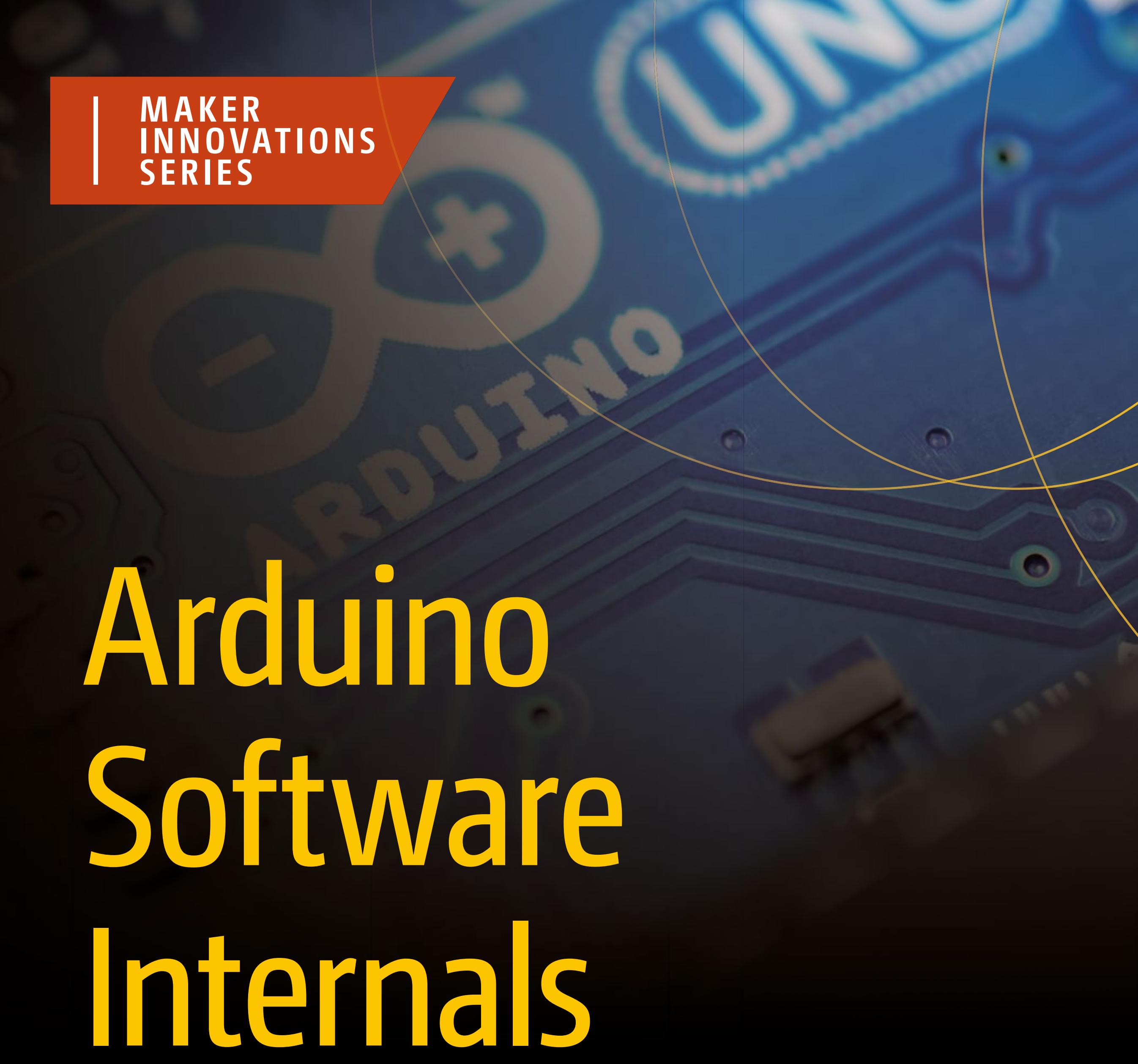

MAKER INNOVATIONS SERIES

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.

1 Introduction

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.

2 Arduino Compilation

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.