Centro Universitário de Brasília – UniCEUB
Faculdade de Ciências Exatas e de Tecnologia – FAET
Curso de Engenharia da Computação
Disciplina: Projeto Final
Professora Orientadora: Profª. M.C. Maria Marony Sousa Farias Nascimento

TRANSMISSÃO MULTINÍVEL E DETECÇÃO E CORREÇÃO DE ERROS NO PROJETO TRANSMISSÃO ALTERNATIVA DE DADOS

João Paulo Barbosa Fernandes
Matrícula: 2.011.497-0

Brasília/DF, 2º semestre de 2005.

RESUMO

Neste trabalho é apresentada a construção de um modelo acadêmico de transmissão de dados do tipo simplex, entre dois microcomputadores, utilizando como dispositivo de transmissão o monitor de vídeo.

O microcomputador transmissor converte o texto a ser transmitido em variações de cores na escala de cinza em seu monitor de vídeo, do qual um módulo de interface, construído especificamente para este fim, reconhece estas variações e reconstrói o texto transmitido escrevendo este na porta serial do microcomputador receptor.

Palavras chave: transmissão de dados, monitor de vídeo, fotônica, paridade, transmissão multinível, microcontrolador, PIC, fotodiodo.

ABSTRACT

In this work is presented the construction of an academic model of data-communication of the simplex type, between two microcomputers, using as transmission device the video display.

The transmitter microcomputer converts the text to be transmitted in variations of colors in the gray scale in your video display, of which a module of interface, constructed specifically for this end, recognizes these variations and reconstructs the transmitted text writing this in the serial port of the receiver microcomputer.

Key-words: transmission of data, video display, photon, parity, multilevel transmission, microcontroller, PIC, photodiode.

AGRADECIMENTOS

Agradecer sempre é difícil: sempre se corre o risco de esquecer alguém. Assim, acho que primeiro eu vou agradecer a todo o mundo, que se eu acabar não citando alguma pessoa que me ajudou, seja direta ou indiretamente, ela não estará esquecida de todo. Mas creio que o mais justo é que eu me lembre dos professores, pois eles participaram diretamente do aprendizado que me levou ao desenvolvimento do projeto de pesquisa que culminou nesta monografia. E fica também meu agradecimento especial a minha professora orientadora Maria Marony, ao professor Francisco Javier, ao professor Abiezer Amarilha e ao professor Miguel Arcanjo.

Existe também um punhado de gente que não tem nada a ver com a monografia, mas tem a ver comigo, então eu não poderia deixar de lembrar deles aqui. Meus pais, por exemplo. Eles não me deram idéias para a dissertação, tampouco corrigiram meus parágrafos, mas em compensação acompanharam toda minha trajetória até aqui. São minha riqueza; e me considero milionário.

Além dos meus pais, eu tenho dois amigos que em especial tiveram papel importante em minha vinda e estabelecimento em Brasília, que por conseqüência resultou em meu ingresso neste Centro Universitário. São eles o Oldair Gomes e o Valdinei Pinto.

Convém também que eu não esqueça de minha namorada, a Quelma Oliveira, que sempre emprestou alguma cor a estes dias cinzentos de Brasília. Sem ela todo este caminho seria, com certeza, bem mais árduo. Agradeço a vários outros amigos que de uma forma ou de outra, ao passar por minha vida tenham contribuído com ensinamentos que carrego até hoje. Em especial cito o Dorcelino Pereira, pessoa sábia em todos os sentidos seja emocional, intelectual ou profissional.

Finalmente, eu preciso agradecer aos amigos Ana Paula Silvestre, Alessandro Catão Mito Kuramoto, Diogo Curto, Toni Gledson Dantas e Sebastião Fabiano Silva pois não foram apenas colegas de faculdade, mas sim verdadeiros amigos e companheiros.

SUMÁRIO

1. INTRODUÇÃO 1
1.1. Contextualização do Projeto 1
1.2. Objetivo do Projeto 2
1.3. Motivação 3
1.4. Estrutura do Trabalho 5
2. REVISÂO BIBLIOGRÁFICA 6
2.1. O Sistema de Cores RGB 6
2.2. A não-linearidade do modelo RGB 6
2.3. A Escala de Cinza 7
2.4. A API Direct X 7
2.5. Técnica de Double Buffering 8
2.6. Técnica de Page-flipping 9
2.7. Paridade de Caractere 10
2.8. Paridade Combinada (ou Biparidade) 11
2.9. Comunicação com RS-232 12
2.10. USART 14
3. CARACTERÍSTICA DOS MÓDULOS ENVOLVIDOS NA COMUNICAÇÃO 15
3.1. Transmissor 15
3.1.1. O monitor de vídeo 16
3.1.2. O Software Transmissor 19
3.2. A Interface 20
3.2.1. O Fotodetector 21
3.2.2. O Microcontrolador 22
3.2.3. O Software Embarcado 25
3.2.4. O Conversor de Nível 26
3.3. Receptor 28
4. CONSTRUÇÃO DO SOFTWARE TRANSMISSOR 29
4.1. Função do Software Transmissor 29
4.2. Modo de Chamada do Software Transmissor 30
4.3. Declaração e Inicialização dos Objetos DirectX Utilizados 31
4.4. Decomposição do Dado a ser Transmitido em Bits 38
4.5. Cálculo da Paridade 40
4.6. Freqüência de Transmissão 40
4.7. Transmissão da Matriz de Bits 41
5. CONSTRUÇÃO DO SOFTWARE RECEPTOR 44
5.1. A API de Comunicação da SUN 44
5.1.1. Reconhecendo as Portas 44
5.1.2. Abrindo Portas para Comunicação 45
5.1.3. Recebendo Dados da Porta Serial 46
5.2. Classes do Software Receptor 48
5.3. O Padrão de Projetos Observer 49
5.4. Interface IPublisher 49
5.5. Interface ISubscriber 50
5.6. Classe Receptor 50
5.7. Classe SoftwareReceptor 50
5.7.1. Validando a Paridade dos Dados Recebidos 51
6. CONSTRUÇÃO DA INTERFACE 52

6.1. O Regulador de Tensão 52
6.2. O Fotodiodo 52
6.3. O Conversor A/D 54
6.4. O Cristal 55
7. CONSTRUÇÃO DO SOFTWARE EMBARCADO NO MICROCONTROLADOR 57
7.1. Fluxograma 57
7.2. Estratégia do Software Embarcado 58
7.3. Arquivos de Cabeçalho e Configurações Gerais 60
7.4. Configuração do Conversor A/D 62
7.5. Configuração do Timer 1 62
7.6. Sincronização com o Monitor de Vídeo 63
7.7. A Rotina de Interrupção 65
8. CONSIDERAÇÕES FINAIS 66
8.1. Dificuldades Encontradas 66
8.2. Resultados Obtidos 66
8.3. Conclusões 67
8.4. Sugestões para Trabalhos Futuros 68
REFERÊNCIAS BIBLIOGRÁFICAS 70
Apêndice A – Código Fonte Completo do Software Transmissor 72
Apêndice B – Código Fonte Completo do Software Receptor 80
Apêndice C – Código Fonte Completo do Software Embarcado do Microcontrolador 89
Apêndice D – Fotos Reais da Interface 91
Apêndice E – Diagrama Elétrico Completo da Interface 98

ÍNDICE DE FIGURAS

Figura 1.1 - Modelo de Transmissão....3
Figura 2.1 - Técnica de Double-Buffering....9
Figura 2.2 - Técnica de Page-Fipping....10
Figura 2.3 - Conector DB9 e seus Respectivos Pinos....14
Figura 3.1 - Diagrama de Blocos do Projeto....15
Figura 3.2 - O tubo de raios catódicos....17
Figura 3.3 - Diferença entre as transmissões em 2 níveis e 4 níveis....20
Figura 3.4 - Diagrama de Blocos da Interface....21
Figura 3.5 - Gráfico de comparação LDR x Fotodiodo....22
Figura 3.6 - Pinagem do PIC16F877A....25
Figura 3.7 - Diagrama de Configuração do MAX232....28
Figura 5.1 - Diagrama de Classes do Padrão Observer....49
Figura 6.1 - Circuito esquemático da alimentação....52
Figura 6.2 - Símbolo esquemático do fotodiodo....53
Figura 6.3 - Funcionamento de um fotodiodo ideal....54
Figura 6.4 - Circuito esquemático do transdutor....54
Figura 6.5 - Esquema do Conversor A/D dos PIC's....55
Figura 6.6 - Clock de um microcontrolador a partir de um cristal de quartzo....56
Figura 6.7 - Sinal de clock do oscilador depois de ser ligada a alimentação....56
Figura 7.1 - Variação da Intensidade Luminosa do Monitor de Vídeo (cor cinza)....59
Figura 7.2 - Variação da Intensidade Luminosa do Monitor de Vídeo (cor Branca)....65
Figura A - Interface em Tempo de Projeto....91
Figura B - Receptor Conectado ao Monitor de Vídeo em Tempo de Projeto....92
Figura C - Versão Final da Interface....93
Figura D - Microcontrolador, Cristal e Botão de Reset....94
Figura E - MAX 232, Capacitores e Conector DB9....95
Figura F - LM 7805, Capacitores, Led de Power e Bateria....96
Figura G - Visão Traseira da Interface....97

ÍNDICE DE TABELAS

Tabela 2.1 - Exemplo de Cálculo de Paridade 11
Tabela 2.2 - Exemplo de Paridade Combinada.....12
Tabela 2.3 - Exemplo de Paridade Combinada com Erro no 4º bit do 4º caractere ..12
Tabela 2.4 - Pinos para Comunicação Serial 13
Tabela 6.1 - Valores de Capacitor utilizados junto ao cristal.....55

LISTA DE SIMBOLOS

API – Application Programming Interface
BCC - Block Check Character
BLIT - Bit Block Transfer
CRT – Tubo Catódico de Raios
DTE - Data Terminal Equipment
EEPROM - Electrically Erasable Programmable Read Only Memory
EIA - Eletronic Industries Association
GDI - Graphical Device Interface
GUID - Globally Unique Identifier
LDR – Light Dependent Resistor
LRC - Longitudinal Redundancy Checking
LVP – Low Voltage Program
MCU - Micro Controler Unit
PC – Computador Pessoal
RAM - Random Access Memory
RGB – Red, Green e Blue
ROM - Read Only Memory
SCI - Interface de Comunicação Serial
UART - Universal Asynchronous Receiver and Transmitter
USB – Universal Serial Bus
VRC - Vertical Redundancy Checking
WDT – Watchdog Timer

1. INTRODUÇÃO

1.1. Contextualização do Projeto

A comunicação é uma das maiores necessidades da sociedade humana desde os primórdios de sua existência. Conforme as civilizações se espalhavam, ocupando áreas cada vez mais dispersas geograficamente, a comunicação à longa distância se tornava cada vez mais uma necessidade e um desafio. Formas de comunicação através de sinais de fumaça ou pombos-correio foram as maneiras encontradas por nossos ancestrais para tentar aproximar as comunidades distantes (Soares 95).

A invenção do telégrafo por Samuel F. B. Morse em 1838 inaugurou uma nova época nas comunicações. Nos primeiros telégrafos utilizados no século XIX, mensagens eram codificadas em cadeias de símbolos binários (código Morse) e então transmitidas manualmente por um operador através de um dispositivo gerador de pulsos elétricos. Desde então, a comunicação através de sinais elétricos atravessou uma grande evolução, dando origem à maior parte dos grandes sistemas de comunicação que temos hoje em dia, como o telefone, o rádio e a televisão (Keiser 89).

Na década de 1970, a comunicação encontrou seu grande aliado na realização da revolução que vivenciamos hoje: o computador.

Mudanças na caracterização dos sistemas de computação ocorreram durante esta década, de um sistema único centralizado e de grande porte, disponível para todos os usuários de uma determinada organização, partia-se em direção à distribuição do poder computacional. O desenvolvimento de minis e microcomputadores de bom desempenho, com requisitos menos rígidos de temperatura e umidade, permitiu a instalação de considerável poder computacional em várias localizações de uma organização, ao invés da anterior concentração deste poder em uma determinada área.

Ambientes de trabalho cooperativos se tornaram uma realidade tanto nas empresas como nas universidades, exigindo a interconexão dos equipamentos nessas organizações.

Também na década de 1970, a Xerox passou a pesquisar o conceito de computador móvel, o Dynabook, um micro do tamanho de um notebook. Este foi o

ponto de partida para a concepção e popularização dos *Personal Digital Assistants* (PDA's ou *Handhelds*), sendo os mais populares os da linha Palm.

A Palm Computing Inc., presidida por Jeff Hawkins, começou a vender o Pilot em abril de 1996. Essa maquininha ágil (depois rebatizada de PalmPilot) cabia num bolso de camisa, armazenava milhares de endereços e compromissos e custava barato. Em 18 meses a Palm vendeu mais de 1 milhão de Pilots.[1]

O número de PDAs no mundo vem crescendo de forma exponencial, mas tendências indicam que em poucos anos os SmartPhones (desenvolvido através da "fusão" entre um PDA e um telefone celular) serão maioria absoluta.[2]

Nestes dispositivos móveis há sempre a necessidade de comunicação de dados com um microcomputador de mesa (*desktop*) para sincronização de agenda, cópia de arquivos texto ou planilhas, transferência de músicas, imagens e *ringtones* (estes últimos principalmente para celulares).

É neste contexto que se insere o trabalho aqui apresentado, demonstrando que é possível, mesmo que com enfoque acadêmico, desenvolver uma nova abordagem de comunicação entre dispositivos móveis e computadores de mesa.

1.2. Objetivo do Projeto

Neste projeto é apresentado um modelo didático de transmissão de dados entre microcomputadores, utilizando como saída dos dados o monitor de vídeo tipo CRT (Tubo Catódico de Raios) do microcomputador transmissor.

O modelo realiza uma transmissão de dados do tipo texto, onde o microcomputador transmissor varia a intensidade de luz emitida pelo seu monitor CRT, de acordo com os bits que compõe o texto a ser transmitido, permitindo desta forma que uma interface, especialmente projetada com o fim de captar estas variações, possa decodificar o texto original e escrever o mesmo na porta serial do computador receptor.

São desenvolvidos neste projeto, além da interface e seu software embarcado, os softwares transmissor e receptor, ambos fundamentais para o estabelecimento do enlace de comunicação entre os microcomputadores envolvidos.

---

[1] ViaPalm (http://www.viapalm.com.br/historia.htm)
[2] Wikipedia (http://pt.wikipedia.org/wiki/PDA)

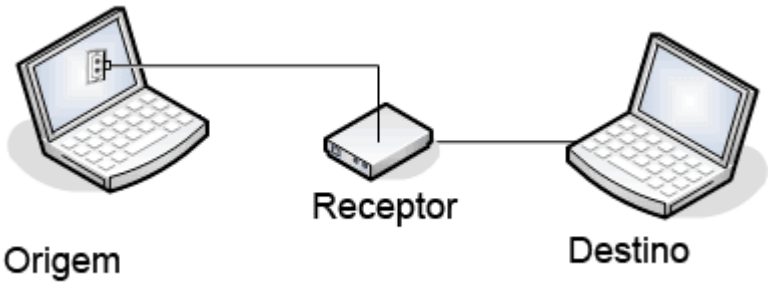


Figura 1.1 - Modelo de Transmissão

1.3. Motivação

Ao tomar contato com o projeto apresentado pelo formando Thiago Mitsuka no fim do 2º semestre de 2004 neste mesmo Centro Universitário de Brasília, intitulado “Transmissão Alternativa de Dados”, vislumbrei várias possibilidades de melhorias e aperfeiçoamentos.

A principal motivação do projeto original é permitir que qualquer dispositivo que possua uma interface receptora, como a desenvolvida, possa captar dados de um monitor de vídeo CRT, evitando assim a necessidade de conexão física em qualquer porta de comunicação do microcomputador transmissor.

Mas quais seriam os benefícios desse tipo de transmissão? Imagine que um usuário possua um telefone celular. Agora imagine que esse usuário deseje receber um arquivo da Internet (como um toque polifônico, uma imagem ou até mesmo um jogo) em seu celular, através de um computador público. Suponha também que, por questões de segurança, as interfaces desse computador, como porta serial, Infravermelho, porta paralela ou USB, estejam desabilitadas ou inacessíveis. Então, o usuário não poderia efetuar a transferência do arquivo para seu celular. Porém, com o uso da interface de conversão, o usuário poderia receber os dados em seu telefone celular tendo o monitor de vídeo como transmissor (Mitsuka 2004).

Dado este cenário, esta pessoa poderia realizar a recepção de um toque polifônico, uma imagem, um jogo ou até mesmo os dados de seu organizador pessoal sem a necessidade de qualquer porta de comunicação, bastando apenas posicionar seu aparelho de telefone celular em frente ao monitor de vídeo do PC.

O trabalho original contava com a transmissão dos dados utilizando dois níveis de transmissão (claro e escuro), possibilitando assim uma transmissão de um bit por variação entre estes dois níveis. A adoção de apenas dois níveis de transmissão limita a velocidade da mesma à freqüência vertical do monitor CRT.

A fim de tornar a transmissão mais rápida, o projeto aqui apresentado, implementa uma transmissão baseada em 4 níveis. Tal evolução faz com que a velocidade de transmissão dobre.

Além disso, o projeto anterior não apresenta qualquer técnica para que se torne a transmissão mais confiável. A transmissão é realizada bit a bit sem qualquer adição de bits extras objetivando a detecção de erros.

Pensando nisto, foi adicionado ao projeto uma técnica de detecção de erros. A técnica da paridade combinada implementada neste projeto, além de detectar erros de até 2 bits, ainda corrige erros de 1 bit por bloco de dados.

A principal vantagem do projeto anterior aproveitada no projeto aqui apresentado foi a utilização das metodologias *Double Buffering* e *Page Flipping* no Software Transmissor.

1.4. Estrutura do Trabalho

Além deste capítulo introdutório, este trabalho está estruturado em 7 capítulos assim distribuídos:

No **Capítulo 2** é apresentada uma revisão bibliográfica onde são abordados os conceitos essenciais no entendimento do protótipo e seus softwares envolvidos.

No **Capítulo 3** são detalhados os módulos envolvidos na comunicação, seus principais componentes e o papel de cada um no sistema como um todo.

No **Capítulo 4** são apresentados, em detalhes, a construção do Software Transmissor, as tecnologias utilizadas e os motivos de escolha de cada uma.

No **Capítulo 5** são apresentadas a API de comunicação serial da Sun Microsystems, as classes desta API que foram utilizadas no projeto e exemplos de utilização de cada uma.

No **Capítulo 6** é apresentado o detalhamento do hardware desenvolvido, seus principais componentes e características técnicas do mesmo, bem como todo o diagrama do circuito criado.

No **Capítulo 7** é apresentada toda a construção do Software Embarcado no microcontrolador, responsável por decodificar a informação recebida e retransmiti-la ao computador receptor.

Por fim, na **Conclusão** são apresentadas as considerações finais sobre o trabalho, as principais contribuições, os resultados obtidos, as dificuldades encontradas e as sugestões para trabalhos futuros.

2. REVISÂO BIBLIOGRÁFICA

2.1. O Sistema de Cores RGB[3]

RGB é a abreviatura do sistema de cores aditivas formado por Vermelho (Red), Azul (Blue) e Verde (Green). É utilizado em meios que têm fundo escuro, como monitores e televisores.

É importante notar que o modelo RGB não define qual é exatamente o significado de “vermelho”, “azul” e “verde”, de modo que os mesmo valores RGB possam descrever cores visivelmente diferentes em dispositivos que empregam este modelo de cor.

Uma aplicação comum do modelo de cores RGB é a exibição das cores em um tubo de raios catódicos, exibição em cristal líquido ou em plasma, tal como um monitor de televisão ou computador.

Cada pixel na tela pode ser representado na memória do computador como valores independentes para o “vermelho”, o “verde” e o “azul”. Estes valores são convertidos em intensidades e emitidos à exibição. Usando a combinação apropriada de intensidades de “vermelho”, “verde” e “azul” a tela pode produzir muitas cores entre os níveis “preto” e “branco”.

Os monitores para computador mais comuns utilizam um total de 24 bits de informação para a representação de cada pixel (conhecidos como *bits per pixels* ou bpp). Isto corresponde à utilização de 8 bits para cada cor, possibilitando um intervalo de 256 valores possíveis para cada cor. Com este sistema é possível representar, aproximadamente, 16,7 milhões de cores, sendo que o olho humano pode distinguir cerca de 10 milhões de cores (este valor varia de pessoa para pessoa, dependendo das condições do olho e da idade da pessoa).

2.2. A não-linearidade do modelo RGB

A intensidade de saída de cores em dispositivos de exibição não é proporcional aos valores R, G e B.

[3] Tradução livre e resumida do texto publicado na Wikipedia (http://en.wikipedia.org/wiki/RGB)

Isto quer dizer que, por exemplo, mesmo sendo o valor 127 muito próximo do valor médio entre 0 e 255, a intensidade da luz exibida (127, 127, 127) é normalmente 18% da intensidade luminosa quando exibindo (255, 255, 255), ao invés de 50% conforme esperado.

2.3. A Escala de Cinza[4]

Em computação, imagens exibidas em *grayscale*, ou escala de cinza, são imagens onde o valor de cada pixel é uma única amostra. No sistema RGB isto quer dizer que a intensidade de cada uma das três cores (*red, green e blue*) é a mesma.

As imagens geradas nesta escala são compostas por máscaras de cinza que variam do preto (a intensidade mais fraca) ao branco (a intensidade mais forte).

As imagens da escala de cinza são diferentes das imagens em preto-e-branco, que no contexto da imagem latente do computador são imagens com somente duas cores, o preto e o branco; já as imagens em *grayscale* possuem muitas cores entre estas duas cores.

2.4. A API Direct X[5]

DirectX é uma API (Application Programming Interface) que fornece um conjunto de funcionalidades de grande utilidade para o desenvolvimento de jogos, sendo considerada padrão na plataforma Windows. Apresenta uma interface de alto nível, permitindo acesso aos recursos avançados de hardware das placas gráficas 3D, sem a necessidade de conhecer as particularidades de cada hardware. O DirectX proporciona isso mediante a criação de uma camada intermediária de software que traduz comandos genéricos nos comandos específicos para cada dispositivo. Essa API também suporta as tecnologias USB, FireWire, AGP e MMX.

Na prática, a tecnologia Directx tira partido dos recursos da aceleração gráfica do Hardware, o que resulta em ganho de performance em aplicações multimídia.

São partes integrantes do DirectX:

- DirectDraw - acesso direto à placa de vídeo para gráficos 2D

[4] Tradução livre e resumida do texto publicado na Wikipedia (http://en.wikipedia.org/wiki/RGB)
[5] Tradução livre e resumida do texto publicado na MSDN (http://msdn.microsoft.com/directx)

- Direct3D - acesso direto à memória de vídeo para gráficos 3D
- DirectSound - acesso direto à placa de som
- DirectPlay - acesso direto a rede
- Direct Input - acesso direto a joysticks

2.5. Técnica de Double Buffering[6]

Qualquer programa passa por toda “burocracia” do sistema operacional. Essa “burocracia” diminui a performance do programa. Para resolver esse problema, o software implementado utiliza um método que permite que o programa desenhe diretamente na tela do monitor, tornando-o mais eficiente. A técnica utilizada é o modo de tela cheia exclusiva, que permite também o acesso às funções de configurações de vídeo, tais como: resolução da imagem, número de bits por pixel e taxa de atualização do monitor.

Supondo que o programa desenhe pixel por pixel ou linha por linha na tela do monitor, o mesmo não teria performance suficiente para fazê-lo na mesma velocidade em que a tela é atualizada pelo monitor. A solução encontrada é a utilização de uma técnica muito aplicada por programadores de jogos, chamada *double-buffering* (armazenamento duplo).

A superfície da tela é normalmente conhecida como *superfície primária* e a imagem não exibida, usada pelo *double-buffering,* é normalmente conhecida como *back buffer*. Portanto, o *double-buffering* trata-se de uma técnica que permite a manipulação de uma imagem no *back buffer* enquanto a *superfície primária* está sendo exibida na tela.

Uma superfície primária é usualmente manipulada através de um objeto gráfico, uma operação de manipulação direta à memória da tela. O ato de copiar o conteúdo de uma superfície para outra é conhecido como *blitting* ou simplesmente *blt*, como representado por intermédio Figura 2.1:

[6] Texto adaptado do site da Sun (http://java.sun.com/docs/books/tutorial/extra/fullscreen/doublebuf.html)

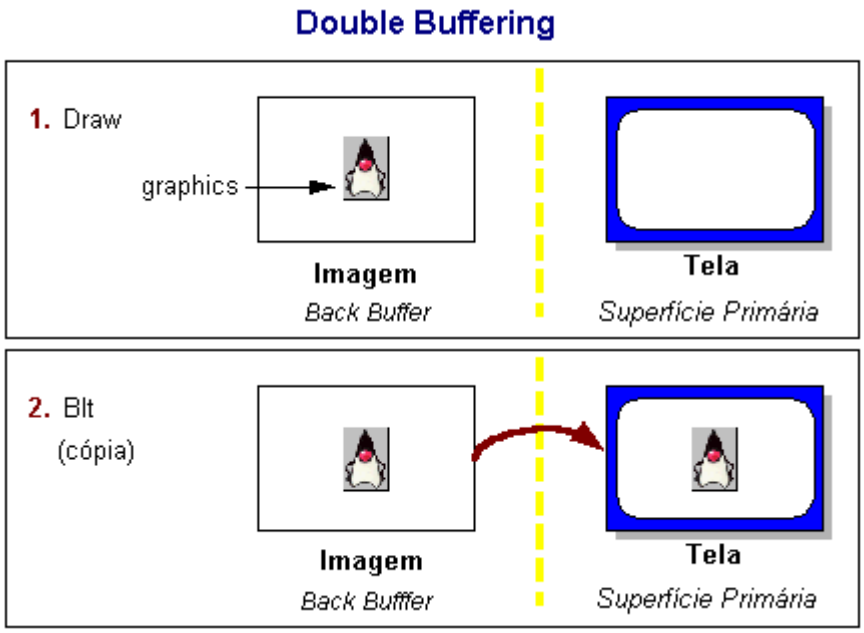


Figura 2.1 - Técnica de Double-Buffering

2.6. Técnica de Page-flipping[7]

Outra técnica que pode ser utilizada com o modo de tela cheia exclusiva é o *page-flipping*, que também é uma forma de *double-buffering*. Muitas placas gráficas têm a noção de ponteiro de vídeo, que é simplesmente um endereço na memória de vídeo. Este ponteiro diz para a placa gráfica onde ela deve procurar pelo conteúdo de vídeo a ser exibido no próximo ciclo de *refresh.* Isso permite que um desenho criado no *back buffer* seja exibido no próximo ciclo, ao invés de ser copiado para a superfície primária.

[7] Texto adaptado do site da Sun (http://java.sun.com/docs/books/tutorial/extra/fullscreen/doublebuf.html)

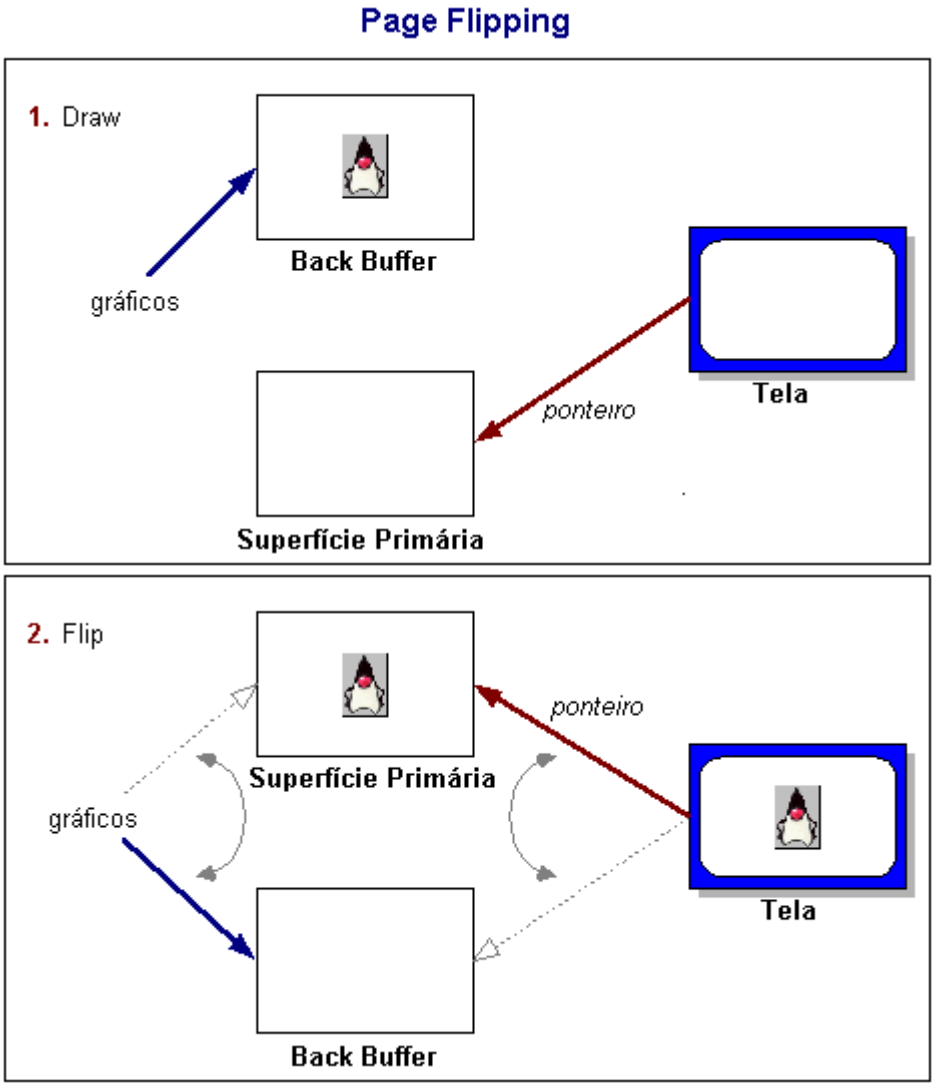


Figura 2.2 - Técnica de Page-Fipping

Em algumas situações, é mais vantajosa a utilização de múltiplos back fuffers. Esta técnica é utilizada principalmente quando o tempo de desenho é maior que o tempo de *refresh*.

Como a técnica de *page-flipping* é mais rápida, ela é utilizada na implementação do Software Transmissor.

2.7. Paridade de Caractere

Paridade de caractere é uma técnica de detecção de erros que consiste em acrescentar um bit extra ao caractere, provocando um número par (ou ímpar) em relação ao número total de bits “1”s.

Tabela 2.1 - Exemplo de Cálculo de Paridade

Caractere	ASCII	Binário	Paridade Par	Paridade Impar
A	41	1000001	1000001 **0**	1000001 **1**
B	42	1000010	1000010 **0**	1000010 **1**
C	43	1000011	1000011 **1**	1000011 **0**

O equipamento transmissor calcula o bit de paridade para cada caractere transmitido. O receptor calcula um novo bit de paridade em cima dos bits recebidos e compara este bit com aquele enviado pelo transmissor. Se forem iguais, a transmissão é considerada correta; se não, haverá a necessidade de retransmissão do caractere. Caso haja um número par de bits com erro, a técnica não consegue detectar, pois a verificação de bits “1”s do caractere recebido permanecerá par ou ímpar, de acordo com o método, satisfazendo ao bit de paridade. Entretanto, a prática mostra que a maioria dos erros são erros simples (Alves 1991).

No processo de paridade combinada, a paridade de caractere é denominada paridade vertical ou VRC (*Vertical Redundancy Checking*).

2.8. Paridade Combinada (ou Biparidade)

A paridade longitudinal (ou paridade horizontal), denominada LRC (*Longitudinal Redundancy Checking*), consiste em acrescentar um bit de paridade para cada nível (posição) de bit dos bytes de um bloco, inclusive para o nível de bit de paridade vertical (VRC) (Tarouco 1985).

À utilização desses dois métodos (VRC + LRC) em conjunto, dá-se o nome de paridade combinada.

A paridade combinada possibilita a formação de um BCC (*Block Check Character*), obtido de todos os bits formados pela paridade horizontal, sendo implementado nos equipamentos a partir da operação “ou exclusivo” dos caracteres a serem transmitidos.

Quando a mensagem chegar a seu destino, o equipamento receptor calcula um novo BCC a partir dos bits recebidos e compara-o com o BCC recebido ao final da mensagem (que foi calculado pelo equipamento transmissor). Ocorrendo a igualdade entre esses BCC’s, a mensagem recebida será considerada correta, sendo aceita e processada. Caso contrário, a mensagem será considerada incorreta, pedindo-se, assim, a retransmissão da mensagem (Tarouco 1985).

Alguns terminais têm a capacidade de não somente detectar o erro, mas também de corrigi-lo, quando o bit errado for devidamente identificado.

A identificação do bit a ser corrigido é possível porque quando um bit errado é transmitido este altera tanto o byte de paridade VRC quanto o byte de paridade LRC.

Ao comparar as paridades VRC e LRC calculadas com aquelas recebidas, o terminal é capaz de identificar a posição dos bits em cada um destes bytes. O cruzamento entre estas duas posições (considerando LRC como um índice de linha e VRC como um índice de coluna) representa a posição exata do bit a ser corrigido dentro do bloco.

Tabela 2.2 - Exemplo de Paridade Combinada

Caractere	1º	2º	3º	4º	5º	6º	7º	Paridade (LRC)
Paridade (VRC)	**1**	**0**	**0**	**(1)**	**1**	**1**	**0**	**(0)**
bit 0	1	1	0	1	0	1	1	**1**
bit 1	0	1	0	1	0	1	1	**0**
bit 2	1	0	0	1	0	0	1	**1**
bit 3	1	0	0	1	1	0	0	**1**
bit 4	1	0	1	**(1)**	1	1	0	**(1)**
bit 5	1	0	1	0	0	1	0	**1**
bit 6	0	0	0	0	1	1	1	**1**

Tabela 2.3 - Exemplo de Paridade Combinada com Erro no 4º bit do 4º caractere

Caractere	1º	2º	3º	4º	5º	6º	7º	Paridade (LRC)
Paridade (VRC)	**1**	**0**	**0**	**(0)**	**1**	**1**	**0**	**(1)**
bit 0	1	1	0	1	0	1	1	**1**
bit 1	0	1	0	1	0	1	1	**0**
bit 2	1	0	0	1	0	0	1	**1**
bit 3	1	0	0	1	1	0	0	**1**
bit 4	1	0	1	**(0)**	1	1	0	**(0)**
bit 5	1	0	1	0	0	1	0	**1**
bit 6	0	0	0	0	1	1	1	**1**

2.9. Comunicação com RS-232

A interface serial mais comumente utilizada nos microcomputadores é a RS-232. Originalmente foi criada para facilitar a interconexão dos terminais e dos equipamentos de comunicação de dados.

Na interface RS-232 os pinos mais comumente utilizados são três, sendo um com a função de enviar e outro com a função de receber os dados. Uns poucos pinos no conector são absolutamente previsíveis conforme mostrado na Tabela 2.4.

Tabela 2.4 - Pinos para Comunicação Serial

Pino	Função
Pino 2	Pino para transmissão
Pino 3	Pino para recepção
Pino 5	Circuito comum

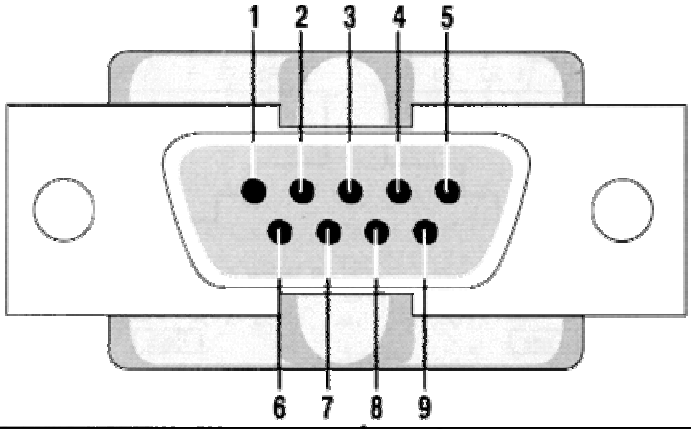
No que diz respeito às características elétricas, o padrão RS-232 define atualmente 4 níveis lógicos. As entradas têm definições diferentes dos dados. Para as saídas, o sinal é considerado na condição de estado “1”, quando a tensão no circuito de transferência, medida no ponto de interface é menor que –5V e maior que –15V, com relação ao circuito de referência (terra). O sinal é considerado na condição de estado “0”, quando a tensão for maior que +5V e menor que +15V, também com relação ao circuito de referência (terra) (Tafner 1996).

Para as entradas, o sinal é considerado em condição de marca, ou estado “1”, quando a tensão no circuito de transferência, medida no ponto de interface, é menor que –3V e maior que –15V, com relação ao circuito terra. O sinal é considerado na condição de espaço ou estado “0”, quando a tensão for maior que +3V e menor que +15V, também com relação ao circuito terra. A região compreendida entre –3V e +3v é definida como região de transição (Tafner 1996).

Durante a transmissão dos dados, a condição de marca é usada para discriminar o estado binário “1”, e a condição de espaço é usada para discriminar o estado binário “0” (Tafner 1996).

Muitas aplicações utilizam a conexão direta via cabo para trocar informações entre dois computadores. As utilidades vão desde o simples compartilhamento de arquivos sem a utilização de placas de rede até o jogo entre dois adversários em

computadores diferentes. Cada computador dispõe de pelo menos uma porta serial, o conector pode ser um DB9 ou um DB25.



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Figura 2.3 - Conector DB9 e seus Respectivos Pinos

2.10. USART

USART é um circuito integrado que controla a porta serial do microcomputador. A principal característica não é apenas a de converter os dados de paralelo para serial, mas também acrescentar alguns bits de controle como bits de início, bits de terminação e bits de paridade para o controle de erro.

Ela se encarrega de fazer com que o computador receba corretamente os bits de informação que são transmitidos pela entrada da porta serial, e também de fazer com que os dados de informação que saem do micro estejam corretos. Por exemplo, uma função da USART é assegurar que os bits, na transmissão serial, estejam sempre com o mesmo espaçamento no tempo, de forma a manter o sincronismo durante a comunicação (Soares 1995).

3. CARACTERÍSTICA DOS MÓDULOS ENVOLVIDOS NA COMUNICAÇÃO

Um sistema de transmissão completa inclui geralmente: um transmissor, um meio de transmissão através do qual a informação é transmitida; e um receptor, que produz uma réplica identificável da informação de entrada (Nascimento 2000).

Na Figura 3.1 é demonstrado, por intermédio de diagramas de blocos, todos os meios envolvidos no sistema de comunicação apresentados neste projeto:

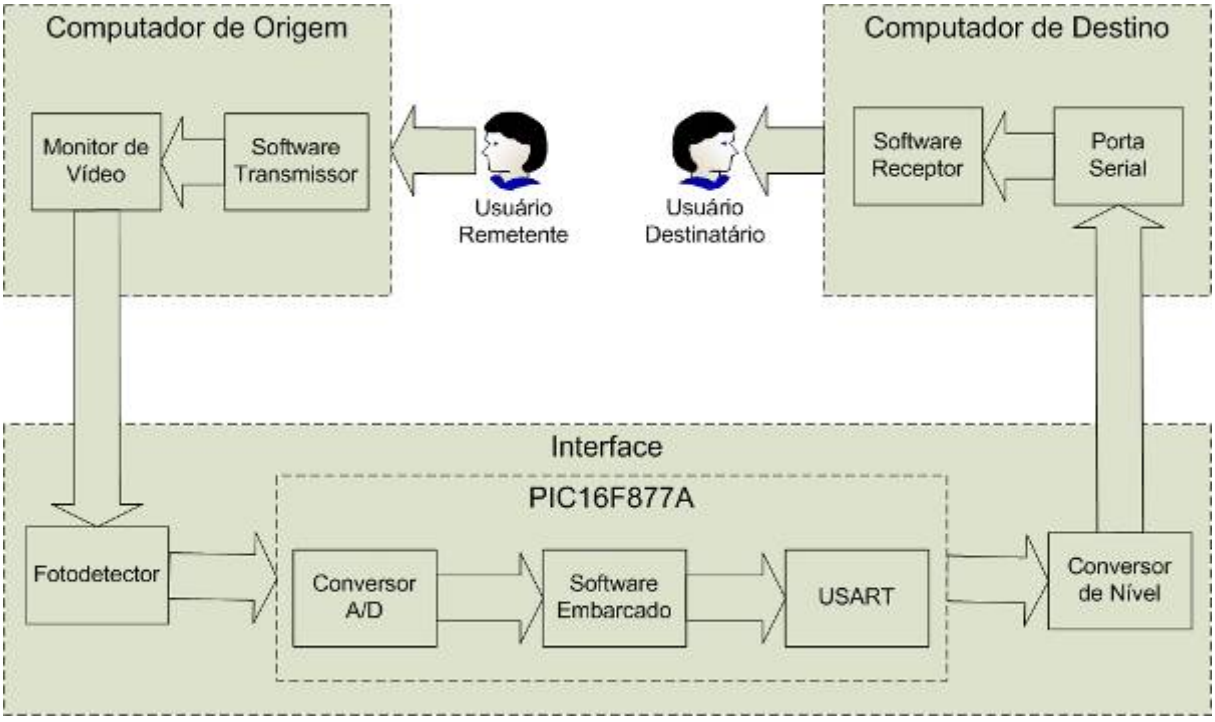


Figura 3.1 - Diagrama de Blocos do Projeto

3.1. Transmissor

Na maioria dos sistemas, a transmissão de informação é estreitamente relacionada com a modulação, isto é, consiste na colocação de dados digitais num sinal analógico com a variação de um determinado sinal denominado portadora. Como fenômeno físico que é, um sinal possui diversas grandezas físicas mensuráveis. Se o emissor produzir variações nestas grandezas de modo a traduzir a informação a transmitir, então o receptor pode detectar estas variações e obter a informação que foi transmitida (Nascimento 2000).

Modulação é um processo pelo qual são modificadas uma ou mais características de uma onda denominada *portadora*, segundo um sinal *modulante* (informação que se deseja transportar pelo meio; no caso de comunicação de dados, é o sinal digital). A modulação pode ser feita variando a amplitude, freqüência ou fase da onda portadora, isoladamente ou conjuntamente. A informação impõe o modo como vai ser modificada a portadora. Ao se analisar, na recepção, as modificações sofridas pela portadora, pode-se recuperar a informação digital. É por isso que se diz que a portadora transporta a informação (Silveira 91).

3.1.1. O monitor de vídeo

No sistema de comunicação de dados utilizados no projeto, a placa de vídeo junto com o monitor de vídeo, é responsável pela modulação do sinal analógico utilizado na transmissão de dados.

O que permite a modulação do sinal de transmissão a partir do monitor como transmissor de dados é a possibilidade de manipulação da intensidade dos feixes de elétrons pelo sistema a partir de um aplicativo. No caso desse projeto, foi implementado um *software* específico para essa função.

O CRT é um dispositivo analógico, constituído por uma tela de vidro recoberta em seu lado interno por uma camada de substância (fósforo) que tem a propriedade de tornar-se luminosa ao ser bombardeada por um feixe de elétrons. Um canhão de elétrons, situado na parte traseira da tela de vidro do tubo, direciona o feixe em um traçado formado por linhas horizontais, de cima para baixo. Ao ser alimentado pelo sinal de vídeo, um circuito faz com que o feixe seja mais ou menos intenso, conforme o ponto correspondente deva ser mais ou menos luminoso (Torres 98).

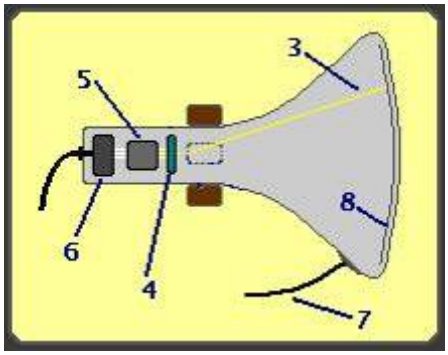


Figura 3.2 - O tubo de raios catódicos

É apresentado por intermédio da Figura 3.2 um tubo de imagem esquematizado. Dentro do tubo, feito de vidro, existe um razoável grau de vácuo, daí o peso do mesmo - o vidro precisa ser espesso, principalmente em sua parte frontal, para suportar a pressão atmosférica sem risco de implodir devido ao vácuo em seu interior. O canhão de elétrons é formado pelo cátodo (6), onde os mesmos são gerados. O CRT utiliza alta tensão para gerar o fluxo de elétrons, cerca de 200 vezes maior do que a tensão da corrente elétrica que alimenta o aparelho. A seguir, estes elétrons são acelerados através de um dispositivo situado logo após o cátodo, indicado em (5). São então focados (4) para formar o feixe concentrado. E é este feixe de elétrons que atinge a superfície interna do tubo (3), recoberto pela camada de fósforo (8): o ponto atingido pelo feixe torna-se luminoso, podendo ser visto do lado de fora do tubo. Para que os elétrons sejam atraídos para a tela, a mesma é energizada de maneira oposta ao cátodo, no ponto indicado por (7), o ânodo (Torres 98).

O feixe de elétrons deve ser direcionado na superfície frontal interna do tubo de forma a descrever uma trajetória em forma de linhas horizontais, uma abaixo da outra. Ao final de cada linha horizontal, um código específico no sinal indica que o feixe chegou ao final do desenho da linha em que está, e que deve descer um pouco e retornar para o outro extremo, para iniciar o desenho da próxima linha. Para que o feixe possa ser direcionado para a esquerda e para a direita e também para cima e para baixo, ao invés de permanecer fixo em um ponto central da superfície frontal do tubo, existem potentes ímãs instalados em meio a sua trajetória. Estes ímãs, na verdade eletro-ímãs (ímãs cuja capacidade de atrair pode ser variada em função da variação da intensidade de corrente elétrica aplicada aos mesmos), atraem o feixe

em sua direção com maior ou menor intensidade, desviando assim sua trajetória (Torres 98).

O tubo possui 4 eletro-ímãs, dois localizados nas partes inferior e superior do tubo (1) para controlar o movimento vertical do feixe (para cima / para baixo) e dois outros localizados nas suas laterais (2) para controlar o movimento horizontal (para os lados) (Torres 98).

O circuito eletrônico lê então os pulsos existentes no sinal, que indicam início/término do desenho de cada linha, transformando-os em variação de intensidade no campo magnético dos eletro-ímãs. Assim, as linhas vão sendo desenhadas na superfície interna do tubo. Ao mesmo tempo, o circuito eletrônico lê a intensidade do sinal a todo momento, controlando a intensidade do feixe emitido pelo canhão. Assim, as nuances da imagem (pontos mais claros, mais escuros) são formadas, completando-se o processo de formação da imagem (traçado + intensidade) (Torres 98).

No tubo de imagem preto e branco a tela de vidro é recoberta por uma camada uniforme de fósforo e existe um só canhão de elétrons. No tubo colorido não existe uma camada uniforme e sim uma camada com milhares de minúsculos círculos ou segmentos coloridos, agrupados sequencialmente nas 3 cores básicas (RGB) do sinal de vídeo. E, ao invés de um só canhão de elétrons, existem 3, emitindo 3 feixes distintos ou então um só, emitindo um feixe único a partir do qual são separados a seguir os 3 feixes (Torres 98).

Cada um dos 3 feixes atinge o mesmo tipo de pontos / segmentos coloridos, ou seja, um dos feixes atinge somente os pontos vermelhos, outro somente os verdes e outro somente os azuis. Para conseguir-se isso, e evitar-se que o feixe ao deslocar-se em sua trajetória no desenho das linhas passe sobre pontos / segmentos das outras cores e os ativem, é acrescentada próximo à superfície interna do tubo (a cerca de 1,5 cm de distância) uma máscara metálica com milhares de minúsculos orifícios. Esta máscara é ajustada com muita precisão, de modo que ao deslocar-se horizontalmente o feixe azul por exemplo seja obstruído ao passar sobre os pontos vermelhos e verdes (Torres 98).

3.1.2. O Software Transmissor

O microprocessador não é capaz de criar imagens, somente manipular dados. Portanto o microprocessador não gera imagens. O que ele na verdade faz é definir como será a imagem e enviar os dados relativos a essa imagem a uma interface capaz de gerar imagens – a interface de vídeo. A interface de vídeo, por sua vez, é conectada a um dispositivo capaz de apresentar as imagens por ela geradas – o monitor de vídeo (Torres 98).

Quando o processador quer escrever dados na tela (desenhar janelas, por exemplo), ele escreve os dados em um lugar chamado memória de vídeo, que está na interface de vídeo. O controlador da interface de vídeo converte os dados presentes na memória de vídeo em sinais eletrônicos compatíveis com o monitor de vídeo (Torres 98).

O Software Transmissor é o responsável em transformar os dados a serem enviados em sinais ópticos exibidos na tela do monitor. Sinais que posteriormente serão capturados pela interface e enviados no padrão serial para o Software Receptor.

O software basicamente converte um dado ou um byte, em um valor binário, calcula a paridade e realiza, por intermédio de acesso direto ao hardware gráfico, a alternância da imagem exibida no monitor de vídeo de forma que a interface possa detectar.

Para acesso direto ao hardware gráfico, é necessária a utilização de alguma API específica para tal fim. As mais consagradas são a OpenGL e a Microsoft Direct X.

Sendo a imagem gerada um retângulo, onde cada par de bits está associado à cor que este é pintado, tal tarefa é trivial para estas API’s, pois utiliza apenas uma de suas funções básicas. Neste caso, não há diferenças entre o uso de uma ou outra tecnologia, ficando a cargo da escolha apenas a facilidade de uso, a integração com o sistema operacional e a afinidade do autor do projeto com a mesma.

O Software Transmissor foi desenvolvido em C++ devido à sua facilidade em acessar diretamente a API do sistema operacional Microsoft Windows e fornecer

uma performance muito superior a qualquer outra linguagem de programação, fato determinante no projeto.

Para a modulação dos dados transmitidos, foi escolhida a utilização de cores na escala de cinza, demonstrada anteriormente no Capítulo 2.

Como a transmissão de dados se dará em 4 níveis, foi determinada utilização das cores Preto (0, 0, 0), Branco (255, 255, 255) e de mais 2 (duas) cores intermediárias, a fim de representar os pares de bits 00, 01, 10 e 11.

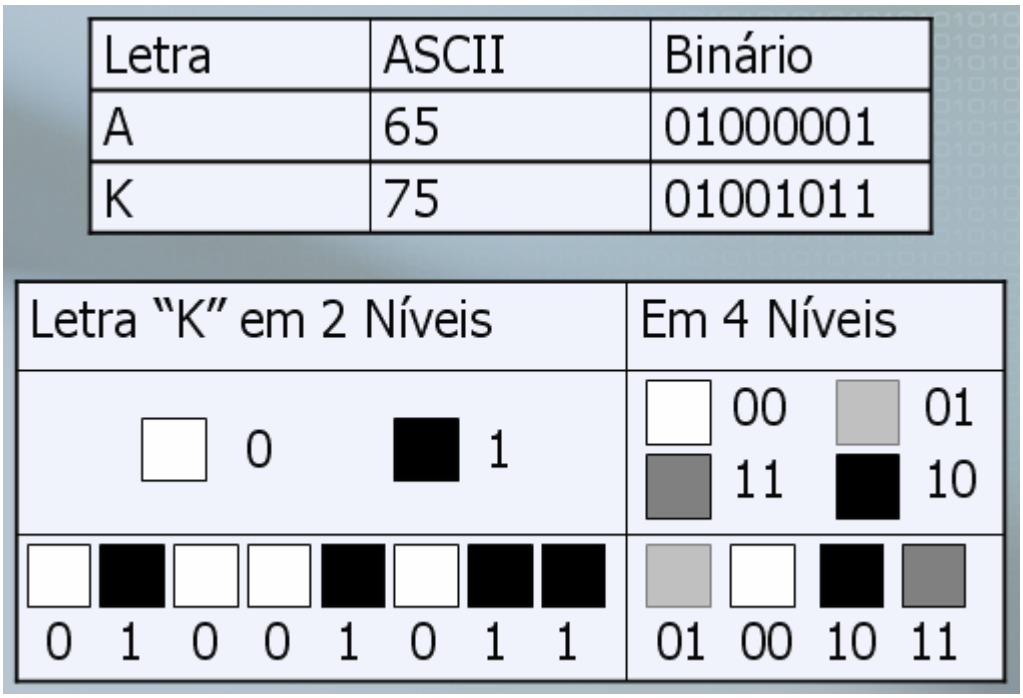


Figura 3.3 - Diferença entre as transmissões em 2 níveis e 4 níveis

3.2. A Interface

A interface de transmissão funciona basicamente como um conversor de sinal analógico para digital, convertendo os sinais ópticos emitidos pelo monitor de vídeo em sinais digitais (nível lógico “0” e nível lógico “1”). Posteriormente, esses sinais convertidos são transmitidos para o dispositivo receptor no padrão serial RS 232. Portanto, a interface de transmissão divide-se em três módulos descritos na Figura 3.4.

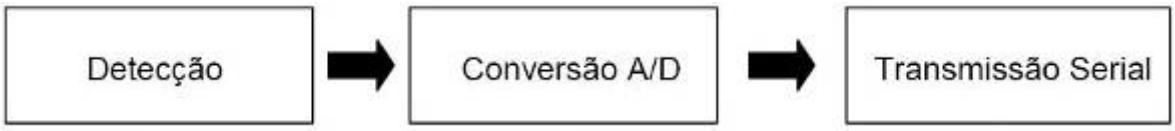


Figura 3.4 - Diagrama de Blocos da Interface

O módulo de detecção é realizado por um transdutor óptico, enquanto a conversão dos sinais A/D e a transmissão no padrão serial são realizadas por um único dispositivo, um microcontrolador PIC.

3.2.1. O Fotodetector

Para transformar em sinais elétricos as informações que se apresentam originalmente na forma de sons ou imagens, os sistemas de comunicação utilizam transdutores eletroacústicos, eletromecânicos ou optoeletrônicos. O fotodetector, utilizado para transformar luz em sinais elétricos, é um exemplo de transdutor optoelétrico utilizado no projeto (Nascimento 2000).

O fotodetector tem um papel importante na transmissão: fazer a tradução do sinal óptico gerado pelo monitor de vídeo em um sinal elétrico para posteriormente ser transmitido para o receptor da mensagem. Foram testados dois tipos de fotodetectores: o fotodiodo e o sensor LDR (Light Dependent Resistor).

Ambos os fotodetectores, quando submetidos a variações da escala de cinza de 0 a 255, forneceram uma variação não-linear (devido à particularidade do modelo RGB), sendo que o fotodiodo apresentou uma maior dispersão conforme demonstrado na Figura 3.5.

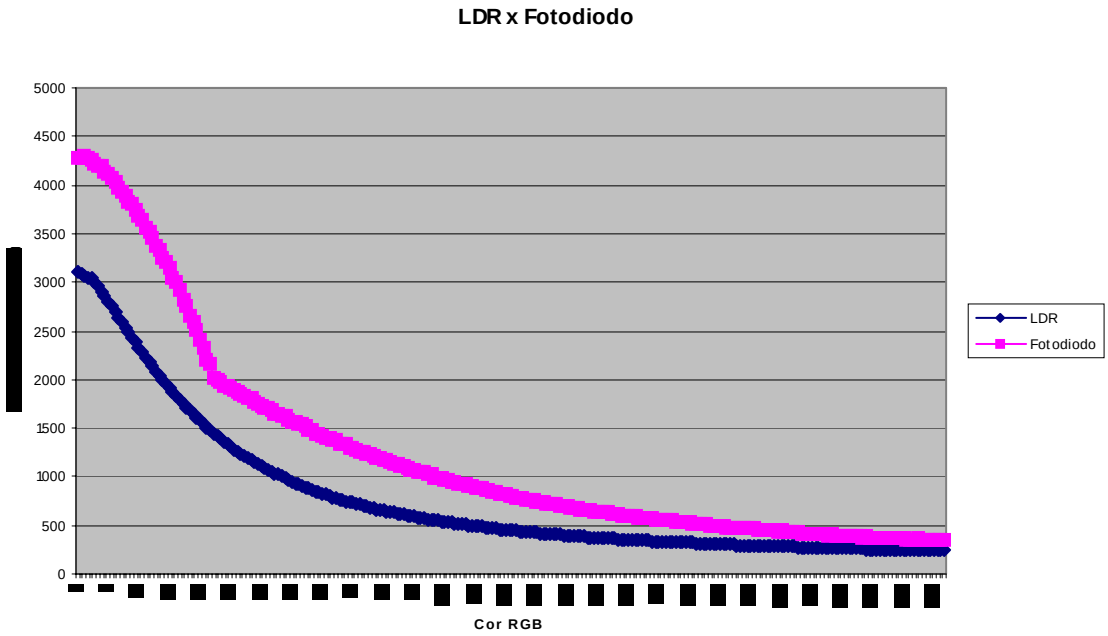


Figura 3.5 - Gráfico de comparação LDR x Fotodiodo

Além do fator dispersão, outro motivo que levou à escolha do fotodiodo como o dispositivo fotodetector utilizado no projeto é o fato do sensor LDR possuir um tempo de resposta entre 10 ms e 100 ms, enquanto o fotodiodo possui um tempo de resposta na ordem de nanossegundos.

O fotodetector, mais especificamente o fotodiodo, possui uma sensibilidade suficiente para capturar o momento em que o feixe eletrônico passa por um ponto na tela do monitor. Tal sensibilidade tornou-se um fator delicado no projeto, pois o fotodiodo acaba sendo sensível à luz ambiente que, em monitores que não possuam um sistema anti-reflexivo, interfere drasticamente na transmissão.

3.2.2. O Microcontrolador

Os MCU´s (Micro Controler Unit) ou microcontroladores, ao contrário dos microprocessadores, são dispositivos mais simples, com memórias RAM (Random Access Memory) e ROM (Read Only Memory) internas, oscilador interno de clock, I/O interno, entre outros, sendo por isso chamados muitas vezes de computadores em um único chip. Tais características tornam mais simples o projeto de dispositivos

inteligentes, pois os MCU´s raramente necessitam de CI´s externos para funcionar, o que contribui para a diminuição de custos e tamanho.

Os microcontroladores PIC são uma família de dispositivos fabricados pela Microchip. Utilizando uma arquitetura RISC (Reduced Instruction Set Computer), com freqüência de clock de até 40MHz, até 2024k palavras de memória de programa e até 3968 bytes de memória RAM. Além disso, podem ser encontrados com diversos periféricos internos, como: até quatro temporizadores/contadores, memória EEPROM (Electrically Erasable Programmable Read Only Memory) interna, gerador/comparador/amostrador PWM, conversores A/D de até 12 bits, interface de barramento CAN, I2C, SPI, entre outros (Pereira 2004).

O dispositivo utilizado na construção da interface é o PIC16F877A. Este dispositivo é membro e líder de uma família intermediária (Mid-range) da Microchip.

Suas principais características são:

- Baixo custo;
- Facilidade de programação;
- Grande diversidade de periféricos internos;
- Memória de programa do tipo FLASH;
- Excelente velocidade de execução.

Além disso, pode-se também destacar as seguintes especificações:

- 8K x 14 bits de memória FLASH;
- 368 x 8 bits de memória SRAM disponíveis para o usuário;
- 256 x 8 bits de memória EEPROM interna;
- Pilha com 8 níveis;
- 5 portas de I/O (entrada ou saída);
- 1 timer/contador de 8 bits;
- 1 timer/contador de 16 bits;
- 1 timer de 8 bits;
- 2 canal PWM com captura e amostragem (CCP) de 10 bits;
- 1 canal de comunicação USART serial;
- 1 conversor A/D interno de 10 bits e 8 canais;
- 2 comparadores analógicos com referência interna programável de tensão;

- 1 timer watchdog;
- 15 fontes de interrupção independentes;
- Capacidade de corrente de 25 mA por pino de I/O;
- 35 instruções;
- Freqüência de operação de desde DC (0 Hz) até 20 MHz;
- Oscilador interno;
- Tensão de operação entre 3.0 a 5.5V;
- Compatível pino a pino com outros microcontroladores de 40 pinos da linha PIC16FXXX.

A presença de todos estes dispositivos em um espaço extremamente pequeno, dá ao projetista ampla gama de trabalho e enorme vantagem em usar um sistema microprocessado, onde em pouco tempo e com poucos componentes externos podemos fazer o que seria trabalhoso fazer com circuitos tradicionais.

Foram esses e outros motivos que levaram à escolha do microcontrolador PIC.

Na Figura 3.6 é mostrada a pinagem do PIC16F877A:

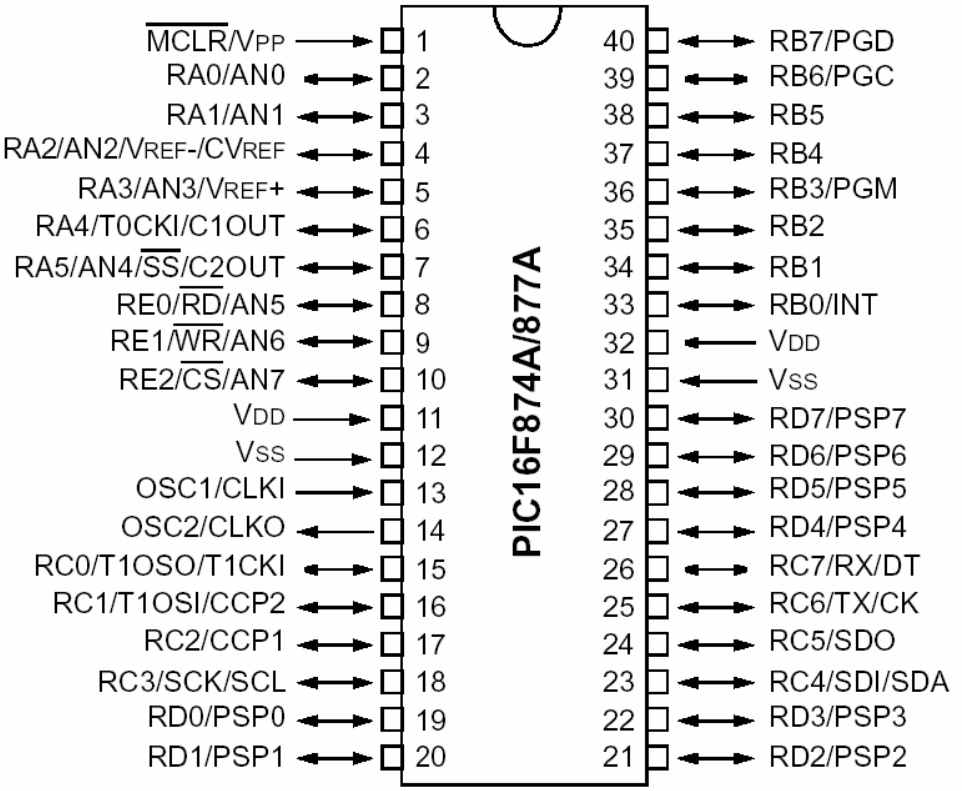


Figura 3.6 - Pinagem do PIC16F877A

3.2.3. O Software Embarcado

Atualmente, a maioria dos microcontroladores disponíveis no mercado conta com compiladores de linguagem C para o desenvolvimento de software. O uso de C permite a construção de programas e aplicativos muito mais complexos do que se fosse usado apenas o *Assembly*.

Além disso, o desenvolvimento em C permite uma grande velocidade na criação de novos projetos, devido às facilidades de programação oferecidas pela linguagem e também à sua portabilidade, o que permite adaptar programas de um sistema para outro com um mínimo de esforço.

Outro aspecto favorável da utilização da linguagem C é a sua capacidade de otimização de código. Otimização de código, no jargão dos compiladores, é a medida do grau de inteligência com que o compilador traduz um programa em C para o código de máquina. Quanto menor e mais rápido o código gerado, maior será a eficiência do programa.

A linguagem C, devido a sua proximidade com o hardware e o *Assembly*, é uma linguagem extremamente eficiente. De fato, C é considerada como a linguagem de médio nível mais eficiente atualmente disponível.

Além disso, a utilização de uma linguagem de alto nível como C permite que o programador preocupe-se mais com a programação da aplicação em si, já que o compilador assume para si tarefas como o controle e localização das variáveis, operações matemáticas e lógicas, verificação de banco de dados de memória etc. (Pereira 2004).

Para o desenvolvimento do Software Embarcado, foi utilizada a IDE (Integrated Development Environment) da própria Microchip, chamada MPLab IDE, que é disponibilizada gratuitamente em seu site. Como linguagem de programação, foi adotada a linguagem C por resultar em um código de programação menor, mais legível e mais fácil de realizar manutenções, em comparação ao assembly.

Toda a programação do microcontrolador PIC16F877A é realizada em um microcomputador e gravada neste por um gravador de PIC's. Para a gravação do software foi necessária a aquisição de um gravador de PIC e o gravador escolhido foi o McPlus da Mosaico Engenharia.

A Mosaico é uma empresa localizada em Sando André (SP) e possui larga experiência no desenvolvimento de soluções envolvendo microcontroladores da linha PIC. Seu gravador, o McPlus, é o gravador mais completo da linha de gravadores que ela produz e é totalmente compatível com o PICStart Plus da Microchip, possibilitando assim uma completa integração com o MPLab IDE.

O Software Embarcado é responsável por executar as tarefas de leitura da tensão sobre fotodiodo (por intermédio do conversor A/D), comparação do nível de tensão lido com os intervalos de tensão definidos para cada par de bit, acumular os bits em uma variável até que se possua uma quantidade de bits mínima para o cálculo da paridade, realização do cálculo da paridade e possível correção do erro encontrado e transmissão do dado recebido por intermédio da USART interna.

3.2.4. O Conversor de Nível

O microcontrolador possui uma USART interna, também chamada de SCI (Interface de Comunicação Serial). Ela é um dispositivo utilizado para fazer a

comunicação serial com elementos externos ao chip, tais como: computadores, modems, terminais, memórias, conversores A/D e D/A etc.

O coração da USART é composto de dois registradores de deslocamento, responsáveis pela conversão paralelo/serial (transmissão), chamado internamente de TSR, e serial/paralelo (recepção), chamado internamente de RSR.

A transmissão serial assíncrona caracteriza-se pela ausência de uma linha de sincronização entre o elemento transmissor e o elemento receptor. Como não há sincronização de clock entre os elementos, utilizam-se velocidades padronizadas que devem ser seguidas por cada elemento de comunicação. Assim, uma vez que dois elementos estejam operando com uma mesma velocidade de comunicação, utilizam-se bits de START (início) e STOP (fim) para sinalizar o início e o fim de uma transmissão.

Assim como o microcontrolador, a maioria dos equipamentos digitais utilizam níveis TTL e CMOS. Portanto, o primeiro passo para conectar um equipamento digital a uma interface RS232 é transformar níveis TTL (0 a 5 volts) em RS232 e vice-versa. Isto é feito por conversores de nível. Existe uma grande variedade de equipamentos digitais que utilizam o *driver* 1488 (TTL => RS232) e o *receiver* 1489 (RS232 => TTL). Estes CIs contém 4 inversores de um mesmo tipo, sejam drivers ou receivers. O *driver* necessita de duas fontes de alimentação +7,5 volts a +15 volts e –7,5 volts a –15 volts. Isto é um problema onde somente uma fonte de +5 volts é utilizada.

Um outro CI que está sendo largamente utilizado é o MAX232 (da Maxim). Ele inclui um circuito de “charge pump” capaz de gerar tensões de +10 volts e –10 volts a partir de uma fonte de alimentação simples de +5 volts, bastando para isso alguns capacitores externos, conforme pode-se observar na Figura 3.7:

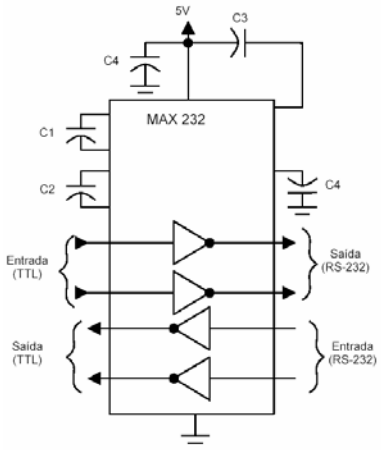


Figura 3.7 - Diagrama de Configuração do MAX232

Este CI também tem 2 receivers e 2 drivers no mesmo encapsulamento. Nos casos onde serão implementadas somente as linhas de transmissão e de recepção de dados, não seriam necessários 2 chips e fontes de alimentação extras. Devido a essas características, o MAX232 foi escolhido para fazer a conversão do nível TTL na saída do microcontrolador para o nível de tensão aceito pelo padrão RS232.

3.3. Receptor

Como a tarefa de decodificar o dado transmitido fica a cargo do Software Embarcado, a função do Software Receptor é realizar a leitura da porta serial, validar a paridade, e escrever os dados recebidos na tela do computador de destino.

O software deve ser configurado para operar com os mesmos parâmetros de transmissão definidos no microcontrolador.

O Software Receptor foi implementado totalmente na linguagem Java, utilizando a API “Java Communications”. A SUN fornece o download gratuito desta API de comunicação serial e paralela em seu site.

4. CONSTRUÇÃO DO SOFTWARE TRANSMISSOR

4.1. Função do Software Transmissor

O Software Transmissor é o responsável em transformar os dados a serem enviados em sinais ópticos exibidos na tela do monitor. Sinais que posteriormente serão capturados pela interface e enviados no padrão serial para o Software Receptor.

O *software* basicamente converte um dado ou um byte, em um valor binário, em outras palavras, ele *quebra* o byte em bits.

Após a conversão dos dados em bits, os mesmos são agrupados em *palavras* de 8 bits e é calculada a paridade (paridade bidimensional), adicionando assim, aos bits de dados, bits de controle que serão posteriormente utilizados pelo Software Embarcado na detecção e possível correção de erros nos bits de dados transmitidos.

Os bits resultantes desta operação são então agrupados 2 a 2 e exibidos da seguinte forma:

- Se forem encontrados dois bits 0 (00), a tela do monitor é preenchida com uma cor na escala de cinza um pouco mais escura que a cor branca;
- Se forem encontrados um bit 0 e um bit 1 (01), a tela do monitor é preenchida com uma cor na escala de cinza um pouco mais escura que a cor adotada para os bits “00”;
- Se forem encontrados dois bits 1 (11), a tela do monitor é preenchida com uma cor na escala de cinza um pouco mais clara que a cor negra, utilizada para os bits “11”;
- Se forem encontrados um bit 1 e um bit 0 (10), a tela do monitor é preenchida com a cor negra;

Portanto, se a tela do monitor é preenchida com uma cor de intensidade muito alta (cores claras como o branco), a interface não entende como sendo um dado (bit), não transmitindo nada. Para todas as outras intensidades de cores podem ser entendidas, se encaixando em alguma das faixas definidas.

O Software Transmissor é o item mais crítico do projeto, pois o seu funcionamento correto influencia toda a performance do sistema. O software tem que ser capaz de redesenhar o dado na tela (bit a ser transmitido) na mesma velocidade que a taxa de atualização do monitor (freqüência vertical). Por exemplo, suponha que o software esteja programado para enviar uma seqüência de quatro bits. Se o software não estiver sincronizado com a taxa de atualização do monitor, e a sua velocidade de redesenho for igual à metade da velocidade de atualização da tela, então, quando o software estiver transmitindo um bit, a tela é atualizada duas vezes. Isso significa que, quando forem transmitidos os bits ‘1100’, a interface irá capturar os bits dobrados: ‘11110000’.

O Software Transmissor foi implementado em C++ utilizando a API Directx. O programa apenas transmite os dados de acordo com os parâmetros recebidos pela linha de comando.

4.2. Modo de Chamada do Software Transmissor

O Software Transmissor foi desenvolvido de forma a ser o mais flexível possível em tempo de utilização.

Isto quer dizer que seus principais parâmetros de trabalho são passados via linha de comando, sem a necessidade de alteração em seu código fonte e posterior compilação do mesmo.

Dentre estes parâmetros estão os valores de cores na escala de cinza que o Software Transmissor irá utilizar quando da transmissão de cada par de bits. Isso torna possível a adaptação do projeto em ambientes diferentes, tanto em relação aos modelos de motores quanto à iluminação ambiente.

O Software Transmissor foi desenvolvido para trabalhar sobre o sistema operacional Microsoft Windows, porém o mesmo não dispõe de interface gráfica de usuário (GUI), sendo o dado a ser transmitido e outros parâmetros passados por linha de comando do *prompt*.

São estes os parâmetros:

- ***cNull***: representa um valor entre 0 e 255 que será usado na composição da cor que representará a ausência de bits a ser transmitido.

- ***c00***: representa um valor entre 0 e 255 que será usado na composição da cor que representará a seqüência de bits “00” na transmissão.
- ***c01***: representa um valor entre 0 e 255 que será usado na composição da cor que representará a seqüência de bits “01” na transmissão.
- ***c10***: representa um valor entre 0 e 255 que será usado na composição da cor que representará a seqüência de bits “10” na transmissão.
- ***c11***: representa um valor entre 0 e 255 que será usado na composição da cor que representará a seqüência de bits “11” na transmissão.
- ***s***: é a *string* de dados que será transmitida.

Exemplo de chamada ao Software Transmissor por intermédio da linha de comandos:

```
transmissor.exe 0 64 128 192 255 “123... Testando.”
```

4.3. Declaração e Inicialização dos Objetos DirectX Utilizados[8]

Para utilização da API DirectX é necessária a declaração de algumas variáveis e a chamada de alguns métodos para criação dos objetos que serão utilizados para manipular o monitor de vídeo.

Conforme o código abaixo, são necessários basicamente, 3 ponteiros para as interfaces **IDirectDraw7** e **IDirectDrawSurface7**.

```
LPDIRECTDRAW7 g_pDD = NULL;
LPDIRECTDRAWSURFACE7 g_pDDSFront = NULL;
LPDIRECTDRAWSURFACE7 g_pDDSBack = NULL;
```

A interface **IDirectDraw7** fornece às aplicações métodos para criar objetos **DirectDraw** e trabalhar com variáveis em “*system-level*”. O objeto **DirectDraw** é o coração de todas as aplicações DirectX. É o primeiro objeto que deve ser criado e, por intermédio dele, podemos criar todos os outros objetos relacionados à manipulação de vídeo (MSDN).

[8] O detalhamento dos objetos foi retirado da MSDN (http://msdn.microsoft.com/directx/)

O objeto **DirectDraw** representa o dispositivo de vídeo e emprega a aceleração de hardware caso o dispositivo de vídeo para o qual foi criado suportar tal aceleração. Cada objeto **DirectDraw** pode representar apenas um dispositivo de vídeo e pode manipular este dispositivo, criar superfícies, paletas e objetos *clipper* que são dependentes do (ou seja, “conectados ao”) dispositivo para o qual foi criado.

A interface **IDirectDrawSurface7** fornece às aplicações métodos para a criação de objetos **DirectDrawSurface** e trabalhar com suas variáveis em “*system-level*” (MSDN).

O objeto **DirectDrawSurface** representa uma superfície, que usualmente está localizada na memória da interface de vídeo, porém pode existir também na memória do sistema caso o dispositivo estiver esgotado ou haja uma requisição explícita para tal.

Neste caso foram declaradas duas variáveis **IDirectDrawSurface7**, uma para representar a superfície primária e outra para representar a superfície *back buffer*. A criação destas superfícies é necessária para utilização da técnica de *page-flipping* descrita no capítulo 2.

O código abaixo demonstra a inicialização do objeto **DirectDraw**:

```
DirectDrawCreateEx(NULL, (VOID**)&g_pDD, IID_IDirectDraw7,
                                                    NULL);
```

Para o DirectX 7 foi introduzido um novo método para criação do objeto **DirectDraw**. A função do método **DirectDrawCreateEx** é criar um objeto **DirectDraw** capaz de expor todo o novo conjunto de interfaces **Direct3D**.

Abaixo, segue o protótipo do método **DirectDrawCreateEx**:

```
HRESULT WINAPI DirectDrawCreateEx(
     GUID FAR *lpGUID,
     LPVOID *lplpDD,
     REFIID iid,
     IUnknown FAR *pUnkOuter
);
```

Seus parâmetros são:

- ***lpGUID***: endereço GUID (Globally Unique Identifier) que representa o *driver* do dispositivo a ser criado. Este parâmentro pode ser *null* para indicar o *driver* ativo. Também podemos passar um dos *flags* de restrição do comportamento do dispositivo com o propósito de realizar depuração do código de programa.
- ***lplpDD***: endereço da variável do tipo **IDirectDraw7** que será atribuído o ponteiro que representa o dispositivo utilizado.
- ***Iid***: este parâmetro deve sempre passar **IID_IDirectDraw7**. A função falhará caso qualquer outra interface for especificada.
- ***pUnkOuter***: permite uma futura integração e compatibilidade com os recursos COM. Atualmente a função retorna um erro para qualquer valor diferente de *null* para este parâmetro.

Após a criação do objeto **DirectDraw**, são invocados dois métodos da interface **IDirectDraw7** a fim de determinar o comportamento da aplicação junto ao dispositivo.

```
g_pDD->SetCooperativeLevel(g_hMainWnd, DDSCL_EXCLUSIVE |
                                       DDSCL_FULLSCREEN );

g_pDD->SetDisplayMode(800, 600, 32, 0, 0);
```

O método **SetCooperativeLevel** é responsável por determinar o comportamento da aplicação em alto nível. Abaixo segue o protótipo deste método:

```
HRESULT SetCooperativeLevel(
     HWND hWnd,
     DWORD dwFlags
);
```

Seus parâmetros são:

- ***hWnd***: é o ponteiro para o *handle* da aplicação. Deve ser passado o *handle* para a janela da aplicação principal, não para as janelas filhas criadas por ela.
- ***dwFlags***: podem ser passados mais de 12 *flags*, e estes podem ser combinados de várias formas. Os *flags* utilizados no Software Transmissor são:
    - **DDSCL_EXCLUSIVE**: requisita a exclusividade de acesso ao dispositivo. Deve ser usada em conjunto com o *flag* **DDSCL_FULLSCREEN**.
    - **DDSCL_FULLSCREEN**: o proprietário do modo exclusivo é responsável por toda a superfície primária. A GDI (Graphical Device Interface) pode ser ignorada.

O método **SetDisplayMode** configura o modo de operação do dispositivo de vídeo. Abaixo segue o protótipo deste método:

```
HRESULT SetDisplayMode(
     DWORD dwWidth,
     DWORD dwHeight,
     DWORD dwBPP,
     DWORD dwRefreshRate,
     DWORD dwFlags
);
```

Seus parâmetros são:

- ***dwWidth*** e ***dwHeight***: definem as dimensões do novo modo.
- ***dwBPP***: define os *bits por píxel* do novo modo.
- ***dwRefreshRate***: configura a freqüência vertical de operação. Pode ser passado “0” para que se utilize a freqüência configurada pelo sistema operacional.
- ***dwFlags***: usado para configurar opções adicionais. Atualmente existe apenas um *flag*: **DDSDM_STANDARDVGAMODE**, que faz com que o

método configure o Modo 13 no lugar do Modo X 320x200x8. Para qualquer outro modo de resolução deve ser passado “0”.

Neste projeto foi adotado para o parâmetro ***dwBPP*** o valor “32” porque esta quantidade de bits possibilita o armazenamento completo dos 3 bytes que representam cada cor no modelo RGB, ao contrário do modo de 16 bits.

No modo 16 bits as cores RGB são distribuídas de forma quase uniforme, sendo 5 bits para a cor vermelha, 5 bits para a cor azul e 6 bits para a cor verde. A cor verde recebe o bit que sobra, pois as pesquisas sobre o olho humano demonstraram que este é mais sensível à luz verde.

O modo de 32 bits é também melhor que o modo de 24 bits, pois as placas gráficas trabalham melhor com números em potência de 2. Isto se dá devido ao fato de que ao utilizar a base binária, os sistemas computacionais conseguem realizar, com números nesta potência, cálculos complexos por intermédio de instruções simples, como por exemplo o deslocamento de bits.

A criação da superfície primária se dá por intermédio do método **CreateSurface** da interface **IDirectDraw7**. Tal método cria um objeto **DirectDrawSurface** e configura um ponteiro para o mesmo.

Abaixo, o protótipo deste método:

```
HRESULT CreateSurface(
     LPDDSURFACEDESC2 lpDDSurfaceDesc2,
     LPDIRECTDRAWSURFACE7 FAR *lplpDDSurface,
     IUnknown FAR *pUnkOuter
);
```

Os parâmetros deste método são:

- ***lpDDSurfaceDesc2***: endereço de uma estrutura do tipo DDSURFACEDESC2 que descreve as características da superfície a ser criada.

- ***lplpDDSurface***: endereço da variável do tipo **IDirectDrawSurface7** que será configurada com o ponteiro para a superfície caso a função seja bem sucedida.
- ***pUnkOuter***: permite a futura compatibilidade com o COM. Atualmente deve ser passado *null*, caso contrário será retornado um erro.

Antes de invorcarmos o método **CreateSurface** devemos configurar uma estrutura do tipo **DDSURFACEDESC2** que será passada como parâmetro deste método. Os elementos desta estrutura, importantes para o projeto, são:

- ***dwSize***: tamanho da estrutura, em bytes. É obrigatória a inicialização deste membro antes que a estrutura possa ser utilizada.
- ***dwFlags***: são definidos muitos *flags* de controle, podendo haver a passagem de um ou mais *flags*. Neste projeto utilizamos os *flags*:
  - **DDSD_CAPS**: ativa a subestrutura **ddsCaps**, do tipo **DDSCAPS2**, que armazena as potencialidades da superfície
  - **DDSD_BACKBUFFERCOUNT**: ativa o elemento da estrutura **dwBackBufferCount** responsável por armazenar a quantidade de *back buffers* que a superfície irá conter.
- ***ddsCaps.dwCaps***: representa *flags* que definem a capacidade da superfície. Neste projeto utilizamos os *flags*:
  - **DDSCAPS_PRIMARYSURFACE**: indica que a superfície é a superfície primária. Isto que dizer que representa o que está atualmente visível.
  - **DDSCAPS_FLIP**: indica que esta superfície é parte de uma estrutura *surface-flipping*, necessária par a utilização da técnica de *page-flipping* apresentada no capítulo 2.
  - **DDSCAPS_COMPLEX**: indica que uma superfície complexa está sendo criada. Uma superfície complexa resulta na criação de mais de uma superfície. As superfícies adicionais são unidas à superfície raiz. A superfície complexa pode ser destruída apenas destruindo a superfície raiz.

Exemplo de código utilizado para criação da superfície primária:

```
DDSURFACEDESC2 ddsd;
DDSCAPS2 ddscaps;

ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                      DDSCAPS_FLIP | DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
g_pDD->CreateSurface(&ddsd, &g_pDDSFront, NULL);
```

Para criar a superfície de *back buffer* é utilizado o método **GetAttachedSurface** da interface **IDirectDrawSurface7**, à qual pertence a superfície primária. Abaixo, o protótipo deste método:

```
HRESULT GetAttachedSurface(
     LPDDSCAPS2 lpDDSCaps,
     LPDIRECTDRAWSURFACE7 FAR *lplpDDAttachedSurface
);
```

Os parâmetros deste método são:

- ***lpDDSCaps***: endereço de uma estrutura do tipo DDSCAPS2 que descreve as características da superfície a ser criada.
- ***lplpDDAttachedSurface***: endereço da variável do tipo **IDirectDrawSurface7** que será configurada com o ponteiro para a superfície caso a função seja bem sucedida.

Exemplo de código utilizado para criação da superfície de *back buffer*:

```
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
g_pDDSFront->GetAttachedSurface(&ddscaps,
                                &g_pDDSBack);
```

4.4. Decomposição do Dado a ser Transmitido em Bits

A função principal do Software Transmissor é transmitir o dado desejado alternando as superfícies de forma que o monitor de vídeo apresente uma cor na escala de cinza que corresponda a cada par de bits da informação.

Dito isto, uma das principais necessidades do Software Transmissor, antes de iniciar a transmissão, é “quebrar” a informação em bits de forma que, posteriormente, estes bits sejam associados a suas cores correspondentes e o processo de alternância de cores.

Abaixo, segue o trecho de código utilizado para tal tarefa:

```
int len = strlen(s);
int j = bitLen = len*8;
bits = new bool[j];

for(int i=--len; i>=0; i--){
     int dado = s[i];
     for(int k=0; k<8; k++){
          bits[--j] = (dado % 2 == 0) ? false : true;
          dado=dado/2;
     }
}
```

A variável “***s***“ armazena a string de dados a ser transmitida. Tal string foi recebida pelo Software Transmissor por intermédio da linha de comandos do sistema operacional.

Por intermédio de um *loop* o Software Transmissor percorre todos os caracteres da *string* e, para cada caractere, é realizada a sua divisão de seu código ASCII pela razão de 2. O resto desta divisão representa o bit do dado e o resultado da divisão é dividido novamente pela mesma razão até que se chegue aos 8 bits do caractere.

O resto das divisões, que representa os bits da informação, são armazenados em uma matriz com o tamanho da *string* original em bytes multiplicada por 8, pois cada byte é composto de 8 bits. Esta matriz resultante é que será posteriormente

utilizada para a decisão de quais cores e em qual seqüência o Software Transmissor irá alternar o monitor de vídeo.

4.5. Cálculo da Paridade

Conforme abordado no Capítulo 2, a técnica de correção e detecção de erros adotada na transmissão de dados deste projeto, é a paridade combinada.

Na paridade combinada o Software Transmissor deve realizar o cálculo da paridade para cada byte a ser transmitido.

Dentre as técnicas que podem ser adotadas são a utilização da tabela ASCII padrão de 7 bits, adicionando o 8º bit como sendo o bit de paridade. Este 8º bit é o bit mais significativo, resultando assim em uma palavra de 8 bits.

A cada 7 palavras é adicionada mais uma palavra, com o tamanho de 8 bits, onde cada bit representa o cálculo da paridade vertical.

A paridade utilizada no projeto é do tipo par, ou seja, caso o número de bits 1's for impar, é adicionado mais um bit 1 a fim de fazer com que o número de bits 1's seja par. Caso a quantidade de bits 1's já seja um número par, é adicionado um bit 0, mantendo a quantidade de bits 1's inalterada.

4.6. Freqüência de Transmissão[9]

O Software Transmissor opera na freqüência do monitor de vídeo. Para tanto é utilizado o método WaitForVerticalBlank da interface IDirectDraw7.

Este método ajuda a aplicação a sincronizar, por si mesma, com a freqüência vertical (vertical-blank) utilizado pelo sistema. O protótipo deste método é apresentado abaixo:

```
HRESULT WaitForVerticalBlank(
    DWORD dwFlags,
    HANDLE hEvent
);
```

Os parâmetros deste método são:

[9] O detalhamento dos objetos foi retirado da MSDN (http://msdn.microsoft.com/directx/)

- ***dwFlags***: determina quanto tempo a aplicação deve esperar por intermédio do tempo de intervalo da atualização vertical. São válidas as seguintes opções:
  - **DDWAITVB_BLOCKBEGIN**: retorna quando o intervalo da atualização vertical iniciar.
  - **DDWAITVB_BLOCKBEGINEVENT**: dispara um evento quando o intervalo da atualização vertical iniciar. Este *flag*, atualmente, não é válido.
  - **DDWAITVB_BLOCKEND**: retorna quando o intervalo da atualização vertical terminar e a atualização do vídeo iniciar.
- ***hEvent***: *handle* do evento disparado quando o intervalo da atualização vertical iniciar. Este parâmetro, assim como o *flag* **DDWAITVB_BLOCKBEGINEVENT** não é suportado atualmente.

4.7. Transmissão da Matriz de Bits[10]

A transmissão dos bits se dá por intermédio da alternância das cores no monitor de vídeo de forma que cada cor represente um par de bits a ser transmitido.

Para tal tarefa é utilizada a técnica de *page-flipping* onde são criadas duas superfícies de desenho, sendo uma chamada de primária e a outra de *back buffer*. Enquanto uma das superfícies (a primária) está sendo exibida, a outra (*back buffer*) é desenhada. Após a exibição, as superfícies são “trocadas” de forma que a superfície que estava sendo desenhada se torna a superfície primária e a superfície primária se torna a superfície *back buffer* que será desenhada de acordo com os próximos bits da seqüência. Este processo se repete até que todos os bits sejam transmitidos.

O desenho das telas nada mais é que o preenchimento total da superfície com a cor definida pelos bits a ser transmitidos. Para esta tarefa é utilizado o método **Blt** da interface **IDirectDrawSurface7**.

Este método executa a transferência de blocos de bit (blit) em uma superfície. A origem do bloco pode ser outra superfície. Abaixo segue o protótipo deste método:

[10] O detalhamento dos objetos foi retirado da MSDN (http://msdn.microsoft.com/directx/)

```
HRESULT Blt(
     LPRECT lpDestRect,
     LPDIRECTDRAWSURFACE7 lpDDSrcSurface,
     LPRECT lpSrcRect,
     DWORD dwFlags,
     LPDDBLTFX lpDDBltFx
);
```

Os parâmetros deste método são:

- ***lpDestRect***: endereço de uma estrutura **RECT** que define os pontos superior-esquerdo e inferior-direito de um retângulo que será preenchido na superfície de destino. Caso seja passado o valor *null* toda a superfície é preenchida.
- ***lpDDSrcSurface***: endereço da superfície de origem dos bits para preenchimento da superfície de destino.
- ***lpSrcRect***: endereço de uma estrutura **RECT** que define os pontos superior-esquerdo e inferior-direito de um retângulo que compreende os bits na superfície de origem. Caso seja passado o valor *null* toda a superfície é utilizada.
- ***dwFlags***: são definidos muitos *flags* de controle, podendo haver a passagem de um ou mais *flags*. Neste projeto utilizamos os *flags*:
    - **DDBLT_WAIT**: indica que, caso o *blitter* esteja ocupado, o sistema deve aguardar que o mesmo seja liberado ou que seja gerado um erro para que o método retorne.
    - **DDBLT_COLORFILL**: indica o uso do membro **dwFillColor** da estrutura do tipo **DDBLTFX** como uma cor RGB que irá preencher o retângulo de destino na superfície de destino.
- ***lpDDBltFx***: endereço da estrutura do tipo **DDBLTFX**.

A estrutura **DDBLTFX** possui muitos membros, sendo que o único utilizado para o desenho das superfícies neste projeto é o **dwFillColor**. Este membro deve

ser configurado com o valor apropriado do píxel no formato da paleta de cores da superfície a ser preenchida.

Após o preenchimento da superfície *back buffer* com a cor desejada, é necessário realizar a “troca” das superfícies. O processo de *page-flipping* dá-se por intermédio do método **Flip** da interface **IDirectDrawSurface7**.

Este método faz com que a superfície em memória associada ao DDSCAPS_**BACKBUFFER** da superfície primária venha a ser a superfície *front-buffer*. Abaixo segue seu protótipo:

```
HRESULT Flip(
     LPDIRECTDRAWSURFACE7 lpDDSurfaceTargetOverride,
     DWORD dwFlags
);
```

Os parâmetros deste método são:

- ***lpDDSurfaceTargetOverride***: endereço de uma superfície do tipo IDirectDrawSurface7 que esteja associada à superfície primária. O padrão para este parâmetro é *null*, informando que o **DirectDraw** deve buscar a próxima superfície da fila. Se este parâmetro não for *null*, o **DirectDraw** irá buscar a superfície desejada, desde que esteja na fila da superfície primária. Caso não esteja, um erro é retornado.
- ***dwFlags***: são definidos muitos *flags* de controle, podendo haver a passagem de um ou mais *flags*. Neste projeto utilizamos apenas o *flag* **DDFLIP_WAIT**.
    - **DDFLIP_WAIT**: indica que o método não retorna até que a troca não tenha sido bem sucedida ou algum erro seja retornado.

5. CONSTRUÇÃO DO SOFTWARE RECEPTOR

5.1. A API de Comunicação da SUN

A SUN fornece como download gratuito a API de comunicação serial.

Basta baixar a API e realizar os procedimentos de instalação. Após baixar a API, descompactá-la, é necessário:

- Copiar o arquivo win32com.dll para o diretório C:\JavaSDK\BIN (isto é, o diretório onde o J2SDK foi instalado no seu PC);
- Copiar o arquivo comm.jar para o diretório C:\JavaSDK\BIN\LIB;
- Copiar o arquivo javax.comm.properties para o diretório C:\JavaSDK\BIN\LIB;
- Em seguida configurar o CLASSPATH para que ele reconheça o arquivo comm.jar.

5.1.1. Reconhecendo as Portas

Antes de iniciar a comunicação com a porta serial, é necessário reconhecer as portas existentes em sua estação de trabalho. A API de comunicação fornece o método getPortIdentifiers() integrante da classe CommPortIdentifier que retorna em uma estrutura Enumeration, as portas disponíveis. A Classe CommPortIdentifier pode ser instanciada e representar uma porta. Para isso, precisamos varrer a estrutura retornada por getPortIdentifiers() e instanciando cada porta por intermédio de uma conversão (casting) simples:

```
Enumeration listaDePortas;
listaDePortas = CommPortIdentifier.getPortIdentifiers();

int i = 0;
portas = new String[10];
while (listaDePortas.hasMoreElements()) {
     CommPortIdentifier ips =
(CommPortIdentifier)listaDePortas.nextElement();
```

```
        portas[i] = ips.getName();
        i++;
}
```

O método hasMoreElements() retorna o próximo elemento da estrutura listaDePortas, mas o loop while garante que todos os elementos sejam passados ao Array portas através do método getName().

5.1.2. Abrindo Portas para Comunicação

O método getPortIdentifier(String porta) da classe CommPortIdentifier retorna um identificador da porta escolhida. É necessário instanciar um objeto para receber esse identificador:

```
CommIdentifier cp =
CommPortIdentifier.getPortIdentifier(minhaPortaEscolhida);
```

Em seguida cria-se uma instância da classe SerialPort utilizando o identificador. Note que uma conversão deverá ser feita. A porta só pode ser instanciada por intermédio do que chamamos de *casting* e ao mesmo tempo abrimos a porta para comunicação:

```
SerialPort porta = (SerialPort)cp.open("SComm",timeout);
```

O método open() tem como parâmetros o nome da classe principal e o valor desejado para timeout. Em seguida, precisa-se atribuir fluxos de entrada e saída. Basta utilizar as classes Abstratas OutputStream e InputStream, já que a classe SerialPort implementa os métodos de entrada e saída dessas classes para comunicação serial.

Para ler dados na porta serial:

```
InputStream entrada = porta.getInputStream();
```

E para escrever dados na porta serial:

```
OutputStream saida = porta.getOutputStream();
```

Em seguida precisamos configurar os parâmetros de comunicação serial, para isso utilizamos o método setSerialPortParams:

```
porta.setSerialPortParams(baudrate, porta.DATABITS_8,
porta.STOPBITS_2, porta.PARITY_NONE);
```

5.1.3. Recebendo Dados da Porta Serial

A API de comunicações Java facilita bastante o trabalho de recepção de dados da porta serial, mas mesmo assim, são várias linhas de código.

Basicamente o que deve ser feito é:

- Criar um fluxo de entrada;
- Adicionar um gerenciador de eventos para dados na porta serial;
- Instanciar uma Thread para aguardar os eventos;
- Tratar o evento e receber os dados

A criação do fluxo de entrada já foi detalhada.

Para adicionar um gerenciador de eventos para a porta serial basta fazer:

```
porta.addEventListener(this);
```

Em seguida é necessário notificar o objeto “porta” criado de que podem existir dados para serem lidos:

```
porta.notifyOnDataAvailable(true);
```

Agora falta apenas tratar o evento. Primeiro instanciamos um Array de bytes. Esse Array será nosso buffer de dados.

```
public void serialEvent(SerialPortEvent ev){
     switch (ev.getEventType()) {
          case SerialPortEvent.DATA_AVAILABLE:
               byte[] bufferLeitura = new byte[20];
```

Já definimos entrada como nosso fluxo de entrada de dados. O método available() retorna sempre 0 se InputStream (nesse caso entrada) é classe da qual ele é invocado.

```
               try {
                    while ( entrada.available() > 0 ) {
                         nodeBytes =
                         entrada.read(bufferLeitura);
                    }
```

O metodo read(byte[] b) faz toda a leitura. Ele copia os bytes lidos para o Array bufferLeitura e retorna um inteiro representando o número de bytes lidos.

Podemos converter esses valores para uma String como mostrado abaixo:

```
                    String Dadoslidos = new
                    String(bufferLeitura);
```

Se a dimensão do buffer for igual à zero, isso nos dirá que nenhum byte foi lido. Se a dimensão do buffer for igual a 1, saberemos que apenas um byte foi lido. Caso contrário, a estrutura bufferLeitura recebe os bytes lidos. O primerio byte lido é armazenado em bufferLeitura[0], o segundo em bufferLeitura[1] e assim por diante.

```
                    if (bufferLeitura.length == 0) {
                         System.out.println("Nada lido!");
                    } else if (bufferLeitura.length == 1 ){
                         System.out.println("Apenas um byte foi
                    lido!");
                    } else {
                         System.out.println(Dadoslidos);
                    }
```

```
            } catch (Exception e) {
                 System.out.println("Erro durante a leitura:
            " + e );
            }
            System.out.println("n.o de bytes lidos : " +
       nodeBytes );
            break;
       }
  }
```

5.2. Classes do Software Receptor

Como pôde ser visto até aqui o estabelecimento de uma conexão e a recepção de dados em uma porta serial com a API da Sun é um procedimento complexo e composto de várias linhas de código com um único propósito: ler os dados disponibilizados na porta serial.

Outro ponto importante é o fato de que a porta serial permite que apenas um processo detenha seu controle e, conseqüentemente, sua leitura. Isto limita a apenas uma *thead* (ou objeto) o acesso à porta serial em um dado momento.

Caso, no futuro, o Software Receptor necessite que mais objetos leiam os dados da porta serial, o programa construído de maneira monolítica se mostraria inadequado.

Com o objetivo de flexibilizar o Software Receptor, e permitir a futura criação de outros objetos interessados nos dados da porta serial de maneira simultânea, foi criada uma classe exclusiva para a leitura constante da porta serial e a disponibilização dos dados lidos aos outros objetos interessados.

A fim de atingir tal objetivo, foi adotado um padrão de projetos largamente utilizado na engenharia de softwares, o padrão *Observer*.

5.3. O Padrão de Projetos *Observer*

O padrão de projetos (*design patterns*) *Observer* é também conhecido como padrão *publisher and subscriber.* Este padrão é útil quando se tem um *publicador* e vários assinantes (um-para-muitos) que estão interessados no estado ou nas mensagens deste.

Adicionalmente, os assinantes interessados possuem a habilidade de se registrar ou cancelar sua assinatura.

Por último, os assinantes são notificados das mensagens do *publicador* automaticamente. A Figura 5.1 é um exemplo típico do padrão *Observer*. [11]

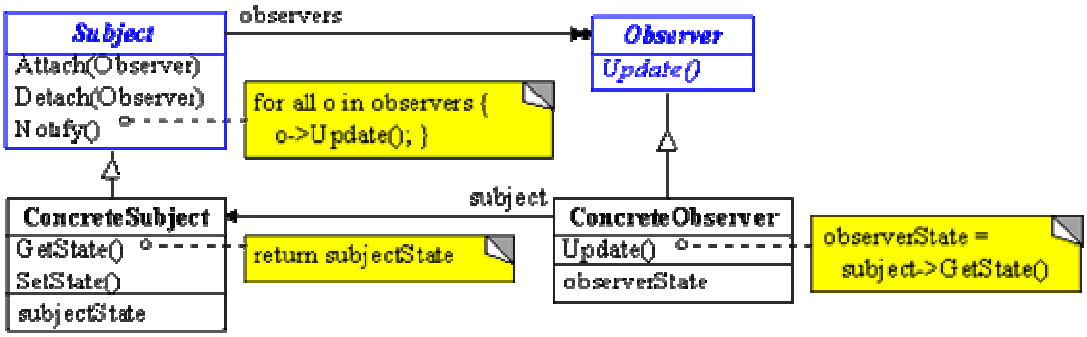


Figura 5.1 - Diagrama de Classes do Padrão Observer

5.4. Interface *IPublisher*

A interface *IPublisher* foi criada com o objetivo de definir métodos que todas as classes que devam ser do tipo *publicadoras* devem implementar.

Esta interface possui a definição de três métodos:

- **attach**: método responsável por colocar o objeto assinante na lista de assinantes da classe publicadora;
- **detach**: método responsável por retirar o objeto assinante da lista de assinantes da classe publicadora;

[11] Tradução livre de artigo do site TheServerSide.COM (http://www.theserverside.com/articles/article.tss?l=SpringLoadedObserverPattern)

- **updateAll**: método responsável por chamar os métodos de atualização dos objetos assinantes;

5.5. Interface *ISubscriber*

A interface *Isubscriber* foi criada com o objetivo de definir métodos que todas as classes que devam ser do tipo *assinante* devem implementar.

Esta interface possui apenas o método updade:

- **update**: método responsável realizar todos os procedimentos que a classe assinante deva executar quando da atualização do objeto publicador;

5.6. Classe *Receptor*

A classe *Receptor* é responsável única e exclusivamente por realizar a leitura constante da porta serial e, quando houver dados lidos disponíveis, avisar às outras classes interessadas nesta informação da existência destes dados.

Para a implementação desta classe, além dos objetos e métodos demonstrados na seção 5.1, foram adicionados mais membros e métodos a fim de atender ao padrão *Observer*, descrito na seção 5.3.

Os métodos adicionais são provenientes da interface *IPublisher*.

5.7. Classe *SoftwareReceptor*

Dentro do conceito do padrão de projetos *Observer*, pode-se definir a classe *SoftwareReceptor* como sendo uma classe assinante da classe *Receptor*.

A classe *SoftwareReceptor* realiza as seguintes tarefas:

- Cria uma nova instância da classe *Receptor*, caso a mesma ainda não exista;
- Registra a si mesma como assinate da classe *Receptor*;
- Aguarda até que seu método *update* seja chamado pela classe *Receptor;*

Assim que a classe *Receptor* realiza a chamada do método *update* a classe *SoftwareReceptor* executa as seguintes ações:

- Realiza a leitura dos dados recebidos pela classe *Receptor*;
- Acumula os dados recebidos em uma *string*;
- Caso o tamanho da *string* acumuladora for igual ou maior que 8 caracteres, é acionado o procedimento de paridade e escrita do dado validado / corrigido;
- Caso contrário a classe *SoftwareReceptor* continua a aguardar por mais dados.

5.7.1. Validando a Paridade dos Dados Recebidos

Quando os dados são lidos da porta serial, estes são acumulados em uma matriz de bytes de tamanho 8.

Após preencher os 8 bytes da matriz de bytes, é realizado o cálculo e validação da paridade vertical e horizontal. Caso seja encontrado algum erro nas paridades vertical e horizontal, o processo de localização e correção do bit transmitido de forma errada é executado.

6. CONSTRUÇÃO DA INTERFACE

6.1. O Regulador de Tensão

De um modo geral, o primeiro passo para a construção da interface é uma alimentação correta, pois é de grande importância para o bom funcionamento do sistema de um microcontrolador. Pode comparar-se este sistema a um homem que precisa respirar. É provável que um homem que respire ar puro viva mais tempo que um que viva num ambiente poluído (Mitsuka 2004).

Para que o circuito funcione convenientemente, é necessário usar uma fonte de alimentação estável. O regulador de tensão de 3 conexões é um dispositivo comumente utilizado na regulagem de tensão. Pode-se imaginá-lo como um tipo especial de zener. Este dispositivo apresenta três conexões (entrada, saída e terra) e são regulados na fábrica para uma saída fixa de tensão (positiva para a família 78xx e negativa para 79xx). A função do regulador é manter constante a tensão, garantindo que a tensão seja limitada a um valor preestabelecido (Mitsuka 2004).

Pequenos capacitores são colocados em paralelo com a entrada e a saída do regulador, para protegê-lo de algum possível ruído elétrico e estabilizar a saída sob certas circunstâncias. Os valores destes capacitores obedece tabela localizada no *datasheet* do componente.

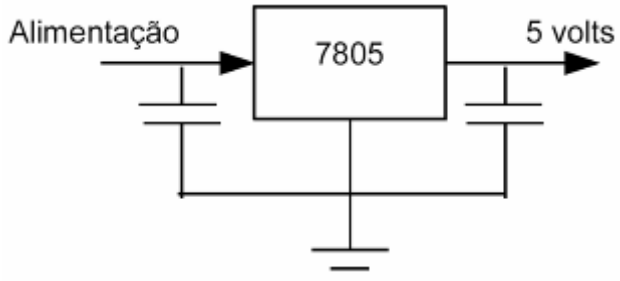


Figura 6.1 - Circuito esquemático da alimentação

6.2. O Fotodiodo

O componente responsável pela transdução dos sinais ópticos (dados luminosos) enviados pelo monitor de vídeo em um sinal elétrico analógico é o fotodiodo.

Uma das componentes da corrente reversa num diodo é o fluxo de portadores minoritários. Esses portadores existem porque a energia térmica mantém os elétrons de valência desalojados de suas órbitas, produzindo assim elétrons livres e lacunas. A vida média dos portadores minoritários é curta, mas enquanto eles existirem, podem contribuir para a permanência da corrente reversa (Malvino 95).

Quando a energia luminosa bombardeia uma junção *pn*, ela pode deslocar elétrons de valência. Quanto mais intensa for a luz incidente na junção, maior será a corrente reversa num diodo. Um fotodiodo é otimizado para ter uma alta sensibilidade à luz incidente. Nesse diodo, uma janela deixa passar a luz através do encapsulamento da junção. A luz penetrante produz elétrons livres e lacunas. Quanto maior a intensidade luminosa, maior o número de portadores minoritários e maior a corrente reversa (Malvino 95).

Na Figura 6.2 é ilustrado o símbolo elétrico de um fotodiodo. As duas setas representam a luz penetrante.

Figura 6.2 - Símbolo esquemático do fotodiodo

O fotodiodo funciona da seguinte forma: se for polarizado inversamente na presença de luz ele permite a passagem de corrente elétrica e produz uma queda de tensão; se for polarizada reversamente na ausência de luz ele só permite a passagem de uma corrente muito pequena (praticamente nula) chamada de corrente reversa. Portanto, um fotodiodo ideal funcionaria como uma chave comum que fecha quando há presença de luz e abre quando há ausência de luz. A Figura 6.3 resume a idéia de chave (Mitsuka 2004).

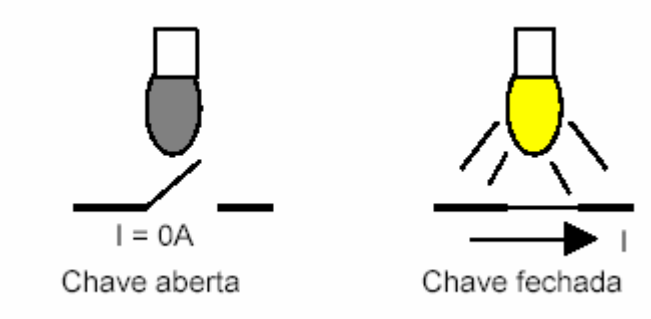


Figura 6.3 - Funcionamento de um fotodiodo ideal

O circuito de detecção foi projetado de forma que, a cada estímulo luminoso, a corrente reversa que passa no fotodiodo passe também em uma resistência colocada em série com o mesmo, gerando assim uma tensão sobre a resistência. O circuito faz uma transdução de corrente para tensão:

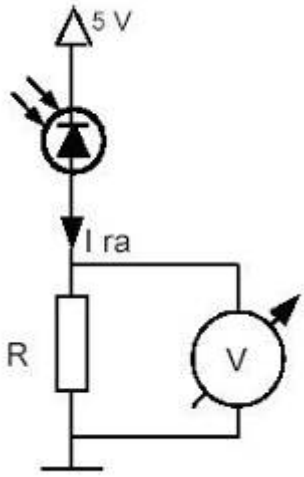


Figura 6.4 - Circuito esquemático do transdutor

6.3. O Conversor A/D

Como os sinais dos periféricos são substancialmente diferentes daqueles que um microcontrolador pode entender (zero e um), eles devem ser convertidos num formato que possa ser compreendido pelo microcontrolador. Esta tarefa é executada por intermédio de um bloco destinado à conversão analógica-digital ou com um conversor A/D. Este bloco é responsável pela conversão de uma informação

analógica para um número binário e pelo seu trajeto através do bloco do CPU, de modo a que este o possa processar de imediato.

No caso do microcontrolador PIC16F877A, utilizado neste projeto, o mesmo possui 8 canais de conversão AD com um registrador AD de 10 bits.

Figura 6.5 - Esquema do Conversor A/D dos PIC's

6.4. O Cristal

O oscilador de cristal está contido num invólucro de metal com os pinos ligados aos pinos OSC1 e OSC2 do microcontrolador. Dois capacitores cerâmicos devem estar ligados a cada um dos pinos do cristal e ao terra. O valor destes capacitores varia de acordo com a Tabela x.x extraída do *datasheet* do PIC 16F877A:

Tabela 6.1 - Valores de Capacitor utilizados junto ao cristal

Tipo de Osc.	Freq. Do Cristal	Intervalo do C1	Intervalo do C2
LP	32 kHz	33 pF	33 pF
	200 kHz	15 pF	15 pF
XT	200 kHz	47-68 pF	47-68 pF
	1 MHz	15 pF	15 pF
	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
	8 MHz	15-33 pF	15-33 pF
	20 MHz	15-33 pF	15-33 pF

Quando se projeta um dispositivo, a regra é colocar o cristal tão perto quanto possível do microcontrolador. Este procedimento tem por objetivo evitar qualquer interferência nas linhas que ligam o cristal ao microcontrolador devido à indução elétrica externa ou de outros componentes do circuito.

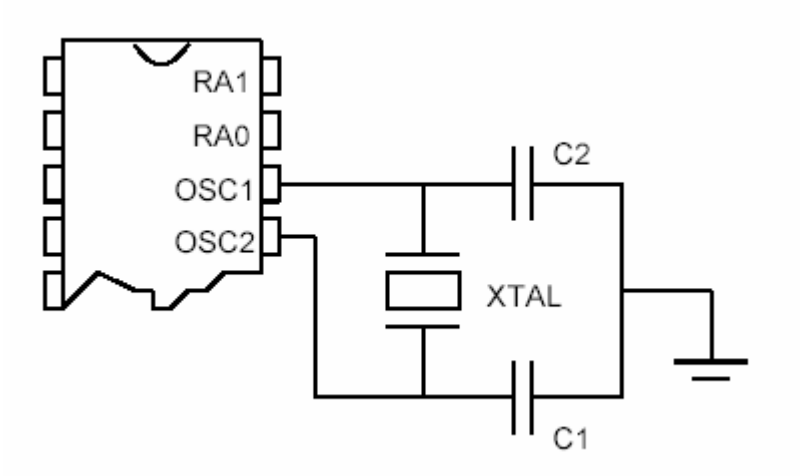


Figura 6.6 - Clock de um microcontrolador a partir de um cristal de quartzo

Ao ligar a alimentação do circuito o oscilador começa a trabalhar, porém de forma instável. Apenas após alguns microssegundos o mesmo estabiliza em um período e amplitude constantes.

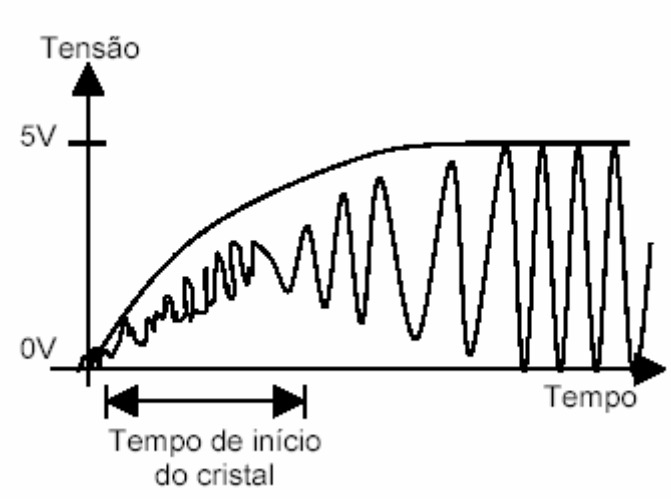
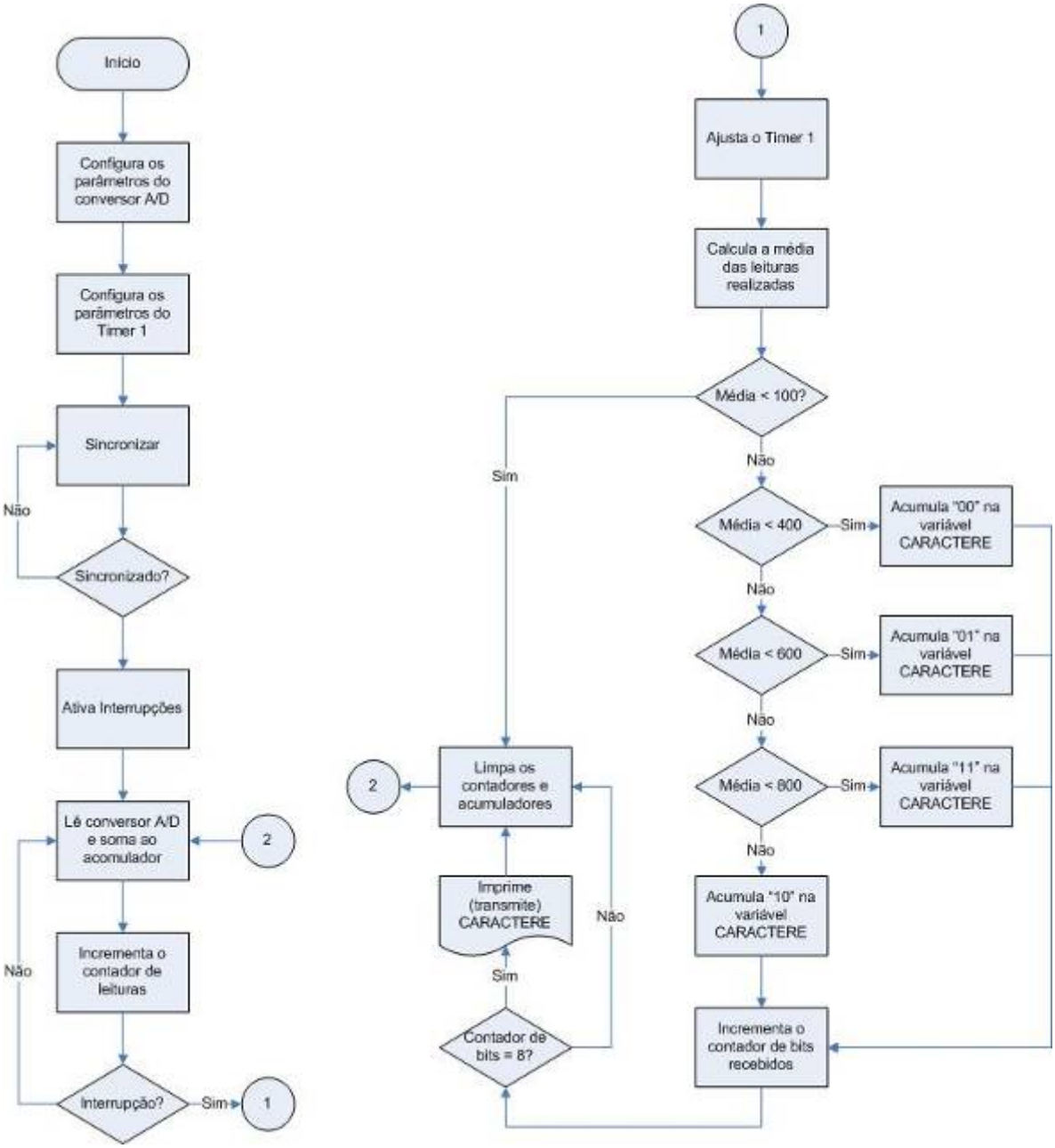


Figura 6.7 - Sinal de clock do oscilador depois de ser ligada a alimentação

7. CONSTRUÇÃO DO SOFTWARE EMBARCADO NO MICROCONTROLADOR

O Software Embarcado está residente no microcontrolador, possibilitando que a interface realize a tarefa de reconhecimento das cores e repassando os dados recebidos para a USART que está conectada à porta serial do microcomputador receptor.

7.1. Fluxograma



7.2. Estratégia do Software Embarcado

A interface deve funcionar na mesma freqüência que a utilizada como freqüência vertical no monitor de vídeo do computador transmissor. Matematicamente falando, isto significa que o intervalo de tempo entre um par de bits e outro é de $16,\overline{66}$ milissegundos.

$$Freq = 60Hz$$
$$T = \frac{1}{Freq} = \frac{1}{60} = 16,\overline{66}\ ms$$

Equação 7.1 - Cálculo do Período de Atualização do Monitor de Vídeo

O tempo de aquisição do conversor AD varia entre 10 μs e 19 μs. Este tempo de aquisição depende, dentre outros fatores, da quantidade de bits utilizados na conversão, da impedância de entrada no pino do canal do conversor, do *clock* utilizado etc.

De toda forma, o tempo de aquisição do conversor AD é muito menor que o tempo de atualização do monitor de vídeo, logo o microcontrolador, teoricamente, ficaria parado aguardando até que a próxima atualização de vídeo ocorresse.

Além deste fato foi constatado de maneira experimental que existe uma pequena variação na intensidade luminosa do monitor de vídeo durante este tempo. A causa desta variação é o fato de que o sensor é muito maior que um *pixel* do monitor de vídeo. Como o sensor capta vários *pixies*, o feixe de elétrons é detectado muito antes de passar pela região de foco do sensor.

Por intermédio do gráfico da Figura 7.1 e é possível notar que o comportamento desta variação conta com um pico e um vale, podendo apresentar valores diferentes quando lidos em momentos diferentes.

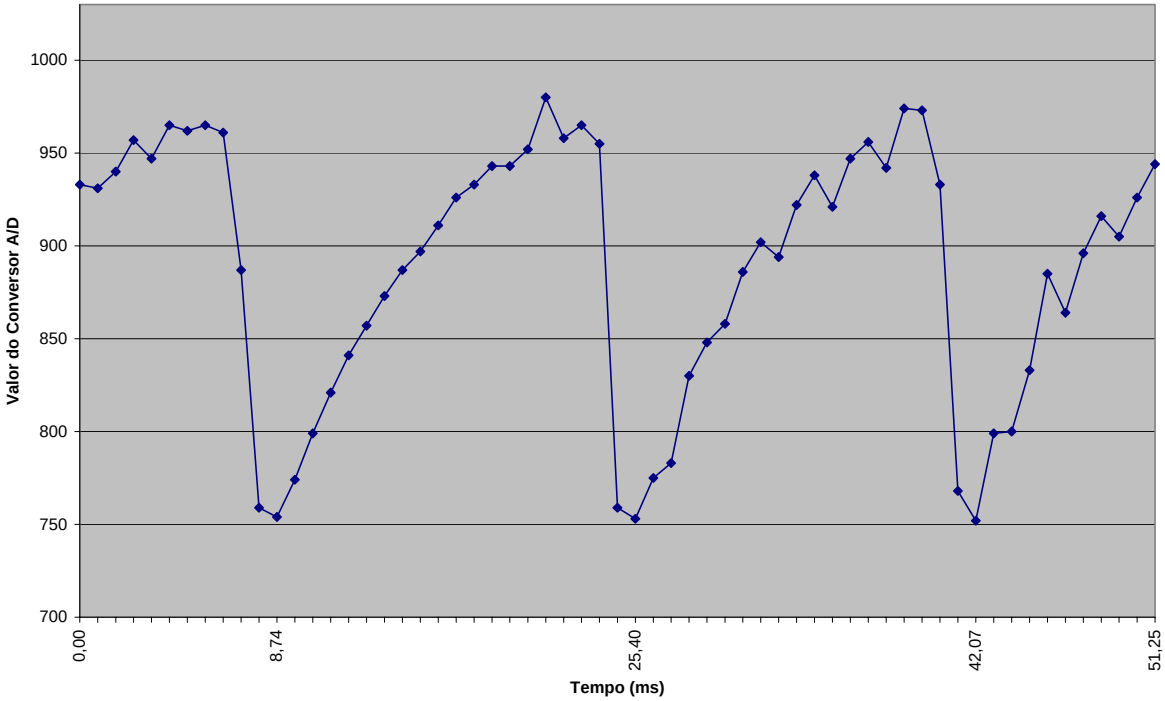


Figura 7.1 - Variação da Intensidade Luminosa do Monitor de Vídeo (cor cinza)

Com o objetivo de contornar esta situação e para não deixar o microcontrolador ocioso entre as leituras do conversor AD, foi adotado o seguinte procedimento:

a) São criadas duas variáveis globais, sendo uma, um acumulador de leituras e outra um contador da quantidade de leituras realizadas;
b) O Timer 1 foi configurado de modo a gerar uma interrupção a cada $16,\overline{66}$ milissegundos;
c) Dentro de um *loop* infinito são realizadas indefinidas leituras do valor do conversor AD;
d) A cada iteração o valor lido no conversor AD é somado à variável acumuladora e o contador de leituras realizadas é incrementado;
e) Quando uma interrupção é gerada pelo Timer 1, a função associada a esta interrupção divide o valor acumulado pelo contador, chegando a uma média das leituras realizadas;

f) Este valor calculado é então utilizado no processo de decisão dos bits aos quais representa e, quando totalizar um byte, transmitido ao computador receptor;
g) O acumulador e o contador são então zerados e a função retorna para o *loop* infinito, recomeçando o processo.

7.3. Arquivos de Cabeçalho e Configurações Gerais

A diretiva de compilador **#include** realiza a inclusão do conteúdo do arquivo especificado no ponto onde se encontra a diretiva.

No Software Embarcado são incluídos dois arquivos, conforme o código:

```
#include <16f877a.h>
#include <regs_16f87x.h>
```

Estes arquivos possuem a declaração de várias constantes específicas para o microcontrolador utilizado. Estas constantes possuem por objetivo mapear endereços de memória, pinos de I/O, opções de configuração etc.

Além da diretiva **#include**, é necessária a utilização das diretivas **#use**, **#fuse** e #**device**.

A diretiva **#use** é utilizada aqui com dois de seus principais modos de configuração: **delay** e **rs232**.

O primeiro tem por objetivo informar ao compilador qual o *clock* utilizado pelo microcontrolador e, assim, ativar as funções **delay_ms** e **delay_us**. Já **rs232** informa ao compilador o *baud rate* e os pinos usados como entrada e saída da porta serial.

Conforme o código apresentado abaixo, são configurados o *clock* de 20 MHz e uma transmissão serial de 115.200 bps utilizando o pino C6 como transmissor e o pino C7 como receptor.

```
#use delay(clock=20000000)
#use rs232(baud=115200, xmit=PIN_C6, rcv=PIN_C7)
```

Na prática o Software Embarcado não utiliza o pino de entrada (receptor) pois o microcontrolador não recebe informações do computador de destino. Como sua função é apenas enviar os dados recebidos, o único pino utilizado é o C6 como transmissor.

Já a diretiva **#fuse** é responsável por definir quais opções de configuração do microcontrolador serão utilizadas quando o mesmo for gravado (programado).

```
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
```

As opções do exemplo acima (e utilizadas no projeto) definem:

- **HS**: Indica que é utilizado um oscilador do tipo HS, que representa um cristal / ressonador de alta freqüência. Na interface projetada foi adotado um cristal de 20 MHz.
- **NOWDT**: Desativa o *Watchdog Timer*. O WDT é uma espécie de temporizador que tem como função reiniciar o microcontrolador quando ocorrer uma situação não prevista e o software “travar” em uma situação qualquer.
- **PUT**: Ativa o *Power Up Timer*. É um temporizador que garante que o microcontrolador somente comece a operar depois que a fonte de alimentação estiver estabilizada.
- **NOBROWNOUT**: Desabilita o *Brown Out Detect*. O *Brown Out Detect* força um reset do PIC sempre que a tensão de alimentação cair abaixo de 4 volts durante um intervalo de tempo maior que 100 microsegundos.
- **NOLVP**: Desativa a *Low Voltage Program*. A LVP permite que o PIC possa ser colocado em modo de gravação em nível TTL, ou seja, com 5 volts ao invés dos 13 volts típicos.

Por fim, a diretiva **#device** informa ao compilador qual o modelo do microcontrolador utilizado e algumas diretivas com as características deste microcontrolador, como por exemplo, o número de bits que será utilizado pela função **read_adc()**. Esta função é utilizada para realizar a conversão AD e sua leitura.

```
#device adc=10
```

7.4. Configuração do Conversor A/D

Na função **main()** (que em C é o ponto de entrada do programa), foram inseridos alguns comandos para configuração do módulo de conversão AD.

```
setup_adc_ports (RA0_ANALOG_RA3_RA2_REF);
setup_adc(ADC_CLOCK_INTERNAL);
set_adc_channel(0);
```

A função **setup_adc_ports** é responsável por configurar os pinos do conversor AD como analógicos, digitais ou ambos. As combinações possíveis variam entre cada modelo de microcontrolador. No exemplo apresentado acima, é informado ao compilador que utilizaremos o pino RA0 como analógico e os pinos RA3 e RA2 como referência de tensão externa para o conversor AD.

Já a função **setup_adc** configura o *clock* do conversor AD. No exemplo acima o *clock* utilizado é o próprio *clock* interno do microcontrolador.

Por fim, a função **set_adc_channel** informa qual canal será utilizado na próxima leitura **read_adc**. No caso do PIC16F877A, que possui 8 canais de conversão AD, podem ser passados como parâmetros os números entre 0 e 7 correspondentes aos canais disponíveis neste dispositivo.

Como configurado pela função **setup_adc_ports**, só é utilizado o pino RA0 configurado como canal 0. Logo é a única opção disponível para a função **set_adc_channel**.

7.5. Configuração do Timer 1

Para que o Timer 1 possa disparar uma interrupção a cada $16,\overline{66}$ milisegundos, é necessário lançar mão do conceito de *prescaler*.

O *prescaler* é responsável por dividir o *clock* interno por um número inteiro que resultará em uma freqüência menor para o dispositivo que o implementa. No caso do Timer 1 são possíveis a utilização de 2, 4, 8 ou nenhum um *prescaler*.

O Timer 1 é um contador de 16 bits, logo este timer terá seu “estouro” e conseqüente interrupção quando o mesmo atingir o valor de 65.535, pois o mesmo se inicia em zero.

Na **Equação 7.2** é demonstrado o cálculo do valor inicial de carregamento do Timer 1 a fim de obtermos a freqüência desejada:

$$Frequencia = 60Hz$$
$$Clock = 20MHz$$
$$\Pr escaler = 2$$
$$x = 2^{bits} - \left(\frac{Clock}{4 \times \Pr escaler \times Frequencia} \right) = 65.536 - \left(\frac{20.000.000}{4 \times 2 \times 60} \right) \cong 23.870$$

Equação 7.2 - Cálculo do Valor de Inicialização do Timer 1

Abaixo é apresentado o trecho de código que configura o Timer 1 e sua respectiva interrupção:

```
setup_timer_1 (T1_INTERNAL | T1_DIV_BY_2);
set_timer1(23870);
enable_interrupts (GLOBAL | INT_TIMER1);
```

A função **setup_timer_1** é utilizada aqui com dois parâmetros. O parâmetro **T1_INTERNAL** indica que o *clock* base deste timer é o próprio *clock* interno. Já o parâmetro **T1_DIV_BY_2** indica que o *prescaler* utilizado é 2, dividindo assim o *clock* interno pela metade.

Conforme calculado, o valor de 23.870 é configurado como valor inicial do Timer 1 por intermédio do comando **set_timer1**.

Por fim, a função **enable_interrupts** ativa a interrupção global do microcontorlador e a interrupção individual do Timer 1.

7.6. Sincronização com o Monitor de Vídeo

Um dos grandes problemas encontrados no desenvolvimento da interface era o de garantir que uma sucessão de leituras realizadas no intervalo de $16,\overline{66}$

milissegundos colha amostras dentro do mesmo intervalo que o monitor de vídeo está exibindo uma determinada cor.

Um exemplo de como a ausência deste sincronismo poderia acarretar em sérios problemas com a transmissão é se, por exemplo, fossemos transmitir a seguinte seqüência de bits: “0111”. Neste caso teríamos a exibição de duas cores na escala de cinza: uma para “01” e outra para “11”.

Supondo ainda que após um tempo de 8 milissegundos (aproximadamente metade do período de $16,\overline{66}$ milissegundos) de exibição da 1ª cor a interface iniciasse o processo de coleta, neste caso, a coleta se daria uma parte sobre a 1ª cor e outra parte sobre a 2ª cor, acarretando assim em uma leitura errada para uma das duas cores ou ainda para uma terceira, desde que existisse uma cor possível entre as duas envolvidas.

Dado este problema foi analisado o comportamento da cor branca, que neste projeto foi adotada como sendo a cor de não-transmissão de dados. Por intermédio da Figura 7.2 nota-se que, de certa forma a variação é periódica, sendo o seu menor valor igual ou inferior a 5 unidades do conversor A/D.

Com base nesta constatação, foi construída a rotina de sincronismo da interface com o monitor de vídeo do computador transmissor. Esta rotina consiste em um *loop* com parada quando o valor lido no conversor A/D for inferior a 7 unidades do conversor.

Após abandonar a rotina de sincronismo o Software Embarcado dá início à coleta e acúmulo das leituras do conversor A/D.

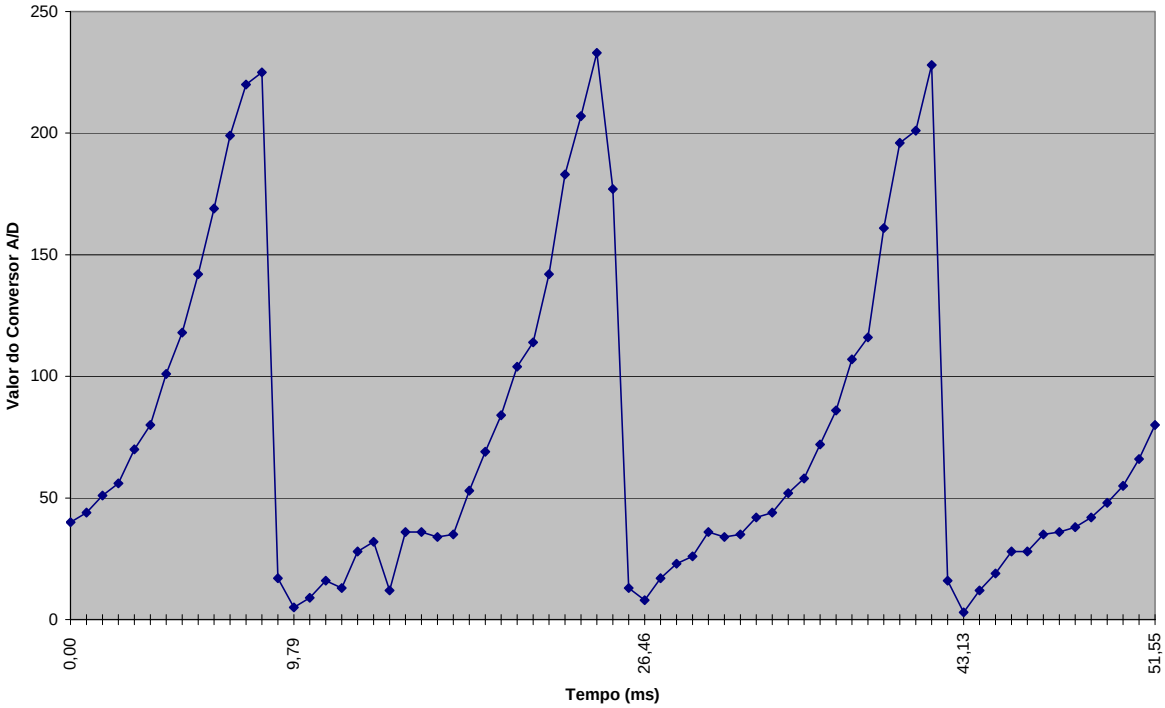


Figura 7.2 - Variação da Intensidade Luminosa do Monitor de Vídeo (cor Branca)

7.7. A Rotina de Interrupção

Quando o Timer 1 atinge o valor de 65.535 e é incrementado, o mesmo vai para o valor 0 e dispara uma interrupção, fazendo com que a execução do Software Embarcado seja desviada para a rotina de interrupção correspondente ao timer.

A rotina de interrupção do Software Embarcado realiza a divisão do acumulador de leituras do conversor A/D pelo valor do contador de leituras. O resultado desta divisão representa a média dos valores lidos no intervalo entra a última interrupção e a interrupção atual. De posse deste valor a rotina de interrupção o compara com as faixas de valores correspondentes a cada par de bits.

Ao associar o valor médio com o par de bits correspondente, este par de bits é acumulado em uma variável até que a mesma possua 8 bits, ou seja, a cada 4 disparos da interrupção é formado um byte de informação.

Assim que se atinge a quantidade de bits necessária para se formar um byte, este byte é enviado para a porta serial que será lida pelo Software Receptor.

Ao final da rotina de interrupção o contador e acumulador de leituras do conversor A/D são zerados, preparando-os para o próximo bit a ser lido.

8. CONSIDERAÇÕES FINAIS

8.1. Dificuldades Encontradas

Durante todo o projeto a maior dificuldade encontrada foi, de alguma forma, realizar leituras de uma mesma cor no monitor de vídeo com variações mínimas entre estas leituras.

Já nos primeiros testes a leitura de uma mesma cor no monitor de vídeo poderia apresentar valores diferentes e abrangendo um intervalo muito grande, reduzindo assim as possibilidades de separação das faixas que corresponderiam aos pares de bits.

Foram realizados vários experimentos, mudanças nos códigos e estratégias de leitura. Chegou-se a testar a utilização de um LDR no lugar do fotodiodo, porém o primeiro apresentou dois problemas básicos: mudança brusca entre os estados de claro e escuro e tempo de resposta inadequado às necessidade do projeto.

Após a definição do fotodiodo como componente eletrônico responsável pela captação dos sinais luminosos, foi necessária a elaboração de uma estratégia de leitura mais eficaz que a simples leitura no período pretendido.

Foi necessário adotar a estratégia de realização de várias leituras (tantas quanto forem possíveis) no intervalo de tempo correspondente ao período de atualização do monitor de vídeo. Estas leituras foram então acumuladas e a média dessas leituras é que foi utilizada na classificação do sinal capturado com os bits de dados que este representava.

Os estudos sobre os microcontroladores da linha PIC demandaram um tempo considerável entre pesquisa, testes e definições.

8.2. Resultados Obtidos

O principal resultado obtido foi a comprovação de que é possível a evolução de um projeto de transmissão de dados utilizando como transmissor apenas o monitor de vídeo de um microcomputador.

Tal comprovação é sedimentada pelo projeto físico da interface implementada e do sucesso na transmissão de uma seqüência de texto entre os dois

microcomputadores, sem que os dois possuíssem qualquer outro tipo de canal de comunicação entre eles.

Como resultados obtidos deve-se considerar as diversas contribuições metodológicas e práticas deste trabalho, como:

- Apresentação do sistema RGB de cores e uma de suas principais características: a não-linearidade;
- Descrição aprofundada dos componentes Direct X utilizados na confecção do Software Transmissor; bem como das técnicas utilizadas na atualização do monitor de vídeo;
- Sedimentação dos conceitos de paridade e de paridade combinada e sua utilização nos sistemas de transmissão de dados;
- Aprofundamento dos conceitos inerentes a comunicação serial, principalmente a do tipo assíncrona;
- Descrição da arquitetura do PIC e exposição de três de seus módulos mais úteis: o conversor A/D, do Timer 1 e a USART interna;
- Comparação entre o fotodiodo e o resistor do tipo LDR, apresentando resultados obtidos em medidas realizadas experimentalmente e seu respectivo gráfico;
- Apresentação passo a passo de como criar um canal de comunicação serial em Java por intermédio da utilização da API Java Comm da Sun;
- Enumeração de diversos trabalhos futuros que poderão ser utilizados por futuros formandos deste curso em projetos finais ou em projetos de iniciação científica.

8.3. Conclusões

O modelo realiza uma transmissão de dados do tipo texto, onde o microcomputador transmissor varia a intensidade de luz emitida pelo seu monitor CRT, de acordo com os bits que compõe o texto a ser transmitido, permitindo desta forma que uma interface, especialmente projetada com o fim de captar estas variações, possa decodificar o texto original e escrever o mesmo na porta serial do computador receptor.

O uso do monitor como transmissor pode ser útil em ambientes onde as portas de comunicação do computador não são acessíveis ao usuário, porém, a velocidade de transmissão de dados por esse método se limita à freqüência vertical do monitor (Mitsuka 2004).

Além de reafirmar a conclusão acima, o atual trabalho traz luz à possibilidade de uma melhoria contínua neste novo meio de transmissão, onde os obstáculos limitadores são superados.

O presente trabalho apresentou duas destas melhorias: o aumento no número de níveis de transmissão e a implementação da técnica de paridade combinada, a fim de diminuir os possíveis erros de transmissão.

Ambos os objetivos foram alcançados e quando antes tínhamos uma transmissão de 60 bits por segundo sem qualquer minimização de erros, agora se pode contar com uma transmissão de 120 bits por segundo e com a garantia de que a maior parte dos erros de transmissão será sanada pelo próprio sistema de transmissão.

8.4. Sugestões para Trabalhos Futuros

Diversas linhas de pesquisa podem ser criadas a partir do trabalho aqui apresentado. Dentre elas podemos destacar:

- Incrementar novamente o número de níveis de transmissão possibilitará, como possibilitou a este projeto, uma evolução na velocidade de transmissão. O número de níveis de transmissão deve ser sempre um número na potência de 2. Neste projeto foi apresentado a transmissão em 2^2 níveis; o próximo passo seria 2^3 ou 2^4 níveis;
- Utilização da transmissão paralela, ou seja, a divisão da área do monitor de vídeo em partes iguais e a leitura destas partes por intermédio de vários sensores fotoelétricos, um para cada parte da tela;
- Confeccionar o Software Transmissor em uma versão para a internet, via WWW. A dificuldade aqui é superar os limites impostos por esta tecnologia, tanto em performance quanto em limitações de acesso direto ao hardware;

- Realizar transmissão com monitores LCD é uma linha de pesquisa interessante devido ao fato destes monitores estarem ganhando espaço nos lares brasileiros. A maior dificuldade aqui será superar o fato dos monitores LCD emitirem radiações muito baixas e, consequentemente, muito parecidas com as cores extremas: preto e branco;
- Diminuir o tamanho físico da interface a fim de acoplá-la a algum outro dispositivo. Sua miniaturização pode levar a interface a ser inserida em PDA’s e celulares;
- Implementar a comunicação via USB da interface com o computador receptor. O Software Receptor deve ser adaptado a fim de ler a porta USB do micro e a interface deve possuir a capacidade de enviar informações neste meio.

REFERÊNCIAS BIBLIOGRÁFICAS

ALVES. Luiz. **Protocolos para Redes de Comunicações de Dados**. Atlas, São Paulo, 1991.

GIORDAN, M. **O Ensino de Ciências nos Tempos da Internet. em Ciência, Ética e Cultura na Educação**. Chassot e Oliveira (orgs), Ed. Unisinos, São Leopoldo, RS, 1998.

KEISER, G. **Local Área Network.** McGraw-Hill Book Co., 1989.

MALVINO, Albert Paul. **Eletrônica: volume 1**. 4º edição. Ed. Makron Books, São Paulo, 1995.

Microchip Inc. **Data Sheet do PIC16F877A.** http://ww1.microchip.com/downloads/en/DeviceDoc/39582b.pdf

Microsoft Corporation. **Documentação do DirectX 9c.** http://msdn.microsoft.com/directx/

MITSUKA, Tiago Almeida. **Transmissão Alternativa de Dados**. Centro Universitário de Brasília, Brasília, 2004.

NASCIMENTO, Juarez do. **Telecomunicações**. 2ª Edição. Ed. Makron Books, 2000.

PEREIRA, Fábio. **Microcontroladores PIC - Programação em C**. 3º Edição. Editora: Érica, 2004.

SILVEIRA, Jorge Luiz. **Comunicação de Dados e Sistemas de Teleprocessamento**. Ed. Makron Books, São Paulo, 1991.

SOARES, Luiz Fernando Gomes **Redes de Computadores: das LANs, MANs e WANs às redes ATM.** 2º edição revisada e ampliada. Ed. Campus, Rio de Janeiro, 1995.

SOUZA, David Jose de. **Conectando o Pic: Recursos Avançados**. 2º Edição. Editora: Érica

Sun Microsystems. **Documentação da API do Java 1.5** http://java.sun.com/j2se/1.5.0/docs/api/

TAFNER, Malcon Anderson; LOESCH, Cláudio; STRINGARI, Sérgio. **Comunicação de Dados Usando Linguagem C**.FURB, Blumenal, 1996

TAROUCO, Liane M. R. **Redes de Comunicação de Dados**. 3º edição, Editora LTC, Rio de Janeiro, 1985

TORRES, Gabriel. **Hardware Curso Completo**. 2º Edição. Ed. Axcel Books, Rio de Janeiro, 1998.

WHITE, Ron. **COMO FUNCIONA O COMPUTADOR III**. Quark, 1997.

Apêndice A – Código Fonte Completo do Software Transmissor

```
#include <stdafx.h>
#include <windows.h>
#include <stdio.h>
#include <ddraw.h>
#include <iostream>
#include <cstdlib>
#include <math.h>

/*--- Variáveis Globais--*/
HWND           g_hMainWnd;
HINSTANCE      g_hInst;
LPDIRECTDRAW7        g_pDD = NULL;

LPDIRECTDRAWSURFACE7     g_pDDSFront = NULL;
LPDIRECTDRAWSURFACE7     g_pDDSBack = NULL;

int cNull, c00, c01, c10, c11;
char* s;
bool *bits;
int bitLen;

/*---Prioridade--------*/
HANDLE hProcess, hThread;
DWORD ClassPriority;
int ThreadPriority;

/*---Protótipos---*/

HWND InitWindow (int iCmdShow);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam);
void ProcessIdle();
int InitDirectDraw();
void CleanUp();
void DDSFill( LPDIRECTDRAWSURFACE7 pDDS, int cor);
DWORD CreateRGB( int r, int g, int b );

//Função de Entrada do Programa C++
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{

     if (__argc < 5) {
          CleanUp();
          MessageBox(g_hMainWnd, "Número de parâmetros
inválido.\nFavor digitar: Transmissor [cNull c00 c01 c10 c11
{Menssagem}].", "Error", MB_OK | MB_ICONEXCLAMATION);
          return 0;
     }
```

```
  int freq;
  HDC hdc;
  hdc = GetDC(NULL);
  freq = GetDeviceCaps(hdc, VREFRESH);
  ReleaseDC(NULL, hdc);

  char  buffer[200];

     MessageBox(g_hMainWnd, "Prepare a Interface." , "Atenção",
MB_OK | MB_ICONEXCLAMATION);

     cNull     = atoi(__argv[1]); //valor Nulo
     c00  = atoi(__argv[2]); //valor 00
     c01  = atoi(__argv[3]); //valor 01
     c10  = atoi(__argv[4]); //valor 10
     c11  = atoi(__argv[5]); //valor 11
     s    = __argv[6];

     int bitParidade, bitWordParidade = 0, qtletras = 0;
     int len = strlen(s);
     int wordParidade [8] = {0, 0, 0, 0, 0, 0, 0, 0}; //Palavra
de paridade

     float qtdWordParidade = len;
     qtdWordParidade = ceil(qtdWordParidade/7);

     int j = bitLen = (len + qtdWordParidade) *8; //Bits totais
da transmissão
     bits = new bool[j]; //Matriz de bits para transmissão

     int restoWordParidade = len % 7;

     int qtdWritedWord = 0; //Quantidade de palavras de
paridade adicionadas

     j = j - 8; //Reserva espaço para palavra de paridade

     for(int i=--len; i>=0; i--){
          int dado = s[i];
          if (dado <= 127) { //Transmite apenas caracteres
ASCII de 7 bits
               qtletras++;
               bitParidade = 0;

               for(int k=1; k<8; k++){ //Insere cada bit do
caracter na matriz de bits
                    bits[--j] = (dado % 2 == 0) ? false : true;
                    if (bits[j]) {
                         bitParidade++;
```

```
                         wordParidade [k-1]++; //Monta a
palavra de paridade
                    }
                    dado=dado/2;
               }
               bits[--j] = (bitParidade % 2 == 0) ? false :
true; //Insere o bit de paridade na matriz de bits
          }

          if (qtletras == 7 || qtletras == restoWordParidade) {
//Insere a palavra de paridade na matriz de bits
               restoWordParidade = 0;
               qtdWritedWord++;
               int l = j + ((qtletras + 1) * 8);
               qtletras = 0;
               for (int k=0; k<7; k++) { //Insere os bits da
palavra de paridade na matriz de bits
                    bits[--l] = (wordParidade [k] % 2 == 0) ?
false : true;
                    if (bits[l]) {
                         bitWordParidade++;
                    }
                    wordParidade [k] = 0;
               }
               bits[--l] = (bitWordParidade % 2 == 0) ? false :
true; //Insere o bit de paridade da palavra de paridade na
matriz de bits
               j = j - 8;
          }
     }

     g_hInst = hInstance;
     g_hMainWnd = InitWindow(nCmdShow);

     if(!g_hMainWnd)
          return -1;

     if(InitDirectDraw() < 0)
     {
          CleanUp();
          MessageBox(g_hMainWnd, "Could start DirectX engine in
your computer. Make sure you have at least version 7 of
DirectX installed.", "Error", MB_OK | MB_ICONEXCLAMATION);
          return 0;
     }

     hProcess = GetCurrentProcess();
     hThread = GetCurrentThread();

     ClassPriority = GetPriorityClass(hProcess);
     ThreadPriority = GetThreadPriority(hThread);
```

```
     SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
     SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);

     g_pDD->WaitForVerticalBlank(DDWAITVB_BLOCKEND, 0);
     g_pDD->WaitForVerticalBlank(DDWAITVB_BLOCKEND, 0);

     while( true ) {
          MSG msg;
          if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) ) {
               if( msg.message == WM_QUIT )
                    break;
               TranslateMessage( &msg );
               DispatchMessage( &msg );
          }else{
               //for (int j = 0; j <60; j++) {
                    //g_pDD-
>WaitForVerticalBlank(DDWAITVB_BLOCKEND, 0);
               //}

               ProcessIdle();
          }
     }

     SetPriorityClass(hProcess, ClassPriority);
     SetThreadPriority(hThread, ThreadPriority);

     CleanUp();

     delete [] bits;
     return 0;
}

//Realiza o Preenchimeto e Troca das Cores
void ProcessIdle()
{
     HRESULT hRet;
     static BOOL cor = false;
     static int pos=0;
     static cont=0;
     static BOOL teste = false;

     if (cor) {
          if(bits[pos]){
               if(bits[++pos]) {
                    DDSFill( g_pDDSBack, c11 );
               } else {
                    DDSFill( g_pDDSBack, c10 );
               }
          }else{
               if(bits[++pos]) {
```

```
                    DDSFill( g_pDDSBack, c01 );
               } else {
                    DDSFill( g_pDDSBack, c00 );
               }
          }

          if(++pos >= bitLen){
               cor=false;
               pos = 0;
          }

     } else {
          DDSFill( g_pDDSBack, cNull );
     }

     if(cont++ == 200){
          cor = true;
     }

     hRet = g_pDDSFront->Flip(NULL, DDFLIP_WAIT );

}

//Inicialização da Janela da Aplicação Windows em C++
HWND InitWindow(int iCmdShow)
{
     HWND hWnd;
     WNDCLASS wc;
     wc.style = CS_HREDRAW | CS_VREDRAW;
     wc.lpfnWndProc = WndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = g_hInst;
     wc.hIcon = LoadIcon(g_hInst, IDI_APPLICATION);
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH )GetStockObject(BLACK_BRUSH);
     wc.lpszMenuName = TEXT("");
     wc.lpszClassName = TEXT("Basic DD");
     RegisterClass(&wc);
     hWnd = CreateWindowEx(
          WS_EX_TOPMOST,
          TEXT("Basic DD"),
          TEXT("Basic DD"),
          WS_POPUP,
          0,
          0,
          GetSystemMetrics(SM_CXSCREEN),
          GetSystemMetrics(SM_CYSCREEN),
          NULL,
          NULL,
          g_hInst,
```

```
          NULL);
     ShowWindow(hWnd, iCmdShow);
     UpdateWindow(hWnd);
     SetFocus(hWnd);
     ShowCursor(false);
     return hWnd;
}

//Trata as Menssagens do Windows
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
     switch (message)
     {
          case WM_KEYDOWN:
               if(wParam == VK_ESCAPE)
               {
                    PostQuitMessage(0);
                    return 0;
               }
               break;
          case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
     }

     return DefWindowProc(hWnd, message, wParam, lParam);
}

//Inicializa o DirectDraw
int InitDirectDraw()
{
     DDSURFACEDESC2 ddsd;
     DDSCAPS2 ddscaps;
     HRESULT hRet;

     hRet = DirectDrawCreateEx(NULL, (VOID**)&g_pDD,
IID_IDirectDraw7, NULL);
     if( hRet != DD_OK )
          return -1;

     hRet = g_pDD->SetCooperativeLevel(g_hMainWnd,
DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
     if( hRet != DD_OK )
          return -2;

     hRet = g_pDD->SetDisplayMode(640, 480, 32, 0, 0);
     if( hRet != DD_OK )
          return -3;

     ZeroMemory(&ddsd, sizeof(ddsd));
```

```
     ddsd.dwSize = sizeof(ddsd);
     ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
     ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
DDSCAPS_FLIP | DDSCAPS_COMPLEX;
     ddsd.dwBackBufferCount = 1;

     hRet = g_pDD->CreateSurface(&ddsd, &g_pDDSFront, NULL);
     if( hRet != DD_OK )
          return -1;

     ZeroMemory(&ddscaps, sizeof(ddscaps));

     ddscaps.dwCaps = DDSCAPS_BACKBUFFER;

     hRet = g_pDDSFront->GetAttachedSurface(&ddscaps,
&g_pDDSBack);
     if( hRet != DD_OK )
          return -1;

     return 0;
}

//Libera os Objetos Instanciados
void CleanUp()
{
     if(g_pDDSBack)
          g_pDDSBack->Release();
     if(g_pDDSFront)
          g_pDDSFront->Release();
     if(g_pDD)
          g_pDD->Release();
}


//Preenche a Superfície
void DDSFill ( LPDIRECTDRAWSURFACE7 pDDS, int corRGB)
{
     HRESULT hr;
     DDBLTFX ddbfx;

     if (pDDS == NULL)
          return;

     ddbfx.dwSize = sizeof( ddbfx );
     ddbfx.dwFillColor = CreateRGB(corRGB, corRGB, corRGB);

     hr = pDDS->Blt( NULL, NULL, NULL, DDBLT_WAIT |
DDBLT_COLORFILL , &ddbfx );
}
```

```
//Transforma o valor RGB em um endereço da paleta
DWORD CreateRGB( int r, int g, int b )
{
     return (r<<16) | (g<<8) | (b); // 32 bits
}
```

Apêndice B – Código Fonte Completo do Software Receptor

```
//*************************************************************
// Interface IPublisher
//
// Interface que deve ser implementada pela classe Receptor
// com a finalidade de prover os métodos de inclusão e
// exclusão dos objetos da lista de interessados na porta
// serial e do método de atualização de todos os objetos
// cadastrados

// João Paulo Barbosa Fernandes
//*************************************************************

public interface IPublisher {
     public void attach(ISubscriber sub);
     public void detach(ISubscriber sub);
     public void updateAll();
}


//*************************************************************
// Interface ISubscriber
//
// Interface que deve ser implementada pelas classes que
// desejam ser notificadas pelo objeto publicador
//
// João Paulo Barbosa Fernandes
//*************************************************************

public interface ISubscriber {
     void update();
}


//*************************************************************
// Classe Receptor
//
// Classe responsável por estabelecer a comunicação com a
// porta serial e notificar os objetos interessados quando da
// existência de novos dados. Implementa a interface
// IPublisher
//
// João Paulo Barbosa Fernandes
//*************************************************************

import java.io.IOException;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.TooManyListenersException;
```

```
import javax.comm.CommPortIdentifier;
import javax.comm.PortInUseException;
import javax.comm.SerialPort;
import javax.comm.SerialPortEvent;
import javax.comm.SerialPortEventListener;
import javax.comm.UnsupportedCommOperationException;
import javax.swing.*;
import java.awt.event.*;
import java.util.*;

import java.util.LinkedList;

public class Receptor implements Runnable,
SerialPortEventListener, IPublisher {

     // Lista de Subscribes para aviso
     private LinkedList listOfSubscribes = new LinkedList();

     // Garante que apenas uma instância desta classe seja
criada
     public static boolean temInstancia = false;
     public static Receptor receptor;

     public static final int CARACTER = 0, BIT = 1;
     private String ultimoDadoRecebido;
     public boolean dadoDisponivel = false;
     static CommPortIdentifier portId;
     static Enumeration portList;
     InputStream inputStream;
     public static SerialPort serialPort;
     Thread readThread;
     public static int opcaoRecebimento = 0;

     private int bytes;
     private byte[] readBuffer = new byte[2048];

     public LinkedList buffer;

     // Construtor da classe
     public Receptor() {

          try {
               serialPort = (SerialPort)
portId.open("Receptor", 2000);
          } catch (PortInUseException e) {
               JOptionPane.showMessageDialog(null,
e.getMessage());
          }

          try {
               inputStream = serialPort.getInputStream();
```

```
            } catch (IOException e) {

        JOptionPane.showMessageDialog(null,e.getMessage());
            }

            try {
                serialPort.addEventListener(this);
            } catch (TooManyListenersException e) {

        JOptionPane.showMessageDialog(null,e.getMessage());
            }

            serialPort.notifyOnDataAvailable(true);

            try {
                serialPort.setSerialPortParams(115200,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
            } catch (UnsupportedCommOperationException e) {
                JOptionPane.showMessageDialog(null,
    e.toString());
            }

            buffer = new LinkedList();

            readThread = new Thread(this);
            readThread.start();
        }


        // Método de entrada do Thread
        public void run() {
            try {
            Thread.currentThread().sleep(20000);
            } catch (InterruptedException e) {}
        }

        // Trata os eventos da porta serial
        public void serialEvent(SerialPortEvent event) {
            switch(event.getEventType()) {
                case SerialPortEvent.BI:
                case SerialPortEvent.OE:
                case SerialPortEvent.FE:
                case SerialPortEvent.PE:
                case SerialPortEvent.CD:
                case SerialPortEvent.CTS:
                case SerialPortEvent.DSR:
                case SerialPortEvent.RI:
                case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
                    break;
```

```
                case SerialPortEvent.DATA_AVAILABLE:
                     readData();
                     dadoDisponivel = true;
                     break;
           }
     }

     public String getUltimoDadoRecebido() {
          dadoDisponivel = false;
          return ultimoDadoRecebido;
     }

     //Métodos da Interface Publisher

     public void attach(ISubscriber subscriber) {
          listOfSubscribes.add(subscriber);
     }
     public void detach(ISubscriber subscriber) {
          listOfSubscribes.remove(subscriber);
     }
     public void updateAll() {
          Iterator it = listOfSubscribes.iterator();

          while(it.hasNext()) {
                ISubscriber subscriber = (ISubscriber)
it.next();
                subscriber.update();
          }

     }


     // Método de criação de instância do Receptor
     public static Receptor instaciarReceptor () {
          if (!temInstancia) {
                receptor = null;

                Object[] possiveisValores =
{"COM1","COM2","COM3","COM4"};
                String valorSelecionado =
                     (String)
JOptionPane.showInputDialog(null,"Porta", "Selecione a porta",
JOptionPane.INFORMATION_MESSAGE, null, possiveisValores,
possiveisValores[0]);

                if(valorSelecionado != null &&
!valorSelecionado.equals("")){
                     portList =
CommPortIdentifier.getPortIdentifiers();
                     while (portList.hasMoreElements()) {
```

```
                         portId = (CommPortIdentifier)
portList.nextElement();
                         if (portId.getPortType() ==
CommPortIdentifier.PORT_SERIAL) {
                              if
(portId.getName().equals(valorSelecionado)) {
                                   receptor = new Receptor();
                              }
                         }
                    }

                    if(serialPort == null){

     JOptionPane.showMessageDialog(null,"Porta não existe");
                         System.exit(0);
                    }
               }else{
                    System.exit(0);
               }
          }
          temInstancia = true;
          return receptor;
     }

     // Leitura da porta serial e chamada dos assinantes
     public void readData()
     {
          String str = "", stracm = "";
          try{
               while (inputStream.available() > 0 ){
                    bytes = inputStream.read(readBuffer);
                    if (bytes > 0){
                         if (bytes > readBuffer.length) {

     System.out.println(serialPort.getName()+": Input buffer
overflow!");
                         }
                         updateAll();
                    }

               }

          }
          catch (IOException ex) {
               System.out.println(serialPort.getName()+ ":
Cannot read input stream");
          }
     }


     // Retorna o texto recebido
```

```
     public String getTextoPuro() {
          return new String(readBuffer, 0, bytes);
     }

}

//*************************************************************
// Classe SoftwareReceptor
//
// Classe responsável por exibir os dados lidos da porta
// serial no computador receptor. Faz uso da classe Receptor
// para acesso a porta serial. Implementa a interface
// ISubscriber
//
// João Paulo Barbosa Fernandes
//*************************************************************

import java.awt.BorderLayout;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JScrollPane;
import java.awt.GraphicsConfiguration;
import java.awt.HeadlessException;

import java.awt.event.*;
import java.awt.*;

import javax.swing.JFrame;
import java.awt.FlowLayout;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.math.*;;

public class Receber extends JFrame implements ActionListener,
ISubscriber {

     private JPanel jContentPane = null;
     private static String dadosAcumulados = "";

     JButton btSair;
     JComboBox combo;
     Receptor rc;

    int i;

     private JScrollPane jScrollPane = null;

     private JTextArea jTextArea = null;
```

```
     // Construtor da classe
     public Receber() throws HeadlessException {
          super();
          initialize();
     }

     // Inicialização dos objetos da classe
     private void initialize() {

          this.setSize(541, 574);
          this.setContentPane(getJContentPane());
          this.setTitle("Receber");

          setVisible(true);

          rc = Receptor.instaciarReceptor();

     this.setDefaultCloseOperation(javax.swing.WindowConstants.
DISPOSE_ON_CLOSE);
               this.setCursor(new
java.awt.Cursor(java.awt.Cursor.DEFAULT_CURSOR));
               this.setSize(new java.awt.Dimension(123,228));
          rc.attach(this);


     }

     // Inicialização do JContentPane
     private JPanel getJContentPane() {
          if (jContentPane == null) {
               jContentPane = new JPanel();
               jContentPane.setLayout(null);
               jContentPane.add(getJScrollPane(), null);
          }
          return jContentPane;
     }


     // Inicialização do JSrollPane
     private JScrollPane getJScrollPane() {
          if (jScrollPane == null) {
               jScrollPane = new JScrollPane();
               jScrollPane.setBounds(new
java.awt.Rectangle(4,6,521,529));
               jScrollPane.setAutoscrolls(true);
               jScrollPane.setViewportView(getJTextArea());
          }
          return jScrollPane;
     }
```

```
     // Inicializaçao da JTextArea
     private JTextArea getJTextArea() {
          if (jTextArea == null) {
               jTextArea = new JTextArea();
          }
          return jTextArea;
     }


     // Método da interface ISubscriber
     public void update() {
          char[] bloco;
                dadosAcumulados += rc.getTextoPuro();
          while (dadosAcumulados.length() > 7) {
               bloco = dadosAcumulados.substring(0,
8).toCharArray();
               getJTextArea().append(validaParidade(bloco));
                        if (dadosAcumulados.length() == 8) {
                            dadosAcumulados = "";
                        } else {
                            dadosAcumulados =
dadosAcumulados.substring(8);
                        }
          }
     }


     // Validação da paridade
     private String validaParidade(char[] bloco) {
          int bitParidade;
          int row, col, LRCDif = 0, VRCDif = 0;

          for(int i=0; i < 8; i++) {
               VRCDif ^= bloco[i];

               bitParidade = 0;
               for(int j=0; j < 8; j++) {
                    bitParidade += (bloco[i] &
(char)(Math.pow(2, j)))/(Math.pow(2, j));
               }
               LRCDif += (char) ((bitParidade%2) * Math.pow(2,
i));

               bloco[i] = (char) (bloco[i] & (char) 127);
          }

                if (LRCDif > 0 && VRCDif > 0) {

                    row = (int) (Math.log(LRCDif) /
Math.log(2));
```

```
                    col = (int) (Math.log(VRCDif) /
Math.log(2));

                    if (LRCDif == Math.pow(2, row) && VRCDif
== Math.pow(2, col)) {
                                    bloco[row] ^= VRCDif;
                    }
                }

          return String.valueOf(bloco).substring(0, 7);
     }

     public void actionPerformed(ActionEvent e) {}
}
```

Apêndice C – Código Fonte Completo do Software Embarcado do Microcontrolador

```
#include  <16f877a.h>
#device adc=10
#use delay(clock=20000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#use rs232(baud=115200, xmit=PIN_C6, rcv=PIN_C7)
#include <regs_16f87x.h>

#define INICIO_TIMER 23870

//Variáveis Globais
static int8 bits = 0, caractere = 0;
static int16 valor = 0, contador = 0;
static int32 soma = 0;

//Rotina da Interrupção do Timer1
#int_timer1
void enviaDado ()
{
     set_timer1(INICIO_TIMER + get_timer1());

     valor = (soma/contador);

     if (valor <= 100) {
          soma = 0;
          contador = 0;
          return;
     }

     if (valor <= 400) {
          shift_left(&caractere, 1, 0);
          shift_left(&caractere, 1, 0);
     }
     else if (valor <= 600) {
          shift_left(&caractere, 1, 0);
          shift_left(&caractere, 1, 1);
     }
     else if (valor <= 800) {
          shift_left(&caractere, 1, 1);
          shift_left(&caractere, 1, 1);
     }
     else {
          shift_left(&caractere, 1, 1);
          shift_left(&caractere, 1, 0);
     }

     bits=bits + 2;
```

```
     if (bits == 8) {
          putc(caractere);
          bits = 0;
     }

     soma = 0;
     contador = 0;

}

//Ponto de Entrada do Programa C
void main() {
     setup_ADC_ports (RA0_ANALOG_RA3_RA2_REF);
     setup_adc(ADC_CLOCK_INTERNAL);
     set_adc_channel(0);

     setup_timer_1 (T1_INTERNAL | T1_DIV_BY_8);
     set_timer1(0);

     while (read_adc() > 7) {}

     set_timer1(INICIO_TIMER);

     enable_interrupts (GLOBAL);
     enable_interrupts (INT_TIMER1);

     while (true) {
          soma += read_adc();
          contador++;
     }
}
```

Apêndice D – Fotos Reais da Interface

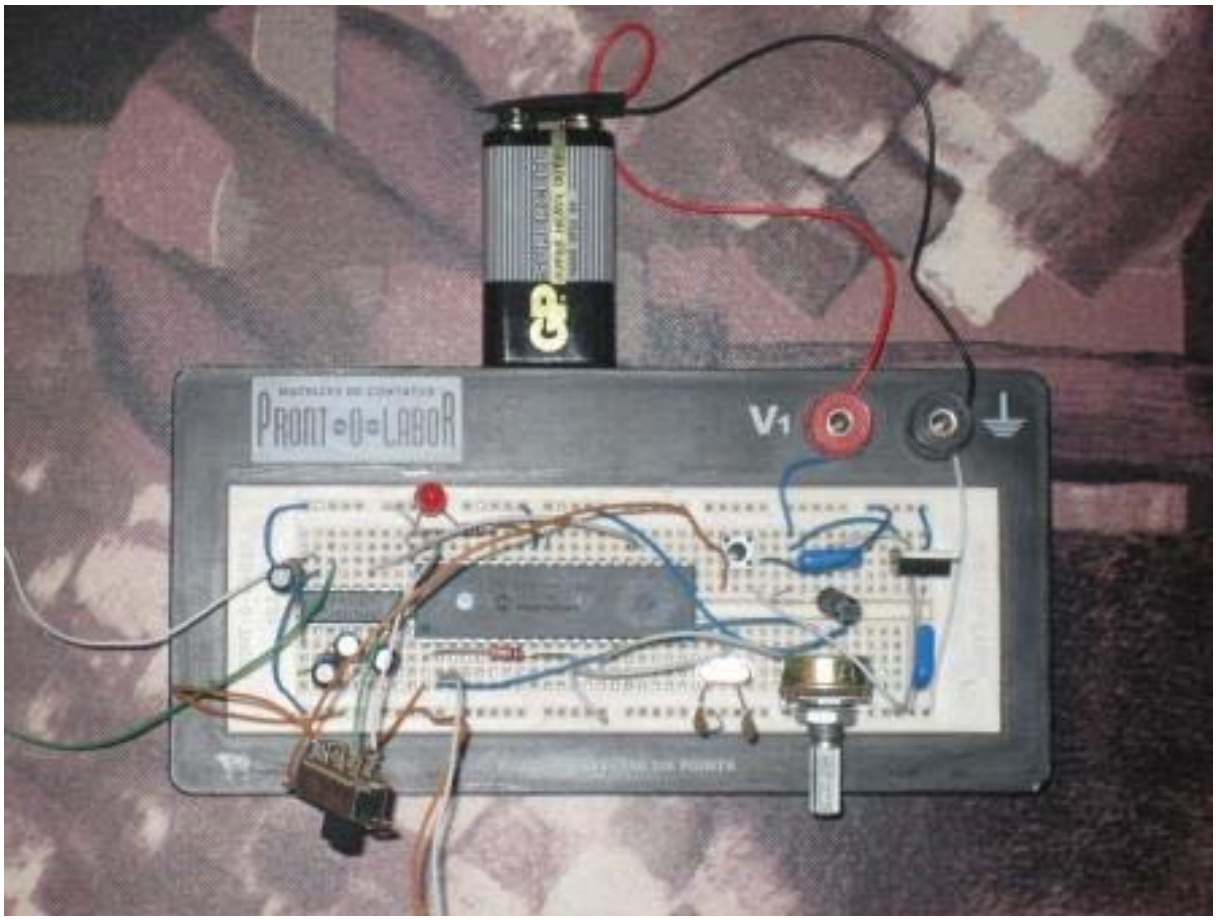


Figura A - Interface em tempo de projeto

Figura B – Receptor conectado ao monitor de vídeo em tempo de projeto

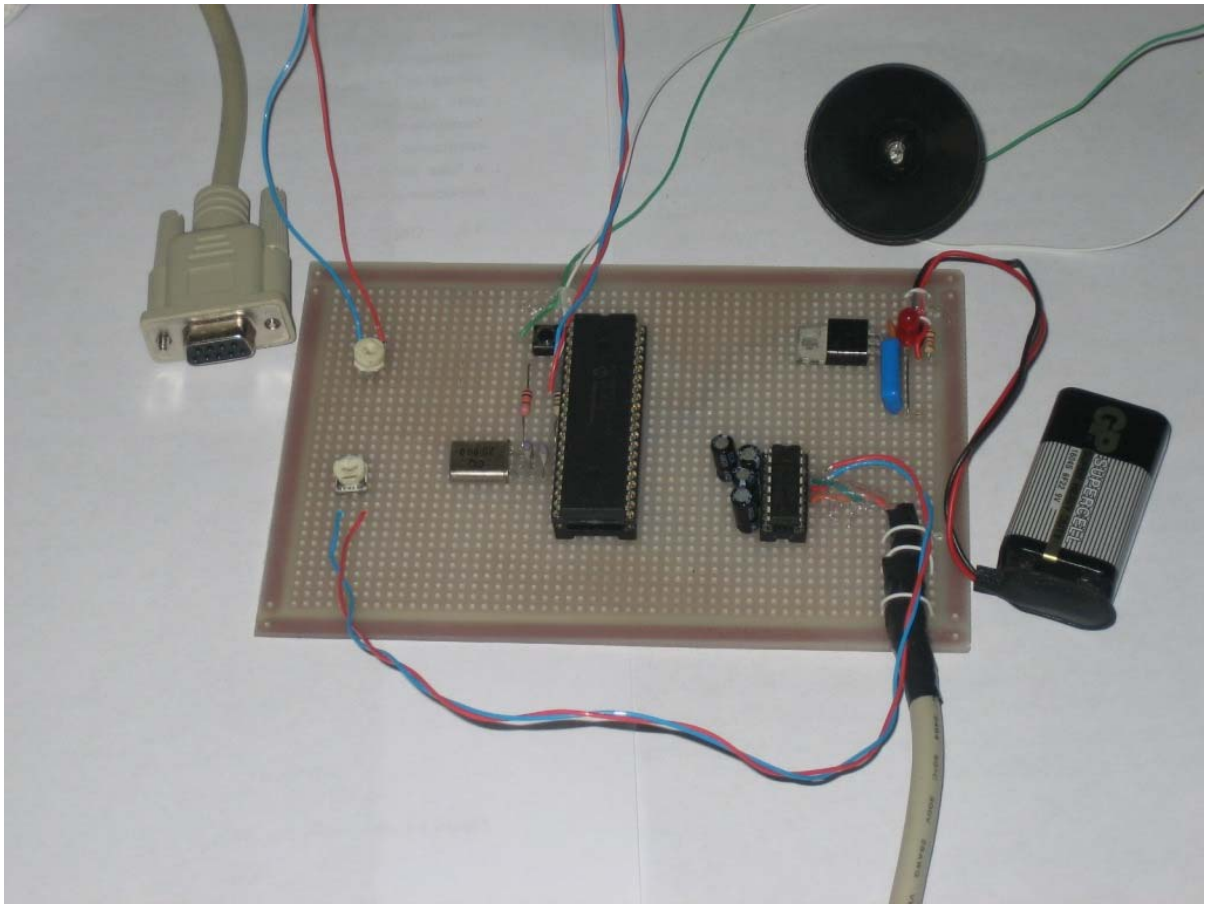


Figura C – Interface (versão final)

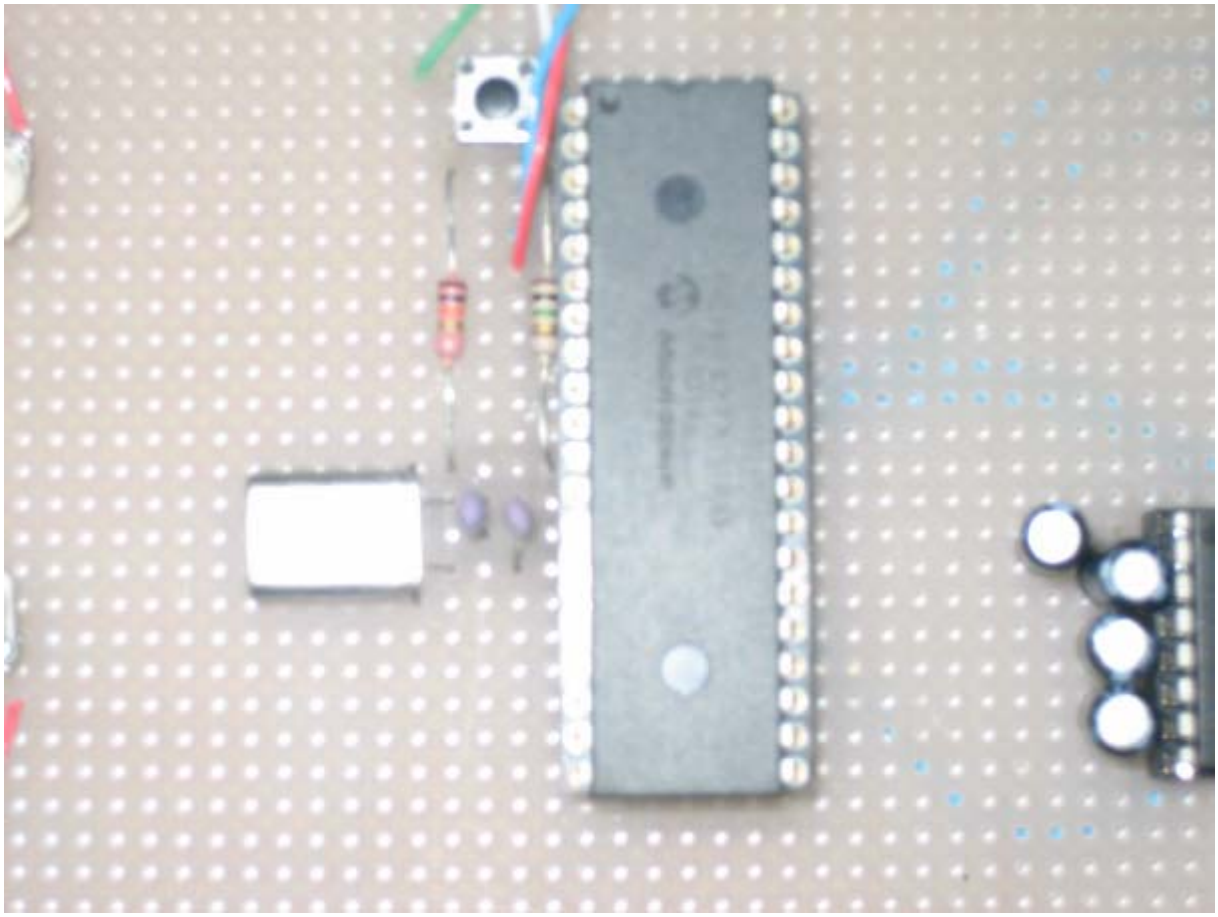


Figura D – Microcontrolador, cristal e botão de reset

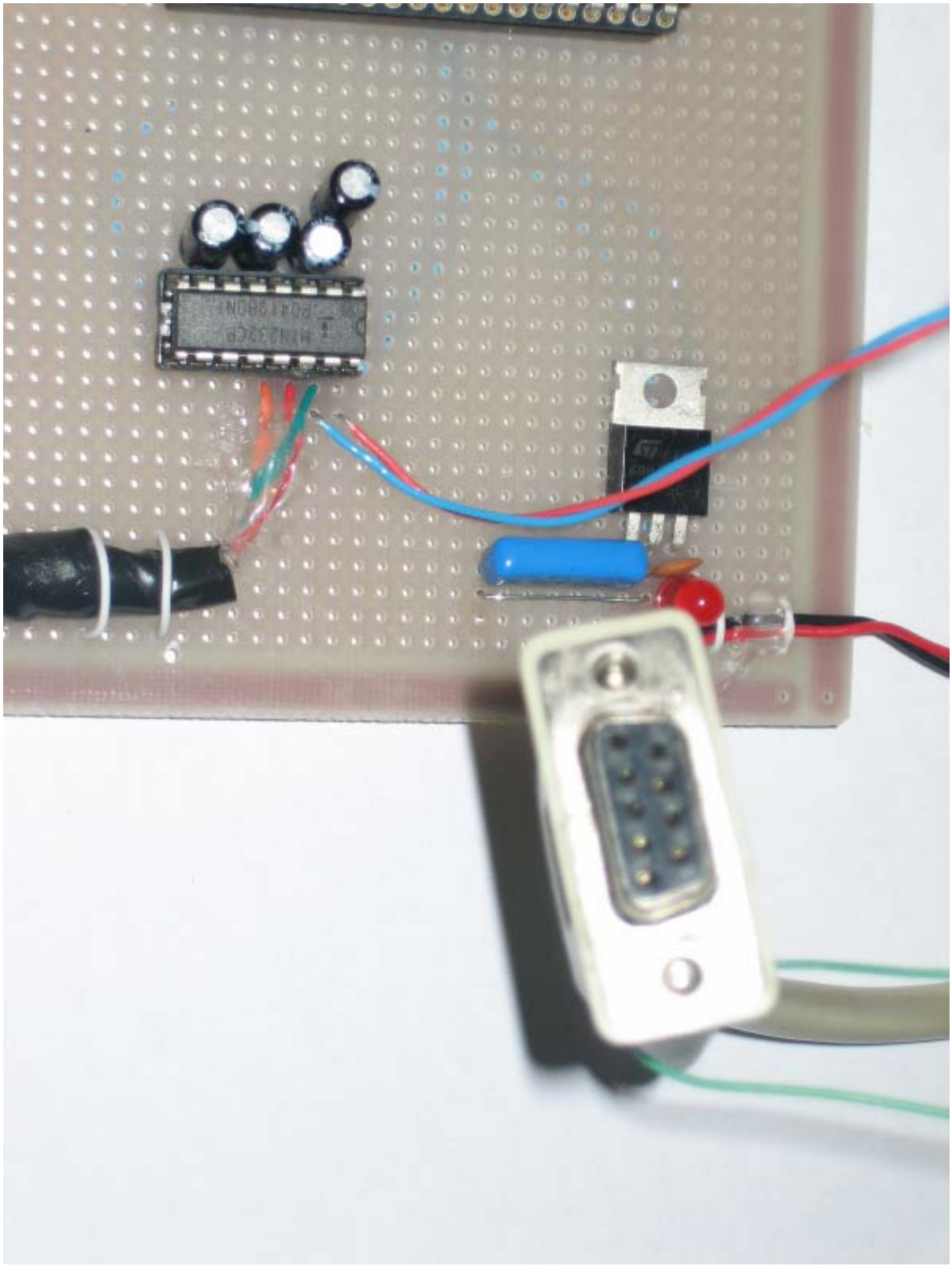


Figura E – MAX 232, capacitores e conector DB9

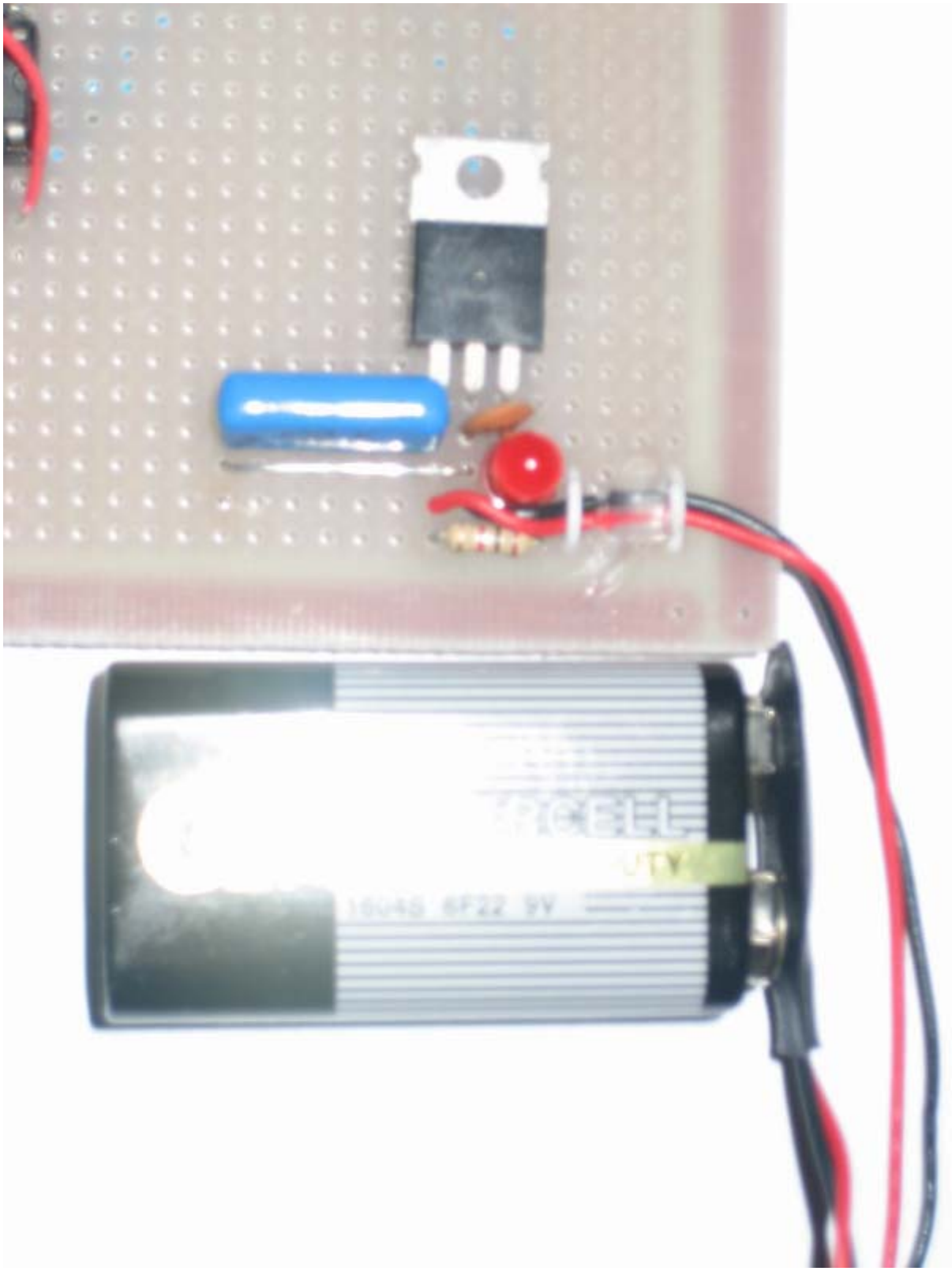


Figura F – LM 7805, capacitores, led de power e bateria

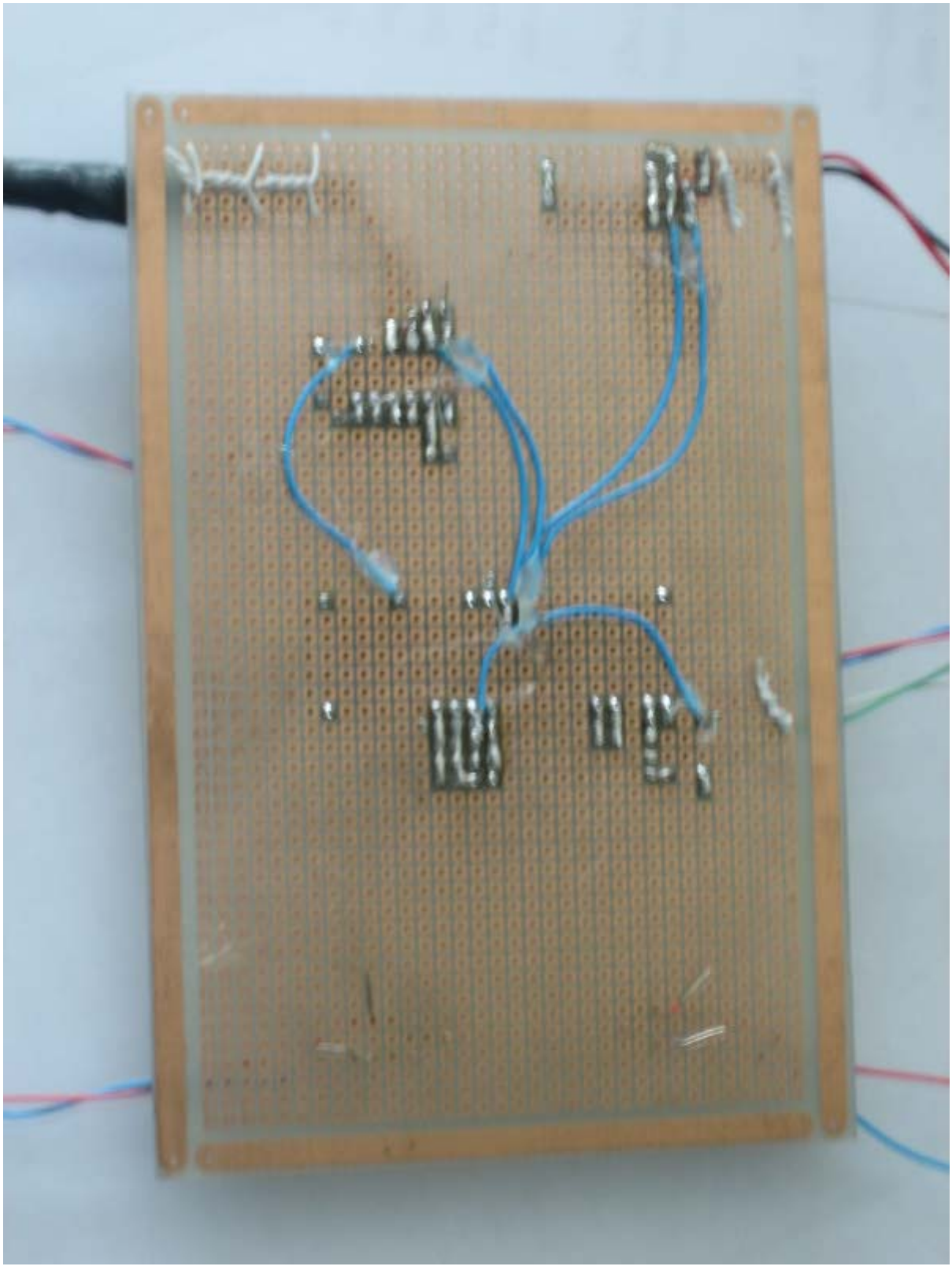


Figura G – Visão traseira da Interface

Apêndice E – Diagrama Elétrico Completo da Interface