DDRA, AVR8::in8(IO::DDRA) | (1 << RST)); //Enable RST as output.

/*SPI Specific Initialization.*/
//Set SS, CLK and MOSI as output.
AVR8::out8(IO::DDRB, AVR8::in8(IO::DDRB) |
((1 << SS_PIN) | (1 << SCK_PIN) | (1 << MOSI_PIN)));
//Set SS and CLK high
AVR8::out8(IO::PORTB, AVR8::in8(IO::PORTB) |
((1 << SS_PIN) | (1 << SCK_PIN)));
AVR8::out8(IO::SPCR, (1<<SPE)|(1<<MSTR));//Enable SPI module and master operation.
AVR8::out8(IO::SPSR, (1 << SPI2X)); //Enable doubled SPI speed in master mode.

/*reset the radio*/
setRSTlow();
setSLP_TRlow();
Alarm::delay(500000); //time to reset
setRSThigh();
// setFrequency();
}
```

```
void AT86RF230_HAL::setFrequency(){//Parece que nao esta funcionando, mas continuar
// OStream freqprint;
// unsigned char aux = AT86RF230_HAL::readRegister(AT86RF230_HAL::TRX_CTRL_0_REG);
// aux = aux & 0xf0; //Clear first 4 bits
// aux = aux | 0x04; //Write the value 4(8mhz) on first 3 bits
// aux = aux & ~0x08;//Clock rate changes immediately
// AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_CTRL_0_REG, aux);
// AT86RF230_HAL::writeRegister(AT86RF230_HAL::TRX_CTRL_0_REG, aux);
// //bit 0=0  1=0 2=0 p/ 8mhz
// freqprint << "TRX_CTRL_0: "<< AT86RF230_HAL::readRegister(AT86RF230_HAL::TRX_CTR
}

unsigned char AT86RF230_HAL::readRegister(AT86RF230_HAL::Register reg){

unsigned char value;

SPI_SS_low();
SPI_transmit((REG_READ | reg), SPI_dummy);//send command
SPI_transmit(SPI_dummy, value); //read value
SPI_SS_high();

return value;
}

void AT86RF230_HAL::writeRegister(Register reg, unsigned char value){
SPI_SS_low();
SPI_transmit((REG_WRITE | reg), SPI_dummy); //send command
SPI_transmit(value, SPI_dummy); //send value
SPI_SS_high();
}

void AT86RF230_HAL::readFrameBuffer(AT86RF230_HAL::Frame *frame){
SPI_SS_low();
```

```
SPI_transmit(FB_READ, SPI_dummy); //send command
SPI_transmit(SPI_dummy, frame->frame_length); //read frame size

//read frame
for(int i = 0; i < frame->frame_length; ++i)
SPI_transmit(SPI_dummy, frame->data[i]);

SPI_transmit(SPI_dummy, frame->lqi); //read lqi value

SPI_SS_high();
}

void AT86RF230_HAL::writeFrameBuffer(AT86RF230_HAL::Frame *frame){
SPI_SS_low();

SPI_transmit(FB_WRITE, SPI_dummy); //send command
SPI_transmit(frame->frame_length, SPI_dummy); //send frame size

//write frame
for(int i = 0; i < frame->frame_length; ++i)
SPI_transmit(frame->data[i], SPI_dummy);

SPI_SS_high();
}

void AT86RF230_HAL::setRSTlow(){
AVR8::out8(IO::PORTA, AVR8::in8(IO::PORTA) & ~(1 << RST));
}

void AT86RF230_HAL::setRSThigh(){
AVR8::out8(IO::PORTA, AVR8::in8(IO::PORTA) | (1 << RST));
}

void AT86RF230_HAL::setSLP_TRlow(){
AVR8::out8(IO::PORTB, AVR8::in8(IO::PORTB) & ~(1 << SLP_TR));
```

```
}

void AT86RF230_HAL::setSLP_TRhigh(){
AVR8::out8(IO::PORTB, AVR8::in8(IO::PORTB) | (1 << SLP_TR));
}

/*
 * Interrupt handling
 */

typedef enum{
IRQ_0  = 0, // ??
IRQ_1,// ??
IRQ_2,// ??
IRQ_3,// ??
IRQ_4,// ??
IRQ_5,// radio
IRQ_6,// button 1
IRQ_7,  // button 2
} irqNumber_t;

// interrupt activation condition.
typedef enum
{
IRQ_LOW_LEVEL,     // The low level generates an interrupt request.
IRQ_ANY_EDGE,      // Any edge generates an interrupt request.
IRQ_FALLING_EDGE,  // Falling edge generates an interrupt request.
IRQ_RISING_EDGE    // Rising edge generates an interrupt request.
} irqMode_t;

void enable_external_int(irqNumber_t irqNumber, irqMode_t irqMode){
// IRQ pin is input
CPU::out8(IO::DDRE, CPU::in8(IO::DDRE) & ~(1 << irqNumber));
CPU::out8(IO::PORTE, CPU::in8(IO::PORTE) | (1 << irqNumber));
unsigned char ui8ShiftCount = (irqNumber - IRQ_4) << 1;
```

```
// Clear previous settings of corresponding interrupt sense control
CPU::out8(IO::EICRB, CPU::in8(IO::EICRB) & ~(3 << ui8ShiftCount));
// Setup corresponding interrupt sence control
CPU::out8(IO::EICRB, CPU::in8(IO::EICRB) | ((irqMode & 0x03) << ui8ShiftCount));
// Clear the INTn interrupt flag
CPU::out8(IO::EIFR, CPU::in8(IO::EIFR) & ~(1 << irqNumber));
}

bool AT86RF230_init_interrupts_ok = false;

void AT86RF230_HAL::initInterrupts(){

if(AT86RF230_init_interrupts_ok)
return;

AT86RF230_init_interrupts_ok = true;

enable_external_int(IRQ_5, IRQ_RISING_EDGE);
interruptEnable();
}

void AT86RF230_HAL::interruptEnable(){
AVR8::out8(IO::EIMSK, CPU::in8(IO::EIMSK) | (1 << IRQ_5));
AT86RF230_HAL::readRegister(AT86RF230_HAL::IRQ_STATUS_REG);//Reset Interrupt Reg
}

void AT86RF230_HAL::interruptDisable(){
AVR8::out8(IO::EIMSK, CPU::in8(IO::EIMSK) & ~(1 << IRQ_5));
}

void AT86RF230_HAL::setInterruptHandler(Machine::int_handler handler){
Machine::int_vector(IC::IRQ_IRQ5, handler);
}

}
```