TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UMA INTERFACE USB PARA AQUISIÇÃO DE DADOS DE UM MEDIDOR DE CONSUMO DE ENERGIA ELÉTRICA

Andre Cataldo Sterf
Gabriel Queiroz Silva

Brasília, julho de 2008

UNIVERSIDADE DE BRASILIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UMA INTERFACE USB PARA AQUISIÇÃO DE DADOS DE UM MEDIDOR DE CONSUMO DE ENERGIA ELÉTRICA

Andre Cataldo Sterf
Gabriel Queiroz Silva

Relatório submetido ao Departamento de Engenharia Elétrica da Faculdade de Tecnologia da Universidade de Brasília como requisito parcial para obtenção do grau de Engenheiro Eletricista

Banca Examinadora

Prof. Ricardo Zelenovsky, Doutor, UnB/ENE (Orientador)	_____
Eng°. Marcelo Alejandro Villegas Sanchez (Examinador Interno)	_____
Eng°. Giuler Alberto Cruz Silva (Examinador Interno)	_____

FICHA CATALOGRÁFICA

SILVA, GABRIEL QUEIROZ
STERF, ANDRE CATALDO

Desenvolvimento de uma interface USB para aquisição de dados de um medidor de consumo de energia elétrica

Trabalho de Conclusão de Curso, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 59p.

1. USB
2. Firmware
3. Medição de energia
4. ARM

REFERÊNCIA BIBLIOGRÁFICA

SILVA, GABRIEL QUEIROZ; STERF, ANDRE CATALDO (2008). Desenvolvimento de uma interface USB para aquisição de dados de um medidor de consumo de energia elétrica.

Trabalho de Conclusão de Curso, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 59p.

CESSÃO DE DIREITOS

AUTORES: Gabriel Queiroz Silva e Andre Cataldo Sterf

ORIENTADOR: Ricardo Zelenovsky

TÍTULO: Desenvolvimento de uma interface USB para aquisição de dados de um medidor de consumo de energia elétrica

ANO: 2008

Dedicatória

Dedico este trabalho à todos meus parentes e amigos que me incentivaram durante toda a realização do curso.

Gabriel Queiroz Silva

À minha família, amigos e pessoas queridas, Aline em especial, que estiveram presentes e foram fonte de estímulo e exemplo ao longo desta caminhada.

Andre Cataldo Sterf

SUMÁRIO

LISTA DE FIGURAS

INTRODUÇÃO

1.1 INTERFACE ENTRE O COMPUTADOR E O MEDIDOR DE ENERGIA

Esse projeto surgiu da idéia de desenvolver uma interface entre uma placa medidora de consumo elétrico e um computador de maneira a poder realizar o armazenamento das informações coletadas.

Estabelecido a idéia principal a ser desenvolvida, as premissas e características que serão adotadas visando à definição da arquitetura de *hardware* e *software* necessárias para a execução desse projeto e a sua implementação efetiva são o foco desse trabalho e serão abordadas nos próximos capítulos .

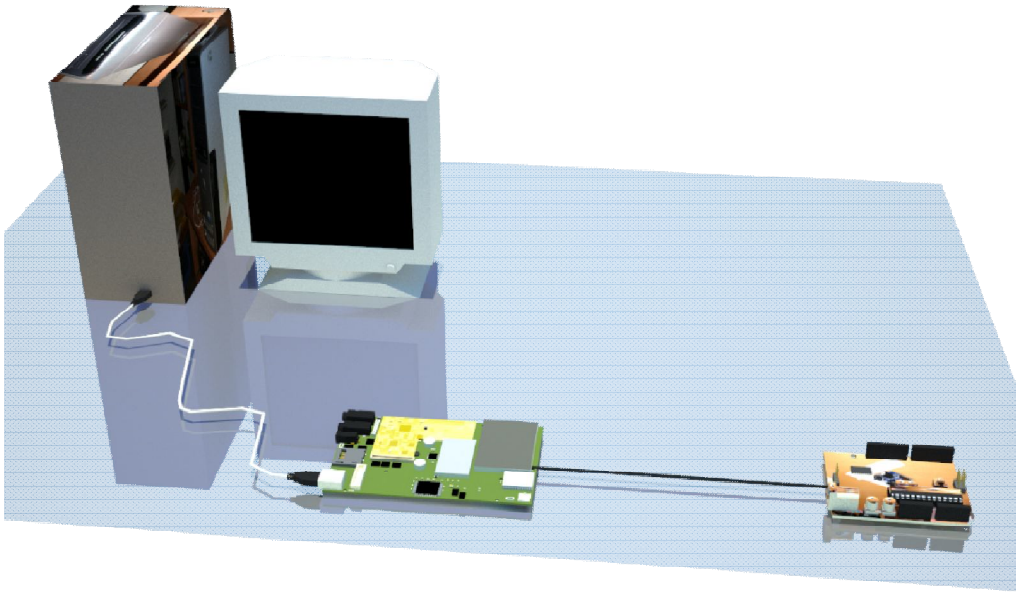

Figura 1 - Ilustração de uma arquitetura de comunicação entre um periférico e um computador

MEDIDOR DE ENERGIA

2.1 INTRODUÇÃO

O objetivo do projeto esta centrado na exibição do consumo elétrico de uma carga qualquer a partir dos dados aferidos por um equipamento com essa finalidade especifica, a placa MCP390X, fabricada pela Microchip. Essa placa foi adotada por permitir que a proposta seja cumprida efetivamente e por já estar disponível no ENE, uma vez que estava sendo utilizada em outro projeto semelhante.

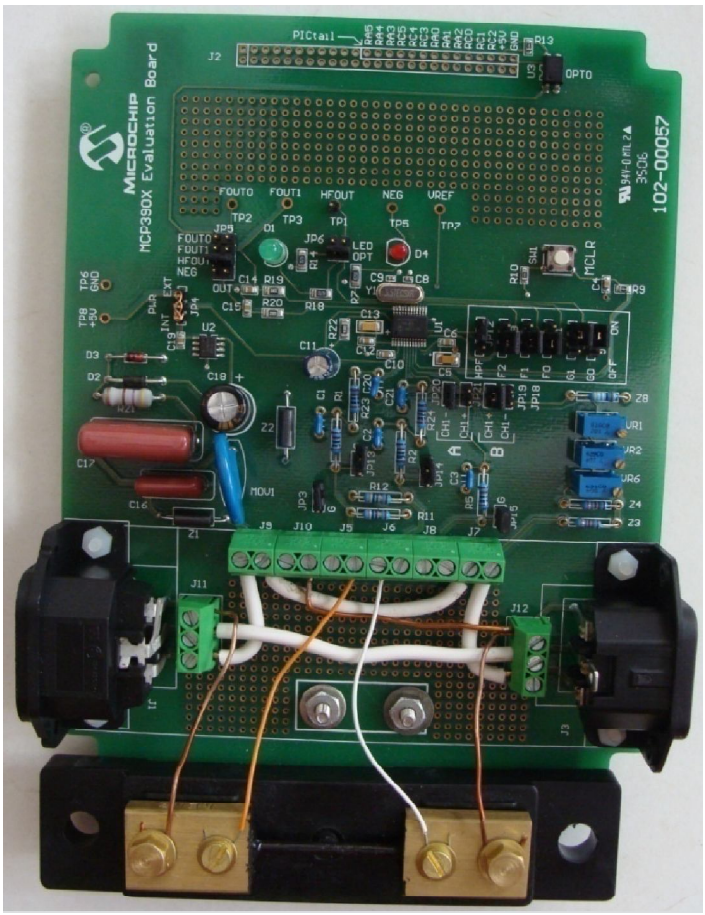

Figura 2 - Placa Medidora MCP390x

2.2 PRINCIPAIS CARACTERÍSTICAS

A placa medidora é controlada pelo circuito integrado que dá nome a esta linha de medidores, o MCP3905. Outros componentes importantes da placa são:

- Duas tomadas trifásicas 2P+T padrão NEMA, uma de entrada e uma de saída, para serem alimentadas com corrente alternada de alta voltagem
- Borders de conexão, que estão conectados aos pinos de entrada do CI e permitem realizar várias configurações de circuitos elétricos típicos de potência, além de servir para adicionar componentes medidores de corrente como um resistor shunt
- Jumpers e potenciômetro para configurar a resistência interna do circuito;
- Conversor AC-DC de baixo custo para alimentar o CI;
- Leds de verificação de funcionamento;
- Barramento de 14 pinos para a conexão de microcontroladores do tipo PIC, facilitando a criação de outros hardwares que utilizem as os dados coletados;

Além disso, a placa foi desenhada para caber no interior de caixas plásticas padrão para aplicações de alta-voltagem.

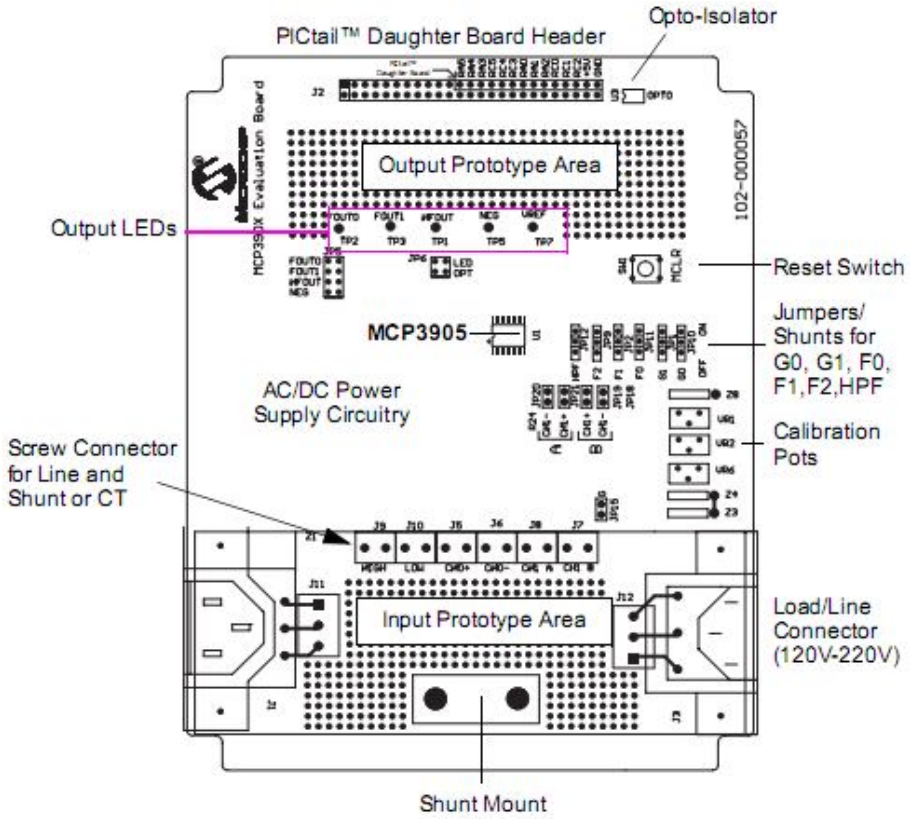

Figura 3 – *Layout* da Placa Medidora MCP3906

2.3 CI MCP3905

O CI MCP3906 é um encapsulamento de 24 pinos desenhado seguindo os padrões da IEC (*International Electrotechnical Commission*), que é uma organização internacional responsável por criar e publicar *International Standards* para tecnologias eletroeletrônicas. A especificação mais recente é a IEC 62053, que é plenamente cumprida, juntamente com as anteriores IEC 1036/61036/687, o que garante um funcionamento com um erro típico de 0,1%.

Internamente oferece:

- Dois conversores AD delta-sigma de 16 bits e dois conversores DA de segunda ordem;

- Amplificador de Ganho Programável (PGA) com relação de 32:1, permitindo correntes *shunt* de baixo valor;
- Saída direta para motores de passo de bifásicos;
- Saída HF_{OUT} com a potência instantânea ativa (real);

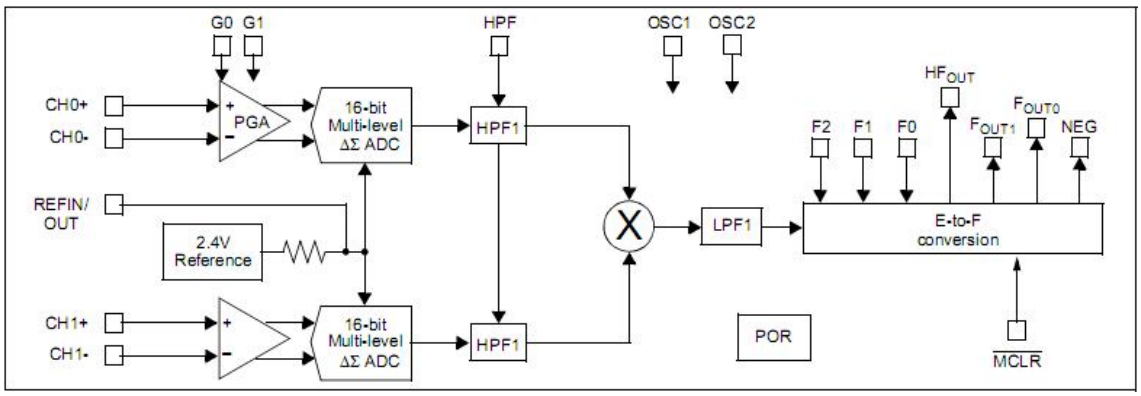

Figura 4 - Diagrama de blocos da funções do CI MCP3905

2.4 SINAL DE SAÍDA HF_{OUT}

Das três saídas oferecidas pelo equipamento, a que é mais adequada para nossa proposta é a HF_{OUT}, que representa a potência ativa verificada. Esse sinal é um pulso TTL de lógica positiva, com um valor mínimo quando ativo de 4 volts e máximo quando inativo de 0,5 volts. O período médio t_{HP} da onda quadrada são de 90 milisegundos enquanto que a freqüência varia proporcionalmente à energia consumida em um processo que será explicado mais à frente.

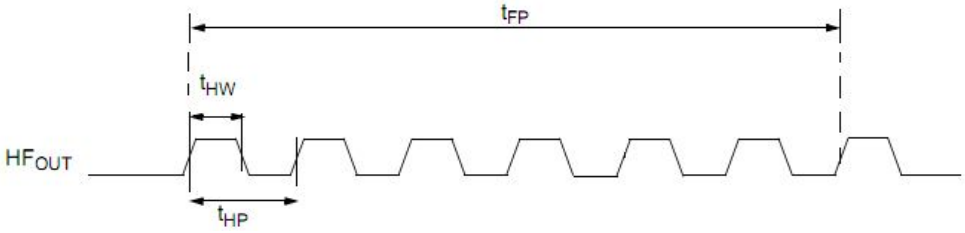

Figura 5 – Pulso TTL de resposta

2.5 FUNCIONAMENTO BÁSICO

Ao aplicar uma carga qualquer na saída do medidor, uma corrente flui pelos terminais da placa. Entre esses terminais, em série com a carga alimentada, temos duas resistências em paralelo: uma resistência de alta precisão e baixíssimo valor nominal, conhecida como *resistor shunt* e que chamaremos de R_S; e a resistência interna do sistema, que chamaremos de R_M. A condição básica para utilizarmos a placa MCP390X é que o *resistor shunt* deve ser calculado para que $R_S << R_M$,normalmente na ordem de 10^{-6} Ω << Ω, de tal maneira que a corrente I_S que fluir por essa resistência represente quase que a totalidade da corrente exigida pela carga. Tem-se então uma corrente I_M mínima fluindo para dentro do medidor, com amplitude apropriada para circuitos eletrônicos de encapsulamento.

De maneira simplista, I_M é verificada pelo canal CH0 do CI MCP3905, amplificada pelo PGA, utilizada pelo conversor AD e este sinal é multiplicado por outro oriundo de um processo semelhante sofrido pela corrente que é obtida da resistência da ponte de resistores, verificada no CH1. O sinal resultante é utilizado para regular o disparo dos pulsos de saída do sistema. Temos então que o pulso é uma resposta direta à potência utilizada pela carga, mas exige uma calibração prévia para determinar essa relação.

Desta forma um ensaio utilizando uma tensão de entrada igual à dos sistemas elétricos a serem medidos no futuro e com uma carga consumindo uma potência ativa já conhecida nos permite utilizar os *jumpers* e o potenciômetro para variar a resistência interna e calibrar o sistema, estabelecendo relação de potência consumida/pulso emitido, que por sua vez irá determinar a freqüência, por vezes variável, do sinal HF_{OUT}. Uma vez estabelecida esta relação é possível programar corretamente a plotagem do gráfico para exibir o valor de energia consumida ao longo do tempo.

KIT DE DESENVOLVIMENTO ARM

3.1 INTRODUÇÃO

O *hardware* adotado para realizar a interface física e lógica entre o medidor de energia e o computador é o kit de desenvolvimento AT91SAM7S256-EK, fabricado pela Atmel. A sua adoção se justifica, em primeiro lugar, porque, por se tratar de um kit de desenvolvimento, é indicado para aplicações gerais que necessitem da implementação simulada de um *hardware* dedicado baseado em microcontroladores. Em segundo lugar, se justifica por ser adequado para projetos que necessitem de desenvolvimento de *software* embarcado, uma vez que oferece funcionalidades como programação e *debugging* de *firmware* por meio da porta paralela do PC (JTAG) e uma grande memória interna. E em terceiro lugar esse kit possui variadas interfaces de entrada e saída de sinais o que confere uma grande flexibilidade ao projeto e que serão abordadas no próximo tópico.

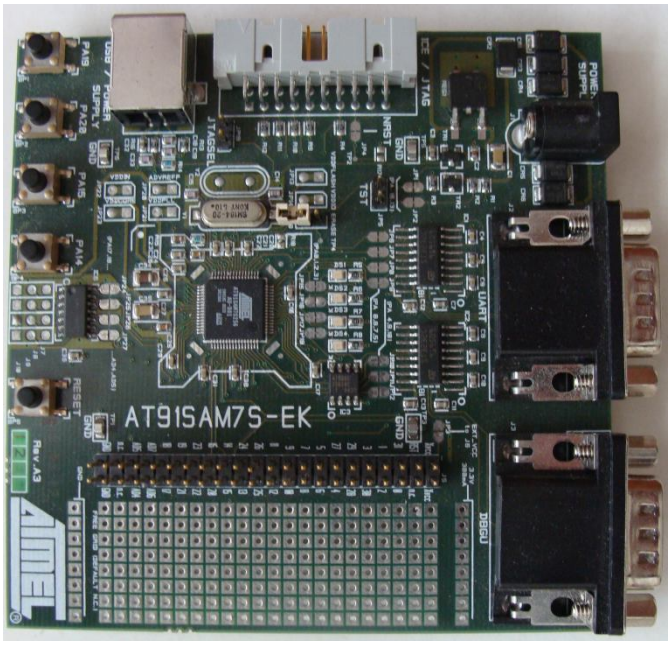

Figura 6 – Kit de desenvolvimento AT91SAM7S256-EK

3.2 µCONTROLADOR AT91SAM7S

O funcionamento do kit de desenvolvimento está centrado no microcontrolador AT91SAM7S, da família de processadores ARM7, e com arquitetura RISC de 32 bits. Essa linha é capaz de realizar até 130 MIPS, enquanto que o modelo específico é capaz de trabalhar até 0,9 MIPS/MHz. Com duas fontes de clock independentes, uma em um circuito RC operando entre 3 e 20 MHz, e outra em um circuito PLL funcionando em até 55 MHz, temos uma capacidade máxima de até 49,5 MIPS.

3.3 PERIFÉRICOS

Os principais componentes que complementam o funcionamento do microcontrolador são:

- 256 Kbytes de memória FLASH
- 64 Kbytes de memória RAM
- Conversor ADC de 8 canais
- Barramento paralelo (*PIOA*)
- *USB Device Port* (*UDP*)
- Duas portas *USART* com taxas independentes (*IrDA* e *full Modem line*)
- Interface Serial para Periféricos Mestre/Escrava (*SPI*)
- Saída Serial Síncrona (*SSC*)

- Conversor ADC de 8 canais com resolução de 10 bits
- Suporte a *ICE JTAG* para *debugging* pela porta paralela do PC

No nosso projeto dois periféricos merecem uma maior atenção:

- Barramento paralelo *PIOA*: possui 32 portas I/O com tolerância de 5V e com resistores *pull-up* programáveis individualmente. O controlador *PIO* é responsável pela monitoração do estado lógico dos pinos.
- *USB Device Port*: este módulo controla a conexão USB e é compatível com o padrão 2.0 *full-speed*, transferindo dados até 12 Mbps.

INTERFACE USB

4.1 INTRODUÇÃO

O padrão USB (*Universal Serial Bus*) foi desenvolvido por um consórcio de empresas, destaque para a Microsoft, Apple, Hewlett-Packard, NEC, Intel e Agere. O objetivo comum era criar um *industry-standard* com foco em permitir a conexão de periféricos aos PCs, baseando-se em um modelo único de conector e em melhorias das capacidades de *plug-and-play* destes equipamentos, permitindo a conexão e desconexão sem a necessidade de desligar ou reiniciar o computador.

Em sua versão 2.0, lançada em abril do ano 2000 sob o modelo EHCI (*Enhanced Host Controller Interface*), alguns dos critérios adotados na definição da sua arquitetura foram os seguintes:

- Fácil expansão de periféricos do PC
- Solução de baixo custo que suporte taxas de transferência de até 480 Mb/s
- Suporte em tempo real de dados para voz, áudio e vídeo
- Compatibilidade com várias configurações de PCs
- Possibilidade de criação de novas classes de periféricos
- Compatibilidade completa com equipamentos de versões anteriores

Assim, surgiu um padrão que permite ao SO e à placa-mãe diferenciar, transparentemente:

- A classe do equipamento (dispositivo de armazenamento, placa de rede, placa de som, etc);

- As necessidades de alimentação elétrica do dispositivo, caso este não disponha de alimentação própria;
- As necessidades de largura de banda (para um dispositivo de vídeo, serão muito superiores às de um teclado, por exemplo);
- As necessidades de latência máxima;
- Eventuais modos de operação internos ao dispositivo (por exemplo, máquina digital pode operar, geralmente, como uma *webcam* ou como um dispositivo de armazenamento - para transferir as imagens).

Figura 7 – Logomarca do padrão USB 2.0

Observa-se que os requisitos foram muitos na busca de satisfazer uma série de necessidades, o que acabou por gerar uma grande e vasta documentação. A Especificação 2.0, por exemplo, possui 650 páginas em sua versão original, além dos anexos definem outros componentes do padrão. Uma vez que o foco do projeto é utilizar essa tecnologia e dada à impossibilidade de explicá-la, nos capítulos posteriores estaremos conceituando os aspectos realmente importantes e que foram foco de nossa atenção durante o desenvolvimento da idéia. Deve ficar claro que vários embasamentos fundamentais do padrão não serão abordados neste leitura, mas são parte importante do conhecimento para sua manipulação.

4.2 TOPOLOGIA USB

4.2.1 FÍSICA

A arquitetura típica da conexão USB se baseia em um *host* (hospedeiro), nos equipamentos conectados a este, e na interconexão entre eles. Esta última estabelece como os periféricos se conectam e comunicam com o *host*, definindo a topologia de barramento, as relações intercamadas, os modelos de transferência de dados e o agendamento de transferências.

Por meio de uma relação de barramento mestre-escravo, as conexões estão orientadas em uma topologia estrela centrada no *host*, onde este recebe e controla todo o tráfego na interface. Tal organização exige a presença de um *root* hub junto ao *host*, que com o cascateamento de até outros 5 hubs em série, pode ser expandida a uma quantidade de periféricos até um total de 127 unidades por porta/*host*.

No caso geral, o computador realiza o papel de hospedeiro, e o fato de este acumular todo o controle das conexões acaba por aumentar a complexidade demandada ao *software* neste lado do barramento, mas garante que apenas este irá iniciar uma conversação. Essa estrutura é proposital e foi pensada de maneira a facilitar o desenvolvimento e implementação de dispositivos periféricos escravos.

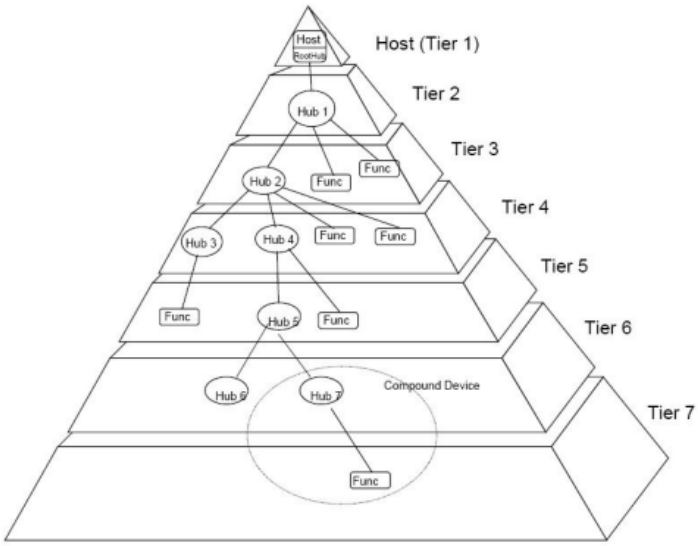

Figura 8 - Topologia do barramento USB

Até a sua última especificação estão definidas três velocidades de transferência de dados entre os dispositivos:

- *low-speed :* suporta até 1,5 Mbps (187,5 KBps), foi especificada na versão USB 1.0 e é utilizada por dispositivos que não requerem muita banda;
- *full-speed*: suporta até 12 Mbps (1,5 MBps) e foi especificada na versão USB 1.1;
- *high-speed*: suporta até 480 Mbps (60 MBps) e foi especificada na versão USB 2.0;

As últimas informações dão conta que a Intel prepara a especificação USB 3.0, prevista ainda para o ano de 2008, estabelece uma velocidade de transferência *super-speed*, com suporte para taxas de até 4,8 Gbps (600 MBps), utilizando um conexão de fibra ótica juntamente com o tradicional fio metálico de cobre.

4.2.2 LÓGICA

Enquanto hospedeiro, o computador é responsável por inicializar e operar as conexões com os dispositivos a ele plugados, função esta realizada pelo *software*, que quando executando essa função específica é mais comumente chamado de *driver*.

Como uma das premissas do padrão USB é permitir o *plug-and-play*, esse processo de inicialização é permanente e contínuo enquanto o *host* estiver ligado. Uma vez que um dispositivo é conectado ao hospedeiro, este inicia um processo de identificação do periférico conhecido como enumeração, que fornece um endereço e será tratado mais a frente.

A topologia lógica da conexão USB é relativamente simples:

- *Endpoint*: é o ponto de referência lógico de envio e recebimento de bits, um componente presente no periférico, basicamente um buffer de bytes múltiplos. Ele

possui um endereço que varia de 0 a 15 e pode ser de entrada ou saída (IN ou OUT), ou os dois simultaneamente para o caso do *control endpoint* (*endpoint* de controle), que necessariamente recebe o endereço 0;

- *Interface*: é o conjunto dos vários *endpoints* e podem existir várias simultâneas e concorrentes;
- *Pipe*: é a conexão lógica que associa um *device driver* no hospedeiro para cada interface, permitindo que um dispositivo possua várias em paralelo controladas individualmente;
- *Configuração*: é o conjunto de interfaces e apenas uma está ativa por vez;

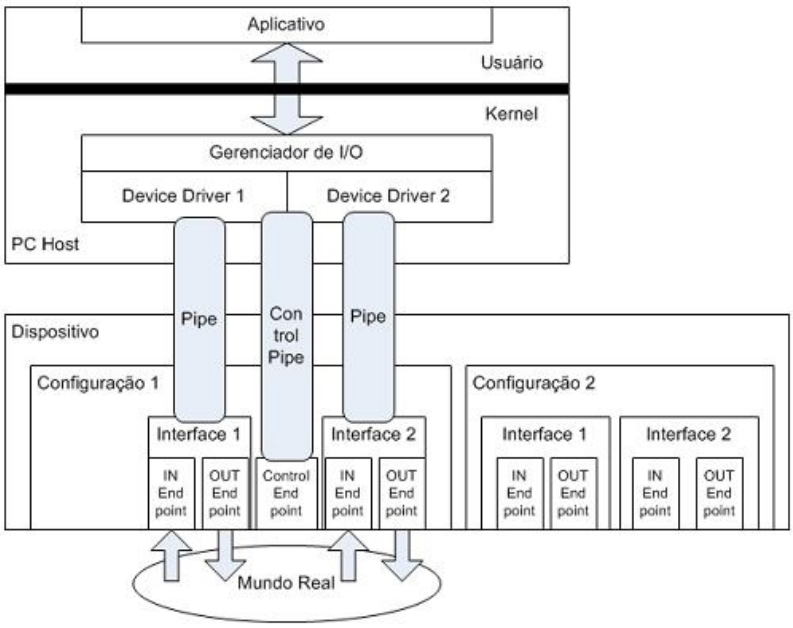

Figura 9 - Topologia do barramento USB

4.3 PROTOCOLO USB

A conexão USB foi estruturada para que, diferentemente de outros padrões de conexão serial, existem camadas de protocolo para garantir a eficiência da troca de informações. Cada comunicação se baseia em transações efetuadas entre o receptor e o transmissor e estas são compostas por pacotes, que por sua vez são formados por campos de bits seqüenciais. Veremos individualmente cada um desses componentes a seguir.

4.3.1 TIPOS DE CAMPOS DE PACOTES

Para facilitar o entendimento, analisamos primeiramente o nível lógico mais baixo do processo, que são os campos de pacote que efetivamente possuem os bits que formam um trem de informação, com os LSBs enviados primeiramente. Os mais comuns são:

- *SYNC*: sempre inicia um pacote, tem 8 bits de e tem como função sincronizar o *clock* entre o receptor e o transmissor do pacote;
- *PID*: abreviação de *Packet IDentificator*, indica o tipo de pacote enviado, é composto por 4 bits repetidos duas vezes para garantir o recebimento correto, criando uma palavra de 8 bits;

Group	PID Value	Packet Identifier
Token	0001	OUT Token
	1001	IN Token
	0101	SOF Token
	1101	SETUP Token
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	0110	NYET (No Response Yet)
Special	1100	PREamble
	1100	ERR
	1000	Split
	0100	Ping

Figura 10 – Definições de *PID*

- *ADDR*: especifica para qual dispositivo o pacote está endereçado, composto por 7 bits, o que limita em 127 os dispositivos controlado por um único *host*. O endereço 0000000 é reservado ao *host*;
- *ENDP*: determina o endereço do *endpoint* por meio de 4 bits;
- *CRC*: abreviação de *Cyclic Redundancy ChecksI*, responsável pela checagem de erros na transmissão, podendo ser de 5 ou 16 bits. Baseado no campo de dados, o transmissor envia um valor para ser comparado com o valor gerado pelo receptor, que só então aceita as informações em caso de igualdade;
- *EOP*: abreviação de *End Of Packet*, representado por dois *SE0* (Single *Ended Zero*);

4.3.2 CATEGORIA DE PACOTES

Em um nível lógico intermediário, temos os pacotes. Cada transação consiste na transmissão seqüencial de três categorias de pacotes:

- *Token*: sempre enviado pelo hospedeiro, define a transação que irá acontecer e o *endpoint* para qual está endereçada, podendo ser do tipo *IN*, que informa ao dispositivo USB que o hospedeiro deseja ler uma informação; do tipo *OUT*, que informa ao dispositivo USB que o hospedeiro deseja enviar uma informação; e do tipo *SETUP*, que inicia uma transferência do tipo *Control*;

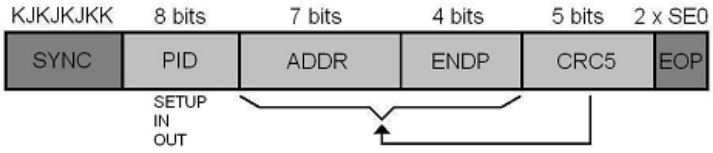

Figura 11 – Formato de um pacote *Token*

- *DATA*: posterior ao cabeçalho *token*, o pacote de dados carrega a informação que efetivamente está sendo transmitida pelo *host* ou pelo *device* e podem ser do tipo *DATA0*, *DATA1*, *DATA2* e *MDATA*. A quantidade de bits enviada varia de acordo com a velocidade da conexão, sendo que os tipos *DATA2* e *MDATA* são exclusivos para interfaces *high-speed*;

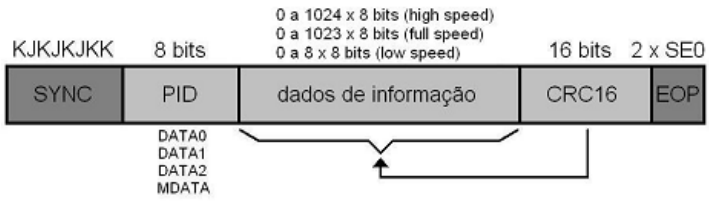

Figura 12 - Formato de um pacote DATA

- *Handshake*: são enviados na direção oposta ao último pacote *token* ou *data* enviado, e com isso sinalizam o correto ou incorreto recebimento pelo destinatário. Podem ser do tipo *ACK*, que confirma o recebimento; do tipo *NAK*, que indica a impossibilidade do receptor de enviar ou receber pacotes; do tipo *STALL*, que informa a ocorrência de algum erro no recebimento da informação ou o

recebimento de algum comando não suportado; e do tipo *NYET*, que é uma resposta negativa ao pacote *PING*;

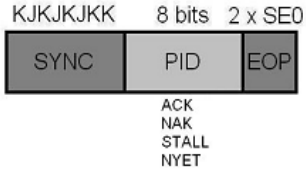

Figura 13 - Formato de um pacote HANDSHAKE

Temos ainda outros tipos de pacotes que ocorrem eventualmente ou que não fazem parte de uma transação, mas são importantes no protocolo:

- *Special*: os pacotes do tipo *PRE*, *SPLIT* e *ERR* servem para a comunicação entre o *root hub* e os *hubs* por ele controlados. O primeiro controla a velocidade da conexão, por exemplo, indicando um dispositivo *low-speed* conversando com um *host high-speed;* o seguinte reduz o armazenamento de dados pelos *hubs* e o último indica erros nessa operação. Ocorre também o pacote do tipo *PING*, já citado anteriormente, que é uma evolução da especificação 2.0 para conexões high-speed. É enviado pelo hospedeiro previamente ao envio de um pacote OUT e recebe como resposta um pacote *ACK* ou *NYET*, e por ser menor que o pacote que precede reduz a ocupação desnecessária do barramento no caso impossibilidade de recebimento;

- *SOF (Start of Frame)*: com a função específica de manter o sincronismo entre hospedeiro e os dispositivos a ele conectados, o *root hub* envia este pacote para todos, sem distinção de endereço, a cada 1 milisegundo. O intervalo entre dois desses pacotes é chamado de *frame*, e no caso de uma conexão *high speed* serão utilizados ainda oito *microframes* em intervalos de 125 microsegundos por meio de *SOFs* extras;

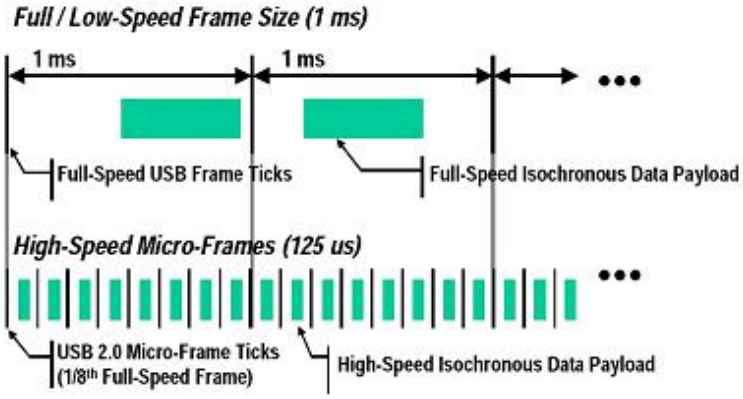

Figura 14 – Divisão de *frames* e *sub-frames*

O *frame* atual é definido pelo valor existente no campo de dados do pacote, que funciona como um contador com 11 bits e cujo valor é repetido pelos *micro-frames*. Esse valor zera após 2048 contagens, ou a cada 2048 milisegundos. O campo CRC é de 5 bits e os outros campos presentes são o de SYNC, PID e EOP.

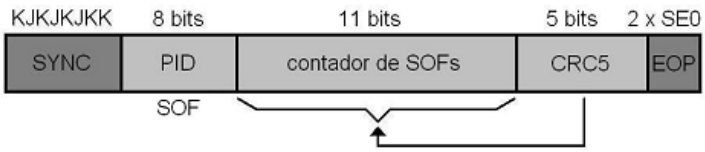

Figura 15 - Formato de um pacote *SOF*

4.3.3 TIPOS DE TRANSFERÊNCIAS

Por fim temos os processos de nível lógico macro, que se utilizam dos anteriores para implementar o protocolo de comunicação de maneira a atender as variadas possibilidades de aplicação proporcionada pela conexão USB.

As características básicas de uma transferência são ocorrer em até um *frame* e uma vez iniciada, não poder ser interrompida por outras transações. O seu fluxo básico é disparado pelo hospedeiro, que envia um pacote *token* com as definições da transação, seguido pela transferência de um pacote *data* com as características

especificadas anteriormente e finalizado com o envio do pacote de *handshake* na direção inversa que confirma o recebimento. Assim, temos quatro tipos de transações classificadas quanto ao tempo no qual deve ocorrer o envio/recebimento dos dados e na integridade com a qual eles devem ser entregues, descritas abaixo:

- *Control*: é a transação padrão para qualquer dispositivo USB e é fundamental para o projeto em tela, por isso será tratada em um tópico independente mais à frente;
- *Bulk*: adotada em situações onde existe o envio de grandes quantidades de dados com precisão, mas sem a necessidade de uma alta taxa de transferência dos dados, como em impressoras e *scanners*. Por essa característica é suportada apenas por conexões *full* ou *high-speed*;
- *Interrupt*: utilizada por dispositivos com necessidade de precisão e regularidade dos dados transmitidos, com a latência de envio definida, tal qual teclados e *mouses*, o que permite que seja suportada por todas as especificações USB;
- *Isochronous*: garante que os dados sejam enviados com uma taxa constante, eliminando a correção de erros, sendo utilizada em dispositivos que necessitam de dados em tempo real e que admitem erros eventuais, como aplicações de áudio e/ou vídeo. Suportada apenas pela conexões full e high speed;

4.3.4.1 TRANSAÇÃO *CONTROL* – SETUP, DATA E STATUS

A transação ou transferência do tipo *control* são sempre iniciadas pelo *host* sendo utilizada para funções de enumeração, obtenção de informações sobre os dispositivos conectados, definição do endereço, e qualquer outro tipo de transferência de dados, sendo suportada por todas as versões de conexão. São três os seus estágios:

- SETUP: é formado por três pacotes seqüenciais - um *Token* do tipo *SETUP*, que contém o endereço e o número do *endpoint* atribuído ao dispositivo que está se

conectando; um pacote *DATA* do tipo *DATA0* com os parâmetros *bmRequestType* (1 byte – define a direção da próxima transação, o tipo e o destinatário do *request*), *bRequest* (1 byte - define a informação solicitada pelo *host*), *wValue*, *wIndex* e *wLength* (2 bytes cada); por fim um pacote *Handshake*.

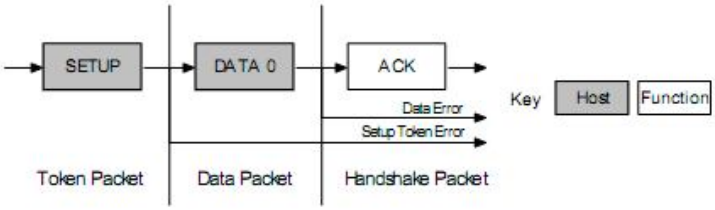

Figura 16 - Formato de um transação control de SETUP

- DATA: é um estágio opcional, composta por informações de entrada-leitura (IN) ou saída-escrita (OUT) cujo tamanho foi definido no estágio anterior. Quando está pronto para receber pacotes, o *host* envia um pacote *Token IN*, e caso a resposta do dispositivo coincida o *PID*, ele pode enviar um pacote DATA, STALL ou NAK. Quando precisa enviar um pacote de dados de controle para um dispositivo, o hospedeiro envia um pacote *Token OUT* seguido de um pacote DATA com as definições, que pode ser respondido com um ACK, NAK ou STALL.

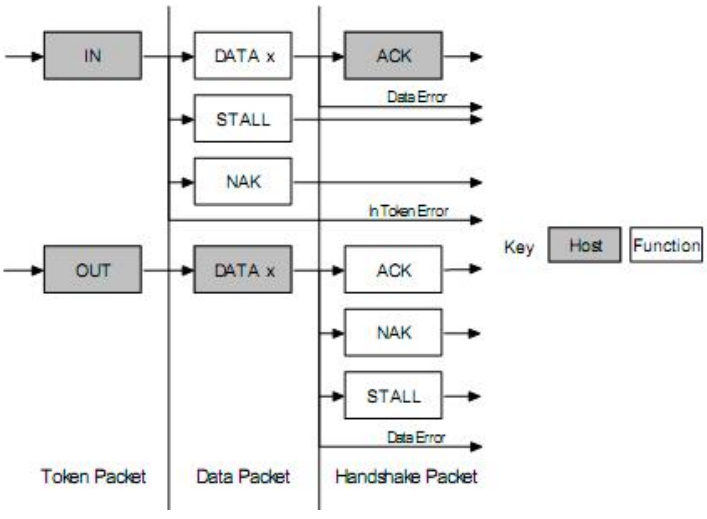

Figura 17 - Formato de um transação control de DATA

- STATUS: como o próprio nome remete, informa a situação das solicitações e por isso sempre é realizado pelo dispositivo, podendo ser de entrada (IN) ou saída (OUT), de acordo com a direção da transferência. No primeiro caso, se o host envia um Token IN durante o estágio de DATA solicitando uma informação, é necessário que ocorra a confirmação do correto recebimento destes dados. Assim, um Token OUT é enviado com pacote DATA vazio (comprimento 0) no caso positivo, um STALL é enviado para o caso de um erro de processamento do comando ou um NAK é respondido caso ainda esteja processando a informação anterior. Da mesma maneira ocorre para um Token OUT emitido pelo host, que é seguindo de um pacote DATA vazio e pode ser respondido como ACK, STALL ou NAK.

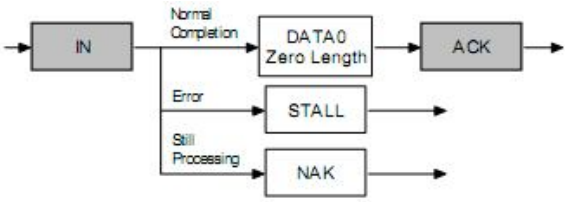

Figura 18 - Formato de um transação control de SETUP IN

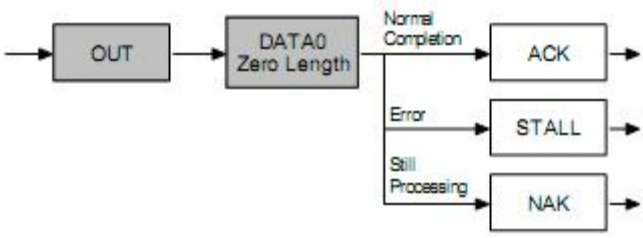

Figura 19 - Formato de um transação control de SETUP OUT

PROJETO

5.1 COMPONENTES BÁSICOS

Este projeto partiu de uma proposta inicial de medir o consumo de uma determinada carga e apresentar essas informações em um gráfico de consumo vs tempo. Com esta premissa definiu-se uma topologia de hardware que possibilitaria realizar a aferição analógica dos dados, digitalizá-los, enviá-los para um PC e por fim gerar o gráfico.

Uma vez definidos os componentes de *hardware* responsáveis pela aferição e aquisição das medições de consumo elétrico avaliou-se qual seria a melhor maneira de transmitir as informações coletadas para o PC. Uma vez que o kit de desenvolvimento AT91SAM7S256-EK possui um módulo de controle USB foi fácil optar por essa interface, visto que teríamos mais liberdade e seria menos trabalhoso o desenvolvimento dos *drivers* e outros requisitos necessários para a perfeita interação entre o kit de desenvolvimento e o computador .

Posterior à arquitetura física a ser adotada, foi necessário estabelecer como seriam tratadas essas informações no PC. Partindo da proposta final do projeto de apresentar os dados coletados em um gráfico, optou-se por adotar uma aplicação baseada na plataforma Java, visto que essa implementação exige um desenvolvimento simplificado e rápido, o que permite que modificações futuras sejam feitas sem grandes dificuldades. Além disso, trata-se de uma linguagem que possui um vasto suporte oferecido pela comunidade de desenvolvedores de software, o que facilita a execução fim-a-fim do projeto, como veremos mais a frente.

5.2 RECEBIMENTO DA SAÍDA DO MEDIDOR DE ENERGIA

Analisando o sinal fornecido pela saída HFOUT da placa MCP3905, foi observado que a calibração não havia sido realizada. Desta forma, construímos um circuito temporizador que simulasse a saída da placa. Na saída deste circuito temos um

pulso assíncrono com duração de aproximadamente 90ms e amplitude de no mínimo 4,0 Volts, aproximadamente o mesmo sinal que seria enviado pela placa medidora.

Utilizando a entrada paralela do ARM, associamos um pino específico ao sinal de medição do consumo do medidor. O firmware do controlador está constantemente verificando o nível lógico deste barramento em uma freqüência suficientemente alta, na casa dos MHz, de maneira que quando o sinal de entrada estiver em nível lógico positivo este será sempre percebido, visto que a freqüência máxima HFOUT seria de 11 Hz.

O medidor emite os pulsos em uma freqüência diretamente relacionada com a potência mensurada. Entretanto o valor de potência consumida necessário para a emissão do pulso deve ser previamente calibrado por meio de ensaios.

No código fonte do firmware temos o método `main()` que é chamado na inicialização do microcontrolador e será visto em detalhes mais a frente. Este método estabelece entre outras coisas, o pino que será responsável pela aquisição do sinal de saída do medidor de energia:

```
// Set in PIO mode and Configure in Input
AT91F_PIOA_CfgPMC();
AT91F_PIO_CfgInput(AT91C_BASE_PIOA, AT91C_PIO_PA24);
```

Temos pelo código acima que o pino do barramento PIOA do microcontrolador que será responsável pela aquisição é o número 24.

O próximo passo é fazer com que o dispositivo ARM esteja constantemente analisando esta entrada, de forma a capturar os pulsos enviados pelo MCP3905:

```
unsigned int actualInput = 0;
while (1){
  unsigned             int             inputMedidor             =
          (AT91F_PIO_GetInput(AT91C_BASE_PIOA)&AT91C_PIO_PA24);
  if (inputMedidor && !actualInput)
  {
    // sets actual input to true, so the pulse is not captured
twice
```

```
        actualInput = 1;

        // Sends in the report
        HID.SendReport(&HID, 1);
      }
      else
      {
        actualInput = 0;
      }
    }
```

O código acima cria um loop infinito e envia um *report* ao HUB USB toda vez que o sinal de saída do medidor varia logicamente de negativo para positivo. O *driver* que for controlar deverá ser programado de forma a reconhecer estes *reports* e disponibilizar como saída uma tabela de dados visando a interpretação da medição pelo aplicativo Java, permitindo a plotagem do gráfico.

5.3 INTERFACE USB-ARM COM O SO PARA ENVIO DOS DADOS

Considerando o baixo volume transmitido, a freqüência reduzida e a ausência da necessidade de uma alta precisão no envio dos dados, é possível aproveitar uma facilidade oferecida pelo padrão USB: utilizar a classe HID (*Human Interface Device*) como padrão para implementação da interface, já que esta foi especificada para variadas aplicações que não necessariamente interações humanas, suportando comunicação de dispositivos de baixa complexidade. Desta maneira é necessário programar o microcontrolador para que o mesmo funcione como um dispositivo USB-HID.

No caso do processador AT91 da Atmel, existe um *framework* desenvolvido pelo próprio fabricante para a implementação de dispositivos USB. Este *framework* já possui o código que inicializa o controlador, de forma que não precisamos nos preocupar com PLL e rotinas de tratamento de interrupção.

Inicialmente o dispositivo foi configurado para que funcionasse como um mouse, de forma que foi possível assegurar que a comunicação USB com o sistema operacional estava funcionando. Após conseguir isso, o código foi alterado para que o micro-controlador exercesse o interfaceamento USB com o PC.

O arquivo *main.c* é chamado na inicialização e inclui os seguintes arquivos do *framework*:

- *board.h*: inicialização do controlador, onde são definidos o PLL, interrupções e barramentos
- *dbgu.h*: definição das rotinas de debug
- *hid_enumerate.h*: contém as funções específicas para o funcionamento do controlador como um dispositivo USB HID. Neste arquivo estão definidos os métodos de resposta aos comandos SETUP do hub, os descritores do dispositivo HID, assim como suas configurações, interfaces e endpoints.

O arquivo *hid_enumarate.h* inclui um outro arquivo chamado *hid_enumerate.c* que define o método `HID.IsConfigured()`, responsável por verificar se o dispositivo já foi reconhecido pelo computador (procedimento de SETUP concluído). Temos:

```
//*-----------------------------------------------------------
//* \fn    AT91F_UDP_IsConfigured
//*  \brief  Test  if  the  device  is  configured  and  handle
enumeration
//*-----------------------------------------------------------
static uchar AT91F_UDP_IsConfigured(AT91PS_HID pHid)
{
      AT91PS_UDP pUDP = pHid->pUdp;
      AT91_REG isr = pUDP->UDP_ISR;

      if (isr & AT91C_UDP_ENDBUSRES) {
            pUDP->UDP_ICR = AT91C_UDP_ENDBUSRES;
```

```
                // reset all endpoints
                pUDP->UDP_RSTEP  = 0xf;
                pUDP->UDP_RSTEP  = 0;
                // Enable the function
                pUDP->UDP_FADDR = AT91C_UDP_FEN;
                // Configure endpoint 0
                pUDP->UDP_CSR[0]      =      (AT91C_UDP_EPEDS      |
AT91C_UDP_EPTYPE_CTRL);
          }
          else if (isr & AT91C_UDP_EPINT0) {
                pUDP->UDP_ICR = AT91C_UDP_EPINT0;
                AT91F_HID_Enumerate(pHid);
          }
          return pHid->currentConfiguration;
    }
```

Para o caso do dispositivo ainda não ter sido reconhecido, é realizada uma chamada para o método `AT91F_HID_Enumerate()` para que este responda aos comandos de controle SETUP quando solicitado pelo barramento.

Para implementar esse dispositivo, seguindo a especificação USB, necessitamos definir apenas uma configuração, que estabelece apenas uma *interface* e um *endpoint*. O método `AT91F_HID_Enumerate()` assegura que os seguintes descritores sejam enviados durante a rotina de SETUP:

```
    const short interfaceMedidorDescriptor[] = {
      0x0105, // Usage Page (Generic Desktop)
      0xABCD, // Usage (interface medidor)
      0x01A1, // Collection (Application)
      0x0109, //
      0x00A1, //  Collection (Physical)
      0x0905, //    Usage Page
      0x0119, //    Usage Minimumù (01)
      0x0329, //    Usage Maximum (03)
      0x0015, //    Logical Minimum (0)
      0x0125, //    Logical Maximum (1)
      0x0395, //    Report Count (3)
```

```
  0x0175, //    Report Size (1)
  0x0281, //
  0x0195, //    Report Count (1)
  0x0575, //    Report Size (6)
  0x0181, //    6 bit padding
  0x0105, //    Generic desktop
  0x3009, //    Usage (X)
  0x3109, //    Usage(Y)
  0x8115, //    Logical Minimum (-127)
  0x7F25, //    Logical Maximum (127)
  0x0875, //    Report Size (8)
  0x0295, //    Report Count (2)
  0x0681, //    2 position bytes
  0xC0C0
};

// Check http://www.USB.org/developers/hidpage/#Class_Definition
const char devDescriptor[] = {
      /* Device descriptor */
      0x12,   // bLength
      0x01,   // bDescriptorType
      0x10,   // bcdUSBL
      0x01,   //
      0x00,   // bDeviceClass:
      0x00,   // bDeviceSubclass:
      0x00,   // bDeviceProtocol:
      0x08,   // bMaxPacketSize0
      0xFF,   // idVendorL
      0xFF,   //
      0xCD,   // idProductL
      0xAB,   //
      0xEF,   // bcdDeviceL
      0xCD,   //
      0x00,   // iManufacturer    // 0x01
      0x00,   // iProduct
      0x00,   // SerialNumber
      0x01    // bNumConfigs
};
```

```
const char cfgDescriptor[] = {
      /* ============== CONFIGURATION 1 =========== */
      /* Configuration 1 descriptor */
      0x09,   // CbLength
      0x02,   // CbDescriptorType
      0x22,   // CwTotalLength 2 EP + Control
      0x00,
      0x01,   // CbNumInterfaces
      0x01,   // CbConfigurationValue
      0x00,   // CiConfiguration
      0xA0,   // CbmAttributes Bus powered + Remote Wakeup
      0x32,   // CMaxPower: 100mA

      /* Interface Descriptor Requirement */
      0x09, // bLength
      0x04, // bDescriptorType
      0x00, // bInterfaceNumber
      0x00, // bAlternateSetting
      0x01, // bNumEndpoints
      0x03, // bInterfaceClass: HID code
      0x01, // bInterfaceSubclass
      0x82, // bInterfaceProtocol: Mouse
      0x00, // iInterface

      /* HID Descriptor */
      0x09, // bLength
      0x21, // bDescriptor type: HID Descriptor Type
      0x00, // bcdHID
      0x01,
      0x00, // bCountryCode
      0x01, // bNumDescriptors
      0x22, // bDescriptorType
      sizeof(mouseDescriptor), // wItemLength
      0x00,

      /* Endpoint 1 descriptor */
      0x07,   // bLength
```

```
        0x05,   // bDescriptorType
        0x80 + EP_NUMBER, // bEndpointAddress, Endpoint 01 - OUT
        0x03,   // bmAttributes      INT
        0x04,   // wMaxPacketSize: 3 bytes (button, x, y)
        0x00,
        0x0A    // bInterval
};
```

Os descritores acima definem uma interface HID padrão, porém com `bUsagePage` e `bUsageId` não reconhecido pelo *driver* USB padrão do computador.

O protocolo de comunicação utilizado é o mesmo utilizado pelo dispositivo mouse, uma vez que já existem classes próprias para lidar com as mensagens transmitidas, poupando tempo de programação e *debugging*.

Dentre os valores descritos acima, temos, no descritor da configuração (`cfgDescriptor`) a definição de HID como classe utilizada. Esta definição é feita pelo campo `bInterfaceClass` do descritor da interface, conforme descrito na especificação USB.

O campo `bInterfaceProtocol` será utilizado pelo *driver* para reconhecer este dispositivo. Ao definir o ARM como um dispositivo genérico e não padrão do sistema operacional permitimos que o aplicativo de destino da informação tenha acesso irrestrito aos dados recebidos pelo hub USB e asseguramos que não entre em conflito com outros *drivers*.

5.4 UTILIZAÇÃO DA JAVAX PARA LEITURA DOS DADOS

Baseados nas premissas anteriores do projeto e identificada a necessidade de um componente de *software* para facilitar a comunicação entre o dispositivo ARM e o aplicativo Java do lado do PC, verificamos a existência do *javax.*USB, uma solução específica para a plataforma que permite estabelecer a comunicação com USB.

Os primeiros testes nos mostraram que a biblioteca não era compatível com o *framework* USB da Atmel. O dispositivo USB-HID que funcionava perfeitamente com o SO não conseguia enviar dados para o *javax.*USB após a rotina de enumeração. Após analisar o código com a ajuda das ferramentas de *debugging* do IAR e da *libUSB* (biblioteca em C para comunicação com dispositivos USB) foi identificado que o controlador não respondia aos callbacks `GET_INTERFACE` e `SET_INTERFACE`. Após correção do *framework*, o dispositivo foi reconhecido pela *javax.USB*.

A classe HID implementada em nível de software pela *javax.USB* foi utilizada na criação da classe `InterfaceMedidorDriver` no código Java do aplicativo, de modo que o os atributos da classe USB são herdados e modificados visando tratar corretamente os *reports* enviados pelo microcontrolador facilitando a posterior análise. A partir deste momento a leitura dos dados enviados para o barramento pelo ARM será feita por esta nova classe.

Temos então que fazendo uso da pesquisa por interfaces disponível na ferramenta, é realizada uma busca por dispositivos que implementem a classe HID e que possuam o `bUsagePage` e `bUsageId` iguais aos definidos no firmware do dispositivo microcontrolador. Caso o dispositivo seja encontrado, o mesmo é então submetido ao controle da classe `InterfaceMedidorDriver`.

Os dados coletados pela classe `InterfaceMedidorDriver` são então enviados à classe `GraphPlotter`, que processará e exibirá o gráfico para o usuário.

5.5 ANÁLISE DOS DADOS PELO APLICATIVO JAVA

A classe Java responsável pela análise dos dados recebidos pela classe `InterfaceMedidorDriver` é chamada `GraphPlotter`. Esta classe recebe os dados repassados e a armazena em uma pilha, guardando o timestamp do recebimento da mensagem.

Para a plotagem do gráfico, esta pilha de dados é agrupada em intervalos de tempo parametrizáveis para posteriormente exibir a potência instantânea medida acumulada ao longo deste intervalo.

Após agrupados, estes dados são plotados com a ajuda da biblioteca open source para geração de gráficos `JfreeChart`.

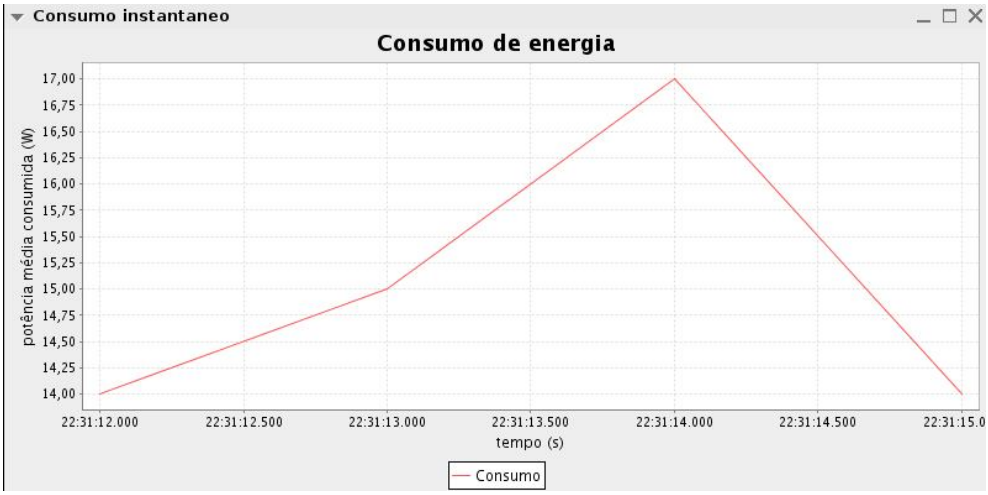

Figura 20 – Exemplo de dados processados e plotados

CONCLUSÕES

A proposta do projeto foi uma sugestão dada pelo Professor Zelenovsky com o intuito claro de nos estimular a desenvolver e aprimorar as aptidões necessárias para prática da Engenharia com foco no desenvolvimento e implementação de plataformas de hardware.

Desde o início sabiamos que o desafio seria árduo e fatigante, mas ao desenrolar dos trabalhos acabamos por verificar que a dificuldade era maior do que pensavamos.

Após a elaboração das premissas teóricas de topologia de *hardware* do projeto, e da definição da plataforma de *software*, a primeira dificuldade que enfrentamos foi na adoção do Sistema Operacional (S.O.) baseado na plataforma *Windows* ou *Linux*, visto que são os dois mais difundidos e com melhor suporte para desenvolvedores.

Gastou-se algumas semanas na tentativa de implementar o funcionamento das bibliotecas *Javax.USB* no *Windows*, mas após muita pesquisa e insucessos na tentativa de adquirir os dados optamos por pela outra opção de S.O., visto que existia melhor documentação para implementação do projeto.

Resolvida essa questão nos deparamos com problemas com a placa medidora de energia MCP3906, que não mais fornecia um pulso de saída TTL. Depois de muitas tentativas de verificar a falha, decidimos por montar um circuito oscilador baseado em um CI 555 que permitia gerar um sinal TTL compatível com o esperado na saída da placa medidora e assim simular o seu funcionamento.

Assim, após muitas semanas e vários percalços inesperados, conseguimos finalizar o interfaceamento da placa medidora utilizando o kit de desenvolvimento ARM com inegável sucesso. Além do que sua futura implementação por outros interessados está muito facilitada, visto a solução adotada permitir uma fácil reprodução do projeto e existir espaço para evoluções.

REFERÊCIAS BIBLIOGRÁFICAS

[1] Salomão, A.; Friedman, A. (2005) Desenvolvimento de uma interface USB para comunicação com o microcontrolador 8051. Monografia de Graduação, Publicação ENE 01/2005, ENE, UnB/FT, Brasília, DF, 118p.

[2] Sasaki, M.; Caixeta, O. (2006). Desenvolvimento de uma interface USB para aquisição de dados de um arranjo de microfones: aplicação em Prótese Auditiva . Monografia de Graduação, Publicação ENE 02/2006, ENE, UnB/FT, Brasília, DF, 134p.

[3] Zelenovsky, R.; Mendonça, A. PC: Um Guia Prático de Hardware e Interfaceamento. 3ª ed. Rio de Janeiro: MZ, 2002. 1023.

[4] Site de referência do padrão USB - USB Implementers Forum - http://www.usb.org

[5] Documentação das classes USB em versão final - http://www.usb.org/developers/devclass_docs

[6] Site de desenvolvimento da bibliteca Javax.USB - http://javax-usb.org

[7] ATMEL. AT91SAM7S Series Preliminary. [S.l.], Abril 2006. Disponível em http://www.atmel.com/products/AT91/

[8] ATMEL. AT91SAM7S Series Preliminary. [S.l.], Novembro 2006. Disponível em http://www.atmel.com/products/AT91/

ANEXOS

I.1 main.c

```
//*----------------------------------------------------------------------------
//*      ATMEL Microcontroller Software Support  -  ROUSSET  -
//*----------------------------------------------------------------------------
//* The software is delivered "AS IS" without warranty or condition of any
//* kind, either express, implied or statutory. This includes without
//* limitation any warranty or condition with respect to merchantability or
//* fitness for any particular purpose, or against the infringements of
//* intellectual property rights of others.
//*----------------------------------------------------------------------------
//* File Name           : USB HID example
//* Object              :
//* Translator          :
//* 1.0 05/Oct/04 ODi     : CReation
//* 1.1 04/Nov/04 JPP     : Add led1 On at power supply
//*----------------------------------------------------------------------------

#include "board.h"
#include "dbgu.h"
#include "hid_enumerate.h"

struct _AT91S_HID       HID;

//*----------------------------------------------------------------------------
//* \fn    AT91F_USB_Open
//* \brief This function Opens the USB device
//*----------------------------------------------------------------------------
void AT91F_USB_Open(void)
{
   // Set the PLL USB Divider
   AT91C_BASE_CKGR->CKGR_PLLR |= AT91C_CKGR_USBDIV_1 ;

   // Specific Chip USB Initialisation
   // Enables the 48MHz USB clock UDPCK and System Peripheral USB Clock
```

```
    AT91C_BASE_PMC->PMC_SCER = AT91C_PMC_UDP;
    AT91C_BASE_PMC->PMC_PCER = (1 << AT91C_ID_UDP);

    // Enable UDP PullUp (USB_DP_PUP) : enable & Clear of the corresponding PIO
    // Set in PIO mode and Configure in Output
    AT91F_PIO_CfgOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);
    // Clear for set the Pul up resistor
    AT91F_PIO_SetOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);
    AT91F_PIO_ClearOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);

    // CDC Open by structure initialization
    AT91F_HID_Open(&HID, AT91C_BASE_UDP);
}

//*-----------------------------------------------------------------------------
//* Function Name       : main
//* Object              :
//*-----------------------------------------------------------------------------
int main ( void )
{
  unsigned int pioStatus;

  //Init trace DBGU
  //AT91F_DBGU_Init();
  //AT91F_DBGU_Printk("\n\r-I- BasicUSB 1.1 (USB_DP_PUP) \n\r0) Set Pull-UP 1)
Clear Pull UP\n\r");

  // Init USB device
  AT91F_USB_Open();

  // Initialize Input
  AT91F_PIO_CfgInput(AT91C_BASE_PIOA,AT91C_PIO_PA24);
  AT91F_PIO_ClearInput(AT91C_BASE_PIOA,AT91C_PIO_PA24);

  // Set in PIO mode and Configure in Input
  AT91F_PIOA_CfgPMC();
```

```
        unsigned int actualState = 0;

        // Wait for the end of enumeration
        while (!HID.IsConfigured(&HID));

        // Start waiting some cmd
        while (1) {
          // Check enumeration
          if (HID.IsConfigured(&HID)) {
            // Wait for several ms
            while ( !((*AT91C_RTTC_RTSR) & AT91C_SYSC_RTTINC) );

            if ( (AT91F_PIO_GetInput(AT91C_BASE_PIOA)&AT91C_PIO_PA24) &&
!actualState )
            {
              actualState = 1;
              HID.SendReport(&HID, CLICKL_ON, 1, 1);
            }
            else if ( (AT91F_PIO_GetInput(AT91C_BASE_PIOA)&AT91C_PIO_PA24) &&
actualState )
            {
              actualState = 1;
            }
            else
            {
              actualState = 0;
            }
          }
        }
      }
```

I.2 hid_enumerates.c

```
//*-----------------------------------------------------------------------
//*      ATMEL Microcontroller Software Support  -  ROUSSET  -
//*-----------------------------------------------------------------------
//* The software is delivered "AS IS" without warranty or condition of any
//* kind, either express, implied or statutory. This includes without
//* limitation any warranty or condition with respect to merchantability or
//* fitness for any particular purpose, or against the infringements of
//* intellectual property rights of others.
//*-----------------------------------------------------------------------
//* File Name           : cdc_enumerate.c
//* Object              : Handle HID enumeration
//*
//* 1.0 Oct 05 2004       : ODi Creation
//*-----------------------------------------------------------------------
#include "board.h"
#include "hid_enumerate.h"

typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned int   uint;

#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define EP_NUMBER 1


const short medidorDescriptor[] = {
  0x0105, // Usage Page (Generic Desktop)
  0x0209, // Usage
  0x01A1, // Collection (Application)
  0x0109, //  Usage
  0x00A1, //  Collection (Physical)
  0x0905, //    Usage Page
  0x0119, //    Usage Minimum
  0x0329, //    Usage Maximum
  0x0015, //    Logical Minimum (0)
```

```
  0x0125, //    Logical Maximum (1)
  0x0395, //    Report Count (3)
  0x0175, //    Report Size (1)
  0x0281, //    3 Button bits
  0x0195, //    Report Count (1)
  0x0575, //    Report Size (6)
  0x0181, //    6 bit padding
  0x0105, //    Generic desktop
  0x3009, //    Usage
  0x3109, //    Usage
  0x8115, //    Logical Minimum (-127)
  0x7F25, //    Logical Maximum (127)
  0x0875, //    Report Size
  0x0295, //    Report Count
  0x0681, //    2 position bytes
  0xC0C0
};

// Check http://www.usb.org/developers/hidpage/#Class_Definition
const char devDescriptor[] = {
        /* Device descriptor */
        0x12,   // bLength
        0x01,   // bDescriptorType
        0x10,   // bcdUSBL
        0x01,   //
        0x00,   // bDeviceClass:
        0x00,   // bDeviceSubclass:
        0x00,   // bDeviceProtocol:
        0x08,   // bMaxPacketSize0
        0xFF,   // idVendorL
        0xFF,   //
        0x00,   // idProductL
        0x00,   //
        0x01,   // bcdDeviceL
        0x00,   //
        0x00,   // iManufacturer    // 0x01
        0x00,   // iProduct
```

```
        0x00,   // SerialNumber
        0x01    // bNumConfigs
};

const char cfgDescriptor[] = {
        /* ============== CONFIGURATION 1 =========== */
        /* Configuration 1 descriptor */
        0x09,   // CbLength
        0x02,   // CbDescriptorType
        0x22,   // CwTotalLength 2 EP + Control
        0x00,
        0x01,   // CbNumInterfaces
        0x01,   // CbConfigurationValue
        0x00,   // CiConfiguration
        0xA0,   // CbmAttributes Bus powered + Remote Wakeup
        0x32,   // CMaxPower: 100mA

        /* Mouse Interface Descriptor Requirement */
        0x09, // bLength
        0x04, // bDescriptorType
        0x00, // bInterfaceNumber
        0x00, // bAlternateSetting
        0x01, // bNumEndpoints
        0x27, // bInterfaceClass: HID code
        0x38, // bInterfaceSubclass
        0x52, // bInterfaceProtocol
        0x00, // iInterface

        /* HID Descriptor */
        0x09, // bLength
        0x21, // bDescriptor type: HID Descriptor Type
        0x00, // bcdHID
        0x01,
        0x00, // bCountryCode
        0x01, // bNumDescriptors
        0x22, // bDescriptorType
        sizeof(medidorDescriptor), // wItemLength
```

```
        0x00,

        /* Endpoint 1 descriptor */
        0x07,   // bLength
        0x05,   // bDescriptorType
        0x80 + EP_NUMBER,   // bEndpointAddress, Endpoint 01 - OUT
        0x03,   // bmAttributes      INT
        0x04,   // wMaxPacketSize: 3 bytes (button, x, y)
        0x00,
        0x0A    // bInterval
};



/* USB standard request code */
#define STD_GET_STATUS_ZERO          0x0080
#define STD_GET_STATUS_INTERFACE     0x0081
#define STD_GET_STATUS_ENDPOINT      0x0082

#define STD_CLEAR_FEATURE_ZERO       0x0100
#define STD_CLEAR_FEATURE_INTERFACE  0x0101
#define STD_CLEAR_FEATURE_ENDPOINT   0x0102

#define STD_SET_FEATURE_ZERO         0x0300
#define STD_SET_FEATURE_INTERFACE    0x0301
#define STD_SET_FEATURE_ENDPOINT     0x0302

#define STD_SET_ADDRESS              0x0500
#define STD_GET_DESCRIPTOR           0x0680
#define STD_SET_DESCRIPTOR           0x0700
#define STD_GET_CONFIGURATION        0x0880
#define STD_SET_CONFIGURATION        0x0900
#define STD_GET_INTERFACE            0x0A81
#define STD_SET_INTERFACE            0x0B01
#define STD_SYNCH_FRAME              0x0C82

/* HID Class Specific Request Code */
```

```
#define STD_GET_HID_DESCRIPTOR      0x0681
#define STD_SET_IDLE                0x0A21

static uchar AT91F_UDP_IsConfigured(AT91PS_HID);
static void AT91F_HID_SendReport(AT91PS_HID, char button, char x, char y);
static void AT91F_HID_Enumerate(AT91PS_HID);


//*-----------------------------------------------------------------------------
//* \fn    AT91F_HID_Open
//* \brief
//*-----------------------------------------------------------------------------
AT91PS_HID AT91F_HID_Open(AT91PS_HID pHid, AT91PS_UDP pUdp)
{
        pHid->pUdp = pUdp;
        pHid->currentConfiguration = 0;
        pHid->IsConfigured = AT91F_UDP_IsConfigured;
        pHid->SendReport   = AT91F_HID_SendReport;
        return pHid;
}

//*-----------------------------------------------------------------------------
//* \fn    AT91F_UDP_IsConfigured
//* \brief Test if the device is configured and handle enumeration
//*-----------------------------------------------------------------------------
static uchar AT91F_UDP_IsConfigured(AT91PS_HID pHid)
{
        AT91PS_UDP pUDP = pHid->pUdp;
        AT91_REG isr = pUDP->UDP_ISR;

        if (isr & AT91C_UDP_ENDBUSRES) {
                pUDP->UDP_ICR = AT91C_UDP_ENDBUSRES;
                // reset all endpoints
                pUDP->UDP_RSTEP  = 0xf;
                pUDP->UDP_RSTEP  = 0;
                // Enable the function
                pUDP->UDP_FADDR = AT91C_UDP_FEN;
```

```
                        // Configure endpoint 0
                        pUDP->UDP_CSR[0]        =       (AT91C_UDP_EPEDS        |
AT91C_UDP_EPTYPE_CTRL);
                }
                else if (isr & AT91C_UDP_EPINT0) {
                        pUDP->UDP_ICR = AT91C_UDP_EPINT0;
                        AT91F_HID_Enumerate(pHid);
                }
                return pHid->currentConfiguration;
        }


        //*-----------------------------------------------------------------------------
        //* \fn    AT91F_HID_SendCoordinates
        //* \brief Send Data through the control endpoint
        //*-----------------------------------------------------------------------------
        static void AT91F_HID_SendReport(AT91PS_HID pHid, char button, char x, char y)
        {
                AT91PS_UDP pUdp = pHid->pUdp;

                // Send report to the host
                pUdp->UDP_FDR[EP_NUMBER] = button;
                pUdp->UDP_FDR[EP_NUMBER] = x;
                pUdp->UDP_FDR[EP_NUMBER] = y;
                pUdp->UDP_CSR[EP_NUMBER] |= AT91C_UDP_TXPKTRDY;

                // Wait for the end of transmission
                while ( !(pUdp->UDP_CSR[EP_NUMBER] & AT91C_UDP_TXCOMP) )
                        AT91F_UDP_IsConfigured(pHid);

                // Clear AT91C_UDP_TXCOMP flag
                if (pUdp->UDP_CSR[EP_NUMBER] & AT91C_UDP_TXCOMP) {
                        pUdp->UDP_CSR[EP_NUMBER] &= ~(AT91C_UDP_TXCOMP);
                        while (pUdp->UDP_CSR[EP_NUMBER] & AT91C_UDP_TXCOMP);
                }
        }
```

```
//*-----------------------------------------------------------------------------
//* \fn    AT91F_USB_SendData
//* \brief Send Data through the control endpoint
//*-----------------------------------------------------------------------------
static void AT91F_USB_SendData(AT91PS_UDP pUdp, const char *pData, uint length)
{
        uint cpt = 0;
        AT91_REG csr;

        do {
                cpt = MIN(length, 8);
                length -= cpt;

                while (cpt--)
                        pUdp->UDP_FDR[0] = *pData++;

                if (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) {
                        pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
                        while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
                }

                pUdp->UDP_CSR[0] |= AT91C_UDP_TXPKTRDY;
                do {
                        csr = pUdp->UDP_CSR[0];

                        // Data IN stage has been stopped by a status OUT
                        if (csr & AT91C_UDP_RX_DATA_BK0) {
                                pUdp->UDP_CSR[0]                                &=
~(AT91C_UDP_RX_DATA_BK0);
                                return;
                        }
                } while ( !(csr & AT91C_UDP_TXCOMP) );

        } while (length);

        if (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) {
                pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
```

```
                while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
            }
        }

        //*-----------------------------------------------------------------------
        //* \fn    AT91F_USB_SendZlp
        //* \brief Send zero length packet through the control endpoint
        //*-----------------------------------------------------------------------
        void AT91F_USB_SendZlp(AT91PS_UDP pUdp)
        {
            pUdp->UDP_CSR[0] |= AT91C_UDP_TXPKTRDY;
            while ( !(pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) );
            pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
            while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
        }

        //*-----------------------------------------------------------------------
        //* \fn    AT91F_USB_SendStall
        //* \brief Stall the control endpoint
        //*-----------------------------------------------------------------------
        void AT91F_USB_SendStall(AT91PS_UDP pUdp)
        {
            pUdp->UDP_CSR[0] |= AT91C_UDP_FORCESTALL;
            while ( !(pUdp->UDP_CSR[0] & AT91C_UDP_ISOERROR) );
            pUdp->UDP_CSR[0]        &=        ~(AT91C_UDP_FORCESTALL        |
AT91C_UDP_ISOERROR);
            while      (pUdp->UDP_CSR[0]      &      (AT91C_UDP_FORCESTALL      |
AT91C_UDP_ISOERROR));
        }

        //*-----------------------------------------------------------------------
        //* \fn    AT91F_HID_Enumerate
        //* \brief This function is a callback invoked when a SETUP packet is received
        //*-----------------------------------------------------------------------
        static void AT91F_HID_Enumerate(AT91PS_HID pHid)
        {
            AT91PS_UDP pUDP = pHid->pUdp;
```

```
uchar bmRequestType, bRequest;
ushort wValue, wIndex, wLength, wStatus;


if ( !(pUDP->UDP_CSR[0] & AT91C_UDP_RXSETUP) )
        return;

bmRequestType = pUDP->UDP_FDR[0];
bRequest      = pUDP->UDP_FDR[0];
wValue        = (pUDP->UDP_FDR[0] & 0xFF);
wValue       |= (pUDP->UDP_FDR[0] << 8);
wIndex        = (pUDP->UDP_FDR[0] & 0xFF);
wIndex       |= (pUDP->UDP_FDR[0] << 8);
wLength       = (pUDP->UDP_FDR[0] & 0xFF);
wLength      |= (pUDP->UDP_FDR[0] << 8);

if (bmRequestType & 0x80) {
        pUDP->UDP_CSR[0] |= AT91C_UDP_DIR;
        while ( !(pUDP->UDP_CSR[0] & AT91C_UDP_DIR) );
}
pUDP->UDP_CSR[0] &= ~AT91C_UDP_RXSETUP;
while ( (pUDP->UDP_CSR[0]  & AT91C_UDP_RXSETUP)  );

// Handle supported standard device request Cf Table 9-3 in USB specification
Rev 1.1
switch ((bRequest << 8) | bmRequestType) {
case STD_GET_DESCRIPTOR:
        if (wValue == 0x100)      // Return Device Descriptor
                AT91F_USB_SendData(pUDP,                devDescriptor,
MIN(sizeof(devDescriptor), wLength));
        else if (wValue == 0x200)  // Return Configuration Descriptor
                AT91F_USB_SendData(pUDP,                cfgDescriptor,
MIN(sizeof(cfgDescriptor), wLength));
        else
                AT91F_USB_SendStall(pUDP);
        break;
case STD_SET_ADDRESS:
```

```
                AT91F_USB_SendZlp(pUDP);
                pUDP->UDP_FADDR = (AT91C_UDP_FEN | wValue);
                pUDP->UDP_GLBSTATE  = (wValue) ? AT91C_UDP_FADDEN : 0;
                break;
        case STD_SET_CONFIGURATION:
                pHid->currentConfiguration = wValue;
                AT91F_USB_SendZlp(pUDP);
                pUDP->UDP_GLBSTATE    = (wValue)  ?  AT91C_UDP_CONFG  :
AT91C_UDP_FADDEN;
                pUDP->UDP_CSR[EP_NUMBER] = (wValue) ? (AT91C_UDP_EPEDS
| AT91C_UDP_EPTYPE_BULK_IN)   : 0;
                break;
        case STD_GET_CONFIGURATION:
                AT91F_USB_SendData(pUDP, (char *) &(pHid->currentConfiguration),
sizeof(pHid->currentConfiguration));
                break;
        case STD_GET_STATUS_ZERO:
                wStatus = 0;
                AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
                break;
        case STD_GET_STATUS_INTERFACE:
                wStatus = 0;
                AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
                break;
        case STD_GET_STATUS_ENDPOINT:
                wStatus = 0;
                wIndex &= 0x0F;
                if ((pUDP->UDP_GLBSTATE & AT91C_UDP_CONFG) && (wIndex <=
3)) {
                        wStatus          =          (pUDP->UDP_CSR[wIndex]          &
AT91C_UDP_EPEDS) ? 0 : 1;
                        AT91F_USB_SendData(pUDP,      (char      *)      &wStatus,
sizeof(wStatus));
                }
                else  if  ((pUDP->UDP_GLBSTATE  &  AT91C_UDP_FADDEN)  &&
(wIndex == 0)) {
```

```
                        wStatus = (pUDP->UDP_CSR[wIndex] &
AT91C_UDP_EPEDS) ? 0 : 1;
                        AT91F_USB_SendData(pUDP, (char *) &wStatus,
sizeof(wStatus));
                }
                else
                        AT91F_USB_SendStall(pUDP);
                break;
        case STD_SET_FEATURE_ZERO:
                AT91F_USB_SendStall(pUDP);
           break;
        case STD_SET_FEATURE_INTERFACE:
                AT91F_USB_SendZlp(pUDP);
                break;
        case STD_SET_FEATURE_ENDPOINT:
                wIndex &= 0x0F;
                if ((wValue == 0) && wIndex && (wIndex <= 3)) {
                        pUDP->UDP_CSR[wIndex] = 0;
                        AT91F_USB_SendZlp(pUDP);
                }
                else
                        AT91F_USB_SendStall(pUDP);
                break;
        case STD_CLEAR_FEATURE_ZERO:
                AT91F_USB_SendStall(pUDP);
           break;
        case STD_CLEAR_FEATURE_INTERFACE:
                AT91F_USB_SendZlp(pUDP);
                break;
        case STD_CLEAR_FEATURE_ENDPOINT:
                wIndex &= 0x0F;
                if ((wValue == 0) && wIndex && (wIndex <= 3)) {
                        if (wIndex == 1)
                                pUDP->UDP_CSR[1] = (AT91C_UDP_EPEDS |
AT91C_UDP_EPTYPE_BULK_OUT);
                        else if (wIndex == 2)
```

```
                                        pUDP->UDP_CSR[2]    =    (AT91C_UDP_EPEDS    |
AT91C_UDP_EPTYPE_BULK_IN);
                                    else if (wIndex == 3)
                                        pUDP->UDP_CSR[3]    =    (AT91C_UDP_EPEDS    |
AT91C_UDP_EPTYPE_ISO_IN);
                                    AT91F_USB_SendZlp(pUDP);
                                }
                                else
                                    AT91F_USB_SendStall(pUDP);
                                break;

                        case STD_GET_INTERFACE:
                            AT91F_USB_SendZlp(pUDP);
                          break;
                        case STD_SET_INTERFACE:
                            AT91F_USB_SendZlp(pUDP);
                          break;

                          // handle HID class requests
                          case STD_GET_HID_DESCRIPTOR:
                                if (wValue == 0x2200)      // Return Mouse Descriptor
                                    AT91F_USB_SendData(pUDP,        (const        char        *)
medidorDescriptor, MIN(sizeof(medidorDescriptor), wLength));
                                else
                                    AT91F_USB_SendStall(pUDP);
                                break;

                          case STD_SET_IDLE:
                                AT91F_USB_SendZlp(pUDP);
                                break;

                          default:
                                AT91F_USB_SendStall(pUDP);
                            break;
                          }
                }
```