PROGRAMAÇÃO COM ARDUINO

COMEÇANDO COM SKETCHES

SIMON MONK

série tekne

bookman

O autor

Simon Monk é bacharel em cibernética e ciência da computação e doutor em engenharia de software. Desde os tempos de escola é um aficionado atuante de eletrônica. Ocasionalmente, publica artigos em revistas dedicadas à eletrônica. Ele também é o autor de *30 Arduino Projects for the Evil Genius* e *15 Dangerously Mad Projects for the Evil Genius*.

M745p Monk, Simon.
Programação com Arduino [recurso eletrônico] : começando com Sketches / Simon Monk ; tradução: Anatólio Laschuk. – Dados eletrônicos. – Porto Alegre : Bookman, 2013.

Editado também como livro impresso em 2013.
ISBN 978-85-8260-027-6

1. Programação de computadores. 2. Programação - Arduino. I. Título.

CDU 004.42

Catalogação na publicação: Ana Paula M. Magnus CRB 10/2052

SIMON MONK

PROGRAMAÇÃO COM ARDUINO

» COMEÇANDO COM SKETCHES

Tradução:

Anatólio Laschuk
Mestre em Ciência da Computação pela UFRGS
Professor aposentado pelo Departamento de Engenharia Elétrica da UFRGS

Versão impressa
desta obra: 2013

bookman

2013

Obra srcinalmente publicada sob o título
Programming Arduino: Getting Started with Sketches, 1st Edition
ISBN 0071784225 / 9780071784221

Gerente editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Editora: *Verônica de Abreu Amaral*

Capa e projeto gráfico: *Paola Manica*

Leitura final: *Susana de Azeredo Gonçalves*

Editoração eletrônica: *Techbooks*

Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre – RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

Unidade São Paulo
Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center
Vila Anastácio – 05095-035 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Aos meus filhos, Stephen e Matthew,
de um pai muito orgulhoso.

Agradecimentos

Sou grato à Linda por ter me proporcionado tempo, espaço e apoio para que eu pudesse escrever este livro e por ter tolerado as várias confusões que os meus projetos criaram pela casa.

Sou grato também ao Stephen e ao Matthew Monk por terem se interessado pelo o que o pai deles estava fazendo e pela ajuda que deram durante os trabalhos com os projetos.

Finalmente, eu gostaria de agradecer ao Roger Stewart, à Sapna Rastogi e a todos que estiveram envolvidos na produção deste livro. É um prazer trabalhar com uma equipe tão excelente.

Sumário

INTRODUÇÃO 1

Afinal, o que é o Arduino? 1
O que precisarei? 2
Usando este livro 2
Material de apoio 3

capítulo 1 *ESTE É O ARDUINO 5*

Microcontroladores 6
Placas de desenvolvimento 7
Um passeio por uma placa de Arduino 7
Fonte de alimentação 7
Conexões de alimentação elétrica 8
Entradas analógicas 8
Conexões digitais 9
Microcontrolador 9
Outros componentes 10
As srcens do Arduino 10
A família Arduino 11
Uno, Duemilanove e Diecimila 11
Mega 12
Nano 13
Bluetooth 14
Lilypad 15
Outras placas "oficiais" 15
Clones e variantes do Arduino 16
Conclusão 16

capítulo 2 *COMEÇANDO 17*

Ligando a alimentação elétrica 18
Instalando o software 18
Instalando o seu primeiro Sketch 18
O aplicativo Arduino 23
Conclusão 25

capítulo 3 *FUNDAMENTOS DE LINGUAGEM C 27*

Programando 28
O que é uma linguagem de programação? 29
Blink (pisca-pisca) – novamente! 33
Variáveis 35
Experimentos em C 36
Variáveis numéricas e aritméticas 38
Comandos 40
if 40
for 41
while 44
A diretiva #define 44
Conclusão 45

capítulo 4 *FUNÇÕES 47*

O que é uma função? 48
Parâmetros 49
Variáveis globais, locais e estáticas 50
Retornando valores 52
Outros tipos de variáveis 53
float 53
boolean 54
Outros tipos de dados 55
Estilo de codificação 56
Recuo 57
Abrindo chaves 57
Espaço em branco 58
Comentários 58
Conclusão 59

capítulo 5 *ARRAYS E STRINGS 61*

Arrays 62
SOS em código Morse usando arrays 65
Arrays do tipo string 65
Literais do tipo string 66
Variáveis do tipo string 66
Um tradutor de código Morse 67
Dados 67
Globais e setup 68
A função loop 69
A função flashSequence 71
A função flashDotOrDash 72
Juntando tudo 73
Conclusão 75

capítulo 6 *ENTRADA E SAÍDA 77*

Saídas digitais 78
Entradas digitais 80
Resistores de pull-up 82
Resistores internos de pull-up 84
Debouncing 85
Saídas analógicas 90
Entrada analógica 92
Conclusão 93

capítulo 7 *A BIBLIOTECA PADRÃO DO ARDUINO 95*

Números aleatórios 96
Funções matemáticas 98
Manipulação de bits 98
Entrada e saída avançadas 100
Geração de som 100
Alimentando registradores deslocadores 101
Interrupções 101
Conclusão 103

capítulo 8 *ARMAZENAMENTO DE DADOS 105*

Constantes 106
A diretiva PROGMEM 106
EEPROM 107
Armazenando um valor int em uma EEPROM 108
Armazenando um valor float em uma EEPROM (Union) 109
Armazenando uma string em uma EEPROM 110
Limpando os conteúdos de uma EEPROM 110
Compressão 111
Compressão de faixa 111
Conclusão 112

capítulo 9 *DISPLAYS LCD 113*

Uma placa USB de mensagens 115
Usando o display 117
Outras funções da biblioteca LCD 117
Conclusão 118

capítulo 10 *PROGRAMAÇÃO ETHERNET DO ARDUINO 119*

Shields de Ethernet 120
Comunicação com servidores de web 120
HTTP 120
HTML 121
O Arduino como servidor de web 122
Ajustando os pinos do Arduino através de uma rede 125
Conclusão 129

capítulo 11 *C++ E BIBLIOTECAS 131*

Orientação a objeto 132
Classes e métodos 132
Exemplo de biblioteca predefinida 132
Escrevendo bibliotecas 133
O arquivo de cabeçalho 133
O arquivo de implementação 134
Completando a sua biblioteca 135
Palavras-chaves 135
Exemplos 135
Conclusão 138

ÍNDICE 139

Introdução

As placas Arduino de interface oferecem uma tecnologia de baixo custo que pode ser usada facilmente para criar projetos baseados em microcontrolador. Com um pouco de eletrônica você pode fazer todo tipo de coisa usando o seu Arduino, desde controlar as lâmpadas de uma instalação de arte até gerenciar a potência elétrica fornecida por um sistema de energia solar.

Muitos livros baseados em projetos mostram como ligar coisas ao seu Arduino. Entre esses está *30 Arduino Projects for the Evil Genius*, deste mesmo autor. Contudo, neste livro, o objetivo maior é mostrar como programar o Arduino.

Este livro explica como tornar a programação do Arduino simples e prazerosa, evitando as dificuldades de um código não cooperativo que tão frequentemente aflige um projeto. Você será conduzido passo a passo pelo processo de programação do Arduino, começando com os fundamentos da linguagem C de programação usada nos Arduinos.

Afinal, o que é o Arduino?

O Arduino é uma pequena placa de microcontrolador contendo um plugue de conexão USB (universal serial bus*) que permite a ligação com um computador. Além disso, contém diversos outros terminais que permitem a conexão com dispositivos externos, como motores, relés, sensores luminosos, diodos a laser, alto-falantes e outros. Os Arduinos podem ser energizados por um computador através do plugue USB, por uma bateria de 9 V ou por uma fonte de alimentação. Eles podem ser controlados diretamente pelo computador, ou então podem ser programados pelo computador e, em seguida, desconectados, permitindo assim que trabalhem independentemente do computador.

O projeto da placa é aberto. Isso significa que qualquer um pode construir placas compatíveis com o Arduino. Essa competição resultou em placas de baixo custo.

As placas básicas são complementadas por placas acessórias, denominadas *shields*, que podem ser encaixadas por cima da placa do Arduino. Neste livro, usaremos dois shields – um shield de display LCD e um shield Ethernet – que nos permitirão fazer o nosso Arduino funcionar como um pequeno servidor web.

* N. deT.: Barramento serial universal.

O software de programação do seu Arduino é de fácil uso e encontra-se disponível gratuitamente para computadores Windows, Mac e LINUX.

O que precisarei?

Este é um livro dirigido a principiantes, mas que também pretende ser útil às pessoas que já usam Arduino há algum tempo e desejam aprender mais sobre a programação ou obter uma maior compreensão dos seus fundamentos.

Você não precisa de conhecimento técnico ou experiência anterior em programação, e os exercícios do livro não exigem soldagem de componentes. Tudo que você precisa é o desejo de fazer alguma coisa.

Se você quiser tirar o máximo do livro e fazer alguns dos experimentos, então será útil ter à mão o seguinte material:

- Alguns pedaços de fio rígido encapado
- Um multímetro de baixo custo

Com alguns reais, os dois podem ser comprados facilmente em qualquer loja de componentes eletrônicos. Naturalmente, você precisará também de uma placa de Arduino Uno.

Se você quiser ir além e fazer experimentos com Ethernet e um display de cristal líquido (LCD), então você precisará comprar shields, que estão à venda em lojas virtuais. Para maiores detalhes, veja os Capítulos 9 e 10.

Usando este livro

Este livro foi estruturado para que você possa dar os primeiros passos de modo realmente simples e para que possa realizar os projetos de forma gradual, usando o que você já aprendeu. Entretanto, você poderá pular ou fazer uma leitura mais rápida de algumas partes dos primeiros capítulos até encontrar o nível adequado para acompanhar o livro.

O livro está organizado de acordo com os seguintes capítulos:

- **Capítulo 1: Este é o Arduino** É uma introdução ao hardware do Arduino. Este capítulo descreve o que ele pode fazer e as diversas placas disponíveis de Arduino.
- **Capítulo 2: Começando** Aqui você realiza os primeiros experimentos com a sua placa de Arduino: a instalação do software, a ligação da alimentação elétrica e a transferência (uploading) do seu primeiro sketch (programa).
- **Capítulo 3: Fundamentos de linguagem C** Este capítulo aborda os fundamentos da linguagem C. Para os totalmente iniciantes em programação, os capítulos também servem de introdução à programação em geral.

- **Capítulo 4: Funções** Este capítulo explica o conceito-chave do uso e escrita de funções para os sketches do Arduino. Esses sketches são demonstrados ao longo de todo o texto usando exemplos executáveis de códigos.
- **Capítulo 5: Arrays e strings** Aqui você aprende a construir e usar estruturas de dados que são mais avançadas do que simples variáveis inteiras. Um projeto que usa código Morse é desenvolvido passo a passo servindo de exemplo para ilustrar os conceitos que estão sendo explicados.
- **Capítulo 6: Entrada e saída** Aqui você aprende a usar as entradas e saídas, digitais e analógicas, do Arduino em seus programas. Um multímetro será útil para mostrar o que está acontecendo nas conexões de entrada e saída do Arduino.
- **Capítulo 7: A biblioteca padrão do Arduino** Este capítulo explica como usar as funções padronizadas do Arduino, as quais fazem parte da biblioteca padrão do Arduino.
- **Capítulo 8: Armazenamento de dados** Aqui você aprende a escrever sketches que salvam dados em EEPROM (electrically erasable programmable read-only memory) e a usar a memória flash interna do Arduino.
- **Capítulo 9: Displays LCD** Neste capítulo, usando a biblioteca do Shield de LCD, você programa e constrói uma placa de mensagens USB simples.
- **Capítulo 10: Programação Ethernet do Arduino** Aqui você aprende a fazer que o Arduino funcione como um servidor de web. Ao mesmo tempo, aprende alguns fundamentos da linguagem HyperText Markup Language (HTML) e do protocolo HyperText Transfer Protocol (HTTP).
- **Capítulo 11: C++ e bibliotecas** Neste capítulo você vai além da linguagem C, aprendendo como incluir orientação a objetos e como escrever as suas próprias bibliotecas para o Arduino.

Material de apoio

Este livro é complementado pelo seguinte site:

www.arduinobook.com*

Nele você encontrará todos os códigos-fonte dos sketches usados neste livro, assim como outros materiais de apoio em inglês.

* N. de T.: O Arduino faz parte de um conceito dinâmico em contínuo desenvolvimento. O leitor está convidado a visitar regularmente o site do autor para se manter atualizado em relação ao conteúdo do livro. Também é de grande valia conhecer e acompanhar o site www.arduino.cc dos criadores do Arduino.

capítulo 1

Este é o Arduino

O Arduino é uma plataforma de microcontrolador que atraiu a imaginação dos entusiastas de eletrônica. A sua facilidade de uso e a sua natureza aberta fazem dele uma ótima opção para qualquer um que queira construir projetos eletrônicos.

Basicamente, permite que você conecte circuitos eletrônicos aos seus terminais, de modo que ele possa controlar dispositivos – por exemplo, ligar ou desligar lâmpadas e motores, ou medir coisas como luz e temperatura. Essa é a razão pela qual o Arduino algumas vezes recebe o atributo de *computação física*. Como os Arduinos podem ser conectados ao seu computador por meio de um barramento serial universal (USB), isso significa também que você pode usar o Arduino como placa de interface e controlar aqueles mesmos dispositivos a partir de seu computador.

Este capítulo é uma introdução ao Arduino, incluindo a sua história e os seus fundamentos, assim como uma visão geral do seu hardware.

Objetivos deste capítulo

- Conhecer microcontroladores e placas de desenvolvimento
- Conhecer uma placa de Arduino
- Conhecer a história do Arduino
- Conhecer os diversos tipos de placas de Arduinos existentes

Microcontroladores

O coração do seu Arduino é um microcontrolador. A maioria dos diversos componentes da placa destina-se ao fornecimento de energia elétrica e à comunicação da placa com o computador.

Na realidade, um microcontrolador é um pequeno computador dentro de um chip. Ele tem tudo que havia nos primeiros computadores domésticos e ainda outras coisas. Ele contém um processador, um ou dois quilobytes de memória RAM* para guardar dados, uns poucos quilobytes de memória EPROM** (memória flash) para armazenar os programas, e ainda pinos de entrada e saída. Esses pinos de entrada/saída (E/S) ligam o microcontrolador aos demais componentes eletrônicos.

As entradas podem ler dados digitais (a chave está ligada ou desligada?) e analógicos (qual é a tensão em um pino?). Isso possibilita a conexão de muitos tipos diferentes de sensores de luz, temperatura, som e outros.

As saídas também podem ser analógicas ou digitais. Assim, você pode fazer um pino estar ativado ou desativado (0 volts ou 5 volts) permitindo que diodos emissores de luz (LEDs) sejam ligados ou desligados diretamente, ou você pode usar a saída para controlar dispositivos com potências mais elevadas, tal como um motor. Esses pinos também podem fornecer uma tensão de saída analógica. Isto é, você pode fazer a saída de um pino apresentar uma dada tensão, permitindo que você ajuste a velocidade de um motor ou o brilho de uma lâmpada, em vez de simplesmente ligá-los ou desligá-los.

O microcontrolador de uma placa de Arduino é o chip (circuito integrado) de 28 pinos que está encaixado em um soquete no centro da placa. Esse único chip contém a memória, o processador e toda a eletrônica necessária aos pinos de entrada e saída. Ele é fabricado pela empresa Atmel, que é uma das maiores fabricantes de microcontroladores. Cada um desses fabricantes produz dúzias de diferentes microcontroladores que são agrupados em diversas famílias. Nem todos os microcontroladores são criados especialmente para aficionados de eletrônica como nós. Nós somos apenas uma pequena fatia desse vasto mercado. Na realidade, esses dispositivos destinam-se ao uso em produtos de consumo, como carros, máquinas de lavar roupa, tocadores de DVD, brinquedos infantis e até mesmo aromatizadores de ambiente.

O importante a respeito do Arduino é que ele reduz essa enorme variedade de possíveis escolhas a um único microcontrolador padrão, que pode ser adotado de forma permanente. (Bem, como veremos mais adiante, essa afirmação não é exatamente verdadeira, mas está muito próxima da verdade.)

* N. de T.: RAM (Random Access Memory, ou Memória de Acesso Aleatório).

** N. de T: EPROM (Erasable Programmable Read Only Memory, ou Memória Apenas de Leitura, Programável e Apagável).

Isso significa que, quando você embarcar em um novo projeto, você não precisará analisar primeiro todos os prós e contras dos diversos tipos de microcontroladores.

Placas de desenvolvimento

Vimos que o microcontrolador é, na realidade, apenas um chip. Um chip não consegue trabalhar sozinho a menos que conte com uma eletrônica de apoio para alimentá-lo com uma fonte de eletricidade precisamente regulada (os microcontroladores são exigentes no que se refere a isso) e para propiciar um meio de comunicação com o computador que será usado na programação do microcontrolador.

Esse é o ponto em que as placas de desenvolvimento entram em cena. Uma placa de Arduino é, na verdade, uma placa de desenvolvimento baseada em microcontrolador, cujo projeto é independente e de hardware aberto (open source). Isso significa que os arquivos do projeto da placa de circuito impresso (PCB) e os diagramas esquemáticos estão todos disponíveis publicamente. Qualquer pessoa pode usar livremente esses projetos para fabricar e vender as suas próprias placas de Arduino.

Todos os fabricantes de microcontroladores – incluindo a Atmel que produz o microcontrolador ATmega328 usado na placa do Arduino – também oferecem as suas próprias placas de desenvolvimento e o software de programação. Embora usualmente sejam bem baratas, elas tendem a ser destinadas a engenheiros eletrônicos profissionais em vez de a aficionados amadores. Isso significa que tais placas e o software são mais difíceis de usar e exigem um maior investimento de aprendizagem antes de você começar a tirar qualquer coisa útil delas.

Um passeio por uma placa de Arduino

A Figura 1-1 mostra uma placa de Arduino. Vamos dar um rápido passeio pelos vários componentes da placa.

Fonte de alimentação

Na Figura 1-1, diretamente abaixo do conector USB, está o regulador de tensão de 5 volts (5V). Ele recebe qualquer tensão (entre 7V e 12V) que esteja sendo fornecida pelo conector de alimentação e a converte em uma tensão constante de 5V.

Como componente de uma placa de circuito impresso, o chip regulador da tensão de 5V tem, na realidade, um tamanho bem grande. Isso possibilita uma dissipação elevada de calor, tal como é necessário para regular a tensão quando a corrente solicitada é razoavelmente alta. Isso é útil para acionar dispositivos eletrônicos externos.

Conexões de alimentação elétrica

A seguir, vamos examinar os conectores de alimentação elétrica na parte de baixo da Figura 1-1. Próximo dos conectores, você pode ler os seus nomes. O primeiro é o de Reset. Ele faz a mesma coisa que o botão de Reset do Arduino. De forma semelhante ao que ocorre quando reiniciamos um computador PC, se ativarmos o conector de Reset do Arduino, o microcontrolador será inicializado começando a executar o seu sketch desde o início. Para inicializar o microcontrolador usando o conector de Reset, você deve manter esse pino momentaneamente em nível baixo (conectando-o a 0V).

Os demais pinos desta seção simplesmente fornecem diversas tensões (3,3V, 5V, GND e 9V), conforme estão indicadas na placa. GND (ground ou terra) significa simplesmente zero volts. É a tensão que serve de referência a todas as outras tensões da placa.

Entradas analógicas

Os seis pinos indicados como Analog In, de A0 a A5, podem ser usados para medir a tensão que está sendo aplicada a cada pino, de modo que os seus valores podem ser usados em um programa (sketch). Observe que nos pinos são medidas as tensões e não as correntes. Como os pinos têm uma resistência interna muito elevada, apenas uma diminuta corrente entrará

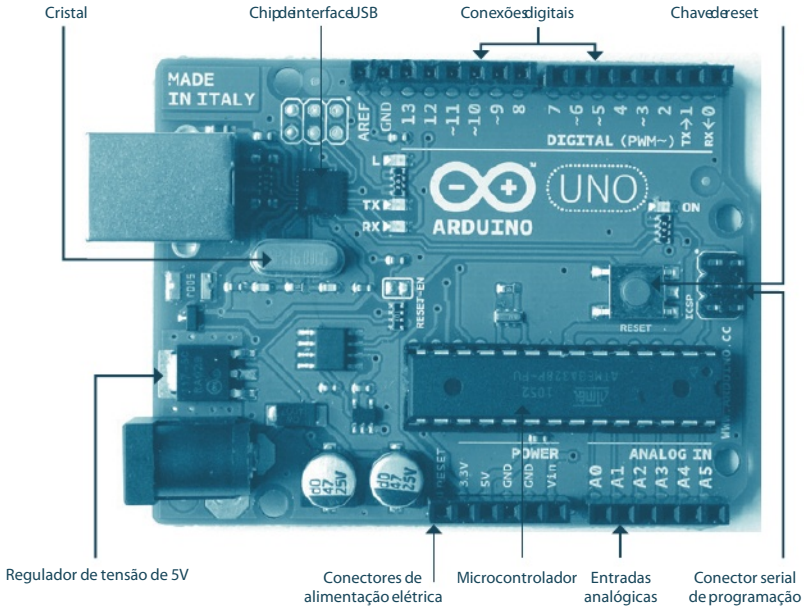

Figura 1-1 Uma placa do Arduino Uno.

em cada pino passando internamente até o pino GND. Isto é, a elevada resistência interna dos pinos permite que somente uma corrente muito baixa consiga entrar neles.

Embora essas entradas estejam indicadas como analógicas, sendo entradas analógicas por default,* essas conexões também poderão ser usadas como entradas ou saídas digitais.

Conexões digitais

Agora passaremos para o conector da parte de cima da Figura 1-1 começando pelo lado direito. Aqui encontramos os pinos denominados Digital, de 0 a 13. Eles podem ser usados como entradas ou como saídas. Quando usados como saídas, eles se comportam como as tensões da alimentação elétrica, discutidas anteriormente nesta seção, exceto que agora todas são de 5V e podem ser ligadas ou desligadas a partir de um sketch. Assim, se você ligá-las em seu sketch, elas ficarão com 5V. Se você desligá-las, elas ficarão com 0V. Como no caso das conexões da alimentação elétrica, você deve tomar cuidado para não ultrapassar as suas capacidades máximas de corrente. Os primeiros dois pinos (0 e 1), também denominados RX e TX, são para recepção e transmissão. Essas conexões estão reservadas para uso na comunicação. Indiretamente, são as conexões da recepção e transmissão USB usadas pelo Arduino para se comunicar com o seu computador.

Essas conexões digitais podem fornecer 40 mA (miliamperes) com 5V. Isso é mais do que suficiente para acender um LED comum, mas é insuficiente para acionar diretamente um motor elétrico.

Microcontrolador

Continuando o nosso passeio pela placa do Arduino, o microcontrolador em si é o dispositivo retangular preto com 28 pinos. Ele está encaixado em um soquete do tipo dual in-line (DIL), de modo que pode ser facilmente substituído. O chip de 28 pinos do microcontrolador usado na placa do Arduino Uno é o ATmega328. A Figura 1-2 mostra um diagrama de blocos com as características deste dispositivo.

O coração – ou talvez mais apropriadamente o cérebro – do dispositivo é a unidade central de processamento (CPU de Central Processing Unit). Ela controla tudo que acontece dentro do dispositivo, buscando as instruções do programa armazenado na memória flash e executando-as. Isso significa buscar dados na memória de trabalho (RAM), alterá-los e então colocá-los de volta no lugar. Ou, pode significar uma alteração de tensão em uma das saídas digitais de 0V para 5V.

A memória EEPROM é um pouco parecida com a memória flash no sentido de que não é volátil. Isto é, se você desligar o dispositivo e voltar a liga-lo, ele não esquecerá o que estava na EEPROM. A memória flash é destinada ao armazenamento de instruções de programa (sketches), ao passo que a EEPROM é usada no armazenamento dos dados que você não quer perder no caso de ocorrer um reset ou de ser desligada a alimentação elétrica.

* N. de T.: *Default* é um termo inglês muito usado em diversas situações e indica algo que será automaticamente adotado se nada houver em contrário.

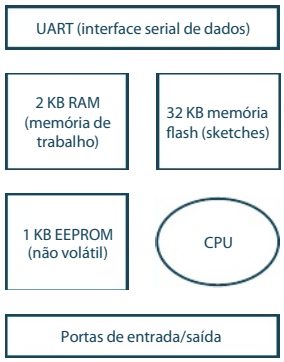

Figura 1-2 Diagrama de blocos do ATmega 328.

Outros componentes

Acima do microcontrolador encontra-se um pequeno componente retangular prateado. É um oscilador a cristal. Ele realiza 16 milhões de ciclos ou oscilações por segundo e, em cada um desses ciclos, o microcontrolador pode executar uma operação – adição, subtração ou alguma outra operação matemática.

À direita do cristal, está a chave de Reset. Quando se aperta essa chave, um pulso lógico é enviado ao pino de Reset do microcontrolador, fazendo o microcontrolador iniciar o seu programa do zero e limpar a memória. Observe que qualquer programa armazenado no dispositivo será preservado, porque ele está em uma memória flash não volátil – isto é, memória que não esquece mesmo quando o dispositivo não está sendo energizado.

À direita do botão de Reset, encontra-se o Conector Serial de Programação. Ele oferece um outro meio para programar o Arduino sem que a porta USB seja usada. Como nós já temos uma conexão USB e um software que torna conveniente o seu uso, nós não iremos utilizar esse recurso.

No canto superior esquerdo da placa junto ao soquete USB, encontra-se o chip de interface USB. Esse chip converte os níveis de sinal usados pelo padrão USB em níveis que podem ser usados diretamente pela placa do Arduino.

As srcens do Arduino

Originalmente, o Arduino foi desenvolvido como recurso auxiliar no ensino dos estudantes. Mais adiante (em 2005), ele foi desenvolvido comercialmente por Massimo Banzi e David Cuartielles. Desde então, ele se tornou um produto extremamente bem-sucedido junto a fabricantes, estudantes e artistas, devido à sua facilidade de uso e durabilidade.

Um outro fator-chave do seu sucesso é que todos os projetos com Arduino estão disponíveis gratuitamente sob uma licença da Creative Commons. Isso permitiu que aparecessem muitas

placas alternativas de custo menor. Somente o nome Arduino está protegido, de modo que tais clones frequentemente têm nomes do tipo "*duino", tais como Boarduino, Seeeduino e Freeduino. Entretanto, as placas oficiais fabricadas na Itália continuam sendo vendidas de forma extremamente bem-sucedida. Muitas lojas de renome vendem somente as placas oficiais, que são de excelente qualidade e vêm dentro de embalagens primorosas.

Uma outra razão para o sucesso do Arduino é que ele não se limita a placas com microcontrolador. Há um número enorme de placas acessórias (denominadas shields) compatíveis com o Arduino. Essas placas são encaixadas diretamente por cima da placa do Arduino. Como há shields disponíveis para praticamente qualquer aplicação que se possa imaginar, você frequentemente poderá dispensar o uso do ferro de soldar e, em vez disso, poderá conectar diversos shields empilhando-os entre si. A lista a seguir mostra apenas alguns poucos exemplos dos shields mais populares:

- Ethernet, shield que dá recursos para que uma placa de Arduino funcione como servidor de web
- Motor, shield que aciona motores elétricos
- USB Host (Hospedeiro USB), shield que permite o controle de dispositivos USB
- Relays (Relés), shield que comanda relés a partir do seu Arduino

A Figura 1-3 mostra um Arduino Uno acoplado a um shield de Ethernet.

A família Arduino

É útil conhecer um pouco das diversas placas de Arduino. Como dispositivo padrão, nós usaremos a placa de Arduino Uno. Na verdade, essa placa de Arduino é de longe a mais usada, mas todas são programadas com a mesma linguagem e a maioria usa as mesmas conexões com o mundo exterior, de modo que você pode facilmente usar uma placa diferente.

Uno, Duemilanove e Diecimila

O Arduino Uno é a última encarnação da série mais popular de placas Arduino. A série inclui o Diecimila (10.000 em italiano) e o Duemilanove (2009 em italiano). A Figura 1-4 mostra um clone do Arduino. A esta altura, você já deve ter adivinhado que o Arduino é uma invenção italiana.

Essas placas mais antigas são muito semelhantes ao Arduino Uno. Todas têm os mesmos conectores e um soquete USB, sendo geralmente compatíveis entre si.
A diferença mais significativa entre o Uno e as placas anteriores é que o Uno usa um chip USB diferente. Isso não afeta o modo de você usar a placa, mas facilita a instalação do software e permite velocidades de comunicação mais elevadas com o computador.

Com sua fonte de alimentação de 3,3V, o Uno também pode fornecer uma corrente maior e sempre vem equipado com o ATmega328. As placas anteriores têm um ATmega328 ou um ATmega168. A memória do ATmega328 é maior, mas isso não fará diferença, a menos que você esteja criando um sketch de grande porte.

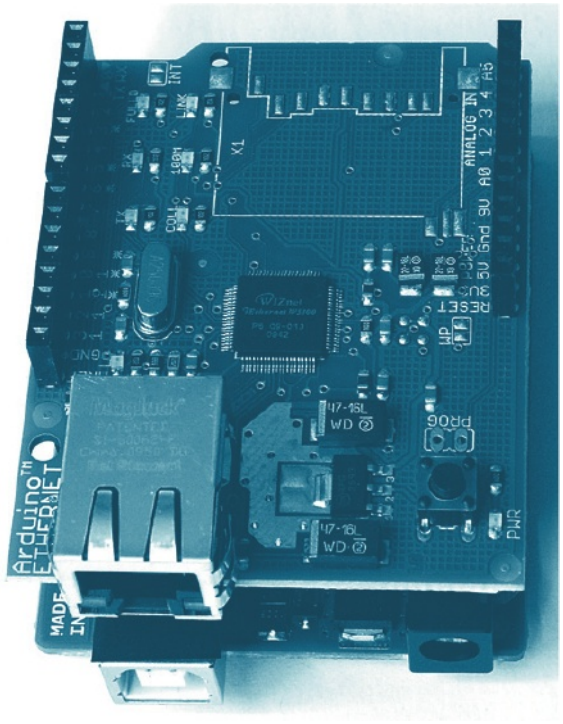

Figura 1-3 Um Arduino Uno com um shield de Ethernet.

Mega

O Arduino Mega (Figura 1-5) é o carro de alta performance das placas de Arduino. Ele ostenta uma grande coleção de portas de entrada e saída. Isso é feito engenhosamente colocando conectores extras em um dos lados da placa, de tal modo que a placa permanece compatível pino a pino com o Arduino Uno e todos os shields disponíveis de Arduino.

Ele usa o processador ATmega1280 que tem mais pinos de entrada e saída. Como esse chip é do tipo de *montagem superficial*, isso significa que ele está fixado de forma permanente à placa e, diferentemente do Uno e de outras placas similares, você não poderá substituí-lo caso você venha a danificá-lo acidentalmente.

Os conectores extras estão dispostos em um dos lados da placa. Entre as características extras oferecidas pelo Mega, estão as seguintes:

- 54 pinos de entrada e saída
- 128KB de memória flash para armazenar sketches e dados fixos (em comparação com os 32KB do Uno)
- 8KB de RAM e 4KB de EEPROM

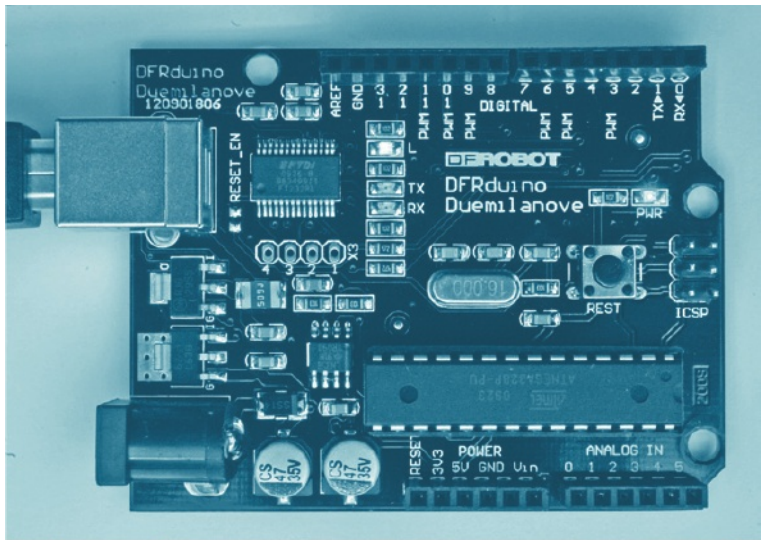

Figura 1-4 O Arduino Duemilanove.

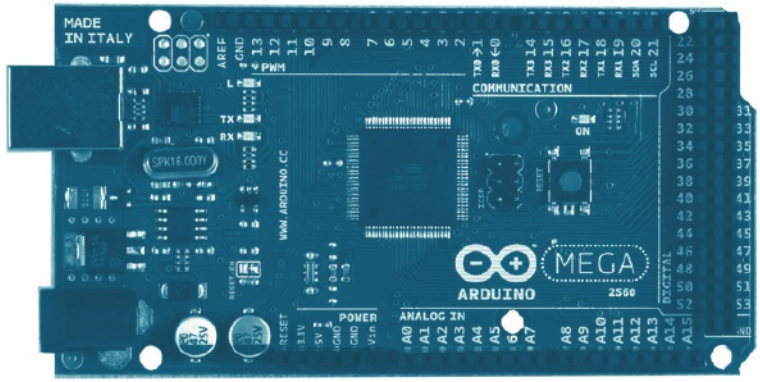

Figura 1-5 Uma placa de Arduino Mega.

» Nano

O Arduino Nano (Figura 1-6) é um dispositivo muito útil para ser usado com protoboards (matrizes de contatos) que dispensam o uso de soldador. Se você colocar pinos nele, você poderá simplesmente encaixá-lo no protoboard como se fosse um chip.

A desvantagem do Nano é que ele não aceita os shields do Uno por ser muito menor do que ele.

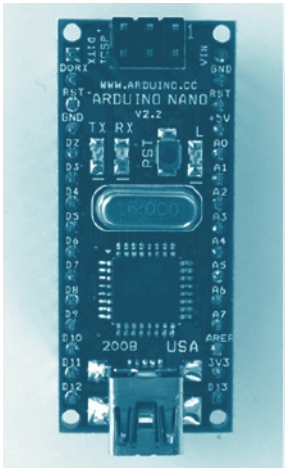

Figura 1-6 O Arduino Nano.

Bluetooth

O Arduino Bluetooth (Figura 1-7) é um dispositivo interessante porque inclui hardware de Bluetooth em lugar do conector USB. Isso permite que o dispositivo seja programado sem o uso de cabos de conexão.

O Arduino Bluetooth é uma placa de custo mais elevado. Frequentemente, sai mais barato usar um módulo Bluetooth de outro fabricante e anexá-lo a um Arduino Uno comum.

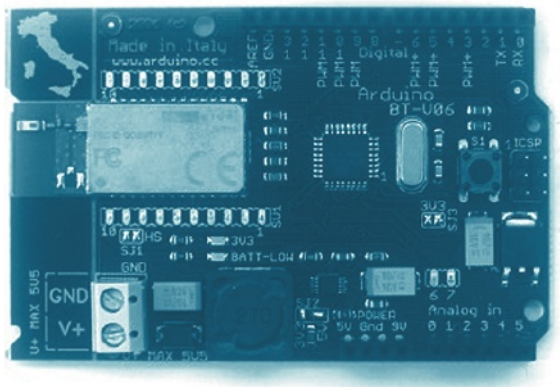

Figura 1-7 O Arduino Bluetooth.

Lilypad

O Lilypad (Figura 1-8) é uma placa de Arduino pequena e de tão pouca espessura que pode ser costurada a uma vestimenta e ser usada em aplicações que se tornaram conhecidas como computação vestível (wearable computing).

Como o Lilypad não tem conexão USB, você deve usar um adaptador em separado para programá-lo. O seu visual é excepcionalmente bonito.

Outras placas "oficiais"

As placas de Arduino recém-descritas são as mais úteis e populares. Entretanto, a oferta de placas de Arduino está constantemente crescendo. Você poderá ter uma visão completa e atualizada da família Arduino na lista que está disponível no site oficial do Arduino em www.arduino.cc/en/Main/Hardware.

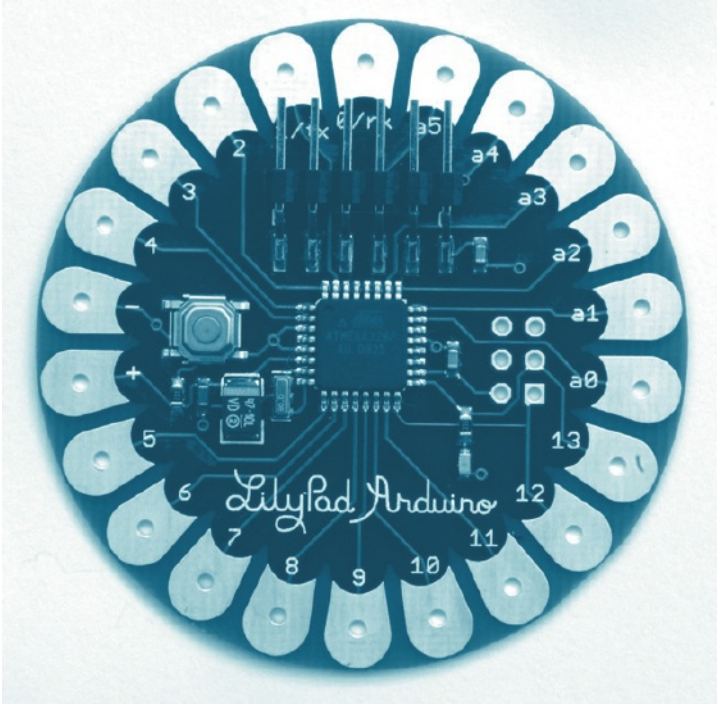

Figura 1-8 O Arduino Lilypad.

Clones e variantes do Arduino

As placas de Arduino não oficiais dividem-se em duas categorias. Alguns fabricantes baseiam-se nos projetos padrões de hardware open source do Arduino e fabricam placas mais baratas. A seguir, estão alguns nomes de placas dessa natureza que você pode pesquisar:

- Roboduino
- Freeduino
- Seeeduino (sim, com três *e*'s)

Mais interessante ainda é que alguns projetos compatíveis com o Arduino destinam-se a ampliar ou aperfeiçoar o Arduino de algum modo. Novas variantes aparecem constantemente e são numerosas demais para serem mencionadas aqui. Contudo, algumas das variantes mais interessantes e populares são as seguintes:

- Chipkit, uma variante de alta velocidade baseada em um processador PIC, mas que é bem compatível com o Arduino
- Femtoduino, um Arduino muito pequeno
- Ruggeduino, que é uma placa de Arduino com proteção interna de entrada e saída
- Teensy, com uma placa muito pequena e de baixo custo

Conclusão

Agora que você explorou um pouco o hardware do Arduino, chegou o momento de instalar o software do Arduino.

» capítulo 2

Começando

Depois de introduzir o Arduino e ter aprendido um pouco a respeito do que estamos programando, chegou o momento de instalar em nosso computador o software que precisaremos para começar a trabalhar com os códigos de programa.

Objetivos deste capítulo

- Aprender a ligar eletricamente o Arduino
- Aprender a instalar o software aplicativo do Arduino
- Aprender a instalar um Sketch na placa do Arduino
- Fazer um pequeno passeio pelo aplicativo Arduino

Ligando a alimentação elétrica

Quando você compra uma placa de Arduino, geralmente ela vem com um programa de Blink (pisca-pisca) já instalado. Esse programa fará piscar o pequeno diodo emissor de luz (LED) que faz parte da placa.

O LED marcado com *L* está ligado a um dos terminais de entrada e saída da placa. Ele está conectado ao pino digital 13. Isso faz o pino 13 poder ser usado apenas como saída. Entretanto, como o LED consome somente uma pequena quantidade de corrente, você ainda pode conectar outras coisas a esse pino.

Tudo o que você precisa fazer para colocar o seu Arduino em funcionamento é alimentá-lo com energia elétrica. A maneira mais fácil de fazer isso é conectando-o na porta USB do seu computador. Você precisará de um cabo USB do tipo A para tipo B. Esse cabo é o mesmo usado normalmente para conectar uma impressora a um computador.

Se tudo estiver funcionando corretamente, o LED deverá piscar. As novas placas de Arduino já vêm com esse sketch de Blink (pisca-pisca) instalado, de modo que você poderá verificar se a placa está funcionando.

Instalando o software

Para que você possa instalar novos sketches na sua placa de Arduino, você precisa fazer outras coisas além de alimentá-la com energia elétrica por meio da USB. Você precisa instalar o software aplicativo do Arduino (Figura 2-1).

Instruções completas e abrangentes de como instalar esse software em computadores Windows, Linux e Mac podem ser encontradas no site do Arduino (www.arduino.cc).

Depois de ter instalado com sucesso o software do Arduino e, dependendo da sua plataforma, os drivers de USB, você então poderá fazer o upload (transferência) de um programa para a placa do Arduino.

Instalando o seu primeiro Sketch

O LED pisca-pisca é o equivalente Arduino do programa "Alô Mundo". Quando estamos aprendendo uma nova linguagem, esse é o primeiro programa que tradicionalmente costuma ser executado. Para testar o ambiente de programação, vamos instalar esse programa na placa do Arduino e, em seguida, modificá-lo.

Quando você abre o soft ware aplicativo do Arduino no seu computador, ele começa com um sketch (programa) vazio. Felizmente, esse aplicativo vem acompanhado de muitos exemplos úteis. Assim, a partir do menu File (Arquivo), abra o sketch Blink (pisca-pisca), como está mostrado na Figura 2-2.

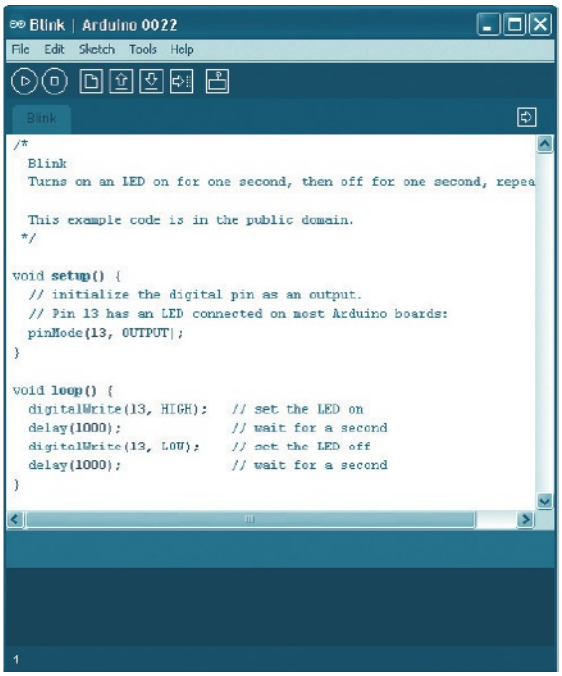

Figura 2-1 O programa aplicativo do Arduino*

Agora você precisa transferir (upload) esse sketch para a sua placa de Arduino. Para isso, conecte a placa de Arduino ao seu computador usando o cabo USB. Você verá que o LED com a indicação de "ON" (Ligado) do Arduino irá brilhar. Provavelmente, a placa do Arduino estará piscando, porque as placas geralmente vêm com o sketch de Blink (pisca-pisca) já instalado. Mas nós o instalaremos novamente para poder modificá-lo.

Se você estiver usando um Mac, é possível que, quando você conectar a placa, apareça uma mensagem alertando que uma nova interface de rede foi detectada. Simplesmente clique na opção de cancelamento. O seu Mac está confuso e pensa que o Uno é um modem USB.

Antes de transferir um sketch, o programa aplicativo Arduino precisa saber qual é o tipo de placa que você utilizará e qual será a porta serial que você usará para fazer a conexão. As Figuras 2-3 e 2-4 mostram como você deve fazer isso a partir do menu Tools (Ferramentas).

Em uma máquina Windows, a porta serial é geralmente a COM4. Nas máquinas Mac e Linux, você terá uma lista muito maior de dispositivos seriais (veja a Figura 2-5). O dis-

* N. de T.: Como foi indicado em nota de rodapé na Introdução, o Arduino está constantemente em desenvolvimento e aperfeiçoamento. Esta figura ilustra o software em sua versão 0022. Para se manter atualizado, o leitor poderá acessar o site do autor em http://www.arduinobook.com/home.

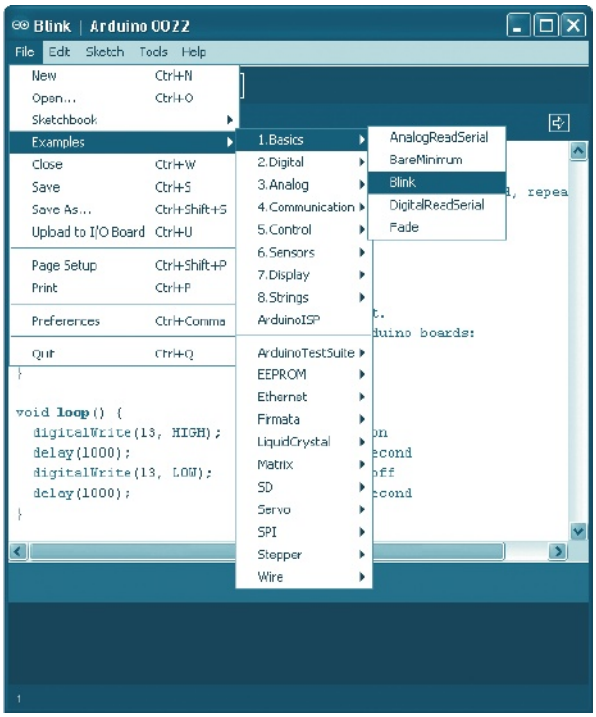

Figura 2-2 O sketch Blink (pisca-pisca).

positivo será normalmente o que está bem no topo da lista, com um nome similar a /dev/tty.usbmodem621.

Agora clique no ícone de Upload (transferir para a placa de Arduino) da barra de ferramentas. Isto está destacado na Figura 2-6.

Depois de clicar no botão, haverá uma breve pausa enquanto o sketch é compilado e então começará a transferência para o Arduino. Se tudo estiver funcionando, haverá um pisca-pisca furioso de LEDs à medida que o sketch é transferido. No final, na parte de baixo da janela do aplicativo Arduino você deverá ver a mensagem "Done Uploading" (ou seja, "Transferência

Realizada") e uma outra mensagem similar a "Binary sketch size: 1018 bytes (of a 14336 byte maximum)" (ou seja, "Tamanho do sketch em binário: 1018 bytes (de um máximo de 14336 bytes)").

Depois da transferência, a placa começa automaticamente a executar o sketch e você verá o LED começar a piscar.

Se não funcionar, então verifique os seus ajustes de porta serial e tipo de placa.

Agora vamos modificar o sketch para que o LED pisque mais rapidamente. Para isso, vamos alterar os dois lugares do sketch onde há um retardo de 1.000 milissegundos – delay(1000)

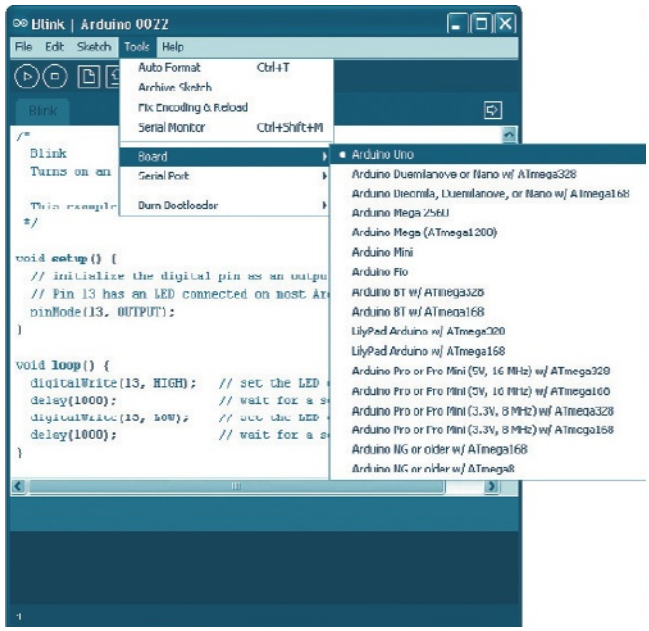

Figura 2-3 Selecionando o tipo de placa.

Figura 2-4 Selecionando a porta serial (no Windows).

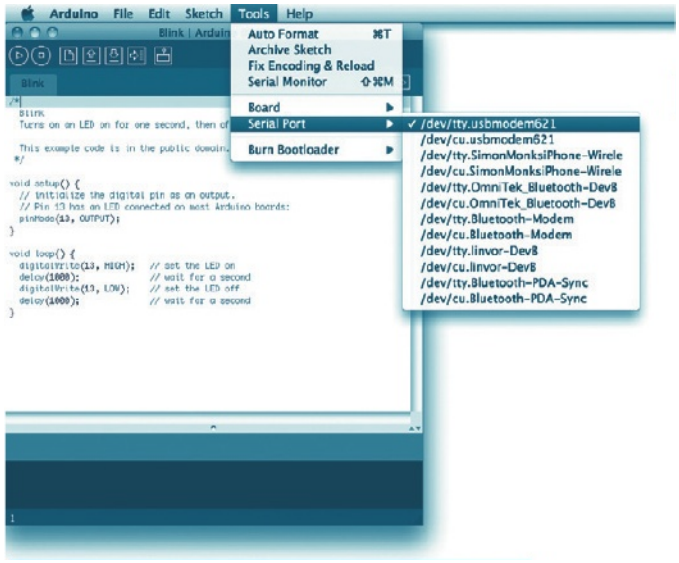

Figura 2-5 Selecionando a porta serial (em um Mac).

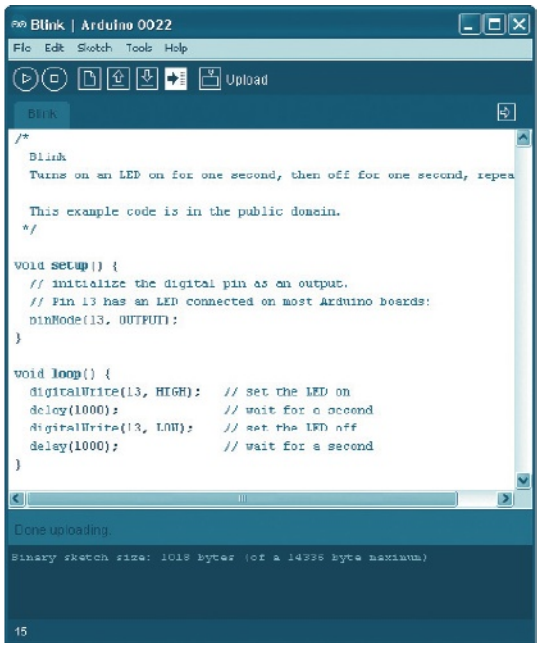

Figura 2-6 Transferindo (Uploading) o sketch para a placa de Arduino.

– de modo que o retardo passe a ser de 500 milissegundos – delay(500). A Figura 2-7 mostra o sketch modificado, com as alterações realçadas.

Clique novamente no ícone de Upload para fazer a transferência do sketch. Então, você deverá ver o LED começando a piscar duas vezes mais rápido do que antes.

Parabéns, agora você está pronto para começar a programar o seu Arduino. Primeiro, contudo, vamos dar um pequeno passeio pelo aplicativo Arduino.

O aplicativo Arduino

Os sketches do Arduino são como documentos em um editor de texto. Você pode abri-los e copiar partes de um para outro. Assim, você terá opções como Open, Save e Save As (Abrir, Salvar e Salvar Como) no menu File (Arquivo). Normalmente você não usará a opção Open porque o aplicativo Arduino usa o conceito denominado Sketchbook ("Livro de sketches") no qual todos os seus sketches são guardados e cuidadosamente organizados em pastas. Você poderá ter acesso ao Sketchbook a partir do menu File. Como você acabou de instalar e abrir

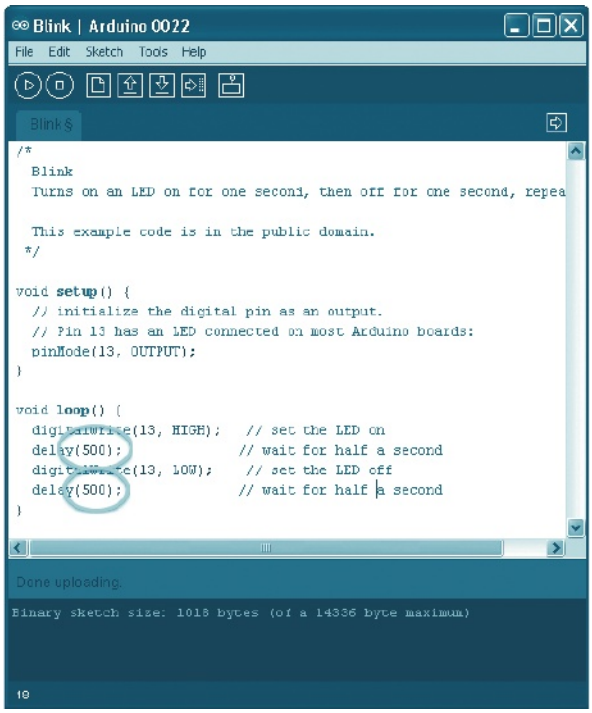

Figura 2-7 Modificando o sketch Blink (pisca-pisca).

o aplicativo Arduino pela primeira vez, o seu Sketchbook está e ficará vazio até que você crie alguns sketches.

Como você viu, o aplicativo Arduino vem com uma coleção de exemplos de sketches que pode ser muito útil. Se você tentar salvar o sketch Blink (pisca-pisca) depois de modificá-lo, aparecerá uma mensagem dizendo que alguns arquivos são somente de leitura e, portanto, você deverá salvar o sketch em algum outro local.

Tente fazer isso agora. Aceite o local que está sendo proposto, mas mude o nome do arquivo para MyBlink (MeuPiscaPisca), como está mostrado na Figura 2-8.

Agora, se você for para o menu File e clicar em Sketches, você verá MyBlink como um dos sketches listados. No caso de um computador PC, se você olhar o sistema de arquivos, você verá que o sketch foi salvo em Meus Documentos. Em um Mac ou Linux, ele estará em Documents/Arduino.

Todos os sketches usados neste livro podem ser baixados como um arquivo zip (Programming_Arduino.zip) no site www.arduinobook.com. Agora é o momento de baixar esse arquivo e descompactá-lo na pasta Arduino que contém os sketches. Em outras palavras, depois de descompactá-lo, haverá duas pastas dentro da pasta Arduino: uma para o MyBlink que acabou de ser salvo, e uma denominada Programming Arduino* (veja a Figura 2-9). A pasta Programming Arduino conterá todos os sketches deste livro, numerados de acordo com os capítulos, de modo que o sketch 03-01, por exemplo, é o sketch 1 do Capítulo 3.

Esses sketches não aparecerão no menu do Sketchbook até você sair do aplicativo Arduino e entrar de novo nele. Faça isso agora. O seu menu do Sketchbook deverá ser parecido com o mostrado na Figura 2-10.

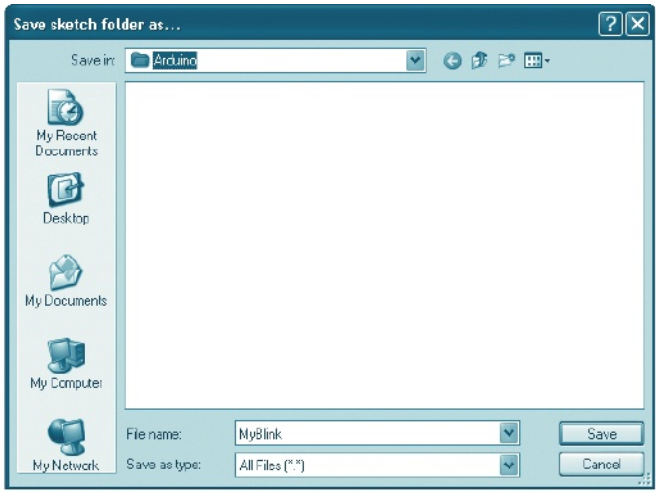

Figura 2-8 Salvando uma cópia do sketch Blink (pisca-pisca).

* N. deT.: *Programming Arduino* é o título srcinal deste livro em inglês.

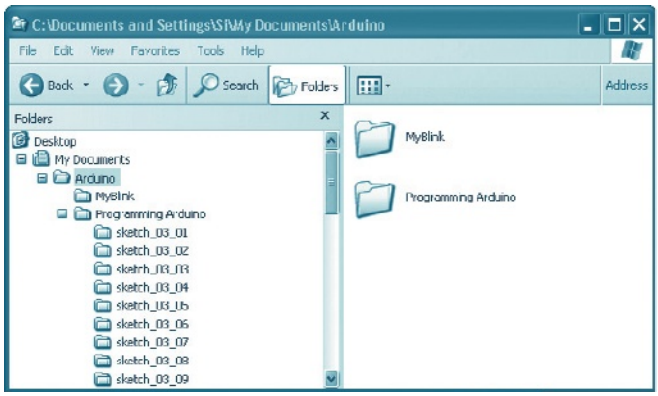

Figura 2-9 Instalação dos sketches do livro.

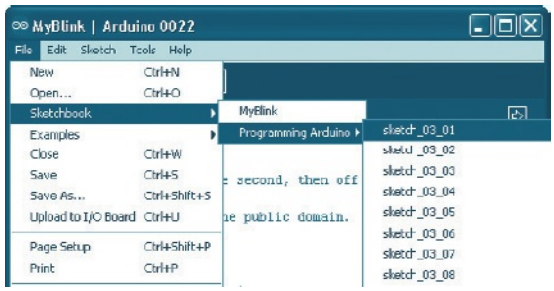

Figura 2-10 Sketchbook com os sketches do livro instalados.

Conclusão

O seu ambiente de trabalho está completamente instalado e pronto para ser usado.

No próximo capítulo, iremos examinar alguns dos princípios básicos da linguagem C usada pelo Arduino e escrever alguns códigos de programa.

» capítulo 3

Fundamentos de linguagem C

A linguagem de programação usada para programar Arduinos é a linguagem C. Neste capítulo, você aprenderá os fundamentos da linguagem C. Como programador de Arduino, você usará o que aprender aqui em todos os sketches desenvolvidos por você. Para tirar o máximo do Arduino, você precisa compreender estes fundamentos.

Objetivos deste capítulo

- Entender o que é programação e como funciona uma linguagem de programação
- Entender um pouco das funções setup e loop
- Aprender fundamentos da linguagem C
- Aprender o que é uma variável em C
- Explorar alguns comandos de decisão e repetição da linguagem C

Programando

É comum as pessoas falarem mais de uma língua. De fato, quanto mais você aprende, mais fácil é aprender a falar outras línguas, uma vez que você começa a se dar conta das estruturas comuns de gramática e vocabulário. O mesmo acontece com as linguagens de programação. Assim, se anteriormente você usava alguma outra linguagem, então você irá assimilar com rapidez a linguagem C.

A boa notícia é que o vocabulário de uma linguagem de programação é muito menor do que o de uma língua falada. Como você escreve em vez de falar, o dicionário sempre poderá ser usado se você precisar consultá-lo. Além disso, a gramática e a sintaxe de uma linguagem de programação são extremamente regulares e, tão logo você domine alguns conceitos, você começará a aprender com rapidez de forma fácil e natural.

O melhor é pensar que um programa – ou sketch, como são denominados os programas em Arduino – representa uma lista de instruções, as quais devem ser executadas na ordem em que foram escritas. Por exemplo, suponha que você tivesse escrito o seguinte:

```
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
```

Cada uma dessas três linhas faria alguma coisa. A primeira linha faria o pino 13 de saída digital passar para o nível alto (HIGH). Na placa do Arduino, esse é o pino que já vem com um LED conectado, de modo que, nessa linha, o LED seria aceso. A segunda linha iria simplesmente realizar um retardo (delay) de 500 milissegundos (meio segundo). Finalmente, a terceira linha iria colocar a saída em nível baixo (LOW) apagando o LED. Assim, o conjunto dessas três linhas alcança o objetivo de fazer o LED piscar uma vez.

Nessas três linhas, você pode ver um conjunto de sinais de pontuação, usados de maneira confusa e estranha, e também palavras sem espaço separando-as. Uma frustração de muitos programadores iniciantes é "Eu sei o que quero, mas não sei o que devo escrever!". Não se assuste, tudo será explicado.

Em primeiro lugar, vamos tratar dos sinais de pontuação e da forma como as palavras são formadas. Ambos são parte do que se denomina sintaxe da linguagem. A maioria das linguagens exige que você seja extremamente preciso a respeito da sintaxe, e uma das regras principais é que os nomes das coisas devem ter somente uma palavra. Isto é, não podem incluir espaços em branco. Assim, **digitalWrite** (escrever na saída digital) é o nome de alguma coisa. No caso, é o nome de uma função interna (mais adiante você aprenderá mais sobre as funções). Na placa de Arduino, essa função faz um dos pinos de saída digital ser colocado em um dado nível lógico. Além de não ser permitido usar espaços em branco nos nomes, há regras para o uso de letras maiúsculas e minúsculas. Desse modo, você deve escrever **digitalWrite** e não **DigitalWrite** ou **Digitalwrite**.

A função **digitalWrite** precisa saber qual é o pino que deve ser ativado e se esse pino deve ser colocado em nível HIGH (alto) ou LOW (baixo). Esses dois dados são denominados *argumentos*. Dizemos que esses argumentos são *passados* para uma função quando ela é *chamada*. Os parâmetros de uma função devem ser colocados ente parênteses e separados por vírgulas.

A convenção é abrir parênteses imediatamente após a última letra do nome da função e colocar um espaço em branco entre a vírgula e o parâmetro seguinte. Entretanto, se desejar, você poderá espalhar espaços em branco dentro dos parênteses.

Se a função tiver apenas um argumento, então não há necessidade de vírgula.

Observe que cada linha termina com um ponto e vírgula. Seria mais lógico esperar que fosse usado um ponto, porque se trata do final de um comando, tal como no final de uma frase.

Na próxima seção, você descobrirá mais um pouco sobre o que acontece quando você aperta o botão de Upload (transferir sketch) do aplicativo Arduino, ou mais exatamente no assim denominado ambiente de desenvolvimento integrado (IDE, de Integrated Development Environment). Depois disso tudo, você poderá começar a experimentar com alguns exemplos.

O que é uma linguagem de programação?

Talvez seja um pouco surpreendente que tenhamos chegado a este Capítulo 3 em um livro sobre programação sem ter definido exatamente o que é uma linguagem de programação. Podemos reconhecer um sketch de Arduino e provavelmente ter uma ideia rudimentar do que ele está tentando fazer, mas é necessário que examinemos com mais profundidade como um código escrito em uma linguagem de programação passa de palavras escritas em uma página de texto para algo que faz alguma coisa realmente acontecer, como acender e apagar um LED.

A Figura 3-1 resume o processo que ocorre desde o momento em que escrevemos um código no IDE do Arduino até a execução do sketch na placa.

Quando você clica no ícone de Upload do IDE do Arduino, ele dispara uma sequência de passos que resulta na instalação do seu sketch no Arduino e na sua execução. Isso não é tão sim-

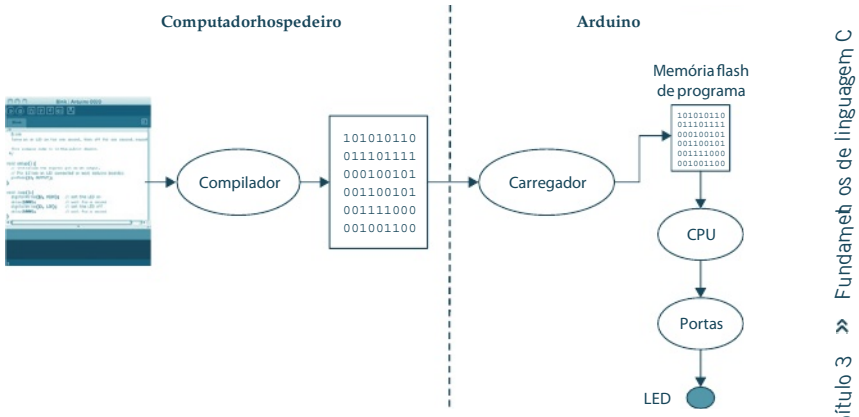

Figura 3-1 Do código para a placa.

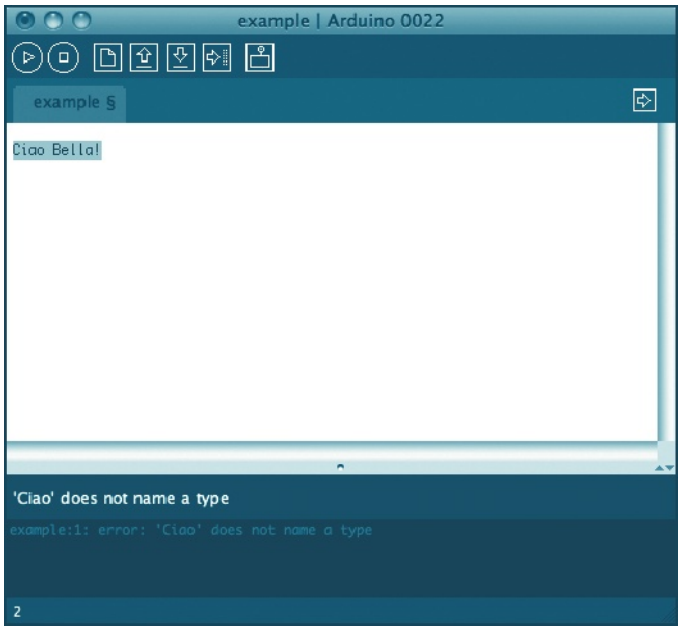

Figura 3-2 Os Arduinos não falam italiano.

ples como pegar o texto que você escreveu no editor e simplesmente transferi-lo para a placa do Arduino.

O primeiro passo é fazer algo denominado *compilação*. Nesse passo, o código que você escreveu é traduzido para o código de máquina – a linguagem binária que o Arduino compreende. Se você clicar no botão triangular de Verify (Verificar) no IDE do Arduino, haverá uma tentativa de compilar o programa em C escrito por você sem que o código seja transferido para a placa do Arduino. Um outro resultado da compilação do código é uma verificação para assegurar se ele está de acordo com a linguagem C.

Se você escrever **Ciao Bella*** no IDE do Arduino e clicar no botão de Play, os resultados serão os mostrados na Figura 3-2.

O Arduino tentou compilar as palavras "Ciao Bella", mas ele não tem ideia do que você está dizendo apesar de sua srcem italiana. Esse texto não está em C. Por isso, o resultado é que, na parte de baixo da tela, temos uma mensagem obscura de erro alertando que palavra Ciao não é o nome de algum tipo usado em C ("error: Ciao does not name a type"). Na realidade, isso significa que há algo muito errado com o que você escreveu.

Vamos tentar um outro exemplo. Desta vez, tentaremos compilar um sketch sem código (veja a Figura 3-3).

* N. de T: "Alô Bela", em italiano.

Figura 3-3 Nem *setup* nem *loop* estão presentes.

Agora, o compilador está indicando um erro de compilação (Error compiling.) e dizendo que o seu sketch não contém as funções **setup** (inicialização) e **loop** (laço). Como você já sabe do exemplo Blink do Capítulo 2, em um sketch você deve incluir algumas linhas padronizadas antes de começar a inserir o seu próprio código. Na programação do Arduino, esse código padronizado assume a forma das funções "setup" e "loop", que devem estar sempre presentes em um sketch.

Nós aprenderemos muito mais sobre funções nas próximas seções deste livro. Por enquanto, vamos aceitar que essas linhas de código padronizado são necessárias e vamos adaptá-las ao nosso sketch de modo que possamos compilá-lo (veja Figura 3-4).

O IDE do Arduino examinou o programa que resultou dos seus esforços de escrever um código e verificou que ele é aceitável. Ele informa isso dizendo que foi "realizada a compilação" ("Done Compiling") e que o tamanho do seu sketch é 450 bytes. O IDE também está informando que o tamanho máximo disponível para programar é 32.256 bytes, de modo que você ainda tem muito espaço para ampliar o seu sketch.

Agora, vamos examinar essas linhas padronizadas de código. Elas serão sempre o ponto de partida de todos os sketches que você vier a escrever. Há algumas coisas novas aqui, como por exemplo a palavra **void** e algumas chaves. Primeiro vamos tratar da palavra **void**.

A linha **void setup()** significa que você está predefinindo uma função de nome **setup** (inicialização). No Arduino, algumas funções já estão definidas para você, como **digitalWrite** e

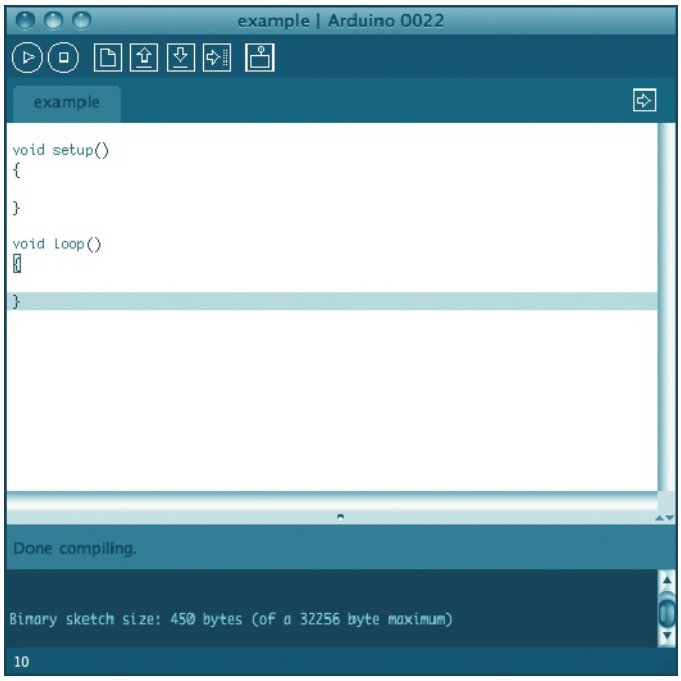

Figura 3-4 Um sketch para ser compilado.

delay. No entanto, você deverá ou poderá definir outras funções que serão usadas por você. As funções **setup** e **loop** são duas que você deve definir em todos os seus sketches.

O importante a ser compreendido aqui é que você não está chamando **setup** ou **loop** (laço de repetição) do mesmo modo que chamaria **digitalWrite**. Na realidade, você está criando essas funções de modo que o próprio sistema Arduino possa chamá-las. Esse é um conceito difícil de assimilar, mas uma forma de imaginá-lo é como uma definição em um documento legal.

Em muitos documentos legais, há uma seção de "definições" que pode dizer, por exemplo, algo como o seguinte:

```
Definições.

O Autor: A pessoa ou pessoas responsáveis pela criação do livro
```

Definindo um termo dessa forma – por exemplo, simplesmente usando a palavra "autor" como uma abreviação para "A pessoa ou pessoas responsáveis pela criação do livro" – os advogados podem tornar os seus documentos mais curtos e legíveis. As funções são muito semelhantes a essas definições. Você define uma função que você ou o próprio sistema poderá então usar em outros locais dos seus sketches.

Voltando à palavra **void**, essas duas funções (**setup** e **loop**) não retornam valor algum, diferentemente do que acontece com algumas outras funções. Portanto, você deve dizer que elas nada devem retornar. Para isso você deve usar a palavra-chave **void** (que em inglês significa

vazio). Se você imaginar uma função de nome **sin** (seno) que executa a função trigonométrica desse mesmo nome, então essa função fornece ou retorna um valor. O valor retornado que deve ser usado como resultado da chamada da função é o seno do ângulo que foi passado como argumento durante a chamada da função.

Assim como uma definição usa palavras para definir um termo, nós primeiro escrevemos as funções em C para depois chamá-las de dentro dos nossos programas em C.

Após a palavra especial **void**, vem o nome da função e os parênteses dentro dos quais estão os argumentos. No nosso caso não há argumentos, mas mesmo assim nós devemos incluir os parênteses porque nós estamos definindo uma função e não a chamando. Assim, precisamos dizer o que acontece quando a função é efetivamente chamada.

Quando a função é chamada, as coisas que devem acontecer são colocadas dentro de chaves. As chaves e o código contido dentro delas são conhecidos como um *bloco* de código. Esse é um conceito que você voltará a encontrar mais adiante.

Observe que, na realidade, embora você deva definir as funções **setup** e **loop**, você não precisa incluir linha de código nelas. Entretanto, se você deixar de incluir algum código, o seu sketch será um tanto maçante, sem graça.

» *Blink (pisca-pisca) – novamente!*

A razão de o Arduino ter as funções de **setup** e **loop** é para separar as coisas que devem acontecer uma única vez – quando o Arduino começa a executar o sketch – das coisas que devem se repetir continuamente.

Quando o sketch começa a ser executado, a função **setup** é chamada uma única vez. Vamos acrescentar algumas linhas para que o LED da placa pisque. Inclua mais linhas no seu sketch como mostrado a seguir. Depois, faça o upload para a sua placa.

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

void loop()
{
}
```

A própria função **setup** chama duas funções internas ou embutidas, **pinMode** e **digitalWrite**. Você já conhece a função **digitalWrite**, mas a **pinMode** é nova. A função **pinMode** (modo do pino) define o modo de um pino funcionar, como entrada ou saída. Portanto, o processo de acender um LED exige, na realidade, duas etapas. Primeiro, você define que o pino 13 deverá ser uma saída e, em seguida, você precisa fazer sua saída estar em nível alto (5V).

Quando você executa esse sketch, você verá que o LED da sua placa acende e depois permanece assim. Como isso não é muito empolgante, vamos tentar pelo menos fazer que ele pisque. Para tanto, vamos acendê-lo e apagá-lo na função **loop** e não na função **setup**.

Você pode chamar a **pinMode** de dentro da função **setup** porque você precisa chamá-la apenas uma única vez. Esse sketch também funcionaria se você movesse essa chamada para dentro do loop de repetição. Na realidade, não há necessidade disso e é uma boa prática de programação fazer as coisas apenas uma só vez quando é necessário que elas sejam feitas apenas uma única vez. Assim, modifique o seu sketch deixando-o como segue:

```
void setup()
{
  pinMode(13, OUTPUT);
}
void loop()
{
 digitalWrite(13, HIGH);
  delay(500);
 digitalWrite(13, LOW);
}
```

Execute esse sketch e veja o que acontece. Talvez não seja bem o que você está esperando. Basicamente, o LED permanece aceso o tempo todo. Por que isso está acontecendo?

Tente ir passo a passo executando mentalmente uma linha de cada vez do sketch:

1. Execute **setup** e faça o pino 13 ser uma saída.
2. Execute **loop** e faça o pino 13 passar para nível HIGH, ou alto (LED acende).
3. Execute um retardo (delay) de meio segundo.
4. Faça o pino 13 passar para nível LOW, ou baixo (LED apaga).
5. Execute novamente **loop**, voltando para o passo 2 e fazendo o nível do pino 13 ser alto, ou HIGH (LED acende).

O problema está entre os passos 4 e 5. O que está acontecendo é que o LED está sendo apagado, mas a primeira coisa que acontece logo em seguida é que ele é novamente aceso. Isso ocorre tão rapidamente que o LED parece estar aceso todo o tempo.

O chip microcontrolador do Arduino pode realizar 16 milhões de instruções por segundo. Não se trata de 16 milhões de comandos em linguagem C, mas mesmo assim ainda é muito rápido. Portanto, o nosso LED ficará apagado somente por uns poucos milionésimos de segundo.

Para resolver o problema, depois de apagar o LED você precisa acrescentar outro retardo. Agora, o seu código deverá ser como o seguinte:

```
// sketch 03-01

void setup()
{
  pinMode(13, OUTPUT);
}
void loop()
{
 digitalWrite(13, HIGH);
 delay(500);
 digitalWrite(13, LOW);
 delay(500);
}
```

Tente novamente. O seu LED deverá ficar piscando alegremente uma vez por segundo.

Na parte superior da listagem de comandos, você deve ter observado um comentário dizendo "sketch 3-01." Para poupar você de escrevê-lo, nós já colocamos à sua disposição no site do livro todos os sketches com esses comentários no início de cada um. Você poderá baixá-los de http://www.arduinobook.com.

Variáveis

No exemplo de Blink (pisca-pisca), você usou o pino 13 e fez referência a ele em três locais. Se você decidisse usar um pino diferente, então você teria que alterar o código nesses três lugares. Do mesmo modo, se você quisesse mudar a velocidade do pisca-pisca, que é controlado pelo argumento do retardo (delay), você teria que trocar o 500 por outro valor em diversos lugares.

As variáveis podem ser entendidas como um processo que dá um nome a um número. Na realidade, elas podem fazer muito mais do que isso, mas por enquanto você irá usá-las com essa finalidade.

Quando você está definindo uma variável em C, você deve especificar o seu tipo. Nós queremos que cada variável seja um número inteiro, o que corresponde ao atributo denominado **int** em C. Assim, para definir uma variável de nome **ledPin** (pino do LED) com o valor 13, você precisa escrever o seguinte:

```
int ledPin 13;
```

Observe que, como **ledPin** é um nome, então as mesmas regras dos nomes das funções devem ser aplicadas. Assim, não deve haver espaços em branco no nome e, se o nome for composto por múltiplas palavras, a convenção é começar a primeira palavra com uma letra minúscula e iniciar as demais palavras com letras maiúsculas.

Vamos incluir isso no seu sketch de Blink (pisca-pisca) como segue:

```
// sketch 03-02
int ledPin = 13;
int delayPeriod = 500;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
 digitalWrite(ledPin, HIGH);
 delay(delayPeriod);
 digitalWrite(ledPin, LOW);
 delay(delayPeriod);
}
```

Nós também usamos uma outra variável de nome **delayPeriod** (período ou duração do retardo).

Em todos os locais do sketch onde você antes fazia referência a 13, agora você está fazendo referência a **ledPin** e, em todos os locais onde antes você fazia referência a 500, agora você está fazendo referência a **delayPeriod**.

Se você desejar fazer o sketch piscar mais rapidamente, você pode simplesmente mudar o valor de **delayPeriod** em um único lugar. Tente mudar o valor para 100 e execute o sketch na sua placa de Arduino.

Há outras coisas engenhosas que você pode fazer com as variáveis. Vamos modificar o seu sketch de modo que o pisca-pisca comece muito rapidamente e aos poucos vai se tornando mais e mais lento, como se o Arduino estivesse ficando cansado. Para fazer isso, tudo o que você precisa fazer é somar alguma coisa à variável **delayPeriod** toda vez que o LED dá uma piscada.

Modifique o sketch e acrescente uma linha no final da função **loop**, como aparece na listagem seguinte. Depois, execute o sketch.

```
// sketch 03-03
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
 digitalWrite(ledPin, HIGH);
 delay(delayPeriod);
 digitalWrite(ledPin, LOW);
 delay(delayPeriod);
 delayPeriod = delayPeriod + 100;
}
```

Agora o seu Arduino está fazendo aritmética. Sempre que a função **loop** é chamada, ela executa o pisca-pisca normal do LED e, em seguida, soma 100 à variável **delayPeriod**. Em breve, voltaremos a fazer mais aritmética. Antes, contudo, você precisa de algo melhor do que simplesmente um LED piscando para ver o que o Arduino pode fazer.

Experimentos em C

Você precisa de um meio que permita testar os seus experimentos em C. Uma forma é colocar dentro da função **setup** o trecho em C que você quer testar, em seguida executar o sketch no Arduino e então fazer o Arduino mostrar o resultado por meio de algo denominado "Serial Monitor" (Monitor Serial), como está mostrado nas Figuras 3-5 e 3-6.

O Serial Monitor faz parte do IDE do Arduino. Para acessá-lo, na barra de ferramentas você deve clicar no ícone que está mais à direita. A sua finalidade é atuar como um canal de comunicação entre o computador e o Arduino. Você escreve uma mensagem na área de entrada de texto na parte superior do Serial Monitor e, quando você aperta Return ou clica em Send (enviar), a sua mensagem é transferida para o Arduino. Por outro lado, quando o Arduino tem

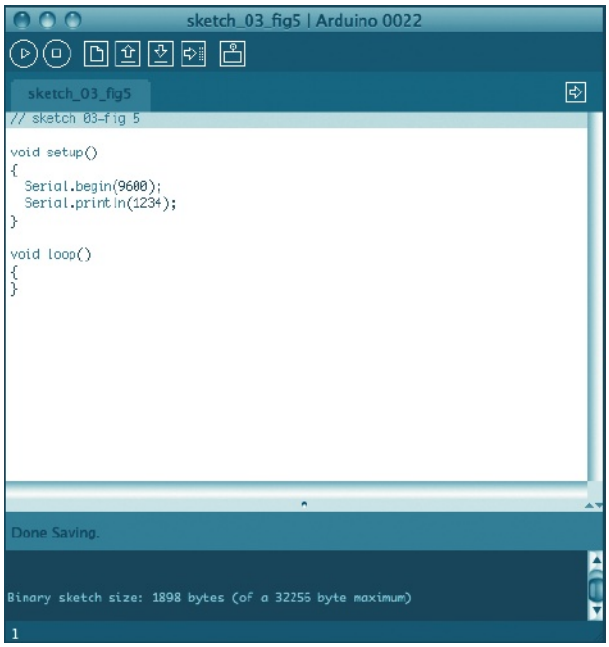

Figura 3-5 Escrevendo C no setup.

algo a dizer, a mensagem aparece no Serial Monitor. Em ambos os casos, a informação é enviada através da porta USB.

Como seria de esperar, há uma função embutida que pode ser usada em seus sketches para enviar uma mensagem ao Serial Monitor. Essa função é denominada **Serial.println**. Ela espera um único argumento, isto é, a informação que você deseja enviar. Geralmente essa informação é uma variável.

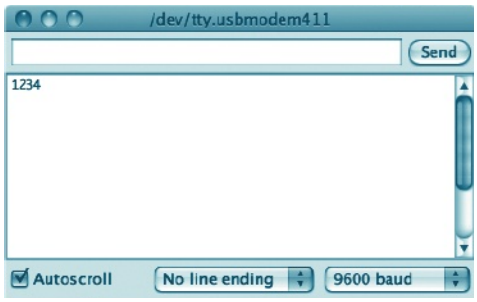

Figura 3-6 O Serial Monitor.

Você usará esse mecanismo para testar algumas coisas que você pode fazer com as variáveis e a aritmética em C. Sinceramente, esse é o único modo de ver os resultados dos seus experimentos em C.

Variáveis numéricas e aritméticas

A última coisa que você fez para que o intervalo de pisca-pisca aumentasse gradativamente foi acrescentar a seguinte linha ao sketch de pisca-pisca:

```
delayPeriod = delayPeriod + 100;
```

Examinando mais de perto, essa linha consiste em um nome de variável, seguido de um sinal de igual e então o que costuma ser denominado uma expressão (**delayPeriod + 100**). O sinal de igual faz algo denominado atribuição, isto é, ele atribui um novo valor a uma variável e o valor atribuído é determinado pelo que vem após o sinal de igual e antes do ponto e vírgula. Neste caso, o novo valor a ser dado à variável **delayPeriod** é o valor anterior de **delayPeriod** mais 100.

Para ver o que o Arduino pode fazer, vamos testar esse novo mecanismo. Para isso, vamos entrar com o sketch seguinte, executá-lo e abrir o Serial Monitor:

```
// sketch 03-04
void setup()
{
   Serial.begin(9600);

   int a = 2;
   int b = 2;
   int c = a + b;
   Serial.println(c);

}

void loop()

{}
```

A Figura 3-7 mostra o que deverá ser visto no Serial Monitor depois da execução desse código.

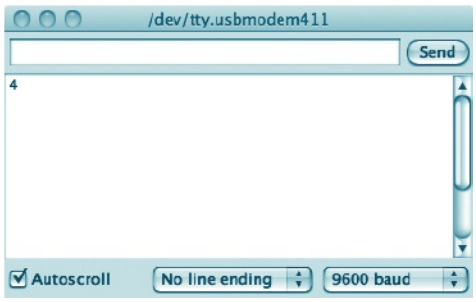

Figura 3-7 Aritmética simples.

Agora vamos examinar um exemplo ligeiramente mais complexo. A fórmula para converter uma temperatura de graus Celsius para graus Fahrenheit é multiplicar por 9, dividir por 5 e então somar 32. Assim, você pode escrever um sketch como segue:

```
// sketch 03-05
void setup()
{
   Serial.begin(9600);
   int degC = 20;
  int degF;
   degF = degC * 9 / 5 + 32;
   Serial.println(degF);
}
void loop()
{}
```

Aqui há algumas coisas que devem ser observadas. Primeiro, veja a linha seguinte:

```
int degC = 20;
```

Quando nós escrevemos essa linha, nós estamos, na realidade, fazendo duas coisas: estamos declarando uma variável **int** de nome **degC** (grau Celsius) e estamos dizendo que o seu valor inicial é 20. Alternativamente, você poderia ter separado essas duas coisas e ter escrito o seguinte:

```
int degC;

degC = 20;
```

Qualquer variável deve ser declarada somente uma vez, basicamente para dizer ao compilador qual é o tipo da variável – neste caso, **int**. Entretanto, você poderá atribuir valores a uma variável tantas vezes quanto desejar:

```
int degC;
degC = 20;
degC = 30;
```

Assim, no exemplo da conversão de Celsius para Fahrenheit, você está definindo a variável **degC** e dando-lhe um valor inicial de 20, mas quando você define a variável **degF** (grau Fahrenheit) ela não tem um valor inicial. O seu valor é atribuído na linha seguinte, de acordo com a fórmula de conversão. Após, ela é enviada para o Serial Monitor para que você possa vê-la.

Examinando a expressão, você pode ver que o asterisco (*) é usado na multiplicação e a barra (/) na divisão. Os operadores aritméticos +, –, * e / têm uma ordem de precedência, isto é, primeiro são feitas as multiplicações, a seguir as divisões e finalmente as somas e subtrações. Isso está de acordo com o que se costuma fazer em aritmética. Entretanto, algumas vezes o uso de parênteses nas expressões torna as operações mais claras. Assim, por exemplo, você poderia escrever o seguinte:

```
degF = ((degC * 9) / 5) + 32;
```

As expressões que você escreve podem ser tão longas e complexas quanto for necessário e, além dos operadores aritméticos usuais, há outros operadores menos usados e uma grande coleção de diversas funções matemáticas que estão disponíveis. Mais adiante, você irá conhecê-las.

Comandos

A linguagem C tem diversos comandos embutidos. Nesta seção, iremos explorar alguns deles e ver como podem ser usados em seus sketches.

if

Em nossos sketches até agora, foi suposto que as linhas de programação são executadas ordenadamente uma depois da outra, sem exceções. Mas, o que acontecerá se você não quiser que seja assim? Como fica se você quiser executar apenas uma parte do sketch quando uma dada condição é verdadeira?

Vamos voltar ao exemplo do LED pisca-pisca que aos poucos fica mais lento. Como está, ele se torna mais e mais lento até que cada intervalo de pisca-pisca esteja se estendendo por horas. Vamos ver como podemos mudá-lo para que, quando tiver chegado a uma determinada lentidão, volte à sua velocidade rápida inicial.

Para isso, você deve usar o comando **if** (se). O sketch* modificado é o seguinte. Experimente-o.

```
// sketch 03-06
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
  digitalWrite(ledPin, LOW);
   delay(delayPeriod);
   delayPeriod = delayPeriod + 100;
   if (delayPeriod > 3000)
    {
     delayPeriod = 100;
    }
}
```

O comando **if** é parecido com uma definição de função, mas essa semelhança é apenas superficial. A palavra dentro dos parênteses não é um argumento. É o que se denomina *uma condição*. No caso, a condição é que a variável **delayPeriod** (período do retardo) tenha um valor maior que 3.000. Se isso for verdadeiro, então os comandos dentro das chaves serão executados. Neste caso, o código fará o valor de **delayPeriod** voltar a 100.

Se a condição não for verdadeira, então o Arduino simplesmente irá adiante com o próximo comando. No caso, nada há após o "if", de modo que o Arduino executará novamente a função **loop**.

* N. de T.: Pelas razões expostas em nota anterior na Introdução, se você estiver usando os sketches baixados do site do autor, é possível que você encontre pequenas diferenças fáceis de entender.

A execução mental da sequência de eventos ajudará você a compreender o que está acontecendo. Assim, o que ocorre é o seguinte:

1. O Arduino executa **setup** e inicializa o pino do LED para que seja uma saída.
2. O Arduino começa a executar o **loop**.
3. O LED acende.
4. Ocorre um retardo.
5. O LED apaga.
6. Ocorre um retardo.
7. O valor 100 é somado a **delayPeriod.**
8. Se o período do retardo for maior que 3.000, torne-o novamente igual a 100.
9. Volte ao passo 2.

Nós usamos o símbolo >, que significa maior do que. Esse é um exemplo dos denominados operadores de comparação. A tabela ao final da página apresenta um resumo desses operadores:

Para comparar dois números, você usa o comando ==. Esse sinal duplo de igual é facilmente confundido com o caractere =, que é usado para atribuir valores às variáveis.

Há uma outra forma de **if** que permite fazer uma coisa quando a condição é verdadeira e uma outra quando é falsa. Usaremos isso em alguns exemplos práticos mais adiante neste livro.

» for

Frequentemente, além de executar comandos diferentes em circunstâncias diferentes, é possível que você deseje executar uma série de comandos um certo número de vezes em um programa. Você já viu um modo de fazer isso, usando a função **loop**. Depois que todos os comandos dentro da função **loop** terminam de ser executados, automaticamente voltam a ser executados novamente. Algumas vezes, entretanto, você pode necessitar de um controle com mais recursos do que esse.

Operador	Significado	Exemplos	Resultado
<	Menor do que	9 < 10	verdadeiro
		10 < 10	falso
>	Maior do que	10 > 10	falso
		10 > 9	verdadeiro
<=	Menor do que ou igual a	9 <= 10	verdadeiro
		10 <= 10	verdadeiro
>=	Maior do que ou igual a	10 >= 10	verdadeiro
		10 >= 9	verdadeiro
==	Igual a	9 == 9	verdadeiro
!=	Não igual a, diferente de	9 != 9	falso

Assim, por exemplo, vamos supor que você deseje escrever um sketch que faz um LED piscar 20 vezes, em seguida fazer uma pausa de 3 segundos e então começar tudo de novo. Você poderia fazer isso simplesmente repetindo o mesmo código sem parar dentro da sua função **loop**, como ilustrado a seguir:

```
// sketch 03-07
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
 digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
 digitalWrite(ledPin, LOW);
  delay(delayPeriod);
 digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
 digitalWrite(ledPin, LOW);
  delay(delayPeriod);
 digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
 digitalWrite(ledPin, LOW);
  delay(delayPeriod);
// repita as 4 linhas anteriores mais 17 vezes
  delay(3000);
}
```

Para tanto, foi necessário escrever um sketch bem longo. Por outro lado, há formas melhores de fazer isso. Vamos começar examinando como você pode usar um laço de **for** (para) e, em seguida, como escrever o sketch usando um contador e um comando **if**.

Como você pode ver a seguir, o sketch que realiza isso usa um laço de **for** e é bem menor e mais fácil de manter do que o sketch anterior.

```
// sketch 03-08
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
   for (int i = 0; i < 20; i ++)
   {
  digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
```

```
  digitalWrite(ledPin, LOW);
    delay(delayPeriod);
   }
 delay(3000);
}
```

O laço de **for** assemelha-se a uma função de três argumentos, embora aqui esses argumentos estejam separados por ponto e vírgula e não por vírgulas. Essa é uma peculiaridade da linguagem C. O compilador lhe avisará quando você usá-lo de forma errada.

Após o **for**, a primeira coisa dentro dos parênteses é uma declaração de variável. Essa variável é usada como contador e um valor inicial está sendo-lhe atribuído – neste caso, 0.

A segunda parte é uma condição que deve ser verdadeira para que você permaneça dentro do laço de **for**. Neste caso, você permanecerá dentro do laço de **for** enquanto **i** for menor que 20, mas, logo que **i** for igual ou maior que 20, o programa interromperá o que estiver fazendo dentro do **for**.

A terceira parte indica o que deverá ser feito após você ter executado todos os comandos que estão dentro do laço de **for**. Neste caso, o **i** deve ser incrementando de 1. Sendo assim, teremos que, após 20 repetições do laço de **for**, o **i** deixará de ser menor que 20, obrigando o programa a sair de dentro do **for**.

Tente entrar com o código anterior e execute-o. A única maneira de se familiarizar com a sintaxe e com todos os detalhes incômodos de pontuação é escrevendo o código e fazendo o compilador mostrar quando você faz algo errado. Com a prática, todas essas coisas começarão a fazer sentido.

Uma desvantagem potencial dessa abordagem é que a função **loop** vai precisar de muito tempo. Isso não é um problema neste caso, porque tudo que o sketch está fazendo é comandar o pisca-pisca de um LED. Mas frequentemente, em um sketch, a função **loop** também estará verificando quais botões foram pressionados e se alguma comunicação serial foi recebida. Se o processador estiver ocupado dentro de um laço de **for**, então ele não poderá fazer essas verificações. Geralmente, é uma boa ideia fazer que a função **loop** seja executada tão rapidamente quanto possível para que ela seja executada o maior número possível de vezes.

O sketch seguinte mostra como conseguir isso:

```
// sketch 03-09
int ledPin = 13;
int delayPeriod = 100;
int count = 0;
void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
 digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
 digitalWrite(ledPin, LOW);
  delay(delayPeriod);
 count ++;
```

```
  if (count == 20)
  {
    count = 0;
     delay(3000);
  }
}
```

Você deve ter observado a seguinte linha:

```
count ++;
```

Essa é simplesmente uma forma abreviada em C para

```
count = count + 1;
```

Portanto, para cada vez que o **loop** for executado, serão necessários um pouco mais de 200 milissegundos, menos na vigésima vez, quando será necessário o mesmo tempo mais os três segundos de retardo entre cada conjunto de 20 ciclos de pisca-pisca do LED. De fato, para algumas aplicações, mesmo assim é muito lento. Os puristas diriam que você não deveria usar nem mesmo **delay**. A melhor solução depende de cada aplicação.

» while

Uma outra maneira de fazer um laço de repetição em C é usando o comando **while** (enquanto) no lugar do comando **for**. Você pode conseguir a mesma coisa que obteve no exemplo do **for**, usando um comando **while** como segue:

```
int i = 0;
while (i < 20)
{
  digitalwrite (ledPin, HIGH);
   delay(delayPeriod);
  digitalwrite (ledPin, LOW);
   delay(delayPeriod);
  i ++;
}
```

A expressão em parênteses após o comando **while** deve ser verdadeira para que a execução permaneça dentro do laço de **while**. Quando não for mais verdadeira, então o sketch passará a executar os comandos que estão após o sinal de "fecha chave".

» *A diretiva #define*

Para valores constantes, como atribuições de pinos que não mudam durante a execução de um sketch, há uma alternativa ao uso de variáveis. Você pode usar um comando denominado diretiva **#define**. Essa diretiva faz um valor ser atribuído a um nome. Antes que a compilação do seu sketch seja realizada, esse nome será substituído pelo valor atribuído em todos os lugares do sketch onde ele aparece.

Como exemplo, uma atribuição de pino de LED poderia ser definida como segue:

```
#define ledPin 13
```

Observe que a diretiva **#define** não usa o sinal "=" entre o nome e o valor. Ela não precisa nem de um ";" no final. Isso acontece porque, na verdade, essa diretiva não faz parte da linguagem C em si. Ela é uma diretiva de pré-compilação que é executada antes da compilação.

Na opinião do autor, essa maneira é mais difícil de ler do que quando se usa uma variável. No entanto, ela tem a vantagem de não precisar de memória para ser armazenada. É algo para considerar quando há pouca memória disponível.

Conclusão

Esse capítulo permitiu que você desse os primeiros passos em C. Você pode fazer os LEDs piscarem de diversas formas empolgantes e conseguir que o Arduino lhe envie resultados através da porta USB usando a função **Serial.println**. Você aprendeu como usar os comandos **if** e **for** para controlar a ordem de execução de seus comandos e também como o Arduino faz aritmética.

No próximo capítulo, você examinará mais de perto as funções. O capítulo também apresentará outros tipos de variáveis além do tipo **int** que você usou neste capítulo.

capítulo 4

Funções

Este capítulo dedica-se principalmente aos tipos de funções que você mesmo pode escrever, em vez das funções internas tais como **digitalWrite** e **delay** que já foram predefinidas para você.

Você precisa saber como escrever as suas próprias funções porque, quando os sketches começam a se tornar um pouco complicados, as suas funções **setup** e **loop** irão crescer até se tornarem longas e complexas, ficando difícil entender como funcionam.

Objetivos deste capítulo

- Entender o que é uma função
- Aprender como passar parâmetros a funções
- Conhecer os tipos de variáveis
- Mostrar formas de estruturar o código

O maior problema do desenvolvimento de software de qualquer tipo é como lidar com a complexidade. Os melhores programadores escrevem software fácil de ser lido e compreendido, requerendo pouca explicação.

As funções são uma ferramenta-chave para criar sketches de fácil compreensão, que podem ser modificados sem dificuldade nem risco de a coisa toda se transformar em uma grande confusão.

O que é uma função?

Uma função é algo um pouco parecido com um programa dentro de um programa. Você pode usá-la para "empacotar" algumas coisas que você deseja realizar. Uma função definida por você poderá ser chamada de qualquer lugar do seu sketch e contém as suas próprias variáveis e a sua própria lista de comandos. Quando os seus comandos terminam de ser efetuados, a execução do sketch continua no ponto imediatamente após a linha no código em que a função foi chamada.

O código que faz um diodo emissor de luz (LED) piscar é um ótimo exemplo de um código que pode ser transformado em função. Assim, vamos modificar o nosso sketch básico que "faz o LED piscar 20 vezes" de tal modo que possamos usá-lo como uma função de nome **flash** (piscar) criada por nós.

```
// sketch 04-01
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
   for (int i = 0; i < 20; i ++)
   {
      flash();
   }
   delay(3000);
}

void flash()

{
  digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
  digitalWrite(ledPin, LOW);
   delay(delayPeriod);
}
```

Assim, tudo o que realmente fizemos aqui foi retirar de dentro do laço de **for** as quatro linhas de código que fazem o LED piscar e colocá-las dentro de uma função de nome **flash** (piscar). Agora você poderá fazer o LED piscar a qualquer momento que desejar. Bastará simplesmente

escrever **flash()** para que a nova função seja chamada. Observe os parênteses vazios após o nome da função. Isso indica que a função não necessita de parâmetros. O valor do retardo que ela usa é fixado pela mesma função **delayPeriod** que você usou antes.

Parâmetros

Quando você divide o seu sketch em funções, frequentemente vale a pena pensar sobre o que poderia ser feito por uma função. No caso de **flash**, isso é bem óbvio. Mas, desta vez, vamos dar parâmetros a essa função, dizendo o número de vezes (numFlashes) que o LED deve piscar e qual deve ser a duração (d) de cada ciclo de pisca-pisca. Examine o código seguinte. Logo após, eu explicarei o funcionamento dos parâmetros com um pouco mais de detalhes.

```
// sketch 04-02
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  flash(20, delayPeriod);
   delay(3000);
}

void flash(int numFlashes, int d)
{
   for (int i = 0; i < numFlashes; i ++)
   {
    digitalWrite(ledPin, HIGH);
      delay(d);
    digitalWrite(ledPin, LOW);
      delay(d);
   }
}
```

Examinado a nossa função **loop**, vemos que ela tem apenas duas linhas. O grosso do trabalho foi movido para dentro da função **flash**. Ao chamar a função **flash**, observe que agora deveremos fornecer dois argumentos dentro dos parênteses.

Na parte inferior do sketch, no local onde a função é definida, nós temos que declarar o tipo das variáveis dentro dos parênteses. Nesse caso, ambas são do tipo **int**. Na realidade, estamos definindo novas variáveis. Entretanto, essas variáveis (**numFlashes** e **d**) podem ser usadas apenas dentro da função **flash**.

Essa é uma boa função porque ela contém tudo que é necessário para fazer um LED piscar. A única informação necessária, obtida fora da função, é o pino de conexão do LED. Se quisesse, isso também poderia ser transformado em um parâmetro – algo que vale a pena fazer quando você tem mais de um LED ligado ao Arduino.

Variáveis globais, locais e estáticas

Como mencionado antes, os parâmetros de uma função podem ser usados apenas dentro dela. Assim, se você escrever o código seguinte, você terá um erro:

```
void indicate(int x)
{
  flash(x, 10);
}
x = 15
```

Por outro lado, suponha que você escreva o seguinte:

```
int x;
void indicate(int x)
{
  flash(x, 10);
}
x = 15
```

Esse código não gera um erro de compilação. Entretanto, na realidade, você precisa ser cuidadoso porque agora você tem duas variáveis de nome **x**, com valores diferentes. Aquela que você declarou na primeira linha é uma *variável global*. É denominada *global* porque pode ser usada no programa em qualquer lugar que você desejar, mesmo dentro das funções.

Entretanto, como o mesmo nome de variável **x** também está sendo usado como parâmetro dentro da função, você não poderá usar a variável global **x** simplesmente porque, sempre que você se referir à variável **x** de dentro da função, a prioridade será desta versão "local" de **x**. Diz-se que o parâmetro **x** é uma sombra da variável global de mesmo nome. Isso pode levar a uma confusão quando você tenta depurar (debugar) um projeto.

Além da definição de parâmetros, você também pode definir variáveis que não são parâmetros e que serão usadas apenas dentro da função. Essas são as chamadas *variáveis locais*. Por exemplo,

```
void indicate(int x)
{
   int timesToFlash = x * 2;
  flash(timesToFlash, 10);
}
```

A variável local **timesToFlash**(vezes que o LED deve piscar) existirá apenas enquanto a função estiver sendo executada. Quando a função tiver executado o seu último comando, a variável desaparecerá. Isso significa que as variáveis locais não podem ser acessadas de nenhum outro lugar do programa, a não ser de dentro da função na qual foram definidas.

Assim, o código seguinte, por exemplo, causará um erro:

```
void indicate(int x)
{
   int timesToFlash = x * 2;
  flash(timesToFlash, 10);
}
timesToFlash = 15;
```

Os programadores experientes geralmente tratam as variáveis globais com desconfiança. A razão é que elas vão contra o princípio do encapsulamento. A ideia do *encapsulamento* é que você deve criar um pacote, embrulhando em um único volume tudo o que tem a ver com alguma coisa em particular. Sendo assim, as funções servem muito bem para o encapsulamento. O problema com as "globais" (como são frequentemente denominadas as variáveis globais) é que geralmente são definidas no início de um sketch, podendo então ser usadas em todo o sketch. Algumas vezes, há uma razão perfeitamente legítima para isso. Outras vezes, as pessoas as usam por preguiça, quando na verdade seria muito mais apropriado passar parâmetros. De nossos exemplos até agora, **ledPin** é um bom exemplo de variável global. Ela também se mostra muito conveniente, podendo ser facilmente localizada no início do sketch. Isso facilita o trabalho de modificá-la. Mesmo que você possa alterá-la e compilar novamente o sketch, na realidade **ledPin** opera como se fosse uma constante, porque é improváve l que seja alterada durante a execução do sketch. Por essa razão, talvez seja preferível usar o código **#define** que descrevemos no Capítulo 3.

Uma outra característica de uma variável local é que o seu valor é inicializado sempre que a função é executada. Em nenhum lugar isso é mais verdadeiro (e frequentemente mais inconveniente) do que na função **loop** de um sketch de Arduino. Vamos tentar usar uma variável local em vez de uma global em um dos exemplos do capítulo anterior:

```
// sketch 04-03
int ledPin = 13;
int delayPeriod = 250;
void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
  int count = 0;
  digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
  digitalWrite(ledPin, LOW);
   delay(delayPeriod);
  count ++;
  if (count == 20)
  {
    count = 0;
     delay(3000);
  }
}
```

O sketch 4-03 baseia-se no sketch 3-09. Para fazer a contagem do número de vezes que o LED pisca, estamos tentando usar uma variável local em vez de global.

Esse sketch está com falhas. Ele não funciona porque toda vez que **loop** é executado, a variável de contagem **count** recebe o valor 0 novamente, de modo que **count** nunca alcança 20 e o LED permanece piscando para sempre. A única razão para termos feito **count** ser uma variável global desde o início foi para evitar que seu valor fosse zerado. No entanto, o

único local em que usamos **count** é dentro da função **loop**. Portanto, esse é o lugar onde ela deveria estar.

Felizmente, há um mecanismo em C que permite contornar essa dificuldade. É a palavra-chave **static** (estática). Em uma função, quando você usa a palavra-chave **static** na frente de uma declaração de variável, ela tem o efeito de inicializar a variável apenas na primeira vez em que a função é executada. Perfeito! Isso é exatamente o que precisamos nesta situação. Poderemos manter a nossa variável na função em que é usada, sem que ela volte a 0 toda vez que a função é executada. O sketch 4-04 mostra isso em funcionamento:

```
// sketch 04-04
int ledPin = 13;
int delayPeriod = 250;
void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  static int count = 0;
 digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
 digitalWrite(ledPin, LOW);
  delay(delayPeriod);
 count ++;

 if (count == 20)
 {
   count = 0;
    delay(3000);
 }
}
```

Retornando valores

A ciência da computação, como disciplina acadêmica, tem como ancestrais a matemática e a engenharia. Essa herança estende-se a muitos dos nomes associados com a programação. A própria palavra *função* é um termo matemático. Na matemática, a entrada de uma função (o argumento) determina completamente a saída. Nós havíamos escrito funções que recebiam uma entrada, mas nenhuma delas nos devolvia ou retornava algum valor. Todas as nossas funções eram "void" (vazias, no sentido de que nada devolvem). Se uma função retorna um valor, então você deve especificar um tipo para o valor de retorno.

Vamos examinar como escrever uma função que recebe a temperatura em graus Celsius e retorna o equivalente em graus Fahrenheit:

```
int centToFaren(int c)
{
   int f = c * 9 / 5 + 32;
  return f;
}
```

Agora a definição da função começa com **int** em vez de **void** para indicar que a função irá retornar um valor **int** para quem vier a chamá-la. Isso poderia ser usado em um código curto como o que está a seguir, que retorna em graus Fahrenheit o valor correspondente a uma temperatura agradável (pleasantTemp) de 20 graus Celsius:

```
int pleasantTemp = centToFaren(20);
```

Qualquer função não "void" deve conter um comando de **return** (retornar). Se você não incluí-lo, o compilador avisará que ele está fazendo falta. Você pode ter a mesma função com mais de um **return**. Isso poderá ocorrer se houver um comando **if** com ações alternativas podendo ser executadas com base em alguma condição. Alguns programadores não veem isso com bons olhos, mas se suas funções forem pequenas (como todas as funções deveriam ser), então essa prática não será um problema.

O valor após o **return** pode ser uma expressão, não se limitando a ser somente o nome de uma variável. Assim, você pode compactar o exemplo anterior no seguinte:

```
int centToFaren(int c)
{
   return (f = c * 9 / 5 + 32);
}
```

Se a expressão retornada for maior do que simplesmente um nome de variável, então ela deverá estar dentro de parênteses, como no exemplo anterior.

Outros tipos de variáveis

Até agora, todos os nossos exemplos de variáveis têm sido variáveis **int**. De longe esse é o tipo de variável mais comumente usado, mas você também deveria conhecer outros tipos.

float

Um desses tipos, que é relevante para o exemplo anterior de conversão de temperatura, é o **float** (flutuante). Esse tipo de variável representa os números de ponto flutuante, isto é, números que podem ter um ponto decimal*, tal como 1.23. Esse tipo de variável é necessário quando os números inteiros não são precisos o suficiente.

Observe a fórmula seguinte:

```
f = c * 9 / 5 + 32
```

Se você der o valor 17 a **c**, então **f** será 17 * 9 / 5 + 32 ou 62.6. Mas, se **f** for **int**, então o valor será truncado transformando-se no valor inteiro 62.

* N. de T.: É importante ter em conta que, quando trabalhamos com o Arduino, nós estamos em um ambiente de linguagem C. Por essa razão, usamos o ponto no lugar de vírgula. Assim, por exemplo, em C o número 1,23 é representado como 1.23. Os nossos raciocínios serão realizados em termos da linguagem C. Portanto, usaremos o ponto e não a vírgula nesses casos.

O problema torna-se ainda pior se não tomarmos cuidado com a ordem de fazer os cálculos. Por exemplo, suponha que fizéssemos primeiro a divisão, como segue:

```
f = (c / 5) * 9 + 32
```

Em termos matemáticos normais, o resultado ainda seria 62.6, mas se todos os números forem do tipo **int**, então os cálculos serão feitos como segue:

1. O 17 é dividido por 5, dando 3.4, que depois de truncado torna-se 3.
2. Então, o 3 é multiplicado por 9 e, em seguida, o 32 é adicionado, dando um resultado de 59, que é um valor bem distante do 62.6.

Em circunstâncias como essa, podemos usar variáveis do tipo **float**. No exemplo seguinte, vamos escrever novamente a função de conversão de temperatura com o uso de variáveis do tipo **float** ao invés de **int**.

```
float centToFaren(float c)
{
   float f = c * 9.0 / 5.0 + 32.0;
  return f;
}
```

Observe que acrescentamos .0 às nossas constantes. Isso assegura que o compilador irá tratá-las como variáveis do tipo **float** ao invés de **int**.

boolean

Uma variável booleana é uma variável lógica. Ela tem um valor que é ou verdadeiro ou falso.

Na linguagem C, o nome *boolean* (booleano) é escrito com um *b* minúsculo. O nome *boolean* vem do nome do matemático George Boole, inventor da lógica booleana, que é crucial para a ciência da computação.

Talvez não tenha se dado conta, mas antes você encontrou valores booleanos quando estávamos examinado o comando **if**. Uma condição em um comando **if**, como **(count==20)**, é na realidade uma expressão que fornece um resultado do tipo **boolean** (booleano). O operador == é denominado operador de comparação. Enquanto + é um operador aritmético que soma dois números, o == é um operador de comparação que compara dois números e retorna um valor lógico que é ou verdadeiro ou falso.

Você pode definir variáveis booleanas e usá-las como veremos a seguir, onde a variável "tooBig" (grandeDemais) é declarada como sendo do tipo booleano:

```
boolean tooBig = (x > 10);
if (tooBig)
{
  x = 5;
}
```

Os valores booleanos podem ser manipulados usando os operadores booleanos. Assim, do mesmo modo que você faz cálculos aritméticos usando os números, você também pode realizar operações lógicas usando os valores booleanos. Os operadores booleanos mais comumente usados são o **and** (e), escrito como **&&**, e o **or** (ou), escrito como ||.

AND

B \ A	falso	verdadeiro
falso	falso	falso
verdadeiro	falso	verdadeiro

OR

B \ A	falso	verdadeiro
falso	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro

Figura 4-1 Tabelas-verdade.

A Figura 4-1 mostra as tabelas-verdade para os operadores **and** e **or**.

Examinando as tabelas-verdade da Figura 4-1, você pode ver que, no caso do operador **and** (e), temos que, quando A e B são ambos verdadeiros, então o resultado é verdadeiro. Em caso contrário, o resultado é falso.

Por outro lado, para o operador **or** (ou), quando ou A ou B ou quando ambos, A e B, são verdadeiros, então o resultado é verdadeiro. Se nem A nem B são verdadeiros, então o resultado é falso.

Além de **and** e **or**, há o operador **not** (não), escrito como **!**. Você não se surpreenderá ao aprender que "não verdadeiro" é falso e "não falso" é verdadeiro.

Você pode combinar esses operadores formando expressões booleanas e usá-las em comandos **if**, como no seguinte exemplo:

```
if ((x > 10) && (x < 50))
```

Outros tipos de dados

Como você viu, os tipos de dados **int** e ocasionalmente **float** são bons para a maioria das situações. Entretanto, outros tipos numéricos podem ser úteis em algumas circunstâncias particulares. Em um sketch do Arduino, o tipo **int** usa 16 bits (dígitos binários). Isso permite que os números inteiros sejam representados entre –32767 e 32768.

A Tabela 4-1 resume também os outros tipos de dados disponíveis. Ela está sendo fornecida principalmente para ser usada como referência. Você usará alguns desses tipos à medida que avançar no livro.

Algo que deve ser levado em consideração é que, se o valor de uma variável estiver fora do intervalo de representação correspondente ao tipo de dado dessa variável, então coisas estranhas acontecerão. Assim, se o valor de uma variável do tipo byte for 255 e você acrescentar 1 a esse valor, você obterá 0. Mais alarmante, se você tiver uma variável **int** com o valor 32767 e você somar 1, você obterá –32768.

Até que você se sinta completamente confortável com todos esses tipos diferentes de dados, eu recomendo que você fique com o tipo **int**, porque funciona bem para muitas coisas.

Tabela 4-1 Tipos de dados em C

Tipo	Memória (bytes)	Intervalo	Observações
boolean	1	verdadeiro ou falso (0 ou 1)	
char	1	–128 até +128	Usado para representar um código de caractere ASCII (American Standard Code for Information Interchange)*. Por exemplo, *A* é representado como 65. Normalmente, os seus números negativos não são usados.
byte	1	0 até 255	Frequentemente usado na comunicação serial de dados. Veja o Capítulo 9.
int	2	–32768 até 32767	
unsigned int	2	0 até 65536	Pode ser usado para ter uma precisão extra, quando não há necessidade de números negativos. Use com cautela, porque a aritmética que usa o tipo **int** pode produzir resultados inesperados.
long	4	–2,147,483,648 até 2,147,483,647	Necessário apenas para representar números muito grandes.
unsigned long	4	0 até 4,294,967,295	Veja **unsigned int**.
float	4	–3.4028235E+38 até + 3.4028235E+38	
`double`	4	O mesmo que **float**	Normalmente, seriam 8 bytes com uma precisão mais elevada que **float** e um intervalo de representação maior. No Arduino, entretanto, é o mesmo que **float**.

Estilo de codificação

Na realidade, o compilador C não se importa com a forma de apresentação final do seu código quando você organiza a disposição das linhas. Para o compilador, você pode escrever tudo em uma única linha, usando pontos e vírgulas para separar os comandos. Entretanto, um código bem organizado, claro e limpo, é muito mais fácil de entender e manter do que um código pobremente apresentado. Nesse sentido, a leitura de um código é como a leitura de um livro: a sua formatação é importante.

Até certo ponto, a formatação é uma questão de gosto pessoal. Ninguém gosta de pensar que tem mau gosto, de modo que qualquer discussão sobre a formatação de um programa torna-se um assunto pessoal. Há programadores que, quando solicitados a fazer alguma coisa com o código de outra pessoa, começam fazendo uma nova formatação de todo o código de acordo com o seu estilo preferido de apresentação.

* N. de T.: ASCII (American Standard Code for Information Interchange ou Código Americano Padrão para Intercâmbio de Informação).

Para dar conta dessa questão, frequentemente são publicadas normas de codificação encorajando todos a apresentarem os seus códigos da mesma forma e a adotarem a "boa prática" quando estiverem escrevendo programas.

A linguagem C segue uma norma de apresentação que evoluiu com os anos. Em termos gerais, seremos fiéis a esse padrão neste livro.

Recuo

Nos exemplos de sketches que vimos, podemos observar frequentemente que o texto com o código do programa encontra-se afastado ou recuado da margem esquerda. Por exemplo, quando definimos a função **void**, a palavra-chave **void** ficou na margem esquerda, assim como o mesmo aconteceu com o sinal de "abre chave" na linha seguinte. Em seguida, a partir deste ponto até o sinal de "fecha chave", todo o texto está recuado. Na realidade, o tamanho do recuo não importa. Algumas pessoas usam dois espaços, outras quatro. Você também pode apertar a tecla TAB para fazer um recuo. Neste livro, usaremos dois espaços para o recuo.

Se você tiver um comando `if` dentro de uma definição de função, então as linhas que estão dentro das chaves do comando **if** serão novamente recuadas mais dois espaços, como no exemplo seguinte

```
void loop()

{ static int count = 0;
  count ++;
  if (count == 20)
  {
    count = 0;
     delay(3000);
  }
}
```

Você poderia colocar outro **if** dentro do primeiro **if**, acrescentando mais um nível de recuo e totalizando seis espaços desde a margem esquerda.

Isso tudo pode parecer um tanto trivial, mas se você examinar os sketches mal formatados de alguém, você verá como isso pode ser bem difícil.

Abrindo chaves

Há duas escolas de pensamento que opinam sobre o local onde o sinal de "abre chave" deve ser colocado em uma definição de função, comando **if** ou laço **for**. Uma forma é colocar o sinal de "abre chave" na linha imediatamente após o restante do comando, como temos feito até agora em todos os exemplos. A outra forma é colocá-lo na mesma linha, como segue:

```
void loop() {
  static int count = 0;
  count ++;
  if (count == 20) {
```

```
    count = 0;
     delay(3000);
  }
}
```

Esse estilo é muito usado na linguagem Java de programação, a qual tem muita coisa em comum com a sintaxe da linguagem C. Eu prefiro a primeira forma, que parece ser a forma mais comumente usada no mundo do Arduino.

Espaço em branco

O compilador ignora espaços, parágrafos e novas linhas. Limita-se apenas a usá-los como uma forma de separar as palavras do seu sketch. Assim, o exemplo seguinte será compilado sem problemas:

```
void loop() {static int
count=0;count++;if(
count==20){count=0;
delay(3000);}}
```

Esse código funciona, mas lhe desejamos boa sorte quando você tentar lê-lo.

Nos locais em que são feitas as atribuições, algumas pessoas escrevem o seguinte:

```
int a = 10;
```

Outros irão escrever como segue:

```
int a=10;
```

Não é importante qual dessas duas formas você usa, mas é uma boa ideia ser consistente. Eu uso a primeira forma.

Comentários

Os comentários são textos incluídos em seu sketch juntamente com o código real de programação, mas que, na realidade, não exercem função alguma. O único propósito dos comentários é ajudar a lembrar por que o código foi escrito do jeito como está. Um título também pode ser incluído usando uma linha de comentário.

O compilador irá ignorar completamente qualquer texto que tenha sido marcado como comentário. Neste livro, até agora, nós já incluímos títulos na forma de comentários bem no início de muitos sketches.

Há duas formas de sintaxe para os comentários:

- O comentário de linha única que começa com // e termina no final da linha
- O comentário de muitas linhas que começa com /* e termina com */

O exemplo seguinte ilustra ambas as formas de comentário.

```
/* Uma função loop não muito útil.
Escrita por: Simon Monk
Para ilustrar o conceito de comentário
*/
void loop() {
  static int count = 0;
  count ++;  // um comentário de linha única
  if (count == 20) {
    count = 0;
     delay(3000);
  }
}
```

Neste livro, eu quase que exclusivamente uso o formato de linha única para fazer comentários.

Os bons comentários ajudam a explicar como você usa o sketch ou como ele funciona. Poderão ser úteis quando outras pessoas vão usar o seu sketch, mas também serão igualmente úteis para você mesmo quando você tiver que examinar um sketch após algumas semanas sem usá-lo.

Em alguns cursos de programação, é dito que, quanto mais comentários, melhor fica. Os programadores muito experientes dirão que um código bem escrito requer poucos comentários porque é autoexplicativo. Você deve usar comentários pelas seguintes razões:

- Para explicar qualquer coisa em que você usou algum artifício ou que não seja imediatamente óbvia.
- Para descrever qualquer coisa que o usuário deve fazer e que não faça parte do programa como, por exemplo, **//este pino deve ser conectado ao transistor que controla o relé.**
- Para deixar uma mensagem para você mesmo como, por exemplo, **//ParaFazerDepois: ponha ordem nesta parte – está uma bagunça**.

Esse último caso ilustra uma técnica útil de **ParaFazerDepois** usando comentários. Frequentemente, os programadores colocam diversos comentários do tipo **ParaFazerDepois** em seu código. Isso permite que eles se lembrem de coisas que devem ser feitas depois. Sempre é possível encontrar todas as ocorrências de **//ParaFazerDepois** no seu programa. Para isso, você pode usar o recurso de busca (Find) do próprio ambiente de desenvolvimento integrado (IDE) do Arduino.

Os seguintes casos *não* são bons exemplos de razões para você usar comentários:

- Para explicar o que é óbvio. Por exemplo, **a = a + 1; // some 1 à variável a**.
- Para explicar um código mal escrito. Não faça comentários. Simplesmente escreva-o claramente desde o início.

Conclusão

Este capítulo foi um tanto teórico. Você teve que absorver alguns novos conceitos abstratos, voltados à organização de sketches na forma de funções e à adoção de um estilo de programação que irá poupar o seu tempo a longo prazo.

» capítulo 5

Arrays e strings

Depois da leitura do Capítulo 4, você ficou com uma ideia razoável de como estruturar os seus sketches para facilitar a sua vida. Se há alguma coisa que um bom programador gosta, é uma boa vida. Agora, a nossa atenção irá se voltar aos dados que você usa nos sketches.

O livro *Algoritmos + Estruturas de Dados = Programas* de Niklaus Wirth tem estado por aí já há muito tempo, mas ainda capta as essências da ciência da computação e da programação em particular. Posso recomendá-lo fortemente para qualquer um que foi fisgado pela programação. Ele também capta a ideia de que, para escrever um bom programa, você precisa pensar tanto no algoritmo (o que você faz) como na estrutura dos dados que você usa.

Você viu os comandos **loop** e **if** e também aquilo que se denomina o lado "algorítmico" da programação de um Arduino. Agora, você verá como estruturar os seus dados.

Objetivos deste capítulo

- Mostrar como estruturar dados em um sketch usando arrays e strings
- Reforçar a importância do uso de funções para estruturar um programa
- Mostrar a construção de um tradutor de código Morse

Arrays

Um array é uma maneira de organizar uma lista de valores. As variáveis que você usou até agora continham apenas um valor, usualmente um **int**. Por outro lado, um array contém uma lista de valores e você pode acessar qualquer um desses valores fornecendo a sua posição na lista.

Como na maioria das linguagens de programação, a linguagem C começa a indexação das posições com 0 e não com 1. Isso significa que o primeiro elemento é, na realidade, o elemento zero.

Para ilustrar o uso dos arrays, poderíamos criar um exemplo de aplicação que fica transmitindo continuamente a sequência "SOS" em código Morse, fazendo piscar o LED da placa do Arduino.

Nos séculos XIX e XX, o código Morse costumava ser um método vital de comunicação. Devido ao seu método de codificação das letras como uma série de pontos e traços, o código Morse pode ser enviado através de fios telegráficos, links de rádio ou sinais luminosos. As letras "SOS" (um acrônimo para "save our souls", ou "salve nossas almas") continuam sendo reconhecidas como um sinal internacional de pedido de socorro.

Primeiro, vamos ver como você vai criar um array de **int**s que conterá as durações (durations).

```
int durations[] = {200, 200, 200, 500, 500, 500, 200, 200, 200};
```

Para dizer que uma variável contém um array você deve colocar [ ] após o nome da variável.

Neste caso, você cria o array e, ao mesmo tempo, atribui os valores das durações. A sintaxe para fazer isso consiste em usar chaves que contêm os valores, separados entre si por vírgulas. Não esqueça o ponto e vírgula no final da linha.

Para acessar um dado valor do array, você deve usar a notação de colchetes. Assim, se você quiser o primeiro elemento do array, você poderá escrever o seguinte:

```
durations[0]
```

Como forma de ilustrar isso, vamos criar um array e, em seguida, exibir todos os seus valores no Serial Monitor:

```
// sketch 05-01
int ledPin = 13;

int durations[ ] = {200, 200, 200, 500, 500, 500, 200, 200, 200};

void setup()
{
  Serial.begin(9600);
  for (int i = 0; i < 9; i++)

  {
    Serial.println(durations[i
  }
}

void loop() {}
```

Transfira o sketch para a sua placa e então abra o Serial Monitor. Se tudo estiver correto, você verá algo como a Figura 5-1.

Isso está bem claro. Se você quiser acrescentar mais durações ao array, tudo o que você deverá fazer é acrescentar os valores à lista que está dentro das chaves e mudar o "9" no laço de **for** pelo novo tamanho de array.

Você tem que tomar um pouco de cuidado com os arrays, porque o compilador não impede que você acesse dados que estão além do final do array. Na realidade, isso ocorre porque o array é um apontador de endereços de memória, como está mostrado na Figura 5-2.

Os programas mantêm os seus dados, tanto os comuns como os arrays, na *memória*. A memória de um computador é organizada de forma muito mais rígida do que a memória humana. Ficará mais fácil se pensarmos que a memória de um Arduino é como uma coleção de escaninhos. Por exemplo, quando você define um array de nove elementos, os nove escaninhos disponíveis seguintes são reservadas para o seu uso. Dizemos que a variável aponta para o primeiro escaninho ou *elemento* do array.

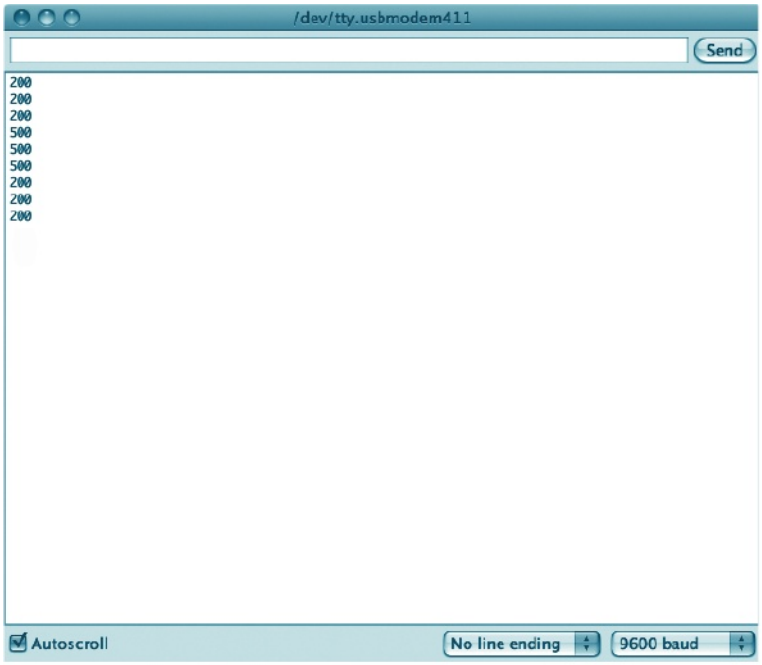

Figura 5-1 O Serial Monitor exibindo a saída do sketch 5-01.

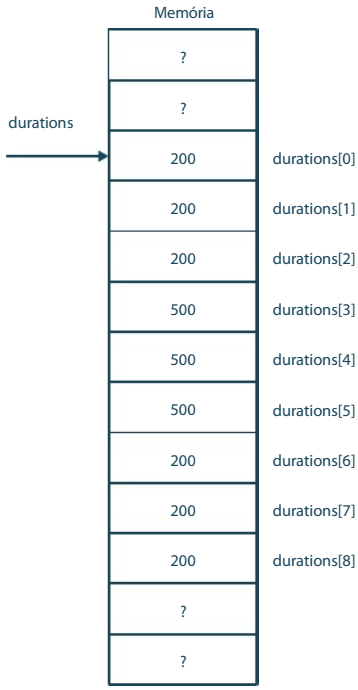

Figura 5-2 Arrays e apontadores.

Vamos voltar à questão de acessar posições além dos limites do seu array. Se você decidir acessar **durations[10]**, você ainda obterá um valor do tipo **int**, mas o valor desse **int** pode ser qualquer coisa. Isso em si não é prejudicial. No entanto, se acidentalmente você acessar um valor que esteja fora dos limites do array, é muito provável que o sketch produzirá resultados confusos.

Entretanto, muito pior ainda é quando você muda um valor que está além dos limites do array. Por exemplo, se você incluir algo como o que está a seguir no programa, então o resultado poderá simplesmente arruinar o seu sketch.

```
durations[10] = 0;
```

O escaninho **durations[10]** pode estar sendo usado por uma variável completamente diferente. Portanto, sempre se assegure de que você não está indo além dos limites do array. Se o seu sketch começar a se comportar de forma estranha, verifique se esse tipo de problema não está ocorrendo.

SOS em código Morse usando Arrays

O sketch 5-02 mostra como você pode usar um array para gerar o seu sinal SOS de emergência:

```
// sketch 05-02
int ledPin = 13;
int durations[] = {200, 200, 200, 500, 500, 500, 200, 200, 200};
void setup()
{
 pinMode(ledPin, OUTPUT);
}
void loop()
{
  for (int i = 0; i < 9; i++)
  {
     flash(durations[i
  }
  delay(1000);
}
void flash(int delayPeriod)
{
 digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
 digitalWrite(ledPin, LOW);
  delay(delayPeriod);
}
```

Uma vantagem óbvia dessa abordagem é a facilidade para alterar a mensagem. Basta modificar simplesmente o array **durations**. No sketch 5-05, você dará um passo adiante no uso de arrays construindo um transmissor luminoso de código Morse para usos mais gerais.

Arrays do tipo String

No mundo da programação, a palavra *string* (barbante) nada tem a ver com um cordão fino e comprido no qual você dá nós. Uma string é uma sequência de caracteres. É o modo que permite usar textos no Arduino. Por exemplo, a cada segundo o sketch 5-03 envia repetidamente o texto "Hello" ("Alô") ao Serial Monitor:

```
// sketch 05-03
void setup()
{
   Serial.begin(9600);
}
void loop()
{
   Serial.println("Hello");
   delay(1000);
}
```

Literais do tipo String

Literais do tipo string, ou simplesmente literais string, são colocadas entre aspas duplas. Elas são literais no sentido de que a string é uma constante, do mesmo modo que **int** 123.

Como seria de esperar, você pode colocar strings em uma variável. Há também uma biblioteca avançada de strings, mas por enquanto você usará as strings padrões da linguagem C, como a do sketch 5-03.

Em C, uma literal string é, na realidade, um array do tipo **char**. O tipo **char** é um pouco como o tipo **int** no sentido de que é um número, mas um número entre 0 e 127 representando um caractere. O caractere pode ser uma letra do alfabeto, um número, um sinal de pontuação, ou um caractere especial como o de tabulação (tab) ou de nova linha. Esses códigos numéricos de letras usam o padrão denominado ASCII. Alguns dos códigos ASCII mais comumente usados estão mostrados na Tabela 5-1.

A literal string "Hello" ("Alô") é, na realidade, um array de caracteres, como está mostrado na Figura 5-3.

Observe que a literal string tem um caractere especial de nulo no final. Esse caractere é usado para indicar o final da string.

Variáveis do tipo String

Como você poderia esperar, uma variável do tipo string, ou simplesmente variável string, é muito similar às variáveis do tipo array, exceto que há um método abreviado que é útil para definir o seu valor inicial.

```
char name[] = "Hello";
```

Esse comando define um array de caracteres e o inicializa com a palavra "Hello". Ele também acrescenta ao array um valor nulo final (ASCII 0) que é usado para indicar o final da string.

O exemplo anterior está bem consistente com o que você sabe sobre o modo de escrever um array. No entanto, mais comumente, o que se escreve mesmo é o seguinte:

Figura 5-3 A literal string "Hello".

Tabela 5-1 Códigos ASCII comuns

Caractere	Código ASCII (em decimal)
a–z	97–122
A–Z	65–90
0–9	48–57
espaço	32

```
// sketch 5-04
char message[] = "Hello";

void setup()
{
   Serial.begin(9600);
}

void loop()
{
   Serial.println(message);
   delay(1000);
}
```

Um tradutor de código Morse

Vamos reunir o que você aprendeu sobre arrays e strings e construir um sketch mais complexo, capaz de aceitar qualquer mensagem vinda do Serial Monitor e transmiti-la usando o LED da placa.

As letras do código Morse estão mostradas na Tabela 5-2.

Algumas das regras do código Morse dizem que a duração de um traço é igual a três vezes a duração de um ponto, a duração de um espaço entre traços e/ou pontos é igual à duração de um ponto, a duração de um espaço entre duas letras tem a mesma duração de um traço e, finalmente, a duração de um espaço entre duas palavras tem a mesma duração de sete pontos.

Neste projeto, não vamos nos preocupar com a pontuação, embora seja um exercício interessante tentar incluí-la no sketch. Para uma lista mais completa dos caracteres do código Morse, acesse en.wikipedia.org/wiki/Morse_code.

Dados

Você construirá este exemplo dando um passo de cada vez e começando com a estrutura de dados que será usada para representar os códigos.

É importante compreender que não há uma única solução para este problema. Diferentes programadores chegariam a diversas maneiras de resolvê-lo. Portanto, seria um erro se você pensasse que "Eu nunca teria chegado a essa solução". Bem, talvez não, mas é muito possível que você chegasse a algo diferente e melhor. Cada pessoa pensa de modo diferente, e a solução apresentada aqui é a que primeiro ocorreu na cabeça do autor.

A representação dos dados consiste em descobrir uma maneira de expressar a Tabela 5-2 em linguagem C. De fato, você dividirá os dados em duas tabelas: uma para as letras (letters) e outra para os números (numbers). A estrutura de dados para as letras é a seguinte:

```
char* letters[] = {
  ".-", "-...", "-.-.", "-..", ".", // A-I
  "..-.", "--.", "....", "..",
  ".---", "-.-", ".-..", "--", "-.", // J-R
  "---", ".--.", "--.-", ".-.",
  "...", "-", "..-", "...-", ".--", // S-Z
  "-..-", "-.--", "--.."
};
```

O que você tem aqui é um array de literais do tipo string. Portanto, como uma literal string é de fato um array do tipo **char**, então o que temos aqui realmente é um array de arrays – algo perfeitamente legal e verdadeiramente muito útil.

Isso significa que, para encontrar o código Morse da letra *A*, você deve acessar **letters[0]**, o que lhe dará .– como string. Essa abordagem não é muito eficiente, porque você está usando um byte inteiro (oito bits) de memória para representar o traço ou o ponto, que poderiam ser representados cada um usando um bit. Entretanto, você pode facilmente justificar esse método dizendo que o número total de bytes está em torno de apenas 90 e nós temos 2K disponíveis para usar. Igualmente importante, essa abordagem facilita a compreensão do código.

Os números podem ser representados do mesmo modo:

```
char* numbers[] = { "..---", "...--", "....-",
".....", "-....", "--...", "---..", "----."};
```

Globais e setup

Você precisa definir duas variáveis globais: uma para a duração de um ponto (dotDelay) e uma para definir o pino de conexão do LED:

Tabela 5-2 Letras em código Morse

A	.-	N	-.	0	-----
B	-…	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	..	Q	--.-	3	…--
E	.	R	.-.	4	….-
F	..-.	S	…	5	…..
G	-.	T	-	6	-….
H	….	U	..-	7	--…
I	..	V	…-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-		
L	.-..	Y	-.--		
M	--	Z	--..		

```
int dotDelay = 200;
int ledPin = 13;
```

A função **setup** é bem simples. Você precisa apenas definir **ledPin** como sendo de saída e inicializar a porta serial:

```
void setup()
{
  pinMode(ledPin, OUTPUT);

}  Serial.begin(9600);
```

A função *loop*

Agora você dará início ao trabalho real de processamento que é executado na função **loop**. O algoritmo dessa função é o seguinte:

- Se houver um caractere para ser lido na entrada USB:
  - Se for uma letra, transmita-a pelo LED usando o array de letras
  - Se for um número, transmita-o pelo LED usando o array de números
  - Se for um espaço, espere quatro vezes a duração de um ponto

Isso é tudo. Você não deve pensar muito mais além disso. Esse algoritmo representa o que você quer fazer, ou qual é a sua *intenção*. Esse estilo de programação é chamado *programação por intenção*.
Se você escrever esse algoritmo em C, ele será como segue:

```
void loop()
{
  char ch;
  if (Serial.available() > 0)
   {
    ch = Serial.read();
     if (ch >= 'a' && ch <= 'z')
      {
      flashSequence(letters[ch - 'a'
      }
     else if (ch >= 'A' && ch <= 'Z')
      {
      flashSequence(letters[ch - 'A'
      }
     else if (ch >= '0' && ch <= '9')
      {
      flashSequence(numbers[ch - '0'
      }
     else if (ch == ' ')
      {
      delay(dotDelay * 4); // intervalo de tempo entre palavras
      }
   }
}
```

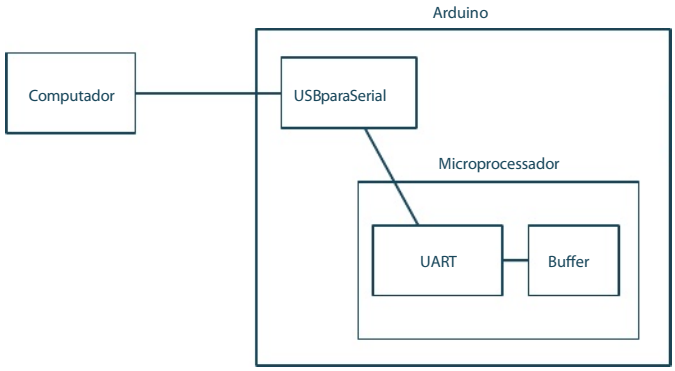

Figura 5-4 Comunicação serial com o arduino.

Há umas poucas coisas que necessitam de explicação. Primeiro, há **Serial.available()** (Serial.disponível). Para compreendê-la você inicialmente precisa conhecer um pouco sobre a maneira do Arduino comunicar-se com o seu computador através da conexão USB. A Figura 5-4 resume esse processo.

Quando o computador está enviando dados do Serial Monitor para a placa do Arduino, os dados que chegam a USB são convertidos do protocolo e dos níveis de sinal USB em algo que pode ser usado pelo microcontrolador da placa de Arduino. Essa conversão acontec e em um chip de uso especial que está nessa placa. Em seguida, os dados são recebidos por uma parte do microcontrolador denominada Universal Asynchronous Receiver/Transmitter (UART)*. A UART coloca os dados recebidos em um buffer. O buffer é uma área especial da memória (128 bytes) que pode armazenar dados. Esses dados são removidos logo após serem lidos.

Essa comunicação acontece independentemente do que o seu sketch esteja fazendo. Assim, mesmo que você esteja feliz fazendo os LEDs piscar, os dados continuarão chegando e sendo armazenados no buffer, e ali ficarão até que você esteja pronto para lê-los. Você pode imaginar o buffer como algo semelhante a uma caixa de entrada de e-mail.

A maneira de você verificar se "tem e-mail" é usando a função **Serial.available()** . Essa função retorna o número de bytes de dados que estão disponíveis (available) no buffer esperando que você os leia. Se não há mensagens esperando para serem lidas, então a função retorna o valor 0. Essa é a razão do comando **if** verificar se há mais do que zero bytes disponíveis esperando para serem lidos. Se houver bytes esperando, então a primeira coisa que o comando faz é ler o próximo **char** disponível, usando a função de leitura serial **Serial.read()**. O valor retornado por essa função é atribuído à variável local **ch** (character, ou caractere)

Em seguida, há outro **if** para decidir o que você deseja fazer:

* N. deT.: Receptor/Transmissor Assíncrono Universal.

```
if (ch >= 'a' && ch <= 'z')
{
  flashSequence(letters[ch - 'a'
}
```

No início, isso pode parecer um tanto estranho. Você está usando <= e >= para comparar caracteres. Você pode fazer isso porque, na realidade, cada caractere é representado por um número (o seu código ASCII). Desse modo, se o código do caractere está em algum lugar entre *a* e *z* (97 e 122), então você sabe que o caractere que veio do computador é uma letra minúscula. A seguir, você chama uma função, que ainda não foi escrita, denominada **flashSequence** (sequência de flashes), para a qual você passará uma string de pontos e traços. Por exemplo, para transmitir *a*, você deve passar a sequência .– como seu argumento.

Você está dando a essa função a responsabilidade de acender e apagar (flash) o LED. Como você não está fazendo isso dentro do **loop**, a leitura do código se torna fácil.

Aqui está o comando em C que determina qual é a string com a sequência de pontos e traços que você precisa enviar para a função **flashSequence**:

```
letters[ch - 'a']
```

Novamente, isso parece um pouco estranho. Aparentemente, a função está subtraindo um caractere de outro. Na verdade, isso é algo perfeitamente razoável de ser feito, porque a função está, na realidade, subtraindo os valores ASCII.

Lembre-se de que você está armazenando os códigos das letras em um array. Assim, o primeiro elemento do array contém uma string com a sequência de pontos e traços para a letra *A*, o segundo elemento contém os pontos e traços da letra *B*, e assim por diante. Desse modo, você precisa encontrar no array a posição correta da letra que você acabou de buscar no buffer. A posição de qualquer letra minúscula será o código de caractere da letra menos o código de caractere da letra minúscula *a*. Assim, por exemplo, *a* – *a* é, na realidade, 97 – 97 = 0. Do mesmo modo, *c* – *a* é, na realidade, 99 – 97 = 2. Portanto, se **ch** for a letra *c*, então a expressão dentro dos colchetes dará como resultado o valor 2, com o qual você obtém o elemento 2 do array, que tem – . – . como string.

O que esta seção acabou de descrever está relacionado com as letras minúsculas. Você também deve levar em consideração as letras maiúsculas e os números. Ambos são tratados de forma similar.

» A função *flashSequence*

Nós assumimos que uma função de nome **flashSequence** estava disponível e a usamos, mas agora precisamos escrevê-la. Para isso, nós imaginamos que ela recebe uma string contendo uma sequência de pontos e traços e que corretamente acende e apaga (flash) o LED com os intervalos de tempo necessários.

Pensando em um algoritmo capaz de fazer isso, você pode dividi-lo nos seguintes passos:

- Para cada elemento da string de pontos e traços (tal como . – . –)
  - Faça o LED piscar conforme seja um ponto ou um traço

Usando o conceito de programação por intenção, vamos manter a função limitada a isso.

Os códigos Morse das letras não têm o mesmo comprimento. Desse modo, você deve ficar repetindo o procedimento com a string até encontrar o marcador final, \0. Você também precisa de uma variável de contagem denominada i, que inicia em 0 e é incrementada sempre que cada ponto ou traço for processado:

```
void flashSequence(char* sequence)
{
  int i = 0;
  while (sequence[i] != '\0')
   {
      flashDotOrDash(sequence[i
      i++;
   }
  delay(dotDelay * 3); // intervalo de tempo entre letras
}
```

Tanto para produzir o pulso luminoso correspondente a um ponto (dot) ou um traço (dash), você novamente delegará o trabalho real de fazer o LED piscar (flash) a uma nova função denominada **flashDotOrDash** (piscarPontoOuTraço). É essa função que realmente acende ou apaga o LED. Finalmente, quando o programa executou toda a sequência de pontos e traços, ele precisa fazer uma pausa com uma duração equivalente a três pontos. Observe como é útil incluir um comentário.

A função *flashDotOrDash*

A última função desta série de funções é a que realmente executa o trabalho de ligar e desligar o LED. Como argumento, a função tem um único caractere que é ou um ponto (.) ou um traço (–).

Tudo que a função precisa fazer é acender o LED e, se o argumento for um ponto, deverá esperar o intervalo de tempo de um ponto e, se o argumento for um traço, deverá esperar três vezes o intervalo de tempo de um ponto. Em seguida, a função deve apagar o LED. Finalmente, ela precisa aguardar o tempo de um ponto para fazer a separação entre os pontos e traços.

```
void flashDotOrDash(char dotOrDash)
{
  digitalWrite(ledPin, HIGH);
   if (dotOrDash == '.') // pontoOuTraço
   {
   delay(dotDelay); // espera o tempo de um ponto (dot)
   }
   else // deve ser um traço - (dash)
   {
     delay(dotDelay * 3); // espera o tempo de um traço
   }
  digitalWrite(ledPin, LOW);
   delay(dotDelay); // intervalo de tempo entre os flashes
}
```

Juntando tudo

Reunindo tudo isso, a listagem final está mostrada no sketch 5-05. Transfira-o para a sua placa do Arduino e experimente. Lembre-se de que para usá-lo você deve abrir o Serial Monitor e escrever algum texto na área que está na parte superior e clicar Send (Enviar). Então, você deverá ver o texto sendo transmitido em código Morse pelo LED.

```
// sketch 5-05
int dotDelay = 200;
int ledPin = 13;
char* letters[] = {
   ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....", "..", // A-I
   ".---", "-.-", ".-..", "--", "-.", "---", ".--.", "--.-", ".-.", // J-R
   "...", "-", "..-", "...-", ".--", "-..-", "-.--", "--.." // S-Z
};
char* numbers[] = {"-----", ".----", "..---", "...--", "....-",
   ".....", "-....", "--...", "---..", "----."};
void setup()
{
  pinMode(ledPin, OUTPUT);
   Serial.begin(9600);
}
void loop()
{
  char ch;
   if (Serial.available() > 0)
   {
    ch = Serial.read();
     if (ch >= 'a' && ch <= 'z')
      {
       flashSequence(letters[ch - 'a'
      }
     else if (ch >= 'A' && ch <= 'Z')
      {
       flashSequence(letters[ch - 'A'
      }
     else if (ch >= '0' && ch <= '9')
      {
       flashSequence(numbers[ch - '0'
```

```
      }
      else if (ch == ' ')
      {
       delay(dotDelay * 4); // intervalo de tempo entre palavras
      }
   }
}
void flashSequence(char* sequence)
{
   int i = 0;
   while (sequence[i] != '\0')
   {
      flashDotOrDash(sequence[i
      i++;
   }
   delay(dotDelay * 3); // intervalo de tempo entre letras
}
void flashDotOrDash(char dotOrDash)

{
  digitalWrite(ledPin, HIGH);
   if (dotOrDash == '.')
   {
     delay(dotDelay); // espera o tempo de um ponto (dot)
   }
   else // deve ser um traço - (dash)
   {
     delay(dotDelay * 3); // espera o tempo de um traço
   }
  digitalWrite(ledPin, LOW);
   delay(dotDelay); // intervalo de tempo entre os flashes
}
```

Esse sketch contém uma função **loop** que é chamada automaticamente e que repetidamente chama a função **flashSequence** que você escreveu. Por sua vez, a **flashSequence** chama repetidas vezes a função **flashDotOrDash**, que você também escreveu, a qual chama as funções **digitalWrite** e **delay**, que são fornecidas pelo próprio Arduino!

É assim que seus sketches devem ser construídos. Quando você decompõe as coisas que devem ser feitas em funções, fica muito mais fácil colocar o seu código em funcionamento, além de facilitar o seu uso quando você volta a executá-lo após um tempo.

Conclusão

Além de ter examinado as strings e os arrays neste capítulo, você também construiu um tradutor de código Morse mais complexo. Espero que isso tenha reforçado a importância de escrever o seu código usando funções.

No próximo capítulo, você aprenderá a usar as entradas e saídas de sinais analógicos e digitais do Arduino.

» capítulo 6

Entrada e saída

O Arduino trata da chamada computação física. Isso significa ligar circuitos eletrônicos à placa do Arduino. Portanto, é necessário que você compreenda como usar as várias opções de conexão dos seus pinos.

As saídas podem ser digitais, o que significa assumir valores de 0V ou 5V, ou analógicas, o que significa que você pode fazer a tensão assumir qualquer valor entre 0V e 5V – embora não seja tão simples assim, como veremos.

Objetivos deste capítulo

- Entender como funcionam as saídas e as entradas digitais
- Aprender a estabelecer a comunicação entre o Arduino e um computador PC
- Aprender a informar ao Arduino qual pino será utilizado
- Aprender os tipos de entradas e saídas
- Aprender a usar resistores de pull-up e a realizar debouncing
- Aprender a usar a entrada analógica

De modo semelhante, as entradas podem ser digitais (por exemplo, determinar se um botão está pressionado ou não) ou analógicas (como no caso de um sensor luminoso).

Em um livro que trata basicamente de software e não de hardware, procuraremos não nos envolver em discussões aprofundadas de eletrônica. Neste capítulo, entretanto, seria útil se você pudesse dispor de um multímetro e de um pedaço de fio rígido para que você compreenda o que está acontecendo.

Saídas digitais

Nos capítulos anteriores, você usou o LED conectado ao pino 13 da placa do Arduino. Por exemplo, no Capítulo 5, você o usou no sinalizador de código Morse. A placa do seu Arduino tem um conjunto completo de pinos digitais à sua disposição.

Vamos experimentar um outro pino do Arduino. Usaremos o pino digital 4. Para ver o que está acontecendo, ligue alguns fios ao multímetro e ao Arduino. A Figura 6-1 mostra as ligações. Se o seu multímetro tem ponteiras do tipo jacaré, primeiro descasque as extremidades de alguns pedaços curtos de fio rígido. A seguir, conecte um jacaré a uma das extremidades e insira a outra extremidade do fio rígido no terminal de conexão que está na placa do Arduino. Se o seu multímetro não tiver ponteiras com jacarés, então enrole as extremidades descascadas dos fios diretamente nas ponteiras.

O multímetro precisa estar na faixa de corrente contínua (DC) de 0–20V. A ponteira negativa (preta) deve ser conectada ao pino de massa ou terra (GND) e a ponteira positiva (vermelha), ao pino D4. Para isso, as extremidades descascadas de fio rígido são simplesmente conectadas às ponteiras do multímetro e inseridas nos terminais de conexão da placa do Arduino.

Carregue o sketch 6-01:

```
//sketch 6-01
int outPin = 4; // Pino de saída = 4
void setup()
{
  pinMode(outPin, OUTPUT);
   Serial.begin(9600);
  Serial.println("Enter 1 or 0");
   // "Entre com 1 ou 0"
}

void loop()
{
  if (Serial.available() > 0)
   {
     char ch = Serial.read();
     if (ch == '1')
      {
     digitalWrite(outPin, HIGH);
      }
     else if (ch == '0')
      {
```

Figura 6-1 Uso de um multímetro para medir as saídas.

```
      digitalWrite(outPin, LOW);
       }
    }
}
```

No início do sketch, você pode ver o comando **pinMode** (modo do pino). Para cada pino que você estiver usando em um projeto, você deverá usar esse comando. Desse modo, o Arduino pode configurar o circuito eletrônico conectado a esse pino para que atue como entrada ou saída. Veja o exemplo seguinte:

```
pinMode(outPin, OUTPUT);
```

Como você já deve ter notado, o comando **pinMode** é uma função predefinida do Arduino. O seu primeiro argumento é o número do pino em questão (um **int**) e o segundo argumento é o modo, que pode ser ou **INPUT** (entrada) ou **OUTPUT** (saída). Observe que o nome do modo deve ser escrito todo em letras maiúsculas.

O **loop** fica esperando que um comando de **1** ou **0** venha do Serial Monitor do seu computador. Se for **1**, então o pino 4 é ligado (5V), senão ele é desligado (0V).

Transfira o sketch para o seu Arduino e então abra o Serial Monitor (mostrado na Figura 6-2).

Desse modo, com o multímetro funcionando e ligado ao Arduino, você deverá ver as leituras mostrando valores que mudam entre 0V (nível baixo–LOW) e cerca de 5V (nível alto–HIGH). Isso acontece quando você usa o Serial Monitor e envia comandos para a placa, teclando **1** e em seguida **Return** (ou **Send**), ou teclando **0** e em seguida **Return** (ou **Send**).

Se não houver pinos suficientes com a indicação "D" (digital) para o seu projeto, então você também poderá usar os pinos com a indicação "A" (analógicos) como saídas digitais. Para isso, você precisa simplesmente acrescentar *14* ao número do pino analógico. Tente fazer isso. Para tanto, modifique a primeira linha do sketch 6-01 para que o pino 14 seja usado e ligue a ponteira positiva do seu multímetro ao pino A0 do Arduino.

Isso é tudo que há a respeito de saídas digitais. Agora, vamos passar rapidamente para as entradas digitais.

Entradas digitais

O uso mais comum das entradas digitais é detectar se uma chave está fechada. Uma entrada digital pode estar ligada ou desligada. Se a tensão da entrada for menor que 2,5V (metade de 5V), então ela será 0 (desligada) e, se for maior que 2,5 V, ela será 1 (ligada).

Desconecte o seu multímetro e transfira o sketch 6-02 para a placa do seu Arduino:

```
//sketch 06-02
int inputPin = 5; // Pino de entrada = 5
void setup()
{
  pinMode(inputPin, INPUT);
   Serial.begin(9600);
}
void loop()
{
  int reading = digitalRead(inputPin);
   // leitura (reading)
   Serial.println(reading);

}  delay(1000);
```

Como fizemos antes quando usamos uma saída, agora você precisa dizer ao Arduino na função **setup** que você vai usar um determinado pino como entrada. O valor presente em uma entrada digital pode ser lido (read) usando a função **digitalRead** de leitura digital. Ela retorna 0 ou 1.

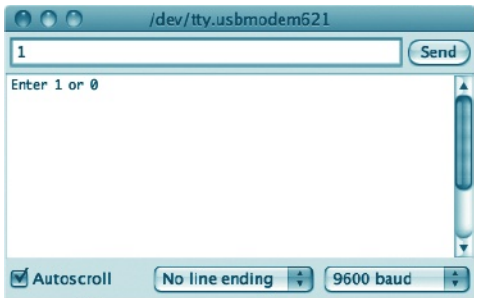

Figura 6-2 O Serial Monitor.

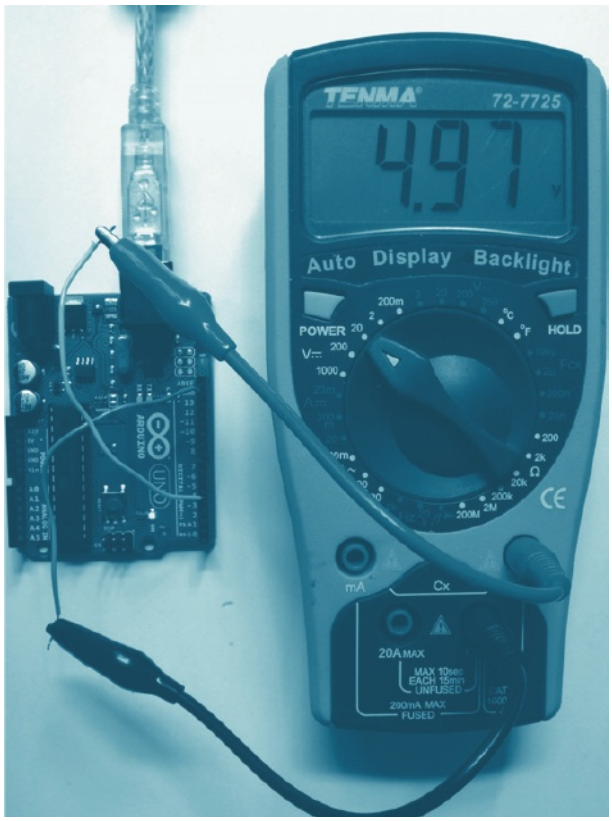

Figura 6-3 Colocando a saída em nível alto (HIGH).

Resistores de Pull-Up

A cada segundo, o sketch lê o pino de entrada e envia o seu valor para o Serial Monitor. Desse modo, faça o upload do sketch e abra o Serial Monitor. Você deve ver um novo valor aparecendo a cada segundo. Insira uma extremidade do seu pedaço de fio no terminal de conexão D5 e aperte com os dedos a outra extremidade do fio, como está mostrado na Figura 6-4.

Continue apertando e soltando a extremidade do fio por alguns segundos e observe o texto que aparece no Serial Monitor. Você deverá ver uma mistura de zeros (0) e uns (1) aparecer no Serial Monitor. A razão é que as entradas da placa do Arduino são muito sensíveis. Você está atuando como uma antena, captando interferência elétrica.

Pegue a extremidade do fio que você está apertando e a insira no terminal de conexão de +5V, como está mostrado na Figura 6-5. Agora o texto mostrado pelo Serial Monitor deverá mudar para uma sequência de uns (1).

Em seguida, pegue a extremidade que tinha sido inserida em +5V e leve-a para uma das conexões de massa (GND) do Arduino. Agora, como seria de esperar, o Serial Monitor passou a exibir zeros.

Um uso típico de pino de entrada é a sua conexão a uma chave. A Figura 6-6 mostra como você pode fazer essa ligação.

O problema é que, quando a chave não está fechada, o pino de entrada não está ligado a coisa alguma. Diz-se que ele está flutuando, podendo facilmente produzir uma leitura falsa. É neces-

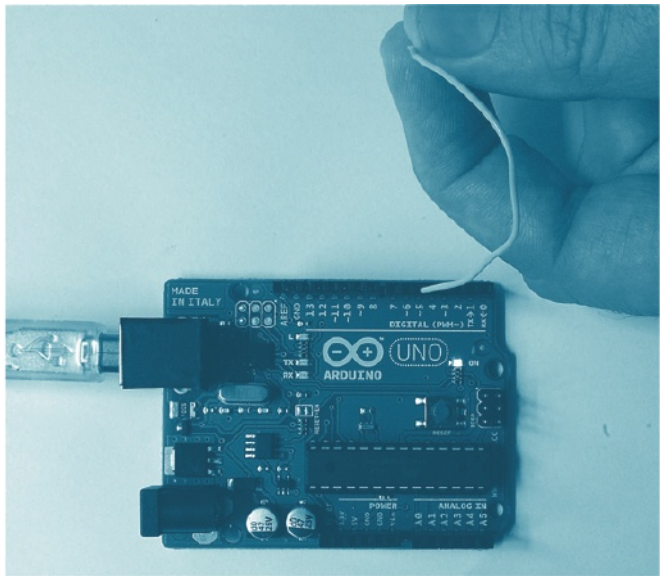

Figura 6-4 Uma entrada digital com uma antena humana.

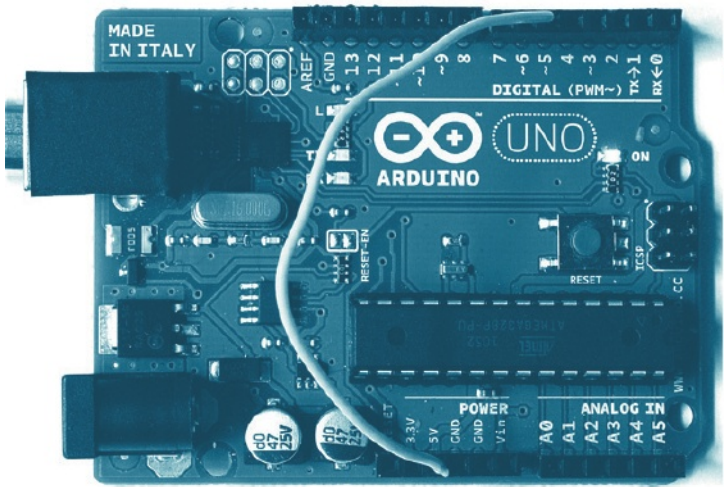

Figura 6-5 O pino 5 conectado a +5V.

Figura 6-6 Conexão de uma chave à placa do Arduino.

sário que a sua entrada seja mais previsível, e a maneira de conseguir isso é usando o chamado resistor de pull-up. A Figura 6-7 mostra como um resistor de pull-up normalmente é usado. O efeito obtido com ele é que, se a chave estiver aberta, então o resistor puxará a entrada flutuante para cima (pull-up) até 5V. Quando você aciona a chave fechando o contato, ela se sobrepõe ao resistor e forçará a entrada para 0V. Um efeito paralelo é que, enquanto a chave permanecer

Figura 6-7 Chave com um resistor de pull-up.

fechada, uma tensão de 5V estará sendo aplicada ao resistor, o que causará a circulação de uma corrente. Desse modo, o valor do resistor deve ser escolhido de tal forma que seja suficientemente baixo para se tornar imune a qualquer interferência elétrica e, ao mesmo tempo, suficientemente alto para que uma corrente excessiva não circule quando a chave está fechada.

Resistores internos de Pull-Up

Felizmente, a placa do Arduino tem resistores de pull-up que são configuráveis por software e que estão associados aos pinos digitais. Normalmente, eles estão desligados. Assim, tudo o que você precisa fazer para habilitar o resistor de pull-up do pino 5 do sketch 6-02 é acrescentar a seguinte linha:

```
digitalwrite (inputPin, HIGH);
```

Essa linha é inserida na função **setup** logo após definir o pino como sendo de entrada. Pode parecer um tanto estranho usar um **digitalWrite** em uma entrada, mas é assim que funciona.

O sketch 6-03 é a versão modificada. Transfira-o para a placa do seu Arduino e teste-o atuando como antena novamente. Desta vez você verá que a entrada permanece em 1 no Serial Monitor.

```
//sketch 06-03
int inputPin = 5;
void setup()
{
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin, HIGH);
   Serial.begin(9600);
}
void loop()
{
   int reading = digitalRead(inputPin);
    Serial.println(reading);
    delay(1000);
}
```

Debouncing

Ao apertar uma chave de botão, você espera que ocorra uma única mudança de valor de 1 (com resistor de pull-up) para 0 quando o botão é pressionado. A Figura 6-8 mostra o que acontece realmente quando você aperta o botão. Os contatos de metal no interior do botão entram em vibração (bouncing, em inglês) até chegarem ao repouso. Consequentemente, um único aperto no botão converte-se em uma série de abre e fecha contato, que no final acaba se estabilizando.*

Tudo isso ocorre muito rapidamente. Quando se aperta o botão, o intervalo total de tempo no traçado de osciloscópio é de apenas 200 milissegundos. Essa é uma chave velha, em más condições. Um botão novo, do tipo declicar, pode inclusive nemvibrar.

Em alguns casos, esse efeito de bouncing não tem importância. Por exemplo, o sketch 6-04 ligará o LED enquanto o botão estiver pressionado. Na realidade, você não usaria um Arduino para fazer esse tipo de coisa. Aqui, estamos lidando com questões mais teóricas do que práticas.

```
//sketch 06-04

int inputPin = 5;
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(inputPin, INPUT);
```

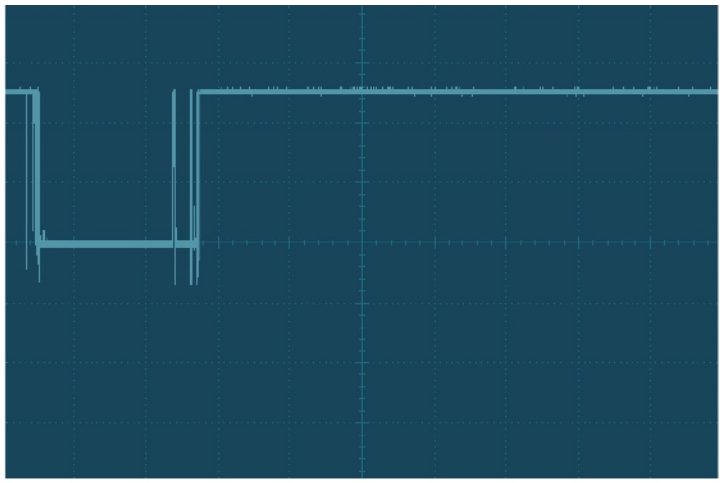

Figura 6-8 Traçado no osciloscópio de um pressionamento de botão.

* N. de T.: Bouncing é o efeito semelhante ao de uma bola que é solta da mão e põe-se a quicar contra o solo cada vez com menos altura até finalmente parar.

```
  digitalWrite(inputPin, HIGH);
}
void loop()
{
   int switchOpen = digitalRead(inputPin);
   digitalWrite(ledPin, ! switchOpen);
}
```

Olhando a função **loop** do sketch 6-04, a função lê a entrada digital e atribui esse valor a uma variável de nome **switchOpen** (chave aberta). Será 0 se o botão estiver acionado (chave fechada) e 1 em caso contrário (lembre-se de que a tensão no pino está sendo puxada para cima – para 1 – quando o botão não está sendo pressionado).

Quando você programa o comando **digitalWrite** para ligar ou desligar o LED, você precisa inverter esse valor. Isso é feito usando o operador **!** ou **not**.

Se você transferir esse sketch e conectar um fio entre D5 e GND (veja a Figura 6-9), você deverá ver o LED acender. Nesse caso, um efeito de bouncing pode estar acontecendo aqui, mas provavelmente é tão rápido que você não consegue ver, não sendo importante na realidade.

Uma situação em que o efeito de bouncing torna-se importante é quando você está fazendo o LED ser ligado ou desligado sempre que você pressiona o botão. Isto é, quando você aperta o botão, o LED é aceso, permanecendo assim. Quando você pressiona o botão novamente, o

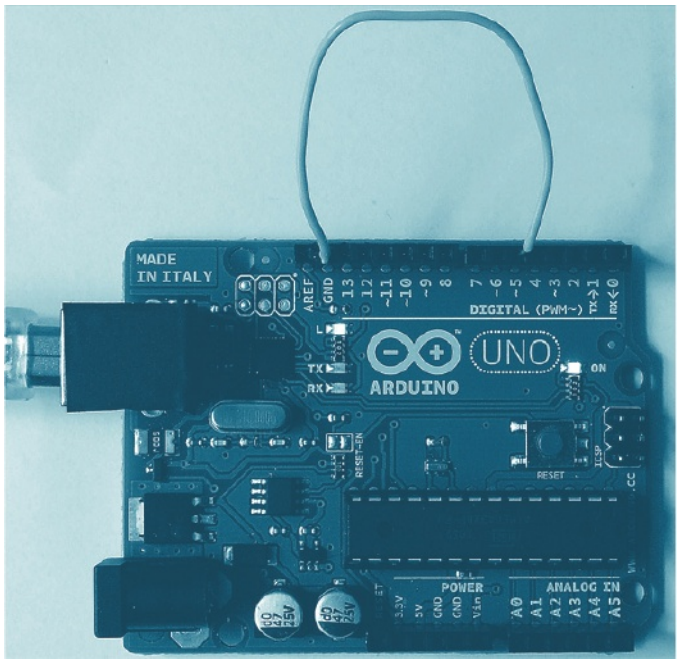

Figura 6-9 Usando um pedaço de fio como chave.

LED é desligado, permanecendo assim. Se ocorresse efeito de bouncing no contato interno do seu botão, então o LED seria ligado ou desligado dependendo de o número de abre e fecha contato no botão ter sido ímpar ou par.

O sketch 6-05 simplesmente liga e desliga o LED sem qualquer tentativa de levar em consideração o efeito de "bouncing". Experimente usar o seu fio como se fosse uma chave entre o pino D5 e o GND.

```
//sketch 06-05

int inputPin = 5;
int ledPin = 13;
int ledValue = LOW; // Valor do led = BAIXO

void setup()
{
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin, HIGH);
  pinMode(ledPin, OUTPUT);
}

void loop()
{
   if (digitalRead(inputPin) == LOW)
    {
      ledValue = ! ledValue;
    digitalWrite(ledPin, ledValue);
    }
}
```

Provavelmente você verá que algumas vezes o LED está funcionando corretamente, invertendo o seu estado, mas outras vezes parece não estar funcionando. Isso é o bouncing em ação!

Uma maneira simples de lidar com esse problema é adicionar um retardo logo que você detectar que o botão foi pressionado no início do bouncing, como está mostrado no sketch 6-06:

```
//sketch 06-06
int inputPin = 5;
int ledPin = 13;
int ledValue = LOW;

void setup()
{
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin, HIGH);
  pinMode(ledPin, OUTPUT);
}

void loop()
{
   if (digitalRead(inputPin) == LOW)
    {
      ledValue = ! ledValue;
    digitalWrite(ledPin, ledValue);
       delay(500);
    }
}
```

Colocando um retardo (delay) aqui, nada poderá acontecer durante os próximos 500 milissegundos, quando ao final qualquer efeito de bouncing terá cessado. V ocê descobrirá que isso tornará o efeito de ligar e desligar o LED muito mais confiável. Um resultado interessante é que, se você mantiver o botão pressionado (ou seja, o fio conectado), o LED ficará piscando.

Se isso é tudo que o sketch precisa fazer, então esse retardo não será um problema. Entretanto, se você fizer mais coisas dentro do **loop**, o uso do delay pode ser um problema. Por exemplo, o programa não consegue detectar se algum outro botão foi pressionado durante esses 500 milissegundos.

Desse modo, algumas vezes essa abordagem não é boa o suficiente. Você precisará ser bem mais engenhoso, escrevendo você mesmo um código mais complexo (denominado debouncer*) para lidar com esse problema. Isso pode ser bem complicado, mas felizmente todo o trabalho já foi feito para você por alguns sujeitos admiráveis.

Para fazer uso do trabalho dessas pessoas, você deve acrescentar uma biblioteca (library) ao seu ambiente Arduino. Para tanto, você deve baixar a própria biblioteca (denominada Bounce) que está disponível na página http://www.arduino.cc/playground/Code/Bounce, na forma de um arquivo zip.

Após baixar o arquivo, faça unzip e coloque o conteúdo, denominado Bounce, na subpasta libraries (bibliotecas) da pasta em que todos os seus sketches estão salvos. No Windows, essa pasta é Meus Documentos\Arduino, e no Mac e no Linux é Documents/Arduino. Se a subpasta libraries (bibliotecas) não existir, então será necessário criá-la**. A Figura 6-10 mostra a estrutura de pastas e subpastas no Windows depois do acréscimo da biblioteca Bounce.

Após o acréscimo da biblioteca, você precisa reiniciar o aplicativo Arduino para que as alterações tornem-se efetivas. Depois disso, você poderá usar a biblioteca Bounce em qualquer sketch que você vier a escrever.

O sketch 6-07 mostra como você pode usar a biblioteca Bounce. Faça o upload para a sua placa e veja quão confiável ficou o funcionamento do liga e desliga do LED.

```
//sketch 06-07
#include <Bounce.h>
int inputPin = 5;
int ledPin = 13;

int ledValue = LOW;
Bounce bouncer = Bounce(inputPin, 5);

void setup()

{
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin, HIGH);
  pinMode(ledPin, OUTPUT);
}
```

* N. de T.: Debouncer é o nome dado a um recurso, programado ou eletrônico, que é usado para eliminar os efeitos não desejados do bouncing.

** N. de T.: O nome desta pasta deve ser em inglês (libraries).

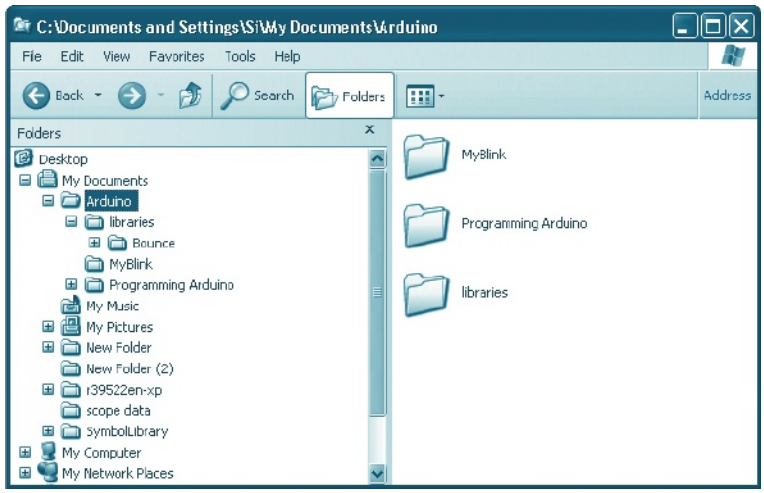

Figura 6-10 Acrescentando a biblioteca Bounce no Windows.

```
void loop()
{

  if (bouncer.update() && bouncer.read() == LOW)
  {
    ledValue = ! ledValue;
   digitalWrite(ledPin, ledValue);
  }
}
```

O uso dessa biblioteca é bem simples. A primeira coisa que você vai observar é a seguinte linha:

```
#include <Bounce.h>
```

Isso é necessário para informar o compilador que deve usar a biblioteca Bounce. Você tem então a seguinte linha:

```
Bounce bouncer = Bounce(inputPin, 5);
```

Por enquanto não se preocupe com a sintaxe dessa linha. Trata-se de sintaxe C++ em vez de C, e C++ será visto no Capítulo 11. Por enquanto, você vai ter que se contentar em saber apenas que isso inicializa um objeto de nome **bouncer** associado ao pino especificado, com um período de espera (debounce) de 5 milissegundos para que o contato da chave estabilize.

A partir de agora, você usará esse objeto **bouncer** para descobrir o que a chave está fazendo em vez de ler diretamente a entrada digital. Ele coloca uma espécie de papel de embrulho em torno do seu pino de entrada. Assim, para decidir se um botão foi apertado usaremos a seguinte linha:

```
if (bouncer.update() && bouncer.read() == LOW)
```

A função **update** (atualizar) retorna um valor booleano em nível lógico alto (verdadeiro) quando alguma coisa mudou no objeto **bouncer** e a segunda parte da condição verifica se o botão passou para nível **LOW** (baixo ou 0).

Saídas analógicas

Alguns dos pinos digitais – a saber, os pinos digitais 3, 5, 6, 9, 10 e 11 – podem fornecer uma saída variável, em vez de apenas 5V ou nada. Esses são os pinos da placa com um ~ ou "PWM" próximo deles. PWM significa Modulação por Largura de Pulso (Pulse Width Modulation) e refere-se ao modo de controlar a intensidade da potência que está sendo fornecida na saída. Isso é feito ligando e desligando rapidamente a saída.

Os pulsos são sempre entregues com a mesma taxa (aproximadamente 500 por segundo), mas a duração dos pulsos pode ser variada. Se você usar PWM para controlar o brilho de um

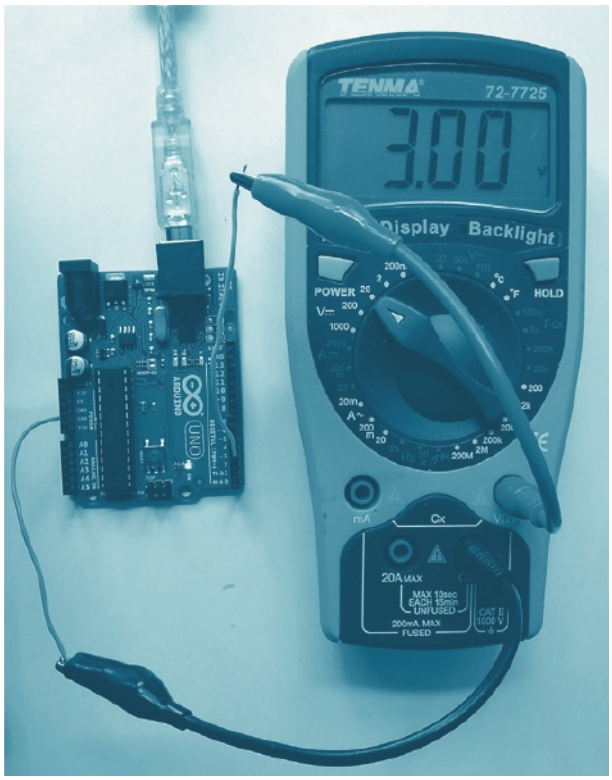

Figura 6-11 Medição da saída analógica.

LED, então, quando o pulso for longo, o seu LED estará aceso todo o tempo. Entretanto, se os pulsos forem curtos, então o LED estará aceso na realidade apenas durante um tempo curto. Isso acontece com rapidez demasiada para que o observador possa mesmo ver que o LED está piscando. Parece simplesmente que o LED está com mais ou menos brilho.

Antes de tentar usar o LED, você pode testá-lo com seu multímetro. Ajuste o multímetro para medir a tensão entre GND e pino D3 (veja a Figura 6-11).

Agora, transfira o sketch 6-08 para a sua placa e abra o Serial Monitor (veja a Figura 6-12). Entre com um único dígito 3 e ap erte Return (ou Se nd). Você deverá ver o voltímetro mostrar um valor em torno de 3V. A seguir, você pode experimentar com outros núme ros entre 0 e 5.

```
//sketch 06-08

int outputPin = 3;

void setup()
{
  pinMode(outputPin, OUTPUT);
   Serial.begin(9600);
  Serial.println("Enter Volts 0 to 5");
  //Entre com os Volts 0 a 5
}

void loop()
{
  if (Serial.available() > 0)
  {
    char ch = Serial.read();
     int volts = (ch - '0') * 51;
   analogWrite(outputPin, volts);
  }
}
```

O programa determina o valor da saída PWM entre 0 e 255 multiplicando a tensão desejada (0 a 5) por 51. (Os leitores poderão consultar a Wikipedia para uma descrição mais detalhada de PWM.)

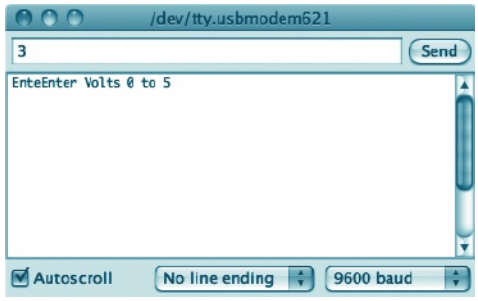

Figura 6-12 Ajustando a tensão de uma saída analógica.

Você pode ajustar o valor de saída usando a função **analogWrite** (escrever analógico), que requer um valor de saída entre 0 e 255, em que 0 é desligado e 255 é potência máxima. Na realidade, esse é um ótimo meio de controlar o brilho de um LED. Se você estivesse tentando controlar o brilho pela variação da tensão do LED, você veria que nada aconteceria até que você tivesse cerca de 2V. Então muito rapidamente o LED ficaria com bastante brilho. Se você controlar o brilho usando PWM e variar o tempo médio em que o LED permanece aceso, você conseguirá um controle muito mais linear do brilho.

Entrada analógica

Uma entrada digital fornece apenas um valor de ligado ou desligado em relação ao que está acontecendo na entrada de um dado pino da placa do Arduino. Por outro lado, uma entrada analógica fornece um valor entre 0 e 1023 dependendo da tensão presente no pino de entrada analógica.

Um programa pode ler (read) a entrada analógica (analog) usando a função **analogRead**. A cada meio segundo, o sketch 6-09 exibe, no Serial Monitor, a leitura e a tensão real que está presente no pino analógico A0. Portanto, abra o Serial Monitor e observe as leituras aparecerem.

```
//sketch 06-09

int analogPin = 0;
void setup()
{
   Serial.begin(9600);
}

void loop()
{
  int reading = analogRead(analogPin);
  float voltage = reading / 204.6;
 Serial.print("Reading="); //"Leitura="
  Serial.print(reading);
  Serial.print("\t\tVolts=");
  Serial.println(voltage);
  delay(500);
}
```

Quando você executar esse sketch, você verá que as leituras variam muito. Isso ocorre porque a entrada está flutuando, como nas entradas digitais.

Pegue uma extremidade de um fio e coloque-a em um terminal de conexão de GND, de modo que A0 esteja conectado a GND. Agora, as suas leituras devem permanecer em 0. Mova a extremidade do fio que estava em GND e coloque-a em 5V. Você deverá ter uma leitura em torno de 1023, que é a leitura máxima. Em seguida, se você conectar A0 ao terminal de conexão de 3.3V da placa do Arduino, o sketch deverá lhe dizer que você tem cerca de 3.3V.

Conclusão

Isso conclui o nosso capítulo a respeito dos fundamentos de entrada e saída de sinais no Arduino. No próximo capítulo, examinaremos alguns dos recursos disponibilizados pela biblioteca padrão do Arduino.

» capítulo 7

A biblioteca padrão do Arduino

É na biblioteca padrão do Arduino que vivem as guloseimas. Até agora você utilizou apenas o núcleo da linguagem C. O que você realmente precisa é uma grande coleção de funções para usar em seus sketches.

Você já encontrou algumas dessas funções, como **pinMode**, **digitalWrite** e **analogWrite**. Na verdade, há muito mais delas. São funções que você pode usar para realizar operações matemáticas, gerar números aleatórios, manipular bits, detectar pulsos em um pino de entrada e lidar com algo denominado interrupção.

Objetivos deste capítulo

- Ajudar a descobrir outras funções diponíveis na biblioteca padrão
- Entender a função Random
- Conhecer as funções matemáticas mais usadas
- Aprender a realizar manipulação de bits
- Entender algumas funções que ajudam a realizar atividades de entrada e saída
- Aprender a usar interrupções para estruturar a realização de diversas tarefas simultâneas

A linguagem do Arduino é baseada em uma biblioteca mais antiga denominada Wiring, que complementa uma outra biblioteca denominada Processing. A biblioteca Processing é muito similar à Wiring, mas se baseia na linguagem Java em vez de na C e é usada no seu computador para fazer a conexão com o Android através de USB. De fato, o aplicativo Arduino que você executa em seu computador é baseado em Processing. Se você estiver com vontade de escrever uma interface com um visual especial no seu computador para se comunicar com um Arduino, então dê uma olhada em Processing (www.processing.org).

Números aleatórios

Apesar das situações inesperadas que ocorrem com muitas pessoas quando estão trabalhando com um PC, os computadores são, na realidade, muito previsíveis. Às vezes, pode ser útil deliberadamente tornar imprevisível o Arduino. Por exemplo, você pode querer construir um robô que faça uma caminhada "aleatória" (randômica) em um quarto. Ele avança em uma direção durante um tempo aleatório, gira um número aleatório de graus e vai em frente novamente. Ou, com base no Arduino, você pode pensar em construir um dado de jogar que produz números entre 1 e 6.

A biblioteca padrão do Arduino fornece um recurso que faz exatamente isso: é a função denominada **random** (aleatório). A função **random** retorna um **int** e pode ter um ou dois argumentos. Se ela recebe um único argumento, então o número aleatório que ela fornece está entre zero e o valor do argumento menos um.

A versão com dois argumentos produz um número aleatório que está entre o primeiro argumento (inclusive) e o segundo argumento menos um. Assim, **random(1,10)** produz um número aleatório que está entre 1 e 9.

O sketch 7-01 produz números entre 1 e 6, os quais são enviados ao Serial Monitor.

```
//sketch 07-01
void setup()
{
   Serial.begin(9600);
}
void loop()
{
   int number = random(1, 7);
    Serial.println(number);
    delay(500);
}
```

Se você fizer o upload desse sketch para o seu Arduino e abrir o Serial Monitor, você verá algo como a Figura 7-1.

Depois de executá-lo algumas vezes, você provavelmente ficará surpreso de ver que você obtém sempre a mesma série de números "aleatórios".

A saída não é realmente aleatória. Os números são denominados *pseudoaleatórios* porque têm uma distribuição aleatória. Isto é, se você executasse esse sketch e obtivesse 1 milhão de

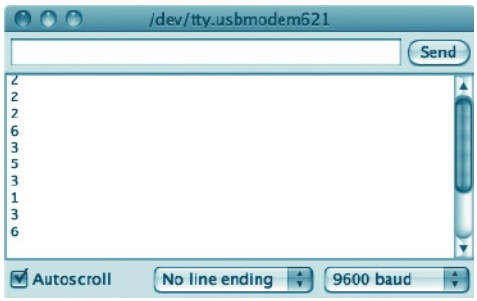

Figura 7-1 Números aleatórios.

números, as quantidades de uns, dois, três e assim por diante obtidas por você seriam muito aproximadamente as mesmas. Os números não são aleatórios no sentido de serem imprevisíveis. De fato, serem imprevisíveis é algo tão contra a natureza dos microcontroladores que eles não podem ser aleatórios sem que haja uma intervenção do mundo real.

Para que a sua sequência de números seja menos previsível, você pode fazer isso dando uma *semente* (seed, em inglês) ao gerador de números aleatórios. Isso basicamente dá um ponto de partida para a sequência. Mas, se você pensar a respeito, verá que não é possível simplesmente usar **random** para fornecer a semente do gerador aleatório de números. Um truque muito usado baseia-se no fato (discutido no capítulo anterior) de que uma entrada analógica flutua. Assim, você pode usar o valor lido em uma entrada aleatória como semente para o gerador de números aleatórios.

A função que faz isso é denominada **randomSeed**. O sketch 7-02 mostra como você pode conseguir mais aleatoriedade do seu gerador de números aleatórios.

```
//sketch 07-02

void setup()
{
   Serial.begin(9600);
   randomSeed(analogRead(0));
}

void loop()
{
   int number = random(1, 7);
   Serial.println(number);

   delay(500);
}
```

Aperte algumas vezes o botão de Reset. Agora você poderá ver como a sequência aleatória é diferente a cada vez.

Esse tipo de geração de números aleatórios não pode ser usado em tipo algum de loteria. Para conseguir uma geração bem melhor de números aleatórios, você precisaria usar a geração de números aleatórios por hardware, a qual algumas vezes se baseia em eventos aleatórios como, por exemplo, os raios cósmicos.

Funções matemáticas

Em raras ocasiões, você precisará fazer operações matemáticas complexas no Arduino, muito além das aritméticas. No entanto, se você precisar, está à sua disposição uma grande biblioteca com funções matemáticas. As mais úteis estão resumidas na tabela seguinte:

Função	Descrição	Exemplo
abs	Retorna o valor sem sinal do seu argumento.	abs(12) retorna 12 abs(-12) retorna 12
constrain	Restrição aplicada a um número para evitar que caia fora dos limites de um intervalo. O primeiro argumento é o número que sofrerá restrição. O segundo argumento é o início do intervalo e o terceiro argumento é o final do intervalo de valores que o número pode assumir.	constrain(8, 1, 10) retorna 8 constrain(11, 1, 10) retorna 10 constrain(0, 1, 10) retorna 1
map	Toma um número que está dentro de uma faixa de valores e o converte em um número que está dentro de outra faixa de valores. O primeiro argumento é o número que será convertido. O segundo e terceiro argumentos são os limites da faixa de valores srcinais. Os dois últimos números são os limites da faixa de valores convertidos. Essa função é útil quando se deseja mudar a faixa de valores de uma entrada analógica.	map(x, 0, 1023, 0, 5000)
max	Retorna o maior de seus dois argumentos.	max(10,11) retorna 11
min	Retorna o menor de seus dois argumentos.	min(10,11) retorna 10
pow	Retorna o primeiro argumento elevado ao expoente dado pelo segundo argumento.	pow(2, 5) retorna 32
sqrt	Retorna a raiz quadrada de um número.	sqrt(16) retorna 4
sin, cos, tan	Executa funções trigonométricas. Elas não são muito usadas.	
log	Calcula a temperatura de um termistor logarítmico (por exemplo).	

Manipulação de Bits

Um bit é um dígito simples de informação binária, isto é, ou 0 ou 1. A palavra *bit* é uma contração de *binary digit* (dígito binário). Na maior parte do tempo, você usa variáveis do tipo **int** que, na realidade, contêm 16 bits. Será um desperdício de bits se tudo o que você precisar for armazenar um único valor de verdadeiro ou falso (1 ou 0). Na verdade, a menos que você esteja trabalhando com pouca memória, o desperdício de bits não será um problema, quando

comparado com a criação de um código de difícil compreensão. No entanto, algumas vezes a capacidade de saber armazenar dados de forma compacta pode ser útil.

Cada bit de **int** pode ser pensado como correspondendo a um valor decimal. Você poderá encontrar o valor decimal do **int** somando os valores de todos os bits que são 1. Assim, na Figura 7-2, o valor decimal de **int** é 38. Na realidade, fica mais complicado quando se lida com números negativos, mas isso só acontece quando o bit mais à esquerda é 1.

Quando você está pensando em bits individuais, os valores decimais realmente não funcionam muito bem. Em um número decimal como 123, é muito difícil imaginar quais bits são 1. Por essa razão, os programadores usam frequentemente algo denominado *hexadecimal*, ou, mais comumente, apenas *hex*. Hex é um número de base 16. Assim, em vez de ter apenas os dígitos de 0 a 9, você tem seis dígitos extras, A até F. Isso significa que cada dígito hex representa quatro bits. A tabela seguinte mostra as relações existentes entre decimal, hex e binário para os números de 0 a 15:

Decimal	Hex	Binário (quatro dígitos)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Assim, em hex, qualquer **int** pode ser representado como um número hex de quatro dígitos. Por exemplo, o número binário 10001100 é 8C em hex. A linguagem C tem uma sintaxe especial para o uso de números hex. Você pode atribuir um valor hex a um **int** antepondo o prefixo 0x como mostrado a seguir:

```
int x = 0x8C;
```

A biblioteca padrão do Arduino fornece algumas funções que lhe permitem manipular individualmente cada um dos 16 bits dentro de **int**. A função **bitRead** usada para ler (read) bit retorna o valor de um dado bit de um **int**. Por exemplo, no caso seguinte, o valor 0 é atribuído à variável de nome **bit**:

```
int x = 0x8C; // 10001100
int bit = bitRead(x, 0);
```

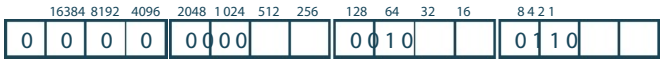

Figura 7-2 Um *int*.

No segundo argumento, a posição dos bits vai de 0 até 15, começando com o bit menos significativo. Portanto, indo da direita para a esquerda, o primeiro bit é o bit 0, o próximo bit é o bit 1, e assim por diante até o bit 15.

Como seria de esperar, o oposto de **bitRead** é o comando **bitWrite** usado para escrever (write) bit, que usa três argumentos. O primeiro é o número a ser manipulado, o segundo é a posição do bit e o terceiro é o valor do bit. O exemplo seguinte muda o valor de **int** de 2 para 3 (em decimal ou hex):

```
int x = 2; // 0010
bitWrite(x, 0, 1);
```

Entrada e saída avançadas

Há algumas pequenas funções que você pode usar para facilitar a sua vida quando for realizar várias atividades de entrada e saída.

Geração de som

A função **tone** (som, tom) permite a geração de um sinal de onda quadrada (veja a Figura 7-3) em um dos pinos digitais de saída. O motivo mais comum para fazer isso é a geração de um som audível usando um alto-falante ou um sinalizador sonoro.

A função pode ter dois ou três argumentos. O primeiro argumento é sempre o número do pino em que o sinal sonoro é gerado, o segundo argumento é a frequência do sinal sonoro em hertz (Hz) e o terceiro argumento, opcional, é a duração do sinal sonoro. Se a duração não for especificada, então o sinal sonoro será produzido continuamente, como é o caso no sketch 7-03. É por isso que colocamos a função **tone** no **setup** e não na função **loop**.

```
//sketch 07-03

void setup()
{
  tone(4, 500);
}

void loop() {}
```

Para interromper um som que está sendo gerado, você usa a função **noTone** Essa função tem apenas um argumento, que é o pino em que o sinal sonoro está sendo produzido.

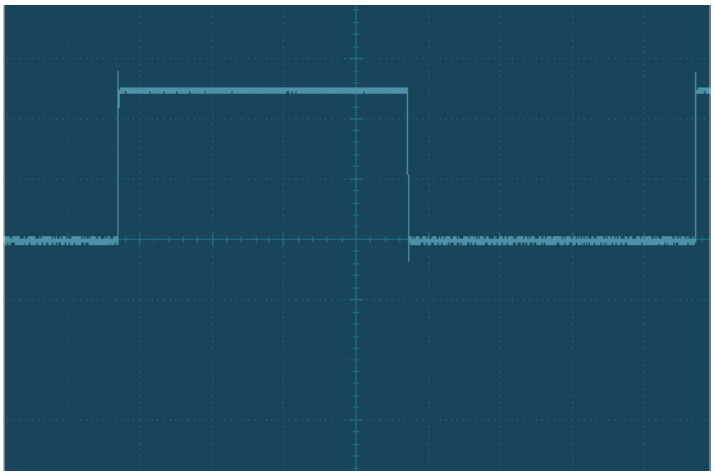

Figura 7-3 Um sinal de onda quadrada.

Alimentando registradores deslocadores

Algumas vezes, o Arduino não tem pinos em quantidade suficiente. Por exemplo, quando um grande número de LEDs é ligado, uma técnica comum é usar um chip com um registrador deslocador (shift). Esse chip lê um bit de cada vez dos dados e, quando já tem todos os bits necessários, ele os armazena de uma vez só em um conjunto de saídas (uma para cada bit).

Para ajudá-lo nessa técnica, há uma função prática denominada **shift-Out**. Ela recebe quatro argumentos:

- O número do pino em que aparecerá o bit que está sendo enviado.
- O número do pino que será usado como pulso de relógio. Sempre que um bit é enviado, ocorre um pulso.
- Um flag (sinalizador) para determinar se os bits serão enviados, começando primeiro (first) pelo bit menos significativo ou pelo bit mais significativo. Isso pode ser uma das constantes **MSBFIRST** ou **LSBFIRST.***
- O byte de dados que será enviado.

Interrupções

Umas das coisas que tendem a frustrar os programadores de sistemas de grande porte é que o Arduino só pode fazer uma coisa de cada vez. Se você gosta de ter diversas tarefas sendo

* N. de T.: Em inglês, bit mais significativo é Most Significant Bit (MSB) e bit menos significativo é Least Significant Bit (LSB).

executadas ao mesmo tempo em seus programas, então você não está com sorte. Nos tipos de usos que são implementados com o Arduino, geralmente não há necessidade dessa capacidade, embora haja projetos desenvolvidos por algumas pessoas que podem realizar tarefas múltiplas desse modo. O mais próximo que um Arduino consegue chegar dessa forma de execução é usando interrupções.

Dois dos pinos do Arduino (D2 e D3) podem funcionar como pinos de interrupção. Isto é, esses pinos podem atuar cada um como uma entrada que, se receber um sinal segundo uma forma especificada, então o processador do Arduino interromperá o que estiver fazendo, seja lá o que for, e executará uma função associada a essa interrupção.

O sketch 7-04 faz um LED piscar, mas o período desse pisca-pisca muda quando uma interrupção é recebida. Você pode simular uma interrupção conectando um fio entre os pinos D2 e GND e usando o resistor interno de pull-up para manter a interrupção em nível alto normalmente.

```
//sketch 07-04

int interruptPin = 2;
int ledPin = 13;
int period = 500;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT);

  digitalWrite(interruptPin, HIGH); // pull-up
  attachInterrupt(0, goFast, FALLING);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
   delay(period);
  digitalWrite(ledPin, LOW);
   delay(period);
}

void goFast()
{
   period = 100;
}
```

Nesse sketch, a linha-chave da função **setup** é a seguinte:

```
attachInterrupt(0, goFast, FALLING);
```

O primeiro argumento especifica qual das duas interrupções você deseja usar. Isso é um tanto confuso, pois um 0 significa que você está usando o pino 2, ao passo que um 1 significa que você está usando o pino 3.

O segundo argumento é o nome da função que deve ser chamada quando há uma interrupção e o terceiro argumento é uma constante que pode ser **CHANGE** (mudança), **RISING** (para cima) ou **FALLING** (para baixo). A Figura 7-4 resume essas opções.

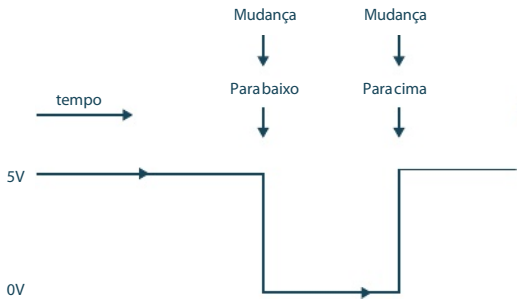

Figura 7-4 Tipos de sinais de interrupção.

Se o modo de interrupção é CHANGE (mudança), então uma interrupção será disparada tanto por uma mudança de nível do tipo RISING (para cima) de 0 para 1, como por uma mudança de nível do tipo FALLING (para baixo) de 1 para 0.

Você pode desabilitar as interrupções usando a função **noInterrupts** (nenhuma interrupção). Isso detém todas as interrupções de ambos os canais de interrupção. Você pode habilitar novamente o uso de interrupções chamando a função **interrupts** (interrupções).

Conclusão

Neste capítulo, você viu alguns dos recursos práticos disponibilizados pela biblioteca padrão do Arduino. Esses recursos irão lhe poupar trabalho de programação. Se há uma coisa que um bom programador gosta, é usar os trabalhos bem feitos já realizados por outras pessoas.

No próximo capítulo, iremos ampliar o que aprendemos sobre as estruturas de dados do Capítulo 5. Veremos também como, após desligar o Arduino, você pode conservar os dados.

» capítulo 8

Armazenamento de dados

Quando você dá valores às variáveis, a placa do Arduino conservará esses dados na sua memória somente enquanto estiver ligado. No momento em que você desligá-lo ou der reset na placa, todos os dados serão perdidos.

Neste capítulo, iremos examinar algumas maneiras de manter esses dados.

Objetivos deste capítulo

- Aprender maneiras de armazenar dados depois de desligar o Arduino
- Aprender a usar a diretiva PROGMEM
- Aprender a usar a memória EEPROM para armazenar dados permanentes que podem ser alterados
- Aprender o modo mais eficiente de representar os dados quando eles são maiores que o espaço disponível

Constantes

Se os dados que você deseja armazenar são sempre os mesmos, então você pode simplesmente inicializar os dados sempre que o Arduino inicia a execução de um sketch. Um exemplo dessa abordagem é o caso do array de letras do seu tradutor de código Morse do Capítulo 5 (sketch 5-05).

Você usou o seguinte código para definir uma variável do tamanho correto e preenchê-la com os dados que você precisava:

```
char* letters[] = {
   ".-", "-...", "-.-.", "-..", ".",
  "..-.", "--.", "....", "..",          // A-I
   ".---", "-.-", ".-..", "--", "-.",
  "---", ".--.", "--.-", ".-.",         // J-R
   "...", "-", "..-", "...-", ".--",
"-..-", "-.--", "--.."                 // S-Z
};
```

Você talvez ainda lembre que, depois de fazer os cálculos, você decidiu que havia espaço suficiente para ser usado nos seus magros 2K. Entretanto, se a memória fosse mais escassa, teria sido melhor armazenar esses dados nos 32K da memória flash – usada para armazenar programas – em vez de nos 2K da RAM. Há um modo de fazer isso. Trata-se de uma diretiva denominada **PROGMEM**. Ela mora em uma biblioteca e seu uso é complicado.

A diretiva PROGMEM

Para armazenar os seus dados na memória flash, você tem que incluir a biblioteca **PROGMEM** como segue:

```
#include <avr/pgmspace.h>
```

A finalidade desse comando é dizer ao compilador que a biblioteca **pgmspace** deverá ser usada nesse sketch. Nesse caso, uma biblioteca é um conjunto de funções que alguém mais escreveu e que você pode usar em seus sketches sem necessidade de compreender todos os detalhes de como funcionam essas funções.

Quando você usa essa biblioteca, a palavra-chave **PROGMEM** e a função de leitura **pgm_read_word** estão disponíveis. Você irá usá-las nos sketches que seguem.

Essa biblioteca está incluída no software do Arduino e é uma biblioteca oficialmente suportada. Uma boa coleção dessas bibliotecas oficiais está disponível, e muitas bibliotecas não oficiais, desenvolvidas por pessoas como você e feitas para serem usadas por outros, também estão disponíveis na Internet. Essas bibliotecas não oficiais devem ser instaladas no seu ambiente Arduino. No Capítulo 11, você aprenderá mais sobre essas bibliotecas e também verá como escrever as suas próprias bibliotecas.

Quando você usa **PROGMEM**, você deve se assegurar de que serão usados tipos especiais de dados adequados à biblioteca **PROGMEM**. Infelizmente, ela não inclui um array contendo arrays **char**. Na realidade, você tem que definir uma variável para cada string usando um tipo

de string **PROGMEM** e então colocando todas elas em um tipo de array **PROGMEM**, como mostrado a seguir:

```
PROGMEM prog_char sA[] = ".-";
PROGMEM prog_char sB[] = "-...";
// e assim por diante para todas as letras
PROGMEM const char* letters[] =
{
   sA, sB, sC, sD, sE, sF, sG, sH, sI, sJ, sK, sL, sM,
   sN, sO, sP, sQ, sR, sS, sT, sU, sV, sW, sX, sY, sZ
};
```

Eu não listei o sketch 8-01 aqui porque é um tanto longo, mas você pode carregá-lo e verificar que ele trabalha do mesmo modo que a versão baseada em RAM.

Além de especialmente criar dados dessa forma, você também deverá ter um modo especial para ler os dados de volta. Para que seu código possa pegar a string com o código Morse de uma letra, você terá que modificar o código, ficando como segue:

```
strcpy_P(buffer, (char*)pgm_read_word(&(letters[ch - 'a'])));
```

Esse comando usa uma variável de nome **buffer**, na qual a string **PROGMEM** é copiada, podendo ser usada como um array **char** comum. Para isso, é necessário definir uma variável global, como mostrado a seguir:

```
char buffer[6];
```

Esse modo funcionará apenas se os dados forem constantes, isto é, não mudam enquanto o sketch estiver sendo executado. Na próxima seção, você aprenderá a usar a memória EEPROM, cuja finalidade consiste em armazenar dados permanentes que podem ser alterados.

» *EEPROM*

O ATMega328, que está no coração do Arduino Uno, tem um quilobyte de EEPROM (Electrically-Erasable Programmable Read-Only Memory)*. Uma EEPROM é projetada para poder lembrar o seu conteúdo por muitos anos. Na realidade, apesar do nome, ela não é apenas de leitura. Você pode escrever nela.

Os comandos do Arduino para ler e escrever na EEPROM são tão trabalhosos como os usados com a **PROGMEM**. Em uma EEPROM, você deve ler ou escrever um byte de cada vez.

O exemplo do sketch 8-02 permite que você forneça o código de uma letra de dígito simples no Serial Monitor. Então, o sketch memoriza o dígito e o envia repetidamente para ser exibido no Serial Monitor.

```
// sketch 08-02
#include <EEPROM.h>

int addr = 0;
char ch;
```

* N. deT.: Memória Apenas de Leitura, Programável e Eletricamente Apagável.

```
void setup()
{
   Serial.begin(9600);
  ch = EEPROM.read(addr);
}
void loop()
{
  if (Serial.available() > 0)
   {ch = Serial.read();
   EEPROM.write(0, ch);
      Serial.println(ch);
   }
   Serial.println(ch);
   delay(1000);
}
```

Para usar esse sketch, abra o Serial Monitor e entre com um novo caractere. Então, desligue e ligue novamente o Arduino. Quando você voltar a abrir o Serial Monitor, você verá que a letra foi lembrada.

A função **EEPROM.write** usa dois argumentos. O primeiro é o endereço, que é a posição na memória EEPROM e que deve estar entre 0 e 1023. O segundo argumento é o dado que será escrito naquela posição. Esse dado deve ter um único byte. Um caractere é representado por oito bits. Isso é bom, mas você não pode armazenar diretamente um **int** de 16 bits.

Armazenando um valor *int* em uma EEPROM

Para armazenar um **int** de dois bytes nas posições 0 e 1 da EEPROM, você deve fazer o seguinte:

```
  int x = 1234;
 EEPROM.write(0, highByte(x));         \\ byte alto ou mais significativo
 EEPROM.write(1, lowByte(x));         \\byte baixo ou menos significativo
```

As funções **highByte** (byte alto, ou byte mais significativo) e **lowByte** (byte baixo, ou byte menos significativo) são úteis para dividir um **int** em dois bytes. A Figura 8-1 mostra como esse **int** está realmente armazenado na EEPROM.

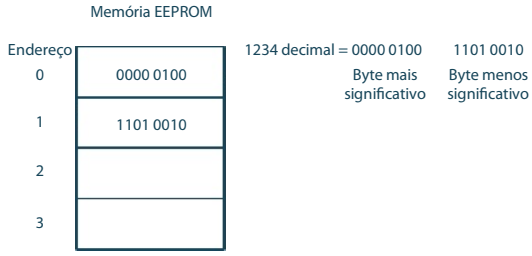

Figura 8-1 Armazenando um inteiro de 16 bits em uma EEPROM.

Para ler o **int** na EEPROM, você precisa ler os dois bytes da EEPROM e reconstruir o **int** como segue:

```
byte high = EEPROM.read(0);      \\ byte da parte mais significativa
byte low = EEPROM.read(1);       \\ byte da parte menos significativa
int x = (high << 8) + low;
```

O operador << é um operador de deslocamento de bits que move o byte da variável high para o local correspondente ao byte mais significativo de **int x** e, em seguida, adiciona o byte da variável low*.

Armazenando um Valor *float* em uma EEPROM (Union)

O armazenamento de um **float** na EEPROM é um pouco mais trabalhoso. Para fazer isso, você pode usar um recurso do C denominado **union** (união). Essa estrutura de dados é interessante porque pode ser pensada como uma maneira de tornar a mesma área da memória acessível a mais de uma variável. E mais ainda, essas variáveis podem ser de tipos diferentes desde que tenham o mesmo tamanho em bytes.

A seguinte definição de **union** permite que um **float** e um **int** refiram-se aos mesmos dois bytes de memória:

```
union data
{
  float f;
  int i;
} convert;
```

Agora, você pode colocar um **float** na **union** como segue:

```
  float f = 1.23;
  convert.f = f;
```

Então, você poderá separar um inteiro em seus dois bytes para serem armazenados na EEPROM, como segue:

```
  EEPROM.write(0, highByte(convert.i));
  EEPROM.write(1, lowByte(convert.i));
```

Para ler como **float**, é necessário que você faça o inverso. Primeiro, você combina os dois bytes em um **int** simples. Então, você coloca o **int** na **union** e lê novamente como **float**.

```
byte high = EEPROM.read(0);
byte low = EEPROM.read(1);
convert.i = (high << 8) + low;
float f = convert.f;
```

* N. de T.: A variável *high* (parte mais significativa) e a variável *low* (parte menos significativa) têm ambas um byte, e a variável *x* tem dois bytes. O deslocamento faz *high* ocupar o byte mais significativo de *x*. Depois da soma, o byte de *low* ocupará o byte menos significativo de *x*, resultando um int *x* de dois bytes.

Armazenando uma String em uma EEPROM

A escrita e a leitura de strings de caracteres na EEPROM são bem imediatas. Você deve simplesmente escrever um caractere de cada vez, como no exemplo seguinte:

```
char *test = "Alô";
int i = 0;
while (test[i] != '\0')
{
 EEPROM.write(i, test[i
   i++;
}
EEPROM.write(i, '\0');
```

Para ler a string de volta na forma de um array de caracteres, você pode fazer algo como o seguinte:

```
char test[10];
int i = 0;
char ch;
ch = EEPROM.read(i)
while (ch != '\0' && i < 10)
{
  test[i] = ch;
  ch = EEPROM.read(i);
   i++;
}
```

Limpando os conteúdos de uma EEPROM

Quando você escreve em uma EEPROM lembre-se de que, mesmo que você faça o upload de um novo sketch, isso não limpará a EEPROM. Você poderá encontrar valores de um projeto anterior. O sketch 8-03 limpa todo o conteúdo da EEPROM preenchendo-a com zeros.

```
// sketch 08-03
#include <EEPROM.h>

void setup()
{
   Serial.begin(9600);
  Serial.println("Clearing EEPROM")// Limpando a EEPROM
  for (int i = 0; i < 1023; i++)
   {
   EEPROM.write(i, 0);
   }
  Serial.println("EEPROM cleared")// EEPROM Limpa
}

void loop()
{
}
```

Leve em consideração também que você pode escrever em uma posição de EEPROM apenas umas 100 mil vezes antes que deixe de ser confiável. Portanto, só escreva um valor em uma

EEPROM quando você realmente tiver necessidade. Uma EEPROM também é bem lenta, necessitando de aproximadamente 3 milissegundos para que seja escrito um byte.

Compressão

Quando você guarda dados em uma EEPROM ou quando você usa **PROGMEM**, você verá que algumas vezes você tem mais dados para guardar do que espaço disponível. Quando isso acontece, vale a pena descobrir qual é o modo mais eficiente de representar os dados.

Compressão de faixa

Você pode ter um valor que deve ser do tipo **int** ou **float**, ambos de 16 bits. Por exemplo, para representar uma temperatura em graus Celsius, você pode precisar de um **float** como 20.25. Se você o armazenasse na EEPROM, a vida seria bem mais fácil se você pudesse colocá-lo em um único byte. Além disso, você poderia armazenar o dobro de dados se não usasse **float**.

Uma maneira de fazer isso é mudando os dados antes de armazená-los. Lembre-se de que um byte permite armazenar um número positivo entre 0 e 255. Assim, se para você for suficiente conhecer a temperatura arredondada até o grau Celsius inteiro mais próximo, então você poderá simplesmente converter o **float** em um **int** e descartar a parte fracionária depois do ponto decimal. O exemplo seguinte mostra como fazer isso:

```
int tempInt = (int)tempFloat;
```

A variável **tempFloat** contém o valor em ponto flutuante. O comando **(int)** é denominado um *type cast* e é usado para converter uma variável de um tipo em outro compatível. Nesse caso, o *type cast* converte o **float** 20.25 (por exemplo) em um **int** simplesmente truncando a parte fracionária do número e ficando apenas a parte inteira 20.

Se você souber que a temperatura máxima que lhe interessa é 60 graus Celsius e que a temperatura mínima é 0 grau Celsius, então você poderá multiplicar as temperaturas por 4 antes de converter cada uma em um byte, salvando-o em seguida. Quando você ler o dado de volta da EEPROM, você o divide por 4 para obter um valor que tem uma precisão de 0.25.

No exemplo de código seguinte, o sketch 8-04 salva essa temperatura na EEPROM e, em seguida, a lê de volta, exibindo-a no Serial Monitor como prova:

```
//sketch 8-04

#include <EEPROM.h>
void setup()
{
  float tempFloat = 20.75;
  byte tempByte = (int)(tempFloat * 4);
 EEPROM.write(0, tempByte);

  byte tempByte2 = EEPROM.read(0);
  float temp2 = (float)(tempByte2) / 4;
   Serial.begin(9600);
```

```
    Serial.println("\n\n\n");
    Serial.println(temp2);
}
void loop(){}
```

Há outros meios de compactar os dados. Por exemplo, se você estiver fazendo leituras de temperatura que variam muito lentamente – de novo, variações de temperatura são um bom exemplo disso –, então você precisa registrar a primeira temperatura com precisão máxima e, a partir daí, registrar apenas a variação (diferença) de temperatura entre uma leitura e a anterior. Essa variação normalmente será pequena e ocupará poucos bytes.

Conclusão

Agora você já conhece o básico para que os seus dados fiquem preservados depois de desligar o Arduino. No próximo capítulo, você verá os displays LCD.

» capítulo 9

Displays LCD

Neste capítulo, você verá como escrever software para controlar displays LCD. A Figura 9-1 mostra o tipo de display LCD que será usado.

Este é um livro sobre software e não hardware, mas neste capítulo teremos que explicar um pouco do funcionamento da eletrônica desses displays para compreender o modo de acioná-los.

O módulo LCD que usaremos é um shield pré-fabricado para Arduino, que pode ser simplesmente encaixado em cima de uma placa de Arduino. Além do seu display, ele também tem alguns botões. Há muitos shields diferentes, mas a grande maioria usa o mesmo chip controlador de LCD (o HD44780). Sendo assim, procure utilizar um shield que use esse chip controlador.

Objetivos deste capítulo

- Entender um pouco do funcionamento da eletrônica dos displays LCD para saber como acioná-los
- Aprender a usar algumas funções da biblioteca de LCD

Eu usei o LCD Keypad Shield da DFRobot para Arduino. Esse módulo fornecido pela DFRobot (www.robotshop.com) é barato e contém um display LCD de 16 caracteres por duas linhas e também tem seis botões de pressão.

O shield vem montado, de modo que não há necessidade de soldagem. Você simplesmente o encaixa em cima da placa do Arduino (veja a Figura 9-2).

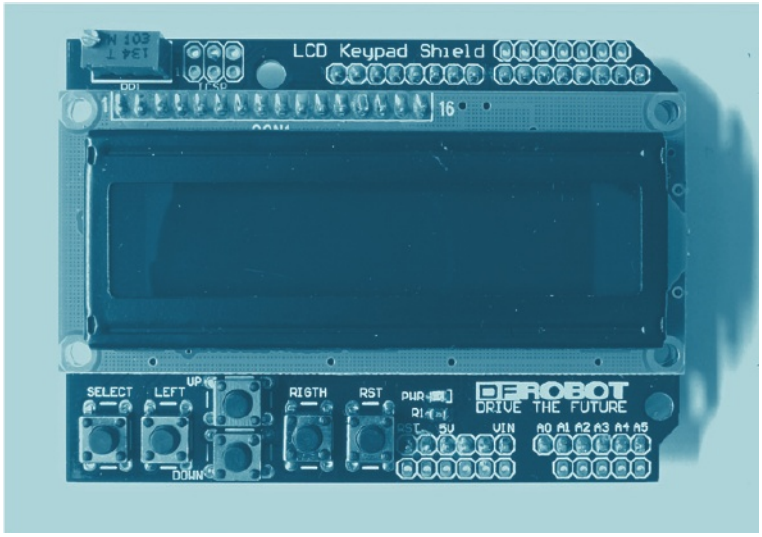

Figura 9-1 Um shield com LCD alfanumérico.

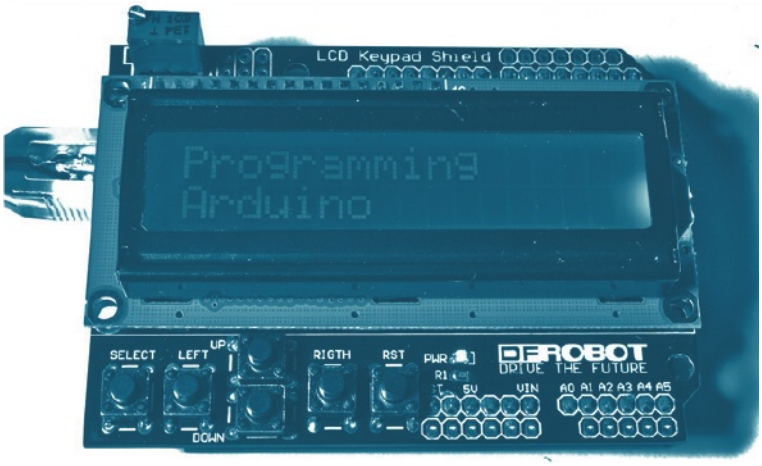

Figura 9-2 Um shield de LCD acoplado a uma placa de Arduino.

O shield LCD usa sete dos pinos do Arduino para controlar o display LCD e um pino analógico para os botões. Assim, não poderemos usar esses pinos do Arduino para outra finalidade.

Uma placa USB de mensagens

Como exemplo de uso simples do display, faremos uma placa USB de mensagens. Esse módulo irá exibir mensagens enviadas pelo Serial Monitor.

O IDE do Arduino vem com uma biblioteca para LCD. Isso simplifica bastante o processo de usar um display LCD. A biblioteca fornece funções úteis que você pode chamar, tais como:

- **clear** limpa o display, deixando-o sem nenhum texto.
- **setCursor** ajusta a posição do local onde aparecerá a próxima coisa que você quer mostrar. A posição é definida por uma coluna e uma linha.
- **print** escreve uma string nessa posição.

```
// sketch 09-01 Placa USB de Mensagens

#include <LiquidCrystal.h>

// lcd(RS, E, D4, D5, D6, D7)
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);
int numRows = 2; // número de linhas (rows)
int numCols = 16; // número de colunas

void setup()
{
   Serial.begin(9600);
   lcd.begin(numRows, numCols);  // Número de linhas e colunas
lcd.clear();                   // Limpa o display
  lcd.setCursor(0,0);            // Cursor bem à esquerda na primeira linha
  lcd.print("Arduino");          // Exibe " O Arduino"
  lcd.setCursor(0,1);            // Cursor bem à esquerda na segunda linha
 lcd.print("Rules");            // Exibe "Rules" ("Comanda")
}

void loop()
{
   if (Serial.available() > 0)
    {
   char ch = Serial.read();
   if (ch == '#')

   {  lcd.clear();
   }
   else if (ch == '/')
   {
    // nova linha
    lcd.setCursor(0, 1);
   }
```

```
    else
    {
      lcd.write(ch);
      }
    }
}
```

Como em todas as aplicações do Arduino, você deve começar incluindo a biblioteca para que o compilador saiba que ela está disponível.

A próxima linha define quais pinos do Arduino serão usados pelo shield e para que finalidade. Se você usar um outro shield, poderá acontecer que os pinos usados sejam diferentes. Portanto, verifique como os pinos são usados na documentação do shield.

Neste caso, os seis pinos usados para controlar o display são D4, D5, D6, D7, D8 e D9. A finalidade de cada um desses pinos está descrita na Tabela 9-1.

A função **setup** é simples. Você inicia a comunicação serial para que o Serial Monitor possa enviar comandos e inicializar a biblioteca LCD com as dimensões (linhas e colunas) do display que será usado. Você também exibirá a mensagem "Arduino Rules" ("Regras do Arduino") em duas linhas, colocando primeiro o cursor no ponto mais à esquerda da linha de cima e mandando exibir a mensagem "Arduino". Em seguida, você move o cursor para o início da segunda linha e manda exibir "Rules".

A maior parte da ação ocorre na função **loop**, a qual continuamente verifica se há algum caractere chegando do Serial Monitor. O sketch lida com os caracteres um a um.

Além dos caracteres comuns que o sketch exibe, há também dois caracteres especiais. Se o caractere for um #, então o sketch limpa o display inteiro, e se o caractere for um /, o sketch vai para a segunda linha. Caso contrário, o sketch simplesmente exibe o caractere na posição corrente do cursor usando **write**. A função **write** é como **print**, mas exibe apenas um caractere em vez de uma string de caracteres.

Tabela 9-1 Atribuições de pinos para o shield de LCD

Parâmetros para LCD()	Pino do Arduino	Finalidade
RS	8	Register Select (Seleção de Registrador). Esse pino é ajustado para 1 ou 0, dependendo se o Arduino está enviando dados de caractere ou de instrução. Por exemplo, uma instrução pode fazer o cursor piscar.
E	9	Enable (Habilita). A saída desse pino apresenta um pulso para dizer que os dados dos quatro pinos seguintes estão prontos para serem lidos.
Data4	4	Esses quatro pinos são usados para transferir dados. O chip controlador do LCD usado pelo shield pode operar com dados de oito ou quatro bits. O shield usado aqui trabalha com quatro bits. Nesse caso, são usados os bits 4–7 em vez de 0–7.
5 Data	5	
6 Data	6	
7 Data	7	

Usando o display

Experimente o sketch 9-01 transferindo-o para a placa e, em seguida, encaixando o shield. Lembre-se de que você sempre deve desligar eletricamente a placa do Arduino antes de conectar um shield.

Abra o Serial Monitor e experimente escrever o texto mostrado na Figura 9-3.

Outras funções da biblioteca LCD

Além das funções que você já utilizou neste exemplo, há diversas outras que você poderá usar:

- **home** é o mesmo que **setCursor(0,0)**: ela move o cursor para a posição mais à esquerda da linha de cima.
- **cursor** exibe o cursor.
- **noCursor** especifica que o cursor não deve ser exibido.
- **blink** faz o cursor piscar.
- **noBlink** faz o cursor parar de piscar.
- **noDisplay** desliga o display sem remover o conteúdo.
- **display** volta a ligar o display após um **noDisplay**.
- **scrollDisplayleft** faz todo o texto do display ser deslocado (scroll) uma posição de caractere para a esquerda (left).
- **scrollDisplayright** faz todo o texto do display ser deslocado (scroll) uma posição de caractere para a direita (right).
- **autoscroll** ativa um modo no qual, à medida que novos caracteres são acrescentados na posição do cursor, o texto existente é deslocado (scroll) no sentido determinado pela função **leftToRight** (esquerda para direita) ou **rightToLeft** (direita para esquerda).
- **noAutoscroll** desliga o modo **autoscroll**.

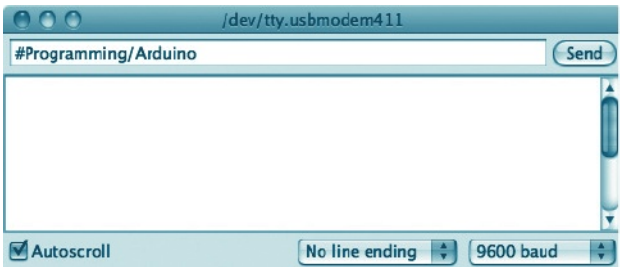

Figura 9-3 Enviando comandos ao display.

Conclusão

Você pode ver que a programação de shields não é difícil, particularmente quando há uma biblioteca capaz de fazer uma grande parte do trabalho.

No próximo capítulo, você aprenderá a usar um shield de Ethernet que permite a conexão do Arduino à Internet.

» capítulo 10

Programação Ethernet do Arduino

Neste capítulo, você usará um shield de Ethernet que capacitará o seu Arduino a trabalhar com a sua rede doméstica (veja a Figura 10-1).

Objetivos deste capítulo

- Conhecer um pouco sobre HTTP e HTML
- Aprender a usar um shield de Ethernet permitido que o Arduino funcione como servidor de web
- Conhecer algumas funções da biblioteca padrão para interface com o shield de Ethernet
- Aprender a usar postagem de dados permitindo ajustar os pinos do Arduino através de uma rede

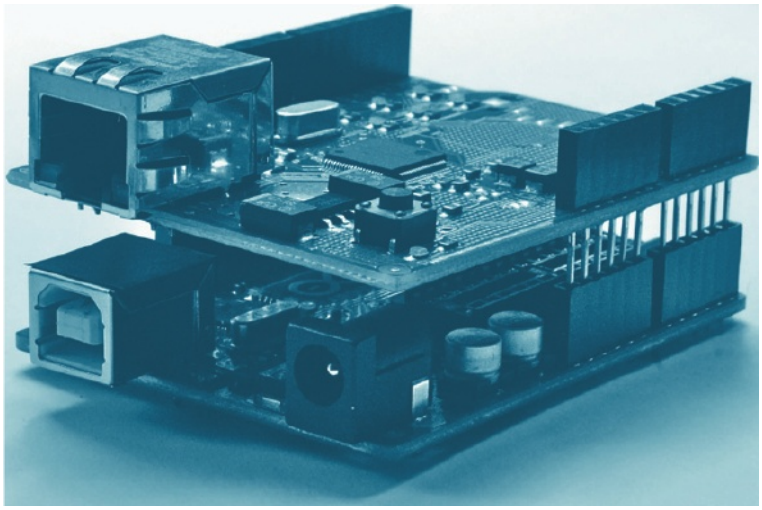

Figura 10-1 Arduino com Ethernet.

Shields de Ethernet

Quando você compra um shield de Ethernet, você deve tomar um certo cuidado. Você precisa usar um shield "oficial" baseado no chipset Wiznet *e não* um shield com uma placa mais barata e mais difícil de usar, baseada no chip controlador de Ethernet ENC28J60.

Os shields de Ethernet consomem muita energia elétrica. Assim, você precisará de uma fonte de alimentação de 9V ou 12V, capaz de suprir 1A ou mais. Essa fonte de alimentação será ligada ao conector de alimentação elétrica do Arduino.

Comunicação com servidores de web

Antes de ver como o Arduino lida com a comunicação entre um navegador e o servidor de web que ele está usando, você precisa conhecer algumas coisas sobre HyperText Transfer Protocol (HTTP) e HyperText Markup Language (HTML).

HTTP

O HyperText Transfer Protocol é o método usado pelos navegadores de web para se comunicar com um servidor de web.

Quando você visita uma página usando um navegador de web, o navegador envia uma solicitação ao servidor que hospeda aquela página dizendo o que deseja. O que o navegador pede pode ser simplesmente o conteúdo de uma página HTML. O servidor de web está sem-

pre prestando atenção a essas solicitações e, quando recebe uma, ele a processa. Neste caso simples, o processamento da solicitação significa simplesmente enviar de volta a página HTML que foi especificada por você no sketch do Arduino.

HTML

A HyperText Markup Language é uma maneira de acrescentar informações de formatação a um texto comum para que ele fique visualmente bem apresentável quando o navegador exibi-lo. Por exemplo, o seguinte código é em HTML e, quando ele é usado para ser mostrado em uma página de navegação, ele será exibido como na Figura 10-2:

```
<html>
<body>
<h1>Programming Arduino</h1>
<p>A book about programming Arduino</p>
</body>
</html>
```

A HTML contém tags (etiquetas ou rótulos). As tags podem ser de abertura e de fechamento, contendo normalmente outras tags entre elas. Uma tag de abertura consiste em um <, um nome de tag e um >, como por exemplo **<html>**. A correspondente tag de fechamento é similar, exceto que tem um / após o <. No exemplo anterior, a tag mais externa é **<html>**, a qual contém outra tag denominada **<body>** (corpo). Todas as páginas da web devem começar com essas tags e você poderá ver as correspondentes tags de fechamento no fim do arquivo. Observe que você deve colocar as tags na ordem correta. Portanto, a tag **body** deve ser fechada antes da tag **html**.

Agora vamos a um trecho interessante localizado no meio do código HTML, as tags **h1** e **p**. Essas são as partes do exemplo que serão realmente exibidas.

A tag **h1** indica um cabeçalho (header) de nível 1. O efeito é que o texto contido nela é exibido com uma fonte de tamanho grande em negrito. A tag **p** é uma tag de parágrafo, fazendo todo o texto contido nela ser exibido como um parágrafo.

Figura 10-2 Um exemplo de HTML.

Isso mal arranha a superfície dos tópicos relativos à HTML. Muitos livros e recursos de Internet estão disponíveis para quem quiser aprender HTML.

O Arduino como servidor de web

O primeiro exemplo de sketch simplesmente usa o Arduino e o shield de Ethernet para construir um pequeno servidor de web. Definitivamente, ele não tem o porte de um servidor Google, mas permitirá que você envie uma solicitação de web para o seu Arduino e veja os resultados no navegador do seu computador.

Antes de fazer o upload do sketch 10-01, há duas modificações que você precisa fazer. Se você olhar na parte inicial do sketch, você verá as seguintes linhas:

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192, 168, 1, 30 };
```

A primeira delas, o endereço **mac**, deve ser único entre todos os dispositivos que estiverem conectados à sua rede. A segunda é o endereço IP. A maioria dos dispositivos que você liga na sua rede doméstica tem os seus endereços de IP atribuídos automaticamente por um protocolo denominado Dynamic Host Configuration Protocol (DHCP). Entretanto, no caso do shield de Ethernet, isso não é verdadeiro. No nosso dispositivo, você terá que definir manualmente um endereço de IP. Esse endereço não pode ser dado por quatro números quaisquer. Devem ser números que os validem como endereços internos de IP e que estejam dentro do intervalo de endereços esperados por seu roteador doméstico. Tipicamente, os três primeiros números serão algo como 10.0.1.*x* ou 192.168.1.*x*, onde *x* é um número entre 0 e 255. Alguns desses números estarão sendo usados por outros dispositivos da sua rede. Para encontrar um número de IP que não esteja sendo usado, mas que seja válido, conecte-se com a página do administrador do seu roteador doméstico e procure uma opção que diz "DHCP". Você deverá encontrar uma lista de dispositivos com os seus endereços de IP semelhante à mostrada na Figura 10-3. Selecione um número final para ser usado no seu endereço de IP. Neste caso, 192.168.1.30 pareceu uma boa tentativa e, realmente, funcionou muito bem.

Ligue o Arduino ao seu computador usando o cabo USB e faça o upload do sketch. Agora você pode desconectar o cabo USB e ligar o cabo Ethernet e a fonte de alimentação ao seu Arduino.

No navegador do seu computador, abra uma conexão para o endereço de IP que você atribuiu ao shield de Ethernet. Deverá aparecer algo muito semelhante ao que está na Figura 10-4.

A listagem do sketch 10-01 com um exemplo de servidor simples é a seguinte:

```
// sketch 10-01 Um Exemplo Simples de Servidor

#include <SPI.h>
#include <Ethernet.h>

// O endereço MAC deve ser único. O seguinte deve funcionar:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

Figura 10-3 Encontrando um endereço de IP não usado.

Figura 10-4 Um exemplo simples de servidor com Arduino.

```
// O endereço de IP dependerá da sua rede local:
byte ip[] = { 192, 168, 1, 30 };

Server server(80);

void setup()
{
  Ethernet.begin(mac, ip);
   server.begin();
   Serial.begin(9600);
}

void loop()
{
```

```
  // aguardando por solicitações de clientes
  Client client = server.available();
 if (client)
  {
  while (client.connected())
     {
       // envie uma resposta com um cabeçalho http padrão
     client.println("HTTP/1.1 200 OK");
    client.println("Content-Type: text/html");
        client.println();
       // envie o corpo (body) do arquivo HTML
        client.println("<html><body>");
    client.println("<h1>Arduino Server</h1>");
        client.print("<p>A0=");
        client.print(analogRead(0));
        client.println("</p>");
        client.print("<p>millis=");
        client.print(millis());
        client.println("</p>");
        client.println("</body></html>");
        client.stop();
     }
     delay(1);
  }
}
```

Assim como ocorreu com a biblioteca LCD discutida no Capítulo 9, uma biblioteca padrão do Arduino se encarregará da interface com o shield de Ethernet.

A função **setup** inicializa a biblioteca Ethernet usando os endereços de **mac** e IP que você definiu antes.

A função **loop** é responsável por atender todas as solicitações que chegam ao servidor de web oriundas de um navegador. Se uma solicitação estiver esperando por uma resposta, então, quando chamamos **server.available**, teremos como retorno um cliente. Um cliente é um objeto. Você aprenderá mais sobre o que isso significa no Capítulo 11. Por enquanto, tudo que você precisa saber é se há um cliente esperando (teste feito pelo primeiro comando **if**). A seguir, chamando**client.connected**, você poderá determinar se ele está conectado ao servidor de web.

As três próximas linhas de código enviam um cabeçalho (header) de retorno. Isso simplesmente diz ao navegador qual é o tipo de conteúdo que será exibido. Neste caso, o navegador mostrará um conteúdo HTML.

Depois do envio do cabeçalho, tudo o que falta fazer é enviar o restante do código HTML ao navegador. Isso deve incluir as tags usuais de **<html>** e **<body>** e também uma tag de cabeçalho **<h1>** e duas tags **<p>** que exibirão o valor da entrada analógica A0 e o valor retornado pela função **millis**. Este valor é o número de milissegundos decorridos desde que o Arduino foi inicializado pela última vez.

Finalmente, **client.stop** diz ao navegador que a mensagem está completa, e o navegador exibirá então a página.

Ajustando os pinos do Arduino através de uma rede

Este segundo exemplo de uso do shield de Ethernet permite que os pinos D3 a D7 sejam ligados ou desligados usando um formulário web (web form).

Diferentemente do exemplo de servidor simples, agora você deverá encontrar um modo de passar ao Arduino os ajustes dos pinos.

O método para fazer isso é denominado *posting data* (postagem de dados) e é parte do protocolo padrão HTTP. Para que esse método funcione, você deverá construir o mecanismo de posting dentro do HTML de modo que o Arduino retorne dados do tipo HTML ao navegador para que este exiba um formulário (form). O formulário (mostrado na Figura 10-5) tem uma seleção de ligado (On) ou desligado (Off) para cada pino e um botão de atualização (update) que enviará os ajustes dos pinos para o Arduino.

Quando o botão de Update é pressionado, uma segunda solicitação é enviada ao Arduino. Ela será como a primeira solicitação, exceto que a solicitação conterá parâmetros com os valores de ajustes para os pinos.

Conceitualmente, um parâmetro de solicitação é similar a um parâmetro de função. Um parâmetro de função permite que você passe informação a uma função, tal como o número de vezes que um LED deve piscar. Um parâmetro de solicitação permite que você passe dados ao Arduino quando ele atende à solicitação de web. Quando o Arduino recebe a solicitação de web, ele pode extrair os valores dos ajustes para os pinos e atualizar os pinos reais a partir dos parâmetros da solicitação.

O código para o sketch do segundo exemplo é o seguinte:

```
// sketch 10-02 Pinos de Internet

#include <SPI.h>
#include <Ethernet.h>
```

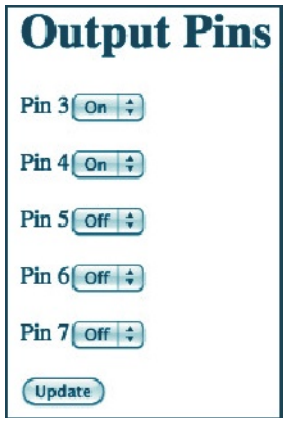

Figura 10-5 O formulário de envio de mensagem.

```
// O endereço MAC deve ser único. O seguinte deve funcionar:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
// O endereço de IP dependerá da sua rede local:
byte ip[] = { 192, 168, 1, 30 };
Server server(80);

int numPins = 5;

int pins[] = {3, 4, 5, 6, 7};
int pinState[] = {0, 0, 0, 0, 0};

char line1[100];

void setup()
{
   for (int i = 0; i < numPins; i++)
   {
    pinMode(pins[i], OUTPUT);
   }
   Serial.begin(9600);
  Ethernet.begin(mac, ip);
   server.begin();
}

void loop()
{
  Client client = server.available();
  if (client)
   {

   while (client.connected())
      {
         readHeader(client);
      if (! pageNameIs("/"))
         {
            client.stop();
            return;
         }
     client.println("HTTP/1.1 200 OK");
    client.println("Content-Type: text/html");
         client.println();

        // envie o corpo (body) do arquivo HTML
         client.println("<html><body>");
    client.println("<h1>Output Pins</h1>");
    client.println("<form method='GET'>");
         setValuesFromParams();
         setPinStates();

        for (int i = 0; i < numPins; i++)
         {
     writeHTMLforPin(client, i);
         }
     client.println("<input type='submit' value='Update'/>");
         client.println("</form>");
         client.println("</body></html>");

         client.stop();
     }
```

```
    }
}
void writeHTMLforPin(Client client, int i)
{
  client.print("<p>Pin ");
   client.print(pins[i
  client.print("<select name='");
   client.print(i);
   client.println("'>");
  client.print("<option value='0'");
   if (pinState[i] == 0)
   {
   client.print(" selected");
   }
   client.println(">Off</option>");
  client.print("<option value='1'");
   if (pinState[i] == 1)
   {
   client.print(" selected");
   }
   client.println(">On</option>");
   client.println("</select></p>");
}
void setPinStates()
{
   for (int i = 0; i < numPins; i++)
   {
   digitalWrite(pins[i], pinState[i
   }
}
void setValuesFromParams()
{
   for (int i = 0; i < numPins; i++)
   {
     pinState[i] = valueOfParam(i + '0');
   }
}
void readHeader(Client client)
{
   // leia primeira linha do cabeçalho
  char ch;
   int i = 0;
   while (ch != '\n')
   {
   if (client.available())
      {
      ch = client.read();
      line1[i] = ch;
     i ++;
      }
   }
   line1[i] = '\0';
```

```
   Serial.println(line1);
}
boolean pageNameIs(char* name)
{
   // o nome da página começa na posição 4 de caractere
   // e termina com espaço
   int i = 4;
   char ch = line1[i];

   while (ch != ' ' && ch != '\n' && ch != '?')
   {
     if (name[i-4] != line1[i])
      {
     return false;
      }
      i++;
    ch = line1[i];
   }
  return true;
}
int valueOfParam(char param)
{
   for (int i = 0; i < strlen(line1); i++)
   {
     if (line1[i] == param && line1[i+1] == '=')
      {
       return (line1[i+2] - '0');
      }
   }
  return 0;
}
```

O sketch usa dois arrays para controlar os pinos. O primeiro, **pins**, simplesmente especifica quais são os pinos usados. O array **pinState** contém o estado de cada pino: 0 ou 1.

Para obter a informação vinda do formulário do navegador especificando quais pinos devem estar ligados e quais devem estar desligados, é necessário ler o cabeçalho enviado pelo navegador. De fato, tudo o que você precisa está contido na primeira linha do cabeçalho. Você usará um array de caracteres **line1** para guardar a primeira linha do cabeçalho.

Quando o usuário está no navegador e clica no botão de Update (Atualizar) submetendo o formulário para ser enviado ao shield de Ethernet, a URL da página será algo como o seguinte:

```
http://192.168.1.30/?0=1&1=1&2=0&3=0&4=0
```

Os parâmetros da solicitação estão após o **?** e cada um está separado por um **&**. Examinando o primeiro parâmetro (**0=1**), isso significa que o primeiro pino do array (**pins[0]**) deve ter o valor 1. Se você fosse olhar a primeira linha do cabeçalho, você veria lá esses mesmos parâmetros:

```
GET /?0=1&1=1&2=0&3=0&4=0 HTTP/1.1
```

Na frente dos parâmetros, está o texto **GET /**. Isso especifica a página solicitada pelo navegador. Nesse caso,**/** indica a página raiz.

Dentro do loop do sketch, você chama a função **readHeader** para ler a primeira linha do cabeçalho. A seguir, você usa a função **pageNameIs** (o nome da página é) para verificar se a página solicitada é a página raiz **/**.

Agora, o sketch gera o cabeçalho e o início do formulário HTML que deverá ser exibido. Antes de escrever o código HTML para cada pino, o sketch chama a função **setValuesFromParams** para ajustar os valores a partir dos parâmetros da solicitação, lendo cada um dos parâmetros e colocando os valores adequados no array **pinStates** com os estados dos pinos.

A seguir, esse array é usado para definir os valores das saídas dos pinos antes que a função **writeHTMLforPin** seja chamada para cada um dos pinos escrevendo o código HTML de cada um. Essa função gera uma lista de seleção para cada pino. Essa lista precisa ser construída passo a passo. O comando **if** assegura que as opções apropriadas sejam selecionadas.

As funções **readHeader**, **pageNameIs** e **valueOfParam** são funções úteis de uso geral que você poderá utilizar em seus próprios sketches.

Você pode usar o seu multímetro como fez no Capítulo 6 para verificar que os pinos estão realmente sendo ligados e desligados. Se você quiser se aventurar um pouco, você ainda poderá conectar LEDs ou relés aos pinos para controlar coisas.

» *Conclusão*

Depois de usar shields e as respectivas bibliotecas nos dois últimos capítulos, chegou o momento de investigar os recursos que permitem escrever bibliotecas e de aprender a escrever as suas próprias bibliotecas.

capítulo 11

C++ e bibliotecas

Os Arduinos são simples microcontroladores. Na maior parte do tempo, os sketches de Arduino são bem pequenos, de modo que o uso da linguagem C de programação funciona muito bem. Entretanto, a linguagem de programação do Arduino é, na realidade, C++ em vez de C. A linguagem C++ é uma extensão da linguagem C de programação que contém algo denominado *orientação a objeto*.

Objetivos deste capítulo

- Aprender alguns fundamentos da linguagem C++ e de orientação a objeto
- Examinar mais de perto a biblioteca LCD
- Criar uma biblioteca simples para entender os conceitos que estão por trás

Orientação a objeto

Este é apenas um pequeno livro, de modo que uma explicação aprofundada da linguagem C++ de programação está além dos seus propósitos. No entanto, o livro cobrirá alguns fundamentos de C++ e de orientação a objeto, sabendo que o seu objetivo principal é aumentar o *encapsulamento* dos seus programas. O encapsulamento mantém juntas as coisas relevantes, algo que torna a linguagem C++ muito adequada para escrever bibliotecas como as que você usou nos sketches de Ethernet e de LCD em capítulos anteriores.

Há muito livros bons sobre C++ e a programação orientada a objeto. Procure os livros mais recomendados sobre esses tópicos na sua livraria virtual preferida.

Classes e métodos

A orientação a objeto usa um conceito denominado *classe* para auxiliar no encapsulamento. Geralmente, uma classe é como uma seção de um programa, incluindo variáveis – denominadas *variáveis membro*– e *métodos*, que são como funções mas que se aplicam às classes. Essas funções podem ser *public* (públicas), caso em que os métodos e funções podem ser usados por outras classes, ou *private* (privadas), caso em que os métodos podem ser chamados apenas por outros métodos dentro da mesma classe.

Embora um sketch esteja contido em apenas um arquivo, quando você está trabalhando com C++, a tendência é usar mais de um arquivo. De fato, há geralmente dois arquivos para cada classe: um arquivo de cabeçalho, que tem a extensão .h, e o *arquivo de implementação*, que tem a extensão .cpp.

Exemplo de biblioteca predefinida

Nos dois capítulos anteriores, usamos a biblioteca LCD. Agora, iremos examiná-la mais de perto e ver com mais detalhes o que está acontecendo.

Voltando ao sketch 9-01 (abra esse sketch no seu IDE do Arduino), você pode ver que o comando **include** contém o arquivo LiquidCristal.h:

```
#include <LiquidCrystal.h>
```

Esse arquivo é o arquivo de cabeçalho da classe denominada **LiquidCrystal**. Esse arquivo diz ao sketch do Arduino o que ele precisa para poder usar a biblioteca. Você poderá consultar esse arquivo se você for à pasta de instalação do seu Arduino e procurar o arquivo libraries/LiquidCrystal. Você deverá abrir o arquivo em um editor de texto. Se você está usando um Mac, então clique com o botão direito no próprio arquivo app do Arduino e selecione a opção de menu "Show Package Contents". A seguir, navegue até Contents/Resources/Java/libraries/LiquidCrystal.

O arquivo LiquidCrystal.h contém muitos códigos, já que se trata de uma biblioteca bem grande de classes. O código de fato da classe em si, onde estão os comandos realmente necessários para exibir uma mensagem, está no arquivo LiquidCrystal.cpp.

Na próxima seção, criaremos um exemplo simples de biblioteca, mostrando na prática os conceitos que estão por trás de uma biblioteca.

Escrevendo bibliotecas

Pode parecer que a criação de uma biblioteca de Arduino é o tipo de coisa que somente um experiente veterano de Arduino poderia tentar, mas na realidade é bem simples construir uma biblioteca. Por exemplo, você pode colocar em uma biblioteca a função **flash** do Capítulo 4, aquela que faz um LED piscar um número determinado de vezes.

Para criar os arquivos C++, necessários para conseguir isso, você precisará de um editor de texto no seu computador – algo como TextPad em Windows ou Text-Mate no Mac.

O arquivo de cabeçalho

Comece criando uma pasta que conterá todos os arquivos da biblioteca. Essa pasta deverá ser criada dentro da pasta denominada libraries (bibliotecas) que, por sua vez, está dentro da pasta de documentos do Arduino. No Windows, a sua pasta libraries estará em Meus Documentos\Arduino. No Mac, você irá encontrá-la na sua pasta inicial, Documents/Arduino/, e no Linux ela estará na pasta sketchbook da sua pasta inicial. Se a pasta libraries não estiver presente no seu Arduino, então crie uma.

A pasta libraries é onde devem ser instaladas quaisquer bibliotecas que você escrever, ou quaisquer outras bibliotecas "não oficiais."

Dê o nome Flasher a essa pasta. Abra o editor de texto e escreva o seguinte:

```
// LED Flashing library (biblioteca para LED pisca-pisca)

#include "WProgram.h"

class Flasher
{
   public:
     Flasher(int pin, int duration);
     void flash(int times);
   private:
    int _pin;
    int _d;
};
```

Salve esse arquivo na pasta Flasher com o nome Flasher.h. Esse é o arquivo de cabeçalho (header) da biblioteca. Esse arquivo especifica as diferentes partes da classe. Como você pode ver, ele está dividido em partes pública (public) e privada (private).

A parte pública contém o que parece com a parte inicial de duas funções. São os denominados métodos e diferem das funções somente porque estão associados a uma classe. Podem ser usados apenas como parte da classe. Diferentemente das funções, não podem ser usados isoladamente.

O primeiro método, **Flasher**, começa com uma letra maiúscula, algo que você não usaria com um nome de função. Além disso, ele tem também o mesmo nome da classe. Esse método é denominado um *construtor*, que você poderá aplicar quando quiser criar um novo objeto **Flasher** para ser usado em um sketch,

Por exemplo, você poderia colocar o seguinte em um sketch:

```
Flasher slowFlasher(13, 500);
```

Isso criaria um novo **Flasher** denominado **slowFlasher** que faria o pino D13 pulsar com um período de duração de 500 milissegundos.

O segundo método da classe é denominado **flash**. Esse método recebe um argumento simples com o número de vezes que deve piscar. Como está associado a uma classe, quando você quiser chamá-lo você deve fazer referência ao objeto que você criou antes, como segue:

```
slowFlasher.flash(10);
```

Isso faria o LED piscar 10 vezes, tendo o período que você especificou no construtor do objeto **Flasher**.

A seção privada da classe contém duas definições de variáveis: uma para o pino e uma para a duração, que é simplesmente denominada **d**. Sempre que você criar um objeto da classe **Flasher**, ele terá essas duas variáveis. Isso permite que ele lembre o pino e a duração quando um novo objeto **Flasher** é criado.

Essas variáveis são denominadas variáveis membro porque são membros da classe. Geralmente, os seus nomes não são comuns porque começam com um caractere de sublinhado. Entretanto, isso é apenas uma convenção comum, não é uma necessidade de programação. Uma outra convenção comumente usada para dar nome é o uso de um *m* minúsculo (de *membro*) como a primeira letra do nome de uma variável.

O arquivo de implementação

O arquivo de cabeçalho definiu de forma simples qual é a aparência de uma classe. Agora você precisa de um arquivo separado que fará o trabalho de fato. Será o arquivo de implementação e terá a extensão .cpp.

Assim, crie um novo arquivo contendo as linhas seguintes. Depois, salve-o como Flasher.cpp na pasta Flasher:

```
#include "WProgram.h"
#include "Flasher.h"

Flasher::Flasher(int pin, int duration)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
  _d = duration / 2;
}

void Flasher::flash(int times)
{
   for (int i = 0; i < times; i++)
   {
```

```
    digitalWrite(_pin, HIGH);
      delay(_d);
    digitalWrite(_pin, LOW);
      delay(_d);
    }
}
```

Há uma sintaxe que não é familiar nesse arquivo. Os nomes dos métodos têm o prefixo **Flasher::**. Isso indica que os métodos pertencem à classe **Flasher**.

O método construtor (**Flasher**) simplesmente atribui cada um de seus parâmetros à variável membro privada adequada. O parâmetro de duração **duration** é dividido por dois antes de ser atribuído à variável membro **_d**. Isso é assim porque o retardo (delay) é chamado duas vezes, sendo mais lógico que a duração total seja o tempo em que o LED permanece aceso mais o tempo em que está apagado.

A função **flash** é a que se encarrega realmente da tarefa de fazer o LED piscar. Ela repete o laço um número apropriado de vezes, mantendo o LED ligado e desligado durante intervalos adequados de tempo.

Completando a sua biblioteca

Agora você já viu todos os fundamentos necessários para completar a biblioteca. Você poderia disponibilizar essa biblioteca para ser utilizada, e ela trabalharia muito bem. Entretanto, há mais dois passos que você deve dar para completar a sua biblioteca. O primeiro é definir as palavras-chaves usadas na biblioteca, de modo que o IDE do Arduino possa exibi-las com a cor apropriada quando os usuários estiverem editando um código. O segundo é incluir alguns exemplos de como usar a biblioteca.

Palavras-chaves

Para definir as palavras-chaves (keywords), você deve criar um arquivo de nome keywords.txt, que é salvo na pasta Flasher. Esse arquivo contém apenas as duas linhas seguintes:

```
Flasher  KEYWORD1
flash    KEYWORD2
```

Isso é basicamente um arquivo de texto contendo uma tabela de duas colunas. A coluna da esquerda é a palavra-chave e a coluna da direita é uma indicação do tipo da palavra-chave. Os nomes para classes devem ser **KEYWORD1** e para métodos devem ser **KEYWORD2**. Não importa quantos espaços ou tabulações você insere entre as colunas, mas cada palavra-chave deve começar em uma nova linha.

Exemplos

Como um bom cidadão Arduino, a outra coisa que você deve incluir como parte da biblioteca é uma pasta de exemplos. Nesse caso, a biblioteca é tão simples que um único exemplo será suficiente.

Os exemplos devem ser todos colocados em uma pasta de nome Examples (exemplos) dentro da pasta Flasher. Na realidade, o exemplo é apenas um sketch de Arduino. Desse modo, você pode criar o seu exemplo usando o IDE do Arduino. Primeiro, no entanto, você deve sair e voltar a abrir o IDE do Arduino para que ele tome conhecimento da nova biblioteca.

Depois de voltar a abrir o IDE do Arduino, abra o menu selecionando File (arquivo) e então New (novo) para criar uma nova janela de sketch. A seguir, no menu, escolha Sketch e a opção Import Library (importar biblioteca). As opções devem ser parecidas com as da Figura 11-1.

No submenu, as bibliotecas acima da linha são oficiais e as que estão abaixo dessa linha são contribuições de bibliotecas "não oficiais". Se tudo estiver bem, entãovocê deverá ver Flasher na lista.

Se Flasher não estiver na lista, é muito provável que a pasta Flasher não está na pasta libraries da sua pasta de sketches. Portanto, volte atrás e faça uma verificação.

Na janela do sketch que acabou de ser criado, escreva o conteúdo seguinte:

```
#include <Flasher.h>

int ledPin = 13;
int slowDuration = 300;
int fastDuration = 100;
```

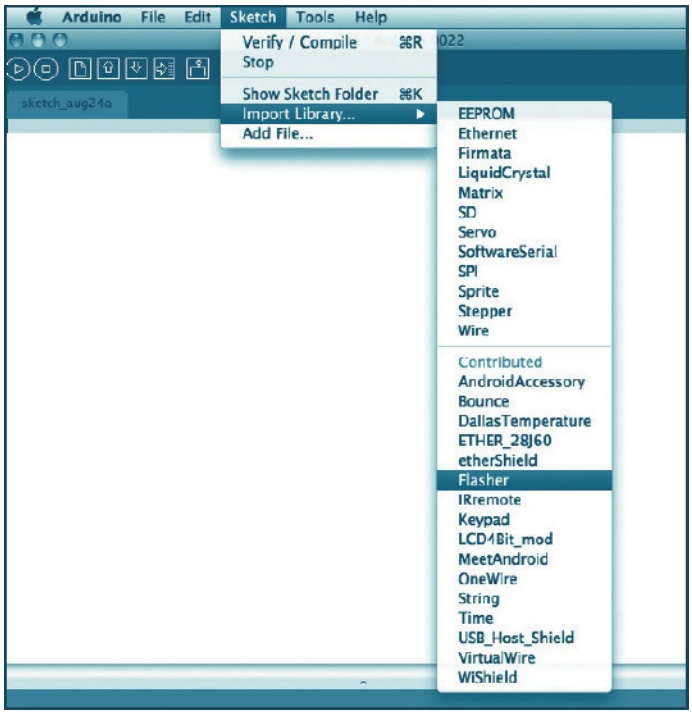

Figura 11-1 Importando a biblioteca Flasher.

```
Flasher slowFlasher(ledPin, slowDuration);
Flasher fastFlasher(ledPin, fastDuration);

void setup(){}

void loop()
{
   slowFlasher.flash(5);
   delay(1000);
   fastFlasher.flash(10);
   delay(2000);
}
```

O IDE do Arduino não permitirá que você salve o exemplo de sketch diretamente na pasta libraries, de modo que você deve salvá-lo em algum outro lugar com o nome Simple Flasher Example (exemplo de flasher simples) e então mover toda a pasta Simple Flasher Example que você acabou de salvar para dentro da pasta de Examples na sua biblioteca.

Se você voltar a abrir o seu IDE do Arduino, você verá que agora é possível abrir o sketch com o exemplo a partir do menu, como está mostrado na Figura 11-2.

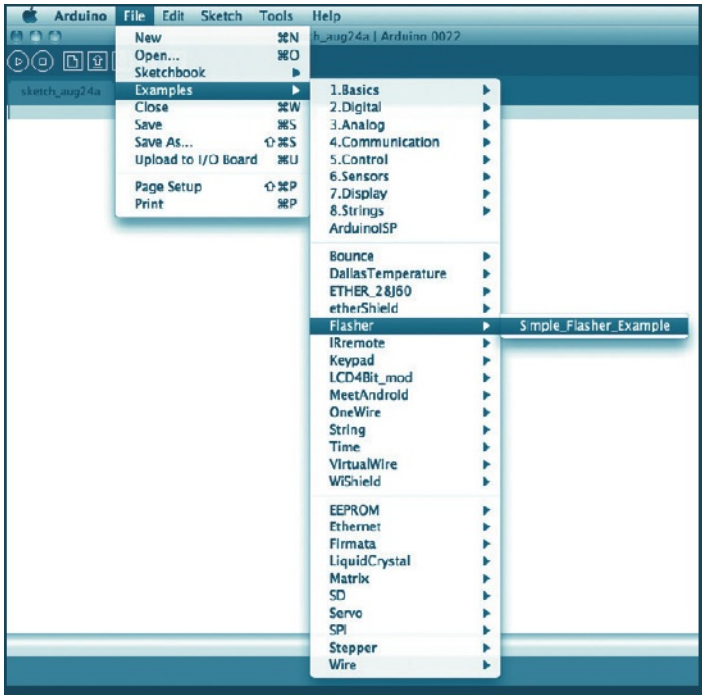

Figura 11-2 Abrindo o sketch com o exemplo.

Conclusão

Há mais coisas a respeito de C++ e sobre como escrever programas para formar bibliotecas, mas este capítulo deve lhe dar as condições para poder começar, o que deve ser suficiente para a maioria das coisas que você provavelmente fará com o Arduino. Os Arduinos são dispositivos pequenos, e frequentemente surge a tentação de exagerar nas soluções que, de outro modo, poderiam ser bem simples e imediatas.

Com este capítulo, concluímos a parte principal do livro. Para maiores informações sobre para aonde ir agora, um bom ponto de partida é sempre o site oficial do Arduino em www.arduino.cc. Também, recorra por favor ao site do livro em www.arduinobook.com, onde você encontrará a errata e outros recursos úteis.

Se você estiver procurando por auxílio ou orientação, a comunidade do Arduino em www.arduino.cc/forum é de grande ajuda. Lá você encontrará também o autor deste livro com o nome de usuário Si.

Índice

Símbolos

>= (maior do que ou igual a), operador de comparação, 41
<= (menor do que ou igual a), operador de comparação, 41
– (subtração), operador, 39–40
!= (não igual a, diferente de), operador de comparação, 41
/ (barra) como operador de divisão, 39–40
; (ponto e vírgula), na sintaxe de programação, 29
|| (or, ou), operador, 54–55
+ (soma) como operador de adição, 39–40
= (atribuição), operador de atribuição de valores às variáveis, 37–38
== (igual a), operador de comparação, 41, 54–55
&& (and, e), operador de manipulação de valores lógicos, 54–55
* (asterisco) como operador de multiplicação, 39–40
[] (colchetes), na sintaxe de arrays, 62
<< (operador de deslocamento de bit), 108–109
< (menor do que), operador de comparação, 41
> (maior do que), operador de comparação, 41

A

abs, função, funções matemáticas da biblioteca, 97–98
adição (+), operador, 39–40
Algoritmos + Estruturas de Dados = Programas (Wirth), 61–62
alimentação elétrica das placas de Arduino, 17–19
analogRead, função, 91–92
analogWrite, função, 91–92
and (&&), operador, manipulação de valores lógicos, 54–55
Arduino, srcens do, 10–11
Arduino Bluetooth, 14–15
Arduino Diecimila, 11–13
Arduino Duemilanove, 11–13
Arduino Lilypad, 14–15
Arduino Mega, 12–13
Arduino Nano, 12–15
Arduino Uno
  na família Arduino de placas de desenvolvimento, 11–13
  processador ATMega328 no, 107
argumentos
  da função tone (som), 100–101
  de funções aleatórias, 96–97
  passagem para as funções, 28–29
aritmética
  operadores, 39–40
  variáveis numéricas e, 37–40
armazenamento de dados
  armazenando floats em EEPROM, 108–110
  armazenando ints em EEPROM, 108–109
  armazenando strings em EEPROM, 109–110
  compressão e, 110–112
  constantes, 105–107
  diretiva PROGMEM, 106–107
  EEPROM, 107–108
  limpando os conteúdos da EEPROM, 110–111
  visão geral de, 105–106
arquivo de implementação (.cpp)
  criando para biblioteca, 134–135
  necessário para cada classe C++, 132–133
arquivos de cabeçalho (.h)
  criando para biblioteca, 133–134
  necessários para cada classe C++, 132–133

arrays
  arrays de string. *Veja* strings
  exemplo de sinal de SOS, 64–66
  para tradução de código Morse. *Veja* tradutor de código Morse
  PROGMEM e, 106–107
  visão geral de, 61–65
ASCII, código, 66–67
asterisco (*) como operador de multiplicação, 39–40
ATmega1280, Mega e, 12–13
ATmega168, Uno e, 11–13
ATmega328
  Uno e, 11–13, 107
  uso em placas de Arduino, 7–8
atribuição (=), operador, atribuição de valores a variáveis, 37–38
autoscroll, função da biblioteca LCD, 117–118

B

barra (/) como operador de divisão, 39–40

bibliotecas, Arduino
  biblioteca PROGMEM, 106–107
  funções de manipulação de bit das, 97–100
  funções matemáticas, 97–98
  interrupções, 100–103
  números aleatórios, 95–98
  shift–Out, função, 100–101
  tone, função, 100–101
  visão geral de, 95–96
bibliotecas C++
  criando arquivo de cabeçalho para biblioteca, 133–134
  criando arquivo de implementação para biblioteca, 134–135
  criando exemplos para biblioteca, 135–137
  definindo palavras–chaves para, 135–136
  escrevendo, 132–133
  exemplo usando biblioteca predefinida, 132–133
bitRead, função, 99–100
bits, 98–99
bitWrite, função, 99–100
blink, função da biblioteca LCD, 117–118
Blink, sketch
  default, 18–21
  execução, 34–35
  funções de setup e loop no, 33–34
  LEDs e, 17–18
  modificando, 20–23
blocos de código, 33–34
Bluetooth, 14–15
byte, tipos de dados, 55–56

C

chamando funções, 28–29
CHANGE, constante, tipos de sinais de interrupção, 102–103
char
  literal string e, 65–66
  PROGMEM e, 106–107
  tipos de dados da linguagem C, 55–56
chave de botão, debouncing, 85
chaves
  conectando ao pino de entrada da placa, 82–84
  resistores de pull–up e, 84
  usando fio como, 86–88
chaves ({})
  estilos ou normas de formatação, 57–58
  na sintaxe dos blocos de código, 33–34
chip controlador para ser usado no shield de LCD, 113–114
classes, C++, 131–133
clear, função da biblioteca LCD, 115
código
  blocos de, 33–34
  compilação de, 29–32
  estilos ou normas de formatação, 55–57
  transferência para placa, 29–30
código Morse
  exemplo de SOS usando arrays, 64–66
  história do, 62
colchetes ([]) na sintaxe de array, 62

COM3 como porta serial em computadores com Windows, 19–21
comandos da linguagem C
  comando for, 41–44
  comando if, 39–41
  comando while, 44
  visão geral dos, 39–40
comentários
  estilos ou normas de formatação, 58–59
  razões para usá–los ou não, 59–59

compressão, armazenamento de dados e, 110–112
compressão de faixa, 110–112
computação física, Arduino descrito como, 5–6
computadores Mac
  criando bibliotecas C++ e, 133
  TextMate como editor de texto, 133
condições, o comando if e, 40
Conector Serial de Programação, 10
conexões da alimentação elétrica, 7–9
conexões digitais nas placas de Arduino, 7–9
constantes
  armazenamento, 105–107
  definição, 50–51
constrain, funções matemáticas da biblioteca, 97–98
constructor, métodos, 134–135
convenções para dar nome às variáveis, 134
cos, funções matemáticas da biblioteca, 97–98
Creative Commons, licença para projetos com Arduino, 10–11
cursor, função da biblioteca LCD, 117–118

D

dados, representação de números e letras, 67–69
dados, tipos da linguagem C, 55–57
debouncing, quando se aperta um botão, 85–91
#define, diretiva
  para associar um valor a um nome, 44–45
  para definir constantes, 50–51
delayPeriod, variável
  condições usadas com a, 40
  exemplo de uso da, 35–36
DHCP (Dynamic Host Configuration Protocol), 122
Diecimila, 11–13
digitalRead, função, 80–82
digitalWrite, função, 33–34, 84
DIL (dual inline) soquete, 7–9
diodos emissores de luz (LEDs), 17–19
display, função da biblioteca LCD, 117–118
displays LCD
  enviando mensagens para, 116–117
  funções que podem ser usadas com, 117–118
  placa USB de mensagens, 115–117
  visão geral de, 113–114
double, tipos de dados da linguagem C, 55–56
dual inline (DIL), soquete, 7–9
Duemilanove, 11–13
Dynamic Host Configuration Protocol (DHCP), 122

E

E/S. *Veja* entrada/saída (E/S)
editores de texto para criar arquivos C++, 133
EEPROM (electrically erasable read–only memory)
  compressão de dados e, 110–111
  floats armazenados em, 108–110
  ints armazenados em, 108–109
  lendo de/escrevendo em, 107–108
  limpando os conteúdos da, 110–111
  strings armazenados em, 109–110
elemento de arrays, 63–64
ENC28J60, chip controlador de Ethernet, 120–121
encapsulamento
  aumentando a orientação a objetos, 131–132
  variáveis globais, 50–51
endereços de mac, usando o Arduino como servidor de web e, 122
endereços IP, uso do Arduino como servidores de web e, 122–124
entrada/saída (E/S)
  aperto de botão e debouncing, 85–91
  entradas analógicas, 7–9, 91–93
  entradas digitais, 80–82
  funções avançadas de, 100–103
  pinos do microcontrolador, 5–7
  resistores de pull–up e, 80–84
  resistores internos de pull–up, 84–85
  saídas analógicas, 90–92
  saídas digitais, 77–81
  visão geral de, 77–78
entradas digitais
  aperto de botão e debouncing, 85–91
  resistores de pull–up e, 80–84
  resistores internos de pull–up, 84–85
  visão geral das, 80–82
EPROM (erasable programmable read–only memory), 5–6
espaços em branco, estilos ou normas de formatação de código, 58

Ethernet
comunicação com servidores de web, 120–121
enviando o ajuste de pinos através de uma rede, 124–129
escolhendo um shield oficial baseado no Wiznet, 120–121
shields de Arduino, 10–12, 119–120
usando o Arduino como servidor de web, 122–125
exemplos, criando para a biblioteca C++, 135–137

F

FALLING, constante, tipos de sinais de interrupção, 102–103
Femtoduino, placas não oficiais de Arduino, 15–16
flashDotOrDash, função, 72–73
flashSequence, função, 71–72
float
armazenando em EEPROM, 108–110
compressão de faixa e, 110–111
tipos de dados da linguagem C, 55–56
visão geral de, 53–54
fonte de alimentação
na placa Arduino, 7–9
requerida pelo shield de Ethernet, 120–121
for, comando, 41–44
Freeduino, placas não oficiais de Arduino, 15–16
funções
chamando e passando argumentos para, 28–29
coleção de. *Veja* bibliotecas, Arduino
comentários, 58–59
definidas *versus* predefinidas, 47–48
espaços, 58
estilos ou normas de formatação, 55–57
examinando em código padronizado, 31–34
floats, 53–54
funções de setup e loop no sketch Blink (piscapisca), 33–34
para uso com displays LCD, 117–118
parâmetros, 49–50
predefinidas *versus* definidas, 32–33
recuo, 55–58
sinal de chave, 57–58
tipos de dados da linguagem C, 55–57
tipos de variáveis, 52–53
valores retornados, 52–53
variáveis booleanas e, 53–56
variáveis globais, locais e estáticas, 49–52
visão geral, 47–49
funções de manipulação de bit, 97–100
funções matemáticas da biblioteca Arduino, 97–98

G

gerador de número aleatório, 97–98

H

HD44780, chip controlador usado com o shield de LCD, 113–114
highByte, função, 108
home, função da biblioteca LCD, 117–118
HTML (HyperText Markup Language), 120–121, 124–125, 128–129
HTTP (HyperText Transport Protocol), 120–121, 124–125

I

if, comando, 39–41, 54–55
igual a (==), operador de comparação, 41, 54–55
int
16 bits usados em representação numérica, 55–56
armazenando em EEPROM, 108–109
compressão de faixa e, 110–111
declarando, 49–50
função random retornando ints, 96–97
tipos de dados da linguagem C, 55–56
valor decimal de, 98–99
valor hex de, 99–100
valores retornados e, 52–53
interferência, resistores de pull–up e, 80–82
interrupções, 100–103

L

LEDs (light–emitting diodes), 17–19
letras, representação de dados no tradutor de código Morse, 67–69
Lilypad, 14–15

linguagem binária, compilação de código para, 29–30
linguagem C
  C++ como extensão da, 131–132
  comando for, 41–44
  comando if, 39–41
  comando while, 44
  comandos, 39–40
  #define, diretiva, 44–45
  estilos de codificação, 55–57
  examinando funções em código padronizado, 31–34
  execução do sketch Blink, 34–35
  funções de setup e loop no sketch Blink, 33–34
  processo de compilação, 29–32
  testando experimentos em, 36–38
  tipos de dados da, 55–57
  variáveis, 34–36
  variáveis numéricas e processos aritméticos em, 37–40
  zero como índice na, 62
linguagem C++
  classes e métodos, 131–133
  criando arquivo de cabeçalho para biblioteca, 133–134
  criando arquivo de implementação para biblioteca, 134–135
  criando exemplos de biblioteca, 135–137
  definindo palavras–chaves para biblioteca, 135–136
  escrevendo bibliotecas, 132–133
  exemplo usando biblioteca predefinida, 132–133
  orientação a objeto, 131–132
linguagens de programação
  linguagem C. *Veja* linguagem C
  linguagem C++. *Veja* linguagem C++
  sintaxe das, 28–29
  transferindo código para a placa, 29–30
  vocabulário das, 27–28
literal string, 65–69
log, funções matemáticas da biblioteca, 97–98
long, tipos de dados da linguagem C, 55–56
loop, função
  comando if e, 40–41
  definido, 32–33
  laços de for, 41–44
  laços de while, 44
  no sketch Blink de exemplo, 33–34
  para o tradutor de código Morse, 68–72
  para shield de Ethernet, 124–125
  para shield de LCD, 116–117
  requerida em todos sketches, 30–31
lowByte, função, 108
LSBFIRST, constante, 100–101

M

maior do que (>), operador de comparação, 41
maior do que ou igual a (>=), operador de comparação, 41
map, funções matemáticas da biblioteca, 98
max, funções matemáticas da biblioteca, 98
Mega, 12–13
memória
  dados na memória do computador, 63–64
  memória flash, l, 106–107
memória flash
  armazenando dados em, 106–107
  em microcontrolador, 5–6
menor do que (<), operador de comparação, 41
menor do que ou igual a (<=), operador de comparação, 41
menu File, acessando o Sketchbook a partir de, 23–24
métodos, C++, 131–133
métodos privados, C++, 132–133
métodos públicos, C++, 132–133
microcontroladores
  na placa de desenvolvimento do Arduino, 6–8
  velocidade de processamento dos, 34
  visão geral dos, 5–7
min, funções matemáticas da biblioteca, 97–98
Modulação por Largura de Pulso (PWM), 90–92
MSBFIRST, constante, 100–101
multímetro, medindo saídas com, 77–79, 90–92

N

Nano, 12–15
não igual a, diferente de (!=), operador de comparação, 41
navegadores de web
  comunicação via HTTP, 120–121
  formatação de texto usando HTML, 120–121

noAutoscroll, função da biblioteca LCD, 117–118
noBlink, função da biblioteca LCD, 117–118
noCursor, função da biblioteca LCD, 117–118
noDisplay, função da biblioteca LCD, 117–118
noInterrupts, função, 102–103
números, representação de dados no tradutor de código Morse, 67–69
números de ponto flutuante. *Veja* float
números pseudoaleatórios, 96–97

O

objetos, criando, 134
operador (/) de divisão, 39–40
operador de deslocamento de bit (<<), 108–109
operador de multiplicação (*), 39–40
operadores, 53–56
  aritméticos, 39–40
  de atribuição (=), 37–38
  de comparação, 41
  de deslocamento de bit(<<), 108–109
operadores de comparação
  comparação de valores usando o operador igual (==), 54–55
or (| |), operador, 54–55
orientação a objetos em C++, 131–132
oscilador de cristal de quartzo em placas de Arduino, 10–11

P

pageNameIs, função, 128–129
palavras–chaves, definindo para a biblioteca C++, 135–136
ParaFazerDepois, comentários em um código, 59–59
parâmetros
  acrescentando às funções, 49–50
  sintaxe das linguagens de programação, 28–29
  variáveis globais e, 49–51
PCB (printed circuit board), 7–8
pgm_read_word, função, 106–107
pinMode, função
  chamada pela função setup, 33–34
  para configurar a eletrônica dos pinos, 78–80
pinos
  atribuições de pinos para o shield de LCD, 116
  controlando um display LCD, 113–114
  enviando o ajuste de pinos através de uma rede, 124–129
  função shift–Out para alimentar um registrador deslocador, 100–101
  interrupções associadas a, 102
  pinos de entrada/saída em um microcontrolador, 5–7
pinState, array, 128–129
placa Chipkit, 15–16
placa de circuito impresso (PCB), 7–8
placas de desenvolvimento
  Bluetooth, 14–15
  componentes das, 7–8
  conexões analógicas e digitais, 7–9
  conexões da alimentação elétrica, 7–9
  cristal, chave de Reset, Conector de Comunicação Serial e conexão USB, 10–11
  da família Arduino, 11–12
  escolhendo o tipo de, 19–21
  fonte de alimentação, 7–9
  Lilypad, 14–16
  Mega, 12–13
  microcontroladores das, 9–10
  Nano, 12–15
  Uno, Duemilanove e Diecimila, 11–13
  visão geral das, 6–8
pontuação, sintaxe das linguagens de programação, 28–29
porta serial, selecionando a partir do menu Tool, 19–22
posting data e HTTP, 124–125
pow, funções matemáticas da biblioteca, 97–98
print, função da biblioteca LCD, 115
Processing, biblioteca, 95–96
processo de compilação
  arrays e, 63–64
  tradução do programa para o código de máquina, 29–32
PROGMEM, diretiva
  armazenamento de dados em memória flash, 106–107
  compressão de dados e, 110–111
programação por intenção, 69–70

programas. *Veja* sketches
pull–up, resistores de. *Veja* resistores de pull–up
PWM (Modulação por Largura de Pulso), 90–92

R

RAM (random access memory), 5–6
random, função, 95–98
randomSeed, função, 97
readHeader, função, 128–129
recuo, estilos ou normas de, 55–58
registradores deslocadores, 100–101
relés, shields de Arduino, 10–11
Reset, botão da placa do Arduino, 10
Reset, conector, conexões de alimentação elétrica, 7–9
resistores de pull–up
  habilitação de resistores internos, 84–85
  simulação de interrupção usando, 102
  visão geral de, 80–84
return, funções e valores de retorno, 52–53
RISING, constante, tipos de sinais de interrupção, 102–103
Roboduino, placas não oficiais de Arduino, 15–16
Ruggeduino, placas não oficiais de Arduino, 15–16

S

saída. *Veja* entrada/saída (E/S)
saídas analógicas, 90–92
saídas digitais
  função pinMode para configuração eletrônica dos pinos, 78–80
  medição com multímetro, 77–81
  visão geral das, 77–78
scrollDisplayLeft, função da biblioteca LCD, 117–118
scrollDisplayRight, função da biblioteca LCD, 117–118
Seeeduino, placas não oficiais de Arduino, 15–16
sequências de caracteres. *Veja* strings
Serial.available(), 69–70
Serial Monitor
  exemplo usando aritmética e variáveis numéricas, 37–39
  exibindo mensagens enviadas pelo, 115
  lendo interferência elétrica, 80–84
  testando experimentos em C, 36–13
  testando o tradutor de código Morse, 72–75
  vendo array no, 62–63
servidores de web
  comunicação com, 120–121
  usando Arduino como, 122–125
setCursor, função da biblioteca LCD, 115
setup, função
  definindo, 32–33
  no exemplo de sketch Blink, 33–34
  no shield de Ethernet, 124–125
  no shield de LCD, 116–117
  no tradutor de código Morse, 68–69
  requerida em todos sketches, 30–31
setValuesFromParams, função, 128–129
shield, placas
  de display LCD, 113–114
  de Ethernet, 119–120
  escolhendo um shield oficial baseado em Wiznet, 120–121
  lista de shields populares, 10–11
Shield de LCD e Keypad da DFRRobot, 113–114
shield de Motor, 10–11
Shield LCD alfanumérico, 114
shift–Out, função, 100–101
sin, funções matemáticas da biblioteca, 97–98
sintaxe das linguagens de programação, 28–29
Sketchbook, 23–25
sketches
  acessando a partir do menu File, 23–24
  baixando, 24–25
  código padronizado para, 30–31
  coleção de funções para. *Veja* bibliotecas, Arduino
  como programas, 28–29
  instalando primeiro, 18–23
software, instalação, 18–19
soma (+) como operador de adição, 39–40
sqrt, funções matemáticas da biblioteca, 97–98
strings
  armazenando em EEPROM, 109–110
  literal string, 65–67
  para tradução de código Morse. *Veja* tradutor de código Morse
  variável string, 66–67
  visão geral de, 65–66
subtração (–), operador, 39–40

T

tabelas–verdade usando valores lógicos, 54–55
tags, em HTML, 121, 124–125
tags de cabeçalho, HTML, 121,124–125
tags de parágrafo, HTML, 121, 124–125
tags do corpo de um arquivo HTML, 124–125
tan, funções matemáticas da biblioteca, 97–98
tarefas, execução de uma única, 100–102
Teensy, placas não oficiais de Arduino, 15–16
tempFloat, variável, 111
TextMate (computadores Mac), 133
texto, formatação em HTML, 120–121
TextPad (computadores Windows), 133
tipo de letra, sintaxe das linguagens de programação e, 28–29
tone (som), função, 100–101
Tools, menu, 19–22
tradutor de código Morse
  armazenando constantes no, 105–107
  função flashDotOrDash, 72–73
  função flashSequence, 71–72
  função loop para o, 68–72
  representação de dados, 67–69
  testando no Serial Monitor, 72–75
  variáveis globais e função setup para o, 68–69
  visão geral, 66–68
type cast, convertendo float para int, 111

U

UART (Universal Asynchronous Receiver/Transmitter), 70
unions, armazenando floats em EEPROM, 108–110
Uno
  na família Arduino de placas de desenvolvimento, 11–13
  processador ATMega328 do, 107
unsigned int, tipos de dados da linguagem C, 55–56
unsigned long, tipos de dados da linguagem C, 55–56
USB
  componente das placas Arduino de desenvolvimento, 10–11
  comunicação via, 69–70
  conectando a placa Arduino à porta USB, 17–18
  conectando via, 5–6
  instalando drivers para, 18–19
USB, placa de mensagens, 115–117
USB Host, shield, 10–11

V

valor decimal e equivalente hexadecimal e binário, 98–100
valores
  armazenando. *Veja* armazenamento de dados
  atribuindo às variáveis, 37–38
  diretiva #define para associação com um nome, 44–45
  funções que retornam valores, 52–53
  lista de valores em arrays, 61–62
  manipulando valores, 53–56
valores binários, equivalentes hexadecimal e decimal, 98–100
valores hexadecimais, equivalentes decimal e binário, 98–100
valueOfParam, função, 128–129
variáveis
  booleanas e, 53–56
  definindo na linguagem C, 34–36
  estáticas, 51–52
  float, 53–54
  globais, 49–51, 68–69
  locais, 50–52
  numéricas, 37–40
  para objeto Flasher, 134
  tipos de, 52–53
  tipos de dados da linguagem C, 55–57
  variáveis membro em C++, 131–132
variáveis de buffer, ao usar char array com PROGMEM, 107

variáveis membro, C++, 131–132, 134
variável booleana
  manipulação de valores lógicos, 53–56
  tipos de dados da linguagem C, 55–56
variável ledPin, 35–36
variável string, 66–67
Verify, botão para verificar o código, 29–30
vocabulário das linguagens de programação, 27–28
void, palavra–chave, 31–33

W

while, comando, 44
Windows, computadores
  criando bibliotecas C++, 133
  porta serial para, 19–21
  TextPad como editor de texto, 133
Wiring, biblioteca do Arduino baseada em, 95–96
Wiznet, chipset, 120–121
write, função da biblioteca LCD, 116–117
writeHTMLforPin, função, 128–129