UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Uma pilha de protocolos Bluetooth adaptável à aplicação

Eduardo Afonso Billo

Florianópolis, fevereiro de 2003.

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Uma pilha de protocolos Bluetooth adaptável à aplicação

Trabalho de conclusão de curso de graduação apresentado à Universidade Federal de Santa Catarina para a obtenção do grau de Bacharel em Ciência da Computação.

Autor:
Eduardo Afonso Billo
Orientador:
Dr. Antônio Augusto M. Fröhlich

Banca examinadora:
Dr. Antônio Augusto M. Fröhlich
Dr. José Mazzucco Jr.
M.Sc. Fernando Barreto

Florianópolis, fevereiro de 2003.

TERMO DE APROVAÇÃO

Eduardo Afonso Billo

Uma pilha de protocolos Bluetooth adaptável à aplicação

Trabalho de conclusão de curso de graduação apresentado à Universidade Federal de Santa Catarina para a obtenção do grau de Bacharel em Ciência da Computação.

____
Dr. Antônio Augusto M. Fröhlich
Orientador

____
Dr. José Mazzucco Jr.
Co-orientador

____
M.Sc. Fernando Barreto
Banca examinadora

Florianópolis, fevereiro de 2003.

SUMÁRIO

LISTA DE ABREVIATURAS ..... **VI**

LISTA DE FIGURAS ..... **VII**

LISTA DE TABELAS ..... **VIII**

RESUMO ..... **IX**

ABSTRACT ..... **X**

1 INTRODUÇÃO **1**

1.1 VISÃO GERAL DA APRESENTAÇÃO 2

2 FUNDAMENTOS BLUETOOTH **3**

2.1 TOPOLOGIAS DE REDES 3
2.1.1 Restrições de uma Scatternet *4*
2.2 CONEXÕES SÍNCRONAS E ASSÍNCRONAS 5
2.3 A PILHA DE PROTOCOLOS BLUETOOTH 6
2.4 UMA APLICAÇÃO BLUETOOTH 7
2.5 ESTADOS PARA ESTABELECIMENTO DE CONEXÕES 9
2.5.1 StandBy *10*
2.5.2 Inquiry *10*
2.5.3 Inquiry Scan *10*
2.5.4 Inquiry Response *10*
2.5.5 Demais estados *11*
2.6 MODOS DE OPERAÇÃO 11
2.6.1 Active Mode *11*
2.6.2 Sniff Mode *11*
2.6.3 Hold Mode *12*
2.6.4 Park Mode *12*

3 HARDWARE BLUETOOTH **13**

3.1 CONFIGURAÇÕES BÁSICAS 13
3.1.1 PC como HOST – Módulo Bluetooth externo *13*
3.1.2 Microcontrolador como HOST – Módulo Bluetooth externo *14*
3.1.3 Aplicação Integrada ao módulo *15*
3.1.4 Aplicação Integrada a um microprocessador *16*
3.2 HARDWARE USADO 16
3.3 ESPECIFICAÇÕES DE HARDWARE 17
3.3.1 Especificação de Rádio *18*
3.3.1.1 Banda de freqüência 18
3.3.1.2 Modulação 19
3.3.2 Especificação do link físico *22*
3.3.2.1 Espalhamento do sinal no espectro de freqüência 22

4 PILHA DE PROTOCOLOS BLUETOOTH ADAPTÁVEL À APLICAÇÃO **24**

4.1 MOTIVAÇÃO 24
4.2 PILHA DE PROTOCOLOS ADAPTADA À APLICAÇÃO 25
4.2.1 Camada Básica de Comunicação Bluetooth 27
4.2.1.1 HCI 31
4.2.1.2 Suporte a USB 37
4.2.2 Módulos de Adaptação 39
4.2.2.1 Módulo para controle centralizado 39
4.3 EXEMPLOS DE APLICAÇÕES 39
4.3.1 Vigilância Centralizada em Grandes Edificações 40
4.3.2 Controle Centralizado de Aparelhos Elétricos em Grandes Edificações 41
5 CONCLUSÃO 43
6 REFERÊNCIAS BIBLIOGRÁFICAS 45
APÊNDICES 47
APÊNDICE 1 – ARTIGO 48
ANEXOS 54
ANEXO 1 – ESPECIFICAÇÃO DO HARDWARE USADO 55
ANEXO 2 – CÓDIGO HCI 57
ANEXO 3 – CÓDIGO DO MÓDULO PARA CONTROLE CENTRALIZADO 68
ANEXO 4 – CÓDIGO DA APLICAÇÃO (NO COMPUTADOR CENTRAL) 71

LISTA DE ABREVIATURAS

ACL – Asynchronous Connection-Less
AM – Amplitude Modulation
DSSS- Direct Sequence Spread Spectrum
FHSS- Frequency Hopping Spread Spectrum
FM – Frequency Modulation
FSK – Frequency-Shift Keying
HCI – Host Controller Interface
ISM - Industrial Scientific Medicine
L2CAP – Logical Link Control and Adaptation Protocol
OHCI – Open Host Controller Interface
PC – Personal Computer
PDA – personal digital assistant
PIN – Personal Identification Number
PM – Phase Modulation
POP – Post Office Protocol
PPP – Point-to-Point Protocol
SCL – Synchronous Connection-Oriented
SDP – Service Discovery Protocol
SIG – Special Interest Group
TCP/IP – Transfer Control Protocol / Internet Protocol
TCS – Telephony Control protocol Specification
UART – Universal Asynchronous Receiver/Transmitter
UHCI – Universal Host Controller Interface
USB – Universal Serial Bus

LISTA DE FIGURAS

FIGURA 2.1: TOPOLOGIAS DE REDES BLUETOOTH 4
FIGURA 2.2: MULTIPLEXAÇÃO NO TEMPO 5
FIGURA 2.3: PILHA DE PROTOCOLOS BLUETOOTH 6
FIGURA 2.4: DIAGRAMA DE ESTADOS PARA ESTABELECIMENTO DE CONEXÕES 9
FIGURA 3.1: CONFIGURAÇÃO COM PC COMO HOST 13
FIGURA 3.2: CONFIGURAÇÃO COM MICROCONTROLADOR COMO HOST 14
FIGURA 3.3: CONFIGURAÇÃO COM APLICAÇÃO INTEGRADA AO MÓDULO 15
FIGURA 3.4: CONFIGURAÇÃO COM APLICAÇÃO INTEGRADA A UM MICROPROCESSADOR 16
FIGURA 3.5: FOTO DO MÓDULO BLUETOOTH USADO 17
FIGURA 3.6: BLOCOS FUNCIONAIS DE UM MÓDULO BLUETOOTH 17
FIGURA 3.7: MODULAÇÃO EM AMPLITUDE 19
FIGURA 3.8: MODULAÇÃO EM FREQÜÊNCIA 20
FIGURA 3.9: MODULAÇÃO EM FASE 21
FIGURA 3.10: COMPORTAMENTO FHSS NO TEMPO 22
FIGURA 4.1: PILHA DE PROTOCOLOS ADAPTADA À APLICAÇÃO 25
FIGURA 4.2: PILHA DE PROTOCOLOS COMPLETA SUGERIDA PELA SIG 26
FIGURA 4.3: CAMADA BÁSICA DE COMUNICAÇÃO SUGERIDA PELA SIG 28
FIGURA 4.4: DIAGRAMA DE FAMÍLIAS DE UM SISTEMA DE COMUNICAÇÃO 28
FIGURA 4.5: DIAGRAMA DE CLASSES DE UM SISTEMA DE COMUNICAÇÃO 29
FIGURA 4.6: FORMATO DO FRAME DE COMANDO 32
FIGURA 4.7: FORMATO DO FRAME DE EVENTO 34
FIGURA 4.8: DIAGRAMA DE INICIALIZAÇÃO DE UM DISPOSITIVO BLUETOOTH 35
FIGURA 4.9: FORMATO DO FRAME DE TRANSMISSÃO 36
FIGURA 4.10: DIAGRAMA DE ESTADOS DE UMA APLICAÇÃO BLUETOOTH 41

LISTA DE TABELAS

TABELA 1: ROTINAS X FUNÇÕES 32

RESUMO

Este trabalho apresenta a proposta de uma pilha de protocolos a ser usada em um sistema de comunicação Bluetooth.

Como Bluetooth é uma tecnologia desenvolvida essencialmente para ser usada em sistemas embutidos onde a capacidade de memória, processamento, etc., são bastante limitados, surge a necessidade de desenvolver uma pilha de protocolos onde seu tamanho seja sensivelmente diminuído, sem que se perda em funcionalidade e eficiência.

Para oferecer uma pilha de protocolos pequena sem nenhuma perda, surge a idéia de uma pilha de protocolos adaptável à aplicação. Basicamente, a pilha de protocolos será moldada de forma a ter APENAS aquelas funcionalidades exigidas pela aplicação. Não é toda a aplicação, por exemplo, que necessita de comunicação segura, com dados encriptados. Esta seria então, uma das funcionalidades não fornecidas pela pilha de protocolos Bluetooth.

A adaptação da pilha de protocolos à aplicação é conseguida através do uso de duas técnicas: metaprogramação estática e uso de módulos. Metaprogramação estática é usada para oferecer uma camada mínima de comunicação que contenha apenas as funcionalidades exigidas pela aplicação. Já os módulos são usados para adicionar funcionalidades à pilha de protocolos, tornando menos árduo a programação de uma aplicação Bluetooth.

Além de se adaptar à aplicação em questão, a pilha de protocolos é projetada de modo a ser facilmente integrada a um sistema genérico de comunicação, independente da tecnologia de comunicação empregada (Ethernet, Myrinet, etc.).

Tanto o uso de metaprogramação estática na camada mínima de comunicação, quanto à integração do sistema Bluetooth a um sistema de comunicação genérico seguem a metodologia sugerida no sistema operacional EPOS [6].

Palavras-chave: Bluetooth, comunicação sem fio, metaprogramação estática, módulos.

ABSTRACT

This document proposes a protocol stack to be used by a Bluetooth Communication System.

Since Bluetooth is a technology commonly used by embedded systems, where the memory size, processing capacity, etc., are very limited, it comes up the need of developing a protocol stack with a smaller size, but with no functionality nor efficiency losses.

To design a smaller protocol stack without any losses, it is proposed a protocol stack adaptable to the application. Basically, the protocol stack will be tailored in a way that ONLY those functionalities waited by the application will be offered. For example, not all the applications need a secure communication link, with encrypted data. Then, that would be one of the functionalities to be excluded from the protocol stack.

The adaptation of the protocol stack to the application is possible thanks to two techniques: static metaprogramming and the use of modules. Static metaprogramming is used to offer a minimum communication layer that contains only those functionalities waited by the application. In the other hand, modules are used to add functionalities to the protocol stack, making the task of programming a Bluetooth application, much easier.

Besides the adaptation issue, the protocol stack is also designed in such a way that it can be easily integrated to a generic communication system, regardless of the adopted technology (Ethernet, Myrinet, etc.).

Both the use of static metaprogramming in the minimum communication layer as well as the integration with a generic communication system follows the methodology proposed by the operational system EPOS [6].

Keywords: Bluetooth, wireless, static metaprogramming, modules.

1 INTRODUÇÃO

Atualmente, existem várias formas de se estabelecer comunicação entre computadores. Através das consagradas redes Ethernet, redes de fibras ópticas, redes sem fio, dentre outras formas. Porém, até pouco tempo atrás, ainda não existia um sistema de comunicação barato e eficaz que pudesse ser usado não só por computadores, mas por dispositivos eletrônicos em geral, para se comunicarem a uma pequena distância. Imagine-se, por exemplo, o caso de um computador que controle todos os aparelhos eletro-eletrônicos de uma casa. Como controlá-los por meio de um computador?

Desta carência do mercado, surgiu uma tecnologia conhecida como Bluetooth. É uma tecnologia bastante barata, que oferece um link de pequena distância, para transmissões na ordem de dezenas a centenas de kbytes, ideal para sistemas embutidos.

Os sistemas Bluetooth comercialmente distribuídos apresentam um grave problema: os módulos de Software contidos nestes sistemas são muito grandes para serem portados em um sistema embutido, onde os recursos são muito limitados. Daí surge a necessidade de modificar tais módulos de maneira que possam ser mais facilmente utilizados nestes sistemas. Este é justamente o tema deste trabalho: propor uma maneira de modificar o Software de um sistema Bluetooth, de maneira que ele torne-se menor sem, contudo, perder em funcionalidade e eficiência.

Isto tudo é conseguido desenvolvendo um Software adaptável à aplicação em questão, com o uso de conceitos de metaprogramação e desenvolvimento modular como será visto ao longo do trabalho.

1.1 Visão geral da apresentação

No capítulo dois será dado um embasamento sobre a tecnologia Bluetooth, mostrando seus principais conceitos. No capítulo três, será explicado brevemente como funciona um Hardware Bluetooth. Finalmente no capítulo quatro, mostrar-se-á como desenvolver o Software Bluetooth adaptável à aplicação.

2 FUNDAMENTOS BLUETOOTH

Bluetooth é uma tecnologia para comunicação de dispositivos eletrônicos (não apenas computadores) que surgiu inicialmente com intuito de substituir o cabeamento necessário para interconexão de tais dispositivos.

O padrão Bluetooth foi desenvolvido pelo SIG, um grupo formado por algumas empresas líderes mundiais nas telecomunicações, computação e indústrias de redes. É um padrão com três características essenciais: consumo de potência baixíssimo, baixo alcance, taxas de transmissão baixas. Essas características contrastam com o padrão IEEE 802.11, que tem consumo maior, maior alcance e taxas de transmissão superiores. Então, porque trocar uma tecnologia que oferece alcance e taxas de transmissão maiores pelo Bluetooth? O principal motivo é o custo. Um chip Bluetooth pode ser encontrado no mercado por menos de cinco dólares americanos. Outro motivo importante é o consumo de potência, que é menor. Além desses motivos, não são todas as aplicações que necessitam de um alcance e taxas de transmissão altas.

2.1 Topologias de Redes

O sistema Bluetooth provê conexões ponto-a-ponto (apenas dois dispositivos Bluetooth envolvidos), ou conexões ponto-multiponto. Nas conexões ponto-multiponto, o canal é compartilhado entre alguns dispositivos Bluetooth, formando uma **piconet.** Em uma piconet, um dos dispositivos Bluetooth funciona como *master* (mestre), enquanto os demais funcionam como *slaves* (escravos). O master controla o acesso dos dispositivos slaves, determina o clock responsável pela sincronização, dentre outras funções.

Múltiplas piconets com áreas sobrepostas formam uma **scatternet.** A seguir são mostrados alguns exemplos de topologias possíveis:

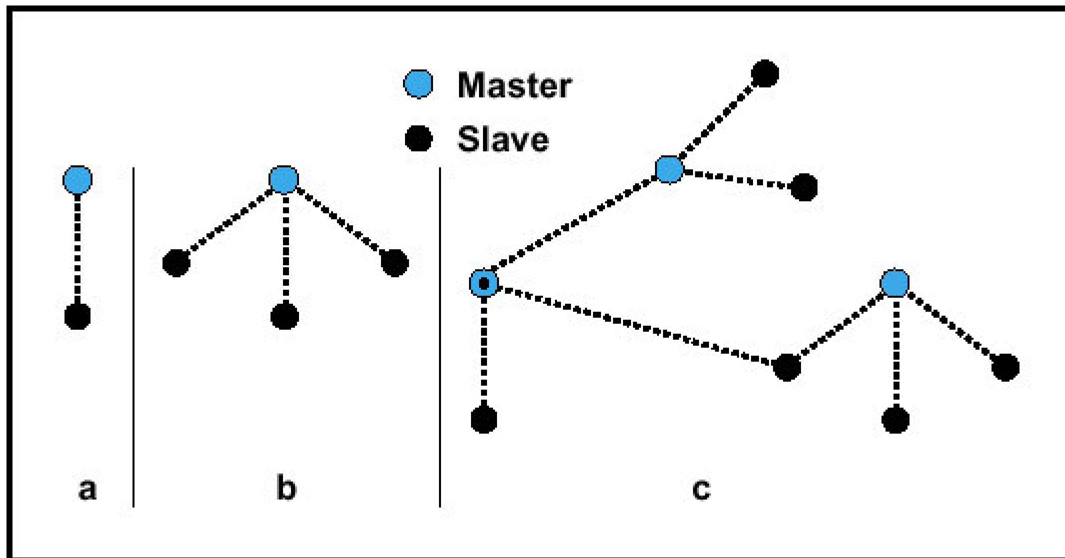


FIGURA 2.1: TOPOLOGIAS DE REDES BLUETOOTH
(Extraído de [10])

Em ***a,*** tem-se uma piconet com um único escravo. Em ***b,*** tem-se uma piconet com múltiplos escravos. Em ***c*,** tem-se uma possível configuração de uma scatternet.

2.1.1 Restrições de uma Scatternet

É condição necessária em um sistema Bluetooth, que cada piconet tenha apenas um master, porém, escravos podem participar de diferentes piconets (inclusive o master de uma piconet, pode ser slave de outra piconet. Isso é ilustrado no exemplo ***c,*** na figura 2.1). O compartilhamento do canal é possível graças a multiplexação no tempo, como mostrado na figura a seguir:

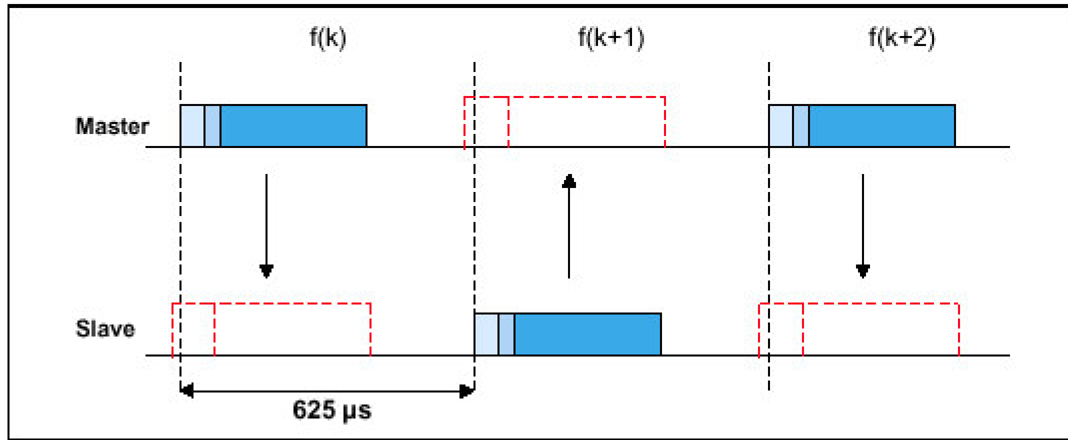


FIGURA 2.2: MULTIPLEXAÇÃO NO TEMPO

(Extraído de [10])

Como visto acima, o canal é dividido em slots de tempo. Cada slot tem a duração de 625μs. No slot f(k) o master transmite seus pacotes. Em f(k+1), o slave transmite seus pacotes e assim sucessivamente. Para uma Piconet com vários *slaves,* o mesmo raciocínio é aplicado.

2.2 Conexões Síncronas e Assíncronas

Em um sistema Bluetooth, deve ser possível estabelecer links físicos síncronos e assíncronos. O link físico síncrono é um link simétrico, ponto-a-ponto entre o dispositivo master e um slave específico. É ideal para dados contínuos, como por exemplo, a voz, pois slots são pré-reservados para cada dispositivo Bluetooth envolvido. O link síncrono é considerado conexão de chaveamento por circuito. Já link assíncrono, provê uma conexão assimétrica, ponto-multiponto e tira proveito dos slots não usados pelas conexões síncronas para transmissão dos dados. É considerada uma conexão de chaveamento por pacotes.

2.3 A pilha de protocolos Bluetooth

Para abordar o sistema Bluetooth em mais detalhes, nada melhor do que entender como o sistema foi especificado. O sistema Bluetooth foi especificado por meio de uma pilha de protocolos, como mostrado abaixo:

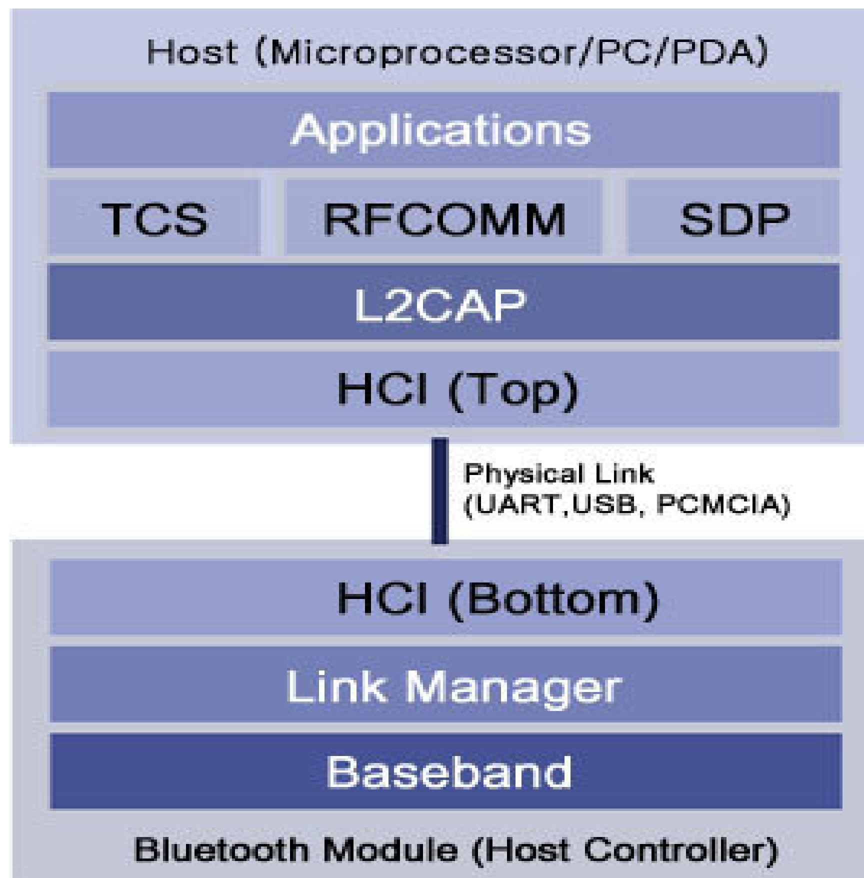


FIGURA 2.3: PILHA DE PROTOCOLOS BLUETOOTH
(Extraído de [8])

Esta pilha de protocolos carrega consigo toda a funcionalidade esperada de um sistema Bluetooth: transmissão via ondas de rádio, estabelecimento de links síncronos e assíncronos, suporte a criptografia, etc.

Esta pilha de protocolos será objeto de estudo ao longo deste documento. Porém, para se ter uma idéia inicial da suas funcionalidades, será mostrado a função de cada uma destas camadas em uma aplicação real.

2.4 Uma Aplicação Bluetooth

A aplicação tomada como exemplo, trata de um uso corriqueiro do sistema Bluetooth: uma pessoa no saguão de um hotel deseja checar seu e-mail a partir de um dispositivo portátil (PDA ou laptop, por exemplo) que tenha suporte a Bluetooth. Para isso, os seguintes passos seriam tomados:

Inquiry: O dispositivo realiza um processo conhecido como *inquiry*. Neste processo, ele "pergunta" quais os pontos de acesso que estão próximos. Todos os pontos de acesso respondem com seu endereço físico (cada módulo Bluetooth tem um endereço físico único no mundo, gravado no momento da fabricação). O dispositivo móvel escolhe então um destes pontos de acesso.

Paging: A seguir, é iniciado um processo conhecido como *paging*. Neste processo, o dispositivo móvel procura sincronizar-se com o ponto de acesso, em termos de *clock offset* e fase do salto em freqüência (esses dois parâmetros serão explicados na seção que aborda o Hardware Bluetooth), além de outras inicializações necessárias.

Estabelecimento de um link: Como se trata de uma aplicação que não demanda um fluxo contínuo de dados, é estabelecido um link assíncrono (ACL). A camada responsável por estabelecer tal link é a LMP (Link Manager Protocol).

Procura por Serviços: Depois de estabelecido um link físico, o dispositivo móvel tenta descobrir por meio do SDP (Service Discovery Protocol), quais os serviços disponíveis no ponto de acesso. No caso, verificará se é possível acessar e-mail a partir do ponto de acesso. Suponha-se que seja possível, pois caso contrário a aplicação não funcionaria.

L2CAP: Baseado na informação obtida pelo SDP, um canal L2CAP será criado para possibilitar a comunicação entre os dois dispositivos.

RFCOMM: Um canal RFCOMM é criado sobre o canal L2CAP. O canal RFCOMM emula uma interface serial. Desta forma, pode-se transmitir dados entre os dispositivos Bluetooth através de uma interface semelhante, por exemplo, a “/dev/ttyS1” do linux.

Segurança: Caso o ponto de acesso restrinja o acesso a um grupo específico de usuários, é realizado um processo de autenticação, onde o dispositivo móvel deve saber o PIN correto para acessar o serviço. Além disso, se os dispositivos estiverem operando no modo seguro, os dados serão encriptados.

PPP: Para acessar um e-mail, é usado o TCP/IP. No nível da aplicação do TCP/IP, tem-se o protocolo POP responsável pelo acesso à conta de e-mail. Este protocolo faz uso de uma conexão PPP. O PPP, geralmente é executado a partir de um link serial (um modem em uma conexão dial-up, por exemplo). No caso do sistema Bluetooth, o protocolo PPP rodará a partir do canal RFCOMM, que emula similarmente um link serial. A partir daí, os protocolos usuais da Internet podem ser executados normalmente e, neste caso, o e-mail pode ser acessado.

2.5 Estados para estabelecimento de conexões

Para melhor entender os estados existentes no estabelecimento de uma conexão, mencionados acima, vale a pena analisar o seguinte diagrama:

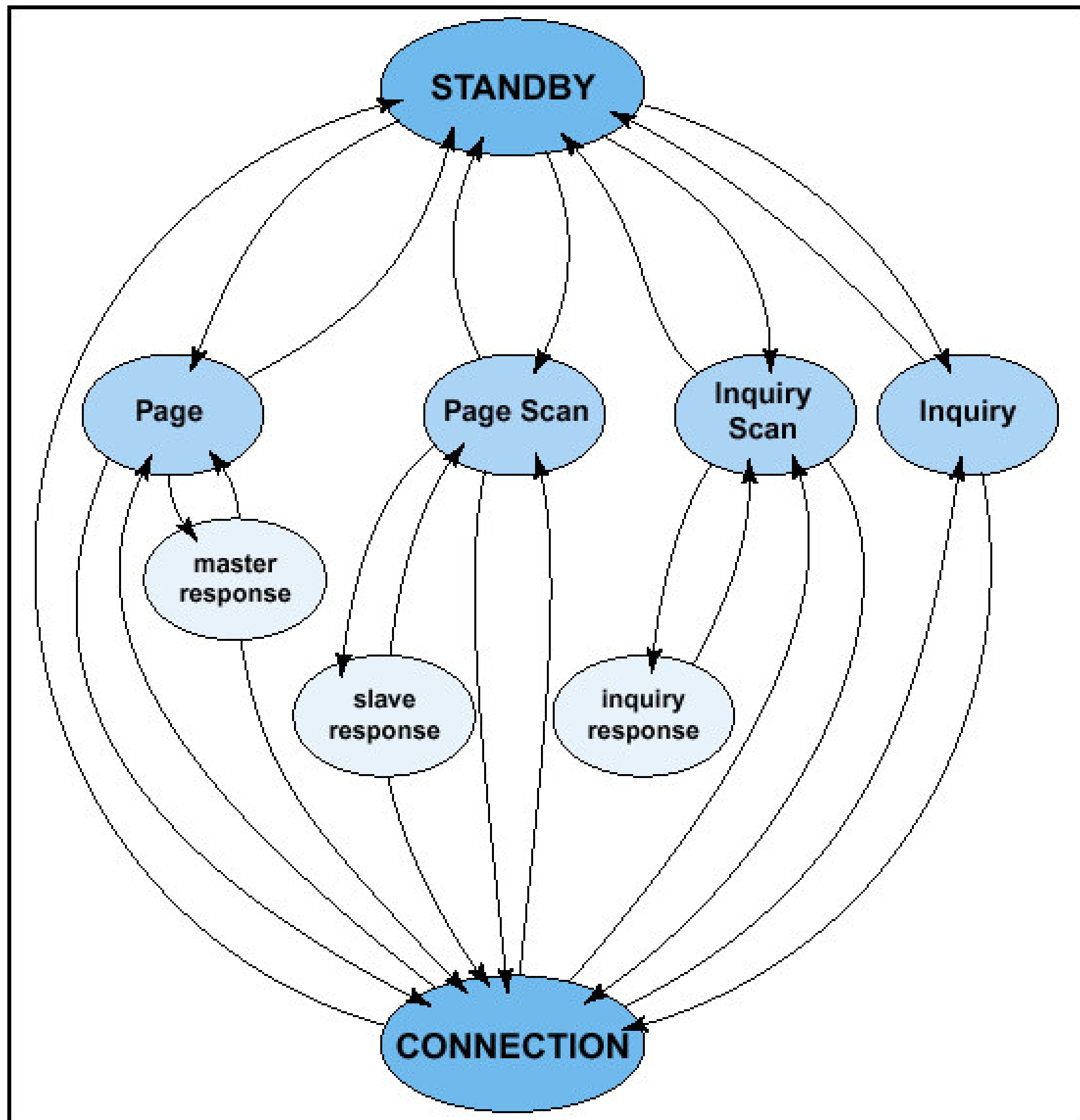


FIGURA 2.4: DIAGRAMA DE ESTADOS PARA ESTABELECIMENTO DE CONEXÕES

(Extraído de [10])

Um dispositivo Bluetooth permanece no estado “standby” por default, podendo transicionar para os estados: **inquiry, inquiry scan, page** e **page scan** quando for conveniente.

2.5.1 StandBy

Neste estado, apenas o clock nativo, necessário para que o dispositivo continue comunicando-se com os demais dispositivos, continua ativo. É um modo com consumo de potência extremamente baixo.

2.5.2 Inquiry

Neste estado, o dispositivo envia um tipo de pacote especial chamado de *inquiry packet* para descobrir quais os dispositivos presentes na sua área de alcance.

2.5.3 Inquiry Scan

De tempos em tempos, o dispositivo vai para este estado caso queira ser descoberto por outros dispositivos. Se receber algum *inquiry packet* enquanto estiver neste estado, responderá com um *inquiry response.* Cabe ressaltar, que um dispositivo nem sempre deseja ser descoberto. Existe um parâmetro de configuração nos dispositivos Bluetooth chamado de *inquiry enabled.* Caso este parâmetro esteja desativado, o dispositivo não entrará no estado *inquiry scan* e, desta forma, ele não será descoberto.

2.5.4 Inquiry Response

Neste estado, o dispositivo responde com seu endereço físico, permitindo que o dispositivo que fez o *inquiry* tome conhecimento da sua presença.

2.5.5 Demais estados

Os estados page, page scan e page response são análogos ao inquiry, inquiry scan e inquiry response respectivamente, porém, têm a função de realizar procedimentos de inicialização que permitem o estabelecimento de um link físico.

2.6 Modos de Operação

Uma vez que o procedimento de *paging* tenha sido realizado, os dispositivos estão prontos para estabelecer um link e trocar informações. Neste ponto, quatro modos de operação são suportados:

2.6.1 Active Mode

Neste modo, o dispositivo Bluetooth participa ativamente do canal. Master e slaves transmitem em slots alternados. São feitas diversas otimizações no sentindo de economizar potência. Por exemplo, o master informa ao slave quando ele será novamente escalonado, de maneira que até lá, ele possa "dormir". No modo ativo, o master consulta regularmente os slaves para verificar se eles querem transmitir (*polling).*

2.6.2 Sniff Mode

Este é um modo econômico na qual o tempo de escuta do canal é diminuído. O dispositivo slave é programado com três parâmetros: intervalo (Tsniff), offset (Dsniff) e

número de vezes (Nsniff). Desta forma, no modo sniff, o slave escuta as transmissões em intervalos fixos (Tsniff), no período de offset (Dsniff) **n*sniff*** vezes.

2.6.3 Hold Mode

Neste modo, o dispositivo slave fica em um estado de espera, impossibilitado de realizar transmissões assíncronas. Em vez disso, *scanning, inquiring, paging,* etc., são realizados.

2.6.4 Park Mode

Este é um modo com consumo baixíssimo de potência, onde o dispositivo passa a usar um endereço global (*parked address),* em vez do seu endereço físico. Em tempos regulares, o dispositivo escuta o canal para verificar eventuais transmissões.

3 HARDWARE BLUETOOTH

Antes de se entender o que existe em termos de Hardware em um sistema Bluetooth, é necessário antes entender as configurações possíveis deste sistema. A seguir são mostradas quatro configurações básicas.

3.1 Configurações Básicas

3.1.1 PC como HOST – Módulo Bluetooth externo

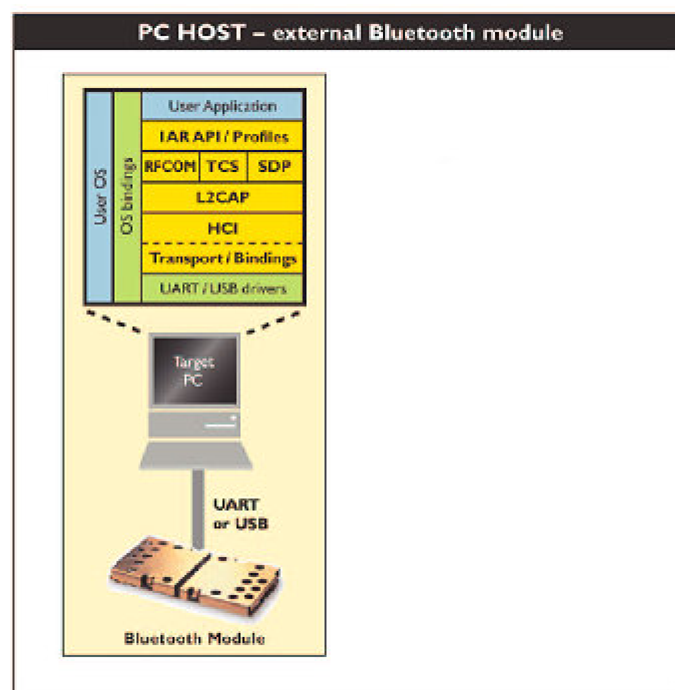


FIGURA 3.1: CONFIGURAÇÃO COM PC COMO HOST

(Adaptada de [13])

Nesta configuração, toda pilha de protocolos Bluetooth é implementada em Software e executada em um computador pessoal (PC) que é conectado ao módulo Bluetooth através de USB ou UART. É a típica configuração usada durante a fase de prototipagem.

3.1.2 Microcontrolador como HOST – Módulo Bluetooth externo

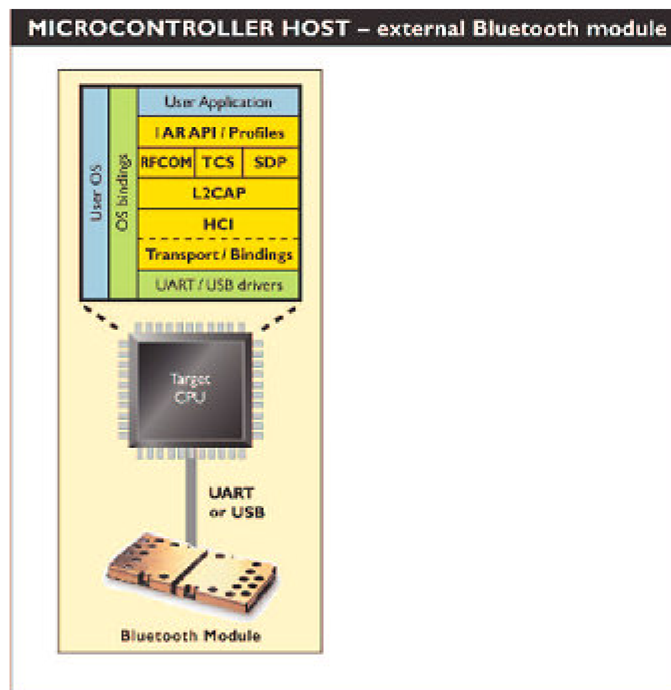


FIGURA 3.2: CONFIGURAÇÃO COM MICROCONTROLADOR COMO HOST

(Adaptada de [13])

Idêntica a configuração anterior, porém nesse caso, a pilha de protocolos é executada em um microcontrolador embutido. Essa é a típica configuração para desenvolvimento em sistemas embutidos.

3.1.3 Aplicação Integrada ao módulo

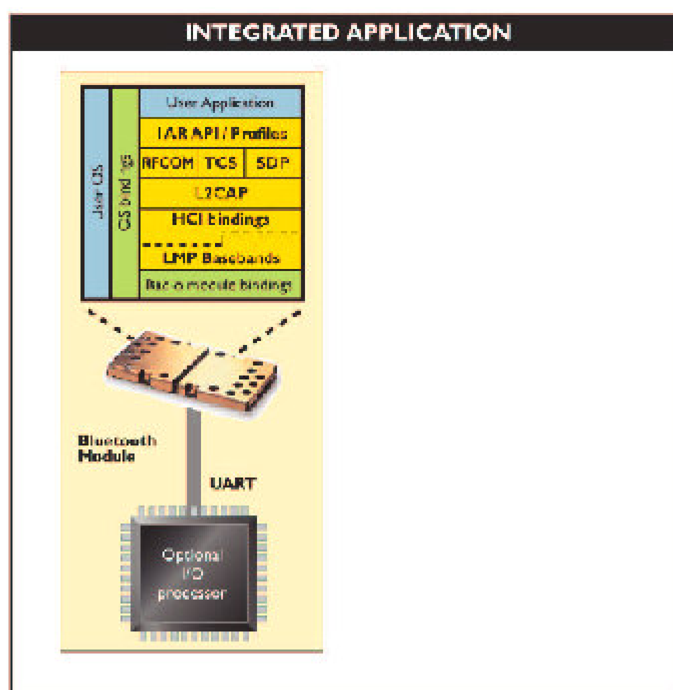


FIGURA 3.3: CONFIGURAÇÃO COM APLICAÇÃO INTEGRADA AO MÓDULO
(Adaptada de [13])

Nesta configuração, a aplicação é executada a partir da pilha de protocolos implementada no próprio módulo Bluetooth. Opcionalmente, pode ser usado um processador de I/O para interfaces externas.

3.1.4 Aplicação Integrada a um microprocessador

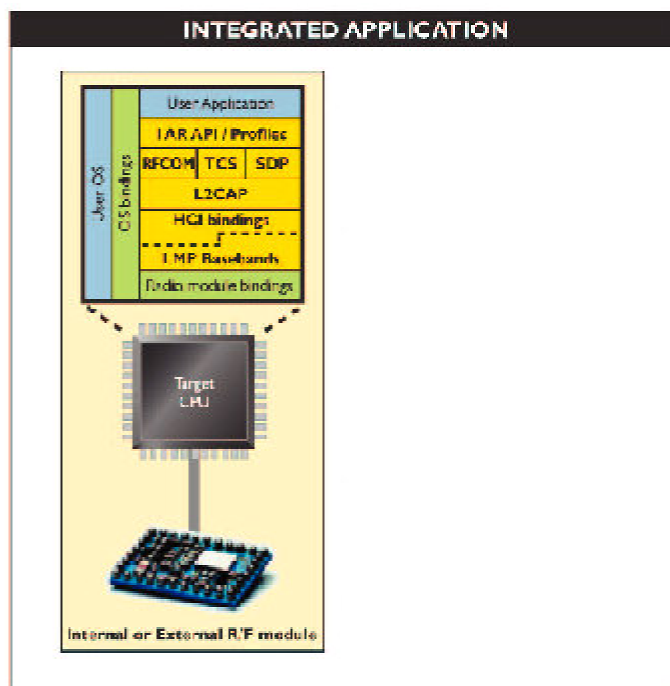


FIGURA 3.4: CONFIGURAÇÃO COM APLICAÇÃO INTEGRADA A UM MICROPROCESSADOR (Adaptada de [13])

Nesta configuração, a aplicação é executada em um microprocessador com funcionalidade Bluetooth embutida.

3.2 Hardware Usado

A configuração que será tratada daqui por diante é a primeira (fazendo uso de um PC). Foi feita esta escolha uma vez que não se quer desenvolver um produto final, nem algo compacto, mas sim, ter o mínimo suporte de Hardware necessário para se construir uma pilha de protocolos Bluetooth, em Software, com características que serão vistas adiante.

Em cima desta configuração, foram usados dois módulos Bluetooth, ambos com interface USB. Abaixo, é mostrada a foto de um dos módulos:

FIGURA 3.5: FOTO DO MÓDULO BLUETOOTH USADO

Para obter mais detalhes do módulo usado, consultar o ANEXO 1.

3.3 Especificações de Hardware

Todos os módulos Bluetooth devem seguir as especificações definidas pela SIG para que seus chips consigam comunicar-se com os demais chips existentes no mercado. Internamente, um módulo Bluetooth provê uma Interface de rádio e um enlace físico entre dois ou mais dispositivos:

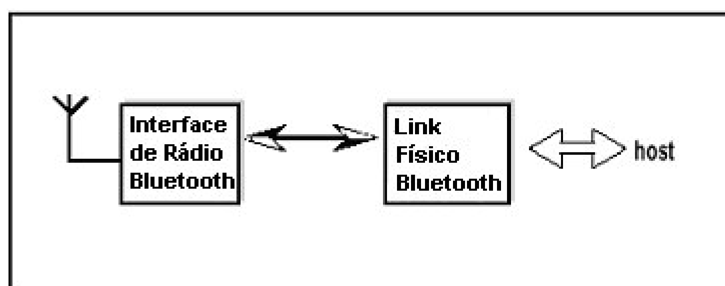


FIGURA 3.6: BLOCOS FUNCIONAIS DE UM MÓDULO BLUETOOTH
(Adaptado de [10])

O funcionamento básico do módulo é simples: um sistema microcontrolado sinaliza a Interface de rádio (partes que efetivamente realizam a comunicação: módulos de

transmissão/recepção, antena), para que seja estabelecido um link físico entre dois ou mais dispositivos.

3.3.1 Especificação de Rádio

Quando se especifica uma rede sem fio, seja Bluetooth ou qualquer outro tipo de rede sem fio, é necessário especificar como se dará a comunicação no nível mais baixo: o nível físico. Existem várias maneiras de prover comunicação sem fio, envolvendo diferentes parâmetros: faixas no espectro de freqüência, alocação de canais, modulações, níveis de ruídos tolerados, sensibilidade do receptor, potência mínima e desejada de transmissão, etc. Todas essas características são detalhadas para o sistema Bluetooth em [10]. A seguir serão mostradas algumas das principais características em relação ao nível físico tais como, banda de freqüência, alocação de canais, modulação, suficientes para dar uma boa idéia do que se espera do nível físico Bluetooth.

3.3.1.1 Banda de freqüência

O sistema Bluetooth opera na banda de 2,4 Ghz ISM (banda destinada a aplicações médicas e científicas). Na maioria dos países do mundo, a faixa destinada vai de 2400 a 2483.5 MHz, com exceção de alguns poucos países que possuem faixas mais limitadas. Para estes países, foram desenvolvidos algoritmos de comunicação especiais para suprir tal limitação [10].

A banda de freqüência é dividida em canais, usados para realizar a comunicação entre dispositivos Bluetooth. Estes canais existem graças a uma técnica de espalhamento espectral conhecida como FHSS, que será detalhada logo a seguir [10].

3.3.1.2 Modulação

Para permitir que a onda de rádio, que é uma onda analógica, consiga transmitir dados digitais, é preciso usar uma técnica conhecida como Modulação. Esta técnica permite que sinais analógicos consigam carregar consigo sinais digitais.

Modulação consiste em fazer com que algum parâmetro da onda analógica original (conhecida como portadora), amplitude, freqüência ou fase, varie de acordo com a informação a ser transmitida [7].

A fim de contextualizar o assunto, antes de mostrar a modulação usada no sistema Bluetooth, será dada uma breve explicação sobre as técnicas de modulação mais comuns.

AM – Amplitude Modulation

Varia-se a amplitude da onda analógica (portadora) para permitir a transmissão digital. Como pode ser visto na figura abaixo, o bit 1 é indicado por amplitudes iguais às da portadora. Já no caso do bit 0, a amplitude é cortada. Com isso, o receptor ao receber a onda modulada, consegue identificar o que é bit 0 e o que é bit 1, caracterizando assim, uma comunicação digital.

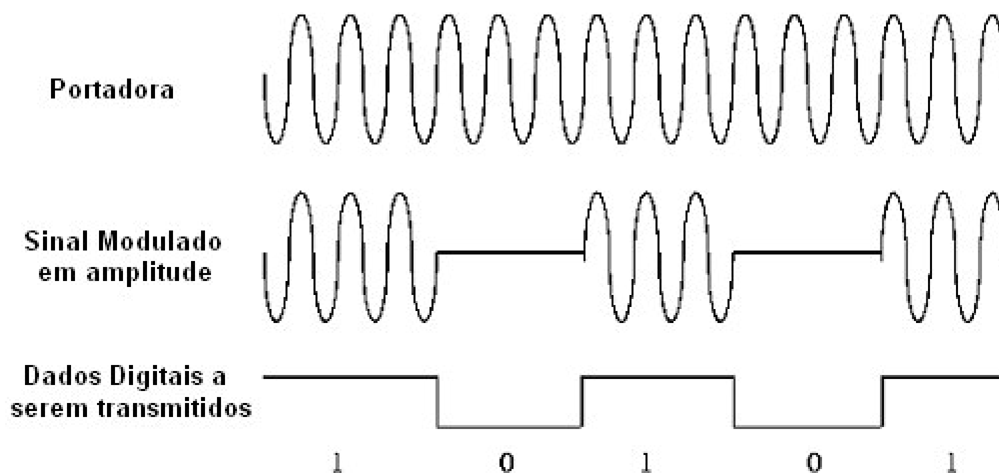


FIGURA 3.7: MODULAÇÃO EM AMPLITUDE

(Adaptado de [1])

FM – Frequency Modulation

Neste caso a variação de freqüência representa os dados digitais. Freqüências iguais à portadora representam bit 0, enquanto que freqüências superiores representam bit 1.

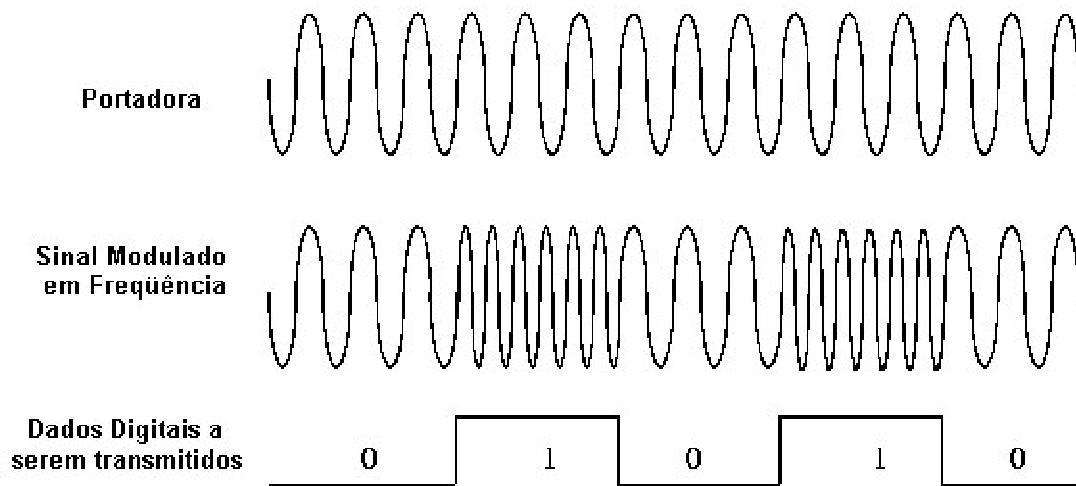


FIGURA 3.8: MODULAÇÃO EM FREQÜÊNCIA
(Adaptado de [1])

PM – Phase Modulation

Neste caso a variação de fase representa os dados digitais. Sinais modulado em fase com a portadora representam bit 0, enquanto que sinais com deslocamento da fase representam bit 1.

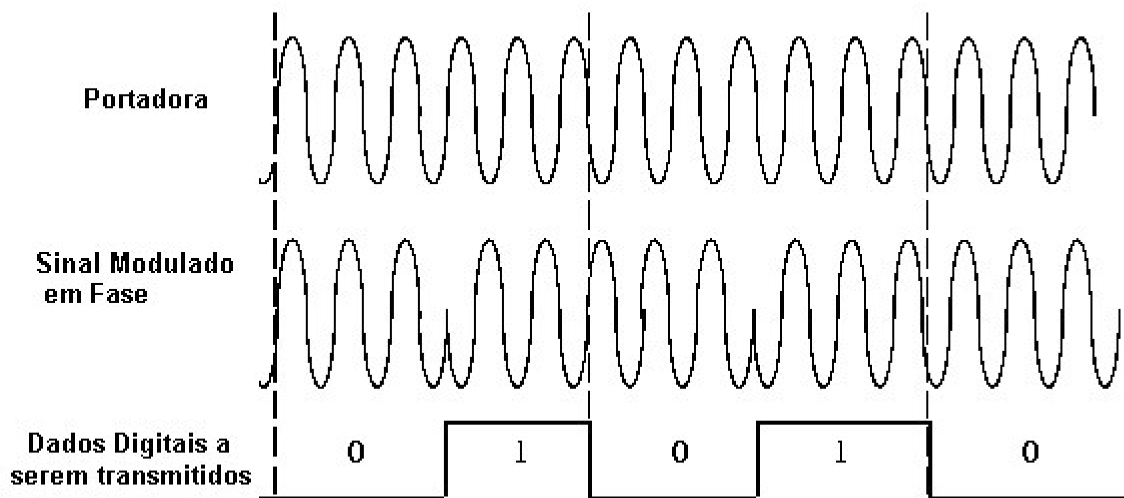


FIGURA 3.9: MODULAÇÃO EM FASE
(Adaptado de [1])

Modulação usada no sistema Bluetooth

A modulação usada pelo sistema Bluetooth é o FSK (Frequency Shift Keying). Esta técnica é semelhante a FM, mostrada anteriormente, porém FSK desloca a freqüência através de dois pontos separados e fixos. A freqüência mais alta é chamada de **marca**, enquanto que a freqüência menor é chamada de **espaço** (No caso de FM, o sinal pode mover-se para qualquer freqüência dentro da sua faixa de variação).

3.3.2 Especificação do link físico

Fazendo uso da interface de rádio, uma parte importante de um Hardware Bluetooth é toda inteligência que possibilita o estabelecimento de um link físico. Para que isso seja possível, a especificação Bluetooth [10] define todo um protocolo: formato de frames, diagramas de seqüência, etc. Como o objetivo deste trabalho não é estudar o Hardware, mas sim, o Software, esta parte não será coberta. O único tópico que será abordado é o espalhamento do sinal no espectro de freqüência. Isto porque esta é uma técnica bastante importante e, graças a ela, que os sistemas Bluetooth funcionam corretamente.

3.3.2.1 Espalhamento do sinal no espectro de freqüência

Como foi visto anteriormente, a banda de 2.4 GHz é uma banda destinada a aplicações médicas e científicas. Sendo assim, esta não é uma banda exclusiva ao sistema Bluetooth.

Por ser uma banda compartilhada, algumas técnicas são aplicadas para evitar os problemas decorrentes (ruídos, interferências, etc.). Estas técnicas fazem um espalhamento do sinal do espectro da freqüência. As duas mais conhecidas são DSSS e FHSS, sendo que o sistema Bluetooth faz uso de FHSS. A seguir é mostrada uma figura explicando a técnica FHSS:

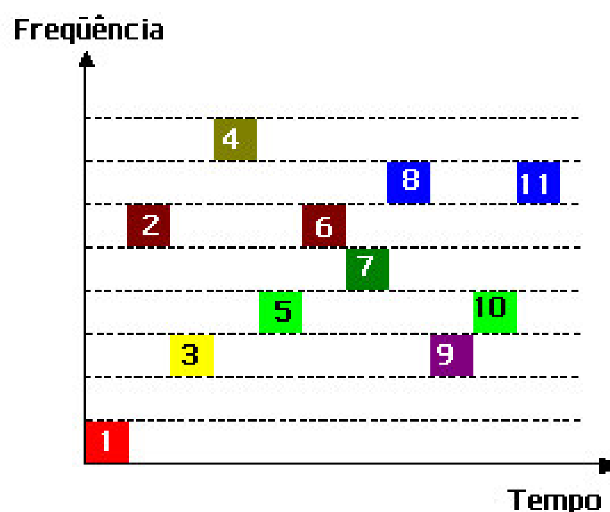


FIGURA 3.10: COMPORTAMENTO FHSS NO TEMPO

A técnica FHSS é simples. Basicamente, ao longo do tempo, os dados são transmitidos em diferentes freqüências. Como mostrado na figura acima, de um slot de tempo para outro, acontecem saltos de freqüência. A ordem com que as freqüências são escolhidas ao longo do tempo forma uma espécie de código. Este código é aleatório, diferente de transmissão para transmissão, obtido a partir do clock do master. Através dele, é possível: diminuir o nível de ruídos e interferências além de oferecer um certo nível de segurança, afinal só se consegue entender o que está sendo transmitido a partir do momento que se tiver conhecimento da maneira como são feitos os saltos de freqüência. Daí o porquê desta técnica ser usada não só pelo padrão Bluetooth, mas também em outros tipos de transmissões móveis como, por exemplo, alguns sistemas celulares.

4 PILHA DE PROTOCOLOS BLUETOOTH ADAPTÁVEL À APLICAÇÃO

Agora que já se tem todo o embasamento do que é o sistema Bluetooth, das suas funcionalidades básicas e do Hardware associado, pode-se começar a analisar o que é o objetivo principal deste trabalho: propor e implementar a parte referente à pilha de protocolos Bluetooth de uma maneira adaptável à aplicação.

4.1 Motivação

A pilha de protocolos sugerida pela SIG, mostrada na figura 2.3, envolve uma série de funcionalidades, sendo que, dependendo da aplicação em questão, algumas destas funcionalidades não serão usadas. Para implementar tudo o que foi especificado pela SIG, ter-se-ia algo imenso, se medido pelo seu tamanho de código. A pilha do Linux (Bluez), por exemplo, precisa mais de 100 Kbytes de memória, para que todos seus módulos sejam carregados e se possa estabelecer comunicação entre dois ou mais dispositivos. Como Bluetooth surge, principalmente, como um padrão para comunicação de pequeno porte (baixa potência, pouca memória, etc.), é claro que mais de 100 Kbytes são bastante significativos. Principalmente em sistemas embutidos, onde a tecnologia Bluetooth tende a ser largamente utilizada, esta quantidade de memória requerida é muito alta.

Neste sentido, surge a necessidade de desenvolver a pilha de protocolos de tal forma que seja mais condensada sem, contudo, perder em eficiência e funcionalidade. A alternativa sugerida é desenvolver uma pilha que se adapte a aplicação em questão.

A especificação Bluetooth inclui uma série de funcionalidades: criptografia, SDP, suporte a áudio, emulação de link serial, dentre outros. Claro que nem todas as aplicações

necessitam de todas as funcionalidades especificadas. Sendo assim, a pilha poderia ser compilada de tal forma a ter apenas aqueles recursos necessários à aplicação que se esteja rodando.

4.2 Pilha de protocolos adaptada à aplicação

A idéia básica para se ter uma pilha adaptada à aplicação é desenvolver uma camada de Software com o mínimo necessário de código, de modo que seja possível a comunicação entre dispositivos Bluetooth. Fazendo uso desta camada, seriam desenvolvidos módulos a parte e, cada aplicação, faria uso APENAS do módulos necessários para que ela funcione corretamente. Abaixo, um diagrama ilustrando esta idéia:

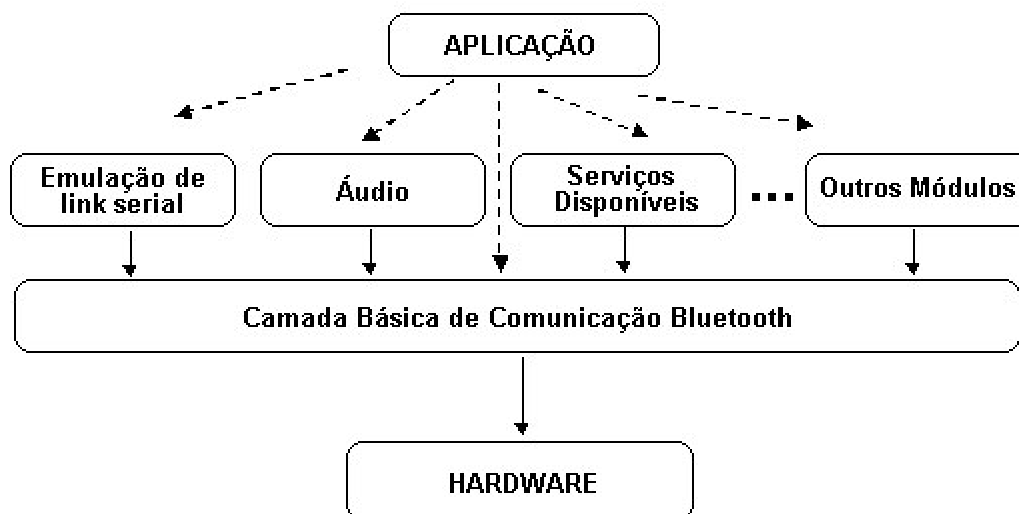


FIGURA 4.1: PILHA DE PROTOCOLOS ADAPTADA À APLICAÇÃO

Como foi dito em seções anteriores, o grupo SIG ao especificar a pilha de protocolos Bluetooth, pensou tanto em detalhes de Software como de Hardware. A parte da especificação que trata do Hardware, foi estritamente seguida, sendo implementada dentro do próprio chip a ser usado (figura 3.5). A parte de Software, porém, seguirá o modelo proposto acima. Abaixo, tem-se a pilha de protocolos sugerida pela SIG:

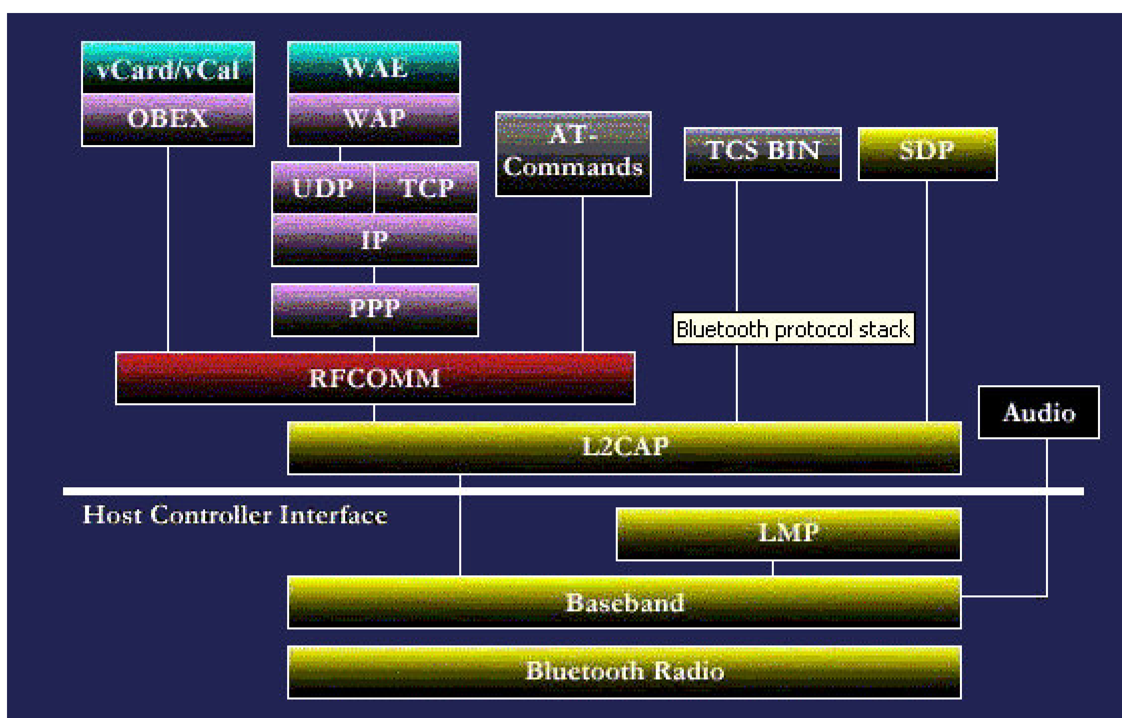


FIGURA 4.2: PILHA DE PROTOCOLOS COMPLETA SUGERIDA PELA SIG
(Extraído de [10])

Na figura, da linha branca para baixo, é implementado em Hardware. Ou seja: LMP, Baseband e Bluetooth Radio. Fazendo uso deste Hardware, da linha branca para cima, estará a proposta de protocolo adaptável à aplicação.

A parte de Software será implementada seguindo o modelo descrito na seção 3.1.1 (PC como HOST – Módulo Bluetooth externo), uma vez que os objetivos são experimentais e não comerciais e este é um modelo próprio para o desenvolvimento experimental.

Nas próximas seções, será descrito como desenvolver o protocolo adaptado à aplicação. Primeiramente, será mostrada a camada básica de comunicação e logo a seguir, módulos de adaptação.

4.2.1 Camada Básica de Comunicação Bluetooth

O objetivo da camada básica de comunicação é prover os meios mínimos necessários para prover a comunicação entre dois ou mais dispositivos Bluetooth. Como ela deve ter o mínimo de funcionalidade possível, nenhum item adicional é adicionado a sua implementação. Segundo a especificação desenvolvida pela SIG, para que haja comunicação entre dispositivos Bluetooth, precisa-se dos seguintes itens de Software:

1- HCI - a camada HCI fornece uma interface de Software básica para que sejam desenvolvidas camadas superiores. Métodos de transmissão/recepção (send/receive), são alguns dos métodos providos por esta camada;

2- Phsical BUS - Para permitir a comunicação entre o Host e o Hardware, existe também uma camada dependente do barramento de comunicação (Physical BUS).

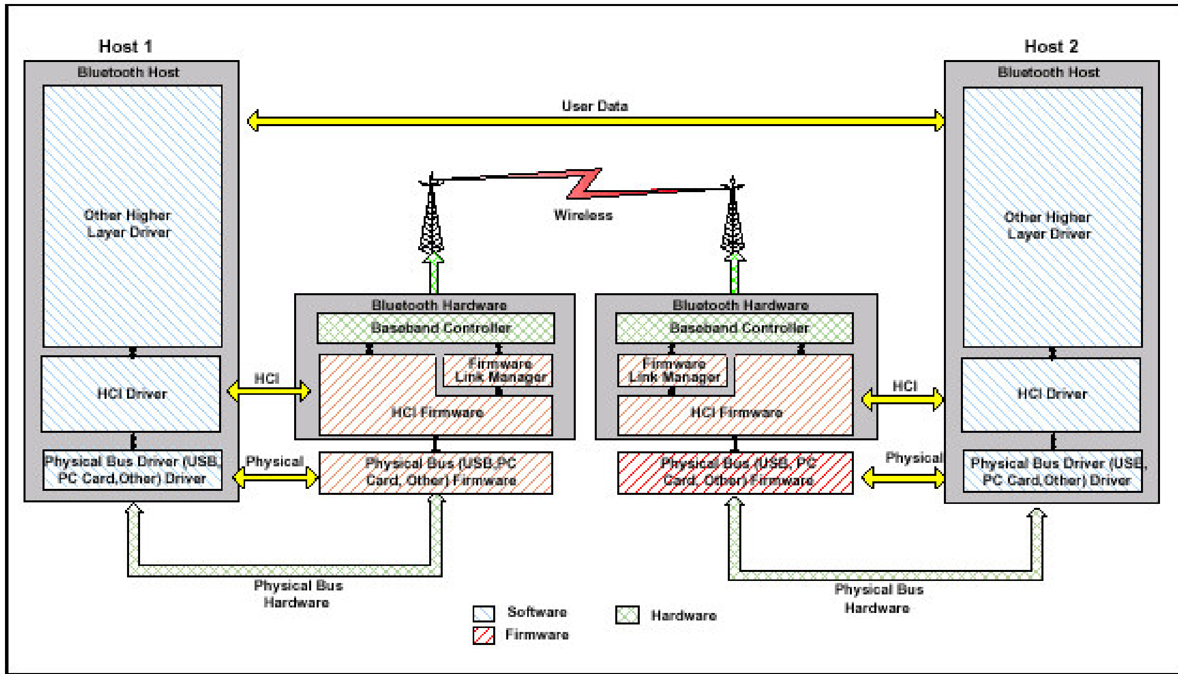


FIGURA 4.3: CAMADA BÁSICA DE COMUNICAÇÃO SUGERIDA PELA SIG

(Extraído de [10])

Este é o modelo proposto pela SIG. O modelo será adaptado de modo que possa ser facilmente integrado a um sistema de comunicação genérico (independente da tecnologia de comunicação empregada). Para tal modelagem, será usado *family based design*, uma técnica para modelagens de alto nível. Um sistema de comunicação genérico teria as seguintes famílias de abstrações:

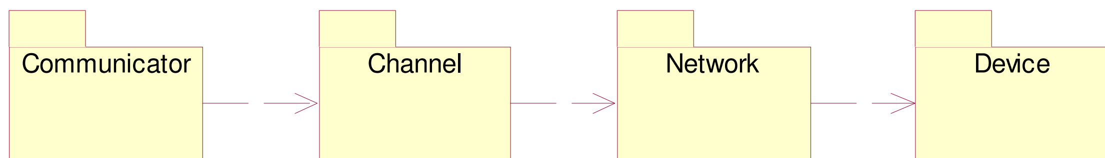


FIGURA 4.4: DIAGRAMA DE FAMÍLIAS DE UM SISTEMA DE COMUNICAÇÃO

Um communicator (comunicador) é o *end-point* de comunicação de processos. Quando uma aplicação cria um communicator, este implicitamente determinará qual será o tipo de comunicação a ser usado. Alguns membros da família communicator: Link, Port, Mailbox, dentre outros.

O communicator faz uso de um channel (canal). O canal é responsável por estabelecer um link lógico entre processos. Pode-se ter um canal para troca de datagramas, seqüência de bytes, etc. Daí surgem alguns dos membros da família channel: Stream, Datagram, ARM, dentre outros.

O channel faz uso de um network (rede) a qual efetivamente provê um link físico. Os membros desta família são as redes existentes atualmente (Ethernet, Myrinet, entre outras) além, é claro, darede Bluetooth.

A última família de abstrações é o device (dispositivo). A network faz uso de um device, que é a abstração de hardware, para realizar a comunicação física entre processos de aplicação.

O diagrama de famílias dá apenas uma amostra vaga de como seria um sistema genérico de comunicação. Em cima deste diagrama, será feito um diagrama de classes de modo a refinar o projeto do sistema e se chegar mais próximo da implementação propriamente dita. A seguir, o diagrama de classes do sistema de comunicação proposto:

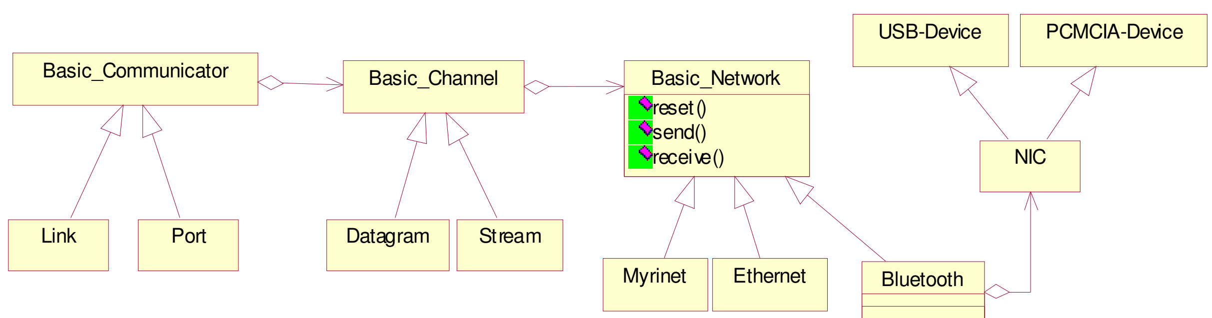


FIGURA 4.5: DIAGRAMA DE CLASSES DE UM SISTEMA DE COMUNICAÇÃO

O diagrama de classes é praticamente todo deduzido a partir do diagrama de famílias. Surgem, contudo, alguns detalhes que refinam o projeto. É o caso da classe NIC, agregação de *Bluetooth*. Esta classe oferece uma interface mais flexível com o Hardware. Ela permite que

dispositivos que têm diferentes modos de acesso, possam ser utilizados por uma network. Por exemplo, um dispositivo acessado através de portas de I/O (entrada e saída) ou através de mapeamento em memória, seria acessado da mesma maneira por um objeto da classe *network*.

Do jeito que está, a modelagem acima já poderia ser implementada e funcionaria corretamente. Porém, ter-se-ia a mesma camada mínima de comunicação independente da aplicação em questão. Para evitar isso e seguir a idéia de uma pilha de protocolos adaptável à aplicação, seguir-se-á a mesma idéia usada pelo sistema operacional EPOS [6], onde as abstrações do sistema de comunicação adaptam-se à aplicação em questão. Para se entender esta idéia, tome-se como exemplo a abstração **Bluetooth,** o qual será um dos membros da família network. Este membro terá algumas *configurable features* (funcionalidades configuráveis):

Connection Request: Enabled
Asynchronous, synchronous: synchronous
Inquiry: Enabled
Num Connection Accepted: 0
Configurable Parameters: PG_TIMEOUT, SCAN_ENABLE
Criptografy: Disabled

Esta funcionalidade pode ser configurada de modo fornecer uma camada mínima de comunicação adaptada à aplicação. Numa aplicação de voz, por exemplo, teria as funcionalidades: Comunicação síncrona, sem criptografia, dentre outras. Conexão síncrona, pois a transmissão de voz exige um fluxo contínuo dos dados. Sem criptografia, pois se considera que a aplicação não exiga segurança na comunicação. Baseado nestas funcionalidades configuradas seria gerado o mínimo de código necessário de modo que as funcionalidades requeridas sejam oferecidas. Isto é feito com o uso de meta-programação estática, com conceitos como: interfaces infladas, aplicação de aspectos de cenários, dentre outros. A explicação de tais conceitos vai além do escopo deste trabalho, podendo ser entendidos consultando-se [6].

Como já mostrado anteriormente (figura 4.3), para prover o suporte mínimo para que haja comunicação entre dispositivos Bluetooth, existem duas sub-camadas. Uma das camadas

é a HCI (Hardware Controller Interface) que oferece uma interface necessária para: transmissão/recepção de pacotes, envio de comandos ao controlador de Hardware, dentre outros. Já a outra camada, permite a comunicação entre HCI e o Hardware. Ela oferece o suporte necessário de maneira que a comunicação realize-se independente do BUS sendo usado: PCMCIA, USB, etc. A fim de entender melhor a implementação do sistema Bluetooth, as duas subcamadas acima serão mais detalhadas.

4.2.1.1 HCI

O HCI implementa algumas rotinas básicas para que se consiga estabelecer a comunicação entre dispositivos Bluetooth:

- *Rotina de Execução de Comandos*
- *Rotina para manipulação de eventos*
- *Rotina de Inicialização*
- *Rotina de Transmissão*
- *Rotina de Recebimento*
- *Rotina de Inquiry*
- *Rotina para Criação de Conexão*

O código completo, com a implementação das rotinas acima, bem como outras rotinas mais simples, pode ser visto no ANEXO 2. O relacionamento entre as funções no código e as rotinas descritas anteriormente é dado pela tabela abaixo:

Rotina descrita anteriormente	**Função no Código**
Rotina de Execução de Comandos	`billotooth_send_cmd`
Rotina para manipulação de eventos	`billotooth_event_packet`
Rotina de Inicialização	`billotooth_dev_open`
Rotina de Transmissão	`billotooth_send_acl`
Rotina de Recebimento	`billotooth_recv_frame`

Rotina de Inquiry	billotooth_inq_req, billotooth_inquiry_list_update, billotooth_inquiry_list_destroy, billotooth_inquiry_result_evt
Rotina para Criação de Conexão	billotooth_create_connection billotooth_conn_request_evt

TABELA 1: ROTINAS X FUNÇÕES

A seguir, uma descrição mais detalhada das rotinas acima.

Rotina de Execução de Comandos

A rotina de execução de comandos é responsável por enviar comandos ao Hardware, de modo que este o execute e envie uma resposta: *comando executado, status de execução, erro,* etc. Este é a interação básica entre Software e Hardware. Ou seja, o Software envia um comando, obtendo uma resposta. De acordo com a resposta, será enviado outro comando ou cancelado e assim por diante.

Para mandar um comando para o controlador de Hardware, existe um formato de frame a ser seguido, como mostrado abaixo:

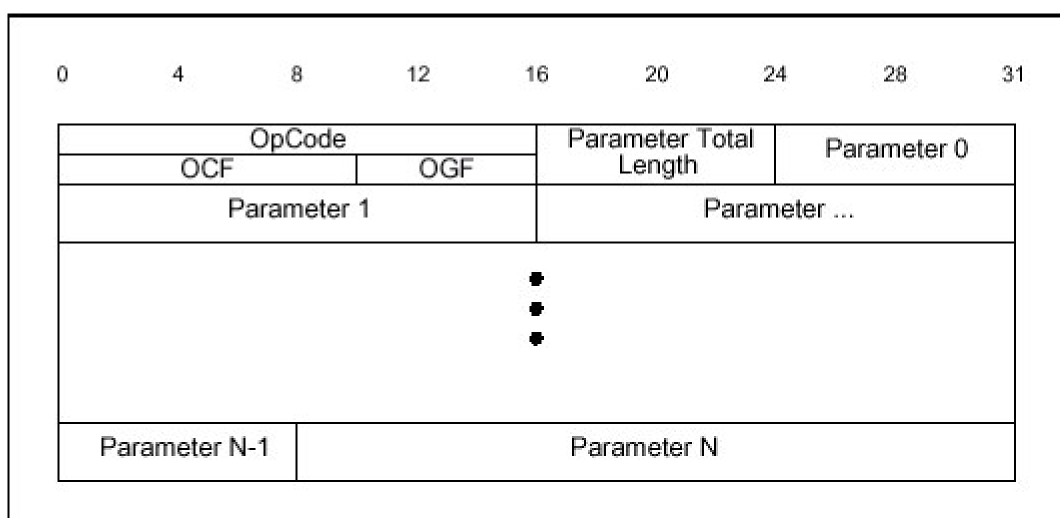


FIGURA 4.6: FORMATO DO FRAME DE COMANDO

(Extraído de [10])

Parâmetros:

OpCode (OCF + OGF): Os comandos possíveis de serem executados pelo controlador são divididos em grupos. Os grupos existentes, definidos pela SIG, são: *link controll, link policy, host controller and baseband, information parameters, status parameters* e *testing commands.* Para maiores informações sobre os comandos e seus grupos, consultar referência [10]. Com base nestes comandos, OGF identifica o grupo, enquanto que OCF identifica qual o comando a ser executado dentro do referido grupo.

Parameter Total Length: Tamanho em bytes dos parâmetros.

Parameter 0, 1, 2,..., N: São os parâmetros a serem passados ao comando. Análogo a uma chamada de função em uma linguagem de alto nível, onde se chama uma função, passando seus parâmetros. O número e tamanho em bytes de cada parâmetro variam de acordo com o comando chamado.

Rotina para manipulação de eventos

Como dito anteriormente, ao se executar um comando, o controlador de Hardware envia uma resposta ao Host. Esta resposta ativa uma função para tratamento deste evento. Para identificar qual evento que ocorreu, semelhantemente ao item anterior, existe um formato de frame pré-definido:

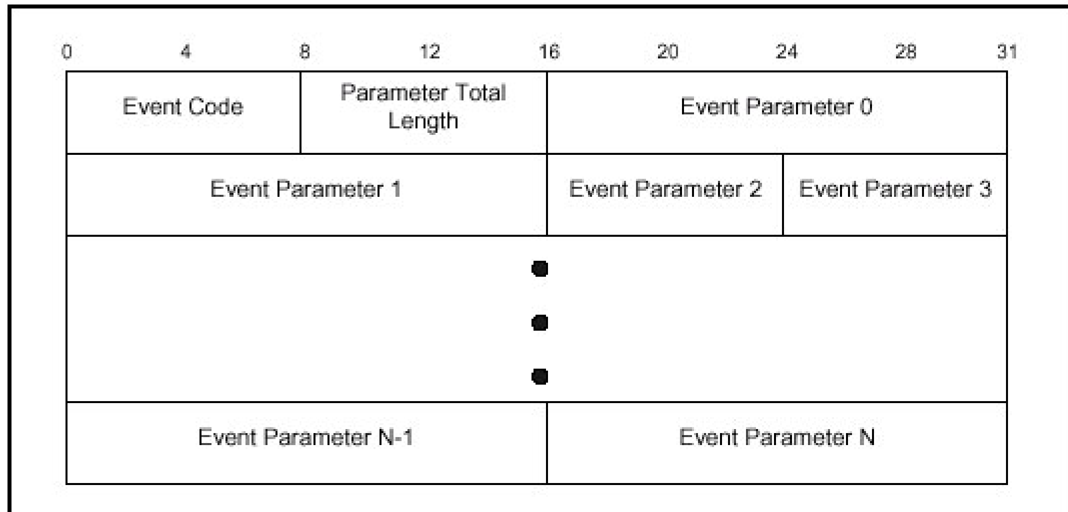


FIGURA 4.7: FORMATO DO FRAME DE EVENTO

(Extraído de[10])

Parâmetros:

Event Code: Código do evento. Permite ao Host identificar qual foi exatamente o evento ocorrido.

Parameter Total Length: Tamanho em bytes dos parâmetros.

Event Parameter 0, 1, 2,..., N: Novamente, tem-se parâmetros associados aos eventos. O número e tamanho em bytes de cada parâmetro variam de acordo com o evento ocorrido. Para se entender a utilidade de um parâmetro, imagine-se, por exemplo, o caso de um dispositivo A tentando conectar-se a um dispositivo B. O dispositivo B, neste caso, receberá um evento

chamado de *connection request*, sendo que um dos parâmetros deste evento é o endereço físico do dispositivo A.

Rotina de Inicialização

A seguir, é mostrado um diagrama ilustrando a rotina de inicialização:

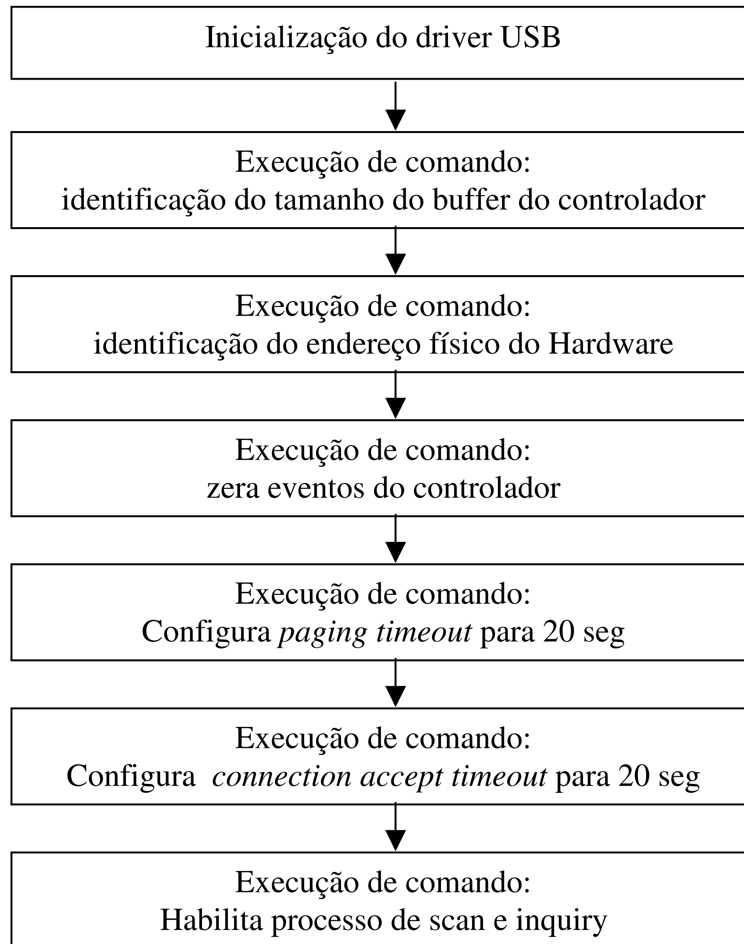


FIGURA 4.8: DIAGRAMA DE INICIALIZAÇÃO DE UM DISPOSITIVO BLUETOOTH

Como já dito anteriormente, quando o host solicita que o controlador de Hardware execute algum comando, este envia uma resposta. No caso do comando para identificação do endereço físico, por exemplo, esta resposta tem exatamente o endereço físico.

Rotina de Transmissão

Para se transmitir um pacote para um outro dispositivo Bluetooth, basta enviar ao controlador Bluetooth um frame com o formato abaixo:

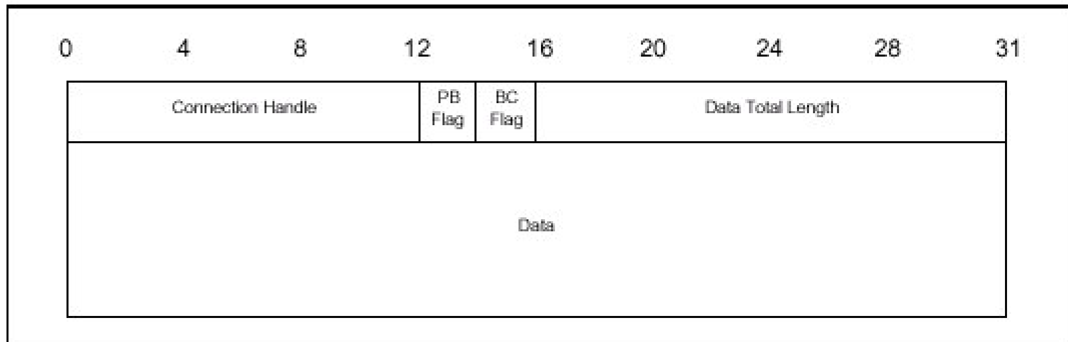


FIGURA 4.9: FORMATO DO FRAME DE TRANSMISSÃO
(Extraído de[10])

Parâmetros:

Connection Handle: Um dispositivo Bluetooth pode ter uma ou mais conexões, com um ou mais dispositivos Bluetooth. Isto permite, que múltiplas aplicações sejam executadas simultaneamente. Este parâmetro é um número que identifica a conexão em questão.

PB Flag: Indica se é uma mensagem fragmentada ou não. Fragmentada relativamente as camadas superiores.

BC Flag: Indica se é uma mensagem de Broadcast ou não.

Data Total Length: Tamanho em bytes a serem transmitidos.

Data: Os dados a serem transmitidos.

Rotina de Recebimento de Pacotes

Ao receber um pacote, inicialmente identifica-se a natureza do pacote. Este pode ser: um evento, um pacote síncrono ou um pacote assíncrono. Se for um evento, será chamada a rotina para manipulação de eventos, descrita acima. Se for um pacote assíncrono ou síncrono, este será repassado às camadas superiores.

Rotina de Inquiry

Esta rotina é simples. Basicamente, é enviado um comando de inquiry ao controlador. Após isto, recebe-se o evento inquiry result, com os dispositivos localizados. Armazenam-se então, os endereços destes dispositivos em uma lista encadeada de maneira que esta possa ser consultada posteriormente pelo host.

Rotina para Criação de Conexão

Esta é outra rotina simples. É enviado um comando de connection request**,** para criação da conexão. Como parâmetro, tem-se o endereço físico do host a qual se deseja realizar a conexão.

4.2.1.2 Suporte a USB

Como mostrado na figura 4.4, uma parte importante em relação ao suporte mínimo de comunicação, é o driver do BUS. A especificação SIG prevê uma série de barramentos possíveis: PCMCIA, placa de PC, USB, dentre outros. O hardware usado tem uma interface USB.

Inicialmente, tinha-se como objetivo a implementação de todo suporte USB. Isto porque seria possível simplificar os drivers USB existentes no mercado, tornando-os menos genéricos, específicos para os dispositivos Bluetooth. Embora a idéia seja boa, uma vez que diminuiria significativamente o tamanho e complexidade do driver USB, o esforço necessário seria muito grande, o que tornou sua implementação inviável. Para ser ter uma idéia, o sistema de suporte USB do linux, levou anos para se tornar estável. Sendo assim, resolveu-se usar os drivers existentes nas distribuições do Linux.

A tecnologia USB foi desenvolvida de modo a ser um barramento facilmente expansível. Ou seja, em uma porta USB, pode-se conectar HUBs (com a mesma funcionalidade de uma rede de computadores), tornando ilimitado, o número de dispositivos que podem ser conectados numa porta, bem como, as topologias possíveis de serem configuradas. Além disso, os dois tipos mais conhecidos de controladores USB são: UHCI e OHCI. No caso do OHCI, não seria difícil o desenvolvimento do seu driver, pois a maioria das funcionalidades está concentrada em Hardware. Porém no caso, do driver UHCI, a parte "inteligente" do barramento é implementada no driver, tornando-o extremamente complexo. Para se transmitir um conjunto de dados, por exemplo, eles devem ser organizados na memória de uma maneira extremamente complexa. Por todos estes motivos a implementação tornou-se inviável, sendo que foi dada atenção a outras partes mais importantes do trabalho.

4.2.2 Módulos de Adaptação

Além da camada de comunicação mínima que se adapta à aplicação em questão, módulos de adaptação também são usados (figura 4.1).

Em vez de explicar uma série de módulos de adaptação, apenas um basta para se entender a idéia. Será explicado e implementado um módulo de controle centralizado.

4.2.2.1 Módulo para controle centralizado

O módulo para controle centralizado é um excelente exemplo de módulo de adaptação, uma vez que ele pode ser usado por inúmeras aplicações Bluetooth. Ele serve para controle centralizado de dispositivos eletrônicos em geral. Exemplos de aplicações onde ele pode ser usado: controle de robôs em uma fábrica, vigilância eletrônica, controle de aparelhos eletro-eletrônicos em uma casa, etc.

O funcionamento deste módulo é simples. Ele oferece uma interface para que a aplicação localizada em um computador central envie comandos para dispositivos eletrônicos que por sua vez enviam respostas. O módulo basicamente emula uma máquina de estado, onde comandos dados pelo computador central, ou respostas dadas pelos dispositivos eletrônicos causam transições de estado. A aplicação que for usar este módulo, apenas terá que definir uma máquina de estados (estados, transições, funções de entrada e saída). Esta idéia ficará mais clara, na explicação de uma aplicação que faz uso deste módulo (seção 4.3.2).

4.3 Exemplos de aplicações

Existem uma infinidade de aplicações possíveis de serem implementadas usando Bluetooth. Uma delas é o acesso a Internet, como descrito na seção 2.4. Outras aplicações possíveis:

- Comunicação entre fone de ouvido e aparelho celular
- Comunicação entre computador e impressora
- Controle de Robôs
- Acesso a Internet via dispositivos Bluetooth
- Comunicação entre aparelhos eletro-eletrônicos

Para demonstrar o sistema Bluetooth, foram implementadas duas aplicações. Sua camada de comunicação mínima foi configurada com as seguintes funcionalidades:

Connection Request: Enabled
Asynchronous, synchronous: Asynchronous
Inquiry: Enabled
Num Connection Accepted: 1
Configurable Parameters: PG_TIMEOUT, SCAN_ENABLE
Criptografy: Disabled

Além disso, ambas aplicações fazem uso do módulo para controle centralizado.

4.3.1 Vigilância Centralizada em Grandes Edificações

Em grandes edificações, como um prédio do governo, por exemplo, nos horários fora do expediente, é preciso que o prédio seja vigiado contra roubos, entrada de pessoas indevidas, etc. Em vez de ter uma pessoa que fique andando pelo prédio para fazer a vigia, pode-se usar câmeras e detectores de presença com suporte a Bluetooth, que mandam as informações para um computador central. Desta forma, apenas uma pessoa pode cuidar de todo um prédio, apenas olhando para a tela de um computador. Para emular esta aplicação, foi usado um detector de presença. Quando ele detecta a presença de alguém, o computador central gera um alarme.

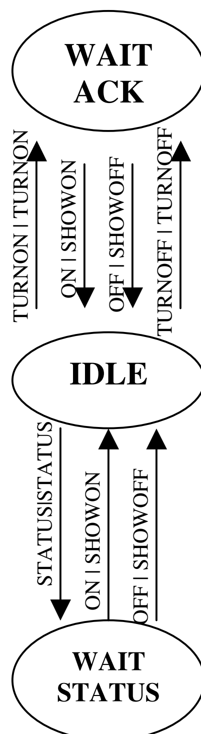
Esta aplicação foi feita de maneira muito simples, fazendo uso do módulo para controle centralizado, descrito acima.

4.3.2 Controle Centralizado de Aparelhos Elétricos em Grandes Edificações

Usando o mesmo exemplo anterior, um prédio do governo, sabe-se que neste tipo de edificação geralmente existe uma série de aparelhos elétricos a serem controlados por uma pessoa encarregada. O Ar Condicionado é um exemplo. Ele deve ser ligado pouco antes do expediente, e desligado após o expediente, ou quando não houver pessoas no recinto. Seguindo a mesma idéia da aplicação anterior, o aparelho também poderia ser ligado e desligado a partir de um computador central, evitando o deslocamento do empregado responsável. Inclusive, poder-se-ia fazer uso da aplicação anterior: caso o computador central verificasse que um determinado recinto está sem nenhuma pessoa por muito tempo (através do detector de presença), o ar condicionado seria automaticamente desligado.

Semelhantemente a aplicação anterior, esta aplicação também fez uso do módulo para controle centralizado, sendo necessário definir a máquina de estados abaixo:

Computador Central **Ar-condicionado**



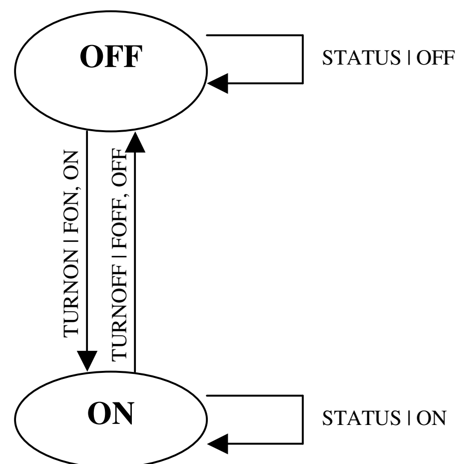


FIGURA 4.10: DIAGRAMA DE ESTADOS DE UMA APLICAÇÃO BLUETOOTH

A máquina de estados funciona da seguinte forma: caso o computador central dê um "TURNON" ou "TURNOFF", ele passará para o estado de "WAITACK". Do lado do ar-condicionado, será chamada a função FON ou FOFF, que liga ou desliga o ar-condicionado respectivamente. Além disso, é retornado ON ou OFF de modo que o computador central tome conhecimento da operação bem sucedida e volte para o estado de IDLE. A operação de STATUS é similar, porém, em vez de o ar-condicionado ser ligado ou desligado, ele apenas retorna seu status atual.

5 CONCLUSÃO

A proposta de uma pilha de protocolos adaptável à aplicação mostrou resultados bastante positivos. As duas aplicações desenvolvidas como exemplo (Vigilância Centralizada em Grandes Edificações e Controle Centralizado de Aparelhos Elétricos em Grandes Edificações), quando executadas usando a pilha de protocolos proposta neste trabalho, requereram ¼ da memória utilizada se rodassem utilizando a pilha de protocolos atualmente distribuída com o sistema operacional Linux.

Além disso, a pilha de protocolos mostrou outras vantagens. Uma delas está em facilitar enormemente a programação de uma aplicação Bluetooth, devido ao uso de módulos de adaptação. No caso das duas aplicações mostradas, o desenvolvimento da aplicação consistia apenas em definir uma máquina de estados. Uma outra vantagem é a sua fácil integração com um sistema de comunicação genérico como o sistema de comunicação do EPOS [6]. Com isso, funcionalidades fornecidas pelo sistema operacional a outras tecnologias de comunicação, seriam fornecidas igualmente a tecnologia Bluetooth.

Este trabalho proporcionou, além do projeto de uma pilha de protocolos, o conhecimento do Hardware Bluetooth, que utiliza tecnologias de ponta, como FHSS, para realizar comunicação sem fio.

Algumas partes ficaram incompletas, existindo uma série de possibilidades de trabalhos futuros. Uma delas, é o desenvolvimento dos *BUS drivers.* Este é um trabalho bastante importante, uma vez que os drivers existentes são feitos de maneira a serem compatíveis com qualquer Hardware, não só Bluetooth. Desta forma, o desenvolvimento destes drivers melhoraria sensivelmente a compactação da pilha de protocolos proposta. Além disso, por ser tratar apenas de um protótipo inicial, a implementação da camada de comunicação básica não seguiu a modelagem mostrada neste trabalho. Para realizar tal implementação, basta seguir os conceitos envolvendo meta-programação propostas em [6].

Por último, módulos de adaptação poderiam ser disponibilizados, podendo até tornar a pilha de protocolos proposta, passível de ser distribuída comercialmente.

6 REFERÊNCIAS BIBLIOGRÁFICAS

[1] BLANCHARD, Eugene. **Introduction to Networking and Data Communications**. Publicado no site:"http://www.thelinuxreview.com/”, sob os termos da licensa GNU, 2001.

[2] BLUEZ Authors. **Official Linux Bluetooth protocol stack**. Disponível no site: "http://bluez.sourceforge.net/". 2002.

[3] BOVET, Daniel P. CESATI, Marco. **Understanding the Linux Kernel**. O’Reilly & Associates. 2000.

[4] ERIKSSON, Hans-Erik. **UML Toolkit**. New York: Wiley Computer Publishing, 1998.

[5] FLIEGL, Detlef. **Programming Guide for Linux USB Device Drivers**. Publicado no site "http://usb.cs.tum.edu/usbdoc/". 2000.

[6] FRÖHLICH, Antônio A., **Application-Oriented Operating Systems**. Berlin: Technical University, 2001. (Ph.D. Thesis).

[7] HELMS, Harry. **Modes and Modulation**. Publicado no site "http://www.dxing.com/modesand.htm". 2000.

[8] KANSAL, Aman. **Bluetooth Primer**. Los Angeles: Red-M. 2002.

[9] RUBINI, Alessandro. CORBET Jonathan. **Linux Device Drivers**, 2nd Edition. O'Reilly & Associates. 2001.

[10] SIG (Special Interest Group). **Specification of the Bluetooth System**. 2001.

[11] SILVA, Ricardo Pereira e. **Eletronica básica : um enfoque voltado à informatica**. Florianopolis: Ed. da UFSC, 1995.

[12] STEVENS, W. Richard. **UNIX network programming**. 2nd Edition. Upper Saddle River: Prentice Hall PTR, 1998.

[13] UNEMYR, Magnus. **A Bluetooth protocol stack for embedded use**. Publicado no site "www.iar.com/FTP/pub/press/articles/ BT_stack_embedded_use.pdf" por um funcionário da IAR Systems. 2002.

APÊNDICES

APÊNDICE 1 – Artigo

Uma Pilha de Protocolos Bluetooth
Adaptável à Aplicação

Eduardo Afonso Billo

Bacharelado em Ciências da Computação, 2003
Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC), Brasil, 88040-050
Laboratório de Integração Software/Hardware
Fone (0xx48) 234-8163, Fax (0xx48) 234-8163
billo@inf.ufsc.br

Resumo

Este artigo apresenta a proposta de uma pilha de protocolos a ser usada em um sistema de comunicação Bluetooth. A justificativa desta proposta está no fato de que Bluetooth é uma tecnologia desenvolvida essencialmente para ser usada em sistemas embutidos onde a capacidade de memória, processamento, etc., são bastante limitados, surgindo assim a necessidade de desenvolver uma pilha de protocolos com tamanho sensivelmente diminuído, sem perdas em funcionalidade e eficiência.
Palavras-chave: Bluetooth, comunicação sem fio, metaprogramação estática, módulos

Abstract

This article proposes a protocol stack to be used by a Bluetooth Comunication System. The justificative of this proposal is that Bluetooth is a tecnology commonlly used by embedded systems, where the memory size, processing capacity, etc., are very limited, coming up the need of developing a protocol stack with a smaller size, with no functionality nor eficiency losses.
Keywords: Bluetooth, wireless, static metaprogramming, modules.

Introdução

Na última década vem sendo cada vez mais comum o uso de redes sem fio em substituição às redes cabeadas para suprir a comunicação de dispositivos móveis: PDAs, telefones celulares, laptops e, ultimamente, comunicação de aparelhos eletrônicos de uso geral. O principal padrão usado em redes sem fio é o IEEE 802.11, que oferece uma banda relativamente larga (cerca de 11 Mbps – o que é alto, em se tratando de comunicação sem fio). Em contraste ao IEEE 802.11, vem surgindo o padrão Bluetooth que, para algumas aplicações, é uma tecnologia que oferece inúmeras vantagens.

Bluetooth

O padrão Bluetooth foi desenvolvido pelo SIG (Special Interest Group), um grupo formado por algumas empresas líderes mundiais nas telecomunicações, computação e indústrias de redes [10].

Bluetooth é um padrão com três características essenciais: consumo de potência baixíssimo, baixo alcance, taxas de transmissão baixas. Essas características contrastam com o padrão IEEE 802.11, que tem consumo maior, maior alcance e taxas de transmissão superiores. Então, porque trocar uma tecnologia que oferece alcance e taxas de transmissão maiores pelo Bluetooth? O principal motivo é o custo. Um chip bluetooth pode ser encontrado no mercado por menos de cinco dólares. Outro motivo importante é o consumo de potência, que é menor. Além desses motivos, não são todas as aplicações que necessitam de um alcance e taxas de transmissão altas.

Pilha de Protocolos Bluetooth

Um chip bluetooth por si só não faz nada. Obviamente que é preciso todo o suporte de Software para permitir a comunicação entre dispositivos móveis. Este suporte é dados através de uma pilha de protocolos.

A especificação da pilha de protocolos Bluetooth é aberta e foi desenvolvida pela SIG. Abaixo, uma figura mostrando esta pilha de protocolos:

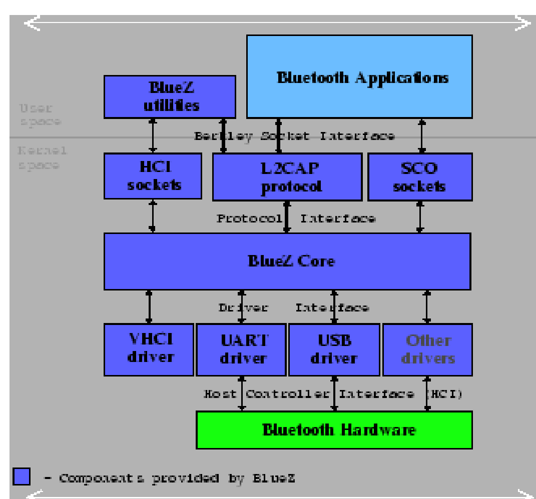


Figura 1: Overview da pilha de protocolos Bluez

A figura acima mostra pilha de protocolos Bluez, parte integrante das distribuições mais recentes do Linux.

Uma outra abordagem para a pilha Bluetooth

A pilha Bluez, bem como outras pilhas de protocolos Bluetooth, são imensas se medidas pelo seu tamanho de código. A pilha Bluez, por exemplo, precisa cerca de 100 Kbytes de memória para que todos seus módulos sejam carregados e se possa estabelecer uma comunicação mínima (sem módulos adicionais), entre dois ou mais dispositivos. Como bluetooth surge,

principalmente, como um padrão para comunicação de comunicação de pequeno porte (baixa potência, pouca memória, etc.), é claro que 100 Kbytes são bastante significativos. Principalmente em sistemas embutidos, onde a tecnologia bluetooth tende a ser largamente utilizada, esta quantidade de memória requerida é muito alta.

Neste sentido, surge a necessidade de desenvolver a pilha de protocolos de tal forma que seja mais condensada sem, contudo, perder em eficiência e funcionalidade. A alternativa sugerida é desenvolver uma pilha que se adapte à aplicação em questão.

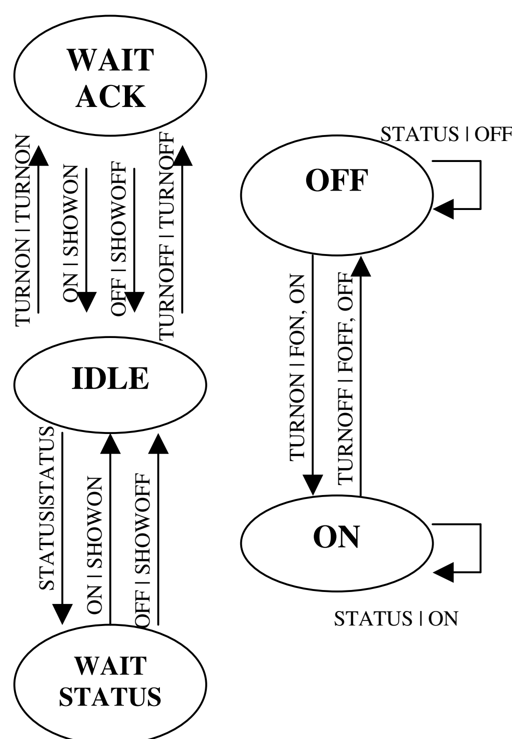
A especificação bluetooth inclui uma série de funcionalidades: criptografia, comunicação simétrica e assimétrica, suporte multi-tarefas, dentre outros. Claro que nem todas as aplicações necessitam de todas as funcionalidades especificadas. Sendo assim, a pilha poderia ser compilada de tal forma a ter **apenas** aqueles recursos necessários à aplicação que se esteja rodando. O desenvolvedor de Software Bluetooth, por sua vez, antes de compilar sua aplicação, teria que editar um arquivo semelhante ao seguinte:

Encryption – On
Transmition Mode – Sync
Multi-Thread – Off
...

Baseado no arquivo acima, os módulos gerados seriam bem menores. Para se chegar a isso, é usado metaprogramação estática, semelhantemente ao sistema operacional EPOS [6], onde o sistema operacional também é compilado de acordo com a aplicação em questão.

Além da camada de comunicação mínima que se adapta à aplicação em questão, módulos de adaptação também são usados.

Em vez de explicar uma série de módulos de adaptação, apenas um basta para se entender a idéia. Será explicado o módulo de controle centralizado. O funcionamento deste módulo é simples. Ele oferece uma interface para que a aplicação localizada em um computador central envie comandos para dispositivos eletrônicos que por sua vez enviam respostas. O módulo basicamente emula uma máquina de estado, onde comandos dados pelo computador central, ou respostas dadas pelos dispositivos eletrônicos causam transições de estado. A aplicação que for usar este módulo, apenas terá que definir uma máquina de estados (estados, transições, funções de entrada e saída), como mostrado abaixo:



Aplicabilidade e desenvolvimento atual

Atualmente foi iniciado o desenvolvimento desta pilha para o sistema operacional EPOS, arquitetura Intel x86. Porém, ela deverá ser facilmente portada para outros sistemas operacionais e arquiteturas. A pilha vem sendo desenvolvida fazendo-se uso de dois módulos bluetooth com interface USB:

Figura 2: Módulo Bluetooth atualmente utilizado

Conclusões

A proposta de uma pilha de protocolos adaptável à aplicação mostrou resultados bastante positivos. Uma aplicação usando o módulo descrito acima, quando executada usando a pilha de protocolos proposta neste trabalho, requereram ¼ da memória utilizada se rodassem utilizando a pilha de protocolos atualmente distribuída com o sistema operacional Linux.
Além disso, a pilha de protocolos mostrou outras vantagens. Uma delas está em facilitar enormemente a programação de uma aplicação Bluetooth, devido ao uso de módulos de adaptação. No caso do módulo acima, o desenvolvimento da aplicação consistia apenas em definir uma máquina de estados. Uma outra vantagem é a sua fácil integração com um sistema de comunicação genérico como o sistema de comunicação do EPOS [6]. Com isso, funcionalidades fornecidas pelo sistema operacional a outras tecnologias de comunicação, seriam fornecidas igualmente a tecnologia Bluetooth.

Referências Bibliográficas

[1] BLANCHARD, Eugene. **Introduction to Networking and Data Communications**. Publicado no site:"http://www.thelinuxreview.com/", sob os termos da licensa GNU, 2001.

[2] BLUEZ Authors. **Official Linux Bluetooth protocol stack**. Disponível no site: "http://bluez.sourceforge.net/". 2002.

[3] BOVET, Daniel P. CESATI, Marco. **Understanding the Linux Kernel**. O'Reilly & Associates. 2000.

[4] ERIKSSON, Hans-Erik. **UML Toolkit**. New York: Wiley Computer Publishing, 1998.

[5] FLIEGL, Detlef. **Programming Guide for Linux USB Device Drivers**. Publicado no site http://usb.cs.tum.edu/usbdoc/". 2000.

[6] FRÖHLICH, Antônio A., **Application-Oriented Operating Systems**. Berlin: Technical University, 2001. (Ph.D. Thesis).

[7] HELMS, Harry. **Modes and Modulation**. Publicado no site "http://www.dxing.com/modesand.htm". 2000.

[8] KANSAL, Aman. **Bluetooth Primer**. Los Angeles: Red-M. 2002.

[9] RUBINI, Alessandro. CORBET Jonathan. **Linux Device Drivers**, 2nd Edition. O'Reilly & Associates. 2001.

[10] SIG (Special Interest Group). **Specification of the Bluetooth System**. 2001.

[11] SILVA, Ricardo Pereira e. **Eletronica básica : um enfoque voltado à informatica**. Florianopolis: Ed. da UFSC, 1995.

[12] STEVENS, W. Richard. **UNIX network programming**. 2nd Edition. Upper Saddle River: Prentice Hall PTR, 1998.

[13] UNEMYR, Magnus. **A Bluetooth protocol stack for embedded use**. Publicado no site "www.iar.com/FTP/pub/press/articles/BT_stack_embedded_use.pdf" por um funcionário da IAR Systems. 2002.

ANEXOS

ANEXO 1 – Especificação do Hardware Usado

Main Specifications

Product Description	Acer Bluetooth Mini USB Adapter - network adapter
Bandwidth	2.4 MHz
Data Link Protocol	Bluetooth
Dimensions (WxDxH)	5 cm x 1.8 cm x 0.9 cm
Form Factor	External
Connectivity Technology	Wireless
Data Transfer Rate	1 Mbps
Compliant Standards	Plug and Play
Interface Type	USB
Device Type	Network adapter
Weight	10 g
Warranty	2 years warranty (24 months implied warranty in Germany)
System Requirements	Microsoft Windows Millennium Edition, Microsoft Windows 98 Second Edition / Windows 2000, Microsoft Windows XP

General

Height	0.9 cm
Depth	1.8 cm
Width	5 cm
Form Factor	External
Interface Type	USB
Device Type	Network adapter
Weight	10 g

Miscellaneous

Compliant Standards	Plug and Play

Warranty
(24 months implied warranty in Germany)

Service & Support	Limited warranty - parts and labour - 2 years - bring-In

Details	
Service & Support	2 years warranty

Expansion / Port(s) Required

Port(s) / Connector(s) Required	1 x serial - USB - 4 PIN USB Type A

Software / System Requirements

Software Included	Drivers & Utilities
Min Operating System	Microsoft Windows Millennium Edition, Microsoft Windows 98 Second Edition / Windows 2000, Microsoft Windows XP

Expansion / Connectivity

Port(s) Total / Connector Type	1 x network - Bluetooth

Networking

Bandwidth	2.4 MHz
Antenna	Integrated
Data Link Protocol	Bluetooth
Connectivity Technology	Wireless
Data Transfer Rate	1 Mbps

ANEXO 2 – Código HCI

```
#include <linux/config.h>
#include <linux/module.h>

#include <linux/types.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/poll.h>
#include <linux/fcntl.h>
#include <linux/init.h>
#include <linux/skbuff.h>
#include <linux/interrupt.h>
#include <linux/socket.h>
#include <linux/skbuff.h>
#include <linux/proc_fs.h>
#include <linux/list.h>
#include <net/sock.h>

#include <asm/system.h>
#include <asm/uaccess.h>
#include <asm/unaligned.h>

#include "Bluetooth.h"
#include "hci_core.h"

#define BT_INQUIRY_LENGTH 8
#define BT_INQUIRY_NUM_RESPONSES 10

int (*list_receptors[10]) (char * data);
int num_receptors = 0;
struct sk_buff * wait_acl_data = 0;

/* This is just a first version of the billotooth stack, so it will just support
   device which will be handled by the mydev struct. */
struct hci_dev *mydev;
/* inquiry device list */
struct inquiry_cache inq_cache;

static void billotooth_event_packet(struct sk_buff *skb);


/*-----------------------------------------------------------------------------------------------
                                        GENERAL-USE ROUTINES
-----------------------------------------------------------------------------------------------*/
char *batostr(bdaddr_t *ba)
{
        static char str[2][18];
        static int i = 1;

        i ^= 1;
        sprintf(str[i], "%2.2X:%2.2X:%2.2X:%2.2X:%2.2X:%2.2X",
                ba->b[0], ba->b[1], ba->b[2],
                ba->b[3], ba->b[4], ba->b[5]);

        return str[i];
}

void data_dump(struct sk_buff *skb) {
        int count;
        char byte;

       BT_DUMP("DATA: ");
        for (count=0 ; count < skb->len; count++) {
          byte = skb->data[count];
          BT_DUMP("%2.2X ",byte);
```

```
        }
        BT_DUMP("\n");
}

/*---------------------------------------------------------------------------------------------
                                    DATA TRANFER ROUTINES
---------------------------------------------------------------------------------------------*/
/* Send Data to the lower stack layer */
int billotooth_send_frame(struct sk_buff *skb)
{
        /* Get rid of skb owner, prior to sending to the driver. */
        skb_orphan(skb);
        BT_DBG("Sending data to the USB driver, len = %d",skb->len);
        data_dump(skb);
        return mydev->send(skb);
        return 0;
}

/* Send ACL data */
int billotooth_send_acl(struct hci_conn *conn, char * data, int len)
{
        struct sk_buff * skb;
        hci_acl_hdr *ah;
        __u16 flags = 0;

        skb = bluez_skb_alloc(HCI_ACL_HDR_SIZE+len, GFP_ATOMIC);
        if (!(skb)) {
                BT_ERR("Lack of memory in kernel to allocate structure");
        }

        skb->dev = (void *) mydev;
        skb->pkt_type = HCI_ACLDATA_PKT;
        /* Assembles the acl header */
        ah = (hci_acl_hdr *) skb_put(skb, HCI_ACL_HDR_SIZE);
        ah->handle = __cpu_to_le16(acl_handle_pack(conn->handle, flags | ACL_START));
        ah->dlen   = __cpu_to_le16(len);
        skb->h.raw = (void *) ah;
        if (len)
                memcpy(skb_put(skb, len), data, len);

        BT_DBG("Sending ACL %d bytes of data through connetion %p flags 0x%x", skb->len,conn,
flags);
        //data_dump(skb);
        return billotooth_send_frame(skb);
        //TODO: fragmented packets
}

/* Process received packet */
void hci_acldata_packet(struct sk_buff * skb)
{
        int i;
        BT_DBG("ACL packet received");
       data_dump(skb);
        //Send packet to upper layers
        for (i=0;i<num_receptors;i++)
        {
               list_receptors[i](skb->data+HCI_ACL_HDR_SIZE);
        }
        kfree_skb(skb);
}

/* Receive a frame from the lower stack layer */
int billotooth_recv_frame(struct sk_buff *skb)
{
        /* Make sure that the hci interface is UP */
        if (!mydev || (!test_bit(HCI_UP, &mydev->flags) &&
                                !test_bit(HCI_INIT, &mydev->flags)) ) {
                kfree_skb(skb);
                return -1;
        }

        BT_DBG("Frame received. Type = %d, Size = %d", skb->pkt_type, skb->len);
```

```
        /* Process frame */
        switch (skb->pkt_type) {
                case HCI_EVENT_PKT:
                        billotooth_event_packet(skb);
                        break;

                case HCI_ACLDATA_PKT:
                        hci_acldata_packet(skb);
                        break;

                case HCI_SCODATA_PKT:
                        BT_DBG("SCO packet received. HANDLER NOT IMPLEMENTED!!!");
                        kfree_skb(skb);
                        break;

                default:
                        kfree_skb(skb);
                        break;
        }
        return 0;
}


/*-----------------------------------------------------------------------------------------------
                                    COMMAND ROUTINES
-----------------------------------------------------------------------------------------------*/
/* Send a command to the HCI controller */
int billotooth_send_cmd(__u16 ogf, __u16 ocf, __u32 plen, void *param)
{
        int len = HCI_COMMAND_HDR_SIZE + plen;
        hci_command_hdr *hc;
        struct sk_buff *skb;

        /* allocate memory to skb */
        if (!(skb = bluez_skb_alloc(len, GFP_ATOMIC))) {
                BT_ERR("Error allocating memory");
                return -ENOMEM;
        }

        /* fill command operands */
        hc = (hci_command_hdr *) skb_put(skb, HCI_COMMAND_HDR_SIZE);
        hc->opcode = __cpu_to_le16(cmd_opcode_pack(ogf, ocf));
        hc->plen   = plen;
        if (plen)
                memcpy(skb_put(skb, plen), param, plen);

        BT_DBG("Sending command. Len = %d Ogf=0x%X Ocf=0x%X Plen=0x%X", skb-
>len,ogf,ocf,plen);

        skb->pkt_type = HCI_COMMAND_PKT;
        skb->dev = (void *) mydev;
        /* send command to the lower stack layer */
        return billotooth_send_frame(skb);
}

/*-----------------------------------------------------------------------------------------------
                                    INQUIRY ROUTINES
-----------------------------------------------------------------------------------------------*/
/* Request for an inquiry (location the devices nearby) */
void billotooth_inq_req(void)
{
        inquiry_cp ic;

        BT_DBG("Initializing inquiry...");

        /* fill command operands */
        ic.lap[0] = 0x33;
        ic.lap[1] = 0x8B;
        ic.lap[2] = 0x9E;
        ic.length = BT_INQUIRY_LENGTH;
        ic.num_rsp = BT_INQUIRY_NUM_RESPONSES;
        /* send command */
        billotooth_send_cmd(OGF_LINK_CTL, OCF_INQUIRY, INQUIRY_CP_SIZE, &ic);
```

```
}

/* Updates the list with a new device just found */
void billotooth_inquiry_list_update(inquiry_info *info)
{
        struct inquiry_entry *e;

        /* allocate memory to the inquiry entry */
        if (!(e = kmalloc(sizeof(struct inquiry_entry), GFP_ATOMIC)))
           return;

        /* insert device in the inquiry list */
        memset(e, 0, sizeof(struct inquiry_entry));
        e->next     = inq_cache.list;
        inq_cache.list = e;
        memcpy(&e->info, info, sizeof(inquiry_info));

        BT_DBG("New Device. Address = %s", batostr(&info->bdaddr));
}

/* Destroy the inquiry list */
void billotooth_inquiry_list_destroy(void) {
        struct inquiry_entry *e = inq_cache.list;
        /* Deallocate all the inquiry entries */
        while (e) {
          kfree(e);
          e = e->next;
        }
        BT_DBG("Inquiry list destroyed");
}


/* Destroy the inquiry list */
struct inquiry_entry *billotooth_inquiry_lookup(bdaddr_t *dst) {
        struct inquiry_entry *e = inq_cache.list;

        while (e) {
               if (!bacmp(&e->info.bdaddr, dst))
                        break;
                e = e->next;
                 BT_DBG("%s\n","Found device in inquiry lookup list");
        }
        return e;
}

/*--------------------------------------------------------------------------------------------
                                    CONNECTION ROUTINES
--------------------------------------------------------------------------------------------*/
struct hci_conn *billotooth_create_connection(bdaddr_t *dst)
{
        struct hci_conn *conn;

        BT_DBG("Creating connection with remote host %s", batostr(dst));

        if (!(conn = kmalloc(sizeof(struct hci_conn), GFP_ATOMIC)))
                return NULL;
        memset(conn, 0, sizeof(struct hci_conn));

        bacpy(&conn->dst, dst);
        conn->type   = ACL_LINK; //so far, only ACL supported
        conn->hdev   = mydev;
        conn->state  = BT_OPEN;
        conn_hash_add(mydev, conn);
        return conn;
}

void billotooth_acl_connect(struct hci_conn *conn)
{
        struct inquiry_entry *ie;
        create_conn_cp cp;

        BT_DBG("Connecting to remote host %s", batostr(&conn->dst));
```

```
        conn->state = BT_CONNECT;
        conn->out   = 1;
        conn->link_mode = HCI_LM_MASTER;

        memset(&cp, 0, sizeof(cp));
        bacpy(&cp.bdaddr, &conn->dst);

        if ((ie = billotooth_inquiry_lookup(&conn->dst))) {
                cp.pscan_rep_mode = ie->info.pscan_rep_mode;
                cp.pscan_mode     = ie->info.pscan_mode;
                cp.clock_offset   = ie->info.clock_offset | __cpu_to_le16(0x8000);
        }

       cp.pkt_type = __cpu_to_le16(mydev->pkt_type & ACL_PTYPE_MASK);
        cp.role_switch  = 0x01;
        billotooth_send_cmd(OGF_LINK_CTL, OCF_CREATE_CONN,
                                CREATE_CONN_CP_SIZE, &cp);
}

struct hci_conn * billotooth_connect(bdaddr_t *dst)
{
        struct hci_conn *acl;

        if (!(acl = conn_hash_lookup_ba(mydev, ACL_LINK, dst))) {
                if (!(acl = billotooth_create_connection(dst)))
                        return NULL;
        }

        if (acl->state == BT_OPEN || acl->state == BT_CLOSED)
                billotooth_acl_connect(acl);
        BT_DBG("%s","Leaving billotooth_connect");
        return acl;
}

void billotooth_acl_disconnect(struct hci_conn *conn, __u8 reason)
{
        disconnect_cp cp;

        BT_DBG("%s","Disconnecting...");

        conn->state = BT_DISCONN;

        cp.handle = __cpu_to_le16(conn->handle);
        cp.reason = reason;
        billotooth_send_cmd(OGF_LINK_CTL, OCF_DISCONNECT,DISCONNECT_CP_SIZE, &cp);

}


void billotooth_conn_complete_evt(struct sk_buff *skb)
{
       evt_conn_complete *cc = (evt_conn_complete *) skb->data;
       struct hci_conn *conn = NULL;

        conn = conn_hash_lookup_ba(mydev, cc->link_type, &cc->bdaddr);
        if (!conn) {
                return;
        }
       conn->handle = __le16_to_cpu(cc->handle);
        conn->state  = BT_CONNECTED;

        /* Set packet type for incomming connection */
        if (!conn->out) {
                       change_conn_ptype_cp cp;
                        cp.handle = cc->handle;
                        cp.pkt_type = (conn->type == ACL_LINK) ?
                                __cpu_to_le16(mydev->pkt_type & ACL_PTYPE_MASK):
                                __cpu_to_le16(mydev->pkt_type & SCO_PTYPE_MASK);

                        billotooth_send_cmd(OGF_LINK_CTL, OCF_CHANGE_CONN_PTYPE,
                                CHANGE_CONN_PTYPE_CP_SIZE, &cp);

       }
```

```
        BT_DBG("Connected with %s thought conn 0x%x", batostr(&cc->bdaddr),conn->handle);
}

void hci_disconn_complete_evt(struct sk_buff *skb)
{
        evt_disconn_complete *dc = (evt_disconn_complete *) skb->data;
        struct hci_conn *conn = NULL;
        __u16 handle = __le16_to_cpu(dc->handle);

        BT_DBG("%s","Disconnected...");

        if (dc->status)
                return;

        conn = conn_hash_lookup_handle(mydev, handle);
        if (conn) {
                conn->state = BT_CLOSED;
                conn_hash_del(mydev,conn);
        }
}

/*-----------------------------------------------------------------------------------------------
                                    EVENT ROUTINES
---------------------------------------------------------------------------------------------*/
/* Command Complete OGF HOST_CTL  */
static void billotooth_cc_host_ctl(__u16 ocf, struct sk_buff *skb)
{
        __u8 status;
        status = *((__u8 *) skb->data);

        switch (ocf) {
        case OCF_SET_EVENT_FLT:
                if (status) {
                        BT_DBG("SET_EVENT_FLT failed %d",  status);
                } else {
                        BT_DBG("SET_EVENT_FLT succeseful");
                }
                break;
        case OCF_WRITE_CA_TIMEOUT:
                if (status) {
                        BT_DBG("OCF_WRITE_CA_TIMEOUT failed %d", status);
                } else {
                        BT_DBG("OCF_WRITE_CA_TIMEOUT succeseful");
                }
                break;

        case OCF_WRITE_PG_TIMEOUT:
                if (status) {
                        BT_DBG("OCF_WRITE_PG_TIMEOUT failed %d",  status);
                } else {
                        BT_DBG("OCF_WRITE_PG_TIMEOUT succeseful");
                }
                break;
        case OCF_WRITE_SCAN_ENABLE:
                if (!status) {
                      BT_DBG("Scan and inquiry enabled");
                }

        }
}

/* Command Complete OGF INFO_PARAM  */
static void billotooth_cc_info_param(__u16 ocf, struct sk_buff *skb)
{
        read_local_features_rp *lf;
        read_buffer_size_rp *bs;
        read_bd_addr_rp *ba;

        switch (ocf) {
        case OCF_READ_LOCAL_FEATURES:
                lf = (read_local_features_rp *) skb->data;
```

```
                if (lf->status) {
                        BT_DBG("READ_LOCAL_FEATURES failed %d", lf->status);
                        break;
                }

                memcpy(mydev->features, lf->features, sizeof(mydev->features));

                /* Adjust default settings according to features
                 * supported by device. */
                if (mydev->features[0] & LMP_3SLOT)
                        mydev->pkt_type |= (HCI_DM3 | HCI_DH3);

                if (mydev->features[0] & LMP_5SLOT)
                        mydev->pkt_type |= (HCI_DM5 | HCI_DH5);

                if (mydev->features[1] & LMP_HV2)
                      mydev->pkt_type |= (HCI_HV2);

                if (mydev->features[1] & LMP_HV3)
                        mydev->pkt_type |= (HCI_HV3);

                BT_DBG("Device features 0x%x 0x%x 0x%x", lf->features[0], lf->features[1], lf-
>features[2]);
                break;

        case OCF_READ_BUFFER_SIZE:
                bs = (read_buffer_size_rp *) skb->data;

                if (bs->status) {
                        BT_DBG("READ_BUFFER_SIZE failed %d", bs->status);
                        break;
                }

                mydev->acl_mtu  = __le16_to_cpu(bs->acl_mtu);
                mydev->sco_mtu  = bs->sco_mtu ? bs->sco_mtu : 64;
                mydev->acl_pkts = mydev->acl_cnt = __le16_to_cpu(bs->acl_max_pkt);
                mydev->sco_pkts = mydev->sco_cnt = __le16_to_cpu(bs->sco_max_pkt);

                BT_DBG("Buffer sizes. mtu: acl %d, sco %d max_pkt: acl %d, sco %d",
                    mydev->acl_mtu, mydev->sco_mtu, mydev->acl_pkts, mydev->sco_pkts);
                break;

        case OCF_READ_BD_ADDR:
                ba = (read_bd_addr_rp *) skb->data;

                if (!ba->status) {
                        bacpy(&mydev->bdaddr, &ba->bdaddr);
                } else {
                        BT_DBG("READ_BD_ADDR failed %d",ba->status);
                }

                break;

        };
}

/* Inquiry Result */
static inline void billotooth_inquiry_result_evt(struct sk_buff *skb)
{
        inquiry_info *info = (inquiry_info *) (skb->data + 1);
        int num_rsp = *((__u8 *) skb->data);

        BT_DBG("%d detected device(s)", num_rsp);
        for (; num_rsp; num_rsp--)
                billotooth_inquiry_list_update(info++);
}

/* Connect Request */
static inline void billotooth_conn_request_evt(struct sk_buff *skb)
{
        evt_conn_request *cr = (evt_conn_request *) skb->data;
        struct hci_conn *conn;
        accept_conn_req_cp ac;
```

```
        BT_DBG("Connection request: %s type 0x%x",
                        batostr(&cr->bdaddr), cr->link_type);

       /* Connection accepted */

        conn = conn_hash_lookup_ba(mydev, cr->link_type, &cr->bdaddr);
        if (!conn) {
           if (!(conn = billotooth_create_connection(&cr->bdaddr))) {
                 BT_ERR("No memmory for new connection");
                 return;
           }
        }
        conn->state = BT_CONNECT;
        bacpy(&ac.bdaddr, &cr->bdaddr);
        ac.role = 0x01; /* Remain slave */

        billotooth_send_cmd(OGF_LINK_CTL, OCF_ACCEPT_CONN_REQ,
                            ACCEPT_CONN_REQ_CP_SIZE, &ac);
}


/* Event Handling */
static void billotooth_event_packet(struct sk_buff *skb)
{
        hci_event_hdr *he = (hci_event_hdr *) skb->data;
        //evt_cmd_status *cs;
        evt_cmd_complete *ec;
        __u16 opcode, ocf, ogf;

        skb_pull(skb, HCI_EVENT_HDR_SIZE);

        BT_DBG("%s evt 0x%x", mydev->name, he->evt);

        switch (he->evt) {
        case EVT_INQUIRY_RESULT:
                billotooth_inquiry_result_evt(skb);
                break;

        case EVT_CONN_REQUEST:
                billotooth_conn_request_evt(skb);
                break;

       case EVT_CONN_COMPLETE:
                billotooth_conn_complete_evt(skb);
                break;

       case EVT_DISCONN_COMPLETE:
                hci_disconn_complete_evt(skb);
                break;

        case EVT_CMD_COMPLETE:
                ec = (evt_cmd_complete *) skb->data;
                skb_pull(skb, EVT_CMD_COMPLETE_SIZE);

                opcode = __le16_to_cpu(ec->opcode);
                ogf = cmd_opcode_ogf(opcode);
                ocf = cmd_opcode_ocf(opcode);

                switch (ogf) {
                case OGF_INFO_PARAM:
                        billotooth_cc_info_param(ocf, skb);
                        break;

                case OGF_HOST_CTL:
                        billotooth_cc_host_ctl(ocf, skb);
                        break;

               default:
                       BT_ERR("EVT_CMD_COMPLETE not handled");
                      break;
               };
                break;
```

```
        default:
                BT_ERR("Event not handled");
                break;
        };
}

/*---------------------------------------------------------------------------------------------
                                        INTERFACE TO UPPER LAYERS
---------------------------------------------------------------------------------------------*/

int billotooth_register_mod(char * desc, int * id, int (*preceptor)(char * data))
{
        BT_DBG("Registering module: %s",desc);
        list_receptors[num_receptors] = preceptor;
        num_receptors++;
        billotooth_inq_req();
        return 0;
}

int billotooth_unregister_mod(void)
{
        BT_DBG("Unregistering module");
        //TODO: reorganize list order
        num_receptors--;
        return 0;
}

/*---------------------------------------------------------------------------------------------
                                        INITIALIZATION ROUTINES
---------------------------------------------------------------------------------------------*/

/* Open device. When interface this is called when the hci interface goes up */
int billotooth_dev_open(void)
{
        int ret = 0;
        set_event_flt_cp ef;
        __u16 param;

        /* Verify if interface is already open */
        if (test_bit(HCI_UP, &mydev->flags)) {
                ret = -EALREADY;
                goto done;
        }
        /* Lower layer opening */
        if (mydev->open(mydev)) {
                ret = -EIO;
                goto done;
        }

        /* Now, the interface is up */
        set_bit(HCI_UP, &mydev->flags);


        /* Mandatory initialization */

        /* Read Local Supported Features */
        billotooth_send_cmd(OGF_INFO_PARAM, OCF_READ_LOCAL_FEATURES, 0, NULL);

        /* Read Buffer Size (ACL mtu, max pkt, etc.) */
        billotooth_send_cmd(OGF_INFO_PARAM, OCF_READ_BUFFER_SIZE, 0, NULL);

#if 0
        /* Host buffer size */
        {
                host_buffer_size_cp bs;
                bs.acl_mtu = __cpu_to_le16(HCI_MAX_ACL_SIZE);
                bs.sco_mtu = HCI_MAX_SCO_SIZE;
                bs.acl_max_pkt = __cpu_to_le16(0xffff);
                bs.sco_max_pkt = __cpu_to_le16(0xffff);
                billotooth_send_cmd(OGF_HOST_CTL, OCF_HOST_BUFFER_SIZE,
                                HOST_BUFFER_SIZE_CP_SIZE, &bs);
        }
```

```
#endif

        /* Read BD Address */
        billotooth_send_cmd(OGF_INFO_PARAM, OCF_READ_BD_ADDR, 0, NULL);

        /* Optional initialization */

        /* Clear Event Filters */
        ef.flt_type  = FLT_CLEAR_ALL;
        billotooth_send_cmd(OGF_HOST_CTL, OCF_SET_EVENT_FLT, 1, &ef);

        /* Page timeout ~20 secs */
        param = __cpu_to_le16(0x8000);
        billotooth_send_cmd(OGF_HOST_CTL, OCF_WRITE_PG_TIMEOUT, 2, &param);

        /* Connection accept timeout ~20 secs */
        param = __cpu_to_le16(0x7d00);
        billotooth_send_cmd(OGF_HOST_CTL, OCF_WRITE_CA_TIMEOUT, 2, &param);

        /* Inquiry and Page scans */
        param = 3;
        billotooth_send_cmd(OGF_HOST_CTL, OCF_WRITE_SCAN_ENABLE, 1, &param);

        /* Initializes inquiry device list */
        inq_cache.list = NULL;

        /* Init list of receptors */
        num_receptors = 0;

        /* Inquiry devices nearby */
        billotooth_inq_req();

done:
        return ret;
}


/* Register HCI device  - Interface to the lower stack layer */
int billotooth_register_dev(struct hci_dev *hdev)
{
        int id = 0;
        BT_DBG("Registering device");

        /* Check for lower layer routines presence */
        if (!hdev->open || !hdev->close || !hdev->destruct)
                return -EINVAL;

        /* Set device parameters */
        sprintf(hdev->name, "hci%d",id);
        hdev->id = id;
        atomic_set(&hdev->refcnt, 1);
        hdev->flags = 0;
        hdev->pkt_type  = (HCI_DM1 | HCI_DH1 | HCI_HV1);
        hdev->link_mode = (HCI_LM_ACCEPT);
        memset(&hdev->stat, 0, sizeof(struct hci_dev_stats));
        atomic_set(&hdev->promisc, 0);

        MOD_INC_USE_COUNT;
        mydev = hdev;
        billotooth_dev_open();
        conn_hash_init(mydev);
        return id;
}


/* Unregister HCI device  - Interface to the lower stack layer */
int billotooth_unregister_dev(struct hci_dev *hdev)
{
        BT_DBG("Unregistering device");
        MOD_DEC_USE_COUNT;
        return 0;
}
```

```
int billotooth_init(void)
{
        BT_DBG("*** BILLOTOOH PROTOCOL STACK IS RUNNING ***");
        return 0;
}

void billotooth_cleanup(void)
{
       BT_DBG("Performing cleanup");
        billotooth_inquiry_list_destroy();
}


module_init(billotooth_init);
module_exit(billotooth_cleanup);
```

ANEXO 3 – Código do módulo para controle centralizado

```
#include <linux/config.h>
#include <linux/module.h>

#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/types.h>
#include <linux/interrupt.h>

#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/skbuff.h>
#include <linux/kmod.h>
#include <sys/io.h>

#include "Bluetooth.h"
#include "hci_core.h"

int id;

#define BT_ENABLE_APP 1

#ifdef BT_ENABLE_APP
#define BT_APP(fmt, arg...)  printk(fmt , ## arg)
#else
#define BT_APP(D...)
#endif

#define REMOTE_ADDRESS {{0x88, 0x92, 0x09, 0x57, 0x60, 0x00}}
#define MOD_NAME "Mod_automatic"
#define MAJOR 155
#define MAX_NUM_STATES 10
#define MAX_NUM_TRANSITIONS 10

typedef struct {
        char inevent;
        char outevent;
       void (*handle) (void);
        int dest_state;
} ttransition;

typedef struct {
        ttransition transition[MAX_NUM_TRANSITIONS];
        int num_transitions;
        char desc[30];
} tstates;

struct hci_conn * conn;
int current_state = 0;
tstates states[MAX_NUM_STATES];

struct hci_conn * conn; //as a prototype, it first works with a single connection
static int mod_send(char * data);
static int define_protocols(void);
static ssize_t mod_write(struct file *pFile, const char *pData, size_t count, loff_t *off);
static struct file_operations mod_fops = {
       write: mod_write,
};

char *batostr(bdaddr_t *ba)
{
        static char str[2][18];
        static int i = 1;

        i ^= 1;
        sprintf(str[i], "%2.2X:%2.2X:%2.2X:%2.2X:%2.2X:%2.2X",
```

```
                ba->b[0], ba->b[1], ba->b[2],
                ba->b[3], ba->b[4], ba->b[5]);

        return str[i];
}

int handle_event(char inevent) {
       int i;
        char outdata[10];
        for (i=0;i<states[current_state].num_transitions;i++) {
          if (states[current_state].transition[i].inevent == inevent) {
            if (states[current_state].transition[i].handle != 0) {
               states[current_state].transition[i].handle(); //call handle function
            }
            if (states[current_state].transition[i].outevent != 0) {
                outdata[0] = states[current_state].transition[i].outevent;
                outdata[1] = 0;
               mod_send(outdata);
            }
            BT_MODAUTOMATIC("Estado Atual: %d Evento entrada: %c ",current_state,inevent);
            current_state = states[current_state].transition[i].dest_state;
            BT_MODAUTOMATIC("Estado Posterior %d Funcao/Evento saida:
%p/%c\n",current_state,states[current_state].transition[i].handle,outdata[0]);
            return 0; //exits function
          }
        }
       return 0;
}


int mod_recv(char * data)
{
        BT_DBG("Module %d received frame. Data[0] = %c",id,data[0]);
        handle_event(data[0]);
       return 0;
}

static int mod_send(char * data)
{
        billotooth_send_acl(conn, data,1);
        return 0;
}


static ssize_t mod_write(struct file *pFile, const char *pData, size_t count, loff_t *off)
{
        BT_DBG("Writing %c, %d bytes\n", pData[0], count );

        handle_event(pData[0]);
        return count;
}

int mod_automatic_init(void)
{
        int err=0;
        char desc[100] = "*** MODULO BLUETOOTH: CONTROLE AUTOMATICO DISPOSITIVOS ELETRONICOS
***";
       bdaddr_t address = (bdaddr_t) REMOTE_ADDRESS;

         BT_DBG("%s",desc);
        billotooth_register_mod(desc,&id,mod_recv);
        conn = billotooth_connect(&address);
        register_chrdev(MAJOR,MOD_NAME,&mod_fops);
        define_protocols();
        return err;
}

void mod_automatic_cleanup(void)
{


        billotooth_acl_disconnect(conn,0x13);
       unregister_chrdev(MAJOR,MOD_NAME);
       billotooth_unregister_mod();
```

```
}

module_init(mod_automatic_init);
module_exit(mod_automatic_cleanup);
```

ANEXO 4 – Código da aplicação (no computador central)

```
void mostraLigado(void) {
        char str[100] = "Ar condicionado esta ligado";
        BT_APP("%s\n",str);
}

void mostraDesligado(void) {
        char str[100] = "Ar condicionado esta desligado";
        BT_APP("%s\n",str);
}


void ligacaoRealizada(void) {
        char str[100] = "Ar condicionado foi ligado";
        BT_APP("%s\n",str);
}

void desligamentoRealizado(void) {
        char str[100] = "Ar condicionado foi desligado";
        BT_APP("%s\n",str);
}

void ausenciaDetectada(void) {
        char str[100] = "*** RECINTO DESOCUPADO ***";
        BT_APP("%s\n",str);
}

void presencaDetectada(void) {
        char str[100] = "*** PRESENCA DETECTADA ***";
        BT_APP("%s\n",str);
}

//As a testing code, 2 applications will be designed in the code below... To simplify, they
use the same state// machine
static int define_protocols(void)
{
        /*
        CONTROLE AR CONDICIONADO
        Eventos:
        A- Usuario solicita status ar condicionado
        B- Usuario solicita que ar condicionado seja ligado
        C- Usuario solicita que ar condicionado seja desligado
        D- Ar condicionado esta ligado
        E- Ar condicionado esta desligado
        F- Reconhecimento de que ar condicionado foi ligado
        G- Reconhecimento de que ar condicionado foi desligado

        CONTROLE VIGILANCIA
        Eventos:
        M- Presenca de pessoa detectada
        N- Ausencia de pessoa detectada
        */

        //IDLE
        states[0].transition[0].inevent = 'A';
        states[0].transition[0].outevent = 'A';
        states[0].transition[0].handle = 0;
        states[0].transition[0].dest_state = 1;
        states[0].transition[1].inevent = 'B';
        states[0].transition[1].outevent = 'B';
        states[0].transition[1].handle = 0;
        states[0].transition[1].dest_state = 2;
        states[0].transition[2].inevent = 'C';
        states[0].transition[2].outevent = 'C';
        states[0].transition[2].handle = 0;
        states[0].transition[2].dest_state = 2;
        states[0].transition[3].inevent = 'M';
        states[0].transition[3].outevent = 0;
        states[0].transition[3].handle = presencaDetectada;
        states[0].transition[3].dest_state = 0;
```

```
        states[0].transition[4].inevent = 'N';
        states[0].transition[4].outevent = 0;
        states[0].transition[4].handle = ausenciaDetectada;
        states[0].transition[4].dest_state = 0;
        states[0].num_transitions = 5;


       //WAIT STATUS
        states[1].transition[0].inevent = 'D';
        states[1].transition[0].outevent = 0;
        states[1].transition[0].handle = mostraLigado;
        states[1].transition[0].dest_state = 0;
        states[1].transition[1].inevent = 'E';
        states[1].transition[1].outevent = 0;
        states[1].transition[1].handle = mostraDesligado;
        states[1].transition[1].dest_state = 0;
        states[1].transition[2].inevent = 'M';
        states[1].transition[2].outevent = 0;
        states[1].transition[2].handle = presencaDetectada;
        states[1].transition[2].dest_state = 1;
        states[1].transition[3].inevent = 'N';
        states[1].transition[3].outevent = 0;
        states[1].transition[3].handle = ausenciaDetectada;
        states[1].transition[3].dest_state = 1;
        states[1].num_transitions = 4;

        //WAIT ACK
        states[2].transition[0].inevent = 'F';
        states[2].transition[0].outevent = 0;
        states[2].transition[0].handle = ligacaoRealizada;
        states[2].transition[0].dest_state = 0;
        states[2].transition[1].inevent = 'G';
        states[2].transition[1].outevent = 0;
        states[2].transition[1].handle = desligamentoRealizado;
        states[2].transition[1].dest_state = 0;
        states[2].transition[2].inevent = 'M';
        states[2].transition[2].outevent = 0;
        states[2].transition[2].handle = presencaDetectada;
        states[2].transition[2].dest_state = 2;
        states[2].transition[3].inevent = 'N';
        states[2].transition[3].outevent = 0;
        states[2].transition[3].handle = ausenciaDetectada;
        states[2].transition[3].dest_state = 2;
        states[2].num_transitions = 4;

       current_state = 0;
       return 0;
}
```