Marco Antonio Santuci Carvalho
Orientador: Claudionor Nunes Coelho Jr.

Um Sistema de Monitoramento Remoto de Pacientes usando Rede sem Fio

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Novembro, 2005

Belo Horizonte

Resumo

Este trabalho implementa um sistema de monitoramento remoto de pacientes através da comunicação pela Internet utilizando o protocolo *TCP/IP* e uma rede sem fio. A aplicação principal do trabalho é voltada para a área biomédica onde o paciente pode ter alguns dados fisiológicos monitorados 24 horas por dia ou enviá-los para o seu médico em caso de emergência.

Para a implementação do trabalho foi utilizado um circuito de supervisão de eletrocardiograma (*ECG*) que teve de ser adaptado para fazer a comunicação por rede sem fio através de um *driver* de dispositivo (*Device Driver*). Duas aplicações que executam em cima da pilha *TCP/IP* foram adaptadas e testadas para funcionamento com fins de monitoramento de pacientes de forma diferenciada. A implementação dos trabalhos considerou aspectos tais como custo, gerenciamento de energia e tolerância à falhas, fatores indispensáveis em um equipamento de monitoramento remoto na área médica.

Os grandes desafios do projeto foram a implementação do *driver* do dispositivo de comunicação sem fio e a adaptação do circuito ECG para comunicação com a interface que implementa a rede sem fio. A grandes restrições de projeto foram a elaboração do software considerando os escassos recursos de um microcontrolador de oito bits e também a adaptação do projeto a partir de um circuito já existente.

Durante o desenvolvimento dos trabalhos para a implementação do sistema de comunicação sem fio, foi identificado a necessidade de alteração do hardware inicial. Devido a limitada disponibilidade de recursos de hardware, algumas funcionalidades originais da placa tiveram de ser substituídas em troca do funcionamento correto do sistema de comunicação. Como solução para este problema é proposto um novo projeto que adiciona pouca modificação no circuito original e mantém a proposta de baixo custo.

Aplicações práticas do projeto foram testadas e comprovaram que ele pode ser utilizado não somente para aplicações médicas, mas também em outras que necessitem de comunicação utilizando rede sem fio.

Abstract

This Work implements a remote system to monitor medical patients by using Internet communication through *TCP/IP* protocol and wireless network. The main application is in the biomedical area where the patient can have some physiological data monitored 24 hours per day or send it to his doctor in an emergency situation.

The system was developed using a circuit already done that measure ECG data. It was modified to support wireless network using a *Device Driver*. Two applications runs in the top of the *TCP/IP* stack were developed and adapted to monitor patients in different ways. The project implementation considered some issues like cost, energy management and fault tolerance.

The main project challenges were the *Device Driver* implementation and the circuit modification to support wireless network. The biggest difficulties were the software implementation considering the poor 8-bit microcontroller resources and project adaptation from the circuit already existed.

Some practical project applications were tested and proved that it is not limited to be used only in the medical area but also in others that needed communication using wireless network.

Agradecimentos

Agradeço primeiramente a Deus que me deu forças e me ajudou a continuar na trajetória do término desta jornada apesar dos momentos mais difíceis quando cheguei a pensar em desistir.

Aos meus pais, Antônio e Cristina que me deram apoio e condições de vida para que eu tivesse chegado até aqui. Aos meus irmãos, Júlio e Patrícia que mesmo distantes sempre tinham uma palavra de incentivo.

À Cíntia que sempre esteve ao meu lado me suportando no que eu precisava e entendendo as noites e fins de semana de incansável trabalho.

Ao meu orientador, Claudionor que me deu liberdade no desenvolvimento dos trabalhos de acordo com as minhas habilidades e conseguiu me motivar a buscar novos conhecimentos.

Aos colegas de trabalho da Jabil que me incentivaram e apoiaram desde o inicio para que eu conseguisse frequentar as aulas e possibilitaram que eu pudesse realizar este trabalho.

A todas as outras pessoas que me auxiliaram nos trabalhos seja direta ou indiretamente.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Problemas do Atendimento Médico de Pacientes

As pessoas, em geral as mais idosas, são carentes de novas tecnologias que facilitem o acesso delas a um atendimento médico mais adequado com a sua condição física, que muitas vezes debilitada, impede a sua locomoção até os hospitais. Os hospitais na maioria das vezes não oferecem o atendimento adequado devido a ausência de recursos financeiros, disponibilidade de profissionais, quantidade de leitos disponíveis dentre outros. Além disso o custo de internação de um paciente, principalmente de um idoso, é muito alto.

Geralmente as pessoas visitam o médico com a freqüência de no mínimo uma vez por ano para uma avaliação geral ou quando apresentam um quadro clínico de uma doença que as incomoda. O médico por sua vez não consegue diagnosticar certos sintomas por falta de um acompanhamento mais contínuo da vida do paciente. Em uma pessoa idosa, de acordo com a natureza humana, é bem mais provável que maiores problemas de saúde possam ocorrer e por isto um acompanhamento mais frequente pode evitar doenças e até mesmo a morte.

1.2 Motivação para o Trabalho

O monitoramento remoto de pacientes permite que vários destes problemas sejam amenizados. Sensores acoplados a um paciente podem coletar dados fisiológicos e os transmitirem para uma central médica remota para que sejam analisados ou para um posterior diagnóstico. Isto evita por exemplo que uma pessoa idosa tenha que se locomover até um hospital ou ao consultório de seu médico para consulta eventual ou de emergência.

O monitoramento remoto possibilita também diminuição dos gastos dos estabelecimentos médicos pois reduz a quantidade de consultas. O monitoramento remoto possibilita ao médico condições de dar um diagnóstico mais preciso porque é possível acompanhar os dados históricos do paciente e não somente aqueles coletados no momento da consulta.

O monitoramento de dados à distancia surge então como solução para diversos problemas do cotidiano de pacientes médicos, representados principalmente pelas pessoas idosas conforme discutido. A comunidade científica é atenta à necessidade de tratamento desta situação tem feito várias pesquisas nesta área que ainda é aberta a vários questionamentos e alternativas.

Diversos aspectos dividem os pesquisadores em relação ao monitoramento remoto de pacientes. Um deles diz respeito ao tipo de controle da medicação. Uma medicação próativa de forma que o paciente possa ter todos os dados em mãos e controlar seu estado de saúde é uma alternativa de acompanhamento médico à distância. Esta forma de acompanhamento não pode ser generalizada com sucesso no caso de idosos porque pode requerer algum conhecimento médico ou a habilidade para lidar com números, dados e/ou gráficos ou sinais luminosos ou sonoros que podem gerar confusão podendo ao invés de ajudar, atrapalhar com a possibilidade de algum diagnóstico errado. O ideal seria que o paciente pudesse coletar seus dados fisiológicos tais como pressão, temperatura, etc e os pudesse repassar remotamente via um sistema de comunicação. Por outro lado, estes dados seriam recebidos por uma central computacional que faria a análise deles. Baseado em dados históricos do paciente, informações de sintomas previamente fornecidas ou até mesmo modelos matemáticos, a central remota faria algum diagnóstico e enviaria para o paciente. O sistema poderia também processar os dados recebidos do paciente e envia-los para o médico quando algum dado saísse fora do padrão esperado cabendo então o médico enviar ou contactar o paciente para lhe dar o diagnóstico e a devida medicação. Uma idéia desta última alternativa é mostrada na figura 1.1.

Os primeiros equipamentos de monitoramento contínuo de pacientes que ainda são utilizados hoje em dia tais como o *Holter* e *MAPA* utilizam o conceito *off-line* onde o paciente tem os dados dos sensores fisiológicos armazenados em algum tipo de memória e depois devem descarrega-los em um outro equipamento que possibilita futura análise. Equipamentos deste tipo tem utilidade limitada sendo aplicáveis em situações onde os dados dos sensores podem ser analisados no final de um dia, não necessitando de acompanhamento em curtos intervalos, de hora em hora por exemplo. Outro inconveniente é o fato de o paciente ter que se dirigir ao centro médico onde se localiza um equipamento para descarregar os dados.

A monitoração dos dados em tempo real (*on-line*) tem revolucionado a medicina através do surgimento de dispositivos embutidos, que incorporam diversos sensores capazes de monitorar várias atividades fisiológicas dos pacientes. Estes dispositivos são dotados de sistemas de comunicação capazes de enviar os dados para serem analisados nos centros médicos especializados. Estes sistemas de comunicação utilizam tecnologia sem fio, o que garante ao paciente mobilidade na sua região de cobertura.

Seguindo este raciocínio, pesquisadores tem concentrado seus estudos em duas áreas principais: tornar os dispositivos embutidos cada vez mais completos em termos de sensores e desenvolver aplicações para análise dos dados pelo próprio dispositivo. Pouco se tem preocupado com a compactação do hardware do dispositivo, que na maioria dos casos implementados, utiliza mini *PCs(Personal Computers)* ou *PDAs(Personal Digital Assistants)*. Esta solução não sendo customizada acaba encarecendo o dispositivo o que é economicamente inviável em se falando em distribuição em larga escala. Outros fatores

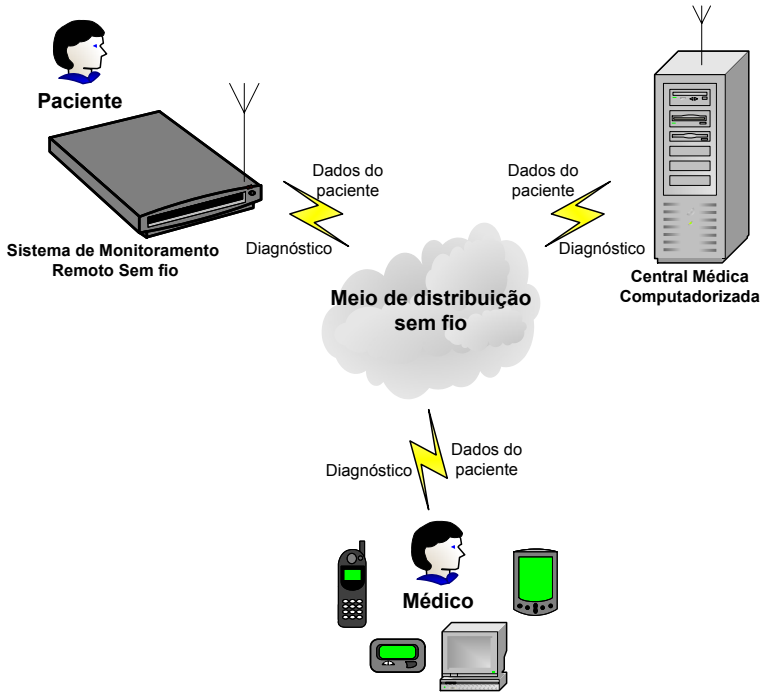


Figura 1.1: Sistema de Monitoramento Remoto com análise e diagnóstico dos dados do paciente feitos remotamente

que também pesam na utilização destes dispositivos com o hardware genérico é o seu peso comparado a uma solução que poderia ser mais compacta e o consumo com a utilização de vários sensores.

Como exemplo de um sistema de monitoramento remoto, pode-se citar dois sistemas desenvolvidos no DCC/UFMG. O primeiro deles é o Monitor de Sinais Vitais Multi-paramétrico Vestível [Con01] que foi implementado para monitoração de parâmetros vitais tais como a supervisão do eletrocardiograma (ECG) e nível de oxigenação do sangue. Neste projeto foram implementados o hardware para aquisição de sinais e o software para aquisição e transmissão dos dados em rede. Este projeto se caracterizou também pelo desenvolvimento de uma vestimenta apropriada de forma a carregar todo o hardware, permitindo mobilidade ao paciente. O projeto possibilitou uma integração de diversas tecnologias de hardware e software mas teve como ponto crítico o peso dos equipamentos e a autonomia de energia.

O Outro sistema desenvolvido no DCC/UFMG foi o Monitor Cardíaco [FLCJ03]. Este trabalho apresentou uma nova proposta para o desenvolvimento de um sistema embutido para monitoramento de dados fisiológicos de seres humanos através da utilização de um hardware com componentes eletrônicos com maior escala de integração, conseguindo-se com isto uma redução do tamanho do hardware. Este projeto, ao contrário do anterior, foi projetado a princípio para comunicação sem fio a partir de uma rede celular embora

não tenha sido implementado nenhuma aplicação para teste deste sistema. Este projeto foi caracterizado pelo esforço na redução do hardware anterior mantendo algumas de suas funções. A parte de software de aplicação e do sistema de comunicação do projeto não foram muito exploradas.

1.3 Objetivos e Contribuições esperadas para o Trabalho

Dando continuidade de pesquisa na área biomédica e desenvolvimento de sistemas embutidos na área de monitoramento remoto de pacientes, este trabalho desenvolve uma aplicação focando na implementação de um sistema de comunicação sem fio utilizando o mesmo hardware do projeto Monitor Cardíaco. Além disso, o projeto pretende ser mais eficiente no consumo de energia comparado com o Monitor Multiparamétrico Vestível.

Pretende-se também mostrar com este trabalho a possibilidade de ter um sistema dedicado para monitoramento de pacientes à distância através de um sistema de comunicação sem fio, mesmo que as características de hardware sejam limitadas. As restrições de hardware também provocaram a busca de soluções e utilização de técnicas que serão mostradas na apresentação do trabalho. Os limites das aplicações serão testados e verificados e será identificado o que é factível de ser implementado com o hardware proposto.

A contribuição esperada para o projeto é a implementação de um sistema remoto de monitoramento de pacientes utilizando uma rede de comunicação sem fio priorizando baixo custo e consumo.

1.4 Organização do Trabalho

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta as necessidades e decisões de projeto de um sistema de monitoramento remoto de pacientes; no Capítulo 3 são analisados trabalhos relacionados com as decisões de projeto do capítulo anterior. O Capítulo 4 apresenta o desenvolvimento do sistema de comunicação utilizado, com as devidas considerações para integração com rede sem fio e também o desenvolvimento das aplicações incluindo suporte para redução do consumo de energia e tolerância a falhas. O Capítulo 5 apresenta os testes, resultados e principais dificuldades. O Capítulo 6 trata da nova versão do projeto e, finalmente, o Capítulo 7 das conclusões e dos trabalhos futuros.

Capítulo 2

Necessidades e Decisões de Projeto

O objetivo deste capítulo é apresentar as considerações, necessidades e decisões para o projeto de um sistema de monitoramento médico remoto com sistema de comunicação sem fio. As necessidades para o projeto serão enfatizadas e algumas soluções para o projeto serão apresentadas. Logo em seguida, baseando-se nestas necessidades e nas restrições de projeto, as decisões de maneira geral serão apresentadas. Decisões e implementações mais específicas do projeto serão mostradas e discutidas nos capítulos seguintes.

2.1 Pré-Requisitos do Sistema de Monitoramento Remoto de Pacientes

Através da observação da figura 1.1 no capítulo 1 que os pré-requisitos de um Sistema de Monitoramento Remoto pode ser dividido basicamente em três subconjuntos: necessidades do Sistema Embutido de Monitoramento Remoto, necessidades do Servidor Remoto de dados (Central médica computadorizada ou equipamentos remotos do médico do paciente) e requisitos do Sistema de distribuição sem Fio.

Como o trabalho é focado no desenvolvimento do Sistema Embutido de Monitoramento Remoto será visto primeiramente as necessidades dele e em seguida as necessidades das outras partes integrantes do Sistema de Monitoramento Remoto de modo a interagirem com a primeira.

2.1.1 Requisitos do Sistema Embutido de Monitoramento Remoto

Primeiramente, para ser um monitor de dados fisiológicos, obviamente o sistema embutido precisa ter a capacidade de monitorar dados fisiológicos dos pacientes. O objetivo aqui não é monitorar diferentes tipos de dados *multiparamétrico* conforme outros projetos mas sim

conseguir monitorar remotamente um ou mais dados. Através deste princípio, o conceito do monitoramento pode ser extensível facilmente, apenas ampliando a quantidade de hardware e processamento de dados via software.

É necessário também que o monitor remoto seja leve e compacto para que possa ser utilizado pelo paciente como um acessório qualquer tal como uma carteira ou um telefone celular sem que isto incomode ou atrapalhe seus movimentos.

O sistema precisa estar habilitado para comunicar sem fio. Isto é necessário para que ele possa ser usado de forma a permitir mobilidade não ficando o paciente preso a fios.

Em termos de aplicação, é necessário que o paciente consiga abrir uma conexão a qualquer momento que desejar para o envio de dados e também que o médico possa acessar os dados remotamente a qualquer momento conseguindo abrir uma conexão com o monitor cardíaco.

É necessário também que o sistema tenha um baixo consumo para que o monitor remoto possa ser utilizado pelo paciente permitindo grandes deslocamentos sem necessidade de recarga de bateria, possibilitando um monitoramento mais prolongado.

Para que seja possível a interpretação do diagnóstico enviado pela central médica ou pelo médico do paciente, é necessário que o sistema embutido de monitoramento remoto seja capaz prover informação de alguma forma como por exemplo através de sinais luminosos, mensagens em um *Display* de Cristal Líquido *(LCD)* ou até mesmo sinais audíveis de voz.

De maneira geral é importante que o código de programa que implemente o sistema de comunicação e as aplicações execute rapidamente, ou seja, possua um código eficiente para que o tempo de execução dele seja rápido. Um outro aspecto a ser considerado é o tamanho do código. O espaço ocupado pelo código não pode ocupar grande parte da memória de programa, para poder ser capaz de dar liberdade para o desenvolvimento ou melhoria de aplicações e funcionalidades.

2.1.2 Requisitos do Servidor Remoto de Dados

Se for pensado que vários sistemas embutidos de monitoramento remoto podem estar funcionando ao mesmo tempo, o servidor remoto que irá receber os dados dos pacientes deve suportar várias conexões. De forma a manter um histórico de dados do paciente, o servidor de dados deve possuir um banco de dados com grande capacidade de armazenamento. Como todo servidor de dados, para preservação dos dados é necessário redundância como meio de tolerância a falhas e também da implementação de uma política de *backup*.

Uma vez que os dados enviados pelo sistema embutido de monitoramento remoto formam, em alguns casos, partes de sinais analógicos, as aplicações no lado do servidor devem suportar uma forma de visualização ou interpretação dos dados que seja facilmente entendida por meio de gráficos ou tabelas, por exemplo. Em termos de velocidade de processamento é necessário capacidade de manipulação de grandes massas de dados que serão transferidas das interfaces de rede sem fio para os discos rígidos do servidor. Para permitir o acesso remoto dos dados pelos médicos ou pelos próprios pacientes é necessário conexão do servidor com a Internet através do meio sem fio e com fio.

Já o médico do paciente pode utilizar diversos meios para recebimento e envio dos dados para o paciente com algumas restrições. No caso de celulares por exemplo somente os de última geração capazes de suportar aplicações e possibilidade de apresentação dos dados podem ser utilizados. Além disto, os celulares devem estar habilitados para utilizarem a mesma ou compatível rede de distribuição sem fio que for utilizada pelo Sistema Embutido de Monitoramento Remoto e pelo Servidor Remoto de dados. Da mesma forma, caso o médico desejar utilizar o seu *PDA* é necessário que este esteja habilitado para comunicação com rede sem fio compatível com a utilizada pelos outros dois lados citados. O médico também pode utilizar seu computador pessoal ou *notebook* para acessar ou enviar os dados para o paciente. Para isto o computador deve estar habilitado ou para comunicação sem fio nos mesmos moldes anteriores ou permitir o acesso aos dados via *Internet* de maneira indireta através do servidor Remoto de Dados.

2.1.3 Requisitos do sistema de Distribuição sem Fio

O sistema de distribuição deve ser bem espalhado e com ampla cobertura para possibilitar a comunicação pelo menos entre distâncias do paciente até o raio de ação de seu médico ou do paciente até a Central médica Computadorizada mais próxima. Esta distância é variável podendo ir de centenas de metros até dezenas de quilômetros. A idéia é que quanto maior a cobertura melhor porque permitirá cobrir grandes áreas não ficando restrita a pequenas regiões possibilitando maior mobilidade entre todos os integrantes do sistema.

O Sistema de comunicação para o meio de distribuição sem Fio deve ser de forma a eliminar ou minimizar problemas de ruído, recepção errada de dados, além de possibilitar ou suportar a interface com dispositivos para acesso à Internet. Deve ainda possibilitar que a comunicação seja veloz em situações de envio e recepção de dados permitindo rapidez no diagnóstico, possibilitando a medicação de pacientes em estado de emergência.

2.2 Análise de algumas soluções para o projeto

2.2.1 O projeto existente : Monitor Cardíaco

O Monitor Cardíaco foi projetado com a finalidade de monitoramento de dados fisiológicos de seres humanos, auxiliando na detecção de problemas relativos a saúde e bem estar, mais especificamente, através de uma avaliação da condição cardíaca e de oximetria do paciente. Este projeto procurou otimizar o tamanho do hardware através da compactação de circuitos utilizados no tratamento dos sinais analógicos vindos dos sensores de medição de eletrocardiograma (ECG).

O Monitor Cardíaco portanto atende ao primeiro pré-requisito para o projeto que seria o monitoramento de pelo menos um dado fisiológico do paciente. Devido a integração do hardware, este sistema ficou com o peso reduzido que incluindo 3 pilhas de tamanho AA

ficou em torno de 170g. Com isto pode-se dizer que o Monitor Cardíaco também atende ao segundo pré requisito de projeto relacionado com o peso do produto.

Foi pensado, na primeira concepção do projeto, em se monitorar os dados remotamente utilizando-se uma rede de telefonia celular no padrão GSM. O formato da placa foi preparado para ser acoplado a um celular. Embora o sistema tenha sido preparado para suportar tal tipo de comunicação, não existem registros e relatos que comprovem a implementação de um sistema para comunicação remota sem fio no sistema em questão. Portanto, pode-se dizer que com as suas características originais, o sistema não atendia ao terceiro pré-requisito que é a comunicação sem fio.

Em relação ao consumo de energia, não foi feita também nenhuma análise do projeto em relação a este pré-requisito. Consequentemente, não é possível afirmar se o projeto atendia ou não tal pré-requisito.

2.2.1.1 Características do projeto Monitor Cardíaco

Para um melhor entendimento das características do projeto Monitor Cardíaco, o Diagrama em blocos dele é ilustrado na figura 2.1.

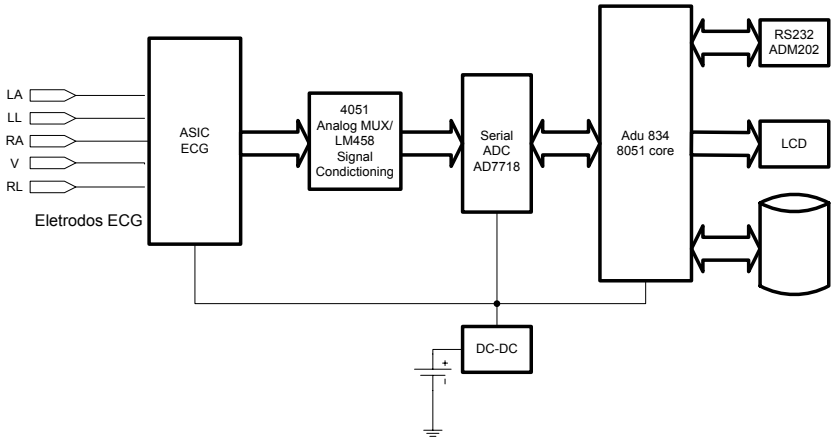


Figura 2.1: Diagrama em Blocos do Monitor Cardíaco

O componente responsável pelo controle das funcionalidades da placa é o microcontrolador de 8 bits ADu834[Dev03] do fabricante *Analog Devices*. Este microcontrolador possui um núcleo básico do popular 8051[Cor98] e é bastante versátil possuindo um conjunto de características bastante completo. Dentre elas pode-se destacar:

- 62Kbytes de Memória *Flash* de Programa;
- 4Kbytes de memória *Flash* de dados;
- 2304 bytes de memória RAM;

- 26 pinos de I/O programáveis;
- 11 fontes de interrupção;
- Portas seriais nos padrões UART*(Universal Asynchronous Receiver-Transmitter)*, SPI*(Serial Peripheral Interface)* e I2C*(Inter-Integrated Circuit)*;
- Temporizador *Watch-Dog*;
- Monitor de alimentação (PSM);
- Consumo da ordem de 2,3mA no modo normal e 20uA no modo de economia de energia;

Com o intuito de compactação do circuito foi utilizado um circuito integrado *ASIC (Application Specific Integrated Circuit)* ECG ASIC da Welch Allyn[Tec01] que implementa uma solução completa de ECG. Como componentes complementares foi utilizado um conversor Analógico-Digital (ADC) de baixo ruído para converter as amostras analógicas das saídas do *ASIC*, um conversor DC-DC compacto para geração das tensões de operação dos componentes, um *driver* padrão para comunicação RS-232 e uma unidade de memória de massa baseado na interface padrão *Compact Flash*.

2.2.2 O padrão *Compact Flash*

O padrão *Compact Flash*[Com03] ou *PC-card* foi introduzido em 1994, inicialmente para funcionar como um dispositivo armazenador de dados de acordo com o padrão *IDE (Integrated Drive Eletronic)*. Este modo de funcionamento iremos chamar de agora em diante de modo memória. O padrão *Compact Flash* é uma versão reduzida de 50 pinos do padrão da *PCMCIA (Personal Computer Memory Card International Association)* [PCMb] que possui 68 pinos. Além do modo memória, o padrão *Compact Flash* pode ser utilizado no que chamaremos de modo I/O onde dispositivos de entrada e saída tais como modems, cartões *Ethernet*, portas seriais, interfaces sem fio *Bluetooth* e *WI-FI* são implementados.

Os cartões *Compact Flash* são utilizados em dispositivos embutidos no modo memória para armazenamento de dados de câmeras fotográficas digitais por exemplo. Já no modo I/O, os cartões *Compact Flash* podem ser utilizados como interface de rede sem fio *WI-FI* ou *Bluetooth* em Assistentes Digitais Pessoais (*PDAs*).

A pinagem do padrão *Compact Flash* utilizada pelo Monitor Cardíaco é mostrada na tabela 2.1. Devido a limitação da quantidade de pinos de Entrada/Saída do microcontrolador do projeto Monitor Cardíaco, este foi configurado para interfacear com o padrão *Compact Flash* utilizando um barramento de oito bits embora o padrão suporte um barramento de 16 bits. Desta forma trataremos neste trabalho somente da configuração do padrão no modo 8 bits.

Desta maneira, uma possível alternativa de projeto utilizando-se o Monitor Cardíaco seria a utilização da interface *Compact Flash* no modo I/O com um cartão de interface de

Nome	Pino(s)	Função
A3:A0	17, 18, 19, 20	Linhas de Endereço
D7:D0	6, 5, 4, 3, 2, 1, 23, 22, 21	Linhas de dados (8 bits)
-CE1	7	habilitação do Cartão (8 bits)
-OE(RD)	9	Habilitação de leitura de memória Comum e de atributos do cartão.
-WR	36	Habilitação de escrita em memória Comum e de atributos do cartão
CD1	26	Detecta se cartão está inserido corretamente no conector Compact Flash

Tabela 2.1: Pinagem do padrão *Compact Flash* com pinos usados pelo projeto Monitor Cardíaco

rede sem fio *WI-FI*, por exemplo. Com isso, o hardware da interface sem fio seria facilmente integrado ao projeto, possibilitando com isto a manutenção do tamanho original do circuito.

2.2.3 A Pilha *TCP/IP*

A pilha de protocolos *TCP/IP(Transport Control Protocol/Internet protocol)* são os padrões mais usado na atualidade para a comunicação em redes de computadores. Eles ganharam dimensão mundial porque foram projetados para prover comunicação entre diferentes tipos de sistemas computacionais e cada um deles podendo rodar um sistema operacional distinto.

Eles surgiram, na década de 1960, através de pesquisas sobre redes comutadas por pacotes feitas *ARPA(Advanced Research Projects Agency)*, uma das agências de pesquisa e desenvolvimento do Departamento de defesa dos Estados Unidos. No inicio eram um padrão fechado mas com a abertura para utilização por universidades ganharam maior popularidade.

Os protocolos *TCP/IP* na verdade são o núcleo principal de protocolos que integram o que podemos chamar de arquitetura da Internet. Esta arquitetura em alguns casos é chamada de pilha *TCP/IP* por possuir os dois principais protocolos, *TCP* e *IP*, e outros divididos em camadas. Estas camadas situam-se uma em cima da outra como se estivessem empilhadas conforme figura 2.2. Neste texto iremos referenciar pilha *TCP/IP* como sendo o conjunto de protocolos que integram a arquitetura de uma rede Internet.

A figura 2.2 sugere que cada nível da pilha faça interação com o outro não sendo restrito

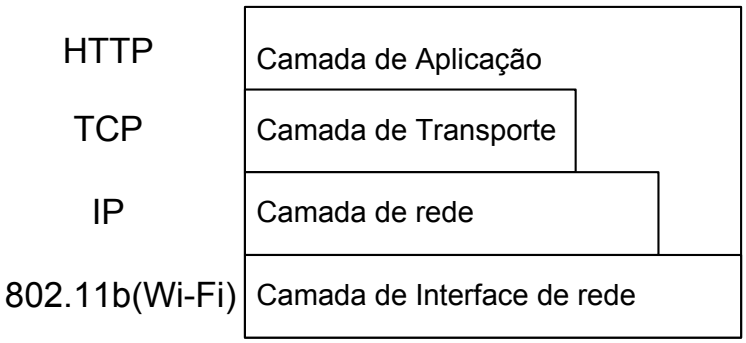


Figura 2.2: Pilha *TCP/IP* com alguns de seus protocolos

a cada protocolo. Por exemplo, as funções executadas pelo protocolo *TCP* podem ocasionalmente executar funções do protocolo *IP* e vice versa. Estas interações são importantes na otimização do código de programa.

Fazendo um breve resumo da função de cada camada pode-se citar:

- Camada de Aplicação: Fornece uma aplicação final para o usuário. No caso do *HTTP*, permite que páginas em *HTML* com dados sejam visualizadas em *Browsers*.
- Camada de Transporte: Identifica qual é o tipo de aplicação a qual os dados se destinam. No caso do *TCP*, implementa um mecanismo de transferência de dados confiável através de transmissão orientada por conexão, onde cada mensagem é identificada, verificada em relação a erros, reconhecida e caso necessário retransmitida para garantir confiabilidade na troca de informações.
- Camada de rede: Faz o gerenciamento dos dados dentro da rede orientada por pacotes. No caso do *IP* Implementa endereçamento e roteamento dos datagramas e permite que redes físicas diferentes sejam interligadas. Não implementa um mecanismo de transferência de dados confiável para as camadas superiores.
- Camada de Interface de rede: Como o nome sugere, faz a interface física entre os níveis superiores e nível de rede. No caso do protocolo *ARP* mapeia endereços de rede em endereços de internet. No caso do 802.11b implementa mecanismo de autenticação, associação e comunicação com outros nós da rede. Tudo isto feito sem o uso de fios *(Wireless)*.

Cada protocolo possui um formato de organização de dados semelhante ao outro sendo iniciado por um cabeçalho com campos variando de bits a 6 ou mais bytes seguido dos dados. Dependendo da camada, os dados de uma delas serão o cabeçalho e dados do nível superior conforme mostrado na figura 2.3.

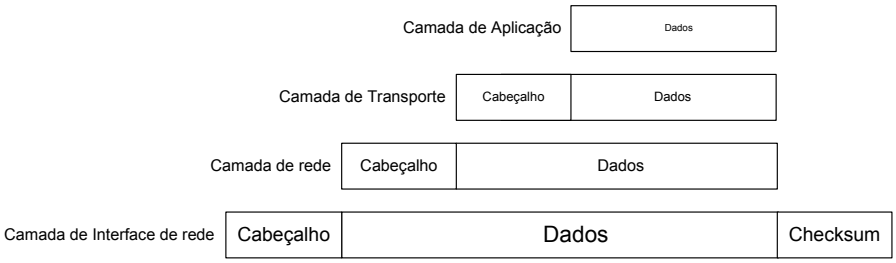


Figura 2.3: Estrutura dos dados em cada nível da pilha de protocolos *TCP/IP*

2.2.4 Protocolo de interface de rede 802.11 - *WI-FI*

O protocolo de interface de rede *IEEE 802.11* [IEE99] surgiu da necessidade de se conectar dispositivos à internet no modo *Wireless* de modo a possibilitar completa mobilidade a eles. Nos dias de hoje, a rede de distribuição dos pontos de acesso(*APs*) para acesso às redes sem fio 802.11 é bem diversificada podendo ser encontrada em restaurantes, redes de *Fast-Food*, livrarias, cafeterias, hotéis, aeroportos, shoppings, universidades, estádios de futebol e em muitos outros. A área de cobertura destas redes padrão 802.11 pode ir desde pequenos escritórios até uma cidade inteira como é o caso de Amsterdã que foi a primeira capital européia a ser coberta por uma rede sem fio[For05].

Uma tendência que favorece a expansão deste protocolo é a sua aceitação pela indústria mundial. Isto contribui para o desenvolvimento de novas tecnologias e produtos relacionados com ele. É o caso da maioria dos *notebooks* atualmente que já dispensam placas de rede sem fio e já embutem o hardware que implementa o protocolo diretamente. Periféricos tais como impressoras, *webcams* e câmeras digitais também estão aderindo a utilização do protocolo. Sistemas de som, televisores, aparelhos de *DVD*, telefone *IP* são apenas alguns eletroeletrônicos onde também é possível encontrar acesso pelo protocolo 802.11.

O protocolo 802.11 também é conhecido por *WI-FI*, ou *Wireless Fidelity*. Ele foi projetado, a princípio, para operar em três meios físicos diferentes sendo dois deles baseados em Espalhamento de frequência de rádio(*Spread Spectrum*) e o outro baseado em infra vermelho. Hoje em dia a tecnologia por espalhamento de freqüência de rádio usando seqüência direta (*Direct Sequence*) é a mais utilizada.

Com a utilização de uma pilha de protocolos no padrão *TCP/IP* que inclui o *WI-FI* como camada de interface de rede é possível a conexão do projeto à Internet. Portanto, o padrão 802.11 pode ser utilizado embutido em um hardware de um cartão *Compact Flash* no modo I/O para atender o pré requisito de conexão do Monitor Cardíaco à Internet e permitir que isto seja feito sem o uso de fios atendendo a outra necessidade de projeto.

2.2.4.1 Versões do Protocolo 802.11

O protocolo 802.11 é descrito como sendo uma combinação de um protocolo de Controle de Acesso ao Meio (*MAC*) e Controle da Camada de Ligação (*LLC*). Hoje em dia exis-

tem versões do protocolo 802.11 que são representadas principalmente pela versão 802.11b criada em 1999 e 802.11g criada em 2003. Existe também o padrão 802.11a criado em 1999 que comercialmente não foi bem aceito devido ao alcance limitado, falta de compatibilidade com o 802.11b e custo. Na versão 802.11b pode-se atingir velocidades em teoria de até 11Mbps e na 802.11g a especificação para a velocidade máxima de comunicação é de 54Mbps. Esta velocidade diminui com a distância conforme mostrado na figura 2.4 encontrada em [Cor03].

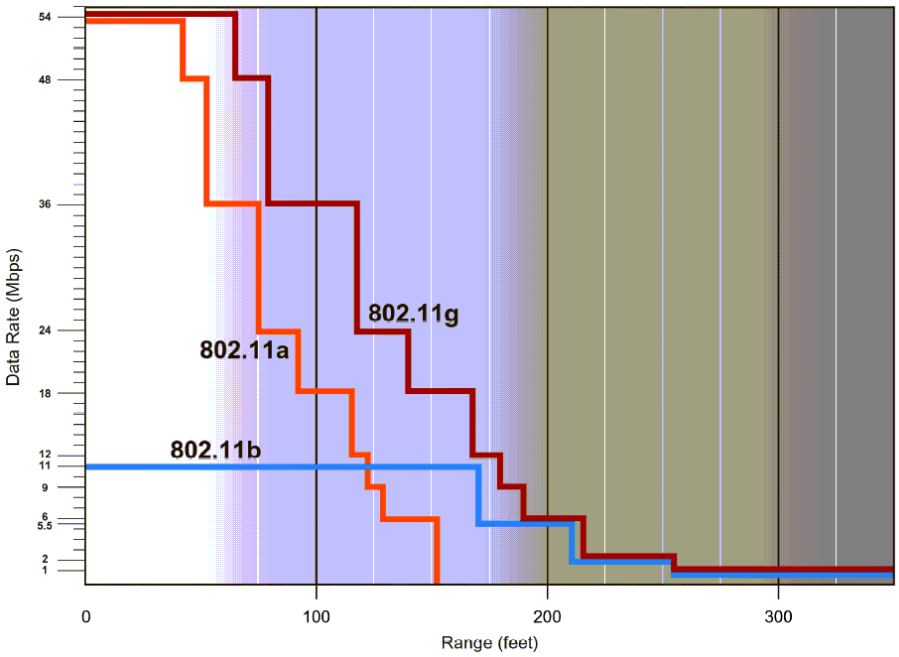


Figura 2.4: Taxas de comunicação dos padrões 802.11 em função da distância do ponto de acesso

Outra característica dos padrões são as freqüências de operação. O 802.11b e 802.11g operam na faixa de freqüência em torno dos 2.4GHz (banda S ISM) enquanto que o 802.11a opera em 5GHz. Estas e outras características podem ser visualizadas na tabela 2.2.

Embora o padrão 802.11g seja mais veloz, como o 802.11b surgiu primeiro, ele hoje possui maior representatividade que em 2003 era de 95%. Mais detalhes sobre os tipos de redes sem fio e sobre o padrão 802.11 podem ser encontrados em [Gas02].

2.2.5 Padrão de Comunicação *Bluetooth*

A idéia do padrão de comunicação *Bluetooth* surgiu em 1994 quando os engenheiros da divisão de comunicações móveis da *Ericsson* investigaram a viabilidade de conectar seus telefones celulares e acessórios por meio de uma interface de rádio de baixo custo e consumo.

Característica	**802.11b**	**802.11a**	**802.11g**
Aprovação do padrão	Julho 1999	Julho 1999	Junho 2003
Taxa de comunicação Máxima	11Mbps	54Mbps	54Mbps
Modulação	CCK	OFDM	OFDM e CCK
Taxa de dados	1, 2, 5,5, 11Mbps	6, 9, 12, 18, 24, 36, 48, 54 MBps	CCK: 1, 2, 5,5, 11Mbps OFDM: 6, 9, 12, 18, 24, 36, 48, 54 MBps
Frequências	2.4-2.497GHz	5.15-5.35GHz, 5.425-5.675GHz, 5.725-5.875GHz	2.4-2.497GHz

Tabela 2.2: Características das versões do Protocolo 802.11

Este conceito rapidamente evoluiu para a incorporaração de um rádio no telefone celular e outro em um computador pessoal de forma a conecta-los sem o uso de fios.

Com o conceito desenvolvido, a *Ericsson* aproximou-se de várias empresas de dispositivos eletrônicos portáteis para discutir o desenvolvimento do *Bluetooth*. Acabou conseguindo o apoio de outras cinco grandes formando o Grupo Especial de Interesse *(SIG - Special Interest Group)* para coordenar o desenvolvimento e promover a tecnologia *Bluetooth*. O *Bluetooth* foi formalmente anunciado em maio de 1998 e em julho deste mesmo ano saiu a versão 1.0 da especificação. Até o momento de publicação deste trabalho a especificação do *Bluetooth* se encontrava na versão 2.0 lançada em novembro de 2004 [SIG04].

O *Bluetooth* também usa a faixa *ISM* de 2,4GHz para se comunicar usando espalhamento por freqüência mas com a tecnologia de "Saltos de Freqüência" *(Frequency Hopping)*. Em relação ao alcance da comunicação foi previsto nas primeiras versões um raio de 10 metros mas atualmente na versão 2.0 já é possível atingir um raio de 100 metros. A velocidade de comunicação também aumentou com a seqüência de versões passando de cerca de 432Kbps *(Full Duplex)* na versão 1.0 para 3Mbps (modo *EDR - Enhanced Data Rate*) na versão 2.0.

Em relação a organização em redes, cada dispositivo *Bluetooth* pode se conectar com até outros 7 dispositivos formando uma *Piconet* onde o primeiro dispositivo a iniciar a rede e se conectar a outro assume o papel de mestre enquanto todos os outros serão os escravos. Quando várias *piconets* são interconectadas entre si através de seus dispositivos mestres é formado uma rede denominada *Scatternet*.

No que diz respeito à pilha de protocolos usada pela tecnologia *Bluetooth* pode-se ter

uma idéia do que precisa ser utilizado a partir da figura 2.5. Neste figura pode ser verificado a quantidade de padrões necessários para comunicar-se com a pilha de protocolos *TCP/IP* por exemplo.

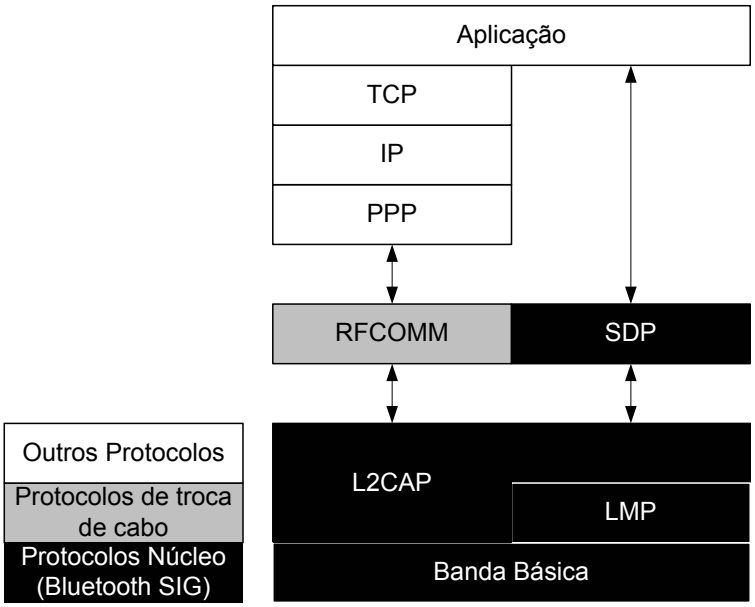


Figura 2.5: Pilha de protocolos usado pelo padrão *Bluetooth*

Assim como o protocolo 802.11, a tecnologia *Bluetooth* também pode ser encontrada na interface *Compact Flash* e desta forma também serve como possibilidade de ser implementada juntamente com o hardware do Monitor Cardíaco.

Mais informações sobre o padrão de Comunicação *Bluetooth* podem ser obtidas em [DK02, Mul00, Mil01]

2.2.6 Diferenças entre o padrão 802.11 e o *Bluetooth*

De acordo com o exposto nas seções anteriores pode-se fazer um pequeno paralelo entre as duas tecnologias de comunicação sem fio e apontar as principais diferenças entre elas. Estas principais diferenças são destacadas na tabela 2.3.

Em relação aos padrões em questão pode-se acrescentar que o *Bluetooth* necessita da implementação do protocolo *RFCOMM* e Ponto a Ponto *PPP* para se comunicar com uma aplicação em rede que utilize uma pilha do tipo *TCP/IP*. No protocolo 802.11 isto não é necessário sendo mais facilmente integrado à aplicação de rede devido a simplificação de software.

O que pode ser observado é que as duas tecnologias são complementares e atendem a propósitos específicos dependendo de onde serão aplicadas.

Característica	**802.11**	**Bluetooth**
Taxa de comunicação Máxima (Mbps)	54 (versão G)	3 (Versão 2.0)
Tecnologia de Espalhamento de Frequência	Sequência Direta (DSSS)	"Saltos de Frequência"(FHSS)
Alcance	até 300 metros	100 metros
Custo	Alto	Baixo
Uso Primário	Redes locais em corporações e Campus para transmissões de dados	Redes Ad-hoc e como eliminador de fios para dispositivos em curtas distâncias
Suporte a voz e telefonia	Sim	Não
Consumo de Energia	Alto	Baixo

Tabela 2.3: Diferenças entre o Protocolo 802.11 e *Bluetooth*

2.3 Decisões de Projeto

A primeira decisão foi utilizar como hardware para o monitoramento remoto de pacientes o Monitor Cardíaco. O principal motivo foi o fato de ser um circuito compacto que reune funcionalidade para monitoramento de sinais vitais de pacientes e também por ser um circuito com peso e tamanho razoáveis de serem carregados por um ser humano sem incomodar. Outros fatores também contribuiram para a escolha tais como ter sido projetado no DCC/UFMG permitindo continuidade de pesquisa, ferramentas de desenvolvimento facilmente disponíveis, etc. Com esta decisão de projeto os primeiros dois requisitos do sistema embutido de monitoramento remoto podem ser atendidos.

Com o circuito sistema embutido de monitoramento remoto já definido, não foi possível escolher muito em relação ao que usar como interface de rede sem fio. Um circuito que envolve comunicação sem fio possui um projeto de hardware complexo principalmente em relação ao desenvolvimento a Placa de Circuito Impresso e está fora do escopo desta dissertação. Portanto seria necessário a utilização de uma solução pronta. Como o projeto do Monitor Cardíaco possuía uma interface *Compact Flash* seria mais fácil usar uma alternativa de comunicação sem fio disponibilizada no modo IO desta interface.

A príncipio poderiam ser utilizados os dois padrões de comunicação sem fio discutidos anteriormente: 802.11 e *Bluetooth*. O padrão *Bluetooth* possui fortes características que proporcionam a integração de sistemas embutidos uns aos outros com menor custo e consumo de energia mas possui melhor desempenho na conexão em curta distância e não suporta distâncias maiores do que 100 metros. Além disso privilegia conexões do tipo

Ad-Hoc que não é o foco do sistema proposto que precisa de uma central de comunicação e centralização dos dados para distribuição dos dados pela Internet. A rede de distribuição do padrão *Bluetooth* ainda é restrita se limitando muito mais a interligação de equipamentos sem fio uns com os outros. Este requisito não é necessário no projeto uma vez que não é previsto que um Sistema embutido de Monitoramento Remoto com dados pessoais de um paciente se comunique outro.

A interface de rede sem fio 802.11 tem um consumo de energia mais elevado em relação ao *Bluetooth*. Este problema pode ser minimizado através de adoção de técnicas de gerenciamento de energia. Em um cartão *Compact Flash*, o circuito possui tamanho relativamente pequeno a um custo acessível para o projeto. As fortes características deste padrão são a velocidade de comunicação que é compatível com o padrão *Ethernet* e o fato de poder ser facilmente integrada à internet de modo sem fio, propósito o qual foi originalmente concebida. Em relação ao sistema de distribuição sem fio, a rede de cobertura se encontra em grande expansão através da implementação de Pontos de acesso em vários locais de uso comum. Isto permite que todos os requisitos do sistema de distribuição sem fio discutidos sejam atendidos. Todos estes fatores somado ao fato da disponibilidade de uso imediato de uma rede 802.11b existente no DCC/UFMG possibilitaram a escolha pelo padrão 802.11 para comunicação sem fio.

Para interligação do dispositivo à internet foi escolhido a pilha de protocolos *TCP/IP* por sua grande difusão, popularidade e adequação ao projeto conforme será visto no capítulo 3.

Os requisitos do servidor remoto de dados dependem muito mais dele mesmo do que o restante das necessidades do sistema de distribuição sem fio e do sistema embutido de monitoramento remoto. Como o padrão de rede sem fio 802.11 é popular, é possível encontrar facilmente um hardware adequado para integração do servidor ao padrão escolhido.

Através da escolha do hardware e do protocolo de comunicação sem fio foi necessário a adoção de técnicas para implementação da pilha *TCP/IP*, do *driver* de dispositivo sem fio e também para o gerenciamento de energia do sistema conforme será visto nos capítulos seguintes.

Capítulo 3

Trabalhos relacionados

O presente capítulo relaciona os principais trabalhos pesquisados relacionados com o tema da dissertação.

Três grandes áreas relacionadas com o trabalho foram pesquisadas para se verificar o estado da arte e a partir disso verificar os pontos em aberto, questões não tratadas, implementações futuras ainda não realizadas e outros de forma que pudessem ser tratados no presente trabalho. O objetivo com isto é enriquecer ou colaborar com a comunidade científica com informações relevantes sobre o tema pesquisado.

A primeira área que diz respeito ao trabalho são as aplicações de sistemas embutidos na área biomédica. Serão apresentados trabalhos que implementam sistemas semelhantes ou com a mesma idéia da plataforma original do projeto previamente escolhida. Esta área nos auxiliará a entender melhor alguns tipos de sistemas de monitoramento remoto que existem hoje na literatura e também a extrair informações que podem ser aproveitadas para a implementação do trabalho. Além disso, os trabalhos pesquisados serão analisados e na medida do possível comparados com a implementação proposta.

A outra área pesquisada está relacionada com as implementações de pilhas *TCP/IP* para sistemas embutidos. Em decisão de projeto feita no capítulo 2 optou-se por fazer a comunicação do sistema de monitoramento remoto sem fio através do padrão 802.11b e para poder fazer com que o sistema seja integrado à internet é necessário a implementação ou desenvolvimento de uma pilha *TCP/IP* para a plataforma de hardware escolhida. Serão analisadas as pilhas *TCP/IPs* existentes na literatura relacionadas com sistemas embutidos.

A última área pretende verificar técnicas utilizadas no gerenciamento de energia de sistemas embutidos. É necessário que a bateria que alimenta o sistema dure tempo suficiente para uma monitoração mais contínua do paciente. Para isto é necessário verificar quais são os métodos de hardware e software adotados para prolongar o uso de uma bateria.

3.1 Sistemas de monitoramento remoto para aplicações médicas

Este trabalho teve como base de pesquisa dois projetos desenvolvidos no DCC/UFMG. O primeiro deles foi o projeto monitor cardíaco [FLCJ03] descrito no capítulo 2. O segundo foi Monitor Paramétrico Vestível*(MMV)*[Con01] que utiliza o conceito de computadores vestíveis[Man98] e é capaz de fazer a supervisão do eletrocardiograma *ECG*, do nível de oxigenação do sangue (oximetria) e da pressão arterial não invasiva.

O *MMV* caracteriza-se por ser um sistema embutido que possui duas placas com processamento próprio, cada uma com o seu respectivo firmware. Uma delas é baseada em um microcontrolador de 8 bits e possui ainda 16 MBytes de *RAM* e 16 Mbytes de memória *Flash*. Ela executa a coleta, conversão e disponibilização dos sinais analógicos. A outra é baseada em um microprocessador AMD 586 133MHz de alto desempenho. Ela é controlada por uma versão do *Linux* conhecida como *Coyote* e é especializada no empacotamento e disponibilização das amostras digitais provenientes da primeira *CPU* para uma rede de computadores. Através da utilização do *linux* foi possível utilizar a pilha *TCP/IP* embutida neste sistema operacional. O projeto ainda possui uma interface de rede local sem fio através de um cartão *PCMCIA* no padrão *Wavelan*. Todos estes circuitos são alimentados por uma bateria recarregável com autonomia de 4 horas.

Através da exploração do conceito dos computadores vestíveis, a equipe de pesquisadores do *Media Lab* no *MIT* tem desenvolvido um conjunto de aplicações voltadas para a área médica denominadas de *Healthwear*[Pen04]. Como hardware básico para a maioria destas aplicações foi desenvolvido o MIThril 2003 [RD03]. Trata-se de um *PDA* Zaurus 5500 da *Sharp* que roda *Linux*. Este *PDA* possui uma porta serial RS232 que é ligado a um outro hardware denominado *Hoarder*[Ger03] que é baseada em um microcontrolador PIC 16F877 responsável por coletar informações de sensores capazes de monitorar o sinal *ECG*, transcondutância da pele (*Galvanic Skin Response - GSR*) e temperatura. O *Hoarder* possui uma interface Compact flash, operando apenas no modo memória, para armazenar os dados dos sensores e ainda um módulo de rádio FM half duplex e um relógio de tempo real (*Real Time Clock - RTC*). O PDA Zaurus do projeto MIThril 2003 possui uma interface *Compact Flash* capaz de interfacear com um cartão de rede sem fio no padrão 802.11.

O projeto *CodeBlue*[KL04, DM04] da Universidade de *Harvard* utiliza uma rede de sensores sem fios baseada no conhecido hardware *MICA*[xbo]. Este hardware fica acoplado ao paciente através de um sensor de eletrocardiograma[TRFFJW04] e os dados coletados por cada nó sem fio são enviados para um *PDA iPAQ* rodando *Windows CE* que recebe os sinais enviados e envia para uma central médica ou através de uma aplicação permitindo que as informações sejam analisadas pelo próprio PDA de um médico que recebeu os sinais da rede de sensores. Devido à limitada largura de banda e capacidade computacional, o *MICA* não pode ser utilizado com protocolos baseados em Internet tais como *TCP/IP* e *ARP*.

Mais recentemente, com o advento da programação em Java voltada para dispositivos móveis celulares, aplicações tem sido desenvolvidas para a área médica. Em [Fer05] é

descrito um exemplo de aplicação que usa um arquivo com dados adquiridos a partir de um ECG e estes são enviados para um servidor por meio de uma conexão *HTTP POST*. A grande dificuldade neste caso é que ainda as operadoras de telefonia ainda não estão completamente adequadas para suportar tais tipos de aplicação e além do mais, cada conjunto de dados enviado pelo paciente é cobrado pela operadora.

3.2 Pilhas *TCP/IP* para Sistemas Embutidos

A implementação de pilha *TCP/IP* mais conhecida, difundida e documentada atualmente é a BSD da Universidade de *Berkeley*. Ela possui código aberto e rapidamente ganhou popularidade na comunidade científica. Ela foi desenvolvida para computadores pessoais ou estações de trabalho que em sua maioria usam arquitetura 32 bits.

Existem várias pilhas *TCP/IPs* implementadas para sistemas embutidos. A maioria delas são variações do código original de *Berkeley*. Estas pilhas variam de acordo com a arquitetura utilizada e tamanho do código. Estes dois fatores permitem que funcionalidades do padrão sejam implementadas ou não.

Algumas pilhas usam um modelo simplificado e fazem suposições ou restrições relacionados com o ambiente onde o sistema será instalado. As suposições mais comuns são que o sistema embutido irá comunicar com um computador ou outro sistema que irá rodar uma pilha *TCP/IP* completa e de acordo com os padrões. Implementações que usam este modelo irão até comunicar de certa forma com outros sistemas nesta condição mas não irão comunicar ou irão ter a performance prejudicada quando comunicarem com sistemas semelhantes.

3.2.1 Simplificações utilizadas nas implementações

Uma das simplificações utilizadas no projeto de pilhas *TCP/IP* é definir apenas uma aplicação na camada de aplicação. Na maioria dos casos esta aplicação é um servidor *web*[Bra02]. Desta forma consegue-se reduzir o trabalho e complexidade a serem executados pela pilha implementando apenas o que é necessário para a aplicação funcionar. Não são implementados por exemplo suporte para dado urgente e não é necessário implementar abertura de conexão uma vez que não é necessário abrir conexão com outros servidores.

A implementação da aplicação tipo servidor *web* é muito utilizada em sistemas embutidos porque possibilita monitorar à distância o status de funcionamento do sistema ou verificar o estado de sensores acoplados aos microcontroladores.

As menores implementações conhecidas em relação ao tamanho do código e memória *RAM*, conhecidas são muito particularizadas para servidores *web* e são caracterizadas por não armazenar nenhum estado da conexão. Nestas implementações, retransmissões não podem ser feitas porque nada se sabe sobre o estado das conexões ativas. Para atingir transferências confiáveis, é necessário confiar no servidor remoto que é responsável pelas retransmissões. Outra simplificação destas implementações é não manter a conexão ativa

por muito tempo, ou seja, a partir de uma requisição, atende-la rapidamente e fechar a conexão. Dentre estas implementações pode-se citar o servidor *Web* baseado no microcontrolador PIC[Ben02],

Em relação a interface de rede implementada para estas pilhas *TCP/IPs* é comum aproveitar a *UART* existente nos microcontroladores e adotar o protocolo *SLIP(Serial in Line Internet Protocol)* para comunicação serial com outro servidor. Um exemplo de uma pilha que implementa o protocolo *SLIP* é a descrita em [Shr]. O protocolo *SLIP* é fácil de ser implementado porque apenas insere códigos simples de caracteres para sinalizar o limite de um bloco de dados.

Outro protocolo ao nível de rede que vem sendo bastante utilizado em conjunto com pilhas *TCP/IP* para sistemas embutidos é o *PPP(Point to Point Protocol)*[Req94]. Ele vem ganhando espaço porque alguns provedores de serviço previnem o acesso do protocolo *SLIP*. O protocolo *PPP* também é atrativo para os provedores de acesso porque implementa um protocolo de autenticação *(PAP)* com ID e senha além de outros relacionados com compressão *(CCP)*, por exemplo. Também tem a vantagem de monitorar a qualidade da linha. Por outro lado, por possuir uma extensa especificação com os vários protocolos descritos, a sua implementação é mais complexa e também o tamanho de código é maior quando comparado com o protocolo *SLIP*.

Em conjunto com o protocolo *SLIP*, algumas pilhas *TCP/IPs* implementam a emulação de um modem através de comandos AT. Esta implementação se faz necessária porque a maioria dos *PCs* utilizam um modem para comunicar com um provedor de serviços de Internet. Desta forma o dispositivo embutido deve ser capaz de responder a comandos de modem AT. O que se faz então é emular no dispositivo embutido fazendo com que a aplicação, como por exemplo um *Browser*, pense estar comunicando com um modem.

Apesar de a maioria das pilhas *TCP/IP* utilizarem como aplicação um servidor *Web*, existem alguns projetos que não seguem a mesma regra e implementam protocolos não muito convencionais tais como a pilha desenvolvida por [Loe99b] que utiliza *PPP* como protocolo de rede, *UDP* como protocolo de transporte e *TFTP* como protocolo de aplicação.

Outras aplicações tais como a pilha *TCP/IP* da *Atmel* [Ben03] não implementam certos mecanismos vitais do *TCP* com o objetivo de redução de código. Elas deixam de implementar mecanismos de controle de congestionamento que é usado para reduzir a taxa de envio de dados quando a rede está sobrecarregada. Não implementando esta funcionalidade, o dispositivo embutido funcionaria perfeitamente quando conectado a apenas um ponto de rede mas enfrentaria problemas se fosse conectado a mais servidores.

A pilha *TCP/IP* para o microcontrolador MSP430 da *Texas Instruments* [Dan04] usa outra simplificação onde é possível manipular apenas uma conexão em um dado momento. Esta simplificação contraria um dos princípios de funcionalidade do protocolo *TCP* que é prover conectividade a principio ilimitada para todos os nós de uma rede. Outra simplificações desta pilha incluem ausência de cálculo do *checksum* nos dados de entrada, falta de suporte para as opções *TCP* e segurança e tipo de serviço(TOS) *IP*.

Uma boa implementação de pilhas *TCP/IPs* é a da *Microchip* [Raj02] . Ela utiliza o conceito de Multitarefa cooperativa onde existe mais de uma tarefa, sendo que cada uma executa o seu trabalho e retorna o seu controle para que a próxima tarefa possa

executar o trabalho dela. Com isto a implementação da pilha não fica tão amarrada a aplicação o que pode ser interessante em casos onde muitas aplicações diferentes rodam ao mesmo tempo. Ela comporta de 2 a 253 conexões que são limitadas pelo compilador e pelo microcontrolador usado. Cada *socket* de conexão consome 36 bytes de *RAM*.

A pilha *CMX-Micronet*, desenvolvida pela empresa *CMX*, foi implementada para uso em microcontroladores com pequena quantidade de memória de dados e de programa. Ela suporta a maioria dos protocolos de Internet e ainda 127 soquetes *UDP* ou *TCP* [Ead04].

A tabela 3.1 mostra a quantidade de memória *RAM* de alguns das principais famílias de microcontroladores de 8 bits utilizados na atualidade:

Modelo	**Fabricante**	***RAM* (bytes)**
COP8SBR9	National	1024
PIC18F4515	Microchip	3968
MC68HC908AP64	Motorola/Freescale	2048
ATmega128	Atmel	4096
ADUc834	Analog Devices	2304

Tabela 3.1: Quantidade de memória *RAM* de alguns Microcontroladores de 8 bits

Percebe-se que em alguns microcontroladores fica quase impossível receber um segmento *Ethernet* (1500 bytes) ou um segmento 802.11(2314 bytes). Por isso algumas implementações descartam um segmento se o tamanho dele for maior que determinado limite. Isto funciona em casos onde o tamanho dos dados em certos tipos de aplicações não seja grande. De qualquer forma, quando receber os dados, quem enviou, na teoria, não saberá desta restrição e tentará retransmitir o dado novamente gerando um maior fluxo na rede. Outro problema também seria que alguns quadros poderiam não serem recebidos devido a esta limitação na recepção dos dados.

Outra restrição imposta por algumas implementações é não suportar fragmentação de pacotes *IP*. Novamente, se esta funcionalidade não é implementada corre-se o risco de caso algum pacote destinado ao dispositivo embutido estiver fragmentado, a aplicação não será capaz de interpreta-lo e com isso de forma semelhante ao parágrafo anterior pode causar aumento de tráfego por causa de retransmissões e em alguns casos até perda do pacote. Implementações *TCP/IP* que são capazes de reconstruir pacotes *IP* fragmentados tais como [kad] são grandes em termos de tamanho de código e necessidades de *RAM* que se tornam impraticáveis para sistemas de 8 bits.

Adam Dunkels foi talvez um dos pesquisadores que mais estudou sobre as pilhas *TCP/IPs* para arquiteturas de 8 bits[Dun03]. Ele implementou duas pilhas *TCP/IPs* com diferentes abordagens e direcionadas para sistemas diferenciados. A estas pilhas ele deu o nome de *lwIP (Lightweight IP)* e *uIP (micro IP)*.

A *lwIP* é uma implementação *TCP/IP* completa mas simplificada que inclui os protocolos *IP*, *ICMP*, *UDP* e *TCP* e é modular o suficiente para ser facilmente estendida

para outros protocolos. A *lwIP* suporta múltiplas interfaces de rede e tem opções de configuração flexíveis o que a torna compatível com a maioria dos dispositivos. O código desta implementação em linguagem C é todo estruturado, o que permite organização e fácil entendimento.

A implementação *uIP* foi projetada para ter apenas o conjunto mínimo de características necessárias para uma pilha *TCP/IP* completa. Ela pode manipular apenas uma interface de rede e não implementa *UDP*, ficando desta forma focada nos protocolos *IP*, *ICMP* e *TCP*.

As implementações de Dunkels procuraram seguir todas as necessidades da RFC1122 [Req89] que afetam a comunicação host para host (*host-to-host communication*), exceto certos mecanismos de interface entre a aplicação e a pilha tais como o report de erros e bits de tipo de serviço configurados dinamicamente no protocolo *TCP*. Uma vez que poucas aplicações fazem o uso destas características não implementadas, elas podem ser removidas sem comprometer o propósito da implementação.

As funcionalidades dos protocolos *TCP/IP* que ambas pilhas projetadas por Dumkels implementam são mostradas na tabela 3.2.

Característica	**uIP**	**lwIP**
Checksum IP e *TCP*	X	X
Fragmentação/Reconstrução IP	X	X
Opções IP		
Múltiplas Interfaces		X
UDP		X
Multiplas conexões *TCP*	X	X
Opções *TCP*	X	X
MSS variável *TCP*	X	X
Estimação de RTT	X	X
Controle de Fluxo *TCP*	X	X
Janela deslizante		X
Controle de Congestionamento	Não necessário	X
Dados *TCP* fora de sequência		X
Dado urgente *TCP*	X	X
Dado armazenado para retransmissão		X

Tabela 3.2: Funcionalidades Implementadas nas pilhas *TCP/IP* desenvolvidas em [Dun03]

3.2.2 Gerenciamento de memória

Devido as arquiteturas de 8 bits terem como fator limitante, quantidade de memória *RAM* conforme mostrado anteriormente na tabela 3.1 geralmente algumas soluções para gerenciamento da memória são adotadas.

A implementação *lwIP*, por exemplo, possui um *buffer* dinâmico e um mecanismo de alocação de memória onde a memória para armazenar o estado da conexão e os pacotes são dinamicamente alocados de um grupo de blocos de memória disponíveis. Pacotes são armazenados em um ou mais *buffers* de tamanho fixo alocados dinamicamente.

Na recepção dos dados, o *driver* do dispositivo de rede aloca *buffers* quando um pacote chega. Se o pacote recebido é maior que um *buffer*, mais *buffers* são alocados e o pacote é dividido entre eles.

Na transmissão dos dados, a aplicação passa o comprimento e um ponteiro do dado para a pilha e também um sinalizador dizendo se o dado é volátil ou não. A pilha *TCP/IP* então aloca *buffers* de tamanho apropriado ao tamanho e dependendo do sinalizador de volatilidade, copia o dado para os *buffers* ou referências ao dado através dos ponteiros. Os *buffers* alocados para a pilha podem alocar além dos dados, os cabeçalhos *TCP/IP* e camada de rede. Depois que os cabeçalhos são escritos, a pilha passa os *buffers* para o driver do dispositivo de rede. Os *buffers* não são desalocados quando o driver do dispositivo termina de enviar os dados ficando estes em uma fila de retransmissão. Se os dados são perdidos na rede e tem que ser retransmitidos, os *buffers* da fila de retransmissão são retransmitidos. Os *buffers* são desalocados quando os dados forem recebidos pelo outro ponto ou quando a conexão é abortada seja pela aplicação local ou remota.

A implementação *uIP* não usa alocação dinâmica de memória. Ela usa um único *buffer* global denominado *uip_buf* para armazenar os pacotes e tem uma tabela fixa para armazenar o estado da conexão. O *buffer* global de pacotes é grande o suficiente para armazenar um pacote de tamanho máximo.

Quando o pacote chega da rede, o driver do dispositivo o coloca no *buffer* global e chama a pilha *TCP/IP*. Uma vez que o *buffer* será sobrescrito pelo próximo pacote de entrada, a aplicação deve processar rapidamente o dado ou copia-lo para um *buffer* secundário para posterior processamento. O *buffer* de pacotes não será sobre escrito por novos pacotes antes que a aplicação tenha processado o dado. Pacotes que chegam quando a aplicação está processando o dado devem ser enfileiradas pelo dispositivo de rede ou pelo *driver* do dispositivo. Se o *buffer* global estiver cheio, o pacote recebido é descartado. Isto causa degradação de performance mas apenas quando múltiplas conexões estão rodando em paralelo. Isto é porque a implementação *uIP* anuncia uma pequena janela para receber os dados o que significa que apenas um único segmento segmento *TCP* estará na rede por conexão.

Na transmissão dos dados, o *buffer* global também é usado. Para enviar o dado, a aplicação passa um ponteiro para o dado e também o tamanho do dado para a pilha. Os cabeçalhos *TCP/IP* são escritos no *buffer* global e juntamente com o dado a ser enviado são transmitidos pela rede pelo *driver* do dispositivo. O dado não é enfileirado para retransmissão e neste caso a aplicação deve regerar o dado novamente se a retransmissão

for necessária.

A pilha criada por Jeremy Bentham[Ben02] restringiu ainda mais o uso de memória *RAM* da pilha *TCP/IP*. O microcontrolador utilizado por ele, o PIC 16C76, possuia pouca memória *RAM* e foi necessário decodificar os cabeçalhos *TCP/IP* em tempo de execução quando recebidos, e prepara-los também em tempo de execução quando transmitido. Para armazenar os dados são utilizados dois *buffers* sendo um de transmissão e outro de recepção. A vantagem desta abordagem é de que os *buffers* serem independentes, ou seja, dados de transmissão e recepção podem ser tratados diferentemente e não é necessário esperar receber um pacote de recepção para enviar um de transmissão. Em compensação eles acabam dividindo o espaço total disponível para *buffers* fazendo com que capacidade do *buffer* de entrada e saída sejam menores do que a de um *buffer* global. Consequentemente, para microcontroladores com pouca memória a capacidade para receber um pacote de dados de tamanho máximo fica consideravelmente limitada.

Seguindo a mesma idéia da solução anterior, a pilha da *Texas*[Dan04] utiliza 3 *buffers* sendo um *buffer* para recepção e dois *buffers* para transmissão. Dos dois *buffers* para transmissão, um deles armazena os dados a serem enviados e todos os cabeçalhos necessários *(TCP, IP e Ethernet)* e o outro além de armazenar o mesmo conteúdo anterior também armazena quadros dos protocolos *ARP* e *ICMP*. Embora possuindo 3 tipos de *buffers* apenas um *buffer* de transmissão e um de recepção podem ser mantidos ao mesmo tempo. Além disso, a pilha precisa esperar por um reconhecimento *(ACK)* do outro lado *TCP* antes de sobrescrever o conteúdo do *buffer* e permitir que novos dados sejam trocados.

Em alguns casos quando é possível, algumas implementações em microcontroladores utilizam a memória *RAM* de outros dispositivos tais como o *buffer* de dados do controlador de rede*(NIC)* para ajudar no armazenamento do cálculo do *checksum*, por exemplo.

3.2.3 Implementações de pilhas *TCP/IP* em Hardware

Algumas empresas implementam as soluções de uma pilha *TCP/IP* no próprio hardware. O primeiro *chip* que apareceu com esta característica foi o *Wiznet w3100A*[Can01, Can02]. Ele possui integrado uma pilha *TCP/IP* que suporta também *ICMP*, *UDP* e *ARP* além de 24KB de memória *RAM* e um controlador de acesso ao meio (MAC) para interface de rede *Ethernet*. Uma das vantagem deste tipo de abordagem é que a solução está pronta e na maioria das vezes fácil de ser utilizada pois algumas utilizam acesso às funcionalidades da pilha por meio de *API´s*.

Outra solução de pilha integrada ao hardware é o *chip* DS80C400[Max04]. Ele é um *chip* derivado da família 8051 que suporta uma pilha *TCP/IP* e um *MAC* para *Ethernet* 10/100. A pilha suporta 32 conexões *TCP* simultâneas e pode fazer transferências de até 5MBps através do *MAC Ethernet*. Além dos protocolos *TCP*, suporta o protocolo *IP* em suas duas versões(IPv4 e IPv6) e também *UDP*, *DHCP*, *ICMP* e *IGMP*.

Uma outra vantagem das pilhas *TCP/IPs* implementadas em hardware é que o trabalho de decodificação dos pacotes de dados fica para o *chip* que implementa a pilha enquanto que o microcontrolador fica somente encarregado de processar os dados ou implementar

protocolos para a aplicação final. Este é o caso da aplicação descrita em [RR00]. Ela utiliza um microcontrolador PIC 16F877 para processar os dados providos pela pilha implementada pelo *chip S-7600A* da *Seiko Instruments*. Ele integra uma pilha *TCP/IP*, 10Kbytes de *RAM*, interface com microcontrolador e uma *UART* em um único *chip*. Uma vez configurado, este *chip* age como se fosse um *buffer* de dados. Dados a serem transmitidos, até 1024 bytes, são armazenados em um *buffer* interno de *RAM* e a pilha *TCP/IP* acrescenta os cabeçalhos e *checksums*. Ela transmite o pacote pela *UART*. Quando o pacote é recebido, a pilha verifica se o endereço *IP* e a porta casam com o que foi configurado, calcula e verifica o *checksum* e transfere o conteúdo de dado do pacote para o *buffer*. Depois, o *chip* avisa ao microcontrolador que o pacote chegou e pode ser processado através de linhas de interrupção. A utilização de interface *UART* como meio físico sugere a conexão de um modem, que na aplicação é feita com uma solução integrada pelos *chips* Si2400/Si3015 da *Silicon Laboratories*.

As desvantagens de se ter as implementações da pilha em hardware é a falta de flexibilidade caso alguma mudança ocorra no padrão ou seja necessário a inclusão de certas funcionalidades. Outro ponto a se destacar é que pode-se ficar preso a um ou outro fabricante no projeto de um sistema embutido, uma vez que as implementações das pilhas *TCP/IPs* são particulares de cada fabricante de *chip*.

Uma previsão feita em [Can01] diz que assim como quase todo microcontrolador hoje em dia possui embutido uma *UART* é bem provável que no futuro, os *chips* já venham com uma pilha *TCP/IP* embutida. Isto porque assim como outros protocolos, a tendência é ter o projeto do protocolo *TCP/IP* congelado podendo com isso ser integrado em uma pastilha de silício.

3.2.4 Tamanho de Código

O tamanho dos códigos implementados pelas pilhas *TCP/IPs* dependem de vários fatores. Dentre estes fatores pode-se citar o compilador utilizado para geração do código. Outro fator é também a quantidade de protocolos utilizados. Algumas implementações não incluem o protocolo *UDP* ou o protocolo *ARP*.

A solução implementada pela *Texas Instruments*[Dan04] ocupa 4,2KB de memória de programa, 100bytes *EEPROM* (constantes) e 700 bytes *RAM*. A implementação de [Loe99a] não incluiu o protocolo *TCP* por causa da necessidade de *ROM* e *RAM*. Esta implementação gastou 145 bytes de *RAM* e 2170 palavras de *ROM*.

A implementação da *Microchip*[Raj02] mostrada na tabela 3.3 dá uma idéia do tamanho do código utilizado para alguns protocolos. Já a implementação de [Bra02] possui tamanho de código para alguns protocolos de acordo com a tabela 3.4.

A tabela 3.5 mostra o tamanho de código e quantidade de memória *RAM* usada pela implementação *uIP*[Dum03] O código foi compilado para um microcontrolador *Atmel* de 8 bits com arquitetura *AVR* usando o compilador *gcc* versão 3.3 com a opção de otimização de código ligada.

O total de memória *RAM* usada pode depender de quantas conexões *TCP* são alocadas,

Módulo	**Memória de Programa(KBytes)**	**Memória de Dados(Bytes)**
MAC	1,8	5(1)
SLIP	1,56	12(2)
ARP	0,78	0
IP	0,79	2
ICMP	0,64	0
TCP	6,6	42
HTTP	2,9	10
Servidor FTP	2,1	35

Tabela 3.3: Tamanho do código para cada implementação conforme [Raj02]

Módulo	**Memória de Programa(KBytes)**
MAC	1,0
ARP	2,5
TCP/IP	9,5
HTTP	3,8
UDP	1,4

Tabela 3.4: Tamanho do código para cada implementação conforme [Bra02]

quantas entradas são alocadas na tabela *ARP* e do tamanho do *buffer* de pacotes. Para a pilha *uIP* por exemplo, cada porta *TCP* em espera por conexão usa 2 bytes de *RAM* conforme mostrado na tabela 3.5. Um exemplo de configuração com 1 conexão *TCP* em estado de escuta, 10 conexões *TCP* em processamento, 10 entradas na tabela *ARP* e um tamanho de pacote de 400 bytes em uma aplicação simples de servidor *HTTP* irá ter as quantidades de memória mostradas na tabela 3.6 considerando o mesmo microcontrolador e compilador usados na tabela 3.5.

Para fazer uso de pouca quantidade de memória *RAM*, a pilha *uIP* também usa pouca memória alocada para chamadas de função que ficam armazenadas na pilha. Existe pouca profundidade de código entre a função principal *main* e as funções de aplicação. Além disto a maioria das funções da pilha *uIP* que são usadas pelos programas de aplicação são implementadas como macros da linguagem *C* e desta forma não usam memória alocada para pilha.

De acordo com o mostrado e também conforme estudado para outras pilhas *TCP/IPs*, o tamanho médio das implementações é em torno de 10 a 20Kbytes de *ROM* e 1Kbyte

Módulo	**Tamanho do código**	***RAM* estática**	***RAM* Dinâmica**
Buffer de pacotes	0	100-1500	0
IP/ICMP/*TCP*	3304	10	35
Conexões *TCP*	646	2	0
UDP	720	0	8
Web server	994	0	11
Checksums	636	0	0
ARP	1324	8	11
Total	7624	1520	65

Tabela 3.5: Tamanho do código e uso da *RAM* em bytes para a pilha uIP no microcontrolador *Atmel* com plataforma *AVR*

Módulo	**Tamanho do código**	***RAM***
ARP	1324	118
IP/ICMP/*TCP*	3304	360
HTTP	994	110
Checksum	636	0
Buffer de pacotes	0	400
Total	6258	400

Tabela 3.6: Exemplo de Tamanho do código(Kbytes) para a implementação uIP

de *RAM*. Desta forma microcontroladores de 8 bits com tamanho de memória *ROM* de 32Kbytes e 2K de *RAM* são suficientes para rodar o protocolo *TCP/IP*. Microcontroladores com mais capacidade de memória, logicamente, poderão rodar além da pilha, outras implementações de protocolos relacionados e também outras aplicações não relacionadas com a pilha tais como um sistema operacional, por exemplo.

Existem outras implementações de pilhas *TCP/IP* que são vendidas no mercado conforme mostrado na coletânea de soluções descrita em [Pea05] mas todas elas são implementadas para microcontroladores de 16 bits ou maiores o que foge ao escopo deste trabalho.

3.2.5 Implementações *TCP/IP* para redes sem fio 802.11 em Sistemas embutidos

O desenvolvimento de um circuito para comunicação pelo protocolo 802.11 é muito complexo. O uso de uma frequência de rádio na ordem de 2,4GHz requer atenção especial devido as preocupações com interferência, ruído, compatibilidade eletromagnética e outros. Além disto o projeto do circuito para o nível físico e nível de interface de rede precisa incorporar todas as características discutidas no capítulo anterior e outras que não foram detalhadas. O desenvolvimento de um protótipo de rádio para comunicação com o protocolo 802.11 fica inviável dentro das necessidades exigidas.

Para suprir esta carência dos circuitos, a indústria de semicondutores criou *chipsets* que agrupam circuitos que implementam ambos os níveis físico e de interface de rede. Estes *chipsets* tem sido utilizados pela indústria de fabricantes de suprimentos para redes sem fio 802.11 que desenvolvem os circuitos utilizando estes *chips* provendo interfaces de comunicação padrões para o mercado de computadores pessoais, *notebooks* e *PDAs*. As principais interfaces de comunicação para os adaptadores, produzidas pelos fabricantes de suprimentos de rede são nos padrões *PCI, USB, PCMCIA* e *Compact Flash*. Em [Tou04] são citados os principais fabricantes, tipos de chipsets e também fabricantes de equipamentos para redes 802.11. Segundo este artigo um dos principais *chipsets* utilizados pelos fabricantes é o *PRISM* [Inc01].

De forma a evitar o desenvolvimento complexo de um sistema de rádio compatível com o padrão 802.11b uma alternativa encontrada por alguns projetistas é a utilização de soluções baseadas em adaptadores de rede prontos. Pensando-se em quantidade de pinos alocados para a interface e lembrando que para um microcontrolador de oito bits isto deve ser plenamente levado em consideração a opção mais adequada neste caso seria a utilização da interface serial *USB* que necessita apenas de 2 pinos de I/O para comunicação. O problema da interface *USB* é que hoje quase não existem microcontroladores com interface *USB* embutida e por isso a interface requer mais um *chip* de interface o que foge do objetivo do trabalho onde o propósito é a utilização de apenas o microcontrolador sem grande modificação do hardware inicial para fazer o trabalho da pilha *TCP/IP* e controle da interface de rede.

Ainda pensando-se em quantidade de pinos, a interface com menos pinos depois da *USB* é a interface *Compact Flash* [Com03] que possui 50 pinos incluindo os pinos de alimentação. Através da implementação do modo de memória e de I/O, necessários para o funcionamento de interfaces de rede 802.11, pode-se implementar uma interface do microcontrolador com o padrão *Compact Flash* utilizando cerca de no mínimo 28 pinos. Com esta quantidade de pinos seriam necessários microcontroladores que geralmente possuem 4 portas de 8 bits ou 32 pinos.

Na implementação descrita em [Ead05] foi elaborado uma placa que comporta uma interface de rede sem fio 802.11b no padrão *Compact Flash* utilizando 28 pinos de um microcontrolador. Para este projeto foi utilizado um cartão *Compact Flash* com o *chipset PRISM*.

Outra aplicação interessante que também utiliza o *chipset PRISM* é a de um sistema que implementa um coletor de dados que armazena a radiação da luz solar[Cyl04]. Pelo propósito do projeto é necessário que o coletor solar fique em cima do teto de uma casa e portanto o projetista decidiu que o projeto fosse alimentado por luz solar. Para não ter a necessidade de ficar com fios saindo da janela e indo para o teto foi utilizado rede sem fio no padrão 802.11b para envio dos dados coletados para o computador do projetista.

O projeto do coletor solar teve como base um microcontrolador RCM3400 da *Rabbit* que possui uma pilha *TCP/IP* fornecida pelo fabricante e também foi implementado um *driver* de dispositivo para ser utilizado pela pilha. Como interface de rede sem fio foi utilizado um cartão *Compact Flash* no padrão 802.11b do fabricante *Linksys*. Devido ao alto consumo dos transmissores *WI-FI*, na camada de aplicação foi utilizado o protocolo *SMTP* para envio de *emails* com os dados coletados durante um dia.

3.3 O Gerenciamento de energia em sistemas embutidos

A utilização do protocolo de rede sem fio 802.11b através de um cartão *Compact Flash* para o desenvolvimento dos trabalhos com o sistema de monitoramento remoto consome cerca de 300mA quando está transmitindo [Cyl04]. Sendo assim é necessário a implementação de técnicas para gerenciamento de energia para redução do consumo de energia no projeto Monitor Cardíaco. Estudos mostram que com o gerenciamento de energia é possível economizar até 50% de energia [YHL00].

Na literatura muito tem-se pesquisado em relação a sistemas de gerenciamento de energia para sistemas embutidos principalmente porque a maioria deles opera com alimentação proveniente de baterias. Em relação ao hardware, cada vez mais tem sido pesquisados e testados combinações de elementos químicos que possam prolongar a vida útil de uma bateria. Uma procura grande por circuitos de baixo consumo também tem sido observada.

Como o circuito de monitoramento remoto se encontra com o hardware projetado o que nos resta a fazer em relação ao hardware são escolher baterias com boa vida útil no que diz respeito ao hardware e usufruir dos recursos e características de software do microcontrolador para se implementar um sistema de gerenciamento de energia.

Os sistemas de gerenciamento de energia podem ser classificados em estáticos e dinâmicos [YHL01]. Os sistemas estáticos de gerenciamento de energia são na maioria das vezes aplicados durante o desenvolvimento do sistema enquanto que os sistemas dinâmicos são aplicados em tempo real enquanto os sistemas estão com pouca carga de trabalho ou estão ociosos.

Os sistemas de gerenciamento de energia possuem diferentes estados de operação denominados de acordo com o seu consumo de energia. De maneira geral pode-se classificar os sistemas em relação ao seu consumo de energia de acordo com o seguinte:

1. **Ativo** - estado onde o consumo de energia é maior e o sistema opera a plena carga;

2. **Stand By** - o sistema opera com algumas funcionalidades principais tais como timers e interrupções mas não consome tanto. Possui funcionalidade limitada;

3. **Power Down ou modo Sleep** - o sistema consome pouca energia e geralmente possui somente o relógio de tempo real funcionando. Ele só é "acordado" por meio de interrupção ou reset por exemplo.

A maioria das técnicas estudadas dizem respeito ao tipo de política de gerenciamento de energia que é adotada. A política de gerenciamento é a lei de controle adotada pelo gerenciador de energia para decidir sobre o estado de operação do sistema [uBM99]. O próprio padrão 802.11 possui uma política de gerenciamento de energia embutida no protocolo. Este padrão requer que o ponto de acesso *AP* envie um sinalizador (*beacon*) a cada 100ms seguido de um mapa de indicação de tráfego (*TIM*). Cada cartão que desejar comunicar precisa ativamente escutar pelo sinalizador para saber se existe algum dado destinado a ele. Se não é necessário transmitir nem receber então o cartão pode ir para o estado de baixo consumo (*Doze*) até o próximo sinalizador.

Em [SVGM00] foram combinados novos algoritmos de gerenciamento e controle de energia para reduzir o consumo de um cartão de rede sem fio. O gerenciamento de energia reduz o consumo seletivamente colocando o cartão em estados de menor consumo quando o usuário não está ativamente comunicando com o cartão. O Controle de energia reduz o nível que o cartão transmite enquanto mantém a mesma performance. Desta forma este trabalho procurou reduzir o consumo de potência nos modos de operação ativo e ocioso do cartão.

Outra técnica também utilizada para reduzir o consumo dos dispositivos embutidos é a denominada de *DVS (Dynamic Voltage Scaling)* [SBA+01]. Esta técnica consiste em mudar a velocidade e a tensão de alimentação do processador durante o tempo de execução do programa de acordo com as necessidades da aplicação que está rodando. Para a implementação desta técnica é necessário que o processador ou microcontrolador suporte mudanças de velocidade e tensão de alimentação em tempo real. Em [IM02] são apresentados vários conceitos ligados ao gerenciamento dinâmico de energia incluindo a técnica *DVS* e também são citados alguns exemplos de aplicação baseados em diferentes políticas de gerenciamento.

Quando o tempo de transição entre os estados de operação é instantâneo, a política de gerenciamento de energia é trivial: os recursos devem ser desligados assim que se tornarem ociosos. Em alguns casos, a transição para o estado de baixo consumo é cara em termos de tempo e energia. No caso dos discos rígidos *(HDD's)*, por exemplo, transições do estado *sleep* para o estado ativo são um processo lento e de alto consumo uma vez que o disco precisa ser acelerado até uma alta velocidade. Quando estas transições são lentas, o problema da otimização da política de gerenciamento de energia torna-se não trivial e políticas efetivas que miniminizam o consumo sem comprometer a performance são necessárias.

Na aplicação desenvolvida em [Cyl04] foi implementado um circuito para desligar a alimentação do cartão *Compact Flash* de uma interface de rede sem fio 802.11b quando esta não estivesse sendo utilizada. A implementação do software otimizou a economia

de energia colocando o microcontrolador em modo *Sleep* somente permitindo a coleta de dados do conversor Analógico-Digital interno. Quando é desejado o envio de *email* com os dados, o software verifica se existe energia suficiente para a comunicação *WI-FI*. Caso exista, a alimentação da interface *Compact Flash* é ligada, o cartão é resetado e configurado para envio do *email*. Através de todas estas técnicas foi possível permitir que a aplicação conseguisse funcionar de maneira autônoma com a energia para alimentação do circuito proveniente apenas de paineis solares e bateria.

Capítulo 4

Implementação do Sistema de Monitoramento Remoto de Pacientes

Este capítulo apresenta as implementações que foram necessárias para que o hardware do Monitor Cardíaco fosse transformado para atender os pré requisitos de projeto. Estas transformações compreenderam algumas mudanças físicas no hardware do projeto. Em relação ao software foi desenvolvido um *driver* de dispositivo (*Device Driver*) para o cartão *Compact Flash* que implementa uma interface de rede sem fio no padrão 802.11b, uma pilha *TCP/IP* apropriada para o microcontrolador utilizado foi adaptada e aplicações para esta pilha foram também adaptadas para atender ao requisito de comunicação com a internet. Em relação ao gerenciamento de energia, técnicas de software são mostradas e implementadas para garantir que o sistema funcione com maior autonomia permitindo maior continuidade na coleta dos dados.

4.1 Interface com o cartão *Compact Flash* - Modo Memória

4.1.1 Leitura e escrita em um cartão *Compact Flash*

O trabalho de implementação da pilha *TCP/IP* com interface de rede sem fio originava na validação do projeto inicial em relação a interface *Compact Flash*. Era necessário saber se o circuito projetado era capaz de se comunicar com algum cartão no padrão *Compact Flash*. Desta forma, os trabalhos se concentraram inicialmente na familiarização com o padrão *Compact Flash*. Isto foi feito através de um estudo aprofundado sobre este padrão sobretudo no que diz respeito a leitura e escrita em um cartão *Compact Flash*.

Para a leitura e escrita no cartão *Compact Flash*, deve-se considerar os mapeamentos existentes para o tipo de dispositivo implementado no cartão. Basicamente, um cartão

Compact Flash possui os seguintes tipos de memória:

- Memória de atributos *(CIS)*;
- Memória Comum (Modo memória);
- *I/O* (Modo *I/O*).

Todo cartão *Compact Flash* possui uma estrutura de informação importante que fica armazenada nos primeiros bytes da chamada *memória de atributos*. Esta estrutura é denominada *CIS (Card Information Structure)* e possui dados que indicam qual é o tipo do cartão (memória ou *I/O*), dados do fabricante, quantidade de memória (modo memória) e para cartões no modo *I/O* com interface de rede sem fio indica por exemplo, qual é o endereço de acesso ao meio *MAC Address* do cartão. Esta estrutura tem 256 bytes de endereço.

A memória Comum existe para o caso de cartão no modo memória e o mapeamento de *I/O* é referente aos registradores existentes no cartão que implementam uma função de entrada e saída como é o caso do cartão de rede sem fio 802.11b.

O padrão *Compact Flash* suporta operação com um barramento de 8 ou 16 bits. Na tabela 4.1 simplificadamente é mostrado a configuração dos pinos necessária para operação em modo 8 bits para um cartão do tipo memória ou *I/O*. A temporização destes sinais é mostrada na figura 4.1. A temporização dos sinais para a leitura da estrutura de configuração *CIS* é a mesma do modo memória, porém o sinal *REG*, deve ser ativado em nível lógico zero ao invés de permanecer em nível lógico um.

Os tempos máximos e mínimos necessários para leitura e escrita foram omitidos na figura 4.1. Maiores informações sobre eles em ambos os modos podem ser obtidas em [Com03, Inc01].

4.1.2 Validação do projeto inicial - interface com o padrão *Compact Flash*

Após o entendimento do procedimento para leitura e escrita no padrão *Compact Flash*, decidiu-se validar o projeto Monitor Cardíaco para a leitura e escrita na memória *Compact Flash*. Para isto foram implementadas funções de leitura e escrita em um cartão de memória com capacidade de 8MB.

Como esta implementação ainda não tinha sido testada, no momento da tentativa de escrita detectou-se que o circuito da fonte da alimentação não suportava a corrente necessária para escrever no cartão de memória. Esta corrente elevada fazia com que a tensão de alimentação do circuito fosse interrompida provocando um *reset* no microcontrolador. Este problema foi inicialmente contornado ligando-se uma tensão de 5VDC direto na alimentação dos circuitos e desligando o circuito do conversor DC-DC da placa.

A implementação das funções de escrita e leitura no modo memória foram feitas com base no diagrama de sinais no tempo mostrados na figura 4.1 [Com03] para operações com

-CE2	-CE1	-REG	-OE	-WE	-IORD	-IOWR	Espaço selecionado
X	0	0	X	X	0	1	Leitura dos registradores de configuração (Modo IO)
X	0	0	0	1	X	X	Leitura dos registradores de configuração (Modo memória)
1	0	1	0	1	X	X	Leitura da memória comum (D7:D0)
X	0	0	1	0	1	0	Escrita nos registradores de configuração (Modo IO)
X	0	0	1	0	X	X	Escrita nos registradores de configuração (Modo memória)
1	0	1	1	1	X	X	Escrita na memória comum (D7:D0)
X	0	0	0	1	X	X	Leitura da estrutura de informação do cartão (CIS)
1	0	0	1	0	X	X	Acesso inválido (escrita na CIS)

Tabela 4.1: Tabela de configuração dos pinos do padrão *Compact Flash* para acesso aos diferentes modos para funcionamento em oito bits

o barramento de dados configurado para oito bits. Nesta figura também são mostrados os sinais *REG* e *CE2* que no projeto original estavam ambos fixos com os pinos em nível lógico um. Alguns pinos tais como *WAIT* e *INPACK* não foram considerados no projeto original principalmente pela escassez dos pinos do microcontrolador.

Foi implementada uma função de escrita de dados no modo memória através da função `CFWriteCommonMemByte`. É passado como parâmetro o endereço e o dado que se deseja escrever. Uma função de leitura foi implementada através da função `CFReadCommonMemByte` onde o endereço a ser lido é passado como parâmetro e o dado lido é retornado.

Ainda foram implementadas as seguintes funções para uma melhor utilização da interface no modo memória:

- `CFWait(unsigned char cf_stacom_mask, unsigned char cf_stacom_ok_mask)` - espera enquanto o sinal stacom_mask atingir o valor cf_stacom_ok_mask. Função usada na espera pelos bits de ocupado*(Busy)*, requisição de dados*(Data Request)* e pronto*(Ready)* do registrador `status` do padrão *Compact Flash*;
- `CFSendCommand(unsigned char cmd, unsigned long address,unsigned char nSectors)`

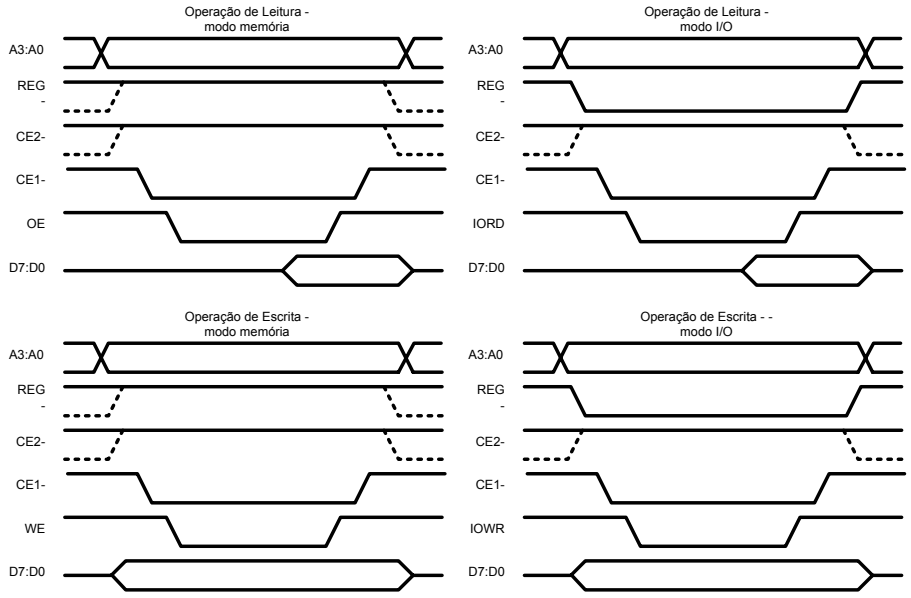


Figura 4.1: Diagrama de tempo para escrita e leitura no padrão *Compact Flash* no modo memória e *I/O*

- envia o comando `cmd` para a *Compact Flash*. No caso de comandos de escrita e leitura, envia também o endereço `address` e a quantidade de setores `nSectors`;

- `CFWriteSector(unsigned long address)` - escreve no endereço `address` um *buffer* com o tamanho do setor da memória *Compact Flash*;

- `CFReadSector(unsigned long address)` - Lê o setor localizado no endereço `address` e armazena os dados em um *buffer*.

Para a verificação do atendimento ao padrão de temporização do barramento e validação das funções escritas foi implementado inicialmente um pequeno programa que escrevia uma *string* na *Compact Flash* em setores diferentes e depois lê o que foi gravado nestes setores. O programa foi testado e funcionou corretamente. Foi possível verificar o resultado do teste através da impressão dos dados lidos da *Compact Flash* na porta serial do microcontrolador ADUc834. Posteriormente foi implementada uma aplicação que permitiu o armazenamento de dados provenientes da aplicação médica. Esta aplicação é explicada no item 4.10.

O primeiro desenvolvimento permitiu que parte do circuito fosse validada principalmente a questão da interface do microcontrolador com a memória *Compact Flash* em relação aos sinais de escrita, leitura, dados e endereços. Outro aspecto relevante que também foi validado foi a questão da temporização dos sinais nas operações de escrita e leitura para o modo memória mostrados na figura 4.1.

4.2 Familiarização com a Pilha de Protocolos *TCP/IP*

Após a validação do projeto inicial em relação ao padrão *Compact Flash* no modo memória partiu-se para uma familiarização com a pilha de protocolos *TCP/IP*. Como ainda não existia uma interface de nível físico sem fio implementada, foi necessário a utilização de outro tipo de interface.

Para uma implementação rápida do protocolo foi adaptado para o projeto Monitor Cardíaco a pilha *TCP/IP* descrita em [Ben02]. Através do livro que acompanha esta implementação foi possível um rápido entendimento das necessidades e limitações do projeto. Seguindo os moldes clássicos de aplicações implementadas em sistemas embutidos foi adaptado um servidor *web* que quando requisitado enviava uma página com a hora local no circuito obtida por meio de um circuito de Relógio de Tempo Real (*RTC*) embutido no próprio microcontrolador. Para a interface de rede foi utilizado a *USART* do microcontrolador ADUc834 com a implementação do protocolo *SLIP*.

Com pouca memória para armazenamento dos pacotes de dados, a implementação conforme descrita no capítulo 3 decodifica um pacote de dados em tempo de execução. Possui um *buffer* de dados de entrada `rx_buff[]` que é preenchido a partir de uma interrupção de chegada de dados na porta serial. Para envio dos dados, preenche um único *buffer* de saída `txbuff[]`. Para esta aplicação utilizando apenas a memória interna de 256 bytes do ADUc834, os tamanhos dos *buffers* de entrada e saída foram setados respectivamente para 80 e 96 bytes. Como o tamanho dos cabeçalhos *TCP* e IP padrões é de 40 bytes sobram no máximo 40 bytes no caso de recepção e 56 bytes de transmissão para os dados das aplicações. Isto é suficiente para atender e responder uma requisição *ICMP* do tipo *ping* de até 40 bytes.

O Controle de entrada e saída de dados é feito através de sinalizadores (*Flags*) que se estiverem setados ativam rotinas para processamento de envio e recebimento de dados. O *Flag* de recepção é ativado durante a rotina de processamento de interrupção de entrada de dados gerada pela *USART* do microcontrolador. Já o *Flag* de transmissão é ativado toda vez que é necessário o envio de dados pela porta serial, seja ele um comando de modem que emula a resposta de um modem até um pacote de resposta a uma requisição de dados no padrão *HTTP*.

De forma semelhante, o controle de utilização dos *buffers* de entrada e saída são feitos simplificadamente com o uso de duas variáveis para cada um. Uma variável de entrada é incrementada toda vez que o *buffer* de entrada ou saída é preenchido com um byte de dados. Uma variável de saída é incrementada toda vez que o *buffer* de entrada ou saída processa um byte de dados. A título de ilustração de parte de funcionamento do programa, fluxograma de controle do programa detalhando a parte de processamento dos dados do pacote *IP* é mostrado na figura 4.2.

A aplicação implementada funcionava como um servidor *web* apenas. Com isto muitas simplificações foram feitas. Uma delas é relacionada com uma limitação de memória do projeto. Como não utiliza memória para armazenamento de pacotes ela é capaz de atender apenas a uma requisição por vez. Outro fator de simplificação foi a aceitação de uma abertura passiva e desta forma a pilha *TCP/IP* não implementava todos os estados, somente os

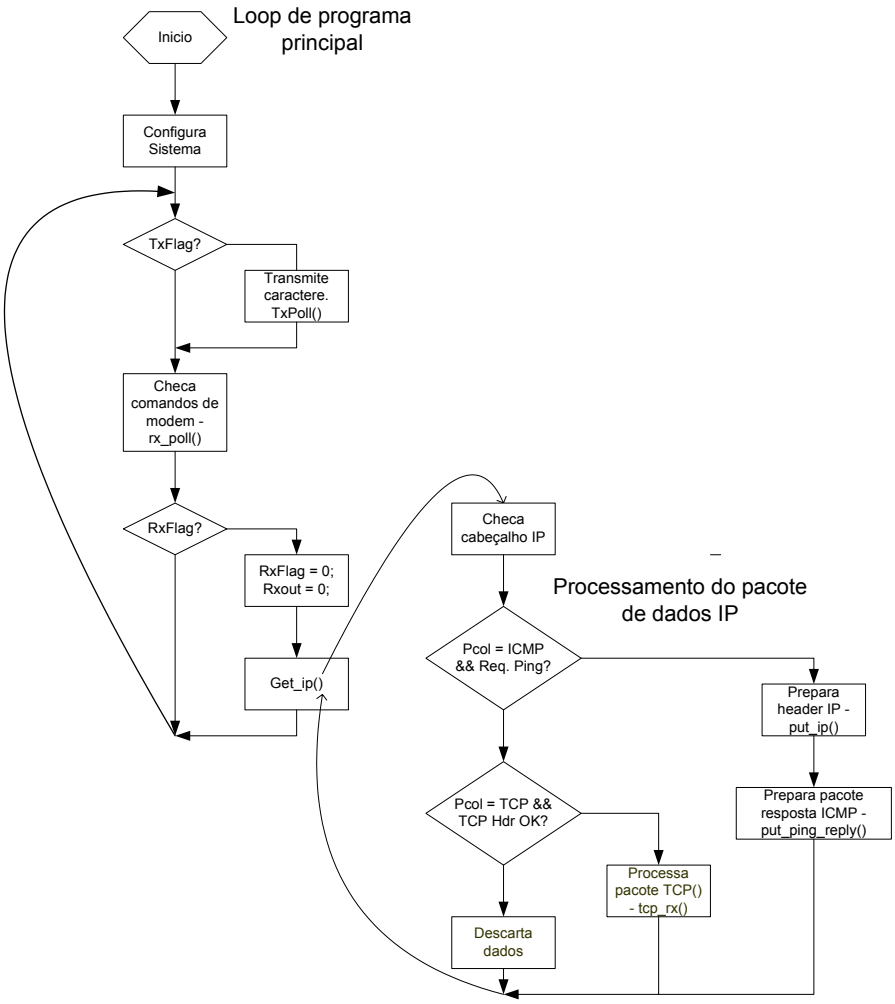


Figura 4.2: Fluxograma de parte do programa de controle da Pilha *TCP/IP*

referentes ao estado passivo. Desta forma não foi necessário armazenar nenhum endereço e nem número de porta, uma vez que era necessário apenas copia-los da mensagem de entrada para a mensagem de saída.

Devido ao fraco gerenciamento de conexão, retransmissões de dados não são possíveis. Para contornar este problema são implementadas transferências de dados curtas que cabem apenas em uma transferência de dados (*One segment TCP data transfer*). Este tipo de transferência espera pela abertura de conexão e uma vez estabelecida, os dados são enviados de uma vez e o servidor fecha a conexão não deixando-a ativa. Caso desejar que os dados variem é possível que apenas 56 bytes, de acordo com esta primeira implementação, sejam transmitidos o que torna a aplicação inviável, uma vez que é necessário o envio de uma quantidade maior de dados. A conexão sendo somente passiva impossibilita que o paciente envie dados remotamente quando desejar, não cumprindo com isto um dos requisitos iniciais

do projeto.

Somando-se aos outros fatores anteriormente citados, por causa da memória limitada, a aplicação *HTTP* teve que ficar fortemente acoplada à pilha *TCP/IP* de forma que a distinção entre elas ficou quase imperceptível o que tornou o programa mais difícil de ser entendido.

Uma das poucas vantagens desta implementação foi o tamanho de código para a pequena aplicação implementada. O código implementado ocupou 5,2Kb e pouco mais de 300 bytes de memória *RAM*, incluindo a memória estendida do microcontrolador ADUc834.

Até este momento cronológico do projeto, fora os pré requisitos atendidos com a escolha do hardware do Monitor Cardíaco, nenhum dos pré requisitos iniciais haviam sido cumpridos. As etapas realizadas até aqui serviram para solidificar o conhecimento e auxiliar no progresso com as etapas posteriores.

4.3 O Cartão *Compact Flash* 802.11b - *chipset PRISM*

A etapa anterior mostrou a possibilidade, embora com alguma limitação, de conectar o Monitor Cardíaco à internet por meio da pilha *TCP/IP* implementada. Como a implementação foi feita utilizando a porta serial como meio de comunicação era necessário implementar a interface de rede sem fio para satisfazer uma das necessidades de projeto.

Iniciou-se o desenvolvimento da interface de comunicação sem fio através da escolha do cartão *Compact Flash* que implementa uma interface de rede sem fio 802.11b. O modelo escolhido foi o *WCF12* do fabricante *Linksys* porque o DCC/UFMG já possuia tal cartão não havendo necessidade de compra. Uma vez escolhido o cartão, a próxima etapa foi descobrir que circuitos haviam dentro daquele cartão e o que era necessário implementar para possibilitar a comunicação com ele.

A pesquisa foi iniciada pela procura de *drivers* de dispositivo para a plataforma Linux. Existem grandes comunidades que desenvolvem vários *drivers* de dispositivo e muitas vezes é fácil achar os códigos prontos para a maioria dos dispositivos e quase sempre apoiados ou suportados pelo fabricante do Hardware. Foram encontradas comunidades [AS, pcma] de desenvolvedores com vários *drivers* de dispositivo desenvolvidos para o padrão *PCMCIA*.

Através de [Tou04] foi possível identificar que o fabricante *Linksys* utiliza em seus cartões *Compact Flash* e *PCMCIA* 802.11b, o *chipset PRISM* originalmente desenvolvido pelo fabricante *Harris/Intersil* onde passou por várias gerações de desenvolvimento e no momento de publicação desta dissertação se encontrava na versão 3 fabricada pela *Conexant*.

O *chipset PRISM* surgiu em 1996 como uma solução altamente integrada de cinco *chips* que implementavam um modem *RF* com tecnologia de espalhamento de frequência por Sequência Direta (*Direct Sequence Spread Spectrum - DSSS*). Hoje em dia na terceira geração devido a alta integração dos *chips*, o *chipset* foi reduzido a apenas dois *chips*. Um destes *chips* que fazem parte do *chipset PRISM*, o *HFA384x*[Inc01], implementa o nível físico *PHY* e de controle de Acesso ao meio *MAC*. O *HFA384x* opera com o barramento *Compact Flash* em 8 bits e em 16 bits.

Como o cartão *Compact Flash* de rede sem fio já estava implementado com todo o hardware necessário para o funcionamento do *chipset PRISM*, os detalhes em relação ao funcionamento interno deste circuito não são relevantes para o projeto e portanto estão fora do escopo do trabalho. O único fator relevante relacionado com o *chipset PRISM* é a interface *Compact Flash* implementada pelo *chip HFA384x* e o funcionamento deste *chip* para que a comunicação no padrão 802.11b seja implementada.

Através desta pesquisa inicial na procura pela forma de controle do cartão de rede sem fio pode-se ter uma idéia através dos *drivers* de dispositivos implementados no Linux do que seria necessário para a implementação da comunicação no projeto no Monitor Cardíaco. O primeiro obstáculo de se descobrir o que havia dentro do cartão *Compact Flash* e como se fazer a comunicação estava superado. Agora era necessário certificar se o cartão possuia tal *chipset* e posteriormente implementar um *driver* para este dispositivo.

4.4 Interface com o Cartão *Compact Flash* 802.11b

O *chip HFA384X* da família *PRISM* possui pinos compatíveis com a interface *Compact Flash* conforme mostrados a seguir. Nestes itens são indicados também as funções dos pinos e onde eles estavam originalmente conectados no projeto Monitor Cardíaco.

- `A9:A0`(I) - Linhas de endereço. `A9:A4` ligados originalmente a GND e `A3:A0` ligados ao ADUc834;
- `D7:D0`(I/O) - Linhas de dados. Todas ligadas originalmente ao ADUc834;
- `-CE1, -CE2`(I) - Habilitação do Cartão (16/8 bits). `-CE1` ligada ao ADUc834 e `-CE2` ligada a VCC
- `-OE(RD)`(I) - Habilitação de leitura de memória de atributos do cartão. Ligado ao ADUc834.
- `-WR`(I) - Habilitação de escrita de memória de atributos do cartão. Ligado ao ADUc834.
- `-IORD`(I) - Habilitação de leitura nos registradores de I/O do cartão. Ligado originalmente a VCC.
- `-IOWR`(I) - Habilitação de escrita nos registradores de I/O do cartão. Ligado a VCC.
- `-REG`(I) - Sinal de habilitação para leitura/escrita nos registradores do cartão no modo IO e de escrita na estrutura de informação do cartão (CIS). Ligado originalmente a VCC.
- `INPACK-`(O) - Sinal gerado pelo HFA384X para sinalizar que um ciclo de IO está em curso. Não conectado originalmente.

- `WAIT-`(O) - Gerado pelo HFA384X para sincronizar leitura e escrita com o microcontrolador. Não conectado.

- `IREQ-/READY`(*I/O*) - Saída para indicar que o cartão está pronto para operação após reset. Após reset serve como pino de interrupção. Não conectado.

- `RESET`(I) - Inicializa o cartão *Compact Flash*. Ligado a GND.

Através da comparação dos sinais de interface do *HFA384X* com os que existiam, percebeu-se que haviam vários sinais de controle necessários para interfacear com o *chip PHY/MAC* tais como *IORD*, *IOWR*, *RESET* que não estavam implementados originalmente no projeto Monitor Cardíaco. Além dos pinos de controle, mais tarde descobriu-se que também era necessário a alocação de mais pinos de endereço por causa do acesso a registradores de *I/O* do *HFA384X*. Não havia pinos disponíveis no microcontrolador ADUc834 que poderiam ser alocados para interfacear com os sinais necessários do *chip HFA384X*. O circuito de interface entre o ECG e o conversor AD já havia sido testado anteriormente pelo projetista e não era necessário testa-lo novamente.

Não foi encontrada uma outra solução, em tempo hábil, para o problema da falta de pinos para interfacear com o barramento *Compact Flash* do cartão. Em virtude disto, decidiu-se sacrificar alguns pinos do microcontrolador para a implementação da interface *Compact Flash*. Estes pinos primeiramente foram escolhidos de acordo com o critério de manutenção da funcionalidade original do circuito ECG. Mesmo assim, devido a necessidade inicial de uso de todos os pinos de endereços do microcontrolador, foi necessário sacrificar o restante do circuito comprometendo a funcionalidade do circuito ECG. Este problema é contornado através de solução proposta nos capítulos seguintes.

Procedeu-se então verificando-se que pinos teriam de ser realocados para serem ligados nos pinos do conector da *Compact Flash*(CF). Chegou-se na tabela 4.2. Nesta tabela também é indicada o nível de comprometimento da remoção do sinal para a funcionalidade do projeto. O pino *CE1-* foi colocado em nível GND, uma vez que o cartão *Compact Flash* 802.11b não possui memória comum.

Uma vez definidos os pinos a serem ligados, as trilhas do circuito impresso da placa montada do projeto Monitor Cardíaco foram cortadas e fios foram soldados ligando os pinos do microcontrolador no conector *Compact Flash*. O resultado da modificação do circuito pode ser observado na figura 4.3.

Pode ser observado na figura 4.3 que a tensão de alimentação do cartão *Compact Flash* (fio branco saindo para fora da placa) teve de ser alterada para 3,3V, uma vez que o cartão suporta somente tal tensão.

Sinal Original	Sinal CF alocado	Tipo sinal CF	Compromete funcionamento ECG?
CE1-	A4	Endereço	Não
CS_ADC	A5	Endereço	Sim
TXD1	A6	Endereço	Não
DOUT	A7	Endereço	Sim
CLK	A8-A9	Endereço	Sim
CS_DSP	REG	Controle	Não
BUZ	IORD	Controle	Não
LED1	IOWR	Controle	Não
CS_ASIC	RESET	Controle	Sim

Tabela 4.2: Sinais originais alocados para os pinos da *Compact Flash*

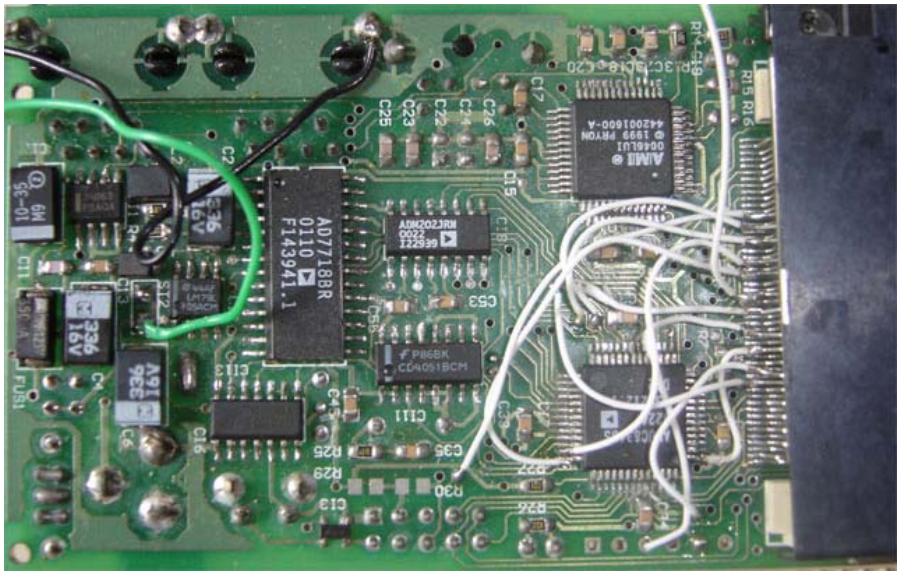


Figura 4.3: Circuito do Monitor Cardíaco modificado para operar junto com o cartão CF de rede sem fio 802.11b

4.5 Documentação para desenvolvimento do *driver* de Dispositivo de rede sem Fio 802.11b

Com a modificação do hardware do projeto era necessário valida-lo para verificar a geração dos sinais de controle e endereços para lerem ou escreverem os dados que se faziam

necessários para controle da interface de rede sem fio.

O *driver* para o cartão de rede sem fio desenvolvido para o sistema operacional Linux [AS] serviu como ponto de partida para saber que funções eram necessárias para implementação do *driver* para o projeto Monitor Cardíaco. Ainda assim era necessário algum documento que descrevesse melhor o funcionamento do *chipset PRISM* de forma que se pudesse desenvolver o software de controle.

Em pesquisas realizadas nos sítios de Internet dos fabricantes foi impossível descobrir alguma referência a alguma informação mais detalhada sobre o funcionamento do *chipset PRISM*. Depois de muita pesquisa foi descoberta a documentação para o desenvolvimento do *driver* para o dispositivo: um manual [Inc02] onde é explicado como usar os serviços fornecidos pelos *chips* de controle de acesso ao meio *HFA384X*. Tal documentação é fornecida pelo fabricante com acordo de não divulgação (*NDA - Non Disclosure Agreement*).

A seguir são fornecidas algumas características de funcionamento do *chip* de MAC *PRISM* para que fosse possível o entendimento das funções necessárias para controle do cartão CF de rede sem fio.

4.6 Características gerais do *MAC PRISM*

O Controlador de Acesso ao Meio *MAC* do *chipset PRISM* fornece um conjunto de serviços para o microcontrolador ou para o software do *driver* de dispositivo incluindo:

- Envio de pacotes de dados formatados em quadros 802.3 *(Ethernet)* e 802.11 *(Wi-Fi)*;
- Recebimento de pacotes de dados formatados em quadros 802.3 e 802.11;
- Reporta mudança de estado de conexão tais como associação, autenticação e disassociação;
- Configuração de vários parâmetros de operação de um adaptador de rede sem fio 802.11.

O microcontrolador pode operar o *MAC PRISM* através de um conjunto de comandos que são configurados através de um registrador de comandos. Para cada comando lançado, o MAC *PRISM* indica o resultado do comando através de eventos que são setados no registrador de status de eventos. A interface do *MAC PRISM* trava a interface de controle não deixando que outros comandos sejam executados enquanto um conjunto de bits de ocupado (*Busy bits*) associados com o registrador de comandos e através de outros comandos e registradores de parâmetros estiverem ativos.

O *MAC PRISM* é projetado para mover e processar dados em blocos que são chamados de *Buffers*. Estes buffers ficam localizados em uma memória de uso específico existente no cartão de interface rede sem fio. O microcontrolador acessa os *Buffers* indiretamente através de chamados Caminhos de Acesso para *Buffers*(*Buffer Access Paths - BAPs*) que

são registradores que permitem endereçamento, leitura e escrita em áreas de memória para os *Buffers* chamadas de Identificadores de Quadro (*Frame Identifiers- FIDs*).

O microcontrolador gerencia e configura os dados por meio do mesmo mecanismo dos *BAPs* usando valores especiais chamados de Identificadores de Recursos (*RIDs*) que podem ser lidos ou escritos em um *buffer* de uso especial. A escrita de um *RID* em um *BAP* copia a informação de configuração da memória do *MAC PRISM* para um *buffer* especial que pode ser acessado pelo microcontrolador para leitura ou escrita.

O *MAC PRISM* é implementado usando um controlador proprietário projetado para processamento de dados e execução do protocolo 802.11 através de um hardware dedicado para processamento de eventos e gerenciamento de memória. Cada máquina de estado do protocolo de MAC 802.11, máquina de estado para controle da camada física e a resposta aos comandos do microcontrolador é operado pelo *firmware* interno do *chip*. Este *firmware* pode ser configurado para operar tanto como uma *Estação* ou como um *Ponto de Acesso*. No caso do cartão *Compact Flash*, o *firmware* foi configurado para operar com o modo de operação *estação*.

4.7 Implementação de funções para controle do cartão de rede sem fio

Para a operação do *MAC PRISM* como cartão de estação de rede sem fio, são necessárias as seguintes principais tarefas:

1. Inicialização do cartão e do *firmware*;
2. Alocação de *buffers* para transmissão;
3. Uso da interface RID, registros de configuração e *frames* de informação para gerenciamento de operação do cartão;
4. Habilitação da recepção de dados e alocação de eventos de recepção;
5. Manipulação de eventos de transmissão.

Em resumo foram criadas três funções principais para controle do dispositivo de rede sem fio. A primeira função denominada `wifidev_init()` executa o que podemos chamar de inicialização geral do cartão englobando as três primeiras tarefas citadas acima além da habilitação da recepção de dados. A função `wifidev_read()` faz o tratamento dos dados recebidos e a função `wifidev_send()` manipula os dados a serem transmitidos pelo cartão.

4.7.1 Inicialização Geral do Cartão

A inicialização do hardware e *firmware* do cartão consiste em uma série de operações a serem executadas de modo a deixar o cartão pronto para funcionamento podendo ser capaz de receber e enviar pacotes de dados. Todo o processo de inicialização é implementado pela função `wifidev_init()`. As tarefas executadas por esta função são mostradas no fluxograma da figura 4.4.

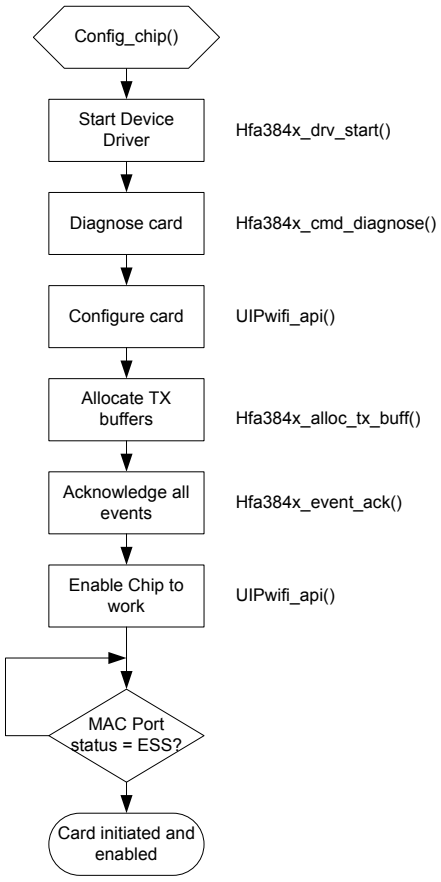


Figura 4.4: Sequência de tarefas executadas pela função wifidev_init() tendo ao lado qual função implementa cada bloco

A função `config_chip()` é uma configuração do microcontrolador apenas. Nela são configurados o *clock* do microcontrolador, configuração de pinos de *I/O*, configuração da *UART* para *debug* de programa através da porta serial, etc .

Toda a sequência de inicialização do cartão de rede sem fio descrita no bloco *Start Device Driver* mostrado na figura 4.4 foi incorporada na função `hfa384x_drvr_start()`. A descrição das tarefas executadas por esta função é descrita no fluxograma mostrado na figura 4.5.

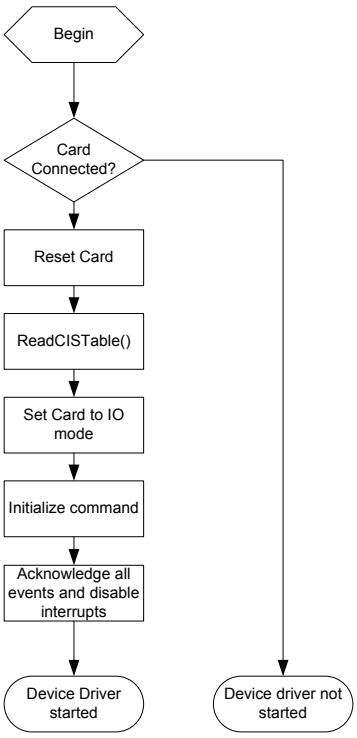


Figura 4.5: Sequência de tarefas executadas pela função hfa384x_drvr_start()

A primeira tarefa relacionada com o cartão de rede sem fio é iniciada através de um *reset* no cartão *Compact Flash* feito colocando-se o pino de *reset* do cartão CF em nível lógico alto esperando-se por quatro ciclos de NOPs[1] e em seguida retornando o pino para nível lógico baixo. Em seguida a estrutura *CIS* é lida através da função `ReadCISTable()` para identificação do *MAC Address* e do endereço do Registrador de Opção de Configuração (*COR*) do cartão. Uma vez descoberto qual o endereço do *COR*, configura-se o cartão para operar no modo *I/O* setando o bit 0 deste registrador. Convém enfatizar que devido ao endereço do *COR* possuir 10 bits foi necessário no retrabalho do circuito, considerar 9 pinos do microcontrolador para acesso a este registrador somente. Foram considerados 9 pinos pois os dois bits mais significativos são somente usados para acesso a este registrador e por possuirem o mesmo valor lógico foram curto circuitados.

Para a leitura da tabela *CIS* e escrita no registrador *COR* são utilizadas funções de leitura e escrita `CFReadMemByte()` e `CFWriteMemByte()` que foram implementadas e validadas seguindo o padrão dos sinais mostrados na figura 4.1. A interação com o cartão *Compact Flash* até esta etapa do programa foi apenas com a memória de atributos do cartão. Esta interação só se faz necessária durante esta primeira etapa de inicialização.

A segunda tarefa inicializa o cartão *Compact Flash* através de um comando de inicial-

[1] 1 NOP $\doteq$ 1us

ização. Este comando de inicialização é passado como parâmetro para a função `hfa384x_cmd()`. Através desta função é possível escrever alguns dos 12 comandos no *MAC PRISM*. A sequência necessária para a escrita de um comando no *MAC PRISM* é mostrada na figura 4.6.

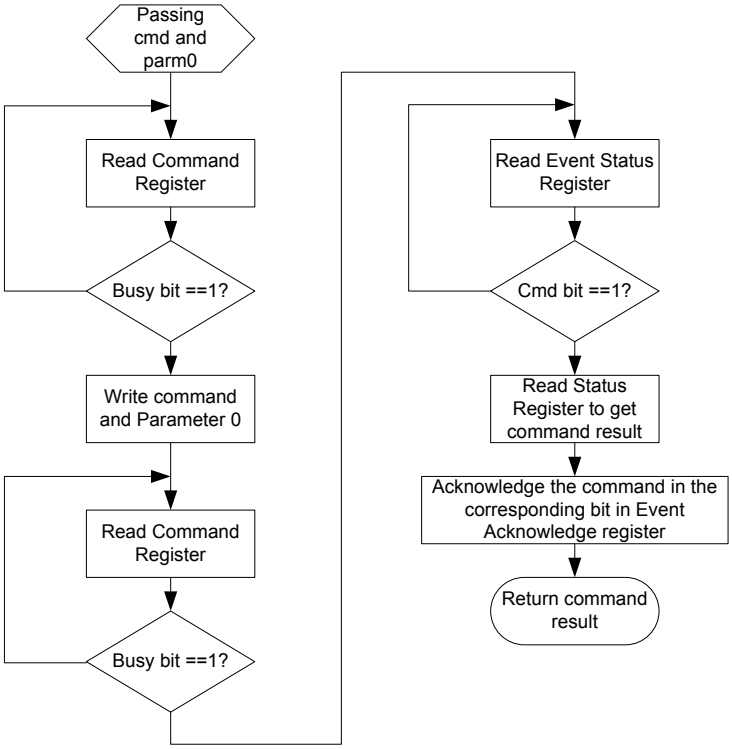


Figura 4.6: Sequência necessária para a escrita de um comando no *MAC PRISM*

Todos os comandos e registradores do *MAC PRISM* possuem formato de 16 bits. Como a interface *Compact Flash* com o microcontrolador é de oito bits foi necessário o desenvolvimento de funções para suportar este tamanho de barramento. Primeiramente foram desenvolvidas as funções `CFWriteIOByte()` e `CFReadIOByte()` para escrita e leitura em oito bits nos registradores de entrada e saída. Posteriormente foram criadas funções as funções `hfa384x_setreg()` e `hfa384x_getreg()` para escrita e leitura nos registradores do *MAC PRISM*. Foi utilizado para isto a chamada das funções anteriores de oito bits duas vezes passando como parâmetro no caso da escrita os bytes de registradores e valores a serem escritos e no caso da leitura os bytes do registrador a ser lido. A função de escrita nos registradores ficou então com o formato mostrado a seguir:

```
void hfa384x_setreg(int val, unsigned int reg)
{
    /* Write the low byte first */
```

```
    CFWriteIOByte((char)val&0xFF, reg);
    CFWriteIOByte((char)(val>>8), reg+1);
    return;
}
```

Seguindo a sequência de inicialização mostrada na figura 4.4, para certificar que o cartão estava funcionando internamente e também que os pinos da interface *Compact Flash* estavam operando normalmente foi utilizado um comando para diagnóstico do cartão. Para execução deste comando foi necessário a implementação de uma função especial denominada `hfa384x_cmd_diagnose()`. A sequência de comandos desta função se assemelha à função `hfa384x_cmd`, porém, ao invés de um parâmetro são passados dois. No final da execução do comando de diagnóstico, se o cartão estiver funcionando internamente é retornado um resultado de sucesso e caso contrário um código de falha. No final também se os pinos da interface estão funcionando corretamente, os dois parâmetros passados anteriormente podem ser lidos em registradores específicos. O teste com esta função ajudou a tirar uma duvida inicial de ter-se danificado o cartão por ter descoberto após alimenta-lo com a tensão de alimentação de 5V que o circuito suportaria somente 3,3V. Após o funcionamento deste comando tanto com o resultado de sucesso retornado quanto os valores lidos foi possível verificar que o cartão poderia ser utilizado sem problemas devido a boa parte do circuito ter sido validada.

Durante o processo de inicialização do cartão vários parâmetros da rede sem fio podem ser configurados. Entre eles pode-se citar:

- Modo de operação da rede sem fio: infra estrutura (*BSS* ou Independente (*IBSS*);
- Nome da rede que se deseja conectar (*SSID*);
- Tamanho máximo do pacote de dados a ser recebido pelo cartão de rede sem fio.

Para configuração destes parâmetros e outros foi desenvolvido uma pequena *API (Application Program Interface)* com o protótipo `uIPwifi_api(unsigned int cmd, unsigned int val)`. Com esta *API*, depois do comando de diagnóstico, a rede foi configurada para operar no modo infraestrutura. O nome da rede a se conectar foi setado de acordo com o teste a ser realizado. O comprimento de um pacote máximo de dados foi ajustado inicialmente para o tamanho do *buffer* interno do microcontrolador de recepção dos dados com tamanho de 1000 bytes.

A *API* desenvolvida utiliza um comando de configuração para alterar os parâmetros do cartão de interface de rede sem fio. Este comando foi implementado por meio da função `hfa384x_cmd_access()`. Através dela é possível ler ou modificar algum registro de configuração. Este registro de configuração possui um código identificador de 16 bits que é denominado de *RID (Resource Identificator)*. Durante o *reset* e inicialização do cartão os registros de configuração são preenchidos com os seus valores padrões. Os registros de configuração são organizados basicamente em dois grupos:

- Parâmetros de rede, entidades de configuração estática - configurados com o cartão em modo desabilitado. Influenciam o comportamento do processo de habilitação do cartão. Exemplos: Tipo de rede a se conectar, configuração do *SSID*, configuração do tamanho máximo de dados.

- Parâmetros de rede, entidades de configuração dinâmica - contem valores que imediatamente influenciam na execução de processos com o cartão em modo habilitado. Exemplos: Configuração do modo gerenciamento de energia, configuração do cartão em modo promíscuo.

A troca de informações entre o microcontrolador e o *MAC PRISM* é feita através de uma estrutura de registros que possui o formato mostrado na tabela 4.4. Esta estrutura de gravação é armazenada em um *buffer* temporário da *API* `uIPwifi_api` e passado como ponteiro para a variável `buf` da função `hfa384x_cmd_access`. De forma geral a *API* tem o seguinte formato:

```
void uIPwifi_api(unsigned int cmd, unsigned int val)
{
Declaração de variáveis


  Switch(cmd)
  {
    case valor_cmd:
      tmp_buff[0]   = tamanho da estrutura;
      tmp_buff[1]   = RID;
      tmp_buff[2]   = Dado 1 a ser configurado no RID;
      ...
      tmp_buff[2+n] = Dado n a ser configurado no RID;
      hfa384x_cmd_access(write, RID, tmp_buff, tamanho da estrutura);


      break;
    ...
    ...
  }
}
```

O acesso ao registro de configuração é feito através dos chamados Caminho de Acesso aos *Buffers* ou *BAPs(Buffer Access Paths)*. O *MAC PRISM* possui dois conjuntos de registradores *BAP0* e *BAP1* que são usados para acesso ao *buffer*. Para a leitura ou

Offset de palavras (Words)	Nome do campo	Tamanho(Palavras)
0	Tamanho do registro	1
1	RID	1
n	Dados(opcionais)	variável

Tabela 4.3: Estrutura de registros para configuração dos parâmetros do *MAC PRISM*

escrita dos dados nos *BAPs* foi desenvolvida a função `char hfa384x_rdwr_bap()`. Esta função é usada não só para modificação ou configuração dos parâmetros do cartão de rede sem fio mas também para a escrita e leitura de dados. A função para acesso aos *BAPs* é explicada com mais detalhes nas seções seguintes.

Posteriormente, ainda na etapa de inicialização do dispositivo são alocados *buffers* identificados por descritores de quadros. A alocação de *buffers* é necessária dentre outras coisas para transmissão de dados. Cada descritor de quadro possui um identificador denominado de *FIDs*. A alocação dos *buffers* de transmissão foi feita através da função `hfa384x_alloc_tx_buff()`. Foram alocados 3 *buffers* de transmissão que se mostrou ser um número suficiente uma vez que eles podem ser reaproveitados. Os *buffers* foram alocados utilizando-se outro comando do *MAC PRISM*, o comando *allocate*. Este comando inicia a alocação de *buffers*, que uma vez alocado é sinalizado através de um bit no registrador de eventos e o *FID* pode ser lido através de um outro registrador específico. O comando allocate é invocado através da função `hfa384x_cmd_allocate()`. O tamanho do *buffer* alocado foi de 1560 bytes que corresponde ao tamanho de uma pacote *Ethernet* somado aos cabeçalhos *Ethernet* e estrutura de quadro do *MAC PRISM*.

As etapas até agora descritas permitiram inicializar e configurar diversos parâmetros do cartão. Após este processo, o cartão está pronto para se conectar a uma rede sem fio 802.11b. Para se conectar à rede é utilizado o comando *Enable*. Este comando é implementado pela *API* através da sintaxe `uIPwifi_api(ENABLE, ON)` que chama a função `hfa384x_cmd` anteriormente descrita. Após este comando, todo o processo de associação, autenticação e outros implementados pelo protocolo 802.11b são executados em plano de fundo e o microcontrolador consegue saber se o processo de conexão foi bem sucedido lendo uma *RID* dinâmica que identifica o estado da conexão. A leitura desta *RID* dinâmica é feita até o *Timeout* de cerca de 30ms. Caso esta *RID* de conexão não fique com o valor esperado até o *Timeout*, o programa reporta um erro de conexão com o meio sem fio.

Após o processo de habilitação do dispositivo, ele está pronto para enviar e receber pacotes, tarefas que são descritas a seguir.

4.7.2 Quadros de Comunicação do *MAC PRISM*

Para entendimento da recepção e transmissão de dados, primeiro se faz necessário o conhecimento do formato da estrutura do quadro de comunicação utilizado pelo *MAC PRISM*.

Este quadro é mostrado na tabela 4.4 com os principais campos mostrados. Nela pode-se notar quatro segmentos distintos sendo eles controle, cabeçalho 802.11, cabeçalho 802.3 (*Ethernet*) e dados.

Offset(Words)	Estrutura do quadro TX	Segmento da estrutura	Estrutura do quadro RX
0	Status	Controle	Status
5			
6	Controle TX		
7	Controle Quadro	Cabeçalho 802.11	Controle Quadro
9	Endereço 1		Endereço 1
12	Endereço 2		Endereço 2
15	Endereço 3		Endereço 3
18	Controle de Sequência		Controle de Sequência
19	Endereço 4		Endereço 4
22	Tamanho dado		Tamanho dado
23	Endereço Destino	Cabeçalho 802.3	Endereço Destino
26	Endereço fonte		Endereço Fonte
29	Tamanho dado		Tamanho dado
30	Dados	Dados	Dados

Tabela 4.4: Formato dos Quadros de Comunicação do *MAC PRISM*

O segmento de controle possui campos que permitem controlar parâmetros de transmissão e obter informações sobre parâmetros de recepção. Na transmissão por exemplo, pode-se controlar a geração de sinalizadores para dados enviados ou não enviados com sucesso, setar qual dos cabeçalhos (802.3 ou 802.11) usar na transmissão dos dados, etc. Já na recepção pode-se obter informações sobre o nível de sinal recebido pelo modem RF

do cartão, taxa de recepção do quadro e outros.

Os segmentos de cabeçalho 802.3 e 802.11 utilizados pelo *MAC PRISM* são padrões exceto que no pacote 802.3, o campo de tipo de pacote recebido (*IP* ou *ARP*) é substituído pelo campo de tamanho de dados. O *MAC PRISM* só recebe *frames* de dados sendo que todos os outros *frames* de controle e gerenciamento são manipulados internamente pelas funções do *MAC PRISM*. Na transmissão, no cabeçalho 802.11, os campos de controle do *frame*, endereço 2 e sequência de controle são preenchidos automaticamente pelo *firmware* do *MAC PRISM*.

O campo de dados possui os dados recebidos ou a serem enviados pelo *MAC PRISM*. Dentro do campo de dados existe encapsulada uma outra estrutura denominada *SNAP(Sub-Network Access Protocol)*. Ela possui oito bytes e nos seus dois últimos bytes é implementado o campo de tipo de pacote de dados (*ARP* ou *IP*) que foi útil na montagem do cabeçalho original 802.3 na recepção dos dados. O campo de dados, incluindo a estrutura *SNAP* possui capacidade máxima de 2304 bytes caso o quadro não esteja encriptado ou a encriptação seja feita pelo *firmware* e 2312 bytes caso a encriptação seja feita pelo programa de controle.

Devido a facilidade de manipulação do cabeçalho *Ethernet* e sua fácil integração com a pilha *TCP/IP* utilizada, ele foi utilizado como cabeçalho para recebimento e transmissão dos dados. Na recepção, o endereço fonte do cabeçalho 802.3 é armazenado em uma tabela *ARP* para depois ser aproveitado no envio de volta para este mesmo endereço fonte. Na transmissão dos dados, o cabeçalho 802.3 é setado na estrutura de controle e o *MAC PRISM* gera automaticamente o cabeçalho 802.11 a partir do 802.3. Com isso não é preciso a manipulação dos endereços dos pontos de acesso intermediários para a conexão sem fio, somente o endereço fonte e destino. Estas simplificações facilitaram bastante o desenvolvimento das funções de transmissão e recepção dos dados.

4.7.3 Recepção de dados do cartão de rede sem fio

Para a recepção dos dados foi implementada a função `wifidev_read()`. Ela é responsável pela verificação de recebimento dos dados e armazenamento dos dados recebidos. A função retorna a quantidade de bytes recebidos durante o processo de recepção. A função de recepção dos dados utiliza duas variáveis que serão utilizadas pela pilha *TCP/IP* utilizada. Estas variáveis representam o *buffer* de dados recebidos, chamado de `uip_buff` e o tamanho deste *buffer*, `uip_len`. A sequência de tarefas executadas pela função de recepção é mostrada na figura 4.7.

O recebimento dos dados é sinalizado através de um evento de recepção, representando por um bit setado no registrador de eventos. Se este bit estiver ativo, a função procede com a recepção dos dados lendo um descritor de arquivos *FID* localizado em um registrador de descritores de arquivo de recepção. Depois a função `get_free_bap()` verifica qual dos dois Caminhos de Acesso ao *Buffer* (*BAPs*) está livre para recepção dos dados e em seguida o processo de acesso aos dados via *BAP* escolhida tem início. O descritor de arquivos é escrito no registrador de seleção do *BAP* escolhido e o deslocamento*(offset)* para recebimento dos

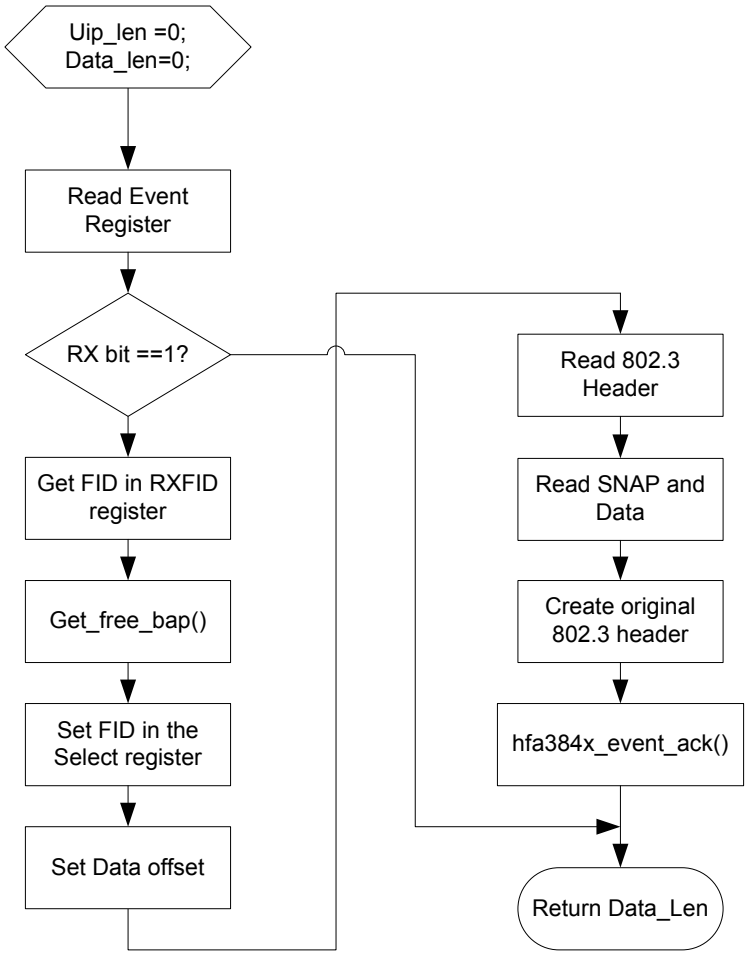


Figura 4.7: Tarefas executadas pela função de recepção wifidev_read

dados é setado.

O *offset* foi configurado para começar no último campo do cabeçalho 802.11 que de acordo com a tabela 4.4 começa no byte 44. A partir dele o tamanho do quadro de dados é lido e armazenado na variável `data_len`. A leitura dos dados é feita através do registrador de dados referente ao *BAP* escolhido. A cada leitura feita neste registrador, um ponteiro interno ao *MAC PRISM* é incrementado. O *buffer* `uip_buf[]` é preenchido a partir do cabeçalho 802.3 com o endereço *MAC* fonte, endereço *MAC* destino, tamanho do pacote novamente, *SNAP* e dados.

Para que o cabeçalho 802.3 fosse utilizado na forma padrão, o campo de tamanho de dados lido e armazenado no *buffer* `uip_buf[]` foi substituído pelo campo de tipo de pacote de dados, existente no protocolo *SNAP*. Esta adaptação permitiu que a pilha *TCP/IP* utilizada não precisasse de muitas modificações uma vez que tinha sido projetada originalmente para ter o protocolo *Ethernet* como nível de interface de rede.

Para que na próxima chamada da função `wifidev_read`, o bit de recepção de dados do registrador de eventos seja interpretado corretamente é necessário que o correspondente bit no registrador de reconhecimento seja reconhecido. Isto é válido para a ocorrência de qualquer evento ocorrido no *MAC PRISM* que esteja associado com o registrador de eventos. Além do evento de recepção, o registrador de eventos é capaz de registrar os seguintes eventos relacionados com o *driver* de dispositivo desenvolvido:

- Comando completo - sinalizado toda vez que a execução de um comando é completada pelo *MAC PRISM*;
- Alocação de *buffer* - sinalizado quando um *buffer* de estrutura de quadro é alocado ou solicitado novamente para transmissão de dados;
- Erro de Transmissão - sinalizado toda vez que existe um erro de transmissão;
- Transmissão - sinalizado toda vez que que os dados são transmitidos com sucesso pelo *MAC PRISM*.

Para cada um dos sinalizadores acima existe um bit no registrador de reconhecimento que deve ser setado toda vez que um dos eventos ocorre. Logo em seguida o *MAC PRISM* se encarrega automaticamente de resetar internamente os bits de reconhecimento. Para o reconhecimento da recepção de dados bem como outros eventos que por ventura vierem a ser gerados foi criado a função `hfa384x_event_ack()` que verifica os bits setados no registrador de eventos e seta o bit correspondente no registrador de reconhecimento.

A função de recepção de dados `wifidev_read` retorna *zero* caso não seja gerado um evento de recepção de dados e *um* caso o tamanho do pacote recebido seja maior do que zero.

4.7.4 Transmissão de dados no cartão de rede sem fio

Para a transmissão dos dados foi elaborada a função `wifidev_send()`. Ela é responsável, além de transmitir os dados, por executar algumas tarefas para que a operação de transmissão seja bem sucedida. Esta sequência de tarefas é mostrada na figura 4.8.

Na recepção dos dados foi necessário transformar o cabeçalho 802.3 para o formato padrão. Na transmissão dos dados é necessário fazer o oposto, ou seja, transformar o cabeçalho 802.3 padrão criado pela pilha *TCP/IP* no cabeçalho 802.3 usado pelo *MAC PRISM*. Isto é possível copiando o tamanho do pacote de dados armazenado na variável `uip_len` para as posições do *buffer* `uip_buf` referentes ao tamanho do pacote.

Posteriormente, um dos três descritores de arquivo alocados no processo de inicialização do dispositivo são usados para copiar os dados de transmissão do *buffer* `uip_buf` para o *BAP* através da função `hfa384x_rdwr_bap()`. Através de um *FID*, os dados do *buffer* de transmissão são passados para a função através de um ponteiro e ela os escreve em um dos *BAPs* do *MAC PRISM*. O funcionamento da função de leitura e escrita nos *BAPs*

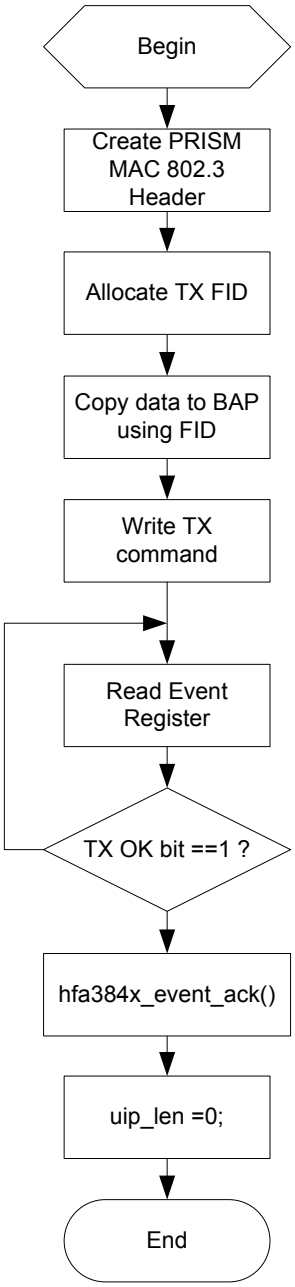


Figura 4.8: Tarefas executadas pela função de transmissão wifidev_send

segue o mesmo processo descrito para a recepção porém, para a transmissão, os dados serão escritos nos $BAPs$ ao invés de serem lidos.

Depois que os dados são copiados para o *MAC PRISM*, a função de transmissão envia um comando de transmissão para ele e em seguida aguarda pelo sucesso do comando que

é indicado por um bit no registrador de eventos. A função de transmissão aguarda pelo sucesso do comando de transmissão até um tempo máximo de cerca de 10 microsegundos. Caso o bit do registrador de eventos esteja setado dentro do *Timeout* do reconhecimento do comando, o evento de transmissão é então reconhecido pela função `hfa384x_event_ack()`. Caso contrário, se o tempo expirar a função reporta um erro de falha no comando de transmissão. Se o comando tiver sido executado com sucesso, o tamanho do *buffer* é resetado para que o *buffer* global possa ser utilizado corretamente pela pilha *TCP/IP* a qual descrevemos a seguir.

4.8 A Pilha *TCP/IP uIP*

Dentre todas as pilhas *TCP/IPs* apresentadas no capítulo 3 a que se mostrou mais apropriada para o projeto foi a pilha *TCP/IP uIP* [Dum03]. Dentre as várias razões para se ter escolhido a pilha *uIP* pode-se destacar primeiramente o fato de ela já ter sido portada para vários microcontroladores e compiladores, inclusive para a plataforma 8051 e Keil51, mostrando compatibilidade com o projeto. Outro fato relevante foi o tamanho do código que também se mostrou bastante adequado perante as necessidades de projeto.

A implementação *uIP* escolhida foi a utilizada em [mur]. Esta implementação além de portar a pilha *uIP* para o microcontrolador AT89C51AC2(família 8051) da Atmel, foi adaptada para o compilador C Keil51. Como interface de rede ela foi originalmente implementada com um *device* driver para o *chip MAC Ethernet* RTL8019AS da *Realtek*. Para o nível de aplicação foi desenvolvido um mini servidor *web* que apresenta algumas estatísticas da pilha *TCP/IP* tais como número de pacotes de cada protocolo aceito, número de pacotes perdidos, estados das conexões, etc. A parte reaproveitada deste código foi a referente à pilha *TCP/IP* e em um primeiro momento a aplicação do mini servidor *web* para teste da implementação dos protocolos.

A idéia geral de funcionamento da pilha *TCP/IP* em relação ao projeto proposto pode ser observado na figura 4.9. Ela pode ser entendida como o ponto central de conexão entre o *Device Driver* e a aplicação. Na recepção dos dados feita pela função `wifi_read()`, o *buffer* de dados `uip_buf` é preenchido com os dados e o tamanho dos dados `uip_len` sendo maior do que zero sinaliza para a pilha *TCP/IP* que existem dados para serem processador. Os dados recebidos são analisados pelas funções da pilha *TCP/IP* e se os cabeçalhos estiverem corretos, os dados são repassados para a aplicação. Durante o processo de conexão e desconexão, a pilha *TCP/IP* implementa toda a máquina de estados *TCP* e outros mecanismos internos a este protocolo para que a conexão seja confiável de modo a garantir a entrega correta dos dados. No envio dos dados, a aplicação passa para a pilha *TCP/IP* os dados a serem enviados e ela se encarrega de preencher os cabeçalhos para os protocolos. Todos os dados de transmissão são colocados no *buffer* `uip_buf` e a variável `uip_len` é configurada com o tamanho do pacote a ser enviado. A partir dai a função `wifi_send()` fica encarregada de enviar estes dados para o *MAC PRISM*.

Não é objetivo do trabalho detalhar o funcionamento da pilha *uIP* porque ela não foi desenvolvida neste trabalho. Ela apenas foi portada e reutilizada para o projeto pro-

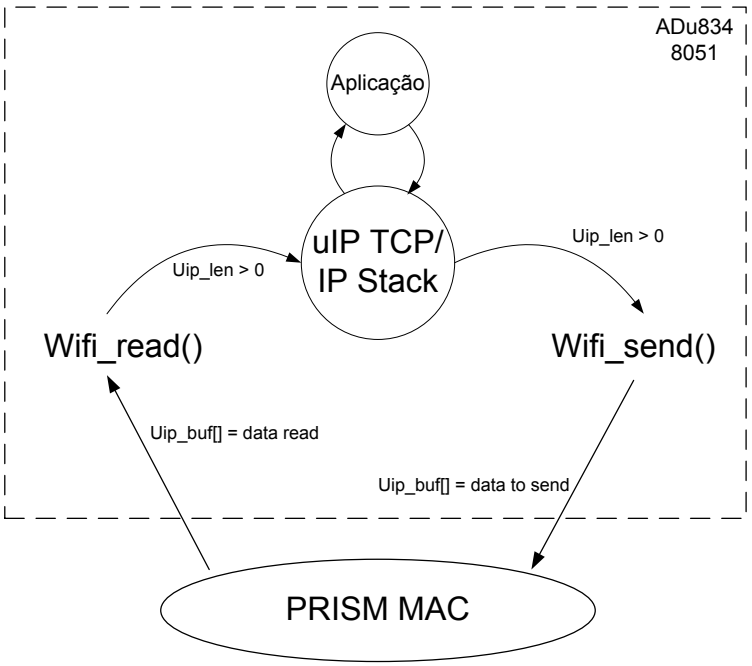


Figura 4.9: Idéia Geral do funcionamento da pilha *TCP/IP*

posto. Portanto serão apresentadas e explicadas somente as funções da implementação, relacionadas com as aplicações desenvolvidas. Estas aplicações são um assunto a ser discutido no item a seguir.

Maiores informações sobre a integração da pilha *TCP/IP uIP* ao Monitor Cardíaco e a outras possíveis aplicações podem ser obtidas no anexo A.

4.9 Aplicações desenvolvidas

Para conseguir que o Monitor Cardíaco funcionasse de acordo com os pré requisitos foi necessário dividir a aplicação principal. De um lado ficou a parte responsável pela coleta de informações da parte ECG do projeto e na outra ficou a aplicação relacionada com a camada de aplicação da pilha *TCP/IP*.

A idéia geral do lado da aplicação médica era que o Monitor Cardíaco sempre coletasse com certa frequência os dados de ECG do paciente e enviasse para a aplicação *TCP/IP*, em determinada situação os dados coletados. Mais precisamente, o que havia se pensado era criar um *buffer* circular que fosse capaz de armazenar os dados referentes a um ECG de um paciente durante 5 minutos. Este tempo de 5 minutos foi pensado como um tempo razoável de análise de dados pelo médico no caso de um acompanhamento contínuo do paciente. Um variação de batimentos cardíacos durante o tempo de 5 minutos antes que o paciente sentisse um mal estar poderia revelar ao médico o que seria necessário para

diagnosticar o paciente.

O paciente em um caso de emergência por exemplo, apertaria um botão da placa e desta forma seria capaz de enviar estes dados para o seu médico ou para uma central médica que tomaria as providências necessárias. Esta idéia original foi descartada porque para se obter uma boa resolução do sinal ECG é necessário uma frequência de amostragem de 1KHz. Durante 5 minutos seriam necessários 300.000 amostras o que implicaria em um *buffer* de tamanho aproximado de 293KB. Como a memória disponível para o armazenamento dos dados é somente a memória *RAM* do microcontrolador que se limita a pouco mais do que 1KB não é possível para o presente projeto suportar tal aplicação médica de ECG. O máximo que se consegue fazer com o projeto em questão é utilizar a memória de dados Flash do microcontrolador ADUc 834 que possui 4Kbytes para armazenamento. Com uma frequência de amostragem de 100Hz, seria possível armazenar 4096 amostras de oito bits em cerca de 41 segundos.

Outro problema encontrado durante o desenvolvimento da aplicação médica foi que sinais importantes para o funcionamento do *ASIC* e do conversor A/D tiveram que ser realocados para suportarem o funcionamento do modo *I/O* da interface *Compact Flash*. Com isto não foi possível a implementação no presente projeto de nenhuma aplicação que suportasse o *ASIC* ECG e o conversor A/D.

Para simular uma aplicação médica foi implementado uma interrupção associada ao temporizador 0 *(Timer 0)* do microcontrolador que supostamente coleta as amostras do conversor A/D mas que na realidade são partes de amostras de um sinal ECG pré gravadas na memória. Esta interrupção é chamada com uma frequência de 24Hz. Ao ser chamada, a interrupção copia para o *buffer* de saída um byte armazenado na memória. A aplicação médica possui inteligência descentralizada, ou seja, não faz o processamento local dos dados. Portanto, é feito somente a coleta, armazenamento em um *buffer* central e envio de dados para o servidor onde eles podem ser analisados e processados posteriormente.

Para resolver o problema da escassez de memória *RAM*, uma possível abordagem não implementada seria coletar os dados até que o *buffer* fosse completado e depois disso fosse enviado. Com uma frequência de amostragem de 100Hz, o *buffer* com 1000 posições seria completado a cada 10 segundos. Para o envio de 5 minutos de dados, seriam necessários 30 envios de *buffers* completos para o servidor.

No que diz respeito a aplicação do lado da camada de aplicação da pilha de protocolos *TCP/IP*, por causa da necessidade de economia de energia, era necessário desenvolver uma aplicação que atuasse sob demanda, ou seja, ficasse em estado de economia de energia. A idéia seria que a aplicação apenas faria a aquisição de dados e quando fosse necessário "acordasse" e enviasse os dados adquiridos. Para atuar desta forma é necessário que a aplicação funcione como um *cliente*. Uma segunda alternativa é permitir que o médico veja o status do paciente remotamente e para isto se conecte ao endereço *IP* do Monitor Cardíaco e seja capaz de acompanhar os dados via um *Browser*, por exemplo. Atuando desta forma, a aplicação funciona como um *servidor*.

Foi elaborado uma aplicação *cliente* que a partir de um botão de emergência, muda o estado de uma variável interna ao programa e conduz a um processo de abertura de conexão ativa com um servidor *web*. Ao ser contactado, o servidor *web*, estabelece conexão

com a placa *ECG*. A aplicação ECG prepara os dados para serem enviados formatando um pequeno cabeçalho que inclui a informação de hora, minuto e segundo provenientes do relógio de tempo real do microcontrolador e também da informação da frequência de amostragem dos dados provenientes do suposto conversor A/D. Depois da preparação dos dados, a aplicação cliente envia uma requisição *HTTP* do tipo *POST* para o servidor que recebe os dados enviados e os processa. O processamento dos dados pelo servidor não foi implementado deixando-o livre para a implementação de uma apresentação dos dados pelo próprio servidor ou mesmo o envio deles por *email* a partir de um *script* no servidor.

A aplicação médica e a aplicação *HTTP* foram escritas em arquivos separados. A pilha *TCP/IP* só faz acesso à função `webclient_appcall()` para manipulação dos dados. Esta função monitora alguns parâmetros da pilha tais como necessidade de retransmissão, timeout, abertura e fechamento de conexão através do teste de alguns sinalizadores da pilha *uIP*. A aplicação ECG é acessada somente uma vez pela aplicação *cliente* através da função `ecg_prepare_data()` que prepara os dados para serem enviados para o servidor. Esta forma de organização permite manter separada e modular as aplicações de acesso à internet e acesso aos dados de ECG.

Para teste inicial da pilha *TCP/IP* e para demonstrar a capacidade de hardware e software para atender a um dos pré requisitos do projeto foi implementado como aplicação de Internet a partir do código [mur] um mini servidor *web* que é capaz de informar o estado da conexão em forma de estatísticas indicando também quantas conexões estão ativas. Através desta aplicação foi possível verificar que é possível com uma pequena modificação no código do programa permitir que um médico acompanhe os dados do paciente através do acesso remoto ao Monitor Cardíaco.

4.10 Aplicação médica no modo memória

De modo a permitir que o Monitor Cardíaco funcionasse no seu modo original de projeto e também possibilitando a ampliação de sua funcionalidade, foi elaborado um outro módulo para o projeto permitindo a incorporação de um cartão *Compact Flash* no modo memória. Com isto, o Monitor Cardíaco pode também ser utilizado como uma espécie de *Holter* que armazena os dados do paciente que posteriormente deve encaminhar-se a um centro médico para que seus dados possam ser descarregados e analisados por um médico. Além de acrescentar o modo memória da *Compact Flash* na aplicação médica, também foi incluído uma funcionalidade que paralisa o sistema caso nenhum cartão esteja conectado ao Monitor Cardíaco.

Para que a aplicação pudesse funcionar também no modo memória foram utilizadas as funções descritas no item 4.1. Além disso, foi necessário o desenvolvimento de outras funções para complementar a integração dos dois modos de funcionamento da aplicação. A sequência de operação necessária para incluir o modo de memória na aplicação final do trabalho é mostrado no fluxograma da figura 4.10.

Foi implementado a função `find_CF_mode()` para tentar descobrir o modo de funcionamento do cartão. Esta função é chamada até 3 vezes e em cada vez, é verificado por 10

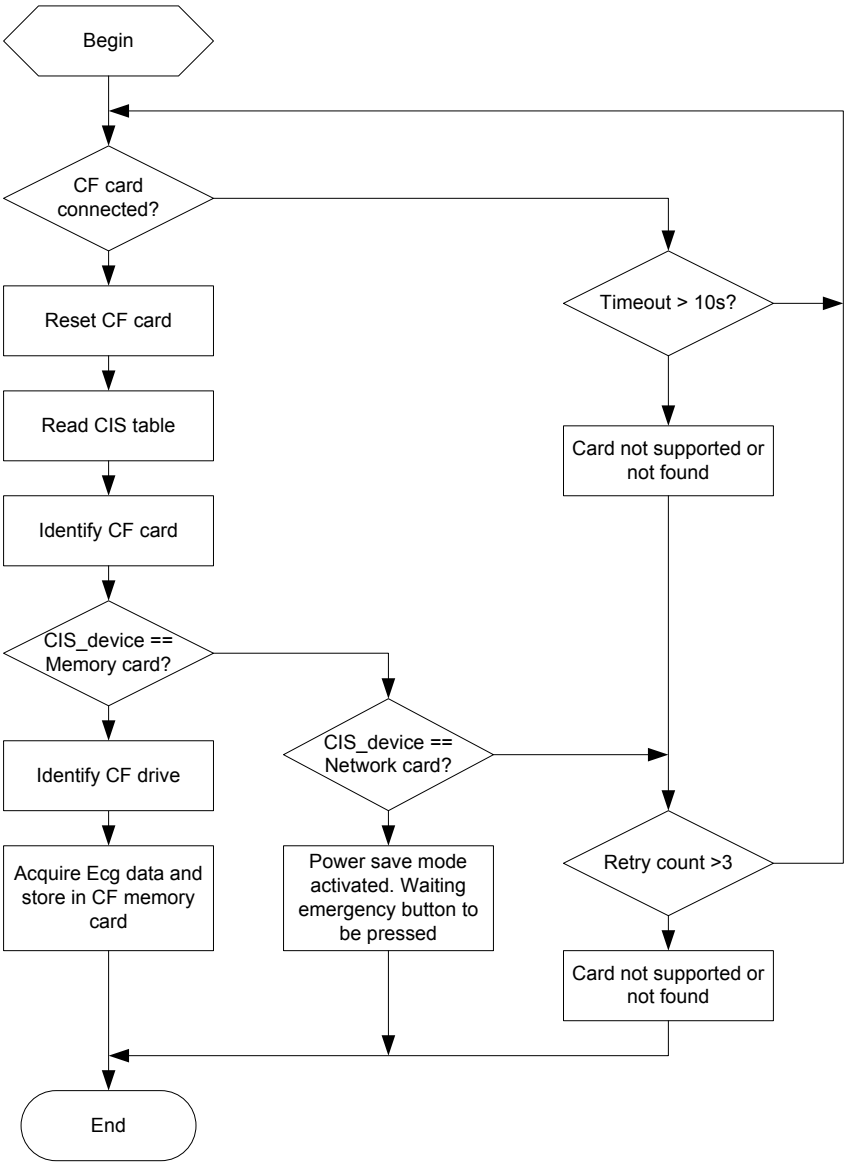


Figura 4.10: Sequência de operação para detecção do cartão *Compact Flash* nos dois modos.

segundos se algum cartão *Compact Flash* está conectado no *slot*. Caso o cartão esteja conectado, é chamado a função `ReadCISTable()` para descobrir qual o tipo de cartão inserido no *slot Compact Flash*. A estrutura *CIS* descrita anteriormente possui a tupla `FUNCID` que descreve a funcionalidade do cartão. O valor da funcionalidade é lido e armazenado na variável `cis_device`. Dependendo do valor desta variável, a aplicação chaveia para o modo *I/O* ou modo memória de funcionamento. Caso a função `find_CF_mode()` tenha sido chamada por três vezes e o modo de funcionamento lido não for suportado, o programa imprime uma mensagem de erro e é paralisado.

Caso o cartão inserido no *slot Compact Flash* seja de memória, a interrupção associada à chave de emergência é desligada e é chamada a função `CFIDDrive()`. Esta função executa o comando de identificação da memória *Compact Flash*. Esta identificação consiste basicamente em extrair do cartão *Compact Flash* a quantidade de setores existentes no cartão. No caso do cartão utilizado cada setor lido foi correspondente a 512 bytes e o número de setores encontrado foi igual a 15680.

Na aplicação médica, na rotina de interrupção do temporizador que coleta os dados foi incluído um código de programa que faz a distinção do modo de operação da *Compact Flash*. Caso o cartão inserido seja de memória, a frequência de estouro do *timer* é aumentada para 1000Hz e a função de leitura de dados do Monitor Cardíaco, `ecg_read()` é invocada com esta taxa de amostragem armazenando os dados no *buffer* principal `uip_buf[]`. Na função principal `main()`, existe uma rotina que, caso o cartão seja de memória, chama a função `CFWriteSector` que escreve o conteúdo do *buffer* `uip_buf[]` no endereço desejado. Quando é atingido o número correspondente a quantidade de setores do cartão de memória, o endereço é resetado fazendo com que os dados sejam sobrescritos na memória *Compact Flash*.

4.11 Gerenciamento de Energia nas aplicações

Para a implementação de um sistema de gerenciamento de energia nas aplicações desenvolvidas foi pensado inicialmente em duas abordagens para o projeto: uma utilizando as características do microcontrolador ADUc 834 e a outra explorando os recursos do cartão 802.11b na tentativa de economia de energia. Verificou-se através dos testes mostrados no capítulo a seguir que a utilização dos recursos do cartão 802.11b não foram satisfatórios e portanto foram descartados no fechamento da aplicação final.

Foi implementado uma máquina de estados que possui 3 estados denominados de *Ativo, Ocioso(Idle)* e *Power Down*. Uma idéia geral da implementação pode ser observada na figura 4.11. Os estados são monitorados por três interrupções diferentes de acordo com o seguinte:

- Interrupção externa INT0 - entra no modo *Ativo* a partir do acionamento da chave de emergência;
- Interrupção do temporizador 0 - controla a coleta de dados no modo *Ocioso* e aciona o modo *Power down* quando a coleta termina;
- Interrupção do Modo *Power Down* - "acorda" o microcontrolador do modo *Power Down* e ativa o modo *Ocioso* para coleta de dados.

Inicialmente, quando o Monitor Cardíaco é ligado, a aplicação entra no estado ocioso onde são coletados por cinco minutos os dados do conversor A/D referentes as amostras de dados provenientes do *ASIC*. Neste estado é utilizado o recurso do microcontrolador onde é ativado o modo ocioso *(idle)*. Neste modo o oscilador continua funcionando mas o clock

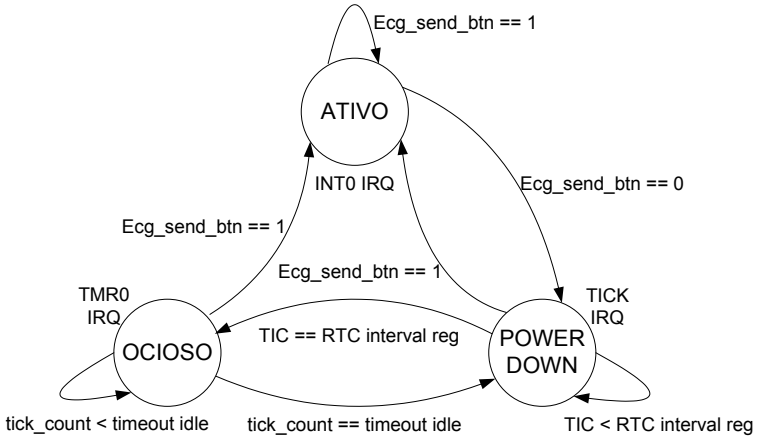


Figura 4.11: Máquina de Estados que implementa o Gerenciamento de Energia na placa ECG

gerado internamente pelo *PLL(Phase Locked Loop)* é parado. Os periféricos internos ao *chip* continuam a receber o clock e o funcionamento deles é mantido. O estado da CPU é preservado com a pilha, contador de programa e todos os outros registradores mantendo os seus dados. O microcontrolador sai deste estado se qualquer interrupção habilitada for acionada ou se o microcontrolador receber um *reset* [Dev03].

Através da interrupção do temporizador 0 do microcontrolador, a variável `tick_count` é incrementada e uma função para coleta dos dados é chamada com a frequência de 24Hz. Quando a variável atinge o equivalente a 5 minutos de coleta de dados [2], o Monitor Cardíaco entra em modo *Power Down*.

No modo *Power Down*, o *PLL* e o *clock* do microcontrolador são parados. O oscilador do *chip* continua a oscilar e uma variável `TIC` associada ao *clock* do oscilador é habilitada. Todos os outros periféricos são desligados. Os pinos mantém todos os níveis lógicos neste modo.

Um conjunto de registradores são programados para que no intervalo de uma hora, uma outra interrupção seja acionada fazendo com que o microcontrolador saia do estado *Power Down* e vá para o estado *Ocioso*. No estado *Ocioso*, conforme descrito anteriormente, o microcontrolador fica 5 minutos coletando dados e depois retorna para o estado *Power Down*. Tanto no modo *Power Down* quanto no modo *Ocioso* o cartão de rede sem fio permanece resetado.

Os dados coletados são armazenados no *buffer* principal `uip_buf` e enviados para um servidor a partir de uma chave controlada pelo paciente que é associada ao pino de interrupção externa *INT0*. Quando a chave é acionada em nível lógico baixo, a interrupção associada a este pino é ativada, setando a variável `ecg_send_button` e mudando o estado do Monitor Cardíaco para *Ativo*. Neste momento, em pouco tempo, a aplicação é chamada

[2]Valor do registrador do temporizador em torno de 7200

abrindo uma conexão com o servidor, enviando os dados para ele e aguardando por um OK no recebimento dos dados via protocolo *HTTP*. O servidor envia o OK, fecha a conexão e o Monitor Cardíaco reconhece voltando para o modo *Power Down* onde permanece até que a interrupção para a coleta de dados seja acionada ou que o modo *Ocioso* para coleta de dados seja ativado.

Ao entrar no estado *Ativo*, o programa aciona o *driver* do cartão de rede sem fio, inicializando-o, conectando ao ponto de acesso configurado e posteriormente implementando todo o processo de conexão *TCP/IP* implementado pela pilha *uIP*. A aplicação desenvolvida se encarrega de repassar os dados coletados de ECG para a pilha *uIP*.

O consumo de energia aproximado foi obtido medindo a corrente contínua através de um multímetro em série com o circuito do Monitor Cardíaco. Os valores foram conseguidos através da medição em cada estado de consumo de energia.

Através da implementação da máquina de estados foi possível reduzir o consumo de energia do Monitor Cardíaco conforme será mostrado no capítulo seguinte.

Capítulo 5

Testes e Resultados

Para verificar se o Sistema embutido de Monitoramento Remoto atendia aos pré-requisitos de projeto conforme o capítulo 2 foram necessários a realização de vários testes e também de estudos. O objetivo foi checar se a maioria dos requisitos pudesse ser validado e desta maneira certificar realmente a possibilidade de uso do hardware do Monitor Cardíaco para o sistema proposto.

Testes com a interface de rede sem fio foram realizados com o intuito de checar a eficiência do protocolo 802.11b. Foram testados também aspectos relacionados com o protocolo tais como velocidade para o envio de dados.

Em relação às aplicações desenvolvidas, foi testado se poderiam ser utilizadas como cliente ou servidor. Outro aspecto testado foi a possibilidade de conexão com a Internet através da utilização de programas de uso comum nesta rede. Foi testado também a funcionalidade embutida no Monitor Cardíaco de ser capaz de funcionar em dois modos de operação do padrão *Compact Flash*: Memória e *I/O*.

O consumo de energia foi testado para verificar e validar a máquina de estados proposta no capítulo anterior e com isto certificar se poderia ser utilizada como mecanismo de gerenciamento e economia de energia. Foi verificado também o consumo baseado em uma possível forma de uso do sistema embutido de Monitoramento Remoto para estimar a duração de um tipo de bateria a ser utilizada no projeto.

De forma a analisar se o projeto poderia ser aceito comercialmente e fosse viável economicamente, foi feito um estudo de custo do projeto. Uma outra análise foi feita desta vez para checar o tamanho do código ocupado pelo programa implementado com o propósito de saber se outras aplicações podem ser incorporadas futuramente ao projeto.

Devido a perda de funcionalidade do projeto em relação ao circuito ECG não foi possível realizar nenhum teste relacionado com a aplicação médica e com isto esta parte do projeto não foi validada neste trabalho.

5.1 Ferramentas para depuração e testes dos programas

Os programas desenvolvidos cujos códigos são apresentados no anexo B, foram testados primeiramente usando-se o simulador do compilador C Keil51. O compilador possui um poderoso sistema de simulação onde pode-se simular valores de variáveis, colocar *break-points* e verificar como o programa está se comportando de maneira geral. Para simular os valores dos registradores necessários para funcionamento do *driver* da interface de rede sem fio foi usado uma variável de simulação que dependendo do seu valor, atribuia valores aos registradores e variáveis. A utilização desta variável de simulação foi muito importante principalmente no desenvolvimento do *driver* do dispositivo de rede sem fio.

Outro recurso bastante utilizado durante o desenvolvimento foi a utilização de impressões de variáveis e informações relevantes sobre a localização da execução do programa na *UART* do microcontrolador ADUc834. A *UART* foi configurada para operar a 4800bps, 8 bits de dados, sem paridade e um bit de parada (4800 8 N 1). As impressões na porta serial eram lidas através do programa *HyperTerminal* do *Windows* conforme mostrado na figura 5.1. Com as impressões foi possível verificar como o programa rodava em diferentes situações.

Uma outra ferramenta utilizada na depuração foi a utilização de dois LEDs*(Light Emitting Diode)* existentes na placa. Através deles pode-se analisar situações onde a impressão de informações via porta serial não foi possível tais como dentro das rotinas de interrupção.

5.2 Testes com a interface de rede sem fio

Os testes de funcionamento da interface de rede sem fio foram feitos com a utilização de um Ponto de Acesso modelo WAP11 da *Linksys*. Este Ponto de Acesso foi configurado para operar com o endereço *IP* 192.168.1.250. Ligado a este Ponto de Acesso estava um Computador PC com placa de rede Ethernet e endereço *IP* 192.168.1.50. A configuração de rede utilizada pode melhor ser resumida através da figura 5.2.

O desenvolvimento do *driver* de dispositivo para a rede sem fio seguiu as especificações do manual [Inc02]. Os primeiros sinais de sucesso com a implementação vieram através da leitura da estrutura da informação do cartão (*CIS*). Os dados começaram a ser lidos e impressos na porta serial para confirmação. Todos os dados estavam de acordo com o manual. Depois do comando de inicialização que foi executado com sucesso, o cartão Compact flash ajudou na indicação do resultado através do *LED* de *Link* do cartão piscando. Ao tentar se conectar com o ponto de acesso, o *LED* de *link* do cartão começou a piscar com uma intensidade maior até que ficou definitivamente aceso. Isto foi o sinal que a interface de rede havia conseguido se conectar com o ponto de acesso. Através de um identificador de recurso (*RID*) foi possível identificar o endereço de Controle de Acesso ao Meio *MAC* do ponto de acesso. Foram os primeiros sinais que indicavam que o *driver* da interface de rede sem fio estava funcionando.

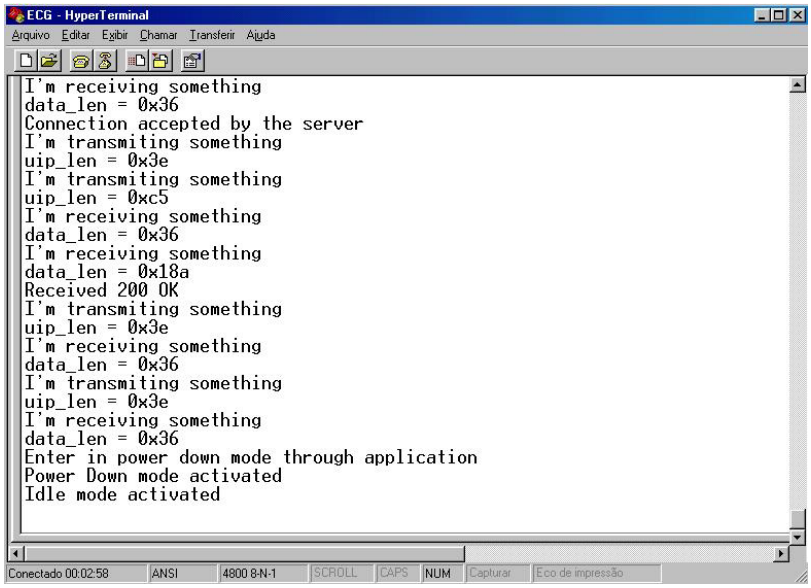


Figura 5.1: Exemplo de uma tela do *Hyperterminal* com uma sequência de mensagens impressas que auxiliaram na depuração dos programas desenvolvidos

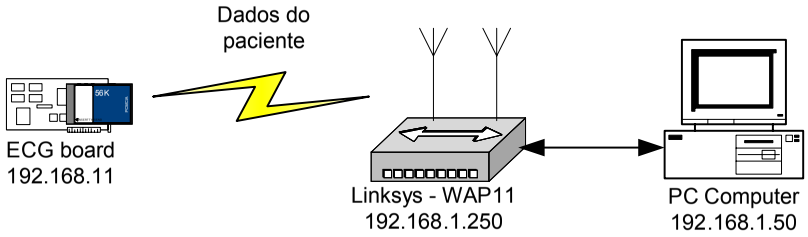


Figura 5.2: Configuração de rede utilizada para teste da Interface de rede sem fio.

Para teste do recebimento e envio dos dados foi utilizado a aplicação *ping* do protocolo *ICMP* que estava embutido na pilha *TCP/IP uIP*. A placa ECG foi configurada com o endereço*IP* fixo 192.168.1.1 e partir do computador o comando *ping* foi lançado. Inicialmente os dados foram recebidos sendo necessários alguns ajustes de deslocamento de forma a receber somente o cabeçalho *Ethernet*. Ao serem ecoados, os dados apresentaram problema e não puderam ser enviados. Depois de alguns dias descobriu-se que o problema estava na inversão da ordem dos *buffers* de saída do *MAC PRISM* que possui ordem inversa em relação ao recebimento o que levou a confundir o desenvolvimento da função de transmissão. Os comandos *ping* foram enviados e retornados com um tempo médio de 236ms.

5.3 Testes com as aplicações

Inicialmente foi adaptada uma aplicação desenvolvida juntamente com a pilha *uIP* para teste do funcionamento da pilha *TCP/IP* uma vez que as funções do *driver* de rede sem fio já se encontravam funcionando. A aplicação foi um mini servidor *web* que fornece informações sobre a pilha *TCP/IP* tais como número de conexões ativas, estatísticas sobre perda de pacotes, pacotes *IP*, *TCP* e *ICMP* recebidos dentre outras.

Para verificação do funcionamento da aplicação foi utilizado o software analisador de protocolo de rede *Ethereal* versão 0.10.9. Uma tela do software *Ethereal* mostrando o acesso a uma das páginas do servidor *web* implementado é mostrado na figura 5.3. A página com estatísticas do servidor pode ser observada na figura 5.4.

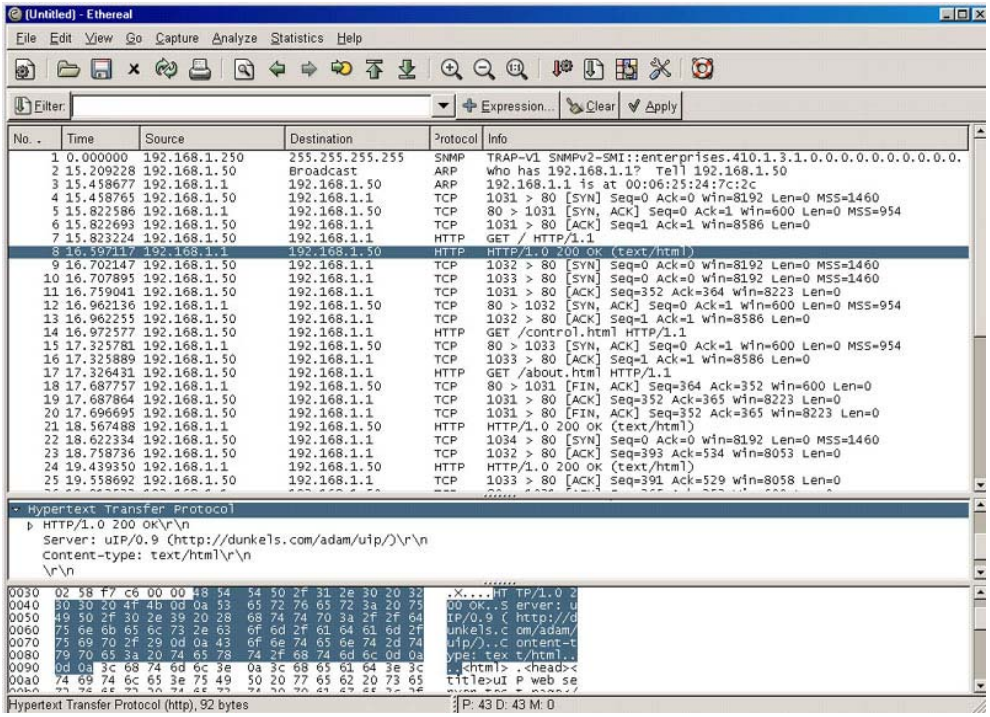


Figura 5.3: Sequência de dados a partir de um acesso ao servidor *Web* implementado na placa ECG

A aplicação médica em conjunto com a aplicação de internet final desenvolvida descrita no item 4.9 também foram testadas com o auxílio do software *Ethereal*. Para teste desta aplicação, a placa *ECG* atua como *cliente* e o computador *PC* onde estava conectado o ponto de acesso sem fio foi configurado para ser um *servidor web*. Para isto foi instalado neste computador o software servidor *HTTP Apache*[apa] versão 1.3.33. Para suporte e teste da aplicação *HTTP POST* implementada no monitor cardíaco foi necessário ainda a instalação e configuração do *PHP* versão 5.0.4.

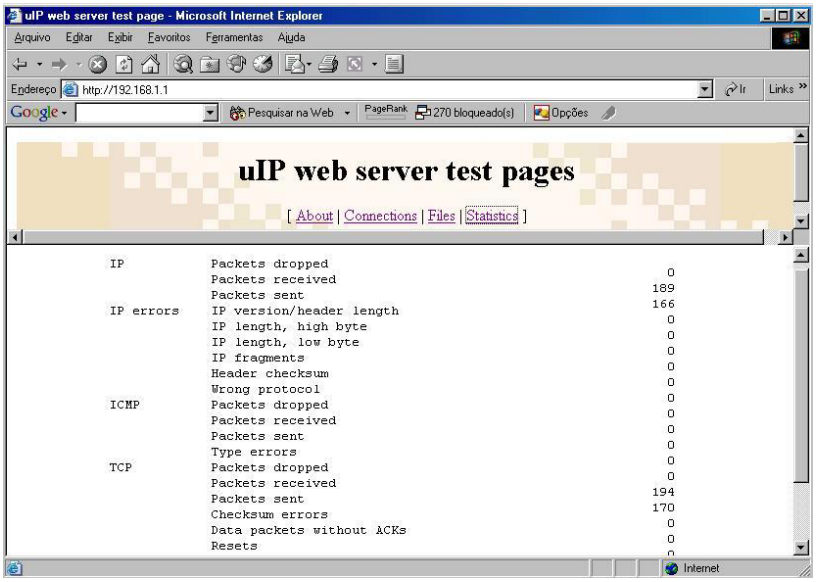


Figura 5.4: Uma das páginas acessadas no Mini-servidor *Web* implementado na placa

Para teste da comunicação no protocolo *HTTP* usando o método *POST*, foi criado um arquivo na linguagem *PHP* no servidor *Web* que apenas checa se o campo *User-Agent:* do cabeçalho *HTTP POST* é do tipo *ECG Webclient*. Caso positivo, uma mensagem de volta é enviada e a conexão é fechada pelo servidor. O cliente *ECG* recebe a mensagem *HTTP* com o cabeçalho OK, decodifica este cabeçalho e reconhece o fim da conexão enviando uma pacote de reconhecimento *ACK* e entrando no modo de energia *Power Down*.

Conforme mencionado, não foi implementado no servidor nenhuma aplicação para tratamento dos dados enviados. Apenas testou-se a capacidade de comunicação da placa *ECG* com um servidor *web*. A tela do software *Ethereal* com a sequência de pacotes *TCP/IP* e *HTTP* enviados pode ser observada na figura 5.5.

Após os testes com a interface de rede sem fio e as aplicações desenvolvidas foi inserido no programa um código para que o Monitor Cardíaco funcionasse tanto no modo Memória quanto no modo *I/O*. O desenvolvimento deste código foi explicado no item 4.10. Para teste do funcionamento em ambos os modos foram colocadas no programa, impressões de mensagens na porta serial do microcontrolador para reportar o progresso da execução conforme explicado no item 5.1. Telas do software *Hyperterminal* mostrando o progresso do programa em cada modo de operação da *Compact Flash* são mostradas nas figuras 5.6 e 5.7.

Na figura 5.6 é mostrado o modo Memória de operação da aplicação final. A aplicação é inicializada com um título e depois parte da tabela *CIS* é apresentada. Logo em seguida é apresentado o número de setores encontrado no cartão que no caso foi de 15.680 ou 0x3d40h. Na sequência, o tipo de dispositivo reconhecido através da variável `CIS_device` é impresso na tela. O número 4 indica que o cartão *Compact Flash* é do tipo armazenador

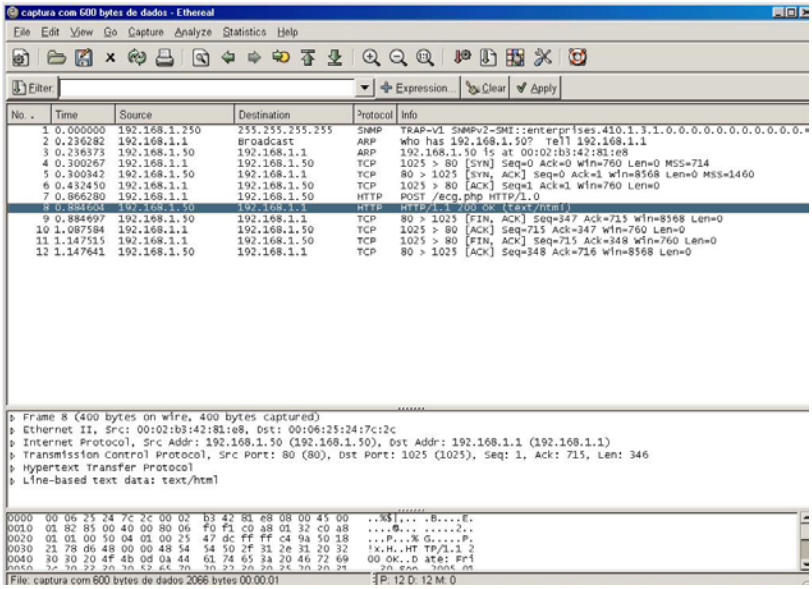


Figura 5.5: Sequência de pacotes trocados entre o cliente ECG e o servidor *HTTP*

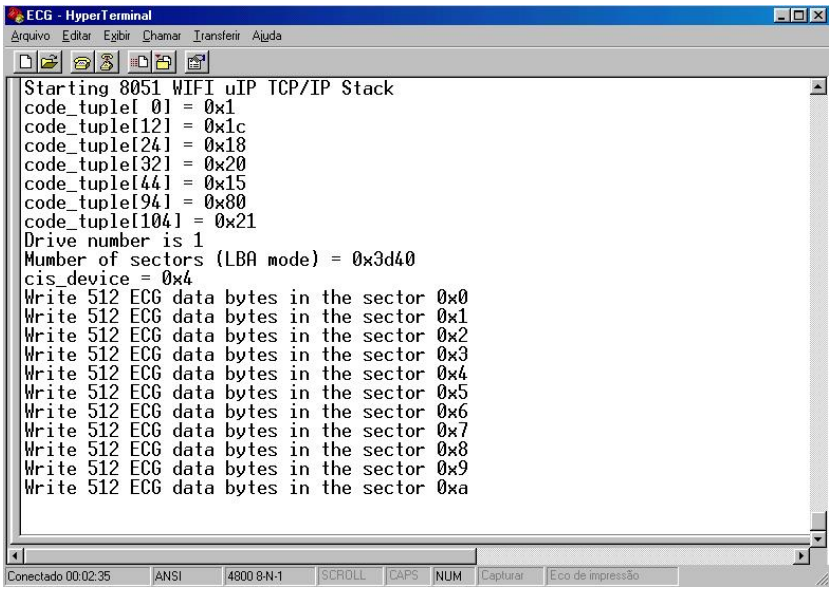


Figura 5.6: Modo de operação Memória da aplicação final

de dados. A próxima etapa é a escrita dos dados nos setores do cartão de memória da *Compact Flash* que pode ser observada pela impressão das mensagens de escrita dos dados ECG na tela do *Hyperterminal*.

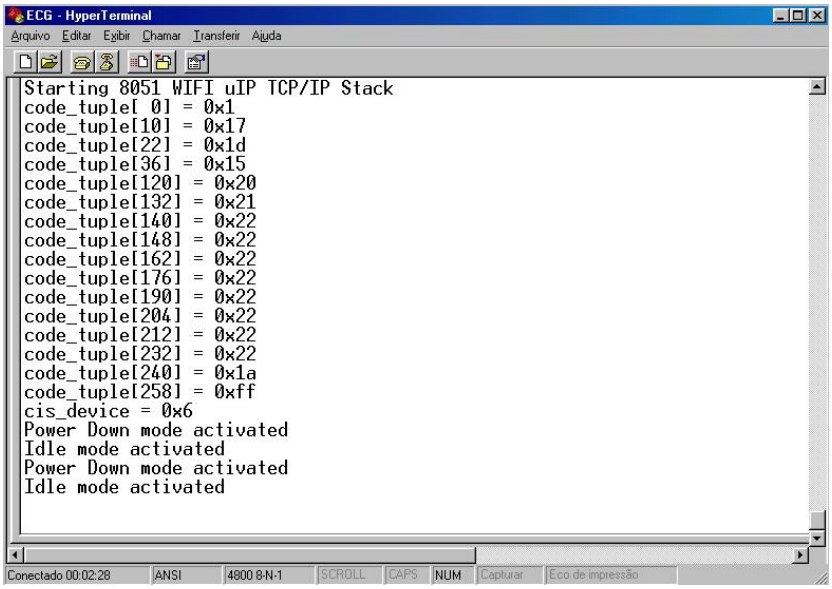


Figura 5.7: Modo de operação *I/O* da aplicação final

A figura 5.7 mostra o modo de operação *I/O*. Pode ser observado que semelhante ao explicado anteriormente, a mensagem inicial é apresentada e parte da tabela *CIS* também. Na sequência, a funcionalidade do cartão *Compact Flash* de rede sem fio é reconhecida através do valor 6 assumido pela variável `CIS_device`. A partir disto, o Monitor Cardíaco é colocado em modo de economia de energia alternando entre os modos *Power Down* e *Ocioso* até que a chave de emergência seja pressionada para o envio dos dados ECG através da interface de rede sem fio.

5.4 Consumo de Energia do Monitor Cardíaco

Como descrito no item 4.11, foram implementados três estados diferentes de consumo de energia: *Ativo, Ocioso e Power Down*. Os modos de energia da placa *Ocioso* e *Power Down* foram configurados para efeito de teste com mudança entre os estados a cada 30 segundos. O consumo de energia foi baseado no consumo de corrente contínua da placa e medido com um multímetro digital Minipa modelo ET-1502. O multímetro foi ligado em série com o circuito do Monitor Cardíaco e os valores o consumo de corrente foram obtidos chaveando-se entre os três estados de consumo de energia.

Primeiramente, ao entrar no modo *Power Down* e *Ocioso*, antes da mudança para cada estado, o cartão de rede sem fio foi desconectado da rede na esperança de reduzir o consumo de energia. Com esta abordagem o consumo foi de 75mA no modo *Ocioso* e 64mA no modo *Power Down*. Decidiu-se colocar o cartão de rede sem fio resetado constantemente uma vez que ele não é utilizado em nenhum momento nestes dois modos. Depois desta alteração o

consumo caiu para 61mA no modo *Ocioso* e 50mA no modo *Power Down*.

Descobriu-se posteriormente que o modo *Ocioso* do microcontrolador tinha sido configurado para ser ativado somente quando se entrava no estado e logo em seguida era desativado pela interrupção do temporizador 0. O programa foi alterado então para ativar o modo *Ocioso* constantemente durante a permanência do software neste estado. Com esta modificação o consumo no modo *Ocioso* caiu para 55mA com a aplicação implementada.

No modo *Ativo* o consumo de energia foi variado com o mínimo de cerca de 70mA até picos de 283mA.

Implementado a máquina de estados e feito os testes partiu-se para a técnica de redução da frequência de *clock* do microcontrolador na expectativa de redução do consumo de energia da placa. Este teste foi possível porque o microcontrolador ADUc834 possui o recurso de alteração de frequência de *clock* através da configuração de um registrador onde é possível setar até oito frequências de *clock* diferentes chegando até a frequência máxima de 12,58MHz.

Em testes preliminares verificou-se que a escrita e leitura do cartão *Compact Flash* no modo *I/O* mostrado no capítulo 4 não funcionava com a frequência de *clock* menor que 12,58MHz, portanto no modo *Ativo* era necessário configurar o registrador para a frequência máxima de *clock*. Restava então alterar o *clock* no modo *ocioso* uma vez que no modo *Power Down* o registrador de *clock* era desligado. O *clock* foi alterado para a frequência padrão do microcontrolador que é de aproximadamente 1,57MHz. Com esta redução de frequência, o consumo no modo *Ocioso* caiu para cerca de 53mA. A frequência foi então configurada para o valor mínimo e com isto atingiu o consumo de cerca de 51mA se aproximando do consumo no modo *Power Down*.

Para verificar a influência do consumo do cartão de rede sem fio, foi feito outro teste desta vez removendo-se o cartão do conector *Compact Flash*. O consumo verificado foi de 19,5mA para o modo *Ocioso* e 14,8mA para o modo *Power Down*. Verificou-se que o consumo neste caso do cartão de rede sem fio é cerca de 35mA o que significa 70% do consumo da placa. Em [Inc01] é citado que o consumo típico de corrente com Vcc=3,6V é de 33mA o que se aproxima do valor que foi medido. Todos os dados relevantes de consumo medido por todos os testes realizados e bem como os príncipais resultados obtidos são mostrados nas tabelas da figura 5.8.

No modo *Ativo* foi verificado através do software *Ethereal* que o tempo para transmissão foi medido conforme a tabela 6.1. Percebe-se que quando são impressas informações via porta serial que foram usadas para *debug* do programa, o tempo para transmissão de dados praticamente dobra. Outro fator relevante é que cerca de 200ms a 300ms de tempo a mais é necessário para o envio de uma amostra de dados de tamanho razoável para aplicação do monitor cardíaco.

Com base no tempo gasto para envio dos dados pode-se estimar o consumo da placa admitindo por exemplo que um paciente enviaria os dados para o seu médico 3 vezes por dia e ficasse 24 horas por dia com o monitor cardíaco. Desta forma teriamos que em um dia 2 horas seriam dedicadas para a coleta de dados no modo *Ocioso*, o que significa um consumo de 110mAh. Admitindo que o paciente enviaria os dados no momento que o monitor cardíaco estivesse no modo *Power Down*, este ficaria 21,99917 horas neste modo,

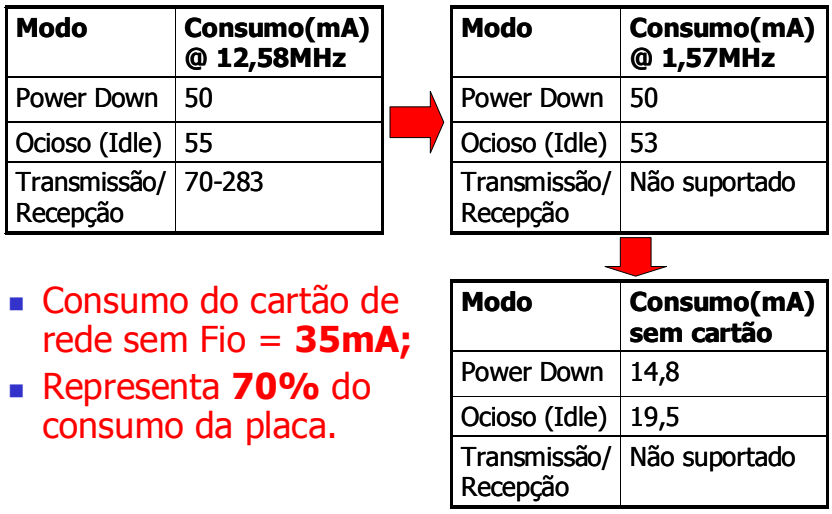


Figura 5.8: Tabelas e principais resultados obtidos com o consumo de corrente do Monitor Cardíaco

Tamanho da amostra de dados(Bytes)	Impressão via serial?	Tempo(seg)
0	Não	0,73
600	Não	1,07
0	Sim	1,82
600	Sim	2,02

Tabela 5.1: Resultados obtidos do tempo de transmissão dos dados pelo monitor cardíaco

consumindo 1099,96mAh. O consumo do envio dos dados seria de 0,23mAh. Com todas estas situações o consumo médio do Monitor Cardíaco seria de 50,42mA. Somando-se tudo chegariamos ao valor de consumo de 1210,2mAh. Usando-se uma bateria de 2500mAh seria possível ao paciente utilizar o monitor cardíaco por 2 dias. Caso o paciente use o equipamento 8 horas por dia pode-se utilizar as baterias por até 6 dias.

A fonte projetada para a placa não suporta o consumo do cartão de rede sem fio por isso não foi possível fazer o teste com baterias no circuito para verificação da autonomia real do circuito. O único teste que foi feito foi o teste das baterias com o circuito sem o cartão de rede sem fio que não acrescenta em nada ao trabalho.

5.5 Tolerância à Falhas

Em alguns experimentos com a medição do consumo da placa foi percebido que em determinados momentos da transmissão dos dados, o cartão de rede sem fio provavelmente por causa do mal contato dos fios de alimentação, não conseguia transmitir. Além disto, embora não verificado, pode-se imaginar situações de conexão onde o acesso à rede sem fio pode ser prejudicado devido redução da qualidade do sinal ou alguma interferência, provocando uma falha causada por perturbação externa.

Uma outra situação de falha pode ocorrer em casos onde a energia de alimentação das baterias não consegue suprir o correto funcionamento do monitor cardíaco levando o sistema a cometer erros podendo até mesmo chegar a enviar dados incorretos sobre o paciente o que é proibido para um sistema desta categoria. Diante deste cenário, verificou-se que era necessário a implementação de um sistema de tolerância à falhas de modo a evitar que o cartão ficasse indefinidamente no modo *ativo*, justamente o de maior consumo de energia.

O microcontrolador ADUc834 fornece suporte para implementação de um sistema de tolerância à falhas através da implementação de um *Watch Dog Timer (WDT)*. Este recurso é implementado através de um registrador que é ligado a um temporizador de 16 bits. Este temporizador é setado para funcionar antes de determinada operação e se a operação não for completada até o *reset* dele, o microcontrolador é reinicializado e um bit do registrador de controle do *Watch Dog Timer* é setado. É possível configurar oito diferentes tempos para o *reset* do temporizador *WDT*.

Como os pontos críticos de funcionamento ou de provável falha no funcionamento do projeto foram identificados inicialmente como sendo relacionados com a interface de rede sem fio, foi habilitado antes das operações principais de inicialização, transmissão e recepção dos dados o temporizador *WDT* para funcionar com um tempo de 2 segundos em cada uma. O tempo de 2 segundos foi considerado como o tempo máximo para abertura de conexão *TCP/IP*, envio dos dados e fechamento de conexão conforme resultados obtidos no item anterior.

Caso o tempo de 2 segundos expire, o microcontrolador é resetado. Sendo resetado, inicialmente ele entra no modo *ocioso* de energia para a coleta dos dados por 5 minutos. Após a coleta dos dados, como houve o reset pelo *WDT* é possível reconhece-lo através de um bit do registrador de controle do temporizador. Desta forma pode ser identificado que houve algum problema no processo de envio dos dados já que o *WDT* é vinculado a operações relacionadas ao cartão de rede sem fio. O estado do monitor cardíaco passa novamente para o *ativo* onde novamente o processo de envio dos dados é acionado. Caso os dados sejam enviados com sucesso, o circuito entra no modo *Power Down* caso contrário o microcontrolador é resetado novamente e o processo de coleta e transmissão dos dados é invocado. Este processo é indicado na figura 5.9.

Testes foram realizados simulando condições de falha na transmissão tais como configurações do *WDT* para tempos inferiores ao de realização das tarefas principais de inicialização, transmissão e recepção dos dados e verificou-se que o *reset* do microcontrolador funcionou. A tentativa de envio dos dados, posterior a coleta após o *reset* via *WDT* foi

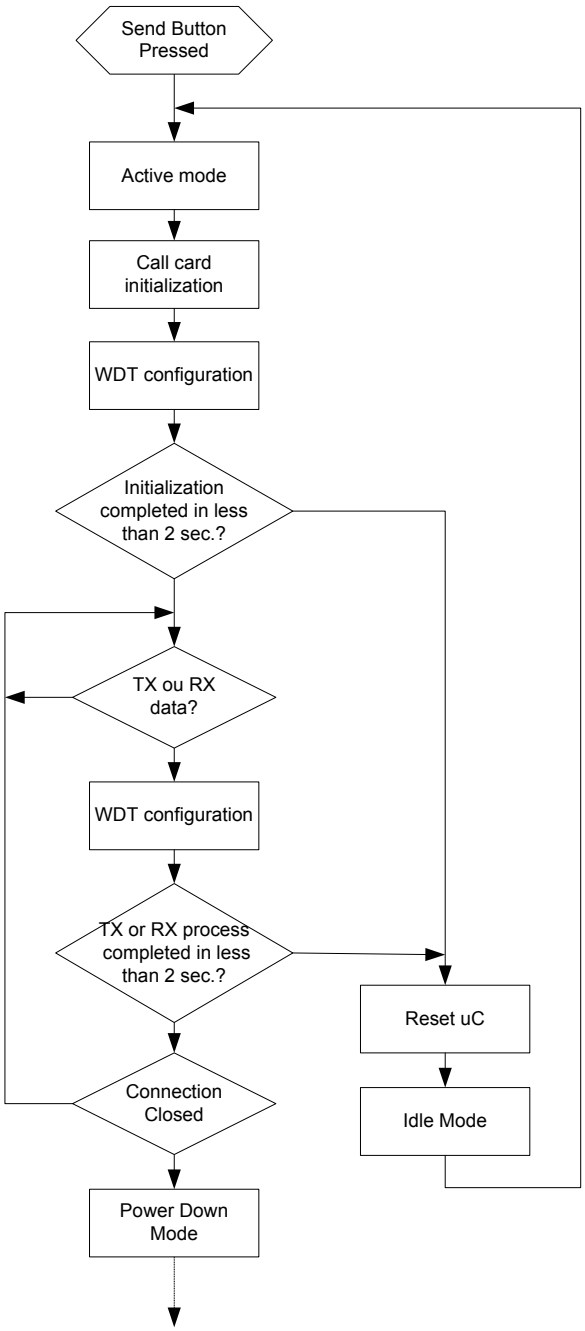


Figura 5.9: Sistema de tolerância à falhas implementado e testado para o monitor cardíaco

verificada após as falhas provocadas comprovando o funcionamento do sistema.

Para resolver o problema da questão da pouca energia para a alimentação do circuito, foi idealizado mas não testado um sistema baseado em outro recurso do microcontrolador

ADUc834. Trata-se do monitor de fonte de alimentação, ou *Power Supply Monitor (PSM)*. Este recurso quando habilitado indica se a alimentação cai abaixo de quatro níveis selecionados pelo usuário que variam de 2,63V a 4,63V.

A queda na alimentação abaixo dos níveis programados pelo usuário é indicada por um bit de um registrador dedicado para esta funcionalidade do microcontrolador. Este bit gera uma interrupção que quando acionada pode permitir que, por exemplo, o último conjunto de dados adquiridos do paciente sejam copiados da memória *RAM* para a memória *Flash* do microcontrolador. Esta ação evitaria a perda de dados que pode ser útil em determinadas situações de necessidade de acompanhamento mais continuo, por exemplo.

5.6 Análise de custo

O projeto do monitor cardíaco utiliza componentes de baixo custo a não ser pelo preço do cartão de rede sem fio que custou em torno de U$50. Em conversas com o projetista do monitor cardíaco foi estimado o custo de protótipo do monitor cardíaco como sendo de U$300 e custo para produção em larga escala de no máximo U$250.

Em comparação com outros sistemas similares na literatura a maioria deles usavam um *PDA* como hardware principal. Considerando o preço de um *PDA* de baixo custo que inclui rede sem fio 802.11b chegariamos ao preço de U$300 desconsiderando o preço do módulo de coleta dos dados fisiológicos. Também como solução para recebimento, processamento e envio dos dados pode-se usar um celular com características especiais que na época do desenvolvimento deste trabalho tinha um custo aproximado de U$400. Este telefone ainda precisa ser acoplado a um módulo de coleta dos dados tendo como padrão de comunicação geralmente utilizado o *Bluetooth*.

As soluções para coleta de dados fisiológicos com as interfaces para o *PDA* e *Bluetooth* são escassas, a maioria delas são projetos não comerciais feito pelos próprios desenvolvedores dos projetos. Isto torna os módulos de coleta caros e difíceis de serem encontrados no mercado.

Pode-se perceber então que em termos de custo o Monitor Cardíaco pode ser uma alternativa mais viável para uso como plataforma de hardware de um sistema de monitoramento remoto de pacientes.

5.7 Tamanho do código das Aplicações

O tamanho do código foi uma importante consideração pois permitiu avaliar a possibilidade de expansão e implementação de novas funcionalidades. A tabela 5.2 sumariza o tamanho do código para as várias aplicações implementadas de acordo com o compilador C Keil51.

Pode-se perceber que a memória *RAM* tanto interna quanto externa são um recurso escasso. A memória *RAM* externa depende muito do número de conexões que a aplicação pode receber e do tamanho do *buffer* `uip_buf()` utilizado. No caso da aplicação *ECG Webclient* são suportadas duas conexões e um tamanho de *buffer* equivalente a um pacote

Aplicação	**Memória RAM de dados interna**	**Memória RAM de dados externa**	**Memória de Programa interna**
ECG Webclient	138	2023	18042
ECG Webclient (Debug Version)	138	2023	18753
ECG Webserver	183	2022	23120

Tabela 5.2: Tamanho de código em bytes para as várias aplicações desenvolvidas

de dados *Ethernet*, ou seja, de 1500 bytes e no caso do *ECG Webserver* são suportados um tamanho de *buffer* de 1500 bytes e três conexões. A aplicação *cliente* possui menos suporte a conexões devido ao suporte para tamanho do cabeçalho *HTTP* enviado que na aplicação *servidor* possui 5 bytes e na aplicação *cliente* possui 56 bytes.

Para permitir que novas variáveis sejam adicionadas em novas aplicações que necessitarem de quantidade de memória maior é necessário ou diminuir o número de conexões ou diminuir o tamanho do *buffer* `uip_buff()`.

Em relação à memória de programa, percebe-se que o espaço ocupado pelos códigos da tabela 5.2 é cerca de um terço do total de memória disponível que é de 62Kbytes. Desta forma é possível ainda implementar cerca de dois terços de programa o que permite grande possibilidade de incorporação de novas funcionalidades e recursos ao sistema.

5.8 Principais dificuldades

As principais dificuldades no desenvolvimento dos trabalhos foram relacionadas com desenvolvimento do *driver* do dispositivo de rede sem fio. A escassez de documentação do *MAC PRISM* demoraram o entendimento do funcionamento do *chip* uma vez que pela documentação do *driver* de dispositivo desenvolvido para o *Linux* era de difícil entendimento. Estes problemas só foram resolvidos quando foi obtida a documentação que descrevia melhor o funcionamento do *PRISM* que possibilitou identificar corretamente o endereço do registrador de *reset* e posterior sucesso na leitura da tabela *CIS* e do desenvolvimento das outras funções de controle discutidas anteriormente.

O *MAC PRISM* possui formatos diferentes para escrita e leitura dos dados nos caminhos de Acesso os Buffers *BAPs*. Isto dificultou principalmente o processo de escrita dos dados e consequentemente a transmissão destes. Alguns dias foram gastos na tentativa de se descobrir qual a ordem correta os dados deveriam ser escritos no *buffer* para que eles fossem transmitidos corretamente.

Em relação a modificação realizada no circuito, a checagem para verificação de quais pinos iriam ser retirados, o corte das trilhas do circuito impresso da placa e a colocação de

fios para ligação dos pinos do microcontrolador aos pinos do conector *Compact Flash* foi um processo demorado e complicado que necessitou de um grande cuidado principalmente no que diz respeito a danificação dos componentes uma vez que, a princípio, não existiam extras para substituição.

Capítulo 6

Novo Projeto

Tendo em vista que o projeto da placa ECG teve que ser alterado para suportar o modo I/O da *Compact Flash*, alguns pinos que originalmente pertenciam à interface com o circuito de ECG tiveram de ser realocados. Por isto, não foi possível implementar uma aplicação que demonstrasse ao mesmo tempo uma aplicação médica com a interface de rede sem fio.

Para resolver este problema é proposto um novo projeto que modifica a interface original para que todos os pinos sejam realocados sem a perda de funcionalidade dos circuitos. Alguns circuitos foram acrescidos ao projeto original para incluir recursos que pretendem melhorar principalmente a questão da economia de energia. Alguns pinos são realocados de posição somente para facilitar a manipulação pelo programa de controle como é o caso da troca do pino mais significativo com o menos significativo e implementação do barramento de endereços na porta 2 do microcontrolador. Todas as modificações no projeto são mostradas no esquema elétrico no Anexo C. Estas modificações acrescentam um custo mínimo no projeto final.

6.1 Modificações no circuito do Microcontrolador

Para acesso aos registradores do *MAC PRISM* são necessários 6 linhas de endereço. Para a leitura de todos os endereços da tabela *CIS* são necessários 8 linhas de endereço e para o acesso ao endereço do registrador de *reset* são necessários 10 linhas de endereço. Se considerarmos que a leitura da tabela *CIS* é apenas necessária para descobrir o endereço de *reset* e que apenas uma leitura parcial é suficiente podemos ignorar os seus dois últimos bits e considerar para o acesso a ela apenas 6 bits. O endereço de *reset* possui os quatro bits mais significativos com valores iguais e portanto podem ser curto circuitados entre si com o sétimo bit mais significativo. Esta simplificação reduz a quantidade de pinos de endereços a serem utilizados de 10 pinos para 7 pinos.

Para acesso do modo *I/O* foi necessário a inclusão dos sinais *IORD*, *IOWR*, *REG* e *RESET*. A necessidade de inclusão destes sinais precisou que fosse acrescido um outro circuito no projeto original de modo a expandir os pinos do microcontrolador. Foi utilizado

para isto um circuito decodificador de 3 linhas para 8 linhas *(3:8)*. A escolha dos sinais teve que ser feita de forma que um não fosse dependente do outro e que não ocorresse ao mesmo tempo, caso contrário não seria possível inclui-lo na linha de decodificação. Como os sinais de controle de escrita e leitura tanto do modo memória quanto no modo *I/O* são independentes e não ocorrem ao mesmo tempo, eles puderam ser incluídos nas linhas de decodificação. A tabela de decodificação dos sinais de controle de escrita e leitura é mostrada na tabela 6.1. Quando a combinação das entradas é atingida, apenas uma saída é ativada em nível lógico baixo enquanto todas as outras permanecem em nível lógico alto.

DEC_C	**DEC_B**	**DEC_A**	**Saída(L)**
0	0	0	RD#
0	0	1	WR#
0	1	0	RESET
0	1	1	LED1
1	0	0	LED2
1	0	1	-IOWR
1	1	0	-IORD
1	1	1	Enable_CF

Tabela 6.1: Tabela de expansão dos sinais de controle do microcontrolador

Uma chave para permitir que o paciente acione em caso de emergência ou quando quiser enviar os dados para o médico também foi acoplada ao projeto. Esta chave é ligada no pino de interrupção *INT0* que é acionada para alterar o estado do monitor cardíaco para *ativo* e proceder com a abertura de conexão no servidor e envio dos dados.

6.2 Aumento da capacidade de Memória

Conforme visto no capítulo 4 para armazenar 5 minutos de amostra de um sinal ECG com freqüência de amostragem de 1000Hz é necessário aproximadamente 293Kb de memória. Como o monitor cardíaco possui pouco mais de 2Kb de memória faz-se necessário um aumento na capacidade de memória de forma que ele reproduza o sinal ECG com mais precisão e detalhe e em um tempo considerável para que o médico consiga diagnosticar caso ocorra algum problema com o paciente.

Existem memórias no mercado com capacidades de folga para comportar muito mais que os cerca de 300Kb necessários para uma reprodução precisa do sinal ECG coletado do paciente. O grande problema é que estas memórias de grande capacidade com armazenamentos da ordem ou maiores que 1Mb exigem grande quantidade de pinos. Esta grande

quantidade de pinos torna o projeto do Monitor cardíaco inviável devido a escassez de pinos do microcontrolador.

Uma alternativa para aumentar a capacidade de memória do Monitor Cardíaco é então o uso de memórias seriais. As memórias seriais existentes são em sua maioria do tipo *EEPROM (Electrically Erasable Programable Read-Only Memory)* e necessitam de no máximo 4 pinos para funcionamento além da alimentação. As tecnologias mais difundidas de funcionamento das memórias seriais são os padrões *I2C* [Phi00] desenvolvido pela *Phillips* e o padrão *SPI* desenvolvido pela *Motorola*. O padrão *I2C* tem a vantagem de possuir apenas dois pinos de interface enquanto que o padrão *SPI* embora tenha 4 pinos de interface possui uma velocidade de comunicação maior que o *I2C*. Além de reduzir a quantidade de pinos necessários para a interface com o microcontrolador, estas memórias possuem capacidade de até 512Kb o que fica perfeitamente adequado em termos de necessidade para armazenamento detalhado de um sinal ECG no tempo de 5 minutos proposto.

O Microcontrolador ADUc834 conforme mencionado anteriormente no capítulo 2 possui as interfaces *SPI* e *I2C* embutidas o que permite que elas sejam facilmente incorporadas nesta nova proposta de projeto. Originalmente os pinos do microcontrolador alocados para a interface de memória foram disponibilizados no conector `CON6`. No novo circuito para utilização da memória serial optou-se por disponibilizar o funcionamento das duas tecnologias de interface, *I2C* e *SPI*. Isto é possível porque os pinos das memórias *SPI* e *I2C* possuem funções compatíveis que permitem a utilização dos mesmos pinos do microcontrolador. O microcontrolador por sua vez possui funções multiplexadas nos mesmos pinos para os dois tipos de interface. Além disto as memórias também são compatíveis no tamanho de encapsulamento.

Para que a memória serial funcione nos dois modos é necessário apenas a mudança da montagem de um resistor no pino 7. Quando a memória for do tipo *I2C*, o sinal do pino 7 é um controle de escrita *(WC)* que deve ser fixado em GND e portanto o resistor R25 deve ser montado e o resistor R24 não. Caso a memória for do tipo *SPI*, o sinal do pino 7 é um sinal de espera *(HOLD)* que deve ser colocado fixo em VCC e desta vez o resistor R24 deve ser montado e o resistor R25 não. Esta configuração pode ser observada no esquema elétrico mostrado na figura C.2 do anexo C.

6.3 Modificações no circuito de alimentação

Como a interface de rede sem fio *Compact Flash* é alimentada com 3,3V há necessidade de se mudar esta alimentação dos 5V originais para 3,3V. Para isto é usado um regulador de tensão que converte os 5V para 3,3V. Um circuito que corta a alimentação do cartão *Compact Flash* também é previsto de forma que seja possível economizar a energia que seria utilizada para alimentar o cartão nos momentos em que ele não é utilizado. O sinal `Enable_CF` mostrado na tabela 6.1 é encarregado de comandar o desligamento da alimentação do cartão CF.

6.4 Considerações sobre a Comunicação por Rede sem Fio

A utilização de um conector padrão *Compact Flash* no projeto do monitor cardíaco permite que sejam utilizados cartões com funcionalidades diversas. Neste trabalho foram considerados o cartão *Compact Flash* como um dispositivo armazenador de dados no modo memória e também como um dispositivo de comunicação de rede sem fio através da utilização do modo *I/O* com um cartão de rede sem fio padrão 802.11b.

Por causa das características apresentadas no capítulo 2 optou-se pelo padrão 802.11b embora poderia ter sido usado também o padrão 802.11g que possui maior velocidade de comunicação. Para utilização do padrão 802.11g não é necessário o desenvolvimento de um novo hardware para suportar este padrão visto que foram incorporados ao novo projeto os pinos anteriormente não previstos para o funcionamento no modo *I/O*. Portanto para incorporação deste padrão de comunicação ao projeto do Monitor Cardíaco é necessário apenas o desenvolvimento de um novo *driver* de dispositivo que faça a comunicação entre a pilha *TCP/IP* e o hardware do cartão 802.11g.

Outro padrão de comunicação de rede sem fio que futuramente pode ser incorporado ao Monitor Cardíaco caso os fabricantes produzirem um cartão *Compact Flash* compatível com ele é o *IEEE* 802.16 [IEE04, Ole05] ou como é também chamado *WiMAX(Worldwide Interoperability for Microwave Access)*. Este protocolo possui dentre outras características que a torna mais atrativa em relação ao padrão 802.11, um alcance da ordem de kilômetros. A proposta deste protocolo é a interconexão de Redes Metropolitanas e portanto é otimizado para ambientes externos. Em relação a escalabilidade, permite reuso de frequência e diferentes larguras de faixa e portanto a incorporação de vários canais de comunicação. A velocidade de comunicação chega a 100Mbps para canais de 20MHz.

O grande problema que levou o protocolo 802.16 a não ser ainda avaliado para incorporação no projeto do Monitor Cardíaco previamente foi que a tecnologia ainda se encontra em fase de aceitação pelos mercados mundiais. No Brasil, a tecnologia vem sendo testada em várias cidades e a previsão de uso está marcada para o final de 2006. Além de estar ainda com aceitação precoce, existem ainda poucos produtos disponíveis no mercado e não foi encontrado até a publicação deste trabalho, um cartão *Compact Flash* compatível com este protocolo. Acredita-se que uma vez existindo tal cartão será possível facilmente adaptar a utilização deste protocolo de rede sem fio ao projeto do Monitor Cardíaco necessitando-se apenas do desenvolvimento de um *driver* de dispositivo para o cartão.

Capítulo 7

Conclusões e Planos Futuros

7.1 Conclusões

Considerando que inicialmente o projeto base para os trabalhos não tinha sido projetado para o funcionamento para uma rede sem fio, a meta de se transformar ou desenvolver um circuito que atendesse aos pré requisitos do projeto foi desafiadora e motivou a busca por soluções que contornassem o problema mantendo-se o mesmo hardware elaborado. Foi necessário um grande esforço e exploração dos recursos de hardware e software, principalmente do microcontolador e da interface de rede sem fio para que a maioria do que foi proposto pudesse ser implementada.

Considera-se desta forma que os resultados obtidos no desenvolvimento dos trabalhos foram muito satisfatórios. Conseguiu-se mostrar através do capítulo 5 que é possível para um projeto até então limitado em termos de comunicação ser integrado à Internet ainda que não tenha sido projetado para tal. Algumas modificações ainda são necessárias para o completo funcionamento do sistema mas estas agregam pouco tamanho ao circuito e representam um custo irrisório diante do resto do projeto.

As aplicações práticas do projeto foram testadas e comprovaram que ele pode ser utilizado como um sistema de monitoramento remoto de pacientes necessitando de alguma pequena melhoria no circuito de alimentação. O *Device Driver* desenvolvido em conjunto com a pilha *TCP/IP* utilizada permitiram integrar o Monitor Cardíaco de modo simples e eficiente à Internet podendo este software desenvolvido ser utilizado para outros sistemas embutidos que demandem necessidades semelhantes ao deste projeto.

7.2 Planos e trabalhos futuros

Alguns aspectos não foram cobertos no desenvolvimento deste trabalho e desta forma surgem como oportunidade para exploração em trabalhos futuros. A seguir são apontadas

as principais oportunidades para continuidade dos trabalhos e algumas oportunidades de pesquisa:

- **Segurança:** Apesar de oferecer suporte à segurança através dos protocolos *WEP(Wired Equivalent Privacy)* e *WPA(Wi-Fi Protected Access)*, não foi objetivo de trabalho explorar os aspectos de segurança da aplicação o que desta forma contribui com um importante aspecto a ser pesquisado e implementado.

- **Implementação de aplicações para tratamento e disponibilização do dados enviados pelo monitor cardíaco:** o trabalho se preocupou mais com o lado cliente da comunicação e não foi elaborado um sistema para tratamento dos dados recebidos pelo servidor. Este sistema poderia ser implementado de forma a poder disponibilizar os dados através de gráficos de *ECG* e poder tratar dados enviados de vários pacientes.

- **Implementação de outros protocolos da família *TCP/IP*:** Para ser utilizado em uma rede sem fio sem a necessidade de um IP fixo é necessário a implementação do protocolo *DHCP*. Outros protocolos de aplicação tais como o *SMTP* podem ser implementados permitindo, por exemplo, o envio de emails diretamente para o médico do paciente.

- **Melhoria do Circuito de alimentação:** é necessário rever o circuito de alimentação para que consiga suportar corretamente o consumo da placa *ECG*.

- **Uso de *ASIC* atualizado:** Verificar alternativa de uso de outro *ASIC* mais avançado, com mais recursos e que consuma menos energia uma vez que o *ASIC* utilizado no projeto do monitor cardíaco pode encontrar-se obsoleto.

Devido ao reduzido tempo para fechamento dos trabalhos, não foi possível montar e testar a nova versão de projeto do monitor cardíaco que incorpora as funções de aquisição das amostras de *ECG*. Isto impediu que resultados mais reais fossem mostrados e analisados. Como continuidade dos trabalhos, pretende-se implementar o circuito projetado, testar e analisar os dados para permitir que novas conclusões sejam tiradas podendo se ter conteúdo de pesquisa suficiente para publicação de artigos em congressos e conferências afins com a área do trabalho desenvolvido.

Referências Bibliográficas

[apa] The apache software foundation. Disponível em http://www.apache.org.

[AS] Inc AbsoluteValue Systems. linux-wlan. http://www.linux-wlan.org/.

[Ben02] Jeremy Bentham. *TCP/IP Lean Web Servers for Embedded Systems*. CMP Books, 2nd edition, 2002.

[Ben03] Etienne Beneteau. Web tcp/ip solutions with atmel c51 flash microcontrollers. Technical report, Atmel, 2003.

[Bra02] J. Brady. Build your own 8051 web server. *Circuit Cellar*, (146), September 2002.

[Can01] Tom Cantrell. I-way the hard way. *Circuit Cellar*, October 2001.

[Can02] Tom Cantrell. I-way the hard(ware) way. *Circuit Cellar*, May 2002.

[Com03] Compact Flash Association. *CF+ and Compact Flash Specification Revision*, 2.0 edition, Maio 2003. Disponível em http://www.compactflash.org.

[Con01] Júlio César Dillinger Conway. Monitor de sinais vitais multiparamétrico vestíveis, July 2001.

[Cor98] Intel Corporation. *Embedded Microcontrollers - Databook*, January 1998.

[Cor03] BroadCom Corporation. Ieee802.11g - the new mainstream wireless lan standard. *Disponível em http://www.54g.org*, July 2003.

[Cyl04] I. Cyliax. Wi-fi sunlogger. *Circuit Cellar*, (172), November 2004.

[Dan04] A. Dannenberg. Slaa137a - msp430 internet connectivity. *Texas Instruments*, February 2004.

[Dev03] Analog Devices. *ADuC834 Datasheet*, 2003.

[DK02] Brian Senese David Kammer, Gordon McNutt. *Bluetooth Application Developers Guide*. Syngress Publishing, Inc., 2002.

[DM04] Matt Welsh et al David Malan, Thaddeus Fulford-Jones. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. *International Workshop on Wearable and Implantable Body Sensor Networks*, April 2004.

[Dum03] A. Dumkels. *uIP 0.9 Reference Manual*, July 2003.

[dun] The uip tcp/ip stack embedded for microcontrollers. http://www.sics.se/ adam/uip/.

[Dun03] Adam Dunkels. Full tcp/ip for 8-bit architectures. Proceedings of the first international conference on mobile applications, systems and services (MOBISYS), 2003.

[Ead04] F. Eady. Tcp/ip stack solution - a detailed look at the cmx-micronet. *Circuit Cellar*, (172), November 2004.

[Ead05] F. Eady. Embedded wi-fi with trendnet. *Circuit Cellar*, (174), January 2005.

[Fer05] Geraldo Antônio Ferreira. Aplicações midp em aparelhos móveis celulares e monitoramento remoto de bio-sinais: considerações e desenvolvimento de uma solução, August 2005.

[FLCJ03] C. J. N. Coelho F. L. C. Junior. Monitor cardíaco. Technical report, DCC/UFMG, 2003.

[For05] Débora Fortes. A explosão das redes sem fio. *Revista Info especial WI-FI Coleção 2005*, page 14, 2005.

[Gas02] Matthew Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly, 2002.

[Ger03] Vadim Gerasimov. Every sign of life, June 2003.

[IEE99] IEEE. *Ansi/IEEE Std 802.11. Part 11 - Wireless Lan Medium Access Control (MAC) and Physical Layer (Phy) Specification*, 1999.

[IEE04] IEEE. *IEEE Standard for local and Metropolitan Area NetworksPart 16:Air Interface for Fixed Broadband Wireless Access Systems*, 2004 edition, October 2004.

[IM02] IBM and Montavista. Dynamic power management for embedded systems, November 2002.

[Inc01] Intersil Americas Inc. *HFA3842 Data Sheet*, June 2001.

[Inc02] Intersil Americas Inc. *PRISM Driver Programmers Manual*, 2.30 edition, June 2002.

[kad] Kwiknet tcp/ip stack. http://www.kadak.com/tcp_ip/tcpip.htm.

[KL04] et al Konrad Lorincz, David Malan. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing, Special Issue on Pervasive Computing for First Response*, Oct-Dec 2004.

[Loe99a] M. Loewen. An724 - using picmicro mcus to connect to internet via ppp. *Microchip Technology Inc*, 1999.

[Loe99b] M. Loewen. Internet appliance interface. *Circuit Cellar*, (108), July 1999.

[Man98] S. Mann. Definition of wearable computer. *http://wearcomp.org/wearcompdef.html*, May 1998.

[Max04] Maxim. *DS80C400 Network Microcontroller*, 2004.

[Mil01] Michael Miller. *Discovering Bluetooth*. Sybex Inc, 2001.

[Mul00] Nathan J. Muller. *Bluetooth Demystified*. McGraw-Hill, 2000.

[mur] Keil c51/ 8051 port of adam dunkels' uip v0.9 tcp/ip stack. http://members.iinet.net.au/ṽanluynm/.

[Ole05] Ron Olexa. *Implementing 802.11, 802.16, and 802.20 Wireless Networks*. Elsevier Inc, 2005.

[pcma] Linux pcmcia information page. http://pcmcia-cs.sourceforge.net/.

[PCMb] PCMCIA. Pcmcia home page. http://www.pcmcia.org.

[Pea05] G. Peacock. Ip and ethernet interfaces. February 2005.

[Pen04] Alex Pentland. Healthwear:medical technology becomes wearable. *Computer Magazine*, (37), May 2004.

[Phi00] Phillips Semiconductors. *The I 2C-Bus Specification*, 2.1 edition, January 2000.

[Raj02] N. Rajbharti. The microchip tcp/ip stack. *Microchip Technology Inc*, 2002.

[RD03] Jonathan Gips Alex Pentland Rich DeVaul, Michael Sung. Mithril 2003: Applications and architecture. 7th IEEE International Symposium on Wearable Computers (ISWC), 2003.

[Req89] Request for Comment. *RFC 1122 - Requirements for Internet Hosts - Communication Layers*, October 1989.

[Req94] Request for Comment. *RFC 1661 - The Point-to-Point Protocol (PPP)*, Julho 1994.

[RR00] Steve Humberd Rodger Richey. Embedding picmicro microcontrollers in the internet. *Microchip Technology Inc*, 2000.

[SBA+01] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 524–529. ACM Press, 2001.

[Shr] H. Shrikumar. Ipic - a match head sized webserver. Technical report, CS/UMASS. Disponível em http://www-ccs.cs.umass.edu/ shri/iPic.html.

[SIG04] Bluetooth SIG. *Specification of The Bluetooth System - V2.0 +EDR*, November 2004.

[SVGM00] Tajana Simunic, Haris Vikalo, Peter Glynn, and Giovanni De Micheli. Energy efficient design of portable wireless systems. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 49–54. ACM Press, 2000.

[Tec01] Welch Allyn OEM Technologies. *ECG ASIC Datasheet*, May 2001.

[Tou04] Jean Tourrilhes. *Linux Wireless LAN Howto*, August 2004.

[TRFFJW04] Gu-Yeon Wei Thaddeus R. F. Fulford-Jones and Matt Welsh. A portable, low-power, wireless two-lead ekg system. pages 2141–2144. 26th IEEE EMBS Annual International Conference, 2004.

[uBM99] T. Šimunić, L. Benini, and G. D. Micheli. Event-driven power management of portable systems. In *International Symposium on System Synthesis*, pages 18–23, 1999.

[xbo] Motes, smart dust sensors, wireless sensor networks. http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[YHL00] Giovanni De Micheli et al Yung-Hsiang Lu, Tajana Simunic. Quantitative comparison of power management algorithms. In *Design Automation and Test in Europe*, pages 20–26. Stanford University, March 2000.

[YHL01] Giovanni de Micheli Yung-Hsiang Lu. Comparing system-level power management policies. *IEEE Design & Test of Computers*, pages 10–19, March 2001.

Apêndice A

Integração da pilha *TCP/IP* com o código do Programa

A.1 Estrutura e modificação dos arquivos da Pilha *TCP/IP uIP*

Todo o código da pilha *uIP* reaproveitado no desenvolvimento do trabalho é dividido nos arquivos conforme a seguir:

- `uip.c`: implementa os protocolos *TCP/IP*, o protocolo ICMP e ainda o protocolo UDP;
- `uip_arp.c`: implementa o protocolo *ARP* com a criação e manutenção das tabelas de endereço *MAC*;
- `uip_arch.c`: implementa as funções para adição de 32 bits e cálculo de *checksums*;
- `uip_opt.h`: arquivo de configuração da pilha *uIP* onde é possível configurar tamanho máximo de *buffer* de dados, endereço IP da aplicação dentre outros.

Em [Dum03] é possível obter todas as informações sobre como são implementados as diversas características dos protocolos *TCP/IP* na pilha *uIP*, descrição detalhada de cada um dos arquivos acima e também como implementar algumas aplicações para serem utilizadas em conjunto com a pilha *uIP* tais como um servidor *HTTP*, servidor *SMTP* e outros. Em [dun] é possível consultar todas as informações necessárias para implementação e integração da pilha *TCP/IP uIP* a qualquer projeto incluindo *download* do código fonte, projetos da pilha com portabilidade para diferentes microcontroladores, necessidades de memória, publicações relacionadas e outros.

Para suporte à interface de rede sem fio foi necessário incluir a seguinte estrutura no arquivo uip_arp.h:

```
struct uip_gen_snap {
  u8_t dsap;
  u8_t ssap;
  u8_t control;
  u8_t oui_1;
  u8_t oui_2;
  u8_t oui_3;
  u16_t type;
};
```

Esta estrutura foi necessária porque na recepção dos dados, após o cabeçalho *Ethernet* era também recebido a estrutura *SNAP(Sub-Network Access Protocol)* e para um processamento correto dos dados recebidos era preciso desconsiderar o tamanho desta estrutura. Para a transmissão dos dados, após o cabeçalho *Ethernet*, foi também necessário incluir a estrutura *SNAP* no *buffer* global `uip_buf[]` para que os dados fossem corretamente interpretados pelo Ponto de Acesso. O valor do *buffer* global foi preenchido com valores constantes obtidos a partir da análise dos dados recebidos quando foram executados testes com o Monitor Cardíaco funcionando com uma aplicação de servidor *Web*.

Outras modificações relevantes foram feitas no arquivo `uip_opt.h` de configuração da pilha *uIP*. Para o Monitor Cardíaco funcionar como cliente foi necessário alterar o estado do parâmetro `UIP_ACTIVE_OPEN` para o valor igual a 1. Para setar o número de conexões suportadas pela aplicação foi necessário alterar o valor do parâmetro `UIP_CONNS`. O parâmetro `UIP_BUFSIZE` teve o valor alterado para 1500 que corresponde ao tamanho do *buffer* global de dados `uip_buf[]`. Como comentado anteriormente, é necessário considerar a estrutura *SNAP* para uma correta interpretação dos dados e por isto o parâmetro `UIP_LLH_LEN` que indica o tamanho do cabeçalho do nível de ligação teve que ser acrescido de 8 em relação ao tamanho do cabeçalho *Ethernet* ficando com um total de 22. Um outro parâmetro acrescido foi o nome do arquivo cabeçalho da aplicação que no caso da aplicação cliente foi `webclient.h`.

A.2 Integração da pilha *uIP* com o protocolo de Interface de rede

A interface da pilha *uIP* com o protocolo de rede sem fio e com a aplicação é feita através da função principal *main*. Todas as três principais funções da interface de rede sem fio descritas no capítulo 4 são chamadas a partir da função *main*. Esta função é estruturada

de acordo com a aplicação que se deseja rodar em cima da pilha *TCP/IP* sendo que o seu princípio básico de funcionamento gira em torno de um loop infinito que periodicamente checa o estado de algumas variáveis e de acordo com este estado, as funções da pilha *TCP/IP*, da interface de rede sem fio ou da aplicação são chamadas para processamento das informações.

A função *main* implementa a máquina de estados descrita no capítulo 4 para otimizar o consumo de energia e as três funções principais da interface de rede sem fio são invocadas a partir do estado *ocupado*. Um pseudo código do estado *ocupado* contido na função *main* é mostrado a seguir com as funções do protocolo de rede sem fio colocadas entre parentesis. O código da função *main* em C pode ser observado com mais detalhes no anexo B.

```
estado Ocupado:
{
   Se botão da ECG pressionado então
   {
      Inicializa Driver do dispositivo de rede sem fio (wifidev_init());
      Desabilta Interrupção Timer 0;
      Chama aplicação;
   }
   Verifica se existem dados lidos da Interface de rede sem Fio (uip_len = wifidev_read());
   Se não existem dados lidos da interface de rede sem fio então
   {
      Para(i = 0; i < Num de conexões; i++)
      {
         Verifica estado da conexão TCP/IP (uip_periodic(i));
         Se existem dados da pilha TCP/IP para serem transmitidos então
         {
            Chama função de formatação de cabeçalho Ethernet para envio de dados;
            Chama função de envio de dados pela Interface de rede sem Fio (wifidev_send());
         }
      }
   }
   Senão (existem dados lidos da Interface de rede sem fio)
   {
      Se os dados são uma pacote com o Tipo de protocolo IP então
      {
         Armazena endereço MAC na tabela ARP ;
         Processa pacote de dados IP (uip_input());
         Se existem dados da pilha TCP/IP para serem transmitidos então
         {
            Chama função de formatação de cabeçalho Ethernet para envio de dados;
            Chama função de envio de dados pela Interface de rede sem Fio (wifidev_send());
         }
      }
      Senão se os dados são uma pacote com o Tipo de protocolo ARP então
      {
       Processa pacote de dados ARP;
         Se existem dados do protocolo ARP para serem transmitidos então
         {
```

```
            Chama função de envio de dados pela Interface de rede sem Fio (wifidev_send());
        }
      }
   }
}
```

Para suporte à interface de rede sem fio a pilha *uIP* implementa duas funções: `uip_input()` e `uip_periodic()`. A função `uip_input()` deve ser chamada pelo driver de dispositivo quando um pacote IP foi recebido e colocado no *buffer* global `uip_buf[]`. A função `uip_input()` vai processar o pacote e quando ela retornar um pacote de saída pode ter sido colocado no *buffer* global `uip_buf[]`. O *driver* de dispositivo deve enviar este pacote para a rede. É possível saber se existem dados a serem enviados após o retorno da função `uip_input()`, através da checagem do tamanho dos dados indicado pela variável `uip_len`. Esta variável possui valor maior que zero caso existir dados a serem enviados. O uso da função `uip_input()` é indicado no pseudo código da função *main* mostrado anteriormente.

A função `uip_periodic()` deve ser invocada periodicamente uma para cada conexão pelo *driver* de dispositivo. Esta função é usada pela pilha *uIP* para gestionar temporizadores dos protocolos e retransmissões. Quando ela retornar pode ter colocado dados a serem enviados no *buffer* global de pacotes `uip_buf[]`. O uso da função `uip_periodic()` também é indicado no pseudo código da função *main* mostrado anteriormente.

A.3 Integração da pilha *uIP* com a aplicação

A integração da pilha *uIP* com a aplicação é feita de maneira muito simples através do rótulo `UIP_APPCALL`. Este rótulo é chamado pela função principal *main* na aplicação cliente desenvolvida para o Monitor Cardíaco e também nas várias funções da pilha *TCP/IP uIP* implementadas no arquivo `uip.c`. As chamadas da pilha *TCP/IP uIP* são feitas ao rótulo `UIP_APPCALL` todas as vezes que ocorre um evento. Cada evento tem uma função de teste correspondente que é usada para distinguir diferentes eventos. Estas funções são implementadas como macros em C e possuem valor binário.

Para o correto funcionamento da aplicação é necessário que seja implementado uma função que leve o rótulo `UIP_APPCALL`. Esta função é a porta de entrada de integração da pilha *TCP/IP* com a aplicação. Ela monitora os vários eventos da aplicação junto a pilha *uIP* tais como o recebimento de dados, retransmissão de dados, fechamento de conexões, relatório de erros, etc.

Caso a função de teste `uip_newdata()` for igual a um, o *host* remoto da conexão enviou um novo dado. O ponteiro `uip_appdata()` aponta para este novo dado recebido. O tamanho do pacote de dados recebido é indicado através da função `uip_datalen()`. O dado não é armazenado pela pilha *uIP* e pode ser sobre escrito depois que a função de aplicação `UIP_APPCALL` retornar. Desta forma a aplicação deve atuar diretamente no dado recebido ou ela mesmo deve copiar o dado recebido em um *buffer* de armazenamento para processamento posterior.

A aplicação envia dados usando a função `uip_send()`. Esta função possui dois argumentos que são o ponteiro para os dados a serem enviados e o tamanho do pacote de dados. A aplicação pode enviar apenas um pacote de dados em determinado momento da conexão e não é possível chamar a função `uip_send()` mais que uma vez quando a aplicação é acionada sendo apenas enviado o dado referente a última chamada a esta função.

A retransmissão de dados é comandada por um temporizador periódico TCP. Cada vez que o temporizador periódico é acionado, o temporizador de retransmissão para cada conexão é decrementado. Se o temporizador atinge o valor zero, a retransmissão dos dados deve ser feita. Como a pilha *uIP* não rastreia nenhum conteúdo de pacote depois que foram enviados para o *driver* de dispositivo, ela necessita que a aplicação tenha grande participação na execução da restransmissão. Quando a pilha *uIP* decide que um pacote de dados deve ser retransmitido, a função da aplicação `UIP_APPCALL` é chamada com o *flag* `uip_rexmit()` setado, indicando que a retransmissão é necessária.

A aplicação fecha a conexão corrente chamando `uip_close()` durante a chamada da aplicação. Para indicar um erro fatal, a aplicação pode querer abortar a conexão e isto pode ser feito chamando a função `uip_abort()`. Se a conexão foi fechada pelo *host* remoto, a função de teste `uip_closed()` retorna um valor lógico verdadeiro.

Existem dois erros fatais que podem acontecer a uma conexão. Um deles ocorre quando a conexão é abortada pelo *host* remoto e o outro ocorre quando a conexão tiver sido abortada por causa da retransmissão do último dado muitas vezes. A aplicação pode usar duas funções de teste `uip_aborted()` e `uip_timedout()` para testar estas condições de erro.

Quando a conexão está ociosa, a pilha *uIP* verifica a aplicação cada vez que o *timer* periódico expira. A aplicação pode usar a função de teste `uip_poll()` para checar se está sendo verificada pela pilha *uIP*. Este evento de verificação em dois propósitos sendo que o primeiro é informar para a aplicação periodicamente que a conexão está ociosa o que pode permitir que a aplicação possa fechar conexões que se tornarem ociosas por muito tempo. O outro propósito é deixar que a aplicação envie um dado novo assim que ele tenha sido gerado. A aplicação pode enviar dado quando invocado pela pilha *uIP* e desta forma o evento de verificação `uip_poll()` é a única forma de enviar dados em uma conexão ociosa.

Para escutar uma porta a pilha *uIP* mantém uma lista de portas *TCP* a serem ouvidas. Uma nova porta é aberta para ser escutada através da função `uip_listen()`. Quando uma requisição de conexão chega a uma porta que esta sendo escutada, a pilha *uIP* cria uma nova conexão e chama a função de aplicação `UIP_APPCALL`. A função de teste `uip_connected()` é verdadeira se a aplicação receber um *ACK* depois de ter enviado um *SYN+ACK* para o host remoto. A aplicação pode checar o campo `lport` da estrutura `uip_conn` para checar em qual porta a nova conexão foi conectada.

Novas conexões podem ser abertas partindo da pilha *uIP* através da função `uip_connect()`. Esta função aloca uma nova conexão e seta um *flag* no estado da conexão que irá abrir uma conexão *TCP* no endereço *IP* e porta na próxima vez que a conexão é verificada pela pilha *uIP*. A função `uip_connect()` retorna um ponteiro para a estrutura `uip_conn` para a nova conexão. Se não existem conexões disponíveis, a função retorna nulo.

Apêndice B

Código dos programas implementados

A seguir são apresentados os códigos dos programas implementados. Basicamente pode-se dividir os códigos implementados em *driver* de dispositivo e aplicações médica e de rede. A função *main* que faz interface com a aplicação e com o *driver* de dispositivo de rede sem fio também é apresentada. O módulo que implementa a interface de memória da *Compact Flash* são também mostrados a seguir. Por questões de simplificação o código da pilha *uIP* não é apresentado podendo ser encontrado em [dun, mur].

Todos os códigos foram desenvolvidos utilizando a linguagem C e o compilador Keil C51 versão 6.23a.

B.1 Código da Função principal *main*

```
/*********************************************************************
* Filename: main.h
* Description: This file implements the header for the main program function
* Developed by: Marco Carvalho
* Created on: 30/03/05
/*********************************************************************/

#ifndef MAIN_H
#define MAIN_H

#include "uip.h"
#include "wifidev.h"
#include "uip_arp.h"
#include "ecg_app.h"

#define BUF ((struct uip_eth_hdr *)&uip_buf[0])
```

```
#define CF_RETRY_TIMES 3

#ifndef NULL
#define NULL (void *)0
#endif /* NULL */

#endif /* MAIN_H */

/*********************************************************************
* Filename: main.c
* Description: This file implements the main program function
* Developed by: Marco Carvalho
* Created on: 30/03/05
*********************************************************************/

#include "main.h"

void main(void)
{
    u8_t xdata i, arptimer;

    /* Initialise the uIP TCP/IP stack. */
    uip_init();

    /* Initialise the app. */
    webclient_init();

    /* Config chip and Initialise the device driver. */
    config_chip();

    /* Initialise the ARP cache. */
    uip_arp_init();

    arptimer = 0;

    /* Find cF mode */
    i = 0;
    do
    {
        find_CF_mode();
        i++;
    }
    while((cis_device == CISTPL_FUNCID_NOT_SUPPORTED)&&(i < CF_RETRY_TIMES));

    /* if device not supported, program stopped */
    if(cis_device == CISTPL_FUNCID_NOT_SUPPORTED)
    {
        printf("Device not Supported. Please put a valid CF card and try again.\n");
        //RED_ON();
```

```
        while(1);
    }

    /* main loop */
    while(1)
    {
        switch(ecg_state)
        {
            case ECG_STATE_DPM_BUSY: //TCP/IP stack running
            {
                /* if the ecg button was pressed call the application to open a connection*/
                if(ecg_send_button)
                {
                    /*init device driver*/
                    wifidev_init();
                    /*disable IRQ0 IRQ */
                    DISABLE_INT0_IRQ();
                    printf("Call application\n");
                    UIP_APPCALL();
                }

                uip_len = wifidev_read();

                if(uip_len == 0)
                {
                    for(i = 0; i < UIP_CONNS; i++)
                    {
                        uip_periodic(i);
                        /* If the above function invocation resulted in data that
                           should be sent out on the network, the global variable
                           uip_len is set to a value > 0. */
                        if(uip_len > 0)
                        {
                            uip_arp_out();
                            wifidev_send();
                        }
                    }

#if UIP_UDP
                    for(i = 0; i < UIP_UDP_CONNS; i++)
                    {
                        uip_udp_periodic(i);
                        /* If the above function invocation resulted in data that
                           should be sent out on the network, the global variable
                           uip_len is set to a value > 0. */
                        if(uip_len > 0)
                        {
                            uip_arp_out();
                            wifidev_send();
                        }
```

```
                    }
#endif /* UIP_UDP */

                    /* Call the ARP timer function every 10 seconds. */
                    if(++arptimer == 20)
                    {
                        uip_arp_timer();
                        arptimer = 0;
                    }
                }
                else /* (uip_len != 0) Process incoming */
                {
                    if(BUF->type == htons(UIP_ETHTYPE_IP))
                    {
                       uip_arp_ipin();
                       uip_input();
                       /* If the above function invocation resulted in data that
                          should be sent out on the network, the global variable
                          uip_len is set to a value > 0. */
                       if(uip_len > 0)
                       {
                           uip_arp_out();
                           wifidev_send();
                       }
                    }
                    else if(BUF->type == htons(UIP_ETHTYPE_ARP))
                    {
                        uip_arp_arpin();
                        /* If the above function invocation resulted in data that
                           should be sent out on the network, the global variable
                           uip_len is set to a value > 0. */
                        if(uip_len > 0)
                        {
                            wifidev_send();
                        }
                    }
                }
        break;
            }//ECG_STATE_DPM_BUSY

            case ECG_STATE_DPM_IDLE:
            {
                ENABLE_IDLE_MODE();
                if(tick_count == 1)
                {
                    /*if CF card = Network, reset card */
                    if (cis_device == CISTPL_FUNCID_NETWORK)
                        CF_RESET_ON();

                    printf("Idle mode activated\n");
```

```
#if PLL_DPM
                    PLLCON = PLLCON | 0x07; //CPU clk = 0,09MHz
#endif

                }
                /* if memory mode, write one sector if it is full */
                if (cis_device == CISTPL_FUNCID_FIXED)
                {
                    /* if one sector is full, write it in CF memory */
                    if (ecg_index == BYTE_PER_SEC)
                    {
                        printf("Write 512 bytes in the sector 0x%Lx\n", lba_address);
                        if (lba_address == CFNumSectors)
                        {
                            lba_address =0;
                        }
                        else
                        {
                            DISABLE_TMR0_IRQ();
                            if(CFWriteSector(lba_address++))
                                printf("CFWriteSector command error \n");
                            ENABLE_TMR0_IRQ();
                        }
                    }
                }

                break;
            }

            case ECG_STATE_DPM_PWRDWN:
            {
                CF_RESET_ON();
                ENABLE_INT0_PD(); //enable Int0 power down bit
                ENABLE_OSC_PD(); //enable oscillator power down bit
                INTVAL = 0x1e; // config timer interval bit to 30 seconds
                //CONFIG_INTERVAL_COUNTER_HOURS();
                //CONFIG_INTERVAL_COUNTER_MINUTES();
                CONFIG_INTERVAL_COUNTER_SECONDS();
                ENABLE_TIMER_INTERVAL_COUNTER(); // Enable timer interval to count
                ENABLE_TIC_IRQ();                // Enable tick IRQ
                printf("Power Down mode activated\n");
#if PLL_DPM
                PLLCON = PLLCON | 0x07; //CPU clk = 0,09MHz
#endif
                ENABLE_POWER_DOWN();
                break;
            }
        }//switch(ecg_state)
    }//while(1)
```

```
    return;
}
```

B.2 Código do Driver de dispositivo de Rede sem Fio

```
/*********************************************************************
* Filename: wifidev.h
* Description: This file implements the header file for the prism device
*              driver for 8051 microcontroller
* Developed by: Marco Carvalho
* Created on: 09/03/05
*********************************************************************/

#include "uip.h"
#include "uip_arp.h"
#include "Cfdev.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <intrins.h>

/* Turn to 1 if you want simulate */
#define SIMULATE 0

/*control variable to test PLL DPM */
#define PLL_DPM 0

// Change WIFI_CPU_XTAL to match hardware
#if PLL_DPM
#define WIFI_CPU_XTAL          98304
#else
#define WIFI_CPU_XTAL       12580000             // 8051 crystal freq in Hz
#endif

#define CISTPL_FUNCID_NETWORK       0x06
#define CISTPL_FUNCID_MEMORY        0x01
#define CISTPL_FUNCID_FIXED         0x04
#define CISTPL_FUNCID_NOT_SUPPORTED 0x00
// Crystal/ Baud rate combinations marked * are well supported with 100.0%
// timing accuracy.
/*                             Xtal Frequency (MHz)

                    3.6864     11.0592     18.4320     22.1184
                ÚÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄ¿
            300 ³      *      ³      *      ³      *      ³      *      ³
                ÃÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄ´
            600 ³      *      ³      *      ³      *      ³      *      ³
                ÃÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄ´
```

```
        1200 3     *     3     *     3     *     3     *     3
             ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
  BAUD  2400 3     *     3     *     3     *     3     *     3
  RATE       ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
        4800 3     *     3     *     3     *     3     *     3
             ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
        9600 3     *     3     *     3     *     3     *     3
             ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
       14400 3     _     3     *     3     _     3     *     3
             ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
       19200 3     _     3     _     3     _     3     *     3
             ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ            */


// Low Baud rate (SCON = 0), X1 CPU mode timing parameters.
// Adjust XTAL and UART_BPS to suit your application and hardware.
#define XTAL                 WIFI_CPU_XTAL      // Crystal freq in Hz

#if PLL_DPM
#define UART_BAUD            1200               // Tranceiver baud rate
#else
#define UART_BAUD            4800               // Tranceiver baud rate
#endif

#define T1_CLOCK             XTAL / 12          // Timer 1 mode 2 clock rate
#define T1_RELOAD            256 - ((T1_CLOCK / 32) / UART_BAUD)

/*  ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
    º                            Private defines.                           º
    ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼  */
#define WIFI_CPU_CLOCK      WIFI_CPU_XTAL / 12    // 8051 clock rate (X1 mode)

// Delay routine timing parameters
#define WIFI_DELAY_CONST    9.114584e-5          // Delay routine constant
#define WIFI_DELAY_MULTPLR  (unsigned char)(WIFI_DELAY_CONST * WIFI_CPU_CLOCK)

// X1 CPU mode timing parameters
#define WIFI_T0_CLOCK            WIFI_CPU_XTAL / 12 // Timer 0 mode 1 clock rate
#define WIFI_T0_INT_RATE         24                // Timer 0 intrupt rate (Hz)
#define WIFI_T0_RELOAD           65536 - (WIFI_T0_CLOCK / WIFI_T0_INT_RATE)

/*********************** Defines CF Memory ***********************/

/*
 * Configuration Option Register (COR)
 */
#define COR_CONFIG_MASK 0x3f
#define COR_MFC_CONFIG_MASK 0x38
#define COR_FUNC_ENA 0x01
#define COR_ADDR_DECODE 0x02
```

```
#define COR_IREQ_ENA 0x04
#define COR_LEVEL_REQ 0x40
#define COR_SOFT_RESET 0x80
#define COR_SOFT_SET        0x00

/*
 * CIS Tuple Codes
 */
#define CISTPL_DEVICE              0x01
#define CISTPL_DEVICE_A            0x17
#define CISTPL_DEVICE_OC           0x1c
#define CISTPL_DEVICE_OA           0x1d
#define CISTPL_VERS_1              0x15
#define CISTPL_MANFID              0x20
#define CISTPL_FUNCID              0x21
#define CISTPL_FUNCE               0x22
#define CISTPL_CONFIG              0x1a
#define CISTPL_CFTABLE_ENTRY       0x1b
#define CISTPL_END                 0xff

//Pins
#define CF_DATA_PORT    P0  // leitura/escrita de dados
#define CF_DATA_PORT_MASK 0xFF

// CF ADDR Port
#define CF_ADDR_PORT_LOW  P2     //portos de enderecamento LSB dos registros da CF( A5..A0 )
#define CF_ADDR_PORT_LOW_MASK 0x30 //00110000

#define CF_ADDR_PORT_HIGH  P3      //portos de enderecamento MSB dos registros da CF( A8..A6 )
#define CF_ADDR_PORT_HIGH_MASK 0x37 //00110111
// OBS: A9=A8 via hardware, A10=GND

#define CF_CS   CS_DSK
#define CF_CS_ON()   CF_CS=1;
#define CF_CS_OFF()  CF_CS=0;

#define CF_RD   RD
#define CF_RD_ON()   CF_RD=1;
#define CF_RD_OFF()  CF_RD=0;

#define CF_WR   WR
#define CF_WR_ON()   CF_WR=1;
#define CF_WR_OFF()  CF_WR=0;

/* IO control */
#define CF_REG       REG
#define CF_REG_ON()  CF_REG=1;
#define CF_REG_OFF() CF_REG=0;

#define CF_IOWR       IOWR
```

```
#define CF_IOWR_ON()  CF_IOWR=1;
#define CF_IOWR_OFF() CF_IOWR=0;

#define CF_IORD       IORD
#define CF_IORD_ON()  CF_IORD=1;
#define CF_IORD_OFF() CF_IORD=0;

#define CF_RESET       RESET
#define CF_RESET_ON()  CF_RESET=1;
#define CF_RESET_OFF() CF_RESET=0;

#define CF_CD1      CD1
#define NOP() _nop_ (); //função intrínsica de NOP

#define ON 1
#define OFF 0

/******************** /* specific defines ECG board ************************/
/* RED LED */
#define RED_ON() T2EX=0 //vermelho ( 0=aceso )
#define RED_OFF() T2EX=1 //vermelho ( 1=apagado )
#define RED_TOGGLE() T2EX=!T2EX
#define RED_LED T2EX

#define SET_TMR0_MODE1()   TMOD = (TMOD & 0xF0)|0x01;
#define TMR0_START()       TR0 = 1; //TCON = (TCON & 0xEF)|0x10;

/* Power Supply Monitor Enable */
#define ENABLE_PSM 1;
#define DISABLE_PSM 0;
#define PSMI 0x20;
#define ENABLE_PSM_IRQ() PSMCON |= PSMI;

/* Power Down and idle mode */
#define PD 0x02;
#define ENABLE_POWER_DOWN() PCON = PCON | PD;
#define IDL 0x01;
#define ENABLE_IDLE_MODE() PCON = PCON | IDL;

#define ENABLE_SERIAL_IRQ() ES = 1;
#define DISABLE_SERIAL_IRQ() ES = 0;
#define ENABLE_INT1_IRQ() EX1 = 1;
#define DISABLE_INT1_IRQ() EX1 = 0;

// Timer 0 trigger value;
// tick time = 12 (machine cycles)*256(Number of TH0 pulses)*DIV(TH0 overflow times) / CPU_CLK
// 50ms tick time with 12.58 MHz clock requires DIV = 205
#define TMR0_DIV  205

// 50 ms ticks with 12.58 MHz clock requires 52417 timer pulses
```

```
// that means a timer0 pre-charge of 65536-52417 = 13119 = 333F
#define TIMER0_SET 0x333F

#define XRAM_ENABLE() CFG834 |= 0x01

/* RTC Configuration */
#define RTC_ENABLE() TIMECON |= 0x01;
#define SET_RTC_TMR(HOUR_VALUE, MIN_VALUE, SEC_VALUE) TIMECON &= 0x00; \
  HOUR=(HOUR_VALUE); \
  MIN=(MIN_VALUE); \
  SEC = (SEC_VALUE); \

#define CONFIG_INTERVAL_COUNTER_HOURS() TIMECON = TIMECON & 0x0F; \
TIMECON = TIMECON | 0x30; \

#define CONFIG_INTERVAL_COUNTER_MINUTES() TIMECON = TIMECON & 0x0F; \
  TIMECON = TIMECON | 0x20; \

#define CONFIG_INTERVAL_COUNTER_SECONDS() TIMECON = TIMECON & 0x0F; \
  TIMECON = TIMECON | 0x10; \

#define ENABLE_TIMER_INTERVAL_COUNTER() TIMECON = TIMECON & 0xFD; \
TIMECON = TIMECON | 0x02; \


#define ENABLE_TIC_IRQ() IEIP2 = IEIP2 | 0x04;

/* WDT */
/* reset WDE bit to clear WDT*/
#define RESET_WDE() DISABLE_GLOBAL_IRQ(); \
ENABLE_WDT_WRITE(); \
WDE = 0; \
ENABLE_GLOBAL_IRQ(); \

/*enable WDT for 2seg timeout*/
#define CONFIG_WDT_2SEG() DISABLE_GLOBAL_IRQ(); \
ENABLE_WDT_WRITE(); \
WDCON = 0x72; \
ENABLE_GLOBAL_IRQ(); \

/* enable INT0 in power down mode */
#define ENABLE_INT0_PD() PCON = PCON | 0x20;

/* disable ALE output */
#define DISABLE_ALE_OUT() PCON = PCON | 0x10;

/* enable Oscillator in Power Down Mode */
#define ENABLE_OSC_PD() PLLCON = PLLCON & 0x7f;

/* tmp_buffer definition */
```

```
#define tmp_buff_size 64
extern unsigned int xdata tmp_buff[tmp_buff_size +1];

/*================================================================*/
/* 802.11 Constants */

/*--- Sizes/offsets ----------------------------------------------*/
#define WLAN_FCS_LEN 4
#define WLAN_ADDR_LEN 6
#define WLAN_CRC_LEN 4
#define WLAN_HDR_A3_LEN 24
#define WLAN_HDR_A4_LEN 30
#define WLAN_SSID_MAXLEN 32
#define WLAN_DATA_MAXLEN 2312

#define WIFI_FCS_LEN 2
#define WIFI_ADDR_LEN 3
#define WIFI_HDR_A3_LEN 12
#define WIFI_HDR_A4_LEN 15

#define FRAME_STRUCTURE_OFF 46
#define ETH_HDR_OFF  14
#define MAC_LEN      6
#define SNAP_LEN     3
#define ETH_MTU_BUFF_SIZE 1500
#define TXFRAME_BUFF_SIZE ETH_MTU_BUFF_SIZE + FRAME_STRUCTURE_OFF + ETH_HDR_OFF

#define TX_BSSID_OFF  18
#define TX_DEST_ADDR_OFF  46
#define TX_SRC_ADDR_OFF  52
#define RX_DST_ADDR_OFF   46
#define RX_SRC_ADDR_OFF  52
#define DATALEN_OFF      58
#define RXDATALEN_OFF    44
#define RXDATA_OFF   60
#define TXDATA_OFF   60

#define BAP_WRITE          1
#define BAP_READ           0

/* API */
#define  BSS   0x01
#define  IBSS   0x02
#define  SSID   0x03
#define  MAX_LEN      0x04
#define  ENABLE   0x05
#define  PM_ENABLE    0x06
#define  PM_MAX_SLEEP 0x07
/*================================================================*/
//* Ethernet Header Layout
```

```
//*=============================================================*/
#define ETH_PCKT_LEN         0x00  //enetpacketLen11
#define ETH_PCKT_DST    0x00  //0x01  //enetpacketDest
#define ETH_PCKT_DST1       0x00  //0x01  //enetpacketDest01 destination mac address
#define ETH_PCKT_DST2        0x01  //0x02  //enetpacketDest23
#define ETH_PCKT_DST3          0x02  //0x03  //enetpacketDest45
#define ETH_PCKT_SRC           0x06  //0x04  //enetpacketSrc
#define ETH_PCKT_SRC1    0x03  //0x04  //enetpacketSrc01 source mac address
#define ETH_PCKT_SRC2    0x04  //0x05  //enetpacketSrc23
#define ETH_PCKT_SRC3        0x05  //0x06  //enetpacketSrc45
#define ETH_PCKT_LEN03    0x0C  //0x07  //enetpacketLen03
#define ETH_PCKT_SNAP    0x0D  //0x0E  //0x08  //enetpacketSnap
#define ETH_PCKT_CTRL    0x0E  //0x10  //0x09  //enetpacketCntrl
#define ETH_PCKT_OUI    0x0F  //0x12  //0x0A
#define ETH_PCKT_TYPE    0x10  //0x14  //0x0B  //enetpacketType type/length field
#define ETH_PCKT_DATA          0x11  //0x16  //0x0C  //enetpacketData  IP data area begins here

/*--- Command Code Constants --------------------------*/
/*--- Controller Commands --------------------------*/
#define HFA384x_CMDCODE_INIT 0x0000 //((int)0x00)
#define HFA384x_CMDCODE_ENABLE 0x0001 //((int)0x01)
#define HFA384x_CMDCODE_DISABLE 0x0002 //((int)0x02)
#define HFA384x_CMDCODE_DIAG 0x0003 //((int)0x03)

/*--- Buffer Mgmt Commands --------------------------*/
#define HFA384x_CMDCODE_ALLOC 0x000A //((int)0x0A)
#define HFA384x_CMDCODE_TX     0x000B // ((int)0x0B)

/*--- Regulate Commands --------------------------*/
#define HFA384x_CMDCODE_NOTIFY 0x0010
#define HFA384x_CMDCODE_INQUIRE 0x0011

/*--- Configure Commands --------------------------*/
#define HFA384x_CMDCODE_ACCESS 0x0021 //((int)0x21)

/*--- Result Codes Status Register ---------------------*/
#define HFA384x_RES_SUCCESS 0x00
#define HFA384x_RES_CARDFAIL 0x01
#define HFA384x_RES_NOBUFF      0x05
#define     HFA384x_RES_COMMERR      0x7F

/*--- Register Test/Get/Set Field macros ------------------------*/

#define HFA384x_CMD_ISBUSY(value) ((int)(((int)value) & HFA384x_CMD_BUSY))
#define HFA384x_CMD_AINFO_GET(value) ((int)(((int)(value) & HFA384x_CMD_AINFO) >> 8))
#define HFA384x_CMD_AINFO_SET(value) ((int)((int)(value) << 8))
#define HFA384x_CMD_MACPORT_GET(value) ((int)(HFA384x_CMD_AINFO_GET((int)(value) & HFA384x_CMD_MACPORT)))
#define HFA384x_CMD_MACPORT_SET(value) ((int)HFA384x_CMD_AINFO_SET(value))
#define HFA384x_CMD_RECL_SET(value) ((int)HFA384x_CMD_AINFO_SET(value))
#define HFA384x_CMD_QOS_SET(value) ((int)((((int)(value)) << 12) & 0x3000)) // nao tem ref.
```

```
#define HFA384x_STATUS_RESULT_GET(value) ((int)((((int)(value)) & HFA384x_STATUS_RESULT) >> 8))
#define HFA384x_EVSTAT_ISCMD(value) ((int)(((int)(value)) & HFA384x_EVSTAT_CMD))
#define HFA384x_CMD_CMDCODE_SET(value) ((int)(value))
#define HFA384x_EVSTAT_ISALLOC(value) ((int)(((int)(value)) & HFA384x_EVSTAT_ALLOC))
#define HFA384x_CMD_WRITE_SET(value) ((int)HFA384x_CMD_AINFO_SET((int)value))
#define HFA384x_EVSTAT_ISALLOC(value) ((int)(((int)(value)) & HFA384x_EVSTAT_ALLOC))

#define HFA384x_OFFSET_ISBUSY(value) ((int)(((int)(value)) & HFA384x_OFFSET_BUSY))
#define HFA384x_OFFSET_ISERR(value) ((int)(((int)(value)) & HFA384x_OFFSET_ERR))

               /*Interrupt enable masks */
#define HFA384x_INTEN_ISTICK(value) ((int)(((int)(value)) & HFA384x_INTEN_TICK))
#define HFA384x_INTEN_TICK_SET(value) ((int)(((int)(value)) << 15))
#define HFA384x_INTEN_ISWTERR(value) ((int)(((int)(value)) & HFA384x_INTEN_WTERR))
#define HFA384x_INTEN_WTERR_SET(value) ((int)(((int)(value)) << 14))
#define HFA384x_INTEN_ISINFDROP(value) ((int)(((int)(value)) & HFA384x_INTEN_INFDROP))
#define HFA384x_INTEN_INFDROP_SET(value) ((int)(((int)(value)) << 13))
#define HFA384x_INTEN_ISINFO(value) ((int)(((int)(value)) & HFA384x_INTEN_INFO))
#define HFA384x_INTEN_INFO_SET(value) ((int)(((int)(value)) << 7))
#define HFA384x_INTEN_ISDTIM(value) ((int)(((int)(value)) & HFA384x_INTEN_DTIM))
#define HFA384x_INTEN_DTIM_SET(value) ((int)(((int)(value)) << 5))
#define HFA384x_INTEN_ISCMD(value) ((int)(((int)(value)) & HFA384x_INTEN_CMD))
#define HFA384x_INTEN_CMD_SET(value) ((int)(((int)(value)) << 4))
#define HFA384x_INTEN_ISALLOC(value) ((int)(((int)(value)) & HFA384x_INTEN_ALLOC))
#define HFA384x_INTEN_ALLOC_SET(value) ((int)(((int)(value)) << 3))
#define HFA384x_INTEN_ISTXEXC(value) ((int)(((int)(value)) & HFA384x_INTEN_TXEXC))
#define HFA384x_INTEN_TXEXC_SET(value) ((int)(((int)(value)) << 2))
#define HFA384x_INTEN_ISTX(value) ((int)(((int)(value)) & HFA384x_INTEN_TX))
#define HFA384x_INTEN_TX_SET(value) ((int)(((int)(value)) << 1))
#define HFA384x_INTEN_ISRX(value) ((int)(((int)(value)) & HFA384x_INTEN_RX))
#define HFA384x_INTEN_RX_SET(value) ((int)(((int)(value)) << 0))

#define HFA384x_EVSTAT_ISTICK(value) ((int)(((int)(value)) & HFA384x_EVSTAT_TICK))
#define HFA384x_EVSTAT_ISWTERR(value) ((int)(((int)(value)) & HFA384x_EVSTAT_WTERR))
#define HFA384x_EVSTAT_ISINFDROP(value) ((int)(((int)(value)) & HFA384x_EVSTAT_INFDROP))
#define HFA384x_EVSTAT_ISINFO(value) ((int)(((int)(value)) & HFA384x_EVSTAT_INFO))
#define HFA384x_EVSTAT_ISDTIM(value) ((int)(((int)(value)) & HFA384x_EVSTAT_DTIM))
#define HFA384x_EVSTAT_ISCMD(value) ((int)(((int)(value)) & HFA384x_EVSTAT_CMD))
#define HFA384x_EVSTAT_ISALLOC(value) ((int)(((int)(value)) & HFA384x_EVSTAT_ALLOC))
#define HFA384x_EVSTAT_ISTXEXC(value) ((int)(((int)(value)) & HFA384x_EVSTAT_TXEXC))
#define HFA384x_EVSTAT_ISTX(value) ((int)(((int)(value)) & HFA384x_EVSTAT_TX))
#define HFA384x_EVSTAT_ISRX(value)  ((int)(((int)(value)) & HFA384x_EVSTAT_RX))

#define HFA384x_EVACK_ISTICK(value) ((int)(((int)(value)) & HFA384x_EVACK_TICK))
#define HFA384x_EVACK_TICK_SET(value) ((int)(((int)(value)) << 15))
#define HFA384x_EVACK_ISWTERR(value) ((int)(((int)(value)) & HFA384x_EVACK_WTERR))
#define HFA384x_EVACK_WTERR_SET(value) ((int)(((int)(value)) << 14))
#define HFA384x_EVACK_ISINFDROP(value) ((int)(((int)(value)) & HFA384x_EVACK_INFDROP))
#define HFA384x_EVACK_INFDROP_SET(value) ((int)(((int)(value)) << 13))
#define HFA384x_EVACK_ISINFO(value) ((int)(((int)(value)) & HFA384x_EVACK_INFO))
```

```
#define HFA384x_EVACK_INFO_SET(value) ((int)(((int)(value)) << 7))
#define HFA384x_EVACK_ISDTIM(value) ((int)(((int)(value)) & HFA384x_EVACK_DTIM))
#define HFA384x_EVACK_DTIM_SET(value) ((int)(((int)(value)) << 5))
#define HFA384x_EVACK_ISCMD(value) ((int)(((int)(value)) & HFA384x_EVACK_CMD))
#define HFA384x_EVACK_CMD_SET(value) ((int)(((int)(value)) << 4))
#define HFA384x_EVACK_ISALLOC(value) ((int)(((int)(value)) & HFA384x_EVACK_ALLOC))
#define HFA384x_EVACK_ALLOC_SET(value) ((int)(((int)(value)) << 3))
#define HFA384x_EVACK_ISTXEXC(value) ((int)(((int)(value)) & HFA384x_EVACK_TXEXC))
#define HFA384x_EVACK_TXEXC_SET(value) ((int)(((int)(value)) << 2))
#define HFA384x_EVACK_ISTX(value) ((int)(((int)(value)) & HFA384x_EVACK_TX))
#define HFA384x_EVACK_TX_SET(value) ((int)(((int)(value)) << 1))
#define HFA384x_EVACK_ISRX(value) ((int)(((int)(value)) & HFA384x_EVACK_RX))
#define HFA384x_EVACK_RX_SET(value) ((int)(((int)(value)) << 0))

/*--- Register Field Masks ------------------------*/
#define HFA384x_CMD_BUSY 0x8000 //BIT15=1
#define HFA384x_STATUS_RESULT 0x7F00 //BIT14 | BIT13 | BIT12 | BIT11 | BIT10 | BIT9 | BIT8 = 1
#define HFA384x_EVSTAT_CMD 0x0010 //BIT4=1

#define HFA384x_OFFSET_BUSY 0x8000 //((int)BIT15)
#define HFA384x_OFFSET_ERR 0x4000 //((int)BIT14)

#define     HFA384x_INT_NORMAL           (HFA384x_EVSTAT_INFO|HFA384x_EVSTAT_RX|HFA384x_EVSTAT_TX|HFA384x_EVSTAT_TXEXC|HFA384x_E

#define HFA384x_INTEN_TICK 0x8000 //((int)BIT15)
#define HFA384x_INTEN_WTERR 0x4000 //((int)BIT14)
#define HFA384x_INTEN_INFDROP 0x2000 //((int)BIT13)
#define HFA384x_INTEN_INFO 0x0080 //((int)BIT7)
#define HFA384x_INTEN_DTIM 0x0020 //((int)BIT5)
#define HFA384x_INTEN_CMD 0x0010 //((int)BIT4)
#define HFA384x_INTEN_ALLOC 0x0008 //((int)BIT3)
#define HFA384x_INTEN_TXEXC 0x0004 //((int)BIT2)
#define HFA384x_INTEN_TX 0x0002 //((int)BIT1)
#define HFA384x_INTEN_RX 0x0001 //((int)BIT0)

#define HFA384x_EVACK_TICK 0x8000 //((int)BIT15)
#define HFA384x_EVACK_WTERR 0x4000 //((int)BIT14)
#define HFA384x_EVACK_INFDROP 0x2000 //((int)BIT13)
#define HFA384x_EVACK_INFO 0x0080 //((int)BIT7)
#define HFA384x_EVACK_DTIM 0x0020 //((int)BIT5)
#define HFA384x_EVACK_CMD 0x0010 //((int)BIT4)
#define HFA384x_EVACK_ALLOC 0x0008 //((int)BIT3)
#define HFA384x_EVACK_TXEXC 0x0004 //((int)BIT2)
#define HFA384x_EVACK_TX 0x0002 //((int)BIT1)
#define HFA384x_EVACK_RX 0x0001 //((int)BIT0)

#define HFA384x_EVSTAT_TICK 0x8000 //((int)BIT15)
#define HFA384x_EVSTAT_WTERR 0x4000 //((int)BIT14)
#define HFA384x_EVSTAT_INFDROP 0x2000 //((int)BIT13)
#define HFA384x_EVSTAT_INFO 0x0080 //((int)BIT7)
```

```
#define HFA384x_EVSTAT_DTIM 0x0020 //((int)BIT5)
#define HFA384x_EVSTAT_CMD 0x0010 //((int)BIT4)
#define HFA384x_EVSTAT_ALLOC 0x0008 //((int)BIT3)
#define HFA384x_EVSTAT_TXEXC 0x0004 //((int)BIT2)
#define HFA384x_EVSTAT_TX 0x0002 //((int)BIT1)
#define HFA384x_EVSTAT_RX 0x0001 //((int)BIT0)

/*--- Register ID macros ------------------------*/

#define HFA384x_CMD HFA384x_CMD_OFF
#define HFA384x_PARAM0 HFA384x_PARAM0_OFF
#define HFA384x_PARAM1 HFA384x_PARAM1_OFF
#define HFA384x_PARAM2 HFA384x_PARAM2_OFF
#define HFA384x_STATUS HFA384x_STATUS_OFF
#define HFA384x_RESP0 HFA384x_RESP0_OFF
#define HFA384x_RESP1 HFA384x_RESP1_OFF
#define HFA384x_RESP2 HFA384x_RESP2_OFF
#define HFA384x_INFOFID HFA384x_INFOFID_OFF
#define HFA384x_RXFID HFA384x_RXFID_OFF
#define HFA384x_ALLOCFID HFA384x_ALLOCFID_OFF
#define HFA384x_TXCOMPLFID HFA384x_TXCOMPLFID_OFF
#define HFA384x_SELECT0 HFA384x_SELECT0_OFF
#define HFA384x_OFFSET0 HFA384x_OFFSET0_OFF
#define HFA384x_DATA0 HFA384x_DATA0_OFF
#define HFA384x_SELECT1 HFA384x_SELECT1_OFF
#define HFA384x_OFFSET1 HFA384x_OFFSET1_OFF
#define HFA384x_DATA1 HFA384x_DATA1_OFF
#define HFA384x_EVSTAT HFA384x_EVSTAT_OFF
#define HFA384x_INTEN HFA384x_INTEN_OFF       /* Interrupt enable offset */
#define HFA384x_EVACK HFA384x_EVACK_OFF
#define HFA384x_CONTROL HFA384x_CONTROL_OFF
#define HFA384x_SWSUPPORT0 HFA384x_SWSUPPORT0_OFF
#define HFA384x_SWSUPPORT1 HFA384x_SWSUPPORT1_OFF
#define HFA384x_SWSUPPORT2 HFA384x_SWSUPPORT2_OFF
#define HFA384x_AUXPAGE HFA384x_AUXPAGE_OFF
#define HFA384x_AUXOFFSET HFA384x_AUXOFFSET_OFF
#define HFA384x_AUXDATA HFA384x_AUXDATA_OFF
#define HFA384x_PCICOR HFA384x_PCICOR_OFF
#define HFA384x_PCIHCR HFA384x_PCIHCR_OFF

/*--- Register I/O offsets ----------------------------*/
#define HFA384x_CMD_OFF (0x0000)
#define HFA384x_PARAM0_OFF (0x0002)
#define HFA384x_PARAM1_OFF (0x0004)
#define HFA384x_PARAM2_OFF (0x0006)
#define HFA384x_STATUS_OFF (0x0008)
#define HFA384x_RESP0_OFF (0x000A)
#define HFA384x_RESP1_OFF (0x000C)
#define HFA384x_RESP2_OFF (0x000E)
#define HFA384x_INFOFID_OFF (0x0010)
```

```
#define HFA384x_RXFID_OFF (0x0020)
#define HFA384x_ALLOCFID_OFF (0x0022)
#define HFA384x_TXCOMPLFID_OFF (0x0024)
#define HFA384x_SELECT0_OFF (0x0018)
#define HFA384x_OFFSET0_OFF (0x001C)
#define HFA384x_DATA0_OFF (0x0036)
#define HFA384x_SELECT1_OFF (0x001A)
#define HFA384x_OFFSET1_OFF (0x001E)
#define HFA384x_DATA1_OFF (0x0038)
#define HFA384x_EVSTAT_OFF (0x0030)
#define HFA384x_INTEN_OFF (0x0032)
#define HFA384x_EVACK_OFF (0x0034)
#define HFA384x_CONTROL_OFF (0x0014)
#define HFA384x_SWSUPPORT0_OFF (0x0028)
#define HFA384x_SWSUPPORT1_OFF (0x002A)
#define HFA384x_SWSUPPORT2_OFF (0x002C)
#define HFA384x_AUXPAGE_OFF (0x003A)
#define HFA384x_AUXOFFSET_OFF (0x003C)
#define HFA384x_AUXDATA_OFF (0x003E)

/*--- Support Constants ---------------------------*/
#define HFA384x_BAP_0     0x0000
#define HFA384x_BAP_1 0x0001
#define HFA384x_PORTTYPE_IBSS 0x0000
#define HFA384x_PORTTYPE_BSS 0x0001 //((int)1)
#define HFA384x_PORTTYPE_WDS 0x0002 //((int)2)
#define HFA384x_PORTTYPE_PSUEDOIBSS 0x0003 //((int)3)
#define HFA384x_PORTTYPE_HOSTAP     0x0006 //((int)6)

/*--- Just some symbolic names for legibility -------*/
#define HFA384x_TXCMD_NORECL 0x0000//((int)0)
#define HFA384x_TXCMD_RECL     0x0001//((int)1)

/*------ Constants -------------------------------------------*/
/*--- Mins & Maxs ---------------------------------*/
#define HFA384x_BAP_OFFSET_MAX 0x1000 //((int)4096)
#define HFA384x_BAP_DATALEN_MAX 0x1000 //((int)4096)
#define HFA384x_CMD_ALLOC_LEN_MIN 0x0004 //((int)4)
#define HFA384x_CMD_ALLOC_LEN_MAX 0x0960 //((int)2400)

#define HFA384x_DRVR_TXBUF_MAX (sizeof(hfa384x_tx_frame_t) + \
WLAN_DATA_MAXLEN - \
WLAN_WEP_IV_LEN - \
WLAN_WEP_ICV_LEN + 2)
#define HFA384x_DRVR_MAGIC (0x4a2d)
#define HFA384x_INFODATA_MAXLEN (sizeof(hfa384x_infodata_t))
#define HFA384x_INFOFRM_MAXLEN (sizeof(hfa384x_InfFrame_t))

/*--- Record ID Constants -------------------------*/
/*-------------------------------------------------------------------
```

```
Configuration RIDs: Network Parameters, Static Configuration Entities
----------------------------------------------------------------------*/
#define HFA384x_RID_CNFPORTTYPE 0xFC00
#define HFA384x_RID_CNFOWNMACADDR 0xFC01
#define HFA384x_RID_CNFDESIREDSSID 0xFC02
#define HFA384x_RID_CNFOWNCHANNEL 0xFC03
#define HFA384x_RID_CNFOWNSSID 0xFC04
#define HFA384x_RID_CNFOWNATIMWIN 0xFC05
#define HFA384x_RID_CNFSYSSCALE 0xFC06
#define HFA384x_RID_CNFMAXDATALEN 0xFC07
#define HFA384x_RID_CNFWDSADDR 0xFC08
#define HFA384x_RID_CNFPMENABLED 0xFC09

#define HFA384x_RID_CNFMAXSLEEPDUR 0xFC0C

/*----------------------------------------------------------------------
Information RIDs:  MAC Information
----------------------------------------------------------------------*/
#define HFA384x_RID_PORTSTATUS 0xFD40
#define HFA384x_RID_CURRENTSSID 0xFD41
#define HFA384x_RID_CURRENTBSSID 0xFD42
#define HFA384x_RID_COMMSQUALITY 0xFD43
#define HFA384x_RID_CURRENTTXRATE 0xFD44

/*----------------------------------------------------------------------
Configuration RIDs: Network Parameters, Dynamic Configuration Entities
----------------------------------------------------------------------*/
#define HFA384x_RID_TXRATECNTL 0xFC84

/*----------------------------------------------------------------------
Information RIDs:  Modem Information
----------------------------------------------------------------------*/
#define HFA384x_RID_CURRENTPOWERSTATE 0xFDC2

/*----------------------------------------------------------------------
Configuration RID lengths: Network Params, Static Config Entities
  This is the length of JUST the DATA part of the RID (does not
  include the len or code fields)
----------------------------------------------------------------------*/
/* TODO: fill in the rest of these */
#define HFA384x_RID_CNFPORTTYPE_LEN     0x0002//((int)2)
#define HFA384x_RID_CNFOWNMACADDR_LEN 0x0006//((int)6)
#define HFA384x_RID_CNFDESIREDSSID_LEN 0x0012//((int)34) --> 18 words = 0x12
#define HFA384x_RID_CNFOWNCHANNEL_LEN 0x0002//((int)2)
#define HFA384x_RID_CNFMAXDATALEN_LEN 0x0002
#define     HFA384x_RID_TXRATE_LEN          0x0002
#define HFA384x_RID_CNFPMENABLED_LEN 0x0002
#define HFA384x_RID_CNFMAXSLEEPDUR_LEN 0x0002

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

```
Communication Frames: Field Masks for Transmit Frames
----------------------------------------------------------------------*/
/*-- Status Field --*/
#define HFA384x_TXSTATUS_ACKERR 0x0020//((int)BIT5)
#define HFA384x_TXSTATUS_FORMERR 0x0008//((int)BIT3)
#define HFA384x_TXSTATUS_DISCON 0x0004//((int)BIT2)
#define HFA384x_TXSTATUS_AGEDERR 0x0002//((int)BIT1)
#define HFA384x_TXSTATUS_RETRYERR 0x0001(//(int)BIT0)
/*-- Transmit Control Field --*/
#define HFA384x_TX_CFPOLL     0x1000//((int)BIT12)
#define HFA384x_TX_PRST     0x0800//((int)BIT11)
#define HFA384x_TX_MACPORT     0x0700//((int)(BIT10 | BIT9 | BIT8))
#define HFA384x_TX_NOENCRYPT 0x0080//((int)BIT7)
#define HFA384x_TX_RETRYSTRAT 0x0060//((int)(BIT6 | BIT5))
#define HFA384x_TX_STRUCTYPE 0x0018//((int)(BIT4 | BIT3))
#define HFA384x_TX_TXEX     0x0004//((int)BIT2)
#define HFA384x_TX_TXOK     0x0001//((int)BIT1)
/*----------------------------------------------------------------------
Communication Frames: Test/Get/Set Field Values for Transmit Frames
----------------------------------------------------------------------*/
/*-- Status Field --*/
#define HFA384x_TXSTATUS_ISERROR(v) \
(((int)(v))&\
(HFA384x_TXSTATUS_ACKERR|HFA384x_TXSTATUS_FORMERR|\
HFA384x_TXSTATUS_DISCON|HFA384x_TXSTATUS_AGEDERR|\
HFA384x_TXSTATUS_RETRYERR))

#define HFA384x_TXSTATUS_ISACKERR(v) ((int)(((int)(v)) & HFA384x_TXSTATUS_ACKERR))
#define HFA384x_TXSTATUS_ISFORMERR(v) ((int)(((int)(v)) & HFA384x_TXSTATUS_FORMERR))
#define HFA384x_TXSTATUS_ISDISCON(v) ((int)(((int)(v)) & HFA384x_TXSTATUS_DISCON))
#define HFA384x_TXSTATUS_ISAGEDERR(v) ((int)(((int)(v)) & HFA384x_TXSTATUS_AGEDERR))
#define HFA384x_TXSTATUS_ISRETRYERR(v) ((int)(((int)(v)) & HFA384x_TXSTATUS_RETRYERR))

#define HFA384x_TX_GET(v,m,s) ((((int)(v))&((int)(m)))>>((int)(s)))
#define HFA384x_TX_SET(v,m,s) ((((int)(v))<<((int)(s)))&((int)(m)))

#define HFA384x_TX_CFPOLL_GET(v) HFA384x_TX_GET(v, HFA384x_TX_CFPOLL,12)
#define HFA384x_TX_CFPOLL_SET(v) HFA384x_TX_SET(v, HFA384x_TX_CFPOLL,12)
#define HFA384x_TX_PRST_GET(v) HFA384x_TX_GET(v, HFA384x_TX_PRST,11)
#define HFA384x_TX_PRST_SET(v) HFA384x_TX_SET(v, HFA384x_TX_PRST,11)
#define HFA384x_TX_MACPORT_GET(v) HFA384x_TX_GET(v, HFA384x_TX_MACPORT, 8)
#define HFA384x_TX_MACPORT_SET(v) HFA384x_TX_SET(v, HFA384x_TX_MACPORT, 8)
#define HFA384x_TX_NOENCRYPT_GET(v) HFA384x_TX_GET(v, HFA384x_TX_NOENCRYPT, 7)
#define HFA384x_TX_NOENCRYPT_SET(v) HFA384x_TX_SET(v, HFA384x_TX_NOENCRYPT, 7)
#define HFA384x_TX_RETRYSTRAT_GET(v) HFA384x_TX_GET(v, HFA384x_TX_RETRYSTRAT, 5)
#define HFA384x_TX_RETRYSTRAT_SET(v) HFA384x_TX_SET(v, HFA384x_TX_RETRYSTRAT, 5)
#define HFA384x_TX_STRUCTYPE_GET(v) HFA384x_TX_GET(v, HFA384x_TX_STRUCTYPE, 3)
#define HFA384x_TX_STRUCTYPE_SET(v) HFA384x_TX_SET(v, HFA384x_TX_STRUCTYPE, 3)
#define HFA384x_TX_TXEX_GET(v) HFA384x_TX_GET(v, HFA384x_TX_TXEX, 2)
#define HFA384x_TX_TXEX_SET(v) HFA384x_TX_SET(v, HFA384x_TX_TXEX, 2)
```

```
#define HFA384x_TX_TXOK_GET(v) HFA384x_TX_GET(v, HFA384x_TX_TXOK, 1)
#define HFA384x_TX_TXOK_SET(v) HFA384x_TX_SET(v, HFA384x_TX_TXOK, 1)

/*--------------------------------------------------------------------
Communication Frames: Field Masks for Receive Frames
--------------------------------------------------------------------*/
/*-- Offsets --------*/
#define HFA384x_RX_DATA_LEN_OFF 0x002c//((int)44)
#define HFA384x_RX_80211HDR_OFF 0x000E//((int)14)
#define HFA384x_RX_DATA_OFF 0x003c//((int)60)

/*-- Status Fields --*/
#define HFA384x_RXSTATUS_MSGTYPE 0xE000//((int)(BIT15 | BIT14 | BIT13))
#define HFA384x_RXSTATUS_MACPORT 0x0700//((int)(BIT10 | BIT9 | BIT8))
#define HFA384x_RXSTATUS_UNDECR 0x0002//((int)BIT1)
#define HFA384x_RXSTATUS_FCSERR 0x0001//((int)BIT0)
/*--------------------------------------------------------------------
Communication Frames: Test/Get/Set Field Values for Receive Frames
--------------------------------------------------------------------*/
#define HFA384x_RXSTATUS_MSGTYPE_GET(value) ((int)((((int)(value)) & HFA384x_RXSTATUS_MSGTYPE) >> 13))
#define HFA384x_RXSTATUS_MSGTYPE_SET(value) ((int)(((int)(value)) << 13))
#define HFA384x_RXSTATUS_MACPORT_GET(value) ((int)((((int)(value)) & HFA384x_RXSTATUS_MACPORT) >> 8))
#define HFA384x_RXSTATUS_MACPORT_SET(value) ((int)(((int)(value)) << 8))
#define HFA384x_RXSTATUS_ISUNDECR(value) ((int)(((int)(value)) & HFA384x_RXSTATUS_UNDECR))
#define HFA384x_RXSTATUS_ISFCSERR(value) ((int)(((int)(value)) & HFA384x_RXSTATUS_FCSERR))

/*--------------------------------------------------------------------
 FRAME STRUCTURES: Information Types
--------------------------------------------------------------------
Information Types
--------------------------------------------------------------------*/
#define HFA384x_IT_HANDOVERADDR 0xF000
#define HFA384x_IT_HANDOVERDEAUTHADDRESS 0xF001
#define HFA384x_IT_COMMTALLIES 0xF100
#define HFA384x_IT_SCANRESULTS 0xF101
#define HFA384x_IT_CHINFORESULTS 0xF102
#define HFA384x_IT_HOSTSCANRESULTS 0xF103
#define HFA384x_IT_LINKSTATUS 0xF200


/* external variables */
extern char xdata cis_device;
extern char xdata cor_data;
extern unsigned int xdata cor_addr;

/* Function prototypes wifidev*/
extern void CFWriteIOByte(char val, unsigned int reg);
extern char CFReadIOByte(unsigned int reg);
extern void CFSetAddr(unsigned int CFaddr);
extern void CFWriteMemByte(char val, unsigned int reg);
```

```
extern char CFReadMemByte(unsigned int reg);

extern void delay_nops(int value);
extern bit CFConnected(void);

void hfa384x_setreg(int val, unsigned int reg);
int hfa384x_getreg(unsigned int reg);
int hfa384x_getdata(unsigned int datareg);
void hfa384x_setdata(int val, unsigned int datareg);

//void hfa384x_copy_to_bap(bit bapbuf, char* buff_data, unsigned int count);
char hfa384x_cmd(unsigned int cmd, unsigned int parm0);
char hfa384x_cmd_allocate(unsigned int len);
char hfa384x_alloc_tx_buff(void);
char hfa384x_alloc_info_buff(void);
char hfa384x_cmd_diagnose(void);

char get_free_bap(void);
void hfa384x_cmd_access(bit write, unsigned int id, unsigned int* buf, int len);
char hfa384x_rdwr_bap(bit write, int id, unsigned int offset,
       char* buf, int len);
void ReadCISTable(void);
void hfa384x_drvr_start(void);
void uIPwifi_api(unsigned int cmd, unsigned int val);
void config_chip(void);
void wifidev_init(void);
unsigned int wifidev_read(void);
void hfa384x_event_ack(void);
void wifidev_send(void);
void find_CF_mode(void);


/*********************************************************************
* Filename: wifidev.c
* Description: This file implements the prism device driver for 8051
*              microcontroller
* Developed by: Marco Carvalho
* Created on: 30/03/05
*********************************************************************/

#include "wifidev.h"

/*control message print*/
#define PRINT_WIFI 0

//tx buffers
#define HFA384x_DRVR_FIDSTACKLEN_MAX 3

/***************************Global Variables ****************************/
```

```
int xdata txfid_queue[HFA384x_DRVR_FIDSTACKLEN_MAX]; // 3 tx buffers
int xdata infofid_queue[1];
char xdata infofid;
char xdata txfid;
unsigned int xdata cor_addr, cmd_data;
char xdata cis_device = 0;
char xdata cor_data;

#if PRINT_WIFI
char xdata bssidc[6];
#endif

char xdata txflag=0;

/* Buffers */
unsigned int xdata tmp_buff[tmp_buff_size +1];
char xdata comframe_buff[60];


/*---------------------------------------------------------------
* delay_nops
*
* Delay a value number os nops function
---------------------------------------------------------------*/
void delay_nops(int value)
{
    for (;value;value--)
        NOP();
}

/*---------------------------------------------------------------
* CFConnected
*
* Check if CF card is connected
---------------------------------------------------------------*/
bit CFConnected(void)
{
    bit connected;

    connected=CF_CD1; //leitura do pino que informa se CF esta conectada
    if(connected) return 0; //Erro se P1.4=1 !
    else return 1;
}

/*---------------------------------------------------------------
* CFSetAddr
*
* Set addr to CF
*   A8  |  A7  |  A6  |  A5  |  A4  |  A3  |  A2  |  A1  |  A0
*  P3.7 | P3.6 | P3.3 | P2.7 | P2.6 | P2.0 | P2.1 | P2.2 | P2.3
```

```
-------------------------------------------------------------*/
void CFSetAddr(unsigned int CFaddr)
{
    A0 = (CFaddr)&0x0001;
    A1 = (CFaddr >> 1)&0x0001;
    A2 = (CFaddr >> 2)&0x0001;
    A3 = (CFaddr >> 3)&0x0001;
    A4 = (CFaddr >> 4)&0x0001;
    A5 = (CFaddr >> 5)&0x0001;
    A6 = (CFaddr >> 6)&0x0001;
    A7 = (CFaddr >> 7)&0x0001;
    A8 = (CFaddr >> 8)&0x0001;
}

/*-------------------------------------------------------------
* CFWriteIOByte()
*
* Write some data (val) in a register (reg) in IO mode
-------------------------------------------------------------*/
static void CFWriteIOByte(char val, unsigned int reg)
{

    CFSetAddr(reg); // Set IO reg to write in CF

    CF_DATA_PORT=val; //write in PORT0

    CF_REG_OFF();
    CF_IOWR_OFF(); // IOwrite=0
    CF_IOWR_ON(); // IOwrite=1
    CF_REG_ON();

    // Set Data outputs as inputs
    CF_DATA_PORT = CF_DATA_PORT_MASK;

    return ;
}

/*-------------------------------------------------------------
* CFReadIOByte()
*
* Read some data From a register (reg) in IO mode
-------------------------------------------------------------*/
char CFReadIOByte(unsigned int reg)
{
    char xdata val;

    CFSetAddr(reg); // Set IO reg to read from CF

    CF_REG_OFF(); //chip select=0
    CF_IORD_OFF(); //IOread=0
```

```
    val = CF_DATA_PORT;  // val = P0 (8051)
    CF_IORD_ON(); //IOread=1
    CF_REG_ON();

    return val;
}

/*-----------------------------------------------------------------
* CFWriteMemByte
*
* write a byte in the CF attribute memory
-----------------------------------------------------------------*/
void CFWriteMemByte(char val, unsigned int reg)
{

    CFSetAddr(reg); // Set Mem reg to write in CF

    CF_DATA_PORT=val; //escrita no PORT0=buf do 8051

    CF_REG_OFF();
    CF_WR_OFF(); // write=0
    CF_WR_ON(); // write=1
    CF_REG_ON();

    CF_DATA_PORT = CF_DATA_PORT_MASK;

    return ;
}

/*-----------------------------------------------------------------
* CFReadMemByte
*
* Read a byte from the CF attribute memory
-----------------------------------------------------------------*/
char CFReadMemByte(unsigned int reg)
{
    char xdata val;

    CFSetAddr(reg); // Set Mem reg to read from CF

    CF_REG_OFF(); //chip select=0
    CF_RD_OFF(); //read=0
    val = CF_DATA_PORT;  // val = P0 (8051)
    CF_RD_ON(); //read=1
    CF_REG_ON();

    return val;
}

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

```
* hfa384x_setreg
*
* Set the value of one of the MAC registers.  Done here
* because different PRISM2 MAC parts use different buses and such.
----------------------------------------------------------------*/
void hfa384x_setreg(int val, unsigned int reg)
{
    /* Write the low byte firt */
    CFWriteIOByte((char)val&0xFF, reg);
    CFWriteIOByte((char)(val>>8), reg+1);
    return;
}

void hfa384x_setdata(int val, unsigned int datareg)
{
    /* Write the High byte first */
    CFWriteIOByte((char)(val>>8), datareg);
    CFWriteIOByte((char)val&0xFF, datareg+1);
    return;
}

/*----------------------------------------------------------------
* hfa384x_getreg
*
* Retrieve the value of one of the MAC registers.  Done here
* because different PRISM2 MAC parts use different buses and such.
----------------------------------------------------------------*/
int hfa384x_getreg(unsigned int reg)
{
    int xdata val=0;

    val = CFReadIOByte(reg)&0x00FF;
    val |= CFReadIOByte(reg+1)<<8;
    return val;

}

int hfa384x_getdata(unsigned int datareg)
{
    int xdata val=0;

    val = CFReadIOByte(datareg)<<8;
    val |= CFReadIOByte(datareg+1)&0x00FF;
    return val;

}

/*----------------------------------------------------------------
* hfa384x_cmd
*
```

```
* Issues the cmd command
*
* Arguments:
* cmd command to set
*   parm0   parameter 0
*
* Returns:
* result
----------------------------------------------------------------*/
char hfa384x_cmd(unsigned int cmd, unsigned int parm0)
{
    char xdata  result;
    unsigned int xdata cmd_status, evstat_data;
    unsigned int xdata timeout_count;

    /* Wait for busy bit */
    do{
        cmd_data = hfa384x_getreg(HFA384x_CMD);
        timeout_count++;
    }while((HFA384x_CMD_ISBUSY(cmd_data)) && (timeout_count<10));
    timeout_count = 0;

    /* Write command and param0 */
    hfa384x_setreg(parm0, HFA384x_PARAM0);
    hfa384x_setreg(cmd, HFA384x_CMD);

    /* Wait for busy bit */
    do{
        cmd_data = hfa384x_getreg(HFA384x_CMD);
    timeout_count++;
    }while((HFA384x_CMD_ISBUSY(cmd_data)) && (timeout_count<10));
    timeout_count = 0;

    /* Wait for command complete event */
    do{
        evstat_data = hfa384x_getreg(HFA384x_EVSTAT);
        timeout_count++;
    }while((!(HFA384x_EVSTAT_ISCMD(evstat_data))) && (timeout_count<5000));
    timeout_count = 0;

    /* Read Command result through status register */
    cmd_status = hfa384x_getreg(HFA384x_STATUS);
    result = HFA384x_STATUS_RESULT_GET(cmd_status);
    /* acknowledge the cmd event */
    hfa384x_setreg(HFA384x_EVACK_CMD, HFA384x_EVACK);

    return(result);
}

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

```
* hfa384x_cmd_allocate
*
* Issues the allocate command instructing the firmware to allocate
* a 'frame structure buffer' in MAC controller RAM.  This command
* does not provide the result, it only initiates one of the f/w's
* asynchronous processes to construct the buffer.  When the
* allocation is complete, it will be indicated via the Alloc
* bit in the EvStat register and the FID identifying the allocated
* space will be available from the AllocFID register.  Some care
* should be taken when waiting for the Alloc event.  If a Tx or
* Notify command w/ Reclaim has been previously executed, it's
* possible the first Alloc event after execution of this command
* will be for the reclaimed buffer and not the one you asked for.
* This case must be handled in the Alloc event handler.
*
* Arguments:
* len allocation length, must be an even value
* in the range [4-2400]. (host order)
*
* Returns:
* 0 success
* 1 f/w reported failure - f/w status code
*
----------------------------------------------------------------*/
char hfa384x_cmd_allocate(unsigned int len)
{
    char xdata result;

    result = hfa384x_cmd(HFA384x_CMDCODE_ALLOC, len);
    return result;
}

/*----------------------------------------------------------------
* hfa384x_alloc_tx_buff()
* allocate some tx FIDs in order to transmit data
----------------------------------------------------------------*/
char hfa384x_alloc_tx_buff(void)
{
    unsigned int xdata fid, evstat_data;
    char xdata i, result;

    /*Reset tx buffer positions and txfid (number of free fids)*/
    for(i=0;i<HFA384x_DRVR_FIDSTACKLEN_MAX;++i)
        txfid_queue[i] = 0;
    txfid = 0;

    // Create a common buffer structure between all TX buffers */
    /* Reset all comframe_buff positions */
    for(i=0;i<(FRAME_STRUCTURE_OFF + ETH_HDR_OFF);++i)
        comframe_buff[i] = 0x00;
```

```
    /* Setup TXControl field in the frame structure */
    comframe_buff[12] = 0x00;
    comframe_buff[13] = 0x06; //comframe_buff[13] =
                              //HFA384x_TX_TXOK_SET(1)|HFA384x_TX_TXEX_SET(1);

    /* SWSupport bytes */
    comframe_buff[6] = 0x55;
    comframe_buff[7] = 0xaa;
    comframe_buff[8] = 0x55;
    comframe_buff[9] = 0xaa;

    /* Alocate Tx buffers */
    for (i=0; i<HFA384x_DRVR_FIDSTACKLEN_MAX; i++)
    {
        result = hfa384x_cmd_allocate(TXFRAME_BUFF_SIZE);
        if (result)
            return 1;
        /* Wait for alloc bit in the event stat register to set */
        do{
            evstat_data = hfa384x_getreg(HFA384x_EVSTAT);
        }while(!(HFA384x_EVSTAT_ISALLOC(evstat_data)));
        /* get FID */
        fid = hfa384x_getreg(HFA384x_ALLOCFID);

        /* acknowledge the Alloc event in the event ack register */
        hfa384x_setreg(HFA384x_EVACK_ALLOC_SET(1), HFA384x_EVACK);
        /* Prepare BAP with 802.11 pre header*/
        result = hfa384x_rdwr_bap( BAP_WRITE, fid, 0x0000, comframe_buff, FRAME_STRUCTURE_OFF + ETH_HDR_OFF);
        if (result)
            return 1;
        /* Put FID in a TX FID Buffer */
        txfid_queue[i] = fid;
        txfid++;
    }
    return 0;
}

/*-------------------------------------------------------------
* hfa384x_alloc_info_buff()
* allocate some info FIDs in order to retrieve information data
-------------------------------------------------------------*/
char hfa384x_alloc_info_buff(void)
{
    unsigned int xdata fid, evstat_data;
    char xdata i, result;

    /*Reset tx buffer positions and txfid (number of free fids)*/
    for(i=0;i<1;++i)
        infofid_queue[i] = 0;
```

```
        infofid = 0;

    // Create a common buffer structure between all info buffers */
    /* Reset all comframe_buff positions */
    for(i=0;i<(FRAME_STRUCTURE_OFF + ETH_HDR_OFF);++i)
        comframe_buff[i] = 0x00;

    /* Alocate Info buffers */
    for (i=0; i<1; i++)
    {
        result = hfa384x_cmd_allocate(10);
        if (result)
            return 1;
        /* Wait for alloc bit in the event stat register to set */
        do{
            evstat_data = hfa384x_getreg(HFA384x_EVSTAT);
        }while(!(HFA384x_EVSTAT_ISALLOC(evstat_data)));
        /* get FID */
        fid = hfa384x_getreg(HFA384x_ALLOCFID);

        /* acknowledge the Alloc event in the event ack register */
        hfa384x_setreg(HFA384x_EVACK_ALLOC_SET(1), HFA384x_EVACK);

        if (result)
            return 1;
        /* Put FID in a TX FID Buffer */
        infofid_queue[i] = fid;
    }
    infofid++;
    return 0;
}

/*----------------------------------------------------------------
* hfa384x_cmd_diagnose
*
* Issues the diagnose command to test the: register interface,
* MAC controller (including loopback), External RAM, Non-volatile
* memory integrity, and synthesizers.  Following execution of this
* command, MAC/firmware are in the 'initial state'.  Therefore,
* the Initialize command should be issued after successful
* completion of this command.  This function may only be called
* when the MAC is in the 'communication disabled' state.
*
* Arguments:
*
* Returns:
* 0 success
* 1 f/w reported failure - f/w status code
*
----------------------------------------------------------------*/
```

```
#define DIAG_PATTERNA ((int)0xaaaa)
#define DIAG_PATTERNB ((int)~0xaaaa)

char hfa384x_cmd_diagnose(void)
{
    unsigned int xdata result_resp0 = 0;
    unsigned int xdata result_resp1 = 0;
    char xdata result = 1;
    unsigned int xdata timeout_count;

    unsigned int xdata cmd_status, evstat_data;

    /* Wait for busy bit */
    do{
        cmd_data = hfa384x_getreg(HFA384x_CMD); //0x0000
        timeout_count++;
    }while((HFA384x_CMD_ISBUSY(cmd_data))&&(timeout_count<10)); //0x8000
    timeout_count = 0;

    /* Write command, param0 and param1 */
    hfa384x_setreg(DIAG_PATTERNB, HFA384x_PARAM1); //0x0004
    hfa384x_setreg(DIAG_PATTERNA, HFA384x_PARAM0); //0x0002
    hfa384x_setreg(HFA384x_CMD_CMDCODE_SET(HFA384x_CMDCODE_DIAG), HFA384x_CMD); //0x0000

    /* Wait for busy bit */
    do{
        cmd_data = hfa384x_getreg(HFA384x_CMD); //0x0000
        timeout_count++;
    }while((HFA384x_CMD_ISBUSY(cmd_data))&&(timeout_count<10)); //0x8000
    timeout_count = 0;

    /* Wait for command complete event */
    do{
        evstat_data = hfa384x_getreg(HFA384x_EVSTAT); //0x0030
    }while(!(HFA384x_EVSTAT_ISCMD(evstat_data))); //0x0010

    /* Read Command result through status register */
    cmd_status = hfa384x_getreg(HFA384x_STATUS); //0x0008
    result = HFA384x_STATUS_RESULT_GET(cmd_status);

    /* acknowledge the cmd event */
    hfa384x_setreg(HFA384x_EVACK_CMD, HFA384x_EVACK); //0x0010, 0x0034

    result_resp0 = hfa384x_getreg(HFA384x_RESP0);
    result_resp1 = hfa384x_getreg(HFA384x_RESP1);

    if ((result_resp0 != DIAG_PATTERNB)||(result_resp1 != DIAG_PATTERNA))
        result = 1;

    return(result);
```

```
}

/*-----------------------------------------------------------------
* Get Free bap
-----------------------------------------------------------------*/
char get_free_bap(void)
{
    unsigned int xdata temp;
    char xdata result;

    result = 2;
    temp = hfa384x_getreg(HFA384x_OFFSET1);
    if(!(HFA384x_OFFSET_ISBUSY(temp)))
        result = 1;
    temp = hfa384x_getreg(HFA384x_OFFSET0);
    if(!(HFA384x_OFFSET_ISBUSY(temp)))
        result = 0;
    return(result);
}

/*-----------------------------------------------------------------
* hfa384x_cmd_access
*
* Requests that a given record be copied to/from the record
* buffer.  If we're writing from the record buffer, the contents
* must previously have been written to the record buffer via the
* bap.  If we're reading into the record buffer, the record can
* be read out of the record buffer after this call.
*
* Arguments:
* write [0|1] copy the record buffer to the given
* configuration record.
* rid_fid RID or fid of the record to read/write.
*           Write to bap = rid
*           Read from bap = fid
* buf host side record buffer.  Upon return it will
* contain the body portion of the record (minus the
* RID and len).
* len buffer length (in bytes, should match record length)
-----------------------------------------------------------------*/
void hfa384x_cmd_access(bit write, unsigned int id, unsigned int* buf, int len)
{
    char xdata result = 0;

    if (write)
    {
        result = hfa384x_rdwr_bap(BAP_WRITE, id, 0x0000, (char*) buf, (len+1)<<1);
        result = hfa384x_cmd(HFA384x_CMDCODE_ACCESS | HFA384x_CMD_WRITE_SET(write), id);
    }
    else
```

```
    {
        result = hfa384x_cmd(HFA384x_CMDCODE_ACCESS, id);
        result = hfa384x_rdwr_bap(BAP_READ, id, 0x0000, NULL, 0x0000);
    }

    return;
}
/*---------------------------------------------------------------
* hfa384x_rdwr_bap
*
* Copies a collection of bytes to the MAC controller memory via
* one set of BAP registers.
*
* Arguments:
* write [0|1] read/write to BAp control
* id FID or RID, destined for the select register (host order)
* offset An _even_ offset into the buffer for the given
* FID/RID.  We haven't the means to validate this,
* so be careful. (host order)
* buf ptr to array of bytes
* len length of data to transfer (in bytes)
*
* Returns:
* 0 success
* 1 f/w reported failure - value of offset reg.
---------------------------------------------------------------*/
char hfa384x_rdwr_bap(bit write, int id, unsigned int offset,
       char* buf, int len)
{
    char xdata result = 0;
    unsigned int xdata fid_ptr;
    unsigned int xdata reg;
    unsigned int xdata fidlen, i;
    char xdata bap;
    char xdata data_lo;
    int xdata *dataptr = (int*)buf;

    /* Disable all IRQs */
    DISABLE_GLOBAL_IRQ();

    bap = get_free_bap();

    if (write) // if write
    {
        if (bap == 1) // if bap = 1
        {
            /* Write id to select reg */
            hfa384x_setreg(id, HFA384x_SELECT1);
            /* Write offset to offset reg */
            hfa384x_setreg(offset, HFA384x_OFFSET1);
```

```
        /* Wait for offset[busy] to clear */
        do{
            reg = hfa384x_getreg(HFA384x_OFFSET1);
        }while(HFA384x_OFFSET_ISBUSY(reg));
        /* Check for error bit in offset reg if not error write in buffer path*/
        reg = hfa384x_getreg(HFA384x_OFFSET1);

        if(HFA384x_OFFSET_ISERR(reg))
            result = HFA384x_RES_CARDFAIL;
        else
        {
            if (txflag)
            {
                for(i=0;i<(len&0xFFFE);)
                {
                    hfa384x_setdata(*dataptr++, HFA384x_DATA1);
                    i+=2;
                }
                if(len % 2)
                {
                    data_lo = buf[i++];
                    hfa384x_setreg(data_lo, HFA384x_DATA1);
                }
            }
            else // if !txflag
            {
                for(i=0;i<(len&0xFFFE);)
                {
                    hfa384x_setreg(*dataptr++, HFA384x_DATA1);
                    i+=2;
                }
            if(len % 2)
            {
                data_lo = buf[i++];
                hfa384x_setreg(data_lo, HFA384x_DATA1);
            }
        }
    }
}
else if (bap == 0)
{
    /* Write id to select reg */
    hfa384x_setreg(id, HFA384x_SELECT0);
    /* Write offset to offset reg */
    hfa384x_setreg(offset, HFA384x_OFFSET0);
    /* Wait for offset[busy] to clear */
    do{
        reg = hfa384x_getreg(HFA384x_OFFSET0);
    }while(HFA384x_OFFSET_ISBUSY(reg));
    /* Check for error bit in offset reg if not error write in buffer path*/
```

```
    reg = hfa384x_getreg(HFA384x_OFFSET0);

    if(HFA384x_OFFSET_ISERR(reg))
        result = HFA384x_RES_CARDFAIL;
    else
    {
        if (txflag)
        {
            for(i=0;i<(len&0xFFFE);)
            {
                hfa384x_setdata(*dataptr++, HFA384x_DATA0);
                i+=2;
            }
            if(len % 2)
            {
                data_lo = buf[i++];
                hfa384x_setreg(data_lo, HFA384x_DATA0);
            }
        }
        else // if !txflag
        {
            for(i=0;i<(len&0xFFFE);)
            {
                hfa384x_setreg(*dataptr++, HFA384x_DATA0);
                i+=2;
            }
            if(len % 2)
            {
                data_lo = buf[i++];
                hfa384x_setreg(data_lo, HFA384x_DATA0);
            }
        }
    }
}
else //if read
{
    if (bap) // if bap = 1
    {
        /* Write id to select reg */
        hfa384x_setreg(id, HFA384x_SELECT1);
        /* Write offset to offset reg */
        hfa384x_setreg(offset, HFA384x_OFFSET1);
        /* Wait for offset[busy] to clear */
        do{
            reg = hfa384x_getreg(HFA384x_OFFSET1);
        }while(HFA384x_OFFSET_ISBUSY(reg));
        /* Check for error bit in offset reg if not error */
        reg = hfa384x_getreg(HFA384x_OFFSET1);

        if(HFA384x_OFFSET_ISERR(reg))
```

```
                result = HFA384x_RES_CARDFAIL;
            else
            {
                fidlen = hfa384x_getreg(HFA384x_DATA1);
                tmp_buff[0] = fidlen;
                if(fidlen)
                {
                    for(fid_ptr=0;fid_ptr<fidlen;fid_ptr++)
                    {
                        tmp_buff[fid_ptr+1] = hfa384x_getreg(HFA384x_DATA1);
                    }
                }
            }
        }
        else // if bap = 0
        {
            /* Write id to select reg */
            hfa384x_setreg(id, HFA384x_SELECT0);
            /* Write offset to offset reg */
            hfa384x_setreg(offset, HFA384x_OFFSET0);
            /* Wait for offset[busy] to clear */
            do{
                reg = hfa384x_getreg(HFA384x_OFFSET0);
            }while(HFA384x_OFFSET_ISBUSY(reg));
            /* Check for error bit in offset reg if not error */
            reg = hfa384x_getreg(HFA384x_OFFSET0);

            if(HFA384x_OFFSET_ISERR(reg))
                result = HFA384x_RES_CARDFAIL;
            else
            {
                fidlen = hfa384x_getreg(HFA384x_DATA0);
                tmp_buff[0] = fidlen;
                if(fidlen)
                {
                    for(fid_ptr=0;fid_ptr<fidlen;fid_ptr++)
                    {
                        tmp_buff[fid_ptr+1] = hfa384x_getreg(HFA384x_DATA0);
                    }
                }
            }
        }
    }

    /* Enable all IRQs */
    ENABLE_GLOBAL_IRQ();

    return(result);
}
```

```
/*-------------------------------------------------------------
* ReadCISTable
* Read The Card Information Structure (CIS)
-------------------------------------------------------------*/
void ReadCISTable(void)
{
    char xdata CISFlag;
    char xdata cor_addr_lo,cor_addr_hi;
    unsigned int xdata cis_addr = 0;
    char xdata code_tuple=0;
    char xdata link_tuple;
    char xdata tuple_link_cnt;
    char xdata tuple_data;
    char xdata funce_cnt = 0;
    char xdata mac_cnt = 0;

    /* Wait until read the first CIS tuple */
    CISFlag = 0;
    do{
        delay_nops(2);
        code_tuple = CFReadMemByte(0x0000);

        if(code_tuple == 1)
            CISFlag = 1;
    }while(!CISFlag);

// printf("Reading CIS Table\n")
    cor_addr = 0;
    CISFlag = 0;
    do{
        code_tuple = CFReadMemByte(cis_addr);
        cis_addr+=2;

        link_tuple = CFReadMemByte(cis_addr);
        cis_addr+=2;

//#if PRINT_WIFI
        printf("code_tuple[%2i] = 0x%Bx\n", cis_addr-4, (char)code_tuple);
//   printf("link_tuple[%2i] = 0x%Bx\n", cis_addr-2, (char)link_tuple);
//#endif

        if(code_tuple+1!=0) //if(code_tuple != CISTPL_END)
        {
            for(tuple_link_cnt=0;tuple_link_cnt<link_tuple;++tuple_link_cnt)
            {
                tuple_data = CFReadMemByte(cis_addr);
                cis_addr+=2;

#if PRINT_WIFI
                printf("CISTable[%2i] = 0x%Bx\n", cis_addr-2, (char)tuple_data);
```

```
#endif

                switch(code_tuple)
                {
                    case CISTPL_DEVICE: break;
                    case CISTPL_DEVICE_A: break;
                    case CISTPL_DEVICE_OC: break;
                    case CISTPL_DEVICE_OA: break;
                    case CISTPL_VERS_1:    break;
                    case CISTPL_MANFID:      break;
                    case CISTPL_FUNCID:
                        if(tuple_link_cnt == 0)
                            cis_device = tuple_data;
                        /* Stop read CIS table if card is a fixed disk */
                        if (cis_device == CISTPL_FUNCID_FIXED)
                            CISFlag = 1;
                    break;
                  case CISTPL_FUNCE:
                        if(funce_cnt == 6) // Prism Datasheet = 4
                            if(tuple_link_cnt>1)
                                uip_ethaddr.addr[mac_cnt++] = tuple_data;
                        if(tuple_link_cnt == (link_tuple-1))
                            funce_cnt++;
                    break;
                    case CISTPL_CONFIG:
                        if(tuple_link_cnt == 2)
                            cor_addr_lo = tuple_data;
                        if(tuple_link_cnt == 3)
                        {
                            cor_addr_hi = tuple_data;
                            cor_addr = ((unsigned int)cor_addr_hi<<8)+((unsigned int)cor_addr_lo&0x00FF);
                        }
                    break;

                    case CISTPL_CFTABLE_ENTRY: break;
                    default: break;
                }
            }
        }
        else
        {
            CISFlag = 1;
        }
    }while(!CISFlag);

#if PRINT_WIFI
    printf("COR address = 0x%x\n", cor_addr);
    printf("CIS device = 0x%Bx\n", (char)cis_device);
#endif
```

```
    return;
}
/*---------------------------------------------------------------
* hfa384x_drvr_start
*
* Issues the MAC initialize command, sets up some data structures,
* and enables the interrupts.  After this function completes, the
* low-level stuff should be ready for any/all commands.
*
----------------------------------------------------------------*/
void hfa384x_drvr_start(void)
{
    char xdata result = 0;
    char xdata cor_data;

    /* Reset CF card */
    CF_RESET_ON()
    delay_nops(4);
    CF_RESET_OFF();

    /* Set CF card to IO mode */
    cor_data = CFReadMemByte(cor_addr);
    delay_nops(5000);
    cor_data |= COR_FUNC_ENA; //0x01
    CFWriteMemByte(cor_data,cor_addr);

    /* Initialize command */
    result = hfa384x_cmd(HFA384x_CMDCODE_INIT, 0);
    switch(result)
    {
        case HFA384x_RES_SUCCESS:
            printf("Card Initialized\r\n");
        break;
        case HFA384x_RES_CARDFAIL:
            printf("Card Failure\r\n");
        break;
        case HFA384x_RES_NOBUFF:
            printf("No buffer space\r\n");
        break;
        case HFA384x_RES_COMMERR:
            printf("Command error\r\n");
        break;
        default:
            printf("Result Code = %x",result);
        break;
    }

    /* make sure interrupts are disabled and any events cleared */
    hfa384x_setreg(0, HFA384x_INTEN);
    hfa384x_setreg(0xffff, HFA384x_EVACK);
```

```
    return;
}

/*-----------------------------------------------------------------
* uIPwifi_api
*
* Implement an api for wifi parameters configuration
-----------------------------------------------------------------*/
void uIPwifi_api(unsigned int cmd, unsigned int val)
{
    char xdata i,j,k;
    char xdata SSID_name[] = "linksys"; // change it to support different SSIDs
    char xdata result = 0;

    switch(cmd)
    {
        case BSS:
            tmp_buff[0] = HFA384x_RID_CNFPORTTYPE_LEN;
            tmp_buff[1] = HFA384x_RID_CNFPORTTYPE;
            tmp_buff[2] = HFA384x_PORTTYPE_BSS;
            hfa384x_cmd_access(BAP_WRITE, HFA384x_RID_CNFPORTTYPE,
                               tmp_buff,
                               HFA384x_RID_CNFPORTTYPE_LEN);
        break;

        case IBSS:
            tmp_buff[0] = HFA384x_RID_CNFPORTTYPE_LEN ;
            tmp_buff[1] = HFA384x_RID_CNFPORTTYPE;
            tmp_buff[2] = HFA384x_PORTTYPE_IBSS;
            hfa384x_cmd_access(BAP_WRITE, HFA384x_RID_CNFPORTTYPE,
                               tmp_buff,
                               HFA384x_RID_CNFPORTTYPE_LEN);
        break;

        case SSID:
            tmp_buff[0] = HFA384x_RID_CNFDESIREDSSID_LEN;
            tmp_buff[1] = HFA384x_RID_CNFDESIREDSSID;
            tmp_buff[2] = strlen(SSID_name);
            /* assembly string vector in tmp_buff */
            j = strlen(SSID_name);
            i=0;
            k=3;
            while(j > 1) /* if SSID_name is even */
            {
                tmp_buff[k] = (SSID_name[i+1]) << 8 | SSID_name[i];
                i+=2;
                ++k;
                j-=2;
            }
```

```
    while(j > 0) /* if SSID name is odd put the last single byte in tmp_buff */
    {
        tmp_buff[k] = SSID_name[i];
        --j;
    }
    hfa384x_cmd_access(BAP_WRITE, HFA384x_RID_CNFDESIREDSSID,
                       tmp_buff,
                       HFA384x_RID_CNFDESIREDSSID_LEN);
break;

case MAX_LEN:
    tmp_buff[0] = HFA384x_RID_CNFMAXDATALEN_LEN;
    tmp_buff[1] = HFA384x_RID_CNFMAXDATALEN;
    tmp_buff[2] = val;
    hfa384x_cmd_access(BAP_WRITE, HFA384x_RID_CNFMAXDATALEN,
                      tmp_buff,
                      HFA384x_RID_CNFMAXDATALEN_LEN);
break;

case ENABLE:
    if (val)
    {
        /*enable the chip to work --> MACPORT = 0 */
        if (result = hfa384x_cmd(HFA384x_CMDCODE_ENABLE | HFA384x_CMD_MACPORT_SET(0), 0))
        {
            printf("Enable macport failed, result=0x%x.\n", result);
        }
    }
    else
    {
        /*disable the chip to work --> MACPORT = 0 */
        if (result = hfa384x_cmd(HFA384x_CMDCODE_DISABLE | HFA384x_CMD_MACPORT_SET(0), 0))
        {
             printf("Disable macport failed, result=0x%x.\n", result);
        }
    }
break;

case PM_ENABLE:
    tmp_buff[0] = HFA384x_RID_CNFPMENABLED_LEN;
    tmp_buff[1] = HFA384x_RID_CNFPMENABLED;
    tmp_buff[2] = val;
    hfa384x_cmd_access(BAP_WRITE, HFA384x_RID_CNFPMENABLED,
                       tmp_buff,
                       HFA384x_RID_CNFPMENABLED_LEN);
break;

case PM_MAX_SLEEP:
    tmp_buff[0] = HFA384x_RID_CNFMAXSLEEPDUR_LEN;
    tmp_buff[1] = HFA384x_RID_CNFMAXSLEEPDUR;
```

```
            tmp_buff[2] = val;
            hfa384x_cmd_access(BAP_WRITE, HFA384x_RID_CNFMAXSLEEPDUR,
                               tmp_buff,
                               HFA384x_RID_CNFMAXSLEEPDUR_LEN);
        break;

        default:
        break;
    }
    return;
}

/*---------------------------------------------------------------
* static void config_chip(void)
*
* Config some 8051 features
---------------------------------------------------------------*/
void config_chip(void)
{

#if PLL_DPM
    PLLCON &=0x00;
    PLLCON = PLLCON | 0x07; //CPU clk = 0,09MHz
#else
    PLLCON &= 0x00;   /* configura o clock interno = 12MHz */
#endif
    PLLCON = PLLCON | 0x08; /*configure Fast interrupt feature */

    /* Disable Ale output */
    DISABLE_ALE_OUT();

    // Serial Configuration
    TR1  = 0;        // Halt Timer 1.
    ET1  = 0;        // Disable Timer 1 interrupt
    SCON = 0x50;     /* mode 1, 8-bit uart, receptor enable */
    TMOD = 0x20;     /* timer 1, modo 2, 8-bit reload */
    TH1  = T1_RELOAD-1; // ?? bps
    TR1  = 1;        /* Start Serial Communication */
    TI   = 1;        /* Interrupt Flag set */

    /* RTC Configuration */
    SET_RTC_TMR(0,0,0);
    RTC_ENABLE();

    /* Power Supply Monitor Configuration */
    /*               Bit7  Bit6  Bit5  Bit4  Bit3  Bit2  Bit1  Bit0 */
    /* PSMCON =      CMPD  CMPA  PSMI  TPD1  TPD0  TPA1  TPA0  PSMEN */
    PSMCON = 0x1E; // 0      0     0     1     1     1     1      0   --> Trip Point = 2.63V

    /* Initialize Timer 0 to generate a periodic 24Hz interrupt. */
```

```
    // Stop timer/ counter 0.
    TR0  = 0;
    // Set timer/ counter 0 as mode 1 16 bit timer.
    TMOD &= 0xF0;
    TMOD |= 0x01;
    // Preload for 24Hz periodic interrupt.
    TH0 = WIFI_T0_RELOAD >> 8;
    TL0 = WIFI_T0_RELOAD;

/* Ports Configuration */
/* Port 1 */
/* All pins configured as inputs */
    DIN   = 0;
    PDET0 = 0;
    WAIT   = 0;
    CD1   = 0;
    LDCH  = 0;

    LED2 = 1;
    /* Port 0 - Set Data as inputs*/
    CF_DATA_PORT = CF_DATA_PORT_MASK;

    /* Cf signals */
    CF_WR_ON();
    CF_RD_ON();
    CF_REG_ON();
    CF_RESET_OFF();
    CF_IORD_ON();
    CF_IOWR_ON();

    printf("Starting 8051 WIFI uIP TCP/IP Stack\n");

    /* enable INT0 Interrupt */
    ENABLE_INT0_IRQ();
    // Restart timer/ counter 0 running.
    TR0 = 1;
    // Enable timer/ counter 0 overflow interrupt.
    ET0 = 1;
    // Enable global interrupt.
    ENABLE_GLOBAL_IRQ();
}

/*-------------------------------------------------------------
/ wifidev_init()
/
/ This function init the wifi CF card
-------------------------------------------------------------*/
void wifidev_init(void)
{
    char xdata result = 0;
```

```
    char xdata i;
    unsigned int xdata timeout_count;

#if PLL_DPM
    /*configure PLL to 12,58MHz frequency */
    PLLCON &= 0xF8;
#endif

    /*enable WDT for 2seg timeout*/
    CONFIG_WDT_2SEG();

    /*start drv performing CF initialization and initialization command*/
    hfa384x_drvr_start();

    /* Diagnose card before enable */
    result = hfa384x_cmd_diagnose();
    if (result)
    {
        printf("Diagnose card failed, result= 0x%x.\n", result);
        return;
    }
#if PRINT_WIFI
    else
        printf("Diagnose card passed!!!!\n");
#endif

    /*configure the PRISM MAC for STA operation - Infra Structure mode*/
    /*set MAC port to 1 */
    uIPwifi_api(BSS, 1);

    /* Set the desired SSID */
    uIPwifi_api(SSID, 1);

    /* lets extend the data length a bit */
    uIPwifi_api(MAX_LEN, 0x05dc); // 1500 bytes = Ethernet MAC data size

    /* Read card MAC address and set as board MAC address */
    hfa384x_cmd_access(BAP_READ, HFA384x_RID_CNFOWNMACADDR, NULL, 0);
    for(i=0;i<MAC_LEN;i++)
    {
        if(i%2)
            uip_ethaddr.addr[i] = tmp_buff[2+i/2]>>8;
        else
            uip_ethaddr.addr[i] = tmp_buff[2+i/2]& 0xFF;
    }

#if PRINT_WIFI
    printf("MAC Address:\n");
    for(i=0;i<MAC_LEN;i++)
        printf("uip_ethaddr.addr[%Bd] = 0x%Bx\n", (char)i, (char)uip_ethaddr.addr[i]);
```

```
#endif

    /* Alloc TX Buffers */
    result = hfa384x_alloc_tx_buff();

    /* Alloc Info Buffers */
    result = hfa384x_alloc_info_buff();

    /* Acknolegde all events */
    hfa384x_setreg(0xffff, HFA384x_EVACK);

    /*Read the current power mode */
    hfa384x_cmd_access(BAP_READ, HFA384x_RID_CURRENTPOWERSTATE, NULL, 0);

#if PRINT_WIFI
    printf("Power mode is 0x%x\n", tmp_buff[2]);
#endif

    /*enable the chip to work --> MACPORT = 0 */
    uIPwifi_api(ENABLE, ON);

    /* Read MAC Port Status */
    do{
        hfa384x_cmd_access(BAP_READ, HFA384x_RID_PORTSTATUS, NULL, 0);
        timeout_count++;
    }while((tmp_buff[2]!=4)&&(timeout_count<30000)); // 4 = Connected to ESS
    timeout_count = 0;

    if (tmp_buff[2]!=4)
        printf("Wireless Connection Failed. Please Try again later. \n");

#if PRINT_WIFI
    /* Read The current BSSID which the station is connected */
    hfa384x_cmd_access(BAP_READ, HFA384x_RID_CURRENTBSSID, NULL, 0);
    for(i=0;i<MAC_LEN;i++)
    {
        if(i%2)
            bssidc[i] = tmp_buff[2+i/2]>>8;
        else
            bssidc[i] = tmp_buff[2+i/2]& 0xFF;
    }

    for(i=0;i<MAC_LEN;i++)
        printf("BSSID[%Bd] = 0x%Bx\n", (char)i, (char)bssidc[i]);

    /* Read CurrentTXrate fid */
    hfa384x_cmd_access(BAP_READ, HFA384x_RID_CURRENTTXRATE, NULL, 0);
    printf("Tx rate is %d\n", tmp_buff[2]);

    /* Read Commsquality */
```

```
    hfa384x_cmd_access(BAP_READ, HFA384x_RID_COMMSQUALITY, NULL, 0);
    printf("CQ.currBSS is %d\n", tmp_buff[2]);
    printf("ASL.currBSS is %d\n", tmp_buff[3]);
    printf("ANL.currFC is %d\n", tmp_buff[4]);
#endif

    /* reset WDE bit to clear WDT*/
    RESET_WDE();

    return;
}

/*-------------------------------------------------------------
/ wifidev_read()
/
/ This function read data from wifi CF card
-------------------------------------------------------------*/
unsigned int wifidev_read(void)
{
    int xdata rxfid;
    int xdata i, j;
    int xdata reg;
    char xdata result = 0;
    char xdata bap;
    unsigned int xdata data_len;

    /*enable WDT for 2seg timeout*/
    CONFIG_WDT_2SEG();

    /* reset uip_len and data_len*/
    uip_len =0;
    data_len = 0;

    /* read the EvStat register for enabled events */
    reg = hfa384x_getreg(HFA384x_EVSTAT);

    /* if event RX proceed receiving data */
    if (HFA384x_EVSTAT_ISRX(reg))
    {
#if PRINT_WIFI
        printf("I'm receiving something\n");
#endif
        /* Get the FID */
        rxfid = hfa384x_getreg(HFA384x_RXFID);

        bap = get_free_bap();

        if (bap) //if bap =1
        {
            /* Write id to select reg */
```

```
hfa384x_setreg(rxfid, HFA384x_SELECT1);
/* Write offset to offset reg to start from DataLen in the structure */
hfa384x_setreg(RXDATALEN_OFF, HFA384x_OFFSET1);
/*wait for offset busy bit to set */
do{
    reg = hfa384x_getreg(HFA384x_OFFSET1);
}while(HFA384x_OFFSET_ISBUSY(reg));
/* Check for errors */
if (HFA384x_OFFSET_ISERR(reg))
{
    result = HFA384x_RES_CARDFAIL;
    /* acknowledge RX event in EvAck Reg */
    hfa384x_setreg(HFA384x_EVACK_RX_SET(1), HFA384x_EVACK);
}
else // if not error
{
    /* Read packet data len */
    data_len = hfa384x_getreg(HFA384x_DATA1);

    /* Check if Data len is bigger than UIP_BUFSIZE,
       if yes set data len to UIP_BUFSIZE */
    if(data_len > UIP_BUFSIZE)
        data_len = UIP_BUFSIZE-UIP_LLH_LEN;

    /* Start to fill uip_buf */
    /* Read Destination MAC address */
    j=0;
    for(i=0;i<3;++i)
    {
        *(int*)(&(uip_buf[ETH_PCKT_DST+j])) = hfa384x_getdata(HFA384x_DATA1);
        j+=2;
    }
    /* Read Source MAC address */
    j=0;
    for(i=0;i<3;++i)
    {
        *(int*)(&(uip_buf[ETH_PCKT_SRC+j])) = hfa384x_getdata(HFA384x_DATA1);
        j+=2;
    }

    /* read to get data Len 802.3 and snap */
    j=0;
    for(i=ETH_PCKT_LEN03;i<ETH_PCKT_DATA;++i)
    {
        *(int*)(&(uip_buf[ETH_PCKT_LEN03+j])) = hfa384x_getdata(HFA384x_DATA1);
        j+=2;
    }

    /* Read data(even) and put in uip_buf */
    j=0;
```

```
                for(i=0;j<(data_len & 0xfffe);++i)
                {
                    *(int*)(&(uip_buf[UIP_LLH_LEN+j])) = hfa384x_getdata(HFA384x_DATA1);
                    j+=2;
                }
                /* If len odd, handle last byte */
                if ( data_len % 2 )
                {
                    reg = hfa384x_getreg(HFA384x_DATA1);
                    uip_buf[data_len + UIP_LLH_LEN -1] = ((char*)(&reg))[0];
                }
            }
        }
        else // if bap =0
        {
            /* Write id to select reg */
            hfa384x_setreg(rxfid, HFA384x_SELECT0);
            /* Write offset to offset reg to start from DataLen in the structure */
            hfa384x_setreg(RXDATALEN_OFF, HFA384x_OFFSET0);
            /*wait for offset busy bit to set */
            do{
                reg = hfa384x_getreg(HFA384x_OFFSET0);
            }while(HFA384x_OFFSET_ISBUSY(reg));
            /* Check for errors */
            if (HFA384x_OFFSET_ISERR(reg))
            {
                result = HFA384x_RES_CARDFAIL;
                /* acknowledge RX event in EvAck Reg */
                hfa384x_setreg(HFA384x_EVACK_RX_SET(1), HFA384x_EVACK);
                //return (int)result;
            }
            else // if not error
            {
                /* Read packet data len */
                data_len = hfa384x_getreg(HFA384x_DATA0);
#if PRINT_WIFI
                printf("data_len = 0x%x\n", data_len);
#endif
                /* Check if Data len is bigger than UIP_BUFSIZE,
                   if yes set data len to UIP_BUFSIZE */
                if(data_len > UIP_BUFSIZE)
                    data_len = UIP_BUFSIZE-UIP_LLH_LEN;

                /* Start to fill uip_buf */
                /* Read Destination MAC address */
                j=0;
                for(i=0;i<3;++i)
                {
                    *(int*)(&(uip_buf[ETH_PCKT_DST+j])) = hfa384x_getdata(HFA384x_DATA0);
                    j+=2;
```

```
            }
            /* Read Source MAC address */
            j=0;
            for(i=0;i<3;++i)
            {
                *(int*)(&(uip_buf[ETH_PCKT_SRC+j])) = hfa384x_getdata(HFA384x_DATA0);
                j+=2;
            }

            /* read to get data Len 802.3 and snap */
            j=0;
            for(i=ETH_PCKT_LEN03;i<ETH_PCKT_DATA;++i)
            {
                *(int*)(&(uip_buf[ETH_PCKT_LEN03+j])) = hfa384x_getdata(HFA384x_DATA0);
                j+=2;
            }

            /* Read data(even) and put in uip_buf */
            j=0;
            for(i=0;j<(data_len & 0xfffe);++i)
            {
                *(int*)(&(uip_buf[UIP_LLH_LEN+j])) = hfa384x_getdata(HFA384x_DATA0);
                j+=2;
            }
            /* If len odd, handle last byte */
            if ( data_len % 2 )
            {
                reg = hfa384x_getreg(HFA384x_DATA0);
                uip_buf[data_len + UIP_LLH_LEN -1] = ((char*)(&reg))[0];
            }
        }
    }
    /* acknowledge RX event in EvAck Reg */
    hfa384x_setreg(HFA384x_EVACK_RX_SET(1), HFA384x_EVACK);

    /* Setup type field 802.3 header */
    /* Copy Snap type field over Ether header len */
    /*in order to create a original ethernet header */
    uip_buf[ETH_PCKT_LEN03] = uip_buf[0x14];
    uip_buf[ETH_PCKT_LEN03+1] = uip_buf[0x15];
}
else // if not event RX
{
    hfa384x_event_ack();
}

/* reset WDE bit to clear WDT*/
RESET_WDE();

if (data_len > 0 )
```

```
    {
        return (data_len+UIP_LLH_LEN);
    }
    else
        return 0;
}

/*---------------------------------------------------------------
* hfa384x_event_ack
* Perform ack on Events
---------------------------------------------------------------*/
void hfa384x_event_ack(void)
{
    unsigned int xdata evstat_data, fid;

    evstat_data = hfa384x_getreg(HFA384x_EVSTAT);

    /* Ack TX event */
    if(HFA384x_EVSTAT_ISTX(evstat_data))
    {
        hfa384x_setreg(HFA384x_EVACK_TX_SET(1), HFA384x_EVACK);
    }
    /* Ack TXEXC event */
    if(HFA384x_EVSTAT_ISTXEXC(evstat_data))
    {
        hfa384x_setreg(HFA384x_EVACK_TXEXC_SET(1), HFA384x_EVACK);
    }
    /* Ack ALLOC event */
    if(HFA384x_EVSTAT_ISALLOC(evstat_data))
    {
        hfa384x_setreg(HFA384x_EVACK_ALLOC_SET(1), HFA384x_EVACK);
    }
    /* Ack CMD event */
    if(HFA384x_EVSTAT_ISCMD(evstat_data))
    {
        hfa384x_setreg(HFA384x_EVACK_CMD_SET(1), HFA384x_EVACK);
    }
    /* Ack INFO event */
    if(HFA384x_EVSTAT_ISINFO(evstat_data))
    {
        fid = hfa384x_getreg(HFA384x_INFOFID); // TBU
        hfa384x_setreg(HFA384x_EVACK_INFO_SET(1), HFA384x_EVACK);
    }
    /* Ack INFDROP event */
    if(HFA384x_EVSTAT_ISINFDROP(evstat_data))
    {
        hfa384x_setreg(HFA384x_EVACK_INFDROP_SET(1), HFA384x_EVACK);
    }
    /* Ack WTERR event */
    if(HFA384x_EVSTAT_ISWTERR(evstat_data))
```

```
    {
        hfa384x_setreg(HFA384x_EVACK_WTERR_SET(1), HFA384x_EVACK);
    }
}


/*---------------------------------------------------------------
* wifidev_send()
*
* This function write data to wifi CF card
---------------------------------------------------------------*/
void wifidev_send(void)
{
    char xdata result = 0;
    int xdata fid = 0;
    int xdata reg;
    int xdata i=0;
    unsigned int xdata timeout_count;

    /*enable WDT for 2seg timeout*/
    CONFIG_WDT_2SEG();

#if PRINT_WIFI
    printf("I'm transmiting something\n");
    printf("uip_len = 0x%x\n", uip_len);
#endif

    /* Change Type field in eth header to length field in uip_buf*/
    uip_buf[12] = (char)(uip_len>>8);
    uip_buf[13] = (char)(uip_len&0xFF);

    /* fill uip_buf with tcp appdata */
    if(uip_buf[31] == UIP_PROTO_TCP)
    {
        for(i=40 + UIP_LLH_LEN; i< uip_len; i++)
        {
            uip_buf[i] = *uip_appdata;
            uip_appdata++;
        }
    }

    /* Allocate TX FID */
    fid = txfid_queue[--txfid];
    if(txfid==0)
        txfid= HFA384x_DRVR_FIDSTACKLEN_MAX;

    /*copy data to bap using fid*/
    txflag=1;
    result = hfa384x_rdwr_bap(BAP_WRITE, fid, TX_DEST_ADDR_OFF,
                              uip_buf, uip_len);
```

```
    txflag=0;

    /* issue TX command */
    result = hfa384x_cmd(HFA384x_CMDCODE_TX | HFA384x_CMD_RECL_SET(1), fid);

    /* Check for TXOk bit */
    do{
        reg = hfa384x_getreg(HFA384x_EVSTAT);
        timeout_count++;
    }while((!HFA384x_EVSTAT_ISTX(reg)) && (timeout_count<10));
    timeout_count = 0;

    /* if TX command fail restore fid */
    if (result)
    {
        txfid++;
        printf("TX command failed. Result = 0x%x\n", result);
    }

    /*ack TX and other events */
    hfa384x_event_ack();

    /* reset uip_len */
    uip_len =0;

    /* reset WDE bit to clear WDT*/
    RESET_WDE();

    return;
}


/*---------------------------------------------------------------
* find_CF_mode()
*
* This function find the CF mode: fixed disk(memory) or IO network
---------------------------------------------------------------*/
void find_CF_mode(void)
{

    unsigned int xdata connect_timeout = 0;

    /* Check if CF is connected for 10 seconds*/
    while ((connect_timeout <50)&&(!CFConnected()))
    {
        if(!CFConnected())
        {
            printf("CF not connected. Connect the card and Try again.\n");
            //return;
        }
```

```
        connect_timeout++;
    }

    if(!CFConnected())
    {
        cis_device = CISTPL_FUNCID_NOT_SUPPORTED;
        return;
    }

    /* Reset CF card */
    CF_RESET_ON()
    delay_nops(4);
    CF_RESET_OFF();

    /* Read CIS table */
    ReadCISTable();

    /* if card not supported reset cis_device variable*/
    if ((cis_device != CISTPL_FUNCID_NETWORK)&&(cis_device != CISTPL_FUNCID_FIXED))
        cis_device = CISTPL_FUNCID_NOT_SUPPORTED;

    /*if cf card = fixed disk disable ECG send button switch */
    if (cis_device == CISTPL_FUNCID_FIXED)
    {
        DISABLE_INT0_IRQ();
        if (CFIDDrive())
        printf("Identify drive command error\n");
    }

#if PRINT_WIFI
    printf("cis_device = 0x%Bx\n", cis_device);
#endif

    return;
}
```

B.3 Código do módulo de memória da *Compact Flash*

```
/*********************************************************************
* Filename: cfdev.h
* Description: This file implements the header file compact flash
*              functions in CF memory mode
* Developed by: Marco Carvalho
* Created on: 30/03/05
/*********************************************************************/


#ifndef __COMPACT_FLASH_H
```

```
#define __COMPACT_FLASH_H

//defines

//CF memory registers
#define CF_IO 0x00 //registro para IO
#define CF_FEATURES 0x01 //registro de Errors (Out) e Features (In)
#define CF_ERROR 0x01 //registro de Errors (Out) e Features (In)
#define CF_SECCOUNT 0x02 //numero de setores transferidos
#define CF_LBA0 0x03 //LBA 0-7
#define CF_LBA1 0x04 //LBA 8-15
#define CF_LBA2 0x05 //LBA 16-23
#define CF_LBA3 0x06 //LBA 24-27 -> LBA bit = 1
#define CF_STACOM 0x07 //registro de Status (Out) e Command (In)

//CF memory Comands
#define CF_READ_SEC 0x20 //read
#define CF_WRITE_SEC    0x30 //write
#define CF_DRIVE_DIAG   0x90
#define CF_ID_DRIVE  0xec // identify drive


#define CF_STACOM_ERROR_MASK   0x01
#define CF_STACOM_DRQ_OK_MASK  0x58 //D6=RDY=1 D4=DSC=1 D3=DRQ=1
#define CF_STACOM_DRQ_MASK     0xF8
#define CF_STACOM_BUSY_MASK    0x80
#define CF_STACOM_BUSY_OK_MASK 0x00  //D8=0
#define CF_STACOM_RDY_MASK     0xF0
#define CF_STACOM_RDY_OK_MASK  0x50 // D6=RDY=1 e D4=DSC=1

#define CF_WAIT_DRQ()  CFWait(CF_STACOM_DRQ_MASK, CF_STACOM_DRQ_OK_MASK)
#define CF_WAIT_RDY()  CFWait(CF_STACOM_RDY_MASK, CF_STACOM_RDY_OK_MASK)
#define CF_WAIT_BUSY() CFWait(CF_STACOM_BUSY_MASK, CF_STACOM_BUSY_OK_MASK)

// registers

#define RD_DATA 0x0
#define WR_DATA 0x0
#define ERROR_REG 0x1
#define FEATURES 0x1
#define SECTOR_CNT 0x2
#define SECTOR_NO 0x3
#define CYLINDER_LO 0x4
#define CYLINDER_HI 0x5
#define DRIVE_HEAD 0x6
#define STATUS 0x7
#define COMMAND 0x7
#define ALT_STATUS 0xe
#define DEVICE_CTRL 0xe
#define DRIVE_ADDRESS 0xf
```

```
// register bits

#define BAD_BLOCK 0x80
#define UNCORRECTABLE 0x40
#define SECTOR_ID_ERROR 0x10
#define ABORT 0x04
#define GENERAL_ERROR 0x01

#define LBA 0x40
#define DRIVE_NO 0x10

#define BUSY 0x80
#define READY 0x40
#define WRITE_FAULT 0x20
#define CARD_READY 0x10
#define DATA_REQUEST 0x08
#define CORRECTABLE_ERROR 0x04
#define ERROR 0x01

#define BYTE_PER_SEC 512

// CF registers

#define COR_REG 0x0200
#define SOCKET_COPY_REG 0x0206
//#define SOCKET_COPY_REG cor_addr+6


// COR register values

#define MEMORY_MAPPED 0
#define IO_MAPPED 1

/* external variables */
extern unsigned long xdata CFNumSectors;

/* Function Prototypes CF */
unsigned char CFSendCommand(unsigned char cmd, unsigned long address,unsigned char nSectors);

bit CFReadSector(unsigned long address);
bit CFWriteSector(unsigned long address);

unsigned char CFWait(unsigned char cf_stacom_mask, unsigned char cf_stacom_ok_mask);
char CFIDDrive(void);
void CFWriteCommonMemByte(char val, unsigned int reg);
char CFReadCommonMemByte(unsigned int reg);

/* end Function prototypes CF memory */
```

```
#endif /* COMPACT_FLASH_H */

/*********************************************************************
* Filename: cfdev.c
* Description: This file implements the compact flash functions for
CF memory mode
* Developed by: Marco Carvalho
* Created on: 10/03/05
/*********************************************************************/

#include "cfdev.h"
#include "wifidev.h"
#include <ctype.h>
#include <stdio.h>

unsigned char xdata CFDrvNumber=0;
unsigned long int xdata CFNumSectors=0;

unsigned char CFSendCommand(unsigned char cmd, unsigned long address,unsigned char nSectors)
{
    unsigned char tmp_data;

    CF_WAIT_BUSY();
    /* Set the drive head */
    CFWriteCommonMemByte((0xe0|CFDrvNumber)|(0xf&(address>>24)),DRIVE_HEAD);
    /* Set cylinder high */
    CFWriteCommonMemByte(0xff&(address>>16),CYLINDER_HI);
    /* Set cylinder low */
    CFWriteCommonMemByte(0xff&(address>>8),CYLINDER_LO);
    /* Set sector number */
    CFWriteCommonMemByte(0xff&address,SECTOR_NO);
    /* Set sector quantity */
    CFWriteCommonMemByte(nSectors,SECTOR_CNT);
    /* Set command */
    CFWriteCommonMemByte(cmd,COMMAND);

    CF_WAIT_BUSY();
    CF_WAIT_DRQ();
    tmp_data = CF_WAIT_BUSY();
    if (tmp_data)
    {
        printf("Erro no registrador de comandos \n");
        return 1;
    }
    else
        return 0;

}

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

```
* CFReadSector(unsigned long address)
*
* Read one sector from CF card
* The sector read will be found in uip_buf after call this
* function
-------------------------------------------------------------------*/
bit CFReadSector(unsigned long address)
{
    unsigned int xdata i;

    /* send Read sector command */
    if(CFSendCommand(CF_READ_SEC,address,1))
        return 1;

    /* read one sector */
    for(i=0;i<BYTE_PER_SEC;i++)
        uip_buf[i]= CFReadCommonMemByte(RD_DATA);

    return 0;
}

/*-------------------------------------------------------------------
* CFWriteSector(unsigned long address)
*
* The sector array to be write must be placed in uip_buf before
* call this function
-------------------------------------------------------------------*/
bit CFWriteSector(unsigned long address)
{
    unsigned int xdata i;

    /* send write sector command */
    if(CFSendCommand(CF_WRITE_SEC,address,1))
        return 1;

    /* write one sector = 512 bytes */
    for(i=0;i<BYTE_PER_SEC;i++)
        CFWriteCommonMemByte(uip_buf[i],WR_DATA);

    return 0;

}

/*-------------------------------------------------------------------
*CFWait(void)
* Wait for some bit masked by cf_stacom_mask until its value becomes
* cf_stacom_ok_mask
-------------------------------------------------------------------*/
unsigned char CFWait(unsigned char cf_stacom_mask, unsigned char cf_stacom_ok_mask)
{
```

```
    unsigned char xdata tmp_data;

    tmp_data=CFReadCommonMemByte(CF_STACOM); //status register read
    if(tmp_data & CF_STACOM_ERROR_MASK) return 1; //error if tmp = 0x01 !

    do
    {
        tmp_data=CFReadCommonMemByte(CF_STACOM); //read status register content
        tmp_data&=cf_stacom_mask; //
    }while(tmp_data!=cf_stacom_ok_mask);

    return 0;
}

/*---------------------------------------------------------------
* IDDrive()
* Identify drive to discover head/cylinder/sectors configuration
---------------------------------------------------------------*/
char CFIDDrive(void)
{
    unsigned int xdata i;
    unsigned char xdata Num_sectors[4];

    /* Get CF drive number */
    CFDrvNumber=0x1&(CFReadMemByte(SOCKET_COPY_REG)>>4);
    printf("Drive number is %Bx\n",(char)CFDrvNumber);
    /* Set Operation mode = Memory mapped */
    cor_data = CFReadMemByte(cor_addr);
    delay_nops(5000);
    cor_data = MEMORY_MAPPED|(cor_data&0xe0);
    CFWriteMemByte(cor_data,cor_addr);

    if(CFSendCommand(CF_ID_DRIVE,0,0))
    {
        printf("Identify drive command error\n");
        return 1;
    }

    for(i=0;i<=123;i++)
    {
        uip_buf[i]=CFReadCommonMemByte(RD_DATA);
        if (i>=120)
            Num_sectors[i-120] = uip_buf[i];
    }

    CFNumSectors = (unsigned long)(Num_sectors[3]<<24)|
                   (unsigned long)(Num_sectors[2]<<16)|
                   (unsigned long)(Num_sectors[1]<<8)|
                   (unsigned long)(Num_sectors[0]);
```

```
    printf("Mumber of sectors (LBA mode) = 0x%Lx\n", CFNumSectors);

    return 0;
}

/*-----------------------------------------------------------------
* CFWriteCommonMemByte
*
* write a byte in the CF Common memory
-----------------------------------------------------------------*/
static void CFWriteCommonMemByte(char val, unsigned int reg)
{

    CF_REG_ON();

    CFSetAddr(reg); // Set Mem reg to write in CF

    CF_DATA_PORT=val; //escrita no PORT0=buf do 8051

    CF_WR_OFF(); // write=0
    CF_WR_ON(); // write=1

    CF_DATA_PORT = CF_DATA_PORT_MASK;

    return ;
}

/*-----------------------------------------------------------------
* CFReadCommonMemByte
*
* Read a byte from the CF common memory
-----------------------------------------------------------------*/
char CFReadCommonMemByte(unsigned int reg)
{
    char xdata val;

    CF_REG_ON();

    CFSetAddr(reg); // Set Mem reg to read from CF

    CF_RD_OFF(); //read=0

    val = CF_DATA_PORT;  // val = P0 (8051)
    CF_RD_ON(); //read=1

    return val;
}
```

B.4 Código da Aplicação de rede cliente

```
/*********************************************************************
* Filename: webclient.h
* Description: This file implements a header to HTTP client for ECG board
* Developed by: Marco Carvalho
* Created on: 30/03/05
/*********************************************************************/

#ifndef __WEBCLIENT_H__
#define __WEBCLIENT_H__

#include <stdio.h>
#include "config.h"
#include "http-strings.h"
#include "ecg_app.h"

#define UIP_APPCALL     webclient_appcall

#define WWW_CONF_MAX_URLLEN 10
#define MAX_HOST_LEN 15
#define MAX_HTTP_HDR_LEN 20

struct webclient_state {
    u8_t timer;
    u8_t state;
    u8_t httpflag;

    u16_t port;
    char host[MAX_HOST_LEN];

    char file[WWW_CONF_MAX_URLLEN];

    u16_t postrequestptr;
    u16_t postrequestleft;

    char httpheaderline[MAX_HTTP_HDR_LEN];
    u16_t httpheaderlineptr;
};

extern xdata struct webclient_state s;

xdata struct webclient_state;

extern unsigned char xdata ecg_send_button;

/* UIP_APPSTATE_SIZE: The size of the application-specific state
stored in the uip_conn structure. */
#ifndef UIP_APPSTATE_SIZE
#define UIP_APPSTATE_SIZE (sizeof(struct webclient_state))
```

```
#endif

/**
 * Callback function that is called from the webclient code if the
 * HTTP connection to the web server has timed out.
 *
 * This function must be implemented by the module that uses the
 * webclient code.
 */
void webclient_timedout(void);

/**
 * Initialize the webclient module.
 */
void webclient_init(void);

unsigned char webclient_post(char *host, u16_t port, char *file);

void webclient_appcall(void);

#endif /* __WEBCLIENT_H__ */

/** @} */

/*****************************************************************
* Filename: webclient.c
* Description: This file implements a HTTP client for ECG board
* Developed by: Marco Carvalho
* Created on: 30/03/05
*****************************************************************/

#include <stdio.h>

#include "uip.h"
#include "webclient.h"

#include <string.h>

#define WEBCLIENT_TIMEOUT 100

#define WEBCLIENT_STATE_STATUSLINE 0
#define WEBCLIENT_STATE_HEADERS    1
#define WEBCLIENT_STATE_DATA       2
#define WEBCLIENT_STATE_CLOSE      3
#define WEBCLIENT_STATE_SENDDATA   4

#define HTTPFLAG_NONE   0
#define HTTPFLAG_OK     1
#define HTTPFLAG_MOVED  2
#define HTTPFLAG_ERROR  3
```

```
#define ISO_nl       0x0a
#define ISO_cr       0x0d
#define ISO_space    0x20

#define WEBCLIENT_CLOSED()  ENABLE_TMR0_IRQ()

#define WEBCLIENT_ABORTED() ENABLE_TMR0_IRQ(); \
s.state = WEBCLIENT_STATE_CLOSE; \

#define WEBCLIENT_TIMEDOUT() WEBCLIENT_ABORTED()

xdata struct webclient_state s;

unsigned char xdata ecg_send_button;

/*---------------------------------------------------------------------
* webclient_init
*
* Setup some initial client application parameters
---------------------------------------------------------------------*/
void webclient_init(void)
{
    ecg_send_button = 0;
    uip_flags = UIP_CLOSE;
    s.state = WEBCLIENT_STATE_STATUSLINE;
    ecg_state = ECG_STATE_DPM_IDLE;
    tick_count = 0;
}

/*---------------------------------------------------------------------
* webclient_post(void)
*
* Open a http connection with server
---------------------------------------------------------------------*/
unsigned char webclient_post(char *host , u16_t port, char *file)
{
    struct uip_conn xdata *conn;
    static u16_t xdata ipaddr[2];

    uip_ipaddr(ipaddr, 192,168,1,50);
    conn = uip_connect(ipaddr, htons(port));

    printf("Try to open a connection with the server\n");

    if(conn == NULL)
    {
        printf("conexão nula retornando 0\n");
        return 0;
    }
```

```
    /* Setup s structure */
    s.port = port;
    strncpy(s.file, file, sizeof(s.file));
    strncpy(s.host, host, sizeof(s.host));
    s.postrequestleft = sizeof(http_post) - 1 + 1 +
                        sizeof(http_10) - 1 +
                        sizeof(http_crnl) - 1 +
                        sizeof(http_host) - 1 +
                        sizeof(http_crnl) - 1 +
                        sizeof(http_content_type) - 1 +
                        sizeof(http_app_wwwformurlenc_type) -1 +
                        sizeof(http_crnl) - 1 +
                        strlen(http_user_agent_fields) +
                        sizeof(http_crnl) - 1 +
                        strlen(s.file) + strlen(s.host) +
                        sizeof(http_crnl) - 1 + sizeof(http_crnl) - 1 +
                        sizeof(http_param) - 1;
    s.postrequestptr = 0;
    s.httpheaderlineptr = 0;

   return 1;
}

/*---------------------------------------------------------------
* copy_string()
*
* Just copy a string and return lenght
---------------------------------------------------------------*/
static unsigned char copy_string(unsigned char *dest,
    const unsigned char *src, unsigned char len)
{
    strcpy(dest, src);
    return len;
}

/*---------------------------------------------------------------
* static void senddata(void)
* Send http post header
---------------------------------------------------------------*/
static void senddata(void)
{
    u16_t xdata len;
    char xdata *postrequest;
    char xdata *cptr;

    if(s.postrequestleft > 0)
    {
        cptr = (char *)uip_appdata;
        postrequest = (char *)uip_appdata;
```

```
        /*create http post header */
        /* POST ecg.php http/1.0 */
        cptr += copy_string(cptr, http_post, sizeof(http_post) - 1);  //POST
        cptr += copy_string(cptr, s.file, strlen(s.file));  //ecg.php
        *cptr++ = ISO_space;  //
        cptr += copy_string(cptr, http_10, sizeof(http_10) - 1);      //http/1.0

        cptr += copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);  //<CR> <NL>

        /* Host: 192.168.1.50 */
        cptr += copy_string(cptr, http_host, sizeof(http_host) - 1);  //Host:
        cptr += copy_string(cptr, s.host, strlen(s.host));  //192.168.1.50

        cptr += copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);  //<CR> <NL>

        /* User-Agent: ECG Webclient */
        cptr += copy_string(cptr, http_user_agent_fields,      //User-Agent: ECG Webclient
        strlen(http_user_agent_fields));

        cptr += copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);  //<CR> <NL>

        /*Content-Type: application/x-www-form-urlencoded */
        //Content-Type:
        cptr += copy_string(cptr, http_content_type, sizeof(http_content_type) - 1)
        //application/x-www-form-urlencoded
        cptr += copy_string(cptr, http_app_wwwformurlenc_type,
                        sizeof(http_app_wwwformurlenc_type) - 1);

        cptr += copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);  //<CR> <NL>
        cptr += copy_string(cptr, http_crnl, sizeof(http_crnl) - 1);  //<CR> <NL>

        /* ecg_data= */
        cptr += copy_string(cptr, http_param, sizeof(http_param) - 1);  //http_ecgdata

        /* call ecg prepare data function */
        ecg_prepare_data();

        /*format post request lenght */
        len = s.postrequestleft+uip_len > uip_mss()? uip_mss():s.postrequestleft+uip_len;

        /* send http post header */
        uip_send(&(postrequest[s.postrequestptr]), len);
    }
}
/*---------------------------------------------------------------
* void acked(void)
*
---------------------------------------------------------------*/
static void acked(void)
```

```
{
    u16_t xdata len;

    if(s.postrequestleft > 0)
    {
        len = s.postrequestleft > uip_mss()? uip_mss():s.postrequestleft;
        s.postrequestleft -= len;
        s.postrequestptr += len;
    }
}
/*---------------------------------------------------------------
* parse_statusline
* parse satus line to get Ok from remote server
---------------------------------------------------------------*/
static u16_t parse_statusline(u16_t len)
{
    char xdata *cptr;

    while(len > 0 && s.httpheaderlineptr < sizeof(s.httpheaderline))
    {
        s.httpheaderline[s.httpheaderlineptr] = *uip_appdata;
        ++uip_appdata;
        --len;
        if(s.httpheaderline[s.httpheaderlineptr] == ISO_nl)
        {

            if((strncmp(s.httpheaderline, http_10, sizeof(http_10) - 1) == 0) ||
           (strncmp(s.httpheaderline, http_11, sizeof(http_11) - 1) == 0))
            {
                cptr = &(s.httpheaderline[9]);
                s.httpflag = HTTPFLAG_NONE;
                if(strncmp(cptr, http_200, sizeof(http_200) - 1) == 0)
                {
                    /* 200 OK */
                    s.httpflag = HTTPFLAG_OK;
                    printf("Received 200 OK\n");
                }
                else if(strncmp(cptr, http_301, sizeof(http_301) - 1) == 0 ||
                        strncmp(cptr, http_302, sizeof(http_302) - 1) == 0)
                {
                    /* 301 Moved permanently or 302 Found. Location: header line
                       will contain the new location. */
                    s.httpflag = HTTPFLAG_MOVED;
                }
                else
                {
                    s.httpheaderline[s.httpheaderlineptr - 1] = 0;
                }
            }
            else
```

```
            {
                uip_abort();
                WEBCLIENT_ABORTED();
                return 0;
            }

            /* We're done parsing the status line, so we reset the pointer
               and start parsing the HTTP headers.*/
            s.httpheaderlineptr = 0;
            s.state = WEBCLIENT_STATE_HEADERS;
            break;
        }
        else
        {
            ++s.httpheaderlineptr;
        }
    }

    return len;
}

/*-----------------------------------------------------------------
* void newdata()
* parse new data, check if the data was received ok and
* close the connection
-----------------------------------------------------------------*/
static void newdata(void)
{
    u16_t xdata len;

    len = uip_datalen();

    if(s.state == WEBCLIENT_STATE_STATUSLINE)
    {
        /* parse http status line and check if the data was received OK */
        len = parse_statusline(len);
        /* if httpflag received was http OK, close connection */
        if (s.httpflag == HTTPFLAG_OK)
        {
            s.state = WEBCLIENT_STATE_CLOSE;
        }
    }
}

/*---------------------------------------------------------------------------
/ void webclient_appcall(void)
/ Main function to http post application. It manages all the steps during the http post
/ open connection We use the uip_ test functions to deduce why we were called. If
/ uip_connected() is non-zero, we were called because a remote host has connected to us. If
/ uip_newdata() is non-zero, we were called because the remote host has sent us new data, and  / if uip_acked() is non-zero, the
```

```
previously sent
/ to it.
---------------------------------------------------------------------------------------*/
void webclient_appcall(void)
{
    char xdata file[]="/ecg.php";
    char xdata host[]="192.168.1.50";

   /* at the first time we are not connected to the server. This is done when the
      TCP/IP stack connect to the server */
    if(uip_connected())
    {
        printf("Connection accepted by the server\n");
        s.timer = 0;
        s.state = WEBCLIENT_STATE_STATUSLINE;
        webclient_connected();
        return;
    }

    /* if Webclient state is close, abort connection and continue to capture data through */
    /* Timer 0 IRQ */
    if(s.state == WEBCLIENT_STATE_CLOSE)
    {
        WEBCLIENT_CLOSED(); //enable TMR0 interrupt to continue acquire data
        uip_abort();
        uip_flags = UIP_CLOSE;
        s.state = WEBCLIENT_STATE_STATUSLINE;
        return;
    }

    /* If uIP tcp/IP aborted connection, continue acquiring data */
    if(uip_aborted())
    {
        WEBCLIENT_ABORTED();  //enable TMR0 interrupt to continue acquire data
        uip_flags = UIP_CLOSE;
        s.state = WEBCLIENT_STATE_STATUSLINE;
    }

    /* if uIP timeout, so try again */
    if(uip_timedout())
    {
        WEBCLIENT_TIMEDOUT();
        s.timer = 0;
        uip_flags = UIP_CLOSE;
        s.state = WEBCLIENT_STATE_STATUSLINE;
    }

    if(uip_acked())
    {
        s.timer = 0;
```

```
        acked();
    }

    /* if uIP request for newdata proceed with it */
    if(uip_newdata())
    {
        s.timer = 0;
        newdata();   //decode http protocol and call ecg function to send data
    }

    if(uip_poll())
    {
        ++s.timer;
        if(s.timer == WEBCLIENT_TIMEOUT)
        {
            WEBCLIENT_TIMEDOUT();
            uip_abort();
            uip_flags = UIP_CLOSE;
            s.state = WEBCLIENT_STATE_STATUSLINE;
            return;
        }
        senddata();
    }

/* if connection is closed and ecg_send_button pressed */
    if(uip_closed() && ecg_send_button)
    {
        printf("Send HTTP POST request\n");
        s.port = 80;
        webclient_post(host, s.port, file);
        ecg_send_button = 0;
    }
    else
    {
        if(uip_closed())
        {
            printf("Enter in power down mode through application\n");

            /* Enable switch IRQ */
            ENABLE_INT0_IRQ();

            /* reset WDE bit to clear WDT*/
            DISABLE_GLOBAL_IRQ();
            ENABLE_WDT_WRITE();
            WDE = 0;
            ENABLE_GLOBAL_IRQ();

            /* Change the state to Power Down */
            ecg_state = ECG_STATE_DPM_PWRDWN;;
        }
```

```
    }
}
```

B.5 Código da Aplicação médica

```
/*********************************************************************
* Filename: ecg_app.h
* Description: ecg_app.c header file
* Developed by: Marco Carvalho
* Created on: 30/03/05
*********************************************************************/

#include "uip.h"
#include "uipopt.h"
#include "webclient.h"

/*********************************************************************
*                            Private defines.
*********************************************************************/
// Change ECG_CPU_XTAL to match hardware
#define ECG_CPU_XTAL       12580000             // 8051 crystal freq in Hz

// Crystal/ Baud rate combinations marked * are well supported with 100.0%
// timing accuracy.
/*                             Xtal Frequency (MHz)

                    3.6864     11.0592     18.4320     22.1184
                  ÚÄÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÄ¿
             300  ³      *³      *³      *³      *³      *³
                  ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
             600  ³      *³      *³      *³      *³      *³
                  ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
            1200  ³      *³      *³      *³      *³      *³
                  ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
  BAUD      2400  ³      *³      *³      *³      *³      *³
  RATE            ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
            4800  ³      *³      *³      *³      *³      *³
                  ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
            9600  ³      *³      *³      *³      *³      *³
                  ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
           14400  ³      _³      *³      _³      *³      *³
                  ÃÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄÅÄÄÄÄÄÄ´
           19200  ³      _³      _³      _³      *³      *³
                  ÀÄÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÄÙ            */


// Low Baud rate (SCON = 0), X1 CPU mode timing parameters.
// Adjust XTAL and UART_BPS to suit your application and hardware.
```

```
#define ECG_XTAL             ECG_CPU_XTAL      // Crystal freq in Hz
#define ECG_UART_BAUD            4800               // Tranceiver baud rate
#define ECG_T1_CLOCK         ECG_XTAL / 12          // Timer 1 mode 2 clock rate
#define ECG_T1_RELOAD            256 - ((ECG_T1_CLOCK / 32) / ECG_UART_BAUD)

#define ECG_CPU_CLOCK      ECG_CPU_XTAL / 12    // 8051 clock rate (X1 mode)

// Delay routine timing parameters
#define ECG_DELAY_CONST    9.114584e-5          // Delay routine constant
#define ECG_DELAY_MULTPLR  (unsigned char)(ECG_DELAY_CONST * ECG_CPU_CLOCK)

// X1 CPU mode timing parameters network mode
#define ECG_T0_CLOCK             ECG_CPU_XTAL / 12 // Timer 0 mode 1 clock rate
#define ECG_T0_INT_RATE_NETWORK  24                // Timer 0 intrupt rate (Hz)
#define ECG_T0_RELOAD_NETWORK    65536 - (ECG_T0_CLOCK / ECG_T0_INT_RATE_NETWORK)

// X1 CPU mode timing parameters memory mode
#define ECG_T0_CLOCK             ECG_CPU_XTAL / 12 // Timer 0 mode 1 clock rate
#define ECG_T0_INT_RATE_MEMORY   1000              // Timer 0 intrupt rate (Hz)
#define ECG_T0_RELOAD_MEMORY     65536 - (ECG_T0_CLOCK / ECG_T0_INT_RATE_MEMORY)

#define ENABLE_TMR0_IRQ() ET0 = 1;
#define DISABLE_TMR0_IRQ() ET0 = 0;

#define ENABLE_INT0_IRQ() EX0 = 1;
#define DISABLE_INT0_IRQ() EX0 = 0;

/* IRQ enable/disable macros */
#define ENABLE_GLOBAL_IRQ() EA = 1;
#define DISABLE_GLOBAL_IRQ() EA = 0;

/* Watch dog timer */
#define ENABLE_WDT() WDE = 1;
#define ENABLE_WDT_WRITE() WDWR = 1;

/* GREEN LED */
#define GREEN_ON() LED2=0 //verde ( 0=aceso )
#define GREEN_OFF() LED2=1 //verde ( 1=apagado )
#define GREEN_LED LED2
#define GREEN_TOGGLE() LED2=!LED2

#define ERASE_ALL() ECON = 0x06;

#define ECG_STATE_DPM_BUSY   0
#define ECG_STATE_DPM_IDLE   1
#define ECG_STATE_DPM_PWRDWN 2

#define CLEAR_TIC_BIT() TIMECON = TIMECON & 0xFB;
#define CLEAR_TIMER_INTERVAL_ENABLE_BIT() TIMECON = TIMECON & 0xFD;
```

```
/*********************************************************************
*                       external variables
*********************************************************************/
extern unsigned char xdata ecg_state;
extern unsigned int xdata tick_count;
extern unsigned int xdata ecg_index;
extern unsigned long xdata lba_address;

/*********************************************************************
*                       function prototypes
*********************************************************************/
char ecg_read(void);
void ecg_prepare_data(void);


/*********************************************************************
* Filename: ecg_app.c
* Description: This file implements the ecg medical application
* Developed by: Marco Carvalho
* Created on: 30/03/05
*********************************************************************/

#include "wifidev.h"
#include "ecg_app.h"

/***************************Global Variables ****************************/
unsigned int xdata tick_count;
unsigned int xdata samples = 0;

#define ECG_POST_COMMAND_LEN 130
u8_t xdata *ecg_apphdr = &uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + ECG_POST_COMMAND_LEN];
u8_t xdata *ecg_appdata = &uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + ECG_POST_COMMAND_LEN + 4];

unsigned char xdata ecg_state;

unsigned int xdata ecg_index = 0;

unsigned long xdata lba_address = 0;

// read adc data buffer.
// This buffer simulate 3 samples from ecg adc. In the future it will be change
// by a real ecg adc sample
unsigned char xdata read_adc[]={0, 34, 18};

static char xdata i_adc=0;

/*********************************************************************
/* ecg_read(void)
/* read an ecg sample
*********************************************************************/
```

```
char ecg_read(void)
{
    char xdata data_read;

    /* Implement circular buffer */
    if (cis_device == CISTPL_FUNCID_NETWORK)
    {
        if(ecg_index >= (UIP_BUFSIZE - UIP_TCPIP_HLEN + UIP_LLH_LEN + ECG_POST_COMMAND_LEN + 4))
        {
            ecg_index = 0;
        }
    }

    /* reset simulated ecg sample data buffer */
    if(i_adc == 3)
        i_adc = 0;

    /* read a simulated ecg sample */
    data_read = read_adc[i_adc++];

    return data_read;
}

/*********************************************************************
*                          ecg_timer0_isr()
*
* This function deal with timer 0 irq. It manages two modes of operation
* according to cis_device variable.
*
* If the CF card is a network card, the timer 0 overflow is setup to
* 24Hz frequency and it calls ecg_read function to collect ecg data.
* In network mode also the machine state goes to busy or power down
* state depending on some variables states.
*
* If the CF is a memory card, the sample frequency is increased to
* 1000Hz and it continues to store ecg data.
*********************************************************************/
static void ecg_timer0_isr(void) interrupt 1 using 2
{

    /*Reload timer/ counter 0 according to CF mode */
    if (cis_device == CISTPL_FUNCID_NETWORK)
    {
        TH0 = ECG_T0_RELOAD_NETWORK >> 8; // 24Hz
        TL0 = ECG_T0_RELOAD_NETWORK;

        // Increment 24ths of a second counter.
        tick_count++;

        /* Read ecg data to get the analog value from adc */
```

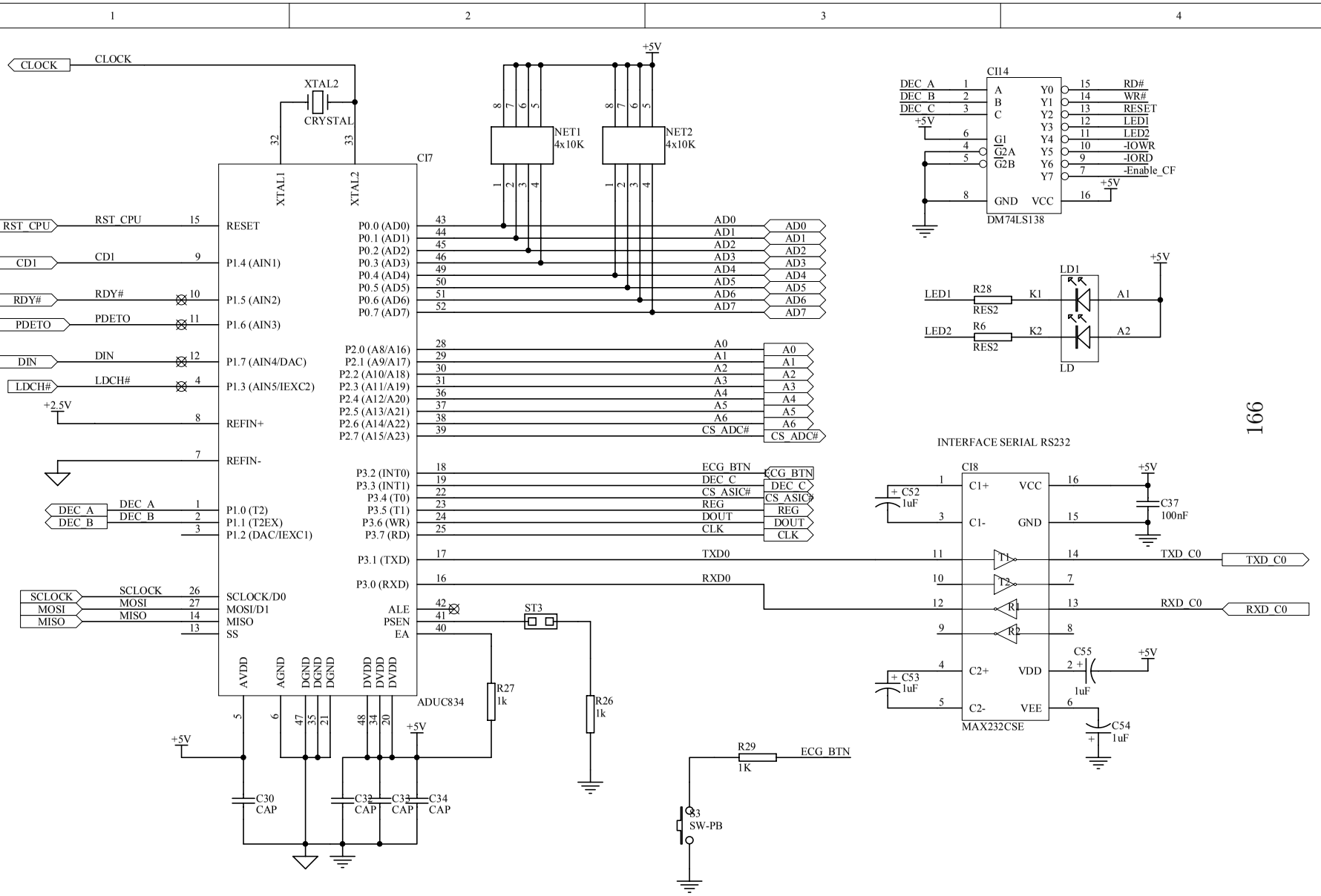
```
        ecg_appdata[ecg_index++] = ecg_read();

        /* Increment the number of samples */
        samples++;

        /* if collect time actived so change to Power down state */
        if(tick_count == 600)
        {
            /* if reset by Watch dog try to send data again*/
            if(WDS)
            {
                ecg_state = ECG_STATE_DPM_BUSY;
                CLEAR_TIMER_INTERVAL_ENABLE_BIT(); // TIEN = TIMECON.1 = 0
                ecg_send_button = 1;
                DISABLE_TMR0_IRQ(); // stop acquiring new data
                tick_count = 0;

                /* set WDE bit to clear WDT*/
                DISABLE_GLOBAL_IRQ();
                ENABLE_WDT_WRITE();
                WDS = 0;
                ENABLE_GLOBAL_IRQ();
            }
            /* enter in power down mode */
            else
            {
                ecg_state = ECG_STATE_DPM_PWRDWN;
            }
        }
    }
    else // if(cis_device == CISTPL_FUNCID_FIXED)
    {
        TH0 = ECG_T0_RELOAD_MEMORY >> 8;  //1000Hz
        TL0 = ECG_T0_RELOAD_MEMORY;

        uip_buf[ecg_index++] = ecg_read();
    }

    return;
}

/*********************************************************************
*                         ecg_tick_isr()
* this interrupt service routine deals with the power down wake-up.
* It clear the tic bit, change the ecg state machine to the idle state
* and reset the tick_count variable
*********************************************************************/
static void ecg_tick_isr(void) interrupt 10 using 3
{
    CLEAR_TIC_BIT(); //TII = TIMECON.2 = 0
```

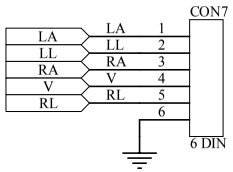
```
    CLEAR_TIMER_INTERVAL_ENABLE_BIT(); // TIEN = TIMECON.1 = 0
    ecg_state = ECG_STATE_DPM_IDLE;
    tick_count = 0; //reset tick counter
    return;
}

/*********************************************************************
*                          ecg_int0_isr()
* This interrupt service routine controls the int0 irq associated to
* the ecg emergency switch. it change the ecg state machine to busy, set
* a variable associate with the irq, stop acquiring new data disabling
* time 0 irq and clear the tick count variable
*********************************************************************/
static void ecg_int0_isr(void) interrupt 0 using 1
{
    /* Wait PLL to lock */
    while(!(PLLCON && 0x40));

    ecg_state = ECG_STATE_DPM_BUSY;
    CLEAR_TIMER_INTERVAL_ENABLE_BIT(); // TIEN = TIMECON.1 = 0
    ecg_send_button = 1;
    DISABLE_TMR0_IRQ(); // stop acquiring new data
    tick_count = 0;

    return;
}

/*********************************************************************
* ecg_prepare_data()
* prepare data to send. After sign the uIP TCP/IP Stack with newdata
* to send
*********************************************************************/
void ecg_prepare_data(void)
{
    /* prepare app header */
    /* get the timestamp */
    ecg_apphdr[0] = HOUR;
    ecg_apphdr[1] = MIN;
    ecg_apphdr[2] = SEC;
    /* set the sample frequency */
    ecg_apphdr[3] = 24;

    /* setup the app len */
    uip_len = samples+4;

    /* reset samples number */
    samples = 0;

    return;
}
```

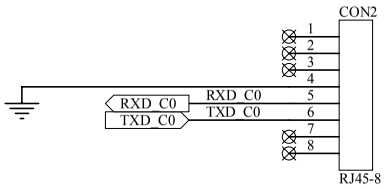
Apêndice C

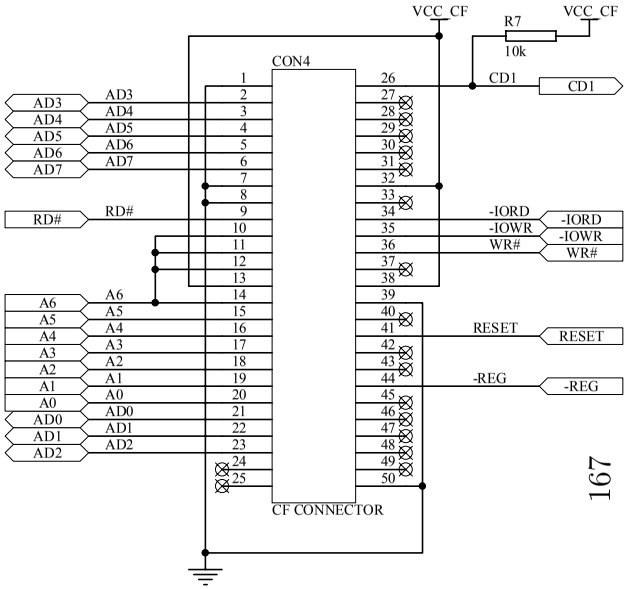
Circuito do Monitor Cardíaco modificado para suportar Comunicação sem fio e circuito ECG

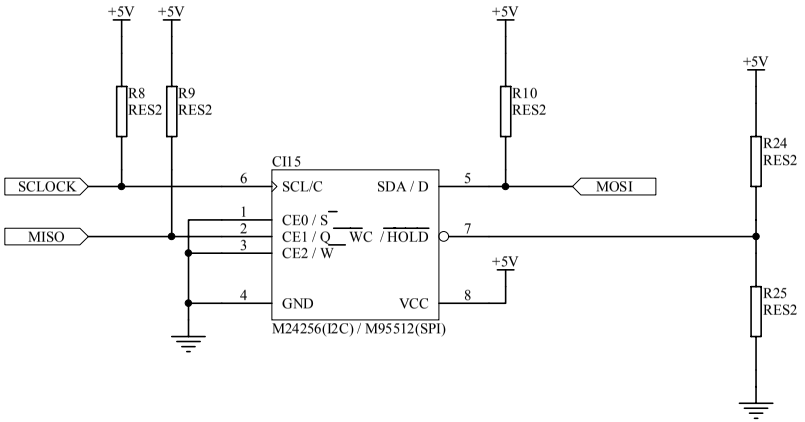
- Figura c-1 - Circuito da CPU
- Figura c-2 - Circuito dos Conectores e Memória Serial
- Figura c-3 - Circuito do *ASIC*
- Figura c-4 - Circuito do ADC
- Figura c-5 - Circuito da Fonte


CLOCK
XTAL2
CRYSTAL
CI7
XTAL1
RST_CPU
CD1
RDY#
PDETO
DIN
LDCH#
+2.5V
REFIN+
REFIN-
DEC_A
DEC_B
SCLOCK
MOSI
MISO
SS
RESET
P1.4 (AIN1)
P1.5 (AIN2)
P1.6 (AIN3)
P1.7 (AIN4/DAC)
P1.3 (AIN5/IEXC2)
P1.0 (T2)
P1.1 (T2EX)
P1.2 (DAC/IEXC1)
SCLOCK/D0
MOSI/D1
P0.0 (AD0)
P0.1 (AD1)
P0.2 (AD2)
P0.3 (AD3)
P0.4 (AD4)
P0.5 (AD5)
P0.6 (AD6)
P0.7 (AD7)
P2.0 (A8/A16)
P2.1 (A9/A17)
P2.2 (A10/A18)
P2.3 (A11/A19)
P2.4 (A12/A20)
P2.5 (A13/A21)
P2.6 (A14/A22)
P2.7 (A15/A23)
P3.2 (INT0)
P3.3 (INT1)
P3.4 (T0)
P3.5 (T1)
P3.6 (WR)
P3.7 (RD)
P3.1 (TXD)
P3.0 (RXD)
ALE
PSEN
EA
AVDD
AGND
DGND
DVDD
ADUC834
NET1
4x10K
NET2
4x10K
+5V
AD0
AD1
AD2
AD3
AD4
AD5
AD6
AD7
A0
A1
A2
A3
A4
A5
A6
CS_ADC#
ECG_BTN
DEC_C
CS_ASIC#
REG
DOUT
CLK
TXD0
RXD0
ST3
R27
1k
R26
1k
R29
1K
S3
SW-PB
C30
CAP
C32
CAP
C33
CAP
C34
CAP
CI14
DM74LS138
DEC_A
DEC_B
DEC_C
G1
G2A
G2B
GND
VCC
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
RD#
WR#
RESET
LED1
LED2
-IOWR
-IORD
-Enable_CF
LD1
R28
R6
RES2
K1
K2
A1
A2
LD
INTERFACE SERIAL RS232
CI8
MAX232CSE
C1+
C1-
C2+
C2-
VCC
GND
VDD
VEE
T1
T2
R1
R2
C52
1uF
C53
1uF
C37
100nF
C55
1uF
C54
1uF
TXD_C0
RXD_C0
166





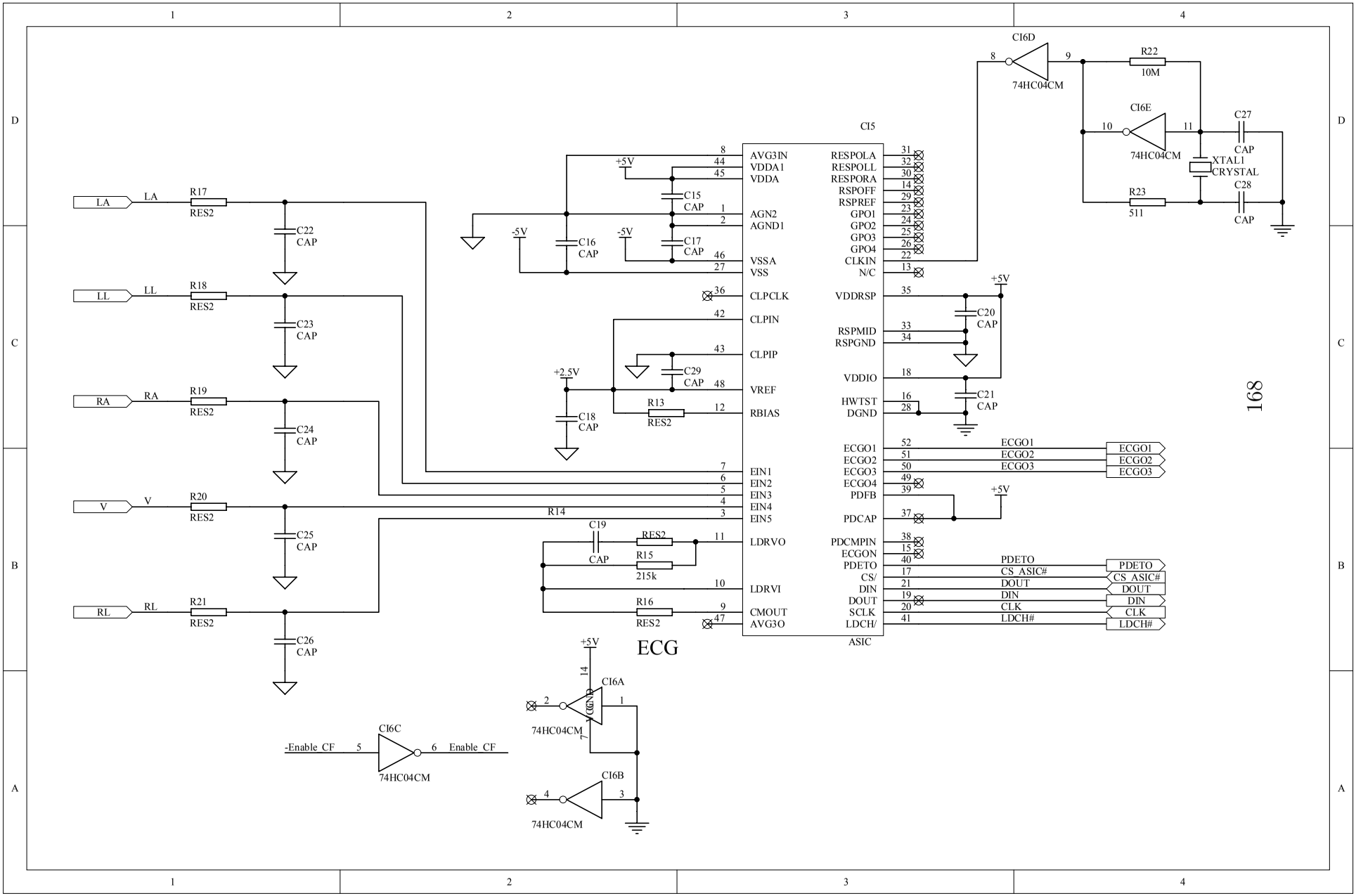


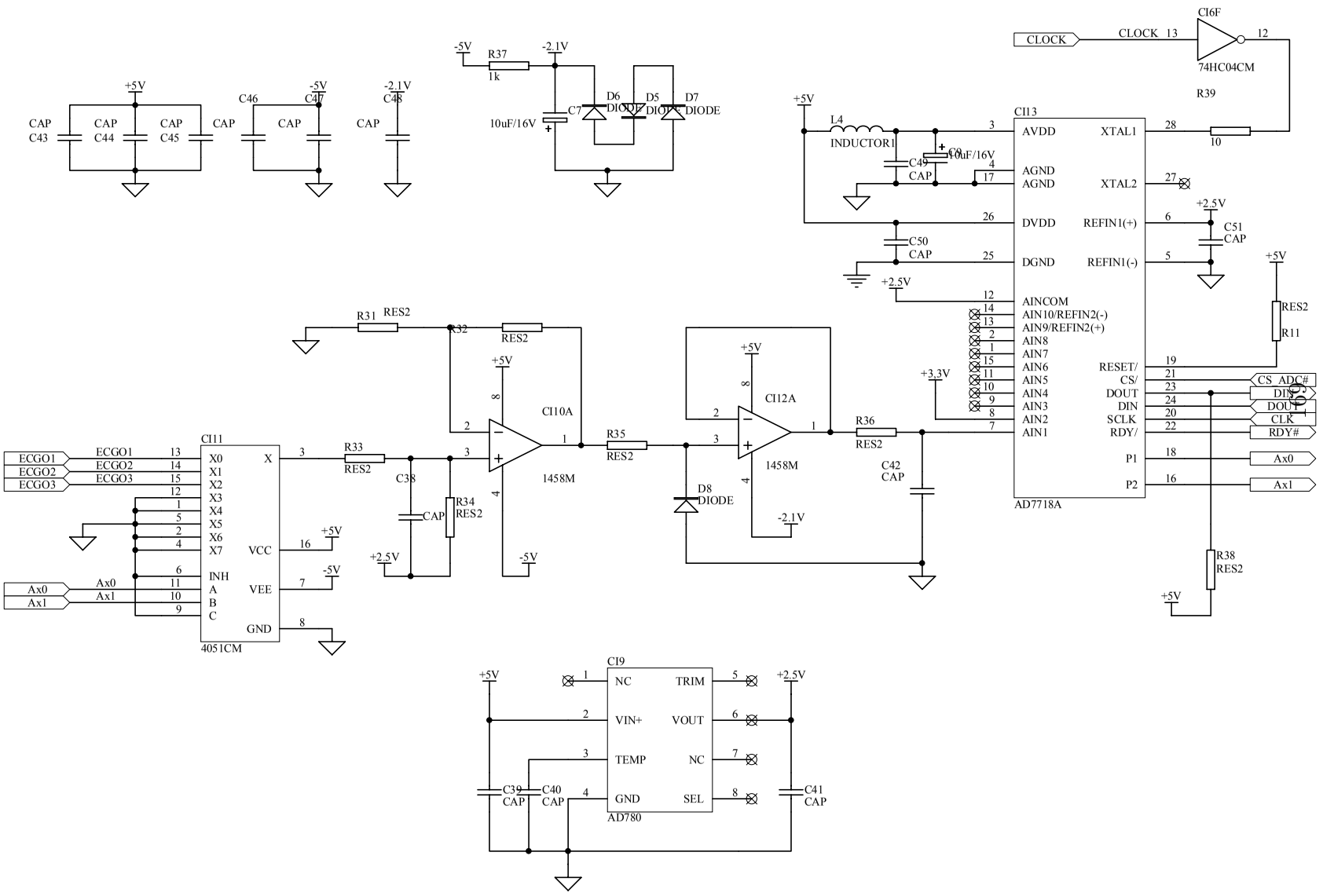


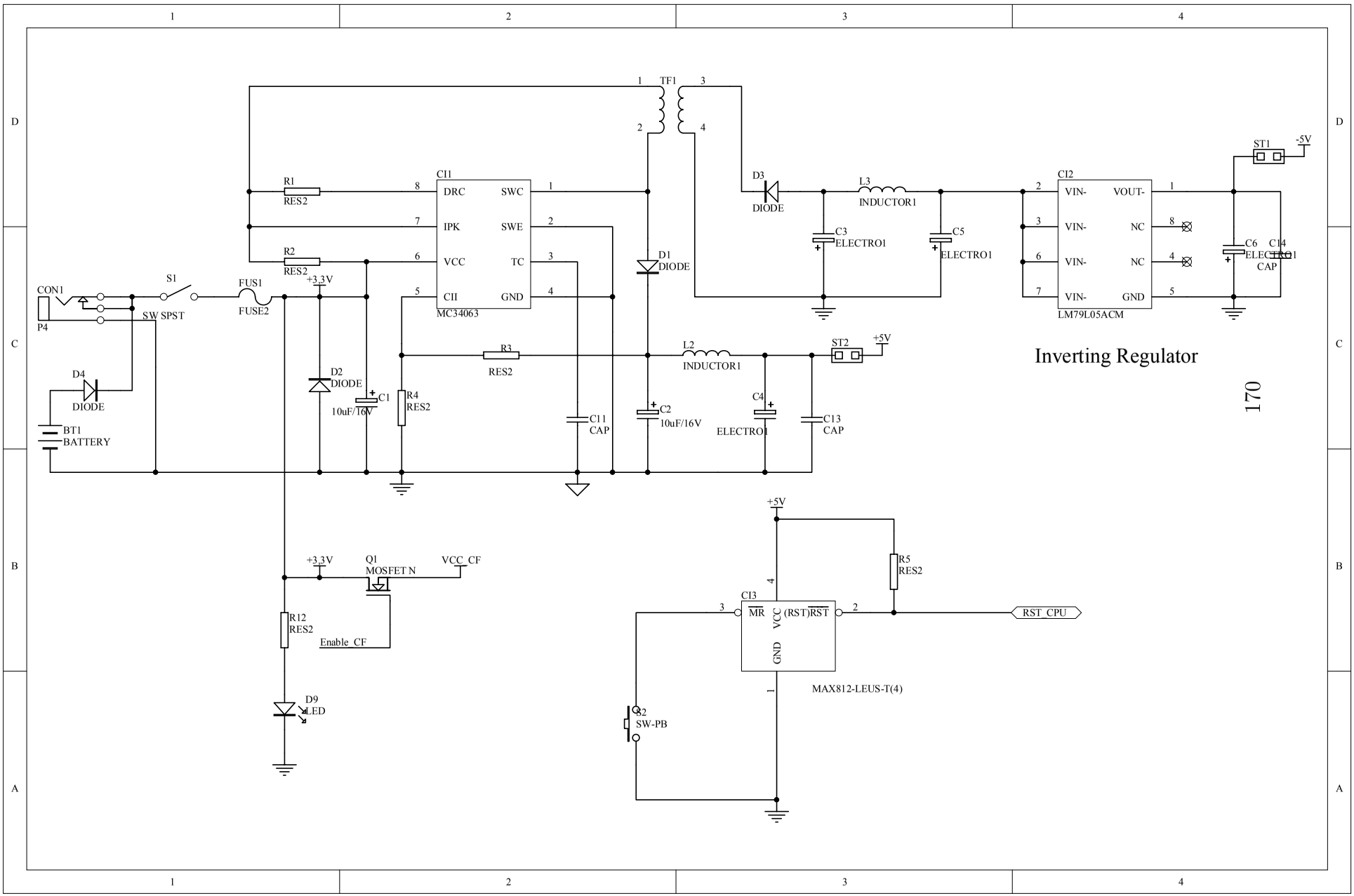
Instrução para montagem da Memória Serial:

Memória I2C: Montar resistor R25 e não montar resistor R24

Memória SPI: Montar resitor R24 e não montar resistor R25


ECG
CI5
ASIC
AVG3IN
VDDA1
VDDA
AGN2
AGND1
VSSA
VSS
CLPCLK
CLPIN
CLPIP
VREF
RBIAS
EIN1
EIN2
EIN3
EIN4
EIN5
LDRVO
LDRVI
CMOUT
AVG3O
RESPOLA
RESPOLL
RESPORA
RSPOFF
RSPREF
GPO1
GPO2
GPO3
GPO4
CLKIN
N/C
VDDRSP
RSPMID
RSPGND
VDDIO
HWTST
DGND
ECGO1
ECGO2
ECGO3
ECGO4
PDFB
PDCAP
PDCMPIN
ECGON
PDETO
CS/
DIN
DOUT
SCLK
LDCH/
LA
LL
RA
V
RL
R17
R18
R19
R20
R21
RES2
C22
C23
C24
C25
C26
CAP
C15
C16
C17
C18
C19
C20
C21
C27
C28
C29
R13
R14
R15
215k
R16
R22
10M
R23
511
XTAL1
CRYSTAL
CI6A
CI6B
CI6C
CI6D
CI6E
74HC04CM
+5V
-5V
+2.5V
-Enable_CF
Enable_CF
CS_ASIC#
DOUT
DIN
CLK
LDCH#
168


CAP C43
CAP C44
CAP C45
CAP
C46
C47
CAP
CAP
C48
-5V
R37
1k
-2.1V
C7
10uF/16V
D6 DIODE
D5 DIODE
D7 DIODE
CI6F
CLOCK
CLOCK 13
12
74HC04CM
R39
10
+5V
L4
INDUCTOR1
C49 CAP
C9 10uF/16V
CI13
AVDD
XTAL1
28
AGND
AGND
XTAL2
27
DVDD
REFIN1(+)
6
+2.5V
C51 CAP
C50 CAP
DGND
REFIN1(-)
5
+2.5V
AINCOM
AIN10/REFIN2(-)
AIN9/REFIN2(+)
AIN8
AIN7
AIN6
AIN5
AIN4
AIN3
AIN2
AIN1
RESET/
CS/
DOUT
DIN
SCLK
RDY/
P1
P2
AD7718A
RES2
R11
CS_ADC#
DIN
DOUT
CLK
RDY#
Ax0
Ax1
R31 RES2
R32
RES2
CI10A
1458M
R33
RES2
C38
CAP
R34
RES2
R35
RES2
CI12A
1458M
R36
RES2
C42
CAP
D8
DIODE
+3.3V
CI11
ECGO1
ECGO2
ECGO3
X0
X1
X2
X3
X4
X5
X6
X7
INH
A
B
C
X
VCC
VEE
GND
4051CM
R38
RES2
CI9
NC
VIN+
TEMP
GND
TRIM
VOUT
NC
SEL
AD780
C39 CAP
C40 CAP
C41 CAP


TF1
CI1
MC34063
DRC
IPK
VCC
CII
SWC
SWE
TC
GND
R1
RES2
R2
RES2
D1
DIODE
D3
DIODE
L3
INDUCTOR1
C3
ELECTRO1
C5
ELECTRO1
CI2
LM79L05ACM
VIN-
VOUT-
NC
ST1
-5V
C6
C14
CAP
CON1
P4
S1
SW SPST
FUS1
FUSE2
+3.3V
D4
DIODE
BT1
BATTERY
D2
DIODE
C1
10uF/16V
R4
RES2
R3
RES2
L2
INDUCTOR1
ST2
+5V
C11
CAP
C2
10uF/16V
C4
ELECTRO1
C13
CAP
Inverting Regulator
170
+3.3V
Q1
MOSFET N
VCC_CF
R12
RES2
Enable_CF
D9
LED
+5V
R5
RES2
CI3
MAX812-LEUS-T(4)
MR
VCC
(RST)RST
GND
RST_CPU
S2
SW-PB