UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ENGENHARIA DE SÃO CARLOS

DESENVOLVIMENTO DE DISPOSITIVO DE AQUISIÇÃO DE DADOS E CONTROLE DE BAIXO CUSTO PARA UTILIZAÇÃO COM NI LABVIEW

Fábio Felipe Mira Machuca

Péricles Bernardes Caravieri

ORIENTADOR: Prof. Dr. Daniel Varela Magalhães

São Carlos

Dezembro, 2012

Machuca, Fábio Felipe Mira
M151d Desenvolvimento de dispositivo de aquisição de dados e controle de baixo custo para utilização com NI LabVIEW / Fábio Felipe Mira Machuca, Péricles Bernardes Caravieri; orientador Daniel Varela Magalhães. – São Carlos, 2012.

Monografia (Graduação em Engenharia Mecatrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2012.

1. Aquisição de dados. 2. Acionamento digital. 3. PWM. 4. LabVIEW. 5. PIC18. 6. USB. I. Titulo. II. Caravieri, Péricles Bernardes.

FOLHA DE AVALIAÇÃO

Candidatos: Fábio Felipe Mira Machuca e Péricles Bernardes Caravieri

Título: Desenvolvimento de dispositivo de aquisição de dados e controle de baixo custo para utilização com NI LabVIEW

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos da
Universidade de São Paulo
Curso de Engenharia Mecatrônica

BANCA EXAMINADORA

Eng. João Marcelo Pereira Nogueira

Nota atribuída: 9,0 (nove)

João Marcelo Pereira Nogueira
(assinatura)

Prof. Dr. Rodrigo Nicoletti

Nota atribuída: 9,0 (nove)

Rodrigo Nicoletti
(assinatura)

Prof. Dr. Daniel Varela Magalhães (orientador)

Nota atribuída: 9,0 (nove)

Daniel Varela Magalhães
(assinatura)

Média: 9,0 (nove)

Resultado: APROVADO

Data: 13/12/2012

FÁBIO FELIPE MIRA MACHUCA

PÉRICLES BERNARDES CARAVIERI

DESENVOLVIMENTO DE DISPOSITIVO DE AQUISIÇÃO DE DADOS E CONTROLE DE BAIXO CUSTO PARA UTILIZAÇÃO COM NI LABVIEW

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos, da
Universidade de São Paulo

Curso de Engenharia Mecatrônica

ORIENTADOR: Prof. Dr. Daniel Varela Magalhães

São Carlos

Dezembro, 2012

Resumo

Este trabalho dedica-se ao desenvolvimento de um dispositivo de baixo custo que seja capaz de realizar tarefas de aquisição de dados e controle. Dotado de entradas e saídas digitais, além de um conversor A/D para leituras analógicas e dois módulos de saída de sinal PWM, o dispositivo comunica-se com um computador através da comunicação USB e pelo *software NI LabVIEW*. Pretendemos oferecer uma solução simples e eficiente para problemas que não exijam extrema precisão e altas taxas de aquisição, visando a uma utilização em meio acadêmico, para aprendizado, e desenvolvimento de projetos domésticos. Ainda assim, para sistemas de baixa frequência e acionamentos digitais, o dispositivo é robusto e tão eficaz quanto qualquer solução comercial.

Palavras-chave: Aquisição de dados. Acionamento digital. PWM. LabVIEW. PIC18. USB.

Abstract

This work focuses on designing a low cost device capable of doing control and data acquisition tasks. The device has digital inputs and outputs, one analog-to-digital converter, to read analog values, and two pulse-width modulation outputs. It communicates with a computer through USB (Universal Serial Bus) protocol and provides its functions as LabVIEW blocks. The objective is to provide a simple and efficient solution to problems that don't require very high precision or high acquisition rates, to be used for teaching, in academic environments, and learning, at home. For low frequency systems and digital actuation the device is just as effective and robust as any commercial solution.

Keywords: Data acquisition. Digital actuation. PWM. LabVIEW. PIC18. USB.

Sumário

1 Introdução

O objetivo deste trabalho é desenvolver um dispositivo para aquisição de dados e acionamento digital que seja acessível, tanto em relação ao custo, quanto em relação à simplicidade de uso. Em relação ao custo, a placa foi projetada para ser construída de maneira fácil e de baixo custo: a placa possui apenas uma face, com trilhas largas, e os componentes utilizados são comuns e facilmente encontrados em lojas de componentes eletrônicos. Do ponto de vista da simplicidade de uso, o protocolo USB foi escolhido para fazer a comunicação entre o dispositivo e o computador, por ser encontrado em praticamente todos os computadores atuais, e para a utilização dos recursos foi escolhido o software LabVIEW, da National Instruments, por ser uma plataforma bastante usada para aquisição de dados e controle, de fácil utilização até mesmo para iniciantes.

O dispositivo possui: 16 entradas e saídas digitais, sendo que cada uma delas pode ser configurada como entrada ou saída independentemente; 8 entradas analógicas, que podem ser utilizadas para leitura de sensores analógicos com uma frequência de aquisição de até 1kHz; e 2 módulos PWM, sendo um com saída simples, e outro com duas saídas complementares, podendo ser usadas para o acionamento de uma carga ligada com ponte H em modo *half-bridge* diretamente.

A motivação para o desenvolvimento deste trabalho vem das disciplinas práticas do curso de Engenharia Mecatrônica, que podem se beneficiar de um dispositivo como o desenvolvido neste trabalho, pois ele é bastante versátil e de fácil acesso aos alunos.

2 Entradas e saídas digitais

2.1 Sistemas digitais

Na ciência, na tecnologia, nos negócios e em muitos outros campos de trabalho, estamos constantemente tratando com quantidades. Quantidades são medidas, monitoradas, guardadas, manipuladas aritmeticamente, observadas ou utilizadas de alguma outra maneira na maioria dos sistemas físicos. Quando manipulamos quantidades diversas, é importante que saibamos representar seus valores de modo eficiente e preciso.

Na representação digital, as quantidades não são representadas por quantidades proporcionais, mas por símbolos denominados dígitos, variando em saltos ou degraus (*steps*). Em outras palavras, essa representação digital varia de maneira discreta (em degraus), quando comparada com a representação da mesma grandeza fornecida por um sistema contínuo ou analógico.

Um sistema digital é uma combinação de dispositivos projetados para manipular informação lógica ou quantidades físicas que são representadas no formato digital; ou seja, as quantidades podem assumir apenas valores discretos. Esses dispositivos são, na maioria das vezes, eletrônicos, mas podem também ser mecânicos, magnéticos ou pneumáticos. Alguns dos sistemas digitais mais conhecidos são os computadores digitais e as calculadoras, os equipamentos digitais de áudio e vídeo, sistema de telefonia (considerado o maior sistema digital do mundo).

2.2 Vantagens das técnicas digitais

Um aumento no número de aplicações em eletrônica, assim como acontece em muitas outras tecnologias, está relacionado ao uso de técnicas digitais para implementar funções que eram realizadas usando-se métodos analógicos, somente. Os principais motivos da migração para a tecnologia digital são:

- Os sistemas digitais são mais fáceis de serem projetados. Isso porque os circuitos utilizados são circuitos de chaveamento, nos quais não importam os valores exatos

de tensão ou corrente, mas apenas a faixa – ALTA (*high*) ou BAIXA (*low*) – na qual eles se encontram;

- Fácil armazenamento de informações. Esta é uma habilidade de dispositivos e circuitos especiais, que podem guardar (*latch*) informação digital e mantê-la pelo tempo necessário, e técnicas de armazenamento de massa (grande quantidade de informação), que podem armazenar bilhões de *bits* de informação em um espaço físico relativamente pequeno;
- Maior precisão e exatidão. Os sistemas digitais podem manipular todos os dígitos de precisão necessários simplesmente acrescentando mais circuitos de chaveamento. Nos sistemas analógicos, a precisão é limitada porque os valores de tensão e corrente são diretamente dependentes dos valores dos componentes do circuito e são afetados por flutuações aleatórias na tensão (ruído);
- As operações podem ser programadas. É bastante fácil projetar um sistema digital cuja operação é controlada por um conjunto de instruções armazenadas denominado "programa". Os sistemas analógicos também podem ser programados, porém a variedade e a complexidade das operações disponibilizadas são bastante limitadas;
- Os circuitos digitais são menos afetados por ruído. Em geral, flutuações espúrias na tensão (ruído) não são tão críticas em sistemas digitais porque o valor exato da tensão não é importante, desde que o ruído não tenha amplitude suficiente que dificulte a distinção entre um nível ALTO (H) e um nível BAIXO (L);
- *CI's* (*chips*) digitais podem ser fabricados com mais dispositivos internos. É verdade que os circuitos analógicos também foram beneficiados com o grande desenvolvimento da tecnologia de circuitos integrados, mas esses circuitos são relativamente complexos e utilizam dispositivos que não podem ser economicamente integrados (capacitores de alto valor, resistores de precisão, indutores e transformadores).

2.3 Limitações das técnicas digitais

Na verdade, há apenas uma grande desvantagem quando se usam técnicas digitais: o mundo real é quase totalmente analógico.

A maioria das quantidades (grandezas) físicas é de natureza analógica, e essas grandezas são muitas vezes as entradas e saídas monitoradas, operadas (alteradas) e controladas por um sistema. Como exemplos temos a temperatura, a pressão, a posição, a velocidade, o nível de um líquido e a vazão, entre outros. Estamos habituados a expressar essas grandezas "digitalmente", quando dizemos que a temperatura é 25 °C, mas o que estamos realmente fazendo é uma aproximação digital para uma grandeza inerentemente analógica.

Para obter as vantagens das técnicas digitais quando tratarmos com entradas e saídas analógicas, três passos devem ser seguidos:

- Converter as entradas analógicas do mundo real para o formato digital;
- Realizar o processamento da informação digital;
- Converter as saídas digitais de volta ao formato analógico (o formato do mundo real).

2.4 Sistema binário

O sistema de numeração decimal não é conveniente para ser implementado em sistemas digitais. Por exemplo, é muito difícil projetar um equipamento eletrônico que opere com dez níveis diferentes de tensão (cada um representando um caractere decimal, 0 a 9). Por outro lado, é muito fácil projetar um circuito eletrônico simples e preciso que opere com apenas dois níveis de tensão. Por esse motivo, quase todos os sistemas digitais usam o sistema de numeração binário (ou de base 2) como sistema básico de numeração para suas operações, embora outros sistemas de numeração sejam, muitas vezes, usados juntamente com o sistema binário.

No sistema binário há apenas dois símbolos ou valores possíveis para os dígitos: 0 e 1. Esse sistema de base 2 também pode ser usado para representar qualquer quantidade que possa ser representada em decimal ou em qualquer outro sistema de numeração. Entretanto, é comum que o sistema binário use um número maior de dígitos para expressar um determinado valor.

No sistema binário, o termo “dígito binário” (*binary digit*) é quase sempre abreviado com o uso do termo *bit*, o qual usaremos deste ponto em diante. O *bit* mais significativo (*most significant bit* – MSB) é o da esquerda (o de maior peso). O *bit* menos significativo (*least significant bit* – LSB) é o da direita (o de menor peso).

A maioria dos microcomputadores e microcontroladores manipula e armazena informações e dados binários em grupos de 8 *bits*, de modo que uma sequência de 8 *bits* recebe o nome de *byte*. Um *byte* é constituído sempre de 8 *bits* e pode representar quaisquer tipos de dados ou informações.

2.5 Representação de quantidades binárias

Em sistemas digitais, a informação processada é normalmente apresentada na forma binária. As quantidades binárias podem ser representadas por qualquer dispositivo que tenha apenas dois estados de operação ou duas condições possíveis. Por exemplo, uma chave tem apenas dois estados: aberta ou fechada. Podemos, arbitrariamente, representar uma chave aberta pelo binário “0” e uma chave fechada pelo binário “1”. Com essas condições, podemos representar qualquer número binário a partir desses estados.

Existem vários outros dispositivos que têm apenas dois estados de operação ou que podem ser operados em duas condições extremas. Entre esses dispositivos temos: lâmpada (acesa ou apagada), diodo (em condução ou em corte), relê (energizado ou não), transistor (em corte ou em saturação), fotocélula (iluminada ou no escuro), termostato (aberto ou fechado), engate mecânico (engatado ou desengatado) e um ponto em um disco magnético (magnetizado ou desmagnetizado).

Em sistemas eletrônicos digitais, uma informação binária é representada por tensões (ou correntes) que estão presentes nas entradas e saídas de diversos circuitos. Tipicamente, os número binários 0 e 1 são representados por dois níveis de tensões nominais. Por exemplo, zero *volt* (0 V) pode representar o binário “0”, e +5 V pode representar o binário “1”. Na realidade, devido às variações nos circuitos, o 0 e o 1 são representados por faixas de tensão. Isso é ilustrado na Figura 1(a), na qual qualquer tensão entre 0 e 0,8 V representa o binário 0 e qualquer tensão entre 2 e 5 V representa o binário 1. Todos os sinais de

entrada e saída estarão dentro de uma dessas faixas, exceto durante as transições de um nível para o outro.

Nos sistemas digitais, o valor exato da tensão não é importante; por exemplo, para os valores de tensões da Figura 1(a), uma tensão de 3,6 V significa o mesmo que uma tensão de 4,3, o que não ocorre em sistemas analógicos, em que a tensão é proporcional à grandeza medida.

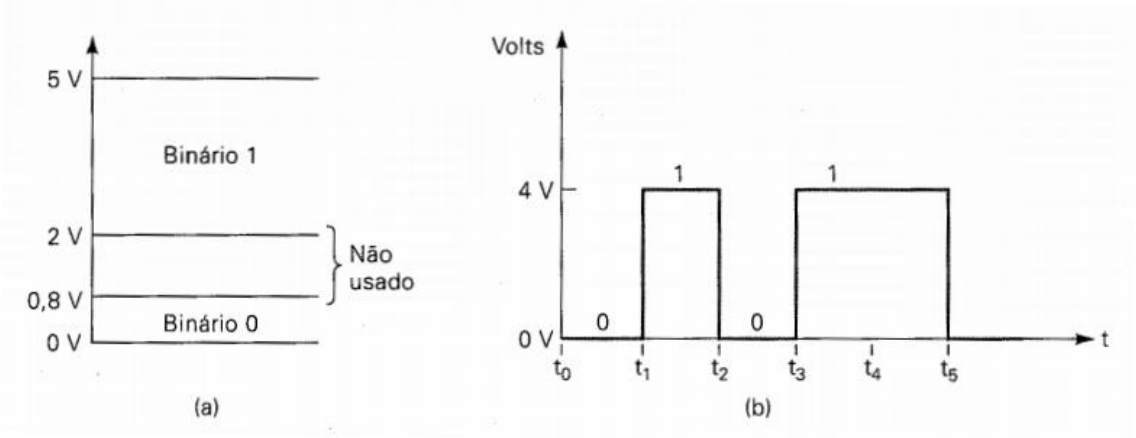


Figura 1 - (a) Valores típicos de tensões em um sistema digital; (b) diagrama de tempo de um sinal digital típico.

2.6 Sinais digitais e diagramas de tempo

A Figura 1(b) mostra um sinal digital típico e como ele varia ao longo do tempo. Na realidade, trata-se de um gráfico de tensão *versus* tempo que é denominado "diagrama de tempo". A escala de tempo, horizontal, é dividida em intervalos regulares, começando em t_0 e passando por t_1, t_2 e assim por diante. Por exemplo, no diagrama de tempo mostrado, o sinal começa em 0 V (que é um binário 0) no instante t_0 e se mantém nesse valor até o instante t_1. Em t_1, o sinal faz uma transição rápida para 4 V (um binário 1). Em t_2, o sinal salta de volta para 0 V. Transições similares ocorrem em t_3 e t_5. Observe que em t_4 o sinal não muda, permanecendo em 4 V de t_3 a t_5.

As transições no diagrama de tempo foram desenhadas como linhas verticais; desse modo, as transições parecem ser instantâneas, sendo que, na realidade, não é isso o que ocorre. Entretanto, em muitas situações, os tempos de transição são tão curtos, comparados com os tempos entre as transições, que podemos mostrá-los como linhas verticais.

Os diagramas de tempo são usados extensivamente para mostrar como os sinais digitais variam no tempo, em especial para mostrar as relações entre dois ou mais sinais digitais de um mesmo circuito ou sistema. Por meio da visualização de um ou mais sinais digitais em um sistema de medição, como um osciloscópio, por exemplo, podemos comparar os sinais com os respectivos diagramas de tempo pretendidos. Essa é uma etapa importante nos procedimentos de teste e verificação de defeitos usada em sistemas digitais.

2.7 Circuitos integrados digitais

Quase todos os circuitos digitais usados nos modernos sistemas digitais são circuitos integrados (*CIs*). A disponibilidade de uma grande variedade de *CIs* lógicos tem tornado possível a implementação de sistemas digitais complexos que são menores e mais seguros que os mesmos circuitos implementados com componentes discretos.

Várias tecnologias de fabricação de circuitos integrados são usadas para a produção de *CIs* digitais, sendo as tecnologias TTL, CMOS, NMOS e ECL as mais comuns. Uma difere da outra pelo tipo de circuito usado para implementar a operação lógica desejada. Por exemplo, a tecnologia TTL (*transistor – transistor logic*, lógica transistor – transistor) usa o transistor bipolar como seu principal elemento de circuito, enquanto a tecnologia CMOS (*complementary metal – oxide – semiconductor*, semicondutor de óxido metálico complementar) usa o MOSFET tipo enriquecimento como seu principal elemento de circuito.

3 *Pulse Width Modulation* (PWM)

3.1 Princípio de funcionamento

Os controles de potência, inversores de frequência, conversores para servomotores, fontes chaveadas e muitos outros circuitos utilizam a tecnologia do PWM, ou Modulação de Largura de Pulso, como base de seu funcionamento.

Para que se entenda como funciona esta tecnologia no controle de potência, partimos de um circuito imaginário formado por um interruptor de ação muito rápida e uma carga que deve ser controlada, de acordo com a Figura 2.

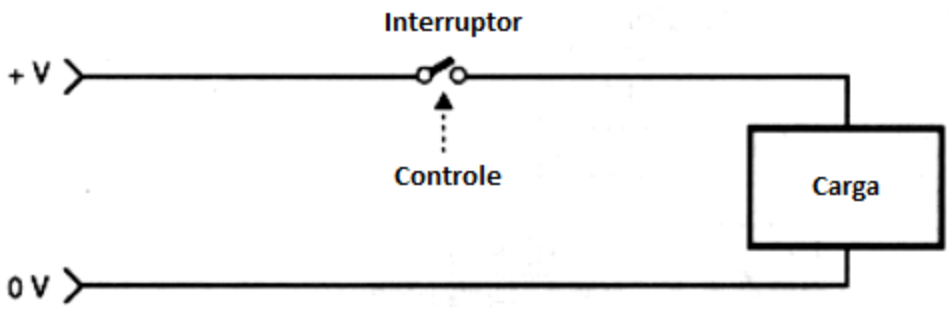


Figura 2 - Quando abrimos e fechamos o interruptor, controlamos a corrente na carga.

Quando o interruptor está aberto não há corrente na carga e a potência aplicada é nula. No instante em que o interruptor é fechado, a carga recebe a tensão total da fonte e a potência aplicada é máxima.

Para obtermos uma potência intermediária, digamos 50%, aplicada à carga, fazemos com que a chave seja aberta e fechada rapidamente, de modo a ficar 50% do tempo aberta e 50% fechada. Isso significa que, em média, teremos metade do tempo com corrente e metade do tempo sem corrente, como ilustra a Figura 3.

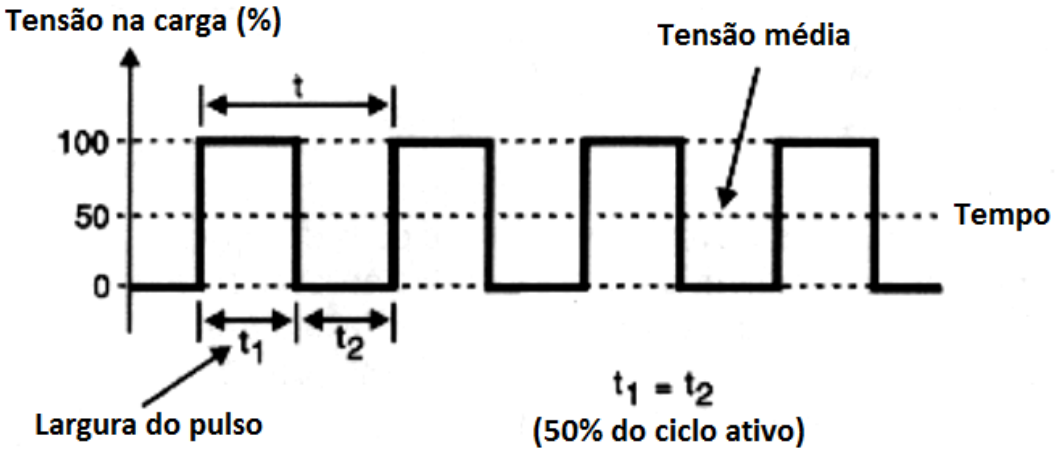


Figura 3 - Abrindo e fechando em tempos controlados variamos a tensão média.

A potência média e, portanto, a própria tensão média aplicada à carga é, neste caso, 50% da tensão de entrada.

Assim, o interruptor fechado pode definir uma largura de pulso pelo tempo em que ele fica nesta condição, e um intervalo entre pulsos pelo tempo em que ele fica aberto. Os dois tempos juntos definem o período e, portanto, uma frequência de controle.

A relação entre o tempo em que temos o pulso e a duração de um ciclo completo de operação do interruptor nos define ainda o *duty cycle*, ou ciclo ativo, ou ciclo de trabalho, conforme é mostrado na Figura 4.

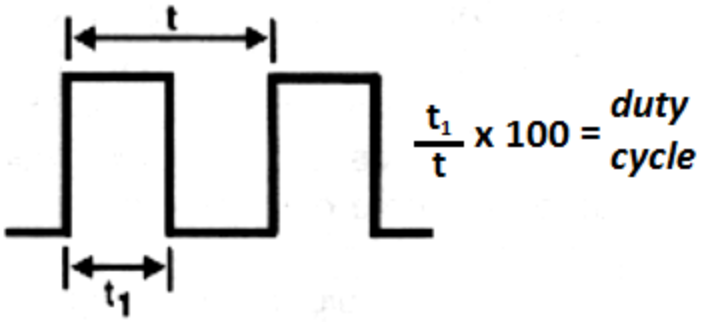


Figura 4 - Definindo o duty cycle.

Variando-se a largura do pulso e também o intervalo de modo a termos ciclos ativos diferentes, podemos controlar a potência média aplicada a uma carga. Assim, quando a

largura do pulso varia de zero até o máximo, a potência também varia na mesma proporção, conforme está indicado na Figura 5.

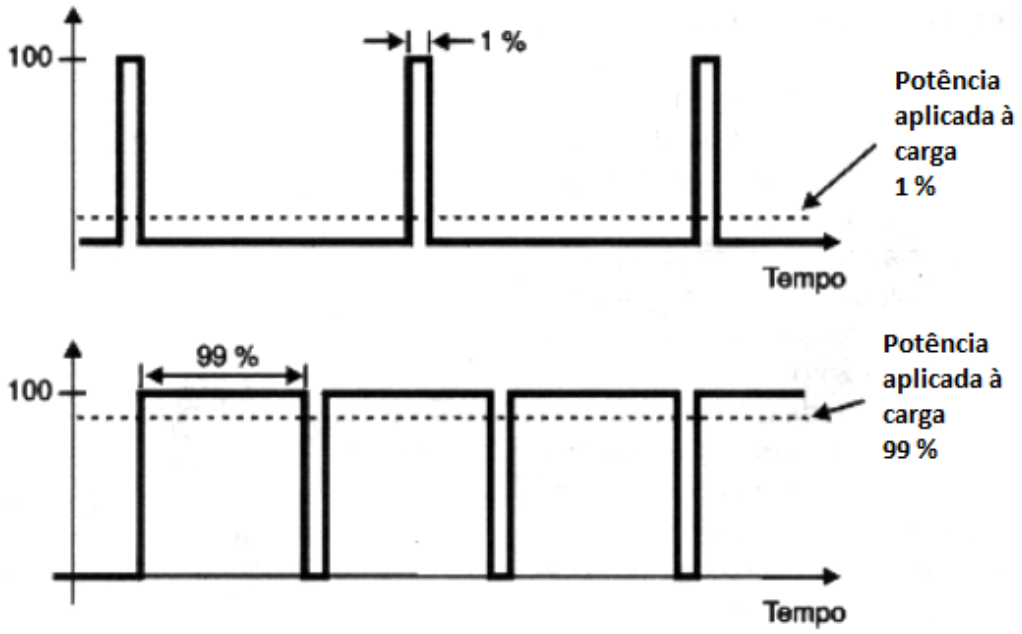


Figura 5 - Controlando a potência pelo duty cycle.

Na prática, o interruptor é substituído por um dispositivo de estado sólido que possa ser capaz de chavear o circuito como, por exemplo, um transistor bipolar, um FET de potência, um IGBT ou até mesmo um SCR. A este dispositivo é então ligado um oscilador que possa ter seu ciclo controlado numa grande faixa de valores. Esse oscilador, muitas vezes, é parte parte de um circuito integrado de um microcontrolador, e o *duty cycle* pode ser controlado através da programação deste microcontrolador. Sistemas dedicados, como o montado com um circuito integrado 4093, também podem ser construídos. Os sinais gerados são, então aplicados ao transistor de potência que comanda a carga.

3.2 Vantagens e cuidados com PWM

Na operação de um controle por PWM existem diversas vantagens a serem consideradas e alguns pontos para os quais o projetista deve ficar atento para se aproveitar ao máximo estas vantagens.

Na condição de aberto, nenhuma corrente circula pelo dispositivo de controle e, portanto, sua dissipação é nula. Na condição de fechado, teoricamente, se ele apresenta uma resistência nula, a queda de tensão é nula, e ele não dissipa também nenhuma potência.

Isso significa que, na teoria, os controles PWM não dissipam potência alguma e, portanto, consistem em soluções ideais para este tipo de aplicação.

Na prática, entretanto, em primeiro lugar, os dispositivos usados no controle não são capazes de abrir e fechar o circuito num tempo infinitamente pequeno. Eles precisam de um tempo para mudar de estado e, neste intervalo, sua resistência sobe de um valor muito pequeno até o infinito e vice-versa, numa curva de comutação semelhante à mostrada na Figura 6.

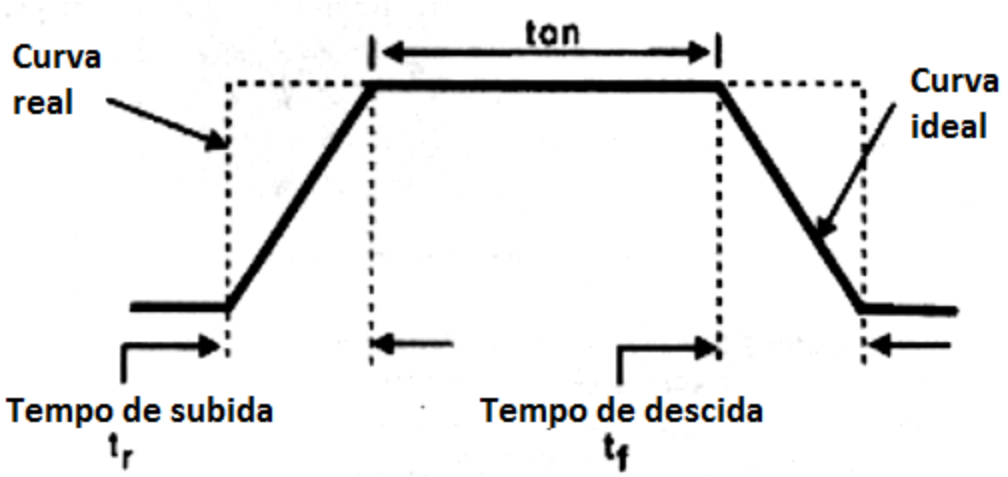


Figura 6 - Nos intervalos de t_r e t_f, o dispositivo gera calor em boa quantidade.

Neste intervalo de tempo a queda de tensão e a corrente através do dispositivo não são nulas, e uma boa quantidade de calor poderá ser gerada conforme a carga controlada. Dependendo da frequência de controle e da resposta do dispositivo usado, uma boa quantidade de calor poderá ser gerada neste processo de comutação.

Entretanto, mesmo com este problema, a potência gerada num controle PWM ainda é muito menor do que num circuito de controle linear equivalente. Transistores de comutação rápidos, FETs de potência, e outros componentes de chaveamento podem ser suficientemente rápidos para permitir que projetos de controles de potências elevadas

sejam implementados sem a necessidade de grandes dissipadores de calor ou sem que tenham problemas de perdas de energia por geração de calor que possam ser preocupantes.

O segundo problema que poderá surgir vem justamente do fato de que os transistores de efeito de campo ou bipolares usados em comutação não se comportam como resistências nulas, quando saturados. Os transistores bipolares podem apresentar uma queda de tensão de até alguns volts quando saturados, o mesmo ocorrendo com os FETs.

Assim, dependendo da aplicação, principalmente nos circuitos de baixa tensão, os transistores de potência bipolares ou mesmo os IGBTs podem ser ainda melhores que os FETs de potência.

4 Entradas analógicas

4.1 Quantidade analógica e sua importância

Uma quantidade analógica é uma quantidade que pode assumir qualquer valor dentro de uma faixa contínua de valores e, mais importante, o seu valor exato é significativo. Por exemplo, uma saída de tensão de 1,7 V de um sensor de temperatura pode representar 17,0°C. Se essa saída for 1,75 V, a temperatura sendo medida não será mais a mesma. Em outras palavras, cada valor apresenta um significado diferente.

A maioria das variáveis físicas é analógica e, portanto, pode assumir qualquer valor dentro de uma faixa contínua de valores. Podemos citar como exemplo: posição, velocidade, aceleração, temperatura, pressão, intensidade luminosa. Em diversas aplicações é necessário realizar medições dessas grandezas físicas. Para isso, usa-se um transdutor, que é um dispositivo que converte uma variável física em uma variável elétrica. A saída analógica de um transdutor é uma tensão ou uma corrente, que é proporcional à variável física sendo medida. Alguns transdutores mais comuns são: termistores, fotocélulas, fotodiodos, transdutores de pressão, tacômetros.

Embora as grandezas medidas normalmente sejam analógicas, os sistemas de processamento, como microcontroladores ou computadores, apenas trabalham com dados digitais. Assim é necessário converter as grandezas físicas para sinais digitais, que podem ser manipulados pelos processadores. Os sistemas de aquisição de dados são responsáveis por realizar essa conversão.

4.2 Sistema de aquisição

Os transdutores são responsáveis por converter as variáveis físicas em variáveis elétricas. A conversão de uma grandeza analógica para uma grandeza digital, por sua vez, é realizada por um conversor analógico digital. Porém existem outros componentes

necessários a um bom sistema de aquisição de dados. A figura abaixo ilustra um sistema de aquisição de dados de quatro canais:

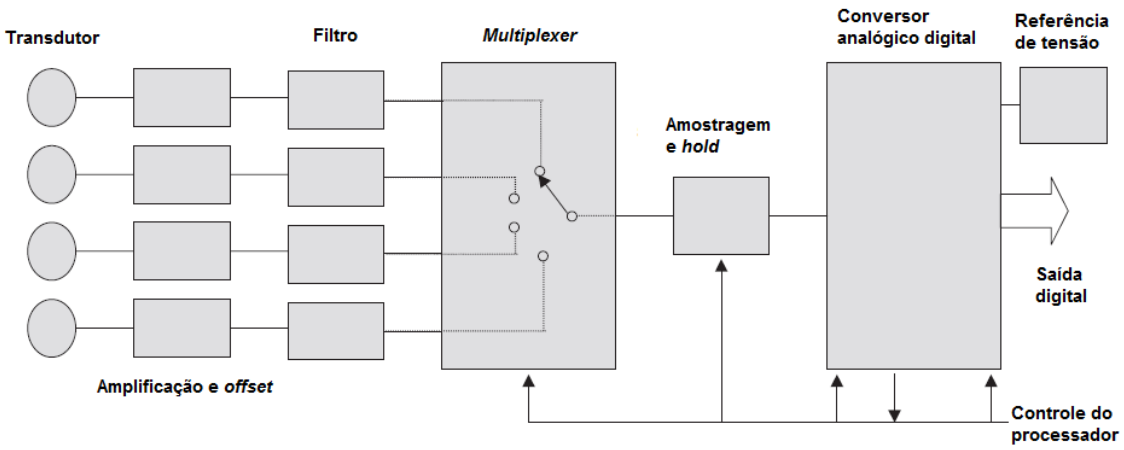


Figura 7: Estrutura de um sistema de aquisição de dados de quatro canais.

Um sistema de aquisição de dados é composto por:

- Transdutor: gera um sinal elétrico, de tensão ou corrente, proporcional à variável física sendo medida;
- Amplificação e *offset*: aplica um ganho e um *offset* ao sinal gerado pelo transdutor, a fim de aproveitar ao máximo a faixa de entrada do conversor analógico digital;
- Filtro: remove componentes indesejadas do sinal, geralmente para evitar *aliasing*;
- *Multiplexer*: seleciona qual canal está conectado ao conversor;
- Amostragem e *hold*: obtém uma amostra do sinal e mantém esse valor durante a conversão;
- Conversor analógico digital: faz a conversão do sinal analógico, em sua entrada, para uma quantidade digital, em sua saída;
- Referência de tensão: serve como referência fixa para o conversor;

O *multiplexer*, a amostragem e *hold*, e o conversor analógico digital são geralmente controlados pelo processador, que determina qual canal será utilizado, qual o tempo de amostragem, e quando deve ser iniciada a conversão. Por sua vez, o conversor envia a saída digital, que é o resultado da conversão, ao processador.

4.3 Conversor analógico digital

A tarefa do conversor analógico digital é determinar um número digital que seja equivalente à tensão de entrada. Vários tipos de conversores já foram desenvolvidos, cada qual com diferentes aplicações em mente. Alguns, como o conversor de rampa dupla, são lentos, mas possuem grande precisão, e são usados geralmente para medidas de precisão, como em multímetros digitais. Outros, como o conversor do tipo *flash*, são rápidos, mas de pouca precisão, e são usados para converter sinais de alta velocidade, como vídeo ou radar. Outros, ainda, como o conversor de aproximações sucessivas, são de média velocidade e média precisão, e são úteis para aplicações de uso geral.

A figura a seguir ilustra o funcionamento de um conversor analógico digital de n bits, com a entrada analógica variando de 0 a V_{max}, e a saída digital variando de 0 a $(2^n - 1)$, mostrada em binário:

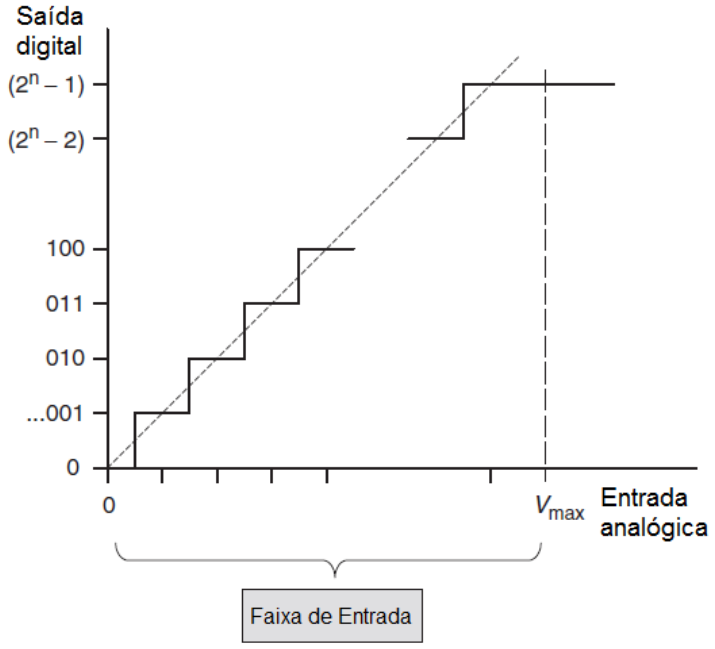


Figura 8: Relação entre a saída digital e a entrada analógica de um conversor analógico digital de n bits.

Ou seja, para cada valor da tensão analógica de entrada, o conversor irá associar um número digital que mais se aproxima do valor, levando em conta a tensão de referência e a resolução do conversor.

Muitos conversores tem a faixa de entrada entre 0 V e 5 V. Outros, entretanto, podem ter uma faixa bipolar, aceitando tensões positivas e negativas, por exemplo, entre +5 V e -5 V. Em todo caso, a faixa de entrada, V_r, será a diferença entre a maior e a menor tensão de entrada. Essa faixa geralmente está relacionada com a referência de tensão do conversor.

Pode-se notar, intuitivamente, pela figura, que quanto maior o número de bits, maior será o número de níveis digitais, e melhor será a conversão. A resolução de um conversor pode ser calculada como $\frac{V_r}{2^n}$, onde V_r é a faixa de entrada, e n é o número de bits do conversor. A resolução indica a menor variação de tensão que pode ser percebida pelo conversor.

Para o melhor aproveitamento do conversor, o ideal é que o sinal a ser convertido use toda a faixa de entrada. Mas nem sempre os transdutores geram sinais compatíveis diretamente com a entrada do conversor. Nesse caso, é necessário aplicar um ganho e/ou *offset* ao sinal, de modo a adequá-lo. Por exemplo, para obter um melhor resultado com um transdutor que gera um sinal entre 100 mV e 200 mV, utilizando um conversor cuja faixa de entrada varia entre 0 V e 5V, aplica-se um ganho de 50, fazendo o sinal variar entre 5 V e 10 V, e em seguida aplica-se um *offset* de – 5 V, resultando em um sinal que variar entre 0 V e 5 V, aproveitando ao máximo a resolução do conversor.

Nos circuitos comerciais os blocos responsáveis pela seleção de canal e pela amostragem e *hold* geralmente são incluídos junto ao próprio conversor. A interface digital com o processador pode ser serial ou paralela.

Um fator que tem grande peso na qualidade da conversão é a tensão de referência, pois o sinal será convertido usando essa tensão como referência. Assim, essa tensão deve ser estável, de boa precisão, e de valor conhecido, para garantir uma boa medição.

4.4 Aliasing

Quando um sinal contínuo, como é o caso dos sinais analógicos gerados pelos transdutores, é transformado em um sinal discreto no tempo pode ocorrer o *aliasing*. O

aliasing ocorre quando um sinal discreto no tempo pode representar diferentes sinais contínuos no tempo, que possuem frequências distintas. Pode-se entender o *aliasing* como uma ambiguidade no domínio da frequência.

Para ilustrar esse efeito, consideremos um sinal discreto no tempo, composto por:

$$x(0) = 0$$

$$x(1) = 0{,}866$$

$$x(2) = 0{,}866$$

$$x(3) = 0$$

$$x(4) = -0{,}866$$

$$x(5) = -0{,}866$$

$$x(6) = 0$$

Na figura a seguir temos esses pontos representados em um gráfico, e dois senos de diferentes frequências que podem representar o sinal contínuo que originou os pontos discretos:

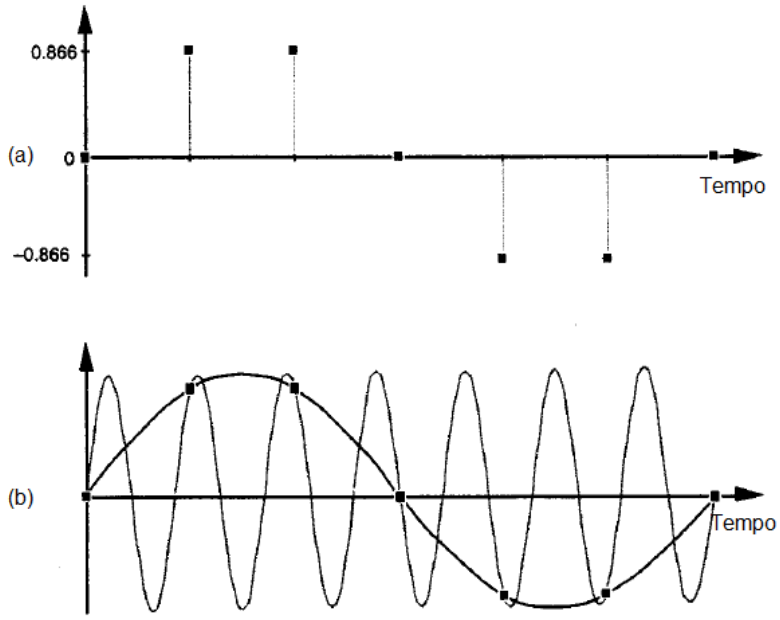


Figura 9: Ambiguidade da frequência: (a) valores discretos no tempo; (b) dois senos diferentes que passam pelos pontos discretos.

Apenas com os pontos discretos fornecidos não é possível determinar qual frequência corresponde ao sinal original. Para evitar o problema de *aliasing*, utiliza-se uma frequência de amostragem, ou seja, a frequência com que são feitas aquisições de dados, que é maior ou igual ao dobro da maior frequência de interesse no sinal analisado. Esse requisito provém do teorema de Nyquist. Para evitar que componentes indesejáveis sejam obtidas durante a aquisição, pode-se usar filtros que removam essas componentes do sinal analógico. Por exemplo, para garantir a condição do teorema de Nyquist pode-se usar um filtro passa baixa com frequência de corte igual à maior frequência de interesse do sinal, usando uma taxa de amostragem que é o dobro dessa frequência.

5 USB

5.1 Sobre

USB (*Universal Serial Bus*, barramento serial universal) é uma especificação de comunicação serial utilizada para interligar dispositivos de diversos tipos, criada, inicialmente, com o propósito de padronizar a conexão de periféricos a computadores pessoais. A especificação é mantida pelo *USB Implementers Forum*, entidade responsável pelo desenvolvimento e por apoiar o avanço e adoção da tecnologia. O conselho de diretores é composto, atualmente, pelas seguintes empresas: *HP*, *Intel*, *LSI*, *Microsoft*, *Renesas Electronicos*, e *ST-Ericsson*.

A especificação descreve as características do barramento, a definição do protocolo, os tipos de transações, o gerenciamento do barramento, e a interface de programação necessária para projetar e construir sistemas e periféricos que são compatíveis com o padrão. Em contraste, outros padrões de comunicação serial, como o RS-232C, definem apenas as características elétricas da comunicação e da codificação dos dados, sem definir um protocolo específico para comunicação.

Com o avanço do padrão USB, alguns padrões de comunicação se tornaram obsoletos. Pode-se citar como exemplo: conector PS/2 (mouse, teclado), porta paralela (impressora, scanner), porta serial (mouse), *game port* (*joystick*, *joypad*). Alguns dispositivos que antes eram exclusivamente internos aos computadores agora possuem versões externas, como placas de som, rede, e modens.

5.2 Vantagens e desvantagens

O padrão USB trouxe grandes vantagens para o usuário final. A facilidade de uso é uma das principais vantagens:

- Interface única: vários dispositivos de diferentes funções são ligados utilizando um mesmo tipo de cabo e conector;

- Configuração automática: ao se plugar um dispositivo, o sistema operacional automaticamente carrega os drivers necessários. Alguns dispositivos podem exigir a instalação de drivers, mas o processo deve ser feito apenas na primeira conexão;
- Fácil de conectar: um computador comum possui várias portas USB, e com o uso de hubs é possível aumentar esse número de forma simples e fácil;
- Cabos convenientes: os conectores USB são pequenos, se comparados a outros padrões, como o RS-232C.
- *Hot pluggable*: os dispositivos podem ser conectados e desconectados mesmo com o computador ligado;
- Sem alimentação externa: vários dispositivos, que não tenham consumo elevado, podem utilizar a alimentação fornecida diretamente através do cabo USB, sem a necessidade de fontes externas;

A comunicação USB é bastante confiável. A especificação garante um ambiente de baixo ruído elétrico, ao especificar as características dos transmissores, receptores, e dos cabos, eliminando a maior parte dos ruídos que causam erros nas transmissões. O protocolo definido pela especificação também suporta a detecção de erros e retransmissão dos dados corrompidos, feito diretamente por *hardware*, sem a necessidade da intervenção do *software* ou do usuário.

A especificação foi feita com economia em mente. Como o computador fornece a maior parte da inteligência para controlar a interface, os componentes necessários aos dispositivos USB são baratos. A economia de energia, que é bastante importante nos dias de hoje, devido tanto aos dispositivos móveis, que utilizam baterias, quanto à preocupação com o meio ambiente, foi levada em conta na especificação, de modo que um dispositivo possa reduzir seu consumo de energia e ainda se manter apto à comunicação.

A principal limitação da especificação diz respeito à distância: para conseguir atender aos requisitos da especificação, os cabos não podem ser muito longos. Isso provém do fato do padrão ter sido desenvolvido principalmente com a conexão de periféricos em mente. Outros padrões, como RS-232C, RS-485, IEEE-1394b, e Ethernet permitem cabos muito mais longos.

Outra limitação é que a conexão USB deve ser feita entre um *host*, geralmente um computador, e um dispositivo. Conexões entre dispositivos, ou entre hosts, não são possíveis.

Uma solução parcial para esse problema foi dada com a criação do USB OTG (*On-The-Go*). Um dispositivo que suporta USB OTG pode funcionar tanto como dispositivo, quando conectado a um host, quanto como um host limitado, quando conectado a um dispositivo. O padrão USB 3.0 definiu um novo cabo, para a velocidade *SuperSpeed*, que permite a conexão entre dois hosts.

Sem suporte a *broadcasting*: o padrão não suporta enviar dados simultaneamente para mais de um dispositivo. O *host* deve enviar os dados individualmente para cada dispositivo. Esse recurso é suportado pelo IEEE-1394b e Ethernet.

5.3 Detalhes técnicos

A topologia, ou o arranjo das conexões, do barramento USB é do tipo rede em estrela com camadas. No centro de cada estrela fica um *hub* e cada conexão é uma ponta da estrela. O *hub* raiz é o *host*. Cada *hub* externo possui uma porta usada para se comunicar com o *host* e duas ou mais portas para se comunicar com dispositivos. Essa topologia descreve apenas a conexão física dos dispositivos, pois a conexão lógica, que é o que importa para o *software* e para o *firmware*, não faz distinção de como o dispositivo está conectado ao *host*. A figura a seguir exemplifica a topologia em questão:

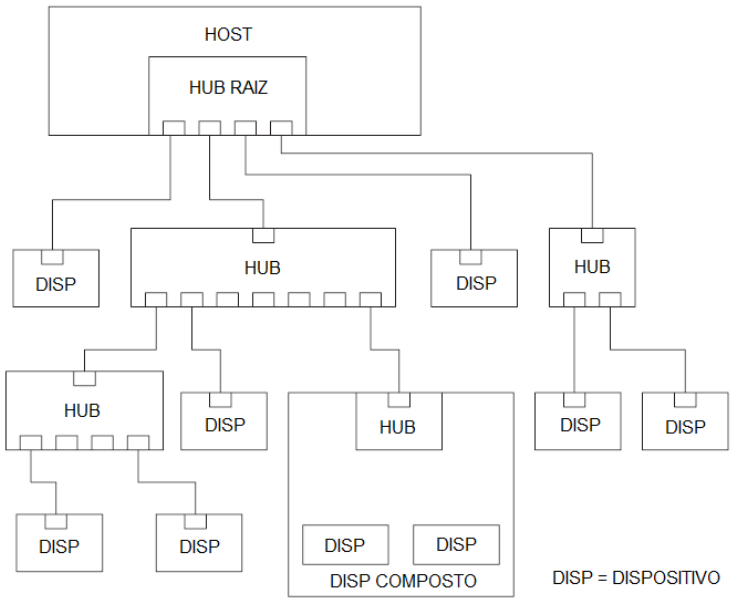


Figura 10: Topologia de rede em estrela com camadas.

A comunicação USB acontece através de canais lógicos de comunicação que são chamados de *endpoints*. Cada *endpoint* é caracterizado por um número e uma direção. O número pode variar de 0 a 15, e a direção pode ser do host para o dispositivo ou do dispositivo para o host. Isso totaliza um máximo de 32 canais lógicos possíveis, sendo 16 do host para o dispositivo e 16 do dispositivo para o host. Existem quatro tipos de *endpoints*:

- *Control*: único tipo de *endpoint* que exige que sejam definidos dois *endpoints* de mesmo número, mas de direções opostas. Ele é baseado na troca de mensagens entre o *host* e o dispositivo. As mensagens são enviadas pelo host pelo endpoint que vai do *host* para o dispositivo, e o dispositivo responde pelo endpoint de mesmo número, mas que vai do dispositivo para o *host*. O *endpoint* de número 0 deve existir e ser obrigatoriamente desse tipo, pois é usado para identificar e configurar o dispositivo. Transfere poucos dados, não possui garantia de taxa de transferência de dados, mas apresenta correção de erros.
- *Bulk*: capaz de transmitir uma quantidade maior de dados, com correção de erros, mas não possui garantia de taxa de transferência. Usado geralmente em dispositivos de armazenamento, impressoras, e scanners.

- *Interrupt*: transmissão periódica de poucos dados, com correção de erros, mas sem taxa de transferência garantida. Geralmente usado em dispositivos como teclado e mouse.
- *Isochronous*: transmite uma grande quantidade de dados com garantia de taxa de transmissão, mas não possui correção de erros. Usado principalmente para *streaming* de áudio e vídeo.

Os dispositivos USB possuem descritores que são consultados pelo sistema operacional, através do *endpoint* 0, para determinar como o dispositivo deve ser usado. Dentre as informações contidas nos descritores temos o *Vendor ID* e o *Product ID*, que identificam, respectivamente, o fabricante do dispositivo e o dispositivo em si. Ambos são valores inteiros de 16 bits. Os Vendor IDs são atribuidos pela *USB Implementers Forum* às empresas, mediante uma taxa de licenciamento. O dispositivo pode conter, opcionalmente, um nome descritivo do fabricante, do dispositivo, e um número serial. O descritor indica a classe e os *endpoints* que são usados pelo dispositivo.

Com a função principal de conectar dispositivos a computadores, o protocolo USB foi desenvolvido com algumas classes de dispositivos genéricas, que são suportadas pela maioria dos sistemas operacionais sem a necessidade de instalação de *drivers* de terceiros. Por exemplo, existem classes definidas pela especificação para: dispositivos de áudio (placa de som), dispositivos de comunicação (Ethernet, Wi-Fi, modem), dispositivos de interface humana (teclado, mouse, controles), impressoras, dispositivos de armazenamento, hub USB, dispositivos de vídeo (webcam), entre outros. Todas essas classes definem um protocolo genérico adequado para o tipo de dispositivo em questão.

Com o intuito de manter o protocolo USB genérico o bastante, para suportar dispositivos não previstos na especificação, existe uma classe que indica para o sistema operacional que o dispositivo necessita de um driver do fabricante. Essa classe é livre para definir o seu próprio protocolo de comunicação.

As classes definidas pela especificação para funções específicas, como a classe para dispositivos de interface humana, possuem em sua especificação a descrição do seu comportamento: quais requisições devem ser atendidas, como devem ser atendidas, quais *endpoints* são obrigatórios e, caso existam, quais são opcionais, como devem ser

transferidos os dados, entra outros detalhes. A classe para dispositivos que necessitam de drivers do fabricante não especifica a configuração dos *endpoints*, apenas que o *endpoint* 0 deve ser para operações de controle.

A classe dos dispositivos de interface humana (HID) define uma classe genérica para qualquer dispositivo que possa servir de interface ente o usuário e o computador. Os dispositivos que seguem a especificação HID usam relatórios para transmitir dados entre o dispositivo e o computador, e vice versa. Existem três tipos de relatórios: *input*, que enviam comandos do usuário para o computador; *output*, que envia comandos do computador para o usuário; e *feature*, que serve para configurar opções, caso existam, do dispositivo. O significado de cada item em um relatório é definido pelos descritores de relatório.

Nos descritores de relatório são especificadas todas as entradas, saídas e configurações disponíveis no dispositivo. Para entrada há, por exemplo, botões e valores analógicos. Para saída há, por exemplo, intensidade do *force feedback ou um indicador luminoso*. Para configuração há, por exemplo, a intensidade máxima do *force feedback*.

Os dispositivos HID não necessitam de *drivers*, pois possuem suporte nativo dos sistemas operacionais. O formato dos dados depende apenas do que é especificado nos descritores de relatórios. Essas características fazem com que a classe HID seja utilizada para obter troca de dados genéricos, ou seja, que não são de interface humana, entre um software e um dispositivo, sem a necessidade do desenvolvimento de drivers. Um dos *bootloaders* USB fornecido pela Microchip utiliza esse recurso, para facilitar o uso do programa pelo usuário final, sem a necessidade de instalar *drivers*.

6 Microcontrolador PIC18F4550

6.1 Microcontroladores

Os microcontroladores (ou MCU) são pequenos dispositivos basicamente constituídos de uma CPU (*Central Processing Unit*, ou Unidade Central de Processamento, em português), memória (para dados e programas) e periféricos (portas E/S, I^2C, SPI, USART, entre outros). Suas dimensões reduzidas são resultantes da alta capacidade de integração, em que milhões de componentes são inseridos em uma única pastilha de silício pela técnica de circuitos integrados (CIs). Eles estão presentes na maioria dos equipamentos digitais, como celulares, MP3 *player*, impressoras, robótica, instrumentação, entre outros.

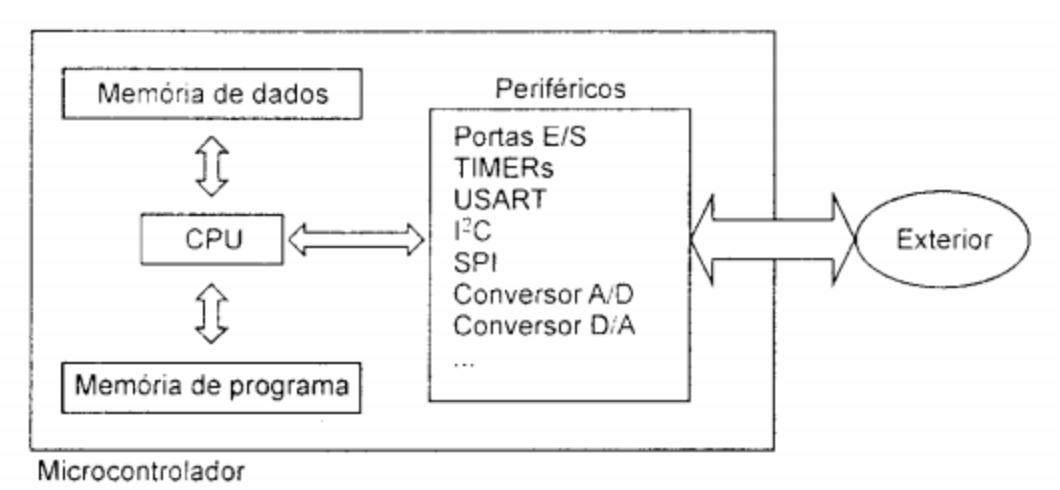


Figura 11 - Diagrama resumido de um microcontrolador.

As duas principais arquiteturas de microcontroladores são *Harvard* e *Von-Neumann*. A arquitetura *Harvard* é caracterizada pela existência de um barramento para o acesso à memória de dados e outro para a memória de programa, resultando em um aumento do fluxo de dados, enquanto na arquitetura *Von-Neumann* as memórias de dados e programas compartilham o mesmo barramento, limitando a banda de operação.

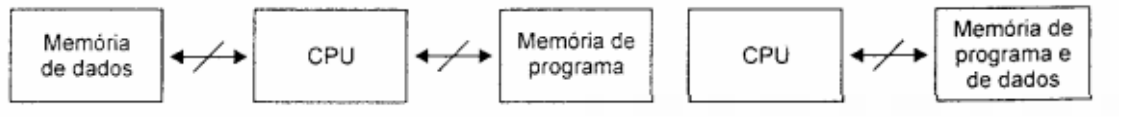


Figura 12 - Diagrama de arquitetura Harvard (à esquerda) e arquitetura Von-Neumann (à direita).

Em geral, as memórias de programa presentes nos microcontroladores são do tipo EEPROM (*Eletrically Erasable Programmable Read Only Memory*), Flash, ROM (*Read Only Memory*), EPROM (*Erasable Programmable Read Only Memory*) ou OTP (*One Time Programmable*). Elas são responsáveis pelo armazenamento do programa, o que significa que sua capacidade de armazenamento deve ser suficiente para reter todo o código desejado. Essas memórias são do tipo não volátil, portanto o código de programa armazenado não é perdido, caso o circuito não esteja sendo alimentado. Vejamos as principais características das memórias citadas.

- ROM: não permite que o conteúdo seja alterado pelo usuário. Ela aceita somente a leitura do conteúdo, o qual foi gravado pelo fabricante. Microcontroladores dotados de memória de programa do tipo ROM normalmente apresentam um baixo custo com relação às memórias FLASH, EPROM e OTP, e são recomendados quando o código do programa não apresenta erros e há necessidade de grande quantidade.
- EPROM: pode ser apagada e/ou programada muitas vezes, porém o conteúdo da memória é apagado através da exposição da janela de quartzo à luz ultravioleta, cujo processo de fabricação apresenta um custo elevado, se comparados com os outros tipos de memória.
- OTP: tolera somente uma gravação. Esse tipo de memória apresenta o menor custo se for comparado com as memórias programáveis (EPROM e FLASH).
- EEPROM: de funcionamento semelhante ao de uma EPROM, essa memória é escrita ou apagada eletricamente.
- FLASH: é a memória mais flexível entre todos os outros tipos de memória de programa, pois pode ser apagada eletricamente e reprogramada 100000 a 1000000 de vezes, dependendo da tecnologia empregada na fabricação desse componente.

Outro tipo de memória existente é a de dados, definida como memória RAM (*Random Access Memory*). Ela é volátil e armazena as variáveis e constantes do sistema. O

conteúdo presente nesse tipo de memória é perdido sempre que a alimentação é cortada. Isso implica que os valores das variáveis devem ser carregados sempre que o sistema for iniciado.

Pinos I/O (*input/output*, ou entrada/saída) digitais estão presentes em todos os microcontroladores. Por meio deles, o MCU se comunica com o mundo exterior, ou seja, é por intermédio destes que o MCU aciona um relé, led, motor, etc. O sentido do fluxo de dados de um pino I/O pode ser definido como entrada ou saída. Se o pino for definido como saída, então ele normalmente será utilizado para controlar periféricos; caso contrário, se for definido como entrada, o dispositivo passa a ler o sinal presente no pino. Chamamos de porta um conjunto de pinos relacionados a ela (exemplo: porta X e pinos X.1, X.2, X.3, X.4, X.5, etc).

Alguns periféricos como conversores A/D, USART (*Universal Synchronous Asynchronous Receiver Transmitter*), TIMER, SPI (*Serial Peripheral Interface*) e I^2C (*Inter-Integrated Circuit*) são muito comuns nos microcontroladores, no entanto existem MCUs mais robustos que, além desses periféricos, também apresentam outros mais específicos, como controladores de LCD, USB (*Universal Serial Bus*), RTC (*Real-Time Clock*), rede CAN, etc.

A velocidade de processamento do microcontrolador está diretamente relacionada com a frequência de *clock*. Quanto maior a frequência de trabalho maior será a capacidade de processamento, assim como o consumo de energia. Essa frequência pode ser gerada por um oscilador interno, que normalmente é um circuito RC, ou então por um cristal de quartzo ou um ressonador conectado externamente. Osciladores internos do tipo RC são normalmente utilizados quando não há grande necessidade de precisão de *clock*; caso contrário, utiliza-se o cristal.

6.2 PIC18F4550

O PIC um microcontrolador fabricado exclusivamente pela *Microchip Tecnology*, e divide-se em várias famílias. Com arquitetura de 8 *bits* tem-se, por ordem crescente de performance e dimensão, as famílias PIC10, PIC12, PIC16 e PIC18. Com arquitetura de 16 *bits*

há as famílias PIC24F e PIC24H e os processadores digitais de sinais (DSPs) dsPIC30 e dsPIC33.

O microcontrolador PIC18F4550, é construído com base na arquitetura *Harvard* com instruções do tipo RISC (Computador com Conjunto Reduzido de Instruções). É um dispositivo de 8 *bits* dotado de 32 KB de memória de programa e 2 KB de memória RAM. Esse dispositivo pode ser alimentado com tensões entre 4 V e 5,5 V, além de operar em frequências de até 48 MHz (12 MIPS – milhões de instruções por segundo). Ele pode ser alimentado diretamente por um oscilador de 48 MHz ou por um cristal associado com o bloco PLL, capaz de multiplicar a frequência do cristal até os limites de 48 MHZ. Além disso, possui um oscilador interno de 8 MHz, que pode ser derivado em 8 MHz, 4 MHz, 2 MHz, 1 MHz, 500 KHz, 250 KHz, 125KHz e 31 KHz.

Esse modelo possui 40 pinos, dos quais 35 podem ser configurados como I/O, e diversos periféricos, tais como memória EEPROM (FLASH) de 256 *bytes*, um módulo CCP (*Capture-Compare-PWM*) e ECCP (*Enhanced Capture-Compare-PWM*), um módulo SPI e I^2C, conversor A/D de 13 canais com 10 *bits* de resolução e tempo de aquisição programável, dois comparadores analógicos, uma comunicação EUSART, um TIMER de 8 *bits* (TIMER2) e três de 16 *bits* (TIMER0, TIMER1 e TIMER3), um módulo de detecção de alta/baixa voltagem (HLVD), além de ter um módulo USB 2.0 capaz de operar no modo *low-speed* (1,5 Mbps) ou *full-speed* (12 Mbps).

A Figura 13 ilustra a distribuição dos pinos e respectivas funções do microcontrolador PIC18F4550. Uma descrição completa da função de cada pino pode ser encontrada no *datasheet* completo do componente (Datasheet PIC18F4550, 2009).

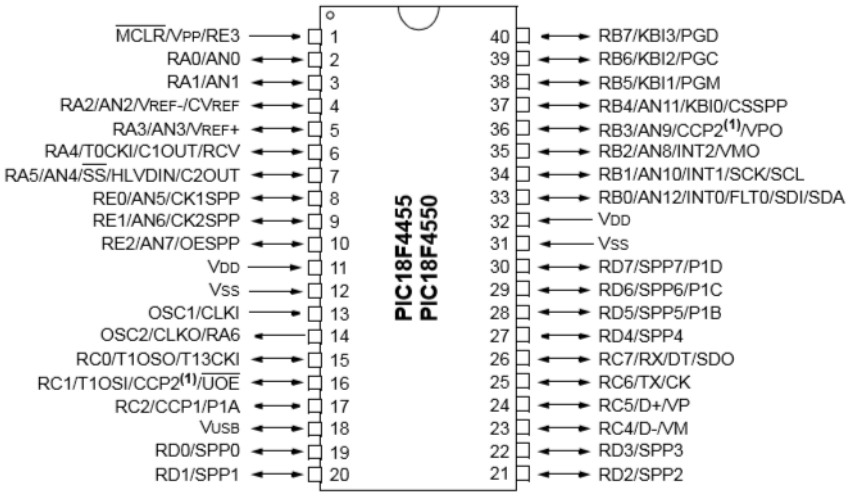


Figura 13 - Pinagem do PIC18F4550.

7 A linguagem C

7.1 Uma visão geral

A linguagem C foi inventada e implementada incialmente por Dennis Ritchie em um DEC PDP-11 que utilizava sistema operacional UNIX. C é o resultado de um processo de desenvolvimento que começou com uma linguagem mais antiga, chamada BCPL, que influenciou uma linguagem chamada B. Na década de 1970, a linguagem B levou ao desenvolvimento de C.

Com a popularidade dos microcomputadores, um grande número de implementações de C foi criado. Porém, por não existir nenhum padrão, havia discrepâncias. Para remediar essa situação, o ANSI (*American National Stantdards Institute*) estabeleceu, em 1983, um comitê para criar um padrão que definiria de uma vez por todas a linguagem C. Assim, todos os principais compiladores já implementaram o padrão C ANSI.

C é uma linguagem chamada de médio nível para computadores. Isso não significa, no entanto, que seja menos poderosa, difícil de usar ou menos desenvolvida que uma linguagem de alto nível como BASIC e Pascal, tampouco implica que C seja similar à linguagem *assembly* e seus problemas correlatos aos usuários. C é tratada como linguagem de médio nível porque combina elementos de linguagens de alto nível com a funcionalidade da linguagem *assembly*.

Como linguagem de médio nível, C permite a manipulação de *bits*, *bytes* e endereços – os elementos básicos com os quais um computador ou microcontrolador funciona.

Todas as linguagens de programação de alto nível suportam o conceito de tipos de dados. Embora C tenha cinco tipos de dados internos, ela não é considerada uma linguagem rica em tipos de dados como Pascal e Ada. C permite quase todas conversões de tipos de dados. Por exemplo, os tipos caractere e inteiro podem ser livremente misturados na maioria das expressões, não sendo efetuada nenhuma verificação – a não ser por recursos de compilador – no tempo de execução, sendo estes de responsabilidade do programador.

Outro aspecto importante sobre a linguagem C é que ela tem apenas 32 palavras-chave (27 originárias do padrão original, mais 5 adicionadas pelo comitê ANSI de padronização), que são os comandos que compõem a linguagem C. As linguagens de alto nível tipicamente tem várias vezes esse número de palavras reservadas. Como comparação, a maioria das versões de BASIC possuem bem mais de 100 palavras reservadas.

7.2 Linguagem estruturada

C é uma linguagem de programação estruturada. A característica especial de uma linguagem estruturada é a compartimentalização do código e dos dados. Trata-se da habilidade de uma linguagem seccionar e esconder do resto do programa todas as informações necessárias para se realizar uma tarefa específica. Com o uso de variáveis locais, é possível escrever sub-rotinas de forma que os eventos que ocorrem dentro delas não causem nenhum efeito inesperado nas outras partes do programa. Essa capacidade permite que programas em C compartilhem facilmente seções de código, sendo possível a criação de funções compartimentalizadas.

Uma linguagem estruturada permite muitas possibilidades na programação. Ela suporta, diretamente, diversas construções de laços (*loops*), como *while*, *do-while* e *for*. Em uma linguagem estruturada, o uso de *goto* é proibido ou, no mínimo, desencorajado. Uma linguagem estruturada permite a inserção de sentenças em qualquer lugar de uma linha e não exige um conceito rigoroso de campo, como em FORTRAN.

7.3 Forma básica de um programa C

Todo programa em C consiste em uma ou mais funções. A única função que necessariamente precisa estar presente é a denominada “main()”, que é a primeira função a ser chamada quando a execução do programa começa. Em um código C bem escrito, “main()” contém, em essência, um esboço do que o programa faz. O esboço é composto pela

chamada de funções. Embora a função principal não seja tecnicamente parte da linguagem C, é tratada como se fosse.

A forma geral de um programa C é ilustrada na Figura 14, em que $f1()$ até $fN()$ representam funções definidas pelo programador.

```
declarações globais

tipo devolvido main(lista de parâmetros)
{
  seqüência de comandos
}

tipo devolvido f1(lista de parâmetros)
{
  seqüência de comandos
}

tipo devolvido f2(lista de parâmetros)
{
  seqüência de comandos
}
.
.
.
tipo devolvido fN(lista de parâmetros)
{
```

Figura 14 - A forma geral de um programa em C.

7.4 Tipos de dados

A linguagem C define cinco tipos de dados básicos: caractere (*char*), inteiro (*int*), ponto flutuante (*float*), ponto flutuante de dupla precisão (*double*) e sem valor (*void*). Todos os outros tipos de dados em C são baseados em um desses tipos básicos. O tipo *void*, especificamente, declara explicitamente uma função que não retorna valor algum ou cria ponteiros genéricos.

Todos os tipos de dados definidos no padrão ANSI são exibidos abaixo.

Tipo	Tamanho em Bytes	Faixa Mínima
char	1	-127 a 127
unsigned char	1	0 a 255
signed char	1	-127 a 127

int	4	-2.147.483.648 a 2.147.483.647
unsigned int	4	0 a 4.294.967.295
signed int	4	-2.147.483.648 a 2.147.483.647
short int	2	-32.768 a 32.767
unsigned short int	2	0 a 65.535
signed short int	2	-32.768 a 32.767
long int	4	-2.147.483.648 a 2.147.483.647
signed long int	4	-2.147.483.648 a 2.147.483.647
unsigned long int	4	0 a 4.294.967.295
float	4	Seis dígitos de precisão
double	8	Dez dígitos de precisão
long double	10	Dez dígitos de precisão

7.5 Compiladores x interpretadores

Os termos compiladores e interpretadores referem-se à maneira como um programa é executado. Existem dois métodos gerais pelos quais um programa pode ser executado. Em teoria, qualquer linguagem de programação pode ser compilada ou interpretada, mas algumas geralmente são executadas de uma maneira ou outra. A maneira pela qual um programa é executado não é definida pela linguagem em si. Interpretadores e compiladores são simplesmente programas sofisticados que operam sobre o código-fonte de um programa.

Um interpretador lê o código-fonte de um programa uma linha por vez, executando a instrução específica contida nessa linha. Um compilador lê o programa inteiro e converte-o em um código-objeto, que é uma tradução do código-fonte em uma forma que o computador possa executar diretamente. O código-objeto é também conhecido como código binário ou código de máquina. Uma vez que o programa tenha sido compilado, uma linha do código-fonte, mesmo que seja alterada, não é mais importante na execução do seu programa.

Quando um interpretador é usado, deve estar presente toda vez que o programa é executado, para examinar uma linha por vez para correção e então executá-la. Esse processo lento ocorre cada vez que o programa for executado. Um compilador, ao contrário, converte seu programa em um código-objeto que pode ser executado diretamente pelo computador. Como o compilador traduz o programa de uma só vez, tudo o que é necessário fazer é executar o programa diretamente. Assim, o tempo de compilação só é gasto uma vez, enquanto o código interpretado incorre neste trabalho adicional em cada execução.

7.6 Bibliotecas

Em princípio, é possível criar um programa útil e funcional que consista apenas nos comandos realmente criados pelo programador. Isso é muito raro, porém, porque C, dentro da atual definição da linguagem, não oferece nenhum método de executar operações de entrada/saída (I/O, ou E/S). Como resultado, a maioria dos programas inclui chamadas a várias funções contidas na biblioteca C padrão.

Todo compilador C vem com uma biblioteca C padrão de funções que realizam as tarefas mais comuns. No entanto, cada compilador conterá muitas outras funções, eventualmente dedicadas a alguma aplicação dedicada à qual se aplica o compilador. Por exemplo.

Muitas das funções utilizadas nos programas mais comuns estão na biblioteca padrão. Elas agem como blocos básicos que podem ser combinados. Ainda, funções específicas ou que sejam utilizadas muitas vezes também podem ser colocadas em bibliotecas. Alguns compiladores permitem que funções sejam adicionadas à biblioteca padrão, mas sempre é possível a criação de uma biblioteca adicional. De qualquer forma, o código estará lá para ser usado repetidamente.

7.7 MPLAB C18

O compilador C da Microchip, C18, é um compilado gratuito para a família de microcontroladores PIC18 em padrão ANSI. Um compilador para utilização em microcontroladores é um compilador como outro, um programa que roda sobre o código para sua interpretação e transformação em linguagem de máquina. Algumas diferenças quanto à funcionalidade e disponibilidade de recursos, no entanto, são notáveis, e devidas às diferenças entre os dispositivos de *hardware* a que um programa-objeto se destina.

Assim, o compilador C18 é um compilador C específico para a família de dispositivos PIC18, microcontroladores programáveis de propósito geral, e, então, otimizado para suprir, exclusivamente, as necessidades desses dispositivos. Então, quando comparado com um compilador C tradicional, este tem algumas limitações. Funções recursivas não estão disponíveis, uma vez que os dispositivos PIC não possuem uma pilha para armazenamento de variáveis, e também devido à maneira como o compilador otimiza o código.

De qualquer forma, o compilador é capaz de implementar construções normais em C, operações de entrada e saída, e operações *bit* a *bit*. Todos os tipos de dados suportados pela linguagem são suportados pelo compilador, além de ponteiros para vetores constantes, operações com ponto flutuante e vetores de *bits*.

8 LabVIEW

8.1 O que é *LabVIEW*

Inicialmente, o *LabVIEW* era somente uma linguagem de programação gráfica desenvolvida com o propósito de facilitar a coleta de dados de instrumentos de laboratório, a partir de sistema de aquisição de dados. Para tais propósitos, o *LabVIEW* foi sempre fácil de usar, uma vez que criar programas de computador ligados por "fios", conectores, e com uma integrada e intuitiva interface gráfica, é bastante simples. Ainda, isso faz muito mais simples e rápida tarefas de aquisição de dados.

Assim, *LabVIEW* (acrônimo para *Laboratory Virtual Instrument Engineering Workbech*) é uma linguagem de programação gráfica, originária da *National Instruments*. A primeira versão surgiu em 1986 para *Macintosh* e atualmente existem ambientes de desenvolvimento para os sistemas operacionais *Windows*, *Linux* e *Solaris*, além de plataformas móveis de integração e a versão gratuita, *RunTime Engine*, somente para execução de programas.

Atualmente, é uma poderosa interface de programação, coleta de dados e controle, considerado um ambiente de desenvolvimento de instrumentação virtual, capaz de realizar tarefas e medições tão precisas – ou mais, dependendo dos sistemas de aquisição – quanto equipamentos dedicados.

8.2 Aplicações

Os principais campos de aplicação do *LabVIEW* são a realização de medições e a automação, sendo as tarefas divididas, basicamente, em:

- Aquisição dos dados a partir de instrumentos;
- Processamento dos dados;
- Análise dos dados (tomada de decisão a partir do processamento);
- Controle de instrumentos e equipamentos.

Para engenheiros, o *LabVIEW* torna possível trazer informações do mundo real, exterior, para dentro de um computador, tomar decisões baseadas na aquisição de dados e enviar os resultados do computador de volta ao mundo real para controlar a operação de um determinado equipamento.

8.3 Programação

A programação é feita de acordo com o modelo de fluxo de dados, o que oferece a esta linguagem diversas vantagens e facilidades para a aquisição de dados e para sua manipulação.

Os programas em *LabVIEW* são chamados *Virtual Instruments* (instrumentos virtuais), ou mais comumente, *VIs*, na sigla em inglês. São compostos pelo painel frontal, que contém a interface, controles, entradas e saídas, como mostrado na Figura 15, e pelo diagrama de blocos, que contém o código gráfico do programa, como mostra a Figura 16. O programa, ao contrário de muitos ambientes de programação "amigáveis" ao usuário, não é processado por um interpretador, mas sim compilado. Deste modo, sua *performance* é comparável à exibida pelas linguagens de programação de alto nível. A linguagem gráfica de programação do *LabVIEW* é chamada "G", nome dado pelo desenvolvedor.

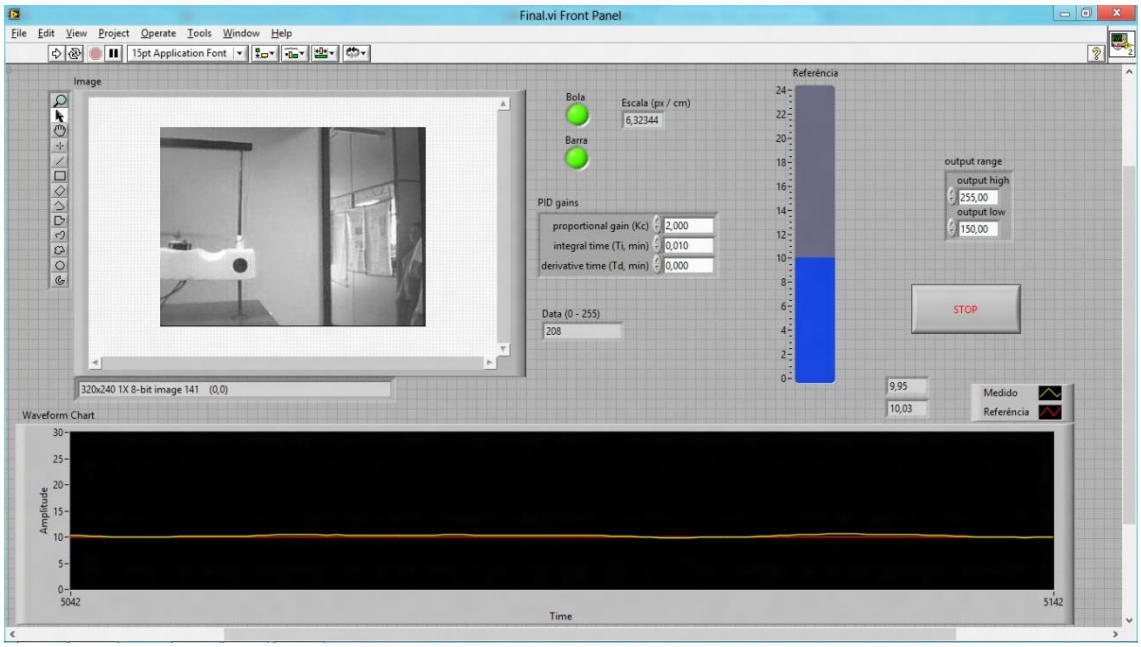


Figura 15 - Típico painel frontal do LabVIEW.

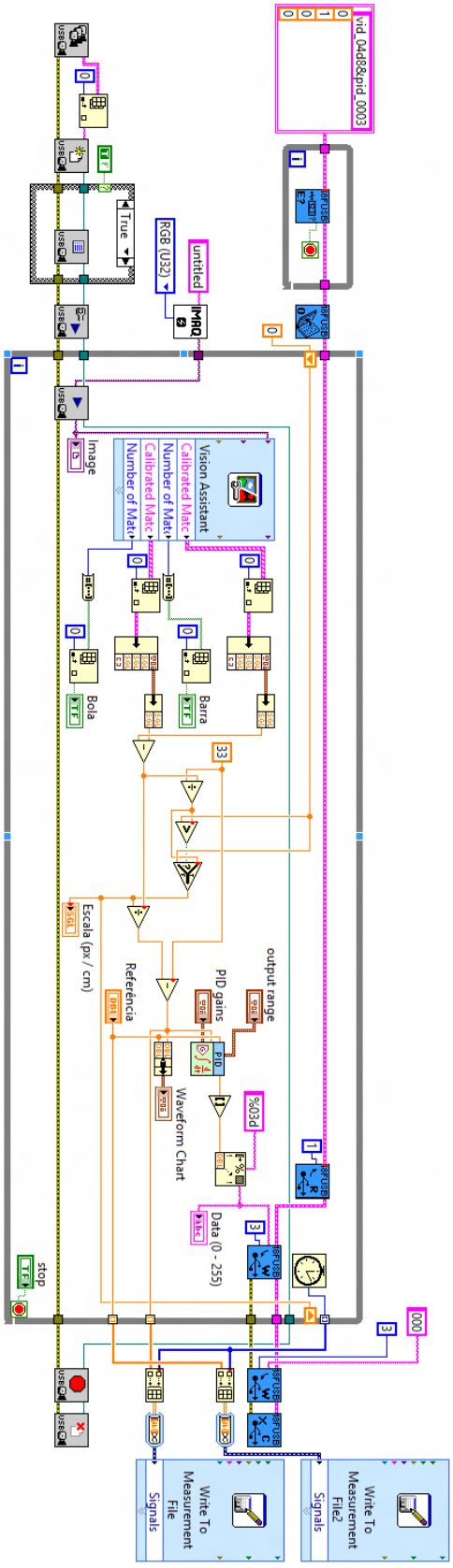


Figura 16 - Exemplo de diagrama de blocos desenvolvido em LabVIEW.

Os blocos de funções são designados por instrumentos virtuais. Isto é assim porque, em princípio, cada programa (*VI*) pode ser usado como subprograma (*subVI*) por qualquer outro ou pode, simplesmente, ser executado isoladamente. Devido à utilização do modelo do fluxo de dados, as chamadas recursivas não são estruturalmente possíveis, podendo-se, no entanto, conseguir esse efeito pela aplicação de algum esforço extra.

O programador liga *Vis* com linhas (*wires* ou fios) de ligação e define, deste modo, o fluxo de dados. As linhas, pela espessura e cor, define o tipo de dado que circula entre os blocos de funções, como ilustra a Figura 17. Cada *VI* pode possuir entradas e/ou saídas. A execução de um *VI* começa quando todas as entradas estão disponíveis; os resultados do processamento são então colocados nas saídas assim que a execução do subprograma tenha terminado. Desta forma, a ordem pela qual as tarefas são executadas é definida em função dos dados. Uma ordem pré-definida (por exemplo, "da esquerda para a direita") não existe.

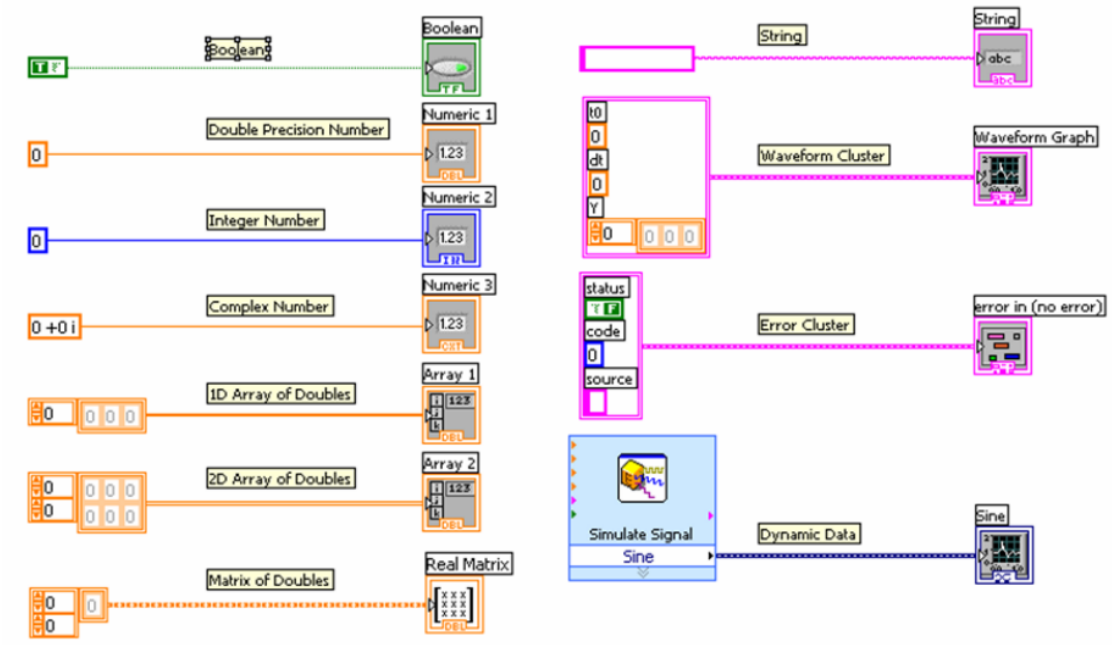


Figura 17 - Tipos de dados manipulados pelo LabVIEW. A composição desses dados pode ser feita através da estrutura do tipo cluster.

Uma importante consequência destas regras é a facilidade com que podem ser criados processos paralelos no *LabVIEW*. Os *subVIs* sem interdependência dos respectivos dados são processados em paralelo.

Os *subVIs* que não possuem entradas são executados no início do programa. Se o subprograma não possuir saídas, os dados resultantes são ignorados ou, então usados pelo exterior: são escritos para o disco rígido ou para a rede, ou enviados para a impressão. Da mesma forma, um *subVI* sem entradas pode receber dados provenientes de aparelhos periféricos ou pode gerar seus próprios dados (um exemplo é um gerador de números aleatórios).

Os subprogramas podem estar interligados com muita complexidade. Muitas das funções próprias do *LabVIEW* são, por sua vez, *VIs* normais, que podem ser modificados pelo programador (o que não é recomendado). Todos os *VIs* se baseiam numa série de funções básicas, chamadas "primitivas", que não podem ser modificadas pelo programador.

Muitos *VIs* e primitivas em *LabVIEW* são polimorfos, ou seja, sua funcionalidade adapta-se aos tipos de dados que recebem. Por exemplo, a função "Build Array" pode ser usada para a criação de quaisquer variáveis, ou seja, de *strings*, de inteiros e também de *arrays* de *clusters*. Também é possível ao programador construir seus próprios *VIs* polimorfos. No fundo, consistem de uma coleção de vários *VIs* com diferentes tipos de dados, entradas e saídas.

Os dados podem ser ligadas ao Painel Frontal através de manipuladores. Por exemplo, a inserção de números pode ser dependente de um manípulo e uma variável de saída *booleana* pode ser realizada por um LED colocado no painel.

O Painel Frontal do *LabVIEW* é um meio confortável para construir programas com uma boa interface gráfica. O programador não necessita de escrever qualquer linha de código, apesar de que essa possibilidade existe, podendo o *VI* receber linhas de código de diversas linguagens comumente utilizadas. A apresentação gráfica dos processos aumenta a facilidade de leitura e de utilização. Uma grande vantagem em relação às linguagens baseadas em texto é a facilidade com que se cria componentes que se executam paralelamente. Em projetos de grande dimensão é muito importante planejar sua estrutura desde o início (como acontece nas outras linguagens de programação).

8.4 Limitações e desvantagens

Por um lado, é muito confortável programar sem código, em princípio, mas não devemos esquecer que no *LabVIEW* é muito importante planejar muito bem o projeto antes de se passar à realização dos *VIs* (em geral, a estruturação prévia do programa é muito mais necessária se comparada a programa feitos em código escrito).

Pequenas mudanças podem obrigar a profundas reestruturações do programa, uma vez que sempre que se insere um novo bloco é necessário voltar e ligar os fios e os símbolos para restabelecer o funcionamento.

Ainda, para evitar confusões de linhas, é habitual introduzir mais variáveis do que aquelas que são estritamente necessárias, diminuindo-se, assim, a velocidade de programação e, mais grave, de execução do código, e contrariando-se, de algum modo, o modelo funcional de fluxo de dados.

9 *Microchip* USB *Framework*

9.1 Microchip Application Libraries

A *Microchip Application Libraries* (MLA) é uma coleção, de bibliotecas e de modelos de projetos de aplicações para os microcontroladores PIC, distribuída gratuitamente. Nem todos os *firmwares* da Microchip, no entanto, são distribuídos nesse pacote; este contém algumas bibliotecas específicas que geralmente podem ser usadas em conjunto. Distribuindo bibliotecas que podem ser usadas em conjunto, exemplos de projetos que utilizam múltiplas bibliotecas e apresentam múltiplos recursos podem, também, ser oferecidos, e acompanham o pacote.

Para prover o máximo de flexibilidade a programadores, usuários e desenvolvedores de soluções, o *framework* é distribuído a partir do código-fonte, o que facilita a customização do *firmware* para cada aplicação.

9.2 USB *Framework*

A *Microchip* possui *software* de suporte à implementação de USB nos microcontroladores PIC de 8 *bits*, 16 *bits* e 32 *bits*. Esse *software* é livre, de código aberto e o pacote acompanha exemplos de projeto com fins de aprendizado.

A família de microcontroladores PIC18 tem suporte a implementação de USB na forma de dispositivos. As famílias PIC24 e PIC32, além do suporte de USB como dispositivos, podem atuar como *host*, e suportam o USB *On-The-Go*. Todas as famílias suportam o modo *full-speed USB* (FS-USB), que opera em 12 Mbps.

A *Microchip* USB *Framework for PIC18, PIC24 & PIC32* é um pacote distribuído com a MLA contendo uma variedade de exemplos de projetos de *firmwares*, além de *drivers* USB e outros recursos para o uso de um computador.

Entre os modelos de projeto, encontram-se *Device CDC demo, Printer demo, bar code Scanner demo, CDC serial emulator, device composite HID and mass storage, generic driver demo, HID mouse example, HID keyboard demo, SD card reader, SD data logger, thumb driver data logger (host)*, além de muitos outros.

9.3 *Driver* genérico USB

O pacote de aplicações em USB acompanha o *Generic USB Driver*, que é um conjunto de bibliotecas para a implementação de um dispositivo genérico de comunicação com o PC via USB. Contém funções de leitura, escrita, bem como a descrição e configuração de um dispositivo USB *full-speed* (12 Mbps).

9.4 Bootloader

Por definição, o *bootloader* é um programa com a função de acessar a memória de programa e carregar a primeira instrução de execução na memória. Nem sempre esse programa é visível; na maioria das vezes, ele é invisível ao usuário.

Em microcontroladores, um *bootloader* é um programa, armazenado em parte da memória do MCU e que se comunica com um PC (via serial ou USB). O *bootloader* recebe o programa do usuário através do PC e escreve-o na memória FLASH do microcontrolador. O programa pode, depois, ser executado normalmente. *Bootloaders* só podem ser usados em dispositivos capazes de escreverem em sua memória FLASH através de *software*. Um *bootloader*, propriamente, deve, no entanto, ser carregado no microcontrolador através de um programador externo.

Para que o *bootloader* seja capaz de ser inicializado depois de cada *reset*, a instrução que chama essa parte do código deve estar entre as 4 primeiras instruções do programa. Ainda, o programa pode ser feito de modo a ter dois modos de operação: modo de programação, em que o *bootloader* é carregado (quando se pressiona um botão, por

exemplo, durante a reinicialização), e o modo de execução de programa, em que o *bootloader* não é carregado, e sim o código principal do programa.

O *Framework* da *Microchip* oferece uma solução de *bootloader* via USB-HID, que dispensa a utilização de *drivers* específicos. Dessa forma, é bastante simples desenvolver aplicações, uma vez que um programador externo só é necessário para a gravação do *bootloader* no microcontrolador.

10 EAGLE

10.1 Visão geral

EAGLE (*Easily Applicable Graphical Layout Editor*) é um *software* de projeto de placas eletrônicas, desenvolvido pela CadSoft, uma subsidiária da Premier Farnell, que é um dos líderes mundiais em distribuição de componentes eletrônicos, produtos e serviços relacionados. O EAGLE pode ser classificado como um *software* ECAD (*Electronic Computer-Aided Design*), pois utiliza recursos computacionais para auxiliar em projetos eletrônicos. Recursos CAM (*Computer-Aided Manufacturing*) também estão presentes, tornando o EAGLE um *software* completo para a criação de placas de circuito impresso, uma vez que ele fornece recursos que auxiliam desde a concepção do esquemático até a construção, propriamente dita, da placa de circuito impresso. Estão disponíveis versões para Windows, Linux e Mac OS X.

O EAGLE é composto basicamente por três módulos:

- Editor de *Layout*: utilizado para projetar as placas de circuito impresso;
- Editor de Esquemático: utilizado para criar o esquemático do circuito;
- *Autorouter*: ferramenta que posiciona as trilhas na placa automaticamente;

Ambos os editores possuem o editor de biblioteca, o processador CAM, e podem executar programas e scripts ULP (User Language Programs). O editor de biblioteca permite a criação de novos componentes, definindo seu símbolo para esquemático e seu encapsulamento para placa. A biblioteca de componentes que acompanha o *software* é bastante completa, e componentes extras podem ser obtidos através da internet. O processador CAM é responsável por gerar os arquivos que são usados na fabricação das placas, como, por exemplo, o formato Gerber, que descreve as trilhas, as máscaras de solda, entre outras coisas, ou o formato Excellon, que descreve os furos. Além das funções presentes no próprio *software*, é possível acrescentar funcionalidades através de programas e scripts ULP.

Um recurso bastante interessante desse *software* é que ele mantém o esquemático e a placa sincronizados, ou seja, modificações feitas no esquemático resultam, automaticamente, em modificações na placa, e vice versa. Desse modo o esquemático e a placa sempre estarão refletindo o mesmo circuito.

Na tela principal do software, chamada de *Control Panel* (painel de controle), é possível consultar e alterar as bibliotecas de componentes (*Libraries*) e os ULPs (*User Language Programs*) que estão instalados. Nessa tela é possível abrir ou criar novos esquemáticos, placas, ULPs, entre outros.

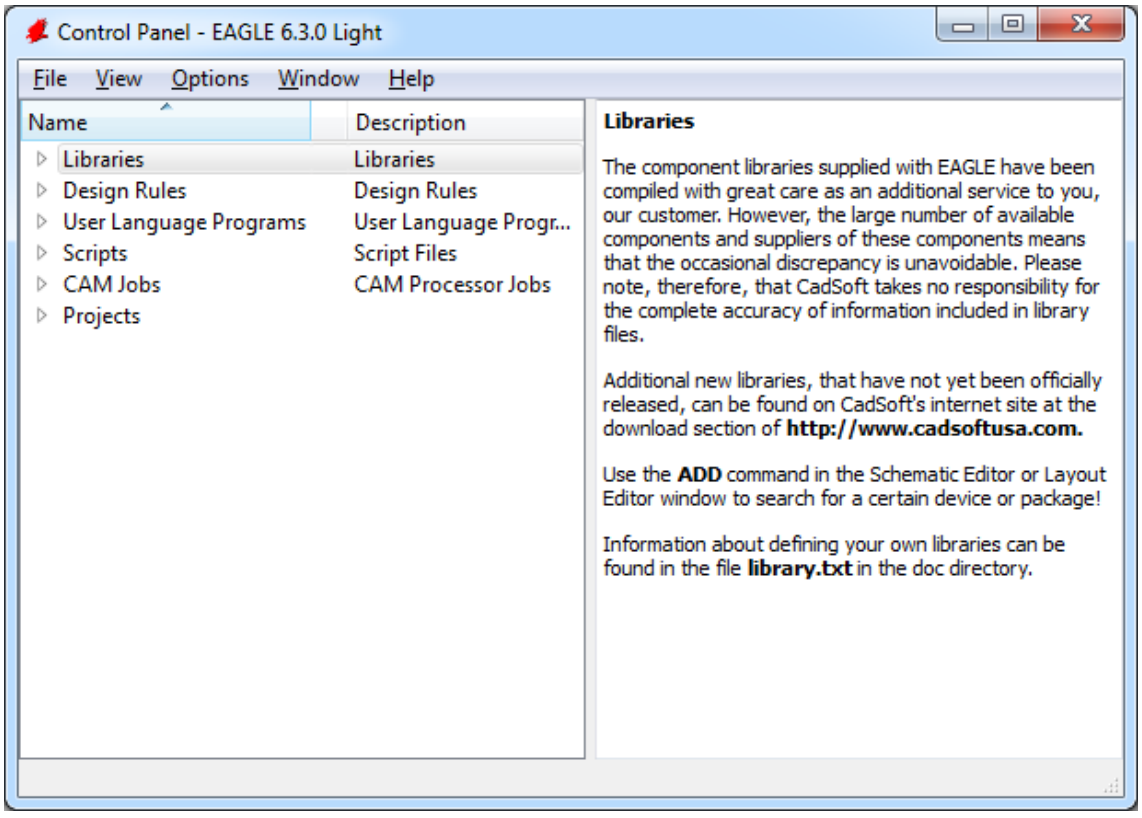


Figura 18 - Control Panel, tela principal do software.

As telas do editor de esquemático e do editor de *layout* são muito semelhantes, pois a maioria das ferramentas é comum aos dois módulos.

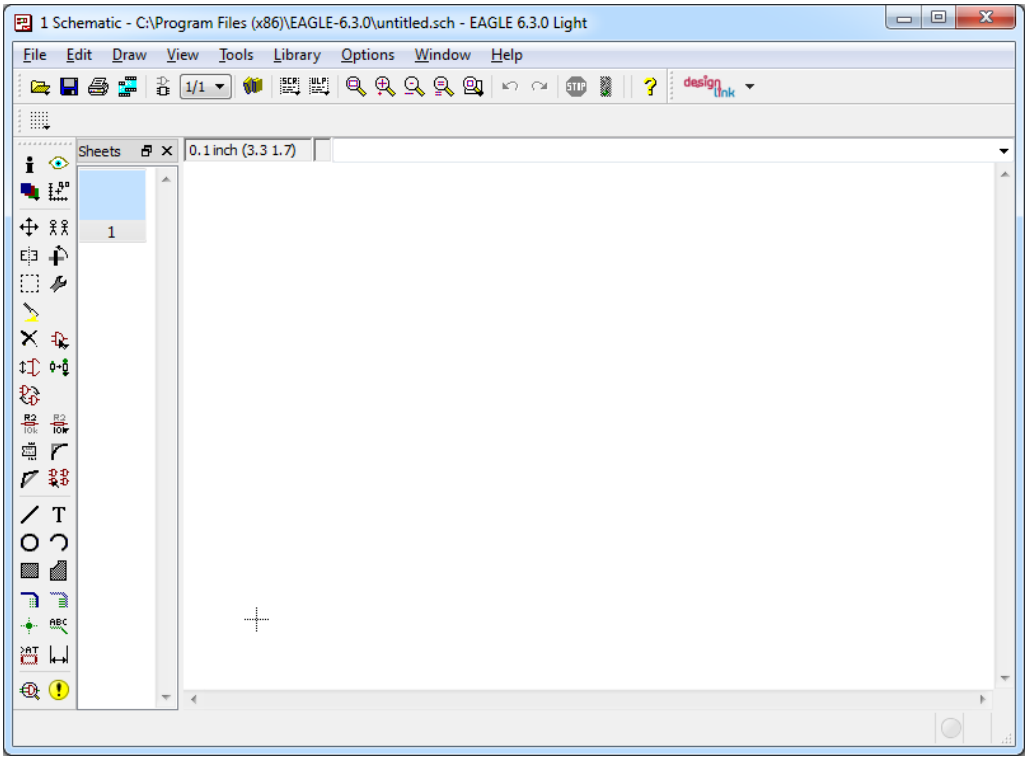


Figura 19: Editor de esquemático.

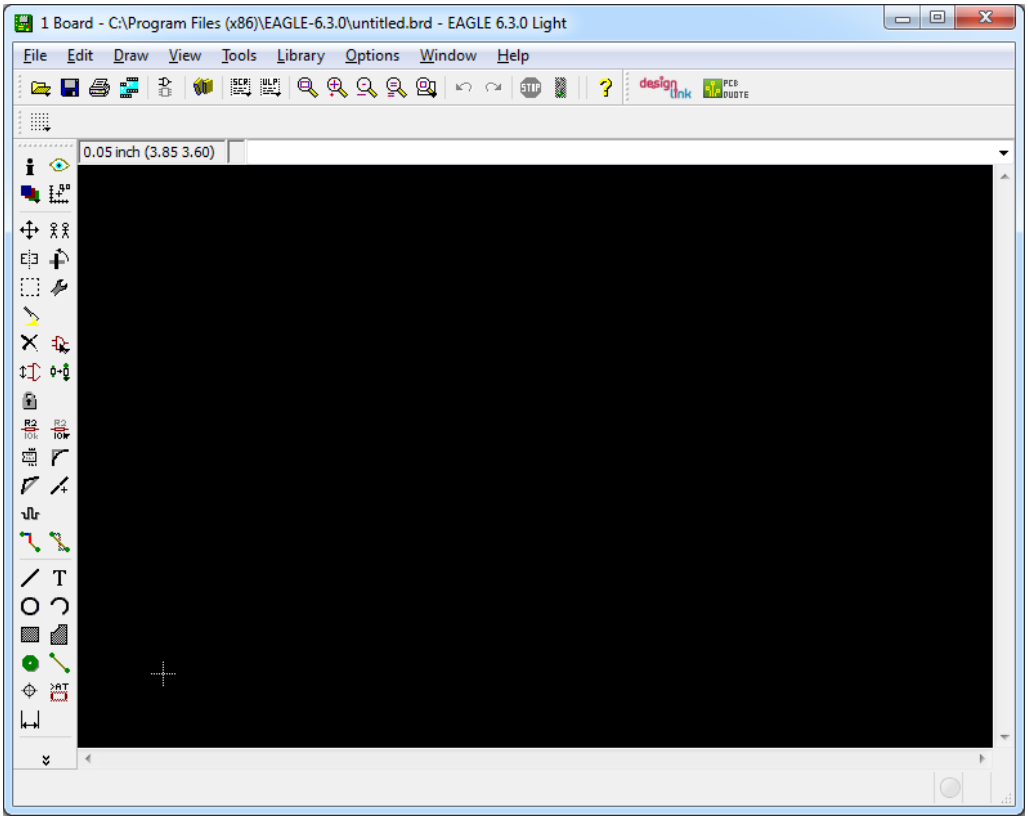


Figura 20: Editor de layout.

11 Desenvolvimento

11.1 Apresentação

O dispositivo conta com os recursos de *hardware* listados a seguir:

- 16 I/O digitais
- 8 ADC de 10 *bits*
- 1 saída PWM com resolução de 10 *bits*
- 1 módulo PWM *half bridge* com resolução de 10 *bits*

Na Figura 21 é indicada a disposição dos recursos e principais componentes do equipamento.

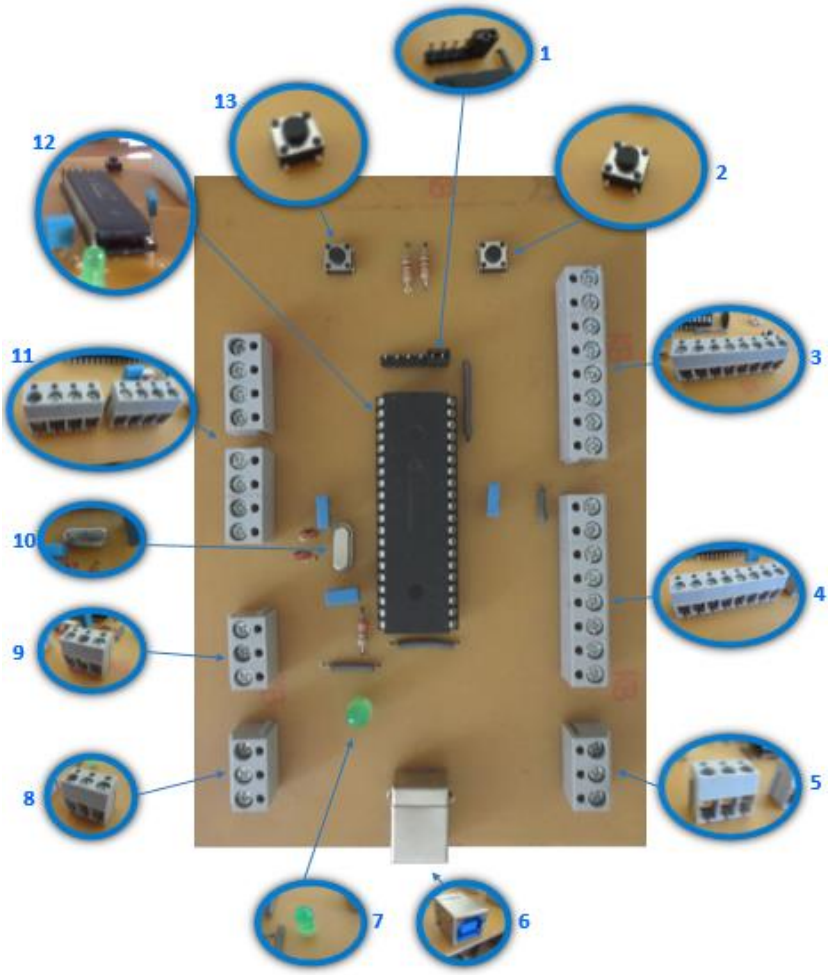


Figura 21- Disposição dos principais recursos e componentes

1. *Header* para programação *in circuit*
2. Botão do modo *bootloader*
3. Entradas/saídas digitais D0 – D7
4. Entradas/saídas digitais D8 – D15
5. VDD
6. Entrada USB tipo B
7. Led indicador de modo *bootloader*
8. VSS
9. Módulos PWM
10. Cristal 24 MHz
11. Saídas analógicas AN0 – AN07
12. Microcontrolador PIC18F4550
13. MCLR (*reset*)

11.2 Motivação

O primeiro protótipo foi construído em *protoboard*, sem que houvesse sido feito um projeto do circuito a ser montado. Esse primeiro protótipo, criado essencialmente para desenvolvimento da ideia e realização de testes, foi exaustivamente submetido a provas.

Criado em 2009, partiu da necessidade de um equipamento que fosse a interface entre o computador e o mundo físico, inicialmente para a realização de trabalhos propostos pelas disciplinas que cursávamos. Buscamos soluções comerciais, que verificamos serem bastante dispendiosas e/ou com recursos muito além dos que necessitávamos realmente para tais aplicações. Como já tínhamos algum conhecimento em desenvolvimento de VIs em *NI LabVIEW* e programação em C, começamos a buscar uma solução alternativa, desenvolvendo nosso próprio equipamento, que fosse programado em C e utilizado através da interface em *NI LabVIEW*.

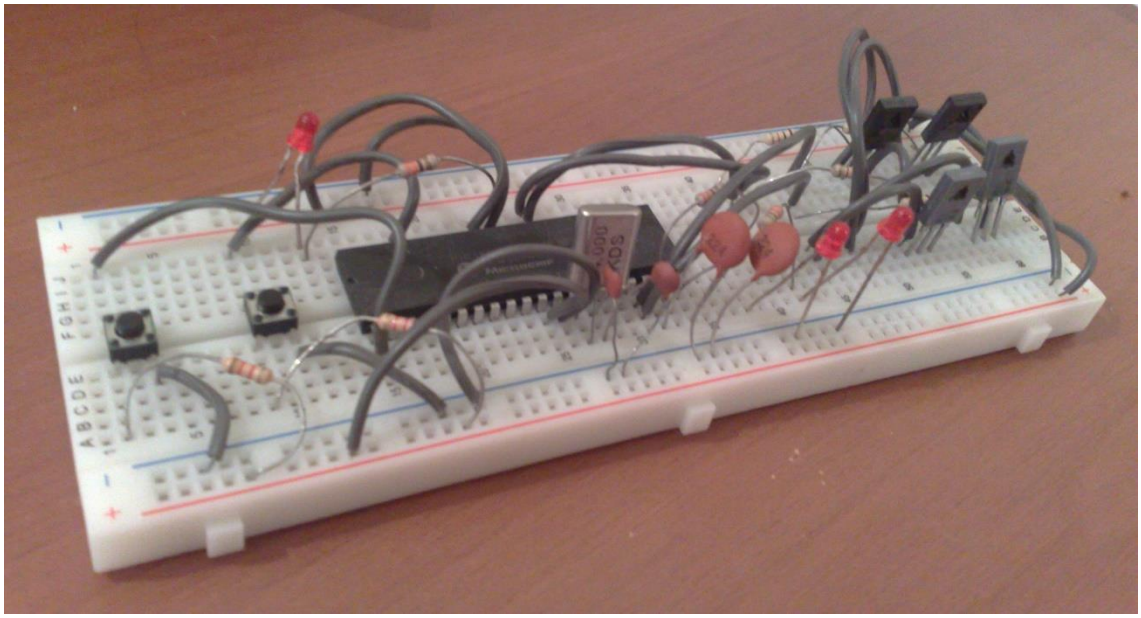


Figura 22 - Foto do protótipo.

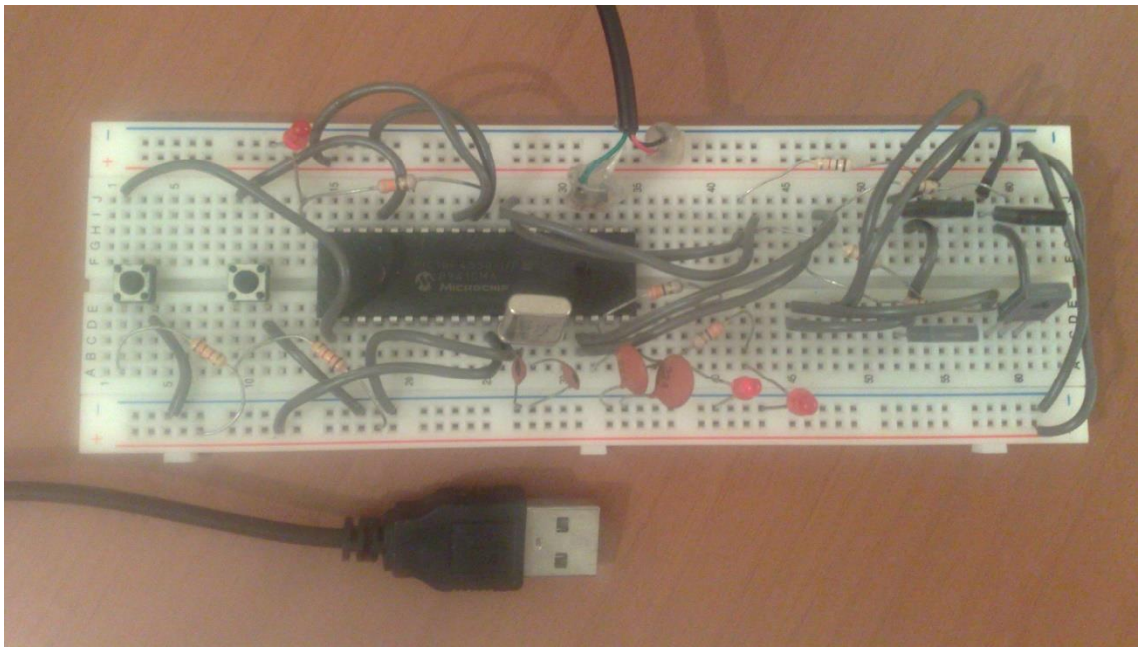


Figura 23 - Foto do protótipo.

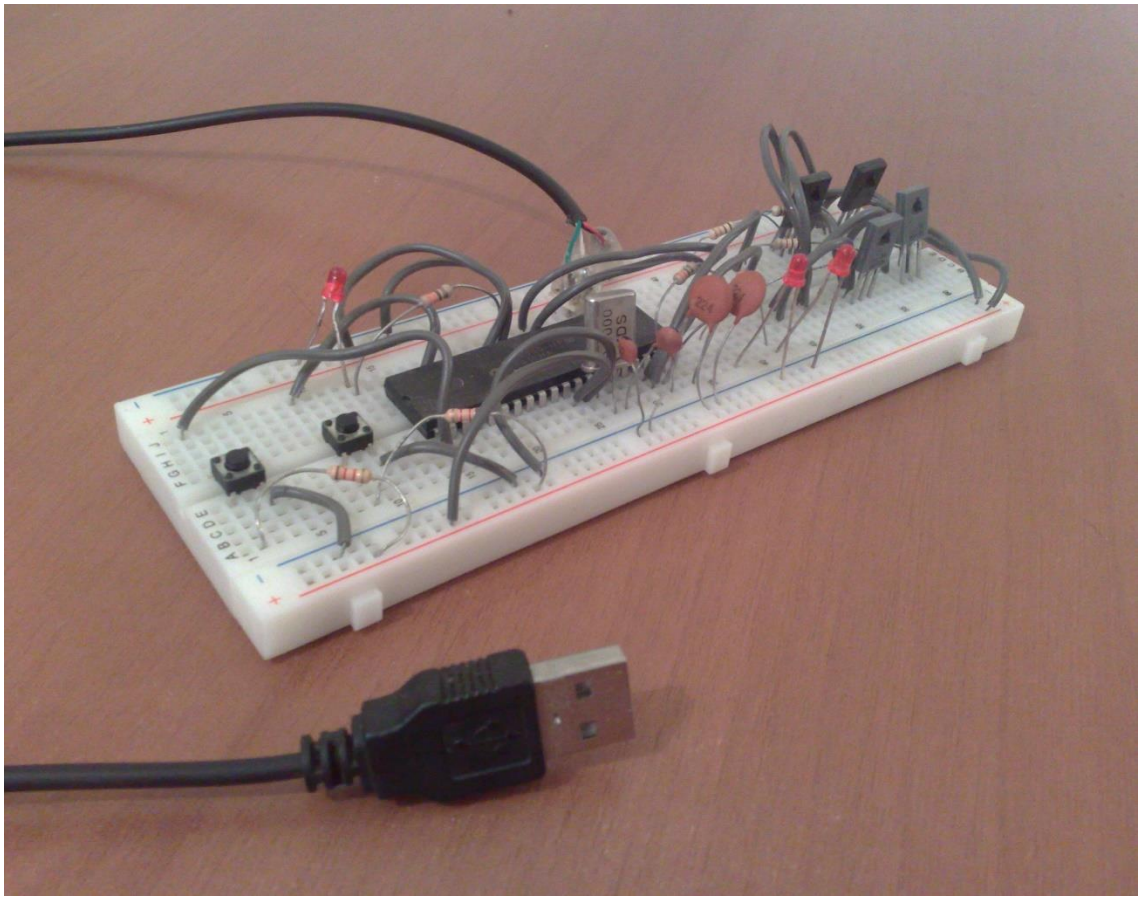


Figura 24 - Foto do protótipo.

O resultado foi o protótipo que veio sendo utilizado, desde então, no desenvolvimento dos projetos de disciplinas como *Problemas de Engenharia Mecatrônica, Modelagem e Simulação de Sistemas Dinâmicos, Sistemas de Controle, Elementos de Automação* e *Projetos de Sistemas Mecatrônicos*, além de diversos projetos caseiros, como o controle de mecanismo barra – bola com motor DC, leitura de *encoder* e controle de motor de passo.

Dessa forma, o equipamento pôde evoluir até se tornar estável e confiável. Ao longo do tempo, a comunicação foi sendo alterada para se tornar mais eficiente, e os recursos foram adicionados e também melhorados.

11.3 Hardware

O projeto do circuito foi desenvolvido, para viabilizar a montagem da placa que acomodaria o circuito final. A Figura 24 mostra o circuito da forma como foi projetado e a associação entre os componentes.

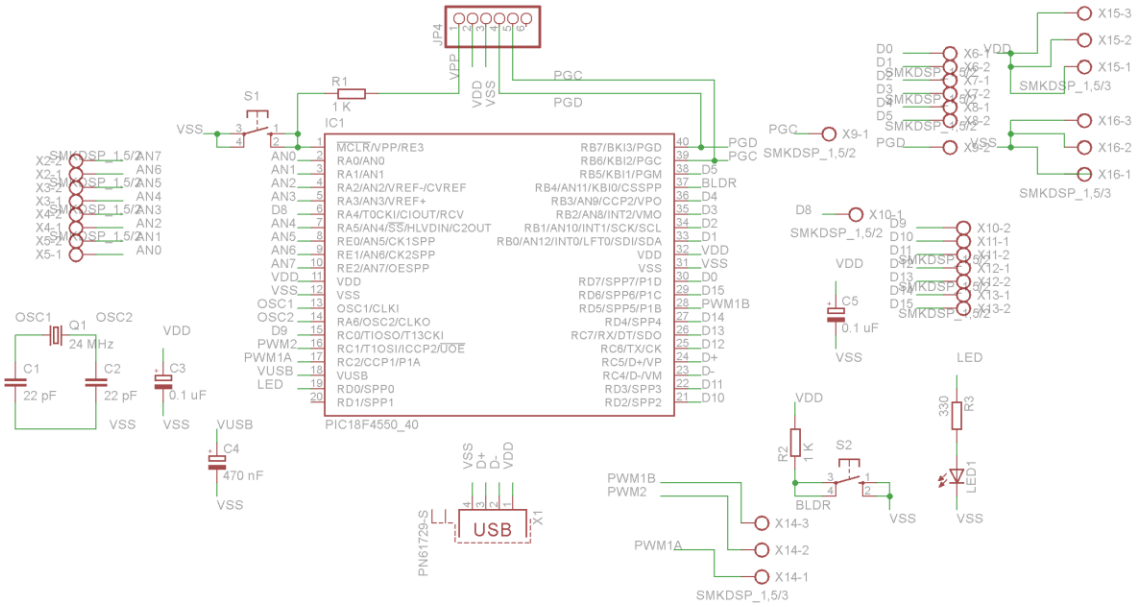


Figura 25 - Esquemático do circuito eletrônico.

Definidos os componentes utilizados e suas respectivas relações, o desenvolvimento da placa final do circuito se torna possível. A lista completa de componentes utilizados é a que segue.

- 1 x Placa fenolite 15 x 20 cm
- 2 x Capacitor cerâmico 22 pF
- 2 x Capacitor poliéster 0,1 μF
- 1 x Capacitor poliéster 0,47 μF
- 1 x Soquete padrão DIP 40 pinos
- 1 x Microcontrolador PIC18F4550
- 1 x *Header* de 6 pinos
- 1 x *Jumper* sem aba
- 1 x Led difuso 5mm cor verde
- 1 x Cristal 24 MHz

- 2 x Resistor 1 $k\Omega$
- 1 x Resistor 330 Ω
- 2 x Chave táctil 4 terminais 5 mm
- 1 x Conector USB-B fêmea
- 12 x Borne KRE 2 terminais 5 mm
- 3 x Borne KRE 3 terminais 5 mm

A partir da lista de peças e feito o esquemático do projeto, é possível criar o modelo da placa final, então. A Figura 25 e Figura 26 mostram o desenho esquemático da placa, com e sem a indicação dos componentes.

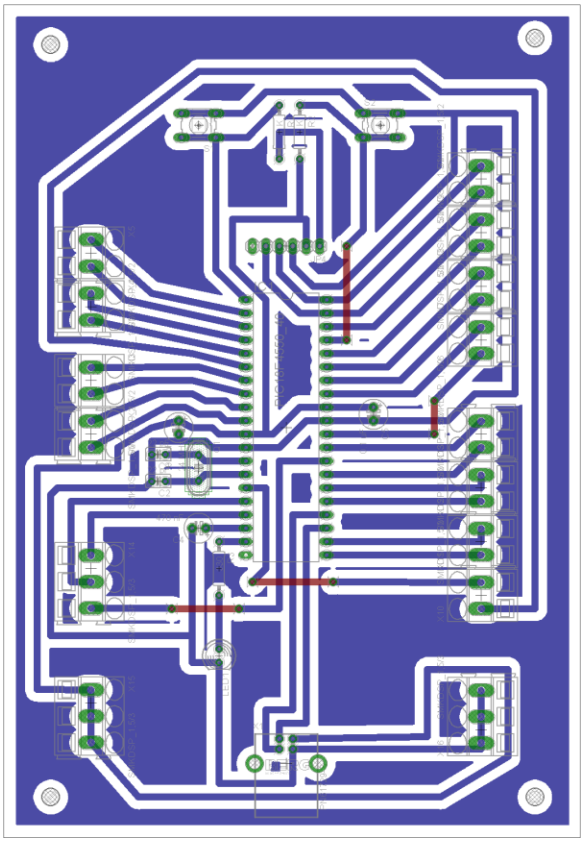


Figura 26 - Desenho esquemático da placa, com indicação dos componentes.

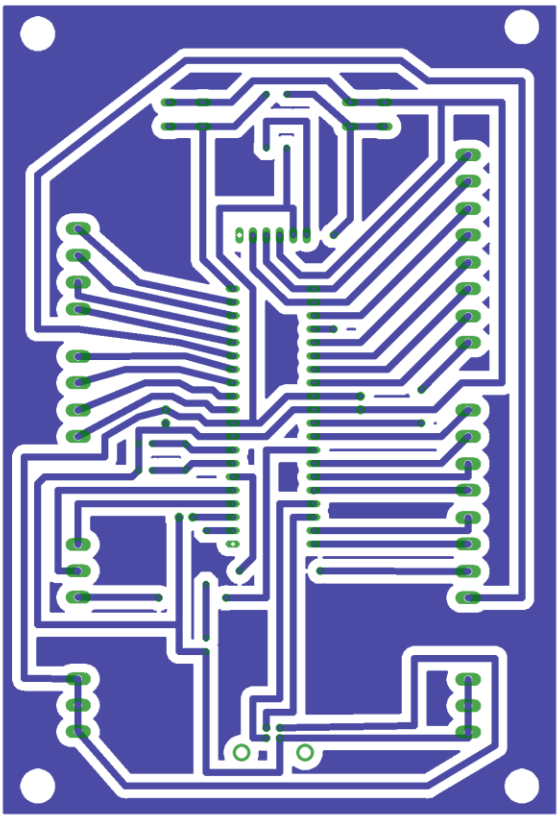


Figura 27 - Desenho esquemático da placa, somente com marcações de furos, ilhas e trilhas.

O trabalho de confecção da placa de circuito foi documentado em imagens, e seguem com as respectivas indicações das etapas de montagem.

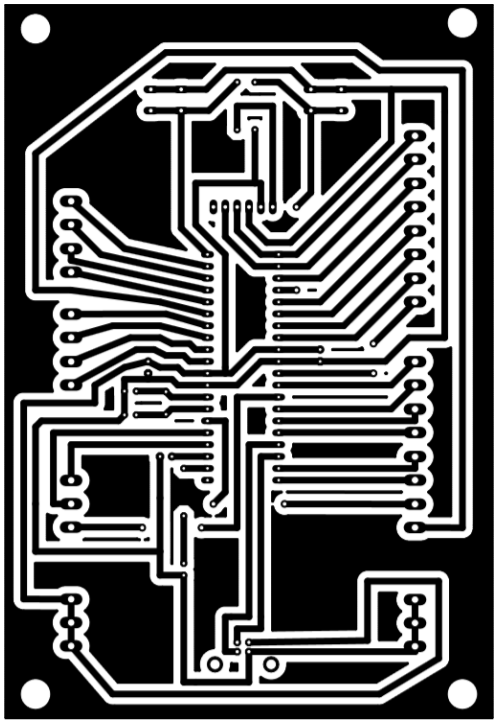


Figura 28 - Imagem da placa impressa em papel transfer.

Depois de impressa em papel apropriado, a imagem é fixada sobre a placa (segundo a orientação da camada, *top* ou *bottom*) e é prensada em alta temperatura. Dessa forma, o *toner* impregnado no papel é transferido para a placa de cobre previamente limpa e lixada, marcando as trilhas, ilhas e indicações de furos sobre o cobre. Para esta operação, a prensa operou por 2 minutos a uma temperatura de 50 °C.

Após essa operação, a placa é imersa em solução de percloreto de ferro, que corroi o cobre das áreas onde não há *toner*. Após cerca de 40 minutos, a placa é retirada e limpa, com solvente capaz de eliminar o *toner* de sua superfície.

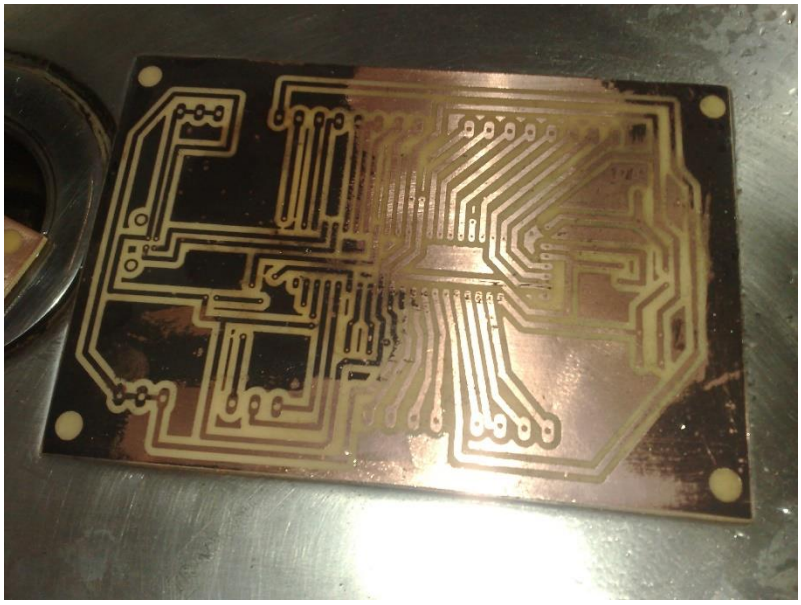


Figura 29 - Placa durante a limpeza.

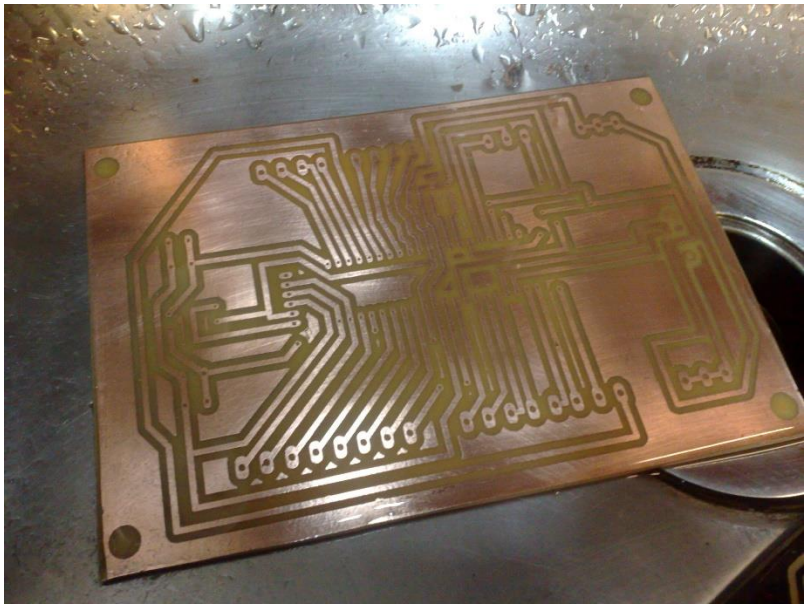


Figura 30 - Placa após a limpeza do toner.

Na sequência, procede a furação da placa nos pontos que receberão os terminais dos componentes ou acessórios de fixação ou apoio.

Figura 31 - Furação.

Finalmente, os componentes são montados em suas respectivas posições e soldados.

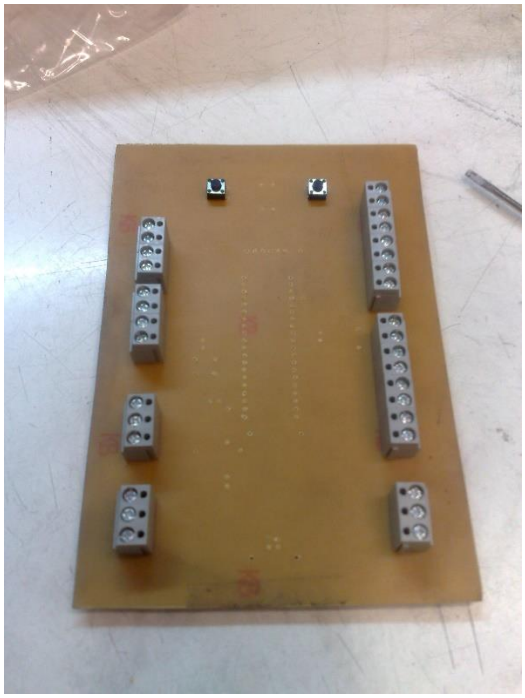


Figura 32 - Montagem.

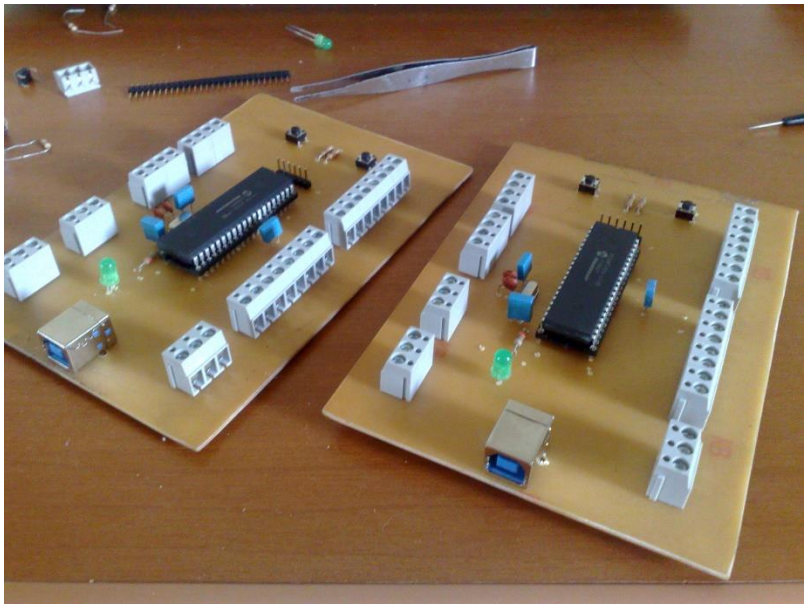


Figura 33 - Placa finalizada.

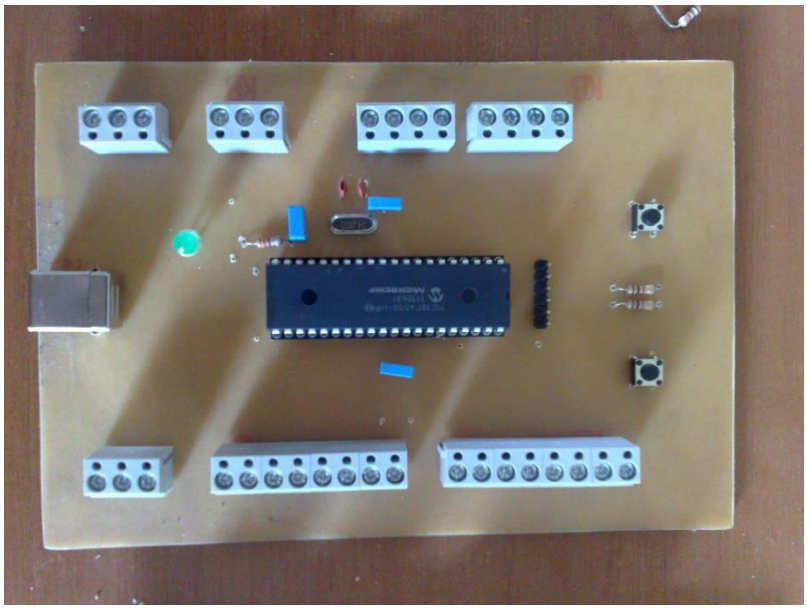


Figura 34 – Placa finalizada.

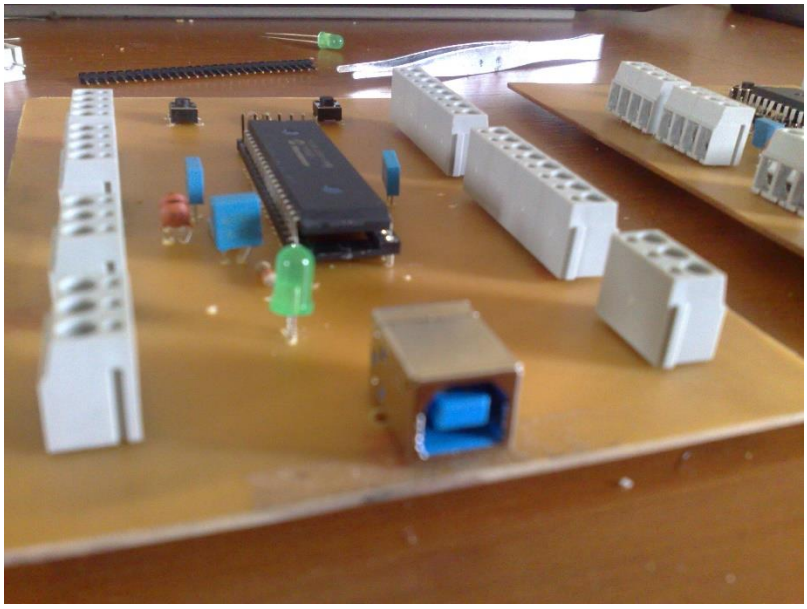


Figura 35 - Placa finalizada.

11.4 Protocolo

Fazendo uso da *Microchip Framework*, é possível estabelecer um canal de troca de dados entre um programa sendo executado no computador e um programa sendo executado no microcontrolador. Para que esses dados tenham sentido é necessário definir um protocolo de comunicação, que especifique como as informações de interesse serão enviadas e recebidas.

Tendo em mente que esse protocolo interliga dois sistemas com velocidades de processamento que são bastante diferentes, é lógico fazer com que a maior parte do processamento seja feito no sistema mais rápido, quando possível. Para tanto, o protocolo deve ser simples do ponto de vista do microcontrolador.

O protocolo se resume, basicamente, à troca de valores digitais e analógicos entre o computador e o microcontrolador. O computador deve enviar ao microcontrolador a configuração desejada para o sistema: quais portas serão entradas digitais, quais serão saídas digitais, quantas portas analógicas serão utilizadas, qual o tempo de aquisição para cada porta analógica, e a frequência do PWM. O computador também deve enviar o estado desejado para as saídas digitais, que pode ser nível alto ou baixo, e o *duty cycle* de cada

saída PWM. O microcontrolador, por sua vez, responde com o estado das entradas digitais, e os valores convertidos das entradas analógicas.

O tamanho do *endpoint* 1, que é o *endpoint* usado para a comunicação com o dispositivo, é o maior número de bytes que podem ser enviados ou recebidos durante uma transação. O ideal é que cada comando seja enviado ou recebido em apenas uma transação, a fim de obter maiores velocidades de comunicação.

O resultado é um protocolo simples, que atende aos requisitos acima, e é composto por apenas quatro comandos. A comunicação entre o computador e o microcontrolador é sempre iniciada pelo computador: os comandos enviados pelo microcontrolador são respostas aos comandos enviados pelo computador. Assim, dois dos comandos são enviados do computador para o microcontrolador, e os outros dois são as respostas enviadas pelo microcontrolador para o computador. Os dados nos comandos estão, sempre que possível, em formatos que são diretamente compatíveis com os registradores especiais do microcontrolador, de modo que necessitem de pouco ou nenhum tratamento por parte do microcontrolador. Os comandos são identificados pelo primeiro *byte*.

O primeiro comando, responsável por configurar o dispositivo, é composto por:

- Identificação do comando (valor 255)
- Direção das portas A, B, C, D
- Número de entradas analógicas
- Tempo de aquisição de cada porta analógica
- *Prescaler* do Timer2
- Período do *Timer2*

A direção das portas é enviada num formato que é compatível diretamente com os registradores do microcontrolador. O único tratamento necessário é limitar quais *bits* podem ser afetados, pois nem todas as portas do microcontrolador serão usadas como entradas e saídas digitais: as entradas analógicas devem ser configuradas como entradas, e as saídas de PWM devem ser configuradas como saídas. Como a placa identifica as portas digitais por Dn, o computador é responsável por converter essa numeração lógica na posição física de cada porta.

O número de entradas analógicas indica quais portas terão os seus valores analógicos convertidos e enviados ao computador. Pode variar de 0, onde nenhuma porta terá seu valor convertido, até 8, onde todas as portas serão convertidas. As portas a serem convertidas são de AN0 até AN(n-1), onde n é o número de portas analógicas. Por exemplo, se o valor passado for 3, as portas AN0, AN1 e AN2 serão habilitadas.

O tempo de aquisição especifica quanto tempo o circuito de amostragem & *hold* ficará conectado à entrada analógica antes de realizar a conversão, para cada entrada. O valor é especificado no formato do registrador que controla o tempo de aquisição, e é definido em função do *clock* do conversor.

O valor do *prescaler* do Timer2 define um divisor para o *clock* que entra no circuito do timer, podendo ser 1, 4 ou 16. Logo, o *clock* do *timer* pode ser $\frac{F_{OSC}}{1}$, $\frac{F_{OSC}}{4}$ ou $\frac{F_{OSC}}{16}$, onde F_{OSC} é o *clock* do microcontrolador. Também em um formato compatível com o respectivo registrador.

O período do Timer2 controla qual o período do timer, ou seja, até qual valor o timer irá contar antes de retornar a 0. Este valor, junto com o *prescaler*, determina qual será a frequência das saídas PWM.

Após receber este comando e fazer as configurações necessárias, o microcontrolador responde com um comando que tem apenas:

- Identificação do comando (valor 255)

Esta resposta serve para o computador verificar se o microcontrolador recebeu o comando e se a comunicação está ocorrendo normalmente.

O segundo comando é responsável por enviar os níveis desejados das saídas digitais e o *duty cycle* de cada saída PWM. Os valores transmitidos são:

- Identificação do comando (valor 0)
- Saída das portas A, B, C, D
- *Duty cycle* do PWM 1 e 2

As saídas digitais são transferidas em um formato compatível diretamente com os registradores do microcontrolador, assim como foram transferidas as direções das portas.

Novamente, o microcontrolador limita os bits que podem ser alterados à apenas os que correspondem às portas digitais usadas. As portas que estão configuradas como entradas digitais não sofrem efeito algum. O computador é responsável por converter a posição física das portas para a numeração lógica usada pelo projeto.

O *duty cycle* de cada saída PWM é transmitido no formato esperado pelos registradores do microcontrolador, que é calculado pelo computador.

Após receber esse comando o microcontrolador altera os valores das saídas de acordo com os valores recebidos, e em seguida faz a leitura das entradas digitais e analógicas, enviando para o computador um comando de resposta com os seguintes valores:

- Identificação do comando (valor 0)
- Entrada das portas A, B, C, D
- Entradas analógicas AN0-AN8

As entradas digitais são especificadas diretamente no formato lido dos registradores do microcontrolador. O computador é responsável por traduzir a posição física das entradas na posição lógica, Dn, usada pelo trabalho. Os valores apenas terão sentido para as portas configuradas como entradas digitais.

As entradas analógicas apenas terão valores significativos para àquelas que foram habilitadas previamente com o comando de configuração. As outras terão valores quaisquer. O valor retornado é o valor lido diretamente do conversor analógico digital.

Portanto, esses quatro comandos simples são suficientes para realizar todas as operações necessárias para o correto funcionamento do dispositivo sendo projetado. A maior parte do processamento é feito no computador, enviando valores calculados para o microcontrolador, que por sua vez apenas os escreve nos respectivos registradores.

11.5 Software

A solução de software para o projeto é composta por duas partes: o driver e as bibliotecas que são executados no computador, e o firmware que é executado no microcontrolador. Ambos implementam o protocolo discutido anteriormente, de modo que

a comunicação possa ser feita. Os VIs do LabVIEW, por sua vez, são implementados usando a biblioteca criada em C, fazendo a adaptação das funções para funcionamento no *LabVIEW*. A decisão de implementar o protocolo em uma biblioteca feita em C permite que a placa seja utilizada em outras linguagens, como C/C++ ou MATLAB, apenas utilizando as funções exportadas pela biblioteca, embora esse não seja o objetivo do trabalho.

Para estabelecer um canal de comunicação entre o computador e o microcontrolador foi utilizada a USB Framework, fornecida, gratuitamente, pela Microchip no seu pacote *Microchip Libraries for Applications*, disponível para download em seu site. No pacote está incluído o USB HID Bootloader, que permite programar o microcontrolador de maneira simples, através da própria porta USB, facilitando bastante o desenvolvimento e, posteriormente, a atualização do firmware.

Incluído na Framework está o código para o microcontrolador que implementa um dispositivo USB genérico, ou seja, que precisa de um driver específico para o seu correto funcionamento. O driver é fornecido junto ao pacote, bem como uma biblioteca que se comunica com o driver. Esses componentes criam um canal de troca de dados genéricos entre o microcontrolador e um programa fazendo uso da biblioteca. Esse canal será usado para implementar o protocolo definido para o projeto.

11.6 Biblioteca

A biblioteca em C que implementa o protocolo foi desenvolvida utilizando a linguagem C, no ambiente de desenvolvimento da Microsoft, o Visual Studio 2010, e foi chamada de DAQ. Ela utiliza as funções da biblioteca fornecida pela Microchip para se comunicar com o microcontrolador. Ela exporta as seguintes funções:

- daq_open_device – usada para criar uma conexão com o dispositivo;
- daq_close_device – fecha a conexão criada pelo daq_open_device;
- daq_configure – configura os parâmetros do dispositivo: direção das portas digitais, número de portas analógicas, tempo de aquisição das portas analógicas, e frequência do PWM;
- daq_write_digital – define o estado de uma saída digital;

- daq_read_digital – faz a leitura de uma entrada digital;
- daq_read_analog – lê o valor de uma entrada analógica;
- daq_set_duty_cycle – define o *duty cycle* de uma saída PWM;
- daq_update – faz a comunicação com o microcontrolador: envia os valores a serem definidos e recebe os valores a serem lidos;

As funções que interagem com o dispositivo não se comunicam imediatamente com o mesmo: quando é definido o estado de uma saída digital ou o *duty cycle* de um PWM, ou quando é lido o valor de uma entrada digital ou analógica, os valores são escritos e lidos em uma estrutura interna. Quando é chamada a função que faz a comunicação com o microcontrolador os valores que devem ser escritos são enviados e os valores a serem lidos são recebidos e armazenados na estrutura interna, para serem lidos posteriormente, pelas respectivas funções. Esse agrupamento dos dados em um único comando permite alcançar maiores velocidades de comunicação.

Segue abaixo o código da biblioteca desenvolvida:

daq.c – código da biblioteca

```
#include <stdio.h>
#include <stdint.h>
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>

#include "daq.h"

/****************** Definições do MCHPUSB ******************/

/* Resultado de uma operação */
#define      MPUSB_FAIL    0 /* Falha */
#define MPUSB_SUCCESS 1 /* Sucesso */

/* Direção do endpoint */
#define MP_WRITE 0 /* Escrita */
#define MP_READ  1 /* Leitura */

/* Número máximo de dispositivos */
#define MAX_NUM_MPUSB_DEV 127

/* Retorna o número de dispositivos com o VID e PID especificados */
DWORD (*MPUSBGetDeviceCount)(PCHAR pVID_PID);

/* Abre um endpoint no dispositivo */
HANDLE (*MPUSBOpen)(DWORD instance, /* Número de instância do dispositivo */
                    PCHAR pVID_PID,    /* VID e PID do dispositivo (exemplo:
"vid_04d8&pid_000b") */
                    PCHAR pEP,         /* Endpoint (exemplo: "\\MCHP_EP1") */
                    DWORD dwDir,       /* Direção do endpoint (MP_READ ou MP_WRITE) */
                    DWORD dwReserved); /* Reservado */
```

```
/* Le de um endpoint */
DWORD (*MPUSBRead)(HANDLE handle,         /* Handle que identifica o endpoint */
                   PVOID pData,           /* Ponteiro para memória que irá receber os
dados */
                   DWORD dwLen,           /* Número de caracteres a serem lidos */
                   PDWORD pLength,        /* Ponteiro para variável que armazena o
número de caracteres lidos */
                   DWORD dwMilliseconds); /* Time-out em mili segundos, ou INFINITE */

DWORD (*MPUSBWrite)(HANDLE handle,         /* Handle que identifica o endpoint */
                    PVOID pData,           /* Ponteiro para memória que contém os
dados a serem escritos */
                    DWORD dwLen,           /* Número de caracteres a serem escritos */
                    PDWORD pLength,        /* Ponteiro para variável que recebe o
número de caracteres escritos */
                    DWORD dwMilliseconds); /* Time-out em mili segundos, ou INFINITE
*/

/* Fecha um endpoint */
BOOL (*MPUSBClose)(HANDLE handle);

/***************** Comandos PIC18 *****************/

#pragma pack(push)
#pragma pack(1)

#define CMD_OUT_CONFIGURE 0xFF
struct cmd_out_configure_t
{
	uint8_t command; /* CMD_OUT_CONFIGURE */
	uint8_t tris[4]; /* TRISA, TRISB, TRISC, TRISD */
	uint8_t an_count; /* Número de entradas analógicas */
	uint8_t adcon2; /* ADCON2[ACQT2:ACQT0] */
	uint8_t t2con; /* T2CON[T2CKPS1:T2CKPS0] */
	uint8_t pr2; /* PR2 */
};

#define CMD_IN_CONFIGURE 0xFF
struct cmd_in_configure_t
{
	uint8_t command; /* CMD_IN_CONFIGURE */
};

#define CMD_OUT_SET 0x00
struct cmd_out_set_t
{
	uint8_t command; /* CMD_OUT_SET */
	uint8_t lat[4]; /* LATA, LATB, LATC, LATD */
	uint8_t ccp1con; /* CCP1CON */
	uint8_t ccpr1l; /* CCPR1L[DC1B1:DC1B0] */
	uint8_t ccp2con; /* CCP2CON */
	uint8_t ccpr2l; /* CCPR2L[DC2B1:DC2B0] */
};

#define CMD_IN_GET 0x00
struct cmd_in_get_t
{
	uint8_t command; /* CMD_IN_GET */
	uint8_t port[4]; /* PORTA, PORTB, PORTC, PORTD */
	uint16_t an[8]; /* AN0, AN1, AN2, AN3, AN4, AN5, AN6, AN7 */
};
```

```
#pragma pack(pop)

/****************** Dados internos ******************/

/* Estado interno do DAQ */
struct daq_internal_t
{
	HANDLE endpoint_in;
	HANDLE endpoint_out;
	int digital_in[16];
	int digital_out[16];
	int analog[8];
	int duty_cycle[2];
	int pr2;
};

/* Identifica uma porta do PIC18 */
typedef struct
{
	int port; /* 0=PORTA, 1=PORTB, 2=PORTC, 3=PORTD */
	int bit; /* Número do bit */
} port_bit_t;

#define PORT_A 0
#define PORT_B 1
#define PORT_C 2
#define PORT_D 3

/* Tabela para conversão entre Dx e portas do PIC18 */
static const port_bit_t port_bit_shuffle[] = {
	{ PORT_D, 7 }, /* D0 */
	{ PORT_B, 0 }, /* D1 */
	{ PORT_B, 1 }, /* D2 */
	{ PORT_B, 2 }, /* D3 */
	{ PORT_B, 3 }, /* D4 */
	{ PORT_B, 5 }, /* D5 */
	{ PORT_B, 6 }, /* D6 */
	{ PORT_B, 7 }, /* D7 */
	{ PORT_A, 4 }, /* D8 */
	{ PORT_C, 0 }, /* D9 */
	{ PORT_D, 2 }, /* D10 */
	{ PORT_D, 3 }, /* D11 */
	{ PORT_C, 6 }, /* D12 */
	{ PORT_C, 7 }, /* D13 */
	{ PORT_D, 4 }, /* D14 */
	{ PORT_D, 6 }, /* D15 */
};

/* Relaciona o tempo de aquisição com os valores do registrador ADCON2 */
typedef struct
{
	int time; /* micro segundos */
	uint8_t adcon2; /* ADCON2 */
} acquisition_time_t;
#define SAMPLE_TIME_COUNT (sizeof(sample_time)/sizeof(sample_time[0]))
static const acquisition_time_t sample_time[] = {
	{  2, (1 << 3) }, /*  2,667us */
	{  5, (2 << 3) }, /*  5,333us */
	{  8, (3 << 3) }, /*  8,000us */
	{ 10, (4 << 3) }, /* 10,667us */
	{ 16, (5 << 3) }, /* 16,000us */
```

```
	{ 21, (6 << 3) }, /* 21,333us */
	{ 26, (7 << 3) }, /* 26,667us */
};

/***************** Funções internas *****************/

/* Tenta abrir o endpoint 1 para leitura e escrita */
static BOOL try_open_device(daq_handle_t handle, int vendor_id, int product_id)
{
	CHAR vid_pid[] = "vid_0000&pid_0000";
	DWORD device_count;
	DWORD i;
	sprintf_s(vid_pid, sizeof(vid_pid), "vid_%04x&pid_%04x", (vendor_id & 0xffff),
(product_id & 0xffff));
	device_count = MPUSBGetDeviceCount(vid_pid);
	if (device_count > 0)
	{
		for (i = 0; i < MAX_NUM_MPUSB_DEV; ++i)
		{
			handle->endpoint_in = MPUSBOpen(i, vid_pid, "\\MCHP_EP1", MP_READ,
0);
			handle->endpoint_out = MPUSBOpen(i, vid_pid, "\\MCHP_EP1",
MP_WRITE, 0);
			if ((handle->endpoint_in != INVALID_HANDLE_VALUE) && (handle-
>endpoint_out != INVALID_HANDLE_VALUE))
			{
				return TRUE;
			}
			if (handle->endpoint_in != INVALID_HANDLE_VALUE)
			{
				MPUSBClose(handle->endpoint_in);
			}
			if (handle->endpoint_out != INVALID_HANDLE_VALUE)
			{
				MPUSBClose(handle->endpoint_out);
			}
		}
	}
	return FALSE;
}

/* Carrega a DLL da Microchip e encontra o endereço das funções usadas */
static BOOL load_mchpusb()
{
	HMODULE hModule;
	hModule = LoadLibrary("MPUSBAPI.dll");
	if (hModule == NULL)
	{
		return FALSE;
	}
	MPUSBGetDeviceCount = (DWORD (*)(PCHAR))GetProcAddress(hModule,
"_MPUSBGetDeviceCount");
	MPUSBOpen = (HANDLE (*)(DWORD,PCHAR,PCHAR,DWORD,DWORD))GetProcAddress(hModule,
"_MPUSBOpen");
	MPUSBRead = (DWORD (*)(HANDLE,PVOID,DWORD,PDWORD,DWORD))GetProcAddress(hModule,
"_MPUSBRead");
	MPUSBWrite = (DWORD (*)(HANDLE,PVOID,DWORD,PDWORD,DWORD))GetProcAddress(hModule,
"_MPUSBWrite");
	MPUSBClose = (BOOL (*)(HANDLE))GetProcAddress(hModule, "_MPUSBClose");
	if ((MPUSBGetDeviceCount == NULL) || (MPUSBOpen == NULL) || (MPUSBRead == NULL)
|| (MPUSBWrite == NULL)|| (MPUSBClose == NULL))
	{
```

```
			return FALSE;
		}
		return TRUE;
}

/****************** Funções exportadas ******************/

daq_handle_t daq_open_device(int vendor_id, int product_id)
{
		daq_handle_t handle;
		handle = (daq_handle_t)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
sizeof(struct daq_internal_t));
		if (handle != NULL)
		{
			if (try_open_device(handle, vendor_id, product_id))
			{
				return handle;
			}
			HeapFree(GetProcessHeap(), 0, handle);
		}
		return NULL;
}

void daq_close_device(daq_handle_t handle)
{
		MPUSBClose(handle->endpoint_in);
		MPUSBClose(handle->endpoint_out);
		HeapFree(GetProcessHeap(), 0, handle);
}

int daq_configure(daq_handle_t handle, int digital_direction, int analog_count, int
analog_sample_time, int pwm_frequency)
{
		struct cmd_out_configure_t configure;
		struct cmd_in_configure_t reply;
		int i;
		int t2ckps;
		int pr2;
		DWORD length;
		configure.command = CMD_OUT_CONFIGURE;
		configure.tris[PORT_A] = 0x00;
		configure.tris[PORT_B] = 0x00;
		configure.tris[PORT_C] = 0x00;
		configure.tris[PORT_D] = 0x00;
		for (i = 0; i < 16; ++i)
		{
			if (digital_direction & (1 << i))
			{
				configure.tris[port_bit_shuffle[i].port] |= (1 <<
port_bit_shuffle[i].bit);
			}
		}
		configure.an_count = max(min(analog_count, 8), 0);
		for (i = 0; i < (SAMPLE_TIME_COUNT - 1); ++i)
		{
			if (sample_time[i].time >= analog_sample_time)
			{
				break;
			}
		}
		configure.adcon2 = sample_time[i].adcon2;
		t2ckps = 1;
```

```
	for (i = 0; i < 3; ++i)
	{
		pr2 = (48000000 / 4 / max(pwm_frequency, 1) / t2ckps);
		if (pr2 < 256)
		{
			break;
		}
		t2ckps *= 4;
	}
	configure.t2con = i;
	configure.pr2 = handle->pr2 = max(min(pr2, 255), 0);
	if (MPUSBWrite(handle->endpoint_out, &configure, sizeof(struct
cmd_out_configure_t), &length, 1000) == MPUSB_FAIL)
	{
		return 0;
	}
	if (length != sizeof(struct cmd_out_configure_t))
	{
		return 0;
	}
	if (MPUSBRead(handle->endpoint_in, &reply, sizeof(reply), &length, 1000) ==
MPUSB_FAIL)
	{
		return 0;
	}
	if ((length != sizeof(struct cmd_in_configure_t)) || (reply.command !=
CMD_IN_CONFIGURE))
	{
		return 0;
	}
	return 1;
}

void daq_write_digital(daq_handle_t handle, int channel, int state)
{
	if  ((channel >= 0) && (channel < 16))
	{
		handle->digital_out[channel] = state;
	}
}

int daq_read_digital(daq_handle_t handle, int channel)
{
	if  ((channel >= 0) && (channel < 16))
	{
		return handle->digital_in[channel];
	}
	else
	{
		return 0;
	}
}

int daq_read_analog(daq_handle_t handle, int channel)
{
	if ((channel >= 0) && (channel < 8))
	{
		return handle->analog[channel];
	}
	else
	{
		return 0;
```

```
	}
}

void daq_set_duty_cycle(daq_handle_t handle, int channel, int duty_cycle)
{
	if ((channel >= 1) && (channel <= 2))
	{
		handle->duty_cycle[channel - 1] = min(max(duty_cycle, 0), 100);
	}
}

int daq_update(daq_handle_t handle)
{
	struct cmd_out_set_t set;
	int i;
	DWORD length;
	uint8_t buffer[64];
	struct cmd_in_get_t* get = (struct cmd_in_get_t*)buffer;
	int duty;
	set.command = CMD_OUT_SET;
	set.lat[PORT_A] = 0x00;
	set.lat[PORT_B] = 0x00;
	set.lat[PORT_C] = 0x00;
	set.lat[PORT_D] = 0x00;
	for (i = 0; i < 16; ++i)
	{
		if (handle->digital_out[i])
		{
			set.lat[port_bit_shuffle[i].port] |= (1 <<
port_bit_shuffle[i].bit);
		}
	}
	duty = 4 * (handle->pr2 + 1) * handle->duty_cycle[0] / 100;
	duty = min(duty, 1023);
	set.ccp1con = ((duty & 0x03) << 4);
	set.ccpr1l = (duty >> 2);
	duty = 4 * (handle->pr2 + 1) * handle->duty_cycle[1] / 100;
	duty = min(duty, 1023);
	set.ccp2con = ((duty & 0x03) << 4);
	set.ccpr2l = (duty >> 2);
	if (MPUSBWrite(handle->endpoint_out, &set, sizeof(struct cmd_out_set_t),
&length, 1000) == MPUSB_FAIL)
	{
		return 0;
	}
	if (length != sizeof(struct cmd_out_set_t))
	{
		return 0;
	}
	if (MPUSBRead(handle->endpoint_in, buffer, sizeof(buffer), &length, 1000) ==
MPUSB_FAIL)
	{
		return 0;
	}
	if ((length != sizeof(struct cmd_in_get_t)) || (get->command != CMD_IN_GET))
	{
		return 0;
	}
	for (i = 0; i < 16; ++i)
	{
		handle->digital_in[i] = !!(get->port[port_bit_shuffle[i].port] & (1 <<
port_bit_shuffle[i].bit));
```

```
	}
	for (i = 0; i < 8; ++i)
	{
		handle->analog[i] = get->an[i];
	}
	return 1;
}

/* Função chamada quando a DLL é carregada */
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	switch (fdwReason)
	{
	case DLL_PROCESS_ATTACH:
		if (load_mchpusb() == FALSE)
		{
			return FALSE;
		}
		break;
	case DLL_PROCESS_DETACH:
		break;
	case DLL_THREAD_ATTACH:
		break;
	case DLL_THREAD_DETACH:
		break;
	}
	return TRUE;
}
```

daq.h – cabeçalho com a declaração das funções

```
#pragma once

struct daq_internal_t;
typedef struct daq_internal_t* daq_handle_t;

/*
 * Abre o DAQ
 * vendor_id - VID do dispositivo
 * product_id - PID do dispositivo
 * retorna - ponteiro para o handle do dispositivo, ou NULL em caso de falha
 */
daq_handle_t daq_open_device(int vendor_id, int product_id);

/*
 * Fecha o DAQ
 * handle - ponteiro para o handle do dispositivo
 */
void daq_close_device(daq_handle_t handle);

/*
 * Configura as entradas e saídas do DAQ
 * handle - ponteiro para o handle do dispositivo
 * digital_direction - bitmask com a direção das portas digitais (0=Output, 1=Input),
D0 no bit 0, D1 no bit 1, e assim por diante
 * analog_count - número de entradas analógicas usadas (de 0 a 8)
 * analog_sample_time - tempo mínimo de amostragem, em micro segundos (valor real será
maior ou igual a esse valor)
 * pwm_frequency - frequência do PWM em hertz (valor real será bastante próximo, com a
maior resolução possível)
```

```
 * retorna - 0=falha, 1=sucesso
 */
int daq_configure(daq_handle_t handle, int digital_direction, int analog_count, int
analog_sample_time, int pwm_frequency);

/*
 * Escreve em uma saída digital
 * handle - ponteiro para o handle do dispositivo
 * channel - número da saída digital (de 0 a 15)
 * state - 0=nível baixo, 1=nível alto
 */
void daq_write_digital(daq_handle_t handle, int channel, int state);

/*
 * Lê uma entrada digital
 * handle - ponteiro para o handle do dispositivo
 * channel - número da entrada digital (de 0 a 15)
 */
int daq_read_digital(daq_handle_t handle, int channel);

/*
 * Lê uma entrada analógica
 * handle - ponteiro para o handle do dispositivo
 * channel - número da entrada analógica (de 0 a 7)
 */
int daq_read_analog(daq_handle_t handle, int channel);

/*
 * Altera o duty cycle do PWM
 * handle - ponteiro para o handle do dispositivo
 * channel - número do canal de PWM (1 ou 2)
 * duty_cycle - duty cycle em porcentagem (de 0 a 100)
 */
void daq_set_duty_cycle(daq_handle_t handle, int channel, int duty_cycle);

/*
 * Faz a comunicação com o microcontrolador, escrevevendo as saídas e lendo as
entradas
 * handle - ponteiro para o handle do dispositivo
 * retorna - 0=falha, 1=sucesso
 */
int daq_update(daq_handle_t handle);
```

daq.def – arquivo que lista as funções a serem exportadas

```
EXPORTS
       daq_open_device
       daq_close_device
       daq_configure
       daq_write_digital
       daq_read_digital
       daq_read_analog
       daq_set_duty_cycle
       daq_update
```

11.7 Firmware

O código do microcontrolador implementa o protocolo usando a Framework USB da Microchip. Para ser compilado é necessário incluir a pasta onde se encontram os códigos que serão apresentados aqui, e também a pasta Microchip\Include, nos *Include directories* do compilador C18. Também é necessário compilar o arquivo Microchip\USB\usb_device.c junto com outros códigos. Os arquivos e pastas extras citados são obtidos através da *Microchip Application Libraries*.

Os códigos do firmware desenvolvido para o microcontrolador PIC18F4550 são apresentados abaixo:

main.c – define a função principal e os vetores de interrupção

```
#include <p18f4550.h>
#include "fw_daq.h"

#pragma code

void main(void)
{
    daq_init();
    while(1)
    {
        daq_loop();
    }
}

#pragma interrupt high_isr
void high_isr()
{
    daq_high_isr();
}

#pragma interruptlow low_isr
void low_isr()
{
    daq_low_isr();
}

#define USE_HID_BOOTLOADER

#ifdef USE_HID_BOOTLOADER

#define HI_INT_ADDR 0x1008
#define LO_INT_ADDR 0x1018

extern void _startup(void);
#pragma code RESET = 0x1000
void _reset (void)
{
    _asm goto _startup _endasm
}
```

```
#else /* USE_HID_BOOTLOADER */

#define HI_INT_ADDR 0x0008
#define LO_INT_ADDR 0x0018

#endif /* USE_HID_BOOTLOADER */

#pragma code HI_INT = HI_INT_ADDR
void hi_int_goto (void)
{
    _asm goto high_isr _endasm
}
#pragma code LO_INT = LO_INT_ADDR
void lo_int_goto (void)
{
    _asm goto low_isr _endasm
}
```

usb_descriptors.c – contém os descritores USB do dispositivo

```
#include "USB/usb.h"

#pragma romdata

/* Device descriptor */
ROM USB_DEVICE_DESCRIPTOR device_dsc = {
    0x12,                   /* Tamanho em bytes */
    USB_DESCRIPTOR_DEVICE,  /* Tipo */
    0x0200,                 /* Versão do USB */
    0x00,                   /* Classe */
    0x00,                   /* Subclasse */
    0x00,                   /* Protocolo */
    USB_EP0_BUFF_SIZE,      /* Tamanho do endpoint 0 */
    0x04D8,                 /* VID */
    0x000C,                 /* PID */
    0x0000,                 /* Relase number */
    0x01,                   /* String do fabricante */
    0x02,                   /* String do produto */
    0x00,                   /* String de serial */
    0x01                    /* Número de configurações */
};

/* Configuration descriptor 1 */
ROM BYTE configDescriptor1[] = {
    /* Configuration descriptor */
    0x09,                         /* Tamanho em bytes */
    USB_DESCRIPTOR_CONFIGURATION, /* Tipo */
    0x20,0x00,                    /* Tamanho total da configuration */
    1,                            /* Número de interfaces */
    1,                            /* Número dessa configuration */
    0,                            /* String que identifica essa configuration */
    _DEFAULT | _SELF,             /* Atributos, ver usb_device.h */
    50,                           /* Consumo máximo (2x mA) */

    /* Interface descriptor */
    0x09,                       /* Tamanho em bytes */
    USB_DESCRIPTOR_INTERFACE,   /* Tipo */
    0,                          /* Número dessa interface */
    0,                          /* Número dessa interface entre as alternativas */
    2,                          /* Número de endpoints */
```

```
    0xFF,                       /* Classe */
    0xFF,                       /* Subclasse */
    0xFF,                       /* Protocolo */
    0,                          /* String que identifica essa interface */

    /* Endpoint descriptors */
    0x07,                       /* Tamanho em bytes */
    USB_DESCRIPTOR_ENDPOINT,    /* Tipo */
    _EP01_OUT,                  /* Endereço */
    _BULK,                      /* Atributos */
    USBGEN_EP_SIZE,0x00,        /* Tamanho */
    1,                          /* Intervalo */

    0x07,                       /* Tamanho em bytes */
    USB_DESCRIPTOR_ENDPOINT,    /* Tipo */
    _EP01_IN,                   /* Endereço */
    _BULK,                      /* Atributos */
    USBGEN_EP_SIZE,0x00,        /* Tamanho */
    1                           /* Intervalo */
};

/* String descriptor (language code) */
ROM struct {
    BYTE bLength;
    BYTE bDscType;
    WORD string[1];
} sd000 = {
    sizeof(sd000),          /* Tamanho */
    USB_DESCRIPTOR_STRING,  /* Tipo */
    { 0x0409 }              /* Linguas*/
};

/* String do fabricante */
ROM struct {
    BYTE bLength;
    BYTE bDscType;
    WORD string[25];
} sd001 = {
    sizeof(sd001),          /* Tamanho */
    USB_DESCRIPTOR_STRING,  /* Tipo */
    /* String */
    { 'M', 'i', 'c', 'r', 'o', 'c', 'h', 'i', 'p', ' ',
      'T', 'e', 'c', 'h', 'n', 'o', 'l', 'o', 'g', 'y', ' ',
      'I', 'n', 'c', '.' }
};

/* String do produto */
ROM struct {
    BYTE bLength;
    BYTE bDscType;
    WORD string[27];
} sd002 = {
    sizeof(sd002),          /* Tamanho */
    USB_DESCRIPTOR_STRING,  /* Tipo */
    /* String */
    { 'M', 'i', 'c', 'r', 'o', 'c', 'h', 'i', 'p', ' ',
      'C', 'u', 's', 't', 'o', 'm', ' ',
      'U', 'S', 'B', ' ', 'D', 'e', 'v', 'i', 'c', 'e' }
};

/* Vetor com os Configuration descriptors */
ROM BYTE* ROM USB_CD_Ptr[] = {
```

```
    (ROM BYTE *ROM)&configDescriptor1
};

/* Vetor com os String descriptors */
ROM BYTE* ROM USB_SD_Ptr[] = {
    (ROM BYTE *ROM)&sd000,
    (ROM BYTE *ROM)&sd001,
    (ROM BYTE *ROM)&sd002
};
```

usb_config.h – configura os parâmetros da Framework USB

```
#ifndef USBCFG_H
#define USBCFG_H

/* Definições */

#define USB_EP0_BUFF_SIZE 8 /* Tamanho do endpoint 0 */
#define USB_MAX_NUM_INT 1 /* Número máximo de interfaces */
#define USB_MAX_EP_NUMBER 1 /* Maior número de endpoint usado */

/* Device descriptor */
#define USB_USER_DEVICE_DESCRIPTOR &device_dsc
#define USB_USER_DEVICE_DESCRIPTOR_INCLUDE extern ROM USB_DEVICE_DESCRIPTOR device_dsc

/* Configuration descriptor */
#define USB_USER_CONFIG_DESCRIPTOR USB_CD_Ptr
#define USB_USER_CONFIG_DESCRIPTOR_INCLUDE extern ROM BYTE *ROM USB_CD_Ptr[]

/* Opções do driver */
#define USB_POLLING
#define USB_PING_PONG_MODE USB_PING_PONG__FULL_PING_PONG
#define USB_PULLUP_OPTION USB_PULLUP_ENABLE
#define USB_TRANSCEIVER_OPTION USB_INTERNAL_TRANSCEIVER
#define USB_SPEED_OPTION USB_FULL_SPEED

#define USB_ENABLE_STATUS_STAGE_TIMEOUTS
#define USB_STATUS_STAGE_TIMEOUT (BYTE)45

#define USB_SUPPORT_DEVICE

#define USB_NUM_STRING_DESCRIPTORS 3

#define USB_ENABLE_SUSPEND_HANDLER
#define USB_ENABLE_WAKEUP_FROM_SUSPEND_HANDLER
#define USB_ENABLE_INIT_EP_HANDLER

#define USB_USE_GEN
#define USBGEN_EP_SIZE          64
#define USBGEN_EP_NUM            1

#endif /* USBCFG_H */
```

HardwareProfile.h – parâmetros extras da Framework USB, relacionados ao hardware

```
#ifndef HARDWARE_PROFILE_H
#define HARDWARE_PROFILE_H

#define self_power 1
#define USB_BUS_SENSE 1
```

```
#endif  /* HARDWARE_PROFILE_H */
```

fw_daq.c – implementa o protocolo

```
#include <p18f4550.h>
#include "USB/usb.h"
#include "USB/usb_function_generic.h"
#include "HardwareProfile.h"

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

#define CMD_IN_CONFIGURE 0xff
struct cmd_in_configure_t
{
	uint8_t command;
	 uint8_t trisa;
	 uint8_t trisb;
	 uint8_t trisc;
	 uint8_t trisd;
	uint8_t an_count;
	uint8_t adcon2;
	uint8_t t2con;
	uint8_t pr2;
};

#define CMD_OUT_CONFIGURE 0xff
struct cmd_out_configure_t
{
	uint8_t command;
};

#define CMD_IN_SET 0x00
struct cmd_in_set_t
{
	uint8_t command;
	 uint8_t lata;
	 uint8_t latb;
	 uint8_t latc;
	 uint8_t latd;
	uint8_t ccp1con;
	uint8_t ccpr1l;
	uint8_t ccp2con;
	uint8_t ccpr2l;
};

#define CMD_OUT_GET 0x00
struct cmd_out_get_t
{
	uint8_t command;
	 uint8_t porta;
	 uint8_t portb;
	 uint8_t portc;
	 uint8_t portd;
	uint16_t an[8];
};

#define LED_TRIS TRISDbits.TRISD0
#define LED_LAT LATDbits.LATD0
```

```
#define PORTA_MASK 0x10
#define PORTB_MASK 0xef
#define PORTC_MASK 0xc1
#define PORTD_MASK 0xdc

#pragma udata

USB_HANDLE USBGenericOutHandle;
USB_HANDLE USBGenericInHandle;

static volatile uint8_t analog_max;
static volatile uint8_t analog_i;
static volatile uint8_t analog_done;

#pragma udata USB_VARS = 0x500

union {
    struct cmd_in_configure_t cmd_in_configure;
    struct cmd_in_set_t cmd_in_set;
    BYTE bytes[USBGEN_EP_SIZE];
} receive_buffer;

struct cmd_out_configure_t cmd_out_configure;
struct cmd_out_get_t cmd_out_get;

#pragma code

static void configure_digital(uint8_t trisa, uint8_t trisb, uint8_t trisc, uint8_t
trisd)
{
    TRISA = (TRISA & ~PORTA_MASK) | (trisa & PORTA_MASK);
    TRISB = (TRISB & ~PORTB_MASK) | (trisb & PORTB_MASK);
    TRISC = (TRISC & ~PORTC_MASK) | (trisc & PORTC_MASK);
    TRISD = (TRISD & ~PORTD_MASK) | (trisd & PORTD_MASK);
}

static void configure_analog(uint8_t an_count, uint8_t adcon2)
{
    analog_max = ((an_count > 8) ? 8 : an_count);
    TRISA |= 0x2f;
    TRISE |= 0x07;
    ADCON0 = 0x00; /* Channel 0, disabled */
    ADCON1 = 0x07; /* Vdd/Vss reference, AN0-AN7 as analog */
    ADCON2 = 0x86 | (adcon2 & 0x31); /* Right justified, Fosc/64 */
}

static void configure_pwm(uint8_t t2con, uint8_t pr2)
{
    TRISCbits.TRISC2 = 0;
    TRISDbits.TRISD5 = 0;
    TRISCbits.TRISC1 = 0;
    T2CON = (t2con & 0x03);
    TMR2 = 0x00;
    PR2 = pr2;
    CCPR1L = 0x00;
    CCPR2L = 0x00;
    CCP1CON = 0x8c; /* Half-bridge, PWM P1A/P1B active-high */
    CCP2CON = 0x0c; /* PWM */
    ECCP1DEL = 0x00;
    ECCP1AS = 0x00;
    T2CONbits.TMR2ON = 1;
}
```

```
static void set_digital(uint8_t lata, uint8_t latb, uint8_t latc, uint8_t latd)
{
    LATA = (LATA & ~PORTA_MASK) | (lata & PORTA_MASK);
    LATB = (LATB & ~PORTB_MASK) | (latb & PORTB_MASK);
    LATC = (LATC & ~PORTC_MASK) | (latc & PORTC_MASK);
    LATD = (LATD & ~PORTD_MASK) | (latd & PORTD_MASK);
}

static void set_pwm(uint8_t ccp1con, uint8_t ccpr1l, uint8_t ccp2con, uint8_t ccpr2l)
{
    CCP1CON = (CCP1CON & 0xcf) | (ccp1con & 0x30);
    CCPR1L = ccpr1l;
    CCP2CON = (CCP2CON & 0xcf) | (ccp2con & 0x30);
    CCPR2L = ccpr2l;
}

static void get_digital(void)
{
    cmd_out_get.porta = PORTA;
    cmd_out_get.portb = PORTB;
    cmd_out_get.portc = PORTC;
    cmd_out_get.portd = PORTD;
}

static void get_analog(void)
{
    if (analog_max == 0)
    {
        return;
    }
    PIR1bits.ADIF = 0;
    INTCONbits.GIE = 1;
    analog_i = 0;
    analog_done = 0;
    ADCON0 = (analog_i << 2);
    ADCON0bits.ADON = 1;
    ADCON0bits.GO = 1;
    while (!analog_done)
    {
        USBDeviceTasks();
    }
    INTCONbits.GIE = 0;
}

void daq_init(void)
{
    /* Reseta os registradores de interrupção */
    RCONbits.IPEN = 0;
    INTCON = 0x00;
    INTCON2 = 0x80; /* Disable PORTB pull-ups */
    INTCON3 = 0x00;
    PIR1 = 0x00;
    PIR2 = 0x00;
    PIE1 = 0x00;
    PIE2 = 0x00;
    IPR1 = 0x00;
    IPR2 = 0x00;
    /* Reseta todos os latches de saída */
    LATA = 0x00;
    LATB = 0x00;
    LATC = 0x00;
```

```
    LATD = 0x00;
    LATE = 0x00;
    /* Reseta todos os pinos para entradas digitais */
    TRISA = 0xff;
    TRISB = 0xff;
    TRISC = 0xff;
    TRISD = 0xff;
    TRISE = 0xff;
    ADCON1 = 0x0f;
    /* Configura e inicializa o pino do LED */
    LED_TRIS = 0;
    LED_LAT = 0;
    /* Configura interrupções de ADC */
    PIE1bits.ADIE = 1;
    INTCONbits.PEIE = 1;
    /* Inicializa o USB */
    USBGenericOutHandle = 0;
    USBGenericInHandle = 0;
    USBDeviceInit();
}

void daq_high_isr(void)
{
    PIR1bits.ADIF = 0;
    cmd_out_get.an[analog_i] = ADRES;
    if ((++analog_i) < analog_max)
    {
        ADCON0 = (analog_i << 2);
        ADCON0bits.ADON = 1;
        ADCON0bits.GO = 1;
    }
    else
    {
        analog_done = 1;
    }
}

void daq_low_isr(void)
{
    ;
}

static void handle_cmd_in_configure(void)
{
    if (USBHandleGetLength(USBGenericOutHandle) != sizeof(struct cmd_in_configure_t))
    {
        return;
    }
    configure_digital(
        receive_buffer.cmd_in_configure.trisa,
        receive_buffer.cmd_in_configure.trisb,
        receive_buffer.cmd_in_configure.trisc,
        receive_buffer.cmd_in_configure.trisd);
    configure_analog(
        receive_buffer.cmd_in_configure.an_count,
        receive_buffer.cmd_in_configure.adcon2);
    configure_pwm(
        receive_buffer.cmd_in_configure.t2con,
        receive_buffer.cmd_in_configure.pr2);
    if (!USBHandleBusy(USBGenericInHandle))
    {
        cmd_out_configure.command = CMD_OUT_CONFIGURE;
```

```
        USBGenericInHandle = USBGenWrite(USBGEN_EP_NUM,
            (BYTE*)&cmd_out_configure, sizeof(cmd_out_configure));
    }
}

static void handle_cmd_in_set(void)
{
    if (USBHandleGetLength(USBGenericOutHandle) != sizeof(struct cmd_in_set_t))
    {
        return;
    }
    set_digital(
        receive_buffer.cmd_in_set.lata,
        receive_buffer.cmd_in_set.latb,
        receive_buffer.cmd_in_set.latc,
        receive_buffer.cmd_in_set.latd);
    set_pwm(
        receive_buffer.cmd_in_set.ccp1con,
        receive_buffer.cmd_in_set.ccpr1l,
        receive_buffer.cmd_in_set.ccp2con,
        receive_buffer.cmd_in_set.ccpr2l);
    if (!USBHandleBusy(USBGenericInHandle))
    {
        cmd_out_get.command = CMD_OUT_GET;
        get_digital();
        get_analog();
        USBGenericInHandle = USBGenWrite(USBGEN_EP_NUM,
            (BYTE*)&cmd_out_get, sizeof(cmd_out_get));
    }
}

void daq_loop(void)
{
    USBDeviceTasks();
    if((USBDeviceState < CONFIGURED_STATE) || (USBSuspendControl == 1))
    {
        return;
    }
    if(!USBHandleBusy(USBGenericOutHandle))
    {
        static uint8_t led_count = 50;
        if ((--led_count) == 0)
        {
            led_count = 100;
            LED_LAT = !LED_LAT;
        }
        switch (receive_buffer.bytes[0])
        {
            case CMD_IN_CONFIGURE:
                handle_cmd_in_configure();
                break;
            case CMD_IN_SET:
                handle_cmd_in_set();
                break;
            default:
                break;
        }
        USBGenericOutHandle = USBGenRead(USBGEN_EP_NUM, (BYTE*)&receive_buffer,
USBGEN_EP_SIZE);
    }
}
```

```
void USBCBSuspend(void)
{
    OSCCON = 0x13; /* Reduz o clock */
}

void USBCBWakeFromSuspend(void)
{
    OSCCON = 0x60; /* Volta para o clock primário */
    {
        unsigned int pll_startup_counter = 800;
        while(pll_startup_counter--);
    }
}

void USBCBInitEP(void)
{
    /* Habilita o endpoint 1 */
    USBEnableEndpoint(USBGEN_EP_NUM,
USB_OUT_ENABLED|USB_IN_ENABLED|USB_HANDSHAKE_ENABLED|USB_DISALLOW_SETUP);
    /* Prepara o endpoint 1 para receber dados */
    USBGenericOutHandle = USBGenRead(USBGEN_EP_NUM, (BYTE*)&receive_buffer,
USBGEN_EP_SIZE);
}

BOOL USER_USB_CALLBACK_EVENT_HANDLER(int event, void* pdata, WORD size)
{
    switch(event)
    {
        case EVENT_SUSPEND:
            USBCBSuspend();
            break;
        case EVENT_RESUME:
            USBCBWakeFromSuspend();
            break;
        case EVENT_CONFIGURED:
            USBCBInitEP();
            break;
        default:
            break;
    }
    return TRUE;
}
```

fw_daq.h – define as funções usadas pela main.c

```
#ifndef FW_DAQ_H
#define FW_DAQ_H

void daq_init(void);
void daq_loop(void);
void daq_high_isr(void);
void daq_low_isr(void);

#endif /* FW_DAQ_H */
```

Além do código, é necessário um *linker script* para que o firmware compilado possa funcionar corretamente com o bootloader:

18f4550_bl.lkr

```
// File: 18f4550_g.lkr
// Generic linker script for the PIC18F4550 processor

#DEFINE _CODEEND _DEBUGCODESTART - 1
#DEFINE _CEND _CODEEND + _DEBUGCODELEN
#DEFINE _DATAEND _DEBUGDATASTART - 1
#DEFINE _DEND _DATAEND + _DEBUGDATALEN

LIBPATH .

#IFDEF _CRUNTIME
  #IFDEF _EXTENDEDMODE
    FILES c018i_e.o
    FILES clib_e.lib
    FILES p18f4550_e.lib

  #ELSE
    FILES c018i.o
    FILES clib.lib
    FILES p18f4550.lib
  #FI

#FI

CODEPAGE     NAME=bootloader START=0x0               END=0xFFF        PROTECTED
#IFDEF _DEBUGCODESTART
  CODEPAGE   NAME=page       START=0x1000            END=_CODEEND
  CODEPAGE   NAME=debug      START=_DEBUGCODESTART   END=_CEND        PROTECTED
#ELSE
  CODEPAGE   NAME=page       START=0x1000            END=0x7FFF
#FI

CODEPAGE   NAME=idlocs     START=0x200000          END=0x200007       PROTECTED
CODEPAGE   NAME=config     START=0x300000          END=0x30000D       PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE          END=0x3FFFFF       PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000          END=0xF000FF       PROTECTED

#IFDEF _EXTENDEDMODE
  DATABANK   NAME=gpre       START=0x0               END=0x5F
#ELSE
  ACCESSBANK NAME=accessram  START=0x0               END=0x5F
#FI

DATABANK   NAME=gpr0       START=0x60              END=0xFF
DATABANK   NAME=gpr1       START=0x100             END=0x1FF
DATABANK   NAME=gpr2       START=0x200             END=0x2FF

#IFDEF _DEBUGDATASTART
  DATABANK   NAME=gpr3       START=0x300             END=_DATAEND
  DATABANK   NAME=dbgspr     START=_DEBUGDATASTART   END=_DEND           PROTECTED
#ELSE //no debug
  DATABANK   NAME=gpr3       START=0x300             END=0x3FF
#FI

DATABANK   NAME=gpr4       START=0x400             END=0x4FF
DATABANK   NAME=gpr5       START=0x500             END=0x5FF
DATABANK   NAME=gpr6       START=0x600             END=0x6FF
DATABANK   NAME=gpr7       START=0x700             END=0x7FF
ACCESSBANK NAME=accesssfr  START=0xF60             END=0xFFF          PROTECTED

#IFDEF _CRUNTIME
  SECTION    NAME=CONFIG     ROM=config
```

```
  #IFDEF _DEBUGDATASTART
    STACK SIZE=0x100 RAM=gpr2
  #ELSE
    STACK SIZE=0x100 RAM=gpr3
  #FI
#FI
```

Uma vez compilado, o firmware pode ser enviado ao microcontrolador utilizando o HID Bootloader, da Microchip, e a placa estará pronta para ser usada.

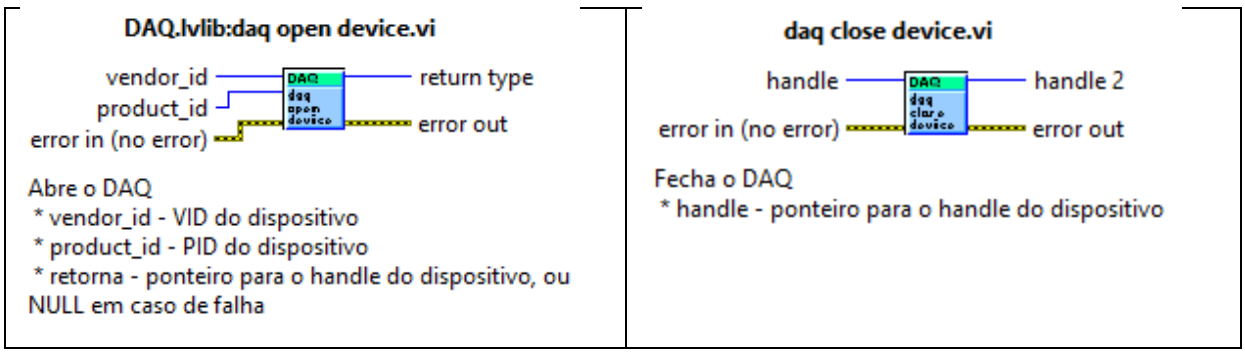
11.8 *LabVIEW*

O *software* da *National Instruments* é o responsável pela manipulação da biblioteca e suas funções, no computador. Assim, é o responsável pela comunicação e interface entre este e o microcontrolador.

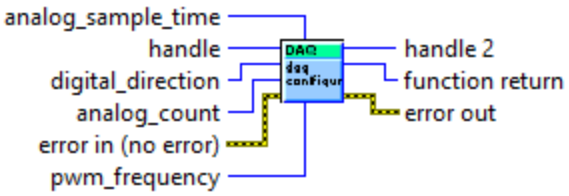
As funções são exploradas no *LabVIEW* através do *Call Library Function Node,* que transforma uma função, com seus parâmetros de saída e entrada, em um bloco funcional do programa.

Para simplificar e condensar todos os recursos de uma função em um único bloco, um *subVI* foi criado para cada função, e em seu interior encontram-se os controles e indicadores dos parâmetros a serem utilizados, o tipo de dependência aplicável a cada parâmetro (se é obrigatório, recomendável ou opcional) e quaisquer manipulações obrigatórias que sejam feitas para adaptação dos dados às entradas ou saídas. Assim, esses recursos tornam-se invisíveis, numa primeira instância, ao usuário, simplificando o uso de tais funções.

Os *subVIs* criados, com seus respectivos parâmetros, são os seguintes:



DAQ.lvlib:daq configure.vi



Configura as entradas e saídas do DAQ
* handle - ponteiro para o handle do dispositivo
* digital_direction - bitmask com a direção das portas digitais (0=Output, 1=Input), D0 no bit 0, D1 no bit 1, e assim por diante
* analog_count - número de entradas analógicas usadas (de 0 a 8)
* analog_sample_time - tempo mínimo de amostragem, em micro segundos (valor real será maior ou igual a esse valor)
* pwm_frequency - frequência do PWM em hertz (valor real será bastante próximo, com a maior resolução possível)
* retorna - 0=falha, 1=sucesso

DAQ.lvlib:daq read analog.vi

Lê uma entrada analógica
* handle - ponteiro para o handle do dispositivo
* channel - número da entrada analógica (de 0 a 7)

daq read digital.vi

Lê uma entrada digital
* handle - ponteiro para o handle do dispositivo
* channel - número da entrada digital (de 0 a 15)

daq set duty cycle.vi

Altera o duty cycle do PWM
* handle - ponteiro para o handle do dispositivo
* channel - número do canal de PWM (1 ou 2)
* duty_cycle - duty cycle em porcentagem (de 0 a 100)

DAQ.lvlib:daq update.vi

Faz a comunicação com o microcontrolador, escrevevendo as saídas e lendo as entradas
* handle - ponteiro para o handle do dispositivo
* retorna - 0=falha, 1=sucesso

daq write digital.vi

Escreve em uma saída digital
* handle - ponteiro para o handle do dispositivo
* channel - número da saída digital (de 0 a 15)
* state - 0=nível baixo, 1=nível alto

Um típico VI utilizado para aquisição de dados e/ou controle, tem um fluxo de dados que respeita, de alguma forma, uma sequência de ações.

Primeiramente, estando o dispositivo já conectado ao computador, deve-se abrir o canal de comunicação entre o computador e o dispositivo. Isso é possível identificando-o,

dentre os dispositivos conectados, através de seu *Product ID* (0x04D8) e *Vendor ID* (0x000C). Depois, a configuração dos recursos desejados é realizada.

A próxima etapa é a em que ocorre toda a leitura, escrita e manipulação – pelo menos inicial – dos dados. Geralmente, a aquisição é feita dentro de um laço de repetição, que possa depender de alguma condição de parada ou que seja limitado a um número de iterações, de modo com que a quantidade de dados necessária seja transferida. O programa é finalizado, geralmente, ao se interromper o laço de repetição. Assim, ao sair do laço, o último passo é fechar o canal de comunicação para que o dispositivo possa ser liberado novamente.

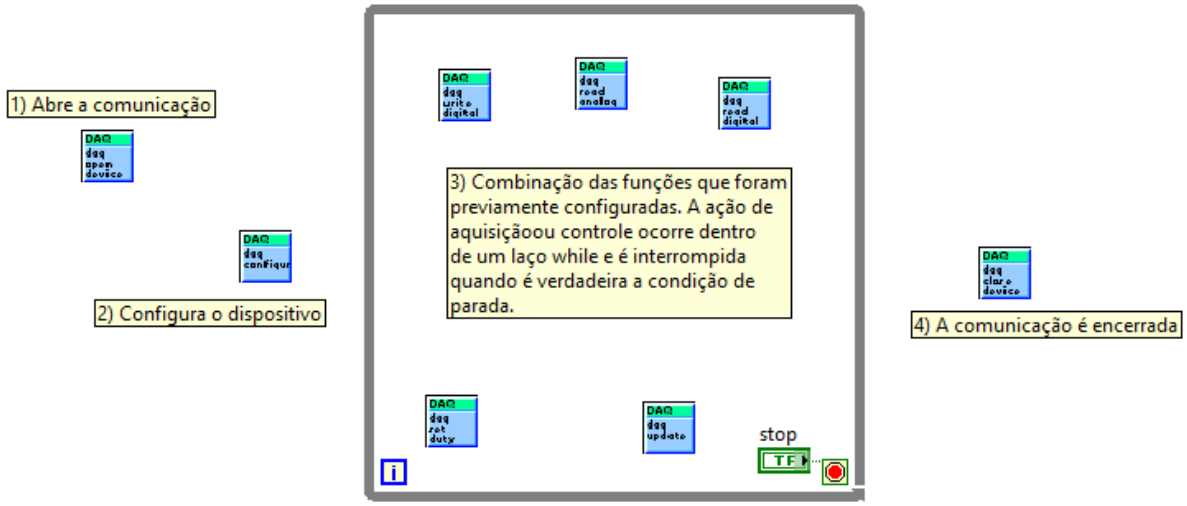


Figura 36 - Típico fluxo de dados em um VI de aquisição e/ou controle.

11.9 Recursos

O dispositivo de aquisição de dados e controle foi desenvolvido para ser capaz de suprir as principais necessidades em um ambiente de automação voltado ao aprendizado, em ambientes escolares e acadêmicos, ou para projetos domésticos. Assim, os recursos presentes no equipamento são os seguintes:

- 16 I/O digitais
- 8 ADC de 10 *bits*, configuráveis, com capacidade de até 1 kS/s (*kiloSamples per second*, milhares de amostras por segundo)
- 1 saída PWM com resolução de 10 *bits*, configurável
- 1 módulo PWM *half bridge* com resolução de 10 *bits*, configurável
- Programação via *software*

11.10 Custo x benefício

Para verificação dos preços de mercado dos componentes, uma cotação foi realizada em três diferentes estabelecimentos, que seguem.

- Farnell Newark, http://www.farnellnewark.com.br
- Solda Fria, http://www.soldafria.com.br
- Pinhé Componentes Eletrônicos, http://www.pinhe.com.br

Assim, obtivemos a seguinte média de preços para os componentes citados.

Quantidade	**Item**	**Unitário (R$)**	**Total (R$)**
1	Placa fenolite 15 x 20 cm	6,70	6,70
2	Capacitor cerâmico 22 pF	0,20	0,40
2	Capacitor poliéster 0,1 μF	0,25	0,50
1	Capacitor poliéster 0,47 μF	0,70	0,70
1	Soquete padrão DIP 40 pinos	2,80	2,80
1	Microcontrolador PIC18F4550	22,90	22,90
1	*Header* de 6 pinos	0,20	0,20
1	*Jumper* sem aba	0,20	0,20
1	Led difuso 5mm cor verde	1,00	1,00
1	Cristal 24 MHz	1,20	1,20
2	Resistor 1 $k\Omega$	0,01	0,02
1	Resistor 330 Ω	0,01	0,01
2	Chave táctil 4 terminais 5 mm	0,25	0,50
1	Conector USB-B fêmea	2,40	2,40

12	Borne KRE 2 terminais 5 mm	0,45	5,40
3	Borne KRE 3 terminais 5 mm	1,50	4,50
Valor Total (R$)			49,43

A partir do custo do nosso equipamento, é possível validar sua posição proposta como produto de uso acadêmico e doméstico. Como comparação, podemos citar alguns produtos que se enquadram nesse mercado.

O *kit* de desenvolvimento Arduino é amplamente difundido e, apesar de não ser definido exatamente como plataforma de aquisição de dados, mas como já citado, módulo de desenvolvimento, conta, em seu módulo básico, com recursos bastante semelhantes aos propostos nesse trabalho. É vendido por cerca de R$ 120,00.

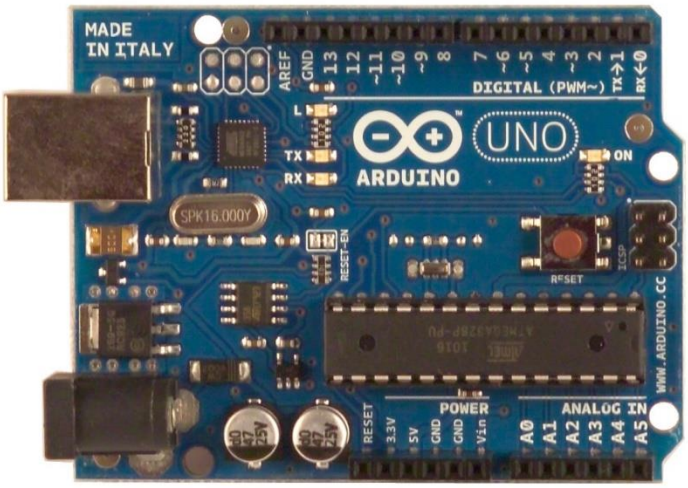


Figura 37 - Arduino.

Na linha de produtos para aquisição de dados, propriamente, temos equipamentos como o *Easy Lab I/O*, dotado de 10 I/O digitais, 1 sinal PWM de 10 *bits* e contador de pulsos, conectado via USB 2.0 (FS-USB), que é vendido por cerca de R$ 200,00.

Figura 38 - Easy Lab I/O.

A *National Instruments* possui soluções profissionais, inclusive com equipamentos capazes de atuar em *real time*. Seu modelo de entrada, o NI USB-6009, já é bastante superior e indicado para algumas aplicações que necessitem de mais recursos. Possui 8 entradas analógicas de *14 bits*, com capacidade de leitura de 48 kS/s (*kiloSamples per second*, ou milhares de amostras por segundo); 2 saídas analógicas de 12 *bits* com capacidade de 150 S/s; 12 I/O digitais; contador de 32 *bits*. Nota-se claramente a maior disponibilidade e extensão de recursos. O NI USB-6009 é vendido por cerca de R$ 1000,00, no Brasil.

Figura 39 - NI USB-6009.

Levando em conta o público a que se destina e as soluções a que se deseja atender, temos um equipamento que se enquadra entre um *kit* básico de desenvolvimento – servindo a esse propósito com os recursos de controle – e um equipamento robusto para aquisição de dados – superior aos modelos "genéricos" comumente encontrados no mercado, mas ainda restrito em relação à linha de entrada de equipamentos de referência. Então, como nossa proposta é oferecer um recurso robusto e eficiente para aquisição de dados e controle dentro dos parâmetros já apresentados, temos um equipamento barato e ainda assim superior em termos de disponibilidade de recursos.

12 Considerações finais

O dispositivo para aquisição e controle desenvolvido neste trabalho pode ser usado para uma grande variedade de aplicações de maneira satisfatória, principalmente onde o custo é o fator preponderante, devido à sua gama de recursos a um custo baixo. A comunicação é bastante robusta, graças à utilização do USB, que garante a velocidade de transmissão e a integridade dos dados. A sua fácil utilização com o ambiente *LabVIEW*, que é bastante usado em aplicações de aquisição e controle, torna-o bastante atrativo para usuários domésticos e alunos universitários.

Como citado anteriormente, esse dispositivo foi utilizado com êxito em diversas disciplinas do curso de Engenharia Mecatrônica, principalmente as de cunho prático. Um fator importante para esse sucesso, além da utilização do ambiente *LabVIEW*, foi a disponibilidade do dispositivo para testes sem a necessidade de utilizar os laboratórios, pois o dispositivo necessita apenas de uma conexão USB para funcionar. Assim, o aluno tem virtualmente todo o tempo possível para desenvolver as aplicações, enquanto testa diretamente com o *hardware*.

Um uso sugerido para este dispositivo seria como um *kit* a ser construído durante o primeiro ano do curso de engenharia. Os alunos seriam apresentados a microcontroladores, componentes e placas eletrônicos e ao ambiente *LabVIEW*, englobando uma boa parte das áreas de conhecimento da Engenharia Mecatrônica, além de ter amplo contato com a prática, como a produção de uma placa eletrônica e a soldagem de componentes. Cada aluno, ou em grupos, construir uma placa, a mesma poderia ser usada durante todo o curso, a fim de desenvolver projetos mais avançados nas disciplinas de caráter prático, tais como as disciplinas de Problemas de Engenharia Mecatrônica.

A versatilidade da solução desenvolvida deve ser reforçada. Ao encapsular o protocolo de comunicação com o microcontrolador em uma biblioteca, é possível comunicar com a placa utilizando outras linguagens que não somente a do ambiente *LabVIEW*, sendo possível utilizar C/C++, MATLAB, ou *Python*. O único requisito é que seja possível carregar a biblioteca e utilizar as suas funções. Devido à presença do *bootloader*, que permite que um *firmware* qualquer seja gravado no microcontrolador, também é possível utilizar a placa para

a aprendizagem e desenvolvimento com microcontroladores, de maneira geral, especificamente o modelo PIC18F4550 da *Microchip*, que, como apresentado, possui diversos recursos diferentes, além daqueles utilizados na aplicação deste trabalho.

Referências

About Us: CadSoft Computer. (2011). Acesso em 04 de Dezembro de 2012, disponível em CadSoft Computer: http://www.cadsoftusa.com/about-us/

Axelson, J. (2009). *USB Complete: The Developer's Guide* (4ª ed.). Lakeview Research.

Cerne. (2008). *http://www.cerne-tec.com.br.* Acesso em 4 de Dezembro de 2012, disponível em Cerne: http://www.cerne-tec.com.br/labview.pdf

Datasheet PIC18F4550. (27 de Outubro de 2009). Acesso em 26 de Outubro de 2012, disponível em Microchip: http://ww1.microchip.com/downloads/en/DeviceDoc/39632e.pdf

Eagle Manual Version 6. (Maio de 2011). Acesso em 04 de Dezembro de 2012, disponível em CadSoft Computer: http://www.cadsoft.de/wp-content/uploads/2011/05/V6_manual_en.pdf

Larsen, R. W. (2010). *LabVIEW for Engineers.* Prentice Hall.

Lyons, R. G. (1996). *Understanding Digital Signal Processing* (1ª ed.). Pearson Education.

Miyadaira, A. N. (2009). *Microcontroladores PIC18 - Aprenda e Program em Linguagem C* (1ª ed.). Editora Érica Ltda.

MPLAB C18 C Compiler User's Guide. (2005). Acesso em 28 de Outubro de 2012, disponível em Microchip: http://ww1.microchip.com/downloads/en/devicedoc/51288f.pdf

Schildt, H. (1996). *C Completo e Total, Edição Revista e atualizada* (3ª ed.). Pearson Education do Brasil.

Tocci, R. J., & Widmer, N. S. (1998). *Sistemas Digitais, Princípios e Aplicações* (7ª ed.). LTC.

USB 2.0 Specification. (27 de Abril de 2000). Acesso em 28 de Outubro de 2012, disponível em Universal Serial Bus: http://www.usb.org/developers/docs/

Wilmshurst, T. (2009). *Designing Embedded Systems with PIC Microcontrollers: Principles and Applications* (2ª ed.). Newnes.