SISTEMA DE MEDIÇÃO DE TEMPO DE RESPOSTA:

UMA APLICAÇÃO PARA A MEDIÇÃO DE TEMPO DE REFLEXO E TREINAMENTO DE ATLETAS

Trabalho de Conclusão de Curso

Engenharia de Computação

Pedro Buarque Caminha Monteiro
Orientador: Prof. Sérgio Campello

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação

PEDRO BUARQUE CAMINHA MONTEIRO

SISTEMA DE MEDIÇÃO DE TEMPO DE RESPOSTA:

UMA APLICAÇÃO PARA A MEDIÇÃO DE TEMPO DE REFLEXO E TREINAMENTO DE ATLETAS

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, Junho de 2014

Declaro que revisei o Trabalho de Conclusão de Curso sob o título "SISTEMA DE MEDIÇÃO DE TEMPO DE RESPOSTA: UMA APLICAÇÃO PARA A MEDIÇÃO DE TEMPO DE REFLEXO E TREINAMENTO DE ATLETAS", e que estou de acordo com a entrega do mesmo.

Recife, ____/_____/_____

_____

Prof. Dr. Sérgio Campello Oliveira

Orientador

Dedico este trabalho a minha família e minha namorada.

Agradecimentos

Agradeço primeiramente a Deus por ter chegado até aqui. Agradeço também a toda minha família por ter me dado suporte nos momentos difíceis e me ajudar a continuar minha jornada. Agradeço a eles pelo apoio e conselhos que me ajudaram a tomar as melhores decisões. Em especial aos meus pais Luiz Carlos e Rita de Cássia e minha irmã Raquel.

Agradeço a minha namorada Deyse por estar sempre presente na minha vida, me dando forças e me encorajando a nunca desistir. Agradeço também a sua família pelo apoio.

Não poderia deixar de agradecer aos meus amigos de curso: Eduardo Salgado, Vanderson Pessoa, Hugo Leonardo, Ronney Bezzerra, Cleyton Silva, Karolyne Cavalcanti pelos momentos de desespero que passamos juntos e também pelos momentos de alegria.

Por fim, gostaria de agradecer a meu orientador, Sérgio Campello, pela paciência, por sempre tirar minhas dúvidas mesmo nos momentos mais impróprios, por ter me ajudado muito no projeto e por sua enorme contribuição em minha vida acadêmica.

Resumo

O aumento da complexidade dos projetos de sistemas embarcados tem exigido novos níveis de abstração em soluções de *software* que possam interagir com o *hardware* da forma mais eficiente possível. Dentro deste contexto, destacam-se sistemas que trabalham com medições de tempo de respostas a determinados estímulos ou eventos. A proposta deste trabalho é a implementação de um sistema computacional embarcado para realizar a medição de tempo entre eventos utilizando nós sensores. Como teste inicial será desenvolvido um sistema completo, composto de *hardware*, *firmware* e *software*, capaz de auxiliar no treinamento do tempo de reação de atletas. O *hardware* desenvolvido neste projeto é dividido em três partes: uma placa microcontrolada que ficará responsável por receber comandos de um computador e transmiti-los para cada uma das placas de interação com o atleta, obedecendo todas as configurações definidas no *software*; uma placa de conexões que fará as ligações entre a placa microcontrolada e todas as placas de interação; uma placa de interação para representar todos os nós sensores. O *firmware* desenvolvido na linguagem C, tem o objetivo de receber comandos do *software* e enviar o tempo de resposta . O *software*, desenvolvido na linguagem C#, tem o objetivo de enviar comandos para a placa microcontrolada, utilizando comunicação USB, e armazenar os tempos de resposta obtidos. O sistema obteve bons resultados quando testado na ferramenta de simulação, e a capacidade de geração de relatórios garante uma maior facilidade ao avaliar o desempenho do atleta.

Palavras-chave: Tempo de resposta, Microcontrolador, Sistema de medição.

Abstract

The increasing complexity of embedded system design has required new levels of abstraction in *software* solutions that interact with the *hardware* in the most efficient way possible. Within this context, we highlight systems that work with time measurements of responses to certain impulse or events. The purpose of this work is the implementation of an embedded computer system to perform the measurement of time between events using sensor nodes. As an initial test, a complete system consisting of *hardware*, *firmware* and *software*, able to assist in the reaction time of athletes training will be developed. The *hardware* developed in this project is divided into three parts: a microcontrolled board that will be responsible for receiving commands from a computer and transmit them to each interaction board, obeying all the settings defined in *software*; a board that will make connections between the microcontrolled board and all interaction boards; a interaction board to represent all sensor nodes. The *firmware* developed in C language, has the purpose of receiving *software* commands and send the response time. The *software*, developed in C# language, aims to send commands to the microcontrolled board using USB communication, and store the response times obtained. The system obtained good results when tested in the simulation tool, and the ability to generate reports ensures greater ease on evaluate the athlete's performance.

Keywords: Response time, Microcontroller, Mesuring system

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Siglas

CDC – *Communications Device Class*

CISC – *Complex Instruction Set Computer*

CPU – *Central Processing Unit*

DPDT – *Double-Pole Double-Throw*

EEPROM – *Electrically-Erasable Programmable Read-Only Memory*

HID – *Human Interface Device*

LED – *Light Emitting Diode*

PIC – *Programmable Interface Controller*

PLL – *Phase Lock Loop*

PWM – *Pulse Width Modulation*

RAM – *Random Access Memory*

RISC – *Reduced instruction set computing*

SIE – *Serial Interface Engine*

ULA – *Unidade Lógica e Aritmética*

USB – *Universal Serial Bus*

Capítulo 1
Introdução

Com a necessidade de sistemas computacionais cada vez mais compactos e eficientes, a utilização de microcontroladores se tornou mais evidente em muitas áreas de aplicação. Segundo Prado (2010), o aumento da complexidade dos projetos de sistemas embarcados tem exigido novos níveis de abstração em soluções de *software* que possam interagir com o *hardware* da forma mais eficiente possível.

Dentro deste contexto, destacam-se sistemas que trabalham com medições de tempo de respostas a determinados estímulos ou eventos. Um exemplo são os sistemas de tempo real, onde suas respostas ao ambiente devem ser dadas dentro de um tempo hábil o suficiente para que o sistema não entre em um estado inconsciente. Para Joseph (1986), um dos problemas de desempenho em um sistema em tempo real, onde um processador tem um conjunto de dispositivos conectados a ele em diferentes níveis de prioridade, é conseguir determinar o tempo de resposta de cada dispositivo. Análises de tempo de reação são úteis também, por exemplo, para identificar melhorias no estado cognitivo de idosos que praticam atividades físicas (ROSSATO, 2011), ou aumentar o desempenho de atletas nos esportes (SILVA, 2008).

A proposta deste trabalho é a implementação de um sistema computacional embarcado para realizar a medição de tempo entre eventos utilizando nós sensores. Como teste inicial será desenvolvido um sistema capaz de auxiliar no treinamento do tempo de reação de atletas. Como mostra a Figura 1, o treinamento será realizado a partir de um *software* controlado pelo usuário, que irá enviar comandos para uma placa microcontrolada. Esta ficará responsável por repassar esses comandos para cada um dos 24 nós sensores que funcionarão como estímulos para o atleta. Deverá ser mensurado o intervalo de tempo entre o recebimento do estímulo por parte do atleta e a ação de resposta realizada por ele ao apertar um botão, indicando o fim da contagem.

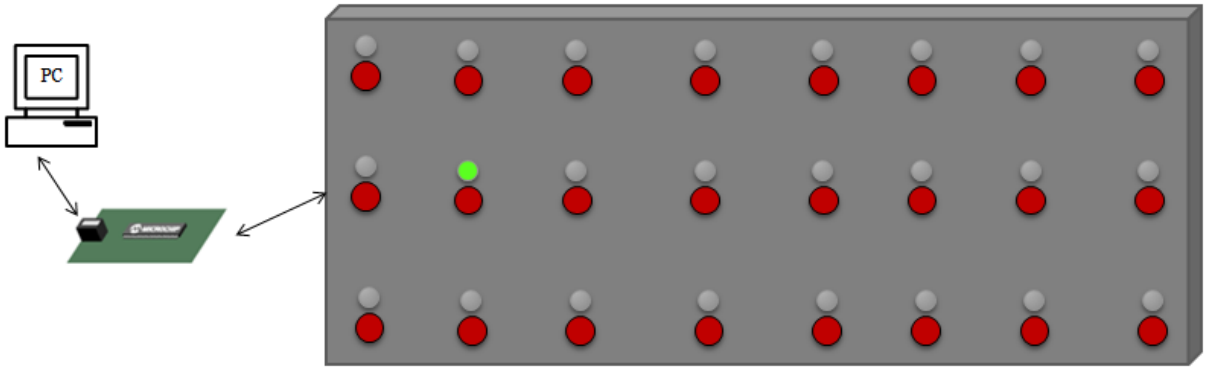

Figura 1. Treinamento do tempo de resposta.

É importante ressaltar que foi escrito um artigo científico com o mesmo título deste trabalho e publicado no XLII Congresso Brasileiro de Educação em Engenharia (COBENGE 2014).

1.1 Objetivo Geral

Esse trabalho tem como objetivo principal a implementação de um sistema computacional embarcado para medição de tempo entre eventos, como teste inicial será desenvolvido um sistema de medição para auxiliar no treinamento do tempo de reação de atletas.

1.2 Objetivos Específicos

Entre os objetivos específicos deste trabalho, estão:

I. Projetar e simular os circuitos elétricos e *layouts* de placas de circuito impresso.
II. Implementar o código do microcontrolador PIC na linguagem C, juntamente com o sistema de interface com o usuário que será feito na linguagem C#.
III. Estudar o funcionamento da comunicação via USB do *hardware* com o sistema que será desenvolvido.
IV. Confeccionar todo o *hardware* necessário para a realização do treinamento.
V. Realizar os testes necessários para garantir o funcionamento do sistema.

1.3 Estrutura do documento

Este trabalho está dividido em 6 capítulos, incuindo este capítulo, que apresenta uma breve introdução ao tema e os objetivos do projeto. Em seguida o Capítulo 2 traz informações acerca dos microcontroladores da família PIC de uma forma mais geral. No Capítulo 3 é abordada uma revisão bibliográfica sobre a comunicação USB para os microcontroladores. O Capítulo 4 apresenta tudo que foi realizado durante o desenvolvimento do projeto. Os testes e resultados alcançados neste trabalho serão mostrados no Capítulo 5, em seguida a conclusão e trabalhos futuros serão comentados no Capítulo 6.

Capítulo 2

Microcontroladores

Os microcontroladores são circuito integrados que têm: processador, pinos de entradas/saídas e memória. Por meio da programação dos microcontroladores pode-se controlar suas saídas, tendo como referência as entradas ou um programa interno (MARTINS, 2005). Os microcontroladores estão presentes em quase tudo o que envolve a eletrônica, diminuindo o tamanho, facilitando a manutenção e gerenciando tarefas internas de aparelhos eletroeletrônicos.

A lógica de operação de um microcontrolador é estruturada na forma de um programa e gravada dentro do componente. Depois disso, toda vez que o microcontrolador for alimentado, o programa interno será executado. Um dos elementos básicos do microcontrolador é a Unidade Lógica e Aritmética (ULA), onde são executadas operações aritméticas e operações lógicas relacionais como deslocamento, transferência, comparação, classificação, etc. Quanto mais poderosa a ULA do componente, maior sua capacidade de processar informações (SOUZA, 2001).

Uma das características fundamentais que diferencia os microcontroladores dos microprocessadores é a capacidade de encapsular todos os componentes necessários ao controle de um processo, ou seja, o microcontrolador está provido

internamente de memória de programa, memória de dados, portas de entrada e/ou saída, *timers* e contadores, comunicação serial, PWMs, conversores analógico-digitais, etc.

Atualmente existem diversas famílias de microcontroladores disponíveis no mercado, cada uma com suas características próprias. Neste capítulo será feito um aprofundamento sobre os microcontroladores da família PIC, fabricados pela *Microchip Tecnology,* que foram utilizados no desenvolvimento deste projeto.

2.1 Arquitetura dos microcontroladores PIC

Segundo MARTINS (2005), o alto desempenho da família de microcontroladores PIC pode ser atribuído às seguintes características de arquitetura RISC:

- Mapa de registradores versátil;
- Todas as instruções com palavras simples;
- Palavra de instrução longa;
- Arquitetura de instruções em *pipeline*;
- Instruções de apenas um ciclo de máquina (duas no máximo);
- Conjunto de instruções reduzido;
- Conjunto de instruções ortogonal (simétrico).

Os microcontroladores PIC apresentam uma estrutura de máquina interna do tipo Harvard, enquanto grande parte dos microcontroladores tradicionais apresenta uma arquitetura tipo Von-Neumann. A diferença está na forma como os dados e o programa são acessados (SOUZA, 2001).

Na arquitetura tradicional, tipo Von-Neumann, existe apenas um barramento interno, por onde passam as instruções e os dados. O mesmo barramento é utilizado tanto para fazer a busca de instruções na memória do programa quanto para acessar (escrever ou ler) a memória de dados (Figura 2).

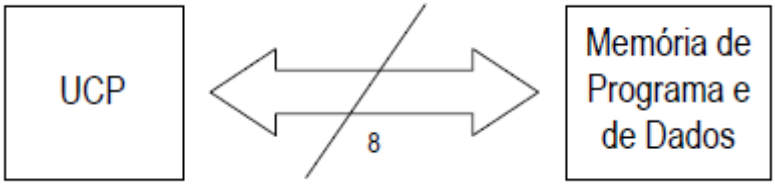

Figura 2. Arquitetura de Von-Neumann.

No caso dos microcontroladores PIC, o barramento de dados é na maioria das vezes de 8 bits e o de instruções pode ser de 12, 14 ou 16 bits. Esse tipo de arquitetura permite que, enquanto uma instrução é executada, outra seja buscada da memória, o que torna o processamento mais rápido. Como mostra a Figura 3, a arquitetura *Harvard* utiliza dois barramentos de endereços distintos para acessar instruções e dados.

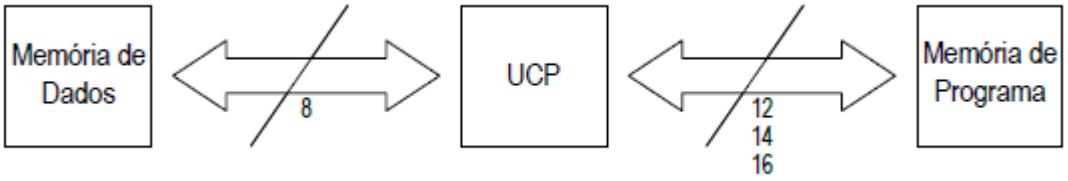

Figura 3. Arquitetura de Harvard.

2.2 Principais componentes

Como já foi mencionado, o microcontrolador possui uma série de periféricos que são necessários para o controle de qualquer processo. Os principais periféricos estão descritos a seguir.

2.2.1 Memória

Para dispositivos com a arquitetura *Harvard*, as memórias de dados e de programa usam barramentos separados. No caso do PIC, existe uma memória não volátil que pode ser programada e apagada diversas vezes, eletricamente – EEPROM (MICROCHIP, 2004).

A memória de programa é onde estão armazenadas as tarefas que o microcontrolador deve executar. Nessa os programas podem modificar configurações, manipular os dispositivos, efetuar comunicação de entrada e saída, executar instruções aritméticas, etc. Segundo SOUZA (2001), os modelos essa memória é do tipo EPROM, que só pode ser gravada uma vez para PICs normais ou várias vezes no caso de PICs janelados. Existem ainda modelos com memória do tipo FLASH, podendo ser gravada várias vezes sem a necessidade de apagar a gravação anterior.

A memória de dados do sistema é a RAM, que é utilizada para guardar todas as variáveis e registradores utilizados pelo programa. Essa memória armazena dados de 8 bits e é volátil. A memória de dados é dividida em dois grupos: registradores especiais, utilizados pelo microcontrolador para a execução do programa e processamento da ULA, e registradores de uso geral, destinados ao armazenamento de variáveis definidas pelo usuário para serem lidas e escritas pelo programa.

2.2.2 ULA

A ULA é a área de uma CPU na qual as operações lógicas e aritméticas são realizadas sobre os dados. O tipo de operação realizada é determinado pelos sinais da unidade de controle. Os dados a serem operados pela ULA podem ser oriundos de uma memória ou de uma unidade de entrada. Os resultados das operações realizadas na ULA podem ser transferidos tanto para uma memória de dados como para uma unidade de saída.

2.2.3 Interrupções

Um programa que está sendo executado pode fazer uso direto ou indireto de funções e sub-rotinas disponibilizadas no *hardware* ou no sistema operacional do sistema como um todo. Este uso é feito colocando uma diretiva específica da interrupção no código para marcar aquele ponto do programa. Assim que a interrupção ocorrer, a execução normal do programa será interrompida e o programa dará um salto para a sub-rotina localizada imediatamente abaixo da diretiva. Após o término dessa rotina, a execução do programa é retomada, passando a ser executada a primeira instrução após aquela que precedeu à interrupção. É possível

realizar a comunicação de dados entre o programa e a rotina de interrupção por meio de registradores específicos (MARTINS, 2005).

Nos microcontroladores da série PIC há a ocorrência de pelo menos quatro tipos de interrupções básicas: aquelas de *timers*, a externa, por mudança de estado e a interrupção de fim na escrita na EEPROM. As interrupções de *timers*, utilizadas neste trabalho, acontecem sempre que um registrador dos contadores de tempo interno estoura o limite de contagem (SOUZA, 2001).

2.2.4 Interfaces de entrada e saída

Para Santos (2009, p. 11), este tipo de componente é responsável por prover formas de comunicação do microcontrolador com dispositivos externos. É o meio usado para a troca de dados que podem ser transmissão serial e paralela em vários protocolos como RS232 e USB.

2.2.5 Registradores especiais

Como já foi mencionado neste capítulo, o PIC possui uma série de registradores especiais que servem exatamente para guardar a configuração e o estado de funcionamento atual da máquina. Os registradores utilizados para o desenvolvimento deste projeto são os do tipo portas e os registradores do tipo contadores.

Os registradores do tipo portas são divididos em dois grupos, registradores TRIS e registradores PORT. Os registradores TRIS servem para configurar os pinos das portas como entrada ou saída. Quando um bit do TRIS é colocado em estado lógico alto, o pino relacionado a ele é configurado como entrada. Analogamente, para configurar o pino como saída, deve-se colocar o bit em um estado lógico baixo. Quando um pino das portas disponíveis do microcontrolador é configurado como entrada, ao lermos o registrador PORT relacionado, encontra-se diretamente o nível lógico aplicado a esse pino.

O número de registradores responsáveis pela configuração dos *timers* depende da quantidade de contadores/temporizadores disponibilizados pelo microcontrolador. De um modo geral, as configurações relacionadas a cada *timer* são: habilitação do sistema de oscilação externa, habilitação das interrupções, ajuste

do *prescaler*, controle de sincronismo, seleção da origem do *clock*, habilitação do *timer*, etc.

2.3 *Bootloader*

Microcontroladores avançados das séries PIC16 e PIC18 da Microchip possuem a capacidade de escrever em seus próprios espaços de memória programável através de um controle interno por *software*. Isto é realizado por uma rotina de *bootloader* localizada no bloco de *boot* protegido na parte superior da memória programável do microcontrolador, normalmente não utilizada pelo *firmware* principal do aplicativo (Microchip, 2010).

Quando o *firmware* do *bootloader* é ativado, um PC *host* pode usar uma comunicação serial para ler escrever e verificar atualizações do *firmware* da aplicação do microcontrolador. Uma vez que o *firmware* do aplicativo é programado, o *bootloader* cede o controle, permitindo a execução normal até que o *bootloader* seja chamado novamente.

2.3.1 Considerações de *Hardware*

A Figura 4 ilustra um circuito básico de *bootloader* USB, onde a chave JP1 controla a entrada de *bootloader*. Se JP1 é fechada e o PIC for reiniciado, o dispositivo irá entrar em modo *bootloader*. Caso a chave JP1 feche depois que o PIC entrar em *reset*, o dispositivo não entrará no modo *bootloader,* e sim no modo de aplicação.

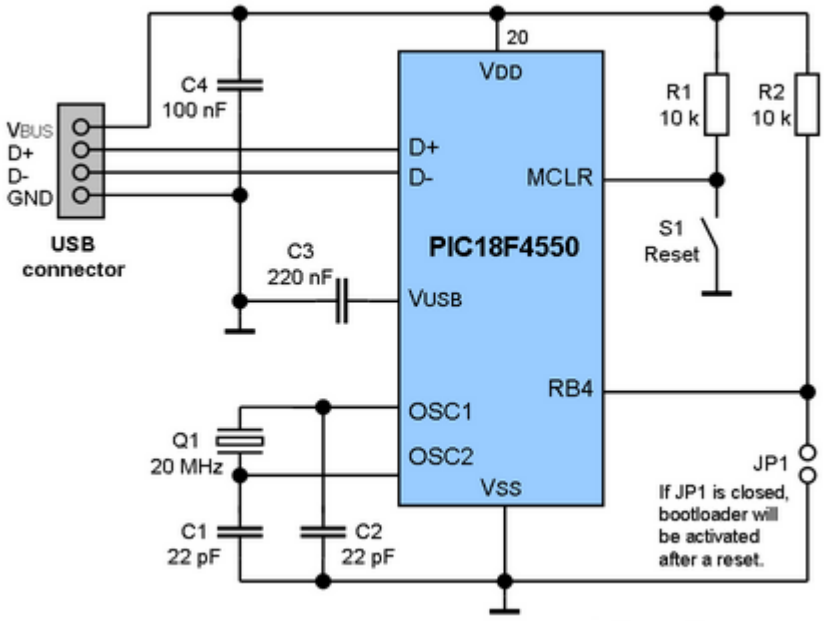

Figura 4. Circuito *Bootloader*.

2.3.2 Considerações de *Firmware*

Existem dois modos de operação: *firmware bootloader* e aplicação. Quando o microcontrolador sai do modo *reset*, a rotina inicial do *bootloader* decide se entra no ciclo de comando *bootloader* (modo *Bootloader*) ou vai para o vetor de ponto de entrada da aplicação (modo de aplicação).

Para que o dispositivo entre em modo *bootloader*, basta que o microcontrolador esteja sem nenhum *firmware* programado ou se o pino RB4 do dispositivo estiver com o estado lógico baixo quando o microcontrolador for reiniciado. Se nenhuma dessas condições forem satisfeitas, o dispositivo entrará em modo de aplicação.

2.3.3 Modo *bootloader*

De acordo com a Microchip (2010), na configuração padrão, o *bootloader* é armazenado no espaço final da memória flash. Manter o *bootloader* neste espaço de memória permite que o *firmware* da aplicação lide com as interrupções de *hardware* no vetor de endereço normal.

A aplicação responsável por gravar o programa no microcontrolador irá escrever um *GOTO* como a primeira instrução no vetor *Bootloader Reset Vector*. Esse vetor apontará para o início do código do *bootloader*. O vetor de *reset* original do *firmware* da aplicação é movido automaticamente para residir pouco antes do bloco de memória do *bootloader*. O *bootloader* usa esse vetor de *reset* para iniciar a aplicação quando o *bootloader* não for solicitado (ver Figura 5).

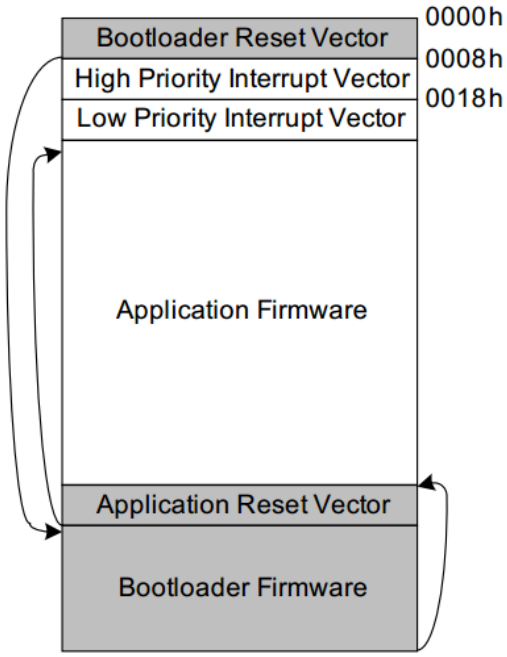

Figura 5. Memória de programa no modo *bootloader*.

Capítulo 3

Universal Serial Bus (USB)

O protocolo USB foi concebido com a finalidade de poder ser implementado em qualquer dispositivo, possuir configuração fácil por parte do usuário, ter um baixo custo de desenvolvimento, ser *Plug and Play* e ter disponível vasta documentação e suporte. A velocidade da transmissão de dados também foi algo em foco no desenvolvimento desse padrão (SANTOS, 2009).

3.1 Organização em camadas

Segundo (MICROCHIP, 2004), A funcionalidade de um dispositivo USB está estruturada em uma organização baseada em camadas, como ilustra a Figura 6.

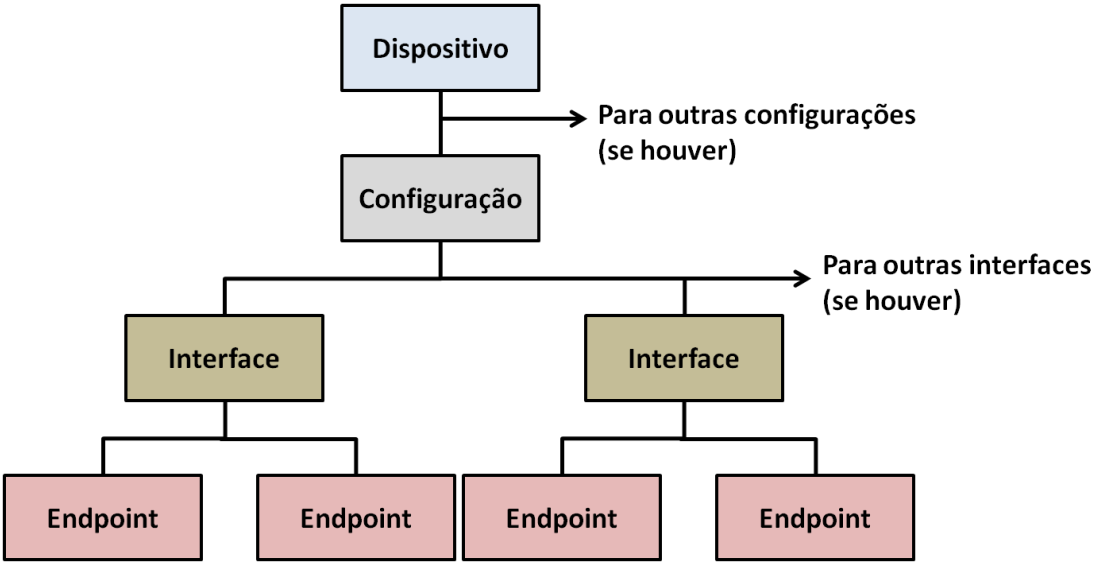

Figura 6. Organização USB em camadas.

A camada mais elevada, abaixo do dispositivo é a de configuração. Um dispositivo pode ter várias configurações, e para cada uma delas pode haver múltiplas interfaces. Cada interface pode suportar um determinado modo dessa configuração. A última camada é o *Endpoint*, os dados são movidos diretamente a

este nível. Por definição, o *Endpoint 0* é sempre responsável pelo controle e, por padrão, quando o dispositivo está ligado, o *Endpoint 0* deve estar disponível para configurar o dispositivo.

3.2 Modos de transferência

Segundo Ibrahim (2008), os dados podem ser transferidos pela USB de quatro maneiras distintas: transferência em massa, interrupção, controle e isossíncrona.

3.2.1 Transferência em massa

São projetadas para transferir grandes quantidades de dados com entrega livre de erros. Em geral as transferências em massa são usadas onde uma baixa taxa de transferência não é um problema. O tamanho máximo do pacote em uma transferência desse tipo é de 8 a 64 bytes em velocidade máxima e 512 pacotes em alta velocidade.

3.2.2 Transferência de interrupção

São usadas para transferir pequenos volumes de dados, onde estes dados têm de ser transferidos o mais rapidamente possíveis, sem atraso. Pacotes de interrupção pode variar em tamanhos de 1 a 8 bytes a baixa velocidade, de 1 a 64 bytes em velocidade máxima e até 1024 bytes em alta velocidade.

3.2.3 Transferência isossíncrona

Este tipo fornece um método de transferência de grandes quantidades de dados (até 1023 bytes) com entrega oportuna garantida, no entanto, a integridade dos dados não é assegurada. Isso é bom para aplicações de *streaming*, onde a perda de dados de pequeno porte não é crítica, como áudio, por exemplo.

3.2.4 Transferência de controle

São utilizadas geralmente para a configuração inicial de um dispositivo *host*. O tamanho máximo do pacote é de 8 bytes em baixa velocidade, 8 a 64 bytes na

velocidade máxima e 64 bytes em alta velocidade.

3.3 Enumeração

Quando o dispositivo é inicialmente conectado ao barramento, o *host* entra em um processo de enumeração, na tentativa de identificar o dispositivo. Essencialmente, o hospedeiro interroga o dispositivo, coleta informações, tais como o consumo de energia, as taxas e tamanho dos dados, protocolo e outras informações descritivas (MICROCHIP, 2004). O processo de enumeração típico seria o seguinte:

- **USB *Reset*:** Reinicializa o dispositivo, pois não está configurado e não possui um endereço.
- ***Get Device Descriptor*:** O *host* solicita uma pequena porção do descritor do dispositivo.
- **USB *Reset*:** Reinicializa o dispositivo novamente.
- ***Set Address*:** O *host* atribui um endereço para o dispositivo.
- ***Get Device Descriptor*:** O *host* recupera o descritor do dispositivo, reunindo informações como fabricante, tipo de dispositivo, tamanho máximo do pacote, etc.
- ***Get Configuration Descriptors*:** o *host* solicita para receber dados de configuração do dispositivo, tais como requisitos de energia e os tipos e número de interfaces suportadas.
- ***Get any Other Descriptors*:** O *host* solicita ao dispositivo algum descritor adicional.
- ***Set a configuration*:** O *host* atribui um valor de configuração para o dispositivo. Assim, todos os *endpoints* nesta configuração irão assumir as mesmas características descritas.

3.4 Descritores

Todos os dispositivos USB têm uma hierarquia de descritores que descrevem várias características do dispositivo: a identificação do fabricante, a versão do dispositivo, a versão do USB suportado por elas, o que o dispositivo é, suas

necessidades de energia, o número e tipo de parâmetros, e assim por diante. De acordo com (MICROCHIP, 2004) e (IBRAHIM, 2008), os descritores mais importantes são descritos a seguir.

3.4.1 Descritores de dispositivo

Esses descritores fornecem informações gerais, tais como fabricante, número do produto, número de série, a classe de dispositivo e o número de configurações. Há apenas um descritor de dispositivo.

3.4.2 Descritores de configuração

O descritor de configuração fornece informações sobre requisitos de energia do dispositivo e quantas interfaces diferentes são suportadas nessa configuração. Pode haver mais de uma configuração por dispositivo.

3.4.3 Descritores de interface

Esse descritor detalha o número de *endpoints* utilizado nesta interface, bem como a classe de interface. Pode haver mais do que uma interface para cada configuração.

3.4.4 Descritores de HID

Um descritor de HID segue sempre um descritor de interface, quando a interface pertence à classe HID.

3.4.5 Descritores de *endpoint*

Esse tipo de descritor identifica o tipo de transferência e direção, bem como alguns outros detalhes do *endpoint*. Pode haver muitos *endpoints* e compartilhados em diferentes configurações.

3.5 Classes de dispositivos

Os dispositivos baseados em comunicação USB necessitam de especificações que informam a que tipo de classe de dispositivos eles pertencem. Abaixo estão listados os tipos de classes mais comuns.

3.5.1 Human Interface Device (HID)

Os dispositivos que fazem parte desta classe USB geralmente ajudam os operadores a se comunicar de maneira fácil com os computadores. Possuem instalação simples, pois já estão com os protocolos de comunicação definidos.

3.5.2 Mass Storage

Os dispositivos que se enquadram nesta classe são normalmente equipamentos que trabalham com transferência de um grande volume de dados, como discos rígidos.

3.5.3 Communications Device Class (CDC)

Os dispositivos do tipo de classe definido como CDC implementam um mecanismo de comunicação de propósito geral que pode ser usado para a comunicação entre a maioria dos dispositivos.

Na maioria dos casos, é exigido um *driver* para que o *host* possa se comunicar com o dispositivo USB. Em aplicações personalizadas, pode haver a necessidade de desenvolver *drivers* específicos para atender aos requisitos do sistema. Felizmente, existem *drivers* disponíveis que atendem às mais diferentes classes de dispositivos, podendo ser reutilizados.

A fim de evitar a instalação de *drivers* para o devido funcionamento da comunicação USB, o dispositivo desenvolvido neste projeto irá funcionar como um dispositivo de interface humana (HID). O padrão HID engloba e inclui muitos drivers de dispositivos genéricos para que ao ligar o dispositivo garanta sua funcionalidade básica.

Capítulo 4

Desenvolvimento do projeto

O desenvolvimento do dispositivo apresentado neste trabalho pode ser dividido em três partes, são elas: projeto de *hardware*, projeto de *firmware* e projeto de *software*.

4.1 Projeto de *Hardware*

O *hardware* desenvolvido neste projeto foi dividido em partes para facilitar o desenvolvimento e a implantação do sistema para o propósito real. O projeto é constituído de três partes, são elas: uma placa microcontrolada que ficará responsável por receber comandos de um computador e transmiti-los para cada uma das placas de interação com o atleta, obedecendo todas as configurações definidas no *software* pelo usuário; uma placa de conexões que irá realizar as ligações entre a placa controladora e todas as placas de interação; vinte e quatro placas utilizadas para a interação com o atleta. A Figura 7 ilustra o diagrama de blocos de todo o *hardware* desenvolvido no projeto. No decorrer do capítulo serão apresentadas cada uma dessas divisões do *hardware* do projeto.

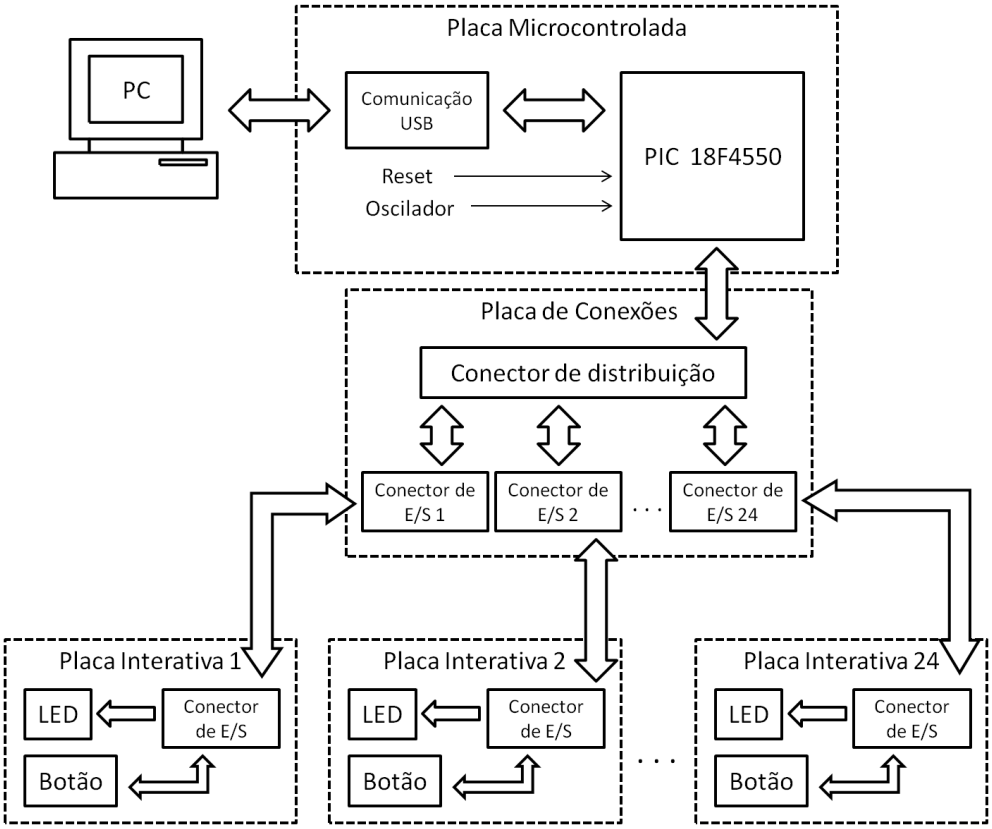

Figura 7. Diagrama de blocos do *hardware*.

4.1.1 Placa microcontrolada

O funcionamento de todas as partes da aplicação dependerá de uma placa principal, responsável por enviar e receber comandos de um computador para que o treinamento seja realizado. Esses comandos serão enviados via USB por um sistema de interface com o usuário, indicando qual placa de interação irá estimular o atleta. O circuito da placa controladora é dividido em seis partes, são elas: o microcontrolador PIC18F4550, o circuito oscilador, a gravação com o *bootloader*, a comunicação USB, o circuito de re-inicialização e o conector de entrada e saída como ilustra a Figura 8.

O microcontrolador estudado e utilizado neste projeto é o PIC18F4550 (Microchip, 2004) que possui todas as vantagens da família PIC18 (alto desempenho computacional a um preço acessível) com adição de um material de alta resistência e programação avançada em memória flash. O PIC18F4550 apresenta melhorias no projeto que tornam esses microcontroladores uma ótima escolha para muitas

aplicações de alto desempenho. O PIC utilizado apresenta características importantes que o tornaram o microcontrolador escolhido para este projeto, são elas:

Durabilidade de memória:

As células de memória de programação e de dados são avaliadas para durar por milhares de ciclos de apagamento e reescrita de valores. De acordo com (MICROCHIP, 2004), o tempo de retenção de dados sem atualização no chip é estimado em mais de 40 anos.

Auto-programação:

Esse dispositivo pode escrever em seus próprios espaços de memória programável através de um controle interno por *software*. Usando uma rotina de *bootloader*, localizada no bloco de *boot* protegido na parte superior da memória de programa, é possível atualizar a aplicação em campo, sem a utilização de um gravador.

Universal Serial Bus (USB):

O PIC18F4550 possui um USB SIE (*Serial Interface Engine*) compatível com *full-speed* e *low-speed* USB o que possibilita a rápida comunicação entre um *host* USB e o microcontrolador. Além disso, o módulo USB possui 16 *endpoints* bidirecionais e suporta quatro tipos de transferências: controle, interrupção, isossíncrona e massa.

Portas bidirecionais de entrada e saída:

Dependendo dos recursos habilitados, existem até cinco portas disponíveis. Nas portas bidirecionais, os sinais são lidos ou escritos via de registradores da porta. A direção dos sinais é controlada pelo registrador TRIS. Existe um registrador TRIS para cada porta, com os valores *um* para leitura e *zero* para escrita (BOLTON, 2010).

Timers:

Segundo (MICROCHIP, 2004), existem quatro *timers* disponibilizados pelo PIC18F4550, são eles TMR0, TMR1, TMR2 e TMR3. Os *timers* 0, 1 e 3 são temporizadores e contadores de 16 bits, enquanto que o timer 2 é um contador de 8 bits. Para todos os quatro *timers* do 18F4550, o estouro da contagem pode gerar

uma interrupção. Utilizando a função *prescaler*, divisor de frequência do *clock* do timer, pode se obter tempos maiores quando o *timer* estiver configurado no modo temporizador.

Frequência de operação até 48Mhz:

Uma das formas de se obter as taxas de comunicação USB é utilizando um PLL com um divisor. Dividir por PLL significa multiplicar a frequência de entrada do oscilador utilizado. Por exemplo: usando um cristal de 12MHz e a PLL com divisor de 4, atinge-se a frequência interna de 48Mhz.

Oscilador:

O oscilador ou *clock* do microcontrolador pode ser configurado de quatro maneiras distintas, dependendo do uso. Esta configuração é feita via *software* e é aceita pelo microcontrolador durante sua gravação. O *clock* determinará a velocidade de operação do microcontrolador. O PIC18F4550 pode operar em doze modos de oscilação possíveis, porém os mais utilizados são os modos XT e HS (MICROCHIP, 2004).

Nos modos de oscilação XT e HS, um cristal é conectado aos pinos OSC1 e OSC2 do PIC e dois capacitores para estabelecer a oscilação do circuito, como ilustra a Figura 8. A Tabela 1 mostra quais frequências cada modo de oscilação permite e os capacitores recomendados.

Tabela 1. Relação modo/frequências/capacitores.

Modo	Frequência	Capacitores recomendados
XT	2MHz – 4MHz	15pF – 33pF
HS	4Mhz – 20MHz	15pF – 33pF

Para este projeto foi utilizado o modo HS, por suportar cristais de frequências mais altas. Neste circuito é utilizado um cristal de 20 MHz e dois capacitores de 22 pF.

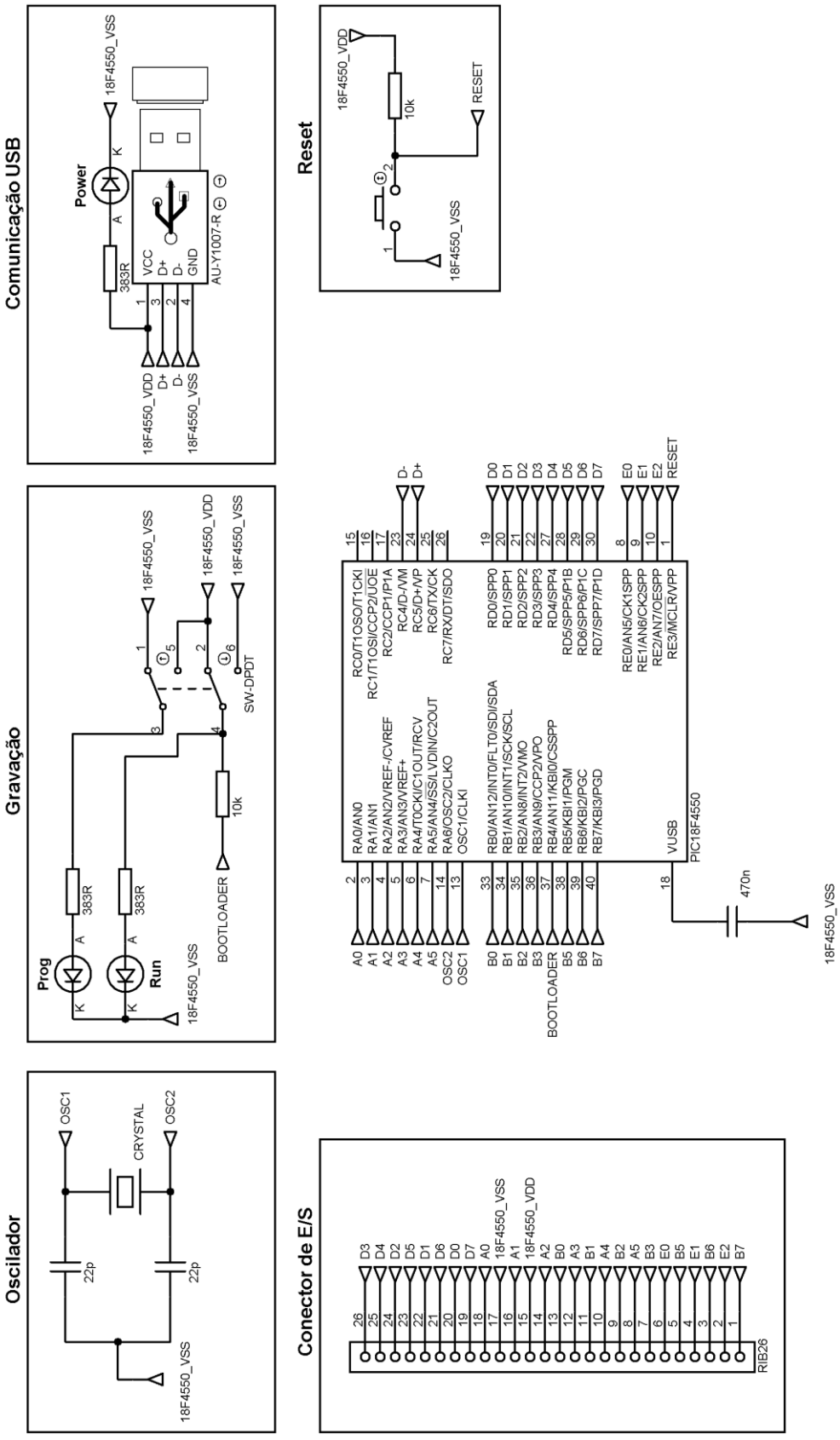

Figura 8. Circuito da placa microcontrolada.

Gravação:

O *bootloader* é um *firmware* que quando instalado no microcontrolador, permite a gravação de programas diretamente através de uma porta USB. Para a gravação do *bootloader* será necessário a utilização de um gravador convencional uma única vez. O circuito de gravação com o *bootloader* possui uma chave DPDT para realizar a seleção entre o modo de gravação e execução. Quando ativado o modo de gravação (o LED “Prog” acenderá), será enviado um sinal alto para o pino RB4 indicando que o PIC entrou em modo *boot.* Da mesma forma, quando ativado o modo execução (o LED “Run” acenderá), será enviado um sinal baixo para o pino RB4 desabilitando o modo boot do PIC. Como já foi mencionado no Capítulo 2, assim que selecionado o modo *bootloader* basta acionar a chave de *reset* do circuito para iniciar a gravação.

Comunicação USB:

O circuito de comunicação via USB é de fundamental importância para o projeto, pois com ele será possível realizar o treinamento, já que é através da USB que ocorre a comunicação entre o sistema de interface com o usuário e a placa controladora. Como mostrado anteriormente, a comunicação USB será essencial também para a atualização do *firmware* na própria placa, sem a necessidade de remoção do microcontrolador.

***Reset*:**

O PIC possui internamente circuitos que controlam o *Power-on reset*. Basicamente isso significa que o microcontrolador necessita de poucos componentes externos para realizar o *start-up.* O circuito de *reset* utiliza um resistor, ligado entre o pino de MCLR e o VCC, no valor de 10k *ohms* e uma chave para realizar o reset manual, ligada entre o MCLR e o terra.

Conector de E/S:

Para realizar a transmissão e recepção dos dados, foi utilizado um conector *header* de 26 vias para cabos *flat*. Serão utilizadas 24 vias para a transmissão de sinais dos pinos do microcontrolador, uma via para o VCC e uma para o terra. Esse conector será utilizado para a comunicação da placa microcontrolada e a placa de conexões. A Figura 9 mostra a placa construída.

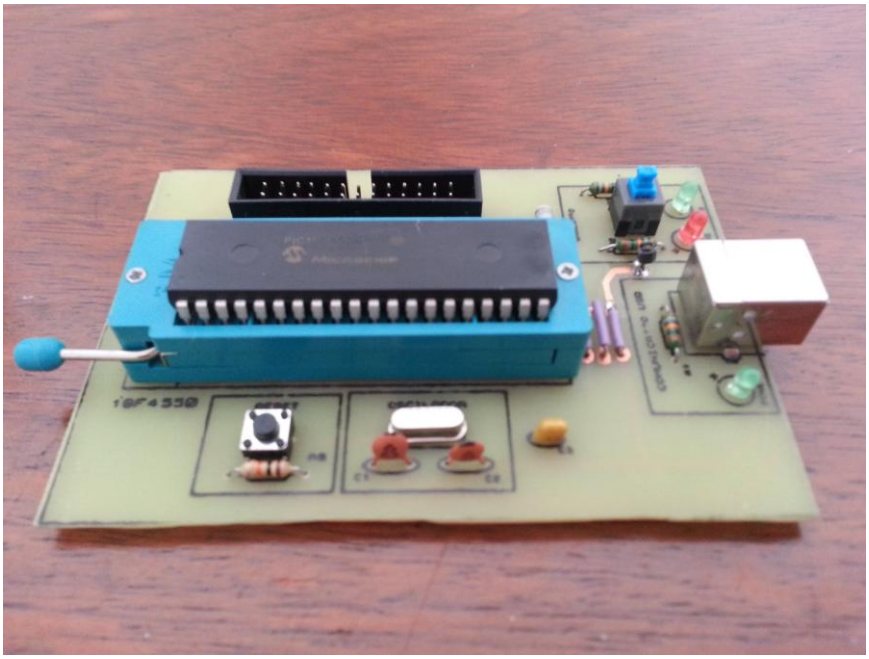

Figura 9. *Hardware* da placa microcontrolada.

4.1.2 Placa de conexões de E/S

Assim como mostra o diagrama da Figura 7, os componentes que fazem essa placa são apenas conectores responsáveis por toda a distribuição do sistema. A placa de conexões de E/S foi adicionada ao projeto para tirar essa responsabilidade da placa microcontrolada e deixar todo o sistema mais simples e fácil de montar. Esta placa possui um conector central de distribuição e 24 conectores para transmitir os sinais para cada uma das placas de interação. Para o conector de distribuição foi usado um conector *header* de 26 vias para cabos *flat*, e para cada conector de E/S foi utilizado conectores KK de 3 vias com os sinais do pino, VCC e terra. A Figura 10 mostra a placa construída.

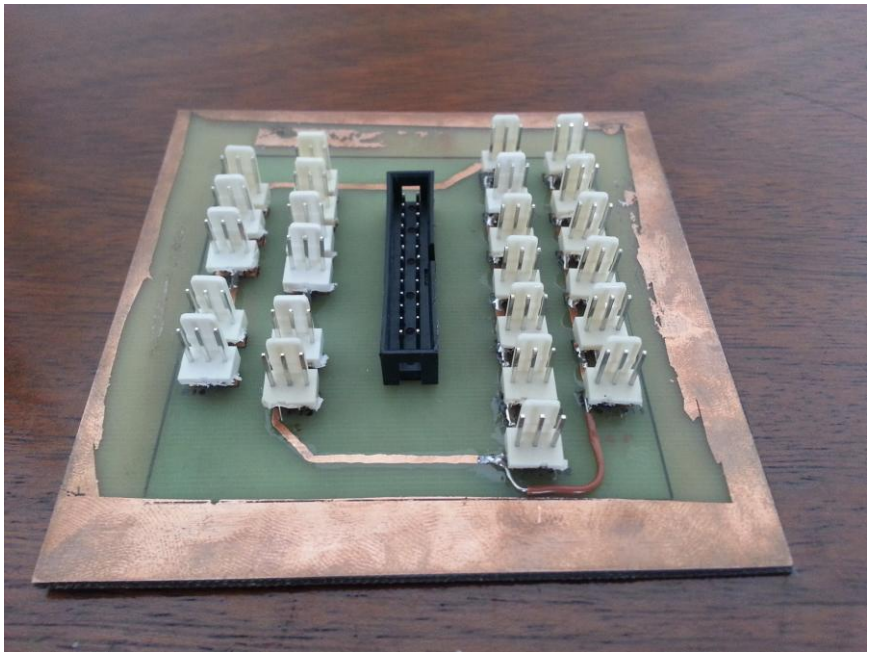

Figura 10. *Hardware* da placa de conexões.

4.1.3 Placa interativa

A principal função da placa interativa é acender um LED para que o atleta seja estimulado visualmente e receber o sinal lógico de um botão que representará a resposta ao estímulo provocado. Para facilitar o treinamento e a visão do atleta, foram colocados três LEDs como ilustra a Figura 11.

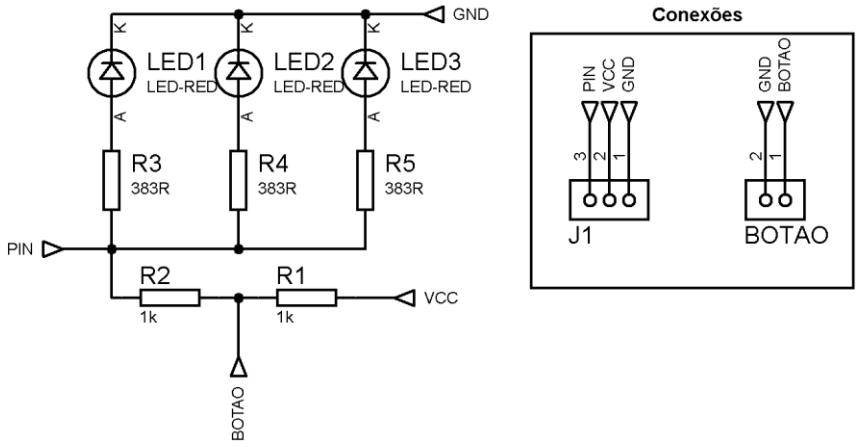

Figura 11. Circuito da placa interativa.

Todas as placas de interação ficarão fixadas em um painel de algum material resistente. Para evitar que a placa quebre com o impacto, o botão será também fixado no painel. Para representar o botão na placa, foi colocado um conector de duas vias para passar os sinais para o botão do painel.

Como o número de portas do microcontrolador é limitado e o número de nós utilizados no projeto é grande, foi necessária a utilização de pinos bidirecionais de E/S do PIC. Dependendo dos recursos habilitados, existem até cinco portas disponíveis. Nas portas bidirecionais, os sinais são lidos ou escritos via registradores da porta. A direção dos sinais é controlada pelo registrador TRIS. Existe um registrador TRIS para cada porta, com os valores um para leitura e zero para escrita (BOLTON, 2010). Como mostra o circuito da Figura 11, os três LEDs estão ligados ao pino do microcontrolador juntamente com o botão. A Figura 12 mostra a placa construída.

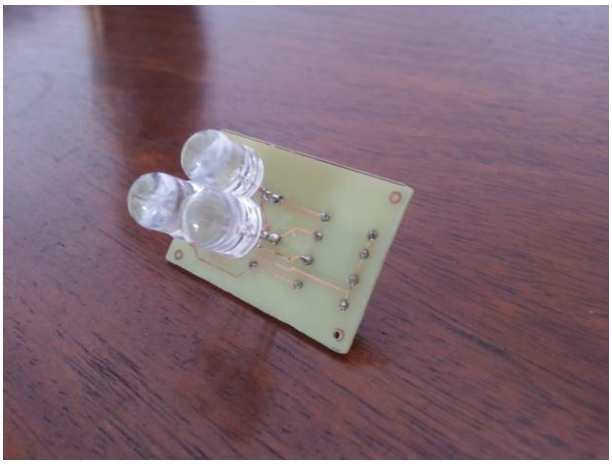

Figura 12. *Hardware* da placa de interação.

4.2 Projeto de *Firmware*

Para LIBERALQUINO (2010), o núcleo de sistema embarcado é um microcontrolador, que possui a responsabilidade de gerenciar quando um comando é acionado, ou quais parâmetros serão passados para os periféricos ligados as suas unidades de entrada e saída, quais unidades serão acionadas.

Também conhecido pela nomenclatura *software embarcado*, o *firmware* é um conjunto de instruções operacionais que são programadas diretamente no *hardware* de equipamentos eletrônicos. Esse sistema ficará gravado na memória programável do microcontrolador e compõe um conjunto de funcionalidades de um determinado módulo.

Esta seção irá apresentar todo o desenvolvimento do *firmware* do projeto, mostrando primeiramente as configurações do microcontrolador realizadas inicialmente, e a explicação detalhada de todo o fluxo do programa principal.

4.2.1 Configurações iniciais

O projeto do código fonte é escrito na linguagem C, segundo a especificação do compilador PICC da empresa CCS (*Custom Computer Services*). Este compilador não é *open source*, mas possui uma biblioteca extensa que reduz a complexidade do desenvolvimento do *firmware*.

Segundo PEREIRA (2003), todo compilador possui uma lista de comandos internos que não são diretamente traduzidos em código, esses comandos são utilizados para especificar determinados parâmetros internos utilizados pelo compilador e são chamados de diretivas do compilador. Essas diretivas servem para configurar informações como módulos utilizados entre as unidades de entrada e saída, configurações de *clock* e *timer*, proteção de código, opções de *reset*, etc. Sendo assim, a primeira etapa para o desenvolvimento do *software* embarcado foi a definição das diretivas de compilação iniciais. Algumas diretivas podem ser observadas na Tabela 2.

Tabela 2. Diretivas de compilação.

Diretivas	Descrição
#include	Incluir arquivos de códigos fonte.
#define	Definição de constantes de cadeia de caracteres.
#fuses	Define os bits de configuração.
#use delay	Informa ao compilador o valor da frequência em Hz.
#byte	Permite acesso a endereços de registradores.

A diretiva *#include* indica arquivos com códigos de programação extras. Esses arquivos de código fonte são adicionados ao programa principal no momento da compilação. No *firmware* desenvolvido foram utilizados os seguintes arquivos:

```
#include <18F4550.h>
#include <pic18_usb.h>
#include <usb.c>
#include <usb_desc_hid.h>
#include <fuses_config.h>
#include <trismap.h>
```

O arquivo *18F4550.h*, fornecido pelo compilador, importa funções específicas do PIC 18F4550 tais como manipulação de temporizadores, entrada e saída de dados, etc. Os arquivos *pic18_usb.h* e *usb.c*, também fornecidos pelo fabricante do compilador, importam funções necessárias para que a comunicação USB funcione corretamente. O *usb_desc_hid.h* contém a implementação dos descritores USB, esse arquivo contém informações específicas como os identificadores de produto e fabricante do dispositivo, necessários para a identificação deste no sistema de interface com o usuário.

Os dois últimos arquivos foram criados neste projeto para facilitar o desenvolvimento e entendimento do código. Na biblioteca *fuses_config.h* são utilizadas diretivas *#fuses* para informar os bits de configuração específicos do microcontrolador PIC18F4550, a Tabela 3 mostra alguns dos bits definidos no projeto.

Tabela 3. Bits de configuração.

Bits	**Função**
HSPLL	Define o modo do oscilador para HS com PLL habilitado.
MCLR	Habilita o pino *Master Clear* do PIC.
PLL5	Frequência do *clock* pré escalar é dividida por 5.
CPUDIV1	Uso de PLL pós escalar com valor 1.
USBDIV	Configura o *clock* do dispositivo USB integrado ao PIC.
NOWDT	Desabilita a função do *Watch Dog Timer*.

VREGEN	Ativação do uso do Vusb como regulador de tensão da USB.

Na biblioteca *trismap.h* são utilizadas diretivas de #*byte* e #*bit* para definir o endereço dos registradores TRIS de cada porta, assim como rotular os endereços dos bits para facilitar a manipulação e o controle de cada bit do registrador TRIS.

A diretiva #*use delay* informa ao compilador qual o *clock* de operação da CPU do microcontrolador e os valores são informados em Hz. A operação deste sistema é baseada em uma frequência de 20 MHz.

4.2.2 Fluxograma

A Figura 13 ilustra o fluxograma do programa principal que é baseado em realizar uma comunicação USB com o sistema externo, podendo receber e enviar comandos para que o treinamento seja realizado. Para que todas as interrupções do PIC possam ser manipuladas, inicialmente o registrador da interrupção global deve ser habilitado conforme o código abaixo:

```
enable_interrupts(global);
```

Dentre os temporizadores disponibilizados pelo microcontrolador, será utilizado o *timer 1* para realizar a contagem do tempo de resposta do atleta. O *timer1* é um temporizador/contador de 16 bits, e caso sua interrupção seja habilitada é gerado um *overflow*. Portanto o tempo de resposta será calculado a partir da quantidade de *overflows* gerados pelo *timer 1.* Para a inicialização do *timer 1* são executados os códigos abaixo:

```
Setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);
Set_timer1(0);
```

Na primeira função são utilizados dois parâmetros de configuração, o "*T1_INTERNAL*" indica que o *timer* será incrementado pelo *clock* interno, e o "*T1_DIV_BY_8*" indica que a frequência do *timer* será dividida por 8. Já a segunda função força o *timer 1* a iniciar com o valor zero. Ainda na fase inicial do programa, é habilitada a comunicação USB e os registradores TRIS das portas são configurados inicialmente como saída, conforme o código abaixo:

usb_init();

usb_task();

usb_wait_for_enumeration();

set_tris_a(0x00);
set_tris_b(0x00);
set_tris_d(0x00);
set_tris_e(0x00);

Conforme ilustra o fluxograma, o programa irá permanecer em um *loop* infinito enquanto a placa microcontrolada estiver ligada. Após a comunicação USB ser estabelecida corretamente, deverão ser realizadas duas verificações a cada iteração: verificação de recebimento de dados via USB e verificação de permissão para leitura do botão.

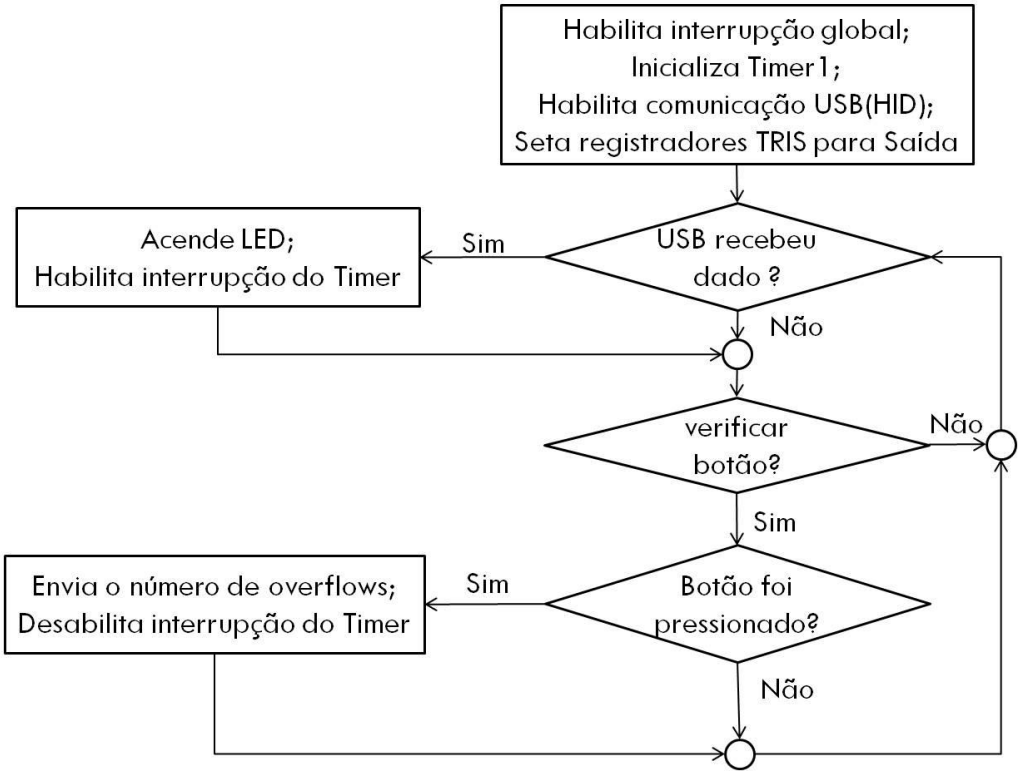

Figura 13. Fluxograma do *firmware*.

Recebimento de dados via USB:

A cada iteração será verificado se foi recebido algum pacote vindo da USB através da função *usb_kbhit(1)*. Em caso positivo, o pacote é recebido e tratado para cada tipo de comando definido, como mostra a Tabela 4.

Tabela 4. Comandos do treinamento

Comando	Função
0	O treinamento deverá ser interrompido.
1 - 24	Ligar LED referente ao número.

Se o comando for *0*, o treinamento será finalizado e todos os LEDs serão desligados. Porém, se o comando for entre *1* e *24***,** os registradores TRIS de todas as portas serão configuradas como saída e o LED referente ao número do comando será ligado. Assim que o LED acender, será habilitada a interrupção do *timer* 1 para que a contagem seja iniciada.

Permissão para leitura do botão:

Já que os pinos do microcontrolador são utilizados tanto para entrada (leitura do botão) como saída (acender LED), o tempo em que cada ação é realizada deve ser dividido. Sendo assim, foi criado um contador que será incrementado a cada iteração para verificar se a leitura do estado lógico do botão pode ser realizada. Ficou definido que para 99% do tempo os registradores TRIS estarão configurados para *saída* e somente 1% para *entrada*.

Esse cuidado foi levado em consideração para que a mudança de um estado para outro fosse imperceptível para o usuário, já que o LED ligado inicialmente deverá ser desligado para que o estado do pino seja conferido, como pode ser visto no código exemplo a seguir:

```
TRIS_B0 = 1;

if(input(pin_b0) == 0) retorno = true;

TRIS_B0 = 0;

retorno ? output_low(pin_b0) : output_high(pin_b0);
```

Na primeira linha o pino RB0 recebe valor lógico alto indicando que será utilizado como pino de leitura. Na linha seguinte é verificado o estado lógico do pino RB0, testando se o botão foi ou não pressionado. Em caso positivo e retornado o booleano *true*. Logo em seguida o pino volta a ser utilizado como escrita e o LED é aceso novamente caso a variável retorno seja verdadeira.

Caso o botão seja realmente pressionado, o LED será apagado, será enviado um pacote via USB contendo o número de overflows contados pelo *timer 1*, a interrupção será desabilitada e o *timer* reiniciado.

4.3 Projeto de *Software*

Esta seção apresenta todo o projeto e desenvolvimento do sistema de interface com usuário proposto para este trabalho. A ideia foi criar um *software* capaz de facilitar o uso do treinamento para o usuário, promovendo uma maior interação entre todos os módulos do projeto.

O termo interface é aplicado normalmente àquilo que interliga dois sistemas. Tradicionalmente, considera-se que uma interface homem-máquina é a parte de um artefato que permite a um usuário controlar e avaliar o funcionamento do mesmo através de dispositivos sensíveis às suas ações e capazes de estimular sua percepção. No processo de interação usuário-sistema a interface é o combinado de *software* e *hardware* necessário para viabilizar e facilitar os processos de comunicação entre o usuário e a aplicação (NORMAN, 1986).

Para que o usuário possa utilizar o sistema com sucesso ele deve saber quais as funções da aplicação são oferecidas pelo sistema e como ele pode interagir com cada uma delas. Então este capítulo tem como principal função mostrar como o usuário pode efetivamente interagir com o sistema e o que esse sistema poderá fazer, isto é, quais as funções oferecidas e uma explicação através de um fluxograma, mostrando passo a passo o funcionamento do *software*.

4.3.1 Funcionalidades do sistema

O *software* implementado tem o objetivo de ajudar o usuário a acompanhar o treinamento em tempo real, e todas as interações do atleta com o sistema. Será possível a visualização dos tempos de resposta em tela, criação de sequências de

treinamento, possibilidade de geração de relatórios, etc. Para o desenvolvimento foi utilizada a linguagem C# e implementado na IDE Visual Studio 2008, por possuir boas bibliotecas para comunicação USB disponibilizadas na internet.

A tela principal do sistema é dividida em três partes mais gerais, são elas: a barra de ferramentas, o painel de treinamento e a barra de status como ilustra a Figura 14.

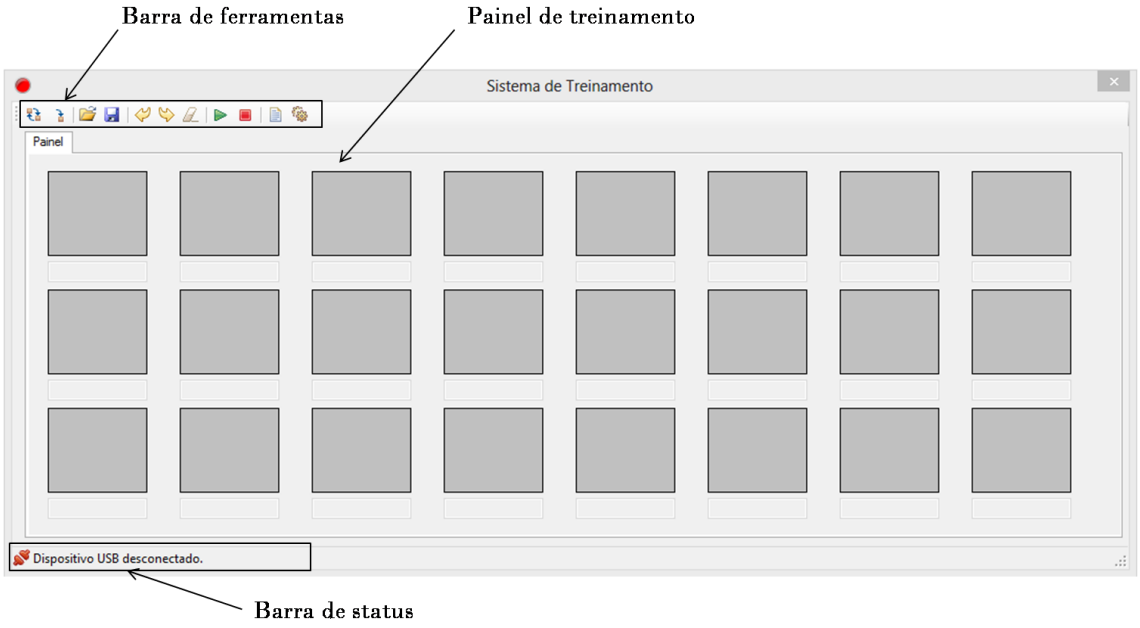

Figura 14. Tela principal do *software*.

Barra de ferramentas:

A barra de ferramentas tem a finalidade de permitir uma ação rápida por parte do usuário, facilitando o acesso a todas as funcionalidades do sistema. A Figura 15 mostra detalhadamente a função de cada botão da barra de ferramentas.

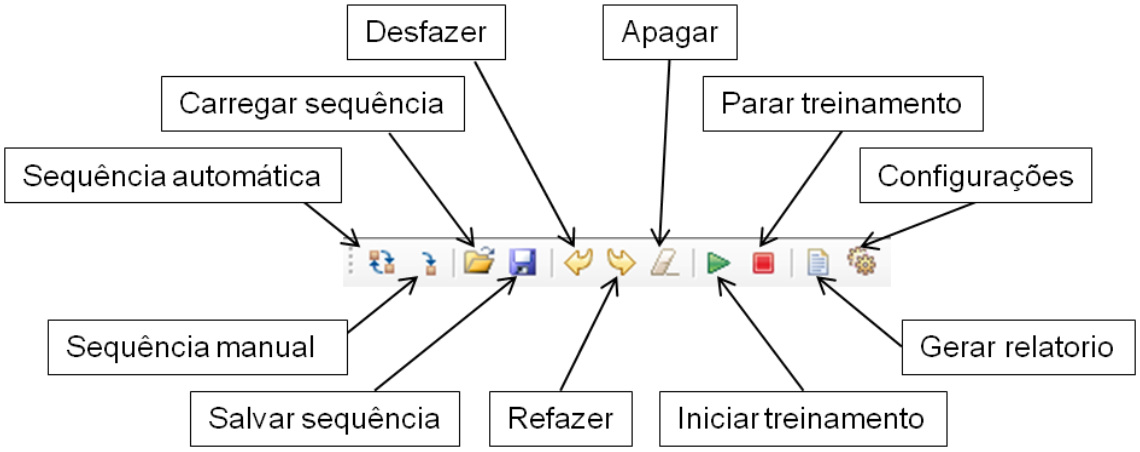

Figura 15. Barra de ferramentas.

As duas primeiras funções presentes na barra são responsáveis pela geração de sequências de treinamento de modo automático por parte do sistema ou de modo manual por escolha do usuário. No modo de geração automática o sistema cria uma sequência aleatória de números de 1 a 24, preenchendo todos os LEDs representados no painel como mostra a Figura 16. Na geração de sequência manual, o próprio usuário irá criar sua própria sequência, escolhendo a ordem de acendimento dos LEDs no painel (Figura 17). A sequência escolhida poderá conter até 96 números, já que por definição do projeto, cada LED poderá ser aceso novamente até 3 vezes.

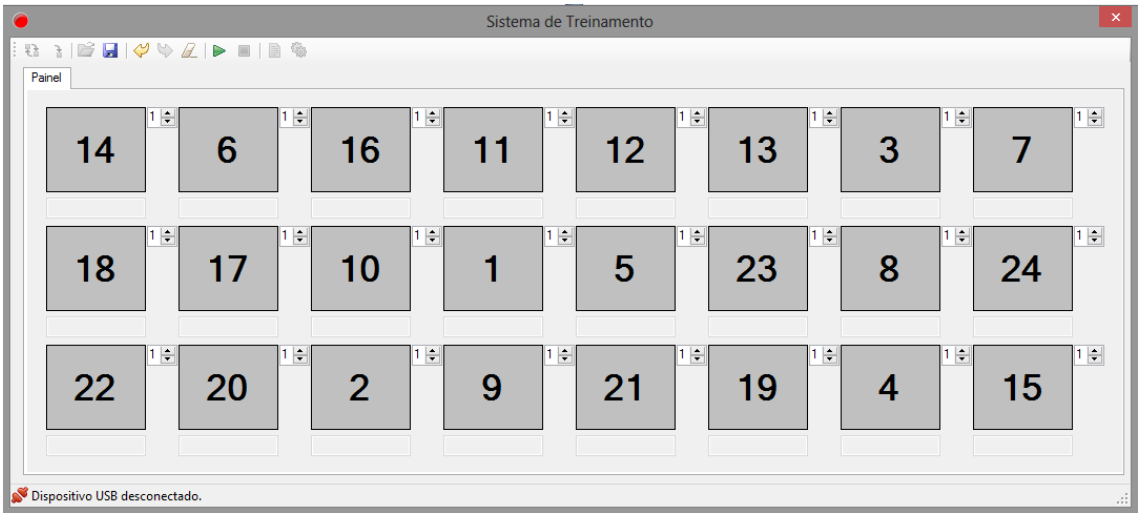

Figura 16. Geração automática de sequência.

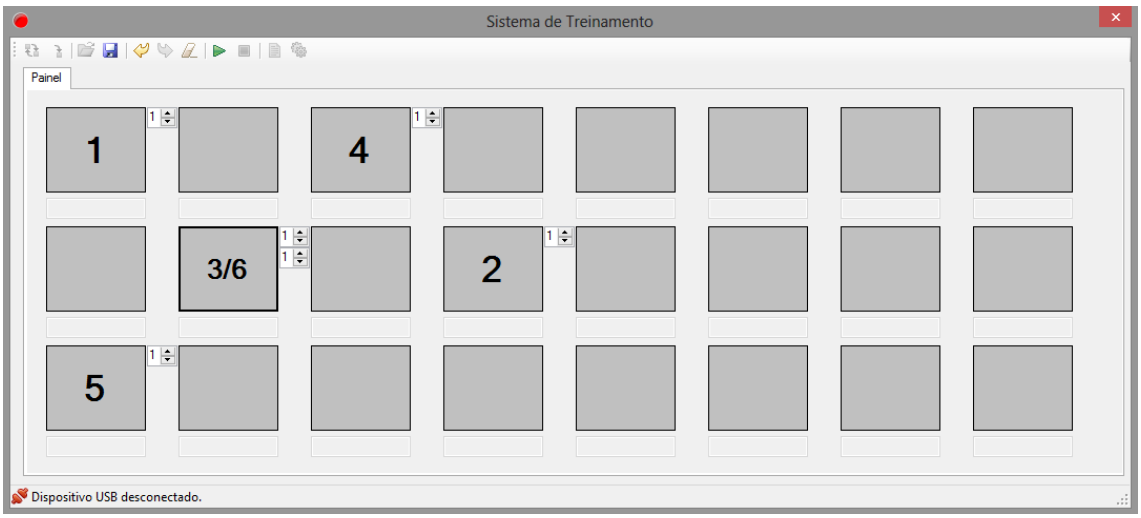

Figura 17. Geração manual de sequência.

Outra funcionalidade importante do *software* é a capacidade de carregamento e salvamento das sequências criadas. Poder reusar as sequências permite que o usuário realize o mesmo treinamento com outros atletas, ou usar a mesma sequência para averiguar se o tempo de resposta daquele atleta realmente diminuiu ou não desde o primeiro treinamento.

Para facilitar ainda mais o uso do sistema proposto, foi implementado uma estrutura de *undo/redo*, ou seja, toda ação executada sobre o painel de treinamento como criar sequência e escolher o tempo de espera de acendimento de cada LED, poderá ser desfeita ou refeita. Na barra de ferramentas existe também a função de limpar o painel de treinamento, apagando toda a sequência escolhida.

Para habilitar a opção de iniciar um treinamento, deve ser escolhida alguma sequência. Passando por essa validação, o sistema enviará cada número da sequência escolhida para a placa microcontrolada. O treinamento só poderá ser interrompido caso o usuário escolhe a opção *Parar treinamento* na barra de ferramentas, podendo ser recomeçado posteriormente.

A opção *Gerar relatório* só ficará habilitada caso o treinamento esteja finalizado completamente. Ao clicar nesta opção, a tela ilustrada na Figura 18 será exibida.

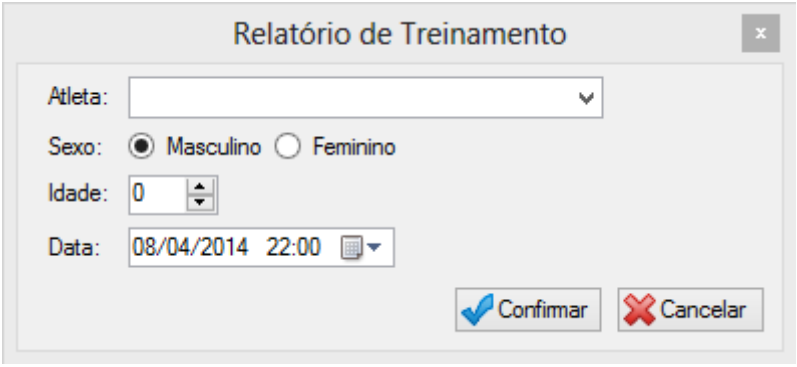

Figura 18. Tela de geração de relatório.

O usuário deverá preencher informações como: nome do atleta, sexo, idade e data de treinamento. Um arquivo ***.txt* será criado com os dados do atleta juntamente com os dados do treinamento e salvo em um diretório padrão escolhido pelo usuário através do botão *Configurações* localizado na barra de ferramentas (Figura 19).

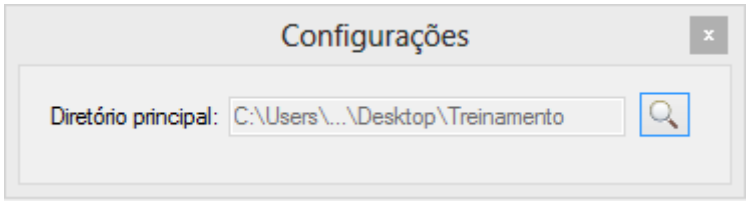

Figura 19. Tela de configurações.

Painel de treinamento:

O sistema possui um painel responsável por realizar a interação entre o treinamento e o usuário. Ele possui três elementos: LED, seletor de tempo de espera e visualizador de tempo de resposta. As regiões que possuem números de 1 a 24 na Figura 20 representarão os LEDs do painel de treinamento real utilizado pelo atleta. Ao lado de cada LED estará um seletor numérico para que o usuário escolha o tempo de espera em segundos para que cada um dos LEDs aguarde para acender. O tempo de espera definido por padrão será de 1s para cada LED.

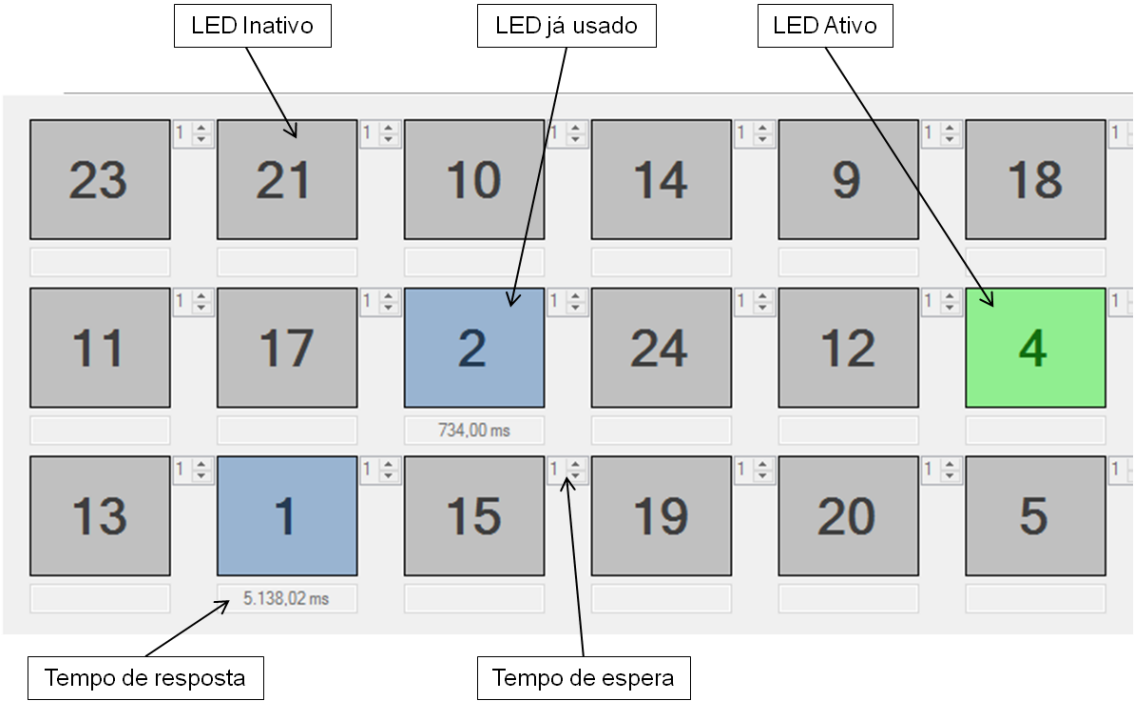

Figura 20. Painel de treinamento.

Como na Figura 20, os tempos de resposta (em milisegundos) do atleta para cada LED aceso ficará disponível para visualização assim que o botão correspondente for pressionado. Para facilitar a visualização, no painel será exibido a cor cinza para o LED que ainda não foi aceso, verde para o LED aceso e azul para o LED que já foi aceso alguma vez.

Barra de status:

Essa barra tem como finalidade informar ao usuário a situação da comunicação do dispositivo USB da placa controladora e o *software*. Abaixo estão as possíveis mensagens que serão exibidas:

- Dispositivo USB conectado.
- Dispositivo USB desconectado.

4.3.2 Fluxograma do *software*

Conforme ilustra o fluxograma da Figura 21, o sistema deverá aguardar até que o dispositivo seja conectado, assim o treinamento poderá ser iniciado se alguma sequência foi definida pelo usuário. A sequência possui números de 1 a 24, no modo automático é gerado de forma aleatória, já no modo manual o usuário deverá escolher a ordem da sequência podendo haver repetição de números.

Iniciado o treinamento com a sequência já definida, o próximo passo é enviar o primeiro comando para o dispositivo. Se o comando for o número *5*, por exemplo, o quinto LED do painel será aceso, já que os LEDs estão numerados com valores de 1 a 24. A cada comando enviado, o sistema aguardará até que o evento de recepção de dados pela USB seja disparado.

Assim que o for detectada o recebimento de dados, o sistema irá calcular o tempo de resposta do atleta em milissegundos a partir do número de *overflows* enviado pelo *firmware*. Segundo (PEREIRA, 2003), o cálculo total do tempo do *timer* 1 é dado por

$$T = T_{TMR} \times prescaler \times 2^{bits} \times overflows, \quad (1)$$

$$T_{TMR} = 4 / f_{CLK}. \quad (2)$$

A variável T_{TMR} é o período do *clock* que vai à entrada do *prescaler*, que no caso deste projeto equivale a 4 / 20 MHz. Ao passar pelo prescaler ocorre uma divisão de frequência de T_{TMR}, isto equivale a multiplicação do período pelo fator de divisão do *prescaler*, que no exemplo do projeto é 8. Para gerar cada estouro no *timer* 1 são necessárias 2^{16} contagens, então basta multiplicar pelo número de estouros recebido. Calculado o tempo de resposta do atleta, o ciclo se repete até toda a sequência escolhida termine, finalizando o treinamento.

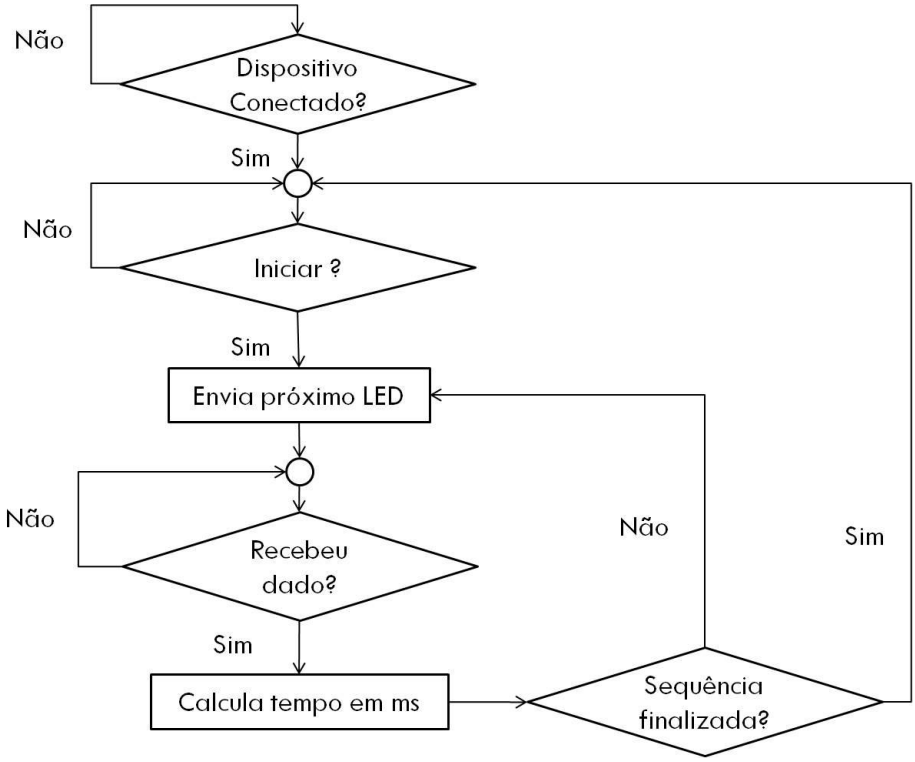

Figura 21. Fluxograma do *software*.

Capítulo 5

Testes da plataforma

Neste capítulo é apresentado como foram realizados os testes e simulações da plataforma. Será gerada uma sequência de comandos aleatória para mostrar o comportamento do sistema e a geração do relatório com os resultados do treinamento. Este capítulo irá apresenntar também a montagem e funcionamento de todo o *hardware* confeccionado durante o desenvolvimento do projeto.

5.1 Testes utilizando ferramenta de simulação

A simulação do sistema e a criação dos circuitos mostrados neste trabalho foram realizadas com a utilização da ferramenta computacional *Proteus*, da empresa *Labcenter Eletronics*. O uso do *Proteus* foi essencial neste trabalho, para a execução de testes de funcionamento dos circuitos e na depuração do código embarcado, auxiliando na correção de erros de programação.

Para a simulação foi criado um conjunto de *LEDs* e botões para representar um painel a ser utilizado por atletas. Como já mencionado, o painel contem 24 *LEDs* e 24 botões correspondentes alinhados como uma matriz de 3x8 (Figura 22). Esse painel funcionará em conjunto com o painel mostrado no *software* de interface com o usuário desenvolvido, o LED que for aceso no painel da simulação será exibido também no *software*, a fim de que o usuário possa acompanhar o treinamento em tempo real.

A Tabela 5 mostra a sequência de comandos gerados automaticamente pelo *software* que será utilizada neste teste. Para cada placa do painel, composta por um LED e um botão, é atribuído um número equivalente à ordem que essa determinada placa será acionada e um valor que representa o tempo de espera do acionamento dela. A Figura 23 mostra como fica a configuração da sequência no painel do *software*.

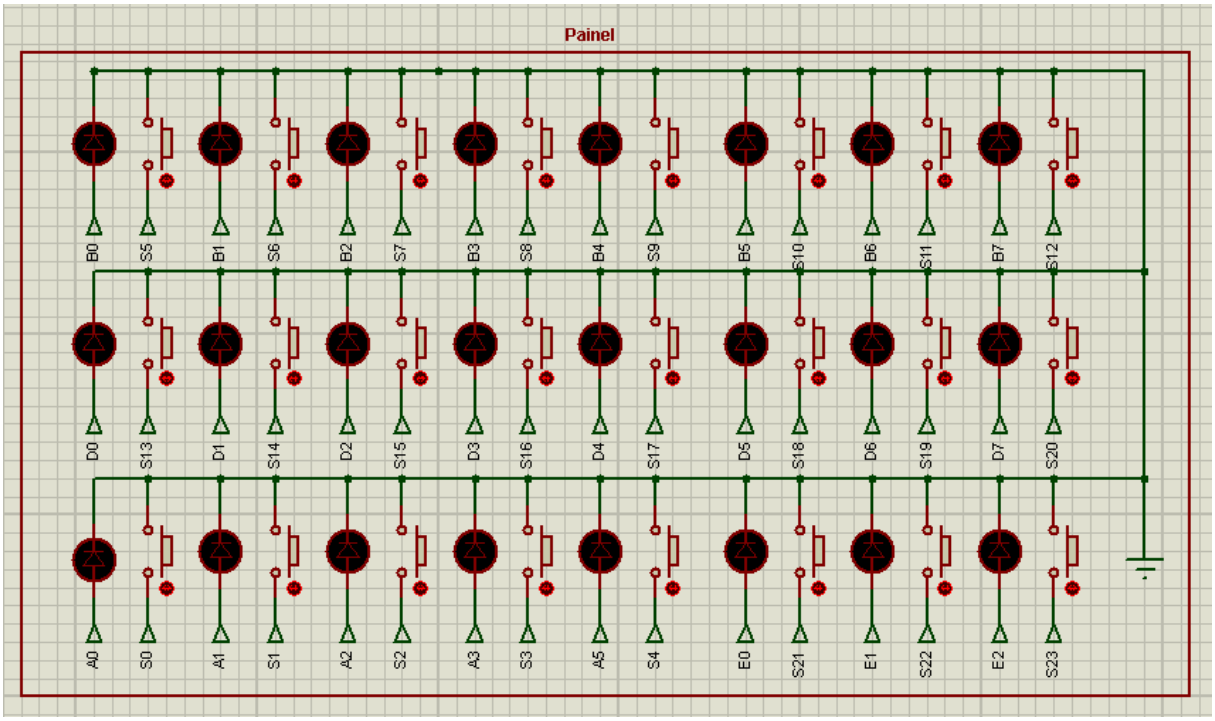

Figura 22. Painel de treinamento da simulação.

Tabela 5. Sequência de comandos para teste.

Placa	Ordem	Tempo de espera	Placa	Ordem	Tempo de espera
1	23	3 segundos	13	2	0 segundos
2	4	2 segundos	14	24	7 segundos
3	20	5 segundos	15	7	1 segundos
4	21	0 segundos	16	19	1 segundos
5	15	4 segundos	17	12	2 segundos
6	10	3 segundos	18	3	1 segundos
7	22	2 segundos	19	14	4 segundos
8	8	3 segundos	20	16	2 segundos
9	9	0 segundos	21	6	2 segundos
10	13	0 segundos	22	1	1 segundos
11	5	0 segundos	23	11	2 segundos
12	18	1 segundos	24	17	2 segundos

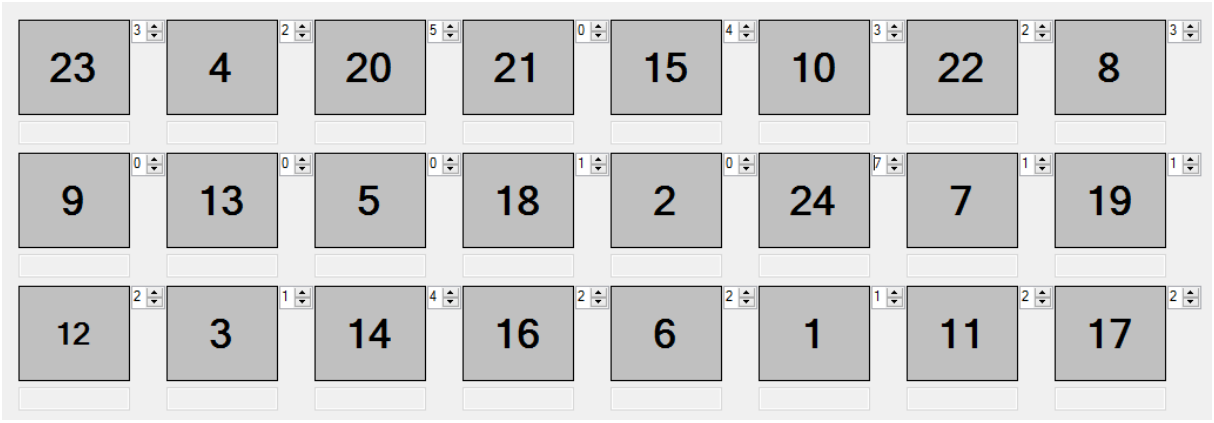

Figura 23. Painel de treinamento com sequência escolhida.

A execução dos comandos pode ser observada através da Figura 24 e Figura 25, mostrando a interação entre a simulação no *Proteus* e a aplicação desenvolvida. Pode ser observado que o tempo de resposta do atleta obtido é exibido assim que cada comando é finalizado. Assim que o treinamento é finalizado, uma mensagem é exibida na tela como ilustra a Figura 26, habilitando a opção de geração de relatórios.

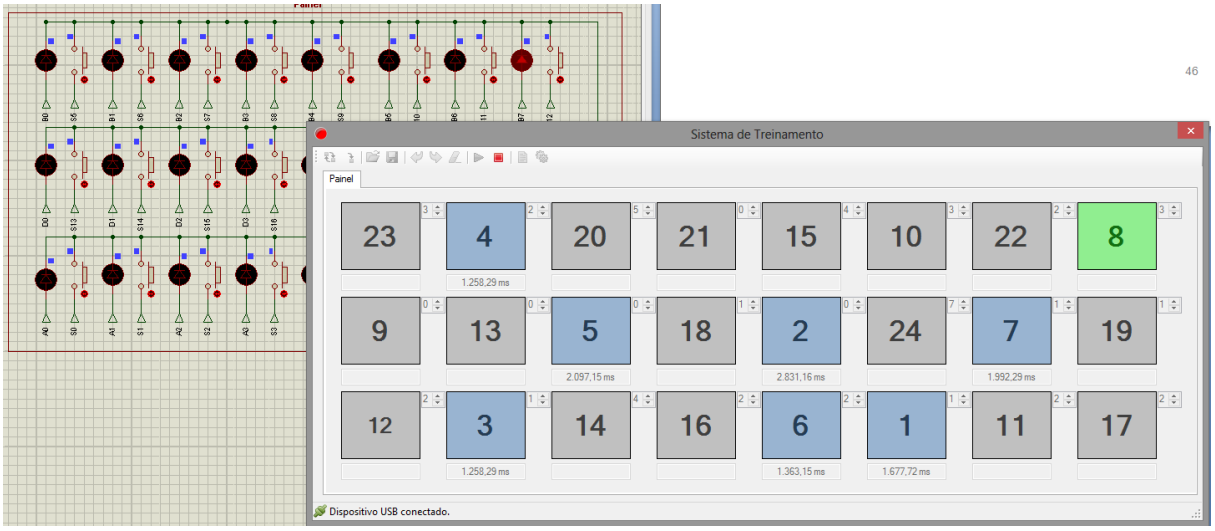

Figura 24. Execução de comandos (1)

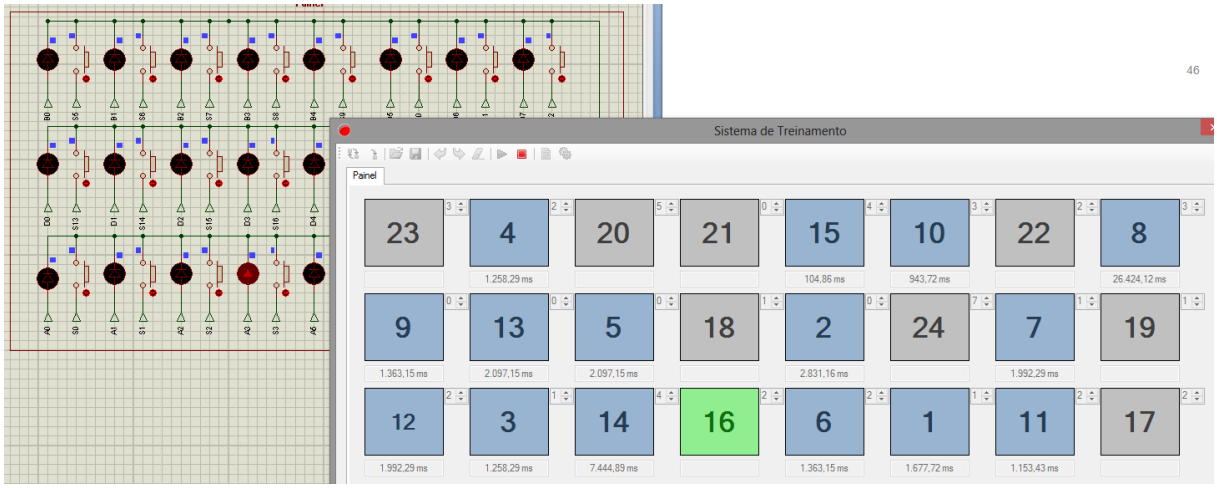

Figura 25. Execução de comandos (2).

Figura 26. Finalização do treinamento.

Poderá ser gerado um relatório com informações referentes ao desempenho do atleta. Como mostra a Figura 27, serão salvas informações como nome do atleta, idade, data e hora do treinamento, e os tempos de espera e resposta para cada LED da sequência utilizada. Através das informações disponibilizadas nos relatórios, poderão ser gerados gráficos comparativos entre atletas de mesma idade, reutilizar as sequências de um atleta para outro, tornando a avaliação do tempo de resposta do atleta mais fácil.

```
==========================================
Atleta: PEDRO BUARQUE
Data: 18/06/2014   10:32
Sexo: Masculino
Idade: 24 anos
Sequência utilizada: 3.txt
==========================================
Botão : Tempo de espera -> Tempo de resposta

22 : 1s <- 1.677,72 ms
13 : 0s <- 2.831,16 ms
18 : 1s <- 1.258,29 ms
2 : 2s <- 1.258,29 ms
11 : 0s <- 2.097,15 ms
21 : 2s <- 1.363,15 ms
15 : 1s <- 1.992,29 ms
8 : 3s <- 6.424,12 ms
9 : 0s <- 1.363,15 ms
6 : 3s <- 943,72 ms
23 : 2s <- 1.153,43 ms
17 : 2s <- 1.992,29 ms
10 : 0s <- 2.097,15 ms
19 : 4s <- 7.444,89 ms
5 : 4s <- 1.104,86 ms
20 : 2s <- 21.390,95 ms
24 : 2s <- 1.677,72 ms
12 : 1s <- 1.992,29 ms
16 : 1s <- 3.670,02 ms
3 : 5s <- 1.258,29 ms
4 : 0s <- 3.145,73 ms
7 : 2s <- 1.468,01 ms
1 : 3s <- 1.572,86 ms
14 : 7s <- 4.299,16 ms
==========================================
```

Figura 27. Relatório de treinamento do atleta.

5.2 Testes realizados no dispositivo

A instalação do sistema é dada de forma simples como já foi mencionado em capítulos anteriores. O dispositivo se comporta como um HID e é instalado no computador facilmente sem a necessidade de *drivers*. Quando o sistema é conectado ao computador é reconhecido como Placa de treinamento conforme Figura 28.

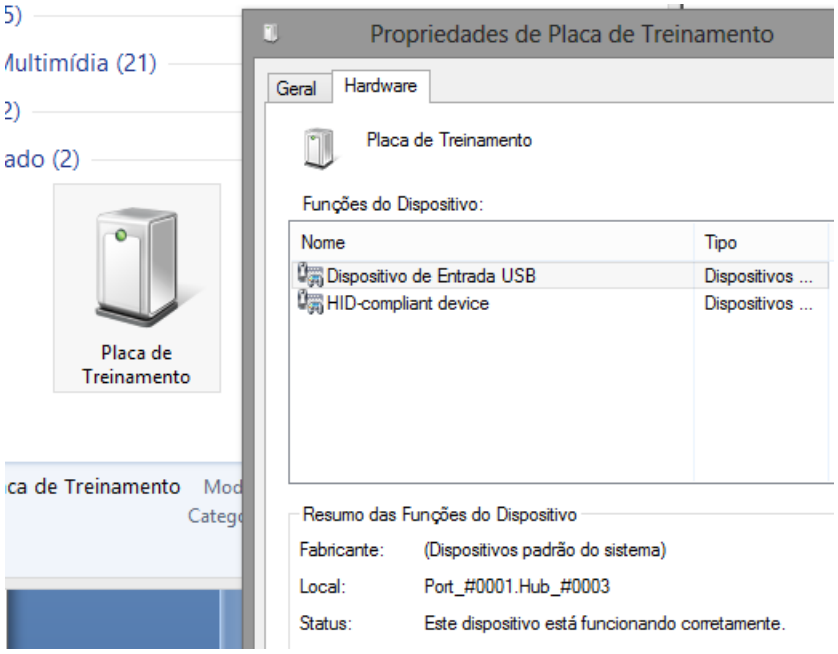

Figura 28. Reconhecimento do dispositivo no PC.

Todo o *hardware* mostrado durante o trabalho foi montado e testado para garantir seu funcionamento básico. A Figura 29 mostra a placa controladora conectada com a placa de conexões e a placa de interação já conectada no botão de emergência a ser utilizando em testes reais.

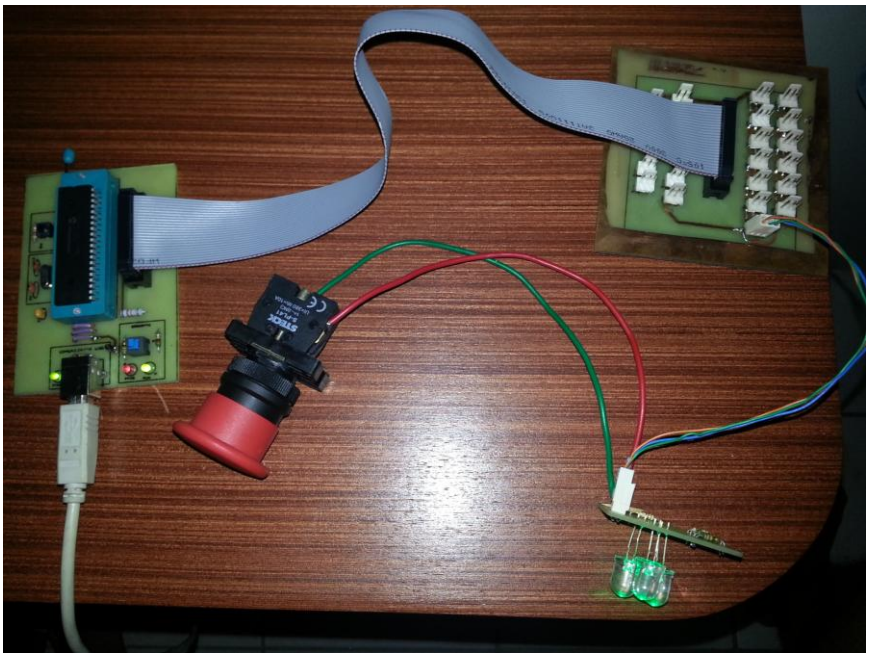

Figura 29. Montagem do circuito de testes (1).

Como uma forma de realizar o teste de todos os 24 LEDs do treinamento, evitando ter que confeccionar todas as 24 placas de interação, foi definida a confecção de uma única placa que viesse a representar o painel utilizado na simulação pelo *Proteus*. A Figura 30 mostra a placa representativa do painel conectada com as outras. Essa placa posui 24 circuitos reduzidos da placa interativa, com apenas um dos três LEDs.

Figura 30. Montagem do circuito de testes (2).

5.3 Resultados

Conforme ilustra a Figura 31, os testes realizados com a ferramenta computacional de simulação *Proteus* mostraram um ótimo funcionamento em todas as etapas do fluxo do sistema embarcado desenvolvido. Porém nos testes com o dispositivo desenvolvido foi notado um mau comportamento em duas etapas: na verificação do estado do botão e consequentemente no envio de pacotes pelo protocolo USB.

Etapa do fluxo do firmware	Simulação	Dispositivo
Reconhecimento pelo PC		
Recebimento de pacotes (USB)		
Verificação do estado do botão		
Envio de pacotes (USB)		

Figura 31. Comparação entre simulação e testes reais.

Falha na verificação do estado do botão:

Foi observado que o LED utilizado na simulação para os testes se comporta como um LED perfeito, sem capacitância parasita. Nos testes reais, a capacitância parasita pode estar presente tanto nos LEDs como nas trilhas e cabos utilizados. A carga dessa capacitância tem que descarregar passando pelo resistor de 1k *ohms*, pelo botão até chegar no terra. Então, todo esse percurso leva um certo tempo para descarregar e provavelmente deve ser maior do que 2 ciclos de máquina (0,4 microsegundos), que é o tempo que estava sendo utilizado. Foi necessário então, deixar por mais tempo o pino do microcontrolador configurado como entrada, dando tempo suficiente para a tensão cair para zero.

Falha no envio de pacotes (USB):

Mesmo com a solução do erro na verificação do estado do botão, a comunicação USB falhou na tentativa do envio de pacotes do *firmware* para o *software*, mas esse problema ainda não foi solucionado. Um modo de resolver seria criar um programa mais simples separado do código principal para testar a comunicação nos dois sentidos, no envio e recebimento de dados, para garantir o correto funcionamento.

Capítulo 6

Conclusão e trabalhos futuros

Este trabalho propoe a construção de uma plataforma que pudesse ser usada para realizar a medição e treinamento do tempo de resposta de atletas, utilizando as funcionalidades do microcontrolador PIC18F4550. Foi realizado o projeto de *software* e construção dos circuitos de *hardware*.

Os testes foram realizados utilizando ferramentas de simulação e foi possível observar um bom comportamento do sistema, tanto o *firmware* como a aplicação desenvolvida para realizar a interação entre o usuário e o sistema. Porém houve um problema não resolvido no envio de pacotes através do protocolo USB, detectado durante os testes com o dispositivo desenvolvido.

Houve uma grande dificuldade inicial para que a comunicação USB funcionasse corretamente. Foi necessário um estudo aprofundado no assunto para poder entender como funcionavam as funções específicas para a comunicação no *firmware*. Para que a comunicação USB do *software* de interação com o usuário funcionasse corretamente, foram testadas muitas bibliotecas disponibilizadas na internet até chegar à escolha ideal.

Com a geração de relatórios, disponibilizada no sistema, que possibilitam a avaliação do atleta durante cada treinamento, tornam esta aplicação muito útil para medir o tempo de reflexo e treinamento de atletas. Em um contexto mais geral, este mesmo sistema poderá ser adaptado e utilizado como ponto de partida para outros projetos com o intuito de medir tempos de resposta não só de humanos, mas também o tempo entre eventos ou fenômenos de qualquer natureza.

6.1 Trabalhos futuros

O primeiro trabalho sugerido é a melhoria do *hardware* desenvolvido para produzir as demais placas de interação para completar o projeto do treinamento. Outra melhoria é realizar uma investigação mais profunda no código desenvolvido a fim de corrigir os erros que permaneceram durante este trabalho.

Mundando o escopo, este projeto pode servir como como ponto de partida, por exemplo, para o desenvolvimento de alguma plataforma capaz de identificar melhorias no estado cognitivo de idosos que praticam atividades físicas, através da medição do tempo de resposta a determinados eventos.

Bibliografia

BOLTON, William. Mecatrônica : uma abordagem multidisciplinar – 4. Ed, Porto Alegre : Bookman, 2010;

IBRAHIM, Dogan. *Advanced PIC Microcontroller Projects in C*. Elsevier, 2008.

JOSEPH, M ; PANDYA, P. *Finding Response Times in a Real-Time System*. *The Computer Journal*. Vol. 29, Nº 5, 1986.

LIBERALQUINO, Diego. Desenvolvimento de Plataforma de Comunicação GSM/GPRS para Sistemas Embarcados. Trabalho de conclusão de curso. 2010, 52 p.

MARTINS, Nardênio Almeida, Sistemas Microcontrolados, São Paulo,SP, Novatec, 2005.

Microchip Technology Inc. Data Sheet do microcontrolador PIC18F4550, 2004. Disponível em <http://ww1.microchip.com/downloads/en/DeviceDoc/39632b.pdf> Acesso em: 11 de maio de 2014.

Microchip Tecnology Inc. High-Speed Serial Bootloader for PIC 16 and PIC18 devices, 2010. Disponível em <http://ww1.microchip.com/downloads/en/AppNotes/01

310a.pdf> Acesso em: 1 de maio de 2014.

Microsoft Corporation. Biblioteca MSDC: Diretivas de pré-processador. Disponível em <http://msdn.microsoft.com/pt-br/library/3sxhs2ty.aspx> Acesso em: 03 de maio de 2014.

NORMAN, D.; DRAPER, S. *User Centered System Design. Hillsdale, NJ. Lawrence Erlbaum*. 1986.

PEREIRA, Fábio. Microcontroladores PIC: Programação em C - São Paulo: Érica, 2003.

PRADO, Sérgio. Sistemas de tempo real. 2010. Disponível em <http://sergioprado.org/sistemas-de-tempo-real-part-1/>. Acesso em: 18 de Junho de 2014.

ROSSATO, Luana Callegaro; CONTREIRA, Andressa Ribeiro; CORAZZA, Sara Teresinha. Análise do tempo de reação e do estado cognitivo em idosas praticantes de atividades físicas . Fisioterapia e Pesquisa, [S.l.], v. 18, n. 1, p. 54-59 , mar. 2011. Disponível em: <http://www.revistas.usp.br/fpusp/article/view/12236>. Acesso em: 18 Jun. 2014.

SANTOS, L. Sistema de comunicação USB com microcontrolador. Trabalho de conclusão de curso. 2009, 69 p.

SILVA, V.F.; POLY, M.W.O.; RIBEIRO JÚNIOR, S.M.S.; CALOMENI, M.R.; PINTO, M.V.M.; SILVA, A.L.S. Efeito agudo da estimulação cerebral através de luz e som no tempo de reação motora de jovens atletas. Revista Digital - Buenos Aires - Ano 13 - N° 120 - Maio de 2008.

SOUZA, David José de. Desbravando o PIC. Edição 12. Editora Érica. 2001.

Apêndice A

Código fonte do *Firmware*

```
#include <18F4550.h>
#include <fuses_config.h>
#use delay(clock=48000000)

// Bibliotecas USB
#include <pic18_usb.h>
#include "usb_desc_hid.h"
#include <usb.c>

// Biblioteca de mapeamento do registrador TRIS
#include <trismap.h>

//Variáveis globais
int temp;
long time=0;
long int countIO=0;
unsigned char HID_InData[64];
unsigned char HID_OutData[64];

// Método responsável por desligar todos os LEDs
void turnOff()
{
   // Setando a Porta B bit a bit por conta do pino B4
   output_bit(pin_b0,0);
   output_bit(pin_b1,0);
   output_bit(pin_b2,0);
   output_bit(pin_b3,0);
   output_bit(pin_b5,0);
   output_bit(pin_b6,0);
   output_bit(pin_b7,0);

   output_d(0x00);
   output_a(0x00);
   output_e(0x00);
}

//Método responsável por setar todas as portas como entrada ou saída
void setTris(int value)
{
   set_tris_d(value);
   set_tris_a(value);
   set_tris_e(value);
```

```
  if(value == 0x00){
    set_tris_b(0x10);
    turnOff();
  }
  else
    set_tris_b(value);
}

void alterarLed(char key,int value)
{
  switch (key)
  {
    case 1:  setTris(0x00); output_bit(pin_b0,value);
         break;
    case 2:  setTris(0x00); output_bit(pin_b1,value);
         break;
    case 3:  setTris(0x00); output_bit(pin_b2,value);
         break;
    case 4:  setTris(0x00); output_bit(pin_b3,value);
         break;
    case 5:  setTris(0x00); output_bit(pin_a4,value);
         break;
    case 6:  setTris(0x00); output_bit(pin_b5,value);
         break;
    case 7:  setTris(0x00); output_bit(pin_b6,value);
         break;
    case 8:  setTris(0x00); output_bit(pin_b7,value);
         break;
    case 9:  setTris(0x00); output_bit(pin_d0,value);
         break;
    case 10: setTris(0x00); output_bit(pin_d1,value);
         break;
    case 11: setTris(0x00); output_bit(pin_d2,value);
         break;
    case 12: setTris(0x00); output_bit(pin_d3,value);
         break;
    case 13: setTris(0x00); output_bit(pin_d4,value);
         break;
    case 14: setTris(0x00); output_bit(pin_d5,value);
         break;
    case 15: setTris(0x00); output_bit(pin_d6,value);
         break;
    case 16: setTris(0x00); output_bit(pin_d7,value);
         break;
    case 17: setTris(0x00); output_bit(pin_a0,value);
         break;
    case 18: setTris(0x00); output_bit(pin_a1,value);
         break;
    case 19: setTris(0x00); output_bit(pin_a2,value);
```

```
            break;
        case 20: setTris(0x00); output_bit(pin_a3,value);
            break;
        case 21: setTris(0x00); output_bit(pin_a5,value);
            break;
        case 22: setTris(0x00); output_bit(pin_e0,value);
            break;
        case 23: setTris(0x00); output_bit(pin_e1,value);
            break;
        case 24: setTris(0x00); output_bit(pin_e2,value);
            break;
        default:
            break;
    }
}

//Método responsável por verificar se o botão foi pressionado
//Coloca o pino em modo de entrada, realiza a leitura e coloca o pino em modo de saída novamente
boolean checkButton(int key)
{
    boolean retorno = false;

    switch (key)
    {
        case 1:
            TRIS_B0 = 1;
            if(input(pin_b0) == 0) retorno = true;
            break;
        case 2:
            TRIS_B1 = 1;
            if(input(pin_b1) == 0) retorno = true;
            break;
        case 3:
            TRIS_B2 = 1;
            if(input(pin_b2) == 0) retorno = true;
            break;
        case 4:
            TRIS_B3 = 1;
            if(input(pin_b3) == 0) retorno = true;
            break;
        case 5:
            TRIS_A4 = 1;
            if(input(pin_a4) == 0) retorno = true;
            break;
        case 6:
            TRIS_B5 = 1;
            if(input(pin_b5) == 0) retorno = true;
            break;
```

```
case 7:
    TRIS_B6 = 1;
    if(input(pin_b6) == 0) retorno = true;
    break;
case 8:
    TRIS_B7 = 1;
    if(input(pin_b7) == 0) retorno = true;
    break;
case 9:
    TRIS_D0 = 1;
    if(input(pin_d0) == 0) retorno = true;
    break;
case 10:
    TRIS_D1 = 1;
    if(input(pin_d1) == 0) retorno = true;
    break;
case 11:
    TRIS_D2 = 1;
    if(input(pin_d2) == 0) retorno = true;
    break;
case 12:
    TRIS_D3 = 1;
    if(input(pin_d3) == 0) retorno = true;
    break;
case 13:
    TRIS_D4 = 1;
    if(input(pin_d4) == 0) retorno = true;
    break;
case 14:
    TRIS_D5 = 1;
    if(input(pin_d5) == 0) retorno = true;
    break;
case 15:
    TRIS_D6 = 1;
    if(input(pin_d6) == 0) retorno = true;
    break;
case 16:
    TRIS_D7 = 1;
    if(input(pin_d7) == 0) retorno = true;
    break;
case 17:
    TRIS_A0 = 1;
    if(input(pin_a0) == 0) retorno = true;
    break;
case 18:
    TRIS_A1 = 1;
    if(input(pin_a1) == 0) retorno = true;
    break;
case 19:
```

```
            TRIS_A2 = 1;
            if(input(pin_a2) == 0) retorno = true;
            break;
        case 20:
            TRIS_A3 = 1;
            if(input(pin_a3) == 0) retorno = true;
            break;
        case 21:
            TRIS_A5 = 1;
            if(input(pin_a5) == 0) retorno = true;
            break;
        case 22:
            TRIS_E0 = 1;
            if(input(pin_e0) == 0) retorno = true;
            break;
        case 23:
            TRIS_E1 = 1;
            if(input(pin_e1) == 0) retorno = true;
            break;
        case 24:
            TRIS_E2 = 1;
            if(input(pin_e2) == 0) retorno = true;
            break;
        default:
            break;
    }
    // Transformando o próximo comando em um default para evitar que entre nos
cases
    // caso o botão já tenha sido pressionado.
    if(retorno)
      temp = 0;

    return retorno;
}

//Método responsável pelas configurações da USB HID
void Setup_USB_HID(){
    usb_init();                    // Inicializa o USBb
    usb_task();
    usb_wait_for_enumeration();      // Aguarda o USB ser configurado pelo HOST
}

//Métdo responsável por tratar a recepção de dados da USB
void reception ()
{
    usb_get_packet(1,HID_InData,64); // Recebe dado via USB
    temp = HID_InData[0];

    // Comando para parar o treinamento
```

```
  if(temp ==  0){
    turnOff();
  }
  else{
    alterarLed(temp,1);      // Acende o próximo led
    enable_interrupts(int_timer1);  // Liga a interrupção do timer0
  }
}

//Método responsável por tratar a interrupção do TIMER 1
#INT_TIMER1
void timer()
{
  time++;
}

//Programa principal
void main (void)
{
  enable_interrupts(global);     //habilita a interrupção global
  Setup_timer_1(T1_INTERNAL|T1_DIV_BY_8);       //configura o timer um com o
preescaler de 1:8
  set_timer1(0);         //inicializa o timer com 0

  Setup_USB_HID();

  setTris(0x00);

  while (true)
  {
    //usb_task();
    if(usb_enumerated())
    {
      if (usb_kbhit(1))
        reception();


      if(countIO >= 2000 && countIO <= 4000)
      {
        if(checkButton(temp))
        {
          HID_OutData[0] = time;
          usb_put_packet(2,HID_OutData,64,USB_DTS_TOGGLE);

          // Reinicialização
          disable_interrupts(int_timer1);
          time = 0;
        }
      }
```

```
        else
        {
          if (countIO > 9999)
            countIO = 0;

          alterarLed(temp,1);
        }


        countIO++;
      }

    }
}
```

Apêndice B

Código fonte do *Software*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO.Ports;
using System.Threading;
using System.IO;
using UsbLibrary;
using TrainSystem.Properties;
using TrainSystem;

namespace Treinamento
{
   public partial class Main : Form
   {
      #region Constructor
      public Main()
      {
         InitializeComponent();

         #region Cores dos labels
         SILVER_STYLE = Color.Silver;
         GREEN_STYLE = Color.LightGreen;
         BLUE_STYLE = SystemColors.ActiveCaption;
         #endregion

         #region Mapeamento dos LEDs
         _Leds = new List<LED>();
         _Leds.Add(new LED(1, lblLed1, new List<NumericUpDown>() { num11,
num12, num13, num14 }, tbTime1));
         _Leds.Add(new LED(2, lblLed2, new List<NumericUpDown>() { num21,
num22, num23, num24 }, tbTime2));
         _Leds.Add(new LED(3, lblLed3, new List<NumericUpDown>() { num31,
num32, num33, num34 }, tbTime3));
         _Leds.Add(new LED(4, lblLed4, new List<NumericUpDown>() { num41,
num42, num43, num44 }, tbTime4));
         _Leds.Add(new LED(5, lblLed5, new List<NumericUpDown>() { num51,
num52, num53, num54 }, tbTime5));
```

```
            _Leds.Add(new LED(6, lblLed6, new List<NumericUpDown>() { num61,
num62, num63, num64 }, tbTime6));
            _Leds.Add(new LED(7, lblLed7, new List<NumericUpDown>() { num71,
num72, num73, num74 }, tbTime7));
            _Leds.Add(new LED(8, lblLed8, new List<NumericUpDown>() { num81,
num82, num83, num84 }, tbTime8));
            _Leds.Add(new LED(9, lblLed9, new List<NumericUpDown>() { num91,
num92, num93 ,num94 }, tbTime9));
            _Leds.Add(new LED(10, lblLed10, new List<NumericUpDown>() {
num101,num102,num103,num104 }, tbTime10));
            _Leds.Add(new LED(11, lblLed11, new List<NumericUpDown>() {
num111,num112,num113,num114 }, tbTime11));
            _Leds.Add(new LED(12, lblLed12, new List<NumericUpDown>() {
num121,num122,num123,num124 }, tbTime12));
            _Leds.Add(new LED(13, lblLed13, new List<NumericUpDown>() {
num131,num132,num133,num134 }, tbTime13));
            _Leds.Add(new LED(14, lblLed14, new List<NumericUpDown>() {
num141,num142,num143,num144 }, tbTime14));
            _Leds.Add(new LED(15, lblLed15, new List<NumericUpDown>() {
num151,num152,num153,num154 }, tbTime15));
            _Leds.Add(new LED(16, lblLed16, new List<NumericUpDown>() {
num161,num162,num163,num164 }, tbTime16));
            _Leds.Add(new LED(17, lblLed17, new List<NumericUpDown>() {
num171,num172,num173,num174 }, tbTime17));
            _Leds.Add(new LED(18, lblLed18, new List<NumericUpDown>() {
num181,num182,num183,num184 }, tbTime18));
            _Leds.Add(new LED(19, lblLed19, new List<NumericUpDown>() {
num191,num192,num193,num194 }, tbTime19));
            _Leds.Add(new LED(20, lblLed20, new List<NumericUpDown>() {
num201,num202,num203,num204 }, tbTime20));
            _Leds.Add(new LED(21, lblLed21, new List<NumericUpDown>() {
num211,num212,num213,num214 }, tbTime21));
            _Leds.Add(new LED(22, lblLed22, new List<NumericUpDown>() {
num221,num222,num223,num224 }, tbTime22));
            _Leds.Add(new LED(23, lblLed23, new List<NumericUpDown>() {
num231,num232,num233,num234 }, tbTime23));
            _Leds.Add(new LED(24, lblLed24, new List<NumericUpDown>() {
num241,num242,num243,num244 }, tbTime24));
            #endregion

            _SequenceFinished = false;
            _IsTraining = false;
            _Time = 0.0;
            _Sequence = new List<int>();

            _RefreshToolStrip();
        }
        #endregion
```

```
        #region Attributes
        private static Color SILVER_STYLE;
        private static Color GREEN_STYLE;
        private static Color BLUE_STYLE;
        private int _CurrentIndex;
        private List<int> _Sequence;
        private List<LED> _Leds;
        private Boolean _IsTraining;
        private double _Time;
        private bool _SequenceFinished;
        private bool _IsConnected;

        private Stack<List<ChangeObject>> _UndoActionsCollection = new
Stack<List<ChangeObject>>();
        private Stack<List<ChangeObject>> _RedoActionsCollection = new
Stack<List<ChangeObject>>();
        private Stack<List<int>> _UndoSequenceCollection = new Stack<List<int>>();
        private Stack<List<int>> _RedoSequenceCollection = new Stack<List<int>>();
        #endregion

        #region Methods

        #region Overrides
        protected override void OnHandleCreated(EventArgs e)
        {
            base.OnHandleCreated(e);
            usbHID.RegisterHandle(Handle);
        }

        protected override void WndProc(ref Message m)
        {
            usbHID.ParseMessages(ref m);
            base.WndProc(ref m);
        }
        #endregion

        #region Event Handlers
        #region Locals
        private void btnStart_Click(object sender, EventArgs e)
        {
            if (!_IsTraining)
            {
                _IsTraining = true;
                _RefreshToolStrip();
                _CurrentIndex = 0;
                _SequenceFinished = false;

                // Limpando as pilhas de Undo/Redo ao iniciar
                _UndoActionsCollection.Clear();
```

```
            _RedoActionsCollection.Clear();
            _UndoSequenceCollection.Clear();
            _RedoSequenceCollection.Clear();

            // Limpa os controles
            _CleanAllControls(false);

            // Desabilita todos os botões
            _EnableAllButtons(false);

            //Envia os dados
            this._SendData();
        }
        else
        {
            _IsTraining = false;
            _RefreshToolStrip();

            if(!_SequenceFinished)
                _CleanAllControls(false);

            _EnableAllButtons(true);

            // Envio do comando para parar o treinamento
            byte[] DataWrite = new byte[65];
            DataWrite[1] = (byte)0;
            this.usbHID.SpecifiedDevice.SendData(DataWrite);
        }
    }

    private void showLabelNumber(object sender, EventArgs e)
    {
        if (!_IsTraining)
        {
            Button button = sender as Button;
            LED led = _Leds.Find(l => l.Button.Equals(button));

            if (_Sequence.Count < (_Leds.Count * 4) && led.Repetitions <= 4)
            {
                _UndoSequenceCollection.Push(new List<int>(_Sequence));
                _UndoActionsCollection.Push(_BuildStruct());

                _Sequence.Add(led.Key);

                _RefreshToolStrip();
                switch (led.Repetitions)
                {
                    case 1:
                        button.Font = new Font("Microsoft Sans Serif", 25, FontStyle.Bold);
```

```
                    button.Text = _Sequence.Count.ToString();
                    led.WaitTimes[0].Value = 1;
                    led.WaitTimes[0].Visible = true;
                    break;
                case 2:
                    button.Font = new Font("Microsoft Sans Serif", 19, FontStyle.Bold);
                    button.Text += "/" + _Sequence.Count.ToString();
                    led.WaitTimes[1].Value = 1;
                    led.WaitTimes[1].Visible = true;
                    break;
                case 3:
                    button.Font = new Font("Microsoft Sans Serif", 19, FontStyle.Bold);
                    button.Text += (_Sequence.Count < 10 ? "   " : " ") +
_Sequence.Count.ToString();
                    led.WaitTimes[2].Value = 1;
                    led.WaitTimes[2].Visible = true;
                    break;
                case 4:
                    button.Font = new Font("Microsoft Sans Serif", 16, FontStyle.Bold);
                    button.Text += "/" + _Sequence.Count.ToString();
                    led.WaitTimes[3].Value = 1;
                    led.WaitTimes[3].Visible = true;
                    break;
            }
            led.Repetitions++;
        }
    }
}

private void saveSequence_Click(object sender, EventArgs e)
{
    saveFileDialog1.Title = "Salvar sequência";
    saveFileDialog1.FileName = string.Empty;
    openFileDialog1.FilterIndex = 0;


    if (_Sequence != null)
    {
        if (saveFileDialog1.ShowDialog(this) == DialogResult.OK)
            File.WriteAllText(string.Concat(saveFileDialog1.FileName, ".txt"),
_ConvertSequenceToText());
    }
    else
        MessageBox.Show("A sequência deve ser gerada.", "Erro",
MessageBoxButtons.OK, MessageBoxIcon.Error);
}

private void loadSequence_Click(object sender, EventArgs e)
{
```

```
            string text = string.Empty;

            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                _CleanAllControls(true);
                text = File.ReadAllText(openFileDialog1.FileName);

                if (!string.IsNullOrEmpty(text))
                {
                    _ConvertTextToSequence(text);
                    _EnableAllButtons(true);
                    _RefreshToolStrip();
                }
                else
                    MessageBox.Show("O arquivo está vazio", "Erro",
MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }

        private void btnSaveReport_Click(object sender, EventArgs e)
        {

        }

        private void btnReport_Click(object sender, EventArgs e)
        {
            TrainSystem.Report rpt = new TrainSystem.Report(_Leds, _Sequence);
            rpt.ShowDialog();
        }

        private void btnConfig_Click(object sender, EventArgs e)
        {
            Configurations config = new Configurations();
            config.ShowDialog();
        }

        private void btnErase_Click(object sender, EventArgs e)
        {
            _UndoActionsCollection.Push(_BuildStruct());
            _UndoSequenceCollection.Push(new List<int>(_Sequence));
            _CleanAllControls(true);
            _Sequence = new List<int>();
            _RefreshToolStrip();
        }

        private void btnAutomatic_Click(object sender, EventArgs e)
        {
            _UndoActionsCollection.Push(_BuildStruct());
            _UndoSequenceCollection.Push(new List<int>(_Sequence));
```

```
    _Sequence = _Shuffle();
    _EnableAllButtons(true);
    _CleanAllControls(true);
    _GenerateAutomaticSequence(true);

    tabControl1.Enabled = true;
    _RefreshToolStrip();
}

private void btnManual_Click(object sender, EventArgs e)
{
    _Sequence = new List<int>();
    _EnableAllButtons(true);
    _CleanAllControls(true);

    tabControl1.Enabled = true;
    _RefreshToolStrip();
}
#endregion

#region USB HID

private void Form3_Activated(object sender, EventArgs e)
{
    // Seta os valores PID e VID para que o dispositivo seja reconhecido.
    try
    {
        this.usbHID.ProductId = 32;//1;
        this.usbHID.VendorId = 1121;//4660;
        this.usbHID.CheckDevicePresent();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}

private void usbHID_OnSpecifiedDeviceArrived(object sender, EventArgs e)
{
    statusStrip1.Items[0].Text = "Dispositivo USB conectado.";
    statusStrip1.Items[0].Image = Resources.connect;

    _IsConnected = true;
    _RefreshToolStrip();
}

private void usbHID_OnSpecifiedDeviceRemoved(object sender, EventArgs e)
{
```

```
            statusStrip1.Items[0].Text = "Dispositivo USB desconectado.";
            statusStrip1.Items[0].Image = Resources.disconnect;

            _CleanAllControls(true);
            _EnableAllButtons(false);
            _Sequence = new List<int>();
            _IsConnected = _IsTraining =false;
            _RefreshToolStrip();
        }

        private void usbHID_OnDataRecieved(object sender, DataRecievedEventArgs
args)
        {
            if (InvokeRequired)
            {
                try
                {
                    Invoke(new DataRecievedEventHandler(usbHID_OnDataRecieved),
new object[] { sender, args });
                }
                catch (Exception ex)
                {
                    statusStrip1.Items[0].Text = ex.Message;
                }
            }
            else
            {
                try
                {
                    int data = (int)args.data[1];
                    _Time = (data * 65536 * 8 * 0.2) / 1000;

                    this._FillTimes();

                    if (_CurrentIndex < _Sequence.Count - 1)
                        _CurrentIndex++;
                    else
                        _SequenceFinished = true;

                    this._SendData();
                }
                catch (Exception e)
                {
                    statusStrip1.Items[0].Text = e.Message;
                }
            }
        }

        private void usbHID_OnDeviceArrived(object sender, EventArgs e)
```

```
{
    this.usbHID.CheckDevicePresent();
}

private void usbHID_OnDeviceRemoved(object sender, EventArgs e)
{
    this.usbHID.CheckDevicePresent();
}
#endregion

#region UndoRedo
private void btnUndo_Click(object sender, EventArgs e)
{
    if (_UndoActionsCollection.Count == 0) return;

    List<ChangeObject> UndoStruct = _UndoActionsCollection.Pop();
    List<ChangeObject> oldStruct = new List<ChangeObject>();
    List<int> oldSequence = _Sequence;
    _Sequence = _UndoSequenceCollection.Pop();
    foreach (var obj in UndoStruct)
    {
        LED led = _Leds.Find(l => l.Key == obj.Key);
        oldStruct.Add(new ChangeObject(led));

        led.Button.Text = obj.ButtonText;
        led.Repetitions = obj.Repetitions;
        for (int i = 0; i < led.WaitTimes.Count; i++)
        {
            led.WaitTimes[i].Value = obj.WaitTimes[i].Value;
            led.WaitTimes[i].Visible = obj.WaitTimes[i].Visible;
        }
    }

    _RedoActionsCollection.Push(oldStruct);
    _RedoSequenceCollection.Push(oldSequence);


    _RefreshToolStrip();
}

private void btnRedo_Click(object sender, EventArgs e)
{
    if (_RedoActionsCollection.Count == 0) return;

    List<ChangeObject> UndoStruct = _RedoActionsCollection.Pop();
    List<ChangeObject> newStruct = new List<ChangeObject>();
    List<int> oldSequence = _Sequence;
    _Sequence = _RedoSequenceCollection.Pop();
    foreach (var obj in UndoStruct)
```

```
            {
                LED led = _Leds.Find(l => l.Key == obj.Key);
                newStruct.Add(new ChangeObject(led));

                led.Button.Text = obj.ButtonText;
                led.Repetitions = obj.Repetitions;
                for (int i = 0; i < led.WaitTimes.Count; i++)
                {
                    led.WaitTimes[i].Value = obj.WaitTimes[i].Value;
                    led.WaitTimes[i].Visible = obj.WaitTimes[i].Visible;
                }
            }

            _UndoActionsCollection.Push(newStruct);
            _UndoSequenceCollection.Push(new List<int>(oldSequence));

            _RefreshToolStrip();
        }

        private List<ChangeObject> _BuildStruct()
        {
            List<ChangeObject> newStruct = new List<ChangeObject>();
            foreach (LED led in _Leds)
                newStruct.Add(new ChangeObject(led));

            return newStruct;
        }
        #endregion
        #endregion

        #region Auxiliaries
        private void _RefreshToolStrip()
        {
            bool isTraining = (_IsConnected && _IsTraining);

            btnOpenSequence.Enabled = btnAutomatic.Enabled = btnManual.Enabled =
btnConfig.Enabled = (_IsConnected && !_IsTraining);
            btnStart.Enabled = btnErase.Enabled = btnSaveSequence.Enabled =
!isTraining && (_Sequence != null && _Sequence.Count > 0);
            btnStop.Enabled = isTraining;

            btnUndo.Enabled = !isTraining && (_UndoActionsCollection.Count > 0);
            btnRedo.Enabled = !isTraining && (_RedoActionsCollection.Count > 0);

            btnReport.Enabled = _SequenceFinished;
        }

        private void _SendData()
        {
```

```
        try
        {
            if (this.usbHID.SpecifiedDevice != null)         // Verifica se o dispositivo foi
conectado.
            {
                int data = _Sequence[_CurrentIndex];
                LED led = _Leds.Find(l => l.Key.Equals(data));
                int value = (int)(led.WaitTimes[led.WaitCount++].Value * 1000);

                if (!_SequenceFinished)
                {
                    _FillLabels(data);
                    Thread.Sleep(value);

                    // Envio de dados
                    byte[] DataWrite = new byte[65];
                    DataWrite[1] = (byte)data;
                    this.usbHID.SpecifiedDevice.SendData(DataWrite);
                }
                else
                {
                    MessageBox.Show("A sequência de treinamento foi finalizada.",
"Conclusão", MessageBoxButtons.OK, MessageBoxIcon.Information);
                    _RefreshToolStrip();
                    this.Invoke(new EventHandler(btnStart_Click));
                }
            }
            else
            {
                MessageBox.Show("Não foi possível enviar dados.");
            }
        }
        catch (Exception e)
        {
            statusStrip1.Items[0].Text = e.Message;
        }
    }

    private void _FillTimes()
    {
        int data = _Sequence[_CurrentIndex];
        LED led = _Leds.Find(l => l.Key == data);
        led.TimesPressed++;
        led.Button.BackColor = led.TimesPressed == led.Repetitions - 1 ?
BLUE_STYLE : SILVER_STYLE;
        led.ResponseTimes.Add(_Time);
        led.ResponseBox.Text = string.Concat(_Time.ToString("#,##0.00"), " ms");
    }
```

```
        private void _FillLabels(int data)
        {
            LED led = _Leds.Find(l => l.Key == data);
            led.Button.BackColor = GREEN_STYLE;
        }

        private List<int> _Shuffle()
        {
            List<int> list = new List<int>();
            foreach (LED item in _Leds)
                list.Add(item.Key);

            Random rng = new Random();
            int n = list.Count;
            int loops = 3;

            while (loops > 0)
            {
                while (n > 1)
                {
                    n--;
                    int k = rng.Next(n + 1);
                    int value = list[k];
                    list[k] = list[n];
                    list[n] = value;
                }
                loops--;
            }

            return list;
        }

        private void _GenerateAutomaticSequence(bool cleanWaitCount)
        {
            for (int i = 0; i <= _Sequence.Count - 1; i++)
            {
                LED led = _Leds.Find(l => l.Key == _Sequence[i]);
                switch (led.Repetitions)
                {
                    case 1:
                        led.Button.Font = new Font("Microsoft Sans Serif", 25,
FontStyle.Bold);
                        led.Button.Text = (i + 1).ToString();
                        led.WaitTimes[0].Value = (cleanWaitCount ? 1 :
led.WaitTimes[0].Value);
                        led.WaitTimes[0].Visible = true;
                        break;
                    case 2:
```

```
                    led.Button.Font = new Font("Microsoft Sans Serif", 19,
FontStyle.Bold);
                    led.Button.Text += "/" + (i + 1).ToString();
                    led.WaitTimes[1].Value = (cleanWaitCount ? 1 :
led.WaitTimes[1].Value);
                    led.WaitTimes[1].Visible = true;
                    break;
                case 3:
                    led.Button.Font = new Font("Microsoft Sans Serif", 19,
FontStyle.Bold);
                    led.Button.Text += ((i + 1) < 10 ? "   " : " ") + (i + 1).ToString();
                    led.WaitTimes[2].Value = (cleanWaitCount ? 1 :
led.WaitTimes[2].Value);
                    led.WaitTimes[2].Visible = true;
                    break;
                case 4:
                    led.Button.Font = new Font("Microsoft Sans Serif", 16,
FontStyle.Bold);
                    led.Button.Text += "/" + (i + 1).ToString();
                    led.WaitTimes[3].Value = (cleanWaitCount ? 1 :
led.WaitTimes[3].Value);
                    led.WaitTimes[3].Visible = true;
                    break;
            }
            led.Repetitions++;
        }
    }

    private void _EnableAllButtons(bool enable)
    {
        foreach (LED led in _Leds)
        {
            led.Button.Enabled = enable;
            led.WaitTimes.ForEach(w => w.Enabled = enable);
        }
    }

    private void _CleanAllControls(bool cleanSequence)
    {
        foreach (LED led in _Leds)
        {
            if (cleanSequence)
            {
                led.Button.Text = string.Empty;

                led.WaitTimes.ForEach(w => w.Value = 0);
                led.WaitTimes.ForEach(w => w.Visible = false);
                led.Repetitions = 1;
                led.Button.Font = new Font("Microsoft Sans Serif", 25);
```

```
            }

            led.Button.BackColor = SILVER_STYLE;
            led.ResponseTimes.Clear();
            led.TimesPressed = 0;
            led.ResponseCount = 0;
            led.ResponseBox.Text = string.Empty;
            led.WaitCount = 0;
        }
    }

    private string _ConvertSequenceToText()
    {
        string text = string.Empty;
        int key = 0;
        string wTime = string.Empty;

        if (_Sequence != null)
        {
            _Leds.ForEach(l => l.WaitCount = 0);

            for (int i = 0; i < _Sequence.Count; i++)
            {
                // Pega o valor da chave i da sequência
                key = _Sequence[i];
                // Pega o valor do tempo de espera referente à chave
                LED led = _Leds.Find(l => l.Key == key);
                wTime = led.WaitTimes[led.WaitCount++].Value.ToString();
                // Concatena a chave com o tempo na variável text. ex.: 10:2;4:6;8:0 ...
                text += i == (_Sequence.Count - 1) ? string.Concat(key, ":", wTime) :
string.Concat(key, ":", wTime, ";");
            }
        }
        return text;
    }

    private void _ConvertTextToSequence(string text)
    {
        string[] tuples = text.Split(';');
        string[] values = null;
        _Sequence = new List<int>();
        int key;
        decimal wTime;
        bool errorOccurred = false;

        if (tuples != null)
        {
            _Leds.ForEach(l => l.WaitCount = 0);
```

```
            foreach (string num in tuples)
            {
                values = num.Split(':');
                if (values != null && values.Length == 2)
                {
                    if (decimal.TryParse(values[1], out wTime))
                    {
                        key = int.Parse(values[0]);
                        _Sequence.Add(key);
                        LED led = _Leds.Find(l => l.Key == key);
                        led.WaitTimes[led.WaitCount++].Value = wTime;
                    }
                    else
                    {
                        errorOccurred = true;
                        break;
                    }
                }
                else
                {
                    errorOccurred = true;
                    break;
                }
            }
            if (!errorOccurred)
                _GenerateAutomaticSequence(false);
            else
                MessageBox.Show("O arquivo está em um formato inválido.", "Erro",
MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }

    #endregion

    #endregion

}

public class ChangeObject
{
    public ChangeObject(LED led)
    {
        Key = led.Key;
        ButtonText = led.Button.Text;
        Repetitions = led.Repetitions;
        WaitTimes = new List<Numeric>();

        foreach(var time in led.WaitTimes)
            WaitTimes.Add(new Numeric(time.Value, time.Visible));
```

```
        }

        public int Key;
        public string ButtonText;
        public int Repetitions;
        public List<Numeric> WaitTimes;
    }

    public class Numeric
    {
        public Numeric(decimal value, bool visible)
        {
            this.Value = value;
            this.Visible = visible;
        }

        public decimal Value;
        public bool Visible;
    }
}
```