MAKER INNOVATIONS SERIES

Arduino Software Internals

A Complete Guide to How Your Arduino Language and Hardware Work Together

Second Edition

Norman Dunbar

Apress®

Maker Innovations Series

Norman Dunbar

Arduino Software Internals

A Complete Guide to How Your Arduino Language and Hardware Work Together

Second Edition

Apress®

Norman Dunbar
Rawdon
West Yorkshire, UK

ISBN-13 (pbk): 979-8-8688-0170-9 ISBN-13 (electronic): 979-8-8688-0171-6
https://doi.org/10.1007/979-8-8688-0171-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Miriam Haidara
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover designed by eStudioCalamar

This book is dedicated to my wife, Alison, who occasionally allows me to have some time to myself, programming, attempting to build things *(with or without "*Internet of*"), and writing notes, articles, and/or this book.*
Another person to whom I am grateful is Alison's late maternal grandmother, Minnie Trees (yes, I did call her Bonsai!), who gifted me an Arduino Duemilanove starter kit and rekindled my long-lost (for over 35 years) interest in building things with electronics.
The book is also dedicated to the myriad of people and companies or organizations around the world who freely give their time and skills to produce open source software and hardware, for the benefit of others or just for fun.
If I may paraphrase the words of Isaac Newton*, I too stand on the shoulders of giants, so here's to the giants, the little people, and all the medium-sized ones too, who may or may not become giants themselves. Let's hope the fun never stops.*
Finally, my own motto is Don't think! Find out! *Hopefully, this book will help you do exactly that.*

Contents

About the Author

Norman Dunbar is a retired Oracle database administrator who lives with his wife, Alison, and a cockapoo dog, Wesley, to keep him out of trouble.

Norman has had a long-running relationship with electronics since childhood and computers since the late 1970s, and the Arduino was a perfect marriage of the two interests.

With a love of learning new things, examining and explaining the Arduino Language and the hardware became a bit of a hobby, and as his piles of notes expanded, Apress decided to publish his work as *Arduino Software Internals*.

Since then, Norman has been diving into the slightly trickier aspects of the Arduino—interrupts—with a view to documenting them for his own ease of use. Once more, his notes became a book—*Arduino Interrupts*—published by Apress in December 2023.

Because he never remembers exactly how much work is involved and how hard it is to write a technical book, Norman is now writing a third book about the Arduino, with a view to completing his trilogy.

Norman's motto continues to be *don't think, find out*.

About the Technical Reviewer

Sai Yamanoor is an embedded engineer based in Oakland, CA. He has over ten years of experience as an embedded systems expert, working on hardware and software design. He is a coauthor of three books on using Raspberry Pi to execute DIY projects, and he has also presented a Personal Health Dashboard at Maker Faires across the country. Sai is also working on projects to improve the quality of life (QoL) for people with chronic health conditions. Check out his projects at https://saiyamanoor.com.

Acknowledgments

I would like to thank everyone involved in the production of this book—the people you almost never hear about. Without them, there would be no book.

I'm grateful to the following people at Apress/Springer:

Miriam Haidara who convinced me to update *Arduino Software Internals* to cover changes for the new version of the IDE and software. Jessica Vakili who has had the misfortune to have worked with me on three books now, thanks! Also, Joseph Quatela, James Markham, and Nirmal Selvaraj who kept answering my silly questions!

The people who turned my manuscript into a proper book, who did the indexing for me, the people running the print machines, and the cover designers. You never get named or mentioned, but authors do appreciate you—thanks.

Finally, to my wife Alison and Wesley the cockapoo, thanks for letting me write an update and keeping me exercised!

Preface

If I have seen further it is by standing on ye sholders of Giants.

Sir Isaac Newton (1643–1727), in a letter to Robert Hooke, 15 February 1676

There are many books which discuss the abilities of the Arduino *hardware* and how best the maker can use this to their benefit. I have many of them in my bookcase and digital versions on my phone and tablet—in case I get bored with life and need something interesting to read. Many of these books explain what the hardware does, and some even dig deeper into the hardware to explain how, in fairly easy-to-understand terms, it does it.

There are no books which take a similar view of the Arduino *software*. There is now!

This book takes you on a *journey* (why do we *always* have to be on a journey these days?) into the world of Arduino sketches and the various files involved in the compilation process. It will delve deep into the supplied software and look at the specific parts of the Arduino Language which deal with the underlying hardware, the ATmega328P (or ATmega328AU) microcontrollers—henceforth, referred to as ATmega328P.

Once the Arduino Language has been explained, the book takes a short look at how you can strip away the Arduino *hand-holding* and get down and dirty with the naked hardware. It's not easy, but equally it's not too difficult. Don't worry; this is still the C/C++ language; there's no assembly language required. Perhaps!

Rawdon, UK   Norman Dunbar

Preface to the Second Edition

Since the first edition of *Arduino Software Internals* was published in 2020, the Arduino environment has moved onward with new microcontroller boards being added, numerous bugs being fixed, new bugs introduced—albeit, not deliberately—and many improvements made.

One of the bigger changes has been to the IDE itself; it is now at release 2.1.0 and has changed completely from the old 1.x releases. It now provides a much more modern experience with code completion, IntelliSense, and much, much more. It still has drawbacks—when you open a new sketch, you get a new IDE—but progress has indeed been made in lots of areas.

Another big change is the Arduino Command Line Interface. It has moved on from version 0.6 to version 0.30, and it has become a very usable tool. A couple of major improvements that come immediately to mind are the ability to upload code with an ICSP device and the ability to burn bootloaders. It has improved so much that the Arduino IDE has replaced the old preprocessing and compilation subsystems with the "arduino-cli" under the covers. Sadly, it still cannot upload EEPROM data.

PlatformIO, another alternative to the Arduino IDE, has itself improved and now, at the time of writing—May 2023—supports over 1,500 boards, 50 platforms, and 24 different frameworks, not to mention over 13,400 libraries!

The standard IDE for use with PlatformIO is *Visual Studio Code* (VSCode) rather than Atom or the command line, although those are still available. Don't worry if you don't like or use VSCode; PlatformIO Core—the command-line option—can still generate project files for an even larger number of common IDEs such as *Atom*, *CLion*, *Code::Blocks*, *Eclipse*, *Emacs*, *NetBeans*, *Qt Creator*, *Sublime Text*, *Vim*, *Visual Studio*, and *VSCode*.

The first edition of this book occasionally mentioned Windows, and at that time, I had limited access to Windows 7. The current version, as of May 2023, is Windows 11, but unfortunately, I no longer have access to any versions of Windows.

I hope you find the second edition as useful as, if not more than, the first edition.

Introduction 1

The Arduino is a great system for getting people into making with electronics and microcontrollers. I was reintroduced to a long-lost hobby when I was gifted an Arduino Duemilanove (a.k.a. 2009) by my wife's late grandmother, and since then, I've had lots of fun learning and *attempting* to build *things*. I've even built a number of Arduino clones based on just AVR microcontrollers and a few passive components—it's cheaper than fitting a new Arduino into a project!

Much has changed over the intervening years; LEDs used to cost about £10 each and came in one color, red. These days, I can get a pack of 100 LEDs for about £2 in various different colors. Even better, my old faithful Antex 15W soldering iron still worked, even after 35 years. Sadly, after the first edition was published, it finally died. I bought another one, exactly the same!

The Arduino—and I'm concentrating on either the Uno version 3 or the Duemilanove here as those are two which I've actually purchased (or been given)—is based around an Atmel ATmega328 microcontroller. On the Uno, it's the ATmega328PAU, while the Duemilanove uses the ATmega328PPU.

Roughly, the only difference between the two is the Uno's ATmega328PAU version is a surface mount, while the ATmega328PPU version is a 28-pin through-hole device. They are, or should be, identical to program, although the ATmega328PAU version does have two additional analog pins that are not present on the ATmega328PPU.

Occasionally though, I may mention in passing the Mega 2560 R3—as I have a cheap Chinese clone of one of these—which is based on the Atmel ATmega2560 microcontroller.

Some older Arduino boards had the ATmega168 microcontroller, which also was a 28-pin through-hole version, but it only had 16 Kb of flash memory as opposed to the 32 Kb in the later 328 chips. The EEPROM and RAM size is also half that of the ATmega328P devices.

The Arduino was designed for ease of use, and to this end, the software and the "Arduino Language" hides an awful lot from the maker and developer. Hopefully, by the time you have finished reading this book, you will understand more about what it does and why and, when necessary, how you can bypass the Arduino Language (it's just C or C++ after all) and use the bare-metal AVR-specific C or C++ code instead. Doing this can lead to more space for your code, faster execution, and lower power requirements—some projects can be run for months on a couple of batteries.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_1

1.1 Arduino Installation Paths

The version of the Arduino IDE described in this book is 2.1.0. The version of the underlying Arduino Language is 1.8.6.

I used an installation on Linux Mint while writing this book as Linux is my operating system of choice, plus I do not have access to Windows anymore. The IDE was installed by downloading the zip file version, as opposed to the flatpak version, and extracted. The location I extracted into is

- `/home/norman/arduino-2.1.0/arduino-ide`

On first execution, the IDE will create two new hidden directories—if they don't already exist:

- `/home/norman/.arduino15` which will contain the appropriate Arduino Language files for your chosen board(s)
- `/home/norman/.arduinoIDE` which holds installation log files and is now where the IDE preferences are stored

When I compiled a sketch for my Uno board, I was prompted to install the AVR package required by my Uno. I agreed, and AVR package version 1.8.6 was installed into `/home/norman/.arduino15`, which just happens to be the same location used by the 1.x version of the IDE.

The range of preferences offered by the new IDE is not as large as previous versions. Well, it seems that that is the case, but all is not as it seems. This shall be explained soon!

Within this book, there are references to various files provided by the Arduino software. Because of the way I've installed my software and the fact that the installer versions of the download may install to a different location, most paths used in this book will be relative to `/home/norman/.arduino15`.

Paths used will be as follows:

When executing the IDE, it will be found in `/home/norman/arduino-2.1.0/arduino-ide` where the downloaded zip file was extracted. However, most of the interesting files, those for the Arduino Language, are to be found elsewhere.

- `$ARDBASE` is `/home/norman/.arduino15`, the location where the IDE installed the AVR packages for the Uno and other AVR boards.
- `$ARDINST` is `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6`, the location of the main Arduino files for AVR microcontrollers. This is where the various cores, bootloaders, and so on can be found, beneath this directory.
- `$ARDINC` is `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6/cores/arduino`, the location of many of the `*.h` header files and most of the `*.c` and `*.cpp` files that comprise the Arduino Language for AVR microcontrollers. This is `$ARDINST/cores/arduino`.
- `$TOOLS` is where the AVR tools reside in the downloaded packages for the AVR boards. Here, you will find *avrdude* and the AVR Library which underlies a lot of the Arduino Language itself. On my system, this is `/home/norman/.arduino15/packages/arduino/tools`.
- `$AVRINC` is where the header files for the version of the AVR Library provided by the Arduino IDE are located. In the new IDE, these are now dependent on the version of the compiler in use—*avr-g++* by default—so the path can be quite convoluted.

  The Arduino Language (eventually) compiles down to calling functions within the AVR Library (henceforth referred to as AVRLib), and the header files are to be found in location

`/home/norman/.arduino15/packages/arduino/tools/avr-gcc/xxxxx/avr/include/avr`.

Here, "xxxxx" is the *avr-g++* compiler version and name, currently `7.3.0-atmel3.6.1-arduino7`, but this may change as new releases of the compiler are implemented by the IDE.

So, if you see a file referred to as `$ARDINC/Arduino.h` in the text, you will know that this means the file

- `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6/cores/arduino/Arduino.h` on Linux.

You can see why I'm using abbreviations now, can't you?

If you wish to examine the files on your system that I am discussing in the book, see Appendix A for a couple of useful tips on how to avoid always having to type the full paths.

1.2 Coding Style

Code listings in the book will be displayed as follows:

```
#define ledPin LED_BUILTIN
#define relayPin 2
#define sensorPin 3
...

void loop()                                                    (1)
{
    // Flash heartbeat LED.
    digitalWrite(ledPin, HIGH);
    delay(100);
    digitalWrite(ledPin LOW);                                  (2)
    ...
}
```

(1) This is a callout that attempts to bring your attention to something in the code which will be described beneath the code listing in question.
(2) This is another callout; there can be more than one.

In the book's main text, where you see words formatted like `USCR0A` or `PORTB`, then these are examples of Arduino pin names, AVR microcontroller registers, bits within those registers, and/or flags within the ATmega328P itself, as well as references to something listed in the data sheet for the device. Where code listings are being explained, then variables from the code will be shown in this style too.

Arduino pin numbers will be named `Dn` or `An` as appropriate. This is slightly different from the normal usage of the digital pins, which normally just get a number; I prefer to be a little more formal and give the digital pins their full title. <grin>

Tip
Tips are exactly that. They give you a clue about something that may not be too well known in the Arduino world, but which might be incredibly useful. (Or, maybe, just slightly useful!)

Note
This is a note. It brings your attention to something that may require a little more information. It could be useful to pay attention to these notes. Maybe!

Warning
Warnings are there to highlight potential problems with something in the software or just something that the data sheet needs you to take extra care over. There may be a possibility of damage to your Arduino if you don't pay particular attention. Occasionally, the data sheet warns against doing something—so it's best not to do what it says not to do!

1.2.1 Number Formats

Throughout this book, I need to refer to numbers in decimal, binary, or hexadecimal, from time to time. To this end

Binary	Binary numbers are written with a prefix of "0b" and a space every four bits, for example, 0b0101 1011 0000 1101. All binary numbers will have this prefix, apart from those which are single bit, that is, 0 and 1.
Hexadecimal	Hexadecimal numbers are written with a prefix of "0x". There are no spaces in hexadecimal numbers, for example, 0x5B0E.
Decimal	These numbers are written as you and I would normally write them, with no prefixes. Commas will be used to separate the major groupings, for example, 23,310.

1.3 The Arduino Language

I should perhaps point out that there isn't really such a thing as the Arduino Language. I may refer to it frequently within the pages of this book, but technically, it doesn't exist. What it is is simply an abstraction of the C/C++ language, written in such a way as to make life easier for people learning to make stuff with their Arduino. Which of the following is easier to understand?

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

or

```
PORTB |= (1 << PORTB5);
```

The first is definitely the easiest to understand; however, the latter is by far the quicker of the two as it just does what it says; it sets pin 5, on `PORTB` of the ATmega328P, to `HIGH`. The name, `digitalWrite()`, appears to be a different language, but it isn't; it's that abstraction away from plain AVR C/C++ which makes life easier for us all.

1.4 Coming Up

In Chapter 2, I explain how a sketch gets massaged into a proper C++ program and how the libraries used in the sketch are incorporated into it. Following the brief overview of how compiling a sketch operates, I then document the Arduino's `main()` function, the various header files that it includes, and the initialization carried out by the `init()` function. These initializations are part of every sketch that you compile, so it helps if you know what the Arduino system is doing, hidden in the background, just for you.

In Chapter 3, I explain about the features and facilities of the Arduino Language. This will include all the commands such as `pinMode()`, `digitalWrite()`, and so on. I talk through all the functions that relate to the Arduino, with particular emphasis on the code that applies to the standard Arduino boards, those based on the ATmega328P family of AVR microcontrollers.

Chapter 4 looks into a number of the C++ classes (or objects) which are supplied with, and used by, the Arduino Language. The classes of main interest here are the `HardwareSerial` class which provides us with the Serial interface and its commands like `Serial.begin()` or `Serial.println()`. However, the `HardwareSerial` class is not fully self-contained, so the other, lesser known, supporting classes are also explained in this chapter.

Chapter 5 takes a brief look at how to cast off the bonds of the Arduino Language and delve into the brazen world of AVR C++ itself, where you bypass the likes of `pinMode()` calls and talk to the AVR microcontroller in something akin to its own language. Here, you will learn how you can set the `pinMode()` for up to eight pins with a single instruction or how to `digitalWrite()` those same eight pins, again with one instruction, and other efficient methods of communication with your board.

Chapter 6 demonstrates a couple of alternatives to the Arduino IDE. Some people don't get on with it; I myself have a sort of love-hate relationship with it as I find versions 1.x of the editor a little clumsy and slow for my liking. The new, improved versions 2.x of the IDE are much, much better, however.

In this chapter, I will show you how you can write code for Arduino boards in both the Arduino Language and plain AVR C/C++ code using the "PlatformIO" system and also give you a hefty overview of the latest release of the *arduino-cli* utility used in versions 2.x of the IDE but available for stand-alone use in `Makefiles`.

Chapter 7 is where I delve deeper into some features of the ATmega328P which, while not strictly software, are fundamental to configuring the ATmega328P how you might like it and not as the Arduino designers, however talented they may be, have decided.

In this chapter, I'll be looking at the ATmega's fuses, power reduction modes, sleep modes, and similar features which determine how the ATmega328P works, but not necessarily what it does.

Chapters 8 and 9 are where I delve deeper into some more features of the ATmega328P which, while not strictly software, are either important in understanding the Arduino Language or just useful to know about. Hardware features such as the Analog Comparator (AC), Timer/counters—referred to as timers henceforth—the Analog-to-Digital Converter (ADC), and the Universal Synchronous/Asynchronous Receiver/Transmitter (USART) are covered in some detail.

Finally, in the Appendixes, there are a number of topics that may be of interest or are kept together in one place for reference. In here, you will find all the helpful reference material you might need, such as pinout diagrams, and potentially useful (or unusual) code to upload to your Arduino.

There's even an index!

Without any further ado, let's dive in to what happens when you want to compile a sketch in the Arduino IDE.

Arduino Compilation

2

This chapter is all about what happens when you compile an Arduino sketch and how the various header files are used. Hopefully, by the time you have read (and understood) this part of the book, you'll have a much better idea of what happens during the compilation of an Arduino sketch. However, before we dive into the gory details of a sketch's compilation, we need to understand a bit about some of the text files that live in and around the `$ARDINST` directory.

These files are used to set up the IDE's menu options and to define the AVR microcontroller and Arduino board to be used. Additionally, the IDE needs to know how to compile and upload sketches, and with lots of different boards nowadays, not just those with AVR microcontrollers, these numerous text files help the IDE configure the build tools and so on, for the specific board chosen from the Boards menu in the IDE.

Once we have discussed the various text files, we can then get down and dirty in the compilation process and also take a look at the hidden C++ files that the Arduino environment keeps well away from us.

2.1 Settings.json

The file `settings.json` holds all the preferences for the Arduino IDE, and under versions 2.x of the IDE, it is found in `/home/norman/.arduinoIDE` which is a new hidden directory, created on the first run of the version 2 IDE. On Windows, this would be `C:\Users\norman\.arduinoIDE`.

You should be able to find the file after the first run of the IDE; if you have not yet done so, there will not be a `settings.json` file to be found.

As you may have guessed, the file is in JSON format, which is still a text format, but has a different layout to the `preferences.txt` file in previous versions of the IDE.

In the IDE, if you click File ▹ Preferences, a dialog will be displayed showing the current preferences. These have been read from `settings.json`. Just like the old `preferences.txt` file, there are limited preferences that can be set on this dialog. However, there are a lot more preferences than meets the eye!

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_2

2.1.1 Finding Other Hidden Settings

The new IDE has a hidden preferences system similar to that in *VSCode*. You access it via the CTRL+SHIFT+P key combination. This opens a new search bar in the editor and waits for you to type something. Type "settings" without the quotes, which will give you a number of options. For example:

- Open Settings (UI)
- Open User Settings
- Open Workspace Settings
- Open Workspace Settings (JSON)

The last option will allow you to edit the `settings.json` file directly in the IDE; this is very useful and, at times, quicker than using CTRL+SHIFT+P. The others all appear to display the same dialog in the IDE, and there are numerous settings available, too many to be honest.

A small example of using this option follows where we will search for the current tab size and change it to "4," but inserting spaces instead of a hard TAB character.

2.1.2 Setting Tab Stops

Now, you would think that an editor, for writing code, would at least allow you the ability to easily adjust the width of the tab stops—not so the Arduino IDE!

All is not lost, as we do have that ability, but it involves editing the `settings.json` file; however, we don't have to edit it manually. These instructions only apply to version 2.x of the IDE:

- Open the IDE if not already open.
- Press CTRL+SHIFT+P.
- Type "settings" without quotes.
- Choose "Open Settings (UI)."

A new tab named "Preferences" will open in the IDE. There are two main options to the left side:

- User
- Workspace

The former will affect everything the current user does in the IDE; the latter will affect only the current workspace or sketch.

- Click "User" and note that there are a number of categories of settings that can be changed.
- In the search box, type "tab size" without quotes. One of the displayed options is "Editor: Tab Size."
- Change the default of "2" to some other value that you prefer; I like four spaces, so I've configured mine to be "4."

Another useful setting to search for is "spaces." This will allow you to configure the IDE to insert spaces instead of actual tab characters. "Editor: Insert Spaces" is the appropriate setting.

Search or scroll through any of the other settings and configure the defaults for your user as you desire. The changes take place immediately, and you no longer have to close the IDE and reopen it to make the changes happen.

Once happy with all your changes, close the Preferences tab.

The preceding changes cause tabs to indent four characters from the default of two characters. I don't know about you, but I find two character indents quite unreadable when looking at the structure of a sketch; I use four for just about everything I do. This makes editing in the IDE a little more comfortable, in my opinion.

The `settings.json` file will be updated with your changes. If you subsequently upgrade the IDE to a newer version, as they become available, then your changes will not be overwritten.

2.2 Globally Defined Properties

Before the various text files are read, the Arduino IDE defines some properties defining various paths, and so on, for itself. These properties are global and can be used within any of the other configuration files, including your own. These globally defined properties are listed next.

`runtime.platform.path`	The absolute path of the directory which is the folder containing the current `boards.txt` file (`$ARDINST`, for example).
`runtime.hardware.path`	The absolute path of the hardware directory which is the folder containing the current `platform.txt` file (also `$ARDINST`).
`runtime.ide.path`	The absolute path of the directory where the *arduino-ide* application, the Arduino IDE, or the *arduino-cli*, if that is currently being used to compile a sketch, is found. For the version, this is simply where you extracted the zip file. If you installed the flatpak version, it's where the *arduino-ide* and/or *arduino-cli* executables are to be found.
`runtime.ide.version`	The version number of the Arduino IDE as a six-digit number. Each component of the version number will be converted to use two digits. Then all the dots are stripped out, and finally, any leading zeros are removed, leaving the final value. For example, the Arduino IDE version 2.1.0 will become "02.01.00" which becomes "020100" before finally being assigned as `runtime.ide.version=20100`. IDE versions prior to version 1.6.0 used a single digit for the IDE version number. For example, version 1.5.6 was 156 as opposed to 10506.
`ide_version`	An alias for `runtime.ide.version`, used for compatibility with previous versions.
`runtime.os`	The operating system that the IDE is currently executing on. The values are "linux," "windows," and "macosx."
`software`	The name of the software. Set to "ARDUINO."

`name`	The name of the platform vendor.
`_id`	The board ID of the board that the sketch is being compiled for. Taken from the "name" parameter for the board in the `boards.txt` or `boards.local.txt` files.
`build.fqbn`	The board's fully qualified board name. Used when compiling sketches. For an Uno, this will be "arduino.avr.uno" in the format of "vendor.architecture .board_id". For my Nano, it's "arduino.avr.nano:cpu =atmega328" which is the format "vendor.architecture.board_id:menu_idv=option". Multiple options are permitted, comma separated.
`build.source.path`	The absolute path of the sketch being compiled. If the sketch has not yet been saved, this will point to a temporary directory.
`build.library_discovery_phase`	If zero, then this is the normal phase of the build. If one, then this is in the discovery phase of the build where the IDE does lots of work in the background to ensure that your sketch becomes a valid C++ source file, with all headers included, function prototypes inserted, and so on.
`compiler.optimization_flags`	"Debug" or "Release" according to the compilation in progress. The IDE sets this using the Sketch ▷ Optimize for Debugging option.
`extra.time.utc`	Unix time, in seconds since the epoch—00:00:00 on 01/01/1970—as per the machine that the build is running on. UTC.
`extra.time.local`	Unix time with local timezone offsets and Daylight Saving Time (DST) applied.
`extra.time.zone`	Local timezone offset from UTC. Does not include any DST adjustments.
`extra.time.dst`	Local timezone offset for DST.

These global settings may be used in `platform.txt`, `boards.txt`, or perhaps, but not very likely, in `programmers.txt`. You may also use these paths in your amendments to the configuration files or in the various "local" versions that you create.

Note
Interestingly, while the IDE version 1.8.19 correctly gives 10819 for `runtime.ide.version`, IDE version 2.1.0 is hardcoded to use 10607 which implies that it is really IDE version 1.6.7!

You can see the setting in a *clean* verbose compile. Find the line that states "Compiling sketch…"; the line after that is the first compilation line. In the command-line option, "-DARDUINO=xxxxx", "xxxxx" is the `runtime.ide.version`.

Tip
Various configuration files can have a local version; `boards.txt`, for example, may have `boards.local.txt`. This local version allows you to make changes to the system configuration and not have to reconfigure every time the Arduino IDE is updated.

Unfortunately, not all of the configuration files have a local version—`programmers.txt` is one that I have come across that doesn't. See https://github.com/arduino/Arduino/issues/8556 for details, if you are interested.

2.3 Boards.txt

The `$ARDINST/boards.txt` file defines the various menu options for different types of microcontroller devices. These options will either appear on the Boards menu in the Arduino IDE or will be used when a specific board is selected from that menu. The file is read and the various options are decoded and used by the IDE at startup.

New boards can be added quite simply, if desired, by editing this file, although it's better to add any changes to the `boards.local.txt` instead—to prevent your changes from being overwritten when an update is applied.

You should be aware that changes have been made to the manner in which some parameters are listed in `boards.txt` and `boards.local.txt`. This appears to be a result of changes made to the IDE between versions 1 and 2.

You will see both forms of the settings from time to time, as the following two versions of the same setting show:

```
uno.upload.tool=arduino:avrdude
uno.upload.tool=avrdude
```

The first line is the format used in the newer IDE, while the second is the old style. Currently, both variants are accepted—at least, in version 2.1.0 of the IDE.

Let's look inside `boards.txt` at the entry for the Arduino Uno.

2.3.1 Arduino Uno Example

The following is the complete listing of all entries for the Arduino Uno, in the IDE version 2.1.0:

```
uno.name=Arduino/Genuino Uno                        (1)

uno.vid.0=0x2341                                    (2)
uno.pid.0=0x0043
uno.vid.1=0x2341
uno.pid.1=0x0001
uno.vid.2=0x2A03
uno.pid.2=0x0043
uno.vid.3=0x2341
uno.pid.3=0x0243
uno.vid.4=0x2341
uno.pid.4=0x006A
uno.upload_port.0.vid=0x2341                        (3)
uno.upload_port.0.pid=0x0043
uno.upload_port.1.vid=0x2341
uno.upload_port.1.pid=0x0001
uno.upload_port.2.vid=0x2A03
uno.upload_port.2.pid=0x0043
uno.upload_port.3.vid=0x2341
uno.upload_port.3.pid=0x0243
uno.upload_port.4.vid=0x2341
uno.upload_port.4.pid=0x006A
uno.upload_port.5.board=uno

uno.upload.tool=avrdude                             (4)
uno.upload.protocol=arduino
uno.upload.maximum_size=32256
uno.upload.maximum_data_size=2048
uno.upload.speed=115200

uno.bootloader.tool=avrdude                         (5)
uno.bootloader.low_fuses=0xFF
uno.bootloader.high_fuses=0xDE
uno.bootloader.extended_fuses=0xFD
uno.bootloader.unlock_bits=0x3F
uno.bootloader.lock_bits=0x0F
uno.bootloader.file=optiboot/optiboot_atmega328.hex

uno.build.mcu=atmega328p                            (6)
uno.build.f_cpu=16000000L
uno.build.board=AVR_UNO
uno.build.core=arduino
uno.build.variant=standard
```

(1) This is the board name.
(2) This section defines identification settings used to determine the board's identity when it is plugged into the USB port on your computer.
(3) This section defines serial discovery port properties used to determine the board's identity when it is plugged into the USB port on your computer. These settings are only used if the platform supports serial discovery.
(4) These settings define parameters used for uploading compiled code to the board.
(5) Bootloader settings are listed in this section.
(6) Various build options are specified here.

The Arduino Wiki at https://arduino.github.io/arduino-cli/0.30/platform-specification/ states that

> This file contains definitions and meta-data for the boards supported. Every board must be referred through its short name, the board ID. The settings for a board are defined through a set of properties with keys having the board ID as prefix.

The board ID prefix mentioned is, in this case, "uno." This is extracted from each of the "xxx.name" entries in the `boards.txt` file.

2.3.1.1 Board Identifier

The name parameter here identifies the board and defines what name will be displayed in the Tools ▷ Board menu in the IDE:

```
uno.name=Arduino/Genuino Uno
```

When you select "Arduino/Genuino Uno" from the board selector dialog, then the properties of an Uno are read from the `boards.txt` file, ready for use.

2.3.1.2 Identification Settings

This section's settings help to identify a genuine Arduino Uno. When you plug a device into a USB port, the device is queried to obtain a vendor and product identifier. This helps the system load the correct drivers (mainly for Windows) or, on the very first time, to prompt you to load the appropriate drivers for the device. For the Uno, the following five pairs of vendor and product identifiers are known to be genuine:

```
uno.vid.0=0x2341
uno.pid.0=0x0043
uno.vid.1=0x2341
uno.pid.1=0x0001
uno.vid.2=0x2A03
uno.pid.2=0x0043
uno.vid.3=0x2341
uno.pid.3=0x0243
uno.vid.4=0x2341
uno.pid.4=0x006A
```

In the settings

- Vid is the vendor identifier.
- Pid is the product identifier for the specific vendor.

From this, we can clearly see two vendors—"0x2431" and "0x2A03"—and the appropriate product identifiers to suit each vendor. Bear in mind that it isn't necessarily the actual manufacturer of the Arduino board that is being identified, it is most likely to be the chip that converts the data on the USB port into the correct format for the microcontroller. Some Uno boards have another AVR microcontroller taking care of the communications, while others have an FTDI chip—both will register as different pids.

Note
Genuine boards, such as my own Duemilanove, which use an FTDI chip for communications, will not necessarily be recognized as the correct board. This is due to the FTDI chip which uses a generic pid and vid and is used by numerous different boards. However, this is nothing to worry about.

Following the vendor and product IDs, we have pluggable discovery settings based on the five pairs of vendor and product IDs earlier. These are not available for every board, only those which the platform supports pluggable discovery.

```
uno.upload_port.0.vid=0x2341
uno.upload_port.0.pid=0x0043
uno.upload_port.1.vid=0x2341
uno.upload_port.1.pid=0x0001
uno.upload_port.2.vid=0x2A03
uno.upload_port.2.pid=0x0043
uno.upload_port.3.vid=0x2341
uno.upload_port.3.pid=0x0243
uno.upload_port.4.vid=0x2341
uno.upload_port.4.pid=0x006A
uno.upload_port.5.board=uno
```

These properties, which are associated with a port, can be used to identify a board connected to that port. There is a simple algorithm involved:

If a board's properties in `boards.txt` and/or `boards.local.txt` declare a set of `upload_port` properties like these, then if there is a match with the port properties discovered by the serial discovery phase, then this board is a likely candidate.

As there may be different boards with the same properties, the board is only a candidate until properly identified.

The port properties returned from the serial discovery are

```
"port": {
  "address": "/dev/ttyACM0",
  "properties": {
    "pid": "0x006A",
    "vid": "0x2341",
    "serialNumber": "1234567897",
    "name": "ttyACM0"
}
```

Here, we have a "pid" and a "vid" that can be used to identify the board. If `boards.txt` lists a board with those two properties, then the board can be identified by its name in the file. In this example, this corresponds to a board setting for `uno.upload_port.4`, so, in this case, the board must definitely be an Uno.

If you wish to find out more about serial discovery, https://arduino.github.io/arduino-cli/0.30/pluggable-discovery-specification/ will be a good place to start reading.

2.3.1.3 Upload Settings

When you click the upload button in the IDE, the settings defined in this section of the `boards.txt` file are used to set various parameters as desired, to enable proper communication with the currently chosen board:

```
uno.upload.tool=avrdude
uno.upload.protocol=arduino
uno.upload.maximum_size=32256
uno.upload.maximum_data_size=2048
uno.upload.speed=115200
```

To specify the tool to be used to carry out the upload, the `upload.tool` parameter is used. In this example, the tool in use is the program named *avrdude*. This tool is installed at the same time as the Arduino IDE.

The communications protocol to be used when uploading is defined in the `upload.protocol` parameter, while the maximum Flash and Static RAM sizes for the particular AVR microcontroller in use are defined in `upload.maximum_size` and `upload.maximum_data_size` parameters, respectively.

You may be wondering, if the maximum size of an ATmega328P's Flash RAM is 32,768 bytes, why is the `upload.maximum_size` only allowing 32,256 bytes? It's because the remaining 512 bytes are used for the bootloader. The Optiboot bootloader is 500 bytes in size, so at least that amount of Flash RAM needs to be reserved from the maximum available. The configuration options inside the ATmega328P only allow four different settings, and the nearest equal or larger setting to 500 is 512 bytes, so that's the size of the bootloader section for an Uno board.

Communications will be carried out at the baud rate specified in `upload.speed`. For this Uno example, that will be at 115,200 baud.

2.3.1.4 Bootloader Settings

This section of the `boards.txt` file defines various parameters to be used when you choose "Burn bootloader" from the IDE menu.

It should be obvious that burning a bootloader will require an In-Circuit System Programmer (ICSP) device as the ATmega328P you are burning a bootloader into doesn't yet have a bootloader, and even if it did, it wouldn't be able to easily overwrite itself.

Warning
You should be very careful to ensure that you have selected the correct board when burning a bootloader—on a good day, it will simply fail to work. On a bad day, it will set the fuses to something that might cause you some grief trying to unravel and get reprogrammed. On a *really* bad day, it could convert your prized Arduino board into something resembling a brick.

OK, it's probably not as bad as that warning insinuates, but you might end up with a need to purchase a new ATmega328P chip, and hopefully, it will be one that comes complete with an Uno bootloader burned in. Otherwise, you'll have to do the bootloader burning exercise all over again. You really don't want to be desoldering the surface mount version of the ATmega328P either—so take extra care!

Yes, I admit it, I did brick an Arduino board, so that's how I know. It was a Digispark board with an ATtiny85 microcontroller, but I bricked it anyway! Investigation showed that I set the fuse to disable the Reset pin so that it can be used as a normal I/O pin. No amount of programming with a high voltage programmer would rescue it, so there must have been some other settings that I broke as well.

I still have the device, somewhere, and one day, I will find out what I did wrong and, hopefully, fix it. Perhaps.[1]

Continuing to look at the standard settings for an Arduino Uno, we can see the following settings:

```
uno.bootloader.tool=avrdude
uno.bootloader.low_fuses=0xFF
uno.bootloader.high_fuses=0xDE
uno.bootloader.extended_fuses=0xFD
uno.bootloader.unlock_bits=0x3F
uno.bootloader.lock_bits=0x0F
uno.bootloader.file=optiboot/optiboot_atmega328.hex
```

[1] If you are wondering, no, I haven't fixed it yet!

To specify the tool to be used to carry out the upload, the `bootloader.tool` parameter is defined. In the case of the Uno we are looking at here, the tool in use is the program named *avrdude*—the same as before for uploading compiled sketches.

As described in Section 7.1, the AVR microcontroller has a number of fuses that can be utilized to set various configurations of the AVR microcontroller itself. The settings for `bootloader.low_fuses`, `bootloader.high_fuses`, and `bootloader.extended_fuses` define the required hardware settings for the microcontroller on the board.

Finally, in this section, the `bootloader.file` parameter defines which of the many bootloaders supplied with the IDE are to be used for this board. The Uno uses the file `optiboot/optiboot_atmega328.hex` which is to be found in the `$ARDINST/bootloaders/` directory.

You can, if you wish, change the bootloader by either editing the `boards.txt` file to change the appropriate parameter or duplicating an existing section and changing the bootloader. The latter option is preferred. It's worth bearing in mind that any updates to the IDE will most likely overwrite your changes to `boards.txt`, so how do we avoid this problem? See Section 2.4 for details.

2.3.1.5 Build Settings

```
uno.build.mcu=atmega328p
uno.build.f_cpu=16000000L
uno.build.board=AVR_UNO
uno.build.core=arduino
uno.build.variant=standard
```

The `build.mcu` setting defines the name of the microcontroller for this particular board. For the Uno, only an ATmega328P is defined. For other boards, the Nano, for example, there are two different microcontrollers available, the ATmega168 and the ATmega328P. Within each of those two boards, there are two different configurations each, and the `boards.txt` has entries for each variant with global settings for all Nanos as well as the specific settings for the different microcontroller boards and the variants thereof.

The parameter `build.f_cpu` defines the system clock (CLKcpu) for the board. The Uno has a 16 MHz crystal installed, so that's the speed that is defined in this example. This setting is used in your sketches, although you won't actually see it, as the `F_CPU` variable is used, for example, when calculating the desired baud rate when using the Serial interface.

The `build.board` property is used to set a compile-time variable `ARDUINO_<build.board>` to allow the use of conditional code between `#ifdefs` in sketches and/or header files. The Arduino IDE automatically generates a `build.board` value if not defined. In this example, the variable defined at compile time will be `ARDUINO_AVR_UNO`.

To determine which file path is to be used when the compiler is looking for various files, `main.cpp`, for example, the `build.core` setting is used. The parameter is used to build a path to the files in `$ARDINST/cores/<uno.build.core>/`, which, for the Uno in this example, will be `$ARDINST/cores/arduino/`.

The variant of the board is then defined using the `build.variant` setting. This is used to build a path to the files that live in `$ARDINST/variants/<uno.build.variant>/` and is where you will find the file named `pins_arduino.h` which defines any variations over the standard settings that apply to this particular board. For this example of an Uno, the path defined will be `$ARDINST/variants/standard/`.

Note
The IDE defines a number of global settings for the various paths to the cores and variants. These are available in other configuration files, but they don't have the board's prefix, so `uno.build.core` would correspond to the IDE's global setting of `build.core`. If the board doesn't specify a setting, the global one will be used; however, where a board does have an appropriate setting, that will override the global one created by the IDE, when the appropriate board is selected from the Boards menu in the IDE. You can see some of these global settings in the file `platform.txt`.

2.3.1.6 Configuring an ICSP

If you always want to use an ICSP (In-Circuit System Programmer) to program a particular board, you can add the following line to the `$ARDINST/boards.local.txt` file, as part of the build settings as detailed earlier, for example:

```
uno.upload.using=USBtinyISP
```

The name you use here is one of the ones that are to be found in the `programmers.txt` file which is itself described later in this chapter, in Section 2.6.

You should make this change while the IDE is closed. When you next open the IDE, any time you select the Uno device as your board, it will automatically select the USBtinyISP device, in this case, to perform the uploads, rather than the bootloader.

If you wish to make this change as the default for all boards, then you should edit the `settings.json` file, as documented in Section 2.1 earlier in this part of the book.

2.4 Boards.local.txt

Since release 1.6.6 of the Arduino IDE, a new file has been introduced, `$ARDINST/boards.local.txt`, in which you can define various parameters that you wish to use instead of those in the `boards.txt` file. To continue the preceding example of changing the bootloader, you could create the file, if it doesn't exist, and add the following to it:

```
uno.bootloader.file=path/to/myBootloaderAtmega328.hex
```

This assumes that you won't need any additional Flash RAM space for the bootloader over and above that required by the current bootloader. If you do, then add the following as well:

```
uno.upload.maximum_size=<what ever is now required>
```

You might also need to change the fuses that determine the size of the bootloader section. See Section 7.1 for details.

2.5 Platform.txt

The `$ARDINST/platform.txt` file defines platform-specific features and command-line tools, where libraries live and what they are called, and so on. It contains the various recipes used by the IDE in order to compile, build, upload, and/or program various devices and boards according to their different needs.

What is a platform? Well, in the case of the ATmega328P, or other AVR microcontrollers, the platform defines all the tools, compilers, linkers, command lines to be used, and so on, for Atmel AVR microcontrollers. Other non-AVR microcontroller boards will have their own platform to define the specific tools and others for that particular microcontroller. Arduino boards with, for example, an ARM chip on them will use a different platform to those with the AVR microcontrollers.

Using this method allows for a fairly simple manner in updating the system to cope with new boards.

The Arduino system requires that this file define the following metadata:

```
name=<platform name>
version=<platform version>
```

The name will be shown in the Tools ▷ Boards menu of the Arduino IDE, in grayed out text, above the list of boards that conform to this particular platform. According to the documentation on the Arduino website at https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5-3rd-party-Hardware-specification, the version is currently unused and is reserved for future use.

For the Arduino IDE version 2.1.0, we see this at the top of the file:

```
name=Arduino AVR Boards
version=1.8.6
```

Obviously, the version number of the platform, 1.8.6, differs from the IDE version, 2.1.0. Don't be confused when you see the different version numbers.

2.5.1 Build Recipes

The `platform.txt` file, as mentioned earlier, contains a large amount of metadata that configures the IDE to be able to compile sketches and upload them, among other things, for the Arduino boards running with AVR microcontrollers. It does this using *recipes*. Having different recipes for all the different platforms allows the IDE[2] to be used for a myriad of different devices.

When you select a build in the IDE, a small number of settings are created automatically for you; these are the following:

`build.path`	The path to the temporary folder to store various files created by the build process.
`build.project_name`	The project (sketch) name.
`build.arch`	The microcontroller architecture, which, in our case, is "avr" but, depending on the board, may be "sam," "arm," and so on. The IDE gets this from the paths to `$ARDINST`. On my system, `$ARDINST` is defined as `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6`,

[2] I say "IDE," but in actual fact, it is the underlying *arduino-cli* which does the actual work required.

and the path portions define the hardware folder location, `$ARDBASE/packages`; then the vendor name, `arduino`; the `hardware` directory; and finally, the architecture, `avr` for boards with the ATmega328P. For Arduino SAM boards, the path would be `$ARDBASE/packages/arduino/hardware/sam` instead. The final part of the path gives the `build.arch` name.

A number of additional settings are defined within `$ARDINST/boards.txt`, based on the particular board chosen on the Tools ▹ Boards menu—see Section 2.3 for those details—and the IDE global variables can be used within this file too. You can find more details on those variables in Section 2.2.

The compilation process can read source files written in plain C—these are the `*.c` source files, C++ (`*.cpp`), or AVR assembly language source files (`*.S`). It has to know how to convert these files into object files (`*.o`) which can be gathered together by the linker to create a library or an executable file. The way this happens is by using the recipes within the `platform.txt` file.

The recipes are defined in the format

```
recipe.<source_format>.o.pattern
```

And the “source_format” is simply the file extension for the files in question. This gives us the recipes:

- `recipe.c.o.pattern`: To convert C source files to object files
- `recipe.cpp.o.pattern`: To convert C++ source files to object files
- `recipe.S.o.pattern`: To convert assembly language source files to object files

You will notice, I hope, that the source format in each is case sensitive. Assembly language files must have an uppercase “.S” extension.

Taking one of these as an example, this is what I found in my `platform.txt` file for the Arduino IDE version 2.1.0:

```
## Compile c files
recipe.c.o.pattern="{compiler.path}{compiler.c.cmd}"
{compiler.c.flags} -mmcu={build.mcu} -DF_CPU={build.f_cpu}
-DARDUINO={runtime.ide.version} -DARDUINO_{build.board}
-DARDUINO_ARCH_{build.arch} {compiler.c.extra_flags}
{build.extra_flags} {includes} "{source_file}"
-o "{object_file}"
```

If we break the preceding recipe, which is all on a single line, into its constituent parts, we see the following variables and command-line options:

`"{compiler.path}{compiler.c.cmd}"`	Defines where the compiler tool can be found—`{compiler.path}`—and what it is called—`{compiler.c.cmd}`. The use of double quotes allows for spaces and other non-alphanumeric characters in the path or command name. In the IDE, I see that `{compiler.path}` is defined as `runtime.tools.avr-g++.path/bin/`, and as you can see, these recipes

	can refer to other variables defined in this file or elsewhere. The `{compiler.c.cmd}` is defined as `avr-g++`, and this is not actually the compiler, but a front end to all phases of the compilation process and which can be used to control the whole process.
`{compiler.c.flags}`	Defines a list of flags and options to be passed to the `{compiler.c.cmd}` utility to define how the build should progress, what outputs are required, and so on. There are numerous options in the default compiler flags, but one in particular, `-c`, tells the compiler front end to stop compiling after the object file has been created and not to run the link phase.
`-mmcu={build.mcu}`	Defines another compiler option. It tells the C compiler which microcontroller is in use on the board. It comes from `boards.txt` and, for the Uno, is defined as `uno.build.mcu=atmega328p`. The name part, uno, is stripped off first.
`-DF_CPU={build.f_cpu}`	This variable is very useful when writing code for multiple boards. The speed of the AVR microcontroller clock is defined in the compilation process, as opposed to being hardcoded in the actual source files, for example, `#define F_CPU 16000000L`. This would require editing before running on a board with a different clock speed. The variable is defined in `boards.txt`, and, for the Uno, it is `uno.build.f_cpu =16000000L`. The name part, "uno.", is stripped off first.
`-DARDUINO={runtime.ide.version}`	Defines a numeric value for the Arduino IDE version in use. It is created automatically by the IDE and is described in Section 2.2. For IDE version 2.1.0, for example, it becomes 20100. Well, it should, but is actually hardcoded in the *arduino-cli* source as 10607. There is an issue raised about this; see https://github.com/arduino/arduino-cli/issues/725 for details.
`-DARDUINO_{build.board}`	References the `uno.build.board` variable from `boards.txt`—the example shown is, once more, for the Uno. This can be used in conditional code to determine the board in use and, from that, whether certain features are available or otherwise. In this example, it would define a variable named `ARDUINO_AVR_UNO`.

`-DARDUINO_ARCH_{build.arch}`	As described earlier, it defines the architecture we are building for. For the purposes of this book, this will be “avr” giving `ARDUINO_ARCH_avr`.
`{compiler.c.extra_flags}`	Are some additional flags that you or I can define in `platform.local.txt` to be added to the command line for this recipe. By default, these are blank.
`{build.extra_flags}`	Are some additional flags that you or I can define in `boards.local.txt` to be added to the command line for this recipe. By default, these are blank.
`{includes}`	Is the list of paths that the compiler will search for files `#included` in the source file. The format is `-I/include/path` and so on. You can have more than one path. The documentation online has this to say: Note that older IDE versions used the `recipe.preproc.includes` recipe to determine includes, which is undocumented here. Since Arduino IDE 1.6.7 (arduino-builder 1.2.0) this was changed and `recipe.preproc.includes` is no longer used. This is not really very helpful, as includes remains undocumented, and even the “no longer used” recipe `recipe.preproc.includes` actually has {includes} as part of its definition.
"{source_file}"	This is the path to the single source file being compiled. The double quotes allow for spaces and other non-alphanumeric characters in the file name.
-o"{object_file}"	This is the path to the single object file which will be created by the compilation phase. The double quotes allow for spaces and other non-alphanumeric characters in the file name.

2.5.2 Pre- and Post-Build Hooks

Pre- and post-build hooks were introduced at Arduino version 1.6.5 and *were* found in the `platform.txt` file. In version 2.1.0, they appear to have been removed again.

2.6 Programmers.txt

The `$ARDINST/programmers.txt` file is very much the least documented of the various text files used by the Arduino IDE. The Wiki pages describe the other files, but nothing at all, other than a brief mention of its name, for `programmers.txt`.

It is assumed, possibly incorrectly, that people creating and building new ICSP devices know what all the parameters mean and will supply a list of required entries, for their device, to be added to `programmers.txt`.

It is also likely that whenever anyone creates or updates a programming device, or settings, it would be submitted to the Arduino maintainers for inclusion in the next release of software.

Warning
The `programmers.txt` file will be overwritten by each new IDE update, so if you have made any changes, you really should keep a record of them prior to upgrading.

`Programmers.txt` holds details about the various programming devices that the Arduino IDE can use to upload code to your Arduino board. Unlike `boards.txt` and `platform.txt`, the IDE doesn't seem to recognize a local variant, `programmers.local.txt`, even if one exists. Therefore, any changes that you make to your own installation will need to be made to the supplied `programmers.txt` file, and this *will be overwritten* when the IDE is upgraded.

I logged issue 8556 about it at https://github.com/arduino/Arduino/issues/8556[3] as this is a bit of a nuisance. It appears that `programmers.local.txt` is not supported, nor will it be.

The `programmers.txt` file contains parameters that are relevant to the various ICSP programming devices that can be used, and depending on the settings, these may appear in the various menu options under the Tools menu in the IDE.

An example of an entry in the file is as follows:

```
usbtinyisp.name=USBtinyISP
usbtinyisp.protocol=usbtiny
usbtinyisp.program.tool=avrdude
usbtinyisp.program.extra_params=
```

This is for the "USBtiny" programmer and shows the following:

- The device name, as it will appear in the Tools ▹ Programmer menu. In this case, it is "USBtinyISP." You can change this if you prefer to use a different name.
- The protocol to be used when executing the IDE option to "Upload using programmer."
- The tool that will be used when uploading. Here, we can see that it is defined as using the *avrdude* utility.
- Any extra parameters that may be needed to do the upload. In this example, there are none. However, if any were needed, they would be required to be consistent with the syntax of the programming tool in use.

If, for example, the device required a serial port to be used for the upload, then you could add the following:

```
usbtinyisp.program.extra_params=-P{serial.port}
```

This would allow the command line passed to *avrdude* to be supplied with the `-P` option to select a serial port, and it would be set to the value chosen in the IDE on the Tools ▹ Port menu option.

[3]Now redirected to https://github.com/arduino/arduino-cli/issues/989

Note
This is obviously just an example; the USBtiny device doesn't need a serial port.

For brief documentation on `programmers.txt`, see the Arduino documentation at https://arduino.github.io/arduino-cli/0.30/platform-specification/#programmerstxt. This file is still pretty much undocumented.

2.7 Compiling a Sketch

When you open a project in the Arduino IDE, you will notice that all files in the project directory with an `.ino`, `.h`, `.c`, `.cpp`, or `.S` extension get placed on a tab of their own. These are assumed to be all the source files that make up your project. You can, if you wish, open other files within the IDE, but these will not automatically open in separate tabs when you subsequently reopen the project; they will have to be manually opened if editing or viewing is required.

You should also be aware that there is not a function called `main()` in any of the files open in the project. Anyone who has programmed in C or C++ will know that `main()` is the program's entry point. What's going on?

The Arduino IDE supplies its own `main()` function, so that you don't have to. In order to make life easier for budding microcontroller makers and developers, the Arduino system hides a lot of stuff from you. I'll be taking a look at the `main()` function soon.

When you compile a project in the Arduino IDE, a number of things take place, and these separate processes are described in the remainder of this chapter. Your sketch—all of the `.ino` files—will, first of all, be converted into a single C++ file.

2.7.1 Arduino Sketch (*.ino) Preprocessing

An Arduino sketch is a very much simplified C++ source file which may be composed of many files with the extension `.ino` and occasionally some additional files with the extension `.cpp` or `.S`. To convert the sketch into a valid C++ file, a number of actions are carried out:

- *Maybe* create a temporary compiler working folder in the system's main temporary folder or directory. This will be `/tmp/arduino/sketches/<BIG_NUMBER>/` on Linux. In this path, "<BIG_NUMBER>" is just a long string of 31 hexadecimal characters. This happens only on the very first compilation of this particular sketch.

  For the rest of this discussion, I shall refer to this folder as `$TMP`.
- If your sketch is composed of a number of `.ino` files, those files are concatenated into a single `.ino.cpp` file, in the `$TMP/sketch` subfolder, starting with the main sketch file which is the `.ino` file with the same name as the sketch's folder name. The remainder of the `.ino` files are appended to the end of the main one, in alphabetical order. If your sketch was named `Blink.ino`, then the generated file will be named `$TMP/sketch/Blink.ino.cpp`.
- The line `#include <Arduino.h>` is added at the beginning of the `.ino.cpp` file, if not already present.

- All libraries used in the sketch are detected, and the include paths for those libraries are discovered. This is done by running a dummy compilation with the output being discarded (by being sent to the `/dev/null` device on Linux) and processing any relevant error messages.
- Prototypes for all functions in the `.ino.cpp` file are generated. If, as occasionally happens, a valid function prototype cannot be automatically generated, you will need to add one explicitly to the `.ino` file that defines the failing function.
- The `.ino.cpp` file is processed so that there are relevant compiler preprocessor `#line` and `#file` directives so that error reporting will be accurate and refer to the correct lines in the correct source files, as opposed to referencing the lines within the concatenated `.ino.cpp` file.

These actions used to be performed by the *arduino-preprocessor* tool which lived on GitHub at https://github.com/arduino/arduino-preprocessor but are now carried out by the new *arduino-cli* utility, which is now distributed with the version 2 IDE. See Section 6.3 for details.

2.7.2 Arduino Sketch (*.ino) Build

After preprocessing, the build used to be completed by the *arduino-builder* tool, found at https://github.com/arduino/arduino-builder, but is also carried out by the new *arduino-cli* utility distributed with the version 2 IDE.

The build carries out a number of different steps:

- It compiles the `.ino.cpp` file, created by the preprocessing stage, into a module with a `.ino.cpp.o` extension, and this module file is stored in the `$TMP/sketch` subfolder created in the preprocessing stage described earlier.
- It compiles all other `.S`, `.c`, or `.cpp` files, including `main.cpp`, into separate `.o` modules in the `$TMP/sketch` subfolder.

Note
If the sketch's configuration—the board and so on—has not changed since the previous compilation, then some of these modules will be reused rather than recompiled. This saves time on the second and subsequent compilations.

- Any libraries used by the sketch will be compiled as separate modules too. Once again, these will be written as .o files in the `$TMP/libraries` subfolder.
- The Arduino core files are compiled as .o files into `$TMP/core`. These core files are the likes of `wiring_analog.c` or `wiring_digital.c` and so on, as installed in the `$ARDINST` location (`/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6` on my system).
- The individual core modules (`*.o`) are then built into a single static library, `core.a`, in the `$TMP/core` subfolder.
- After all the modules have been created, the linker combines them all into a single elf format binary file. This file lives in the main temporary folder created earlier and will be named `$TMP/<sketch_name>.ino.elf`, for example, `$TMP/Blink.ino.elf`.

- The `$TMP/<sketch_name>.ino.elf` file is then used to create a file named `$TMP/<sketch_name>.ino.eep` which contains data to be written to the AVR microcontroller's EEPROM area. This file is sadly never used even by the 2.1.0 release of the IDE. It is not possible, yet, to upload EEPROM data easily with the Arduino IDE.
- The `$TMP/<sketch_name>.ino.elf` file is also used to create a file named `$TMP/<sketch_name>.ino.hex` which contains the code used to flash the AVR microcontroller with your sketch. This code is in "Intel Hex"[4] format.

This ends the compilation process. If the upload button was clicked in the IDE, rather than the compile or verify button, then the `$TMP/<sketch_name>.ino.hex` file is uploaded to the AVR microcontroller, using an Arduino-specific version of the *avrdude* tool, which can be found on GitHub at https://github.com/arduino/avrdude-build-script.

The `$TMP/<sketch_name>.ino.eep` file is not currently uploaded if the upload button was clicked. This means that any EEPROM data the sketch is expecting will not be found when the sketch is running.

Tip
You can see all of this happening before your very eyes if you edit the preferences in the IDE to show verbose compiling and upload messages (File ▹ Preferences).

2.7.3 After the Build

The menu option, Sketch ▹ Export Compiled Binary, will export various binary files and the sketch's `.hex` files to the sketch's folder in a new subdirectory structure.

A directory named `build` will be created first. Beneath the `build` directory, another directory named after the fully qualified board name will be created. For the Uno, this will be `arduino.avr.uno`, and within this directory, you will find the following files:

- `Blink.ino.elf`: The actual compiled binary for the sketch.
- `Blink.ino.hex`: The uploadable version of the sketch.
- `Blink.ino.eep`: The EEPROM data for the sketch.
- `Blink.ino.with_bootloader.hex`: An uploadable version of the sketch, also containing the bootloader. This file requires an ICSP to upload it.
- `Blink.ino.with_bootloader.bin`: A binary version of the sketch plus the bootloader. This is exactly the same size, 32 KB in the case of the Uno, as the Uno's Flash RAM. This file is effectively a binary image of the Flash RAM in the Uno as it would appear after uploading the original sketch.

These file names are obviously for the standard Blink example.

The export binary option can be used for distributing copies of your application for anyone to upload, without letting them see your source code.

If you have a bootloader installed on your ATmega328P, then you can use it to upload the `.hex` files using *avrdude*; otherwise, an ICSP will be required.

[4]https://en.wikipedia.org/wiki/Intel_HEX

Normally, when using an ICSP to program your device, the bootloader will be overwritten when the chip is programmed. However, if you use the ICSP to upload the `<sketch_name>.ino.with_bootloader.hex` file, then you will burn a new bootloader as well as your sketch's code into the Arduino board.

Note
All of the preprocessing and build just works, and it helps make our Arduino lives easy; however, what is the Arduino system hiding from you?

The following sections describe the various files that your sketch ends up including or using during the compilation process.

2.8 The Arduino main() Function

As previously noted, the `main()` function is where all C or C++ applications start executing. As an Arduino developer though, you don't have to supply one as the system does it for you. The Arduino `main()` function in version 2.1.0 is found in the file `$ARDINC/main.cpp` beneath the Arduino installation directory and is shown in Listing 2-1.

Listing 2-1 The Arduino main() function

```
#include <Arduino.h>                                        (1)

// Declared weak in Arduino.h to allow user redefinitions.
int atexit(void (* /*func*/ )()) { return 0; }

// Weak empty variant initialization function.
// May be redefined by variant files.
void initVariant() __attribute__((weak));
void initVariant() { }

void setupUSB() __attribute__((weak));
void setupUSB() { }

int main(void)
{
    init();                                                 (2)
    initVariant();                                          (3)

#if defined(USBCON)                                         (4)
    USBDevice.attach();
#endif

    setup();                                                (5)

    for (;;) {                                              (6)
```

```
        loop();

        if (serialEventRun) serialEventRun();            (7)
    }

    return 0;
}
```

(1) The first point to note is the inclusion of the file `Arduino.h` (found in the folder `$ARDINC`), and this is where numerous constants and other definitions specific to the Arduino are declared. If you look at this file, it makes interesting reading, there are numerous tests to determine which board is in use and which features of the AVR microcontroller can be used. The `Arduino.h` header file is described in Section 2.9.

(2) Within the `main()` function itself, there is a call to `init()` which is found in `$ARDINC/wiring.c`. This initializes a whole raft of features for the Arduino and carries this out based on the actual microcontroller in use on the board. If you decide to dive into this function, make sure that you are armed with a copy of the data sheet for your specific microcontroller; otherwise, nothing much will make sense.

(3) The next function call, `initVariant()`, carries out any special initialization for boards that are possibly not covered by the standard initialization. The function defaults to doing nothing (you can see it at the top of `main.cpp`), but as it is declared as weak, it can be overridden, as required, in a sketch.

(4) Some boards like the Leonardo use USB for serial communications and thus require USB setting up, so there is a test to see if this is required; if so, then the `USBDevice.attach()` function is called to do the needful.

(5) The sketch's `setup()` function is called next. This is where the sketch's own initialization gets carried out.

(6) After the call to `setup()`, an endless `for` loop is entered where the sketch's `loop()` function is called once on every pass through this `for` loop.

(7) The function `serialEventRun()` is called each time through the loop as well. This then calls out to another weak function named `serialEvent()`, if it exists in the sketch, and this is used to collect up any data that has been received into the Serial input buffer but not yet read by the sketch. There is an example of its use on the Arduino Tutorials website at www.arduino.cc/en/Tutorial/SerialEvent.

This is how the sketch's `ino` file fits into the real world; the `setup()` function is called once, and then `loop()` gets called repeatedly until the Arduino runs out of power or is turned off.

Calling the `loop()` function many times in this manner will impart some overhead to each execution. There is the stack frame setup prior to the function call and the stack teardown at the end after the function returns. These housekeeping instructions take time to execute and can slow down your code. You can avoid this by defining your own loop with a `for`, `while`, or `do` block within the `loop()` function, so that `loop()` only ever gets called once and never has to return to `main()`. Listing 2-2 shows a brief example.

Listing 2-2 A "never returning" loop() function

```
void loop() {
    while (1) {
        // Do your loop code here.
    }
}
```

Warning
Bear in mind that if you do decide to create your own loop in the manner described earlier, you *might* cause problems with any serial communications that would have been processed by the call to `serialEventRun()` in the `main()` function.

Check the documentation on the Arduino Tutorials website at www.arduino.cc/en/Tutorial/SerialEvent to be sure that you will not be causing yourself any worries.

2.9 Header File Arduino.h

As I have already mentioned, when you compile a sketch in the Arduino IDE, there is a certain amount of reorganization taking place to convert your sketch into something resembling a proper C/C++ program. The file `Arduino.h`, which can be found in `$ARDINC` as can all the other Arduino-specific header files, is included at the top of the converted source code. It is in, or from, this file that much of the initialization of a sketch takes place.

The following list outlines the actions of the `Arduino.h` file:

- Various standard C/C++ header files are included. I will not be discussing those here.
- A number of AVR-specific header files are included from the AVRLib sources, in `$AVRINC`; these are as follows:
  - `avr/pgmspace.h`
  - `avr/io.h`
  - `avr/interrupt.h`
- A weird[5] header file, `binary.h`, is included next.
- All the Arduino-specific function headers are defined, for example, `pinMode()`, `digitalWrite()`, and so on, along with a number of useful constants such as `HIGH`, `LOW`, `INPUT`, and `OUTPUT`, among others.
- If the compilation is using the C++ compiler (*avr-g++*) as opposed to the C compiler (*avr-gcc*), then
  - `WCharacter.h` is included.
  - `WString.h` is included.
  - `HardwareSerial.h` is included.
  - `USBAPI.h` is included.
- If the compiler discovers that the microcontroller in use has both hardware serial and CDC serial, then it cannot continue with the compilation, so an error is displayed and the compilation ends.
- Finally, the header file `pins_arduino.h` is included.

[5]Well, I think it's weird!

The relevant header files are described in the following sections, as are any other headers that they themselves include.

2.9.1 Header File avr/pgmspace.h

This header file is included from `$AVRINC` to allow `pins_arduino.h`, as described in the following, to create lookup tables within the program space—in flash memory—as opposed to in the scarce Static RAM (SRAM) on the device. It defines a number of `typdefs` and functions to copy data between the program space, in flash, and the variable space in RAM. It doesn't make for very interesting reading I'm afraid.

2.9.2 Header File avr/io.h

This file, from `$AVRINC`, sets up all the AVR-specific stuff for the appropriate AVR microcontroller that is in use on the Arduino board. The settings you chose in the IDE for Tools ▹ Board will determine the specific device file that will be included. In the majority of cases, and for our purposes here, this will be the ATmega328P, and so this file simply causes the AVR definitions for that microcontroller to be read in from the file `avr/iom328p.h`.

This file also includes `sfr_defs.h` to set up numerous macros for memory address simplification and some functions to handle looping until a bit is clear (or set) and so on. These are not discussed here.

Other files included by `avrio.h` are

- `avr/portpins.h`
- `avr/common.h`
- `avr/version.h`
- `avr/xmega.h`: Only if we are compiling for an XMega device, which we are not, so this header will not be discussed further.
- `avr/fuse.h`
- `avr/lock.h`

The AVRLib header files are to be found in `$AVRINC`, under your Arduino 2.1.0 installation, unless otherwise stated. These header files are not part of the Arduino IDE per se, but are supplied as part of the AVRLib, which the IDE uses.

2.9.2.1 Header File avr/iom328p.h

If you've ever looked at the data sheet for the ATmega328P, then you will notice that the various registers, and the bits thereof, have strange sounding acronyms. This header file, from `$AVRINC`, is the one which creates all the constants so that you can refer to those acronyms in your code. In addition to these acronyms, various other constants are defined to manage RAM sizes, fuse bits, sleep modes, interrupt vectors, and so on. This is another important header file, but it really doesn't make for good bedtime reading.

If your Arduino board uses a different AVR microcontroller device, then a different `iomxxx.h` file will be included, rather than this one, so the definitions will be suitable for the board and/or microcontroller in use.

The exact file which will be included is defined by the IDE's Tools ▹ Board settings.

2.9.2.2 Header File avr/portpins.h

This header file, from `$AVRINC`, is a continuation of the device-specific `avr/iom328p.h` file and defines some additional constants which are common to all of the other devices. Some of the definitions in this file will not be relevant to all devices, but the code in this file does do some checks to see if a definition will be relevant, before defining it. It does this by checking for constants defined in the `avr/iom328p.h` header file and, if defined, sets up the additional constants.

Note
Obviously, if the board in use is not based on the ATmega328P, then the reference to `avr/iom328p.h` earlier would of course be to a different header file for the device actually in use on the board.

2.9.2.3 Header File <avr/common.h>

According to the comments in this header

> This [sic] purpose of this header is to define registers that have not been previously defined in the individual device IO header files, and to define other symbols that are common across AVR device families.

I think that about covers it!
The file can be found in `$AVRINC`.

2.9.2.4 Header File <avr/version.h>

This is a header file specific to the AVRLib code and not to the AVR devices. It defines various constants to indicate which version of the AVRLib is in use. For example, for a version 2.0.0 AVRLib, we see the defined constants in Listing 2-3.

Listing 2-3 AVRLib constants

```
#define __AVR_LIBC_VERSION_STRING__ "2.0.0"
#define __AVR_LIBC_VERSION__ 20000UL
#define __AVR_LIBC_DATE_STRING__ "20150208"
#define __AVR_LIBC_DATE_ 20150208UL
#define __AVR_LIBC_MAJOR__ 2
#define __AVR_LIBC_MINOR__ 0
#define __AVR_LIBC_REVISION__ 0
```

You could use these constants to check whether a specific version of the library is in use and, from that, determine if some function can be used or otherwise.

Once again, the file can be found in `$AVRINC`.

2.9.2.5 Header File <avr/fuse.h>

Fuses are programmable bits in one, two, or three bytes inside the AVR microcontroller. These are used to set various features of the hardware and are covered in some detail in Section 7.1. The data sheet for the appropriate AVR device has full details and warnings!

This header file, from `$AVRINC`, sets up a structure type (`__fuse_t`) that corresponds to the fuse bits for the appropriate board.

2.9.2.6 Header File <avr/lock.h>

This header file, from $AVRINC, sets up lock bit details for the specific AVR microcontroller in use. This will not be discussed further here, so see the data sheet for full details.

2.9.3 Header File avr\interrupt.h

Because the init() function, as described in the following, sets up Timer 0 with an interrupt routine to keep track of the number of milliseconds (millis) that have passed since a sketch started, this header file is required. In essence, and among many checks, it creates the ISR() macro which allows you to create interrupt handlers when using AVR-specific C/C++ code. The Arduino Language uses a slightly different system, attachInterrupt(), for example, for external interrupts. Arduino interrupts will be described in Section 3.6.

This file is also part of the AVRLib and is found in $AVRINC.

2.9.4 Header File binary.h

This header file defines a constant for every numeric value from 0 through to 255, in the value's *binary* number format. The constants defined, using #define, are of the format

```
#define B0 0
#define B00 0
#define B000 0
#define B0000 0
#define B00000 0
#define B000000 0
#define B0000000 0
#define B00000000 0
#define B1 1
#define B01 1
#define B001 1
#define B0001 1
#define B00001 1
#define B000001 1
#define B0000001 1
#define B00000001 1
...
...
#define B11111110 254
#define B11111111 255
```

It looks strange, yes? The header is defining many different binary style constants for every number between 0 and 255. Why does 0 get so many different constants, while 255 only has one? This allows the programmer to specify zero in as many ways as there are leading zeros in the binary representation of the number zero. This applies to all the numbers, but once you reach 128, there are no more leading zeros, so those values only have a single constant defined.

As I said, a little strange, but it allows you to write code such as

```
DDRD = B11110000;
```

This one line sets PORTD on the AVR microcontroller to have the top 4 bits as OUTPUT and the bottom 4 as INPUT, equivalent to the following Arduino code:

```
pinMode(7, OUTPUT); // Port D, pin 7.
pinMode(6, OUTPUT); // Port D, pin 6.
pinMode(5, OUTPUT); // Port D, pin 5.
pinMode(4, OUTPUT); // Port D, pin 4.
pinMode(3, INPUT); // Port D, pin 3.
pinMode(2, INPUT); // Port D, pin 2.
pinMode(1, INPUT); // Port D, pin 1.
pinMode(0, INPUT); // Port D, pin 0.
```

Tip
Only one line of code? To do all that? Yes, one line of AVR code can correspond to numerous lines of Arduino code. This is another example of how the Arduino makes life easier for the beginner—which of the preceding two code sections is the easier to read and understand?

2.9.5 Header File WCharacter.h

This header file defines a number of *inlined* functions which can be used to determine if a character is numeric, alphanumeric, and so on. This is not specific to the Arduino software and will not be discussed further.

What are inlined functions?

Inlined functions get copied verbatim into the code where they are called. Normally, functions are set up in the executable once and called from many places. Inlining them improves runtime efficiency but at the expense of code size.

Have a look at the header file, and if you are short of space in your device, try to avoid using the functions defined there frequently. If you do have to use them often, then try to do something similar to the code in Listing 2-4.

Listing 2-4 Avoiding inlined code to save space

```
// Define a function to avoid having
// many copies of 'isAlphaNumeric()'
// inlined.
// See 'WCharacter.h' for details.
boolean isAlphaNum(int c) {
    return isAlphaNumeric(int c);
}
```

Then call the `isAlphaNum()` function frequently from your own code, rather than calling the `isAlphaNumeric()` function frequently.

2.9.6 Header File WString.h

The `WString.h` header file defines a C++ class named `String`. I have to admit to not seeing any Arduino code that uses this class, but maybe I haven't been reading enough code. As this isn't specifically Arduino code, even though the class has been written for Arduino, it will not be discussed further.

OK, I lied; `String` is used by the `Serial` class, but deep down. `Serial` inherits from `Stream` which inherits from `Print`, and `Print` uses `String` internally.

Warning
Using this class will seriously increase the size of your sketches and may result in very difficult to diagnose and intermittent runtime errors if too much dynamic memory is allocated—which this class does internally.

2.9.7 Header File HardwareSerial.h

This is the header file that defines the `Serial` interface whereby your Arduino board can talk back to your main computer over the USB cable. There's a lot going on in this file, and it makes interesting reading to see how the Arduino library works.

If your device has less than 1,024 bytes of RAM, then two buffers of 16 characters each are created. One is for serial receiving, and the other is for serial transmission. The buffer sizes are increased to 64 characters if you have more than 1,024 bytes of RAM available. The ATmega328P has 2,048 bytes, so the larger-sized buffers are created.

The buffers are set up as what is known as *circular buffers*. They have a pointer to the first free character for insertions into the buffer and a pointer to the next character to be removed from the buffer. Hopefully, never the twain shall meet, but if they do happen to meet, then the buffer is full, and the sketch will suspend for a while until some data has been removed from the buffer, allowing the new data to be inserted.

Circular buffers are described in some detail at https://en.wikipedia.org/wiki/Circular_buffer.

Also set up in this file are a couple of interrupt routines which are called automatically by the AVR microcontroller whenever there is an empty transmit or a full receive buffer for the built-in USART device. The `Serial` class will, in the case of a transmission, read the next character from the Arduino transmit buffer and write it into the hardware register as appropriate to have it transmitted out via the USART. A similar process takes place for the receive interrupt. The hardware buffers on the AVR microcontroller are a single byte in size.

In the case where the AVR device has more than one USART, the Mega 2560 series, for example, then these are also set up from other header files included by this one. These additional hardware serial devices are not discussed further as the default Arduino board using the ATmega328P only has a single USART, and they are all similar.

2.9.8 Header File USBAPI.h

This is a header specifically for devices which have hardware USB features built in to the device—these being boards based around the ATmega32U4 microcontroller which is onboard the Arduino Leonardo, Pro Micro, Micro, and a few other models. As the ATmega328P doesn't have hardware USB onboard, this file will not be discussed any further.

2.9.9 Header File pins_arduino.h

The version of `pins_arduino.h` that is `#included` is dependent on the Arduino board in use and, thus, the AVR microcontroller in use on that specific board. For the default board, the file is located in `$ARDINST/variants/standard` and uses an ATmega328P, while the Adafruit Gemma board has its `pins_arduino.h` in `$ARDINST/variants/gemma` and uses an ATtiny85 device. The included file sets up the pin assignments for the appropriate device.

The board is chosen by the developer using the IDE's Tools ▹ Board menu option.

It is this header file that defines constants for the analog pins, `A0`, which has the value 14, through `A7` which is defined as 21, for example.

Also created within the program space on the AVR device are a number of small lookup tables that are used to

- Convert an AVR port name (`Px`) to the Data Direction Register for that port (`DDRx`). This is table `port_to_mode_PGM`.
- Convert an AVR port name (`Px`) to the output port register for that port (`PORTx`). This is table `port_to_output_PGM`.
- Convert an AVR port name (`Px`) to the input port register for that port (`PINx`). This is table `port_to_input_PGM`.
- Convert a digital pin number (0 through 13) to the AVR port (`PORTB`, `PORTC`, or `PORTD`). This is table `digital_pin_to_port_PGM`.
- Convert a digital pin number (0 through 13) to the specific pin number (or bit number) on the AVR port (PORTB, PORTC, or PORTD). The entry stored in this table is a bitmask with only one bit set, the bit that corresponds to the pin number. This is table `digital_pin_to_bit_mask_PGM`.
- Convert a digital pin number (0 through 13) to one of the timer outputs on the device (six on the standard Arduino board). This is table `digital_pin_to_timer_PGM` and is used in the `analogWrite()` function for PWM.

2.10 The init() Function

This function is located within the file `$ARDINC/wiring.c`.

At the start of *every* Arduino sketch, the `init()` function is responsible for

- Enabling the global interrupt flag
- Configuring Timer 0 to provide PWM on pins `D5` and `D6` and initiating the `millis()` counter facility by setting up the Timer 0 Overflow Interrupt handler
- Configuring Timer 1 to provide PWM on pins `D9` and `D10`
- Configuring Timer 2 to provide PWM on pins `D3` and `D11`
- Initializing the Analog-to-Digital Converter
- Disabling the USART from pins `D0` and `D1`

2.10.1 Enabling the Global Interrupt Flag

The function `init()` begins by enabling interrupts globally as shown in Listing 2-5. Arduino boards require interrupts to be enabled so that the `millis()` function can begin counting, once the appropriate timer, Timer 0, is configured and started.

Note
In the following walk-through of the source code for the `init()` function, and as with many other code listings in this book, the code that is not relevant to the ATmega328P has been removed to reduce the amount of source code listed and to avoid confusing this author!

Listing 2-5 Setting interrupts on

```
void init()
{
// this needs to be called before setup() or some
// functions won't work there.
sei();                                              (1)
```

(1) This turns on global interrupts. This is required to make functions such as `millis()` and `micros()` work.

The code continues to enable Timer 0 next.

2.10.2 Enabling Timer 0

Timer 0 is used to count the milliseconds which have passed since the sketch began operating after power on, system reset, or after uploading a sketch. It does this by setting up the following for Timer 0:

- The prescaler for Timer 0 is set to divide the 16 MHz system clock by 64 so that every 64 ticks of the system clock, the timer's own clock will tick once and increment the counter value by 1. As this is an eight-bit timer, it can only count from 0 to 255, then roll over, or overflow, to zero again, and so on. The overflow will occur every 256 timer clock ticks which equates to $64 * 256$ system clock ticks.
- The interrupt on Timer 0 overflow is set up and enabled. The interrupt will fire every time the timer's value overflows from 255 to 0. The Timer 0 Overflow Interrupt will update the `millis()` counter once every 256 timer clock ticks. This is calculated as

$$1/(CPU\ Frequency/prescaler) * Ticks\ to\ overflow$$

which, when we substitute the actual values, is

$$1/\,(F\_CPU/64) * 256$$
$$= 1/\,(16,000,000/64) * 256$$

$$
\begin{aligned}
&= 1/250,000 * 256\\
&= 4\,uS * 256\\
&= 1,024\,uS\\
&= 1.024\,mS
\end{aligned}
$$

The interrupt routine increments the millisecond counter and takes account of the additional 24 microseconds and will adjust the `millis()` result to account for them whenever they accumulate enough to add an extra millisecond to the total.

Timer 0 also provides eight-bit PWM, Pulse Width Modulation, for `analogWrite()` on pins `D5` and `D6`.

The `init()` function code walk-through continues in Listing 2-6.

Listing 2-6 Timer 0 configuration

```
    // on the ATmega168, timer 0 is also used for fast
    // hardware PWM (using phase-correct PWM would mean
    // that timer 0 overflowed half as often resulting in
    // different millis() behavior on the ATmega8 and
    // ATmega168)

#if defined(TCCR0A) && defined(WGM01)
    sbi(TCCR0A, WGM01);                                    (1)
    sbi(TCCR0A, WGM00);
#endif

// set timer 0 prescale factor to 64
...
#elif defined(TCCR0B) && defined(CS01) && defined(CS00)

    // this combination is for the standard
    // 168/328/1280/2560.
    sbi(TCCR0B, CS01);                                     (2)
    sbi(TCCR0B, CS00);
    ...
#else
    #error Timer 0 prescale factor 64 not set correctly
#endif

    // enable timer 0 overflow interrupt
    ...
#elif defined(TIMSK0) && defined(TOIE0)
    sbi(TIMSK0, TOIE0);                                    (3)
#else
    #error Timer 0 overflow interrupt not set correctly
#endif
```

(1) Setting these two bits in the TCCR0A register ensures that the PWM waveform generator is running in Fast Hardware PWM mode, instead of Phase Correct PWM mode, which would interfere with the timer for the millis() function.
(2) Setting these two bits in register TCCR0B sets the timer clock to be the system clock divided by 64. That equates to 16 MHz for the system clock, divided down to 250 MHz, or one tick of the timer clock for every 64 ticks of the system clock.
(3) Setting this bit in the TIMSK0 register enables the Timer 0 Overflow Interrupt. Now every time the timer goes from 255 to 0, the interrupt routine will be called to accumulate counts for millis() and micros().

2.10.3 Timer 0 Overflow Interrupt

The Timer 0 Overflow Interrupt is used to update the millis() count. It does this every 1.024 milliseconds, and, as this is slightly over 1 millisecond, it accumulates these extra fractions, and when there are enough accumulated, the millis() count gets incremented by an extra leap millisecond. This takes place roughly every 42 interrupt handler executions—it's actually every 41.666 (recurring) executions, but you cannot have a fraction of an execution!

The code to do all this is shown in Listing 2-7, taken from $ARDINC/wiring.c. Listing 2-7 is separate from the code in the init() function. The remainder of the init() function walk-through follows in the next section.

Listing 2-7 Timer 0 overflow interrupt handler

```
#if defined(TIM0_OVF_vect)
ISR(TIM0_OVF_vect)
#else
ISR(TIMER0_OVF_vect)
#endif
{
    // copy these to local variables so they can be
    // stored in registers (volatile variables must be
    // read from memory on every access)
    unsigned long m = timer0_millis;                (1)
    unsigned char f = timer0_fract;

    m += MILLIS_INC;                                (2)
    f += FRACT_INC;
    if (f >= FRACT_MAX) {
        f -= FRACT_MAX;
        m += 1;
    }
    timer0_fract = f;                               (3)
    timer0_millis = m;
    timer0_overflow_count++;                        (4)
}
```

(1) The current values of the variables `timer0_millis` and `timer0_fract` are copied locally from memory (Static RAM) so that they can be used in registers for faster processing.
(2) The current `timer0_millis count`, in `m`, is incremented by `MILLIS_INC`. The current accumulated fractions of a millisecond, `timer0_fract`, used locally in variable `f`, is incremented by `FRACT_INC`. If `f` is then larger than `FRACT_MAX`, then an extra "leap" millisecond is accumulated and the counts adjusted accordingly.
(3) The new values are copied back to the original two variables.
(4) A counter, `timer0_overflow_count`, keeps track of the number of times the ISR (interrupt service routine) has been fired. This counter is used in the `millis()` function which is itself used in the `delay()` function.

What are `MILLIS_INC`, `FRACT_INC`, and `FRACT_MAX` I hear you ask?
These are defined in `$ARDINC/wiring.c`, and an extract is shown in Listing 2-8.

Listing 2-8 Variables used in counting millis

```
// the whole number of milliseconds per timer0 overflow
#define MILLIS_INC (MICROSECONDS_PER_TIMER0_OVERFLOW / 1000)

// the fractional number of milliseconds per timer0 overflow.
// we shift right by three to fit these numbers into a byte.
// For the clock speeds we care about -- 8 and 16 MHz -- this
// doesn't lose precision.)

#define FRACT_INC
    ((MICROSECONDS_PER_TIMER0_OVERFLOW % 1000) >> 3)

#define FRACT_MAX (1000 >> 3)
```

The other helper definitions we need here are as follows; the first is also defined in `$ARDINC/wiring.c`:

```
// the prescaler is set so that timer0 ticks every 64
// clock cycles, and the the overflow handler is called
// every 256 ticks.
#define MICROSECONDS_PER_TIMER0_OVERFLOW \
    (clockCyclesToMicroseconds(64 * 256))
```

and from `$ARDINC/Arduino.h`, we have

```
#define clockCyclesToMicroseconds(a) \
     ( (a) / clockCyclesPerMicrosecond() )
```

and also

```
#define clockCyclesPerMicrosecond() ( F_CPU / 1000000L )
```

So, working backward, we see that `clockCyclesPerMicrosecond` is 16,000,000/1,000,000 or 16. From that, we can then see that `MICROSECONDS_PER_TIMER0_OVERFLOW` is (64 ∗ 256)/16, which gives us 1,024.

This then allows `MILLIS_INC` to be calculated as 1,024/1,000 which is a solitary one, as this is integer division, not floating point. And, finally, `FRACT_INC` is ((16,384/16)%1,000) >> 3, or 24/8 which gives us 3.

`FRACT_MAX` is easy, it's effectively 1000/8 or 125.

So, every 256 Timer 0 clock ticks, we increment the number of millis by 1, add 3 to the fractions accumulator, and if that is more than 125, we add an extra 1 to millis and reduce the fractions accumulator by 125, thus holding on to any spare fractions. Eventually, these will add up and generate another millisecond.

If you are wondering why we add 3 and check for 125, then consider that there are 24 microseconds spare each time through the interrupt handler; that's 41.666 (recurring) to gain an extra millisecond. 125 /3 is exactly the same value, 41.666 (recurring)—so it works out the same.

Note
Is adding 3 and checking against 125 more efficient than adding 24 and checking against 1000? Yes, indeed, the former method fits a byte, while the latter requires a 16-bit value, and the ATmega328P is an 8-bit device without a 16-bit compare instruction.

2.10.4 Configuring Timer 1 and Timer 2

On the ATmega328P, Timer 1 is a 16-bit timer; however, the Arduino system sets it up so that it appears as an 8-bit timer which makes it similar to Timer 0 and Timer 2.

Timers 1 and 2 are used to provide PWM on four of the six pins that are PWM enabled on an ATmega328P.

Both timers have their prescaler set to divide the system clock by 64 and are set up in 8-bit Phase Correct PWM mode.

Timer 1 provides PWM on pins `D9` and `D10`, while Timer 2 provides PWM on pins `D3` and `D11`. The PWM feature of the timers is used by the `analogWrite()` function.

The `init()` function source code continues in Listing 2-9, where it configures Timer 1.

Listing 2-9 Timer 1 configuration

```
    // timers 1 and 2 are used for phase-correct
    // hardware PWM. this is better for motors as it
    // ensures an even waveform
    // note, however, that fast PWM mode can achieve a
    // frequency of up 8 MHz (with a 16 MHz clock) at
    // 50% duty cycle
#if defined(TCCR1B) && defined(CS11) && defined(CS10)
    TCCR1B = 0;                                          (1)

    // set timer 1 prescale factor to 64
    sbi(TCCR1B, CS11);                                   (2)
```

```
#if F_CPU >= 8000000L
    sbi(TCCR1B, CS10);                                      (3)
#endif
    ...
#endif

    // put timer 1 in 8-bit phase correct PWM mode
#if defined(TCCR1A) && defined(WGM10)
    sbi(TCCR1A, WGM10);                                     (4)
#endif
```

(1) This shouldn't be necessary as `init()` is called at the start of a sketch, after a reset or on power on, so the default for register `TCCR1B` is zero anyway. However, sometimes, it's best to be explicit.
(2) Setting only the `CS11` bit sets the timer's prescaler to divide by eight, which is fine for slow system clock speeds. This would give a standard Arduino board a 2 MHz timer clock speed. A tad excessive perhaps!
(3) For faster clock speeds, setting `CS10`, plus `CS11` earlier, finally sets the prescaler to divide by 64, giving the required 250 KHz timer clock speed.
(4) The `WGM10` bit, in the `TCCR1A` register, sets the PWM waveform generator to run in eight-bit Phase Correct PWM mode.

After configuring Timer 1, the next part of the `init()` function sets up Timer 2 as shown in Listing 2-10.

Listing 2-10 Timer 2 configuration

```
    // set timer 2 prescale factor to 64

#if defined(TCCR2) && defined(CS22)
    ...
#elif defined(TCCR2B) && defined(CS22)
    sbi(TCCR2B, CS22);                                      (1)
//#else
// Timer 2 not finished (may not be present on this CPU)
#endif

// configure timer 2 for phase correct PWM (8-bit)
#if defined(TCCR2) && defined(WGM20)
    ...
#elif defined(TCCR2A) && defined(WGM20)
    sbi(TCCR2A, WGM20);                                     (2)
//#else
// Timer 2 not finished (may not be present on this CPU)
#endif
    ...
```

(1) Setting bit `CS22` in register `TCCR2B` sets the timer's prescaler to divide the system clock by 64. This results in a 250 KHz timer clock.
(2) Setting the `WGM20` bit, in the `TCCR2A` register, sets the PWM waveform generator to run in eight-bit Phase Correct PWM mode.

The function continues, now that all three timers are configured, to set up the Analog-to-Digital Converter (ADC) so that the Arduino `analogRead()` function will work.

2.10.5 Initializing the Analog-to-Digital Converter

According to the data sheet for the ATmega328P, the Analog-to-Digital Converter, the ADC, runs best, and most accurately, when it is running at a speed between 50 and 200 KHz. The system clock on the microcontroller is running at 16 MHz, so is a little on the speedy side.

In order to get the ADC into a valid speed range, it has its prescaler set to divide the system clock by 128. This puts the speed at 125 KHz, which is within the desired range specified by the data sheet.

The ADC is then enabled, as shown in Listing 2-11, which is a continuation of the `init()` function.

Listing 2-11 ADC configuration

```
#if defined(ADCSRA)
    // set a2d prescaler so we are inside the
    // desired 50-200 KHz range.

    #if F_CPU >= 16000000 // 16 MHz / 128 = 125 KHz
        sbi(ADCSRA, ADPS2);                                (1)
        sbi(ADCSRA, ADPS1);
        sbi(ADCSRA, ADPS0);

        // Code removed - not relevant.

    #endif

    // enable a2d conversions
    sbi(ADCSRA, ADEN);                                     (2)
#endif
```

(1) The system clock needs to be divided down to obtain an ADC clock speed in the range 50 to 200 KHz, which, according to the data sheet, is the optimal clock range for the ADC. For the standard Arduino boards, this requires a divisor of 128 to get the 16 MHz system clock into this range. The resulting ADC clock speed is 125 KHz, which is well within the requirement.
(2) Setting the `ADEN` bit in the `ADCSRA` register ensures that the ADC is enabled. It will not start converting until it is told to do so by `analogRead()`.

The preceding code implies that even if you don't want the ADC in your sketches, it is active and consuming additional power that might be better used keeping your batteries from running down!

If you are sure that you don't need or want `analogRead()` in your sketch, and you are running on batteries, then perhaps adding the following lines to your `setup()` function could help:

```
#include <wiring_private.h>
...
cbi(ADCSRA, ADEN);
```

This disables the ADC. If you also add

```
sbi(PRR, PRADC);
```

then you will also stop power reaching the ADC clock, saving a few more microamps, alternatively:

```
#include <avr/power.h>
...
void setup()
{
    ...
    power_adc_disable();
}
```

which uses the AVRLib facility to turn off the power to the ADC clock and, in my opinion, is a lot more readable, and understandable, than the preceding code.

2.10.6 Disabling the USART

The final task for the `init()` function is to disable the Universal Synchronous/Asynchronous Receiver/Transmitter or USART for short.

This is left attached to Arduino pins `D0` and `D1` by the bootloader, and the two pins used need to be disconnected so that they can be reused for `digitalRead()` and/or `digitalWrite()` in sketches. On the ATmega328P, these *digital pins* are the *physical pins* 2 and 3.

If the USART is required for the Serial Monitor tool, for example, then the two USART pins will be reconnected by a call to `Serial.begin()` in the sketch. Listing 2-12 shows the pins being disconnected from the USART.

Listing 2-12 USART configuration

```
    // the bootloader connects pins 0 and 1 to the
    // USART; disconnect them here so they can be used
    // as normal digital i/o; they will be reconnected
    // in Serial.begin()
#if defined(UCSRB)
    ...
#elif defined(UCSR0B)
    UCSR0B = 0;
#endif
```

```
} // End of init().
```

This concludes the initialization that occurs at the start of every sketch and the walk-through of the `init()` function's source code.

Arduino Language Reference

3

In this chapter, I look at the Arduino-specific features of the C/C++ language which relate to the AVR microcontroller and how it operates, as opposed to looking at the C/C++ language in general.

Warning
This chapter and the following one are long chapters—my apologies for that. I would advise that you do not try to get through them both in one sitting. Take a break every so often and go and do something with your Arduino—to take your mind off it! Sorry!

3.1 What Are We Looking At?

The features of the Arduino that I will be covering in the next two chapters of the book are those that the Arduino Reference site—www.arduino.cc/reference/en/—refers to as

Digital I/O — All the digital I/O functions from `$ARDINC/wiring_ digital.c`:

- `pinMode()`
- `digitalRead()`
- `digitalWrite()`

Analog I/O — The analog functions in `$ARDINC/wiring_analog.c`:

- `analogReference()`
- `analogRead()`
- `analogWrite()`

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_3

Advanced I/O — The various functions handling note generation and pulse measurements in `$ARDINC/wiring_shift.c`:

- `tone()`
- `noTone()`
- `pulseIn()`
- `pulseInLong()`
- `shiftIn()`

Time — The timing and delay functions from `$ARDINC/wiring.c`:

- `delay()`
- `delaymicroseconds()`
- `micros()`
- `millis()`

Interrupt — Interrupt handling language features from `$ARDINC/ WInterrupts.c`:

- `interrupts()`
- `noInterrupts()`
- `attachInterrupt()`
- `detachInterrupt()`

Bit manipulation — Functions as found in the header files `$ARDINC/ Arduino.h` and `$ARDINC/wiring_private.h` to manipulate bits in variables and registers

I will not be discussing the general C/C++ language functions, only those related to the Arduino Language. For the general ones, you should arm yourself with a good book on the subject.

Where possible, each function listed earlier will be listed here in full, then dissected and explained. If there's any foibles to be aware of, those will be discussed too. However, as the Arduino software for AVR microcontrollers covers many different types of AVR microcontroller, I shall restrict the discussion of the software to that pertaining to the ATmega328P, and I will not be covering other microcontrollers—unless absolutely necessary.

Finally, in the next chapter, I will discuss the various C++ classes declared by the Arduino software that are included in almost every sketch. These are the `Print`, `Printable`, `Stream`, `HardwareSerial`, and `String` classes—although I don't have much to say on the latter, apart from *avoid*!

Read on...

3.2 Digital Input/Output

This section takes a look at the functions which carry out digital input and output within the Arduino Language. These functions are `pinMode()` to set the pin's mode and direction; `digitalRead()` to read the voltage state, `HIGH` or `LOW`, on a pin; and `digitalWrite()` to set the pin's voltage `HIGH` or `LOW`.

3.2.1 Function pinMode()

In Arduino sketches, you will often see code such as that shown in Listing 3-1.

Listing 3-1 Example pinMode() usage

```
#define switchPin 2;
#define sensorPin 3;

void setup()
{
    pinMode(LED_BUILTIN, OUTPUT);
    pinMode(switchPin, INPUT_PULLUP);
    pinMode(sensorPin, INPUT);
    ...
}
```

The `pinMode()` function sets the direction of a specific pin so that it can be used for input or output depending on what purpose the project is designed for. The three modes shown in the example code in Listing 3-1 are the only three that are available. These allow that particular pin to be used for

- Input, where the pin state is determined by the voltage applied to it. The pin's state would be read using `digitalRead()` and will result in a returned value of `HIGH` or `LOW` according to whatever voltage is currently being applied to the pin by external devices or components.
- Input with the internal pull-up resistor enabled, where the pin is used again for input, but the default state is pulled to `HIGH` when nothing else attached to the pin is attempting to pull it `LOW`. Using pull-up resistors in this way can be done internally, as with the `pinMode(switchPin, INPUT_PULLUP)` example in Listing 3-1, or externally where there would be a resistor of about 10 KΩ connected to the pin and to 5 V or 3.3 V depending on your Arduino board.
- Output, where the pin state is set to `HIGH` or `LOW` by a call to `digitalWrite()`. Output pins when set `HIGH` or `LOW` by a call to `digitalWrite()` will see the supply voltage (5 V or 3.3 V) on the pin if set `HIGH` or 0 V if set `LOW`.

Tip
A pin may be configured as an `INPUT` pin, but then written to, as if it was an `OUTPUT` pin, with `digitalWrite()` to set it `HIGH`. This will enable the internal pull-up resistor which means that a `digitalRead()` on the pin will now see a `HIGH` unless the pin is being pulled to ground by some external influence. This is exactly how the `INPUT_PULLUP` setting for `pinMode()` works.

When reading or writing digital pins, the pin can take one of two different values, which are defined in `$ARDINC/Arduino.h` as `HIGH` and `LOW`, but what does this mean in relation to the voltage applied to, or seen on, the pin itself?

For Arduino boards running on a 5 V supply, a call to `digitalRead()` will return `HIGH` if the voltage on the appropriate pin is 3 V or higher. A `LOW` will be returned if the voltage on the pin is less than 1.5 V.

For Arduino boards running on a 3.3 V supply, a call to `digitalRead()` will return `HIGH` if the voltage on the appropriate pin is 2 V or higher. A `LOW` will be returned if the voltage on the pin is less than 1 V.

What about voltages in between? These are considered to be *floating* voltages, and the call to `digitalRead()` could return either a `HIGH` or a `LOW` depending on other circumstances, and not necessarily the same result each time it is called for the same voltage. For this reason, it is best to avoid having input pins floating—so either use pull-up resistors (internal or external) or, alternatively, pull-down resistors, which are external only.

Warning
Floating pins are a really bad thing to have. A pin that is not electrically connected to supply or ground is a problem waiting to happen. How does your code see the value on the pin? It could be seen as `HIGH` sometimes or `LOW`, and the code thinks that it is a valid reading; it is not.

The value seen on the pin may be affected by many things—temperature, stray capacitance on the board, induced currents from external sources, or even you walking past. *Never* leave a pin floating—unless the data sheet says to do so.

It may not be a major problem to have floating pins on a project designed to illuminate an LED from time to time, but for a high-powered laser cutter, for example, you really don't want the laser turning on because the Arduino board thought a button had been pressed!

The file `$ARDINC/Wiring_digital.c` is where the source code for the digital functions `pinMode()`, `digitalRead()`, and `digitalWrite()` can be found. Additionally, there is one other function in this file, but it can only be called from the three functions listed. This helper function is `turnOffPWM()`, which is not discussed further, is declared static, and is there simply to turn off any PWM on a pin that is about to be used for `digitalRead()` or `digitalWrite()` purposes.

The `pinMode()` function takes two input parameters, a pin number and a mode, and sets the requested pin to the mode given. The modes are as discussed earlier, while the pin number is just a number corresponding to the actual pin required.

You may not be aware, but the eight analog pins `A0` through `A7` on your Arduino board can also be used for digital I/O. They are numbered from `D14`, for pin `A0`, to `D21` for pin `A7`. So, a call to `digitalWrite(14, HIGH)` will set pin `A0` to `HIGH`. This is useful when you need more digital pins than apparently supplied on the Arduino board.

Hang on! What do I mean `A7`? Surely I mean `A5`?

Some Arduino boards have been built with the surface mount versions of the ATmega328 device, the ATmega328PAU. These surface mount devices have a couple of extra pins connected to the ADC input, these being `A6` and `A7` in Arduino speak. Many clone boards have also added two extra connectors to allow the board to use these two additional pins, while some have not.

If you have an Arduino Nano, for example, then look carefully at the pin labels and you will see `A6` and `A7`. These extra pins are not present on the 28-pin through hole style ATmega328P devices.

Sometimes, you might see code referencing an additional ADC input pin, pin `A8`, which is an internal connection for the temperature sensor built in to the AVR microcontroller itself. You can read this input and get an idea of how hot the AVR microcontroller is running. Sadly, the Arduino Language does not make this visible using `analogRead()`. See the sketch in Appendix E for details of how to use this internal feature.

Getting back to `pinMode()`, we need to be aware first of all that the Arduino pin numbering system is completely different from that used by Atmel, now Microchip, that manufactures the AVR devices. What we call `D1` is known to Microchip as `PD1`, and the built-in LED on Arduino pin `D13` is attached to the ATmega328P's `PB5` pin.

It helps if there's a pinout diagram for our specific AVR microcontroller. Look at Figure 3-1 which shows the pin functions and names for an ATmega328P.

The Arduino pin numbers are easily enough recognized as they are listed by name in the two columns labeled *Arduino*, and there you will see names like `D0` or `A5` and so on. The Arduino Language has given these names to the various pins that are accessible using that language.

Atmel, when they designed the ATmega328P, named the pins differently, and the Atmel pin names can be seen in the *AVR* columns. Here, you see names like `PB2` or `PD4` and so on. These are the actual pins that are used for digital input and output or analog input.

On an AVR microcontroller, pins are arranged in banks of up to eight pins, which happily is the same number of bits in a byte. On the ATmega328P, there are three banks of pins; these are `B`, `C`, and `D`. In order to use `pinMode()` on an Arduino pin, you need three things:

- The bank's Data Direction Register, or DDR
- The bank's Pin Input Register, or PIN
- The bank's Pin Output Register, or PORT

The ATmega328P has three separate banks of pins, so there are a total of nine different registers to consider.

On the ATmega328P, the nine different registers are

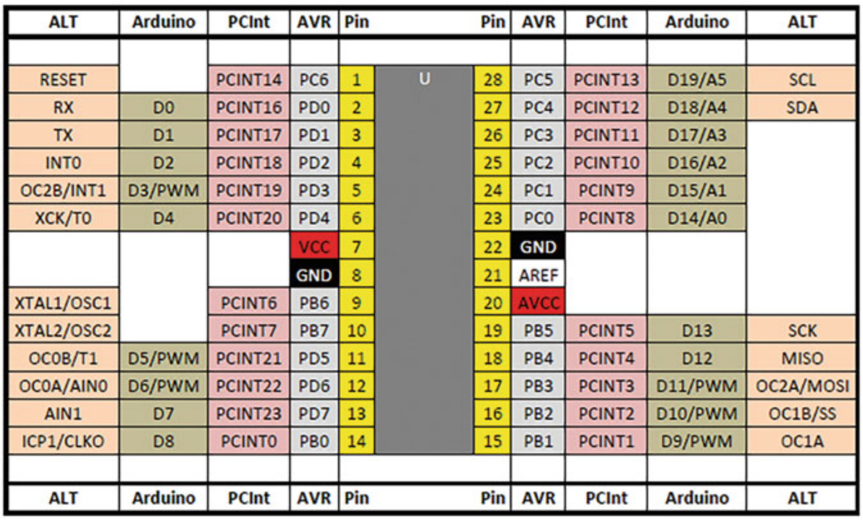
- `DDRB`, `DDRC`, and `DDRD`
- `PORTB`, `PORTC`, and `PORTD`
- `PINB`, `PINC`, and `PIND`

Note
The ATmega328P has only three banks: B, C, and D. Other AVR microcontrollers, such as the Mega 3560, have many more banks of pins.

On the pinout image, Figure 3-1, when a pin is named `PCn`, where "n" is a number, then that particular pin belongs to bank C and uses registers `DDRC`, `PORTC`, and `PINC`.

The `pinMode()` function, among others, has to convert between the Arduino pin naming convention and the AVR's own names. If, for example, pin `D2` is being set to `OUTPUT`, the `pinMode()` function needs to convert `D2` to `DDRD`, `PORTD`, and `PIND2` so that manipulating that pin in Arduino code manipulates the `PD2` pin on the ATmega328P.

Getting from `D2`, which is nothing more than the value two, to a PORT, PIN, and DDR is done with the help of a few small data tables, set up in `$ARDINST/variants/standard/pins_arduino.h`.



ALT	Arduino	PCInt	AVR	Pin		Pin	AVR	PCInt	Arduino	ALT
RESET		PCINT14	PC6	1	U	28	PC5	PCINT13	D19/A5	SCL
RX	D0	PCINT16	PD0	2		27	PC4	PCINT12	D18/A4	SDA
TX	D1	PCINT17	PD1	3		26	PC3	PCINT11	D17/A3	
INT0	D2	PCINT18	PD2	4		25	PC2	PCINT10	D16/A2	
OC2B/INT1	D3/PWM	PCINT19	PD3	5		24	PC1	PCINT9	D15/A1	
XCK/T0	D4	PCINT20	PD4	6		23	PC0	PCINT8	D14/A0	
			VCC	7		22	GND			
			GND	8		21	AREF			
XTAL1/OSC1		PCINT6	PB6	9		20	AVCC			
XTAL2/OSC2		PCINT7	PB7	10		19	PB5	PCINT5	D13	SCK
OC0B/T1	D5/PWM	PCINT21	PD5	11		18	PB4	PCINT4	D12	MISO
OC0A/AIN0	D6/PWM	PCINT22	PD6	12		17	PB3	PCINT3	D11/PWM	OC2A/MOSI
AIN1	D7	PCINT23	PD7	13		16	PB2	PCINT2	D10/PWM	OC1B/SS
ICP1/CLKO	D8	PCINT0	PB0	14		15	PB1	PCINT1	D9/PWM	OC1A
ALT	Arduino	PCInt	AVR	Pin		Pin	AVR	PCInt	Arduino	ALT

Figure 3-1 Pin names on the ATmega328P

Listing 3-2 is the code which makes up the `pinMode()` function which can be found in `$ARDINC/wiring_digital.c`.

Listing 3-2 The pinMode() function

```
void pinMode(uint8_t pin, uint8_t mode)
{
    uint8_t bit = digitalPinToBitMask(pin);            (1)
    uint8_t port = digitalPinToPort(pin);              (2)
    volatile uint8_t *reg, *out;

    if (port == NOT_A_PIN) return;                     (3)
        // JWS: can I let the optimizer do this?
        reg = portModeRegister(port);                  (4)
        out = portOutputRegister(port);

        if (mode == INPUT) {                           (5)
            uint8_t oldSREG = SREG;
            cli();
            *reg &= ~bit;
            *out &= ~bit;
            SREG = oldSREG;
        } else if (mode == INPUT_PULLUP) {             (6)
            uint8_t oldSREG = SREG;
            cli();
            *reg &= ~bit;
            *out |= bit;
            SREG = oldSREG;
        } else {                                       (7)
```

```
            uint8_t oldSREG = SREG;
            cli();
            *reg |= bit;
            SREG = oldSREG;
        }
    }
```

(1) If we continue the preceding example with D2, then this pin has the value two. This call to digitalPinToBitMask() converts D2 into an eight-bit value in which only bit 2 is set. This is therefore the value four as bit 2 in a byte indicates whether there are any fours present in the value. The bitmask returned will look like 0b0000 0100 with only bit 2 set.
(2) D2's value, two, is used again in a call to digitalPinToPort() which returns a value known as PD from the table digital_pin_to_port_PGM. PD is defined in Arduino.h to be the value four. We now have the bitmask in bit and the port in reg. These are still not AVR microcontroller register names yet, they are still just numbers—both of them are four in this example.
(3) The port is validated, and if it is NOT_A_PIN, which has the value -1, we exit from the function.
(4) The value for our port, four, is then converted to a DDR register and a PORT using portModeRegister() and portOutputRegister(). These two functions read the tables port_to_mode_PGM and port_to_output_PGM and return pointers to the internal registers named DDRx and PORTx for the appropriate pin. In this example, these will be DDRD and PORTD. At this stage, pointers to the desired DDR and PORT registers are available in reg and out and can be manipulated to set a single pin to output mode.
(5) If the requested mode is INPUT, we have to clear the appropriate bit in the DDR to configure an input pin, and as pull-up has not been requested, the appropriate bit in the PORT register is also cleared to turn off the input pull-up resistor for the pin. The current state of the status register is saved, and interrupts are turned off for the duration of the preceding changes. When the status register is restored, interrupts are reset to how they were before being disabled.
(6) If the requested mode is INPUT_PULLUP, we have to clear the appropriate bit in the DDR to configure an input pin as before, and as pull-up has been requested, the appropriate bit in the PORT register is set to turn on the input pull-up resistor for the pin. As before, the current state of the status register is saved, and interrupts are turned off for the duration of the preceding changes. When the status register is restored, interrupts are reset to how they were before being disabled.
(7) If the requested mode is OUTPUT, we have to set the appropriate bit in the DDR to configure an output pin. There is no pull-up on output pins. As before, when these changes are being made, the current state of the status register is saved, and interrupts are turned off. When the status register is restored, interrupts are reset to their previous setting.

In the AVR microcontroller, setting a bit in the DDRx register configures the appropriate pin to be an output; while clearing a bit configures the pin to input. Input is the default when the AVR microcontroller is reset or powered on.

Tip
Even though the default mode for a pin is INPUT in Arduino code, it is always beneficial to ensure that you explicitly set or clear the pin's bit in the DDR register. It isn't mandatory, but it can help make your code more readable and self-documenting.

As the example needs to set PORTD, bit 2, to output, then a one is required in the *third* bit of the bitmask—remember, bits number from zero—and that is all. The code line `*reg |= bit;` does exactly that; it takes the bitmask 0b0000 0100 and ORs it with whatever is currently in the register DDRD. This sets the pin to output, as required, and does not change the direction of any other pins on PORTD.

Had the mode requested been INPUT, then bit 2 in the DDRD register would need to be set to zero. The code `*reg &= ~bit;` does this by inverting the bitmask from 0b0000 0100 to 0b1111 1011 and then ANDing that with the current contents on the DDRD register. That would change only the third bit to a zero and would not affect any other pin. `*out &= ~bit;` then ensures that the pull-up resistor is disabled for this pin.

If the mode was INPUT_PULLUP, then the `*out |= bit;` code makes sure that the pin is set to have its internal pull-up resistor enabled by setting the pin's bit in the PORTD register.

3.2.2 Function digitalRead()

Once a pin has been set for INPUT with `pinMode()`, then you can read the voltage on that pin with the `digitalRead()` function and change the behavior of your sketch according to the result obtained. The function will return either HIGH or LOW according to the voltage on the pin at the time of the function call.

Listing 3-3 shows the source code for the `digitalRead()` function which can be found in `$ARDINC/wiring_digital.c`.

Listing 3-3 The digitalRead() function

```
int digitalRead(uint8_t pin)
{
    uint8_t timer = digitalPinToTimer(pin);               (1)
    uint8_t bit = digitalPinToBitMask(pin);               (2)
    uint8_t port = digitalPinToPort(pin);                 (3)

    if (port == NOT_A_PIN) return LOW;

      // If the pin supports PWM output, we need to turn it
      // off before getting a digital reading.

    if (timer != NOT_ON_TIMER) turnOffPWM(timer);         (4)

    if (*portInputRegister(port) & bit) return HIGH;      (5)

    return LOW; (6)
}
```

(1) Convert the pin's number to a timer number. This will be Timer 0, Timer 1, or Timer 2. Timers are attached to the pins that we can use `analogWrite()` upon. This is required as any pin which can be used for `analogWrite()` may be set to a value which is not a HIGH and not a LOW—a floating value in other words—and we need to avoid floating values. The timer will be disabled at (4).

(2) The pin's number is converted to an eight-bit value where the only bit set will correspond to this pin's position in its PIN register. Given the `D2` example from earlier, this would be a bitmask of 0b0000 0100 with only bit 2 set.
(3) The pin number is then converted to the correct PIN register.
(4) In order to read a digital value, `LOW` or `HIGH`, the pin should not be carrying out PWM. If the pin is one of the six that can be used with `analogWrite()`, then its ability to do so is temporarily disabled.
(5) The correct PIN register is read and ANDed with the pin's bitmask. If the result of the AND operation leaves the pin's bit set in the PIN register value, then `HIGH` is returned.
(6) The pin must be at GND potential, so return a value of `LOW`.

Note
Timers—or, more correctly, timer/counters—are internal hardware features of the ATmega328P. These will be discussed in great detail in Chapter 8, along with many other useful features of the AVR microcontroller. What follows here is a brief discussion with only as much information as necessary to help understand the `digitalRead()` and `digitalWrite()` functions.

The timers in the ATmega328P are named Timer 0, Timer 1, and Timer 2. As I have previously described, Timer 0 is used to ensure that the `millis()` count is incremented correctly (see Section 2.10 for details). All three timers are used to provide Pulse Width Modulation (PWM) facilities, used to implement the `analogWrite()` function, on two pins each. If there is a call to `digitalWrite()` for pins `D3`, `D5`–`D6`, or `D9`–`D11`, then the PWM must be turned off. This is done by finding out if the pin in question is connected to a timer and, if so, calling `turnOffPWM()` for that particular timer.

The table `digital_pin_to_timer_PGM`, which is defined in `$ARDINST/variants/standard/pins_arduino.h`, is used to convert the pin number to a specific timer.

As with `pinMode()`, the port and bitmask are worked out from the two tables set up in `$ARDINST/variants/standard/pins_arduino.h`, and the port name (`PD`, for example) is converted to an actual PIN register, and the current value of that register is read. To continue the `D2` example, this would be `PIND`, and the bitmask would be 0b0000 0100, where only bit 2 is set.

The PIN registers are connected to the physical pins on the AVR microcontroller, and reading that register returns an eight-bit value where any external pin connected to a high enough voltage will be set and the others will be cleared, if they are seeing a low enough voltage. Floating pins, always a bad idea, will return a fairly random value in the bit, which cannot be relied upon.

As `digitalRead()` is only interested in one single pin's value, all the other bits are masked out by ANDing the returned value with the bitmask holding the correct pin. The function returns the result according to whether or not the bit in the PINx register was set or cleared.

3.2.3 Function digitalWrite()

Once a pin has been set for `OUTPUT` with `pinMode()`, then you can set the voltage on that pin with the `digitalWrite()` function and change the behavior of your project by lighting up LEDs or activating relays and so on.

Listing 3-4 shows the source code for the `digitalWrite()` function which is found in `$ARDINC/wiring_digital.c`.

Listing 3-4 The digitalWrite() function

```
void digitalWrite(uint8_t pin, uint8_t val)
{
    uint8_t timer = digitalPinToTimer(pin);                   (1)
    uint8_t bit = digitalPinToBitMask(pin);                   (2)
    uint8_t port = digitalPinToPort(pin);                     (3)
    volatile uint8_t *out;

    if (port == NOT_A_PIN) return;                            (4)

    // If the pin supports PWM output, we need to turn it
    // off before doing a digital write.
    if (timer != NOT_ON_TIMER) turnOffPWM(timer);             (5)

    out = portOutputRegister(port);                           (6)

    uint8_t oldSREG = SREG;                                   (7)
    cli();

    if (val == LOW) {                                         (8)
        *out &= ~bit;
    } else {
        *out |= bit;
    }

    SREG = oldSREG;                                           (9)
}
```

(1) The pin number is converted to a timer number. As with `digitalRead()`, this is required in case the pin is capable of being used with the `analogWrite()` function call.
(2) The pin number is also converted to an eight-bit value where the only bit set will correspond to this pin's position in its PORT register.
(3) The pin number is converted to a port number.
(4) If the port number is discovered to be invalid, `digitalWrite()` will quietly exit without changing the requested pin and without error. Your sketch will be none the wiser!
(5) If the requested pin supports PWM, then the pin has PWM turned off.
(6) The port number returned is converted to an actual PORT register address and stored in a pointer variable, `out`.
(7) The current value of the status register is saved to preserve the current state of global interrupts. Interrupts are then disabled globally. This will affect the status register and stops `millis()` from being accumulated.
(8) If the value to be written to the pin is `LOW`, then the appropriate bit in the PORTx register is cleared; otherwise, it is set. This turns the physical pin on the AVR microcontroller `LOW` or `HIGH` as appropriate.
(9) The status register is restored which restores the previous state of the global interrupts.

As previously mentioned, all three timers available in the ATmega328P are used to provide PWM facilities on two pins each. If there is a call to `digitalWrite()` for pins `D3`, `D5`–`D6`, or `D9`–`D11`, then the PWM on the requested pin must be turned off. This is done by finding out if the pin in question is connected to a timer and, if so, calling `turnOffPWM()` for the particular timer.

As with `digitalRead()`, the port and bitmask are worked out from the two tables set up in `$ARDINST/variants/standard/pins_arduino.h`, and the port name (PD, for example) is converted to an actual PORTx register. To continue the `D2` example, this would be `PORTD`, and the bitmask would be 0b0000 0100 in binary, where only bit 2 is set.

The PORTx registers are, like the PINx registers, connected to the physical pins on the AVR microcontroller, and writing to that register will cause the voltage on the physical pin to change to supply or ground potential, depending on whether the bit in the PORTx register is a one or a zero.

As `digitalWrite()` is only interested in setting a single pin's value, all the other bits are masked out by ANDing or ORing the bitmask holding the correct pin with the current contents of the `PORTD` register; this affects only the bit that is set in the bitmask, and none of the other pins change. AND is used to clear the bit, while OR is used to set it.

3.3 Analog Input/Output

This section takes a look at the functions which carry out analog input and output within the Arduino Language. These functions are `analogReference()` to set the reference voltage for the analog circuitry in the ATmega328P; `analogRead()` to read a voltage between 0 V and the reference voltage on a pin; and `analogWrite()` to set a pin's voltage to somewhere between 0 V and the reference voltage. Reference voltages must not exceed 5 V obviously.

3.3.1 Function analogReference()

The AVR microcontroller, in this case, the ATmega328P, has the ability to read analog voltages—those that are not just defined as `HIGH` or `LOW`—using the `analogRead()` function. In order to do this, the comparator built in to the device needs a reference voltage to compare the unknown voltage against. This can be supplied from a number of different sources, these being

- The default, which is to use the supply voltage of 5 V or 3.3 V depending on the device. The ATmega328P on Arduino boards use a 5 V supply.
- An internally generated 1.1 V reference voltage. This must be used if the internal temperature sensor is being used as the ADC input. See Appendix E for a small sketch showing how this can be done.
- An external reference voltage on the `AREF` pin. This must be between 0 V and 5 V, or damage to the AVR microcontroller will occur.

Warning

The data sheet for the ATmega328P warns that

> If the user has a fixed voltage source connected to the AREF pin, the user must not use the other reference voltage options in the application, as they will be shorted to the external voltage. If no external voltage is applied to the `AREF` pin, the user may switch between AVCC and 1.1 V as reference selection.

The preceding warning *must* be noted. On all my own Arduino boards, AREF isn't connected at all—according to the schematics—and the temperature measuring sketch mentioned earlier works fine. There is a location on one of the headers labeled "AREF" where the maker can supply a voltage to the AREF pin. I have never connected anything to that pin, so I'm safe.

Be aware that there are many, many videos on YouTube, on the Internet in general, and in some books where circuit diagrams and so on are found which explain how you can remove all the extraneous components from an Arduino board to create your own pseudo-Arduino on a breadboard. These usually show that there are three connections to the 5 V supply—VCC, AVCC, and AREF.

The connection of AREF to VCC is completely wrong as it prevents you from being able to select the internal 1.1 V reference for the ADC or the Analog Comparator and will have the result, if you upload a program that does select the internal reference voltage, of potentially bricking your AVR microcontroller. Not a good idea.

My advice is to treat those circuits with complete disdain and never connect AREF to any supply voltage, unless you absolutely need to do so, as this will help your AVR microcontroller live long and prosper. (This, I think, is a phrase taken from some 1960s space exploration series on TV!)

The source code for the analogReference() function follows in Listing 3-5. This code can be found in the file $ARDINC/wiring_analog.c.

Listing 3-5 The analogReference() function

```
uint8_t analog_reference = DEFAULT;

void analogReference(uint8_t mode)
{
    // can't actually set the register here because
    // the default setting will connect AVCC and the
    // AREF pin, which would cause a short if there's
    // something connected to AREF.
    analog_reference = mode;
}
```

As you can see, it just changes the value in the analog_reference variable, which will be used later by analogRead(). The values that can be passed to this function, for the ATmega328P, are

- DEFAULT which has value 1
- INTERNAL which has value 3
- EXTERNAL which has value 0

All of these are defined as constants in the $ARDINC/Arduino.h header file. You may be wondering about why those exact values have been used. The description of analogRead() will tell all!

3.3.2 Function analogRead()

The `analogRead()` function connects the pins `A0–A5`, or `A0–A7` if your board has the surface mount version of the ATmega328 and the manufacturer chose to connect `A6` and `A7` to header pins, to the multiplexed inputs of the AVR microcontroller's Analog-to-Digital Converter (ADC).

The ADC can read a voltage on those pins, and using a method called successive approximation, it can work out, with reasonable accuracy, what the voltage was. Wikipedia, https://en.wikipedia.org/wiki/SuccessiveapproximationADC, has a good explanation of how successive approximation works if you are interested further.

If you think back to the `analogReference()` function, you may remember I asked why the constants defined for `DEFAULT`, `INTERNAL`, and `EXTERNAL` had the values zero, three, or one. The simple reason is because when they are shifted left by six places, they take up position in the `REFS1` and `REFS0` bits of the `ADMUX` register and are ready to go without any further processing being required. Sneaky! (And efficient.)

The register names may not be very meaningful to you at this stage; however, in Chapters 7, 8, and 9 where I look at the hardware features of the ATmega328P—which the code here depends upon—all will, hopefully, become clear. You may, if you wish, skip to Section 9.2 and read all about the ADC or just take my word for it until then!

Note
Basically, there are two bits in the control registers for the ADC which tell it where to obtain the reference voltage it needs to convert from an analog voltage to a digital value representing the voltage. Those two bits are named `REFS1` and `REFS0`. By shifting the `DEFAULT`, `EXTERNAL`, or `INTERNAL` values into those bits, the correct reference voltage is selected.

Table 3-1 shows the different values allowed and how they relate to the analog reference voltage used by the ADC.

Warning
The data sheet notes that the value 2 or 0b10 is reserved and should not be used.

It should be noted that the data sheet for the ATmega328P states that if `INTERNAL` or `EXTERNAL` references are being used, there should be a small capacitor between the `AREF` pin and ground. The Duemilanove and Uno boards use a 100 nF capacitor according to the schematics.

Listing 3-6 shows the source code for the `analogRead()` function with only the code appropriate to ATmega328P Arduino boards. The `analogRead()` function is found in `$ARDINC/wiring_analog.c`.

Table 3-1 AnalogReference values and sources

Name	Value	REFS1:0	Reference Used
DEFAULT	0	00	Default reference is the supply voltage, 5 V or 3.3 V, depending on the device
EXTERNAL	1	01	Default reference is the voltage supplied on the AVCC pin, 5 V or 3.3 V, depending on the device
INTERNAL	3	11	Default reference is the internally generated 1.1 V voltage

Listing 3-6 The analogRead() function

```
int analogRead(uint8_t pin)
{
    if (pin >= 14) pin -= 14;                          (1)

    // set the analog reference (high two bits of
    // ADMUX) and select the channel (low 4 bits).
    // this also sets ADLAR (left-adjust result)
    // to 0 (the default).

    ADMUX = (analog_reference << 6) | (pin & 0x07);   (2)

    // without a delay, we seem to read from the       (3)
    // wrong channel
    //delay(1);

#if defined(ADCSRA) && defined(ADC)
    // start the conversion
    sbi(ADCSRA, ADSC);                                 (4)

    // ADSC is cleared when the conversion finishes
    while (bit_is_set(ADCSRA, ADSC));                  (5)

    // ADC macro takes care of reading ADC register.
    // avr-g++ implements the proper reading order:
    // ADCL is read first.
    return ADC;                                        (6)
#endif
}
```

(1) Here, the pin number passed in is adjusted to ensure that it is between zero and seven. In case anyone passed `D14`, or 14, for `A0`, which is perfectly valid, this adjustment ensures `D14` becomes `A0` which has the numeric value of zero.

(2) Whatever the user set as the desired value for `analog_reference` is copied up into the appropriate bits of the `ADMUX` register, alongside the correct three bits for the desired analog pin.

(3) This comment is obviously incorrect in this version of the Arduino IDE, as the desired `delay(1)` is itself commented out!

(4) Ask the ADC to initiate a conversion of the voltage on the requested pin to a digital value. `ADSC` is the “ADC Start Conversion” bit.

(5) Hang around here, burning CPU cycles, while the ADC does its conversion. When it is complete and a result is available, bit `ADSC` in the `ADCSRA` register will be cleared. The result of the ADC’s conversion will be available in the `ADCL` and `ADCH` registers.

(6) The ADC macro ensures that when reading the two eight-bit registers which contain the result of the Analog-to-Digital conversion, although only ten bits are actually used, we must read the low value from register `ADCL` first, then the high value from `ADCH`; otherwise, we would potentially

get an incorrect reading. The ADC macro ensures that the two registers are read in the correct order and the result is returned from the function.

Note
The source code shown earlier is not exactly as it appears in `$ARDINC/wiring_analog.c`. I have stripped out a lot of checks, function calls, and assignments which are not relevant to the ATmega328P. Hopefully, this makes things a lot easier to understand. It certainly saves space on the page!

3.3.3 Function analogWrite()

The code for the `analogWrite()` function is found in the file `$ARDINC/wiring_analog.c`.

The `analogWrite()` function is used to write an eight-bit data value to one of the six pins that support Pulse Width Modulation (PWM) which allows the voltage read on the pin to appear as a value between `GND` and `VCC`. Section 8.1 explains the various timer features, including the various forms of PWM that are available, how PWM works, and how a supposedly digital device is able to make analog voltages appear on its pins.

The `analogWrite()` function takes a value between 0 and 255 and uses it to define the duty cycle (see Section 8.1.6.1) of the PWM timer connected to the appropriate pin. The higher this value is, the longer the duty cycle of the PWM signal on the pin will be and, therefore, the higher the apparent voltage on the pin will appear to be.

The `analogWrite()` function will always set the appropriate pin to be in `OUTPUT` mode, and if the pin requested is not one that allows PWM, then a `digitalWrite()` takes place on the pin, with values less than 128 indicating that the pin should be set `LOW` and higher values setting the pin to `HIGH`.

As you will see from Listing 3-7, all that the `analogWrite()` function does is decide which timer and channel the requested pin number should be connected to, connect it to that timer and channel, and set the duty cycle. Each timer has two separate channels available for PWM output, and as there are three timers on the ATmega328P, we have PWM on six pins.

The pins with PWM are `D3`, `D5`–`D6`, and `D9`–`D11`, and Table 3-2 shows which pin is controlled by which timer and that timer's channel. The timers have two channels each, hence why there are only six PWM pins on an ATmega328P.

Table 3-2 PWM pins, timers, and channels

PWM Pin	Timer	Channel
D3	2	B
D5	0	B
D6	0	A
D9	1	A
D10	1	B
D11	2	A

The three timers are separate parts of the ATmega328P and operate separately from the brains of the microcontroller—the CPU. This allows the timers to be set up and left to get on with timing or counting, while the CPU continues running the program.

Warning
The two channels of each timer operate independently of each other. This allows pin `D5` to have one value written by `analogWrite()` and pin `D10` to have another, different value. This applies because of the PWM mode chosen by the designers of the Arduino Language and is explained in some details in Section 8.1, which deals with the timer hardware in the ATmega328P.

The source code for the `analogWrite()` function, found in `$ARDINC/wiring_analog.c`, is shown in Listing 3-7 and, as usual, has had all nonrelevant sections removed.

Listing 3-7 The analogWrite() function

```
void analogWrite(uint8_t pin, int val)
{
  // We need to make sure the PWM output is enabled for
  // those pins that support it, as we turn it off when
  // digitally reading or writing with them. Also, make
  // sure the pin is in output mode for consistently with
  // Wiring, which doesn't require a pinMode call for the
  // analog output pins.

  pinMode(pin, OUTPUT);                                  (1)
  if (val == 0) {
    digitalWrite(pin, LOW);
  }
  else if (val == 255) {
    digitalWrite(pin, HIGH);
  }
  else {
    switch(digitalPinToTimer(pin)) {                     (2)

      #if defined(TCCR0A) && defined(COM0A1)             (3)
      case TIMER0A:
      // connect PWM to pin on timer 0, channel A
      sbi(TCCR0A, COM0A1);
      OCR0A = val; // set PWM duty
      break;
      #endif

      #if defined(TCCR0A) && defined(COM0B1)             (4)
      case TIMER0B:
      // connect PWM to pin on timer 0, channel B
      sbi(TCCR0A, COM0B1);
      OCR0B = val; // set PWM duty
      break;
      #endif

      #if defined(TCCR1A) && defined(COM1A1)                 (5)
      case TIMER1A:
      // connect PWM to pin on timer 1, channel A
      sbi(TCCR1A, COM1A1);
      OCR1A = val; // set PWM duty
      break;
      #endif

      #if defined(TCCR1A) && defined(COM1B1)                 (6)
      case TIMER1B:
      // connect PWM to pin on timer 1, channel B
      sbi(TCCR1A, COM1B1);
      OCR1B = val; // set PWM duty
      break;
      #endif

      #if defined(TCCR2A) && defined(COM2A1)                 (7)
      case TIMER2A:
      // connect PWM to pin on timer 2, channel A
      sbi(TCCR2A, COM2A1);
      OCR2A = val; // set PWM duty
      break;
      #endif

      #if defined(TCCR2A) && defined(COM2B1)                 (8)
      case TIMER2B:
      // connect PWM to pin on timer 2, channel B
      sbi(TCCR2A, COM2B1);
      OCR2B = val; // set PWM duty
      break;
      #endif

      case NOT_ON_TIMER:                                     (9)
      default:
      if (val < 128) {
        digitalWrite(pin, LOW);
      } else {
        digitalWrite(pin, HIGH);
      }
    } // End switch
  } // End if
}
```

(1) The pin is made an `OUTPUT` pin, and as a quick test and return, if the value is either 0 or 255, the two limits for `analogWrite()`, then the pin is simply set to ground or supply voltage using `digitalWrite()`. This avoids a slight timing error when the timer is in PWM mode and is set to one of its limits. The data sheet has details, if you wish to investigate further.
(2) The pin is converted to a timer and channel by calling `digitalPinToTimer()` which uses the table `digital_pin_to_timer_PGM` created in `$ARDINST/variants/standard/pins_arduino.h` to determine if the pin is a PWM pin or otherwise. This returns a value of `NOT_ON_TIMER` if the pin is purely digital and that will be handled by the default case.
(3) This is where pin `D6` is configured. The pin is connected to the timer, and the `OCR0A` register is loaded with the value passed to `analogWrite()` to enable the correct duty cycle for the timer's PWM output. `D6` is on Timer 0, channel A.
(4) Configuration of pin `D5` is performed here and sets up `D5` on Timer 0, channel B.
(5) This is where pin `D9` is configured to use Timer 1, channel A.
(6) Pin `D10` configuration. Pin `D10` uses Timer 1, channel B.
(7) Pin `D11` is configured here and uses channel A on Timer 2.
(8) Pin `D3` is configured here. `D3` is configured to use channel B on Timer 2.
(9) In the event that the supplied pin number is not able to output PWM, this part of the code digitally sets the pin `LOW` or `HIGH` according to the requested PWM value passed to `analogWrite()`.

The Arduino Reference website, www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/, warns that

> The PWM outputs generated on pins 5 and 6 will have higher-than-expected duty cycles. This is because of interactions with the `millis()` and `delay()` functions, which share the same internal timer used to generate those PWM outputs. This will be noticed mostly on low duty-cycle settings (e.g. 0–10) and may result in a value of 0 not fully turning off the output on pins 5 and 6.

Looking at the code in Listing 3-7, a value of zero will ignore PWM altogether and simply use `digitalWrite()` to turn off the pin. I suspect the warning in the comments may refer to older types of Arduino boards. The ATmega328P data sheet also advises against PWM values of zero or `TOP` where `TOP` is the configured highest value for the timer in use. I would say that the checks in Listing 3-7 which test for 0 or 255 are obviously there to get around the problem.

Note
A pin can be `HIGH` or `LOW`. PWM turns a pin `HIGH` and `LOW`, and the sum of one `HIGH` plus one `LOW` is the period. This is related to the PWM frequency which is defined by the timer's prescaler. Section 8.1.6.2 has all the gory details.

Duty cycle is usually expressed as a percentage. It defines the time that the pin is `HIGH` as a percentage of the period. A duty cycle of 10% means that the pin is `HIGH` for 10% of its period and `LOW` of the remaining 90%. A 10% duty cycle would appear as a voltage very close to 10% of `VCC` on the PWM pin.

3.4 Advanced Input/Output

In this section, I take you through the advanced input/output functions which allow you to make sounds and measure logic levels on pins to determine how long a specific state was held for and an easy way to shift a byte value from a variable out onto a digital pin, and vice versa.

3.4.1 Function tone()

The code for the `tone()` function is found in the file `$ARDINC/Tone.cpp`.

The `tone()` function generates a square wave of the specified frequency, with a 50% duty cycle, on any pin. A duration can be specified; otherwise, the wave continues until a call to `noTone()` is made. The pin can be connected to a piezo buzzer or other speakers to play tones.

If `tone()` has been called, then PWM on pins `D3` and/or `D11` will be affected. These analog pins are maintained by Timer 2, and it is Timer 2 that the `tone()` function uses to generate a square wave.

If you have, for example, a pair of LEDs, fading in and out, on pins `D3` and `D11`, then whenever the `tone()` function is called, and while sounding a tone, the LEDs will be off. Fading LEDs on the other pins available for `analogWrite()` will not be affected. Those are pins `D5–D6` and `D9–D10`. Only pins `D3` and `D11` are affected by this problem.

The Arduino documentation for the `tone()` function, www.arduino.cc/reference/en/language/functions/advanced-io/tone/, has a link to Brett Hagman's GitHub site, https://github.com/bhagman/Tone#ugly-details, where we find this warning:

Warning
Do not connect the pin directly to some sort of audio input. The voltage on the output pin will be 5 V or 3.3 V and is considerably higher than a standard line level voltage (usually around 1 V peak to peak) and can damage sound card inputs, etc. You could use a voltage divider to bring the voltage down, but you have been warned.

You *must* have a resistor in line with the speaker, or you will damage your microcontroller.

The resistor mentioned is shown on Brett's circuit diagram as having a value of 1 KΩ. That should, if I can do the calculations properly, restrict the current to 5 mA at 5 V.

Note
Brett is the author of the Tone Library, a simplified version of which has been included with the Arduino IDE since version 0018.

Listing 3-8 shows how the `tone()` function can be called in one of two ways.

Listing 3-8 Example tone() function calls

```
tone(pin, frequency);

// or

tone(pin, frequency, duration);
```

The duration, if it is omitted or zero, causes the tone to sound forever or until `noTone()` is called.

On the Arduino boards based around the ATmega328P, only one pin can be generating a tone at any time. If a tone is already playing on a pin, a call to `tone()` with a different pin number will have no effect unless `noTone()` was called first. If the call is made for the same pin as the one currently playing, the call will set the tone's frequency to that specified in the most recent call.

In order for `tone()` to function correctly, a couple of tables are required to be set up to control which timer will be used to generate a tone and to keep a record of all the pins that are currently generating. This code is shown in Listing 3-9.

As with other listings in this book, I have removed any code that is not relevant to the ATmega328P.

Listing 3-9 Variables used by the tone() function

```
#define AVAILABLE_TONE_PINS 1                                      (1)
#define USE_TIMER2                                                 (2)

const uint8_t PROGMEM tone_pin_to_timer_PGM[] =
    { 2 /*, 1, 0 */ };                                             (3)

static uint8_t tone_pins[AVAILABLE_TONE_PINS] =
    { 255 /*, 255, 255 */ };                                       (4)
```

(1) This shows that the ATmega328 family has only one pin that can play at any one time—at least on the version of the code included with the Arduino IDE version 2.1.0.
(2) This tells the code, later on, which interrupt routine to use to do the actual tone generation.
(3) This array, which is created in the AVR microcontroller's flash memory, holds a list of the various timers that can be used to generate tones. As the `tone()` function is a cut-down version of Brett's library, only a single timer is used; currently, this is Timer 2.
(4) This array, which is created in the AVR microcontroller's Static RAM, holds a list of all the pin numbers that are currently playing a tone, or 255 if nothing is playing. There is one entry in the table for each timer that can be used to generate tones. That means there is one entry in total for the ATmega328P boards. There can be only one. (Well, I *am* a Highlander (www.imdb.com/title/tt0091203/)!)

As you can see, the ATmega328P-based boards only have a single timer in use to generate tones, this being Timer 2.

In addition to the tables listed in Listing 3-9, the variables in Listing 3-10 are also required by the `tone()` function, specifically, in the interrupt service routine (ISR), see Listing 3-14, which does the actual sound generation.

Listing 3-10 Variables used by the tone() function ISR

```
volatile long timer2_toggle_count;
volatile uint8_t *timer2_pin_port;
volatile uint8_t timer2_pin_mask;
```

Note
You will note that all of these are declared `volatile`. This is because they will be used in an interrupt service routine (ISR), and any variable you wish to read or write during an interrupt must be declared as being `volatile`. If you forget, your code may not work because the compiler optimized the variable away.

The variable `timer2_toggle_count` holds the number of times that the interrupt routine will be called, if a duration for the tone is requested. If no duration is requested, this is unused.

`Timer2_pin_port` is related to the internal register to be used for the PORTx port for the pin which will be generating the sound, while `timer2_pin_mask` is a bitmask with a single bit, corresponding to the required pin, set to one. This indicates which bit in the PORTx register is being used to generate the sound.

The `tone()` function works by working out a suitable prescaler for the timer clock so that the number of ticks to be counted per transition (`LOW` to `HIGH` or `HIGH` to `LOW`) of the pin falls into the range 0 to 255. This is required because Timer 2 is only eight bits wide and can only count within this range.

If the frequency chosen is such that even with the maximum prescaler, 1,024, in use and the value still cannot fall into the range required, the code simply attempts to carry on regardless. This may render the tone generated to be the wrong frequency.

The first part of the source code for the `tone()` function is shown in Listing 3-11 and continues in Listing 3-12, and sections which are not relevant to ATmega328P devices have been removed for clarity.

Listing 3-11 The tone() function

```
// frequency (in hertz) and duration (in milliseconds).
void tone(uint8_t _pin,
          unsigned int frequency,
          unsigned long duration)
{
  uint8_t prescalarbits = 0b001;
  long toggle_count = 0;
  uint32_t ocr = 0;
  int8_t _timer;

  _timer = toneBegin(_pin);                                    (1)

  if (_timer >= 0) {
    // Set the pinMode as OUTPUT
    pinMode(_pin, OUTPUT);

    // if we are using an 8 bit timer, scan through prescalars
    // to find the best fit.

    if (_timer == 0 || _timer == 2) {
      ocr = F_CPU / frequency / 2 - 1;                         (2)
      prescalarbits = 0b001; // ck/1: same for both timers
      if (ocr > 255) {
        ocr = F_CPU / frequency / 2 / 8 - 1;
        prescalarbits = 0b010; // ck/8: same for both timers
        if (_timer == 2 && ocr > 255) {
           ocr = F_CPU / frequency / 2 / 32 - 1;
          prescalarbits = 0b011;
        }

        if (ocr > 255) {
          ocr = F_CPU / frequency / 2 / 64 - 1;
          prescalarbits = _timer == 0 ? 0b011 : 0b100;

          if (_timer == 2 && ocr > 255) {
            ocr = F_CPU / frequency / 2 / 128 - 1;
            prescalarbits = 0b101;
          }

          if (ocr > 255) {
            ocr = F_CPU / frequency / 2 / 256 - 1;
            prescalarbits = _timer == 0 ? 0b100 : 0b110;
            if (ocr > 255) {
                // can't do any better than /1024
                ocr = F_CPU / frequency / 2 / 1024 - 1;   (3)
                prescalarbits = _timer == 0 ? 0b101 : 0b111;
            }
          }
        }
      }

    TCCR2B = (TCCR2B & 0b11111000) | prescalarbits;       (4)
    } // End if (_timer == 0 || _timer == 2)
```

(1) The call to `toneBegin()` returns a timer number. On the standard Arduino boards, based around the ATmega328 family of AVR microcontrollers, this will be Timer 2. The `toneBegin()` function is discussed in Listing 3-13.

(2) These lines onward attempt to fit the required frequency into the range 0–255 using any of the available Timer 2 prescaler values.
The frequency of the system clock, 16 MHz, is divided by twice the required frequency because in order to make a tone, the pin must be raised and lowered—`LOW` -> `HIGH` -> `LOW`. This is then divided by the prescaler value being considered. The subtraction of one from the result is because the AVR microcontroller counts from zero. If the result fits into an eight-bit value, then the current prescaler value will be used.

(3) This is the last resort at fitting the frequency into range. If the prescaler set to divide by 1,024 still cannot fit, then the code simply carries on regardless. This could result in the frequency being a tad wrong.

(4) After the call to `toneBegin()`, Timer 2 is set up to run with no prescaling, so the calculated prescaler bits must be set up in the `TCCR2B` register, in bits `CS22:20`. See Section 8.1 for a full description of prescalers. These bits will be set to whatever is in the lowest three bits of the `prescalarbits` variable. This sets the timer to the correct prescaler value for the frequency that is required to be generated.

For example, if we assume that the system clock is 16 MHz, and we wish to produce a tone of 440 Hz for the note A4 above middle C, then the repeated tests earlier will result in the following:

- With a prescaler of 1, the value calculated is 18,180 which cannot be used in an eight-bit timer.
- With a prescaler of 8, the value calculated is 2,271 which cannot be used in an eight-bit timer.
- With a prescaler of 32, the value calculated is 567 which cannot be used in an eight-bit timer.
- With a prescaler of 64, the value calculated is 283 which cannot be used in an eight-bit timer.
- With a prescaler of 128, the value calculated is 141 which *can* be used in an eight-bit timer.
- With a prescaler of 256, the value calculated is 70 which *could* be used but won't be as prescaler 128 was found to fit.
- With a prescaler of 1,024, the value calculated is 16 which *could also* be used but won't be as prescaler 128 was found to fit.

Of these, the first one to fit into the range 0 to 255 is a prescaler of 128. So the `ocr` variable is set to 141, and the `prescalarbits` variable is set to 0b101 to select a divide by 128 prescaler.

The source code for the `tone()` function continues in Listing 3-12.

Listing 3-12 The tone() function, continued

```
    ...

    // Calculate the toggle count
    if (duration > 0) {
      toggle_count = 2 * frequency * duration / 1000;     (1)
    } else {
      toggle_count = -1;
    }

    // Set the OCR for the given timer,
    // set the toggle count,
    // then turn on the interrupts

    switch (_timer) {

#if defined(OCR2A) && defined(TIMSK2) && defined(OCIE2A)

      case 2:
        OCR2A = ocr;                                      (2)
        timer2_toggle_count = toggle_count;               (3)
        bitWrite(TIMSK2, OCIE2A, 1);                      (4)
        break;
#endif
    } // end switch (_timer)
  } // End if (_timer >= 0)
}
```

(1) The duration is checked, and if it is specified, then the tone must only be generated for that length of time. The duration supplied to `tone()` is specified in milliseconds; however, as frequency is measured in cycles per second, the value is converted to seconds.

Continuing the preceding example of a 440 Hz tone, if the duration is required to be 1.5 seconds, then the `toggle_count` variable is set to $2 * 440 * 1.5$ or 1,320. This is the number of times in 1.5 seconds that the pin must be toggled to ensure that a 440 Hz tone is generated.

(2) The `ocr` value is the maximum count that the timer counts up to before it clears to zero and causes an interrupt. Unlike the `millis()` interrupt routine, which counts from 0 to 255 (on Timer 0), the maximum value for this timer, when generating tones, is based on the frequency and the calculated prescaler value.

Once more, with the current example, the timer will count from 0 to 141 and toggle the pin, then again from 0 to 141 and toggle the pin, and so on for the desired duration.

(3) If a duration was specified, the number of times the pin must be toggled is written to the `timer2_toggle_count` variable which will be used to disable the tone, in the interrupt routine, when the duration has passed. This is -1 if no duration was requested.

(4) The setting of the bit `OCIE2A` in register `TIMSK2` enables the Timer 2 Compare Match A Interrupt. That interrupt routine is the code that actually toggles the pin to cause the sound to be generated and is discussed in Listing 3-14.

The `tone()` function begins by calling `toneBegin()` to obtain a timer number to use. Listing 3-13 shows the relevant parts of the `toneBegin()` function. As before, parts of the code that are not relevant to ATmega328P devices have been removed for clarity.

Listing 3-13 The toneBegin() function

```
static int8_t toneBegin(uint8_t _pin)
{
  int8_t _timer = -1;

  // if we're already using the pin, the timer
  // should be configured.

  for (int i = 0; i < AVAILABLE_TONE_PINS; i++) {         (1)
    if (tone_pins[i] == _pin) {
      return pgm_read_byte(tone_pin_to_timer_PGM + i);
    }
  }

  // search for an unused timer.
  for (int i = 0; i < AVAILABLE_TONE_PINS; i++) {         (2)
    if (tone_pins[i] == 255) {
      tone_pins[i] = _pin;
```

```
        _timer = pgm_read_byte(tone_pin_to_timer_PGM + i);
        break;
      }
    }

  if (_timer != -1) {                                        (3)

    // Set timer specific stuff
    // All timers in CTC mode
    // 8 bit timers will require changing prescalar values,
    // whereas 16 bit timers are set to either ck/1
    // or ck/64 prescalar

    switch (_timer) {

      // Code removed - not relevant.

      #if defined(TCCR2A) && defined(TCCR2B)
      case 2:                                                (4)
        // 8 bit timer
        TCCR2A = 0;
        TCCR2B = 0;
        bitWrite(TCCR2A, WGM21, 1);
        bitWrite(TCCR2B, CS20, 1);
        timer2_pin_port =
          portOutputRegister(digitalPinToPort(_pin));
        timer2_pin_mask =
          digitalPinToBitMask(_pin);
        break;
      #endif

      // Code removed - not relevant.

    } // End switch (_timer)
  } // End if (_timer != -1)

  return _timer;
}
```

(1) The `toneBegin()` code starts by checking the `tone_pin_to_timer_PGM` array to see if the requested pin is currently generating a tone. If it is, the code simply returns the timer number to `tone()`. On an ATmega328P-based Arduino board, this is always Timer 2.

(2) The `tone_pins` array is searched for an unused pin. If an entry is found, the `tone_pin_to_timer_PGM` array is read to determine the timer number that can be used. There's only one timer on Arduino boards, Timer 2.

(3) If there were no free slot(s) in the `tone_pins` array, `toneBegin()` exits, returning −1 to the calling code in `tone()`.

(4) Much bit twiddling then ensues to initialize Timer 2 with

- Bit WGM21 set in register TCCR2A to put the timer into "Clear Timer on Compare" mode. The counter value will reset to zero whenever it reaches the value in OCR2A.
- Bit CS20 set in register TCCR2B to put the timer into "no prescaling" mode—the timer's clock will run at the system clock speed—16 MHz. This will be amended on return to tone() when the frequency-dependent prescaler value is calculated.
- The variable timer_pin_port is determined next. This is based on the pin requested and will be one of PORTB, PORTC, or PORTD on the ATmega328P.
- The required pin's bitmask, in timer2_pin_mask, is calculated next. This will be an eight-bit value, with a single bit set to indicate the required pin, on the just calculated PORTx register.

Most of the setup for tone() is now complete. All that is required is an interrupt service routine to do the actual tone generation. The interrupt in use is the "Timer 2 Compare Match A" interrupt, which is discussed in Listing 3-14.

Listing 3-14 The ISR for the tone() function

```
#ifdef USE_TIMER2
ISR(TIMER2_COMPA_vect) {
  if (timer2_toggle_count != 0) {                            (1)

    // toggle the pin
    *timer2_pin_port ^= timer2_pin_mask;                     (2)

    if (timer2_toggle_count > 0)                             (3)
      timer2_toggle_count--;
  } else {

    // need to call noTone() so that the tone_pins[]
    // entry is reset, so the timer gets initialized
    // next time we call tone().
    // XXX: this assumes timer 2 is always the first
    // one used.

    noTone(tone_pins[0]);                                    (4)

    // disableTimer(2);
    // *timer2_pin_port &= ~(timer2_pin_mask);
  }
}
#endif
```

(1) If there are still toggles to be done, then the pin must be toggled.
(2) The appropriate PORTx register has its bits XOR'd with the pin's bitmask. This toggles just the bit that is set to indicate which pin is being used to generate a tone. If the bit is currently a one, then XORing with the one bit in the mask will set the PORTx register's bit to a zero, and vice versa. This sets the physical pin HIGH or LOW accordingly.
(3) The number of toggles remaining for the required duration is reduced.

(4) If there were no more toggles to be done, then a call is made to the `noTone()` function to silence the sound. The old code following this line has been commented out. This is because `noTone()` disables the timer and sets the pin to `LOW`.

The Arduino Reference for the `tone()` function, www.arduino.cc/reference/en/language/functions/advanced-io/tone/, has the following to say about `tone()`:

> If you want to play different pitches on multiple pins, you need to call `noTone()` on one pin before calling `tone()` on the next pin.
> It is not possible to generate tones lower than 31Hz. For technical details, see Brett Hagman's notes. (https://github.com/bhagman/Tone#ugly-details)
> Use of the `tone()` function will interfere with PWM output on pins 3 and 11 (on boards other than the Mega).

3.4.2 Function noTone()

The code for the `noTone()` function is found in the file `$ARDINC/Tone.cpp`.

The `noTone()` function, as discussed in Listing 3-15, should be called to stop the generation of a square wave triggered by the `tone()` function. When the function is called, it turns off tone generation on the supplied pin, assuming that it was generating a tone, and then sets the pin's state to `LOW` regardless of whether or not it was previously generating a tone.

Listing 3-15 The noTone() function

```
void noTone(uint8_t _pin)
{
  int8_t _timer = -1;
  for (int i = 0; i < AVAILABLE_TONE_PINS; i++) {
    if (tone_pins[i] == _pin) {
      _timer = pgm_read_byte(tone_pin_to_timer_PGM + i);  (1)
      tone_pins[i] = 255;                                 (2)
      break;
    }
  }

  disableTimer(_timer);                                   (3)
  digitalWrite(_pin, 0);                                  (4)
}
```

(1) If the pin number supplied is currently generating a tone, then it is converted to a timer number. In ATmega328P variants of the Arduino board, this will always be Timer 2—that's the only timer currently used for tone generation.
(2) Indicate that no pins are generating a tone.
(3) If the pin was not generating a tone, the `_timer` variable will be set to -1, and that will have no effect in `disableTimer()`. Otherwise, Timer 2 will be disabled, stopping the generation.
(4) The pin is set `LOW`, regardless of whether it was generating a tone or not. This contradicts the documentation for `noTone()` which states that *this has no effect if no tone is being generated on the specified pin when called*. This is definitely not the case if the pin was actually `HIGH` when `noTone()` was called on it.

The `disableTimer()` function, called from `noTone()`, is discussed in Listing 3-16. As with other listings, those parts of the code not relevant to the ATmega328P have been removed.

Listing 3-16 The disableTimer() function

```
void disableTimer(uint8_t _timer) {
  switch (_timer) {
    case 2:                                                (1)
      #if defined(TIMSK2) && defined(OCIE2A)
        bitWrite(TIMSK2, OCIE2A, 0);
      #endif

      #if defined(TCCR2A) && defined(WGM20)                (2)
        TCCR2A = (1 << WGM20);
      #endif

      #if defined(TCCR2B) && defined(CS22)                 (3)
        TCCR2B = (TCCR2B & 0b11111000) | (1 << CS22);
      #endif

      #if defined(OCR2A)                                   (4)
        OCR2A = 0;
      #endif

    break;
  }
}
```

(1) Setting bit `OCIE2A` in register `TIMSK2` turns off the "Timer 2 Compare Match A" interrupt which had been turned on by the `toneBegin()` function itself called from the `tone()` function.
(2) This resets Timer 2 into "Phase Correct PWM" mode originally set up in `init()` but which was changed to "Clear Timer on Compare" mode by the `toneBegin()` function, called from `tone()`.
(3) This resets the prescaler back to divide by 64 for Timer 2. The `tone()` function changed that setting according to the frequency of the tone that was requested.
(4) This resets the "Output Compare A" value for the timer.

The effect of the code in Listing 3-16 is to restore the state of Timer 2, and its interrupts and so on, back to those configured in the initial setup within the `init()` function as described in Section 2.10.

www.arduino.cc/reference/en/language/functions/advanced-io/notone/ has the following to say on `noTone()`:

> If you want to play different pitches on multiple pins, you need to call `noTone()` on one pin before calling `tone()` on the next pin.

3.4.3 Function pulseIn()

The `pulseIn()` function, found in the file `$ARDINC/wiring_pulse.c`, is used to measure the period of a pulse on any pin. There is an alternative function, `pulseInLong()`, which is discussed later, which is better at handling long duration pulses than `pulseIn()`; however, `pulseIn()` can be used in sketches where interrupts are disabled.

The function reads a pulse—`HIGH` or `LOW`—on any pin and returns the length of the pulse in microseconds. There are two methods of calling this function as shown in Listing 3-17.

Listing 3-17 Calling the pulseIn() function

```
unsigned long microseconds = 0;
microseconds = pulseIn(pin, state, timeout);

// or

// Timeout defaults to 1 second...
microseconds = pulseIn(pin, state);
```

The return value is an `unsigned long`.

If no timeout is specified, the default is one second. The timeout is specified in microseconds—millionths of a second.

If the function is called with a `LOW` state to be measured, it will

- Wait for the pin in question to become `HIGH` if it currently is `LOW`. If the timeout expires before the pin goes `HIGH`, the function returns zero. Thus, the code is waiting for the current pulse to end.
- Wait for the pin to go `LOW` again—this is the start of the pulse to be measured. Once more, if the remaining time left in the timeout expires, the function returns zero.
- Wait for the pin to go `HIGH` again—this is the end of the pulse to be measured. If the remaining time left in the timeout expires, then the function again returns zero; otherwise, it returns the time, in microseconds, that the pin remained `LOW`.

Note
To be absolutely clear, the timeout you pass to the function is used for *all* three stages of the function—waiting for the current pulse to finish, waiting for the new pulse to start, and then waiting for the new pulse to complete. The timeout has to remain unexpired throughout all three stages.

Similar, but opposite, actions take place when `pulseIn()` is called with a state of `HIGH`.

Unlike the function `pulseInLong()`, `pulseIn()` can be executed when interrupts are disabled as it does not require the `micros()` function.

It measures the pulse length by calling a function, `countPulseASM()`, written in AVR assembly language. This function can be seen in the file `$ARDINC/wiring_pulse.S`, and according to the source code, it started life as a C routine, similar to that shown in Listing 3-18. I will not be discussing the assembly language version of `countPulseASM()` here—looking at compiler output is quite tedious it has to be said.

Listing 3-18 C code version of countPulseASM()

```
unsigned long pulseInSimpl(volatile uint8_t *port,
                           uint8_t bit,
                           uint8_t stateMask,
                           unsigned long maxloops)
{
  unsigned long width = 0;                                  (1)

  // wait for any previous pulse to end
  while ((*port & bit) == stateMask)
    if (--maxloops == 0)
      return 0;                                             (2)

  // wait for the pulse to start
  while ((*port & bit) != stateMask)
    if (--maxloops == 0)
      return 0;                                             (3)

  // wait for the pulse to stop
  while ((*port & bit) == stateMask) {
    if (++width == maxloops)
      return 0;                                             (4)
  }

  return width;                                             (5)
}
```

(1) The pulse length will be returned in the `width` variable; here, it is initialized to zero.
(2) The function returns zero if the timeout expired while waiting for the current pulse to end.
(3) The function returns zero if the timeout remaining expired while waiting for the new pulse to begin.
(4) The function returns zero if the timeout remaining expired while the new pulse was being measured.
(5) The pulse has been successfully measured, and the variable `width` holds its width as the number of `while` loops which were executed while measuring the pulse length.

Listing 3-19 shows the source code for the current version of the `pulseIn()`.

Listing 3-19 The pulseIn() function

```
/*
 * Measures the length (in microseconds) of a pulse on the
 * pin; state is HIGH or LOW, the type of pulse to measure.
 * Works on pulses from 2-3 microseconds to 3 minutes in
 * length, but must be called at least a few dozen microseconds
 * before the start of the pulse.
 *
 * This function performs better with short pulses in
 * noInterrupt() context
 */

unsigned long pulseIn(uint8_t pin,
                      uint8_t state,
                      unsigned long timeout)
{

  // cache the port and bit of the pin in order to speed
  // up the pulse width measuring loop and achieve finer
  // resolution. calling digitalRead() instead yields
  // much coarser resolution.

  uint8_t bit = digitalPinToBitMask(pin);               (1)
  uint8_t port = digitalPinToPort(pin);                 (2)
  uint8_t stateMask = (state ? bit : 0);                (3)

  // convert the timeout from microseconds to a number
  // of times through the initial loop; it takes
  // approximately 16 clock cycles per iteration

  unsigned long maxloops =
      microsecondsToClockCycles(timeout)/16;            (4)

  unsigned long width =
      countPulseASM(portInputRegister(port),
                    bit, stateMask, maxloops);          (5)

  // prevent clockCyclesTomicroseconds to return
  // bogus values if countPulseASM timed out

  if (width)                                            (6)
    return clockCyclesTomicroseconds(width * 16 + 16);
  else
    return 0;
}
```

(1) The supplied pin number is converted to a bitmask using the function `digitalPinToBitMask()` which returns the bitmask as an eight-bit binary value, with a single bit set to one. This bit represents the desired physical pin on the AVR microcontroller and its appropriate position in the PORTx and PINx registers.
(2) The appropriate port (`PB`, `PC`, or `PD`) is obtained from the pin number.
(3) The `stateMask` variable is set to zero for `LOW` or to the pin's bitmask for `HIGH`.
(4) The timeout counter is initialized to the required number of microseconds multiplied by 16 as the assembly code will loop in system clock cycles as opposed to microseconds.

(5) The assembly language code is called to

- Wait for the current pulse on the pin to end or the timeout to expire
- Wait for the pin to begin a new pulse or the timeout to expire
- Wait for the pulse to end or the timeout to expire
  The call to `portInputRegister()` will convert the pin's port number, `PB`, `PC`, or `PD`, to the appropriate PINx register in the AVR microcontroller, and this will be passed to the assembly code.

(6) The `width` value is converted from the number of `while` loops executed while measuring the pulse to microseconds. Each loop takes 16 microseconds, so width is multiplied by 16. It now holds the number of clock cycles it spent in the loop. An additional 16 is added for the overhead of calling and returning from the `countPulseASM()` function. These clock cycles are then converted to microseconds by the `clockCyclesToMicroseconds()` function.

Note
The reason that the code calls out to an assembly language routine is at first a little mysterious, especially as the assembly code was, it appears, created from C code originally. Why not just call the C code again?

Well, what happens to the timings in the C code if, somehow, a compiler version that is later used by the IDE implements a new optimization that makes the generated code run much more quickly? The timings will be off somewhat.

Running a standard assembly language routine, where the actual timings of each and every instruction are known, means that no matter what improvements are made in the compiler, the existing `countPulseASM()` function will continue to run at exactly the same speed, thus keeping the results of `pulseIn()` consistent.

The Arduino Reference for `pulseIn()`, www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/, has the following note about the function:

> The timing of this function has been determined empirically and will probably show errors in longer pulses. [It] Works on pulses from 10 microseconds to 3 minutes in length.

There is also an example of the function's use on the website, as shown in Listing 3-20.

Listing 3-20 Example usage of pulseIn()

```
int pin = 7;
unsigned long duration;

void setup()
{
  pinMode(pin, INPUT);
}

void loop()
{
  duration = pulseIn(pin, HIGH);
}
```

However, the example code doesn't do anything with the returned result!

3.4.4 Function pulseInLong()

The `pulseInLong()` function is found in the file `$ARDINC/wiring_pulse.c`. It is an alternative to the `pulseIn()` function previously described and is better at handling long duration pulses. It cannot, however, be used in sketches where interrupts are disabled.

The function reads a pulse—`HIGH` or `LOW`—on any pin and returns the length of the pulse in microseconds. There are two methods of calling this function as shown in Listing 3-21.

Listing 3-21 Calling the pulseInLong() function

```
unsigned long microseconds = 0;
microseconds = pulseInLong(pin, state, timeout);

// or

// Timeout defaults to 1 second
microseconds = pulseInLong(pin, state);
```

The return value is an `unsigned long`.

If no timeout is specified, the default is one second. The timeout should be specified in microseconds—millionths of a second.

If the function is called with a `HIGH` state to be measured, it will

- Wait for the pin in question to become `LOW` if it currently is `HIGH`. If the timeout expires before the pin goes `LOW`, the function returns zero. Thus, the code is waiting for the current pulse to end.
- Wait for the pin to go `HIGH` again—this is the *start* of the pulse to be measured. Once more, if the *remaining* time left in the timeout expires before the pin goes `HIGH`, the function returns zero.
- Wait for the pin to go `LOW` again—this is the *end* of the pulse to be measured. If the *remaining* time left in the timeout expires before the pin goes `LOW`, then the function again returns zero; otherwise, it returns the time, in microseconds, that the pin remained `HIGH`.

Similar, but opposite, actions take place when called with a state of `LOW`.

The source code for the `pulseInLong()` function is discussed in Listing 3-22, which has been slightly reformatted so that it can fit on the page.

Listing 3-22 The pulseInLong() function

```
/* Measures the length (in microseconds) of a pulse on the
* pin; state is HIGH or LOW, the type of pulse to measure.
* Works on pulses from 2-3 microseconds to 3 minutes in
* length, but must be called at least a few dozen
* microseconds before the start of the pulse.
*
* ATTENTION:                                             (1)
* this function relies on micros() so cannot be
* used in noInterrupt() context
```

```
*/

unsigned long pulseInLong(uint8_t pin,
                          uint8_t state,
                          unsigned long timeout)
{
  // cache the port and bit of the pin in order to speed
  // up the pulse width measuring loop and achieve finer
/ / resolution. Calling digitalRead() instead yields much
  // coarser resolution.

  uint8_t bit = digitalPinToBitMask(pin);                (2)
  uint8_t port = digitalPinToPort(pin);                  (3)
  uint8_t stateMask = (state ? bit : 0);

  unsigned long startMicros = micros();                  (4)

  // wait for any previous pulse to end                  (5)
  while ((*portInputRegister(port) & bit) == stateMask) {
    if (micros() - startMicros > timeout)
      return 0;
    }

  // wait for the pulse to start                         (6)
  while ((*portInputRegister(port) & bit) != stateMask) {
    if (micros() - startMicros > timeout)
      return 0;
  }

  unsigned long start = micros();                        (7)
  // wait for the pulse to stop                          (8)
  while ((*portInputRegister(port) & bit) == stateMask) {
    if (micros() - startMicros > timeout)
      return 0;
  }
  return micros() - start;                               (9)
}
```

(1) Your attention is drawn to this comment. Because the code uses the `micros()` function, it cannot be used if global interrupts have been turned off. This is because the value returned from the call to `micros()` will not change without interrupts[1] running. The `micros()` function is discussed in Section 3.5.3.
(2) The supplied pin number is converted to a bitmask.
(3) The appropriate port (`PB`, `PC`, or `PD`) is obtained from the pin number.

[1] Specifically, the Timer 0 overflow interrupt.

(4) The timeout counter is initialized. Because this uses `micros()`, interrupts must be enabled, and the default interrupt handler for the Timer 0 Overflow Interrupt must be configured. (This is done at startup in the `init()` function as discussed in Section 2.10.)

(5) The wait for the pin to stop being in the state we are considering happens here. If the timeout expires during the wait, the function returns zero. The pulse did not occur.
The call to `portInputRegister()` will convert the pin's port number, `PB`, `PC`, or `PD`, to the appropriate PINx register in the AVR microcontroller.

(6) The wait for the pin to start the next pulse happens here. If the remaining timeout expires during the wait, the function returns zero. The pulse did not occur.

(7) A new pulse has started, so the time that it started is recorded in `start`.

(8) The function waits while the pulse continues. If the timeout expires while waiting, the function will once again return zero to indicate that the pulse did not get measured completely within the timeout period.

(9) If the pulse did complete before the timeout expired, the function returns the time that the pin was in the state that the sketch was interested in.

The `pulseInLong()` documentation at www.arduino.cc/reference/en/language/functions/advanced-io/pulseinlong/ gives the following notes about the `pulseInLong()` function:

> The timing of this function has been determined empirically and will probably show errors in shorter pulses. Works on pulses from 10 microseconds to 3 minutes in length. This routine can be used only if interrupts are activated. Furthermore the highest resolution is obtained with large intervals.
> This function relies on `micros()` so cannot be used in the `noInterrupts()` context.

There is also an example of the function's use on the Arduino website, reproduced in Listing 3-23.

Listing 3-23 Example usage of pulseInLong()

```
int pin = 7;
unsigned long duration;

void setup() {
  pinMode(pin, INPUT);
}

void loop() {
  duration = pulseInLong(pin, HIGH);
}
```

However, the example doesn't do anything with the returned result!

3.4.5 Function shiftIn()

The code for the `shiftIn()` function is found in the file `$ARDINC/wiring_shift.c`.

The `shiftIn()` function is used to read an eight-bit data value, one bit at a time, from an external device, for example, a shift register.

The operation may be carried out with the most significant bit (MSB) being shifted in first or with the least significant bit (LSB) first, according to how the external device sending the data is configured to send it.

The Arduino only requires two pins and can read eight bits or more, depending on the number of bits the device wishes to send to the Arduino. The pins required are

- A data pin, which is the pin that the bits in the external device will be presented on, ready to be read by the Arduino
- A clock pin, which is set `HIGH` to signal the external device that the Arduino is ready to receive a single bit of data and held `LOW` when the Arduino doesn't wish to read the data or has just finished reading a bit

The source code for the `shiftIn()` function is shown in Listing 3-24.

Listing 3-24 The shiftIn() function

```
uint8_t shiftIn(uint8_t dataPin,
                uint8_t clockPin,
                uint8_t bitOrder) {

  uint8_t value = 0;
  uint8_t i;

  for (i = 0; i < 8; ++i) {
    digitalWrite(clockPin, HIGH);                             (1)
    if (bitOrder == LSBFIRST)                                 (2)
      value |= digitalRead(dataPin) << i;                     (3)
    else
      value |= digitalRead(dataPin) << (7 - i);               (3)
    digitalWrite(clockPin, LOW);                              (4)
  }

  return value;                                               (5)
}
```

The function works by reading eight separate bits, one at a time, by

(1) Raising the clock pin `HIGH` to signal the external device that the Arduino is ready to receive data.
(2) Reading a one or zero from the data pin. The external device should have placed its data bit on the pin when the clock pin was raised.
(3) Accumulating the newly read bit into the value variable. Yes, there *are* two places where this happens—it depends on whether the code is being called with `LSBFIRST` or `MSBFIRST` and which of the two lines is actually executed.
(4) Bringing the clock pin `LOW` to end the reading of this one bit.
(5) The data is returned as a single eight-bit, unsigned, value.

The documentation at www.arduino.cc/reference/en/language/functions/advanced-io/shiftin/ has the following to say about the `shiftIn()` function:

> If you are interfacing with a device that is clocked by rising edges, you'll need to make sure that the clock pin is `LOW` before the first call to `shiftIn()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.
> This is a software implementation; see also the Arduino SPI library that uses a hardware implementation, which is faster but only works on specific pins.

3.4.6 Function shiftOut()

The code for the `shiftOut()` function is found in the file `$ARDINC/wiring_shift.c`.

The `shiftOut()` function is used to pass an eight-bit data value, one bit at a time, from the AVR microcontroller to an external device such as a shift register. This is carried out in software so that any Arduino pin can be used for the data pin. There is also the ability to use the hardware of the AVR microcontroller itself, which uses the Arduino's SPI library, but can only be used with certain pins on the ATmega328P.

The operation can be carried out with the most significant bit (MSB) being shifted out first or with the least significant bit (LSB) shifting first, according to how the device receiving the data desires it.

Shift registers, for example, are useful devices for reducing the number of pins required by the Arduino board for certain purposes. For example, to flash eight LEDs would require eight data pins on the Arduino, but with a shift register in place, this is reduced to two, possibly three, depending on the shift register type. The pins required are as follows:

- A data pin, which is the pin that the bits in the data to be shifted out are sent on.
- A clock pin, which is toggled HIGH and then LOW to latch the data bit into the external device.
- Optionally, but not used in `shiftOut()`, some shift register devices have an enable pin, so that the output pins on the shift register are all set to the desired state at once when all the bits have been received, and not, as and when each bit is latched into the device.

Tip
A benefit of shift registers is that they can be cascaded together. With two in a circuit, 16 LEDs can be flashed, for only two or three data pins on the Arduino. This can be extended up to 24, 32, and so on for as many shift registers, and LEDs, as you have handy.

I have also seen shift registers being used to reduce the number of Arduino pins required to drive a pair of stepper motors (www.youtube.com/watch?v=OeqQPlD3mNA).

There are tutorials on using the `shiftOut()` function, with a shift register, on the Arduino Tutorials website at https://arduino.cc/en/Tutorial/ShiftOut.

The function works by looking at each individual bit in the value to be shifted out, starting at the appropriate end, and raising the data pin `HIGH` if the bit is a one or `LOW` if it is a zero.

Once the bit has been presented on the data pin, the clock pin is toggled `HIGH` and then `LOW` to signal the external device that the data bit is valid and should be "clocked in" to the device.

On some devices, this will have the side effect of changing the output state, not always a good thing, while on others, the data are buffered internally by the device until it receives an "enable" signal, then all the data are presented on the device output pins simultaneously. This version of `shiftOut()`

does not have this ability; however, your sketch code can easily assign a pin to be used to enable this feature.

The source code for the `shiftOut()` function is as shown in Listing 3-25.

Listing 3-25 The shiftOut() function

```
void shiftOut(uint8_t dataPin,
              uint8_t clockPin,
              uint8_t bitOrder,
              uint8_t val)
{
  uint8_t i;

  for (i = 0; i < 8; i++) {
    if (bitOrder == LSBFIRST) {
      digitalWrite(dataPin, val & 1);                  (1)
      val >>= 1;
    } else {
      digitalWrite(dataPin, (val & 128) != 0);         (2)
      val <<= 1;
    }

    digitalWrite(clockPin, HIGH);
    digitalWrite(clockPin, LOW);
  }
}
```

(1) If the data are being shifted out with the least significant bit first, this line writes the state of the lowest bit, bit 0, to the data pin. The pin will be `HIGH` if the bit is 1 or `LOW` if it is 0.
(2) If the data are being shifted out with the most significant bit first, this line writes the state of the highest bit, bit 0, to the data pin. The pin will be `HIGH` if the bit is one or `LOW` if it is zero.

The documentation at www.arduino.cc/reference/en/language/functions/advanced-io/shiftout/ has the following to say about the `shiftOut()` function:

> If you are interfacing with a device that's clocked by rising edges, you'll need to make sure that the clock pin is `LOW` before the call to `shiftOut()`, e.g. with a call to `digitalWrite(clockPin, LOW)`.
> This is a software implementation; see also the SPI library, which provides a hardware implementation that is faster but works only on specific pins.

3.5 Time

This section looks into those functions which deal with timings on the Arduino. Here, we investigate the `delay()` function which holds up processing for a few milliseconds, `delayMicroseconds()` which delays for a few microseconds, and the two functions which return details of how long the sketch has been running since power on, reset, or upload time—`micros()` and `millis()`.

3.5.1 Function delay()

This function causes the sketch to pause and do almost nothing for a certain number of milliseconds.

The documentation on `delay()` at www.arduino.cc/reference/en/language/functions/time/delay/ notes that while the `delay()` function is running, you cannot

- Call `digitalRead()`, `digitalWrite()`, `pinMode()`, `analogRead()`, or `analogWrite()` to manipulate the board's pins, to read sensors, and so on
- Carry out any calculations
- Transmit data to Serial

Note
Transmission of data to the USART (the `Serial` interface) is carried out under an interrupt service routine, just like the receipt of serial data, so *should* still work. I assume—always a bad idea—that what they are meaning is simply that while `delay()` is running, your sketch cannot call the `Serial.print()` functions.

You can be sure, however, that the following will still work:

- *Receipt* of data from `Serial`, which will be saved for later use.
- Values written to PWM pins with `analogWrite()` will be *maintained*, but cannot be *changed*. Motors attached to PWM pins will still run, and LEDs will remain lit; however, their speed and/or brightness during the `delay()` will remain constant.
- Interrupt routines will work. This includes the Timer 0 Overflow Interrupt which keeps the millis counter up to date. Calls to `millis()` or `micros()`, after a call to `delay()`, will accurately reflect the passage of time for the sketch.

Note
Interestingly, `delay()` calls `micros()` to measure the delay period. The `micros()` function *does* disable interrupts while it reads `timer0_overflow_count` which is updated every time that Timer 0's eight-bit counter overflows (every 1,024 microseconds). So, technically, while `delay()` itself doesn't disable interrupts, it does cause interrupts to be disabled and enabled quite frequently within the delay loop.

The maximum delay period that can be requested is $2^{32} - 1$ milliseconds, or 4,294,967.295 seconds. This is almost 50 days; it's exactly 49 days, 17 hours, 2 minutes, 47 seconds, and 295 milliseconds! I imagine a sketch that delays for that long should really consider being put to sleep—see Section 7.4 for details.

Listing 3-26 shows the source code for the `delay()` function which has been extracted from the file `$ARDINC/wiring.c`.

Listing 3-26 The delay() function

```
void delay(unsigned long ms)
{
  uint32_t start = micros();

  while (ms > 0) {
    yield();                                             (1)

    while ( ms > 0 && (micros() - start) >= 1000) {
      ms--;
      start += 1000;
    }
  }
}
```

(1) The call to the `yield()` function, as shown in Listing 3-27, is interesting.

All that `delay()` is doing is fetching the current `micros()` value, then entering a busy loop to waste time until the required number of milliseconds has elapsed. This is why you are almost unable to carry out anything else in your sketch while there is a `delay()`—because the call to `delay()` uses all the time and CPU cycles until the delay period has finished.

In the ATmega328P-based versions of Arduino boards, `yield()` doesn't do much; in fact, it's defined as an empty function in the file `$ARDINC/hooks.c`.

Listing 3-27 The yield() function

```
/**
* Empty yield() hook.
*
* This function is intended to be used by library writers to
* build libraries or sketches that support cooperative
* threads.
*
* It's defined as a weak symbol and it can be redefined to
* implement a real cooperative scheduler.
*/

static void __empty() {
  // Empty
}

void yield(void) __attribute__ ((weak, alias("__empty")));
```

So, in these boards, calling `delay()` has an overhead of calling the `__empty()` function, but the calculation in the `delay()` function's loop takes this into account.

Other boards that can use the Arduino IDE and language, such as those based around the ESP8266, for example, have internal schedulers that *must* be kept active during long-running code; otherwise, the microcontroller may reset itself, assuming that something has hung. The AVR microcontroller has a similar feature known as the Watchdog Timer, which will be discussed later in Section 7.3.

Because the `delay()` function could take too long and starve these devices of scheduler time, the `yield()` function is defined as `weak`, and this allows the developers to define their own `yield()` function in sketches so that the processor doesn't suffer from random or strange resets.

I presume that a maker could, if they were so inclined, write a `yield()` function in an AVR microcontroller–based Arduino board and do some processing during the delay; however, it would need to be carefully considered, and whatever was being executed in the `yield()` function should be kept as short and quick as possible to avoid affecting any `delay()` loops that require reasonably accurate delay periods.

www.arduino.cc/reference/en/language/functions/time/delay/ has the following notes about the `delay()` function:

> While it is easy to create a blinking LED with the `delay()` function, and many sketches use short delays for such tasks as switch debouncing, the use of `delay()` in a sketch has significant drawbacks.
> No other reading of sensors, mathematical calculations, or pin manipulation can go on during the `delay()` function, so in effect, it brings most other activity to a halt.
> For alternative approaches to controlling timing see the `millis()` function and the sketch listed (in Listing 3-28). More knowledgeable programmers usually avoid the use of `delay()` for timing of events longer than 10's of milliseconds unless the Arduino sketch is very simple.
> Certain things do go on while the `delay()` function is controlling the Atmega chip however, because the `delay()` function does not disable interrupts. Serial communication that appears at the RX pin is recorded, PWM (`analogWrite()`) values and pin states are maintained, and interrupts will work as they should.

Listing 3-28 is the sketch mentioned in the notes. It uses the `millis()` function to time the blinking of the LED, rather than calling the `delay()` function.

Listing 3-28 Blink without using delay()

```
/*
  Blink without Delay
  created 2005 by David A. Mellis
  modified 8 Feb 2010 by Paul Stoffregen
  modified 11 Nov 2013 by Scott Fitzgerald
  modified 9 Jan 2017 by Arturo Guadalupi

  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/BlinkWithoutDelay
*/

const int ledPin = LED_BUILTIN; // the number of the LED pin
int ledState = LOW; // ledState used to set the LED
unsigned long previousMillis = 0;                        (1)
const long interval = 1000;                              (2)

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

void loop() {
  unsigned long currentMillis = millis();                  (3)

  if (currentMillis - previousMillis >= interval) {        (4)
    previousMillis = currentMillis;                        (5)

    if (ledState == LOW) {                                 (6)
      ledState = HIGH;
    } else {
      ledState = LOW;
    }

    digitalWrite(ledPin, ledState);                        (7)
  }
}
```

(1) The variable `previousMillis` holds the previous time that the LED was toggled.
(2) The delay between blinks is set here, in `interval`.
(3) The current time is obtained into `currentMillis`.
(4) Check here to see if the required interval has passed.
(5) If the LED is to be toggled, record the current time.
(6) Calculate the LED's new state—`HIGH` or `LOW`.
(7) Finally, toggle the LED.

Note
Please note I have removed most comments and wrapped longer lines of code in Listing 3-28 to preserve space on the page and to avoid the code running off the edge of the page. The original code on the Web is very well commented.

3.5.2 Function delayMicroseconds()

This function, like the `delay()` function, causes your sketch to pause and do almost nothing for a certain number of microseconds—millionths of a second. While the `delayMicroseconds()` function is running, you cannot

- Call function `digitalRead()`, `digitalWrite()`, `analogRead()`, `analogWrite()`, or `pinMode()` to manipulate the board's pins
- Carry out any calculations
- *Transmit* data to `Serial`

The following will still work while the function is delaying:

- *Receipt* of data from `Serial`, which will be saved for later use.
- Values written to PWM pins with `analogWrite()` will be *maintained*, but cannot be *changed*. Motors attached to PWM pins will still run, and LEDs will remain lit; however, their speed and/or brightness during the `delayMicroseconds()` will remain constant.
- Interrupt routines will work. This includes the Timer 0 Interrupt which keeps the `millis()` counter up to date. Calls to `millis()` or `micros()`, after a call to `delayMicroseconds()`, will accurately reflect the passage of time for the sketch.

Unlike the `delay()` function, `delayMicroseconds()` has no call to `yield()` as the delay period is most likely far too short to cause those Arduino boards, with potential scheduler problems, to reset. It is assumed that this function will only be used for fairly small delays; otherwise, the developer would be advised to call `delay()` instead.

As with `delay()`, this function simply burns CPU cycles, time, and power until the required delay period has passed; however, unlike `delay()` which allows a delay period of up to nearly 50 days, `delayMicroseconds()` takes an `unsigned int` parameter for the delay period. This is only a 16-bit variable (`delay()` uses 32 bits), so the maximum delay period is 65,536 microseconds, or 0.065536 seconds.

Listing 3-29 shows most of the source code for the `delayMicroseconds()` function which has been extracted from the file `$ARDINC/wiring.c`. Similarly to other listings, I have removed some of the larger comments and all the parts of the code which are not relevant to the ATmega328P on standard Arduino boards.

Listing 3-29 The delayMicroseconds() function

```
/*
 * Delay for the given number of microseconds.
 * Assumes a 1, 8, 12, 16, 20 or 24\,MHz clock.
 */

void delayMicroseconds(unsigned int us)
{
  // call = 4 cycles + 2 to 4 cycles to init us
  // (2 for constant delay, 4 for variable)
  // calling avrlib's delay_us() function with low
  // values (e.g. 1 or 2 microseconds) gives delays
  // longer than desired.
  //delay_us(us);                                        (1)

#if F_CPU >= 16000000L
  // for the 16\,MHz clock on most Arduino boards:

  // for a one-microsecond delay, simply return. the overhead
  // of the function call takes 14 (16) cycles, which is 1us

  if (us <= 1) return; // = 3 cycles, (4 when true)       (2)
  // the following loop takes 1/4 of a microsecond (4 cycles)
  // per iteration, so execute it four times for each microsecond of
  // delay requested.

  us <<= 2; // x4 us, = 4 cycles                               (3)

  // account for the time taken in the preceding commands.
  // we just burned 19 (21) cycles above, remove 5, (5*4=20)
  // us is at least 8 so we can subtract 5

  us -= 5; // = 2 cycles,                                      (4)
#endif

  // busy wait
  __asm__ __volatile__ (                                   (5)  (6)
    "1: sbiw %0,1" "\n\t" // 2 cycles
    "brne 1b" : "=w" (us) : "0" (us) // 2 cycles
  );
  // return = 4 cycles
}
```

(1) Interesting! It appears, from this comment, that the `delayMicroseconds()` function used to call the `delay_us()` function in the AVRLib library. It's therefore worth noting that the comment matches exactly with a warning on the Arduino Reference for the function, www.arduino.cc/reference/en/language/functions/time/delaymicroseconds/, about inaccuracies when called with low values for the delay.
I rather suspect that the problem with periods below three microseconds only applies to the old versions of the Arduino Language where that call to `delay_us()` is still being made.
The warnings from the Arduino Reference site are reproduced as follows for your information.

(2) The function begins by returning if the delay is one microsecond, or less, as that period will have elapsed by the time it gets to that point in the function. Calling the function takes one microsecond, so there's nothing more to be done.

(3) Because the actual delay is carried out by an assembly language routine, and as each iteration of the loop takes only a quarter of a microsecond, the `us` delay period needs to be multiplied by four to get the correct delay in microseconds.

(4) The `us` variable now needs to be adjusted to take account of the time that has passed doing all the preceding checks. As either 19 or 21 cycles (19 or 21 quarters of a microsecond) have gone by, 5 (microseconds) is subtracted from `us` to account for those values. This is *almost* correct as it equates to 20 cycles rather than the observed 19 or 21.

(5) The adjusted value in `us` is then passed to the assembly code and the loop executed `us` times. All that the assembly function does is to subtract one from the value in a register pair, which started off with the `us` value, then loop around doing that subtraction over and over until the `us` value reduces to zero. The compiler will pick a suitable register pair to hold the initial us value and to be decremented in the code.

(6) The assembly routine is declared `volatile` to prevent the compiler from optimizing the routine away completely—as it appears to do nothing.

The documentation at www.arduino.cc/reference/en/language/functions/time/delayMicroseconds/ notes the following about the `delayMicroseconds()` function:

> This function works very accurately in the range 3 microseconds and up. We cannot assure that `delayMicroseconds()` will perform precisely for smaller delay-times. As of Arduino 0018, `delayMicroseconds()` no longer disables interrupts.

Note
I am certain that the preceding warning about very small delays is no longer applicable and only applied to previous versions of the function which called `delay_us()`—which the current version no longer uses.

3.5.3 Function micros()

The `micros()` function returns the number of microseconds since

- The Arduino board was powered on.
- The board was reset.
- The board was reprogrammed with a new sketch and began execution.

The return value, an `unsigned long`, can hold up to 32 bits and will overflow to zero after approximately 70 minutes according to the reference notes for this function. On 16 MHz Arduino boards, the value returned by the `micros()` function is always a multiple of four microseconds. On 8 MHz Arduinos, the result is always a multiple of eight microseconds.

I wonder how approximate that 70-minute overflow period is? We know that the variable `m` can hold up to $2^{32} - 1$ microseconds, so

$$\begin{aligned} 2^{32} - 1 &= 4{,}294{,}967{,}295 \text{ microseconds} \\ \text{Divide by } 1{,}000{,}000 &= 4{,}294.967295 \text{ seconds} \\ \text{Divide by } 60 &= 71.58278827 \text{ minutes} \\ &= 71 \text{ minutes } 34 \text{ seconds } 967295 \text{ microseconds.} \end{aligned}$$

So, the counter has a wee bit more than 70 minutes before it overflows on standard Arduinos.

Listing 3-30 shows the source code for the `micros()` function which has been extracted from the file `$ARDINC/wiring.c`.

Listing 3-30 The micros() function

```
unsigned long micros() {
  unsigned long m;
  uint8_t oldSREG = SREG, t;                            (1)

  cli();                                                (2)
  m = timer0_overflow_count;                            (3)

#if defined(TCNT0)
  t = TCNT0;                                        (4)
#endif

#ifdef TIFR0
  if ((TIFR0 & _BV(TOV0)) && (t < 255))             (5)
    m++;
#endif

  SREG = oldSREG;                                   (6)

  return ((m << 8) + t)
    * (64 / clockCyclesPerMicrosecond());           (7)
}
```

(1) The status register is copied to preserve the current state of the interrupts and other flag bits.
(2) Global interrupts are disabled. This stops the interrupt routine that calculates `millis()` from executing and updating the variable `timer0_overflow_count` while it is being copied.
(3) The current Timer 0 overflow count is copied from `timer0_overflow_count` into the variable `m`. There are 256 timer clock ticks per overflow, so there is one overflow every 1,024 microseconds.
(4) The current count for Timer 0 is read from the register `TCNT0` into variable `t`. This register is incremented once every 16,000,000/64 system clock ticks, or every four microseconds.
(5) If Timer 0 has just overflowed from 255 to 0, the `TOV0` bit will be set unless the interrupt handler has completed, so the copy of `timer0_overflow_count` is incremented because the timer has just overflowed again while we were messing about with the current values.
(6) The status register is restored, thus re-enabling interrupts if they were running previously.
(7) The return value is calculated by multiplying the `timer0_overflow_count` by 256, adding the fraction, and multiplying that result by 4, the number of microseconds in each timer clock tick. See the following for the actual calculation. (I had to wrap this line to fit the page.)

The result is required to be the number of microseconds since the time began for the sketch, so we have the following:

- The value in variable `m` is the count of Timer 0 overflows, which we already know, from `init()`, occurs every 256 system clock ticks.
- The value in variable `t` is the count of the Timer 0 clock ticks since the last overflow.
- The total number of clock ticks since the sketch began is therefore $(256 * m) + t$. Shifting `m` left by eight places is the same as multiplying by 2^8 which is 256.
- The prescaler, 64, is the number of system clock ticks which occur for each timer clock tick.
- The function `clockCyclesPerMicrosecond()`, defined in `$ARDINC/Arduino.h` as $F\_CPU/1,000,000$, returns 16, and the 64 is the system clock prescaler amount. This calculation gives us four microseconds per timer clock tick.

So we need to return the value equivalent to $((m * 256) + t) * 4$ microseconds, and that's exactly what the final line does. For a standard Arduino at 16 MHz, the function `clockCyclesPerMicrosecond()` returns 16. The `clockCyclesPerMicrosecond()`

function is based on the board's F_CPU (system clock speed) and will return the correct result for micros() regardless of the actual speed of the board this code is running on.

3.5.4 Function millis()

This function returns the number of milliseconds since

- The Arduino board was powered on.
- The board was reset.
- The board was reprogrammed with a new sketch and began execution.

The source code for the function can be seen in Listing 3-31 or, if you wish, by examining the file $ARDINC/wiring.c on your system.

The init() function, which is executed at the start of any sketch, initializes the ATmega328P's Timer 0 so that every time it overflows from 255 to 0, it executes an interrupt routine to count up the milliseconds. When a sketch calls the millis() function, it simply reads the return value from a global variable named timer0_millis.

Listing 3-31 The millis() function

```
unsigned long millis()
{
  unsigned long m;
  uint8_t oldSREG = SREG;                                   (1)

  // disable interrupts while we read timer0_millis or
  // we might get an inconsistent value (e.g. in the
  // middle of a write to timer0_millis)

  cli();                                                    (2)
  m = timer0_millis;                                        (3)
  SREG = oldSREG;                                           (4)

  return m;                                                 (5)
}
```

(1) A copy of the status register is taken to preserve the current interrupt flag, plus other flags, stored there.
(2) Global interrupts must be disabled—see the comment in the code for the reason.
(3) The current value of the millis counter is read.
(4) The status register is restored, turning interrupts back on, if they were originally on.
(5) The result of the millis() function is returned.

The code must disable global interrupts before reading the value from timer0_millis. This is to prevent the counter being incremented while this code is in the middle of reading the current value.[2] As disabling the interrupts changes the status register in the AVR microcontroller, a copy is taken so that it can be restored later. After retrieving the current value, the status register is restored and the value is returned to the calling code. This minimizes the time that interrupts are disabled.

You should be aware that whenever the interrupts are disabled, and other functions such as `digitalRead()` will disable interrupts, then the `millis()` count will stop incrementing. This implies that some normal Arduino code can affect the `millis()` and `micros()` function results, however briefly.

For further information, you can read about the `init()` function in Section 2.10.

Note
According to the Arduino Reference for `millis()` at www.arduino.cc/reference/en/language/functions/time/millis/, the value returned from `millis()` will roll over to zero if the Arduino has been running for approximately 50 days. You may need to be aware of this if you are using `millis()` for anything important and make sure that your code handles situations where the value read from `millis()` at the start of something is greater than the value at the end. This would indicate a rollover has taken place.

Remember, it is only when the Arduino has been powered on, or a sketch has been running for around 50 days, not a particular part of your sketch's code.

How approximate is the 50 days value quoted?

An `unsigned long` stores data in 32 bits and so has a maximum value of $2^{32} - 1$ or 4,294,967,295 milliseconds. Divide that by the number of milliseconds in one day ($24 * 60 * 60 * 1000 = 86{,}400{,}000$), and the result is 49.71026962 days, which works out at 49 days, 17 hours, 2 minutes, and 47 seconds (plus one solitary millisecond to cause the rollover).

So it appears that the rollover isn't quite as long as 50 days; it's short by 7 hours, 22 minutes, and 53 seconds.

A warning from www.arduino.cc/reference/en/language/functions/time/millis/:

> Please note that the return value for `millis()` is an `unsigned long`. Logic errors may occur if a programmer tries to do arithmetic with smaller data types such as `int`s. Even `signed long` may encounter errors as its maximum value is half that of its `unsigned` counterpart.

3.6 Interrupts

The Arduino Language allows your sketches to set up functions which will be called automatically, whenever a certain type of event happens.

These are called interrupt service routines, or ISRs, and there are some functions in the Arduino Language which help you in setting up and using these ISRs. These are `interrupts()` and `noInterrupts()` to enable and disable interrupts for the whole board and `attachInterrupt()` and `detachInterrupt()` which link and unlink your sketch's functions to the interrupt handling system of the ATmega328P.

[2] The variable is 32 bits in size, and the ATmega328P does not have the ability to read or write data larger than 8 bits without being interrupted. If the read of the variable's value was interrupted to update it, the reading code would end up reading a "corrupted" value with some of the bits relating to the old value and some to the new value.

3.6.1 Function interrupts()

The `interrupts()` function is defined in the file `$ARDINC/Arduino.h` as per Listing 3-32.

Listing 3-32 The interrupts() function

```
#define interrupts() sei()
```

Its purpose is to enable global interrupts which it does by calling the `sei()` function. This itself is defined in `$AVRINC/interrupt.h` as the assembly language instruction `sei`, which sets the interrupt flag in the status register, thus enabling interrupts.

3.6.2 Function noInterrupts()

The `noInterrupts()` function is defined in the file `$ARDINC/Arduino.h` as per Listing 3-33.

Listing 3-33 The noInterrupts() function

```
#define noInterrupts() cli()
```

Its purpose is to disable global interrupts which it does by calling the `cli()` function. This itself is defined in `$AVRINC/interrupt.h` as the assembly language instruction `cli`, which clears the interrupt flag in the status register and thus disables global interrupts.

3.6.3 Function attachInterrupt()

The code for the `attachInterrupt()` function is found in the file `$ARDINC/WInterrupts.c` and also extracted in Listing 3-38.

This function is used to attach an *external interrupt* to a function within a sketch. Each time the interrupt fires, the function will be called. The function *must* be defined as shown in Listing 3-34.

Listing 3-34 A skeleton interrupt handling function

```
void myInterruptRoutine() {
  ...
}
```

The function must not return any value, nor does it receive any parameters. The processing it carries out should be kept as short as possible. If it requires to access any variables in the main part of the sketch, then these must be defined as `volatile`, as follows:

```
volatile int interruptFlag = 0;
```

External interrupts are limited in number on some AVR microcontrollers, and on the ATmega328P family, there are only two of these. The two pins that can be used on the ATmega328P are Arduino pins `D2` and `D3` only. These pins, corresponding to ATmega328P pins `PD2` and `PD3`, are the physical pins

4 and 5 on the device. They are able to respond to an external `stimulus` even if they are configured as `OUTPUT`. In addition, if they are `OUTPUT` pins, and a sketch changes the state of the pin, then the change may cause the interrupt to be fired.

Arduino pin `D2` is connected to the AVR microcontroller's `INT0` Interrupt, and Arduino pin `D3` is connected to the `INT1` Interrupt. These are high priority interrupts, only a `RESET` interrupt is higher, and `INT0` takes priority over `INT1` if the two arrive together.

The documentation states that the function should be called as shown in Listing 3-35.

Listing 3-35 Example usage of attachInterrupt()

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

However, `attachInterrupt()` takes the first parameter, the interrupt, as an `unsigned` eight-bit value, but `digitalPinToInterrupt()` returns a `signed` value when the pin passed to it is not `D2` or `D3`. In this case, it returns -1 which is defined as `NOT_AN_INTERRUPT`. Passing -1 to an `unsigned` will convert it to 255.

This is not a major bug, or problem, it's just inconsistent and, as purists would probably say, wrong! (And I tend to agree, this time, with the purists.)

The parameters required are as follows:

- `Pin` which must be either two for `D2` or three for `D3`. However, these values should not be passed directly to `attachInterrupt()` on the ATmega328P-based boards. The pin number should be passed to `digitalPinToInterrupt()` and the returned value then passed to `attachInterrupt()`.
- `ISR` is the name of a `void` function, taking no parameters, to be called when the interrupt fires.
- `Mode` is the definition of the external stimulus which will cause the interrupt to fire. It can be one of the following:
  - `LOW` to trigger the interrupt whenever the pin is `LOW` and will fire constantly for as long as the pin is held `LOW`. Also, if a sketch sets the pin `LOW`, then the interrupt routine will keep firing until the pin is taken `HIGH` again.
  - `CHANGE` to trigger the interrupt whenever the pin changes value from `HIGH` to `LOW` or from `LOW` to `HIGH`. This can be in response to an external device or to a sketch changing the state of the pin with `digitalWrite()`, for example.
  - `RISING` to trigger the interrupt whenever the pin goes from `LOW` to `HIGH`.
  - `FALLING` to trigger the interrupt whenever the pin goes from `HIGH` to `LOW`.

If the function is called with a `pin` value other than `D2` or `D3`, *nothing will happen*. It is advisable to use the helper function `digitalPinToInterrupt()`, because not all boards have the same pins connected to the same external interrupts. Using the helper function in this way ensures that the sketch could be run, with the code unchanged, on other boards with different AVR microcontrollers. Some other wiring changes may be necessary of course.

The helper function `digitalPinToInterrupt()` is defined in `$ARDINST/variants/standard/pins_arduino.h` and is shown in Listing 3-36.

Listing 3-36 The digitalPinToInterrupt() function

```
#define digitalPinToInterrupt(p) \
  ((p) == 2 ? 0 : ((p) == 3 ? 1 : NOT_AN_INTERRUPT))
```

It can be seen that passing pin `D2` will return zero, passing pin `D3` will return one, and any other value will return `NOT_AN_INTERRUPT` which is −1—thus, `INT0` for pin `D2` and `INT1` for pin `D3`.

`NOT_AN_INTERRUPT` is defined in `$ARDINC/Arduino.h` as

```
#define NOT_AN_INTERRUPT -1
```

An example from the Arduino Reference website on using the function `attachInterrupt()` is shown in Listing 3-37, while Figure 3-2 shows one possible breadboard circuit to use the sketch. It's a simple circuit to turn an LED on when a switch is pressed and off again when it is released.

Listing 3-37 Example sketch using attachInterrupt()

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);                         (1)
  attachInterrupt(digitalPinToInterrupt(interruptPin),
  blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);                                 (2)
}

void blink() {
  state = !state;                                              (3)
}
```

(1) The pin that the switch and interrupt are attached to is pulled `HIGH` by the internal pull-up resistor.
(2) The main loop simply sets the built-in LED to the current value of state. This will be `LOW` until the switch is pressed once, whereupon it will toggle.
(3) The interrupt service routine (ISR) changes the value of state from `LOW` to `HIGH` or from `HIGH` to `LOW` each time that pin `D2` changes its state. The state variable is always the opposite of the state on pin `D2`.

In Figure 3-2, the resistor R1 is 330 Ω and is there to limit the current drawn by the LED. It is connected from the LED's cathode to ground. The anode, which is the longest lead of the two, is attached to `D13` as the built-in LED is not very big or bright.

Given that switches bounce quite a lot, I suspected that Listing 3-37 was unlikely to work correctly, and, indeed, it did not. The LED's state is at best described as "random," regardless of what you do with the switch.

In theory, when the switch is pressed or held down, it connects `D2` to ground, this registers as a change in state for `D2`, and the ISR fires to change the `state` variable to `HIGH`. After the ISR has finished executing, the `loop()` code then turns the LED on. When the switch is released, the internal

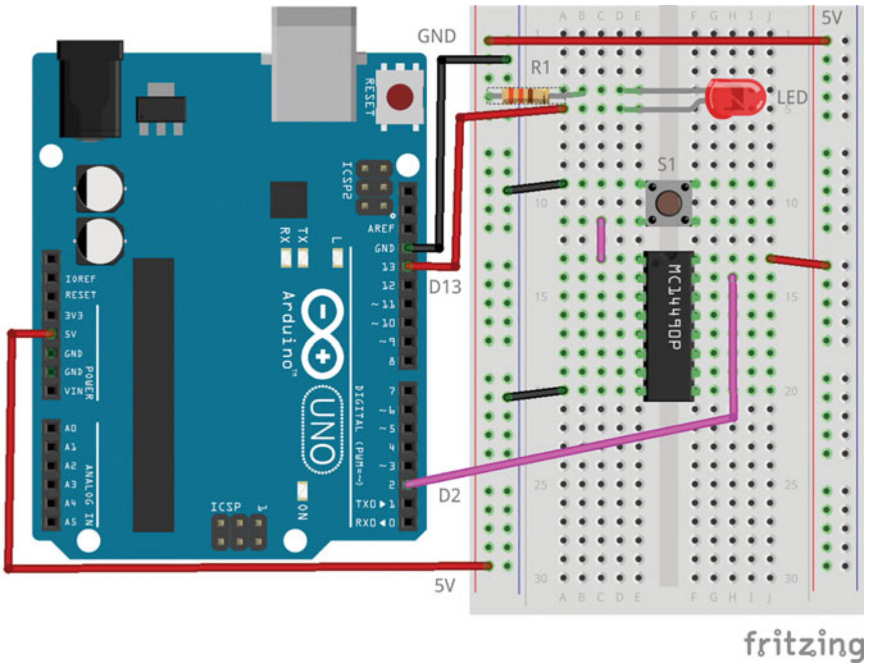


Figure 3-2 Breadboard layout for the attachInterrupt() sketch

pull-up resistors pull pin `D2 HIGH` again, which registers as a change of state, so the ISR executes again and the state variable toggles to `LOW`. This then results in the `loop()` code turning off the LED.

In practice, I have tested this code, and it's very difficult to get it to be consistent due to switch bounce. It makes and breaks numerous times before settling to a steady state. I've seen the LED remain lit while the switch was released and go out when it was pressed. This code needs a good debouncing routine or, if necessary, a couple of extra components need adding to the breadboard to debounce the switch in hardware.

I used a hardware solution and added the "MC14490 Hex Contact Bounce Eliminator" chip you can see in Figure 3-2. I connected the switch's output pin to pin 1, the `Ain` pin, on the MC14490, and pin 15, the `Aout` pin, to Arduino pin `D2`. Once the circuit was debounced, the code worked perfectly. The fully debounced circuit is shown in Figure 3-2. The MC14490 allows up to six switches to be debounced, and they are very cheap on Ebay, plus they debounce on "make" as well as "break."

Without debouncing of some kind, problems would occur if, for example, the switch changed state only *once*, so the ISR was called. While the ISR was executing, however short a time that was, interrupts were disabled, and thus *no further interrupts are able to be actioned*. However, if other interrupts were received, a flag bit was set each time to show that one or more interrupts had occurred during the execution of the ISR. When the ISR finishes executing and returns to the main code, one instruction would be executed, then the ISR would be executed again due to the flag bit being set. Regardless of how many interrupts were received during the first ISR's execution, it would only be executed again *once*.

If `state` was `LOW` and the switch was pressed, then the ISR would change the value of `state` to `HIGH` and the LED would come on. However, if any number of bounces, let's say four, occurred while the ISR was executing, then `state` should have changed four more times, so from `HIGH`—as currently set by the ISR—to `LOW->HIGH->LOW->HIGH`, and the LED should remain on.

Unfortunately, those four changes only got recorded as having occurred at least once, not how many actually occurred, so the ISR executes once and changes state from `HIGH` to `LOW`, and the LED goes off even though the button is still being pressed.

In order for the LED to follow the state of the switch, there must be no bouncing, or the ISR must be executed an even number of times, hence the requirement for a good debouncing function, or my solution in hardware, as it's almost impossible for a switch to always bounce an even number of times!

After all that excitement, the source code for the `attachInterrupt()` function follows in Listing 3-38. As with most other listings, large comment blocks and sections of code have been removed if they are not relevant to the ATmega328P devices. I have also wrapped some of the longer code lines to fit on the page.

Listing 3-38 The attachInterrupt() function

```
void attachInterrupt(uint8_t interruptNum,
                     void (*userFunc)(void), int mode) {

  if(interruptNum < EXTERNAL_NUM_INTERRUPTS) {           (1)

    intFunc[interruptNum] = userFunc;                    (2)

    switch (interruptNum) {
      case 0:                                            (3)
      #if defined(EICRA) && defined(ISC00) && defined(EIMSK)
        EICRA = (EICRA &
          ~((1 << ISC00) | (1 << ISC01))) |
          (mode << ISC00);
        EIMSK |= (1 << INT0);
      #endif
        break;

      case 1:                                            (4)
      #if defined(EICRA) && defined(ISC10) && \
        defined(ISC11) && defined(EIMSK)
          EICRA = (EICRA &
            ~((1 << ISC10) | (1 << ISC11))) |
            (mode << ISC10);
          EIMSK |= (1 << INT1);
      #endif
        break;
    } // End switch
  } // End if
}
```

(1) `EXTERNAL_NUM_INTERRUPTS` is defined as 2 in `$ARDINC/wiring_private.h` and is the total number of external interrupts available on an ATmega328P. You should note that `interruptNum` is unsigned here; however, the function `digitalPinToInterrupt()` can, if passed an invalid pin number, return `NOT_AN_INTERRUPT` which is negative. However, this

code will still work correctly as `NOT_AN_INTERRUPT` converts to 255 when passed as −1 to an `unsigned` variable.

(2) The supplied pointer to the interrupt function in the sketch is saved in a table. There are `EXTERNAL_NUM_INTERRUPTS` slots in the table.
(3) This sets up the `INT0` external interrupt and enables it.
(4) This sets up the `INT1` external interrupt and enables it.

The `attachInterrupt()` function is quite simple in operation. It saves the sketch's function addresses in a table, sets up for just this reason, and then, depending on the interrupt requested, sets the bits in the `EICRA` and `EIMSK` registers so that the interrupt will fire on the appropriate stimulus.

You should be aware that the functions passed to `attachInterrupt()` are *not* interrupt service routines. They are merely a function that will be called from the actual ISR. The real ISR is set up as per the source code in Listing 3-39, extracted from the file `$ARDINC/WInterrupts.c`.

Listing 3-39 The real ISR for attachInterrupt()

```
#define IMPLEMENT_ISR(vect, interrupt) \
ISR(vect) { \
  intFunc[interrupt](); \
}

IMPLEMENT_ISR(INT0_vect, EXTERNAL_INT_0)
IMPLEMENT_ISR(INT1_vect, EXTERNAL_INT_1)
```

This code connects the function `INT0_vect` with `INT0` and `INT1_vect` with `INT1`. It is these two functions that get called when the appropriate interrupt fires. The brief snippet of code in Listing 3-40 is that generated for the `INT0` Interrupt handler.

Listing 3-40 ISR for the INT0 Interrupt

```
ISR(INT0_vect) {
  intFunc[0]();                                      (1)
}
```

(1) This one line accesses the user-supplied ISR function in the table and executes it. For the `INT1` ISR, the `intFunc[]` array index would be one rather than zero.

www.arduino.cc/reference/en/language/functions/external-interrupts/attachinter- rupt/ has the following to say about the `attachInterrupt()` function:

> The first parameter to `attachInterrupt()` is an interrupt number. Normally you should use `digitalPinToInterrupt(pin)` to translate the actual digital pin to the specific interrupt number. For example, if you connect to pin three, use `digitalPinToInterrupt(3)` as the first parameter to `attachInterrupt()`.
> Inside the attached function, `delay()` won't work and the value returned by `millis()` will not increment as `millis()` relies on interrupts to count, so it will never increment inside an ISR. Since `delay()` also requires interrupts to work, it will not work if called inside an ISR—because interrupts are disabled while processing an ISR.
> The `micros()` function will work initially, but will start behaving erratically after 1-2 mS.
> However, the `delayMicroseconds()` function does not use any counter/interrupt, so it will work as normal.
> Serial data received while in the function may be lost.

You should declare as `volatile` any variables that you modify within the attached function. Typically global variables are used to pass data between an ISR and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as `volatile`.
Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time, other interrupts will be executed after the current one finishes in an order that depends on the priority they have.

Regarding the potential loss of serial data received, you should be aware that serial data is copied from the USART to the serial receive buffer under control of an interrupt handler. Obviously, when processing your interrupt function, interrupts are off, but the USART still runs in the background as it is not controlled by the main CPU.

There is room in the USART for two bytes only, so if your interrupt function takes too long and data are being received by the USART, it will buffer up the first two bytes and then suffer a buffer overrun error. Subsequent bytes received will be lost, until the buffer is cleared by reading the two buffered data bytes.

You must always remember the following when dealing with interrupts either in your own interrupt functions or in an ISR to handle interrupts that the Arduino Language doesn't:

- Keep the ISR or interrupt functions physically as short as possible. If you must do work as a result of an interrupt, use the ISR or function to simply set a flag, a variable value or similar, and have the main `loop()` code check it and execute the desired actions.
- Do not call `delay()` or any of the `Serial.print()` functions from inside an ISR or interrupt function. That way, dragons lie!
- All variables you wish to share with the main code should be declared `volatile` and, if necessary, when being accessed may require to be wrapped in a "critical section," which means calling `noInterrupts()` before accessing the shared data and calling `interrupts()` afterward. This, obviously, is required within the main code, not in the interrupt function.
- Finally, don't enable interrupts within an ISR or interrupt function unless you really, really know what you are doing.

3.6.4 Function detachInterrupt()

The code for the `detachInterrupt()` function is found in the file `$ARDINC/WInterrupts.c` and also in Listing 3-41.

This function detaches an interrupt function from an external interrupt. The function will have been previously attached to the interrupt by an `attachInterrupt()` call. There isn't always a need to call `detachInterrupt()` unless your sketch is finished with the ability to process the appropriate interrupt and wants to prevent any further processing of the interrupt function from taking place.

This function works by disabling the `INT0` and/or `INT1` Interrupt bits in the `EIMSK` register and blanks out the entry for the function in the function pointer table populated in `attachInterrupt()`.

The source code for `detachInterrupt()` is shown in Listing 3-41, and, as with previous listings, sections of the source that are not relevant to the standard Arduino boards have been omitted for clarity.

Listing 3-41 The detachInterrupt() function

```
void detachInterrupt(uint8_t interruptNum) {

  if(interruptNum < EXTERNAL_NUM_INTERRUPTS) {              (1)

  switch (interruptNum) {
    case 0:
      EIMSK &= ~(1 << INT0);                                (2)
      break;

    case 1:
      EIMSK &= ~(1 << INT1);                                (3)
      break;
    }

    intFunc[interruptNum] = nothing;                        (4)
  }
}
```

(1) Passing an invalid pin number to `digitalPinToInterrupt()` returns `NOT_AN_INTERRUPT`, −1, which is a `signed` value, as previously discussed. The `detachInterrupt()` function will still act correctly in that case and do nothing at all.
(2) Disable the INT0 Interrupt handler from Arduino pin `D2`.
(3) Disable the INT1 Interrupt handler from Arduino pin `D3`.
(4) The sketch's interrupt function pointer is removed from the array of interrupt functions.

The value `nothing` is defined in `$ARDINC/WInterrupts.c` as shown in Listing 3-42. It defines an empty function so that its address can be used in the interrupt function table as an "empty" value. `NULL` could have been used, but in that case detaching an interrupt function would have caused the `NULL` pointer to be dereferenced if a subsequent interrupt occurred on the appropriate pin. Dereferencing a `NULL` pointer is a bad thing to do!

Listing 3-42 The nothing() interrupt handler function

```
static void nothing(void) {
}
```

The documentation at www.arduino.cc/reference/en/language/functions/external-interrupts/detachinterrupt/ states that the `detachInterrupt()` function should be called as shown in Listing 3-43.

Listing 3-43 Example of calling the detachInterrupt() function

```
detachInterrupt(digitalPinToInterrupt(pin)); // Recommended

// or:

detachInterrupt(interrupt); // Not recommended)
```

In Listing 3-43 the interrupt parameter is the value which was returned from the call to `digitalPinToInterrupt()` prior to calling `attachInterrupt()` for this pin.

It is advisable to use the recommended calling method shown in Listing 3-43 when detaching interrupts.

3.7 Bits and Bobs

This section deals with a few "bits and bobs"—macros which allow you to do bit handling at the lowest level.

3.7.1 Macro bit()

The `bit()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define bit(b) (1UL << (b))
```

It returns the value of 2^b where "b" is the bit number. For example, calling `bit(5)` will return 32 as 2^5 is 32. You will hopefully notice that the returned value is an `unsigned long` from the initializer `1UL`, so there are 32 bits to play with, but remember to number the bits from 0 through to 31 with bit 31 being the most significant bit.

Listing 3-44 shows an example of how to use the bit() macro.

Listing 3-44 Example usage of the bit() macro

```
...
Serial.print("2 to the power 10 is: ");
Serial.println(bit(10));
...
```

This will display "2 to the power 10 is: 1024" on the Serial Monitor.

Table 3-3 lists the powers of two corresponding to an `unsigned long` variable, and these are the values that the `bit()` function will return.

Table 3-3 Bits and their values

Bit	Value	Bit	Value	Bit	Value	Bit	Value
0	1	8	256	16	65,536	24	16,777,216
1	2	9	512	17	131,072	25	33,554,432
2	4	10	1,024	18	262,144	26	67,108,864
3	8	11	2,048	19	524,288	27	143,217,728
4	16	12	4,096	20	1,048,576	28	268,435,456
5	32	13	8,192	21	2,097,152	29	536,870,912
6	64	14	16,384	22	4,194,304	30	1,073,741,824
7	128	15	32,768	23	8,388,608	31	2,147,483,648

3.7.2 Macro bitClear()

The `bitClear()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
```

The macro simply clears the requested bit, in the value passed, to zero and returns the resulting new value. For example, you could lowercase a character by clearing bit 5, as shown in Listing 3-45.

Listing 3-45 Example usage of the bitClear() macro

```
char lowerCaseA = bitClear('A', 5);
```

However, there are probably much better ways to achieve this!

3.7.3 Macro bitRead()

The `bitRead()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
```

This macro returns a one or a zero depending on the state of the bit requested in the value passed. Listing 3-46 continues the rather absurd example from before.

Listing 3-46 Example usage of the bitRead() macro

```
char upperCaseA = 'A';
char lowerCaseA;

if (bitRead(upperCaseA, 5)) {
  slowerCaseA = bitClear('A', 5);
}
```

3.7.4 Macro bitSet()

The `bitSet()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define bitSet(value, bit) ((value) |= (1UL << (bit)))
```

This macro can be called to set a specific bit in a variable or value to a one. Listing 3-47 is an example of the use of `bitSet()`.

Listing 3-47 Example usage of the bitSet() macro

```
char lowerCaseA = 'a';
char upperCaseA;

upperCaseA = bitSet(lowerCaseA, 5);
```

3.7.5 Macro bitWrite()

The `bitWrite()` macro is defined in the file `\$ARDINC/Arduino.h` as follows:

```
#define bitWrite(value, bit, bitvalue) \
    (bitvalue ? bitSet(value, bit) : bitClear(value, bit))
```

The purpose of this macro is to call either `bitSet()` or `bitClear()` for the appropriate bit in a value or variable, depending on whether it is to be set to a one or a zero. An example is shown in Listing 3-48.

Listing 3-48 Example usage of the bitWrite() macro

```
char someCharacter = 'H';

char lowerCase = bitWrite(someCharacter, 5, 0);
char upperCase = bitWrite(someCharacter, 5, 1);
```

3.7.6 Macro bitToggle()

The `bitToggle()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define bitToggle(value, bit) ((value) ^= (1UL << (bit)))
```

The purpose of this macro is to toggle a particular bit in a value. If the bit is one, it becomes zero, and if it is zero, it becomes one.

An example is shown in Listing 3-49 which shows a modified version of the standard Blink sketch's `loop()` function. Here, we are talking directly with the ATmega328P's registers where we read the `PORTB` register, toggle the bit corresponding to `LED_BUILTIN`, and write the new value back to `PORTB`.

This version of Blink reduces the size of the sketch to 714 bytes simply by removing `digitalWrite()` from `loop()`.

Listing 3-49 Example usage of the bitToggle() macro

```
void loop() {
  byte portbValue = PORTB;
  PORTB = bitToggle(portbValue, 5);
  delay(1000);
}
```

3.7.7 Macro highByte()

The `highByte()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define highByte(w) ((uint8_t) ((w) >> 8))
```

This macro returns the value in the higher eight bits of a value or variable. It returns a `uint8_t` which is guaranteed to be an unsigned eight-bit value. Listing 3-50 shows how to use the `highByte()` macro to extract the top eight bits from the value 513.

Listing 3-50 Example usage of the highByte() macro

```
Serial.println(highByte(513));
```

The code in Listing 3-50 will print “2” on the Serial Monitor. 513 is 0x201 which converts to 0b0000 0010 0000 0001. The high eight bits are 0b0000 0010 which is 2 in decimal.

3.7.8 Macro lowByte()

The `lowByte()` macro is defined in the file `$ARDINC/Arduino.h` as follows:

```
#define lowByte(w) ((uint8_t) ((w) & 0xff))
```

This macro returns the value in the lower eight bits of a value or variable. It returns a `uint8_t` which is guaranteed to be an unsigned eight-bit value. Listing 3-51 shows how to use the `lowByte()` macro to extract the bottom eight bits from the value 513.

Listing 3-51 Example usage of the lowByte() macro

```
Serial.println(lowByte(513));
```

The code in Listing 3-51 will print “1” on the Serial Monitor. 513 is 0x201 which converts to 0b0000 0010 0000 0001. The low eight bits are 0b0000 0001 which is 1 in decimal.

3.7.9 Macro sbi()

The sbi() macro is defined in the file $ARDINC/wiring_private.h as follows:

```
#ifndef sbi
  #define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif
```

In order to use this macro, you must include the header file wiring_private.h as shown in Listing 3-52.

The macro sets the requested bit, in the register, to one. You cannot call this with a value or a variable, it must be one of the lowest 32 I/O registers, as defined in the file $AVRINC/iom328p.h, which is included automatically for you by $AVRINC/io.h, itself included by $ARDINC/Arduino.h.

For example, to ensure that Arduino pin D13, the built-in LED pin, was set to OUTPUT, but without using pinMode(), you would do this in a sketch as shown in Listing 3-52, given that Arduino pin D13 is AVR pin PB5 which corresponds to bit PORTB5 in register PORTB.

Listing 3-52 Example sbi() macro call

```
#include <wiring_private.h>

void setup() {
  // Avoid pinMode() and make D13 OUTPUT.
  sbi(PORTB, PORTB5);
  ...
}
```

3.7.10 Macro cbi()

The cbi() macro is defined in the file $ARDINC/wiring_private.h as follows:

```
#ifndef cbi
  #define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
```

In order to use this macro, you must include the header file wiring_private.h as shown in Listing 3-53.

The macro clears the requested bit, in the register, to zero. You cannot call this with a value or a variable, it must be a register, as defined in the file $AVRINC/iom328p.h, which is included automatically for you by $AVRINC/io.h, itself included by $ARDINC/Arduino.h.

For example, to ensure that Arduino pin D13, the built-in LED pin, was set to INPUT, but without using pinMode(), you would do this as shown in Listing 3-53, given that Arduino pin D13 is actually AVR pin PB5 which is PORTB5 on register PORTB.

Listing 3-53 Example cbi() macro call

```
#include <wiring_private.h>

void setup() {
  // Avoid pinMode() and make D13 INPUT.
  cbi(PORTB, PORTB5);
  ...
}
```

The definition of PORTB5 (and others from PORTB0 to PORTB7) allows you to refer to the individual bits in the PORTB register. Similar definitions exist for PORTC and PORTD, as well as the three DDRx and PINx registers. All the bits in all the registers are predefined for you when you compile a sketch. This means that you don't have to use the Arduino Language all the time, especially if there is an ATmega328P feature that isn't available from the Arduino Language. For example, the Analog Comparator, which will be discussed in Section 9.1, must be accessed using the registers directly.

Tip
The two macros described here, sbi() and cbi(), should *probably* not be used as they may fail to do what is required in some circumstances. This is because the ATmega328P has some of its I/O registers outside the range of addresses that these two instructions can access. Both sbi() and cbi() can only access the lowest 32 of the various I/O registers in the ATmega328P.

You *might* get away with it, but then again, you might not. You have been warned. It is especially galling that, by default, the C/C++ compiler doesn't give any error messages if you do try to access a register that is out of range. Be careful.

4 Arduino Classes

This chapter investigates the various C++ classes supplied as part of the Arduino Language and which help—in most cases—to make the programmer's life easier when using features of the Arduino board (and the ATmega328P) such as the `Serial` interface.

Note
In the remainder of this chapter, I will not be showing *all* the source code for the various Arduino classes. Some of these classes have a large amount of code, and the book would become very large and unwieldy as a result. I will be explaining the important parts though.

4.1 The Print Class

The `Print` class is found in the two files `Print.h` and `Print.cpp`, both located in the `\$ARDINC` directory. A tutorial on implementing your own classes which inherit from `Print` can be found online at https://playground.arduino.cc/Code/Printclass/.

The `Print` class will be inherited by descendant classes and provides the ability to call `print()` and `println()` in those classes. It is used by, among others, the `Serial` interface and the `Printable` class. By far, the vast majority of the functions exposed by this class boil down to internal calls to a virtual function named `write()`, which has to be provided by the descendant class as it is declared `pure virtual` in the `Print` class.

Descendants of the `Print` class inherit from `Print` the ability to call `print()` and `println()` to "print" some information from within the descendant class. The `Print` class allows the following data types to be "printed" using `print()`:

- `String` class variables
- Data stored in Flash memory
- `char` variables and `char` arrays
- `int` variables
- `long` variables
- `double` variables
- Class variables descending from the `Printable` class

The `println()` function allows all of the preceding data types, plus `void` parameters, which simply means a call to `println()` without passing any parameters. This simply "prints" a line feed to the interface. A line feed in this case is a Windows-style line feed consisting of a carriage return and a line feed—ASCII characters 13 and 10.

Note
I use "print" in quotes as the descendant class in the hierarchy will output the bytes as appropriate. The `Serial` interface, for example, "prints" bytes to a buffer which will then be sent to the USART for transmission, while the `LiquidCrystal` library class displays the characters on screen. Other interfaces will most likely have their own manner of "printing," which could mean sending the data down the network cable to the Internet or to a file on an SD card and so on.

Listing 4-1 shows the four `virtual` functions defined in the `Print` class.

Listing 4-1 The Print class virtual functions

```
virtual size_t write(uint8_t) = 0;                             (1)

virtual size_t write(const uint8_t *buffer, size_t size);(2)
virtual int availableForWrite() { return 0; }                  (3)
virtual void flush() { /* Empty */ }                           (4)
```

(1) Only the `write(uint8_t)` is a `pure virtual` function and, as such, must be implemented by a descendant class. In the file `\$ARDINC/HardwareSerial.cpp`, for example, the `write(uint8_t)` function sends a single `unsigned char` to the USART's transmit buffer. The `Serial` interface descends from the `Stream` class, which itself descends from the `Print` class, and as long as at least one class in the hierarchy implements the `write(uint8_t)` function, the code should compile.
(2) If required, descendant classes may implement this function which should be called to provide a manner of "printing" entire `char` arrays. The `Print` class provides a default implementation of this function, which simply sends each character in the array to the `write(uint8_t)` function.
(3) This function defaults to returning zero to indicate that a single call to `write()` may block. This obviously depends on the interface in use. The descendant classes may wish to re-implement this function if, for example, they use some form of buffering which helps prevent blocking. Other nonzero values indicate that the call to `write()` may not block. The Serial class has done this in file `\$ARDINC/HardwareSerial.cpp` as it buffers data for transmission, and also for receipt, and uses interrupts to transmit and receive data from/to the buffers.
(4) The `flush()` function is provided for backward compatibility with older versions of the Arduino Language. In the current version, 2.1.0, it is an empty function which does nothing. Classes may re-implement this function if necessary—as the `Serial` class does.

4.1.1 Class Members

The `Print` class defines the following public members. They are used when outputting numeric data in bases other than the default, base 10.

`BIN`	Defined as the value 2. Used when outputting data in binary, or base 2.
`OCT`	Defined as the value 8. Used when outputting data in octal, or base 8.
`DEC`	Defined as the value 10. Used when outputting data in decimal, or base 10, the default.
`HEX`	Defined as the value 16. Used when outputting data in hexadecimal, or base 16.

The following functions are exposed by this class:

- `Print()` is the constructor. This does very little other than setting the error flag to show that, so far, no errors have occurred.
- `size_t print()` sends data to the output without a trailing newline. This can be used to "print" many different data types.
- `size_t println()` is identical to `print()` but terminates the output with a Windows-style carriage return and line feed.
- `int getWriteError()` returns any error codes from a `write()` function call.
- `void clearWriteError()` clears any existing write error codes.
- `size_t write(const char *str)` may be overridden by descendant classes. This is the default function to "print" a character array.
- `size_t write(const uint8_t *buffer, size_t size)` "prints" an array of a known size of unsigned characters. This may also be overridden by descendant classes.
- `size_t write(const char *buffer, size_t size)` simply calls the `write (const uint8_t *buffer, size_t size)` function to "print" a signed character array.
- `int availableForWrite()` returns zero if the descendant class's `write()` function will, or may, block. This may be overridden by descendant classes.
- `void flush()` flushes the output buffer to ensure that all data are correctly written. This may be overridden by descendant classes.

Tip
If your own class which inherits from the `Print` class has defined, as it must, the pure virtual write() function, then your class can send bytes to whatever it needs to. If your class header file includes `using Print::write;`, it will also be able to use the `Print` class's functions which write other data types. As these all call down to the `virtual write()` function eventually, it will call your class's own `write()` function, so you get the ability to "print" data types other than a single `unsigned char` for free.

4.1.2 Using the Print Class

An example of the use of the `Print` class is in the library for the Adafruit LiquidCrystal display. If you have installed it, the library can be found in the two files `src/LiquidCrystal.h` and `src/LiquidCrystal.cpp` located in your sketchbook location, under `libraries/LiquidCrystal`.

The following description includes only the relevant parts of the code—those which show the use of the `Print` class by the LiquidCrystal library.

In the header file, `LiquidCrystal.h`, we see the code shown in Listing 4-2.

Listing 4-2 LiquidCrystal.h

```
#include "Print.h"                                   (1)

  ...

class LiquidCrystal : public Print {                 (2)
public:
  ...
  virtual size_t write(uint8_t);                     (3)
  ...
  using Print::write;                                (4)

private:
  void send(uint8_t, uint8_t);
  void write4bits(uint8_t);
  void write8bits(uint8_t);
  void pulseEnable();
  ...
}
```

(1) When writing a library, you are responsible for including all the required header files. The IDE doesn't do this for you anymore! The `Print.h` file is required as we are inheriting from the `Print` class.
(2) The `LiquidCrystal` class inherits from the `Print` class. This gives objects of the `LiquidCrystal` class the ability to call `print()` and `println()`, which are member functions in the base class, `Print`.
(3) Descendant classes, inheriting from `Print`, must define the `write()` function as it is declared as `pure virtual` in the `Print` class.
(4) Descendant classes don't need to redefine all the other `write()` functions which "print" different data types if they use this line of code. This gives access to all the functions in the `Print` class, but each will call out to the descendant class's `write()` function.

We now need to look into the `LiquidCrystal.cpp` file to see how this class has implemented the requirements of the `Print` class. Listings 4-3, 4-4, and 4-5 show the relevant extracts from the source code.

Listing 4-3 LiquidCrystal's write() function

```
inline size_t LiquidCrystal::write(uint8_t value) {
  send(value, HIGH);
  return 1; // assume success
}
```

This is the function that all of `Print`'s descendant classes must implement themselves. In the `LiquidCrystal` class, this simply makes a call to the `send()` function—see Listing 4-4—which is declared as `private` in the header file, `LiquidCrystal.h`. I note from the comment that this function always assumes that nothing went wrong. Hmmm.

Listing 4-4 LiquidCrystal's send() function

```
void LiquidCrystal::send(uint8_t value, uint8_t mode) {
  digitalWrite(_rs_pin, mode);

  // if there is a RW pin indicated, set it low to Write
  if (_rw_pin != 255) {
    digitalWrite(_rw_pin, LOW);
  }

  if (_displayfunction & LCD_8BITMODE) {
    write8bits(value);
  } else {
    write4bits(value>>4);
    write4bits(value);
  }
}
```

If we ignore the code that's setting up the pins to enable the sketch to write to the display, we eventually end up at the calls to `write8bits()` and `write4bits()`. LiquidCrystal displays can be configured to receive eight bits of data from the Arduino or just four bits—some displays only cope with four bits. For this explanation, I shall concentrate on eight-bit displays—the main difference being that four-bit displays need two data writes to receive a character, while eight-bit displays get the whole character in one write.

We can easily see from Listing 4-4 that the `send()` function sets the display to receive data, then calls out to the `write8bits()` function, passing over the data which is to be displayed. Listing 4-5 shows the full code for the called `write8bits()` function.

Listing 4-5 LiquidCrystal's write8bits() function

```
void LiquidCrystal::write8bits(uint8_t value) {
  for (int i = 0; i < 8; i++) {
    digitalWrite(_data_pins[i], (value >> i) & 0x01);
  }

  pulseEnable();
}
```

That's all there is to it! The passed data, in parameter `value`, is used to set the bits on the eight data lines for the display. (Other displays can use the I²C interface which uses a lot fewer data lines; see the `LiquidCrystal_I2C` library for details.)

The code in Listing 4-5 sets a data pin `HIGH` if the current bit being examined in the `for` loop is set and `LOW` if it is clear. Once all eight data pins have been set correctly, `pulseEnable()` is called to, ahem, pulse the enable pin for the display and latch the data into the display whereupon it will be "printed" to the LCD screen.

The LiquidCrystal library provides an example. In the Arduino IDE, go to File ▹ Examples ▹ LiquidCrystal ▹ HelloWorld to load the example sketch which is shown, in full but minus all comments, in Listing 4-6 and is also online with circuit diagrams at www.arduino.cc/en/Tutorial/HelloWorld.

Listing 4-6 The HelloWorld example sketch

```
#include <LiquidCrystal.h>                                  (1)

const int rs = 12, en = 11,
          d4 = 5, d5 = 4,
          d6 = 3, d7 = 2;                                   (2)

LiquidCrystal lcd(rs, en, d4, d5, d6, d7);                  (3)

void setup() {
  lcd.begin(16, 2);                                         (4)
  lcd.print("hello, world!");                               (5)
}

void loop() {
  lcd.setCursor(0, 1);                                      (6)
  lcd.print(millis() / 1000);
}
```

(1) We must include the library's header file; normally, this is done with the Sketch ▹ Include Library ▹ Liquid Crystal menu option.
(2) Here, some constants are defined for the various pins that the `LiquidCrystal` objects require. In this case, we are only using four bits of data, not eight. The `rs` pin is the display's register select pin, `en` is the enable pin—used to latch data onto the display—while `d4`, `d5`, `d6`, and `d7` are the four data pins.
(3) An object of type `LiquidCrystal` is declared and initialized here. The various pins to be used are passed to the constructor. On return, `lcd` is our object through which we can manipulate the display.
(4) The display is further initialized to two rows of 16 characters. Other options are available; my own display has four rows of 20 characters, but it works happily in either mode.
(5) This is what we have been working up to. The `lcd` object can call the `Print` class's `print()` function. Within the `Print` class, the function which prints an array of `char` will be called and that will pass each character to the `LiquidCrystal` class's `write()` function. The characters in the array are then printed to the display using the code in Listings 4-3 through 4-5.
(6) The `loop()` code just sets the cursor to the start of the second line (row number from 0) and uses the `print()` function to print the number of `millis()` that have passed since the sketch began. Here, the value to be printed is an `unsigned long`, and within the `Print` base class, that call will work its way down to the `LiquidCrystal` class's `write()` function in the manner described previously.

4.2 The Printable Class

The Printable class can be used by your own classes, using inheritance, to allow them to "print" their own class data using the Print class functions print() and println() to interfaces which themselves inherit from the Print class, for example, the Serial interface. This is different from calling the print() or println() function as we saw in the previous chapter, for example:

```
Serial.print(12345);
```

where you only pass a *data value* to be "printed"; the Printable class allows the *entire object* to be "printed" with a single call:

```
Serial.print(aPerson);
```

In this case, the printTo() function would handle the streaming of the aPerson object's data. For example, it might print the name, address, and phone numbers of the specific person. We are, of course, assuming that aPerson is an object of some class that describes a person.

The Printable class has no implementation file, a cpp file; only the header file, \$ARDINC/Printable.h, is required. This file can be seen in Listing 4-7, which is all that is required to make it possible for your own classes to print themselves. The code in Listing 4-7 is the entire source code of the header file, but as usual, I have removed a few comments for brevity.

Listing 4-7 The Printable.h header file

```
#ifndef Printable_h
#define Printable_h

#include <stdlib.h>

class Print;                                                 (1)
class Printable
{
  public:
    virtual size_t printTo(Print& p) const = 0;              (2)
};

#endif
```

(1) This is a forward declaration of the Print class. It can be done without including the Print.h header file as we don't need anything from that class within this header file other than to know that a class named Print exists.
    In the case being examined here, the print.h header isn't that large, so the time saved by not having to read it in is insignificant. Other, larger class header files may not be so forgiving and will extend your compilation times.
(2) The Printable class cannot be instantiated by itself as it has only pure virtual members. It must be used as an ancestor class, and the descendant classes must implement the printTo() function.

To use the Printable class, therefore, your classes have to

- Include the Printable.h header file
- Inherit from the Printable class
- Implement a function named printTo which takes a reference to a Print class and returns a size_t value which is the number of bytes printed

That's all there is to it; your class can now be printed over the Serial interface, or similar, provided that the specific interface inherits from the Print class.

4.2.1 An Example Printable Class

The source code shown in Listings 4-8 and 4-9 is that of a very simple class which has been set up to allow itself to be sent over, for example, the Serial interface. For a more useful, but more complicated, example, have a look at the files \$ARDINC/IPAddress.h and \$ARDINC/IPAddress.cpp which are used by Arduino boards that have, for example, Ethernet interfaces built in.

Listing 4-8 is the header file, Person.h, for a simple class named Person, which has Printable as an ancestor class. You can clearly see that it has included the Printable.h header file, as required, and declares a public member function printTo().

Listing 4-8 Printable example class header

```
#ifndef Person_h
#define Person_h

#include <Printable.h>

class Person : public Printable {
  private:
    String _foreName;
    String _surName;
  public:
    // Constructors:
    Person(const String foreName, const String surName);
    Person(const char *foreName, const char *surName);
    Person();

    // Printable requires this:
    virtual size_t printTo(Print& p) const;
};

#endif // Person_h
```

I've given this simple class a set of three constructors so that objects of this type can be instantiated by passing char arrays or String values. If nothing is passed, then a default will be used. This is explained in Listing 4-9 which shows the file Person.cpp which implements the new class.

It simply copies any supplied parameters, String or char arrays, into the two class member Strings and implements the code to facilitate the mandatory printTo() function. If nothing is passed to the constructor, you will get my name as the default.

Listing 4-9 Printable example class implementation

```
#include <Print.h>
#include "Person.h"

// Constructor 1:
Person::Person(const String foreName, const String surName) :
  _foreName(foreName),
  _surName(surName) {
};

// Constructor 2:
Person::Person(const char *foreName, const char *surName) :
  _foreName(String(foreName)),
  _surName(String(surName)) {
};

// Constructor 3:
Person::Person() :
  _foreName("Norman"),
  _surName("Dunbar") {
};

// This is for Printable:
size_t Person::printTo(Print &p) const {
  size_t bytesPrinted = 0;
  bytesPrinted += p.print(_surName);
  bytesPrinted += p.print(", ");
  bytesPrinted += p.print(_foreName);

  return bytesPrinted;
}
```

Tip
In the constructors in Listing 4-9, the initialization of the member variables might look strange; however, I'm reliably informed by those in the know about C++ that this is deemed the correct way to initialize member variables. Who am I to argue?

We can see from the code that the printTo() function accepts a reference to a class object which must be a descendant of a Print class and calls its print() function three times, passing over data from the private members of the class. The function returns the number of bytes printed, which is a requirement.

Now that the class has been defined, it can be used as shown in Listing 4-10. This is a brief sketch showing how Printable class descendants can stream themselves over the interface to any `Print` class descendants. In this case, I'm using the `Serial` interface. In the example in Listing 4-10, the `Serial` object is passed to the `me` and `wife` objects' `printTo()` function as the `Print` class descendant. C++ can at times mess with your head! This line of code:

```
Serial.println(me);
```

is effectively equivalent to this line:

```
me.printTo(Serial);
```

However, the first version is how it should be done.

Listing 4-10 Printable example class usage

```
#include "Person.h"

// Declare a class object.
Person me;
Person wife("Alison", "Dunbar");

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

void loop() {
  Serial.print(me);
  Serial.print(" + ");
  Serial.println(wife);

  delay(1000);
}
```

If you upload the code to an Arduino board, then you should see "Dunbar, Norman + Dunbar, Alison" written back to the Serial Monitor—over and over again. You can, if you wish, substitute your own name(s) in the code shown in Listing 4-10.

It is not mandatory to use `Serial`. Any ancestor of a `Print` class will suffice. In the previous chapter, we saw how the LiquidCrystal library descended from `Print`, which means that the `me` and `wife` objects in the code could be passed to a `LiquidCrystal` class object, such as `lcd`, as was used in the example in Listing 4-6.

4.3 The Stream Class

The Stream class is the base class which defines the data reading functions, Serial.read(), for example. Stream supplies the reading features complementary to the Print class's writing features.

The source for the Stream class can be found in \$ARDINC/Stream.h and \$ARDINC/Stream.cpp.

The Stream class, like Printable, is designed to be used as a base class and allows character, or binary, "streaming" of data into a descendant class's variables and so on. It cannot be instantiated on its own. The descendant classes are expected to implement the three pure virtual functions detailed in Listing 4-11.

Listing 4-11 The Stream class's pure virtual functions

```
virtual int available() = 0;                          (1)
virtual int read() = 0;                               (2)
virtual int peek() = 0;                               (3)
```

(1) The available() function is required to inform the descendant class, Serial, for example, that the underlying stream has data available to be read. The function returns the number of bytes that have already been *received* by the underlying stream but which have yet to be *read* by the sketch.
(2) The read() function is implemented to allow the descendant classes to physically read the available data. This function will remove the data it reads from the input stream.
(3) The peek() function is implemented to allow the descendant classes to take a sneaky peek at the available data, but in a nondestructive manner. The data may subsequently be read by the read() function.

As Stream descends from the Print class, any class that inherits from Stream also inherits the features of the Print class. This will require the descendant class to implement the pure virtual function write() from the Print class, in addition to the requirements of the Stream class.

Examples of Arduino classes which descend from Stream are

- HardwareSerial: The standard Serial interface.
- USBAPI: Serial interface for the Leonardo boards. These use USB serial as opposed to the ATmega328P's TX and RX pins.
- Client: Part of the software for the various Ethernet shields.
- Ethernet: A networking library.
- SD: A third-party library for accessing SD cards.

4.3.1 Class Members

Listing 4-12 shows how the Stream class defines the LookaheadEnum enumeration, which is used in some of the following functions, when scanning for numeric values.

Listing 4-12 The Stream class's LookaheadMode enumeration

```
enum LookaheadMode{
  // All invalid characters are ignored.
  SKIP_ALL,

  // Nothing is skipped, and the stream is not
  // touched unless the first waiting character is valid.
  SKIP_NONE,

  // Only tabs, spaces, line feeds & carriage returns
  //are skipped.
  SKIP_WHITESPACE
};
```

The class also defines the following `public` functions, in addition to the definitions and functions exposed by the `Print` class, from which it inherits.

- `Stream()` is the constructor. It sets the default timeout to one second (1,000 milliseconds). The timeout is used in many of the following functions to limit the scanning process—to prevent code hangups or blocking if the data in the stream, for example, has not fully arrived.
- `void setTimeout(unsigned long timeout)` sets the maximum timeout, in milliseconds, to wait for stream data.
- `unsigned long getTimeout(void)` returns the current timeout for the stream.
- `int available()` must be overridden in descendant classes. It returns the number of bytes which have been received by the stream, but which have yet to be read by the sketch.

The following `Stream` functions return `TRUE` if the required target was found, `FALSE` if not or if the scan timed out. The scan is destructive in that it removes data from the stream's internal buffers while scanning.

- `bool find(char *target)`
- `bool find(uint8_t *target)`
- `bool find(char *target, size_t length)`
- `bool find(uint8_t *target, size_t length)`
- `bool find(char target)`

The following `Stream` functions return `TRUE` if the required target was found, `FALSE` if not or if the scan timed out. The scan is destructive in that it removes data from the stream's internal buffers while scanning; however, these scans end at the first occurrence of the `terminator` *string*. Characters beyond the terminating string will be safe, for now!

- `bool findUntil(char *target, char *terminator)`
- `bool findUntil(uint8_t *target, char *terminator)`
- `bool findUntil(char *target, size_t targetLen, char *terminate, size_t termLen)`
- `bool findUntil(uint8_t *target, size_t targetLen, char *terminate, size_t termLen)`

The following `Stream` functions return numeric values from the stream:

- `long parseInt(LookaheadMode lookahead = SKIP_ALL, char ignore = NO_IGNORE_CHAR)`
- `float parseFloat(LookaheadMode lookahead = SKIP_ALL, char ignore = NO_IGNORE_CHAR)`

The following `Stream` functions return -1 if there was a timeout, or they return a single character from the stream. Both of these functions are required to be implemented in a `Stream`'s descendant class:

- `int peek()` returns characters from the stream without removing them from the internal buffer for the stream.
- `int read()` returns characters from the stream and removes them from the internal buffer for the stream.

The following functions return the number of characters read from the stream while copying the data read into a buffer. If a timeout occurs, -1 will be returned, and the contents of the buffer will be undefined. Only `length` characters maximum will be copied into the buffer. If no valid data are found, then zero will be returned.

The `until` versions of the functions stop scanning the stream's buffer when the terminator *character* is read:

- `size_t readBytes( char *buffer, size_t length)`
- `size_t readBytes( uint8_t *buffer, size_t length)`
- `size_t readBytesUntil( char terminator, char *buffer, size_t length)`
- `size_t readBytesUntil( char terminator, uint8_t *buffer, size_t length)`

The following two functions read an Arduino `String` class variable from the stream. The until version stops scanning the stream's buffer when the terminator *character* is read.

- `String readString()`
- `String readStringUntil(char terminator)`

In the `Stream` class, the vast majority of the `public` functions eventually find their way down to the descendant class's `read()`, `peek()`, or `available()` functions. For this reason, I will not be describing all of the preceding functions—I suspect you would get bored very quickly—only the ones which do the actual work.

Most of the preceding functions are simply wrappers around a slightly lower-level function which does a similar thing or takes slightly different parameters. For example, Listing 4-13 shows the two separate `find()` functions.

Listing 4-13 Various Stream class cascading find() functions

```
// find returns true if the target string is found

bool Stream::find(char *target)
{
  return findUntil(target, strlen(target), NULL, 0);      (1)
}

// reads data from the stream until the target string of
// given length is found returns true if target string is
// found, false if timed out

bool Stream::find(char *target, size_t length)
{
  return findUntil(target, length, NULL, 0);              (2)
}
```

(1) Starting with a simple `find()` with a single `char` array parameter, it does nothing except call down to `findUntil()`, passing a few more parameters. `FindUntil()` is shown in Listing 4-14.
(2) This is a different `find()` function which takes an additional `length` parameter to limit the search to that number of characters. Again, it passes control down the ladder to the same `findUntil()` as noted earlier.

So far, so good; the called `findUntil()` functions from Listing 4-13 are listed in Listing 4-14. We are not done climbing the ladder yet!

Listing 4-14 Various Stream class cascading findUntil() functions

```
// as find but search ends if the terminator string is found
bool Stream::findUntil(char *target, char *terminator)
{
  return findUntil(target, strlen(target),
                   terminator, strlen(terminator));       (1)
}

// reads data from the stream until the target string of
// the given length is found search terminated if the
// terminator string is found.
// returns true if target string is found, false if terminated
// or timed out
bool Stream::findUntil(char *target, size_t targetLen,    (2)
                       char *terminator, size_t termLen)
{
  if (terminator == NULL) {                               (3)
    MultiTarget t[1] = {{target, targetLen, 0}};
    return findMulti(t, 1) == 0 ? true : false;
  } else {                                                    (4)
    MultiTarget t[2] = {{target, targetLen, 0},
                        {terminator, termLen, 0}};
    return findMulti(t, 2) == 0 ? true : false;
  }
}
```

(1) This version of `findUntil()` accepts a pair of parameters, so is not called from the `find()` functions in Listing 4-13. It does, however, pass control down to the same `findUntil()` as the `find()` functions do. Nearly there! This variant ends the search at the given `terminator` character or string.
(2) Here we are, finally! We have reached the `findUntil()` function that everyone eventually gets to. Regardless of which of the `find()` or `findUntil()` functions we originally called, this is where we arrive.
(3) If the terminator is `NULL`, then we have arrived from `find()` (or perhaps the first `findUntil()`). In this case, we create a `MultiTarget` array in variable `t`, with a single entry, and call yet another function, `findMulti()`, to do the actual searching.
(4) If the terminator is not `NULL`, we have been passed some text to use as the end of search marker. In this case, we create a `MultiTarget` array in variable `t`, with a pair of entries, and again call `findMulti()` to do the actual searching.

The `MultiTarget` structure is defined in `Stream.h` as a `protected` structure, alongside the `findMulti()` function, and so, as internal-only helpers, they are not described here.

The various `read()` functions are quite simple in that they call down to a protected `timedRead()` function, as shown by the example in Listing 4-15, which shows the `readBytes( char *buffer, size_t length)`.

Listing 4-15 One of the readBytes() functions

```
// read characters from stream into buffer
// terminates if length characters have been read, or
// timeout (see setTimeout)
// returns the number of characters placed in the buffer
// the buffer is NOT null terminated.
//
size_t Stream::readBytes(char *buffer, size_t length)
{
  size_t count = 0;
  while (count < length) {
    int c = timedRead();                                      (1)
    if (c < 0) break;
    *buffer++ = (char)c;
     count++;
  }

  return count;
}
```

(1) All of the various `read()` functions call out to the protected function `timedRead()`. That function, although `protected`, is quite small and is discussed in Listing 4-16 as it does require more investigation as it uses timeouts, and it is where your `Stream` descendant class finally gets accessed! The purpose is to return a single character from the underlying stream within a given timeout period.

Listing 4-16 The protected timedRead() function

```
// protected method to read stream with timeout
int Stream::timedRead()
{
  int c;
  _startMillis = millis();                              (1)
  do {                                                  (2)
    c = read();                                         (3)
    if (c >= 0) return c;                               (4)
    } while(millis() - _startMillis < _timeout);        (5)
  return -1; // -1 indicates timeout                    (6)
}
```

(1) The function has to use the `millis()` counter because calling `delay()` would not act as a timeout, but more of a block on any processing. The timeout is required to prevent the sketch hanging up, or blocking, because the underlying stream hasn't send enough data or is running too slowly. The error code passed back, −1, can be used to loop around, if necessary, and try reading data again.
(2) This `do` loop will execute for as long as the timeout has not expired.
(3) This is where your class gets to earn a living. The `Stream` class is now, finally, calling down to the descendant class's implementation of the `read()` function to fetch a single character from the stream.
(4) We have received a valid character, so we can exit the `do` loop and return the character to the calling function.
(5) On return from the descendant class's `read()` function, if nothing was retrieved, the tail end of the loop checks that the timeout has not yet expired and, if not, will resume the `do` loop for another iteration.
(6) If the timeout expired and we fell through the bottom of the `do` loop, −1 is returned to indicate the fact that we couldn't read any data from the underlying stream within the current timeout period.

4.4 The HardwareSerial Class

The `Serial` interface, in the Arduino Language, is an instance of a class known as `HardwareSerial` and provides the ability to read and write from the hardware serial port built into the ATmega328P. Other AVR microcontrollers such as the Mega 2560 have more than one hardware serial port, up to four in some devices. Only the code relating to the Uno's Serial port will be discussed here as the others are very similar.

Other devices have serial ports that connect directly to the USB port—the Leonardo, for example. These boards are not discussed here.

The HardwareSerial class is defined in \$ARDINC/HardwareSerial.h and implemented in

- \$ARDINC/HardwareSerial_private.h where the constructor and the *USART Receive Complete* interrupt handler helper function _rx_complete_irq() can be found.
- \$ARDINC/HardwareSerial.cpp where most of the public functions are implemented, alongside the *USART Data Register Empty* interrupt handler helper function, _tx_udr_empty_irq().
- \$ARDINC/HardwareSerial0.cpp where the actual ISRs USART_RX_vect() and USART_UDRE_vect() are found.
  These two ISRs call out to _rx_complete_irq() and _tx_udr_empty_irq(), respectively, to do the actual work. This is also the file where the instantiation of Serial as an instance of the HardwareSerial class is carried out.

The HardwareSerial class inherits from the Stream class and from Stream's ancestor class, Print, and this inheritance is the reason that the Serial object can read from and write to the ATmega328P's USART device. The USART is described in detail in Section 9.3.

4.4.1 Interrupt Handlers

The HardwareSerial class has two interrupt handlers, one of which will be fired whenever the USART receives a single byte, the USART Receive Complete interrupt. The other will fire whenever the USART's transmit buffer is empty and ready to be reloaded with the next byte to be transmitted; this is the USART Data Register Empty interrupt.

The two interrupt handlers are created simply to call the two helper routines. Both are implemented in the file \$ARDINC/HardwareSerial0.cpp as _rx_complete_irq(), in Listing 4-18, and _tx_udr_empty_irq() in Listing 4-20.

In the code listings that follow, only those parts relevant to the ATmega328P are listed.

4.4.1.1 USART Receive Complete Interrupt

The source code for the USART Receive Complete interrupt is as per Listing 4-17, and that listing is extracted from the file \$ARDINC/HardwareSerial0.cpp.

Listing 4-17 USART Receive data interrupt handler

```
ISR(USART_RX_vect)
{
  Serial._rx_complete_irq();
}
```

As can be seen, it simply calls out to the appropriate helper function. That function, _rx_complete_irq(), is found in \$ARDINC/HardwareSerial_private.h and is displayed in Listing 4-18. I've slightly massaged the code to get it to fit on the page.

Listing 4-18 USART Receive data interrupt helper

```
void HardwareSerial::_rx_complete_irq(void)
{
  if (bit_is_clear(*_ucsra, UPE0)) {                        (1)
    unsigned char c = *_udr;                                (2)

    rx_buffer_index_t i =                                   (3)
      (unsigned int)(_rx_buffer_head + 1) %
      SERIAL_RX_BUFFER_SIZE;

      if (i != _rx_buffer_tail) {                           (4)
        _rx_buffer[_rx_buffer_head] = c;
        _rx_buffer_head = i;
      }
  } else {
    // Parity error, read byte but discard it
    *_udr;                                                  (5)
  };
}
```

(1) This line of code checks to see if the character received had a parity error. If not, processing will be allowed to continue.

(2) The received byte is read from the USART Data Register `UDR0`.
This clears the data received flag, `RXC0`, in register `UCSR0A`, and readies the USART to receive the next byte. The USART has a two-byte internal buffer which, if it fills and another character is received, causes a data overrun error.

(3) The receive buffer head pointer is advanced by one byte. This might cause it to wrap around to the start again as this is a circular buffer. This new position is where the just received byte will be stored.

(4) If the new receive buffer head pointer is not yet the same as the current tail pointer, the byte received can be stored and the head pointer updated to the most recently stored byte in the receive buffer. If, on the other hand, the two pointers are equal, the byte just received is quietly lost.

(5) If there was a parity error, then the byte must still be read from the USART Data Register `UDR0`. This will clear the data received flag, `RXC0`, in register UCSR0A. The character read is discarded and not written to the buffer.

Note
The head pointer is the first free location in the buffer where new received data will be stored. The tail pointer is the next data byte to be read by the sketch. If the head pointer plus one equals the tail pointer, then the buffer must be full, and there is nowhere to store any further data without overwriting currently unread data.

4.4.1.2 USART Data Register Empty Interrupt

The source code for the USART Data Register Empty interrupt is as per Listing 4-19, which is extracted from `\$ARDINC/HardwareSerial0.cpp`.

Listing 4-19 USART Data Register Empty interrupt handler

```
ISR(USART_UDRE_vect)
{
Serial._tx_udr_empty_irq();
}
```

As with the receive handling code, it calls out to the appropriate helper function. That function, `_tx_udr_empty_irq()`, is located in the file `\$ARDINC/HardwareSerial.cpp` and can be seen in Listing 4-20. I have reformatted the code slightly to fit on the page.

Listing 4-20 USART Data Register Empty interrupt helper

```
void HardwareSerial::_tx_udr_empty_irq(void)
{
  unsigned char c = _tx_buffer[_tx_buffer_tail];          (1)
  _tx_buffer_tail = (_tx_buffer_tail + 1) %               (2)
                    SERIAL_TX_BUFFER_SIZE;

  *_udr = c;                                              (3)

  *_ucsra = ((*_ucsra) & ((1 << U2X0) |                   (4)
                          (1 << MPCM0))) |
            (1 << TXC0);

  if (_tx_buffer_head == _tx_buffer_tail) {               (5)
    cbi(*_ucsrb, UDRIE0);
  }
}
```

(1) The next byte to be transmitted is retrieved from the tail end of the transmit buffer. As this is an interrupt handler, then interrupts must be enabled, which is only true when there are data in the transmit buffer ready to be sent through the USART.
(2) The buffer tail pointer is updated to point at the next character in the buffer. This may cause it to wrap around to the start again.
(3) The next byte to be transmitted is stored in the USART Data Register `UDR0`, in the case of the ATmega328P. Storing a byte here automatically starts the transmission—when the previous byte has been transmitted.
(4) The Transmit Complete (`TXC0`) flag is cleared in register `UCSR0A`. This bit is automatically cleared when the USART Transmit Complete interrupt fires, but the Arduino doesn't use that interrupt; it uses the USART Data Register Empty interrupt instead, so the code must clear it manually. This line of code also preserves the `U2X0` and the `MPCM0` flags—the USART Double Speed and Multiprocessor Communications flags (see the ATmega328P data sheet for details).

The clearing of the TXC0 bit looks strange; it is already set to a 1, so if it is ANDed with another 1, it will remain as it is, then when written back to UCSR0A, it will clear it to zero. Weird? Really weird?

The other bits and flags in the UCSR0A register will be cleared by this line of code, unless they too need to be cleared by writing a 1, of course. You should note that the bits for frame error, FE0; data overrun, DOR0; and parity error, UPE0, must all be zero when writing any value to the UCSR0A register. See the data sheet for details.

(5) If the transmit buffer's head and tail pointers are the same, then the buffer is empty, interrupts are then disabled, and this stops transmission attempts.

The choice of interrupt handler is interesting here. Why not use the USART Transmit Complete interrupt rather than the USART Data Register Empty interrupt? There is much confusion about this it seems; however, the answer is relatively simple.

The UDR0 register can be empty while the transmission is still in progress. The register contains a single byte, or eight bits. The USART has to transmit a *frame* of more than eight bits—there are the start and stop bits, the parity bit if required, as well as the eight bits of data—so the USART Data Register Empty interrupt will fire and allow the next byte to be loaded into UDR0 while the previous byte is still in the process of being transmitted. This should increase performance a tiny bit, depending on how many actual bits are in a frame.

4.4.2 Class Functions and Macros

The HardwareSerial class defines the following macros and functions in the file \$ARDINC/HardwareSerial.h.

4.4.2.1 Macro SERIAL_TX_BUFFER_SIZE

This is the number of bytes to be used in the transmit buffer for the Serial interface. It is defined, provided it doesn't already have a definition, as in Listing 4-21. The buffer is set to be 16 bytes if there is less than 1 Kb of Static RAM in the AVR microcontroller or 64 bytes if there is more. The Arduino boards with the ATmega328P will use 64-byte buffers for both transmit and receive.

Listing 4-21 Definition of SERIAL_TX_BUFFER_SIZE

```
#if !defined(SERIAL_TX_BUFFER_SIZE)
  #if ((RAMEND - RAMSTART) < 1023)
    #define SERIAL_TX_BUFFER_SIZE 16
  #else
    #define SERIAL_TX_BUFFER_SIZE 64
  #endif
#endif
```

If you need to change the buffer size, there is a simple way to do this. At the very top of your main sketch file, define the new buffer size you need. Remember to make the new value a power of two.

```
#define SERIAL_TX_BUFFER_SIZE 128
```

This didn't work in the old version 1.x of the IDE, but works perfectly in version 2.

4.4.2.2 Macro SERIAL_RX_BUFFER_SIZE

This is the number of bytes to be used in the receive buffer for the `Serial` interface. It is defined, provided it doesn't already have a definition, as per the code in Listing 4-22. The buffer is set to be 16 bytes if there is less than 1 Kb of Static RAM in the AVR microcontroller or 64 bytes if there is more.

Listing 4-22 Definition of SERIAL_RX_BUFFER_SIZE

```
#if !defined(SERIAL_RX_BUFFER_SIZE)
  #if ((RAMEND - RAMSTART) < 1023)
    #define SERIAL_RX_BUFFER_SIZE 16
  #else
    #define SERIAL_RX_BUFFER_SIZE 64
  #endif
#endif
```

Once again, if you need to change the buffer size, simply define the new size required at the very top of the sketch file. The new value must be a power of two.

```
#define SERIAL_RX_BUFFER_SIZE 128
```

4.4.2.3 Typedef tx_buffer_index_t and rx_buffer_index_t

These are `typedefs` for the data type of the head and tail pointers into the transmit and receive buffers. By default, these are eight bits wide (`uint8_t`); however, if the appropriate buffers are larger than 256 bytes, then the index data types are increased to 16 bits (`uint16_t`) to cope with the larger sized buffer.

The definition of the transmit buffer index type is shown in Listing 4-23, while the receive buffer definition is shown in Listing 4-24.

Listing 4-23 Definition of tx_buffer_index_t

```
#if (SERIAL_TX_BUFFER_SIZE>256)
  typedef uint16_t tx_buffer_index_t;
#else
  typedef uint8_t tx_buffer_index_t;
#endif
```

Listing 4-24 Definition of rx_buffer_index_t

```
#if (SERIAL_RX_BUFFER_SIZE>256)
  typedef uint16_t rx_buffer_index_t;
#else
  typedef uint8_t rx_buffer_index_t;
#endif
```

Table 4-1 Configuration parameters for the Serial.begin() function

Define	Value	Description
SERIAL_5N1	0x00	5 bits, no parity, 1 stop bit
SERIAL_6N1	0x02	6 bits, no parity, 1 stop bit
SERIAL_7N1	0x04	7 bits, no parity, 1 stop bit
SERIAL_8N1	0x06	8 bits, no parity, 1 stop bit (default)
SERIAL_5N2	0x08	5 bits, no parity, 2 stop bits
SERIAL_6N2	0x0A	6 bits, no parity, 2 stop bits
SERIAL_7N2	0x0C	7 bits, no parity, 2 stop bits
SERIAL_8N2	0x0E	8 bits, no parity, 2 stop bits
SERIAL_5E1	0x20	5 bits, even parity, 1 stop bit
SERIAL_6E1	0x22	6 bits, even parity, 1 stop bit
SERIAL_7E1	0x24	7 bits, even parity, 1 stop bit
SERIAL_8E1	0x26	8 bits, even parity, 1 stop bit
SERIAL_5E2	0x28	5 bits, even parity, 2 stop bits
SERIAL_6E2	0x2A	6 bits, even parity, 2 stop bits
SERIAL_7E2	0x2C	7 bits, even parity, 2 stop bits
SERIAL_8E2	0x2E	8 bits, even parity, 2 stop bits
SERIAL_5O1	0x30	5 bits, odd parity, 1 stop bit
SERIAL_6O1	0x32	6 bits, odd parity, 1 stop bit
SERIAL_7O1	0x34	7 bits, odd parity, 1 stop bit
SERIAL_8O1	0x36	8 bits, odd parity, 1 stop bit
SERIAL_5O2	0x38	5 bits, odd parity, 2 stop bits
SERIAL_6O2	0x3A	6 bits, odd parity, 2 stop bits
SERIAL_7O2	0x3C	7 bits, odd parity, 2 stop bits
SERIAL_8O2	0x3E	8 bits, odd parity, 2 stop bits

4.4.2.4 Serial Communications Parameters

There are numerous configuration definitions for `Serial.begin()`. These are named "SERIAL _bps" where "b" is the number of bits, "p" is the parity required, and "s" is the number of stop bits required.

The USART can transmit data sizes between five and nine bits wide; however, the Arduino software doesn't permit the nine-bit data size. Parity can be "N," "O," or "E" for none, odd, or even parity. The number of stop bits is restricted to one or two. The complete set of configuration options is listed in Table 4-1.

If `Serial.begin()` is called with a single parameter, a baud rate, then the default is eight bits, no parity, and one stop bit—`SERIAL_8N1`. There are two versions of the `begin()` function in the file `\$ARDINC/HardwareSerial.h`. The default variant is shown in Listing 4-25, while Listing 4-29 shows the actual `begin()` function which does the hard work of initializing the Serial interface.

Listing 4-25 Default Serial.begin() function

```
void begin(unsigned long baud) { begin(baud, SERIAL_8N1); }
```

You can see it simply calls the overloaded `begin()` function (see Listing 4-29) with the required two parameters.

4.4.2.5 Macro HAVE_HWSERIAL0

In the case of the standard Arduino board, this macro will always be defined. There are up to three additional Serial interfaces that may exist, on the Mega boards, for example, but these are not discussed here.

HAVE_HWSERIALn macros are able to be used in code to determine whether or not the microcontroller has the specific serial port as numbered. The ATmega328P has only one serial port, so HAVE_HWSERIAL0 will return true. The Mega 2560 Arduino boards have four serial ports, so all of HAVE_HWSERIAL0 through HAVE_HWSERIAL3 will return true.

Listing 4-26 Defining Serial as extern

```
#if defined(UBRRH) || defined(UBRR0H)
  extern HardwareSerial Serial;
  #define HAVE_HWSERIAL0
#endif
```

Listing 4-26 shows how the first serial port on the ATmega328P is defined, and also the actual Serial interface, an instance of a HardwareSerial object, is declared extern. This is required as Serial is actually instantiated as an object in the file \$ARDINC/HardwareSerial0.cpp, shown in Listing 4-27.

Listing 4-27 Actual definition of Serial

```
#if defined(UBRRH) && defined(UBRRL)
  ...
#else
  HardwareSerial Serial(&UBRR0H,
                        &UBRR0L,
                        &UCSR0A,
                        &UCSR0B,
                        &UCSR0C,
                        &UDR0);
#endif
```

The Serial variable is an object of type HardwareSerial. The following functions are exposed by this class.

4.4.2.6 Constructor HardwareSerial()

This is the class constructor. Because of the different internal AVR microcontroller register names on different boards and microcontrollers, the constructor takes *pointers* to the registers required for serial communications. Listing 4-28 is extracted from the file \$ARDINC/HardwareSerial_private.h and shows the constructor.

Listing 4-28 HardwareSerial constructor()

```
HardwareSerial::HardwareSerial(
  volatile uint8_t *ubrrh, volatile uint8_t *ubrrl,
  volatile uint8_t *ucsra, volatile uint8_t *ucsrb,
  volatile uint8_t *ucsrc, volatile uint8_t *udr) :      (1)
  _ubrrh(ubrrh), _ubrrl(ubrrl),
  _ucsra(ucsra), _ucsrb(ucsrb), _ucsrc(ucsrc),
  _udr(udr),
  _rx_buffer_head(0), _rx_buffer_tail(0),
  _tx_buffer_head(0), _tx_buffer_tail(0)
{                                                   (2)
}
```

(1) This constructor is using the "colon" manner of initializing the member variables from the parameters passed to the constructor.
(2) All the initialization has been done; the body of the constructor is empty.

The preceding manner of initializing an object in the constructor is considered the correct method in modern versions of the C++ standards.

4.4.2.7 Function begin(unsigned long baud)

This function is called to commence serial communications at the specified baud rate, with a `config` of `SERIAL_8N1` for eight bit, no parity, and one stop bit communications. This function calls the overridden `begin(unsigned long, uint8_t)` function in Listing 4-29, passing the desired `baud` rate and `SERIAL_8N1`.

4.4.2.8 Function begin(unsigned long, uint8_t)

The `begin()` function is called to initialize serial communications at the desired `baud` rate and configuration. Listing 4-29 shows the code that performs the actual initialization. There are other overloaded versions of the `begin()` function which take fewer parameters; however, they all eventually arrive at the code in Listing 4-29.

The code in Listing 4-29 has been massaged slightly to fit the page.

Listing 4-29 The HardwareSerial::begin() function

```
void HardwareSerial::begin(unsigned long baud, byte config)
{
    // Try u2x mode first                           (1)
    uint16_t baud_setting = (F_CPU / 4 / baud - 1) / 2;
    *_ucsra = 1 << U2X0;

    if (((F_CPU == 16000000UL) &&                   (2)
      (baud == 57600)) || (baud_setting > 4095))
    {
      *_ucsra = 0;
      baud_setting = (F_CPU / 8 / baud - 1) / 2;
    }

    *_ubrrh = baud_setting >> 8;                    (3)
    *_ubrrl = baud_setting;

    _written = false;                               (4)
    *_ucsrc = config;                                    (5)

    sbi(*_ucsrb, RXEN0);                                 (6)
    sbi(*_ucsrb, TXEN0);

    sbi(*_ucsrb, RXCIE0);                                (7)
    cbi(*_ucsrb, UDRIE0);
}
```

(1) This assumes that double speed communications will be used and sets bit U2X0 in the UCSR0A register to enable double speed communications mode. All other bits are cleared. The baud_setting variable here is *not* the actual baud rate desired—that's in baud. The calculation here is working out a value for the USART Baud Rate Register 0 or UBRR0, which will define the actual baud rate for communications.

(2) If an older board is in use, with a clock speed of 16 MHz, and a baud rate of 57,600 is chosen, or if the baud_setting calculated earlier is 4,096 or higher on *any* board, then the double speed mode is disabled and baud_setting recalculated for the normal speed mode. There is a comment in the code which states that this line is a

> Hardcoded exception for 57600 for compatibility with the bootloader shipped with the Duemilanove and previous boards and the firmware on the 8U2 on the Uno and Mega 2560. Also, The baud_setting cannot be > 4095, so switch back to non-u2x mode if the baud rate is too low.

(3) UBRR0 is a 12-bit register—well, it's a 16-bit register, but the top four bits of the high byte are ignored. It is used as a counter for the serial clock generator. Every time that it counts down to zero, it will be reset to the value calculated in baud_setting. This is the baud rate generator for the USART. The calculated baud_setting is split into two parts and loaded into the high and low bytes of the UBRR0 register. 16-bit registers in the ATmega328P must, usually, be loaded high byte first then low byte.

(4) The _written flag is set whenever a byte is transmitted. This is used as a simple shortcut, so that calls to flush() can return quickly if no actual transmissions have taken place.

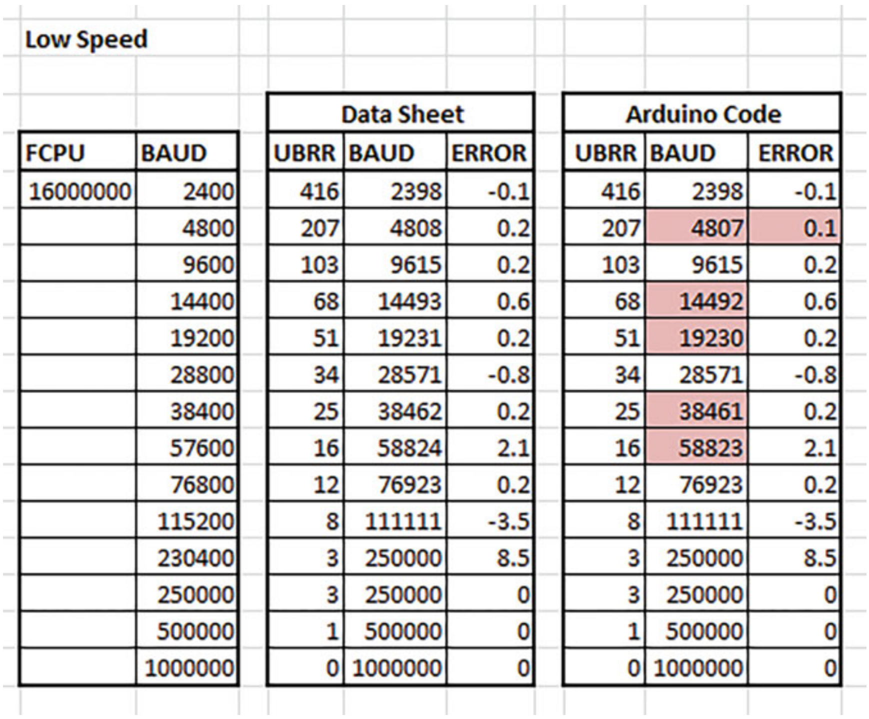
(5) The desired data width, parity, and stop bits are set up here. The default is eight bits, no parity, and one stop bit.

(6) These two lines enable data receipt and transmission. This has the effect of removing Arduino pins D0 and D1 from general use—they are now in the care of the USART.

(7) The final two lines enable the interrupts for receiving and transmitting data. This is why the Serial interface cannot be used within an interrupt handler because interrupt handlers disable interrupts while executing. The interrupts enabled are the USART Receive Complete interrupt and the USART Data Register Empty interrupt.

The Arduino code calculates the UBRR0 value (in baud_setting) differently from the data sheet. In the data sheet, the formula for double speed communications is given as this:

$$(F\_CPU/(8 * baud)) - 1$$



Low Speed

		Data Sheet			Arduino Code		
FCPU	BAUD	UBRR	BAUD	ERROR	UBRR	BAUD	ERROR
16000000	2400	416	2398	-0.1	416	2398	-0.1
	4800	207	4808	0.2	207	4807	0.1
	9600	103	9615	0.2	103	9615	0.2
	14400	68	14493	0.6	68	14492	0.6
	19200	51	19231	0.2	51	19230	0.2
	28800	34	28571	-0.8	34	28571	-0.8
	38400	25	38462	0.2	25	38461	0.2
	57600	16	58824	2.1	16	58823	2.1
	76800	12	76923	0.2	12	76923	0.2
	115200	8	111111	-3.5	8	111111	-3.5
	230400	3	250000	8.5	3	250000	8.5
	250000	3	250000	0	3	250000	0
	500000	1	500000	0	1	500000	0
	1000000	0	1000000	0	0	1000000	0

Figure 4-1 Normal speed baud rate calculations

which translates to

$$(F\_CPU/8/baud) - 1$$

The Arduino code calculates the "−1" as *part* of the division, not *after* the division, which it performs in two parts, as follows:

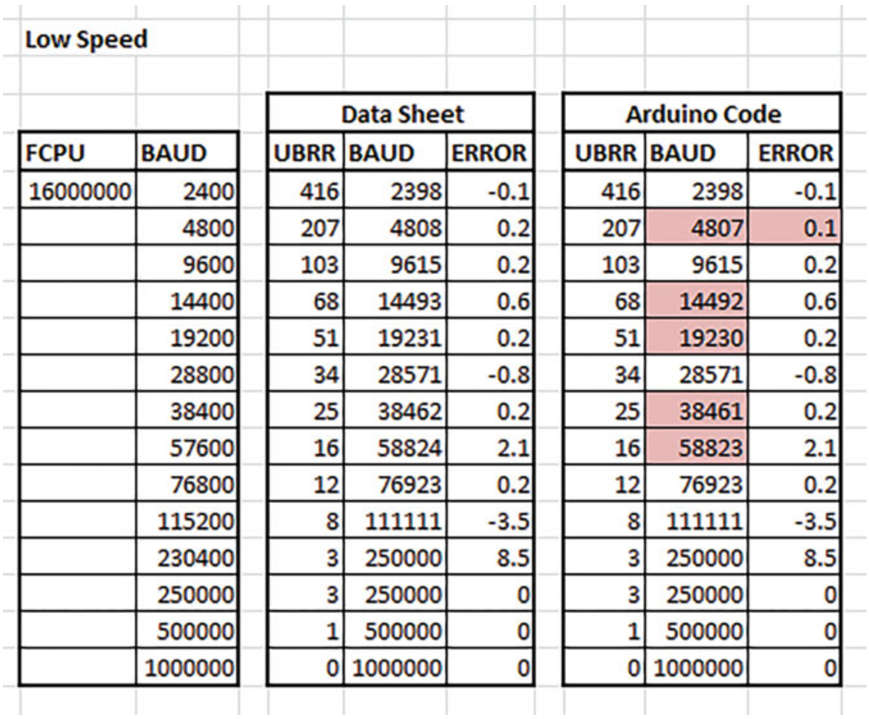
$$(F\_CPU/4/baud - 1)\,/2$$

However, both the Arduino and the data sheet *usually* agree on the final, integer, result. This applies to both low and double speed communications calculations. Figures 4-1 and 4-2 show the calculated values and error rates.

In addition to the slightly different calculation, the Arduino code works with unsigned integers, `unsigned long` and `uint16_t`, while the data sheet appears to calculate using floating-point arithmetic with rounding up or down carried out at the very end. At least, that's the only way I could get the same answers as the data sheet. I am therefore of the opinion that the data sheet is incorrect!

A similar discrepancy exists between how the Arduino and the data sheet calculates normal speed `UBBR0` settings and also error rates for the various settings.

Baud Rate Calculations The images in Figures 4-1 and 4-2 show the required values for the `UBRR0` register as calculated by the data sheet and by the Arduino code. There are separate images for the normal speed and double speed modes. Both images are from a spreadsheet which I used to perform the calculations.



Low Speed

		Data Sheet			Arduino Code		
FCPU	BAUD	UBRR	BAUD	ERROR	UBRR	BAUD	ERROR
16000000	2400	416	2398	-0.1	416	2398	-0.1
	4800	207	4808	0.2	207	4807	0.1
	9600	103	9615	0.2	103	9615	0.2
	14400	68	14493	0.6	68	14492	0.6
	19200	51	19231	0.2	51	19230	0.2
	28800	34	28571	-0.8	34	28571	-0.8
	38400	25	38462	0.2	25	38461	0.2
	57600	16	58824	2.1	16	58823	2.1
	76800	12	76923	0.2	12	76923	0.2
	115200	8	111111	-3.5	8	111111	-3.5
	230400	3	250000	8.5	3	250000	8.5
	250000	3	250000	0	3	250000	0
	500000	1	500000	0	1	500000	0
	1000000	0	1000000	0	0	1000000	0

Figure 4-2 Double speed baud rate calculations

The areas highlighted show discrepancies between what the data sheet calculates and the Arduino's result for the same calculation due to rounding. In the figures, the following apply:

- The clock speed for the AVR microcontroller is 16 MHz.
- The data sheet figures use floating-point calculations and are rounded at the very end. Rounding is up or down according to where the fractional parts are in relation to 0.5—equal or higher rounds up and lower down.
- The data sheet baud rates are again calculated from the floating-point values with rounding at the end. This to my mind at least is incorrect as the value in `UBRR0` cannot possibly be a floating-point value!
- The Arduino figures use unsigned integer values throughout with truncation downward as opposed to rounding up or down as appropriate.
- The difference between floats with fractions and unsigned integers account for the variances highlighted—even when it appears that the data sheet and the Arduino code have the same `UBRR0` figures.

The normal speed baud rates are calculated as

$$F\_CPU/16 * (UBRR0 + 1)$$

This means that the baud rate, in normal speed mode, ranges from 16 MHz/(16 $*$ (0 + 1)), which is 1,000,000 baud, down to 16 MHz/(16 $*$ (4,095 + 1)), which equals 244 baud.

The double speed baud rates are calculated as

$$F\_CPU/8 * (UBRR0 + 1)$$

This means that the baud rate, in double speed mode, ranges from 16 MHz$/(8 * (0 + 1))$, which is 2,000,000 baud, down to 16 MHz$/(8 * (4{,}095 + 1))$, which equals 488 baud.

Baud Rate Errors Just about all desired baud rates are not quite *exactly* achievable. This is because calculating the `UBRR0` value loses accuracy when the fractional parts are lost—registers don't have room for fractions after all. This means that the USART may not quite be running at exactly the baud rate requested by the sketch.

Because the `UBRR0` value may not always be exactly as calculated, then the actual baud rate calculation will not match up to the baud rate requested. The spreadsheet images in Figures 4-1 and 4-2 show error rates using the calculation from the data sheet, which is

$$Error\% = (ActualBaud/DesiredBaud - 1) * 100$$

The data sheet figures for the error rate appear to be rounded to a single decimal place at the end of the calculation.

The data sheet advises avoiding those baud rates where the calculated error rate is plus or minus 0.5% or higher.

The data sheet figures should be taken with a pinch of salt! You cannot count with register values holding floating-point values—unless you are using a floating-point unit (FPU) of course, but the AVR microcontroller doesn't have one.

Note
You should be aware that you are not limited to the baud rates noted in Figures 4-1 and 4-2. The Arduino code will accept any value for the requested baud rate and attempt to calculate a suitable value for `UBRR0`.

The value written to `UBRR0` is used as a divider of the system clock and can be anything between 0 and 4,095; it is not the baud rate, it is used to calculate the correct timings to give the required baud rate by prescaling the system clock.

Single and Double Speed Communications According to the data sheet, setting the `U2X0` bit in register `UCSR0A` will

> Reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

It goes on to state that

> Setting this bit will reduce the divisor of the baud rate divider from 16 to 8, effectively doubling the transfer rate for asynchronous communication. Note however that the Receiver will in this case only use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery, and therefore a more accurate baud rate setting and system clock are required when this mode is used. For the Transmitter, there are no downsides.

So, it appears that this bit may affect data receipt while not affecting data transmission.

4.4.2.9 Function end()

Calling the Serial.end() function disables serial communication and flushes the transmission buffer so that any bytes which were in the process of being transmitted will be allowed to complete. Interrupts for transmission and receipt of data are then disabled, and finally, Arduino pins D0 and D1 are disconnected from the USART and can now be used for normal input/output operations. The code for the end() function is as shown in Listing 4-30.

Listing 4-30 The HardwareSerial::end() function

```
void HardwareSerial::end()
{
    // wait for transmission of outgoing data
    flush();                                              (1)

    cbi(*_ucsrb, RXEN0);                                  (2)
    cbi(*_ucsrb, TXEN0);
    cbi(*_ucsrb, RXCIE0);                                 (3)
    cbi(*_ucsrb, UDRIE0);

    // clear any received data
    _rx_buffer_head = _rx_buffer_tail;                    (4)
}
```

(1) Any data currently in the transmission buffer is allowed to complete its transmission.
(2) The USART transmit and receive functions are disabled. Pins D0 and D1 return to normal Arduino input/output mode.
(3) Transmit and Receive interrupts are disabled.
(4) The receive buffer is emptied ready for subsequent receipt of data. The data may not have been read by the sketch yet, but it is now gone.

4.4.2.10 Operator bool()

The bool() function, in Listing 4-31, will return true if the specified serial port is available. It is called, for example, as in if (Serial) {...} and will only ever return false if called in a sketch which is running on a Leonardo board, for example, and the USB CDC serial connection is not yet ready.

On the standard Arduino boards, the function always returns true which can clearly be seen from Listing 4-31.

Listing 4-31 The HardwareSerial::operator bool() function

```
operator bool() {
  return true;
}
```

The code in Listing 4-31 is from a standard Arduino, obviously!

4.4.2.11 Function available(void)

This function overrides the `virtual` one in the `Stream` ancestor class and returns the number of bytes available in the receive buffer, which have yet to be read by the sketch. Listing 4-32 shows the full code for the `available()` function, and it was reformatted slightly to fit on the page.

Listing 4-32 The HardwareSerial::available() function

```
int HardwareSerial::available(void)
{
  return (
    (unsigned)(SERIAL_RX_BUFFER_SIZE +
                _rx_buffer_head -
                _rx_buffer_tail)) % SERIAL_RX_BUFFER_SIZE;
}
```

The buffer's head is where the next byte received by the USART will be placed; the buffer's tail is the next byte to be read into the sketch. The difference between the two is the number of bytes available. The calculation accounts for any wraparound that takes place when the addition of new bytes to the buffer by the USART causes the head pointer to point back at the start of the buffer while the tail pointer is still at a (now) higher address.

If, for example, the buffer was 64 bytes long at address 100 and the head pointer has wrapped around and is now pointing at address 105, while the tail pointer is pointing at address 155, the amount of data available to read is

$$\begin{aligned} &(64 + \text{head} - \text{tail}\%64) \\ = {} &(64 + 105 - 155)\ \%64 \\ = {} &(169 - 155)\ \%64 \\ = {} &14\%64 \\ = {} &14 \end{aligned}$$

So, there are 14 bytes of data not yet read by the sketch. These are in the buffer at addresses 155 through 163 and bytes 100 through 105, which, if you use your fingers like I just did, is exactly 14 bytes. Remember, the tail pointer is the first byte to be read from the buffer and passed to the sketch, while the head pointer is where the next byte read in from the USART will be stored—it is the first free location in the sketch's receive buffer.

4.4.2.12 Function peek(void)

Listing 4-33 shows the `peek()` function. This function overrides the `virtual` one in the ancestor class, `Stream`, and returns the next character that will be returned when the `read()` function—see Listing 4-34—is called. The character remains in the buffer, so this is a nondestructive read. If there are no characters in the buffer, -1 is returned.

Listing 4-33 The HardwareSerial::peek() function

```
int HardwareSerial::peek(void)
{
  if (_rx_buffer_head == _rx_buffer_tail) {              (1)
    return -1;
  } else {
    return _rx_buffer[_rx_buffer_tail];                  (2)
  }
}
```

(1) If the head and tail are equal, there's nothing in the buffer. An invalid character code, −1, is returned.
(2) The character at the tail end of the buffer is returned, without changing the tail pointer.

4.4.2.13 Function read(void)

The `read()` function, as shown in Listing 4-34, overrides the virtual function in the ancestor class, Stream, and returns the next character from the receive buffer and adjusts the tail pointer to remove the character read from the buffer. If there are no characters in the buffer, −1 is returned. Listing 4-34 was reformatted slightly to fit on the page.

Listing 4-34 The HardwareSerial::read() function

```
int HardwareSerial::read(void)
{
  if (_rx_buffer_head == _rx_buffer_tail) {              (1)
    return -1;
  } else {
    unsigned char c = _rx_buffer[_rx_buffer_tail];       (2)
    _rx_buffer_tail =
      (rx_buffer_index_t)(_rx_buffer_tail + 1) %         (3)
      SERIAL_RX_BUFFER_SIZE;

    return c;                                            (4)
    }
}
```

(1) If the head and tail are equal, there's nothing in the buffer. An invalid character code, −1, is returned.
(2) The next, unread, character is extracted from the buffer.
(3) The tail pointer is adjusted to the next character in the buffer, which may cause the tail pointer to wrap around to the first character in the buffer.
(4) The extracted character is returned to the sketch.

4.4.2.14 Function availableForWrite(void)

This function, shown in Listing 4-35, overrides the `virtual` one in the `Print` ancestor class and returns the number of bytes of free space remaining in the `Serial` interface's transmit buffer.

Listing 4-35 The HardwareSerial::availableForWrite() function

```
int HardwareSerial::availableForWrite(void)
{
  tx_buffer_index_t head;                              (1)
  tx_buffer_index_t tail;

  TX_BUFFER_ATOMIC {                                   (2)
    head = _tx_buffer_head;
    tail = _tx_buffer_tail;
  }

  if (head >= tail)
    return SERIAL_TX_BUFFER_SIZE - 1 - head + tail;    (3)

  return tail - head - 1;                              (4)
}
```

(1) The transmit buffer's head and tail pointers will be copied to these two variables, so they are declared with the same data type as the actual head and tail pointers for the buffer.
(2) Wrapping these two lines in `TX_BUFFER_ATOMIC` is necessary if the buffer size is bigger than 256 bytes as reading an 8-bit value is atomic—it cannot be interrupted—but reading a 16-bit value could be interrupted, and the value may be updated in between reading the low and high bytes. The `TX_BUFFER_ATOMIC` macro is defined as shown in Listing 4-36.
The protected code block simply copies the current values for the head and tail pointers into the two local variables. The tail pointer is the next location in the sketch's transmit buffer that will be copied to the USART's transmit register, while the head pointer is where the sketch will store the next byte sent from the sketch.
(3) If the head is ahead of the tail, this calculation returns the bytes between the head and the tail. If the head equals the tail, then the buffer is empty, and this calculation returns `SERIAL_TX_BUFFER_SIZE - 1`.
(4) If the head has wrapped back to the start of the buffer and is now behind the tail, this calculation returns the difference between them accounting for the wraparound.

The `TX_BUFFER_ATOMIC` macro is defined as shown in Listing 4-36.

Listing 4-36 The TX_BUFFER_ATOMIC macro

```
#if (SERIAL_TX_BUFFER_SIZE>256)
#define TX_BUFFER_ATOMIC ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
#else
#define TX_BUFFER_ATOMIC
#endif
```

`ATOMIC_BLOCK(ATOMIC_RESTORESTATE)` is from AVRLib and means that whatever state the interrupts were in before this block will be restored after the block. The block will disable interrupts for the duration.

If the buffer size is less than 256 bytes, the `TX_BUFFER_ATOMIC` macro expands to nothing as no special handling is required for transferring eight-bit values as they cannot be interrupted.

4.4.2.15 Function flush(void)

The `flush()` function in Listing 4-37 overrides the `virtual` function in the ancestor class, `Print`, and ensures that any data currently in the process of being transmitted is allowed to continue until completion. This empties the sketch's transmit buffer and ensures that the entire contents are indeed transmitted.

Listing 4-37 has been slightly reformatted to fit on the page.

Listing 4-37 The HardwareSerial::flush() function

```
void HardwareSerial::flush()
{
  if (!_written)                                          (1)
    return;

  while (bit_is_set(*_ucsrb, UDRIE0) ||                   (2)
         bit_is_clear(*_ucsra, TXC0)) {

    if (bit_is_clear(SREG, SREG_I) &&                     (3)
        bit_is_set(*_ucsrb, UDRIE0))

    // Interrupts are globally disabled, but the DR
    // empty interrupt should be enabled, so poll the
    // DR empty flag to prevent deadlock

    if (bit_is_set(*_ucsra, UDRE0))                       (4)
      _tx_udr_empty_irq();
  }

  // If we get here, nothing is queued anymore (DRIE is
  // disabled) and the hardware finished transmission
  // (TXC is set).
}
```

(1) If we have never transmitted a byte, since `Serial.begin()`, then there is no need to flush. This special check is needed since there is no way to force `TXC0`—the transmit complete flag bit—to 1 during initialization of the USART which could cause `flush()` to block forever if called when no data had ever been transmitted.

(2) The `while` loop will execute as long as the USART Data Register Empty interrupt remains enabled or if data is currently being transmitted by the USART. The comment at the bottom of the function shows the conditions that will be in force when the `while` loop exits, these being
- The sketch's transmit buffer is empty.
- The transmit complete bit, `TXC0`, in the `UCSR0A` register has been set.
- The `UDRIE` bit in `UCSR0B` is clear to disable the USART Data Register Empty interrupt.

(3) If global interrupts are currently disabled but the USART Data Register Empty interrupt is still enabled, then we must still have data in the transmit buffer waiting to be sent.

(4) This line explicitly calls the helper function for the USART Data Register Empty interrupt handler if the UDR0 register is currently empty and waiting for another byte.

In other words, the code ensures that even if global interrupts are not enabled, as long as data remains to be transmitted to the USART and beyond, it will indeed be transmitted.

4.4.2.16 Function write(uint8_t)

This function overrides the virtual one in the ancestor class, Print, and defines how a single unsigned char will be transmitted to the Serial interface. The write() function is split into two separate parts. The first part, the code which follows in Listing 4-38, deals with those occasions when both the transmit buffer in the sketch and the USART's data register, UDR0, are also empty.

Rather than adding the byte to be transmitted to the sketch's transmit buffer and waiting for the interrupt handler to forward it to the USART, the code in Listing 4-38 cuts out the middleman and writes the byte directly into the USART's data register for transmission. The function then returns the number of bytes written—which will always be a single byte.

The code in Listings 4-38 and 4-39 have been reformatted to fit the page.

Listing 4-38 The HardwareSerial::write() function

```
size_t HardwareSerial::write(uint8_t c)
{
  _written = true;                                      (1)

  if (_tx_buffer_head == _tx_buffer_tail &&             (2)
      bit_is_set(*_ucsra, UDRE0)) {

    // If TXC is cleared before writing UDR and the
    // previous byte completes before writing to UDR,
    // TXC will be set but a byte is still being
    // transmitted causing flush() to return too soon.
    // So writing UDR must happen first.

    // Writing UDR and clearing TXC must be done atomically,
    // otherwise interrupts might delay the TXC clear so the
    // byte written to UDR is transmitted (setting TXC)
    // before clearing TXC. Then TXC will be cleared when
    // no bytes are left, causing flush() to hang.

    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {                 (3)
      *_udr = c;                                        (4)
      *_ucsra = ((*_ucsra) & ((1 << U2X0) |
      (1 << MPCM0))
      ) | (1 << TXC0);                                  (5)
    }
    return 1;                                           (6)
  }
```

(1) This flag is used by `flush()` to determine if anything has been written yet, so it must be set any time a byte is supplied to be transmitted. In `flush()`—see Listing 4-37—this flag is used as a "quick exit" as it tells `flush()` whether or not it has work to do.
(2) This is a performance shortcut to load the passed byte directly into the USART's data register if the sketch's transmit buffer is empty—`_tx_buffer_head` equals `_tx_buffer_tail`—and the USART's data register is also currently empty. This reduces overhead and makes higher baud rates more reliable.
(3) The comment above this line explains it all. The code must be careful to not get interrupted, so is wrapped in an atomic block which will disable interrupts if necessary, make the required changes, and re-enable interrupts if they were previously enabled. The `ATOMIC_BLOCK` and `ATOMIC_RESTORESTATE` macros are defined in the depths of the AVRLib code.
(4) The code here simply writes the data byte to the USART's data register ready to be transmitted.
(5) This line updates the `USCR0A` register to preserve the state of the double speed (`U2X0`) and multiprocessor (`MPCM0`) flags while clearing the Transmit Complete flag (`TXC0`) by writing a 1 to its location. If it was not set, this change will have no effect; if it was previously set, then ANDing it with a 1 will cause a new 1 to be written back, thus clearing the flag. This is required because with an empty transmit buffer, USART interrupts are disabled, so this bit will not be cleared automatically.
(6) The function exits, returning the number of bytes written to the buffer, always one, which is not quite true as the code has completely bypassed the sketch's transmit buffer.

The `write()` function continues in Listing 4-39 if the sketch's transmit buffer and the USART's `UDR0` register were both not found to be empty on entry to the `write()` function.

Listing 4-39 The HardwareSerial::write() function—continued

```
tx_buffer_index_t i =
  (_tx_buffer_head + 1) % SERIAL_TX_BUFFER_SIZE;        (1)

// If the output buffer is full, there's nothing
// for it other than to wait for the interrupt
// handler to empty it a bit

while (i == _tx_buffer_tail) {                          (2)
  if (bit_is_clear(SREG, SREG_I)) {

    // Interrupts are disabled, so we'll have to poll
    // the data register empty flag ourselves. If it is
    // set, pretend an interrupt has happened and call
    // the handler to free up space for us.

    if(bit_is_set(*_ucsra, UDRE0))
      _tx_udr_empty_irq();                              (3)

  } else {                                              (4)
    // nop, the interrupt handler will free up
    // space for us
  }
  }
  _tx_buffer[_tx_buffer_head] = c;                          (5)

  // make atomic to prevent execution of ISR between
  // setting the head pointer and setting the interrupt
  // flag resulting in buffer retransmission.

  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {                        (6)
    _tx_buffer_head = i;
    sbi(*_ucsrb, UDRIE0);                                   (7)
  }
  return 1;
}
```

(1) The first free location in the sketch's transmit buffer is found. This will be used later to update the head pointer and also to determine if the buffer is currently full up. The head pointer is the first free byte in the transmit buffer.
(2) The `while` loop will execute for as long as the buffer remains full. Obviously, the variable `i` will never be updated, so the code depends on the sketch's transmit buffer tail pointer—the location in the buffer where bytes are removed and copied to the USART—to change, as it will be when the buffer is being emptied by the interrupt handler, if the interrupt and global interrupts are enabled.
(3) The helper function for the USART's transmission code is called manually here as interrupts are not enabled, so the transmit buffer will not empty under interrupt control.
(4) If global interrupts are enabled, the buffer will eventually empty by itself,[1] so there is nothing that needs to be done here.
(5) The new data byte is stored in the buffer at the current head location as there is finally some space in the buffer to do so.
(6) This block of code is wrapped in an atomic block to ensure that interrupts do not adjust the head pointer while this block is doing so. This could cause the same byte to be transmitted twice. Whenever there are data in the buffer, then the USART's transmit interrupt is enabled to ensure that they are written out to the `Serial` interface. The interrupt is disabled when `end()` is called or when the interrupt handler's helper, `_tx_udr_empty_irq`, has emptied the buffer.
(7) This sets the USART Data Register Empty interrupt enable bit to configure the USART to send bytes over the serial link while there are some left to transmit. It will be disabled when no more bytes are left in the sketch's transmit buffer.

The following functions facilitate the transmission of different numeric data types:

- `write(unsigned long n)`
- `write(long n)`
- `write(unsigned int n)`
- `write(int n)`

[1] Actually, by the interrupt handler.

The following `using` call allows the `Serial` interface to transmit `String` class variables:

```
using Print::write;
```

The various `write()` functions will all eventually call down to the `write()` function in Listings 4-38 and 4-39 to do the actual transmission via the USART.

4.4.2.17 Function _rx_complete_irq(void)

This is the receive data interrupt helper function. It doesn't handle the receipt of data itself, it simply hands off control to the appropriate handler. It must not to be called directly in your sketch code.

4.4.2.18 Function _tx_udr_empty_irq(void)

This is the transmit data interrupt helper. Like `_rx_complete_irq`, it doesn't handle the transmission of data itself, it simply hands off control to the appropriate handler. It must not to be called directly in your sketch code.

Warning
The two interrupt handler helper functions are visible as they are declared `public`, but are most definitely *not* intended to be called by sketch code, only by the interrupt ISRs themselves or other internal code in the `HardwareSerial class`.

4.5 The String Class

The `String` class is defined in `\$ARDINC/WString.h` and implemented in `\$ARDINC/WString.cpp` and provides a simple C++ method of creating C++ `strings`, as opposed to C's *old-fashioned* `char` arrays.

`Strings` can be added together, converted to and from numbers, and so on. They are quite useful in this respect. However, they do use a lot of dynamic memory allocation and reallocation, and this has certain drawbacks, mainly the ability to use up your scarce Static RAM, causing all sorts of possible corruptions and hard-to-find crashes.

Warning
The `String` class uses a lot of dynamic memory allocation. This means that there could be runtime errors when used on AVR microcontrollers with minimal available RAM. The author of the Arduino JSON library, Benoit Blanchon, has a few warnings and pointers as to why you should *never* use the `String` class. You can read the article or, at least, the "Disclaimer" at https://blog.benoitblanchon.fr/arduino-json-v5-0/#disclaimer. Then, if you so wish, avoid the use of the `String` class in your Arduino code.

As a simple example, when making a string longer, there must be enough free Static RAM to allocate the existing allocated space plus the new amount that is required—this can, briefly, double the amount of Static RAM required and, under certain circumstances, exceed the amount available, leading to corruption.

Just because it compiles OK doesn't mean that it will (always) run OK. Beware.

Interestingly enough, I don't know of anyone who uses this class in their sketches. I've never seen one used either—but that doesn't mean there are no sketches out there in the wild which use `String` variables, of course.

In Section 4.2, when describing the `Printable` class, I showed how a class, named `Person`, could be streamed by inheriting from the `Printable` class. That particular example did use the `String` class (because it was easier to type in!); however, it did take up a lot more Flash RAM than had I written the class to use char arrays rather than `Strings`. However, I did a quick experiment and created another class using plain old-fashioned `char` arrays instead of `Strings`.

With `Strings`, the example consumed 3,222 bytes of Flash and 234 bytes of Static RAM. When converted to use plain `char` buffers, the Flash usage dropped to 1,808 bytes, and Static RAM dropped to 232 bytes. This was for a pair of `char[10]` buffers instead of the `String` variables, and Flash usage was only 56% of that of the `String` version for the same features. Obviously, bigger buffers will result in more Static RAM usage.

Given the simplicity of the `Person` class, `String` variables were a valid option. Apart from streaming them, nothing else was done with them at all. With only that sort of usage, `String` variables are perfectly acceptable. Had I perhaps written the class in such a way that it was required to manipulate those `String` variables, for example, changing data within them, adding extra characters, or other changes, then those actions would each be a potential source of random crashes if and when the various dynamic allocations and copying of `String` data around in Static RAM exceeded the amount of RAM available.

The ATmega328P has 32 Kb of Flash RAM, but only a paltry 2 Kb (2,048 bytes) of Static RAM. Static RAM is where your sketch's variables get kept while the sketch code goes into the Flash RAM. You see the memory used in each area at the end of a compilation and upload.

Tip

Because the use of `String` variables is fraught with potential danger, I strongly advise against their usage. However, should you wish to use them, so be it.

The Arduino Reference website has all the details that you will need to create `Strings` from various other data types, and there is a full explanation of the various functions and methods that are available to operate on `Strings` there too.

You can find all the documentation at www.arduino.cc/reference/en/language/variables/data-types/stringobject/.

Converting to the AVR Language 5

This chapter briefly explains how you can begin to wean yourself off of the helpful features of the Arduino Language and write code that is in the AVR's own variant of C/C++. A small change to convert an Arduino function can often greatly reduce the size of your compiled code and could make the difference between fitting your project into an ATtiny85, for example, rather than needing a full-blown ATmega328P.

Writing AVR C/C++ also turns off all the hand-holding that the Arduino gives you. You are talking directly to the device, rather than having your needs and wishes interpreted by an intermediary before, eventually, being passed along to the device.

You should be warned, writing in AVR C/C++ will entail frequent reading the data sheet for the AVR microcontroller in your Arduino board, and writing code that is—initially at least—a lot harder to understand than the Arduino Language that you are used to. However, you will find yourself writing small code libraries that get used frequently in your own code, which make life easier again. In addition, these libraries will be written in AVR C/C++ and will be far more efficient than the Arduino equivalent.

This chapter covers the *simple* things that you can do right now, even within your existing sketches to save a bit of Flash RAM, time, and battery power for those projects you want to power from batteries. Chapters 7, 8, and 9 of this book delve a lot deeper into the hardware of the ATmega328P, and in those chapters, there's a *lot more* low-level information.

5.1 Introduction

In this chapter, I will be looking at how you can convert, fairly easily, some of the Arduino Language features to AVR-specific C++. AVR-specific C++ is actually what the Arduino Language maps down onto anyway, as you will see later in Chapters 7, 8, and 9. Using AVR C++, the "middleman" gets cut out, and things become smaller and faster, with less power required too. The information in this chapter should give you ideas on how to *start* migrating from the Arduino Language to talking directly to the microcontroller.

Having said that it is "fairly easy," you should also be reminded that the Arduino Language is very *readable*, and this is extremely helpful for beginners and experienced makers alike. Plain AVR C/C++ is, how shall I put it, not *quite* so user-friendly. It's not impossible, but you should be aware that *properly* commenting your code is most likely a must from now on.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_5

Later on, in Chapter 6, I shall introduce you to an application named PlatformIO which allows you to code in pure AVR C/C++ without needing the Arduino Language or IDE. This really allows you to get down and dirty in the code. It also allows you to continue to create Arduino Language sketches, if you so desire, and can be used as an alternative to the Arduino IDE. PlatformIO allows you to use your preferred editor and/or development IDE[1] to write Arduino code.

If you like the Arduino IDE, then fear not, as that IDE can also be used to write plain AVR C/C++ code, provided your main source file retains the `.ino` extension.

PlatformIO is a cross-platform and runs on Windows, Linux, and Mac, just like the Arduino IDE. It is also able to write and compile code for numerous different development boards, not just Arduino.

However, I shall start gently and continue using the Arduino IDE.

The rest of this chapter assumes that you

- Have a reasonable level of understanding of C/C++ code. I'll try to keep things as simple as I can though.
- Know about binary and hexadecimal number systems.
- Understand logical operations on binary values, `AND`, `OR`, `NOT`, and `XOR` specifically.

Just in case, here's a recap.

5.2 Numbering Systems

Writing code involves knowing a little about various numbering systems. It's not all decimal—although that's the one we are most used to, having ten digits on our hands and feet. Computers and microcontrollers work in binary, base two, where something is on or off, high or low, or a 1 or a 0; there is nothing else. I'll start this chapter with a recap of the various numbering systems.

5.2.1 Decimal Numbering

It makes sense to start with something familiar!

In decimal, a digit's position in the number represents the count of the power of 10 at that point, counting from zero upward and from right to left. The rightmost digit is the number of 10^0 or units, the next digit from the right is the number of 10^1s, and the next digit from the right is the number of 10^2s. Thinking back to my primary school days, we have

100s	10s	1s
1	2	3

This number counts the number of 100s that we have, plus the number of 10s, plus the number of 1s. That's $(1 * 100) + (2 * 10) + (3 * 1)$ giving 123. Simple? Too easy? Let's move on.

[1] PlatformIO can generate project files for a good many different IDEs.

5.2.2 Binary Numbering

Binary numbers use only the digits 0 and 1. This is another positional numbering system, similar to decimal, in that the rightmost digit, or bit, represents the units or 2^0, the next represents the 2^1, and so on, up to 2^7 in an eight-bit number. We see increasing powers of two, or doublings, as we move right to left.

In binary, the value 0b0111 1011—we use a prefix of "0b" for binary here—is

128s	64s	32s	16s	8s	4s	2s	1s
0	1	1	1	1	0	1	1

This is $(1 * 64) + (1 * 32) + (1 * 16) + (1 * 8) + (1 * 2) + (1 * 1)$, which is 123.

The problem with binary is that it easily gets unwieldy. Up to about 255 is fine; there are eight bits (binary digits) to cope with; after that, it tends to get a bit hard to follow. Hexadecimal is a good way to keep things easily understandable and reduces the number of digits required by a factor of four compared with binary.

5.2.3 Hexadecimal Numbering

Hexadecimal is based on powers of 16. And straightaway, we can see a problem as we only have ten digits, zero to nine.

To get around this minor setback, hexadecimal uses the digits 0–9 as normal, plus the letters A–F—where "A" is 10, "B" is 11, and so on, up to "F" for 15—so that there are 16 hexadecimal "digits" in use to represent hexadecimal numbers. Each hexadecimal digit replaces four binary bits, and this makes a good reason to use hexadecimal in situations where binary would be too cumbersome.

For the value 0x7B—we indicate hexadecimal by a prefix of "0x"—we have

16s	1s
7	B

This value represents $(7 * 16) + (11 * 1)$ and is, once again, exactly 123.

As with decimal, this is a positional numbering system, where the columns show powers of 16, starting on the far right with 1s, then 16s, then 256s, then 4,096s, and so on.

In order to represent the first 16 hexadecimal values, 0 through 15, it requires four binary bit values—8, 4, 2, and 1. That's 16 different values that can be created using just those bits. If then the binary number is split into groups of four bits, starting at the right, each of those groups can be converted to a hexadecimal digit using Table 5-1.

Using the previous example, we can split the eight bits of 0b0111 1011 into two groups of four bits, that is, 0b0111 and 0b1011. Converting to hexadecimal using Table 5-1 results in 0x7 and 0xB or 0x7B. Instead of requiring a table to convert, it's easy to convert the binary bits in each four-bit group into a decimal number and then make that into hexadecimal by adjusting the digits if the result is ten or more.

0b0111 is $(0*8)+(1*4)+(1*2)+1$ and gives the answer 7. 0b1011 is $(1*8)+(0*4)+(1*2)+1$ and gives the answer 11. 11 is not a valid hexadecimal digit, but its equivalent is 0xB. Once more, the result is 0x7B.

Table 5-1 Binary, hexadecimal, and decimal conversion

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

That's about all there is to it. Hexadecimal is a much better way to represent values in computer registers, memory, and so on, rather than binary. Decimal could be used but, for some reason, isn't, at least not very often.

5.3 Binary Logical Operations

Computers use binary. The various digital logic gates that computers and microcontrollers are built from rely on logic to work. There are only a few basic gates known or used, and many of them are actually made up from something called a NAND gate or sometimes a NOR gate. The following sections deal with the most used gates in microcontroller construction.

Now you might be wondering what this has to do with the Arduino. Fear not, the truth tables for hardware gates are exactly the same as the truth tables for those bitwise operations done in a sketch. There will be more on this later on in this chapter, so for now, consider the following a short reminder.

5.3.1 NOT

The `NOT` operation takes a single binary bit as input and results in its opposite value as the output. It inverts the bit, in other words.

The truth table for the `NOT` operation is thus

A	Not A
0	1
1	0

5.3.2 AND

Given any two, or more, binary digits, or bits, they can be `AND`ed together to give a result which depends on the two (or more) inputs. `AND` works as follows:

- If *all* inputs are 1, then the result will also be 1.
- Otherwise, the result will be 0.

The truth table for the AND operation, with two inputs, is thus

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

5.3.3 OR

Given any two, or more, binary digits, or bits, they can be ORd together to give a result which operates as follows:

- If *any* of the inputs are 1, then the result will also be 1.
- Otherwise, the result will be 0.

The truth table for the OR operation, with two inputs, is thus

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

5.3.4 XOR

Given any two, or more, binary digits, or bits, they can be XORd together to give a result which works as follows:

- If *all* the inputs are the same, then the result will be 0.
- Otherwise, the result will be 1.

The truth table for the XOR operation, with two inputs, is thus

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

The remainder of this chapter will require you to be at least slightly familiar with the truth tables listed earlier. Panic not, it will be explained as required. Now we are ready to start looking at losing the Arduino Language hand-holding, but in a gentle manner.

5.4 Replacing the Arduino Language

As I introduced, in Chapter 2, the Arduino IDE does a fair amount of hand-holding to make life easy for the beginner. It sets up various *stuff* in the background, it provides easy-to-understand function names, and so on. In the Arduino Language, you never *need* to see what's happening under the covers with the actual hardware of the ATmega328P. If you wish to carry on in this manner, then I'm afraid that this chapter of the book is not for you!

Still here? Good! Read on, and prepare to cast off the shackles of an easy life!

5.4.1 The ATmega328P Pins and Ports

On the ATmega328P, there are numerous pins, as you are aware, and these pins live in three different "banks," each bank consisting of up to a maximum of eight pins. The pins in each bank were named, by Atmel, "Pxn" where "x" is the bank, and "n" is the pin number on that bank—`PB5`,[2] for example, which corresponds to the Arduino's `D13` pin. On the ATmega328P, the banks are not all the same—not all have the full complement of eight pins.

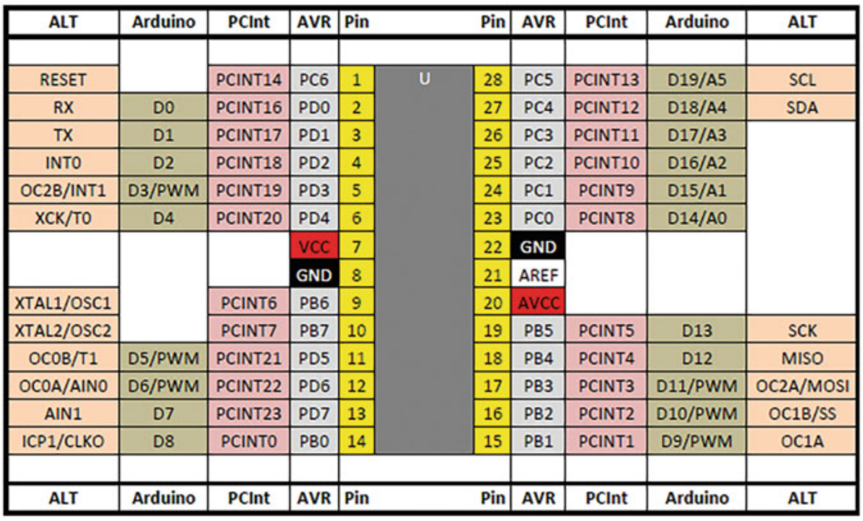
- Bank B has eight pins, `PB0` through `PB7`. Most of the pins in this bank are usable on an Arduino board apart from pins `PB6` and `PB7` as these are used for the 16 MHz crystal oscillator, so there are only six available pins on bank B.
- Bank C only has seven pins, `PC0` through `PC6`. Pin `PC6` is special in that normally it is used as the RESET pin. It can also be used as an additional I/O pin if the appropriate fuse bit (see Section 7.1 for details), `RSTDISBL`, is programmed. Doing this, however, prevents the device from being programmed (or reset), and if further programming is required, a special high voltage or a parallel programmer must be used instead. In normal use, bank C therefore has six pins available.
- Bank D has all eight pins, `PD0` through `PD7`, available for use on the Arduino.

See Figure 5-1 for a pinout diagram of the ATmega328P. You will find the Arduino and AVR pin labeling names on the diagram.

To summarize, the following pins are the only ones available to us on an ATmega328P-based Arduino board:

- `PB0` through `PB5`: Pins `PB6` and `PB7` are used for the 16 MHz crystal oscillator and so are unavailable on Arduino boards.
- `PC0` through `PC5`: Pin `PC6` is the RESET pin and should *really* be left well alone!
- `PD0` through `PD7`.

[2] You may also see this written as `PORTB5` in some documentation, and, indeed, this is a macro as defined in the AVRlib to allow manipulating the individual bits of the PORT registers by name.



ALT	Arduino	PCInt	AVR	Pin		Pin	AVR	PCInt	Arduino	ALT
RESET		PCINT14	PC6	1	U	28	PC5	PCINT13	D19/A5	SCL
RX	D0	PCINT16	PD0	2		27	PC4	PCINT12	D18/A4	SDA
TX	D1	PCINT17	PD1	3		26	PC3	PCINT11	D17/A3	
INT0	D2	PCINT18	PD2	4		25	PC2	PCINT10	D16/A2	
OC2B/INT1	D3/PWM	PCINT19	PD3	5		24	PC1	PCINT9	D15/A1	
XCK/T0	D4	PCINT20	PD4	6		23	PC0	PCINT8	D14/A0	
			VCC	7		22	GND			
			GND	8		21	AREF			
XTAL1/OSC1		PCINT6	PB6	9		20	AVCC			
XTAL2/OSC2		PCINT7	PB7	10		19	PB5	PCINT5	D13	SCK
OC0B/T1	D5/PWM	PCINT21	PD5	11		18	PB4	PCINT4	D12	MISO
OC0A/AIN0	D6/PWM	PCINT22	PD6	12		17	PB3	PCINT3	D11/PWM	OC2A/MOSI
AIN1	D7	PCINT23	PD7	13		16	PB2	PCINT2	D10/PWM	OC1B/SS
ICP1/CLKO	D8	PCINT0	PB0	14		15	PB1	PCINT1	D9/PWM	OC1A
ALT	Arduino	PCInt	AVR	Pin		Pin	AVR	PCInt	Arduino	ALT

Figure 5-1 ATmega328P pinout

Note

Regarding pins `PB6` and `PB7`, you can configure the ATmega328P to use its own internal 8 MHz oscillator rather than the external one and free up these two pins for general I/O use. Sadly, this cannot easily be done when the device is embedded in an Arduino board as there are no headers on an Arduino board that connects those pins to the outside world.

This works best if the microcontroller is used in a breadboard or in a circuit board of your own design. For example, see Appendix H for a breadboarded Arduino doing exactly this.

Figure 5-1 shows the location of the various banks of pins. The numbers in the columns labeled "Pin" are the physical pin numbers. The columns labeled "AVR" columns next to those are the Atmel pin names. The "PCInt" columns list the names used when processing pin change interrupts, and next to those are the "Arduino" pin names like `D3` and so on. Finally, on the outermost columns, we have the "ALT" or alternative functions for the pins as some pins can be configured for multiple, but separate, tasks.

You can see from the Figure 5-1 that all the pins in a bank are not necessarily located adjacent to each other. Look at where `PB6` and `PB7` (physical pins 9 and 10) are to be found, within the pins of bank D—between pins `PD4` and `PD5`, if we ignore the power and ground pins, of course.

There are three ATmega328P registers which control the numerous pins in the three banks. These are as follows:

- The Data Direction Register, DDRx, which is used to configure a pin as either an `INPUT`, the default, or an `OUTPUT`. Each bank of pins has its own DDRx, and "x" is the bank name—`DDRB` for bank B, for example.

- The bank's Data Register, PORTx, which is used to set the associated pins `HIGH` or `LOW` when configured as an `OUTPUT`. Each bank of pins has its own PORTx, and "x" is again the bank name—`PORTC` for bank C, for example. The PORTx register can also be used with `INPUT` pins but only for one specific reason which is covered in the next section on replacing the `pinMode()` function.
- The bank's Input Pin Register, PINx, which is used to read the state of a pin that has been configured for `INPUT`. Each bank of pins has its own PINx where "x" is, once more, the bank name—`PIND` for bank D, for example. As with the PORTx registers, the PINx registers can be used with pins configured as `OUTPUT`, and, again, this is for one specific reason only. This will be explained later when discussing the replacement of `digitalWrite()`.

Note
Each pin in a bank corresponds to a single bit in the preceding three registers. Pin `PD0`, for example, is bit zero in the `DDRD`, `PORTD`, and `PIND` registers. It might have been nice if Atmel had allowed code to use the same name as the pins, but no, they didn't. Instead of setting bit `PD5` in the `DDRD` register, for example, you have to set bit `DDD5`. The other two registers have their own names too; the same bit in `PORTD` is named `PORTD5`, and in the `PIND` register, it is `PIND5`.

5.4.2 Replacing pinMode()

One of the first things any sketch does, usually, is to set up pins for input or output as required. In the Arduino Language, this is accomplished using the `pinMode()` function, similar to the example in Listing 5-1.

Listing 5-1 Using pinMode() in a sketch

```
#define LED LED_BUILTIN
#define BUTTON 2
#define SENSOR 3
#define RELAY 7

void setup() {
  pinMode(LED, OUTPUT);
  pinMode(RELAY, OUTPUT);
  pinMode(BUTTON, INPUT_PULLUP);
  pinMode(SENSOR, INPUT);
}
```

Here, we can see the three different modes that an Arduino pin can be configured: `INPUT`, `OUTPUT`, and `INPUT_PULLUP`.

In the end, after much processing and checks, the prime purpose of `pinMode()` is simply to set or clear one single bit in the DDRx register for the bank that the pin is located on.

If the bit in the DDRx register is a zero, the corresponding pin is an `INPUT` pin. A one bit in the DDRx register configures the pin as an `OUTPUT`.

When an AVR microcontroller is reset, or powered on, all pins are configured internally as input pins. Alternatively, you can be 100% certain that a pin is correctly configured as input if you explicitly do it yourself—just write a zero to the appropriate bit in the DDRx register to make it an input pin.

Because input is the default state for all the pins, you are not required to *explicitly* configure any pins as input. It is helpful, from a code readability point of view, to do so. Bear in mind, however, that doing so will use up some additional space in the flash area on the microcontroller. If program space is *really* at a premium, you could omit configuring the input pins and save a little space. The ATmega328P has 32 Kb of Flash, so there should be ample. The Arduino bootloader takes around 2 Kb of that on my Duemilanove, but only 512 bytes on my Uno.

Tip
When you are *really* stuck for space in a sketch, you might be surprised at how many bytes of Flash RAM you can save by omitting the `pinMode()` calls and writing directly to the DDRx registers instead.

Tip
When an Arduino resets or powers up, all the pins are configured as `INPUT`. The `init()` function described way back in Section 2.10 does not change this configuration.

Setting pins as `OUTPUT`, on the other hand, must be explicitly specified. This requires setting the appropriate bit in the DDRx register.

The example in Listing 5-2 shows how all of bank D, `PD0–PD7`, could be set as input pins, while four pins on bank C, `PC0–PC3`, could be set as output pins.

Listing 5-2 Replacing pinMode()

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

#define ALL_INPUT 0b00000000
#define FOUR_OUTPUT 0b00001111

void setup() {
  DDRD = ALL_INPUT;        // D0-D7 = INPUT.
  DDRC = FOUR_OUTPUT;      // D14-D17 = OUTPUT, plus
                           // D18-D19 = INPUT.
}
```

It should be obvious, from Listing 5-2, that setting multiple bits in each DDRx register can easily be done in a single instruction rather than having to set each pin individually using `pinMode()`. To perform the equivalent of this in Arduino code would require a minimum of four `pinMode()` calls to configure `A0–A3` as `OUTPUT`; optionally, further eight calls to configure `D0–D7` as `INPUT`; and two more optional `pinMode()` calls to configure `D18` and `D19` as `INPUT` pins. In total, that's 14 `pinMode()` calls, which is a *lot* of overhead even if loops are used. Listing 5-3 shows the equivalent Arduino code to the AVR code in Listing 5-2.

Listing 5-3 Numerous pinMode() calls

```
void setup() {
  // Pins D0--D7 are INPUT.
  for (int pin = 0; pin < 8; pin++) {
    pinMode(pin, INPUT);
  }

  // Pins A4/D18 and A5/D19 are also INPUT.
  pinMode(18, INPUT);
  pinMode(19, INPUT);

  // Pins A0 through A3 are OUTPUT.
  // Note: A0 = D14, A1 = D15, ...
  for (int pin = 14; pin < 18; pin++) {
    pinMode(pin, OUTPUT);
  }
}
```

Tip
From Figure 5-1, you will realize, I hope, that pins `PC0` through `PC5` are Arduino pins `A0` through `A5` which can be used as digital pins if necessary. These are pins `D14` through `D19`. Likewise, pins `PD0` through `PD7` are Arduino pins `D0` through `D7`.

There are three different states that a pin can be configured as, `INPUT` and `OUTPUT` we now know about, but what about `INPUT_PULLUP`? The Arduino allows this mode, so what does the DDRx register do as it only has two states for a bit?

Given that a single bit in the DDRx registers can only be a one or a zero, does this mean that we have a problem with `INPUT_PULLUP`? There's no other value that can be written to the DDRx register to make the pin take on the desired mode. How then is it possible to set up a pin as `INPUT_PULLUP`? In the previous section, I mentioned that when a pin is an `INPUT` pin, we can still use the corresponding PORTx register for a special purpose; this is that purpose.

It appears that the designers of the ATmega328P decided that as the PORTx register has no use normally with input pins, it could be used to enable the internal pull-up resistors for the pin. If you write a 1 to any of the PORTx register bits, then input pins will be configured with their internal pull-up resistors enabled.

Tip
I have seen it advised, in examples in books, on the Web, and in the data sheets, to set the PORTx bits to pull-up *before* configuring the DDRx register to set the pins as input. I'm not 100% convinced to be honest.

- If the microcontroller has been reset, the pins are already `INPUT` pins by default, so enabling the pull-up is all that is required.
- If the pins are already configured as `OUTPUT`s, writing a 1 to the PORTx register will set the pins `HIGH` and *might*, briefly, enable some feature of the project into a dangerous state. If the Arduino is running a high power laser cutter, setting a pin `HIGH`, however briefly, might not always be a safe option.

Because of this, I consider it best to configure the pin as input first, then activate the pull-up resistors by writing to the PORTx register. That's just my opinion.

In my AVR code, I configure the pins as `INPUT` and then write to the PORTx register to enable the pull-ups, which is also what the Arduino Language does deep in the code for `pinMode()`.

5.4.3 Replacing digitalWrite()

The `digitalWrite()` function, as described in Section 3.2 and much loved by users of the Blink sketch, is used to set a physical pin on the Arduino board to either supply voltage, when set `HIGH`, or to ground, when set `LOW`. The function does a lot of checking and so on before getting down to the real purpose of its existence. This is, quite simply, setting or clearing a single bit in one of the PORTx registers.

One of the failings of `digitalWrite()` is that it can only be applied to a single pin at a time, so if you wanted to *simultaneously* set a number of pins—perhaps connected to LEDs—to `HIGH`, then you cannot do that with `digitalWrite()` as that function only affects a single pin.

When using AVR C/C++ and avoiding the helpfulness of the Arduino Language, you can set a number of pins to a given state at the same time. This relies on the pins all being on the same *bank*. Bank D, for example, allows up to eight separate pins to be set high together, using the AVR C/C++ language, whereas this would require eight separate calls to the `digitalWrite()` function and, obviously, would not set all eight pins at *exactly* the same instant.

To configure an output pin `HIGH`, simply set the appropriate bit in the PORTx register corresponding to the bank that the pin is in. To configure the pin `LOW` instead, the bit should be cleared. If you need to configure multiple pins high or low, set or clear the appropriate bits—perhaps in an eight-bit variable, such as a `uint8_t`—and write the variable's resulting value to the PORTx register.

Note
Although I say that "you *can* set a number of pins to a high state at the same time," that obviously applies to all the pins on the same bank. It would not be possible to set all pins on bank C *and* also those on bank D to `HIGH` together; it would have to be one bank's pins first, then the other bank.

Some examples follow in Listing 5-4.

Listing 5-4 Replacing digitalWrite() examples

```
#define ALL_OUTPUT 0xFF // All pins are output pins
#define ALL_ON 0xFF // 11111111 Binary
#define ALL_OFF 0 // 00000000 Binary

void setup() {
  // Set banks B, C and D to all outputs.
  DDRB = ALL_OUTPUT;
  DDRC = ALL_OUTPUT;
  DDRD = ALL_OUTPUT;
  ...
}

void loop() {
  // Set all pins on Bank B to low.
  PORTB = ALL_OFF;                                              (1)

  // Set all pins on Bank D to high.
  PORTD = ALL_ON;                                               (2)

  // Set pin PC5 to high.
  // Leave all other Bank C pins unchanged.
  PORTC |= (1 << PORTC5);                                       (3)

  // Now, turn pin PD0 low.
  // Leave all other Bank D pins unchanged.
  PORTD &= ~(1 << PORTD0);                                      (4)
  ...
}
```

(1) This turns off all pins in bank B. Pins in the bank which are nonfunctional, or not present, are simply ignored. `PB6` and `PB7`, if used for the external crystal oscillator, would not be affected.
(2) This turns on all pins in bank D.
(3) This turns on pin `PC5`, Arduino `D19/A5`, in bank C, but without affecting any other pin in that bank.
(4) This turns off pin `PD0`, Arduino `D0`, in bank D without affecting any other pin in that bank.

In normal circumstances, you may wish to set a pin high or low without affecting any other pins in the same bank, as per the latter two examples. This is where binary bit twiddling comes to the fore.

5.4.4 Enabling Internal Pull-Up Resistors

I previously mentioned the use of the PORTx register with pins configured as `INPUT`, which was described in Section 5.4.2. I'm including it here as it is relevant to the PORTx registers. If you set a bit in the PORTx register for any pin configured as input, then you will enable that pin's pull-up resistor.

5.4.4.1 Bit Twiddling

The introductory section in this chapter discussed various binary logic operations. When using PORTs, PINs, and DDRs, a certain amount of binary logic is required. It is considered impolite, and sometimes dangerous, to simply set all the bits in a register when you only need to change one or two. To that end, the following will explain how individual bits can be turned on or off at will.

To turn on a single bit in a value or register, use the bitwise `OR` operator, "|", with a mask with the appropriate bit set, and only that single bit will be affected. You can see this in the third example from Listing 5-4:

```
PORTC |= (1 << PORTC5);
```

If you need more than one bit, then just keep adding bits in. For example, to turn on bits `PC1–PC3`, simply use

```
    PORTC |= ((1 << PORTC3)|(1 << PORTC2)|(1 << PORTC1));
```

To turn off a single bit in a value, use the bitwise `AND` operator, "&", with a mask with the appropriate bit cleared, and only that single bit will be affected. All other pins that you do not want to turn off should have their bits set.

To set a single bit to zero easily, initially create a mask with only the required bit set, then invert it with the "~" operator. This creates the desired mask. This is what is done in the final example in Listing 5-4:

```
    PORTD &= ~(1 << PORTD0);
```

Once again, if you need to turn off more bits, just keep adding bits in to the mask. For example, to turn off bits `PC1–PC3`, simply use

```
    PORTC &= ~((1 << PORTC3)|(1 << PORTC2)!(1 << PORTC1);
```

So much typing, so much to get wrong! If you are setting up a bitmask with multiple bits configured, you may wish to predefine them as `const uint8_t`, similar to Listing 5-5.

Listing 5-5 Defining and using multi-bit bitmasks

```
const uint8_t PORTC_1_2_3_ON = ((1 << PORTC3) | \
                                (1 << PORTC2) | \
                                (1 << PORTC1));


const uint8_t PORTC_1_2_3_OFF = ~(PORTC1TO3_ON);

```

```
...

  // Pins PC1--PC3 go high, other pins unchanged.
  PORTC |= PORTC_1_2_3_ON;

  // Pins PC1--PC3 go low, other pins unchanged.
  PORTC &= PORTC_1_2_3_OFF;

  ...
```

When using these types of constants, the *compiler* does all the work, so any bit shifting and/or inverting is done once at compile time and not at all at runtime.

When you are getting down and dirty in some of the internals of the ATmega328P, or looking at code which does so, you may see code that shifts a zero bit such as

```
PORTC |= ((1 << PORTC0) | (0 << PORTC2) ... );
```

This is quite a good idea as it explicitly shows the state of *all* the bits. It's perhaps not so useful in the PORTx register as all the bits have pretty much the same meaning, but in timer control registers, for example, where each bit has vastly different functions, and there are numerous configuration options, it can be a lot more useful to see the settings for all the bits in the register.

It's also a lot easier to change later when you realize that you got it wrong first time and have to add in a bit or two extra!

5.4.5 Replacing digitalRead()

The `digitalRead()` function is used to read the voltage present on a physical pin on the Arduino board. It will return either a `HIGH` or a `LOW` depending on the voltage on the pin. When a pin is left floating, then the value returned will be untrustworthy to say the least.

Tip
Never, ever let `INPUT` pins float. You have been warned!

When you wish to read the state of a pin, or pins, you read the PINx register for the bank appropriate to those pins. You can only read the state of multiple pins when they are on the same bank. If a specific bit in the PINx register is set, then the pin is `HIGH`; otherwise, it is `LOW`. There is a PINx register for each bank of pins on the ATmega328P.

As with `digitalWrite()`, you can only `digitalRead()` a single pin at a time in the Arduino Language. When you are talking directly with the ATmega328P, bypassing the Arduino's "hand-holding," you can read up to eight pins simultaneously. They do have to be on the same bank, of course.

Listing 5-6 shows how to read eight pin states in a single instruction.

Listing 5-6 Reading multiple pin states simultaneously

```
void setup() {
  DDRD = 0; // All of bank D are inputs.
  ...
}

void loop() {
  ...
  uint8_t bankD = PIND;
  ...
}
```

That's it! You have just read the state of all eight pins in bank D in one operation, by reading the PINx register for the bank in question.

Reading a PINx register returns an unsigned eight-bit value (`uint8_t`), which has a clear bit for every `LOW` pin and a set bit for every `HIGH` pin. If you only need to read a single pin, or perhaps a few, you can set up a bitmask with a set bit in the positions representing the pins you wish to interrogate. Then read the PINx register and use the bitwise `AND` operator ("&") to mask out the unwanted bits, as shown in Listing 5-7.

Listing 5-7 Reading a single pin's state

```
// I'm interested in Pin PD1 only (Arduino D1).
const uint8_t PIN_D1 = (1 << PIND1);

void setup() {
  DDRD = 0; // All of bank D are inputs.
  ...
}

void loop() {
  ...
  // Read PIND and extract PD1 only.
  uint8_t pin_d1 = (PIND & PIN_D1);

  if (pin_d1) {
    // PD1 was HIGH.
    ...
  } else {
    // PD1 was LOW.
    ...
  }

  ...
}
```

You can also extract the values for more than one pin by ANDing the PINx value with a suitable bitmask.

5.4.6 Toggling Output Pins

Back in the introduction to this chapter, I promised to tell you about the special use of the PINx register with pins configured as OUTPUT. If you write code to set a bit in the PINx register for an output pin, then whatever state the pin is currently in will toggle. Given how simple this is, I'm mildly surprised that the Arduino Language doesn't include a `digitalToggle()` function, especially as the Blink sketch appears to be extremely popular!

Listing 5-8 would work for `digitalToggle()` and uses the features of writing to the PINx register to toggle an output pin.

Listing 5-8 The digitalToggle() function

```
void digitalToggle(uint8_t pin)
{
  uint8_t timer = digitalPinToTimer(pin);                  (1)
  uint8_t bit = digitalPinToBitMask(pin);
  uint8_t port = digitalPinToPort(pin);
  volatile uint8_t *in;

  if (port == NOT_A_PIN) return;

  // If the pin supports PWM output, we need to turn it off
  // before doing a digital write.
  if (timer != NOT_ON_TIMER) turnOffPWM(timer);

  in = portInputRegister(port);                            (2)

  uint8_t oldSREG = SREG;                                  (3)
  cli();
  *in |= bit;                                              (4)
  SREG = oldSREG;                                          (5)
}
```

(1) The code begins by converting the pin number, in the usual Arduino manner, to a timer number, a bitmask with a single bit set representing the pin, and a port name. The function quietly exits if the pin turns out to be invalid. This is in keeping with the actions of `digitalRead()` and `digitalWrite()`.

(2) This is where we get the PINx register for the pin in question.

(3) From here, the code saves the status register and disables interrupts. Is this actually necessary? In `digitalWrite()`, it is required because the operation to change the bit is not atomic and could be interrupted, so I am playing safe here too. See issue 146 at https://github.com/arduino/Arduino/issues/146 which appears to be why it was added to the `digitalWrite()` function.

(4) Setting the bit in the PINx register will toggle the pin, regardless of the current state.

(5) Restore the status register and, by implication, turn interrupts back on—if they were previously on.

5.4.7 Installing digitalToggle()

You unfortunately cannot simply add the code to any sketch you write that needs to toggle a pin, either in the sketch itself or in a separate tab in the IDE. This is down to the fact that the function `turnOffPWM()` is declared `static` in the file `$ARDINC/wiring_digital.c` and, as such, cannot be called from anywhere outside that file. If you wish to use the `digitalToggle()` function I've described in Listing 5-8, you will have to add it to that very same file.

To install the function, proceed as follows with the Arduino IDE closed:

- Add the code in Listing 5-9 to `Arduino.h`.

Listing 5-9 Adding digitalToggle() to Arduino.h

```
void digitalToggle(uint8_t);
```

- Save the file.
- Add the code for `digitalToggle()` in Listing 5-8 to the file `wiring_digital.c`.
- Save the file.

Tip
If you do decide to go ahead and install this function, be aware that you will have to reinstall it every time you upgrade the Arduino IDE and software. Maybe, just maybe, I'll submit this as a patch and see if it can be included in the official releases.

Once installed, you can test it with a sketch containing the code shown in Listing 5-10.

Listing 5-10 Example digitalToggle() sketch

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

void loop() {
  digitalToggle(LED_BUILTIN);
  delay(1000);
}
```

After compiling and uploading the `digitalToggle` sketch, the built-in LED is yet again flashing every second, and as a bonus, it was only 890 bytes compiled rather than the standard blink sketch's 928 bytes. That's 96% of the standard blink, which doesn't sound much, but it could be the difference between getting a sketch to upload and not.

As we are considering casting off some parts of the Arduino Language to save on resources and potentially increase performance, we can do even better than Listing 5-10. As we all now know all about the DDRx, PORTx, and PINx registers, we can rewrite the Blink sketch in a much more precise way, as shown in Listing 5-11.

Listing 5-11 Hard core Blink sketch

```
void setup() {
  DDRB = (1 << DDB5);
}

void loop() {
  PINB |= (1 << PINB5);
  delay(1000);
}
```

This version compiled down to a size of only 598 bytes, or 64% of the original. Yes, I know, blink isn't representative, but you get the idea. The hand-holding and helpfulness of the Arduino Language can lead to slightly bloated programs.

That's a very brief introduction to replacing some of the Arduino Language with less resource-intensive, but less readable, code. You can carry on using the Arduino IDE and simply replace your `pinMode()`, `digitalRead()`, and `digitalWrite()` calls with code similar to that shown and reap the benefits of losing a bit of weight from your sketches and gaining a few milliseconds of better performance.

In order to start replacing functions like `analogRead()` or `analogWrite()`, for example, you need a better understanding of the ATmega328P's internals. That comes in Chapters 7, 8, and 9 of this book. Before that, however, I thought I would introduce you to a couple of alternatives to the Arduino IDE. Chapter 6 is next and covers a couple of replacements, should you be interested.

Alternatives to the Arduino IDE 6

In this chapter, I will be taking a look at alternatives to the Arduino IDE. There are a number of such IDEs, but the two I shall be concentrating on are *PlatformIO* and, although it's no longer an actual alternative to the Arduino IDE, the *Arduino CLI*.

6.1 What Are the Alternatives?

A number of alternatives to the Arduino IDE exist. Some are massive overkill such as the *Microchip Studio* (www.microchip.com/en-us/tools-resources/develop/microchip-studio), based on *Microsoft Visual Studio* and which only runs on Windows. *MPLAB-X* (www.microchip.com/mplab/mplab-x-ide) is another large application, but at least is cross-platform. Others such as the *AVR Eclipse Plugin* (https://marketplace.eclipse.org/content/avr-eclipse-plugin) are plug-ins for the *Eclipse IDE*, a Java-based IDE on steroids and, again, quite large.

This chapter looks at two smaller alternatives, and the two which I will be investigating here are

- *PlatformIO* (https://platformio.org/) which is both a command-line version and also can be used to convert your favorite text editor into an IDE to develop Arduino software—if your favorite editor is *Atom* or *VSCode* that is—but fear not, it can also be used to create project files for a number of popular IDEs.

Note
Support for the *Atom* editor is being deprecated. All advice on the support forum at https://community.platformio.org now advises to use *VSCode* instead.

- The all new *Arduino CLI* (https://github.com/arduino/arduino-cli) which is now out of its alpha release status and is being used under the covers in the new version 2 Arduino IDE. It can, as its name implies, be used stand-alone—with *make*, for example.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_6

6.2 PlatformIO

PlatformIO is a system which allows you to write, compile, and upload programs to your Arduino board, either as Arduino sketches—as you are used to in the Arduino IDE—or as AVR C/C++ format, which removes the hand-holding that you get from the Arduino IDE. Assembly language is also available as a development language, as indeed it is in the Arduino IDE.

The PlatformIO package comes in two flavors:

PlatformIO Core	which installs only the command-line utilities.
PlatformIO IDE	which installs a plug-in for the *Visual Studio Code* editor—henceforth known as *VSCode*. Other IDE systems—*Eclipse*, *Qt Creator*, and *Code::Blocks*, among others—don't have a PlatformIO plug-in as such, but PlatformIO can generate project files for those IDEs to allow you to develop AVR code in your favorite development environment.

There are two ways to install PlatformIO. The first method is to install PlatformIO Core so that it is usable from the command line, then install PlatformIO IDE into *VSCode*.

The second method is to install PlatformIO into *VSCode* and then add the installation folder to your `PATH` so that the command-line tools are available via the code installed for *VSCode* and do not require to be installed separately. This is my preferred method and will be discussed in this section.

6.2.1 System Requirements

Before installing PlatformIO IDE, you need to ensure that you have the requirements documented at https://docs.platformio.org/en/latest/core/installation/requirements.html. In summary, these are the following:

- Python 3.6 or a higher release: For Windows users, *do not* use the Python installation available from the Windows Marketplace. It doesn't work. You should download the official Python kit from www.python.org/downloads/ for best results and a lot less frustration!
- https://docs.arduino.cc/tutorials/generic/DriverInstallation has instructions for Windows users who will need to install a suitable USB driver. If you are already using the Arduino IDE, then you probably have the correct driver installed already.
- https://docs.platformio.org/en/latest/core/installation/udev-rules.html#platformio-udev-rules has instructions for Linux users who will need to install a udev rules file to ensure that communication with the USB subsystem is permitted.
- https://hallard.me/enable-serial-port-on-raspberry-pi/ is the location for Raspberry Pi users who need to enable the Serial Port.
- Finally, Linux users will need to be added to the groups `dialout` and/or `plugdev` for Debian, Ubuntu, and their derivatives or `uucp` and/or `lock` for Arch and its own derivatives. After configuring your user, log out and back in again to use the new groups.

```
## Debian/Ubuntu etc:
sudo usermod -a -G dialout YOUR USER NAME
sudo usermod -a -G plugdev YOUR USER NAME
```

```
## Arch:
sudo usermod -a -G uucp YOUR USER NAME
sudo usermod -a -G lock YOUR USER NAME
```

It doesn't matter if you add your user to a group that isn't present on your system; just ignore any errors that are reported.

6.2.2 Installing PlatformIO IDE

This section assumes that you already have *VSCode* installed and are happy using it to install extensions. PlatformIO IDE is easy to install:

- Click View ▷ Extensions.
- In the search box, type "PlatformIO" without the quotes, and press the return/enter key. The top search result will be "PlatformIO IDE."
- Click the "install" button.

The installation can take some time, depending on your download speed, but eventually it will complete and request that you click a button to restart the editor. Do not close the window or move to another editor tab until you are advised that the installation is complete.

If you check your extensions again after the restart, you will find that you have PlatformIO IDE and three Microsoft C++ extensions. You may also notice the following:

- A new editor tab appears named "PIO Home" and an alien's head as an icon.
- The alien's head icon appears on the *VSCode* toolbar at the left of the screen.
- The command palette (View ▷ Command Palette or CTRL+SHIFT+P) has a number of new PlatformIO commands listed.

Warning
If you have any other C++ extensions installed, PlatformIO may not operate correctly as these have been known to interfere with the Microsoft extensions for C++. You might have to uninstall the additional ones.

That's all that needs to be done. If you have a Python interpreter already installed—as per the system requirements—then the PlatformIO extension should use it, although on some systems it may also have downloaded a minimal version of Python to use. In the PlatformIO extension's preferences, you can decide whether to use the system-wide or the downloaded version of Python.

On my Linux Mint system, PlatformIO didn't download a new version of Python, it linked its own "downloaded" version to the system installed one using a symbolic link.

To link the PlatformIO Core commands, so that they can be used in a terminal session, all that needs to be done is to add the `bin` directory that has been created to your `PATH`. On Linux, this is

```
export PATH=$PATH:/home/norman/.platformio/penv/bin
```

This line should be added to your `.bashrc` or `.profile` file, located in your `$HOME` directory, but remember to change my username to yours. When you next start a terminal session, the various pio commands will be available. `pio --help` is a good command to start with.

6.2.2.1 The Toolbar

A new button has been added to the *VSCode* toolbar at the left of the screen. It has an alien head as the icon, and this is the entry into PlatformIO IDE.

If you click the tool button, a list of PlatformIO tasks will appear. Under "Quick Access," you will find "PIO Home"; click the "Open" option, and the PIO Home screen will open in a new tab in the editor.

Other tasks are available on the "Quick Access" tab in the task list, but many of those are also accessible from the Home screen.

6.2.2.2 PIO Home Tab

On the PIO Home tab in the editor, you have another new toolbar and some options in the main display area.

On the toolbar, there are icons for the following:

Home	Takes you back to the PIO Home page, if you happen to have chosen one of the following options.
Projects	Allows you to search for projects in the list that appears. These are your most recently opened projects. If the list is empty, you haven't opened anything recently. From here, you can add existing projects to the list or create new ones.
Inspect	Allows various details of an existing project to be inspected and displayed. The code is analyzed as is memory usage and so on.
Libraries	Allows you to search for and install libraries which may be required by some Arduino and AVR projects.
Boards	Allows you to search for a board. Try entering "Duemilanove" and clicking the search icon. The two variants of the board will be shown alongside some important details of the boards such as the platform and framework required, the memory sizes, and so on.
Platforms	Displays any installed platforms and allows you to uninstall them or to install any new requirements for a new board.
Devices	Assuming you have an Arduino board attached, this option will display details of the board, the type of communication chip, and the port it is attached to. Similarly to the *arduino-cli*, this option will not detect an ICSP device.

While on the main page itself, we can see these options:

New Project	This should be fairly obvious. It allows you to create a new project and pick as many boards for it as you wish.
Import Arduino Project	This option lets you navigate to an existing Arduino project and import it into PlatformIO's favored format.
Open Project	This option opens an existing project within the editor and allows you to continue developing it.
Example Projects	This option displays a few example projects and allows you to import them into PlatformIO for inspection or learning. You can, of course, compile the example projects.

At the bottom of the screen is a couple of items of recent PlatformIO news and a list of your most recent projects. If this is a new install, you probably don't have any listed. If you did, you have options here to hide the projects from this list or to open it. Clicking the "open" link opens the *VSCode* explorer on the left side of the screen to the top-level directory for the project. From there, you can open files for editing in the usual manner.

6.2.2.3 Creating a New Project

A new project is created from the PIO Home screen. If it is not already being displayed, then click the alien's head[1] icon in the left side toolbar to open the PlatformIO menu. Under quick access, open the list of options under "PIO Home" and click "Open." A new tab appears named "PIO Home."

On this tab, click the "New project" option, then fill in (or select) the appropriate options for the project name, the board you want to use, and the location for the project. You can supply a custom location or simply accept the default, which will be displayed if you hover over the "?".

If you wish to use the same source code for numerous different boards, simply keep adding new ones until you have everything set up as you wish.

Click the "Finish" button to create the project. After a short delay, the PIO Home tab will close. On the left, the explorer will open with the various files and directories of the new project on display. The PIO Home tab will then reopen, after another short delay. I usually keep it closed as it is of little use during development—it can be disabled in the settings.

You can now edit your files in the usual manner.

6.2.2.4 Import Arduino Project

On this dialog, you are able to select a board to be used with the project and to select the existing sketch to be imported. Projects imported in this manner are given a default name of `YYMMDD-HHMMSS-BOARD_NAME` which is most unhelpful! However, you can rename the directory after closing the editor.

6.2.2.5 Opening Existing Projects

If your project already exists, then you can use the PIO Home tab to open it. As before, make sure the tab is visible in the editor, then click the "Open Project" option, navigate to the project's location, and click the "Open <project name>" button.

As before, the PIO Home tab will close, and the explorer on the left will open at the project's location. The project can now be edited as required.

6.2.2.6 Project Examples

Clicking this option will create a new project based on an example application or sketch. You select the desired example from a drop-down list, but if you type into the selection area, it will filter the results.

As with importing Arduino projects, the new project is given a default name based on the date and time—`YYMMDD-HHMMSS-EXAMPLE_NAME`.

6.2.2.7 Editing the Project

The PlatformIO system expects to find header files within the `include` directory and source files in the `src` directory under the project's home, although header files located in `src` will also be found.

After creating a new project, create a new file, `src/main.cpp`,[2] and type in the code from Listing 6-1. This is not an Arduino sketch, but a simple AVR C++ project to blink the built-in LED on Arduino pin `D13`.

[1] Is it an alien's head? Or is it an ant's head? I think it's an alien.

[2] If you are on Windows, then this file *may* have been created for you.

Listing 6-1 Yet another blink sketch!

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  // D13 is actually PortB Pin 5. Configure
  // that pin as an output.
  // This equates to the Arduino setup() function.
  DDRB = (1 << DDB5);

  // This equates to the Arduino loop() function.
  while (1)
  {
    _delay_ms(1000);

    // Toggle the LED by writing to PINB.
    PINB |= (1 << PINB5);
  }

  // We never get here, but it keeps the compiler quiet!
  return 0;
}
```

6.2.2.8 Compiling a Project

Once editing is done, you can compile the project in a number of different ways:

- From the task list on the left, select Project Tasks ▹ YOUR BOARD ▹ General ▹ Build.
- Type CTRL+ALT+B.
- Click the tool button with a tick (or check mark) icon on the toolbar at the bottom of the display.
- Click Quick Access ▹ Miscellaneous ▹ PlatformIO Core CLI which will open a new terminal with the CLI commands enabled, then `pio run` will build the project. This can be executed even if the PlatformIO Core commands have not been added to the `PATH`.
- If you already have the PlatformIO Core commands on the `PATH`, run a PIO terminal, Quick Access ▹ Miscellaneous ▹ New Terminal, then execute the `pio run` command.

  Additionally

- If you have the PlatformIO Core commands on the `PATH`, run a *VSCode* terminal session. On the main menu bar, click Terminal ▹ New Terminal, then execute the `pio run` command.
- Open a command-line session and execute the `pio run` command.

There's far too many options here, so I advise using one or more of the first three options. I use the tick/check mark tool button myself.

If all looks OK with the compilation, we are ready to upload the sketch to the board.

6.2.2.9 Upload a Project

- From the task list on the left, select Project Tasks ▷ YOUR BOARD ▷ General ▷ Upload.
- Type CTRL+ALT+U.
- Click the tool button with a "right pointing arrow" icon on the toolbar at the bottom of the display.
- Click Quick Access ▷ Miscellaneous ▷ PlatformIO Core CLI which will open a new terminal with the CLI commands enabled, then `pio run -t upload` will build the project.
- If you already have the PlatformIO Core commands on the `PATH`, run a PIO terminal, Quick Access ▷ Miscellaneous ▷ New Terminal, then execute the `pio run -t upload` command.

Additionally

- If you have the PlatformIO Core commands on the `PATH`, run a `VSCode` terminal session. On the main menu bar, click Terminal ▷ New Terminal, then execute the `pio run -t upload` command.
- Open a command-line session and execute the `pio run -t upload` command.

Once again, this is far too much choice. I therefore advise using one or more of the first three options again.

By default, uploads use the bootloader.

To use an ICSP, you need to slightly modify the `platformio.ini` file in the project's directory. Add the following lines which obviously only apply to the USBTiny ICSP device:

```
; Required for a usbTinyISP device.
upload_protocol = usbtiny
```

That's a very quick and high-level overview of something that's becoming very popular in the Arduino world, especially with users of 3D printers running the Marlin firmware as Marlin 2.0 onward uses PlatformIO to build and upload the firmware.

I personally have swapped over from the Arduino IDE to PlatformIO for most of my projects and experimented with AVRs and Arduinos. I find it a lot faster to develop software using PlatformIO, and I'm more comfortable in the *VSCode* editor than in the Arduino IDE.

I also use the PlatformIO Core commands on the command line quite often. Sometimes, it's just quicker than starting up *VSCode* and clicking around. The next section gives an overview of the Core CLI.

6.2.3 Testing PlatformIO Core

Once you have added the PlatformIO IDE location to your `PATH`, then any new terminal sessions have access to the many commands available in the Core CLI. The command `pio` or its alias `platformio` is now available. Because I'm lazy, I use the shorter version, `pio`. Feel free to use the longer version if this appeals to you.

6.2.3.1 Set Up a New Project

Find out where PlatformIO expects to find your projects. The default will be displayed using the following command which displays all the current settings:

```
pio settings get
```

A more specific command to display only the `projects_dir` setting is the following:

```
pio settings get projects_dir
```

On my Linux system, it returns the following:

```
Name Value [Default]
-------------------------------------------------------------
projects_dir /home/norman/Documents/PlatformIO/Projects
```

Note
In order to save wrapping text around, I've trimmed some excess text from the preceding output. The output also lists a description of the setting name, which is sometimes useful.

If you don't like the default location, you can easily change it:

```
mkdir -p ~/SourceCode/PlatformIO/Projects
pio settings set projects_dir ~/SourceCode/PlatformIO/Projects
```

The result of executing this command is

```
The new value for the setting has been set!
Name Value [Default]
--------------------------------------------------------------
projects_dir /home/norman/SourceCode/PlatformIO/Projects [...]
```

The previous setting is also listed, at the end of the line, in square brackets, but is not shown here.

Note
PlatformIO has the idea of a default location for projects, but strangely, it is not used except in PlatformIO Home which is a browser-based pseudo-IDE, which is discussed later on in Section 6.2.9.

When creating new projects, you must either be located in the directory where you wish to create the new project, or you should use the `-d` (or `--project-dir`) option to specify the *full path* to the project directory, which must already exist.

Change to the projects directory, as listed by `pio settings get projects_dir`, and create a new folder to house the project:

```
cd ~/SourceCode/PlatformIO/Projects
mkdir TestProject
cd TestProject
```

Determine the name to be used for your specific board. In this example, it's a Duemilanove:

```
pio boards duemil
```

This gives me two options:

```
Platform: atmelavr
------------------------------------------------------------
ID                   MCU        ... Name
------------------------------------------------------------
diecimilaatmega168 ATMEGA168  ... Arduino Duemilanove ...
diecimilaatmega328 ATMEGA328P ... Arduino Duemilanove ...
```

The output text has a lot more information which is far too wide for the page, so I've trimmed it of irrelevant detail.

As I'm using the latter version with the ATmega328P, I need to initialize the new project with the `diecimilaatmega328` board ID. If you have an Arduino Uno, the process is similar, but has many more results. You should be looking under the `Platform: atmelavr` for your Uno, which will, surprisingly, be called `uno`.

Note
There is no need to install any tools, compilers, etc., for the various boards as PlatformIO will do this automatically for you if it detects that you are using a board for which no tools yet exist.

PlatformIO doesn't use any existing tools that were installed by the Arduino IDE, so you may end up with two separate versions of the AVR compiler, etc. This is not a major problem and does mean that when you decide to continue writing Arduino code using PlatformIO, instead of the Arduino IDE, you can simply uninstall the Arduino IDE and still be able to compile with PlatformIO.

6.2.3.2 Initialize the Project

This example sets up a new project for the Duemilanove board which I'm using. You can also create the same project for numerous boards. To show how this can be done, I'll create the project for an Arduino Uno as well as my Duemilanove.

Tip
If you accidentally forget to initialize a second or subsequent board, you can easily do it by making another call to `pio init -board` or `pio init -b` with the additional board(s). Those new boards will be added to the current project.

```
pio init --board diecimilaatmega328 --board uno
```

After a very short delay, the screen will be filled with useful information about the new project and a list of commands to compile, upload, and clean the project files.

As the messages indicate, some files and directories have been created. These are the following:

`platformio.ini`	Is a file that holds all the configuration for the project. Any changes you make to this file will only affect the project in the current directory; however, some or all environments in the project may be affected. The file itself contains settings for each of the boards with which the project was initialized.

`include`	Is a directory expected to be used for any header files for your project. These are the `*.h` files.
`src`	Is a directory where you are expected to save all the C/C++/ino files that make up the project.
`lib`	Is a directory where any libraries, private to this project, should be saved or copied. A `readme.txt` file is also created within this directory explaining how it should be used.
`test`	Is a directory for test-driven development (TDD) test files.

In addition to the information regarding created files and directories, some `pio` commands are displayed as examples of how to build and upload sketches to your board.

You can now edit `platformio.ini` and change the long-winded name, if necessary, to something more memorable. I changed mine to the following as I couldn't be bothered typing "diecimilaatmega328" all the time. ("2009" is what "duemilanove" means in Italian, while "diecimila" is "2010".)

```
[env:2009]
platform = atmelavr
board = diecimilaatmega328
framework = arduino

[env:uno]
...
```

Now I can compile for the Duemilanove with the name "2009" instead. It's easier to remember and less typing too.

If, like me, you have an ICSP device, then you need to edit the `platformio.ini` file, or programming the board will not work. In my case, I added a new *environment* so that I could use either the bootloader or the ICSP device. The new environment is simply a copy of the existing one, with an extra line added to identify the upload_protocol, as shown in the `[env:2009_icsp]` and `[env:uno_icsp]` sections in Listing 6-2.

Listing 6-2 Example platformio.ini file

```
; Settings common to *all* environments.
[env]
platform = atmelavr
framework = arduino

; Duemilanove settings using the Arduino bootloader.
[env:2009]
board = diecimilaatmega328

; Uno settings using the Arduino bootloader.
[env:uno]
board = uno

; Duemilanove settings using the USBTiny.
[env:2009_icsp]
board = diecimilaatmega328
```

```
upload_protocol = usbtiny

; Uno settings using the USBTiny.
[env:uno_icsp]
board = uno
upload_protocol = usbtiny
```

Tip
Lazy typists, like me, who occasionally create the `platformio.ini` file by hand, will love the ability to extract all the common settings to the `[env]` environment as shown in Listing 6-2. This will allow only those settings specific to each board, to be specified in the boards' own environments.

The PlatformIO documentation at https://docs.platformio.org/en/latest/platforms/atmelavr.html is the documentation for the Atmel AVR development platforms and has details of what is required by each known ICSP device. In my case, as I have a USBtiny clone, I used this line from Listing 6-2:

```
upload_protocol = usbtiny
```

The command to use to upload a sketch to the Duemilanove using the programmer is

```
pio run -e 2009_icsp -t upload
```

And for the Uno, using the bootloader, it is

```
pio run -e uno -t upload
```

You can now upload using the bootloader or the ICSP device. Be aware that once you have used the programmer, *you no longer have a bootloader* and cannot then use `pio run -t -e 2009 upload` to upload using the bootloader; you must now use `pio run -t -e 2009_icsp upload`—until you recreate the bootloader of course.

I suppose we need to create the ubiquitous Blink sketch now?

6.2.4 PlatformIO for Arduino-Style Projects

As already mentioned, PlatformIO allows you to continue using the Arduino Language in your projects. This section will explain how the Blink sketch can be converted and compiled in the new environment.

Create a new project, then a new file, `src/main.cpp` (or whatever name you wish—mine is `Blink.cpp`), and add the code in Listing 6-3 to it.

Listing 6-3 Arduino blink sketch

```
#include "Arduino.h"                                   (1)

// Is the built in LED already named?
#ifndef LED_BUILTIN                                    (2)
  #define LED_BUILTIN 13
#endif

// This runs once, in the usual Arduino manner.
void setup()
{
  // Make sure the LED is an output pin.
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  // LED on, then wait 1,000 milliseconds.
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);

  // LED off, then wait another 1,000 milliseconds.
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

(1) You normally do not need to do this in the Arduino IDE, but in PlatformIO, you must include it.
(2) This is just a safety check to ensure that the built-in LED has been given a name in the `Arduino.h` file. In most cases, it has been done, but it's always best to check and avoid any compilation errors that may arise.

As you can see, not much has changed. You can now use your favorite text editor to write code for your Arduino.

6.2.4.1 Compiling Arduino Projects

Compile the code by first making sure that you are located in the directory where the file `platformio.ini` exists. If you have changed into the `src` directory to edit the file as per Listing 6-3, then please change back up one level.

Now run this command:

```
pio run -e 2009
```

The `-e` option relates to the environment (or board) that you wish to compile the code for. As I created two, I need to inform PlatformIO which board I wish to compile for. In this case, it's the Duemilanove (which I renamed to "2009") and not the Uno. If you omit this option, *all* environments in the `platformio.ini` file will be compiled.

This command will compile, but will not upload the code, in case there are errors. The compilation produced the following (slightly abridged) output:

```
...
PLATFORM: Atmel AVR > Arduino Duemilanove ... ATmega328  (1)
SYSTEM: ATMEGA328P 16MHz 2KB RAM (30KB Flash)            (2)

Library Dependency Finder ... [URL removed for brevity]  (3)
LDF MODES: FINDER(chain) COMPATIBILITY(soft)
Collected 24 compatible libraries                        (4)

Scanning dependencies...                                 (5)
No dependencies

Compiling .pioenvs/2009/src/blink.cpp.o                  (6)

Archiving .pioenvs/2009/libFrameworkArduinoVariant.a     (7)
Indexing .pioenvs/2009/libFrameworkArduinoVariant.a
Compiling .pioenvs/2009/FrameworkArduino/CDC.cpp.o
...

Archiving .pioenvs/2009/libFrameworkArduino.a            (8)
Indexing .pioenvs/2009/libFrameworkArduino.a
Linking .pioenvs/2009/firmware.elf

Checking size .pioenvs/2009/firmware.elf
Building .pioenvs/2009/firmware.hex                      (9)

Memory Usage -> http://bit.ly/pio-memory-usage           (10)
DATA: [ ] 0.4% (used 9 bytes from 2048 bytes)
PROGRAM: [ ] 3.0% (used 928 bytes from 30720 bytes)
============ [SUCCESS] Took 1.06 seconds ============

============ [SUMMARY] ============                       (11)
Environment 2009 [SUCCESS]
Environment uno [SKIP]
============ [SUCCESS] Took 1.06 seconds ============
```

(1) This line summarizes the platform and board in use. In this case, the platform is Atmel AVR and the board is a Duemilanove (even though I renamed it to "2009" in the `platformio.ini` file).
(2) This line summarizes the capacity of the chosen device.
(3) The PlatformIO dependency finder goes hunting for anything that it thinks needs to be included in this compilation. I removed the URL that's normally listed here to get the line on the page.
(4) This is the number of files and libraries that the finder thinks are required for this compilation.
(5) The system is looking for any dependencies here. It decided that there were none.
(6) This is where my source file got compiled. Your file name should appear here too.
(7) Do you recognize these file names? They are all the files that are included by default when you compile an Arduino sketch in the Arduino IDE.

(8) All the Arduino code is statically linked to the elf file. This will be converted to a hex file for uploading.
(9) The hex file is the one that will be uploaded to the Arduino board. It has only been compiled at present, not yet uploaded.
(10) The memory usage section shows that this Arduino sketch used 9 bytes of RAM and 928 bytes of flash program memory. This is standard for the default Arduino Blink sketch. The blank spaces between the square brackets here look pointless; however, when you compile large sketches, this shows a histogram of the amounts of Static and Flash RAM used.
(11) The "SUMMARY" shows that this compilation only affected the "2009" environment/board and that the "Uno" environment was not touched.

Note
On the first compilation of any target device, PlatformIO will download the required toolchain. In my example, it downloaded the *gcc-avr* compiler toolset. This already existed under my Arduino IDE installation, but is sadly not looked for, found, or used by PlatformIO.

6.2.4.2 Uploading Arduino Projects

Uploading compiled sketches to an Arduino board is carried out with a bootloader or with an ICSP device. In the former case:

```
pio run -e 2009 -t upload
```

Or with an ICSP device:

```
pio run -e 2009_icsp -t upload
```

Don't forget that you need to edit the `platformio.ini` file if you want to use a programmer instead of the bootloader—see Listing 6-2.

You should note that in the absence of an `-e` option, all environments/boards in the current project will be compiled and uploaded; this is best avoided as you could end up with Uno code uploaded to your Duemilanove, which may not work correctly—it depends on which board is attached to the USB at the time and which environment is the last one listed in `platformio.ini`.

You could see something similar to the following when uploading:

```
...
Configuring upload protocol...
AVAILABLE: arduino
CURRENT: upload_protocol = arduino
Looking for upload port...
...
Auto-detected: /dev/ttyUSB0
Uploading .pioenvs/2009/firmware.hex

avrdude: AVR device initialized, ready to accept instructions

Reading | ################################################## | 100% 0.01s
```

```
avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: reading input file ".pioenvs/2009/firmware.hex"
avrdude: writing flash (928 bytes):

Writing | ######################################## | 100% 0.51s

avrdude: 928 bytes of flash written
avrdude: verifying flash memory ...
avrdude: load data flash data from input file ...
avrdude: input file ... contains 928 bytes
avrdude: reading on-chip flash data:

Reading | ######################################## | 100% 0.44s

avrdude: verifying ...
avrdude: 928 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done. Thank you.
============ [SUCCESS] Took 3.33 seconds ============
```

You should now be able to see the built-in LED flashing away merrily in the usual manner.

So that's how easy it is to create Arduino-style projects using the command-line versions of the PlatformIO Core code.

The next section shows how to create the Blink sketch as a plain AVR C/C++ program.

Note
You can use PlatformIO to import existing Arduino sketches. This is not really possible from the command line, yet you have to use the `pio home` command, which is discussed later in Section 6.2.9.

6.2.5 PlatformIO for AVR-Style Projects

The preceding project used the standard Arduino Language and compiled down to a hex file the same size as you would have seen if the Arduino IDE had been used instead. You can, however, *go commando* and bypass the entire Arduino system, as shown in the following. Remember, however, that it is your responsibility to make all the decisions about ports and pins and so on.

If you are still in the `TestProject` directory, change back up one level to the standard location for PlatformIO projects. Remember, this is no longer an Arduino, it's a plain vanilla Atmel AVR development board—it just happens to look like an Arduino!

```
mkdir testAVR
cd testAVR
pio init -b uno
```

Once again, if you wish to upload using a programmer, then edit the `platformio.ini` file and add a new environment which is a copy of the one just created, with the additional lines for your particular programmer. Alternatively, mine will be as per Listing 6-4, where I have added a common `env` section—which has settings common to all other environments—and removed the comments for brevity.

Listing 6-4 The new platform.ini file

```
 [env]
platform = atmelavr
board = uno

 [env:uno]
 ; Nothing required here!

 [env:uno_icsp]
upload_protocol = usbtiny
```

You can hopefully see that I've also deleted the line `framework = arduino` as this is no longer required for plain AVR programming. Leaving it in will cause all the Arduino files to be compiled regardless of the fact that they are not used.

Create `src/Blink.cpp` containing the code in Listing 6-5.

Listing 6-5 Another blink sketch

```
#include <avr/io.h>                                   (1)
#include <util/delay.h>                               (2)

int main(void)
{
  // D13 is actually PortB Pin 5. Configure
  // that pin as an output.
  // This equates to the Arduino setup() function.
  DDRB = (1 << DDB5);                                 (3)

  // This equates to the Arduino loop() function.
  while (1)
  {
    _delay_ms(1000);
                                                      (4)
    // Toggle the LED by writing to PINB.
    PINB = (1 << PINB5);                              (5)
  }

  return 0;                                           (6)
}
```

(1) This brings in the correct settings, register names, pin numbers, and other definitions for the particular AVR microcontroller in use.
(2) We need this to enable us to call the `_delay_ms()` function (delay milliseconds).
(3) This is effectively `pinMode(13, OUTPUT)`. Digital pin 13 is on `PORTB` and is bit 5 on that port, so we have to set bit 5 of the `DDRB` register to configure `D13` as an output pin.
(4) Delays for 1,000 milliseconds or 1 second.
(5) Toggling pin `D13/PB5` is carried out by setting to the pin's bit in the `PINB` register.
(6) We never get to this line. Because `main()` is declared as returning an `int` the compiler will complain if we omit this statement. There are other ways to silence the compiler, but this is the easiest, in my opinion.

6.2.5.1 Compiling AVR Projects

Compiling an AVR-style program is exactly the same as before:

```
pio run -e uno
```

The `-e` option can be omitted if there is only a single environment in the project, excluding the common `env` environment. The output from the command will be as follows, and no Arduino files will have been included in the compilation. The following has been slightly abridged to fit on the page.

```
...
PLATFORM: Atmel AVR (4.0.0) > Arduino Uno                (1)
HARDWARE: ATMEGA328P 16MHz 2KB RAM (31.50KB Flash)       (2)
...
Library Dependency Finder ...                            (3)
LDF MODES: Finder ~ chain, Compatability ~ soft
Collected 0 compatible libraries                         (4)

Scanning dependencies...                                 (5)
No dependencies

Building in release mode
Checking size .pio/build/uno/firmware.elf
Building .pioenvs/uno/firmware.hex                       (6)

Memory Usage -> http://bit.ly/pio-memory-usage           (7)
Advanced Memory Usage is available ...
RAM : [ ] 0.0% (used 0 bytes from 2048 bytes)
FLASH: [ ] 0.5% (used 158 bytes from 32256 bytes)
============ [SUCCESS] Took 0.36 seconds ============
```

(1) This line summarizes the platform and board in use.
(2) This line summarizes the memory capacities of the chosen board.
(3) The PlatformIO dependency finder goes hunting for anything that it thinks needs to be included in this compilation. The URL (not shown) that would normally be on this line is that of the documentation for the Library Dependency Finder. I removed it to fit the page width.

(4) This is the number of libraries that the finder thinks are required. It says nothing else will be required.
(5) The system is looking for any dependencies here. It decided that there were none.
(6) This is where my source file got compiled. Your file name should appear here too.
(7) The memory usage section shows that this AVR sketch used no RAM and only 158 bytes of flash program memory.

In bare-bones AVR code, the standard blink sketch takes 158 bytes of flash program memory rather than the Arduino's 928. It also does not create any variables in Static RAM.

6.2.5.2 Uploading AVR Projects

There's no difference in uploading the compiled AVR C/C++ code to the AVR microcontroller than previously when uploading Arduino projects. It is done with a bootloader or with an ICSP device as normal—in the former case:

```
pio run -e uno -t upload
```

Alternatively, with an ICSP device:

```
pio run -e uno_icsp -t upload
```

Remember to edit the `platformio.ini` file if you want to use a programmer instead of, or as well as, the bootloader—see Listing 6-4.

The output from the upload is very similar to that when an Arduino-style project is uploaded, so has not been reproduced here.

The plain AVR style of programming takes less time to compile as it is not required to compile all the Arduino support code—what you see in the program is what you get.

Note
Those readers who are slightly ahead of me here will realize that this means that you don't get things like `millis()` and `micros()`, Timer 0 Overflow Interrupts, or even interrupts enabled when you compile an AVR-style project. Everything you need, you have to enable. The Arduino does a heck of a lot of stuff in the background to make your life easy.

If you wish to create AVR-style projects, but need functions like `millis()` and so on, you might want to take a look at https://github.com/NormanDunbar/AVRmillis.

You should now be able to see the built-in LED flashing away merrily in the usual manner, but this time, using far fewer bytes of flash—158 as opposed to 928—and even better zero bytes of scarce Static RAM for variables, as opposed to nine bytes previously.

6.2.6 Burning Bootloaders

The PlatformIO software, in the *VSCode* IDE or on the command line, has the ability to burn a bootloader for your device. When you have initialized the board in a project, you need simply do the following:

- Edit the `platformio.ini` file and add the `upload_protocol` line required to use an ICSP device, the fuse settings for the board, and the path to the new bootloader:

```
upload_protocol = usbtiny

; Optiboot copied into project directory.
board_bootloader.file = optiboot_atmega328.hex
board_bootloader.lfuse = 0xFF
board_bootloader.hfuse = 0xDE
board_bootloader.efuse = 0xFD
board_bootloader.lock_bits = 0x0F
board_bootloader.unlock_bits = 0x3F
```

- Run the command `pio run --target bootloader` to do the burn.

As I have mentioned, you will have to use an ICSP device or a spare Arduino to use as an ICSP, but it does work and is simple.

6.2.7 Using Your Own IDE

I briefly mentioned that PlatformIO allows you to generate project files for numerous, currently available, development IDEs. I use *Qt Creator* for other development work, and if I decided I wanted to use that IDE to develop my Arduino/AVR C++ projects with PlatformIO, then it is simple. I just have to initialize a new project and specify the IDE I wish to use.

Note
This option to create various IDE project files can only be used from the command line, not from within VSCode.

```
mkdir QtPio
cd QtPio
pio init -b uno --ide qtcreator
```

In addition to the usual files and directories created by PlatformIO, this project has the following files:

- A `Makefile` to build the project, upload, and so on
- A *Qt Creator* project file, `platformio.creator`

Other *Qt Creator*–specific files are also created. You should note that PlatformIO has not created a *Qt Creator* project file named `*.pro`, but this is not a problem.

Once the project files have been generated, I can open *Qt Creator* and

- Select File ▹ Open File or Project, browse to the `QtPio` directory, and select `platformio.creator` to be opened.
- Click File ▹ New File, choose a C/C++ source file as the file type, and click the "Choose" button.
- Name the file `blink.cpp` and ensure that the path is set to the project's `src` directory.
- Click "Next."
- Click "Finish."

This creates a new source file in the project. I am now able to type in the ubiquitous Blink sketch, yet again, and save it. This works for Arduino- or AVR-style projects.

To build the project, I select Build ▹ Build project "platformio" or press CTRL+B. The Compiler Output tab shows that the sketch has been successfully built.

Now what? Clicking Build ▹ Run has no effect! Neither does Build ▹ Deploy? What to do? Well, in order to run the sketch, we first need to upload it.

- Click the "Configure Projects" icon, which looks like a wrench/spanner on the left of *Qt Creator*'s main screen.
- Choose "Run" under "Build & Run" on the left side. We are configuring *Qt Creator* to run a PlatformIO project.
- On the right side, set the executable name to "make" without quotes.
- Set the command-line arguments to "upload," again, without quotes.
- Set the working directory to the *full* path to `QtPio` in this case. The "Browse" button is useful here.
- Click back into the editor, using the "Edit" toolbar button, on the far left.

Now, if you click the run button—the green triangle—or press CTRL+R, you will see the code being uploaded to the board; once successfully uploaded, the sketch will run.

What if I already have a PlatformIO project, but I didn't initialize it with the ability to develop in *Qt Creator*? Simple:

```
cd projectWithoutIDE
pio init --ide qtcreator
```

And the *Qt Creator* files are added, without changing the rest of the project.

6.2.8 Adding Additional Boards

I have written some code for an Arduino Uno using PlatformIO, and I have just realized that the same code can be run on my Nano as well.

```
cd projectWithoutNano
pio init --board nanoatmega328new
```

This will update the `platformio.ini` file and add the default settings for an Arduino Nano with an ATmega328 microcontroller with the new bootloader installed. I will, of course, be changing the environment name from “nanoatmega328new” to “nano” because I’m a lazy typist!

6.2.9 PlatformIO Home

This is an *interesting* development in the PlatformIO system. Running the `pio home` command for the very first time will download some files, if necessary, and then this text will appear on the screen:

```
  ___I_
 /\-_--\   PlatformIO Home
/  \_-__\
|[]| [] |  http://127.0.0.1:8008/
|__|____|______________________

Open PlatformIO Home in your browser by this URL
=> http://127.0.0.1:8008/
PIO Home has been started. Press Ctrl+C to shutdown.
```

After a short delay, the default browser on your system should open at http://localhost:8008 where an IDE-alike screen will be waiting for you.

Note
The terminal session which executed the `pio home` command will hang until CTRL+C is pressed. If you do this while the browser is open, then the browser will stop responding as the HTML server has just been “crashed.”

On this browser screen, you can search for boards, libraries, etc., and install them. You can add new or update existing platforms and so on. However, this is also where you can create new projects without needing to remember all those `pio init` commands and options.

The options available in your browser are identical to those in VSCode on the PIO Home tab there. The options are

- New Project
- Import Arduino Project
- Open Project
- Project Examples

6.2.9.1 Creating Projects

Click the “New Project” button, and on the following dialog, give the project a name, and choose a board and a framework—if there are options available for the chosen board, the framework will usually be automatically selected based on the board you have chosen. When choosing a board, start typing and the list will be filtered according to your typed text.

This is where the `pio settings get projects_dir` is used. As you may have noticed, although you set this way back when installing PlatformIO, it was never apparently used for new projects—you always had to be in the desired location to create a new project. So, leaving the box ticked to "use default location," the project will be created there. Hover over the "?" to see where the default location will be. You can also click the "custom" link to define where you want to create the new project.

Click the "Finish" button, and after a small delay, a message will appear telling you that the project has been created and will be found wherever you chose and that you can process it with the command `platformio run`. Hmmm, and there was me thinking I could use Pio Home to edit projects, no such luck I'm afraid.

6.2.9.2 Importing Arduino Sketches

Another useful feature of Pio Home is that existing Arduino projects can be imported from the Pio Home main screen. To do this, you simply click "Import Arduino Project" and follow the on-screen prompts to

- Choose a board
- Select the existing Arduino Project

Then click the "Import" button.

A new folder will be created in your default PlatformIO project directory. It will not be named in a meaningful manner, so renaming it might be helpful and wise. Within that directory, you will find the usual PlatformIO directories and files, and, finally, the original sketch will be found in the `src` directory, with its original `.ino` name.

To compile the imported project, simply execute the command `pio run` within the project's top-level directory where the file `platformio.ini` is to be found.

6.2.9.3 Opening Projects

It's useful to create projects, but the option to "Open Project" simply tells you to go to the location where the project is and open it in your favorite IDE. However, maybe the future will be different, and we can edit and so on within the browser. That would be fun!

So, after creating the project, you are pretty much back in your favorite IDE or editor, editing and running `pio run` commands to compile the code and `pio run -t upload` to upload with the bootloader.

Tip

To be honest, pio home doesn't appear to be of much use, as described earlier, other than being an easy way to import existing Arduino sketches. However, when you use the PlatformIO IDE variant, the home screen created in *VSCode* is much more useful.

Coming up next, I'll be taking a look at the new Arduino CLI—a command-line utility which is now used by version 2 of the Arduino IDE. This utility, *arduino-cli*, allows you to use your own favorite editor and the *make* utility to create and build your Arduino projects.

6.3 Arduino Command Line

The Arduino IDE has always had a sort of command-line version, at least on Linux; Windows had a separate version due to the way that Windows GUI and command-line applications are so different. However, way back in August 2018, a new Arduino command-line application, *arduino-cli*, came out in alpha test.

In the first edition of *Arduino Software Internals*, the *arduino-cli* was a separate download, was only at version 0.6.0, and had a few missing features such as the ability to burn bootloaders and so on.

With the new version 2 Arduino IDE, *arduino-cli* replaces the whole compilation process for sketches.

The utility can be found in the directory `resources/app/lib/backend/resources/` underneath the location where you extracted the downloaded IDE zip file. At the time of writing, May 2023, the *arduino-cli* utility supplied with the IDE is version 0.32.2 and is dated April 12, 2023.

Warning
It is probable that what I explain here will be likely to some changes as time and new releases go by.

The use of *arduino-cli*, other than building sketches in the version 2 IDE, is to help in getting code built using the *make* utility and for incorporating into other IDEs. It has been noted on the Arduino Developers Google Group[3] that visually impaired users have great difficulty with IDEs and that command-line versions are much better, so this can only be helpful.

6.3.1 Obtaining the Arduino CLI

If you have installed version 2 of the IDE, then you already have the utility installed. It will be found in the directory `resources/app/lib/backend/resources/` as noted previously. You can easily make a copy and save it in a location that is on your path. I have a copy in `/home/norman/bin` on my Linux Mint system.

Tip
When I say *copy*, I mean a symbolically linked copy. This is handy when I update the IDE and get a new version, as I will always be using the latest release.

https://github.com/arduino/arduino-cli/releases/latest is the URL that always takes you to the latest release version. Scroll down to "assets" where you will find 32- and 64-bit versions for Linux, a 32-bit version for Windows, 64-bit versions for macOS and macOS ARM, and many more. Click the link to download the appropriate version for your system. The source code is available here too.

https://github.com/arduino/arduino-cli/releases is the URL for all known releases which are still available for download.

[3] https://groups.google.com/a/arduino.cc/forum/#!forum/developers

6.3.2 Installing the Arduino CLI

As I mentioned, you already have the utility available if you have version 2 of the Arduino IDE installed. It will be found in the `resources/app/lib/backend/resources/` directory beneath the location where you unzipped the IDE download. You can easily make a copy and save it in a location that is on your path. I have a symbolic link to `/home/norman/bin` on my Linux Mint system.

If you have downloaded, or are about to, a separate version to that supplied with the IDE, then you will need to install it.

The documentation at https://arduino.github.io/arduino-cli/0.32/installation/ has full details of how to install the Arduino CLI if you need any further information. If the version you downloaded is different, simply click the version on the green bar at the top, and select the correct release to suit your system.

After downloading, simply unzip on Windows, or use *tar* on Linux and macOS, as shown in the following Linux example:

```
tar -zvjf arduino-cli_0.32.2_Linux_64bit.tar.gz
```

On Windows, right-click the file and "extract," or in the command line, the following will suffice:

```
unzip arduino-cli_0.32.2_Windows_64bit.zip
```

The current version, 0.32.2, expands to two files:

- `arduino-cli` which is the utility itself
- `License.txt` which is the license

If the location you unzipped/untarred the download isn't on your path, you can either add it to the path or copy the `arduino-cli` file to a directory that is on your path:

```
cp arduino-cli ~/bin/
```

Test that the downloaded file works with this command:

```
arduino-cli version
arduino-cli Version: 0.32.2 Commit: 2661f5d9 Date: 2023-04-12
```

You can get a feel for the various commands and options available in the latest version with the following command:

```
arduino-cli help
```

This command produces the following output on Linux. Please note that I have wrapped some lines to ensure that they fit on the page.

```
Arduino Command Line Interface (arduino-cli).

Usage:
  arduino-cli [command]

Examples:
  arduino-cli <command> [flags...]

Available Commands:
  board           Arduino board commands.
  burn-bootloader Upload the bootloader.
  cache           Arduino cache commands.
  compile         Compiles Arduino sketches.
  completion      Generates completion scripts
  config          Arduino configuration commands.
  core            Arduino core operations.
  daemon          Run as a daemon on port: 50051
  debug           Debug Arduino sketches.
  help            Help about any command
  lib             Arduino commands about libraries.
  monitor         Open a communication port with a board.
  outdated        Lists cores & libraries that can be upgraded.
  sketch          Arduino CLI sketch commands.
  update          Updates the index of cores and libraries
  upgrade         Upgrades installed cores and libraries.
  upload          Upload Arduino sketches.
  version         Shows version number of Arduino CLI.

Flags:
  --additional-urls strings   Comma-separated list of
                              additional URLs for the Boards
                              Manager.
  --config-file string        The custom config file (if not
                              specified the default will be
                              used).
  --format string             The output format for the logs,
                              can be: text, json, jsonmini,
                              yaml (default "text")
  -h, --help                  Help for arduino-cli
  --log-file string           Path to the file where logs will
                              be written.
  --log-format string         The output format for the logs,
                              can be: text, json
  --log-level string          Messages with this level and
                              above will be logged. Valid
                              levels are: trace, debug, info,
                              warn, error, fatal, panic
  --no-color                    Disable colored output.
  -v, --verbose                 Print the logs on the
                                standard output.

Use "arduino-cli [command] --help" for more information about
a command.
```

6.3.3 Configuring the CLI

By default, the configuration assumes a number of details about your system. It's best we find those out before we dive in and start creating sketches:

```
arduino-cli config dump
```

This command, on Linux, produces the following output:

```
board_manager:
  additional_urls: []
daemon:
  port: "50051"
directories:
  data: /home/norman/.arduino15
  downloads: /home/norman/.arduino15/staging
  user: /home/norman/Arduino
library:
  enable_unsafe_install: false
logging:
  file: ""
  format: text
  level: info
metrics:
  addr: :9090
  enabled: true
output:
  no_color: false
sketch:
  always_export_binaries: false
updater:
  enable_notification: true
```

The documentation at https://arduino.github.io/arduino-cli/0.32/configuration/[4] is the source of all knowledge regarding what these settings relate to. For the most part, they are perfectly acceptable as defaults. However, if you wish to make any changes, you must first generate a configuration file, as follows:

```
arduino-cli config init
```

When the command completes, you will see the following on the screen:

```
Config file PATH: /home/norman/.arduino15/arduino-cli.yaml
```

You now have all the default configuration stored in the file name listed. In previous releases, it was a case of editing the file with your favorite text editor to change any of the configuration. This is still possible and is probably easier to do than using *arduino-cli*. However, with the latest version of the utility, we also have some configuration commands:

```
arduino-cli config help

Arduino configuration commands.

Usage:
  arduino-cli config [command]

Examples:
  arduino-cli config init

Available Commands:
  add     Adds one or more values to a setting.
  delete  Deletes a settings key and all its sub keys.
  dump    Prints the current configuration
  init    Writes current configuration to a configuration file.
  remove  Removes one or more values from a setting.
  set     Sets a setting value.

Flags:
  -h, --help help for config

...

Use "arduino-cli config [command] --help" for more information
about a command.
```

I won’t examine all of these, but given that I might want to program my ATtiny85 devices with *arduino-cli*, then I should really configure the board settings for those. I need to add an additional URL to allow *arduino-cli* the ability to download and use the required software to program ATtiny devices.

[4] If you have a later version than 0.32, select yours from the drop-down list for up-to-date information.

The add command is the one I have to use:

```
arduino-cli config add board_manager.additional_urls \
https://raw.githubusercontent.com/damellis/attiny/ide-1.6.x-boards-
   manager/package_damellis_attiny_index.json
```

Note
I have used the Linux continuation marker, the backslash immediately followed by a newline, for the command prior to the URL. The URL itself should all be on a single line. Mine has wrapped around to fit on the page. There are no spaces in the URL.

Arduino-cli will find, and use, cores from the Arduino IDE versions prior to 2.0.0. It will not, however, find cores from IDE versions greater than 2.0.0.

You can configure the "data" directory, in the configuration file, to determine where *arduino-cli* will search for cores and so on.

After adding the URL, the configuration file looks like this:

```
board_manager:
  additional_urls:
  - https://raw.githubusercontent.com/damellis/attiny/ide-1.6.x-boards-
    manager/package_damellis_attiny_index.json
daemon:
  port: "50051"
...
```

Once again, the URL has wrapped around to fit on the page.

6.3.4 Creating Sketches

To create new sketches, we use the "sketch new" command, for example:

```
arduino-cli sketch new MyFirstSketch
```

This will create a new directory, `MyFirstSketch`, within the current directory and a file within it, `MyFirstSketch.ino`, similar to how the IDE would do when we save a new sketch.

The "Getting Started" guide at https://arduino.github.io/arduino-cli/0.32/getting-started/ is a very useful web page which helps you get started with the *arduino-cli* utility; however, it doesn't yet note that there are a couple of foibles when creating new sketches.

- The "user" directory, known as the Sketchbook in the Arduino IDE, is listed in the configuration file; however, it is *not used*!
- If only a sketch name is passed to the utility, then it will be created within the current directory.

- If a sketch name, with a relative path, is passed to the utility, then it will be created within the path specified, relative to the current directory.
- If a sketch name, with an absolute path, is passed to the utility, then it will be created within the path passed.

To create a new sketch directory, within the current directory, and an `ino` file within it, simply pass the name of the sketch.

```
cd /home/norman
arduino-cli sketch new MyFirstSketch

Sketch created in: /home/norman/MyFirstSketch
```

To create a new sketch in a directory relative to the current one, pass the desired directory name as a prefix to the sketch name. If the directory name passed doesn't yet exist, it will be created. Because of this, it's best to make sure your spelling is better than mine!

```
cd /home/norman
arduino-cli sketch new Arduino/MyFirstSketch

Sketch created in: /home/norman/Arduino/MyFirstSketch
```

To create a new sketch in an absolute location, simply pass the location and the sketch name. If the full path to the sketch doesn't yet exist, it will be created. When complete, the command will have created a "blank" sketch template in the sketch's directory, ready to be edited.

```
cd /home/norman
arduino-cli sketch new /tmp/testing/MyFirstSketch

Sketch created in: /tmp/testing/MyFirstSketch
```

The sketch created, wherever you specified it, will be a completely "blank" one, as shown in Listing 6-6.

Listing 6-6 A new sketch

```
void setup() {
}

void loop() {
}
```

You can see from Listing 6-6 that there's not much to a new sketch, but it's a start on developing your next great project.

The almost obligatory Blink sketch is obviously required at this point, so edit the generated `MyFirstSketch.ino` file with your favorite text editor to make it resemble the code in Listing 6-7.

Listing 6-7 MyFirstSketch.ino

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

void loop() {
  for (short x = 0; x < 4; x++) {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(150);
    digitalWrite(LED_BUILTIN, LOW);
    delay(150);
  }

  delay(500);
}
```

Well, I can't just keep using the *same* blink sketch all the time, can I? I need a bit of variety! We are now ready to compile the sketch, but as this is a brand-new installation, we have a bit of preliminary work to do. First, we should update the system to make sure we have the latest package index file.

Note
I have configured *arduino-cli* to use different directories to the IDE, so I'll be seeing some messages that you might not.

```
arduino-cli core update-index

Downloading index: library_index.tar.bz2 downloaded
Downloading index: package_index.tar.bz2 downloaded
Downloading index: package_index.tar.bz2 downloaded
```

If you already have the Arduino IDE installed, you may see only the following:

```
arduino-cli core update-index

Updating index: package_index.json downloaded
```

Now, connect your Arduino board to the USB port in the normal manner. Leave a few seconds for it to initialize, and search to see what you have. I have a genuine Duemilanove and a non-genuine Uno attached. Let's see what the system recognizes. First is the Uno:

```
arduino-cli board list

Downloading missing tool builtin:mdns-discovery@1.0.8...
builtin:mdns-discovery@1.0.8 downloaded
....
```

```
Installing builtin:serial-discovery@1.4.0...
builtin:serial-discovery@1.4.0 installed

Port         Protocol Type              Board Name FQBN Core
/dev/ttyUSB0 serial   Serial Port (USB) Unknown
```

In my case, *arduino-cli* has downloaded some tools to assist it in determining which boards, if any, are connected. The final two lines show that my Uno clone is not recognized; however, this is not a problem. The Duemilanove, when plugged in, also results in

```
arduino-cli board list

Port         Protocol Type              Board Name FQBN Core
/dev/ttyUSB0 serial   Serial Port (USB) Unknown
```

Interesting! But nothing to worry about. According to the Arduino developers, when I raised this, the board isn't recognized when there's an FTDI connection. Apparently, my two boards, one genuine and one clone, have FTDI chips on board. This is the output when a recognized Uno board is attached:

```
arduino-cli board list

Port         ... Board Name   FQBN            Core
/dev/ttyACM0 ... Arduino Uno  arduino:avr:uno  arduino:avr
```

I have omitted the protocol and type details as they are unchanged from the unrecognized Uno board. This helps fit the output on the page and is more readable than the wrapped around lines that would result.

The FQBN and Port parameters are useful when it comes to compiling and uploading, but there's no need to panic if the board is not recognized. Boards are detected based on a vendor and product ID, VID/PID. On boards with FTDI chips, these identifiers are generic, so the actual board cannot be determined. This is not a major problem here, however, as both my boards have at least been identified as being present.

Note
Boards will also not be detected if they are connected to your computer with an ICSP device.

6.3.5 Unknown Boards

If your boards, like some of mine, are listed as unknown, we can still find out the required core to install by querying the core and/or board list:

```
arduino-cli core search uno

ID              Version  Name
arduino:avr     1.8.6    Arduino AVR Boards
arduino:megaavr 1.8.8    Arduino megaAVR Boards
```

Alternatively, and more intuitively:

```
arduino-cli board search uno

Board Name             FQBN                  Platform ID
Arduino Uno            arduino:avr:uno       arduino:avr
Arduino Uno Mini       arduino:avr:unomini   arduino:avr
Arduino Uno WiFi       arduino:avr:unowifi   arduino:avr
Arduino Uno WiFi Rev2                        arduino:megaavr
```

My Uno requires the arduino:avr core to be installed from these results.

Note
When searching for my Duemilanove, I don't get any results. However, the Duemilanove is the same as the Diecimila, so search for that instead! Alternatively, leave the search term off (`arduino-cli board search`) and scan or `grep` the results! You still won't find Duemilanove, but that's life I suppose! It might work for other boards though.

Tip
The Arduino CLI "getting started" pages at https://arduino.github.io/arduino-cli/0.32/getting-started/#connect-the-board-to-your-pc note that you can find your unknown board's details using the command `arduino-cli board listall uno`, but that doesn't work until such time as you have installed a core!

As this is a clean install with no IDE installed, we need to use the `arduino-cli board search uno` command instead.

6.3.6 Installing Platforms or Cores

At the moment, you only have the command-line tool itself. In order to compile code for Arduino boards with AVR microcontrollers, you must first install a "core" or "platform." In my own case, I need the arduino:avr core for all my boards. The command I require is

```
arduino-cli core install arduino:avr

Downloading packages...
arduino:avr-g++@7.3.0-atmel3.6.1-arduino7 downloaded
arduino:avrdude@6.3.0-arduino17 downloaded
arduino:arduinoOTA@1.3.0 downloaded
arduino:avr@1.8.6 downloaded
Installing arduino:avr-g++@7.3.0-atmel3.6.1-arduino7...
arduino:avr-g++@7.3.0-atmel3.6.1-arduino7 installed
....
Configuring platform....
Platform arduino:avr@1.8.6 installed
```

After installing, we can check which cores we have installed:

```
arduino-cli core list

ID          Installed Latest Name
arduino:avr 1.8.6     1.8.6  Arduino AVR Boards
attiny:avr  1.0.2     1.0.2  attiny
```

Tip
Don't worry. You don't need to jump through all these hoops every time you want to compile with *arduino-cli*, just the first time, or when you add a new board that has a different core of course.

Finally, we are ready to compile a sketch.

6.3.7 Compiling Sketches

The time has, at last, arrived. We are now ready to compile our first sketch. You *should* still be located within the sketch's directory. That's `/home/norman/Arduino/MyFirstSketch` in my case, so let's compile the sketch—first, for my Uno:

```
cd ~/Arduino/MyFirstSketch
arduino-cli compile --fqbn arduino:avr:uno MyFirstSketch

Error opening sketch: no such file or directory:
/home/norman/Arduino/MyFirstSketch/MyFirstSketch
```

Oh well, that didn't go well. Why not? Remember the various manners of specifying a sketch's location when creating new sketches? Well, similar rules apply to compiling and uploading sketches.

If the current location is the sketch's directory, don't pass a sketch name; only the FQBN is required. The sketch compiled will be the `.ino` file with the same name as the current directory.

```
cd ~/Arduino/MyFirstSketch
arduino-cli compile --fqbn arduino:avr:uno

Sketch uses 954 bytes (2%) of program storage space. Maximum
  is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving
  2039 bytes for local variables. Maximum is 2048 bytes.

Used platform Version Path
arduino:avr 1.8.6 /home/norman/.arduinocli/packages/
arduino/hardware/avr/1.8.6
```

If the current location is the parent of the sketch's directory, then pass the sketch name and it will compile properly.

```
cd ~/Arduino
arduino-cli compile --fqbn arduino:avr:uno MyFirstSketch

Sketch uses 954 bytes (2%) of program storage space. Maximum
  is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving
  2039 bytes for local variables. Maximum is 2048 bytes.

Used platform Version Path
arduino:avr 1.8.6 /home/norman/.arduinocli/packages/
arduino/hardware/avr/1.8.6
```

If the current location is anywhere else, you must pass a relative or the full path to the sketch's directory.

```
cd /home/norman
arduino-cli compile --fqbn arduino:avr:uno Arduino/MyFirstSketch

## Or: arduino-cli compile --fqbn arduino:avr:uno /home/norman/Arduino/
    MyFirstSketch

Sketch uses 954 bytes (2%) of program storage space. Maximum
  is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving
  2039 bytes for local variables. Maximum is 2048 bytes.

Used platform Version Path
arduino:avr 1.8.6 /home/norman/.arduinocli/packages/
arduino/hardware/avr/1.8.6
```

When compiling a sketch, you must supply the board's FQBN. If the compilation succeeds, the usual details about the RAM usage will be displayed. If you need verbose output, simply add the `-v` command-line option to the preceding compile command.

Now, let's compile the same source code for the Duemilanove:

```
cd Arduino/MyFirstSketch
arduino-cli compile \
--fqbn arduino:avr:diecimila
```

With the latest release of *arduino-cli*, this worked perfectly. However, it is possible that for some boards, the Duemilanove and the Nano, where there have been versions with the ATmega168 and the ATmega328 microcontroller, you might need to specify the cpu in use:

```
arduino-cli compile \
--fqbn arduino:avr:diecimila:cpu=atmega328
```

You may wish to use the `-v` option on the compile command to see the full, verbose, compilation text. This is similar to that produced in the Arduino IDE when you have the verbose compilation option enabled and is sometimes useful.

6.3.8 Uploading Sketches

Uploading a sketch uses a similar command to compiling, and you must remember to always use the same FQBN, or it will ask you to compile the sketch first.

The port name is always required when uploading. If you forget the port, then the `arduino-cli board listall uno` command[5] is your friend.

Be aware that the same rules about the current location and the sketch location also apply when uploading.

Uploading to my Uno, from the sketch directory, is carried out as follows:

```
arduino-cli upload \
--fqbn arduino:avr:uno \
--port /dev/ttyUSB0
```

If you had to add the cpu details when compiling, you do not have to do so when uploading. If you do, however, it will be ignored.

Warning
In the Arduino IDE, if you attempted an upload and there had been changes made to the source, then the source code would be automatically recompiled. This does *not* happen with the Arduino CLI. If you make changes, you must recompile before uploading.

There is no indication that all worked well, apart from your Uno now running a different sketch. If you need to, you may add `-v` to the command line and see the same verbose output that the Arduino IDE would give when uploading in verbose mode.

6.3.8.1 Permissions Errors?

It is possible that when uploading, you may see the following error message on Linux:

```
ser_open(): can’t open "/dev/ttyUSB0": Permission denied
Error: exit status 1
Error during upload
```

This is simply because the user that you are logged in as is not a member of the group which owns the port that the upload command was trying to use to communicate with the board. You can check and fix the problem as follows:

```
ls -l /dev/ttyUSB0
crw-rw-rw- 1 root dialout 188, 0 Nov 30 19:09 /dev/ttyUSB0
```

[5] Well, it is if your board isn’t encumbered by an FTDI chip!

We can see that root owns the device, and it is in the `dialout` group. Now, check if my user is in that group:

```
groups
norman: norman adm cdrom sudo dip lpadmin sambashare
```

`Dialout` is not listed, so I need to add it as a group that my user is a member of:

```
usermod -a -G dialout norman
```

Unfortunately, you will need to log out and back in again to pick up the new group. This is mildly irritating, so here's a quick tip on temporarily working around the need to log out. Instead of logging out and back in again, simply run this command in the affected session:

```
su - norman
```

Change "norman" to your own username of course! You will be asked for your password and will start a new shell with the new group assigned. Test it by running the `groups` command again, and note that now you have `dialout` present.

Now you can upload to your Arduino board.

6.3.9 Uploading Sketches with an ICSP

The more recent versions of *arduino-cli* permit the uploading of a sketch using an In-Circuit System Programmer, or ICSP, as opposed to using the Arduino bootloader—as we have been doing up until now. In my case, I have a USBtinyISP device. To allow me to upload a sketch using that tool, I can use this command:

```
arduino-cli upload \
--fqbn arduino:avr:uno \
--port /dev/ttyUSB0 \
--programmer usbtinyisp
```

This will, of course, delete the bootloader.

How do we determine what the correct programmer names are?

```
arduino-cli board details --fqbn arduino:avr:uno \
--list-programmers

Id                      Programmer name
usbtinyisp              USBtinyISP
arduinoisporg           ArduinoISP.org
parallel                Parallel Programmer
...
jtag3isp                Atmel JTAGICE3 (ISP mode)
arduinoasispatmega32u4 Arduino as ISP (ATmega32U4)
arduinoisp              ArduinoISP
```

We need to use the programmer name specified in the *Id* column, not the name in the "Programmer name" column. That's not confusing, is it?

Warning
There is out-of-date documentation online, from some time prior to 2020, which informs us to use `arduino-cli xxxxx -fqbn arduino:avr:uno -programmer list`, where "xxxxx" is either `upload` or `burn-bootloader`, to list the programmers. Unfortunately, those commands no longer apply here.

6.3.10 Burning Bootloaders

Well, after uploading a sketch with the ICSP, we have overwritten our bootloader. How do we get it back or burn one in a brand-new ATmega328, for example? Arduino-cli can once again help us.

```
arduino-cli burn-bootloader \
--fqbn arduino:avr:uno \
--programmer usbtinyisp
```

As with uploading a sketch, there is no indication that it has succeeded. You should make sure to use the correct board identifier as, for example, the Uno uses a different bootloader from that used in the Duemilanove.

To burn the bootloader, you will require an ICSP device or a second Arduino board running the *Arduino as ISP* sketch, in which case the programmer is confusingly named "arduinoisp."

The `$ARDINST/boards.txt` file contains all the fuse and bootloader settings that you will need, unlike the PlatformIO option to burn a bootloader, which requires you to specify the fuse settings and bootloader file that you want.

6.3.11 Serial Usage

If you need to monitor what the Arduino is sending to the Serial Monitor, you are now able to do so with the *arduino-cli* utility, although using the Linux *screen* utility or the *Putty* application for Linux and Windows and so on are still options. If you wish to use *arduino-cli*, then the command is

```
arduino-cli monitor --port /dev/ttyUSB0
```

This assumes that the Serial Monitor has been configured with a baud rate of 9600, the default. If you wish to change the baud rate, as many of us do, then run this command:

```
arduino-cli monitor --port /dev/ttyUSB0 \
    --config baudrate=115200

Monitor port settings:
baudrate=115200
Connected to /dev/ttyUSB0! Press CTRL-C to exit.
```

Other Serial Port settings can also be changed. To determine which ones can be changed, run this command:

```
arduino-cli monitor --port /dev/ttyUSB0 --describe

ID        Setting   Default Values
baudrate  Baudrate  9600    300, 600, 750, 1200, 2400, 4800,
                            9600, 19200, 31250, 38400, 57600,
                            74880, 115200, 230400, 250000,
                            460800, 500000, 921600, 1000000,
                            2000000

dtr       DTR       on      on, off
bits      Data bits 8       5, 6, 7, 8, 9
parity    Parity    none    none, even, odd, mark, space
rts       RTS       on      on, off
stop_bits Stop bits 1       1, 1.5, 2
```

- The “ID” column is the name that you must supply when making changes to the defaults.
- Multiple changes are permitted but must be comma separated.
- Spaces are not permitted anywhere in the configuration parameters.

The following command is a valid example:

```
arduino-cli monitor --port /dev/ttyUSB0 \
    --config baudrate=115200,dtr=off,bits=7
```

However, the next command is not valid as there are spaces after the commas. It will result in errors, unless you wrap the entire configuration string in double quotes.

```
arduino-cli monitor --port /dev/ttyUSB0 \
    --config baudrate=115200, dtr=off, bits=7
```

6.3.12 Profiles

When compiling and uploading, having to always specify the fully qualified board name, or the port for the Arduino board, can get a wee bit tedious. The problem is worse if you are able to use the same source files and compile it for a number of different Arduino boards. To get around this fairly minor problem, we can create a profile.

```
arduino-cli compile --fqbn arduino:avr:uno --dump-profile

Sketch uses 11636 bytes (36%) of program storage space.
Maximum is 32256 bytes.

Global variables use 308 bytes (15%) of dynamic memory,
leaving 1740 bytes for local variables. Maximum is 2048 bytes.

Used library Version Path
LibPrintf 1.2.13 /home/norman/Arduino/libraries/LibPrintf

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6

profiles:
  uno:
    fqbn: arduino:avr:uno
   platforms:
     - platform: arduino:avr (1.8.6)
   libraries:
     - LibPrintf (1.2.13)
```

Now, in theory, and according to documentation on the Arduino blog, from 2021, we should be able to build a profile directly, as follows:

```
arduino-cli compile --fqbn arduino:avr:uno \
--dump-profile > sketch.yaml
```

The `sketch.yaml` file name is mandatory, I'm afraid, and it must be located in the root directory of your sketch, in the same location as the `.ino` file, in other words. Sadly, this doesn't appear to be working as the compiler information about the sketch size and global variables are mixed in with the profile information. However, we can manually edit the `sketch.yaml` file and delete everything up to the word "profiles:" and then save it; we now have a profile for our Uno board.

See https://arduino.github.io/arduino-cli/0.32/sketch-project-file/ for more details on using profiles. One advantage of using profiles is the fact that in five years time, when things have "moved on," recompiling the sketch using a profile will result in all the original versions of the libraries and suchlike being downloaded again, if required,[6] and the new build, five years hence, will be identical to the original.

It is a simple matter to dump out profile information each time we compile the sketch for use with a different board. I have, in my collection, an Uno, a pair of Nanos, and a Duemilanove, among others. If I have a sketch that I can build for those three boards, then I could set up a profile similar to Listing 6-8 and simply choose the profile I want at compile time. The first line specifies the default upload port for every profile in the `sketch.yaml` file. For some reason, there isn't a separate port parameter permitted for each individual profile. Luckily, all my boards—at least, those listed—use the same port.

[6] And, hopefully, if they still exist online!

Listing 6-8 Sketch.yaml

```
default_port: /dev/ttyUSB0

profiles:
  uno:
    fqbn: arduino:avr:uno
    platforms:
      - platform: arduino:avr (1.8.6)
    libraries:
      - LibPrintf (1.2.13)

  nano:
    fqbn: arduino:avr:nano
    platforms:
      - platform: arduino:avr (1.8.6)
    libraries:
      - LibPrintf (1.2.13)

  duemilanove:
    fqbn: arduino:avr:diecimila
    platforms:
      - platform: arduino:avr (1.8.6)
    libraries:
      - LibPrintf (1.2.13)
```

Now a compilation for my Uno is as simple as

```
arduino-cli compile --profile uno
```

Uploading is now equally as simple:

```
arduino-cli upload --profile uno
```

The profile supplies the FQBN for the compile and upload, `default_port` supplies the, ahem, default port for the upload, and *arduino-cli* will, if necessary, download and install all the specified versions of the platforms and libraries required by the sketch at compile time.

If you don't require to ensure that specific versions of platforms and libraries are used for the rest of eternity, you can default a sketch FQBN and upload port as follows:

```
arduino-cli board attach \
--fqbn arduino:avr:uno \
--port /dev/ttyUSB0

Default port set to: /dev/ttyUSB0
Default FQBN set to: arduino:avr:uno
```

If a `sketch.yaml` file didn't already exist, one will be created. If there was a `sketch.yaml`, it will have the following added to the end, regardless of whether there already was a default setup.

```
default_port: /dev/ttyUSB0
default_fqbn: arduino:avr:uno
```

Now compiling and uploading the sketch, for an Uno board, simply require

```
arduino-cli compile

Sketch uses 11636 bytes (36%) of program storage space.
Maximum is 32256 bytes.

Global variables use 308 bytes (15%) of dynamic memory,
leaving 1740 bytes for local variables. Maximum is
2048 bytes.

Used library Version Path
LibPrintf 1.2.13 /home/norman/Arduino/libraries/LibPrintf

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

This is followed by a simple "upload" command to transfer the new sketch to the board.

```
arduino-cli upload
```

If you have other, non-default profiles in the `sketch.yaml` file, you can still pick and choose between them in the manner described previously.

Note
The `default_port` setting in a profile is only applicable to the `compile` and `upload` commands. It is not used by the `monitor` command, for example. With `monitor`, you must still specify a port.

6.3.13 A Simple Makefile

One of the selling points of *arduino-cli* was to be able to simply use a `Makefile` and the *make* utility to build and flash sketches to boards. I have a simple, default `Makefile` which defines a number of defaults:

- The board is defaulted to arduino:avr:uno.
- The upload port is defaulted to /dev/ttyUSB0.
- The default ICSP is USBtinyISP.
- The default baud rate for the Serial Monitor is 115200.
- The default verbosity is silent. Well, as silent as possible!

The beauty of using a `Makefile` is that any of these defaults can be changed from the command line at build time.

Listing 6-9 shows my simple `Makefile`. This file is saved into the same location as the sketch's `.ino` files where I want to compile with *arduino-cli* and *make*.

Listing 6-9 A simple Makefile

```
# Compile and/or upload a sketch to an Arduino Uno board
# using the arduino-cli. This test was performed using
# arduino-cli version 0.32.2.

# Which board? See arduino-cli board listall uno.
BOARD = arduino:avr:uno

# Upload port. See arduino-cli board list.
PORT = /dev/ttyUSB0

# Config for Serial Monitor
CONFIG = baudrate=115200

# Which ICSP is in use? See arduino-cli board details \
# --fqbn arduino:avr:uno \
# --list-programmers.
ICSP = usbtinyisp

# Verbose compile and upload/program/bootloader? Values are
# blank, "-v" or "--verbose" only.
VERB =

#============================================================
# NOTE: The spaces at the start of each command line are NOT
# SPACES! They are a single TAB character. Make sure that
# your editor saves TABs as TABs and doesn't convert them to
# spaces or make will not work. A difficult one to debug!
#============================================================
.PHONY:
compile:
    arduino-cli compile $(VERB) --fqbn $(BOARD)

upload: compile
    arduino-cli upload $(VERB) --fqbn $(BOARD) --port $(PORT)

program: compile
    arduino-cli upload $(VERB) --fqbn $(BOARD) \
    --port $(PORT) --programmer $(ICSP)

.PHONY:
bootloader:
    arduino-cli burn-bootloader $(VERB) --fqbn $(BOARD) \
    --programmer $(ICSP)
```

```
.PHONY:
monitor:
    arduino-cli monitor --port $(PORT) --config $(CONFIG)

.PHONY:
details:
    arduino-cli board details --fqbn $(BOARD)

.PHONY:
settings:
    @echo "Board: $(BOARD)."
    @echo "Port: $(PORT)."
    @echo "ICSP: $(ICSP)."
    @echo "Config: $(CONFIG)."
    @echo "Verbose: '$(VERB)' (Blank, -v or --verbose only)."

.PHONY:
help:
    @echo "compile: Compile a sketch."
    @echo "upload: Upload a sketch using the bootloader."
    @echo "program: Upload using an ICSP."
    @echo "bootloader: burn the bootloader."
    @echo "monitor: Open the Serial Monitor."
    @echo "details: Display details about the board."
```

There are a number of commands which this `Makefile` will execute on our behalf.

6.3.13.1 Verbosity

The default for the compile, upload, program, and bootloader commands is, as in the Arduino IDE, to run as quietly as possible. If you wish to see more details for these commands, then you may either edit the text file named `Makefile` to default the `VERB` setting to either `-v` or `--verbose`. This will be passed to the commands listed, and full details of those actions will be displayed.

You may also leave the default, in `Makefile`, as is, but pass one of the two permitted values, `-v` or `--verbose`, on the command line:

```
make VERB=-v compile
```

```
make VERB=--verbose upload
```

If you do change the `Makefile` default, but occasionally wish to execute the commands in quiet mode, then simply pass a blank value on the command line:

```
make VERB=
```

6.3.13.2 Compile

The `compile` command compiles the sketch in the current directory:

```
make compile
```

```
Sketch uses 2032 bytes (6%) of program storage space. Maximum
is 32256 bytes.

Global variables use 212 bytes (10%) of dynamic memory,
leaving 1836 bytes for local variables. Maximum is
2048 bytes.

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

Any of the settings listed by the settings command can be changed on the command line:

```
make BOARD=arduino:avr:mega compile
```

```
Sketch uses 2754 bytes (1%) of program storage space. Maximum
is 253952 bytes.

Global variables use 212 bytes (2%) of dynamic memory,
leaving 7980 bytes for local variables. Maximum is
8192 bytes.

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

You may, if you wish, turn on verbose compilation by passing a value for the `VERB` option:

```
make VERB=--verbose compile
```

```
Using board 'uno' from platform in folder:
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6

Using core 'arduino' from platform in folder:
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6

Detecting libraries used...
...
Large amount of compilation and linking details omitted here!
...
Sketch uses 2032 bytes (6%) of program storage space. Maximum
is 32256 bytes.

Global variables use 212 bytes (10%) of dynamic memory,
leaving 1836 bytes for local variables. Maximum is 2048 bytes.
Used platform Version Path

arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

6.3.13.3 Upload

The `upload` command uploads the sketch in the current directory, using the Arduino bootloader. The `Makefile` will always recompile the sketch prior to the upload as *arduino-cli* doesn't normally do this prior to an upload in the same way that the IDE does.

```
make upload
```

```
Sketch uses 2032 bytes (6%) of program storage space. Maximum
is 32256 bytes.

Global variables use 212 bytes (10%) of dynamic memory,
leaving 1836 bytes for local variables. Maximum is
2048 bytes.

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

By default, there is no indication that the upload worked, other than flickering lights on the attached Arduino board. If you wish to see the full upload details, pass a `VERB` option as described previously.

You can change the board and port, if necessary, to use something other than the defaults in `Makefile`:

```
make BOARD=arduino:avr:mega PORT=/dev/ttyACM0 upload
```

6.3.13.4 Program

The `program` command uploads a sketch from the current directory, using the configured ICSP device. The sketch will, as with the upload command, be recompiled prior to programming the board:

```
make program
```

```
Sketch uses 2032 bytes (6%) of program storage space. Maximum
is 32256 bytes.

Global variables use 212 bytes (10%) of dynamic memory,
leaving 1836 bytes for local variables. Maximum is
2048 bytes.

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

As with the `upload` command, there is no indication that the programming succeeded. You may enable verbose details by setting the `VERB` option on the command line.

You can, of course, change the ICSP device, used to program your Arduino board, on the fly:

```
make ICSP=arduinoisp program
```

6.3.13.5 Bootloader

The `bootloader` command burns a bootloader using the current ICSP setting:

```
make bootloader
```

The bootloader command defaults to quiet mode. If you need to change this, you can enable *avrdude*'s verbose output mode:

```
make VERB=-v bootloader
```

As with the `program` command, you can change the ICSP device on the command line:

```
make ICSP=arduinoisp bootloader
```

6.3.13.6 Monitor

The `monitor` command opens the Serial Monitor and displays the output from the currently uploaded sketch:

```
make monitor
```

```
Monitor port settings:
baudrate=115200
Connected to /dev/ttyUSB0! Press CTRL-C to exit.
...
```

The configuration can be changed to suit different baud rates and so on:

```
make CONFIG=baudrate=9600,dtr=on,bits=8 monitor
```

```
Monitor port settings:
baudrate=9600

Monitor port settings:
dtr=on

Monitor port settings:
bits=8

Connected to /dev/ttyUSB0! Press CTRL-C to exit.
...
```

6.3.13.7 Details

The `details` command displays information about the current board:

```
make details
```

```
Board name:           Arduino Uno
FQBN:                 arduino:avr:uno
...
Lots and lots of output here!
....
Programmers: ID         Name
             arduinoisp ArduinoISP
             avrisp     AVR ISP
             usbtinyisp USBtinyISP
             ...
```

The output is vast, too much to list here, and doesn't even need the board to be connected.

The board settings can be changed permanently by editing the `Makefile` or by passing a different board ID on the command line:

```
make BOARD=attiny:avr:ATtinyX5 settings
```

```
Board name: ATtiny25/45/85
FQBN: attiny:avr:ATtinyX5
...
Lots and lots of output here!
....
Programmers: ID         Name
```

Surprisingly, it appears that there are no ICSP devices listed as being suitable for programming ATtiny microcontrollers. This is news to me, and my USBtinyISP, as that's what I've been using for years to upload code to my ATtiny85s!

6.3.13.8 Settings

Using the `settings` command will display all the current defaults configured in the `Makefile`:

```
make settings
```

```
Board:   arduino:avr:uno.
Port:    /dev/ttyUSB0.
ICSP:    usbtinyisp.
Config:  baudrate=115200.
Verbose: '' (Blank, -v or --verbose only).
```

While it is obviously possible to change these defaults by passing different values on the command line, it is not particularly useful to do that when running the `settings` command!

```
make BOARD=arduino:avr:nano VERB=-v settings
```

```
Board:   arduino:avr:nano.
Port:    /dev/ttyUSB0.
ICSP:    usbtinyisp.
Config:  baudrate=115200.
Verbose: '-v' (Blank, -v or --verbose only).
```

6.3.13.9 Help

The `help` command simply displays a list of the commands that the `Makefile` supports:

```
make help
```

```
compile:    Compile a sketch.
upload:     Upload a sketch using the bootloader.
program:    Upload using an ICSP.
bootloader: Burn the bootloader.
monitor:    Open the Serial Monitor.
details:    Display details about the board.
```

6.3.14 Library Manager

The library manager tool in the Arduino IDE is replicated by *arduino-cli*. Libraries can be

- Downloadable in the manner of the library manager in the Arduino IDE
- Zip files, located on your local computer
- Online, in GitHub repositories

The library manager uses the setting `directories.user` in the configuration file. This corresponds to your Arduino Sketchbook location if you have the Arduino IDE installed. Libraries are installed into the `libraries` directory, beneath the Sketchbook directory. This means that *arduino-cli* and the IDE will both have access to the same libraries—unless you amend the configuration for *arduino-cli*.

6.3.14.1 Downloadable Libraries

As an example, I much prefer to use the *LibPrintf* library, by Embedded Artistry, rather than using `Serial.print()` and `Serial.Println()` calls. I can search for this library as follows:

```
arduino-cli lib search libprintf
```

```
Downloading index: library_index.tar.bz2 downloaded
Name: "LibPrintf"
  Author: Embedded Artistry
  Maintainer: Embedded Artistry <contact@embeddedartistry.com>
  Sentence: Library adding support for the printf family of
            functions to the Arduino SDK.
  Paragraph: This library provides support for printf() and
             other printf-like functions with full
             format-string support. Default output is to
             Serial, but can be customized.
Website: https://github.com/embeddedartistry/arduino-printf
Category: Communication
Architecture: *
Types: Contributed
Versions: [1.0.0, 1.0.1, 1.1.2, 1.1.3, 1.2.6,
           1.2.10, 1.2.13]
Provides includes: LibPrintf.h
```

If there were more than one library with “libprintf” in its name, they would all be listed. Installation of the library is next:

```
arduino-cli lib install libprintf

Downloading LibPrintf@1.2.13
LibPrintf@1.2.13 downloaded
Installing LibPrintf@1.2.13
Installed LibPrintf@1.2.13
```

Listing 6-10 is a quick sketch to show the library in action. It was, of course, created with

```
arduino-cli sketch new ~/Arduino/LibPrintfTest
```

Listing 6-10 LibPrintfTest.ino

```
#include "LibPrintf.h"

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

void loop() {
  static int entryCount = 0;
  printf("%d: Hello World!\n", entryCount);
  entryCount++;
  delay(500);
}
```

Compilation is as before. I could have used *make* and my `Makefile` here, I agree, but this is about *arduino-cli* and not about my `Makefile`!

```
cd ~/Arduino/LibPrintfTest
arduino-cli compile --fqbn arduino:avr:uno
```

```
Sketch uses 11484 bytes (35%) of program storage space.
Maximum is 32256 bytes.

Global variables use 306 bytes (14%) of dynamic memory,
leaving 1742 bytes for local variables. Maximum is
2048 bytes.

Used library Version Path
LibPrintf 1.2.13 /home/norman/Arduino/libraries/LibPrintf

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

Notice that the library/libraries used have been listed in the compilation output. If you don't see the expected libraries listed, then check your source for problems.

The next step is to upload to my Uno board and monitor the output:

```
arduino-cli upload --fqbn arduino:avr:uno --port /dev/ttyUSB0
```

```
arduino-cli monitor --port /dev/ttyUSB0
```

```
Connected to /dev/ttyUSB0! Press CTRL-C to exit.
0: Hello World!
1: Hello World!
2: Hello World!
3: Hello World!
...
```

6.3.14.2 Zip File Libraries

A locally stored zip file can be installed using the `--zip-path` option.

```
arduino-cli lib install --zip-path \
/data/SourceCode/ArduinoLibraries/AVRint0.zip
```

Unfortunately, installing Zip and GitHub libraries is disabled by default. Attempting to install either will result in this error message:

```
--git-url and --zip-path are disabled by default, for more
information see:
https://arduino.github.io/arduino-cli/0.30/configuration/#configuration-
    keys
```

If we look at the named URL and scroll down, we see this information:

Ensure `enable_unsafe_install` - set to `true` to enable the use of the `--git-url` and `--zip-file` flags with `arduino-cli lib install`. These are considered "unsafe" installation methods because they allow installing files that have not passed through the Library Manager submission process.

So, we need to make a configuration change before we can install these libraries. We can use *arduino-cli* to assist:

```
arduino-cli config set library.enable_unsafe_install true
```

And after that, we can try again:

```
arduino-cli lib install --zip-path \
/data/SourceCode/ArduinoLibraries/AVRint0.zip
```

```
--git-url and --zip-path flags allow installing untrusted
files, use it at your own risk.

Library installed.
```

6.3.14.3 Git Repository Libraries

A version controlled library from a GitHub or GitLab repository can be installed using the `--git-url` option.

As we already know, Zip and GitHub library installations are disabled by default. Attempting to install either will result in an error message, so if we have not yet configured *arduino-cli* to allow installation of these libraries, we have a configuration change to make:

```
arduino-cli config set library.enable_unsafe_install true
```

And after that, we can install the library:

```
arduino-cli lib install --git-url \
https://github.com/NormanDunbar/avrDigitalPin
```

```
--git-url and --zip-path flags allow installing untrusted
files, use it at your own risk.

Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Compressing objects: 100% (18/18), done.
Total 18 (delta 2), reused 9 (delta 0), pack-reused 0

Library installed
```

The URL is the actual repository URL, the one you would use when opening the repository in a browser or cloning it with a git clone command.

6.3.14.4 Listing Libraries

The list of installed libraries can be displayed using the following command:

```
arduino-cli lib list
```

The output is far too wide to be meaningfully represented on the page unfortunately. For each library installed, the listing will show

- The library name, “Adafruit Unified Sensor,” for example.
- The installed version, 1.1.7, for example.
- Any available upgrade versions, 1.1.8, for example.
- The installed location, usually `LIBRARY_LOCATION_USER`, which is a reference, I assume, to the configuration file.
- A description. All of mine are blank.

The list can be limited to only those libraries which have updates available:

```
arduino-cli lib list --updatable
```

The listed libraries can all be upgraded by

```
arduino-cli lib upgrade
```

6.3.14.5 Library Examples

As we all know, installing most libraries in the IDE results in a number of hopefully useful example sketches also being installed. Can we see, compile, and upload those examples using *arduino-cli*? Of course!

First, we list the available examples:

```
arduino-cli lib examples libprintf
```

```
Examples for library LibPrintf
  - /home/.../libraries/LibPrintf/examples/default_to_serial
  - /home/.../libraries/LibPrintf/examples/override_putchar
  - /home/.../libraries/LibPrintf/examples/specify_print_class
```

Then we compile one of the example sketches:

```
arduino-cli compile --fqbn arduino:avr:uno \
/home/.../libraries/LibPrintf/examples/default_to_serial
```

```
Sketch uses 11660 bytes (36%) of program storage space.
Maximum is 32256 bytes.

Global variables use 300 bytes (14%) of dynamic memory,
leaving 1748 bytes for local variables. Maximum is
```

```
2048 bytes.

Used library Version Path
LibPrintf 1.2.13 /home/norman/Arduino/libraries/LibPrintf

Used platform Version Path
arduino:avr 1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

After the compilation, we upload the example sketch to our Uno board:

```
arduino-cli upload --fqbn arduino:avr:uno \
--port /dev/ttyUSB0 \
/home/.../libraries/LibPrintf/examples/default_to_serial
```

And, after checking the source for the baud rate, we configure the Serial Monitor and watch the output from the example:

```
arduino-cli monitor --port /dev/ttyUSB0 \
--config baudrate=115200
```

```
Monitor port settings:
baudrate=115200
Connected to /dev/ttyUSB0! Press CTRL-C to exit.

I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
...
```

You may have noticed that this process has shown how to

- List available examples for a library
- Compile an example using its full path
- Upload the compiled example, again using its full path

6.3.15 Board Manager

Previously, in Section 6.3.3, I briefly mentioned that *arduino-cli* can now be used to add various configuration data to the configuration file, `arduino-cli.yaml`. This process is how we add boards to, or remove boards from, the *arduino-cli* environment.

As I have a number of ATtiny85 devices, I would, occasionally, like to throw together a quick Arduino-style sketch to upload. The *arduino-cli* board manager is how I would carry out this task.

First, I would obtain the appropriate JSON file to add to *arduino-cli.yaml*, and once the URL for the JSON file has been obtained, I can add it to the board manager as follows:

```
arduino-cli config add board_manager.additional_urls \
https://raw.githubusercontent.com/damellis/attiny/ide-1.6.x-boards-
   manager/package_damellis_attiny_index.json
```

The full URL should be on a single line, but it has wrapped here to fit on the page. There are no spaces or line breaks in the URL.

Once added, I need to update the core index so that it is aware of the change and can download the appropriate index files for the new boards:

```
arduino-cli core update-index
```

```
Downloading index: package_index.tar.bz2 downloaded
Downloading index: package_damellis_attiny_index.json
downloaded
```

Now, as we discovered, until I install a core for these boards, attempting to list them will not return any results. We need to search for a suitable core and install it:

```
arduino-cli core search attiny
```

```
ID         Version Name
attiny:avr 1.0.2   attiny
```

And having discovered the core that is required, it can be installed:

```
arduino-cli core install attiny:avr
```

```
Downloading packages...
attiny:avr@1.0.2 downloaded
Installing platform attiny:avr@1.0.2...
Configuring platform....
Platform attiny:avr@1.0.2 installed
```

Now, I should be able to see the new boards:

```
arduino-cli board listall attiny
```

```
Board Name       FQBN
ATtiny24/44/84   attiny:avr:ATtinyX4
ATtiny25/45/85   attiny:avr:ATtinyX5
```

When compiling, I must specify the actual cpu in use; otherwise, the default is to use the ATtiny25:

```
arduino-cli compile \
--fqbn attiny:avr:ATtinyX5:cpu=attiny85
```

```
Sketch uses 446 bytes (5%) of program storage space. Maximum
is 8192 bytes.

Global variables use 9 bytes (1%) of dynamic memory, leaving
503 bytes for local variables. Maximum is 512 bytes.

Used platform Version Path
attiny:avr    1.0.2
/home/norman/.arduinocli/packages/attiny/hardware/avr/1.0.2
arduino:avr   1.8.6
/home/norman/.arduinocli/packages/arduino/hardware/avr/1.8.6
```

Likewise when uploading:

```
arduino-cli upload \
--fqbn attiny:avr:ATtinyX5:cpu=attiny85 \
--programmer usbtinyisp
```

Obviously, I need to use an ICSP device, my USBtinyISP again, as the ATtiny series doesn't have a bootloader.

ATmega328P Configuration and Management 7

In this, and the next two chapters, I'll dig deeper into the ATmega328P microcontroller itself and take a look at some very important features of the device, many of which are either ignored, unused, or just too confusing as you read through the data sheet.

Chapter 7 looks at those features which allow the ATmega328P to be configured and which provide a certain amount of protection and power reduction for application code—this can help it run on battery power alone in some cases.

Chapters 8 and 9 look deeper at the actual, usable hardware of the device. These two chapters will, hopefully, assist you in developing projects using the AVR C++ rather than the Arduino Language—should that be your wish. Even if you have no wish to discard the Arduino Language, the source code for the numerous functions I discussed back in Chapters 2, 3, and 4 rely on the information you will find here.

7.1 ATmega328P Fuses

Fuses are special areas of the AVR microcontroller by which the device can be configured with a number of different settings. The ATmega328 has three fuses—low, high, and extended—each of which has different responsibilities.

Warning
It is easy to brick your device by setting incorrect fuses, so be exceedingly careful and don't go playing around with things you don't understand, yet! Ask me how I know!

When a brand-new device is shipped from the factory, the fuses are set to a default configuration, and this default may not be as required for any specific purpose.

There are many online fuse calculators—www.engbedded.com/fusecalc is a good one—where you can select various options and be shown the commands and/or values with which to program your fuses.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_7

Table 7-1 ATmega328P low fuse bits

Bit	Fuse	Purpose	Default
7	CKDIV8	Divide clock by eight	Programmed
6	CKOUT	Clock output on pin `PB0`	Unprogrammed
5	SUT1	Select startup time, bit 1	Unprogrammed
4	SUT0	Select startup time, bit 0	Programmed
3	CKSEL3	Select clock source, bit 3	Programmed
2	CKSEL2	Select clock source, bit 2	Programmed
1	CKSEL1	Select clock source, bit 1	Unprogrammed
0	CKSEL0	Select clock source, bit 0	Programmed

Note
You should also be very careful to remember that a fuse that is unprogrammed has a bit value of one, while a programmed fuse has a bit value of zero. The online fuse calculators show programmed fuses with a ticked checkbox and an unprogrammed fuse with a clear checkbox. This is the opposite way round from what I consider normal, but that's how the AVR microcontroller works.

7.1.1 Low Fuse Bits

The interesting bits in the low fuse, at least those of interest to Arduino users, are the `SUT` and `CKSEL` fuses and possibly `CKDIV8`. The latter simply divides the chosen clock source (see `CKSEL`) by eight to obtain the final system clock speed.

Table 7-1 describes the low fuse bits.

7.1.1.1 SUT Fuse Bits

The `SUT` fuses comprise of two bits in the low fuse byte. These determine how much of a delay there will be after the power is applied or the system reset in order to allow everything to settle down and become stable. Some power supplies, for example, need a bit more time than others to stabilize at the required voltage, and if the Arduino (or AVR microcontroller) was to start running too soon, it could "brown out" or otherwise not behave correctly. The startup and reset delays also allow the oscillator to stabilize.

As each different possible oscillator has different startup delays from power on, or from a wake-up call during a Power Save sleep mode, you are advised to check the data sheet for your particular device as there are numerous variations in each of the settings for these two fuse bits for each particular oscillator.

7.1.1.2 CKSEL Fuse Bits

The `CKSEL` fuse bits, shown in Table 7-2, select the desired internal or external oscillator to be used as the system clock. On the Arduino boards, this is always an external crystal oscillator running at 16 MHz, so the other options are not of much use there, but can be used if you build your own boards and do away with the crystal.

Table 7-2 ATmega328P CKSEL oscillator choice fuse

CKSEL3:0	Oscillator to Be Used
0b0000	External clock
0b0001	Reserved, do not use
0b0010	Internal calibrated 8 MHz RC oscillator
0b0010	Internal 128 KHz RC oscillator
0b0100–0101	Low-frequency crystal oscillator
0b0110–0111	Full swing crystal oscillator
0b1000–1111	Low-power crystal oscillator

Table 7-3 ATmega328P CKSEL oscillator frequency ranges

CKSEL3:1	Frequency
0b100	0.4–0.9 MHz
0b101	0.9–3 MHz
0b110	3–8 MHz
0b111	8–16 MHz

When using the low-frequency crystal oscillator, the `CKSEL3:1` bits determine the frequency of the oscillator in use, as shown in Table 7-3. Our Arduinos with their 16 MHz oscillators will be using the 0b111 option.

If a low-power crystal oscillator is being used, then the `SUT1:0` fuse bits can be used to specify differing power-on/wake-up delays as defined in the data sheet.

7.1.2 Low Fuse Factory Default

From the factory, the ATmega328P is shipped with this fuse set to 0x62 which is 0b0110 0010 and is configured as follows:

`CKDIV8`	Bit 7 is programmed (zero) and causes the system clock to be divided by eight. See also the `CKSEL` bits.
`SUT0`	Bit 4 is programmed and sets the default startup time, after a reset, to 14 clock cycles plus 65 milliseconds over and above that of the power-up time. From initial power up or wake from Power Save sleep mode, the startup delay time is six clock cycles. These delays allow the device and power supply to settle down before anything important starts running.
`CKSEL3:0`	Bits 3–0 are set to 0b0010 which defines that the system clock is configured to be the internal 8 MHz oscillator. Given that the default also programs `CKDIV8`, then the device will only be running at 1 MHz. This does mean that while slower, it draws less current and can run from a lower supply voltage.

Fuses `CKOUT` (bit 6), `SUT1` (bit 5), and `CKSEL1` (bit 1) remain unprogrammed by default, and this means that the system clock pulse does not appear on `PORTB`, pin zero (`CKOUT`). `SUT1` affects the startup time, and `CKSEL1` affects the default system clock.

7.1.3 Arduino Low Fuse Settings

The Arduino boards running with the ATmega328P microcontroller set the low fuse to 0xFF or 0b1111 1111—at least on the Nano with an ATmega328P, the Duemilanove with an ATmega328P, the Diecimila, and the Uno.

This disables everything covered by this fuse and means that, according to the data sheet

- The chosen oscillator is not divided by 8 (`CKDIV8` unprogrammed); an external crystal oscillator in the range 8–16 MHz is in use (`CKSEL3:1` unprogrammed) with a slow rising power supply (`SUT1:0` unprogrammed alongside `CKSEL0` unprogrammed). This allows for 16,384-clock cycle startup delay at power on or from a power save sleep wake-up call, with an additional 14 clocks plus 65 milliseconds from a reset.
- `SUT1:0`, which defines the startup time for the device, is set to 0b11, and this is a setting described as *reserved* in the ATmega328P data sheet. Hmmm, interesting.

7.1.4 High Fuse Bits

Table 7-4 lists the high fuse bits and their purpose.

High fuse bits of interest to Arduino users are the `BOOTSZ` bits shown in Table 7-5 and the `BOOTRST` fuse bit shown in Table 7-6. It's probably not wise to play with the others!

The Arduino Uno sets the `BOOTSZ1:0` fuses to 0b11, while the Duemilanove, Diecimila, and Nano use 0b01, resulting in a bootloader of 256 words (512 bytes) for the Uno and 1,024 words (2,048 bytes) for the others.

The `BOOTRST` fuse has two possible values. The Arduino always clears this fuse to zero, meaning programmed. The possible values for this fuse are shown in Table 7-6. The ATmega328P's RESET vector can be configured to point either at the bootloader or at the application address space in Flash RAM.

Table 7-4 ATmega328P high fuse bits

Bit	Fuse	Purpose	Default
7	RSTDISBL	Disable RESET pin and use for I/O	Unprogrammed
6	DWEN	Debug wire enable	Unprogrammed
5	SPIEN	Enable serial programming and data downloading	Programmed
4	WDTON	Watchdog Timer always on	Unprogrammed
3	EESAVE	Preserve EEPROM during chip erasure	Unprogrammed
2	BOOTSZ1	Select bootloader size, bit 1	Programmed
1	BOOTSZ0	Select bootloader size, bit 0	Programmed
0	BOOTRST	Select RESET vector location	Unprogrammed

Table 7-5 ATmega328P BOOTSZ bootloader fuse

BOOTSZ1:0	Boot Area Size	Application Address	Bootloader Address
0b11	256 Words	0–0x3EFF	0x3F00–0x3FFF
0b10	512 Words	0–0x3DFF	0x3E00–0x3FFF
0b01	1,024 Words	0–0x3BFF	0x3C00–0x3FFF
0b00	2,048 Words	0–0x37FF	0x3800–0x3FFF

Table 7-6 ATmega328P BOOTRST reset vector fuse

BOOTRST	Purpose
0	The RESET vector points to the bootloader address (see Table 7-5)
1	The RESET vector points at the application start address (address zero)

The Arduino therefore sets this fuse so that the RESET vector will start executing at the bootloader rather than the application code, which is a good thing as it means that you will be able to reprogram the Arduino as the bootloader watches for programming instructions before jumping into the application code if none are forthcoming.

7.1.5 High Fuse Factory Default

From the factory, this fuse defaults to 0xD9 or 0b1101 1001 and means that

`SPIEN`	Bit 5 is programmed so serial programming (SPI) and data downloading are enabled.
`BOOTSZ1:0`	Bits 2 and 1 are programmed, giving the device 2,048 words of flash starting at address 0x3800 for use as the bootloader area.

The remaining fuses are not programmed, therefore:

`RSTDISBL`	Bit 7 being unprogrammed means that the RST pin, physical pin 1, is not disabled and works as a reset pin. If this fuse is programmed, you cannot program the AVR microcontroller unless you use a high-power programmer device. (This is the fuse setting that bricked one of my ATtiny85 devices when I managed to program it!)
`DWEN`	Bit 6 being unprogrammed results in the Debug Wire interface being disabled. Debug Wire is beyond the scope of this book; however, you may wish to watch this video, www.youtube.com/watch?v=7Pm9_LtXdhM, on YouTube and check out the code on GitHub, https://github.com/felias-fogg/dw-link, for the DW-Link sketch that turns your Arduino Uno into a hardware debugger!
`WDTON`	Bit 4 not being programmed means that the Watchdog Timer is not always running and unable to be disabled. This means that the Arduino can program the watchdog on or off as required.
`EESAVE`	Bit 3 remaining unprogrammed results in the internal EEPROM being wiped clean each and every time the AVR microcontroller is programmed. If you need to save data in the EEPROM between program changes and uploads, then you should maybe program this bit or use a command-line option in `programmers.txt` to prevent EEPROM wipes.
`BOOTRST`	Bit 0 being unprogrammed means that the device will not jump to the bootloader address at startup. On reset, the device will start execution at the normal reset vector at address zero.

7.1.6 Arduino High Fuse Settings

This fuse is set to 0xDE 0b1101 1110 on the Uno, but to 0xDA, 0b1101 1010 on the Duemilanove, the Diecimila, and the Nano. This programs the following fuses:

`SPIEN`	Bit 5 is the same as the factory default and enables SPI.

Table 7-7 ATmega328P extended fuse bits

Bit	Fuse	Purpose	Default
2	BODLEVEL2	Brown out detection trigger level, bit 2	Unprogrammed
1	BODLEVEL1	Brown out detection trigger level, bit 1	Unprogrammed
0	BODLEVEL0	Brown out detection trigger level, bit 0	Unprogrammed

Table 7-8 ATmega328P BOD voltage ranges

BODLEVEL2:0	BOD V_{min}	BOD $V_{typical}$	BOD V_{max}
0b111	Disabled	Disabled	Disabled
0b110	1.7 V	1.8 V	2.0 V
0b101	2.5 V	2.7 V	2.9 V
0b100	4.1 V	4.3 V	4.5 V

`BOOTSZ1`	Bit 2 is unprogrammed on the Uno and programmed on the other boards.
`BOOTRST`	Bit 0 is programmed.

The Uno is programmed with 128 words of bootloader space, while other boards based on the ATmega328P use 512 words. A reset causes execution to start from the bootloader address.

7.1.7 Extended Fuse Bits

The extended fuse byte controls the BOD or brown out detector in the microcontroller. Not all bits of this fuse byte are used, and the relevant bits are described in Table 7-7.

All other bits of the extended fuse are unused and should always be programmed as 1 to avoid possible problems. Table 7-8 shows how the BOD fuse bits can be configured.

All other values for the `BODLEVEL2:0` fuse bits are reserved and must not be used.

7.1.8 Extended Fuse Factory Default

From the factory, this fuse is set to 0xFF or 0b1111 1111—so the brown out detection is disabled. Only fuse bits 2 to 0 are used (`BODLEVEL2:0`); the rest should remain as 1.

7.1.9 Arduino Extended Fuse Settings

The Arduino sets this fuse to 0xFD or 0b1111 1101 which sets the brown out detection threshold voltage to 2.7 V as only `BODLEVEL1` is programmed.

This is *potentially* a bad setting as the Arduino boards are running with an external 16 MHz crystal, and the data sheet advises that this will only be reliable between 4 V and 5.5 V—so this fuse should really be set to 0xFC, 0b1111 1100, to give a 4.1 V threshold.

7.2 Brown Out Detection

A brown out is a feature of electrical supplies when the supply voltage is not stable and varies up and down from the typical voltage that the supply should be.

Some electrical devices either don't run, don't run properly, get confused, and/or reset themselves if the supply voltage goes too low.

Brown out detection, or BOD, is a means by which the AVR microcontroller keeps a watchful eye on the power supply (VCC) and, when it drops below a configured level, initiates a system reset condition by pulling and holding the RESET pin low internally—the pin itself is not physically pulled low. When the power supply stabilizes again, the reset condition will be released, allowing the board to restart.

The BOD is enabled and configured using the `BODLEVEL2:0` fuses as described in Section 7.1.7. Remember, a fuse is not programmed when it has a value of one, but is programmed when it has the value zero.

The Arduino boards using the ATmega328P microcontroller are fused so that the BOD is 2.7 V typical, while a brand-new ATmega328P, supplied from the factory, will have the fuses unprogrammed, and, thus, BOD will be disabled (0b111).

Warning
Given that the data sheet for the ATmega328P shows that the minimum voltage for a 16 MHz AVR microcontroller is 4 V, the default setting for an Arduino Uno, in this case, is a little out of range!

The default fuse settings can be found in the `\$ARDINST/boards.txt` file. For a Nano with the ATmega328p, Duemilanove, and Uno, the extended fuse is set to 0xFD. This is 0b1111 1101 which means that `BODLEVEL2:0` is 0b101 and that sets the BOD threshold at 2.7 V, which is a long way below the voltage which the data sheet says is safe for a 16 MHz system.

The ATmega328P is stable when running with an external 16 MHz crystal, with VCC between 4.0 V and 5.5 V. Above this and the AVR microcontroller will probably let the magic blue smoke out and stop working; below this and the ATmega328P will possibly not work correctly, and this may affect any sensors that are attached especially if accurate timings are required.

Note
It is a well-known fact that all electronic devices seem to run on magic blue smoke. The reason for this belief is that when you somehow let the smoke out, most electronic devices stop working!

The available BOD settings on an ATmega328P are shown in Table 7-9.

The *typical* values are what the BOD settings define and are highlighted in bold in Table 7-9.

The ATmega328P can run safely at 16 MHz if it has power supplied at at least 4.0 V. Anything below 4.0 V means the chip is likely to misbehave. This would be a problem if the device were being used with sensors to take measurements or relied on accurate timings.

To prevent this possible problems, the BOD trigger threshold voltage should be set to the 4.3 V typical setting shown earlier, by setting the `BODLEVEL2:0` fuses to 0b100.

The trigger level has a built-in hysteresis to ensure that the BOD doesn't keep triggering and putting the AVR microcontroller into a BOD reset loop as the supply voltage fluctuates. This hysteresis allows the voltage to briefly drop below the threshold level, and if it quickly rises back above it, no BOD reset will take place.

The two hysteresis levels should be interpreted as

$$V_{bot}+ = V_{bot} + V_{hyst}/2$$

and

$$V_{bot}- = V_{bot} - V_{hyst}/2$$

where the value for V_{bot} is the typical voltage value from Table 7-9 for the BODLEVEL2:0 fuse bit settings on the board. For the Arduino Uno and friends, it is set to 2.7 V. The data sheet for the ATmega328P lists the hysteresis as 50 millivolts, which implies that the typical values for an Arduino board running with an ATmega328P at 16 MHz are

$$V_{BOT}+ = 2.7\ V + 25\ mV \ = \ 2.725\ V$$

and

$$V_{BOT}- = 2.7 \ - +25\ mV \ = \ 2.675\ V$$

The Arduino fuse settings *should* be defining the BOD level to be 4.3 V, but are not, at least at the time of writing.

Some AVR microcontrollers allow the BOD to be disabled in software; others don't. The ATmega328P is one of the devices that can disable the BOD—this is useful when entering various sleep modes as the BOD uses power that may be scarce, but disabling the BOD could leave your project open to apparently random resets if the voltage isn't stable.

7.3 The Watchdog Timer

The Watchdog Timer (WDT) is an internal *safety switch* which can be used to prevent code hang-ups or runaways and so on or which can be used to fire off an interrupt every so often. It runs off of a dedicated internal 128 KHz oscillator and can be programmed to fire at a number of preset intervals.

The Watchdog Timer interrupt can be used to wake the device from the bottom of a really deep sleep; this can be used to save power and preserve battery life. The AVR microcontroller can be put into a Power Down sleep mode, and the interrupt will wake it periodically, do some essential processing, and then put it back to sleep again ready for its next wake-up call.

Table 7-9 ATmega328P BOD voltage ranges

BODLEVEL2:0	BOD V_{min}	BOD $V_{typical}$	BOD V_{max}
0b111	BOD is disabled		
0b110	1.7 V	**1.8 V**	2.0 V
0b101	2.5 V	**2.7 V**	2.9 V
0b100	4.1 V	**4.3 V**	4.5 V

Sleep modes are discussed in Section 7.4.1.

7.3.1 Watchdog Timer Modes of Operation

The Watchdog Timer has three separate modes of operation:

Watchdog Reset (WDR)	If the Watchdog Timer has not been reset within the timeout period, the whole AVR microcontroller will be forced to a reset. On restarting after the reset, bit `WDRF` will be set to a 1 in the MCU Status Register, `MCUSR`. The `WDRF` bit can be interrogated to determine if the AVR microcontroller was reset by the Watchdog or otherwise. On waking from a Watchdog-induced reset, the Watchdog Timer is still enabled; however, it is now firing with the *shortest* timeout period, not what you originally configured.
Watchdog Interrupt (WDI)	An interrupt will be fired at the end of every Watchdog Timer timeout period.
Interrupt and Reset	The interrupt will fire on the first Watchdog Timer timeout, and on the next timeout, the system will be reset, unless `WDIE` is set again to prevent it. As long as the `WDIE` bit is constantly being set, the system reset will not fire, only the interrupt.

The data sheet has the following warning about the use of the Watchdog Timer:

> If the Watchdog is accidentally enabled, for example by a runaway pointer or brown-out condition, the device will be reset and the Watchdog Timer will stay enabled. If the code is not set up to handle the Watchdog, this might lead to an eternal loop of time-out resets. To avoid this situation, the application software should always clear the Watchdog System Reset Flag (`WDRF`) and the `WDE` control bit in the initialization routine, even if the Watchdog is not in use.

It should be noted that the Arduino bootloader *might* be making this check, but that depends on the bootloader in question. It is therefore still possible that your code could lead to a constantly resetting AVR microcontroller under the circumstances mentioned in the data sheet. The Uno bootloader does disable the Watchdog immediately on starting, but the bootloader for the Duemilanove does not, and this board does indeed suffer from Watchdog Reset Loops.

The code shown in Listing 7-1 is from the Uno bootloader, Optiboot version 4.4, and shows the test being made and the Watchdog Timer disabled.

Listing 7-1 Arduino bootloader code to prevent Watchdog Timer reset loops

```
// Adaboot no-wait mod
ch = MCUSR;
MCUSR = 0;
if (!(ch & _BV(EXTRF))) appStart();                          (1)

// Bootloader runs here.
...

void appStart() {
  watchdogConfig(WATCHDOG_OFF);                              (2)
  __asm__ __volatile__ (
     ...
    // Jump to RST vector
    "clr r30\n"                                             (3)
    "clr r31\n"
#endif
    "ijmp\n"                                                (4)
  );
}
```

(1) This tests the `EXTRF` bit to determine if an external reset was performed and, if not, jumps to `appStart()` to run the sketch code, *and not* the bootloader code.
(2) This disables the Watchdog.
(3) This clears registers `R30` and `R31` to zero.
(4) This executes the code at the address held in registers `R30` and `R31`, which is where the sketch code begins.

We can hopefully see that the Optiboot bootloader will always start the application if, and only if, the board was reset by *any other method* than pressing the reset switch. Pressing reset[1] would appear to be the only manner in which the bootloader will run on the Uno. The `EXTRF` bit in `MCUSR` indicates that the board was reset externally.

You should be aware that the `BOOTRST` fuses in the ATmega328P determine the starting position of the RESET vector, which is where the sketch code should start executing. On the Arduino boards, the default is to be at address zero. However, this can be changed. The Optiboot code assumes that this will *always* be address zero. Beware if you go changing the `BOOTRST` fuse!

7.3.2 Amended Sketch setup() Function

Perhaps the `setup()` code in Listing 7-2 *could* be added at the top of our sketches to ensure that potentially rogue Watchdog Timer reset loops can, possibly, be avoided.

I say *could* because the Optiboot bootloader for the Uno zeroes out the `MCUSR` register, so it is always zero when our sketch starts executing. This isn't too much of a problem, as this particular bootloader safely disables the Watchdog before any chance of a reset loop being triggered.

For the Duemilanove's bootloader, this is not done, and so if a Duemilanove board ever suffers a Watchdog reset, it will *always* enter a Watchdog reset loop and constantly reset itself.

This happens because the Watchdog is internally reconfigured to time out in 16 mS, and as the bootloader takes longer than 16 mS to run, the Watchdog constantly reboots the board while the bootloader is still executing.

Listing 7-2 Amended setup() function to prevent Watchdog Timer reset loops

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

void ensureWDTisOff() {
  // wdt_disable() will disable interrupts and
  // call wdt_reset first, then disable the WDT.
  wdt_disable();                                             (1)
```

[1]Or using an FTDI or ICSP device that effectively presses reset for you, of course.

```
    // Reset the MCU Status Register.
    MCUSR=0;                                                  (2)
}

void setup() {
    // Ensure the WDT has not gone rogue!
    ensureWDTisOff();

    // Do the sketch's own initialisation here.
    pinMode(...);
    ...
}
```

(1) Calling `wdt_disable()` will turn off global interrupts to prevent them from interfering with the timed sequence of instructions to disable/enable the WDT, clear the Watchdog Timer counter, and then completely disable the Watchdog Timer. After disabling, interrupts will be enabled again if they were enabled when `wdt_disable()` was called.
(2) Reset the MCU Status Register to a known state.

The Watchdog Timer can now be enabled to a known and desired state, if required, in `setup()` and a call to `wdt_reset()` executed each time through the `loop()` and any other long-running processes.

7.3.3 Watchdog Timer Reset

To keep your code running, without the Watchdog Timer resetting the AVR microcontroller, your code must, periodically, clear the Watchdog Timer counter by executing the `wdr` assembly language instruction prior to the Watchdog Timer timeout period expiring. This instruction has been defined in the AVRLib code as follows:

```
#define wdt_reset() __asm__ __volatile__ ("wdr")
```

So, if you are using the Watchdog Timer, your code must include the file `avr/wdt.h` and call `wdt_reset()` at regular intervals. How regular? You must reset the Watchdog Timer within the configured timeout period.

Note
The call to `wdt_reset()` is only necessary when the WDT is configured to run in "WDT Reset" or "Reset and Interrupt" modes. It is not necessary to call `wdt_reset()` when running in "WDT Interrupt" (WDI) mode as that mode does not cause the microcontroller to be reset.

Table 7-10 Watchdog control register

Bit	Name	Comments
7	WDIF	Watchdog interrupt flag
6	WDIE	Watchdog interrupt enable
5	WDP3	Watchdog timer prescaler, bit 3
4	WDCE	Watchdog change enable
3	WDE	Watchdog reset enable
2	WDP2	Watchdog timer prescaler, bit 2
1	WDP1	Watchdog timer prescaler, bit 1
0	WDP0	Watchdog timer prescaler, bit 0

7.3.4 The Watchdog Timer Control Register

The Watchdog Timer Control Register, `WDTCSR`, is eight bits wide, and the individual bits have the usages defined in Table 7-10.

The bits are used as follows:

`WDIF` is set when the Watchdog Timer times out and the Watchdog Timer interrupt is enabled. If global interrupts are also enabled, then the Watchdog Timer interrupt ISR fires, and this bit will be cleared automatically.
User code may, if desired, clear this bit by writing a 1 to it when global interrupts are off. If it is not cleared, then the Watchdog Timer interrupt ISR will execute as soon as the global interrupts are subsequently re-enabled.

`WDIE` if set, enables Watchdog Timer Interrupt mode (`WDI`). Depending on the setting of the `WDE` bit

- If `WDE` is also set, the Watchdog Timer is now in Reset and Interrupt mode. The first timeout will execute the Watchdog Timer interrupt ISR, clear `WDIF` as in the preceding text, clear `WDIE` to disable WDI, and leave `WDE` set to ensure WDR mode. The second timeout will cause a system reset.
- If `WDE` is clear, then the mode is Watchdog Timer Interrupt or WDI. Each time the timeout expires, the Watchdog Timer Interrupt ISR will be executed. No system reset will occur.

`WDCE` is used in the timing sequence that allows the Watchdog Timer to be configured. The configuration allows for

- Setting or clearing the `WDE` bit
- Setting or clearing the prescaler bits, `WDP3:0`

`WDE` enables the Watchdog Timer in Watchdog Reset (WDR) mode. If the Watchdog Timer is not reset before the timeout period expires, the AVR microcontroller will be reset, and on restarting, bit `WDRF` (Watchdog Timer Reset Flag) in `MCUSR` will be set to 1 showing that the reset occurred due to the Watchdog Timer.
If `WDIE` is also set, then the Reset and Interrupt modes are active. The first timeout will cause the interrupt to fire and execute the ISR, will clear `WDIE` to disable the Watchdog Timer interrupt, and will enable Watchdog Reset (WDR) mode. The second timeout will reset the system, unless `WDIE` was again set to enable the Watchdog Timer interrupt.

`WDP3:0`	sets the Watchdog Timer prescaler to give the desired timeout period. If the Watchdog Timer counter is not cleared within this period, then the system will be reset, or the interrupt will be fired, depending on the settings on `WDE` and `WDIE`.

Warning
If the `WDTON` fuse has been programmed (i.e., has value 0), then you are unable to ever change bits `WDE` and `WDIE` in `WDTCSR`. The Arduino default is that this fuse bit is not programmed, so the Watchdog Timer can be enabled or disabled as you might wish.

With this fuse programmed, `WDE` is always 1, while `WDIE` is always 0, so the Watchdog Timer is always running in Watchdog Timer Reset (WDR) mode, and you cannot use the Watchdog Timer interrupt.

Given the preceding discussion, we can pick and choose the Watchdog Timer modes that we wish to configure in our code as follows:

Watchdog Reset (WDR)	`WDE` is 1; `WDIE` is 0. The system will reset if the Watchdog Timer counter is not itself cleared within the timeout period. No interrupts will fire.
Watchdog Interrupt (WDI)	`WDE` is 0, and `WDIE` is 1. The system will set `WDIF` on the timeout occurring, then fire the appropriate ISR if global interrupts are enabled.
Watchdog Reset and Interrupt	`WDE` and `WDIE` are both 1.

In this final mode of operation, if the Watchdog Timer was initialized at time T with timeout period P, then the interrupt will fire at time $T + P$ (assuming global interrupts are enabled of course).

If `WDIE` is again set to 1 after the first P timeout, but before the second P timeout, then the system reset will *not* occur at time $T + P + P$; the Watchdog Interrupt ISR will fire again instead.

If, on the other hand, `WDIE` is 0 prior to the end of the next timeout period, P, then the system will be reset. This will occur at $T + P + P$.

This sequence allows for such things as using the ISR to save any data that must be updated between restarts, etc., shutting down any peripherals, motors, laser cutters, etc., to a safe state before the restart occurs. Until the system has restarted, and fully initialized, the state of various pins is potentially unknown, and a runaway laser cutter, for example, is not a good thing to have close by!

Warning
When the system is reset by the WDT, on restarting, the Watchdog Timer is still enabled; however, it is now enabled at the *smallest possible timeout setting*, 16 milliseconds, and not perhaps as you configured it prior to the reset. This could cause Watchdog Timer reset loops. The Arduino bootloader *should* be checking and disabling Watchdog Timer, but this depends on the bootloader. See Listing 7-2 for details on preventing this possibility.

7.3.5 Enabling the Watchdog Timer

In order that rogue programs don't cause problems by accidentally setting the `WDT`, and to try and ensure that any changes are valid ones, there is a certain timed sequence of events that must be

Table 7-11 Watchdog timeout settings

Timeout	WDP3:0
16 mS	0b0000
32 mS	0b0001
64 mS	0b0010
0.125 S	0b0011
0.25 S	0b0100
0.5 S	0b0101
1.0 S	0b0110
2.0 S	0b0111
4.0 S	0b1000
8.0 S	0b1001

followed in order to configure `WDTCSR` when either `WDE` or the prescaler bits, `WDP3:0`, are being changed:

- Disable global interrupts. This will prevent any existing interrupt handlers from firing during the critical timed sequence, thus preventing a valid change to `WDTCSR` if the ISR in question takes longer than four system clock cycles to execute, which it will!
- Reset the Watchdog Timer. It may already be running, and it should not be allowed to reset the AVR microcontroller while it is being reconfigured, especially when reducing the timeout period.

  If the prescaler is being changed to reduce the timeout, and the new timeout period has already expired since the previous reset of the Watchdog Timer counter, then the Watchdog Timer *will* time out as soon as the configuration completes.
- To start changing `WDTCSR`, you must write a 1 to both `WDCE` and `WDE` *in the same instruction*. If `WDE` is already a 1, you must still write a 1 to it.
- Within four system clock cycles, write the desired configuration bits for the prescaler (see Table 7-11), interrupt enable, etc., to `WDTCSR` and include a 0 in bit `WDCE`. All bits must be set and/or cleared *in the same instruction*.
- Enable global interrupts.

The AVRLib functions defined in `avr/wdt.h` take care of all this when you make calls to the function `wdt_enable()`.

7.3.6 Setting the Watchdog Timer Timeout

The Watchdog Timer can be configured to timeout by setting bits WDP3:0. Table 7-11 shows the various configuration options.

All other `WDP3:0` values from 0b1010 to 0b1111 are reserved and should not be used.

Note
The Watchdog timings in Table 7-11 are only valid for systems running at 5 V. If the voltage is 3.3 V, then the timings will be longer. The data sheet states this, but does not give any relevant information as to how much longer!

Listing 7-3 shows an example Arduino sketch to enable the Watchdog Timer in Watchdog Timer Reset (WDR) mode.

Listing 7-3 Arduino code to enable the Watchdog Timer

```
#include "avr/wdt.h"

void setup() {
  wdt_reset();

  // Fire WDT every 8 seconds.
  wdt_enable(WDTO_8S);
}

void loop() {
  // Make sure we reset the WDT.
  wdt_reset();

  // Do our loopy stuff here. It must
  // complete in less time than the
  // WDT timeout period.
  ...
}
```

Unfortunately, the `wdt_enable(WDTO_8S)` instruction in Listing 7-3 only sets the `WDE` and prescaler bits, it does not set the interrupt enable bit, `WDIE`, for you.

At present, the AVRLib doesn't have the ability to set the Watchdog Timer interrupt, so if Watchdog Timer interrupts are required, you need to configure everything manually.

Listing 7-4 is a function which will enable WDI mode, without also enabling the WDR mode. Your project can then use the Watchdog Timer Interrupt handler to carry out some work periodically and not have to worry about calling `wdt_reset()` within the timeout period.

Listing 7-4 Arduino code to enable the Watchdog Timer interrupt

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

void wdt_interrupts(uint8_t value) {
  // Save existing interrupt state.
  uint8_t oldSREG = SREG;                              (1)

  // Set the WDP3-WDP0 bits for the prescaler.
  uint8_t wdt_setting;
  value = (value > 9) ? 9 : value;                     (2)
  wdt_setting = (value > 7) ? (1 << WDP3) : 0;
  wdt_setting |= (value & 7);

  // Disable interrupts and reset WDT.
  noInterrupts();                                      (3)
  wdt_reset();
  // Clear WDT restarted flag.
  MCUSR &= ~(1 << WDRF);                                     (4)

  // Do the timed sequence next.
  #if defined WDTCSR
  // ATmega168/328/2560 etc
  WDTCSR |= ((1 << WDCE) | (1 << WDE));                      (5)
  WDTCSR = (wdt_setting | (1 << WDIE));

  #elif defined WDTCR
    // ATtiny25/45/85 etc
    WDTCR |= ((1 << WDCE) | (1 << WDE));                     (5)
    WDTCR = (wdt_setting | (1 << WDIE));
  #else
    #error "Unknown WDT Control Register on your AVR."
  #endif

  // Put interrupts back as they were previously.
  SREG = oldSREG;                                            (6)
}
```

(1) Saving the status register preserves the state of global interrupts.
(2) There are four bits available here, so up to 16 values; however, we can only have nine as the maximum value. If the value is eight or nine, then we need to set `WDP3`.
(3) Interrupts and the Watchdog Timer must be disabled if we are going to change the Watchdog Timer settings. This will prevent rogue resets part way through changing the settings.
(4) The data sheet says we must clear `WDRF`.
(5) This is the timed sequence of instructions that we must complete in order that the changes to the Watchdog Timer will be considered valid. Because I use ATtiny devices, I have coded for those as well.
(6) Restore the global interrupt bit in the status register to how it was on entry to this function.

That takes care of enabling the Watchdog Timer interrupt. To disable it, use the function in Listing 7-5.

Listing 7-5 Arduino code to disable the Watchdog Timer interrupt

```
#include "avr/wdt.h"

void wdt_noInterrupts() {
  // Disable WDT interrupts leaving
  // everything else untouched.
  #if defined WDTCSR
    // ATmega328 etc
    WDTCSR &= ~(1 << WDIE);                                  (1)

  #elif defined WDTCR
```

```
    // ATtiny85 etc
    WDTCR &= ~(1 << WDIE);                                  (1)

  #else
    #error "Unknown WDT Control Register on your AVR."
  #endif
}
```

(1) This clears the `WDIE` bit and leaves the other bits unaffected. The `WDE` and `WDP3:0` bits cannot be changed except under the terms and conditions of the previously mentioned timed sequence. The code here doesn't violate those rules. As in Listing 7-4, this code also attempts to determine between the ATmega328P and ATtiny85 microcontrollers, both of which I use.

The code in Listings 7-4 and 7-5 could now be used to program yet another Blink sketch. Listing 7-6 shows how a regular blink could be applied to an LED using nothing but the Watchdog Timer Interrupt.

Listing 7-6 Using the Watchdog Timer interrupt in the Blink sketch

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  wdt_interrupts(WDTO_1S);
}

void loop() {
  ; //Do nothing.
}

ISR(WDT_vect) {
  PINB |= (1 << PINB5);
}
```

The `loop()` function does nothing at all. All the blinking takes place in the ISR for the Watchdog Timer interrupt. In code like this, where an ISR is doing all the work, the main loop should really put the AVR microcontroller to sleep. And this is something we will deal with in Section 7.4.

7.3.7 Disabling the Watchdog Timer

The Watchdog Timer can be disabled by performing the following sequence of events.

Warning
If the `WDTON` fuse has been programmed (i.e., has value 0), then you will be unable to disable the Watchdog Timer. In this case, the Watchdog Timer is always running in Watchdog Reset (WDR) mode.

- Disable global interrupts. This will prevent any existing interrupt handlers from firing during the critical timed sequence, thus preventing a valid change to WDTCSR.
- Reset the Watchdog Timer. It may already be running, and it should not reset the AVR microcontroller while it is being reconfigured.
- Ensure that bit WDRF in MCUSR is cleared.
- To start changing WDTCSR, you must write a 1 to both WDCE and WDE *in the same instruction*. If WDE is already a 1, you must still write a 1 to it.

  You are advised to preserve the existing state of bits WDP3:0 to prevent unintentional Watchdog Timer timeouts which *will* occur if the timeout period is being reduced, explicitly or implicitly, and the new timeout has already expired.
- Within four system clock cycles, write a 0 to bits WDCE and WDE. All bits must be set and/or cleared in the same instruction. The data sheet advises clearing the entire register, which completely disagrees with its previous instruction to *preserve the state of bits* WDP3:0.
- Enable global interrupts.

The AVRLib functions defined in avr/wdt.h take care of all this when you make calls to the function wdt_disable().

Listing 7-7 is an example Arduino sketch to completely disable the Watchdog Timer.

Listing 7-7 Arduino code to disable the Watchdog Timer

```
#include "avr/wdt.h"

void setup() {
  wdt_reset();
  MCUSR &= ~(1 << WDRF);                                    (1)
  wdt_disable();                                            (2)
}
```

(1) This clears the flag in the MCU Status Register which indicates that the Watchdog Timer reset the AVR microcontroller.
(2) The function wdt_disable() is defined in the avr/wdt.h header file and carries out all the necessary instructions to disable the Watchdog Timer, including disabling interrupts and enabling them afterward, if appropriate.

You could, obviously, extract the preceding code to a function of your own and call that whenever the Watchdog Timer needed to be disabled.

7.4 Putting the AVR to Sleep

The ATmega328P comes with six different sleep modes built in. These can be used to vastly reduce power requirements of the AVR microcontroller when it doesn't need to be polling for button presses, etc. When sleeping, interrupts must normally be used to wake the device, and let it run the necessary processing as required.

In other words, if the code is written in such a way as to not require the processor to be constantly polling sensors, etc., in the main loop, then it can be put to sleep which will save power and make batteries last much, much longer.

Chapter 9 of the data sheet, "Sleep Modes," has the following points of note:

> When entering a sleep mode, all port pins should be configured to use minimum power. The most important is then to ensure that no pins drive resistive loads. In sleep modes where both the I/O clock (CLK_{io}) and the ADC clock (CLK_{adc}) are stopped, the input buffers of the device will be disabled. This ensures that no power is consumed by the input logic when not needed. In some cases, the input logic is needed for detecting wake-up conditions, and it will then be enabled.
>
> If the input buffer is enabled and the input signal is left floating or have an analog signal level close to $VCC/2$, the input buffer will use excessive power.
>
> For analog input pins, the digital input buffer should be disabled at all times. An analog signal level close to $VCC/2$ on an input pin can cause significant current even in active mode. Digital input buffers can be disabled by writing to the Digital Input Disable Registers (`DIDR1` and `DIDR0`).

The "analog input pins" referred to are those of the ADC and the Analog Comparator. Sections 9.1 and 9.2 describe how to disable the digital input buffers when using either the Analog Comparator or the ADC.

The following sleep modes are normally available:

- Idle, mode 0
- ADC Noise Reduction, mode 1
- Power Down, mode 2
- Power Save, mode 3
- Standby, mode 6 (only when there is an external crystal or ceramic resonator)
- Extended Standby, mode 7

To put the AVR microcontroller to sleep, the code must

- Set bits `SM2:0` in `SMCR` (Sleep Mode Control Register) to select the sleep mode required
- Set bit `SE` (Sleep Enable) in `SMCR` to enable the desired sleep mode
- Execute the `sleep` (assembly language) instruction

The values defined for the various modes are as follows and can be found in the header file `\$ARDINC/avr/iom328p.h`:

```
#define SLEEP_MODE_IDLE (0x00 << 1)
#define SLEEP_MODE_ADC (0x01 << 1)
#define SLEEP_MODE_PWR_DOWN (0x02 << 1)
#define SLEEP_MODE_PWR_SAVE (0x03 << 1)
#define SLEEP_MODE_STANDBY (0x06 << 1)
#define SLEEP_MODE_EXT_STANDBY (0x07 << 1)
```

Note
The bits `SM2:0` in the `SMCR` register are bits 3:1, hence the use of the shift left instructions in the preceding definitions.

To put an AVR microcontroller to sleep in idle mode, for example, you could execute code similar to that shown in Listings 7-8 to 7-10. You could, but I wouldn't bother myself because it doesn't work as expected with the code shown! Don't worry as all will be made clear soon.

Listing 7-8 is a simple function to flash the built-in LED a few times. It's called from `setup()` to show that we are indeed alive and from the `loop()` when it's doing "real" work.

Listing 7-8 Nonfunctioning sleep sketch, flashLED()

```
#include <avr/sleep.h>
#include <avr/interrupt.h>

void flashLED(byte flashes) {
  for (byte x = 0; x < flashes; x++) {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(250);
    digitalWrite(LED_BUILTIN, LOW);
    delay(250);
  }
}
```

Listing 7-9 is the `setup()` function which sets the requirements for idle sleep mode, then flashes the LED a couple of times to show we are alive and well, so far. Setting the sleep mode doesn't put the Arduino to sleep at that point; that comes later.

Listing 7-9 Nonfunctioning sleep sketch, setup()

```
void setup() {
  set_sleep_mode(SLEEP_MODE_IDLE);
  pinMode(LED_BUILTIN, OUTPUT);

  // Show we are alive
  delay(1500);
  flashLED(2);
}
```

In Listing 7-10, we can see the `loop()` function. This is where the code would normally be doing application work, then sleeping until some stimulus wakes it up for the next pass through the loop.

Listing 7-10 Nonfunctioning sleep sketch, loop()

```
void loop() {
  noInterrupts();                                        (1)

  // Enable sleep mode and disable Brown Out Detection.
  // BOD disable is permitted on ATmega328P.
  sleep_enable();                                        (2)
  sleep_bod_disable();

  // Enable interrupts and execute the sleep_cpu() -
  // Which guarantees that the sleep_cpu() will execute
  // before any new interrupts will be fired.
  interrupts();                                          (3)
  sleep_cpu();                                        (4)

  // When we wake up, on an interrupt, disable
  // sleep mode while processing.
  sleep_disable();                                    (5)

  // Do some application "stuff" here after waking up ...
  flashLED(4);                                        (6)

  // On the next pass of the loop, we will go back to sleep.
}
```

(1) It is recommended to disable interrupts when changing sleep modes.
(2) In `setup()`, we configured idle sleep mode. This function call enables sleep modes, but still doesn't put the device to sleep. Also here, we turn off the brown out detector (BOD) while sleeping.
(3) Sleep modes need an interrupt to wake the device. We need interrupts turning on, or we will simply sleep forever.
(4) Finally, we put the device to sleep. The device will sleep in idle mode until woken.
(5) This is where the application does some work after it wakes up from sleep. It should disable sleeping until ready to go back to sleep.
(6) This is the "real" work that the application has to do!

If, while the device is asleep, an interrupt occurs, the device will wake up but will then halt itself for four clock cycles before executing the interrupt routine. Those four clock cycles are in addition to any startup time requirements.

Once the interrupt routine has been executed, control returns to the instruction after the sleep instruction that put the device to sleep previously. The sleep instruction is built in to the ATmega328P. Our C++ code can call it as it has been defined elsewhere. Normally, it is easier to use the various sleep functions in AVRLib.

So, what's wrong with the `setup()` in Listing 7-9? Why is `SLEEP_MODE_IDLE` not the best sleep mode for this demonstration?

You will possibly have realized that while an Arduino is in this sleep mode, that Timer 0 which is used to count `millis()` and so on is running with its overflow interrupt enabled. As an interrupt wakes the AVR microcontroller when it fires, then in `SLEEP_MODE_IDLE` the main loop appears to just run and run. It does actually sleep, but only until the timer overflows and that happens every 1,024 microseconds.

If you were to recompile the code, but use `SLEEP_MODE_PWR_DOWN` instead, once the LED flashes twice in `setup()`, you won't see it again until a proper interrupt occurs; however, as I haven't enabled any particular interrupts, the Arduino will simply go to sleep and never wake up. There's a much better example in Section 7.4.2.

7.4.1 Sleep Modes

Table 7-12 shows a list of the various sleep modes available and whether they can be used on an Arduino.

Table 7-12 Sleep modes

Sleep Mode	Usable on Arduino	Comments
SLEEP_MODE_IDLE	Maybe	Only usable if Timer 0 has the overflow interrupt disabled
SLEEP_MODE_ADC	Yes	Could be used before an analogRead() call
SLEEP_MODE_PWR_DOWN	Yes	
SLEEP_MODE_PWR_SAVE	No	On Arduinos, this is effectively identical to SLEEP_MODE_PWR_DOWN which should be used instead
SLEEP_MODE_STANDBY	Yes	This is just SLEEP_MODE_PWR_DOWN but with the main oscillator running
SLEEP_MODE_EXT_STANDBY	No	This is just SLEEP_MODE_PWR_SAVE but with the main oscillator running

A summary of the individual sleep modes is given in Table 7-13, which shows the clocks, oscillators, and peripherals which are running and the various wake-up stimuli that each mode will react to. You should note that where the Timer 2 asynchronous mode of operation is mentioned, it cannot be used on an Arduino board; however, other boards fitted with ATmega328P microcontrollers may use those modes.

In the discussions to follow, regarding each different sleep mode, I give the sleep modes numbers. Idle, for example, is sleep mode zero. I indicate the value in binary as well, in three bits. These are the three bits that need to be written to the `SM2:0` bits in the Sleep Mode Control Register (`SMCR`). These three bits control which sleep mode the AVR microcontroller will enter when the Sleep Enable, `SE`, bit is set and a `sleep` instruction executed.

While there are three bits available, giving eight possible sleep modes (0b000 to 0b111), sleep modes 4 (0b100) and 5 (0b101) are reserved and should not be used, leaving only six modes. Of these six, one cannot be used on the Arduino as it requires Timer 2 to be running in asynchronous mode, which is not possible on an Arduino.

To put the AVR microcontroller to sleep, the steps are as follows:

- Set the sleep mode required in bits `SM2:0` of the `SMCR` register. You can do this in a sketch by calling `set_sleep_mode()` or directly in your own code. The examples in this section use the easy method—calling `set_sleep_mode()` in AVRLib.
- Set the Sleep Enable bit, `SE`, to 1 in `SMCR` by calling `sleep_enable()` as part of the `loop()`.
- Execute the `sleep` (assembly language) instruction by calling `sleep_cpu()` in the `loop()`.
- On wake-up, the data sheet advises that you immediately clear the `SE` bit to 0; you can do this simply by calling `sleep_disable()` before starting the code that is to be carried out upon waking from sleep.

The ATmega328P will be woken from sleep modes if an enabled interrupt occurs or if it is reset while asleep. The wake-up process is as follows, for an interrupt:

- The interrupt fires and the AVR microcontroller wakes up from sleep.
- The device is then halted for four clock cycles (over and above the device wake-up time).
- The interrupt service routine (ISR) is executed.
- Execution then continues from the instruction immediately following the `sleep` (or `sleep_cpu()`) instruction.

Table 7-13 Sleep mode summary

Sleep Mode Summary		0	1	2	3	6	7
Clocks	CLK_{cpu}	–	–	–	–	–	–
	CLK_{flash}	–	–	–	–	–	–
	CLK_{io}	Y	–	–	–	–	–
	CLK_{adc}	Y	Y	–	–	–	–
	CLK_{asy}	X	X	–	X	–	X
Oscillators	Main System	Y	Y	–	–	Y	Y
	Asynchronous Timer 2	X	X	–	X	–	X
Peripherals	ADC	Y	Y	–	–	–	–
	Analog Comparator	Y	–	–	–	–	–
	Core (CPU)	–	–	–	–	–	–
	Static RAM	–	–	–	–	–	–
	EEPROM	–	–	–	–	–	–
	External interrupts	Y	–	Y	Y	Y	Y
	Flash RAM	–	–	–	–	–	–
	SPI	Y	–	–	–	–	–
	Timers	Y	–	–	–	–	–
	TWI/I2C	Y	–	Y	Y	Y	Y
	USART	Y	–	–	–	–	–
	Watchdog	Y	Y	Y	Y	Y	Y
Wake On	External reset	Y	Y	Y	Y	Y	Y
	INT0 Interrupt	Y	L	L	L	L	L
	INT1 Interrupt	Y	L	L	L	L	L
	Pin change interrupt	Y	Y	Y	Y	Y	Y
	TWI/I2C address match	Y	Y	Y	Y	Y	Y
	Asynchronous timer interrupt	X	X	–	X	–	X
	SPM/EEPROM ready interrupt	Y	Y	–	–	–	–
	ADC conversion complete interrupt	Y	Y	–	–	–	–
	Watchdog timer interrupt	Y	Y	Y	Y	Y	Y
	Timer, USART interrupts	Y	–	–	–	–	–
	Analog Comparator interrupt	Y	–	–	–	–	–
	Brown out reset	Y	Y	Y	Y	Y	Y

^{Y} Clock, oscillator, and peripherals are running. Will wake on indicated stimulus
^{X} Clock and oscillator are running on ATmega328P, but not on Arduino boards. Will wake non-Arduino boards on indicated stimulus, but it is not possible for an Arduino to be wakened
^{L} Wakes on `LOW`-level `INT0/INT1` Interrupt only
$^{–}$ Clocks or oscillators are not running. Peripherals cannot be used. Will not wake on indicated stimuli

7.4.1.1 Features of All Sleep Modes

Table 7-13 shows a summary of each of the available sleep modes for Arduino boards, as well as for other systems which run the ATmega328 family of microcontrollers. We can see that there are some commonality between all sleep modes. These are described in this section, while those other features, specific to the individual sleep modes, are discussed in the appropriate sections to follow.

Clocks	In all sleep modes, the CLK_{cpu} and CLK_{flash} clocks are stopped.

Peripherals In all sleep modes, the Watchdog will be available if enabled, while the following will be unavailable:

- The core processor—the CPU in other words
- Static RAM
- Flash RAM
- EEPROM

Wake-up stimuli Regardless of the sleep mode, the board will be woken by

- An external reset—when the reset button is pressed
- A pin change interrupt
- A TWI/I^{2}C address match interrupt
- A Watchdog Interrupt
- An `INT0/INT1` Interrupt for sleep mode zero only
- A `LOW`-level `INT0/INT1` Interrupt for all other modes
- A brown out reset

During sleep, if the BOD is disabled, it cannot monitor VCC while asleep. On wake-up, the BOD is re-enabled, and if VCC has dropped during sleep, the board will be held in brown out mode until such time as VCC reaches the configured voltage level. Once VCC is stable again, the brown out state will be released and the board will reset.

Timer 2's asynchronous clock is able to run for certain sleep modes; however, this is not possible on an Arduino board, so is considered unusable in all sleep modes for those boards. For other boards using the ATmega328P microcontroller, it is potentially possible to enable this clock, CLK$_{asy}$, and use the available sleep and wake-up features only on those non-Arduino boards. This is not possible on Arduino boards as the asynchronous clock requires an external 32.768 KHz oscillator connected to the two pins that the 16 MHz crystal is currently attached to.

Warning
In the discussions to follow, it will be assumed that CLK$_{asy}$ is not running as we are dealing with Arduino boards. In addition, the Timer 2 interrupts will not be able to wake the board, again, as we are discussing Arduino boards. Bear in mind, however, that other ATmega328P boards *may* be able to make use of those features.

7.4.1.2 Idle Sleep Mode

This is sleep mode zero (0b000) and is the lightest sleep mode; it saves power but not really a lot as only the CLK$_{cpu}$ and CLK$_{flash}$ are stopped.

This sleep mode is *almost* useless on an Arduino board as the overflow interrupt for Timer 0 triggers every 1,024 microseconds, and this interrupt wakes the board from any sleep mode. However, it does sleep and, while sleeping, reduces power consumption.

Clocks In idle sleep mode, all clocks, apart from those mentioned in Section 7.4.1.1, are running.

Oscillators The main system oscillator is running in this sleep mode.

Peripherals All peripherals are running and available apart from those mentioned in Section 7.4.1.1.

Wake-up stimuli	In addition to the wake-up stimuli mentioned in Section 7.4.1, the following will also wake the board from this sleep mode:

- SPM/EEPROM ready interrupt.
- ADC conversion complete interrupt.
- Any interrupt described in the data sheet as "other." This means any timer and/or USART interrupt.
- The Analog Comparator interrupt.

As demonstrated in Listings 7-8 through 7-10, this sleep mode is of very limited use on an Arduino as the overflow interrupt for Timer 0 is active when sketches are compiled. You would have to disable the Timer 0 Overflow Interrupt in your sketch to be able to use this sleep mode.

The Analog Comparator is able to be used in this sleep mode, and this is the only sleep mode where the Analog Comparator can be used to wake the board. If wake-up from the Analog Comparator interrupt is not required, the Analog Comparator should be powered down by setting the `ACD` bit in the Analog Comparator Control and Status Register, `ACSR`.

If the ADC is enabled, an ADC conversion will be started automatically when the idle sleep mode begins after the `sleep` instruction's execution.

7.4.1.3 ADC Noise Reduction Sleep Mode

This is sleep mode 1 (0b001) and is used when there is a need to reduce any noise coming from the AVR microcontroller itself, so that the ADC can take better analog readings. This sleep mode is usable on Arduinos.

Clocks	In ADC Noise Reduction sleep mode, all clocks, in addition to those mentioned in Section 7.4.1.1, are stopped apart from CLK_{adc}, which is still running.
Oscillators	The main system oscillator is running in this sleep mode.
Peripherals	All peripherals are unavailable apart from the ADC and the Watchdog.
Wake-up stimuli	In addition to the wake-up stimuli mentioned in Section 7.4.1, the following will also wake the board from this sleep mode:

- SPM/EEPROM ready interrupt
- ADC conversion complete interrupt

This sleep mode improves the noise environment for the ADC, enabling higher resolution measurements. If the ADC is enabled, a conversion will be started automatically when this sleep mode is entered.

7.4.1.4 Power Down Sleep Mode

This is sleep mode 2 (0b010); this is a deep sleep and saves the most power. Consequently, it has fewer wake-up stimuli. This sleep mode is usable on Arduinos.

Clocks	In Power Down sleep mode, all clocks are stopped.
Oscillators	The main system oscillator is stopped in this sleep mode.
Peripherals	Only the peripherals mentioned in Section 7.4.1 are available.
Wake-up stimuli	Only the wake-up stimuli mentioned in Section 7.4.1 will wake the board from this sleep mode.

7.4.1.5 Power Save Sleep Mode

This is sleep mode 3 (0b011); this is also a deep sleep and saves much power. However, it is slightly less a deep sleep than Power Down sleep mode as the asynchronous clock, CLK_{asy}, and the asynchronous oscillator are running. However, this is not possible on Arduino boards, so sleep mode 2, Power Down, should be used instead.

This sleep mode is usable on ATmega328P boards, but not on Arduino boards as the asynchronous mode for Timer 2 is not usable.

7.4.1.6 Standby Sleep Mode

This is sleep mode 6 (0b110). The data sheet advises not to use this sleep mode unless there is an external crystal running the main clock. This is appropriate for Arduino boards due to the 16 MHz crystal which is used to run the main system oscillator.

This mode is identical to the Power Down mode described earlier, apart from the main oscillator running. This sleep mode is usable on Arduinos.

Clocks	In Standby sleep mode, all clocks are stopped.
Oscillators	The main system oscillator is running in this sleep mode.
Peripherals	Only the peripherals mentioned in Section 7.4.1 are available.
Wake-up stimuli	Only the wake-up stimuli mentioned in Section 7.4.1 will wake the board from this sleep mode.

On a wake-up call, when sleeping in this mode, the device is back up and running in six clock cycles.

7.4.1.7 Extended Standby Sleep Mode

This is sleep mode 7 (0b111). The data sheet advises not to use this sleep mode unless there is an external crystal running the main clock. This mode is identical to the Power Save mode described earlier, except the main system oscillator and the asynchronous oscillator, if available, are running.

This sleep mode is best avoided on Arduinos as the asynchronous timer clock isn't running. Use Standby sleep mode instead.

Clocks	In Standby sleep mode, all clocks are stopped.
Oscillators	The main system oscillator is running in this sleep mode.
Peripherals	Only the peripherals mentioned in Section 7.4.1 are available.
Wake-up stimuli	Only the wake-up stimuli mentioned in Section 7.4.1, plus any Timer 2 async interrupts, will wake the board from this sleep mode.

On a wake-up call, when sleeping in this mode, the device is back up and running in six clock cycles.

7.4.2 Sleep and the Analog Comparator

The data sheet is not very clear on whether or not the Analog Comparator interrupt can be used to wake the device from some of the sleep modes. Believe me, I searched in vain for the detail! To this end, I created a sketch and configured each of the available and usable sleep modes to test if the Analog Comparator could wake the Arduino board. The circuit I used is exactly the same to the one described in Chapter 9 on the Analog Comparator, where there is a full description of the circuit and how it works.

In the code in Listings 7-11 to 7-14, I set the Arduino into various sleep modes in `setup()` and then tested to see if the comparator would wake the AVR microcontroller from its slumbers.

Listing 7-11 is the `setupComparator()` function which has not changed from Chapter 9, other than to `#include` the `avr/sleep.h` and `avr/interrupt.h` header files at the top of the function.

Listing 7-11 Analog Comparator wake-up, setupComparator()

```
//=======================================================
// The purpose of the sketch is to test the various sleep
// modes and to see if the AC will wake the Arduino.
//=======================================================

#include <avr/sleep.h>
#include <avr/interrupt.h>

// This function sets up the comparator to fire an interrupt
// each time the ACO bit toggles. It uses D6 as the reference
// voltage and D7 as the voltage to be compared.
void setupComparator() {
  ACSR &= ~((1 << ACIE)|(1 << ACD));
  DIDR1 |= ((1 << AIN0D) | (1 << AIN1D));
  ACSR &= ~(1 << ACBG);
  ADCSRB &= ~(1 << ACME);
  ACSR |= ((1 << ACIS1) | (1 << ACIS0)|(1 << ACIE));
}
```

Listing 7-12 is pretty much the same `setup()` function to that which will be discussed in detail later, in Chapter 9, with the minor addition of the call to function `set_sleep_mode()`. It is here that I tested each and every valid sleep mode to see which, if any, would be interrupted by the Analog Comparator's interrupt.

Listing 7-12 Analog Comparator wake-up, setup()

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  setupComparator();

  // Here is where I set the various sleep modes.
  set_sleep_mode(SLEEP_MODE_IDLE);
}
```

An interrupt is required to wake the ATmega328P, but in this instance, we don't actually need to do any work inside the ISR, just wake up the microcontroller. Listing 7-13 shows the code required for just such a purpose; it is an empty ISR to handle the Analog Comparator interrupt.

Listing 7-13 Analog Comparator wake-up, empty ISR

```
// Analog Comparator Interrupt Handler. Simply used to
// wake up the device, so no code required.
EMPTY_INTERRUPT(ANALOG_COMP_vect);
```

Listing 7-14 is the main `loop()` function for the sketch. It puts the device to sleep in whatever mode I used in `setup()` and waits for a wake-up call. If one arrives, it will flash the built-in LED to show that it woke up. It will then go back to sleep.

Listing 7-14 Analog Comparator wake-up, loop()

```
void loop() {
  noInterrupts();
  sleep_enable();
  sleep_bod_disable();

  // Kill timer 0 and its overflow interrupt otherwise
  // it will wake the AVR from SLEEP_MODE_IDLE thus
  // negating the test. I need the AC to do the wake
  // up call!
  TCCR0B &= ~((1 << CS02) | (1 << CS01) | (1 << CS00));

  interrupts();

  // Go to sleep now.
  sleep_cpu();

  // Interrupt occurred, I'm awake.
  sleep_disable();

  // Reset Timer 0 to divide by 64 or delay() doesn't!
  TCCR0B |= ((1 << CS01) | (1 << CS00));

  // Flash the LED on wake up.
  for (short x = 0; x < 4; x++) {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(250);
    digitalWrite(LED_BUILTIN, LOW);
    delay(250);
  }
}
```

And the results? The Analog Comparator does indeed wake up the Arduino when in `SLEEP_MODE_IDLE` but doesn't wake it up in any other mode which is as I thought would be the case, but at least I now know for certain.

7.5 Power Reduction

Many AVR microcontrollers come with numerous different, potentially power-hungry, peripherals. If these are not being used by a sketch, then they can be disabled by setting a bit in the Power Reduction Register, `PRR`. This will shut them down and reduce overall power consumption by the AVR microcontroller, thus increasing battery life on battery-powered devices.

Numerous parts of the AVR microcontroller can be disconnected or shut down to save power, and it is recommended that this be done if those peripherals are not in use. The Arduino system cannot determine which parts you are not using, so it is up to you, the maker, to decide and disable accordingly.

Power consumption and the various sleep modes discussed in Section 7.4.1 are a great way to reduce the power requirements of your project and can help, in some cases, to dramatically increase running time from battery power.

7.5.1 Power Consumption

The ATmega328P can be run at a number of supply voltages. On Arduino boards with a 16 MHz crystal or ceramic resonator, the supply voltage is limited to between 4.0 V and 5.5 V. The data sheet gives the figures shown in Table 7-14 for power consumption for each of those voltages, at 16 MHz, when the device is idle (in sleep mode idle) and active or not sleeping.

Note

You should note that these figures are for a bare-bones AVR microcontroller, and *not* for the whole Arduino board, complete with power-hungry devices like the voltage regulator, the always-on power LED, and so on.

You may be surprised to find out exactly how few components you actually need to run a device with an ATmega328P as the microcontroller and even fewer if your device can be run using an ATtiny85!

From Table 7-14, it can be seen that an active ATmega328P, running at 16 MHz with a VCC of 4 V, uses half the power of the same device running with a VCC of 5.5 V. Sadly, our Arduino boards are fixed at 5 V for VCC, so we are not able to do much in that area to reduce power consumption. However, for a bare-bones AVR microcontroller on a breadboard or circuit board of our own design, we do have the option.

Note

While the Arduino boards themselves are excellent for prototyping, they are a tad expensive, and power-hungry, to embed in the finished product. In the case where a device is deemed to be market ready, the Arduino itself will normally be replaced by a minimal ATmega328P circuit, with far fewer resource requirements and a much lower cost.

Table 7-14 ATmega328P power consumption

VCC	Idle Current	Idle Power	Active Current	Active Power
4.0 V	1.75 mA	7.0 mW	7.0 mA	28 mW
4.5 V	2.1 mA	9.45 mW	8.2 mA	36.9 mW
5.0 V	2.4 mA	12.0 mW	9.6 mA	48.0 mW
5.5 V	2.8 mA	15.4 mW	11.0 mA	60.5 mW

Table 7-15 ATmega328P peripheral power consumption

Peripheral	Typical Current	Active Extra	Idle Extra
ADC	295.28 uA	4.1%	22.1%
USART	100.25 uA	1.4%	7.8%
SPI	186.5 uA	2.9%	15.7%
Timer 0	61.13 uA	0.9%	4.8%
Timer 1	176.25 uA	2.7%	14.5%
Timer 2	224.25 uA	3.3%	17.8%
TWI	199.25 uA	3.0%	16.6%

Table 7-15 shows the power consumption of the various ATmega328P internal peripherals and is taken from the data sheet. It should be noted that the typical figures listed are those for an AVR microcontroller running with a VCC of 5 V and a clock frequency of 8 MHz. This will not be accurate for an Arduino at 16 MHz, but read on, as all will become clear.

7.5.1.1 Calculating Power Requirements

As the data sheet explains, it is possible to calculate the power requirements for each of the seven peripherals, listed in Table 7-15, based on the voltage and frequency of the crystal or ceramic oscillator in use. The preceding typical figures are based on a 5 V AVR microcontroller running at 8 MHz; the Arduino runs at 16 MHz, so we need to get the calculator out.

It's actually quite simple; take the current drawn from Table 7-14 for the voltage your device is using, then add on the percentage for active or idle from Table 7-15 for the peripheral in question. If there are more than one peripheral enabled, simply add each percentage.

If we consider the AVR microcontroller running on a 5 V supply at 16 MHz, with all peripherals enabled and running, what is the idle power required?

$$\begin{aligned}
\text{Idle power at 5 V} &= 2.4 \text{ mA} \\
\text{Peripheral percentages} &= 22.1 + 7.8 + 15.7 + 4.8 + 14.5 + 17.8 + 16.6 \\
&= 99.3\% \\
99.3\% \text{ of } 2.4 \text{ mA} &= 2.3832 \text{ mA} \\
\text{Added to } 2.4 \text{ mA} &= 4.7832 \text{ mA} \\
\text{Resulting power consumption} &= 4.7832 \text{ mA} * 5\text{V} \\
&= 22.916 \text{ mW}
\end{aligned}$$

The same calculation in active mode results in

$$\begin{aligned}
\text{Idle power at 5 V} &= 9.6 \text{ mA} \\
\text{Peripheral percentages} &= 4.1 + 1.4 + 2.9 + 0.9 + 2.7 + 3.3 + 3.0 \\
&= 18.3\% \\
18.3\% \text{ of } 9.6 \text{ mA} &= 1.7568 \text{ mA} \\
\text{Added to } 9.6 \text{ mA} &= 11.3568 \text{ mA}
\end{aligned}$$

$$\begin{aligned}\text{Resulting power consumption} &= 11.3568\text{ mA} * 5\text{V}\\ &= 56.784\text{ mW}\end{aligned}$$

It should be obvious that these calculations include all of the available peripherals. If you only need to use one or two peripherals, simply add the percentages for those you have enabled when carrying out the calculations.

7.5.2 Power Reduction Register

Not all AVR microcontrollers have the same set of peripherals, and the relevant ones in the ATmega328P are as follows:

- Analog-to-Digital Converter (ADC)
- Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART)
- Two Wire Interface (TWI), a.k.a. I²C Interface
- Timer/counter 0
- Timer/counter 1
- Timer/counter 2
- Serial Peripheral Interface (SPI)

Those seven peripherals each have a single bit in the Power Reduction Register or PRR so that when set to a 1, that particular peripheral is powered off. The bits in question are

`PRADC`	Enables or disables power to the ADC
`PRUSART0`	Enables or disables power to the USART
`PRSPI`	Enables or disables power to the SPI
`PRTIM0`	Enables or disables power to Timer 0
`PRTIM1`	Enables or disables power to Timer 1
`PRTIM2`	Enables or disables power to Timer 2
`PRTWI`	Enables or disables power to the TWI

What these bits do is to stop the clock to the peripheral. Once the clock has been stopped, that peripheral is effectively suspended, and there is no ability to write to, or read from, the device's registers. The data sheet warns that

> Resources used by the peripheral when stopping the clock will remain occupied, hence the peripheral should in most cases be disabled before stopping the clock.

What this means is that whatever peripheral you wish to power off should be disabled before powering it down. In the case of the ADC, for example, this would entail writing a zero to `ADEN` in the `ADCSRA` register to disable the ADC, then writing a one to `PRADC` in the `PRR` to power it off.

The on-off switches for each preceding peripheral are listed in Table 7-16 and, unless otherwise noted, should be written with a zero to disable the appropriate peripheral.

To power up a peripheral from its powered down state, simply write a zero to the appropriate bit in the `PRR`. The peripheral will wake up again and will resume the state that it was in when powered down. You have to write a zero because that disables the power reduction for that peripheral.

Table 7-16 Disabling ATmega328P peripherals

Peripheral	Bit(s)	Register	Comments
ADC	ADEN	ADCSRA	
USART	RXEN0	UCSR0B	Shuts down the USART receiver only
USART	TXEN0	UCSR0B	Shuts down the USART transmitter only
SPI	SPE	SPCR	
Timer 0	CS02:00	TCCR0B	Set to 0b000 to disable Timer 0
Timer 1	CS12:10	TCCR1B	Set to 0b000 to disable Timer 1
Timer 2	CS22:20	TCCR2B	Set to 0b000 to disable Timer 2
TWI	TWEN	TWCR	
Analog Comparator	ACD	ACSR	

Note
The Analog Comparator doesn't have a bit in the PRR. It does, however, have the ACD bit in the Analog Comparator Control and Status Register, ACSR. Write a one to that bit to power off the Analog Comparator.

7.5.3 Saving Arduino Power

Now you know what peripherals can be disabled and powered down, you are able to perhaps save a little of your board's power by using the `setup()` function to power off all those bits of the ATmega328P that you don't need for a sketch.

Taking the old favorite blink sketch, yet again, what does it actually need? Nothing more than the I/O pins and a timer to work the `delay()` function. The `delay()` function as well as `millis()` and `micros()` all depend on Timer 0. Listings 7-15 to 7-17 show an example blink sketch with unwanted peripherals turned off.

The AVRLib library has some useful power maintenance functions, and these can be used to power off the peripherals we don't need in our blink sketch. There is a trade-off of course; adding code to `setup()` to disable and power off these peripherals will increase the code size of the final sketch.

In the case of the blink sketch, it's unlikely that this will be a problem, but for other sketches that might be pushing at the capacity of the AVR microcontroller, it could be a problem and you might have to revert to the direct manner of setting the registers in your code, rather than using the AVRLib library functions.

The sketch in Listings 7-15, 7-16, and 7-17 could be used to enhance battery life for an Arduino device, running the blink sketch. Listing 7-15 is a function, `disable()`, which disables all the unwanted peripherals based on parameters passed to it, prior to powering them all off.

Listing 7-15 Low-power blink, disable() function

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

void disable(bool ADCdisable,    bool USARTdisable,
             bool SPIdisable,    bool TIMER0disable,
             bool TIMER1disable, bool TIMER2disable,
             bool TWIdisable,    bool ACdisable)
{
  // Disable ADC.
  if (ADCdisable)
    ADCSRA &= ~(1 << ADEN);

  // Disable USART0 RX and TX.
  if (USARTdisable)
    UCSR0B &= ~((1 << RXEN0) | (1 << TXEN0));

  // Disable SPI.
  if (SPIdisable)
    SPCR &= ~(1 << SPE);

  // Disable Timer/Counter 0
  if (TIMER0disable)
    TCCR0B &= ~((1 << CS02) | (1 << CS01) | (1 << CS00));

  // Disable Timer/Counter 1.
  if (TIMER1disable)
    TCCR1B &= ~((1 << CS12) | (1 << CS11) | (1 << CS10));

  // Disable Timer/Counter 2.
  if (TIMER2disable)
    TCCR2B &= ~((1 << CS22) | (1 << CS21) | (1 << CS20));

  // Disable TWI.
  if (TWIdisable)
    TWCR &= ~(1 << TWEN);

  // Disable Analog comparator.
  if (ACdisable)
    ACSR &= ~(1 << ACD);
}
```

The `setup()` function in Listing 7-16 calls `disable()` with a list of peripherals to disable, then calls the AVRLib's `__power_all_disable()` function which powers down all the peripherals. It then powers Timer 0 back on to ensure that it is still working as it is needed by the sketch.

Listing 7-16 Low-power blink, setup() function

```
void setup() {
  // Disable the peripherals we don't want.
  // They are still powered on though.
  disable(
    /* ADCdisable = */    true,
    /* USARTdisable = */  true,
    /* SPIdisable = */    true,
    /* TIMER0disable = */ false,
    /* TIMER1disable = */ true,
    /* TIMER2disable = */ true,
    /* TWIdisable = */    true,
    /* ACdisable = */     true);

  // Power down everything except Timer/Counter 0.
  // It's quicker this way, and less code bloat!
  __power_all_disable();
  power_timer0_enable();

  // Finally, do the sketch stuff.
  pinMode(LED_BUILTIN, OUTPUT);
}
```

The `loop()` in Listing 7-17 just blinks the LED as usual.

Listing 7-17 Low-power blink, loop() function

```
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

The sketch uses 1,006 bytes of Flash RAM now, which is obviously more than the standard blink sketch would use. In larger sketches, the overhead should be less.

7.5.4 The Power Functions

As I mentioned earlier, there are useful power handling functions in the AVRLib library. These are, for the ATmega328P, as shown in Table 7-17.

Not all devices have these functions, it's another one of those configuration things. The `iom328p.h` header file sets up the appropriate functions which are available for peripheral devices on the ATmega328P.

Note
You are required to manually turn off the Analog Comparator if you do not need it; there doesn't appear to be a utility function to do so in the AVR Lib code.

Table 7-17 AVRLib power functions

Function Name	Description
power_adc_enable()	Enables power to the ADC
power_adc_disable()	Disables power to the ADC
power_spi_enable()	Enables power to SPI
power_spi_disable()	Disables power to SPI
power_timer0_enable()	Enables power to Timer 0
power_timer0_disable()	Disables power to Timer 0
power_timer1_enable()	Enables power to Timer 1
power_timer1_disable()	Disables power to Timer 1
power_timer2_enable()	Enables power to Timer 2
power_timer2_disable()	Disables power to Timer 2
power_twi_enable()	Enables power to TWI
power_twi_disable()	Disables power to TWI
power_usart0_enable()	Enables power to the USART
power_usart0_disable()	Disables power to the USART
__power_all_enable()	Enables power to *all* peripherals
__power_all_disable()	Disables power to *all* peripherals

7.6 Bootloaders

The ATmega328P on an Arduino board is supplied already programmed with a bootloader. This is a small area of the Flash RAM set aside for a special program, and when the device is reset or powered on, a jump to the bootloader takes place.

The standard Uno bootloader delays startup of the ATmega328P for a brief period to check that no programming commands are being received on the USART pins (`D0` and `D1`). If there are no commands, the application code starts normally and the blink sketch, or whatever you programmed last, starts executing.

If there are specific commands being sent to the USART, then the bootloader starts running those commands and may, depending on what it is being commanded to do, overwrite the previously uploaded code with a new version or just upload a new sketch to the application area of the Flash RAM. The bootloader cannot update itself with a new version.

7.6.1 Flash Memory

The Flash memory in the ATmega328P is divided into two sections:

Application section	This is where your sketch code is written to by the bootloader or the ICSP device.
Bootloader section	The BLS is where the bootloader lives.

There are fuses, `BOOTSZ1:0` and `BOOTRST`, to determine the size and address of the bootloader sections, and in the case of the `BOOTRST` fuse, it determines whether the device starts executing the bootloader or the application code on startup and/or reset.

Table 7-18 Device lock bits 0

LB Mode	LB2:1	Description
1	11	The device is totally unprotected. It can be programmed, or Flash and EEPROM contents and fuse bits can be read at will
2	10	Programming the device—Flash or EEPROM—is disabled in serial or parallel modes. The fuse bits are also locked. Any code already programmed can still be read
3	00	Programming and reading the device—Flash or EEPROM—is disabled in serial or parallel modes. The fuse bits are also locked

Note

While it is possible and indeed permitted for the bootloader to write to the application section, the converse is not true. The application cannot access the bootloader section.

You should also note that the whole of Flash RAM can be used as the application section if no bootloader is required. This is easily done by using an ICSP device to do the programming which will overwrite the bootloader section, if necessary.

The sections are considered completely separate by the device, and they can have different protection levels. This protection is determined by special lock bits. Boot Loader Lock Bits 0 protect the application section, while Boot Loader Lock Bits 1 protect the bootloader section. There are two other lock bits that protect the entire ATmega328P from either being reprogrammed and/or having its Flash RAM read out.

7.6.2 Lock Bits

The lock bits can be set in software and in serial or parallel programming modes. To clear the *bootloader* lock bits, a full chip erase command must be performed. It may not be possible—I have not checked this on my own devices—to ever unlock the device for programming if the device lock bits have been set. I have not checked as I don't wish to lock myself out of my own devices, if the locking can't be undone!

7.6.2.1 Device Lock Bits

The device lock bits can prevent anyone from reading or changing the contents of the AVR microcontroller's Flash RAM and/or EEPROM. The lock bits are named `LB1` and `LB2` and have the *LB Modes* and settings shown in Table 7-18:

The default is Mode 1 which allows the device to be programmed and verified (read back) as required. This also means that your code can be extracted and used to program other devices.

Tip

In case you are wondering, serial programming is when either an ICSP device or a high voltage serial programmer is used and requires only a few pins; parallel, on the other hand, requires many more pins and is uncommon outside of large establishments which need to program many devices at once.

There is a very good description and circuit diagrams of both methods at Nick Gammon's blog at www.gammon.com.au/forum/?id=12898 if you are interested.

Note
As with fuses, lock bits are considered programmed when zero and unprogrammed when one.

7.6.2.2 Bootloader Lock Bits

With the two sets of lock bits, the following options can be selected. Details follow for the modes mentioned:

- The entire Flash RAM can be protected from a software update—BLB0 Mode 2 and BLB1 Mode 2.
- Only the bootloader section can be protected—BLB0 Mode 1 and BLB1 Mode 2 or 3.
- Only the application section can be protected—BLB0 Mode 2 or 3 and BLB1 Mode 1.
- The entire Flash RAM can be unprotected—BLB0 Mode 1 and BLB1 Mode 1.

Bootloader Lock Bits 0 These bits protect the *application* section of the device. There are two bits here, `BLB01` and `BLB02`, and these are set according to a *BLB Mode*, named `BLB0` Mode. Table 7-19 summarizes the different modes.

Bootloader Lock Bits 1 These bits protect the *bootloader* section of the device. There are two bits here, `BLB11` and `BLB12`, and these are set according to BLB Mode `BLB1`. Table 7-20 summarizes the different modes.

Table 7-19 Bootloader lock bits 0

BLB0 Mode	BLB02:1	Description
1	0b11	The application section is totally unprotected
2	0b10	The Store Program Memory (`SPM`) instruction is not allowed to write to the application section. The application section is fully protected
3	0b01	The `SPM` instruction cannot write to the application section, and, at the same time, the Load Program Memory (`LPM`) instruction cannot read from it if executing from the bootloader. See the following note
4	0b00	The `LPM` instruction, if executing from the bootloader, is not allowed to read from the application section. See the following note

Note: If interrupt vectors are located in the bootloader section, interrupts will be disabled while code is executing from the application section

Table 7-20 Bootloader lock bits 1

BLB1 Mode	BLB02:1	Description
1	0b11	The bootloader section is unprotected and can be written to, but *not* by code running in the application section
2	0b10	The Store Program Memory (SPM) instruction is not allowed to write to the bootloader section. The bootloader section is fully protected
3	0b01	The SPM instruction cannot write to the bootloader section, and, at the same time, the Load Program Memory (LPM) instruction cannot read from it if executing from the application section. See the following note
4	0b00	The LPM instruction, if executing from the application section, is not allowed to read from the bootloader section. See the following note

Note: If interrupt vectors are located in the application section, interrupts will be disabled while code is executing from the bootloader section

7.6.3 Installing the Uno (Optiboot) Bootloader

The Uno bootloader is 500 bytes in size, so takes up less of your precious Flash RAM when installed. It is found, should you wish to examine it in detail, in `\$ARDINST/bootloaders/optiboot/optiboot.c`.

Although it's commonly referred to as the *Uno bootloader*, it is, in fact, quite easily installable into other devices. The comments in the source code mention that it is compatible with the Duemilanove, Diecimila, and other ATmega168- or ATmega328P-based devices.

If you wish to use a much smaller bootloader on your Duemilanove, for example, and save 1.5 Kb of Flash RAM for your own programs, it's easy:

- Close the Arduino IDE if it is open.
- Create or open the file `\$ARDINST/boards.local.txt` in your favorite text editor.
- Add the following three lines. These settings assume the Duemilanove is the ATmega328P variant and all are on one line each—there's no wrapping:

```
diecimila.menu.cpu.atmega328.upload.maximum_size=32256
diecimila.menu.cpu.atmega328.bootloader.high_fuses=0xDE
diecimila.menu.cpu.atmega328.bootloader.file=
optiboot/optiboot_atmega328.hex
```

- Save the file.
- Open the IDE again.
- Make sure that the correct board is selected under Tools ▹ Board.
- Choose Tools ▹ Burn Bootloader.

You now have a smaller, faster bootloader and an additional 1.5 Kb of Flash for your program—and all for free.

Installing the Optiboot bootloader for the Duemilanove means that it will no longer suffer from Watchdog reset loops!

7.6.4 Optiboot Bootloader Operation

When the device is powered on, or reset, and if the `BOOTRST` fuse is set to enable the bootloader to be executed at startup, then the bootloader code for the device begins executing.

One of the first tasks that the bootloader does is to check the `MCUSR` register to determine if this was an external reset or not. It does this by checking the `EXTRF` bit in the `MCUSR` register. If that bit is set, then the device was reset by pulling pin 1, RST low and not by a power on, Watchdog, or BOD reset. This could have been done by the user pressing the reset button on the Arduino board or by the programming device using the DTR pin to pull the ATmega328P's RST pin low. In any case, bit `EXTRF` will have been set.

In the case when this bit is not set, the bootloader assumes that the device is not about to be reprogrammed and jumps immediately to the application code, bypassing the rest of the bootloader itself. If the bit is set, then the bootloader continues executing.

The Watchdog Timer is set to fire after one second, the onboard LED is flashed once to indicate that it is waiting, and an infinite loop is entered to wait for characters coming in over the USART. If nothing is received after the one-second timeout by the Watchdog, then the device will reset again, but this time by the Watchdog. On restarting from a Watchdog-induced reset, the `EXTRF` bit in the

`MCUSR` register will no longer be set—the `WDRF` bit, on the other hand, will be set—so the application code is immediately executed.

The bootloader, if it continues executing, must have read at least one byte from the USART. These bytes are assumed to be commands from a subset of the STK500 communications protocol—the Optiboot bootloader currently only implements a subset of the STK500 instructions—and these commands are used to communicate with, usually, *avrdude*.

It is beyond the scope of this book to delve into the various bootloader commands as they really have little to do with application programming. Suffice it to say that the bootloader sits in a loop, reading characters from the USART and acting upon them, in addition to resetting the Watchdog Timer to prevent the device from being reset by the Watchdog and messing up the programming.

Should you really wish to examine the Optiboot bootloader, there is a compilation listing for the ATmega328P, in the location `\$ARDINST/bootloaders/optiboot/optiboot_atmega328.lst`—it does make interesting[2] reading.

[2]For certain values of *interesting*!

ATmega328P Hardware—Timers and Counters 8

This chapter, and the next, looks at the hardware features of the ATmega328P and links up with the information presented in Chapters 2, 3, and 4. You should, I hope, see how the Arduino Language talks to the hardware described in these chapters. This chapter starts by looking, long and hard, at the facilities of the ATmega's timer/counters, while the following chapter delves into the ADC and serial communications hardware, the USART.

8.1 Timers

The ATmega328P has a total of three timers named Timer/counter 0, Timer/counter 1, and Timer/-counter[1] 2. The first and last of these are both 8-bit timers and have a maximum value of 255, while Timer 1 is 16 bits and its maximum is 65,535.

At power on, or reset, all timers are disabled and must be enabled in software. The Arduino's `init()` function does a lot of timer initialization so that the `millis()`, `micros()`, and `analogWrite()` functions work.

Timer/counters are so called because they have two separate functions; they can

- Count up and down, depending on the mode, with a clock source based off of the ATmega328P's system clock. This is when it acts as a timer.
- Count up and down, based on an external rising or falling edge attached to a specific pin. This is when it acts as a counter.

The data sheet for the ATmega328P advises that

> When setting various timer modes, any output pins affected should be set to OUTPUT *after* setting the required modes.

In much of what follows, you may see references to `TOP`, `MAX`, and/or `BOTTOM`. These are the definitions that are used in the data sheet for the ATmega328P and refer to the following:

`BOTTOM` It is easy; it is always zero.

[1] Henceforth, to be abbreviated to just "timer." I'm a lazy typist!

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_8

`MAX` It is also easy; it is always the maximum value that can be held in the timer's TCNTn register according to however many bits the timer is configured for. This is calculated as $2^{bits} - 1$.

For Timers 0 and 2, this is always eight bits, and so `MAX` always equals 255. For Timer 1, `MAX` varies as follows:

- In 8-bit mode, MAX = 255.
- In 9-bit mode, MAX = 511.
- In 10-bit mode, MAX = 1,023.
- In 16-bit mode, MAX = 65,535.

`TOP` It depends on the timer's mode and is either `MAX` for some modes or as defined by various other timer registers such as OCRnA, OCRnB, ICR1, etc.

8.1.1 Timer 0 (Eight Bits)

This timer has six different modes of operation defined by the Waveform Generation bits, `WGM02:0`, in the registers `TCCR0A` and `TCCR0B` (Timer 0 Control Registers A and B). Bit `WGM02` is found in `TCCR0B`, while `WGM01` and `WGM00` are in `TCCR0A`.

Table 8-1 shows the various settings for the modes available in Timer 0.

Modes 4 and 6 are reserved and should not be configured.

When you write a new value to register OCR0x, the value doesn't get written until the timer's value, `TCNT0`, reaches that shown in Table 8-1, where "NOW" means that the new value is written as soon as your code executes it, and any other value means that the new value will be written when `TCNT0` reaches that given value. Likewise, the overflow bit, `TOV0`, is set when the timer's count reaches the indicated value.

8.1.2 Timer 1 (16 Bits)

This timer has 15 different modes of operation defined by the Waveform Generation bits, `WGM13:0`, in the registers `TCCR1A` and `TCCR1B` (Timer 1 Control Registers A and B). Bits `WG13` and `WGM12` are found in `TCCR1B`, while `WGM11` and `WGM10` are in `TCCR1A`.

Table 8-2 shows the various settings for the modes available in Timer 1.

Mode 13 is reserved and should not be configured.

When you write a new value to register OCR1x, the value doesn't get written until the timer's value, `TCNT1`, reaches that shown in Table 8-2, where "NOW" means that the new value is written as soon as your code executes it, and any other value means that the new value will be written when `TCNT1`

Table 8-1 Timer 0 modes

Mode	WGM02:00	TOP	OCR0x Updated	TOV0 Set	Mode of Operation
0	0b000	255	NOW	255	Normal
1	0b001	255	TOP	0	PWM, Phase Correct
2	0b010	OCR0A	NOW	255	CTC
3	0b011	255	0	255	PWM, Fast
5	0b101	OCR0A	TOP	0	PWM, Phase Correct
7	0b111	OCR0A	0	TOP	PWM, Fast

Table 8-2 Timer 1 modes

Mode	WGM13:10	TOP	OCR1x Updated	TOV1 Set	Mode of Operation
0	0b0000	65,535	NOW	65,535	Normal
1	0b0001	255	TOP	0	PWM 8 bit, Phase Correct
2	0b0010	511	TOP	0	PWM 9 bit, Phase Correct
3	0b0011	1,023	TOP	0	PWM 10 bit, Phase Correct
4	0b0100	OCR1A	NOW	65,535	CTC
5	0b0101	255	0	TOP	PWM 8 bit, Fast
6	0b0110	511	0	TOP	PWM 9 bit, Fast
7	0b0111	1,023	0	TOP	PWM 10 bit, Fast
8	0b1000	ICR1	0	0	PWM, Phase and Frequency Correct
9	0b1001	OCR1A	0	0	PWM, Phase and Frequency Correct
10	0b1010	ICR1	TOP	0	PWM, Phase Correct
11	0b1011	OCR1A	TOP	0	PWM, Phase Correct
12	0b1100	ICR1	NOW	65,535	CTC
14	0b1110	ICR1	0	TOP	PWM, Fast
15	0b1111	OCR1A	0	TOP	PWM, Fast

Table 8-3 Timer 2 modes

Mode	WGM22:20	TOP	OCR2x Updated	TOV2 Set	Mode of Operation
0	0b000	255	NOW	255	Normal
1	0b001	255	TOP	0	PWM, Phase Correct
2	0b010	OCR2A	NOW	255	CTC
3	0b011	255	0	255	PWM, Fast
5	0b101	OCR2A	TOP	0	PWM, Phase Correct
7	0b111	OCR2A	0	TOP	PWM, Fast

reaches that given value. Likewise, the overflow bit, `TOV1`, is set when the timer's count reaches the indicated value.

Note
Timer 1 has an extra control register, `TCCR1C`, which the other two timers don't have.

8.1.3 Timer 2 (Eight Bits)

This timer has six different modes of operation defined by the Waveform Generation bits, `WGM22:0`, in the registers `TCCR2A` and `TCCR2B` (Timer 2 Control Registers A and B). Bit `WGM22` is found in `TCCR2B`, while `WGM21` and `WGM20` are in `TCCR2A`.

Table 8-3 shows the various settings for the modes available in Timer 2.

Modes 4 and 6 are reserved and should not be configured.

When you write a new value to register OCR2x, the value doesn't get written until the timer's value, `TCNT2`, reaches that shown in Table 8-3, where "NOW" means that the new value is written as soon as your code executes it, and any other value means that the new value will be written when `TCNT2` reaches that given value. Likewise, the overflow bit, `TOV2`, is set when the timer's count reaches the indicated value.

Table 8-4 Timer clock sources

Prescaler	Timer 0 CS02:00	Timer 1 CS12:10	Timer 2 CS22:20	Frequency	Period
Disabled	0b000	0b000	0b000	–	–
Divide by 1	0b001	0b001	0b001	16 MHz	0.0625 uS
Divide by 8	0b010	0b010	0b010	2 MHz	0.5 uS
Divide by 32	–	–	0b011	500 KHz	2 uS
Divide by 64	0b011	0b011	0b100	250 KHz	4 uS
Divide by 128	–	–	0b101	125 KHz	8 uS
Divide by 256	0b100	0b100	0b110	62.5 KHz	16 uS
Divide by 1,024	0b101	0b101	0b111	15.625 KHz	64 uS
External falling	0b110 pin T0	0b110 pin T1	–	–	–
External rising	0b111 pin T0	0b111 pin T1	–	–	–

8.1.4 Timer Clock Sources

The three timers all have their own individual clock sources, mostly derived from the main system clock. Each timer also has a dedicated prescaler which can be set to divide the system clock by various values, giving the timer's desired clock speed. Timers 0 and 1 can also be configured to clock on an external pin's rising or falling edge.

While all three timers have mostly the same prescaler settings, however, Timer 2 does not have the ability to be externally clocked, but it does have the ability to run with divide by 32 and divide by 128 prescalers, which the others cannot. Timer 2 can also be configured to run in asynchronous mode with a 32.768 KHz crystal attached to the TOSCn pins. This is beyond the scope of this book, however, as the Arduino uses those two pins for a 16 MHz crystal, thus rendering asynchronous mode unusable.

Table 8-4 shows the various clock sources for the three timers.

Note
Pin `T0` in Table 8-4 is physical pin 6 on the ATmega328P, which is `PD4` in AVR terminology or `D4` in the Arduino Language. Pin `T1` is physical pin 11, also known as `PD5` in the AVR terminology and corresponds to Arduino pin `D5`.

8.1.5 Timer Non-PWM Operating Modes

As you have already seen, the three timers can operate in a number of different modes. Not all timers have the same modes though. Tables 8-1 through 8-3 show the modes available for each timer individually.

8.1.5.1 Timers Disabled

This is not really a timer mode, but setting the CSn2, CSn1, and CSn0 bits to zero in register TCCRnB will disable the timers. The "n" in the bit and register name refers to the timer in question.

Disabling Timer 0 will disable the ability to use the `millis()` and `micros()` functions if you are using the Arduino Language. It will also prevent the use of `analogWrite()` on pins `D5` and `D6`.

Disabling Timer 1 will prevent the use of `analogWrite()` on pins `D9` and `D10`.

Disabling Timer 2 will prevent the use of `analogWrite()` on pins `D3` and `D11` and will also affect the ability of the `tone()` function to be used as that uses Timer 2.

Due to the usefulness of the various timers, it is unlikely that they will ever be disabled. However, should your code not require any of the timers, you can save a few microamps of power consumption by disabling power to the timers in the `PRR` (Power Reduction Register), by setting the required bits PRTIMn, where, as ever, "n" refers to the timer number. Section 7.5.2 discusses the PRR in some detail.

Warning
Should you decide to do this, Timer 2 must be running in synchronous mode off of the internal oscillator or on a 16 MHz crystal as per Arduino boards. If the timer is running in asynchronous mode with a 32 KHz or higher crystal, plus `AS2` set in register `ASSR`, then writing a 1 to `PRTIM2` in the `PRR` register will not stop Timer 2.

This latter mode of operation is, of course, not possible on an Arduino board as the 16 MHz crystal is attached to the two pins that an external 32 KHz crystal needs to use for asynchronous mode. On a bare-bones ATmega328P though, this need not be the case.

Table 8-5 shows the settings required to disable the timers.

8.1.5.2 Normal Mode

In normal mode, a timer normally starts counting upward from zero, but this can be changed by writing a new value to TCNTn register, which holds the timer value and counts up every time the timer's clock source ticks, until it reaches `MAX`—refer back to the beginning of this chapter for details of `MAX`, `BOTTOM`, and `TOP` in relation to the three timers—which is 255 for Timer 0 and 2 or 65,535 for Timer 1 in this mode.

On the next clock tick after the timer reaches `MAX`, the timer's value, in TCNTn, will roll over from `MAX` to zero—`BOTTOM`.

A number of things happen when the timer rolls over or overflows:

- The Timer Overflow bit, TOVn, is set in the Timer Interrupt Flag Register, TIFRn, on the same clock tick as the timer's value became zero—after $MAX + 1$ counts—assuming that you didn't change TCNTn. This flag is only ever set, so unless your program uses the timer's overflow interrupt, it will remain set until manually cleared by your sketch writing a 1 to it.
- The Timer Overflow Interrupt will be fired if the Timer Overflow Interrupt Enable bit, TOIEn, is set in the Timer Interrupt Mask Register, TIMSKn, and global interrupts are enabled. In this case, the TOVn bit will be cleared automatically when the interrupt service routine is entered.
- Pin OCnA will perform an action. The action depends on the values in the COMnA1:0 bits, as described in Table 8-6, when TCNTn matches with OCRnA.
- Pin OCnB will perform an action. The action depends on the values in the COMnB1:0 bits, as described in Table 8-7, when TCNTn matches with OCRnB.

Table 8-5 Disable timer settings

Timer	Mode	TTCRnB Bits	Value
0	0	CS02:00	0b000
1	0	CS12:10	0b000
2	0	CS22:20	0b000

Table 8-6 COMnA1:0 settings in normal mode

COMnA1:0	Description
0b00	No effect on pin OCnA
0b01	Pin OCnA toggles on match with OCRnA
0b10	Pin OCnA is set LOW on match with OCRnA
0b11	Pin OCnA is set HIGH on match with OCRnA

Table 8-7 COMnB1:0 settings in normal mode

COMnB1:0	Description
0b00	No effect on pin OCnB
0b01	Pin OCnB toggles on match with OCRnB
0b10	Pin OCnB is set LOW on match with OCRnB
0b11	Pin OCnB is set HIGH on match with OCRnB

Table 8-8 Normal mode settings

Timer	Mode	TTCRnB Bits	Value
0	0	WGM02:00	0b000
1	0	WGM13:10	0b0000
2	0	WGM22:20	0b000

While running in this mode, you can, if you wish, write a value to the timer's TCNTn register, which can reduce the time it takes for the overflow bit to be set or the interrupt to be fired.

Table 8-8 shows the settings required to put the timers into normal mode.

Note that bits WGMn0 and WGMn1 are found in register TCCRnA, bit WGMn2 is found in TCCRnB, and, for Timer 1 only, `WGM13` is also found in `TCCR1B`.

Microchip's data sheets advise that

> The Output Compare unit can be used to generate interrupts at some given time. Using the Output Compare unit to generate waveforms in Normal mode is not recommended, since this will occupy too much of the CPU time.

So, it sounds like we should use overflow interrupts for useful purposes, and this is what the Arduino uses for its `millis()` function, etc., but can we use `OCR0A` and `OCR0B` to generate interrupts as well? Well, Listings 8-1 through 8-3 tell all.

Warning
I did originally write this code using Timer 0, but as the `TIMER0_OVF` handling routine in the Arduino library is already using that timer's overflow interrupt, I got linker errors due to there being two copies of the interrupt handler. Any Arduino code with this vector in use will not be able to be compiled with the IDE.

The sketch initializes Timer 2 with three interrupts in `setup()`. The three interrupts are

- The Overflow interrupt which will toggle an LED on pin `D13`
- The Compare Match A interrupt which will toggle an LED on pin `OC2A` which is Arduino pin `D12`
- The Compare Match B interrupt which will toggle an LED on pin `OC2B` which is Arduino pin `D11`

Listings 8-1, 8-2, and 8-3 show the `setup()`, `loop()`, and interrupt handler code, respectively.

Listing 8-1 Normal timer mode setup() function

```
//========================================================
// This sketch uses the Timer/Counter 2 as follows:
//
// Overflow Interrupt to toggle LED_BUILTIN (D13)
// COMPA Interrupt to toggle D12
// COMPB Interrupt to toggle D11.
//========================================================
void setup() {
  TCCR2A = 0;                                              (1)
  TIMSK2 = ((1 << OCIE2B) |                                (2)
            (1 << OCIE2A) |
            (1 << TOIE2));

  // Set up the compare values.
  OCR2A = 8;                                               (3)
  OCR2B = 172;

  TCCR2B = ((1 << CS22) |                                  (4)
            (1 << CS21) |
            (1 << CS20));

  // D11, D12 and D13 are outputs.
  pinMode(13, OUTPUT);                                     (5)
  pinMode(12, OUTPUT);
  pinMode(11, OUTPUT);
}
```

(1) This clears the timer register to a known starting configuration. This enables normal mode.
(2) This enables interrupts on Overflow and Compare Match A and B.
(3) A couple of random values to compare against, for the interrupts to trigger.
(4) Set the prescaler to 1,024 and start the timer.
(5) Configure the LED pins as OUTPUT after setting the timer configuration, as per the data sheet.

Listing 8-2 Normal timer mode loop() function

```
void loop() {
  // Nothing happening here, move along now!
}
```

As you can see, the `loop()` function is empty—the timer interrupts take care of flashing the LEDs without needing the `loop()` to do anything.

Listing 8-3 Normal timer mode ISRs

```
// Toggle pin D13 which is PortB pin 5.
ISR(TIMER2_OVF_vect) {
  // Fast pin toggle.
  PINB |= (1 << PINB5);
}

// Toggle pin D12 which is PortB pin 4.
ISR(TIMER2_COMPA_vect) {
  // Fast pin toggle.
  PINB |= (1 << PINB4);
}

// Toggle pin D11 which is PortB pin 3.
  ISR(TIMER2_COMPB_vect) {
  // Fast pin toggle.
  PINB |= (1 << PINB3);
}
```

If you set this up with an LED on pin `D13` (or use the built-in LED), another on pin `D12`, and a third on pin `D11`, then they will all flash, so the interrupts are working. All three will flash at exactly the same frequency because there is an Overflow interrupt every 256 clock ticks, and both `OCR2A` and `OCR2B` will match `TCNT2` once every 256 clock ticks also.

The timer is running with a frequency of $F\_CPU/prescaler$, which means

$$\begin{aligned}\text{Timer Frequency} &= 16{,}000{,}000/1{,}024\\ &= 15{,}625\text{ Hz}\end{aligned}$$

We are toggling every 256 counts of the clock, and it takes two toggles to make one flash of the LED, so that's

$$\begin{aligned}\text{LED Frequency} &= \text{Timer Frequency}/256/2\\ &= 15{,}625/256/2\\ &= 30.5176\text{ Hz}\end{aligned}$$

And that means we have a flash every 32.768 milliseconds. (The period is 1/frequency.)

Attaching my Labrador oscilloscope to the LEDs one at a time shows that they all have the same frequency, and it's calculated as 31.03 Hz, so it's not far off. It's obviously my ability to accurately place the cursors on the display to get the correct measurements that is affecting the results, but it's close enough.

Don't forget that when an interrupt fires, it disables further interrupts, plus it takes four clock cycles to process the interrupt handler jump and another four to return, and those delays are not being considered here.

So we now know that interrupts work in normal mode; what about toggling the pins by setting the various COM2An and COM2Bn bits to toggle the pins when there's a compare match? Listings 8-4 to 8-6 show an amended sketch to do just that. Note that while `D13` still has the same connections, the LED on `D12` has to be moved to `D3` because now we are using the timer's hardware to toggle the pins and not the interrupts. The pins that the hardware toggles for us are `OC2A` or `D11` and `OC2B` or `D3`.

Listing 8-4 Getting the timer to flash LEDs, setup()

```
//==========================================================
// This sketch uses the Timer/Counter 2 as follows:
//
// Overflow Interrupt to toggle LED_BUILTIN (D13)
// OC2A to toggle D11
// OC2B to toggle D3.
//==========================================================
void setup() {
  // Initialise Timer/counter 2 in normal mode
  // with OC2A (D11) and OC2B (D3) toggling on match.
  TCCR2A = ((1 << COM2A0) | (1 << COM2B0));            (1)

  // Enable overflow interrupt (on D13 = PB5)
  TIMSK2 = (1 << TOIE2);                               (2)

  // Set up the compare values.
  OCR2A = 8;                                           (3)
  OCR2B = 172;

  // Prescale by 1024, and start the timer.
  TCCR2B = ((1 << CS22) |                              (4)
            (1 << CS21) |
            (1 << CS20));

  // D11, D12 and D3 are outputs.
  pinMode(13, OUTPUT);                                 (5)
  pinMode(11, OUTPUT);
  pinMode(3, OUTPUT);
}
```

(1) Here, we put the timer into normal mode again, but configure it to also toggle pins `OC2A` and `OC2B` when there is a compare match. This does not require the use of interrupts—the toggling of the pins is controlled by the timer alone. The CPU is not involved.
(2) We still use the overflow interrupt to toggle `D13` as before. The CPU will be involved here.
(3) I'm using the same values as before.

(4) The timer's clock source is configured as divide by 1,024, and this starts the timer running.
(5) As before, we have to set the output pins after setting up the timer.

As before, the `loop()` function, Listing 8-5, is empty and has nothing to do.

Listing 8-5 Getting the timer to flash LEDs, loop()

```
void loop() {
  // Nothing to see here, move along now!
}
```

Finally, the code in Listing 8-6 now has a single ISR; this one is required for the Overflow interrupt. The two Compare Match interrupts are no longer required as the timer will toggle the LEDs without the use of the main CPU.

Listing 8-6 Getting the timer to flash LEDs, ISR

```
// Toggle pin D13 which is PortB pin 5.
ISR(TIMER2_OVF_vect) {
  // Fast pin toggle.
  PINB |= (1 << PINB5);
}
```

It looks like those settings work too. The frequency and period of the flashing LEDs are exactly as before on all the pins—30.5176 Hz on all three LEDs—we are still getting one flash every 256 counts on the timer.

8.1.5.3 Clear Timer on Compare Match Mode

In Clear Timer on Compare Match mode (CTC), the timer counts upward from `BOTTOM` (or from the value your code wrote to TCNTn) until it reaches `TOP` which is the value stored in OCRnA, whereupon, on the next timer clock pulse, the value in TCNTn will be cleared to zero. This is mode 2 for Timer 0 and Timer 2 and mode 4 for Timer 1. Timer 1 also has mode 12 CTC, which is discussed separately.

You can change the values in OCRnA and/or OCRnB at any time, but you must be careful as double buffering is not enabled on those registers in this mode. Any changes you make are written directly to the register(s) at the time that your sketch does so. In other modes, these registers do not get changed until a certain point in the count as the values are held in an internal buffer until the specific point is reached. This prevents what the data sheet refers to as "glitches" in those other modes. There is no such protection in CTC mode.

If you change the value to a new value close, or closer, to `BOTTOM` while the counter is running with a low or no prescaler, then CTC mode might miss a match if the current value in TCNTn is higher than the value just written to the OCRnx register. It will count right up to the timer's maximum value and then roll over to zero before it can start the normal sequence of events again—a glitch, in other words. It is better to control the changes to the OCRnx registers by utilizing the Overflow interrupt to make the changes as doing so means that changes always happen at `BOTTOM` which helps to avoid the glitches.

At `TOP`, the timer's value, in TCNTn, will roll over from `TOP` to zero, `BOTTOM`, on the following tick of the timer's clock.

Table 8-9 COMnA1:0 settings in CTC mode

COMnA1:0	Description
0b00	No effect on pin OCnA
0b01	Pin OCnA toggles on match with OCRnA
0b10	Pin OCnA is set LOW on match with OCRnA
0b11	Pin OCnA is set HIGH on match with OCRnA

Table 8-10 COMnB1:0 settings in CTC mode

COMnB1:0	Description
0b00	No effect on pin OCnB
0b01	Pin OCnB toggles on match with OCRnB
0b10	Pin OCnB is set LOW on match with OCRnB
0b11	Pin OCnB is set HIGH on match with OCRnB

Some things happen when the timer clears:

- The Timer Overflow bit, TOVn, is set in the Timer Interrupt Flag Register, TIFRn, on the *same clock tick* as the timer's value became zero—after TOP +1 counts, assuming that you didn't change TCNTn. This flag is only ever set, and, unless your program has enabled the timer's overflow interrupt, it will remain set until manually cleared by your sketch. You manually clear this bit by writing a 1 to it.
- The Timer Overflow Interrupt will be fired if the Timer Overflow Interrupt Enable bit, OCFnA, is set in the Timer Interrupt Mask Register, TIMSKn, and global interrupts are enabled. In this case, the TOVn bit will be cleared automatically when the interrupt service routine is entered.
- Pin OCnA will perform an action. The action depends on the values in the COMnA1:0 bits, as described in Table 8-9, when TCNTn matches with OCRnA.
- Pin OCnB will perform an action. The action depends on the values in the COMnB1:0 bits, as described in Table 8-10, when TCNTn matches with OCRnB.

Warning
The value in OCRnA is always the `TOP` value. If OCRnB is higher than the value in OCRnA, then there will be no effect on the OCnB pin, as the value in TCNTn will never reach the value in OCRnB. You will only see the desired effect on the OCnB pin if the value in OCRnB is less than or equal to the value in OCRnA.

The data sheet for the ATmega328P doesn't make this clear, and many online forums have lots of confusion on the matter.

Note
The actions carried out when bits `COMnA1:0` and `COMnB1:0` are used are the same for both non-PWM timer modes, Normal and CTC.

Timers 0 and 2 don't have any additional CTC modes, so the preceding description applies to those timers.

Timer 1 has two CTC modes, modes 4 and 12. In mode 4 CTC, Timer 1 acts exactly as described.

When Timer 1 is configured in CTC mode 12, the `TOP` value is defined by the value in the `ICR1` or Input Capture Register, and this is attached to the Input Capture Unit for this timer, as described in Section 8.3. The Input Capture Unit copies the Timer 1 value from `TCNT1` to the `ICR1` register each time an "event" occurs. This value is then used as `TOP` in CTC mode 12 for Timer 1.

In CTC mode 12, the following will occur when `TCNT1` matches `ICR1`:

- Bit `ICF1` is set in the Timer 1 Interrupt Flag Register `TIFR1`. This bit will be cleared if the Input Capture Interrupt is enabled by setting bit `ICIE1` in register `TIMSK1`, when the interrupt routine is executed. If interrupts are not used, then your sketch must clear the `ICF1` flag by writing a 1 to it.
- The Input Capture Interrupt will be fired, automatically clearing the `ICF1` flag, if configured to do so.
- The effects of the `COM1A1:0` and `COM1B1:0` bits are exactly as previously described.

While running any timer in any CTC mode, you can, if you wish, write a value to the timer's counter register, TCNTn, which can reduce the time it takes for the overflow bit to be set or the interrupt to be fired. The interrupt itself can write a new value to TCNTn if necessary.

Table 8-11 shows the settings required to put the timers into CTC mode.

Bits WGMn0 and WGMn1 are found in register TCCRnA, bit WGMn2 is found in TCCRnB, and, for Timer 1 only, `WGM13` is also found in `TCCR1B`.

The maximum frequency at which the OCnA and/or OCnB pins will toggle is given by

$$Frequency = F\_CPU/(2 * prescaler * (1 + OCRnA))$$

which, if OCRnA is zero, makes the maximum frequency possible equal to

$$F\_CPU/(2 * prescaler))$$

The value to be loaded into OCRnA is calculated as

$$OCRnA = (F\_CPU/(Frequency * 2 * prescaler)) - 1$$

for any desired timer frequency, provided that the answer fits onto the appropriate timer's OCRnA register.

Listings 8-7 and 8-8 illustrate a sketch which sets up Timer 2 in CTC mode 2, turns off all interrupts from the timer, and sets pins `OC2A` and `OC2B` to toggle whenever the value in `TCNT2` matches either of `OCR2A` or `OCR2B`. `TCNT2` will be cleared to zero on the timer clock pulse after it equals `OCR2A`.

`OCR2A` defines the `TOP` value for this sketch. In Listing 8-7, it is set to 200, giving 201 counts per cycle. `OCR2B` is initialized to the value 86, which is less than `OCR2A`, so it will be affected by the running timer and will flash. The prescaler is again 1,024. From these values, the expected frequency

Table 8-11 CTC mode settings

Timer	Mode	TTCRnB Bits	Value
0	2	WGM02:00	0b010
1	4	WGM13:10	0b0100
1	12	WGM13:10	0b1100
2	2	WGM22:20	0b010

of both LEDs will be

$$\begin{aligned}\text{LED Frequency} &= 16{,}000{,}000/(2 * 1{,}024 * (1 + 200))\\ &= 16{,}000{,}000/(2{,}048 * 201)\\ &= 16{,}000{,}000/411{,}648\\ &= 38.8682\ \text{Hz}\end{aligned}$$

Listing 8-7 is the `setup()` function for the sketch.

Listing 8-7 CTC example sketch, setup() function

```
//==========================================================
// This sketch uses the Timer/Counter 2 in CTC mode 2 as
// follows:
//
// OC2A to toggle D11 when TCNT2 matches OCR2A.
// OC2B to toggle D3 when TCNT2 matches OCR2B.
//
// Frequency = F_CPU / (2 * prescaler * (OCR2A + 1)
//==========================================================

void setup() {
  // Initialise Timer/counter 2 in CTC mode. (Mode 2)
  TCCR2A = ((1 << WGM21) |                            (1)
  (1 << COM2A0) |
  (1 << COM2B0));

  // Disable interrupts on Timer 2.
  TIMSK2 = 0;                                         (2)

  // Set up the compare values.
  OCR2A = 200;                                        (3)
  OCR2B = 86;

  // Prescale by 1024, and start the timer.
  TCCR2B = ((1 << CS22) |                             (4)
            (1 << CS21) |
            (1 << CS20));

  // D11 and D3 are outputs.
  pinMode(11, OUTPUT);                                (5)
  pinMode(3, OUTPUT);
}
```

(1) Timer 2 is configured here with CTC mode 2 and both the `OC2A` (`D11`) and `OC2B` (`D3`) pins to toggle when there is a compare match between `TCNT2` and either `OCR2A` or `ORC2B`.

(2) All interrupts are disabled for Timer 2.
(3) The two required match values are set up here. As `OCR2A` is higher than `OCR2B`, both output pins will be affected when the counts match.
(4) This is the point where the timer's prescaler is set to divide the system clock by 1,024, which starts the timer running.
(5) The output pins are configured after the timer, as stated in the data sheet.

Listing 8-8 shows a very empty `loop()` function.

Listing 8-8 CTC example sketch, loop() function

```
void loop() {
  // Nothing happening here, move along now!
}
```

Using my trusty Labrador oscilloscope, I measured a frequency of 39 Hz on both LEDs, so I was close to the expected 38.8682 Hz.

You will notice that `loop()` is empty, again. The timer hardware is doing all the hard work of toggling the LEDs for us. We could add some code to `loop()` to do something useful and still have the two LEDS flashing away unaffected. Listings 8-9 and 8-10 show the changes that need to be made to Listings 8-7 and 8-8 to get the `loop()` function working hard.

Add the line in Listing 8-9 to the `setup()` function, just after the existing `pinMode()` calls.

Listing 8-9 CTC example, setup() function changes

```
  pinMode(LED_BUILTIN, OUTPUT);
```

Change `loop()` to resemble Listing 8-10.

Listing 8-10 CTC example, loop() function changes

```
void loop() {
  // Toggle D13 and use a delay().
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
}
```

Compile and upload the sketch. The LED on `D13` will toggle at the usual rate of once every second, controlled by `loop()`, while the other two LEDs are completely unaffected by the calls to `delay()` and continue "blinking" at a frequency of almost 39 Hz. I use quotes around "blinking" as the rate is quite fast on a 16 MHz Arduino, so the LEDs appear on if you stare at them directly. If you see them in your peripheral vision, you will make out a flashing.

As the frequency is roughly 39 Hz, the period is 25.64 mS or 12.82 mS on and 12.82 mS off. Pretty quick, but you can see it in your peripheral vision. The human eye is truly amazing, sometimes.

Note
Yes, I know I'm mixing and matching Arduino code and AVR code, but that's what happens sometimes. I've seen many sketches where the vast majority was written in Arduino code—it's far easier on the eye after all—with only the timing-critical parts of the code written in plain AVR language. This is usually because the facilities of the ATmega328P being used are not available in the Arduino Language.

8.1.6 Timer PWM Modes

The timers can be configured to generate Pulse Width Modulation (PWM) on certain pins. As with the modes already discussed, the timers have a given frequency—which can be changed; see Section 8.1.6.2—however, unlike the other modes, the amount of time that the pins stay `HIGH` can also be changed, even on the fly as the code is running.

The frequency of a waveform is the number of times a second that it moves through a single wave—from crest to crest or trough to trough. The period is the time it takes to do so. If the frequency is 400 Hz, then the period is $1/400$, or 2.5 mS.

In non-PWM modes, the pin connected to the waveform generator is `HIGH` for 50% of the time and `LOW` for the other 50%. With PWM waveforms, the time that the pin is `HIGH` in each period is adjustable and not stuck at 50%.

8.1.6.1 Duty Cycle

The amount of time that a pin stays high during each period is normally specified as a percentage and is called the duty cycle. On the Arduino, the duty cycle for the `analogWrite(value)` function call is simply

$$(value * 100)/256$$

Calling `analogWrite(pin, 128)` is setting a duty cycle of 50%—the pin will be `HIGH` for 50% of the period and `LOW` for 50%.

The image in Figure 8-1 was created on a Labrador oscilloscope which was monitoring pin D9 when the Arduino was executing the statement `analogWrite(9, 128)`.

In the top-right corner of Figure 8-1, you can see that the frequency (f) is listed as 489.22 Hz which is approximately 490 Hz as the Arduino documentation states. That will be the same on all PWM pins except `D5` and `D6`. For those, as will be explained, the image in Figure 8-2 applies.

This time, the frequency shows as 973.66 Hz and is roughly the approximate 980 Hz as mentioned in the Arduino documentation. The documentation is a bit too approximate though as the actual frequency is only 976.5625—so my measurement is a tad closer! This example trace was taken from pin `D6`, and the statement executing this time, to give a different graph, was `analogWrite(6, 64)` for a 25% duty cycle.

The general calculation to work out a duty cycle is

$$DutyCycle = (TimeHIGH * 100)/(TimeHIGH + TimeLOW)$$

The duty cycle is useful as it causes what appears to be an analog voltage on the output pin, rather than a digital `HIGH` or `LOW`. The voltage that appears to be present on the pin, and can be measured,

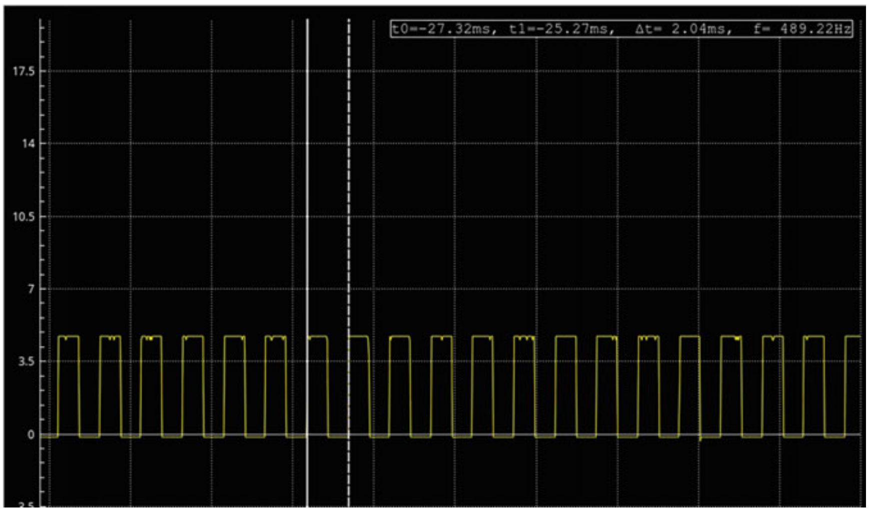


Figure 8-1 Phase Correct PWM with 50% duty cycle

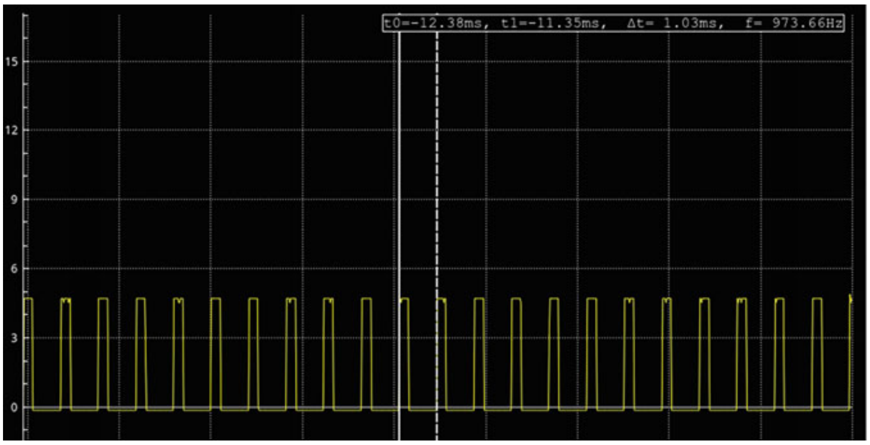


Figure 8-2 Fast PWM with 25% duty cycle

is given by the formula:

$$Duty\ Cycle * VCC$$

So if, for example, the duty cycle is 50% and `VCC` is 5 V, we appear to see a voltage of 2.5 V on the output pin. If the duty cycle is 25%, then we appear to see only 1.25 V on the output pin. This is why an LED can be made to fade in brightness, or a motor with an appropriate driver can be made to speed up or down.

8.1.6.2 PWM Frequencies

The Arduino is set up with two fixed but different PWM frequencies. The frequencies are fixed as a result of the three timers being run in eight-bit mode with a divide by 64 prescaler, but different as Timer 0 is configured for Fast PWM, while the other timers are configured for Phase Correct PWM.

Table 8-12 Prescaler values and PWM frequencies

Prescaler	Fast PWM Frequency	Fast PWM Period	Phase Correct Frequency	Phase Correct Period
1	62.5 KHz	16 uS	31.3725 KHz	31.87 uS
8	7.8125 KHz	128 uS	3.9216 KHz	254.99 uS
32	1.95315 KHz	512 uS	980.3922 KHz	1,020 uS
64	976.5625 Hz	1,024 uS	490.1960 Hz	2,040 uS
128	488.2125 Hz	2,048 uS	245.0980 Hz	4,080 uS
256	244.1406 Hz	4,096 uS	122.5490 Hz	8,160 uS
1,024	61.0352 HZ	16,384 uS	30.6372 Hz	32,640 uS

Table 8-12 shows the relationship between the prescaler values and the two PWM frequencies which correspond to the prescaler value.

If necessary, you can get faster or slower PWM frequencies if your specific project requires them, by taking over the timers/counters and changing things around. If you must do this, then Table 8-12 will help you avoid having to do the arithmetic. It assumes a 16 MHz system clock and an eight-bit counter, like the Arduino.

Only Timer 2 can use the 32 and 128 prescaler values in the table.

Be aware that changing Timer 0 in this fashion will mess up things like `millis()`, `delay()`, and other such things that rely on a prescaler of 64 for accuracy.

The data sheet advises that

> When measuring PWM waveforms, the period is deemed to be measured between each `TOP` (highest) value of the counter.

The PWM frequency, in any PWM mode, is changed by changing the timer's prescaler.

8.1.6.3 Fast PWM Mode

In Fast PWM mode, the value in register TCNTn will increment from zero—`BOTTOM`—until it reaches `TOP`. On the next clock pulse, TCNTn will be reset to zero, `BOTTOM`, and will then continue counting upward again. There are therefore $TOP+1$ steps in each cycle. The `TOP` value is determined by the mode and can be as follows:

- For all timers, the value 255—this is the Arduino default.
- For all timers, the value in register OCRnA.
- For Timer 1 only, the value 511.
- For Timer 1 only, the value 1,023.
- For Timer 1 only, the value in register `ICR1`.

The data sheet advises that

> When changing the `TOP` value the program must ensure that the new `TOP` value is higher or equal to the value of all of the Compare Registers. If the `TOP` value is lower than any of the Compare Registers, a compare match will never occur between the TCNTn and the OCRnx.

This PWM mode is called "single slope" as a graph of TCNTn's value would slope upward, then fall immediately back down to zero like a sawtooth. Figure 8-3 shows the slope of the count in TCNTn as it rises and resets to zero.

Each timer has two pins upon which it can generate PWM waveforms. The Arduino initialization carried out in the `init()` function discussed in Section 2.10 for each and every sketch sets all three

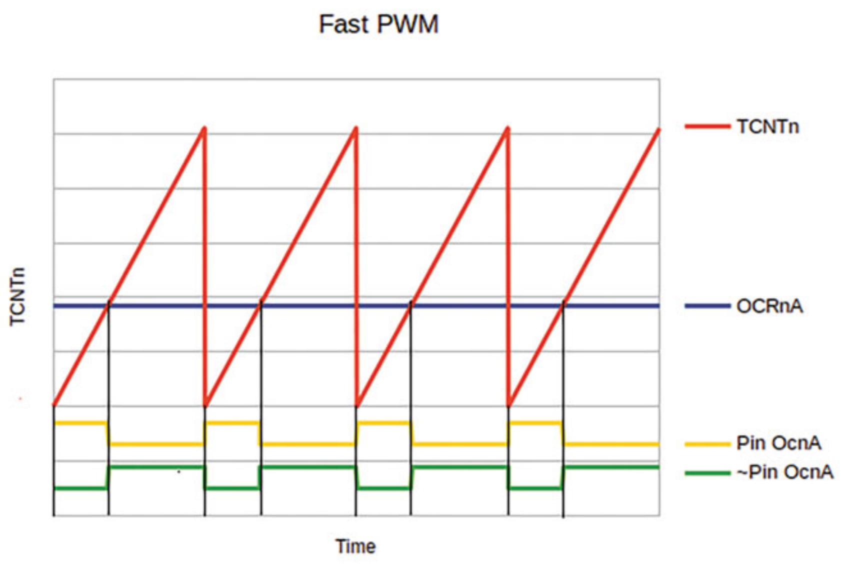


Figure 8-3 Eight-bit Fast PWM with TOP at 255

timers to run in eight-bit mode with a prescaler of 64 and a `TOP` value of 255. Timer 0 is configured to run in Fast PWM mode, while the other two are configured in Phase Correct PWM mode.

Timers 0 and 2 have two Fast PWM modes, modes 3 and 7. The differences are as follows:

- In mode 3, `TOP` is always 255—this mode is used by the Arduino.
- In mode 7, `TOP` is always the value in OCRnA.

On Arduino boards, Timer 0 runs Fast PWM on Arduino pins `D5` and `D6`, which corresponds to the AVR pins named `PD5` and `PD6`.

Timer 1 has five different Fast PWM modes available for use, modes 5, 6, and 7, plus modes 14 and 15. The differences are as follows:

- In mode 5, `TOP` is always 255, and the count in `TCNT1` is always eight bits—this mode is used by the Arduino.
- In mode 6, `TOP` is always 511, and the count in `TCNT1` is always nine bits.
- In mode 7, `TOP` is always 1,023, and the count in `TCNT1` is always ten bits.
- In mode 14, `TOP` is always the value in register `ICR1`, the Input Capture Unit register.
- In mode 15, `TOP` is always the value in register `OCR1A`.

In these Fast PWM timer modes

- There are $TOP + 1$ cycles.
- When TCNTn is zero, the appropriate PWM pin goes `HIGH`.
- When TCNTn equals OCRnA or OCRnB, the corresponding pin, OCnA or OCnB, will go `LOW`.

The ATmega328P can be configured to invert the PWM output pins, OCnA and OCnB, so that they go `LOW` instead of `HIGH` and `HIGH` instead of `LOW`.

Warning
It should be obvious when running in a Fast PWM mode where `TOP` is defined by OCRnA that the value in OCRnB must be less than OCRnA, or the PWM will not work on that pin.

Note
In modes where OCRnA defines `TOP`, you may be somewhat restricted in what you can do with pin OCnA.

The Arduino configures Timer 0 to run in Fast PWM mode with a prescaler of 64. The other two times/counters are configured in Phase Correct PWM mode.

For Fast PWM mode, the PWM frequency is calculated as

$$F\_CPU/(prescaler * (TOP + 1))$$

This works out on the Arduino boards as

$$\begin{aligned} PWM\ Frequency &= 16{,}000{,}000/(64 * (255 + 1)) \\ &= 16{,}000{,}000/(64 * 256) \\ &= 16{,}000{,}000/16,384 \\ &= 976.5625 \text{ Hz} \end{aligned}$$

This is the frequency which Timer 0 runs. Table 8-12 should save you the effort of working out the Fast PWM frequencies and periods for any given prescaler.

Figure 8-3 shows the details of Fast PWM with `TOP` fixed at 255, which is mode 3 for Timers 0 and 2 or mode 5 for Timer 1.

Note
In Figure 8-3, the value in OCRnA is constant as per the Arduino initialization code. It can be changed in code or in an interrupt handler, to vary the duty cycle of the PWM waveform, but this is not used on the Arduino.

In the following descriptions, everything that applies to OCRnA and OCnA applies equally to OCRnB and OCnB, but the latter are not shown in Figure 8-3 to avoid clutter:

- The jagged line in Figure 8-3 is the value in TCNTn as it rises from zero (`BOTTOM`) to `TOP`, which, in the case of an Arduino board, is set to 255, although this can be changed as per the data sheet. After 255, the value drops to zero on the next timer clock pulse.
- The horizontal line is the constant value in OCRnA; in this example, it is 255 as per Arduino initialization code. The value in OCRnA can be changed, usually by the interrupt handler, if the duty cycle is required to be varied. Changes to the value in OCRnA are double buffered and applied at `BOTTOM` when TCNTn has just become zero.

- The two square wave lines at the bottom represent the non-inverting and the inverting waveform generated on pin OCnA.
- If you look at the line for pin OCnA, the non-inverting output will be `HIGH` when TCNTn is zero and will then go `LOW` when TCNTn equals OCRnA. At the same time as the match is made, the timer's Timer Compare Match interrupt flag is set.
- The bottom square wave of the two shown is the inverting output line, and this is simply the opposite to pin OCnA.
- The timer's overflow flag will be set when TCNTn reaches `BOTTOM`.

  The data sheet advises that

> This high frequency makes the Fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost. It is not advised for motors as they much prefer Phase Correct PWM.

So, what happens in Fast PWM? Many things:

- When TCNTn equals `TOP`, the Timer n Overflow Interrupt bit TOVn is set in register TIFRn, and if enabled, this interrupt can be used to update the value in the OCRnA and/or OCRnB registers to change the duty cycle of the PWM waveform. In this timer mode, the OCRnA and OCRnB registers are double buffered, and the value written by your sketch, to these registers, will not be copied into the register until TCNTn resets to zero (`BOTTOM`), only then does the register value change.

  This bit is not cleared unless your sketch clears it or if the interrupt handler is enabled by setting bit TOIEn in register TIMSKn in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- When TCNTn reaches OCRnA, then bit OCFnA is set in register TIFRn. This bit is not cleared unless your sketch clears it, or the interrupt handler is enabled by setting bit OCIEnA in register TIMSKn in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- Pin OCnA will perform an action. The action depends on the values in the COMnA1:0 bits, as described in Table 8-13, when OCRnA matches TCNTn.
- When TCNTn reaches OCRnB, then bit OCFnB is set in register TIFRn. This bit is not cleared unless your sketch clears it, or the interrupt handler is enabled by setting bit OCIEnB in register TIMSKn in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- Pin OCnB will perform an action. The action depends on the values in the COMnB1:0 bits, as described in Table 8-14, when OCRnB matches TCNTn.

Table 8-15 shows the settings required to put the timers into Fast PWM mode.

Note that bits WGMn0 and WGMn1 are found in register TCCRnA, bit WGMn2 is found in TCCRnB, and, for Timer 1 only, `WGM13` is also found in `TCCR1B`.

You should be aware that because the PWM pins always go `HIGH`, at least in non-inverting mode, at `BOTTOM`, then they are always `HIGH` at the start of each cycle and `LOW` at the end, no matter what value is used for `TOP`. Sometimes, this isn't suitable—motors apparently don't like this—and for that, you would use Phase Correct PWM instead.

As mentioned, Fast PWM is set up by the Arduino `init()` function for every sketch, on Timers 0 and 2. The sketch in Listings 8-11, 8-12, and 8-13 is effectively what the `init()` function does to set up Timer 0, but without the overflow interrupt handler that updates `millis()` and `micros()` and with a much slower prescaler.

Table 8-13 COMnA1:0 settings in Fast PWM mode

COMnA1:0	Description
0b00	No effect on pin OCnA
0b01	Timers 0 and 2, Fast PWM mode 3: No effect on pin OCnA
0b01	Timers 0 and 2, Fast PWM mode 7: OCnA will toggle when TCNTn matches OCRnA
0b01	Timer 1, in modes 5, 6, or 7: `OC1A` is unaffected
0b01	Timer 1, Fast PWM mode 14: `OC1A` will toggle when `TCNT1` matches `ICR1`
0b01	Timer 1, Fast PWM mode 15: `OC1A` will toggle when `TCNT1` matches `OCR1A`
0b10	Pin OCnA is `LOW` on match with OCRnA and `HIGH` at `BOTTOM`. This is non-inverting mode
0b11	Pin OCnA is `HIGH` on match with OCRnA and `LOW` at `BOTTOM`. This is inverting mode

Table 8-14 COMnB1:0 settings in Fast PWM mode

COMnB1:0	Description
0b00	No effect on pin OCnB
0b01	Timers 0 and 2, Fast PWM modes 3 and 7: Reserved. Do not use
0b01	For Timer 1, Fast PWM modes 5, 6, 7, 14, and 15: `OC1B` is unaffected
0b10	Pin OCnB is `LOW` on match with OCRnB and `HIGH` at `BOTTOM`. This is non-inverting mode
0b11	Pin OCnB is `HIGH` on match with OCRnB and `LOW` at `BOTTOM`. This is inverting mode

Table 8-15 Fast PWM mode settings

Timer	Mode	TTCRnB Bits	Value
0	3	WGM02:00	0b011
0	7	WGM02:00	0b111
1	5	WGM13:10	0b0101
1	6	WGM13:10	0b0110
1	7	WGM13:10	0b0111
1	14	WGM13:10	0b1110
1	15	WGM13:10	0b1111
2	3	WGM22:20	0b011
2	7	WGM22:20	0b111

You might need to adjust the `delay()` at the end of `loop()` if you can't see the fade up and down of the LEDs.

Listing 8-11 is a number of macros used to reduce the amount of bit shifting in the main code. It simply creates a definition for the two channels on Timer 0, for both the `HIGH` and `LOW` states. These are used when the PWM value is 255 or 0.

Listing 8-11 Fast PWM sketch, defines

```
//=========================================================
// This sketch uses the Timer/Counter 0 in Fast PWM mode
// to fade down an LED on pin D5 while fading up an LED
// on D6. The prescaler is 1,024.
//=========================================================

#define PWM_A_LOW (~(1 << PORTD5))
#define PWM_A_HIGH ((1 << PORTD5))
#define PWM_B_LOW (~(1 << PORTD6))
#define PWM_B_HIGH ((1 << PORTD6))
```

The preceding definitions could have been declared as, for example:

```
const uint8_t PWM_A_LOW = (~(1 << PORTD5));
```

This would have had the same effect and would have been more "type safe" in that the compiler would be able to catch errors where the code was attempting to execute instructions that cannot use `uint8_t` data types.

Listing 8-12 is the `setup()` function for the sketch.

Listing 8-12 Fast PWM sketch, setup() function

```
void setup() {
  // Set Timer 0 into Fast PWM mode 3
  // with TOP = 255 and OC0A and OC0B
  // toggling on match.
  TCCR0A = ((1 << WGM01) |                        (1)
            (1 << WGM00) |
            (1 << COM0A1) |
            (1 << COM0B1));

  // Timer 0 prescaler = 1,024.
  TCCR0B = ((1 << CS02) | (1 << CS00));            (2)

  // Need to set the pins to output.
  DDRD = ((1<< DDD5) | (1 << DDD6));               (3)
}
```

(1) This sets Timer 0 into Fast PWM mode, with `TOP` = 255. This is the same as the Arduino `init()` function usually does. In addition, pins `OC0A` and `OC0B` will toggle on a compare match. These equate to pins `PD5` and `PD6` or Arduino pins `D5` and `D6`.
(2) Unlike the Arduino's `init()` function, the prescaler here is set to divide by 1,024.
(3) This is simply `pinMode(5, OUTPUT);` and `pinMode(6, OUTPUT);` but both pins are set in one statement, not two.

The code in Listing 8-13 is that of the `loop()` function, which is where the hard work of fading the two LEDs up and down takes place. In the code, checks have to be made for the limits of the timer—0 and 255—as the data sheet advises against setting those values for a PWM waveform. The `analogWrite()` function also checks for these values and, if found, calls `digitalWrite()` to set the pin `LOW` or `HIGH` as appropriate and ignores the PWM for those values.

Listing 8-13 Fast PWM sketch, loop() function

```
void loop() {
  // Current PWM duty cycle and increments.
  static uint8_t aa = 0;                                   (1)
  static uint8_t bb = 255;
  uint8_t increment = 1;

  if ((aa != 0) && (aa != 255)) {                          (2)
    OCR0A = aa;
    OCR0B = bb;
  } else {                                                 (3)
    if (aa == 255) { // then bb == 0
      PORTD |= PWM_A_HIGH;
      PORTD |= PWM_B_LOW;
    } else { // then bb == 255
      PORTD |= PWM_A_LOW;
      PORTD |= PWM_B_HIGH;
    }
  }

  aa += increment;                                         (4)
  bb -= increment;

  // Even at 1,024 prescaling, it's too quick!
  delay(1);                                                (5)
}
```

(1) Declaring variables in a function as `static` means that they are initialized with the given value on the first call to the function, then on the next call to the function, they have the value from the previous call—they retain their value across function call and exit, in other words.
(2) If `aa` is not on a PWM limit (0 or 255), then `bb` is also not on a PWM limit; simply set the OCR0x registers with the values of `aa` and `bb`—we have two valid PWM values which we can use.
(3) If `aa` is on a limit, then `bb` must be on the other limit. In this case, we simply do what `analogWrite()` would do and effectively `digitalWrite()` a `LOW` or `HIGH` to the appropriate pins, depending on which limit `aa` and `bb` are on.
(4) The fade values in `aa` and `bb` are incremented and decremented by the current amount.
(5) As even the biggest prescaler runs the timer way too quickly, there is a small delay to allow the fading effect to be seen. Feel free to adjust this if necessary.

The code is effectively the same as `analogWrite()` which checks for values of 0 or 255, which the data sheet advises avoiding, and handles those separately. All other values get written to the

OCR0A and OCR0B registers to control the duty cycle of the square wave generated on pins D5 and D6. An LED (and 330 Ω resistor!) on these two pins should fade up on D5 and fade down on D6.

As variables aa and bb are unsigned eight-bit variables, they will roll over when increment is added or subtracted, which is why I only need to check explicitly for 0 or 255 in the code. Even if you change the increment value, it will still work.

If you run this on a normal Arduino, it might be flashing far too quickly to see properly, but one LED should be fading up while the other fades down. On a breadboard setup with the ATmega328P running at 8 MHz, the flashing is more obvious. You can change the delay() statement at the end of the loop() to slow things down a little, if necessary.

The LEDs will reset to their starting values when they reach the end of their fade—aa will toggle from full on to full off, while bb does the opposite. This means that D5 will start dark, fade up to full brightness, and then fade to dark again, while D6 does the opposite.

8.1.6.4 Phase Correct PWM Mode

Phase Correct PWM is called "dual slope" because the counter, TCNTn, counts from zero, BOTTOM, up to TOP, then counts back down again to BOTTOM. It takes twice as long to repeat the cycle and, thus, runs at half the frequency of the Fast PWM for the same prescaler value. The graph of the timer's value against time slopes upward, then back down again, and doesn't exhibit the sudden drop from TOP to BOTTOM that Fast PWM does. Figure 8-4 shows the slope of the count in TCNTn as it rises and falls back to zero.

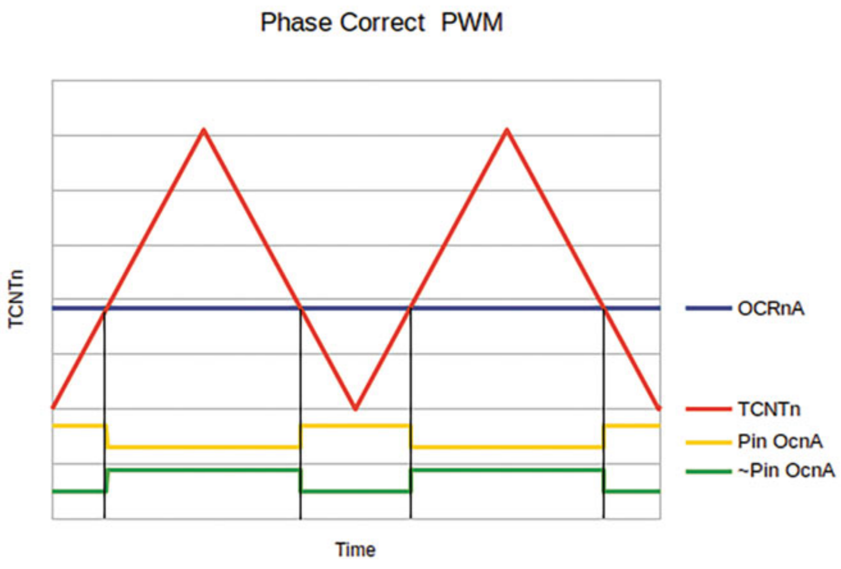


Figure 8-4 Eight-bit Phase Correct PWM with TOP at 255

The TOP value is determined by the mode and can be

- For all timers, the value 255—this is the Arduino default.
- For all timers, the value in register OCRnA.
- For Timer 1 only, the value 511.
- For Timer 1 only, the value 1,023.
- For Timer 1 only, the value in register ICR1.

The data sheet advises that

> When changing the TOP value the program must ensure that the new TOP value is higher or equal to the value of all of the Compare Registers. If the TOP value is lower than *any* of the Compare Registers, a compare match will never occur between the TCNT1 and the OCR1x.

It should be obvious, as with Fast PWM, that when running in a mode where TOP is defined by OCRnA or ICR1, the value in OCRnB must be less than OCRnA or the PWM will not work on that pin.

Also, in modes where OCRnA defines TOP, then you will be somewhat restricted in what you can do with pin OCnA.

On an Arduino, Timer 1 runs Phase Correct PWM mode 1 on Arduino pins D9 and D10, which corresponds to the AVR pins named PB1 and PB2. Timer 2 runs Phase Correct PWM mode 1 on Arduino pins D3 and D11, which corresponds to the AVR pins named PD3 and PB3.

Figure 8-4 shows the details for Phase Correct PWM with TOP fixed at 255—mode 1 on all three timers.

> **Note**
> In Figure 8-4, the value in OCRnA is a constant value, as per the Arduino. It *can* be changed, in code or in an interrupt handler, and so vary the duty cycle of the PWM waveform.

- The triangular line is the value in TCNTn as it rises from zero (BOTTOM) to TOP, which, in the case of an Arduino board, is set to 255, although this can be changed as per the data sheet. After TOP, the value counts back down to zero (BOTTOM) and then repeats as shown in the diagram. (Only two full cycles are shown here.)
- The horizontal line is the constant value in OCRnA, in this example. The value in OCRnA can be changed, usually by the interrupt handler, if the duty cycle is required to be varied. Changes to the value in OCRnA are double buffered and applied at TOP. This is when TCNTn is at 255 on Arduino boards.
- The two square wave lines at the bottom represent the non-inverting and the inverting waveform generated on pin OCnA.
- If you look at the line for pin OCnA, the non-inverting output, you should see that when TCNTn reaches OCRnA while counting upward, the appropriate output pin will go LOW. It stays LOW until TCNTn hits OCRnA again while counting downward whereupon it goes HIGH. At the same time as the match is made, the timer's Timer Compare Match interrupt flag is set.
- The bottom square wave of the two shown is the inverting output line, and this is simply the opposite to pin OCnA.
- The timer's overflow flag will be set when TCNTn reaches BOTTOM.

The data sheet advises that

Due to the symmetric feature of the dual-slope PWM modes, these modes are preferred for motor control applications.

Each timer has two pins upon which it can generate PWM waveforms. The Arduino initialization for each and every sketch sets the timers to all run in eight-bit mode with a prescaler of 64; however, only Timers 1 and 2 are set to run in Phase Correct PWM mode.

Timers 0 and 2 have two Phase Correct PWM modes, modes 1 and 5. The differences are as follows:

- In mode 1, `TOP` is always 255.
- In mode 5, `TOP` is always the value in OCRnA.

Timer 1 has five different Phase Correct PWM modes available for use, modes 1, 2, and 3, plus modes 10 and 11. The differences are as follows:

- In mode 1, `TOP` is always 255, and the count in `TCNT1` is always eight bits.
- In mode 2, `TOP` is always 511, and the count in `TCNT1` is always nine bits.
- In mode 3, `TOP` is always 1,023, and the count in `TCNT1` is always ten bits.
- In mode 10, `TOP` is always the value in register `ICR1`, the Input Capture Unit register.
- In mode 11, `TOP` is always the value in register `OCR1A`.

In these timer modes

- There are $TOP * 2$ cycles.
- When TCNTn equals OCRnA or OCRnB while counting upward, the appropriate PWM pin goes `LOW`.
- When TCNTn equals OCRnA or OCRnB when counting downward, the appropriate PWM pin will go `HIGH`.

The ATmega328P can be configured to invert the output PWM pins, so that they go `LOW` and `HIGH` in the opposite manner to that specified.

For Phase Correct PWM mode, the PWM frequency is calculated as

$$F\_CPU/(prescaler * (TOP * 2))$$

This works out on the Arduino boards as

$$\begin{aligned} PWM\ Frequency &= 16{,}000{,}000/(64 * (255 * 2)) \\ &= 16{,}000{,}000/(64 * 510) \\ &= 16{,}000{,}000/32{,}640 \\ &= 490.1960 \text{ Hz} \end{aligned}$$

This is the frequency that Timers 1 and 2 run. The period of the PWM frequency is, for Phase Correct PWM configured as per the Arduino code, 2,040 microseconds, or 2.04 milliseconds.

Table 8-12 should save you the effort of working out the Phase Correct PWM frequencies and periods for any given prescaler.

The following lists the various changes that occur in Phase Correct PWM:

- TCNTn is only ever at `TOP` or `BOTTOM` for one clock. It will hold the value of all other counter values twice, once while counting up, once when counting down.
- When TCNTn equals `BOTTOM`, the Timer n Overflow Interrupt bit TOVn is set in register TIFRn, and if enabled, this interrupt can be used to update the value in the OCRnA and/or OCRnB registers to change the duty cycle of the PWM waveform. In this timer mode, the OCRnA and OCRnB registers are again double buffered, and the value written by your sketch to these registers will not be copied into the register until TCNTn hits `TOP`, only then do the register values change.
  This interrupt flag bit is not cleared unless your sketch clears it, or if the interrupt handler is enabled by setting bit TOIEn in register TIMSKn in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- When TCNTn reaches OCRnA then bit OCFnA is set in register TIFRn. This bit is not cleared unless your sketch clears it or if the interrupt handler is enabled by setting bit OCIEnA in register TIMSKn in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- Pin OCnA will perform an action. The action depends on the values in the COMnA1:0 bits, as described in Table 8-16, when a compare match with TCNTn occurs.
- When TCNTn reaches OCRnB, then bit OCFnB is set in register TIFRn. This bit is not cleared unless your sketch clears it, or if the interrupt handler is enabled by setting bit OCIEnB in register TIMSKn in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- Pin OCnB will perform an action. The action depends on the values in the COMnB1:0 bits, as described in Table 8-17, when a compare match with TCNTn occurs.
- Only when TCNTn matches `TOP` are any changes made by the sketch to OCRnA or OCRnB applied.

Table 8-16 COMnA1:0 settings in Phase Correct PWM mode

COMnA1:0	Description
0b00	No effect on pin OCnA
0b01	Timers 0 and 2, P/C PWM mode 1: No effect on pin OCnA
0b01	Timers 0 and 2, P/C PWM mode 5: OCnA will toggle when TCNTn matches OCRnA
0b01	Timer 1, P/C PWM modes 1, 2, 3, and 10: `OC1A` is unaffected
0b01	Timer 1, P/C PWM mode 11: `OC1A` will toggle when `TCNT1` matches `OCR1A`
0b01	• Pin OCnA for Timers 0 and 2, in mode 1, is not affected • In mode 5 for those timers, OCNa will toggle when TCNTn matches OCRnA • For Timer 1 in mode 11, pin `OC1A` will toggle when `TCNT1` matches `OCR1A`. Pin `OC1B` will be unaffected • For Timer 1, in mode 9, pin `OC1A` will toggle when `TCNT1` matches `OCR1A`. Pin `OC1B` will be unaffected • For Timer 1, in modes 1, 2, 3, or 10 `OC1A` is unaffected
0b10	Pin OCnA is set `LOW` on count-up match with OCRnA and `HIGH` on count-down match. This is non-inverting mode
0b11	Pin OCnA is set `HIGH` on count-up match with OCRnA and `LOW` on count-down match. This is inverting mode

Table 8-17 COMnB1:0 settings in Phase Correct PWM mode

COMnB1:0	Description
0b00	No effect on pin OCnB
0b01	Timers 0 and 2, PF/C PWM modes 1 and 5: No effect on pin OCnB
0b01	Timer 1, P/C PWM modes 1, 2, 3, 10, and 11: `OC1B` is unaffected
0b10	Pin OCnA is set `LOW` on count-up match with OCRnA and `HIGH` on count-down match. This is non-inverting mode
0b11	Pin OCnA is set `HIGH` on count-up match with OCRnA and `LOW` on count-down match. This is inverting mode

Table 8-18 Phase Correct PWM mode settings

Timer	Mode	TTCRnB Bits	Value
0	1	WGM02:00	0b001
0	3	WGM02:00	0b011
1	1	WGM13:10	0b0001
1	2	WGM13:10	0b1010
1	3	WGM13:10	0b0011
1	10	WGM13:10	0b1010
1	11	WGM13:10	0b1011
2	1	WGM22:20	0b001
2	3	WGM22:20	0b011

Table 8-18 shows the settings required to put the timers into Phase Correct PWM mode.

Bits WGMn0 and WGMn1 are found in register TCCRnA, bit WGMn2 is found in TCCRnB, and, for Timer 1 only, `WGM13` is also found in `TCCR1B`.

As mentioned, Phase Correct PWM is set up by the Arduino `init()` function for every sketch, on Timers 1 and 2. The sketch in Listings 8-14, 8-15, and 8-16 is effectively what the `init()` function does to set up Timers 1 and 2 for use with the `analogWrite()` function, but in the example, I'm hijacking Timer 0 instead, just to be different!

Warning
Manipulating the PWM mode of Timer 0 in this manner is harmless on the ATmega328P, but for some microcontrollers—the ATmega168, for example—this will affect the accuracy of the `millis()` counter and all that which relies upon it.

You might need to adjust the `delay()` at the end of `loop()` if you can't see the fade up and down of the LEDs.

Listing 8-14 is a number of `#defines` used to reduce the amount of bit shifting in the main code. It simply creates a definition for the two channels on Timer 0, for both the `HIGH` and `LOW` states. These are used when the PWM value is 255 or 0.

Listing 8-14 Phase Correct PWM sketch, defines

```
//==========================================================
// This sketch uses the Timer/Counter 0 in Phase Correct
// PWM mode to fade down an LED on pin D6 while fading up
// an LED on D5. The prescaler is 1,024.
```

```
//========================================================

#define PWM_A_LOW (~(1 << PORTD5))
#define PWM_A_HIGH ((1 << PORTD5))
#define PWM_B_LOW (~(1 << PORTD6))
#define PWM_B_HIGH ((1 << PORTD6))
```

The preceding definitions could have been declared as, for example:

```
const uint8_t PWM_A_LOW = (~(1 << PORTD5));
```

This would have had the same effect operationally, but would have given the compiler more leeway in checking for type errors. Listing 8-15 is the `setup()` function for the demonstration sketch.

Listing 8-15 Phase Correct PWM sketch, setup() function

```
void setup() {
  // Set Timer 0 into Phase Correct PWM mode 1
  // with OC0A and OC0B toggling on match
  // and TOP = 255.
  TCCR0A = ((1 << WGM00) |                                  (1)
            (1 << COM0A1) |
            (1 << COM0B1));

  // Timer 0 prescaler = 1,024.
  TCCR0B = ((1 << CS02) | (1 << CS00));                     (2)

  // Need to set the pins to output.
  // Using AVR speak here.
  DDRD = ((1<< DDD5) | (1 << DDD6));                        (3)
}
```

(1) This sets Timer 0 into Phase Correct PWM mode, with `TOP` = 255. This is the same as the Arduino `init()` function usually does for the other two timers. In addition, pins `OC0A` and `OC0B` are configured to toggle on a compare match. These equate to pins `PD5` and `PD6` or Arduino pins `D5` and `D6`.
(2) Unlike the Arduino's `init()` function, the prescaler here is set to divide by 1,024.
(3) This is simply `pinMode(5, OUTPUT);` and `pinMode(6, OUTPUT);` but both pins are set in one statement, not two.

The code in Listing 8-16 is that of the `loop()` function, which is where the task of fading the two LEDs up and down lies. As we did with Fast PWM, checks have to be made for the limits of the timer—0 and 255—as the data sheet advises against setting those values for a PWM waveform. If found, the code effectively calls `digitalWrite()` to set the pin `LOW` or `HIGH` as appropriate and ignores the PWM for those values.

Listing 8-16 Phase Correct PWM sketch, setup() function

```
void loop() {
  // Current PWM duty cycle and increments.
  static uint8_t aa = 0;                                        (1)
  static uint8_t bb = 255;
  uint8_t increment = 1;

  if ((aa != 0) && (aa != 255)) {                               (2)
    OCR0A = aa;
    OCR0B = bb;
  } else {                                                      (3)
    if (aa == 255) { // then bb == 0
      PORTD |= PWM_A_HIGH;
      PORTD |= PWM_B_LOW;
    } else { // then bb == 255
      PORTD |= PWM_A_LOW;
      PORTD |= PWM_B_HIGH;
    }
  }

  aa += increment;                                              (4)
  ba -= increment;

  // Even at 1,024 prescaling, it's too quick!
  delay(1);                                                     (5)
}
```

(1) Declaring variables in a function as `static` means that they are initialized with the given value on the first call to the function, then on the next call to the function, they have the value from the previous call—they retain their value across function call and exit, in other words.
(2) If `aa` is not on a PWM limit (0 or 255), then `bb` is not either, so simply set the OCR0x registers with the values of `aa` and `bb`—we have two valid PWM values which we can use.
(3) If `aa` is on a limit, then `bb` must be on the other limit. In this case, we effectively `digitalWrite()` a `LOW` or `HIGH` to the appropriate pins, depending on which limit `aa` and `bb` are on.
(4) The fade values in `aa` and `bb` are incremented and decremented by the current amount.
(5) As even the biggest prescaler runs the timer way too quickly, there is a small delay to allow the fading effect to be seen. Feel free to adjust this if necessary.

Listing 8-16 is effectively the same as `analogWrite()` which checks for values of 0 or 255, which the data sheet advises avoiding, and handles those separately. All other values get written to the `OCR0A` and `OCR0B` registers to control the duty cycle of the square wave generated on pins `D5` and `D6`. An LED and 330 Ohm resistor on these two pins should fade up on `D5` and fade down on `D6`.

As variable `aa` and `bb` are unsigned eight-bit variables, they will roll over when `increment` is added or subtracted, which is why I only need to check explicitly for 0 or 255 in the code. Even if you change the increment value, it will still work.

If you run the preceding code on a normal Arduino, it might be flashing far too quickly to see properly, but one LED should be fading up while the other fades down. On a breadboard setup with the ATmega328P running at 8 MHz, the flashing is more obvious. You can change the `delay()` statement at the end of the `loop()` to slow things down a little, if necessary.

The LEDs will reset to their starting values when they reach the end of their fade—`aa` will toggle from full on to full off, while `bb` does the opposite. This means that the LED on pin `D5` will start dark, fade up to full brightness, and then fade down to dark again, while the LED attached to pin `D6` will do the opposite.

While the sketch is running, you should—hopefully—notice that the flickering of the LEDs is different with Phase Correct PWM to that of Fast PWM—this is noticeable when both sketches use the same `delay()` at the end of the loop.

8.1.6.5 Phase and Frequency Correct PWM Mode

Timer 1 has two additional PWM modes that the other timers do not have. This is Phase and Frequency Correct PWM and is mostly identical to Phase Correct PWM. The main difference is that the PWM generated is always symmetrical around the `TOP` value. This is because the `OCR1A` and/or `ICR1` registers are updated with new values at `BOTTOM`, unlike Phase Correct PWM, which updates these two registers at `TOP`. Because of the update at `BOTTOM`, the count upward from `BOTTOM` to `TOP` is always the same as the count downward from `TOP` to `BOTTOM`. In Phase Correct modes, changing the value at `TOP` means that the appropriate pin could be `HIGH` for a different time to that on the way back down.

The data sheet advises that

> When changing the `TOP` value the program must ensure that the new `TOP` value is higher or equal to the value of all of the Compare Registers. If the `TOP` value is lower than any of the Compare Registers, a compare match will never occur between the `TCNT1` and the OCR1x.

The data sheet also states that

> There is little difference between Phase Correct and Phase and Frequency Correct PMW modes, when using a fixed `TOP` value. However, if you need to vary the `TOP` value, then Phase and Frequency Correct mode is best as it is symmetrical about `TOP`.

What exactly is symmetrical about this mode? As the data sheet says, the PWM period begins and ends at `TOP`. This means that the falling slope of the waveform is determined by the old `TOP` value, while the rising slope that follows is determined by the new value in `TOP`. At the point where the two `TOP` values are different, the slopes will have a different length, and, thus, the duty cycle of the output waveform will be different.

Note
This is not a problem on the Arduino boards as none of the timers are configured to run in this PWM mode.

On Timer 1, the only one which has this PWM mode, the two Phase and Frequency Correct PWM modes are modes 8 and 9. The differences are

- In mode 8, `TOP` is always the value in register `ICR1`, the input capture register.
- In mode 9, `TOP` is always the value in `OCR1A`.

So, what happens in Phase and Frequency Correct PWM?

Table 8-19 COM1A1:0 settings in Phase and Frequency Correct PWM mode

COM1A1:0	Description
0b00	No effect on pin OC1A
0b01	Mode 8: No effect on pin OC1A
0b01	Mode 9: OC1A toggles on match with OCR1A
0b10	Pin OC1A is set `LOW` on count-up match and `HIGH` on count-down match. This is non-inverting mode
0b11	Pin OC1A is set `HIGH` on count-up match and `LOW` on count-down match. This is inverting mode

Table 8-20 COM1B1:0 settings in Phase and Frequency Correct PWM mode

COM1B1:0	Description
0b00	No effect on pin OC1B
0b01	No effect on pin OC1B
0b10	Pin OC1B is set `LOW` on count-up match and `HIGH` on count-down match. This is non-inverting mode
0b11	Pin OC1B is set `HIGH` on count-up match and `LOW` on count-down match. This is inverting mode

- `TCNT1` is only ever at `TOP` or `BOTTOM` for one clock. It will hold the value of all other counter values twice, once while counting up, once when counting down. This is identical to Phase Correct PWM.
- When `TCNT1` equals `BOTTOM`, any new value for `OCR1A` or `OCR1B` will be written into the appropriate register; also, the Timer 1 Overflow Interrupt bit `TOV1` is set in register `TIFR1`, and if enabled, this interrupt can be used to update the value in the `OCR1A` and/or `OCR1B` registers to change the duty cycle of the PWM waveform. In this timer mode, the `OCR1A` and `OCR1B` registers are again double buffered, and the value written by your sketch, to these registers, will not be copied into the register until `TCNT1` hits `BOTTOM`.
- When `TCNT1` reaches `TOP`, then the `OCF1A` or `ICF1` bit is set in register `TIFR1`. The bit set depends on which mode the timer is executing. In mode 8, `TIF1` will be set; in mode 9, it will be `OCF1A`.

  These two bits are not cleared unless your sketch clears them, or the appropriate interrupt handler is enabled by setting bit `ICIE1` in mode 8 or `OCIE1A` in mode 9, in register `TIMSK1`. In this case, assuming also that global interrupts are enabled, the bits will be automatically cleared. To clear the bits manually, in a sketch, you must write a 1 to them.
- Pin `OC1A` will perform an action. The action depends on the values in the `COM1A1:0` bits, as described in Table 8-19, when a compare match with `TCNT1` occurs.
- When `TCNT1` reaches `OCR1B`, then bit `OCF1B` is set in register `TIFR1`. This bit is not cleared unless your sketch clears it or if the interrupt handler is enabled by setting bit `OCIE1B` in register `TIMSK1` in which case, assuming also that global interrupts are enabled, the bit will be automatically cleared. To clear the bit manually, in a sketch, you must write a 1 to it.
- Pin `OC1B` will perform an action. The action depends on the values in the `COM1B1:0` bits, as described in Table 8-20, when `TCNT1` matches with `OCR1B`.
- When `TCNT1` matches `BOTTOM`, any changes made by the sketch to `OCR1A` or `OCR1B` are applied.

Table 8-21 shows the settings required to put the timers into Phase and Frequency Correct PWM mode.

Note that bits `WGM10` and `WGM11` are found in register `TCCR1A`, while bits `WGM12` and `WGM13` are found in `TCCR1B`.

Table 8-21 Phase and Frequency Correct PWM mode settings

Timer	Mode	TTCR1B Bits	Value
1	8	WGM13:00	0b1000
1	9	WGM13:20	0b1001

8.1.7 Too Much to Remember? Try AVRAssist

So many timers, so many modes, so many bits to be set or cleared, etc. Does it have to be this hard?

If you point your favorite browser at https://github.com/NormanDunbar/AVRAssist which is the AVRAssist GitHub page, you will come across my very easy-to-use AVR header files. These headers can be `#included` in your own source files, which will make life a lot easier when setting up timers and suchlike.

In use, you end up with something like Listing 8-17.

Listing 8-17 Setting up Timer 0 with AVRAssist

```
#include <timer0.h>

using namespace AVRAssist;


ISR(TIMER0_OVF_vect) {
  ...
}

void setup() {
  Timer0::initialise(MODE_FAST_PWM_255,
                     Timer0::CLK_PRESCALE_64,
                     Timer0::OCOA_TOGGLE | OCOB_TOGGLE,
                     INT_OVERFLOW);


  ...
}

void loop() {
  ...
}
```

The code in Listing 8-17 will set Timer 0 to have PWM on both pins `D5` and `D6` as per the Arduino `init()` function, with a prescaler of 64 and with an interrupt handler for the Timer 0 Overflow interrupt enabled, which, I think, is a little better to read and understand than a number of separate instructions listing various bit and register names, one after the other.

The code should set up Timer 0 in the mode specified and with all the settings that the Arduino sets up in the background for Timer 0 when you compile a sketch. However, the code will not compile and link when used in a sketch compiled with the Arduino IDE. This is because the IDE silently includes an interrupt handler for the Timer 0 Overflow interrupt, and that means that any code in a sketch, compiled by the IDE, cannot specify an interrupt handler for that same interrupt.

If you do try this, you will get a linker error telling you that there are two separate interrupt handlers for the overflow interrupt. Ask me how I know!

If you still wish to do something like this, you will need to compile your code outside of the Arduino IDE, and this means without using any of the Arduino Language. You will need to code in AVR C++ code instead. The code works perfectly in the PlatformIO environment—if, and only if, you remember to enable global interrupts—for example, and an interrupt handler for the overflow interrupt can be defined. But then, if you do it that way, you lose the `millis()` function and all that depend on it from the Arduino environment. Decisions, decisions!

Tip
While this was correct for the first edition of *Arduino Software Internals*, I subsequently wrote a replacement for PlatformIO to allow non-Arduino code to use `millis()` and `micros()`. You can find it at https://github.com/NormanDunbar/AVRmillis.

There are a few more details about AVRAssist in Appendix K.

8.2 Counting

Previously in this chapter, you learned, in some detail, all about the three timers in the ATmega328P. However, all you learned about were the *timer* modes. They can also be used as *counters*, and instead of being triggered by a regular clock signal, generated from the main system clock via a prescaler, the value in TCNTn can be incremented according to an external rising or falling edge on a pair of specific pins. This facility is only available on Timers 0 and 1. Timer 2 has other features, not available on those two.

8.2.1 Setting External Counting

Table 8-22 shows the configuration required to set the timers into counter mode.

Pin `T0` is physical pin 6 on the ATmega328P, Arduino pin `D4` or AVR pin `PD4`. Pin `T1` is physical pin 11 on the ATmega328P, Arduino pin `D5` or AVR pin `PD5`. These are the only two pins that can be used in this way. If your sketch is using the pins as counter stimulus, then they obviously cannot be used as normal I/O pins.

All the waveform generation modes are still available when clocking from the external pins, and usually, it would be expected that some form of clock signal, perhaps generated by the ubiquitous 555 timer, would be utilized to run the counter—if a regular count was required. On the other hand, it could be used to count the number of times a door was opened in a given time—it's down to the maker to decide.

Table 8-22 Setting timers into counting mode

CSn1:0 Bits	Value	Description
0b110	6	External clock on Tn pin, counts on a falling edge
0b111	7	External clock on Tn pin, counts on a rising edge

All the interrupts, compare matching on OCRnA or OCRnB, overflow bit setting, etc., all work as expected when running in counter mode.

8.2.2 Counter Example

The circuit in Figure 8-5 and the corresponding sketch in Listings 8-18 and 8-19 show a simple model of a door counter system—the Serial Monitor will record "door openings" each time the switch is pressed. In real use, the switch would be debounced and mounted in such a position as to record the door opening.

Warning
As I have not debounced the switch, it will also show how bouncy the particular switch I'm using happens to be!

Figure 8-5 shows the breadboard layout for this example. The circuit is very simple; the high side of SW1 is connected to 5 V from the Arduino. The low side of the switch is connected to R1, which is a 10 KΩ pull-down resistor to `GND`, and also to the Arduino pin `D5` which is Timer 1's `T1` pin. The LED and R2, which is 330 Ω, are connected between `D13` and `GND` in the normal manner.

Listings 8-18 and 8-19 make up the complete sketch for this example.

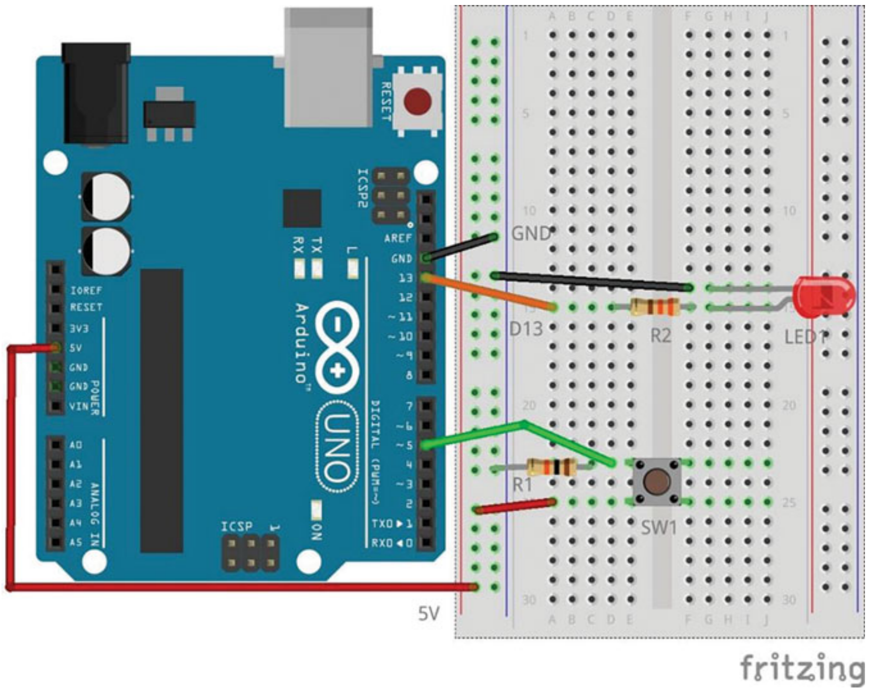


Figure 8-5 Door counter circuit

Listing 8-18 The door counter sketch setup()

```
void setup() {
  // Serial monitor is required.
  Serial.begin(9600);

  // Initialise Timer 1 to be triggered externally
  // by a rising edge on pin D5. The timer runs in normal
  // mode as we don't need waveforms.
  TCCR1A = 0; // Sets WGM11 and WGM10 to zero.

  // Disable interrupts on Timer 1.
  TIMSK1 = 0;

  // Clocked on a rising edge, and start the timer.
  TCCR1B = ((1 << CS12) | (1 << CS11) | (1 << CS10));

  // Make sure everything is reset.
  TCNT1 = 0;

  // T1=PD5=D5 is an input. PB5=D13 an Output.
  DDRB |= (1 << DDB5);
  DDRD |= (1 << DDD5);
}
```

Listing 8-18 sets up Timer 1 to be clocked externally on a rising edge on pin `T1` and with the timer's interrupts disabled. Listing 8-19 displays the count of the number of times the door was opened.

Listing 8-19 The door counter sketch loop()

```
void loop() {
  // Save the previous value of TCNT1.
  static uint16_t lastTCNT1 = 0;

  // What's the current value?
  uint16_t thisTCNT1 = TCNT1;

  if (thisTCNT1 != lastTCNT1) {
    Serial.print("TCNT1 = ");
    Serial.println(thisTCNT1);
    lastTCNT1 = thisTCNT1;
  }

  // Flash the LED and delay ...
  // ... to show that the timer still works.
  PINB |= (1 << PINB5);
  delay(2000);
}
```

Each time through the `loop()`, the current value of `TCNT1` is sent to the Serial Monitor if it changed since its previous value. Regardless of any changes, `loop()` always toggles the LED on Arduino pin `D13` and then delays for two seconds. The delay is simply there to show that the timer will still record switch presses during a delay which is tying up the main CPU.

When the LED is on or off, press the switch a few times quickly. When the delay is complete, the next value displayed in the Serial Monitor will show multiple hits have taken place and been recorded.

The following list shows the first ten results I obtained with a random switch from my spares box. These were all single presses, and the results appear quite good, not many bounces. I was of course suspicious! Surely cheap switches shouldn't be this good?

```
TCNT1 = 1
TCNT1 = 2
TCNT1 = 3
TCNT1 = 6
TCNT1 = 7
TCNT1 = 8
TCNT1 = 9
TCNT1 = 11
TCNT1 = 12
TCNT1 = 14
```

I tried a few more times with single presses, and it seems that I had picked the best switch in the world; I *mostly* only ever got a single increment. Is something wrong with the sketch? Or do I just have a really good switch?

Warning
After you read the following sentence, *don't* touch the 5 V wire to the `GND` side of the resistor, you will short out the power supply and might destroy your Arduino.

I decided to check and removed the wire from 5 V to the high side of the switch. I then touched it to the *switch side* of R1, the pull-down resistor. That's better; it bounced—a lot!

```
TCNT1 = 31
TCNT1 = 32
TCNT1 = 33
TCNT1 = 35
TCNT1 = 36
TCNT1 = 104
TCNT1 = 148
TCNT1 = 243
TCNT1 = 401
TCNT1 = 417
```

The first five presses used the button, and I rarely saw a bounce. For the remaining five, I used a wire to bypass the switch. Those touches bounced massively!

Tip
You can purchase *guaranteed non-bounce switches*, it appears. Getting hold of one and testing it against the sketch in Listings 8-18 and 8-19 might prove interesting.

In summary, as you can plainly see, setting a timer to run as a counter is far, far simpler than setting one to run as timer.

8.3 Input Capture Unit

Timer 1, as you know, is the only 16-bit timer on the ATmega328P. It is also the only timer which has an *input capture unit*. The data sheet advertises this feature as

> ... an Input Capture unit that can capture external events and give them a time-stamp indicating time of occurrence. The external signal indicating an event, or multiple events, can be applied via the ICP1 pin or alternatively, via the analog-comparator unit. The time-stamps can then be used to calculate frequency, duty-cycle, and other features of the signal applied. Alternatively the time-stamps can be used for creating a log of the events.

As you will see, this is not *quite* as useful as it sounds, but let's carry on with the data sheet, which goes on to state that

> When a change of the logic level (an event) occurs on the Input Capture pin (ICP1), alternatively on the Analog Comparator output (ACO), and this change confirms to the setting of the edge detector, a capture will be triggered. When a capture is triggered, the 16-bit value of the counter (TCNT1) is written to the Input Capture Register (ICR1). The Input Capture Flag (ICF1) is set at the same system clock as the TCNT1 value is copied into [the] ICR1 Register.
>
> If enabled (ICIE1 = 1), the Input Capture Flag generates an Input Capture interrupt. The ICF1 Flag is automatically cleared when the interrupt is executed. Alternatively the ICF1 Flag can be cleared by software by writing a logical one to its I/O bit location.

Sounds useful? Maybe! Think about Timer 1; it has a number of prescaler values we can use to slow down its counting frequency. The Arduino's main clock runs at 16 MHz, which at full speed, with a prescaler of 1, will cause Timer 1 to overflow after only 0.004096 seconds.

So, we need to slow it down; the biggest prescaler value for Timer 1 is 1,024. Now it overflows every 4.194304 seconds, which is still pretty much unusable as a timestamp, as intimated in the data sheet.

We could use a `uint16_t` variable in our code to store the timestamp values and increment another `uint16_t` variable as an overflow counter every time Timer 1 overflows—there's a handy interrupt that would take care of that—and use the overflow counter as the high 16 bits and the value from `ICR1` in the low 16 bits—that would work? Yes? No?

Maybe. The 16-bit count of overflows would increment every 4.194304 seconds and can store up to 65,536 of those before it too overflows. That's a total of 274,877.9069 seconds, which works out at 4,581.298449 minutes or 76 hours, 21 minutes, and 17.9069 seconds.

So, as long as all the events you wish to record, and timestamp, arrive within that time span, then having a spare 16-bit counter around to hold the overflow counts should work.

If 76 hours is still too short a timespan for all the expected events, would using a `uint32_t` variable to hold the overflow count and a separate `uint16_t` variable to hold the `TCNT1` value be any better? How long do we have to record all our events now?

We still overflow every 4.194304 seconds, but we can now accommodate 2^{32} of them. Doing the calculations, I think that works out as follows:

$$\begin{aligned} Overflow\ Time &= 4.194304 * 2^{32} \\ &= 1.8014398e10\ seconds \\ &= 300{,}239{,}975.1\ minutes \end{aligned}$$

$$= 5{,}003{,}999.586\ hours$$

$$= 208{,}499.9527\ days$$

$$= 570\ years\ 307\ days\ 11\ hours\ 35\ minutes\ 9.482\ seconds$$

That should be long enough surely?

I have yet to see any AVR or Arduino code that uses the input capture unit. For further details of using the unit, please refer to the data sheet; however, I can't leave you in suspense, so Listings 8-20 and 8-21 show a small sketch that demonstrates using the input capture unit.

Listing 8-20 Input capture unit, setup()

```
//============================================================
// This sketch uses the Timer/Counter 1 input capture
// unit to "timestamp" the arrival of a rising edge on
// Arduino pin D8, AVR pin ICP1/PB0, physical pin 14 on
// the ATmega328P.
//============================================================

void setup() {
  // Initialise the LED pin (D13) as OUTPUT and
  // pin D8/PB8/ICP1 as INPUT_PULLUP.
  // D13/PB5 as output.
  DDRB = (1 << DDB5);                                  (1)

  // D8/PB0/ICP1 as input pullup.
  PORTB = (1 << PORTB0);                               (2)

  // Initialise the ICU to no scaler, no noise cancel,
  // and rising edge detection.
  TCCR1A = 0; // Normal mode.                          (3)
  TCCR1B = ((1 << ICES1) | (1 << CS10));               (4)

  Serial.begin(9600);
}
```

(1) This sets all of PORTB as input, with pin PB5, a.k.a D13, a.k.a LED_BUILTIN as output. I'm using the built-in LED as a flag to show that something is happening while we wait for an event to happen.

(2) Writing to the PORT pin for an input pin enables the internal pull-up resistor. This pin will be held HIGH unless pulled to ground externally.

(3) Timer 1 is in normal mode.

(4) ICES1 enables input capture on a rising edge; CS10 enables full speed timer clock based on the system clock.

Listing 8-21 Input capture unit, loop()

```
void loop() {
  // This is a polled wait, so it's inefficient! However
  // this loop() is not doing much else.
  //
  // Wait for ICF1 to be set in TIFR1 then send
  // ICR1 to the serial port. Toggle the built
  // in LED while we wait. (Quite quickly!)
  while (!(TIFR1 & (1 << ICF1))) {                   (1)
    PINB |= (1 << PINB5);
    delay(100);
  }

  // Clear the ICF1 bit (no interrupts running you see)
  TIFR1 &= (1 << ICF1);                              (2)

  // Grab the timestamp.
  Serial.println(ICR1);                              (3)
}
```

(1) Wait here, just toggling the LED, until `ICF1` goes `HIGH` to signal an event. That event copies the `TCNT1` value into the `ICR1` register while the timer carries on counting. This is a tight loop with only a small delay, and so the LED will blink quite rapidly. In a real project, the LED would probably be controlled by a timer.
(2) Because we are not running input capture interrupts, we must write a 1 to the `ICF1` bit to clear it for the next event.
(3) Grab the event timestamp from `ICR1` and write it to the serial port.

Compile and upload the sketch as shown and plug a jumper wire into pin `D8` on your Arduino board and plug the other end into the `GND` location on the header.

Now open the Serial Monitor, which will reset the board and start the sketch running. Nothing should appear on the monitor window.

Pull the jumper wire out of the `GND`, the pull-up resistor will start to pull the pin `HIGH`, and the ICU will register that as an event. The LED might flicker, briefly, and a couple of numbers will appear on the monitor output. I got these:

```
3802
4474
```

If you see only one number, well done! You managed to not cause any bounce when you removed the jumper wire.

Plug the jumper back into `GND` again; this will pull the pin `LOW`, and there should be no more numbers. However, given the slowness of a human being, in contrast with an ATmega328P running at 16 MHz, the chances are slim. You will see a few more numbers hitting the Serial Monitor output. Mine read

```
60412
60488
17441
19431
```

Note how the numbers count up and then appear lower, but counting up again? That's a demonstration of my point about the period available for grabbing all your events and timestamping them.

Note
Your numbers might be bigger than mine or roll over faster. My test bed for this experiment was a breadboard *NormDuino*—see Appendix H—running on an internal 8 MHz oscillator to free up the two pins normally used for the 16 MHz crystal.

9 ATmega328P Hardware—AC, ADC, and USART

This chapter continues our look at the various hardware components of the ATmega328P. Some of them are not visible ("surfaced") in the Arduino IDE or Language, so they may at first appear new to you, the Analog Comparator, for example.

The information in this and Chapter 8 should link up with what you have already seen in Chapters 2, 3, and 4, which covered the compilation process and the Arduino Language and classes.

9.1 Analog Comparator (AC)

The ATmega328P has a built-in device, called the Analog Comparator, which compares the input voltage on pin `AIN0`, the positive input, and pin `AIN1`, the negative input.

If the voltage on the positive input, `AIN0`, is higher than the voltage on the negative input, `AIN1`, the Analog Comparator Output bit, `ACO` (that's a letter OH and not a digit zero), in the Analog Comparator Control and Status Register (`ACSR`), will be set to 1.

If the voltage on the positive input, `AIN0`, is equal or lower than the voltage on the negative input, `AIN1`, then `ACO` is cleared to 0.

In addition to setting the ACO bit, the comparator can also be set to trigger

- The Timer 1 Input Capture function.
- A dedicated interrupt, exclusive to the Analog Comparator. The interrupt can be configured to trigger when
  - The comparator output, `ACO`, is rising—from `LOW` to `HIGH`.
  - The comparator output, `ACO`, is falling—from `HIGH` to `LOW`.
  - The comparator output, `ACO`, is toggling—from `LOW` to `HIGH` or from `HIGH` to `LOW`.

Section 8.3 deals with Timer 1's Input Capture Unit.

Digital pins `D6` and `D7` are the Arduino's comparator input pins, with `D6` being `AIN0`, the positive input, and `D7` being `AIN1`, the negative input. `D6` is therefore the reference voltage to which the voltage on `D7` can be compared. However, `AIN1` can optionally be configured to be any one of the ADC input pins, `A0` through `A7` (if you have `A6` and `A7`, of course!), as will be explained.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6_9

The Arduino Language does not facilitate easy access to the Analog Comparator. You have to do it the hard way yourself, by manipulating the individual register bits—there's no easy option here I'm afraid!

9.1.1 Reference Voltage

The reference voltage, on the positive input, `AIN0`, is used as the basis for the comparator. The voltage that is being sampled or compared will be checked against the voltage on the comparator's positive input. The positive input can be configured to be either:

- An internally generated 1.1 V known as the bandgap reference voltage
- An external voltage supplied on the `AIN0` pin, also known as `D6`

9.1.2 Sampled Voltage

The voltage being compared against the reference voltage can be either

- The `AIN1` pin, also known as `D7`
- Any one of the ADC input pins, which on the Arduino are `A0` through `A5` plus `A6` and `A7` if your board has the surface mount version of the ATmega328P and the manufacturer has routed those two extra pins to a header somewhere

9.1.3 Digital Input

The pins `D6` and `D7` cannot be used as normal I/O pins when being used by the comparator. To this end, they should have their I/O buffers disabled—to save wasting power. This is done by setting bits `AIN0D` and/or `AIN1D` in register `DIDR1`, the Digital Input Disable Register 1.

Bit 0 in the `DIDR1` register is `AIN0D`, bit 1 is `AIN1D`, and bits 2–7 are reserved and should not be written. They will always be zero when read.

When either the `AIN0D` or `AIN1D` bit is set to 1, the digital I/O on the `AIN0`/`D6` or `AIN1`/`D7` pin is disabled. When the pins are disabled in this manner, reading the `PIND` register (the input register that these two pins belong to) will always return a value of zero for whichever of the two pins has been disabled.

The ATmega328P data sheet has this to say:

> When an analog signal is applied to the AIN0/1 pin and the digital input from this pin is not needed, this bit [`AIN0D` or `AIN1D`] should be written logic one to reduce power consumption in the digital input buffer.

The following sections summarize the steps necessary to use the comparator.

9.1.4 Enable the Analog Comparator

- Write a 0 to bit `ACIE` in the Analog Comparator Control and Status Register, `ACSR`. This disables the Analog Comparator Interrupt Enable as an interrupt can occur when the `ACD` bit is changed.
- Write a 0 to bit `ACD` in ACSR.

These steps enabled the comparator and disabled interrupts from it. If interrupts from the comparator are required, then this can be configured at a later step.

9.1.5 Select Reference Voltage Source

The reference voltage applied to the positive input to the comparator can be either

- The internal bandgap reference voltage
- An external voltage on pin `AIN0`

Only one of these can be selected.

9.1.5.1 External Reference

To use the external reference voltage on pin `AIN0`, you must

- Write a 1 to bit `AIN0D` to disable the I/O facilities on pin `D6`.
- Write a 0 to bit `ACBG` in the `ACSR` register.

9.1.5.2 Internal Bandgap Reference

To use the internal reference voltage instead of `AIN0`, you must

- Write a 1 to bit `ACBG` in register `ACSR`.

9.1.6 Select Sampled Voltage Source Pin

The voltage to be compared with that on the comparator's positive input can be either

- Pin `AIN1`
- One of the pins `A0` through `A7`

9.1.6.1 Sample Voltage on Pin AIN1

To compare an external reference voltage on pin `AIN1`/`D7`, you must

- Write a 1 to bit `AIN1D`, in register `DIDR1`, to disable the I/O facilities on pin `D7`.

Then, either

- Write a 0 to bit `ACME` in register `ADCSRB`.
- Write a 1 to bit `ACME` in register `ADCSRB` and write a 1 to bit `ADEN` in register `ADCSRA`.

You therefore have two choices to set up the system to use `AIN1`.

9.1.6.2 Sample Voltage on Pins A0–A7

– Write a 0 to bit `PRADC` in register `PRR` to power on the ADC.
– Write a 0 to bit `ADEN` in register `ADCSRA` to disable the ADC from using the ADC multiplexer.
– Write a 1 to bit `ACME` in register `ADCSRB` to allow the comparator to use the ADC multiplexer.

Table 9-1 Analog Comparator negative input summary

ACME	ADEN	MUX2:0	Negative Input
0	N/A	N/A	AIN1/D7
1	1	N/A	AIN1/D7
1	0	0b000	ADC0/A0
1	0	0b001	ADC1/A1
1	0	0b010	ADC2/A2
1	0	0b011	ADC3/A3
1	0	0b100	ADC4/A4
1	0	0b101	ADC5/A5
1	0	0b110	ADC6/A6
1	0	0b111	ADC7/A7

– Write the pin number, 0 to 7, to bits `MUX2:0` in the `ADMUX` register to select the desired input pin from `A0` through `A7`.

9.1.6.3 Sampled Voltage Summary

Table 9-1 summarizes the pin settings for all possible negative input settings.

9.1.7 Comparator Outputs

So far, we have looked at the numerous ways that we can set up the two inputs to the comparator. How then do we get an output from it? The Analog Comparator Control and Status Register (`ACSR`) is where we need to be looking. We have already seen that bits `ACD` and `ACBG` disable/enable the comparator and select the reference voltage; the other bits are as follows:

- `ACO` (Analog Comparator Output): This bit is connected to the comparator output and is synchronized with the comparator output when it changes. This synchronization takes one to two clock cycles to settle and delays changing this bit when the comparator changes.
- `ACI` (Analog Comparator Interrupt Flag): This bit is set when a comparator output triggers according to the mode set in bits `ACIS1:0` (see the following text). If global interrupts are enabled and the `ACIE` bit set, then the appropriate ISR will be executed and this bit cleared by hardware. Otherwise, it can be cleared by the writing of a 1 in the normal back-to-front AVR manner!
- `ACIE` (Analog Comparator Interrupt Enable): This bit enables or disables the firing of an interrupt when the comparator output takes on a certain state as defined by bits `ACIS1:0`, which are described as follows. When the bit is zero, no interrupts will fire.
- `ACIC` (Analog Comparator Input Capture Enable): Writing this bit to 1 will enable Timer 1's input capture function to be triggered by the Analog Comparator. In addition to setting this bit, bit `ICIE1` in register `TIMSK1` must also be set to enable the Timer 1 Input Capture Interrupt.
- `ACIS1:0` (Analog Comparator Interrupt Mode Select): These two bits must always be changed after disabling the comparator's `ACIE` bit by writing it to zero. These two bits select the interrupt mode for the comparator and determine when the interrupt will be fired. The possible values are as follows:
  – 0b00: The interrupt fires when the comparator output toggles.
  – 0b01: Reserved—do not use.
  – 0b10: Interrupt fired on `ACO` falling edge—when the `ACO` changes from one to zero.
  – 0b11: Interrupt fired on `ACO` rising edge—when the `ACO` changes from zero to one.

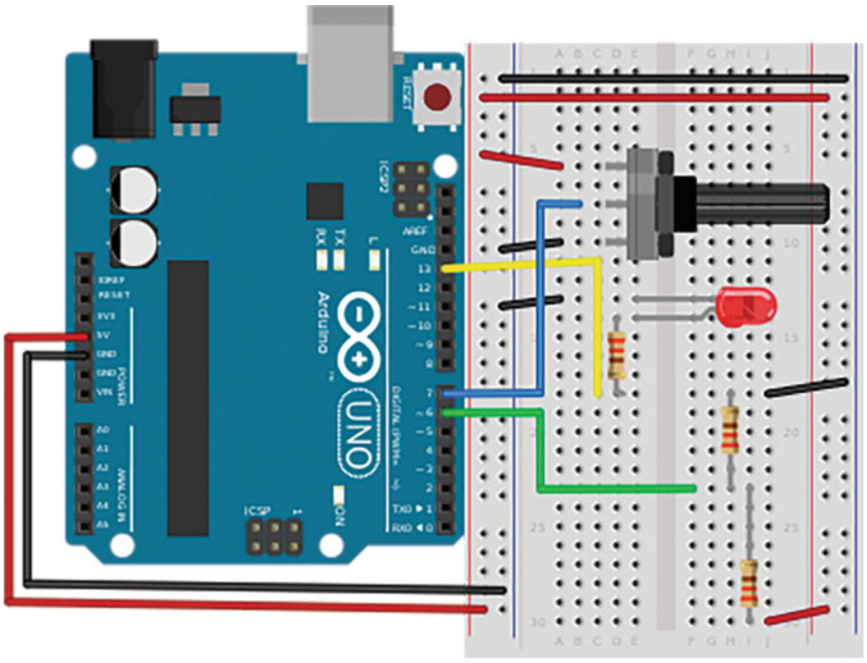


Figure 9-1 Analog Comparator circuit

9.1.8 Comparator Example

The sketch in Listing 9-1 shows the use of the Analog Comparator to turn an LED on or off depending on whether the voltage at `D6` is higher or lower than the voltage at `D7`.

In the circuit I used for this experiment, I created a voltage divider using two resistors; I used the same value, but it isn't necessary. The output from this was fed into `D6` and used as the reference voltage.[1] I connected one end to the Arduino `VCC` (5 V) and the other to Arduino Ground.

I also wired up a potentiometer to Arduino `VCC` and `GND` as well and fed the middle pin of the potentiometer to pin `D7`. By turning the potentiometer, I was able to vary the voltage on pin `D7`, and the LED turned off or on depending on the voltage on `D7`.

You can see the breadboard circuit in Figure 9-1.

When the project was running, turning the potentiometer varied the voltage on `D7`, and if the reference voltage on `D6`, which was 2.5 V due to my voltage divider, was higher than the variable voltage on `D7`, the LED turned on; otherwise, it turned off.

I've added a bigger LED to pin `D13` to better show the effect, but you don't have to do this. Just remember, there are two LEDs on `D13`, and you need to keep the current below 20 mA.

The LED I'm using has a voltage drop of 1.8 V, so subtract that from the 5 V supply from the Arduino to get 3.2 V. The resistor is 330 Ω, and so, by division, we get 3.2/330 which gives 9.69 mA.

The Arduino has an absolute maximum of 40 mA per pin (but with other restrictions—see Appendix C for details—but 20 mA is preferred, so there should be no problems with this resistor value.

[1] I could have simply taken a jumper wire from the Arduino's 3.3 V output in the headers and fed that to `D6`.

Listing 9-1 Analog Comparator sketch

```
//==========================================================
// This sketch uses the analog comparator with pin D6
// as the reference voltage and D7 as the voltage to be
// compared with D6. When D6 is higher than D7 then the
// LED will light. When D6 is lower than D7, the LED goes
// out. So, not a blink sketch this time!
//==========================================================
void setupComparator() {                                    (1)
  // Disable AC interrupts.
  ACSR &= ~(1 << ACIE);

  // Enable AC by disabling the AC Disable bit!
  ACSR &= ~(1 << ACD);

  // Disable digital I/O on D6 and D7.
  DIDR1 |= ((1 << AIN0D) | (1 << AIN1D));

  // D6 will be the reference voltage.
  ACSR &= ~(1 << ACBG);

  // D7 to compare with D6.
  ADCSRB &= ~(1 << ACME);

  // Fire AC interrupt on ACO toggle.
  ACSR |= ((0 << ACIS1) | (0 << ACIS0));

  // Enable AC Interrupt.
  ACSR |= (1 << ACIE);

  // Enable Global Interrupts.
  sei();
}

void setup() {
  // You can still use Arduino code as well -
  // but I'm not! D13 = output.
  DDRB |= (1 << DDB5);                                      (2)
  setupComparator();
}

void loop() {                                               (3)
  ; // Do nothing.
}

// Analog Comparator Interrupt Handler.Reads the ACSR
// register and sets the LED to the state of the ACO bit.
```

```
ISR(ANALOG_COMP_vect) {                                  (4)
  if (ACSR & (1 << ACO)) {
    PORTB |= (1 << PORTB5); // LED HIGH);
  } else {
    PORTB &= ~(1 << PORTB5); // LED LOW);
  }
}
// This function sets up the comparator to fire an interrupt
// each time the ACO bit toggles. It uses D6 as the reference
// voltage and D7 as the voltage to be compared.
```

(1) The `setupComparator()` function initializes the Analog Comparator as described in the text and the code comments.
(2) This is just a very short `pinMode(LED_BUILTIN, OUTPUT)` call.
(3) The `loop()` function does nothing. Everything happens in the ISR.
(4) The ISR fires any time that the Analog Comparator output toggles. It sets the LED to on, if the `ACO` bit is set; otherwise, it turns the LED off.

You should note that when the comparator output is set or cleared, it remains that way until a change is necessary. The interrupt is only called when the `ACO` bit toggles from set to clear or from clear to set.

It is effectively like a light switch; it's on or off until it changes.

9.2 Analog-to-Digital Converter (ADC)

The ATmega328P has a single, ten-bit Analog-to-Digital Converter which has up to nine separate inputs, depending on which ATmega328P your Arduino is using. The Dual In-Line Package (DIP) with 28 pins has seven inputs, while the surface mount version has all nine. On Arduino boards, these are pins `A0`–`A5` (or `A0`–`A7` for the surface mount version), plus the internal temperature sensor input. The ADC inputs can also be used by the Analog Comparator as described in Section 9.1 at the start of this chapter.

As mentioned, the ADC has ten-bit resolution, which means that it can return a value between 0 and $2^{10} - 1$, or 1,023. The value is indicative of the voltage on the `AREF` or `AVCC` pin, depending on the configuration, as compared with whichever ADC input has been selected. Only one of the available ADC inputs can be used at a time. A result of 0 represents `GND`, and 1,023 represents the reference voltage. The reference voltage can be configured to one of the following three options:

- The 1.1 V internal bandgap reference voltage
- The voltage on the `AVCC` pin
- An external voltage on the `AREF` pin, which must not exceed `VCC`

The voltage on the ADC input pin is therefore

$$(Reference\ Voltage/1{,}024) * ADC\ Result$$

The ADC works by using a capacitor to sample and hold the input voltage, which is useful if the voltage changes during the time that the ADC is still calculating the result as it ensures that the sampled voltage remains stable throughout the calculation. The ADC uses successive approximation to calculate the ten bits of the result.

In order to further improve the accuracy of the ADC, there is a special sleep mode, ADC Noise Reduction, which shuts down almost everything which is not the ADC, in order to stop all the "digital noise" that the microcontroller generates internally, so that the ADC can do its job in relative peace and quiet.

The ADC can be configured to take a single shot—just as the Arduino `analogRead()` function call does—where ADC conversions are started manually on request, or to run in free running mode where each completed conversion starts another conversion automatically, and only the very first conversion has to be manually started.

The ADC has an interrupt that may be configured to fire when the conversion is complete to avoid the need for your code to sit in a polling loop, waiting for the result. The interrupt is required in auto trigger mode, which can be triggered by one of many different sources—more on that subject to follow.

9.2.1 ADC Setup and Initiation

Assuming you are not using the Arduino's `analogRead()` function, then the following steps are required in order to take an ADC reading:

- Power up the ADC.
- Select a suitable prescaler to configure the ADC to run at a frequency within its required range of 50–200 KHz.
- Select a suitable reference voltage source.
- Decide on whether the result is to be left or right aligned.
- Select an input source.
- Disable digital input for the selected input pin.
- Enable interrupts, if required.
- Select single shot or auto trigger, and if auto trigger, choose a trigger.
- Enable the ADC and initiate a conversion.

Easy? Of course, it is!

9.2.1.1 Powering the ADC

Probably the easiest step of all, you simply have to write a 0 to the `PRADC` bit of the `PRR` register. If that bit is a 1, then the ADC is powered off.

```
PRR &= ~(1 << PRADC);
```

9.2.1.2 Selecting the Prescaler

The ADC runs most accurately at a frequency between 50 and 200 KHz. This frequency range is mandatory if you wish to get the full ten-bit result; however, if you require less than ten-bit resolution, you can run the ADC at higher frequencies. On an Arduino, the CPU is running at 16 MHz, which is a tad on the high side for the ADC. There is a prescaler for the ADC to bring the frequency down to

Table 9-2 ADC prescaler settings and frequencies

ADPS2:0	Prescaler	16 MHz	8 MHz
0b000	Divide by 1	16,000 KHz	8,000 KHz
0b001	Divide by 2	8,000 KHz	4,000 KHz
0b010	Divide by 4	4,000 KHz	2,000 KHz
0b011	Divide by 8	2,000 KHz	1,000 KHz
0b100	Divide by 16	1,000 KHz	500 KHz
0b101	Divide by 32	500 KHz	250 KHz
0b110	Divide by 64	250 KHz	**125 KHz**
0b111	Divide by 128	**125 KHz**	**63 KHz**

within the desired range. To set the prescaler, you must write a suitable value to the `ADPS2:0` bits in the ADC Control and Status Register A, `ADCSRA`.

Table 9-2 shows the required settings for the prescaler and the resulting ADC frequencies, in KHz, for 16 MHz and 8 MHz systems, with the acceptable frequencies highlighted.

On Arduinos, only the final 0b111 setting, divide by 128, brings the ADC frequency down into the desired range. You might be successful with the 0b110 setting, divide by 64, which results in a frequency of 250 KHz—but it's probably not advisable especially if you need all ten bits of resolution. I have seen it in Arduino code—once—in the source code for an Arduino-based oscilloscope.

As I run the odd occasional *NormDuino* board at 8 MHz—see Appendix H—I can use either of the previous two settings and possibly also the divide by 32 option, 0b101.

Assuming you are running an Arduino of some kind and are bypassing `analogRead()`, the following code would set the desired frequency:

```
ADCSRA = ((1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0));
```

This code *overwrites* all settings in the `ADCSRA` register. If you are compiling in the Arduino IDE, this is a good idea as it will overwrite the Arduino's default settings.

Regardless of the IDE, doing this means that you know exactly where you start in the ADC setup. All further settings in `ADCSRA` can be OR'd or ANDed in the usual manner. The examples that follow will all begin by clearing the appropriate register on its first use and adding in additional requirements on all subsequent uses.

9.2.1.3 Selecting the Reference Voltage Source

There are three separate and selectable voltage references which can be used by the ADC, although only one can be selected at any one time.

Warning
The data sheet advises against selecting either of the two internal options if there is an external voltage already applied to the `AREF` pin. Doing this will most likely brick your ATmega328P. It's best to check if your particular device has anything connected *before* changing the reference voltage source.

I've looked at the schematics for the Uno and the Duemilanove, and neither of those connects the `AREF` pin to any voltage. NormDuino also does not have any voltages on that pin.

Table 9-3 ADC reference voltage selection settings

REFS1:0	Description
0b00	Use the AREF pin as the reference voltage. The external voltage applied must not exceed VCC
0b01	Use the internal voltage on AVCC as the reference voltage. For best results, there should be a 100 nF capacitor between AREF and GND
0b10	Reserved
0b11	Use the internal 1.1 V bandgap voltage as the reference. Again, it's best to have a 100 nF capacitor between AREF and GND

The same cannot be said for a number of breadboard Arduino layouts to be found on the Internet, where they connect AVCC to VCC as required, but for some reason, also connect AREF to VCC, thus creating a time bomb, just waiting to happen, and for no good reason. Beware!

Even the Arduino Language delays setting the `analogReference()` setting until the time that `analogRead()` actually executes, and the source comes with a warning against having voltage applied to AREF.

The reference voltage is set by bits `REFS1:0` in the ADC Multiplexer Selection Register, also known as ADMUX. The permitted values for these two bits are shown in Table 9-3.

In the following example, we set the ADMUX register with an initial value for the reference voltage source, selecting the AVCC voltage, and will add to it as we progress:

```
ADMUX = ((0 << REFS1) | (1 << REFS0));
```

9.2.1.4 Left or Right Alignment?

The result of an ADC conversion is a value between 0 and 1,023. This represents the voltage on the ADC input pin (see the following text) as compared with the reference voltage, both with respect to GND.

There are two registers that must be read to obtain the result, ADCH and ADCL. ADCH holds the highest bits of the result, while ADCL holds the lowest bits. Reading these registers must be done in a specific order; ADCL must be read first, then ADCH.

Once you have read ADCL, the ADC is no longer permitted to write to either ADCL or ADCH until after you have completed reading ADCH. This blocking method ensures that when you read the result of a conversion, both registers are giving you the appropriate bits of the same conversion result.

The ADC generates its ten-bit result in the ADC data registers, ADCH and ADCL. By default, the result is presented right adjusted, but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX to a 1.

```
ADMUX |= (1 << ADLAR);
```

The default is for the result to be right aligned, which you can ensure by

```
ADMUX &= ~(1 << ADLAR);
```

The data sheet states that

If the result is left aligned and no more than 8-bit precision is required, it is sufficient to read ADCH.

Table 9-4 ADC left/right alignment options

ADLAR	Alignment	ADCH	ADCL
0	Right	xxxxxx98	76543210
1	Left	98765432	10xxxxxx

Table 9-5 ADC input voltage source settings

MUX3:0	Input Source	MUX3:0	Input Source
0b0000	ADC0/A0	0b1000	ADC8
0b0001	ADC1/A1	0b1001	Reserved
0b0010	ADC2/A2	0b1010	Reserved
0b0011	ADC3/A3	0b1011	Reserved
0b0100	ADC4/A4	0b1100	Reserved
0b0101	ADC5/A5	0b1101	Reserved
0b0110	ADC6/A6	0b1110	1.1 V Bandgap
0b0111	ADC7/A7	0b1111	0 V (GND)

So, what is the difference? The default, right alignment, returns `ADCL` with bits 9:8 of the result in bits 1:0 of `ADCH` and bits 7:0 of the result in bits 7:0 of `ADCL`.

In left alignment, bits 9:2 of the result are in bits 7:0 of `ADCH` and bits 1:0 of the result are in bits 7:6 of `ADCL`.

In Table 9-4, "x" means we don't care about this bit of the result as it is outside the resolution of the ADC.

Tip
In Arduino and AVR C++ code, you can read `ADCW` to get the correct result and not worry about reading `ADCH` and `ADCL` in the correct order.

9.2.1.5 Selecting an Input Source

It is now time to select an input source. This is the pin that will receive the voltage that we are comparing against the reference voltage source. There are four bits in register `ADMUX`, `MUX3:0`, which are used to select the input source. This gives up to 16 different sources, all of which are listed in Table 9-5; however, a number of the options are reserved and should not be used. `ADC8`, while listed in the data sheet as one of the reserved values, is actually an exception in that it *can* be safely used.

The data sheet lists a pair of warnings in relation to the use of pins while the ADC is being utilized:

> If `ADC3`, `ADC2`, `ADC1` or `ADC0` are used as Digital Outputs and not as ADC inputs, then they must not switch while an ADC conversion is in progress.

> If `ADC4` or `ADC5` are being used for I^{2}C/TWI (Two Wire Interface) purposes, then using that will affect only `ADC4` and `ADC5`, not the other ADC inputs.

Now, the final two entries in Table 9-5 are interesting, perhaps? I can only assume that they are there to enable some form of configuration perhaps. If you set `MUX3:0` to 0b1110, then the ADC always reads 227–229 (at least, mine does), which works out at 1.11–1.12 V. Using 0b1111 for `MUX3:0` returns zero on my devices, representing `GND`. Working on the assumption that this is indeed some form of configuration test, my ADC seems to be quite accurate—assuming, of course, that the 1.1 V bandgap reference voltage is itself 1.1 V.

Note
If your code decides to change the ADC input channel while a conversion is underway and has not yet completed, nothing will happen until the current conversion finishes.

9.2.1.6 Disable Digital Input

When using a pin as an ADC input, you are required to disable its digital input buffer by setting the appropriate bit for the pin in the Digital Input Disable Register 0 or `DIDR0`.

Only the pins corresponding to `ADC0` (Arduino `A0`) through `ADC5` (Arduino `A5`) have the ability to have their digital input buffers disabled. Pins `ADC6` and `ADC7`, the two new ones on surface mount versions of the ATmega328, and `ADC8` do not have digital input buffers, so cannot have them disabled.

To disable the digital input buffer for a pin, you must write a 1 to the appropriate ADCnD bit of the `DIDR0` register, where the "n" represents the ADCn pin number. To disable the digital input buffer for pin `ADC3`, Arduino pin `A3`, for example, you would code

```
DIDR0 |= (1 << ADC3D);
```

Don't forget to re-enable the buffer after the pin's use as an ADC input is finished; otherwise, it will always read `LOW`. This is done by writing a 0 to the appropriate bit in `DIDR0`, as follows:

```
DIDR0 &= ~(1 << ADC3D);
```

9.2.1.7 ADC Interrupt

There is a single interrupt attached to the ADC, the ADC interrupt, or ADC conversion complete interrupt, accessed through `ISR(ADC_vect)` in your code. This interrupt is enabled by setting the `ADIE`, ADC Interrupt Enable, bit in register `ADCSRA` as follows:

```
ADCSRA |= (1 << ADIE);
```

If global interrupts are also enabled, then the interrupt will fire every time that the ADC has completed a conversion and the result is available. Any time that the ADC conversion is complete, the `ADIF` bit in `ADCSRA` will be set and will remain set until either

- The interrupt handler executes, whereupon `ADIF` will be automatically cleared.
- The code writes a 1 to `ADIF` in the usual AVR manner.

If your code is not using the ADC interrupt, it should monitor `ADIF` and remember to clear it; otherwise, a further ADC conversion will not begin unless the ADC is in free running mode. This is described next.

The data sheet warns that we should

> Beware that if doing a Read-Modify-Write on `ADCSRA`, a pending interrupt can be disabled. This also applies if the `SBI` and `CBI` instructions are used.

Table 9-6 ADC auto-trigger sources

ADTS2:0	Trigger Source
0b000	Free running mode
0b001	Analog Comparator interrupt
0b010	External interrupt INT0
0b011	Timer 0, Compare Match A
0b100	Timer 0, Overflow
0b101	Timer 0, Compare Match B
0b110	Timer 1, Overflow
0b111	Timer 1, Input capture event

9.2.1.8 Single Shot or Auto Trigger?

In single-shot mode, a single conversion is carried out, and then the ADC stops working until the next request for a conversion. The conversion is started on demand by the code, and the ADC will make the reading and then stop.

In auto trigger mode, the conversion is triggered by an event or can be put in free running mode, which causes the ADC to continually make a new conversion as soon as the previous one has completed. This mode usually requires the ADC interrupt to be enabled to advise the code that a conversion has finished and that the result is available. Free running mode still requires the first conversion to be manually started, as described in the following text.

The various triggers available are set up in register `ADCSRB` by setting bits `ADTS2:0` as per Table 9-6.

These bits are only used if the `ADATE` bit in register `ADCSRA` is also set. To set the ADC into auto trigger mode with free running, the code required would be

```
ADCSRB = ((0 << ADTS2) | (0 << ADTS1) | (0 << ADTS0));
ADCSRA |= (1 << ADATE);
```

And yes, I know, ORing with zero has no effect, but it will have an effect for other auto trigger sources if I ever need to change the trigger source.

Warning
Register `ADCSRB` also contains, in bit 6, the `ACME` bit which is used by the Analog Comparator—see Section 9.1. Setting `ADCSRB` in the preceding manner will clear that bit which might affect the running of the comparator if your device needs it. In that case, the preceding code should probably be changed to preserve the `ACME` bit before ORing the desired auto trigger bits.

Auto triggering is initiated when a positive edge occurs on the selected trigger signal. When this occurs, the ADC's prescaler is reset, and a new conversion is started. If the triggering signal is still positive when the current conversion finishes, a new conversion will not be automatically started. Additionally, if another triggering positive edge is detected during an ADC conversion, the new triggering edge will be ignored.

The various Timer 0- and Timer 1-related auto triggering sources can be used to cause an ADC conversion to be initiated at regular intervals.

Even when auto triggering is enabled, your code can still manually request a single-shot conversion by initiating the conversion as described in the next section.

9.2.1.9 Enabling the ADC and Initiating Conversions

Everything is now configured; all that remains is to enable the ADC and, if necessary, initiate the first conversion. The ADC is enabled by writing a 1 to the ADEN bit in register ADCSRA, as follows:

```
ADCSRA |= (1 << ADEN);
```

When ADEN is set as before, the following events occur:

- The ADC starts consuming power.
- The ADC prescaler starts counting.
- If configured, auto triggering events will now initiate an ADC conversion.

An ADC conversion is manually requested, in single-shot or free running modes, by writing a 1 to the ADSC bit in register ADCSRA:

```
ADCSRA |= (1 << ADSC);
```

When this bit is set as before, the following events occur:

- The ADC prescaler is reset so that each conversion takes the same time.
- The sample and hold circuitry charges its capacitor with the voltage on the ADC input pin.
- The chosen reference voltage is enabled.
- The input channel selection is made, and the appropriate input is connected to the ADC.
- The conversion then starts at the next rising edge of the ADC clock.

9.2.1.10 ADC Conversions

The very first conversion takes 25 ADC clock cycles to complete so that the internal analog circuitry can be initialized. Subsequent conversions take only 13 ADC clock cycles.

If the ADC's reference voltage is the internal bandgap voltage, then it will take, according to the data sheet, *a certain time* for the voltage to stabilize. If it is not stabilized, the first conversion's result may be wrong. The data sheet, sadly, does not specify how long *a certain time* should be. My own code simply throws away the first reading in that mode of operation, although I have seen calls to `delay(20)` in some code as a suitable delay to allow the stabilization to occur.

The sample and hold of the input voltage takes place after 13.5 ADC clock cycles for the first conversion and after only 1.5 ADC clock cycles after the start of subsequent conversions.

When an ADC conversion is complete, the result is written to the data registers ADCH and ADCL, and then bit ADIF in ADCSRA is set. When running in single-shot mode, ADSC in ADCSRA is cleared simultaneously with the setting of ADIF.

If the code then sets bit ADSC, a new conversion will be initiated on the next rising edge of the ADC clock signal.

In any of the auto triggering modes, the ADC prescaler is reset as soon as the triggering event occurs. This ensures a fixed delay from the triggering event occurring to the start of a new ADC conversion. In this mode, the sample and hold takes place two ADC clock cycles after the rising edge on the trigger source signal. An additional three CPU clock cycles, not ADC clock cycles, are used for synchronization logic.

In free running mode, a new conversion will start as soon as the previous one completes, and this will occur even if the ADIF flag in the ADCSRA register is not cleared.

9.2.1.11 Noise Reduction

There is a sleep mode especially for the ADC. It disables many of the internal clocks, leaving the ADC to make its conversion with as few noise sources internally as possible. This sleep mode is discussed in Section 7.4.1.3.

It should be noted that this Noise Reduction mode is available only in single-shot ADC mode. The data sheet specifies that to enable this sleep mode, the following procedures should be used:

- Make sure that the ADC is enabled and is not busy converting.
- Single conversion mode must be selected, and the ADC conversion complete interrupt must be enabled.
- Enter ADC Noise Reduction sleep mode. The ADC will start a conversion once the CPU has been halted.
- If no other interrupts occur before the ADC conversion completes, the ADC interrupt will wake up the CPU and execute the ADC conversion complete interrupt routine. If another interrupt wakes up the CPU before the ADC conversion is complete, that interrupt will be executed, and an ADC conversion complete interrupt request will be generated when the ADC conversion completes.
- The CPU will remain in active mode until a new `sleep` command is executed.

It also gives the following point to note:

> The ADC will not be automatically turned off when entering other sleep modes than Idle mode and ADC Noise Reduction mode. The user is advised to write zero to `ADEN` before entering such sleep modes to avoid excessive power consumption.

9.2.1.12 Temperature Measurement

See Appendix E for an example of using this facility of the ADC. To select this ADC input, your code needs to

- Select the internal 1.1 V bandgap as the ADC reference voltage (`REFS1:0` = 0b11).
- Select ADC8 as the ADC input channel (`MUX3:0` = 0b1000).
- Set the prescaler to divide by 128 on an Arduino board at 16 MHz (`ADPS2:0` = 0b111).
- Execute single conversions as and when required. Auto triggering is not permitted.

The data sheet advises that readings from an uncalibrated sensor are

- −40 °C = 0x010D or 269.
- 25 °C = 0x0160 or 352.
- 125 °C = 0x01E0 or 480.

This works out at approximately 1.2769 per °C and that matches with the data sheet which states that it is *approximately* 1 LSB [°K] or 1 °C. The data sheet states that the temperature in °C is calculated as

$$(((((ADCH << 8) + ADCL) - (273 + 100 - TS\_OFFSET)) * 128)/TS\_GAIN) + 25$$

Warning
Given the rules of arithmetic precedence, the preceding code would cause ADCH to be read before ADCL, and this would not be the correct order! Using ADCW instead of $((ADCH << 8) + ADCL)$ would give the correct result and cause the two registers to be read in the correct order:

$$(((ADCW - (273 + 100 - TS\_\text{OFFSET})) * 128)/TS\_GAIN) + 25$$

TS_OFFSET	It is the calibration offset for the sensor and is stored in the device itself. It is a signed two's complement value.
TS_GAIN	It is the sensor gain factor and is also stored in the device. It is an unsigned, fixed point, eight-bit value representing 1/128th units, hence the need to multiply by 128 earlier.

In the data sheet, there is a small assembly language routine to obtain both TS_OFFSET and TS_GAIN from the device. I do not use that particular method of temperature conversion, so I have not discussed that routine here.

There are, however, all over the Internet, various methods of converting the temperature from what the ADC reads to °C; some I have seen are

1. $ADC - an\_offset$: The offset is dependent on the individual AVR device.
2. $(ADC - 247)/1.22$: From the developer help note mentioned earlier and linked in the following text.
3. $(((ADC - (273 - 100 - TS\_OFFSET)) * 128)/TS\_GAIN) + 25$: From the data sheet itself.
4. $ADC - 273$: From the application note on calibrating the sensor, linked in the following text.
5. $(ADC - an\ offset)/a\ gain\ factor$: From the "MySensors" code, linked in the following text.

The Microchip documents mentioned earlier are as follows:

- The developer help note is at https://microchipdeveloper.com/8avr:avradc.
- Application Note AVR122, on calibrating the temperature sensor, is at http://ww1.microchip.com/downloads/en/AppNotes/Atmel-8108-Calibration-of-the-AVRs-Internal-Temperature-Reference_ApplicationNote_AVR122.pdf and may prove useful if your maths[2] is better than mine!
- The MySensors code mentioned in the preceding text can be found at https://github.com/mysensors/MySensors/blob/bde7dadca6c50d52cc21dadd5ee6 d3623be5f3c6/hal/architecture/AVR/MyHwAVR.cpp.

Interestingly, the calibration document, *AVR122*, mentioned in the preceding text, states something a little different from the information in the data sheet, in that

> The output from the ADC is given in LSBs or °K, so the calibration values ADCT1 and ADCT2 have to be converted to °C. This is done by subtracting 273 from the values

[2] In the UK, we say *Maths*, plural, from Mathematics; I believe in the United States, it is *Math*, singular (from Mathematic?), which sounds really weird to my Scottish ears. At least we agree on *Arithmetic*.

This, to my mind, implies that the temperature sensor is outputting a value representing °K (Kelvin) whereby 0 °C is 273 °K, hence the need to subtract 273 from the ADC reading to get a value for Centigrade. In practice though, this does not always compute!

9.2.2 ADC Example

The example sketch initializes the ADC in free running mode and uses an interrupt to send the ADC reading to the Serial Monitor. The code was written and compiled in the Arduino IDE, but uses the plain AVR C language to set up the ADC.

As this code can be run on my Uno or Duemilanove at 16 MHz, or on one of my 8 MHz NormDuino boards, the prescaler for the AVR is calculated based on the `F_CPU` chosen in the Arduino IDE. The code will correctly determine whether the board is 16 MHz or 8 MHz and sets the correct prescaler accordingly.

Figure 9-2 shows the breadboard layout where I simply connected a potentiometer to pin `A0` to vary the voltage. An LED on pin `D9` with a 560 Ω resistor gives visual feedback as it brightens and dims according to where I had turned the potentiometer.

The first part of the sketch is shown in Listing 9-2 and is the function `setupADC()` which sets up the ADC directly. As mentioned earlier, it will be set up in free running auto trigger mode and will use the ADC interrupt to pass the readings to the main `loop()` as each one becomes available.

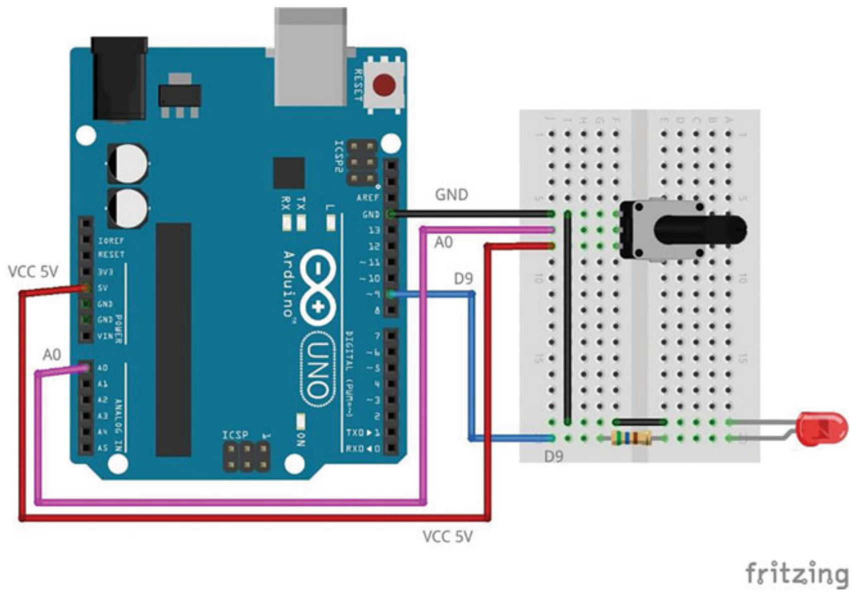


Figure 9-2 ADC example sketch breadboard layout

Listing 9-2 ADC example, setupADC() function

```
void setupADC() {
  // Ensure ADC is powered.
  PRR &= ~(1 << PRADC);                                          (1)

  // Slow the ADC clock down to 125 KHz
  // by dividing by 128 or 64. 128 is for a 16MHz Arduino
  // 64 for an 8MHz NormDuino. Does not cater for other
  // clock speeds here. BEWARE.
  #if F_CPU == 16000000                                          (2)
    ADCSRA = (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
  #else
  // Non-standard 8MHz clock in use.
    ADCSRA = (1 << ADPS2) | (1 << ADPS1) | (0 << ADPS0);
  #endif

  // Initialise the ADC to use the
  // internal AVCC 5V reference voltage.
  ADMUX = (0 << REFS1) | (1 << REFS0);                           (3)

  // Ensure result is right aligned.
  ADMUX &= ~(1 << ADLAR);

  // Use the ADC multiplexer input
  // ADC0 = Arduino pin A0.
  ADMUX |= ((0 << MUX3) | (0 << MUX2) |                          (4)
            (0 << MUX1) | (0 << MUX0));

  // Disable ADC0 Digital input buffer.
  DIDR0 |= (1 << ADC0D);

  // Use the interrupt to advise when a result is available.
  ADCSRA |= (1 << ADIE);                                         (5)

  // Set auto-trigger on, and choose Free Running mode. As
  // we are not using the Analog Comparator, we don't care
  // about the ACME bit in ADCSRB.
  ADCSRA |= (1 << ADATE);                                        (6)
  ADCSRB = 0;

  // Enable the ADC and wait for the voltages to settle.
  ADCSRA |= (1 << ADEN);                                         (7)
  delay(20);
}
```

(1) This powers the ADC by disabling the "disable the ADC" bit.
(2) These lines work out the prescaler for 16 MHz or 8 MHz devices. Other speeds are not catered for here. I only have two speeds on my devices.
(3) This selects the internal 5 V reference voltage, which is fed by pin `AVCC` which must be connected to `VCC` plus or minus 2%.
(4) I know it's all zeros, but it's easy to change this line for different ADC input pins.
(5) We are using interrupts, so they must be enabled, as must the global interrupts. This is always the case when compiling in the Arduino IDE, but not necessarily in other IDEs. Beware.
(6) Setting `ADATE` enables auto trigger mode. Setting `ADCSRB` to zero enables free running mode. It also messes up the Analog Comparator, but we don't care here. Other code might care, so bear it in mind.
(7) Enable but do not start the ADC. From here on, the ADC draws power and has initialized the reference voltage selector and the reference voltage source. It is ready to go.

After executing the code in Listing 9-2, the ADC is now fully initialized and enabled; however, it has not yet been started. In free running and single-shot modes, the ADC must be started manually. The `startADC()` function in Listing 9-3 does exactly that by setting bit `ADSC` in `ADCSRA`.

Listing 9-3 ADC example, startADC() function

```
void startADC() {
  ADCSRA |= (1 << ADSC);
}
```

Listing 9-4 sets up a `volatile` variable, `ADCReading`, to hold the result passed back from the interrupt handler. The variable must be defined as `volatile`; otherwise, the compiler might notice that it doesn't seem to be changing in value and may simply "optimize" it away. Any global variables that you wish to change from inside an interrupt handler should be defined as `volatile` to prevent this from happening.

Following the variable declaration, we need an interrupt handler for the ADC interrupt. All that it needs to do is to copy the ADC result from `ADCW` into `ADCReading`. The `loop()` function, in Listing 9-6, will do some work with this result.

Listing 9-4 ADC example, the interrupt handler

```
// Somewhere for the ADC Interrupt to store the result.
volatile uint16_t ADCReading = 0;

// The interrupt handler.
ISR(ADC_vect) {
  ADCReading = ADCW;
}
```

The `setup()` function, in Listing 9-5, initializes the ADC using the `setupADC()` function from Listing 9-2, then sets up the Serial Monitor to display the results and fires up the ADC for its first reading. Once the first reading is complete, the ADC will then be in free running mode and will constantly be initiating a new conversion as soon as the current one finishes.

Listing 9-5 ADC example, the setup() function

```
void setup() {
  setupADC();

  // Use the Serial Monitor for output.
  Serial.begin(9600);
  Serial.println("Arduino Direct ADC Testing");

  // Add an LED and 560R resistor to pin 9 for feedback.
  pinMode(9, OUTPUT);

  // Now, fire up the ADC.
  startADC();
}
```

And finally, Listing 9-6 shows the `loop()` function, which takes the most recent ADC reading and sends it to the Serial Monitor, firstly as a plain value between 0 and 1,023 and secondly as a voltage.

As there are 1,024 different values that can be returned from the ADC and with 0 representing `GND` and 1,023 representing `AVCC` or 5 V, anything in between must be equal to $5/1,024$ per division. We simply multiply `ADCReading` by this fraction, 0.004882812 V (4.88 millivolts and a little bit), to get the voltage on the `ADC0` or `A0` pin. We must be careful to cast the result to a float, or we will lose accuracy and only see integer values.

Listing 9-6 ADC example, the loop() function

```
void loop() {
  Serial.print("ADC = ");
  Serial.print(ADCReading);

  // The voltage is ADCReading * (5V/1024)
  Serial.print(", Voltage = ");
  Serial.println((float)(ADCReading * 5.0 / 1024.0));

  // Light up the LED to a value representing the voltage
  // on pin A0 (ADC0).
  analogWrite(9, map(ADCReading, 0, 1023, 0, 255));

  delay(500);
}
```

There's a small `delay()` at the end of the `loop()` to prevent the numbers from scrolling up the screen too quickly. Don't worry about the ADC though; it will carry on taking readings as the interrupt handler works outside of any `delay()` periods and is not affected. Bear in mind, however, that the `loop()` will not be able to display every ADC reading taken; there will be losses during the `delay()` when it is unable to display anything.

9.3 USART

The USART is the Universal Synchronous/Asynchronous Receiver Transmitter. It's easier to type USART!

On the Arduino, it is connected from physical pins 2 and 3 to the laptop or desktop's USB port via either a separate Microchip AVR microcontroller or an FTDI chip—depending on which Arduino you have.

Pin 2 is the receive or `RX` pin, while pin 3 is the transmit or `TX` pin. You don't have to worry about this on an Arduino, but when you have your own naked ATmega328P on a breadboard or PCB, and you want to communicate with it, you do have to be careful. With a serial device such as an FTDI connector, at least the one I have, the pins are marked `TXO` and `RXO` for output. So the `TXO` pin on the FTDI connects to the ATmega328P's `RX` pin, and the `RXO` on the FTDI connects to `TX` on the AVR.

Some FTDI devices have the pins marked as `TX` and `RX`, and with *some of those*, you connect the `TX` to the `TX` and the `RX` to `RX`. With others, you connect the `TX` to the `RX` and the `RX` to `TX`.

Confused? You should be!

It will not break anything if you get them crossed over, but nothing will work. If that happens to you, just swap the wires over at the AVR end.

The USART can be set up to transmit only, receive only, or to do both, and it can use interrupts to facilitate this, having three dedicated interrupts available.

In synchronous modes, the USART will require a clock pin and a data pin, whereas asynchronous mode does not need a clock and can use one pin of the two available to the USART, as the `TX` pin and the other for the `RX` pin. The Arduino uses the latter mode, asynchronous.

9.3.1 Baud Rates

The baud rate is configured by the value stored in the USART Baud Rate Register, UBRRn, where "n" is the USART number. On the ATmega328P, there is only `USART0`. The Mega 2560 has four, `USART0` through `USART3`.

`UBRR0` is connected to a counter which counts down at the `F_CPU` frequency, and whenever it reaches zero, a USART clock is generated, and this clock controls the USART's transmission and/or receipt of data. When the counter reaches zero, it is reloaded with the value in `UBRR0`. This clock is the Baud Rate Generator Clock and has the frequency given by

$$F\_CPU/(UBRR0 + 1)$$

The Baud Rate Generator Clock is divided down by the USART's transmitter by 2, 8, or 16 depending on the configured mode.

The receiver circuitry, on the other hand, does no such division and uses the Baud Rate Generator Clock directly as input to its data recovery unit. Within the data recovery unit, there is a state machine which uses 2, 8, or 16 states to determine correct receipt of the transmitted data.

9.3.2 Double Speed

The transfer rate of the USART can be doubled by setting bit `U2X0` in register `UCSR0A`. However, this bit should only be set in *asynchronous* operation; it should be zero for synchronous modes. Setting the bit causes the baud rate divider to be reduced to 8 from the usual 16, and this effectively doubles the transfer rate.

The receiver, however, will also only sample 8 times, rather than 16, in the data recovery unit, thus doubling the receive speed too, but it should be noted that in this mode, a more accurate baud rate setting is required as well as a more accurate system clock. Some baud rates can cause excessive error rates. Errors are discussed later.

For the transmission side of the USART, doubling the speed has no apparent drawbacks. (At least, that's what the data sheet says.) The Arduino almost always runs in double speed mode, unless the system clock is running at 16 MHz *and* the requested baud rate is 57,600. This is, apparently, for compatibility with the bootloader shipped with the Duemilanove and earlier boards.

9.3.3 Baud Rate Calculations

In single speed asynchronous mode, the baud rate is calculated as

$$BAUD = F\_CPU/16 * (UBRR0 + 1)$$

Normally, however, you are more interested in setting a specific baud rate, so you would need to calculate `UBRR0` for the desired rate. This is done using the formula:

$$UBRR0 = (F\_CPU/(16 * BAUD)) - 1$$

You will probably have figured out that you can just about define any baud rate you wish by setting `UBRR0` to any given value.

In double speed asynchronous mode, the formulas change to

$$BAUD = F\_CPU/8 * (UBRR0 + 1)$$

$$UBRR0 = (F\_CPU/(8 * BAUD)) - 1$$

What would the required `UBRR0` setting be for an Arduino running at 16 MHz, for a baud rate of 9,600 in double speed mode?

$$\begin{aligned} UBRR0 &= (F\_CPU/8 * BAUD) - 1 \\ &= (16{,}000{,}000/8 * 9600) - 1 \\ &= (16{,}000{,}000/76{,}800) - 1 \\ &= 208.3333 - 1 \\ &= 207.3333 \end{aligned}$$

Do you see a problem? We are working with registers, and registers cannot have fractions, so the preceding calculation introduces errors because the result is not an integer. The question is do we round down and use the value 207 or round up to use 208?

What are the actual baud rates obtained with those values? If we feed them back into the equation, we will see first for `UBRR0` = 207:

$$\begin{aligned} BAUD &= F\_CPU/8 * (UBRR0 + 1) \\ &= 16{,}000{,}000/8 * (207 + 1) \end{aligned}$$

$$\begin{aligned} &= 16{,}000{,}000/8 * 208 \\ &= 16{,}000{,}000/1{,}664 \\ &= 9615.3846 \end{aligned}$$

and now for UBRR0 = 208:

$$\begin{aligned} BAUD &= F\_CPU/8 * (UBRR0 + 1) \\ &= 16{,}000{,}000/8 * (208 + 1) \\ &= 16{,}000{,}000/8 * 209 \\ &= 16{,}000{,}000/1{,}672 \\ &= 9569.3780 \end{aligned}$$

Neither of these are 9,600, which we wanted, so it looks like rounding down, in this case, works out closer to our desired baud rate.

9.3.4 Baud Rate Errors

With `UBRR0` set to 207 or 208, as calculated, we have rounded baud rates of 9,615 and 9,569. My rounding here is simply to truncate the value and drop the fractional part.

We wanted 9,600, but we get either 9,615 or 9,569. One is running high, the other low. Which has the lowest error rate?

The data sheet calculates the error rate, as a percentage, as

$$error\% = ((BAUD\_got/BAUD\_wanted) - 1) * 100$$

The data sheet supplies numerous tables showing the desired baud rates, the `UBRR0` setting, and error rates for many different values of `F_CPU`, and almost all appear to be wrong!

Taking the best value calculated earlier and checking the data sheet, it shows the error rate as being 0.2% for `UBRR0` having the value 207. I feel the need to run a quick check myself and calculate the error rates for `UBRR0` = 207 and 208.

$$\begin{aligned} error\% &= ((BAUD\_got/BAUD\_wanted) - 1) * 100 \\ &= ((9615/9600) - 1) * 100 \\ &= (1.0015625 - 1) * 100 \\ &= 0.1526\% \end{aligned}$$

$$\begin{aligned} error\% &= ((BAUD\_got/BAUD\_wanted) - 1) * 100 \\ &= ((9569/9600) - 1) * 100 \\ &= (0.996770833 - 1) * 100 \\ &= -0.3229\% \end{aligned}$$

Hmm, neither of those are not quite the 0.2% that the data sheet mentions. So, let's try again for 207, but this time, include any fractions:

$$\begin{aligned} error\% &= ((BAUD\_got/BAUD\_wanted) - 1) * 100 \\ &= ((9615.3846/9600) - 1) * 100 \\ &= (1.001602564 - 1) * 100 \\ &= 0.1602\% \end{aligned}$$

That's *still* not 0.2%. There are many other discrepancies in the data sheet on the matter. To be honest, I think the data sheet is rounding the calculated error rates up to the next 0.1%.

Regardless of the data sheet's approximate error rates, the advice given is to choose a baud rate which gives an error rate of ±0.5%. On an Arduino board, running at 16 MHz, a 9,600 baud rate is within specifications. Using this advice, it would seem that `UBRR0` can safely be set to 207 as calculated.

9.3.5 What Is a Frame?

According to the data sheet:

> A serial frame is defined to be one character of data bits with synchronization bits (start and stop bits), and optionally a parity bit for error checking.

Start and stop bits are used to synchronize the transmitting and receiving devices, while parity bits are used to apply rudimentary error checking. The start bit signals to the receiver that a frame is about to be transmitted.

The USART is able to be configured to use any combination of the following:

- A single start bit
- Five to nine data bits
- None, even, or odd parity
- One or two stop bits

A frame always begins with the single start bit; this is followed by five to nine data bits with the least significant bit first. If parity is enabled, the parity bit follows and, finally, the one or two stop bits.

When a frame has been transmitted, it can either be followed by another frame, or the line can be set to `HIGH` for the idle state.

It is because of the frame structure that regarding the baud rate as the number of characters per second is incorrect. For an 8-bit character set, a frame can be as much as 12 bits long.

9.3.6 Parity

The USART operates, or can be configured to do so, in odd or even parity or with no parity at all. When configured to use parity, the way it is calculated is to exclusive OR, or XOR, each of the *data bits*, but not the bits in the frame, then to XOR the result with a zero bit for even parity or a one bit for odd parity.

If even parity is in use, the parity bit is used to make the number of one bits in the data even. Odd parity makes the number of one bits odd. The parity bit will be found between the final data bit and the first stop bit of a frame.

As an example, the letter “A,” in ASCII, has code 65, 0x41, or 0b0100 0001. There are two bits that are 1, so for even parity, the parity bit must be a 0, and for odd parity, it must be a 1.

The letter “C,” on the other hand, has code 67, 0x43, or 0b0100 0011. This has three 1 bits, so the parity bit in even parity will be a 1, and for odd parity, it will be a 0.

9.3.7 USART Interrupts

The USART has three separate interrupts that can be used; two are for transmission and one for receiving data. These are

- TX Complete
- TX Data Register Empty
- RX Complete

9.3.7.1 TX Complete Interrupt

This interrupt fires when the data written to the `UDR0` register has been framed in start, stop, and parity bits as appropriate, and the whole of the frame has been transmitted.

The Arduino software does not use this interrupt.

9.3.7.2 TX Data Register Empty Interrupt

This interrupt fires when the data written to the `UDR0` register has been written into the shift register buffer internally to be framed. The USART may still be in the midst of sending the byte down the line, but the `UDR0` register is now empty, and another byte can be written to it.

This is the interrupt used by the Arduino’s Serial interface and allows for a slightly quicker processing of data to be written to the USART as it can be written into the `UDR0` register even as the previous byte is still being wrapped in its frame and transmitted.

9.3.7.3 RX Complete

When this interrupt fires, a new data byte has been received by the USART and is waiting to be retrieved from the `UDR0` register. The USART has a two-byte buffer for received characters. If your code doesn’t read the `UDR0` register fast enough, and a third data byte is detected to be arriving—by detecting a new start bit—then a buffer overrun error will be the result.

Note

It appears that the `UDR0` register is used by both the transmit and receive parts of the USART. How can this be when the USART, in full duplex mode, can be both transmitting and receiving? When data are read from `UDR0`, the byte of data recently received is returned to the calling code, while when data are written to `UDR0`, it is forwarded on to the transmitter side of things. The ATmega328P knows what it is doing!

9.3.8 Initializing the USART

It goes without saying that the USART will have to be initialized before any communication can take place. The process usually requires the USART to be powered up, choosing the USART mode, setting the baud rate, setting the frame requirements, and enabling the transmitter and/or the receiver depending on the requirements of the code.

When running code with interrupt-driven USART operations, the global interrupts should be disabled during the setup.

If the USART needs to be reconfigured, perhaps to change the frame format or the baud rate, then the code must ensure that the existing settings have been finished with and that all current transmissions and receipts are completed. This can be carried out by checking the `TXC0` and/or `RXC0` bits in the `UCSR0A` register.

The USART has three separate control registers:

`UCSR0A`	is used for various error flags and transmit and receive complete flags and to set double speed mode and multiprocessor communications mode.
`UCSR0B`	is used to enable interrupts, to enable transmit and receive modes, and to hold the ninth bit of nine-bit data frames for transmission and receipt and one of the three bits used to set the data size; the other two are in `UCSR0C`.
`UCSR0C`	is used to select the USART mode, parity settings, stop bits, the remaining two bits of the data size settings, and the clock polarity.

In order to ensure that your USART is fully initialized, without relying on some defaults that may not be as expected, it is best to start from a known, clear configuration by clearing all three control registers:

```
UCSR0A = 0;
UCSR0B = 0;
UCSR0B = 0;
```

Any options that the code requires can now be safely OR'd into the appropriate registers. The following examples will assume this mode of operation.

9.3.8.1 Powering the USART

The USART is powered up by clearing the `PRUSART0` bit in the Power Reduction Register, `PRR`, as described in Section 7.5.2.

```
PRR &= ~(1 << PRUSART0);
```

At any time the USART is powered off and then on again, it must be *fully reinitialized* to ensure proper operation. You cannot rely on the previous settings to remain in force through a power down of the USART.

9.3.8.2 Choosing the USART Mode

The USART can be operated in one of three modes:

- Asynchronous USART (the default)

Table 9-7 USART mode settings

UMSEL01:00	USART Mode
0b00	Asynchronous USART (the default)
0b01	Synchronous USART
0b10	Reserved
0b11	Master SPI

- Synchronous USART
- Master SPI

The default mode is asynchronous USART, and the desired mode is defined by setting the `UMSEL01:00` bits in the `UCSR0C` register, USART Control Status Register C, as defined in Table 9-7.

Synchronous modes require a clock and a data line, whereas asynchronous mode doesn't require a clock line. There are two pins available to the USART, so because asynchronous mode doesn't use a clock, the two pins can be configured as `TX` and `RX`. In this mode, full duplex, transmission, and reception can occur at the same time. Arduino boards run in asynchronous mode.

Setting the USART to run in, for example, Master SPI mode, your code would execute the following:

```
UCSR0C |= ((1 << UMSEL01) | (1 << UMSEL00));
```

9.3.8.3 Baud Rate Setting

Setting the baud rate has already been described. You must calculate the appropriate value for the `UBRR0` register and load it. For example:

```
// Settings for double speed 9600 baud rate.
#define BAUD 9600
#define UBRR0_9600 ((F_CPU) / 8 * (BAUD)) - 1
...
UBRR0 = UBRR0_9600;
```

9.3.8.4 Frame Settings

The frame settings define the start bit, which is always present, the number of data bits to be transmitted/received, whether or not a parity bit is required, and the number of stop bits.

9.3.8.5 Setting Parity

Bits `UPM01:00` in the `UCSR0C` register, USART Control Status Register C, define the USART parity mode. Table 9-8 shows the bit settings for the different parity modes.

Table 9-8 USART parity settings

UPM01:00	Parity
0b00	No parity (the default)
0b01	Reserved
0b10	Even parity
0b11	Odd parity

Table 9-9 USART stop bit settings

USBS0	Stop Bits
0	1 stop bit (the default)
1	2 stop bits

Table 9-10 USART data width settings

UCSZ02:00	Data Size
0b000	Five bits
0b001	Six bits
0b010	Seven bits
0b011	Eight bits (the default)
0b100	Reserved
0b101	Reserved
0b110	Reserved
0b111	Nine bits

To configure the USART with even parity:

```
UCSR0C |= ((1 << UPM01) | (0 << UPM00));
```

9.3.8.6 Setting Stop Bits

The USBS0 bit in the UCSR0C register defines the number of stop bits as shown in Table 9-9.

The code to configure the USART with two stop bits would therefore be

```
UCSR0C |= (1 << USBS0);
```

9.3.8.7 Setting Data Width

Bits `UCSZ01:00` in register `UCSR0C`, along with bit `UCSZ02` in register `UCSR0B`, define the number of data bits in a frame. The default at power on/reset is eight bits. Table 9-10 shows the valid settings for the data width which the USART will use. You will note some settings are not permitted.

To set, for example, nine bits of data in the frame, your code should execute the following:

```
UCSR0B |= (1 << UCSZ02);
UCSR0C |= ((1 << UCSZ01) | (1 << UCSZ01));
```

9.3.8.8 Enabling Double Speed Mode

To double the speed of communications, both transmission and receipt, in asynchronous mode only, set bit `U2X0` in register `UCSR0A` as follows:

```
UCSR0A |= (1 << U2X0);
```

If the USART is to be operated in synchronous mode or asynchronous single speed mode, then this bit should be explicitly cleared:

```
UCSR0A &= ~(1 << U2X0);
```

Table 9-11 USART interrupts

Bit	Interrupt
RCXCIE0	RX Complete interrupt will be enabled. The code in ISR(USART_RXC_vect) will handle reading the UDR0 register to retrieve the byte just read
TXCIE0	TX Complete interrupt will be enabled. The code in ISR(USART_TXC_vect) will handle writing a new data byte to the UDR0 register ready to be transmitted
UDRIE0	USART Data Register Empty interrupt will be enabled. The code in ISR(USART_UDRE_vect) will handle writing a new data byte to the UDR0 register ready to be transmitted

9.3.8.9 Enabling Interrupts

The USART has thee interrupts, as detailed earlier. Bits RCXCIE0, TXCIE0, and UDRIE0, in register UCSR0B, control which, if any, of the interrupts will be used, and the settings are displayed in Table 9-11.

The latter two interrupts *appear* to do the same thing. They do, but slightly differently.

The TX Complete interrupt is fired when the complete data *frame* of up to 12[3] bits has been transmitted. At this point, any new data written to UDR0 will have to be framed before it can be transmitted.

The USART Data Register Empty interrupt is fired whenever the data most recently written to UDR0 has been copied to the transmit shift buffer internally to the AVR microcontroller. There can be up to eight[4] bits of data, and so this interrupt can fire when the previous character is still in the midst of being framed and transmitted, and, in doing so, you can get a better throughput as the byte can be framed and transmitted as soon as the previous byte is on its way down the line.

The Arduino uses the USART Data Register Empty interrupt for better throughput.

Enabling these interrupts is a matter of setting to the appropriate bit in the UCSR0B register. You should always remember to clear the interrupt flag bits in UCSR0A prior to enabling the interrupt. This will avoid spurious interrupts firing:

```
// Turn off interrupts.
cli();

// Clear interrupt flags.
UCSR0A &= ~((1 << RXC0)|(1 << TXC0));

// Enable RX Complete & USART data register empty interrupts.
UCSR0B |= ((1 << RCXCIE0) | (1 << UDRIE0));
...
sei();
```

[3] 13 bits on non-Arduino boards.

[4] Nine bits on non-Arduino boards.

Warning
If interrupts are to be used, it is considered best to disable global interrupts while initializing the USART. This will prevent spurious firing of the USART interrupts when the USART is not fully configured.

9.3.8.10 Enabling Data Transmission

To enable transmission of data, the USART is configured as follows:

```
UCSR0B |= (1 << TXEN0);
```

Doing so will override the normal function of the `TX` pin on the AVR microcontroller. This corresponds to Arduino pin `D1` or AVR pin `PD1`.

9.3.8.11 Enabling Data Receipt

To enable receipt of data, the USART is configured as follows:

```
UCSR0B |= (1 << RXEN0);
```

Doing so will override the normal function of the `RX` pin on the AVR microcontroller. This corresponds to Arduino pin `D0` or AVR pin `PD0`.

9.3.8.12 Transmitting or Receiving Nine-Bit Data

Note
Nine-bit data is not something that your Arduino board can facilitate. However, the ATmega328P can, so I have included it here for completeness.

Nine-bit data? What's all that about? It can happen that some data transmissions require nine bits for each character; the ATmega328P's USART can cope with this. Given that data bytes in registers are only eight bits long, where does the ninth bit get stored?

Bits `TXB80` and `RXB80` in register `UCSRB0` are the storage locations. These bits hold the ninth bit when transmitting and receiving nine-bit data. You should note that

- When transmitting data, the ninth bit must be written to `TXB80` before writing the remaining eight bits to the `UDR0` register.
- When receiving data, the ninth bit must be read from `RXB80` before reading the rest from `UDR0`.

In either case, the appropriate bit in register `UCSRB0` holds the most significant bit of the nine-bit data; this will be bit number eight as bit numbers from zero upward remember. The lower eight data bits will be in the `UDR0` register when transmitting or receiving nine-bit data.

9.3.9 USART Completion and Error Checking

When all the initialization has been completed and the USART is now transmitting and/or receiving data quite happily, how do you check for completion or errors?

With interrupts in force, you will know when data are received and/or transmitted without problems as the appropriate interrupt will fire. However, if you are not using interrupts, you will have to poll various bits in the control registers to see if data have been received or transmitted.

All of the bits to be checked or polled are found in register `UCSR0A`.

9.3.9.1 USART Receive Complete Bit

Bit `RXC0` in register `UCSR0A` is set when the current byte being received has been received and unframed. The data byte will be available for reading from the `UDR0` register.

When the `UDR0` register is subsequently read, `RXC0` will be automatically cleared, as it will if the USART RX Complete interrupt is enabled and the ISR has been executed.

Code may also clear this bit by the usual manner of writing a 1 to it.

9.3.9.2 USART Transmit Complete Bit

Bit `TXC0` in register `UCSR0A` is set when the frame in the transmit buffer (a simple shift register) has been completely shifted out onto the data line, and no new data is waiting in the `UDR0` register for transmission.

This bit will be automatically cleared when the USART TX Complete interrupt handler has been executed or can be cleared in code by writing a 1 to it.

9.3.9.3 USART Data Register Empty Bit

Bit `UDRE0` in register `UCSR0A` indicates that the register `UDR0` is now empty, its previous contents having been copied into the transmit buffer ready for framing and transmission.

This bit will be automatically cleared when the USART Data Register Empty interrupt handler has been executed. Application code may also clear this bit by writing a 1 to it.

On reset or power up, this bit is initialized to a 1 to show that the transmit data register is ready to accept new data.

9.3.9.4 USART Frame Error

Bit `FE0` in register `UCSR0A` will be set if the received data byte had a framing error in that the first stop bit was detected as a zero. `FE0` remains set until `UDR0` is read and so should be checked prior to reading the data. When writing to the `UCSR0A` register, this bit should always be written as zero.

9.3.9.5 USART Data Overrun

Bit `DOR0` in register `UCSR0A` will be set whenever a data overrun condition is detected.This will occur when

- The USART's receive buffer is full—it can only hold a maximum two characters.
- There is a character waiting in the USART's receive shift register to be copied into the receive buffer.
- A new start bit is detected on the `RX` pin.

`DOR0` will remain set until `UDR0` is read, freeing up space for new data. When writing to the `UCSR0A` register, this bit should always be written as zero.

9.3.9.6 USART Parity Error

Bit UPE0 in register UCSR0A will be set if the next character in the receive buffer had a parity error when received but only if the USART had parity checking enabled at the time the data was received (bit UPM01 in register UCSR0C was set).

UPE0 will remain set until UDR0 is read. When writing to the UCSR0A register, this bit should always be written as zero.

9.3.10 USART Example

The code that follows in Listings 9-7 through 9-15 is a sketch to demonstrate the use of the USART without help from the Arduino Language and without using the Serial interface as we would normally do. The code that follows is all one sketch but is split into separate functions here for explanation.

The sketch begins with the setupUSART() function in Listing 9-7.

Listing 9-7 USART sketch, setupUSART() function

```
#define BAUD 115200
#define BUFFER_SIZE 100

#define SET_UBRR0(x) ((F_CPU) / (8 * (x))) - 1          (1)

void setupUSART(unsigned long baudRate) {
  // Sets up the USART to send and receive at a given
  // baud rate, 8 data bits, one stop bit, no parity.
  // It doesn't use interrupts.

  // Ensure we have power/clock.
  PRR &= ~(1 << PRUSART0);                               (2)

  // Calculate the baud rate setting.
  UBRR0 = SET_UBRR0(baudRate);                           (3)

  // Initialise registers, then set TX and RX on,
  // 8 data plus one stop bit. The others are defaulted.
  // Double speed communications are in use.
  UCSR0A = UCSR0B = UCSR0C = 0;                          (4)
  UCSR0A |= (1 << U2X0);                                 (5)
  UCSR0B |= ((1 << TXEN0) | (1 << RXEN0));               (6)
  UCSR0C |= ((1 << UCSZ01) | (1 << UCSZ00));             (7)
}
```

(1) This is a quick way to convert the desired baud rate into the required value for UBRR0. Change the 8 to 16 if using single speed communications.
(2) Always remember to power up the USART first.
(3) We set the required baud rate here.
(4) Clearing all the control registers is a good way to set up the system to a known configuration.

(5) This line enables double speed communications. Some baud rates have too many errors in single speed, 115,200, for example.
(6) This line enables transmission and receipt of data.
(7) These two bits set the data size in the frame to eight data bits.

You will note that much of the setup was not mentioned. Where, for example, did I put the USART into asynchronous mode? My initialization of the three control registers to zero did the following for me:

`UCSR0A`	This sets both double speed and multiprocessor modes off. I then enabled double speed communications after initializing the register.
`UCSR0B`	This sets all interrupts off.
`UCSR0C`	This sets the mode to asynchronous and defines no parity and one stop bit.

All that the rest of `setupUSART()` had to do was enable eight-bit data size and turn on transmission and receipt of data.

The next function, shown in Listing 9-8, is the code to receive a single byte of data from the RX line. This code will block if there is nothing currently being received.

Listing 9-8 USART sketch, receiveByte() function

```
uint8_t receiveByte() {
  // Wait for bit RXC0 to be set in UCSR0A.
  loop_until_bit_is_set(UCSR0A, RXC0);                 (1)
  return UDR0;                                         (2)
}
```

(1) Wait here until the `RXC0` bit gets set; once that happens, the data in the `UDR0` register is valid.
(2) Retrieve and return the character just read.

Receiving one byte at a time is OK, but sometimes you just want more! The next function, `receiveText()`, receives a whole string of characters. Listing 9-9 has the details.

Listing 9-9 USART sketch, receiveText() function

```
uint8_t receiveText(char *buffer, uint8_t howMany) {

  // Receive a string of text up to howMany characters
  // or until a terminating linefeed is received. Any
  // carriage returns are ignored for Windows users, we
  // only care about linefeeds.
  //
  // MAKE SURE that the serial monitor is set to send a
  // NEWLINE or this code will fail to return until the
  // buffer fills.
  //
  // Assumes the caller knows what s/he is doing! The
  // buffer should be one more than howMany in length.
  uint8_t i = 0;
  while (i < howMany) {                                    (1)
    uint8_t c = receiveByte();

    // We are not interested in carriage returns.
    if (c == '\r')                                         (2)
      continue;

    // We are interested in linefeeds though.
    if (c == '\n') {                                       (3)
      buffer[i] = '\0';
      return i;
    }

    buffer[i++] = c;                                       (4)
  }
  // We must have received howMany characters.
  buffer[i] = '\0';                                        (5)
  return howMany;
}
```

(1) This loop will exit on one of two conditions:

- The buffer is filled up with `howMany` characters, and a line feed has not been seen.
- The last character received was a line feed.

(2) If the character just received was a carriage return, much loved by Windows, then we simply ignore it and wait for another character.
(3) If the character just received is a new line, overwrite it in the buffer with a string terminator and return to the caller.
(4) Otherwise, store the character just read and loop around again.
(5) If we exit the loop here, we have filled the buffer with `howMany` characters. Add a buffer terminating "\0" character and return.

This is a pretty simple function to be honest, but it works. As long as the buffer has `howMany` characters plus one, it will work perfectly if the input received is less than the buffer size, which is something that is the responsibility of the programmer.

If the received data is longer, then it is possible that the USART will suffer a data overrun error and lose characters. You can see this by running the sketch and holding down a key until there have been more than the buffer size typed, then press ENTER.

The first buffer full of characters will be correctly displayed, and the first two characters after that will also be displayed. Anything typed after those two will be lost as the USART has an internal, two-character receive buffer. When two characters remain unread in the buffer and a new character is received, the third and all subsequent characters are lost until such time as the buffered characters have been read.

Interrupts would be a better solution to this problem, but even the Arduino's interrupt-driven Serial interface will drop characters if there is a buffer overrun.

So far, that's all we really need for simple USART receipt of data; what about sending data out? Listing 9-10 shows how we can send a single byte down the wire via the USART.

Listing 9-10 USART sketch, sendByte() function

```
void sendByte(uint8_t c) {
  // Wait for bit UDRE0 to be set in UCSR0A then
  // buffer up the data byte.
  loop_until_bit_is_set(UCSR0A, UDRE0);                  (1)
  UDR0 = c;                                              (2)
}
```

(1) Wait here until the data buffer is empty.
(2) Add the character to be sent to the buffer where it will be wrapped in a frame and transmitted.

Again, sending one byte is no fun, so the `sendText()` function in Listing 9-11 sends a whole string of characters.

Listing 9-11 USART sketch, sendText() function

```
void sendText(const uint8_t *text) {
  // Transmit a string of text. One byte
  // at a time.
  uint8_t *i = text;
  while (*i)
    sendByte(*i++);
}
```

The code simply walks through the passed buffer sending each character out through the USART transmitter until it finds the end of string character, which does not get sent.

The `sendNumber()` function in Listing 9-12 allows you to send numeric data in almost any radix (number base) that you desire, although the default is decimal. This only handles signed integer values; I leave it as an exercise for the reader to write a `sendFloat()` function.[5]

Listing 9-12 USART sketch, sendNumber() function

```
void sendNumber(const long x, const uint8_t r = 10) {
  // Transmit a long integer to the USART. Only 32 bits
  // can be sent.
  char buffer[40];                                       (1)
  ltoa(x, buffer, r);                                    (2)
  sendText(buffer);                                      (3)
}
```

[5] Ah, yes! The old "I leave it as an exercise for the reader" excuse for not doing something!

(1) The length of a long variable is 32 bits, so 40 characters is enough of a buffer to cope without crashing. 2^{32} is 4,294,967,296 which is ten digits in size. There is plenty of room in the 40-character buffer to hold a signed or unsigned number in decimal, binary, hex, or octal. The smallest negative number being −2,147,483,648 in decimal.
(2) The `ltoa()` (long to ASCII) function does all the hard work. It also adds on a terminating "\0" character to the buffer, which is another reason for having a bit extra on the end.
(3) The buffered ASCII representation of the number is then transmitted.

The `communicate()` function in Listing 9-13 demonstrates some of the code we have seen earlier in action. It sends out the number $2^{32} - 1$ in various formats.

Listing 9-13 USART sketch, communicate() function

```
void communicate() {
  const long number = 4294967295;
  sendText("Number = 2^32 -1 in HEX: ");                    (1)
  sendNumber(number, 16);
  sendByte('\n');

  sendText("Number = 2^32 -1 in DEC: ");                    (2)
  sendNumber(number, 10);
  sendByte('\n');

  sendText("Number = 2^32 -1 in OCT: ");                    (3)
  sendNumber(number, 8);
  sendByte('\n');

  sendText("Number = 2^32 -1 in BIN: ");                    (4)
  sendNumber(number, 2);
  sendByte('\n');

  sendText("\n\n");
  sendText("Type some text or numbers ... \n");
}
```

(1) Sends out a big number in hexadecimal. The number is $2^{32} - 1$ and is the biggest that will fit into a `long` data type.
(2) Sends out a big number in decimal. In this case, it gets printed as -1 because the parameter is `signed` in the call to `ltoa()` and $2^{32} - 1$ is indeed −1 when dealing in signed values.
(3) Sends out a big number in octal.
(4) Sends out a big number in binary.

The well-known and much loved `setup()` function is shown in Listing 9-14.

Listing 9-14 USART sketch, setup() function

```
void setup() {
  // Initialise the USART without needing Serial.
  setupUSART(9600);

  // Play with the USART.
  communicate();
}
```

This function simply initializes the USART by calling the `setupUSART()` function, then calls the `communicate()` function to “show off”! Finally, Listing 9-15 is the main `loop()` function.

Listing 9-15 USART sketch, loop() function

```
void loop() {
  uint8_t howManyChars;
  char buffer[101];

  // Buffer is one more than we want to receive.
  // Beware of buffer overruns, the code will lose
  // characters as the USART can only store two characters.
  howManyChars = receiveText(buffer, 100);
  sendNumber(howManyChars);
  sendByte('=');
  sendByte('>');
  sendText(buffer);
  sendByte('\n');
}
```

The `loop()` function loops around—that’s its job after all—and receives strings of text from the Serial Monitor. That will need to be configured to add a newline to the end of the sent text, or the code will not print any output until it has received a full buffer of 100 characters of text.

The function just receives text and prints it out, preceded by the number of characters it received. It uses two calls here to `sendByte()` which could, obviously, have been a single call to `sendText()`, but that’s demonstrated in the next line.

If your input text is shorter than the buffer, the preceding example will work fine; if not, there’s a strong possibility that characters will be lost. If, as I did, you send *exactly* 102 characters to a buffer that holds 100, you get two lines of output, the first 100 characters and then the two remaining characters. However, if you send more than 102 characters, you get exactly the same output—the first hundred get copied to the buffer, the next two are still stored internally in the USART, and the rest, sadly, get dropped due to a buffer overrun error.

Welcome to the world of serial communications!

Arduino Paths A

The installation paths, versions, etc., used in the book, relating to the Arduino IDE and the files to be found within that installation, are listed in the following text. Be aware that these paths are valid for a download of the Arduino software as a zip file only. Downloading the installer or, on Linux, running an install using the package manager for your distribution may result in different locations.

My main workstation is Linux based, so most of the paths, etc., in the book refer to that, unless there is a specific need to refer to a Windows file or folder for any reason.

$ARDVERS	is 2.1.0, the version of the IDE.
$LANGVERS	is 1.8.6, the version of the AVR package for my AVR-based boards.
$GCCVERS	is `7.3.0-atmel3.6.1-arduino7`, the version, and directory name, of the GCC compiler package for my AVR-based boards.
$ARDBASE	is the location where the Arduino IDE installed the AVR packages for the Uno and other AVR boards. My location is `/home/norman/.arduino15`.
$ARDIDE	is the location which the Arduino IDE version 2.x created to store settings, caches, and other workspace files. My location is `/home/norman/.arduinoIDE`.
$ARDINST	is the location of the main Arduino files for AVR microcontrollers. This is `$ARDBASE/hardware/arduino/avr/$LANGVERS` and, on my Linux system, expands to `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6`. This is where the various cores, bootloaders, board variants, and so on, are to be found.
$ARDINC	is the location of most of the Arduino Language source files. This location holds the `*.h` header files and most of the `*.c` and `*.cpp` files that comprise the Arduino Language for AVR microcontrollers. This is, on my setup, `$ARDINST/cores/arduino`, which expands to the path `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6/cores/arduino`.
$TOOLS	is where the AVR tools reside in the downloaded packages for the AVR boards. Here, you will find *avrdude* and the AVR Library. On my system, this expands to `/home/norman/.arduino15/packages/arduino/tools`.
$AVRINC	is where the header files for the version of the AVR Library provided by the Arduino IDE are located. The Arduino Language (eventually) compiles down to calling functions within the AVR Library, and the header files for this library

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

are to be found in `$TOOLS/avr-gcc/$GCCVERS/avr/include/avr`. On my system, this equates to `/home/norman/.arduino15/packages/arduino/tools/avr-gcc/7.3.0-atmel3.6.1-arduino7/avr/include/avr`. However, as versions of the *avr-gcc* compiler system change and are implemented into the Arduino toolset, this location will no doubt change.

The following will be helpful on a Linux computer, if you wish to follow the text of the book and view the source code files referred to when I am describing the contents of such files. Listing A-1 is for Linux or MacOS users. Each export command is on a line of its own; there are no continuation lines. The PDF file may be showing wraparound for some of the longer settings, but these are all, as noted, on a single line.

Listing A-1 Shell_exports.sh for Linux and MacOS

```
export ARDVERS=2.1.0
export LANGVERS=1.8.6
export GCCVERS=7.3.0-atmel3.6.1-arduino7
export ARDIDE="${HOME}"/.arduinoIDE
export ARDBASE="${HOME}"/.arduino15"
export ARDINST="${ARDBASE}"/packages/arduino/hardware/avr/"${LANGVERS}"
export ARDINC="${ARDINST}"/cores/arduino
export TOOLS="${ARDBASE}"/packages/arduino/tools
export AVRINC="${TOOLS}"/avr-g++/"${GCCVERS}"/avr/include/avr
```

Setting up the variable is simple:

```
source shell_exports.sh
```

You should, obviously, change paths to suit your name and installation and also after any upgrades to the IDE or the underlying AVR packages. You can now look at files, on Linux, by running this command:

```
view ${ARDINC}/Arduino.h
```

You may, of course, replace *view* with your preferred editor, be it *emacs*, *nano*, or similar. If you have the *xdgutils* package installed, then the `xdg-open` command will open files in the default application:

```
xdg-open ${ARDINC}/Arduino.h
```

ATmega328P Pinout B

Figure B-1 shows the position and names of the pins on an ATmega328P microcontroller. This is only relevant to the 28-pin, through-hole version of the ATmega328.

In Figure B-1, we have various columns representing multiple pin names and functions.

The dark gray area in the center of the image is a representation of the ATmega328P, if you use your imagination that is! At the top and bottom are labels identifying the contents of the appropriate columns.

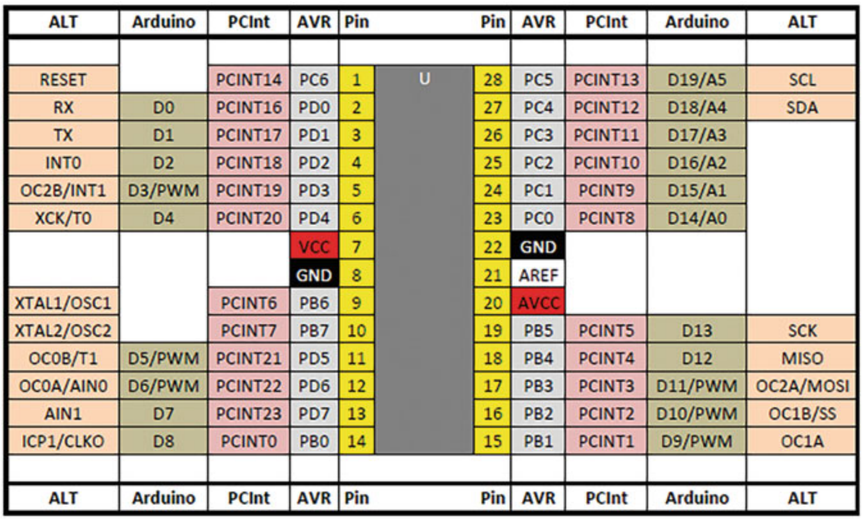
Closest to the ATmega328P are the columns labeled “Pin,” and the numbers in those two yellow-colored columns are the physical pin numbers on the device. The ATmega328P is a 28-pin device.

The next column outward is colored light gray and labeled “AVR” and contains the names of the pins as defined by Atmel. The Arduino uses a different naming standard. When reading the data sheet for the ATmega328P, these are the names that will be used.

The pink-colored columns labeled “PCInt” list the appropriate pin names, again defined by Atmel, to be used when running code that handles pin change interrupts. Here, you see names such as `PCINT0`, `PCINT5`, etc.

The next column outward, labeled “Arduino” and colored olive green, indicates the Arduino pin names which can be used in your sketches. You should be familiar with names like `D0`, `A5`, etc., by now, I hope! Do remember the various Dn pins are not named like that in sketches, they just use the number. Pin `D5` will be specified as just `5` in a sketch. The analog pins `A0` through `A5` do use the A prefix.

Finally, in the columns labeled “ALT,” colored peach, we have the list of alternate functions for a number of the pins. These alternate functions can be enabled using fuses in some cases or can be selected by setting bits in various control registers as necessary.



ALT	Arduino	PCInt	AVR	Pin		Pin	AVR	PCInt	Arduino	ALT
RESET		PCINT14	PC6	1	U	28	PC5	PCINT13	D19/A5	SCL
RX	D0	PCINT16	PD0	2		27	PC4	PCINT12	D18/A4	SDA
TX	D1	PCINT17	PD1	3		26	PC3	PCINT11	D17/A3	
INT0	D2	PCINT18	PD2	4		25	PC2	PCINT10	D16/A2	
OC2B/INT1	D3/PWM	PCINT19	PD3	5		24	PC1	PCINT9	D15/A1	
XCK/T0	D4	PCINT20	PD4	6		23	PC0	PCINT8	D14/A0	
			VCC	7		22	GND			
			GND	8		21	AREF			
XTAL1/OSC1		PCINT6	PB6	9		20	AVCC			
XTAL2/OSC2		PCINT7	PB7	10		19	PB5	PCINT5	D13	SCK
OC0B/T1	D5/PWM	PCINT21	PD5	11		18	PB4	PCINT4	D12	MISO
OC0A/AIN0	D6/PWM	PCINT22	PD6	12		17	PB3	PCINT3	D11/PWM	OC2A/MOSI
AIN1	D7	PCINT23	PD7	13		16	PB2	PCINT2	D10/PWM	OC1B/SS
ICP1/CLKO	D8	PCINT0	PB0	14		15	PB1	PCINT1	D9/PWM	OC1A
ALT	Arduino	PCInt	AVR	Pin		Pin	AVR	PCInt	Arduino	ALT

Figure B-1 ATmega328P pinout diagram

ATmega328P Power Restrictions C

The ATmega328P is limited in the power it can source or sink:

- In total, the ATmega328P can source or sink up to 200 mA maximum.
- On each port, however, only up to 100 mA maximum is allowed.
- On each individual pin, the limit is 40 mA maximum, but 20 mA is the preferred limit.

Warning
Exceeding one or more of the power restrictions will probably damage your device and may render it useless, so take care. If you hear a click, notice a strange smell, and see blue smoke, then like me,[1] you have overdone it!

The limits given mean that you should be thinking of using some form of driver, a transistor or a MOSFET, if you need to drive anything bigger than an LED on each pin.

LEDs, with a 560 Ω resistor in series, run around 6 mA; at least, the red or yellow ones do. Green LEDs have a higher forward voltage, around 3 V in some cases, and draw about 4 mA. However, it's always best to check your own LEDs to be absolutely certain.

A 2N2222 NPN transistor will safely drive up to 1,000 mA and, with a 220 Ω resistor between the Arduino and the base pin, will draw only around 15 mA from the Arduino—well within limits.

C.1 Power in Total

The ATmega328P is restricted to a maximum of 200 mA power, sourced or sunk, in total over all the ports and pins.

C.2 Power per Port

You are advised, in the data sheet, that while each pin within a port can source or sink up to 20 mA, the total current for a single port must be limited to a maximum of 100 mA. This therefore limits each port to a maximum of five pins running at full power. However, on the ATmega328P, there are three

[1] I connected the `VCC` pin to 12 V because my breadboard power supply puts 5 V down the power lines on the left of the breadboard and `VIN` on the power lines on the right side. Bad design!

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

ports, and the power restrictions on the entire chip are limited to 200 mA, so it is not possible to drive the microcontroller at capacity on all three ports.

C.3 Power per Pin

Although each pin can source or sink up to 40 mA, the data sheet warns that this should be restricted to a preferred maximum of 20 mA per pin. Bear in mind that each pin belongs to a port, and ports have their own maximum power limit as does the device as a whole.

Predefined Settings D

The Arduino `init()` function sets up a number of different features of the AVR microcontroller so that your sketch can make best use of the same. These are briefly described in the following text, all in one place, for reference.

D.1 Global Interrupts

Interrupts are enabled globally.

D.2 Timer 0

Timer 0 is configured with a divide by 64 prescaler and in eight-bit Fast Hardware PWM mode to allow `analogWrite()` on pins `D5` and `D6`. The PWM frequency is

$$\begin{aligned} PWM\ Frequency &= F\_CPU/(Prescaler * 256) \\ &= 16{,}000{,}000/(64 * 256) \\ &= 16{,}000{,}000/16{,}384 \\ &= 976.5625\ Hz \end{aligned}$$

The Fast Hardware PWM mode simply counts up from 0 to 255, which is 256 different values, then rolls over to 0 again for the next count up. The data sheet suggests that controlling motor speeds with Fast Hardware PWM isn't the best of ideas. (No, I don't know why either!) This would suggest that if you want to control motors, `D5` and `D6` are not the best pins to be using.

Timer 0 is also set up with an Overflow Interrupt which updates the `millis()` counter via variables `timer0_millis`, `timer0_overflow_count`, and `timer0_fract`. These three variables account for the nine bytes of Static RAM (SRAM) that every sketch uses as a minimum. There are two `unsigned longs` taking up four bytes each and one `unsigned char` using up the final byte.

`Timer0_millis` and `timer0_fract` are used by the `millis()` function, via the Timer 0 Overflow Interrupt ISR, while `timer0_overflow_count` is used by the `micros()` function and, from there, by the `delay()` function.

The Timer 0 prescaler is set to divide the system clock by 64, meaning that the overflow occurs every 1 millisecond plus 24 microseconds. The `millis()` function carefully accounts for this.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

Timer 0 is set to use Fast Hardware PWM as using Phase Correct PWM would have interfered with the Timer 0 Overflow Interrupt, potentially breaking the `millis()` function by giving different values on the ATmega8 and ATmega168/328 devices. The other two timers both use eight-bit Phase Correct PWM.

Warning
If you disable Timer 0 or reconfigure it with a different prescaler value, for example, you will affect `micros()`, `millis()`, and `delay()`, as well as possibly affecting PWM on pins `D5` and `D6`. Beware.

D.3 Timers 1 and 2

Timer 1 is a 16-bit timer, but is configured to provide 8-bit Phase Correct PWM on pins `D9` and `D10`.

Timer 2 is configured to provide eight-bit Phase Correct PWM on pins `D3` and `D11`, and it is indeed an eight-bit timer.

Both timers set the prescaler to divide the system clock by 64. This means that the PWM frequency is

$$\begin{aligned} PWM\ Frequency &= F\_CPU/(Prescaler * 510) \\ &= 16{,}000{,}000/(64 * 510) \\ &= 16{,}000{,}000/32{,}640 \\ &= 490.1960\ Hz \end{aligned}$$

Phase Correct PWM counts up from 0 to 255, then back down to 0, and so on. This is only 510 different values because 0 and 255 are not counted twice—the sequence would be 0, 1, 2 ... 253, 254, 255, 254, 253 ... 2, 1, 0, 1, 2, 3 ... ad infinitum.

According to the data sheet, motors prefer Phase Correct PWM to control their speed, so if you want to control motors, `D3`, `D9`, `D10`, and `D11` are your friends.

Disabling or reconfiguring these timers will affect PWM (`analogWrite()`) on pins `D9` and `D10` (Timer 1) and/or pins `D3` and `D11` (Timer 2).

Timer 1 is also used by the `tone()` function. Using `tone()` in a sketch will affect `analogWrite()` on pins `D9` and `D10`.

Timer 2's ability to run in asynchronous mode, with an external 32 KHz crystal, cannot be used. This is because the Arduino boards come with a 16 MHz crystal attached to the pins that the asynchronous timer mode needs, plus the ATmega328P has its fuses set to disable the calibrated internal RC oscillator so that the external one with the 16 MHz crystal can be used.

D.4 USART

The USART, which was enabled by the bootloader, is subsequently disabled by the `init()` function, allowing pins `D0` and `D1` to be used as digital pins in the normal manner. This therefore requires that pins `D0` and `D1` be reconfigured for the USART when serial communications are required. This is automatically carried out by calling `Serial.begin()` in a sketch.

Note
Some other AVR microcontrollers, notably the Mega 2560, have multiple USARTs, but the ATmega328P only has one, and it uses these two pins for communicating with the outside world.

D.5 Analog-to-Digital Converter

The ADC is configured with a prescaler of 128 which divides the system clock by 128 to get an ADC clock frequency of 125 KHz, which is within the desired range of between 50 and 200 KHz. In addition, the ADC is enabled and powered on in every sketch, whether or not it is used.

You can disable the ADC and power it down to save a few microamps, if it is not required in your sketch, by adding the code in Listing D-1 to your `setup()` function.

Listing D-1 Disable and power down the ADC

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

void setup() {
    // Disable ADC if not used.
    ADCSRA &= ~(1 << ADEN);

    // Power off ADC.
    power_adc_disable();

    // Rest of setup() goes here...
    ...
}
```

ADC Temperature Measuring

E

Some AVR microcontrollers have an internal temperature measuring device, which can be selected to be used as an input to the ADC and used to query the actual running temperature of the AVR microcontroller itself. This is not the temperature of the air around the Arduino board/AVR microcontroller, it is the temperature of the AVR microcontroller itself. Measuring the air temperature requires some kind of external temperature sensor.

This internal sensor is not accessible directly from the Arduino Language, but with a little effort, it can be done. The data sheet for the ATmega328P states that

> If the user has a fixed voltage source connected to the `AREF` pin, the user may not use the other reference voltage options in the application, as they will be shorted to the external voltage. If no external voltage is applied to the `AREF` pin, the user may switch between `AVCC` and 1.1 V as reference selection.

This means *don't* connect a voltage source to the `AREF` pin if you are going to set up the ADC to use any of the two internal reference voltages. If you do, your AVR microcontroller will possibly allow the magic blue smoke out and will stop working. The `analogReference()` function allows you to set the reference voltage source, but it's not until you call `analogRead()` that it gets configured.

Bearing the preceding warning in mind and looking at the schematics, the Arduino Duemilanove and the Uno Revision 3 do not normally have `AREF` connected to a voltage source. There is, however, a location on one of the headers[1] labeled "AREF" where you can supply a voltage to the `AREF` pin. This is limited to a maximum of 5.5 V and shouldn't be higher than the supply voltage.

As previously discussed, the ATmega328P has an on-chip temperature sensor. This can be read using channel eight of the ADC and returns a value which represents the temperature in degrees Kelvin (°K). Kelvin is similar to Centigrade (°C), but is offset by 273 degrees, so 0 °C is 273 °K, and 0 °K is the point at which all atomic movement stops and is exceedingly cold!

In order to set up the ADC to measure the AVR microcontroller's own temperature, you need to configure certain registers, as follows:

- The ADC reference voltage must be configured to use the internal 1.1 V reference by setting bits `REFS1:0` to 0b11 in register `ADMUX`. You cannot use any other reference voltage for internal temperature measurements using the ADC. However, to use the internal 1.1 V reference, you must not have the external `AREF` pin connected to any source of voltage.
- Register `ADMUX`, bits `MUX3:0`, must be set to 0b1000 to enable the temperature sensor as the ADC input source.

[1] At least, there is on my Uno R3 clone.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

- As with all ADC measurements, the ADC clock must be in the range 50–200 KHz. The Arduino runs with a 16 MHz crystal attached, so the system clock speed is far too high. In order to reduce the clock speed, the ADC prescaler must be set so that the system clock is divided down by a suitable amount to get the ADC clock into the required range. Dividing by 128 will give a value of 125 MHz on a board running at 16 MHz, so the `ADCSRA` register bits `ADPS2:0` should be set to 0b111 to achieve this.
- Register `ADCSRA`, bits `ADEN` and `ADSC`, should both be set to 1 to enable the ADC and to automatically start the first measurement.

In the example code in this appendix, the ADC Noise Reduction settings are not being used. See the data sheet or Section 7.4.1.3 for details if you wish to enable that feature. It's a special sleep mode which powers everything down, apart from the ADC, in order to reduce interference which might cause the ADC readings to be inaccurate.

In Listing E-1, the `setup()` function carries out all of the initialization steps already mentioned. The constant, `MOVING_AVERAGE_LIMIT`, defines the number of temperature readings that will be averaged out to give a more meaningful reading. You can change this at your pleasure.

Listing E-1 Initializing the ADC for temperature measurements

```
const int MOVING_AVERAGE_LIMIT = 100;

void setup() {
    // Initialise the ADC to use the
    // internal 1.1V reference voltage.
    ADMUX = (1 << REFS0) | (1 << REFS1);                  (1)

    // Use the ADC multiplexer input
    // number 8, the temperature sensor.
    ADMUX |= (1 << MUX3);                                 (2)

    // Slow the ADC clock down to 125 KHz
    // by dividing by 128. Assumes that the
    // standard Arduino 16 MHz clock is in use.
    // Comment out to use NormDuino.
    ADCSRA = (1 << ADPS2) | (1 << ADPS1) |                (3)
             (1 << ADPS0);

    // Non-standard 8MHz clock in use.
    // Uncomment to use for NormDuino.
    //ADCSRA = (1 << ADPS2) | (1 << ADPS1) |              (4)
               (0 << ADPS0);

    // Enable the ADC and discard the first reading as
    // it is always 351 on my device.
    ADCSRA |= (1 << ADEN) | (1 << ADSC);                  (5)
    (void)readADC();

    // Use the Serial monitor for output.
    Serial.begin(9600);
    Serial.println("Arduino Internal Temperature");
}
```

(1) This sets the internal 1.1 V bandgap as the ADC's reference voltage. This is mandatory when using the temperature sensor. You must make sure that there are no external voltages on the `AREF` pin of the Arduino, or else this setting lets the magic blue smoke out and causes the device to stop working. A 100 nF capacitor between `AREF` and `GND` is acceptable, but connecting to `AREF` to 5 V is not.
(2) This use of a reserved value for the `ADMUX` register is the one that selects the internal temperature sensor as the ADC input source.
(3) I have an Arduino Duemilanove, an Uno, and a few homemade "NormDuinos," and I need to uncomment one of these two lines depending on which board I'm using. My NormDuinos run at 8 MHz, while the Arduino boards run at 16 MHz. The main clock frequency needs to be divided down to get it into the range of 50 to 200 KHz.
(4) This is the line for my 8 MHz devices.
(5) I've noticed that the first reading from the ADC is always a bit weird, at least, when using the temperature sensor. These lines start the ADC conversion and wait for completion before throwing away the result

The `readADC()` function called from `setup()`—to throw away the first reading—and on each pass through the `loop()` can be seen in Listing E-2.

Listing E-2 Reading the temperature

```
// Read the ADC result from the most recent conversion and
// start another before returning the current reading.
uint16_t readADC() {
    // Make sure the most recent ADC read is complete.
    while ((ADCSRA & (1 << ADSC))) {                  (1)
        ; // Just wait for ADC to finish.
    }

    uint16_t result = ADCW;                           (2)

    // Initiate another reading.
    ADCSRA |= (1 << ADSC);                            (3)

    return result;
}
```

(1) The previous call to `readADC()` initiated a new read request, so these lines simply make sure that the request has completed and a reading is available. The `ADSC` bit is used to start a conversion and remains set to 1 until the conversion finishes, whereupon it is cleared.
(2) We grab the result from `ADCW` which takes care of reading the high and low bytes of the ADC result in the correct order. This ensures that we get the correct value read and not some intermediate result.

(3) Prior to returning the result just read from the sensor, this line initiates a new reading. The temperature sensor can only be read in single shot mode; there's no opportunity to initiate a free running mode with the sensor.

Listing E-3 shows the code that calculates the moving average of `MOVING_AVERAGE_LIMIT` ADC readings. The code in the function is based on a description of moving averages at https://en.wikipedia.org/wiki/Moving_average. You can uncomment the debugging lines, where indicated, if you would like to see the moving average details as each new data point is added to the moving average so far. The code resets the moving average after `MOVING_AVERAGE_LIMIT` ADC readings.

Listing E-3 Averaging the temperature readings

```
float movingAverage(uint16_t data) {
    // https://en.wikipedia.org/wiki/Moving_average

    // How many data points so far make up the average.
    static float pointCount = 0.0;

    // Current moving average of the pointCount data points.
    static float movingAve = 0.0;

    // Uncomment for a running commentary.
    /*
    Serial.print("PointCount = ");
    Serial.print(pointCount);
    Serial.print(", movingAve = ");
    Serial.println(movingAve);
    */

    movingAve = ((float)data +
                 (pointCount * movingAve)) /
                 (pointCount + 1.0);
    pointCount += 1.0;

    // Are we done for this running average?
    if ((int)pointCount >= MOVING_AVERAGE_LIMIT) {
        pointCount = 0.0;
        movingAve = 0.0;
    }

    return movingAve;
}
```

The loop() function in Listing E-4 gathers a running average of 100 readings from the ADC and then converts the final total to °C which is then printed to the Serial Monitor.

Listing E-4 Displaying the temperature

```
void loop() {
    // Current ADC reading.
    uint16_t ADCReading;

    // Current moving average.
    float ADCAverage;

    // Calculate a moving average over 100 readin          (1)
    for (uint8_t n = 0; n < MOVING_AVERAGE_LIMIT; n++) {
        // Fetch ADC reading n& average it.
        ADCReading = readADC();
        ADCAverage = movingAverage(ADCReading);            (2)
    }

    // Print the ADC temperature.
    float degreesC = (ADCAverage - 324.31) / 1.22;         (3)
    Serial.print(degreesC);
    Serial.print("C, ");

    // Convert to Fahrenheit. C * 1.8 + 32.
    Serial.print(degreesC * 1.8 + 32);                     (4)
    Serial.println("F.");

    // Delay a second more between readings.
    delay(1000);
}
```

(1) A running average of 100 readings, or whatever `MOVING_AVERAGE_LIMIT` has been set to, from the sensor is calculated here. This avoids most of the weirdness that can sometimes arise, especially as the code is not using the ADC Noise Reduction sleep mode to try and get better readings. Whatever else is happening on the device might be affecting the readings.
(2) Here, we add the latest `ADCReading` into our moving average so far.
(3) I've decided to use this method of converting the ADC reading from something related to °K to °C. This method is closest to my reality and my measured temperatures in the office.
(4) And for those who like their temperatures in "old money," this line converts °C to °F.

According to the Internet, example documents from Atmel, Microchip, and elsewhere, there are many ways to calculate the internal temperature from the reading returned by the ADC. Here are some that I've come across:

1. *ADC-some random offset*, which is different for every device.
2. *(ADC-247)/1.22*
3. *ADC-273*: This one looks promising as it corresponds to converting °K to °C.
4. $(((ADC - (273 - 100 - TS_OFFSET)) * 128)/TS_GAIN) + 25$: Sadly, getting the `TS_OFFSET` and `TS_GAIN` is not simple. The documents mentioned in the following warning have the details.
5. $(ADC - 324.31)/1.22$

I'm using the final method, as it's the one closest to my actual temperature measurements.

Each time around the `loop()`, the ADC has to be coaxed into running another conversion, so bit `ADSC` in register `ADCSRA` is again set to trigger another reading.

Converting to Fahrenheit (°F), for those who still measure temperature on that particular scale, is done usually by multiplying °C by 1.8 and adding 32, so that's what happens next, and the resulting temperature is printed out to the Serial Monitor. Using `float` data types in a sketch pulls in a lot more code than normal, so it's best, if possible, to avoid them—unless you have lots of spare Flash RAM, of course.

Warning

According to the data sheet, uncalibrated sensor data is accurate to plus or minus 10 °C. That's quite a range of possible temperatures then. The Application Note at http://ww1.microchip.com/downloads/en/AppNotes/Atmel-8108-Calibration-of-the-AVRs-Internal-Temperature-Reference_ApplicationNote_AVR122.pdf shows how the device can be calibrated, and even after that, there's only a possible best accuracy of plus or minus 2 °C .

Assembly Language—Briefly F

I know I said that there wouldn't be any assembly language code in the book, but I did say *probably wouldn't*. I am, however, writing a new book, *Arduino Assembly Language*, which covers the subject in some detail—assuming it eventually gets published, of course!

This appendix is a very brief introduction to Arduino assembly language, which is supported in the IDE now, whereas it wasn't all that many versions ago that you had to make some changes to the IDE source code and recompile in order to get assembly code noticed.

One thing you must remember is assembly source files have the extension ".S" in capitals. If you have the ".s" in lowercase, it won't compile as the file won't be found.

Open the Arduino IDE and create a new sketch, call it `BlinkASM.ino`, and immediately save it. Edit the sketch to make it resemble the code in Listing F-1.

Listing F-1 Arduino BlinkASM sketch

```
// Make sure the compiler can find 'blink' and 'setup' which
// are written elsewhere in assembly language.
extern "C" {                                              (1)
    void setup();
    void blink();
}

// There is no setup() here! It is called from the
// assembly file automatically.                           (2)
void loop() {
    blink();                                              (3)
    delay(1000);
}
```

(1) This is how we tell the IDE that we have `void` functions named `blink()` and `setup()`, which take no parameters and which are written in a language other than C++. We have to use `"C"` to make sure that the compiler doesn't attempt to do any C++ "name mangling" of function names and/or parameter names.
(2) There is no `setup()` function in this sketch file. It is defined in the assembly file, and as it is `global`, the Arduino-defined `main()` function can call it quite happily, even though it is written in assembly language.
(3) This is where we call the assembly language function `blink()` to toggle the LED on pin `D13`.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

So far, so good? Nothing to be afraid of here. Next, though, we have the assembly language code itself.

- Add a new tab to the opened sketch in the IDE. There's a "· · ·" toolbutton on the far right of the IDE's tab bar, where the tab for `BlinkASM.ino` is currently showing. Click "· · ·" and choose "New Tab."
- Enter the name `BlinkASM.S` in the file name prompt area at the bottom of the screen and click OK. Make sure the file's extension is "S" in uppercase.

Now enter the assembly language code in Listing F-2 into the new tab.

Listing F-2 Arduino assembly language

```
#define __SFR_OFFSET 0                                  (1)
#include <avr/io.h>                                     (2)

.section .text                                          (3)
.global blink                                           (4)
.global setup

blink:                                                  (5)
    // digitalWrite(13, !(digitalRead(13)));            (6)
    ldi r18, (1 << PORTB5) ; PORTB5 = Arduino D13
    out PINB, r18          ; Toggle PORTB5 = D13
    ret

setup:                                                  (7)
    // pinMode(13, OUTPUT);                             (8)
    ldi r18,(1 << DDB5)    ; R18 = Arduino D13
    out DDRB,r18           ; D13 is an OUTPUT pin now
    ret
```

(1) Apparently, a hack! But this makes sure we get the correct offset for the `PINB` and `PORTB` names used below. AVRs are weird and these things—I/O Registers[1]—have two addresses in memory. (Technically incorrect, but pretty much accurate!)
(2) This fetches the correct header file for the device we are using. This means we can refer to `PORTB` and `PINB` by name and not as a number.
(3) Code, mostly, lives in the `.text` section. We need our code to live there too.
(4) As we wish to call `blink()` and `setup()` from other files, our sketch in particular, we need to make the entry point for the two functions globally visible.
(5) This is the entry point to the `blink()` function.
(6) This is the Arduino Language equivalent to the two lines of assembly code that follows, ignoring, of course, the `ret` instruction which simply terminates the `blink()` function.
(7) This is the entry point to the `setup()` function.

[1] Also known as SFRs, or Special Function Registers. These control the timers, ADC, Analog Comparator, USART, and other internal hardware of the ATmega328P.

(8) This is the Arduino Language equivalent to the following two lines of assembly code, again, ignoring the `ret` instruction.

Now compile the sketch. You can do this from either tab in the IDE. You should see the usual text scrolling up the screen—if you have verbose compilation messages enabled in the File ▹ Preferences dialog. At the very end of the compilation, you should see the usual details on the size of the sketch, etc. Mine used 612 bytes of Flash RAM and the usual nine bytes of Static RAM. This is another reduction in the size of the blink sketch, and we are still using the `delay()` function from the Arduino Language.

You should now be able to upload the sketch to your board and be amazed as the built-in LED starts blinking, yet again!

Obviously, this is a really minimal example, and unless you really need the speed and space reductions of code written in plain AVR assembly, then you should not often need to resort to assembly language, but at least it's now available in the IDE.

You do still need a `sketch_file.ino` to be able to use the IDE for assembly language; that factor hasn't yet gone away—you cannot, as yet, write a complete sketch in assembly language. Well, you can but you would be using *AVR Studio* or *PlatformIO* to do it. The Arduino IDE or *arduino-cli* both require a sketch file.

Just for your interest, the assembly language in this sketch will be assembled by the *avr-as* application supplied with the Arduino IDE, but hidden from you! The reason there are 612 bytes of Flash RAM used is because of all the startup code, interrupt handlers, and initialization code that the compiler system builds into your sketches. If this was written in pure assembly, it would reduce further to only 30 bytes. That's not a misprint, it assembles to 15 words or 30 bytes in total.

Smallest Blink Sketch?

G

OK, here's one *final* blink sketch, and could this one be the smallest we can get, I wonder? It requires the PlatformIO system as the Arduino IDE or command line wraps too much extraneous code around an assembly language module. The code in Listing G-2 compiled to 478 bytes with 9 bytes of SRAM used in the Arduino IDE and also with the *arduino-cli* utility. In *PlatformIO*, it's 162 bytes in total with no bytes of precious SRAM used.

There is, however, a neat trick that allows the Arduino IDE to compile to exactly the same size as PlatformIO, which I will explain later.

Here's the process:

```
mkdir SmallestBlink
cd SmallestBlink
pio init --board uno
```

Listing G-1 shows what the `platformio.ini` file should look like for an Arduino Uno build. The `framework = arduino` line must be removed, or PlatformIO will compile all the usual Arduino code into the finished file. I had to add in the `upload_port` line as the code wouldn't upload without it, and a helpful error message advised me what to do. However, this doesn't appear necessary with more recent versions of PlatformIO. (My Uno clone has an FTDI chip.)

Listing G-1 The platform.ini file

```
[env:uno]
platform = atmelavr
board = uno
upload_port = /dev/ttyUSB0
```

We now need to determine the flash rate for a 16 MHz Arduino Uno running at a blink rate of 1 Hz with a divide by 256 prescaler. The formula for the *frequency* of a Timer 1 in CTC mode is

$$Frequency = F\_CPU/((2 * prescaler) * (1 + OCR1A))$$

This can be rearranged to give the formula to find the required value for register `OCR1A`:

$$OCR1A = ((F\_CPU/(2 * prescaler))/Frequency) - 1$$

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

This gives

$$\begin{aligned} OCR1A &= ((F\_CPU/(2 * prescaler))/Frequency) - 1 \\ &= ((16{,}000{,}000/(2 * 256))/1) - 1 \\ &= (16{,}000{,}000/512) - 1 \\ &= 31{,}250 - 1 \\ &= 31{,}249 \end{aligned}$$

This is the value for register `OCR1A`.

Type in the code shown in Listing G-2 into the file `src/SmallestBlink.S`.

Listing G-2 The smallest blink sketch?

```
#define __SFR_OFFSET 0                                     (1)
#include <avr/io.h>

.section .text
.global main


#define FLASH_RATE 31249


main:
    ; Set up Timer 1:
    ldi r18, (1 << COM1A0)
    sts TCCR1A, r18                                        (2)

    ldi r18, hi8(FLASH_RATE)                               (3)
    sts OCR1AH, r18
    ldi r18,lo8(FLASH_RATE)
    sts OCR1AL, r18

    ; Finish setup of Timer 1.
    ldi r18, ((1 << WGM12)|(1 << CS12))                    (4)
    sts TCCR1B, r18

    ; Set PB1/D9 as output after timer initialization.
    ldi r18, (1 << DDB1)
    out DDRB, r18                                          (5)

loop:
    rjmp loop                                              (6)
```

(1) This is a hack! It allows me to use `DDRB` instead of having to wrap every I/O register name in the `__SFR_IO_ADDR` macro.
(2) This configures the timer to toggle the `PB1/D9` pin every time we hit the value in `OCR1A`.
(3) These four lines load the value 31,249 into the `OCR1A` register. It must be done in two eight-bit chunks, and the high byte *must always be written first*.
(4) The timer is now set to run with a divide by 256 prescaler and CTC mode (Clear Timer on Compare). When the timer's value reaches that in `OCR1A`, the timer's value resets to zero and the `PB1` pin will toggle, thanks to (2).
(5) This sets pin `PB1/D9`, as `OUTPUT`. The data sheet advises that we do this after initializing the timer.
(6) This is effectively like an empty `loop()` in a sketch; it does nothing except burn CPU cycles looping around itself.

Warning
As noted earlier, when writing 16-bit values to important registers, you must note the advice in the data sheet to load the bytes in the order given. When you write the high byte, it is stored in a temporary register. When the low byte is then written, both bytes are written to the register as one operation. This prevents the possibility of a "split" value in the register.

When reading 16-bit registers like `OCR1A`, you read the low byte first, then the high byte to get the correct value.

Save the file and exit the editor. Run a test compile:

```
pio run
```

There should be no errors; now upload the code:

```
pio run -t upload
```

There should be an LED connected to pin `PB1/D9` on the Uno and connected to ground through an appropriate resistor—I used a 330 Ω resistor on a 5 V setup, giving a maximum current of

$$\begin{aligned} I &= (5\ V - 1.8\ V)/330 \\ &= 3.2\ V/330 \\ &= 9.69\ mA \end{aligned}$$

To correctly calculate the current drawn, you must subtract the forward voltage of the LED from the supply voltage. A red LED has a forward voltage of around 1.8 V—at least, according to my data sheet, it has—but this is an acceptable default for a red LED. The resulting current should be within the maximum limits as discussed in Appendix C, and 9.69 mA is well within the limit of 20 mA per pin.

The LED should be blinking away merrily at around 1 Hz, or 1 flash per second, but it seems to be running faster—why? Figure G-1 shows the oscilloscope trace on the Uno's pin `PB1/D9` while the code was running.

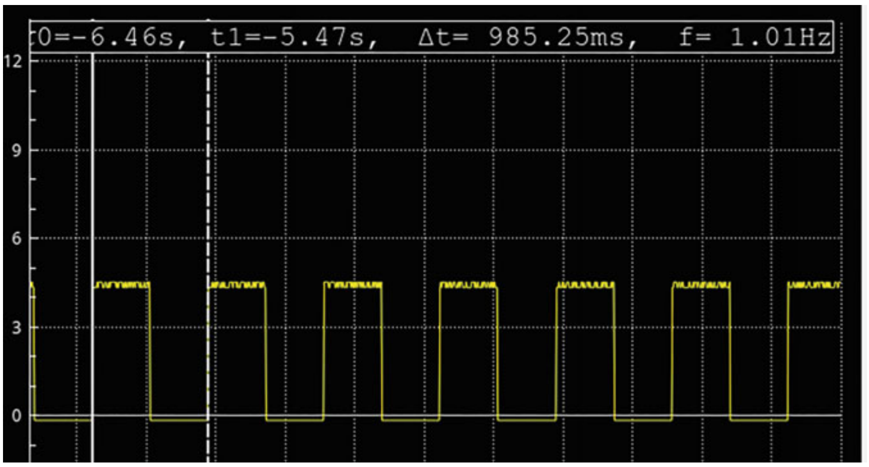


Figure G-1 SmallestBlink oscilloscope trace

It clearly states in the upper-right corner that the frequency is 1 Hz. However, while that is indeed true, it does mean that because the waveform is `HIGH` for 50% and `LOW` for the other 50% of that time, hence the LED is actually flashing every half second. Oops! To get a one-second flash time, I need to calculate a 0.5 Hz frequency.

Edit the code in Listing G-2 to change one setting:

```
#define FLASH_RATE 62499
```

Recompile and upload, and finally the LED is flashing once per second. I can check my calculations by feeding the value I recalculated back into the equation for frequency:

$$\begin{aligned} Frequency &= F\_CPU/((2*prescaler)*(OCR1A+1)) \\ &= 16{,}000{,}000/((2*256)*(62499+1)) \\ &= 16{,}000{,}000/(512*62{,}500) \\ &= 16{,}000{,}000/32{,}000{,}000 \\ &= 0.5\ Hz \end{aligned}$$

So, that seems to be correct. The oscilloscope now reads 504 mHz in the upper left, so that's pretty close to 0.5 Hz. I'm happy with that. The "mHz" stands for millihertz.

Tip
The best thing about this blink sketch is that Timer 1 is doing all the blinking by itself. The CPU is doing nothing except burning cycles in a tight loop. It would be entirely possible to put the ATmega328P to sleep to reduce power or to actually have it do "something useful" with its time.

G.1 Doing It in the Arduino IDE

I mentioned a trick that would allow the Arduino IDE to compile code to the same size as PlatformIO. This is what you do:

- Create a new sketch in the normal manner.
- In the editor, click Edit ▷ Select All.
- Press the DELete key.
- You should now have a completely blank file. Save this as `SmallestBlink.ino`.
- Click the "· · · " toolbar icon, and select "New Tab."
- Name the new tab as `SmallestBlink.S` with an uppercase ".S" extension.
- Type the code from Listing G-2 into the new tab.
- Save the sketch again.
- Compile and upload.

You will notice that this time the Arduino IDE has compiled to exactly the same size as the PlatformIO version discussed earlier. The LED on pin `PB1/D9` will be flashing.

G.2 Actually . . .

Actually, there is a way to reduce the code in Listing G-2 right down to only 30 bytes!

In order to do this, you need to use a "free standing" AVR assembler. The *avr-gcc* compiler does have an assembler built in, *avr-as*, but that always creates a lot of extra content in the generated `.hex` file. The AVR assembler will not do this, unless you instruct it to do so.

Assembling Listing G-3, which is a very minor rearrangement of Listing G-2 to comply with the *gavrasm* assembler, available from www.avr-asm-tutorial.net/gavrasm/index_en.html at no cost, resulted in 30 bytes total size.

G.2.1 Assembling the Source

The source file on disk has far more comments than I have reproduced here and includes instructions on how to assemble and upload the code using either an Arduino bootloader or an ICSP device, specifically the USBTinyISP.

The command to assemble the source, which assumes that you have downloaded and installed *gavrasm* somewhere on your `PATH`, is

```
gavrasm Timer1_LED_blink.asm
```

G.2.2 Uploading

In order to upload the code, you will need a copy of *avrd*ude and a configuration file, `avrdude.conf`. The files supplied by the Arduino IDE can be used; they are in `$TOOLS/avrdude/bin` and `$TOOLS/avrdude/etc`. I have copied[1] *avrdude* and `avrdude.conf` from my Arduino IDE installation into a directory on my path to make it easy to use without having to type the entire path to the Arduino installation. The steps to obtain those two files are

- Open the Arduino IDE and any sketch at all; I used the Blink example sketch under File ▹ Examples ▹ 01.Basics ▹ Blink.
- Click File ▹ Preferences and make sure that verbose mode is set for uploading sketches, then click OK.
- Compile the sketch.
- Upload the sketch.

In the output log at the bottom of the screen, the last command line listed is the one to execute *avrdude*; it will resemble the following, but all on a single line:

```
"/path/to/arduino/tools/avrdude/bin/avrdude" \
"-C/path/to/arduino/tools/avrdude/etc/avrdude.conf" \
...
```

This tells me where I can find the Arduino-specific version of *avrdude* and its Arduino configuration file. I use a local `bin` directory under my `$HOME` directory, so copy, or symbolically link, those two files into `/home/norman/bin`:

```
cp /path/to/arduino/tools/avrdude/bin/avrdude ~/bin
cp /path/to/arduino/tools/avrdude/etc/avrdude.conf ~/bin
```

Now, I am able to run the Arduino-specific version of avrdude without having to remember the full path, and because the `avrdude.conf` file is in the same location as the binary, I don't have to specify it when executing the *avrdude* utility.

Once Listing G-3 has been assembled and is error free, it can be uploaded using the Arduino bootloader easily:

```
avrdude -p m328p -c arduino -P /dev/ttyUSB0 \
    -U flash:w:Timer1_LED_blink.hex
```

As ever, this should all be on a single line with no breaks. I have had to break the line to fit the page, hence the Linux continuation character, the "\", at the end of the first line.

[1] Not really copied, just sym-linked.

If you use a USBtinyISP, then the command to upload the code using that device is similar:

```
avrdude -p m328p -c usbtiny -U flash:w:Timer1_LED_blink.hex
```

Listing G-3 shows the source code for the assembler to use. I have, as mentioned, removed most of the comments for brevity.

Listing G-3 The ultimate blink sketch?

```
.DEVICE ATmega328P

.CSEG
.ORG 0

.equ FLASH_RATE = 62499

main:
    ; Timer1 is CTC, Toggle PB1
    ldi r18,(1<<COM1A0)
    sts TCCR1A,r18
    ldi r18,((1 << WGM12) | (1 << CS12))
    sts TCCR1B,r18

    ; OCR1A is FLASH_RATE and we must write to
    ; OCR1AH before OCR1AL or else.
    ldi r18,high(FLASH_RATE)
    sts OCR1AH,r18
    ldi r18,low(FLASH_RATE)
    sts OCR1AL,r18

    ; Pin PB1/D9 is OUTPUT
    ldi r18,(1 << DDB1)
    out DDRB,r18

loop:
    ; The loop does nothing here, but it could!
    rjmp loop
```

And just to show my workings—as my maths teachers used to say—here is the output from *gavrasm* when the code in Listing G-3 was assembled. Fifteen words is 30 bytes.

```
    $ gavrasm Timer1_LED_blink.asm
```

```
+------------------------------------------------------------+
| gavrasm gerd's AVR assembler Version 4.9 (C)2020 by DG4FAC |
+------------------------------------------------------------+

Compiling Source file: Timer1_LED_blink.asm
-------
```

```
Pass: 1
84 lines done.

Pass 1 ok.
-------
Pass: 2
84 lines done.

Warning 001: 1 symbol(s) defined, but not used!

15 words code, 0 words constants, total=15 = 0.1%

One warning!
Compilation completed, no errors. Bye, bye ...
```

That's it! No more blink sketches and definitely no more assembly language!

NormDuino

H

You can build your own “Arduino” on a breadboard and play with it there, if you feel the need. I have done this for some of the experiments in this book—just to be sure that things worked perfectly whether I was running a full speed, 16 MHz, Arduino board like my Duemilanove or my Uno, or a breadboarded experimental system running at 8 MHz on the internal oscillator.

This is the sort of thing you might find interesting; after all, there’s no point having a fully blown Arduino board, with all the attendant paraphernalia such as USB communications, always-on power LED, etc.—things that can waste power when they are not needed in your finished project. It’s great to use the Arduino for prototyping, but when you go into “production” you only need a handful of components.

If you look at the the schematics in Figure H-1, you will notice that there’s a diode between the board and the battery. This is for two reasons: it drops 0.7 V of the incoming 6 V battery supply, taking the voltage down to 5.3 V and within the range of the ATmega328P’s 5.5 V maximum, and it prevents damage if you connect the battery the wrong way around.

A 16 MHz crystal and a pair of 22 pF capacitors will be required if you can’t get hold of an ATmega328P *without* the Uno bootloader already burned in. It is quite simple to burn a new bootloader so that the ATmega328P doesn’t need the 16 MHz crystal, etc. It is, however, a bit of a faff (that’s a technical term) as you have to create a new entry in `boards.txt` and burn an 8 MHz bootloader and reset a couple of fuses along the way. This is the sort of thing that requires an ICSP device, although you could use an existing Arduino as the ICSP.

So, if you have an ATmega328P with an Uno bootloader, you will need the crystal and the capacitors plus a 6 V supply (which drops to 5.3 V thanks to the diode), and if you get an ATmega328P without an Uno bootloader, you will be able to run it at 3 V without the crystal and capacitors, but 4.5 V will be better as the diode will drop that to 3.8 V which is ample.

Figure H-1 is the schematic layout for what I’m calling NormDuino. You can see that it is quite simple; all it needs is a power supply, a spare ATmega328P, a couple of resistors, a switch, a diode, and a handful of capacitors.

As you can see, there’s nothing to it. You will note, I hope, that the `AREF` pin is not connected to anything other than a 100 nF capacitor to smooth out the power if you decide to connect it to `VCC`. For safety, and to avoid letting the magic blue smoke out, it’s best to heed the warnings in the data sheet and do not set the ADC reference voltage to the internal one if you have the `AREF` pin connected to an external power source.

By not connecting it in the breadboard setup, you avoid this problem and can safely use the internal 1.1 V reference for the Analog Comparator and/or the ADC.

Figure H-2 shows how it looks when laid out on a half-sized breadboard, with power lines running down both sides.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

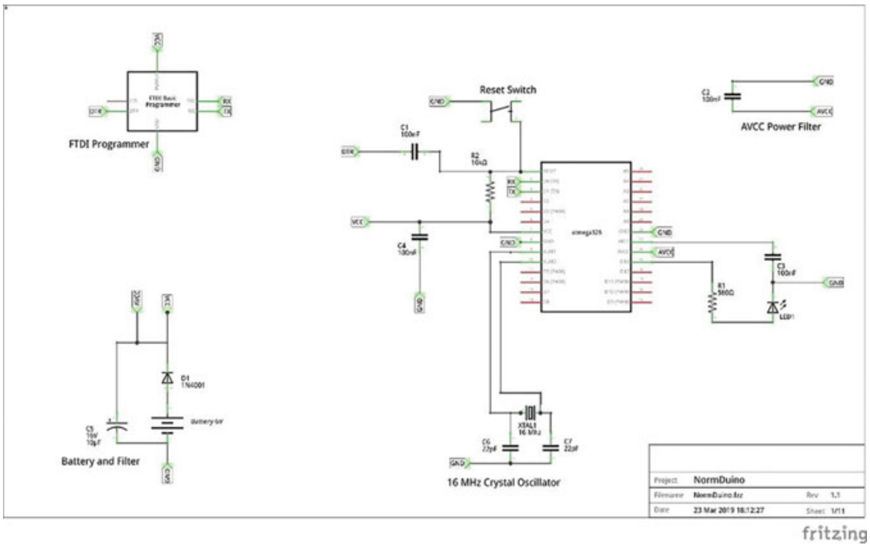


Figure H-1 NormDuino schematic

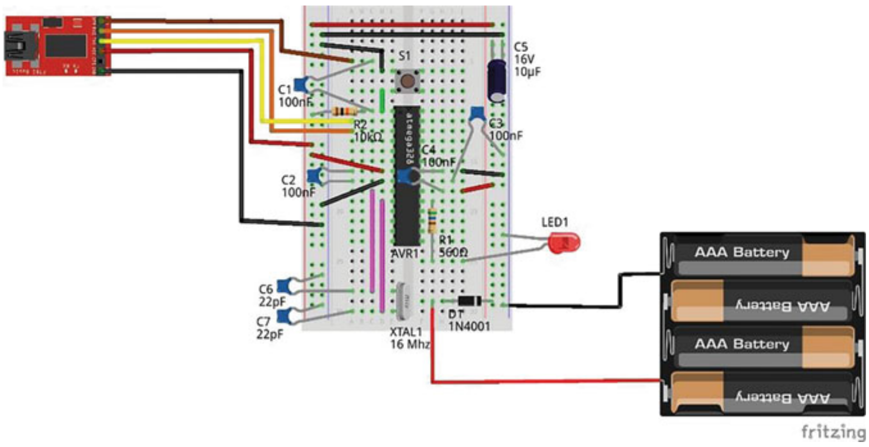


Figure H-2 NormDuino breadboard layout

As mentioned earlier, the crystal (XTAL1), two capacitors (C6 and C7), and the two wires, at the bottom left of the breadboard, may be omitted if you have an ATmega328P without an Uno bootloader programmed in. It will, in the factory default settings, be configured to run off of the 8 MHz internal oscillator with a divide by eight prescaler, giving your breadboard Arduino a top speed of a whopping 1 MHz, but that can be changed.

When done and tested on the breadboard, you wouldn't want to put the breadboard into a box and use that in your finished project, so you can purchase copper clad board that is designed to resemble a breadboard, and it's an easy task then to move each component from the breadboard to the copper board and solder it in place. You would probably need or wish to add five pins to the board as well to allow the FTDI access to the ATmega328P, in order to reprogram it, should this ever be necessary.

Alternatively, some normal stripboard can be used, and just break each track up the middle and you have your own, handmade breadboard on a copper clad board. I have two NormDuinos that I built in this way, one on a copper breadboard layout and the other on a simple stripboard.

Note

Some FTDI devices have their `TX` and `RX` pins labeled the other way around. The ATmega328P's physical pin 2 is its `RX` pin and that needs to be connected to the programmer's `TX` pin. The ATmega328P's physical pin 3 is its `TX` pin and that needs to be connected to the programmer's `RX` pin. On my FTDI, the pins are labeled `RXO` and `TXO` (RX and TX Output), and those need connecting to the ATmega328P's corresponding input pins, so `RXO` goes to `TX` and `TXO` goes to `RX`. Your device, of course, may well differ, so if it doesn't work, just swap the wires over and try again—you won't damage anything if the wires are crossed.

Confused? I was too; when I first started using the FTDI, I couldn't get it to see the ATmega328P, so it couldn't program it. In the end, swapping the wires over solved the problem.

Welcome to the world of old-style serial communications. Nowadays, the pins are connected like to like—for example, `MOSI` connects to `MOSI` and `MISO` connects to `MISO`—there's a lot less confusion that way.

Your project is now complete. The Bill of Materials (BOM) for the NormDuino is shown in Table H-1.

If your ATmega328P came with an Uno bootloader, then you can use an FTDI device or an ICSP to program it. If the microcontroller came without a bootloader, and you don't have an ICSP of your own, you could use an existing Arduino as an ICSP, as detailed in Appendix I, which is coming next.

Tip

If you would like step-by-step instructions on building a 16 MHz version of the Arduino, on a breadboard, similar to NormDuino, check out the Arduino tutorial at https://docs.arduino.cc/hacking/hardware/building-an-arduino-on-a-breadboard for details. However, that link appears to be currently broken. It is referenced from a tutorial on the same subject at https://docs.arduino.cc/built-in-examples/arduino-isp/ArduinoToBreadboard/, which *is* still present.

Table H-1 BOM for a single NormDuino

Item	Qty	Label	Purpose
ATmega328P	1	AVR1	The brains!
560 Ω resistor, 1/4 watt	1	R1	For the LED on D13, pin 19
10 KΩ resistor, 1/4 watt	1	R2	Pull-up resistor on RST pin, pin 1
Red 5 mm LED	1	LED1	Built-in LED, Arduino style
Momentary push button switch	1	S1	Reset button
100 nF ceramic capacitor	4	C1, C2, C3, C4	Decoupling capacitors
10 uF 16 V electrolytic capacitor	1	C5	Decoupling capacitor for power supply
1N4001 diode	1	D1	Reverse polarity protection, voltage dropper
16 MHz crystal	1	XTAL1	Optional: For ATmega328P with Uno bootloader present
22 pF ceramic capacitor	2	C6, C7	Optional: For ATmega328P with Uno bootloader present
Battery pack	1	VCC1	Power supply. 6 V for ATmega328P with Uno bootloader, 4.5 V otherwise

No ICSP? No Problem!

So, you managed to buy an ATmega328P, but it came without an Uno bootloader burned in. You don't have an ICSP to program it yourself, so what can you do? Well, you cannot burn a bootloader without an ICSP, and that could leave you a tad stuck, but Arduino to the rescue.

It's quite easy to do, and there are recent details on the Arduino Tutorials section of the Arduino website at www.arduino.cc/en/Tutorial/ArduinoISP. The steps are as follows:

- In the Arduino IDE, open the example sketch "ArduinoISP"—go to File ▹ Examples ▹ ArduinoISP ▹ ArduinoISP.
- Compile and upload to your Arduino in the normal manner.
- Connect the breadboarded ATmega328P to your Arduino as detailed in Figure I-1 and in the text.
- Burn the bootloader.

I.1 ArduinoISP Sketch

This is a sketch that converts an Arduino into an ICSP device. It is supplied with the IDE and can be found under File ▹ Examples ▹ ArduinoISP ▹ ArduinoISP. Simply open the sketch, compile, and upload it to your Arduino in the normal manner.

Your Arduino is now able to be used as an ICSP device.

Note
From this point onward, the Arduino you have just programmed will be referred to as "Arduino (ICSP)," while the device you wish to program will be known as "Breadboard ATmega328P."

One thing you will also need is a 10 uF capacitor between the `RES` and `GND` holes in the Arduino (ICSP) board header. This is required to prevent the Arduino (ICSP) from being reset when it starts to upload the new sketch to the breadboard ATmega328P. However, with my Uno clone, I have so far never needed this capacitor. Better to be safe than sorry though.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

Warning
The chances are that you will be using an electrolytic capacitor, so make sure that the negative lead is pushed into `GND`; otherwise, you might cause the capacitor to burst.

I.2 Connections

To program the breadboard ATmega328P, you need to connect four wires to it from the Arduino ICSP, in addition to the power and ground lines, of course. The connections are shown in Table I-1.

Figure I-1 shows how the connections should look. Only the bare essential components are installed on the breadboard; this is just to burn the bootloader, nothing else. Capacitors C2 and C3 are optional, but extra power supply filtering isn't a bad thing! Capacitor C1 may be required if your ArduinoISP ends up being reprogrammed instead of the breadboard ATmega328P. With regard to C1, the documentation says

> The 10 uF electrolytic capacitor connected to `RST` and `GND` of the programming board is needed only for the boards that have an interface between the microcontroller and the computer's USB, like Mega, UNO, Mini, Nano. Boards like Leonardo, Esplora and Micro, with the USB directly managed by the microcontroller, don't need the capacitor.

There are three additional connections you can make if you are a big fan of flashing LEDs. These are optional and listed in Table I-2.

Table I-1 ArduinoISP connections

ArduinoISP	ATmega328P	Description
D10	Pin 1, RST	Used to reset the breadboard ATmega328P
D11	Pin 17, MOSI	ArduinoISP talks to breadboard ATmega328P
D12	Pin 18, MISO	ArduinoISP reads from breadboard ATmega328P
D13	Pin 19, SCK	System clock to synchronize both devices

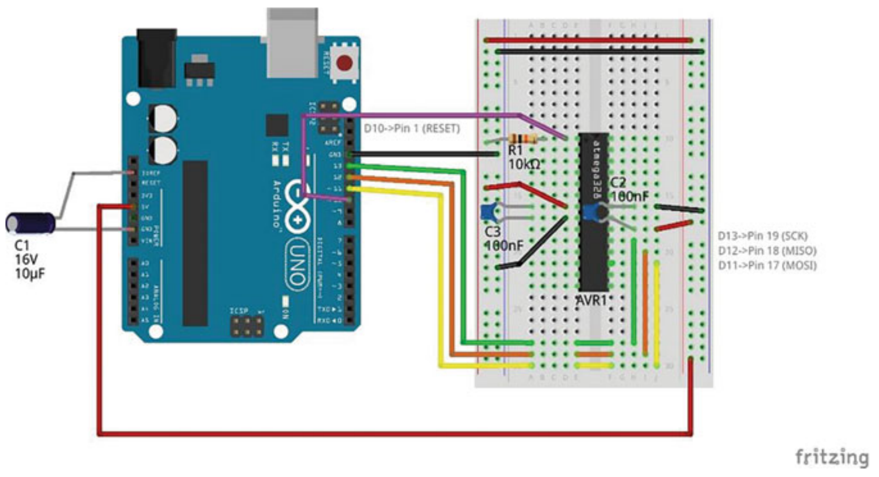


Figure I-1 ArduinoISP

Table I-2 ArduinoISP optional LED connections

ArduinoISP	Description
D7	Blinks when breadboard ATmega328P is being programmed
D8	Illuminates on programming errors
D9	Blinks a “heartbeat” to indicate we are still alive

These pins should be connected to a resistor, with a value between 330 Ω and 560 Ω, and from there to the positive (longest) lead of an LED. The short lead of the LED goes to `GND`. I use a red LED for the error indicator, green for the heartbeat, and yellow for the programming indicator.

I.3 Choose Your Programmer

In the IDE, select Tools ▹ Programmer and look for the option “Arduino as ISP.” Do not select “ArduinoISP”—that is not the programmer you want; trust me, I know from bitter experience! It’s a bit of a shame that the sketch is named “ArduinoISP” and that there is a programmer option, “ArduinoISP” as well. However, that is not the correct programmer.

I.4 Burn the Bootloader

In the IDE, choose the settings you require for the breadboard ATmega328P by selecting the appropriate choices from the Tools ▹ Board ▹ Boards Manager and, if required, Tools ▹ Processor, etc. Normally, your selection here would be to simply pick an Arduino Uno as the board. The ArduinoISP sketch can also be used to program other devices; I have programmed my ATtiny85 boards with it. Just be aware that you should not connect boards that run from 3.3 V to a 5 V ArduinoISP device!

And finally, Tools ▹ Burn Bootloader is all you need.

After a couple of seconds, you should see a message that the bootloader has been burned. You now have burned the Uno bootloader and set the required fuses. The breadboard ATmega328P can now be programmed with an FTDI adaptor.

Warning
You can only select the preceding Uno if you have the 16 MHz crystal and associated capacitors. If you don’t have those, and NormDuino doesn’t, then Appendix J shows how you can set up a special breadboard device that runs at 8 MHz on the internal oscillator—just like NormDuino.

For more information on using an Arduino as an ICSP device, there is documentation on the Arduino documentation site at https://docs.arduino.cc/built-in-examples/arduino-isp/ArduinoISP.

Breadboard 8 MHz Board Setup J

If you have built the NormDuino on a breadboard, you might be interested to know that you can remove the 16 MHz crystal and supporting capacitors and free up two additional GPIO pins by running the board at 8 MHz rather than 16. In addition to the extra pins, the board will now run with a smaller power requirement leading to longer lasting batteries.

The following steps should be carried out with the IDE closed.

We first need to discover if your installation has the directory named $ARDINST present. If you remember, this is `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6`, but obviously, your path will most likely be different.

Find the correct location and, to be sure, check to see if `boards.txt`, `platform.txt`, and `programmers.txt` exist. If so, you are in the correct location.

Check for the existence of an 8 MHz bootloader. There should be one supplied for the ATmega boards. Look for a file named `ATmegaBOOT_168_atmega328_pro_8MHz.hex` in the directory `$ARDINST/bootloaders/atmega/`. If you find it, all is good.

If the 8 MHz bootloader file is not found on your system, then please follow the instructions at https://docs.arduino.cc/built-in-examples/arduino-isp/ArduinoToBreadboard#minimal-circuit-eliminating-the-external-clock instead of mine. You will need a few more files to make the breadboard 8 MHz Arduino (or NormDuino) work. The section I link to earlier is the one entitled "Minimal Circuit (Eliminating the External Clock)." There you will find a list of steps to follow on downloading and adding support for a breadboarded 8 MHz device.

I'm assuming that you did find the bootloader on your system? If so, go to your own $ARDINST location—mine is `/home/norman/.arduino15/packages/arduino/hardware/avr/1.8.6`—and add the code in Listing J-1 to the end of `boards.local.txt` if the file exists. If it doesn't exist, simply create a new file with that name and add the code in Listing J-1 to it.

The code defines an "Arduino" on a breadboard, which is desired to be run on the internal oscillator, at 8 MHz—similar to NormDuino described in Appendix H. Because the bootloader already exists, all that is needed is the `boards.local.txt` file.

Listing J-1 The boards.local.txt settings

```
B8.name=Breadboard 8 MHz
B8.upload.protocol=arduino
B8.upload.maximum_size=30720
B8.upload.speed=57600
B8.bootloader.low_fuses=0xE2
B8.bootloader.high_fuses=0xDA
B8.bootloader.extended_fuses=0x05
B8.bootloader.file=atmega/ATmegaBOOT_168_atmega328_pro_8MHz.hex
B8.bootloader.unlock_bits=0x3F
B8.bootloader.lock_bits=0x0F
B8.build.mcu=atmega328p
B8.build.f_cpu=8000000L
B8.build.core=arduino:arduino
B8.build.variant=arduino:standard
B8.build.board=BB8
B8.bootloader.tool=arduino:avrdude
B8.upload.tool=arduino:avrdude
```

Once the `boards.local.txt` file has been saved, you may restart the IDE and connect your board.

You will need an ICSP or an Arduino as an ISP as described in Appendix I to program the bootloader into the AVR microcontroller. The steps are as follows:

- Open the IDE, and using the drop-down selection box on the main toolbar, choose “Select other board and port.” In the following dialog, search for “Breadboard 8 MHz.” Once listed, click its name, select the correct port, and click the OK button.
- Click Tools ▷ Programmer and choose the appropriate programmer. Mine is “USBtiny,” but “Arduino as ISP” is another good option especially if you still have the setup described in Appendix I.
- Choose Tools ▷ Burn bootloader.

After a few seconds of activity, you should see a prompt that the bootloader has been burned. You can now remove the ICSP device and switch back to an FTDI converter to do the subsequent programming. Don’t forget to select Tools ▷ Programmer in the IDE, and change the programmer back to “AVRISP mk11” so that you can start using the bootloader.

You now have an Arduino-compatible device that is running on a breadboard at 8 MHz and doesn’t require a 16 MHz crystal and two associated capacitors and which can be easily programmed from the Arduino IDE using an FTDI adaptor.

How easy was that?

Well, I have a confession. When I was testing this on my NormDuino, I burned a bootloader successfully with the USBtiny and uploaded a sketch—yes, it *was* the blink sketch—using the bootloader. However, if I immediately tried to upload again with the bootloader, it failed. The situation was such that after burning a bootloader, the IDE would only ever upload a sketch once to the NormDuino—no matter what I did.

After much wailing and gnashing of teeth, none of which helped, I pulled the ATmega328P from my Duemilanove and swapped it for the NormDuino one. I then burned a Duemilanove bootloader, and sketches could be uploaded time after time with no errors. I then chose the 8 MHz Breadboard Arduino and burned a bootloader again.

Sketches still continued to upload perfectly while the ATmega328P sat in my Duemilanove. Time to swap it back.

On attempting to insert the microcontroller back into the breadboard, I realized that the capacitor between the `RST` pin, pin 1 on the ATmega328P, and the FTDI's `DTR` line was not connected to the AVR anymore! Could this be the solution?

After plugging the AVR back into the breadboard and reattaching the capacitor to the `RST` pin, all was well.

Breadboards are fine to prototype your projects, but if you intend to keep and use the project, then it's a good idea to build it into a circuit board or at least a strip board. Too many things fall out of breadboards.

AVRAssist

K

AVRAssist is a small number of header files which can be #included in your AVR C++ code or #included in an Arduino sketch where you want to get down and dirty in the various hardware bits of the AVR microcontroller. The latest version of *AVRAssist* can always be obtained from https://github.com/NormanDunbar/AVRAssist/releases/latest.

K.1 Components

The following AVR internal devices can be set up with the current version of *AVRAssist*:

- Timers: All three timers have separate header files.
- Analog-to-Digital Converter.
- The Analog Comparator.
- The Watchdog Timer.

K.2 Using AVRAssist

The *AVRAssist* header files make configuration of the AVR registers for the components in the previous section a tad easier. As an example, the code in Listing K-1 will configure Timer 0 with

- Fast PWM mode with TOP = 255
- A prescaler of 64
- OC0A and OC0B = Pins 11 and 12 = Arduino D5 and D6 = AVR PD5 and PD6 in normal GPIO mode.
- Interrupt to fire on compare match A.
- Interrupt to fire on compare match B.

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6

Listing K-1 AVRAssist Timer 0 initialization

```
#include <timer0.h>

using namespace AVRAssist;

void setup() {
    ...

    Timer0::initialise(
    Timer0::MODE_FAST_PWM_255,
    Timer0::CLK_PRESCALE_64,
    Timer0::OCOX_DISCONNECTED,
    Timer0::INT_COMP_MATCH_A | Timer0::INT_COMP_MATCH_B
    );

    ...
}
```

Only the first two parameters are actually required though; the rest have sensible defaults. (For certain values of sensible perhaps?)

Later on in the code, you can force a comparison between `TCNT0` and either `OCR0A` or `OCR0B` should you have the need. This will not fire any configured interrupts but will set `OC0A` or `OC0B` to the state they would take without a forced compare if they happen to match `TCNT0` when forced. This is simple to do and is shown in Listing K-2.

Listing K-2 AVRAssist Timer 0 force compare

```
...

Timer0::forceCompare(Timer0::FORCE_COMPARE_MATCH_A |
                     Timer0::FORCE_COMPARE_MATCH_B

...
```

The *AVRAssist* method is, hopefully, much easier and less prone to fat-fingered typist syndrome (something I suffer from, frequently!) especially when attempting to type the equivalent code for Listing K-1, as shown in Listing K-3.

Listing K-3 Equivalent code to Listing K-1.

```
TCCR0A = ((0 << COM0A1) | (0 << COM0A0) |
          (0 << COM0B1) | (0 << COM0B0) |
          (1 << WGM00)  | (1 << WGM01));

TCCR0B = ((0 << FOC0A) | (0 << FOC0B) |
          (0 << WGM02) |
          (0 << CS02)  | (1 << CS01)  | (1 << CS00));

TIMSK0 = ((1 << OCIE0A) | (1 << OCIE0B) | (0 << TOIE0));
```

Well, I find it easier! And, yes, I did manage to type the code incorrectly when writing this description. Sigh!

Note
I know! Yes, I have included all the zero bits in the preceding equivalent code, but that makes it easier to change the bits later, especially if I needed to change the mode, prescaler, interrupts, etc.

Index

N. Dunbar, *Arduino Software Internals*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0171-6