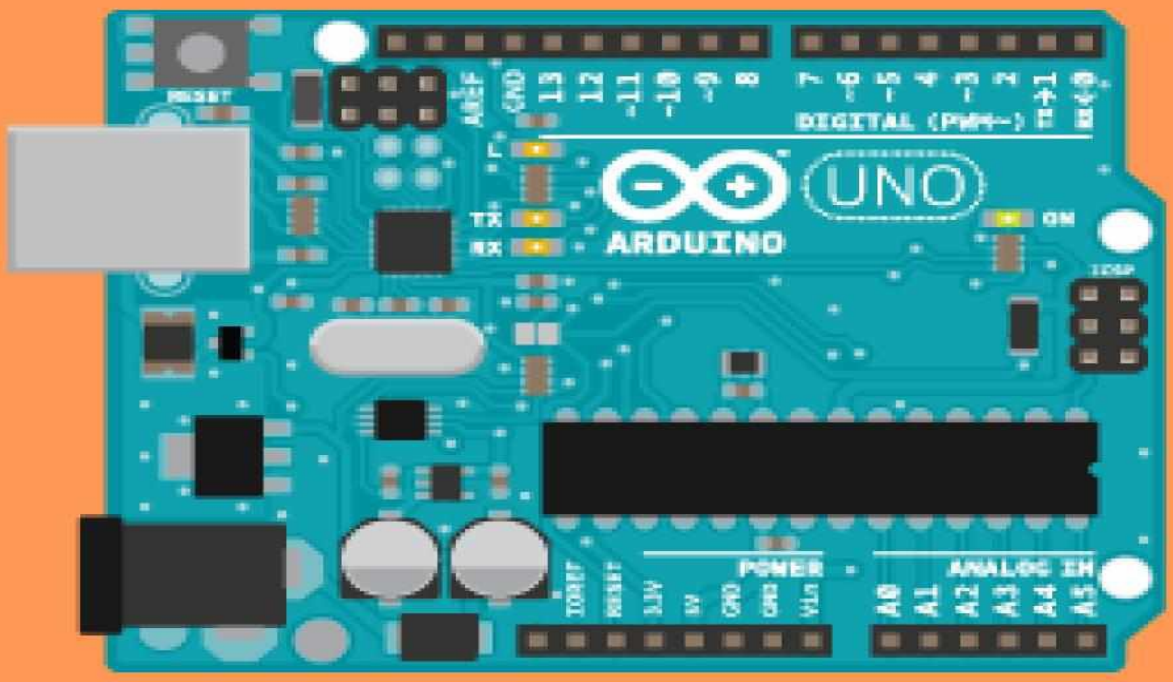
Experimentos com o ARDUINO

Edição 2.1



João Alexandre Silveira

2015

João Alexandre da Silveira
ordemnatural@outlook.com

Experimentos com o Arduino

Monte seus próprios projetos com o Arduino utilizando as linguagens C e Processing

Prefácio

Conheci o João Alexandre em junho de 1976 quando começamos a trabalhar na Embratel, mais precisamente na Estação Terrena de Comunicações por Satélites de Tanguá, no Rio de Janeiro. Por uma destas sincronicidades da vida, que costumam chamar destino, caímos na mesma turma de operações da Unidade de Rastreamento e Controle dos satélites Intelsat e logo descobrimos que ambos tínhamos sido mordidos pela mosca azul da Eletrônica experimental.

Em 1980, se a memória não me trai, escrevi meu primeiro artigo técnico para a saudosa revista "mamãe" Antenna. Ele se entusiasmou com a idéia e eu o apresentei ao também inesquecível Dr. Gilberto Affonso Penna e começamos a escrever juntos.

Tempos depois, ele se "emancipou" e partiu para a "carreira solo" criando uma seção na revista com o nome de Projetos do Alex. Mas, foi um "divórcio amigável", e eu continuei metendo o bedelho nos seus escritos e dando sugestões. De repente, quis novamente a sincronicidade que nos separássemos, pois ele resolveu assumir o desafio de ir para uma estação nova de satélites da Embratel em Morungaba, lá pelas bandas do interior de Sampa. Nossos contatos passaram a ocorrer no período dos cometas e ficavam quando vez mais distantes.

Paramos de escrever na Antenna por razões que não cabe aqui desenterrar. Tudo numa boa. São aquelas amizades de que fala Fernão Capelo Gaivota. Dois amigos não precisam estar fisicamente perto para continuarem amigos, basta que comunguem as mesmas idéias e os espíritos estarão sempre juntos em algum "lugar" do indecifrável cosmos. De repente meu celular toca, olho o número e reconheço que é do Alex. O cometa está passando por perto de novo. Atendo e diz, ele: escrevi um livro e queria que você desse a sua opinião.

Bem, em 2009, me pediram para instalar uns sensores de presença num projeto de arte chamado SeriaLab. Como nunca digo não para os desafios, topei. Chagando lá, a coisa não era tão simples assim. Os tais sensores tinham que "conversar" com uma plaquinha que eles chamavam de Arduino, mas não sabiam me dizer o que era e como funcionava. Pesquisei e descobri que o Arduino é uma plataforma que funciona como um microcontrolador de modo quase semelhante ao PIC. Estudei o dito cujo e fiquei maravilhado com as possibilidades que ele oferecia e simplicidade de uso. Saí fazendo propaganda do Arduino para todo mundo, escrevi um tutorial, fiz uma palestra na Escola Técnica Estadual Ferreira Vianna no Rio de Janeiro e contaminei alunos e professores.

Há tempos não falava com Alex e de repente ele me liga. Eu estava empolgado com o Arduino, passei a idéia para ele que também ficou contaminado. Pensei em escrever um livro sobre o Arduino, mas como o dia só tem 24 horas não consegui porque eu sou do tipo que "abraça o mundo com as pernas". Pois é, o Alex saiu na frente. O livro que ele escreveu e queria que eu opinasse era sobre o

Arduino. Mas, eu não fiquei de fora. Ele me mostrou o livro e me “intimou”: você VAI prefaciar meu livro. O que dizer numa ora dessas? Que no fundo eu queria que ele me pedisse isso?

Está aí o livro que eu queria ter escrito, talvez muito melhor do que eu iria fazer. Pelo menos eu escrevi o prefácio. Leia livro e fique também contaminado pelo Arduino como nós ficamos.

Paulo Brites

Sumário

Prefácio 5

Apresentação 12

O que o leitor vai aprender 13
O que o leitor vai montar 14
Autor e leitor na internet 14

Capítulo 1 - Microcontroladores 15

Introdução 16
O microcontrolador AVR 18
A CPU e os registradores internos 20
Memórias e *clock* 21
Programação 22
Aplicações 23

Resumo do capítulo 1 24

Capítulo 2 - O Hardware do Arduino 26

Um pouco de história 27
A placa do Arduino 28
Os *Ports* digitais de E/S 31
As entradas analógicas e a saída *PWM* 33

Resumo do capítulo 2 34

Capítulo 3 - O IDE do Arduino 36

Introdução 37
Instalação do *IDE* do Arduino 38
Aplicações 19

A barra de Menus 42
File 42
Edit 44
Sketch 45
Tools 46
Help 47

A Barra de Controles 47
As Barras de Mensagens 48

Capítulo 4 - A Linguagem do Arduino 49

Introdução 50
A estrutura da linguagem do Arduino 51
Constantes 51
Variáveis 51
Funções 52
As principais funções da linguagem do Arduino 53
pinMode(pino,modo) 54
digitalRead(pino) 55
digitalWrite(pino,valor) 55
analogRead(pino) 56
analogWrite(pino,valor) 56
delay(ms) 57
millis( ) 57
random(min,max) 57
Serial.begin(taxa) 58
Serial.println(dado) 58

Os operadores 59
Operadores aritméticos 59

Operadores lógicos 59
Operadores de comparação 59
Operadores compostos 60

Os principais comandos da linguagem 60
O comando *if* 60
O comando *while* 62
O Comando *do ... while* 63
O Comando *for* 63

Matrizes 64

Resumo do capítulo 4 65

Capítulo 5 - Primeiras Experiencias 67

Introdução 63
Experimento #1 – Hello Word! 69
Experimento #2 – Geração de Áudio 73
Experimento #3 – Entrada Analógica 75
Experimento #4 – Saída Analógica 77
Experimento #5 – Controle de Motores 81
5.1 - Controlando um motor CC 81
5.2 - Servo-motores 83
5.3 - Motores de passo 87
5.4 - Vibra-motor 89
Experimento #6 – LEDs e Mostradores 90
6.1 - LEDs 90
6.2 - Mostradores de 7 segmentos 94
6.3 - Um contador de dois dígitos 96
6.4 - LCD 104
Experimento #7 – Solenoides, Reles e Acopladores 107
7.1 - Solenoide 108
7.2 - Relés 109
7.3 - Acoplador ótico 110
Experimento #8 – Sensores 112
8.1 - LDRs 112
8.2 - Fototransistores e Fotodiodos 113
8.3 - Transdutores Piezoelétricos 115
8.4 - Temperatura 117

Resumo do capítulo 5 119

Capítulo 6 - A linguagem Processing 122

Introdução 123
As funções de desenho da linguagem *Processing* 128
size(largura,altura) 129
point(x,y) 129
line(x1,y1,x2,y2) 130
triangle(x1,y1,x2,y2,x3,y3) 130
quad(x1,y1,x2,y2,x3,y3,x4,y4) 131
rect(x,y,largura,altura) 131
ellipse(x,y,largura,altura) 131
arc(x,y,largura,altura,inicio,fim) 132

Outras funções importantes de desenho plano 133
smooth( ) 133
strokeWeight( ) 134
strokeJoin( ) 134
strokeCap( ) 135

Plotando gráficos simples 136

Resumo do capítulo 6 139

Capítulo 7 - Arduino e *Processing* 142

Introdução 143
Experimento #9 – Comunicação serial 144
Experimento #10 – Módulo de Desenvolvimento 150
Experimento #11 – *Shield* matriz de contatos 154
Experimento #12 – Monitor de batimentos cardiacos 157
Parte 1: "Vendo" os pulsos cardíacos 157
Parte 2: Uma interface gráfica em *Processing* 163

Apendices:
1 - Monte seu próprio Arduino 169

Introdução 170
O circuito proposto do Arduino 171
Descrição do Circuito 172
A placa de Circuito Impresso 176
Alimentando o Arduino 177

2 - Programador de *Bootloader* 178

Introdução 178
Gravando o *bootloader* 182
Conectando o Arduino a um computador 185
Descrição do Circuito 186

Índice remissivo 179

Apresentação

Este é um livro sobre uma ferramenta eletronica baseada em um microcontrolador de 8 bits que a partir de sensores conectados às suas entradas pode ser programada para controlar outros circuitos eletronicos conectados às suas saídas. Essa ferramenta é o Arduino e o microcontrolador nela embarcado é o **ATmega328** da empresa *Atmel Corp*. O Arduino é um pequeno módulo eletronico, uma placa de circuito impresso, onde são montados o **ATmega328** e alguns outros componentes eletronicos e que pode se comunicar diretamente com qualquer computador que possua uma interface serial. Compondo o Arduino existe tambem um programa que roda em qualquer sistema operacional, seja ele *Windows*, *Linux* ou *Mac OS-X*. E o mais interessante é que tanto o circuito quanto o programa são *open source*, ou sistemas abertos, o leitor pode comprar ou montar seu próprio módulo e obter gratuitamente o aplicativo pela internet.

O que o leitor vai aprender

Escrevemos este livro para estudantes, projetistas e profissionais de Eletronica que desejam conhecer o Arduino. Não é necessário ter um conhecimento profundo de microcontroladores e sua programação para que você caro leitor comece a criar seus próprios projetos baseados nessa plataforma de *hardware*. A placa já montada e testada do Arduino pode ser adquirida com bastante facilidade nas lojas de comercio eletronico, particularmete naquelas que vendem pela internet, no Brasil ou no exterior. O leitor verá que programar tarefas para o Arduino é muito fácil com o auxilio de um aplicativo conhecido como IDE (*Integrated Development Enviroment*) ou Ambiente de Desenvolvimento Integrado, que pode ser encontrado gratuitamente na *web*. A linguagem padrão para programação do Arduino é baseada na já consagrada linguagem C/C++. Com uns poucos comandos dessa linguagem o leitor verá que o Arduino pode controlar pequenas cargas e circuitos simples conectados às suas saídas. Tambem o leitor vai fazer com que o seu Arduino se comunique com o seu computador pessoal via porta serial, e mostre graficamente na sua tela variáveis físicas captadas por sensores utilizando uma outra linguagem muito fácil de aprender, a linguagem *Processing*.

O que o leitor vai montar

Já nos primeiros capítulos o leitor poderá fazer experiências com leds, potenciometros, sensores resistivos, solenoides e mesmo pequenos motores conectados diretamente às portas de entrada e saída do seu Arduino. Aqueles leitores mais experientes em Eletronica poderão montar alguns circuitos propostos no livro, como mostradores de 7-segmentos e LCD, controladores de motores servo e de passo ou um monitor de batimentos cardíacos. Para os leitores mais ousados propomos montar seu próprio Arduino, desde a confecção da placa de circuito impresso até a programação do *bootloader*, um pequeno programa residente em memória que faz do ATmega328 o verdadeiro Arduino.

Autor e leitor na internet

No endereço web do autor *www.ordemnatural.com.br* o leitor vai encontrar todo o suporte para as montagens que aparecem no livro, como novas fotos, novas versões de programas dessas montagens e eventuais correções do texto do livro. O leitor poderá tambem baixar os códigos fonte tanto em C quanto em *Processing* de todas as montagens e conhecer novos circuitos que não foram incluidos no livro. Há ainda uma área para que o leitor entre em contato com o autor para esclarecer dúvidas, enviar sugestões e até mesmo enviar seus projetos com o Arduino para ali ser publicados. Aguardo sua visita. Seja bem-vindo ao mundo do Arduino e boa leitura.

Capítulo 1

Microcontroladores

Introdução

Até alguns anos atrás quando queríamos montar um circuito eletronico de média complexidade tínhamos que juntar muitos componentes, como resistores, capacitores, transistores, circuitos integrados, e até mesmo indutores e chaves mecanicas de vários polos e posições. O circuito final era dedicado a uma só aplicação e não era muito fácil fazer correções ou melhoramentos uma vez montado, pois era preciso cortar filetes na placa de circuito impresso, refazer soldas e mesmo montar outros componentes sobre os já existentes. Os microcontroladores vieram para facilitar a vida dos projetistas e experimentadores de circuitos eletronicos mais complexos já que muitos dos fios e das trilhas metálicas e vários componentes discretos nas placas podem ser substituidos por linhas de código de programas que rodam nesses novos componentes. Linhas de programas são muito mais fáceis de mudar que componentes soldados numa placa de circuito impresso. Somente com a reescrita de umas poucas linhas do código de um programa podemos mudar radicalmente o funcionamento do nosso circuito e testar novas possibilidades no mesmo período de tempo que levaríamos para trocar alguns resistores numa placa. Para isso o técnico e o engenheiro eletronico precisam de um conhecimento mínimo de uma linguagem de programação dedicada a esses novos dispositivos.

O termo microcontrolador é usado para descrever um sistema minimo que inclui uma CPU, memória e circuitos de entrada e saída, tudo montado num único circuito integrado, que pode ter de 8 a até mais de 100 pinos. Alguns microcontroladores podem vir com contadores decimais internos,

conversores analógico-digitais, comparadores de tensão e circuitos de comunicação serial, tudo embutido no mesmo encapsulamento. Tambem chamados de **microcomputador em um chip**, os microcontroladores podem ser de 8, 16 ou mesmo 32 bits, conforme o barramento que conecta cada circuito interno. Outro termo hoje muito comum é se referir aos microcontroladores como controladores embarcados, já que eles são montados dentro do aparelho ou instrumento que controlam. A diferença dos microcontroladores para os microprocessadores como os que integram nossos computadores pessoais é que estes ultimos são formados por somente uma CPU de 8, 16, 32 ou 64 bits e necessitam de memória externa e circuitos para controle de entrada e saída de dados para formar um sistema inteligente de processamento e controle.

Os microcontroladores funcionam seguindo uma lista de instruções armazenadas em forma de códigos binários em uma memória de programa interna. Nos computadores pessoais, ou PCs, esse programa é armazenado em um disco magnetico rígido, conhecidos por HDs (*Hard Disks*). Essas instruções são apanhadas uma a uma da memória, decodificadas por circuitos lógicos internos à CPU e então executadas. Uma sequencia de códigos pode por exemplo instruir o microcontrolador a fazer com que um de seus pinos de saída acione o circuito de um aquecedor se a temperatura lida por um sensor em um outro pino agora de entrada decrescer a certo valor. As instruções para um microcontrolador tradicionalmente são escritas na linguagem *Assembly*, hoje porem existe uma tendência a se usar cada vez mais a linguagem C e outras mais fáceis de aprender e programar, como BASIC e PASCAL.

Existem muitos fabricantes de microcontroladores e os mais conhecidos são os das empresas Microchip, Atmel, Intel e Freescale, entre outras. Cada fabricante tem sua própria arquitetura interna de registradores e capacidade de memória e sua própria lista de instruções; portanto um programa escrito para um dado microcontrolador não pode ser executado por um de outro fabricante. A arquitetura interna mais simples de um microcontrolador consiste em uma CPU, memória de programa, memória de dados e circuitos de controle de entrada e saída de dados, todos interligados por um barramento (*bus*) de 8 ou 16 bits, e tudo encapsulado em um só circuito integrado. A CPU (*Central Processing Unit*) ou Unidade Central de Processamento é a parte do microcontrolador responsável pelas operações matemáticas e pelas operações lógicas, como AND, OR e NOT. Nos microcontroladores a memória interna é separada em dois tipos conforme sua função: memória de programa e memória de dados. A primeira armazena as instruções escritas pelo programador e que dizem o que o sistema terá que fazer. A segunda armazena as informações que são trocadas de forma temporária entre a CPU e os diversos circuitos internos, como contadores e registradores.
Os microcontroladores normalmente são alimentados com uma tensão padrão de 5 volts. Muitos podem operar com tensões de 2 volts e outros com até 6 volts. Todos os microcontroladores necessitam de um circuito de relógio (*clock*) para sincronização de seus circuitos internos que normalmente são mantidos por um cristal ou ressonador cerâmico externo. A maioria dos pinos de um microcontrolador são usados para entrada e saída de dados, são os chamados *Ports*.

O microcontrolador AVR

A sigla AVR vem de *Advanced Virtual RISC* ou *Alf and Vegard RISC*, os nomes dos projetistas desse microcontrolador. *RISC* vem de *Reduced Instruction Set Computer*, ou computador com um conjunto reduzido de instruções, que é uma referência ao pequeno número de instruções do microcontrolador se comparado aos primeiros microprocessadores cuja lista podia passar de 200 instruções.

O primeiro Arduino foi baseado no circuito básico com um microcontrolador AVR ATmega8 da empresa americana *Atmel Corporation* no ano de 2005 numa escola de artes interativas e *design*, na Italia. São tres blocos básicos: a CPU, o bloco de memória e o de registradores e circuitos de entrada e saída, veja o diagrama em blocos ao lado. A CPU é formada por uma ALU (*Arithmetic Logic Unit*) ou Unidade de Lógica e Aritmética e um conjunto de registradores de uso geral. O bloco de memórias agrega as memórias de programa e memória de dados. No bloco de E/S estão os *ports* que são circuitos de interfaces de entrada e saída, mais alguns registradores e implementações de circuitos *pwm* e conversores A/D.
Depois de 2005 a Atmel lançou novas versões desse microcontrolador, com mais memória e outros recursos, porem mantendo a compatibilidade com aquele. Novos modelos de Arduinos surgiram com esses novos dispositivos. Os microcontroladores ATmega fazem parte do grupo mega da família AVR de microcontroladores de 8 bits da Atmel. Outro grupo dessa família são os microcontroladores **Tiny**, como o conhecido ATtiny13A de 8 pinos.

Na figura 2, acima, temos o diagrama simplificado em blocos do ATmega328. As tres principais blocos de qualquer microcontrolador ali estão reperesentadas: a CPU, a memória e o circuito de E/S. Na parte central à direita vemos o barramento de dados de 8 bits (1 *byte*) que interliga todos os módulos do microcontrolador que necessitam trocar informações. Normalmente os microcontroladores são projetados de acordo com uma arquitetura chamada *Harvard* onde os dados e os endereços são separados em dois barramentos independentes.

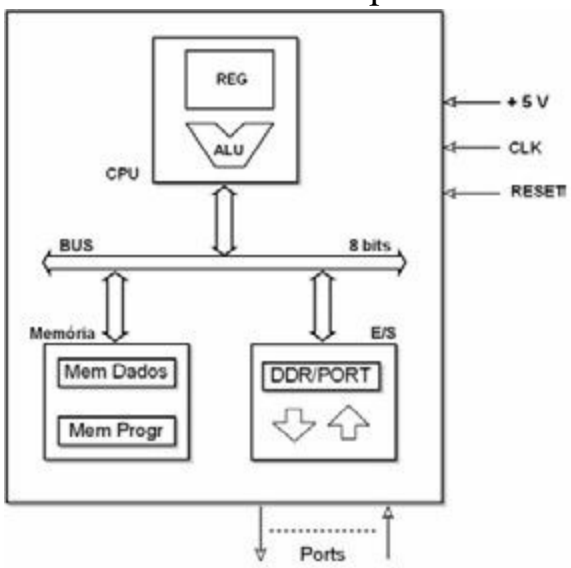


Figura 1: um AVR em blocos

A CPU e os registradores internos

A CPU é formada pela ALU ou Unidade de Lógica e Aritmética, 32 registradores de uso geral, um Registrador de Instruções, um Contador de Programa (que aparece aqui fora da área tracejada), um

Decodificador de Instruções e mais dois ou tres outros registradores especializados que no diagrama da figura 2 foram omitidos para maior clareza.

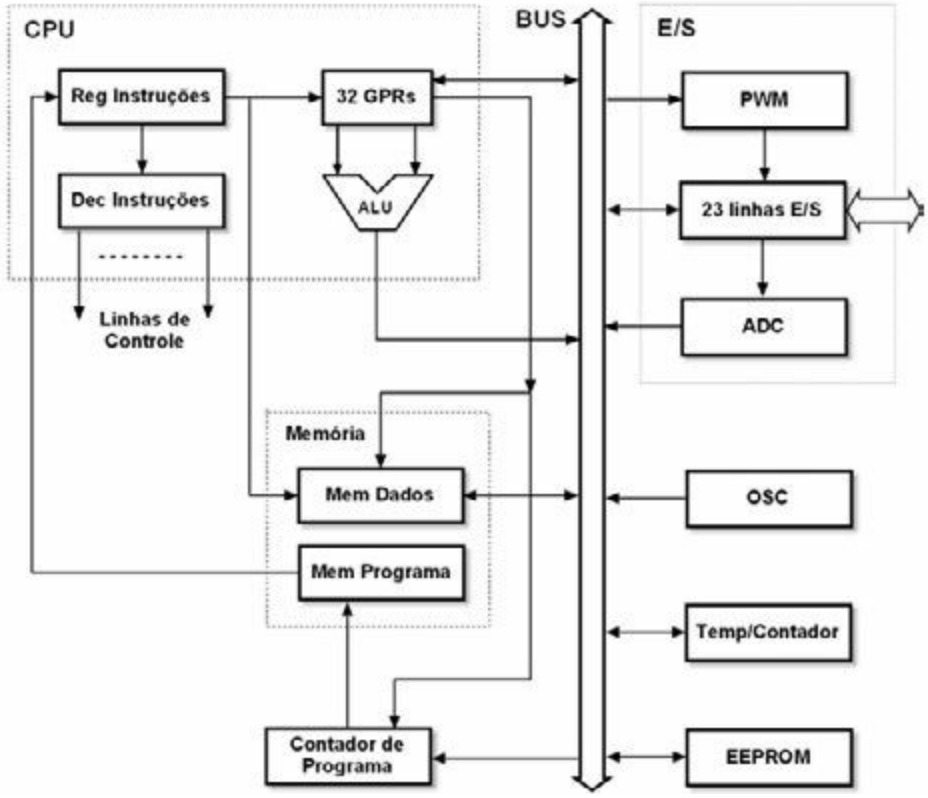


Figura 2: o microcontrolador ATmega 328 em blocos

Um registrador é um bloco digital formado por um grupo de flip-flops do tipo D ou JK. No caso de um microcontrolador de 8 bits seus registradores podem ser blocos de 8 ou 16 flip-flops encadeados de modo que a passagem de bits de um flip-flop para outro possa ser controlada por linhas externas ao bloco. Cada flip-flop de um registrador pode ser carregado individualmente com um bit 1 ou 0, quando dizemos que o flip-flop foi setado ou resetado, respectivamente. Todo o registrador pode ser carregado com uma sequência qualquer de bits, ou ele pode ser todo resetado. Tambem podemos deslocar todos os bits em um registrador para a esquerda ou para a direita, uma ou mais posições. Todas essas operações dentro dos registradores podem realizadas por instruções dentro de programas ou por controle direto da CPU como resultado de outras operações internas.

Dentro da CPU do ATmega328 existe um grupo de 32 registradores de 8 bits conectados diretamente a ALU, a Unidade de Lógica e Aritmética. São os registradores GPR (*General Purpose Registers*) ou registradores de uso geral que são usados pela CPU para armazenar temporariamente os operandos, resultados das operações de soma e subtração ou operações lógicas realizadas pela ALU. Aqui tambem são guardadas as variáveis locais e ponteiros de programas durante sua execução. O registrador de instruções (*Instruction Register*) armazena o *byte* de controle que representa uma instrução que recebe da memória de programa e o passa para o decodificador de instruções (*Instruction Decoder*), que é uma matriz de portas lógicas programáveis (*Programmable Array Logic*) cuja saída gera controles para as várias partes do microcontrolador.
O contador de programas (*Program Counter*) é um registrador que sempre aponta para o endereço na memória de programa da próxima instrução a ser executada. Ele é automaticamente incrementado depois que uma instrução é executada.

Memórias e clock

A memória do ATmega328 é composta por dois blocos principais: a memória de dados, que é do tipo volátil, RAM estática, para variáveis que não são guardadas em registradores e informações temporárias coletadas externamente; e a memória de programa que é do tipo não-volátil mas reprogramável, normalmente memória *Flash*, que guarda o programa que deve ser executado pelo microcontrolador. O ATmega328 possui 2K de memória para dados, 32K de memória para programa e tambem 1K de memória do tipo EEPROM, não-volátil e reprogramável, que no diagrama simplificado da figura 2 aparece fora do bloco de memórias, para armazenamento de informações que desejamos guardar mesmo depois de retirarmos a alimentação do circuito, como a senha em um *chip* no cartão de crédito.

Todos os microcontroladores requerem um circuito de relógio (*clock*) para sincronizar suas operações internas e externas. Esse sincronismo é realizado por um oscilador interno mas com uma referência externa, como um cristal e dois capacitores, um ressonador cerâmico ou mesmo um circuito RC. O ATmega328 pode operar com um cristal ou ressonador de até 20Mhz. Existem alguns blocos internos nos microcontroladores que trabalham sincronizados com o circuito de relógio, são os contadores e os temporizadores. Contadores são circuitos digitais montados com flip-flops, como os registradores, mas com a capacidade de incrementar ou decrementar um bit ou um pulso de cada vez, como o Contador de Programas visto acima. Contadores tambem podem ser resetados ou carregados com uma sequência de bits e a partir daí serem incrementados ou decrementados. Os contadores são tambem chamados de temporizadores (*Counters/Timers*) já que podem ser programados para indicar períodos decorridos a partir da contagem de um certo número de pulsos com duração fixa. O ATmega328 possui dois contadores de 8 bits e um de 16 bits de uso geral e um contador de tempo real.

O microcontrolador ATmega328 que estamos estudando é de encapsulamento em linha dupla (DIL ou *Dual-In-Line*) de 28 pinos, mais fáceis de se encontrar no mercado. Desses 28 pinos, 23 são pinos que podem ser programados para enviar (*output*) um nível lógico TTL (5 volts ou 0 volt) do microcontrolador para um circuito externo a ele conectado, ou podem ser programados para receber (*input*) um nível lógico TTL de um circuito externo para o microcontrolador. Esses 23 pinos são agrupados em dois conjuntos de 8 pinos e um de 7 pinos, que recebem o nome de *Ports*.

Nem todos os pinos nos microcontroladores são exclusivamente digitais, alguns deles podem receber sinais analógicos. No ATmega328 seis dos 23 pinos de entrada e saída podem ser programados para se tornarem entradas de sinais analógicos. Qualquer desses 6 pinos, após passar por um seletor analógico interno, é diretamente conectado a um conversor A/D (*Analog-Digital Converter*) de 10 bits de resolução. A tensão analógica na entrada desse conversor pode variar de 0 a +5 volts, no máximo. Tambem outros seis no ATmega328 podem ser programados para a função PWM (*Pulse Width Modulated*) ou modulação por largura de pulso.
Um sinal PWM tem a forma de uma onda quadrada com frequencia fixa porem com a largura de seus pulsos diretamente proporcional à amplitude de um outro sinal chamado de modulante. Um sinal

modulante pode variar a largura dos pulsos de um sinal PWM entre 0% e 100% da sua forma original. Outras funções tambem disponíveis nos pinos desses microcontroladores são um comparador analógico de tensão e um canal de comunicação serial atraves de uma USART (*Universal Synchronous-Asynchronous Receiver-Transmitter*) implementada internamente.

Programação

Já sabemos que um microcontrolador é um pequeno computador em um *chip*; porem não bastam a CPU, a memória e os circuitos de entrada e saída para ele funcionar, é necessário tambem um programa que instrua esse *hardware* o que tem que ser feito. O *hardware* é a parte física, visível, o corpo do computador; o programa, ou *software*, é a parte invisível ou alma deste. O programa fica residente em algum tipo de memória do computador. Nos computadores pessoais *PC* ou *Mac* o programa fica no disco rígido; nos microcontroladores ele fica na memória de programa interna do tipo *Flash*. A memória *Flash* é uma versão da memória EEPROM (*Electrically ErasableProgrammable ROM*) e onde são gravados o *bootloader*, que é um programa opcional de inicialização do microcontrolador, e posteriormente os programas do usuário.

O Arduino tem um *bootloader*, um pequeno programa residente numa parte da memória *Flash*, chamado *bootblock*. Logo após o *reset* do microcontrolador o *bootloader*, ou simplesmente *boot*, como é tambem chamado, se comunica via interface serial com o computador para receber o programa do usuário, e o grava na parte restante da memória *Flash* do microcontrolador e logo em seguida o executa. O *boot* é gravado com um programador externo e o programa do usuário é gravado pelo *boot*.
Programas são listas com instruções escritas numa linguagem que um computador entenda e as execute. A linguagem que o computador entende é a chamada linguagem de máquina, são blocos com longas sequências de 1´s e 0´s que a CPU lê, interpreta e distribui ordens para as suas várias partes internas, e destas controles para circuitos e dispositivos externos. Mas não escrevemos programas diretamente na linguagem das máquinas, os escrevemos em linguagens que nós humanos entendemos, para depois entregarmos a um outro programa, um compilador, que os converte para a linguagem dos computadores. Os programas para o Arduino são escritos na linguagem C. Para escrever nossos primeiros programas em C basta conhecermos umas poucas estruturas de controle e suas regras. Existe um aplicativo onde podemos editar, verificar ocorrência de erros e compilar nossos códigos em C para o Arduino, é o Ambiente de Desenvolvimento Integrado ou IDE (*Integrated Development Enviropment*) do Arduino, que veremos em detalhes no capítulo 3.

Aplicações

Um microcontrolador é uma ferramenta poderosa que permite ao projetista eletronico criar sistemas complexos sob controle de um programa. Encontramos microcontroladores em todos os aparelhos eletronicos que nos cercam, como telefones celulares e GPS, televisores e seus controles remotos, fornos de micro-ondas e refrigeradores, impressoras e *mouses*, e até naqueles cartões de crédito e

débito que possuem o já tão conhecido *chip* que carregamos em nossos bolsos. Um automóvel moderno pode ter até dez microcontroladores embarcados: dois ou tres para monitoração e controle do motor e transmissão, outro para os freios ABS, outro para os instrumentos no painel, para os alarmes e trancas das portas. O circuito mais simples que podemos montar com um microcontrolador ATmega328 precisa somente de um cristal de 16 Mhz, dois capacitores ceramicos de 22pF, um resistor de 10K e uma chave de *reset*, conforme vemos na figura 3, acima. para alimentá-lo qualquer fonte regulada que forneça 5 volts CC serve. Se nesse circuito simples conectarmos um sensor de temperatura no pino 23 (a linha 0 do *Port* C) e um transistor acionando um relé no pino 19 (linha 5 do *Port* B) , por exemplo; e na memória *Flash* do microcontrolador carregarmos um programa de umas poucas linhas poderemos controlar a temperatura interna de um forno elétrico mantendo-a entre dois limites previamente escolhidos. E se ainda conectarmos diretamente a outros pinos do microcontrolador um teclado simples e um mostrador LCD ou de 7-segmentos, bastaria incluirmos mais algumas linhas de código em nosso programa para entrarmos com os limites de temperatura desejados pelo teclado e termos a leitura instantanea da temperatura interna do forno no mostrador digital.

Resumo do capítulo 1

Um microcontrolador é um único circuito integrado que pode ter de 8 a até mais de 100 pinos e que inclui uma CPU, memória e circuitos de entrada e saída. Alguns modelos podem vir com contadores/temporizadores decimais, conversores A/D, comparadores de tensão e circuitos de comunicação serial, tudo embutido no mesmo encapsulamento. Os microcontroladores como qualquer computador funcionam seguindo uma lista de instruções armazenadas em forma de códigos binários em uma memória de programa. Essas instruções, são escritas em Assembly, C, BASIC ou Pascal, e apanhadas uma a uma da memória e decodificadas por circuitos lógicos internos à CPU e então executadas. A CPU é a parte do microcontrolador responsável pelas operações matemáticas e pelas operações lógicas; a memória é separada em dois tipos conforme sua função: memória de programa, do tipo *Flash*, e memória de dados, do tipo RAM; os *Ports* formam os circuitos de entrada e saída do microcontrolador.

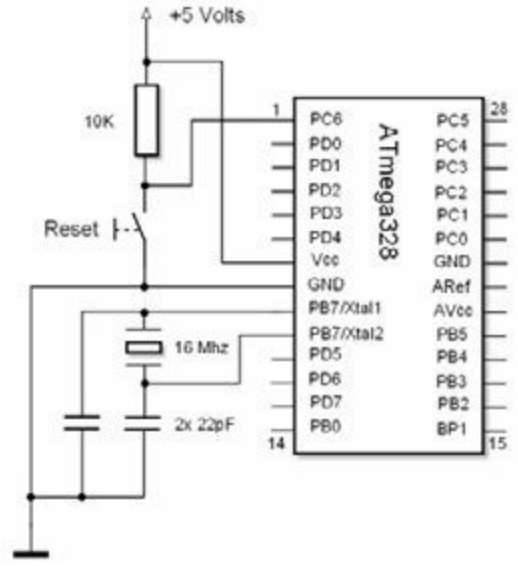


Figura 3: um sistema minimo com AVR

O primeiro Arduino foi projetado com o microcontrolador ATmega8 da Atmel em 2005. Os mais novos Arduinos vêm com o ATmega1280 que tem 16 vezes mais memória de programa e tres vezes mais pinos de E/S. Registradores são blocos digitais formados por grupos de 8 ou 16 flip-flops do tipo D ou JK e são usados pela CPU para guardar resultados de operações internas e troca de dados com circuitos externos. O ATmega328 possui um grupo de 32 registradores de uso geral de 8 bits conectados diretamente a ALU, a Unidade de Lógica e Aritmética. Outros registradores presentes em qualquer microcontrolador são: o Contador de Programas, o Registrador de Instruções e o Decodificador de Instruções. Cada grupo de *Ports* é tambem um registrador de 8 bits conectados fisicamente aos pinos do microcontrolador. No ATmega328 seis dos 23 pinos de entrada e saída que formam os *Ports* são diretamente conectados a um conversor A/D de 10 bits de resolução, enquanto outros seis pinos podem ser programados para a função de saída PWM, ou modulação por largura de pulso.

Hardware é a parte física, visível, que compõe o corpo do computador; o programa, ou *software*, é a parte invisível ou alma deste. Um microcontrolador sozinho é sómente uma combinação de circuitos eletronicos digitais montados em um mesmo encapsulamento que não tem nenhuma aplicação se não houver um *software* que instrua esse *hardware* no que tem que ser feito. E as instruções para o *hardware* são montadas em forma de listas escritas em uma linguagem inteligível para o programador, e depois convertidas para uma linguagem inteligível pela CPU do computador.

Capítulo 2

O *hardware* do Arduino

Um pouco de história

O primeiro Arduino foi criado em janeiro de 2005 no Instituto de Interatividade e Design, uma escola de Artes visuais na cidade de Ivrea, Italia, a partir de uma ideia dos professores de Computação Física *David Cuartielles* e *Massimo Banzi*. O objetivo era criar uma ferramenta de *hardware* única que fosse facilmente programável por não especialistas em computadores e que tambem não fosse muito cara, para o desenvolvimento de estruturas interativas no curso de Arte e Design. Computação Física é o estudo da interação entre o nosso mundo físico e o mundo virtual dos computadores, e para isso envolve tambem o estudo de sensores eletricos e atuadores eletro-mecanicos.

À dupla de professores juntaram-se outros especialistas de programação que criaram um Ambiente

de Desenvolvimento Integrado, uma ferramenta de *software* que traduz uma linguagem de alto nível como C para a linguagem de máquina que o Arduino entende. Tudo isso permitiu que artistas e designers pudessem dar às suas ideias uma forma física com protótipos que interagiam com os usuários e o ambiente onde estão.

Todo o projeto Arduino foi concebido segundo o princípio do *open source*, que em essência quer dizer que todos os seus programas são de domínio público, ou seja são livres para cópia, modificação e melhoramentos por qualquer pessoa. Da mesma forma os circuitos eletronicos tanto do Arduino quanto aqueles que foram criados para ser conectados a ele podem ser copiados e modificados. Existe na *web* centenas de comunidades de artistas, projetistas e programadores que disponibilizam suas criações para o mundo inteiro.

A placa do Arduino

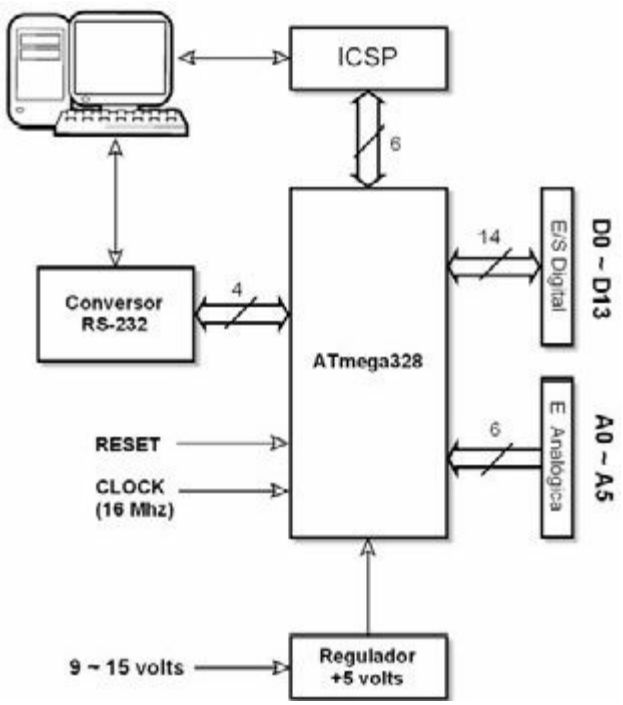
O Arduino é composto por duas partes principais: um *hardware*, um conjunto básico de componentes eletronicos montados numa placa de circuito impresso, que é a plataforma para o desenvolvimento de protótipos; e um *software*, um aplicativo, o *bootloader*, residente na memória de programas do microcontrolador embarcado no Arduino. Existe tambem uma interface gráfica, um programa que roda num computador padrão PC em ambiente Windows ou Linux, ou ainda numa máquina Apple com o sistema operacional Mac OS X. É nessa interface gráfica ou Ambiente de Desenvolvimento Integrado (IDE – *Integrated Development Environment*) que o leitor vai criar seus programas e depois carregar para o *hardware* do Arduino. São esses programas, ou na linguagem do Arduino: *sketches*, que vão dizer ao *hardware* o que deve ser feito.

ARDUINO	Diecimila	Duemilanove168	
Duemilanove328	Mega		
Processador	ATmega8	ATmega168	ATmega328	ATmega1280
Memória flash	8 K	16 K	32 K	128 K
Memória RAM	1 K	1 K	2 K	8 K
Memória EEPROM	512 bytes	512 bytes	1 K	4 K
Pinos digitais	14	14	14	54
Pinos analógicos	6	6	6	16

Saídas PWM	3	6	6	14

O *hardware* do Arduino é baseado nos microcontroladores AVR da Atmel, principalmente nos modelos ATmega8, ATmega168, ATmega328 e no ATmega1280. Conforme o microcontrolador utilizado o Arduino recebe um codinome em italiano. Veja na tabela comparativa abaixo as diferenças principais entre os Arduinos em relação ao microcontrolador nele embarcado.

O primeiro Arduino, batizado depois de Diecimila que em italiano quer dizer dez mil, marca de vendas desse modelo, usava o ATmega8 com uma fonte de alimentação simples com o regulador **LM7805**, um circuito de conversão para comunicação serial RS-232, e alguns conectores para os *Ports* do microcontrolador. Tambem foi acrescido ao projeto original um conector para a programação do ATmega8 no circuito, o *ICSP* ou *In-Circuit Serial Programming*. Veja o diagrama em blocos do Arduino na figura 4.



O circuito do conversor RS-232 originalmente foi projetado com transistores bipolares **BC557**. Depois surgiram versões como o Arduino Freeduino MaxSerial que utilizavam o circuito integrado conversor **MAX232N**. A função do conversor RS-232 é apenas compatibilizar os níveis TTL do ATmega8 com os níveis de tensão padronizados do protocolo RS-232 na comunicação com um computador PC ou Apple.

Na placa do Arduino o conector macho ICSP (*In-Circuit Serial programming*) de 6 pinos é ligado diretamente a alguns pinos do microcontrolador e é conectado atraves de um cabo com a porta paralela ou serial de um computador, para a programação do *bootloader* no microcontrolador. A fonte de alimentação de 5 volts é composta por um único regulador integrado LM7805 e filtros capacitivos. A entrada da fonte pode receber tensões de 9 a 15 volts não regulados de um carregador de baterias ou de uma fonte de bancada. Tambem é possível alimentar o módulo com uma bateria comercial de 9 volts.

Por fim o Arduino tem dois outros conjuntos de conectores: um com 14 pinos para entradas ou saídas de sinais digitais e um de 6 pinos para entrada de sinais analógicos. Veremos mais adiante que esses pinos de entradas analógicas tambem podem ser programados para entradas ou saidas digitais, totalizando 20 pinos para esse fim. No módulo temos ainda um botão para resetar o microcontrolador e um LED que vai conectado ao pino digital 13 para testes simples e monitoração.

O Arduino possui um total de sete conectores. Veja a figura 5 acima. São dois conectores com os 14 pinos digitais que ficam na parte superior da placa e são identificados com serigrafia como "DIGITAL" e numerados de 0 a 13, da direita para a esquerda. Os pinos 0 e 1 são os dois pinos RX e TX de comunicação serial entre o Arduino e o computador. Na parte inferior da placa à direita fica o conector de 6 pinos identificado como "ANALOG IN" para sinais analógicos (tensões de entradade 0 a +5 volts). Aqui os pinos são contados da esquerda para a direita e com serigrafia são identificados com os números de 0 a 5. Esses pinos podem ser programados tambem como entradas ou saídas digitais da mesma forma que os 14 pinos digitais, podendo assim obtermos 20 pinos digitais. O pino de entrada de tensão de referência ("AREF") para esse conversor fica no conector digital da esquerda, é o último pino. À direita deste pino fica um pino de terra ("GND").

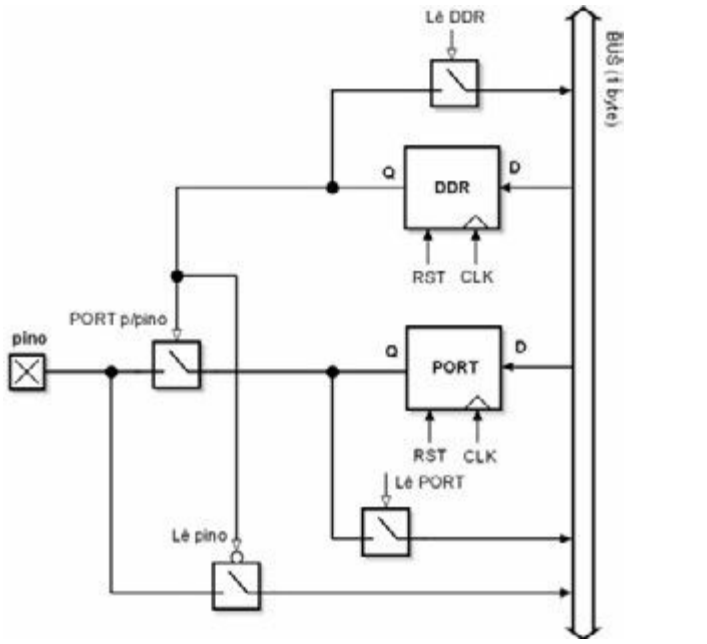
À esquerda de "ANALOG IN" existe um conector auxiliar identificado como "POWER" com tensões para alimentar um circuito externo eventualmente conectado ao Arduino. Um último conector ao lado botão RESET fica o ICSP. Uma vez programado o *bootloader* toda a programação do Arduino passa a ser feita pela porta serial USB que fica à esquerda da placa. Ainda à esquerda da placa, logo abaixo do conector serial, fica a entrada para a alimentação do Arduino, um conector tipo *jack* que pode receber tensões de 9 a 15 volts CC com o positivo no pino central.

Os Ports digitais de E/S

O termo *Port* se refere a um grupo de 8 pinos de um microcontrolador que são fisicamente

conectados a registradores de 8 bits no circuito digital de E/S. O ATmega328 tem tres *Ports* que são conectados a tres registradores chamados de PORT B, com 8 flip-flops, PORT C, com 7 flip-flops, e PORT D, tambem com 8 flip-flops. Não existe o PORT A. Podemos enviar ou receber qualquer combinação de níveis lógicos por esses *Ports* físicos para esses registradores na forma de 1 byte. Tambem podemos programar individualmente cada pino de qualquer desses *Ports* de forma que uns possam ser saídas enquanto outros possam ser entradas de níveis lógicos. É importante conhecermos mais detalhadamente o circuito dos *Ports* porque são eles são a conexão física entre a CPU e o mundo exterior. Na figura 6 podemos ver o diagrama em blocos do circuito interno de um *Port* típico.

No ATmega328 cada *Port* tem na verdade dois registradores de 8 bits*:* um registrador de direção de dados chamado de DDR (*Data Direction Register*) e um registrador de dados propriamente dito ou simplesmente registrador PORT. O equivalente ao DDR nos microcontroladores PIC é o registrador TRIS. A saída Q de cada flip-flop do registrador DDR é que determina o sentido do fluxo de dados em cada pino do microcontrolador, se da CPU para o pino ou se do pino para a CPU. O registrador PORT somente lê e armazena o nível lógico que deve ser transferido do barramento para o pino. Quem comanda todas as linhas de controle tanto no DDR quanto no PORT é o Decodificador de Instruções que vimos no capítulo 1. O nível lógico nas entradas D desses registradores são transferidos para suas saídas Q com o pulso de habilitação em suas linhas de *clock* (CLK).



O registrador DDR funciona da seguinte forma: se o nível lógico transferido de uma de suas entradas D para a correspondente saída Q for baixo (estado lógico 0) a chave digital nessa saída que vai ao pino a ela associada é aberta; com isso esse pino permanece no modo entrada, ficando em alta impedancia; a chave digital mais abaixo é fechada permitindo a transferencia de níveis lógicos externos para o barramento e daí para a CPU. Se o nível lógico na saída Q desse registrador DDR for alto (nível lógico 1) a primeira chave é fechada, o que faz com que a saída Q do registrador PORT seja transferida para o pino a ela associado, o pino permanece no modo saída. A segunda chave digital é aberta.

Nos microcontroladores *PIC* ocorre o contrário: o nível lógico 1 na saída do registrador de direção *TRIS* faz o pino de um *Port* ser entrada, o nível 0 faz esse pino ser saída.

Chaves digitais são portas não inversoras que possuem uma linha de controle para ligar e desligar sua entrada de sua saída, são conhecidas como *tri-state buffers*. A qualquer momento as saídas Q dos registradores DDR e PORT podem ser lidas pela CPU atraves do controle de *clock* de outras duas chaves digitais. Esses registradores são resetados quando o microcontrolador é inicializado ou resetado e todos os pinos são colocados em estado de alta impedancia.

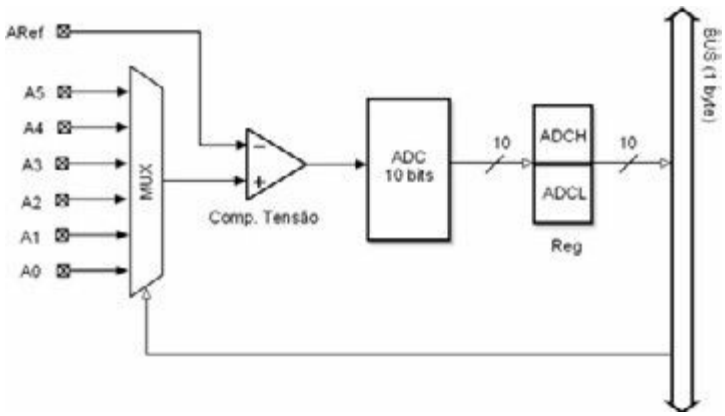


Figura 7: o conversor A/D do microcontrolador ATmega328

As entradas analógicas e a saída PWM

Seis dos pinos digitais do *Port* C do ATmega328 podem ser configurados para receber qualquer nível de tensão entre 0 e 5 volts que vai para a entrada de um conversor analógico-digital (A/D) de 10 bits de resolução. Somente um desses pinos pode ser selecionado de cada vez por um multiplexador (MUX), um tipo de chave seletora, para ser entrada do conversor. Os 10 bits que representam aquele nível de tensão contínua na entrada do conversor são armazenados em dois registradores internos, um de 8 bits e outro de 2 bits. Veja o diagrama em blocos na figura 7. Observe que entre o conversor A/D e o MUX existe um comparador de tensão que recebe uma referência externa de um dos pinos do ATmega328, o pino *Aref*, que determina a faixa de conversão para o circuito. No Arduino essas entradas analógicas são os pinos A0 a A5 (ou pinos 14 a 19).

Existe tambem no ATmega328 um circuito comparador de tensão independente que pode usado para monitorar dois níveis de tensão externos e disparar o contador digital interno *Timer/Counter1* quando uma dessas tensões for maior que a outra. As entradas desse comparador são os pinos 6 e 7 do *Port* D. No Arduino são os pinos digitais 6 e 7.

Sistemas digitais tambem podem gerar sinais analógicos com circuitos conversores digitais-analógicos (D/A). Embora o microcontrolador AVR usado no Arduino não tenha um conversor D/A

interno, é possível gerar voltagens analógicas de 0 a 5 volts em alguns de seus pinos de saídas digitais empregando uma técnica conhecida por PWM (*Pulse Width Modulation*) ou Modulação por Largura de Pulso. Com essa técnica pode-se construir um sistema eletronico que gera em sua saída um trem de pulsos TTL de cadencia fixa mas com a largura dos seus pulsos positivos variando de acordo com uma combinação de bits programada em um registrador de uso geral (GPR) do microcontrolador. Esse trem de pulsos PWM só precisa passar por um filtro RC passa-baixas semelhante àquele usado em fontes de alimentação para se obter uma tensão CC de 0 a 5 volts. No Arduino esses pinos PWM são os pinos digitais 3, 5, 6 e 9 a 11.

Resumo do capítulo 2

O primeiro Arduino foi criado em janeiro de 2005 na Italia por *David Cuartielles* e *Massimo Banzi*. O objetivo era criar uma plataforma de hardware que fosse facilmente programável por não especialistas em computadores. Todo o projeto Arduino foi concebido segundo o princípio do *open source*, onde todos os programas e circuitos são de domínio público.

O Arduino é composto por duas partes principais: um *hardware*, um microcontrolador e alguns componentes eletronicos montados numa placa de circuito impresso; e um *software* residente na memória de programas do microcontrolador e tambem uma interface gráfica que roda num computador padrão PC ou numa máquina Apple.

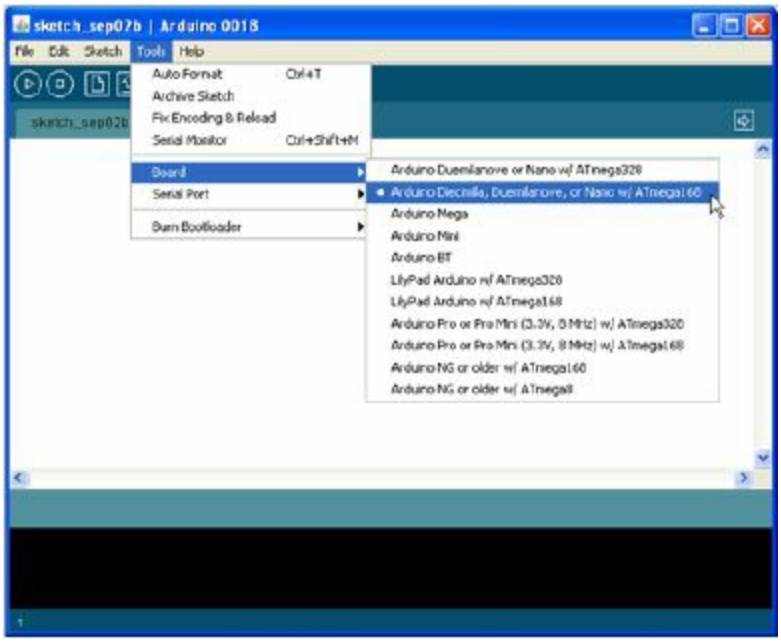
O *hardware* do Arduino é baseado nos microcontroladores AVR ATmega8, ATmega168, ATmega328 ou ATmega1280. Para se interligar ao mundo exterior o Arduino possui um conjunto de dois conectores: um com 14 pinos para entradas ou saídas de sinais digitais e um de 6 pinos para entrada de sinais analógicos.

Ports são a conexão fisica entre a CPU e o mundo exterior, são grupos de 8 pinos do microcontrolador que são fisicamente conectados a registradores de 8 bits no circuito digital de E/S e por onde podemos enviar ou receber qualquer combinação de níveis lógicos na forma de 1 byte. Cada *Port* tem dois registradores: o registrador de direção de dados DDR e o registrador de dados propriamente dito ou simplesmente PORT. O equivalente ao DDR nos microcontroladores PIC é o registrador TRIS. Quem comanda todas as linhas de controle tanto no DDR quanto no PORT é o Decodificador de Instruções que fica na CPU do microcontrolador. Um nível lógico alto numa linha do DDR configura o pino correspondente como saída, um nível baixo como entrada. Nos microcontroladores PIC essa configuração é invertida.

O ATmega328 possui um conversor analógico-digital (A/D) de 10 bits de resolução no *Port* C com seis entradas que pode receber qualquer nível de tensão entre 0 e 5 volts. No Arduino essas entradas analógicas são os pinos A0 a A5 (ou pinos 14 a 19). Para gerar voltagens analógicas de 0 a 5 volts em alguns de seus pinos de saídas digitais o ATmega328 emprega uma técnica conhecida por PWM ou Modulação por Largura de Pulso. No Arduino esses pinos PWM são os pinos digitais 3, 5, 6 e 9 a 11.

No próximo capítulo vamos conhecer o Ambiente de Desenvolvimento Integrado, ou IDE (*Integrated*

Development Environment), do Arduino, onde você leitor vai poder editar e depurar os programas, ou *sketches*, que serão gravados na memória *Flash* do ATmega328.



Capítulo 3

O IDE do Arduino

Introdução

No capítulo 2 vimos que o Arduino é composto por duas partes principais: uma plataforma de *hardware* com um programa residente em parte de sua memoria *Flash*, o seu *bootloader,* e tambem um aplicativo gráfico que roda em qualquer computador PC com *Windows* ou *Linux*, ou num computador *Apple* com *Mac OS-X,* o seu Ambiente de Desenvolvimento Integrado, ou IDE. Nesse capítulo vamos conhecer em detalhes o que é e para que serve esse IDE, que pode ser baixado sem nenhum custo na página *web* oficial do Arduino em **www.arduino.cc/en/Main/Software**. Observe o leitor que as palavras *Main* e *Software* são grafadas com iniciais maiúsculas.

Instalação do IDE do Arduino

Se o seu PC é *Windows*, saiba que o IDE do Arduino roda sem problemas em qualquer versão desse sistema operacional, desde o antigo*Windows 95* até o *Windows 7*. Comece criando uma pasta *Arduino* em *C:\Arquivos de Programas*. Da página *web* do Arduino baixe o arquivo *.zip* do IDE para *Windows* para dentro da pasta criadae o descompacte com um duplo-clique e veja que

automaticamente é criada uma outra pasta com o nome *arduino-0021*, onde os quatro dígitos representam a versão mais recente do IDE. Se quiser você pode até apagar o arquivo compactado original. Por fim, crie um atalho do executável *arduino.exe*, que está dentro dessa pasta recem-criada, para sua área de trabalho.

O IDE é um conjunto de aplicativos acessíveis por uma única interface gráfica, uma só tela principal, para a edição, depuração, compilação e gravação de programas na parte restante da memória *flash* do seu Arduino. A tela principal do IDE do Arduino é aberta com um só clique no atalho que está na sua área de trabalho. Veja a figura 8.

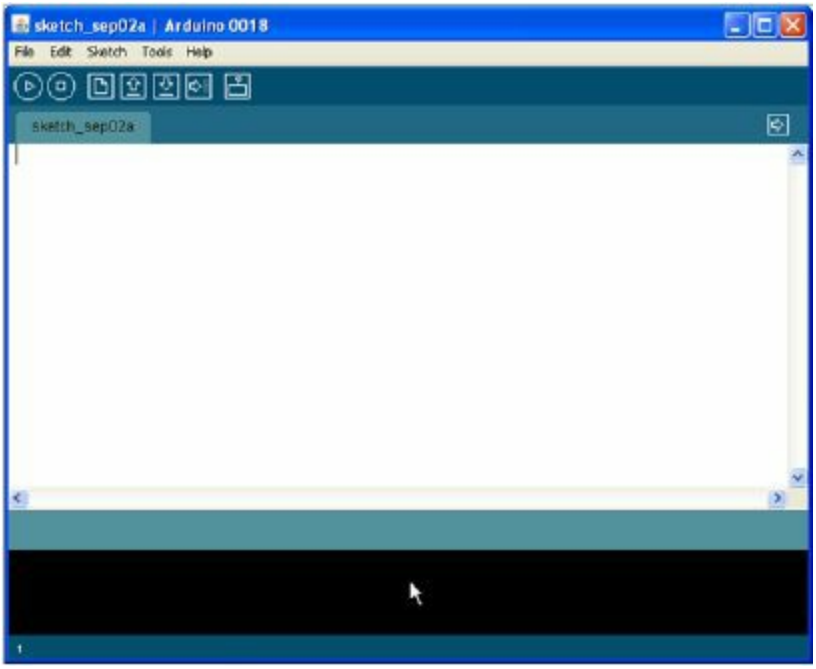


Figura 8 - O IDE do Arduino

A primeira coisa que você precisa fazer no IDE é selecionar em uma lista de menu o modelo da placa do Arduino que você leitor possui. Para isso, na barra de menus, clique na opção **Tools** e selecione **Board**. Na lista de opções que abrir selecione o modelo do seu Arduino. Se, por exemplo, a sua placa for o modelo *Duemilanove* com o ATmega328 selecione a segunda opção, como na figura 9. Se você tiver um modelo mais antigo com o ATmega8 ou ATmega168 selecione uma das duas últimas opções.

Agora é necessário configurar a porta serial de comunicação. Se o seu computador e o seu Arduino tiverem uma porta RS-232C, clique na barra de menus em *Tools* e na opção *Serial Port*, logo abaixo da opção *Board,* e selecione na lista que surgir qualquer das portas RS-232 disponíveis, COM1 ou COM2, como na figura 10.

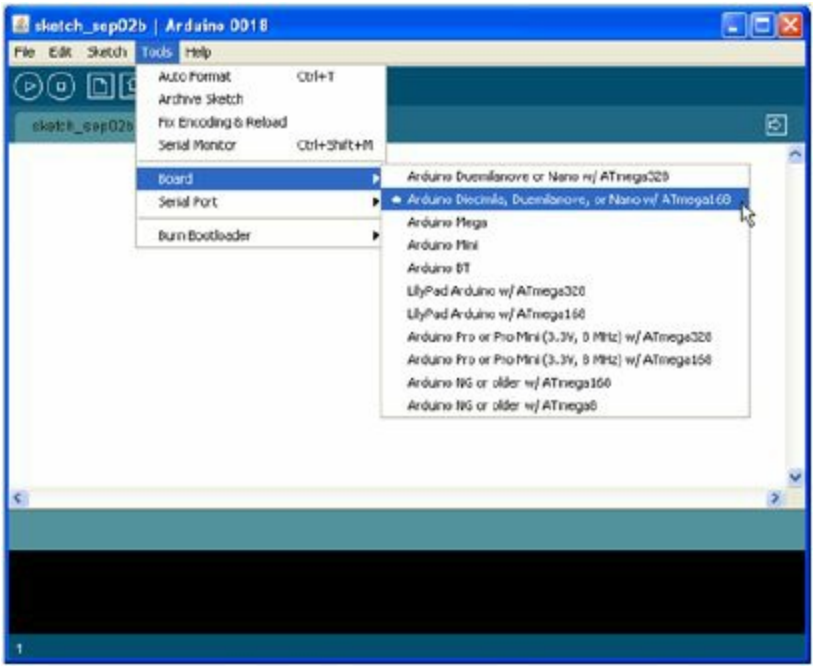


Figura 9 - Configuração do modelo da placa do Arduino

Se o seu Arduino tiver uma porta USB você precisará instalar um *driver*, um programa que vai permitir a comunicação do Arduino com o seu computador. A maneira mais fácil de instalar o *driver* no Windows é simplesmente conectando o Arduino no seu computador atraves do cabo USB. O IDE vai reconhecer a porta USB e no menu **Tools > Serial Port** deverá aparecer a porta COM3, como na figura 10.

Tudo pronto. Agora alimente o Arduino normalmente. Vamos começar analisando um pequeno programa que é gravado na memória *flash* do Arduino junto com o seu *bootloader*. Esse programa, **Blink.pde**, faz piscar continuamente o LED verde que normalmente está conectado ao pino 13 do Arduino. Para ver o código fonte desse programa, ou *sketch* como são conhecidos os programas do Arduino, na janela do IDE clique em *File,* a primeira opção da barra de menus; depois em *Examples* para ver o primeiro nível de uma lista de *sketches* prontos; em seguida clique em *Basics* para listar o segundo nível dessa lista; por fim clique no nome do *sketch*, *Blink.* Sequencialmente: **File > Examples > Basics > Blink**. A listagem em linguagem C desse *sketch* vai surgir no IDE.

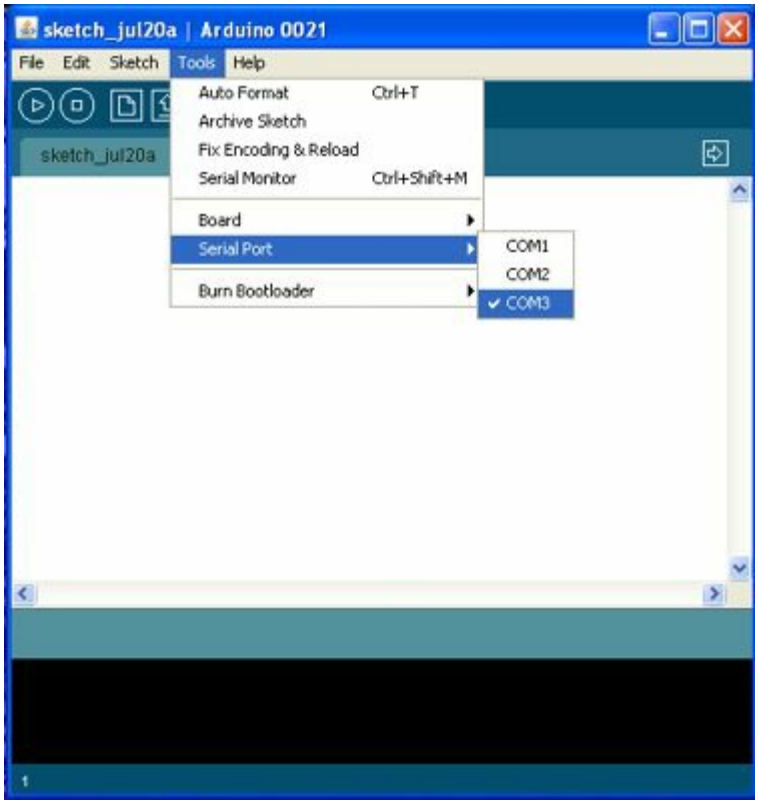


Figura 10 - Seleção da porta serial de comunicação com o Arduino

Se você nunca programou em C, não se preocupe, no capítulo 5 você vai conhecer os principais comandos dessa linguagem e vai poder escrever seus primeiros *sketches*.
Logo nas primeiras linhas do código fonte listado no IDE vemos que entre os caracteres /* e */ existe um pequeno texto que explica o que o *sketch* faz, são comentários que são ignorados quando da compilação do programa que vai ser gravado no Arduino.

Igualmente, são comentários de uma só linha os textos escritos após os caracteres //.

Esse *sketch* configura o pino 13 do Arduino como saída digital, onde está conectado um LED verde. Tudo o que esse *sketch* faz é acender o LED por um segundo, apagá-lo durante mais um segundo e repetir esse processo indefinidamente.

A parte do *sketch* ***Blink*** que faz isso está na Listagem abaixo:

```
void loop()  {
  digitalWrite(13, HIGH);          // set the LED on
  delay(1000);                 // wait for a second
  digitalWrite(13, LOW);           // set the LED off
  delay(1000);                 // wait for a second
  }
```

A função *digitalWrite (13, HIGH)* envia uma tensão de 5 volts (*HIGH*=5 volts) para o pino 13 para acender o LED verde; a função *delay (1000)* pausa o programa por 1000 milissegundos (1 segundo); a função *digitalWrite (13, LOW)* envia 0 volt (*LOW*=terra) para o pino 13 e apaga o LED; e

novamente ocorre uma pausa no programa de 1 segundo. No capítulo 4 veremos essas e outras funções da linguagem C do Arduino em mais detalhes.

Agora maximize a janela do IDE e modifique somente essa parte do *sketch* entre as chaves '{' e '}' da função *loop( )*, acrescentando mais quatro linhas conforme a listagem abaixo:

```
void loop()   {
   digitalWrite(13, HIGH);            // acende o led no pino 13
   delay(2000);                    // espera 2 segundos
   digitalWrite(13, LOW);             // apaga o led
   delay(200);                       // espera 200 milisegundos
   digitalWrite(13, HIGH);            // acende o led
   delay(200);                       // espera 200 milisegundos
   digitalWrite(13, LOW);             // apaga o led
   delay(200);                      // espera 200 milisegundos
   }
```

Primeiro verifique se não tem nenhum erro na edição do *sketch* clicando no botão *Verify* na barra de controles do IDE, ele é o primeiro à esquerda, com um triangulo inscrito num círculo. Durante a verificação a mensagem **Compiling...** vai surgir na barra de *status*, logo abaixo da janela de edição do IDE, e se não houver erros de edição logo vai surgir nessa mesma barra a mensagem **Done compiling**.

Após essa mensagem seu código foi compilado, ou seja, convertido sem erros para linguagem de máquina do ATmega328 por um programa imbutido no IDE do Arduino, o compilador AVR-GCC.

Para gravar o programa na memória *Flash* do Arduino clique no botão **Upload**, aquele com a seta para a direita dentro de um quadrilátero. Espere alguns segundose... observe o LED verde do seu Arduino! O que aconteceu? Você acaba de fazer a sua primeira programação no Arduino! O LED fica aceso por dois segundos e dá duas piscadas de 200 milisegundos. Seu código compilado foi gravado no microcontrolador e executado logo após o Arduino ser automaticamente resetado.

Como você viu, o IDE do Arduino é uma interface gráfica muito fácil de usar. No topo do IDE fica uma barra azul com o nome do *sketch* aberto e a versão do IDE; à direita os clássicos botões para minimizar, expandir ou fechar o aplicativo. A seguir vem a barra de menus com somente cinco títulos, e logo abaixo dela uma barra de controle com sete botões. O título do *sketch* carregado aparece tambem numa aba logo acima da janela de edição. O botão com uma seta, no extremo direito do título, serve para você carregar e editar outros *sketches* no mesmo IDE. Na parte de baixo do IDE existem tres outras barras, a de *status*, a de mensagens diversas e uma que indica a linha onde está o cursor de edição. Veja figura 11.

Vamos conhecer em mais detalhes cada um desses controles do IDE do Arduino.

A barra de Menus

A barra de menus do IDE do Arduino possui cinco títulos: *File, Edit, Sketch, Tools* e *Help*.

FILE - O primeiro título é *File* e corresponde à opção **Arquivo**, primeira opção padrão em todo aplicativo *Windows*, e suas opções, com as teclas de atalhos entre parenteses, são as seguintes:

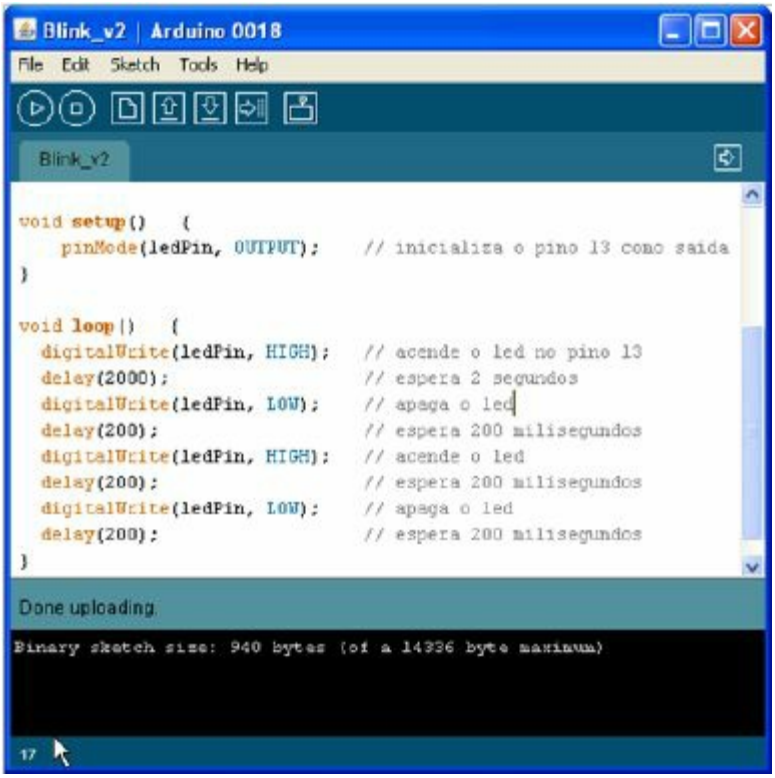


Figura 11 - O IDE com o sketch Blink e as barras de mensagens

New (Ctlr+N)

Abre uma nova janela de edição de *sketches* sobre aquela já ativa na tela do IDE.

Open... (Ctrl+O)

Abre uma janela para localizar e carregar um *sketch* existente em alguma pasta no seu computador.

Sketchbook

Abre a pasta de projetos de *sketches*. Veja *Preferences* abaixo.

Examples

Abre uma lista com uma coleção de *sketches* prontos para você carregar, eventualmente modificar, e testar.

Close (Ctrl+W)

Fecha a tela ativa do IDE.

Save (Ctrl+S)
Salva com o mesmo nome o *sketch* ativo na janela de edição.

Save as... (Ctrl+Shit+S)
Salva o *sketch* ativo com outro nome.

Upload to I/O Board (Ctrl+U)
Grava na memória *Flash* do Arduino o *sketch* ativo. Faz a mesma função do botão *Upload* na barra de controle.

Page Setup (Ctrl+Shift+P)
Abre uma janela de configuração do formato de impressão do *sketch* ativo.

Print (Ctrl+P)
Abre a janela de impressão.

Preferences (Ctrl+,)
Abre uma janela para apontar a localização da pasta de *sketches*, entre outras opções.

Quit (Ctrl+Q)
Fecha todas as janelas e sai do IDE.

EDIT - O segundo título da barra de menus é *Edit* e corresponde à função **Editar** em todo aplicativo *Windows*. Tem as seguintes opções e atalhos:

Undo deletion (Ctrl+Z)
Cancela ou desfaz a última digitação na janela de edição ou o último coman do realizado.

Redo (Ctrl+Y)
Repete a última digitação na janela de edição ou o último comando realizado.

Cut (Ctrl+X)
Copia para o *clipboard* e apaga a área marcada na janela de edição.

Copy (Ctrl+C)
Copia para o *clipboard* sem apagar a área marcada na janela de edição.

Copy for Forum (Ctrl+Shift+C)
Copia para o *clipboard* e formata para publicação no Forum do Arduino o texto marcado na janela de edição.

Copy as HTML (Ctrl+Alt+C)
Formata em *HTML* e copia no *clipboard* o texto marcado na janela de edição.

Paste (Ctrl+V)
Copia na janela de edição o texto no *clipboard*.

Select All (Ctrl+A)
Seleciona todo o texto na janela de edição.

Comment/Uncomment (Ctrl+/)
Coloca duas barras no início da linha onde está o cursor transformando-a em comentário. Ou retira as duas barras desfazendo o comentário existente.

Increase Indent (Ctrl+])
Desloca todo o texto na linha onde está o cursor duas posições à direita.

Decrease Indent (Ctrl+[)
Desloca todo o texto na linha onde está o cursor duas posições à esquerda.

Find… (Ctrl+F)
Abre uma janela para pesquisa de caracteres na janela de edição e, opcionalmente, os substitui.

Find Next (Ctrl+G)
Busca dos caracteres seguintes na pesquisa com *Find...*

SKETCH - É o terceiro título da barra de menus e tem somente cinco opções:

Verify / Compile (Ctrl+R)
Verifica a consistência do código fonte na janela de edição e o compila.

Stop
Interrompe qualquer processo em andamento no IDE.

Show Sketch Folder (Ctrl+K)
Mostra o conteúdo da pasta com os *sketches* abertos.

Import Library…
Inclui uma ou mais bibliotecas no código fonte na janela de edição.

Add File…

Abre um novo *sketch* em uma outra aba no IDE.

TOOLS - O próximo título da barra de menus é *Tools*, ferramentas, com as seguintes opções:

Auto Format (Ctrl+T)
Alinha automaticamente em colunas todo o texto na janela de edição.

Archive Sketch
Abre a pasta de projetos de sketches. Veja *Preferences* na barra de menus.

Fix Encoding & Relod
Corrige pequenos erros de sintaxe no código fonte carregado no IDE.

Serial Monitor (Ctrl+Shift+M)
Abre o aplicativo Terminal de Comunicação Serial. Usado para monitoração de Tx e Rx serial.

Board
Abre as opções de modelos de placas Arduino para configuração do IDE.

Serial Port
Abre as opções de portas seriais disponíveis no computador para configura ção do IDE.

Burn Bootloader
Abre as opções de *hardware* de programadores de *bootloader* para o Arduino. Veja o apendice 1 no final do livro.

HELP - O último título da barra de menus é *Help* e corresponde ao Ajuda nos aplicativos *Windows*. As primeiras opções abrem páginas *HTML* de ajuda armazenadas na subpasta *reference* da pasta *arduino-0021* que foi criada automaticamente quando você instalou o IDE do Arduino. A opção *Visite Arduino.cc* abre o site oficial do Arduino na *web*. A última opção *About Arduino* abre uma janela com os nomes dos desenvolvedores do projeto Arduino.

A Barra de Controles

Figura 12 - A Barra de Controles do IDE

A barra de controles tem sete botões, alguns repetem as funções principais da barra de menus, vistos

acima. Se você posicionar o cursor do mouse sobre cada botão dessa barra ele é destacado e a sua função aparece à direita do último botão. Veja figura 12. Suas funções são as seguintes, da esquerda para a direita:

Verify
Verifica a consistência e compila o código fonte na janela de edição.

Stop
Interrompe qualquer processo em andamento no IDE.

New
Abre um novo aplicativo IDE sobre aquele já ativo na tela.

Open
Abre uma janela para localizar e carregar um *sketch* novo.

Save
Salva o *sketch* ativo na janela de edição.

Upload
Grava no Arduino o *sketch* ativo.

Serial Monitor
Abre o aplicativo Terminal de Comunicação Serial, usado para monitoração de Tx e Rx serial.

O botão com uma seta para a direita que aparece alinhado com a aba com o nome do *sketch* serve para você abrir outras abas e editar outros *sketches* no mesmo IDE.

As Barras de Mensagens

Por fim, o IDE do Arduino tem em sua parte inferior tres barras de mensagens. A primeira é a barra de *status*, normalmente azul, onde no IDE aparecem informações durante o processo de compilação. Esta barra muda para marron se uma mensagem de erro for mostrada. A segunda barra é sempre preta e exibe mensagens em letras brancas com o tamanho do arquivo compilado; ou em letras vermelhas se ocorrereram erros durante o processo de compilação do *sketch*. A última barra informa a posição do cursor na janela de edição. Veja figura 13.

Figura 13 - As Barras de Mensagens do IDE

```
int i=0;                    //contador de pinos e segmentos
int j=0;                          //contador de digitos
int pinoDigital[7]={5,6,7,8,9,10,11}; //matriz 1D com os pinos usados
    // segmentos:  a b c d e f g
int digito[10][7]={   //↓ ↓ ↓ ↓ ↓ ↓ ↓    matriz 2D com os segmentos
    {1,1,1,1,1,1,0},      //digito '0'
                    {0,1,1,0,0,0,0},     //digito '1'
                    {1,1,0,1,1,0,1},    //digito '2'
                    {1,1,1,1,0,0,1},     //digito '3'
                {0,1,1,0,0,1,1},     //digito '4'
    {1,0,1,1,0,1,1},      //digito '5'
                {1,0, 1,1,1,1,1},       //digito '6'
                {1,1,1,0,0,0,0},       //digito '7'
                {1,1,1,1,1,1,1},       //digito '8'
                {1,1,1,1,0,1,1}      //digito '9'
   };
void setup()  {
 for(i=0; i<7; i++)
 pinMode(pinoDigital[i],OUTPUT);  //cada pino como saida
 pinMode(12,OUTPUT);     //pino 12 saida, base do transistor
```

Capítulo 4

A linguagem de Programação do Arduino

Introdução

Conhecemos o IDE do Arduino no capítulo 3, um ambiente integrado de desenvolvimento onde você, leitor, vai escrever os *sketches* que vão ser gravados pelo *bootloader* no Arduino na memoria de programas do ATmega328. Neste capítulo vamos conhecer a sintaxe dos elementos mais comuns da linguagem de programação do Arduino, e sedimentar esse conhecimento com a montagem de alguns circuitos simples de interação com o mundo físico. A linguagem de programação do

Arduino é derivada da linguagem C e estruturada da mesma forma que outra linguagem de código aberto, a linguagem *Processing*, voltada para criações visuais interativas e criada em 2001 pelos programadores gráficos, Casey Reas e Benjamin Fry, do *MIT* (*Massachusetts Institute of Technology*). No próximo capítulo veremos como nosso Arduino pode controlar aplicações visuais escritas em *Processing*.

A estrutura da linguagem do Arduino

Constantes

Na linguagem de programação do Arduino tres grupos de valores são predefinidos e chamados de **constantes**, e porisso não mudam nunca, são eles:

TRUE/FALSE (verdadeiro/falso) – Verdadeiro e falso são constantes chamadas *booleanas* (relativas a álgebra de Boole) que definem estados lógicos 1 e 0, ligado e desligado. Verdadeiro é tudo que for diferente de zero; e falso o que for zero.

HIGH/LOW (alto/baixo) – Essas constantes definem os níveis de tensão nos pinos do Arduino. Alto é o nível de +5 volts, ligado; baixo é o nível de terra, ou zero volt, desligado.

OUTPUT/INPUT (saída/entrada) – Essas constantes são usadas com a função *pinMode( )* para definir o modo como um pino qualquer do Arduino será configurado, se como saída (*output*) ou como entrada (*input*).

Variáveis

Variáveis são nomes que os programas associam a posições de memória. Como a própria palavra sugere, o conteúdo dessas posições de memória pode mudar durante a execução do programa. Todas as variáveis devem ser declaradas logo no início do programa, antes de ser chamadas. Declaramos uma variável definindo para ela um **tipo**, um **nome** e, opcionalmente, um **valor inicial** que pode ser mudado por funções dentro do programa. Os nomes dados às variáveis devem sempre começar por uma letra ou um traço sublinhado, como em *_ledPin*. Na linha de código abaixo, do *sketch* **Blink** que vimos no capítulo 3, definimos uma variável com o nome *ledPin* do tipo **int** e lhe atribuimos o valor 13.

```
int ledPin = 13;
```

As variáveis podem ser do tipo **int** (*integer*), que armazenam números inteiros, sem casa decimal, positivos ou negativos de até 2 bytes na RAM; tambem podem ser do tipo **long** (*long*), que guardam valores inteiros de até 4 bytes; ou podem ser tambem do tipo **boolean**, para valores de até 1 byte; podem ser tambem do tipo **float** (*float*), que guardam valores em ponto flutuante, com casas

decimais, de até 4 bytes; ou ainda podem ser do tipo **char** (*character*) de 1 byte que guardam um caracter ASCII.

Funções

Todo programa, ou *sketch*, do Arduino é composto por duas partes principais: a função **setup( )** e a função **loop( )**. Função é um conceito que vem da Matemática e que relaciona um **argumento** a um **valor numérico** por meio de uma regra de associação, ou formula. Funções em linguagens de programação são como subrrotinas ou procedimentos, são pequenos blocos de programas usados para montar oprograma principal. Em C todas as funções recebem um nome seguido por um par de parenteses, como em **main( )**. Elas podem ser escritas pelo programador para realizar tarefas repetitivas simples ou podem ser incorporadas prontas ao programa principal em forma de bibliotecas (*library*, em ingles). As funções podem ou não computar valores; as primeiras retornam um valor como resultado de uma operação; as segundas não retornam valor nenhum, podem apenas executar outras funções que tambem podem ou não retornar valores. Na linguagem C a função *main( )* é a única obrigatória; na linguagem de programação do Arduino as funções *setup( )* e *loop( )* são as únicas obrigatórias.

Toda função deve ter um nome, e os procedimentos que ela vai executar poderão vir entre uma chave de abertura '**{**' e uma de fechamento '**}**', como na função *setup( )*
abaixo, do *sketch* **Blink**, que chama a função *pinMode( )* para configurar o pino 13 do Arduino como saída.

```
void setup( )   {
      pinMode(13, OUTPUT);
     }
```

Observe que o corpo da função, os procedimentos entre as chaves '{' e '}', deve obrigatoriamente ser finalizada com ponto-e-vírgula; sua omissão causará erro quando da compilação do *sketch*. A declaração **void** antes da função *setup( )* indica que nenhum valor é retornado por essa função, ela apenas chama uma outra função que configura um pino do Arduino. Dentro dos parenteses do nome da função vêm os parâmetros que a função deve receber; no fragmento de código acima, a função *pinMode( )* recebe dois parâmetros, o número do pino e como ele deve ser configurado; já a função *setup( )* não recebe parametro nenhum e é a primeira a ser chamada, e apenas uma vez, quando o *sketch* é executado.

A função *loop( )* é chamada logo depois da função *setup( )* e todos os seus procedimentos para controle do fluxo de dados nos pinos do Arduino são executados repetidamente, daí o nome *loop*. Vejamos de novo o conteúdo da função *loop( )* no *sketch* **Blink**:

```
void loop()   {
    digitalWrite(13, HIGH);    // set the LED on
    delay(1000);                // wait for a second
```

```
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

A função *loop( )* tambem não recebe nenhum parâmetro e tambem não retorna nenhum valor. Essa função no entanto chama duas outras funções, a *digitalWrite( )* e a *delay( )*. A primeira coloca +5 volts no pino 13 do Arduino quando recebe o parâmetro HIGH, e tambem coloca terra nesse mesmo pino quando recebe o parâmetro LOW. Isso faz com que o LED conectado no pino 13, atraves de um resistor limitador de corrente, acenda ou apague. A segunda função chamada, *delay( )*, faz o *sketch* pausar por um período definido pelo parâmetro que recebe, aqui de 1000 milisegundos, ou um segundo.

Todo texto na mesma linha depois das duas barras '//' é somente comentário e é ignorado pelo compilador no IDE do Arduino. Blocos de comentários, com duas linhas ou mais, são colocados entre '/*' e '*/' e aparecem quase sempre no cabeçalho do código do programa com o nome do *sketch* e do seu autor, da data de sua criação e um resumo do que o *sketch* faz.

As principais funções da linguagem do Arduino

Funções em qualquer linguagem de programação de computadores (ou de microcontroladores) são blocos de código da linguagem com atribuições bem especificadas. Um bloco pode, por exemplo, ler e armazenar informações disponíveis nas entradas de portas físicas do computador; outros blocos de códigos podem realizar cálculos com esses dados e os resultados podem ser enviados a outros blocos que depois de os formatar os enviarão para algum tipo de mostrador externo, como uma tela de LCD. Termos como procedimentos, subprogramas ou subrrotinas e métodos que são normalmente dados a esses blocos são essencialmente funções da linguagem.

São 10 as principais funções da linguagem de programação do Arduino. Com essas 10 funções o leitor poderá montar os 8 experimentos do capítulo seguinte. Outras funções vão aparecendo ao longo do caminho. Existe funções que são agrupadas em bibliotecas que você pode importar para o seu *sketch*; veremos isso ainda nesse capítulo. As 10 principais funções da linguagem do Arduino são as seguintes:

1. pinMode(pino,modo);
2. digitalRead(pino);
3. digitalWrite(pino,valor);
4. analogRead(pino);
5. analogWrite(pino,valor);
6. delay(ms);
7. millis( );
8. random(min,max);
9. Serial.begin(taxa);

10.Serial.println(dado);

As 5 primeiras funções formam o bloco de controle de entrada (sensores) e de saída (atuadores). As outras cinco são funções utilitárias. Vejamos cada uma delas.

1. pinMode(pino,modo)

Um pino é um canal físico atraves do qual o Arduino vai se comunicar com o mundo externo. O Arduino tem 20 desses canais físicos, sendo 14 digitais e 6 analógicos, que são numerados de 0 a 13 e de 14 a 19, respectivamente. Esses 6 pinos analogicos podem ser configurados tambem como pinos digitais, totalizando assim 20 pinos digitais. Nos pinos digitais só são permitidos dois tipos de sinais: ou alto (+5 volts) ou baixo (terra). Se o sinal vem do Arduino o pino está configurado como saída; se o sinal vem de um circuito externo, o pino é entrada. A função *pinMode( )* é sempre usada dentro da função *setup( )* e serve para estabelecer a direção do fluxo de informações em um determinado pino do Arduino. Essa função recebe dois parâmetros, um é **pino**, o número do pino que queremos configurar; o outro parametro é **modo** que determina a direção do fluxo de dados, se do Arduino para fora ou se de fora para o Arduino. Já conhecemos essa função quando vimos o *sketch* **Blink**:

```
pinMode(13, OUTPUT);
```

Aqui ela configura o pino 13 do Arduino para ser um canal de saída para um circuito externo, no caso do **Blink**, um LED. Se quisermos que pino 2 se comporte como entrada para o Arduino, escrevemos:

```
pinMode(2, INPUT);
```

Depois que o Arduino é resetado todos os seus pinos são automaticamente configurados como entrada; desse modo, após o *reset*, não há necessidade de explicitamente declarar qualquer pino como entrada com a função *pinMode( )*. Quando um pino está configurado como entrada, ele fica em estado de alta impedância esperando que uma voltagem qualquer o afete. Nesse estado até mesmo a eletricidade estática do seu dedo pode afetar o pino. Quando esse pino está como saída seu estado é de baixa impedância, e quando alto, pode suprir até 40 miliamperes de corrente para um circuito externo a ele conectado.

2. digitalRead(pino)

Essa função lê o estado lógico de um pino do Arduino que foi configurado previamente como entrada. Ela é usada para ler o estado de uma chave ou qualquer outro elemento que retorne 1 ou 0, verdadeiro ou falso, ligado ou desligado. No exemplo abaixo a variável **chave** do tipo *int* vai guardar o estado lógico lido no pino 2.

```
int chave = digitalRead(2);
```

3. digitalWrite(pino,valor)

Essa função envia para **pino** um nível lógico alto ou baixo, conforme especificado em **valor**. O pino deve ser configurado previamente como saída pela função *pinMode( )*, como em:

```
pinMode(13,OUTPUT);
digitalWrite(13,HIGH);
```

Aqui o pino 13, configurado como saída, recebe +5 volts com uma corrente máxima de 40 mA do Arduino, e pode acender um LED se em série com um resistor de 220 ohms limitador de corrente. Não é comum ver nos livros sobre Arduino, mas essas funções podem ser reescritas substituindo as constantes OUTPUT e HIGH por 1, assim:

```
pinMode(13,1);
digitalWrite(13,1);
```

Opostamente aos microcontroladores PIC da Microchip, as funções de controle de direção nos AVR da Atmel são configurados como saída com '1' e entrada com '0'.

Existe tambem uma facilidade interessante nos microcontroladores ATmega que é a configuração de resistores de 20Kohms de *pull-up* por *software*. É feito da seguinte forma:

```
pinMode(10,INPUT);    // ou pinMode(10, 0);
digitalWrite(10,HIGH);    // ou digitalWrite(10, 1);
```

Repare que, diferentemente dos exemplos anteriores, aqui não configuramos o pino 10 como saída, mas como entrada. A função *digitalWrite(10,HIGH)* apenas ativa um resistor interno de *pull-up* no pino 10, que foi configurado como entrada para uma chave mecânica, por exemplo.

4. analogRead(pino)

Essa função lê o valor de um dos 6 pinos analógicos do Arduino, e retorna um inteiro entre 0 e 1023 que representa o nível da tensão analógica entre 0 e 5 volts presente no pino especificado. Exemplo:

```
int sensor = analogRead(14);
```

Aqui a variável **sensor** guarda o valor convertido para um inteiro entre 0 e 1023 da tensão analógica no pino 14 do Arduino (pino analógico 0). Para uma tensão de 2,5 volts no pino 14, por exemplo, a função *analogRead( )* vai retornar o valor de 512 para a variável **sensor**. Você tambem pode escrever comando acima da seguinte forma:

```
int sensor = analogRead(A0);
```

O pino 14 é pino analógico A0, o pino 15 é A1; e assim até o pino 19 que é o pino analógico A5 do Arduino.

5. analogWrite(pino,valor)

Um Arduino com o ATmega328 possui 6 pinos, dentre os 14 pinos digitais, que tambem podem gerar saídas com *PWM* (*Pulse Width Modulation*). São os pinos 3,5,6,9,10 e 11. *PWM* é uma técnica para gerar tensões analógicas com sinais digitais. A função *analogWrite( )* gera uma onda quadrada de frequencia fixa no pino analógico especificado cuja largura do pulso, a parte positiva da onda, é por **valor**, um dos argumentos que lhe são passados. Essa onda quadrada modulada vai gerar tensões analógicas médias de 0 volt quando **valor** for 0, e 5 volts quando **valor** for 255. A função abaixo vai gerar uma tensão média de 2,5 volts no pino digital 10 do Arduino.

```
analogWrite(10,128);  // envia 2,5 volts para o pino 10
```

Note que, diferente dos pinos digitais, não é necessário configurar previamente um pino analógico como entrada ou mesmo como saída com a função *pinMode()*.

6. delay(ms)

A função *delay( )* interrompe o programa que está sendo executado por um período de tempo em milissegundos passado pelo parâmetro **ms**. Assim para pausar a execução de um *sketch* por 1 segundo escrevemos:

```
delay(1000);    // pausa de 1 segundo
```

Enquanto essa função está ativa todo o processamento no Arduino é suspenso, porisso uma outra função, vista a seguir, a função *millis( )*, é normalmente preferida para temporização.

7. millis( )

A função *millis( )*, diferentemente da função *delay( )*, não pede nenhum parâmetro e somente retorna o número de milissegundos decorridos desde o momento em que o programa foi iniciado. Como essa função não suspende as atividades do Arduino ela pode ser usada para medir o tempo entre dois ou mais eventos.

```
long timer = millis( );  // timer guarda o tempo decorrido desde          // o reset do Arduino
```

Note que essa função sempre vai ser do tipo *long* por requerer muita memória para armazenar longos períodos de tempo.

8. random(min,max)

A função *random( )* retorna um número aleatório entre os valores **min** e **max** passados como parâmetros. Exemplo:

```
int numRand = random(100,200);    // inteiro entre 100 e 200
```

9. Serial.begin(taxa)

Todo Arduino tem ao menos uma porta serial de comunicação, RS-232 ou USB. Essa comunicação é implementada utilizando a USART interna do microcontrolador ATmega328, e é feita pelos pinos digitais 0 e 1, sendo o primeiro o pino de recepção (RxD) e o segundo o de transmissão (TxD). A função *Serial.begin( )* abre esse canal serial de comunicação entre o Arduino e um computador PC ou Apple e recebe como parâmetro a taxa de transferência em bits por segundo (baud). Essa velocidade pode variar entre 300 a 115200 baud. Exemplo:

```
Serial.begin(9600);  // inicia porta serial em 9600 bits/s
```

Normalmente os pinos digitais 0 e 1 são reservados somente para essa comunicação serial, porisso raramente esses pinos são utilizados para entrada ou controle de dados externos.

10. Serial.println(dado)

Uma vez estabelecida a conexão serial entre o Arduino e o PC, a função *Serial.println( )* inicia a transmissão de caracteres ASCII no sentido do Arduino para o computador. Para monitorar essa transmissão de dados pressione o botão **Serial Monitor** na barra de controle do IDE do Arduino, e na janela do terminal que surgir na tela selecione a mesma taxa de transmissão especificada no programa. Vejamos um exemplo:

```
void setup( ) {
   Serial.begin(9600);  // estabelece conexão serial em 9600 bits/s
   }
void loop( )  {
    Serial.println(“ Hello World! “);  // inicia a transmissão serial da                  // frase ‘Hello World!’
   delay(2000);       // pausa de 2 segundos entre                // cada transmissão
   }
```

O *sketch* acima imprime na janela do terminal uma nova linha com a mensagem *Hello World!* a cada dois segundos.

Os operadores

Operadores são simbolos usados com operandos para se construir expressões. Por exemplo, na expressão **a=x+y** os simbolos '=' e "+" são um operadores e as letras 'x' e 'y' são operandos. O primeiro operador é o operador de atribuição, o segundo um dos operadores aritméticos. Podemos classificar os operadores da linguagem do Arduino em quatro grupos: os operadores **aritméticos**, os **lógicos**, os de **comparação** e os operadores **compostos**. Vejamos esses operadores em mais detalhes.

1. Operadores aritméticos

Os operadores aritméticos retornam o resultado de uma operação aritmética entre dois operandos. São eles:

Operador	Símbolo	Exemplo	Retorno
Soma	+	a = x + 2	'a' guarda o resultado da soma
Subtração	-	b = y - 3	'b' guarda o resultado da subtração
Produto	*	i = j * 4	o produto de 'j' e 4 é atribuido a 'i'
Divisão	/	n = k / 5	O quociente da divisão é atribuido a 'n'

2. Operadores lógicos

Esses operadores comparam duas expressões e retornam 1 ou 0 (verdadeiro ou falso). São tres os operadores lógicos:

Operador	Símbolo	Exemplo	Retorno
AND	&&	x > 0 && y < 3	Retorna 1 se as expressões forem verdadeiras
OR	||	x > 0 || y > 5	Retorna 1 se as expressões forem verdadeiras
NOT	!	! x > 0	Retorna 1 se a expressão for falsa

3. Operadores de comparação

Comparam duas variáveis ou constantes e retornam 1 ou 0 (verdadeiro ou falso). São seis os operadores de comparação no Arduino:

Operador	Símbolo	Exemplo	Retorno
Comparação	= =	x = = y	Retorna 1 se 'x' é igual a 'y', 0 se não
Diferença	! =	x ! = y	Retorna 1 se 'x' é diferente de 'y'
Menor que	<	x < y	Retorna 1 se 'x' é menor que 'y'
Maior que	>	x > y	Retorna 1 se 'x' é maior que 'y'
Menor ou igual	< =	x < = y	Retorna 1 se 'x' é menor ou igual a 'y'
Maior ou igual	> =	x > = y	Retorna 1 se 'x' é maior ou igual a 'y'

Importante: o leitor deve atentar que o operador "=" é um operador de atribuição e "= =" um operador de comparação; assim em x=5 o inteiro 5 é atribuido à variável x, enquanto em x= =5 a variável x é comparada ao inteiro 5.

4. Operadores compostos

Os operadores compostos combinam um operador aritmético com o operador de atribuição '=' para simplificar expressões:

Operador	Símbolo	Exemplo	Equivalência
Soma	+ =	x + = y	x = x+y
Subtração	- =	x - = y	x = x -y
Produto	* =	x * = y	x = x * y
Divisão	/ =	x / = y	x = x / y
Incremento	+ +	x + +	x = x + 1
Decremento	- -	x - -	x = x - 1

Os principais comandos da linguagem Arduino

A linguagem do Arduino é derivada da C e dela herdou os quatro principais comandos:

1. **if**
2. **while**
3. **do … while**
4. **for**

1. O comando if

O comando *if* é um comando de seleção, tambem chamado de comando condicional; os outros tres são comandos de iteração, tambem chamados de comandos *loops* (laço). Outros comandos da linguagem C aceitos pelo Arduino vão sendo comentados conforme surgirem nos *sketches* que apresentaremos. O comando *if* testa uma expressão para determinar se o seu resultado é verdadeiro ou se é falso. Verdadeiro é qualquer resultado, mesmo negativo, diferente de zero. Falso é um resultado zero.

Com base nesse teste o comando *if* executa todos os comandos entre as chaves '{' e '}' se a expressão for verdadeira; ou não se a expressão for falsa. A sintaxe do comando *if* é:

```
if (expressão)  {
bloco de comandos;
}
```

As chaves são usadas para agrupar comandos ou declarações em um único bloco, e cada linha de comando dentro do bloco deve obrigatoriamente terminar em ponto e vírgula. Opcionalmente podemos incluir um subcomando *else* que executa um segundo bloco de comandos se a expressão avaliada por *if* for falsa. A sintaxe é a seguinte:

```
if (expressão)  {
bloco de comandos1;
}
else  {
bloco de comandos2;
}
```

A linguagem C possui um operador, chamado de **operador ternário**, que pode simplicar bastante as sentenças que utilizam o operador de seleção *if*, é o operador '**?**'. Sua sintaxe é a seguinte:

```
expressão ? instrução1 : instrução2;
```

O operador ternário '?' avalia **expressão** e se esta for verdadeira **instrução1** é executado, se **expressão** for falsa então **instrução2** que é executado. Note o uso e a posição entre as duas instruções de dois pontos na sintaxe do operador. Vejamos um exemplo simples do uso desse comando:

```
int x = 8;
y = (x > 10) ? 15 : 20;
```

Aqui o valor de **y** vai depender da avaliação da expressão do operador ternário; como o valor de x vale 8, a expressão (x>10) é falsa, porisso o inteiro 20 será atribuido a y; se o valor atribuido a x fosse maior que 10, y seria 15. Essa mesma expressão com o comando *if* ficaria assim:

```
int x = 8;
if (x > 10) y = 15; else y = 20;
```

Observe que podemos eliminar as chaves '{' e '}' do operador *if* e do subcomando *else* quando o comando a ser executado é composto por somente uma linha.

2. O comando while

Em C os tres comandos de iteração – *while, do...while* e *for* – são formados por uma condição e um bloco de comandos. O comando *while* vai executar continuamente o bloco de comandos entre as chaves enquanto a condição for verdadeira (diferente de zero). Sua sintaxe é a seguinte:

```
while (condição) {
    bloco de comandos;
    }
```

No exemplo abaixo a variável **x** é decrementada a cada dois segundos a partir de 10 até atingir 0, quando a condição do comando *while* se torna falsa e o *loop* é quebrado. Teste-a carregando-a no Arduino e ativando seu monitor serial.

```
int x=10;       // o valor 10 é atribuido a x
void setup() {
   Serial.begin(9600);   // porta serial em 9600
   }
void loop() {
   while (x<=10 && x>0)  {   // verifica se x está                  // entre 0 e 10
    Serial.println(x);      // se sim, transmite o                 // valor de x
    delay(2000);      // pausa 2 segundos
    x--;        // decrementa x e volta                  // ao teste
```

```
    }   // se x>10 interrompe programa.
  }
```

3. O Comando do ... while

O comando *do ... while* funciona de forma parecida com o comando *while*, só que a condição agora é testada no final do bloco de comandos e não no início. Se a condição testada for verdadeira o bloco de comandos é executado mais uma vez.Repare que a linha **while(condição)** vem depois do bloco de comandos e é terminada com ';' (ponto-e-vírgula). No *do ... while* o bloco de comandos é executado ao menos uma vez, no início, independente da condição ser verdadeira ou falsa. Veja a sintaxe:

```
do {
bloco de comandos;
}
while (condição);
```

No exemplo abaixo o valor atribuido à variável **x** é primeiramente mostrado no monitor com *Serial.println(x)*, depois essa variável é incrementada com *x++* e somente apos uma pausa de 2 segundos é que a condição do comando é testada. Quando **x** alcança o valor de 10 os comandos entre as chaves não são mais executados, o *loop* é quebrado e o programa fica travado com o comando *while(1)*, um *loop* infinito. Experimente marcar como comentário esse *loop* infinito e execute de novo esse *sketch*.

```
int x=0;        // x é zerado
void setup() {
   Serial.begin(9600);
   }
void loop() {
   do {
     Serial.println(x);     //transmite o valor de x
     x++;         //incrementa x
     delay(2000);    //pausa de 2 segundos
       }
  while (x<10);    //se x<10 repete comandos acima
  while(1); { }      //se x=10, o programa trava.
  }
```

4. O Comando for

O comando *for* é usado para repetir um bloco de comandos um determinado número de vezes. Sua sintaxe:

```
for (inicialização; condição; incremento) {
```

```
comandos;
}
```

O primeiro termo entre parenteses, **inicialização**, geralmente é uma variável local que é inicializada com um valor inteiro e serve de parâmetro de controle para o comando *for*. O segundo termo, **condição**, é uma expressão relacional que determina quando o *loop* será quebrado. Por último, o termo **incremento** conta quantas vezes o *loop* deve ser repetido. No exemplo abaixo o LED no pino 13 do Arduino vai piscar 10 vezes, parar por um segundo e reiniciar.

```
void setup() {
 pinMode(13,OUTPUT); // pino 13 configurado como saida
 }
        void loop() {
          int i;       //declaração da variavel local i
          for (i=0; i <=10; i++) {  //inicialização, teste, incremento
  digitalWrite(13,HIGH);  //envia +5 volts para pino 13
  delay(200);      //pausa de 200ms
  digitalWrite(13,LOW);   //envia terra para o pino 13
  delay(200);      //pausa de 200ms
  }
        delay(1000);       //pausa de 1 segundo
        }        //repete toda operação.
```

Embora aqui declarada dentro da função *loop( )*, normalmente a variável local **i** é declarada logo no início do código do programa, antes da função *setup( )*.

Matrizes

Uma matriz é uma coleção de variáveis do mesmo tipo que podem ser individualmente acessadas por meio de um identificador, chamado de índice. Cada um desses elementos tem um endereço de sua posição dentro da matriz, que é referenciado por um número associado ao nome da matriz. As matrizes, como as variáveis, devem ser declaradas e podem tambem ser inicializadas com valores conforme o seu tipo. Sua forma geral é a seguinte:

```
int nomeMatriz [ ] = { valor1, valor2, valor3, ...}
```

As matrizes podem ser declaradas somente com o seu tamanho; os valores podem ser passados *a posteriori*, fazendo-se referencia aos índices dos seus elementos, da seguinte forma:

```
int nomeMatriz [ 10 ];  //matriz do tipo int com 10 elementos
nomeMatriz [ 4 ] = 64;  //inteiro 64 na 5ª posição de nomeMatriz
```

Observe que o inteiro 64 não é armazenado na 4ª posição de *nomeMatriz[ ]*, mas na 5ª porque a

posição 0 é que é a primeira posição da matriz.

Para recuperar um valor em uma posição qualquer de uma matriz usamos uma variável da seguinte forma:

```
x = int nomeMatriz [ 4 ];  // x = 64
```

Resumo do capítulo 4

Nesse capítulo vimos que:

- são **constantes** na linguagem do Arduino: TRUE/FALSE, HIGH/LOW e OUTPUT/INPUT; cada uma dessas constantes pode ser substituída no código fonte por **1/0**.

- **variáveis** são posições físicas na memória RAM, e porisso os conteúdos dessas posições podem mudar durante a execução do programa. As variáveis e seu tipo devem ser declaradas logo no início do programa.

- uma **função** é como uma subrrotina, um bloco de comandos desenhado para uma dada finalidade. Um conjunto de junções formam o programa principal.
- os **operadores aritméticos** retornam o resultado de uma operação aritmética entre dois operandos; como em (b= x + 2), onde a soma dos operandos x e 2 é atribuido a b.

- **operadores lógicos** comparam duas expressões lógicas e retornam 1 ou 0 (verdadeiro ou falso) como resultado da comparação; a expressão (!x > 0) retorna 1 se x não for maior que 0.

- os **operadores de comparação** comparam duas variáveis ou constantes e tambem retornam 1 ou 0 como resultado; a expressão (x= =3) retorna 1 se x for igual a 3.

- os **operadores compostos** combinam um operador aritmético com o operador de atribuição '=' para simplificar expressões; exemplo: i++ equivale a i = i+1.

- o comando **if** testa somente uma vez a expressão entre parenteses, se esta for verdadeira (diferente de zero) o bloco de comandos entre as chaves é executado.

- o comando **while** tambem testa a expressão entre parenteses, mas somente se a expressão for verdadeira o bloco de comandos entre as chaves é executado; após a execução do bloco a expressão é testada novamente e se esta ainda for verdadeira o mesmo bloco de comando é novamente executado.

- já no comando **do...while** o seu bloco de comandos é executado ao menos uma vez, porque sua condição não é testada logo no início do comando mas somente no final; e tambem enquanto esta for

verdadeira o mesmo bloco de comando é repetidamente executado.

- o comando **for** é usado para repetir um mesmo bloco de comandos um determinado número de vezes determinado por um contador que incrementa (ou decrementa) a cada execução.

- devemos sempre declarar uma **matriz** no início do programa com um nome e o tipo de valores que nela será armazenado; uma matriz pode ser inicializada já com todos os valores de seus elementos, ou somente com o número de elementos que vai receber, o seu tamanho; para armazenar um elemento devemos indicar o nome da matriz e a posição desse elemento dentro da matriz; para recuperar um elemento de uma matriz devemos nomear uma variável com o nome da matriz e a posição do elemento dentro dela.

Capítulo 5

Primeiras experiências com o Arduino

Introdução

Enfim já podemos testar nossa criatividade fazendo nossas primeiras experiências com *hardware* e com *software* com o nosso Arduino! Esse capítulo é dividido em 8 experimentos fáceis de ser montados e modificados pelo leitor com algum conhecimento de Eletronica e pouca experiência prática em programação de microcontroladores. Cada experimento é composto por uma montagem física (*hardware*) e uma ou mais montagens de códigos estruturados (*software*).

São esses os oito primeiros experimentos:

Experimento #1 – Hello Word!
Experimento #2 – Geração de Áudio
Experimento #3 – Entrada Analógica
Experimento #4 – Saída Analógica
Experimento #5 – Controle de Motores
Experimento #6 – Leds, Mostradores de 7 segmentos e LCD
Experimento #7 – Solenoides, Reles e Acopladores óticos
Experimento #8 – Sensores

Já vimos em capítulos anteriores que os *sketches* do Arduino são compostos obrigatoriamente pelas funções *setup( )* e *loop( )*, e tambem por um ou mais elementos de inicialização que aparece antes da primeira dessas funções. Esses elementos são compostos por linhas de código onde são declaradas as variáveis que vão ser usadas pelo programa. Uma variável declarada nessa posição tem o escopo de **variável global** e poderá ser chamada por qualquer função dentro do programa. Quando uma variável é declarada dentro de uma função tem o escopo de **variável local** e somente poderá ser usada por aquela função. A função *setup( )* é a primeira a ser chamada, e somente uma vez na primeira execução do *sketch*. Dentro dessa função definimos como cada pino do Arduino vai se comportar. A função *loop( )*, que é repetidamente chamada logo em seguida, contem as funções e variáveis que controlam o fluxo de entrada e saída de informações pelos pinos do Arduino.

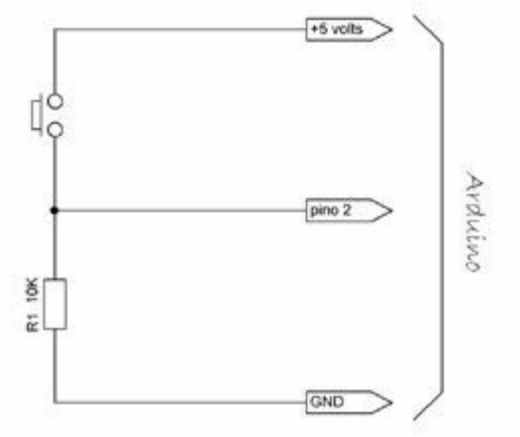
Experimento #1 – Hello Word!

Em 1978 foi publicado o primeiro livro sobre a linguagem C, "*The C programming Language*", onde seus autores, Brian Kernighan e Dennis Ritche, sugerem que o único caminho para se aprender uma nova linguagem de programação é escrever programas nessa linguagem. E o primeiro programa desse livro classico de pouco mais de 200 páginas era um código de poucas linhas que imprimia a mensagem "*hello, word*" no console de um computador PDP-11 da *Bell Laboratories*, nos EUA.

Era uma saudação da máquina aos humanos. A saudação do Arduino é dada pelo programa **Blink**, aquele que faz o LED no pino 13 piscar a cada segundo. No nosso primeiro programa na linguagem do Arduino no capítulo 4 nós modificamos o código fonte do *sketch* **Blink** para o LED no pino 13 dar duas piscadas; agora vamos modificá-lo novamente para controlar esse LED no pino 13 atraves de um outro pino que será configurado como entrada. Nesse pino conectamos uma chave mecânica do tipo campanhia (*push-button*) e dizemos para o Arduino só acender o LED quando essa chave for pressionada. Aqui está o código fonte:

```
int ledPin = 13;       // LED no pino 13
int buttonPin = 2;       // chave no pino 2
void setup( ) {
 pinMode(ledPin,OUTPUT);  // pino 13 como saída
 pinMode(buttonPin,INPUT);  // pino 2 como entrada
 }
```

```
void loop( ) {
  int buttonState = digitalRead(buttonPin);  // lê o estado da chave
  if (buttonState == HIGH) {    // se a chave for pressionada
    digitalWrite(ledPin,HIGH);  // acende o LED
    }
else {          // se não
  digitalWrite(ledPin,LOW);  // apaga o LED
  }
  }
```



Agora só precisamos acrescentar uma chave do tipo *push-button* e um resistor de 10 K ohms, conforme o circuito da figura 14, ao nosso Arduino. Quase todas as versões de *hardware* do Arduino já vem com um LED e um resistor de 220 ohms conectados no pino digital 13. Se o seu Arduino não tem esse LED você pode montar esses dois componentes numa matriz de contatos (*protoboard*) ou diretamente sobre os pinos do Arduino. A tensão de +5 volts pode ser tomado do conector POWER, ao lado do conector ANALOG, e o terra no pino GND, ao lado do pino 13.

Figura 14: Circuito para o experimento#1

As duas primeiras linhas do código formam o elemento de inicialização do *sketch*. Aqui são declaradas duas variáveis globais, *ledPin* e *buttonPin*, ambas do tipo **int**, e a elas são atribuidos valores inteiros que representam os pinos digitais 13 e 2, onde são conectados um LED e uma chave mecânica, respectivamente. A função *setup( )* chama duas vezes a função *pinMode( )*, e em cada chamada passa como parâmetros os pinos digitais que serão usados e as constantes *booleanas* que definem o modo de operação de cada um desses pinos - o pino 13 como saída e o pino 2 como entrada.

A função *loop( )*, que é chamada logo em seguida, contem o código que vai verificar continuamente o estado do pino 2. No corpo dessa função existe uma variável local, *buttonState*, do tipo **int**, que recebe como parametro o estado lógico do pino digital 2, que é lido pela função *digitalRead( )*. A decisão de acender ou não o LED é tomada dentro do comando *if*. Se a variável *buttonState* for verdadeira, isto é, se o estado do pino 2 for *HIGH*, a função *digitalWrite( )* vai enviar +5 volts para o pino 13 e assim acender o LED. Se essa variável local for falsa o pino 13 vai receber o nível terra

da função *digitalWrite( )* e o LED permanecerá apagado. É importante relembrar que o sinal simples de igual '=' é um operador de atribuição, enquanto o duplo igual '= =' é um operador de comparação.

Agora abra o IDE do Arduino no seu *PC* ou *Apple* e copie todo o código fonte para a janela de edição, e pressione o botão **verify** na barra de controle. Se não houver erros de digitação, após a compilação do código fonte, a barra de mensagens vai indicar que o tamanho do *sketch* é 968 bytes. Salve o *sketch* pressionando o botão **Save** (o quinto da esquerda para a direita) como, por exemplo, *HelloWorld*. Agora conecte o cabo RS232 (ou USB) entre o seu computador e o Arduino e ligue a fonte de alimentação. O LED no pino 13 não deverá acender ainda, a chave está aberta. Agora pressione a chave e veja se o LED acende. Solte a chave e veja se o LED apaga. Se tudo funcionou... Parabéns! Você acaba de testar seu primeiro programa no Arduino! Se não funcionou, tudo bem!, verifique se você fez as ligações nos pinos certos ou se não é problema de mau contato na fiação.

Agora, só para testar novas possibilidades, vamos reescrever nosso programa com algumas mudanças, a versão 2 do *HelloWorld*. Veja a listagem abaixo.

```
 #define LED 13      // define LED como o pino 13
 #define CHAVE 2     // define CHAVE como o pino 2
 void setup( ) {
   pinMode(LED,1);    // pino 13 como saída
 }
void loop( ) {
 if (digitalRead(CHAVE) == 1)   // verifica o estado da chave
 digitalWrite(LED,1);           // se fechada acende o LED
else                  // se não
 digitalWrite(LED,0);           // apaga o LED
 }
```

Neste exemplo as variáveis globais *ledPin* e *buttonPin* do *sketch* anterior foram substituidas pelas **diretivas do preprocessador** *#define*, que não tem ponto-e-vírgula no final da linha. Preprocessador é um pequeno programa que recebe o código fonte criado pelo programador, acrescenta alguns comandos, as **diretivas do preprocessador**, e cria um código fonte expandido. Este segundo código é passado para o compilador, que cria o código objeto final. No código acima toda vez que o preprocessador encontrar a palavra LED, por exemplo, vai substitui-la por 13.

Ainda no código acima repare que não configuraramos o pino 2 como entrada, por *default* todos os pinos do ATmega328 são entradas; e tambem as chaves '{' e '}' foram eliminadas nos comandos *if* e *else*, isso só é permitido quando o bloco de comandos é formado por somente uma linha.

Agora, caro leitor, veja a versão 3 ainda desse mesmo *sketch* listada abaixo. Mais enxuta ainda, não? Aqui o comando de seleção *if* foi substituido pelo **operador ternário** *expressão? comando1: comando2*. Inicialmente expressão testa o pino 2, se este for alto **comando1** é executado; se este pino for baixo é **comando2** que é executado. Carregue o código fonte no Arduino e teste-o.

```
 #define LED 13
```

```
void setup( ) {
  pinMode(LED,1);
  }
void loop( ) {
  (digitalRead(2) == 1)? digitalWrite(LED,1): digitalWrite(LED,0);
  }
```

Agora um desafio ao leitor: o que a versão 4 desse mesmo *sketch*, listada abaixo, faz?

Analise com cuidado cada linha e tente descobrir sua função. A dica está na primeira linha do código. Depois copie o código no IDE e após compilá-lo e carregá-lo no Arduino teste-o acionando e retendo por uns segundos a chave no pino 2.

```
int flipFlop = 1;
void setup( ) {
  pinMode(13,1);
   }
void loop( ) {
  if (digitalRead(2) == 1)
  digitalWrite(13,flipFlop);
  delay(100);
  flipFlop = 1 - flipFlop;
  while (digitalRead(2) == 0) { }
  }
```

E essa outra versão abaixo? Qual a diferença desta para a versão anterior?

```
int flipFlop = 1;
#define LED 13
void setup( ) {
  pinMode(LED,1);
  }
void loop( ) {
  (digitalRead(2) == 1)? digitalWrite(LED,flipFlop)
  :digitalWrite(LED,!flipFlop);
  while (digitalRead(2) == 0) { }
  }
```

Experimento #2 – Geração de Áudio

Nesse experimento vamos fazer barulho. Mantenha o circuito da chave no pino 2 do experimento anterior e conecte um pequeno *buzzer* entre o pino digital 3 e o terra do Arduino. Você tambem pode usar no lugar do *buzzer* um pequeno alto-falante em série com um capacitor eletrolítico de 1uF x 50 volts. Edite o código fonte abaixo no IDE e o carregue no Arduino. Quando a chave for pressionada o *buzzer* vai emitir um sinal sonoro intermitente com intervalos de 500 microsegundos.

```
void setup( ) {
  pinMode(3,OUTPUT);
  }
void loop( ) {
  while (digitalRead(2) == 1) {    // se chave pressionada
    digitalWrite(3,HIGH);     // envia 5 volts para pino 3
  delayMicroseconds(500);    // por 500 us
  digitalWrite(3,LOW);       // envia terra para pino 3
  delayMicroseconds (500);    // por 500 us
  }
}
```

A função *delayMicroseconds( )* interrompe o programa que está sendo executado por um período de tempo em microssegundos passado pelo parâmetro entre os parenteses. Com o valor 500 para ambos os parâmetros das funções *delayMicroseconds ( )* a frequencia de áudio emitida pelo Arduino será de 1000 Hz. Experimente valores abaixo e acima de 500 nas duas funções. Com esses valores iguais a saída pelo pino 3 é uma onda quadrada simétrica; tente parâmentros diferentes nas duas funções para ouvir ondas assimétricas. Veja adiante o experimento #4 com sinais *PWM*.

A listagem a seguir mostra o mesmo *sketch* acima porem utilizando o operador ternário '?' chamando função criada *pulse( )* e a função *digitalWrite( )*.

```
void pulse( );      // declaração da função pulse( )
void setup( )  {
  pinMode(3,1);    // pino 3 como saida
  }
void loop( )  {
  (digitalRead(2) == 1)? pulse( ):digitalWrite(3,0);
            // chama pulse( ) se pino 2 for alto
  }
void pulse()  {          // definição da função pulse ( )
digitalWrite(3,1); delayMicroseconds(500);   // pino 3 alto por 500 us
digitalWrite(3,0); delayMicroseconds (500);  // pino 3 baixo por 500 us
  }
```

Observe que a função *pulse( )* foi declarada logo no início do programa, e ela é definida, ou seja, criada, no final do programa, depois da função *loop( )*. Nesse *sketch* essa função *pulse( )* é chamada pelo operador ternário dentro de *loop( )* se a chave no pino 2 for pressionada.

Existe uma função no Arduino própria para gerar tons de áudio, é a função *tone( )*, que gera uma onda quadrada numa determinada frequencia com ciclo de trabalho de 50%. A sintaxe dessa função é a seguinte:

```
  tone (pino, frequencia, duração);
```

O parâmetro **pino** recebe o número do pino do Arduino por onde vai sair o sinal sonoro; o parâmetro **frequencia** determina o número de pulsos gerados por segundo (*hertz*); e **duração** dá a duração do sinal em milissegundos. Se o parâmetro duração for omitido o tom gerado será contínuo.
Experimente o *sketch* abaixo com essa nova função, primeiro com a última linha da função *loop( )* desabilitada (com as duas barras de comentário) e depois com esta linha habilitada (sem //) e a penúltima linha desabilitada (com //).

```
void setup( ) {
  pinMode(3,1);
  }
void loop( ) {
  while (digitalRead(2) == 1)   // enquanto pino 2 for alto
  tone( 3,1000,2000 );      // gera um tom de 1Khz durante 2 s
  // tone(3,1000);
  }
```

Tente modificar o *sketch* acima para aceitar mais uma chave no pino 4, por exemplo, e programar uma delas para aumentar e a outra para diminuir a frequencia de saída. Você tambem pode criar funções que, quando chamadas por chaves mecânicas, gerem as frequencias das notas musicais. Experimente!

Experimento #3 – Entrada Analógica

Nos dois primeiros experimentos testamos nosso Arduino para enviar e receber níveis digitais. No primeiro um LED acendia quando o Arduino enviava ao seu pino 13 uma tensão de 5 volts, e apagava quando a esse pino era enviado terra. Fizemos o LED piscar criando um *sketch* que gerava uma onda quadrada, com uma frequencia bem baixa. Depois fizemos o Arduino gerar sinais sonoros controlados por uma chave mecanica conectada a um pino configurado como entrada. Nesses experimentos usamos as funções *pinMode( )* para configurar os pinos que vamos usar, *digitalWrite( )* para enviar um nivel de tensão para o pino usado como saida e *digitalRead( )* para ler o estado do pino configurado como entrada. No próximo experimento vamos testar a entrada analógica do Arduino e aprender a usar a função *analogRead( )* para ler um nível de tensão analógica em um pino configurado como entrada.

Vimos no capítulo 2 que o Arduino possui 6 pinos especiais que são reservados para entradas de sinais analógicos – são os pinos A0 a A5 (ou 14 a 19) do conector ANALOG. Esses pinos são as entradas de uma chave seletora analógica de 6 posições do conversor A/D interno do microcontrolador ATmega328. Esse conversor A/D tem uma resolução de 10 bits, isto quer dizer que a sua saída vai retornar um número inteiro entre 0 e 1023 de acordo com o nível da tensão na sua entrada.
Para testar qualquer porta analógica do Arduino vamos precisar de um potenciometro de 10 K ohms e tres pequenos pedaços de fios soldados aos seus 3 terminais. Os fios externos devem ser conectados nos pinos de 5 volts e de terra do Arduino; o fio do meio, do cursor do potenciomentro,

deve ser conectado à primeira entrada analógica, pino A0.

Nesse nosso experimento vamos aproveitar um dos *sketches* já prontos que vieram com a instalação do Arduino. Na barra de menus do IDE clique em **File > Examples > Analog** e carregue o *sketch* **AnalogInput**. Esse *sketch* lê a tensão no pino analógico A0 (pino 14) e faz com que a taxa das piscadas do LED do pino digital 13 varie em função do valor dessa tensão. O potenciometro conectado ao Arduino comporta-se como um circuito divisor resistivo de tensão. Quando o cursor do potenciometro estiver em um extremo o pino A0 vai receber 0 volt. Quando o cursor estiver no extremo oposto a entrada A0 receberá 5 volts. Outras tensões entre esses dois extremos vão aparecer no pino A0 de acordo com a posição do cursor. Veja a listagem daquele *sketch* reescrita abaixo.

```
void setup( ) {
  pinMode(13, 1);
  }
void loop( ) {
  digitalWrite(13, HIGH);     // acende led
  delay(analogRead(A0));  // pot determina pausa em ms
  digitalWrite(13, LOW);     // apaga led
  delay(analogRead(A0));        // pot determina pausa em ms
  }
```

Agora analise esse mesmo sketch reescrito abaixo usando um flip-flop virtual:

```
int flipFlop=1;        // variavel flipFlop inicializada
void setup( )  {
  pinMode(13, 1);
  }
void loop( ) {
  digitalWrite(13, flipFlop);    // acende/apaga led
  delay(analogRead(A0));        // pot determina pausa em ms
  flipFlop = !flipFlop;    // inverte ultimo estado do led.
  }
```

Nesse *sketch* a variável inteira *flipFlop* é inicializada com 1 e o pino 13 é configurado como saída; o pino A0 por *default* já é configurado como entrada quando o ATmega328 é resetado. Dentro de *loop( )* a função *digitalWrite( )* vai enviar 5 volts para o pino 13 e o LED vai acender; a função *delay( )* vai pausar o programa por um período proporcional ao valor lido pela função *analogRead( )* no pino analógico A0. Depois disso a variável *flipFlop* vai ter seu estado lógico complementado, invertido, e quando o ciclo for reiniciado o LED vai apagar ao receber agora 0 da função *digitalWrite( )*. De novo o programa vai pausar por um período determinado pela tensão no pino A0. Como a função *loop( )* é executada continuamente a velocidade com que o LED vai acender e apagar é determinada pela posição do cursor do potenciometro.

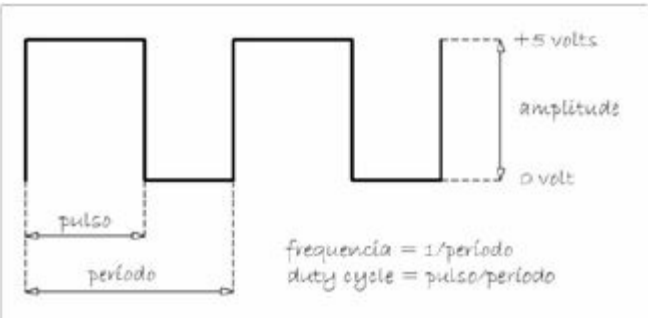
Experimente agora o *sketch* abaixo ainda com o potenciometro conectado no pino analógico A0:

```
void setup() {
  Serial.begin(9600);
  }
void loop() {
  Serial.println(analogRead(A0), DEC);
  delay(1000);
  }
```

Nesse experimento a função *setup( )* inicializa a porta serial em 9600 baud. A função *loop( )* mostra na janela do monitor serial do IDE a posição instantanea do cursor do potenciometro em numeros decimais. Depois de carregar esse *sketch* no Arduino não esqueça de ativar o monitor serial clicando no botão **Serial Monitor** na barra de controle do IDE. Mova o cursor do potenciometro todo para um extremo e depois para o outro extremo observe as mudança de valores nas linhas que surgirem no monitor.

Experimento #4 – Saída Analógica

O Arduino é uma plataforma para desenvolvimento de projetos inteligentes; ele aceita qualquer tipo de sinal elétrico de sensores locais ou remotos (por rádio ou mesmo pela *web*) em qualquer de suas muitas entradas e, após processá-lo de acordo com um programa escrito por você, pode controlar qualquer outro sistema elétrico, como leds, alarmes sonoros, solenoides, motores e até computadores, que podem estar conectados em rede. Como um computador em um *chip* o Arduinocontem uma CPU, muitos registradores, varios tipos de memórias e circuitos de controle externo baseados em dois níveis lógicos – alto e baixo. Já vimos que o Arduino possui um conversor A/D interno que pode ler uma tensão analógica em uma de suas entradas analógicas e transformar essa informação em um sinal de saída digital sequencial de 1's e 0's.
Sabemos que o **período** de um sinal digital é medido entre duas subidas de pulsos adjacentes, tambem chamados de pulsos positivos do sinal. Veja a figura 15 abaixo.



A relação entre a largura do pulso positivo e o período do sinal digital é conhecida por *duty cycle*, ou ciclo de trabalho; essa relação é expressa de forma percentual. Assim uma onda quadrada simétrica, onde a largura do pulso positivo é a metade da do período, tem um ciclo de trabalho de 50%. O Arduino pode gerar sinais digitais de até 500 Hz (períodos de 2 milissegundos) e ter o seu ciclo de trabalho controlado em até 100%. Esse sinal pulsante pode ser então integrado para produzir uma tensão analógica cuja amplitude é a média do sinal digital original. Veja os exemplos na figura

16 abaixo.

Observe que quando o ciclo de trabalho for 50% a tensão média de saída será 2,5 volts; quando esse ciclo de trabalho for 20% a tensão média será de 1 volt. Para se ter uma tensão de saída de 5 volts o ciclo de trabalho deverá ser de 100%, e para 0 volt, 0%. Essa técnica de produzir tensões analógicas a partir de sinais digitais recebe o nome de **Modulação por Largura de Pulso**, ou PWM (*Pulse Width Modulation*). O Arduino trabalha com uma resolução de 8 bits para gerar sinais PWM em seis dos 14 pinos digitais, assim para produzir uma amplitude média de 5 volts o valor da variável que vai modular o sinal resultante será 255.

Para produzir tensões analógicas em qualquer desses seis pinos PWM do Arduino utilizamos a função *analogWrite( )*. Sua sintaxe é a seguinte:

```
analogWrite (pino, valor);
```

Figura 15: Uma onda quadrada típica

O parâmetro **pino** indica qual dos pinos PWM será usado para gerar a tensão analógica, o parâmetro **valor** é qualquer valor inteiro entre 0 e 255.

No *sketch* abaixo vamos usar um potenciometro de 10 K ohms, como no experimento #3, para variar a largura do pulso positivo do sinal digital e consequentemente o brilho de um LED conectado, em série com um resistor de 220 ohms, no pino PWM 10 do Arduino. Observe as fotos na figuras 17 e 18 tiradas da tela de um osciloscopio quando variamos o cursor do potenciometro.

```
void setup ( ) { }       // nenhuma configuracao necessaria
void loop ( ) {
 analogWrite (10,analogRead(A0)/4); // converte e envia para                    // pino 10
 }
```

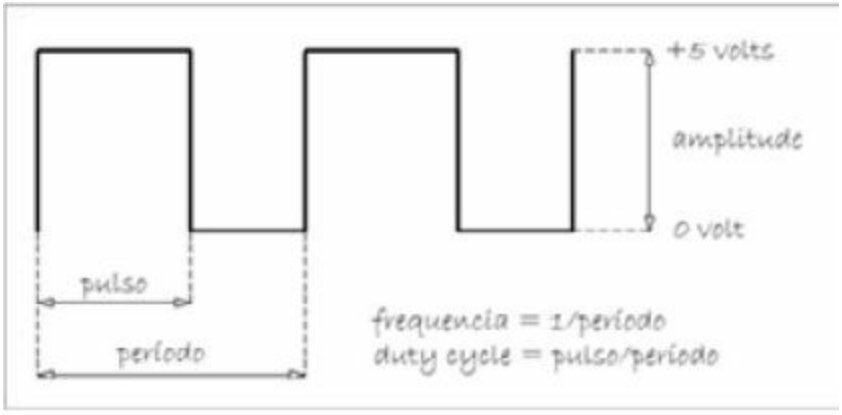


Figura 16: Sinais PWM com duty cycle de 50%, 70% e 20%

Nesse *sketch* observe que na função *setup( )* não configuramos o pino PWM 10 como saída, isso porque os pinos analógicos, diferentemente dos digitais, não precisam ser configurados nem como entrada nem como saída, basta enviar o comando *analogWrite( )* ou *analogRead( )*; mas essa função deve ser sempre incluida, mesmo que vazia, no código fonte. O único comando dentro da função *loop( )* lê o nível de tensão no pino A0, divide esse nível por 4 para compatibilizá-lo com a saída PWM que é de 0 a 255, e envia o resultado para o pino 10. Teste o programa e experimente substituir o potenciometro por um circuito divisor de tensão com um LDR.

Figura 17 *Figura 18*

O *sketch* a seguir aumenta e diminui gradativamente o brilho do LED no pino 10. Repare que dentro da função *loop( )* o comando *for* e a função chamada *ledOn( )* formam um único comando terminando em ponto-e-vírgula.

```
int i=0;      // variavel que controla o brilho do   led
void ledOn();    // declaração da função criada
void loop( ) {
 for (i=0; i <= 255; i++)  ledOn();  // aumenta o brilho do led
 for (i=255; i >= 0; i--)  ledOn();  // diminui o brilho do led
```

```
  }
 void ledOn() {         // função que acende o led
  analogWrite (10, i);
  delay  (10);          // pausa de 20 ms.
  }
```

Você vai encontar mais uma versão desse controle do brilho gradativo de um LED em um pino PWM no IDE do Arduino. Abra o IDE e clique em **File > Examples > Basics > Fade**. O *sketch* abaixo será carregado, teste-o.

```
 // Fade - This example code is in the public domain.

 int brightness = 0;       // how bright the LED is
 int fadeAmount = 5;    // how many points to fade the LED by
 void setup()  {       // declare pin 10 to be an output
 /* pinMode(10, OUTPUT);  obs: essa linha pode ser omitida */
  }
 void loop()  {
  analogWrite(10, brightness);      // set the brightness of pin 10
  brightness = brightness + fadeAmount;// change the brightness                  // for next time
through the loop
  if (brightness == 0 || brightness == 255) {
  fadeAmount = - fadeAmount ;   // reverse the direction of the
            // fading at the ends of the fade
  }
  delay(30);    // wait for 30 milliseconds to see the dimming effect
  }
```

Nesse *sketch* o autor usou uma tática interessante para testar a variável *brightness* dentro do comando *if*; se essa variável alcança um dos extremos da saída PWM, 0 ou 255, a variável *fadeAmount*, que é somada a (ou subtraida de) *brightness*, passa de 5 para -5 e vice-versa. A função *setup( )* pode ser deixada vazia porque não é necessário configurar um pino analógico como saída ou como entrada.

Experimento #5 – Controle de Motores

Motores elétricos são sistemas eletromecanicos que convertem energia elétrica em energia cinética, a energia do movimento. Diversos tipos de motores elétricos, tanto de corrente contínua quanto de corrente alternada, estão praticamente em todos os aparelhos modernos a nossa volta. O leitor pode aproveitar pequenos motores CC retirados de acionadores de CD/DVD com defeito e de impressoras desativadas para fazer experiencias com o Arduino. Vejamos alguns exemplos.

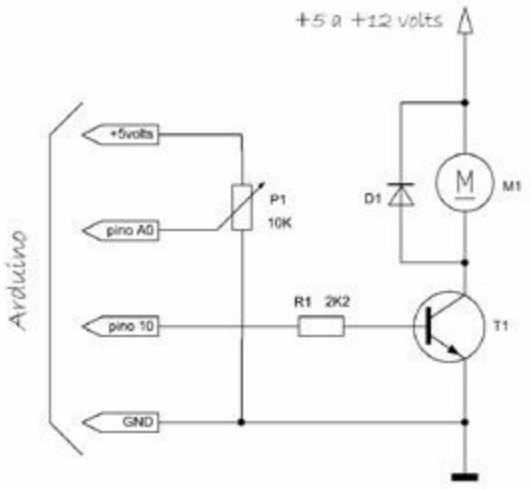
5.1 - Controlando um motor CC

A máxima corrente fornecida por qualquer um dos pinos do Arduino é de 40 miliamperes, o suficiente para controlar diretamente um pequeno motor CC de baixa corrente entre o pino 10 e o terra com quaisquer dos sketches do experimento #4. Se precisar controlar um motor CC de maior corrente e com tensões maiores que 5 volts, como aquele retirado de uma velha impressora, incorpore à saída do Arduino um driver com um transistor de baixa ou média potencia como o da figura 19.

```
void setup() {
  pinMode(10, OUTPUT);    //base transistor no pino 10
  }
void loop() {
  analogWrite(10, analogRead(A0)/4);
  }
```

Figura 19: Controle PWM de um pequeno motor CC

Nesse experimento podemos controlar a velocidade de um pequeno motor CC em um só sentido usando a saída PWM do pino 10 do Arduino. O diodo 1N4001 em paralelo com o motor serve para proteger o transistor curto-circuitando tensões reversas que podem ser geradas no próprio motor. Motores controlados por PWM tem a vantagem de não perderem o torque em baixas velocidades, como acontece com os circuitos que controlam diretamente o nível da tensão aplicada ao motor. Se quisermos controlar alem da velocidade tambem o sentido que um motor CC gira, se no mesmo sentido de movimento dos ponteiros do relógio ou sentido contrário a esse, devemos empregar um circuito conhecido por ponte-H onde são usados quatro transistores, dois PNP e dois NPN.



5.2 - Servo-motores

Tambem com pulsos PWM podemos controlar facilmente um outro tipo de motor, os servo-motores. Servomotores ou simplesmente servos são sistemas eletro-mecanicos compactos constituidos por um pequeno motor CC, um conjunto de engrenagens e um circuito de controle eletronico com realimentação, que são usados principlamente em aeromodelos e outros sistemas controlados remotamente por radio-frequencia. Mas as aplicações dos servos vão muito além, desde brinquedos e controles de varreduras de áreas externas com câmeras de vídeo a sistemas de rastreamento de satélites e robôs. Diferentemente dos motores comuns que têm um eixo central que gira continuamente, os servos normalmente não completam uma rotação completa, mas em passos de 1^0 desde 0^0 até 180^0. Assim, sob controle eletronico externo, é possível posicionar o eixo de um servo em qualquer angulo entre esses valores. Dentro da caixa de um servo, acoplado a uma de suas engrenagens, existe um potenciometro que, funcionando como sensor de posição, realimenta o circuito de controle do motor. Na figura 20 abaixo vemos um modelo de servo que usaremos em nossas experiências.

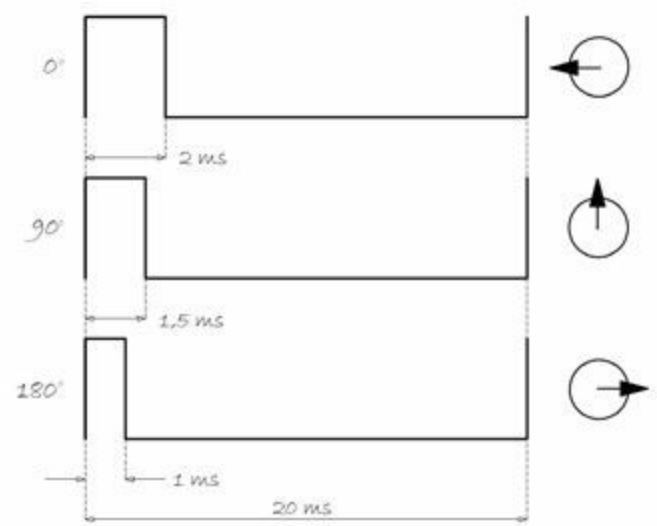
Normalmente os servos têm 3 fios com cores diferentes: o fio vermelho é o de alimentação e deve ser ligado a uma fonte de 5 volts; o fio preto ou marrom é o comum e deve ser ligado ao terra da fonte; e o amarelo ou laranja é o fio de controle de rotação que deve ser conectado à saída do sistema que vai posicionar o eixo do servo atraves de um sinal PWM.

Figura 20: Servo-motor

São duas as principais especificações para todo servo: seu **torque** dado em kg/cm, que em essencia representa a força do servo; e sua **velocidade de rotação**, dada em segundos/graus. O pequeno servo da foto abaixo, por exemplo, tem um torque de 1,5kg/cm e velocidade máxima de 0,3 seg/60 graus, ou seja, ele pode movimentar uma carga de 1500 gramas por centímetro com uma rotação de 60^0 em 300 ms, quando alimentado com 4,8 volts. Os servos são usados para acionar pequenos sistemas mecanicos, como garras, braços ou plataformas com rodas, e porisso é comum virem acompanhados de pequenos adaptadores com furos feitos de plástico rígido ou alumínio que podem ser montados no seu eixo rotativo, e alguns parafusos.

Figura 21: Pulsos típicos de controle de servo-motores

O controle do posicionamento do eixo de um servo é feito por pulsos positivos de largura variável entre 1 e 2 milissegundos, e repetidos a cada 20 milissegundos. Pulsos repetidos de 1,5 milissegundos posicionam o servo em sua posição central de 90^0. Veja a figura 21.



Para suas primeiras experiencias, conecte um servomotor ao seu Arduino conforme a figura 22 e carregue o *sketch* abaixo, que coloca o eixo do servo em sua posição central de 90^0. À variável *servoWd* é atribuido o valor da largura do pulso que vai posicionar o eixo do servo. Valores para essa variável abaixo de 100^0 posicionam o servo em 0^0, e valores acima de 200^0 o posicionam em 180^0. Experimente outros valores entre esses limites. Devido às diferenças mecanicas entre os diversos tipos e fabricantes de servomotores, provavelmente você terá que ajustar esses valores para os angulos corretos.

```
int controlPin = 10;           // controle pino 10 fio amarelo
int servoWd =1500;         // largura do pulso 1,5 ms: 90 graus
void setup( ) {
  pinMode(controlPin, OUTPUT);
  }
void loop( ) {
  digitalWrite(controlPin, 1);
  delayMicroseconds(servoWd);  // posiciona servo 90 graus
  digitalWrite(controlPin, 0);          // para servo
  delay(20);                  // pausa para envio de novo pulso.
  }
```

Existe uma biblioteca específica para controle de servomotores com o Arduino, a biblioteca **Servo.h**; o *sketch* acima reescrito utilizando essa biblioteca fica bastante enxuto, confira:

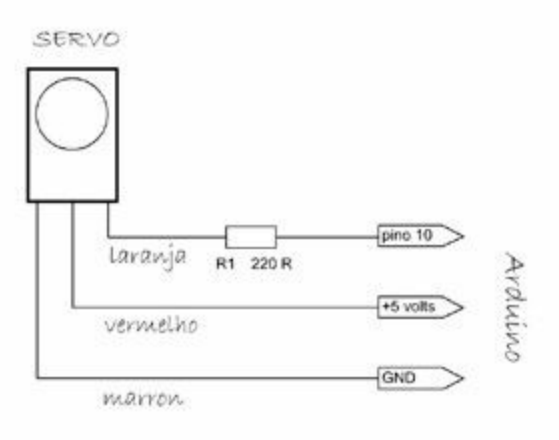


Figura 22: Conexões de um servo-motor

```
 #include <Servo.h>       // importa biblioteca Servo.h
Servo meuServo;      // criação do objeto meuServo
void setup() {
  meuServo.attach(10);     // vincula meuServo ao pino 10
  meuServo.write(90);     // move eixo servo para 90 graus
  }
void loop() {
     /* aqui vão os seus comandos */
  }
```

Note que todo o comando para mover o eixo do servo para 90^0 está dentro da função *setup( )*. Dentro da função *loop( )* vão os comandos complementares da aplicação que você projetou; por exemplo, o acionamento de um aviso sonoro ou uma solenoide.
O *sketch* abaixo rotaciona o eixo do servo continuamente entre seus dois extremos. A função criada *servoPulse( )* é que monta o sinal PWM que aciona o eixo.

```
 int controlPin = 10;            // controle fio laranja no pino 10
 int servoAngle =0;                  // angulo de excursão do servoint     int pulseWd;                    //
largura do pulso do servo
 void setup() {
  pinMode(controlPin,1);
  }
 void loop () {                  // passos de 1 grau
  for (servoAngle = 0; servoAngle <= 2000; servoAngle ++)
  servoPulse(controlPin, servoAngle);   // chama servoPulse
  for (servoAngle = 2000; servoAngle >= 0; servoAngle - - )
  servoPulse(controlPin, servoAngle);
  }
 void servoPulse (controlPin, servoAngle) {
  pulseWd = (servoAngle) + 1000;     // converte angulo para uS
```

```
  digitalWrite(controlPin, 1);        // aciona servo durante
  delayMicroseconds(pulseWd);         // pulseWd ms
  digitalWrite(controlPin, 0);        // e para servo
  delay(20);
  }
```

Utilizando a biblioteca Servo.h o *sketch* acima pode ser muito simplificado, veja a listagem abaixo.

```
int i=0;
unsigned int j=300;
#include <Servo.h>          // inclui biblioteca Servo.h
Servo meuServo;             // cria objeto meuServo
void setup() {
  meuServo.attach(10);      // meuServo no pino 10
  //Serial.begin(9600);
  }
void loop() {
  (i <= 180)? meuServo.write(i): meuServo.write(j);
  i++;
  j- -;
  delay(20);
  //Serial.println(i);//Serial.println(j);
  }
```

5.3 - Motores de passo

Tambem podemos fazer experiencias com motores de passo como nosso Arduino. Motores de passo, ou *stepper motors* em inglês, são um tipo especial de motor de corrente contínua que não giram continuamente, mas em incrementos fixos ou passos (*stepping*) que são controlados externamente por pulsos digitais. Para cada pulso recebido o motor de passo gira um certo grau. Um motor desse tipo que precise de 100 passos para fazer uma revolução completa, por exemplo, tem um passo de $3{,}6^0$ ($360^0/100$). Motores de passo são usados principalmente em sistemas eletro-mecanicos que requeiram perfeito posicionamento linear, aliado a partidas e paradas rápidas tanto no modo direto quanto no modo reverso; como impressoras, escaneadores, traçadores gráficos (*plotters*), acionadores de disco rígido e de CD/DVD e tambem robôs.

Esses motores têm normalmente dois enrolamentos independentes que podem ter ou não tomada central (*center tap*). Aqueles com tomada central nos enrolamentos são motores do tipo **unipolar** e podem ter 5, 6 ou 8 fios para interligação; os sem tomada central são do tipo **bipolar** e têm somente 4 fios de interligação. Os motores bipolares podem ter seus enrolamentos totalmente separados em quatro bobinas independentes e assim as tomadas devem ser formadas externamente interligando os extremos dessas bobinas. Na figura 23 vemos como são dois dos tipos de enrolamentos de motores de passo.

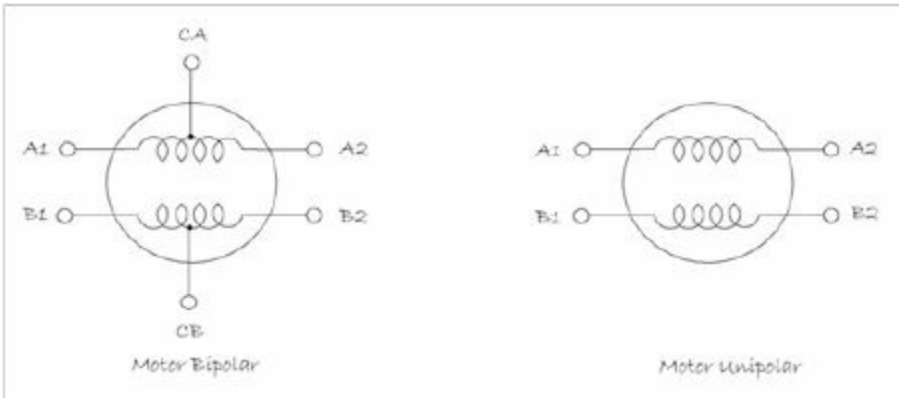


Figura 23: Enrolamentos de um motor de passo

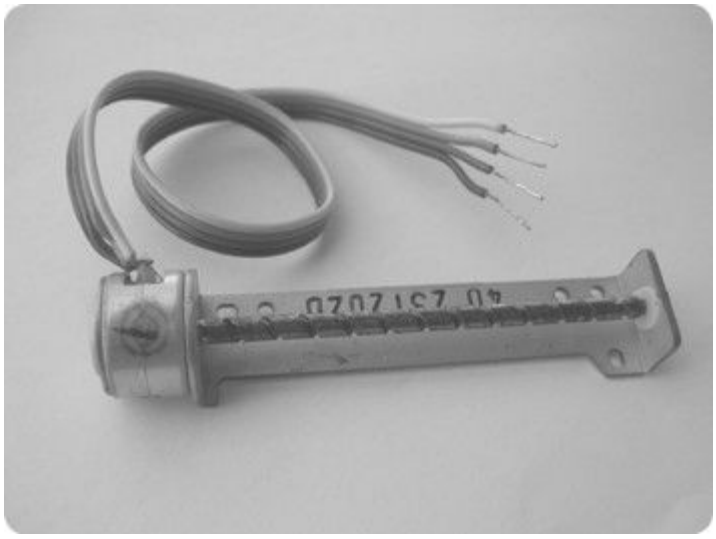


Figura 24: Motor de acionador de disquete de 3,5”

Normalmente as tomadas de cada enrolamento no tipo bipolar são interligadas para formar um motor de 5 fios. Tambem motores desse tipo podem funcionar como unipolar se as tomadas centrais forem isoladas. Você pode aproveitar esses motores de velhas impressoras, acionadores de CD/DVD com defeito ou dos antigos acionadores de disco flexível (*floppy disk*) de antigos computadores. O motor da figura 24, que aproveitaremos em nossas experiências, foi retirado de um desses acionadores de disco de 3,5” e movimentava linearmente a cabeça de leitura e gravação magnética. No lugar do cabo flexível original soldamos um pequeno cabo com 4 fios paralelos, mais conhecido como *flat cable*.

Esse é um motor de passo do tipo bipolar, com 4 fios, modelo PL15S-B20 da empresa asiática ***Minebea***; sua alimentação é de 5 volts e seu passo é de 18^0, ou seja, ele precisa de uma combinação sequencial de 20 pulsos para dar um giro completo.

Sequencia	A1		A2	
	B1	B2		
1	-	+	+	-
2	-	+	-	+
3	+	-	-	+
4	+	-	+	-

Antes de começar nossas experiências precisamos identificar os dois enrolamentos independentes do motor, o que pode ser feito facilmente verificando a continuidade em cada par de fios com qualquer multímetro numa escala baixa de resistência. Nesse pequeno motor de *floppy disk* cada enrolamento tem cerca de 10 ohms. Uma vez identificado cada enrolamento aplique uma tensão de 5 volts sobre um deles e observe se o eixo do motor dá um pequeno giro e para, se não, inverta a polaridade da tensão aplicada. Depois aplique a mesma tensão sobre o outro enrolamento everifique de novo se ocorreu um pequeno giro, inverta a tensão se necessário. Todo motor de passo obedece a uma tabela de chaveamento que mostra a sequência de ativação dos dois enrolamentos para que o motor gire em um sentido ou em outro. A tabela do motor que estamos usando é a seguinte:

Para girar completamente o eixo desse motor precisamos enviar uma sequência de 20 pulsos de curta duração em cada enrolamento.

5.4 - Vibra-motor

Existe mais um tipo de motor que podemos fazer experiências com o Arduino, é o *vibra-motor* ou motor vibratório. É um pequeno motor CC que tem no seu eixo presa uma pequena massa metálica em forma de meia-lua que o desequilibra quando gira, fazendo-o vibrar. São esses motores que fazem vibrar o telefone celular quando estão no modo *vibra-call*. Veja na figura 25 dois desses motores; o menor foi retirado de um telefone celular e o maior de controle com *joystick* do *videogame playstation 2*.

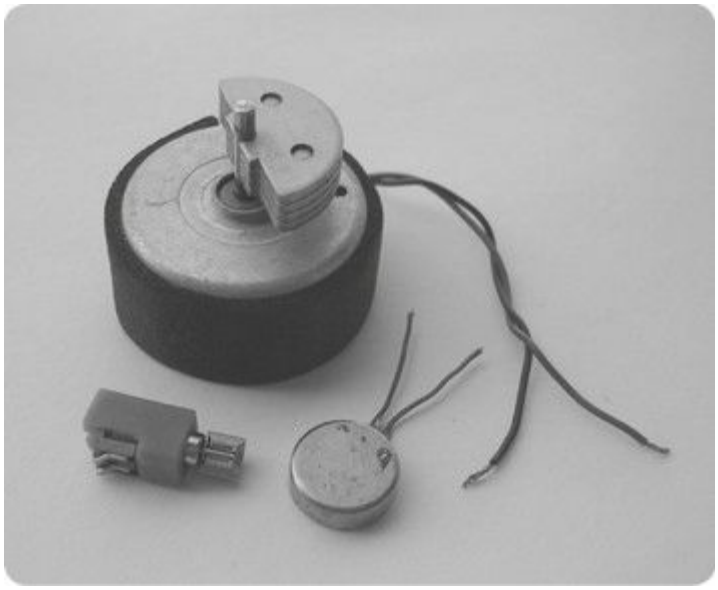


Figura 25: Exemplos de vibra-motores

Os *vibra-motors* de telefones celulares são normalmente alimentados com somente 1,5 volts e consomem uma corrente direta de cerca de 15 mA; aqueles retirados de controles de videogames podem ser alimentados com 5 volts e consomem cerca de 40 mA, e porisso podem ser conectados ao Arduino com o circuito da figura 19 e testados com os *sketches* dos experimentos com motores CC.

Experimento #6 – LEDs e Mostradores

Experimentos com LEDs e mostradores de 7-segmentos e LCD são sempre interessantes e bastante fáceis de implementar.

6.1 - LEDs

Nesse experimento vamos usar um LED RGB e misturar suas cores aleatoriamente. Os LEDs RGB são fisicamente parecidos com os LEDs comuns, porem com quatro terminais; internamente são formados por tres LEDs de tres cores diferentes – um vermelho (*Red*), um verde (*Green*) e um azul (*Blue*) – montados em um mesmo encapsulamento de 5 mm de diamentro. Os catodos de cada um desses LEDs são interligados em um terminal comum, o mais comprido dos quatro; os anodos formam os outros tres terminais. As características elétricas de cada LED interno não são muito diferentes daquelas dos seus correspondentes de dois terminais. Para uma corrente direta de 20 miliamperes o LED vermelho atinge sua luminosidade máxima com 2 volts, os LEDs verde e azul com 3,2 volts. Os quatro terminais são dispostos em linha reta, o do LED vermelho fica isolado ao lado do terminal do catodo comum, o mais comprido; depois vem o do LED verde e o do azul.

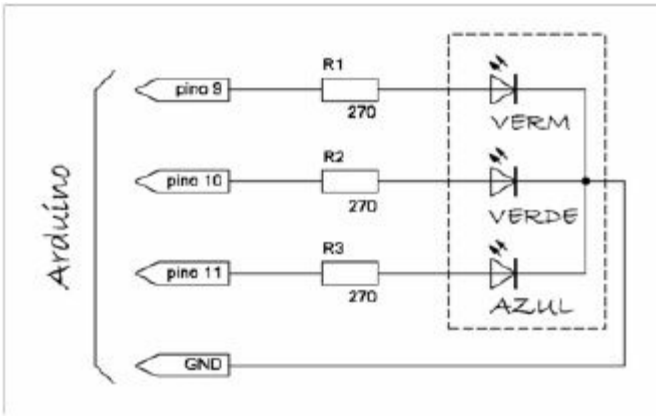


Figura 26:

Para esse experimento conecte um LED RGB e tres resistores de 390 ohms conforme a figura 26 e carregue o *sketch* abaixo no Arduino. Você pode usar tres LEDs comuns de cores diferentes de dois terminais no lugar do LED RGB, mas não esqueça de colocar em série com cada LED um resistor de 390 ohms. Para maior difusão da luz emitida lixe o encapsulamento do LED com uma lixa fina de papel.

```
int i=255;        // variavel entre 0 a 255
int j=1000;       // intervalo entre mudanca cores
void setup()  {  }      // nenhuma configuração
void loop()  {
  analogWrite(9,random(i));  // led vermelho no pino 9
  analogWrite(10,random(i));  // led verde no pino 10
  analogWrite(11,random(i));  // led azul no pino 11
  delay(j);        // pausa de j milisegundos.
  }
```

Nesse *skretch* a função *random( )* gera números aleatórios entre zero e o valor atribuido à variável *i*; a função *analogWrite( )* dentro de *loop( )* gera pulsos PWM baseados no retorno de *random( )* em cada um dos pinos onde estão conectados os LEDs; a função *delay( )* pausa o programa por *j* milisegundos.

Monte o LED RGB, ou os tres LEDs discretos de cores diferentes, no interior de uma bola de ping-pong ou outro objeto translúcido como uma peça média de cristal bruto ou polido, e observe o efeito luminoso multicolorido resultante desse experimento. Experimente mudar o valor das variáveis *i* e *j*.

Nesse outro experimento solde fios a tres potenciometros de 10 K ohms conforme a figura 27 e interligue aos pinos analógicos A0, A1 e A2 do Arduino, carregue o *sketch* abaixo e misture manualmente as cores dos LEDs.

```
void setup() { }    // nenhuma configuração
```

```
void loop( ) {
  analogWrite(9, (analogRead(A0)/4));//led verm pino 9  pot A0
  analogWrite(10,(analogRead(A1)/4));//led verde pino 10 pot  A1
  analogWrite(11,(analogRead(A2)/4));//led azul pino 11 pot A2.
  }
```

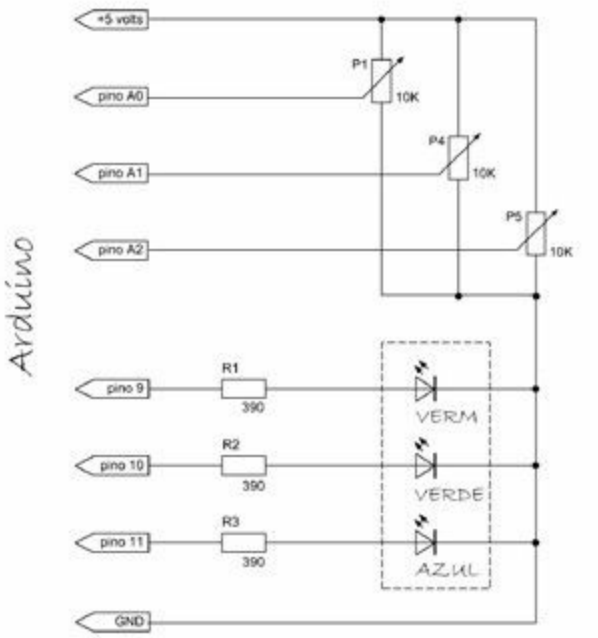
Esse *sketch* lê as posições dos cursores dos potenciometros de 10 K ohms nos pinos analógicos A0, A1 e A2 e as divide por 4, para ajustar a faixa de leitura de 0 a 1023 para a de escrita de 0 a 255, e acende os LEDs nos pinos digitais PWM 9, 10 e 11.

Nesse outro experimento interessante vamos usar um LED bicolor, um resistor de 270 ohms e um potenciometro de 10 K ohms conectados ao Arduino conforme o circuito da figura 28. Esse LED bicolor é formado internamente por um LED verde e um vermelho em anti-paralelo. Pra acender um dos LEDs um dos dois terminais deve receber um nível lógico alto e o outro baixo; para acender o segundo LED basta inverter esses níveis lógicos. Os terminais do LED bicolor devem ser conectados a dois pinos digitais do Arduino, um deles em série com um resistor limitador de corrente. O potenciometro vai variar o nível entre os valores 0 e 5 volts na entrada analógica A0. Veja abaixo o *sketch* para esse experimento. A variável *pinNivel* e seu complemento *!pinNivel* é que vão acender um dos LEDs, e quando complementados vão acender o outro LED. O tempo em que os dois LEDs acendem alternadamente é função da posição do cursor do potenciometro.

```
int pinLedA=7;        // pinoA do LED no pino digital 7
int pinLedB=2;        // pinoB do LED no pino digital 2
int pinNivel;        // variavel que guarda nivel logico
void setup()  {
  pinMode(pinLedA,OUTPUT);    // pino digital 7 como saida
  pinMode(pinLedB,OUTPUT);   // pino digital 2 como saida
  }
void loop()  {
  digitalWrite(pinLedA, pinNivel);  // estado logico no pino 7
  digitalWrite(pinLedB, !pinNivel);  // oposto do pino 7 no pino 2
  delay(analogRead(A0));    // pausa dependente do pot                // no pino A0
  pinNivel=!pinNivel;      // inverte estados logicos
```

Figura 27

Agora um desafio para o leitor: monte o circuito da figura 29 com quatro LEDs de cores diferentes e escreva um *sketch* para fazê-los piscar um a um sequencialmente à direita e à esquerda; e depois de forma acumulativa, ou seja, os LEDs vão acendendo da esquerda para a direita e apagando em sentido inverso. Experimente tambem acendê-los aleatoriamente usando a função *random( )*.



6.2 - Mostradores de 7 segmentos

Interligar um mostrador de 7 segmentos do tipo catodo comum ao Arduino é muito simples, bastam 8 resistores de 330 ohms e um só transistor NPN de uso geral, do tipo BC547. Veja o circuito na figura 30. Cada segmento do mostrador é um LED que vai ligado a sete pinos digitais do Arduino, pinos 5 a 11; o catodo comum dos segmentos vai ligado, atraves do transistor BC547, ao pino digital 12. O terminal do ponto decimal (*dp*, no diagrama) fica sem conexão. Se o mostrador for do tipo anodo comum troque o transistor por um PNP, como o BC557, e interligue o seu emissor ao 5 volts. Nesse tipo de mostrador qualquer número é formado por combinações de segmentos (LEDs) que são identificados por letras de '**a**' até '**g**'. Para mostrar um número cada LED do grupo de segmentos correspondentes a esse número deve ser aceso. Para acender o número **8** todos os LEDs deverão ser acesos.

Figura 28

Mas com multiplexação digital podemos acender um só LED de cada vez de ummesmo grupo de segmentos numa velocidade tal que vemos o número desse grupo devido ao fenômeno da persistência ótica.

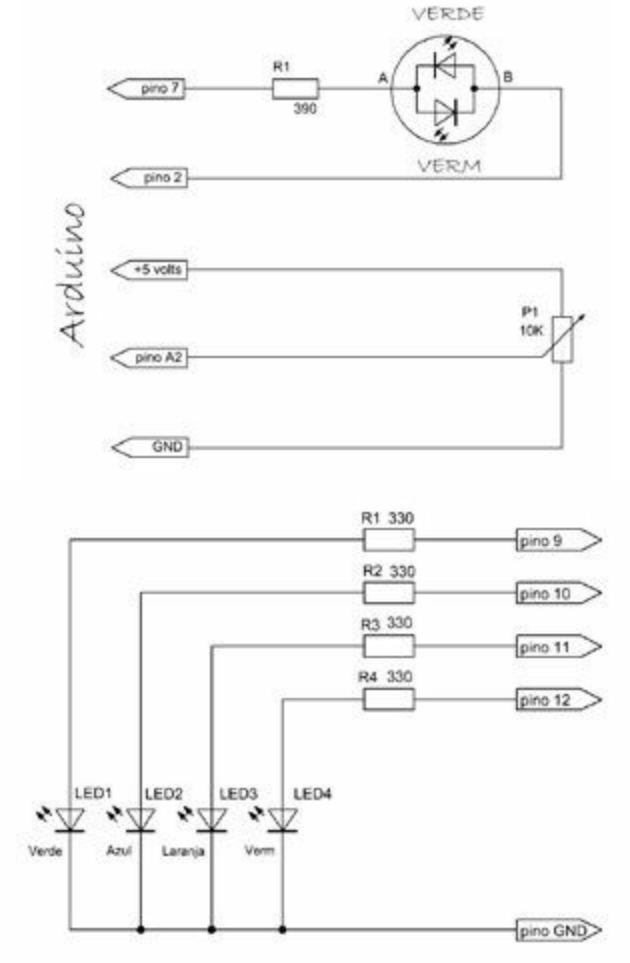


Figura 29: Circuito do desafio ao leitor

Nesse experimento vamos montar um contador com dois mostradores de 7-segmentos, mas antes vamos entender como funciona o código que faz acender um dígito de cada vez em um mostrador. Veja, na listagem abaixo, que estamos usando duas matrizes: uma unidimensional com os sete pinos digitais do Arduino que estão conectados a cada um dos sete LEDs do mostrador; e uma matriz bidimensional que guarda o padrão dos segmentos que forma cada um dos dez digitos. Repare que o padrão de cada digito aparece em linhas separadas somente para facilitar a compreensão do código.

O laço de repetição **for**, dentro de *setup( )*, simplifica a configuração como saída dos sete pinos do Arduino que vão aos segmentos; o pino 12 é configurado como saída para controlar a base do transistor BC547 que vai colocar terra no catodo comum do mostrador. Dentro da função *loop( )* os dois laços de repetição *for* fazem uma varredura na matriz 2D e ativam somente os segmentos correspondentes a cada dígito que deverá ser mostrado; a seguir o transistor mostra o dígito ao conduzir por um segundo. A variável global **i** é aproveitada duas vezes, uma para contar segmentos e uma para contar pinos.

```
int i=0;                    //contador de pinos e segmentos
int j=0;                        //contador de digitos
```

```
int pinoDigital[7]={5,6,7,8,9,10,11}; //matriz 1D com os pinos usados
        // segmentos:  a b c d e f g
int digito[10][7]={       //matriz 2D com os segmentos
     {1,1,1,1,1,1,0},      //digito '0'
                  {0,1,1,0,0,0,0},    //digito '1'
                  {1,1,0,1,1,0,1},    //digito '2'
                  {1,1,1,1,0,0,1},    //digito '3'
              {0,1,1,0,0,1,1},    //digito '4'
   {1,0,1,1,0,1,1},      //digito '5'
              {1,0, 1,1,1,1,1},      //digito '6'
              {1,1,1,0,0,0,0},      //digito '7'
              {1,1,1,1,1,1,1},      //digito '8'
              {1,1,1,1,0,1,1}     //digito '9'
  };
void setup()  {
 for(i=0; i<7; i++)
 pinMode(pinoDigital[i],OUTPUT);  //cada pino como saida
 pinMode(12,OUTPUT);   //pino 12 saida, base do transistor
 }
void loop()  {
 for (j=0; j<10; j++)  {
 for (i=0; i<7; i++)
 digitalWrite(pinoDigital[i],digito[j][i]);//ativa cada pino do Arduino
 digitalWrite(12,HIGH);       //ativa todo o mostrador
 delay(1000);                          //pausa de 1 segundo
 }
 }
```

6.3 - Um contador de dois dígitos

Entendido o código acima, vamos agora montar um contador de dois dígitos que recebe pulsos externos pelo pino digital 2 do Arduino. Tudo o que precisamos é de mais um mostrador de 7 segmentos, outro transistor BC547 e outro resistor de 330 ohms para o segundo transistor. Os sete pinos correspondentes aos sete segmentos desse segundo mostrador são diretamente soldados aos seus correspondentes no primeiro mostrador; assim o segmento **a** do primeiro mostrador deve ser conectado ao segmento **a** do segundo, o segmento **b** do primeiro ao **b** do segundo, e assim até o segmento **g**. Com isso usamos somente sete resistores limitadores de corrente para os dois mostradores. Veja o circuito na figura 31. Veja como ficou a montagem
em uma pequena placa padrão perfurada de 3,5 x 4,0 cm na figura 32. O cabo para interligação do contador ao Arduino é do tipo *flat-cable* e o conector para encaixar no Arduino foi feito com os pinos de uma barra de pinos com passo de 2,54 mm. O *sketch* para esse circuito foi aproveitado do código visto acima. Veja a listagem abaixo.

Figura 30: Mostrador de 7-segmentos catodo comum no Arduino

Esse é o maior *sketch* que até agora montamos e porisso vamos dividí-lo em partes para melhor entendê-lo. Primeiramente, no topo da listagem, temos a declaração de uma matriz bidimensional que usaremos para montar os 10 dígitos com seus 7 segmentos em cada mostrador e uma matriz simples para configurar os pinos do Arduino como saídas para esses segmentos e para os dois transistores que ativam os mostradores. A seguir vêm as declarações de todas as variáveis usadas no programa com seus comentários. Dentro da função *setup( )* configuramos os pinos usados do Arduino; repare que numa única linha configuramos todos os pinos que usaremos como saída chamando cada um desses pinos de uma matriz simples dentro de um comando de iteração **for**; tambem ativamos o resistor de *pull-up* no pino 2 porque queremos contar as bordas de descida dos pulsos gerados por um circuito multivibrador astável com o temporizador LM555 nesse pino.

Dentro da função *loop( )* chamamos tres funções criadas especialmente para esse experimento e uma própria do Arduino, a função *attachInterrupt( )*. Veja a listagem do código abaixo.

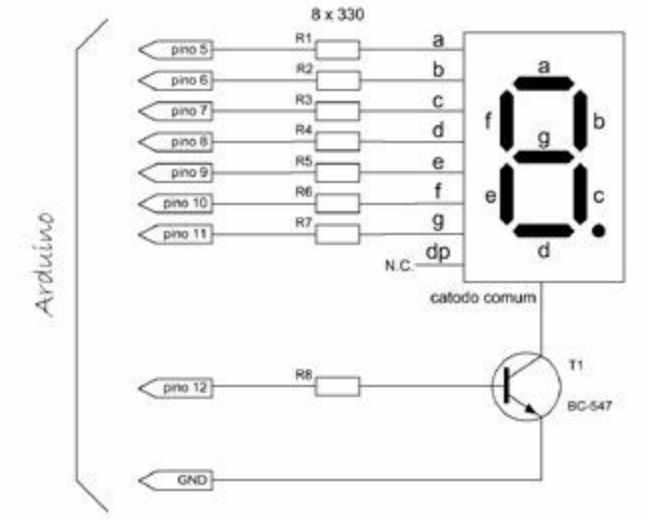
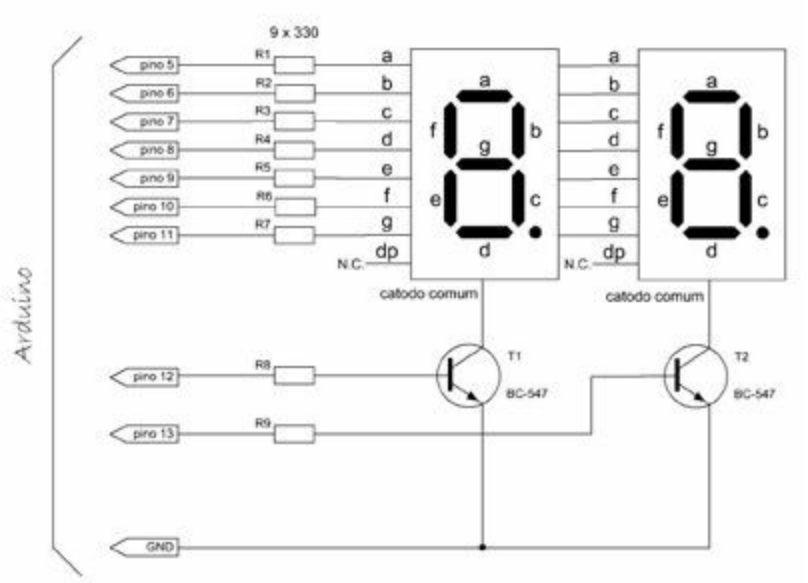


Figura 31: Circuito completo do contador de 2 dígitos

```
//CONTADOR DE DOIS DÍGITOS (attachInterrupt)
//
//Declaração das matrizes e variáveis
// segmentos:       a b c d e f g
int digit[10][7]={ //  ↓ ↓ ↓ ↓ ↓ ↓ ↓    matriz 2D com os segmentos       {1,1,1,1,1,1,0},      //digito
'0'
                    {0,1,1,0,0,0,0},      //digito '1'
              {1,1,0,1,1,0,1},      //digito '2'
                    {1,1,1,1,0,0,1},      //digito '3'
```

```
                {0,1,1,0,0,1,1},      //digito '4'
                {1,0,1,1,0,1,1},      //digito '5'
                {1,0,1,1,1,1,1},      //digito '6'
                {1,1,1,0,0,0,0},      //digito '7'
                {1,1,1,1,1,1,1},      //digito '8'
                {1,1,1,1,0,1,1}       //digito '9'
        };
//
```




```
int outputPin[9]={5,6,7,8,9,10,11,12,13};   //matriz 1D com os pinos usados
int inputPin = 2;                    //pino de entrada dos pulsos
int i=0;                             //contador de pinos e segmentos
int j=0;                             //contador de unidades dos pulsos
int k=0;                             //contador de dezenas dos pulsos
int m=0;                             //variavel para o contador j
int n=0;                             //variavel para o contador k
int number;                          //passa o digito da matriz 2D
int catodePin;                       //controla o catodo do mostrador
int gateState=HIGH;                  //controla a entrada dos pulsos
long posPulse=0;                     //periodo pos da porta de entrada
long negPulse=0;                     //periodo neg da porta de entrada
//
//Configurações de hardware:
void setup()  {
for(i=0; i<9; i++) pinMode(outputPin[i],OUTPUT);   //pinos 5 a 13 saidas
pinMode(inputPin,INPUT);             //pino 2 como entrada
digitalWrite(inputPin,HIGH);         //ativa resistor de pull-up no pino 2
}
//
//
void loop()  {
clock();                             //chama função controle da porta
```

```
displayOn(12,m);                //chama função das unidades
displayOn(13,n);                //chama função das dezenas
attachInterrupt(0,counters,FALLING);  //chama função de contagem
}
//
//Função que gera o pulso de controle de entrada para o contador:
void clock()  {
if(millis() - posPulse > 10)  {
gateState = LOW;                    //cria pulso positivo de 10 ms
posPulse = millis();                //atualiza variavel posPulse
}
if (millis() - negPulse > 1000)  {
gateState = HIGH;                   //cria pulso negativo de 1 seg
negPulse = millis();                //atualiza variavel negPulse
m=j; n=k; j=0; k=0;                 //prepara unidades e dezenas
}
}
//
//
//Função que mostra os digitos:
void displayOn(int catodePin,int number)  {
digitalWrite(catodePin,HIGH);         //habilita transistor no catodo
for(i=0;i<7; i++)
digitalWrite(outputPin[i],digit[number][i]); //ativa pinos/segmentos
delay(10);                            //10 mS para persistência ótica
digitalWrite(catodePin,LOW);          //desabilita mostrador.
}
//Função para contagem dos pulsos:
void counters()  {
j++;                          //incrementa unidades na descida do pulso
(j==10)? j=0, k++ : j;      //se unidades=10 limpa unidades e incrementa k
(k==10)? k=0 : k;           //se dezenas=10 limpa k, senão mantem valor de k
}
/*-----------Fim do programa ----------------*/
```

A primeira função chamada é a função *clock( )* que gera uma onda quadrada de período positivo de 10 ms e negativo de 1000 ms, durante o qual os pulsos de entrada no pino 2 serão contados. A função *displayOn( )* recebe dois parâmetros, o número do pino que controla o transistor de unidades ou de dezenas dos mostradores, e o dígito que dever ser resgatado da matriz bidimensional e exibido nos mostrados de 7-segmentos; ela é chamada duas vezes, uma para exibir as unidades e outra para as dezenas da contagem. A função *attachInterrupt( ),* conhecida como ISR (*Interrupt Service Routine*), chama a função *counters( )* quando ocorre uma interrupção nos pinos digitais 2 ou 3. Sua sintaxe é a seguinte:

attachInterrupt (pino de interrupção, função chamada, modo de disparo);

São tres os parâmetros requeridos por essa função. Primeiro vem um número que indica qual pino digital vai detetar a interrupção: 0 para o pino 2 e 1 para o 3. Depois deve ser passado o nome da função que vai ser chamada quando a interrrupção ocorrer. Por último o modo que a interrupção deverá ser disparada, se na borda de subida do pulso no pino de interrupção, se na borda de descida, se quando o pulso for baixo ou se na mudança de nível desse pulso. Uma descrição mais detalhada dessa função pode ser vista na ***Cartilha para Programação em C para o Arduino***, no final do livro. Em nosso *sketch* estamos fazendo a chamada da função *counters( )* na borda de descida do pulso no pino digital 2.

Figura 32: Montagem do contador de 2 dígitos

Para testarmos esse contador de 2 dígitos vamos precisar de um gerador de pulsos. Uma sugestão simples é montar um multivibrador astável com o temporizador LM555, como o circuito da figura 33, numa matriz de contatos ou numa pequena placa de circuito impresso perfurada padrão, e alimentá-lo pelo próprio Arduino. A frequencia em *hertz* do sinal gerado pelo LM555 é dada pela formula:

*F = 1,4 / (R1 +2*R2)*C1*

Vamos testar agora nosso contador de dois dígitos usando uma nova função da linguagem do Arduino, a função **PulseIn( )**. Primeiramente vamos implementá-lo no Terminal Serial do Arduino. Veja a listagem abaixo.

```
//
int inputPin=2;
float pulseWidth;      // periodo total do pulso de entrada
void setup()  {
  pinMode(inputPin,INPUT);      //configura o pino 2 como entrada
  Serial.begin(9600);          //inicia porta serial em 9600
  }
```

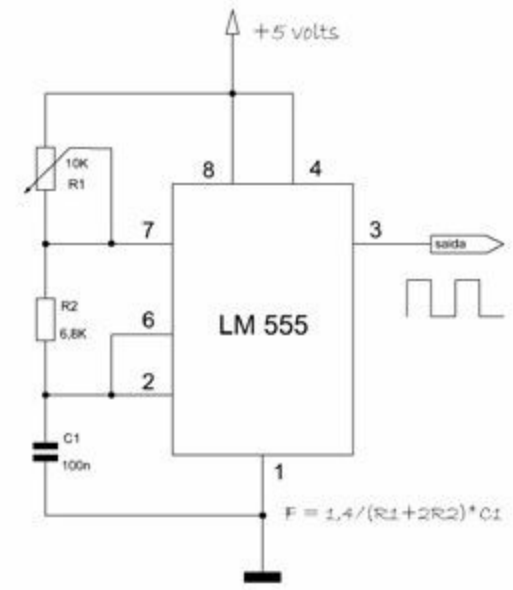
```
void loop()  {
  pulseWidth=(pulseIn(inputPin,LOW) + pulseIn(inputPin,HIGH));
Serial.println(int(1/((pulseWidth)/1000000)));     //converte para freq
delay(1000);           //pausa.
}
```

Figura 33: Multivibrador astável com o LM555

A função *PulseIn( )* mede o período em microssegundos de um pulso alto ou baixo. Sua sintaxe é a seguinte:

pulseIn(pino, valor, espera)

O parâmetro **pino** é o número do pino digital do Arduino que vai receber o pulso a ser medido; o parâmetro **valor** indica se esse pulso vai ser alto (HIGH) ou baixo (LOW); o parâmetro **espera** é opcional e determina quanto tempo em microssegundos a função vai esperar pela ocorrência do pulso; se esse parâmetro não for enviado o tempo de espera da função por *default* é de um segundo. Por exemplo, se o parâmetro **valor** é *LOW,* quando o estado lógico em **pino** for baixo, e após o tempo definido por **espera,** a função *PulseIn( )* inicia a contagem do tempo em microssegundos até o estado em **pino** mudar para alto. Nesse *sketch* a variável *pulseWidth*guarda a soma da largura de um pulso baixo e a de um pulso alto consecutivo, que forma um período completo, em microssegundos, no pino 2 do Arduino. A variável *pulseFreq* guarda o valor da frequencia desse sinal de entrada, que é o inverso do período, guardado em *pulseWidth,* dividido por 1000000.



Lista de Componentes	
CI1	LM555
R1	10K ohms

R2	6,8 K ohms
C1	100 nF

Para finalizar esse tópico vamos montar um contador de dois dígitos com dois mostradores de 7-segmentos usando a função *PulseIn( ),* veja o *sketch*:

```
//CONTADOR DE DOIS DÍGITOS (PulseIn)
//
//segmentos:        a b c d e f g
int digit[10][7]={  {1,1,1,1,1,1,0},      //digito '0'
                    {0,1,1,0,0,0,0},      //digito '1'
                    {1,1,0,1,1,0,1},      //digito '2'
                    {1,1,1,1,0,0,1},      //digito '3'
                    {0,1,1,0,0,1,1},      //digito '4'
                    {1,0,1,1,0,1,1},      //digito '5'
                    {1,0,1,1,1,1,1},      //digito '6'
                    {1,1,1,0,0,0,0},      //digito '7'
    {1,1,1,1,1,1,1},      //digito '8'
                    {1,1,1,1,0,1,1}      //digito '9'
        };
int outputPin[9]={5,6,7,8,9,10,11,12,13};
int inputPin = 2;                       //pino entrada dos pulsos do 555
int i=0;                                //contador pinos e segmentos
int j=0;                                //contador de unidades
int k=0;                                //contador de dezenas
int m=0;                                //contador j
int n=0;                                //contador k
int number;                             //digito da matriz 2D
int catodePin;                          //controla o catodo do mostrador
float pulseWidth;                       //porta para o sinal de entrada
float pulseFreq;                        //frequencia do sinal de entrada
//
void setup()  {
for(i=0; i<9; i++)
pinMode(outputPin[i],OUTPUT);     //pinos digitais 5 a 13 saidas
pinMode(inputPin,INPUT);                //pino 2 como entrada
digitalWrite(inputPin,HIGH);            //ativa resistor de pull-up
}
void loop()  {
pulseWidth=(pulseIn(inputPin,LOW) + pulseIn(inputPin,HIGH));
pulseFreq=(1/(pulseWidth/1000000));    //calcula frequencia do sinal
displayOn(12,int(pulseFreq)%10);          //mostra unidades
```

```
displayOn(13,int(pulseFreq)/10);          //mostra dezenas
}
//
void displayOn(int catodePin,int number)  {
digitalWrite(catodePin,HIGH);             //habilita transistor do mostrador
for(i=0;i<7; i++)
digitalWrite(outputPin[i],digit[number][i]);   //ativa pinos/segmentos
delay(10);                                //10 mS para persistência ótica
digitalWrite(catodePin,LOW);            //desabilita mostrador.
}
/*-----------Fim do programa ----------------*/
```

Como um teste para o que você, leitor, aprendeu, experimente reescrever o *sketch* acima para medir o *duty cycle* (ciclo de trabalho) do sinal que entra no pino 2.

6.4 - LCD

Os mostradores de cristal líquido, ou LCD (*Liquid Crystal Display*), há muito vêm substituindo os mostradores de 7 segmentos como interface visual homem-máquina. Eles foram criados especialmente para ser interligados com microcontroladores. Os mais comuns, do tipo monocromático, estão nos relógios digitais, nas calculadoras, nos fornos de microondas, nas impressoras, nos instrumentos de teste, nos aparelhos biomédicos, nos paineis dos automóveis, enfim em todo lugar onde existe um microcontrolador. Os LCDs do tipo policromático, que vem substituindo os tubos de raios catódicos, ou CRT (*Catode Rays Tube*), estão nos monitores de vídeo, aparelhos de televisão, instrumentos mais complexos de medição e médicos, e os miniaturizados nas câmeras fotográficas e telefones celulares. Em nossos experimentos com o Arduino vamos utilizar um tipo de LCD monocromático que já se tornou padrão em sistemas embarcados – o **HD44780** da japonesa *Hitachi*, de duas linhas por 16 caracteres por linha. Na verdade esse mostrador é uma pequena placa de circuito impresso de cerca de 4,5 x 8 cm onde estão montados um microcontrolador alguns componentes SMD e o mostrador LCD propriamente dito. Veja a figura 34.

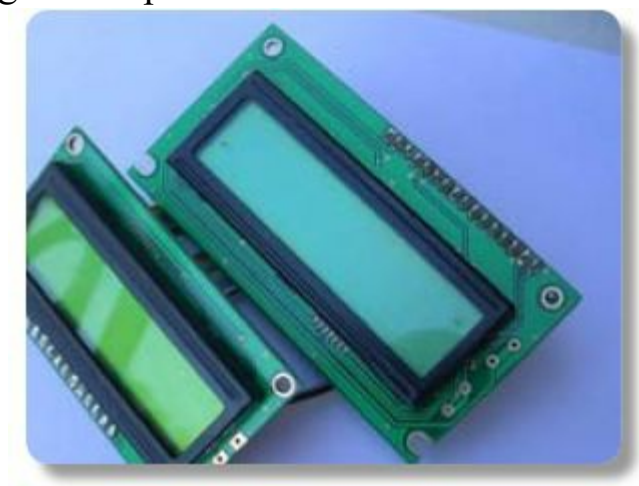


Figura 34: Mostradores LCD

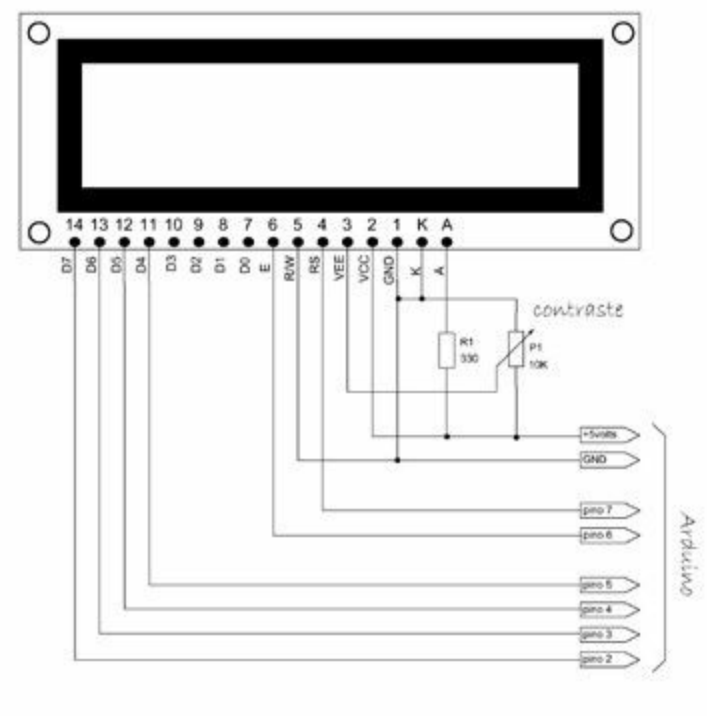
Cada caracter exibido nesse mostrador é uma matriz de 5 x 8 pontos repetida 16 vezes por linha. Podem ser formadas todas as letras maiúsculas e minúsculas do alfabeto, pontuações e símbolos matemáticos. A conexão a um microcontrolador externo é feito atraves de 14 pinos numerados da direita para a esquerda, quando os pinos ficam abaixo do mostrador; e da esquerda para a direita quando os pinos ficam acima do mostrador. Desses 14 pinos oito corresponde ao *byte* de dados que é enviado pelo controlador externo com o caracter que deverá ser exibido; os outros seis são alimentação e controle de operação do módulo. Existe tambem dois outros pinos que vêm sempre à direita da linha de pinos e são marcados com as letras **K** e **A**, que correspondem ao catodo e ao anodo de um LED, que se conectados à fonte de alimentação atraves de um resistor fixo de 330 ohms ou um potenciometro (*trimmer*) ajusta a iluminação de fundo do mostrador. Esses componentes são normalmente omitidos.

Em sua configuração mais simples, e mais utilizada, esse mostrador LCD pode funcionar com 4 *bits* de dados recebidos de cada vez e não 1 *byte* inteiro de uma vez, são quatro pinos a menos tomados do microcontrolador externo, ou do Arduino. Veja na figura 35 o circuito que utilizaremos em nossos experimentos. O potenciometro de 10 K ohms que controla o contraste dos caracteres exibidos no mostrador atraves do pino 3 (VEE) tambem pode ser omitido. Os pinos 4, 5 e 6 são os pinos de controle do mostrador LCD. O nível lógico no pino *RS* (*Register Select*) informa ao módulo se a informação nos pinos D0 a D7 (ou D4 a D7) são caractertes que devem ser exibidos ou se são comandos para limpar o mostrador, posicionar ouapagar o cursor. O pino *R/W* (*Read/Write*) coloca o mostrador no modo leitura de seu registrador de *status* ou no modo escrita de caracteres ou comandos; em nosso circuito esse pino é aterrado para somente envio de caracteres e comandos para o mostrador. O pino *E* (*Enable*) habilita a recepção ou a transmissão de dados entre o mostrador e o controlador externo.

Figura 35: Circuito do mostrador LCD

O *sketch* abaixo envia uma mensagem para cada linha de um mostrador LCD conectado conforme o circuito acima. Aqui a biblioteca **LiquidCrystal.h** que é incluido no código vem com o pacote de arquivos que foi instalado junto com o IDE do Arduino. Essa biblioteca tem 20 funções para controlar qualquer mostrador LCD compatível com o Hitachi HD44780. Na segunda linha do código a função *LiquidCrystal( )* cria o objeto **lcd** e configura os pinos do Arduino para controlar o mostrador LCD. Sua sintaxe para 4 bits de dados e o pino *R/W* aterrado tem o seguinte formato:

LiquidCrystal (RS, EN, D4, D5, D6, D7);



Os parâmetros *RS* e *EN* correspondem aos pinos do Arduino que vão conectados aos pinos RS e Enable do mostrador LCD; os outros parâmetros são os pinos do Arduino que vão conectados aos pinos 4, 5, 6 e 7 do mostrador.

```
//MOSTRADOR LCD
//
#include <LiquidCrystal.h>    //inclui a biblioteca LiquidCrystal.h
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);  //configura pinos do Arduino
void setup() {
  lcd.begin(16, 2);      //configura tipo de mostrador
  }
void loop() {
  lcd.setCursor(0, 0);     //cursor na posicao 0 da 1a. linha
  lcd.print("primeira linha");   //envia 1a.mensagem
  lcd.setCursor(0, 1);     //cursor na posicao 0 da 2a. linha
  lcd.print("segunda linha");   //envia 2a. mensagem.
  }
```

A função *begin( )* especifica o número de colunas e linhas do mostrador que vai ser utilizado; no nosso caso são 16 colunas (ou caracteres) por 2 linhas. A função *setCursor( )* posiciona o cursor onde o proximo caracter deve ser exibido na tela do mostrador. A função *print( )* envia os dados que devem ser exibidos no mostrador. Observe que os textos devem ser colocados entre aspas.

Existe funções dessa biblioteca para limpar e para apagar o mostrador; exibir um caracter recebido pela porta serial; mostrar ou esconder o cursor ou fazê-lo piscar; deslocar o cursor ou o texto para a direita ou para a esquerda; e até criar novos caracteres. Para uma descrição com exemplos de cada

uma dessas funções consulte a pagina do Arduino **http://arduino.cc/en/Reference/LiquidCrystal?from=Tutorial.LCDLibrary**. Outros exemplos de *sketches* podem ser carregados no IDE em **File > Examples > LiquidCrystal**.

Experimento #7 – Solenoides, Reles e Acopladores

Nesse experimento vamos conhecer outras formas de interfacear nosso Arduino com o mundo exterior atraves de solenóides, relés e acopladores óticos.

7.1 - Solenoide

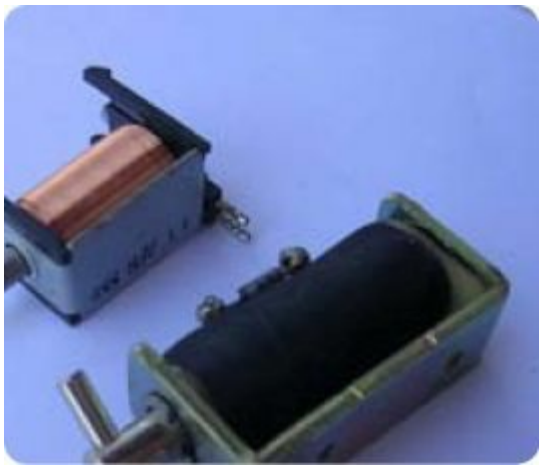


Figura 36: Solenóides

Uma solenoide em sua forma mais simples é composta por uma bobina cilindrica e um eixo de ferro que corre livre em seu interior. Uma das extremidades desse eixo é normalmente acoplada a algum mecanismo que é puxado quando a bobina é energizada. Pequenas solenoides podem ser adquiridas em lojas de material eletronico ou aproveitadas de antigos aparelhos eletronicos, como os videocassetes. Na figura 36 abaixo temos dois tipos de solenoides, o da direita foi retirado de um videocassete Panasonic. A corrente requerida por uma solenoide dessas para atrair seu eixo de ferro é de cerca de 1 ampere, não podendo portanto ser ligada diretamente a um pino do Arduino, que fornece no máximo 40 miliamperes. Para isso precisamos de um circuito de interface como o da figura 37 com um só transistor.

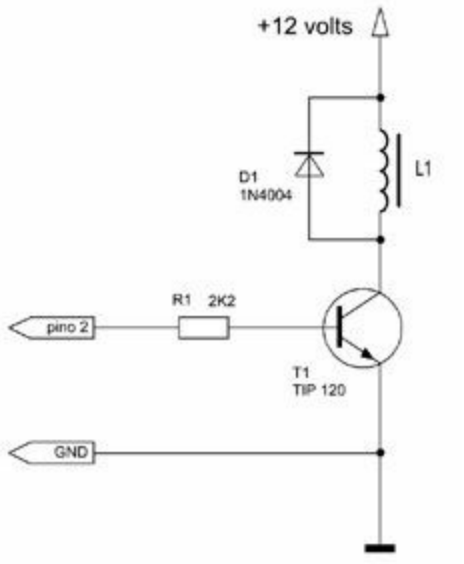
Utilize qualquer transistor NPN de média potencia, como o TIP120, desde que montado em um pequeno dissipador de calor. O diodo polarizado reversamente em paralelo com a bobina é uma proteção contra a tensão reversa que surge nos terminais da solenoide quando esta é desenergizada. O *sketch* para acionar a solenoide deve somente fazer um dos pinos digitais do Arduino alto, e para desligar fazer esse mesmo pino baixo, como no código abaixo.

```
//
void setup() {
  pinMode(2,OUTPUT);    // pino digital 2 como saida
  }
void loop() {
  digitalWrite(2,HIGH);    // +5 volts no pino 2
  delay(500);        // pausa de 0,5 segundo
  digitalWrite(2,LOW);    // terra no pino 2.
  }
```

Figura 37: Interface para solenóides

7.2 - Relés

Quando desejamos isolar eletricamente um circuito de grande potencia do nosso circuito de controle normalmente recorremos aos relés. Relés são como solenoides com uma armadura que aciona um conjunto de contatos mecanicos. O Arduino pode energizar um relé de 12 volts ou mais com o mesmo circuito da figura 37; no lugar da solenoide conectamos a bobina do relé, inclusive com o diodo de proteção. Para pequenos relés de 5 volts que normalmente consomem uma corrente muito baixa o transistor TIP 120 pode ser substituido por um 2N2222, ou equivalente de pequena potencia.



7.3 - Acoplador ótico

Figura 38: Acoplador ótico

Tambem podemos controlar cargas de pequena potencia com o equivalente em estado sólido do relé -- o acoplador ótico. Esse componente eletronico é composto por um LED e um fototransistor montados em um mesmo encapsulamento DIL de 6 pinos. O LED é o transmissor dentro do acoplador ótico e é eletricamente isolado mas oticamente acoplado ao receptor, um transistor, quase sempre NPN, como o 4N25. O transmissor de um acoplador ótico é interligado ao Arduino da mesma forma que um LED comum, com um resistor limitador de corrente em série. Veja figura 38.

A tensão máxima entre emissor e coletor do acoplador 4N25 é de 30 volts e a corrente máxima é de 150 mA, o que é suficiente para acionar diretamente pequenos motores. O pino 3 não tem qualquer conexão externa. No circuito da figura 37 o optoacoplador 4N25 é usado para interfacear uma carga, um motor ou uma solenóide representados por RL, ao Arduino. Observe que o terra do circuito externo é isolado do terra do Arduino. Se o resistor de *pull-up* interno do ATmega168 for ativado no pino 10 do Arduino não será necessário usar um resistor externo no circuito, como no *sketch* abaixo.

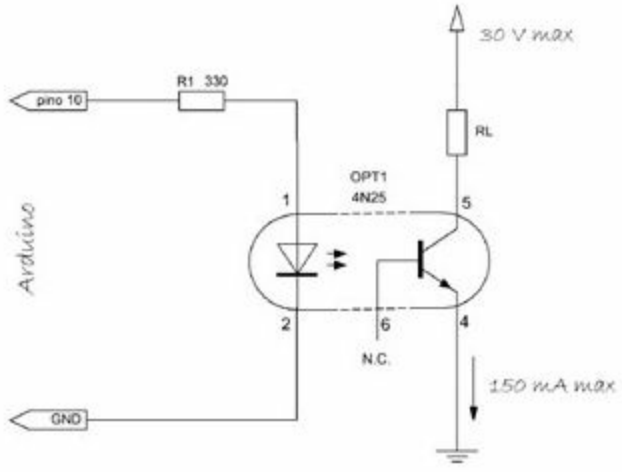
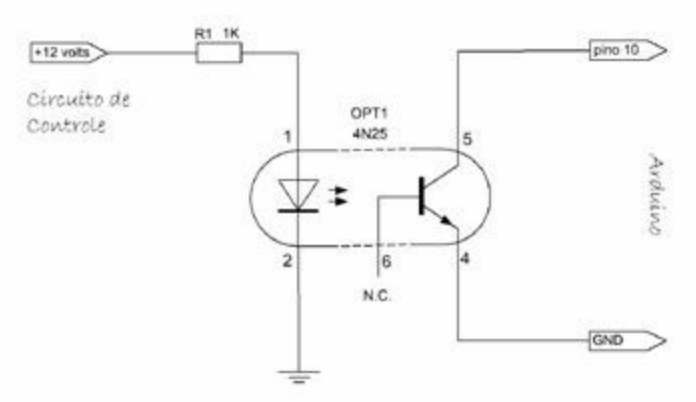


Figura 39: Acoplador ótico

```
//
void setup() {
  pinMode(10,INPUT);  // pino 10 configurado com entrada
  digitalWrite(10,HIGH);  // ativa resistor de pull-up no pino 10
  }
void loop() {
     /* seu codigo aqui */
  }
```

Outra forma de isolar o Arduino de um circuito externo com um acoplador ótico está no circuito da

figura 39. Aqui o transmissor do acoplador vai a um circuito externo. Observe que aqui tambem o terra do circuito externo é isolado do terra do Arduino.



Experimento #8 – Sensores

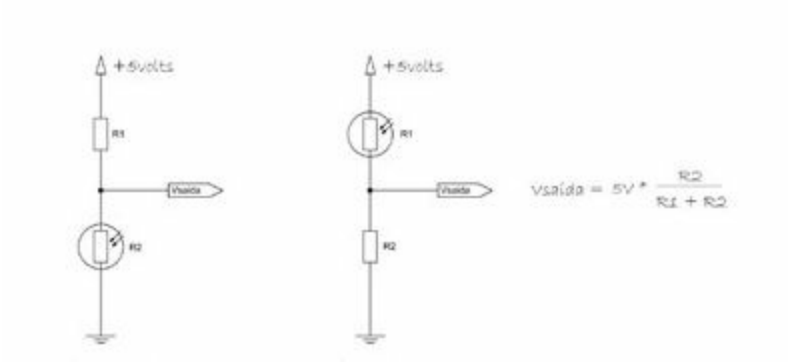
Os sensores eletronicos são extensões de nossos sentidos. Com os sensores podemos detetar quase todos os fenomenos físicos que, uma vez convertidos em sinais elétricos e amplificados, podem ser observados com medidores e registradores gráficos, ou podem ser digitalizados e tratados por sistemas computadorizados. Os sensores eletronicos respondem a estímulos externos como luz, temperatura, vibração, movimento, aceleração, campo magnetico e outros. Nessa seção vamos fazer experimentos com alguns desses sensores e o Arduino.

8.1 - LDRs

Comecemos com o fotorresistor, ou LDR (*Light Dependent Resistor*), um sensor de luz muito fácil de encontrar no comércio de componentes eletronicos que, na ausência da luz tem uma resistência elétrica de cerca de 100Kohms, mas que diminui gradativamente conforme a intensidade luminosa que nele incide. A variação da resistência de um LDR pode ser mudada para variação de tensão com um simples circuito divisor de tensão. Veja a figura 40. No circuito da esquerda a tensão de saída é inversamente proporcional à luz incidente no LDR; no circuito da direita ao contrário, a tensão de saída é uma função direta da intensidade de luz no LDR.

Figura 40: Circuitos divisores de tensão com LDRs

Experimente um e outro circuito com o resistor fixo igual a 10 K ohms; a saída do divisor de tensão deverá ser conectado ao pino analógico A0 (pino 14) do Arduino com o *sketch* abaixo.




```
//
void setup( ) {
  pinMode(13, OUTPUT);     // pino 13 como saida
  }
void loop( ) {
  digitalWrite(13, HIGH);    // acende led
  delay(analogRead(A0));  // pausa conforme leitura LDR
  digitalWrite(13, LOW);     // apaga led
  delay(analogRead(A0));          // pausa conforme leitura LDR
  }
```

Agora carregue esse outro *sketch* e clique no botão do **Serial Monitor** do IDE para observar na janela que surgir a variação no pino analógico A0.

```
//
void setup() {
  Serial.begin(9600);    //inicia porta serial em 9600 b/s
  }
void loop() {
  Serial.println(analogRead(A0));  //envia leitura pino 14
  delay(1000);      //pausa 1 segundo.
  }
```

8.2 - Fototransistores e Fotodiodos

Se você precisar de respostas mais rápidas o LDR pode ser substituido por um fototransistor do tipo **TIL78** ou equivalente. Veja o circuito na figura 39 abaixo. Os fototransistores normalmente são montados em encapsulamentos de LEDs de 3 mm com dois terminais somente; a base do transistor não tem conexão externa e o coletor é o terminal mais curto (aquele que tem o chanfro como nos LEDs).

No *sketch* desse experimento, abaixo, se o teste do comando condicional **if** for verdadeiro, nível alto no pino digital 2, o fototransistor está cortado pois não existe luz incidente, o LED no pino 13 ficará apagado; com luz o fototransistor conduz aterrando o pino 2 e fazendo com que teste seja falso, o que

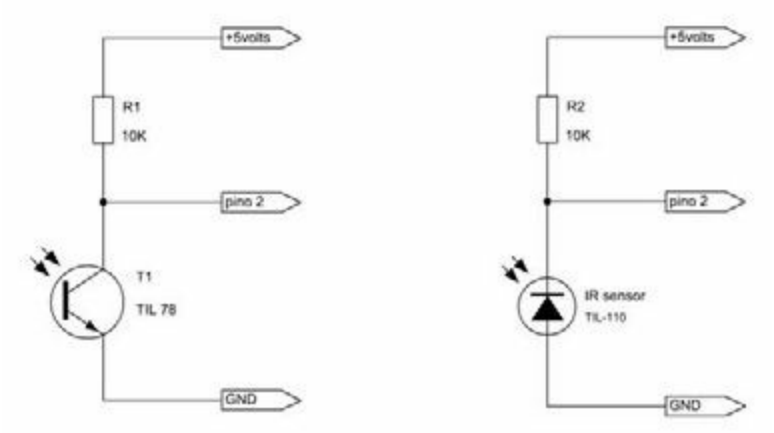
faz o LED acender. Troque o fototransistor por um fotodiodo infra-vermelho do tipo **TIL100** ou equivalente, conforme o circuito da direita da figura 41, e com o mesmo *sketch* abaixo teste qualquer controle remoto de aparelhos de áudio ou TV.

```
//
void setup(){
  pinMode(13,OUTPUT);
  }
void loop(){
  if(digitalRead(2)) digitalWrite(13,LOW);  //teste do pino 2
  else digitalWrite(13,HIGH);      //LED acende com luz.
  }
```

Esse mesmo sketch pode ainda ser reescrito usando o operador ternário '**?**', veja listagem abaixo.

```
//
void setup() {
  pinMode(13,OUTPUT);
  }
void loop() {
  digitalRead(2)?digitalWrite(13,LOW):digitalWrite(13,HIGH);
  }
```

Figura 41: Divisores de tensão com fototransistor e fotodiodo



8.3 - Transdutores Piezoelétricos

Transdutores são elementos que convertem uma forma de energia em outra. Os sensores são uma categoria de transdutores já que convertem fenômenos físicos em tensão ou corrente elétrica. Um exemplo de transdutor é o alto-falante, que converte sinais elétricos em vibrações mecanicas, mas

que é capaz tambem de converter ondas mecanicas em sinais elétricos. Outro exemplo é o motor CC, que pode ser usado tambem como gerador de tensão contínua se girarmos seu eixo.

Em nosso novo experimento com o Arduino vamos usar um transdutor piezoelétrico, um componente eletronico muito usado tanto em microfones quanto em geradores de avisos sonoros, como nos alarmes de relógios digitais e outros aparelhos eletronicos residenciais. Piezoeletricidade é uma característica que alguns cristais têm deapresentarem cargas elétricas positivas e negativas em faces opostas quando submetidos a pressões mecanicas. Podemos observar esse fenomeno em um transdutor piezoelétrico tipico, que podemos encontrar nas lojas de material eletronico, conectando em suas duas faces um LED de qualquer cor, como na figura 40. Toda vez que dermos uma leve batida com a ponta de um lápis em uma das faces do elemento piezoelétrico o LED vai dar uma piscada. Um elemento piezoelétrico tambem é capaz de gerar vibrações mecanicas quando submetido a cargas elétricas e assim produzir ondas sonoras.

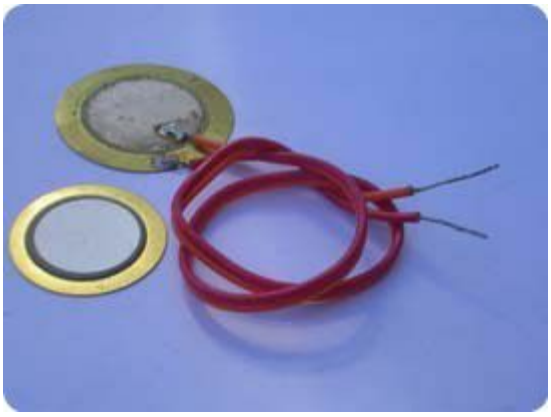


Figura 42: Elementos piezoelétricos

Solde um par de fios em cada face do elemento piezoelétrico e os conecte aos pinos A0 e terra do Arduino. Conecte tambem entre esses pinos, portanto em paralelo com o elemento, um resistor de 1 Mohm e carregue o *sketch* abaixo no seu Arduino.

Dentro da função *loop( )* o pino analógico A0 é continuamente lido e se um valor qualquer é encontrado o LED no pino 13 acende, se nenhum valor é lido o LED apaga. Um pino analógico do Arduino deteta valores a partir de 4,9 milivolts (5 volts/1024), com isso o elemento piezoelétrico fica tão sensível que um sopro a certa distancia ou uma leve pancada em qualquer lugar da mesa onde o sensor repousa faz o LED piscar.

```
//
void setup(){
  pinMode(13,OUTPUT);
  }
void loop(){
  (analogRead(A0)? digitalWrite(13,HIGH):digitalWrite(13,LOW));
  }
```

Se você quiser ter um controle sobre a sensibilidade do transdutor experimente o *sketch* abaixo onde aparece a variável *ajusteSens* que determina o nível de disparo de acendimento do LED. Quanto maior o valor atribuido a essa variável menos sensível se torna a montagem com o transdutor.

```
//
int ajusteSens=10;          //ajuste da sensibilidade do transdutor
void setup()  {
  pinMode(13,OUTPUT);
  }
void loop()  {
  ((analogRead(A0)>=ajusteSens)? digitalWrite(13,1)
  :digitalWrite(13,0));
  }
```

Agora experimente tambem o seguinte *sketch* que envia o valor lido no pino analógico A0 para o seu computador atraves da porta serial. Após carregar esse *sketch* ative o monitor clicando no botão **Serial Monitor** na barra de controles do IDE.

```
//
void setup()  {
  pinMode(13,OUTPUT);          //pino 13 como saida
  Serial.begin(9600);          //inicia a porta serial em 9600 b/s
  }
void loop()  {
  int pinSensor=analogRead(A0);//variavel local guarda valor A0
  if(pinSensor)  {
  digitalWrite(13,HIGH);       //se existir sinal em A0 acende LED
  Serial.println(pinSensor);   //envia leitura para porta serial
  delay(300);                  //pausa entre leituras
  }
  else digitalWrite(13,LOW);       //sem sinal em A0 apaga LED.
  }
```

Essa montagem é tão sensível que até o campo eletrostático de sua mão pode ser sentido pelo transdutor sem mesmo ser tocado. Novamente, se você quiser ter controle sobre a sensibilidade do transdutor mude a expressão de teste dentro do comando condicional if para *if(pinSensor>=10)*. Um experimento interessante com essa montagem talvez seja você colar esse transdutor em algum ponto de uma porta ou janela para detetar batidas de pessoas ou do vento. Tambem embaixo de um tapete é possível saber quando alguem o pisa.

8.4 - Temperatura

Os projetistas de circuitos para o Arduino normalmente usam o sensor integrado **LM335** em suas experiências com medições de temperatura. Aqui vamos fazer experimentos com um diodo retiticador comum, o **1N4148**, como sensor de temperatura aproveitando uma idéia original postada no site da revista de engenharia eletronica EDN **http://www.edn.com/channel/Design_Ideas.php**.

A idéia é aproveitar a característica comum aos diodos de apresentarem sua corrente reversa dependente da temperatura. Como a junção PN de um diodo polarizado reversamente se comporta como um capacitor, então podemos carregá-lo com 5 volts e medir o tempo de sua descarga como função da corrente reversa e que é dependente da temperatura nessa junção. Vamos usar um diodo 1N4148 polarizado reversamente, com o catodo conectado diretamente no pino digital 10 do Arduino e o anodo conectado à terra; em paralelo com o diodo vamos conectar um capacitor ceramico de 1 nF. Veja o circuito na figura 42.

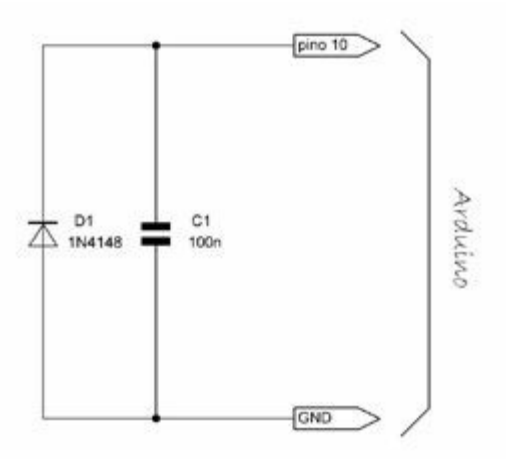


Figura 43: Sensor de temperatura com o 1N4148

O *sketch* para esse experimento é o seguinte:

```
int tempoDesc=0;                  //variavel que mede tempo descarga
unsigned long inicioDesc=0;   //variavel que marca inicio descarga
void setup()  {
  Serial.begin(9600);          //inicia porta serial
  pinMode(13,OUTPUT);           //pino 13 com LED como saida
  }
void loop()  {
  pinMode(10,OUTPUT);  //pino 10 saida: catodo diodo
  digitalWrite(10,HIGH);         //carrega capacitor PN com 5 volts
  delay(100);                    //espera 100 mseg para carregar
  digitalWrite(13,HIGH);           //acende LED no pino 13
  pinMode(10,INPUT);               //inicia leitura do capacitor
  digitalWrite(10,LOW);            //desativa resistor pull-up pino 10
  inicioDesc=millis();              //marca tempo inicio da descarga
  while(digitalRead(10)==1)  { }//espera pela descarga completa
  tempoDesc=millis() - inicioDesc;    //mede tempo de descarga
```

```
Serial.println(tempoDesc);//envia essa medida para porta serial
digitalWrite(13,LOW);              //apaga LED no pino 13.
}
```

Nesse *sketch* inicialmente são declaradas duas variáveis, uma que marca o início e uma que conta o tempo de descarga do capacitor PN. Dentro de *setup( )* a porta serial de comunicação é inicializada em 9600 bits/s e o pino 13 é configurado como saída para o LED. Dentro da função *loop( )* o pino 10 configurado inicialmente como saída para carregar o capacitor PN. Em seguida é estabelecido um tempo de 100 milissegundos para a carga do capacitor e o LED no pino 13 é aceso. O pino 10 é reconfigurado, agora como entrada para a leitura da descarga do capacitor; essa entrada fica em alta impedância com a desativação do resistor interno de pull-up do microcontrolador. A função *millis( )*, vista no capítulo 4, retorna o numero de milissegundos desde que o Arduino foi resetado, e esse valor é usado para marcar o início da descarga do capacitor. O pino 10 fica lendo o capacitor, e enquanto este tiver carga o comando de iteração *while* mantem o programa preso num *loop* infinito. Quando o capacitor se descarrega completamente o *loop* é quebrado e o tempo de descarga é calculado pela variavel *tempoDesc* e enviado pela porta serial ao computador; nesse momento o LED no pino 13 é apagado, e todo o ciclo se repete. Faça experiências aquecendo e resfriando o diodo 1N4148 e observe os resultados no Terminal do seu computador; esses resultados podem ser calibrados ajustando os valores medidos pela variável *tempoDesc* e um termometro confiável.

Resumo do capítulo 5

Nesse capítulo vimos que:

- todas as **variáveis** devem ser declaradas em qualquer parte do programa antes de ser usadas. Quando declaramos uma variável dizemos ao compilador que tipo de dado será armazenada nesa variável.

- são **variáveis globais** aquelas que são declaradas fora do corpo de qualquer função e porisso podem ser usadas e alteradas por qualquer comando do programa; elas são normalmente declaradas logo nas primeiras linhas do código.

- são **variáveis locais** aquelas que são declaradas no corpo de uma função e portanto só podem ser usadas e alteradas por comandos dentro dessa função; essas variáveis devem declaradas antes dos comandos que vão usá-las.

- **diretivas do preprocessador** são comandos acrescentados por um programa, o preprocessador, ao código fonte original e passado ao compilador para gerar o código fonte final.

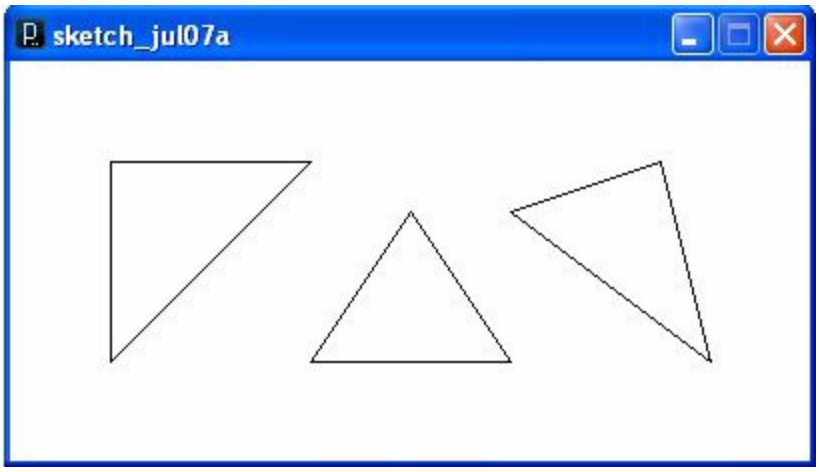
- **ciclo de trabalho** ou *duty cycle* é a relação entre a largura do pulso positivo e o período de um sinal digital, e é expressa de forma percentual.

- **tensões analógicas** de até 5 volts podem ser geradas em qualquer dos seis pinos PWM do Arduino

utilizando a função ***analogWrite( )***.

- A função ***attachInterrupt( )*** é uma ISR (*Interrupt Service Routine*) da linguagem do Arduino que pode monitorar os pinos digitais 2 e 3 e chamar uma outra função se o estado lógico nesses pinos for alterado.

- A função ***PulseIn( )*** serve para medir o período em microssegundos de um pulso alto ou baixo em qualquer pino digital do Arduino.

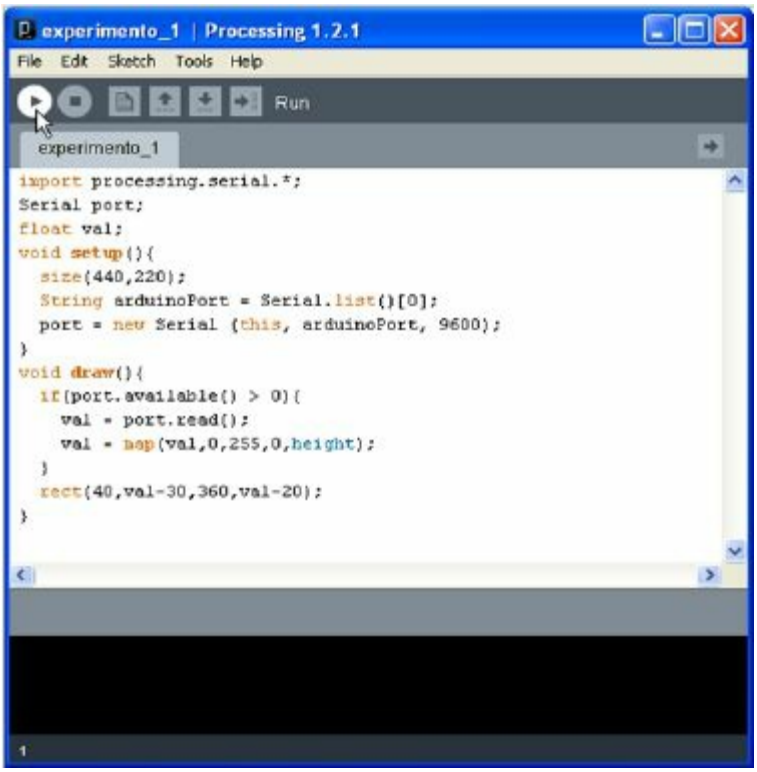


Capítulo 6

A Linguagem *Processing*
[Princípios Básicos]

Introdução

Processing é uma linguagem de programação de computadores que foi criada segundo o conceito de *software* livre. Essa linguagem foi concebida no ano de 2001 por *Casey Reas* e *Benjamin Fry*, na época dois estudantes do **MIT** (*Massachusetts Institute of Technology*), como uma ferramenta para desenvolvimento de projetos gráficos e interativos para artistas e *designers* com pouco conhecimento de programação de computadores. É uma linguagem muito fácil de aprender e usar; com ela podemos criar, por exemplo, aplicações que mostram graficamente na tela do computador o comportamento de um sensor conectado em uma porta analógica do Arduino. A estrutura de sintaxe dessa linguagem é semelhante à do Arduino, inclusive a sua interface gráfica, ou ambiente de desenvolvimento da linguagem Processing, **PDE** (*Processing Development Environment*).



No endereço **http://processing.org/download/** o leitor pode baixar sem nenhum custo a linguagem *Processing* de acordo com o sistema operacional do seu computador, seja ele *Windows*, *Linux* ou *MAC*. O arquivo para *Windows* quando baixado e descompactado gera seis pastas e um arquivo executável, todos numa pasta principal que leva o nome da última versão disponível; por exemplo *processing-1.2.1*. Depois de baixar e descompactar o arquivo mova a pasta criada para **C:\Arquivos de programas**, ou outro local desejado, e crie um atalho do executável *processing.exe* na área de trabalho. Neste capítulo não vamos ver em detalhes a sintaxe dessa linguagem, mas vamos conhecer os principais comandos enquanto formos criando algumas aplicações para muitos dos experimentos que desenvolvemos no capítulo anterior. No endereço **http://processing.org/learning/** o leitor vai encontrar um excelente tutorial sobre a linguagem e dezenas de exemplos de aplicativos e seus respectivos códigos fontes.

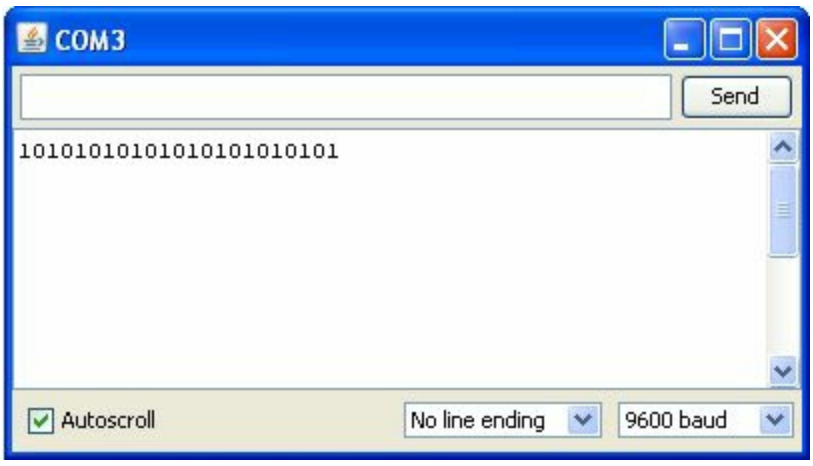
Figura 44: O PDE da linguagem Processing

Como dissemos, o PDE da linguagem *Processing* é muito parecida com o IDE do Arduino, veja a figura 43 abaixo. No topo aparece o nome do *sketch* carregado e o número da versão da linguagem; abaixo vem a barra de menus, depois a de controle, sem o botão que no IDE do Arduino ativa o monitor serial; depois vem a janela de edição com as abas com os nomes dos *sketches* e o botão de abertura de novas abas, bem à direita; por fim, entre duas barras de mensagens na parte inferior do PDE, vem a barra preta de console, onde funciona o **monitor serial** do PDE do *Processing*. Algumas poucas diferenças para o IDE do Arduino podem ser notadas nas opções dos itens da barra de menu.

Para que o leitor tenha uma boa idéia da capacidade gráfica dessa linguagem, abra o PDE do *Processing* e carregue o *sketch* **RGBCube** que está em **File > Examples > 3D > Form**. Agora clique no botão **Run**, o primeiro à esquerda da barra de controle, e experimente interagir com o cubo multicolorido que vai surgir numa pequenajanela, deslizando sobre ele o cursor do seu mouse.

Mas vamos começar nossas experiências com alguma coisa muito mais simples; primeiro vamos carregar no Arduino o primeiro programa que vimos no capítulo 2, o *sketch* **Blink**, aquele que carregamos de **File > Examples > Digital** e que modificamos para fazer o LED no pino 13 ficar aceso por dois segundos e dar duas piscadas curtas de 200 milissegundos. Vamos incluir mais algumas linhas no código para que nosso Arduino se comunique com o um computador pela porta serial. Veja abaixo como ficou o *sketch*.

```
void setup() {
  pinMode(13, OUTPUT);      //pino 13 saida
  Serial.begin(9600);        //inicia comunicação serial
  }
void loop() {
  digitalWrite(13, HIGH);     //LED pino 13 aceso
  Serial.print(1);            //envia caracter '1' porta serial
  delay(2000);                //pausa 2 seg
  digitalWrite(13, LOW);      //LED apagado
  Serial.print(0);            //envia caracater '0' porta serial
  delay(200);                 //pausa 0,2 seg
  digitalWrite(13, HIGH);    //LED aceso
  Serial.print(1);
  delay(200);                 //pausa 0,2 seg
  digitalWrite(13, LOW);    //LED apagado
  Serial.print(0);
  delay(200);                 //pausa 0,2 seg
  }
```



Adicionamos as linhas que enviam para o computador via porta serial os caracteres **1**, quando o LED do pino 13 do Arduino acende, e **0**, quando o LED apaga. Esse *sketch* deve ser carregado no Arduino a partir do seu IDE. Depois de carregado verifique se o LED no pino 13 pisca como descrito acima; ative o **Serial Monitor**, sétimo botão da barra de controle do IDE, e observe que, em sincronismo

com as piscadas do LED, aparece uma sequência de **0**'s e **1**'s no Monitor, como na figura 43, e que são enviados pelo Arduino para porta serial para ser capturados por algum outro programa que esteja sendo executado no seu computador. Se tudo funcionou como descrito acima você pode agora fechar o monitor serial e abrir o PDE da linguagem *Processing* e copiar na sua janela de edição o seguinte código:

```
import processing.serial.*;      //importa biblioteca serial
Serial port;                     //cria o objeto 'port' serial
void setup()  {
 size(200,200);         //cria uma janela de 200x200 pixels
 port=new Serial(this,"COM3" , 9600);// inicia serial COM1 9600
 }
```

Figura 45: O monitor serial do Arduino

```
void draw()  {                    //função equivalente a loop()
 while(port.available()>0) { //testa existencia de dados na porta
 background(255);             //fundo branco na janela criada
 if (port.read()==49) {        //verifica se dado lido é '1' (ascii 49)
  fill (#00FF00);          //se sim escolhe cor verde
  ellipse(100,100,100,100);  //e desenha circulo na janela
  }
 else {                           //se dado não é '1'
  fill (255);                     //seleciona cor branca
 ellipse(100,100,100,100);           //e desenha circulo na janela
  }
 }
 }
```

Antes de executar esse *sketch* perca mais alguns minutos e confira cada linha do código; veja se não esqueceu algum ponto-e-vírgula no fim de cada linha de comando ou algum colchete de fechamento de bloco. Verifique tambem se a porta serial configurada no *sketch* do *Processing* - dentro de *setup( )* em *port=new Serial(this,"COM3" , 9600)* - é a mesma configurada no IDE do Arduino. Depois de executar o *sketch* clicando no primeiro botão da barra de controle do PDE do *Processing* o que você vê? O LED do pino 13 do Arduino surgiu numa janela na telado seu computador! Repare que o LED real e o LED virtual estão em sincronismo, piscam ao mesmo tempo. O seu primeiro programa na linguagem *Processing* está funcionando!

Agora vamos entender cada uma das linhas desse código em *Processing*. A primeira linha depois dos comentários importa a biblioteca de comunicação serial dessa linguagem; a segunda linha cria o objeto **port** que será usado para montar as funções de comunicação serial com o computador. O conceito de objeto, que foge das pretenções desse nosso livro, vem das linguagens orientadas a objeto, como a **C++**.

Como na linguagem do Arduino, temos em *Processing* uma função *setup( )* onde configuramos alguns comportamentos do programa que vamos executar. A função *size( )* cria uma janela de 200x200 *pixels* onde serão exibidos os objetos gráficos que

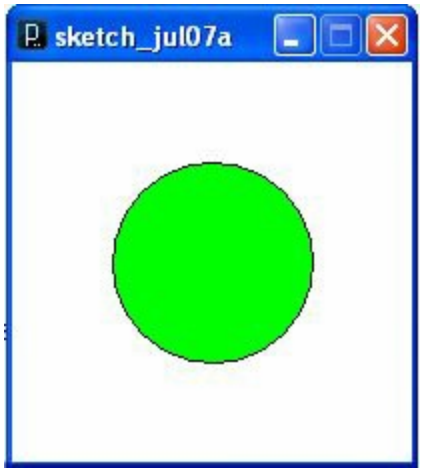


Figura 46: LED virtual com Processing

criarmos no nosso *sketch*. Essa função tem a seguinte sintaxe:

size (largura, altura)

Como os próprios nomes já indicam, o parâmetro **largura** define em *pixels*, ou pontos por polegada, a largura da janela que será criada; o parâmetro **altura** define a altura dessa janela. Opcionalmente existe um terceiro parâmetro para aplicações em gráficos 3D. É recomendável que essa função seja sempre a primeira dentro de *setup( )*. O objeto *port*, criado no topo do código, abre um novo canal de comunicação com o computador na porta serial COM1 em 9600 bits por segundo.

Agora temos a função *draw( )* que é equivalente à função *loop( )* na linguagem do Arduino. O comando de iteração *while* verifica se na porta serial existe um dado novo disponível; se sim a cor branca (ASCII 255) é passada para a função *background( )* que vai mostrar o fundo da janela que será exibida. Depois o comando condicional *if* verifica se o dado enviado pelo Arduino e capturado pela função *read( )* é **1**, código ASCII 49. Se esse dado é mesmo 1 a cor verde, ASCII 00FF00, é passada para a função *fill( )* que vai preencher o círculo criado pela função *ellipse( )*. Essa função desenha uma elípse na janela criada pela função *size( )* e tem a seguinte sintaxe:

ellipse (x, y, largura, altura)

Os parâmetros **x** e **y** definem as coordenadas do centro da elípse em relação à borda esquerda da janela e em relação ao topo da janela, respectivamente; os dois parâmetros seguintes definem a largura e a altura da elípse; se esses parâmetros forem iguais um círculo é desenhado na janela. No *sketch* acima um círculo de 100x100 *pixels* é desenhado com seu centro equidistante 100 *pixels* dos lados superior e esquerdo de uma janela de 200x200 pixels.

Na seção seguinte vamos conhecer as 8 principais funções de desenho da linguagem *Processing*, com as quais podemos montar qualquer forma geométrica complexa.

As 8 principais funções de desenho da linguagem *Processing*

São as seguintes as principais funções de desenho plano da linguagem *Processing*:

1- size(largura,altura)
2- point(x,y)
3- line(x1,y1,x2,y2)
4- triangle(x1,y1,x2,y2,x3,y3)
5- quad(x1,y1,x2,y2,x3,y3,x4,y4)
6- rect(x,y,largura,altura)
7- ellipse(x,y,largura,altura)
8- arc(x,y,largura,altura,inicio,fim)

Vejamos em detalhes cada uma delas.

1 - size(largura,altura)

Essa função não é propriamente uma função de desenho, mas aquela que define na tela do computador as dimensões da janela dentro da qual o desenho será mostrado. São dois os parâmetros requeridos por essa função; o primeiro, **largura**, define a largura da janela; o segundo, **altura**, define a altura dessa janela. No primeiro experimento desse capítulo criamos uma janela quadrangular de 200x200 *pixels* com a seguinte linha de código:

```
size(200,200);      // cria uma janela de 200x200 pixels
```

2 - point(x,y)

Para posicionar um simples *pixel* em algum ponto de uma janela criada com a função *size( )* usamos a função *point( )*, que requer tambem somente dois parâmetros, as coordenadas cartesianas **x** e **y**. A coordenada x é a medida em *pixels* da distancia desde o lado esquerdo da janela até o ponto; a coordenada y é a medida em *pixels* da distancia desde o topo da janela até o ponto. Experimente marcar um ponto preto no meio de uma janela retangular branca de 400x200 pixels executando o seguinte código no PDE do Processing:

```
background(255);  // define como branco o fundo da janela
size(400,200);    // cria uma janela de 400x200 pixels
point(200,100);   // posiciona um ponto no meio da janela.
```

Observe que para marcar um ponto no centro da janela criada os parâmetros da função *point( )* são, em valores absolutos, a metade dos correspondentes parâmetros da função *size( )*. Experimente posicionar o ponto em diferentes lugares alterando os parâmetros da função *point( )*. Como um segmento de reta é uma sucessão de pontos, tente desenhar linhas horizontais, verticais e diagonais na janela marcando pontos sucessivos e distantes 1 *pixel* um do outro. Sugestão: use em seu código fonte o comando de iteração *for* para desenhar essas linhas; se você aumentar a distancia entre os pontos em dois ou tres *pixels* uma linha pontilhada será exibida.

3 - line(x1,y1,x2,y2)

Essa é a função do *Processing* própria para traçar linhas; são requeridos quatro parâmetros: dois para marcar onde a linha deve começar e dois outros para marcar onde a linha deve terminar. O primeiro par de coordenadas, de índice 1, marca o extremo esquerdo da linha; o segundo par, de índice 2, marca o extremo direito da linha.

```
background(255);  // fundo da janela branco
size(400,200);    // tamanho da janela
line(5  0,150,350,50);  // desenha linha diagonal na janela.
```

Essa linha será desenhada a partir da posição 50 *pixels* da borda esquerda da janela e 150 *pixels* a partir do topo da janela; o final da linha estará a 350 *pixels* da borda esquerda da janela e 50 *pixels* do topo dessa janela.

4 - triangle(x1,y1,x2,y2,x3,y3)

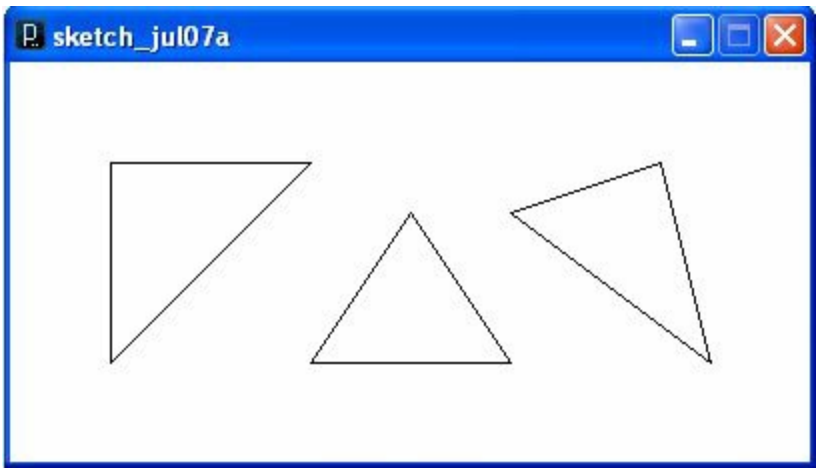


Figura 47: Triangulos desenhados com linguagem Processing

Para desenhar um triângulo utilizamos a função *triangle( )* que requer um par de parâmetros para cada um dos tres vértices do triângulo.

```
background(255);
size(400,200);
triangle(50,50,50,150,150,50);        // triangulo retangulo
triangle(150,150,250,150,200,75);      // triangulo isosceles
triangle(250,75,350,150,325,50);      // triangulo escaleno
```

5 - quad(x1,y1,x2,y2,x3,y3,x4,y4)

A função *quad( )* serve para desenhar um quadrilátero e requer quatro pares de coordenadas, um para cada vértice do quadrilátero.

```
background(255);
size(400,200);
quad(50,150,350,150,300,50,100,50);  // desenha um trapezio.
```

6 - rect(x,y,largura,altura)

Um quadrilátero retângulo, ou simplesmente retângulo, é mais facilmente desenhado com a função *rect( )*, que requer apenas dois pares de parâmetros; o primeiro par são as coordenadas cartesianas x e y do ponto, em relação às bordas esquerda e superior da janela, a partir do qual o retângulo vai ser desenhado; o par seguinte são a largura e a altura do retângulo.

```
background(255);
size(400,200);
rect(50,50,300,100);    // retangulo centralizado na janela.
```

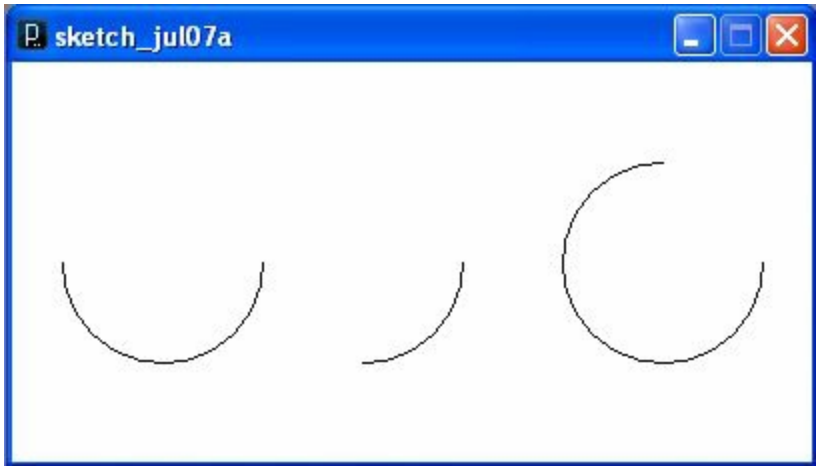
7 - ellipse(x,y,largura,altura)

Para desenhar uma elípse ou um círculo a função *ellipse( )* requer os mesmos tipos de parâmetros da função que desenha o retângulo; primeiro vem as coordenadas x e y, agora, do centro da figura; depois a largura e a altura. Para desenhar um círculo os parâmetros largura e altura devem ser iguais.

```
background(255);
size(400,200);
ellipse(100,100,100,100);    // desenha circulo com altura 100
ellipse(275,100,175,100);    // desenha elipse com altura 100.
```

8 - arc(x,y,largura,altura,inicio,fim)

Arcos de círculos são desenhados com a função *arc( )*, que exige mais dois parâmetros que a função *ellipse( )*. Os dois primeiros parâmetros são as coordenadas x e y do centro do arco; os dois parâmetros seguintes definem a largura e a altura do arco; os dois últimos marcam o ângulo, em radianos, de início e de término do arco. Da Geometria sabemos que ângulos medidos em radianos são múltiplos e submúltiplos do número π (3,1416), e que um círculo completo (360^0) tem 2 π radianos. Veja como alguns arcos podem ser traçados, copiando e executando o código a seguir.




```
background(255);
size(400,200);
arc(75,100,100,100,0,PI);              // arco de 180 graus
arc(175,100,100,100,0,HALF_PI);        // arco de 90 graus
arc(325,100,100,100,0,PI+HALF_PI);     // arco de 270 graus
```

Observe que o desenho do arco começa na origem do primeiro quadrante mas é traçado seguindo o sentido horário. Existe uma função do *Processing* que converte diretamente ângulos em graus para radianos, é a função *radians( )*. Assim o código acima pode ser reescrito da seguinte forma:

Figura 48: Arcos desenhados com a linguagem Processing

```
background(255);
size(400,200);
arc(75,100,100,100,0,radians(180));          // arco de 180 graus
arc(175,100,100,100,0,radians(90));          // arco de 90 graus
arc(325,100,100,100,0,radians(270));         // arco de 270 graus
```

Outras funções importantes de desenho plano

Algumas outras funções, que constituem os atributos de desenho, são bastante úteis quando criamos desenhos e gráficos com a linguagem *Processing*. São elas:

1 – smooth( )
2 – strokeWeight( )
3 – strokeJoin( )
4 – strokeCap( )

1 – smooth( )

Essa função suaviza os contornos das figuras geométricas desenhadas na janela de aplicação criada com *size( )*; linhas com aparência serrilhada, por exemplo, podem ser melhor definidas. Edite e execute esse exemplo e compare as figuras mostradas:

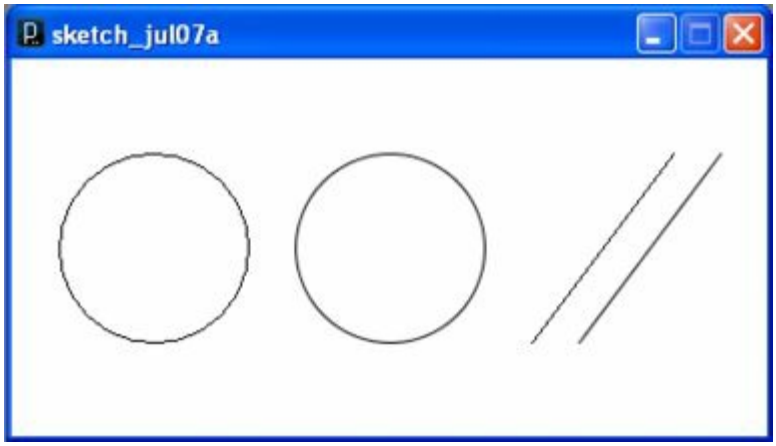


Figura 49: A função smooth( ) da linguagem Processing

```
size(400,200);
background(255);
ellipse(75,100,100,100);  // desenha um circulo
line(275,150,350,50);    // desenha uma linha diagonal
smooth();       // função de suavização ativada
ellipse(200,100,100,100);  // mesmo circulo suavizado
line(300,150,375,50);    // mesma diagonal suavizada.
```

Para desativar a aplicação do efeito de suavização utilize a função *noSmooth( )* antes da função que desenha a figura que não deve ser suavizada.

2 – strokeWeight( )

A função *strokeWeight( )* recebe um único parâmetro para definir a espessura do traçado das figuras desenhadas.

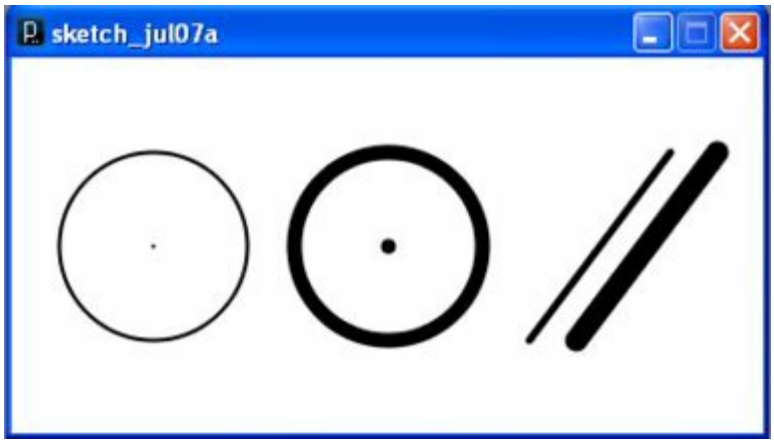
```
size(400,200);
background(255);
smooth();
strokeWeight(2);      // define espessura de 2 pixels
ellipse(75,100,100,100);  // circulo esquerdo
point(75,100);      // ponto de 2 pixels no centro do circulo
strokeWeight(4);      // espessura de 4 pixels
line(275,150,350,50);    // linha esquerda
strokeWeight(8);      // espessura de 8 pixels
ellipse(200,100,100,100);  // circulo direito
point(200,100);      // ponto de 4 pixels no centro do circulo
strokeWeight(12);    // espessura de 12 pixels
```

```
line(300,150,375,50);    // linha direita.
```

3 – strokeJoin( )

Essa função define o tipo de junção que liga os traços em um desenho de acordo com o parâmetro passado; se reto, cortado ou arredondado.

```
size(400,200);
background(255);
smooth();
strokeWeight(12);
strokeJoin(BEVEL);    // chanfra cantos externos
rect(75,50,100,100);    // retangulo
strokeJoin(ROUND);    // arredonda cantos externos
rect(250,50,100,100);    // retangulo.
```



O parâmetro **BEVEL** corta (chanfra) os cantos externos das junções dos traços; **ROUND** arredonda esses cantos; e **MITER**, que é o parâmetro *default*, mantem esses cantos.

4 - strokeCap( )

A função *strokeCap( )* é a equivalente da função anterior para linhas; seus parâmetros são **ROUND**, que é o parâmetro *default*, para arredondar os extremos da linha e **SQUARE**, para fazê-los retos.

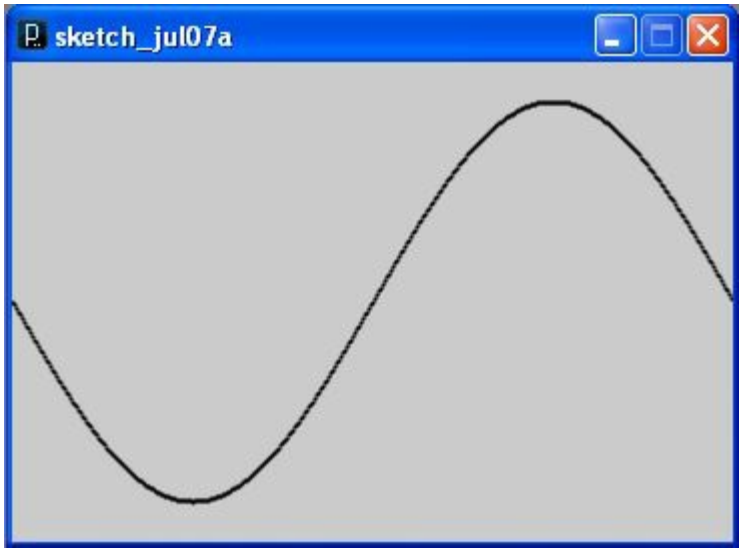
```
size(400,200);
background(255);
smooth();
strokeWeight(16);
line(100,150,200,50);    // linha por default arredondada
strokeCap(SQUARE);
line(250,150,350,50);    // linha com extremidades retas
```

A lista de todas as funções da linguagem *Processing* pode ser encontrada no endereço: **http://www.processing.org/reference/**.

Figura 50: A função strokeWeight( ) da linguagem Processing

Plotando gráficos simples

Conhecendo agora da linguagem *Processing* as oito funções de desenho e seus principais atributos, já podemos criar nossos primeiros gráficos; comecemos com uma senoide. Abra o PDE do *Processing* e copie a listagem abaixo na janela de edição.




```
//Processing – grafico de uma senoide
//desenha senoide numa janela 360x240
size(360,240);        // tamanho da janela
float angulo = 0.0;      // angulo da função seno
float resGrafica=360.0;          // resolução grafica
int amplitude=100;               // nivel p-p da senoide
for(int i=0; i < resGrafica; i++) {  // eixo-x com resGrafica pontos
strokeWeight(2);                 // espessura da linha da senoide
smooth();                     // suaviza contorno dos pontos
point(i, 120+amplitude*sin(angulo));  // desenha a senoide
angulo += TWO_PI/resGrafica;    // passo = 6.28/resolução.
}
```

Como esse *sketch* só será executado apenas uma vez, ainda não precisamos das funções *loop( )* e *draw( )*. Inicialmente dimensionamos uma janela de 360x240 *pixels* com a função *size( )*. A cor do fundo dessa janela é por *default* cinza. Criamos as variáveis em ponto flutuante **angulo** e **resGrafica**, e a variável inteira **amplitude**.

Figura 51: Plotando uma senoide com a linguagem Processing

A variável *angulo* guarda os valores dos ângulos que serão passados como parâmetros à função **seno**, *sin( )*; a variável *resGrafica* determina a resolução do gráfico que queremos plotar, ou seja, o número de 'fatias' da nossa senoide; a variável *amplitude* determina o nível pico-a-pico da senoide plotada. O comando de iteração *for* traça os pontos ao longo do eixo horizontal do gráfico. A função *strokeWeight()* determina o número de *pixels* de cada ponto do traçado da senoide, e a função *smooth()* torna o traçado gráfico mais suave.

É a função *point( )* que desenha a senoide dentro da janela criada no início do programa; essa função recebe como parâmetros os valores do contador **i** para percorrer o eixo horizontal, e os valores dos senos dos ângulos para plotar o eixo vertical. Algumas experiências podem ser feitas nesse *sketch*, por exemplo mudando os valores das variáveis *resGrafica* e *amplitude*, suprimindo a função *strokeWeight( )* ou a função *smooth( )* ou trocando a função *sin( )* por *cos( )* e observando o que acontece com o traçado gráfico. Experimente tambem trocar a função *sin(angulo)* por *random(-1,1)*.

Vamos agora plotar um gráfico interessante, o gráfico do **ruído Perlin** usando a função *noise( )*. O ruído Perlin é uma função matemática usada em programas gráficos para simular efeitos naturais como fogo, fumaça, nuvens e folhagens. Veja o *sketch* a seguir e o gráfico na figura 50.

```
//processing - experimento#
//plota ruido Perlin numa janela 360x240
size(360,240);        //tamanho da janela
float p=0.0;         //variavel ponto flutuante do passo
int amplitude=300;                //nivel p-p do ruido gerado
for(int i=0; i < width; i++) {      //tempo no eixo-x
strokeWeight(3);                 //pixels dos pontos do traçado
smooth();                       //suaviza contorno dos pontos
point(i, amplitude*noise(p));      //ruido amplificado no eixo-y
p+=0.01;                         //passo no eixo-x.
}
```

A função *noise( )* que aparece como parâmetro da função *point( )* gera uma sequência de números aleatórios entre 0,0 e 1,0 e seu parâmetro é incrementado em passos de 0,01.

Se incluirmos as funções *setup( )* e *draw( )* mudamos bastante o comportamento do programa. Veja como fica a listagem desse *sketch*:

```
//plota ruido de Perlin numa janela 360x240
//
int i;
int amplitude=300;                //nivel p-p do ruido gerado
float p=0.0;
void setup()  {
  size(360,240);
  }
void draw()  {
  background(255);
  for(int i=0; i < width; i++) {   //tempo no eixo-x
    strokeWeight(2);     //pixels dos pontos do traçado
    smooth();                    //suaviza contorno dos pontos
    point(i, amplitude*noise(p));  //ruido amplificado no eixo-y
    p+=0.01;                     //passo no eixo-x.
    }
  }
```

Figura 52: Plotando o gráfico de ruído Perlin com Processing

A função *draw( )* é chamada continuamente, assim todas as funções dentro dos colchetes desta função são executadas continuamente, e a uma taxa padrão de 60 vezes por segundo. Para desacelerar a amostragem do gráfico inclua dentro da função *setup( )* a função *frameRate(1)*; depois da função *size( )*. O parâmetro unitário passado a essa função reduz a taxa de atualização a um só quadro por segundo, permitindo visualizar melhor cada gráfico que é plotado. Experimente outras taxas mais altas nessa função *frameRate( )*.

Resumo do capítulo 6

- Processing é uma linguagem de programação de computadores criada segundo o conceito de *software* livre como uma ferramenta para desenvolvimento de projetos gráficos e interativos para artistas e *designers* com pouco conhecimento de programação de computadores.

- Com a função **size( )** definimos na tela do computador as dimensões altura x largura da janela dentro da qual um desenho em *Processing* será mostrado.

- Para posicionar um simples *pixel* em qualquer ponto de uma janela criada com a função *size( )* usamos a função **point( )** que requer os parâmetros da abcissa e da ordenada do ponto.

- Linhas são traçadas com a função **line( )** que requer quatro parâmetros: dois para marcar o início da linha e dois outros para marcar o final da linha.

- Para desenhar um triângulo utilizamos a função **triangle( )** que requer um par de parâmetros para cada um dos tres vértices do triângulo.

- A função **quad( )** serve para desenhar um quadrilátero e requer quatro pares de coordenadas, um para cada vértice do quadrilátero.
- Um retângulo é mais facilmente desenhado com a função **rect( )** que requer apenas dois pares de parâmetros: um para as coordenadas cartesianas de onde começa o retângulo e um par para definir sua largura e altura.

- Para desenhar uma elípse ou um círculo usamos a função **ellipse( )** que requer os mesmos tipos de parâmetros da função que desenha o retângulo. Para desenhar um círculo os parâmetros *largura* e *altura* devem ser iguais.

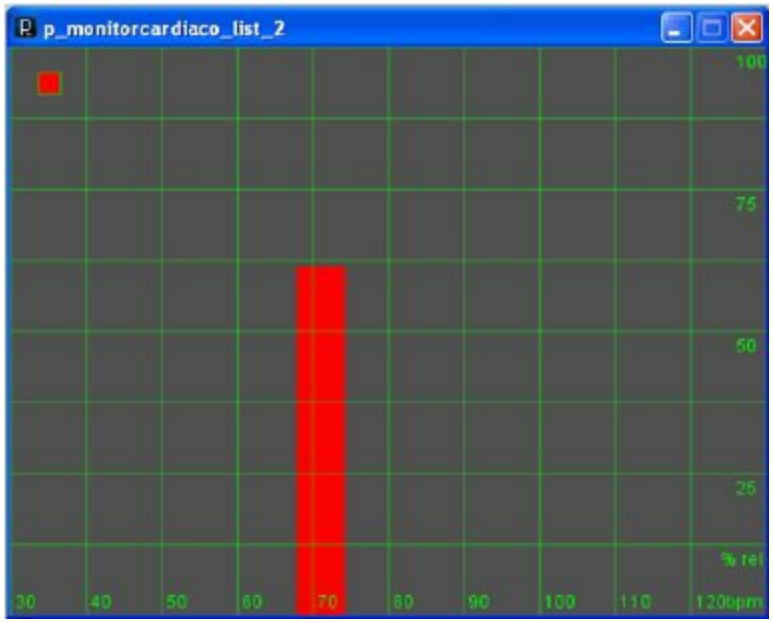
- Arcos de círculos são desenhados com a função **arc( )**, que exige dois parâmetros a mais que a função *ellipse( )*. Os dois últimos parâmetros marcam o ângulo, em radianos, de início e de término do arco.

- Linhas com aparência serrilhada podem ser melhor definidas com a função **smooth ( )** que suaviza os contornos das figuras geométricas desenhadas na janela de aplicação criada com *size( )*.

- Para definir a espessura do traçado das figuras desenhadas usamos a função **strokeWeight( )** *que requer* um único parâmetro.

- A função **StrokeJoin( )** define o tipo de canto, se reto, cortado ou arredondado, que liga as junções em um desenho.

- A função **strokeCap( )** é a equivalente da função *StrokeJoin( )* para linhas; seus parâmetros são **ROUND** para arredondar os extremos da linha e **SQUARE**, para fazê-los retos.



Capítulo 7

Experimentos com o Arduino e a linguagem *Processing*

Introdução

Esse capítulo será todo dedicado a experimentos com o Arduino e a linguagem *Processing*. Chegou a hora de juntarmos tudo o que vimos em todos os capítulos anteriores e começarmos a criar nossos próprios projetos inteligentes com o microcontrolador que existe no Arduino. A partir de agora o Arduino é de fato somente uma plataforma para nos tornarmos *experts* em um tipo de microcontrolador, o Atmel AVR ATmega328 (e por extensão em toda a familia AVR). Um circuito eletrônico desenvolvido com a plataforma Arduino pode ser depois controlado diretamente por um microcontrolador ATmega328 embarcado no próprio circuito, desde que com o *bootloader* do Arduino e um *sketch* gravado nele. E os programas em *Processing* e testados com esse *hardware* podem ser compilados e se tornar um executável no sistema operacional do seu computador, seja ele um PC, com *Windows* ou *Linux*, ou um MAC com *OS-X*. Nesse capítulo vamos desenvolver 4 experimentos, interessantes, são eles:

Experimento #9 - Comunicação Serial RS-232 e USB
Experimento #10 – Módulo de Desenvolvimento para o Arduino
Experimento #11 – Shield Matriz de contatos
Experimento #12 – Monitor de Batimentos Cardíacos

Experimento #9 – Comunicação serial

Comunicação serial é o processo de transmissão e recepção de dados um bit de cada vez entre um computador e um periférico, ou outro computador, atraves de um meio físico. Esse método é normalmente usado quando a taxa de transferência de informações é baixa e os equipamentos envolvidos estão distantes um do outro. É comum se usar o termo **porta serial** para designar a interface em um computador, ou em um periférico, que obedece a um protocolo de comunicação serial assíncrona, como as interfaces **RS-232**, **RS-485** e **USB**.

No protocolo RS-232 uma informação a ser transmitida é composta por 10 a 12 bits que são transmitidos sequencialmente. O primeiro é o bit que sinaliza o início da transmissão, a seguir vem um *byte* que forma o caracter a ser transmitido, depois vem um bit opcional de deteção de erro e por

fim um ou dois bits de indicação de término de transmissão. O formato mais comum é aquele conhecido por **8N1**, que quer dizer: 1 bit de partida, 8 bits com a informação, nenhum bit de paridade ou deteção de erro e 1 bit de parada. Esse é o formato usado pelo Arduino que, junto com a taxa de transferência em bits por segundo, deve ser configurado tambem no seu computador para que a troca de informação entre eles ocorra sem erros.

A porta serial USB se configura automaticamente quando o seu computador reconhece um novo dispositivo a ele conectado, mas precisa que um programa **driver** seja previamente instalado.

No Arduino a comunicação serial assíncrona é realizada por somente dois pinos: o pino digital 0, que é o pino de recepção (RxD), e o pino digital 1, que é o pino de transmissão (TxD) com a linha de terra como caminho de retorno comum. Normalmente esses dois pinos digitais do Arduino ficam reservados para a comunicação com o computador e portanto não podem ser usados para controle digital de entrada ou saída em seus *sketches*. O microcontrolador do Arduino trata esses sinais em níveis TTL que precisam ser convertidos para os níveis de tensão compatíveis com o protocolo RS-232 do computador por um circuito integrado conversor, o **MAX232N**.

Se a porta de comunicação serial do Arduino é USB, esses sinais devem ser convertidos para o protocolo USB por um outro circuito integrado conversor, o **FT232RL**.

Em alguns experimentos do capítulo 5 testamos *sketches* que enviavam valores numéricos pela porta serial para um computador conectado ao Arduino. Naqueles *sketches* para ativar essa comunicação serial em 9600 bits/s com o computador usamos a função **begin( )** da classe **Serial** dentro da função *setup( )*. E para transmitir o valor lido pelo sensor no pino analógico 14 usamos a função *println( )* dentro da função *loop( )*.

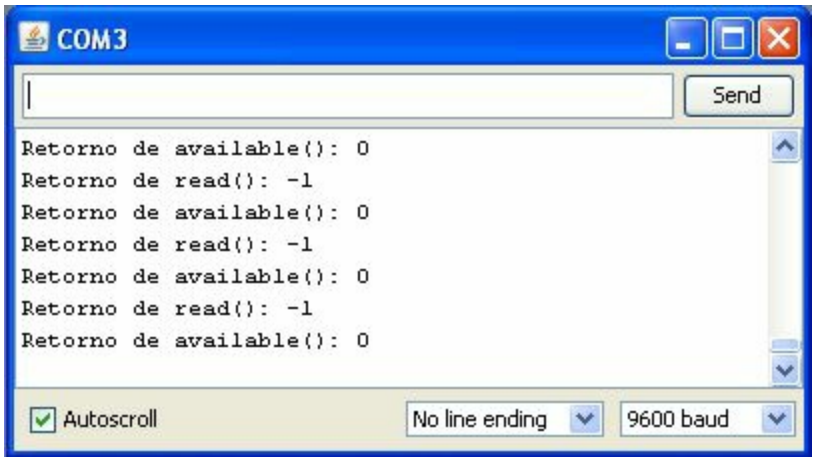
Nesse experimento vamos conhecer mais duas funções dessa **classe Serial** de controle do fluxo de dados entre o Arduino e o computador: a função **available( )**, que nos informa se existe dados disponíveis para leitura no *buffer* do computador, e a função **read( )**, que faz a leitura desses dados para o Arduino.

A função *available( )* retorna o total de dados (*bytes*) disponíveis no *buffer* da porta serial do computador; ela retorna **0** (zero) se nenhum dado estiver disponível. A função *read( )* fica lendo essa porta serial todo o tempo; se existe um dado disponível na porta essa função lê esse dado e o atribui a uma variável do programa em execução; se não há nenhum dado disponível na porta, quando *available( )* retorna 0, a função *read( )* retorna o valor **-1**.

Para maior compreensão do que foi dito carregue o seguinte *sketch* no seu Arduino e ative o terminal no IDE.

```
// Teste de retorno das funções available( ) e read( )
//
void setup() {
  Serial.begin(9600);       //inicia a porta serial
```

```
}
void loop()  {
  Serial.print("Retorno de available(): ");  //envia frase terminal
  Serial.println(Serial.available());    //envia total de dados
  delay(1000);       //pausa 1 seg
  Serial.print("Retorno de read(): ");  //envia frase
  Serial.println(Serial.read());    //envia dado disponivel porta
  delay(1000);       //pausa 1 seg.
  }
```



Ao ativar o terminal serial você verá que a cada segundo o retorno da função *available( )* será 0 e o da função *read( )* será -1, isso ocorre porque nenhum dado está disponível na porta serial do seu computador. A tela do terminal serial deverá ficar como na figura 51. Agora experimente digitar um só caracter no campo à esquerda do botão **Send** do terminal, e após enviá-lo, observar o que surge na tela do terminal. Depois envie, por exemplo, as letras **abcde** juntas e veja de novo o resultado. Observe que a função *available( )* informa inicialmente que existe 5 caracteres a ser enviados; e a função *read( )* mostra o código **ASCII** do primeiro deles, o código **97** que corresponde à letra **a**. Os outros caracteres vão sendo enviados sequencialmente enquanto *available( )* vai decrementando até **0** de novo.

Para nosso primeiro experimento com recepção de dados pelo seu computador vamos montar novamente o circuito da figura 23, aquele com um LED tricolor (ou tres LEDs de cores diferentes) e tres resistores de 390 ohms, e usar os mesmos pinos digitais do Arduino daquele experimento, os pinos 9, 10 e 11.

Carregue o *sketch* abaixo no seu Arduino e no teclado do seu computador digite qualquer uma das letras **r**, **g** ou **b** e as envie pelo terminal serial para acender um de cada vez os LEDs com o Arduino, ora o vermelho, ora o verde, ora o azul.

Nesse *sketch* criamos uma função que vai acender cada LED, a função *acendeLed( )*, declarada logo na primeira linha do código. A porta serial é iniciada em 9600 bits/s dentro de *setup( )*, onde tambem, em uma única linha, o comando *for* configura os tres pinos que vamos usar como saídas. Dentro de *loop( )* aparece a função *available( )* sendo testada por um comando *if* pela presença de dados disponíveis na porta para leitura. Se *available( )* retornar qualquer valor diferente de zero o

comando **switch...case** é executado. Veja a listagem abaixo.

Figura 53: Testando as funções available( ) e read( )

```
//Teste dos LEDs RGB
//
int acendeLed(int j);      //função para acender LED
void setup()  {
  Serial.begin(9600);    //inicia porta serial em 9600 b/s
  for(int i=9;i<=11;i++) pinMode(i,1);  //pinos 9 a 11 como saida
  }
void loop()  {
  if(Serial.available())  {    //teste da presença de dados
  switch(Serial.read())  {    //decisão baseada leitura da porta
    case ('r'):acendeLed(9);  //se 'r' acende LED vermelho
    break;
    case ('g'):acendeLed(10);  //se 'g' acende LED verde
    break;
    case ('b'):acendeLed(11);  //se 'b' acende LED azul
    break;
    default:Serial.println("Escolha 'r', 'g' ou 'b'");
            //mensagem de erro
    }
  }
}
int acendeLed(int j) {      //pino j é passado a função
  for(int i=9;i<=11;i++) digitalWrite(i,0);  //apaga todos LEDs
  digitalWrite(j,1);      //acende LED de acordo com j.
}
```

O comando *switch...case( )* nos permite testar uma mesma variável para varios valores possíveis, e assim tomar uma decisão. Se existe um dado novo na porta serial o comando *switch...case( )* verifica se esse dado é a letra **r** (red), **g** (green), ou **b** (blue). Se não for nenhuma dessas letras uma mensagem de erro é enviada ao terminal; se for qualquer uma dessas tres letras a função a*cendeLed( )* é chamada e a ela é passada o número do pino correspondente pela variável **j**, declarada no início do código junto com a função. Essa função recebe o número do pino, 9, 10 ou 11, apaga qualquer LED que esteja aceso e acende somente aquele LED que corresponde ao pino passado como parâmetro.

Essa capacidade de enviar e receber dados pela porta serial entre o Arduino e um computador pode ser usada como uma ferramenta de depuração (*debugger*) no desenvolvimento de seus programas. Por exemplo, se o leitor quiser conferir o valor intantaneo de uma variável qualquer em algum ponto do programa pode acrescentar a função *println( )* nesse ponto do código e enviar essa variável para o monitor serial, como fizemos acima no *sketch* que testa os retornos das funções *available( )* e *read( )*.

Nosso primeiro experimento integrado do Arduino com a linguagem *Processing* foi aquela em que criamos, no capítulo 6, um LED virtual na tela do computador que piscava em sincronismo com o LED real no pino 13 do Arduino. Essa interface Arduino-*Processing* é feita via cabo serial RS-232 ou USB entre o Arduino e o PC (ou MAC). O programa gravado no ATmega328 do Arduino e o programa no computador, compilado a partir de um código fonte em *Processing*, trocam dados por uma porta serial. Nesse experimento vamos conectar duas chaves mecanicas e dois resistores de 10 K ohms em dois pinos digitais do Arduino e visualizar o acionamento dessas chaves na tela do computador.

Uma chave deve ser conectada entre o pino digital 2 e o pino com a tensão de 5 volts; um resistor de 10 K ohms deve ser conectado desse mesmo pino 2 para o pino de terra. A mesma coisa deve ser feita no pino digital 3. Depois abra o IDE do Arduino e copie e compile o código fonte abaixo. Grave o programa no Arduino.

```
// Teste com duas chaves mecanicas
//
void setup( ) {
  Serial.begin(9600);              //inicia porta serial em 9600 b/s
  }
void loop( ) {
  if(digitalRead(2))
    Serial.write('A');//se chave A do Arduino fechada envia 'A'
  if(digitalRead(3))
    Serial.write('B');//se chave B do Arduino fechada envia 'B'
  if(!digitalRead(2)&&!digitalRead(3))
    Serial.write('0');     //se A E B abertas envia '0'.
}
```

Agora abra o Monitor Serial do IDE do Arduino e pressione cada chave e verifique se no Monitor aparece os caracteres ASCII 68 para uma das chaves e o caracter 69 para a outra chave; se nenhuma delas estiver pressionada o ASCII 0 deverá ser observado.

Desabilite o Monitor Serial do Arduino e minimize o seu IDE. Agora abra o PDE do *Processing* e copie o código a seguir para a janela de edição.

```
// recepcao caracteres  A, B e 0 do Arduino pela porta serialimport processing.serial.*;
//importa biblioteca serial
Serial port;                    //cria objeto port
//
void setup()  {
 size(300,200);           //cria janela 300x200 pixels
 frameRate(30);        //define taxa de 30 quadros/s
 port=new Serial(this, "COM3", 9600);    //inicia primeira porta 9600
```

```
 PFont font;                //cria objeto font
 font = loadFont("ArialMT-48.vlw");  //carrega fonte 'ArialMT-48'
 textFont(font);                //fonte para todos textos
 }
 //
void draw()  {
 while(port.available()>0){
 background(255);             //fundo janela branca
 if(port.read()==65)  {          //verifica se chave A foi fechada
 fill(255,0,0);                 //se sim seleciona cor vermelha
 text("A",100,100);             //e  mostra 'A' na janela aberta
 }
 if(port.read()==66)  {         //verifica se chave B foi fechada
  fill(0);                     //se sim seleciona cor preta
  text("B",150,100);      //e mostra 'A' e 'B' na janela aberta
  }
 if(port.read()==0)  { }         //se nem A nem B para programa.
  }
 }
```

Esse *sketch* em *Processing* reconhece dois caracteres recebidos do Arduino, via porta serial, e mostra numa pequena janela na tela do computador o caracter recebido. Para implentar essa comunicação serial é preciso antes importar uma biblioteca para o código fonte do *sketch*, o que é feito com o comando **import processing.serial.*** na primeira linha depois dos comentários. Tambem é necessário carregar a fonte indicada no código; para isso vá em **Tools > Creat Font...** no PDE do *Processing* e na janela que surgir escolha a fonte **ArialMT** na lista e o tamanho **48** no campo **size.**

A linguagem *Processing* é muito semelhante à linguagem do Arduino; ambas são baseadas na linguagem C, e seus ambientes de desenvolvimento baseados na linguagem **Java**. O IDE do Arduino abre os arquivos fontes da linguagem *Processing*, e vice-versa; muitas vezes até os compilam com a extensão trocada.

Experimento #10 – Montagem de um Módulo de Desenvolvimento para o Arduino

Para por em prática tudo aquilo que aprendemos nos capítulos anteriores vamos precisar de um conjunto de componentes eletronicos como chaves, resistores, potenciometros, capacitores e alguns transistores e circuitos integrados. Para não ter quer ficar montando um *hardware* difererente para cada experimento, sugerimos ao leitor investir um pouco de seu tempo na montagem de uma placa de interface com os principais componentes, para estudar o Arduino.

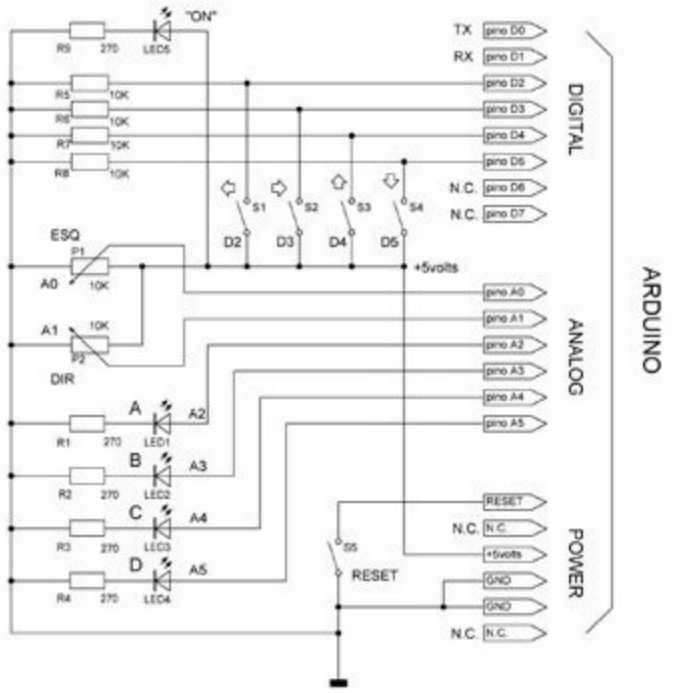


Figura 54: Circuito do Módulo de Desenvolvimento para o Arduino

Já vimos que os blocos de *software* no mundo do Arduino são chamadas de *sketches*, os de *hardware* de *shields*. Essa placa de interface, ou *shield*, contem somente dois potenciometros de 10 K ohms, quatro minichaves mecanicas e quatro LEDs de 5 mm. Os *shields* são feitos de modo que se encaixem sobre os conectores do Arduino, e deste recebam alimentação e acesso a todos os pinos digitais e analógicos e tambem aos pinos de *reset* e do conector **ISP**, aquele de programação do *bootloader*.

Hardware

Para esse experimento utilizamos uma placa padrão já perfurada de 65x52 mm em nossa montagem, como a da figura 55, ao lado. As conexões são feitas por fios que interligam os componentes na face da solda. Mas existe um problema com essa solução simples de se montar um shield: a distancia entre os dois conectores digitais no Arduino não obedece ao padrão 2,54mm da placa padrão; porisso abrimos mão do conector digital da esquerda e montamos somente o conector da direita e os conectores analógico e de alimentação. Veja na figura 54 o circuito do módulo, na figura 55 como o módulo é encaixado no Arduino.

No diagrama do circuito na figura 53 vemos as quatro chaves conectadas aos pinos digitais 2 a 5 (os pinos 0 e 1 são pinos reservados para a comunicação serial) e os cursores dos dois potenciometros ligados nos pinos analógicos 14 e 15. Os quatro LEDs são ligados aos outros quatro pinos analógicos 16 a 19. Foi incluido na montagem uma chave de reset e um LED vermelho de 3 mm somente para indicar que o Arduino está alimentado. Os pinos 6 e 7 não são aqui utilizados. A montagem desse shield é bastante simples pois os únicos componentes que devem ser observados quanto à polarização são os LEDs. Na placa perfurada observe a correta distancia entre as barras de pinos digital e analógica para que se encaixem corretamente nos conectores fêmea do Arduino. Depois de montado o módulo e conferida cada ligação encaixe o shield no Arduino. A alimentação para o

circuito virá do seu computador pelo cabo USB, por onde o Arduino e o seu computador tambem vão se comunicar.

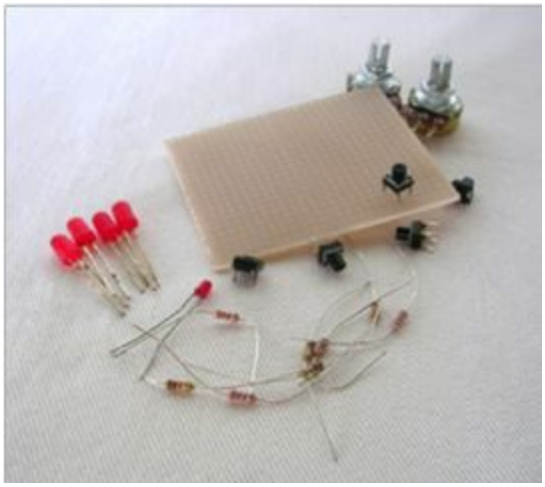


Figura 55: Componentes do Módulo de E/S

Software

Podemos testar as funções *digitalRead( )* e *digitalWrite( )* de entrada e saída digitais da linguagem do Arduino junto com as chaves e os LEDs desse *shield* com o *sketch* abaixo.

```
// Teste das 4 chaves com os 4 LEDs
//
int pinSwitch=2;        //mude para 3,4 ou 5
int pinLed=16;        //mude para 17, 18 ou 19
void setup()  {
  pinMode(pinLed,1);    //pino do LED selecionado saida
void loop()  {
  (digitalRead(pinSwitch) == 1)? digitalWrite(pinLed,1):        digitalWrite(pinLed,0);
}
```

Lista de componentes	
R1 a R4 e R9	Resistor 270 ohms x 1/8W
R5 a R8	Resistor 10Kohms x 1/8 W
P1 e P2	Potenciometro linear 10Kohms
LED1 a LED4	Led vermelho 5 mm

LED5	Led vermelho 3 mm
S1 a S5	Chave miniatura 1/0
C1 a C3	Barras de pinos 2,54mm
Outros	Placa CI perfuração padrão

As variáveis *pinSwitch* e *pinLed* indicam os pinos no Arduino onde estão conectados uma chave mecanica e um LED, respectivamente. O estado de *pinSwitch* é testado pelo operador ternário **?** que vai fazer ou não o LED acender. Note que nesse *sketch* o pino analógico 16 é configurado como saída e o pino 2, por *default* já é entrada; se a chave no pino 2 for fechada o LED no pino 16 vai acender.

Para testar a função *analogRead( )* com um dos potenciometros do módulo carregue o *sketch* abaixo. A posição do cursor do potenciometro depois de digitalizada pela função *analogRead( )* serve de parametro para a função *delay( )* que determina quanto tempo o LED vai ficar aceso (ou apagado). A variável *statusLed,* que a cada ciclo muda de estado, é que vai determinar quando o LED vai estar aceso ou apagado.

```
// Teste dos potenciometros com os 4 LEDs
//
int pinSensor=14;     //pot no pino 14 ou 15
int pinLed=16;        //LED nos pinos 16,17,18 ou 19
int statusLed=LOW;
void setup() {
  pinMode(pinLed,OUTPUT);  //pino do LED como saida
```

```
}
void loop() {
  delay(analogRead(pinSensor));  //leitura do pot
  digitalWrite(pinLed,statusLed);  //acende/apaga LED
  statusLed=!statusLed;     //complementa estado anterior.
}
```

Acrescentando ao *sketch* acima um comando *for* para configurar de uma só vez como saída os quatro pinos com LEDs e um outro comando *for* com a mesma estrutura para selecionar um LED de cada vez podemos criar um **bargraph** para testar os quatro LEDs com um só potenciometro. Veja a listagem abaixo.

Figura 55: Módulo de E/S montado

```
int pinSensor=14;       //pot no pino 14 ou 15
int statusLed=LOW;
void setup() {
  for(int i=16; i<=19; i++) pinMode(i,OUTPUT);  //16 a 19 saidas
void loop() {
  for (int i=16; i<=19; i++) {
int pinLed=i;
delay(analogRead(pinSensor));  //leitura do pot
digitalWrite(pinLed,statusLed);   //acende/apaga LED
statusLed=!statusLed;       //inverte estado anterior.
}
```

Experimente modificar o *sketch* acima para inverter a sequencia de acionamento dos LEDs ou tente escrever um outro que utilize tambem as quatro chaves no mesmo código.

Experimento #11 – Montagem de um *Shield* Matriz de Contatos

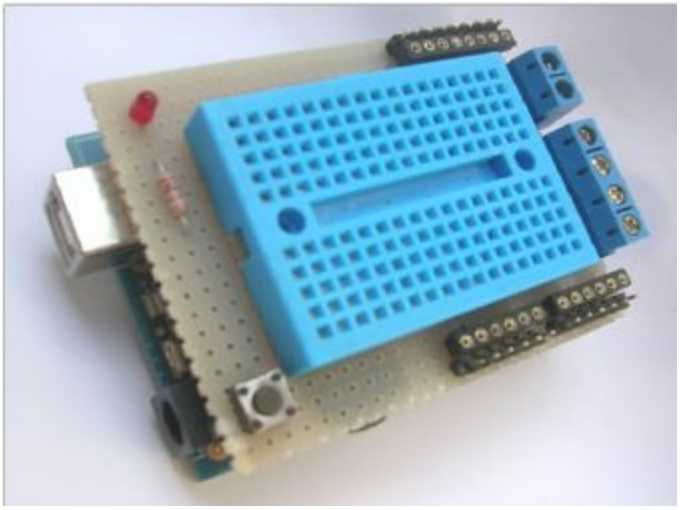


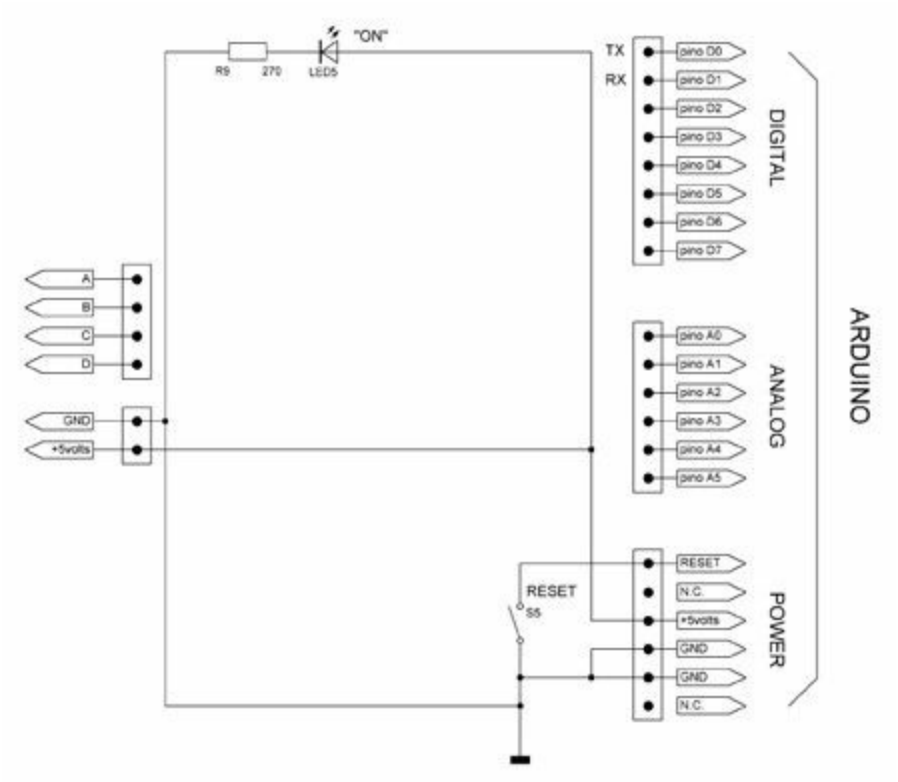
Figura 56: Matriz de Contatos encaixado no Arduino

As matrizes de contatos, tambem conhecidas por *protoboards*, são ainda uma ótima solução quando queremos testar algum circuito novo. E se pudermos conectar esse circuito ao nosso computador via porta serial a experiência fica muito mais interessante. É muito fácil montar uma pequena matriz de contatos sobre o Arduino e usá-lo como interface entre um circuito experimental e o nosso computador. Nessa montagem, na figura abaixo,tem como base uma pequena placa de fibra de vidro de 65x52 mm com perfuração padrão 2,54 mm.

Figura 57: Circuito completo do Módulo Matriz de Contatos

Hardware

A pequena matriz de contatos de 45 x 35 mm vem com uma fina camada de adesivo protegida por uma fita plástica que deverá ser retirada antes de sua fixação na placa perfurada. Monte as tres barras de pinos de 2,54 mm na placa perfurada testando as distancias entre os conectores fêmea do Arduino. Só estamos usando o conector digital da direita e os conectores analógico e de alimentação. A distancia entre os dois conectores digitais no Arduino não obedece ao padrão 2,54 mm da placa perfurada. Junto a cada barra de pinos macho no *shield* existe uma barra de soquetes torneados do tipo usado para circuitos integrados DIL com seus pinos ligados a cada pino digital, analógico e de alimentação do Arduino. Tambem do lado oposto ao LED e ao botão de *reset* existe dois outros conjuntos de conectores do tipo **borne** com parafusos, um com quatro pinos, para conexões entre a matriz de contatos e algum circuito externo à matriz, e outro com dois pinos com + 5 volts e terra para alimentar esse circuito externo.



Lista de componentes	
R1	Resistor 270 ohms x 1/8W
LED1	Led vermelho 3 mm
S1	Chave miniatura 1/0
C1 a C3	Soquetes para CI em tiras
B1 e B2	Bornes com parafusos
C1 a C3	Barras de pinos 2,54mm
Outros	Placa CI perfuração padrão
	Matriz de contatos pequena

Pequenos circuitos com transistores e CIs montados nessa matriz de contatos e nela alimentados com

5 volts podem ter acesso direto aos pinos digitais 0 a 7 e a todos os pinos analógicos do Arduino. Todas as conexões entre os contatos da matriz e os pinos do Arduino serão feitas por *jumps*. Como no shield de entradas e saídas do experimento #2, aqui temos também um botão de reset e um LED vermelho de 3 mm que indica que o Arduino está alimentado. Observe que existe também dois conjuntos de bornes, um de quatro pinos e identificados pelas letras A,B,C e D eoutro de dois pinos com 5 volts e terra, para conexões entre a matriz e circuitos experimentais externos. Junto a cada um desses bornes e também junto aos conectores digital, analógico e de alimentação do Arduino, existe uma pequena barra de pinos torneados, do tipo usado para circuitos integrados DIL, onde deverão ser conectados os jumps para a matriz de contatos. O proximo experimento desse capitulo o leitor poderá montar já nesse *shield* um Monitor de Batimentos Cardiacos.

Experimento#12 – Um Monitor de Batimentos Cardíacos com o Arduino

Parte 1 - "Vendo" os pulsos cardíacos

A Eletronica nos permite criar aparelhos que são de fato extensões de nossos sentidos, como aqueles que são capazes de captar certos sinais fisiológicos que normalmente somos incapazes de sentí-los. Uma vez captados por algum tipo de sensor esses sinais podem ser amplificados e digitalizados, depois podem ser alterados e até mesmo controlados com alguma forma de *feedback*. Um desses sinais fisiológicos é o número de batimentos do nosso coração. Emoções fortes como raiva e excitação causam um aumento da taxa de batimentos do coração humano. Outras emoções como tristeza e pesar fazem essa taxa diminuir. Meditação, contemplação e outros estados mentais tambem alteram essa taxa. Com a ajuda do Arduino e um amplificador de alto ganho com sensor ótico podemos monitorar nossa taxa de batimentos cardíacos na tela do nosso computador. Fica fácil assim montar um instrumento que pode ser usado para experiências com controle de stress, respiração, relaxamento e *biofeedback*.

Hardware

Nosso protótipo foi montado numa pequena placa perfurada de 3,5 x 5,5 cm para ser encaixada sobre o Arduino como um *shield*. Veja a figura 57. Os componentes são interligados com pequenos fios no lado da solda da placa.

Nesse projeto utilizamos o par de amplificadores operacionais dentro do **LM358N**. O circuito completo está na figura 58, e é um projeto por nós adaptado para o Arduino a partir de um circuito

originalmente publicado pela revista inglesa **Elektor** em 2008, sob o título "*Stress-O-Meter*". Esse tipo de monitor de batimentos cardíacos é uma variante de um aparelho médico-hospitalar chamado **pletismógrafo**, que serve para medir (e registrar) variações no volume de um orgão como resultado de flutuações rítmicas da circulação do sangue pelo corpo humano. Normalmente nesse tipo de aparelho essas flutuações são captadas por um sensor ótico posicionado em um lado do lóbulo de uma orelha, ou um lado da ponta de um dos dedos, e com a fonte de luz alinhada no lado oposto. Diferentemente, nosso aparelho mede as pulsações no fluxo de sangue na ponta do dedo de uma das mãos do experimentador com o sensor e a fonte de luz colocados num mesmo plano e montados numa caixa separada do amplificador. Em nosso protótipo usamos como sensor de pulsações um LDR comum com uma resistencia de 1 M ohm sem nenhuma luz incidente, e 400 ohms com incidencia de luz natural direta. Veja a figura 59 abaixo.

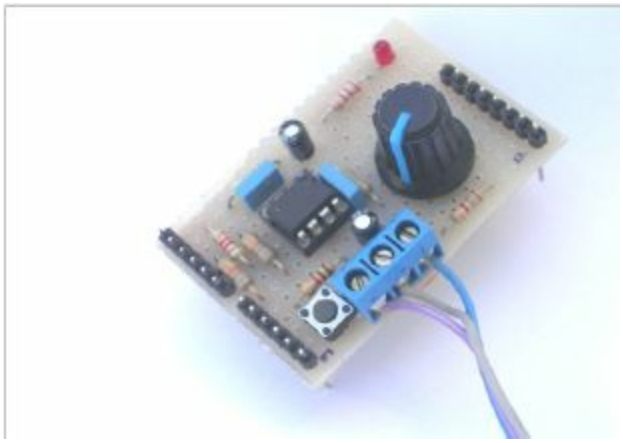


Figura 59: Monitor cardíaco montado

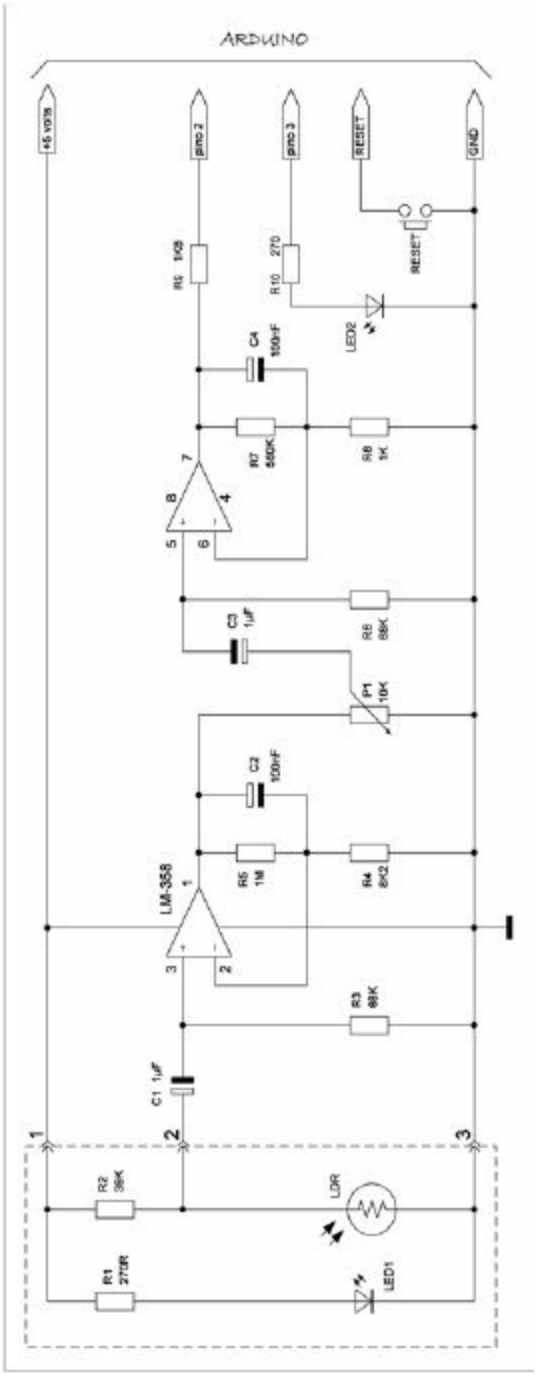


Figura 60: Circuito completo do Monitor de batimentos cardíacos

Como fonte de luz para o sensor usamos um LED vermelho comum de 3 mm. O experimentador deve posicionar o lado oposto à unha de seu dedo indicador sobre o LDR, e a junção entre a ponta e o meio do dedo sobre o LED. A luz que é emitida pelo LED atravessa a pele da junção e é refletida pelo osso sobre uma pequena concentração de artérias bem em cima do LDR. O volume de sangue nesse grupo de artérias pulsa em sintonia com as contrações do músculo cardíaco. Essa informação vai modular a resistência do LDR.

No circuito do monitor o resistor R1 limita a corrente direta atraves de LED1 em cerca de 20 mA. O LDR e o resistor R2 formam um circuito divisor de tensão cuja saída pulsante será função da resistência do LDR que é função da luz refletida pelo dedo do experimentador. Esses pulsos de muito baixa frequencia, entre 1 e 2 hertz, seguem para um filtro passa-altas formado por C1 e R3 e é amplificado pelo primeiro opAmp do LM358N na configuração não-inversor com ganho de 120. C2 e R5 formam um filtro passa-baixas centrado em 1,5 Hz. Essa frequencia corresponde a 90 pulsos

por minuto, que é a metade da frequencia maxima do coração humano.

O potenciometro P1, que é a resistencia de carga do primeiro amplificador, controla a entrada do segundo opAmp tambem não-inversor com ganho de 560. Aqui o sinal modulado com os batimentos do coração do experimentador pode ser entregue para tratamento ao Arduino no pino digital 2. O LED2 será programado para piscar com os batimentos cardíacos.

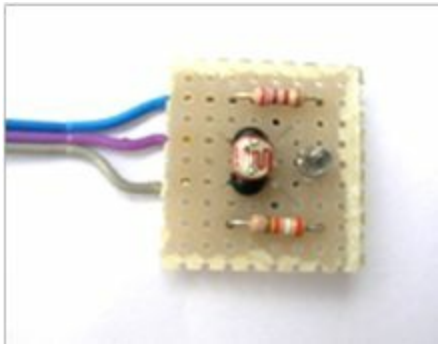


Figura 61: Montagem do sensor

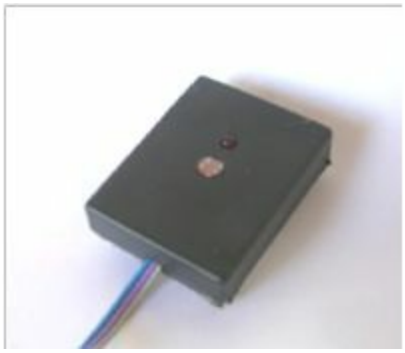


Figura 62: Sensor montado

Sugerimos ao leitor primeiro montar o conjunto fonte de luz/sensor. Para o nosso protótipo montamos esse conjunto em uma pequena caixa plástica de 3x4x1 cm, como na figura 60. Nessa caixa foi montada a pequena placa perfurada com o LED vermelho de 3 mm e seu resistor de 270 ohms e o LDR e seu resistor de 39 Kohms. A distancia entre os centros do LED e do LDR é cerca de 1,0 cm. Da caixa saem tres fios: o de 5 volts, o de saída do divisor de tensão e o comum a esses dois, o terra. O fio de saída do divisor de tensão vai para o capacitor C1 na entrada do primeiro amplificador pelo borne B1.

Software

Depois de conferir mais de uma vez toda a fiação, encaixe o *shield* no seu Arduino, conecte o sensor e depois o cabo USB no seu computador. Para ver os batimentos cardíacos de uma pessoa carregue no seu Arduino o *sketch* da listagem abaixo.

```
int pinLed = 3;
volatile int state = 0;
void setup()  {
 pinMode(pinLed, OUTPUT);       //pino 3 saida
 attachInterrupt(0, pulse, CHANGE);//pino 2 entrada interrupção
}
```

```
void loop()  {
  digitalWrite(pinLed, state);       //LED pisca com batimentos
}
void pulse()  {
  state = !state;                 //complementa state.
}
```

Esse *sketch* configura o pino digital 3 como saída para LED2 no circuito, e o pino digital 2 como entrada de interrupção para a função *attachInterrupt( )* que chama a função *pulse( )* toda vez que uma interrupção externa ocorrer nesse pino. Já conhecemos a função *attachInterrupt( )* do capítulo 5, quando montamos o contador de dois dígitos. Vimos que ela requer tres parâmetros. O primeiro deverá ser o dígito 0 para o pino digital 2, ou 1 para o pino digital 3 do Arduino; o segundo parâmetro é a função que será chamada quando a interrupção ocorrer; e o último parâmetro define o momento em relação ao sinal no pino digital especificado em que a interrupção ocorrerá, que poderá ser de quatro modos: *CHANGE*, na mudança do nível no pino; *LOW*, quando o pino for baixo; *RISING*, quando o pino mudar de baixo para alto; ou *FALLING*, quando o pino mudar de alto para baixo.

Nesse *sketch* toda vez que o nível lógico no pino digital 2 mudar, a função *pulse()* será chamada, e esta vai somente complementar (mudar o estado da) a variável *state*. Essa variável *state* vai mudar de acordo com os batimentos do coração do experimentador e fazer o LED2 acender ou apagar conforme seu estado na função *digitalWrite( )*.

Para testar o circuito basta o leitor repousar a ponta de seu dedo indicador sobre o sensor e girar o cursor do potenciometro P1 para o mínimo ganho até o LED no pino 3 apagar. Depois vá aumentando o ganho até que o LED comece a piscar com os batimentos do seu coração. Depois de algum treino pressionando mais ou pressionando menos o dedo sobre o sensor é possível achar o ponto ótimo para ver oLED piscar regularmente. O ajuste fino é feito com o potenciometro.

Lista de componentes	
R1 e R10	Resistor 270 ohms x 1/8W
R2	Resistor 39K ohms x 1/8W
R3 e R6	Resistor 68K ohms x 1/8W
R4	Resistor 8K2 ohms x 1/8W

R5	Resistor 1M ohms x 1/8W
R7	Resistor 560K ohms x 1/8W
R8	Resistor 1K ohms x 1/8W
R9	Resistor 1K8 ohms x 1/8W
P1	Potenciometro 10K linear
LED1 e LED2	Led vermelho 3 mm
LDR	LDR
S1	Chave miniatura 1/0
C1 e C3	Capacitor 1uF x 25V
C2 e C4	Capacitor 100nF
CI1	LM358N dual opamp
Outros	Placa CI perfuração padrão
	Borne 3 pinos parafuso

Podemos enviar nossos batimentos cardíacos pela porta serial para o computador e ver no terminal serial do *Windows* ou do Arduino sua forma digital como uma combinação de traços horizontais com caracteres ASCII **95** e **45**. Para isso acrescente ao *sketch* acima uma linha para iniciar a porta serial em 9600 bps e os comandos *if* que testam a variável *state*; se esta for falsa o caracter 45 será enviado para o terminal serial com a função *Serial.write( );* se *state* for verdadeira o caracter 95 será enviado. Veja a listagem abaixo e a forma dos pulsos na figura 60. As pausas de 10 ms entre as transmissões evitam a sobrecarga do *buffer* da porta serial.

```
// Enviando os pulsos cardiacos pela porta serial
```

```
//
int pinLed = 3;              //pino 3 saida
volatile int state = 0;
void setup()  {
  pinMode(pinLed, OUTPUT);
  attachInterrupt(0, pulse, CHANGE); //pino entrada interrupção
  Serial.begin(9600);      //porta serial 9600 b/s
  }
void loop()  {
  digitalWrite(pinLed, state);        //LED pisca com batimentos
  if(!state) Serial.write(45);    //envia ascii 45 se falso
  delay(10);
  if(state)Serial.write(95);  //envia ascii 95 se verdadeiro
  delay(10);          //pausa 10 ms
  }
void pulse(){
  state = !state;                  //complementa state.
  }
```

Figura 63: Pulsos captados com o Monitor de batimentos cardíacos

Parte 2 - Uma interface gráfica com *Processing*

Nessa segunda parte de nosso projeto de um Monitor de Batimentos Cardíacos com o Arduino vamos criar uma interface gráfica simples para mostrar dinamicamente os pulsos gerados pelo coração do experimentador e que são captados por um sensor ótico na ponta de seu dedo. Nesse trabalho vamos criar um instrumento virtual com uma tela graticulada com a linguagem *Processing*. Veja a figura 61.

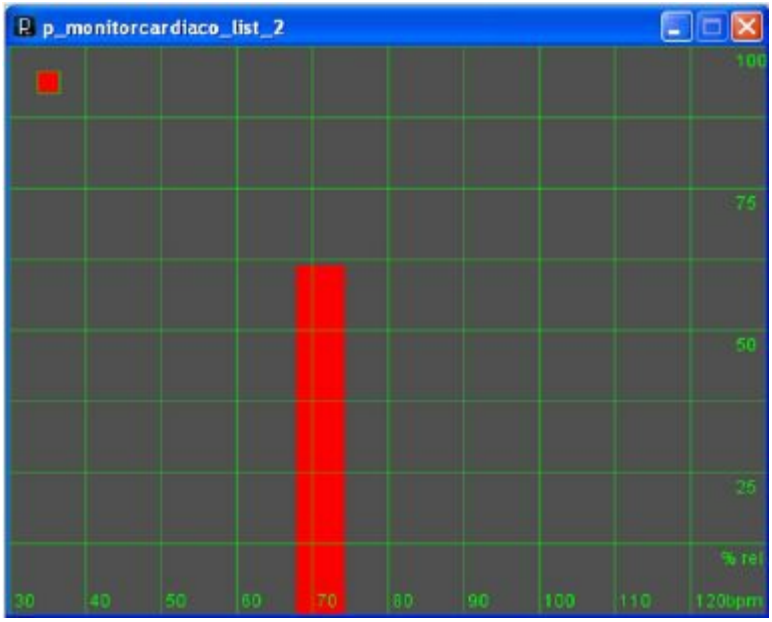


Figura 64: interface gráfica para o Monitor de Batimentos Cardíacos

Hardware

Na primeira parte desse projeto os pulsos de saída do amplificador do sensor iam direto para o pino digital 2 do Arduino que com o primeiro *sketch* fazia um LED no pino digital 3 piscar de acordo com os batimentos cardíacos do experimentador. O segundo *sketch* mostrava na tela do terminal esses batimentos em forma de pulsos digitais. Agora em vez de conectar no pino 2 vamos conectar a saída do amplificador diretamente ao primeiro pino analógico do Arduino, o pino 14.

Software

Primeiramente carregue o *sketch* da listagem abaixo no seu Arduino. O LED do circuito do monitor com esse *sketch* não vai mais piscar.

```
//Monitor de batimentos cardiacos - listagem para o Arduino
//
int pinSensor = 14;      //pino 14 recebe saida do amp
int sensor=0;
void setup() {
  Serial.begin(9600);     //porta serial em 9600 b/s
}
void loop() {
  sensor=analogRead(pinSensor);  //leitura da saida do amp
  Serial.write(sensor>>2);    //converte para 0 a 255
  delay(10);          //pausa 10 ms.
}
```

Nesse *sketch* do Arduino o sinal analógico da saída do amplificador que entra no pino 14 é convertido para digital pela função *analogRead( )* com 10 bits de resolução e guardado na variável *sensor*. Para caber num só *byte* e ser enviada pela porta serial para o seu computador a informação do sensor é dividida por 4 deslocando-se dois bits à direita pelo **operador de deslocamento à direita '>>'**. A sintaxe desse operador tem a seguinte forma:

variavel >> número de bits

O deslocamento de 1 bit à direita divide a variável antes do operador por 2. Existe tambem o **operador de deslocamento à esquerda** '<<' , que multiplica a variável antes do operador por 2 a cada bit deslocado à esquerda.

Agora abra o PDE da linguagem *Processing* e copie nele o código da listagem abaixo. Vamos entender todo o *sketch* dividindo-o em quatro partes: no topo da listagem está o conjunto de variáveis declaradas que serão usadas pelos comandos da linguagem; depois a função *setup( )* e a *função draw( )*; embaixo o conjunto de tres funções que foram criadas para desenhar a grade, colocar texto e um marcador de segundos na tela.

A primeira linha da listagem importa a **biblioteca de comunicação serial** da linguagem *Processing*; a segunda linha cria o objeto **port** que será usado dentro da função *setup( )* para estabelecer a comunicação serial do Arduino com o seu computador. Logo depois algumas variáveis globais são declaradas e o objeto **font** é criado. Essa função *setup( )* é semelhante à da linguagem do Arduino, é chamada somente uma vez e serve para configurar a forma como a interface gráfica com o usuário será apresentada na tela do computador. Dentro dela a função *size( )* cria uma janela de 500x400 *pixels* onde serão exibidos os objetos gráficos que criarmos com o nosso *sketch*. É recomendável que essa função seja sempre a primeira dentro de *setup( )*. A função *background( )* faz o fundo da janela toda de uma mesma cor, aqui preta. O objeto *port*, criado no topo do código, estabelece a porta 3 como o canal de comunicação serial com o Arduino em 9600 bps, por fim a fonte **Arial** é carregada.

```
//Monitor de batimentos cardiacos - listagem para Processing
//
import processing.serial.*;      //importa biblioteca serial
Serial port;           //cria objeto port
float bpm=0;           //
float sensor=0;
float counter=0;
int prevMillis=0;
int trigger=25;
PFont font;            //cria objeto font
void setup()  {
  size(500, 400); smooth();
  background(80);       //fundo tela preto
  port=new Serial(this,"COM3",9600);//inicia porta serial em 9600
  font = loadFont("ArialMT-32.vlw");  //carrega fonte Arial
```

```
}
void draw()  {
  while(port.available()>0)  {    //se tiver dado na porta
  background(80); stroke(255,0,0); strokeWeight(2);
  sensor=map(port.read(),0,255,0,height);  //expande leitura porta
  fill(255,0,0); rect(bpm*10+100,400,30,-sensor);
  if (map(port.read(),0,255,0,100) >= trigger)
  counter++;
  if(millis() - prevMillis >= 1000){
  bpm=counter; counter=0;
  prevMillis=millis();
    }
  grade(); texto(); marker();
  }
}
//
void grade(){
  stroke(0, 255, 0); strokeWeight (0.5);
  for(int i=0; i<=9; i++) line(50*i,0,50*i,400);
  for(int i=0; i<=7; i++) line(0,50*i,500,50*i);
  }
void texto(){
  fill(0,255,0); text("bpm", 473,395);
  for(int i=50; i <= 140; i=i+10)
  text(i-20, -247+i*5, 395);
  text("% rel", 470,365);
  for(float i=0; i <= 8; i+=2)
  text(int(i*12.5), 480, 415-i*50);
  }
void marker() {
  if(millis() - prevMillis >= 500){
  fill(255,0,0);
  rect(18,18,15,15);
  }
}
```

A função *draw( )* é equivalente à função *loop( )* na linguagem do Arduino, os comandos postos aqui são repetidamente executados. O comando de iteração *while( )* verifica se existe na porta serial um dado disponível; se sim as funções *stroke( )* e *strokeWeight( )* definem a cor e a espessura das linhas dos objetos desenhados, respectivamente, e a variável *sensor* vai guardar o valor lido na porta serial com seu máximo expandido para a largura da tela com a função *map( )*. A função *map ( )* converte uma faixa de valores para outra faixa. Sua sintaxe é a seguinte:

```
  map(valor, min1, max1, min2, maz2)
```

O primeiro parametro **valor** é a variável que será convertida; o segundo e o terceiro parametros são os valores mínimo e máximo dessa variável; o quarto e o quinto são os novos valores mínimo e máximo da variavel *valor*.

O sinal da saída do amplificador e agora digitalizado será mostrado com a função *rect( )* na pequena tela preta na forma de uma barra vertical vermelha com sua altura determinada pela variável *sensor* e a sua posição no eixo das abscissas definida pela variável *bpm*. A cada segundo a variável *bpm* é atualizada com o numero de batimentos por minuto acumulado na variável *counter*, que é incrementada toda vez que o nível do sinal de entrada é igual ou maior que aquele apontado pela variável *trigger*. As funções *grade( )*, *texto( )* e *pulser( )* são declaradas no final da listagem e são chamadas para desenhar as linhas de grade, posicionar textos e um marcador de segundos no canto superior esquerdo na tela.

A tela do instrumento de 500 x 400 *pixels* é dividida em 10 colunas verticais e oito horizontais, como a tela de um osciloscópio, com a função *line( )*. No eixo **x** é marcada a faixa de batimentos por minuto de 30 a 120 bpm e no eixo **y** o nível relativo de 0 a 100%, com a função *text( )*. As funções *rect( )* e *line( )* já foram explicadas no capítulo 6.

A altura da barra vermelha dá uma indicação em tempo real da amplitude do sinal na entrada do amplificador com o sensor ótico. A posição dessa barra no eixo x dá uma indicação da frequencia de batimentos cardíacos do experimentador.

Carregue e execute esse *sketch* em *Processing* no PDE e coloque seu dedo indicador sobre o sensor ótico do monitor de batimentos cardíacos. Comece girando o potenciometro no *hardware* do monitor para o mínimo ganho e observe a barra vermelha à esquerda da tela. Agora vá muito suavemente aumentando o ganho até que a barra vermelha comece a pulsar, porem sem se deslocar ainda da esquerdada tela. Veja até que nível relativo o pico do sinal alcança e tente controlá-lo com o potenciometro. Aumente mais o ganho do circuito até que a barra comece a se deslocar sobre o eixo horizontal onde estão marcados as frequencias de batimentos cardíacos. Esse é o momento para você ajustar seu dedo no sensor para um sinal estável de cerca de um pouco mais de um pulso por segundo. A melhor leitura é conseguida quando a barra vermelha dá vários repiques abaixo do nível 25 e uma só incursão longa até 75 nesse tempo. Tome como referência de tempo o pequeno retangulo vermelho no canto superior esquerdo da tela que pisca a cada segundo, que equivale a 60 batimentos por minuto. A sensibilidade do sistema pode ser ajustada tambem mudando o valor da variável *trigger* no código *Processing*; valores proximos de 25 aumentam muito a sensibilidade.

Em suas experiências como esse instrumento de *biofeedback* você pode comprovar que a taxa de batimentos cardíacos é função da respiração; controlando a respiração você pode controlar seu rítmo cardíaco. Experimente tambem controlar a taxa de batimentos tentando mover mentalmente a barra vermelha mais para esquerda ou mais para a direita. Você pode tentar um experimento muito interessante que é controlar seus batimentos cardíacos de modo a sincronizar os picos da barra vermelha com a frequencia do marcador de segundos no canto superior esquerdo. Se você conseguir esse sincronismo seu coração estará dando uma batida por segundo. Você pode tambem mudar a frequencia desse marcador de 60 bpm para 70 ou 75 bpm e tentar se sincronizar com essa medida.

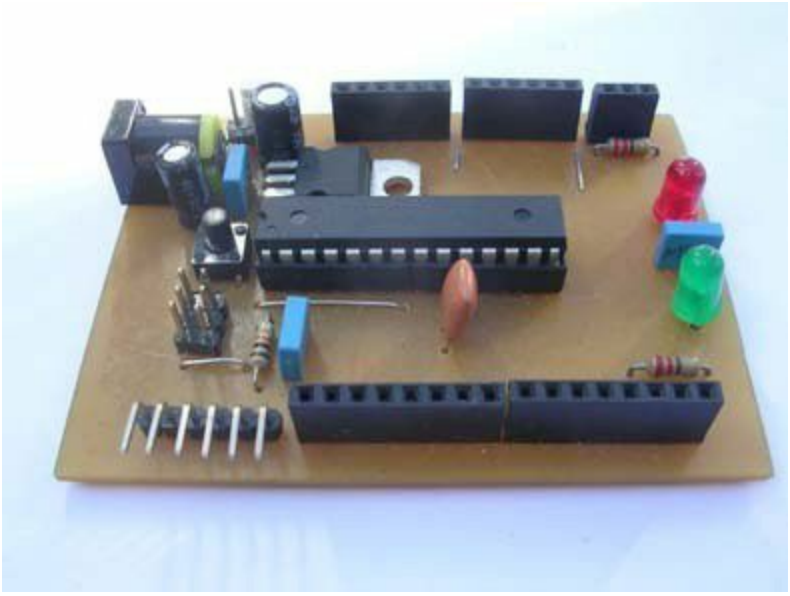
Apendices

1- Monte seu próprio Arduino

Introdução
O circuito proposto
Desc rição do Circuito
A placa de Circuito impresso
Alimentando o Arduino

2- Programador de *Bootloader*

Introdução
Gravando o *bootloader*
Conectando o Arduino a um Computador
descrição do circuito



Apendice 1

Monte seu próprio Arduino

Introdução

O Arduino é uma plataforma para criação de protótipos de Eletronica baseada no conceito de *software* e *hardware* livres, aqueles projetos que são criados para o domínio público e porisso podem ser copiados e modificados por outras pessoas conforme suas necessidades e depois podem ser colocados de volta ao domínio público de modo que outros usuários possam usufruir dessas mudanças em seus próprios projetos. O Arduino foi concebido segundo esse conceito para auxiliar artistas, projetistas e estudantes interessados na criação de objetos e ambientes interativos. Interação é a ação recíproca entre um usuário e um equipamento. Um objeto interativo é aquele capaz de dar respostas imediatas a comandos do usuário. As cenas em um video game são um bom exemplo de ambiente interativo. O Arduino pode servir de plataforma, por exemplo, para você construir um aparelho de biofeedback que mostre na tela do PC imagens que podem mudar de forma interativa com o seu estado mental.

Vimos no capítulo 2 que o Arduino é composto por duas partes principais: um *hardware*, um conjunto básico de componentes eletronicos montados numa placa de circuito impresso, que é a plataforma para o desenvolvimento de protótipos de Eletrônica; e um *software*, composto pelo *bootloader* e uma interface gráfica ou ambiente de desenvolvimento integrado (IDE – *Integrated Development Environment*), onde criamos os programas que vamos carregar no Arduino. São esses programas, ou *sketches*, que vão dizer ao *hardware* o que fazer.

O circuito proposto do Arduino

O leitor pode adquirir em sites de comercio eletronico tanto aqui no Brasil quanto lá fora um Arduino já montado e testado; existem muitas opções de modelos conforme o microcontrolador embarcado, desde o mais simples com o ATmega168 e comunicação RS-232 ou USB até aqueles com o ATmega1280 com mais memória de programa e mais entradas e saídas digitais. Mas tambem o leitor poderá montar seu proprio Arduino a partir de componentes facilmente encontrados em lojas de componentes eletronicos. Nesse caso propomos um circuito básico com o microcontrolador ATmega8 ou com o ATmega168, alguns resistores e capacitores, um regulador de tensão comum e um conjunto de conectores do tipo *mini-latch*. A placa de circuito impresso proposta é de face simples e poderá ser confeccionada por qualquer método caseiro, como o já consagrado método térmico a partir de uma impressão laser. A fonte de alimentação é externa à placa e pode ser qualquer carregador de baterias de aparelhos portáteis, como o de telefones celulares, que forneça tensões entre 9 e 15 volts CC. O circuito é o da figura 63.

Tanto o ATmega8 quanto o ATmega168 pode ser utilizado na montagem do nosso Arduino. Observe que no circuito não aparece o conversor TTL-RS232. Preferimos separar esse circuito da placa principal por uma boa razão: dar a opção ao montador de escolher que tipo de comunicação serial o seu Arduino terá com seu computador, se RS-232 ou USB. Para a comunicação RS-232 existe o conversor MAX232 que é um circuito integrado muito fácil de encontrar em lojas de componentes eletronicos, e por ser do tipo DIL de 16 pinos é tambem muito fácil de montar em uma pequena placa de circuito impresso e ser conectado por um cabo diretamente entre uma porta RS-232 do computador e o conector “SERIAL” que aparece no diagrama da figura 63.

Para a comunicação serial USB o conversor é o FT232RL, um circuito integrado do tipo SMD de 28 pinos, um pouco mais difícil de encontrar no comercio, bem mais caro que o MAX232 e de montagem mais complicada numa placa de circuito impresso caseira devido ao tamanho e espaçamento entre seus pinos. Mas tanto um tipo quanto o outro de conversor funciona normalmente com o Arduino. A grande vantagem da USB é que todo PC moderno tem várias dessas portas disponíveis, e só os de mesa, os desktop, ainda estão vindo com duas ou tres portas RS-232. No capítulo 3 mostraremos os dois circuitos conversores e a montagem dos cabos.

Descrição do Circuito

O microcontrolador ATmega328 é alimentado com 5 volts nos pinos 7 e 20 provenientes do regulador de tensão integrado LM7805, se o jumper JMP1 estiver na posição EXT. Os capacitores C1 a C4 formam os filtros CC de entrada e de saída do regulador. No conector J1 entramos com uma tensão de 9 volts a 15 volts de uma fonte externa ou de um carregador comum de baterias com o positivo no pino central. O diodo D1 serve de proteção para o circuito caso a entrada de alimentação esteja invertida. O LED1, vermelho, acende se a tensão de alimentação do circuito estiver correta. Essa mesma tensão externa tambem é disponibilizada no primeiro pino do conector POWER. A

tensão de 5 volts do regulador LM7805 alem de ser disponibilizada nos pinos dos conectores POWER e SENSOR segue tambem para o conector SERIAL para alimentar a placa externa com o conversor RS-232. À direitado diagrama temos o microcontrolador e os conectores ICSP, os dois digitais e o analógico.

São 14 pinos digitais (D0 a D13) e 6 analógicos (A0 a A5 ou 14 a 19). O pino de entrada de referência (AREF) para o conversor A/D do Arduino fica no segundo conector digital. O LED de teste e o pino 3 do ICSP estão ligados no pino digital D13. Na área tracejada vemos um cristal de 16Mhz e dois capacitores de 22pF que formam o circuito de relógio para o microcontrolador. Esse conjunto cristal-capacitores pode ser substituido por um ressonador cerâmico de 16Mhz que já inclui os dois capacitores no mesmo encapsulamento. O microcontrolador uma vez configurado pelo *bootloader* do Arduino é resetado pelo último pino do PORT C (pino 1do microcontrolador), onde tambem temos um botão de RESET. Esse mesmo sinal de RESET segue para o pino 5 do ICSP e, atraves de um capacitor de 100nF, para o circuito conversor serial externo, para onde vão tambem os pinos digitais D0 e D1 que normalmente são reservados para a comunicação serial do Arduino com um computador ou com outro Arduino. Nesse conector SERIAL pode ser ligado um circuito conversor RS-232 ou um USB. Se for usado um conversor USB o *jumper* JMP1 deve ser mudado para a posição USB e com isso o Arduino passa a ser alimentado pela tensão 5 volts da porta USB do computador. É interessante observar que se não dispusermos de um carregador de baterias para conectarmos em J1 nosso Arduino pode ser alimentado por uma fonte externa de 5 volts diretamente atraves do conector POWER.

Figura 64:

Figura 65:

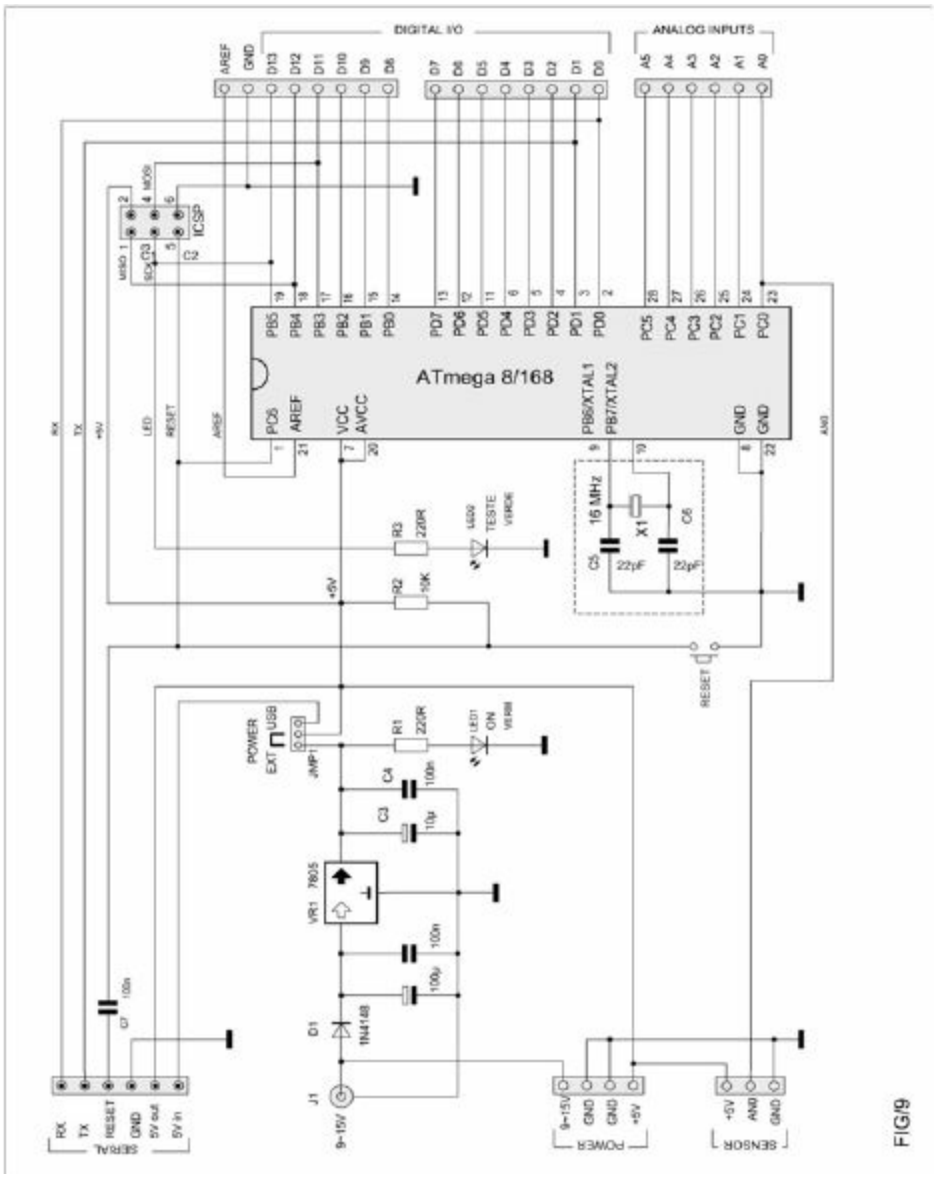


Figura 65: Circuito proposto para montagem do Arduino

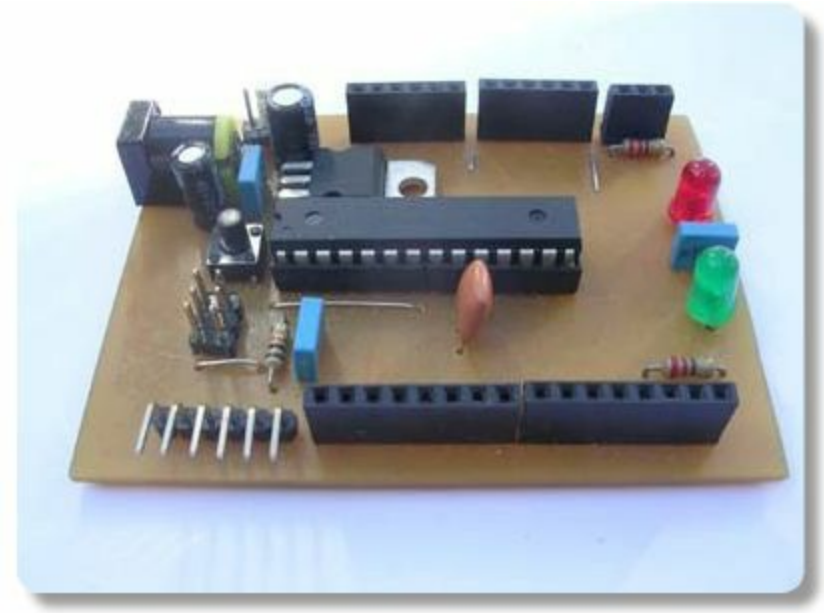


Figura 66: Protótipo do Arduino montado

Lista de componentes			
Item	Componente	Qtde	Obs.
1	Soquete DIL 28 pinos	01	Ou 2x DIL de 14 pinos
2	ATmega8	01	ou ATmega168
3	Cristal 16 Mhz	01	Ou ressonador cerâmico 16 Mhz
4	Regulador LM-7805	01	Não precisa dissipador
5	Diodo 1N4148	01	Qualquer um da série 1N414x
6	LED 5 mm vermelho	01	
7	LED 5 mm verde	01	
8	Capacitor 100uFx25V	01	Tipo axial
9	Capacitor 10uFx25V	01	Tipo axial
10	Capacitor 100nFx100V	03	Tipo poliester
11	Capacitor 22pFx100V	02	Dispensados se usar ressonador
12	Resistor 220 ohms x 1/8W	2	Tipo cerâmico
13	Resistor 10 Kohms x 1/8W	1	
14	Botão tipo	1	Micro-chave para circuito

	campanhia		impresso
15	Jumper 3 pinos	1	Jumper com espaçamento 2,54 mm
16	Mini-latch 8 pinos fêmea	2	Para os pinos digitais
17	Mini-latch 6 pinos fêmea	1	Para os pinos analógicos
18	Mini-latch 6 pinos fêmea	1	Para os pinos de tensões externas
19	Mini-latch 6 pinos 90 macho	1	Para a placa serial
20	Conector xxx 3 pinos fêmea	1	Para a entrada de sensor
21	Conector xxx 6 pinos fêmea	1	Para o ICSP
22	Conector jack	1	Tipo para circuito impresso

Na figura 64 podemos ver como ficou nosso Arduino depois de montado. O microcontrolador aparece no meio da placa. Todos os pinos digitais e analógicos obedecem as mesmas posições e distanciamentos do Arduino padrão. Na parte de cima ficam os conectores digitais e à direita destes o conector SERIAL que acrescentamos para ligarmos a placa do conversor serial que será montada à parte. Na parte de baixo da placa está o *jumper* para selecionarmos que interface serial estamos usando, se RS-232 ou USB, e o conector analógico e o de tensões disponíveis para outros circuitos externos. À esquerda deste acrescentamos tambem um conector de tres pinos, o SENSOR, que é uma extensão do pino analógico A0, a tensão de 5 volts e o terra num só conector. À direita da placa, temos o botão de RESET e acima dele o conector ICSP. Mais abaixo do RESET o conector de entrada de tensão para alimentar o Arduino. Os outros componentes são os capacitores e resistores, o regulador de tensão e os dois LEDs. O LED vermelho acende para indicar que o Arduino está corretamente alimentado e o LED verde é o que vai conectado ao pino digital D13 do Arduino e indica quando carregamos programas para o microcontrolador. Em nosso Arduino utilizamos um ressonador cerâmico de 16 Mhz em vez do cristal e dois capacitores de 22pF que aparecem no diagrama. Nossa placa tem somente quatro pequenos *jumpers*.

A placa de Circuito Impresso

A placa de circuito impresso do nosso Arduino tem 7,5 cm por 5,5 cm e é de face simples. Veja na figura 65 o lado da solda e na figura 66 uma sugestão de serigrafia no lado dos componentes na placa. A placa de circuito impresso pode ser confeccionada por qualquer método caseiro, como o já consagrado método térmico com impressão ou fotocópia a laser numa folha de papel *grassy*. Nesse caso deve-se usar o desenho da figura 67, sem inversão do lado da solda. Note que em nossa placa utilizamos um ressonador cerâmico na posição indicada como XTAL, se for utilizado um cristal e os dois capacitores ceramicos de 22 pF o desenho da placa deverá ser modificado para acomodá-los.
Os pinos 8 e 22 (GND) do microcontrolador já vêm conectados entre si dentro do encapsulamento e portanto no desenho da placa não aparecem conectados.

Uma vez pronta a placa de circuito impresso, o nosso Arduino pode ser todo montado em umas poucas horas, dependendo da experiência em montagens eletronicas
do leitor. As unicas recomendações são para conferir mais de uma vez a montagem dos componentes polarizados, como o diodo, os capacitores eletrolíticos e os LEDs, e para se certificar que o microcontrolador está corretamente alojado e orientado no seu soquete. Confira tambem se os quatro *jumpers* (são quatro pedaços de fios finos) foram corretamente soldados nos seus respectivos lugares. Por fim, coloque o strap de JMP1 na posição EXT.

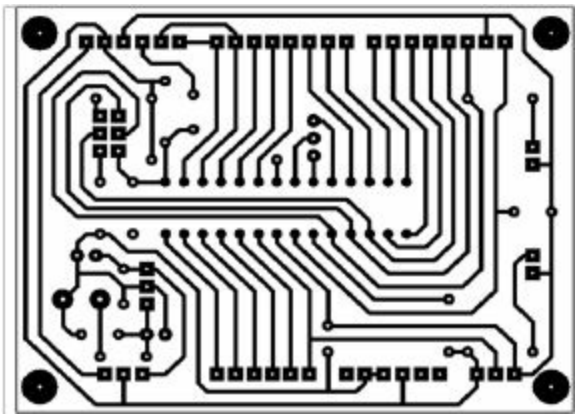


Figura 67: Placa de CI lado da Solda (espelhado)

Alimentando o Arduino

Com o *hardware* do Arduino montado já podemos alimentá-lo, embora ainda não poderemos controlar nada com ele pois não temos nenhum programa carregado. Você pode alimentar o Arduino de duas maneiras diferentes: ou com uma fonte externa ou um carregador para aparelhos portáteis com tensões entre 9V e 15V, ou diretamente com uma bateria comum de 9 volts com um cabo adaptado para o plug da bateria e para o plug do módulo, conforme a figura 62. O positivo da fonte deverá ser o pino central do conector macho que vai ligado ao Arduino. Ao conectar qualquer dessas fontes de tensão ao Arduino somente o LED vermelho deverá acender de imediato indicando que o módulo está alimentado corretamente.

No apendice 2 vamos montar o cabo paralelo e programar o *bootloader* no Arduino. Vamos tambem montar o cabo serial e fazer nossos primeiros experimentos com o nosso Arduino.

Figura 69: Placa de circuito impresso lado da solda (sem inversão)

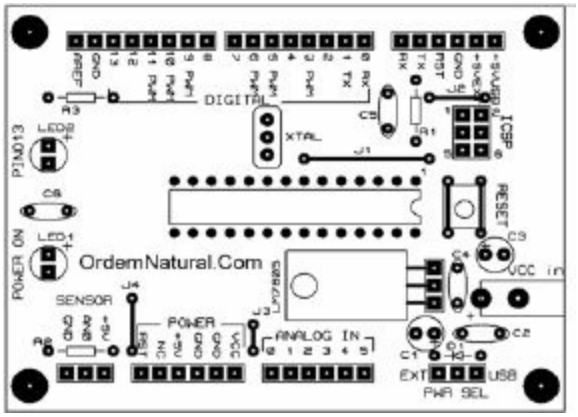
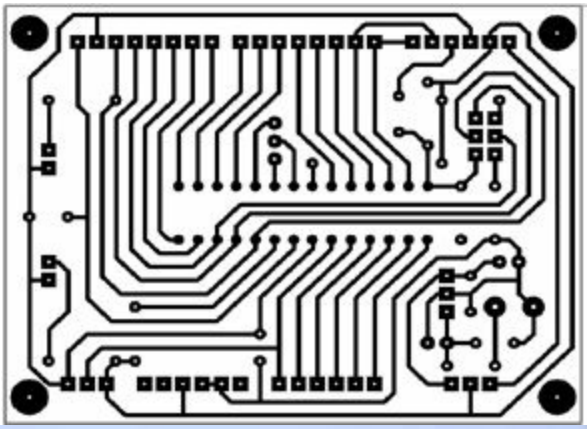
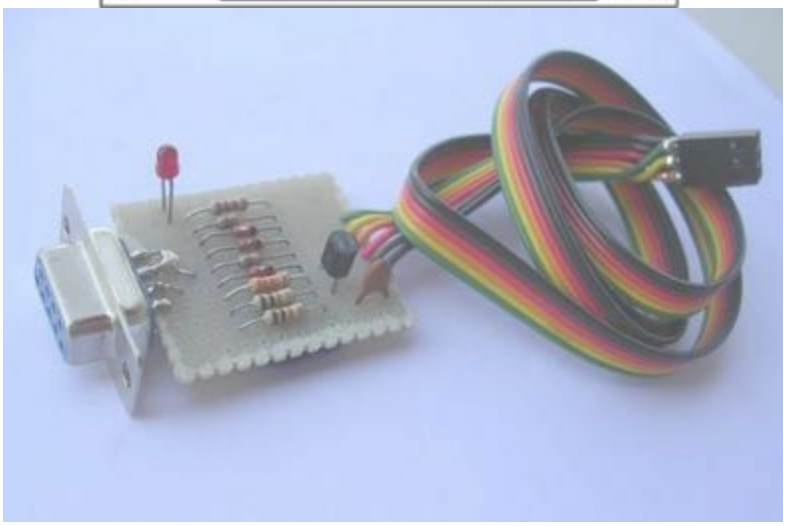


Figura 68: Serigrafia da placa de CI





Apendice 2

Montagem do programador de *bootloader*

Introdução

No capítulo 1 vimos que um microcontrolador é constituido basicamente por uma CPU, memória de programa, memória de dados e controles para entrada e saída de dados. Mas podemos encontrar tambem microcontroladores que já vêm com memória EEPROM, uma ou mais USARTs, conversor A/D com várias entradas, comparadores de tensão e outros circuitos de comunicação e controles mais complexos com o mundo externo. É na memória de programa, do tipo *flash*, onde ficam gravadas as instruções que a CPU vai buscar para executar as ordens que o projetista definiu para o sistema microcontrolado. Esse tipo de memória pode ser regravada quando é necessário corrigir ou redefinir algumas funções que esse sistema controla. Essa gravação (e regravação) da memória *flash* é realizada por um circuito programador externo conectado entre o sistema onde o microcontrolador está embarcado e um computador, onde roda um aplicativo de programação. Esse processo, conhecido como ICSP (*In-Circuit Serial Programming*), ou **programação serial no circuito** é feito de forma serial e não é necessário retirar o microcontrolador de seu circuito. É para isso que serve o conector **ICSP** no Arduino.

Existem dois métodos para gravação *in-circuit* na memória *flash* dos microcontroladores usando um computador: via porta serial RS-232 ou USB e via porta paralela. Nesse capítulo vamos mostrar como montar um programador simples que utiliza a porta serial RS-232 e um *software* livre e que pode ser baixado da *web*, o **PonyProg2000**. O circuito pode ser visto na figura 68 e a sua montagem na figura 69.

Para programar um microcontrolador da família AVR no circuito não é necessário nenhuma outra fonte de tensão externa. Toda a programação e verificação é realizada atraves de tres pinos do Port B do microcontrolador com a própria fonte de alimentação de 5 volts; diferentemente, os microcontroladores PIC da Microchip exigem uma tensão especial de 13,4 volts. Basta um computador com uma saída serial RS-232 e alguns resistores, tres diodos zener de 5,1 volts e um só transistor NPN de uso geral para se montar o programador. São necessárias uma linha de sincronismo de relógio (*clock*) e uma linha de entrada e outra de saída serial dos bits que vão ser gravados na

memória *flash*. Essas linhas devem estar disponíveis, junto com as linhas de reset, de 5 volts e de terra, no conector padronizado ICSP de seis pinos.

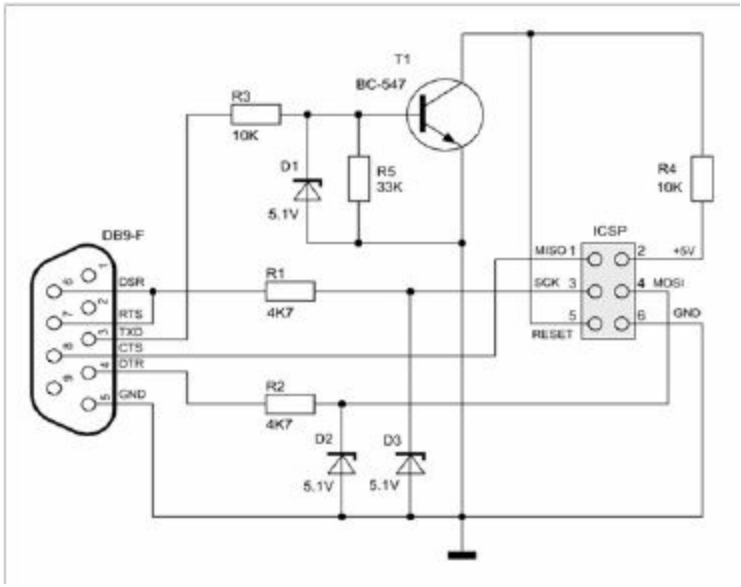
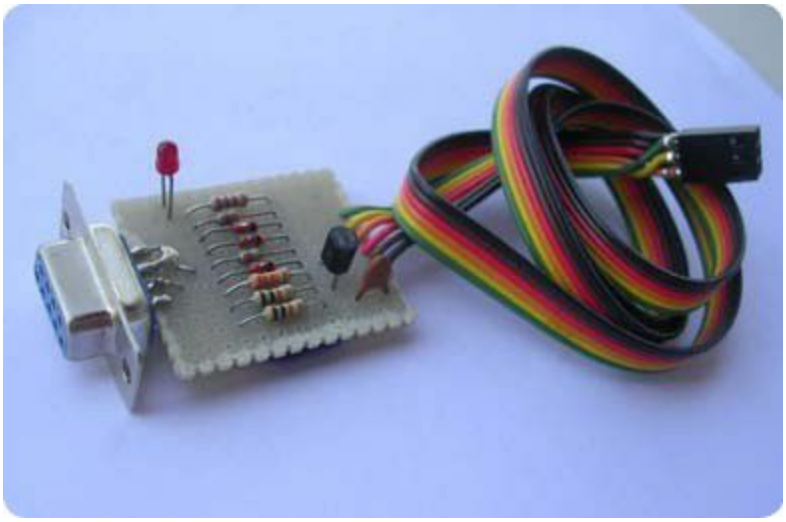


Figura 70: Circuito do programador serial

Figura 71: O programador montado

Na programação o *software* de aplicação, no nosso caso o *PonyProg2000*, assume o controle do microcontrolador atraves do pino de *reset*, mantendo-o ativo (fixo em 0 volt) fazendo a saída TxD, pino 3 da porta RS-232, positiva e saturando o transistor NPN. Os diodos zener formam circuitos *clamp* que protegem o microcontrolador e o transistor contra tensões maiores que 5 volts da interface RS-232. Nesse momento o microcontrolador entra e permanece no modo programação com todos os seus pinos de entrada e saída em alta impedancia (modo entrada de dados). Nesse momento o *PonyProg2000* é o mestre (*Master*) e o microcontrolador o escravo (*Slave*). O mestre fornece o sinal de sincronismo pelo pino SCK (*Serial ClocK*), pino 3 do ICSP, e a cada pulso de *clock* nesse pino é transferido um bit para a memória *flash* pelo pino MOSI (*Master Out – Slave In*), pino 4; imediatamente esse *bit* é retornado da memória pelo pino MISO (*Master In – Slave Out*), pino 1, para verificação.

Uma vez montado o circuito no cabo de programação vamos testá-lo no nosso Arduino. Primeiramente faça o *download* do programa *PonyProg2000* , no endereço **http://www.lancos.com/prog.html**. O arquivo compactado tem pouco mais de 500 *kilobytes* e pode ser descarregado em poucos segundos. Descompacte o arquivo descarregado na sua área de trabalho e execute o *setup.exe* normalmente como qualquer executável *Windows* e o instale na pasta sugerida no **driver C:**.



Esse é o primeiro programa de uma série que você vai precisar para desenvolver projetos com o Arduino, porisso é interessante criar uma pasta própria com o nome “Arduino” na sua área de trabalho para guardar arquivos especiais necessários e os atalhos dos programas que vai usar. Então crie um atalho do executável **PONYPROG2000.EXE** na área de trabalho do seu PC e o mova para a pasta “Arduino”. Execute o programa dando um clique duplo no atalho, ou diretamente em *PONYPROG2000.EXE* na pasta Arquivos de programas.

Dê *OK* nas tres pequenas janelas que surgirem, a tela principal do programa é o da figura 70. Agora tudo o que temos que fazer é configurar o *PonyProg2000* para o tipo de programador AVR que vamos usar, o que é muitíssimo simples. Na barra de menus do programa entre em **Setup** e selecione **Interface Setup**.

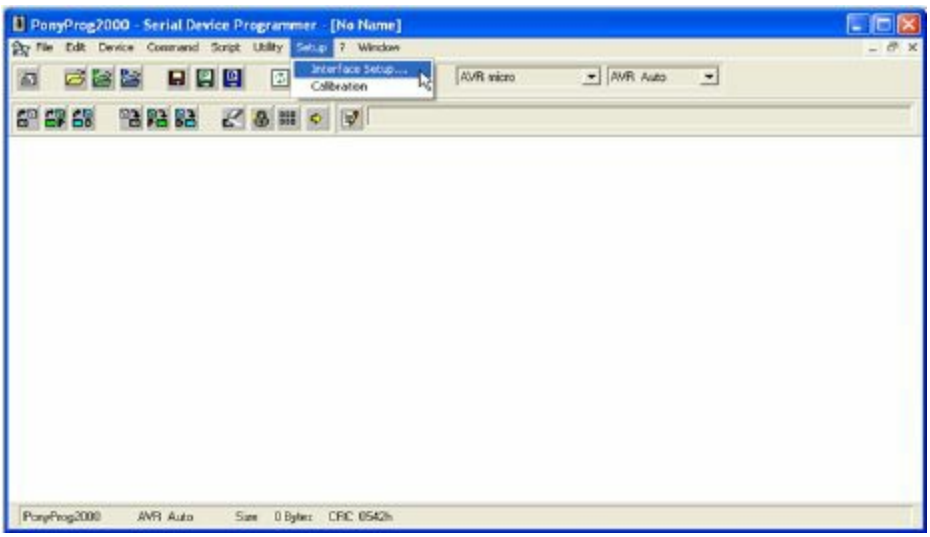


Figura 72: Tela principal do PonyProg2000

Na janela que surgir marque a opção **Serial** e a porta serial que vai usar, no exemplo foi a COM1, e selecione **SI Prog I/O** na caixa combo da esquerda, deixando todas as opções de inversão das linhas de controle sem marcação, como na figura 71. Clique em OK. De volta à janela principal, no topo da tela selecione **AVR micro** na caixa combo da esquerda **ATmega168**.
Pronto! O *PonyProg2000* já está configurado para ser usado com o nosso programador serial. Deixe o *PonyProg2000* aberto na tela do seu PC e insira um microcontrolador ATmega168 novo no soquete

de 28 pinos no Arduino. Agora interligue o conector fêmea ICSP de 6 pinos do programador no seu Arduino, que deve estar sem alimentação, observando com cuidado a correta correspondência entre os pinos. Conecte o circuito com o conector fêmea DB-9 do programador no computador PC e alimente o Arduino. Os leds vermelhos no Arduino e no programador deverão acender. De volta à tela principal do *PonyProg2000* clique no botão **Read Device**, o primeiro à esquerda na segunda linha de botões. Uma pequena janela de *Status* deve surgir indicando o progresso da leitura de toda a memória *flash* do ATmega168, veja a figura 72; após alguns segundos uma outra janela sinaliza o fim do processo com exito; veja a figura 73.

Figura 73: Configurando o PonyProg2000

Dê *OK* nessa pequena janela e observe que toda a área de visualização de dados no *PonyProg2000* mostra somente **FF** até a última posição de memória do ATmega168 novo.

Gravando o *bootloader*

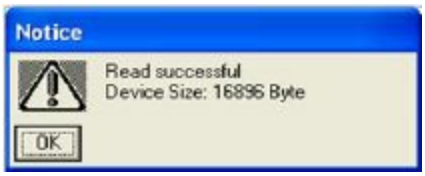
Agora precisamos testar nosso circuito gravando um programa na memória *flash* do microcontrolador. Vamos fazer isso gravando o *bootloader* do Arduino. No capítulo 1 vimos que o *bootloader* é um pequeno programa que fica residente no *bootblock*, uma pequena parte da memória *flash* do microcontrolador. Toda vez que o Arduino é resetado esse *bootloader* é automaticamente executado para verificar se existe algum pacote de dados novos disponíveis na porta serial sendo enviado pelo computador PC. Se existe, o *bootloader* recebe esse pacote de dados e o grava na memoria *flash* antes do *bootblock*. No ATmega168 o *bootloader* do Arduino ocupa cerca de 2k dos 16kbytes de sua memória *flash*. No site oficial do Arduino existe muitas versões de *bootloader* disponiveis para baixar para todos os microcontroladores AVR usados no Arduino. Em nosso projeto vamos usar o *bootloader* **ATmegaBOOT_168_diecimila.hex** que já está, entre outros arquivos, no seu computador na pasta **Arduino-0021/hardware/arduino/bootloaders/atmega**. Se você abrir esse arquivo com o Bloco de Notas do *Windows* vai ver a listagem de caracteres no formato **Intel** hexadecimal, que é o padrão mundialmente adotado para gravaçãoem memórias e microcontroladores.

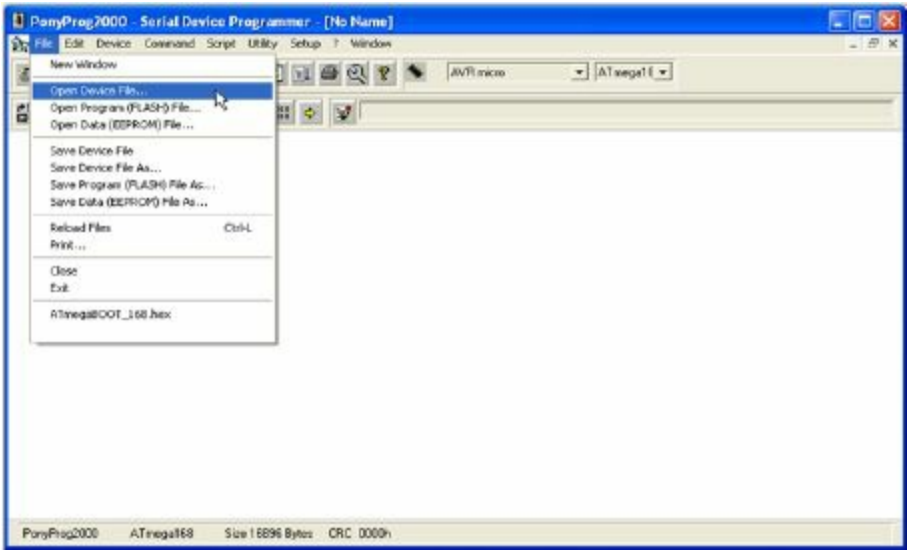
Figura 74: Janela de status

Figura 75: janela de notificação

Temos agora tudo pronto para gravarmos o *bootloader* do Arduino no ATmega168. Encaixe o conector DB-9 fêmea do programador que voce montou no correspondente conector macho no seu computador PC. Sem alimentar ainda o Arduino, identifique o pino 1 no conector ICSP fêmea na ponta do cabo do programador e, identificando tambem o pino 1 do conector ICSP macho na placa, encaixe o cabo com cuidado na placa do Arduino. Agora alimente o Arduino conectando uma fonte externa de 9 a 15 volts CC. Verifique se o LED vermelho na placa acendeu.

Figura 76: Tela do PonyProg2000





No *PonyProg2000*, na barra de menus do programa abra a opção **File** e selecione **Open Device File...** , figura 74, e clique e selecione o arquivo **.hex** que está na pasta indicada acima. O *bootloader* do Arduino começa no endereço **3800** e para visualizá-lo puxe a barra de rolagem à direita da janela do programa até esse endereço e observe que são os mesmos dados você visualizou com o Bloco de Notas. Toda a área da memória *flash* acima do *bootblock* está vazia e será utilizada pelo *bootloader* para gravar os programas que você ainda vai escrever. Veja a figura 76.

Figura 77: Abrindo o arquivo .hex

Para gravar o *bootloader* no ATmega168 clique no botão **Write Device**, o quinto na segunda linha de botões, e dê OK na janela que surgir e espere por alguns segundos o processo terminar observando a barra de status de escrita, figura 77. Após a escrita na memória o *PonyProg2000* vai verificar se o arquivo *.hex* foi transferido sem nenhum erro. Para verificar se o *bootloader* foi gravado clique no botão **Read Device** no *PonyProg2000* e, ao final da leitura, puxe a barra de rolagem de novo até o endereço 3800. A partir desse endereço hexadecimal deve aparecer o codigo do *bootloader* do Arduino que foi gravado no microcontrolador.
Retire a alimentação do Arduino e desconecte o programador. Finalmente, reconecte a alimentação do Arduino e observe se o LED vermelho está aceso e se o LED verde começa a piscar com intervalos de um segundo. Se isso aconteceu, seu Arduino está vivo! Parabéns!! Agora você tem uma grande ferramenta para criar seus projetos com microcontroladores AVR. Agora é hora de fazer o Arduino "conversar" com o seu computador.

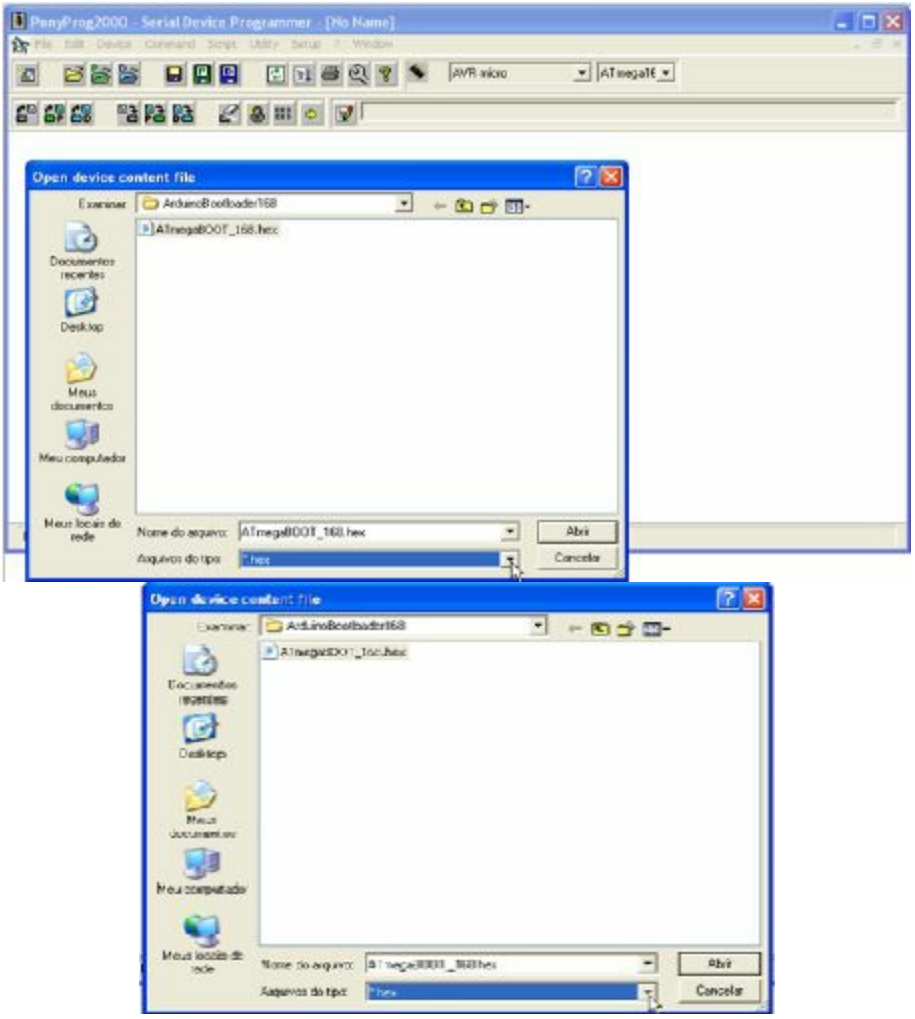


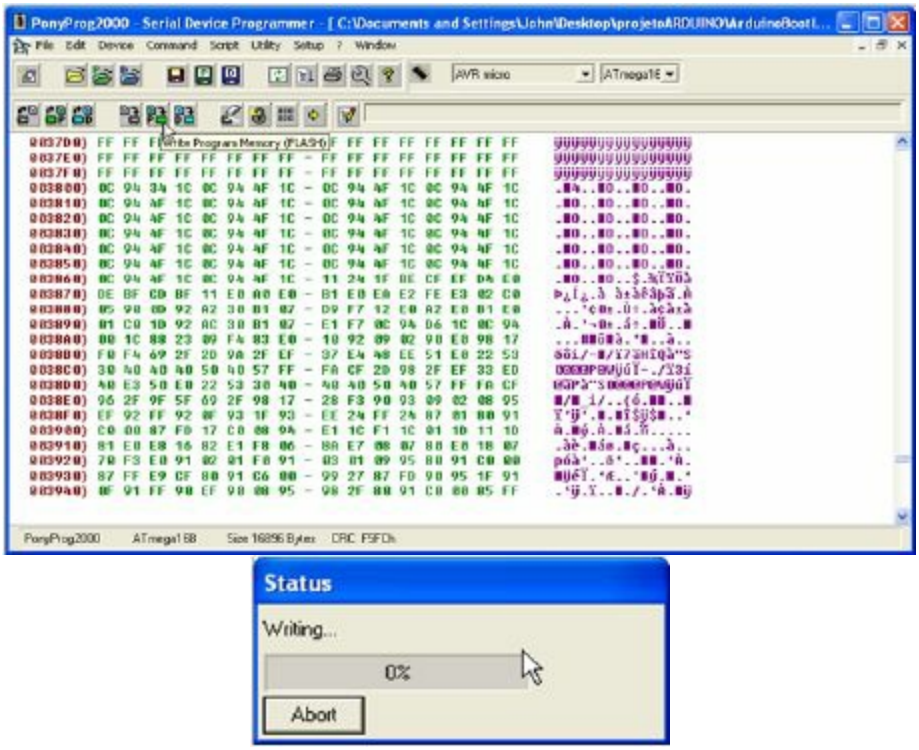
Figura 78: Visualizando o conteúdo da memória flash

Conectando o Arduino a um computador

O Arduino se comunica com um computador PC de forma serial, via porta USB ou RS-232. No nosso projeto utilizaremos a segunda forma de comunicação por ser esta a mais fácil de implementar e de se conseguir os componentes necessários no mercado para montar o circuito de interface. Utilizaremos o circuito integrado DIL (*Dual In-Line*) de 16 pinos MAX232N e alguns capacitores para montar um circuito conversor de níveis TTL do Arduino para os níveis de tensão compatíveis com o protocolo RS-232 do PC.

Figura 79: Gravando no ATmega328

Para se comunicar com o PC o Arduino utiliza somente quatro linhas: as linhas TX, RX, *Reset* e a linha comum de terra. A linha TX é o pino digital 1 do Arduino; o estado logico nesse pino após conversão de nível será enviado pelo pino 2 (RxD) da porta RS232 para o PC. A linha RX é o pino digital 0; o estado logico nesse pino será determinado pelo PC atraves do pino 3 (TxD) da porta RS232. A linha de Reset é linha de controle do PC sobre o Arduino que na porta RS232 é o pino 4 (DTR). Veja o circuito do conversor na figura 78.



A porta USB requer o conversor FT232RL, um circuito integrado no formato SMT (*Surface Mount Technology*) de 28 pinos, e a instalação de um **driver**, um programa

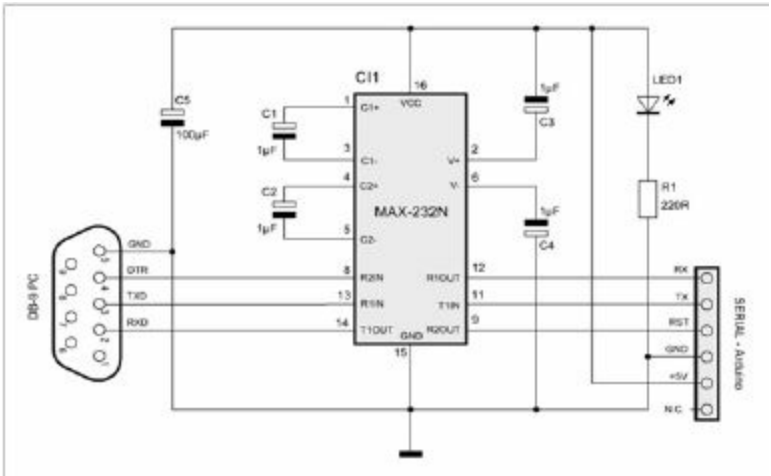
Figura 80: Circuito do conversor TTL-RS232

especial para fazer a interação do novo *hardware* com a versão *Windows* do seu computador. Para a interface serial RS-232 não é preciso instalar nenhum *driver*. Todo o circuito e o conector DB-9 fêmea que vai ao PC poderão ser montados numa pequena caixa plástica, de onde sai um cabo com um conector fêmea na outra ponta para interligar ao Arduino. Podemos ainda acrescentar um LED vermelho de 3 mm ao circuito para indicar que este está alimentado. Veja afigura 80.

Descrição do Circuito

Os capacitores C3 e C4 não estão desenhados incorretamente no circuito, como a princípio se pode pensar, eles são mesmo invertidos nesse tipo de circuito conversor de tensão, conhecido como **charge pump**.

O programador serial com um transistor que montamos antes serviu somente para gravarmos o *bootloader* no ATmega168, e poderá ser útil tambem para programarmos diretamente outros microcontroladores da família AVR em outros projetos fora do Arduino. O conversor serial com o MAX232N servirá para programarmos diretamente no Arduino os programas que ainda vamos escrever. Uma vez montado o conversor serial podemos conectá-lo no Arduino no mesmo conector RS232 do PC que antes ligamos o programador.



Alimente o Arduino e veja se o LED vermelho no conversor acendeu. Você pode polarizar o conector que vai ao Arduino inserindo uma pequena peça de plástico ou outro material no orifício correspondente ao pino N.C. (não conectado). Fazendo isso esse conector fêmea não poderá ser encaixado invertido no conector macho na placa do Arduino.

O leitor poderá testar o cabo se, no conector fêmea que vai ao Arduino, puder entrar com +5 volts e terra e fazer um **loop-back**, ou seja, curto-circuitar os pinos TX e RX. Depois é só conectar a outra ponta com o DB-9 no seu PC e rodar o programa do **Windows Hyper Terminal**, que está em **Todos os programas > Acessórios > Comunicações**. Configure o *Hyper Terminal* com a porta serial COM1, escolha uma velocidade, mantenha o padrão **8N1** e em Controle de fluxo selecione **Nenhum**, como na figura 80.

Dê *OK* e entre com qualquer caracter ou frase pelo teclado e veja se retorna na janela do Terminal. Se sim, a transmissão e a recepção do seu cabo de comunicação serial com o Arduino está funcionando perfeitamente. Agora o leitor tem tudo pronto para começar suas primeiras experiências com o Arduino.

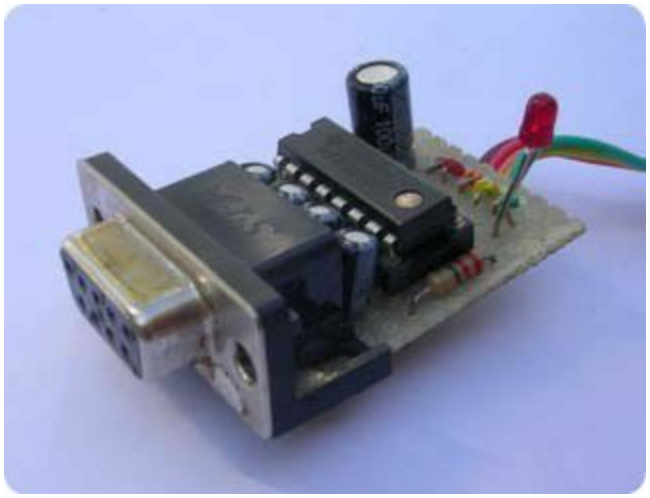


Figura 82: Configurando o HyperTerminal

Índice Remissivo

A

Acoplador ótico 110
alto/baixo 51
ALU 19
ANALOG 71, 75
analogRead 54
analogRead( ) 75, 164
analogWrite 54
Apple 28, 71
arc( ) 132
Arduino 20, 27, 56, 68, 92, 148, 156
argumento 52
ASCII 58, 128, 161
ATmega8 19
ATmega328 20, 24, 72, 76, 143

ATmega1280 28, 171
attachInterrupt( ) 100, 160
available( ) 145
AVR 18, 56, 143

B

background( ) 128, 165
barramento 18
biblioteca 52
biblioteca de comunicação serial 127
biofeedback 157, 167
boot 23
bootblock 23
bootloader 22, 28, 30, 37, 171, 182
buzzer 73
byte 144, 164

C

character 52
chip 24
ciclo de trabalho 78
código fonte 71
comando condicional 60
compilação 71
compilador 23
Computação Física 27
computador 53, 71
comunicação serial 30, 144
constantes 51
contador 33
Contador de Programa 20, 21
contadores 21
conversor A/D 22, 76, 172
conversor D/A 34
CPU 17

D

DDR 31
delay 41, 54
delay( ) 77
delayMicroseconds 74
Diecimila 29

digitalRead( ) 54, 71, 152
digitalWrite( ) 41, 54, 71, 74, 152
diretivas do preprocessador 72
domínio público 28
do ... while 63
draw( ) 128
Duemilanove 39
duty cycle 78

E

EEPROM 22
elemento piezoelétrico 115
ellipse( ) 128, 131, 132
else 61, 72
entrada e saída 17

F

fill( ) 128
Flash 23
float 52
floppy disk 88
for 63
Fotodiodos 113
Fototransistores 113
Funções 52

H

hardware 28, 68, 150, 170
Harvard 20
Hello World 58
HIGH/LOW 51

I

ICSP 29, 172, 174
IDE 23, 28, 38, 125, 171
if 60, 72, 81
inguagem de máquina 23
integer 52
interface gráfica 38

J

jumper 174

L

LCD 53, 104
LDR 80, 112, 159
LED 55, 70, 90, 151
library 52
Light Dependent Resistor 112
line( ) 130
linguagem C 23, 50, 61
linguagem de programação 17
Linguagem Processing 51, 123, 143, 164
Linux 28
LiquidCrystal( ) 106
long 52
loop 52, 61, 71, 74, 75

M

map( ) 166
matriz 2D 95
matriz de contatos 71, 155
Matrizes 64
memória 17
memória de programa 17
memória flash 21, 180
microcontrolador 13, 17, 20, 174
millis 54, 57
Modulação por Largura de Pulso 78
monitor serial 125
Mostrador de 7 segmentos 90, 94
motor 77
motores de passo 87
Motores elétricos 81
multiplexador 33

N

noise( ) 137

O

open source 28
operações lógicas 18
operador de atribuição 71
operador de comparação 59, 71

operador de deslocamento à direita 164
operador de deslocamento à esquerda 164
Operadores 59
operadores aritméticos 59
Operadores compostos 60
Operadores lógicos 59
operador ternário 61, 72
operandos 59
OUTPUT/INPUT 51

P

parâmetro 53
PC 71
PDE 124, 167
período 78
PIC 56, 179
pinMode( ) 51, 54, 75
pixels 127
point( ) 129, 137
PORT 31
porta serial 39, 58, 144
Ports 22, 29, 31
POWER 71, 174
Preprocessador 72
Processing Development Environment 124
protoboard 71
pull-up 56
pulse( ) 74, 160
PulseIn( ) 101
Pulse Width Modulation 78
PWM 20, 22, 34, 56, 74

Q

quad( ) 131

R

random( ) 54, 92
read( ) 128, 145
rect( ) 131, 166
registrador de direção de dados 31
Registrador de Instruções 20
registradores 19

reset 156
ressonador cerâmico 21, 174, 176
RGB 90
RISC 18
RS-232 29, 71, 144
ruído Perlin 137

S

saída/entrada 51
sensores eletronicos 112
SERIAL 175
Serial.begin 54
Serial Monitor 58, 126
Serial.println 54
servo-motores 83
setup 52, 71
shield 151, 152
sin( ) 137
sistema operacional 38
size( ) 128, 129
sketch 28, 40, 71, 72, 152
smooth( ) 133
SMT 186
software 28, 56, 68, 170
Solenoide 77, 108
stroke( ) 166
strokeCap( ) 135
strokeJoin( ) 134
strokeWeight( ) 134, 166
Surface Mount Technology 186
switch...case 147

T

Temperatura 117
temporizadores 21
tone() 74
Transdutores Piezoelétricos 115
triangle( ) 131
TRUE/FALSE 51

U

USART 58

USB 31, 71, 144, 175

V

variável global 71
variável local 71
verdadeiro/falso 51, 61
vibra-motor 89
void 52

W

web 77
while 62
while( ) 119
Windows 28, 161