elektor books

4th edition 2024
covering KiCad 8

LIKE A PRO

KiCad

Advanced Projects and Recipes

Mastering PCB design with real-world projects

HQ NextPCB
www.nextpcb.com

Peter Dalmaris, PhD

elektor
design > share > earn

KiCad Like A Pro

Advanced Projects and Recipes

Mastering PCB design with real-world projects

4th edition covering KiCad 8

Peter Dalmaris, PhD

elektor
design › share › earn

This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
PO Box 11, NL-6114-ZG Susteren, The Netherlands
Phone: +31 46 4389444

All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licencing Agency Ltd., 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's permission to reproduce any part of the publication should be addressed to the publishers.

Declaration

The Author and Publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, and hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident, or any other cause.

All the programs given in the book are Copyright of the Author and Elektor International Media. These programs may only be used for educational purposes. Written permission from the Author or Elektor must be obtained before any of these programs can be used for commercial purposes.

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

This third edition of **KiCad Like a Pro** consists of two books:

- **KiCad Like A Pro | Fundamentals and Projects**
  Getting started with the world's best open-source PCB tool
  590 pages
  ISBN: 978-3-89576-626-8 print
  ISBN: 978-3-89576-627-5 ebook

- **KiCad Like A Pro | Advanced Projects and Recipes**
  Mastering PCB design with real-world projects
  492 pages
  ISBN: 978-3-89576-628-2 print
  ISBN: 978-3-89576-629-9 ebook

© Copyright 2024: Elektor International Media B.V.
Prepress Production: D-Vision, Julian van den Berg
Printed in the Netherlands by Ipskamp Printing, Enschede

Elektor is the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (including magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. **www.elektormagazine.com**

Contents

Did you find an error? 11

About the author 11

How to read Kicad Like A Pro 12

What will you need 13

The book web page 14

KiCad 8 15

Foreword by Wayne Stambaugh 16

An introduction: Why KiCad? 17

About Tech Explorations 21

1: Project - Tiny Solar Power Supply 23

1.1. Introduction 23

1.2. Schematic 26

1.2.1. 1 - Setup 27

1.2.2. 2 - Symbols 28

1.2.3. 2 - Edit Component values 32

1.2.4. 3 - Arrange, Annotate, Associate 34

1.2.5. 4 - Wiring 36

1.2.6. 5 & 6 - Nets and Electrical Rules Check 38

1.2.7. 7 - Comments 39

1.3. Layout 41

1.3.1. 1 - Setup 42

1.3.2. 2 & 3 - Placement and outline 44

1.3.3. 4 - Route 48

1.3.4. 2 - Refine the outline 50

1.3.5. 5 - Copper fills 51

1.3.6. 6 - Silkscreen 56

1.3.7. 7 - Design Rules Check 61

1.3.8. Two more things: 3D models and mounting holes 63

1.3.9. 8 - Export and Manufacture 67

2: Project - A 4 x 8 x 8 LED matrix array 73

2.1. Introduction 73
2.2. Schematic design 78
2.2.1. 1 - Setup 78
2.2.2. 2 - Symbols 81
2.2.3. 3 - Arrange, Annotate 83
2.2.4. 3 - Associate 86
2.2.5. 4 - Wiring 88
2.2.6. 5 - Nets 92
2.2.7. 6 - Electrical Rules Check 94
2.2.8. 7 - Comments 96
2.2.9. Last-minute edits 98
2.3. Layout design editing 100
2.3.1. 1 - Setup 101
2.3.2. 2 - Outline and constraints 103
2.3.3. 3 - Place components 108
2.3.4. 2 - Refine outline 115
2.3.5. 3 - Move footprints 118
2.3.6. 4 - Route 122
2.3.7. 4 - Copper fills 126
2.3.8. 5 - Silkscreen 129
2.3.9. 6 - Design Rules Check 136
2.3.10. 7 - Manufacture 142
2.5. Bonus - 3D shapes 144
2.6. Bonus - Found a bug in the schematic! (and fix) 151
2.7. Assembled PCB 154
3 : Project - MCU datalogger 158
3.1. Project - Introduction 158
3.2. Create the new project and Git repository 162
3.3. Schematic design 165
3.3.1. Schema 1 - Setup 165
3.3.2. Schema 2 - Symbols 167

3.3.3. Schema 2 - Sheet two 170
3.3.4. Schema 3 - Arrange, Annotate 172
3.3.5. Edit component values 173
3.3.6. Schema 3 - Associate 175
3.3.7. Schema 4 - Wiring of sheet 1 177
3.3.8. Schema 4 - Wiring of sheet 2 179
3.3.9. Schema 5 - Nets 183
3.3.10. Schema 6 - Electrical Rules Check 184
3.3.11. Schema 7 - Comments 186
3.4. Create the 2-layer branch in Git 188
3.5. Layout design 189
3.5.1. Layout 1 - Setup 189
3.5.2. Layout 2 - Outline and constraints 191
3.5.3. Layout 3 - Place components 193
3.5.4. Layout 2 - Outline refinement 194
3.5.5. Layout 4 - Route 196
3.5.6. Layout 4 - Copper fills 201
3.5.7. Layout 4 - Routing improvements 207
3.5.8. Layout 5 - Silkscreen 210
3.5.9. Layout 6 - Design Rules Check 216
3.5.10. Layout 7 - Manufacture 217
3.6. 3D shapes 222
3.7. Merge 2-layer branch to main 225
3.8. Design 4 Layer PCB in new Git branch 227
3.9. Four-layer PCB routing 228
3.10. Four-layer PCB manufacturing 231
3.11. Updating layout from changes to the schematic with Git 234
3.12. Finding and correcting a design defect 237
3.12.1. Fix the schematic 239
3.12.2. Fix the 2 layer PCB layout 246
3.12.3. Fix the 4 layer PCB layout 249

4 : Project - An ESP32 clone 252

4.1. Project - Introduction 252

4.2. Schematic design 258

4.2.1. Schema 1 - New KiCad project and Schematic Setup 258

4.2.2. Schema 2 - Symbols 260

4.2.3. Schema 3 - Annotate and set component values 263

4.2.4. Schema 3 - Arrange 267

4.2.5. Schema 3 - Associate 269

4.2.6. Schema 4 - Wiring 270

4.2.7. Schema 5 - Nets and Net Classes 275

4.2.8. Schema 6 - Electrical Rules Check 275

4.2.9. Schema 7 - Comments 279

4.3. Layout design 281

4.3.1. Layout 1 - Setup 283

4.3.2. Layout 2 - Outline and constraints 286

4.3.3. Layout 3 - Place components 291

4.3.4. Layout 2 - refine outline and DRC errors 297

4.3.5. Layout 4 - Route 304

4.3.6. Layout 4 - Copper fills and keep out areas 308

4.3.7. Layout 5 - Silkscreen 312

4.3.8. Layout 4 - Routing improvements 314

4.3.9. Layout 6 - Design Rules Check 315

4.3.10. Layout 7 - Manufacture 316

4.4. 3D shapes 321

4.5. Finding and correcting a design defect 323

4.5.1. Fix the schematic 324

4.5.2. Fix the layout 325

5: Recipes 328

5.1. Interactive delete 328

5.2. Find and Replace (Schematic editor) 330

5.3. Text variables 332

5.4. Board Setup - pre-defined sizes for tracks and vias 337
5.5. Board Setup - Design rules violation severity 341
5.6. Board Setup - Custom design rules 344
5.7. Schematic Setup - Electrical Rules and violation severity 348
5.8. Schematic Setup - Electrical Rules and Pin conflicts map 351
5.9. Import components from Snapeda 354
5.10. The Freerouting autorouter 362
5.10.1. Freerouting plugin installation and operation 365
5.10.2. Using Freerouting 366
5.10.3. Autorouting process and customisation 369
5.11. PCB editor Inspection menu 372
5.12. Single track and differential pair routing 381
5.13. Track length tuning 385
5.14. Differential pair skew tuning 389
5.15. The footprint wizard 391
5.16. Pin and wire highlighter tool 396
5.17. KiCad project management with Git 398
5.17.1. Install Git 403
5.17.2. Git configuration 404
5.17.3. Create a new KiCad project Git repository 405
5.17.4. How to ignore files 407
5.17.5. Basic Git commands: add, commit 410
5.17.6. Basic Git commands: branch 414
5.17.7. Basic Git commands: merge 417
5.18. Sharing your KiCad project on GitHub 419
5.19. Customize the editor color scheme 428
5.20. Import an EAGLE, Altium, or Cadstar schematic 432
5.21. The circuit simulator 435
5.21.1. Prepare the circuit for simulation 437
5.21.2. Configure the simulator 444
5.21.3. Simulate 446

5.22. KiCad project templates 452
5.22.1. Using a system project template 453
5.22.2. Create a user project template 456
5.23. Share projects and configurations with others 458
5.24. Calculate the width of a trace 463
5.25. Design a custom schematic sheet 464
5.26. Python scripting API 472
5.26.1. A simple example 473
5.26.2. Make a change to a PCB 479
5.26.3. Create a KiCad Action Plugin 482
5.26.4. Learning More About KiCad Python Scripting API 487
Index 488

Did you find an error?

Please let us know.
Using any web browser, go to **txplo.re/kicadr**, and fill in the form.
We'll get it fixed right away.

About the author

Dr. Peter Dalmaris is an educator, an electrical engineer, an electronics hobbyist, and a Maker. Creator of online video courses on DIY electronics and author of several technical books. Peter has recently released his book 'Maker Education Revolution', a book about how Making is changing the way we learn and teach in the 21st century.

As a Chief Tech Explorer since 2013 at Tech Explorations, the company he founded in Sydney, Australia, Peter's mission is to explore technology and help educate the world.

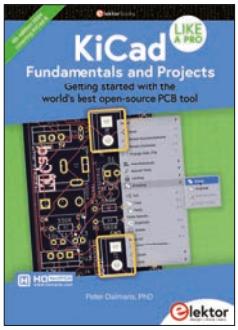
Tech Explorations offers educational courses and Bootcamps for electronics hobbyists, STEM students, and STEM teachers.

A lifelong learner, Peter's core skill lies in explaining difficult concepts through video and text. With over 15 years of tertiary teaching experience, Peter has developed a simple yet comprehensive style of teaching that students from all around the world appreciate.

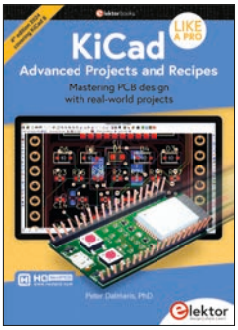
His passion for technology and the world of DIY open-source hardware has been a dominant driver that has guided his personal development and his work through Tech Explorations.

How to read the 4th edition of KiCad Like a Pro

This updated edition of KiCad Like a Pro consists of two complementary but separate books encompassing KiCad 8. The volume **KiCad Like a Pro – Advanced Projects and Recipes** (492 pages) you are holding now includes projects of increasing complexity. It is ideal for readers with a basic working knowledge of KiCad and who have completed the introductory projects described in the book **KiCad Like a Pro – Fundamentals and Projects** (590 pages).



KiCad Like a Pro – Fundamentals and Projects



KiCad Like a Pro – Advanced Projects and Recipes: The book you are holding now!

If you have never used KiCad and have little or no experience in PCB design, please read the introductory **Fundamentals and Projects** first. It is suitable for readers with little or no experience in PCB design. Then return to the volume **Advanced Projects and Recipes**.

At the end of both books, you will find recipes, helpful on their own. In the Recipes part, I have dedicated chapters to show you how to complete specific tasks, such as creating a bill of materials or editing text properties. These are basic skills that you will use in most of your projects. Throughout the book, you will find prompts for these recipes to help reduce the need to repeat instructions.

Numerous figures in this book contain screenshots of KiCad. I created these screenshots using KiCad 8.0, which runs on Mac OS. If you are using KiCad under Windows or Linux, do not worry: KiCad works the same across these platforms and even looks almost the same.

Although I cared to produce clear images, this was sometimes impossible. This is particularly true in screenshots of an entire application window meant to be displayed on a large screen. The role of these images is to help you follow the instructions in the book as you are working on your computer. There is no substitute for experimenting and learning by doing, so the best advice I can give is to use this book as a textbook and companion. Whenever you read it, have KiCad open on your computer and follow along with the instructions.

This book has a web page with resources designed to maximize the value it delivers to you, the reader. Please read about the book web page, what it offers, and how to access it on page 14 in this introductory segment.

Finally, you may be interested in the video course version of this book. This course spans over 25 hours of high-definition video, with detailed explanations and demonstrations of all projects featured in the book. The video lectures capture techniques and procedures that are impossible in text.

Please check the book web page for updates on this project. Be sure to subscribe to the Tech Explorations email list so I can send you updates.

PS1. For an official summary of what's changed and new in KiCad 8, please read the release announcement on the KiCad blog:
https://www.kicad.org/blog/2024/02/Version-8.0.0-Released/

PS2. For a list of all project commits that relate to KiCad 8, please follow this link (it will open the project repository on Gitlab and list all issues with the "8.0" label):
https://txplo.re/kicad8_issues

PS3. If you are curious about what will be in KiCad 9, follow this link:
https://txplo.re/kicad_8.99_issues

What will you need

To make the most out of this book, you will need a few things. You probably already have them:

- A computer running Windows, Mac OS or Linux.
- Access to the Internet.
- A mouse with at least two buttons and a scroll wheel. I use a Logitech MX Master 2S mouse (see https://amzn.to/2ClySq0).
    - With KiCad 8, it is also possible to use the SpaceMouse device by 3Dconnexion with Windows and Mac OS. I have never used this device. However, some people claim that these mice are more ergonomic and intuitive than regular mice.
- Ability to install software.
- Time to work on the book, and patience.

The book web page

As a reader of this book, I invite you to access its online resources.

You can access these resources by visiting the book's web page at https://txplo.re/klp4.

The three available resources are:

1. **Photos and schematics**. Get high-res copies of the book's photos, schematics, and layouts.

2. **Projects**. Use the links on this page to download the full KiCad projects from the book.

3. **Errata**. As I correct bugs, I will post information about these corrections on this page. Please check this page if you suspect that you have found an error. If an error you have found is not listed on the errata page, please use the error report form on the same page to let me know about it.

KiCad 8

KiCad 8 was introduced by the KiCad team in early 2024. It represents the next step of evolution for KiCad, with several incremental improvements and additions over KiCad 7.

Jon Evans, one of the lead developers for KiCad, presented the changes in KiCad 8 in a detailed blog post on the KiCad website. You can read this post at this address:

https://www.kicad.org/blog/2024/02/Version-8.0.0-Released/

I have updated this book for KiCad 8. In preparation for the update, I first checked all the projects in this book and ensured that they worked in KiCad 8. I am pleased to say that all the projects work properly without modifications.

After ensuring that all projects work properly in KiCad 8, I reviewed the KiCad 8 changes list and documented these changes so that they are presented naturally in the course of this book and its projects.

Because the KiCad 8 changes are "evolutionary" (and not "revolutionary" as they were with KiCad 6), I decided that updating this book is preferable to writing a new one. With KiCad's new release schedule (essentially, a new major version of KiCad will be published every year), it is not possible for me to write a new book yearly.

To manage the introduction of evolutionary changes into the existing book, I have adopted a system that incorporates changes according to their impact on the PCB design workflow.

For small changes, such as the ability to use custom fonts, I will be inserting "asides". An aside is a small remark, typically inside a box or as a footnote, that presents the change in line with the rest of the document.

For larger changes, I will introduce new chapters within the existing projects or in the Recipes part.

When a change significantly impacts one of the projects, I will introduce a chapter to that project.

Most changes are documented in new chapters in the Recipes part.

The KiCad 8 user interface is very similar to KiCad 6 and 7. The schematic and layout editors look unchanged without detailed examination. Most dialogue boxes also seem unchanged, except for some widgets (such as text boxes and radio buttons) being moved around. I expect that most people who are familiar with KiCad 6 and 7 will be able to find their way around the KiCad 8 user interface without any trouble.

I used KiCad 8 for all screenshots in this fourth edition of the book.

Foreword by Wayne Stambaugh

In 1992 Jean-Pierre Charras started the KiCad project. From it's humble beginnings as one man's goal to provide an electronics design application for his students to a full fledged open source project with a significant number of contributors, KiCad has become the choice for users who prefer an open source solution for electronics design. As the feature parity gap from proprietary EDAs continues to close, KiCad will continue to become more widely accepted and influential.

One of the enduring hallmarks of a successful open source software project is the publication of a "how to" book. With the publication of "KiCad Like a Pro", the KiCad project joins the other well known open source projects with that distinction. Whether you are a beginner or a seasoned professional user, this book has something for everyone. From properly configuring and using KiCad to design your first printed circuit board to advanced topics such as using git for version control this book is a valuable resource for any KiCad user.

Thanks to the generosity of author Peter Dalmaris and the publisher Tech Explorations, fifty percent of the profits from the sales of the special fundraising edition of this book will be donated to the KiCad project through CERN. On behalf of all of the developers and contributors of the KiCad project, I thank you for your continued support of the project and your generous contributions.

Wayne Stambaugh
KiCad Project Leader

An introduction: Why KiCad?

Since KiCad first appeared in the PCB CAD world in 1992, it has gone through 7 major versions and evolved into a serious alternative to commercial products. I have been using KiCad almost daily since version 4, when I published the first edition of KiCad Like a Pro.

Once thought clunky and barely usable, it is now a solid, reliable CAD application. KiCad has been consistently closing the feature and performance gap against its commercial competitors. It has made leaps in adding powerful features and has significantly improved its stability.
Combined with the benefits of free and open-source software, I believe that KiCad is simply the best PCB CAD software for most use cases.

One benefit is KiCad's active and growing community of users and contributors. KiCad has a dedicated developer team supported by contributing organizations like CERN, the Raspberry Pi Foundation, Arduino LLC, and Digi-Key Electronics. The community is also active in contributing funds to cover development costs. Since joining the Linux Foundation, the KiCad project has received significant development funds in the form of donations from users and industry. The project used this money to buy development time and fund developer conference travel and meetups. To a large extent, this alone guarantees that KiCad's development will accelerate and continue in the future.

Supporting the KiCad core team is the KiCad community. The community consists of over 250 thousand people worldwide who have downloaded a copy. These people support the KiCad project in various ways: they write code, create and share libraries, and help others learn. They write documentation, record videos, report bugs, and share hacks. During each yearly development cycle, the KiCad repository shows tens of thousands of commits in the source code branches from the developer community. A similarly large number of commits is seem in other parts of the project, such as the libraries and the documentation.

Another signal of the strength of the KiCad community is that KiCad includes completed or nearly completed translations to 12 languages. No other CAD software that I am aware of can boast this.

PCB manufacturers have also taken notice. Many of them now publish KiCad-specific tutorials, explaining how to order your boards. Some have allowed uploading the KiCad native layout file from your project instead of generating multiple Gerber files.

And finally, KiCad is part of an expanding CAD ecosystem. You will find KiCad-compatible component libraries on the Internet's major repositories, such as Snapeda and Octopart, and native support in PCB project version control software for teams, such as CADLAB.io.

KiCad's development and prospects have never been brighter than now. KiCad's post-V8 roadmap has exciting new features and capabilities, such as a new Zone Manager feature, the ability to add comments to ERC and DRC exclusions, and a selection filter for schematic/symbol editors.

Why do I use KiCad? Because it is the perfect PCB software for my use.

I am an electrical engineer with a background in electronics and computer engineering. But, above all, I am a technology educator and electronics hobbyist. Most of my PCB projects eventually find themselves in my books and courses. My projects are very similar to those of other hobbyists in complexity and size. I make things for my Arduino and Raspberry Pi courses. As a hobbyist, KiCad proved to be the perfect tool for me.

Your use case may be different. You may be a university student completing an engineering degree. You may be a hobbyist or solo developer working in a startup company. You may be part of a team on commercial projects involving highly integrated multi-layer PCBs.

To help you decide whether KiCad is right for you, I have compiled a list of 12 KiCad Benefits. This list contained ten items in the second edition of the book. I added the last two items to highlight additional benefits brought about with KiCad.

Here they are:

Benefit 1: KiCad is open source. This is very important, especially as I spend more time creating new and more complicated boards. Open source, by definition, means that the code base of the application is available for anyone to download and compile on their computer. It is why Linux, Apache, and WordPress essentially run the Internet (all of them open-source). While I am not extreme in my choices between open-source and closed-source software, whenever a no-brainer open-source option does appear, like KiCad, I take it.

Benefit 2: It is free! This is particularly important for hobbyists. CAD tools can be expensive. This worsens with most CAD software companies switching to a subscription-based revenue model. Regular fees do add up when you are a hobbyist or student or bootstrapping for a startup. Not to mention that most of us would not be using even half of the features of commercial CAD software. It is hard to justify spending hundreds of dollars on PCB software when there is KiCad. This brings me to Benefit 3

Benefit 3: KiCad is unlimited. There are no ''standard'', ''premium'' and ''platinum'' versions to choose from. It's a single download, and you get everything. While there are commercial PCB tools with free licensing for students or hobbyists, there are always restrictions on things like how many layers and how big your board can be, what you can do with it once you have it, who can manufacture it, and much more. There is always the risk that the vendor may change the deal in the future, and you may have to pay a fee to access your projects. I'll say again: KiCad is unlimited and forever! This is so important that I choose to pay a yearly donation to CERN that is higher than the cost of an Autodesk Eagle license to do my part in helping to maintain this.

Benefit 4: KiCad has awesome features. Professional-grade features include interactive routing, length matching, multi-sheet schematics, configurable rules checker, and differential routing. While you may not need to use some immediately, you will eventually

use them. You can add new features through third-party add-ons. The external autorouter is one example. The ability to automate workflows and extend capabilities through Python scripts is another.

Benefit 5: KiCad is continually improved. Since the publication of KiCad 6, I have seen a very aggressive and successfully implemented roadmap. When I wrote the first version of this book (August 2018), KiCad 5 was about one month old. Three years later, KiCad 6 was delivered with a new architecture and features that set the stage for KiCad's future development. I am updating this book in March 2024, and KiCad 8 was delivered on schedule a couple of weeks ago, representing an evolution in the history of KiCad, built on the success of KiCad 6. The roadmap ahead looks exciting indeed.

Benefit 6: KiCad's clear separation of schematics and layout is a bonus to learning and using it. Users of other PCB applications often find this confusing, but I believe that it is an advantage. Schematic design and layout design are indeed two different things. Schematic symbols can be associated with different footprints that depend on the project requirements. You can use the schematic editor independently of the layout editor or in sync. I often create schematic diagrams for my courses that I have no intention of converting into PCBs. I also often create multiple versions of a board using the same schematic. This separation of roles makes both scenarios easy.

Benefit 7: I can make my boards anywhere: I can upload my project to any online fabricator that accepts the industry-standard Gerber files; I can upload it to an increasing number of fabricators that accept the native KiCad layout file; and, of course, I can make them at home using an etching kit.

Benefit 8: KiCad works anywhere. Whether you are a Mac, Windows, or Linux person, you can use KiCad. I use it on all three platforms. I can take my KiCad project from the Mac and continue working on Windows 10 or 11 without worrying about any software or project file glitches. If you use a Mac computer with Apple silicon (such as the M1 and M2 processors), you can now use KiCad 7 natively on your computer instead of relying on the emulation mode.

Benefit 9: KiCad is very configurable. You can assign your favourite keyboard hotkeys and mapping, and with the mouse customisations, you can fully adapt it to your preferences. With the additions of the plugin system and the Python API, extending your instance of KiCad with the exact features you need (or writing them) will be possible.

Benefit 10: If you want to create analog circuits, you will be happy to know that KiCad ships with SPICE. You can draw the schematic in Eeschema and simulate it in SPICE without leaving KiCad. This integration first appeared in KiCad 5, improved over versions 6 and 7, and looks amazing in KiCad 8 with additional simulation types and tighter integration with the schematic editor.

Benefit 11: KiCad's release cycle was somewhat chaotic in the past. New major versions would come out every two or three years, but no one knew beforehand. In the future, KiCad

will operate in a yearly release cycle. This is good for two reasons: One is commercial users who can now better predict how and when the software they depend on will change. Two, as KiCad users, we can expect a reliable development schedule prioritising reliability. KiCad is now mature enough to evolve predictably.

Benefit 12: KiCad is now a serious productivity tool for businesses. If you are an electronics engineer, you can proudly list it in your resume. If you use it in your business, you can contact the KiCad Services Corporation to customise the software to your requirements. I am talking about deep customisation, not just changing the theme and the menu bars. This means that KiCad can fit precisely with your business. As far as I know, no commercial CAD application can do that. For the non-business users among us, we can expect many of these business-led improvements to flow into future software versions in the tradition of open-source software.

These are the twelve most important reasons I have chosen KiCad as my tool of choice for designing PCBs. These reasons might not suit you, but I hope you will consider reading this book before making your own decision.

I have been using KiCad since version 4. I have packed almost everything I have learned as a KiCad user in this book. I have organised it in a way that will make learning KiCad quick. The objective of this book is to make you productive by the time you complete the first project in part four.

If you come from another PCB CAD tool and have experience designing PCBs, I ask that you have an open mind. KiCad is most certainly very different from your current PCB tool. It looks different, and it behaves differently. It will be easier to learn it if you consciously put aside your expectations and look at KiCad like a beginner would. As per the infamous Borg in Star Trek, ''resistance is futile", in learning, like in so many other aspects of life, you are better off if you go with the flow.

Let's begin!

About Tech Explorations

Tech Explorations creates educational products for students and hobbyists of electronics who rather utilize their time making awesome gadgets instead of searching endlessly through blog posts and Youtube videos.

We deliver high-quality instructional videos and books through our online learning platform, techexplorations.com.

Supporting our students through their learning journey is our priority, and we do this through our dedicated online community and course forums.

Founded in 2013 by Peter Dalmaris, Tech Explorations was created after Peter realised how difficult it was to find high-quality definitive guides for the Arduino, written or produced by creators who responded to their reader questions.

Peter was frustrated having to search for Youtube videos and blog articles that almost never seemed to be made for the purpose of conveying knowledge.

He decided to create Teach Explorations so that he could produce the educational content that he wished he could find back then.

Tech Explorations courses are designed to be comprehensive, definitive and practical. Whether it is through video, ebook, blog or email, our delivery is personal and conversational.

It is like having a friend showing you something neat... the "AHA" moments just flow!

Peter left his career in Academia after his passion for electronics and making was rekindled with the arrival of his first Arduino. Although he was an electronics hobbyist from a young age, something that led him to study electrical and electronics engineering in University, the Arduino signalled a revolution in the way that electronics is taught and learned.

Peter decided to be a part of this revolution and has never looked back.

We know that even today, with all the information of the world at your fingertips, thanks to Google, and all the components of the world one click away, thanks to eBay, the life of the electronics hobbyist is not easy.

Busy lifestyles leave little time for your hobby, and you want this time to count.

We want to help you to enjoy your hobby. We want you to enjoy learning amazing practical things that you can use to make your own awesome gadgets.

Electronics is a rewarding hobby. Science, engineering, mathematics, art, and curiosity all converge in a tiny circuit with a handful of components.

We want to help you take this journey without delays and frustrations.

Our courses have been used by over 70,000 people across the world.

From prototyping electronics with the Arduino to learning full-stack development with the Raspberry Pi or designing professional-looking printed circuit boards for their awesome gadgets, our students enjoyed taking our courses and improved their making skills dramatically.

Here's what some of them had to say:

"I'm about half way through this course and I am learning so much. Peter is an outstanding instructor. I recommend this course if you really want to learn about the versatility of the amazing Raspberry Pi" -- Scott

"The objectives of this course are uniquely defined and very useful. The instructor explains the material very clearly." -- Huan

"Logical for the beginner. Many things that I did not know so far about Arduino but easy to understand. Also the voice is easy to understand which is unlike many courses about microcontrollers that I have STARTED in the past. Thanks" -- Anthony

Please check out our courses at techexplorations.com and let us be part of your tech adventures.

1: Project - Tiny Solar Power Supply

1.1. Introduction

This chapter focuses on designing a Printed Circuit Board (PCB) based on the Tiny Solar Supply schematic, initially developed by Clemens Valens for Elektor. Valens is an engineer managing the Elektor Labs online platform and has a track record of contributing to projects that serve practical electronic applications. The Tiny Solar Supply is designed to convert solar power into a regulated voltage output, ideal for powering microcontrollers and similar devices.

In the following chapters, I will use KiCad to develop my Tiny Solar Supply based on Clemen's design. I will follow Clemen's original decisions in most of the design, but occasionally I will deviate. I will mainly use symbols and footprints found in KiCad's core libraries but also external libraries when needed.

As with previous projects in this book, I will develop the Tiny Solar Supply project step-by-step. I will start with the schematic's seven steps and continue with the layout.

The original project by Clemens Valens can be found at Elektor Magazine. It provides a foundation for this chapter's exploration of PCB design using KiCad. You can find this project on Elektor's website at https://www.elektormagazine.com/magazine/elektor-305/62009.

You can see a 3D view of the completed board below.

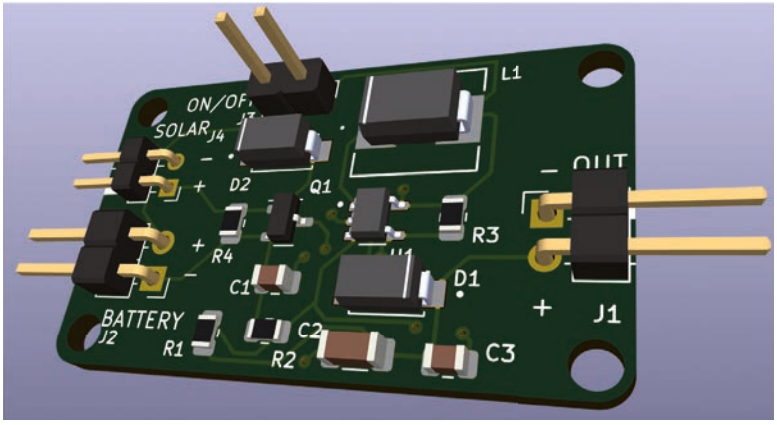


Figure 1.1.1: A 3D view of the tiny solar power supply board.

Here's a summary of what this circuit does and how.

The Tiny Solar Supply is a solar-powered boost converter designed to increase the voltage from a solar panel to a level suitable for powering electronics, such as microcontrollers, that require a stable voltage source. Here's how each part of the circuit contributes to its overall function:

1. Solar Panel and Battery: The solar panel is connected at input K4. During the day, it charges a rechargeable battery (connected at K2). This setup ensures that the device can operate both during the day when sunlight is available and at night using the energy stored in the battery.

2. Boost Converter IC (IC1 - AP3015): The AP3015 step-up DC/DC converter is at the heart of the circuit. This integrated circuit (IC) can work with input voltages as low as 1V and boost them to higher voltages, in this case, a regulated output of 3.3V. The output voltage of the circuit is set by the ratio of resistors R1 to R2, following the formula:

$$V_{OUT} = 1.23 \times (1 + \frac{R1}{R2})$$

With the specified resistor values, the output is nearly 3.3V.

3. Inductor (L1), Diodes (D1, D2), and Capacitors (C1, C2, C3): These components work together to smooth and stabilize the boosted voltage. The inductor and capacitors form an LC filter that reduces the ripple in the output voltage, ensuring a steady DC supply. Diode D2 prevents the backflow of current from the battery to the solar panel, and diode D1 ensures correct current flow in the boost converter circuit.

4. Transistor (T1 - 2N7002): This transistor acts as a switch controlled by the solar panel's output. During daylight, when the solar panel is active, T1 turns off the boost converter (by pulling the SHDN pin of IC1 low) to save energy. At night, or when the solar panel voltage drops too low, T1 allows the boost converter to turn on, providing a regulated voltage output from the battery.

5. Resistors (R1, R2, R3, R4) and Capacitors (C1, C2, C3): These resistors set the output voltage and ensure the IC's proper operation. Capacitors C1, C2, and C3 help filter the input and output, stabilizing the voltage provided to and from the boost converter.

6. Switch/Control (K3): An optional On/Off switch or jumper can be connected here, offering manual control over the circuit's operation.

This circuit cleverly utilizes solar power to provide a regulated 3.3V output suitable for low-power electronics. Converting and stabilizing the solar panel's variable output ensures a reliable power source for projects operating where traditional power sources are unavailable. Through the assembly and analysis of this circuit in KiCad, students gain hands-on experience with renewable energy applications and voltage conversion and regulation principles.

Here is the schematic of the board:

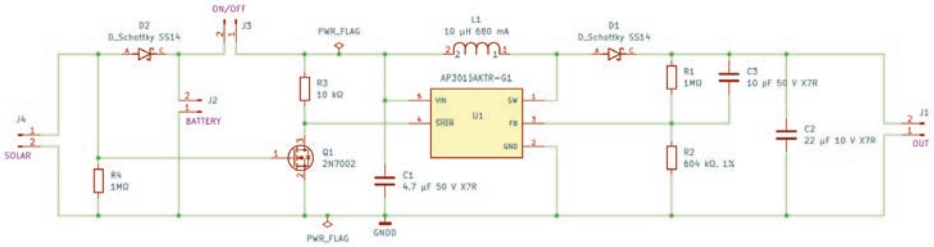


Figure 1.1.2: The project schematic design (final).

Here is the layout:

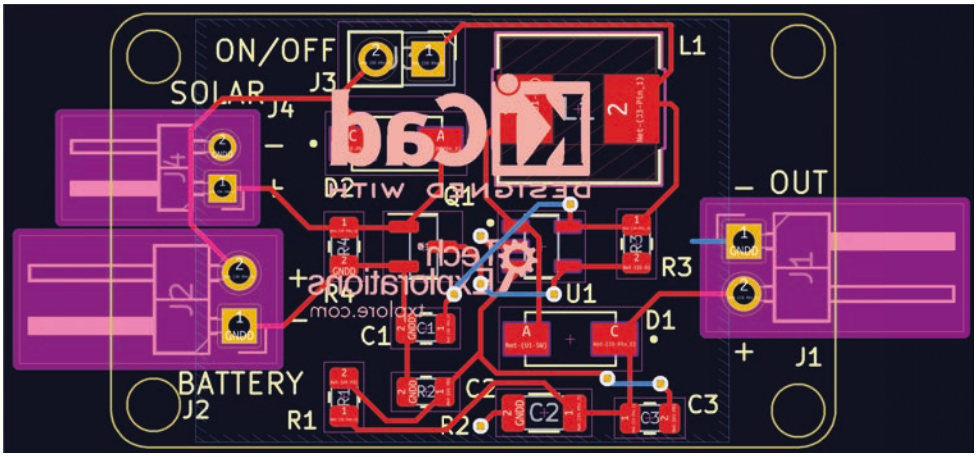


Figure 1.1.3: The project layout design (final).

The actual schematic and design of the board at the end of the process might look slightly different from what you see in the screenshots above.

Below are the components required for this project. I have chosen a footprint different from the one used in the original Elektor project for some components.

Reference	Value	Footprint
R1, R4	1 MΩ, 0.125 W	0805, Resistor_SMD:R_0805_2012Metric
R2	604 kΩ, 1%, 0.125 W	0805, Resistor_SMD:R_0805_2012Metric
R3	10 kΩ, 0.125 W	0805, Resistor_SMD:R_0805_2012Metric
C1	4.7 µF, 50 V, X7R	0805, Capacitor_SMD:C_0805_2012Metric
C2	22 µF, 10 V, X7R	1206, Capacitor_SMD:C_1206_3216Metric
C3	10 pF, 50 V, X7R	0805, Capacitor_SMD:C_0805_2012Metric
L1	10 µH, 680 mA	IND_ASPI-0630LR_ABR (external library from Ultralibrarian)

D1, D2	SS14	DO-214AC, DIOM4325X250N (external library from Snapeda)
IC1	AP3015A	AP3015AKTR-G1:SOT95P280X145-5N (external library from Snapeda)
T1	2N7002	SOT-23
J1, J2, J3	pin header, 1 row, 2 contacts	2.54 mm pitch, right-angle
J4	pin header, 1 row, 2 contacts	2 mm pitch

We'll design this PCB with two copper layers and a GND copper fill in the bottom layer.

Let's begin with the schematic design process.

1.2. Schematic

In this chapter, you will complete the schematic design of this PCB by following the schematic design workflow (see below). You learned about this workflow in Part 6 of the book.

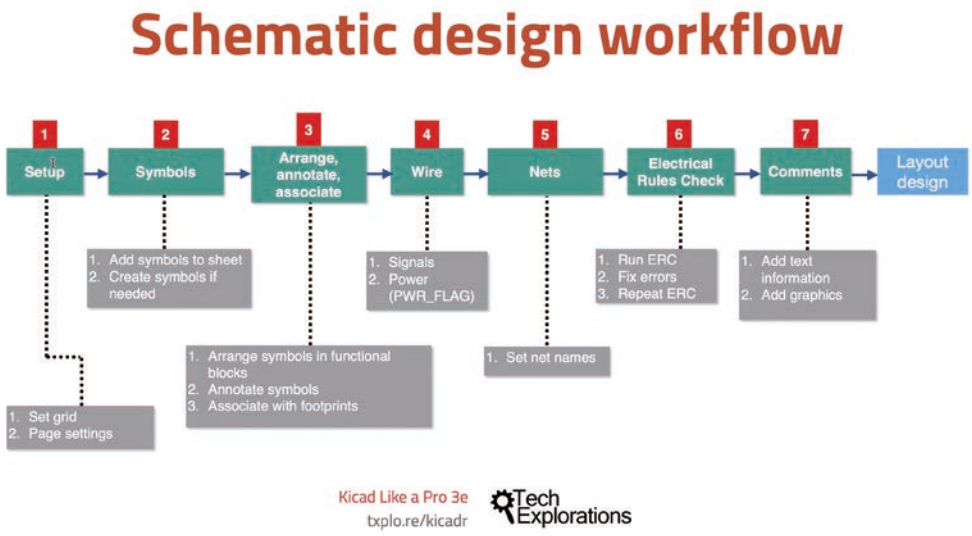


Figure 1.2.1: The schematic design workflow.

Unlike the simple project you completed in Part 3 of the book, you will use the model workflow more realistically in this project. Instead of using it linearly, there will be cases where you must return to a previous step, fix or improve something, and then continue.

Time to start.

1.2.1. 1 - Setup

Open KiCad and create a new project. Give the project a name. I have called mine "Tiny Solar Supply" and saved it in my projects folder.

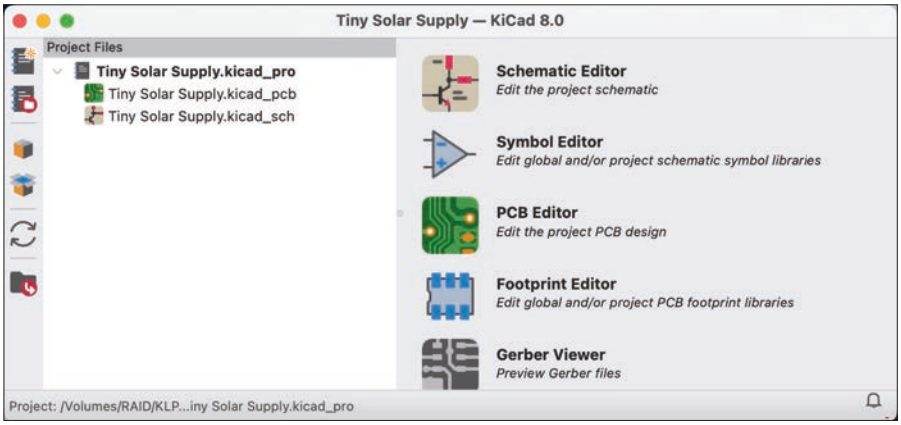


Figure 1.2.1.1: The new KiCad project.

In the project folder, you will see three files:

- The project file: "Tiny Solar Supply.kicad_pro".
- The layout file: "Tiny Solar Supply.kicad_pcb".
- The schematic file: "Tiny Solar Supply.kicad_sch".

Click on the Schematic Editor in the project window to start the schematic editor. The schematic sheet is empty. Let's do a basic setup for the project.

Bring up the Schematic Setup window (File—> Schematic Setup, or click the Setup button from the top toolbar). Review the schematic settings. I will leave these settings as their defaults and add Net Classes later in the project.

Similarly, in the KiCad Preferences window, I will use the default settings. In the Display Options tab (under Preferences—> Schematic Editor), I have set "Snap to Grid" to "Always"and "Cursor Shape" to "Full window crosshair."

In the Editing Options tab, I have selected the Line drawing mode option to "90 deg Angle".

In the Colors tab, I have changed the background color of my theme to white so that the screenshots on these pages look better (the default background color is light grey).

The last setup item in my list is to enter the project details in the Page Settings window. Bring up this window (File —> Page Settings or click the Settings button from the top toolbar). Fill in the text fields with the information you want to show in the sheet's information corner. Also, select a sheet size and enter the issue date. You can see my Page Settings window below:

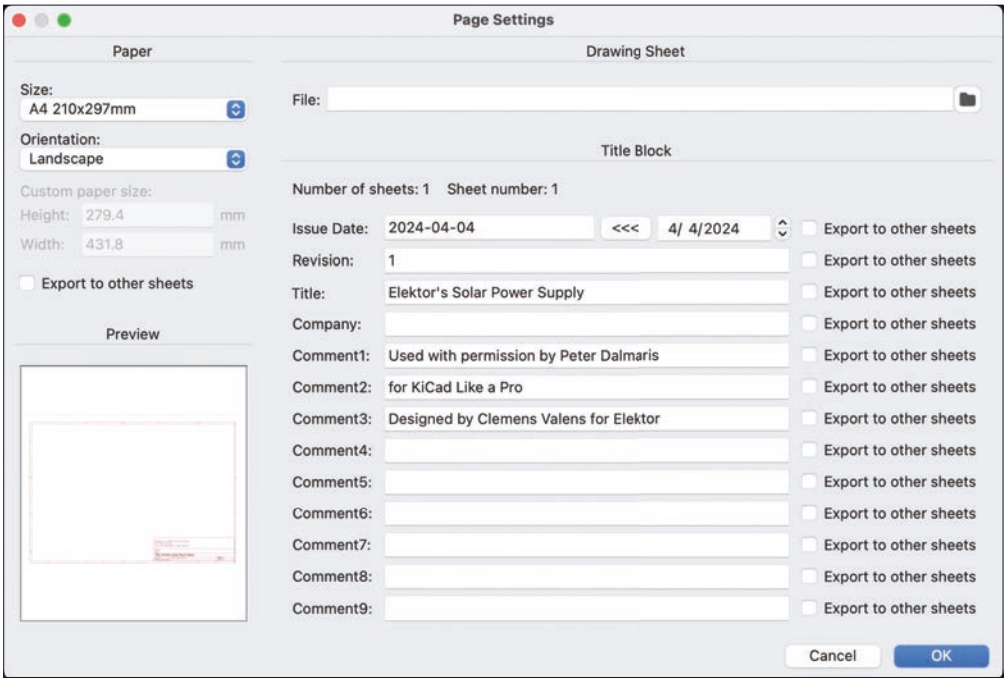


Figure 1.2.1.2: The schematic editor page settings.

This information will appear in the bottom-right corner of the schematic sheet.

The setup is complete; let's continue with step two, adding the component symbols on the editor sheet.

1.2.2. 2 - Symbols

Let's find the required symbols and drop them on the design sheet. Most symbols are available in the KiCad libraries, but you'll have to import one from Snapeda and another from Ultralibrarian. Use the table in the introduction to help you locate the symbols.

Start with the symbols from the KiCad libraries. These are the resistors, capacitors, connectors, and transistors. Here's my sheet at this point:

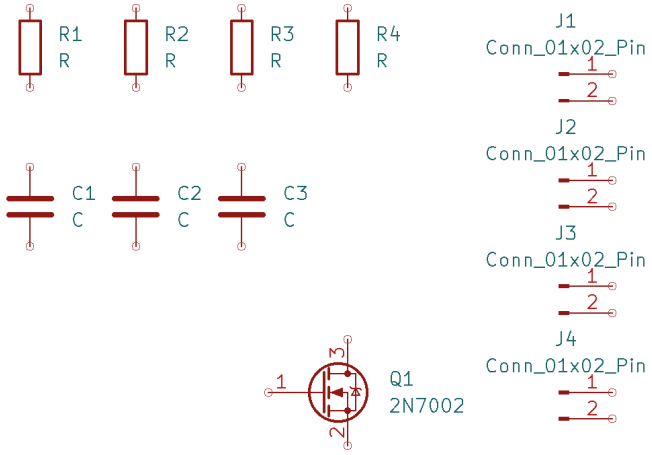


Figure 1.2.2.1: The symbols are from the KiCad libraries.

I could not find suitable symbols for the inductor (L1), the Schottky rectifiers (D1 and D2), and the IC (U1), so I looked online. I found the IC (AP3015AKTR-G1) and inductor on Ultralibrarian and D1-2 on Snapeda:

- U1 (AP3015AKTR Micro Power Step-Up DC-DC Converter): https://txplo.re/AP3015AKTR
- D1, D2 (Schottky rectifier) ASPI-0630LR-100M-T15 inductor): https://txplo.re/ss14 _
- L1 (inductor): https://txplo.re/coil

Download and install these parts' symbols, footprints, and 3D models. The easiest way is to download the ZIP files for the two parts and copy them into the project folder. Below, you can see my project folder after copying the external libraries.

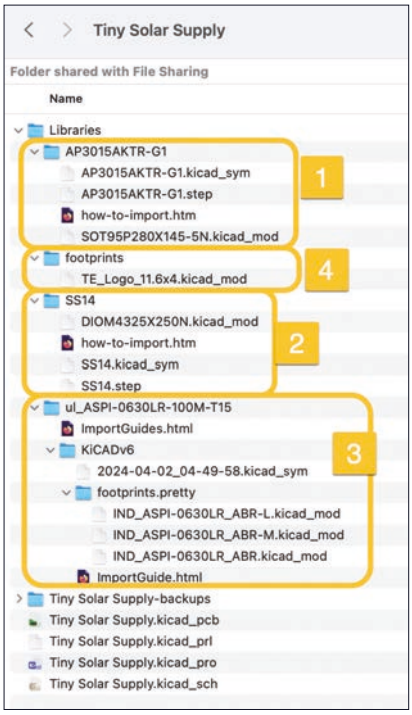


Figure 1.2.2.2: The imported libraries in my project folder.

Regarding the number in the screenshot above, "1" is the U1 integrated circuit symbol and footprint, "2" is the SS14 Schottky diode, and 3 is the L1 coil. I have also added a footprint for the Tech Explorations logo, a non-functional footprint that I will use to decorate the back of the board.

After copying the libraries to the project folder, install them on KiCad. I did this for the project only. You can see my Symbol and Footprint properties windows below:

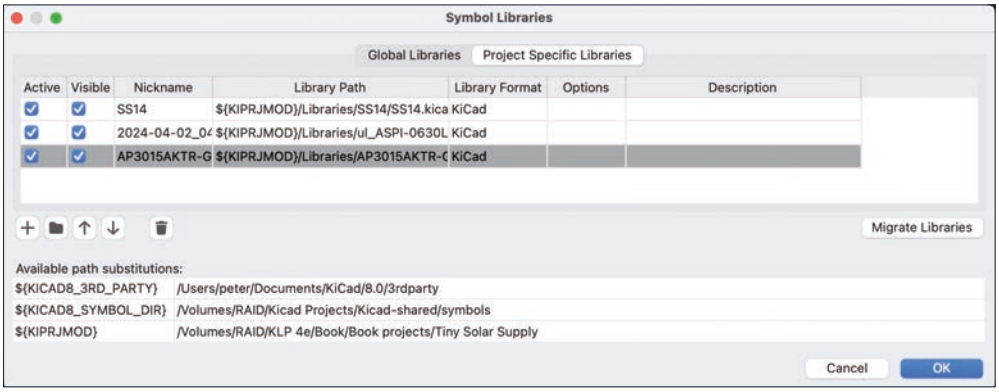


Figure 1.2.2.3: The external symbols installed for the project.

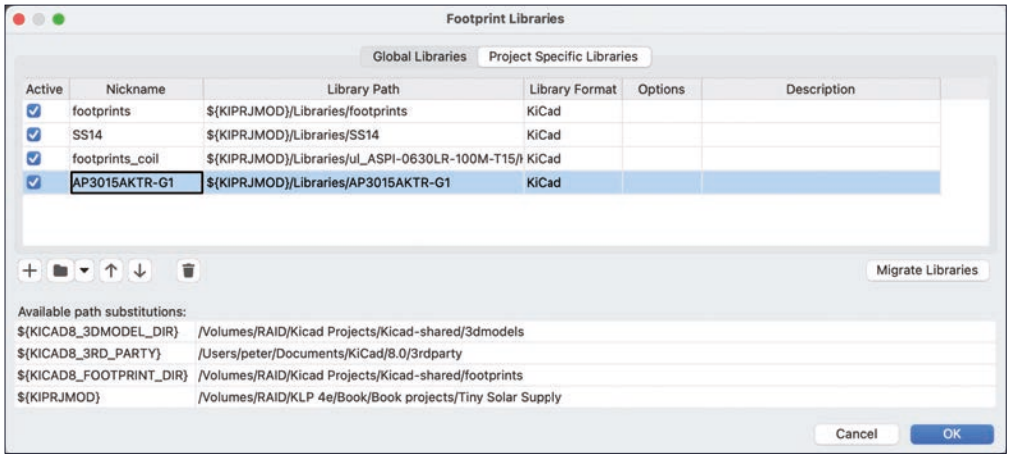


Figure 1.2.2.4: The external footprints installed for the project.

Now that the last required symbols are available for the project, add them to the schematic sheet. Here's my schematic sheet:

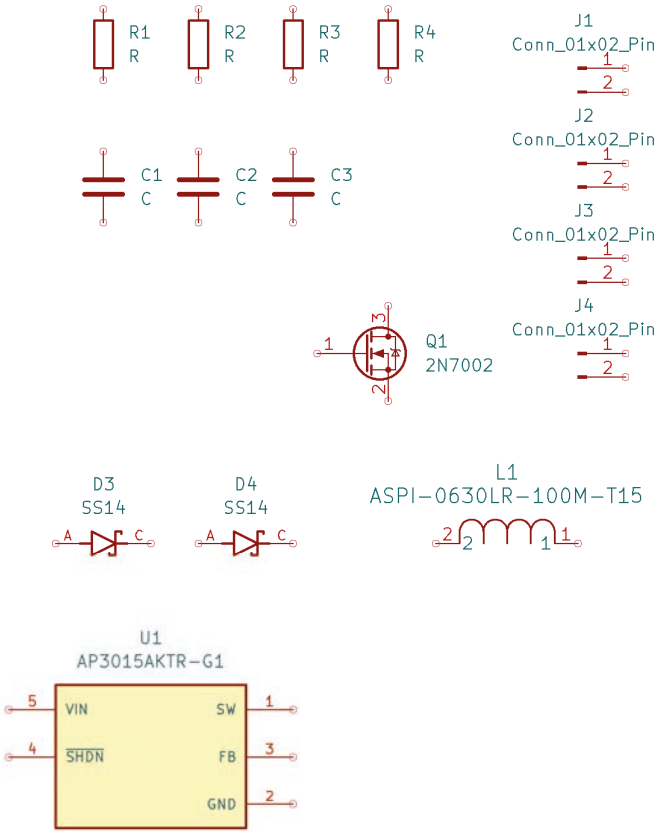


Figure 1.2.2.5: All project symbols are ready to arrange and connect.

Before you continue step three of the workflow, arranging, annotating, and associating the symbols, you will add component values to the capacitors and resistors.

1.2.3. 2 - Edit Component values

Let's continue by editing the value for the capacitors and resistors. The value for each component is included in the introduction of the components table. Unlike how I did this previously (i.e. by opening the properties of each symbol separately), this time, I will edit the component values in bulk using the Symbol Fields Table window. This will allow me to edit all values in one operation. You can open this window by clicking the table button in the top toolbar.

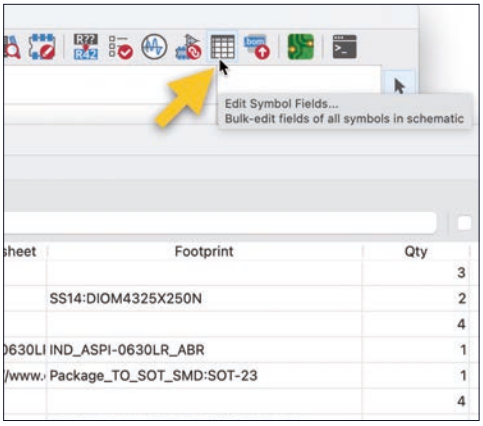


Figure 1.2.3.1: Open the Edit Symbol Fields window.

In this table, components are grouped by type. Click on the ">" symbol to expand each category so that you can edit their value individually. Notice that even though this table also allows you to edit the footprints, I will leave this for later and concentrate on the values for the time being.

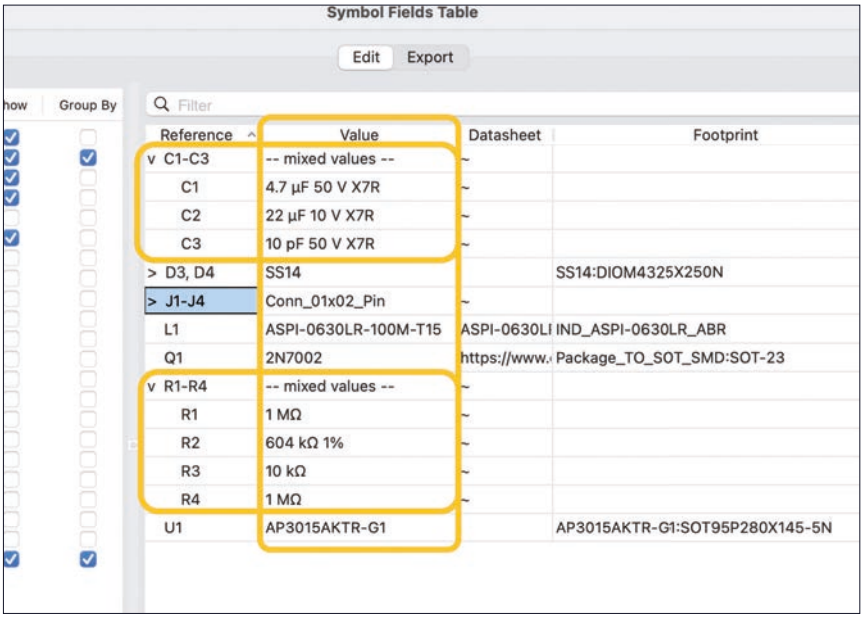


Figure 1.2.3.2: Edited values for the capacitors and resistors in the Symbol Fields Table.

Click OK to submit the changes to the table and close the window. The schematic sheet now should look like this:

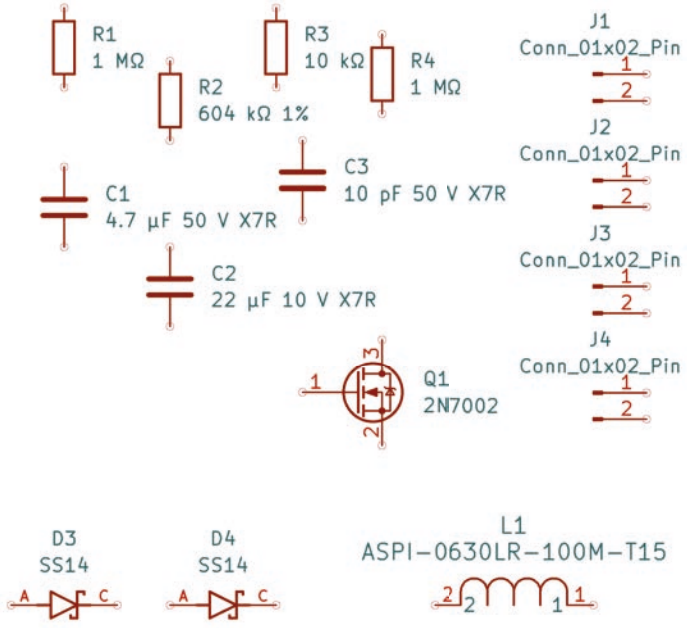


Figure 1.2.3.3: The schematic sheet, with values assigned to the capacitors and resistors.

Step two is complete. Let's continue with Step three, where you will arrange the symbols according to their functional group and associate them with footprints.

1.2.4. 3 - Arrange, Annotate, Associate

Step three of the schematic design workflow is where we:

1. Rearrange the symbols to the (usually) final locations in the sheet.
2. Annotate the symbols with their unique identifiers.
3. ssociate the symbols with the footprints we'd like to use in the layout.

Since KiCad 7, annotation happens automatically when you drop a symbol on the schematic editor sheet (unless you have disabled the auto-annotation feature). Nevertheless, I will take the opportunity to ensure that all symbols are correctly annotated and valued.

Let's begin.

Arrange

This is the easiest part of the entire project. Clemens designed the circuit (in KiCad 6!) and published it on Elektor. So, I stepped on his shoulders and copied his design, starting with the location of the symbols.

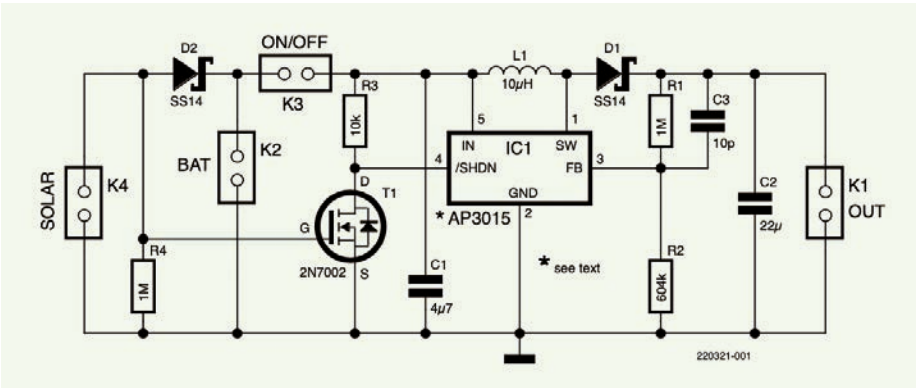


Figure 1.2.4.1: Clemen's design. I will copy it!

I'll move and orient the symbols so they are ready to wire in the next step. I'm using Clemens's original circuit as a guide. Here is my schematic with the symbols arranged:

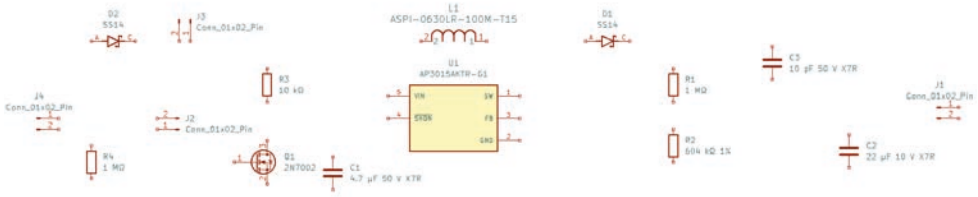


Figure 1.2.4.2: My schematic, with the symbols arranged.

Annotate
In KiCad 8, as with KiCad 7, symbols are automatically given a unique identifier when you add them to the schematic sheet. As a result, I don't have any symbols that require annotation. However, I will take this opportunity to inspect all symbols and ensure that their reference designators match those shown in the parts table in the introduction of this project.

Once I'm satisfied that they do, I will continue with the association task.

Associate
In this task, I will associate symbols with footprints. I will associate the resistors, capacitors, pin headers, and transistors with footprints available in KiCad's footprints library and the coil, diodes, and DC/DC converter with footprints in the external libraries. You should have installed these external footprints and their symbols at the start of this project. Use the table in the introduction for guidance on the footprints for each symbol.

The only through-hole components on this board are the pin headers. Below, you can see my associations table:

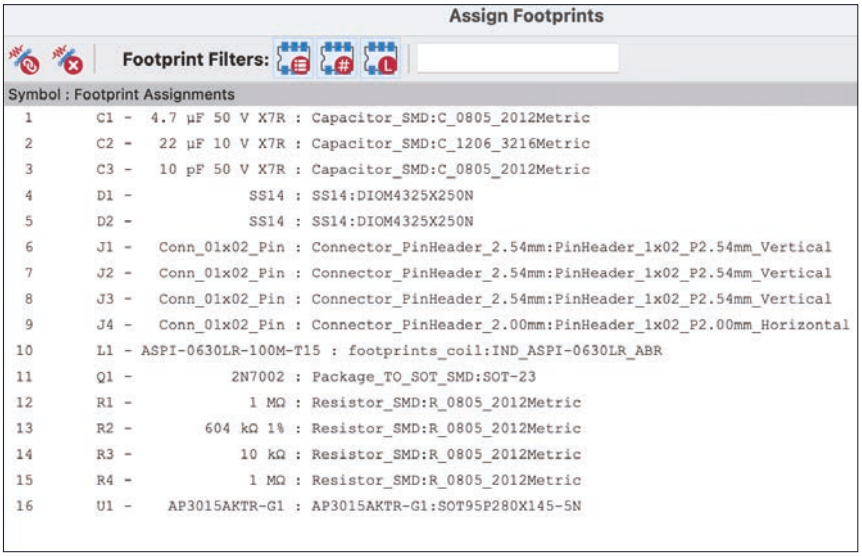


Figure 1.2.4.3: The symbol and footprint associations.

This completes step three of the workflow. Let's continue with the wiring in the next segment.

1.2.5. 4 - Wiring

In this segment, you will complete step four of the schematic workflow, adding wires.

There are two ways to connect symbol pins:

1. Use line wires.
2. Use labels.

This is a simple circuit, so I will use wires everywhere. I will use labels to name nets rather than to create connections.

The wiring will be easy because you arrange the symbols according to the function in step three. You will make most connections between pins that are close to each other.

As with previous projects, I will remind you of these guidelines:

- Keep wire length to a minimum.
- Avoid drawing a wire over another wire.
- Use 90-degree angles.
- Use line wires to connect pins within the same functional group.
- It is OK to move symbols so they are in a better position for wiring.
- Add power flags and other power nets to the ground.
- If you have difficulties completing a wire, try repositioning the symbol as it may not be properly aligned with the grid.

These guidelines generally help create clean, readable schematics. I wired the symbols by starting from the left and working my way to the right.

Before you begin, here's an invaluable tip: select all symbols, right-click to reveal the contextual menu, and select "Align items to Grid." This will ensure that the symbols are properly positioned on the grid so that their pins can be used to start or finish a wire.

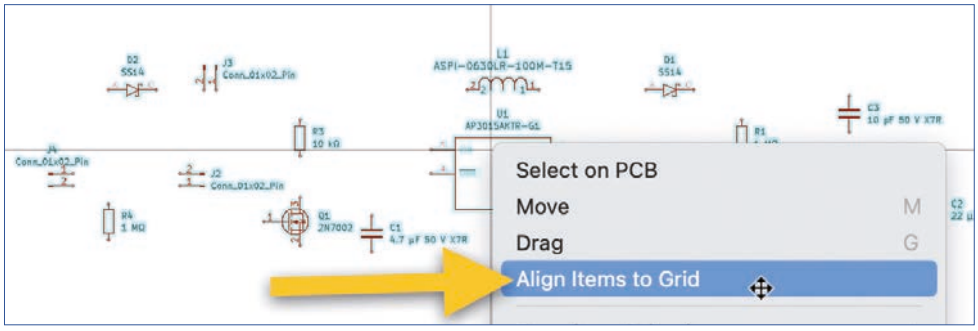


Figure 1.2.5.1: This function ensures that wiring works everytime.

Use the wire tool to wire the circuit fully. Here is mine after this work is complete:

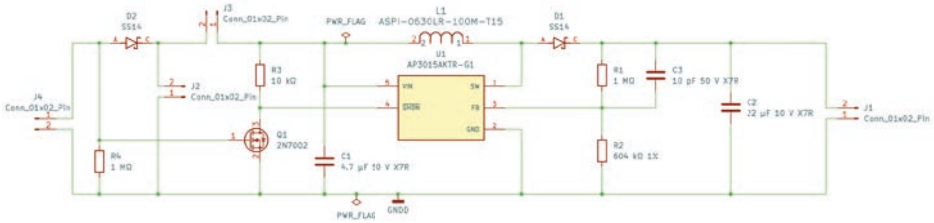


Figure 1.2.5.2: The fully-wired circuit.

Notice the GNDD symbol at the bottom of the circuit and the two PWR_FLAG symbols. What is GNDD, and what is the difference between it and the regular GND symbol we have used elsewhere? Let's have a quick look at this now.

In electrical schematics, the distinction between "GND" and "GNDD" symbols often pertains to the types of ground they represent, serving to differentiate between various ground domains within a circuit for clarity and functional purposes. Here's a brief overview:

"GND" (Ground) is the common reference point in an electrical circuit from which voltages are measured and is a return path for electric current. "GND" is often used as a general notation for ground without specifying the type. It can represent the system ground, chassis ground, or earth ground, depending on the context of the electrical schematic and the design of the circuit.

"GNDD" (Digital Ground) specifically refers to "Digital Ground." This designation is used in circuits with analog and digital components, necessitating the separation of ground paths to prevent noise interference from digital circuits affecting the sensitive analog circuits. The "D" in "GNDD" signifies that this ground is intended for the digital portion of the circuit, providing a common reference point for digital signals and power return for digital components.

The differentiation between GND and GNDD (or similar variations like GNDA for Analog Ground) is particularly important in mixed-signal systems where analog and digital components coexist. By separating the ground paths:

- It minimizes the risk of introducing digital switching noise into the analog sections, which could compromise the performance of analog signal processing.
- It allows for more controlled impedance paths for high-speed digital signals, reducing the risk of signal integrity issues.

Using different ground symbols in a schematic is a part of good circuit design practice, indicating that care has been taken to manage potential interference and ensure the system's reliable operation of both digital and analog components.

This completes step three of the schematic workflow. In the next segment, I have combined steps five (nets) and six (ERC). I have done most of the work on setting up nets, and the ERC for this simple circuit is clear of violations. Nevertheless, I will take the opportunity of the next step to double-check my work.

1.2.6. 5 & 6 - Nets and Electrical Rules Check

I will create named nets using labels and run the Electrical Rules Check in this step. This circuit only has a handful of nets that are important enough to require a name. I have decided to give names to the nets that convey solar panels and battery power and the ones that power the load. You can see these nets in the updated circuit below:

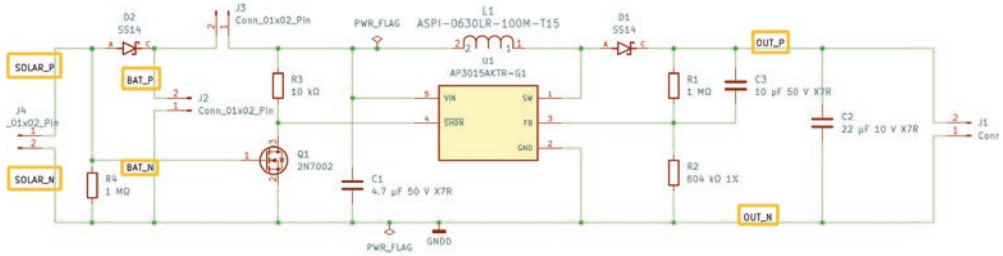


Figure 1.2.6.1: I have added net labels to the schematic.

Let's continue with the Electrical Rules Check. Mine (see below) comes back with only a few warnings. One of those warnings is interesting because it shows a conflict between my "OUT_N" label and the GNDD label. The informative message in the warning indicates that "GNDD" takes precedence and will be used in the netlist.

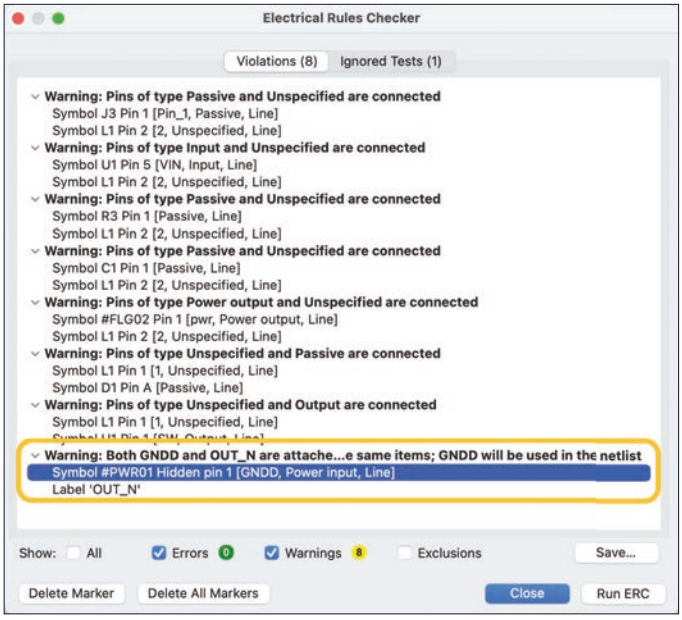


Figure 1.2.6.2: This ERC shows a few warnings only.

While I could have chosen to disregard this warning, I opted to take a proactive approach. I removed the "OUT_N" label and rerun the check. This time, the ERC flagged a similar warning about the BAT_N label, which I promptly removed. A third check yielded yet another warning, this time about the SOLAR_N label. After eliminating all three labels that conflicted with the GNDD net, I decided to overlook the remaining warnings and proceed to Step 7.

1.2.7. 7 - Comments

As in this project, I am working on the original circuit from Elektor, so I will not spend much time adding annotations. I will confine my work to Step 7 of the process to marking the role of the four connectors. This will make it easier to work with them in the layout editor. Clemens also did this in his original circuit, and it is a good idea to copy it here.

Instead of simple text objects, I will create a custom field name at the project level. To add a custom field name, click on the Schematic Setup button ("1" below), then "Field Name Templates" under General ("2"). Click the "+" button to add a new name, and give it a "Short description". Check the "Visible" checkbox so that the values in this field are visible in the schematic by default.

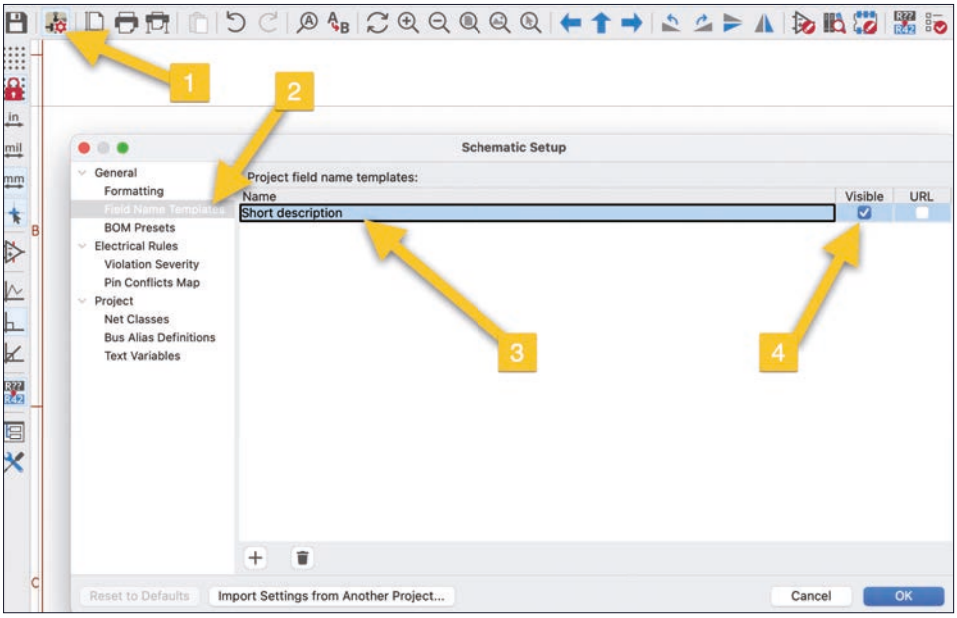


Figure 1.2.7.1: Creating a custom field name.

Let's use the new field. Double-click on J4, and in the "Short description" field, type "SOLAR". See below.

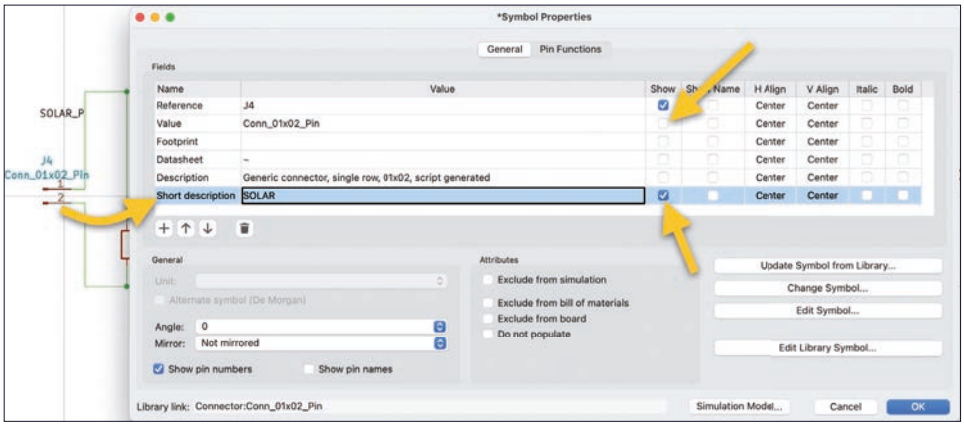


Figure 1.2.7.2: Giving a short description to J4.

To help declutter the schematic, I will uncheck the "Show" radio button for the Value field (see above). Click OK to close the properties window. The short description of J4 appears next to it.

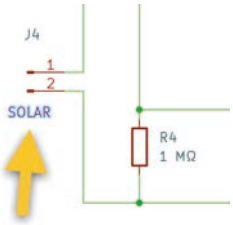


Figure 1.2.7.3: J4 now has a description of its role.

Do the same for J3 ("ON/OFF"), J2 ("BAT") and J1 ("OUT"). Here is the updated schematic:

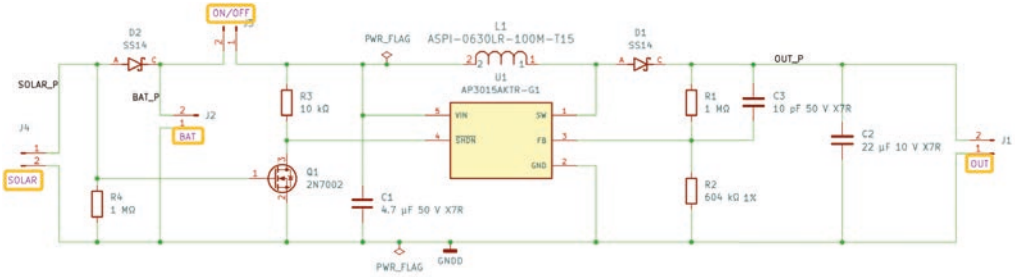


Figure 1.2.7.4: Added helpful information to the connectors.

This completes the schematic design workflow. Ready to proceed with the layout? Follow along in the next chapter.

1.3. Layout

In the previous chapter, you completed the schematic design of the breadboard power supply PCB. In this chapter, you will work on the layout design following the layout design workflow from Part 6 of this book. You can see this workflow below:

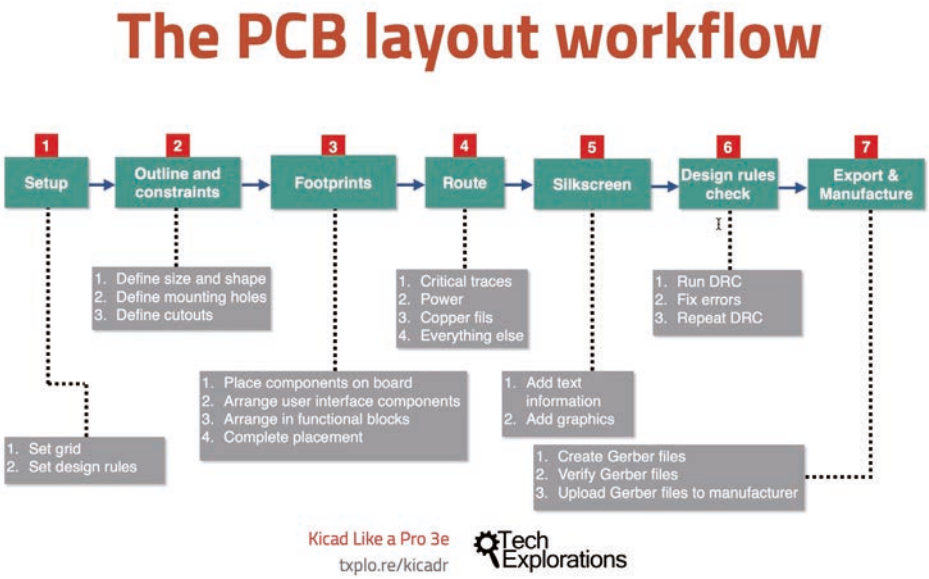


Figure 1.3.1: The layout design workflow.

In the layout work that I document here, my main objective was to produce a board that was as small as possible. I plan to design a small 3D-printed plastic box and use four small screws to secure the board in the box.

Another requirement is to position the pin headers on the left and right edges of the board for easier access. Unlike Clemens's original design, I have opted for 90° headers that will require openings on the right and left side, requiring openings on the project box's right and left sides of the project box.

Finally, I will place the ON/OFF switch to be accessed from the top of the box.

Unlike the breadboard power supply project, I am not bound by specific size constraints in this Tiny Solar Supply project. I am more motivated to make this board as small as possible to reduce cost and design a matching project board in a CAD program like FreeCAD or Fusion 360. My 3D design skills are very rusty, so I will work on this box after finishing this book!

Let's get to work.

1.3.1. 1 - Setup

In this segment, you will set up the layout editor and import the PCB data from the schematic editor. I will follow the same process as in the PCB power supply project for consistency.

Start the layout editor. The first thing to do is import the PCB data from the schematic editor. This is important to do before you make changes to the net classes because the layout editor will not know about the nets in the schematic until the schematic is imported.

Click the PCB import button from the top toolbar to bring up the importer window.

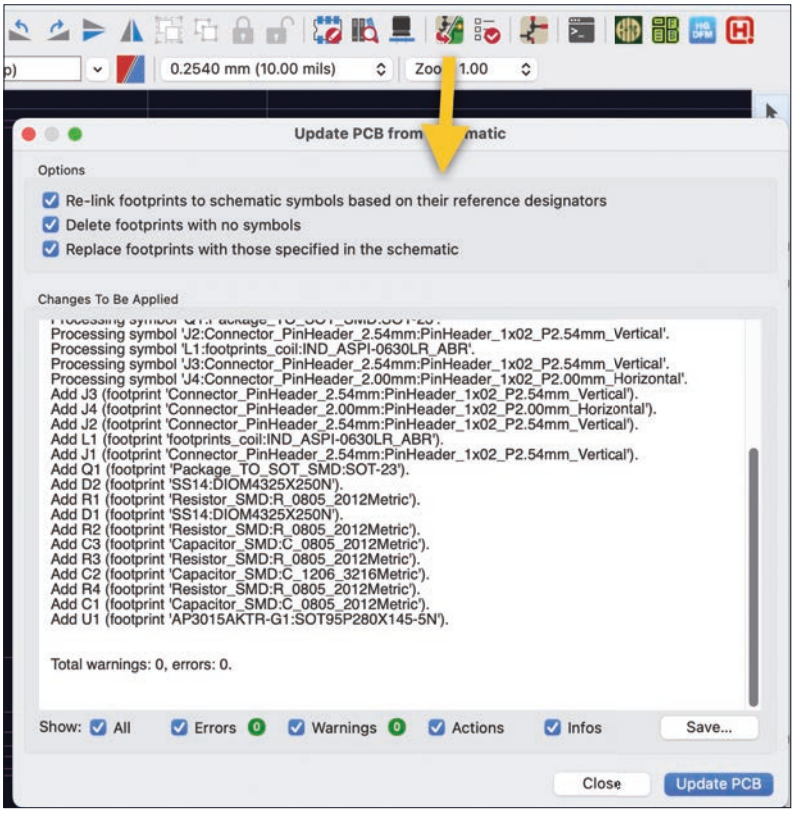


Figure 1.3.1.1: Importing the schematic into the layout (PCB) editor.

Now, let's set the layout editor. Open the Board Setup window and review the most important settings.

- Under Board Stackup:
  - Physical Stackup: confirm that two copper layers are selected.
  - Other settings under Board Stackup are OK in their defaults.

- Text & Graphics:
  - No changes needed.

- Design Rules:
  - Constraints are OK as they are. I have used these constraints with boards manufactured with oshpark.com and nextpcb.com and had no issues.
  - Pre-defined sizes: It is okay to leave those blank for now.
  - Net Classes: I have created a single net class named "Power_in_out" for the power nets. For this class, I have set the track width to 0.35 mm and the via size and hole to 0.9 mm and 0.5 mm, respectively. This will allow for more current to flow through the member tracks. I have assigned the three nets that end with "_P" to the net class using the regular expression "/*_P".
  - Custom Rules: no change to the defaults.

- Violation Severity: no change to the defaults.

Below, you can see the settings in the Net Class table.

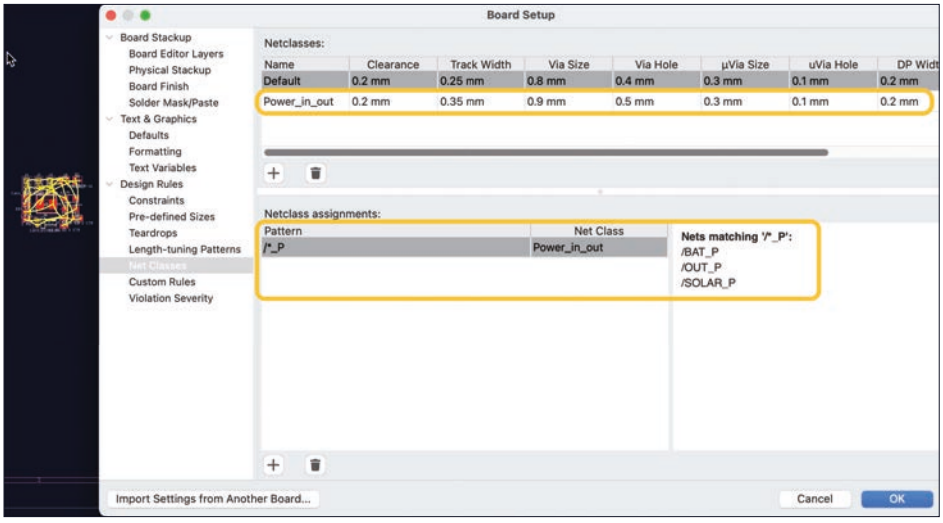


Figure 1.3.1.2: I have created a net class, and assigned three nets to it using a simple regular expression.

At this point, the footprints are now clamped together in the layout editor:

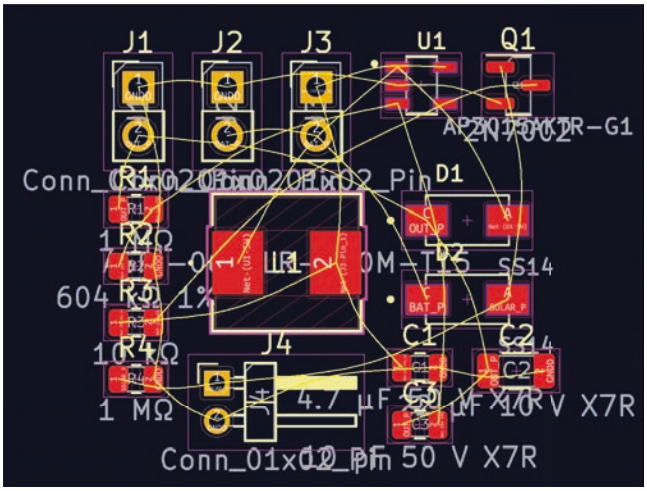


Figure 1.3.1.3: The footprints after importing the schematic into the layout editor.

Let's continue with drawing the outline of the board.

1.3.2. 2 & 3 - Placement and outline

I will combine steps 2 and 3 so that the rough outline of the board closely encloses the components as I place them close to where they will end up in the final version of the board.

Remember that I don't have externally dictated dimensions for the board in this project. My objective is to create a board that is as small as possible. As a consequence, the size of the board is limited by the physical space that the components require on it.

Therefore, I will start by placing the components in the editor per the requirements outlined in the layout introduction. After that, I will draw the board outline in the edge cuts layer.

Here are the placement requirements in some more detail:

- Place the J4 (solar panel) and J2 (battery) connectors on the left side of the board.
- Place the J1 connector (output) on the right side of the board.
- Place the J3 connector (ON/OFF switch) around the middle of the board.
- Include four small mounting holes at the four corners of the board.
- Place the rest of the components between the connectors as closely as possible, keeping in mind these considerations:
  - Minimise track widths and need for vias.
  - Place capacitors close to their active components.
  - As these are SMD components, don't place them so close to each other to make hand-soldering difficult.

With the above requirements in mind, I have placed the components like this:

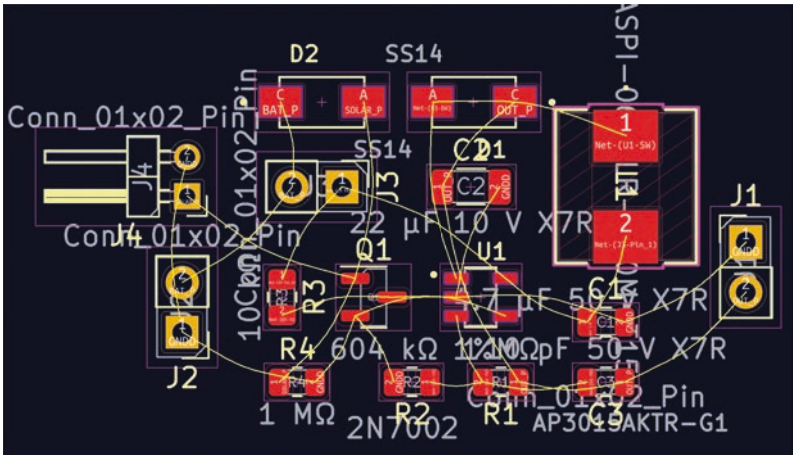


Figure 1.3.2.1: Draft component placement.

This placement is sufficient to allow me to continue with the outline. However, there are three footprints I have noticed that I should change before I continue:

- Footprints J1 and J2 should change to the 90° angled header.
- Footprint J3 should change to a slide switch.

I'll make this change now and then continue with the outline. I will change these footprints one at a time. Start with J2. Double-click on the footprint to bring up its properties, then click the Change Footprint button.

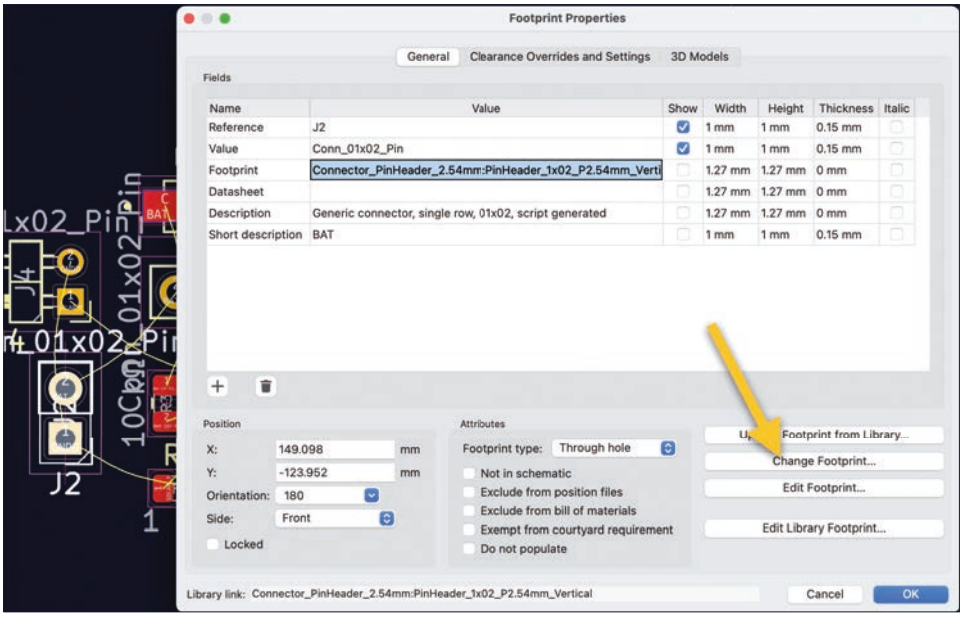


Figure 1.3.2.2: Changing the footprint for J2.

Choose the horizontal 2.54mm pin header ("Connector_PinHeader_2.54mm:PinHeader_1x02_P2.54mm_Horizontal"), as per the screenshot below:

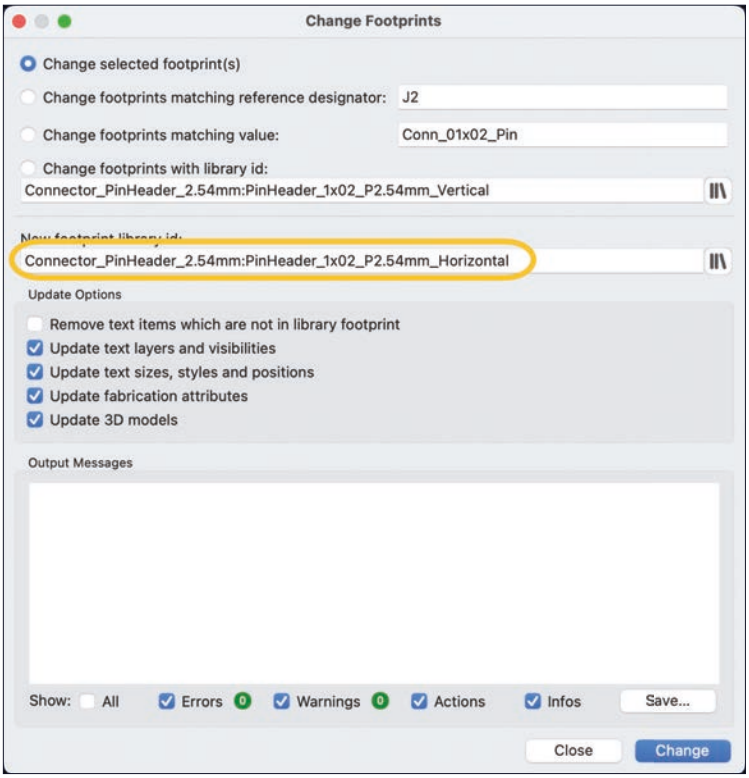


Figure 1.3.2.3: Assigning a new footprint to J2.

Use the same new footprint for J1. For J3 (the On/Off switch), I am using the footprint with ID "Button_Switch_THT:SW_DIP_SPSTx01_Slide_9.78x4.72mm_W7.62mm_P2.54mm".

Here is my layout at the moment:

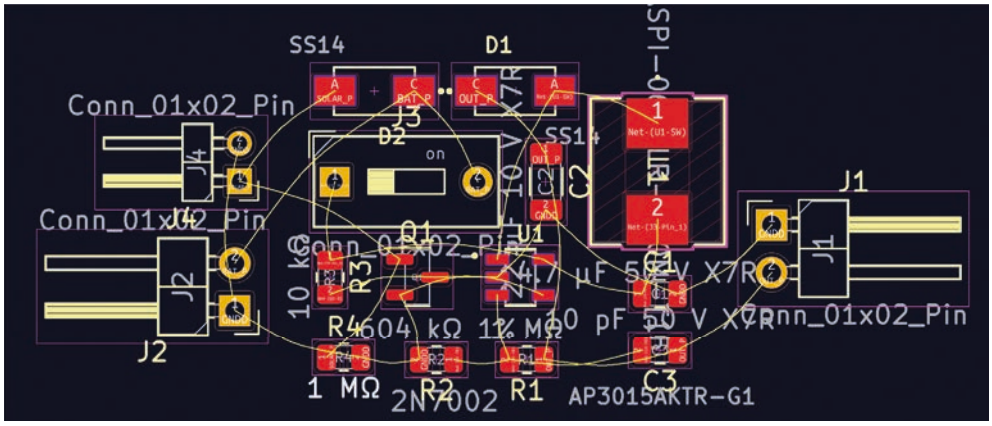


Figure 1.3.2.4: The component placement after the change of the J1, J2, J3 footprints.

After changing the footprints, I also moved and rotated smaller components to make everything fit nicely, reduce track lengths, and minimise crossing ratnest lines.

I can now continue with the outline. Select the Edge.Cuts layer and the rectangle drawing tool. Draw a rectangle around the components, as in the screenshot below.

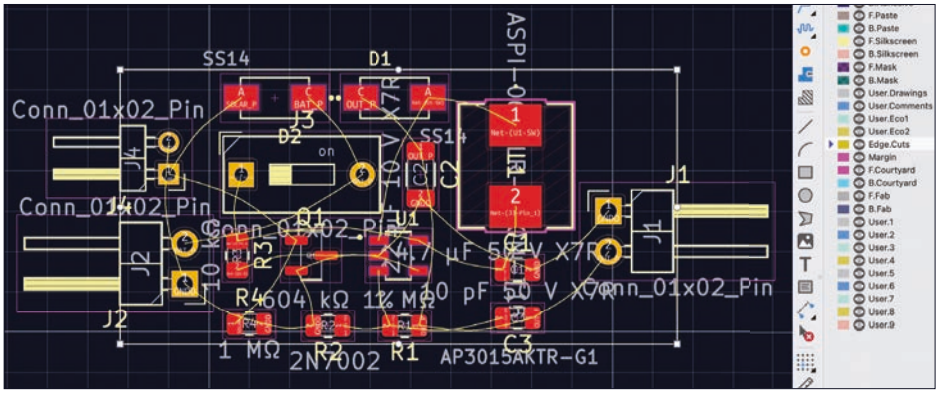


Figure 1.3.2.5: An outline encloses the board components.

In the screenshot above, notice that the yellow outline encloses all components. However, the pins of the J2, J3, and J4 connectors extend outside of the board limits. That is OK, and it is by design. If you prefer that the board's substrate material protect the pins, you can outline so that all of the board, including the pins, are contained within the yellow outline.

Let's continue with step 4 of the process, which is drawing the tracks.

1.3.3. 4 - Route

In this step, you will complete step four of the layout workflow and route the board. Here is the plan:

1. I will use the top and bottom copper layers.
2. I will use the bottom copper layer for GND routes and the top for all other routes.
3. I will use a copper fill in the bottom layer and connect the fill to the GND net. For this reason, I will not do any manual routing between pads that belong to the GND net. I will create the copper fill in the next segment in this chapter (step four of the workflow).
4. I will allow the layout editor to automatically set the track width based on the net class settings I set in the setup step.
5. I will use vias to switch a copper route between the top and bottom layers where necessary.

Before starting, review your grid settings in the KiCad Preferences, under PCB editor, and confirm you are happy with them. Here are my grid settings:

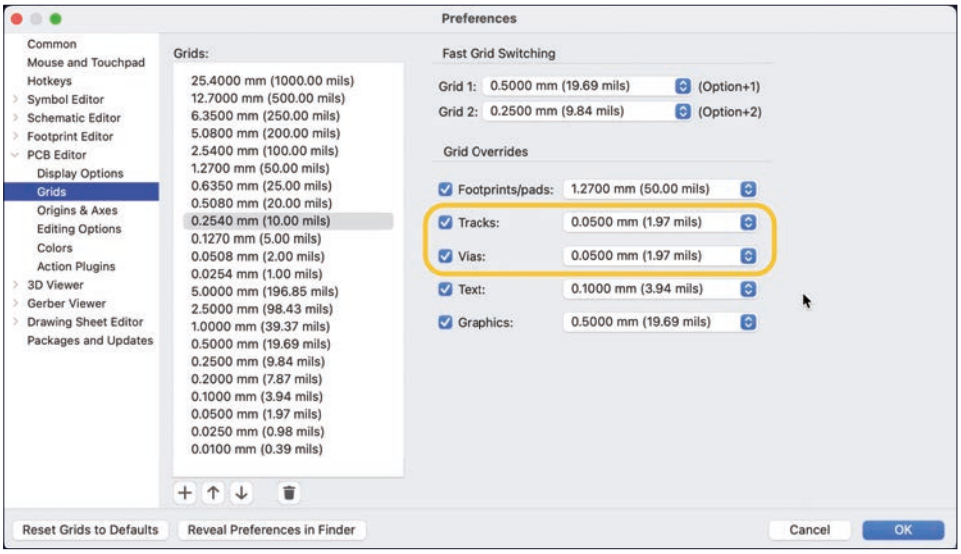


Figure 1.3.3.1: The PCB editor grid settings.

The grids most important for this step of the design process are the Tracks and Vias overrides. I will draw the tracks on the top copper layer from left to right. Use vias only when necessary. Remember, ignore all GND pads and tracks!

Here's my first draft of the non-GND tracks on the top copper layer:

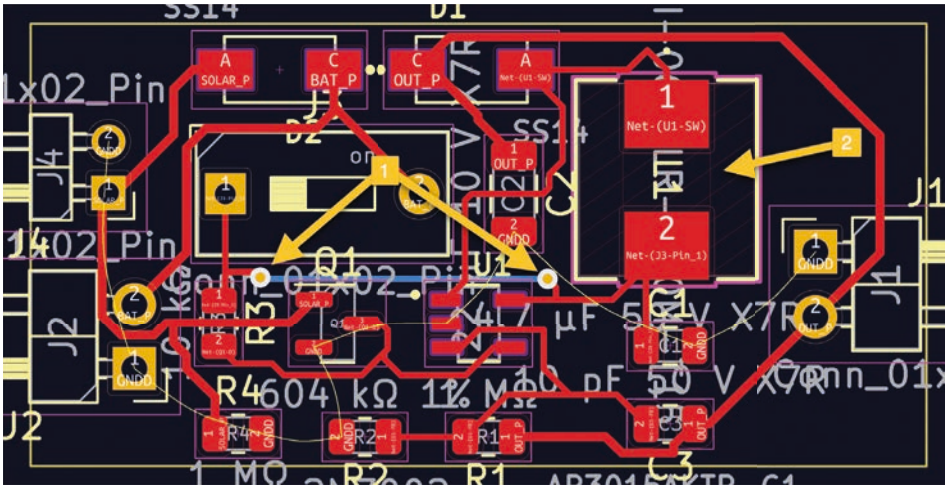


Figure 1.3.3.2: Tracks on the top layer, draft.

I could draw the tracks using only two vias (marked with "1" and arrows above). Also, notice that component L1 (the coil footprint) has thin red diagonal lines that signify a "keep clear" area. This means that tracks (or other components) cannot exist within the marked area. To connect a track to the L1 pads, I had to route them from the footprint's top and bottom side, as you can see in the screenshot.

Also, notice that my tracks have a lot of ugly and 90° angles. Before continuing with the work in the bottom layer, I will improve the tracks on the top side. To do this, I will use the "D" key ("Drag") along with the interactive router. My objective is to remove 90° angles and any redundant segments of the tracks. Here is the result of the optimization work:

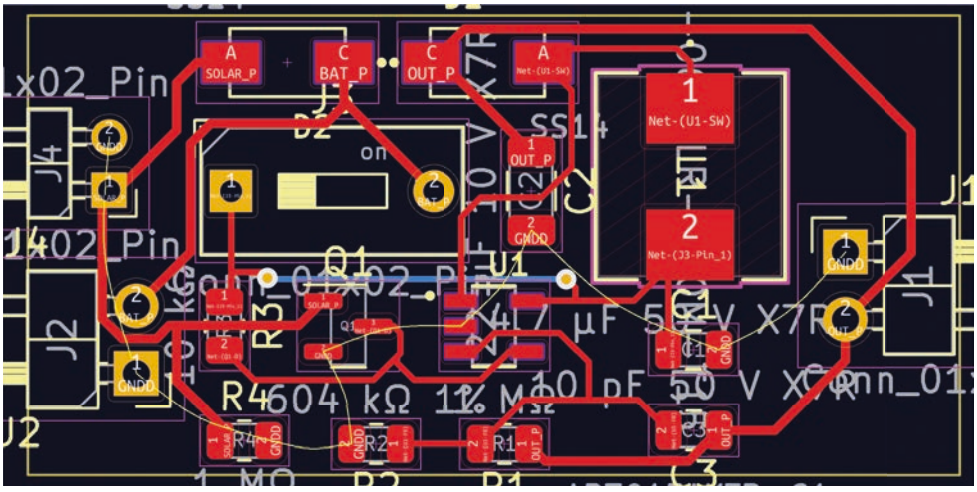


Figure 1.3.3.3: Slightly better track shapes on the top copper layer.

This looks good enough for now. I will take a few minutes to improve the board outline and then continue with the copper zone, which will take care of the GND pads and nets.

1.3.4. 2 - Refine the outline

I am satisfied with my board so far, including the placement of the components and the traces on the top layer. I can take a few minutes now to improve the board layout. I will round the corners and add the mounting holes.

I will start with the rounding of the corners. I will use the technique I described in the PCB power supply project. Because I used the rectangle tool, I will have to delete the entire outline and replace it with straight lines and arc segments. Here is the top right corner of the board:

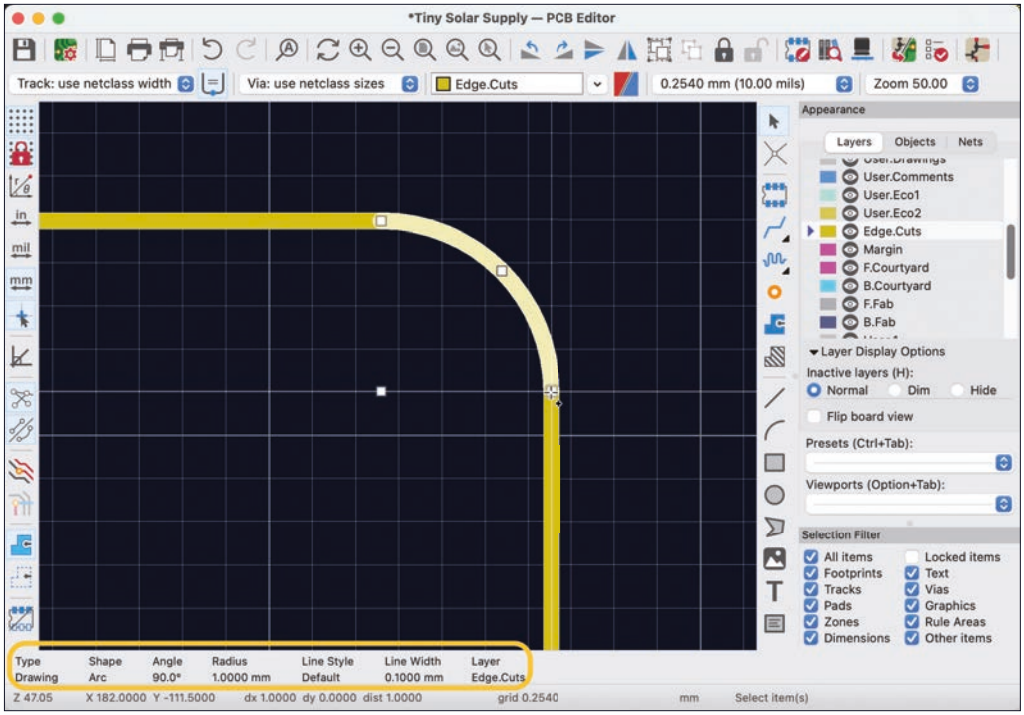


Figure 1.3.4.1: The top right corner of the PCB, with a 1 mm arc segment.

You can see the parameters of the arc segment in the status bar of the layout editor. Rework the other three corners in the same way. You can experiment with duplicating the top-right arc and rotating it as required for each corner instead of drawing it from scratch. Here is the result of this work in my layout editor:

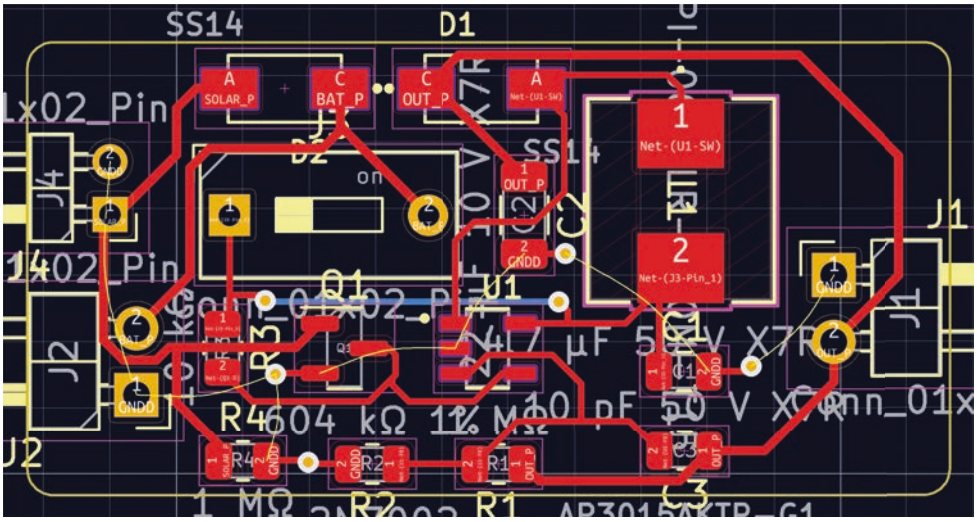


Figure 1.3.4.2: Reworked and improved board outline.

The outline is complete. Let's have a look at the 3D model of this board:

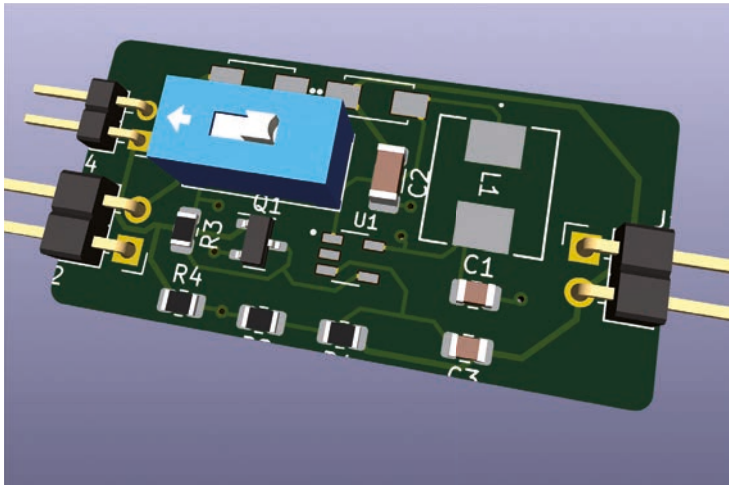


Figure 1.3.4.3: The 3D model of the Tiny Solar Supply

The board is now starting to take its final form. Let's continue with the copper zone.

1.3.5. 5 - Copper fills

Let's continue with the GND pads and tracks. I plan to use the bottom layer for the GND net. However, aside from the headers, all other components are SMDs, which I have placed on the top layer. To expose these nets to the bottom layer, I will use vias and short tracks to connect these vias to the SMD pads on the top side. I describe the process below:

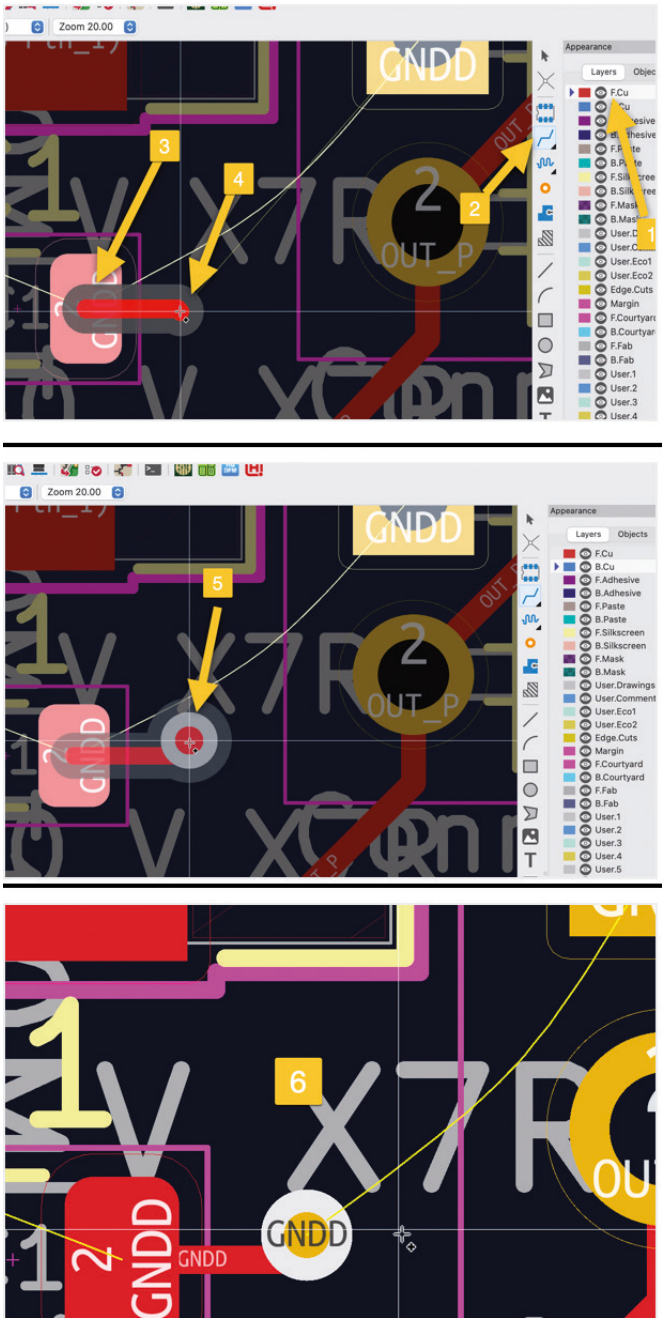


Figure 1.3.5.1: Creating a short track and via to connect top layer GND pads to the bottom layer.

Referring to the screenshots above:

1. Select the top copper layer if not already selected.
2. Enable the Wire tool from the right toolbar.
3. Click on a GND pad and start drawing a track.
4. Draw a short track.
5. Type the "V" key to add a via.
6. Hit the escape key to stop drawing.

That's it. The via type is "through", connecting the top and bottom layers. Here is my layout with the added vias and short tracks to their adjacent pads.

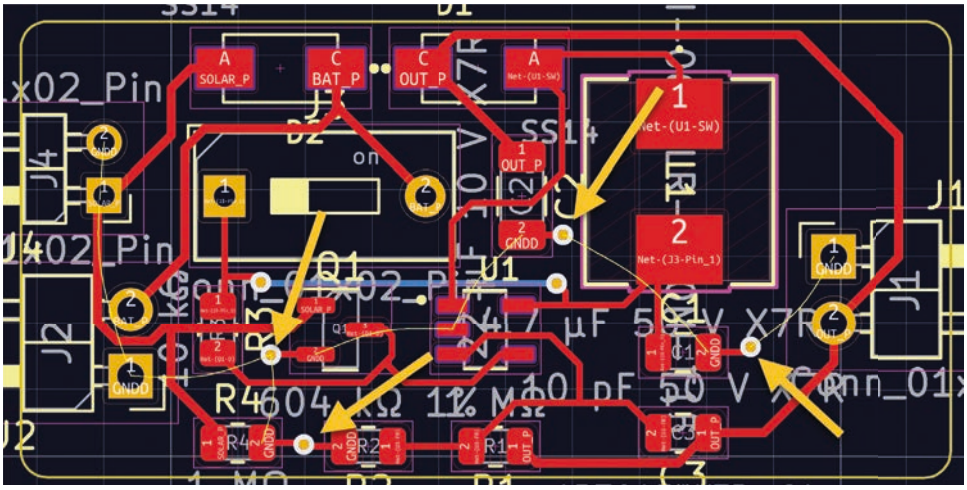


Figure 1.3.5.2: The arrows show the GND vias.

After that, I will create a single copper zone in the bottom layer and assign this fill to the GND net. To create a new copper fill, click the Copper Zones button in the right toolbar and then click near the top-right corner of the board (see below). This will bring up the copper zone properties window, where you can set the zone's layer, net, and other properties. You can see my settings below.

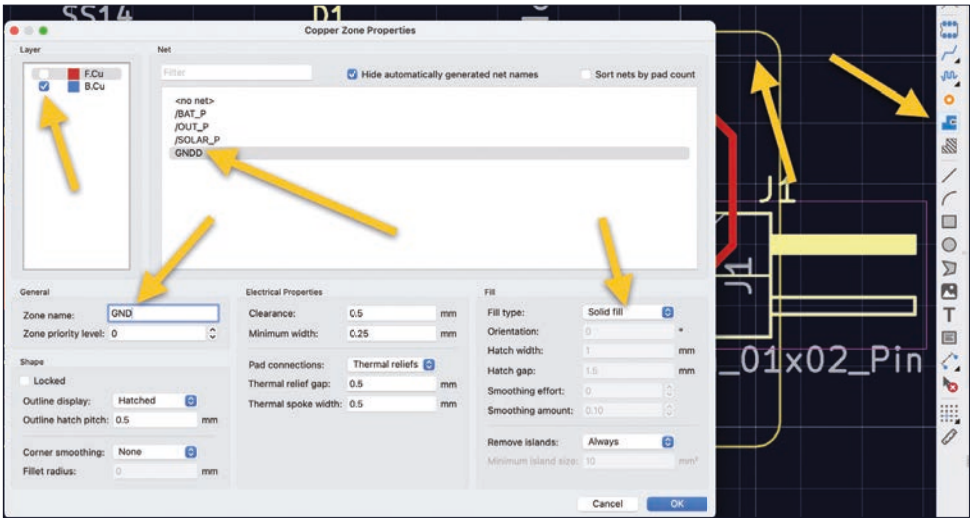


Figure 1.3.5.3: The GND copper zone properties.

I have connected this zone to the GNDD net and placed it in the bottom copper layer. I have also named it a new feature in KiCad 8. Click OK, and continue to draw the zone close to the boundary of the PCB. Here is mine:

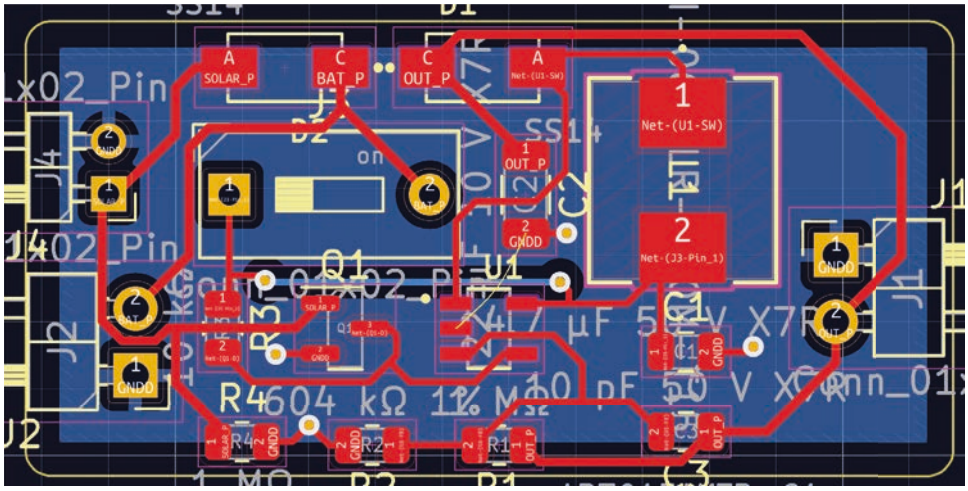


Figure 1.3.5.4: The copper zone in the bottom layer.

Before continuing to step six, run a DRC to ensure no unconnected pads. You can ignore other violations, such as footprint silkscreen issues. My DRC showed one error that I must fix (see below). I forgot to connect the GND pad of U1!

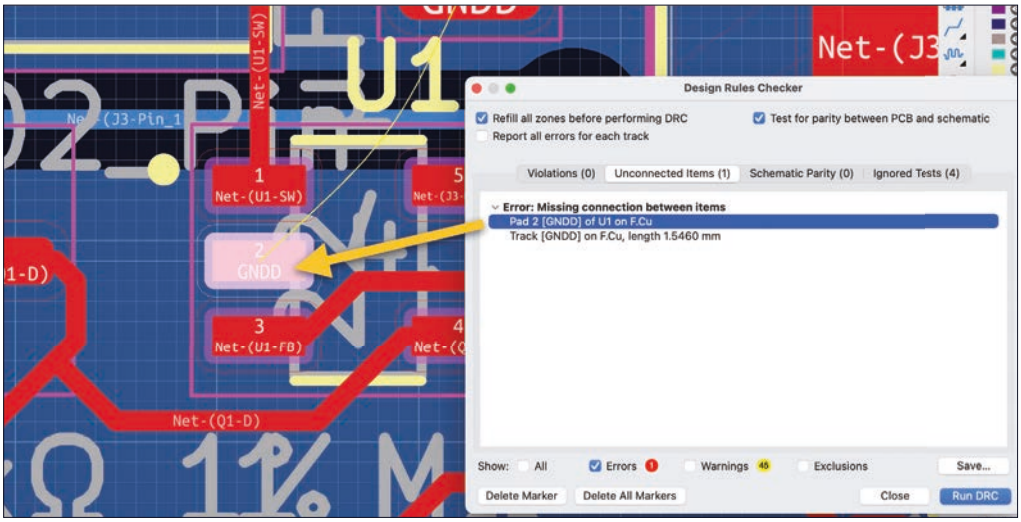


Figure 1.3.5.5: An unconnected pad.

To fix this, I drew a short track on the top copper layer and connected it to a via (see below).

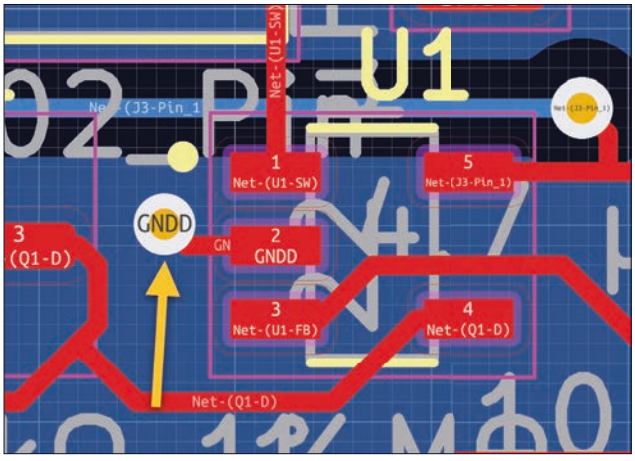


Figure 1.3.5.6: The fix for the unconnected pad.

I re-ran the DRC. The result contained several warnings about silkscreen clippings, as shown below.

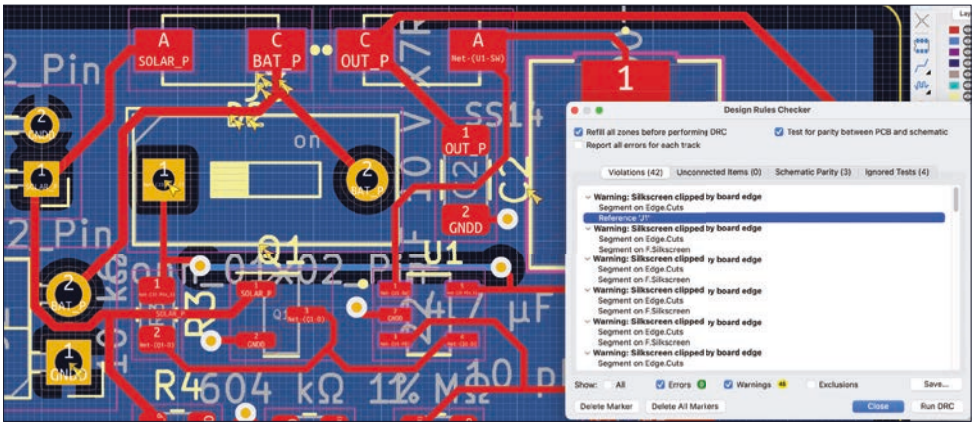


Figure 1.3.5.7: Warnings about silkscreen clippings, which I will ignore.

I will ignore these warnings. In the next step, I will work on the silkscreen and either move the silkscreen objects to eliminate the clippings or remove them from the silkscreen.

1.3.6. 6 - Silkscreen

It's time to add useful board information and decorations in the front and back silkscreen layers. I will add text and graphics in this segment's front and back silkscreen layers. This board already has content in the silkscreen, inherited by its footprints. In general, I endeavor to include the following items in the silkscreen:

- Front Silkscreen:
  - Text labels for the power on/off switch and the other headers.
  - Text labels with the designators for all parts to help with assembly.
  - Other text labels to help me correctly assemble the board, such as "+" and "-" for the diodes.
  - Because of the limited space and to avoid overcrowding, I will not include the values for the resistors and capacitors.

- Back Silkscreen:
  - A version number for the board.
  - The name of the board.
  - A Tech Explorations logo.
  - A KiCad logo.

To reduce clutter in the editor, enable the outline mode for the copper fill (you will find the button in the left toolbar). Enable the F.Silkscreen layer. This is the starting point:

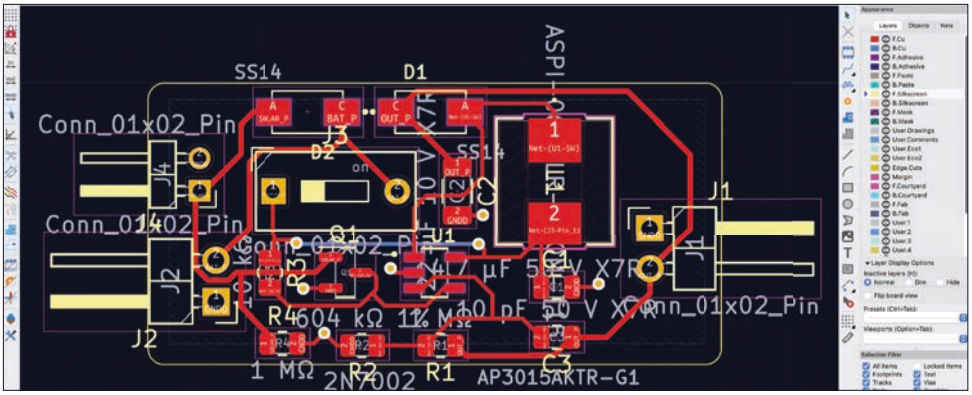


Figure 1.3.6.1: Starting point prior to adding silkscreen text and graphics.

I will start by making all text labels for the component designators smaller. Their current size makes it difficult to place them without clipping. From the Edit menu, select Edit Text and Graphics Properties. For the Scope, select "Reference designators." In the Action group, set the text width and height to 0.8 mm and the thickness to 0.1 mm. The window should look like this:

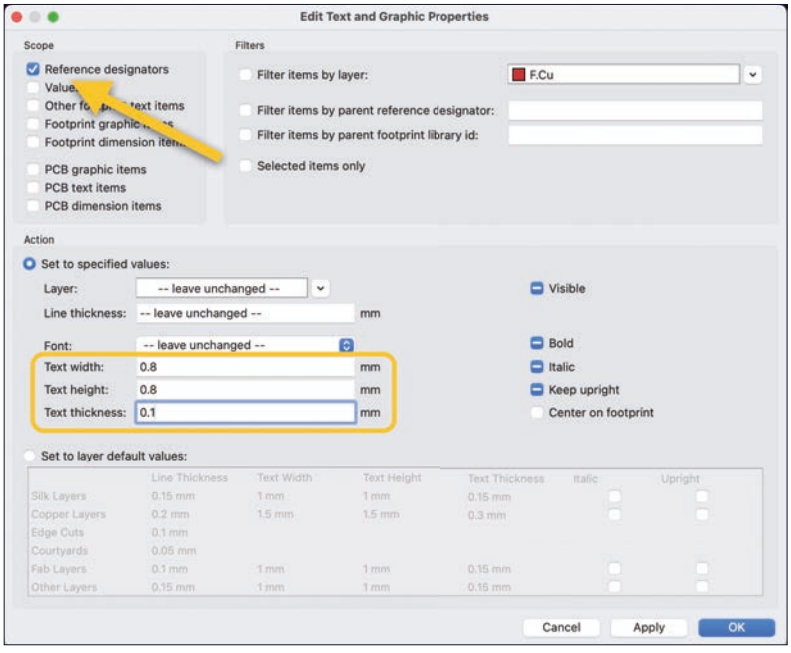


Figure 1.3.6.2: Bulk-change for the reference designators.

Next, I will make invisible any text items that are not reference designators. This will help me de-clutter my board and help me with the positioning of the reference designators. The Properties window looks like this:

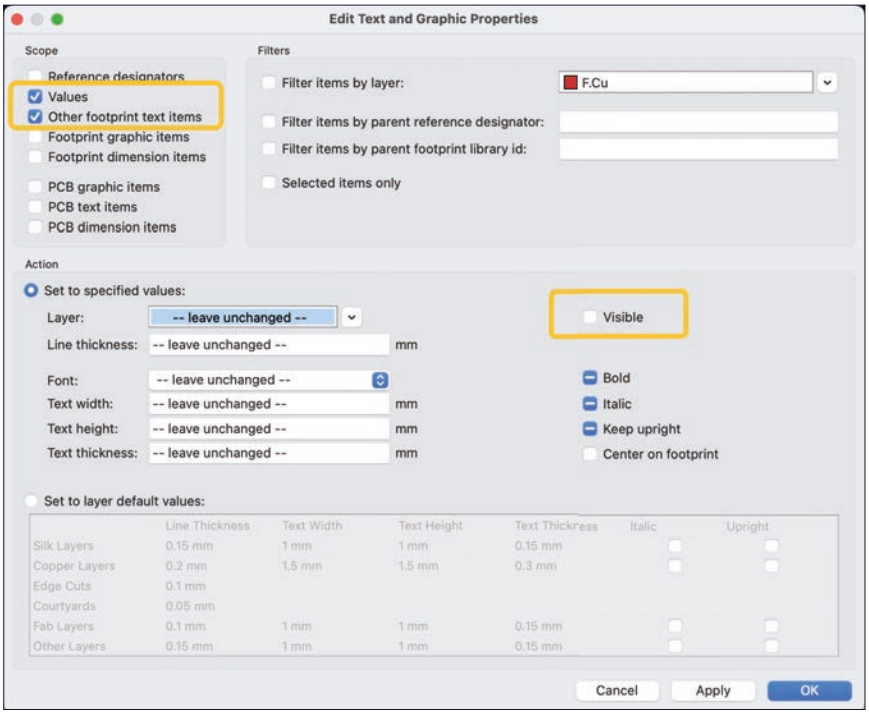


Figure 1.3.6.3: Make all other text objects invisible.

Now, my board looks like this:

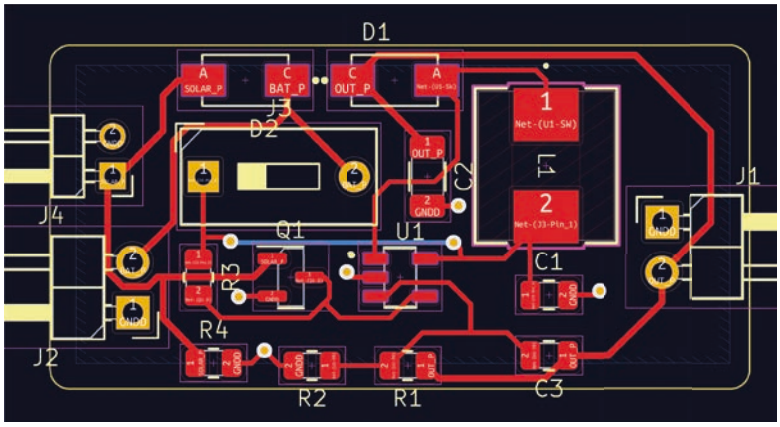


Figure 1.3.6.4: A de-cluttered top silkscreen with only the reference designators visible.

Now, I can position the reference designators. I have also added "-" and "+" text items in the header pins to help me with assembly and use. I have selected only the "Text" option in the selections filter to avoid selecting the wrong object. Here is my board after this work:

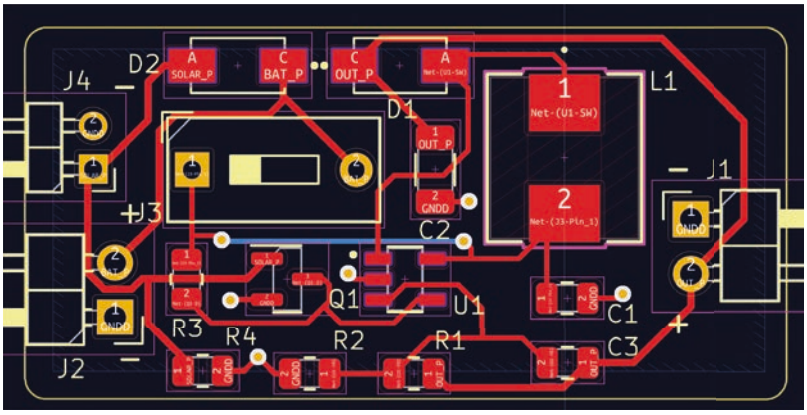


Figure 1.3.6.5: Reference designators and "+", "-" labels positions in the top silkscreen.

In the bottom silkscreen, I will add the following text:

```
Version 1.0
Tiny Solar Power Supply
Created by Clemens Valens for Elektor.
Copied by Peter Dalmaris for KiCad Like a Pro.
```

Here are the properties for this text object:

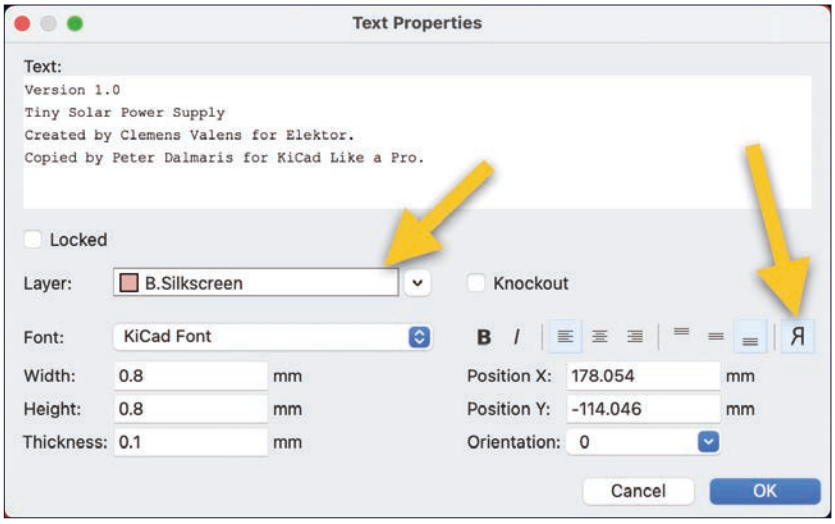


Figure 1.3.6.6: The back silkscreen text block.

I have enabled the reverse mode for this text object, as it is placed in the back silkscreen.

I will also add the KiCad Logo from the footprints selection. Select the footprint tool, and search for "KiCad-Logo2_5mm_SilkScreen". When you place this footprint on the board, it will go on the front silkscreen by default. Double-click it to bring up its properties, and change its side to "Back," as I show in the screenshot below:

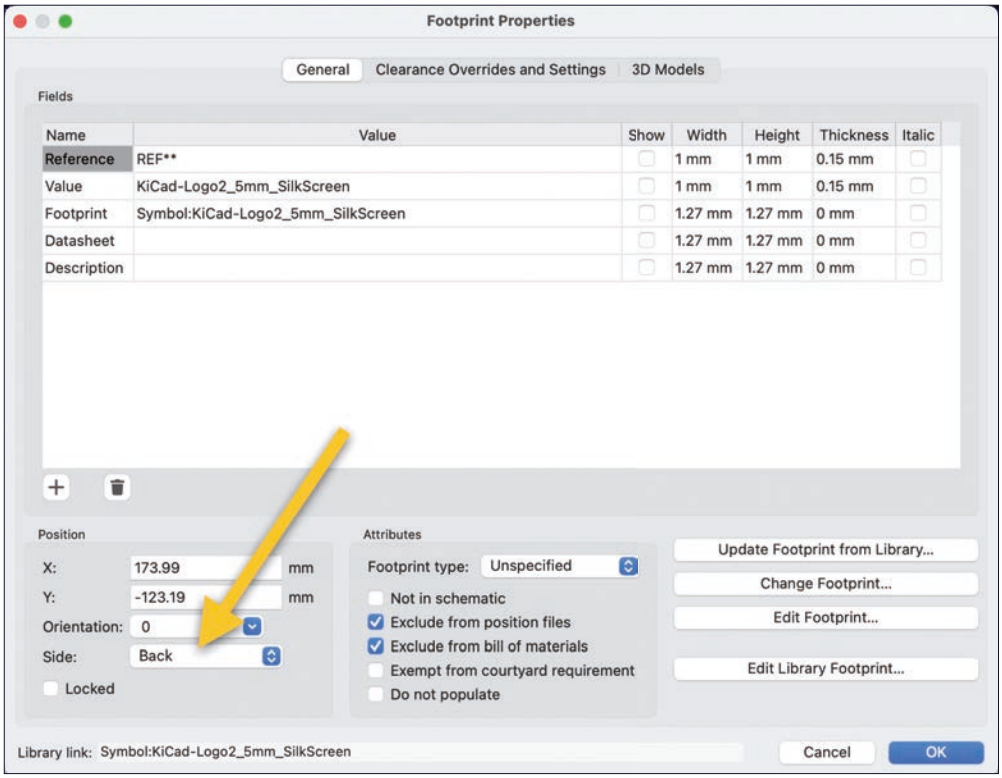


Figure 1.3.6.7: This footprint must be placed in the back silkscreen.

Finally, as always, I will place the Tech Explorations logo on this PCB. Here is the back of the board in 3D:

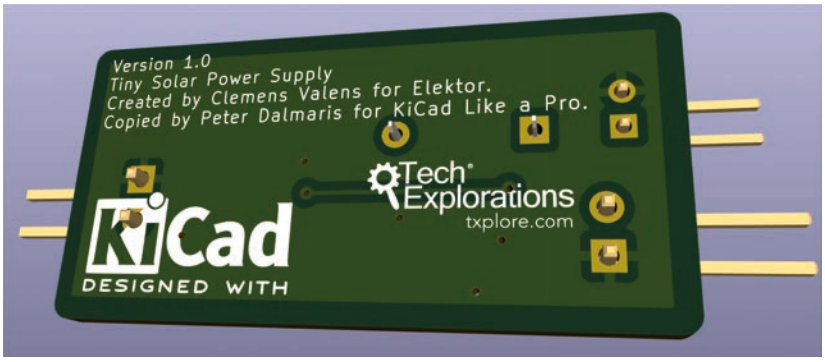


Figure 1.3.6.8: The back of the board, in the 3D viewer.

And here is the front:

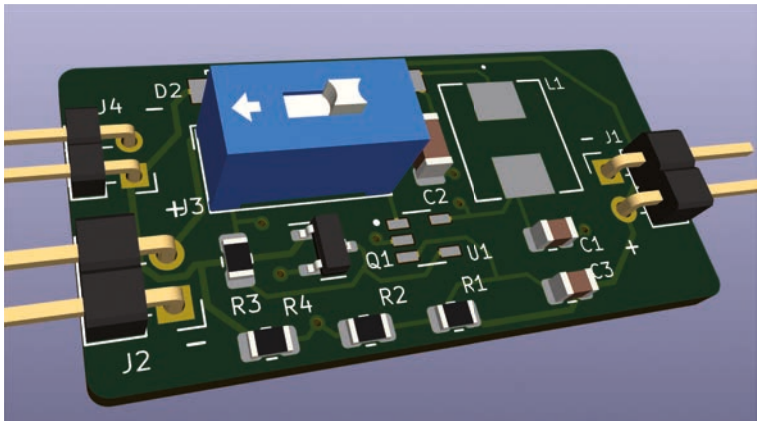


Figure 1.3.6.9: The front of the board, in the 3D viewer.

This completes step six of the layout workflow. Before we complete the project, let's do a quick DRC in the next step.

1.3.7. 7 - Design Rules Check

The Design Rules Check results show a few items I can improve on. Most, however, I can ignore. For example, see the screenshot below:

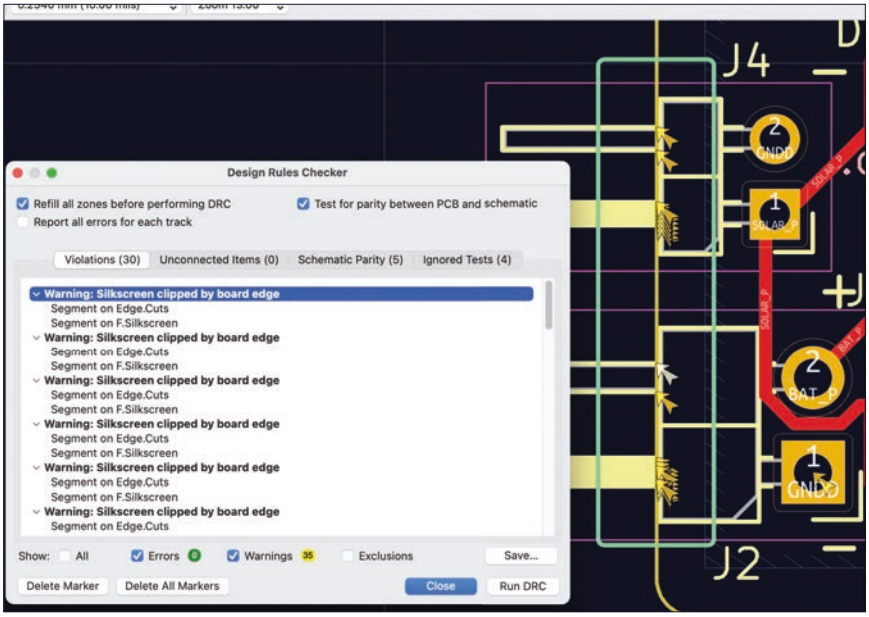


Figure 1.3.7.1: These violations result from the 90° pin header footprint extending beyond the PCB outline.

Several violations relate to my decision to position the 90° pin headers so that their pins extend outside the PCB outline. However, this does not affect the manufacturing of the board, as you will confirm in the next step. Therefore, I can safely ignore these violations.

There are a couple of other violations that I should address, though. In the example below, you can see that the KiCad logo has a part that overlaps with the solder mask around one of the pins of a pad. This can cause manufacturing problems and is best to address. To address this violation, I will move the KiCad logo to a new location where this overlapping does not occur.

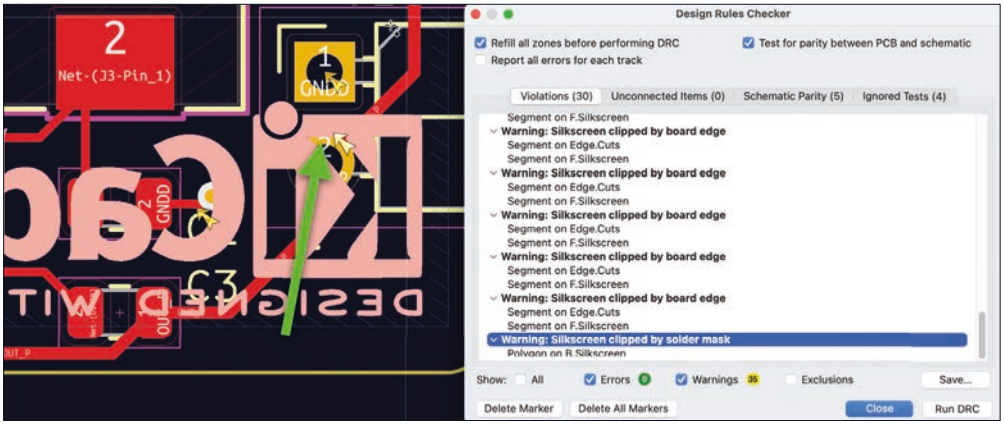


Figure 1.3.7.2: The KiCad logo overlaps the solder mask of one of the connectors.

After moving the logo to a new location, this violation was resolved. Here is the latest version of the board:

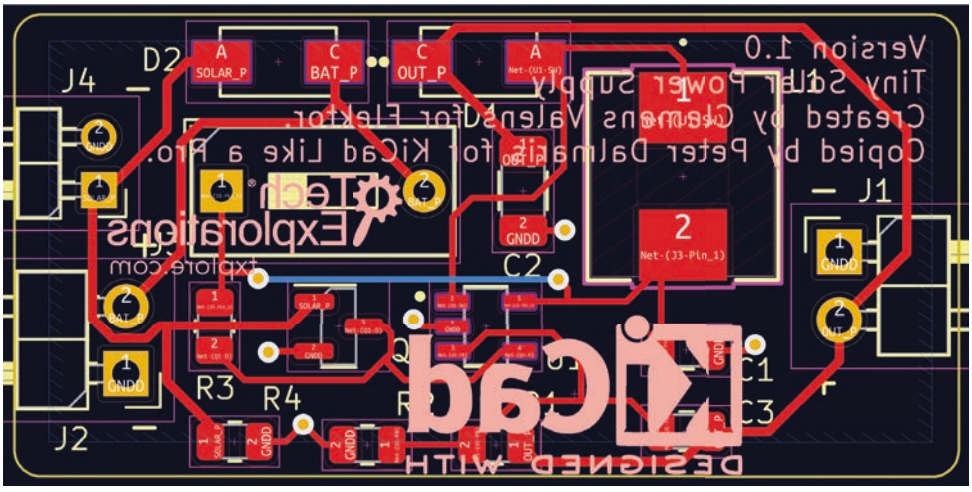


Figure 1.3.7.3: The final version of the board.

I am comfortable sending this board to manufacturing. But before I export the Gerber files in the next step, I'd like to add the missing 3D models (for L1, U1, D1, and D2) to complete the board's 3D representation. I must also add the mounting holes, which I neglected earlier. Let's do both in the next segment.

1.3.8. Two more things: 3D models and mounting holes

We will use the 3D models for the components from the Ultralibrarian and Snapeda libraries. You probably already have the 3D model ".step" files, but if you don't, go to Snapeda or Ultralibrarian using the links in the introduction of this project to get them. For example, you can get the 3D ".step" file for the L1 component from Ultralibrarian by expanding the 3D CAD Model format in the download page (see below).

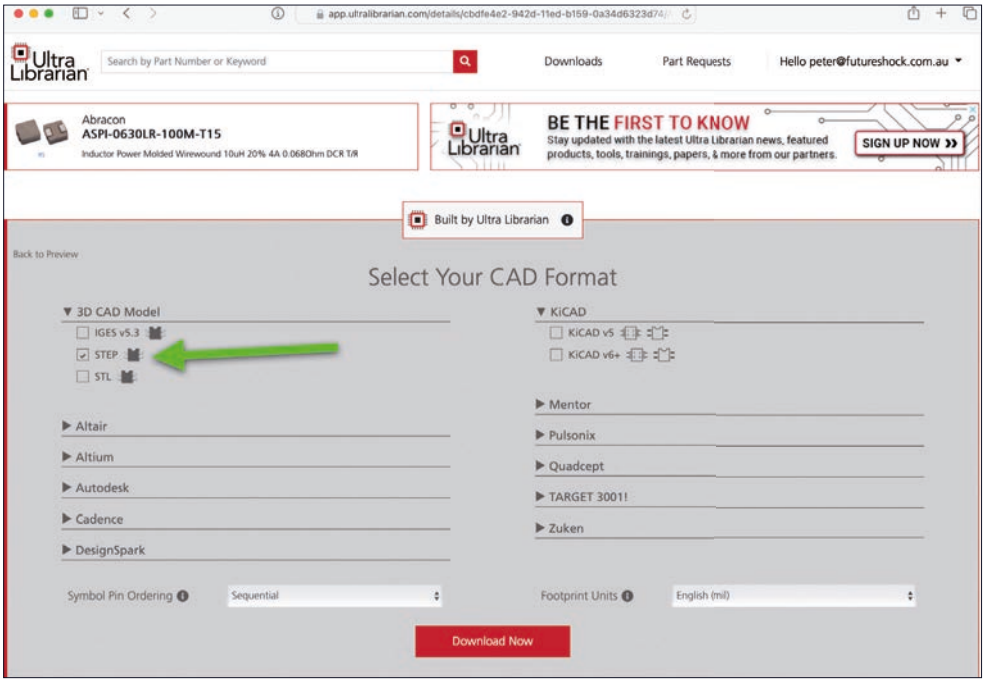


Figure 1.3.8.1: Getting the missing 3D STEP files.

I opt to copy the STEP files in the project library folder, as shown below. I mark the 3D STEP files with the green arrow:

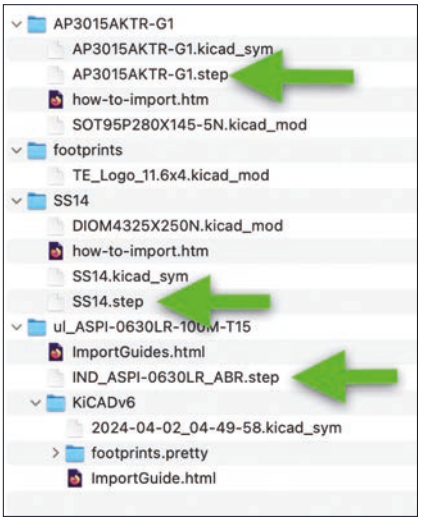


Figure 1.3.8.2: The STEP files are in the library folder of my project.

For L1, double-click on the footprint to open its properties window (1), then click on the folder button in the 3D Models tab (2) to open the file browser. Locate the STEP file for the 3D model and import it. You will see the model in the 3D viewer. This model fits well with the footprint, so we don't have to use the scale, rotation and offset widgets to adjust its position and size.

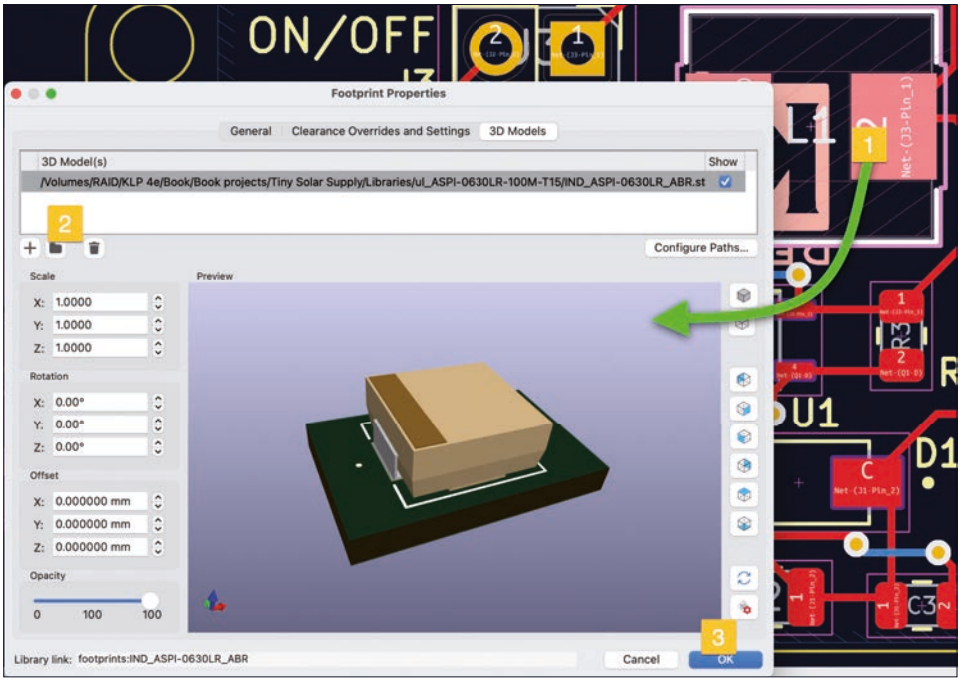


Figure 1.3.8.3: Setting the 3D model for L1.

For U1, the 3D model properties look like this:

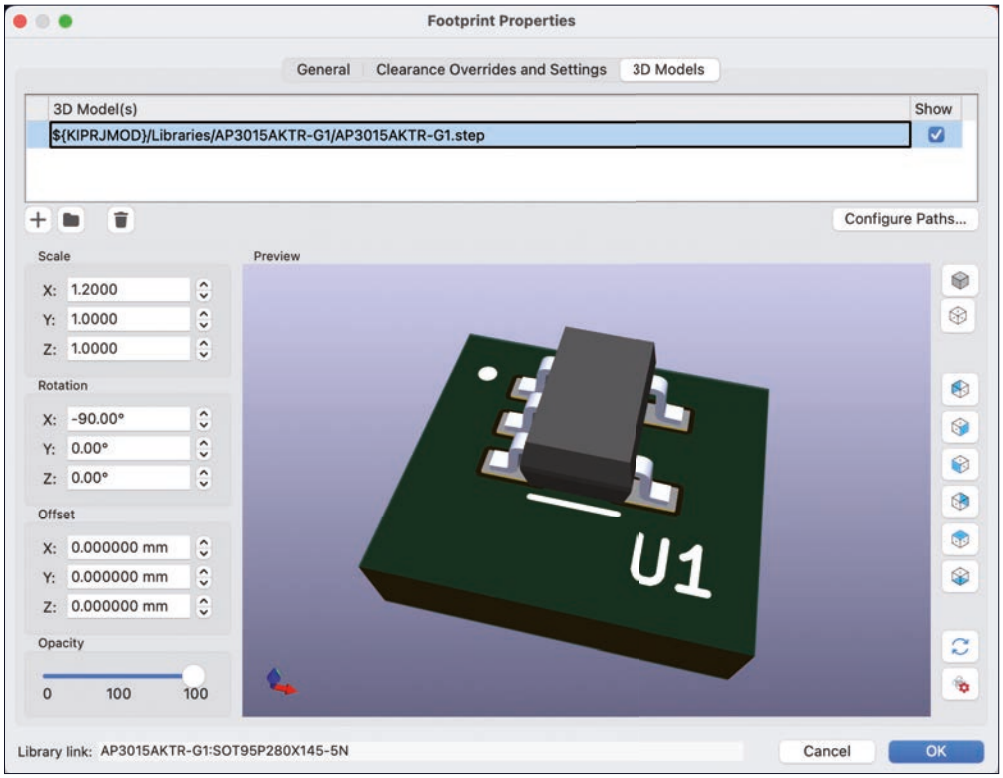


Figure 1.3.8.4: Setting the 3D model for U1.

And for D1, the 3D model properties look like this (they are identical for D2):

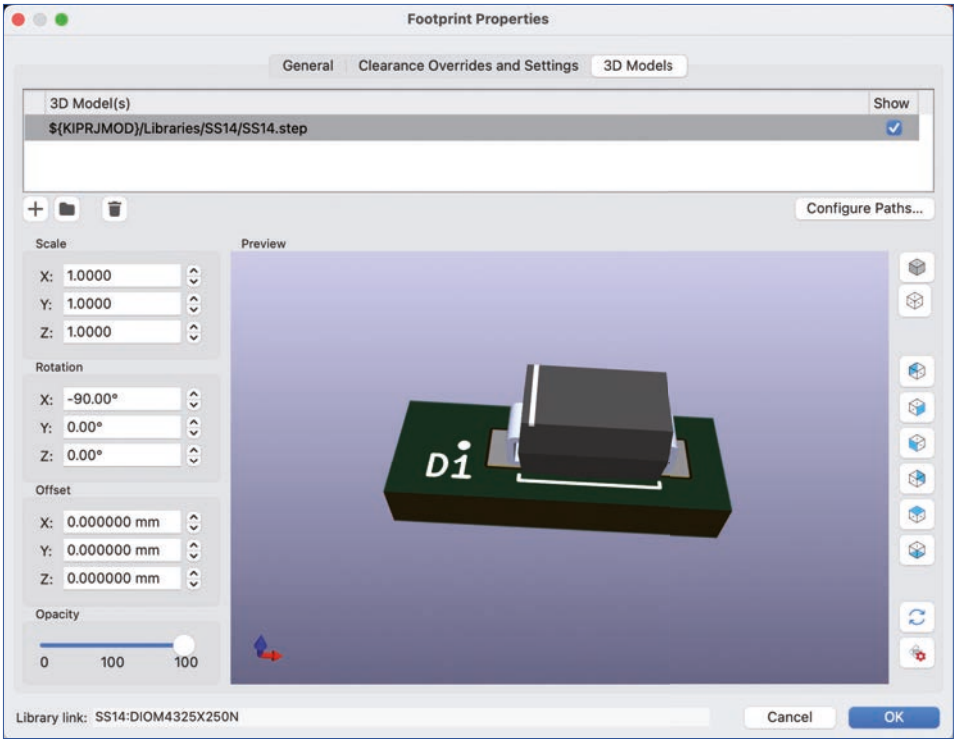


Figure 1.3.8.5: Setting the 3D model for D1 (and D2).

Last but not least, the four mounting holes. For this, I will draw four circles along the board's four corners. I will draw the circles on the Edge.Cuts layer.

Here is the final version of the board:

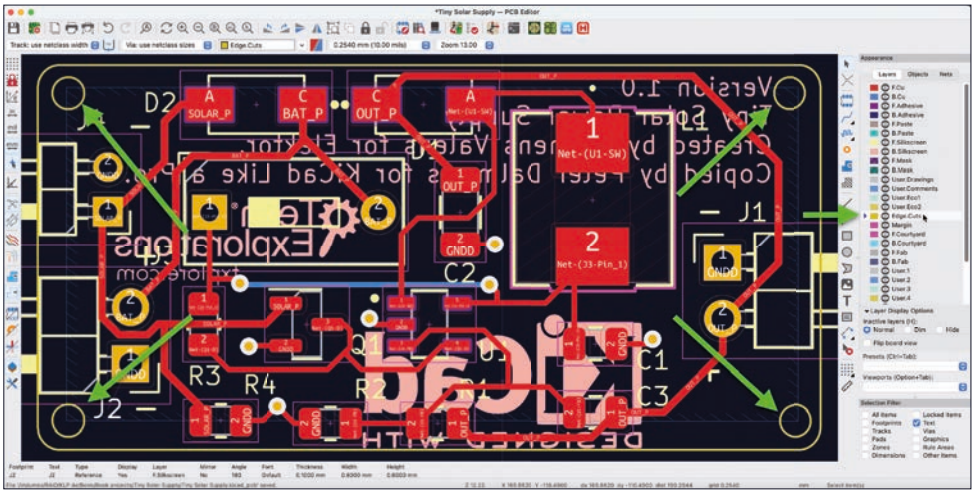


Figure 1.3.8.6: Added the mounting holes.

And here is its 3D representation (front and back):

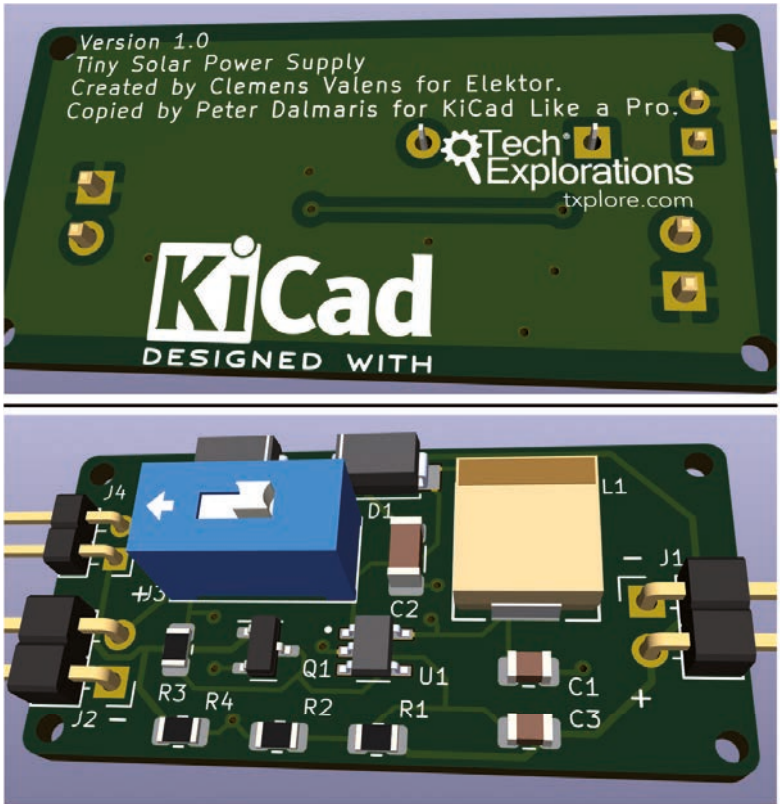


Figure 1.3.8.7: The 3D rendering of the final board.

Let's continue with export for manufacturing.

1.3.9. 8 - Export and Manufacture

To manufacture this board, I will use NextPCB and the generic process that involves exporting the Gerber files, checking them in a Gerber viewer (I will use NextPCB's viewer for this task), and then proceeding with the order. NextPCB has a plugin that allows you to order a PCB from inside the KiCad environment. I explain how to do this in a dedicated chapter.

Go ahead and export the Gerber plot and drill files for this project. The following screenshots show my settings.

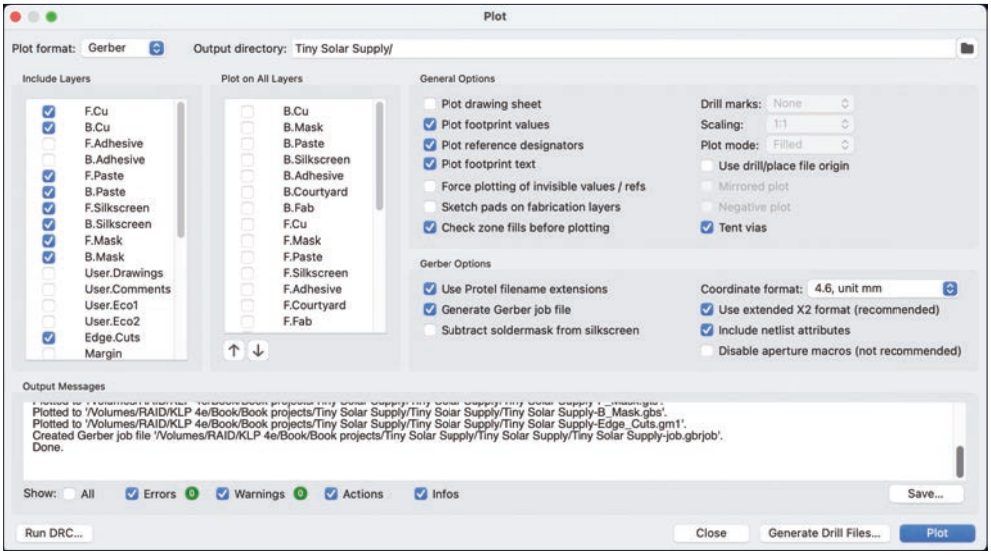


Figure 1.3.9.1: Generate the plot files.

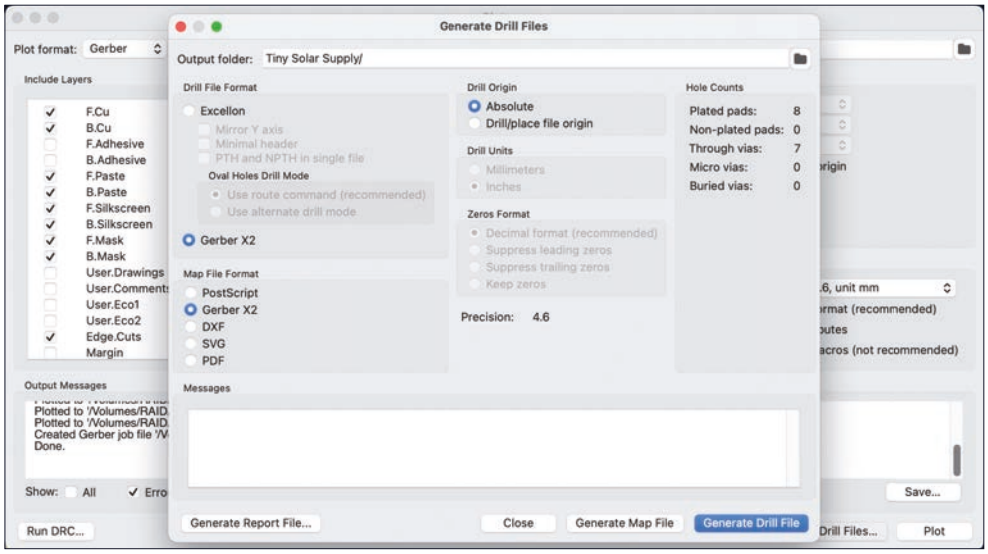


Figure 1.3.9.2: Generate the dril files.

Next, locate the folder that contains the Gerber files (I saved mine in the project directory), and create a ZIP archive of the entire folder. You can see the project directory, including the Gerber files directory and the ZIP file below:

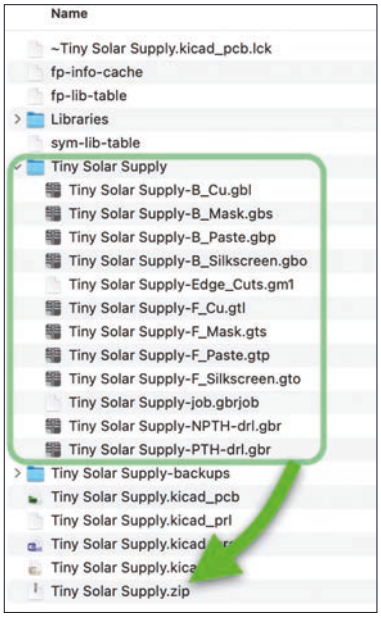


Figure 1.3.9.3: The Gerber files folder and ZIP archive.

Continue by logging in to your NextPCB account. From the top navigation menu, click on "Gerber Viewer". This will open the tool in a new browser. Drag your ZIP file onto the file uploader box (see below).

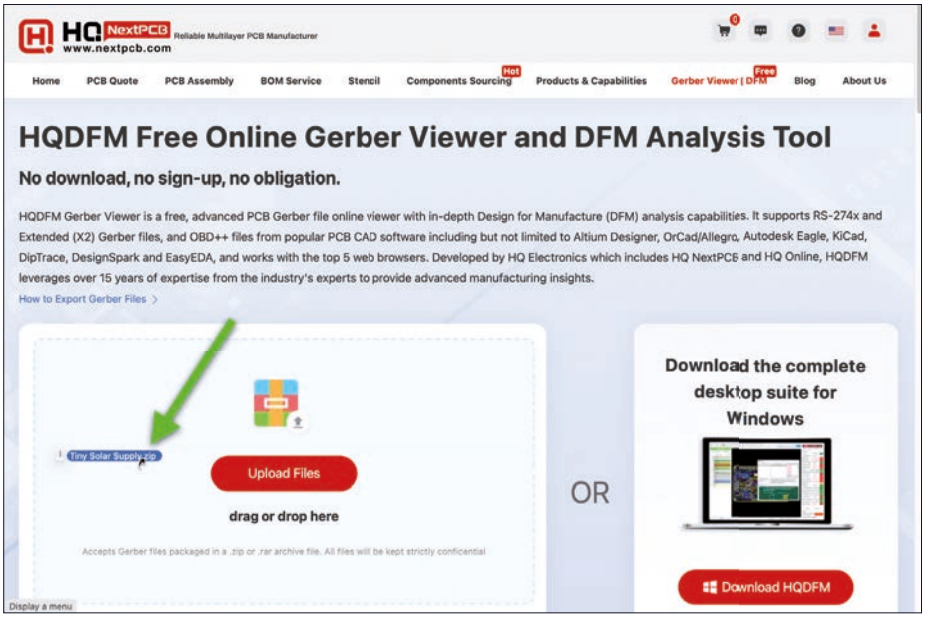


Figure 1.3.9.4: Uploading my Gerber files ZIP archive to the NextPCB online Gerber viewer.

A few moments later, the Gerber Viewer will display the board layers in the middle of the page. On the left of the page, you can use the layer chooser to enable and disable layers as you inspect them. You can see NextPCB's Gerber Viewer below.

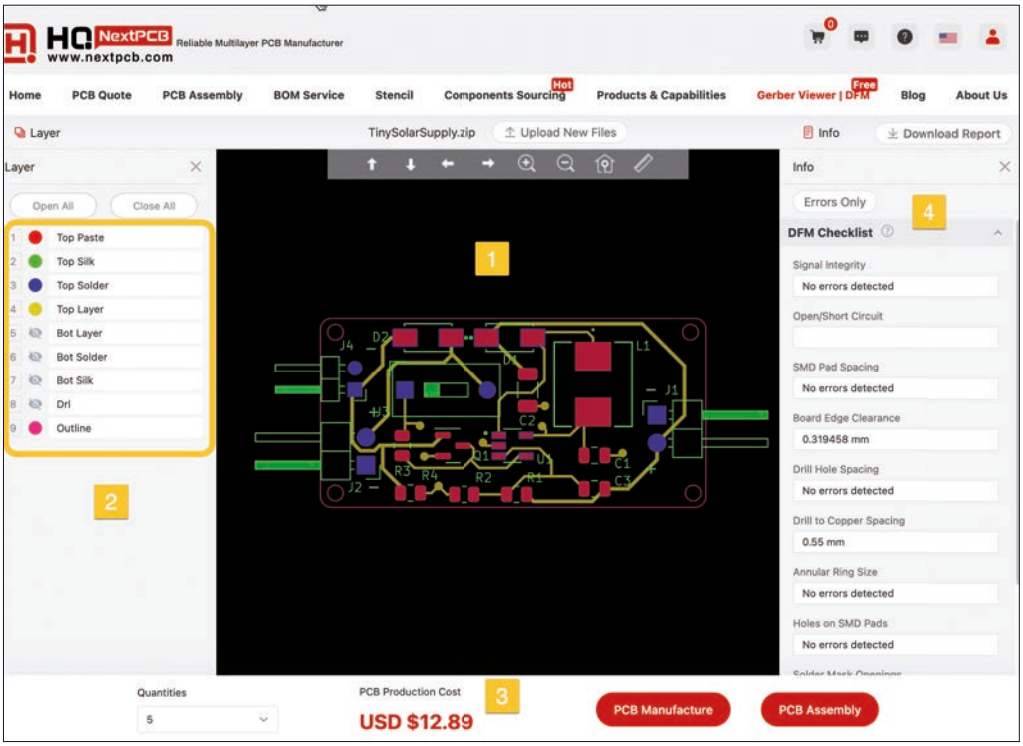


Figure 1.3.9.5: NextPCB's online Gerber Viewer displaying my new board.

The screenshot above shows the board in the middle (1), showing some of the layers in the ZIP archive. I can use the layer chooser on the left (2) to enable or disable layers so that I can review each one uncluttered. You can also see the cost of manufacturing for this board. I'll come back to this in a moment.

On the right side, you can see the results of the manufacturing analysis ("DFM" or "Design for Manufacturing") of this board's Gerber files. The DFM tool shows no errors or warnings for items such as problems with the copper fills, holes, traces, etc. You can also download a detailed PDF report (see the "Download Report" button above the DFM checklist). You can see the first page of this report below:

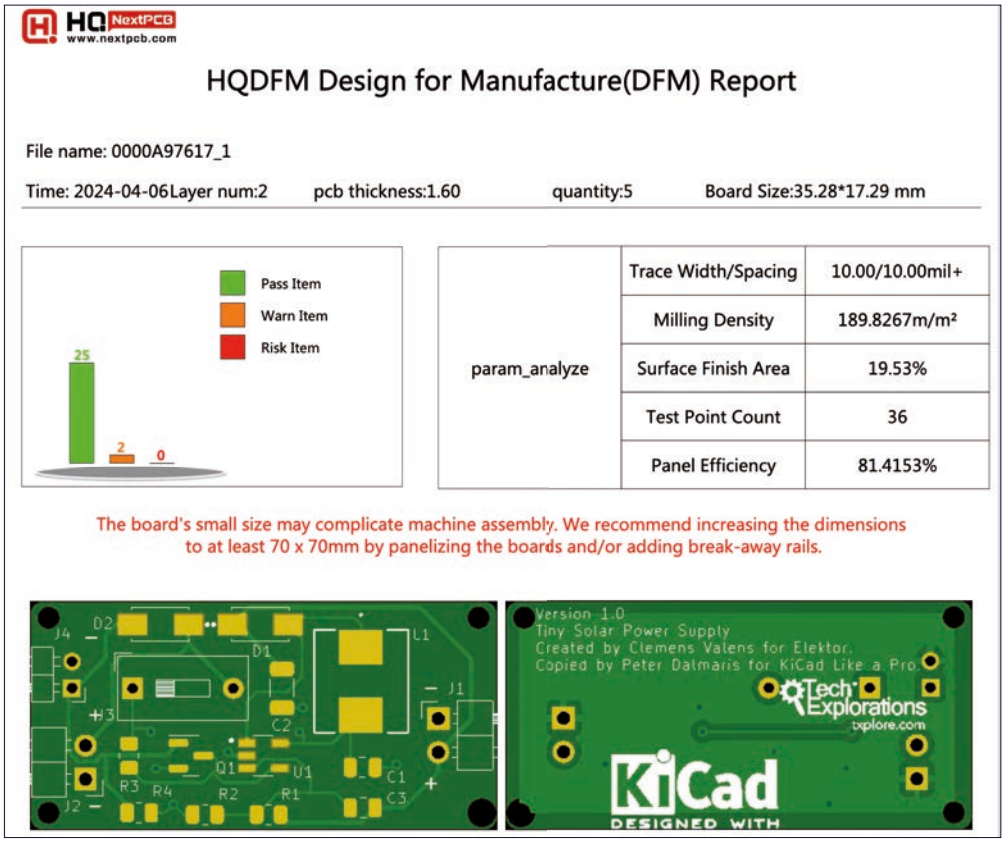


Figure 1.3.9.6: NextPCB's DFM tool provides this detailed report for my PCB.

The report warns me that the board is small and "may complicate machine assembly". This is not a problem because I only plan to manufacture this board, not assemble it. I will hand-solder the components.

After reviewing the detailed report, I conclude that my board is ready to manufacture. From the Gerber Viewer page, click the "PCB Manufacture" button. This will bring up the PCB order form and populate it with the board (its dimensions and Gerber files). You only need to select the various configuration options. If you are comfortable with the defaults, the only remaining consideration is the shipping options. I usually accept the default configuration to keep the price low unless I have specific requirements. You can see some of my selections below:

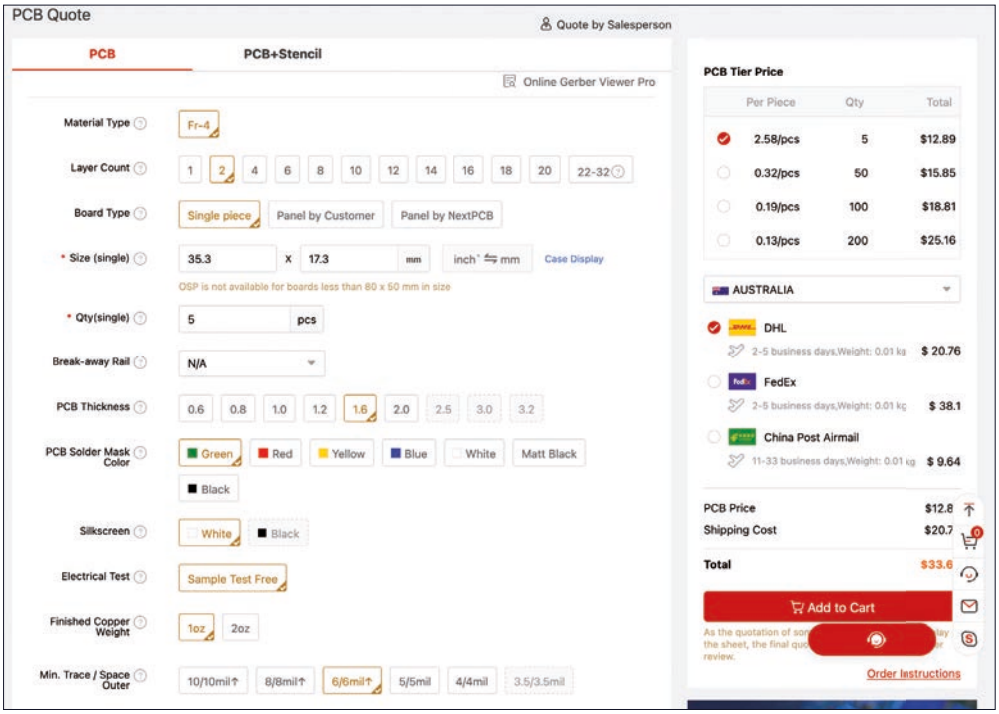


Figure 1.3.9.7: The NextPCB order page.

After adding the order to the cart and placing it, your PCB will go into NextPCB's queue for a quick manufacturing review. This review aims to look for glaring issues that may prevent manufacturing. Since NextPCB's DFM tool has already reviewed the board and did not reveal any manufacturing issues, it is very unlikely that the manual review will reveal any issues. In any case, you must wait until the review is complete to have your order cleared. After that, you will receive an email informing you that payment is pending. A couple of weeks later, your new boards will arrive.

2: Project - A 4 x 8 x 8 LED matrix array

2.1. Introduction

In the following chapters, you will design a PCB that can hold four 8x8 LED matrix displays controlled by an Arduino Pro mini. The board also includes two push buttons to assign arbitrary functions. I plan to use mine as a Pomodoro timer. I will use one button to select the lap duration (say, 15, 20, or 25 minutes) and another to reset the timer. When the duration I have set expires, the display will blink to let me know.

You can see the two sides of the populated PCB in the photograph below:

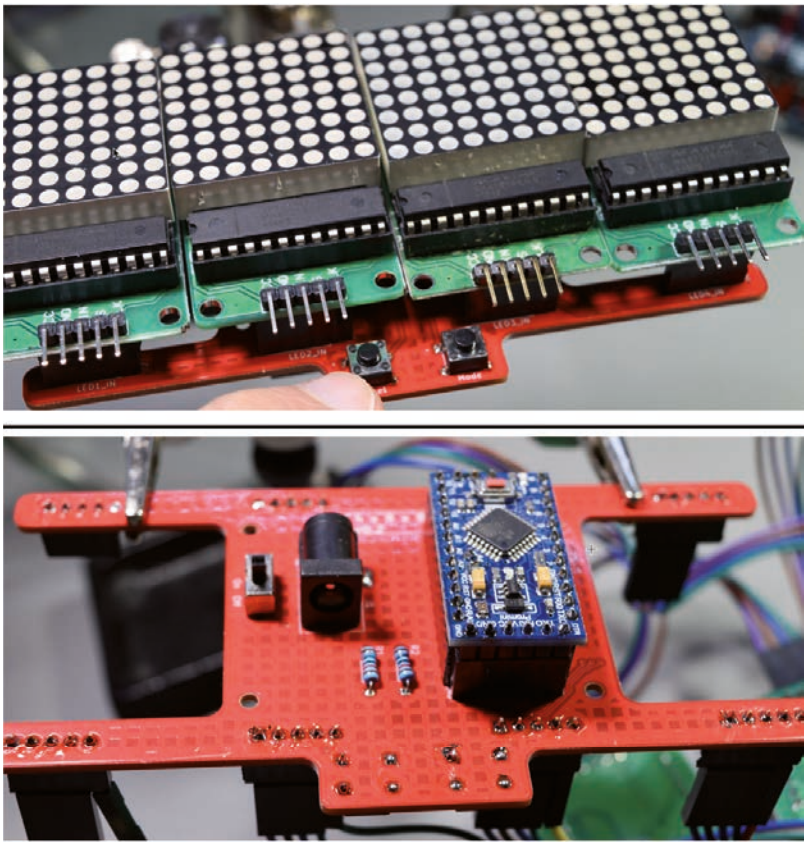


Figure 2.1.1: The PCB in this project, populated.

I often forget to get up from my desk, which inspired this project. I could use a desktop or phone Pomodoro app or even a classic mechanical Pomodoro timer, but why buy one when I can make one?

I decided to use an Arduino Pro Mini with the Atmega328P MCU and an on-board 5V voltage regulator because:

- I have a lot of them.
- They are easy to find in the market
- They are very cheap.
- They are small.
- They have an onboard regulator.
- They are accurate enough to count short periods.
- I don't need a clock function, so the absence of a real-time clock is not an issue.

This board does not contain a UART to USB interface, so you must provide an external bridging device. This device is required to program the microcontroller. I use a USD to serial adaptor from Freetronics, but there are many other options in the market.

I chose the 8x8 LED matrix display module with the MAX7819 controller chip because I like its versatility. An 8x8 LED matrix display can show numbers, text, and simple graphics. I also like how it looks from a distance and can create a simple animation. All this gives me scope to add features to my Pomodoro project in the future.

Below, you can see the schematic for this PCB. You will create this schematic by the end of Chapter 2.2.

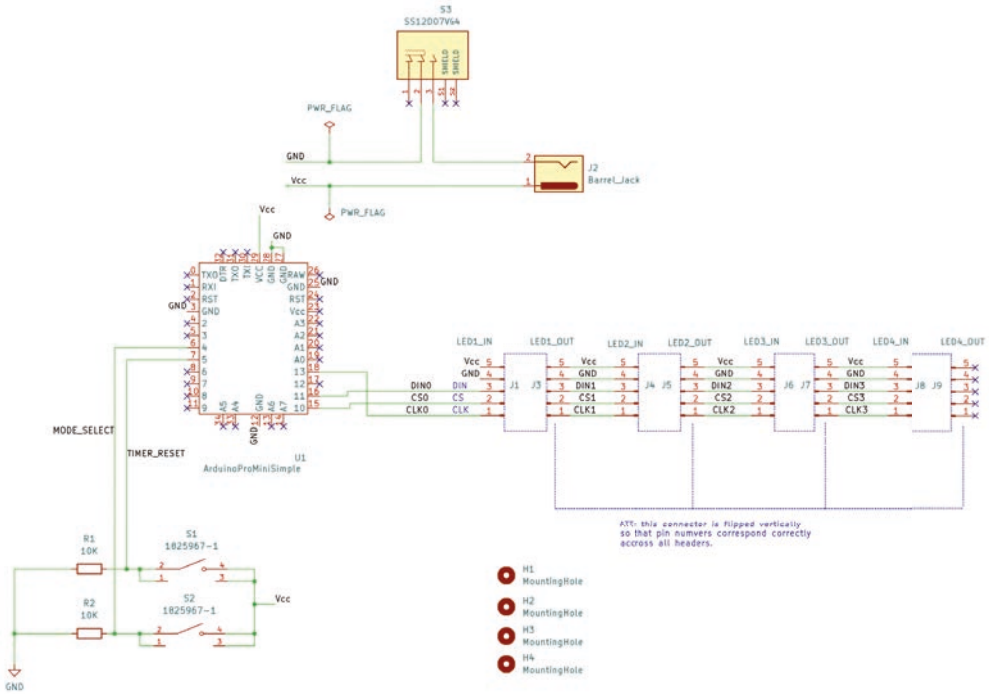


Figure 2.1.2: The project schematic.

I have used line wires for all nearby pins to draw this schematic. I added net labels to all power, data, clock, and button signal nets. When I designed the first version of this project, I could not find a schematic symbol for the Arduino Pro Mini board that I was planning to use, so I created one with a matching footprint. Some new libraries on the KiCad repository include this symbol and footprints (such as the "KiCad Library for Arduino Modules"). Still, I will not use that, so I can use the opportunity to create a custom symbol and footprint.

I have included four mounting holes in the layout. To avoid getting error messages from the DRC, I have associated the mounting hole footprints with mounting hole symbols in the schematic.

Another consideration was how to deal with the LED matrix modules. I was not able to find a symbol and footprint for this device, so I had two options:

1. Create a custom symbol and footprint, as I did with the Arduino Pro Mini module.
2. Ignore the module and concentrate on the headers.

Since I went with option 1 for the Arduino module, I opted for option 2 for the LED modules. Instead of treating each LED module as a single device, I treated it as a set of two-pin headers. My objective, then, was to wire the header symbols correctly and place their footprints precisely on the PCB (see the discussion on the PCB below).

Below is the layout of the PCB, as it will be at the end of Chapter 2.3:

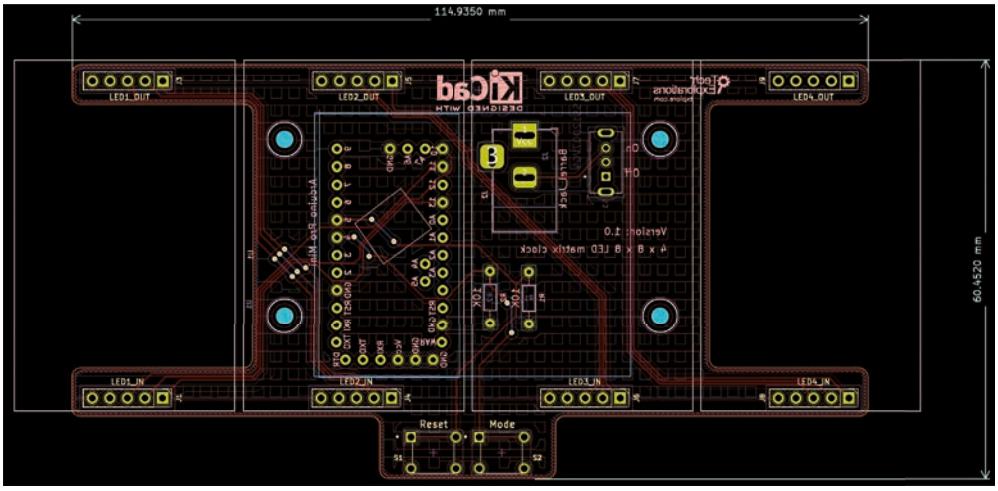


Figure 2.1.3: The project layout.

The dominating feature of this board is its shape. I wanted to experiment with a shape that uses "arms" that extend from its center to hold the LED modules rather than a conventional rectangular shape. I did not do this to reduce the manufacturing cost. Even though the shape you see above has two significant parts of the substrate material removed, the

manufacturing cost relates to the all-inclusive height and width of the board (you can see those dimensions in the figure above). But I think the board with the four arms extending from the center looks great. Along with the rounded edges and the button notch at the bottom, I am satisfied with the physical design aspect of the board.

A significant challenge for the layout design of this board is the position of the pin headers for the LED modules. As mentioned above, I treated the LED modules as two headers each (input and output). The positions of those headers must be very accurate. If the headers are too far from their neighbors, the four-module display will not look continuous but like four individual displays. If they are too close, the assembly will not be possible as there will not be enough space on the board to attach adjacent modules.

This project will allow you to use all of KiCad's measurement and alignment tools to ensure that the end product looks beautiful and works.

Below, you can see the Bill of Materials for this project, as I have extracted it from the KiCad project (learn how later in this book):

Reference	Value	Footprint
H1-H4	MountingHole	MountingHole:MountingHole_2.5mm
J1	LED1_IN	Connector_PinHeader_2.54mm:PinHeader_1x05_P2.54mm_Vertical
J2	Barrel_Jack	Connector_BarrelJack:BarrelJack_Horizontal
J3	LED1_OUT	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
J4	LED2_IN	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
J5	LED2_OUT	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
J6	LED3_IN	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
J7	LED3_OUT	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
J8	LED4_IN	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
J9	LED4_OUT	Connector_PinSocket_2.54mm:PinSocket_1x05_P2.54mm_Vertical
R1, R2	10K	Resistor_THT:R_Axial_DIN0204_L3.6mm_D1.6mm_P7.62mm_Horizontal
S1, S2	1825967-1	1825967-1:SW_1825967-1
S3	SS12D07VG4	SS12D07VG4:SW_SS12D07VG4
U1	ArduinoProMiniSimple	DesktopLibrary:ArduinoProMiniCustom

Table 2.1.1: The Bill of Materials for this project.

Apart from the custom symbol and footprint Arduino Pro Mini, I used Snapeda to find the symbol-footprint pairs for the two buttons (S1 and S2) and the power barrel connector (S3).

Please note that if you want to build this gadget, you must modify the LED module headers. I have written a chapter describing the process of modifying these headers and assembling the gadget. You can find this chapter at the end of this part of the book.

If you plan to assemble this gadget, please read the following carefully. I have designed this PCB to route power from the barrel connector to the Arduino Pro Mini's Vcc pin. As per the specifications of this Arduino, the Vcc pin requires regulated 5V power (I use the 5V version; a 3.3V version is also available). The Vcc pin bypasses the onboard voltage regulator and routes the 5V input directly to the microcontroller. As a result, you must be careful to connect your gadget to a trusted and regulated 5V power supply. I use a mains power supply that provides stable 5V power at 500mA (see photograph below).

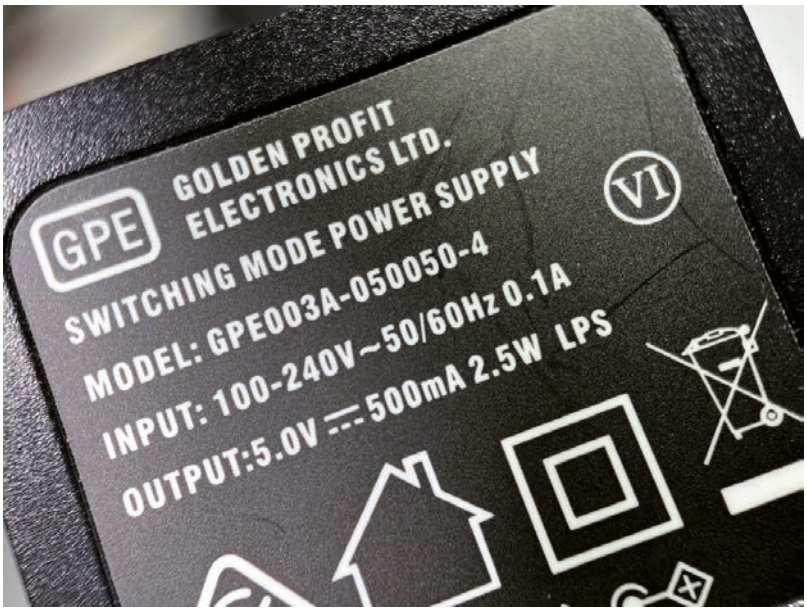


Figure 2.1.4: The power supply I use in this project.

This power supply provides sufficient current for the Arduino and the four LED displays. If you don't have a mains power supply, you can use a USB-to-barrel connector cable, similar to the one in the photograph below.

Figure 2.1.5: A USB to barrel jack connector cable.

You can power this gadget from your computer or a regular USB power supply with the USB to barrel jack connector cable. Either option (the regulated 5V mains power supply or the USB to barrel jack connector cable) allows you to use this gadget without any modifications.

An alternative design choice is to re-wire the positive pin of the (pin 1 of J2 in Figure 2.1.2) to pin 26 (RAW) of U1. According to the specifications, the RAW pin can receive unregulated power between 5V and 12V for the 5V model or 3.35V and 12V (for the 3.3V model). This wiring option uses the onboard voltage regulator, giving you a wider range of safe-to-use power supplies. Consider which option you want to adopt, and make the necessary modifications to the circuit in the wiring step of the schematic workflow.

Let's begin with the schematic design in the next chapter.

2.2. Schematic design

In this chapter, you will complete the schematic design of this PCB by following the schematic design workflow, as you learned in My book Fundamentals and Projects, Part 3 of this book.

2.2.1. 1 - Setup

Let's begin with the schematic design and set up the project. Start KiCad, and click File —> New Project from the main project window. Select a location for the new project and give it a name. I called mine "4x8x8 LED Matrix Clock".

KiCad's main project window looks like this:

Figure 2.2.1.1: The new project files.

Let's continue with the setup of the schematic editor. Start the schematic editor, and inspect your default settings in the Schematic Setup window. I have only made two changes:

1. In Net Classes, I have added the Power net class. I will assign nets to this class in step five of the workflow.
2. In Text Variables, I added a variable named "design_version." For the value, I have set "1.0". I will update this variable when I update the project and export it to the schematic and layout editors' information box.

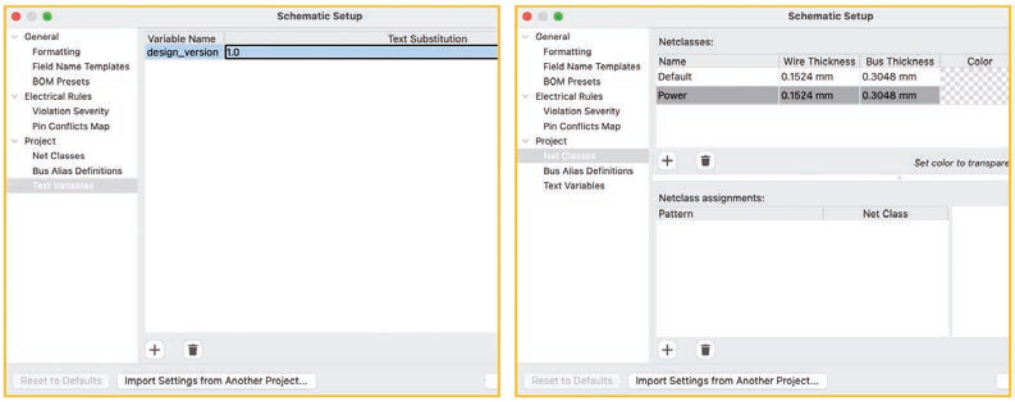


Figure 2.2.1.2: Changes to the Schematic Setup.

Next, inspect the settings in the Preferences window. I have made two changes to the default settings here:

1. Under Colors, I have changed the background to white.
2. Under Field Name Templates, I have added the field name "Purpose."

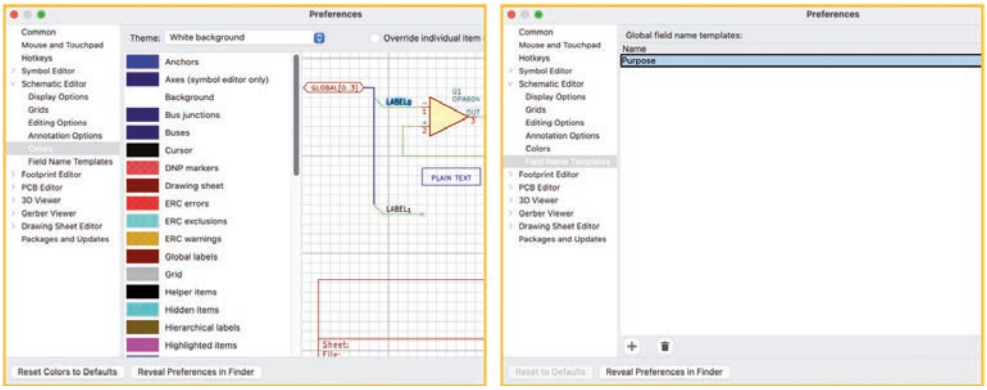


Figure 2.2.1.3: Changes to the Preferences.

Finally, open the Page Settings window and fill in the project information. Notice that in the Revision field, I am using the text variable notation "${text_variable}" to import the text variable with the name "design_version." You can use text variables anywhere there is text.

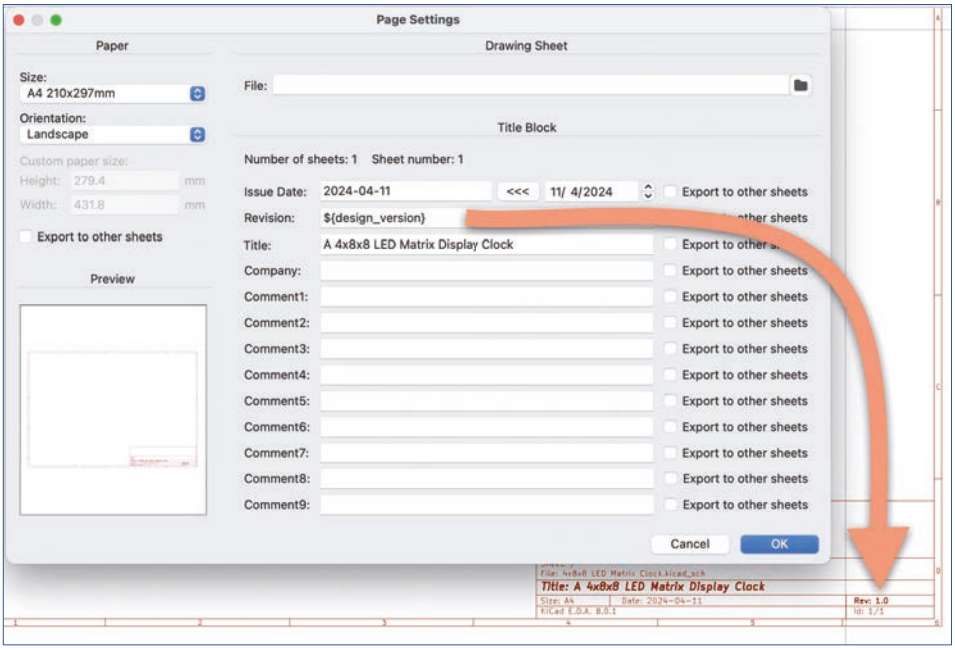


Figure 2.2.1.4: The Page Settings window, using a text variable.

The project setup is complete. Let's continue with the symbols in the next step of the workflow.

2.2.2. 2 - Symbols

In this segment, you will complete step two of the schematic design workflow. In other words, you will add the symbols required for your schematic to the schematic editor.

Most of the symbols needed for this project are available in KiCad's symbol libraries. I will also use symbols from the Digikey library and a couple I downloaded from Snapeda. I have also created one. You can see the relevant part of my Symbol Libraries window below.

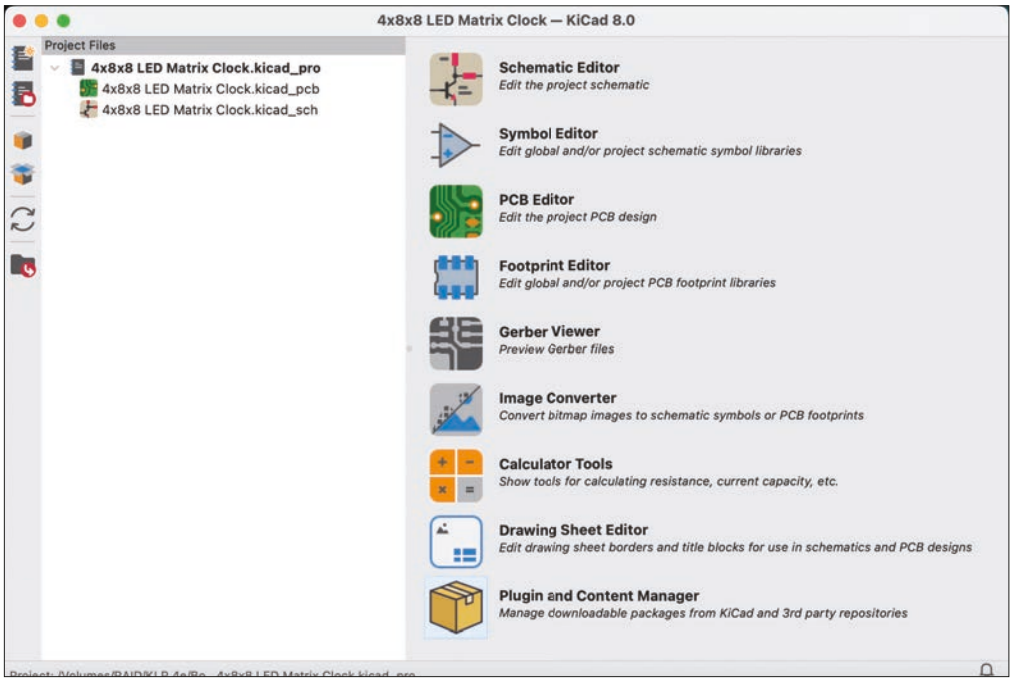


Figure 2.2.2.1: My project symbol libraries.

Here are more details about the symbols I am using in this project:

1. ArduinoProMiniSimple, custom made by Peter, "1" in the figure above.
2. Slide switch, symbol "SS12D07VG4" from Snapeda, "2" in the figure above. (download here[1]).
3. The tactile button "Switch Tactile OFF (ON) SPST Round Button" from Snapeda is "3" in the figure above (download here[2]).
4. Everything else is available in KiCad's libraries

1 https://www.snapeda.com/parts/SS12D07VG4/Shouhan/view-part/5844111/?ref=search&t=SS12D07VG4

2 https://www.snapeda.com/parts/1825967-1/TE%20Connectivity/view-part/398388/?ref=search&t=Switch%20Tactile%20OFF%20(ON)%20SPST%20Round%20Button

You can find all the symbols and footprints I'm using in this project in the project's GitHub repository[3] (see under "Libraries." In the same place, you will also find the complete KiCad project files.

For a refresher on how to find symbols on the Internet and install them on KiCad, please read the chapters "How to find schematic symbols on the Internet" and "How to install symbol libraries in bulk" in My book Fundamentals and Projects. If you want to learn how to create the Arduino Pro Mini custom symbol, the chapter "How to create a custom symbol" is also in My book Fundamentals and Projects.

Once you have installed the required symbols in the KiCad symbol library, add them to the sheet. Below, I am using the Symbol Chooser to add the symbol for the Arduino Pro Mini:

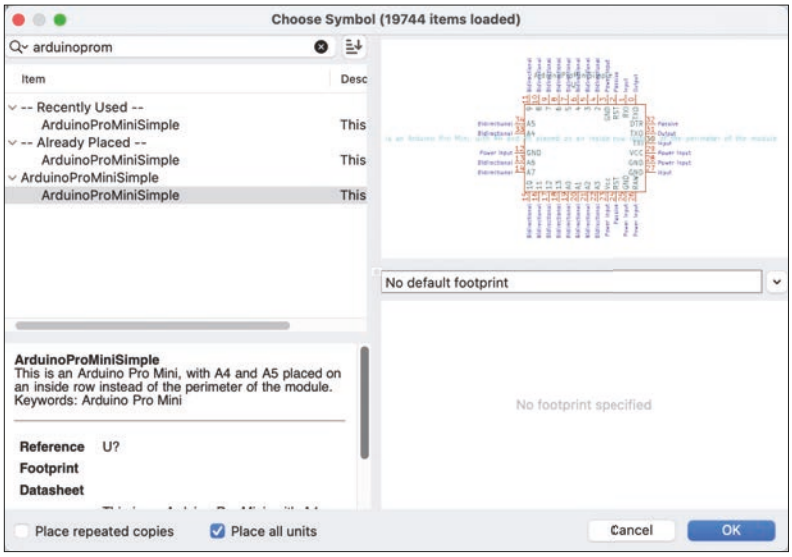


Figure 2.2.2.2: Adding the symbol for the Arduino Pro Mini.

Continue with the rest of the symbols until all symbols I have listed in Table 2.2.1 are in the schematic sheet. By the end of this process, your schematic sheet should look like this:

3 https://github.com/futureshocked/4x8x8-LED-Matrix-Display-project-files

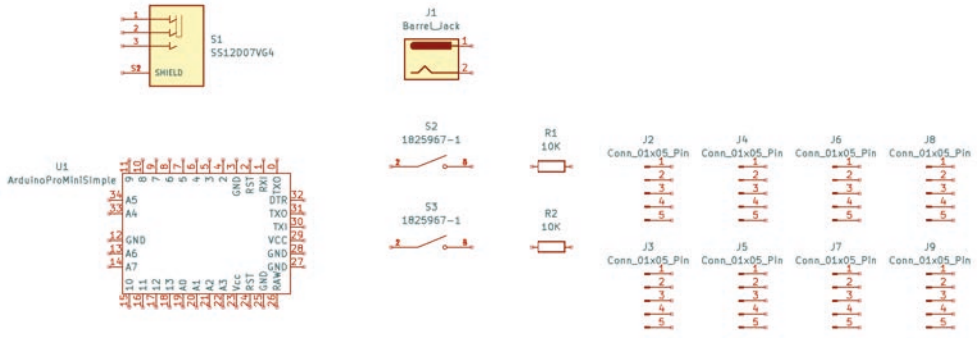


Figure 2.2.2.3: All symbols added to the schematic sheet.

In the figure above, I have added the values for the two resistors, "10K" for each. With the schematic symbols added to the sheet, you can now proceed with step two of the workflow: arranging and annotating the symbols.

2.2.3. 3 - Arrange, Annotate

In this segment, you will arrange the symbols on the sheet to prepare them for wiring in step four. All symbols are already annotated.

The circuit is simple, so the arrangement of the symbols on the sheet is straightforward.

I'll place the two symbols that comprise the power input group together at the top of the schematic. I'll place the Arduino module towards the left of the schematic to allow enough space for the connectors on the right side. Finally, I'll place the two switches and their resistors in a group below the Arduino symbol.

You can see the final placement of the symbols below:

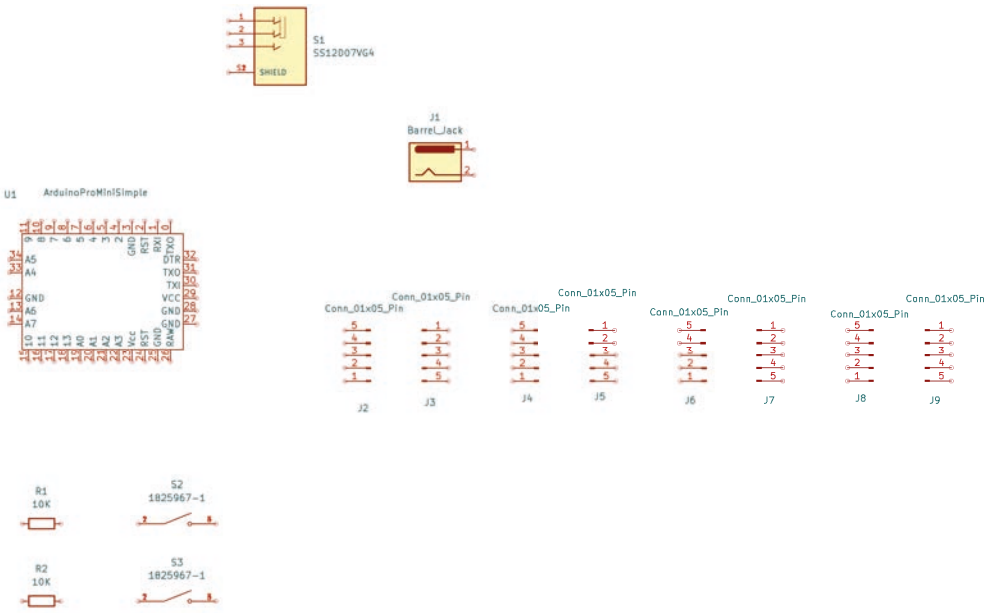


Figure 2.2.3.1: Symbols placed in the schematic sheet.

I spent most of my arrangement time with the connectors. The 8x8 LED display modules use the SPI interface to communicate with the Arduino and other modules in series. They have one five-pin connector to receive data from the Arduino or another module in the series and a second five-pin connector to provide data, clock, and power to the next module.

I use the "Conn_01x05_Male" symbol to represent these connectors. Remember that wires connect to the end of the pin with the small circle. This is important when it comes to orienting the connectors appropriately. For example, the connector that represents the input header of the first display module must be oriented so that the pin circles point toward the Arduino symbol. The second five-pin connector is on the other side of the first display module. This connector's pins must have circular ends pointing toward the second display module.

Because of these orientations, the pin numbers will not match. Instead, you will see that pin 5 of header 1 is opposite pin 1 of header 2, as you can see below:

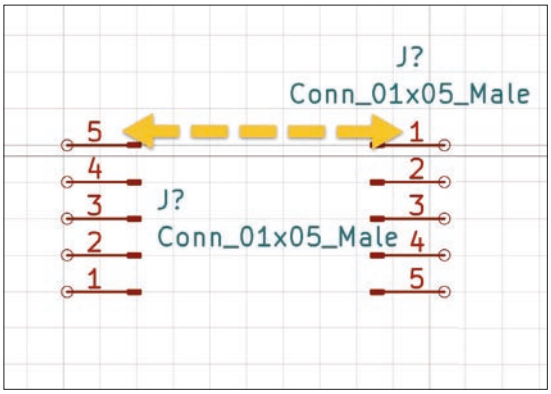


Figure 2.2.3.2: Pins are mismatched for neighboring header symbols.

To make the pin numbers match and properly represent how the pins are arranged in the LED display module, you must flip the data output header vertically. This will maintain the horizontal orientation of the module and only change the pin numbers. To flip a symbol vertically, right-click on the symbol to reveal its context menu, and select "Mirror Vertically":

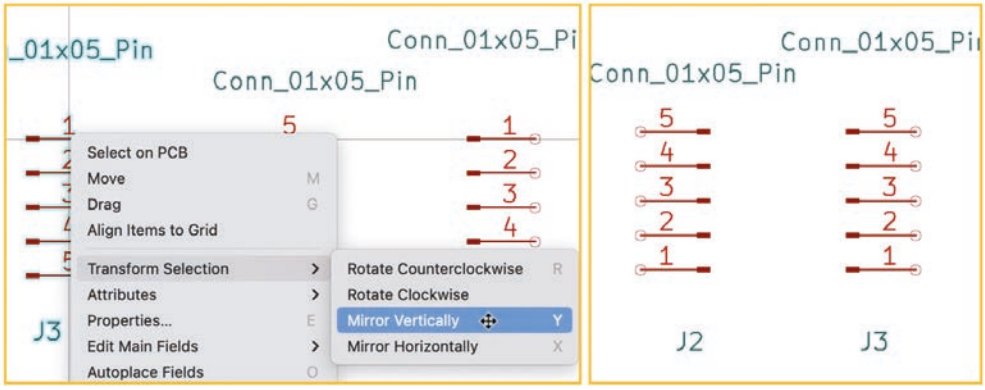


Figure 2.2.3.3: Mirror symbol vertically.

Arrange the connectors and place them appropriately so that the schematic looks like the one in figure 2.2.3.3.

Finally, run the automated annotation tool. The result in my project instance is shown in the figure below.

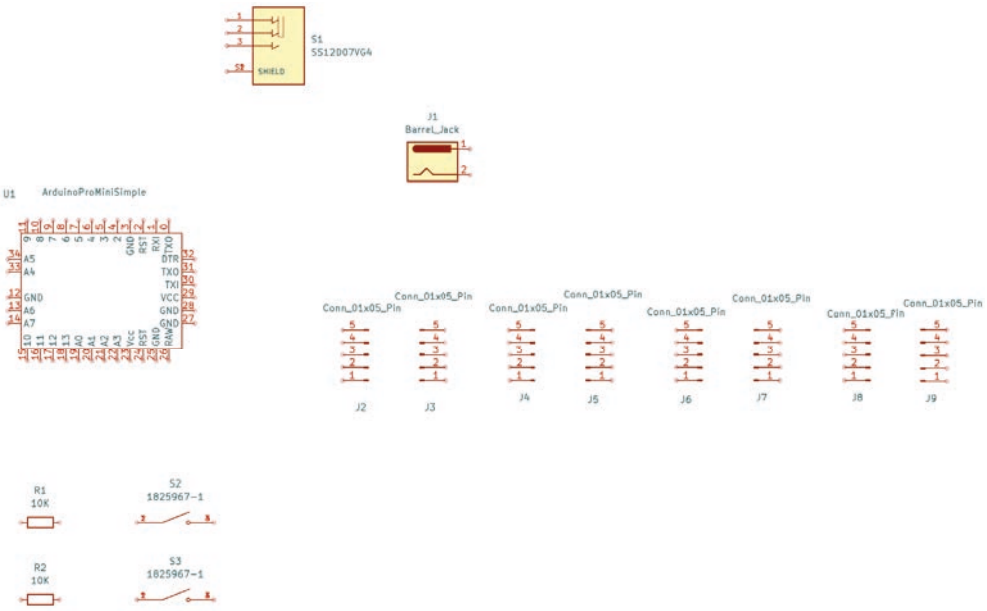


Figure 2.2.3.4: The annotated schematic symbols.

The designators in your instance should match those in the figure above. If there are any discrepancies, you can either leave them as they are and remember them for the remainder of this project or manually change the designators to match what you see above.

When ready, continue with the symbol-footprint associations in the next segment.

2.2.4. 3 - Associate

Let's complete step three of the schematic workflow, where you will set the associations between schematic symbols and their footprint counterparts. In Table 2.2.1, you can see the project symbols in column two and their associated footprints in column three.

The screenshot below shows the footprint libraries I set up in my KiCad's Footprint Libraries table. I have placed the libraries in the "Libraries" folder, which is inside the project directory.

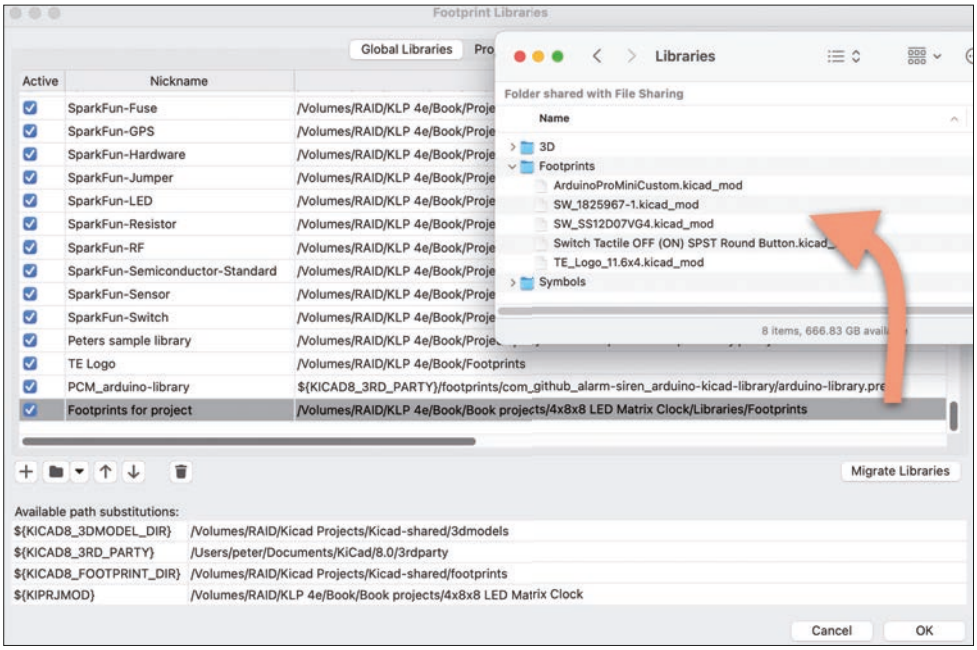


Figure 2.2.4.1: The footprint libraries I will use in this project.

You can find all the symbols and footprints I use in this project in the project's GitHub repository[4] (see under "Libraries). The complete KiCad project files are also in the same place.

Once you install the necessary footprints, return to the schematic editor and open the Associations window. Use the information in Table 2.2.1 to help you find the libraries and footprints in the left and right panes of the window. If KiCad asks you if you want to convert legacy footprints to the new format, select "Yes".

By the end of the process, your association's table will look like this:

4 https://github.com/futureshocked/4x8x8-LED-Matrix-Display-project-files

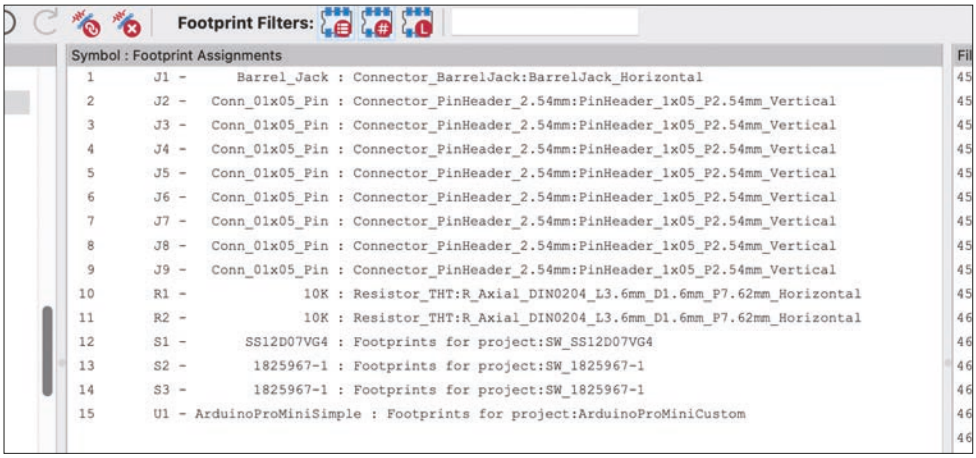


Figure 2.2.4.2: The symbol-footprint association's table.

This completes step three of the schematic design workflow. Let's continue with step four, wiring.

2.2.5. 4 - Wiring

I will work on step four of the schematic design workflow in this segment. I will connect pins by:

1. Drawing wires using the Wire tool.
2. Partially complete step five of the workflow by creating some of the named nets. You will create the remaining named nets in step five of the workflow.

At this point, the schematic looks as I left it at the end of the last segment of this chapter (see Figure 2.2.5.1). I will begin wiring using the Wire tool from the right toolbar to draw lines connecting nearby pins. I will begin with the buttons in the bottom left corner of the schematic:

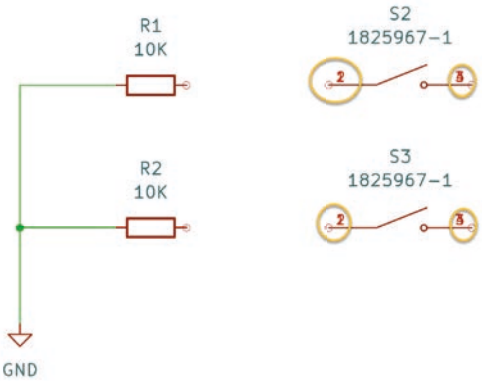


Figure 2.2.5.2: Using the wire tool to connect pins.

As in the figure above, connect the left pin of the two resistors to a GND symbol. Notice that the S2 and S3 symbols have overlapping pin numbers. I have marked those with a yellow circle. I have noticed this with other symbols, too, so this is an opportunity to understand what is happening here and how to solve this problem.

Let's examine S1 more closely. Double-click on it to bring up its Properties window. Then, click on Edit Symbol.

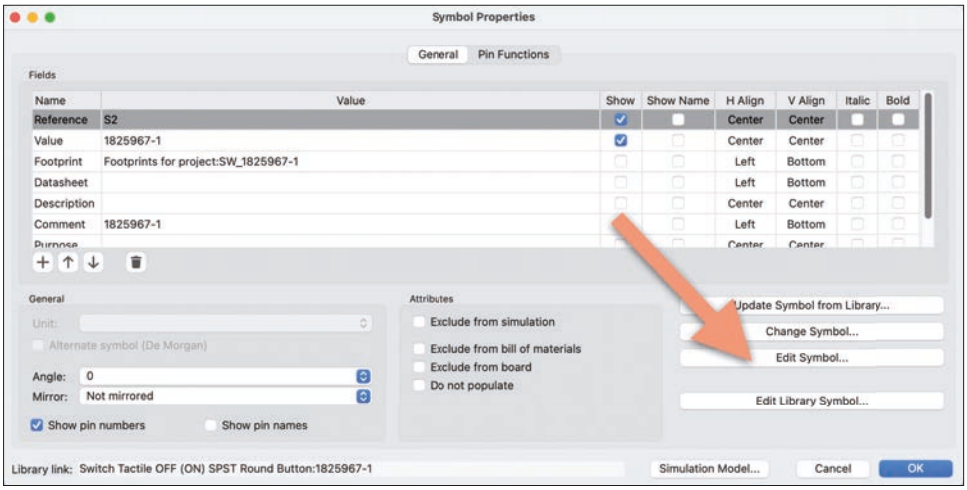


Figure 2.2.5.3: The properties window for S2.

The symbol editor will come up, displaying the S1 symbol. Notice that pins overlap on both sides of the symbol. In the figure below, on the left side is the original state of the symbol with the overlapping pins. On the right side, I have separated the pins by clicking and dragging them downwards.

Figure 2.2.5.4: Separate the overlapping pins by dragging down.

After you have separated the pins, save the symbol and close the symbol editor. You will see that S2 is updated in the schematic sheet, and all four of its pins are visible. Repeat the same process for S3, and complete the wiring so that the resistor and button network group look like this:

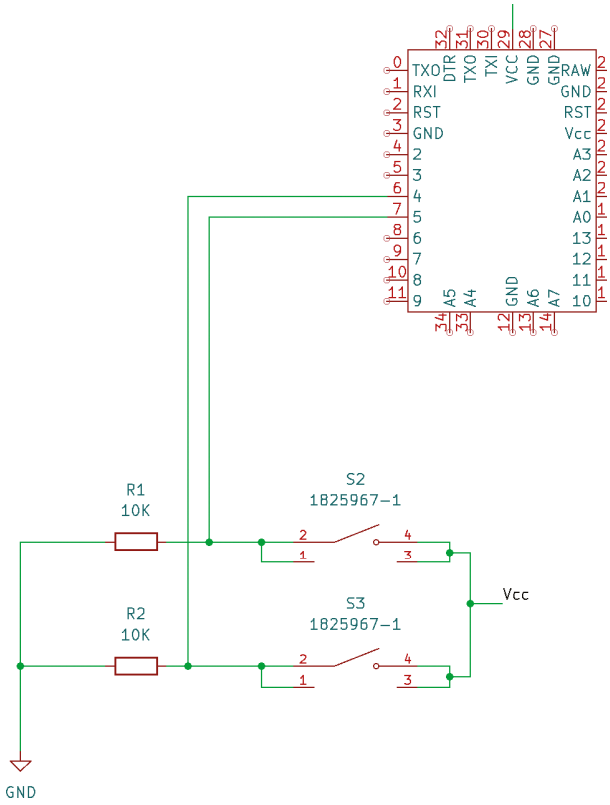


Figure 2.2.5.5: Resistors and buttons are wired.

In the figure above, I have used a net label to connect pins 4 and 3 of S1 and S2 with pin 29 of U1. The net label is "Vcc." I will create more such labels as needed.

Continue with the power input group:

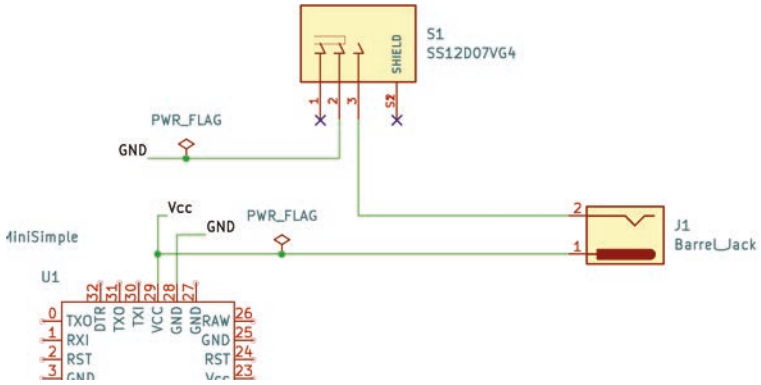


Figure 2.2.5.6: Power input group wired.

In the figure above, notice:

1. In S1, two shield pins overlap. Since those pins are unconnected, I chose to leave them as they are instead of separating them as I did with S2 and S3. As you will find out later when I do the ERC, this decision turned out incorrect. I will need to separate the two pins to pass the ERC. More about this is in the segment in step six of the workflow.
2. S1 pin 1 is unconnected, as are the two SHIELD pins.
3. Use the PWR_FLAG symbol to mark the GND and Vcc nets as power nets.

Continue with the connectors. Below is the final wiring for this group of symbols (not final; more work needs to be done there):

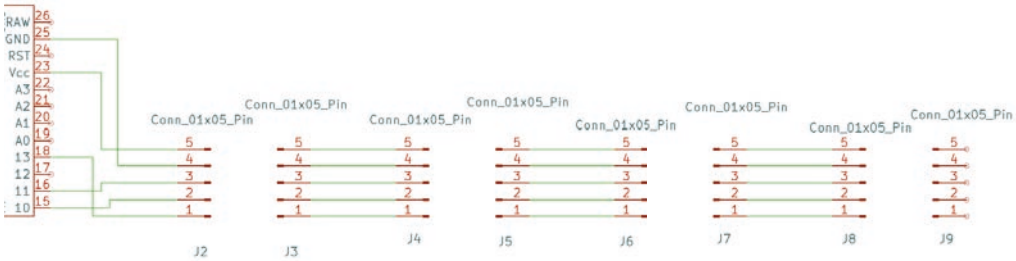


Figure 2.2.5.7: LED modules wiring (in progress).

The wiring is almost complete. Two tasks remain:

1. Mark any unconnected pins.
2. Add remaining net labels. You will work on this in the next segment (step five of the workflow).

Below is the fully wired schematic before finishing the remaining named nets:

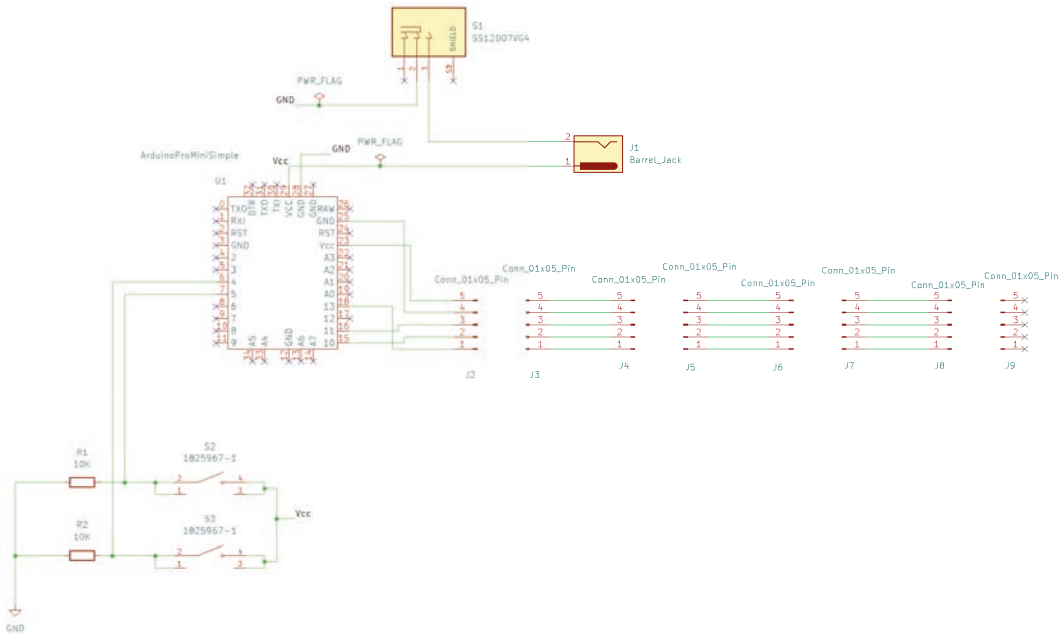


Figure 2.2.5.8: Fully wired schematic.

Let's continue to step five of the workflow, where you will add the remaining named nets.

2.2.6. 5 - Nets

In this segment, you will add the remaining named nets. Most of them will go in the LED display modules, and two belong to the button signal nets.

Start with the button signal nets:

- MODE_SELECT
- TIMER_RESET

Here is the updated schematic for the group:

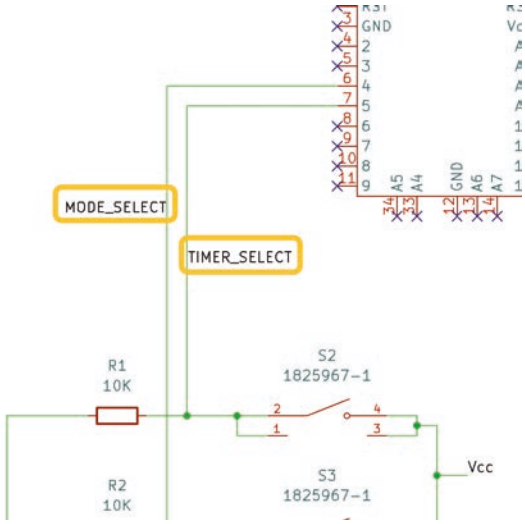


Figure 2.2.6.1: Named nets the button signal nets.

Then, continue with the LED displays group:

- Vcc
- GND
- CLKx
- DINx
- CSs

In the list above, "x" denotes a number from zero to three for the four modules. Here is the updated schematic for this group:

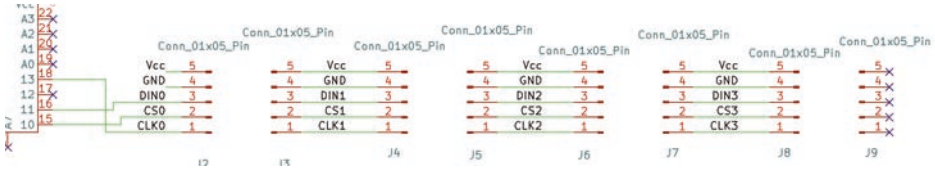


Figure 2.2.6.2: Named nets the LED display nets.

In the figure above, I have removed the original wire lines connecting pins five and four of J1 to pins 29 and 28 of U1 to declutter the schematic. I have marked the pins of J9 with the unconnected symbol.

This completes step five of the schematic design workflow. The schematic is fully wired, and the important nets are named. Below is the final version of the schematic:

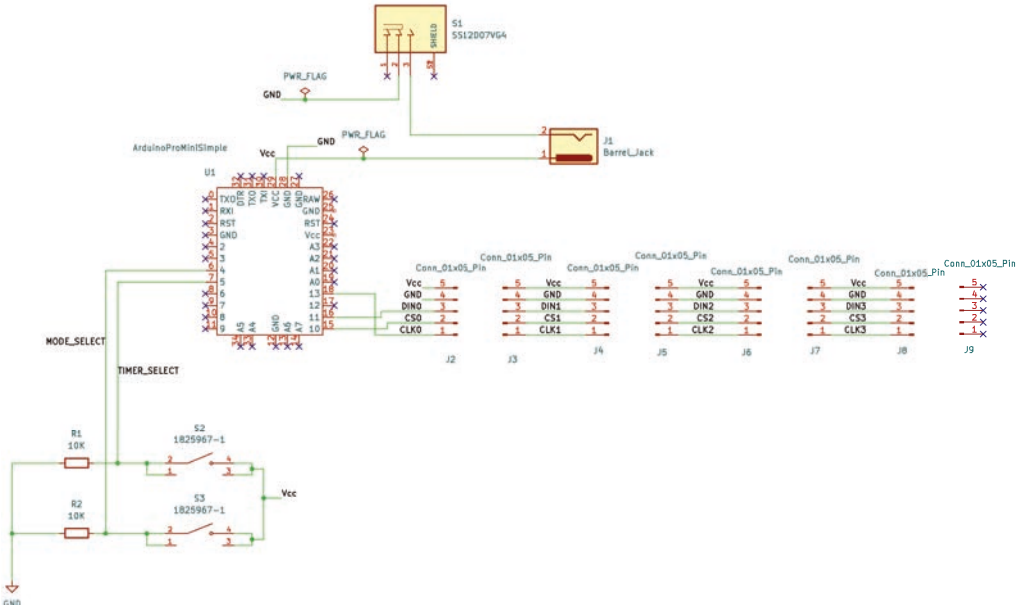


Figure 2.2.6.3: Wiring is complete.

Let's continue with an ERC.

2.2.7. 6 - Electrical Rules Check

With the schematic wiring complete, it is time to do a final Electrical Rules Check. Bring up the ERC window and click "Run ERC." The ERC shows me two warnings and four errors. The warnings are about the two switch symbols I modified, which I can safely ignore. There are two errors about unconnected pins (25 and 23 of U1). After removing the wires that connected them to the J2 connector, I forgot to mark those pins as unconnected. These two errors also relate to the last two, titled "Input Power pin not driven by any Output Power Pins".

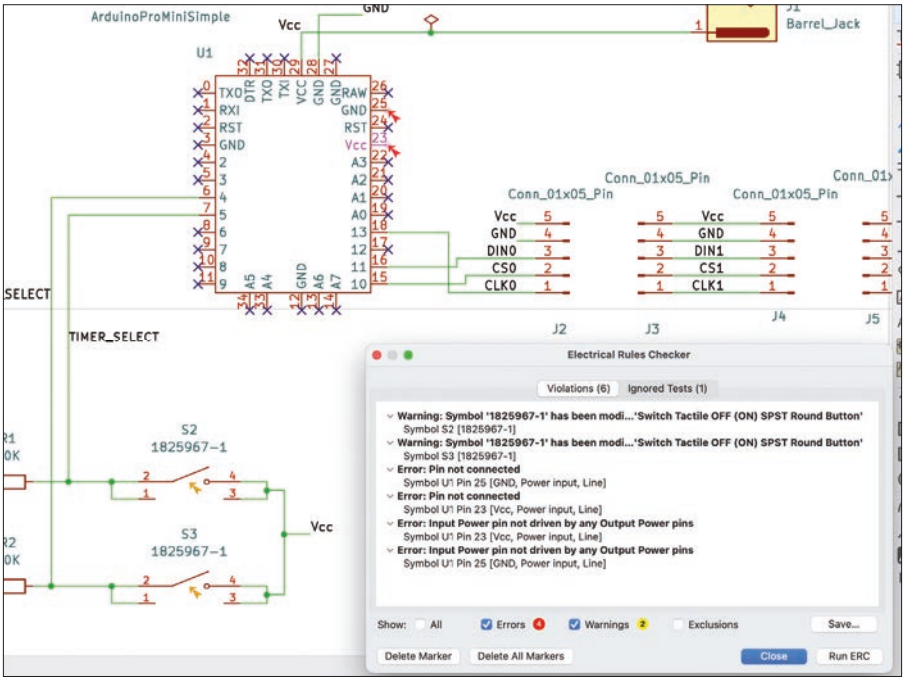


Figure 2.2.7.1: Two violation that I must fix.

As you can see below, I can fix all four errors by attaching a "not connected" symbol to pins 25 and 23 of U1.

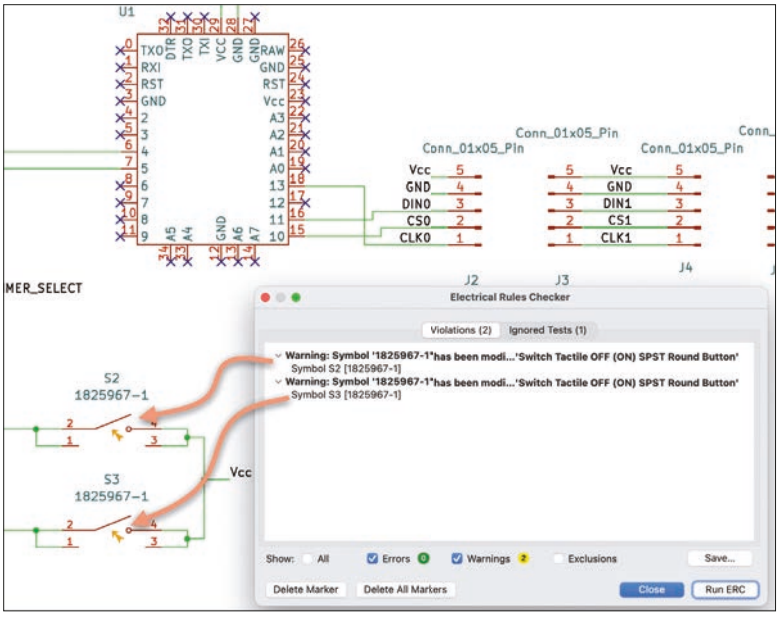


Figure 2.2.7.2: Two warnings are remaining.

The remaining violations warn me that the switch symbols have been modified in the library. I can safely ignore them. After the fixes brought forward by issues revealed by the ERC, the schematic looks like this:

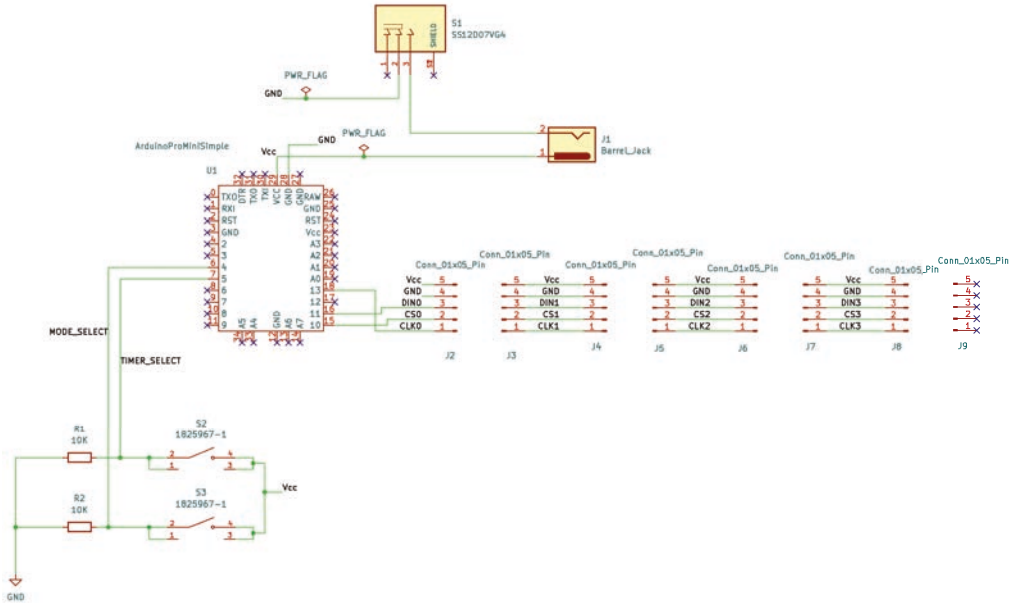


Figure 2.2.7.3: The project schematic, final.

Let's proceed with step seven of the workflow and add a few explanatory comments to the sheet.

2.2.8. 7 - Comments

In this step of the workflow, you will add graphics and text comments to improve the schematic's readability. Here are the comments I suggest you add:

1. Simple line boxes that join the LED display connectors resemble a single module each. They can look like this:

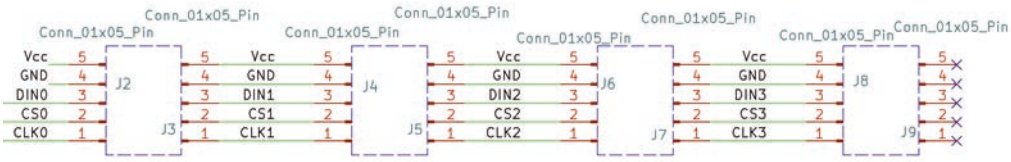


Figure 2.2.8.1: Boxes represent the LED display modules.

2. Replace the original pin header names with more descriptive ones. For example, "Conn_01x05_Male" becomes "LED1_IN". Here is the display block after editing the names:

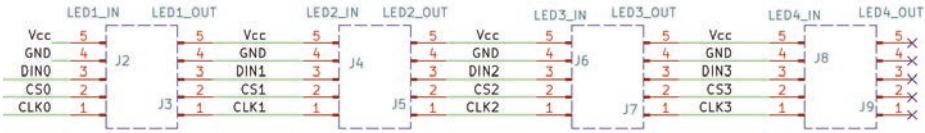


Figure 2.2.8.2: Pin header symbol names edited.

3. A text box with a reminder that specific headers have been flipped.

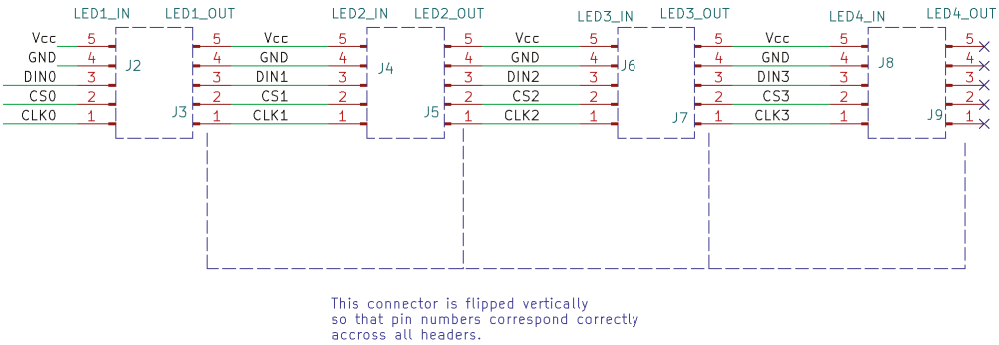


Figure 2.2.8.3: A reminder to self.

With the comments added, the sheet looks like this:

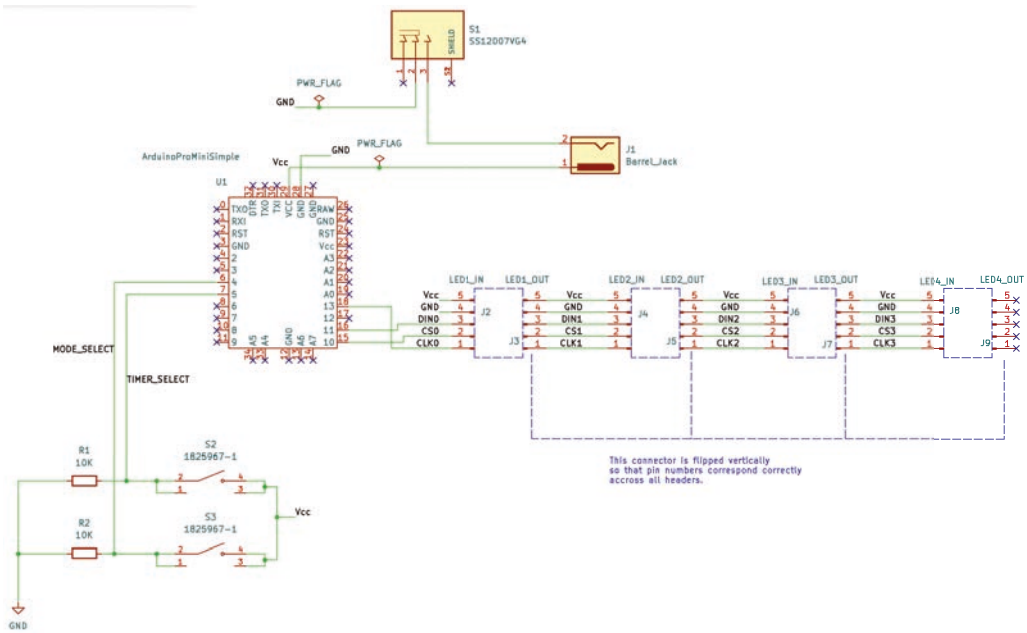


Figure 2.2.8.4: The schematic diagram with comments.

My original plan was to complete the schematic design workflow at this point and continue with the layout. However, I realized that I wanted to make a couple of last-minute edits. Let's look at those right away.

2.2.9. Last-minute edits

There are a couple of last-minute edits I'd like to make to the schematic before continuing with the layout. These are:

1. Add additional net labels that will be useful in the layout design.
2. Add symbols for the mounting holes and associate them with suitable footprints.

Begin by adding four mounting holes to the sheet. From the Symbol Choosers, look for the "MountingHole" symbol in the Mechanical library:

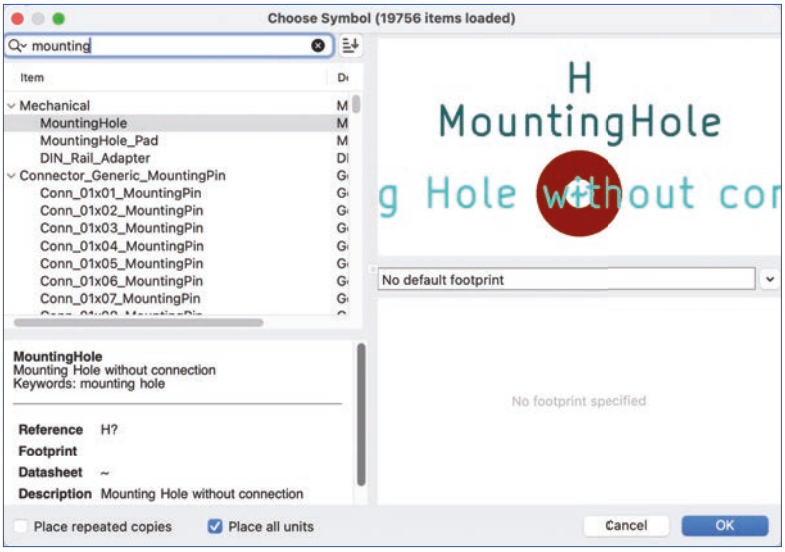


Figure 2.2.9.1: The MountingHole symbol.

Add the first mounting hole symbol to the sheet. Before duplicating it, double-click it to open its properties window and set a footprint. For the footprint, set it for the MountingHole_2.5mm option under the MountingHole library:

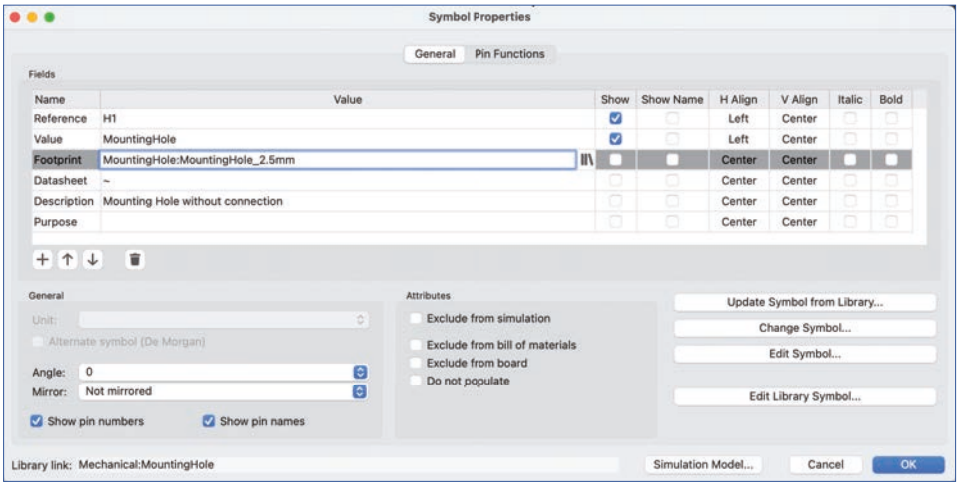


Figure 2.2.9.2: The mounting hole footprint.

Now, go ahead and create three duplicates of this symbol (they will all inherit the same footprint). Here is the result:

Figure 2.2.9.3: Four mounting hole footprints.

Next, work on editing and creating new net classes, and then assign nets to the net classes:

- Power (set earlier), change its name to "GND."
- Vcc
- Signal

Set the net class memberships:

- Vcc net belongs to the Vcc net class.
- GND net belongs to the GND net class.
- All other nets (like CLK0, CS1, etc.) below to the Signal net class.

Notice that in the screenshot below, I use regular expressions to match more than one nets per pattern. See the net class table in the Schematic Setup window below:

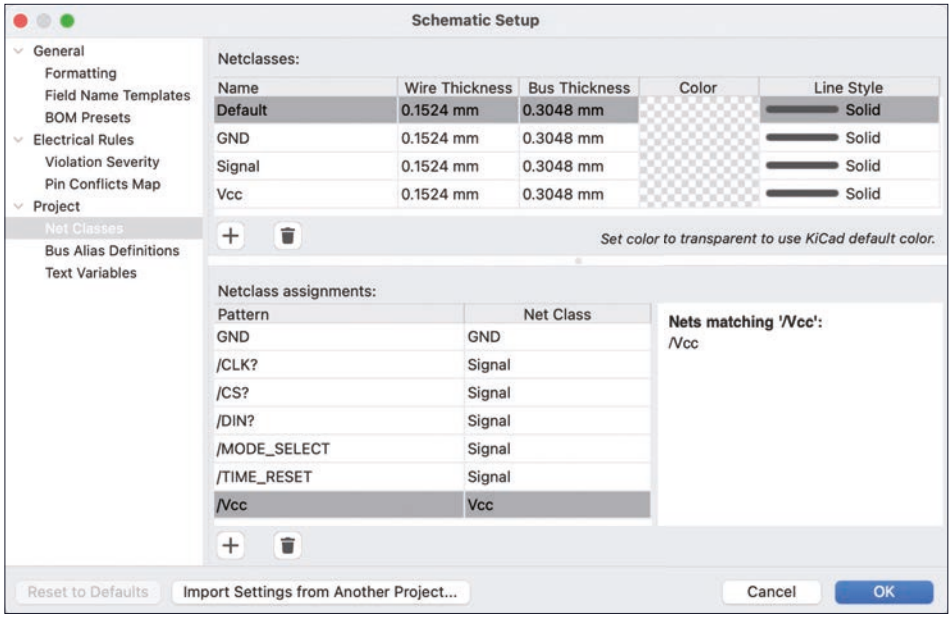


Figure 2.2.9.4: Nets and Net Classes.

This completes the work in the schematic design workflow. In the next chapter, we will continue with the layout process.

2.3. Layout design editing

In the previous chapter, you completed the schematic design of the breadboard power supply PCB. In this chapter, you will work on the layout design, following the workflow you learned in My book Fundamentals and Projects.

The main challenge in the layout design is the geometrical constraints set by the LED matrix display.

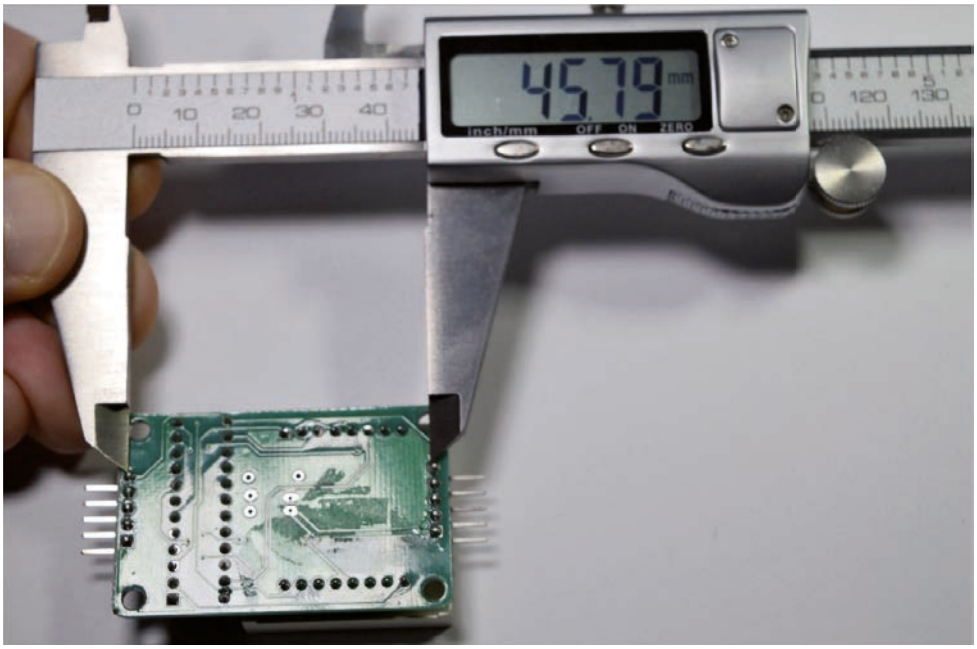


Figure 2.3.1: Measuring the LED matrix display module.

If measurements are not precise enough, the PCB will not be possible to assemble. I will take multiple measurements to help me precisely arrange the LED matrix display pin headers on the PCB.

Let's begin with the setup step.

2.3.1. 1 - Setup

The schematic for this PCB is complete. In this segment, I will continue working with the layout editor. Let's begin with the setup.

Start the layout editor and open the Board Setup window. Below, I list the values for the most important settings. Assume that all other values are the defaults.

- Board Stackup.
  - Physical Stackup.
    - Copper layers: 2 (this is a simple board, so two layers suffice).

- Board Editor Layers.
  - F.Cu: mixed (I will route signal and power tracks in this layer).
  - B.Cu: mixed (I will route signal and power tracks in this layer).

- Text & Graphics.
  - Text Variables: you will see the variable you set in the schematic design editor, named "design_version."

- Design Rules.
  - Net Classes: you will see the net classes and memberships you set in the schematic editor. Set the following values in the net classes table:
    - Vcc track width: 0.3 mm.
    - Vcc Via size: 0.85 mm.
    - Vcc Via Hole size: 0.45 mm.
    - GND track width: 0.3 mm.
    - GND Via size: 0.85 mm.
    - GND Via Hole size: 0.45 mm.

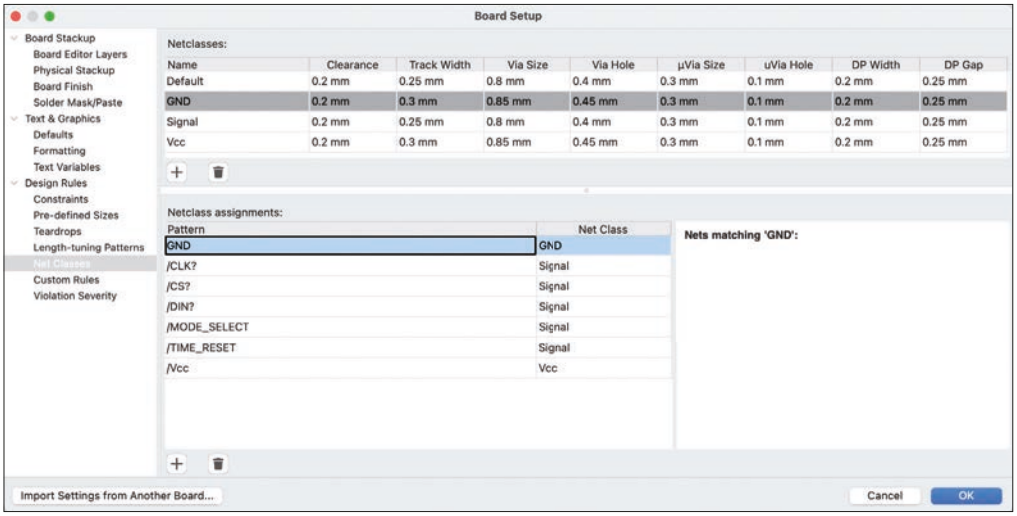
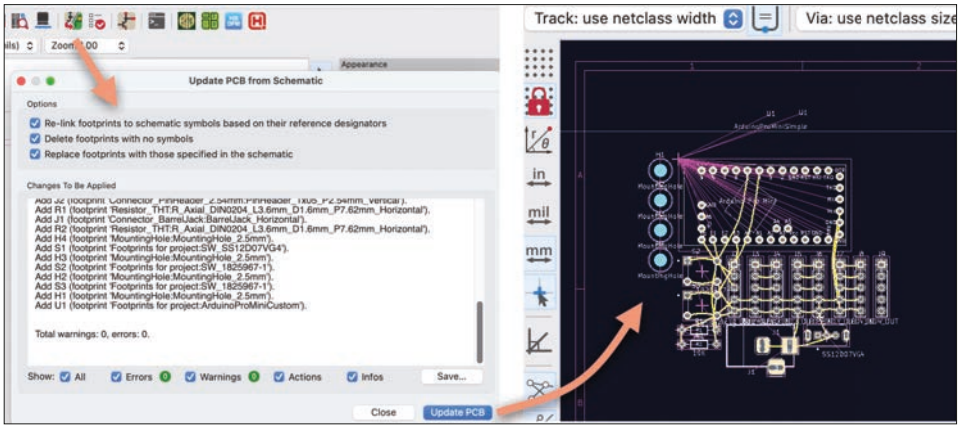


Figure 2.3.1.1: The changes in the Net Classes table.

To finish the setup, update the PCB from the schematic. Click the Update button in the top toolbar, then Update the PCB.



2.3.1.2: Updating the PCB from the schematic data.

With the layout editor updated with the latest schematic data, you can continue with step two of the layout process, where you will draw the rough outline of the PCB based on the board's geometrical and user interface constraints.

2.3.2. 2 - Outline and constraints

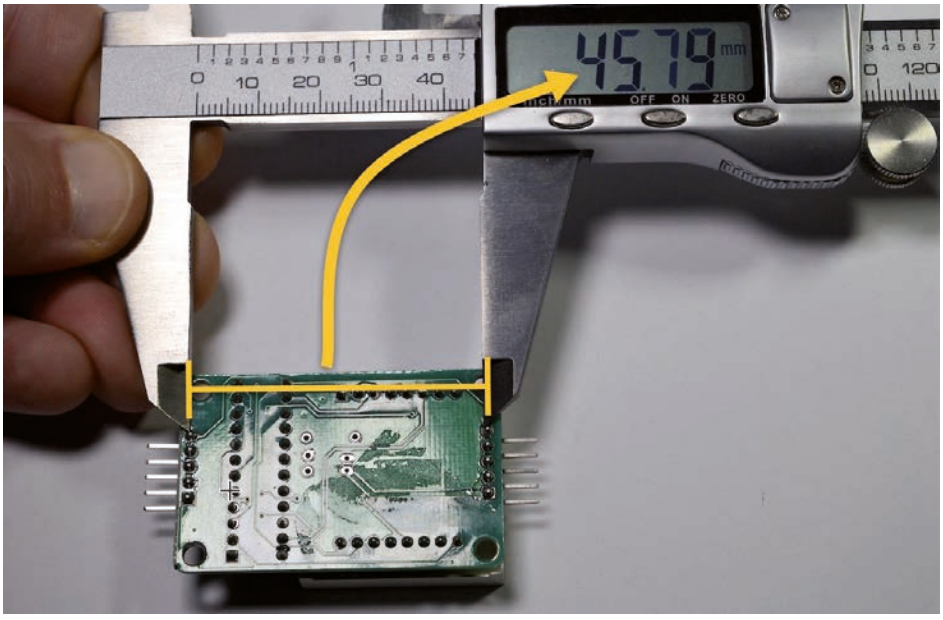
In this segment, you will draw a rough outline for the PCB. You will use measurements primarily from the LED matrix display module to draw this outline. You will also consider the dimensions of the other footprints and the user interface requirements.

The user interface requirements relate to the position of the buttons that the end user can press to control the device's functions, the barrel connector, and the power switch.

The rough outline of the PCB will help you place the footprints inside the PCB in the next segment of this chapter (step three of the process). Following the placement, you will refine the outline to its final shape.

Let's begin with the measurements. The final PCB must accommodate the four LED matrix displays in a tight arrangement so that the four combined displays look like a single unit. If the distance between the display pin headers is too small, even by a millimeter, the assembly will not be possible. If it is too large (even by a millimeter), the gaps between the individual displays will spoil the illusion of a single large display.

Use a caliper to take precise measurements of the LED matrix display module. The first measurement is the distance between the two pin headers:



2.3.2.1: The distance between the pin headers.

The caliper shows 45.79 mm for this distance. You will also need the width and height of the LED matrix module:

2.3.2.2: The width (left) and height (right) of the LED matrix module.

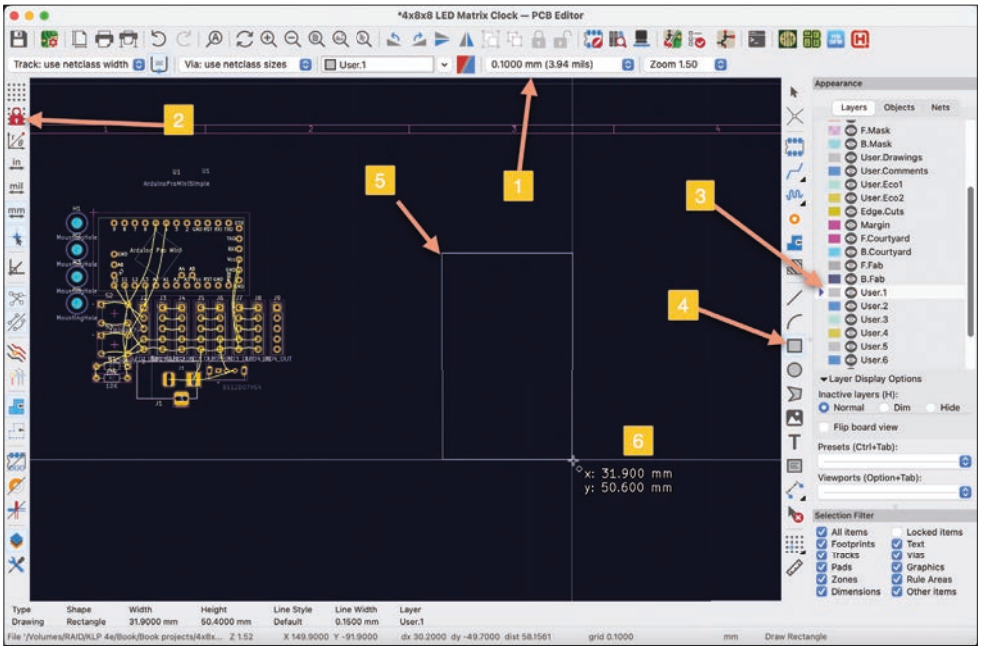
The width of the LED module is 31.88 mm, and its height is 50.59 mm.

The width and height measurements are sufficient to help you draw the rough outline of the board. You will need further measurements to help place the footprints within this outline, but you will do them in step three of the workflow (in the next segment of this chapter).

Return to the layout editor and switch the active layer to User.1. this layer can draw a rectangular graphic element representing the LED display module. Because you will be using the User.1 layer, this graphic element will not be included in the Gerber files. It will assist you in creating the rough outline (and then the final refined version) in the Edge.Cuts layer. I have annotated the screenshot below with numbers to help you follow my drawing process.

With User.1 layer active (3), use the rectangle graphics tool (4) from the right toolbar to draw a rectangle as close as possible to 31.88 mm X 50.59 mm. To achieve better accuracy, use a small grid size. I used 0.1 mm for the grid size (1). I also disabled the grid size override for this task (2) because my grid override setting for graphics differs from the 0.1 mm I want to use now.

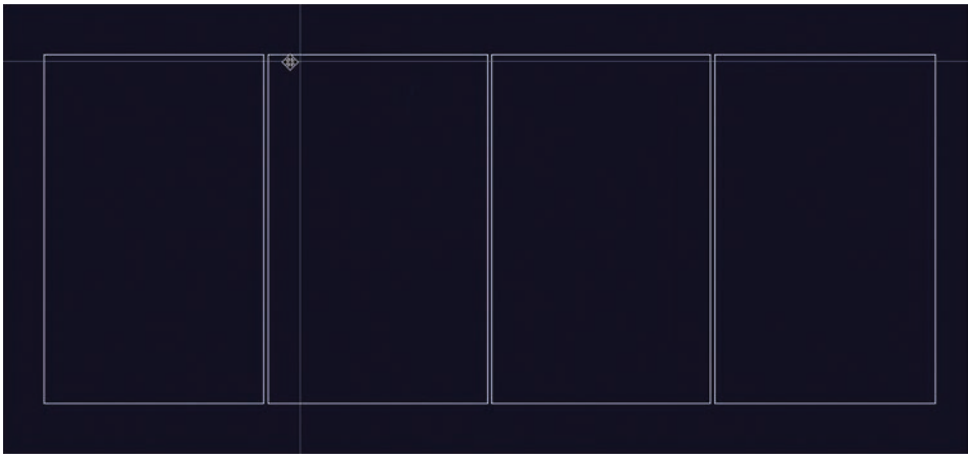
Start drawing the rectangle (5), and get the horizontal dimension as close as possible to 31.88 mm. In my example below, I started the drawing at the rectangle's top left corner and dragged the mouse pointer to the right and down (6). The objective is to get the "x" value as close as possible to 31.88 mm and the "y" close to 50.59 mm. My "x" is at 31.900 mm, and my "y" is 50.600 mm, which is acceptable. I am confident that these sizes are correct because when I measured the width of the module with the caliper, I allowed for a small but real gap between the caliper arms and the module board.



2.3.2.3: Drawing a rectangle representing an LED array module (left) and completed (right).

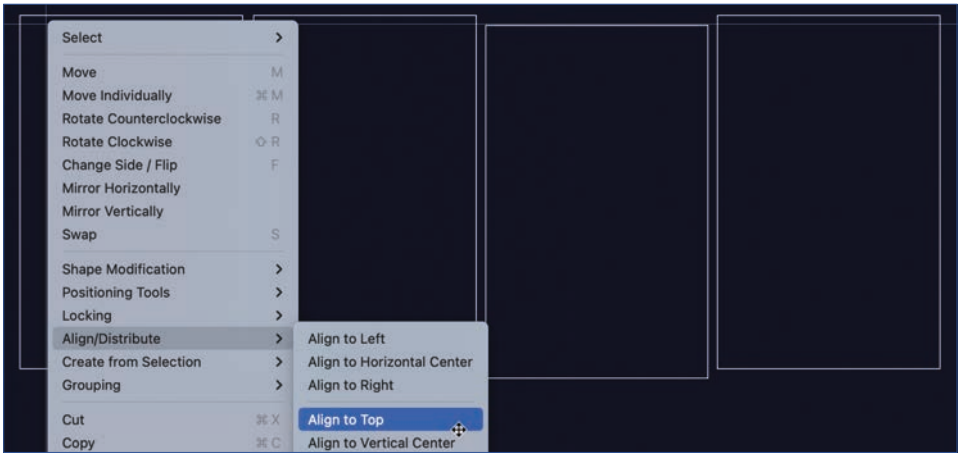
Now, you have a rectangle that represents the outline of a single LED matrix display module. The PCB will contain four of those modules. Create three copies of the rectangle and place them next to each other. To duplicate the rectangle, select it and use the Ctr-D/Cmd-D shortcut.

The result is below:



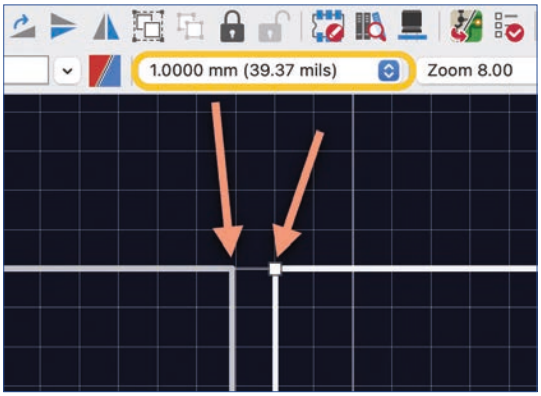
2.3.2.4: The four LED matrix displays.

Ensure that the four rectangles are equally spaced and justified. Select all rectangles using mouse click and drag, or hold down the shift key and click on each rectangle. Once all rectangles are selected, use the "Align/Distribute" functions under the context menus to align the selected items to the top.



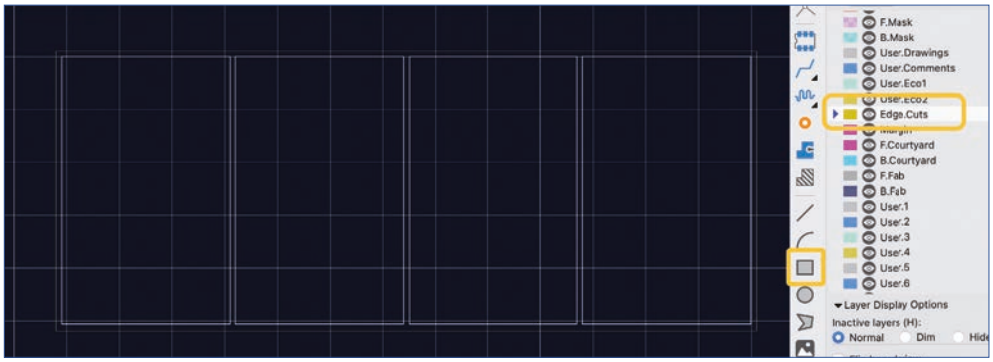
2.3.2.5: Align to top, and distribute evenly horizontally.

Another proper placement technique is to use the grid. Set the grid to a value that you wish to use for spacing the rectangles, and then zoom in so you can see the grid lines. Move the rectangles so that they are precisely one grid block apart, like this:



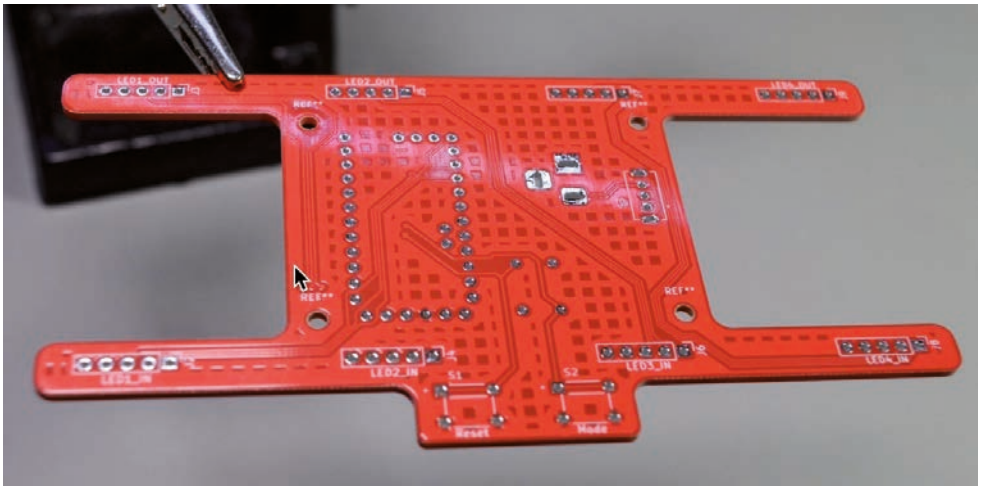
2.3.2.6: Using the grid to place the LED module rectangles.

With the LED matrix display module rectangles complete, you can draw the rough outline for the PCB. Switch the active layer to Edge.Cuts. Use the rectangle tool to draw a new graphic enclosing four smaller rectangles. The result looks like this:



2.3.2.7: Drawing the rough outline in Edge.Cuts.

I will place the component footprints within this rough outline in step three of the workflow; however, I need more guidance. My objective for the PCB shape is to have a center portion that contains the components and four arms that hold the first and fourth LED matrix modules. I want to place all footprints in the center of the board to cut out the substrate of the PCB on the left and right sides. See the final PCB below:



2.3.2.8: A reminder of what the final PCB will look like.

To help me with the footprint placement in the next step:

1. Switch to User.2 layer, and use the rectangle tool to draw a new rectangle in the middle of the rough outline.
2. Size this rectangle large enough to contain the footprints for the Arduino Pro Mini, the power barrel jack, the on-off switch, and four mounting holes.
3. Use the grid (I set mine to 1.00 mm) and the dx, dy values to help you draw.

Here is the result:

2.3.2.9: This rectangle will assist with the footprint placement.

The new rectangle will be helpful in the next step of the workflow. Even if it is too small or too large, it will still guide you toward placing the footprints in the appropriate positions. Because it is in User.2 layer, you can switch it off when you no longer need it.

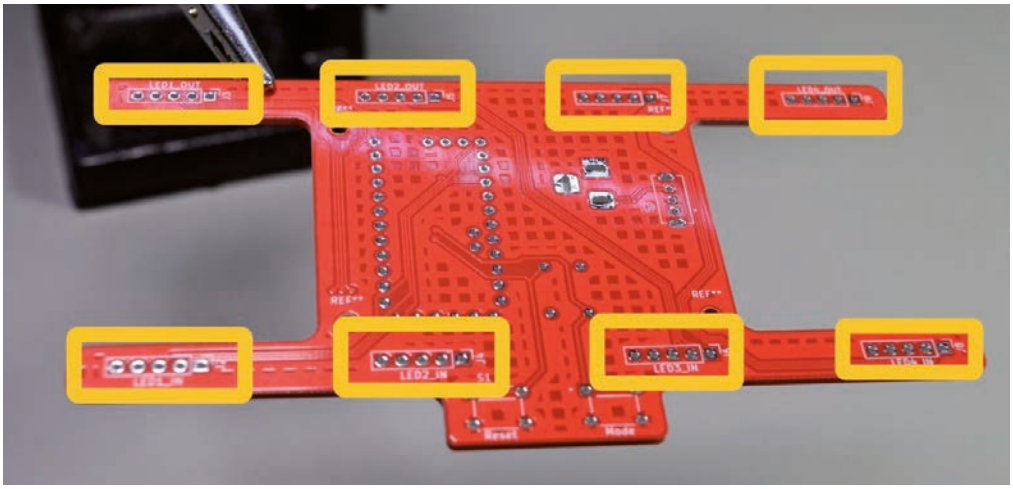
I have not made a provision for the buttons; however, this is not a problem. I will place the buttons using the rough outline as a reference and then enclose them in the perimeter of the PCB when I do the refinement.

Let's continue with the placement of the component footprints in the next segment of this chapter.

2.3.3. 3 - Place components

In the previous segment of this chapter, you prepared the layout editor for the placement of the component footprints. You created a rough outline for the PCB and marked the positions of the LED matrix display modules. In this segment, you will complete step three of the layout workflow and place the component footprints within the rough outline of the PCB.

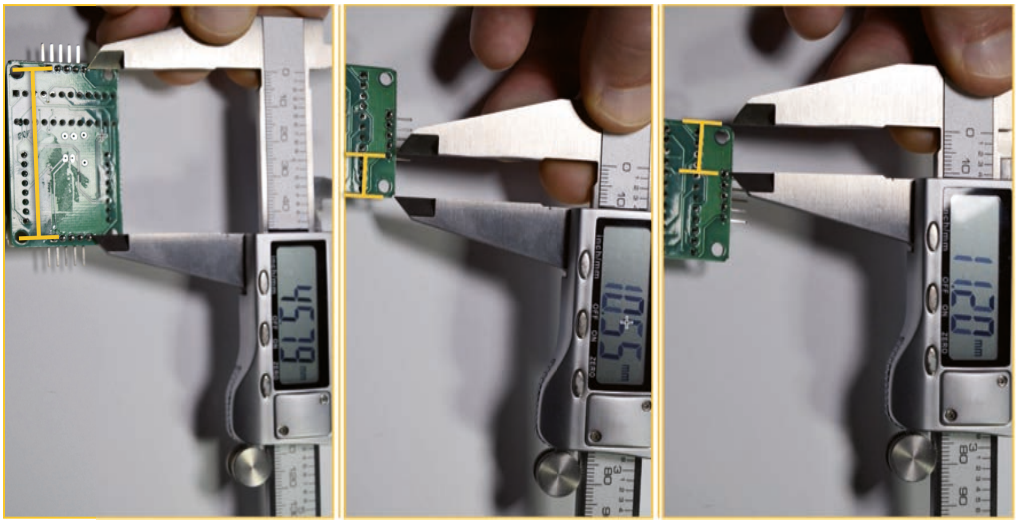
You will start the footprint placement with the footprints that define the PCB's most constraint-important elements: the LED matrix display pin headers.



2.3.3.1: Starting with the LED matrix display pin headers.

To help accurately position these headers within the display rectangles in User.1 layer, you will need to take additional measurements using your caliper. Unfortunately, the pin headers are not placed symmetrically in my LED matrix display modules. The distances between the headers and the modules' sides are different. This means that you will need to measure the distances between the header and the two sides of the PCB and between the two headers.

See my measurements below:

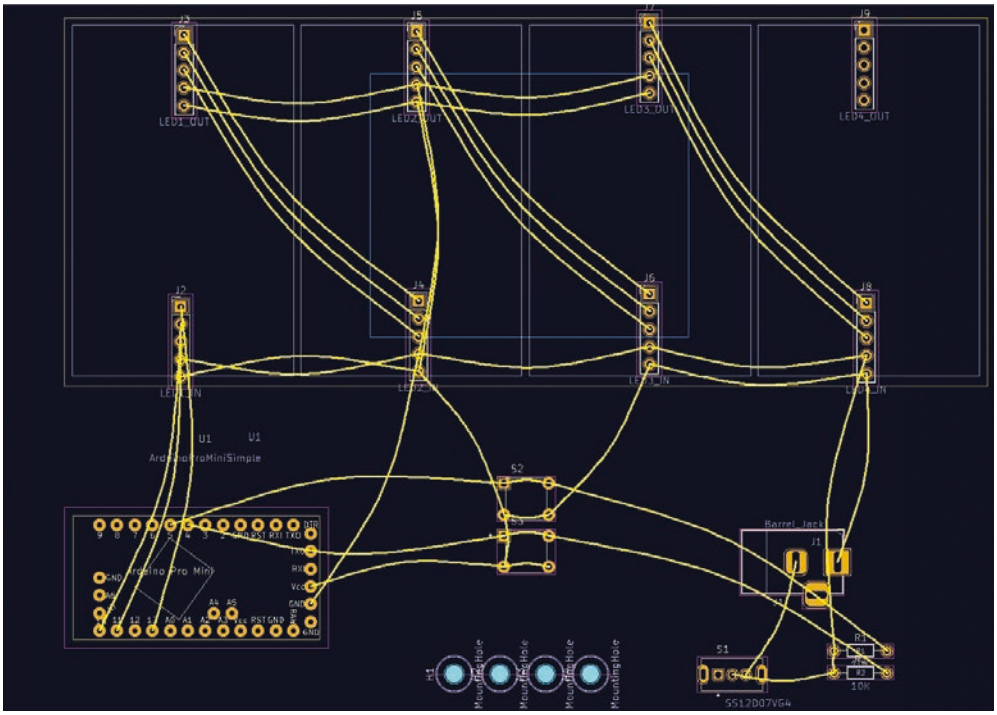


2.3.3.2: Pin header measurements.

The important pin header distances are:

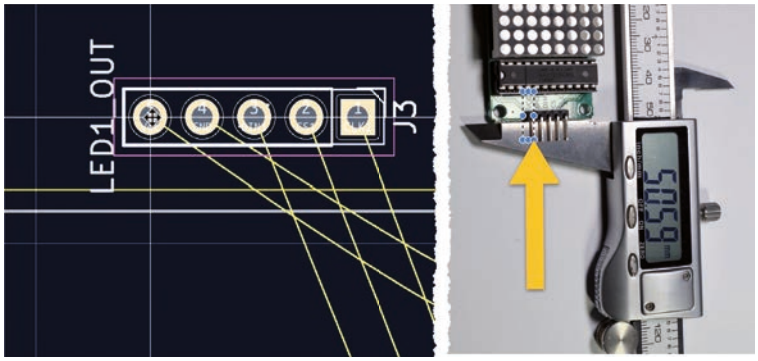
- Left board edge to left-most pin: 11.20 mm.
- Right board edge to right-most pin: 10.55 mm.
- Distance between pin headers: 45.79 mm.

My working grid size is 0.254 mm. Start by sorting out the various footprints. Move the LED display footprints to their approximate positions. See my example below:



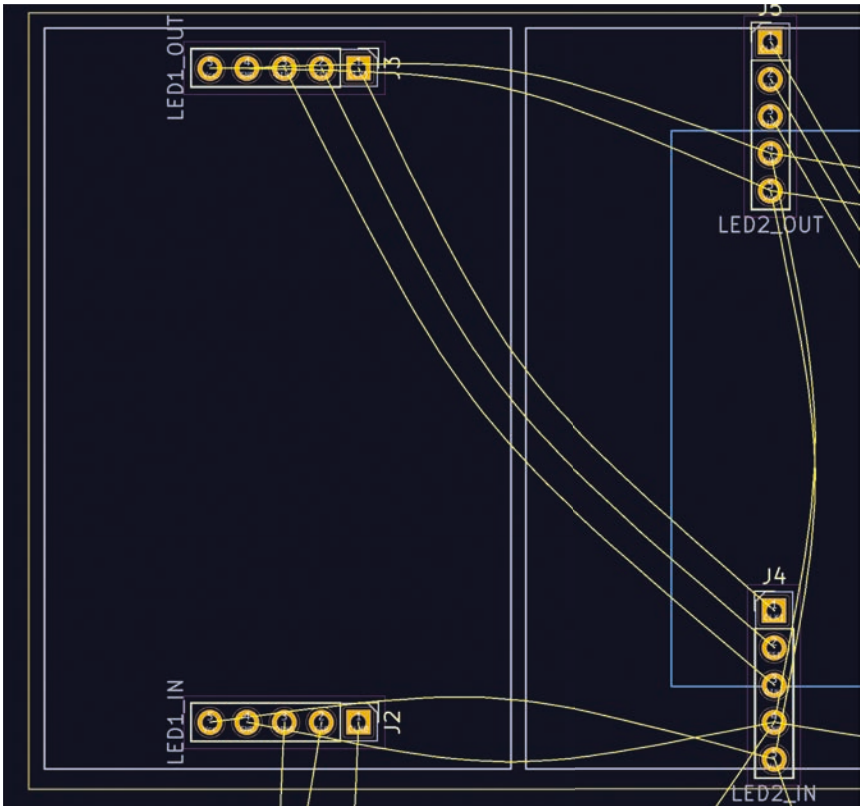
2.3.3.3: Unbundled footprints.

Orient the pin header footprints. Use the original part to find out the correct orientation. For example, for J1 (input header for the first display), the Vcc pin should point left, as in the physical part:



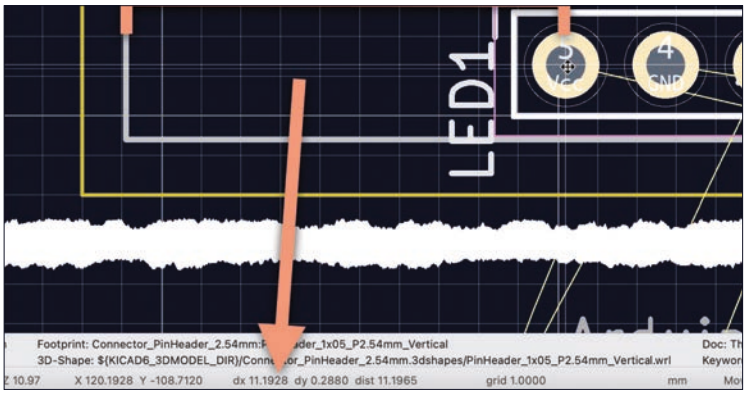
2.3.3.4: Pin header orientation.

The orientation of J3 (the output header for the same display) should have the same orientation, with the Vcc pin towards the left. To achieve satisfactory accuracy in the placements that follow, you will need a relatively fine grid setting. I use 0.0254 (1 mil) for the footprints, tracks, and graphics. Below are J2 and J3, which belong to the first LED display correctly oriented:



2.3.3.5: J1 and J3 oriented.

With J2 and J3 correctly oriented, the next task is to set them in the correct position using the measurements at the segment's start. Use the dx and dy values to measure the center of pins five and one of J2 against the left-most display module rectangle's bottom, right, and left edges. Below, you can see the positioning of J2 against the bottom and left edge of the rectangle:



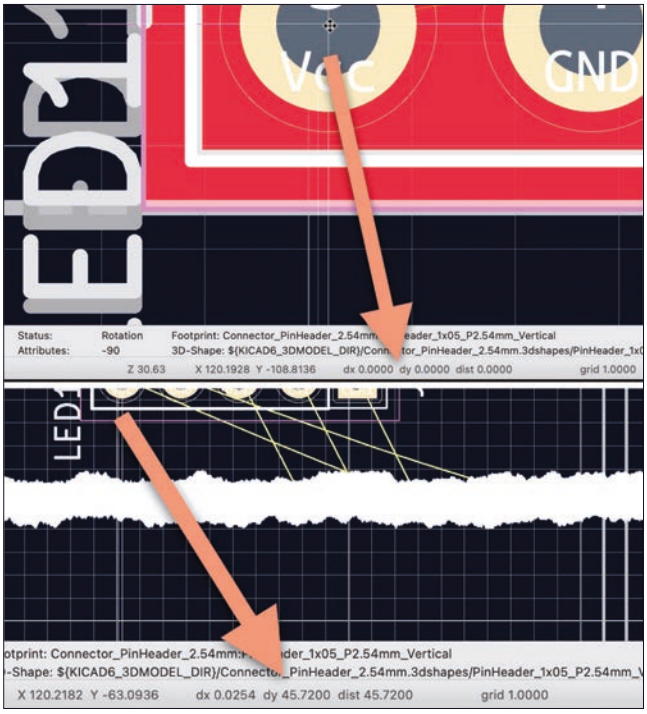
2.3.3.6: J2 pin five is 11.1928 mm from the left edge.

For J2, pin five is 11.1928 mm from the left edge. This is close enough compared to the measurement of 11.20 mm. The distance from J2 pin 5 to the bottom edge is not critical. What is critical is to ensure the correct distance between J2 and J3. In my instance, I placed J2's footprint on the bottom borderline (courtyard) just above the white module outline in the User.2 layer that I drew in the previous step. You can choose to use a different method. For example, you can opt to center headers J2 and J3 inside the module outline. To do this, subtract the distance between J2 and J3 from the module outline and then half it (50.59-45.79=4.8 / 2 = 2.4mm). You can then distance J2 around 2mm to 3mm from the bottom edge.

Continue with J3, the output header for the same display module at the top of the rectangle. Two considerations dictate the placement of J3:

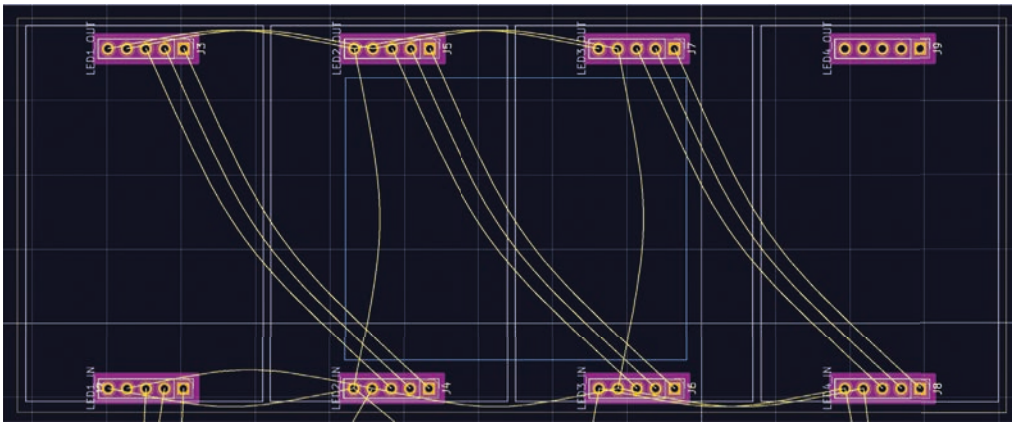
1. The distance between J3 pin five and the left edge of the rectangle. This distance must match that same distance for J2.
2. The distance between J2 and J3 must be equal to or close to 45.79 mm.

Start with the second consideration. Select J3 and move it to the top of J2. Press the space bar to reset the dx and dy counters, then move J3 upwards. You want to maintain dx to zero and set dy close to 45.79 mm. I am using a grid size of 0.0508 mm. With dx equalling zero, you know that J3 is the same distance from the left edge as J2.



2.3.3.7: Placing J3 in relation to J2.

A final operation I did on J2 and J3 was to align them to the left to ensure perfect alignment. Now that you have positioned J2 and J3 lock them to avoid accidental movement. Follow the same process to place the remaining display pin headers. Remember to lock each footprint in place after confirming the correct positioning to avoid inadvertently moving them later. The results are below:



2.3.3.8: Display pin headers are now in place and locked.

Before continuing, I will do one last set of measurements to ensure I have made no placement errors. For this round of measurements, I will use the interactive ruler from the right toolbar:

2.3.3.9: Sanity distance measurements using the interactive ruler.

With these measurements, I confirm that the placements are correct to continue with the placement of the remaining footprints. Try to make everything fit within the blue rectangle in the User.2 layer. Below is the order by which I continued, with comments where necessary:

1. The Arduino Pro Mini is the largest component and will dictate the positions of other components around it.
2. The barrel jack. I oriented it so that the power cable could plug in upwards. I have placed it with plenty of clearance from below to accommodate rigid power cables.
3. The slide switch (on/off) next to the barrel connector.
4. The resistors.
5. The buttons are between pin headers J4 and J6 and below the bottom edge of the board outline. You will enclose them within the board outline when you refine it in the next segment.
6. The mounting holes.

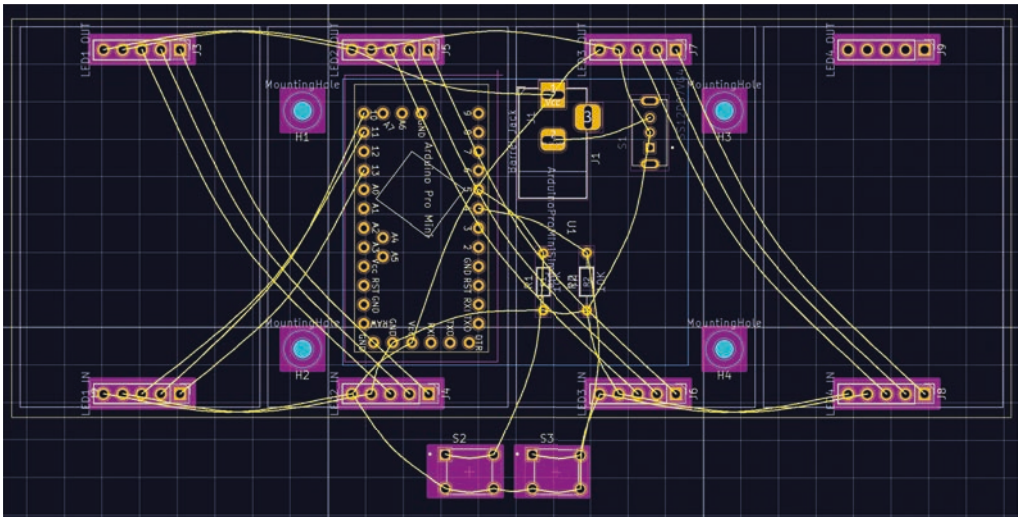
Use the grid and the object alignment tools to justify the buttons horizontally and ensure that they are symmetrically around the center axis of the board.

Do the same for the mounting holes. To design an enclosure, you want the holes to be symmetrical against the center axis of the board and justified horizontally and vertically. This will be easier if the hole positions are regular rather than irregular. Remember to lock the holes after positioning.

Another consideration for the buttons is that, given their size, their position will affect the routing of the copper tracks. Don't place them too close to the Arduino Pro Mini footprint, as this will make it more difficult to route tracks to and from the Arduino.

For the buttons, justification is essential to ensure that your board does not end up with miss-aligned buttons.

When you have finished with the placement, lock all footprints in place. Below, you can see how I placed the footprints in my instance of the board:

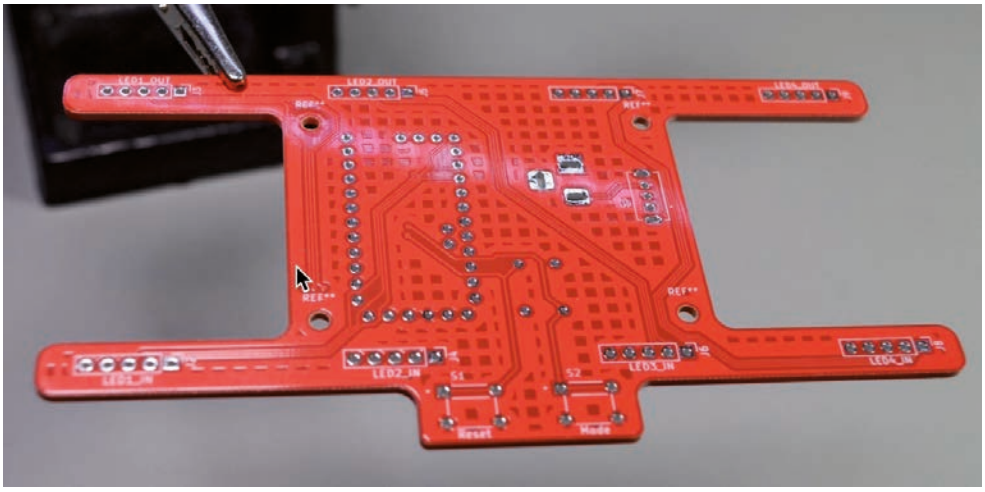


2.3.3.10: Completed positioning of footprints.

With the positioning of the footprints complete, let's refine the outline of the PCB.

2.3.4. 2 - Refine outline

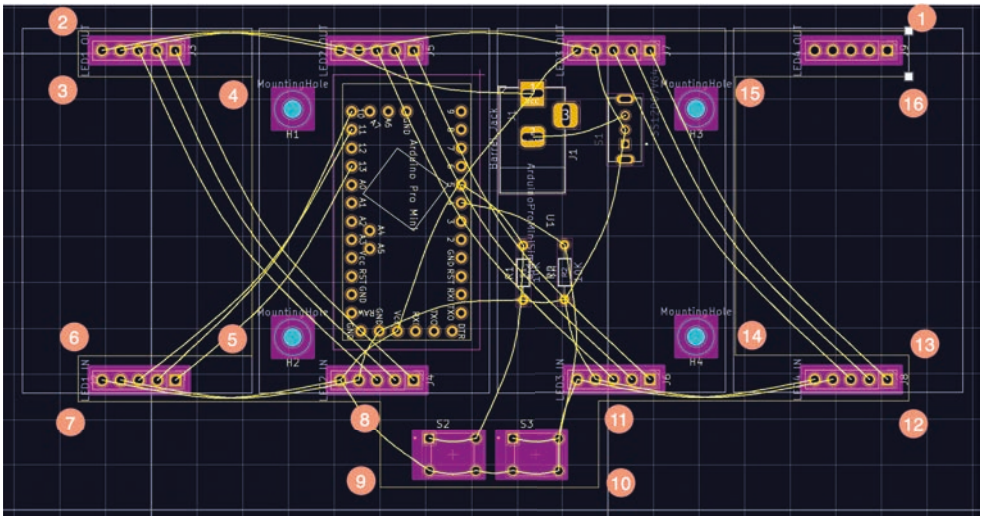
Let's complete the outline of the board. Your objective is to design an outline in the Edge.Cuts layer so that the outcome produces the result you can see below:



2.3.4.1: Draw this shape in the Edge.Cuts layer.

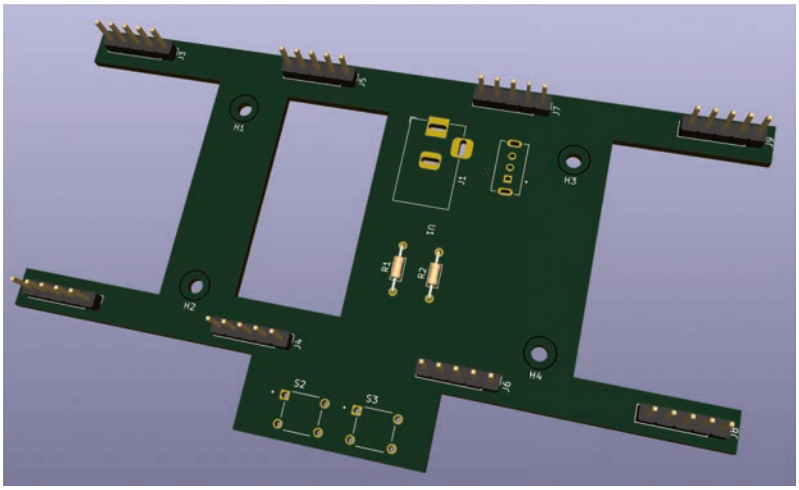
You no longer need the rectangle you draw in User.1 layer. You can either delete it or make it invisible. Either way, the graphics in the User.2 layer and the footprints provide enough guidance for drawing the refined outlines in Edge.Cuts.

Activate the Edge.Cuts layer, select the line tool from the right toolbar and start drawing from the top right corner of the board. My grid size is 0.6350 mm. Work your way around the display module pin headers and the pushbuttons. Use the full-screen crosshair cursor and the dy/dy values to help you draw with symmetry. Below, you can see the result of this work. The numbers mark the positions and sequence of the clicks during my drawing session.



2.3.4.2: Outline with 90-degree corners.

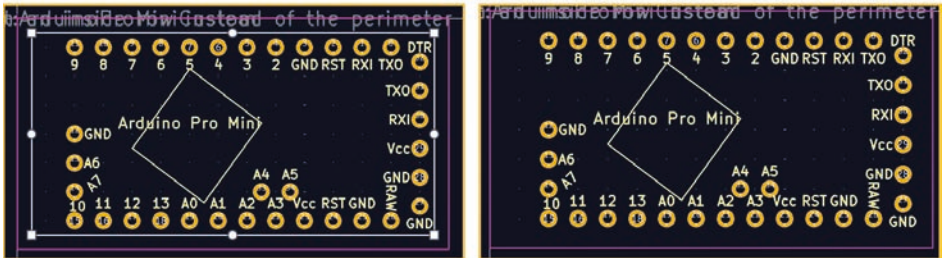
At this point, you have a board outline in the Edge.Cuts layer. It contains all footprints and has 90-degree angles. You can use the 3D viewer to see it in 3D.



2.3.4.3: Board in 3D. There is a hole where the Arduino footprint should be.

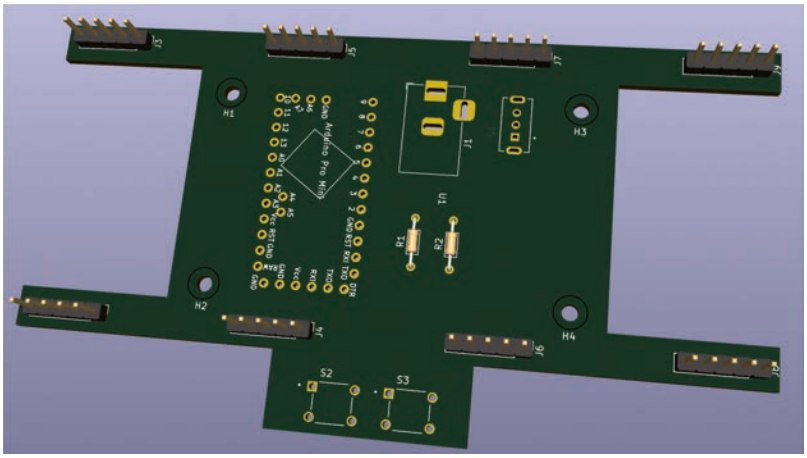
The 3D viewer shows a problem with the Arduino footprint. Instead of the Arduino Pro Mini headers, the 3D viewer shows a rectangle opening. The Arduino Pro Mini footprint I designed contains a rectangle on the Edge.Cuts layer. In orange, you can see this rectangle enclosing the Arduino module's pin headers in 2.3.4.4. I will need to fix this before I continue.

Right-click on the Arduino Pro Mini footprint and select "Open in Footprint Editor." Select the rectangle in the Edge.Cuts layer, delete it and save the footprint.



2.3.4.5: Arduino Pro Mini footprint with Edge.Cuts outline (left) and without (right).

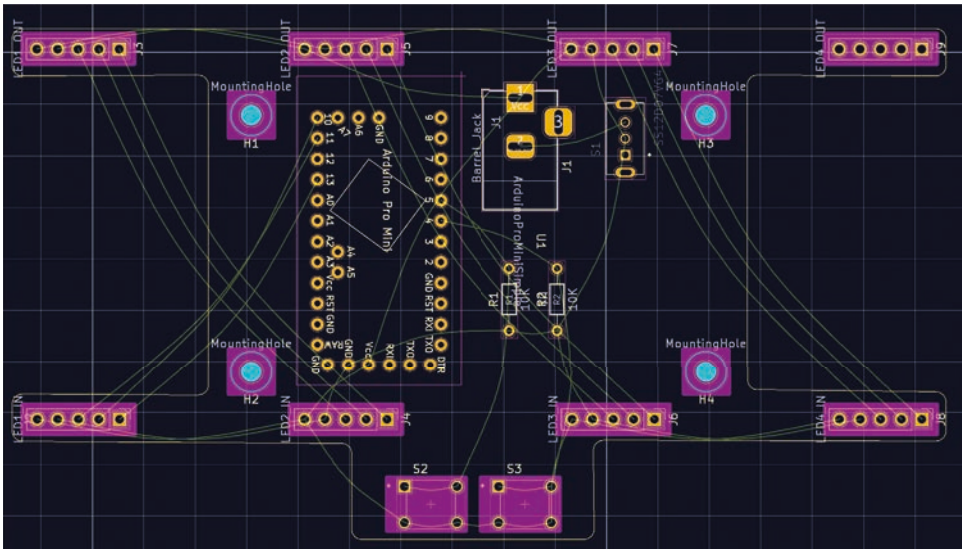
Close the footprint editor and return to the layout editor. The PCB layout is updated, and you can try the 3D viewer again.



2.3.4.6: The Arduino Pro Mini footprint correctly rendered in 3D.

The Arduino Pro Mini footprint is now rendered correctly in the 3D viewer.

Continue to round the corners of the layout as you have done in the previous project. Below is my final refined PCB outline:



2.3.4.7: The final PCB outline with rounded corners.

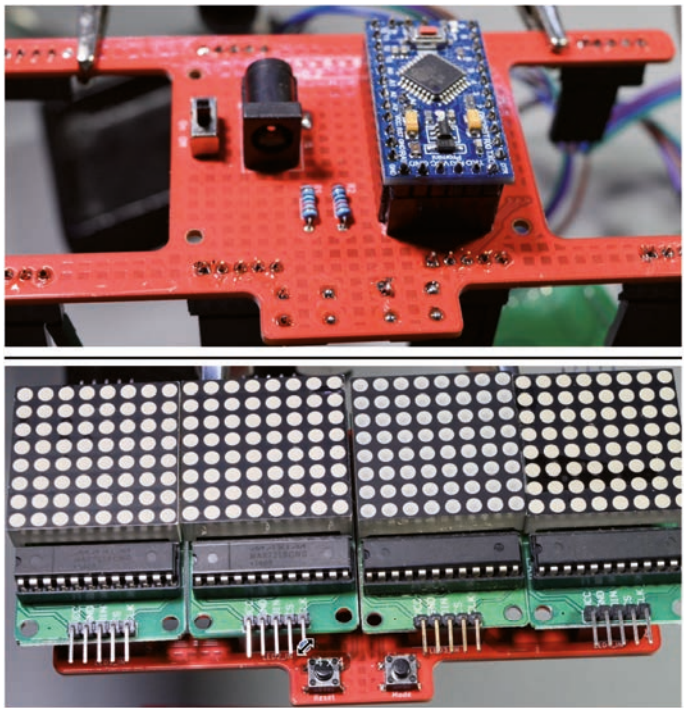
Now that you have refined and completed the board outline, you are almost ready to work on the board copper routes. However, there are first a few changes that I would like to make relating to the footprints. Currently, all footprints are placed on the top copper layer, as this is the default. I want to change the position of the Arduino Pro Mini, barrel connector, switch, and resistors to the back copper layer. This will leave the top copper layer for the displays and buttons.

Let's refine the footprint positions next.

2.3.5. 3 - Move footprints

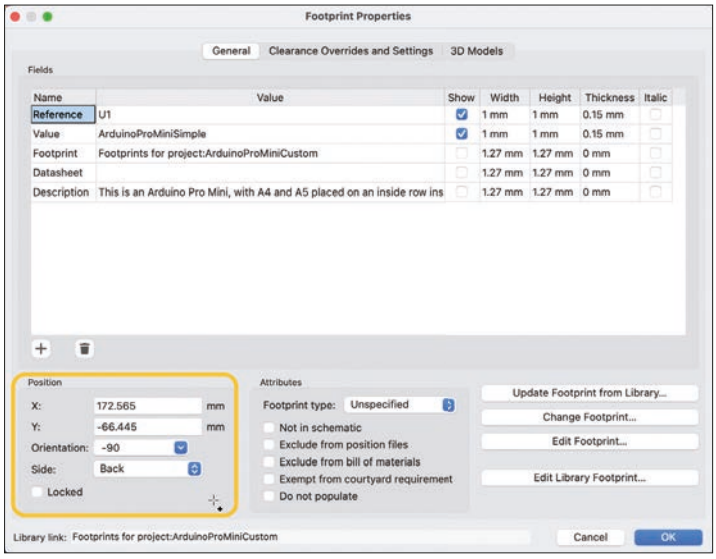
For this PCB, I'd like to place all footprints other than the LED displays and the buttons to the back copper layer. This will allow me to assemble these components out of view of the user and retain the front layer for the user interface components.

You can see the component placement objective in the photographs below:



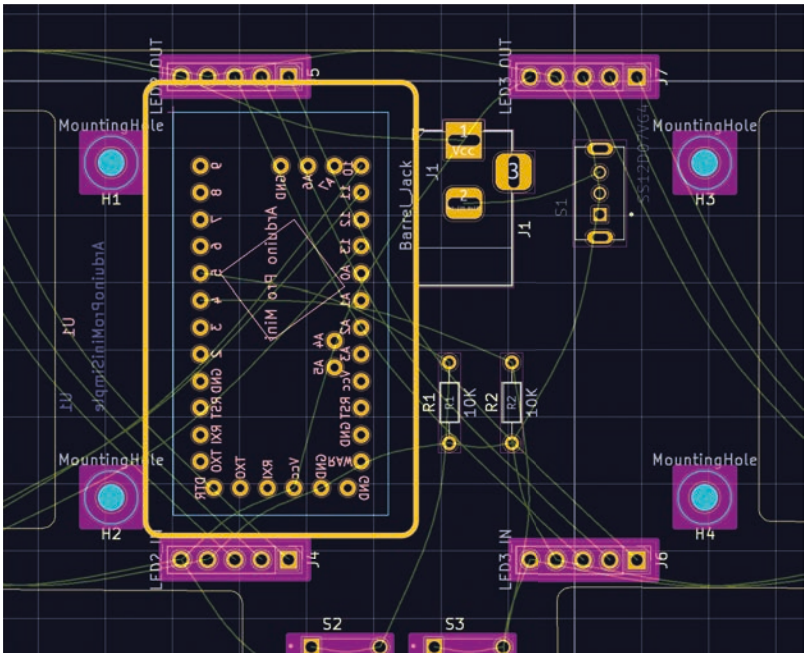
2.3.5.1: The placement of components in the back (top) and front (bottom) copper layers.

To select a placement layer for a footprint, bring up the footprint's properties window (double-click on the footprint), and in the "Position" group, set Side to "Back."



2.3.5.2: Setting the position of a footprint to "Back."

Click OK. The editor will flip the footprint over to the back copper layer and may also move. Move the footprint back to its correct position. In the example below, I have moved the Arduino Pro Mini footprint to the back copper layer and corrected its position. Notice that the elements that make up this footprint (text, graphics) now look mirrored.

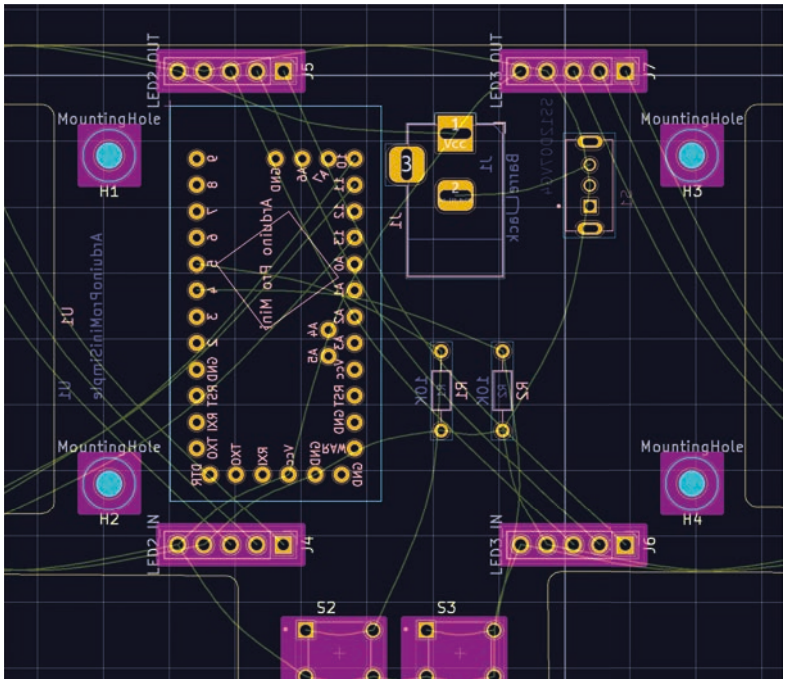


2.3.5.3: The Arduino Pro Mini footprint is now in the back copper layer.

Continue to set the position of the following footprints to "Back":

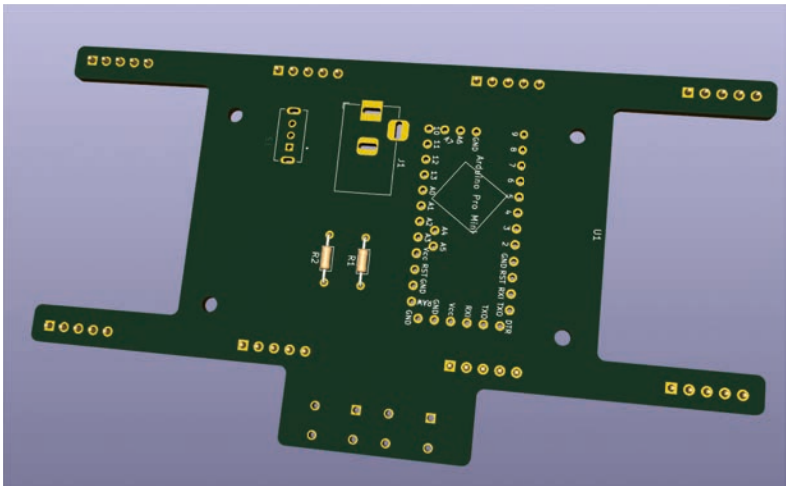
1. Arduino Pro Mini, U1.
2. Barrel connector, J1.
3. On/Off switch S1,
4. Resistors R1, R2

After this work, the board will look like this:



2.3.5.4: All listed footprints are now in the back copper layer.

Use the 3D viewer to see a 3D rendering of the back of the board:



2.3.5.5: 3D rendering of the back of the board.

The 3D viewer shows the footprints of the listed components on the back of the board. You can continue routing with the work on the board outline and the footprint placement complete.

2.3.6. 4 - Route

With the board outline and footprint placement complete, the next step in the layout workflow is routing the copper tracks. You are working on a two-layer board.

You will create most connections between the GND pads in the back copper layer. Instead of drawing copper tracks, you will draw a single copper fill connected to the GND net.

Use the front copper layer for all other connections. If a copper route cannot be completed in the front copper layer, you can use vias to switch to the bottom to complete the routing.

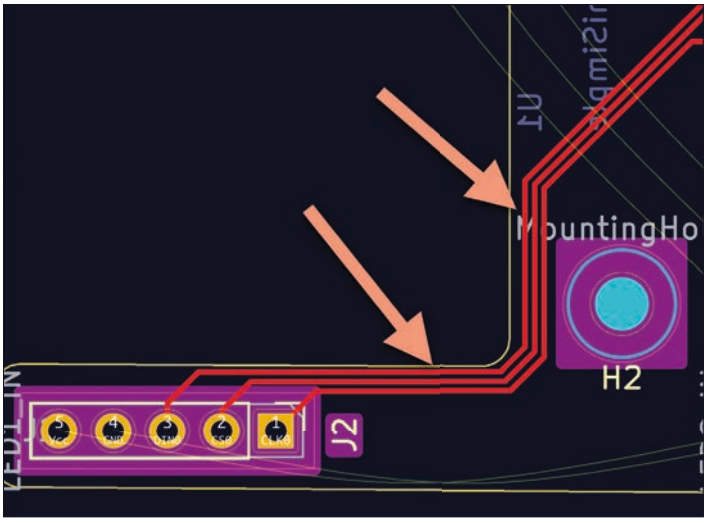
In the front copper layer, you will connect Vcc pads using a copper fill connected to the Vcc net.

Switch to the front copper layer and begin the routing process.

I started my routing session from the J2 and J3 pin headers and continued with the rest of the pin headers, followed by the components in the back copper layer. I set the interactive router to the "walk-around" mode.

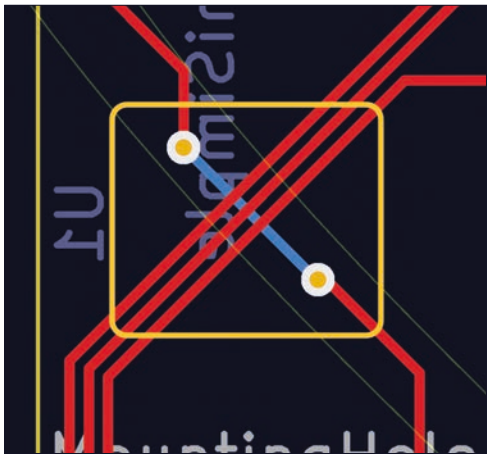
Here are some guidelines to help you with the routing process:

1. Don't route the GND pads. You will route GND pads using the copper fill method, which was also used in the previous project.
2. Don't route the Vcc pads. You will route them using a copper fill in the front copper layer connected to the Vcc net.
3. For the routes that originate in J2, J3, and J8, be careful to draw them close to the edge of their respective PCB arm and the routing hole. This leaves ample space for routing the remaining tracks later in the process (see below).



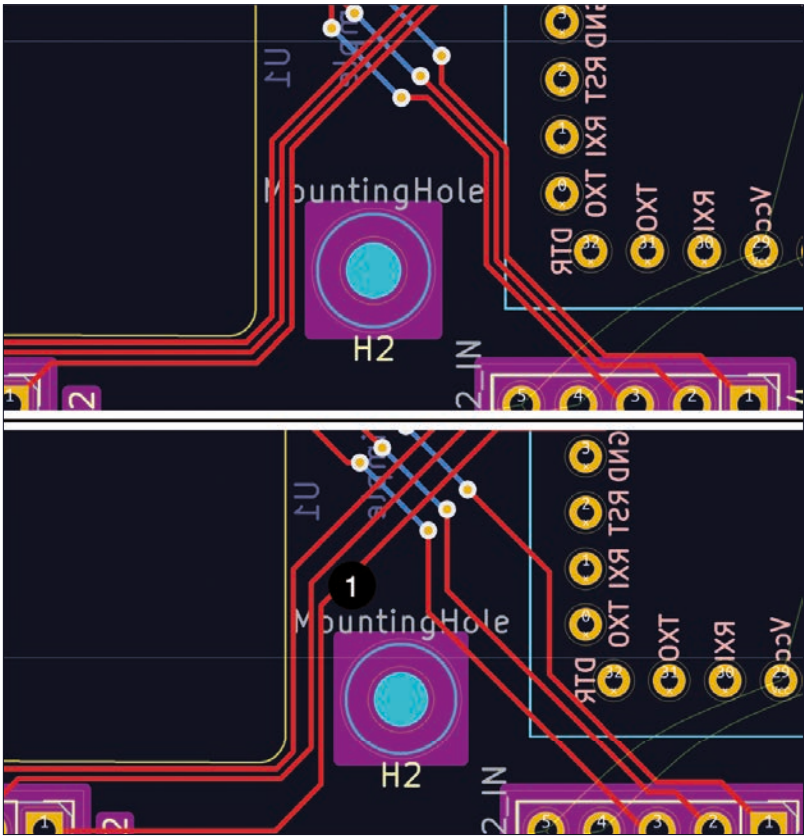
2.3.6.1: Drawing the J1 routes close to the edges of the board.

4. Try not to leave spaces between tracks. Use the Move command (click track to select and type "D" to move) to move tracks and reduce gaps to the minimum track spacing the DRC allows. Large gaps between tracks waste space and make it harder to draw new routes.
5. Opt for shorter tracks whenever possible.
6. If you must choose between a long or shorter track requiring vias, opt for the second option (see below).



2.3.6.2: Using vias to shorten the total length of a track.

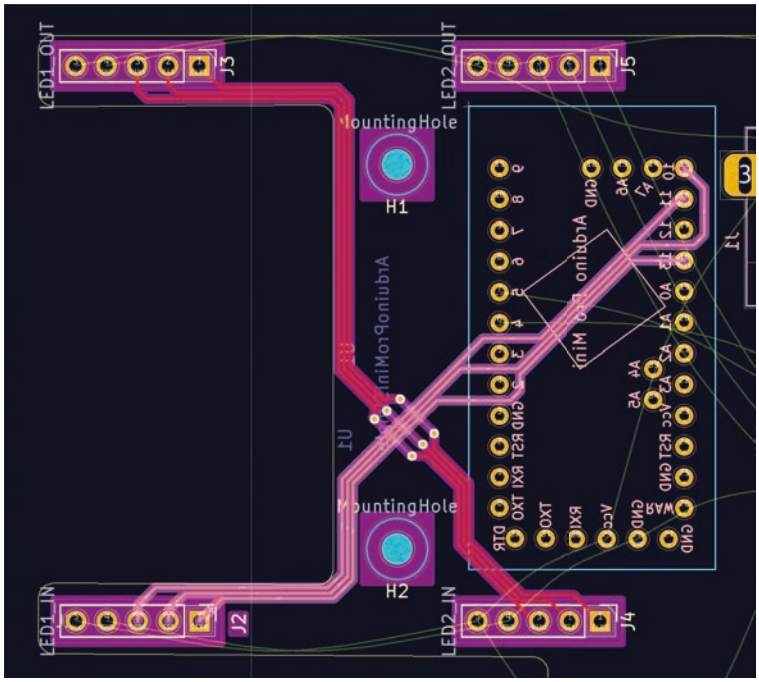
7. When you must switch a signal track to the bottom layer via a track, minimize the length of the track in the bottom copper layer. See the example above.
8. Routing can often be completed by moving existing routes and releasing space on the board instead of using vias. Explore this option first instead of creating a new via.
9. Optimise the use of space by moving existing copper routes when you finish routing a group of pins. For example, in the screenshots below, you can see a completed but not optimized set of routes (bottom) and the same optimized set (top).



2.3.6.3: Copper track shapes original (bottom) and optimized (top).

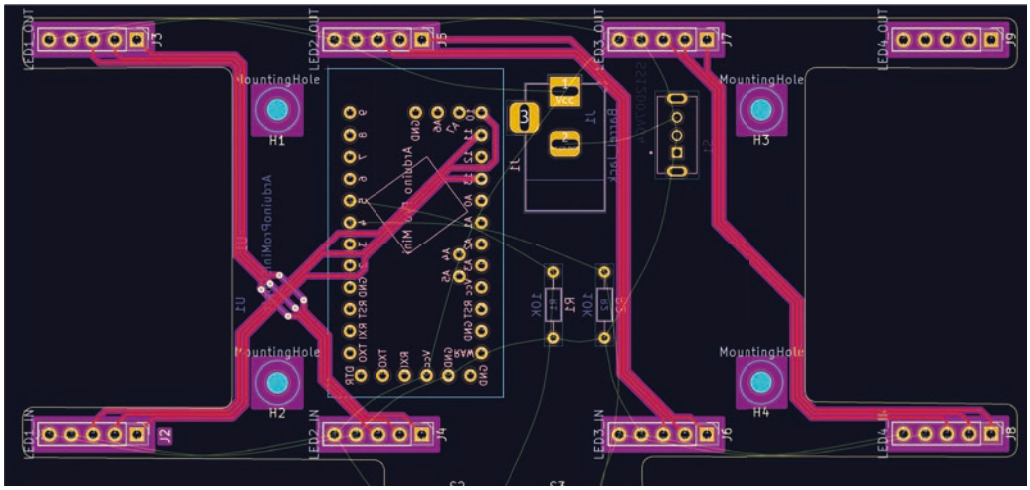
10. If you are satisfied with a route(s), lock it in place. The easiest way to do this is to select one or more segments of a track, then right-click to show the contextual menu and click "Select -> Select All Tracks in Net". Then, type "L" to lock these tracks. Locked tracks have a purple "glow" around them, similar to locked footprints.
11. Use common sense. There is no single correct way of routing a PCB.

Below, you can see the routing completed for the first three-pin headers:



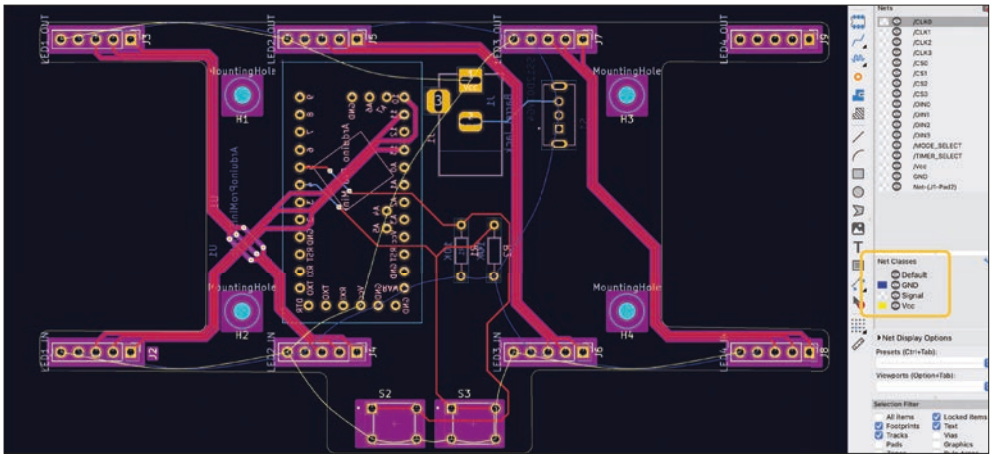
2.3.6.4: Completed routing for J2, J3, J4.

Below, you can see the state of the layout after I have routed all LED display pin headers. Notice that I have not routed the GND and Vcc pads:



2.3.6.5: Routing for LED display pin headers is complete.

Continue with the buttons, the resistors, the barrel connector and the on/off switch. Below, you can see the result of this process in my instance of the project:



2.3.6.6: Routing is complete, except for the GND and Vcc nets.

I have completed routing for all nets except for GND and Vcc, shown in the figure above. In the Appearance pane, you can toggle the GND and Vcc net class visibility to find any non-GND and non-Vcc nets yet to be routed. Turn visibility off for the two nets, and look for unconnected nets that white ratsnest lines would mark. In my example above, the remaining ratsnest lines belong to the GND and Vcc nets.

To finish the routing, you will create two copper fills. One will connect to the GND net and the other to the Vcc net.

2.3.7. 4 - Copper fills

Let's finish the routing by creating two copper fill zones.

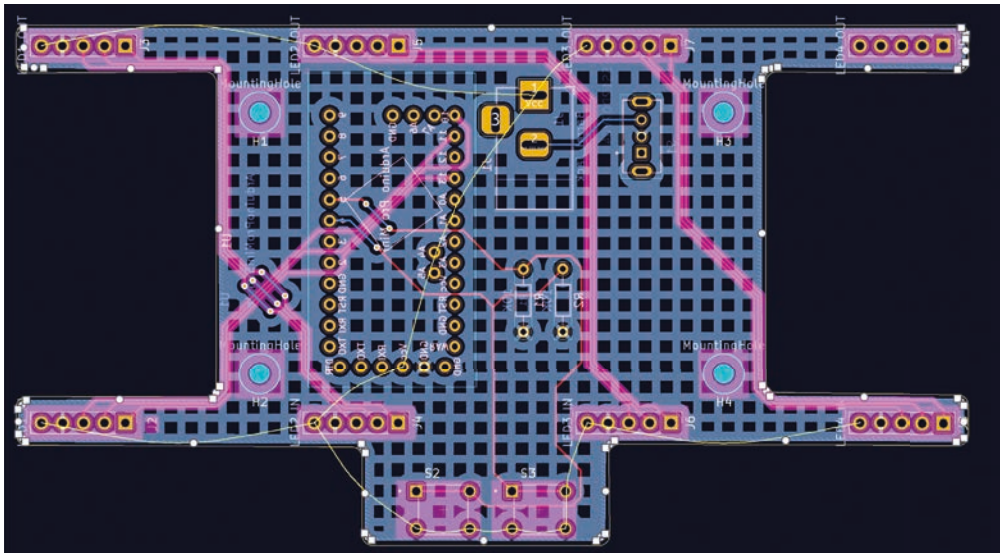
Create the first copper fill zone in the bottom copper layer and connect it to the GND net. This copper fill one will also connect all GND pads.

Create the second copper fill zone in the top copper layer and connect it to the Vcc net. This copper fill one will also connect all Vcc pads.

Activate the bottom copper layer from the layer chooser, and click the copper fills button from the right toolbar. Set the grid to 0.2540 mm, and click in the top right corner of the PCB outline. In the window that appears, make these selections:

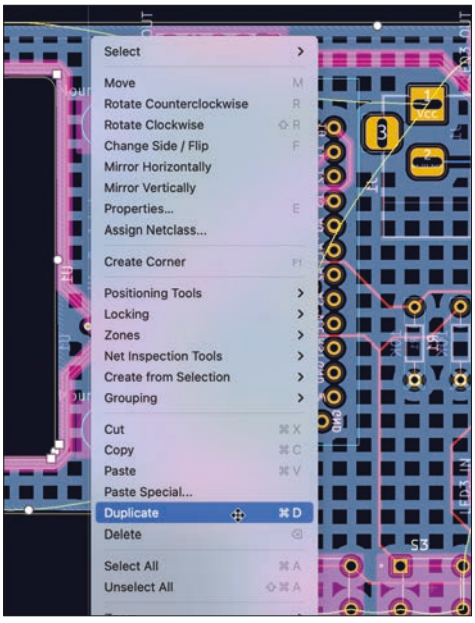
- Layer: "B.Cu."
- Net: GND.
- Fill, Fill type: Hatch pattern.

Click OK and start drawing the fill zone close to the outline. See the result below:



2.3.7.1: Bottom copper fill connected to GND.

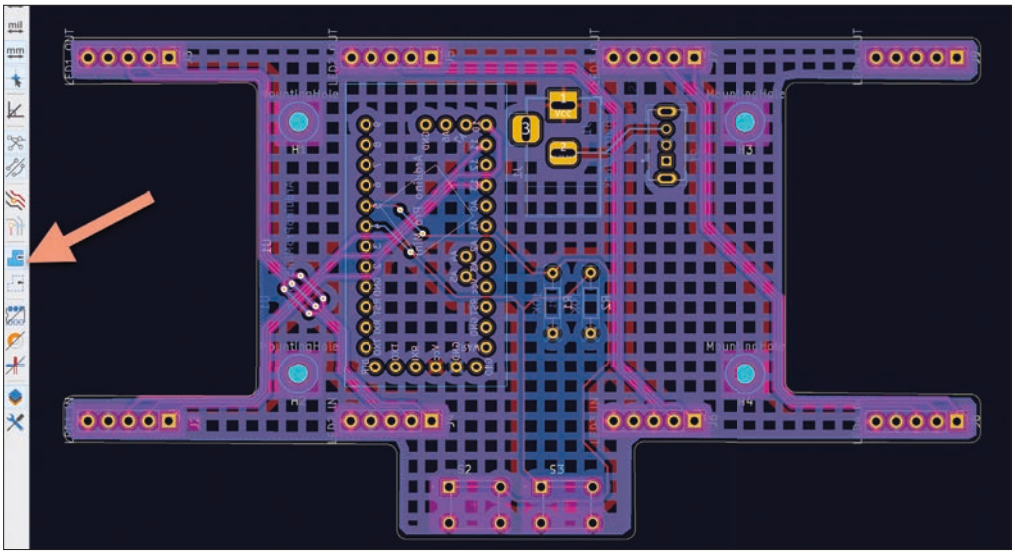
You can repeat the process I outlined above for the top copper layer, or you can duplicate the existing copper fill and change the layer to "F.Cu" and the net to Vcc for the new zone. Duplication will save you the time of drawing the multi-point polygon. To duplicate a zone, right-click on it and click "Duplicate" from the context menu.



2.3.7.2: Duplicating a filled zone.

When the new zone is created, it will have the same shape as the original zone. If you place it exactly over the original zone, it may not be easy to select, so offset it by a few pixels. Double-click the new zone to bring up its properties window, and change the Layer to "F.Cu" and the net to Vcc. Move the new copper fill zone over the original so that they overlap. Finally, right-click on the edge of the zones to bring up its context menu, and click "Fill All" from the Zones submenu.

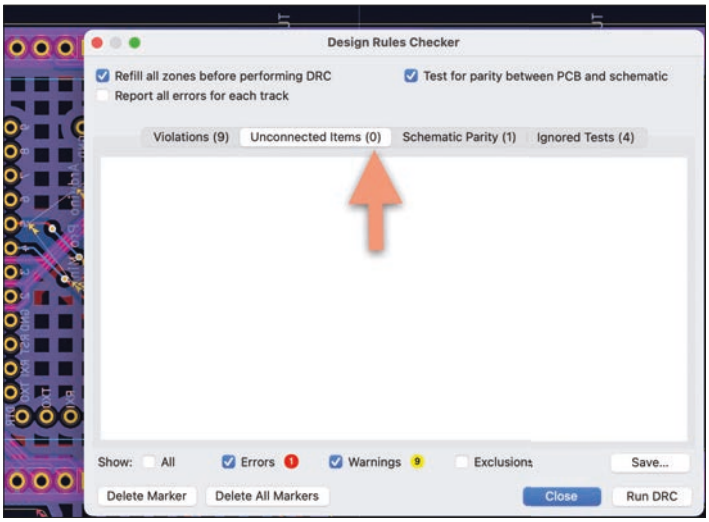
Here is the result:



2.3.7.3: Copper-filled zones are complete.

In the example above, I have clicked on the filled zones button mode in the left toolbar to depict the zones filled with their respective layer color.

A quick DRC shows no unconnected items.



2.3.7.4: No unconnected items.

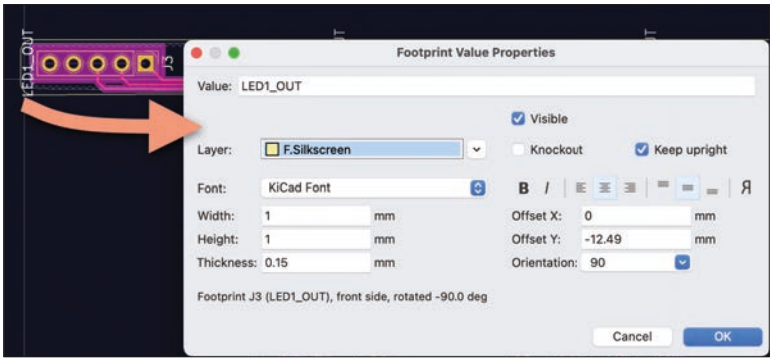
This means the copper fills completed the connections between the GND and Vcc pads. You can now continue with step five of the workflow, the silkscreen.

2.3.8. 5 - Silkscreen

In this segment, you will add silkscreen graphics and text to the top and bottom silkscreen. This work will complete step five of the layout design workflow.

Start with adding or editing the labels for the LED matrix display pin headers. These footprints already have appropriate labels but exist in the F.Fab layer. Change their layer to F.Silkscreen via their properties window, preferably in bulk using the Edit Text and Graphics Properties from the Edit menu.

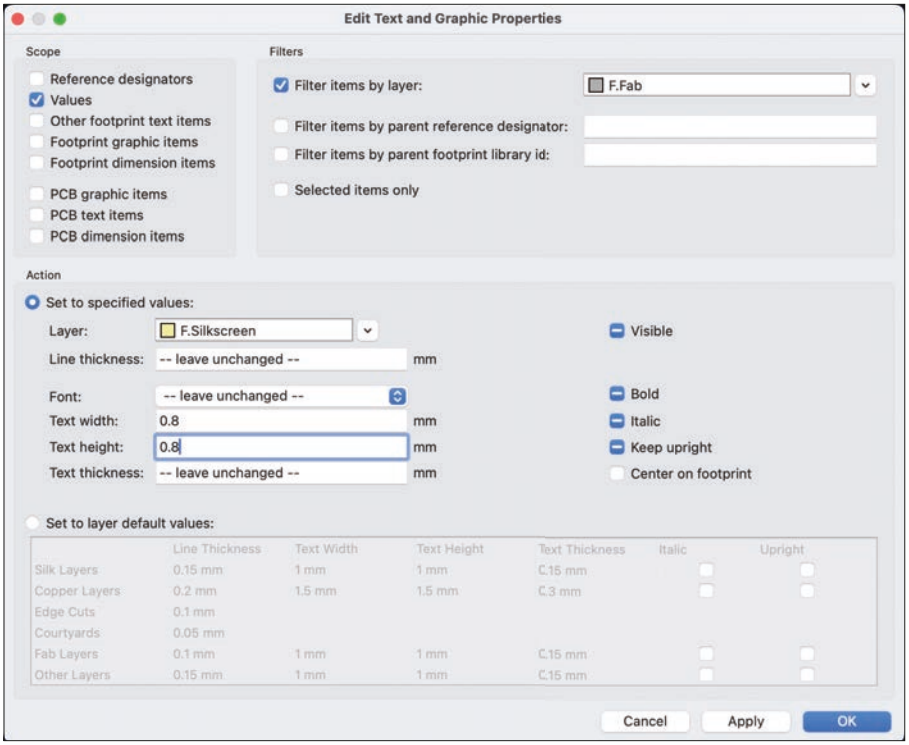
Below, you can see the text label attached to J3, which by default exists in F.Fab:



2.3.8.1: The text label for J3 exists in F.Fab. Change it to F.Silkscreen.

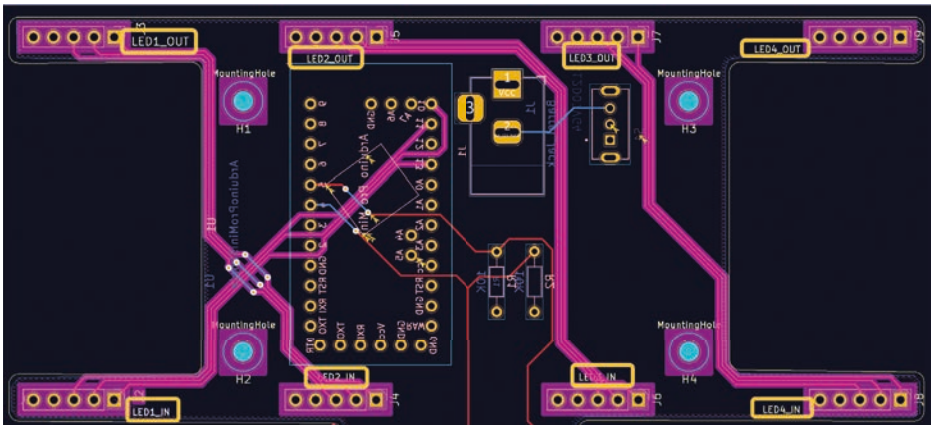
Bring up the Edit Text and Graphs Properties, and set these options (if I don't mention a value below, leave it at the default):

- Scope: Values.
- Filter items by layer: F.Fab.
- Set to specified values:
  - Layer: F.Silkscreen.
  - Line thickness: 0.1 mm
  - Text height: 0.8 mm



2.3.8.2: Moving text labels to F.Silkscreen and changing text attributes.

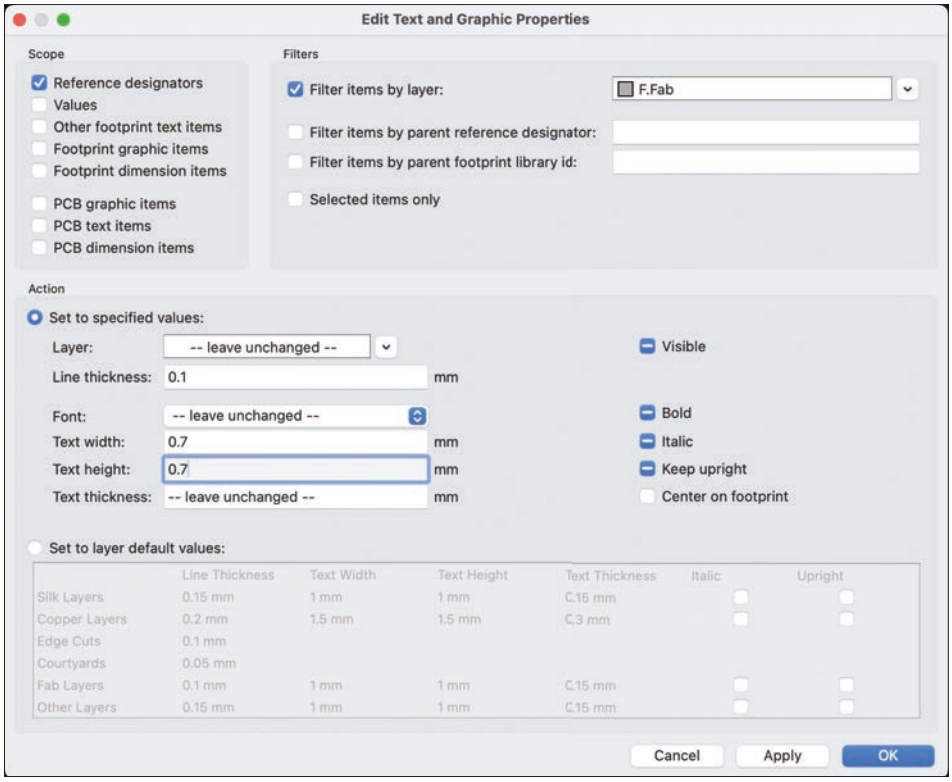
Click OK to close the properties window. The text labels for the pin headers are now in the F.Silkscreen layer. Move them into position above or below the pin header footprints (below, see the text labels in the yellow boxes):



2.3.8.3: Text label for the pin headers in F.Silkscreen.

Continue with the reference designators for all footprints. These items are already in the silkscreen (front or back, depending on the location of the footprint), but I'd like to change their size (text width and height). Again, use the Edit Text and Graphics Properties window to change those attributes for all reference designator text labels quickly. Bring up the Edit Text and Graphics Properties and use these settings:

- Scope: Reference designators.
- Action:
  - Line thickness: 0.1 mm
  - Text width: 0.7 mm
  - Text height: 0.7 mm



2.3.8.4: Changing text attributes for reference designator text labels.

Click OK. Review the reference designator text labels and move them to appropriate positions. Most of these labels were in good positions, though I had to move some because they were placed outside the board outline.

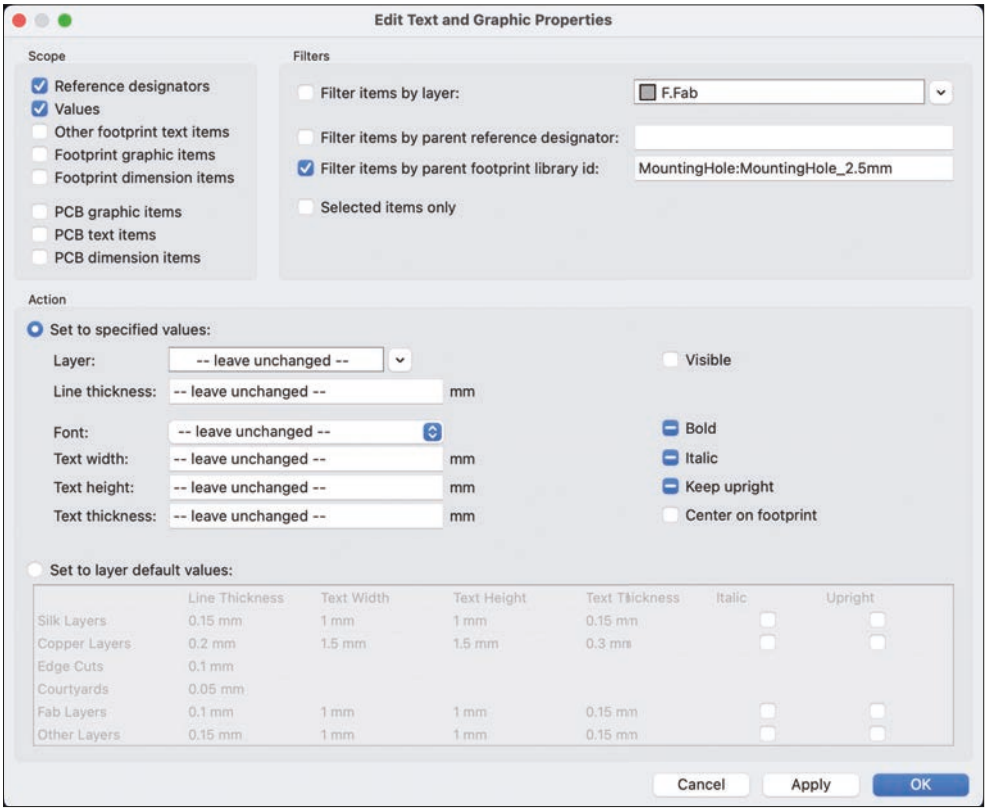
Several text labels already exist in the F.Silkscreen layer that I prefer to make invisible because they are not helpful. For example, four such labels have the text "Mounting Hole." The mounting hole footprints have reference designators H1, H2, H3, and H4, which I also want to make invisible. Those footprints share the same library link, "MountingHole:MountingHole_2.5mm". You can use this library link to make these labels not visible using the "Edit Text and Graphic Properties" window.

Open the Edit Text and Graphic Properties window and set these settings:

- Scope:
  - Reference designators.
  - Values.

- Filters:
  - Filter items by parent footprint library id:
    MountingHole: MountingHole_2.5mm

- Action:
- Visible: uncheck.



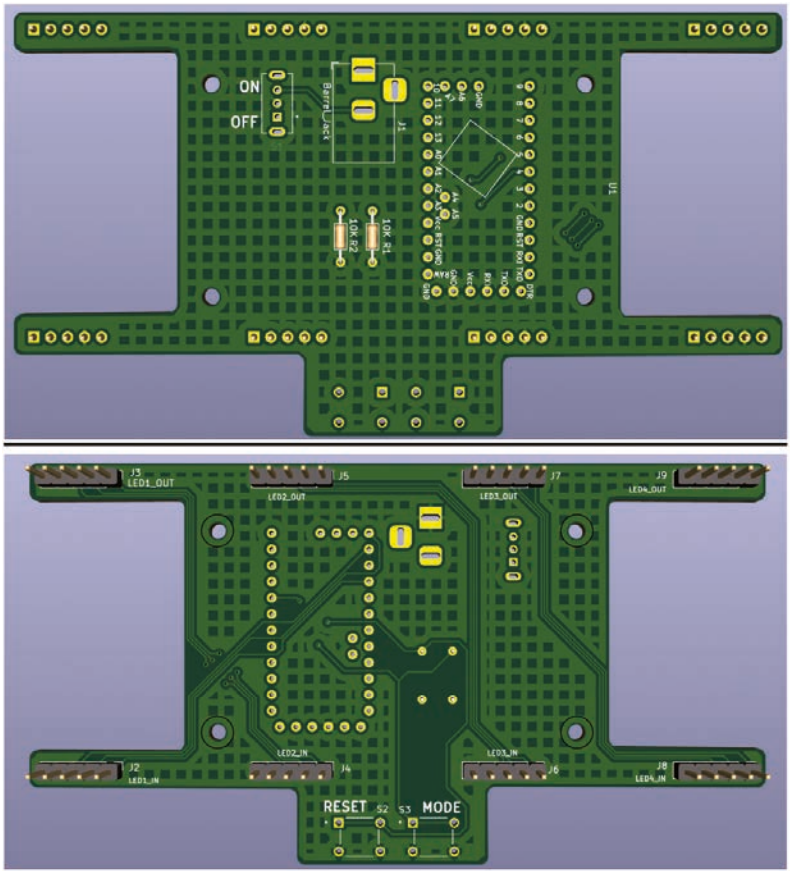
2.3.8.5: Making all mounting hole values and reference designators invisible.

Here is a list of other silkscreen work that I suggest you do at this time:

1. Move reference designators to positions that do not overlap with other silkscreen elements outside the footprint's outer boundary to be visible even when the board is assembled.
2. For footprints S2 and S3, make the value text labels invisible.
3. Add two text labels next to S2 and S3: Reset (over S2) and Mode (over S3).
4. Add two text labels for the on/off switch: "On" (close to pin 3) and "Off" (close to pin 1). Notice the on/off switch footprint in the bottom copper layer. Therefore, the new text labels should be in the B.Silkscreen layer, with the Mirrored checkbox selected.
5. For the resistors, change the layer for the values to B.Silkscreen.
6. move the footprint text property ("Arduino Pro Mini") outside the footprint's perimeter for the Arduino footprint.
7. make the value name ("ArduinoProMiniSimple ") invisible for the Arduino footprint.

8. For the Barrel Jack, J1, change the layer of the value attribute ("Barrel_Jack") to B.Silkscreen.

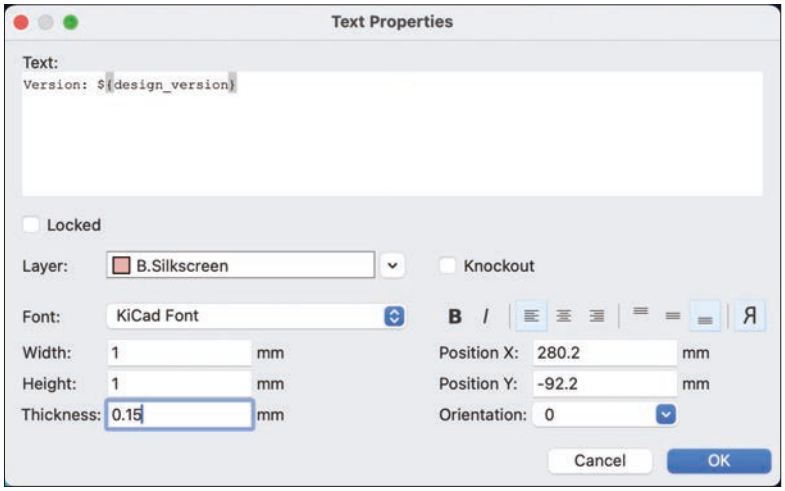
At this point, the back and front of the board look like this:



2.3.8.6: Inspecting the front and back silkscreen in 3D.

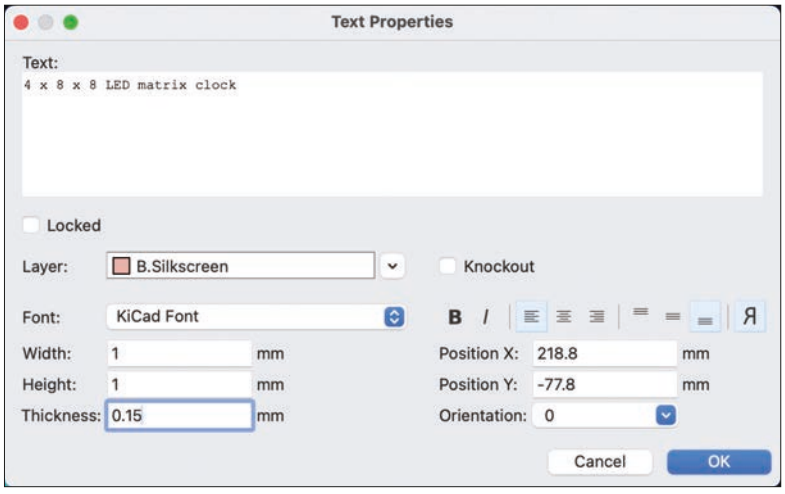
The front and back silkscreen look good. I will complete this work by adding the final silkscreen elements in the back silkscreen:

1. Power information (input voltage and connector polarity.
2. The Kicad logo. Use the footprint from the Symbol library with the name "KiCad-Logo2_5mm_Silkscreen". In the footprint properties, select "Back" from the Side dropdown.
3. The Tech Explorations logo.
4. A text label with information about the project. I will use the project version from the text variable named "design_version" I created earlier in the project (see below).



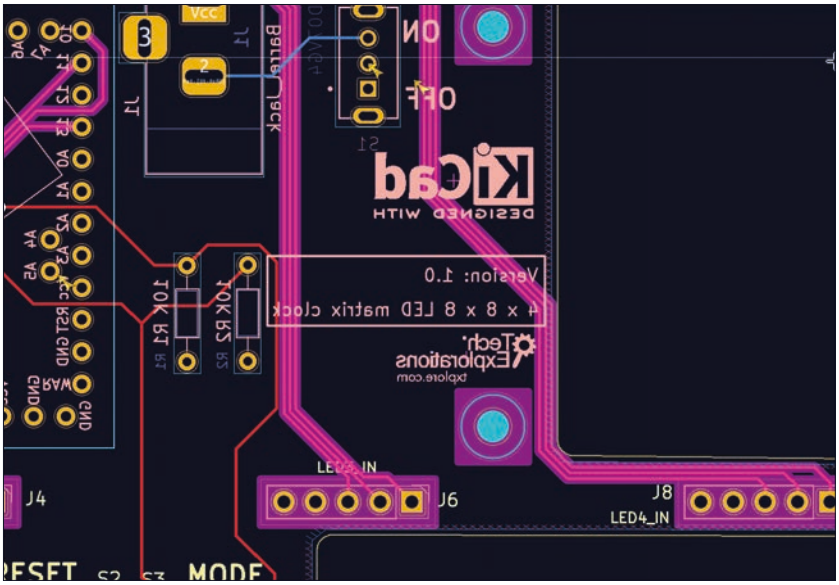
2.3.8.7: Using a text variable in a text label.

5. A text label that contains the name of the project and board, with a box around it:



2.3.8.8: A text label that contains the name of the board and project.

Here is the current state of the silkscreen:



2.3.8.9: Silkscreen work is complete.

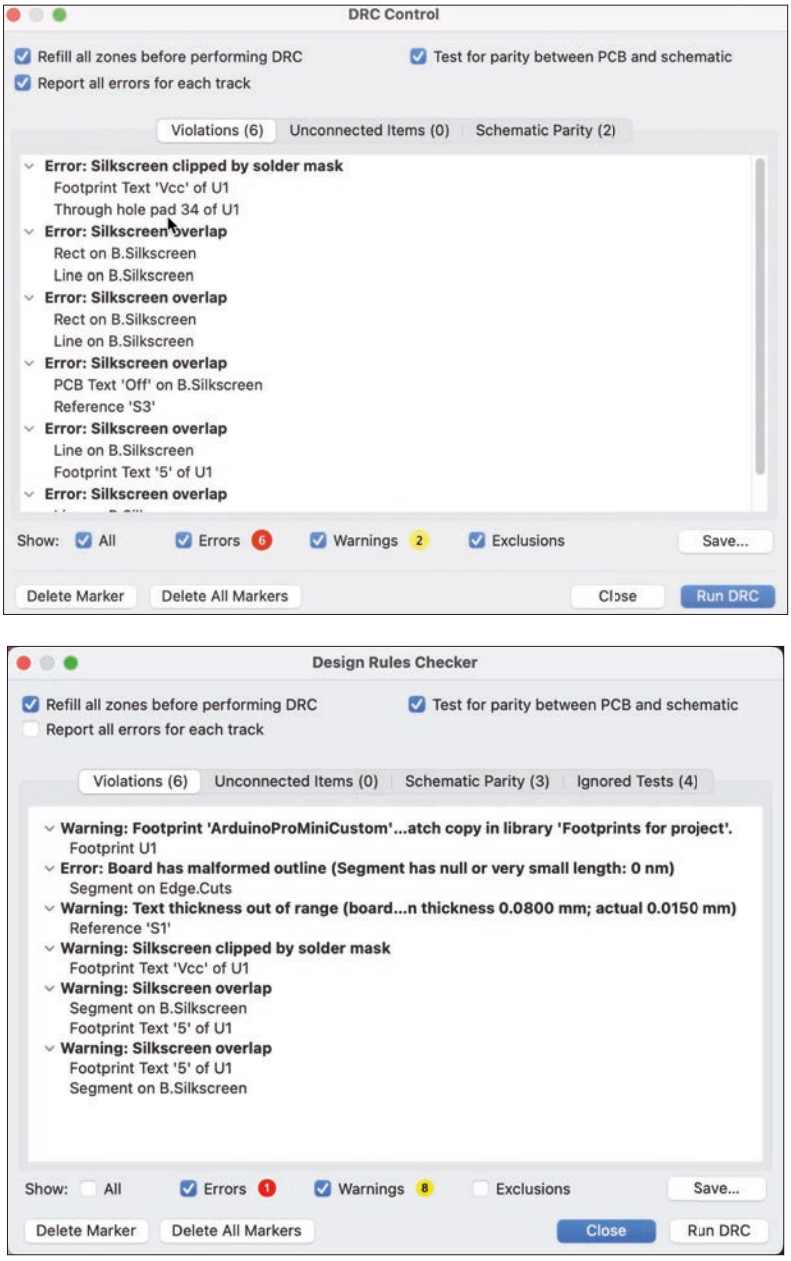
The silkscreen's current state may need to be corrected. For example, the silkscreen lines that mark the perimeter of the barrel jack footprint overlap with the box around the project name and version. The DRC will list this overlap as a rule violation. The power input information is also missing and will be added in the next lecture.

Instead of making these corrections now, I will proceed to the next step in the workflow, where I will run the DRC and correct each violation listed by the checker.

2.3.9. 6 - Design Rules Check

You've done a lot of work on this project and are close to completion. Are there any defects? Let's run a final DRC to find out.

Bring up the DRC tool and click on Run DRC. The results are below:



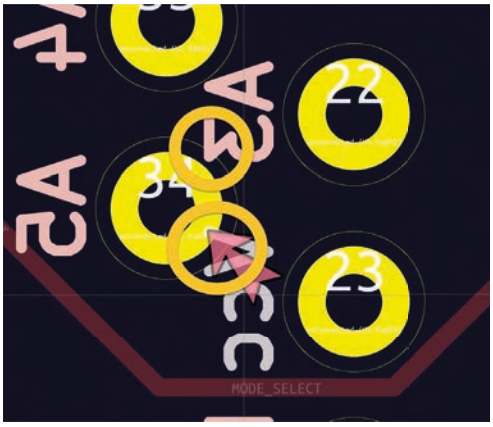
2.3.9.1: Several DRC errors relate to problems with silkscreen items.

Several DRC errors relate to problems with silkscreen items (overlapping and clipping). I have captured a common silkscreen problem in the figure below:

2.3.9.2: A common silkscreen error: two silkscreen items overlapping.

In this example, the problem is with a text label ("5") overlapping two graphic lines. These silkscreen items belong to the Arduino Pro Mini footprint. I made this error when I designed this footprint but did not discover it until I ran the DRC as part of this project.

Below is another common problem:



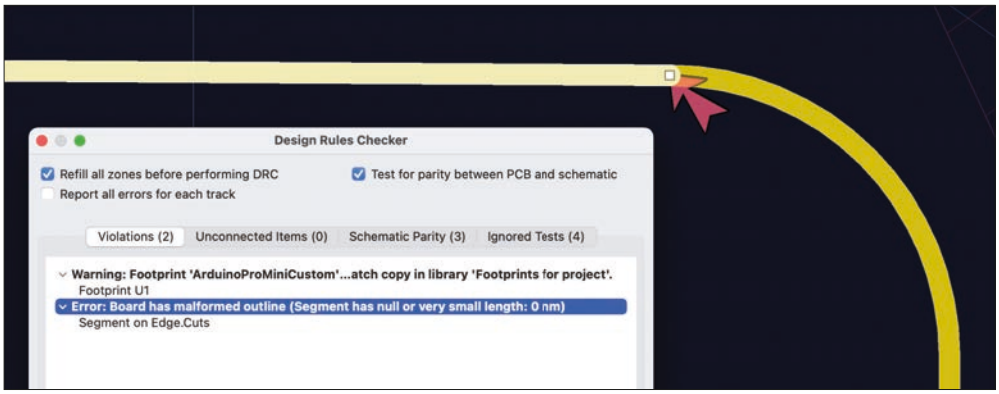
2.3.9.3: A common silkscreen error: silkscreen clipped by solder mask.

In this case, two text items in the silkscreen are inside the solder mask boundary of pad 34.

If these issues appear in the layout, you can move the silkscreen items to new positions so they don't creep in a pad's solder mask or overlap with another. However, because these errors occur within my custom Arduino Pro Mini footprint, you will need to use the footprint editor to correct the errors in the footprint. I have corrected both issues in this project and continued with other items on the list.

Another issue is that the text for "S1" is too thin. The error message for this issue is "Warning: Text thickness out of range". I corrected this by changing the text thickness to 1.15 mm.

Next in my list is an error: "Board has malformed outline". See below:

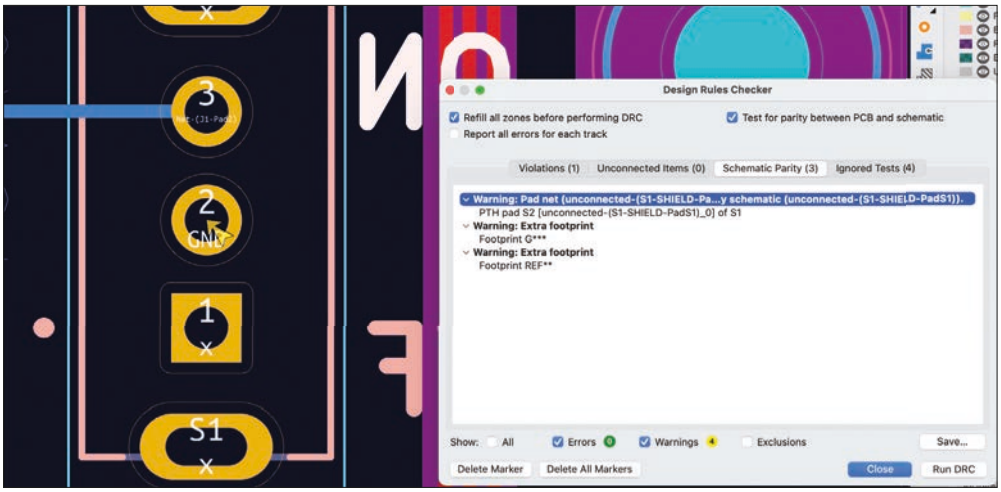


2.3.9.4: An error with the outline.

To fix this error, I moved the two straight lines and the arc and found a stray tiny line segment hiding underneath. I deleted that segment and re-drew the rounded corner.

The last item is a warning about the Arduino Pro Mini footprint not matching the library. This is because I made changes to this footprint outside the library. I will safely ignore this warning and continue.

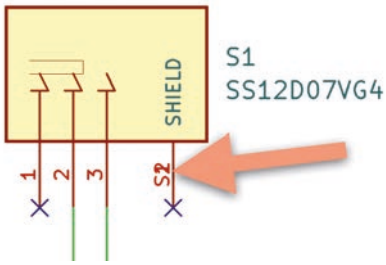
In the Schematic Parity tab, there are three issues:



2.3.9.5: Three issues in the Schematic Parity tab.

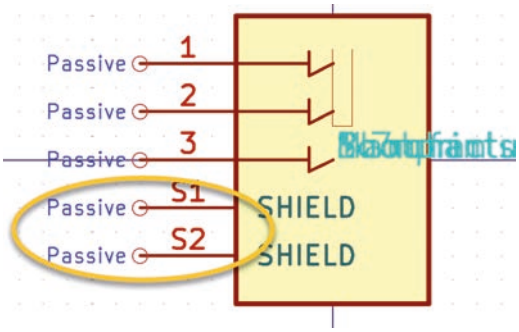
There are two warnings about extra footprints. These are the two logos, and I can ignore them.

The first warning, however, is about an unconnected pad, which can be confusing. The root cause of this error is that the symbol has a pin that is overlapping with another pin. See the symbol below, and notice the two overlapping pins marked with the arrow:



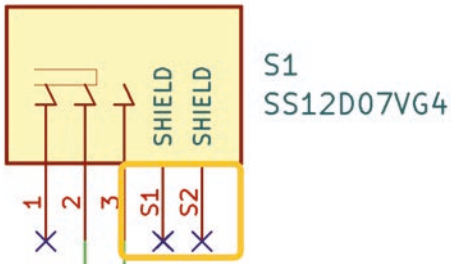
2.3.9.6: The switch symbol has two overlapping pins.

I did not notice this issue when I worked on the schematic design, but it isn't late. Go to the schematic editor, right-click on S1, and select "Edit with Symbol Editor." In the symbol editor, click on the overlapping pins and drag to separate them, as in the screenshot below:



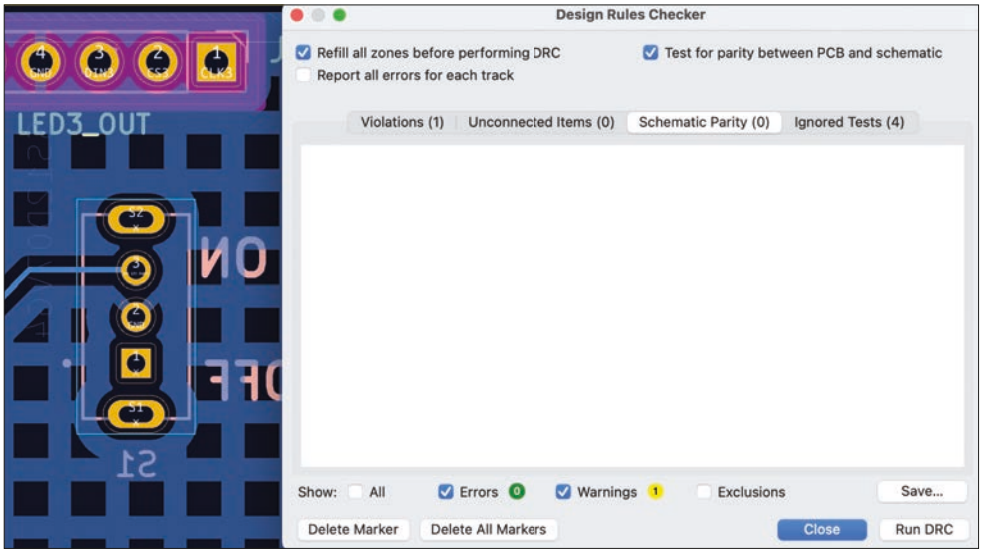
2.3.9.7: Seperated S1 from S2.

Save the symbol, close the symbol editor, and return to the schematic editor. You should now see that S1 and S2 are separated, but one does not have the "not connected" symbol attached. Add the "not-connected" symbol so that S1 looks like this:



2.3.9.8: S1 and S2 pins of the switch are marked as "not connect".

Save the schematic and return it to the PCB editor. Click the "Update PCB from Schematic" button in the top toolbar, update the PCB, and then repeat the DRC. It should come up clear (aside from the U1 footprint warning):



2.3.9.9: Cleared of items requiring action.

The DRC shows no more errors. There are a couple of schematic parity warnings that you can safely ignore. These are caused by the KiCad and Tech Explorations logos, which exist as footprints in the layout but have no symbolic counterpart in the schematic editor.

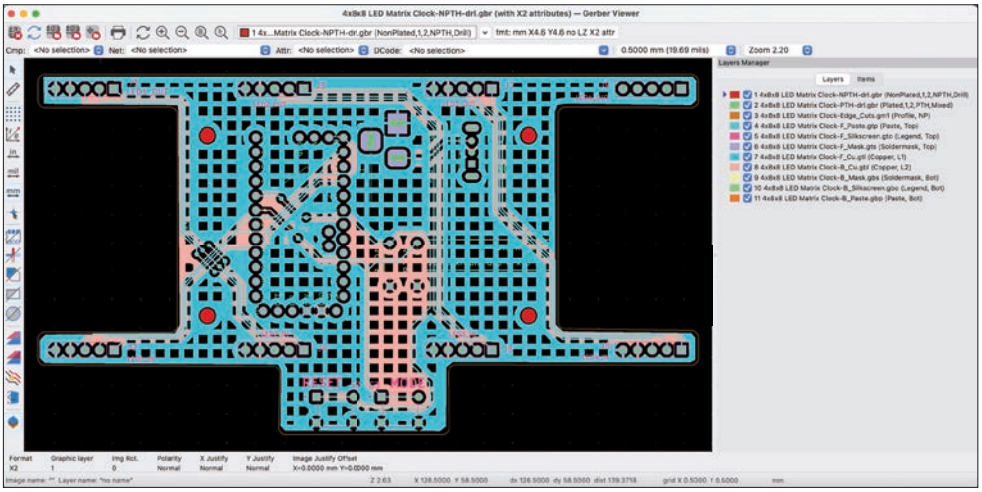
Step six of the workflow is complete. Let's continue with the last step, where I export and upload the Gerber files for manufacturing.

2.3.10. 7 - Manufacture

Let's complete the project by exporting the Gerber files, using the Gerber Viewer app to inspect them, and uploading them to the manufacturer's website.

In the PCB editor, click the plotter button (top toolbar) to open the Plot window. Set the output directory, check the layers to include in the Gerber file export, and generate the Gerber layer and drill files. If you don't remember how, please refer to the chapter "How to export and test Gerber files" in My book Fundamentals and Projects.

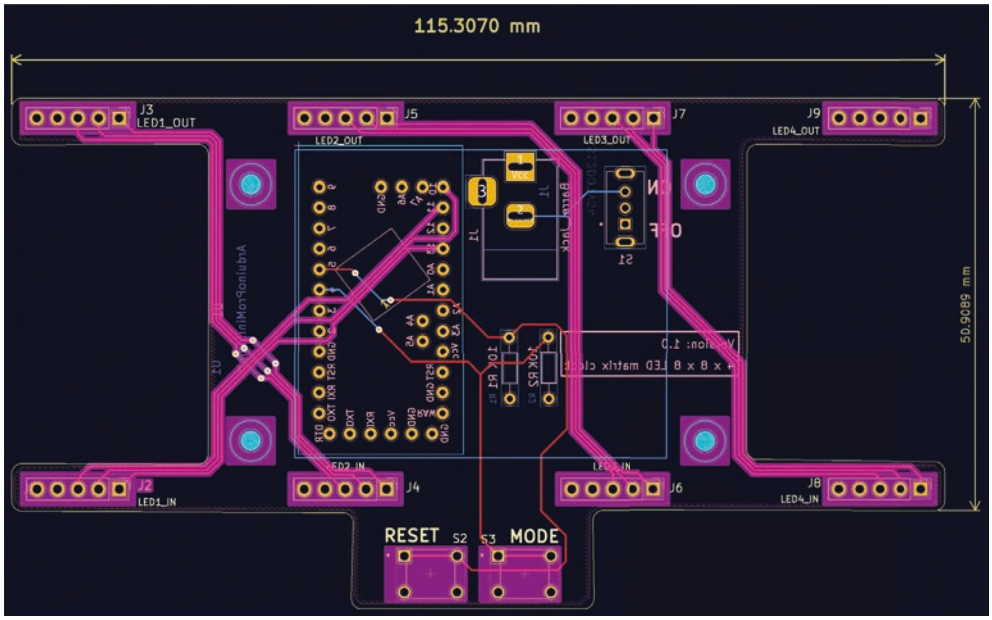
Once you generate the Gerber files, use the Gerber viewer to evaluate the fitness of these files for manufacturing. My Gerber viewer instance looks like this:



2.3.10.1: Examining the exported Gerber files in Gerber Viewer.

Patiently inspect each layer separately to ensure that it is correct. Be particularly careful with text items in the silkscreen, looking for misspellings and typos.

When you finish reviewing the board in Gerber Viewer, return to the PCB editor to measure the board's dimensions. Some manufacturers require you to type those values in the order form. I have used the measurement tool from the right tool mark to mark the width and height in one of the user layers.



2.3.10.2: The final PCB with measurements.

Create a ZIP archive with the Gerber files and upload this file to your preferred manufacturer's website.

The project is now complete. However, there are three more important chapters in this part of the book. In the next chapter, you will learn to add 3D shapes for the switch and barrel connector footprints. Then, a chapter shows how to fix a design bug that I found in the PCB. This bug is something that the DRC cannot detect, and I only discovered it after testing the manufactured prototype board. The last chapter contains photographs of the manufactured and assembled PCB.

Please beware: The Gerber settings I show on this page worked when I placed this specific order. Manufacturers may change their Gerber specifications from time to time. For this reason, you should check the manufacturer's website for the latest Gerber specifications. For example, here are pages that contain manufacturing information from these manufacturers:

- Oshpark: https://docs.oshpark.com/design-tools/kicad/
- NextPCB: https://www.nextpcb.com/helpcenter/files-explanation

2.5. Bonus - 3D shapes

The 3D viewer renders a 3D representation of your PCB. When footprints on the PCB have associated 3D shapes, the 3D viewer will also renter those shapes and give you a PCB visualization with the components to approximate a "real world" representation of the PCB.

Without any footprint and 3D shape associations, the 3D representation of the board of this project looks like this:

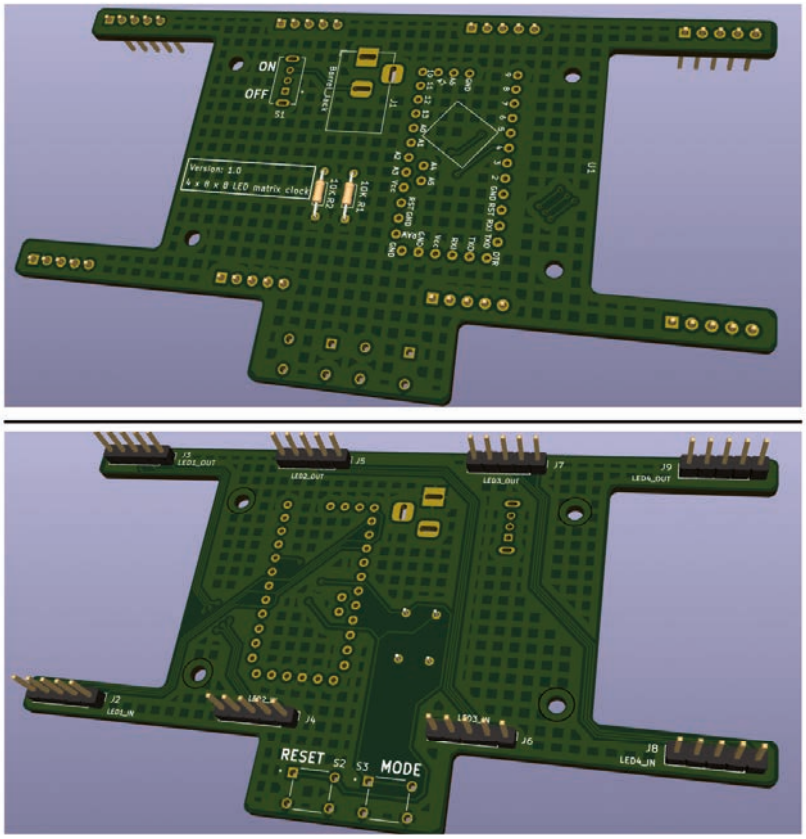


Figure 2.5.1: The 3D viewer showing only the core PCB elements.

Once you associate footprints with appropriate 3D shapes, the 3D-rendered board will look like this:

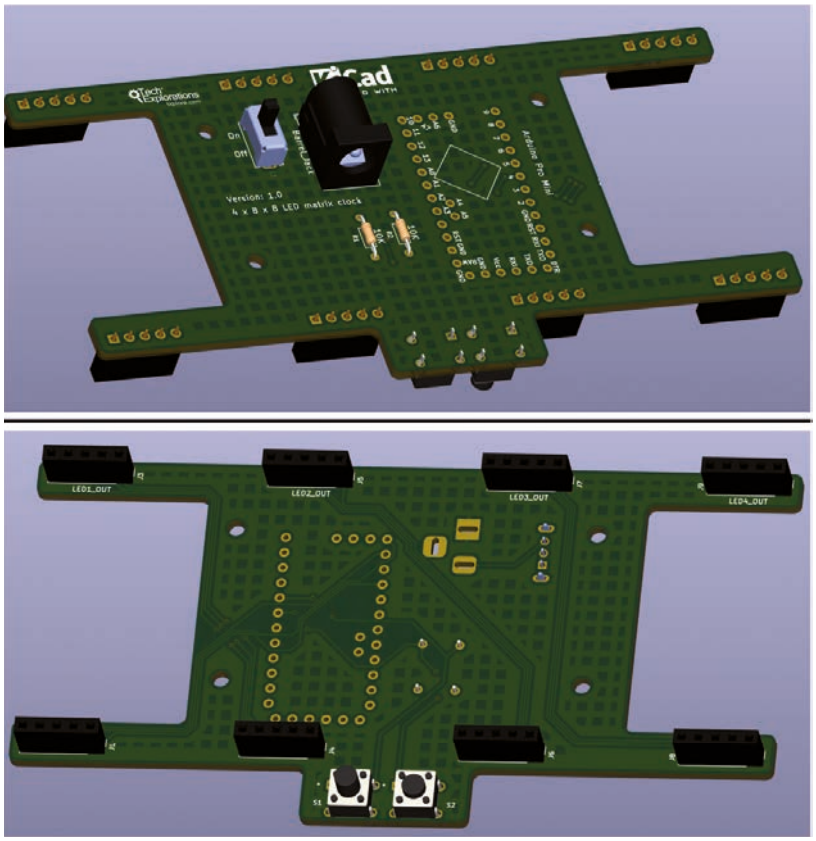


Figure 2.5.2: The 3D viewer showing the PCB with several 3D shapes.

In this chapter, you will learn how to find a 3D shape for the slide switch and associate it with the relevant footprint so that the 3D viewer includes it in the rendering of the PCB.

Please also refer to a chapter dedicated to the 3D viewer in My book Fundamentals and Projects.

I am looking for the slide switch's 3D shape, with reference designator S3. Open this footprint's properties window and click on the 3D Models tab.

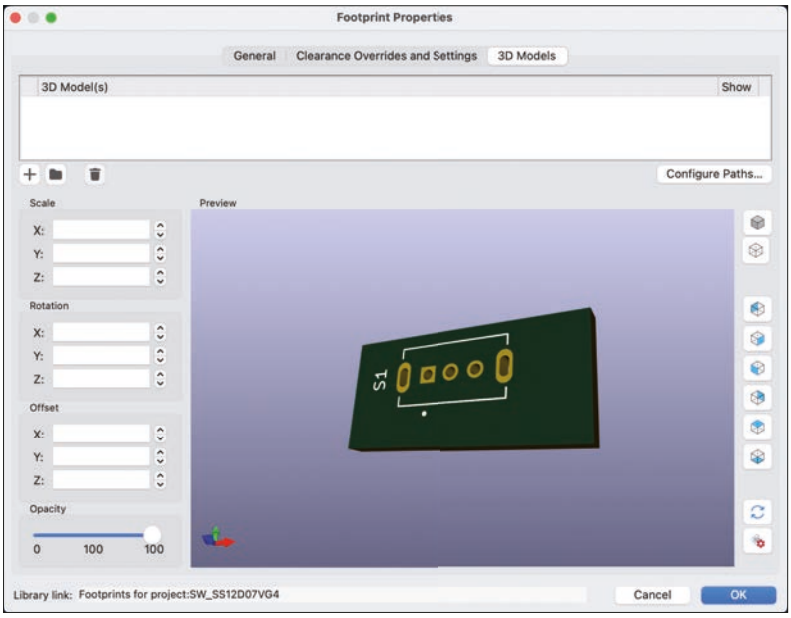


Figure 2.5.3: This footprint has no 3D shape associated.

The 3D Models table is empty. This means that there is no 3D shape associated with this footprint. Let's find one and add it to this table. The model of this slide switch is "SS12D07VG4". You can find this in the Value property of the General tab. I found this footprint in Snapeda, so I will use Snaped again to look for a matching 3D shape. A matching 3D shape does not necessarily need the exact model code. The same shape can match more than one footprint. So, instead of searching only for "SS12D07VG4", you can search for a more general term like "slider switch." With a narrow search, you will find a matching shape quickly - if there is a match.

I have done a comprehensive search and found a matching 3D shape below. You can access this shape using this URL[5].

5 https://www.snapeda.com/parts/OS102011MS2QN1/C&K/view-part/391519/?ref=search&t=OS102011MS2QN1

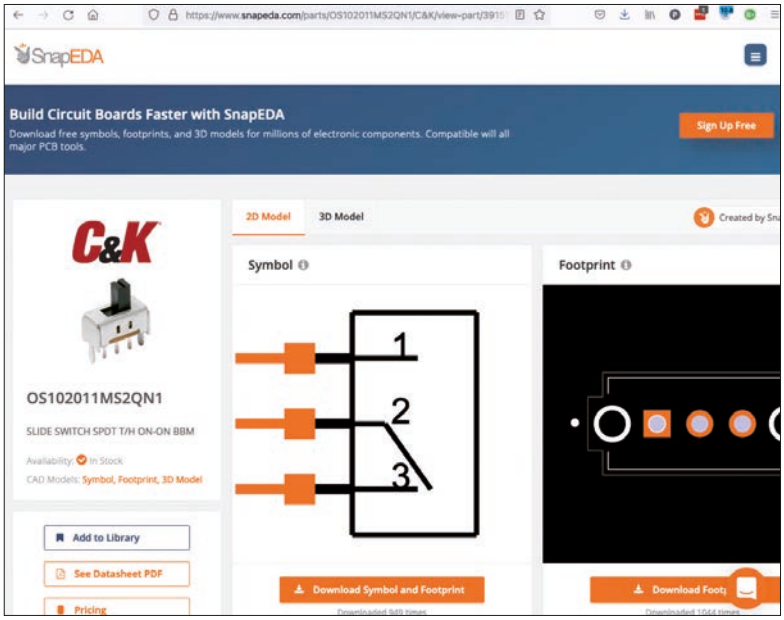


Figure 2.5.4: Found a matching 3D shape.

Download the 3D shape as I explain in the chapter "Finding and using a 3D shape for a footprint". Expand the ZIP file, and note the location of the file with the ".STEP" extension:

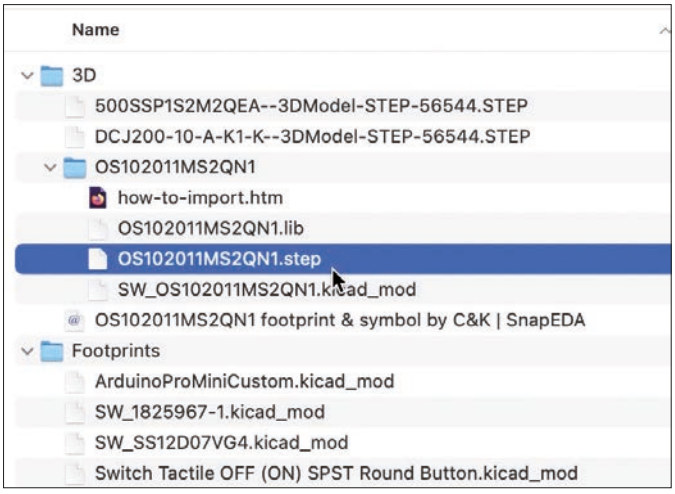


Figure 2.5.5: The 3D shape file ready to use.

Return to the PCB editor and open the slide switch footprint's properties window. Click the 3D Models tab. Under the empty 3D shapes table pane, click the "+" button to add a new row. Then click the folder button on the right side of the row to open the file browser. Navigate to the ".STEP" file's location and double-click on it to add it to the table. The «3D models» tab looks like this:

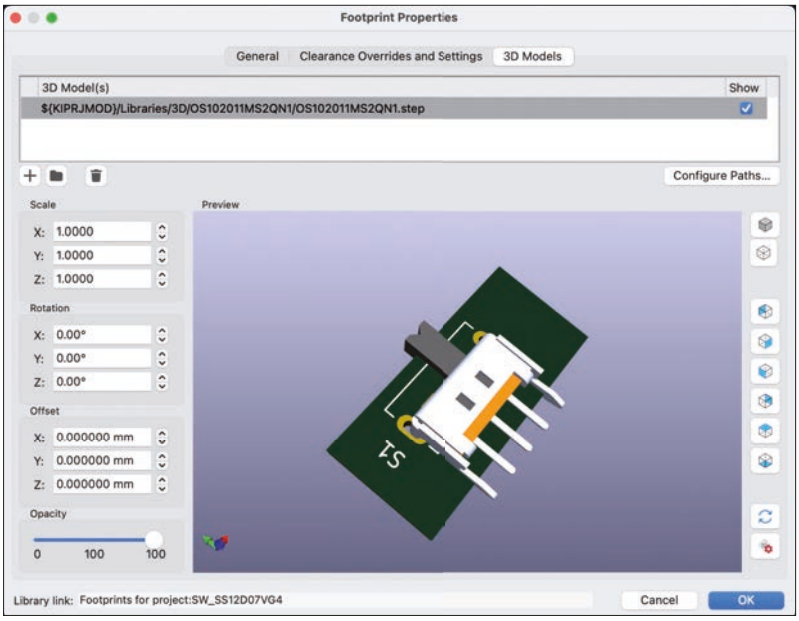


Figure 2.5.6: The 3D shape is associated with the footprint but is misaligned.

The 3D shape is now associated with the footprint but is misaligned. Use the Scale, Rotation, and Offset widgets to align the 3D shape and fit with the footprint. Here's mine after alignment:

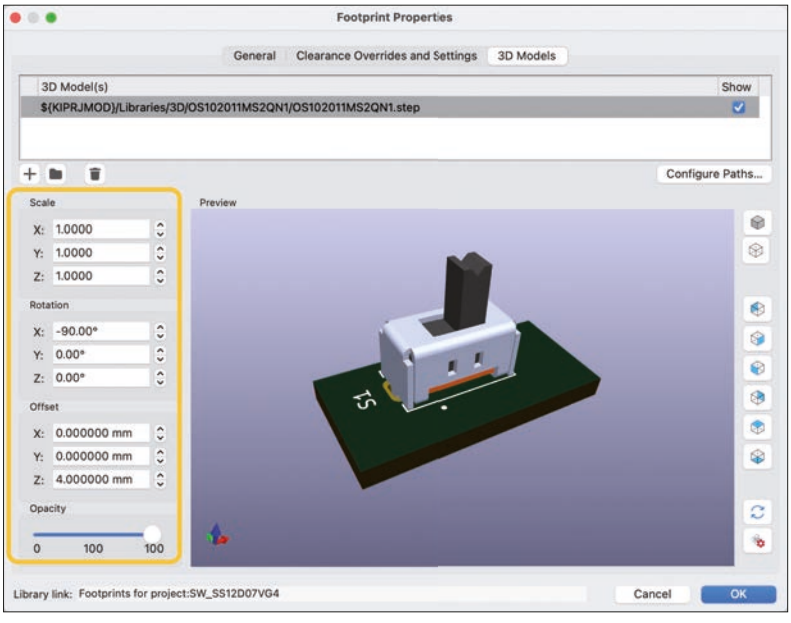


Figure 2.5.7: The 3D shape is associated with the footprint aligned.

Click OK to close the properties window and open the 3D viewer to check the result of this work.

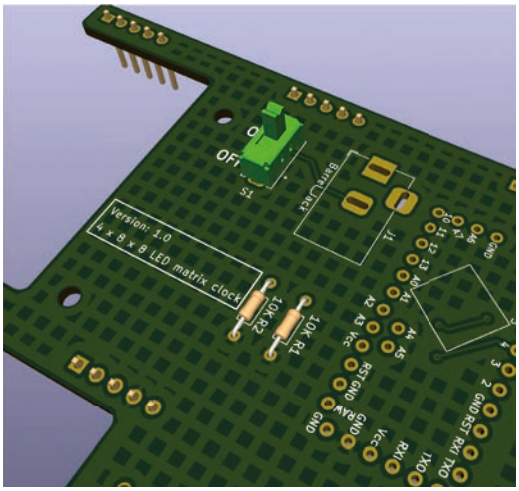


Figure 2.5.8: The new 3D shape is included in the board rendering.

The new 3D shape is now included in the board's rendering in the 3D viewer. Below you can see the slide switch in the assembled board:

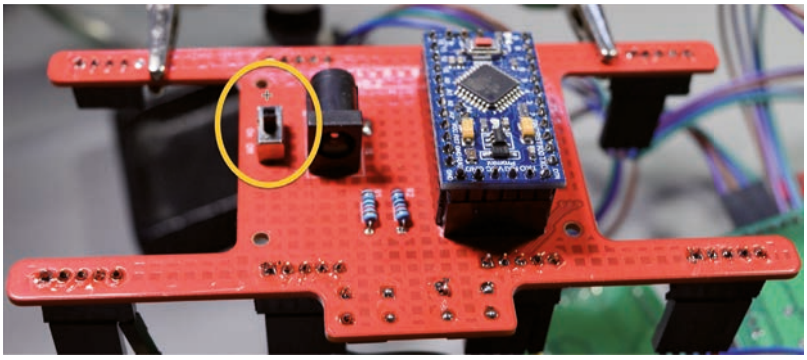


Figure 2.5.9: The slide switch in the assembled PCB.

Below, you can see the final 3D rendering of this PCB after including the power input information and replacing the LED display module connectors from the original male headers with female connectors.

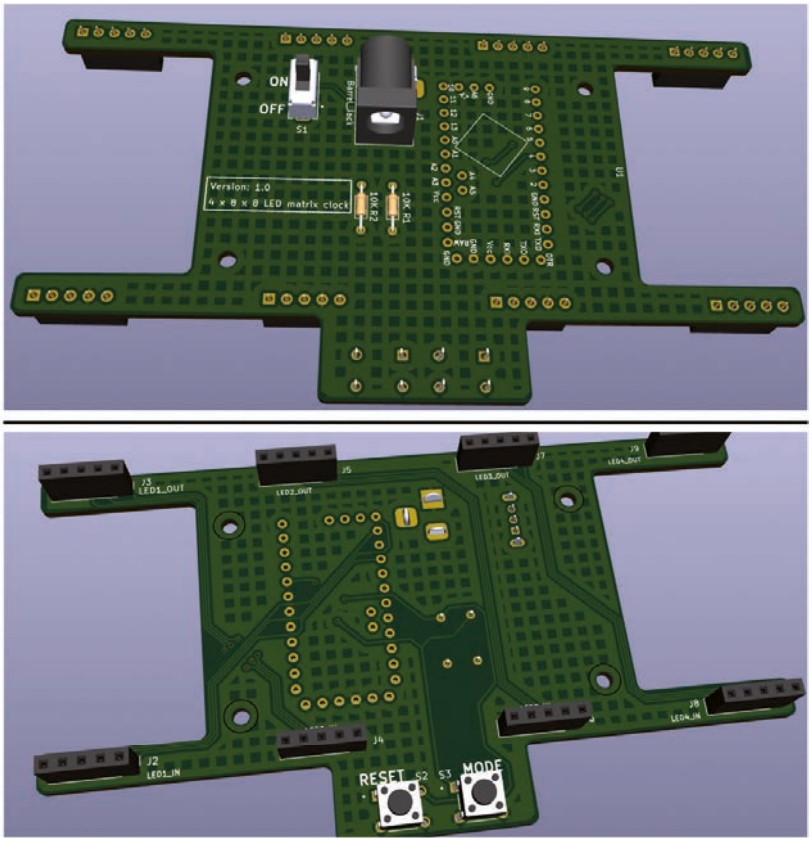


Figure 2.5.10: Final 3D rendering showing all silkscreen.

Finding and adding 3D shapes to your project helps to create a more realistic depiction of your board. However, to increase realism, you will need to expend additional effort. Is it worth it? In some cases, it does; in others, it does not. In general, when I share a project with others, it is worth increasing the realism of the 3D rendering. I rarely add 3D shapes for projects I don't share with others.

2.6. Bonus - Found a bug in the schematic! (and fix)

In an earlier version of this project, I made a mistake in the schematic, which I overlooked until much later in the process (after I had manufactured the first version of the PCB!). I wrote this chapter to show you how I corrected this error. What you will learn in this chapter will be very useful. No amount of care can guarantee 100% error-free PCBs. Further, many changes result from iterative improvements, not errors in the original design. In either case, you will want to return to the schematic to make one or more changes and then carry the changes across to the PCB. Here is how I updated my project to correct a bug.

Let's begin.

Please note that the symbol identifiers do not match the exact version of the project in this book. However, the general design is the same.

While wrapping up an early version of this project, I discovered a minor bug in the schematic. Specifically, I found that I had made a mistake in the net labels attached to some of the wires in the LED matrix display group of pin headers.

Luckily, the error was only in the net labels. The wiring was correct. In this chapter, I will show you how to correct this error in the schematic editor and update the PCB.

First, let's look at the error. See the figure below.

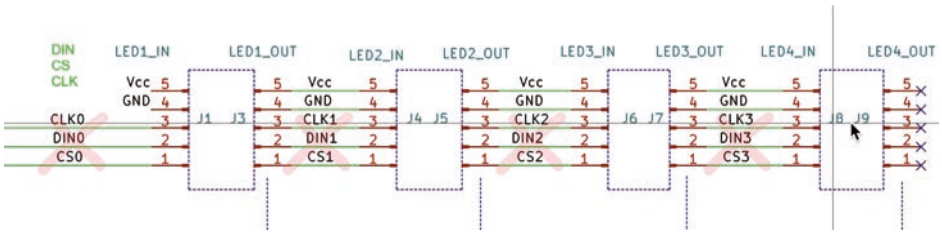


Figure 2.6.1: I have marked the errors with the red "X."

The net labels for pins 1, 2, and 3 are incorrect. They are marked with the red "X." The correct labels are "DINx," "CSx," and "CLKx," where "x" is 0, 1, 2, 3.

To fix this error, delete all incorrect labels and replace them with the correct ones. The corrected LED matrix display group is below:

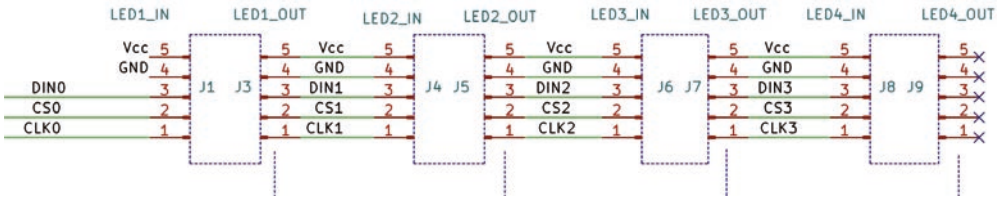


Figure 2.6.2: Net labels corrected.

The schematic has been corrected. To update the layout editor with the changes from the schematic, click the Update button from the top toolbar and click Update PCB.

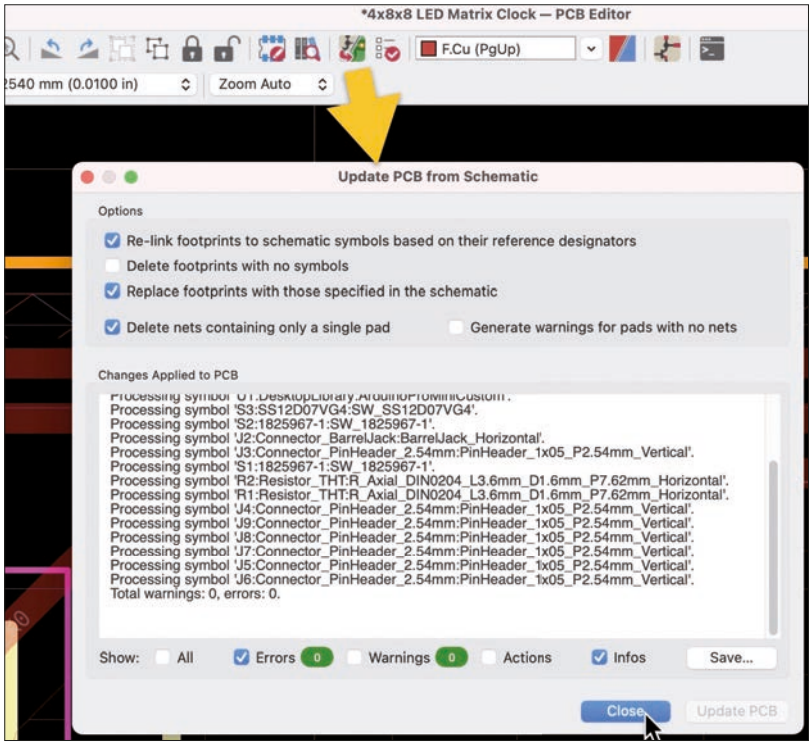


Figure 2.6.3: Updated PCB from schematic.

Click Close and return to the layout editor.

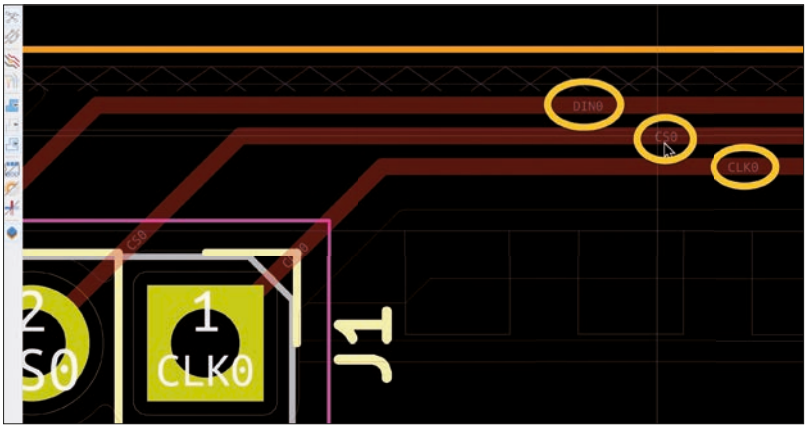


Figure 2.6.4: Confirming the changes in the PCB.

To confirm that the changes in the schematic update the layout, look at the tracks that come out of J1. For example, J1 pin 1 is connected to the CLK0 net. The track that is connected to this pin also belongs to CLK0. Similarly, in the schematic in Figure 2.6.2, pin 1 of J1 is connected to net CLK0.

Double-check with a few other pins and nets to confirm that the schematic and layout agree.

Because the tracks don't change, you don't need to re-export the Gerbers, and if you have already ordered a manufactured PCB, it will still work. This was an example of how to quickly fix an error that originates in the schematic but has no effect on the electrical characteristics of the PCB.

2.7. Assembled PCB

A couple of weeks after ordering the PCB for this project, it arrived in the mail. I did the assembly and testing, and I am glad to say that it worked. Below is a snapshot of the assembled PCB displaying the rolling word "Arduino":

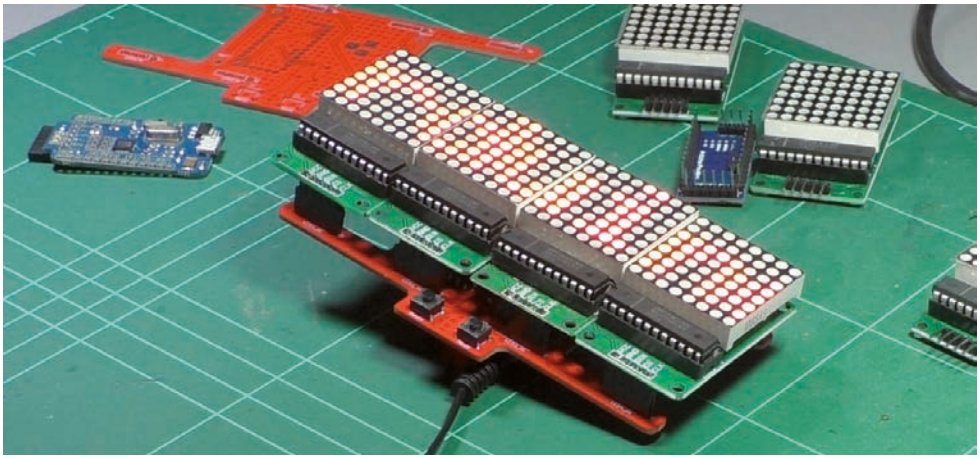


Figure 2.7.1: The assembled and working PCB.

The most challenging aspect of the assembly is related to the LED matrix displays. These displays contain the MAX7819 controller chip and come with 90-degree headers. Here is an example[6] from Amazon. I had to desolder the original headers and replace them with straight headers to plug into the female headers on the PCB.

6 https://amzn.to/3dtpDFH

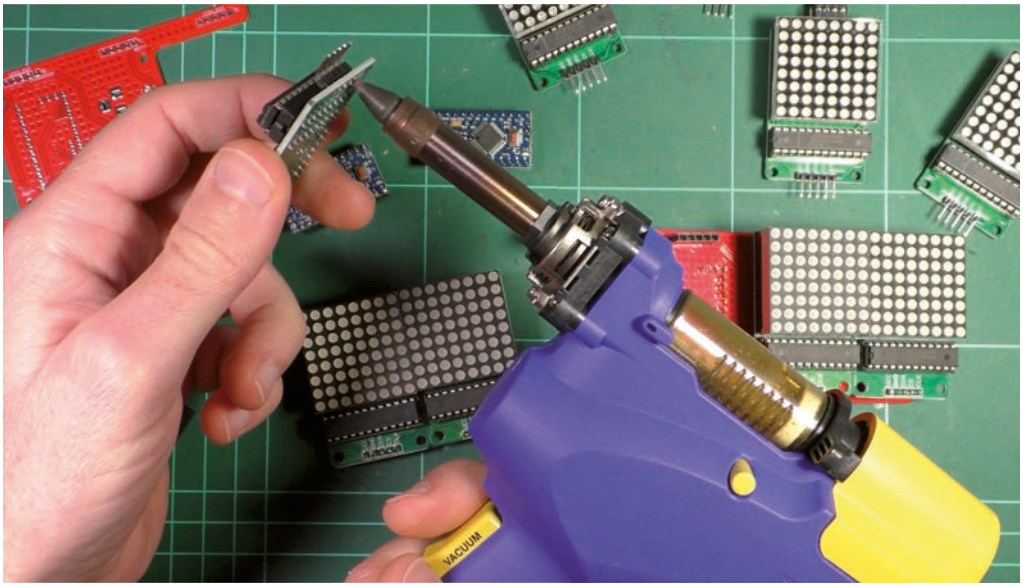


Figure 2.7.2: Desoldering the original headers from an LED matrix display.

Then, attach the displays to the board via the female headers:

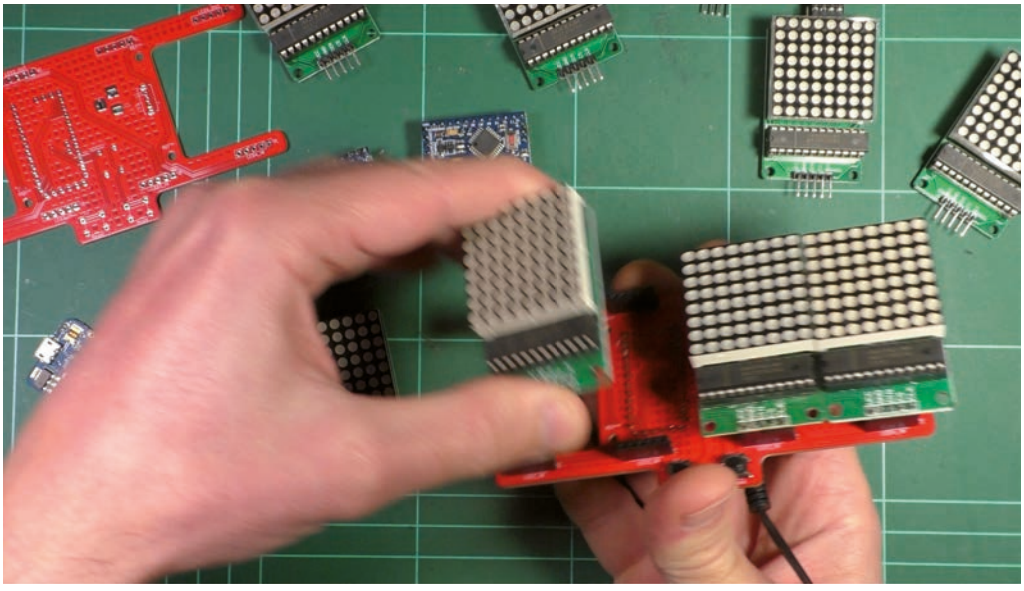


Figure 2.7.3: Attaching an LED matrix display to the board.

The back of the PCB contains the Arduino Pro Mini, barrel connector, slide switch, and two resistors.

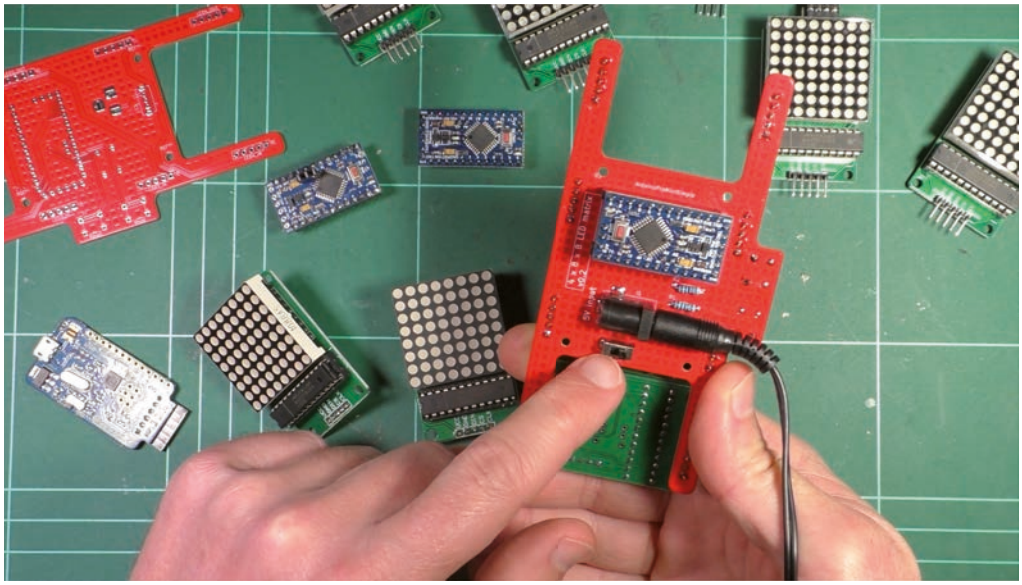


Figure 2.7.4: The back of the PCB.

For the Arduino Pro Mini, I used female headers to detach the Arduino for programming. The Arduino Pro Mini does not contain a programming USB interface, so I used an external USB to UART module for programming.

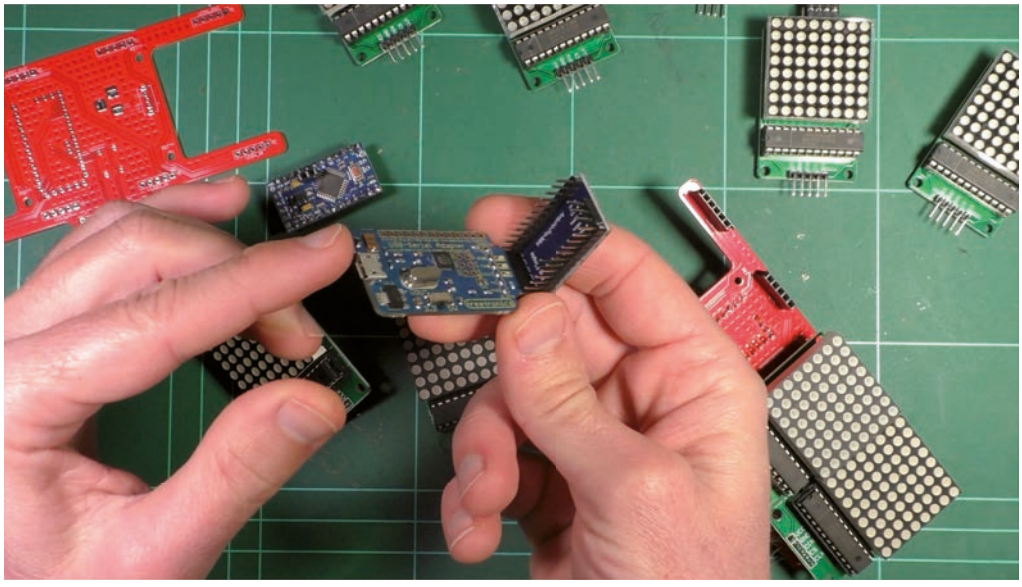


Figure 2.7.5: Programming the Arduino Pro Mini via a USB-UART adaptor.

Here's a view of the final working PCB running this sketch[7] (find it on GitHub):

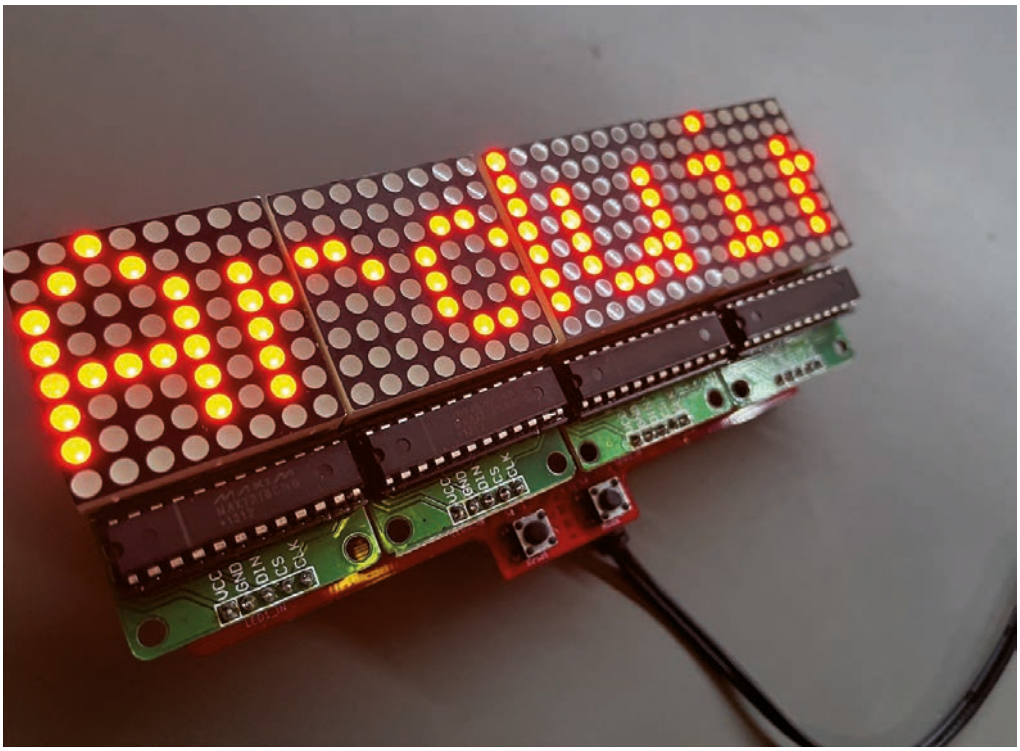


Figure 2.7.6: The final working PCB.

7 https://gist.github.com/futureshocked/1a14e52a9b7d610df06d50f5137f8b29

3 : Project - MCU datalogger

3.1. Project - Introduction

Welcome to a new project. In this project, you will design a printed circuit board for a microcontroller data logger. The data logger is based on an Atmega 328P-AU microcontroller supported by two EEPROMs and a real-time clock. Additional components on the board include status LEDs with their supporting resistors, two crystal oscillators, connectors, and capacitors.

Most components will be used with SMD packages on a rectangular two-layer board with mounting holes on the four corners.

The project highlights are:

1. You will use Git to capture the history of the project's development.
2. You will design two versions of the PCB: one with two layers and one with four layers. Both will use data from the same schematic design.

KiCad, on its own, does not allow the creation of more than one layout for a schematic. Git makes this possible with the use of branches. This project will allow you to practice this aspect of Git-powered PCB design with KiCad.

The schematic design contains components distributed across two sheets. You can see the final schematic below (sheet 1):

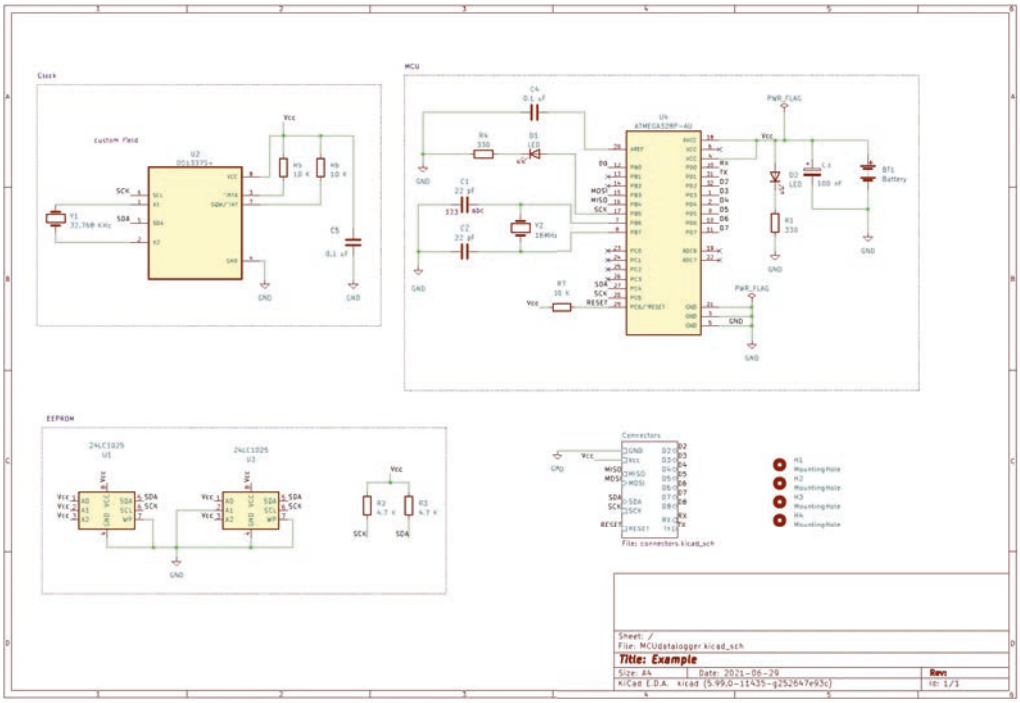


Figure 3.1.1: Sheet 1 of the project's final schematic design.

In Sheet 1, I have placed the main components of the board. Sheet 2 contains the connectors:

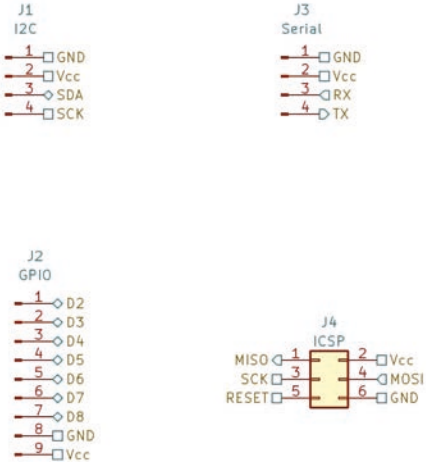


Figure 3.1.2: Sheet 2 of the project's final schematic design.

I have used a combination of line wires and net labels in the schematic design. Other than the distribution of the components across the two sheets, the techniques I have used to draw the schematic should be familiar to you from previous projects.

The most exciting aspect of this project is the layout design: you will design two versions of the PCB. A two-layer and a four-layer version. You can see the final version of the two-layer PCB layout below:

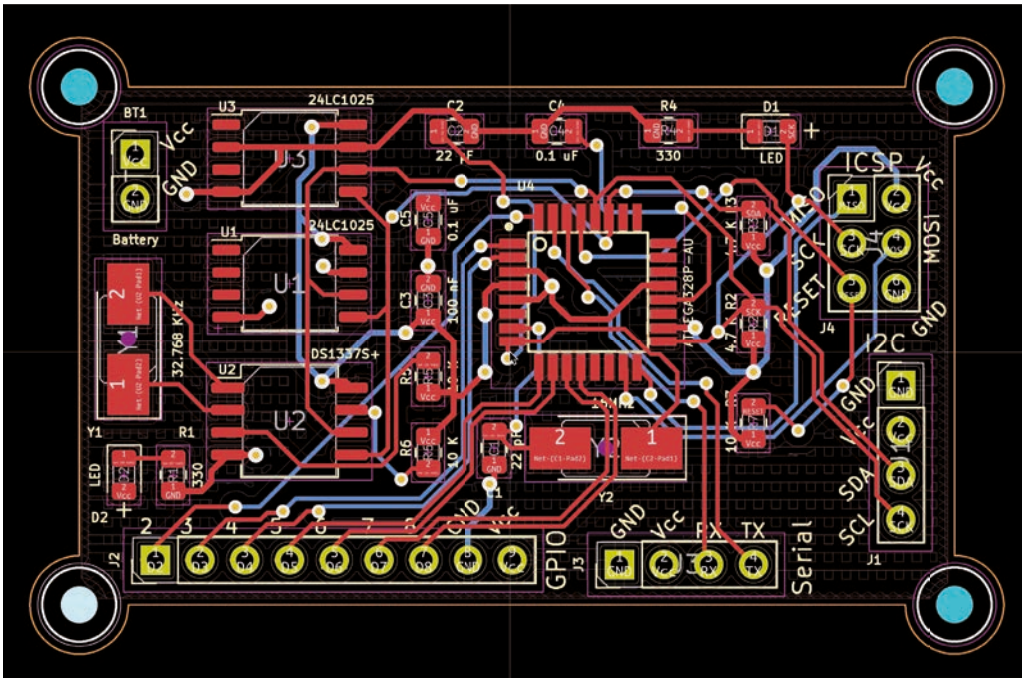


Figure 3.1.3: The project's final layout design (two layers).

You can see the final four-layer PCB below:

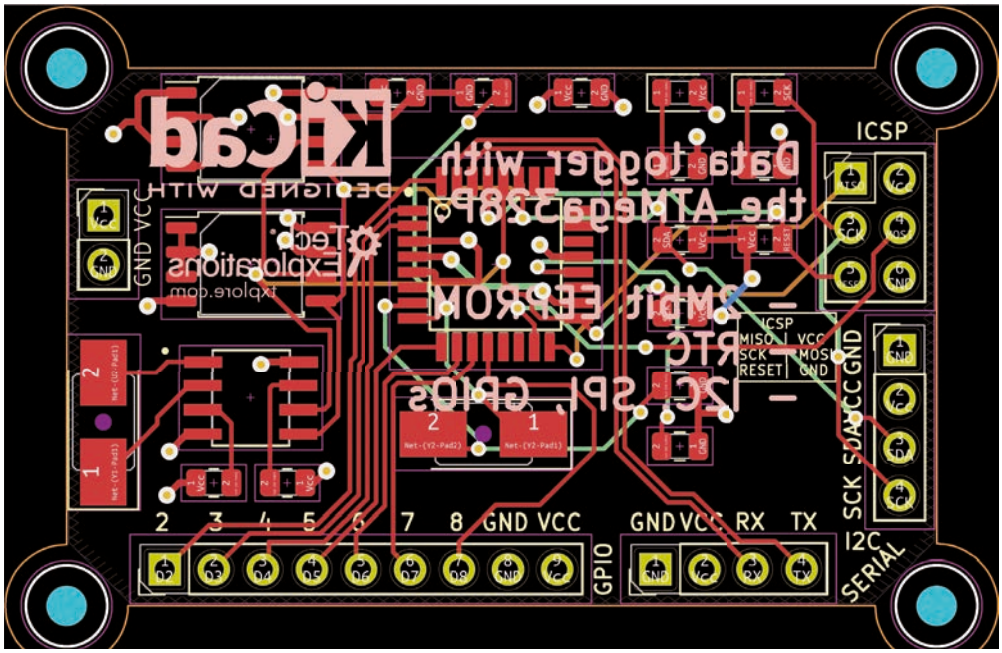


Figure 3.1.4: The project's final layout design (four layers).

The main objectives of this project are:

1. To help you practice skills you acquired in previous projects.
2. To use Git in a non-trivial project to extend KiCad's use cases in a single-schematic and multi-layout project.
3. To gain experience in creating multi-layer PCBs.

Below, you can see the Bill of Materials for this project, as I have extracted it from the KiCad project (learn how later in this book):

Reference	Value	Footprint
BT1	Battery	Connector_PinHeader_2.54mm:PinHeader_1x02_P2.54mm_Vertical
C1, C4	0.1uF	Capacitor_SMD:C_0805_2012Metric
C2, C3	22pF	Capacitor_SMD:C_0805_2012Metric
C5	100nF	Capacitor_SMD:C_0805_2012Metric
D1, D2	LED	LED_SMD:LED_0805_2012Metric
H1-H4	MountingHole	MountingHole:MountingHole_2.1mm
J2	Conn_01x09_Male	Connector_PinHeader_2.54mm:PinHeader_1x09_P2.54mm_Vertical

J1, J3	Conn_01x04_Male	Connector_PinHeader_2.54mm:PinHeader_1x04_P2.54mm_Vertical
J4	Conn_02x03_Odd_Even	Connector_PinHeader_2.54mm:PinHeader_2x03_P2.54mm_Vertical
R1, R2, R6	10K	Resistor_SMD:R_0805_2012Metric
R3, R4	4.7K	Resistor_SMD:R_0805_2012Metric
R5, R7	330	Resistor_SMD:R_0805_2012Metric
U1	DS1337S+	Footprints:SOIC127P600X175-8N
U2, U3	24LC1025	Package_SO:SOIC-8_5.23x5.23mm_P1.27mm
U4	ATMEGA328P-AU	Footprints:QFP80P900X900X120-32N
Y1	32.768 KHz	Crystal:Crystal_SMD_5032-2Pin_5.0x3.2mm_HandSoldering
Y2	16 MHz	Crystal:Crystal_SMD_5032-2Pin_5.0x3.2mm_HandSoldering

Table 3.1.1: The Bill of Materials for this project.

I used Snapeda to find the symbol and footprint pairs for U4. You should be able to find all other symbols and footprints in KiCad's libraries.

In the next chapter, you will begin work on this project by creating a new KiCad project and Git repository.

3.2. Create the new project and Git repository

A primary objective of this project is to help you learn how to use Git within a KiCad project. If you are unfamiliar with Git, please consider reading the chapter "5.17. KiCad project management with Git" in the Recipes part of this book before continuing with the project. To keep the project flowing, I will not explain the meaning of each Git command that I use. Instead, I assume that you already know the basics of Git, and in the chapters that follow, I will focus on showing you how to use this knowledge in the context of a complete KiCad project.

Start KiCad and create a new KiCad project. I have named my instance of the project "MCU Datalogger."

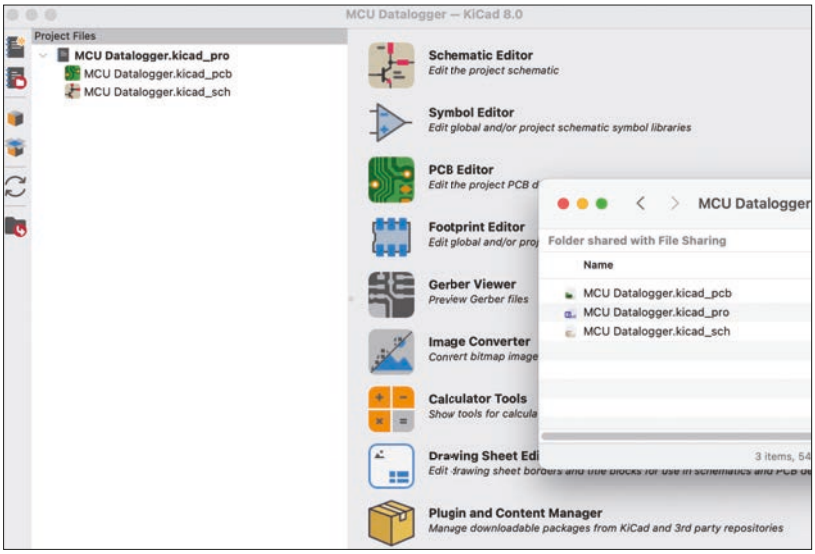


Figure 3.2.1: Starting a new project.

Now that you have a project directory for the project files, create a new Git repository. Open a terminal window and navigate to the project directory. Issue the "init" command to initialize a new repository:

```
% git init
```

Below is my command line session. The second rectangle points to the new ".git" hidden directory containing the project's history.

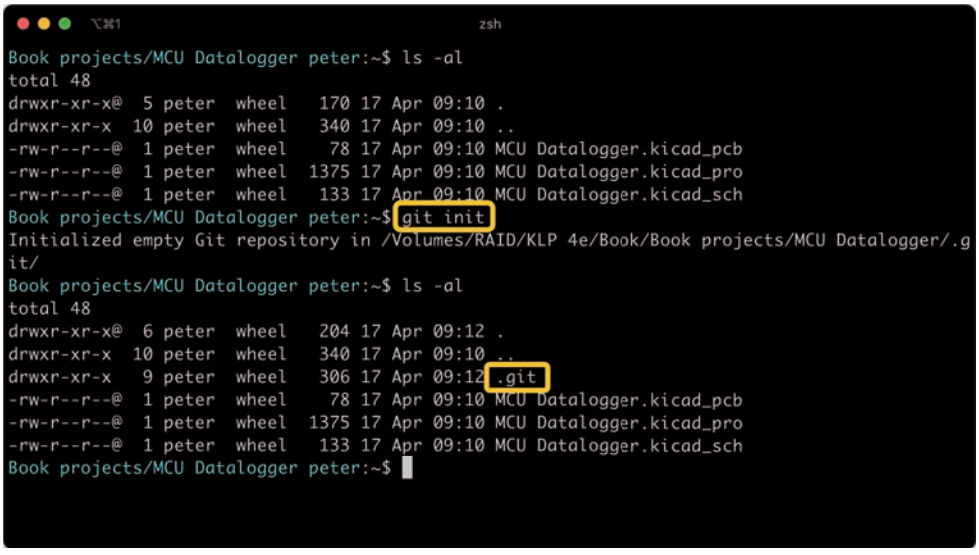


Figure 3.2.2: Creating a new Git repository.

The new Git repository is ready. Let's start tracking the project files. Get the status of the repository to see what is not being tracked:

```
% git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    MCU Datalogger.kicad_pcb
    MCU Datalogger.kicad_pro
    MCU Datalogger.kicad_sch

nothing added to commit but untracked files present (use "git add" to track)
%
```

The repository is not tracking any files yet, and there are no commits. The working branch is "master." Three files are not being tracked. Let's track them. Use the "add" command:

```
 % git add .
```

You can use "git status" again to confirm that the three files are in the staging area but not yet committed. Go ahead and commit them using the "commit" command:

```
% git commit -am "First commit of new project."

[master (root–commit) 7a2b618] First commit of new project.
3 Files changed, 68 insertions(+) E
create mode 100644 MCU Datalogger.kicad_pcb
create mode 100644 MCU Datalogger.kicad_pro
create mode 100644 MCU Datalogger.kicad_sch
%
```

Use the "status" command to confirm that the working tree is clean:

```
% git status
On branch master
nothing to commit, working tree clean
%
```

KiCad automatically creates backup and cache files that don't have to be captured in the repository. I also want to exclude the contents of the Gerbers directory since I can always generate them at will. To exclude those files, use your text editor to edit the" .gitignore" file. Below, you can see the contents of ".gitignore" (please see this recipe for a detailed and up-to-date discussion of .gitignore):

```
fp-inFo-cache
*.kicad_prl
MCU Datalogger-backups/*
MCUDataloggerGerber/*
```

You can see the contents of my ".gitignore" file below:

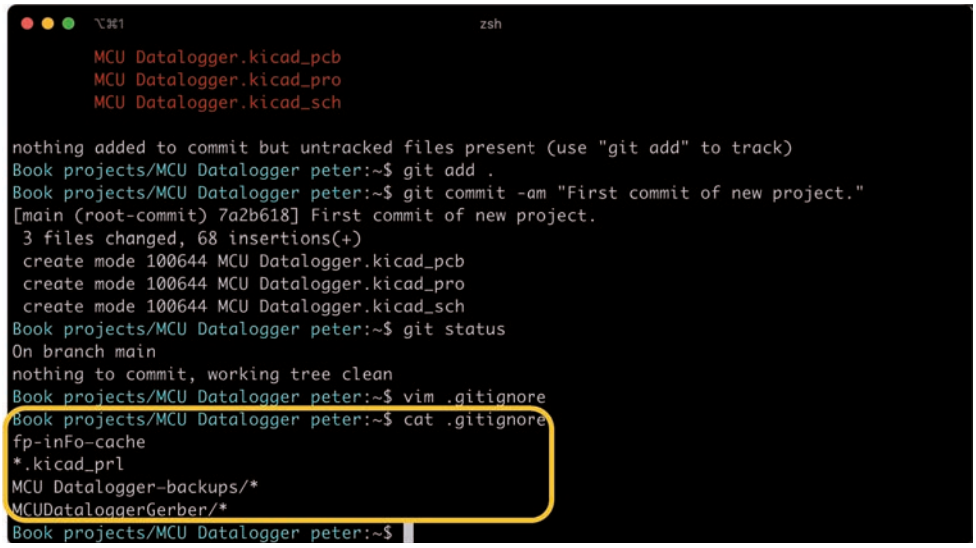


Figure 3.2.3: The contents of the ".gitignore" file.

At this point, you have a new KiCad project, and you are tracking it using Git. Let's continue with the schematic design workflow.

3.3. Schematic design

In this chapter, you will complete the schematic design of this PCB by following the schematic design workflow. You learned about this workflow in the first volume of the book.

3.3.1. Schema 1 - Setup

In this chapter, you will set up your schematic editor. Begin by opening the Preference window (KiCad —> Settings). I have kept the defaults except for the following settings:

- Schematic Editor
  - Display Options
    - Grid thickness: 0.5 px
    - Min Grid spacing: 15 px
  - Colors: Use a custom theme with a white background.
  - Field Name Templates: A new field named "Purpose"; visible.

- Schematic Setup
  - Project
    - Net Classes: create a new net class with the name "Power."
    - Text Variables: created a new variable:
      - Variable Name: project_name
      - Text Substitution: MCU Datalogger with memory and clock

- Page Settings:
  - Issue Date: copied today's date from the date field.
  - Revision: 1
  - Title: ${project_name}

The setup is complete. Don't forget to save the project. Let's continue with updating the Git repository. Use the "status" command to get an update:

```
% git status
On branch master
Changes not staged For commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout —- <file>..." to discard changes in working directory)

     modified: MCU Datalogger.kicad_pro
     modified: MCU Datalogger.kicad_sch

Untracked files:
  (use "git add <File>..." to include in what will be committed)

       .gitignore
       MCU Datalogger-backups/
       ~MCU Datalogger.kicad_sch.lck

no changes added to commit (use "git add" and/or "git commit —a")
%
```

Git reports changes in the project and schematic files as well as three new untracked files. I do not want to track the Mac OS system file ".DS_Store" and the KiCad lock file (.lck), so I will add them to the ".gitignore" file. The other two files should be tracked. Here is my updated ".gitignore" file (please see this recipe for a detailed and up-to-date discussion of .gitignore):

```
.DS_Store
*.lck
fp-inFo-cache
*.kicad_prl
MCU Datalogger-backups/*
MCUDataloggerGerber/*
```

Add the new files to Git (using the file name instead of the wildcard "." character), and commit them:

```
% git add .gitignore
% git add .
% git commit -am "Setup of the schematic editor for the new project."
[master ddb72ca] Setup of EEschema for new project.
4 files changed, 391 insertions(+), 9 deletions(-)
create mode 100644 .gitignore
create mode 100644 MCU Datalogger.kicad_prl
%
```

Use the "log" command to see Git's recent history:

```
% git log
commit 60a5d945aa989409cdaf22c533c53343d25990e7
Author: peter <peter@txplore.com>
Date:   Wed Apr 17 09:25:41 2024 +1000

    Setup of the schematic editor for the new project.

commit 7a2b61815ae3e2db11e34d7820920db0e0cbfe49
Author: peter <peter@txplore.com>
Date:   Wed Apr 17 09:14:25 2024 +1000

    First commit of new project.
```

The log shows the two commits done so far, along with their commit IDs and other information. In the next segment of this chapter, you will add the schematic symbols to the editor.

3.3.2. Schema 2 - Symbols

Let's add the schematic symbols to the editor sheet. Use the BoM table from the introduction chapter to help you find the required symbols. Add all symbols except for the headers. I plan to add the symbols in a second sheet in the next segment of this chapter.

Download the files from Snapeda (all three types: symbols, footprints, and 3D models, if available), and store them in your project folder. Install them as I describe in the relevant chapter from My book Fundamentals and Projects. You should be able to find all symbols in

KiCad's symbol libraries except for those for the microcontroller ("U4") and the real-time clock ("U1"). I installed these libraries in my Project Specific Libraries tab:

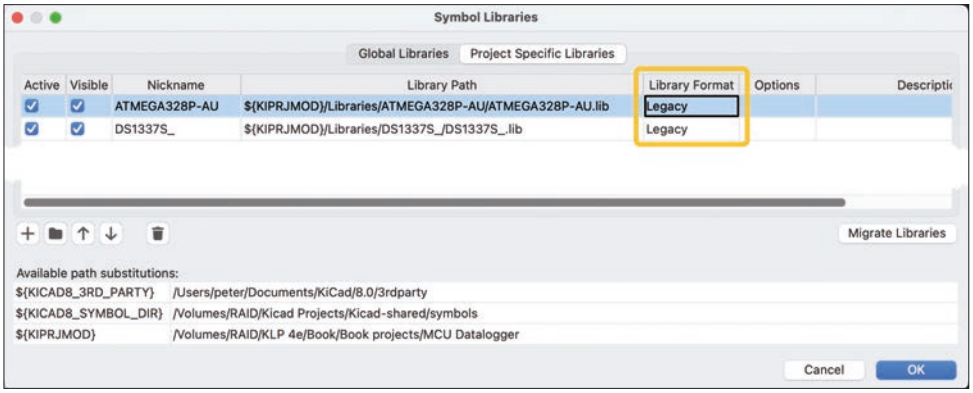


Figure 3.3.2.1: Changes to the Installed external symbol libraries.

In the screenshot above, notice that both libraries are in the Legacy format. This may be different for you if you have used a newer version of these libraries. If you do have the legacy version, you should migrate both libraries to the current KiCad format. To do this, select the library you want to migrate, then click the Migrate Libraries button, and click "OK" in the confirmation window (see below).

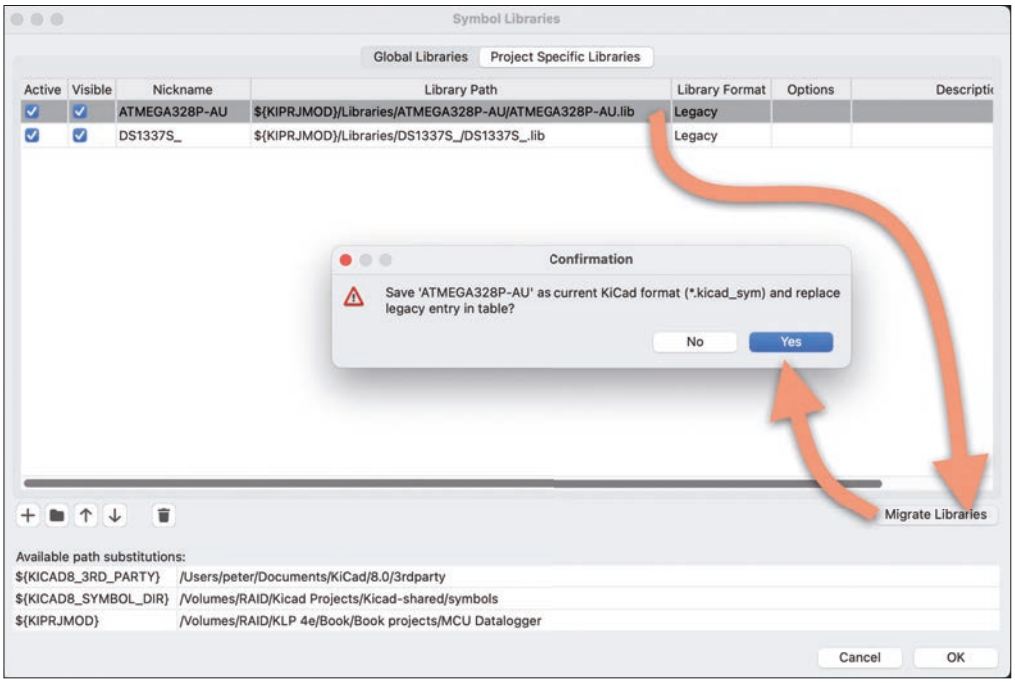


Figure 3.3.2.2: Migrate a legacy library to the current KiCad format.

Repeat the process for the second library. At the end of this process, your project libraries should both be using the current KiCad symbol format.

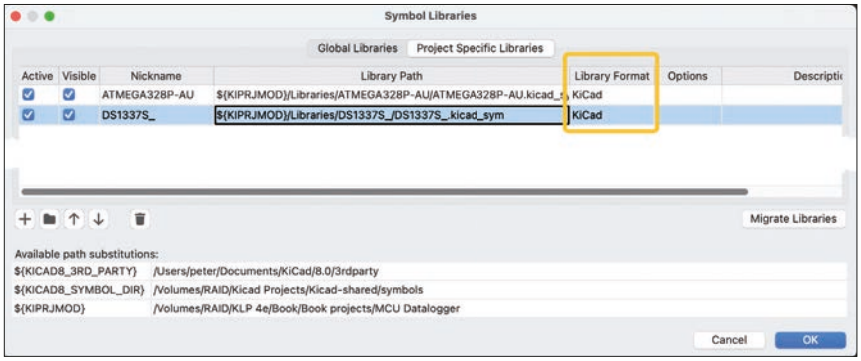


Figure 3.3.2.3: Both project libraries use the current KiCad symbols format

After adding all symbols (except for the headers), your schematic editor sheet will look like this:

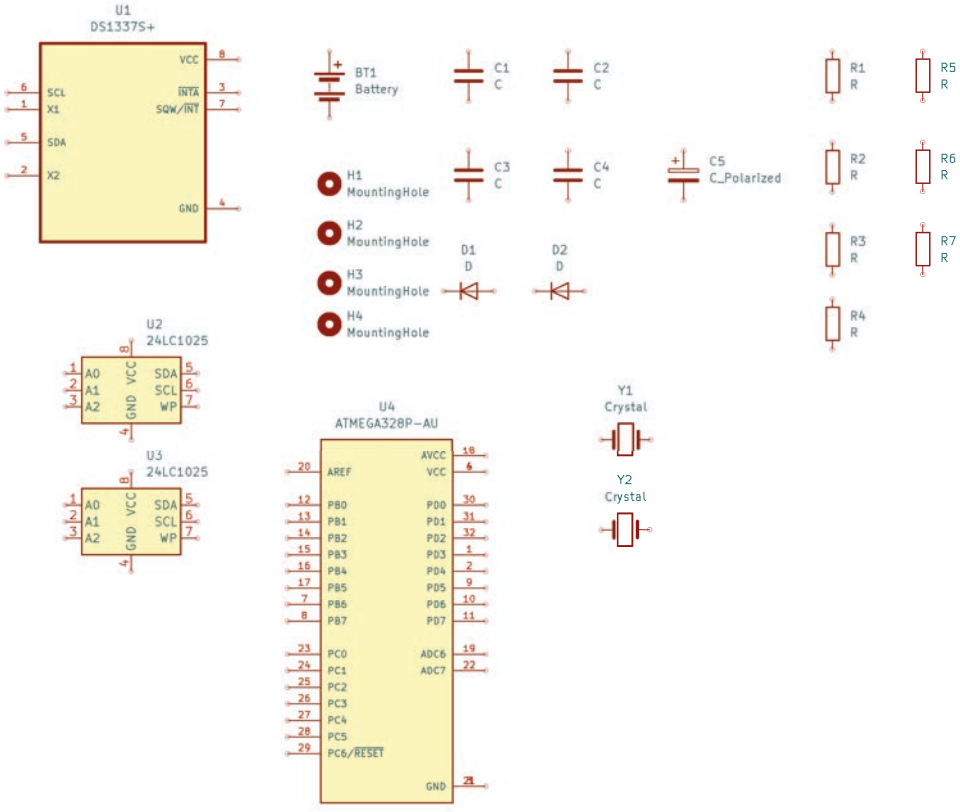


Figure 3.3.2.4: The first sheet of the schematic design.

In the next segment of this chapter, you will create a second sheet for the remaining symbols and the headers.

3.3.3. Schema 2 - Sheet two

Create a hierarchical sheet to contain the connector symbols. Select the hierarchical sheet tool from the right toolbar, click anywhere in the schematic sheet to bring up the sheet properties window, and type in an appropriate name for the sheet and its file. I called mine «Connectors» and «connectors.kicad_sch,» respectively.

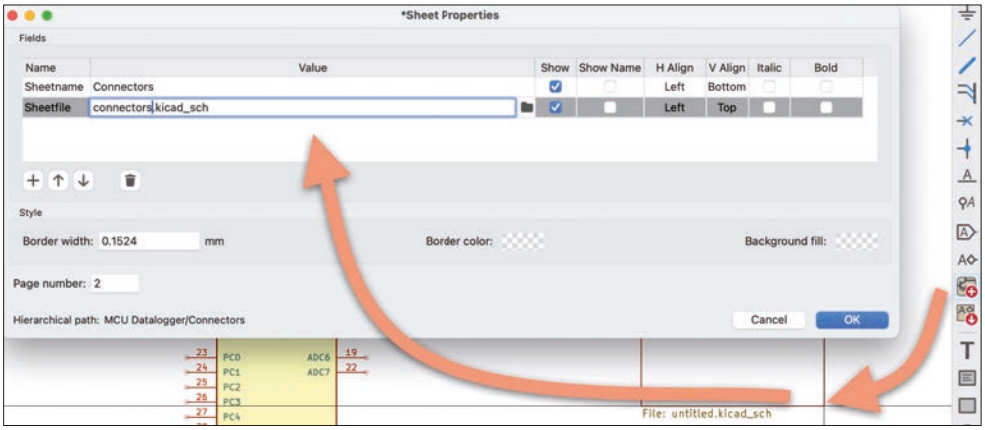


Figure 3.3.3.1: Creating a hierarchical sheet.

Place the hierarchical sheet symbol on the lower-right side of the schematic:

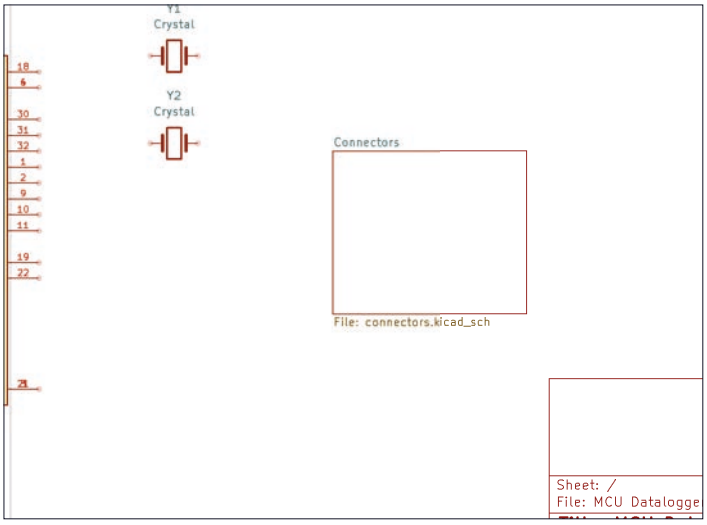


Figure 3.3.3.2: The new hierarchical sheet symbol.

Double-click the hierarchical sheet button to enter the new sheet in the editor. Add the final four symbols for the connectors. The connectors sheet will look like this:

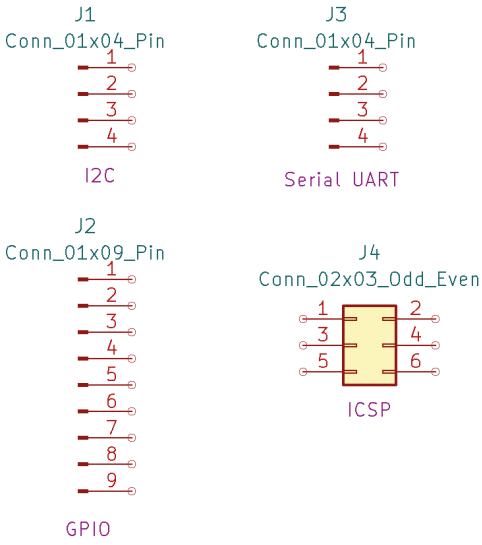


Figure 3.3.3.3: The contents of the "Connectors" sheet.

At this point, all the symbols are in the schematic editor. This is an excellent opportunity to capture the progress in the Git repository. Save your work with KiCad, and check the current status of the repository:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   MCU Datalogger.kicad_pro
       modified:   MCU Datalogger.kicad_sch

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       #auto_saved_files#
       Libraries/
       connectors.kicad_sch
       sym-lib-table

no changes added to commit (use "git add" and/or "git commit -a")
```

Git reports that two tracked files have changed, and a new directory ("Libraries") has been created; three other files are not being tracked. Start tracking for the three new files and commit the changes:

```
% git add .
% git commit -am "Added required symbols to the schematic sheets."
[main 304a0d2] Added required symbols to the schematic sheets.
 18 files changed, 54441 insertions(+), 1 deletion(-)
 create mode 100644 #auto_saved_files#
 create mode 100755 Libraries/3D/DS1337S.STEP
 create mode 100755 Libraries/3D/atmega328p-au.stp
 create mode 100644 Libraries/ATMEGA328P-AU/ATMEGA328P-AU.kicad_sym
 create mode 100755 Libraries/ATMEGA328P-AU/ATMEGA328P-AU.lib
 create mode 100755 Libraries/ATMEGA328P-AU/ATMEGA328P-AU.step
 create mode 100755 Libraries/ATMEGA328P-AU/how-to-import.htm
 create mode 100644 Libraries/DS1337S_/DS1337S_.kicad_sym
 create mode 100755 Libraries/DS1337S_/DS1337S_.lib
 create mode 100755 Libraries/DS1337S_/DS1337S_.step
 create mode 100755 Libraries/DS1337S_/how-to-import.htm
 create mode 100755 Libraries/Footprints/ATMega328P-edited.kicad_sym
 create mode 100755 Libraries/Footprints/QFP80P900X900X120-32N.kicad_mod
 create mode 100755 Libraries/Footprints/SOIC127P600X175-8N.kicad_mod
 create mode 100644 connectors.kicad_sch
 create mode 100644 sym-lib-table
```

Use "git status" to confirm that there are no pending commits.

3.3.4. Schema 3 - Arrange, Annotate

In this segment, you will arrange the symbols on the sheet to prepare them for wiring in step four. I will arrange the symbols according to functional groups:

1. The microcontroller and power group.
2. The real-time clock group.
3. The EEPROM group.
4. The mounting holes group.
5. The connectors.

Below is the final arrangement of the symbols in the root sheet before the automatic annotation:

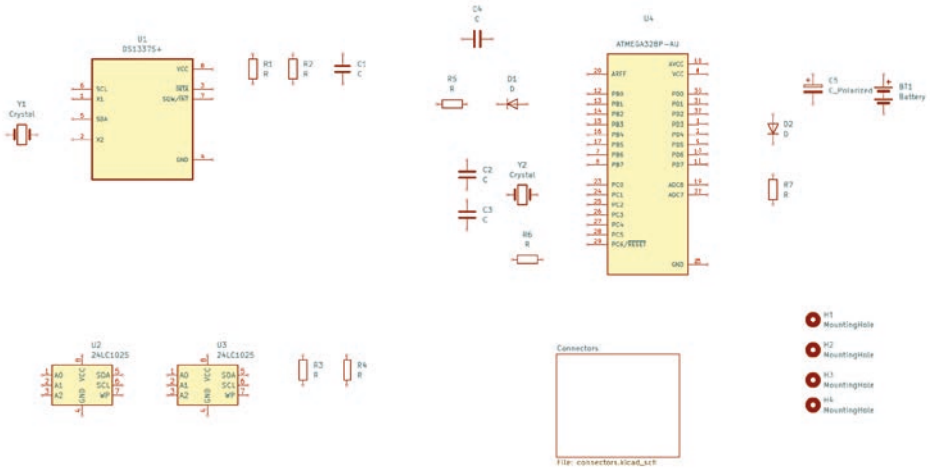


Figure 3.3.4.1: Final arrangement of symbols in the root sheet.

Thanks to the auto-annotator, the schematic is fully annotated. This is an opportunity to commit the change to the Git repository. Here is my session on the command line:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   #auto_saved_files#
      modified:   MCU Datalogger.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Changed locations of symbols (arrange)."
[main caca508] Changed locations of symbols (arrange).
 2 files changed, 159 insertions(+), 165 deletions(-)
```

With the change committed, you can continue with the project's next step, adding the component values.

3.3.5. Edit component values

In this chapter segment, you will add the values for components such as the capacitors and the resistors. Use the information in the BOM table in the introduction of this project as a source. You can use one of these methods to edit the component values:

1. Open each symbol's Properties window (double-click on a symbol), and type the value in the Value field, as in the example below:

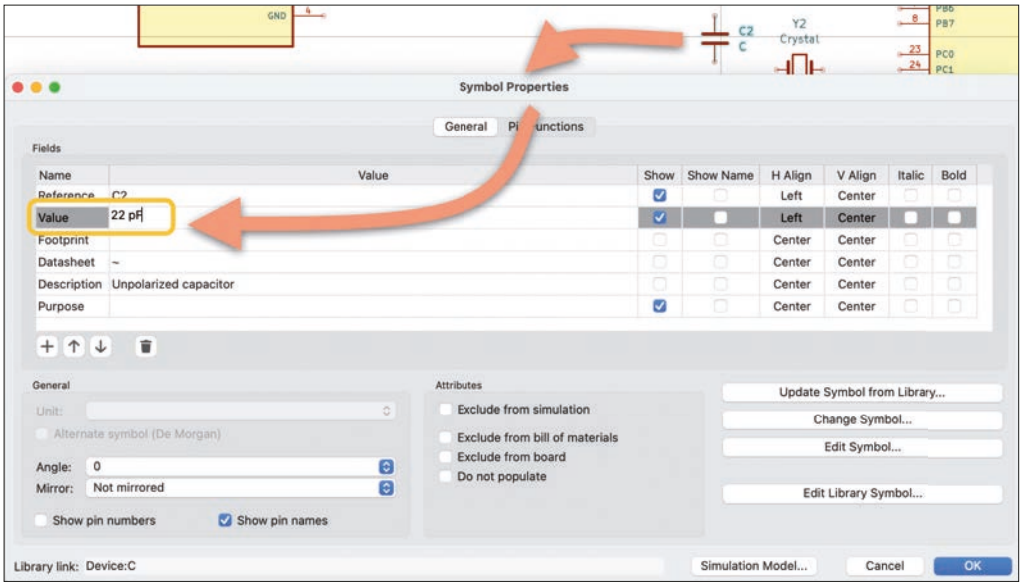


Figure 3.3.5.1: Editing the value field of a symbol.

2. Edit the symbol value fields in bulk using the Symbol Fields Table. Click on the Fields Table button from the top toolbar to open the table window, as in the example below:

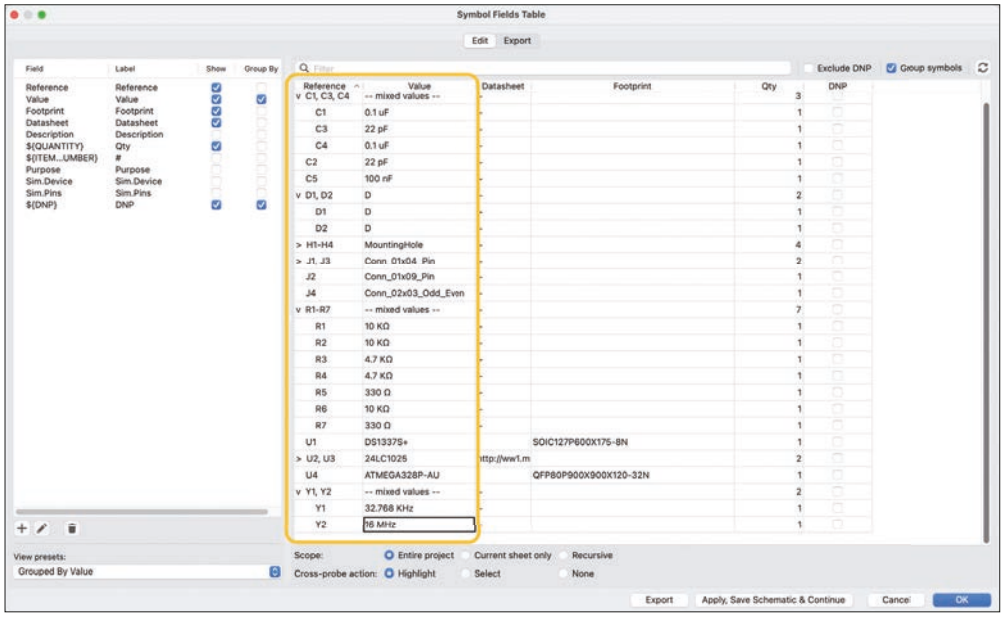


Figure 3.3.5.2: Editing the value field of all symbols using the Symbol Fields Table.

By the end of this process, the values for each of the symbols in the schematic should look like those in Figure 3.3.5.3.

Let's update the Git repository:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   MCU Datalogger.kicad_pro
       modified:   MCU Datalogger.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
% git add .
% git commit -am "Added component values."
[main ee35e14] Added component values.
 2 files changed, 53 insertions(+), 15 deletions(-)
```

Let's continue with the "Associate "part of Step 3 of the schematic workflow.

3.3.6. Schema 3 - Associate

In this step, you will associate the symbols with their footprint counterparts. As with the editing of the values that you finished in the previous segment, you can choose to associate symbols with footprints in two ways:

1. By editing the Footprint field in the symbol's Properties window.
2. By doing the associations in bulk, using the "associations" tool.

I suggest using the bulk option since there are several symbols to work with. Draw from the information presented in the BOM table in the introductory chapter. The "Footprint" column contains the name of the footprint to look for in the Footprints library.

Click the "Associations" button in the top toolbar to open the Assign Footprints window. The middle pane of the window contains the current associations and will be primarily blank (see below).

Follow the process you learned earlier in this book to associate each symbol with a footprint. You can find the footprint names in the BoM table in the introduction. The final Assign Footprints window should look like this:

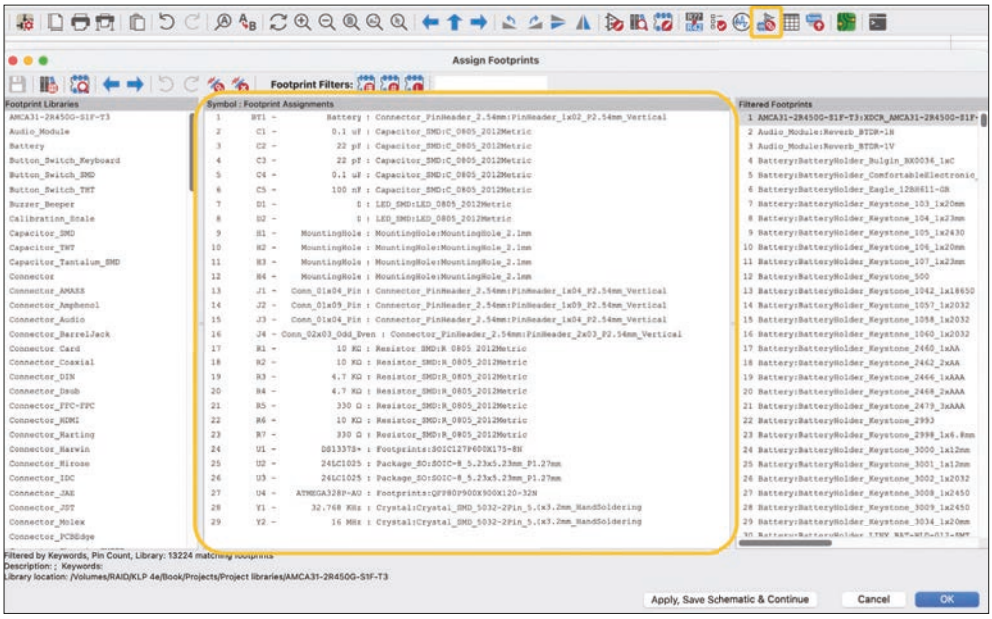


Figure 3.3.6.1: The final symbol-footprint associations.

Let's complete this step by committing the changes to the Git repository:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   MCU Datalogger.kicad_sch
        modified:   connectors.kicad_sch

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fp-lib-table

no changes added to commit (use "git add" and/or "git commit -a")
% git add .
% git commit -am "Completed symbol–footprint associations."
[main 0746b72] Completed symbol–footprint associations.
 3 files changed, 33 insertions(+), 29 deletions(-)
 create mode 100644 fp-lib-table
```

Let's continue with the wiring in Step 4 in the schematic design workflow.

3.3.7. Schema 4 - Wiring of sheet 1

Because the schematic is broken into two sheets, I will do the wiring in two parts. First, I will wire the root schematic using a combination of graphical line wires and net labels. Then, I will connect the symbols in the Connectors sheet with those in the root sheet using hierarchical labels.

To make the wiring process systematic, I will wire symbols that belong to the same functional group before continuing with other components. I will start with the real-time clock group. When the wiring is complete, this group looks like this:

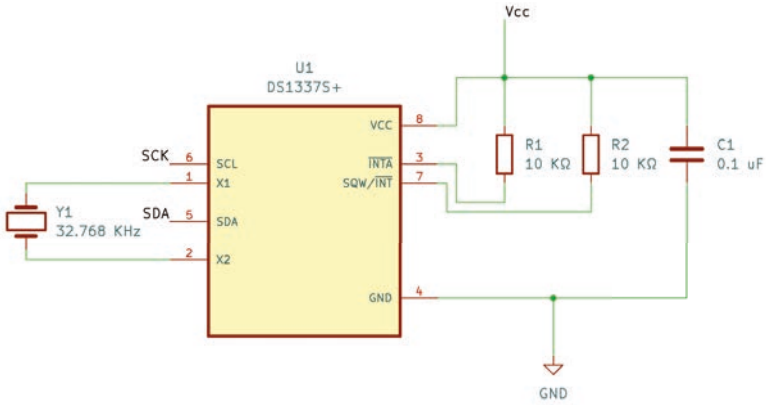


Figure 3.3.7.1: Real-time clock group wiring is complete.

Continue with the memory groups:

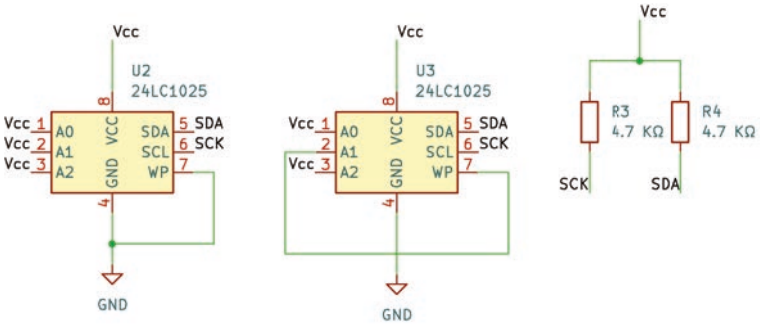


Figure 3.3.7.2: Memory group wiring is complete.

Next, the microcontroller and power group. You should be careful here because U4 has a few overlapping pins, at least in the version of the symbol that I have used. Edit the symbol in the symbol editor to separate the overlapping pins before you continue with the wiring. In the screenshot below, you can see the before-and-after versions of the symbol.

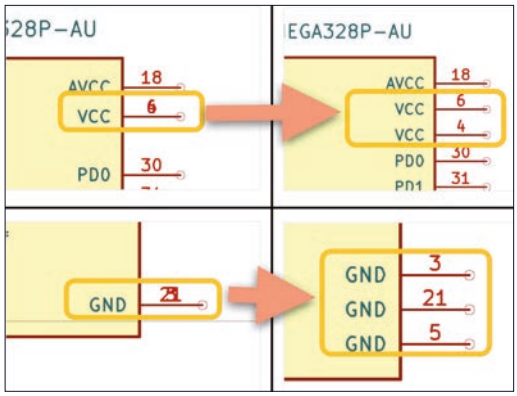


Figure 3.3.7.3: Editing the MCU symbol to separate overlapping pins.

Now, you can proceed with the wiring of the MCU block, as in the screenshot below:

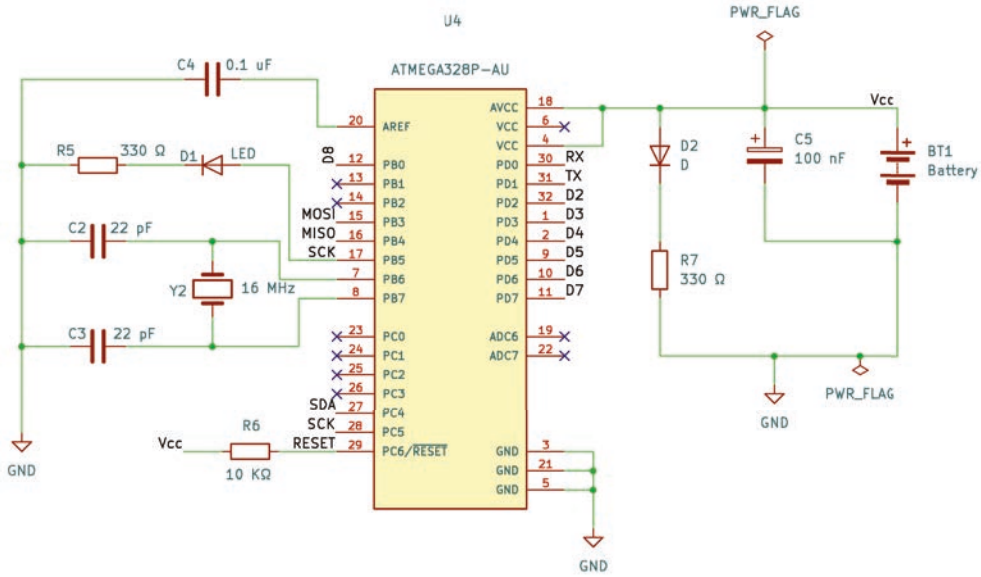


Figure 3.3.7.4: MCU and power group wiring are complete.

The wiring for the root sheet is complete. You can see it below:

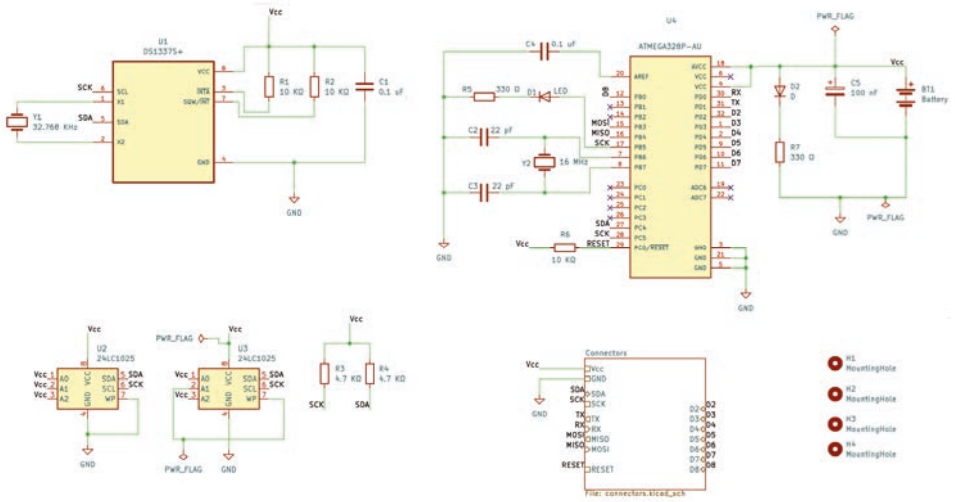


Figure 3.3.7.5: Root sheet wiring is complete.

The second sheet remains to be wired, but this is an excellent opportunity to commit the changes to the Git repository:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       deleted:    MCU Datalogger-backups/MCU Datalogger-2024-04-17_092157.zip
       modified:   MCU Datalogger.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Completed wiring of main sheet."
[main 88d4669] Completed wiring of main sheet.
 2 files changed, 2548 insertions(+), 286 deletions(-)
 delete mode 100644 MCU Datalogger-backups/MCU Datalogger-2024-04-17_092157.zip
```

Let's continue with the wiring of the Connectors sheet.

3.3.8. Schema 4 - Wiring of sheet 2

The wiring of the symbols in the Connectors sheet is pending. Let's complete it now.

The Connectors sheet contains four connector symbols. The pins on those symbols must be connected to pins on symbols in the root sheet. For this reason, I will not use any graphical line wires. Instead, I'll use a combination of net labels and hierarchical sheet labels.

Open the Connectors sheet by double-clicking on the hierarchical sheet symbol in the root sheet. Alternatively, you can open the Schematic Hierarchy pane on the left of the

editor (See "2" below) by clicking on the Hierarchy button in the left toolbar ("1"). Click the hierarchical label button in the right toolbar to create a hierarchical label ("3"). In the Properties window that appears, give the label a name and set its properties (see example below).

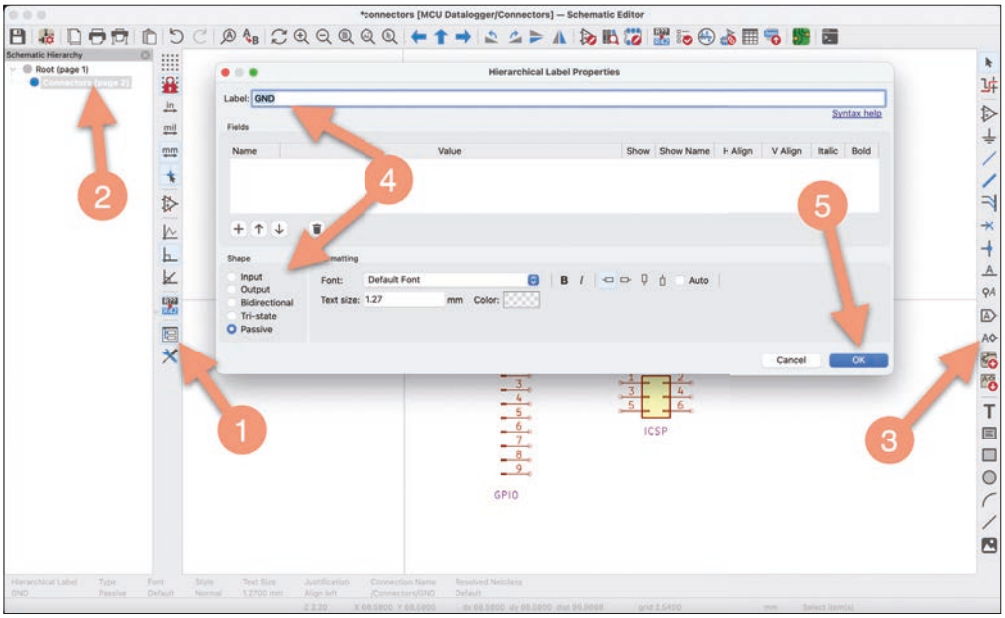


Figure 3.3.8.1: Creating a hierarchical label.

Attach this label to J1 pin 1 like this:

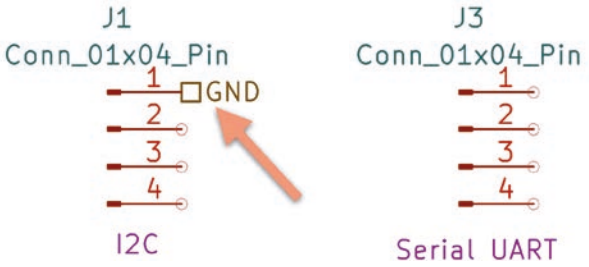


Figure 3.3.8.2: Attaching a hierarchical label to J1 pin 1.

Note that the symbol representing the hierarchical label depends on the shape you choose in its Properties windows. The shape in the example above represents an "output" label. As this pin is connected to the GND net, I will select the "passive" shape.

Continue the same way until you have created hierarchical labels for all pins in this sheet. Once completed, the Connectors sheet will look like this:

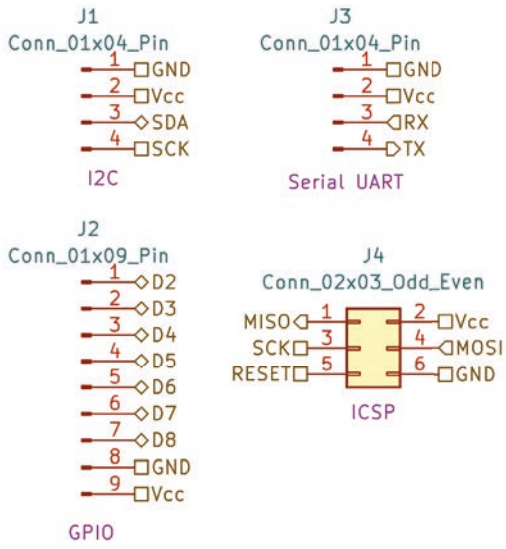


Figure 3.3.8.3: Hierarchical labels attached to all pins.

Save your work and open the root sheet. The hierarchical labels exist in the Connectors sheet. You will need to:

1. Expose those labels to the root sheet.
2. Connect the exposed hierarchical labels to other pins using graphical line wires or net labels.

Complete step one using the "Add a hierarchical sheet (S)" tool from the right toolbar. Click anywhere inside the hierarchical sheet box to drop one of the hierarchical labels and place it along the perimeter with the tool selected. You can see the first hierarchical label attached to the left of the box below:

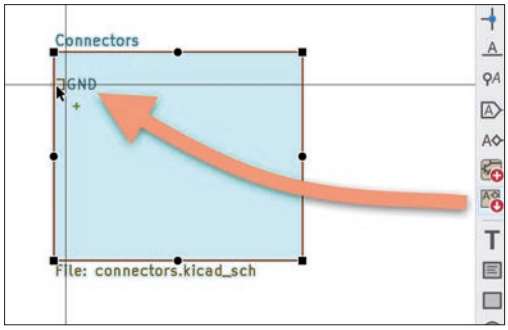


Figure 3.3.8.4: Exposing the hierarchical labels.

Continue in the same manner to expose and place the remaining hierarchical labels. Here is the completed Connectors sheet symbol, with all hierarchical labels in their final positions:

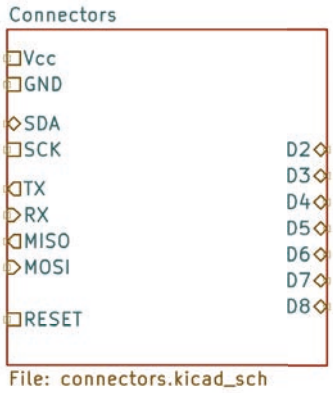


Figure 3.3.8.5: The Connectors hierarchical sheet symbol with exposed hierarchical labels.

I have placed all labels connecting to a digital pin on the right side and the rest on the left.

The next step is to connect the hierarchical labels to their corresponding pins in the root sheet. In the example below, I have used a Vcc net label, a GND symbol, and graphical line wires to connect the Vcc and GND hierarchical labels to the local news:

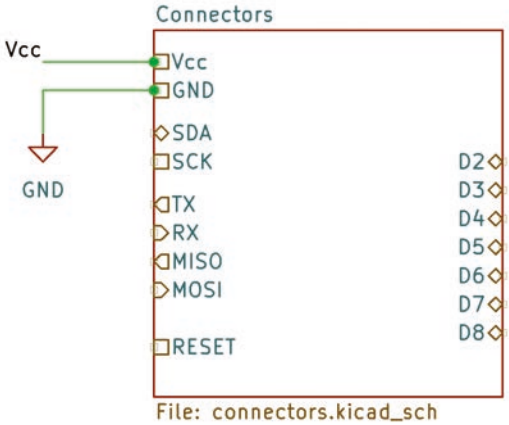


Figure 3.3.8.6: Connected the Vcc and GND hierarchical labels to local nets.

Continue to complete the rest of the connections using net labels. The result of this work is below:

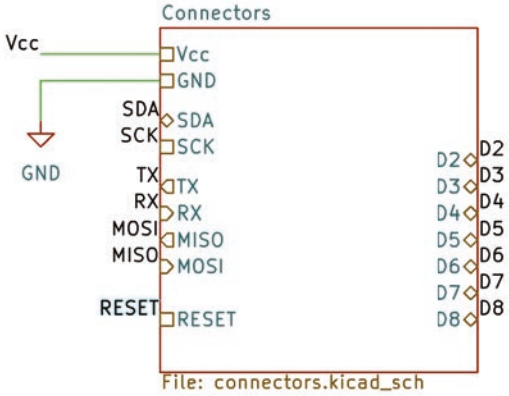


Figure 3.3.8.7: Completed connections of hierarchical labels to local nets.

At this point, the schematic is fully wired. In the next couple of segments, you will allocate nets to net classes, conduct an electrical rules check, and finish the schematic design workflow by adding comments to the schematic.

3.3.9. Schema 5 - Nets

You completed most of the work relating to named nets during the wiring step, using net labels and hierarchical labels to complete the wiring. In this step of the workflow, you will:

1. Cross-check all nets to ensure you haven't missed any.
2. Assign nets to net classes.

You can do the cross-checking visually and confirm that no nets remain unnamed.

Open the Net Classes tab under "Project" in the Schematic Setup window for the net-to-net class assignments. During the workflow setup step, you created the Power net class. Let's assign the Vcc and GND nets to the Power net class. In the "Netclass assignments" box, click on the "+" button to create the two net class assignment patterns and assign them to the Power net class. You can see the two patterns in the figure below.

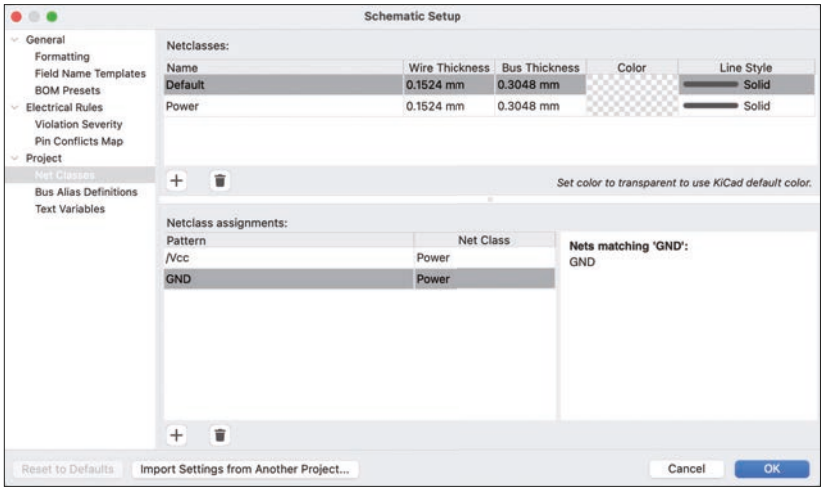


Figure 3.3.9.1: Allocated Vcc and GND nets to the Power net class.

All other nets that don't have a pattern match will remain in the Default net class.

You can create additional net classes, such as one for the digital pin nets and another for the communications nets. For this demonstration project, I will continue with the two net classes you can see in Figure 3.3.9.1. Save your work.

The work in this step of the workflow is complete, so it is time for a new commit to the Git repository:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   MCU Datalogger.kicad_pro
      modified:   MCU Datalogger.kicad_sch
      modified:   connectors.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Allocated GND and Vcc nets to the Power net class."
[main ddc876e] Allocated GND and Vcc nets to the Power net class.
 3 files changed, 719 insertions(+), 5 deletions(-)
```

Continue with a final ERC in the next segment of this chapter.

3.3.10. Schema 6 - Electrical Rules Check

This final ERC should not reveal any defects if you have carefully worked during the schematic design workflow. Nevertheless, always do an ERC before starting work on the layout design.

Open the ERC window and run the check. Mine revealed two minor issues: one warning and two errors.

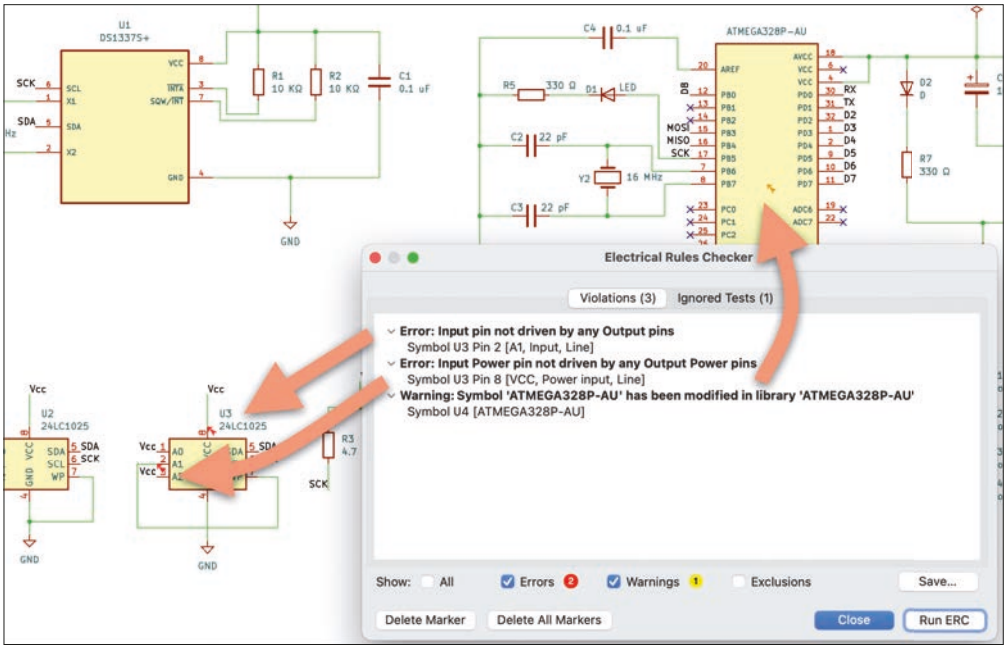


Figure 3.3.10.1: The ERC shows one warning.

The warning relates to a modification that I made to the Atmega328P-AU symbol. It is warning me that the symbol I use in the schematic is different from the one stored in the library. That is not something that concerns me for this project, so I will safely ignore this warning.

The two errors relate to an inconsistency between the roles of pins 2 and 8 of U3 and the nets these pins are connected to. To examine these pins and their assigned roles closely, double-click on U3 and click the Pin Functions tab.

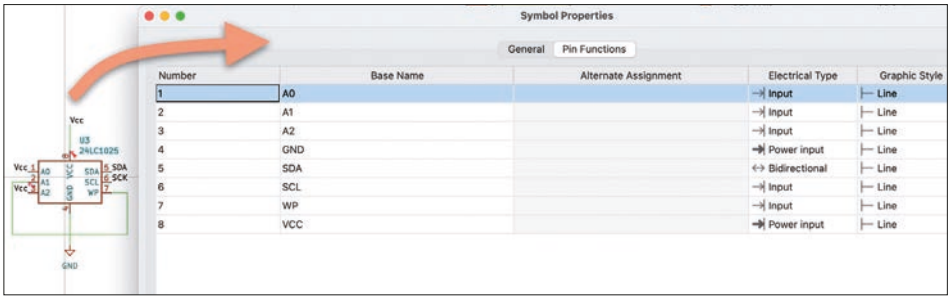


Figure 3.3.10.2: The Pin Functions for U3.

The table shows that pin 2 is an Input, and pin 8 is a Power Input. The ERC detects errors because these roles are incompatible with the GND and Vcc nets, even though the wiring is electrically correct. I will resolve this ERC error by telling the ERC that the wiring is correct using the PWR_FLAG symbol. Attach a PWR_FLAG symbol to the Vcc and GND nets like this:

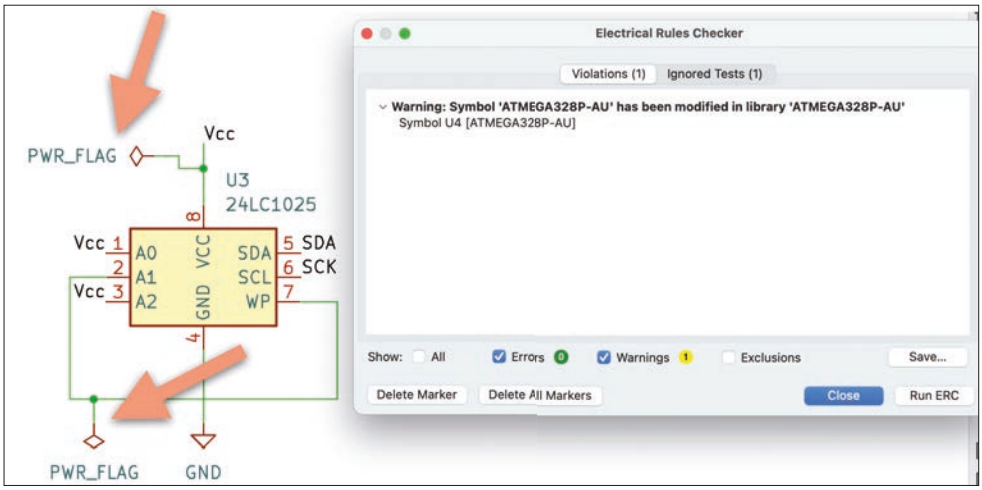


Figure 3.3.10.3: The PWR_FLAG symbols suppress the ERC errors.

When you repeat the ERC, you will see that the two errors have been suppressed. The remaining warning can be ignored.

Save the project and commit the changes to the Git repository. In the next segment, we will complete the work in the schematic design workflow by adding comments to the schematic.

3.3.11. Schema 7 - Comments

You will add simple graphics (boxes) and text to complete work in the schematic design workflow. Here is a list of comment items I have added to my schematic:

- Use lines to box in symbols that belong in the same functional group:
  - Real-time clock.
  - MCU.
  - EEPROM.
  - Mounting holes.
  - Connectors.

- Use text labels to give a name to each functional group.

Below, you can see the two sheets with my comments:

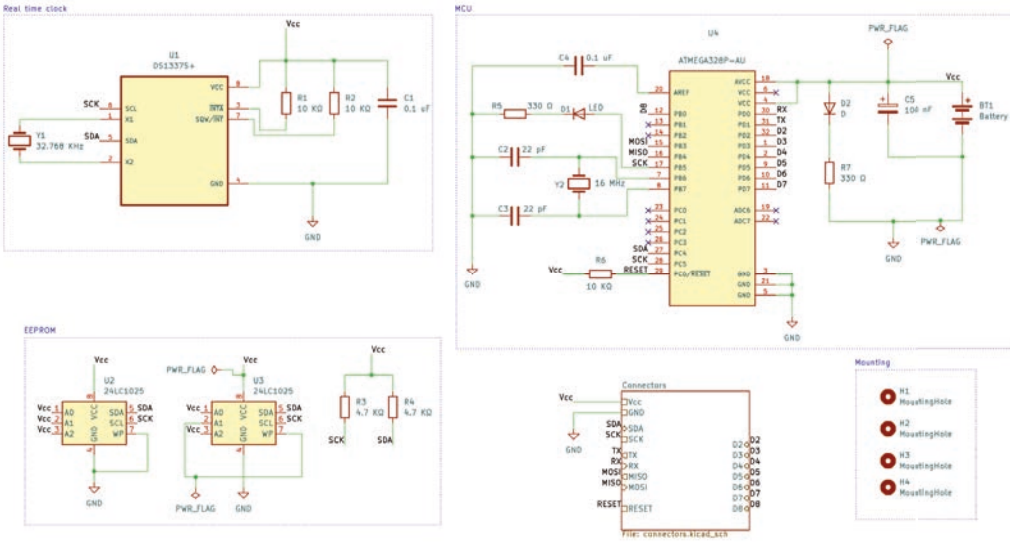


Figure 3.3.11.1: Root sheet with comments.

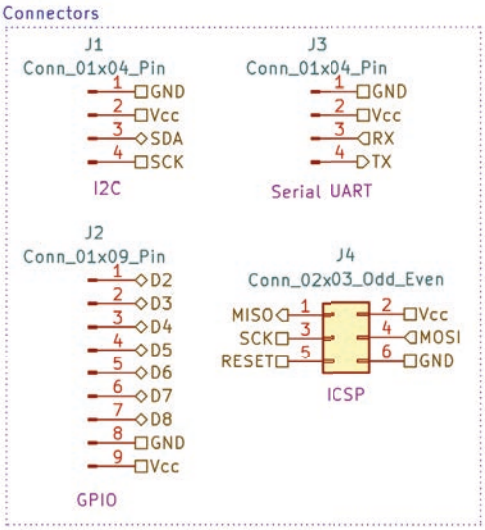


Figure 3.3.11.2: Connectors sheet with comments.

This completes all work in the schematic design workflow. Save the changes, and commit to Git:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

```
        modified:   MCU Datalogger.kicad_sch
        modified:   connectors.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Completed comments."
[main 1e2e322] Completed comments.
 2 files changed, 134 insertions(+), 24 deletions(-)
```

In the next chapter, you will begin work on a two-layer version of the layout design. In a later chapter in this project, you will also create a four-layer layout. To keep the two layouts separate, you will create dedicated Git branches.

So, before starting work on the two-layer layout, you will create the two-layer Git branch. Let's do that in the next chapter.

3.4. Create the 2-layer branch in Git

KiCad cannot support multiple layouts for a schematic. However, this is something that becomes possible with the use of Git. In this project, one of the objectives is to design two versions of the layout: a two-layer PCB and a four-layer PCB. Since KiCad can only "see" one layout file, you will create a dedicated Git branch for each layout version.

In this chapter, you will create a new branch for the two-layer PCB. Later in this project, you will create a separate branch for the four-layer PCB.

Ensure you work in the main branch (use "git branch" for this). Then, create a new branch named "2layer" and switch to it. You can do both in a single command like this:

```
% git checkout -b 2layer
Switched to a new branch '2layer'
```

Check the status of the repository to confirm that its contents are clean, meaning that it matches the contents of the main branch from where the new branch stems:

```
% git status
On branch 2layer
nothing to commit, working tree clean
```

The new branch is clean, and there are no changes to commit.

Finally, use the "branch" command to get a list of branches in the repository:

```
% git branch
* 2layer
  main
```

The repository now contains two branches. The active branch is "2layer". You will continue to commit changes to this layer for all work I describe in the next chapter.

An important question is how to deal with changes to the schematic. You can answer this question in several ways, but I will describe my tested and simplified process below.

If you are working on the layout of a PCB in a dedicated branch and you need to make a change to the schematic, follow this process:

1. Commit any changes to the layout branch.
2. Switch to the main branch, which contains the authoritative version of the schematic.
3. Make the necessary changes to the schematic and save the sheet.
4. Commit the changes to the main branch.
5. Switch to the layout branch.
6. Merge the main branch into the layout branch.
7. Continue work in KiCad and import schematic changes into the layout editor.

You can learn how to merge changes between branches in the Git chapter in the Recipes part of this book.

Now that you have created the two-layer Git branch, you can continue with the layout workflow in the next chapter.

3.5. Layout design

In this chapter, you will work on the layout design of the two-layer version of the PCB. You will be working in the "2layer" branch of the project Git repository that you created in the previous chapter.

Let's begin by setting up the layout design editor.

3.5.1. Layout 1 - Setup

Open the PCB editor and confirm the editor settings. The bullet list below shows any non-default configuration I have applied in my project instance.

- KiCad Preferences.
    - PCB Editor.
        - Display Options.
            - Min grid spacing: 15 px.

- Board Setup.
    - Board Stackup.
        - Physical Stackup.
            - Copper layers: 2

- Text & Graphics.
  - Text Variables: "project_name" variable inherited from the schematic editor.
- Net Classes.
  - Power.
    - Clearance: 0.25 mm
    - Track Width: 0.35 mm
    - Via Size: 0.9 mm

- Page Settings.
  - Issue Date: copy today's date.
  - Revision: 1
  - Title: ${project_name}
  - Comment1: 2-layer PCB version.

Save the setup changes, and then update the PCB from the schematic.

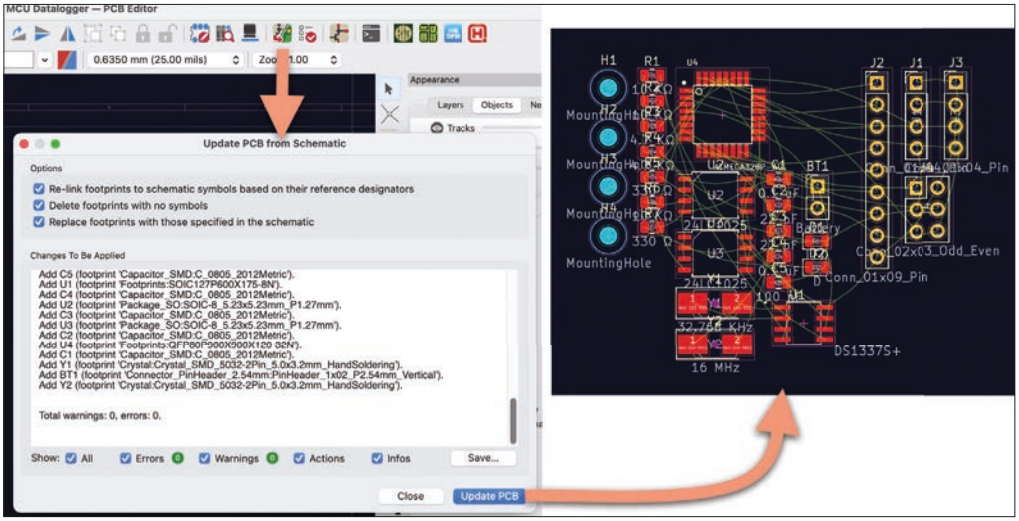


Figure 3.5.1.1: Updated PCB from the schematic.

Save the changes in the layout editor and commit them to the Git repository. You can see my command line session below:

```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      deleted:    #auto_saved_files#
      modified:   MCU Datalogger.kicad_pcb
      modified:   MCU Datalogger.kicad_pro
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
       fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Setup the PCB editor."
[2layer 19f06b7] Setup the PCB editor.
 3 files changed, 7853 insertions(+), 8 deletions(-)
 delete mode 100644 #auto_saved_files#
```

Continue with step two of the layout design workflow, where you will draw the rough outline.

3.5.2. Layout 2 - Outline and constraints

I will follow the same process as in the previous projects, where I first draw a rough outline for the PCB. I will re-draw the final, refined outline once I have placed all footprints within the rough outline. For the rough outline I am about to draw, I have listed the following constraints:

1. The outline should be large enough to accommodate all footprints.
2. The outline should be small to minimize manufacturing costs.
3. The outline should be rectangular, with four mounting holes at its edges, to make it easy to fit inside a project box.
4. The headers should be along the edges of the PCB to make them easier to access.

I start by evaluating the total dimensions of the footprints as they appear in the editor after importing them from the schematic editor. You can see the footprints bundled together below:

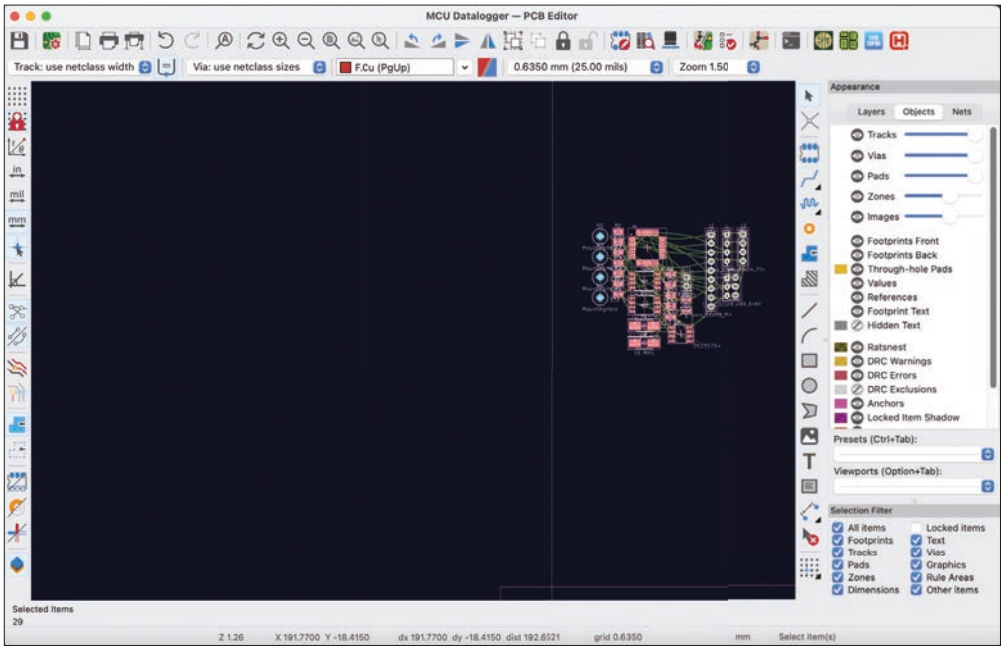


Figure 3.5.2.1: I will create a rough outline for the bundled footprints.

Enable the Edge.Cuts layer and switch the grid size to 1.27 mm. I used the measuring tool to measure the width and height of the footprint bundle. I measured approximately 50 mm for the width and around 30 mm for the height. This gives me enough information to draw a rectangle that can accommodate the footprints.

Use the rectangle drawing tool from the right toolbar and draw a rectangle. My rough outline measures 50.800 mm x 30.480 mm based on my selected grid size. You can see the result below:

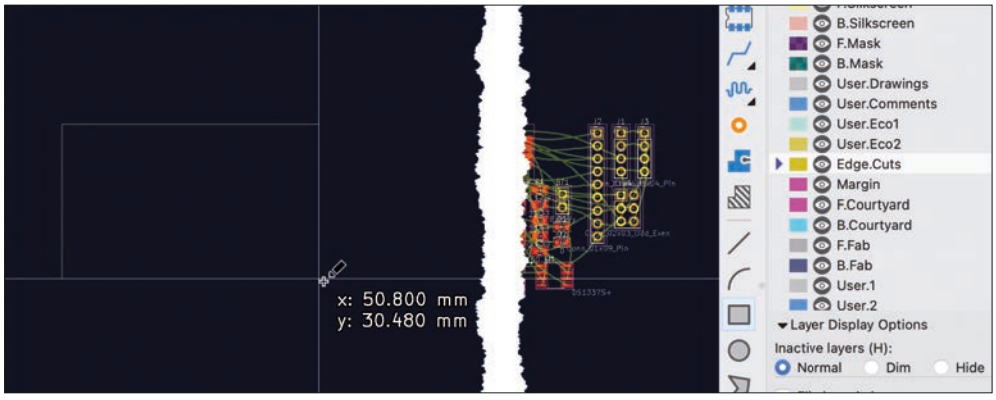


Figure 3.5.2.2: Drawn a rough outline of the board.

The footprints should fit comfortably in this outline. In the next segment, you will place the footprint within the board outline.

3.5.3. Layout 3 - Place components

The rough outline you created in the previous segment will make placing the footprints easier. Compared to the LED matrix display PCB, the footprint placement step in this project is much easier. No external constraints dictate the exact position of footprints.

I have followed a simple process to help me with the placement, which I outline below:

1. Change the grid size to 0.635 mm. This will provide a fine grid for precision spacemen with minimal wasted space.
2. Place the microcontroller footprint in the middle of the board. This will make it easier to connect it to the various peripherals and connectors around it (except for the power connector).
3. Place the mounting holes in the four edges of the outline.
4. Place the connectors along the bottom and right edges.
5. Place the EEPROM and real-time clock footprints on the left of the MCU.
6. Place the oscillator for the real-time clock on the left of the RTC footprint.
7. Place the oscillator for the MCU below the MCU footprint.
8. Place the battery connector on the left edge of the outline.
9. Place the capacitors in the remaining space; try to minimise their distance to their main components using the ratsnest lines as a guide. For example, C1 should be beside U3 and C2 near Y2 to minimise their total copper track lengths.
10. Place the resistors last, as close as possible to their main components. Again, using their ratsnest lines as a guide, try to minimise their copper track lengths.
11. Use the editor alignment tools to ensure all footprints are correctly aligned.

My final placements are below. Yours may differ, but the figure below gives you an example.

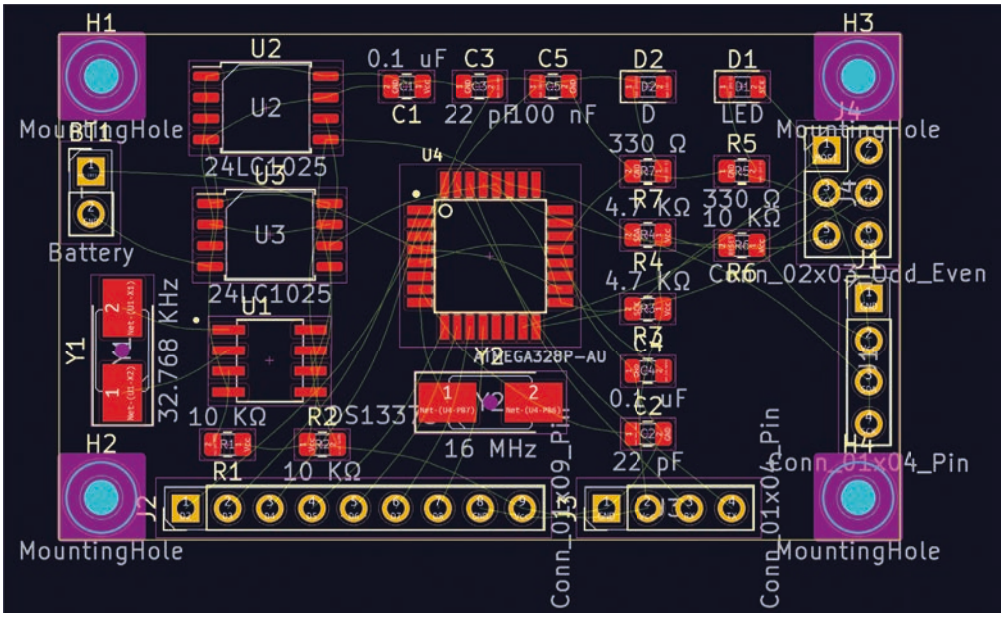


Figure 3.5.3.1: The final placement of the footprints in the rough board outline.

Save your work, and commit the changes to the Git repository:

```
% git commit -am "Completed placement of footprints."
[2layer ef3f728] Completed placement of footprints.
 1 file changed, 437 insertions(+), 422 deletions(-)
```

With the footprint placement complete, you can continue to refine the board outline.

3.5.4. Layout 2 - Outline refinement

The board's rough outline helped you place the footprints within its borders. With the footprints now placed, you can refine this outline and complete the drawing in the Edge. Cuts layer.

Looking at the result of the last two steps of the process (below), you can see that it is possible to reduce the size of the board further by:

1. Reducing the gap between the outer footprints and the outline edges.
2. Placing the mounting holes in outdents in the board's four corners.
3. In the figure below, you can see the progression of the board outline and component placement from start to finish:

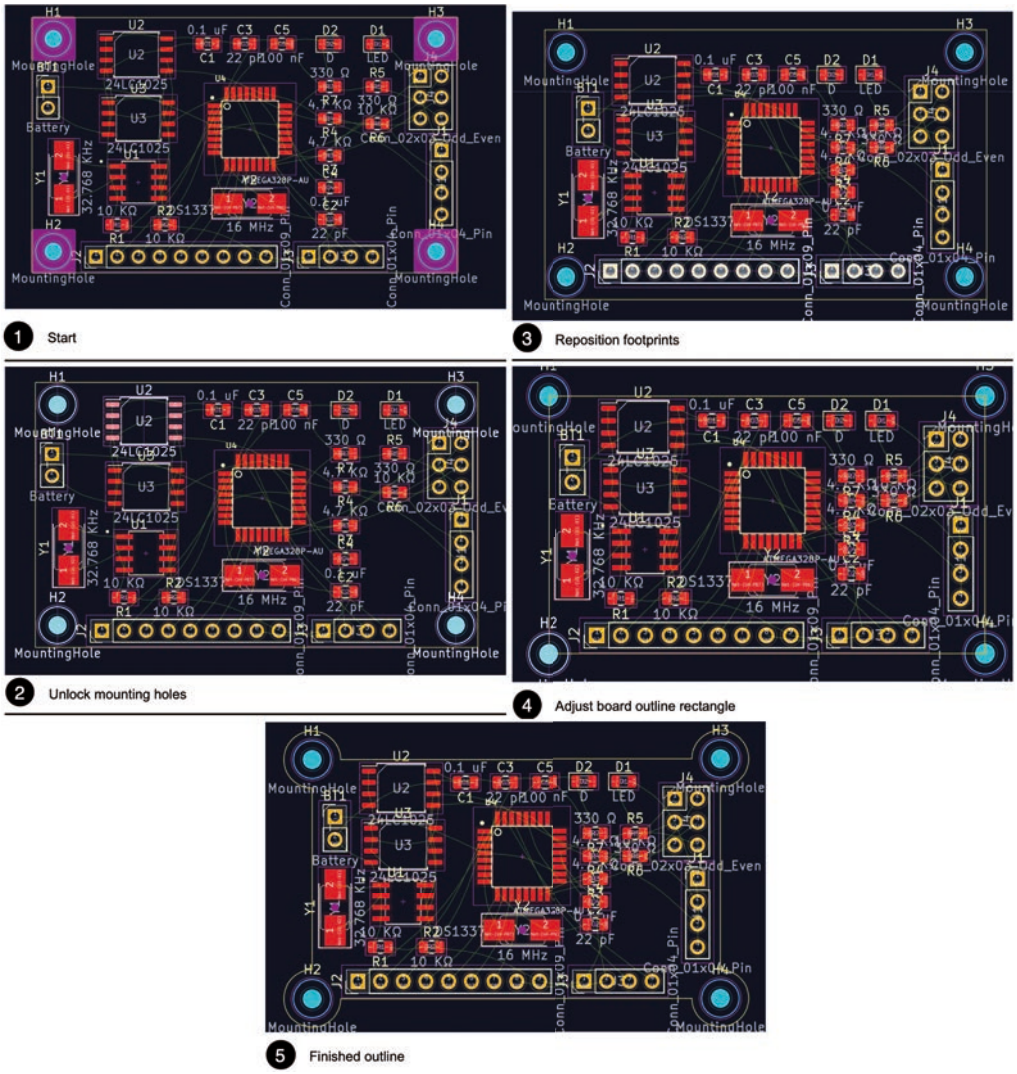


Figure 3.5.4.1: Comparing the rough board outline against the refined version.

The arc tool can be used to create the circular segments of the board. The placement of the mounting holes within the outdents allows you to (carefully) snap them off the board if you don't want to use them so that the board can fit in a smaller box. Below, you can see a larger version of the refined outline:

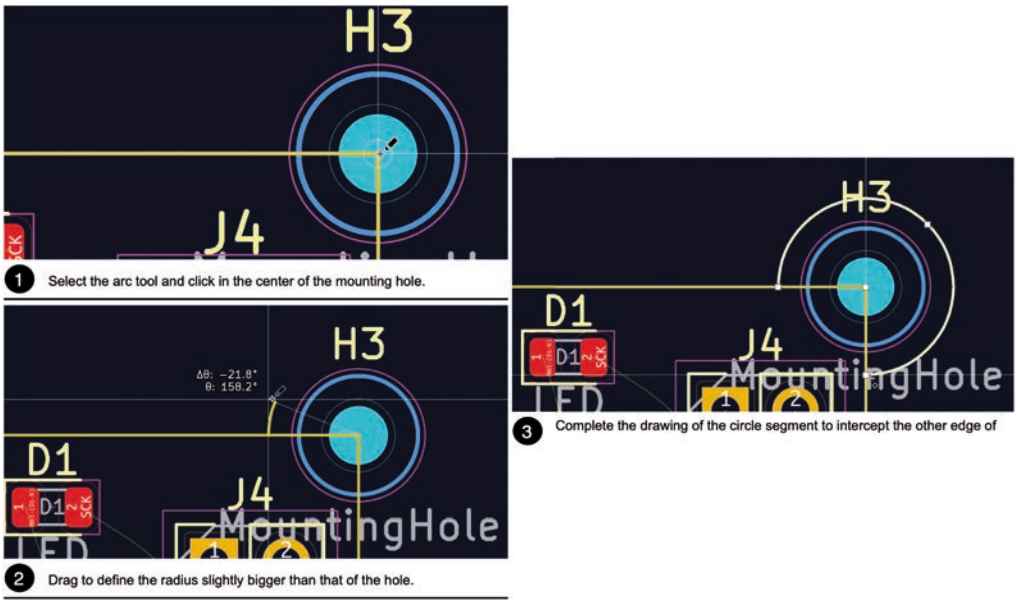


Figure 3.5.4.2: The final refined board outline.

Save the changes and commit them to the Git repository:

```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   MCU Datalogger.kicad_pcb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Refined and completed outline"
[2layer 7fe0eb7] Refined and completed outline
 1 file changed, 103 insertions(+), 34 deletions(-)
```

In the next step of the layout workflow, you will route the copper tracks on the top and bottom layers.

3.5.5. Layout 4 - Route

Manual board routing is usually the most time-consuming step in the layout workflow. It took me approximately one hour to route this board into two layers. This time includes several cases of undoing work that I had already completed to improve and optimize the

placement of tracks. I used the Interactive Router in "Walk around" mode for the entire routing process. Here is an outline of the strategy that I followed:

1. General rule #1: Draw signal and Vcc tracks in the top copper layer.
2. General rule #2: Use a copper fill connected to the GND net to connect all THT pads that belong to the GND net. Therefore, leave THT GND pads unconnected until the copper fills step (in the next segment in this chapter). You may follow the same rule for the Vcc pads and use a copper fill in the top layer connected to the Vcc net to route them.
3. General rule #3: Minimise the space between tracks to reduce unusable space. In the example below, I have placed the red tracks at minimum distances from each other ("1"), leaving enough space for at least one more track ("2").

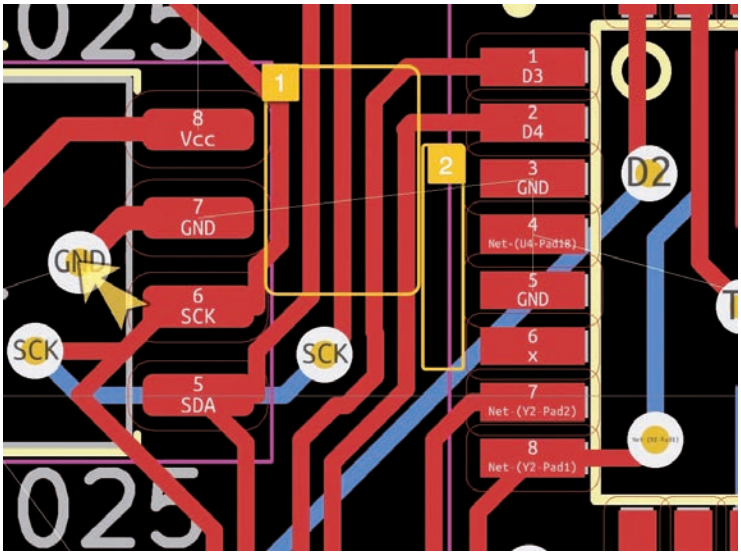


Figure 3.5.5.1: Don't waste space.

4. General rule #4: Exhaust your search for a path for a non-GND track in the top layer before using vias. You can often complete these tracks in the top layer by repositioning existing tracks.
5. General rule #5: For SMD pads that belong to the GND net and are not close to other GND pads, you can create a nearby via and connect it to the bottom layer copper fill. You can see an example of this below:

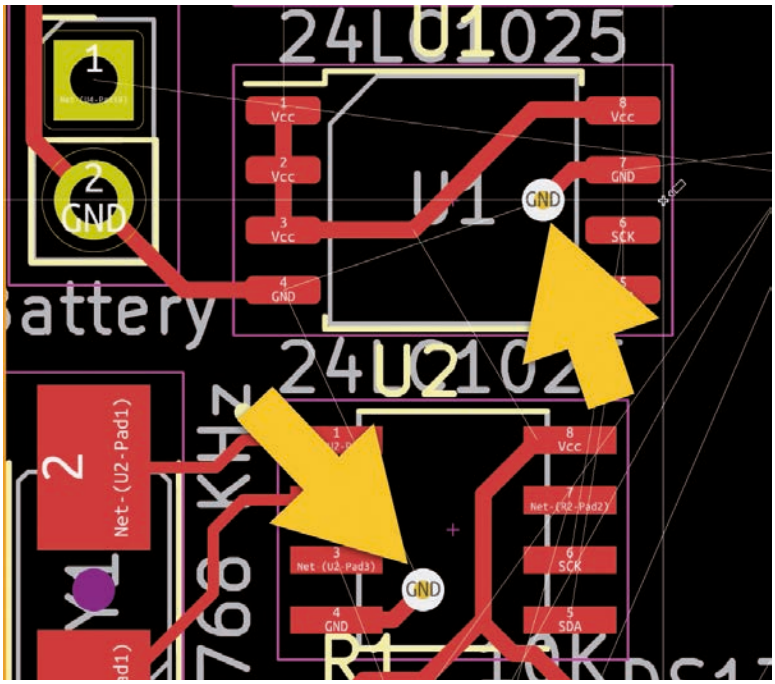


Figure 3.5.5.2: Using nearby vias to connect SMD GND pads to the GND copper fill in the bottom layer.

6. Start by drawing copper routes between adjoining pads. This includes Vcc and GND pads for the capacitors and resistors. Because most footprints in this PCB are SMD, I have drawn GND copper tracks in the top layer and used vias to connect them to the GND copper fill in the bottom layer. Here are a few example tracks that connect adjoining pads:

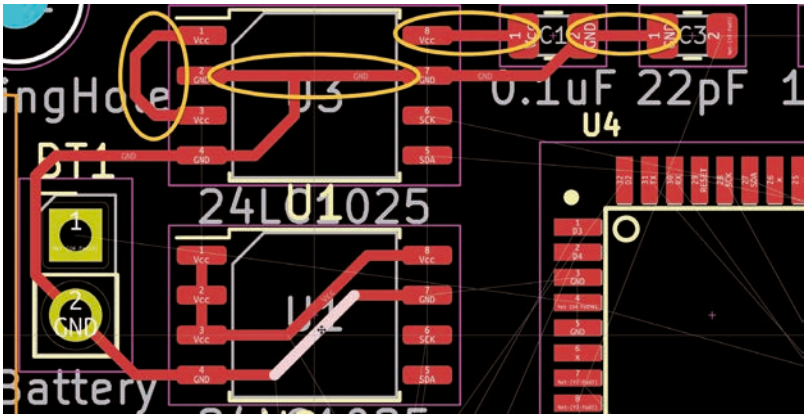


Figure 3.5.5.3: Tracks connecting adjoining pads.

7. Draw tracks connecting the peripherals to the central MCU footprint.
8. Draw the tracks to/from the connectors.
9. Connect a copper fill in the bottom layer to the GND net. This should complete the routing for all THT pads and vias that belong to the GND net (I'll do this in the next segment of this chapter).
10. Create a copper fill in the top layer and connect it to the Vcc net (I'll do this in the next segment of this chapter).

Because this board is more condensed than previous projects, it is more important to confirm the tolerances of your preferred manufacturer. I will be creating routes and vias as tightly as possible in the limited space available to work close to the manufacturer's tolerances. My preferred manufacturer is NextPCB, and their tolerances are available on their website (https://www.nextpcb.com/pcb-capabilities). I have set the constraints in the PCB editor's design rules to match or exceed NextPCB's tolerances. Here is my board setup:

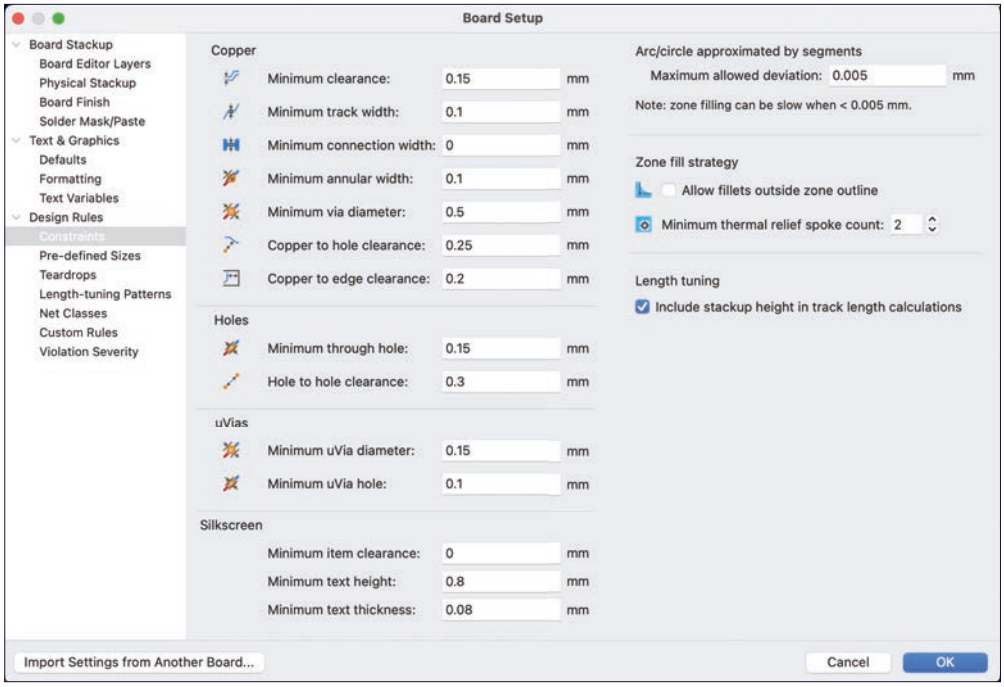


Figure 3.5.5.4: The constraints for this project match or slightly exceed NextPCB's tolerances.

After following this strategy, I produce this partially-routed layout:

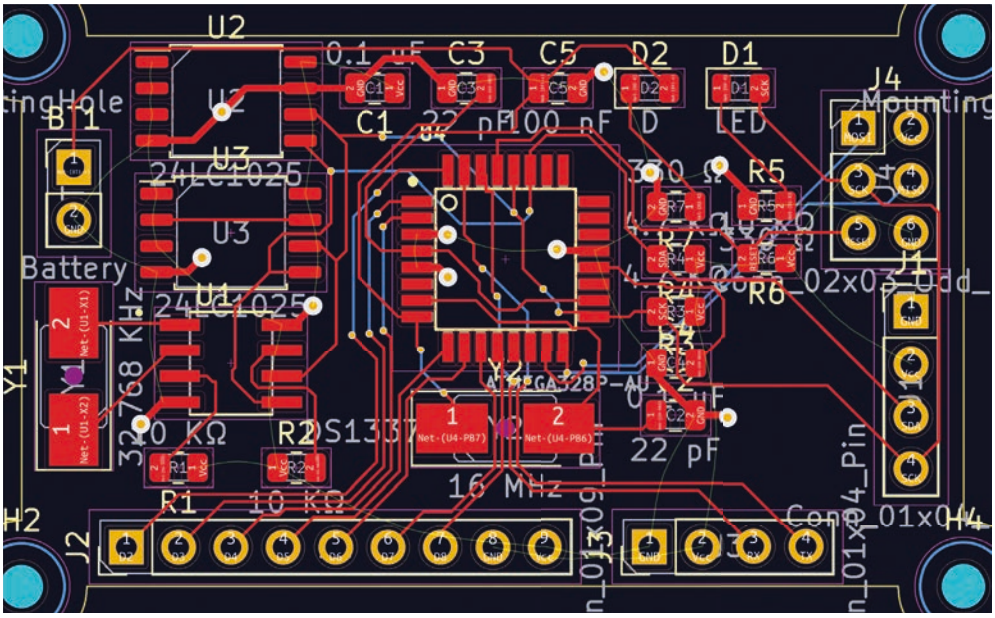


Figure 3.5.5.5: Partially routed PCB; GND and Vcc pads are not connected yet.

The missing tracks are those that complete the connections between the GND pads. My DRC check looks like this:

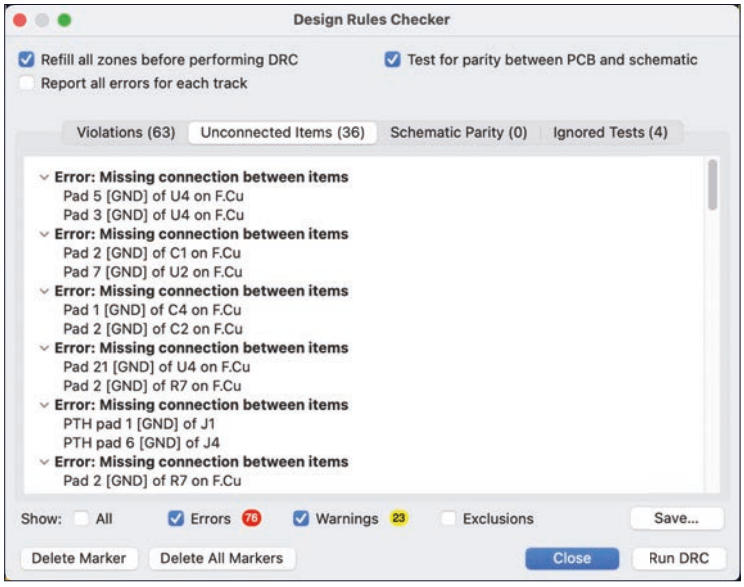


Figure 3.5.5.6: DRC shows unconnected GND pads and vias.

Indeed, the DRC shows that the only unconnected items are GND and Vcc vias and pads. I will address both issues in the next segment of this chapter.

Some of the tracks' shapes could also be improved. I will change the shape of a few tracks that contain 90-degree corners and optimize the use of space after creating the copper fills. Save your work, and commit the changes to the Git repository:

```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   MCU Datalogger.kicad_pcb
       modified:   MCU Datalogger.kicad_pro

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Routing sans GND Vcc complete."
git commit -am "Routing sans GND Vcc complete."
[2layer 901c3c6] Routing sans GND Vcc complete.
 2 files changed, 2282 insertions(+), 10 deletions(-)
```

Let's continue with the copper fills.

3.5.6. Layout 4 - Copper fills

You will create two copper fills in this step of the layout design workflow. You will place the first copper fill in the bottom copper layer and connect it to the GND net. Place the second copper fill in the top copper layer and connect it to the Vcc net. After creating these copper fills, the remaining unconnected item violations that the DRC reported in the previous segment of this chapter should be resolved.

Below is the Properties window for the first copper fill (bottom copper layer, connected to GND):

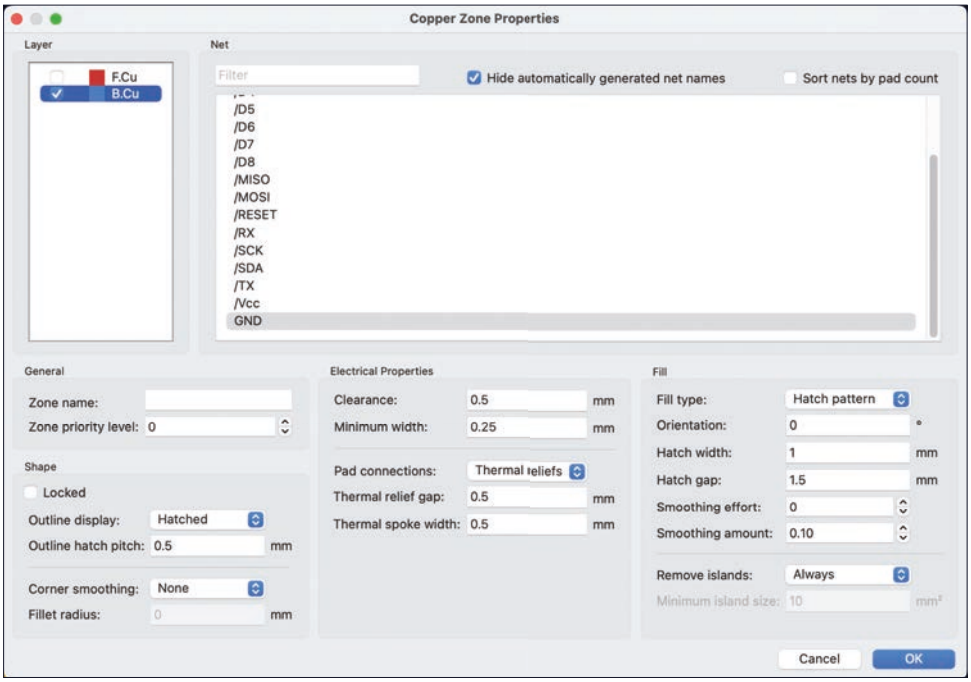


Figure 3.5.6.1: Copper fill properties for the bottom layer.

If you need a reminder on drawing a copper fill, please read the relevant chapter in Part 8 of this book. Go ahead and draw the copper fill in the bottom layer. For the copper fill in the top layer, I will be using a shortcut that can save a lot of time. Instead of drawing the new copper fill, I will duplicate it and connect it to the Vcc net. This way, the top and bottom copper layers will have the same shape. I describe this technique below:

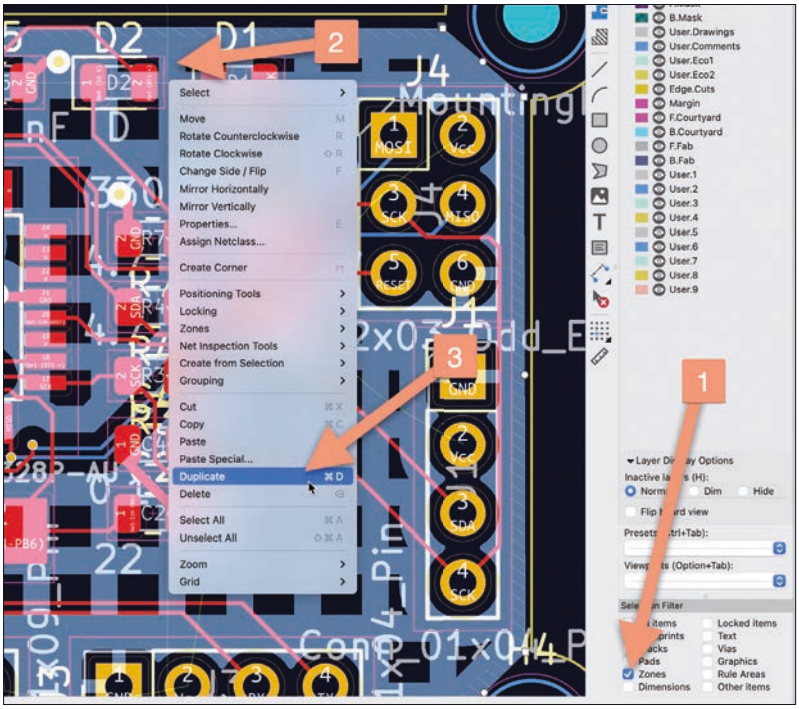


Figure 3.5.6.2: Duplicating a copper fill.

First, ensure that "Zones" is enabled in the Selection filter ("1", above).

Next, right-click on the border of the existing copper fill to bring up the context menu ("2") and click "Duplicate" ("3"). The new copper fill outline will appear attached to the cursor. Move the outline slightly outside the board so that the two copper fills don't overlap, and double-click on it to open its properties window.

Change the Layer setting to "F.Cu" and the Net to "Vcc," as in the example below:

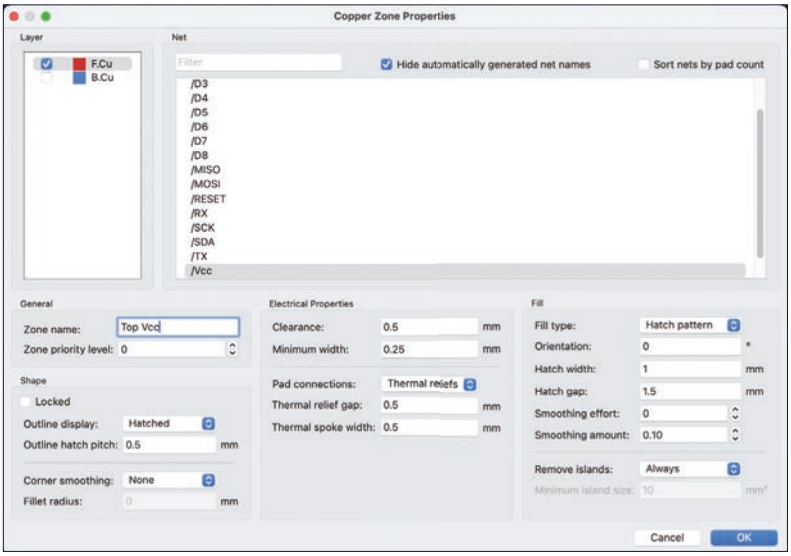


Figure 3.5.6.3: Copper fill properties for the top layer.

Click OK to close the Properties window and move the fill outline to overlap the bottom copper fill. See a detail of the two copper fills overlapping below:

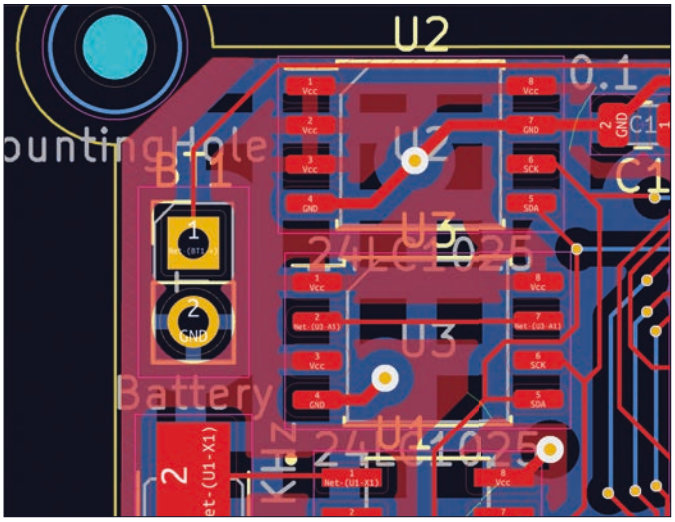


Figure 3.5.6.4: The two copper fills overlapping. Red is the top; blue is the bottom.

If your zones are not filled, fill them in using the "Fill All Zones" context menu:

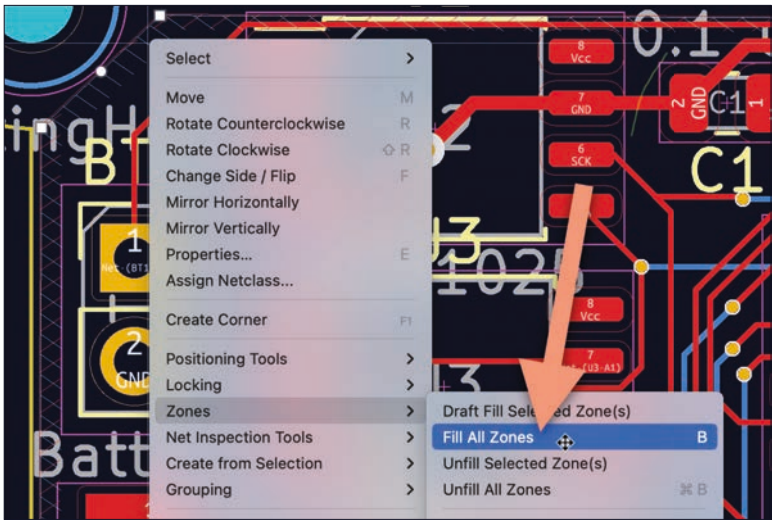


Figure 3.5.6.5: Fill all copper fills.

The result of this work is below:

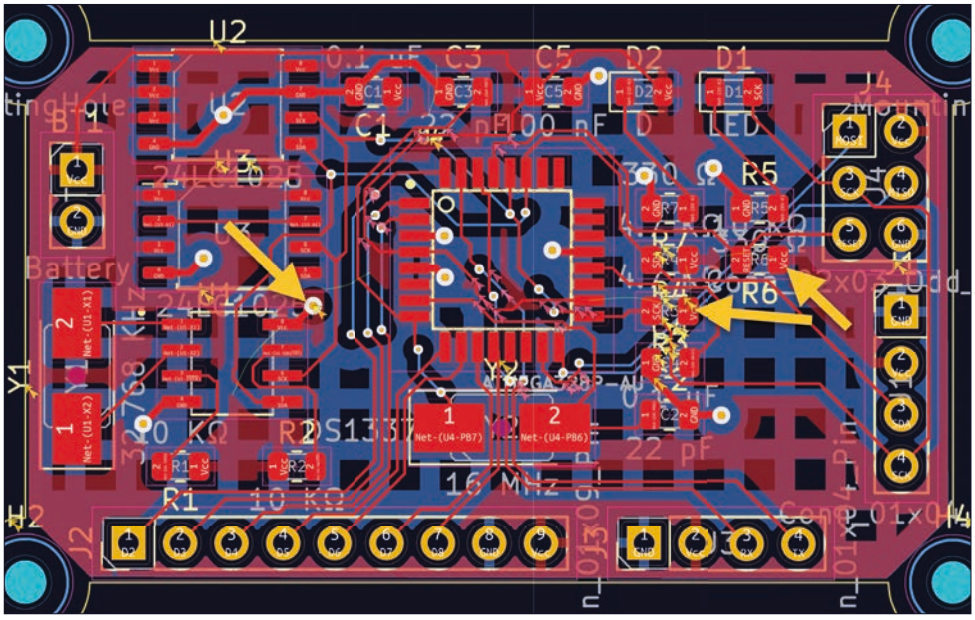


Figure 3.5.6.6: Top and bottom copper layers, filled.

After creating the copper fills, you may see that one or more Vcc or GND pads remain unconnected. This is usually due to the copper fill making segments on the top or bottom layers inaccessible; there is simply no gap large enough for the fill area to enter those segments.

In the figure above, I used arrows to highlight three unconnected Vcc pads. If this happens in your layout, you will need to change the position and shape of the existing tracks and vias to increase the available space through which the copper fills may be able to reach the unconnected items.

I had to spend thirty minutes reworking the tracks to finally make the copper fills reach all Vcc and GND pads. The result is below:

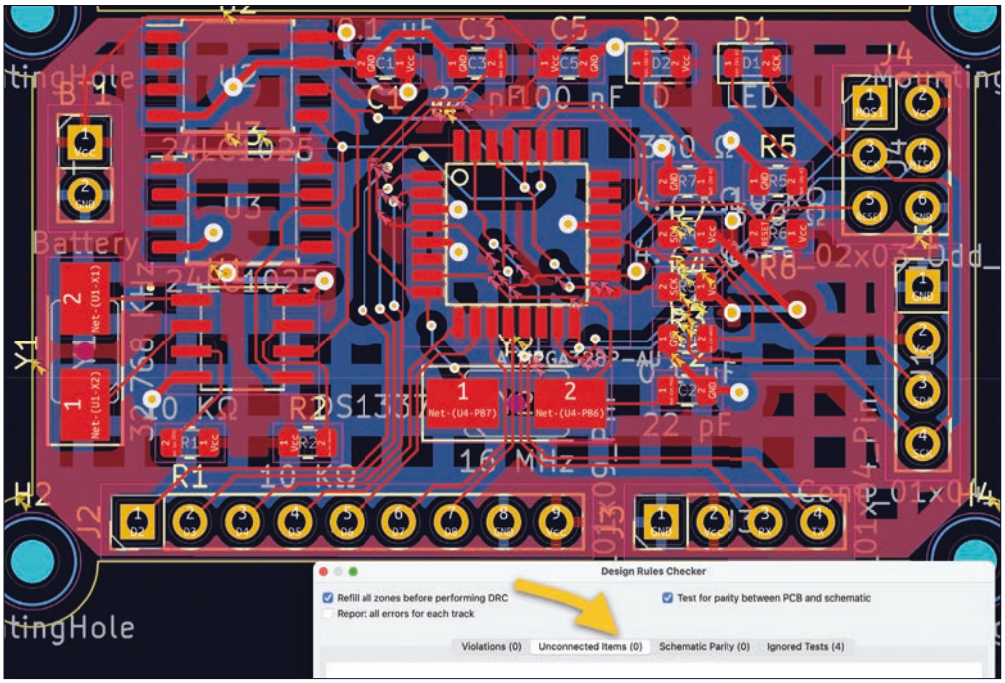


Figure 3.5.6.7: Top and bottom copper layers, filled, completed.

Save your work and commit changes to the Git repository:

```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   MCU Datalogger.kicad_pcb
        modified:   MCU Datalogger.kicad_pro
        modified:   MCU Datalogger.kicad_sch

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fp-info-cache
```

```
no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Completed wiring. Many improvements pending."
[2layer 2cd81d0] Completed wiring. Many improvements pending.
 3 files changed, 3176 insertions(+), 738 deletions(-)
```

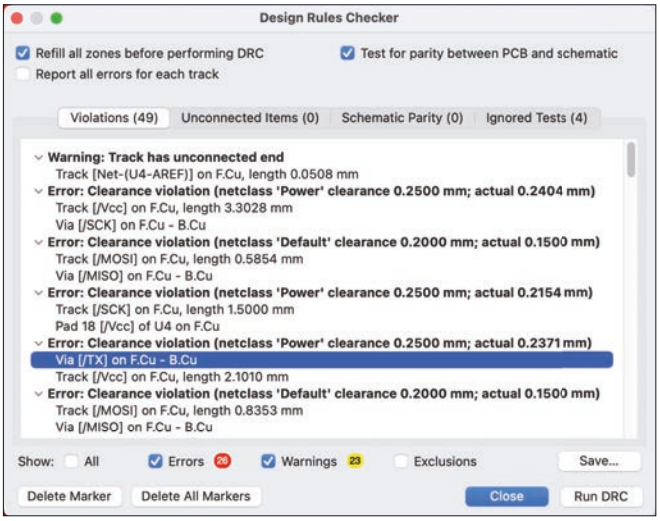
There is one autosave untracked file, which you can ignore, as autosave files do not need to be tracked. In the next segment, I will discuss some of my routing improvements and fixes to the board.

3.5.7. Layout 4 - Routing improvements

There is always scope for improvement. You can improve your board design in several ways. Below is my top-five list:

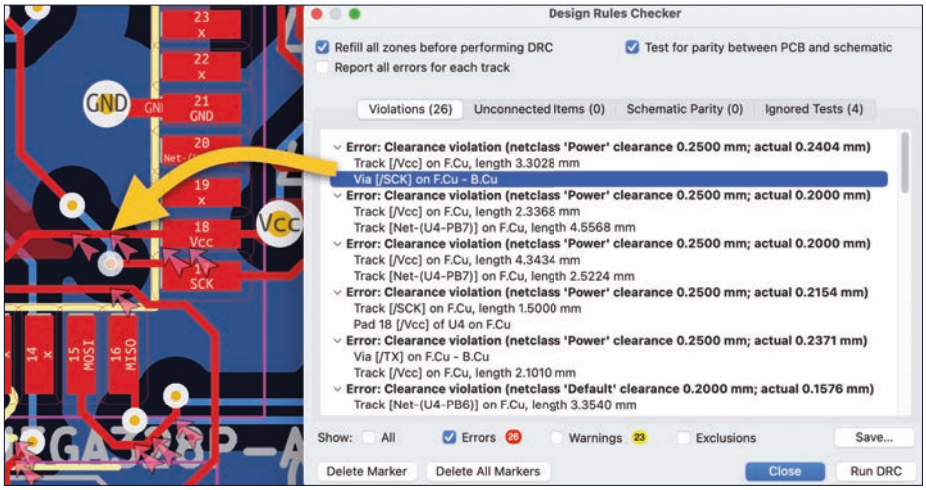
1. Resolve as many ERC errors and warnings as possible.
2. Optimize the positioning of the footprints.
3. Shorten the length of the copper tracks.
4. Improve the shape of the board.
5. Improve the shape of the tracks.

The first item in the least is the most important one. While in the previous segment I corrected all errors relating to unconnected GND and Vcc pads, the DRC still contains errors relating to clearances between tracks, pads, and vias violations. I will work to address and fix all of those errors first, before continuing with items 2 to 6. Below you can see some (out of 26) errors that I have to fix:



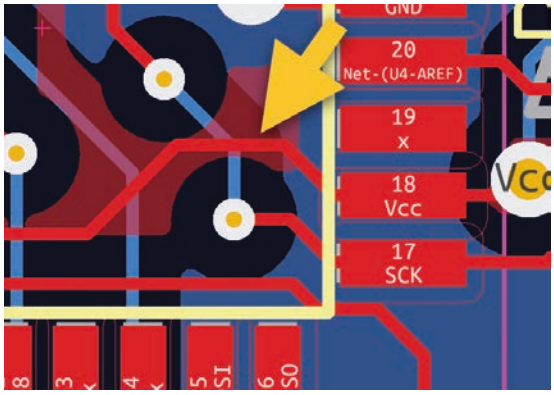
3.5.7.1: Some of the clearance violation errors to be addressed.

To fix these errors, you can do things such as move or redraw segments of tracks or vias. You may also need to move or redraw neighboring tracks or vias to make room. See an example case below:



3.5.7.2: An example clearance violation.

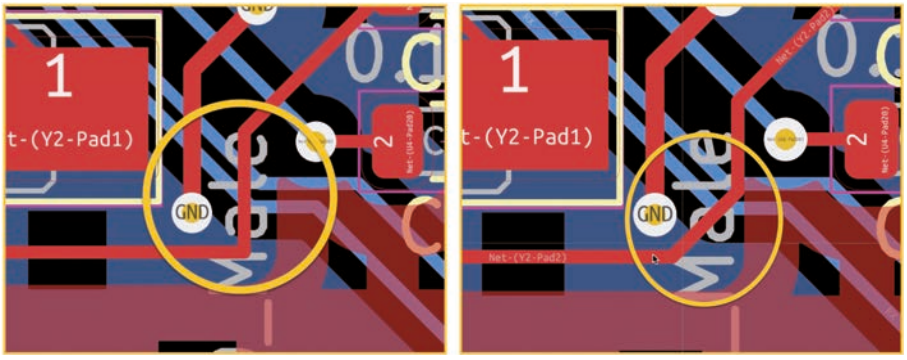
In the example of the screenshot above, the Vcc track the arrow points to is too close to the via below it. To fix such an error, I moved the via located above the Vcc track to make room and then redrew the Vcc track around the via below it so that the clearance minimum is satisfied. Below, you can see what this part of the PCB looks like after the redrawing:



3.5.7.3: Fixed the clearance violation be moving and redrawing elements.

Continue until all clearance violation errors are resolved, and then continue with the remaining items on the list. Remember to run the DRC occasionally to check for resolved and new issues. In summary, the improvements that I made are:

1. Replace any 90-degree corners with 45-degree corners. See an example here:



3.5.7.4: Removing 90-degree corners.

2. Fixed stray track segments, like the one below:

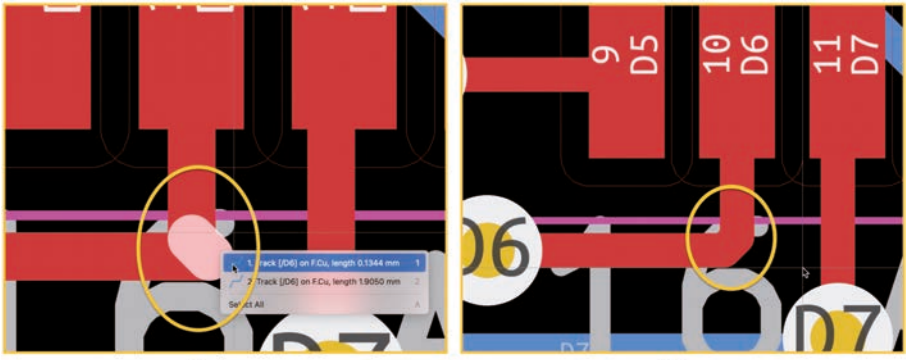
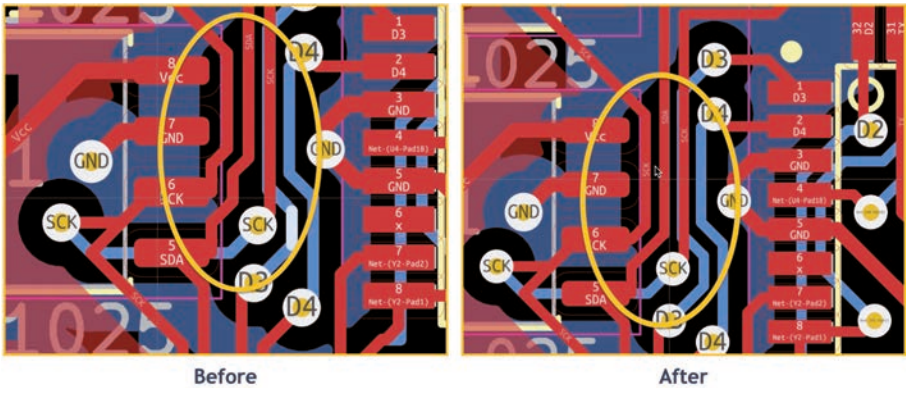


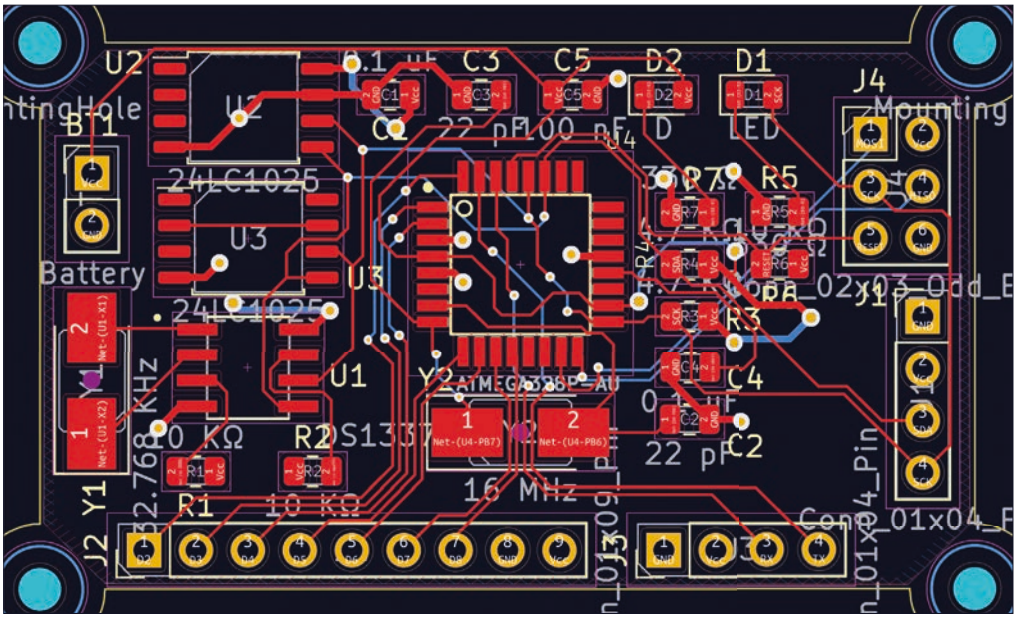
Figure 3.5.7.5: Removing a stray track segment.

3. Reduced the number of bends in a track, like in this example:



3.5.7.6: Reducing the twists and turns.

At the end of this process, my board layout looks like this:



3.5.7.7: Board layout with optimized routing.

You can continue with the optimization process forever. However, this is not practical, so at some point, you should stop and continue to the next step of the workflow.

Save your work and commit the changes to Git:

```
% git commit -am "Fixed all issues and warnings, and refined routing."
[2layer 78b0479] Fixed all issues and warnings, and refined routing.
 1 file changed, 1980 insertions(+), 1893 deletions(-)
```

Let's continue with the silkscreen.

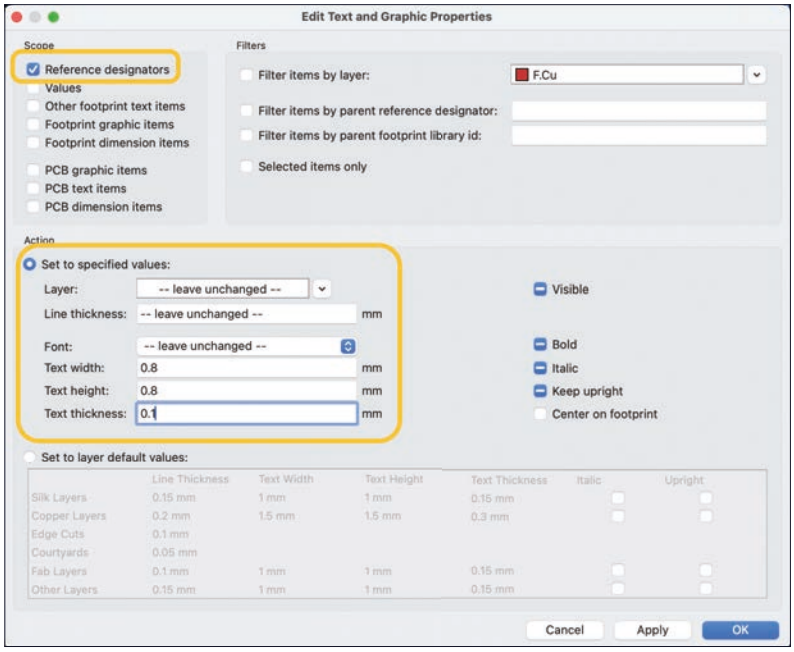
3.5.8. Layout 5 - Silkscreen

In step 5 of the layout workflow, you will add text and graphics in the top and bottom silkscreen layers. Because you will be working specifically in the top and bottom silkscreen layers, you can make invisible any items in layers, such as the front and back fabrication layers. This will remove text clutter in the editor. For the same reason, change the mode of the filled zone dedication to "Show only zone boundaries" and reduce the intensity of the tracks. All this will help you focus on the silkscreen layers and content.

I organize this work into the following parts:

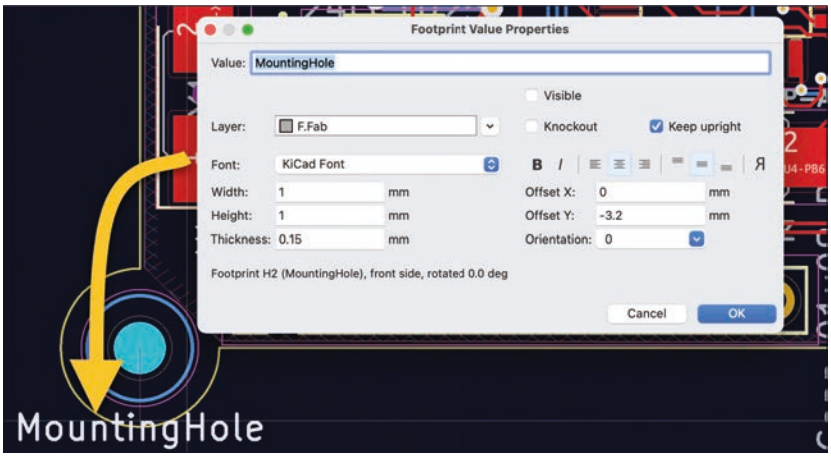
1. Place existing content (footprint values and references) in the correct layer and location. Use the bulk editing tools to standardize the size of this content.

In the example below, I use the "Edit Text and Graphic Properties" window to reduce the size of all reference designators so they can fit in the available space:



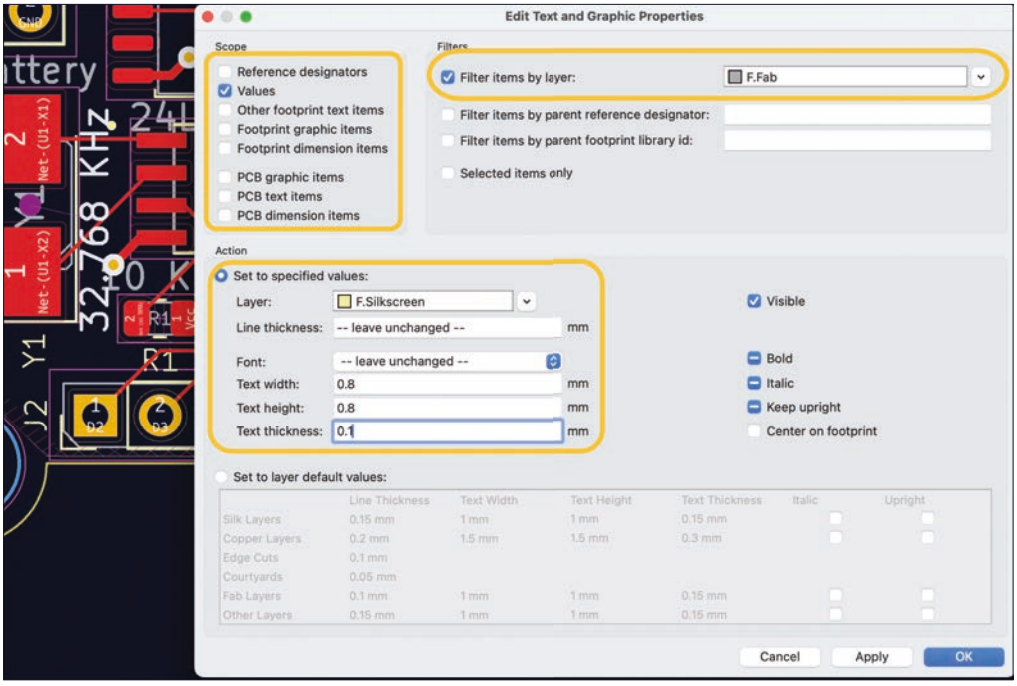
3.5.8.1: Reducing the size of the Reference Designators text.

2. Look for any visible content that should be made invisible and make it so. For example, there is no need to show the values for the mounting hole footprints. Uncheck their "visible" property in their Properties window. In the example below, I have done this for footprint value text, which belongs to one of the mounting holes:



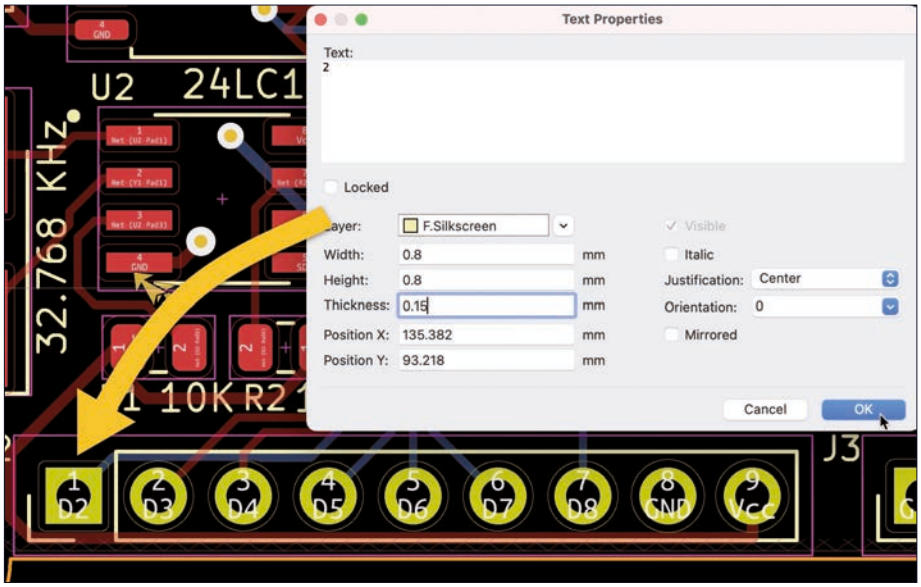
3.5.8.2: Made this text invisible.

3. Look for content in a non-silkscreen layer containing essential information and move it to a silkscreen layer. For example, the crystal oscillator footprints contain their operational frequency in one of the fabrication layers. I want to show this information on the front silkscreen. I have also made invisible component values for the capacitors and resistors to reduce clutter on the silkscreen. Below, I use the text bulk edit tool to do this for all values that exist in the F.Fab layer:



3.5.8.3: Bulk edit values in the F.Fab layer.

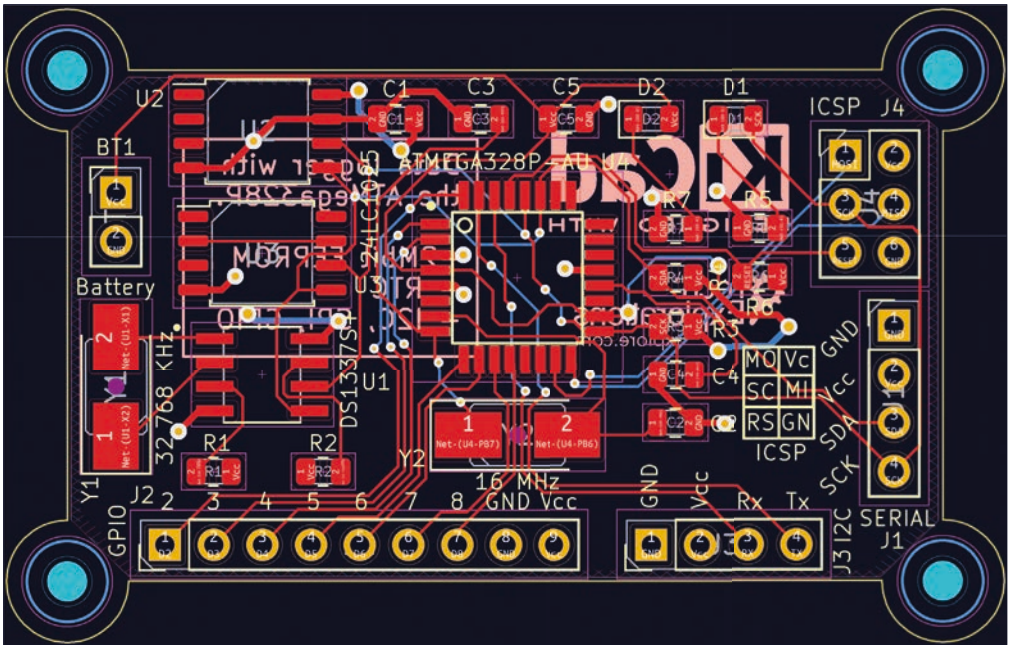
4. Add new text labels to convey pins roles, input voltage, etc. In the example below, I'm creating a text label with the content "2". I will place this label over pin D2 in header J2:



3.5.8.4: Creating a text label.

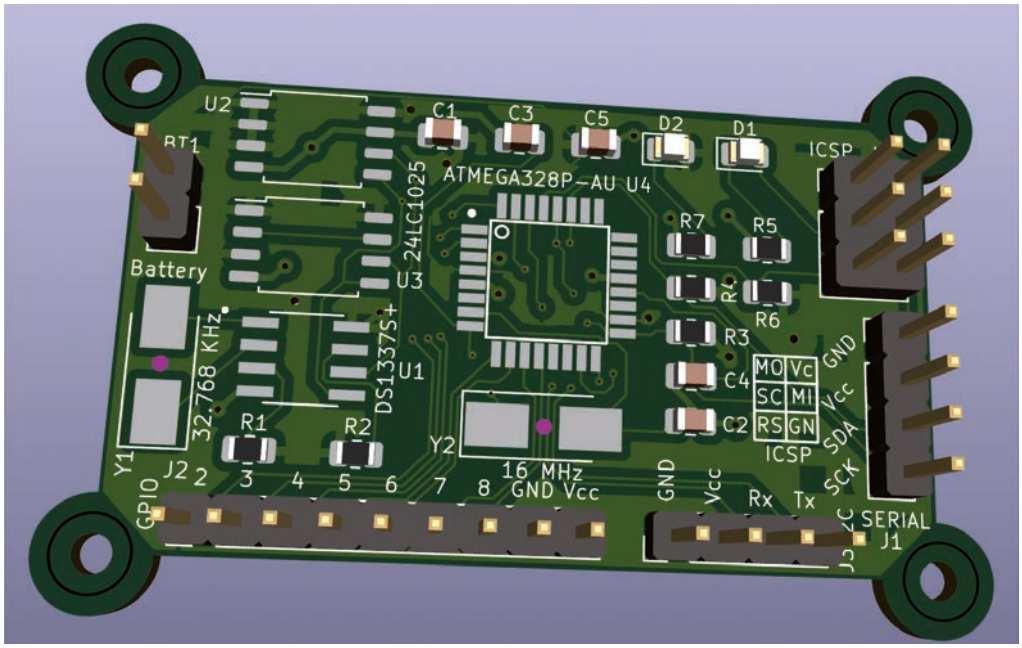
5. Add other elements, such as the board name, version number, logos, etc.

Below, you can see the latest version of my board, with a few outstanding issues that I will address later:



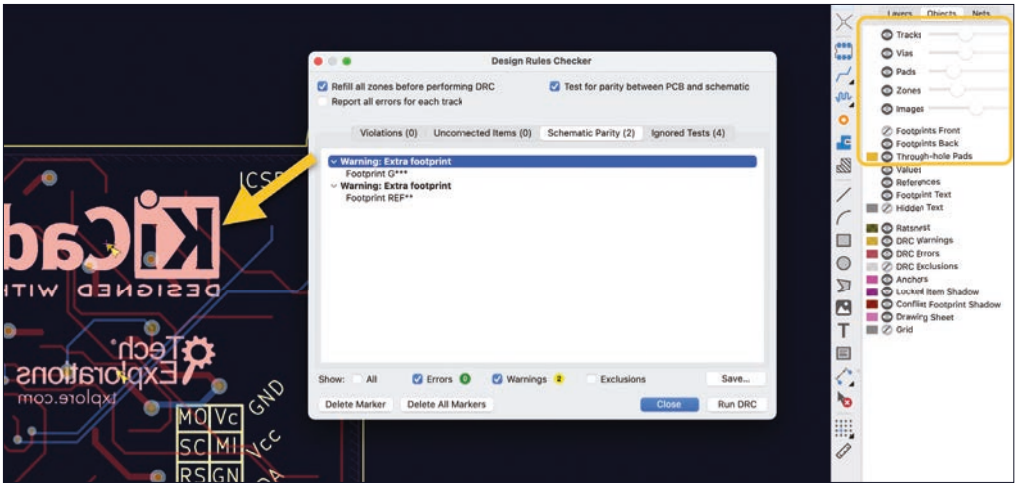
3.5.8.5: Making progress with the silkscreen elements.

Here is a 3D rendering of the front side of the board:



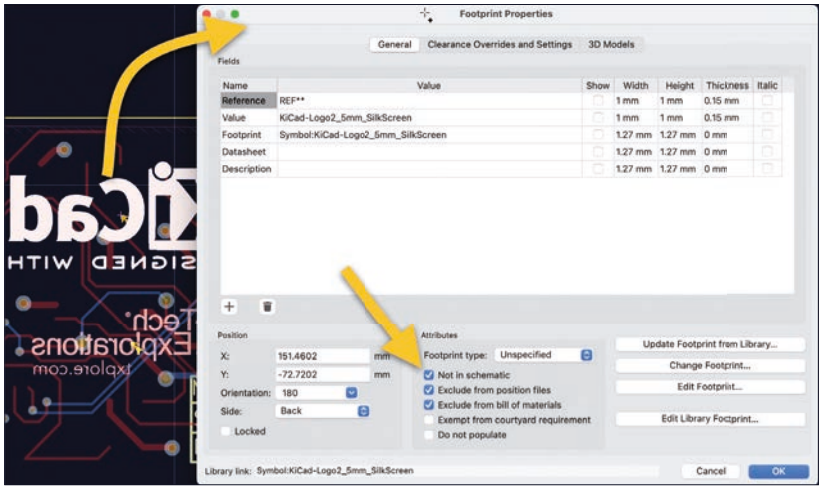
3.5.8.6: A 3D render of the front of the board showing the silkscreen progress.

The DRC shows two warnings that relate to the logo footprints. See the screenshot below. I have used the widgets in the Appearance pane to make the back footprints easier to see.



3.5.8.7: The two remaining warnings relate to the logo footprints.

Both are easy to fix. Let's fix the warning related to the KiCad logo. Double-click on the logo to bring up its properties. Under "Attributes", check the "Not in schematic" option and click OK.



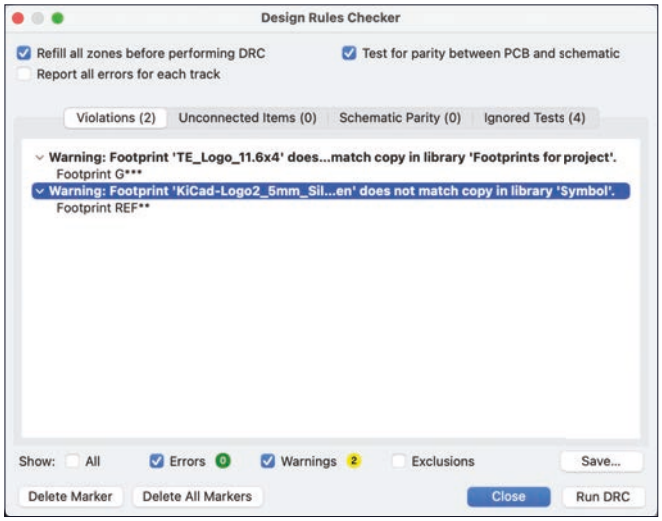
3.5.8.8: This footprint is not in the schematic, so mark it as such.

Repeat the process for other logos. In my case, I will mark the Tech Explorations logo as "Not in schematic." When you rerun the DRC, the schematic parity warnings should not re-appear. You will see two new violations, warning you that the logo footprints have been changed outside the library, which you can safely ignore.

Let's continue with step 6.

3.5.9. Layout 6 - Design Rules Check

You have done a lot of work on this PCB and are very close to passing the finish line. Before exporting the Gerber files, it is time for one last DRC to ensure that you haven't missed anything.



3.5.9.1: The last DRC reveals two warnings.

The last DRC reveals two footprint warnings. Both concern the logo footprints. In the last segment, you marked these footprints as "Not in the schematic," which caused them to differ from their original versions in the library. It is safe to ignore these warnings.

Let's save the changes and commit them to the Git repository:

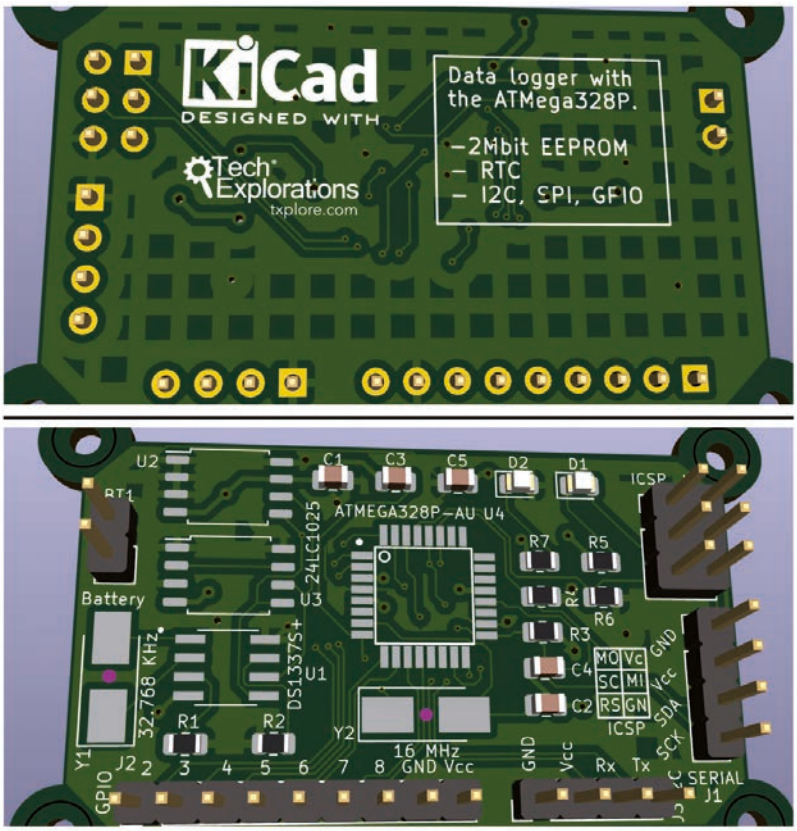
```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   MCU Datalogger.kicad_pcb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
      fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Ready to manufacture version 1."
[2layer d8da77b] Ready to manufacture version 1.
 1 file changed, 2204 insertions(+), 245 deletions(-)
```

I can now declare that this board is ready to export and manufacture.

3.5.10. Layout 7 - Manufacture

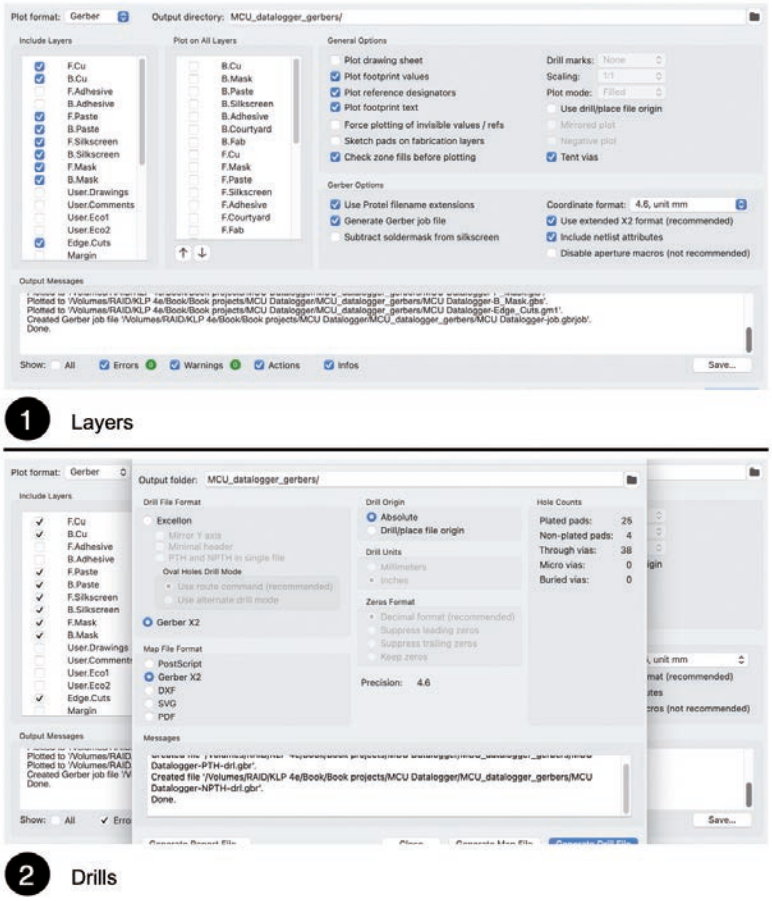
Let's finish the layout design workflow. In this segment, you will export the Gerber files for your new board and upload them to your preferred manufacturer's website.

Start by visually inspecting the front and back layers using the 3D viewer. Check for issues in the silkscreen, pads, copper fills, and missing or mal-positioned components.



3.5.10.1: Inspecting the board in the 3D viewer.

Save the design, and open the plotter window (see chapter "How to export and test Gerber files" if you need a refresher on exporting the Gerber files). You can see my settings for the Gerber and drill files below:

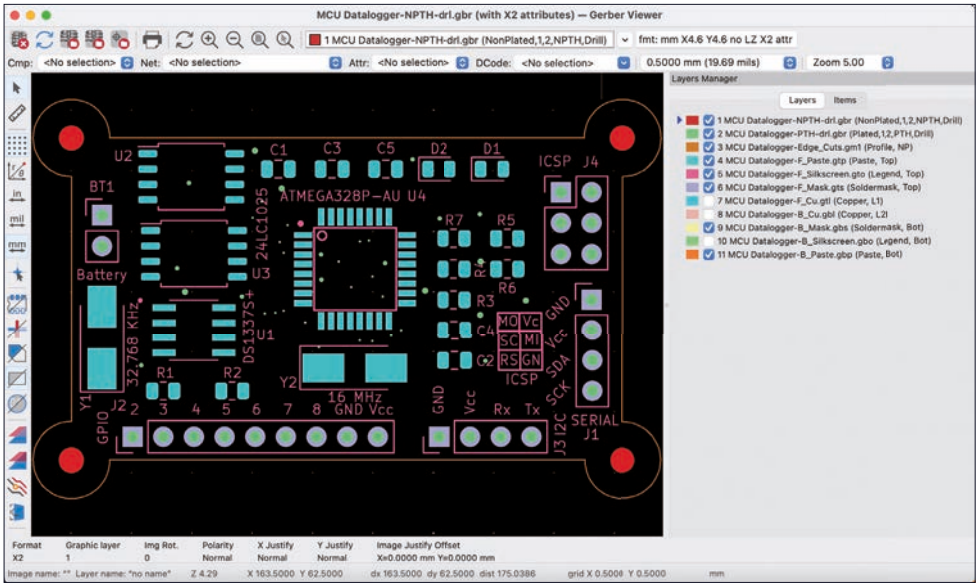


3.5.10.2: Generating the Gerber files.

Please beware: The Gerber settings on this page worked when I placed this specific order. Manufacturers may change their Gerber specifications from time to time. For this reason, you should check the manufacturer's website for the latest Gerber specifications. For example, here are pages that contain manufacturing information from two manufacturers:

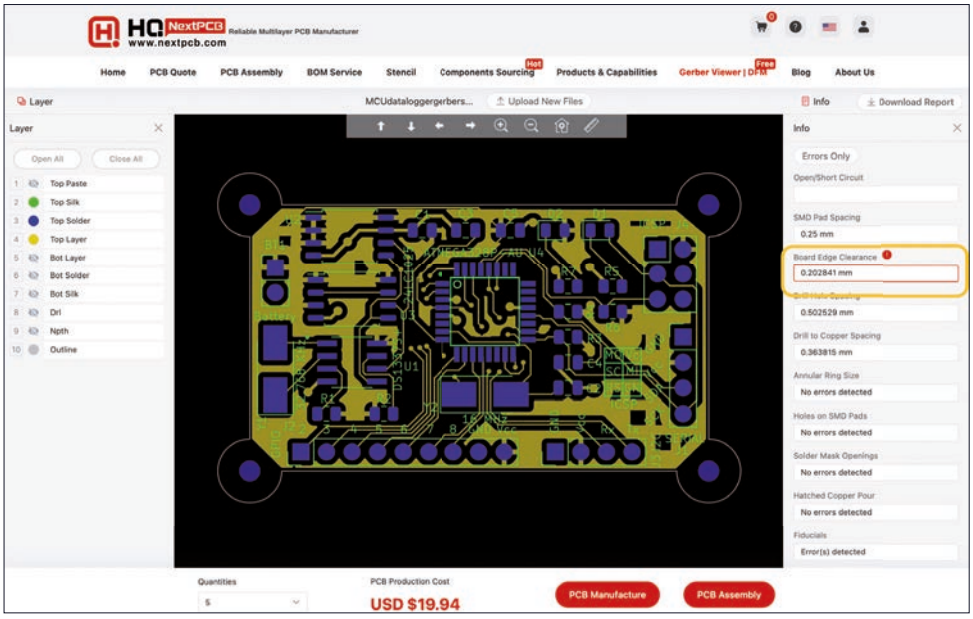
- Oshpark: https://docs.oshpark.com/design-tools/kicad/
- NextPCB: https://www.nextpcb.com/helpcenter/files-explanation

Open the KiCad Gerber viewer to inspect the new Gerber files. Enable and disable each layer and assess their validity. Is anything missing? Is something not looking correct? Now is the time to address any issue affecting the manufactured PCB.



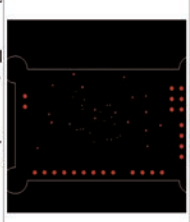
3.5.10.3: The Gerber viewer displaying the front silkscreen layer.

I saw no problems with the Gerber files in my instance, so I continued uploading them to the manufacturer. I will use NextPCB because of its excellent online viewer, which also does manufacturing analysis. I uploaded the Gerber ZIP archive to https://www.nextpcb.com/free-online-gerber-viewer.html and reviewed the analysis results. You can see the summary below:



3.5.10.4: The NextPCB online Gerber viewer and analysis report.

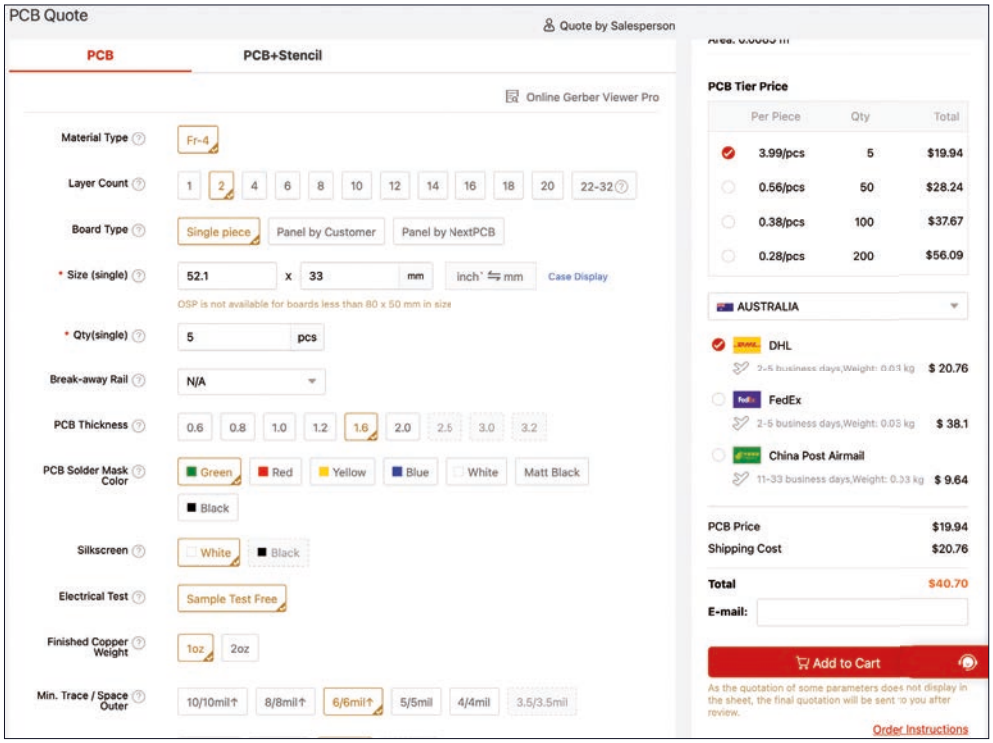
The manufacturing analysis shows potential problems with the board edge clearance and the drill size (the latter is not shown in the screenshot above). The PDF report can be downloaded by clicking the Download Report button in the online viewer. On page three of the report, I can see details about the two errors:

ID	item	rule	value	issue	image	Coordinate	count	level
1	Hole Diameter_Smallest Drill Size	12,10,8	0.20 mm	For most factories, the smallest (mechanical) drill hole size requirement is around 0.2-0.25 mm. Smaller drill bits need to be replaced frequently and are more likely to result in missing holes, misalignment and hole wall roughness, which decrease manufacturing efficiency and yield, and affect the reliability of the boards. 0.1 mm and smaller drill holes require laser drilling and have strict board width requirements. Drill holes 0.20mm in diameter were detected in your design. The diameter should be increased to at least 0.2 mm, or 0.3 mm to avoid additional costs.		141.18,-78.58	1	Risk
2	Board Edge Clearance_Copper-to-Board Edge	8,15,20	0.20 mm	For most factories, the copper-to-board edge clearance requirement is 0.2 mm. Failure to meet the factory's requirements could increase the risk of exposed copper on the edge of the boards or damaged traces/pads, which decrease manufacturing efficiency and yield, and affect the reliability of the boards. Copper-to-board edge spacing of 7.99mil was detected in your design. It is recommended to increase the spacing to at least 0.25 mm for edge routing and 0.4 mm for v-cuts (v-cut spacing may depends on board thickness).		153.70,-93.88	10	Risk

3.5.10.5: The NextPCB online Gerber viewer and analysis report.

I have marked the most exciting parts with yellow rectangles. For both issues, I have used values that precisely match the minimums NextPCB supported. The minimum drill size is 0.20 mm, and my vias have a drill hole of 0.20 mm. The minimum clearance between the board edge and the copper is 0.20 mm, and again, there is one instance on the board that matches that minimum. Based on this information, I have decided to continue with my order. If I had exceeded these minimums, I would return to the PCB editor and make adjustments.

To proceed with the order, click the red PCB Manufacture button at the bottom of the Gerber viewer. A new tab will appear, with the PCB and its information pre-populating the order form. For this board, I will use NextPCB as the manufacturer. You can see the order form below after I have entered the board dimensions:



3.5.10.6: My PCB order form (partial).

I have accepted the defaults and submitted the order. Done! There is one last thing to do before you complete the two-layer PCB project: commit your latest changes to the Git repository. Here is my command line session:

```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   MCU Datalogger.kicad_pcb
        modified:   MCU Datalogger.kicad_pro

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        MCU_datalogger_gerbers.zip
        MCU_datalogger_gerbers/
```

```
        fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Completed 2–layer PCB project."
[2layer fc88c49] Completed 2-layer PCB project.
 2 files changed, 4 insertions(+), 4 deletions(-)
```

In the Git command line session above, you can see that Git detected new untracked files in the "MCU_datalogger_gerbers" directory. I chose not to track those files because they are not needed. Gerber files can always be generated; keeping them in the repository would only increase bloat.

At this point, the project is complete. However, there are two more things that I would like to show you. First, how to add the missing 3D shapes so that the 3D render of the board looks as realistic as possible. Second, how to use the same schematic to produce a four-layer version of the board. This will also allow you to practice your Git skills further.

3.6. 3D shapes

In this chapter, you will improve the realism of the 3D viewer. Currently, the 3D viewer renders the board where only the resistors, capacitors, and pin headers have 3D models. Let's address this by adding the 3D models for the remaining footprints.

The figure below shows the current 3D viewer rendering (left) and its look at the end of this chapter (right).

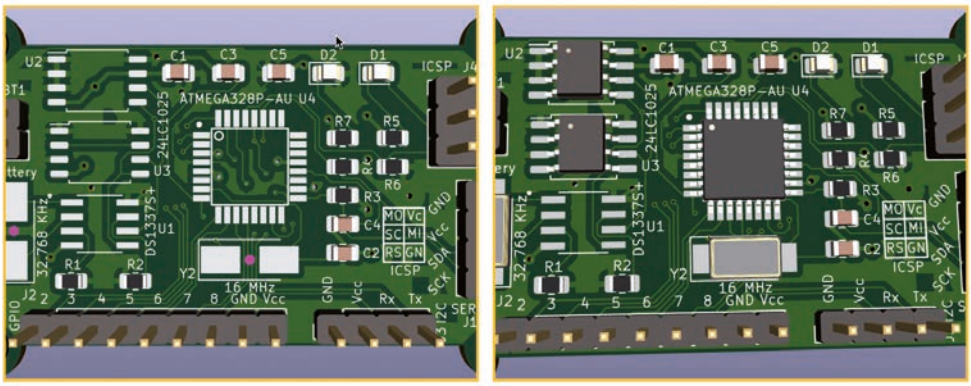


Figure 3.6.1: "Before" and "After" renders from the 3D viewer.

To add the missing 3D models, you must open the properties window for each related footprint. These footprints are missing a 3D model:

- U4 (the MCU).
- U3, U1, U2 (the two EEPROM chips and the real-time clock).
- Y1 and Y2 (the two crystal oscillators).

Please read the relevant chapter if you need a reminder about adding a 3D model to a footprint. For the MCU footprint, I used the TQFP-32_7x7mm model. This model is available in the "Package_QFP" library with KiCad. You can see my settings below:

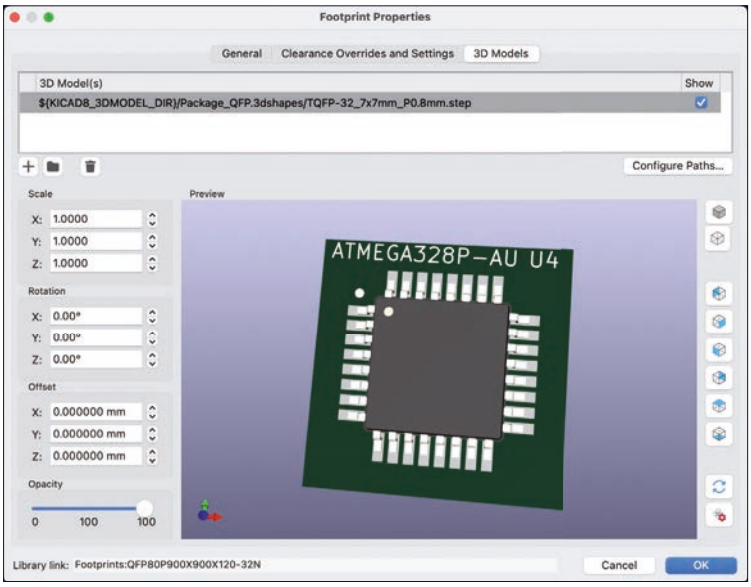


Figure 3.6.2: Settings for the MCU 3D model.

Continue with U2 and U3. The 3D model I used for this footprint is the "SOIC-8_3.9x4.9mm_P1.27mm.wrl" in the "Package_SO" library. See my configuration below:

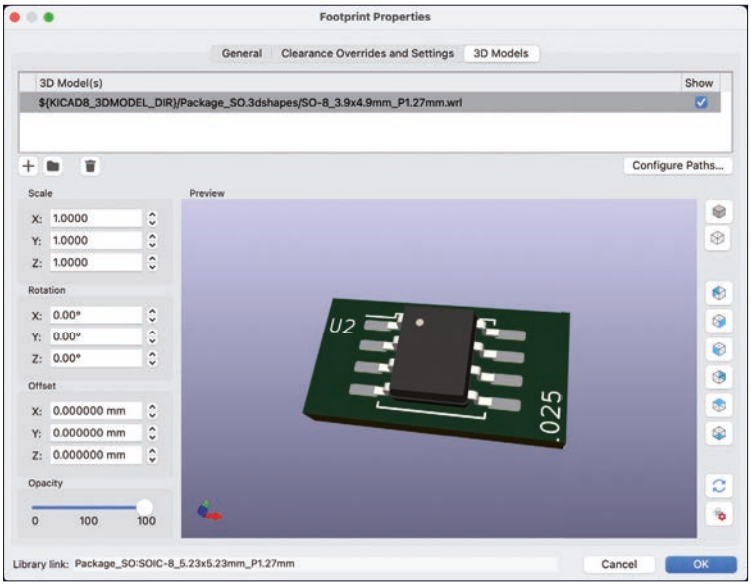


Figure 3.6.3: Settings for the EEPROM chip 3D model.

I used the same model for the second EEPROM chip and the real-time clock. Copy the 3D model path and file and paste it into the 3D model's row for U2 and U3.

Two models remain for the crystal oscillators. For Y2, use model "Crystal_SMD_0603-2Pin_6.0x3.5mm" in the "Crystal" library:

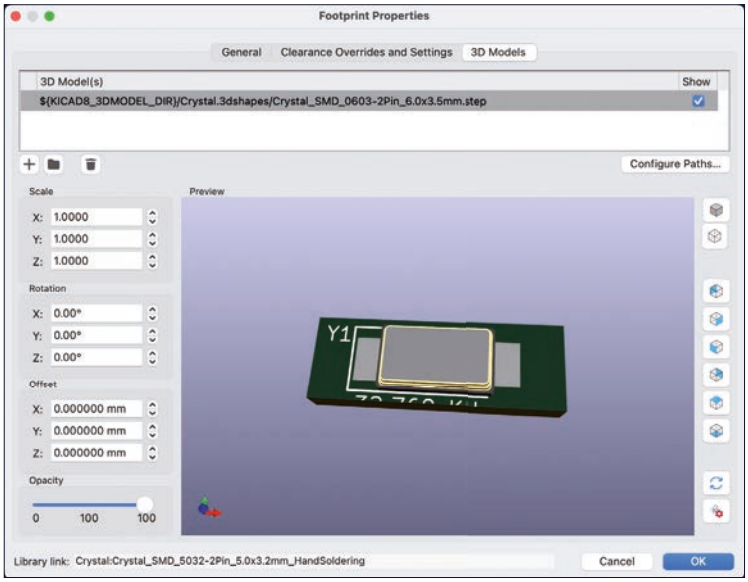


Figure 3.6.4: Settings for the Y1 crystal oscillator.

For Y2, use the same model as in Y1 to copy the model path across.

All footprints now have a 3D model. Here's the resulting 3D rendering:

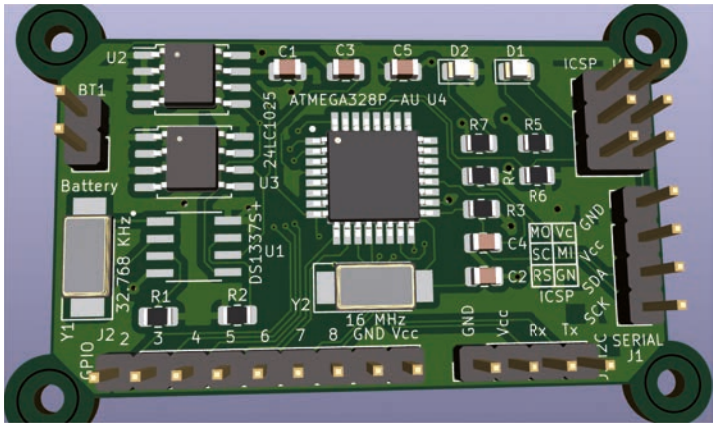


Figure 3.6.5: 3D render of the board with all 3D models assigned.

The 3D viewer can now produce a more realistic rendering of the board.

3.7. Merge 2-layer branch to main

In this chapter, you will merge the "2layer" Git repository branch into "main." The "2layer" branch contains all changes relating to your work in the two-layer PCB. In the next chapter, you will begin work on a four-layer version of the PCB. But first, merging "2layer" into "main" will ensure that "main" contains the latest version of the schematic. I plan to create a new working branch for the four-layer PCB, which will branch out of "main."

Another benefit of merging "2layer" into "main" is that the merge will also bring all two-layer layout changes into "main." This means the new "4layer" branch will inherit the layout from the "main" branch and the refined outline in the Edge.Cuts layer, placement of the footprints, and even the 3D models. To design the four-layer version of the PCB, you will only need to delete the current copper tracks and fills, change the board layout to four layers, and redraw tracks and fills.

Before working with Git, I suggest one naming convention change. Currently, the name of the folder that contains the two-layer Gerbers is "MCU_Datalogger_Gerbers." Change this to "2_layer_MCU_datalogger_gerbers". Soon, you will have a second Gerbers directory called "4_layer_MCU_Datalogger_Gerbers". You will not track these folders in Git, but they will co-exist in your project directory.

Let's continue with Git. Ensure that you have saved all your work. Open your terminal window. Check the current status of the repository with the "status" command:

```
% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   MCU Datalogger.kicad_pcb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       2_layer_MCU_datalogger_gerbers/
       MCU_datalogger_gerbers.zip
       fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
```

You can leave the Gerbers directory untracked. Commit the three changed and tracked files to the working branch:

```
% git commit -am "Added missing 3D models."
[2layer 6a5ad61] Added missing 3D models.
 1 file changed, 17 insertions(+), 6 deletions(-)
```

The "2layer" branch contains all changes. Continue to merge the changes in 2layer with the main branch. You can do this in two steps:

1. Switch your working branch to "main" using the "checkout" command.
2. Merge "2layer" into "main" using the "merge" command.

Here is my Git command line session:

```
% git checkout main
Switched to branch 'main'
% git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
       2_layer_MCU_datalogger_gerbers/
       MCU_datalogger_gerbers.zip
       fp-info-cache

nothing added to commit but untracked files present (use "git add" to track)
% git merge 2layer
Updating 1e2e322..6a5ad61
Fast-forward
 #auto_saved_files#       |     1 -
 MCU Datalogger.kicad_pcb | 14497 ++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++-
 MCU Datalogger.kicad_pro |   221 +-
 MCU Datalogger.kicad_sch |    11 +
 4 files changed, 14713 insertions(+), 17 deletions(-)
 delete mode 100644 #auto_saved_files#
```

Until you switch to a different branch, you will be working in the main Git branch. To avoid Git reminding me to track the Gerber directories and ZIP archives, I will add them to the ".gitignore" file. Below, you can see the updated contents of this file (please see this recipe for a detailed and up-to-date discussion of .gitignore):

```
.DS_Store
*.lck
fp-inFo-cache
*.kicad_prl
MCU Datalogger-backups/*
2_layer_MCU_datalogger_gerbers/*
4_layer_MCU_datalogger_gerbers/*
*_Gerbers.zip
```

Save the updated ".gitignore" file, and commit the change to the ".gitignore" file to the main branch:

```
% git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   .gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Added gerber directories and ZIP archives to .gitignore."
[main e2bd5a7] Added gerber directories and ZIP archives to .gitignore.
 1 file changed, 3 insertions(+), 1 deletion(-)
```

In the next chapter, you will begin work on the four-layer version of the PCB and create a new working branch to track changes in the layout.

3.8. Design 4 Layer PCB in new Git branch

Let's begin work on the four-layer version of the PCB by creating a new branch in the project's Git repository. Currently, the repository contains two branches. The current working branch is "main" (identified as active by the asterisk). The "2layer" branch contains changes relating to the two-layer version of the PCB. The two branches are in sync, and there are no uncommitted changes.

You can use the "branch" command to get information about repository branches:

```
% git branch
  2layer
* main
```

Create a new branch to track changes in the four-layer version of the PCB. The name of the new branch is "4layer". Here is my command line session:

```
% git checkout -b 4layer
Switched to a new branch '4layer'
% git status
On branch 4layer

nothing to commit, working tree clean
%
```

You are now working in the "4layer" branch, which is clean (i.e., there are no pending changes" and tracking for changes. In the next chapter, you will go straight into the PCB editor, change the board configuration to four layers, and remove and redraw all copper tracks and fills.

3.9. Four-layer PCB routing

Because the layout work in the two-layer PCB has already been committed to the Git repository, most of the design work is already complete. Layout workflow steps 2 (outline and constraints), 3 (component placement), and 5 (silkscreen) are complete and do not require any modifications.

This chapter will re-work steps 1 (setup) and 4 (routing and copper fills). You will also do a DRC (step 6). In the next chapter, you will export the new Gerber files, check them for fitness, and upload them to the manufacturer.

Let's begin. Open the PCB editor. First, remove all tracks and fills using the Global Deletions window from the Edit menu:

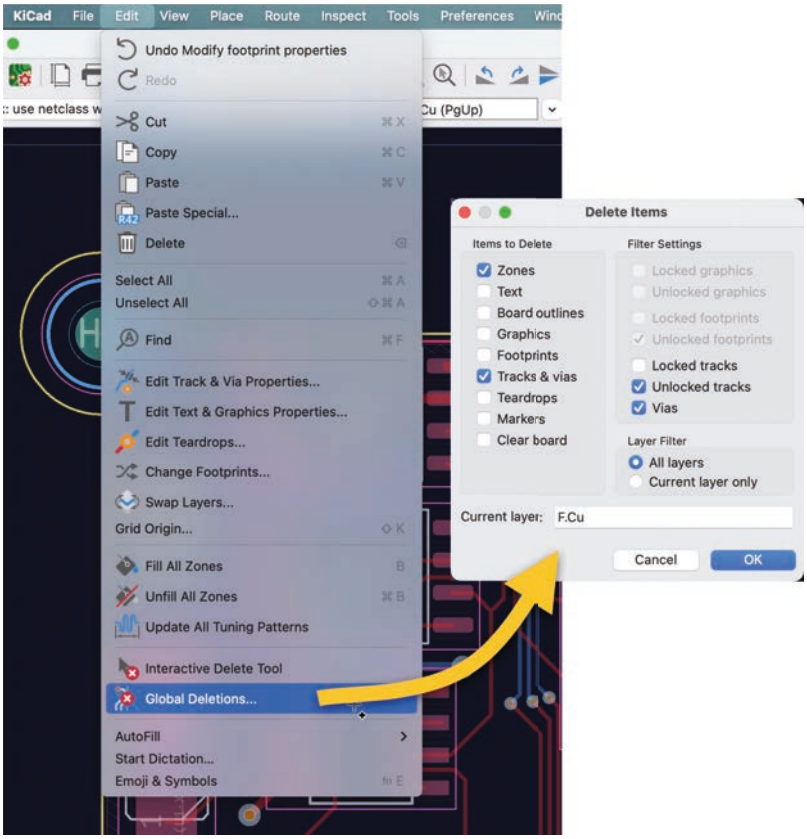


Figure 3.9.1: Delete all tracks, vias and zones.

The layout will retain all other design elements. Here is the current state of the board (I have disabled the silkscreen layers to reduce clutter):

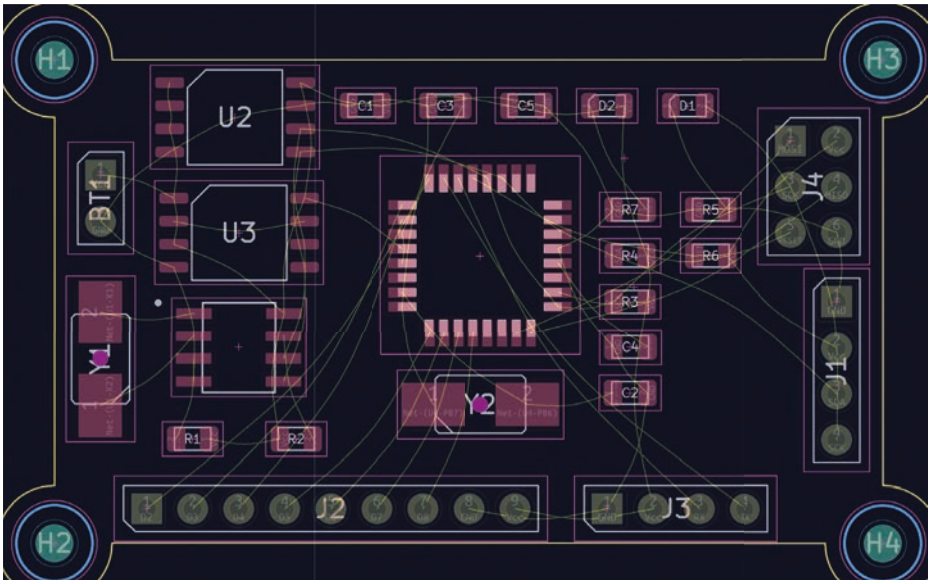


Figure 3.9.2: The board without tracks, vias, and filled zones.

In the Board Setup window, under Board Stackup, Physical Stackup set the number of copper layers to 4:

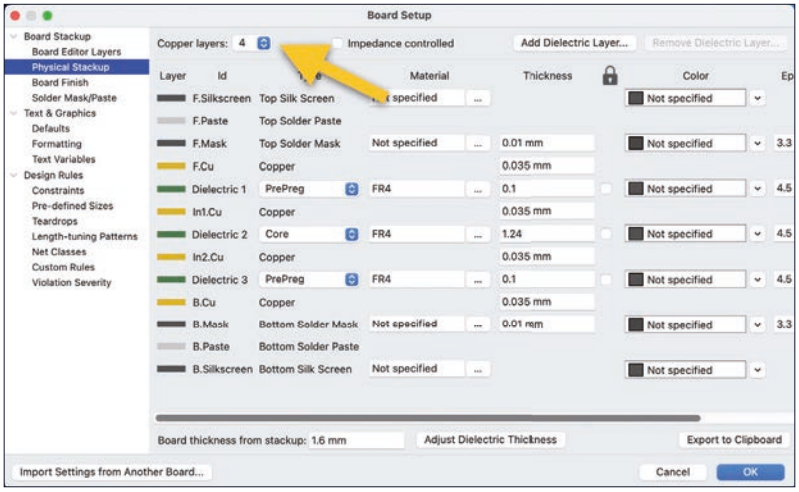


Figure 3.9.3: Switch the number of copper layers to "4".

My routing strategy for the four-layer board is similar to the two-layer board. I aim to draw the route signal routes (all routes other than those that belong to the Vcc and GND nets) in the top, In1, and In2 copper layers.

I will draw the first one connected to the GND net in the bottom layer and the second connected to the Vcc net in the In1 layer for the copper fills. You can see the resulting four-layer PCB below:

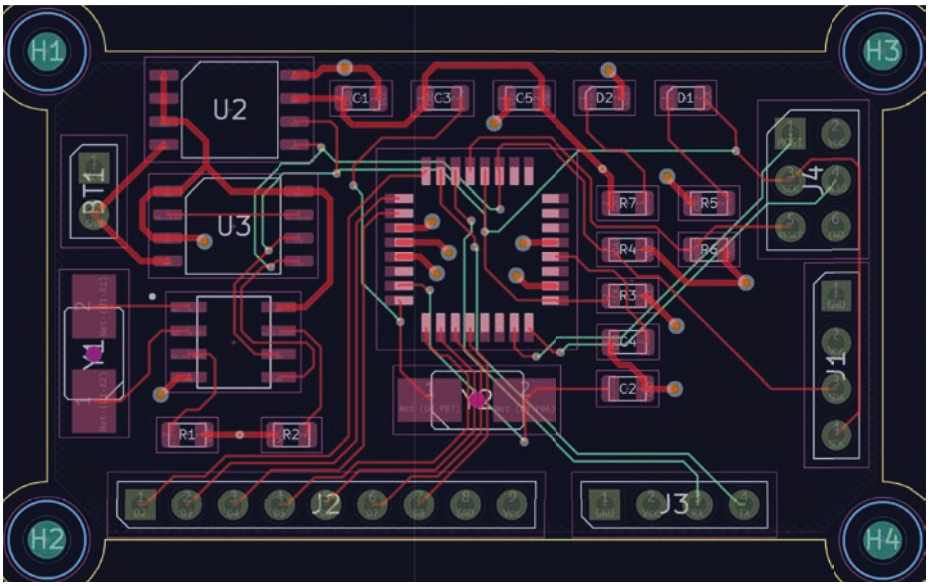


Figure 3.9.4: The completed four-layer PCB.

Do a DRC to ensure that the board does not contain any actionable violations. Save the new layout, and commit the changes to Git:

```
% git status
On branch 4layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   MCU Datalogger.kicad_pcb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit –am "Completed 4-layer pcb routing."
[4layer 6108a4e] Completed 4-layer pcb routing.
 1 file changed, 3151 insertions(+), 3055 deletions(-)
```

Let's continue manufacturing the board.

3.10. Four-layer PCB manufacturing

Let's manufacture the PCB. I will be using the same settings as the two-layer PCB, except I will now choose four layers instead of 2 in the order form. The board dimensions and tolerances remain unchanged.

Generate the Gerber files:

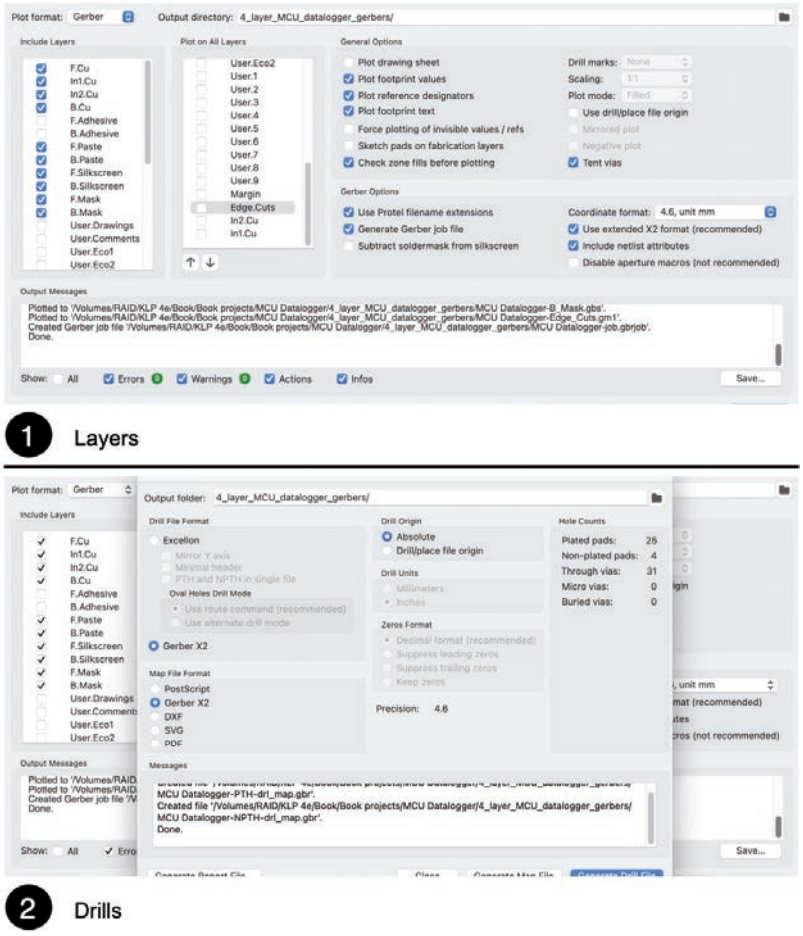


Figure 3.10.1: Generating the Gerber files.

Test the generated Gerber files using the KiCad Gerber Viewer app:

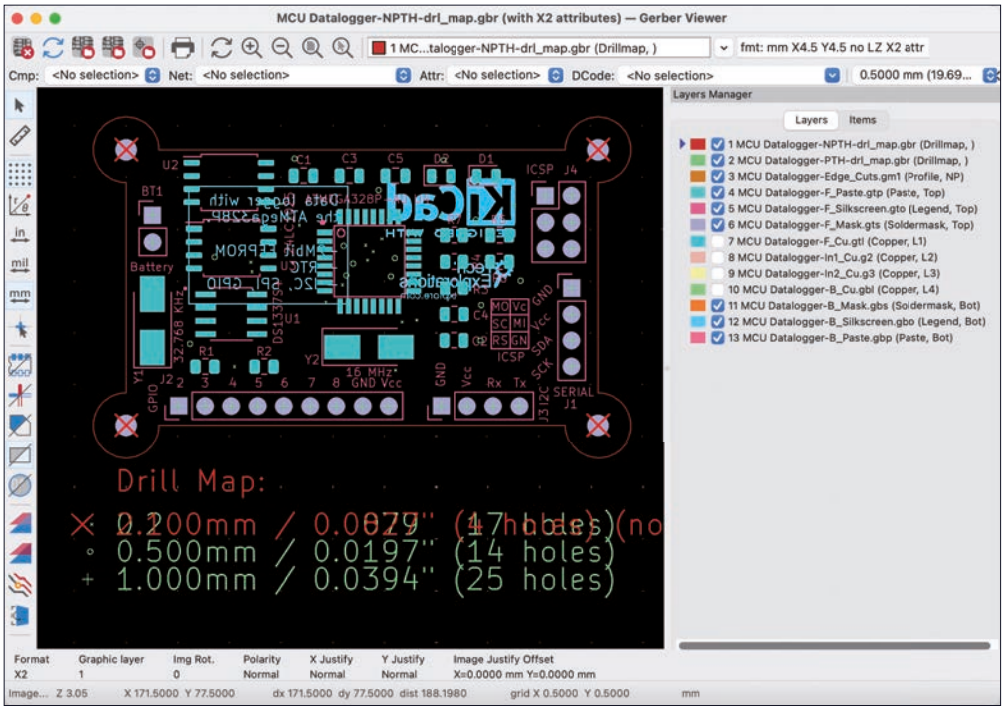


Figure 3.10.2: Testing the Gerber files in Gerber Viewer.

Finally, upload the Gerber files ZIP archive to the manufacturer's website. Again, I'll use NextPCB's online Gerber viewer and manufacturing analysis tool to ensure I haven't missed anything in my new PCB.

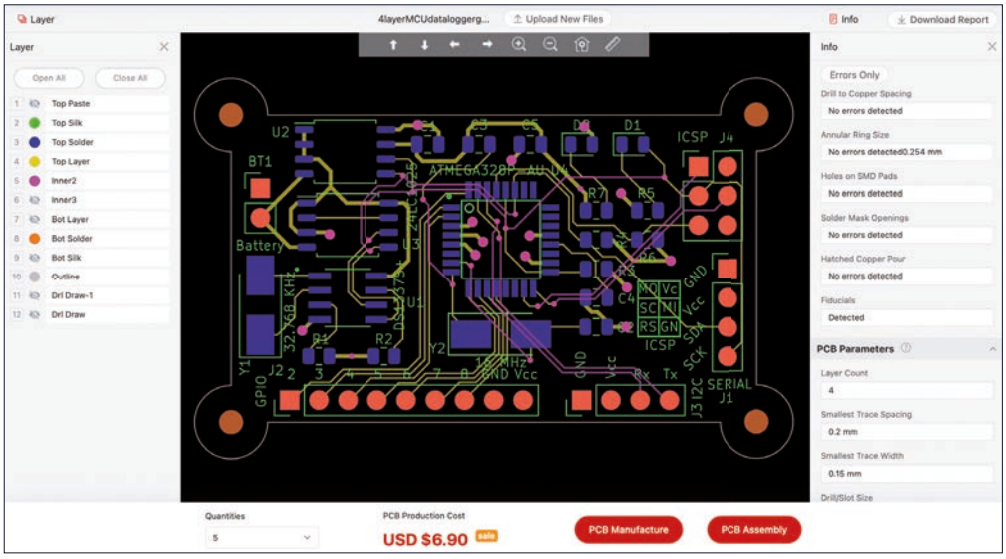


Figure 3.10.3: The four-layer PCB in the NextPCB online Gerber viewer and analyzer.

The manufacturing analyzer shows no errors! The full report PDF confirms that there are 27 check items passed, with zero warnings and risk items. Perfect!

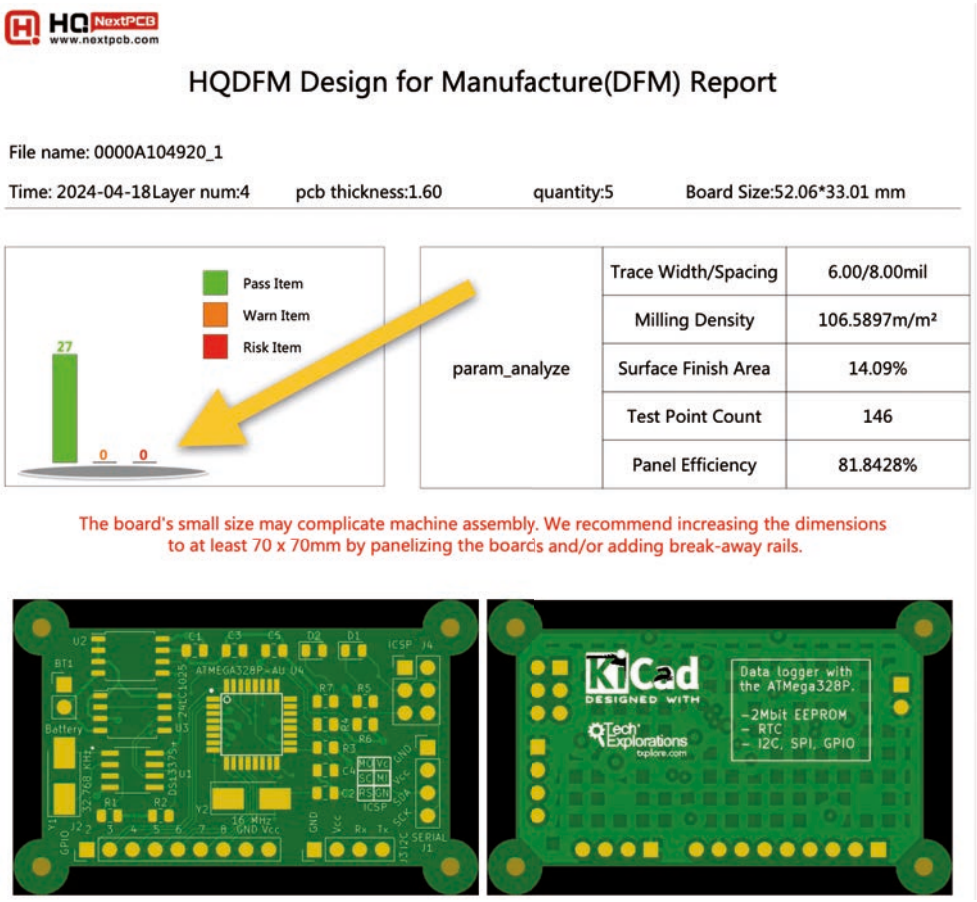


Figure 3.10.4: A PCB with zero manufacturing bugs!

Let's finish this project by committing the changes to the "4layer" Git branch (save changes in the editor if you haven't done so already):

```
% git status
On branch 4layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   MCU Datalogger.kicad_pcb
      modified:   MCU Datalogger.kicad_pro

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
        fp-info-cache

no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Completed 4-layer PCB."
[4layer 39e0b3a] Completed 4-layer PCB.
 2 files changed, 2 insertions(+), 2 deletions(-)
```

With this Git commit, this project is complete. However, I want to demonstrate one more skill in the next chapter. Imagine the situation where you need to change the schematic after you have completed work on the two versions of the PCB layout. At the file level, this change will exist in the schematic file of the repository's main branch. You will then need to sync the changes in this file to the other two branches ("2layer" and "4layer") without modifying the layout files.

How can you do this? I will show you in the next chapter.

3.11. Updating layout from changes to the schematic with Git

In this chapter, I will demonstrate a solution to the problem of using Git to import changes to the schematic into layouts that exist in different branches. This problem is common when a KiCad project has a single schematic but more than one layout. To solve this problem, you will need to use Git. One day, KiCad may be able to support multiple layouts from a single schematic, but this is not yet possible in KiCad 8.

The MCU datalogger project you completed has a single schematic in the main branch. The layout has two versions: a two-layer layout in the "2layer" branch and a four-layer layout in the "4layer" branch. A slightly different scenario would call for a single schematic informing an SMD and a THT version of a two-layer board.

In either case, you will likely want to change the schematic in newer project versions, such as adding new components or fixing a bug. The individual schematic file will need to be synced between the main branch and the layout branches.

You already know how to merge two branches with the "merge" command. The problem with this option is that the" merge" command would merge the schematic file and all files between the branches involved. This is risky, as it could overwrite the layout files of one version of the PCB with that of another.

There is a safer option, which is to use Git to merge-specific files between branches. This gives you maximum flexibility and removes all risks. In the case of the current project, it is possible to use Git to merge changes to the "MCU Datalogger.kicad_sch" file only between branches and ignore everything else. The Git feature I will use in the following example is called "gitattributes" (see https://www.git-scm.com/docs/gitattributes). You can use ".gitattributes" in an extensive range of situations, but in this case, I will use it to set up a merge strategy to allow me to merge a single file between branches. Let's look at an example using the MCU datalogger project, as we left it in the previous chapter. Begin by getting the status and the current list of branches in the repository:

```
% git status
On branch 4layer
nothing to commit, working tree clean
% git branch
  2layer
* 4layer
  main
```

I am currently working on the "4layer" branch, and the repository contains three branches. Say that you want to make a change to the schematic. First, switch to the "main" branch:

```
% git checkout main
Switched to branch 'main'
%
```

Next, create a new file with the name ".gitattributes." You can see the content of this file below (a single line):

```
MCU Datalogger.kicad_pcb merge=pcb
```

This line informs Git to ignore files with "MCU Datalogger.kicad_pcb" during a merge operation. You may also use a wildcard character in the name, if you prefer, such as "*.kicad_pcb," which would cause Git to ignore all files with this extension. The name of this merge rule is "pcb." This is an arbitrary name; you can change it if you wish.

Merge this new file into the main branch:

```
% git add .
% git commit -am "Added gitattributes file to preven PCB Files from merging."
[main 167e733] Added gitattributes file to preven PCB Files from merging.
 2 files changed, 92570 insertions(+)
 create mode 100644 .gitattributes
 create mode 100644 fp-info-cache
```

Open the schematic editor and make a change to the schematic. I have added a new net label in one of the un-named nets, as you can see below:

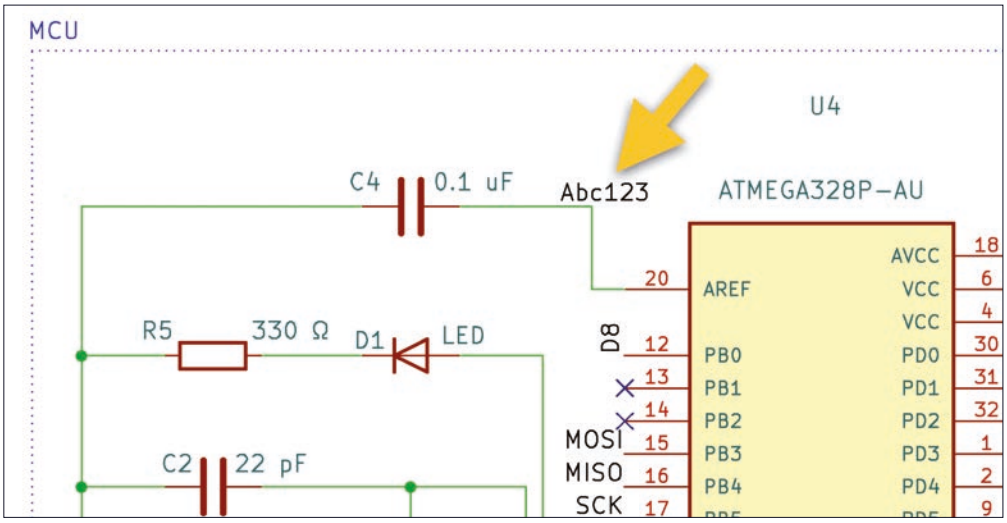


Figure 3.11.1: Made a change to the schematic.

Save the change and commit it to the "main" branch:

```
% git commit -am "Made small change to schematic."
[main 6db6883] Made small change to schematic.
 1 file changed, 11 insertions(+)
```

Now, you would like to update the four-layer PCB with the latest changes in the schematic. Switch your working branch to "4layer":

```
% git checkout 4layer
Switched to branch '4layer'
```

Next, merge the changes in "main" into "4layer". Git will allow you to type in a commit message. I typed this: "Merging an update to the schematic" and then saved the text editor to complete the merge operation.

```
% git merge main
Merge made by the 'ort' strategy.
 .gitattributes           |     1 +
 MCU Datalogger.kicad_sch |    11 +
 fp-info-cache            | 92569 ++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++
 3 files changed, 92581 insertions(+)
 create mode 100644 .gitattributes
 create mode 100644 fp-info-cache
```

Two files were merged. The merge contained the new ".gitattributes" file and the changes in the schematic file ("MCU Datalogger.kicad_sch"). Confirm that the change in the schematic editor appears in the "4layer" branch. Close the schematic editor if it is still open, then re-open it. Confirm that the new net label appears:

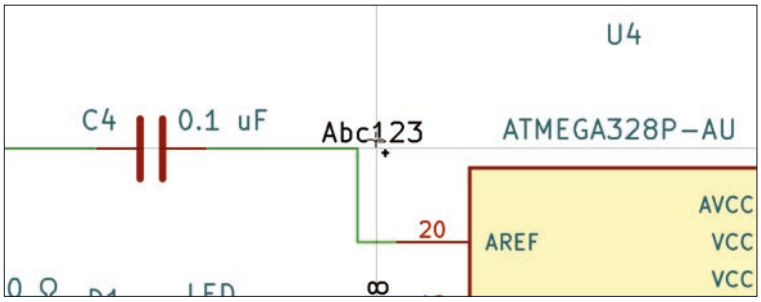


Figure 3.11.2: The new net label appears in the "4layer" branch schematic.

Open the PCB editor and update it with the changes from the schematic editor. Below, you can see a detail of the layout editor that shows the new net label:

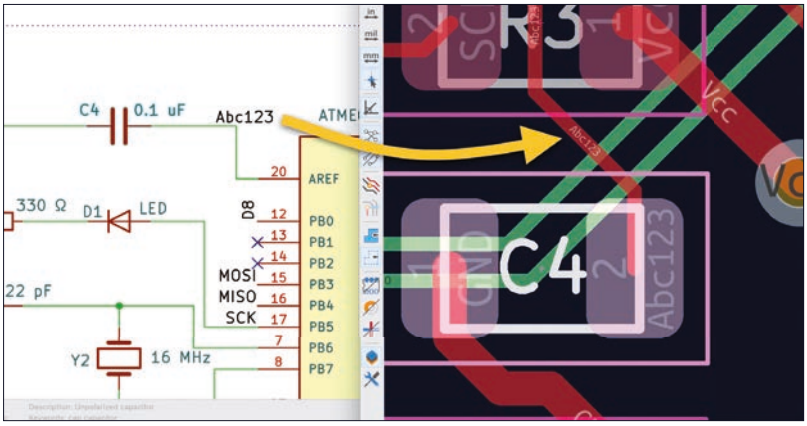


Figure 3.11.3: The new net label appears in the four-layer PCB.

You can repeat the process to update the two-layer PCB with the change in the schematic.

3.12. Finding and correcting a design defect

—PLEASE READ THIS FIRST!—

In the beta version of the 3rd edition of this book, a reader discovered a bug in an early version of the schematic. I decided it would be a good addition to the book to write this chapter in which I show how to fix such errors, instead of updating the book with the bug-free version of the project.

In this 4th edition, I decided to retain this chapter in its original format so that newer readers can reap the benefits. Please know that the version of the project in the preceding

chapters does not have this particular bug. Further, there are minor differences between the project in the 3rd and 4th editions of the book. I used KiCad 6 in the 3rd edition, and KiCad 8 in the 4th edition. Aside from these differences, everything you will learn in this chapter is valid in KiCad 8.

Enjoy!

As part of the beta program of this book, reader Antti found a bug in the schematic of this project. This bug required corrections in both the schematic and the layout. Instead of re-writing the chapters in this part of the book, I introduced this chapter. I preferred not to obscure the reality that errors are a natural part of the engineering that leads to fixes and improvements.

In this chapter, I will document the bug that Antti found and how I iterated through the design of the project's PCB to fix it.

The primary defect

In step four of the schematic workflow, I introduced the bug when I incorrectly wired the SCL pin (from the I2C interface) together with the SCK pin (from the SPI interface). Both SCL and SCK convey clock signals, but, of course, those signals belong to different communication interfaces and should be kept separate.

In the figure below, I use arrows to highlight the location of the bugs.

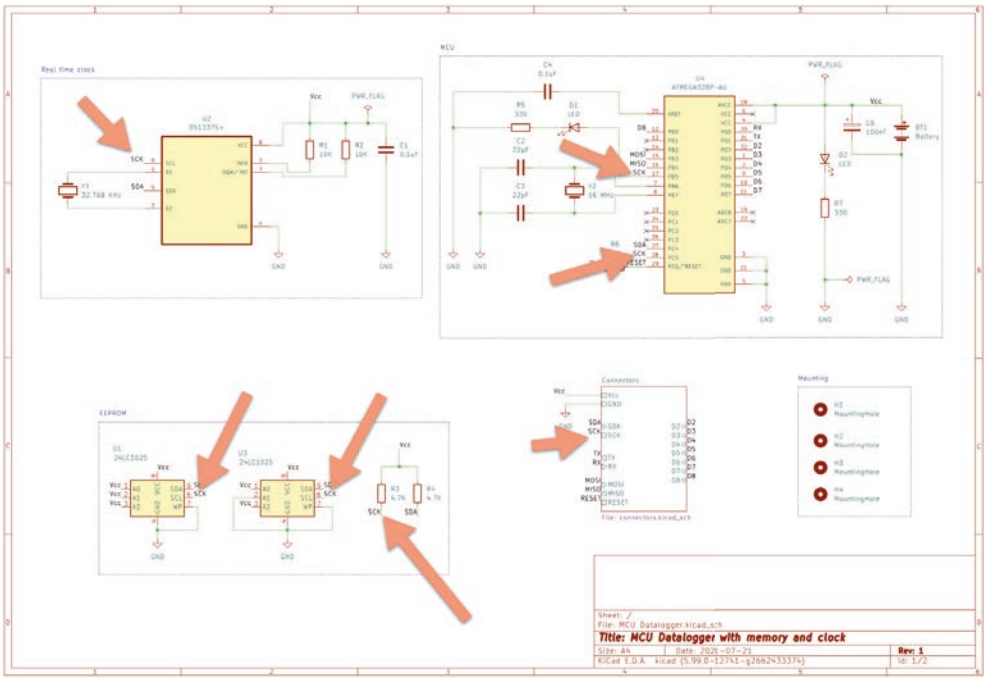


Figure 3.12.1: The location of the defects.

The secondary defect
I worked on the fix for the SCL-SCK bug on KiCad 6.0.0 rc1. I designed the PCB of this project in an earlier version of KiCad. As you can see in the figure above, I have used two PWR_FLAG symbols. I placed one PWR_FLAG in the Vcc net in the MCU group and another in the Vcc net of the "Real time clock" group. While the ERC in the earlier version of KiCad did not complain about the two PWR_FLAG symbols, KiCad 6.0.0 rc1 did. I addressed this issue in the schematic fix. KiCad 7 seems to behave like KiCad 6 in this respect.

The fixes
I created a new "SCL" net label to fix the primary defect and attached it to all SCL pins, including pin 28 of U4. I also added a new hierarchical pin label to the connector in the Connectors sheet.

I deleted one of the two PWR_FLAG symbols to fix the secondary defect. While addressing the defects, I keep the complete history of the changes in the project's Git repository. This was an excellent opportunity to see how Git and KiCad can work together in a real-life situation.

3.12.1. Fix the schematic

I will begin fixing the two defects in the 2layer branch. I plan to:

1. In the 2layer branch, update the schematic (set the correct net labels and delete the redundant power flag).
2. Merge the updated schematic file into the main branch.
3. Merge the updated schematic into the 4layer branch.
4. Return to the 2layer branch to update the 2-layer board layout.
5. Return to the 4layer branch to update the 4-layer board layout.

An essential element of the process I outline above is merging changes in the main branch into the 4layer branch while ignoring changes to the PCB layout file. I have described how to set up this capability in the earlier chapter, "3.11. Updating layout from changes to the schematic with Git." My instructions below will not work without first setting up the ".gitattributes" file in the main branch.

Fix the schematic in the 2layer branch
Open your terminal, switch into the project's KiCad directory, and ensure that you are working in the 2layer branch:

```
% git branch
  2l-experimental
* 2layer
  4l-experimental
  4layer
  main
 %
```

As you can see in my terminal session above, I am working in the 2layer branch. Continue in the schematic editor. Fix the secondary defect (the redundant power flag) by deleting it. I use the interactive delete tool for this:

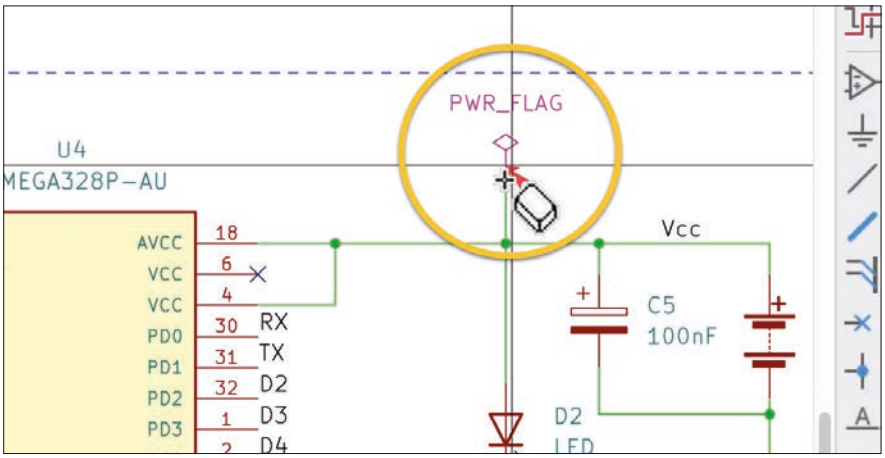


Figure 3.12.1.1: Delete the redundant power flag.

Do an ERC and confirm that the check is no longer complaining about the power flag issue.

Continue to fix the problem with the net labels. In the Atmega328P module, pin PB5 (#17) conveys the SCK signal, the clock for the SPI interface. I have attached the correct net label "SCK" in the schematic. However, pin PC5 (#28) conveys the SCL signal, the clock for the I2C interface. This is the core of the bug (I have incorrectly attached the SCK net label).

To fix this, delete all instances of the SCK net label, except for the one attached to U4 pin 17. Look at Figure 11.12.1 for a guide on the exact location of the SCK labels to delete.

To replace the deleted net label, create a new net label named "SCL." Attach instances of this label in all SCL pins (footprints U2, U4, U1, U3). You must also create a new hierarchical label named "SCL" for the Connectors sheet and expose it to the root sheet.

You can see the two sheets of this project with the net and hierarchical labels corrected in the two figures below.

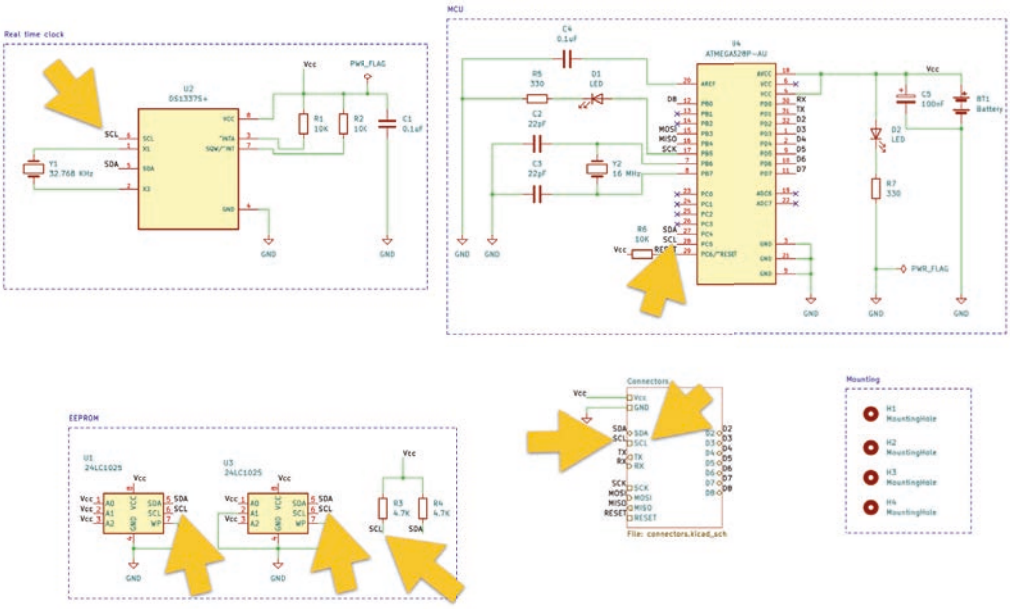


Figure 3.12.1.2: The root sheet with corrected net labels.

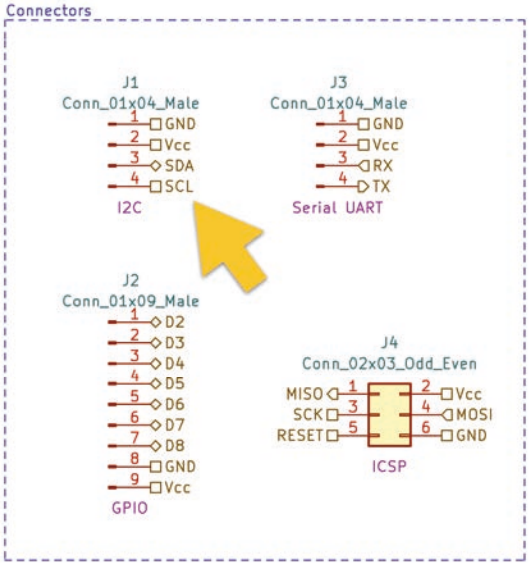


Figure 3.12.1.3: The Connectors sheet with corrected net labels.

Do a visual check to ensure you haven't forgotten anything, and then an ERC. There should be no errors. My ERC showed a couple of symbol violations ("modified in library"), which I can safely ignore. Save the schematic and return to your terminal window. Check the Git status, add the changes to staging, and commit them. Here is my session:

```
 % git status
On branch 2layer
Changes not staged for commit:
       (use "git add <file>..." to update what will be committed)
       (use "git checkout -- <file>..." to discard changes in working directory)

               modified: MCU Datalogger.kicad_pcb
               modified: MCU Datalogger.kicad_prl
               modified: MCU Datalogger.kicad_pro
               modified: MCU Datalogger.kicad_sch
               modified: connectors.kicad_sch

no changes added to commit (use "git add" and/or "git commit –a")
% git add .
% git commit –am
       "Fixed two defects in the schematic."
[2layer c84e21c] Fixed two defects in the schematic.
       5 files changed, 5141 insertions(+), 4860 deletions(-)
% git status
On branch 2layer
nothing to commit, working tree clean
%
```

At this point, you have fixed the bugs in the schematic and have committed the changes to the 2layer branch in Git. Let's merge these changes in the main branch.

Merge changes to 'main'
To switch your working branch to "main", use the "checkout" command:

```
% git checkout main
```

Next, merge the changes committed in the 2layer branch into main. For this, use the "merge" command:

```
% git merge 2layer
Auto–merging MCU Datalogger.kicad_sch
CONFLICT (content): Merge conflict in MCU Datalogger.kicad_sch
Auto-merging .gitignore
CONFLICT (content): Merge conflict in .gitignore
Automatic merge failed; fix conflicts and then commit the result.
%
```

Unfortunately, Git was unable to merge the changes automatically. Two files had conflicting changes, which I had to resolve manually in my instance. Git has inserted markers in those files to help me find the conflicts. Look at the first conflicted file, "MCU Datalogger.kicad_sch". Right-click the schematic editor icon in the main KiCad window and select "Edit in text a editor".

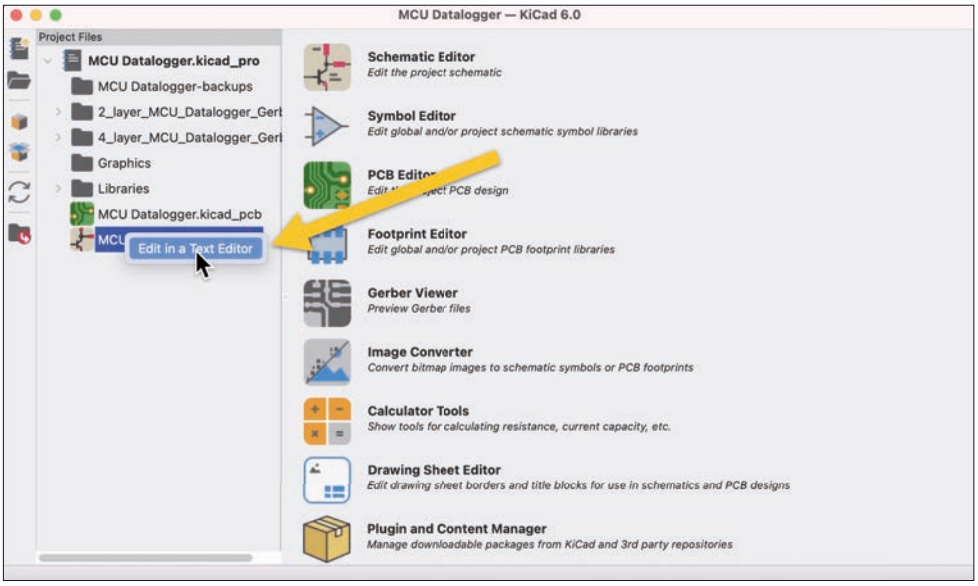


Figure 3.12.1.4: Open file in the text editor.

If you don't see this option, you must set up a text editor in the KiCad Preferences window. The "Common" tab will appear (see below).

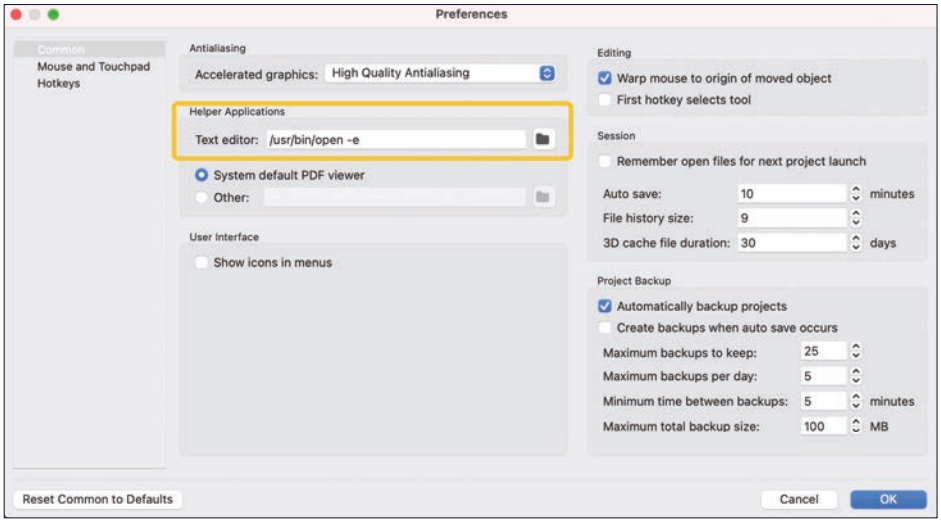


Figure 3.12.1.5: Setup your preferred text editor.

Once the schematic file is loaded in the text editor, scroll down until you find the marker "<<<<<< HEAD". Git marks the conflicted segments of the file with these markers. On top is the text from the main branch, and under the "======" marker, you see the version coming from the 2layer branch. In this case, I will delete the conflicting text from the main branch and keep the one from the 2layer branch. I depict the resolved conflict below:

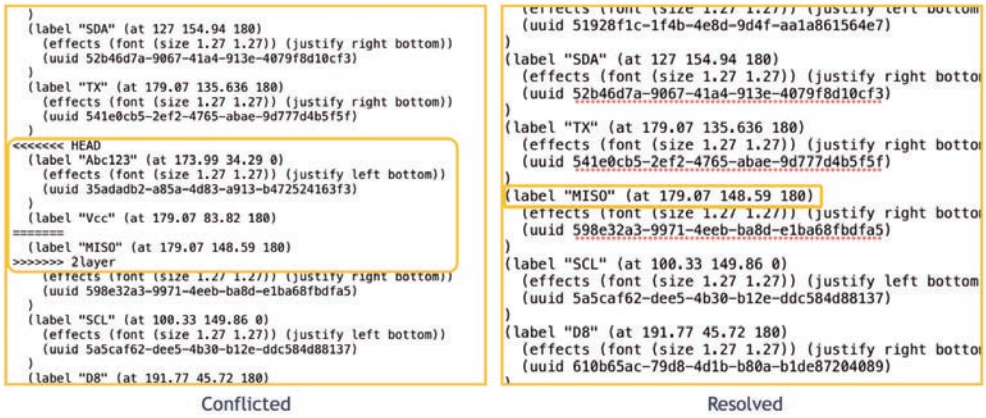


Figure 3.12.1.6: Resolving a Git merge conflict.

Save the text file. Repeat the process for the second conflicting file, ".gitignore". You cannot use the KiCad main window to open ".gitignore" in a text editor, so use your editor to find and open this file manually (please see this recipe for a detailed and up-to-date discussion of .gitignore). I used Atom and again chose to keep the 2layer branch version of the conflicted text:

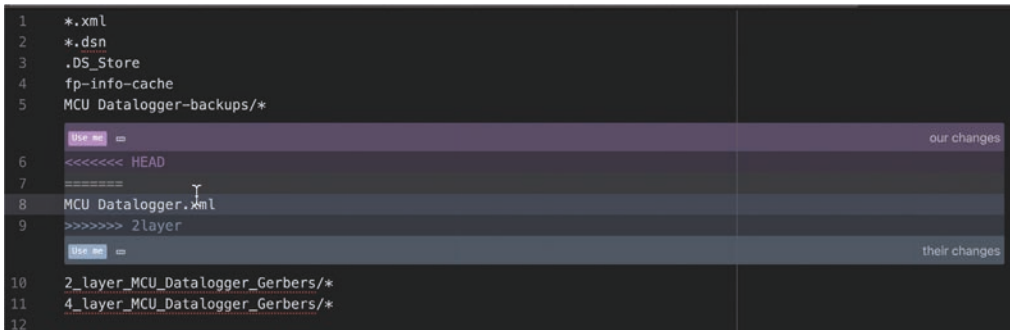


Figure 3.12.1.7: Resolving another Git merge conflict.

Save and close the text editor, and return to the command line terminal. Check the Git status and commit the resolved conflicts:

```
% git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
```

```
  (use "git merge ——abort" to abort the merge)

Changes to be committed:

       modified: MCU Datalogger.kicad_pcb
       modified: MCU Datalogger kicad_prl
       modified: MCU Datalogger.kicad_pro
       modified: connectors kicad_sch

Unmerged paths:
  (use "git add <file>..." to mark resolution)

       both modified: .gitignore
       both modified: MCU Datalogger.kicad_sch

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout —— <file>..." to discard changes in working directory)

       modified: MCU Datalogger.kicad_prl

% git add .
% git commit —am "Fixed bugs in schematic."
[main 4f1c810] Fixed bugs in schematic.
```

Let's recap before going any further. At this point:

1. The bugs are fixed in the schematic.
2. The updated schematic exists in the "main" and "2layer" branches.

What remains to be done is to:

1. Merge the schematic file changes into the "4layer" branch.
2. Fix the layouts for the 2-layer and 4-layer PCBs.

Let's do item "1" now and item "2" in the following two segments of this chapter.

Merge schematic changes into '4layer'
To merge the changes of the main branch into the 4layer branch (while ignoring the layout file thanks to the existence of the ".gitattributes" file), you will use the "merge" command while in the 4layer branch.

When you run the "merge" command, Git will probably ask you for a comment by opening up a text editor in your terminal. Type a description of the merge (such as "Bringing fixes in the schematic", and save the editor content to return to the terminal.

Here is my command line session:

```
% git checkout 4layer
Switched to branch '4layer'
% git merge main
Auto-merging MCU Datalogger.kicad_pro
Auto-merging MCU Datalogger.kicad_pr1
Auto-merging MCU Datalogger.kicad_pcb
Merge made by the 'recursive' strategy.
  .gitignore | 3 +
  MCU Datalogger.kicad_pcb | 7980 ++++++
```

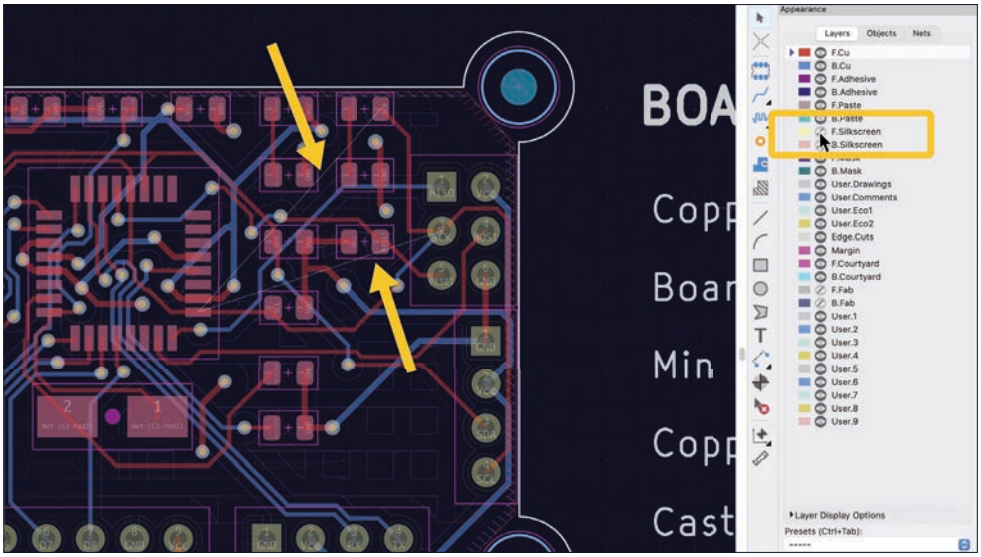


Figure 3.12.2.1: The two ratsnest lines are the result of the changes in the schematic.

When I rework a populated layout that is mostly routed, I use the interactive delete tool to delete tracks that start or finish on pads that I will need to rewire. For example, pad 2 of D1 (the footprint at the top right corner of the PCB) will need rewiring. I have deleted several affected copper tracks in the figure below to make redrawing easier.

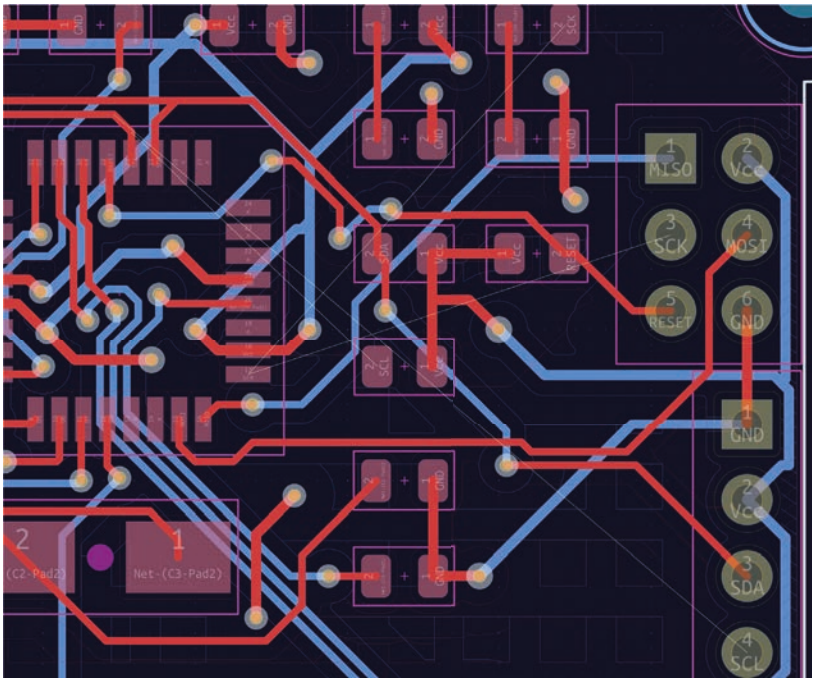


Figure 3.12.2.2: I have deleted several copper tracks.

Draw the new copper traces. Here's the fully-routed instance of my PCB:

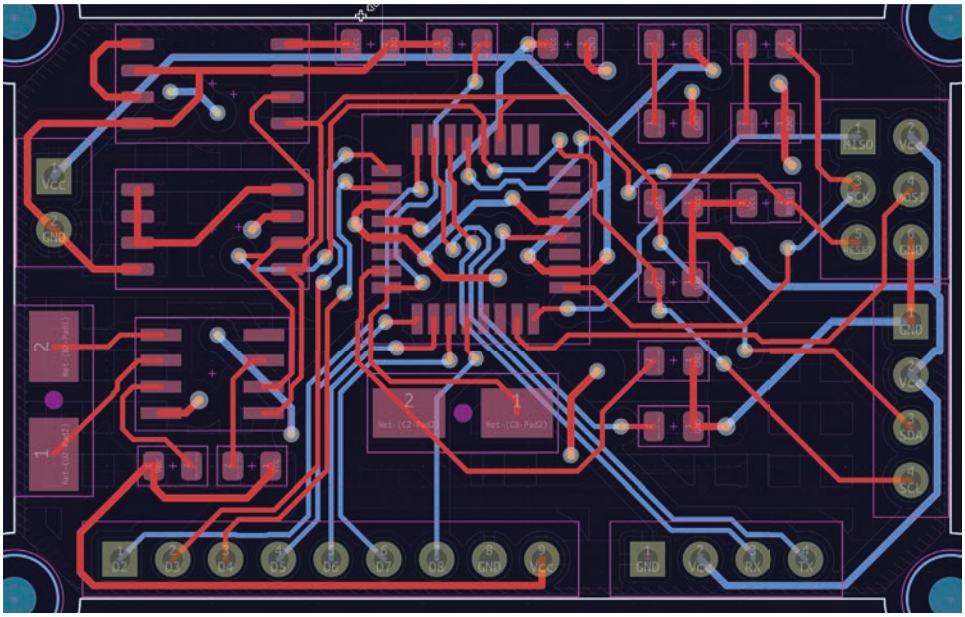


Figure 3.12.2.3: Fully routed, with defects fixed.

The 2-layer layout is now fully routed. Run a DRC to confirm that there are no errors. In my case, the DRC showed two "unconnected end" violations. I chose to address those violations by deleting small segments of copper track that I overlooked in earlier work sessions with this board. You can see one of those below:

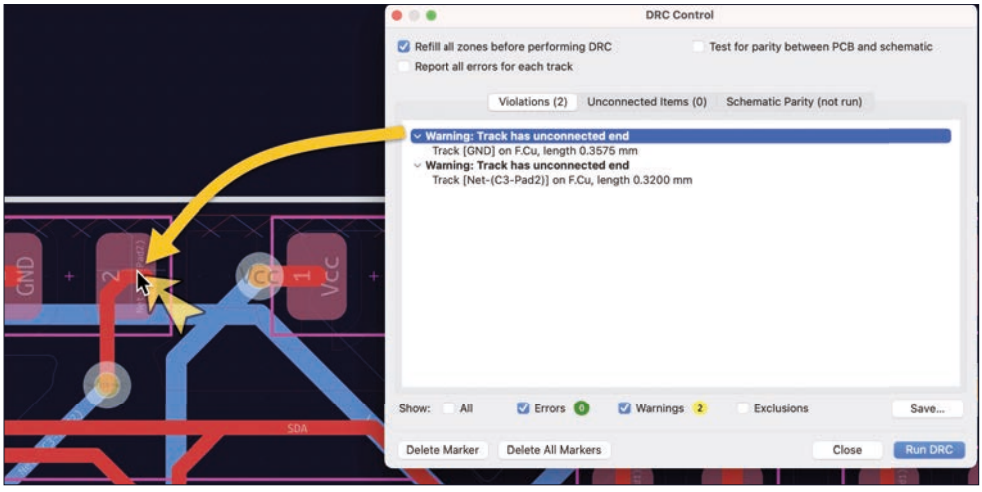


Figure 3.12.2.4: Fixing stray copper trace segments.

```
After addressing both violations, I repeated the DRC, which shows zero errors and
warnings.  Save the layout editor file and return it to the terminal when you are
ready. Check the Git status and commit the changes. Here is my terminal session:

% git status
On branch 2layer
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified: MCU Datalogger.kicad_pcb
        modified: MCU Datalogger.kicad_prl

no changes added to commit (use "git add" and/or "git commit -a")
% git add .
% git commit -am "Fixed 2 layer PCB."
[2layer 6cc3a84] Fixed 2 layer PCB.
  2 files changed, 2758 insertions(+), 2416 deletions(-)
```

The 2-layer PCB now contains the bug fixes. Let's repeat the process for the 4-layer PCB.

3.12.3. Fix the 4 layer PCB layout

Let's finish the update process by applying the fix to the 4-layer version of the PCB. Close the layout editor if you left it open after finishing work in the 2-layer PCB, then go to the terminal and switch your working Git branch to "4layer":

```
% git branch
  2l-experimental
* 2layer
  4l_experimental
  4layer
  main
% git checkout 4layer
Switched to branch '4layer'
%
```

In KiCad, open the schematic editor and double-check that you are working with the latest schematic version. You should have already imported the latest changes, but I find that small "sanity checks" like this can save you a lot of frustration and time if things don't go exactly to plan.

Open the layout editor, and import the schematic changes. As with the 2-layer PCB, you will see a couple of ratsnest lines that give you a clue that the import was successful. I use the same pattern as before: find redundant or incorrect copper tracks, and replace them with new ones. Make your layout editor comfortable to work with by turning off unnecessary

layers. For the sake of brevity, I present my finished and completely routed instance of the 4-layer PCB:

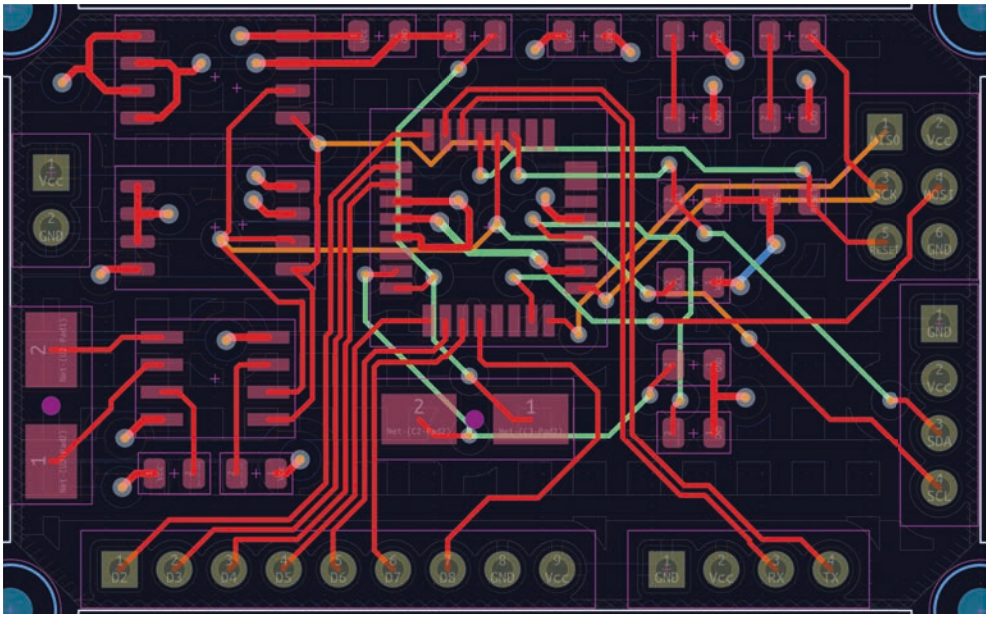


Figure 3.12.3.1: The fully routed 4-layer PCB that contains the fixes.

As always, do a DRC before you commit the changes. My DRC revealed similar warnings to the 2-layer PCB (stray copper track segments), which I fixed. My last DRC returned no errors or warnings. Let's finish up by committing the changes to the Git repository's 4layer branch:

```
% git status
On branch 4layer
Changes not staged for commit:
  (use "git add <File>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified: MCU Datalogger.kicad_pcb

no changes added to commit (use "git add" and/or "git commit –a")
% git add .
% git commit –am "Fixed net bugs in 4 layer version of the PCB."
[4layer 9138eb5] Fixed net bugs in 4 layer version of the PCB.
  1 file changed, 7007 insertions(+), 7101 deletions(-)
% git status
On branch 4layer
nothing to commit, working tree clean
```

This concludes the work needed to fix the bug that Antti found. Both versions of the PCB are now up-to-date, with all change history committed to the project Git repository.

4 : Project - An ESP32 clone

4.1. Project - Introduction

Welcome to a new part of this book. In the chapters that follow, you will design an ESP32 development board. To assist you with the design process, you will use the reference schematic design from Espressif, the original designer and manufacturer of the original ESP32 dev kit module.

You will learn to use and modify reference designs by working on this project. The ESP32 is a widely used and understood board. It is compact yet packs significant computing power and capabilities. Its circuit board is more complex than those in the previous projects of this book but not too complex to make this project long and tedious.

Below you can see the layout design as it will look at the end of the project.

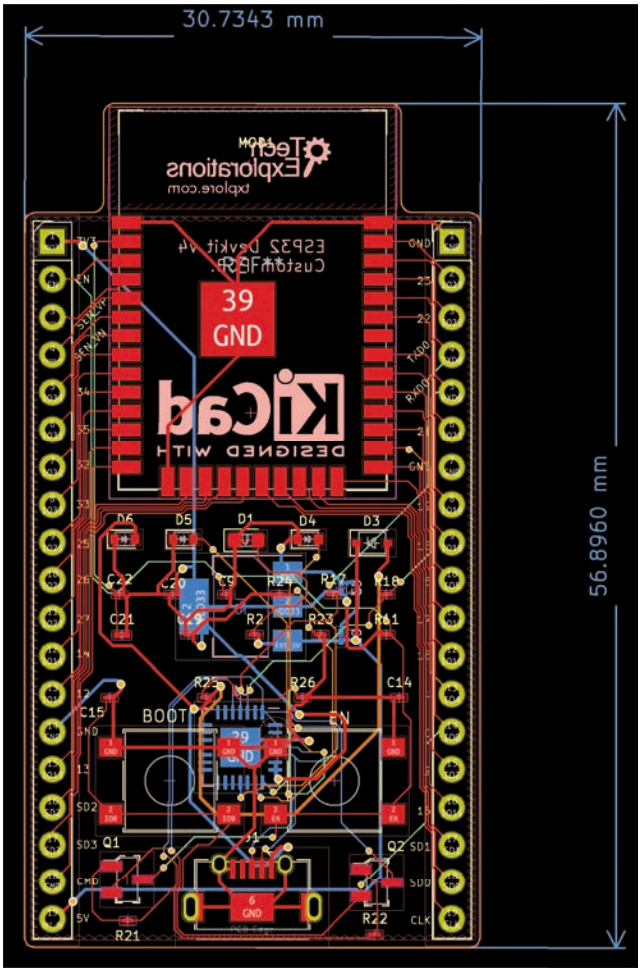


Figure 4.1.1: The ESP32 development kit clone project deliverable.

In this instance of the development kit, I have retained the form factor and shape of the original ESP32 development kit. This is a four-layer board with the ESP32 module at the top, its integrated antenna extending outside the board, the power and serial communications port at the bottom, and the pin headers along the sides. The board also features the two control buttons, "BOOT" and "EN", and uses SMD components on the front and back. Here's a view of the reference layout:

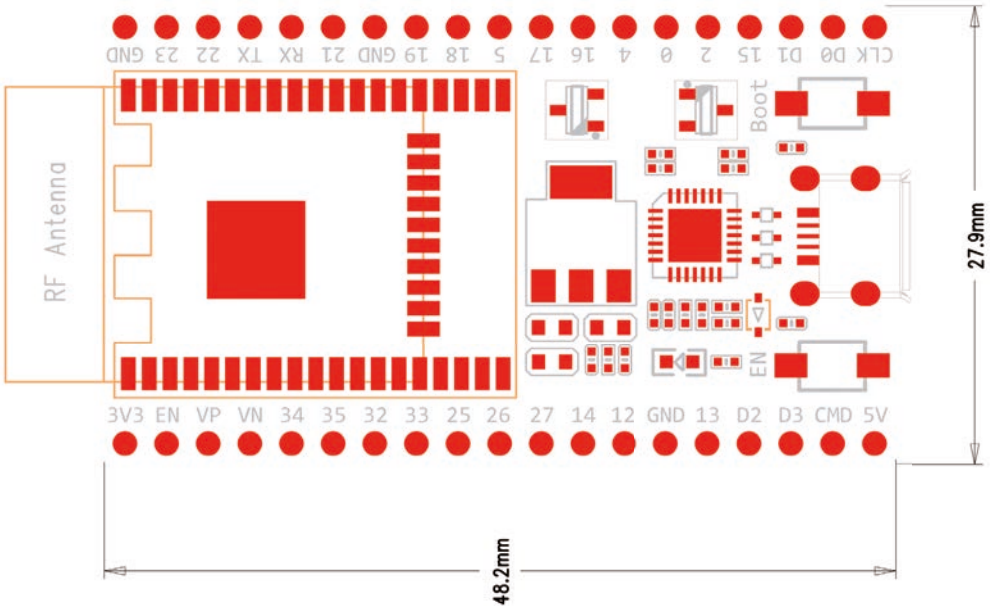


Figure 4.1.2: A view of the ESP32 development kit reference layout.

Once you understand the design elements of this board, you will be able to modify its shape and component placement to match the requirements of your project. For example, in my instance of the board, I have placed the voltage regulator in the back of the PCB, even though the regulator is the front in the reference design. I did this because I wanted to allow more room in the front for small components, like the resistors, to make assembly by hand easier.

Below, you can see the final schematic.

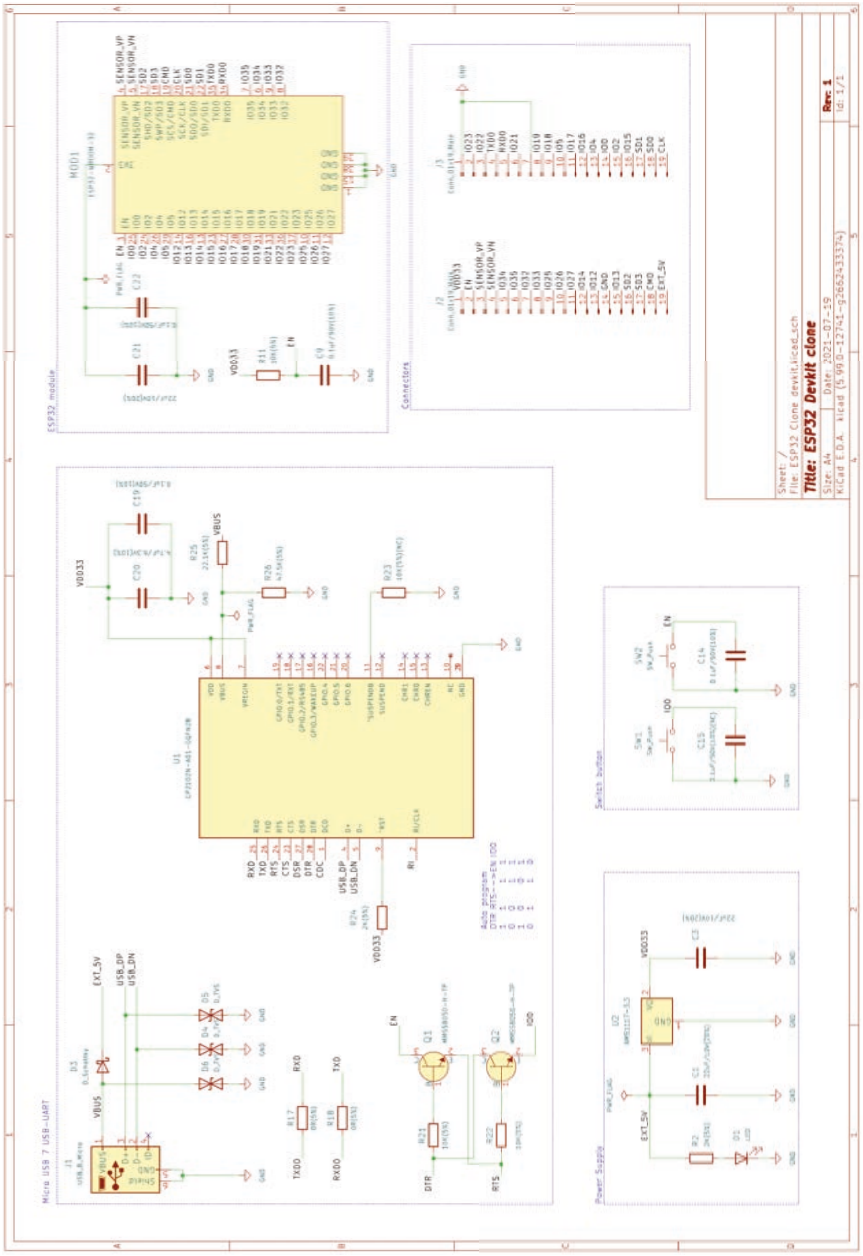


Figure 4.1.3: The ESP32 development kit clone project schematic.

Thanks to the ESP32 module's highly integrated design, the board that hosts the module (i.e., the development kit) is relatively simple. In the development kit design you see above, I have arranged the components in five functional groups and drawn all details from the original Espressif reference schematic design. You can see the reference schematic design below:

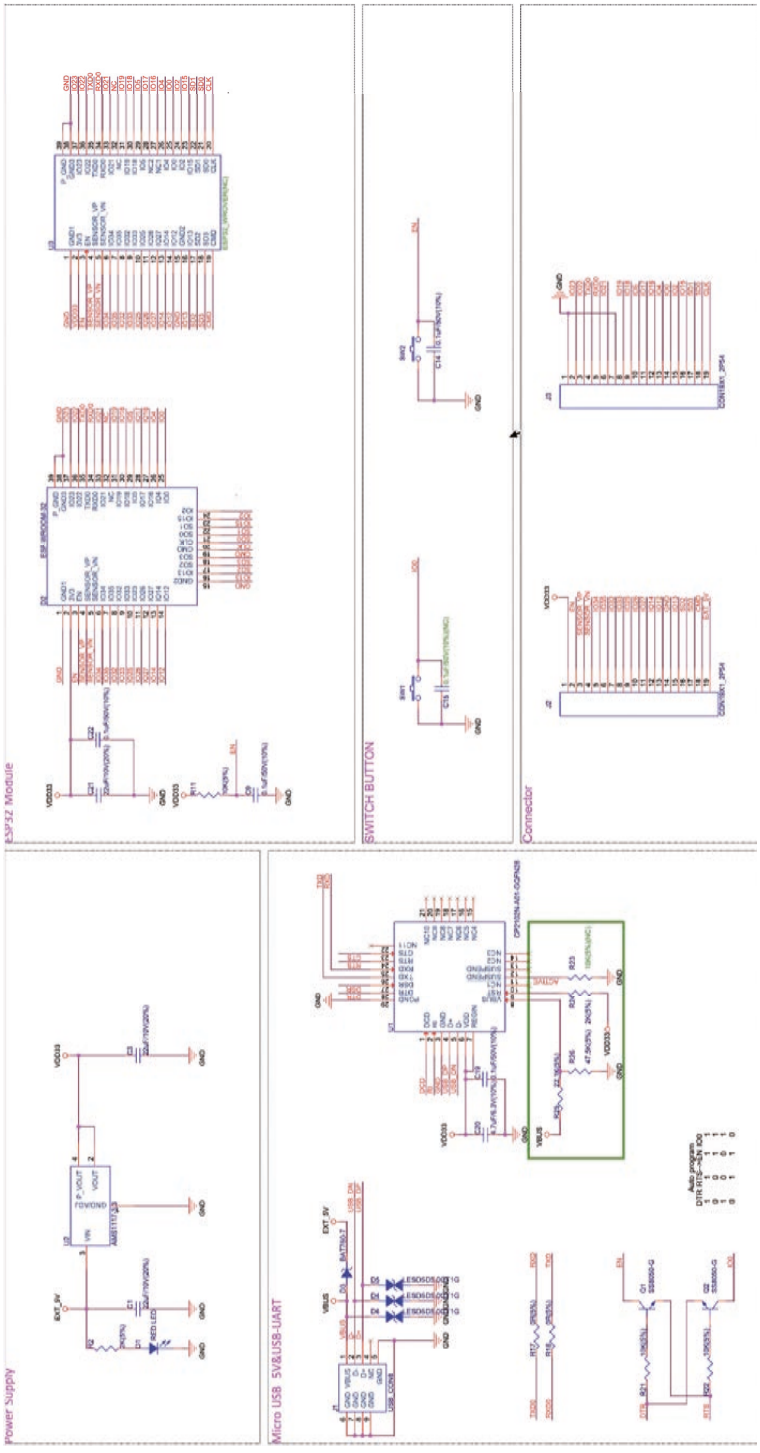


Figure 4.1.4: The ESP32 development kit reference schematic.

You will learn more about this in chapter 2.

Below, you can see the 3D rendering of the development kit.

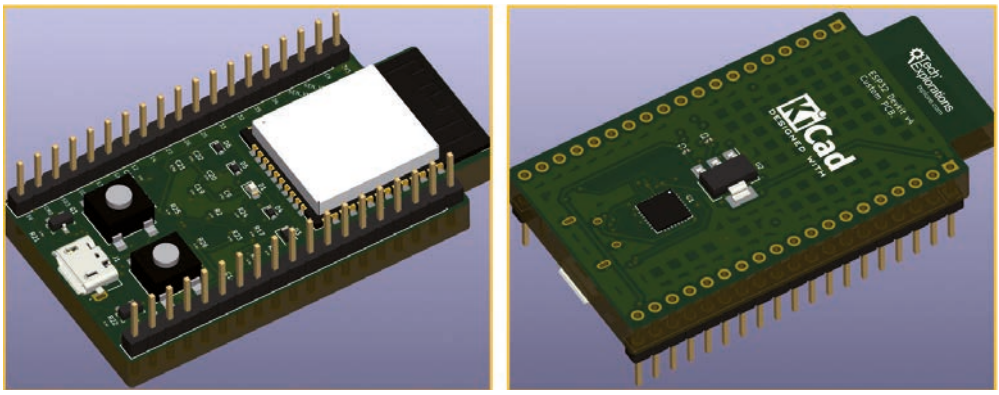


Figure 4.1.5: A 3D rendered view of the project deliverable.

The 3D rendering contains models for all components and is accurate in terms of the appearance of the final manufactured board.

Customizing an existing board begins by finding the original schematic and layout designs. For the ESP32 development kit, you can find the official reference designs on the Espressif website (https://www.espressif.com/en/support/documents/technical-documents). I have used the reference design for the ESP32-DevKitC-v4 board. You can use the search filter to find this design (or try my search URL,"https://www.espressif.com/en/support/documents/technical-documents?keys=ESP32-Devkit").

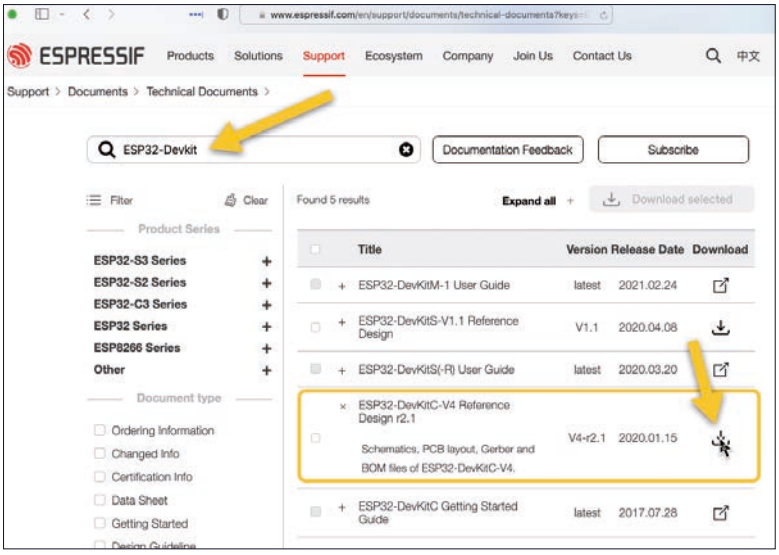


Figure 4.1.6: The source documentation and designs for this project.

Download the ZIP file that contains everything you will need for this project: the schematics, the PCB layout, and the BOM. The ZIP also contains a set of Gerber files you do not need (you will generate a new set at the end of the project).

The reference BOM is important because it gives you all the components and values you need for your custom board. These components and values are tested by Espressif and used in millions of manufactured boards, so you can be confident they will work. I used the reference BOM to help me select symbols and footprints for my board instance. The project BOM that you see below contains information from the reference BOM.

Reference	Value	Footprint
C1, C3, C21	22uF/10V(20%)	Capacitor_SMD:C_0201_0603Metric
C9, C14, C19, C22	0.1uF/50V(10%)	Capacitor_SMD:C_01005_0402Metric
C15	0.1uF/50V(10%)(NC)	Capacitor_SMD:C_01005_0402Metric
C20	4.7uF/6.3V(10%)	Capacitor_SMD:C_01005_0402Metric
D1	LED	LED_SMD:LED_0603_1608Metric
D3	D_Schottky	Diode_SMD:D_SOD-323
D4, D5, D6	D_TVS	Diode_SMD:D_SOD-523
J1	USB_B_Micro	Connector_USB:USB_Micro-AB_Molex_47590-0001
J2, J3	Conn_01x19_Male	Connector_PinHeader_2.54mm:PinHeader_1x19_P2.54mm_Vertical
MOD1	ESP32-WROOM-32	digikey-footprints:ESP32-WROOM-32D
Q1, Q2	MMSS8050-H-TP	digikey-footprints:SOT-23-3
R2, R24	2K(5%)	Resistor_SMD:R_01005_0402Metric
R7, R18	0R(5%)	Resistor_SMD:R_01005_0402Metric
R11, R21, R22	10K(5%)	Resistor_SMD:R_01005_0402Metric
R23	10K(5%)(NC)	Resistor_SMD:R_01005_0402Metric
R25	22.1K(5%)	Resistor_SMD:R_01005_0402Metric
R26	47.5K(5%)	Resistor_SMD:R_01005_0402Metric
SW1, SW2	SW_Push	Button_Switch_SMD:SW_SPST_B3S-1000
U1	CP2102N-A01-GQFN28	Package_DFN_QFN:TQFN-28-1EP_5x5mm_P0.5mm_EP2.7x2.7mm
U2	AMS1117-3.3	Package_TO_SOT_SMD:SOT-223-3_TabPin2

Table 4.1.1: The Bill of Materials for this project.

The BOM shows several very small SMD components, such as the 0402 resistors. If you plan to assemble your custom board by hand, you should consider the difficulty of working with such small components. But since this is your custom board, you can replace those small components with alternatives that you feel are more appropriate for your projects.

This is just an example of modifying a reference design to fit your specific requirements.

The main objectives of this project are:

1. To help you practice skills you acquired in previous projects.
2. To gain experience designing a PCB based on an existing reference design.
3. To gain experience in creating dense, four-layer PCBs.

I used Snapeda to find the symbol-footprint pairs for MOD1 (the ESP32 module). You should be able to find all other symbols and footprints in KiCad's libraries.

Let's begin.

4.2. Schematic design

In this chapter, you will complete the schematic design of this PCB by following the schematic design workflow.

4.2.1. Schema 1 - New KiCad project and Schematic Setup

Let's begin with the new project. Create a new KiCad project and store its files in a new directory. I have named my project "ESP32 Clone Devkit". You can see my new project setup below:

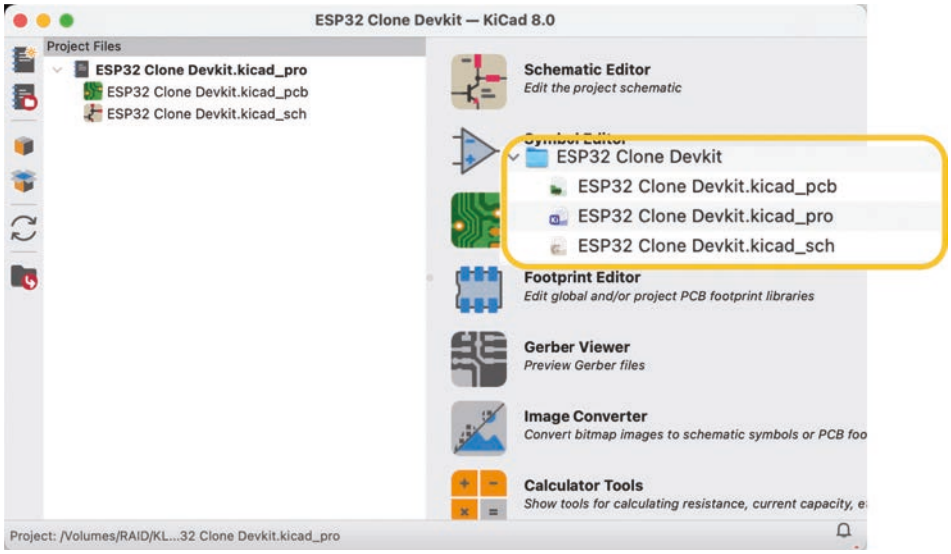


Figure 4.2.1.1: Begun a new project.

Continue to set up the schematic editor. I list my settings below. If a setting is not in the list, you can assume I have not changed its default setting.

- Page Settings:
  - Issue Date: copied today's date from the date field.
  - Revision: 2.
  - Title: ESP32 Devkit clone.

Next, review the reference schematic for symbols, footprints, and 3D models that you will need to download from an online repository like Snapeda.

Three components do not have suitable candidates in KiCad, so you will need to source elsewhere. In the figure below, I have circled the symbols you need to get from Snapeda or the Digikey libraries for KiCad.

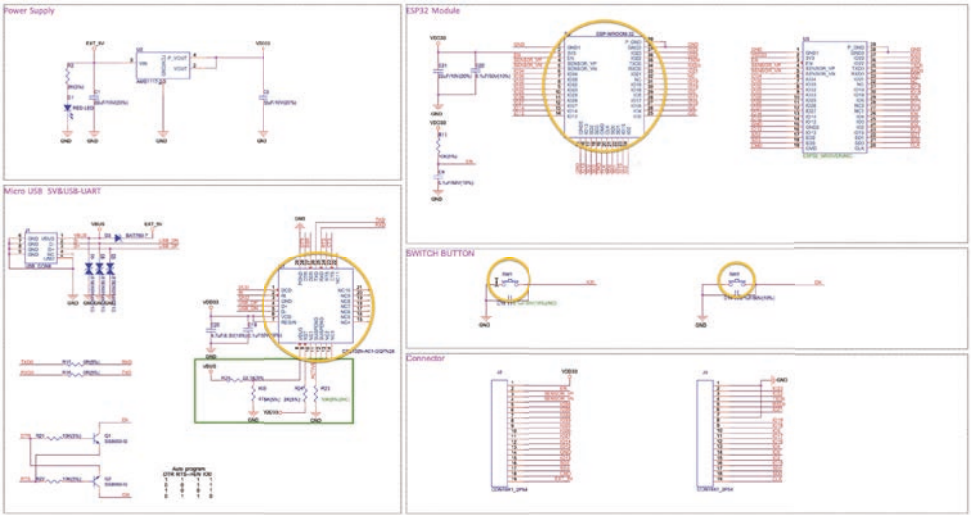


Figure 4.2.1.2: The circled symbols are unavailable in the KiCad symbol libraries.

First, I need a component for the two push buttons. I looked at the libraries that come with KiCad for SMD push buttons, but I could not find the exact model I wish to use. After researching Snapeda, I found a suitable model, the B3S-1000P[8], with a symbol, footprint, and 3D model. Download the ZIP for this component and expand the archive in your project folder in a new "libraries" folder.

Second, a component for the CP2102N chip that implements the USB to UART bridge. While KiCad has a standard QFN28 footprint, it does not have a symbol that matches the one used in the schematic design reference. I could spend time compensating for the discrepancies, but I decided to find an exact match for the symbol instead of attempting to rewire the schematic. The component I use in my design is CP2102N-A01-GQFN28 [9]. Download it and expand it in the project libraries folder. The archive contains all three files (symbol, footprint, and 3D model).

8 https://www.snapeda.com/parts/B3S-1000P/Omron%20Electronics/view-part/?ref=search&t=B3S-1000P
9 https://www.snapeda.com/parts/CP2102N-A01-GQFN28/Silicon%20Labs/view-part/?ref=search&t=CP2102N-A01-GQFN28

Third, a component for the ESP32 module. In previous editions of this book, I used a symbol and footprint available in the DigiKey library. However, this library has not been updated for several years, so I switched to the ESP32WROOM-32 library from Snapmagic. You can download the symbol, footprint, and 3D model from snapeda.com by searching for "ESP32WROOM-32".

In addition to the above, I used a fourth component from Snapeda, the MMSS8050-H-TP transistor. This is a BJT transistor. Although you can find the symbol and a suitable footprint in the libraries that ship with KiCad, I wanted to try the Snapeda equivalent anyway because it comes with the symbol, footprint, and 3D model in a single package. You can find this component on Snapeda by searching for "MMSS8050-H-TP."

You can see the three libraries in my library manager below:

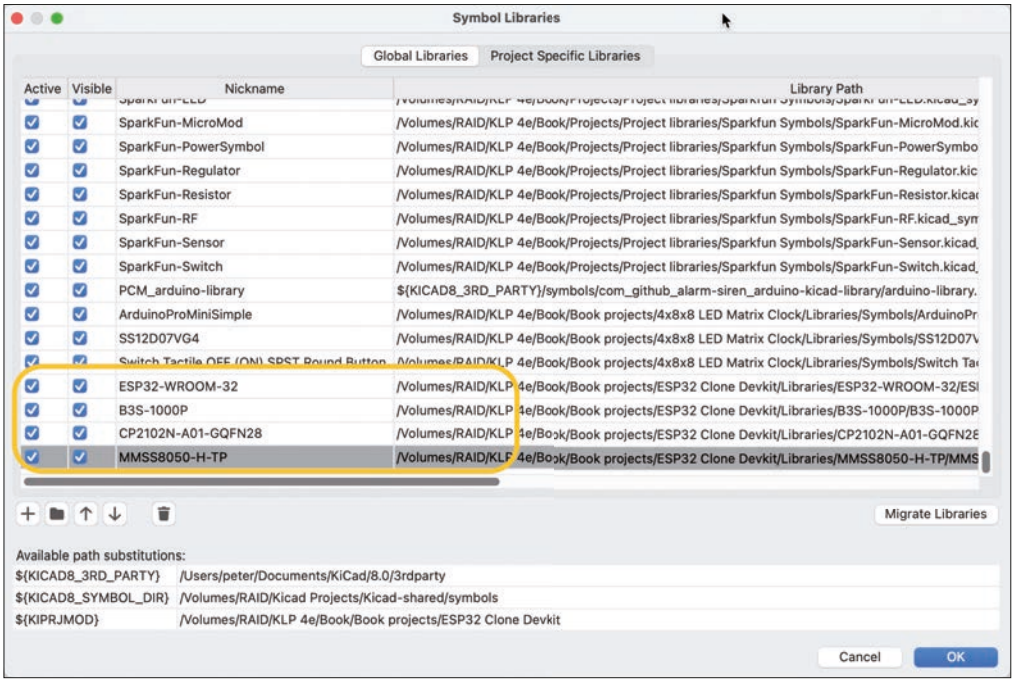


Figure 4.2.1.3: The three external symbol libraries I will use in this project.

Once you have all three symbols downloaded and installed, continue with step two of the workflow.

4.2.2. Schema 2 - Symbols

You prepared your schematic editor in the previous segment and installed the three third-party symbols. It is now time to add the schematic symbols to the design sheet. Use the reference design when adding the symbols to the sheet. I have arranged the schematic editor and PDF viewer windows side by side, as in the example below:

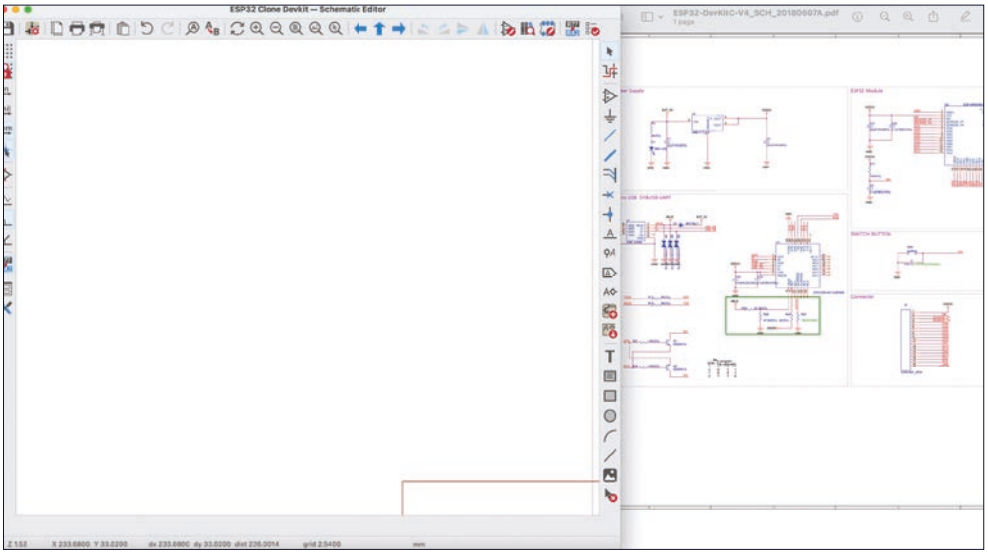


Figure 4.2.2.1: Side-by-side arrangement streamlines work.

I decided to add all large components for this schematic first (starting with the ESP32 module) and then continue with the smaller ones. Below, you can see my schematic after I have added the main components to the sheet:

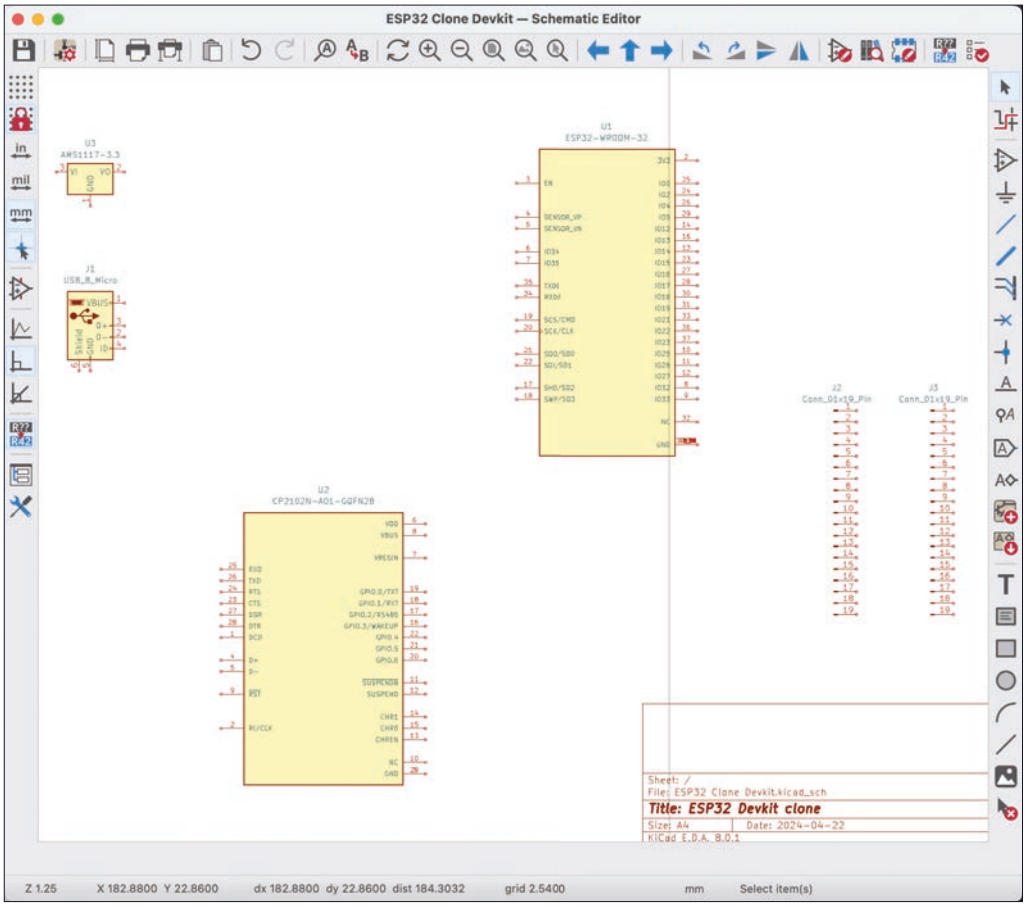


Figure 4.2.2.2: Added main components to the schematic sheet.

Continue with the smaller components, such as the buttons, the USB connector, LEDs, diodes, and resistors. Here's my schematic with all the symbols added to the sheet:

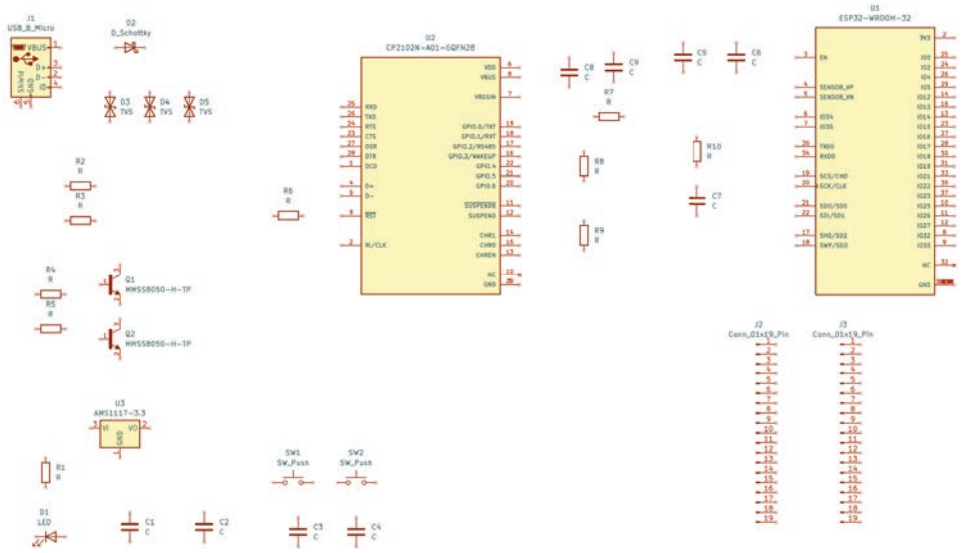


Figure 4.2.2.3: All symbols added to the sheet. I will make changes to the designators to match the reference design.

At this point, all schematic symbols should be on the sheet. In the next step of the workflow, you will annotate those symbols. Unlike the practice from previous projects, I recommend that you annotate this schematic manually so that your schematic follows the designators used in the reference design. If you use the automatic annotator, the chances are that your schematic will contain designators that don't match those in the reference design. This will make it harder to work in the schematic editor, and you will need to be much more cautious as you progress through the copy process.

4.2.3. Schema 3 - Annotate and set component values

You will manually annotate symbols in this segment and add the component values. In earlier projects in this book, you used the auto-annotator to generate unique identifiers for each symbol as you added them to the editor. As in previous projects, you did not work off a reference design, so you did not have to worry about discrepancies between the reference designators in your schematic and those in the reference design.

In this project, however, you use a reference schematic design to aid your design. Discrepancies in the symbol reference designators can be confusing later in the project. Therefore, ensure that the reference designators match your schematic and that the reference schematic is better.

There are two ways by which you can set values for each symbol:

1. You can use the Properties window for each symbol individually.
2. You can use the Symbol Fields Table window to complete this task bulk.

Option 2 is my preference as it is a faster way to complete this task. However, you need to be careful to correctly annotate each symbol without the benefit of seeing it in the schematic. The schematic editor can assist by panning the schematic on the symbol you select in the Symbol Fields Table. To make this work, arrange the schematic editor and symbol fields table windows side-by-side, as you can see in the screenshot below:

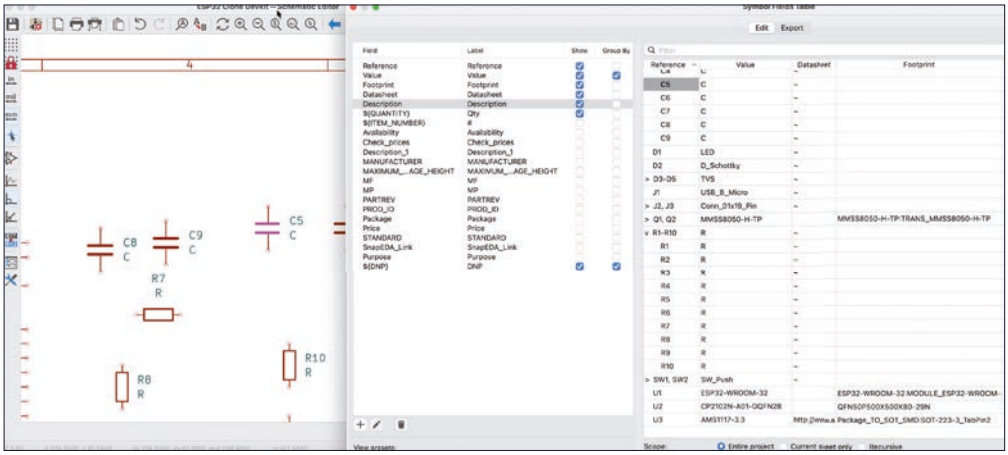


Figure 4.2.3.1: The schematic editor and symbol fields table windows side-by-side.

In the figure below, you can see the Symbol Fields Table window, which will look at the end of this segment. I have already populated all reference designators and values:

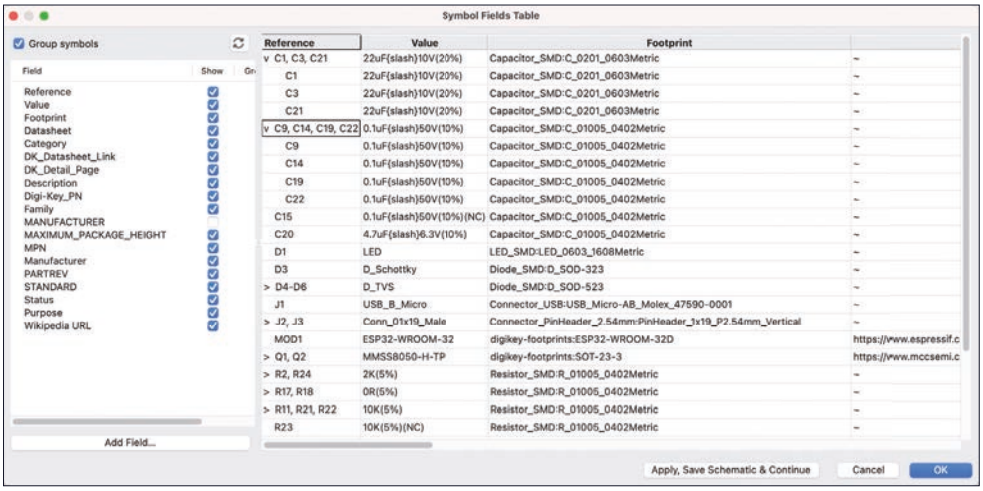


Figure 4.2.3.2: The Symbol Fields Table, populated.

However, at the start of this work, the same window looks like this:

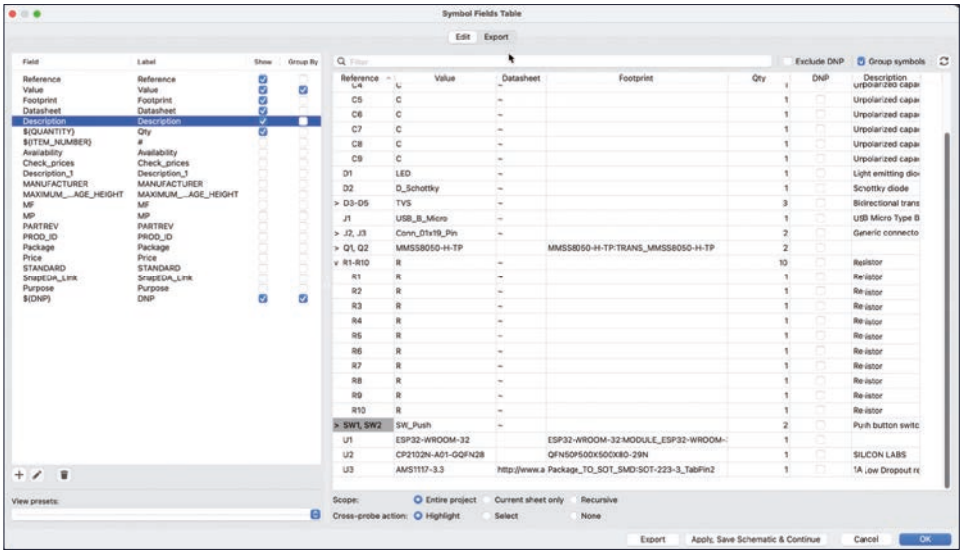


Figure 4.2.3.3: The Symbol Fields Table, not populated.

Unfortunately, you can not manually use the Symbol Fields Table to edit the Reference fields; you can edit all other fields. Therefore, the only option for manually editing the reference designator for each symbol is to use the symbol's Properties window. Since you have to open this window manually for each symbol, you can also set the symbol's value field.

Again, set up your desktop so that the schematic editor and the PDF viewer window that contains the reference design are side-by-side. I chose to begin from the power supply functional block of the schematic and edit each symbol individually. For example, in the figure below, I edit the voltage regulator symbol with the designator "U2", following the reference design:

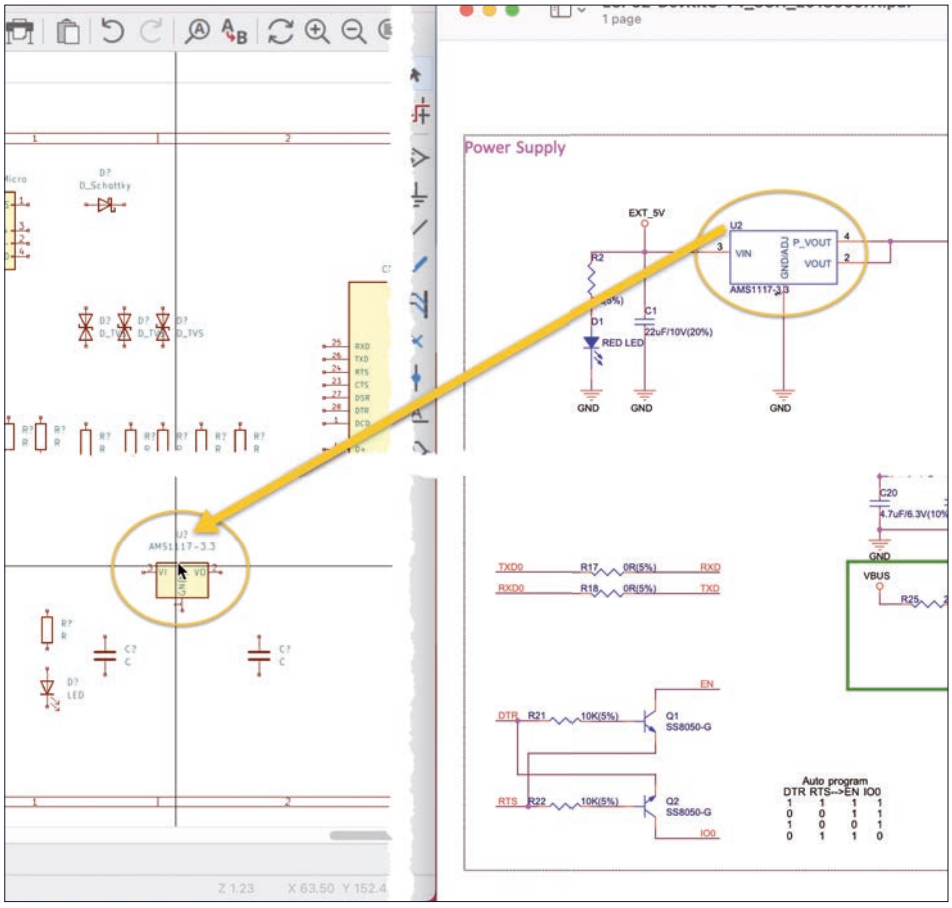


Figure 4.2.3.4: Setting the designator of the voltage regulator to "U2".

Below is the Properties window for the voltage regulator. I have just edited the contents of the Reference and Value fields using information from the reference design.

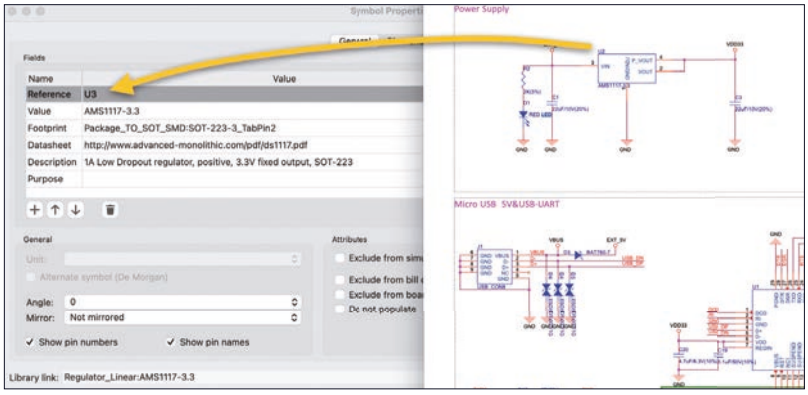


Figure 4.2.3.5: The Properties window for the voltage regulator symbol.

Continue with other symbols in the power supply group (resistor, LED, capacitor). Remember to add the values as you work your way through the symbols. If this suits your working style better, you can leave the values later and edit them in bulk using the Symbol Fields Table window.

Be methodical, and take your time. This is not something you should rush. It took me approximately 15 minutes to manually annotate the schematic. Below is the result:

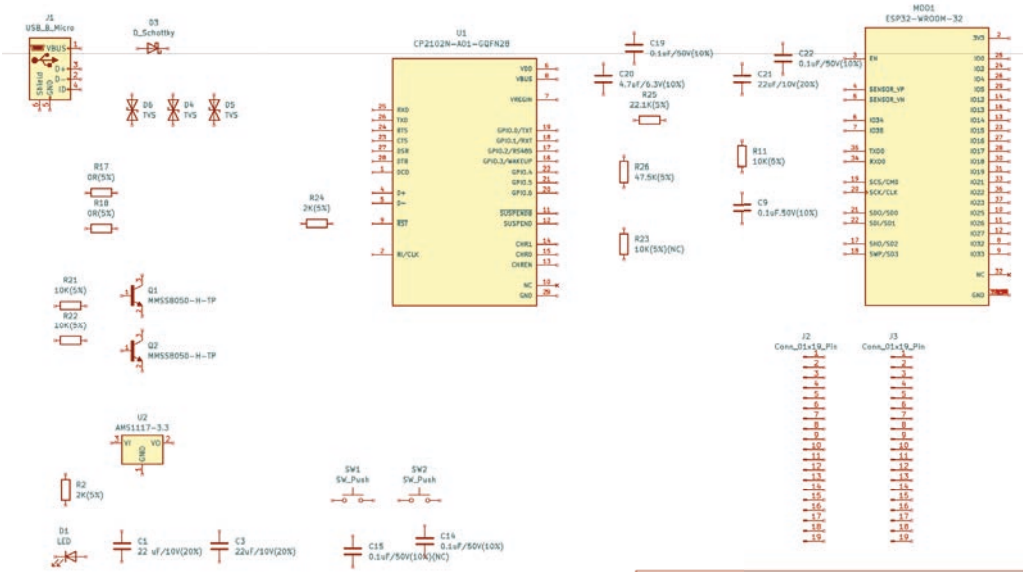


Figure 4.2.3.6: The schematic is fully annotated.

Let's continue to the next step of the workflow, where you will set the symbols in their final positions before the wiring in step four.

4.2.4. Schema 3 - Arrange

You are still working on step three of the schematic design workflow. You have added the symbols in the sheet and manually annotated them. In this segment, you will finalise the position of each symbol in the sheet so that they are in appropriate positions ahead of the wiring step.

Most symbols are close to where I'd like them to be. Most of the work I had to do here was make small changes to those positions. For example, I aligned resistors and capacitors to connect them with straight wire lines later. I also took the opportunity to delineate the functional groups with graphical boxes. This is something that I usually leave for step seven of the workflow ("comments"), but I decided to do it earlier in this project to help me reduce the risk of error as I am working with the reference design.

Below, you can see the schematic as it looked before I started changing the position of the symbols:

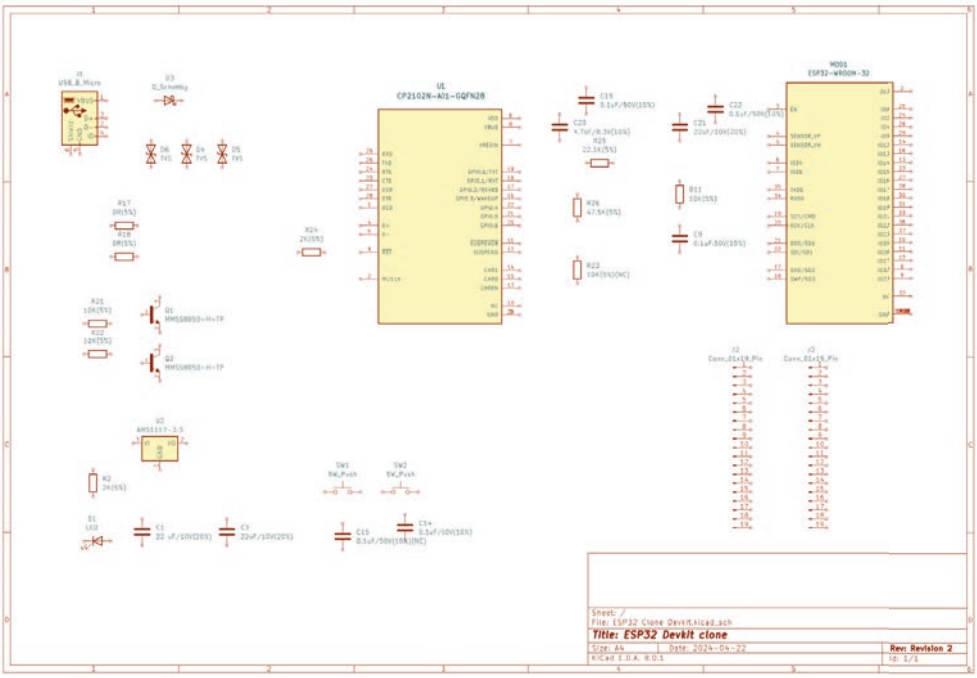


Figure 4.2.4.1: Starting to reposition the symbols.

Below, you see the schematic after I completed this work:

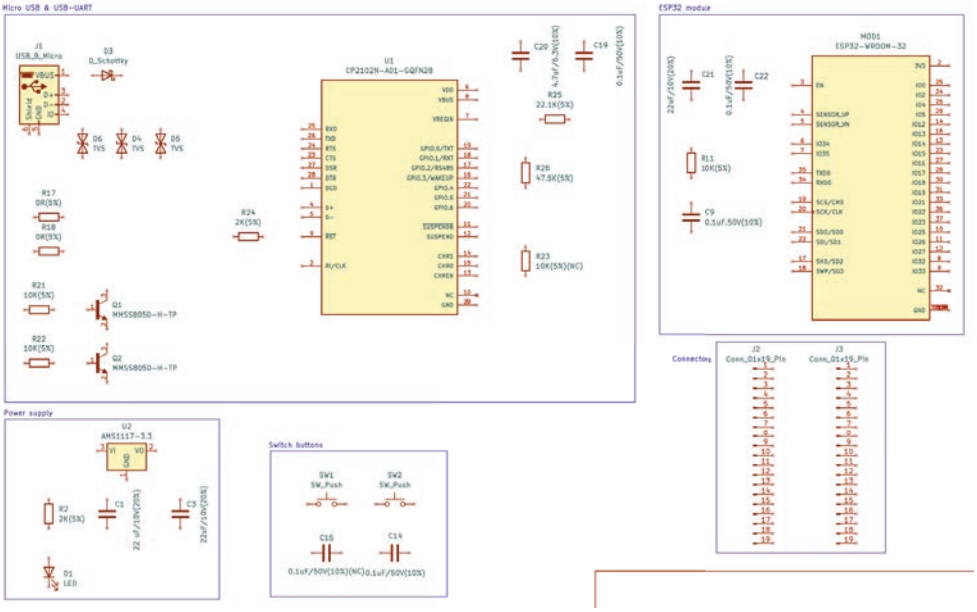
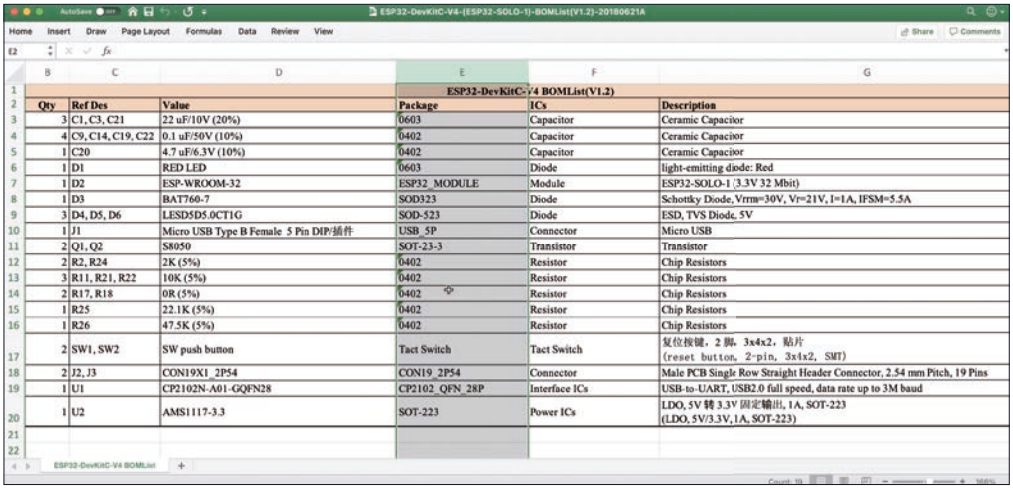


Figure 4.2.4.2: Completed the repositioning of the symbols.

As you can see, the difference between the first and second versions of the schematic is small. However, the newer version is more readable. Let's finish step three of the schematic workflow by associating symbols with their footprint counterpart.

4.2.5. Schema 3 - Associate

This segment will complete step three of the schematic design workflow by associating the symbols with their footprints. To do this work, you will need the BOM file with the reference design archive you downloaded from Espressif. You will find this file inside a folder named "05_BOM List". The file name is "ESP32-DevKitC-V4-(ESP32-SOLO-1)-BOMList(V1.2)-20180621A.xlsx". Below, you can see this file open in Microsoft Excel:



	Qty	Ref Des	Value	Package	ICs	Description
1				ESP32-DevKitC-V4 BOMList(V1.2)		
3	3	C1, C3, C21	22 uF/10V (20%)	0603	Capacitor	Ceramic Capacitor
4	4	C9, C14, C19, C22	0.1 uF/50V (10%)	0402	Capacitor	Ceramic Capacitor
5	1	C20	4.7 uF/6.3V (10%)	0402	Capacitor	Ceramic Capacitor
6	1	D1	RED LED	0603	Diode	light-emitting diode: Red
7	1	D2	ESP-WROOM-32	ESP32_MODULE	Module	ESP32-SOLO-1 (3.3V 32 Mbit)
8	1	D3	BAT760-7	SOD323	Diode	Schottky Diode,Vrrm=30V, Vr=21V, I=1A, IFSM=5.5A
9	3	D4, D5, D6	LESD5D5.0CT1G	SOD-523	Diode	ESD, TVS Diode, 5V
10	1	J1	Micro USB Type B Female 5 Pin DIP/插件	USB_5P	Connector	Micro USB
11	2	Q1, Q2	S8050	SOT-23-3	Transistor	Transistor
12	2	R2, R24	2K (5%)	0402	Resistor	Chip Resistors
13	3	R11, R21, R22	10K (5%)	0402	Resistor	Chip Resistors
14	2	R17, R18	0R (5%)	0402	Resistor	Chip Resistors
15	1	R25	22.1K (5%)	0402	Resistor	Chip Resistors
16	1	R26	47.5K (5%)	0402	Resistor	Chip Resistors
17	2	SW1, SW2	SW push button	Tact Switch	Tact Switch	复位按键，2 脚，3x4x2，贴片 (reset button, 2-pin, 3x4x2, SMT)
18	2	J2, J3	CON19X1_2P54	CON19_2P54	Connector	Male PCB Single Row Straight Header Connector, 2.54 mm Pitch, 19 Pins
19	1	U1	CP2102N-A01-GQFN28	CP2102_QFN_28P	Interface ICs	USB-to-UART, USB2.0 full speed, data rate up to 3M baud
20	1	U2	AMS1117-3.3	SOT-223	Power ICs	LDO, 5V 转 3.3V 固定输出, 1A, SOT-223 (LDO, 5V/3.3V,1A, SOT-223)
21						
22						

Figure 4.2.5.1: The reference design BOM file.

Use the information in column E of the BOM file to find the appropriate footprint. For example, for capacitor C1, the reference design recommends the 0603 package for a ceramic capacitor. In implementing this board, I have followed the reference design recommendations.

In the schematic editor, use the Footprint assignment window to make the associations in bulk. You can also use the Properties window for each symbol (a more time-consuming option).

I recommend you take some time to browse the footprint libraries until you can find the required footprints. You can consult the BOM table to introduce this project if necessary. You will find the complete list of footprints that I have used there.

Below, you can see my completed associations table:

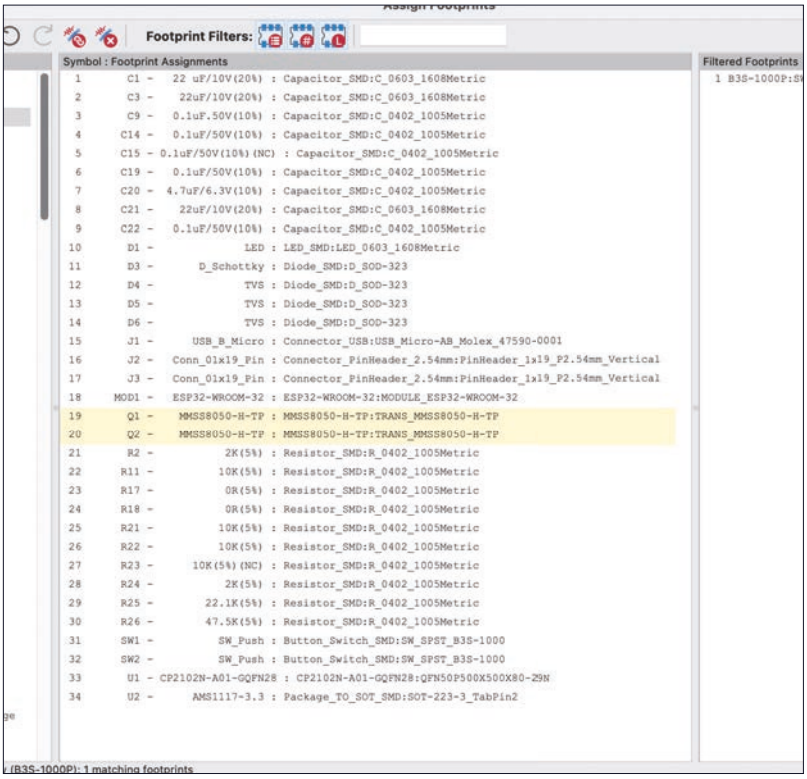


Figure 4.2.5.2: The completed associations table.

If you cannot find any footprints, especially those from Snapeda and Digikey, ensure you have correctly installed the necessary libraries.

Once you have completed the associations, continue with step four of the workflow: wiring.

4.2.6. Schema 4 - Wiring

With the help of the schematic reference from Espressif, wiring your schematic should be straightforward. You have already placed the symbols in their functional groups. As in previous projects, the strategy is to use graphical line wires to connect pins within the same functional group and net labels for everything else.

Place the schematic editor window next to the PDF viewer with the reference design, and start wiring. Below, you can see the completed wiring for the first functional group, the power supply:

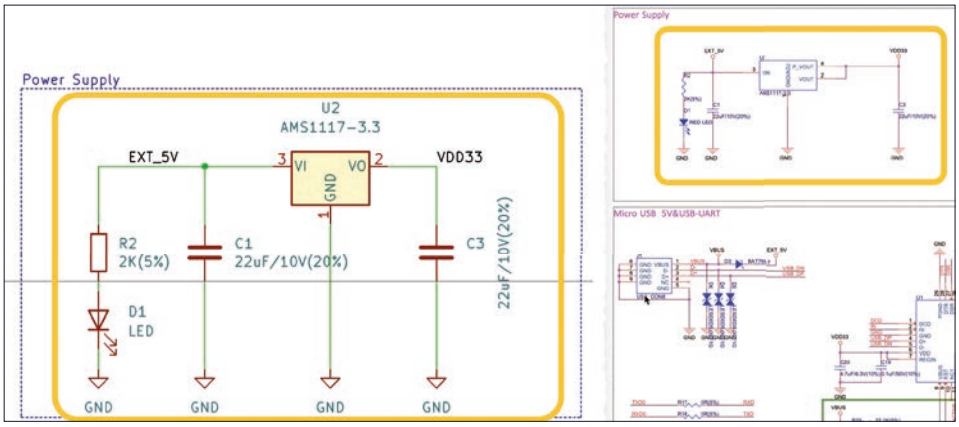


Figure 4.2.6.1: Power supply group wiring is complete. Left: my wiring. Right: reference.

Continue with the switch buttons:

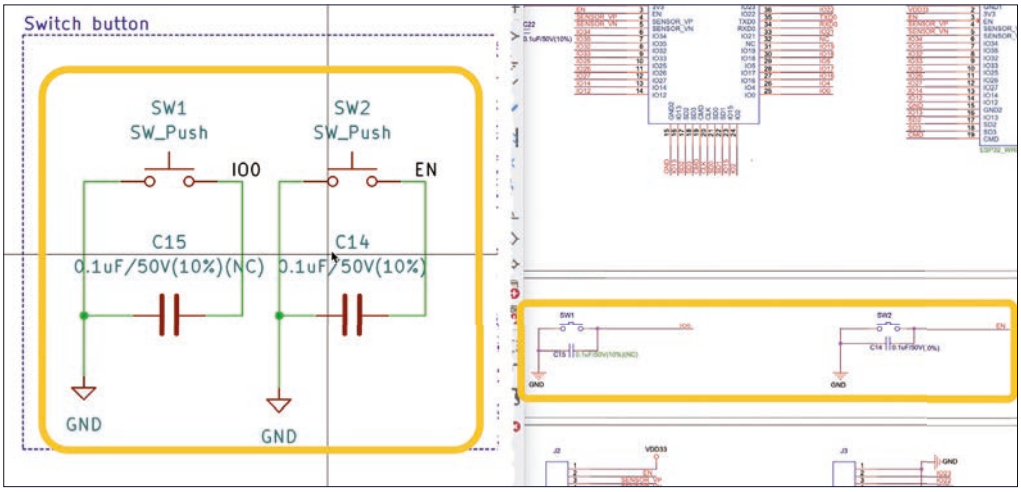


Figure 4.2.6.2: Switch buttons group wiring is complete.
Left: my wiring. Right: reference.

Next up, the connectors. Use net labels for all wirings.

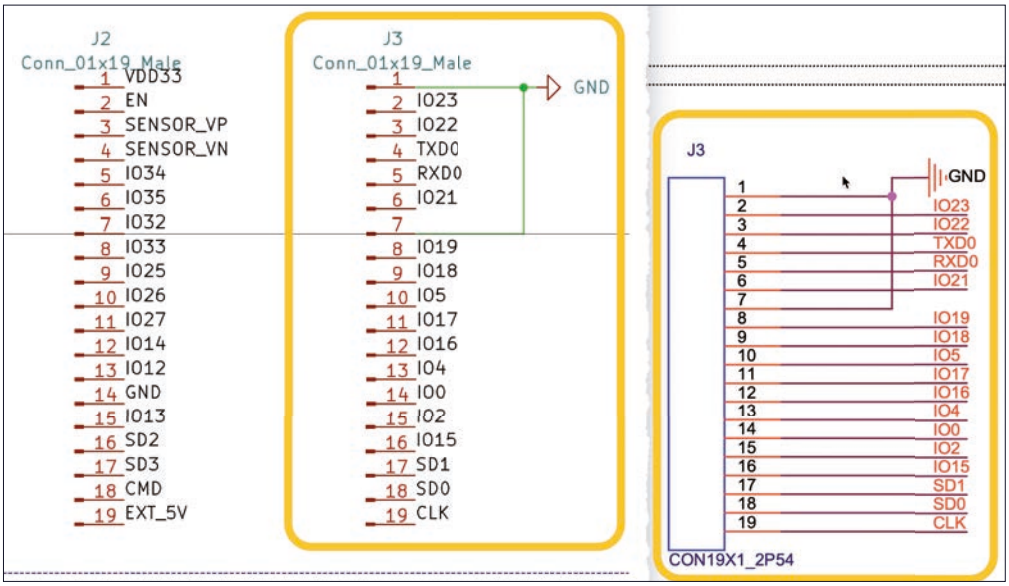


Figure 4.2.6.3: Connectors group wiring is complete. Left: my wiring. Right: reference. It does not showing a reference for J2.

I continued with the wiring of the ESP32 module group components. You can see the result below:

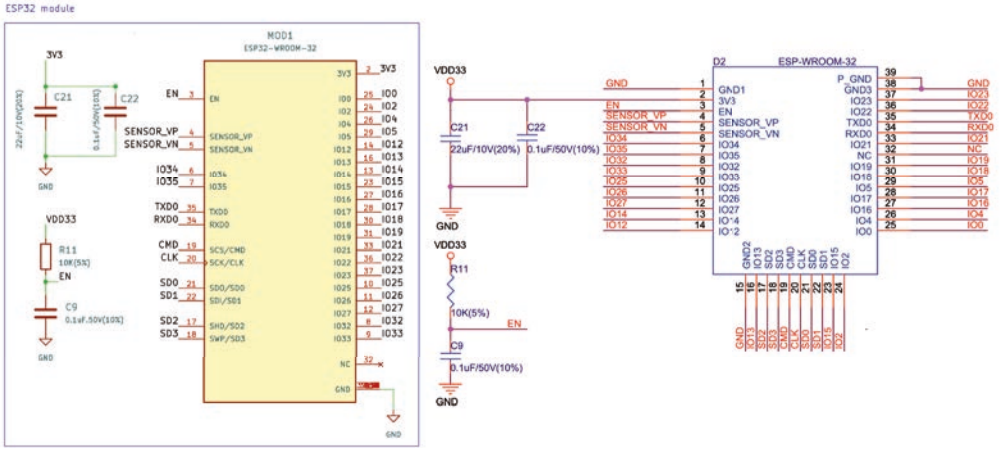


Figure 4.2.6.4: ESP32 group wiring is complete. Left: my wiring. Right: reference.

Finally, the Micro USB group. This is the most complicated part of the schematic. As with symbol MOD1, symbol U1 also has overlapping GND pins. I have edited the symbol with the Symbol Editor to separate the two GND pins (pins 3 and 29). The result, after wiring, looks like this:

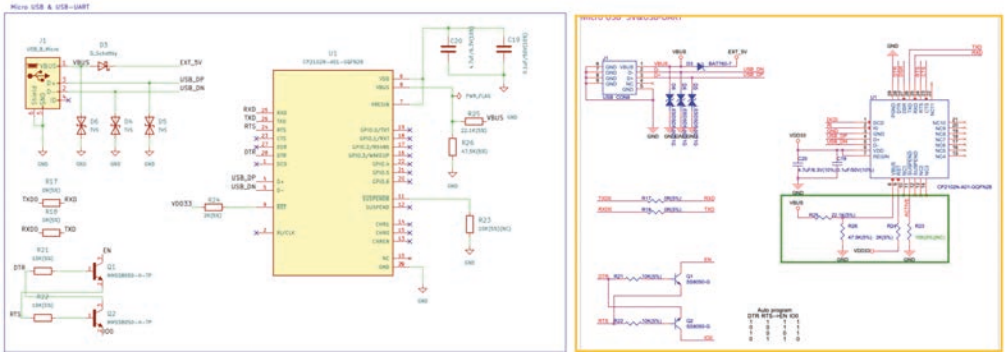


Figure 4.2.6.5: Micro USB group wiring is complete. Left: my wiring. Right: reference.

Here is the schematic, with the wiring complete:

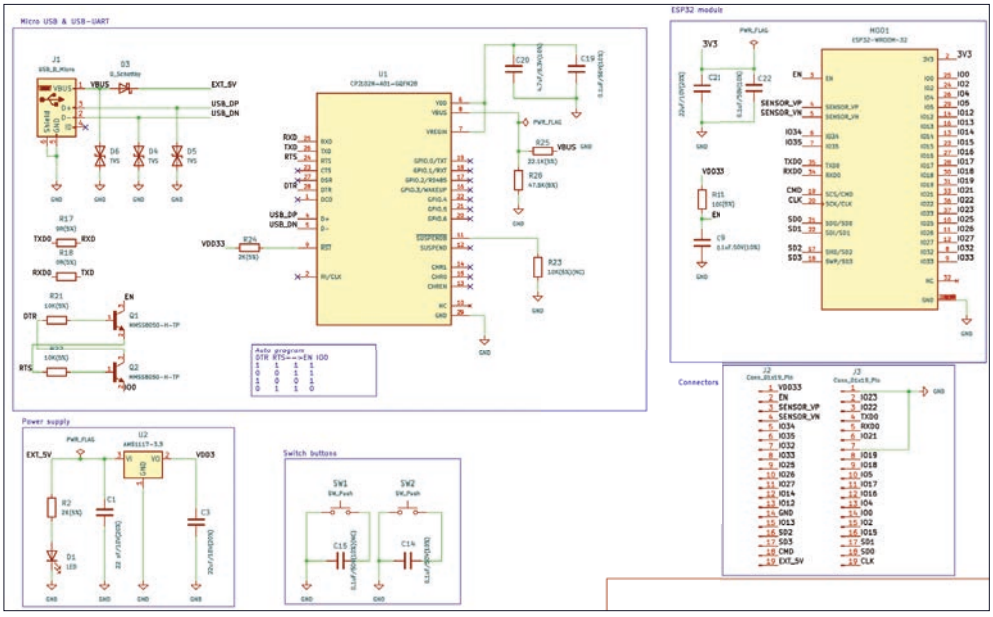


Figure 4.2.6.6: Schematic wiring is complete.

Unfortunately, the schematic in its current version contains a significant bug. Can you find it? Take a moment to find it, and when you are ready, continue to the next segment, where I'll show you (and fix) the bug.

Schema 4 - Wiring bug fix

The schematic in the previous segment contains an important bug. Perhaps you have noticed it and fixed it already. The bug is in pin 2 of the ESP32 module. This pin is the power input, and as per the reference design, it must be connected to the VDD33 net.

I forgot to add the VDD33 net label, and as a result the ESP32 module would be left without power! Let's go ahead and fix this bug.

Simply duplicate the VDD33 net label, and attach it to the wire that comes out of pin 2 of the ESP32 module. Remove the 3V3 flag that I added in the previous segment. Here is the detail:

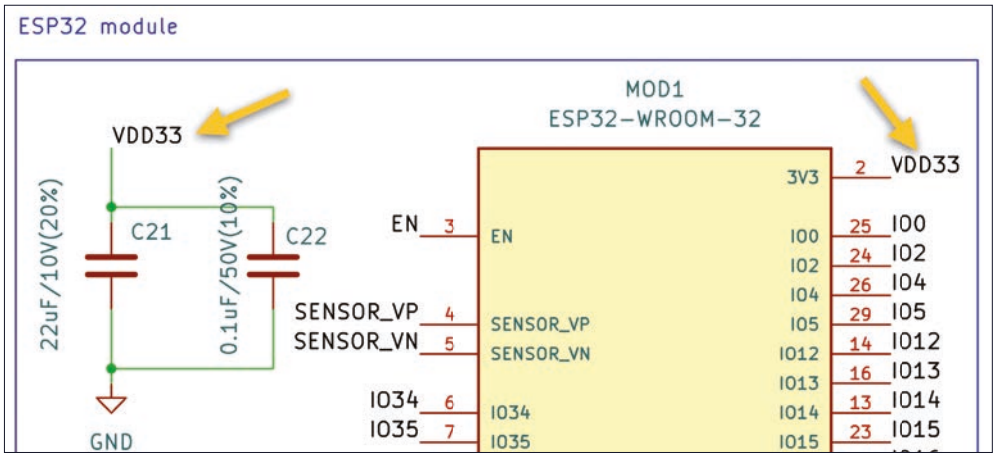


Figure 4.2.6.7: Attached the VDD33 label to pin 2 of the ESP32 module.

Here is the completed schematic with the correction:

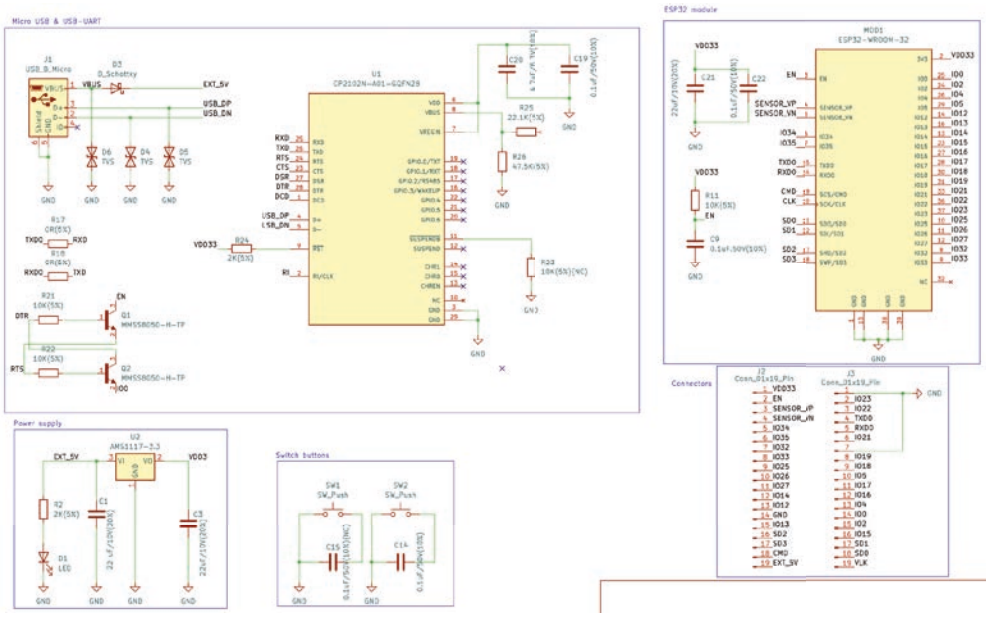


Figure 4.2.6.8: Full schematic that includes the bug fix.

Kudos to the anonymous reader who found and reported this bug. The remainder of this project does not contain this bug.

Let's continue with part five of the workflow. You have already created the nets, so you must create the net classes and associate them with nets.

4.2.7. Schema 5 - Nets and Net Classes

In the wiring step you completed in step four of the workflow, you used net labels alongside graphic wire lines. As a result, you have already created all required named nets. Therefore, to complete step five of the schematic design workflow, you only need to set up the net classes and associate nets with those classes.

In addition to the default class, create three more classes. In each class, assign the nets that I list below:

- Default (already exists): assign all net except those listed below.
- GPIO: All IOxx nets.
- POWER: EXT_5V, VBUS, VDD33, GND.
- USB: RXD, RXD0, TXD, TXD0, USB_DN, USB_DP.

Below is my schematic setup window showing the four net classes with some net allocations.

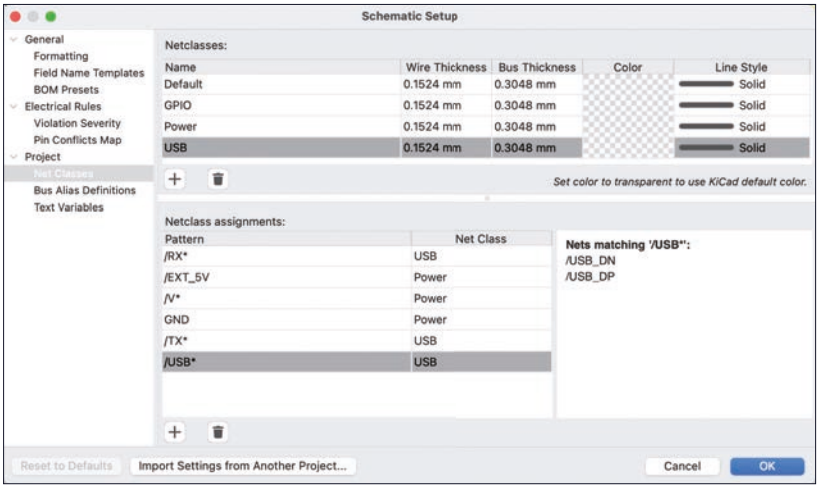


Figure 4.2.7.1: My net class setup.

Let's continue with the Electrical Rules Check.

4.2.8. Schema 6 - Electrical Rules Check

It's time to do the Electrical Rules Check. I usually forget a few simple things, like annotating GND or power labels. My ERC window indicates more issues than I expected. There is a variety of those issues. A few issues related to power pins (that must be marked with a power flag), unconnected pins, and labels that are not properly attached to pins, even

though they look correct in the schematic. See my full list of errors below (of course, your list will likely look different).

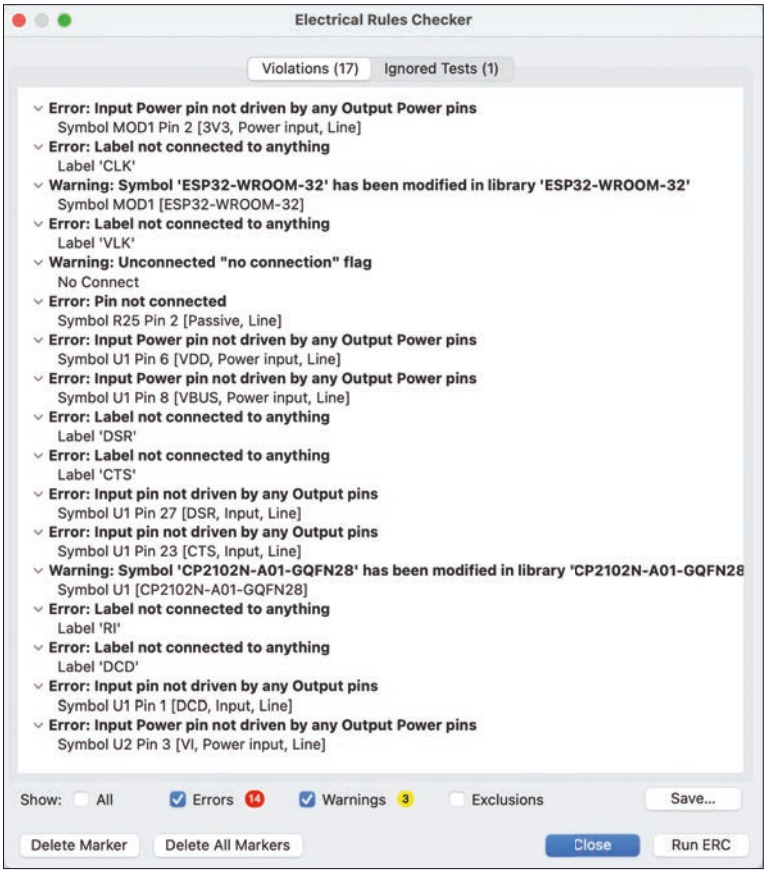


Figure 4.2.8.1: The ERC report.

A few errors are indicating that pins marked as "input power pins" are not connected to an "output power pin." To fix this error, attach POWER_FLAG symbols to the affected nets. Below, you can see the fix for the first error:

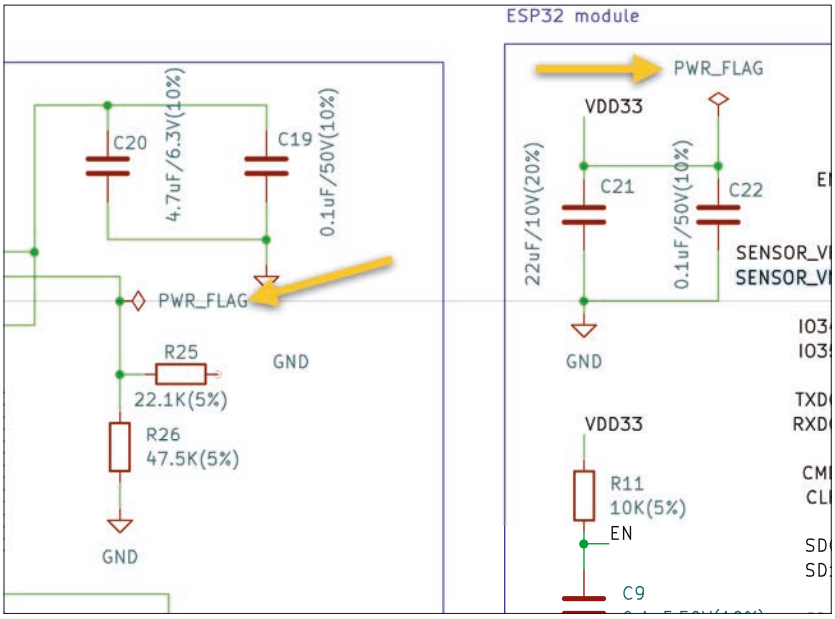


Figure 4.2.8.2: Fixing an "Input Power pin not driven by any Output Power pins" error.

Another group of errors concerns nets that are not connected to anything. I created these nets because they exist in the reference schematic, but on closer examination of the original, I confirm that they are not connected to anything. See three examples below:

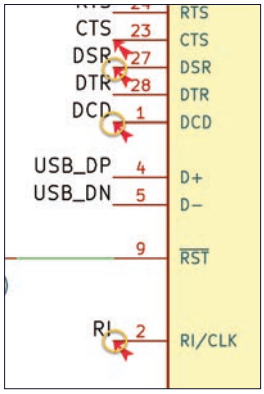


Figure 4.2.8.3: Fixing a "Label not connected to anything" issue.

To fix these errors, delete the net labels "CTS", "DSR", "DCD", and "RI" and replace them with the unconnected symbol like this:

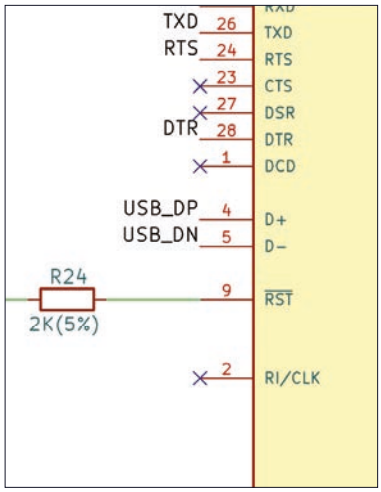


Figure 4.2.8.4: Fixed the "Label not connected to anything" issue.

Another issue I had was the warning "Unconnected "no connection" flag", which was due to a stray unconnected flag. I fixed the issue by deleting the stray "no connection flag". I also had a mistyped net label in J3, where I typed "VLK" instead of "CLK".

There was a "Pin not connected" error because I forgot to attach the net label "VBUS" to the unconnected pin of R25.

The last issue was the error "Input Power pin not driven by any Output Power pins". This was caused by not attaching U1 pin 6 to the VDD33 net. You can see the fix below.

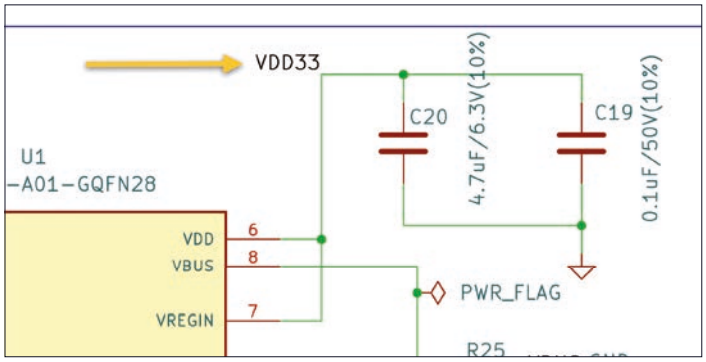


Figure 4.2.8.5: The fix for the input power pin issue.

There are two remaining warnings in the ERC list related to symbols U1 and MOD1. I edited these symbols outside the library to rectify their additional GND pins, so I can safely ignore these warnings and continue with the project.

Let's finish the schematic design workflow in the next segment of this chapter, where you will add additional comments to the schematic.

4.2.9. Schema 7 - Comments

You have already done most of the commenting work earlier in the process. The schematic design contains boxes to enclose the functional groups, and those boxes have names.

The reference design contains a truth table for the DTR and RTS signals, and I'd like to include that in my schematic using a text box. Below, you can see the truth table in the reference design (right) and the location I'd like to place it in my design:

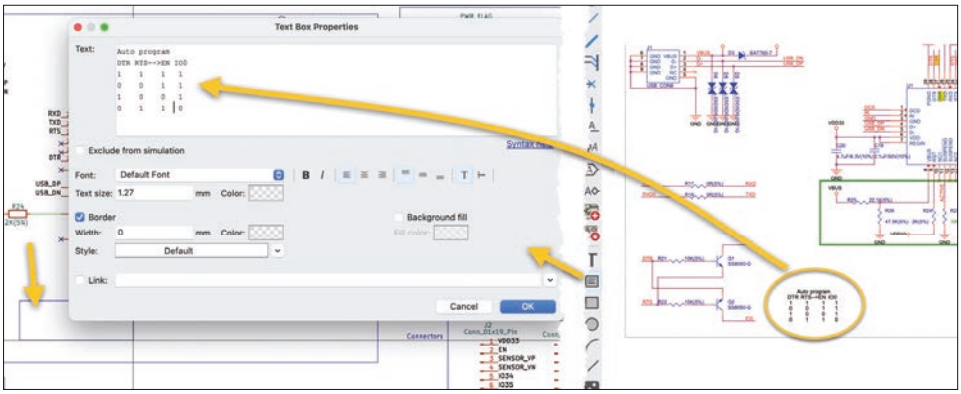


Figure 4.2.9.1: The truth table in the reference schematic.

Below, you can see the text label that contains the truth table text and its location in my schematic design:

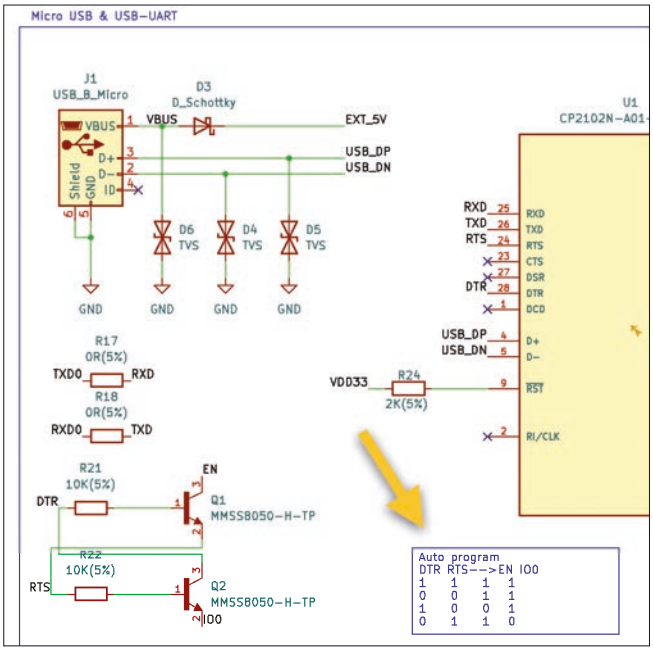


Figure 4.2.9.2: The truth table in my schematic design.

Another improvement I'd like to make is to reduce the size of the component value text. The default size is too big, causing text to spill over other elements of the schematic, like in the example below:

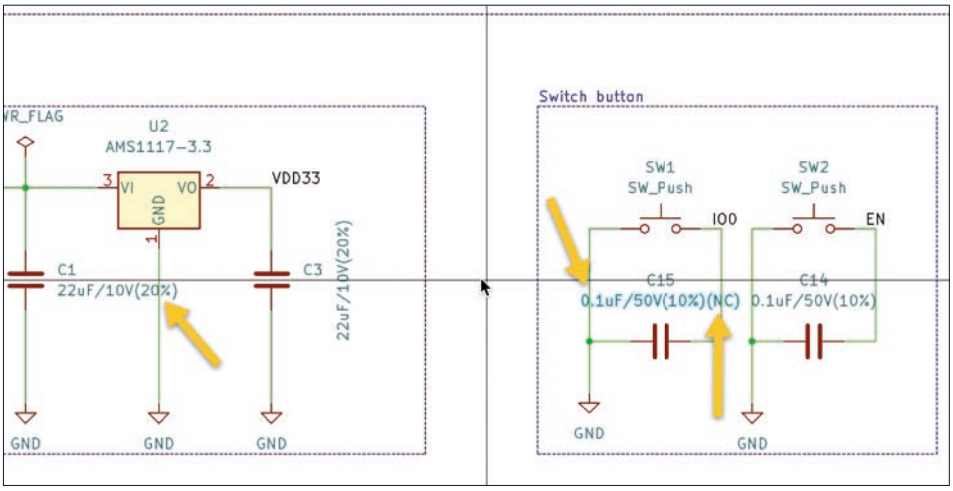


Figure 4.2.9.3: Value text is too big.

To change the size of all value text elements, use the "Edit Text & Graphics Properties..." tool from the "Edit" menu. Set it to change all "Values" to "Text size" "1 mm":

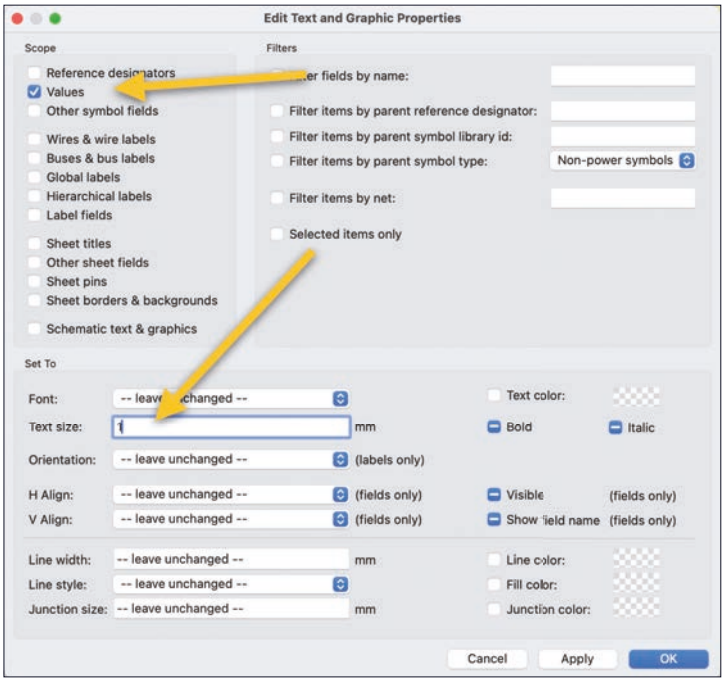


Figure 4.2.9.4: Changing size for all Value text elements.

This change will allow value text elements to better fit in the available space:

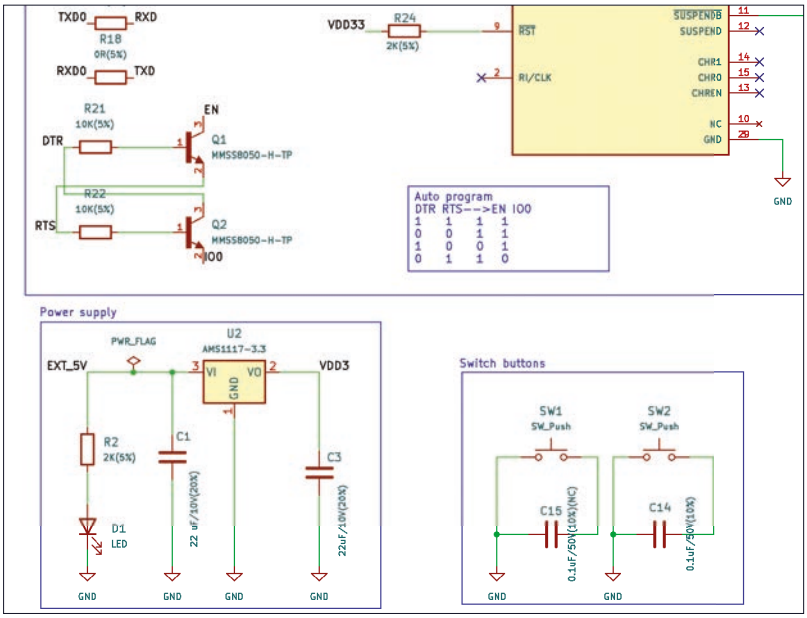


Figure 4.2.9.5: Value text fits better.

This completes the schematic design workflow. Let's continue with the layout.

4.3. Layout design

In the previous chapter, you completed work on the schematic design of the ESP32 development kit clone. You are now ready to begin work on the layout design.

Before you start work in the PCB editor, take a few minutes to review the objectives of the layout design workflow. The main objective is to design an ESP32 development board that can fit in a mini breadboard. It will have a simple rectangular shape, with two pin rows along the long edges and the USB connector at one of the narrow edges. Below is the 3D rendering of the board that I designed as I worked on this project:

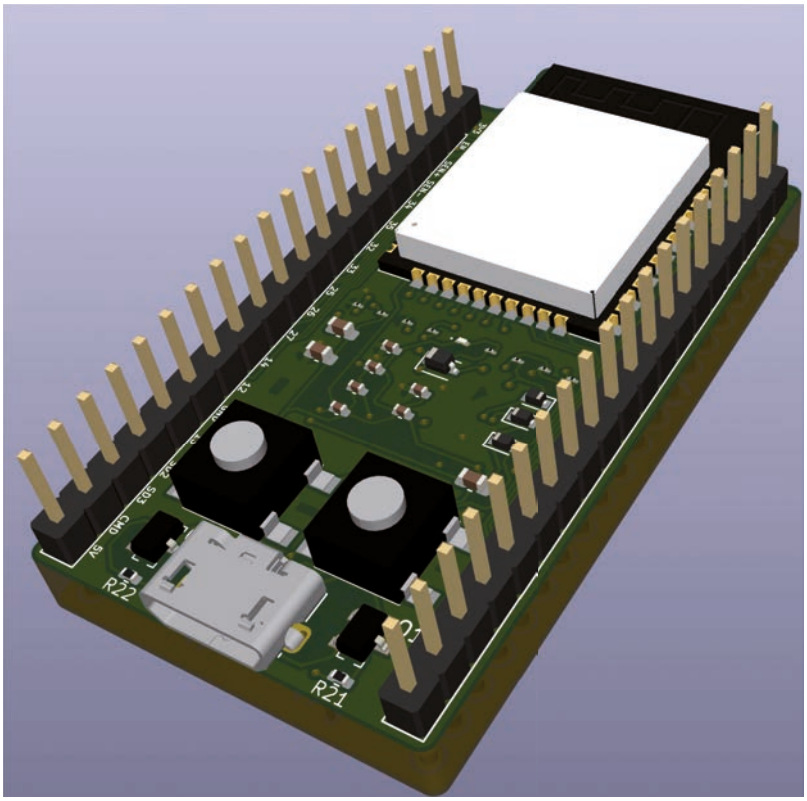


Figure 4.3.1: The main objective of this project.

Below is the layout of the board, as it looked when I finished work on it:

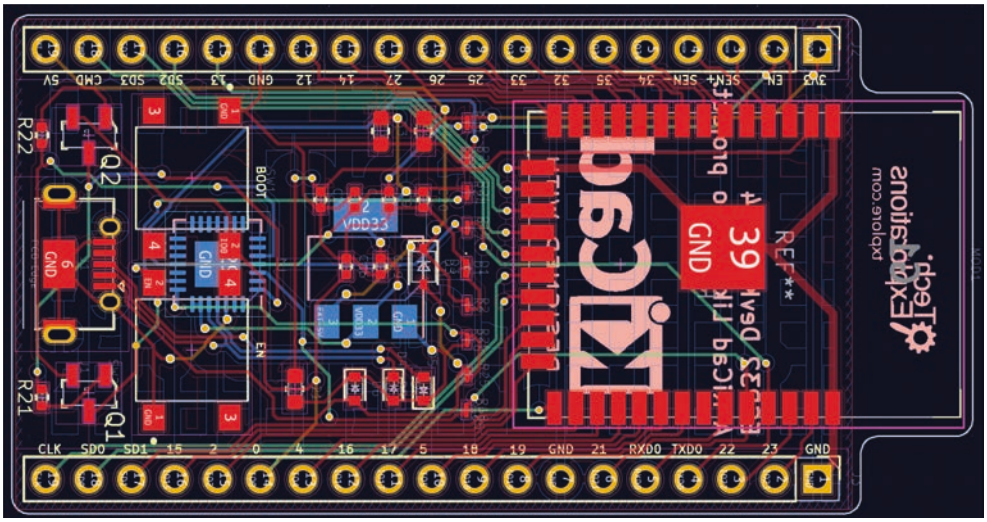


Figure 4.3.2: The PCB as it looks in the layout editor at the end of this project.

There are two main geometrical constraints:

1. Minimize the board's size to minimize its cost.
2. Ensure that it will fit in a double mini-breadboard.

In the photograph below, I show a measurement that I took that shows the required distance between the pin rows:

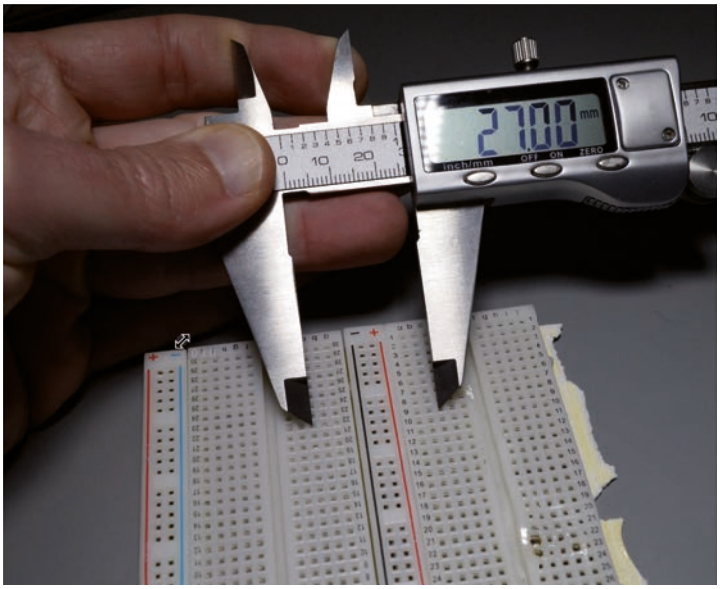


Figure 4.3.3: Measuring the distance between the rows.

Based on this measurement, I will design the PCB so that the distance between the rows is approximately 27 mm. This will allow me to combine my ESP32 development kit with two mini-breadboards, as in the photograph above.

In addition to the above, this project's layout design will allow you to practice:

1. Working with differential pairs.
2. Working with components on both sides of the board.
3. Adjust the default design rules to allow small components that otherwise would cause violations.

Let's begin by setting up the layout design editor.

4.3.1. Layout 1 - Setup

Let's set up the layout editor to match the work requirements ahead. Open the Board Setup window and edit the following settings (if I have not listed a setting below, leave it at its default value). I have checked that the minimums I specified below are within the capabilities of my PCB manufacturer, NextPCB, as published on their website

(https://www.nextpcb.com/pcb-capabilities). You should always check the capabilities of your preferred manufacturer before setting up your project's PCB editor.

- Board Stackup.
  - Physical Stackup.
    - Copper layers: 4.
  - Text & Graphics.
    - Defaults.
      - Silk Layers.
        - Line Thickness: 0.15 mm.
        - Text Width: 0.8 mm.
        - Text Height: 0.8 mm.
        - Text Thickness: 0.1 mm.
  - Design Rules.
    - Constraints
      - Minimum track width: 0.1 mm.
  - Net Classes.
    - GPIO and USB.
      - Clearance: 0.1 mm
      - Track Width: 0.1 mm
      - Via Size: 0.5 mm
      - Via Hole: 0.35 mm
      - uVia Size: 0.25 mm
      - uVia Hole: 0.1 mm
      - DP Width: 0.1 mm
      - DP Gap: 0.1 mm
      - Assigned nets GPIO: all IOxx nets.
      - Assigned nets USB: RXD, RDX0, TXD, TXD0, USB_DN, USB_DP
    - POWER.
      - Clearance: 0.1 mm
      - Track Width: 0.15 mm
      - Via Size: 0.6 mm
      - Via Hole: 0.4 mm
      - uVia Size: 0.25 mm
      - uVia Hole: 0.1 mm
      - DP Width: 0.2 mm
      - DP Gap: 0.2 mm
      - Assigned nets: EXT_5V, VBUS, VDD33, GND
    - Default:
      - Clearance: 0.1 mm
      - Track Width: 0.15 mm
      - Via Size: 0.6 mm

- Via Hole: 0.35 mm
- uVia Size: 0.25 mm
- uVia Hole: 0.1 mm
- DP Width: 0.1 mm
- DP Gap: 0.15 mm

- Data: will be set up later.

I have reduced the GPIO and USB net class sizes because some of the board components are so small that the default design rules would trigger multiple clearance violations. In the example below, the arrows point to resistors that use the 0402 package. To prevent clearance violation errors from occurring, I have changed the minimum clearance for members of the GPIO class to 0.1 mm.

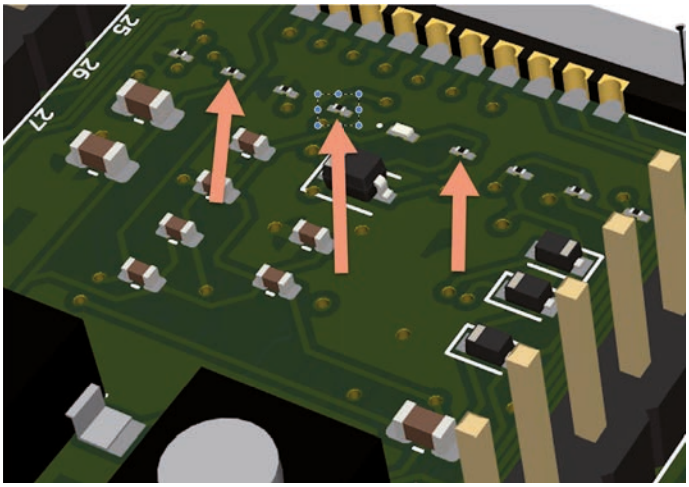


Figure 4.3.1.1: Small components require smaller clearances in the design rules.

Similarly, this board contains a couple of differential pair tracks. Those differential pairs are located between the ESP32 module pads and the pins row in the top left of the board. Because the distance between the pads is very small, I had to reduce the clearance of the differential pairs in the design rules.

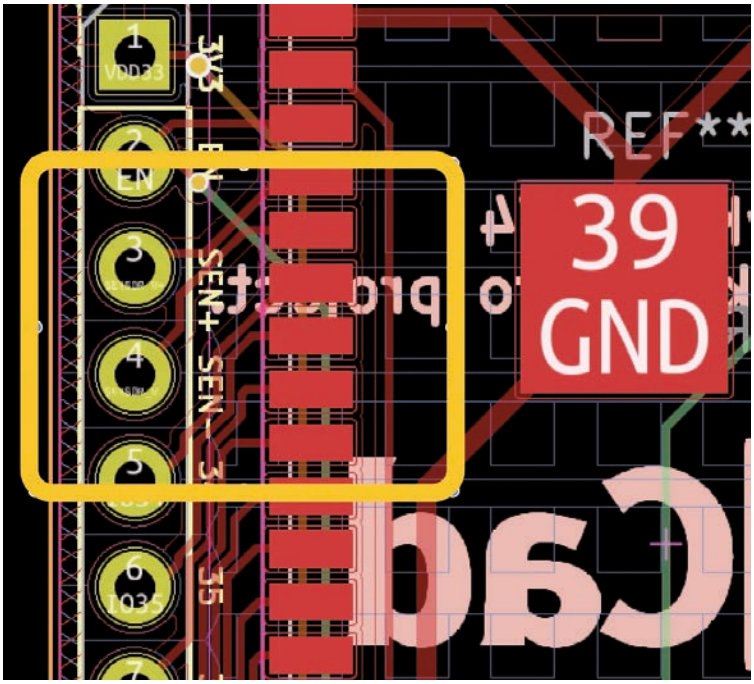


Figure 4.3.1.2: Differential pairs require adjusted clearances.

Finally, set the editor page in the Page Settings window:

- Page Settings.
  - Issue Date: copy today's date.
  - Revision: 1
  - Title: ESP32 Custom Dev Kit

Save your work, and continue to step two of the layout workflow.

4.3.2. Layout 2 - Outline and constraints

Let's continue with Step two of the layout workflow. Here, you will draw a rough outline of the board in the Edge.Cuts layer based on the constraints I described in the setup step.

The primary geometrical constraint is to design the board so that the two pin headers are approximately 27 mm apart. This will allow the board to plug into a double mini-breadboard. In the photo below, I show my caliper measuring the distance between the two rows where I want to place the pin headers:

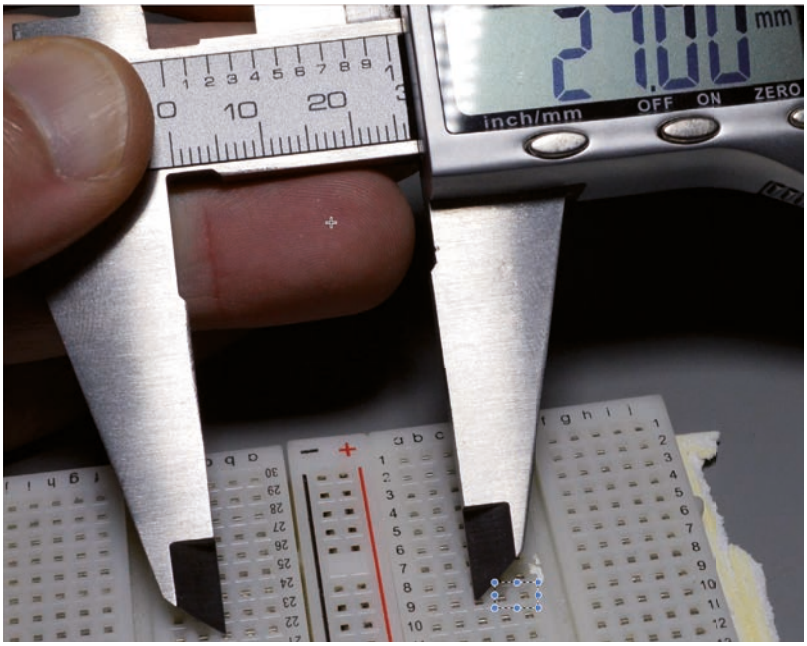


Figure 4.3.2.1: The main geometrical constraint is the distance between the pin headers.

The 27 mm measurement I took with my caliper contains some tolerance, so I can vary it in the design. From my experimentation, a distance between 26.90 mm and 27.10 mm is acceptable. In my design, I will aim for the middle of this range.

Start by importing the footprint in the editor using the "Update PCB from Schematic" tool. You can see my editor window below with the bundle of the footprints just imported:

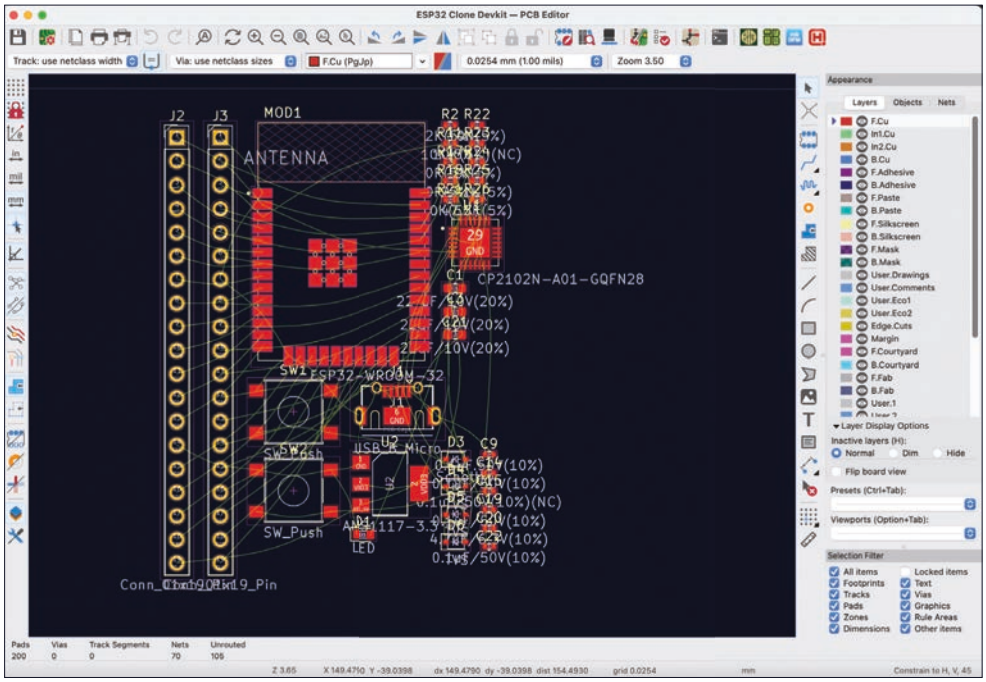


Figure 4.3.2.2: Imported footprints from schematic.

Since the geometrical constraints call for a specific distance between the two pin headers, I will use them to draw the rough outline of the board. Consult the reference design to find the correct side for each pin header. Header J2 should go to the left and J3 to the right.

Take footprint J2, place it towards the middle left of the editor, and lock it. You will use J2 as an anchor to measure a distance of approximately 27 mm from the second header, J3. I have set my grid size to 0.127 mm. I have set J3 to be 26.92 mm away from J2 in the figure below. Notice the dx and dy values and grid size in the bottom status bar. A reminder: to reset the dx and dy counters, press the space bar.

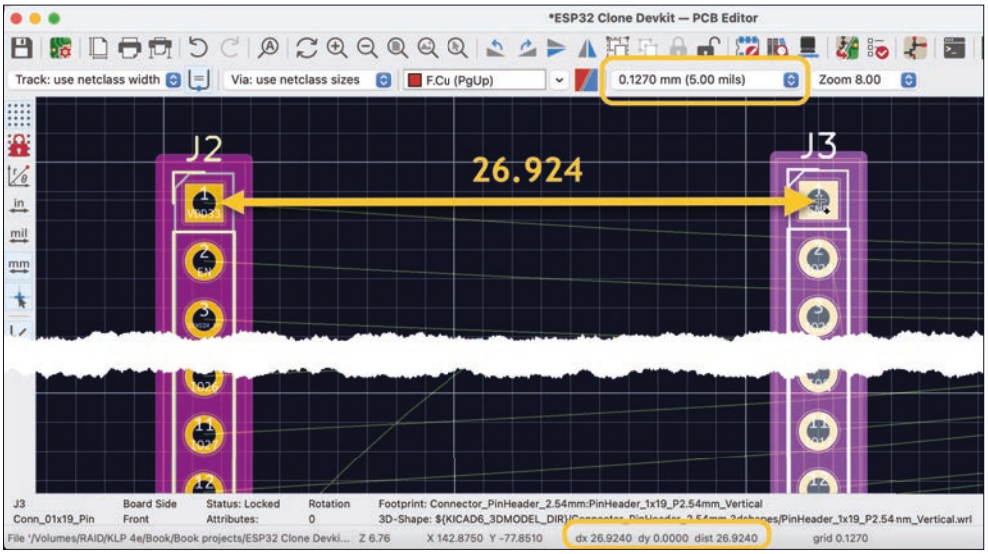


Figure 4.3.2.3: Setting J3 26.92 mm away from J2.

In the figure below, I have completed the placement of J3 and locked it in place. I then used the measurement tool to insert this measurement into the User.1 layer:

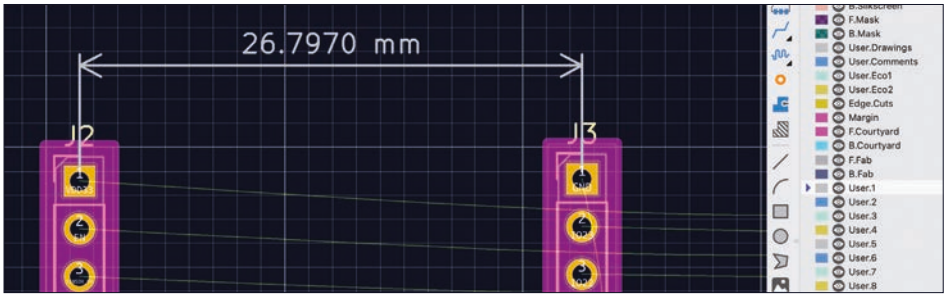


Figure 4.3.2.4: Pin header placement is complete.

Ensure the J2 and J3 are locked. Do a sanity check and confirm that the ESP32 module, the board's largest component, will fit between the two headers. As you can see below, it will:

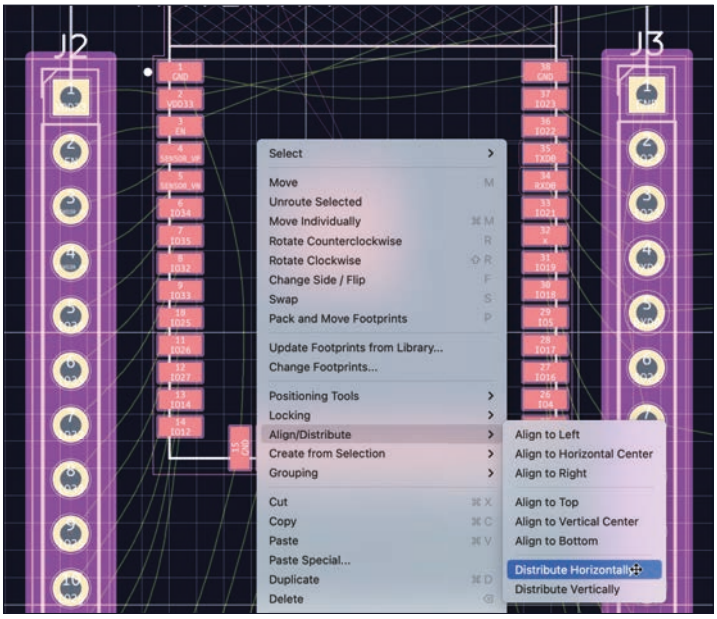


Figure 4.3.2.5: Centering the ESP32 module between the pin headers.

To ensure that the ESP32 module is centered between the pin headers, select all three footprints, right-click to open the context menu, and select "Distribute Horizontally" from the "Align/Distribute" menu. Also, the ESP32 module should be shifted upwards to extend the antenna area outside the mainboard area. This ensures that the antenna circuitry is away from potential interference from other board electronics.

Lock the ESP32 module. You can see the current state of the board below:

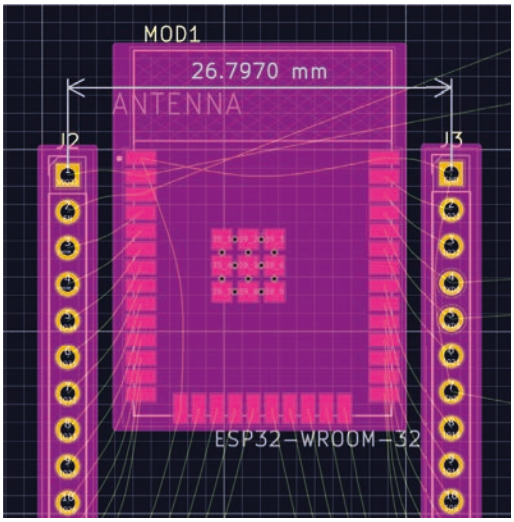


Figure 4.3.2.6: The first three footprints are locked in place.

Now that you have the footprints that define the shape and size of the board, you can proceed to draw the rough outline. Change the grid size to 0.254 mm and enable the Edge. Cuts layer, and choose the rectangle drawing tool. Below, you can see the rough outline in yellow:

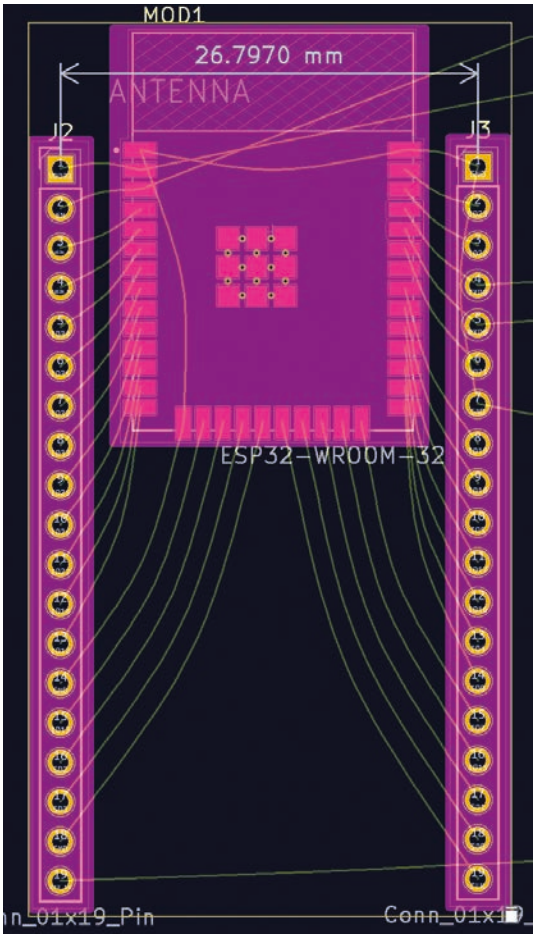


Figure 4.3.2.7: Completed the rough outline of the board.

Continue in the next segment, where you will place the rest of the board footprints inside the rough outline.

4.3.3. Layout 3 - Place components

Continue to move the rest of the footprints within the rough outline. Move the footprints close to the board outline to make them easily accessible. At the start of the placement process, my editor looks like this:

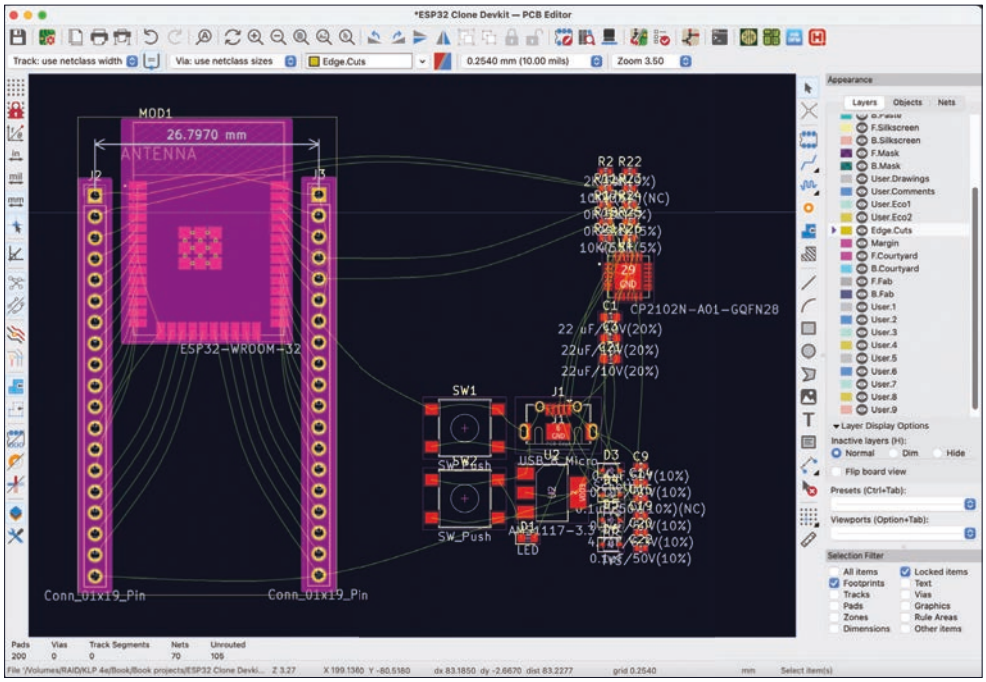


Figure 4.3.3.1: Beginning of the footprint placement process.

You can use the reference design to help with the appropriate position of each footprint but remember that you have the freedom to vary. You can see the reference design tomorrow:

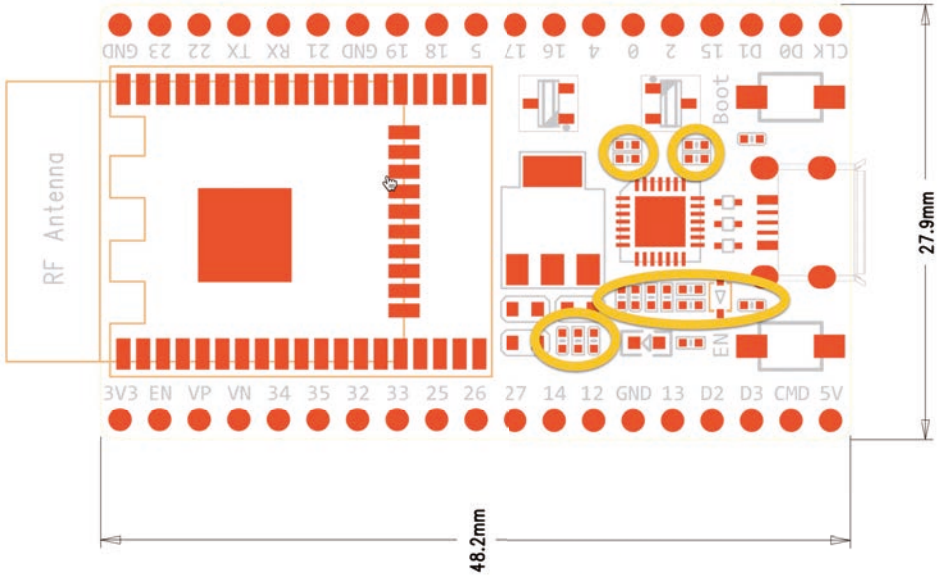


Figure 4.3.3.2: The board reference design.

In the figure above, I have used yellow circles to mark a few very closely positioned components. This is not a problem in the reference design since an assembly machine will manufacture the board. However, I plan to assemble my board by hand, and such packed positioning will make hand assembly very difficult.

Instead of blindly following the reference design, I chose to make a variation. I placed the voltage regulator and USB-UART bridge in the back of the board. This will release much of the front of the board to space out the small capacitor and resistor footprints, making assembly easier. I have decided to place other components, such as the buttons and the USB connector, as they appear in the reference design.

Continue with the USB connector (J1) and the buttons (SW1 and SW2). Use KiCad's alignment tools to center and justify the footprints. Lock J1, SW1, and SW2 in place. Below is my layout at this time:

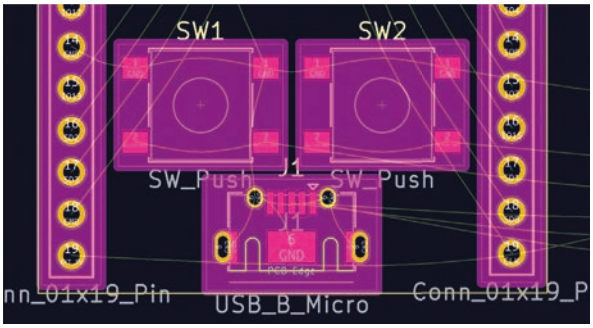


Figure 4.3.3.3: Completed placement for J1, SW1 and SW2.

I have decided to place U1 and U2 on the back of the board. Position them close to the front of the board (where the J1 connector is). To place a footprint in the back of the PCB, select "Back" in the Side field of the footprint's properties window:

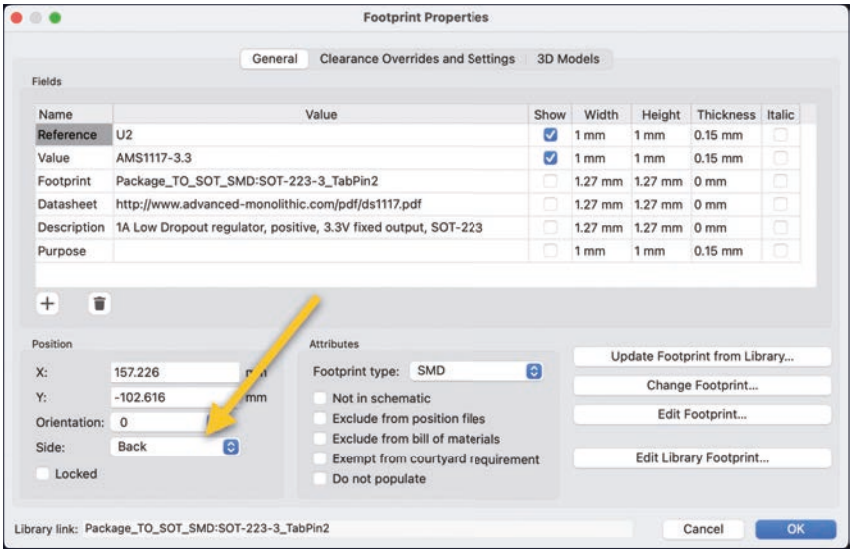


Figure 4.3.3.4: Placing a footprint in the back of the board.

Lock U1 and U2 in place. At this time, my layout looks like this:

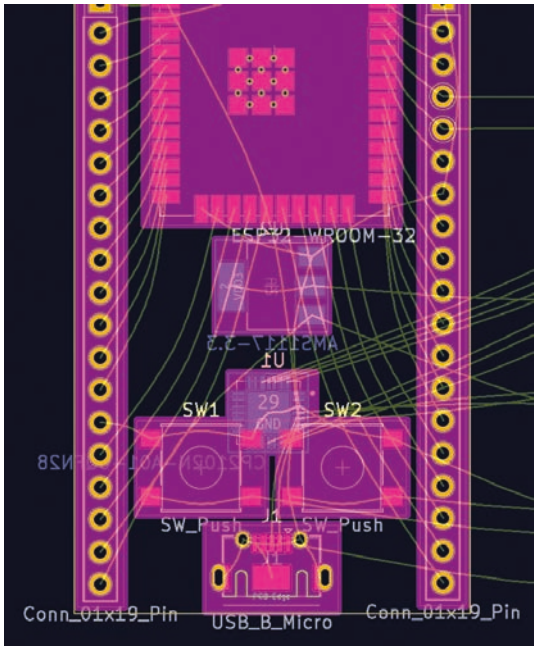


Figure 4.3.3.5: Blue: back layer. Red: front layer.

Ensure that the courtyard lines (purple or light blue lines around the footprints) do not intersect with other lines and are not outside of the orange board outline. Courtyard lines indicate the border of a footprint.

Continue with the transistors Q1 and Q2. Always align and justify, and lock in place when finished.

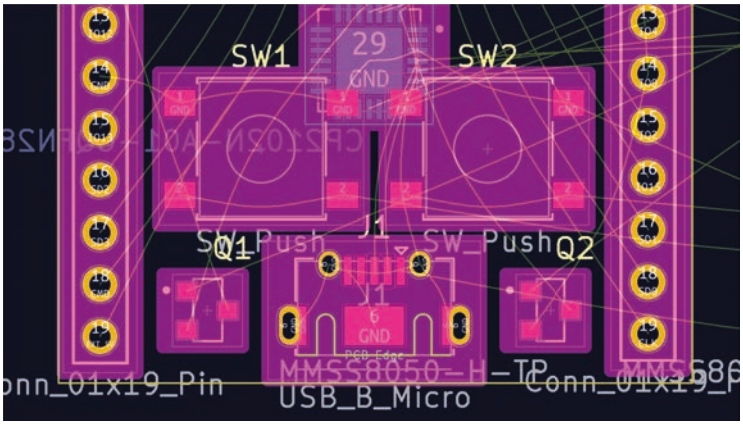


Figure 4.3.3.6: Placed Q1 and Q2.

Finally, continue to place the small footprints: LEDs, resistors, capacitors. These footprints will go in the free space between the buttons and the ESP32 module. To reduce clutter, disable the visibility of the back layer. Here is what my editor looked like before I began work:

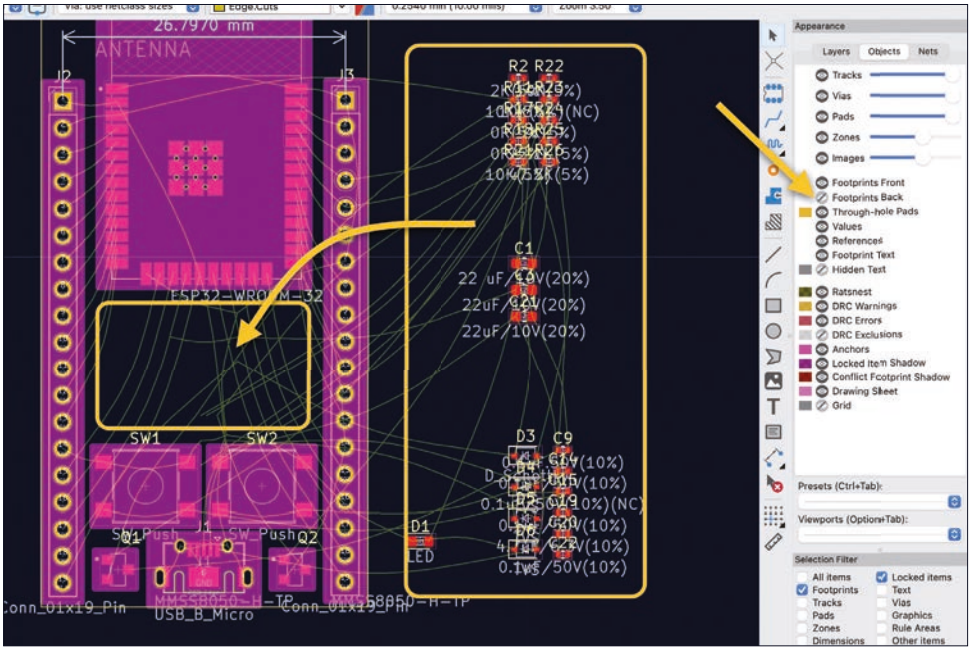


Figure 4.3.3.7: Preparing to place the small footprints.

During placement, observe these guidelines:

- Use ratnest lines and place footprints with pads near their closest connection.
- Place footprints in groups: place resistors close to other resistors and capacitors close to other capacitors.
- Avoid mixing footprints of different types, as this will increase the risk of errors in assembly.
- Justify and align frequently.
- Space out as much as possible so that you make use of all available space. This is very important if you plan to assemble the board by hand.
- Take as much time as you need to produce a high-quality design. Fixing it later will be costly.

You can see my small footprint placement below:

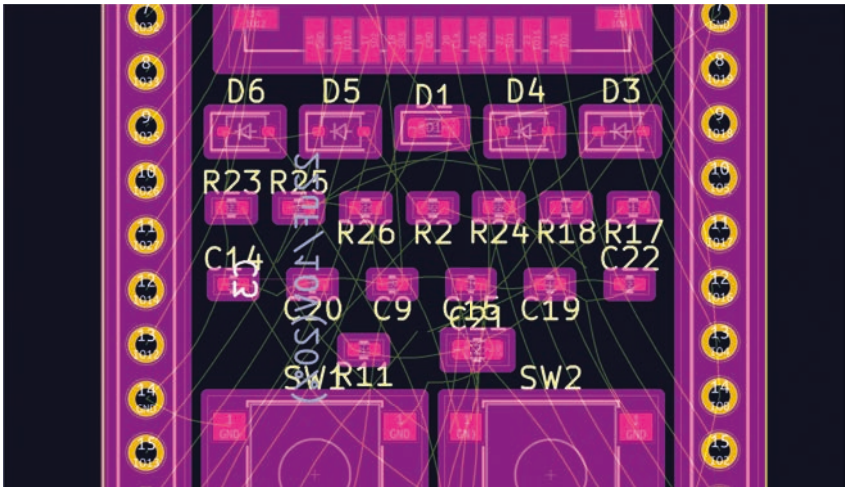


Figure 4.3.3.8: Completed placement of small footprints.

The footprint placement is now complete:

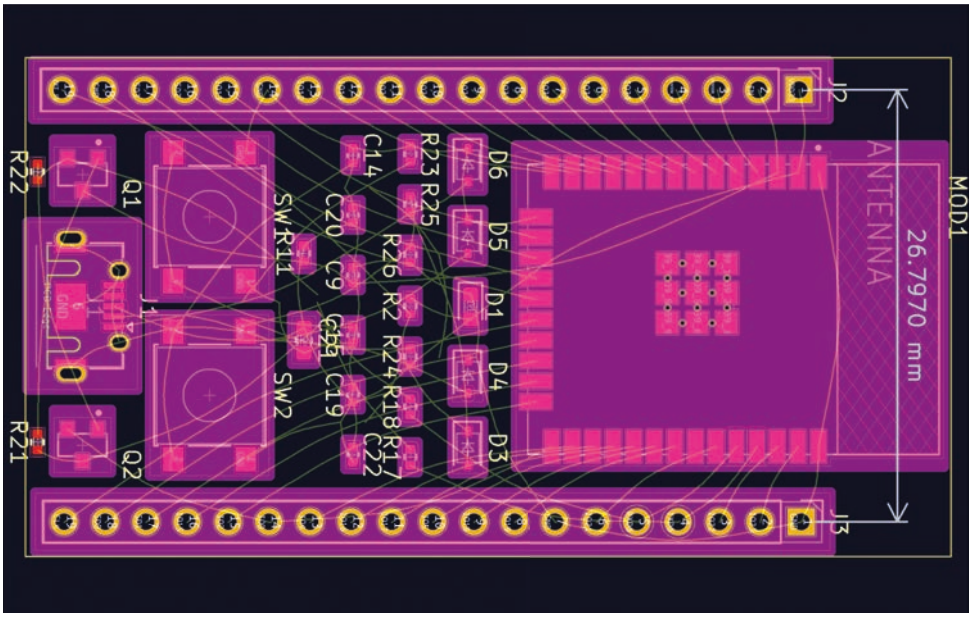


Figure 4.3.3.9: Showing the final placement of all footprints in the front of the board.

In the next segment of this chapter, you will return to step two of the layout workflow and refine the board outline.

4.3.4. Layout 2 - refine outline and DRC errors

You have completed the placement of all footprints in the front and back of the board. You can now give your board its final refined shape. As you have done in previous projects, the refinement will consist of two changes to the rough outline:

1. Replace all 90-degree corners with aesthetically pleasing rounded corners.
2. Remove substrate material that is not necessary.

This is what the board looks like at the moment:

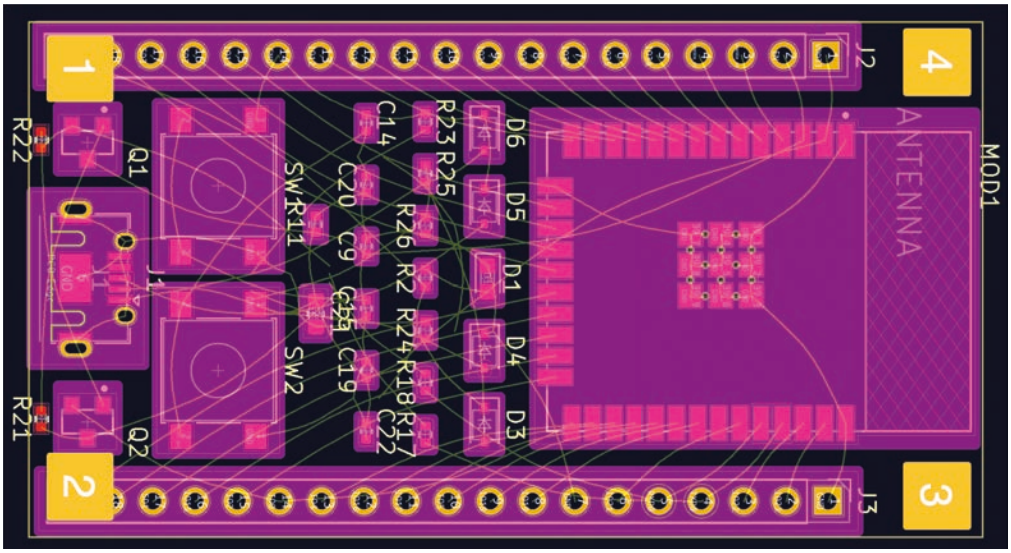


Figure 4.3.4.1: Plan to refine the rough outline.

I will replace the corners "1" and "2" in the figure above with rounded corners. At the top of the board, the unutilized substrate material is in corners "3" and "4." I will remove those, and I believe removing this material will make the board look more modern and streamlined.

As I drew the refined outline, I made multiple errors and warnings that caused problems with the 3D viewer and the DRC. Some of them also have the potential to increase the cost of manufacturing due to their more demanding tolerances. I list some of those errors and how to fix them below.

Through-hole pads of MOD1

The ESP32 module (MOD1) contains GND pins in its centre. Some of those pins are through-hole, and the rest are SMD. The screenshot below shows that pins 39_1 to 39_9 are SMD, while 39_10 to 39_21 are through-hole.

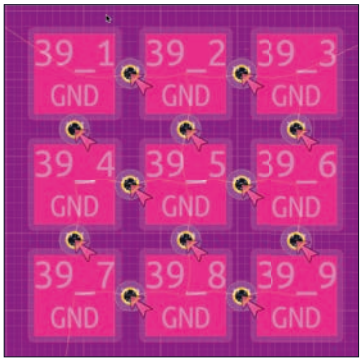


Figure 4.3.4.2: Pins 39_10 to 39_21 are very small and have triggered DRC errors.

The DRC produces an error for pins 39_10 to 39_21 because their annual rings are smaller than the minimum I have specified (the actual is 0.05 mm while the minimum is 0.1mm). While NextPCB can accommodate these sizes, it will increase the cost of my boards. Further, when I look at the actual module that I want to use, it does not include pins 39_10 to 39_20 (it only has the SMD pins for GND).

Based on the above, I have decided to address this issue by modifying the schematic and footprint libraries for MOD1. In the schematic, I will mark pins 39_10 to 39_21 as unconnected; in the footprint, I will delete pads 39_10 to 39_21.

Let's start with the schematic. In the schematic editor, double-click the MOD1 symbol to open its properties window, then click "Edit Symbol." Click the Pin Table button on the top toolbar in the symbol editor. In the table that appears, locate the rows for pins 39_10 to 39_21 and delete them. To delete each row, select it with your mouse and click the rubbish bin button at the bottom right of the window (see below).

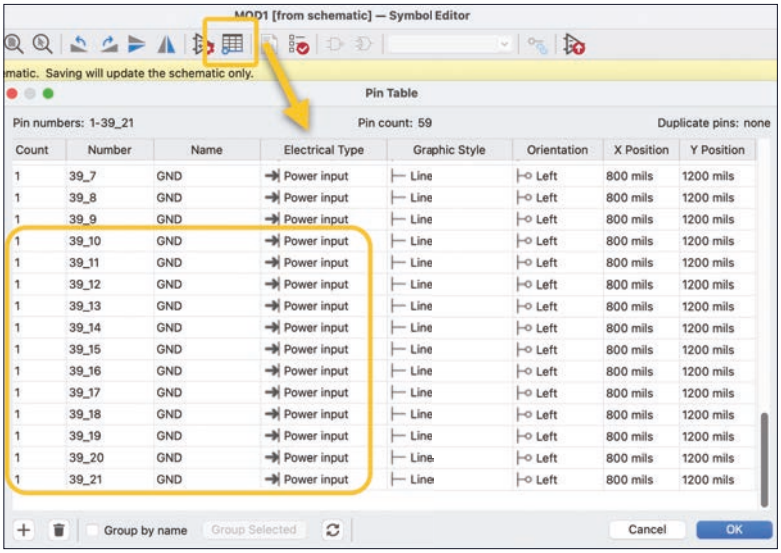


Figure 4.3.4.3: Delete pins 39_10 to 39_21.

Save the symbol and exit the symbol editor. Save the schematic, and continue in the PCB editor.

In the PCB editor, update the layout from the schematic (click the relevant button on the top toolbar). This will update pads 39_10 to 39_21 so they are no longer connected to the GND net. Then double-click the MOD1 footprint to bring up its properties window and click "Edit footprint". Delete pads 39_10 to 39_21 in the footprint editor, then save the updated footprint. The center of the footprint will look like this (showing the footprint editor window):

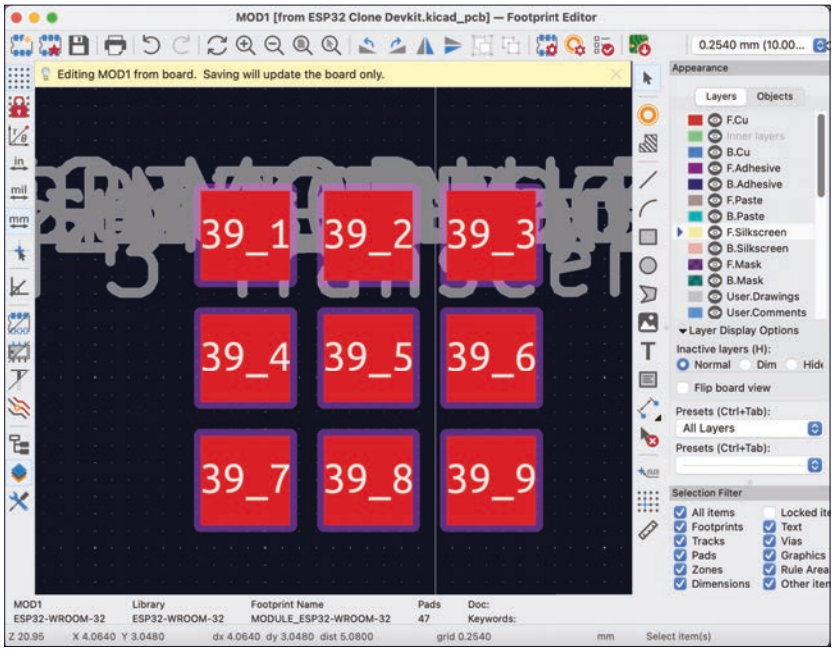


Figure 4.3.4.4: Deleted pads 39_10 to 39_21 in the footprint editor.

In the PCB editor, rerun the DRC tool and confirm that the GND through-hole pad errors are gone. Your layout should look like this:

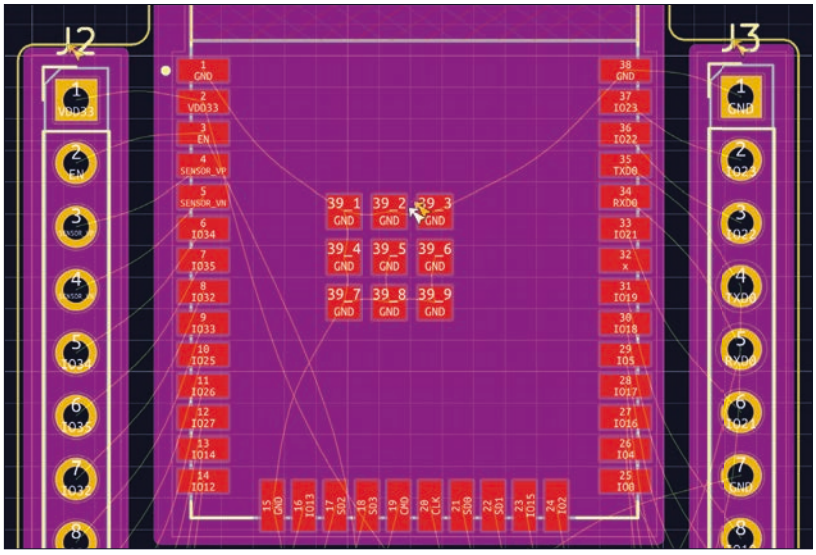


Figure 4.3.4.5: The errors relating pads 39_10 to 39_21 are resolved.

Let's continue with the other DRC errors.

Text labels

The DRC will complain when a text label in the silkscreen overlaps with other elements or outside the board outline. See the example below, where the label "MOD1" is outside the board, and J2 is outside the board and overlaps with other elements.

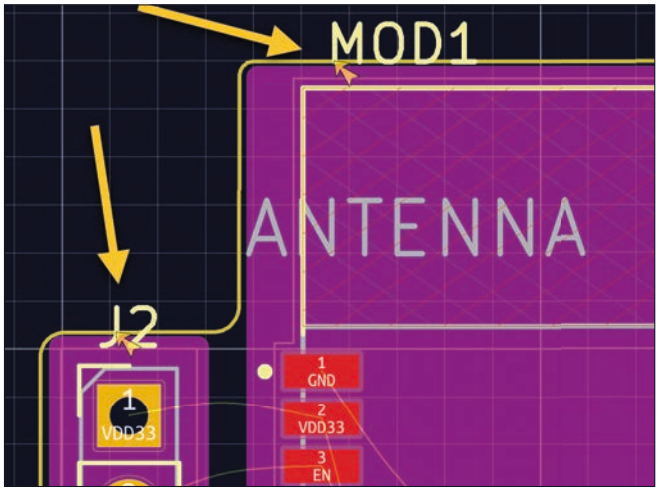


Figure 4.3.4.6: Text label bugs.

To fix such errors, move the labels inside the board outline and ensure that they don't overlap with other elements (this was my choice for MOD1). You can make them invisible (this was my choice for J2 and J3). To help useful text labels fit in the limited space available on this board, I used the Edit Text and Graphic Properties tool to change the text width, height, and thickness to smaller values (0.8, 0.8 and 0.1, respectively).

Malformed courtyard

Courtyard lines should form a closed polygon (the same applies to lines in the Edge. Cuts layer). In the example below, the courtyard line from the ESP32 module footprint is malformed.

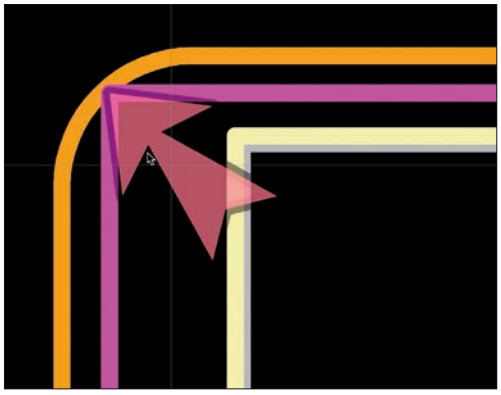


Figure 4.3.4.7: Courtyard line bug.

To fix this, you have two choices:

1. Delete the footprint's courtyard (my choice).
2. Redraw the courtyard to ensure it forms a closed polygon.

I chose option one because it is easier and faster. I am mindful that the DRC cannot use the courtyard to inform me of overlap violations. But since I have already placed all footprints on the board, I don't expect any related problems.

You need to use the footprint editor to delete or redraw the courtyard lines.

Edge.Cut errors in footprints
Footprints may contain errors, which is why it is important to recognize and fix them. In this project, the footprint I used for the USB port contains an open polygon drawn on the Edge.Cuts layer (see below).

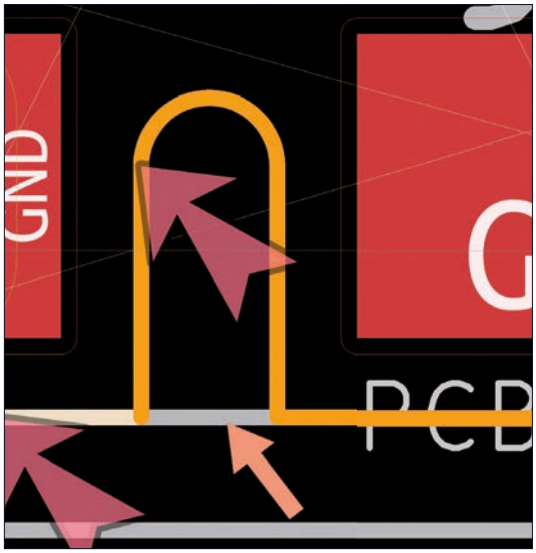


Figure 4.3.4.8: Open polygon graphic in the Edge.Cuts.

The 3D viewer will show an error if it detects an open polygon in the Edge.Cuts layer. Your PCB manufacturer may also have trouble dealing with this and may ask you to fix the layout and re-submit the Gerbers. You must edit the footprint in the footprint editor to fix such errors. I chose to delete the original drawings from the USB connector's footprint.

Run another DRC to ensure no outstanding violations relating to the Edge.Cuts, courtyard, or clearance issues. Below, you can see my board's refined outline:

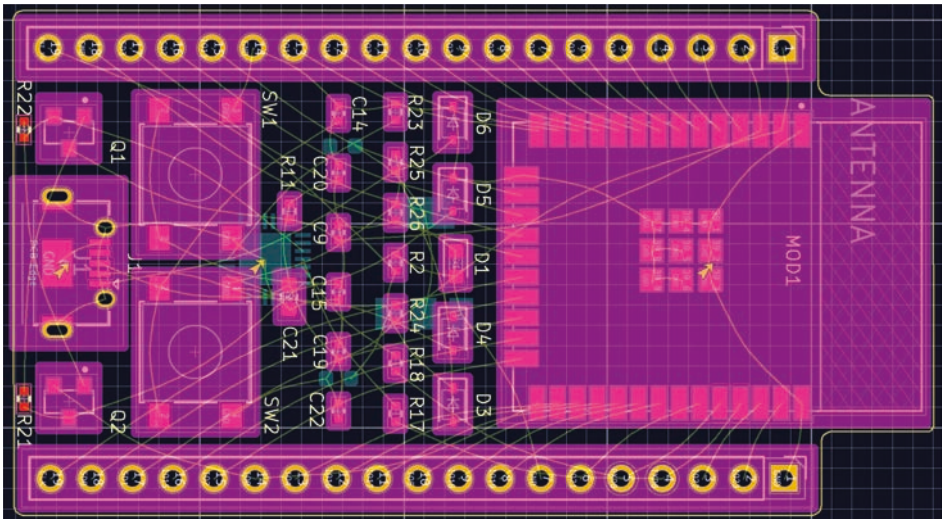


Figure 4.3.4.9: Board refined outline.

And, the board in 3D:

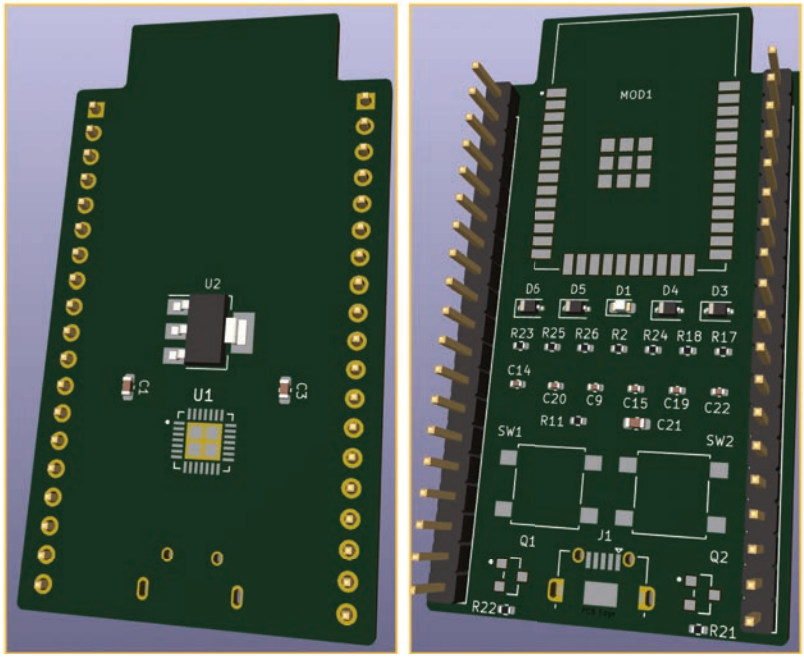


Figure 4.3.4.10: The refined board in 3D.

Let's continue with the routing step.

4.3.5. Layout 4 - Route

This board is not trivial. It contains many footprints and pads arranged in a relatively small board. Using four layers will make routing easier; however, it will still require significant effort and time.

After practicing in previous iterations, completing the routing took me over an hour. Before you begin, please ensure you have enough time to complete this step. Follow the general guidelines I listed in the previous project. For the duration of the routing process, I used the interactive router set to "Walk around" mode, "Fix all segments on click" enabled, and "Optimise entire track being dragged" disabled.

Generally, I begin routing tracks between nearby pads, starting from the central component. For example, below, you can see my first three tracks in the front copper layer, connecting the ESP32 GND pads (1, 38, and 39_1 to 39_9) and the adjacent EN pads (MOD1 pad 3 and J2 pad2):

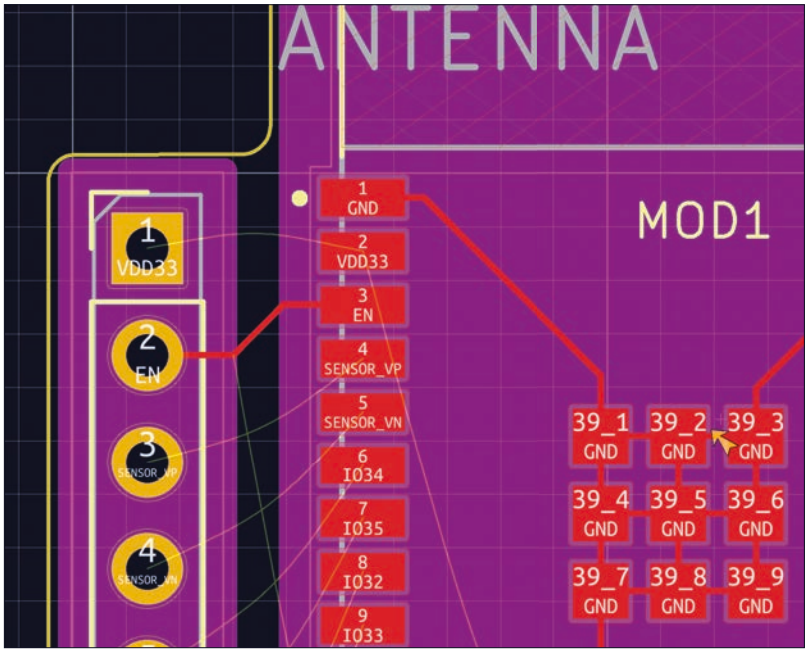


Figure 4.3.5.1: Drawn the first three tracks.

This board contains a differential pair that consists of the SENSOR_VP and SENSOR_VN nets:

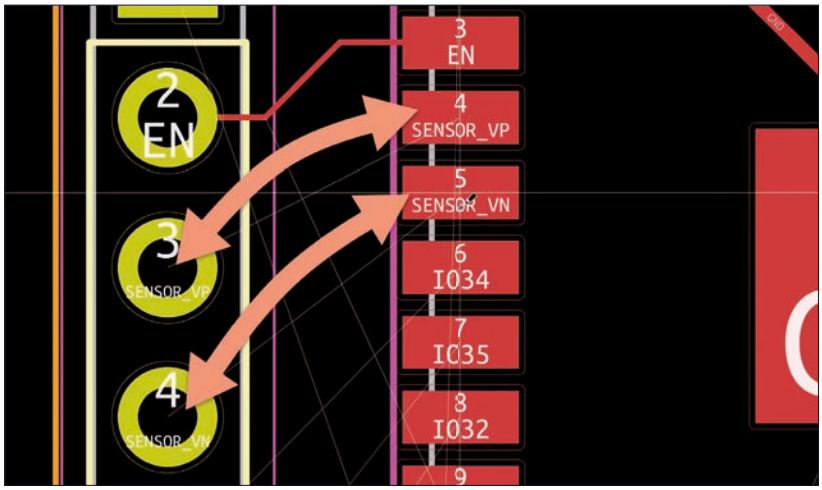


Figure 4.3.5.2: Draw track using differential pair routing.

If you are unfamiliar with the technique of drawing differential pair routes, please read the relevant chapter in the Recipes part of this book. To draw these tracks as differential pairs, select the differential pairs tool from the right toolbar, click on one of the pair pads, like 4 in MOD1, and draw towards one of the differential pair pads in J2, like 3.

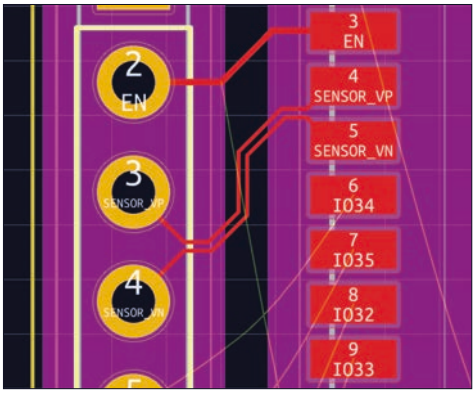


Figure 4.3.5.3: Differential pair routing.

Don't worry if the pair's tracks are imperfect; you can improve them later. Continue with the rest of the pads that belong to MOD1, and then continue with the USB interface and power components. Here are some guideline reminders:

- Try to arrange tracks close to each other to avoid wasted space.
- Avoid 90-degree angles.
- Try to draw tracks in the top layer since the components are surface-mounted. If needed, use vias to access other layers.
- Use the Shift-< hotkey to open the "via" dialogue box and select the layer where you want to continue drawing.

In the example below, I quickly opened the layer selection window to create a new layer. The hotkey for this window is Shift-<:

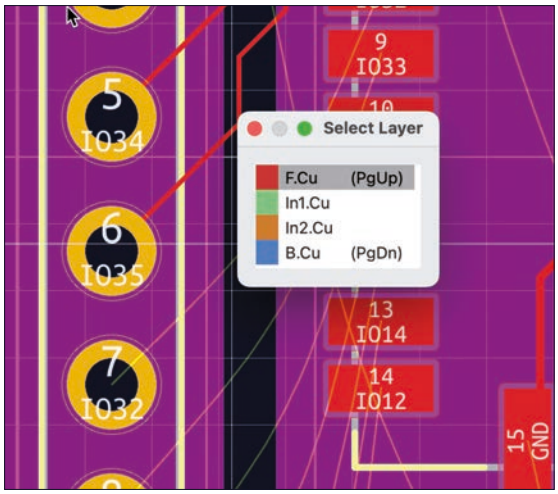


Figure 4.3.5.4: Quickly switching between layers.

To help you with the routing, I provide the following figures that show the routes in each of the four layers:

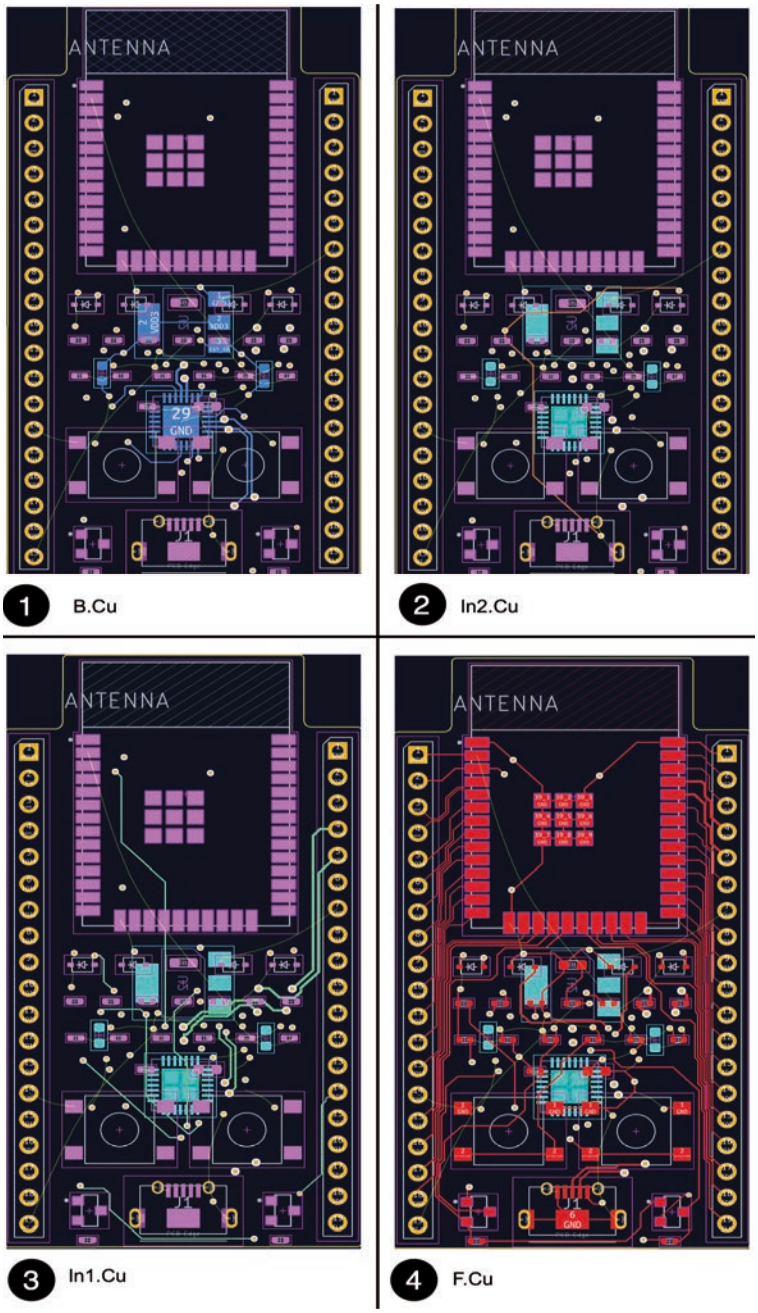


Figure 4.3.5.5: Completed routing, per layer.

Below, you can see my layout at this point. All the routing is complete except for several GND and VDD33 connections, which I will do using copper fills in the next segment of this chapter.

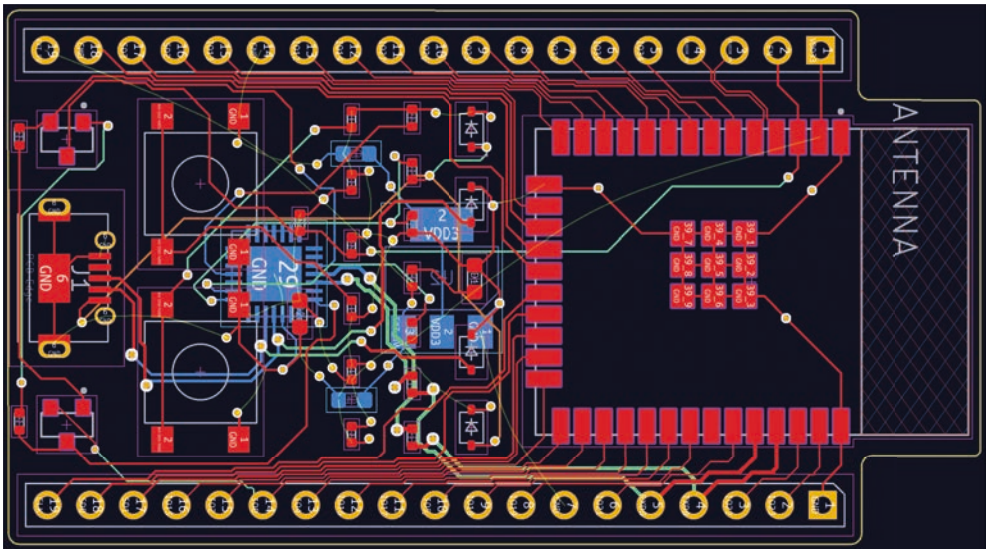


Figure 4.3.5.6: Almost completed routing, pending the copper fills.

Let's finish the routing step in the next segment, where you will draw the copper fills for the GND and VDD33 nets and a keep-out area for the ESP32 module antenna.

4.3.6. Layout 4 - Copper fills and keep out areas

Let's finish work in step four of the layout. In this segment, you will complete two tasks:

1. Create a keep-out area to protect the ESP32 module antenna from accidental placement of other footprints.
2. Connect a copper fill in the bottom copper layer to the GND net. This will also complete any pending GND routes.
3. Connect a copper fill in the top copper layer to the VDD33 net. This will also complete any pending VDD33 routes.

As I was writing this, I realized that I should have done the keep-out area for the ESP32 module antenna earlier in the workflow. Luckily, the footprint I used already has a keep-out area. In the previous edition of this book, I used a footprint that did not have a keep-out area, so I added one. I decided to keep this segment in this edition of the book to give you the opportunity to create one yourself. If you are unfamiliar with keep-out areas (or "keep-out zones"), please review the relevant chapter in the Recipes part of this book.

Select the keep-out zone tool from the right toolbar and click on one of the antenna area corners to begin drawing.

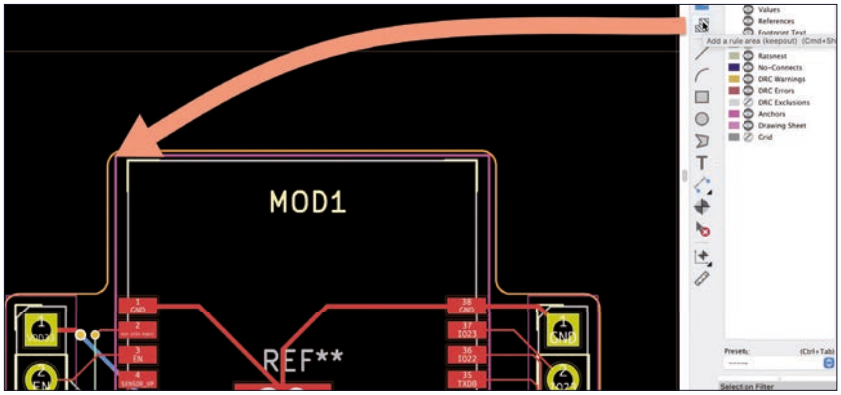


Figure 4.3.6.1: Drawing a keep-out zone.

In the Rule Area Properties window that appears, select all layers (to keep out objects from all selected areas) and give the zone a name:

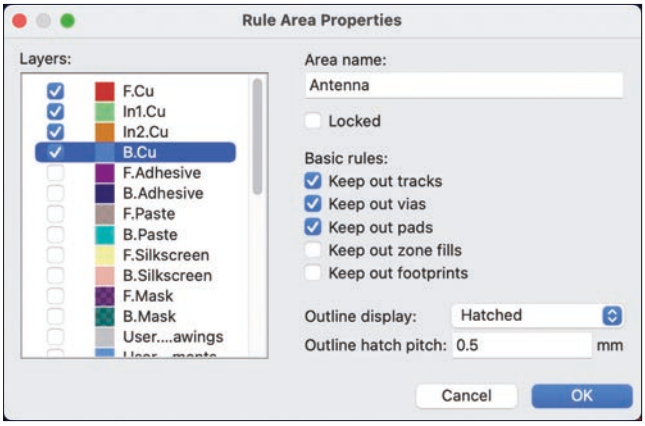


Figure 4.3.6.2: Keep out objects from all layers.

Draw the area so that it looks like this:

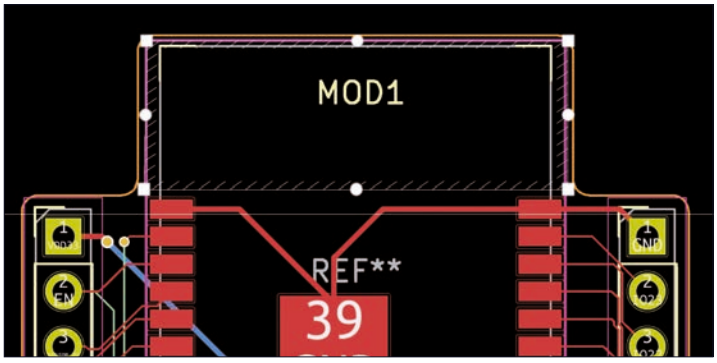


Figure 4.3.6.3: Keep-out zone complete.

Continue to create a copper-filled area in the bottom layer and connect it to the GND net. Here are my settings for this area:

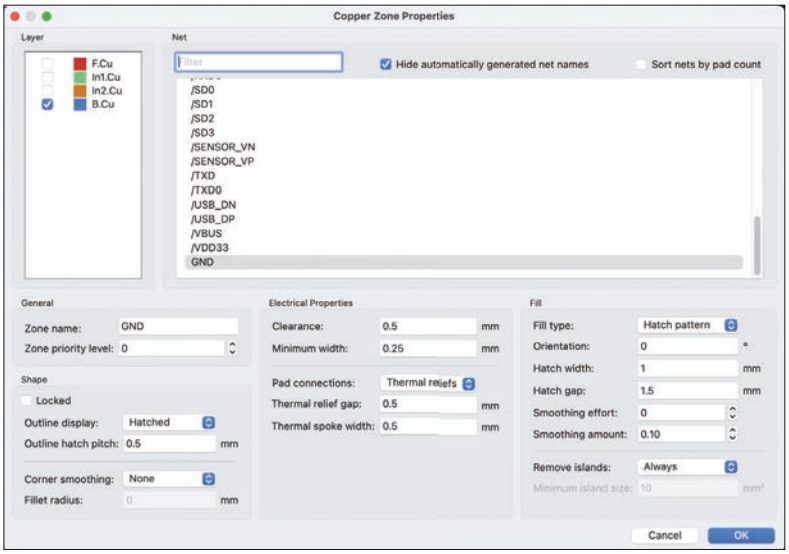


Figure 4.3.6.4: B.Cu copper zone properties.

Finally, create a copper layer in the In2.Cu layer, and connect it to the VDD33 net:

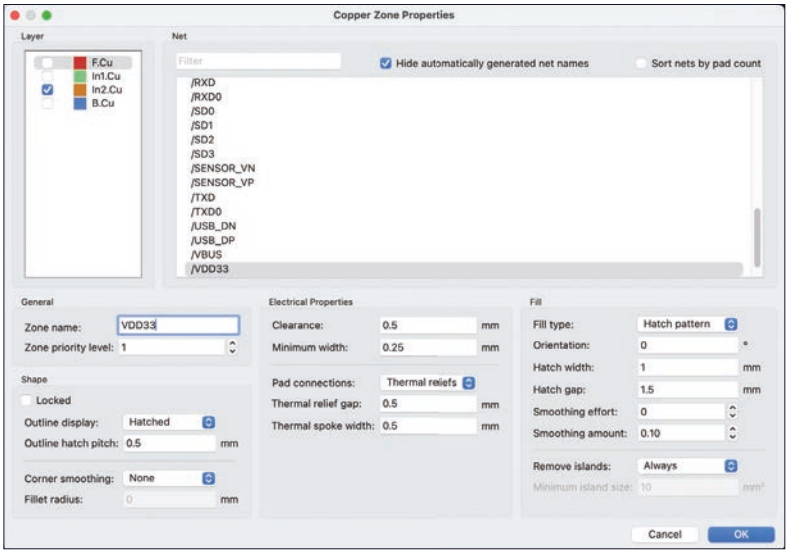


Figure 4.3.6.5: In2.Cu copper zone properties.

When you finish drawing the copper fills, fill them and do a DRC. In my instance, the DRC found several unconnected GND and VDD33 pads. Here is an example:

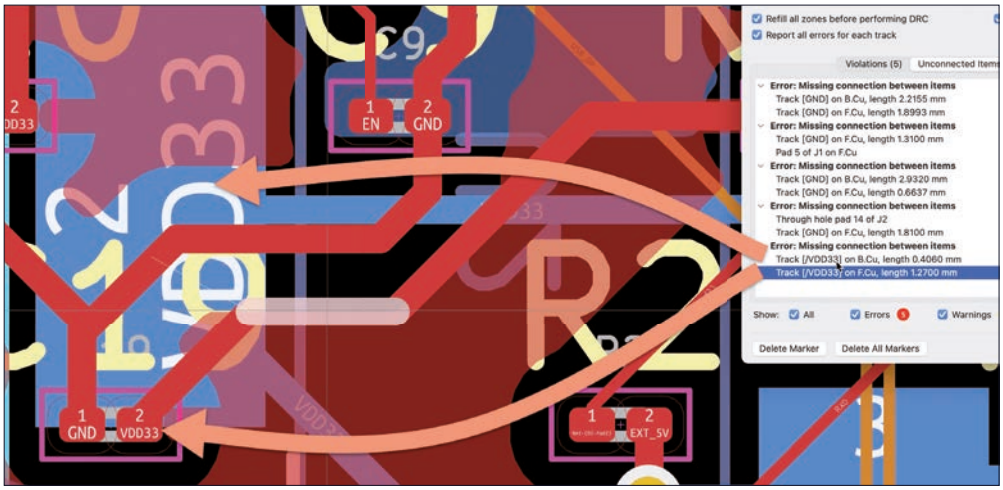


Figure 4.3.6.6: An unconnected VDD33 pad.

The two VDD33 pads, one in the bottom layer and one in the top are unconnected after the copper pours. Similar instances of other VDD33 and GND pads have the same problem.

To fix such bugs, you will need to draw tracks and vias. Use the list of violations in the DRC window to guide you, and disable the copper fills for better visibility.

The violation in figure 4.3.6.6 (above) is easy to fix with a via and two copper track segments:

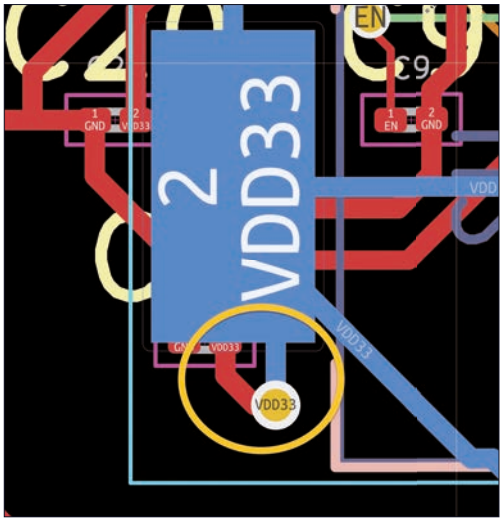


Figure 4.3.6.7: Fixed an "unconnected pad" violation.

Continue in the same manner to fix the rest of the violations. Here is my completed layout, fully routed and filled (I have made the copper fils invisible to improve readability):

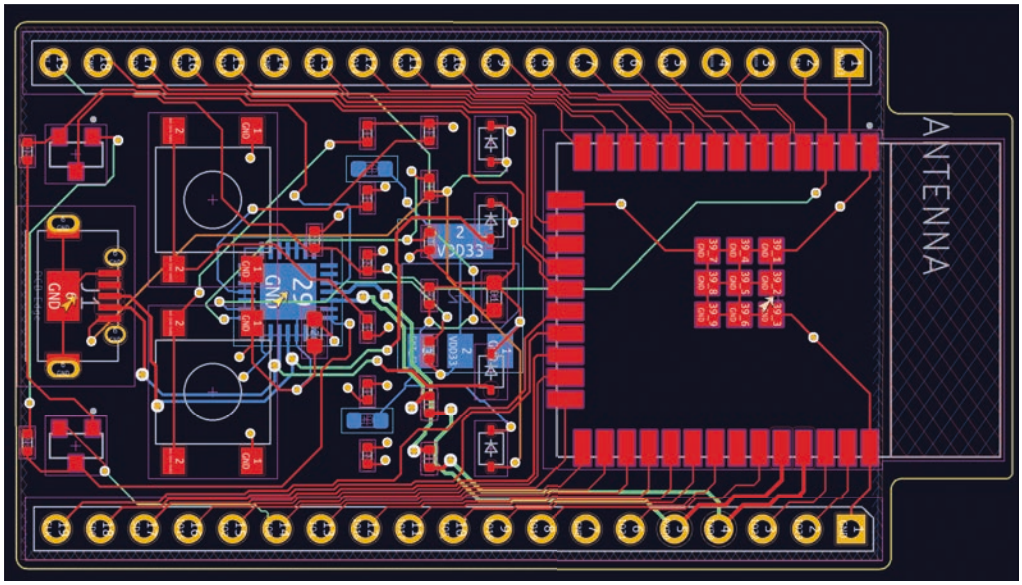


Figure 4.3.6.8: The project PCB, fully routed.

Step four of the workflow is now complete. Let's continue with the silkscreen.

4.3.7. Layout 5 - Silkscreen

In this workflow step, you will work on text and graphical elements on the top and bottom sides of the board. Here is a list of things to do:

1. Optimise the size and location of text labels, like "D6" and "R18".
2. Add text labels for the pin headers.
3. Add text labels for the buttons.
4. Add logos on the back of the PCB.
5. Add board information in the back layer, such as the name and version.
6. Fix silkscreen-related violations reported by the DRC.

Begin by reducing the size of footprint reference text labels. From the Edit menu, choose "Edit Text and Graphic Properties." Select "Reference designators" in the Scope, and set 0.1 mm for the text thickness and 0.8 mm for the text width and height. Click OK, and confirm that the new text size for the values better matches the board size.

One by one, reposition those text items to be near their corresponding footprint so they don't overlap. To make this work easier, use the Edit Text and Graphic Properties window to make all items in the F.Fab layer invisible.

Below, you can see the text in the front silkscreen.

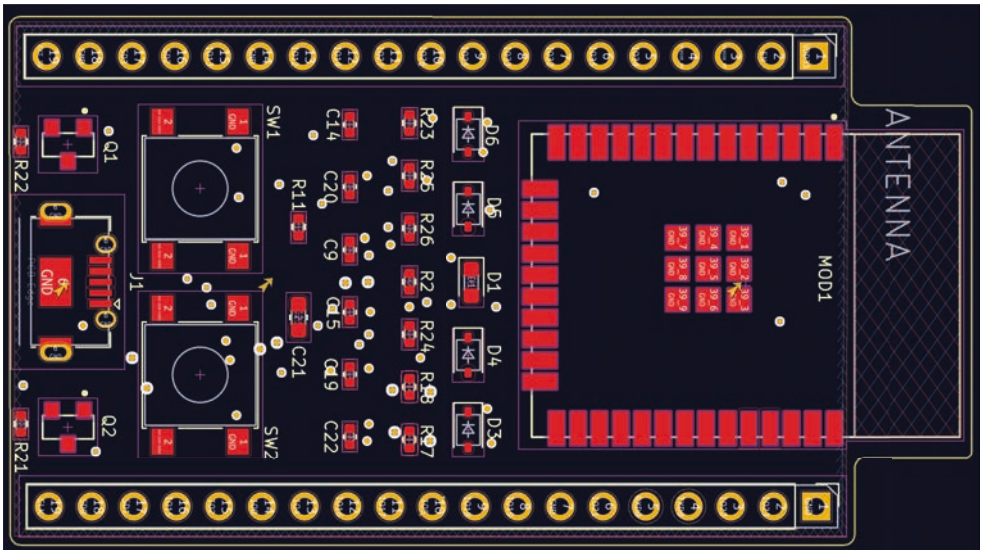


Figure 4.3.7.1: The contents for the front silkscreen layer.

Here are the contents of the bottom silkscreen layer:

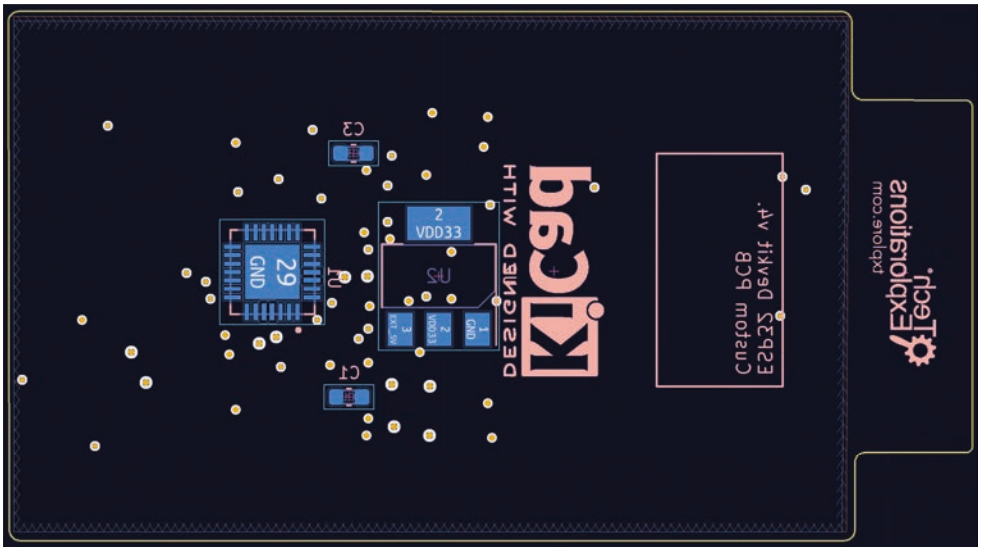


Figure 4.3.7.2: The contents for the back silkscreen layer.

The contents of the silkscreen, back, and front, as you see above, are sufficient. Before the final DRC, I want to make another round of improvements to the routing and fix a few minor issues that I noticed during the silkscreen work.

4.3.8. Layout 4 - Routing improvements

This project is close to completion. I want to make a few final improvements to the copper tracks in this segment. For example, I noticed that the gap between the two tracks in one of the inner layers is too wide:

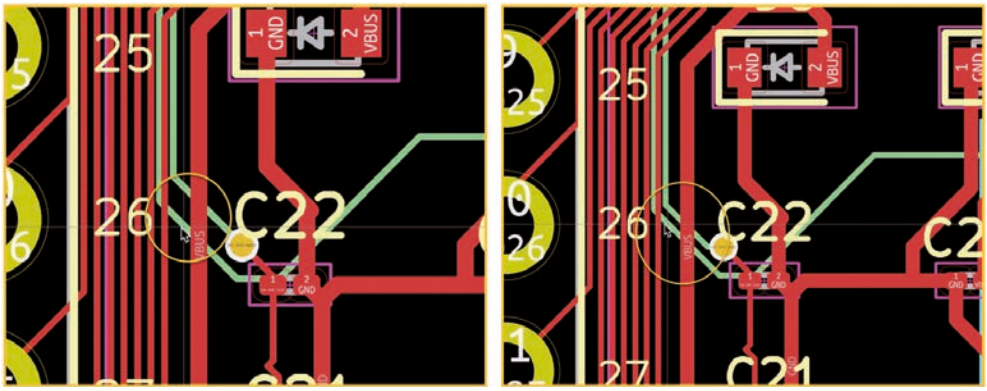


Figure 4.3.8.1: Minimizing gaps.

Large gaps can waste valuable space on a PCB, and this waste can be significant when you need to add elements such as a via to complete a connection. Here is another example:

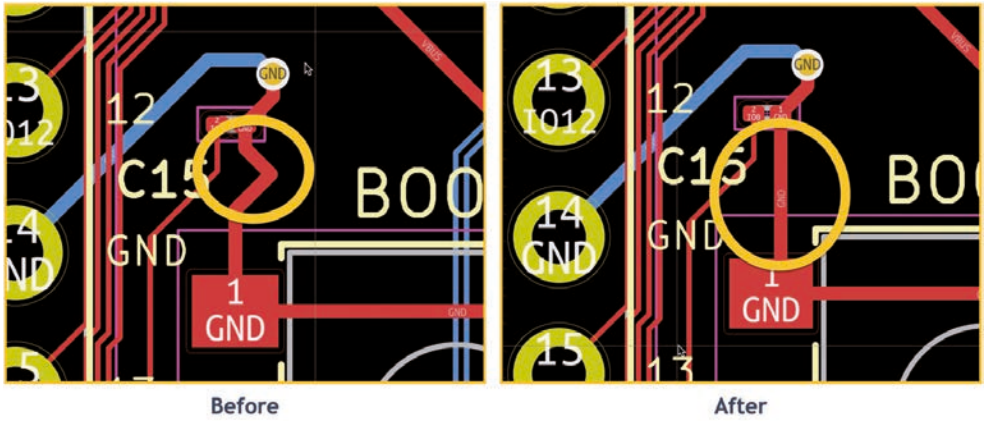


Figure 4.3.8.2: Opt for straight tracks.

The track on the left of the figure above had three bends, even though it connected two nearby pads. Instead, use a straight track like the one in the right example. A final example, where I have replaced a 90-degree corner with 45-degree ones:

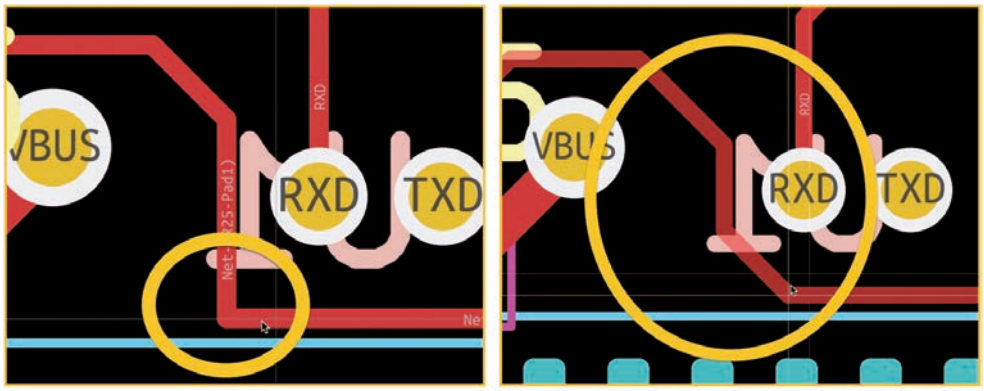


Figure 4.3.8.3: Replace 90-degree corners with 45-degree ones.

Take your time to look for other improvements. Use the Appearance widgets to enable/disable tracks in the different layers to make your work more comfortable by reducing the clutter on the screen. When you are satisfied with the quality of the routing, continue to the last DRC in the next step.

4.3.9. Layout 6 - Design Rules Check

Time for the final design rules check. My check shows two violations and three schematic parity warnings. one violation and twenty schematic parity warnings. The violation relates to the edits I made to footprints J1 and MOD1. You can see them below:

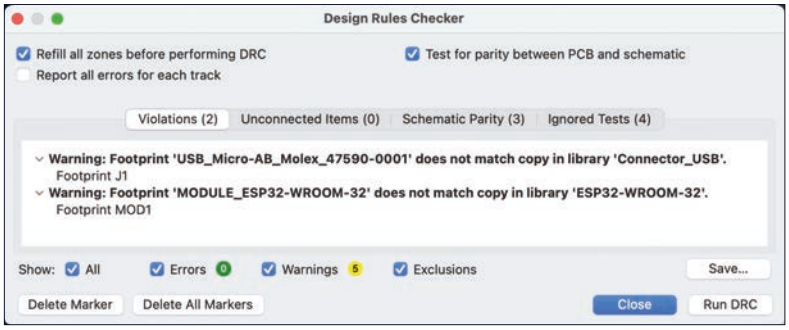


Figure 4.3.9.1: Footprint violations.

These warnings are safe to ignore as I understand their root cause, and I know they do not pose a risk to my design. The other three parity warnings relate to the two logos (which do not have a matching symbol) and a mismatch between U1's used and given footprint.

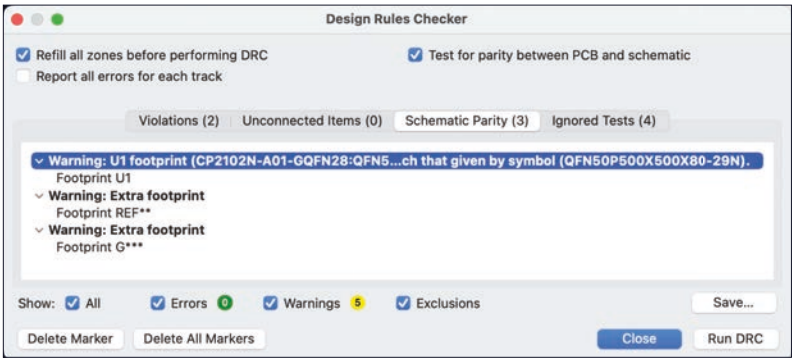


Figure 4.3.9.2: Schematic parity warnings.

I can ignore the two first warnings, but the last one is strange. I have double-checked the assigned footprint for U1 and can see that it is correctly used in the PCB editor. I suspect this warning is erroneous, perhaps due to a bug in KiCad. I will note it for now and continue my work on the project. I will revisit this issue later and probably file a bug report for the KiCad developers.

4.3.10. Layout 7 - Manufacture

The layout is complete and tested. In this segment, you will export the Gerber files, check them for defects, and upload them to the manufacturer's website.

The process is similar to the previous projects, but one important difference exists. In this project, the clearances are smaller than those in previous projects. You must be careful to select the appropriate clearances in the manufacturer's order form that are at least equal to or smaller than the clearances in the layout.

Please beware: The Gerber settings that I show in this page worked at the time I placed this specific order. Manufacturers may change their Gerber specifications from time to time. For this reason, you should check the manufacturer's website for the latest Gerber specifications. For example, here are pages that contain manufacturing information from two manufacturers:

- Oshpark: https://docs.oshpark.com/design-tools/kicad/
- NextPCB: https://www.nextpcb.com/helpcenter/files-explanation

Below, I provide my settings for the Gerber file export:

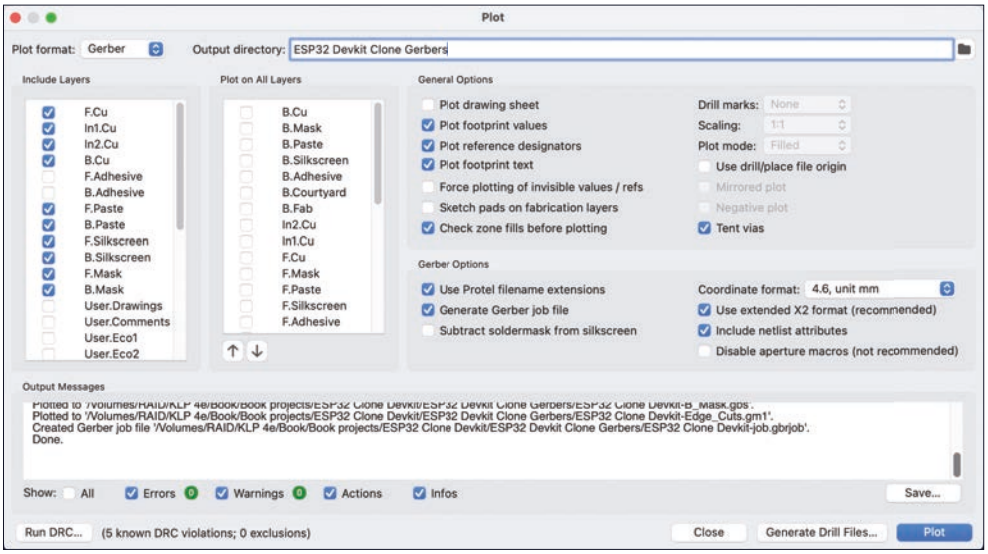


Figure 4.3.10.1: Gerber file export settings.

And the drill files export settings:

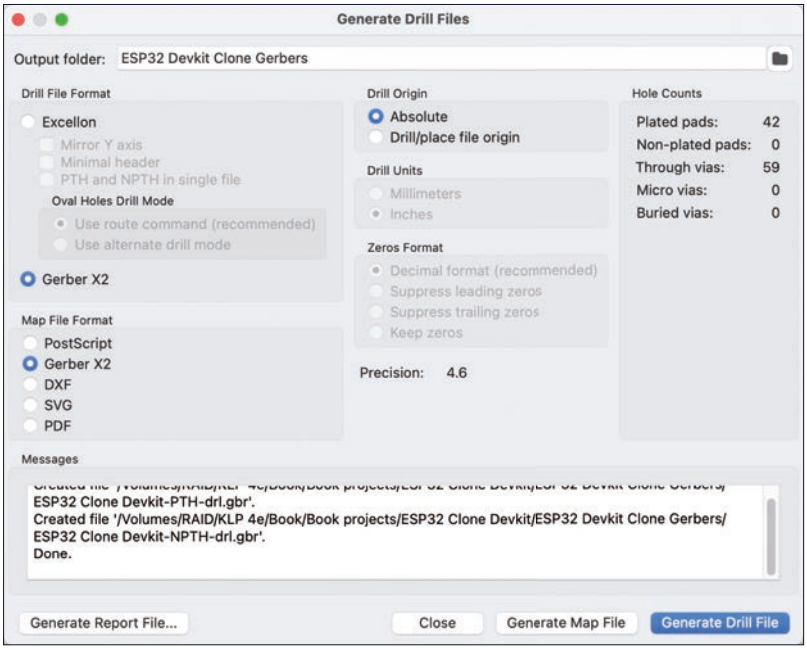


Figure 4.3.10.2: Gerber file export settings for the drills.

Generate the measurements for the board so that you can provide them to the manufacturer's order form (NextPCB will do this automatically):

Figure 4.3.10.3: Board size and characteristics.

To add the board characteristics to the editor, choose "Add Board Characteristics" under the Place menu and select one of the User layers. Use the Gerber viewer to validate the Gerber files:

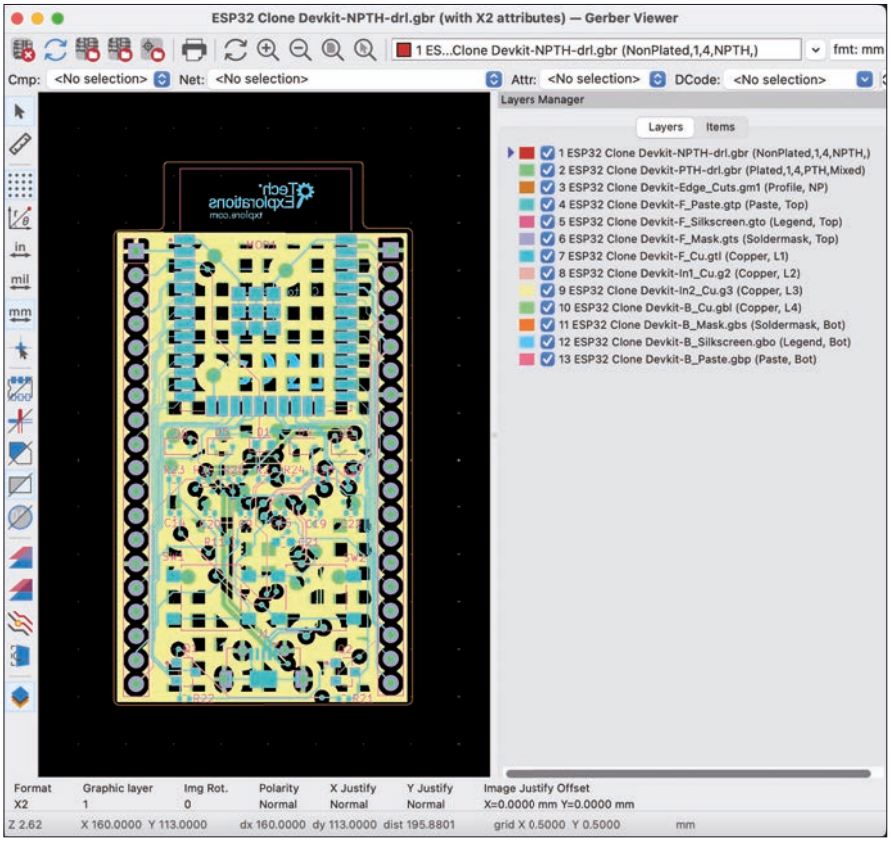


Figure 4.3.10.4: Gerber Viewer showing all layers.

The Gerber Viewer shows no issues of concern; now, I can proceed with the order. I will use NextPCB's KiCad plugin to start the order from inside the KiCad environment. See the screenshot below.

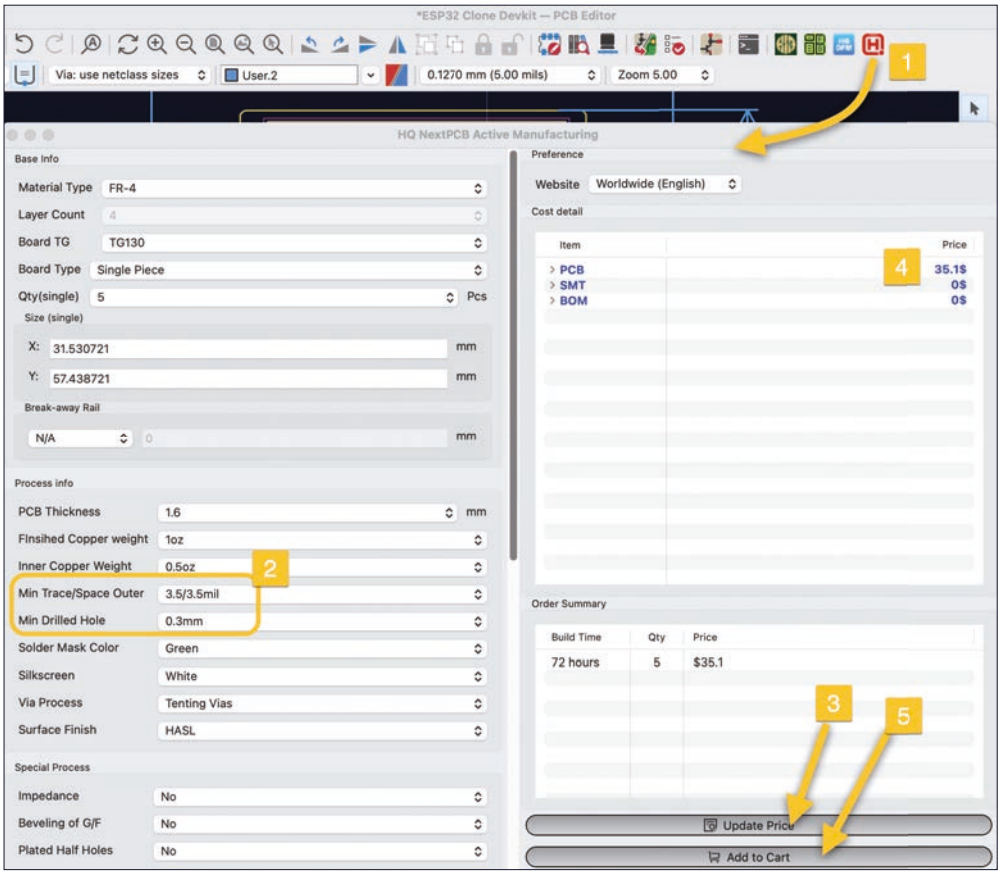


Figure 4.3.10.5: Project PCB order form.

Open the NextPCB Active Manufacturing plugin window by clicking the "H" button on the top toolbar ("1"). In the Process Info area, I have selected the smallest available minimum trace and minimum drill sizes ("2") so that the minimums of my board are properly considered for the pricing. Click the Update Price button ("3") to get the calculated price, and after checking the price ("4"), click "Add to Cart" ("5"). This will send your order to NextPCB. Open the order form in your browser.

Before completing the order, I will also use the NextPCB online Gerber viewer to check the design for in-depth manufacturing. To do this, go to https://www.nextpcb.com/free-online-gerber-viewer.html and upload the project Gerber files in a ZIP archive. The analysis tool does reveal a few interesting issues that I must evaluate before continuing with the order.

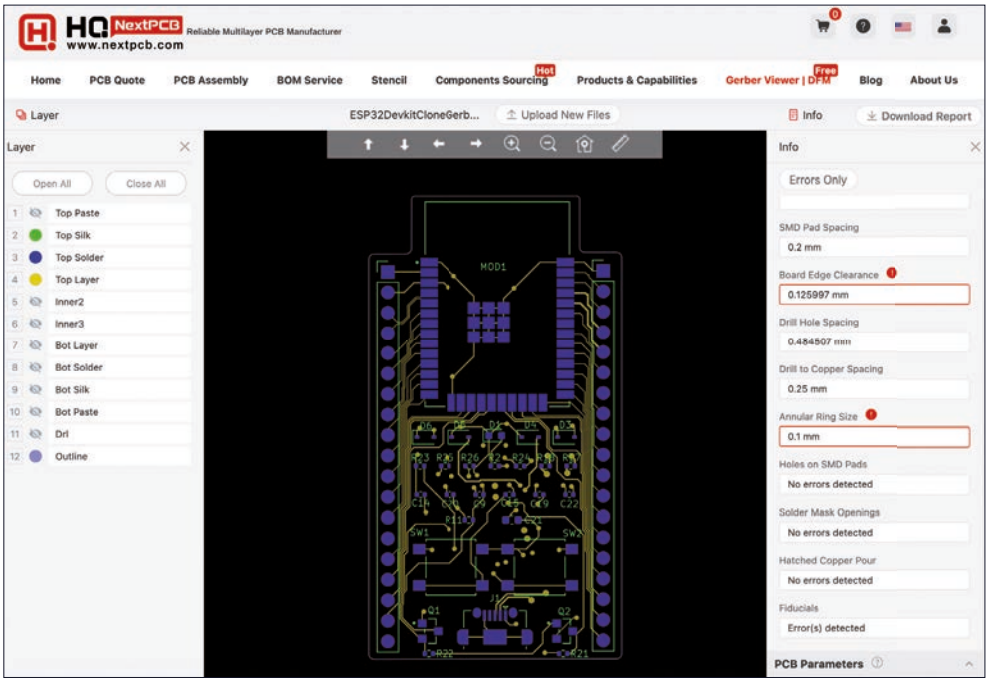


Figure 4.3.10.6: NextPCB's DFM tool has revealed several issues that require further consideration.

The summary on the right side of the page shows issues relating to the board edge (potentially insufficient clearance), the size of one or more annular rings (too small), and the drill size of at least one via (also potentially too small). The full report contains more information that can help me locate the issues. The 7-page PDF provides details for all the detected issues. Below, I provide a few examples.

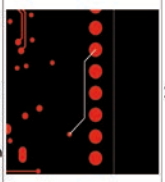
2	Smallest Trace Width_Smallest Trace Width	3.5,4,5	0.10 mm	Traces 3.94mil in width were detected in your design. This could result in overetched traces, which decrease manufacturing efficiency and yield, and affect the reliability of the boards. It is recommended to increase the width to at least 6 mil for regular routing and at least 4 mil in high density areas, such as when routing fine pitch BGAs.		141.22,-115.82	1	Warn

Figure 4.3.10.7: DFM example 1: trace width issue.

3	Smallest Trace Spacing_Trace Spacing	3.5,4,5	0.10 mm	Trace-to-trace spacing of 3.94mil were detected in your design. This could result in incomplete etching between the traces, thereby producing shorts, which decrease manufacturing efficiency and yield, and affect the reliability of the boards. It is recommended to increase the spacing to at least 6 mil for regular routing and at least 4 mil in high density areas, such as when routing fine pitch BGAs.		128.53,-103.12	1	Warn

Figure 4.3.10.8: DFM example 2: trace spacing issue.

8	Signal Integrity_Acute Angle Traces	-,-,-	Error(s) detected	Acute angle traces were detected in your design. The sharp change in direction could cause signal reflections, resulting in transmission line and signal integrity issues. Consider increasing the angle or rounding the traces.		141.03,-78.49	1	Warn

Figure 4.3.10.9: DFM example 3: signal integrity issue.

I decided to address all issues before proceeding with the order. This involves these steps:

1. For each issue identified in the DFM report, use the PCB editor to locate it and consider whether a fix is required or can be ignored. If a fix is required, perform it and continue with the next item in the list.
2. When all issues have been fixed, export the Gerber files and conduct a new DFM analysis.
3. When all issues have been fixed, proceed with the order.

Let's continue with the 3D shapes in the next segment.

4.4. 3D shapes

In this chapter, I will add several missing 3D models so that the 3D viewer can synthesize a complete 3D rendering of the board.

At the end of the layout workflow, my board looks like this in 3D:

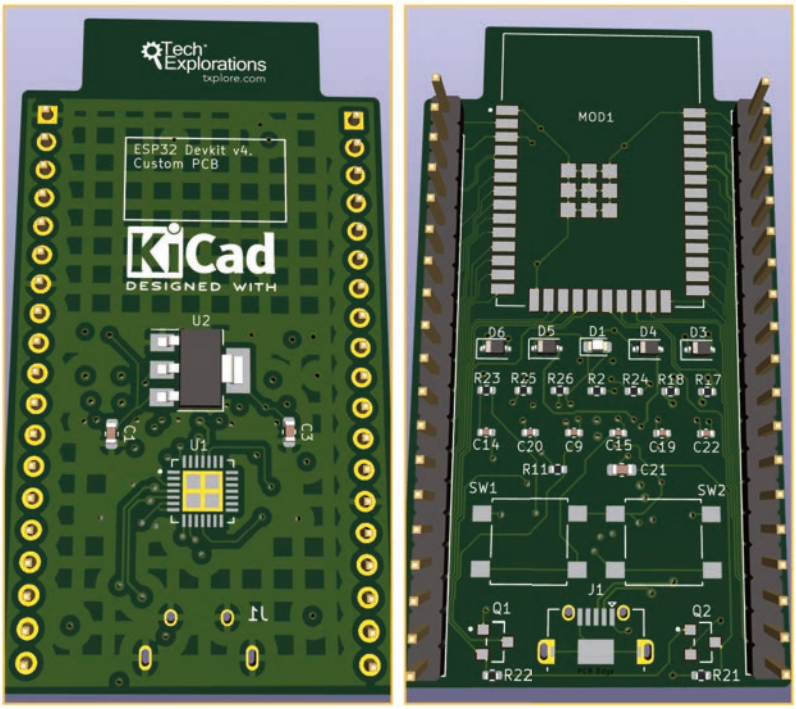


Figure 4.4.1: 3D rendering of my board.

As you can see above, the layout contains 3D models for the voltage regulator in the back and the headers, capacitors, LEDs and resistors in the front. Let's go through the footprints that don't have a 3D model. Here are the sources of the 3D models I used:

- ESP32 module, from Snapeda[10]: ESP32-WROOM-32.step
- CP2102N chip, from Snapeda[11]: CP2102N-A01-GQFN28.step
- Buttons, use one from the KiCad 3D models library: ${KICAD8_3DMODEL_DIR}/Button_Switch_SMD.3dshapes/SW_SPST_PTS645.step
- USB connector, from KiCad's library: ${KICAD8_3DMODEL_DIR}/Connector_USB.3dshapes/USB_Micro-B_Molex_47346-0001.step
- Transistors from KiCad's library: ${KICAD8_3DMODEL_DIR}/Package_TO_SOT_SMD.3dshapes/SOT-23.step

Download, install, and set up these 3D models in your layout. Below, you can see my board in 3D, with all 3D models included:

10 https://www.snapeda.com/parts/ESP32-WROOM-32/Espressif%20Systems/view-part/?ref=search&t=ESP32-WROOM-32
11 https://www.snapeda.com/parts/CP2102N-A01-GQFN28/Silicon%20Labs/view-part/?ref=search&t=CP2102N-A01-GQFN28

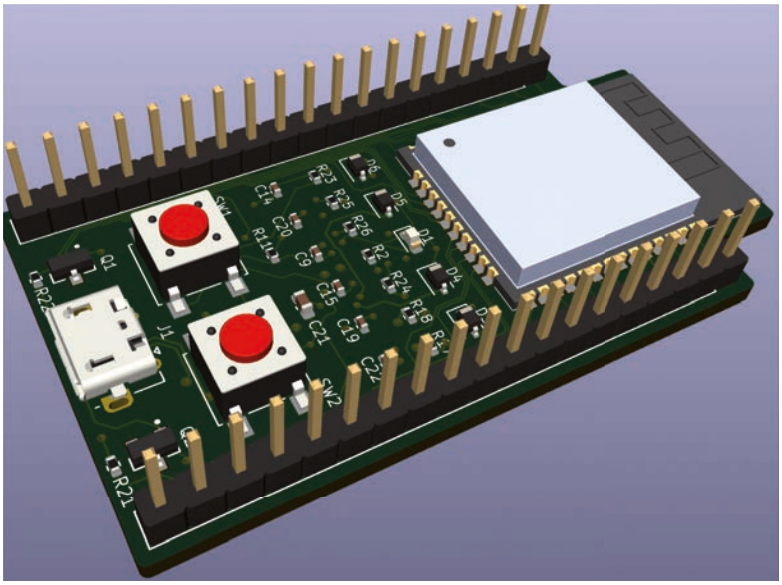


Figure 4.4.2: 3D rendering of my PCB, front, all models included.

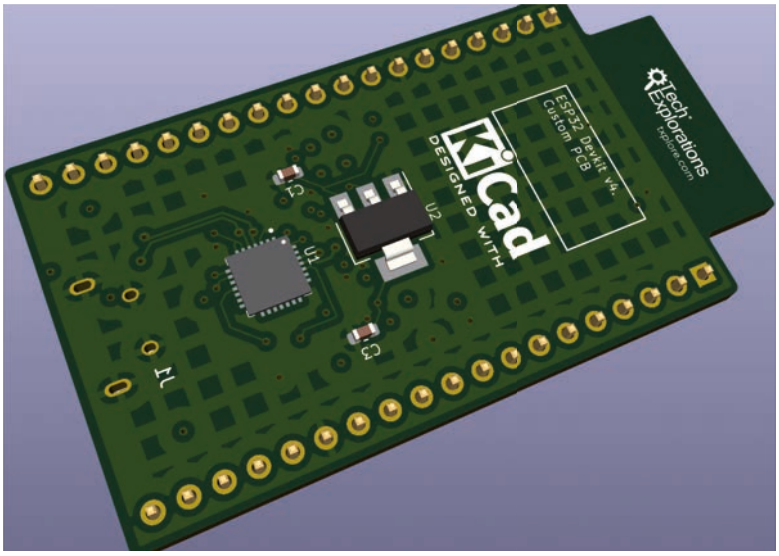


Figure 4.4.3: 3D rendering of my PCB, back, all models included.

Congratulations on completing this project!

4.5. Finding and correcting a design defect

As part of this book's 3rd edition beta program, a reader found a defect in the schematic of this project. In this chapter, I will show you how to quickly fix this defect.

Even though I fixed this bug in the 3rd edition of the book, I made the exact same mistake in the fourth edition! So, let's take this opportunity to learn how to fix an error that requires changes to both the schematic and the layout.

The defect

While copying the elements and wiring from the ESP32 devkit reference design to my KiCad schematic, I did not notice that transistor Q2 was flipped. In the reference design, the emitter of Q2 (pin 2) is connected to DTR, as in the image below (this is a segment of the reference design):

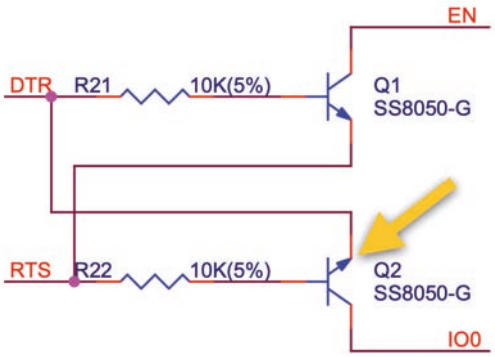


Figure 4.5.1: Segment of the reference design showing the correct wiring for Q2.

When I copied this network to my KiCad schematic, I did not notice the correct orientation of the emitter. The result is that I connected the emitter (pin 2) to IO0 instead of DTR. In the image below, I show the error in the schematic:

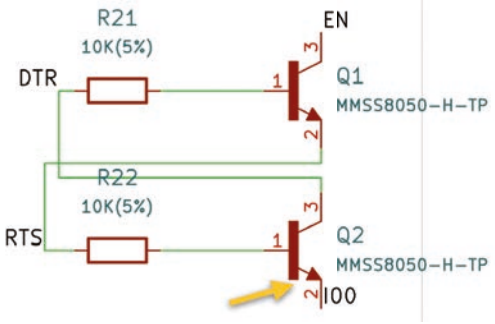


Figure 4.5.2: The arrow shows the error in my schematic instance.
The emitter (pin 2) of Q2 is connected to IO0 instead of DTR.

Fortunately, this defect is easy to fix.

4.5.1. Fix the schematic

First, let's fix the defect in the schematic. I have to re-orient the emitter of Q2 so that it points towards Q1. The easiest way to do this is to use the "Mirror Vertically" option of the Q2 symbol properties (see below).

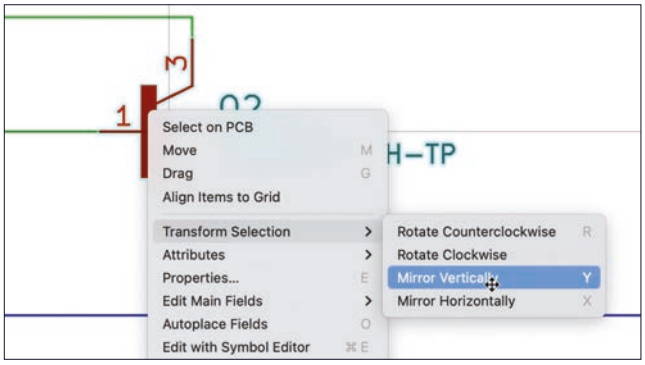


Figure 4.5.1.1: Fix the wiring defect by flipping Q2 vertically.

After the defect is corrected the Q2 and Q1 wiring will look like this:

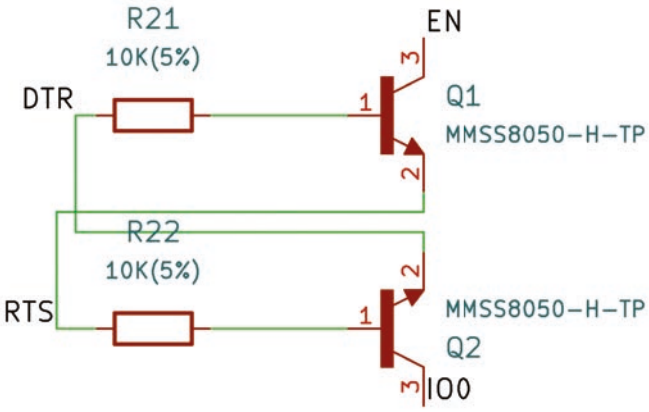


Figure 4.5.1.2: The defect is corrected.

Save the schematic. Let's continue the correction process in the layout editor.

4.5.2. Fix the layout

In the PCB editor, import the changes in the schematic into the layout, and notice a couple of new ratnest lines in Q2 (see below).

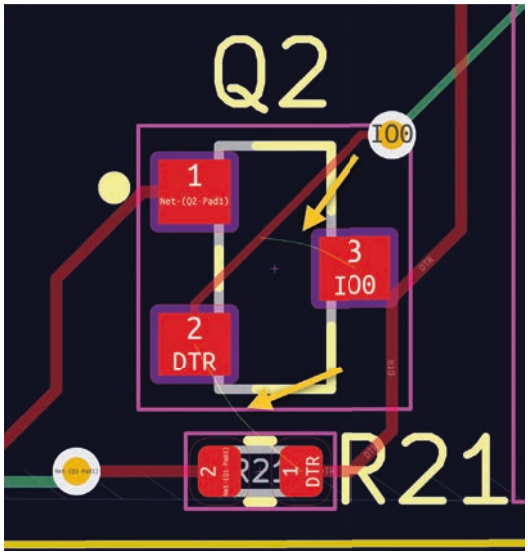


Figure 4.5.2.1: After importing the schematic changes, ratsnest lines appear in Q2.

After the schematic changes were imported, the layout editor did not delete the copper traces connecting Q2 pins 2 and 3 to other parts of the PCB. The only clue that something has changed is the two new ratsnest lines. You must manually delete the original copper traces and draw new ones.

In the image below, I have deleted several obsolete copper traces that initially connected Q2 pins 2 and 3 to the IO0 and DTR nets.

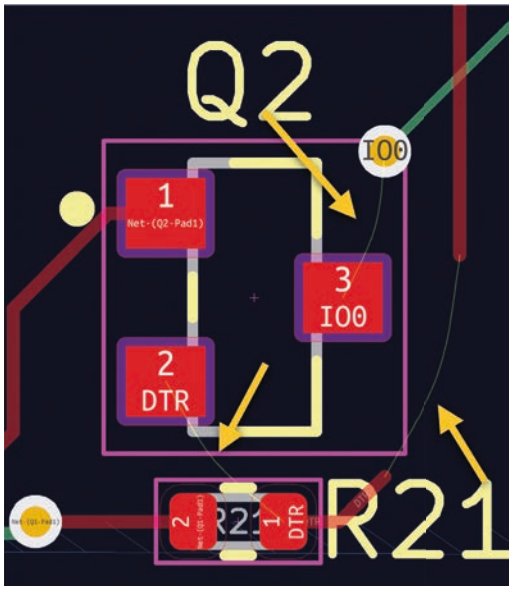


Figure 4.5.2.2: Removed obsolete copper traces.

Finally, re-draw the copper traces. In the figure below, I use arrows to highlight them:

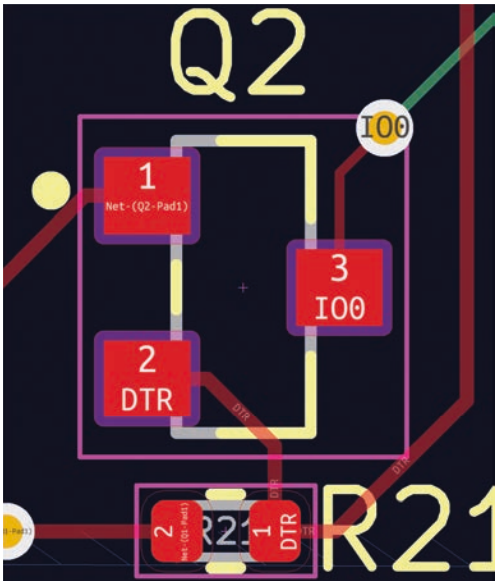


Figure 4.5.2.3: New copper traces from Q2 pins 2 and 3.

With the redraw complete, the original defect is now corrected. Do a DRC to ensure there are no other issues to address. My latest DRC showed no further defects, so I could re-export the Gerber files.

5: Recipes

Welcome to the "Recipes" section of this KiCad guide, which provides step-by-step instructions and tips for enhancing your PCB design and schematic capabilities using KiCad.

Each chapter in this section addresses a specific aspect or tool within KiCad, from bulk modifications of symbols and footprints to advanced routing techniques and project management using Git. Here, you'll find detailed guidance on utilising KiCad's powerful features like the Freerouting autorouter, customising the PCB and Schematic editors, and managing design rules to ensure your projects meet the necessary specifications.

This section also delves into importing and managing components from various sources, leveraging KiCad's integration capabilities to bring in projects from other EDA tools like EAGLE and Altium, and utilising the circuit simulator for validation. Additionally, practical recipes for setting up your board and schematic with predefined configurations will help streamline your design process.

5.1. Interactive delete

Deleting unwanted elements of a schematic or layout always involves risk. Yes, you can undo an incorrect deletion or restore from a backup if things go wrong. But why make a mistake in the first place? In health and medicine, prevention is better than any cure, and the same applies to computer-aided design.

In KiCad, you can delete an element, such as a text item, a symbol, a footprint, or a graphic, by hovering the mouse pointer over the component and pressing the Delete key. You can also right-click on an element to bring up the contextual menu and then select Delete.

The problem with these methods is that it is not always obvious what you are deleting when you are working in busy schematics or layouts.

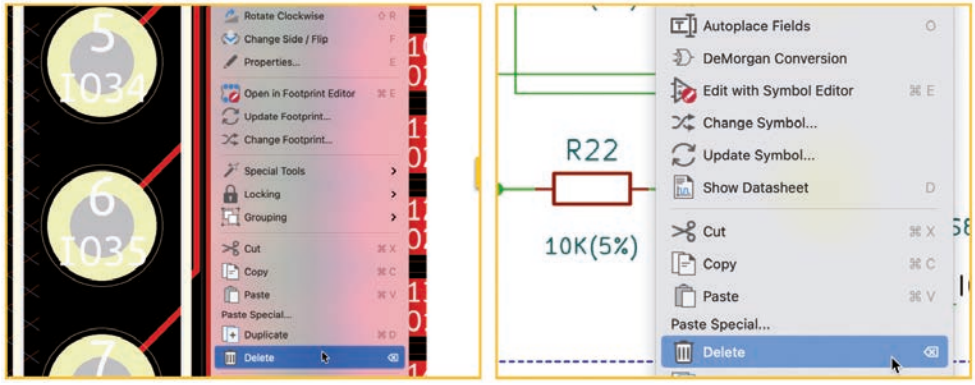


Figure 5.1.1: Deleting something in the PCB editor (left) and the Schematic editor (right) entails some risk.

In Figure 5.1.1 (above), I am using the context menu to delete an unwanted element. In the PCB editor (left), I wanted to delete a track. However, the header footprint was selected instead of the track because of imprecise targeting and the active selection filter settings. You can see that the PCB editor highlighted the header footprint with white instead of regular yellow. In the schematic editor, the item with its context menu activated is marked with a fuzzy halo around it. In both cases, it is easy to make mistakes and delete the wrong thing, especially in busy sheets.

If you use the hover and Delete keypress option, you don't even get an indication of precisely what you are about to delete.

To make it safer to delete schematic or layout elements, KiCad provides the Interactive Delete tool. This tool is available in both the schematic and the PCB editors. You can find it in the Edit menu and at the bottom end of the right toolbar (Figure 5.1.2).

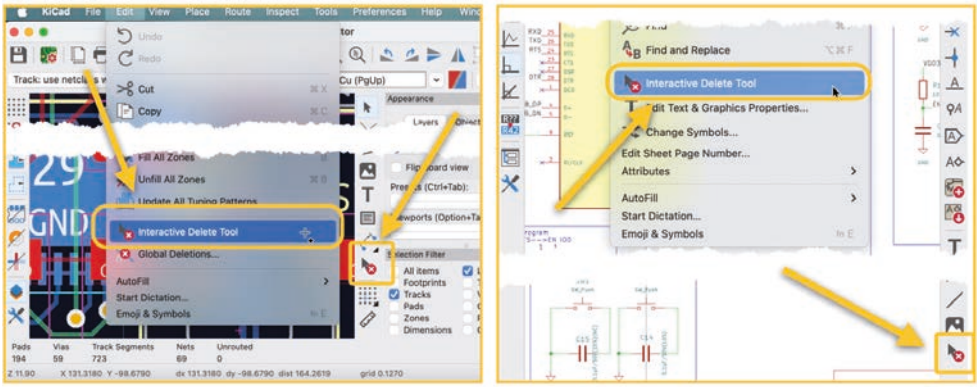


Figure 5.1.2: You can find the Interactive Delete tool in the Edit menu and at the bottom end of the right toolbar. PCB editor (1), Schematic editor (2).

When you select the Interactive Delete tool, the cursor changes to a rubber eraser with a small cross. Anything the cross hovers over is highlighted with a configurable color (you can customize all colors via the Preferences window; then Schematic Editor —> Colors and PCB Editor —> Colors). You can see an example below (Figure 5.1.3).

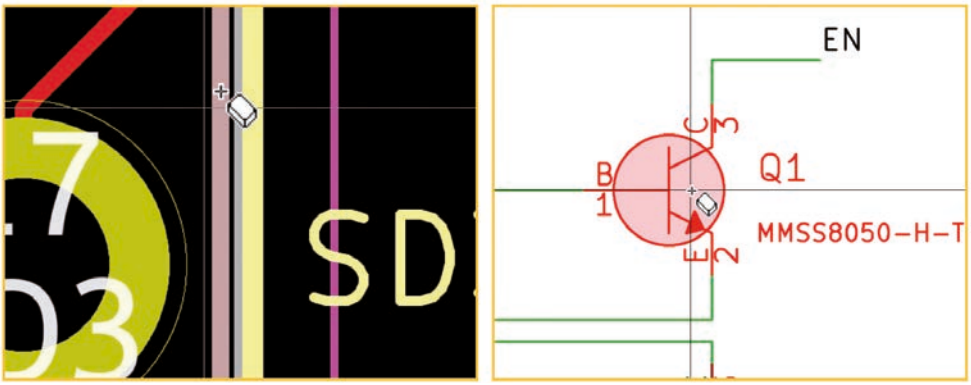


Figure 5.1.3: An example of the Interactive Delete tool.
PCB editor (left), schematic editor (right).

In the layout editor, the Interactive Delete tool works regardless of the selection filter settings, but it does not allow you to delete locked items.

5.2. Find and Replace (Schematic editor)

In Recipe "Change a symbol in bulk" (in my book Fundamentals and Projects), you learned how to quickly change symbols assigned to a component based on common criteria, like the library designator or the value. KiCad offers more bulk-editing tools, and one of them allows you to make bulk operations for text objects in the Schematic Editor. You can access this tool from the Edit menu.

Click on Edit, then click on Find and Replace. Let's look at an example where the Find and Replace tool is useful. I have a schematic that contains several instances of the "RESET" net label. This label exists over two different pages of the schematic. See the images below (Figure 5.2.1).

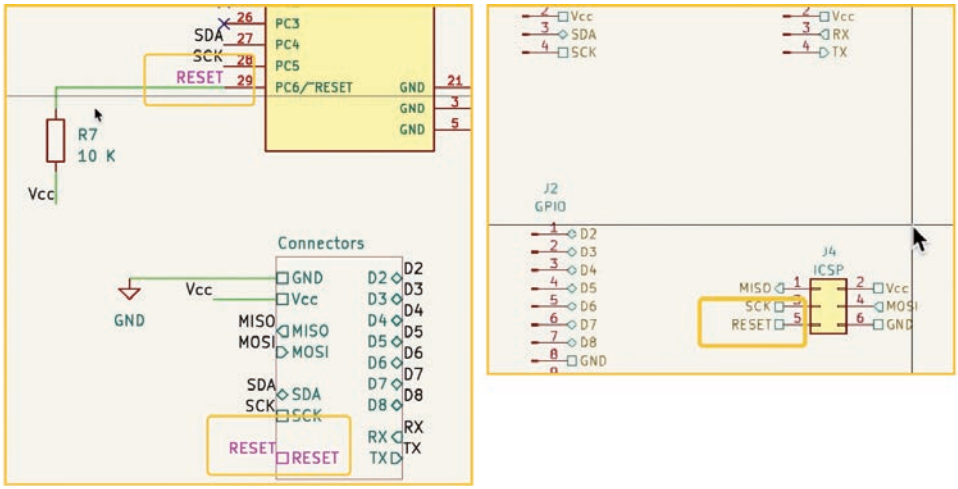


Figure 5.2.1: The net label "RESET." I'll change it to "RST."

Say you want to change "RESET" to "RST" using the Find and Replace tool. Bring up the Find and Replace tool by clicking Edit —> Find and Replace. In the "Search for" field, type "RESET," and in the "Replace with" field, type "RST" (Figure 5.2.2).

Figure 5.2.2: The Find and Replace window is ready to get to work.

Click on "Replace All." The Find and Replace window will not disappear, so you can search and replace more text if you want. You can still pan and zoom in on the schematic to confirm the changes. Click the Close button to dismiss the Find and Replace window.

In the schematic, you can now see "RST" in all net labels that previously contained "RESET" (Figure 5.2.3).

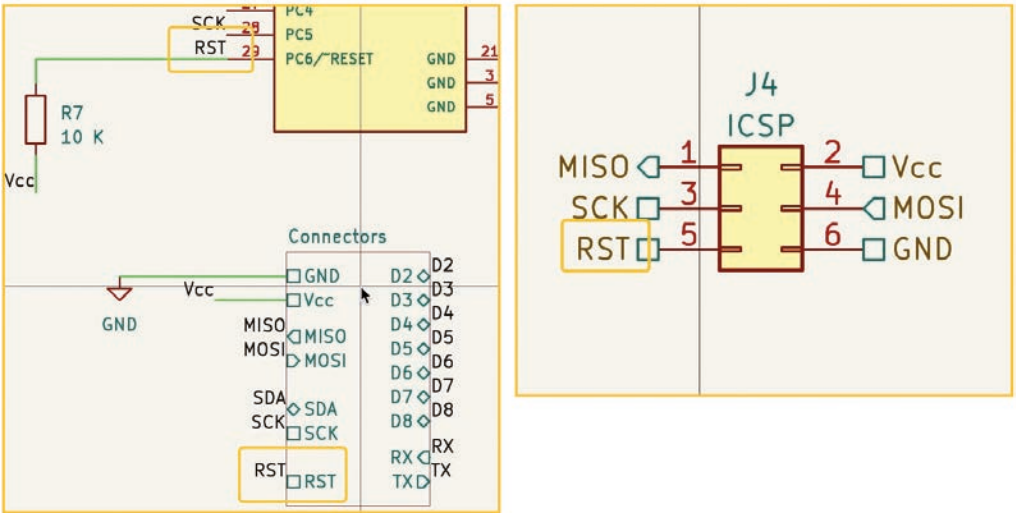


Figure 5.2.3: The net label "RESET" has been changed to "RST."

You can use this tool to find and change any text type, whether a net label or a text item, as long as it appears within the schematic. Text appearing in the schematic page settings area (in the bottom right corner) is unaffected.

5.3. Text variables

A new feature that was introduced in KiCad 6 is Text Variables. With Text Variables, you can create variables that contain any text string, which you can use in a schematic or layout to parametrize anything "text." For example, you can set a Text Variable with a version number. In your schematic and PCB editor designs, you can reference this variable so that it always shows its current value. As you work through new versions of your PCB, you can update the version number at a single location (the text variables window) instead of looking for the various locations in the schematic and layout editors when the version number may appear.

Let's look at an example. Start by creating a Text Variable. You can do this in the Schematic Setup dialogue box in the schematic editor and the Board Setup in the PCB editor. They are in different groups, but they are the same thing. When you create a Text Variable in the schematic editor, it will also appear in the PCB editor and vice versa.

In the schematic editor, click on File and Schematic Setup. Under Project, click on Text Variables (Figure 5.3.1). At the bottom of the Text Variables list, click the "+" button to create a new row (1). For the variable name, enter "project_name" (2), and for the text substitution, enter "Breadboard power supply" (3). When ready, click on OK to finish (4).

Figure 5.3.1: A new Text Variable.

Try the same in the PCB editor. Click on File, then Board Setup. Under "Text & Graphics," you'll see "Text Variables." Click on that to show the list of variables. Notice that the variable you created in the schematic editor also appears here (Figure 5.3.2).

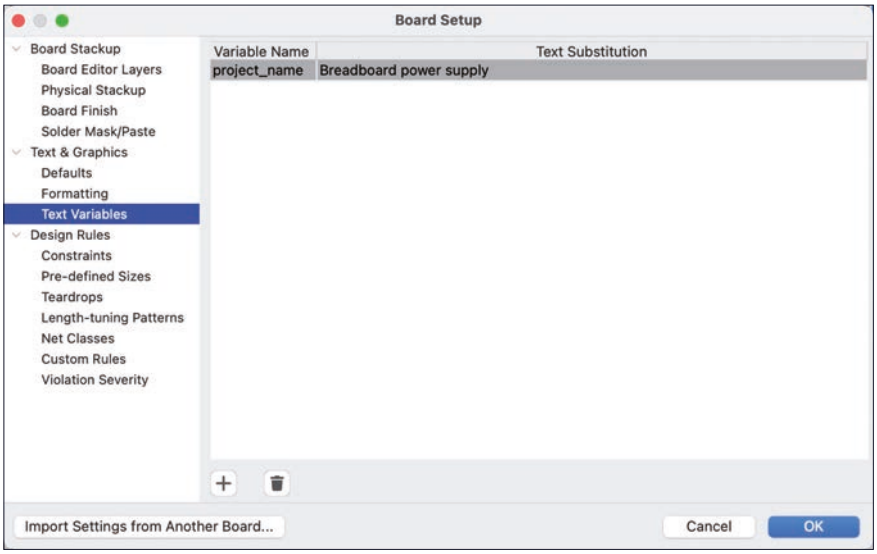


Figure 5.3.2: The new text variable created in the schematic editor also appears in the PCB editor.

Next, create a few references to the new variable using its name. For example, let's use this variable in the schematic editor Page Settings, which appears in the editor sheet's bottom right corner. Bring up the Page Setting window (File —> Page Settings), and enter the variable reference "${project_name}" (without the double quotes) in the Title field (Figure 5.3.3):

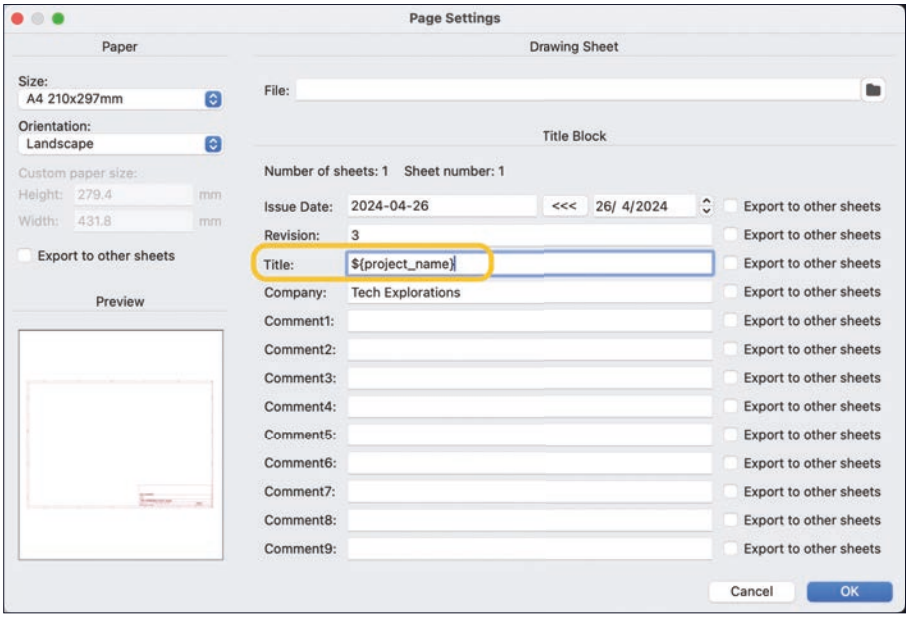


Figure 5.3.3: The variable reference used in the Page Settings.

Click OK to exit the Page Settings window. Pan the editor to the bottom right corner so you can see the Sheet title. It should contain the value of the "project_name" text variable (Figure 5.3.4):

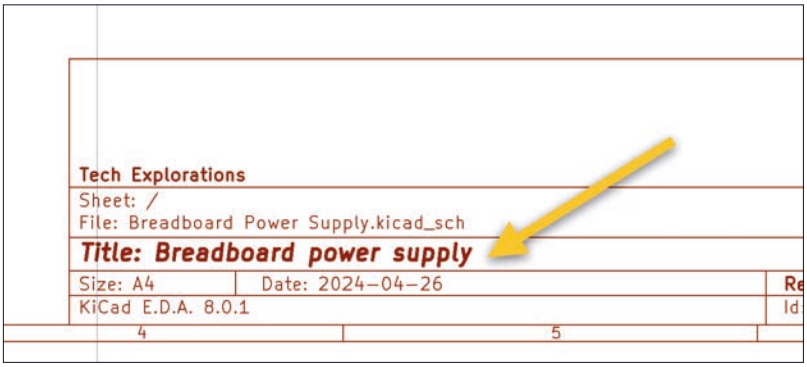


Figure 5.3.4: The variable reference is substituted for its value.

As you can see, the variable reference was substituted by its value.

Let's use the same text variable in the PCB editor. Go to the PCB editor and click File —> Page Settings. In the Title field, type the text variable reference "${project_name}" (without the double quotes) (Figure 5.3.5).

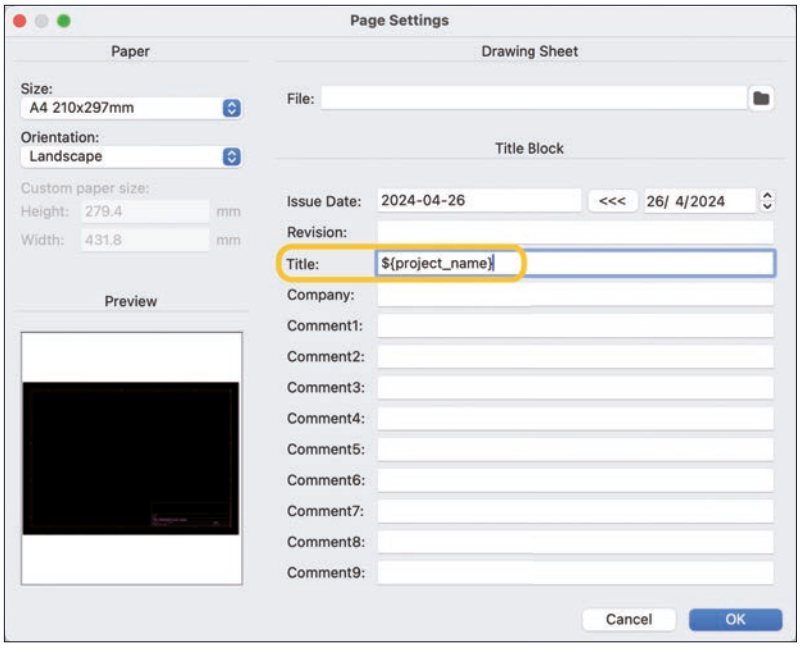


Figure 5.3.5: The variable reference used in the Page Settings in the PCB editor.

Click OK to exit the Page Settings window. Pan the editor to the bottom right corner so you can see the layout Sheet title. It should contain the value of the "project_name" text variable (Figure 5.3.6):

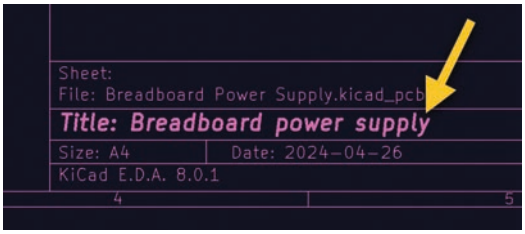


Figure 5.3.6: The variable reference is substituted for its value.

Let's try something else. Let's create a new text variable and use it as a text item in the schematic editor. First, open the Schematic Setup window and create a new variable named "input_circuit_name". For the value, enter "Input power and 5Volt subcircuit" (Figure 5.3.7).

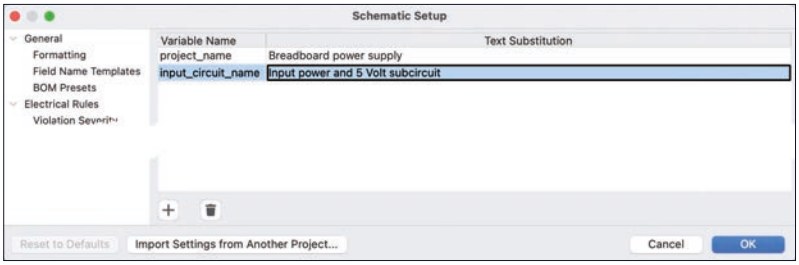


Figure 5.3.7: A new text variable.

Click OK and return to the editor. Create a new text item. In the Text properties window, enter a text string containing the variable reference and a fixed string. Notice that as you type the "{"to start typing in the text variable name, KiCad will give you a list of valid variable names. You can continue typing or choose one of the offered names (Figure 5.3.8).

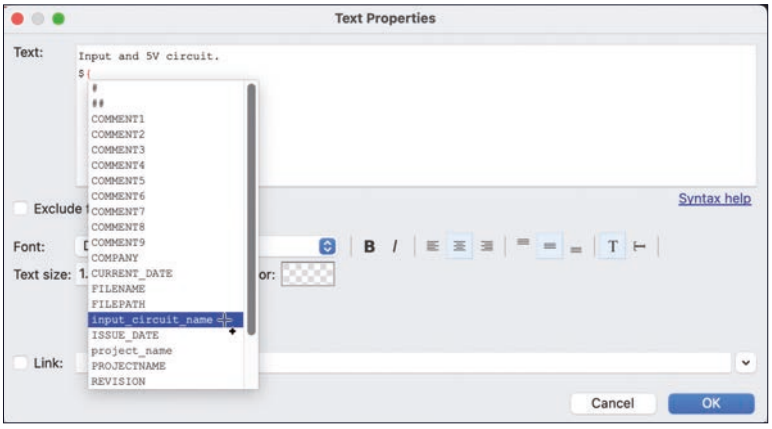


Figure 5.3.8: Code completion works with text variables.

You can see the contents of the Text field below (Figure 5.3.9):

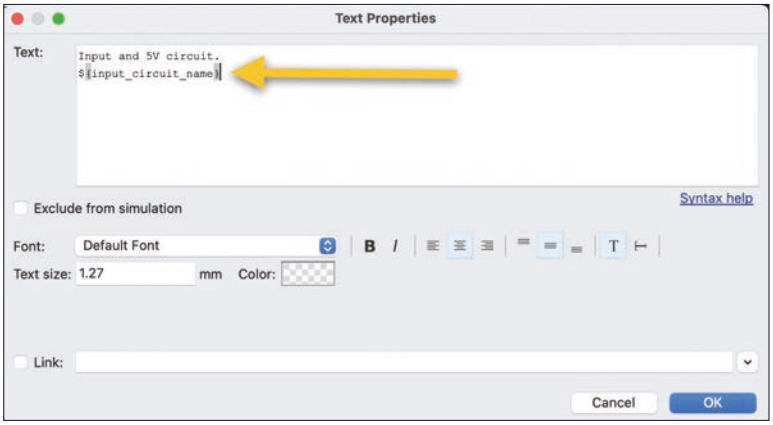


Figure 5.3.9: Example that combines fixed text with a text variable.

Click OK, and see the result of the text variable substitution in the schematic editor (Figure 5.3.10):

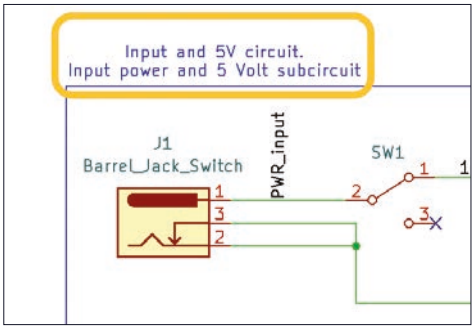


Figure 5.3.10: A text item with a text variable.

Let's see what happens if you change the value of an existing text variable. Change the value of the "input_circuit_name" to something else, like this (Figure 5.3.11):

Variable Name	Text Substitution
input_circuit_name	Input power and 5 Volt subcircuit....
project_name	Breadboard power supply

Figure 5.3.11: I have added a few dots in the text variable.

Click OK. The text item updates its content accordingly (Figure 5.3.12):

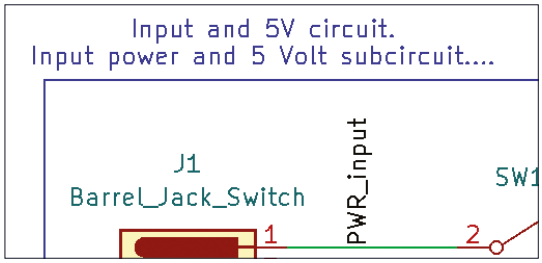


Figure 5.3.12: The changed text variable value in the text item.

You can expect the same behavior for text variables in the PCB editor. Here are some ideas of things that you can do with text variables:

- Better control of PCB version numbers that appear in the silkscreen. Instead of using fixed text version numbers, parametrise them.
- Standard pull-up or pull-down resistor values (or any other value you have a lot of in a design). Use a single text variable that makes it easy to adjust if necessary.
- URLs for PCB usage or specification information.

This is one of those features that once you are accustomed to, it will be hard to work without.

5.4. Board Setup - pre-defined sizes for tracks and vias

In the layout editor, you can set the physical dimensions of tracks, vias, and differential pairs in two ways:

1. Automatically, based on the net class to which these items belong.
2. Manually.

In most cases, you will want to use the automatic method. The dedicated chapter "Board Setup—Design Rules and net classes" explains it better.

The manual method gives you more freedom for tracks that do not neatly fit within an existing net class or for when you want to experiment. When you start the PCB editor for the first time in a project, there are no pre-defined (custom) track widths or via sizes. At the top menu, you can confirm this by clicking on the two drop-down widgets on the right side (Figure 5.4.1):

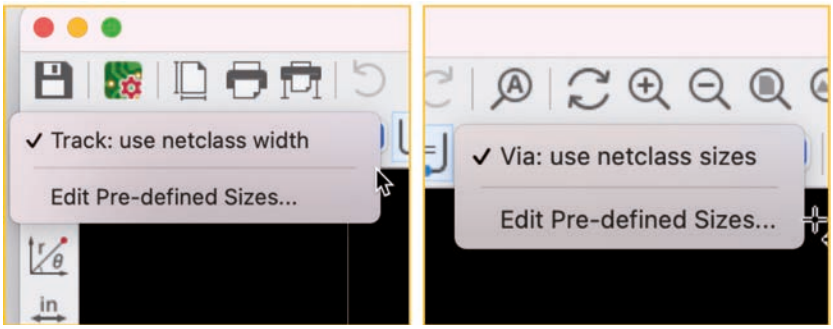


Figure 5.4.1: The track width and via sizes menus in a new project.

The default method for setting track and via width and size attributes is to adopt those specified in the item's net class. As you can see in the screenshots above, the net class option is the default selection in both cases. There is no other pre-defined option, but it is possible to create arbitrary ones that will be available through the same menu.

To create pre-defined track widths and via sizes, click on the last option of either dropdown widget, "Edit Pre-defined Sizes…". You can also click "File," "Board Setup, "and then "Pre-defined Sizes."

You will see the empty "Pre-defined track and via dimensions" list (Figure 5.4.2):

Figure 5.4.2: The pre-defined sizes tables in a new project are empty.

You can click on the "+" button at the bottom of each of the "Tracks", "Vias," and "Differential Pairs" lists.

Let's create a few pre-defined track widths. Each time, click the "+" button at the bottom of the Tracks list. A new row will appear in the list. Click inside the new cell to set the cursor, and type the custom values, such as "0.15", "0.20," and "0.25". While at it, I also created custom via and differential pair sizes. Here's an example from one of my boards (Figure 5.4.3):

Figure 5.4.3: Pre-defined sizes for tracks, vias, and differential pairs.

When finished, click on "OK." Click on the track width and via sizes dropdown in the top menu, and notice that the new pre-defined sizes are now available for use (Figure 5.4.4):

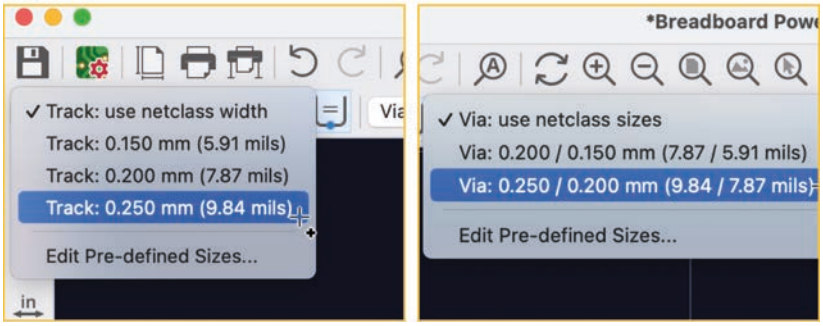


Figure 5.4.4: Pre-defined sizes for tracks and vias are now available for use.

To use the pre-defined sizes, you can use one of the following methods:

1. Select the size first, then create a new track via or differential pair.
2. With the track, via, or differential pair tool already selected, you can right-click in the editor to bring up the context menu and then select one of the available sizes from the Select Track/Via Width menu (Figure 5.4.5).

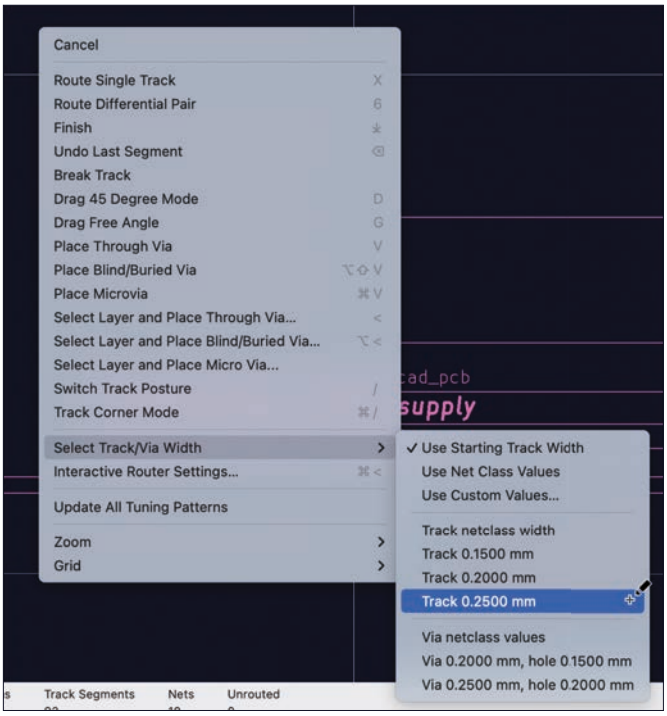


Figure 5.4.5: You can select pre-defined sizes through the context menu; this requires the track, via, or differential pairs tool to be enabled.

I used the second method to draw a single track with three different track widths (Figure 5.4.6):

Figure 5.4.6: A single track composed of segments with different widths.

A track like this would not be possible to draw using the automated net class method.

In summary, while the preferred method for defining the sizes of track and vias is to have them inherited from the net classes (I call this the "automated" method), you can set custom sizes, as you learned in this chapter.

5.5. Board Setup - Design rules violation severity

The PCB editor contains an analysis tool that can detect defects in your layout and then classify those defects into three categories: Errors, Warnings, and Ignore. While the default settings for the classifier are reasonable, you may want to experiment with alternative settings. The PCB editor allows you to modify the violation severity classifier. The schematic editor also has its own violation severity and classifier, which you can customize (learn about it in the dedicated chapter "5.7. Schematic Setup - Electrical Rules and violation severity").

To access the Violation Severity settings, click on File, Board Setup, Violation Severity (Figure 5.5.1).

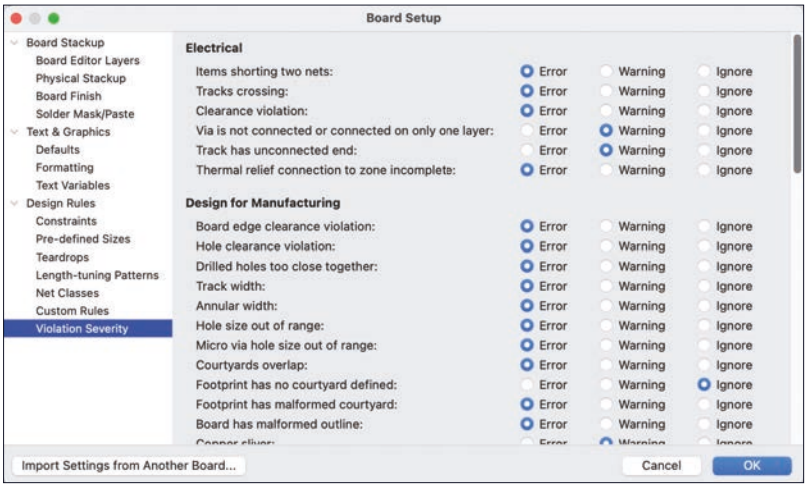


Figure 5.5.1: The Violation Severity settings in the Board Setup window.

In the Violation Severity settings, you can customize the classification of any condition that the Design Rules Checker can identify. For example, you can classify "Hole clearance violation" as an Error or a Warning.

I will explain how the Violation Severity classifier works with the help of an example. Consider the layout below (Figure 5.5.2):

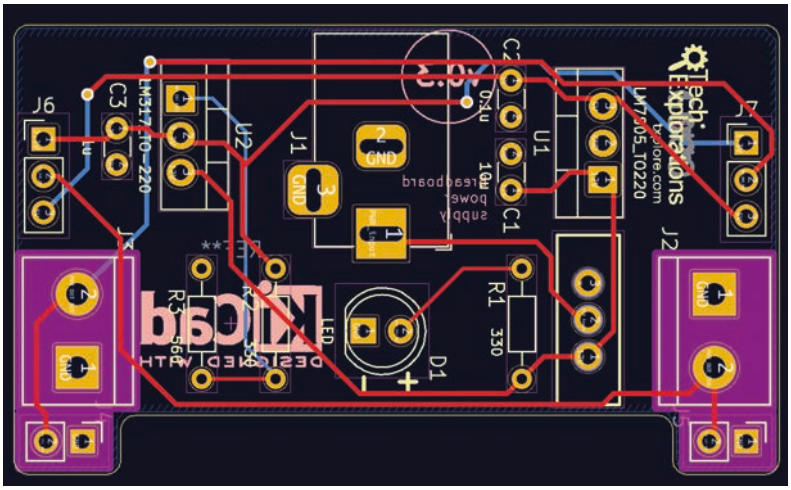


Figure 5.5.2: I'll run the Design Rules Checker on this layout.

The layout shown above comes from one of the projects in this book. When I run the DRC (Design Rules Checker) using the default Violation Severity settings, I can see that four tests are currently being ignored:

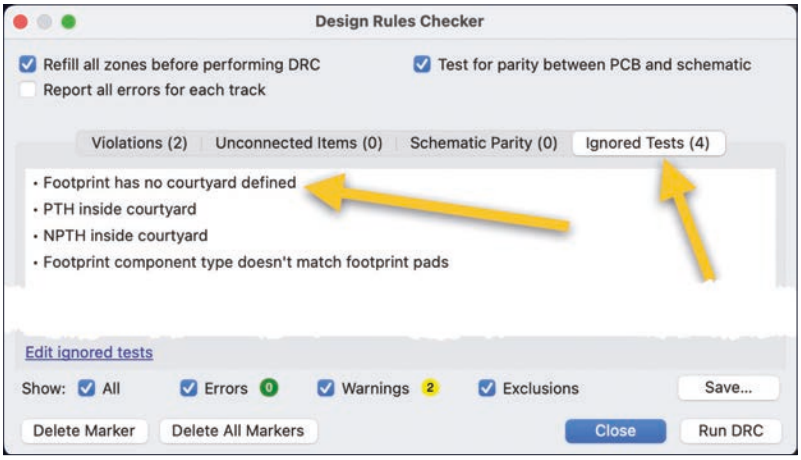


Figure 5.5.3: The DRC output using the default Violation Severity settings.

One of those tests can detect footprints without courtyards. I'd like to enable those tests so that the DRC can give me warnings if such footprints exist in my layout. I can customize the violation severity settings to enable this test.

Bring up the Violation Severity dialogue, and look for the "Footprint has no courtyard defined" setting under "Design for Manufacturing." Change the severity type to "Warning." The dialog should look like this (Figure 5.5.4):

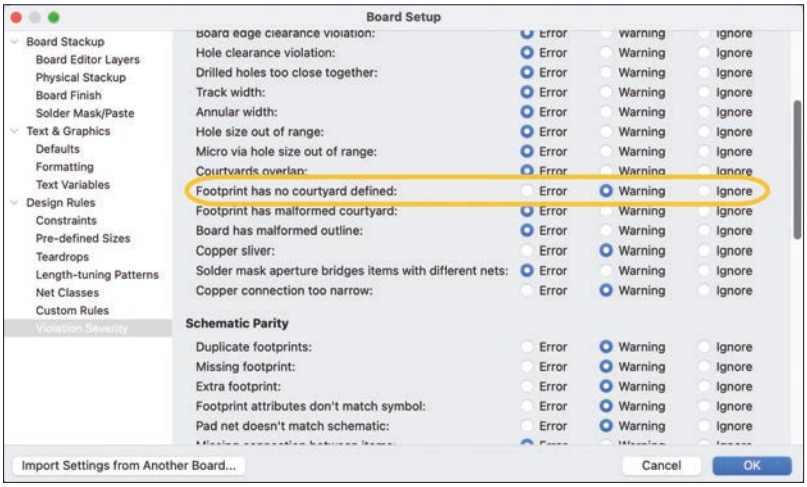


Figure 5.5.4: I have changed the violation severity type to "error" for the "Extra footprint" setting.

Next, perform a new DRC. The result is below (Figure 5.5.5):

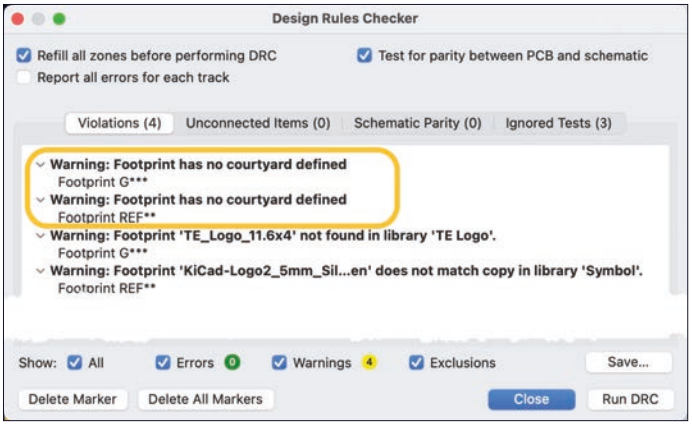


Figure 5.5.5: The two "Extra footprint" issues are now classified as "Error."

The DRC now classifies the two "Footprint has no courtyard defined" conditions as "Warnings." You can make similar changes to any condition the DRC recognizes. The recognizable conditions are listed in the Violation Severity dialogue box. They are numerous and offer a basic level of flexibility for discovering and classifying violations.

Another violation that is useful to consider is the "footprint does not match library" violation. You can see an example below:

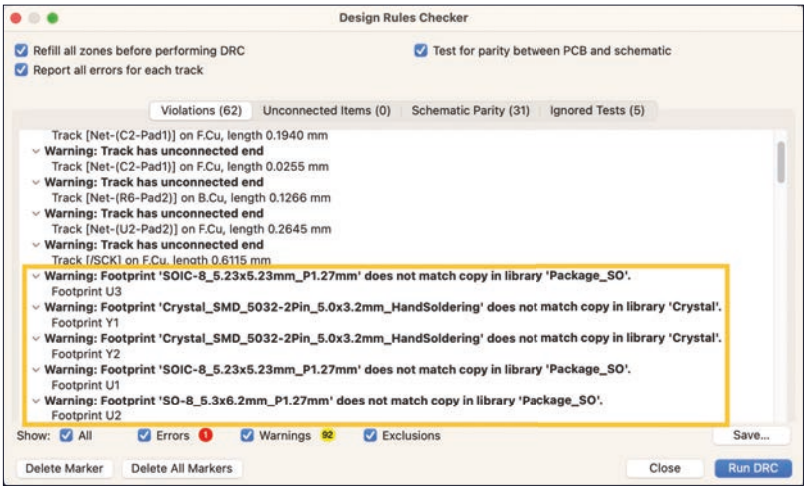


Figure 5.5.6: These footprints do not match the library originals.

This warning highlights that a footprint that initially came from a library has since been modified. Therefore, there is a deviation between the library original and the footprint used in the PCB. It is up to you to decide if this is acceptable or if you should consider associating the symbol with the original footprint to maintain consistency.

Consider using the Custom Rules feature under Design Rules if you need more customization power. You can learn more about this in a dedicated chapter.

5.6. Board Setup - Custom design rules

The PCB editor allows you to customize the classification of design rule violations. A dedicated chapter in this book explains this. The Violation Severity dialogue offers a simple way to change how the violation classifier works.

The powerful "Custom Rules" feature allows you to customize the design rules checker to a much higher level. Custom Rules allows you to write rules using a programmatic language. This language derives from S-Expressions (https://en.wikipedia.org/wiki/S-expression), invented for use in the Lisp programming language. Here is an example of a custom DRC rule:

```
(version 1)

# This is an example DRC rule.
# The name of the rule is ExampleMinPowerNetClearance
# The rule sets a minimum of 1mm clearance between a
# track that belongs to the Power net and other elements.

(rule ExampleMinPowerNetClearance
       (constraint clearance (min 1.0mm))
       (condition "A.NetClass == 'POWER'"))
```

Here is what this code looks like in the Custom Rules dialogue box. You can bring up this boxby clicking on File —> Board Setup —> Custom Rules (Figure 5.6.1):

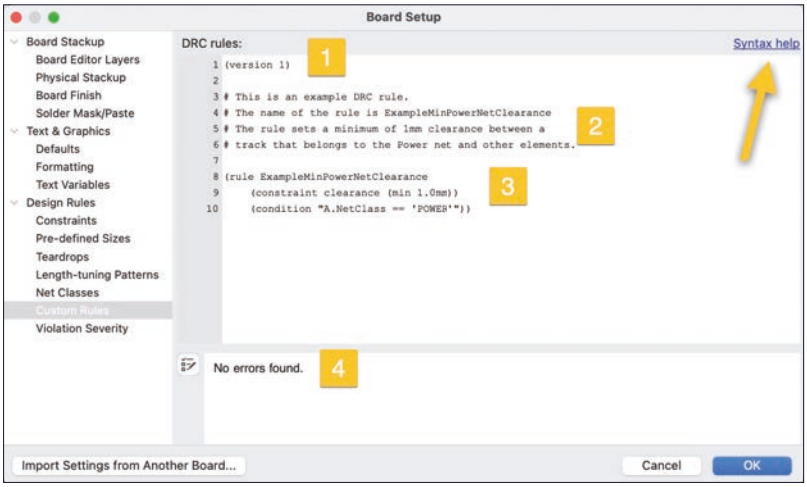


Figure 5.6.1: An example custom DRC rule.

The rule starts with a version number (1). You can add comments by prepending a comments line with the "#" character (2). The rule starts with the "rule" command, followed by a name. In separate lines, you can define the constraints and conditions (3). A custom rule validator at the bottom of the dialogue box looks for bugs in the code. You can invoke the validator by clicking on the "checklist" button on the left side of the validator output text box. In this example, my custom rule has no errors (4).

When you click "Syntax help" at the top right of the Customs Rules dialogue box, a pop-up window will appear with detailed syntax information and rule code examples. Per the S-Expressions notation, you should enclose each clause in parentheses when you write rules.

You can define multiple conditions or constraints using booleans or placing them in separate lines. Here is a modified example:

```
(rule ExampleMinPowerNetClearance
       (constraint clearance (min 1.0mm))
       (constraint track_width (max 2.0mm))
       (condition "A.NetClass == 'POWER'")
       (condition "A.NetClass == 'Data"))
```

This rule contains two constraints and two conditions, each in its line, which helps with clarity. Of course, you can create more than one custom rule. As in a program in Python or Ruby, you can include multiple functions in a single program file. In the PCB editor, you can add multiple custom rules in the DRC rules window. Here is an example:

```
(version 1)

# This is an example DRC rule.
# The name of the rule is ExampleMinPowerNetClearance
# The rule sets a minimum of 1mm clearance between a
# track that belogns to the Power net and other elements.

(rule ExampleMinPowerNetClearance
       (constraint clearance (min 1.0mm))
       (constraint track_width (max 2.0mm))
       (condition "A.NetClass == 'POWER'")
       (condition "A.NetClass == 'Data"))

# This is a second DRC rule.
# The name of the rule is ExampleMinPowerNetClearance2

(rule ExampleMinPowerNetClearance2
       (constraint clearance (min 0.5mm))
       (condition "A.NetClass == 'GPIO"))
```

The second rule, ExampleMinPowerNetClearance2, sets a minimum clearance for tracks belonging to the GPIO net class. The rule editor can do code completion. It will automatically present a list of valid keywords as you type. Below are some examples (Figure 5.6.2):

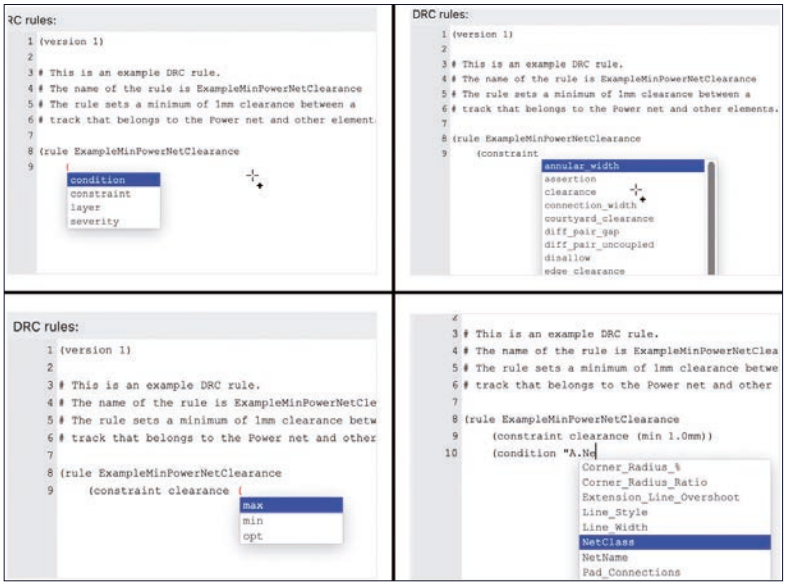


Figure 5.6.2: Examples of custom rules code completion.

I can create custom rules quickly and with (usually) no errors when completing code. Once a custom rule is ready and validated, click the OK button to enable the rule(s). Then, test by running the DRC. Here is the output of the DRC with the custom rule that I listed at the start of this chapter (Figure 5.6.3):

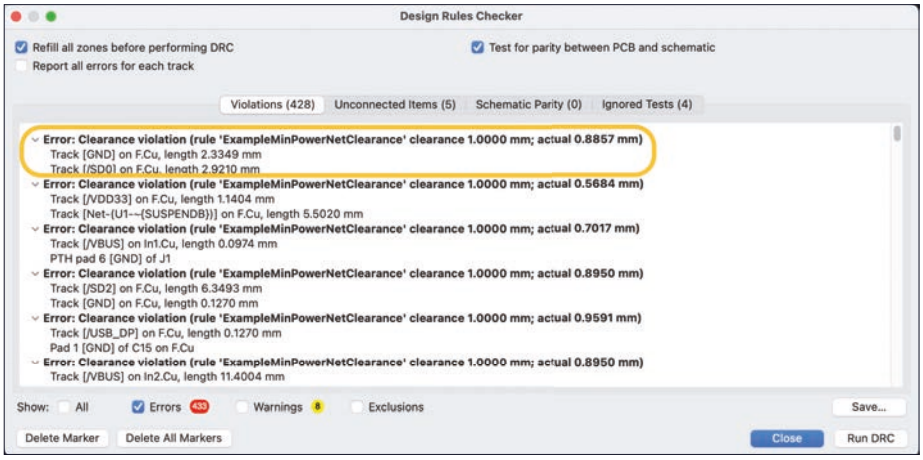


Figure 5.6.3: The custom rule has generated a lot of errors

As you can see, the DRC will use the name of the custom rule in its output so that you can identify the custom rule that fired. In this example, because the constraint that I set was so large (minimum clearance 1mm), the DRC generated 82 errors. Most of those errors are created by custom rules.

With each new release of KiCad, the Custom Rules language gains new vocabulary. For example, in KiCad 7, the language gained two new mechanical constraints ("mechacnical_clearance" and "mechanical_hole_clearance") and four new severities ("ignore," "warning," "error," and "exclusion"). Use the detailed documentation (https://docs.kicad.org/8.0/en/pcbnew/pcbnew.html#custom-design-rules) and the information in the Syntax Help window to look for syntax and directives. I recommend you take a moment to look at the Syntax Help contents after you install an update to find out if there are any new additions or changes to the language.

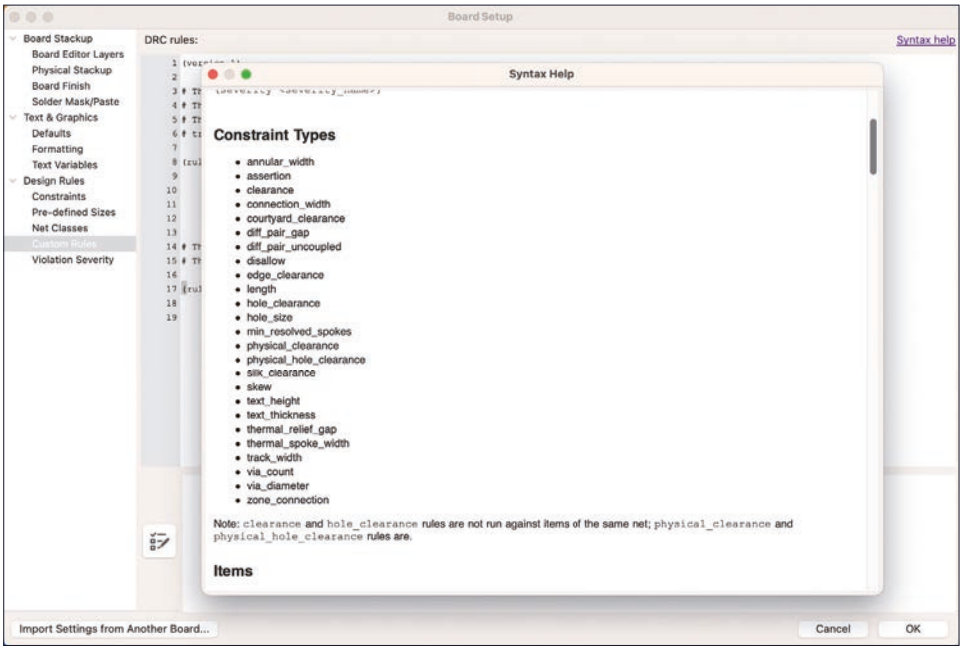


Figure 5.6.4: The Syntax Help window contains useful information.

The Custom Rule feature is powerful, and you should use it carefully. Start by looking at syntax help and becoming familiar with S-Expressions. Continue by writing simple rules for your layouts so that you can quickly become comfortable with this powerful feature.

5.7. Schematic Setup - Electrical Rules and violation severity

Like the PCB editor, the schematic editor provides a tool for checking for violations in your design. This tool is the Electrical Rules Checker (ERC), which I have used extensively in all of the projects in this book.

The ERC's default classification settings, such as the Design Rules Check (DRC) in the PCB editor, can change. You can do this via the Violation Severity dialogue in the Schematic Setup window. Please look at the relevant chapter to learn about the Violation Severity configuration in the PCB editor.

You can access the Violation Severity dialogue in the schematic editor by clicking File —> Schematic Setup —> Violation Severity (in the Electrical Rule group). The dialog box looks like this (Figure 5.7.1):

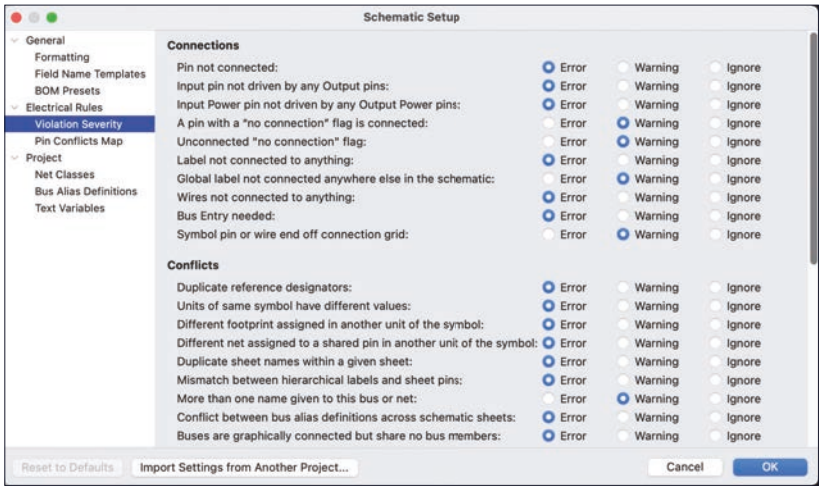


Figure 5.7.1: The Violation Severity dialog box in the PCB editor.

The default settings are reasonable, and you rarely need to make any changes. Let's look at an example to help you understand how to change a violation's classification during an ERC. In my test schematic below, I have introduced a defect by deleting a wire segment (Figure 5.7.2).

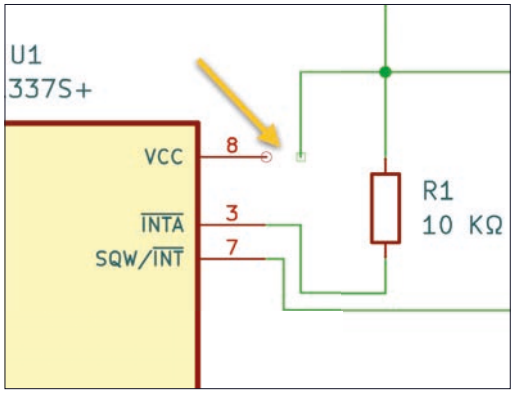


Figure 5.7.2: An evident defect: a wire segment is missing.

Under " Connections, " the violation at the very top of the list," is "Pin not connected." The default classification for this violation is "Error." Go ahead and change this to "Warning" (Figure 5.7.3):

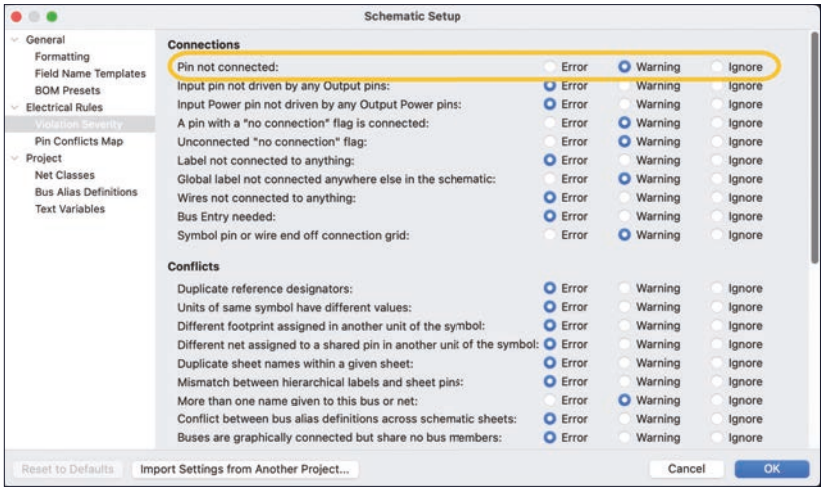


Figure 5.7.3: Changed classification of "Pin not connected" to "Warning."

Let's see the impact of this change on the way that the ERC classifies schematic violations. Bring up the ERC window (Inspect —> Electrical Rules Checker) and click on "Run ERC." You can see the result below (Figure 5.7.4):

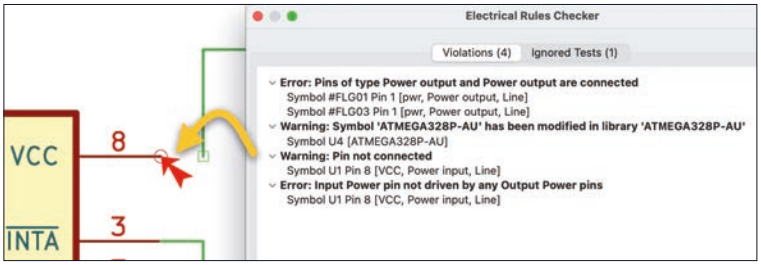


Figure 5.7.4: The ERC lists the missing wire segment as "Pin not connected" and a warning.

The ERC lists the missing wire segment as "Pin not connected," classified as a "Warning." Notice that in this ERC, the total number of errors is zero. Change the "Pin not connected" classification to its original "Error," and repeat the ERC. The result is below (Figure 5.7.5).

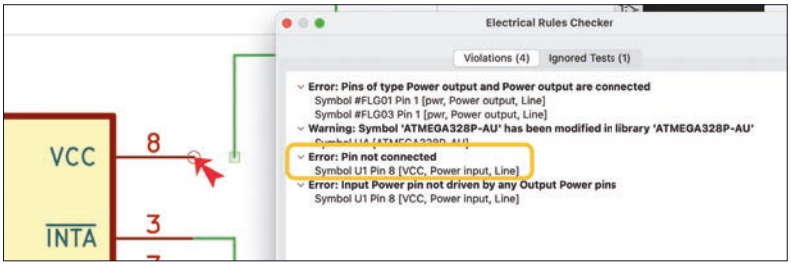


Figure 5.7.5: The ERC lists the missing wire segment as "Pin not connected" and an error.

The same violation that was previously listed as a Warning is now listed as an Error. You can further customize the ERC by looking at the Pin Conflicts Map dialogue box. This allows you to set violation classifications for the various pin-to-pin conditions. You can learn about this in a dedicated chapter.

5.8. Schematic Setup - Electrical Rules and Pin conflicts map

The Pin Conflicts Map dialogue box (in the Schematic Setup window, under Electrical Rules) allows you to set violation classifications for the various pin-to-pin conditions.

To access this dialogue, go to File, Schematic Setup, and click "Pin Conflicts Map" under "Electrical Rules." The user interface of this tool resembles an assignment triangle. The intersecting points between columns and rows are buttons. Each time you click on a button, you can change its value into one of three possible values:

- No error or warning.
- Generate warning.
- Generate error.

Let's look at an example. Bring up the Pin Conflicts Map dialog box. It looks like this (Figure 5.8.1):

Figure 5.8.1: The Pin Conflicts Map

In this example, I have highlighted the conflict classification for a connection between two pins designated as outputs. This type of connection is classified as "Generate error." Similarly, a connection between an open connector pin and an output pin will generate an error.

Also, notice that the last row of the triangle is set to generate errors for all non-connected pins. In my test schematic, I have left pin 23 unconnected, and the ERC lists this as an error (Figure 5.8.2):

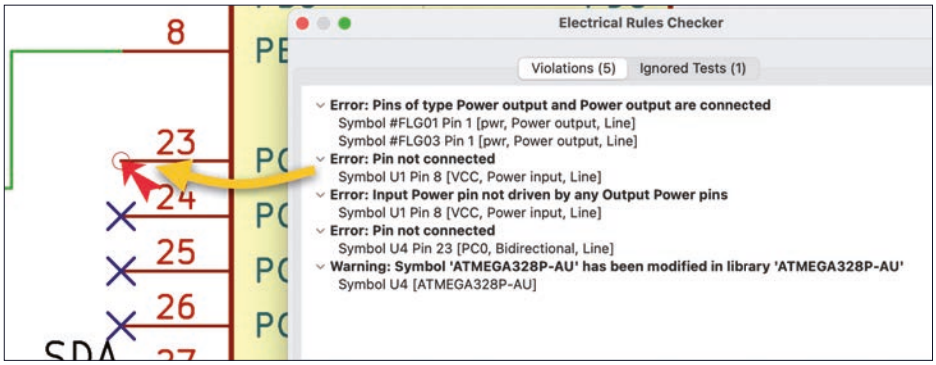


Figure 5.8.2: ERC reports an unconnected pin error.

Let's do a quick experiment that uses this test schematic (Figure 5.8.3):

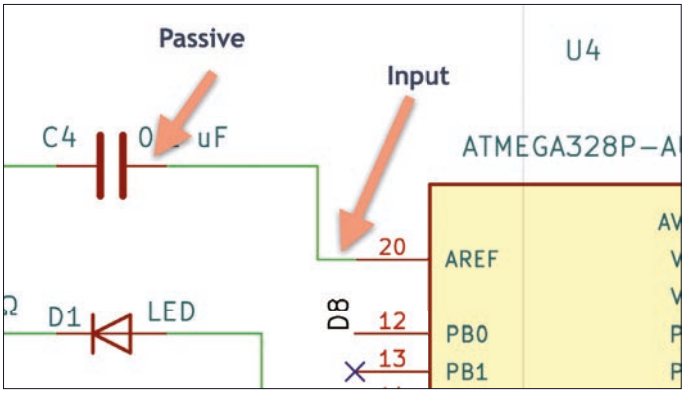


Figure 5.8.3: A connection between a passive and an input pin.

In this example, there is a connection between a capacitor pin (set as "passive" in the symbol properties) and pin 20 of the Atmega328P symbol, which is designated as "input." You can check the pin type by double-clicking on the symbol to bring up its properties window and then click the Pin function tab (Figure 5.8.4).

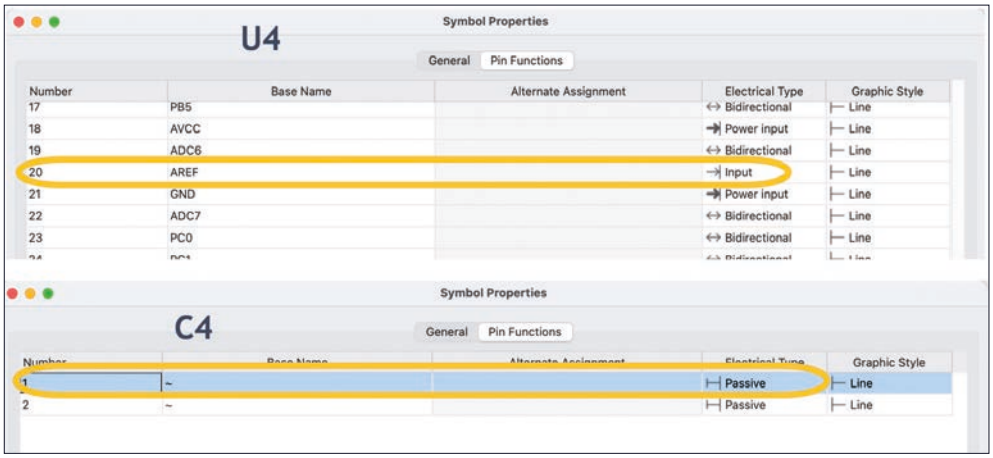


Figure 5.8.4: The type of pins connected in my test schematic.

As you can see in figure 5.8.1, the classification for connection between passive and input pins is "green"; therefore, no error or warning is generated. Truthfully, when I ran the ERC earlier (see Figure 5.8.3), the ERC did not list any relevant violations.

Let's change this classification. Open the Pin Conflicts Map, and click on the button in the junction of the Passive Pin row and Input Pin column to show the red "!" mark (see Figure 5.8.5).

Figure 5.8.5: Passive to input pin connections are now classified as "error".

Click OK, and rerun the ERC. Here is the result (see Figure 5.8.6):

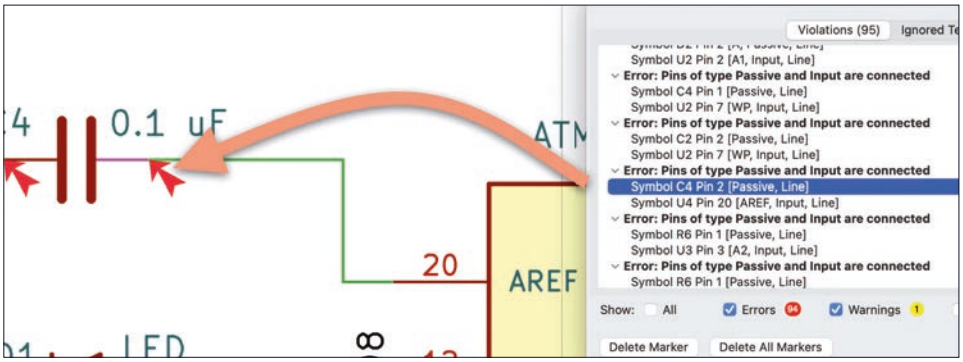


Figure 5.8.6: The ERC detects passive to-input pin connection errors.

The ERC has detected a large number of passive-to-input pin errors. One is between the capacitor in pin 20 of the Atmega328P. I don't want this connection to be classified as an error, so I have restored the violation classification to "green."

The Pin Conflicts Map gives you another way to customize the operation of the ERC. When used with care and consideration, it can help ensure that KiCad meets your project's specific requirements.

5.9. Import components from Snapeda

In this chapter, you will learn how to import symbols, footprints, and 3D models (3D shapes) in KiCad from Snapeda.com. To demonstrate the process, I have set up a new KiCad project. In this project, I want to use a component that does not exist in the Kicad libraries: the Texas Instruments MSC1212Y5PAGT central processing unit.

Snapeda is an online repository of electronic CAD components. In my day-to-day work with KiCad, I turn to Snapeda to look for symbols, footprints, and 3D shapes whenever I need one that does not exist in the libraries that come with KiCad. I often look at Snapeda for better versions of components already existing in KiCad. Because it is such a helpful resource, I decided having a dedicated Recipe chapter in this book is worth it.

If you wish to follow along, go to https://www.snapeda.com and create a new free account, or log in to your account if you already have one. Below is the homepage of snapeda.com (Figure 5.9.1):

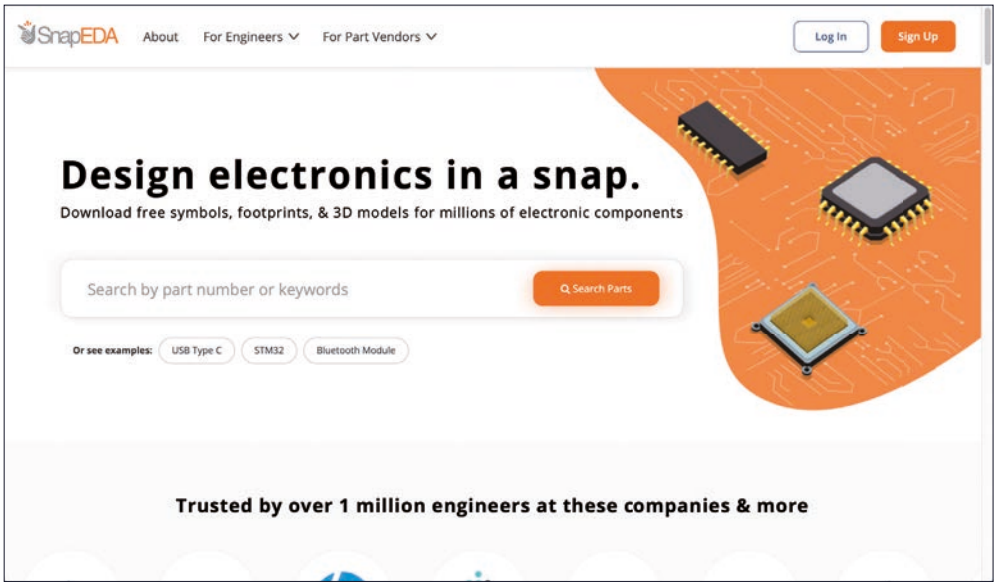


Figure 5.9.1: The Snapeda homepage.

Once logged in to Snapeda, use the search bar at the top of the page to search for the necessary component. You can search with the model number, like "MSC1212Y5PAGT", or a keyword such as "Texas Instruments 8051 CPU". Snapeda will give you a list of hits. Select the one that matches your requirements by clicking on it. In my case, I will be working with this component[12] (Figure 5.9.2):

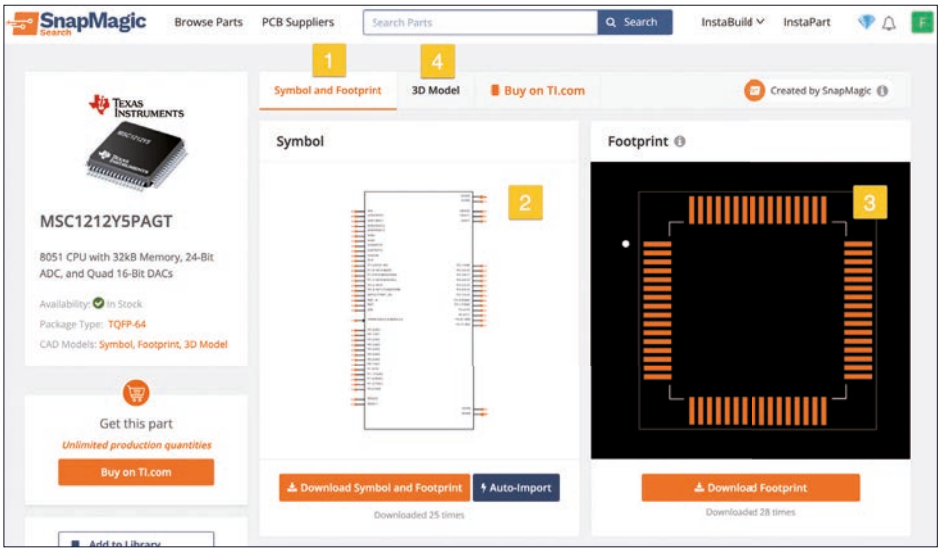


Figure 5.9.2: The TI MSC1212Y5PAGT CPU component.

12 MSC1212Y5PAGT, 8051 CPU with 32kB Memory, 24-Bit ADC, and Quad 16-Bit DACs 64-TQFP -40 to 125
https://www.snapeda.com/parts/MSC1212Y5PAGT/Texas%20Instruments/view-part/41461/?ref=search&t=MSC1212Y5PAGT

The component page contains two main tabs: symbol and schematic (1) and 3D model (4). Under the "Symbol and Schematic" tab, you will find the Symbol (2) and the Footprint (3). To see the 3D model, click on the 3D Model tab. It looks like this (Figure 5.9.3):

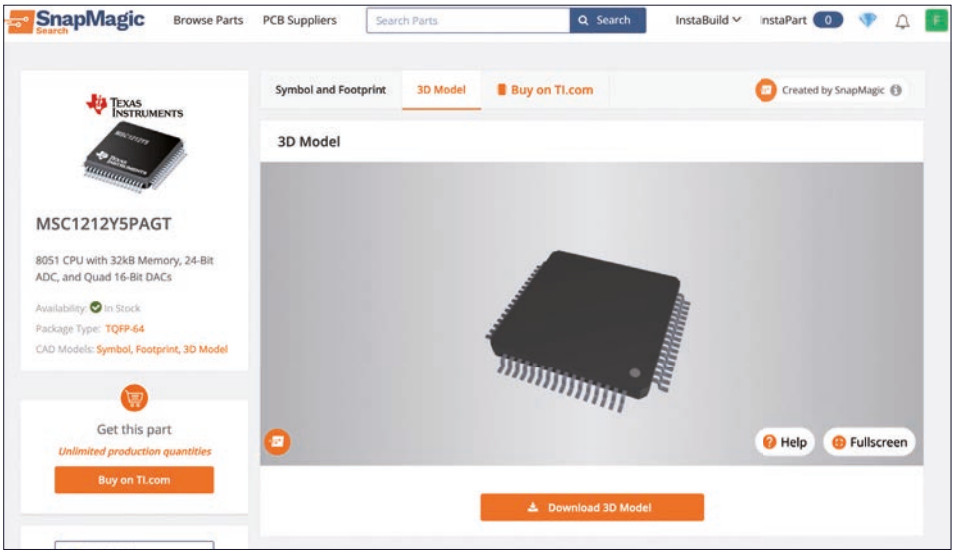


Figure 5.9.3: The 3D model of the component in question.

Back in the results page, the "Data Available" column will indicate whether the 3D model and other parts of the component are available. In the example in Figure 5.9.4 (below), notice the icons under "Data Available":

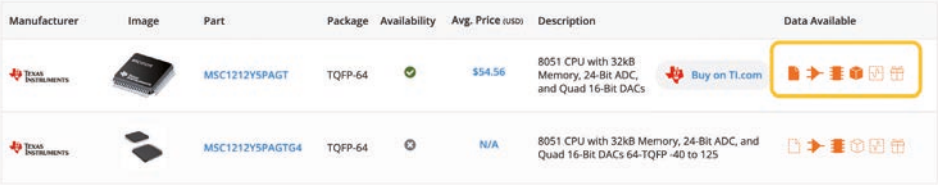


Figure 5.9.4: The available data for the search results.

A solid-colored icon indicates that data is available. From left to right:

- Datasheet.
- Symbol.
- Footprint.
- 3D model.
- Simulator model.
- Sample.

The MSC1212Y5PAGT CPU component I use in this example provides the datasheet, symbol, footprint, and 3D model. Let's download the symbol, footprint, and 3D model. Click on the

"Symbol and Footprint" tab. Then, click on the "Download Symbol and Footprint" button. Snapeda will ask you to select the CAD software you will be using, so click on "KiCad" and select the latest available version of KiCad for the download. As you probably expect, this will download the symbol and the footprint, but it will also download the 3D model. Snapeda will archive these files in a single ZIP file. Extract them from the ZIP file (Figure 5.9.5).

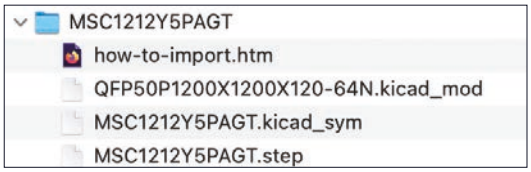


Figure 5.9.5: The symbol, footprint, and 3D model files.

The folder I extracted from the ZIP file contains:

- Symbol file with the ".kicad_sym" extension.
- Footprint file with the ".kicad_mod" extension.
- 3D model file with the ".step" extension.

Now that you have these files on your computer, import them into KiCad. You can learn how to do this in My book Fundamentals and Projects.

I have imported the files to my KiCad instance and will use them. I'll start with the symbol. I type the "A" hotkey in my schematic editor to add a new symbol. This brings up the Symbol Chooser. I quickly type a few of the first letters for the new component to locate it in the library. The Chooser promptly finds the symbol. Click on it to select it and show its preview in the right-side pane (Figure 5.9.6). Because I have also installed the symbol's corresponding footprint, the footprint is already linked to it, and the footprint appears in the preview window underneath the symbol.

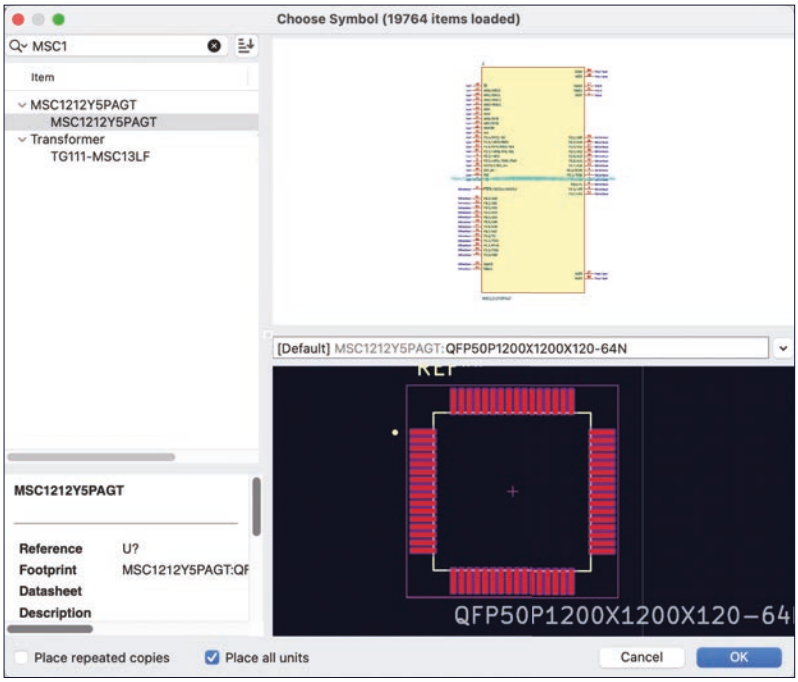


Figure 5.9.6: The new symbol.

Double-click on the symbol to add it to the schematic editor (Figure 5.9.7):

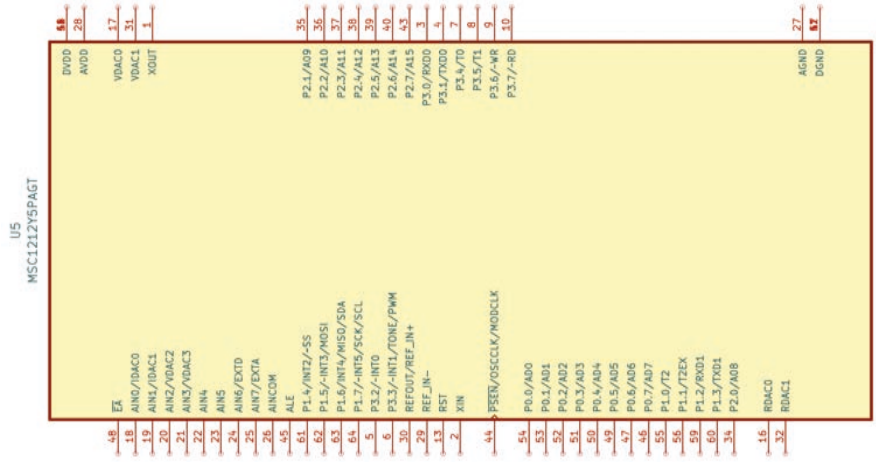


Figure 5.9.7: The new symbol in the editor.

Once the new symbol is in your schematic editor, you can use it with any other symbol. Let's continue with the footprint for the same component. I assume that you have already imported the footprint to KiCad. Double-click on the symbol to open up the symbol properties window. It looks like this (Figure 5.9.8):

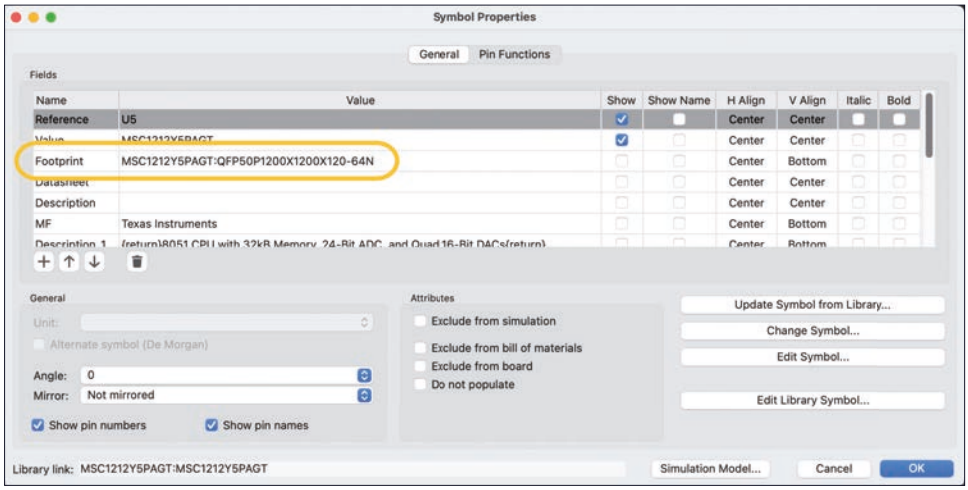


Figure 5.9.8: The new component symbol properties window.

The "Footprint" field contains a link to the default footprint of this footprint that came in the Snapeda package. Of course, you can select a different footprint if you wish. I have now associated the new symbol with its matching footprint.

Next, let's make use of the new 3D shape. To do that, open the PCB editor, update the layout editor with the schematic data, and draw a rectangle graphic around the only footprint in the Edge.cuts layer (Figure 5.9.9).

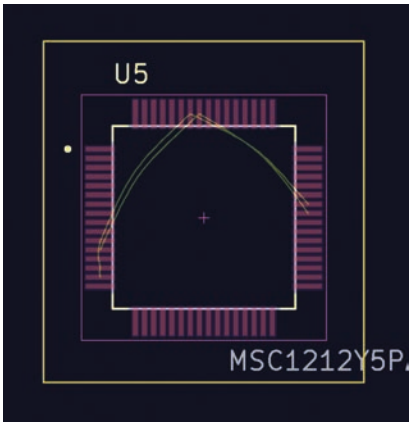


Figure 5.9.9: The new footprint in the PCB editor with an Edge.cuts graphic.

I will set a 3D shape for the new footprint. Double-click on the footprint, and click on the 3D Models tab. By default, this footprint does not have a 3D model defined, so there are no rows in the 3D Model(s) list (Figure 5.9.10).

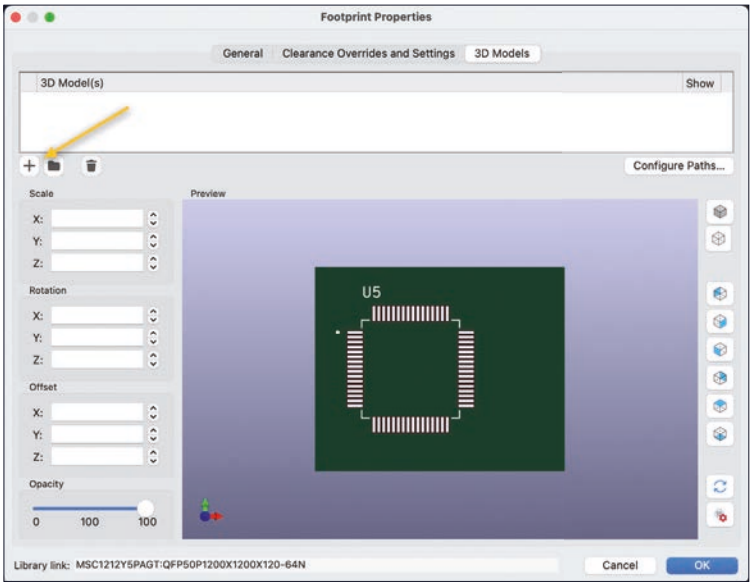


Figure 5.9.10: The new footprint's 3D Models tab.

To add a new 3D symbol, click on the "+" button to add a new row (Figure 5.9.11):

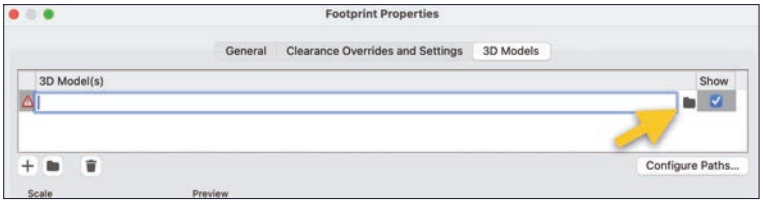


Figure 5.9.11: A new row to hold the path for the 3D model.

Click the folder button (the right side of the new 3D model row). This will bring up a file browser to locate and select the 3D shape (".step") file. In my example, it looks like this (Figure 5.9.12):

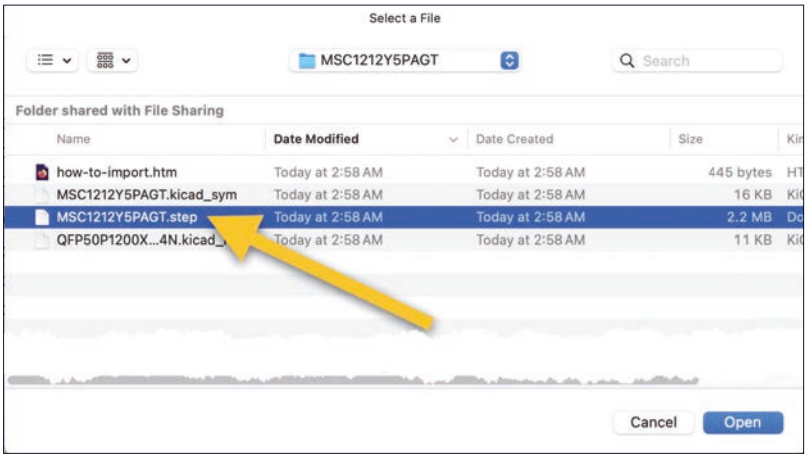


Figure 5.9.12: Double-click on the 3D model ".step" file to add it to the footprint properties.

Double-click the ".step" file to add it to the footprint properties. The Preview pane will superimpose the model over the footprint, but it may initially be out of place. You must adjust some placement parameters (scale, rotation, and offset) to align the model with the footprint. Below you can see my settings (I only changed the X rotation to -90°) (Figure 5.9.13):

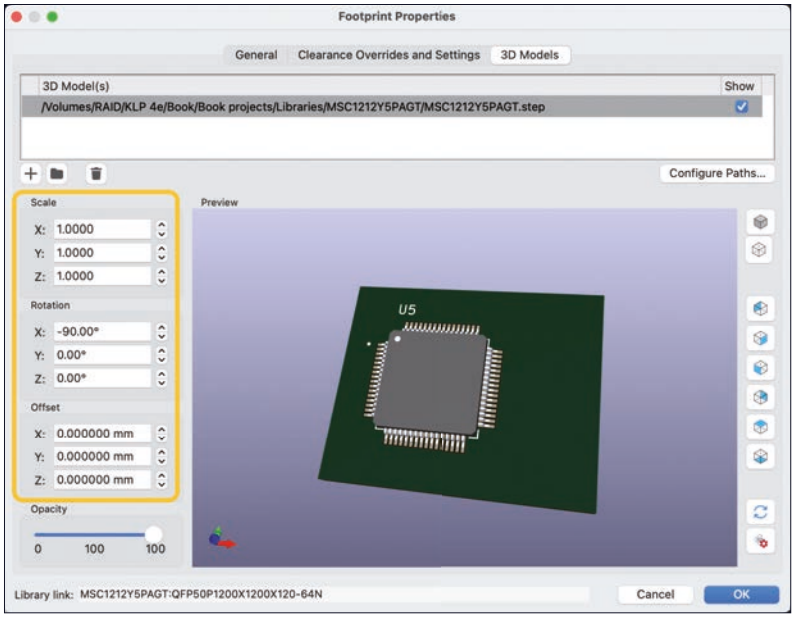


Figure 5.9.13: With a bit of tweaking, the 3D model aligns with the footprint.

Click OK to dismiss the window. Back in the PCB editor, bring up the 3D Viewer from the View menu to see the new 3D model (Figure 5.9.14):

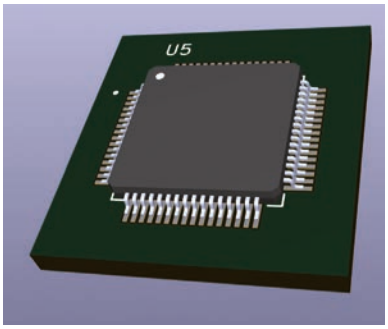


Figure 5.9.14: The new 3D model as it appears in the 3D Viewer.

Snapeda is an excellent source for symbols, footprints, and 3D shapes, but it is not the only one. In previous chapters, I discussed other sources you can consider when looking for project components.

5.10. The Freerouting autorouter

The routing process of a new PCB is often the most time-consuming one. Routing can be tedious and frustrating, especially in small PCBs with many components. An autorouter can help in such cases. An autorouter is software that can automatically create tracks using various algorithms optimized for minimizing the total track length or vias used. Most modern autorouters offer extensive customizability.

Should you use an autorouter? Opinions differ, and appropriate use cases for autorouters vary. Autorouters tend to be "naive." While the designer may customize an autorouter to some extent, autorouters do not "understand" the specific peculiarities of the various circuit segments on which they operate. For example, an autoroute will not be able to "understand" the importance of the shape and length of a differential track that connects a high-speed CPU with a memory chip or the track that connects an RF component to an antenna. Autorouted boards also tend to look less refined, with tracks drawn mechanically distinctively different from tracks drawn by a skilled designer.

Professional PCB designers rely on manual routing for all but trivial PCBs. Autorouters are often used extensively during the prototyping process, and the designer will often edit an autorouted board to improve it.

With all that in mind, auto-routing is a tool that is worth knowing and using when appropriate and with care. This chapter will show you how to use an external auto-router that works well with KiCad and Freerouting.

Below is an example of what an autorouted board looks like using Freerouting (left side). The autorouter completed the operation in approximately 3 minutes. The same board I routed is on the left side of the screenshot. It took me around 90 minutes to complete this work (Figure 5.10.1).

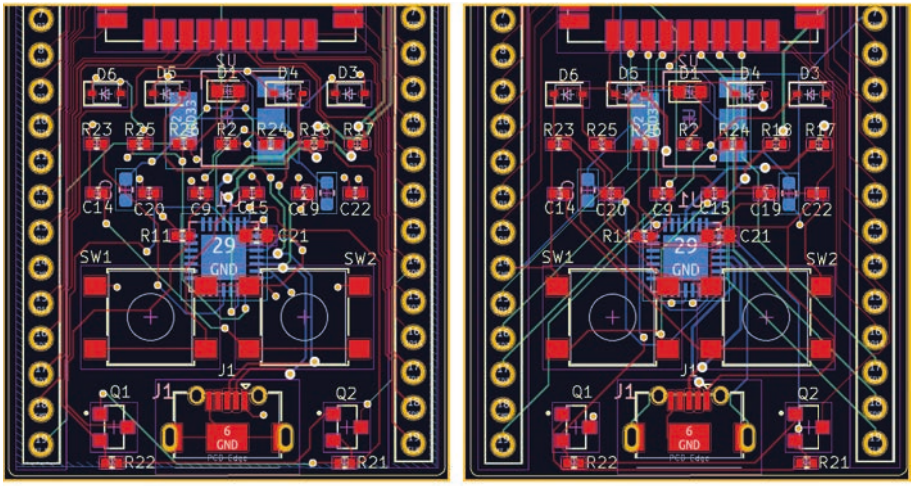


Figure 5.10.1: The same board routed automatically (right) and manually (left).

"Freerouting" is not only an autorouter. You can manually route a board, similar to KiCad's PCB editor. Below is an example (Figure 5.10.2). I have deleted some of the autorouted tracks, enabled the Route functions, and used the mouse to draw a new route that connects the two pads. Freerouting "knows" which pads and pins can be connected because I have imported a file I generated earlier in KiCad (you will learn how to use Freerouting with KiCad later in this chapter).

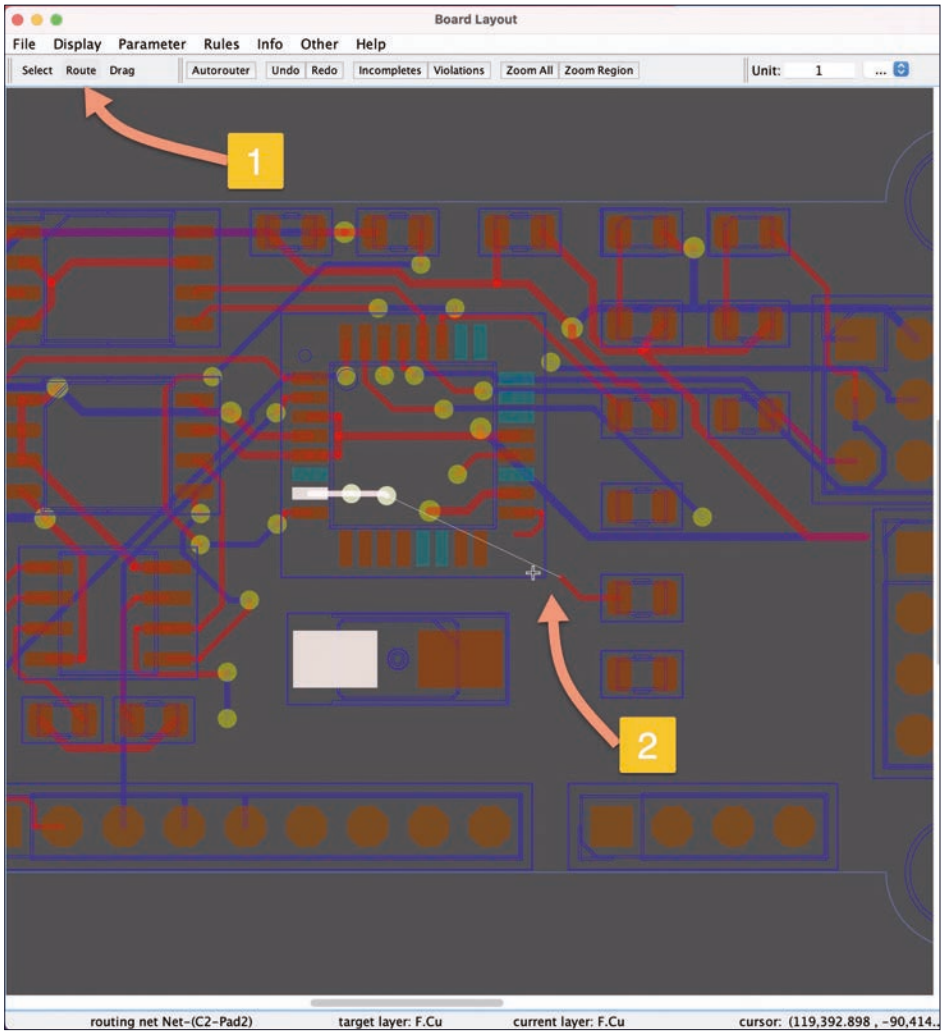


Figure 5.10.2: Freerouting is also useful as a manual router.

Freerouting is an open-source Java application that works with the KiCad, Eagle, and LayoutEditor CAD packages. Before KiCad 7, you would have to export your PCB in a file and then import it to Freerouting to commence the autorouting process. Luckily, with the introduction of the plug-in manager in KiCad 7, Freerouting is now available as a plug-in and can be installed with a single click. From there, you can start Freerouting with the click of a button from the top toolbar of the PCB editor. Once Freerouting starts, it will automatically start routing your board.

Freerouting can automatically route a PCB in multiple layers and is very configurable. You can configure many aspects of the autoroutes operation, such as the preferred direction of the traces on each layer, whether to use vias, and whether to do a post-route pass to try to reduce the number of vias or total trace length.

In the following few segments in this chapter, I will show you how to install Freerouting on KiCad as a plugin and then how to use it to route the most complex PCB in this book automatically.

5.10.1. Freerouting plugin installation and operation

Since KiCad 7, Freerouting has been available as a plugin. With the Freerouting KiCad plugin, the Freerouting functionality is integrated into the layout editor app. You can auto-route a PCB with a single click of your mouse.

In this chapter, you will learn how to install the Freerouting KiCad plugin and use it to route the ESP32 clone PCB automatically. Referencing the screenshot below, begin by opening the Plugin and Content Manager (PCM) from the KiCad project window. Then, select the KiCad official repository ("1" and "2"), select the Plugins tab ("3"), and search for "free" in the search box ("4").

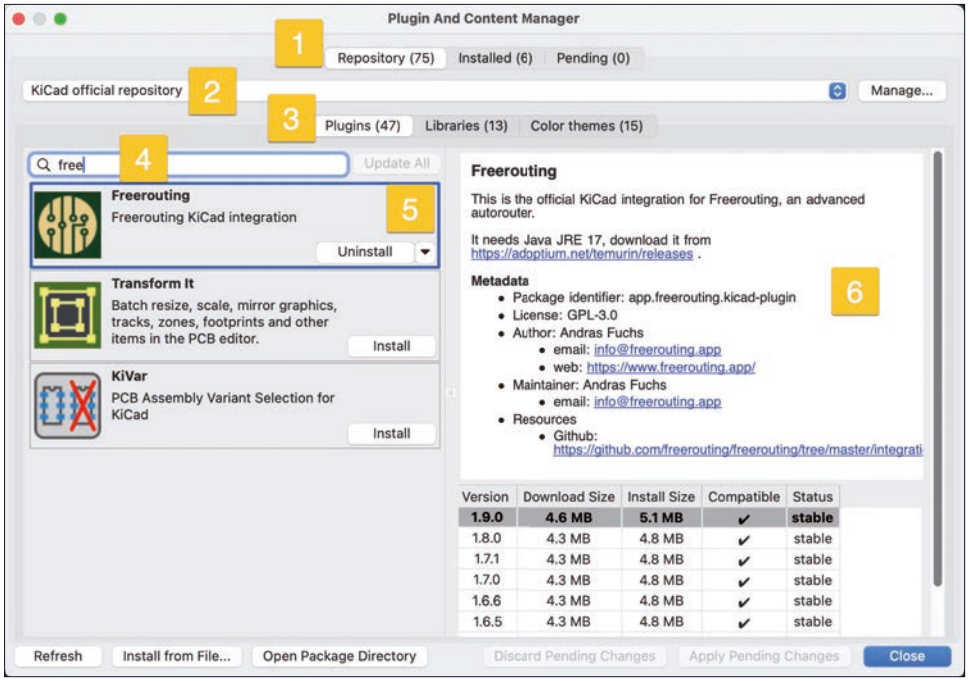


Figure 5.10.1.1: Installing the Freerouting plugin with the PCM.

You will see the Freerouting plugin in the results list. Click on the summary tile ("5") to show the details in the right pane ("6"). Finally, click on the Install button and then "Apply Pending Changes" to install the plugin (I have already installed this plugin, so the button shows "Uninstall").

Notice that the Freerouting plugin requires Java JRE 17. If you don't have the Java JRE installed on your computer, you should do it before you try to use Freerouting. The plugin details contain a link to a download page for the Java JRE installer.

5.10.2. Using Freerouting

Once the plugin is installed, you can use it. Open a PCB in the PCB editor and delete any existing copper traces. In this example, I am using the MCU Dataloger PCB. I deleted all tracks and vias (see below) using the Delete Items tool.

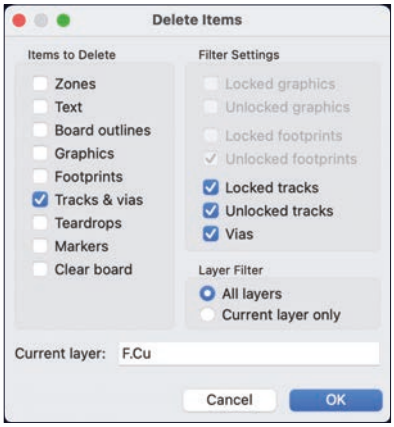


Figure 5.10.2.1: Deleting all tracks and vias.

My PCB now looks like this:

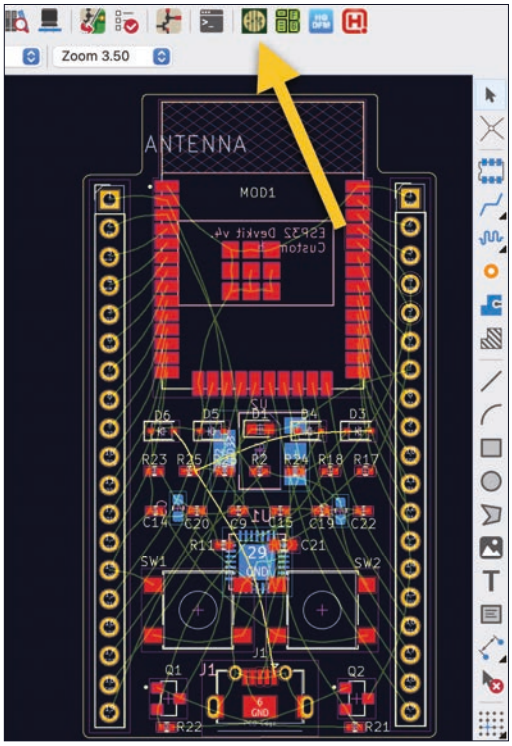


Figure 5.10.2.2: I will auto-route this PCB.

To start the autorouter, click the Freerouting button at the right end of the top toolbar (see the arrow in the screenshot above). You can also start Freerouting from the Tools menu (Tools -> External Plugins -> Freerouting).

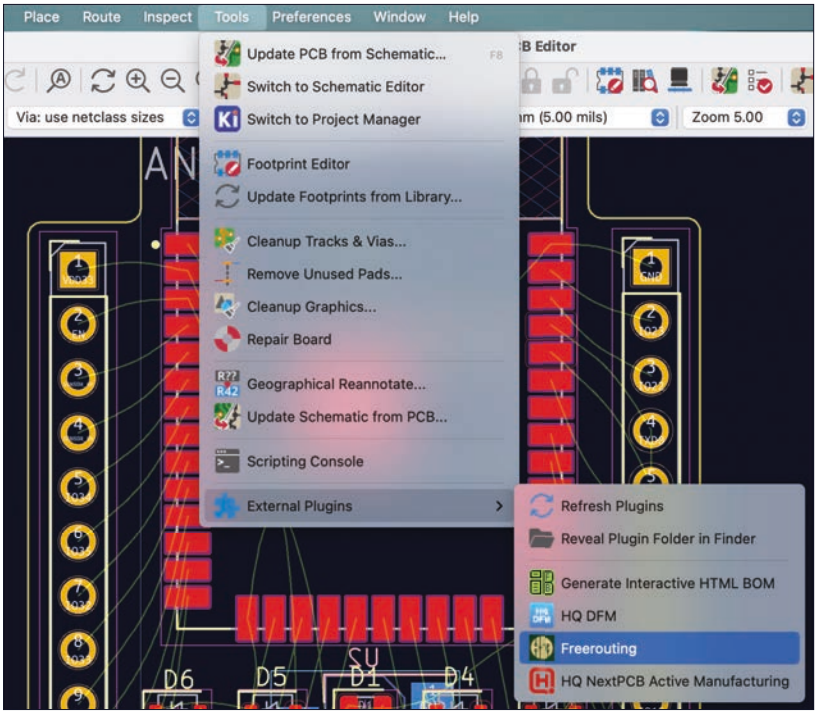


Figure 5.10.2.3: You can start Freerouting from the top menu.

This will start the autorouter and the Freerouting Board Layout window (see below). Depending on the complexity of the PCB, this operation may take several minutes. Freerouting took around three minutes to route my MCU datalogger PCB completely.

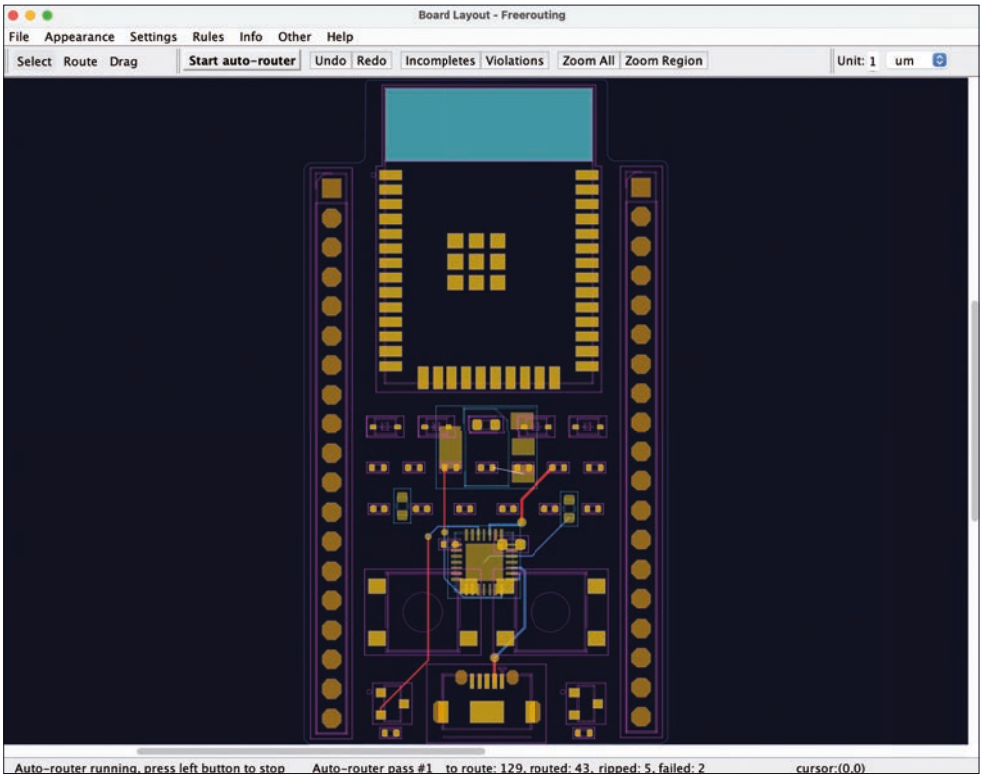


Figure 5.10.2.4: Freerouting working on my PCB.

You can stop the process anytime by closing the Java window or clicking the Terminate button. You don't have to wait for Freerouting to complete all routes. In the example above, two routes have yet to be completed. At this point, I could stop the autorouter and complete the process manually.

When Freerouting finishes, the routed PCB will appear in the layout editor. Here's mine:

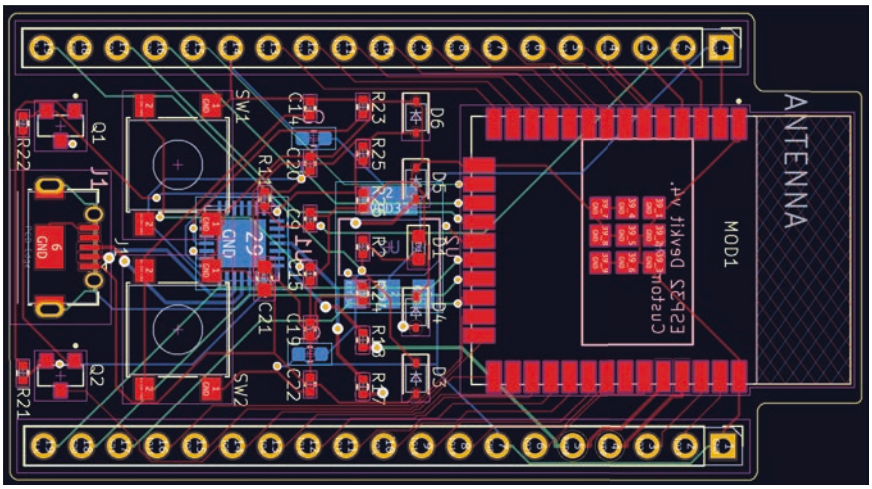


Figure 5.10.2.5: Automatically fully-routed PCB.

The Freerouting plugin offers tight integration between the autorouter and KiCad and is very simple to use. As part of your prototyping process, it can save you many hours of manual trace routing per project.

5.10.3. Autorouting process and customisation

To customize the auto-routing algorithm, use the various options inside the Settings and Rules menus. For example, click on Settings —> General to bring up the General Settings dialog (Figure 5.10.3.1). In the figure below, I also show several of the other configuration windows.

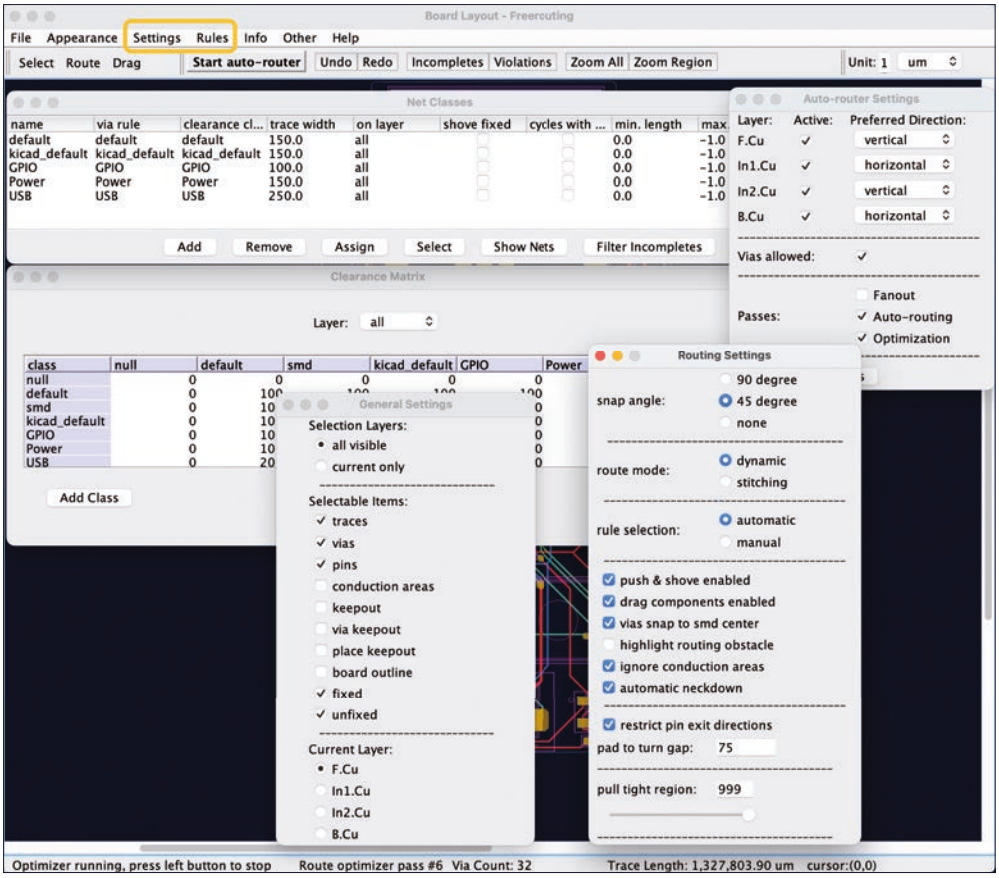


Figure 5.10.3.1: The Autorouter Parameter dialog box.

The General Settings dialogue allows you to select layer visibility during autorouting, selectable items, and the currently enabled layer. Remember, Autorouting, like the KiCad PCB editor, allows you to manually edit the PCB.

The Autoroute Parameter dialogue allows you to choose the preferred drawing direction for the enabled layers and configure the passes mode. Click the Advanced settings button to reveal a secondary dialogue box with further options.

Another example of a way to fine-tune the autorouter algorithm with the Net Classes dialog box (Rules —> Net Classes):

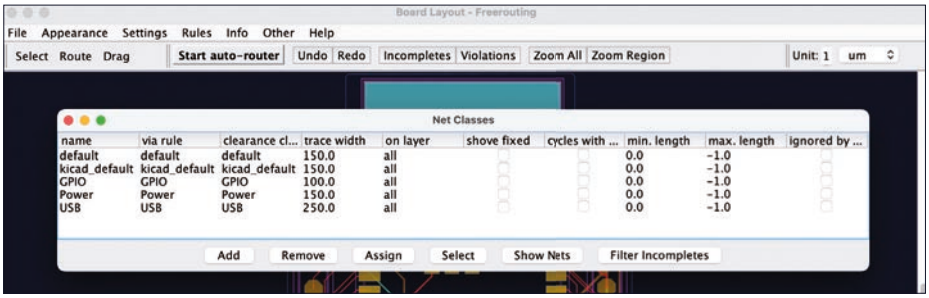


Figure 5.10.3.2: The Net Classes dialog box.

In the Net Classes dialog box, you can see the net classes as set in the KiCad project, imported into Freerouting. You can configure individual properties for each, such as whether tracks that belong to a net class are allowed to shove another track.

In Freerouter, click on the Display menu and the "Layer Visibility." This will reveal a small window that contains four widgets, one for each layer. You can use those widgets to change the visibility of each layer as the autorouter is working. You can keep this window active on the side of the Board Layout window to be available at any time.

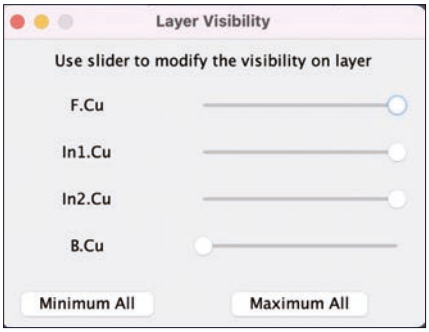


Figure 5.10.3.3: The layer visibility window.

It is worth some time to experiment with the various options and menus. While the autorouter works, you will see new routes appearing and others disappearing and replaced with optimized ones. Also, notice the progress figures in the status bar. In the screenshot below, the autorouter is in its second pass (1), with 73 incomplete routes (2) and 31 complete routes (3):

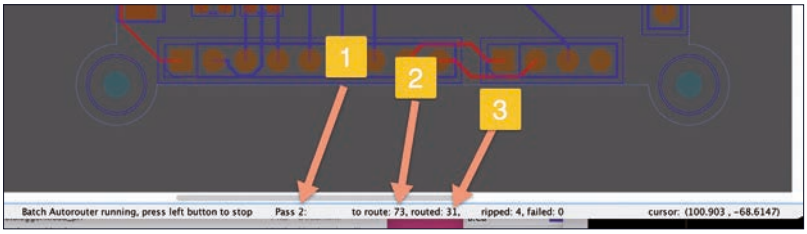


Figure 5.10.3.4: Autorouter progress showing incomplete routes.

A few seconds later, I took another screenshot showing that all routes were complete after 13 passes (1), using 23 vias (2) and with a total trace length of 888.17 mm):

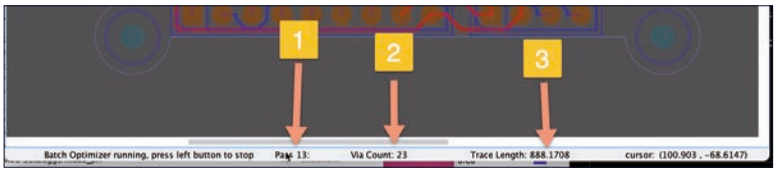


Figure 5.10.3.5: Autorouter progress showing NO incomplete routes.

At this point, the autorouter will continue working, optimizing the routing by reducing the trace length and the number of vias. You can stop it at any time by clicking inside the router window. When the optimizer can no longer improve the routing, it will close, and your autorouted PCB will appear in the main KiCad PCB editor.

Before starting the autorouter, take a few moments to look at the Autorouter Parameter window (where you can select the preferred track direction for each layer) and the Clearance Matrix window (under Rule), where you can set minimum clearances for each class and layer.

5.11. PCB editor Inspection menu

In this chapter, you will learn about the functionalities available under the PCB editor's "Inspect" menu (Figure 5.11.1).

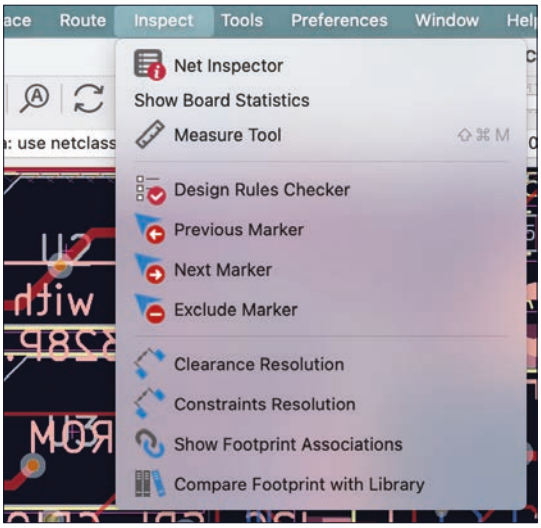


Figure 5.11.1: The "Inspect" menu in the PCB editor.

We'll begin with the Net Inspector.

Net Inspector

Start the Net Inspector by clicking on Inspect —> Net Inspector. The Net Inspector window will appear, and it looks like this (Figure 5.11.2):

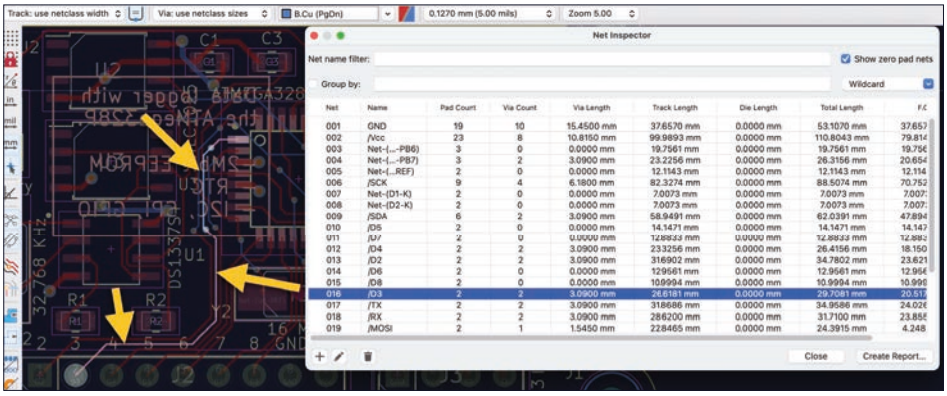


Figure 5.11.2: The Net Inspector window with a net selected and its members highlighted in the editor.

The Net Inspector window contains a list of the nets in the layout and statistics such as the total number of pads, vias, and length of each net. You can click on a net row; the editor will highlight the net members in the layout. You can use the widgets at the top of the window to filter and group the nets that the inspector shows in the list. For example, type "/D" in the name filter text box, and the inspector will only show nets with names that start with "/D."

At the bottom right corner of the Inspector window is the "Create Report..." button. Click on this button and save the resulting CSV file on your computer. Below you can see an example of this report (Figure 5.11.3):

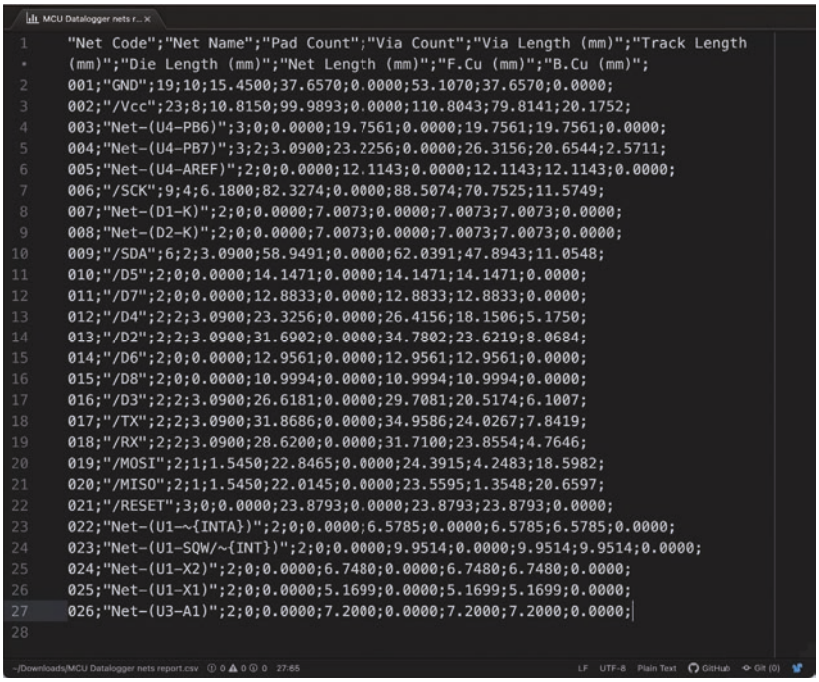


Figure 5.11.3: The Net Inspector exported report.

The report contains the same information you see in the inspector window in text CSV format, which you can use with spreadsheets for further processing.

Board Statistics

Start the Board Statistics by clicking on Inspect —> Show Board Statistics. The Board Statistics window will appear, and it looks like this (Figure 5.11.4):

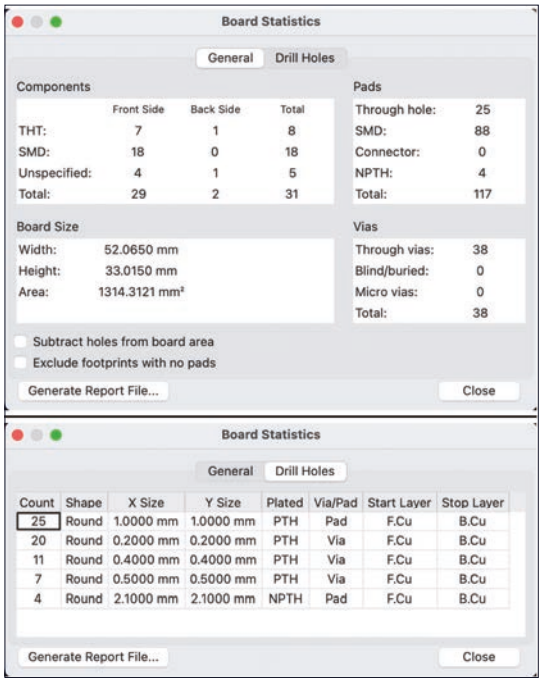
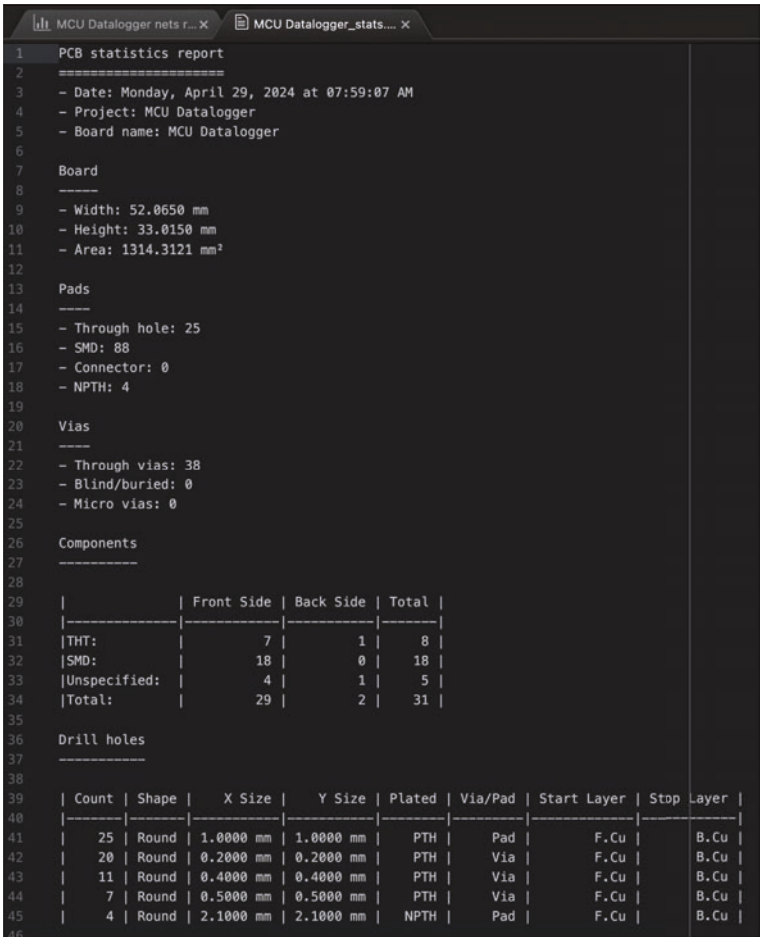


Figure 5.11.4: Board Statistics. General (top), Drill Holes (bottom).

The Board Statistics window contains two tabs. The General tab provides information about the board's components, pads, vias, and dimensions. The Drill Holes tab offers information about the drills, how many are present in the PCB, their dimensions, and start and stop layers.

You can generate a text report that contains all of this information, which looks like this (showing a segment only):



MCU Datalogger nets r... × MCU Datalogger_stats.... ×

```
PCB statistics report
=====================
- Date: Monday, April 29, 2024 at 07:59:07 AM
- Project: MCU Datalogger
- Board name: MCU Datalogger

Board
-----
- Width: 52.0650 mm
- Height: 33.0150 mm
- Area: 1314.3121 mm²

Pads
----
- Through hole: 25
- SMD: 88
- Connector: 0
- NPTH: 4

Vias
----
- Through vias: 38
- Blind/buried: 0
- Micro vias: 0

Components
----------

|              | Front Side | Back Side | Total |
|--------------|------------|-----------|-------|
|THT:          |          7 |         1 |     8 |
|SMD:          |         18 |         0 |    18 |
|Unspecified:  |          4 |         1 |     5 |
|Total:        |         29 |         2 |    31 |

Drill holes
-----------

| Count | Shape |    X Size |    Y Size | Plated | Via/Pad | Start Layer | Stop Layer |
|-------|-------|-----------|-----------|--------|---------|-------------|------------|
|    25 | Round | 1.0000 mm | 1.0000 mm |    PTH |     Pad |        F.Cu |       B.Cu |
|    20 | Round | 0.2000 mm | 0.2000 mm |    PTH |     Via |        F.Cu |       B.Cu |
|    11 | Round | 0.4000 mm | 0.4000 mm |    PTH |     Via |        F.Cu |       B.Cu |
|     7 | Round | 0.5000 mm | 0.5000 mm |    PTH |     Via |        F.Cu |       B.Cu |
|     4 | Round | 2.1000 mm | 2.1000 mm |   NPTH |     Pad |        F.Cu |       B.Cu |
```

Figure 5.11.5: The board statistics text report.

Measure tool

You can invoke the Measure tool from the Inspect menu ("Measure tool"), or the right toolbar:

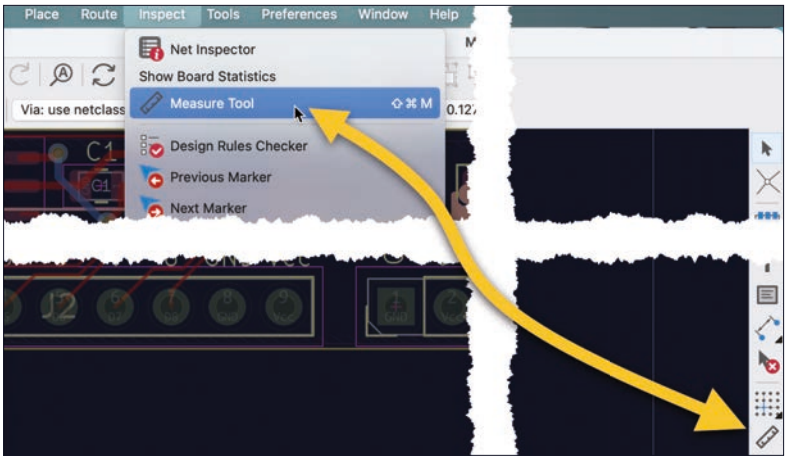


Figure 5.11.6: The Measure tool.

You can use the Measure Tool to make quick distance measurements between any two points in your PCB. Enable the tool, and then click on the layout editor to start measuring, then click again to finish:

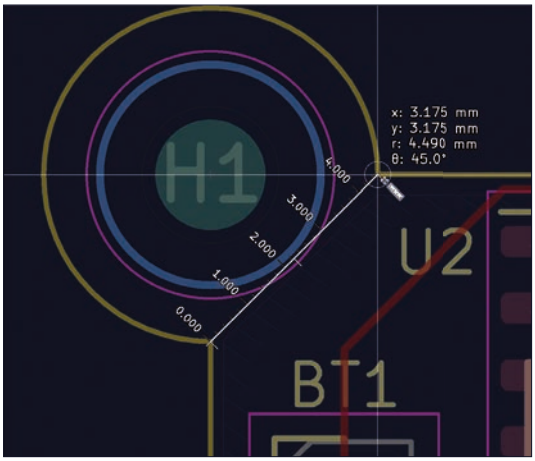


Figure 5.11.7: The Measure tool in action.

Design Rules Checker & Markers

I discuss the Design Rules Checker in detail in dedicated chapters. The Inspect menu includes three related options:

- Previous Marker.
- Next Marker.
- Exclude Marker.

The options allow you to navigate through the results of the DRC by clicking on the menu items or typing the set hotkey. For example, suppose that the DRC contains these results:

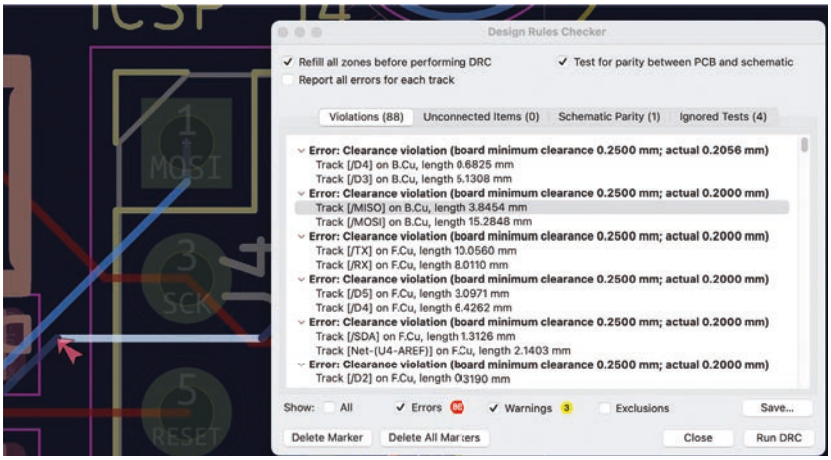


Figure 5.11.8: The DRC results. Click on a row to highlight the source.

You can navigate through the DRC rows by:

1. Clicking on a row.
2. Click on the marker options inside the Inspect menu.
3. Typing the corresponding hotkeys (for my setup, Option-Shift-Left for Previous Marker, and Option-Shift-Right for Next Marker).

You can assign custom hotkeys in the Preferences window, Hotkeys.

Clearance Resolution

This option will generate a report about the clearance resolution between two selected items on the board. For example, in my test board below, I have selected pin 1 of C1 and an adjacent via (hold the Command or Control key down, then click on the items you can choose). Then, click Inspect and "Clearance Resolution":

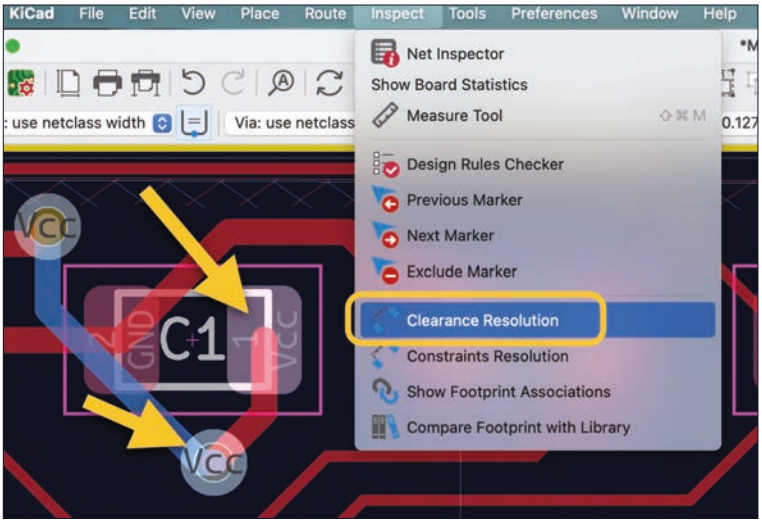


Figure 5.11.9: Select two items from the board, then click Clearance Resolution.

The PCB editor will open up a new window with the report. The report provides detailed clearance information in three tabs.

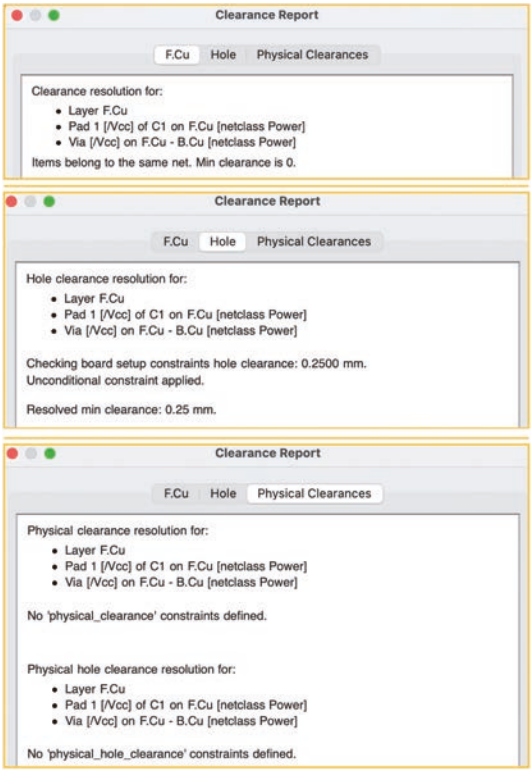


Figure 5.11.10: The clearance resolution report for the two capacitors.

This report will provide a wealth of information about the selected items. For example, it will tell you about their distance, highlight any constraint violations, and whether they belong to the same net. The tabs included in the Clearance Report window depend on your selections. I encourage you to experiment with this tool by trying out various combinations. Choose pairs containing tracks, vias, pads (TH and SMD), and silkscreens, and familiarise yourself with the information this tool returns.

Constraints Resolution

Select one item from the board to use the Constraints Resolution tool, then click on Inspect and Constraints Resolution. In the example below, I have selected the MCU footprint:

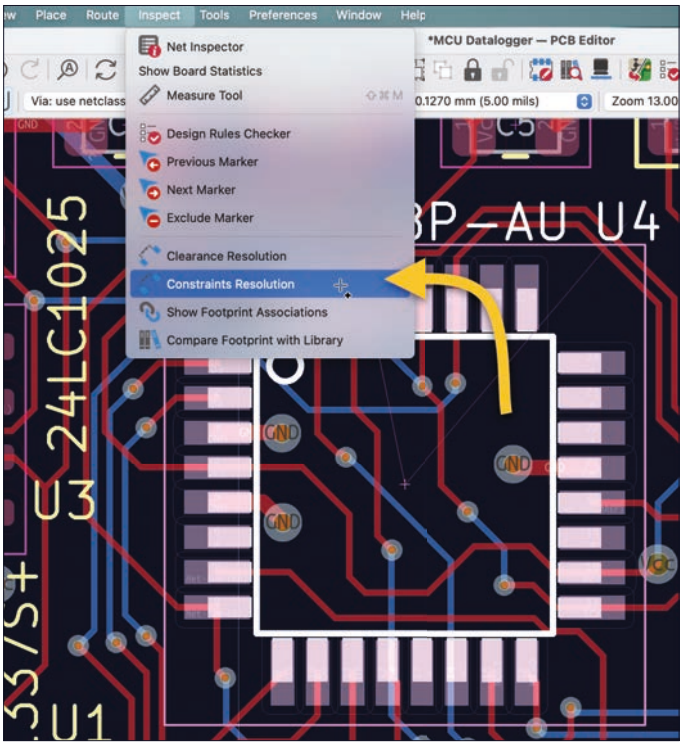


Figure 5.11.11: Select one item from the board, then click Constraints Resolution.

The PCB editor will open up a new window with the report:

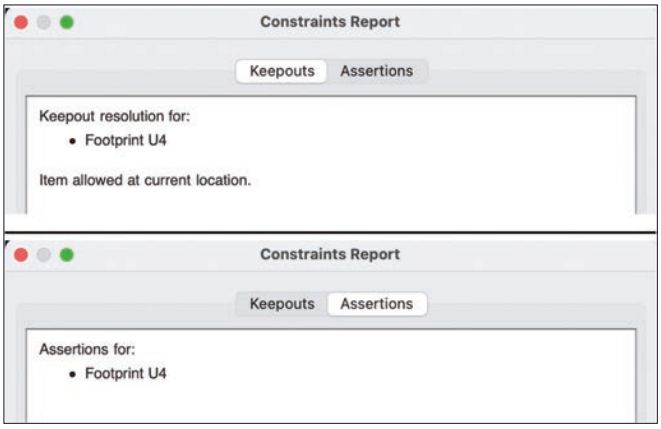


Figure 5.11.12: The Constraints report for U4.

The report provides information about vias, keep-outs, location, and perhaps other constraints that I have not noticed until now. The report may indicate that certain constraints are unmet (such as the keep-out constraint in the example above). The report may contain cases of possible violations that may or may not cause problems in your layout, and it supplements but does not substitute the Design Rules Check.

5.12. Single track and differential pair routing

The PCB editor offers two routing modes: single-track and differential pair-tracking. You can select the mode via the buttons in the right toolbar or the top menu (Figure 5.12.1).

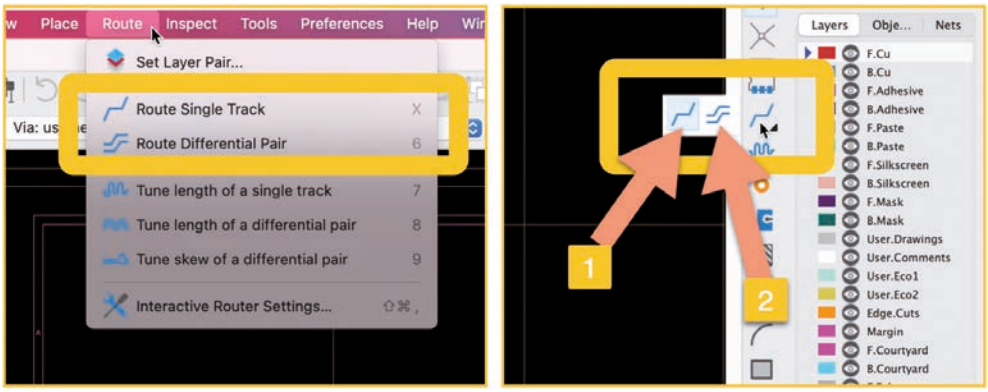


Figure 5.12.1: The two routing modes and how to enable them.

To choose a routing mode from the right toolbar, click and hold on the routing icon to expand it, then click one of the two options. By default, the single-track routing mode is selected. You probably have already used this mode, as I have extensively used it throughout this book. In this chapter, I will formalize this knowledge and show you how to do differential pair routing.

I will begin this demonstration in the schematic editor and design a simple schematic that consists of two 9-pin headers. I have used netlabels that contain a single letter (like "A" or "B") or a letter followed by "+" and "-, "or "_P" and "_N." Here is my demo schematic (Figure 5.12.2):

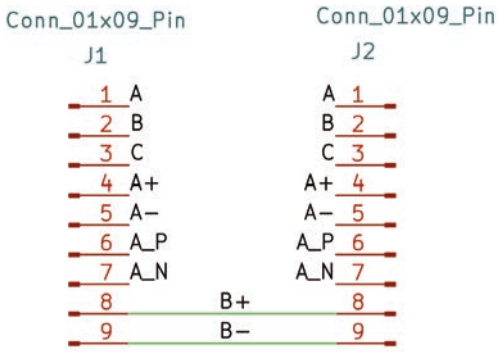


Figure 5.12.2: This schematic contains single nets and differential pair nets.

In the schematic above, I have used wires to connect pins 8 and 9 of J1 and J2. I have made the rest of the connections via the net labels.

A net label that ends with "+" and "-" or "_P" and "_N" represent differential pairs. In a differential pair, information is transmitted between the transmitter and the receiver using signals that travel through two wires. The signal levels (i.e., their voltage) are the same, but their polarity is the opposite. Examples of technologies that utilize differential pairs include DDR SDRAM, PCI Express, and Serial ATA.

In KiCad, we can identify differential pair pins using the "+"/"-" or "_P"/"_N" notation. For example, a pin labeled "data+" and "data1-"consists of a differential pair. Pins labeled "data1_P" and "data1_N" also have a differential pair. You can use either postfix but do not mix them. For example, "data1+" and "data1_N" do not consist of a differential pair.

Let's continue with this example. Import the schematic to the PCB editor so that you have this un-routed layout (Figure 5.12.3):

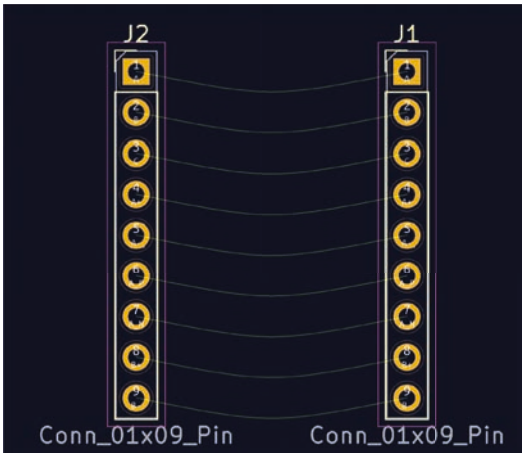


Figure 5.12.3: The un-routed layout.

Notice that the net names are inherited from the schematic in the schematic editor and appear on the pads. This information will help distinguish between single tracks and differential pairs.

For pads "A", "B", and "C", we'll use single track routing since they don't have the "+"/"-" or "_P"/"_N" postfix are single tracks. Click on the single-track routing mode button from the right toolbar (or select "Route Single Track" from the "Route" menu, or type the "X" hotkey on your keyboard). With the single-routing mode selected, click on a pad to start drawing, then click again on the closing matching pad to finish drawing (Figure 5.12.4).

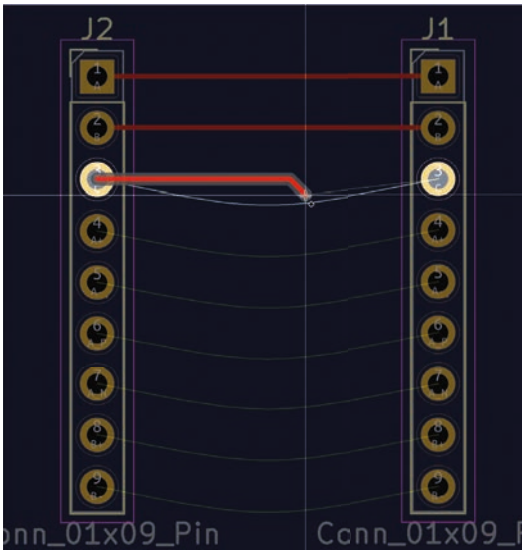


Figure 5.12.4: Single-track drawing.

Right-clicking can create angled track segments. To complete the track, click on the corresponding pad. The ratnest lines can help you find and see the matching pad by linking it with the line and highlighting it to make it stand out.

Let's continue with the differential pars. I'll start with the pair "A+" and "A-". You can route a differential pair as if it were a single pair. Try this out and confirm that it works. However, a better way to route differential pairs is to select the differential pair routing mode. Select the differential pair routing mode by clicking on the button with the two lines from the right menu bar (see "2" in Figure 5.12.1), or click on "Route Differential Pair" under the Route top menu, or just type the "6" hotkey on your keyboard.

With the differential pair routing tool selected, start by clicking on either of the two pads that make up the pair to start drawing ("1" in Figure 5.12.5 below). Continue to draw the two tracks as you navigate the pair closer to the target pads ("2", below). You can click to change directions as if drawing a single track. To finish drawing, click on either pair's target pads ("3", below).

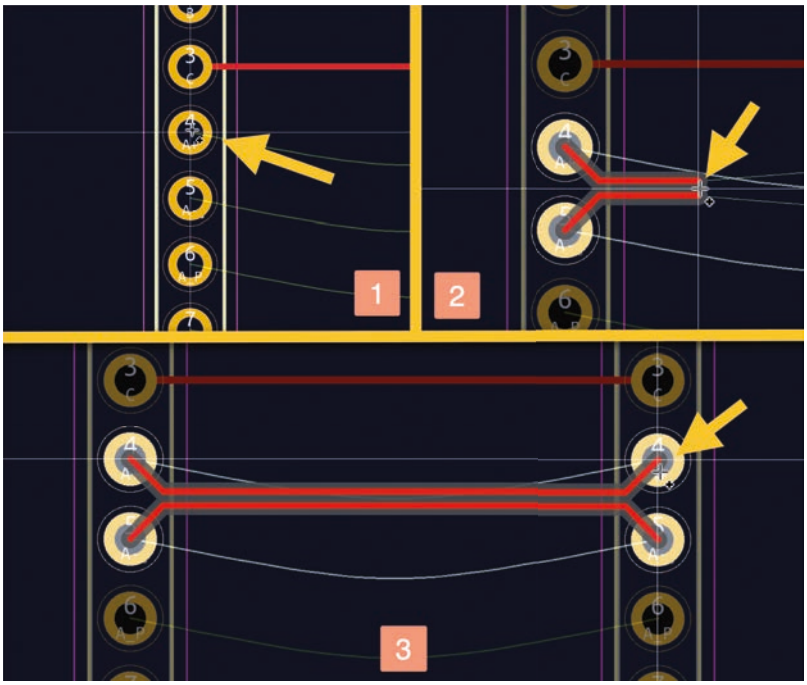


Figure 5.12.5: Drawing differential pair tracks.

Your differential pair tracks are complete. Drawing a differential pair track is similar to drawing a single track in elements, such as adding segments to make the track go around other elements and vias. You can see examples of this in Figure 5.12.6 below. The pairs "A_P"/"A_N" and "B+"/"B-" contain multiple segments and vias. The easiest way to create a via is to use the "V" hotkey while you draw.

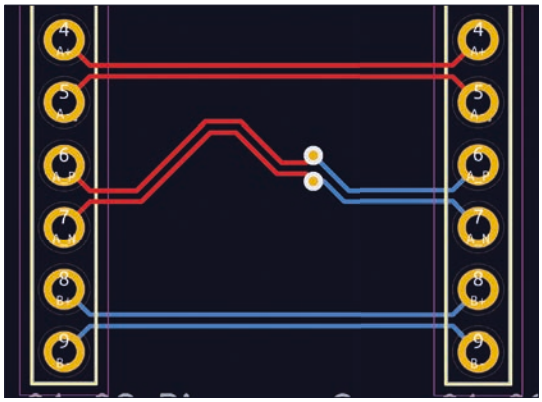


Figure 5.12.6: Differential pair tracks with multiple segments and vias.

5.13. Track length tuning

In this chapter, you will learn how to tune the length of a single track or a differential pair. Track length tuning is a function that allows you to tune the length of a track or differential pair so that it is an exact value. By tuning the length of a track, you can ensure that a signal can propagate from origin to destination in a specific amount of time. Track length tuning is important for applications such as in-memory data or address buses, where signals that belong to the same bus must arrive at the destination within specific timeframes.

The two tools you will learn about in this chapter can increase the total length by introducing new and repeated segments (meanders). You can set the target length; the tool will draw enough new meanders to achieve that goal.

You can access the single track length tuning and differential pair length tuning tools via the top menu, under Route, or via the right-side toolbar (Figure 5.13.1):

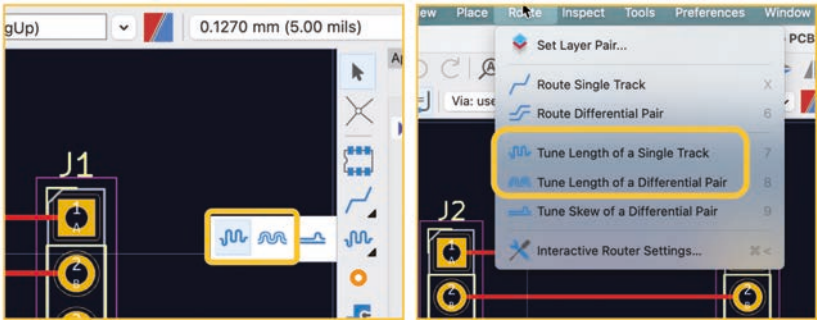


Figure 5.13.1: The two-track length tuning tools.

Let's look at an example of how length tuning works. We'll use the layout from the previous chapter, which looks like this (only showing the top two tracks):

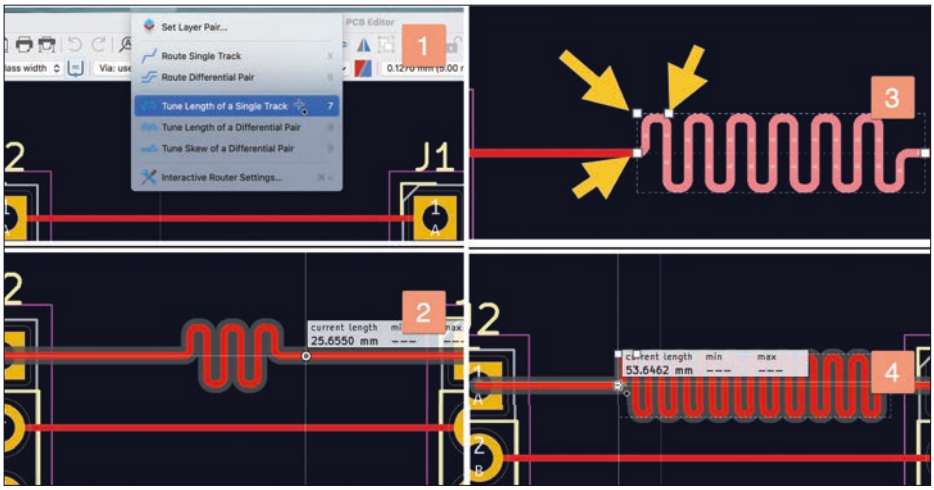


Figure 5.13.2: Fine-tuning the length of a single track.

In Figure 5.13.2 (above), I will increase the length of the track that connects pad 1 of J1 and pad 1 of J2. Start by clicking on "Tune length of a single track" from the Route menu ("1"). I prefer the hotkey for this tool, which is "7". Click anywhere on the track, preferably towards the left or right end, so you have enough space to draw the new segments. I'm starting towards the left end of the track ("2"). Start moving the mouse towards the right end of the track (or whichever way most of the track lies), and notice the new segments have a meander shape. When the label that shows the current and target length becomes green, click again to finish the drawing ("3"). Notice that three handles appear when you finish the operation ("4"). Use these handles to trim the track length with precision further. These handles will appear when you select the meandering segment of the track. The length of the track is now tuned.

It is possible to configure how the length tuner tool works, including the target length. To do this, enable the single (or differential pair) length tuner from the menus or using the hotkeys ("7" for the single length tuner, and "8" for the differential pair length tuner), and click on the track to start the length tuner. Then, double-click (still on the track) to bring up the single or differential pair length tuning window.

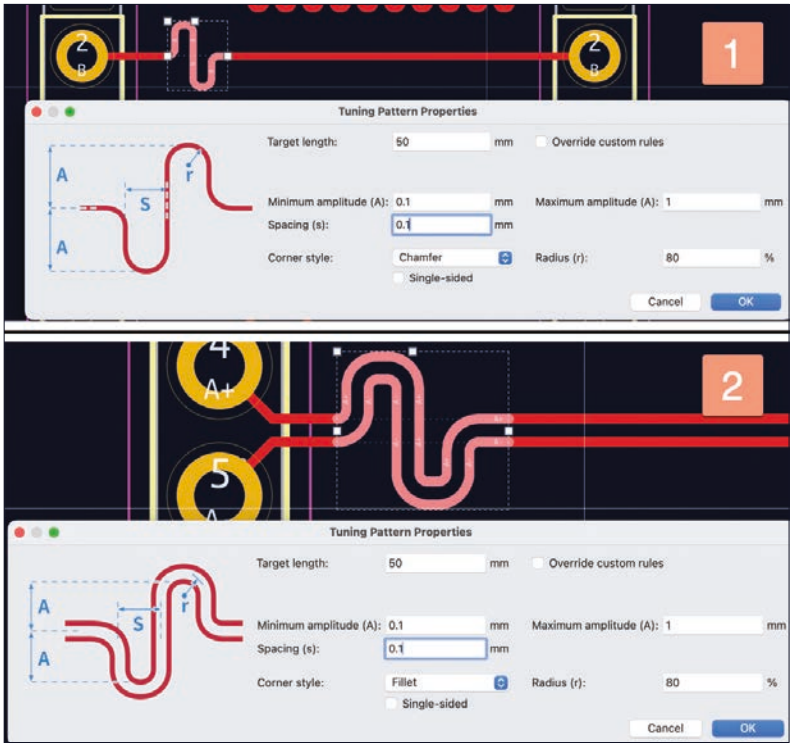


Figure 5.13.3: Length tuning settings for single track and differential pair.

The length tuning settings for single track and differential pair are very similar. You can control the minimum and maximum amplitude, the spacing between the adjoining segments of a meander, the style (arc or 45 degrees), and the radius. Below, you can see the difference between the fillet and chamfer styles:

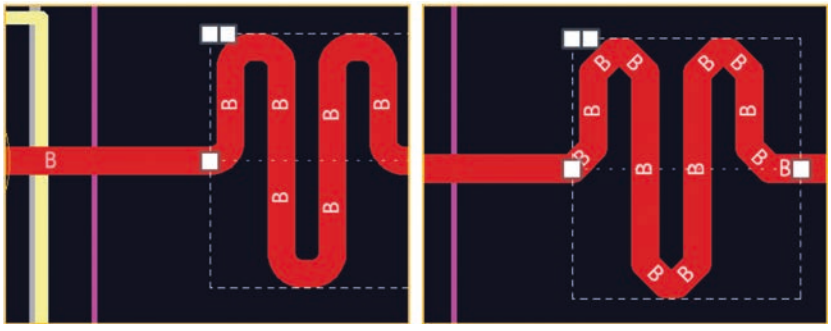


Figure 5.13.4: Meander styles. fillet (left) and chamfer (right).

You can tune the length of a differential pair by selecting the tool "Tune length of a differential pair" from the Route menu or typing the hotkey "8". Use the same drawing process as with the single track. A length-tuned differential pair looks like this (Figure 5.13.5):

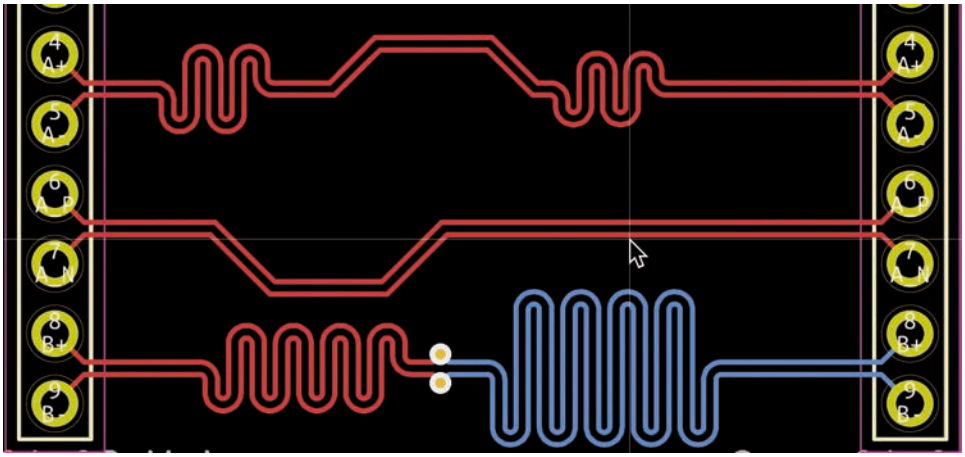


Figure 5.13.5: Length-tuned differential pairs.

In the examples above, I have created two separate meandering segments in the top one. The bottom differential pair includes two vias in the middle. The length tuning tool cannot continue drawing past a via or pad, so I had to stop the drawing and create a new meandering segment on the other side to achieve the required length goal.

It is also possible to change the spacing and amplitude of the meander during drawing. You can do this using the hotkeys 1, 2, 3, 4 (active only during length-tuning drawing) or via the context menu (Figure 5.13.6).

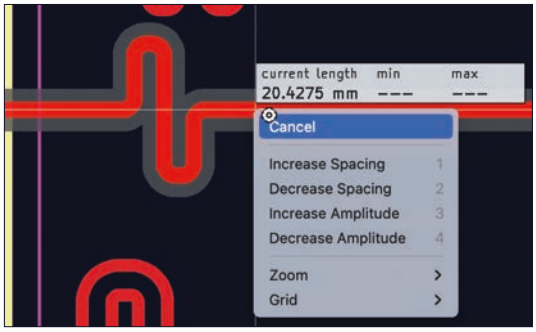


Figure 5.13.6: Space and amplitude control in the context menu.

By increasing or decreasing the amplitude and spacing of the meander, you can change the total footprint on the PCB required by the meander. Of course, as the amplitude gets bigger, the track will need more space around it.

The Route menu contains one more tuner, which you can use to tune the skew of a differential pair. You can learn more about this in the next chapter.

5.14. Differential pair skew tuning

In the previous chapter you learned how to tune the length of a single track or a differential pair. In this chapter, I'll show you how to use another tool in the Route menu, "Tune skew of a differential pair."

Differential skew[13] is a phenomenon that appears in high-speed, high-frequency applications. The phenomenon is the time difference between the two signals in a differential pair. In high-speed applications, a tiny difference in the time it takes for the two differential pair signals to travel to their destination can cause significant processing problems. Such a system would need to detect and recover data losses caused by the differential skew. If we can minimize the differential skew as much as possible, the system would spend less time recovering lost or corrupt data, and its overall performance would increase.

A common cause of differential skew is the difference in length of the two tracks that make up the pair. The differential pair skew tuner allows you to tune one pair of tracks independently of the other to minimize or eliminate the differential skew caused by the track length difference.

To use the tool, choose "Tune skew of a differential pair" from the Route menu, or type the "9" hotkey, or select it from the right toolbar (Figure 5.14.1):

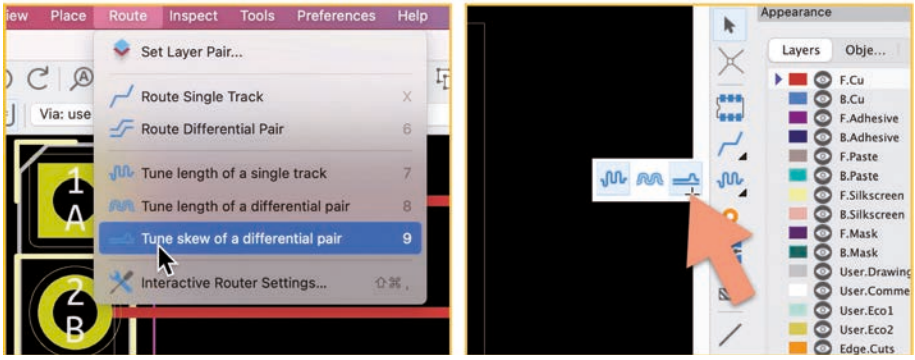


Figure 5.14.1: The differential pair skew tuner tool.

Let's look at an example of how to use the differential pair skew tuner tool. My starting point is a simple straight differential pair track, as you can see below:

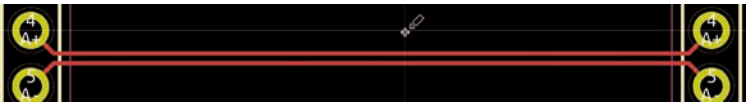


Figure 5.14.2: We'll tune the skew of this differential pair.

Imagine that the signal that flows through the top track ("A+" net) arrives at its destination slightly sooner than the bottom track. Maybe the material used is different, or perhaps the geometry of the track for track of net "A-" is such that it causes a delay. Whatever the

13 For a discussion on differential skew, see https://www.edn.com/handling-differential-skew-in-high-speed-serial-buses/

reason, I would like to equalize the signal's propagation time by introducing a slight delay in the top track.

First, I will find out how long each of the tracks is that make up this differential pair. You can use the Net Inspector for this under the Inspect menu. Invoke the Net Inspector and click on the "A+" and "A—" nets to see their rows.

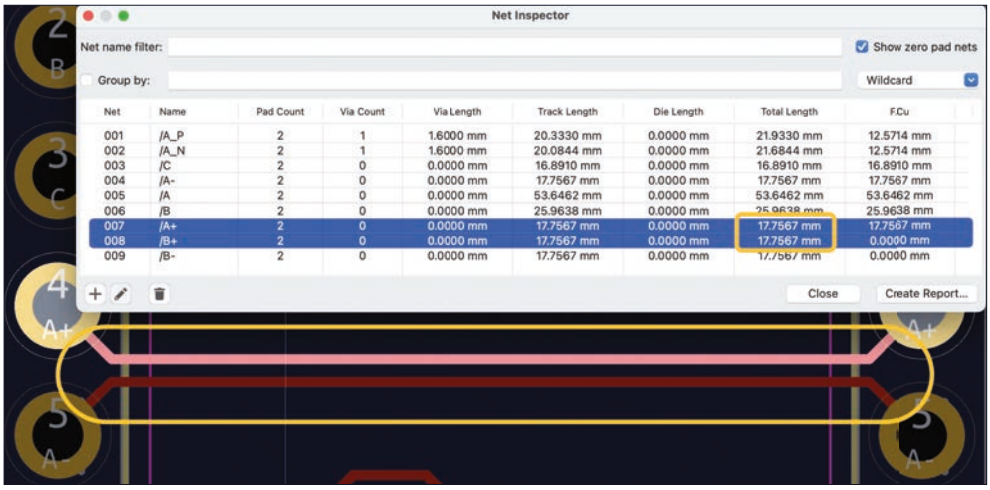


Figure 5.14.3: The current length of the two tracks in the example differential pair.

The two tracks in the example differential pair have an equal length of 17.7567 mm. I will tune the length for the "A+" track so that it has an additional 25 mm.

Select the "Tune skew of a differential pair" tool from the Route menu (I prefer to type the "9" hotkey). Then, click on the track for the A+ net to select it with the tune skew tool, and double-click to open the tuner settings window. The window that appears looks like this:

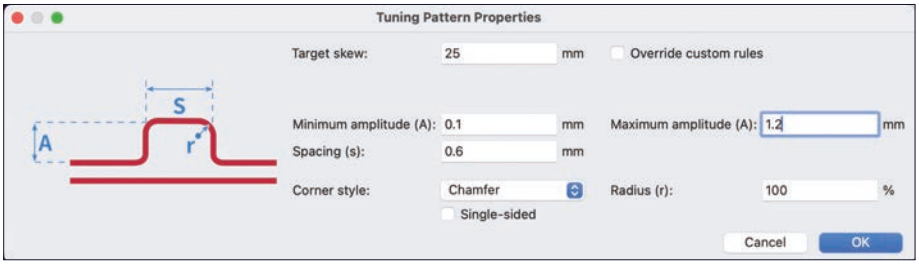


Figure 5.14.4: The Differential Pair Skew Tuning window.

Change the Target skew to 25 mm (the additional track length you want to add to the existing track). The screenshot above shows the rest of my settings. Click OK.

Three handles will appear on the track. You can use these handles to create new track segments so that the total track length eventually reaches the 25 mm target. As you use the handles to draw additional track segments, observe the Routed Length figure in the status bar (see screenshot below). This figure will show you the current length of the track. In this example, I used all available horizontal space but was unable to reach my target of 25 mm.

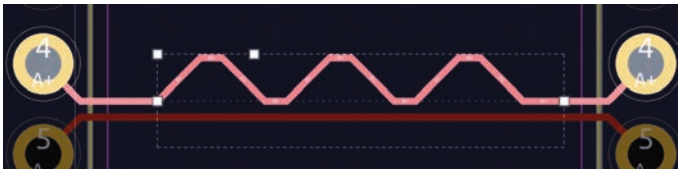


Figure 5.14.5: I have increased the track length but am still short of my 25 mm target.

Several options are available to help me reach my target. Double-click on the new track segments to show the tuner properties window. Change the maximum amplitude to 1.5 mm and spacing to 0.5 mm. Click OK, and continue to draw the additional track segments. You cannot add new track segments when the length reaches 25 mm. The information bubble next to the cursor will confirm that the length target has been reached. The label will become green, indicating that I have attained the set target.

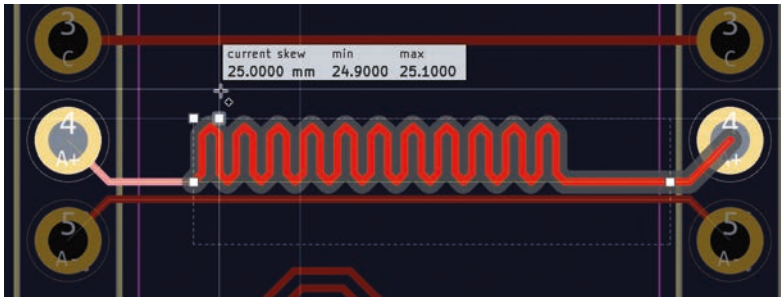


Figure 5.14.6: Increasing the length of one of the tracks in a differential pair.

The differential pair track now has a slightly different length. The track that belongs to the "A+" net is somewhat longer than the track of "A-. "

5.15. The footprint wizard

KiCad ships with an extensive library of footprints. If you need a footprint not available in those libraries, you can find it in repositories like Snapeda. But even if that fails, you can create custom footprints using one of KiCad's tools: the Footprint Editor and the Footprint Wizard. You can learn how to use the footprint editor in a dedicated chapter in My book Fundamentals and Projects.

In this chapter, you will learn how to use the Footprint Wizard. With the footprint wizard, you can quickly and easily create a custom footprint for a standard set of components. The wizard is part of the footprint editor. To use the wizard, start the footprint editor from the main KiCad project window (see frame 1 in Figure 5.15.1).

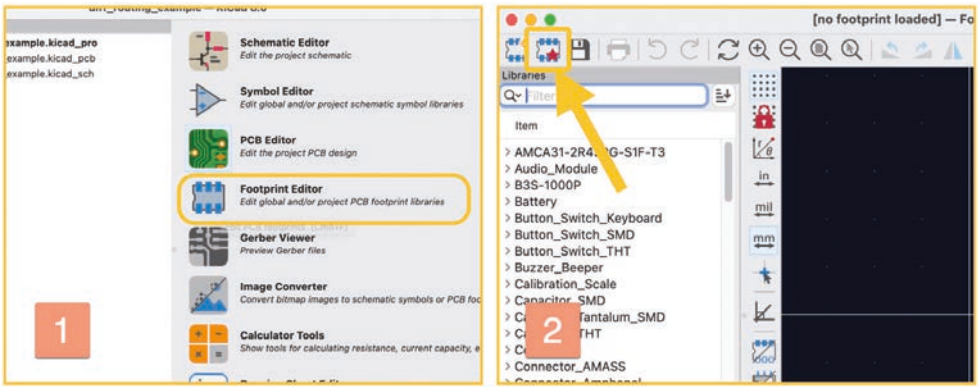


Figure 5.15.1: Starting the footprint wizard.

Once in the footprint editor, click the Wizard button in the top menu (2). The wizard will take you through a series of steps. In the first step, you will select the footprint generator that is most appropriate for the type of footprint you want to create. Generators are available for BGA, QFN, DIP, SOIC, and many more types of components.

Create a DIP footprint

I will create a new footprint for this demonstration for a DIP component. From the list of available generators, I will choose row 8, "S-DIP" (Figure 5.15.2), and click OK.

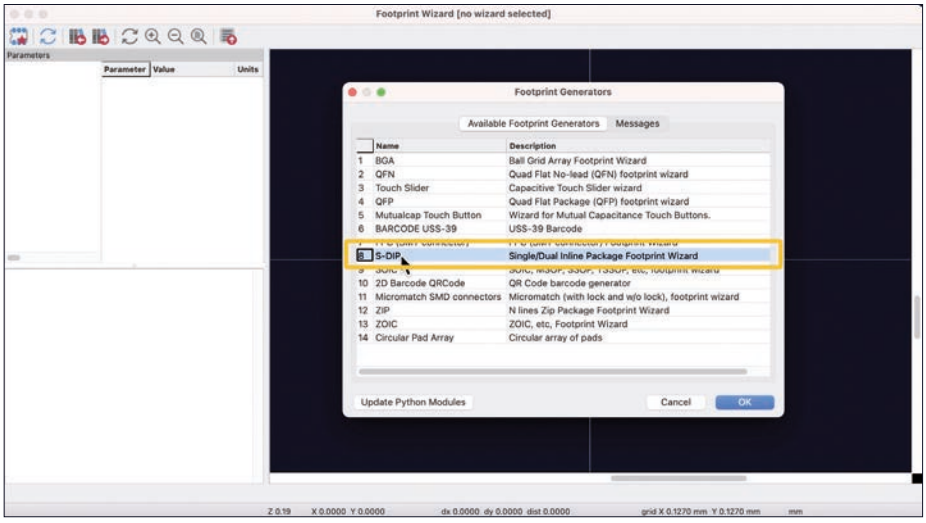


Figure 5.15.2: The footprint wizard, footprint generators window.

Earlier in this book, I used the footprint editor to create a DIP footprint manually. This is an opportunity to experience how the wizard can speed up the work for standardized footprints.

When you dismiss the footprint generator window, the footprint wizard will display its default settings in the right design pane. On the left side is a list of parameters that you can edit and customize the footprint (Figure 5.15.3).

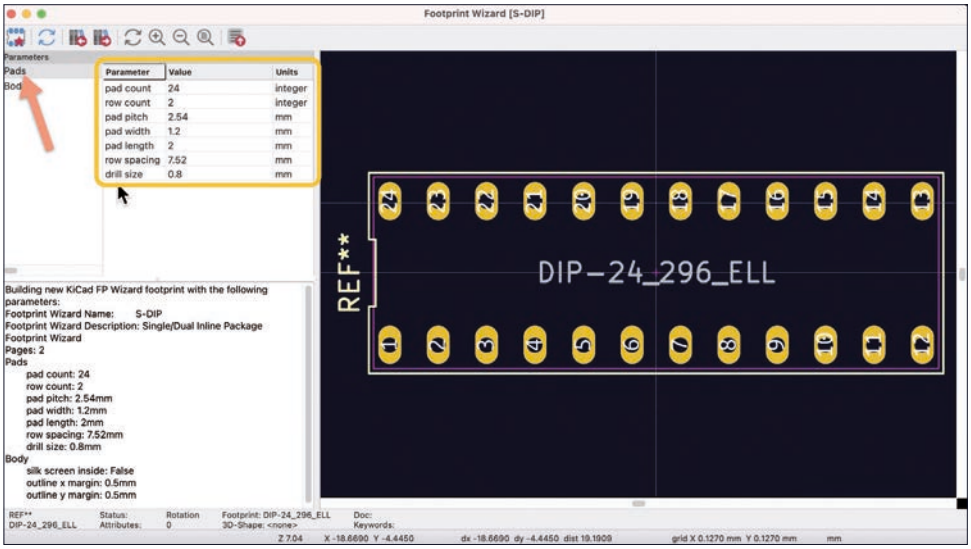


Figure 5.15.3: The Footprint Wizard with the default Pad settings from the DIP generator.

As you change the settings in the parameters pane, the wizard updates the footprint in the right pane. I made these changes:

- Pad count to 32.
- Pad pitch to 1.54 mm.

In the left pane, click on Body. This will show body parameters in the middle pane (Figure 5.15.4):

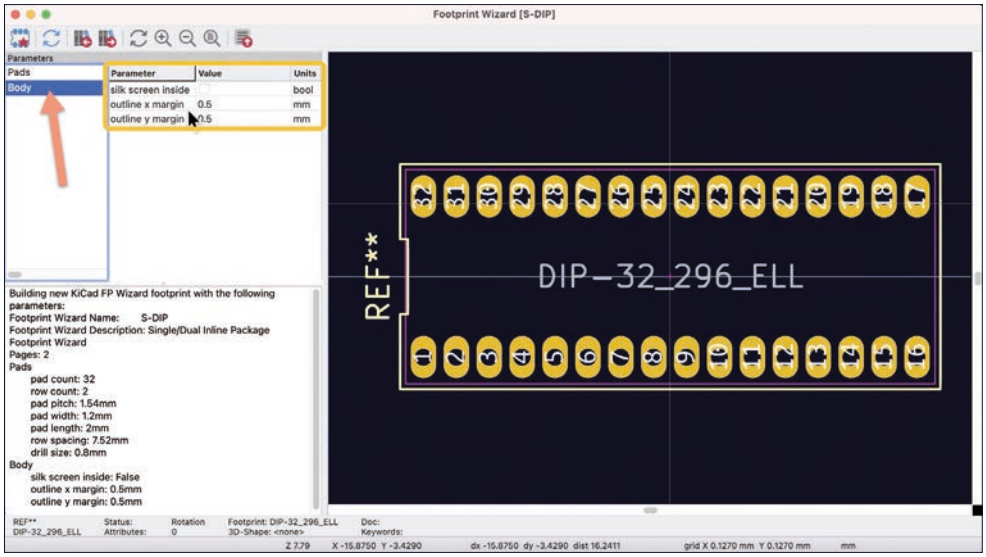


Figure 5.15.4: The Body parameters in the footprints wizard.

I have made a couple of changes in the body parameters:

- Outline x margin to 1.5 mm.
- Outline y margin to 1.5 mm.

Below, you can see my new footprint. It took me around 60 seconds to design:

Figure 5.15.5: Done in 60 seconds.

When you finish work on your new footprint, you can export it from the wizard to the footprint editor by clicking on the import button:

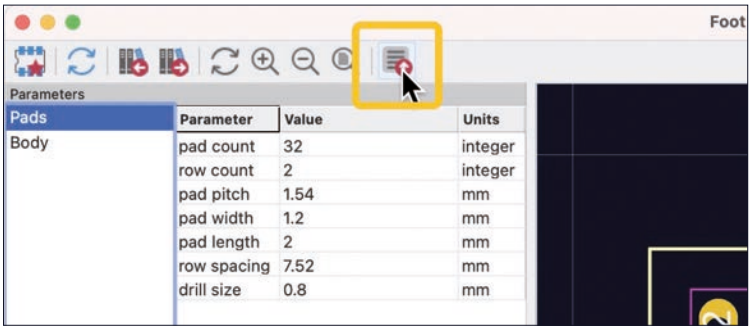


Figure 5.15.6: Import the footprint into the footprint editor.

In the footprint editor, you can continue to customize your new footprint or save it into an existing or new library so you can use it in a project. If you don't know how to do this or need a refresher, see Create a custom footprint, 4, Silkscreen layer from earlier in this book.

Create a Barcode

The footprint wizard contains a 2D Barcode and QRCode generator. You can use this generator to create barcode graphics that encode information, such as a webpage URL with information about your board so that the end-user does not have to type the URL.

I'll show you how that works.

Start the footprint wizard, select the "2D Barcode QRCode" row, and click OK. The wizard will show you this default QR code:

Figure 5.15.7: The default QR code in the footprint wizard.

In the Barcode parameters, I have changed the contents field to "techexplorations.com." I changed the height and width to 1.5 mm in the Caption parameters. Continue as you would with any other wizard-generated footprint:

1. Export the footprint to the footprint wizard.
2. Save the new footprint to a new or existing footprint library.
3. Use the new footprint in your project.

Below, you can see my new QR code footprint on a PCB in the 3D viewer:

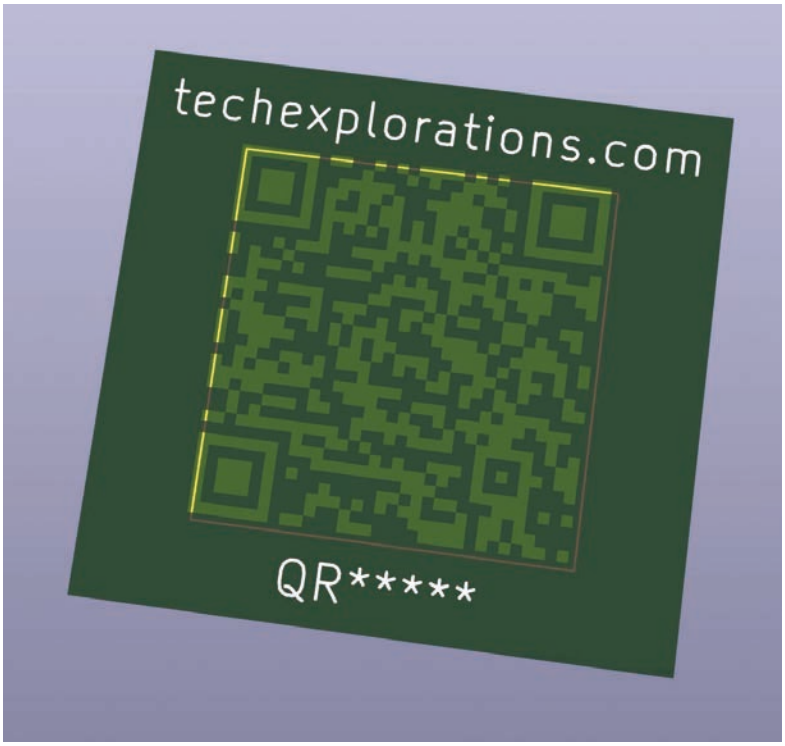


Figure 5.15.8: A QR code footprint generated by the footprint wizard.

5.16. Pin and wire highlighter tool

It is often difficult to see which pins are connected to other pins in busy schematics and quickly determine which wires belong to the same net. To help in such situations, KiCad provides a tool called "Highlight wires and pins of a net." You can find it on the top of the right toolbar in the schematic editor:

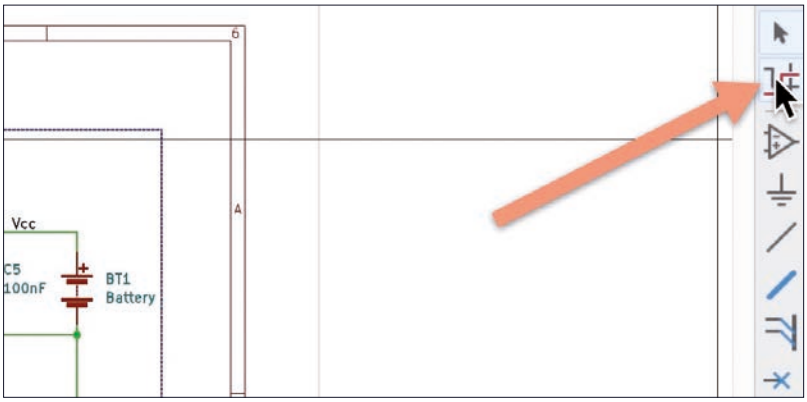


Figure 5.16.1: The "Highlight wires and pins of a net" tool.

I'll show you how it works.

Click on the highlight button in the right toolbar to enable the tool. In the example below, I have clicked on one of the wires or pins that belong to the GND net. The highlighter marked all GND net member wires and pins in pink:

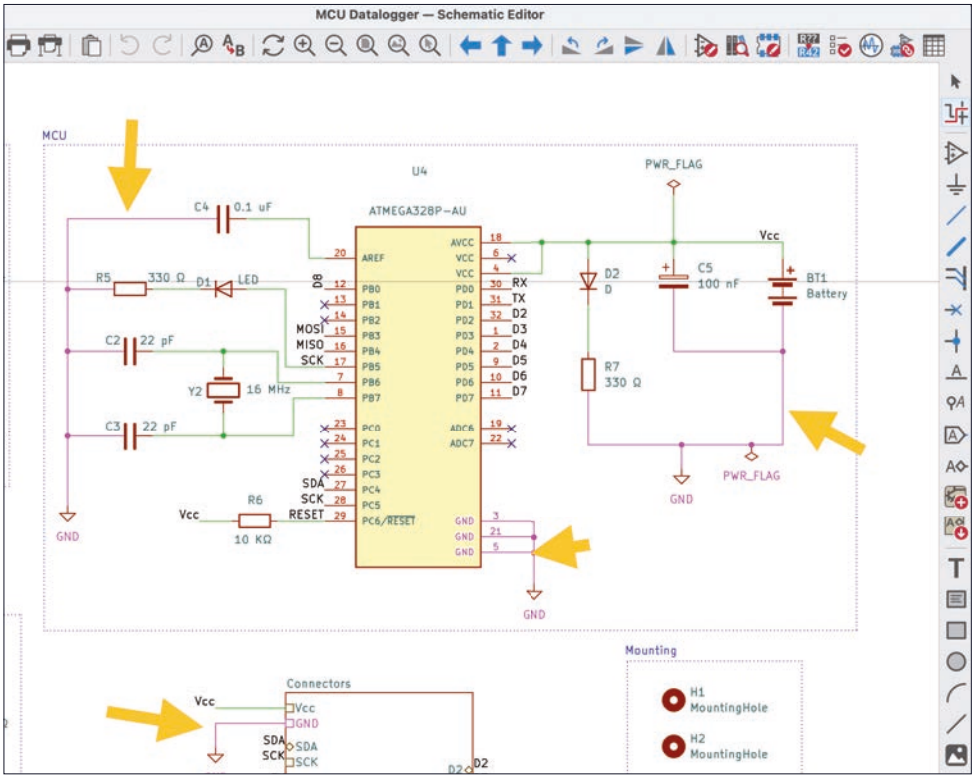


Figure 5.16.2: All members of the GND net are highlighted in pink.

Also, notice the message in the status bar (bottom left corner), indicating the highlighted net is "GND." If your schematic spans multiple pages, the net selection will carry through from one to the other.

By default, the highlighter color is light pink. If you'd like to change it to something else, like red, you can do so via the preferences window. Go to Preferences, Schematic Editor, Colors, and look for "Highlighted items." Click on the color box to bring up the color picker and choose a different color. Click OK to exit the Preferences window.

Below, you can see the members of the GND net, which is highlighted in red.

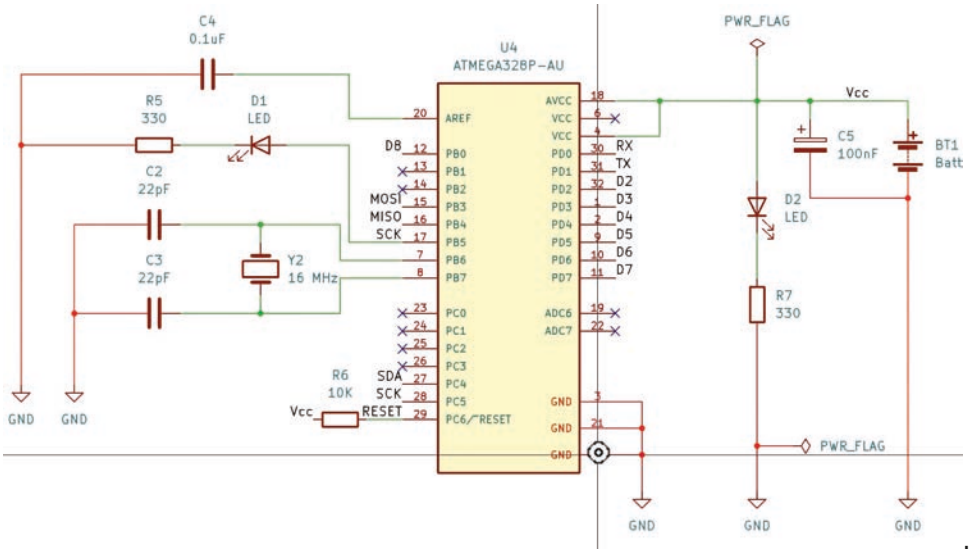


Figure 5.16.3: All members of the GND net are highlighted in red.

5.17. KiCad project management with Git

In this chapter, you will learn how to set up the Git version control system on your computer and use it to manage your KiCad projects. At its core, Git is a distributed version control system that helps you track file changes over time. Git allows you to revisit old versions of your projects, compare changes, and even revert to previous states if necessary.

Once you commit your project to a Git repository, you will record a complete history of its development and explore new features without risking the work you have already done. You can also share your work with others and incorporate changes made by team members. You can quickly return to any committed point in your project's history (no need to keep typing Ctr-Z). You can use tags to mark important milestones in your project with a name that is easy to remember.

Once you become familiar with Git and a simple Git + KiCad workflow, you will find it too risky to work without Git. You can use Git to track KiCad projects because KiCad projects consist mostly of text files. Look inside a KiCad project directory, and you will find files such as these:

- The project ".kicad_pro" file.
- The layout ".kicad_pcb" file.
- The schematic ".kicad_sch" file.

If you have created Gerber files, you will find files such as ".gbl", ".gbs", and ".gbo". If you have libraries, you'll see files such as ".kicad_mod", ".kicad_sym" and ".step".

All of these files contain simple text. Here's examples of the content of some of these files (only showing a few lines for this example):

.kicad_pro:

```
{
  "board": {
    "design_settings": {
      "defaults": {
        "board_outline_line_width": 0.09999999999999999,
        "copper_line_width": 0.19999999999999998,
        "copper_text_italic": false,
        "copper_text_size_h": 1.5,
        "copper_text_size_v": 1.5,
        "copper_text_thickness": 0.3,
        "copper_text_upright": false,
        "courtyard_line_width": 0.049999999999999996,
.....
```

.kicad_pcb:

```
(kicad_pcb (version 20210623) (generator pcbnew)

  (general
    (thickness 1.6)
  )

  (paper "A4")
  (layers
    (0 "F.Cu" mixed)
    (31 "B.Cu" mixed)
    (32 "B.Adhes" user "B.Adhesive")
    (33 "F.Adhes" user "F.Adhesive")
    (34 "B.Paste" user)
```

```
    (35 "F.Paste" user)
    (36 "B.SilkS" user "B.Silkscreen")
    (37 "F.SilkS" user "F.Silkscreen")
….

.kicad_sch:

(kicad_sch (version 20210621) (generator eeschema)

  (uuid 6b311d3d-9ae1-4ec9-af71-d4c3a43f0d08)

  (paper "A4")

  (title_block
    (title "A 4x8x8 LED Matrix Display Clock")
    (date "2021-07-14")
    (rev "${design_version}")
  )

  (lib_symbols
    (symbol "ArduinoProMiniSimple:ArduinoProMiniSimple" (in_bom yes) (on_board
yes)
…..
```

KiCad's projects are composed of text files, making it easy to use a versioning system like Git to track all changes made across the project. Several popular versioning systems are available, but my preference is Git. It is open-source, fast, widely used, and, as you will see in this recipe, it is also straightforward to use.

What you will learn in this recipe is how to use Git and the GitHub online repository to maintain your project's history. Doing so will allow you to:

1. Preserve your project's history. This will allow you to access past versions of any file in your project.
2. Create experimental branches. This is useful if you want to experiment with alternate design options or design different versions of the same board. Each one can be stored in a separate branch of the same repository.
3. Merge or discard different branches. This Git function allows you to merge (unify) two branches into one. For example, you may have the main branch of your project and work on an experimental branch as you investigate a special board feature. If the experiment succeeds, you can merge the experimental branch to the main branch and continue there. If not, you can discard the failed experiment branch and continue with the intact project in the main branch.

These are just three of the many possible scenarios. Those are the three scenarios that I use most often. Using Git alongside an online repository, like GitHub, you will use this versioning system to collaborate with others on the same project. You will also be able to share your project and its history with other people.

In Figure 5.17.1, you can see part of the most recent history[14] of one of the projects in this book, as it appears in the publicly accessible repository on GitHub.

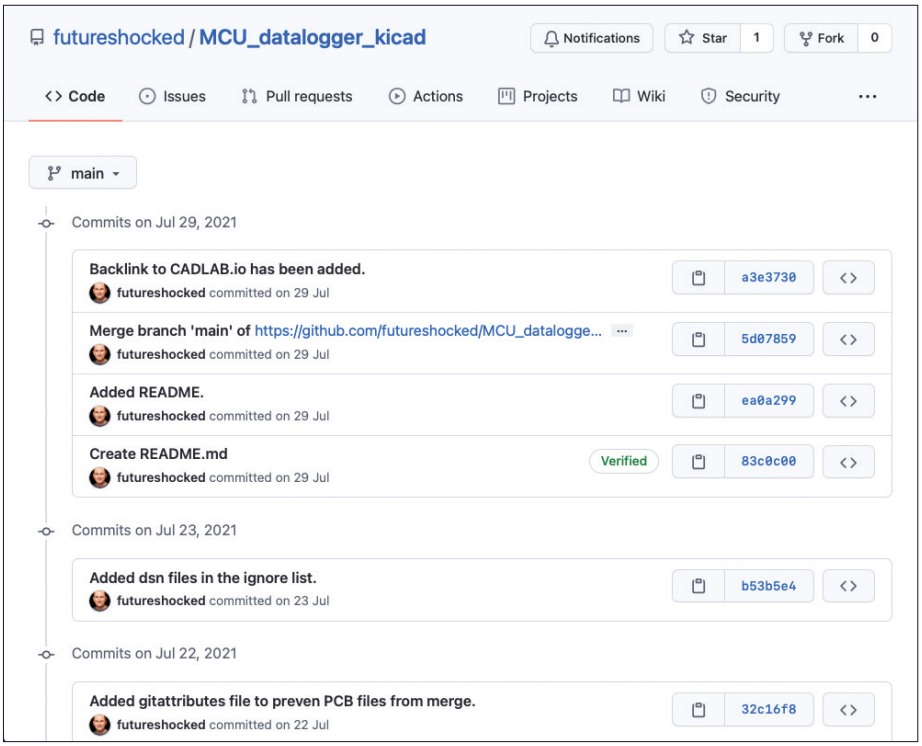


Figure 5.17.1: One of my KiCad projects on GitHub, showing the recent project history.

In Figure 5.17.2, you can see the history of the schematic file[15] for the same project.

14 https://github.com/futureshocked/MCU_datalogger_kicad/commits/main
15 https://github.com/futureshocked/MCU_datalogger_kicad/commits/main/MCU%20Datalogger.kicad_sch

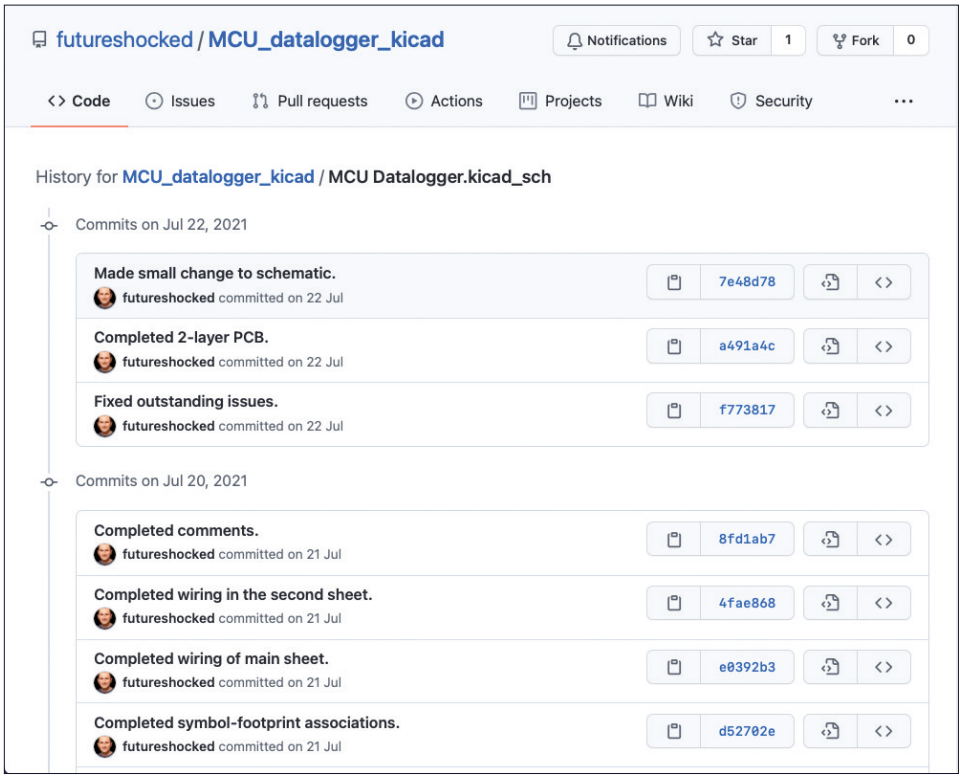


Figure 5.17.2: Part of the history of the schematic file on GitHub.

The numbers on the right side of each column are called 'commits.' Each number is an ID that refers to a commit. The full ID of a commit is a long alphanumeric, like '7e48d7843cbe041f04e2126f5f75c9c94cafce23'. What you see in Figure 5.17.2 is a short hash of that ID. A commit may contain changes in multiple files or additions and removals of files. You can click on the commit number to get detailed information about a commit. The result is a side-by-side comparison[16] of the changes detected in each file, as in the example of Figure 5.17.3.

16 https://github.com/futureshocked/MCU_datalogger_kicad/commit/7e48d7843cbe041f04e2126f5f75c9c94cafce23#diff-abe388af-70d1c33fbaacbcc26ed8f2069764d5bcb9d5515026526cb89e07c571

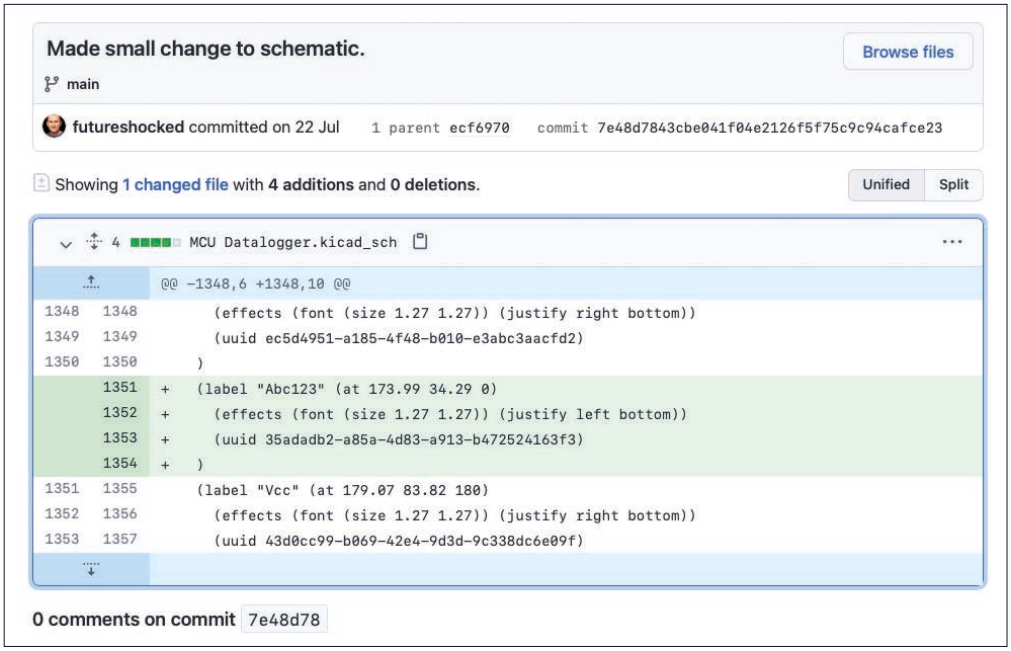


Figure 5.17.3: The changes in this file commit "MCU Datalogger.kicad_sch".

In this chapter, I will give you a brief introduction to Git and GitHub. Git is a big topic, and I encourage you to learn more about it using a specialized source like this Getting Started guide from GitHub.

Read on to learn how to use Git and GitHub in the context of a KiCad project. You will learn how to:

- Install and configure Git on your computer.
- Create a new Git repository for your KiCad project.
- How to exclude files you do not want to track in Git.
- Commit changes to your project to the repository.
- See those changes in the log.
- Check out past commits.
- Create branches.
- How to merge branches.
- How to prevent the merging of specific files from different branches.
- Upload your project to GitHub to share it with others.

5.17.1. Install Git

Git has installers for MacOS, Windows, and Linux. You can download the installed for your operating system at https://git-scm.com/downloads (Figure 5.17.1.1).

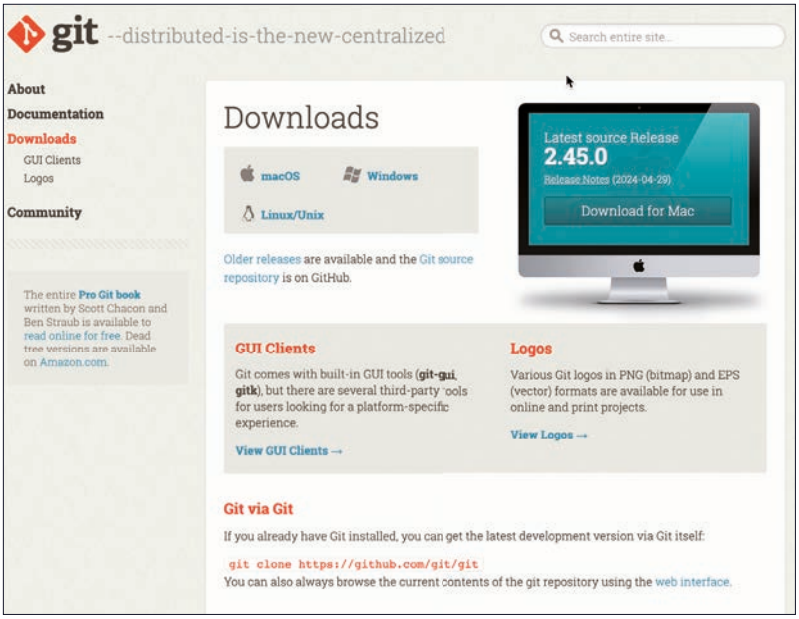


Figure 5.17.1.1: The download page for the Git installer.

The installation process is straightforward and hassle-free. I recommend using the binary installer for your OS, which is much simpler than trying to compile from the source. You can also find clear installation instructions and the binary installer downloads:

- MacOS[17].
- Windows[18].
- Linux[19].

Go ahead and install Git on your computer. After installing Git, continue to the next segment, which will show you how to configure it.

5.17.2. Git configuration

Now that you have installed Git on your computer, you need to make some simple configurations before using it. Git works best when it knows who you are (i.e., your user name) and your email address. It will use this information to attribute operations such as commits and deletes to the user who performed them.

Bring up a terminal window. At the command prompt, configure your git user settings:

```
$ git config --global user.name 'testuser'
$ git config --global user.email 'testuser@example.com'
$ git config —-global credential.helper "cache —-timeout=86400"
```

17 https://git-scm.com/download/mac
18 https://git-scm.com/download/win
19 https://git-scm.com/download/linux

By default, the main branch of Git is called "main." The last configuration, where I use the "credential.helper" option, sets Git to cache my GitHub credentials so that I don't have to authenticate each time I interact with GitHub on the command line. The documentation provides details about all the available configuration options.

That's it; the configuration is complete. Let's continue to the next segment, where we'll create a new Git repository for a simple demonstration KiCad project.

5.17.3. Create a new KiCad project Git repository

Git is now ready to help manage a new (or existing) KiCad project. I'll show you how to create a new repository, make the first few commits, and check the repository status in this segment.

I will use a small KiCad demonstration project from earlier in this book. The project contains a small number of files, as you can see in Figure 5.17.3.1:

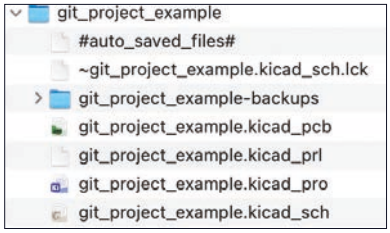


Figure 5.17.3.1: I'll create a Git repository on this KiCad project.

This project has a schematic that contains a few resistors and capacitors. It is very simple and, for this reason, appropriate to demonstrate how to use KiCad to track changes between commits (Figure 5.17.3.2).

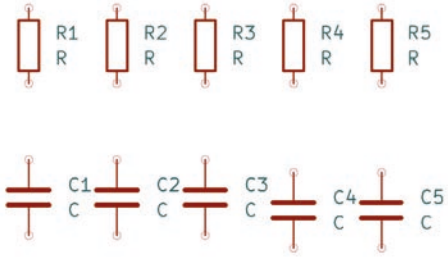


Figure 5.17.3.2: The project schematic before the first commit.

The layout editor is blank. Start by bringing up the terminal. To confirm that there is no Git repository setup yet, use the "status" command (Figure 5.17.3.3):

```
$ git status
Fatal: not a git repository (or any of the parent directories): .git
```

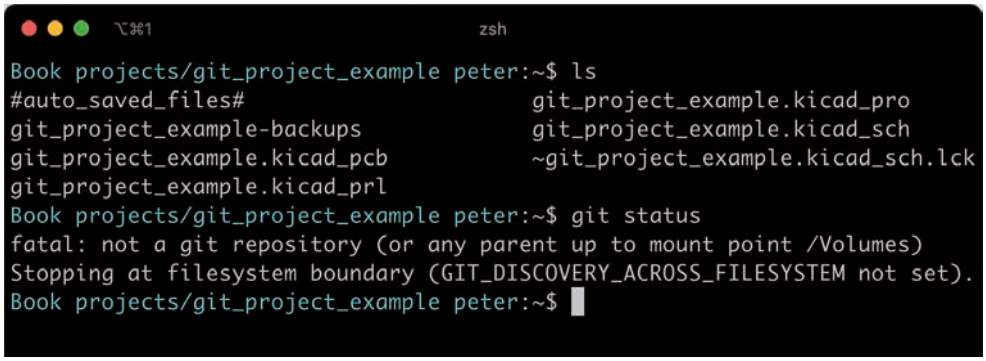


Figure 5.17.3.3: The "git status" command and its output.

The "status" command complained about the absence of the ".git" repository. Git keeps all repository data and configuration inside a directory named ".git," and the lack of this directory indicates that there is no Git repository present (Figure 5.17.3.4).

Let's create a new repository. The command is "init":

```
$ git init
Initialized empty Git repository in /Volumes/RAID/KLP 4e/Book/Book projects/git_
project_example/.git/
```

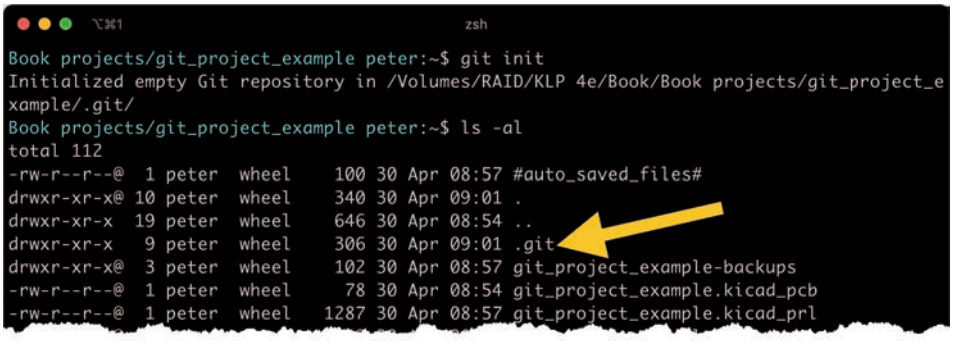


Figure 5.17.3.4: Created a new Git repository.

In the screenshot above, you can see that the ".git" is now present. This means that the KiCad project directory now contains a Git repository. The repository is empty, as you can confirm by using the "status" command once again (Figure 5.17.3.5):

```
$ git status
On branch main

No commits yet

Untracked files:
```

```
  (use "git add <file>..." to include in what will be committed)
        #auto_saved_files#
        git_project_example-backups/
        git_project_example.kicad_pcb
        git_project_example.kicad_prl
        git_project_example.kicad_pro
        git_project_example.kicad_sch
        ~git_project_example.kicad_sch.lck

nothing added to commit but untracked files present (use "git add" to track)
```

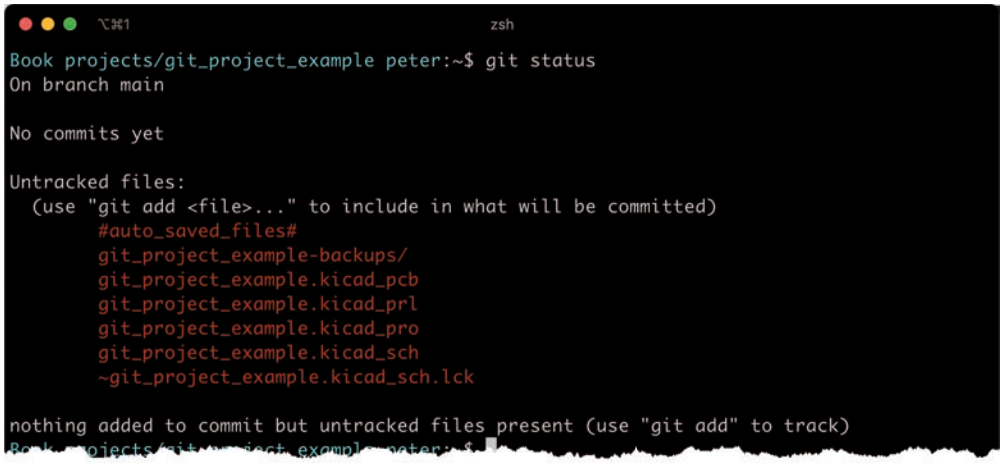


Figure 5.17.3.5: No commits, several un-tracked files.

The "status" command response indicates that there are no commits to the repository and that several files in the working directory are not being tracked. Git even suggests that I use the "add" command to monitor these files.

I'll show you how to do this soon, but first, I want to show you how not to track (or ignore") files that are not necessary. Ignoring such files keeps the repository tidy. An example of such a file is ".DS_Store" is created by MacOS and is not a project file. Ignoring this file is advisable. I'll show you how in the next segment.

5.17.4. How to ignore files

Not all files that are present in a project repository should be tracked. In the previous segment, I created a new Git repository for my simple KiCad project. One of the files in the project repository is ".DS_Store", a MacOS system file. I want to exclude this file since it is irrelevant to the project. It will also benefit my project collaborators working on Linux or Windows computers.

To instruct Git to ignore one or more files or a full directory, create a new file named ".gitignore." Use any text editor for this. Below, you can see the contents of my ".gitignore" file. I have annotated this file to help you understand it.

```
# KiCad specific ignore rules
*.bak
*.kicad_pcb-bak
*-cache.lib
fp-info-cache
*.sch-bak
*~.save
*.pro.user

# Compiled output files
*.dsn
*.ses

# Backup files
*.bck
*.bak
*~

# Zip Archives
*.zip

# Autosave files
_autosave-*

# MacOS system files
.DS_Store
.AppleDouble
.LSOverride

# Icon must end with two \r
Icon

# Thumbnails
._*

# Files that might appear on external disks
.Spotlight-V100
.DocumentRevisions-V100

# Directories potentially created on remote AFP share
.AppleDB
.AppleDesktop
```

```
Network Trash Folder
Temporary Items
.apdisk
```

If you use Windows, replace the segments below "MacOS system files" with this:

```
# Windows thumbnail cache files
Thumbs.db
ehthumbs.db

# Folder config files
Desktop.ini

# Recycle Bin
$RECYCLE.BIN/

# Windows Installer files
*.cab
*.msi
*.msm
*.msp

# Windows system files
.DS_Store
Thumbs.db
ehthumbs.db
Desktop.ini
```

In my .gitignore file, I am excluding several other files and directories in addition to the ".DS_Store" file. I use wild-card characters like "*" to match multiple files. For example, I am ignoring all files with extensions ".ses" and ".dsn" and all files inside any directory that contains the string "Gerber." I am also excluding the cache file "fp-info-cache," which will be recreated by KiCad when needed, and ".bck," used as a backup file. You should consider excluding unnecessary KiCad project files for storing or sending. You can see a listing of all KiCad project files in the KiCad documentation.

You can learn more about the syntax used in gitignore files in the Git documentation. Save this file, and then again try the "status" command. Here is the new output:

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
        #auto_saved_files#
        .gitignore
        git_project_example-backups/
        git_project_example.kicad_pcb
        git_project_example.kicad_prl
        git_project_example.kicad_pro
        git_project_example.kicad_sch
        ~git_project_example.kicad_sch.lck

nothing added to commit but untracked files present (use "git add" to track)
```

As you can see, the ".DS_Store" file is not included in the listing of untracked files. It is now ignored. It is time to make the first commitment, and I'll show you how in the next segment.

5.17.5. Basic Git commands: add, commit

I have configured Git, created a new repository, and set the gitignore file. It is time to make the first commit for this demo KiCad project. Remember that the last time I issued the "status" command, Git indicated that several files were untracked. Git will only commit tracked files, so before I commit, I must instruct Git to track one or more files.

The command for this is "add[20]," as in "add a file(s) to the repository and begin tracking them." For the first commit, I can use the "." operator with the "add" command so that Git will start tracking all non-ignored files. Here is my session:

```
$ git add .
```

That's it. I'll use "status" once again to see the changes (Figure 5.17.5.1):

```
$ git add .
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   #auto_saved_files#
        new file:   .gitignore
        new file:   git_project_example-backups/git_project_
example-2024-04-30_085746.zip
        new file:   git_project_example.kicad_pcb
        new file:   git_project_example.kicad_prl
        new file:   git_project_example.kicad_pro
```

20 https://git-scm.com/docs/git-add

```
        new file:   git_project_example.kicad_sch
        new file:   ~git_project_example.kicad_sch.lck
```

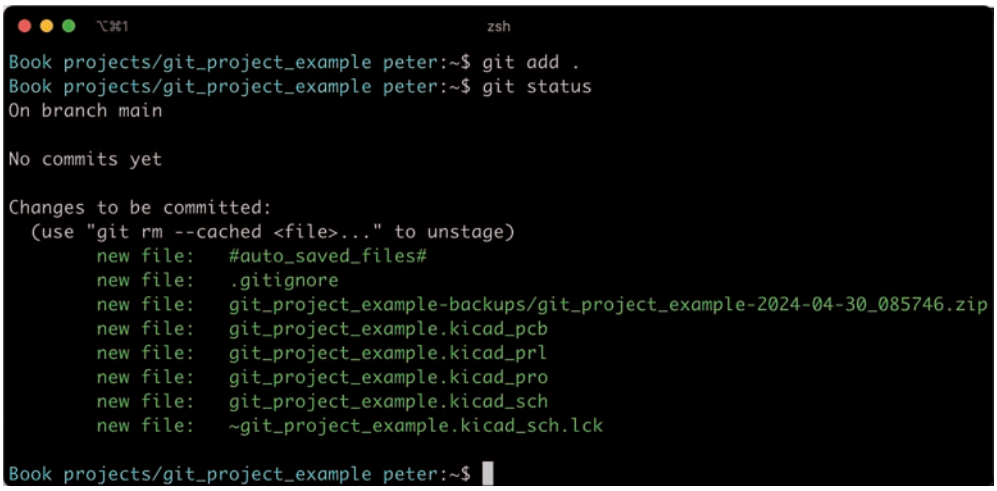


Figure 5.17.5.1: No commits, several tracked files.

The status command reports that several files are being tracked but not committed. Notice that Git shows the tracked files in green. In Figure 5.17.5.4, untracked files are in red. Files tracked but not committed can change, but Git will not capture the changes in the repository until the files are committed.

To commit a file (or files) to the repository, use the "commit[21]" command. This command can receive various modifiers and parameters. You will most likely want to use the "-am" flags. With "a," you tell Git to commit all files waiting in the index (that is, being tracked). With "m," you can include a commit message in the same command (otherwise, Git will open a text editor and ask for one). Here is the first commit command for this repository:

```
$ git commit –am "First commit."
[main (root-commit) 91272af] First commit.
 8 files changed, 1507 insertions(+)
 create mode 100644 #auto_saved_files#
 create mode 100644 .gitignore
 create mode 100644 git_project_example-backups/git_project_
example-2024-04-30_085746.zip
 create mode 100644 git_project_example.kicad_pcb
 create mode 100644 git_project_example.kicad_prl
 create mode 100644 git_project_example.kicad_pro
 create mode 100644 git_project_example.kicad_sch
 create mode 100644 ~git_project_example.kicad_sch.lck
```

21 https://git-scm.com/docs/git-commit

The commit command response confirms that ten files were committed into the master branch, and the hash of the commit ID is 16a0e89. I'll do another "git status" to see the current status of my repository:

```
$ git status
On branch main
nothing to commit, working tree clean
```

A Git project can contain more than one branch. In Git, a "branch" is a powerful concept that allows developers to diverge from the main line of development and continue working independently without affecting the rest of the project. Branching serves as a way to isolate development work without disrupting the functioning or stability of the main project, typically referred to as the "main" branch. As mentioned earlier in this chapter, the default branch in new Git projects is called "main." If your Git project has multiple branches (as we did in the MCU Datalogger project earlier in this book), you may want to check your current branch on the command line.

The command for this is "branch[22]". If you issue "git branch" without any parameters, Git will give you a listing of branches in the repository and will indicate the working branch with a star. Here is an example:

```
$ git branch
* main
```

Another command that I use frequently is "log." With the log command, you can see the recent history of your project. Try it now:

```
$ git log
commit 91272af67cde738577c57f720b7619a6af36a854 (HEAD -> main)
Author: peter <peter@abc.com>
Date:   Tue Apr 30 09:29:48 2024 +1000

    First commit.
```

The log command responds with a list of commits, their IDs, author and date information, and the commit message. Since I have only made a single commit in this example, the response contains a single record.

Let's make a small change to the schematic to learn how to deal with change. Open the schematic, and add a new resistor (Figure 5.17.5.2):

22 https://git-scm.com/docs/git-branch

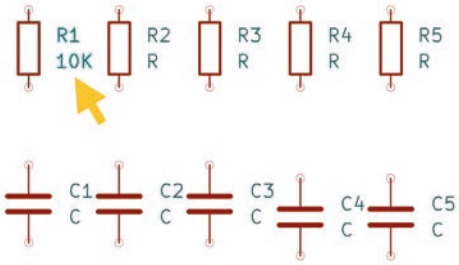


Figure 5.17.5.2: I have added a new resistor (R5) to the schematic.

Save the schematic file. Back in the command line, use the "status" command to see if Git has detected the change:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   git_project_example.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
```

Yes, of course, it did. Git has detected a change in "git_project_example.kicad_sch." You can see the exact changes with the "git diff[23]" command (Figure 5.17.5.3):

```
% git diff
```

23 https://git-scm.com/docs/git-diff

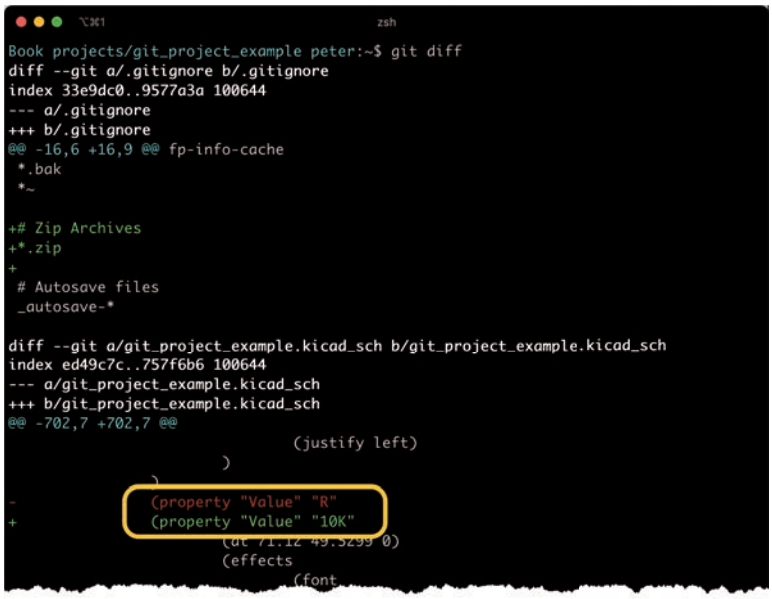


Figure 5.17.5.3: "Git diff" shows the current changes.

The "diff" command will show the new content in green and the removed content in red. I will proceed with the new commit:

```
$ git commit -am "Added the value for R1."
[main ddf7482] Added the value for R1.
 2 files changed, 12 insertions(+), 1 deletion(-)
```

The hash from the new commit ID is ddf7482. You can use this hash later to check out a specific commit. Your repository now contains two commits.

Imagine that you would like to experiment with your design. It may not work out, and you may need to retract to the project's current state. To do this safely, you can create a new Git branch and do the necessary work there. I'll show you how to work with branches in the next segment.

5.17.6. Basic Git commands: branch

In Git convention, the "main" branch contains the "official" state of the project. This may be the project instance you want to share with others, publish on your website, compile for production, or send to the PCB manufacturer. Work that involves adding new features, experimenting, or testing is typically done in dedicated branches.

For example, in my KiCad projects, I often maintain a main branch with the PCB that I am comfortable sharing with students and other branches to experiment with different layout configurations or components. In one of the projects in this course, I maintain separate branches for the 2-layer and 4-layer versions of the same board.

In this segment, I will show you how to work with branches. To create a new branch, use the "checkout[24]" command with the "-b" switch and a name. The "-b" switch will create a new branch. If you use "checkout" without "-b," Git will attempt to checkout (i.e., make active) the specified branch. Here is the command and the response:

```
$ git checkout -b experimental
Switched to a new branch 'experimental'
```

You can verify that the new branch exists using the "branch" command:

```
$ git branch
* experimental
  main
```

In the output above, notice the "*" before the "experimental" branch. This indicates that the starred branch is active, and any new commits will go into this branch. The new branch has inherited all content from the previously active branch, "main." The schematic editor's content is identical to that under the main branch.

I will now introduce a new capacitor, "C6". The new schematic looks like this (Figure 5.17.6.1):

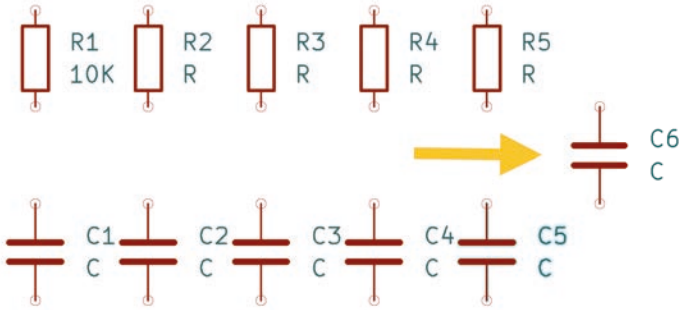


Figure 5.17.6.1: While working in the "experimental" branch, I changed the schematic.

Save the schematic editor and return it to the terminal. Get the new status of the project:

```
$ git status
On branch experimental
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
       modified:   git_project_example.kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
```

24 https://git-scm.com/docs/git-checkout

Git responds with the name of the current branch ("experimental") and the file's name where the change is detected. Let's commit this change with the "commit" command:

```
$ git commit -am "Added capacitor C6."
[experimental 65a2581] Added capacitor C6.
 1 file changed, 82 insertions(+), 12 deletions(-)
```

Git confirms that one new file has been committed to the "experimental" branch. The first seven characters of the commit ID are 65a2581. Also, try a "git log" (Figure 5.17.6.2):

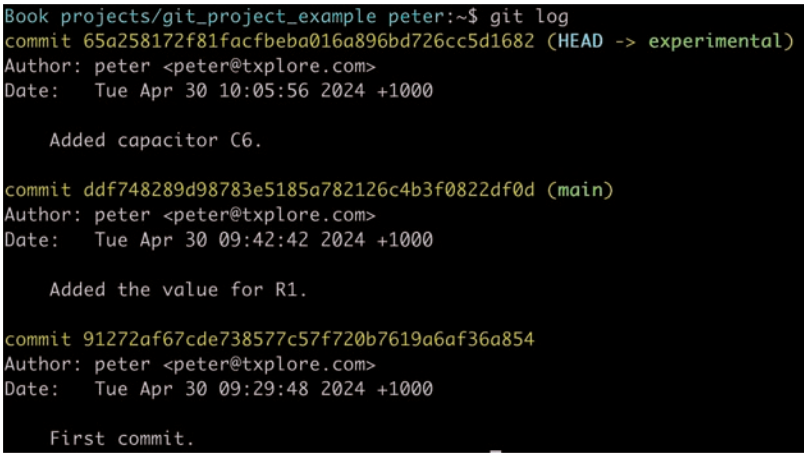


Figure 5.17.6.2: Git log showing the recent repository history.

The "git log" reports the full commit IDs. Notice the complete commit ID of the latest commit (the first in the list) and how the first seven characters of the ID match those reported by the commit command ("65a2581").

Say that you would like to go back to the main branch. You can check out the main branch's latest commit without affecting any work you have done in the experimental branch. To checkout to the HEAD (i.e., the newest commit) of a branch, use the "checkout" command (ensure that the schematic editor window is closed to avoid inadvertently changing the schematic in the main branch):

```
% git checkout main
Switched to branch 'main'
```

Now, use "git log" to see the status of the repository in the main branch:

```
% git log
commit ddf748289d98783e5185a782126c4b3f0822df0d (HEAD -> main)
Author: peter <peter@abc.com>
Date:   Tue Apr 30 09:42:42 2024 +1000
```

```
    Added the value for R1.

commit 91272af67cde738577c57f720b7619a6af36a854
Author: peter <peter@abc.com>
Date:   Tue Apr 30 09:29:48 2024 +1000

    First commit.
```

As you can see, the new capacitor does not exist in the main branch. I created the capacitor while working in the experimental branch and committed this change there. You can verify this by opening the schematic editor (capacitor C6 will not be in the schematic).

Now, assume that you are happy with the experimental branch change and would like to "import" this change into the main working branch. To do this, you will use the "merge" command. I'll show you how in the next segment of this chapter.

5.17.7. Basic Git commands: merge

In the previous segment of this chapter, you created an experimental branch and changed it. Now, you want to import this change to the main working branch. The git term for this operation is "merge." Effectively, you want to merge one branch into another. To do this, you will use the "merge[25]" command.

Start by confirming that you are currently working in the main branch (which is where you want to merge into):

```
$ git branch
  experimental
* main
```

The star next to "main" indicates that you work in the main branch. Do the merge:

```
$ git merge experimental
Updating ddf7482..65a2581
Fast-forward
 git_project_example.kicad_sch | 94 ++++++++++++++++++++++++++++++++++++++++++++----
---
 1 file changed, 82 insertions(+), 12 deletions(-)
```

The merge response provides information about what just happened. One file in the main branch was changed. The file that was changed is "it_project_example.kicad_sch."

The "merge" command is very flexible. You can merge a branch into another branch and a specific commit into a branch using the commit ID. In most cases, using branch names, as in the example above, is sufficient. In the line starting with "Updating," the notation

25 https://git-scm.com/docs/git-merge

ddf7482..65a2581 indicates that Git is applying changes introduced in all commits from after ddf7482 up to and including 65a2581

The merge operation does not overwrite contents; it merges contents. Git has a lot of intelligence embedded to ensure that work is not lost. If Git cannot figure out how to merge, it will ask you to clarify. Merge conflicts happen when, for example, the same lines in a file are changed in both the source and the destination conflict. In such cases, Git will not change anything and ask you to resolve a conflict manually.

With the merge completed, let's verify that the capacitor I added in the experimental branch now exists in the main branch. Open the schematic editor, and look at the current state of the schematic (Figure 5.17.7.1):

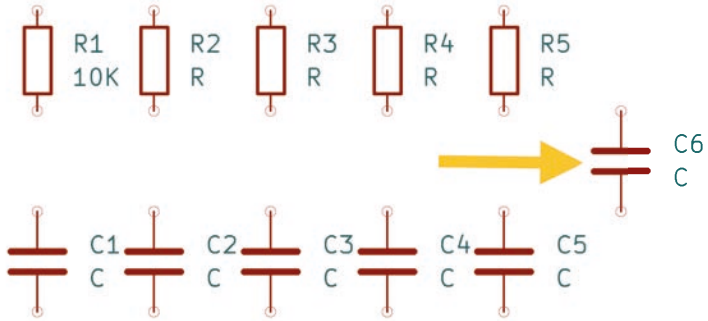


Figure 5.17.7.1: The new capacitor exists in the main branch.

The new capacitor exists in the main branch (as well as in the experimental branch). You can also use the "log" command to get the recent repository history:

```
$ git log
commit 65a258172f81facfbeba016a896bd726cc5d1682 (HEAD -> main, experimental)
Author: peter <peter@abc.com>
Date:   Tue Apr 30 10:05:56 2024 +1000

    Added capacitor C6.

commit ddf748289d98783e5185a782126c4b3f0822df0d
Author: peter <peter@abc.com>
Date:   Tue Apr 30 09:42:42 2024 +1000

    Added the value for R1.

commit 91272af67cde738577c57f720b7619a6af36a854
Author: peter <peter@abc.com>
Date:   Tue Apr 30 09:29:48 2024 +1000

    First commit.
```

As you can see from the output above, commit 65a258 exists in both the main and the experimental branches.

Now that the changes in the experimental branch are merged with "main," we no longer need the experimental branch. It is good practice to delete unneeded branches and keep the Git repository tidy. To delete a branch, use the "branch" command with the "-d" switch like this:

```
$ git branch –d experimental
Deleted branch experimental (was 65a2581).
```

Repeat the "git branch" command to confirm that the repository contains only the main branch. If the branch that you want to delete contains unmarked changes, but you are sure that you do want to delete it, you can use the "-D" switch (instead of "-d"). This will force Git to delete a branch with unmarked changes. Without it, Git will ask you for confirmation. Remember: Git is designed to minimize the risk of losing work. If you try to do something that can result in lost work, Git will let you know and prevent you from losing work.

5.18. Sharing your KiCad project on GitHub

Suppose you would like to publish your project online so other people can access it. You can do this easily by creating a remote Git repository on a cloud service like GitHub. The repository on GitHub is a remote copy of your local repository. When you make a change to your local repository, you can push it to the remote so that your collaborators can access the updates. And vice-versa: if a change is accepted in the remote, you can pull it to your local repository so that you can use the changes.

In this chapter, you will learn how you can use GitHub to share your existing KiCad project with others. The project I will use to demonstrate the process is the one I also used in the chapter "5.17. KiCad project management with Git."

The process has four steps:

1. Create a remote repository on GitHub.
2. Set the origin.
3. Sync the remote repository with your local Git repository.
4. Share the GitHub repository with others.

Let's begin.

1. Create a remote repository on GitHub

Start by creating a free account on GitHub if you don't already have one.

Then, login and create a new blank GitHub repository (Figure 5.18.1) :

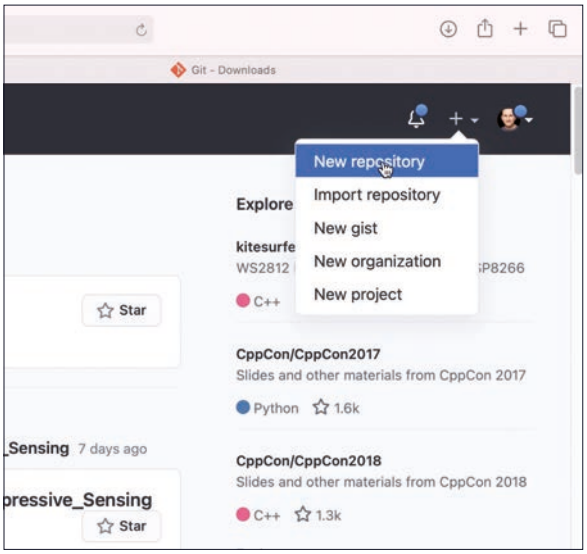


Figure 5.18.1: Create a new blank repository on GitHub.

Give it a name, a description, and an access type. I have set my demo repository to "Public." There are a few optional settings that you can enable, such as the addition of a README or ".gitignore" files (Figure 5.18.2):

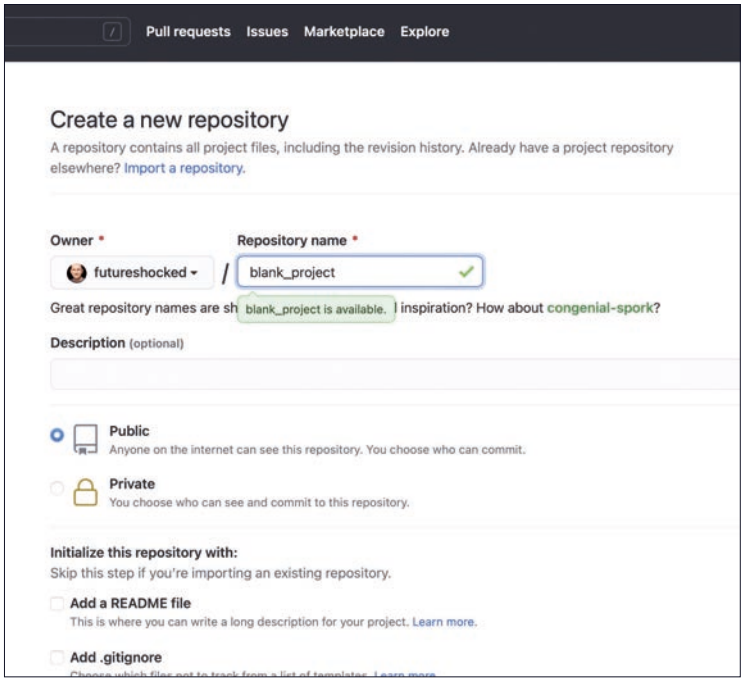


Figure 5.18.2: Create a new repository on GitHub.

Click on the green "Create repository" button to finish this step.

2. Set origin
When GitHub completes creating the new repository, it will show you a page containing the information you need in the next few steps. Copy this information into a text editor so that you can use it later. Most important is the HTTPS and SSH address of your GitHub repository that appears at the top of the page ("1" in Figure 5.18.3):

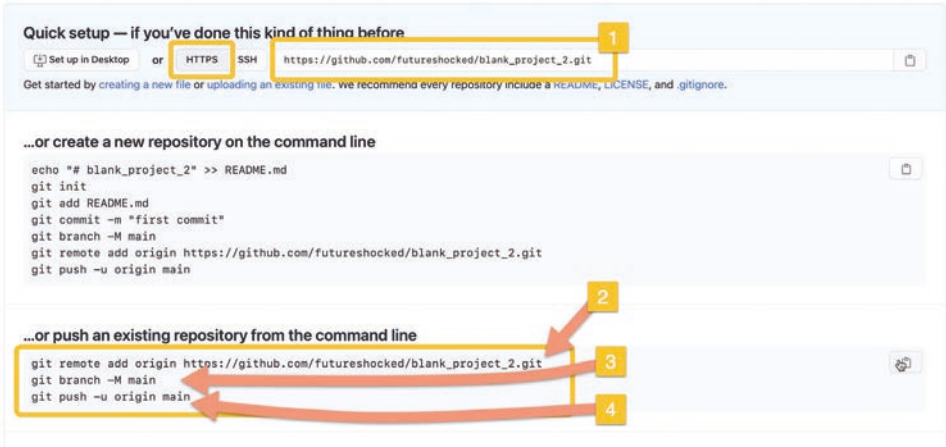


Figure 5.18.3: The URL of the new repository.

The same page contains instructions on setting the "origin" repository for your local Git repository. The name "origin" represents the remote repository of your local Git repository that (in our example) is hosted on GitHub. You may choose a different name if you wish, like "GitHub" or "cloud," however, remember that convention uses "origin" for the primary remote repository.

Also, notice at the bottom of the GitHub instructions in Figure 5.18.3 the commands GitHub suggests referring to the "main" branch. This means that we will be syncing the "main" branch first. If your local repository contains additional branches that you wish to "push" to GitHub, you can do this subsequently. You can choose to push specific branches only.

Next, go to your command line, and browse to the directory where you saved your project.

Copy the first line of the GitHub instructions ("2" in Figure 5.18.3):

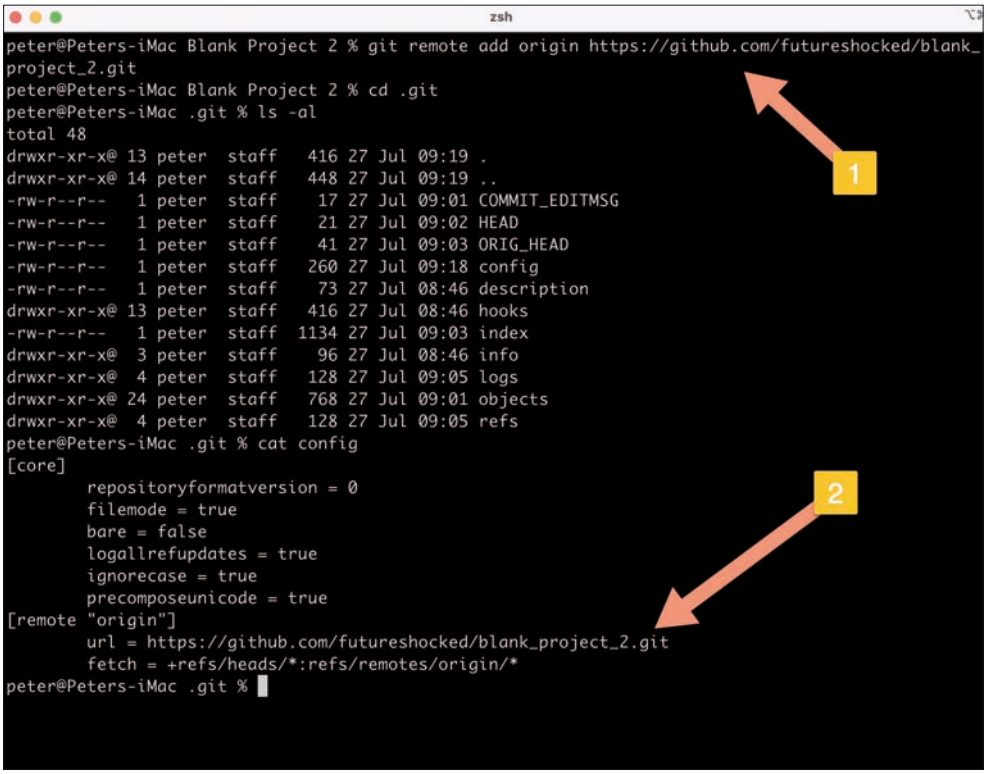


Figure 5.18.4: Setting the origin.

When you set the origin ("1" in Figure 5.18.4), Git will record the origin URL in the config file inside the ".git" folder. You can confirm this by using the "cat" command on the config file (or simply open config in a text editor). See "2" in Figure 5.18.4.

GitHub also recommends that you use "git branch -M main" to switch the name of the unborn branch to "main." This is unnecessary because I have already created this repository's "main" branch. You can still issue this command, but it will not change anything.

Let's continue with step 3.

3. Sync the remote repository with your local Git repository.
I will now push the "main" branch of my local repository to GitHub. To do this, I will use the 3rd command from Figure 5.18.4. This requires setting up a new personal access token to use in place of your regular account password.

To create a personal access token (refer to Figure 5.18.5), go to GitHub, click on your account photo (top right of the GitHub page), then Settings (1), and Developer settings (2).

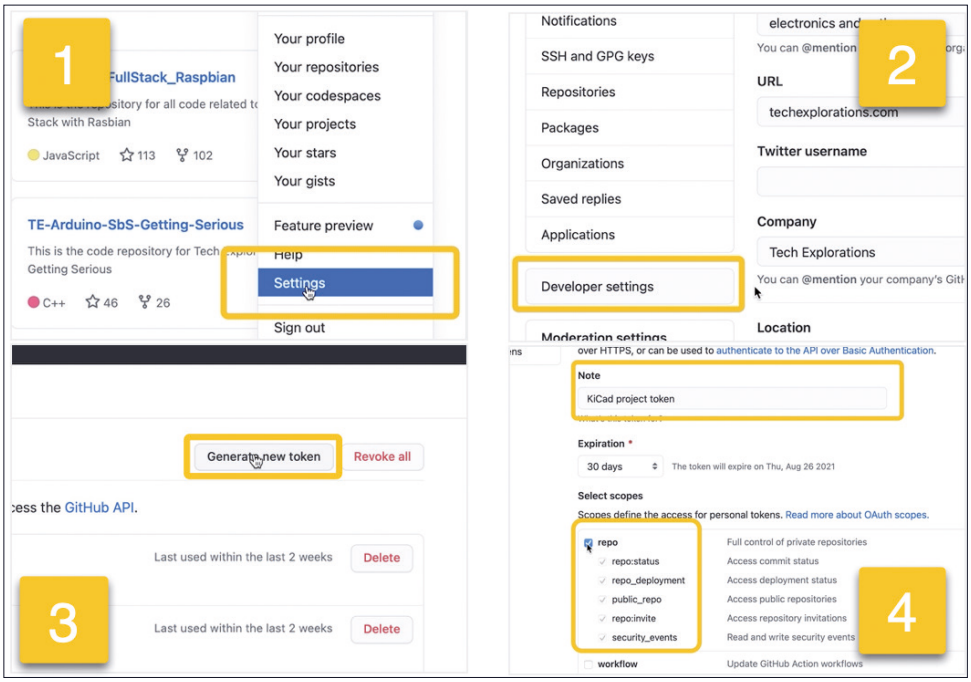


Figure 5.18.5: Generate a new personal access token.

Click on Personal access tokens in the Developer Settings page and then "Generate new token" (3). Give your new token a name and select all radio buttons in the "repo" group (4). Then click on the green "Generate token" button to finish the process.

You now have a new personal access token (Figure 5.18.6):

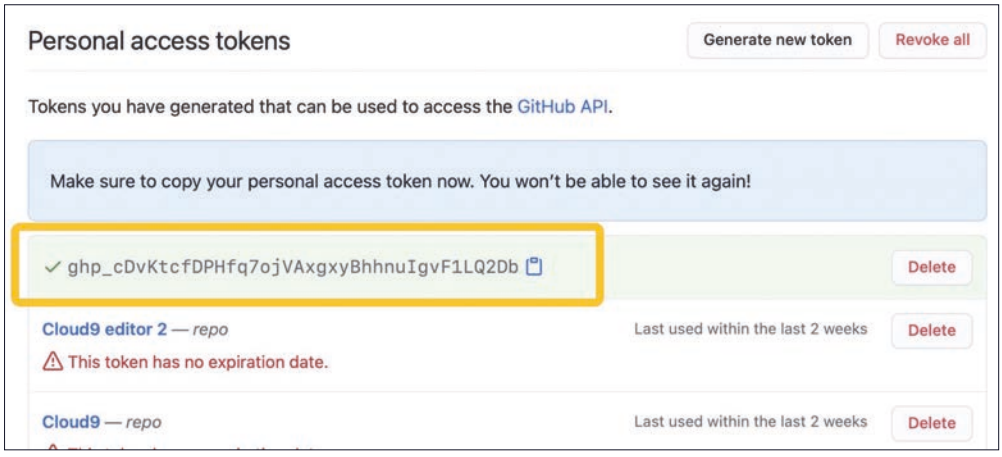


Figure 5.18.6: The new personal access token.

Continue at the command line. Copy command 4 from Figure 5.18.3:

```
% git push –u origin main
```

GitHub will ask for your username and password. Type or copy your username (or email).

For the password, use the personal access token that you created earlier (Figure 5.18.6)

Git will proceed to write "main" branch data to the origin (i.e. to the remote repository):

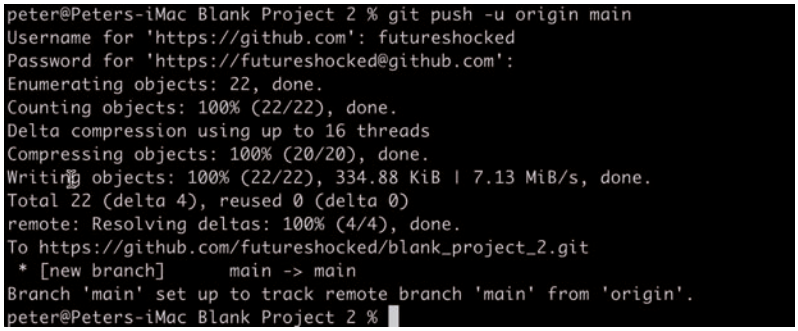


Figure 5.18.7: Synced "main" branch.

Your project ("main" repository branch) is now on GitHub. Refresh the project page to confirm:

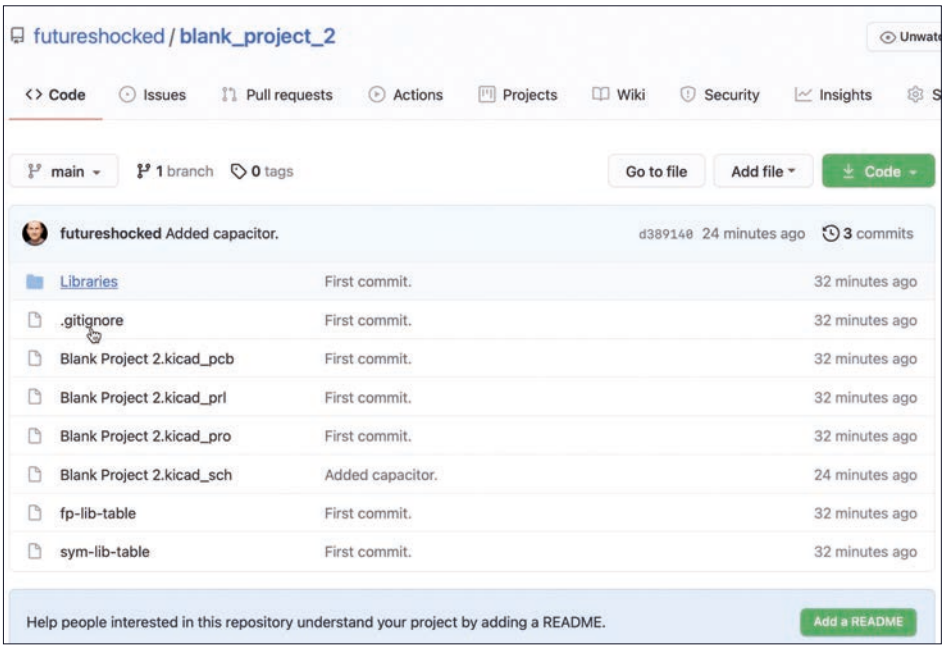


Figure 5.18.8: The KiCad project, "main" branch, on GitHub.

Take a moment to browse the files and look at their contents. Now, say that you want to do some work on the local branch. For example, you'd like to create a new branch for experimental work. As you already know, you can create a new branch like this:

```
% git checkout -b  experimental2
```

Then, go to the schematic and make a small change, like adding a new resistor. Save the schematic. Back on the command line, commit the change, and look at the repository status:

```
% git checkout -b experimental2
Switched to a new branch 'experimental2'
peter@Peters-iMac Blank Project 2 % git branch
* experimental2
  main
% git status
On branch experimental2
Changes not staged for commit:
       (use "git add <file>..." to update what will be committed)
       (use "git checkout -- <file>..." to discard changes in working directory)

               modified: Blank Project 2 kicad_sch

no changes added to commit (use "git add" and/or "git commit -a")
% git commit -am "Added R8."
[experimentalZ @bfaeSZ] Added R8.
 1 file changed, 22 insertions(+)
% git status
On branch experimentalZ
nothing to commit, working tree clean
%
```

The new branch ("experimental2") contains a change I'd like to push to GitHub. Branch "experimental2" does not exist on GitHub. To create the new branch and push its changes from local to "origin," I'll use this command:

```
% git push -u origin experimental2
Enumerating objects: 9, done. I
Counting objects: 100% (9/9), done.
Delta compression using up to 16 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 1.01 KiB | 1.01 MiB/s, done.
Total 7 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
remote:
remote: Create a pull request for 'experimental2' on GitHub by visiting:
```

```
remote: https://github.com/futureshocked/blank_project_2/pull/new/experimental2
remote:
To https://github.com/tutureshocked/blank_project_2.git
* [new branch] experimental2 -> experimental2
Branch 'experimental2' set up to track remote branch 'experimental2' From
'origin'.
%
```

Git did not ask for my credentials again because I have configured it to cache them for 24 hours. Refresh the repository webpage to confirm that the new branch was created on origin (1 in Figure 5.18.8):

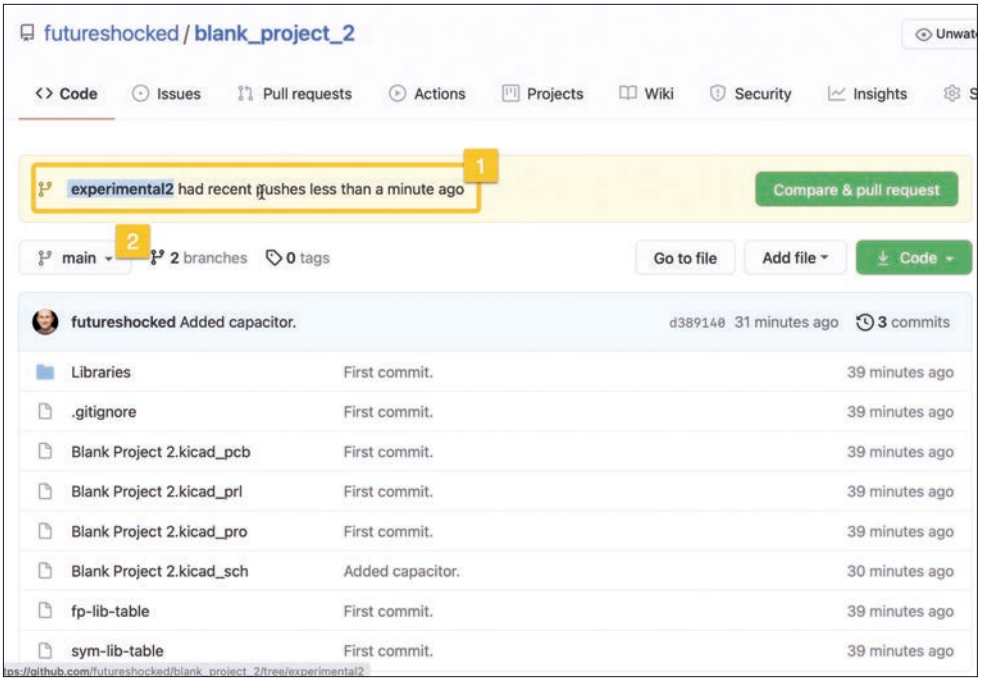


Figure 5.18.9: The new branch at origin.

You can use the branch dropdown on GitHub to switch between branches (2 in Figure 5.18.8).

4. Share the GitHub repository with others

To share your project with others, give them the repository URL on GitHub. Your collaborators can clone the repository on their computer, make changes, and then push the changes to the origin. You can pull the changes to your local repository using the "pull[26]" command:

```
% git pull -u origin experimental2
```

26 https://git-scm.com/docs/git-pull

The command above will pull changes made to the "experimental2" branch. To pull changes from any other branch, change the branch name to something else, like "main." Let's simulate this. You can use GitHub's edit function to change a repository file. Below, I am making a small change to the schematic file and saving the change (all done on GitHub, not the local repository):

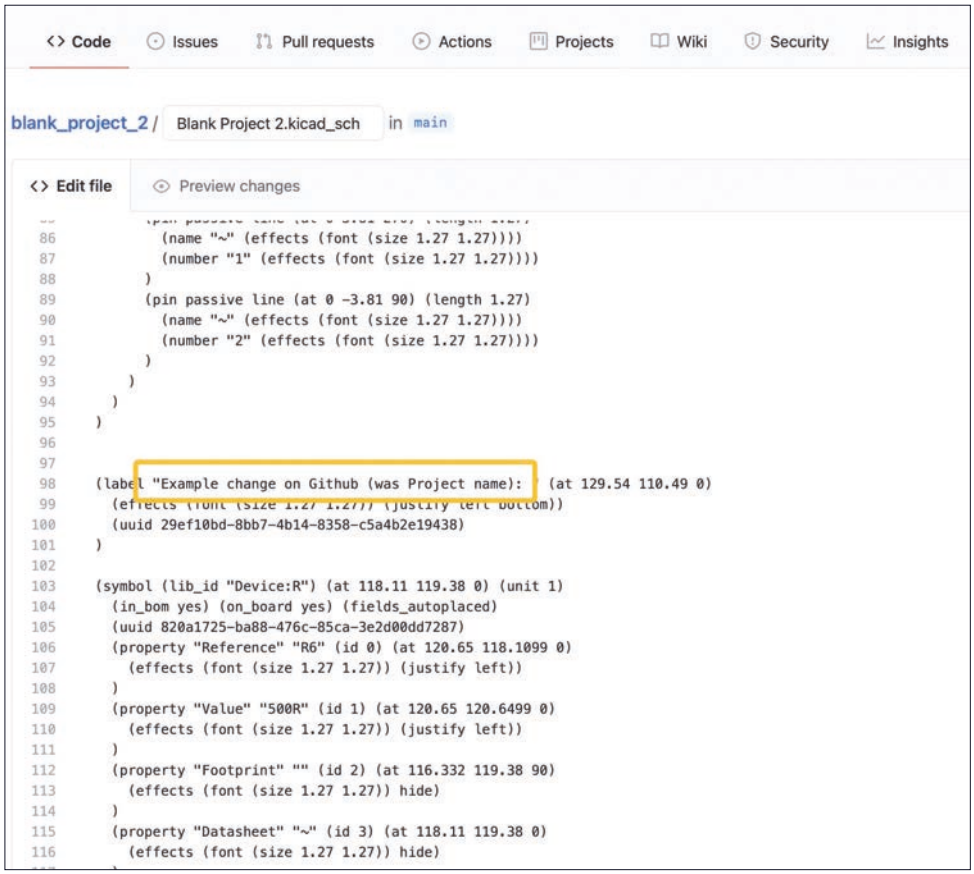


Figure 5.18.10: Made a change to the schematic file at origin.

Click the green "Commit changes" button to commit and exit the editor. Back on the command line, issue the "pull" command on the "main" branch (where I made the change):

```
% git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.

From https://github.com/futureshocked/blank_project_2
```

```
* branch main –> FETCH_HEAD
  d389140..585ff40 main -> origin/main

Auto–merging Blank Project 2.kicad_sch
Merge made by the 'recursive' strategy.
   Blank Project 2.kicad_sch | 2 +–
   1 file changed, 1 insertion(+), 1 deletion(-)
%
```

Git confirms that changes have been pulled in the schematic file of the "main" branch. If you switch your local working branch to "main" and then open the schematic in the schematic editor, you will see the change in the editor:

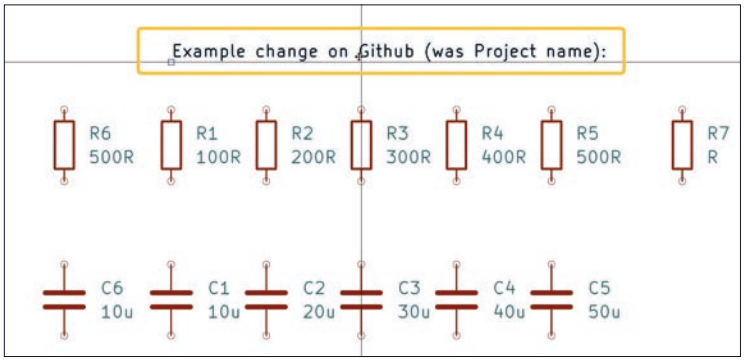


Figure 5.18.11: The change I made on GitHub appears in the schematic editor.

In Figure 5.18.11 (above) (above), you can see that the change that I made to the schematic file on GitHub appears in the KiCad schematic editor.

The process I have described above represents a simple use case of creating a new KiCad repository, sharing it with others using GitHub, and syncing the repository with changes made by the project members.

GitHub is versatile, and you can adjust it to your personal or team workflow requirements. Consider our dedicated GitHub course[27] on the Tech Explorations website to learn more about it.

5.19. Customize the editor color scheme

From its early days, KiCad has been very configurable. For example, the color theme used in the schematic and layout editors can be configured in KiCad. You can create custom color schemes for the schematic and PCB editors in the Preferences window. Each app has its theme settings. You can create multiple themes and quickly switch between them. See Figure 5.19.1 below for reference:

27 https://techexplorations.com/so/git-and-github/

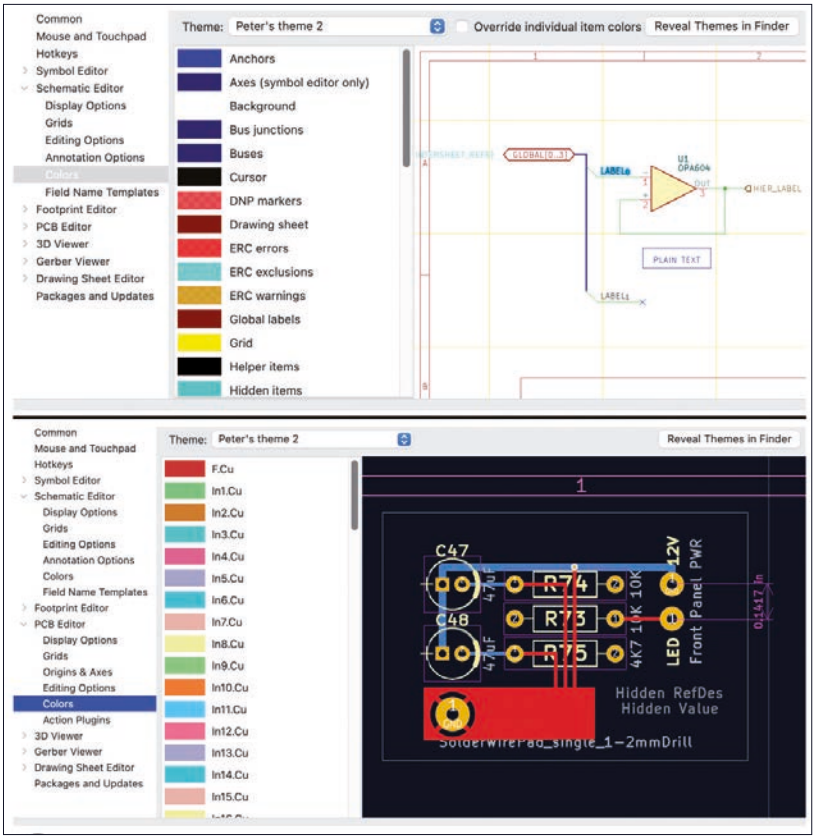


Figure 5.19.1: The Colors theme editors for the schematic and PCB editors.

Both editors work the same. You can select a theme via the drop-down menu at the top of the editor:

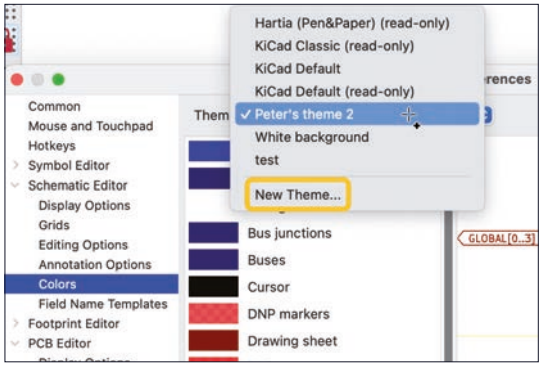


Figure 5.19.2: The theme drop-down.

You can click "New Theme..." to create a new theme. All themes except for the two build-in, "KiCad Classic" and "KiCad Default," are editable. To change a color, double-click on the

box that contains the color and use the color picker or defined colors tabs to select the new color. Then, click on OK to commit the new color. For example, to change the color of the background in the schematic editor, click on the box that contains the background color:

Figure 5.19.3: Assigned Gray 3 to Background.

In the example above, I have selected Grey 3 from the Defined Colors tab for the background. Click OK to dismiss the Color Picker window and OK again to dismiss the Preferences window. The schematic editor now looks like this:

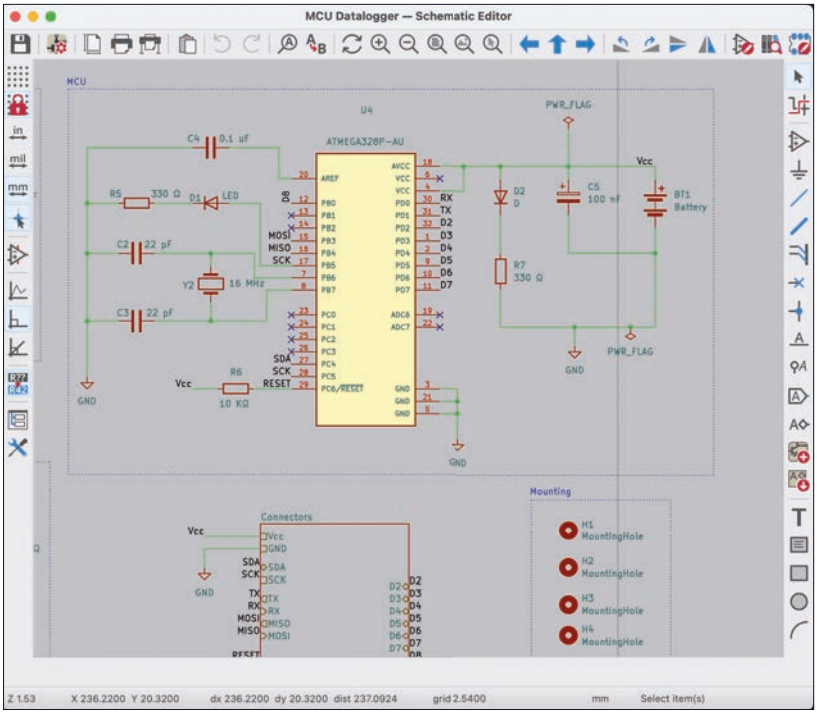


Figure 5.19.4: The schematic editor with Gray 3 for the background.

You can use the same method to set colors for all available items in schematics and PCB editors. It is helpful to have a theme for regular editing work and another for printing. The PCB editor's default theme has a black background, which is unsuitable for paper printing because of the amount of blank ink used. I have a printer-friendly theme with a white background so that my layout printouts look better on paper:

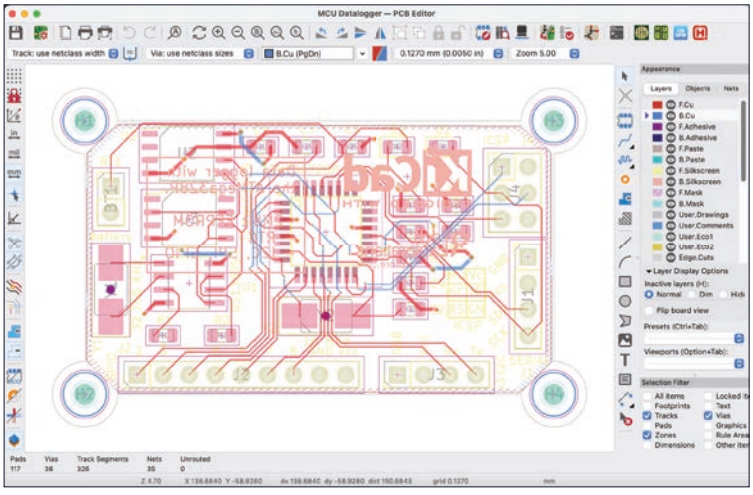


Figure 5.19.5: The PCB editor using a printer-friendly color theme.

5.20. Import an EAGLE, Altium, or Cadstar schematic

KiCad can work with PCB projects created in other CAD applications. It can also import schematics from Eagle, Altium, EasyEDA, and Cadstar. In most cases, imported projects will need editing to correct issues that arose from incompatibilities between these CAD applications and KiCad. I will show you how to use the import tool using an example. I will import an Eagle schematic for the Arduino Uno board into KiCad.

Start by going to the Arduino website, from where you can download the Eagle files (see https://store-usa.arduino.cc/products/arduino-uno-rev3/):

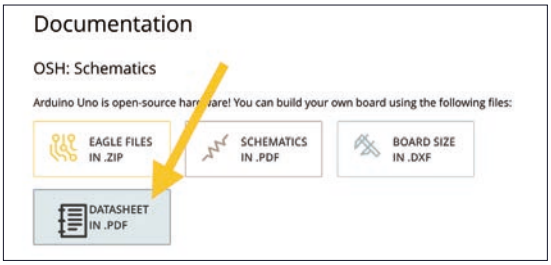


Figure 5.20.1: Get the Eagle project from the Arduino website.

The schematic I will import into KiCad looks like this:

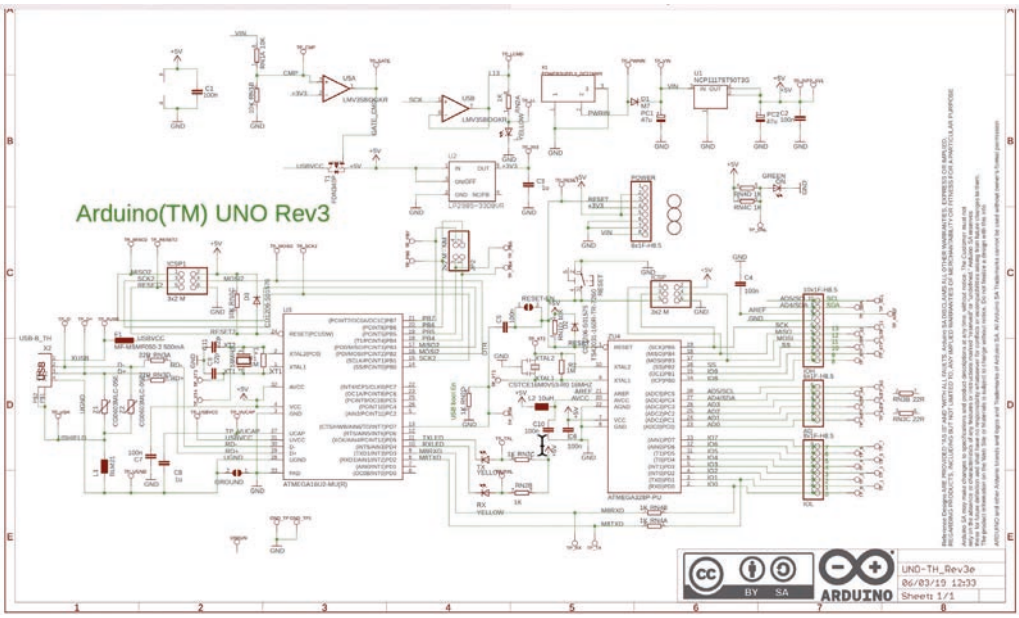


Figure 5.20.2: The Arduino Uno schematic in Eagle.

Expand the ZIP archive so you can see the two files it contains, the schematic (".sch") and the layout (".brd"):

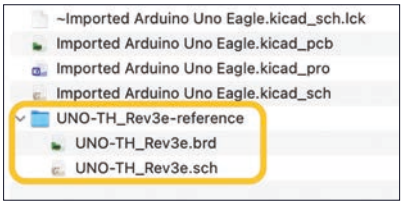


Figure 5.20.3: The two files of the ZIP file.

In KiCad, create a new project and open the schematic editor. Then click on File —> Import —> Non-KiCad Schematic.

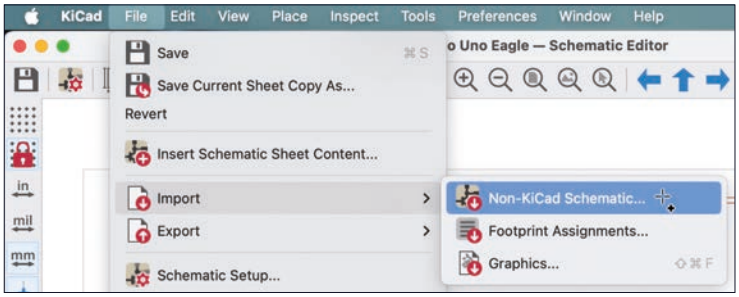


Figure 5.20.4: Importing a non-KiCad Schematic.

A file browser will appear. At the bottom of the browser, select the appropriate import file format ("1" in Figure 5.20.5), and then select the file with the ".sch" extension ("2" in Figure 5.20.5).

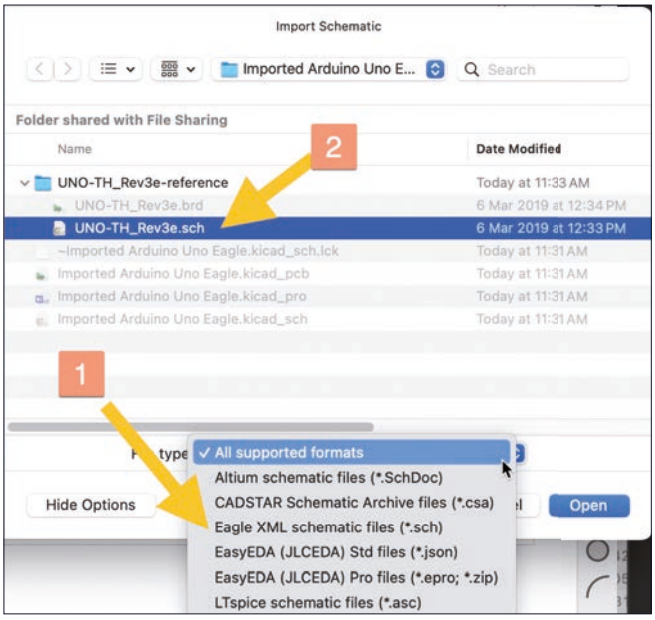


Figure 5.20.5: Select the file type and then the Eagle schematic file.

KiCad will try to import this file. KiCad will likely report one or more issues. For example, it may be unable to find a matching schematic component. KiCad will report this at the end of the import process. You can use this information to fix any issues that arise manually. At first glance, the import looks good.

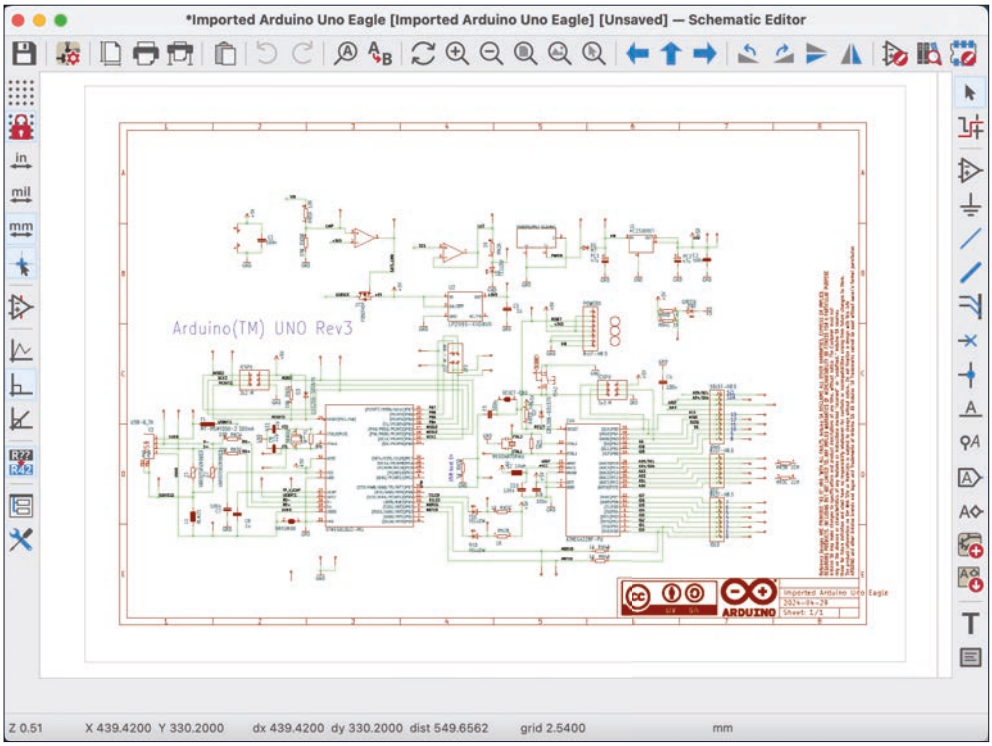


Figure 5.20.6: KiCad has imported the Arduino Uno Eagle schematic.

You can compare the screenshot in Figure 5.20.6 (above) with the one in Figure 5.20.2 and any differences. Things that you should check include:

- The editor sheet size.
- Text size and types.
- Graphics.
- Missing symbols.

You should take some time to fix those issues, especially missing components. Of course, once you adjust the basics, you should continue as you would with any other KiCad schematic and run the ERC. Because different CAD applications have features and capabilities that don't "translate" directly to KiCad, you can import schematics from Eagle, Altium, or Cadstar into KiCad. In most cases, you will need to do a small amount of editing to correct outstanding issues.

5.21. The circuit simulator

KiCad also includes an integrated circuit simulator in the schematic editor. This simulator is essential for engineers and designers seeking to validate and optimize their circuit designs before moving to the physical prototyping stage. This section will introduce the core functionalities and benefits of using the circuit simulator within KiCad. The circuit simulator in KiCad is based on the SPICE engine, a standard in the industry for simulating and analyzing electronic circuits. SPICE (Simulation Program with Integrated Circuit Emphasis) allows for examining the circuit's behavior using a detailed component model, which mimics real-life electrical and thermal properties. KiCad's integration of SPICE provides a seamless environment to simulate both analog and digital components.

You can find the simulator in the schematic under the Inspect menu (Figure 5.21.1).

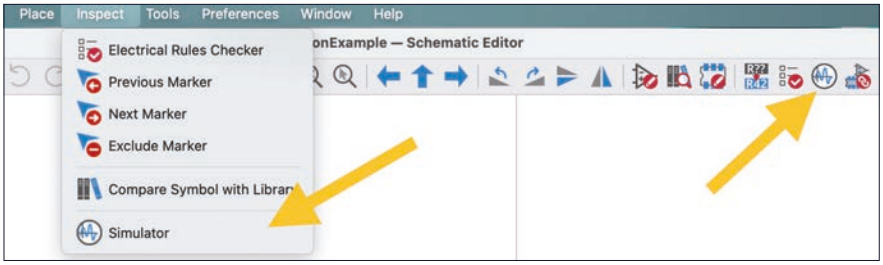


Figure 5.21.1: The circuit simulator in the schematic editor.

The simulator has a steep learning curve (there are entire books dedicated to SPICE and its variants). Still, once you learn the basics, you will be able to do circuit analysis such as the one in Figure 5.21.2 (below):

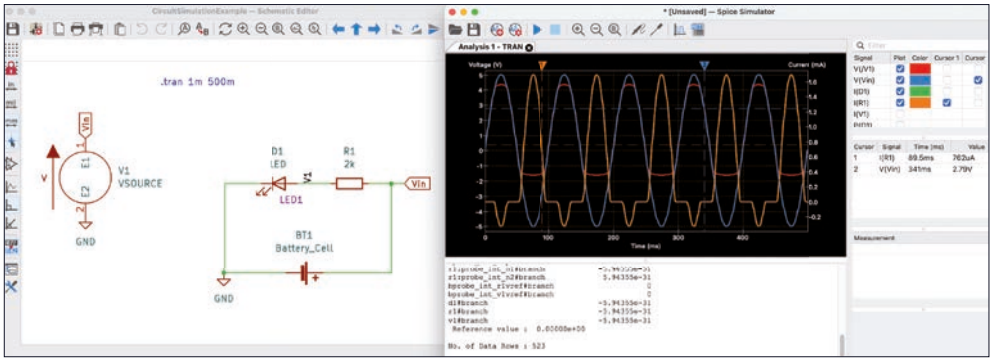


Figure 5.21.2: An example simulated circuit and the analysis outcome.

In the example above, I have simulated the schematic diagram of one of the projects in this course (the "LED torch" project from my book Fundamentals and Projects). The simulation generated a graph of voltages and currents in the circuit, representing 500 ms of operation.

Unfortunately, it is not possible to simulate a standard schematic. First, you will need to modify the schematic so that the simulator can read its components. You will also need to

assign appropriate models to each circuit component and then set the simulation parameters. In the example simulation above (Figure 5.21.3), the actual file that I simulated looks like the one below (Figure 5.21.4):

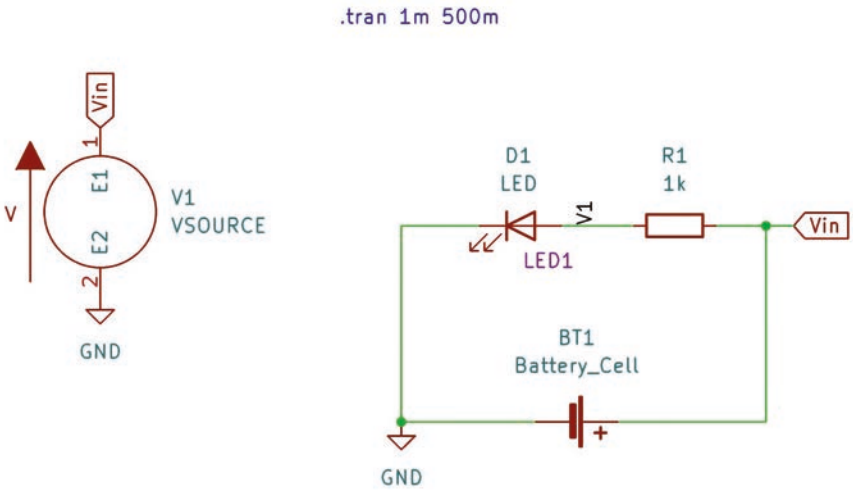


Figure 5.21.4: This circuit schematic is compatible with SPICE.

Notice that the new circuit contains a voltage source, and I removed the switch so the current can circulate. I have also added a text item above the schematic, including the custom simulation directives. I will show how to do all this later in this chapter and its segments.

Before we start, I will discuss the historical underpinnings of the circuit simulator in KiCad. The simulator integrated into KiCad is based on SPICE ("Simulation Program with Integrated Circuit Emphasis"). SPICE is an open-source analog electronic circuit simulator. Its development began almost 50 years ago at Berkley University and was originally written in Fortran. The last version of SPICE from Berkey University was published in 1993. Since then, SPICE has seeded several open-source successors, like XSPICE and CIDER. Commercial versions such as ISPICE, HSPICE, and LTSpice have also been published.

KiCad uses the ngspice variant, which is open-source. KiCad's spice implementation can use models created for the original Spice, as well as LTSpice, PSpice, and HSpice. A model is a file that contains simulation instructions for a component. There are model files for any element in a circuit, such as resistors, transistors, and voltage sources. KiCad's simulator compatibility with models from the various variants for Spice means you can use models published by many authors and companies.

Resources

In this chapter, I will give you a demonstration of the circuit simulator in SPICE. As SPICE is a complicated tool, I will certainly not cover more than some basics. To learn more, I suggest you use these resources:

1. KiCad Spice documentation[28].
2. Ngspice documentation[29].
3. The Spice page @ Berkeley[30].
4. The Spice circuit elements and models[31] (also @ Berkeley).
5. The presentation slides from course ECE220, "Introduction to spice source files[32]" (California State Polytechnic University, Pomona).

5.21.1. Prepare the circuit for simulation

To use KiCad's circuit simulator, the first task is to edit the schematic analyzed by the SPICE simulation engine.

In my example (see Figure 5.21.1.1 below), I have made a couple of small modifications:

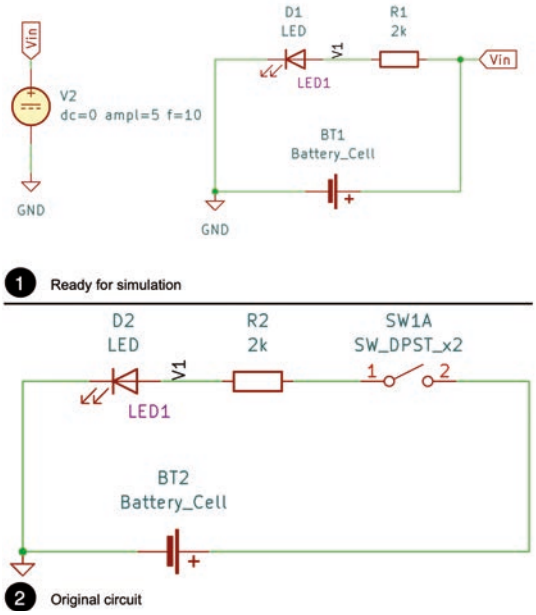


Figure 5.21.1.1: SPICE-compatible schematic (top #1), original schematic (bottom #2).

The first modification is the voltage source. Many components (mainly passive ones, like resistors and capacitors) have symbols with SPICE models already defined. But others, such as the battery symbol in the example above, are not. It is possible to define a custom simulation model for symbols, but in the case of the battery, the simplest way to deal with it is to insert a voltage source symbol from the SPICE library. The voltage source symbol contains configuration parameters the simulator can use in its analysis.

There are two steps to set the voltage source for this circuit:

28 https://www.kicad.org/discover/spice/
29 http://ngspice.sourceforge.net/
30 http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/
31 http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/UserGuide/elements_fr.html
32 https://www.cpp.edu/~prnelson/courses/ece220/220-spice-notes.pdf

1. Disable the battery cell for the simulation since I will not provide it with a SPICE model.
2. Add a voltage source and use either wires or labels to connect it to the circuit (thus replacing the battery cell).

To disable the battery cell from the simulation, double-click on the battery symbol to bring up its properties window, and click on the "Exclude from simulation" checkbox in the Attributes group:

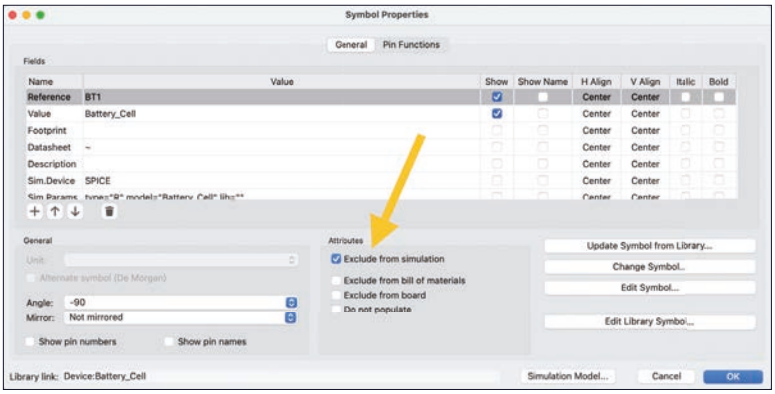
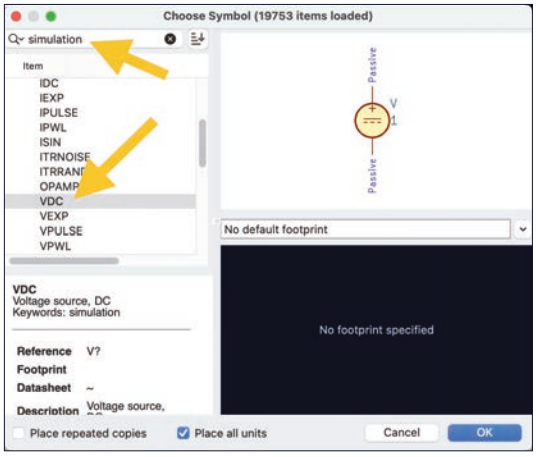


Figure 5.21.1.2: Disable the battery symbol for the simulator.

Next, you must provide a voltage source for the circuit symbol to replace the disabled battery symbol with a compatible SPICE. For this purpose, KiCad offers a library of Spice symbols. Choose the symbol tool from the right toolbar of the schematic editor, and search for the SPICE. One of the symbols in that library is "VDC" (see below):



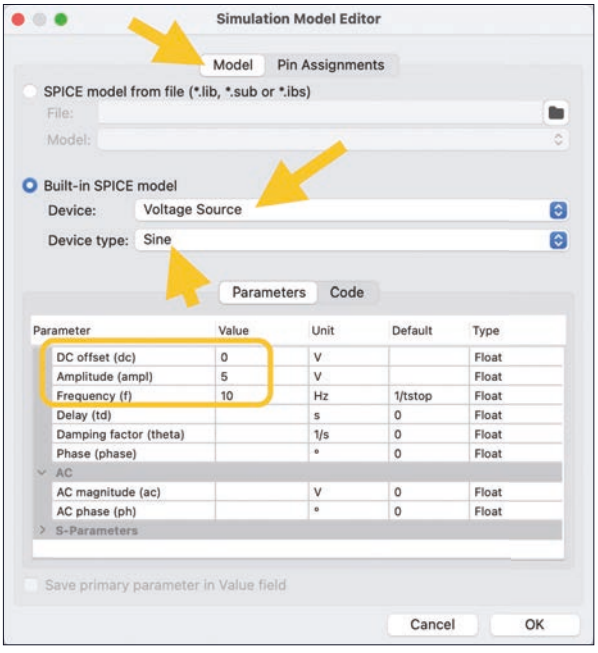
5.21.1.3: A voltage source symbol that is compatible with the simulator.

Do the wiring after you add the new voltage source in the editor. I prefer to use labels for a cleaner look. In Figure 5.21.1.1, notice how I have attached the "Vin" label and the GND symbol to the voltage source and the rest of the circuit schematic around the original battery cell symbol.

Let's examine the voltage source symbol. Double-click on it to bring up its properties window, and click the "Simulation Model" button (bottom right corner of the window) to open the SPICE model editor window. Every symbol in KiCad has a SPICE model editor that you can use to set various parameters related to the simulation, including attaching code representing the simulation model of the real-life component.

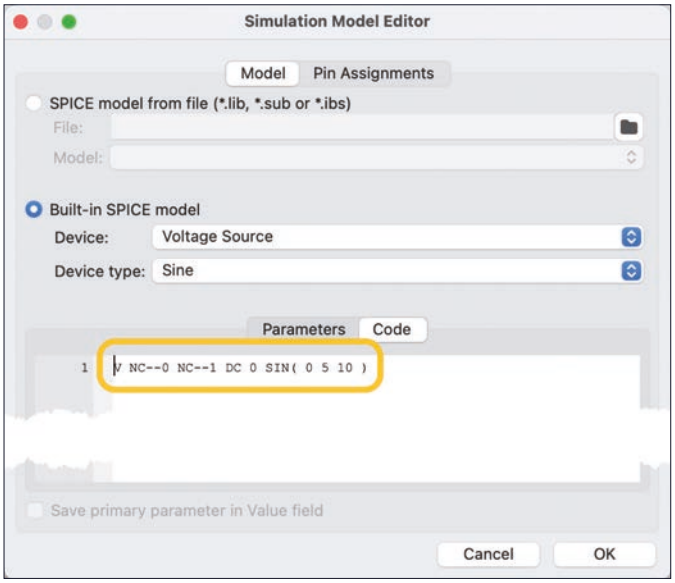
The SPICE model editor window contains two tabs: Model and Pin Assignments. Click on the Model tab. This is where you can configure this symbol (which happens to be a voltage source) for simulation. Explore the various options in the built-in SPICE model group to see the different available devices and types. For example, you can set the source to a pulse, sinusoidal, exponential, or random pattern (to name but a few). Of course, you can put it as a simple DC or AC source. You can also change the device type to something different, such as a transistor, diode, or capacitor.

Below I have set the voltage source to produce a sinusoidal (sine) output with 5V amplitude at 10 Hz (Figure 5.21.1.4).



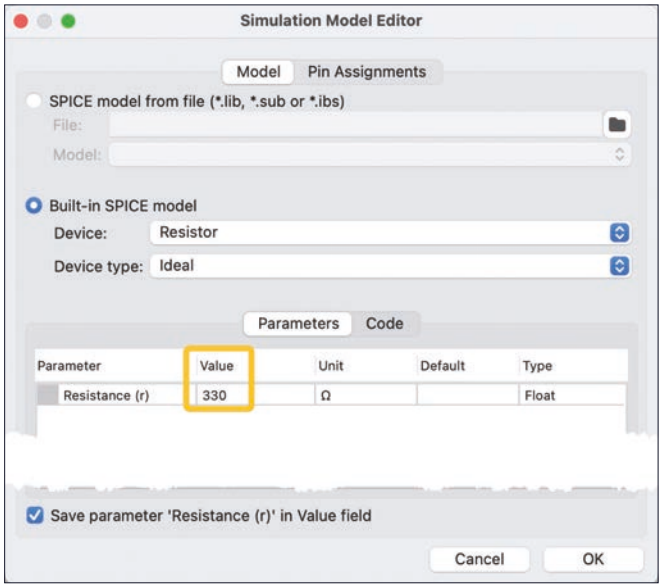
5.21.1.5: The simulation configuration for the voltage source.

As you define the various parameters, the SPICE code for this symbol is constructed in the background. You can inspect the SPICE model of the symbol in the Code tab (see below):



5.21.1.6: The Spice code for the voltage source symbol.

Let's look at the other components in the circuit. First, the resistor. Below, you can see its simulation SPICE model (see below):

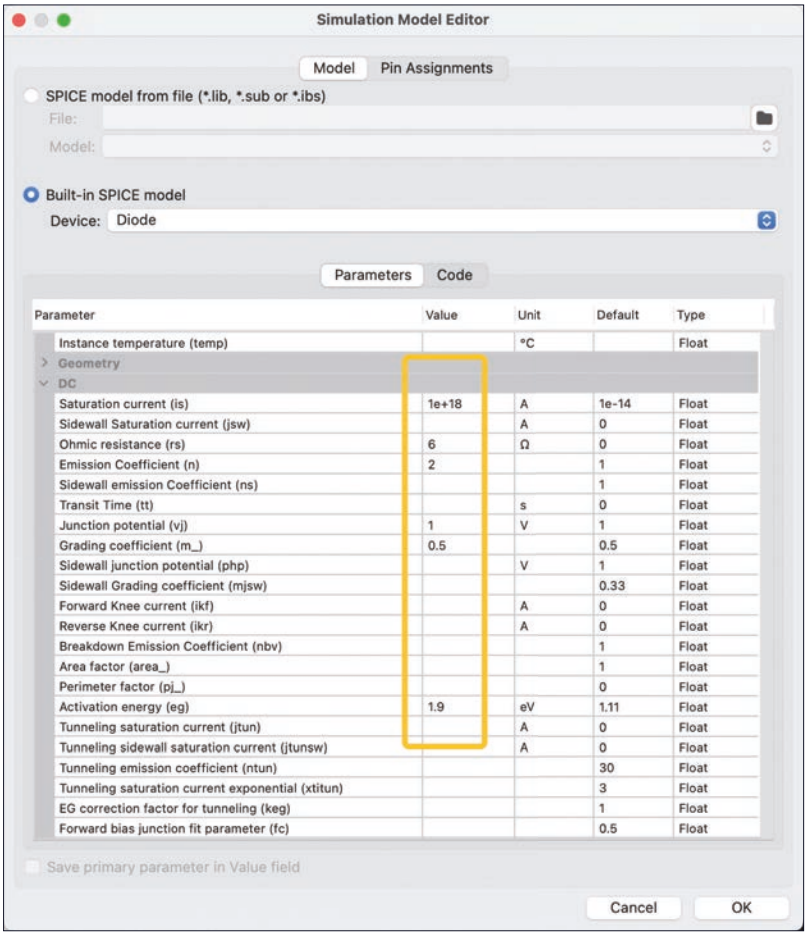


5.21.1.7: The Spice model for the resistor.

In the "Model" tab, select the Built-in SPICE model group, and set Device to "Resistor" and "Type" to "Ideal"—type "330" in the "Value" field in the Parameters table. If the resistor value is already defined in the symbol properties, you will find that the SPICE model value

for the resistance is already populated. If I want to set a 1kΩ resistor, I will type "1k" (notice the SPICE unit symbols table below the fields). In the Code tab, you will see the SPICE code for this symbol: "R NC--0 NC--1 330". Click OK and OK to return to the editor.

Let's work on the LED next. The LED is a semiconductor component, not passive. For this reason, I will need to specify a compatible SPICE model. The SPICE simulator in KiCad offers a diode device, which provides myriad configuration parameters. See the screenshot below for an example of some of them. In that screenshot, I have entered several values that are typical for a 5mm red LED that I plan to use in this circuit.



5.21.1.8: The diode model in SPICE provides many parameters to customize the device.

To determine the parameters for the LED, you will need to do research. You can find typical values for regular 5mm red LEDs or precise values if you know the manufacturer and the manufacturer publishes them. You may also find complete models, as I will show you shortly. Here is a list of typical SPICE model parameters for a 5V red LED:

- IS (Saturation Current): This is usually a very small value, as it represents the leakage current in reverse bias, which is negligible for LEDs. A typical value might be around 1E-12 A (1 picoampere).
- N (Emission Coefficient): This factor describes the ideality of the diode, often close to 2 for LEDs (typical diodes have an N value of about 1 to 2).
- RS (Series Resistance): Represents the ohmic resistance internal to the LED. For a standard red LED, this could be around 6 ohms.
- BV (Reverse Breakdown Voltage): LEDs are not usually operated in reverse bias at the breakdown, but this parameter can be set to a high value, such as 5 volts or higher, to avoid a reverse breakdown during simulations.
- IBV (Current at Reverse Breakdown Voltage): Corresponding to the BV, a small current such as 1E-10 A might be typical.
- CJO (Junction Capacitance): This is typically low for small signal diodes; around 18 pF might be appropriate.
- M (Grading Coefficient): Describes how the junction capacitance changes with voltage, typically around 0.5 for an LED.
- VJ (Junction Potential): Usually around 1 volt.
- EG (Band Gap Energy): A red LED typically emits at about 1.9 eV (equivalent to a forward voltage of about 1.9 V to 2.2 V in operation), so this value might be set closer to 1.9 eV.

You can copy these values into the parameters fields one at a time. You can also copy this SPICE model expression in the Code tab text field:

```
.model RedLED D (IS=1E-12 N=2 RS=6 BV=5 IBV=1E-10 CJO=18p M=0.5 VJ=1 EG=1.9)
```

You can find SPICE models for components by researching on the Internet, or by writing them yourself. Component manufacturers often publish SPICE models with their products (examples: Analog Devices, Microchip, National Instruments, PSpice, Littlefuse, Diodes). Universities or individuals publish models on their websites (Examples: Berkeley University, All About Circuits).

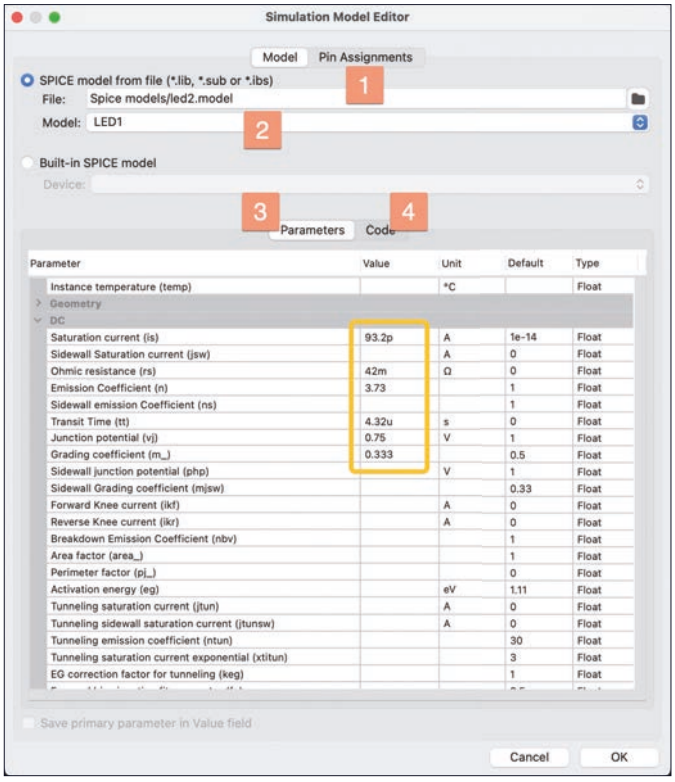
Models are distributed as text files like this (here is my source[33]):

```
*Typ RED GaAs LED: Vf=1.7V Vr=4V If=40mA trr=3uS
.MODEL LED1 D (IS=93.2P RS=42M N=3.73 BV=4 IBV=10U
+ CJO=2.97P VJ=.75 M=.333 TT=4.32U)

*Typ RED,GREEN,YELLOW,AMBER GaAs LED: Vf=2.1V Vr=4V If=40mA trr=3uS
.MODEL LED2 D (IS=93.1P RS=42M N=4.61 BV=4 IBV=10U
+ CJO=2.97P VJ=.75 M=.333 TT=4.32U)

*Typ BLUE SiC LED: Vf=3.4V Vr=5V If=40mA trr=3uS
.MODEL LED3 D (IS=93.1P RS=42M N=7.47 BV=5 IBV=30U
+ CJO=2.97P VJ=.75 M=.333 TT=4.32U)
```

33 https://newbedev.com/simple-led-model-for-circuit-simulation

You can save this model in a text file and import it to the KiCad symbol using the SPICE model editor. I saved the file named "led2.model" within the Spice Models directory inside my KiCad project directory.

For the LED symbol, double-click on it to bring up its Properties window, then click on "Spice Model." Once the SPICE Model Editor window is up, click on the Model tab and load the Spice model text file you created using the File widget under "SPICE model from file." Because this spice model file contains three models, use the Model widget to select "LED1" (see below):



5.21.1.9: Setting the Spice model for the LED symbol.

In the screenshot above, I have used the file method. I saved the model in a text file titled "led2.model" and used the file browser to find it and select ("1" in Figure 5.21.1.9).

The model file contains three individual models. Each has a unique name: "LED1", "LED2", and "LED3". I selected the model I wanted to use using the Model drop-down menu ("2" in Figure 5.21.1.9).

The Code tab shows the SPICE code that KiCad will use to simulate the LED symbol. In the Parameters tab, you can see the mode values in the corresponding parameter field. The various parameters in the model definitions, such as "IBV" (Current at Breakdown Voltage), "BV" (Reverse Breakdown Voltage), and "TT" (Transit-time), are described in the Ngspice user manual. You can find these values in a component's datasheet and use them in your models.

Click OK and OK to get back to the editor. The circuit schematic is now ready for the simulation. In the next segment of this chapter, I will show you how to configure the simulator.

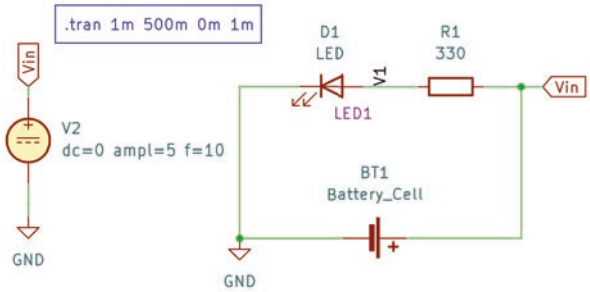
5.21.2. Configure the simulator

I prepared my circuit schematic for the simulator's analysis in the previous segment. In this segment, I will configure the simulator. Compared to the work needed to prepare the circuit for the simulation, the simulator's configuration is much simpler.

To configure the simulator, there are generally two steps to complete:

- First, set the SPICE simulation settings. These settings control the number of simulation points to calculate, the start and stop frequencies (for AC voltage sources), the DC power source(s) behavior, the time step, and initial/final times.
- Second, the simulation probes are set to capture the simulation voltage and current data.

A convenient way to configure the simulator so that the settings persist is to create a text item above the circuit schematic and type the configuration string there. In Figure 5.21.2.1 (below), you can see the configuration string inside the blue box:



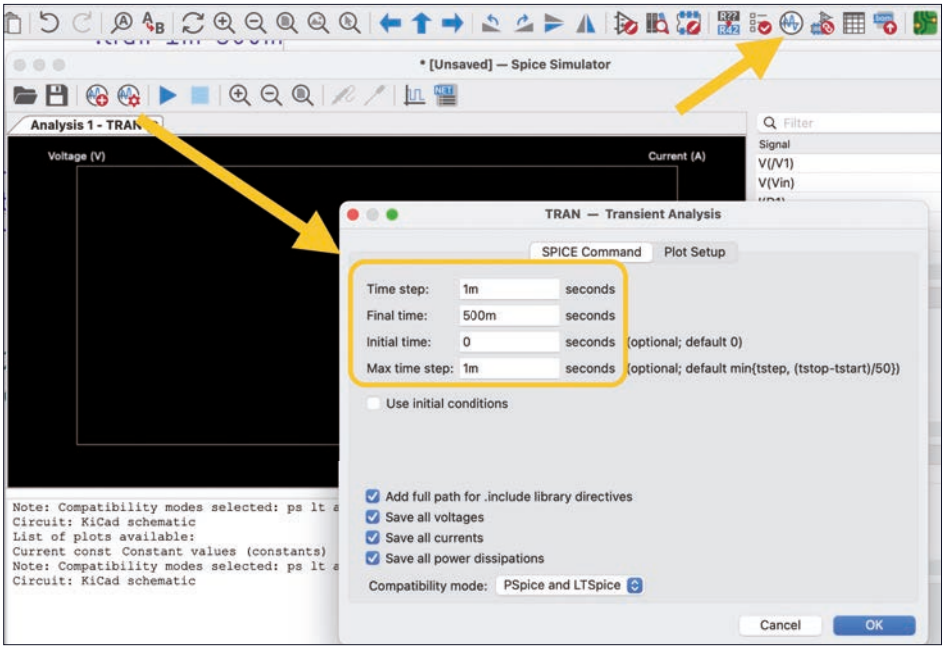
5.21.2.1: The simulation configuration.

The simulator will detect the presence of the configuration string and read it, saving you the hassle of manual configuration every time you open the schematic editor. Let's break down the configuration instructions:

- **.tran:** statement specifies a transient analysis, which is used to analyze the behavior of a circuit over time

- **1m**: This is the time step of the simulation. It tells the simulator to calculate the circuit's response every 1 millisecond (ms). A smaller time step can provide more detail and greater accuracy in the simulation results and increase computation time.
- **500m**: This is the total time the simulation will run, specified here as 500 milliseconds (ms). The simulation will provide output data from time zero up to 500 ms.
- **0m:** The initial printing delay time (when to start recording data) here is 0 milliseconds (immediate start).
- **1m**: The maximum time step allowed in the simulation is 1 millisecond.

Let's see the impact that the configuration string has on the simulator. Bring up the simulator window by clicking on Inspect, Simulator. In the simulator window, click on the Sim Parameters button (Figure 5.21.2.2).



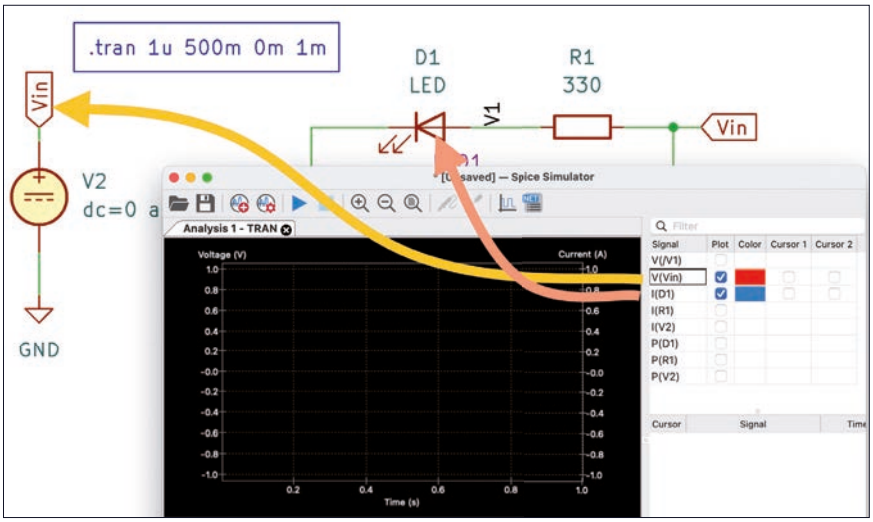
5.21.2.2: The simulation settings.

The simulation settings window will appear. The simulator has already read the information in the settings string from Figure 5.21.2.1 and has populated the "Time step, " "Final time," "Initial time" and "Max time step" text boxes in the SPICE Command tab. Compare the values in the text boxes with those in the configuration string and notice that they match. Before you close the simulation settings window, browse the contents of the Plot Setup tab to see some of the different available options.

My simulation is now configured to start from 0 ms to 500 ms at a step of 1 ms. The simulator will calculate 500 points of whichever values are set next. As the number of

points increases, so does the amount of time the simulator will need to finish the work, so choose reasonable values for the simulation time. If you change the text string in the editor, you must exit and restart the simulator to load the new configuration.

I am almost ready to start the simulation. The last task is to indicate the signals to capture. For example, I want to capture the current from the LED and plot it against the input voltage. To do this, open the Spice Simulator window and check the V(Vin) and I(D1) checkboxes (see below). You can also customize the plot color for each signal.

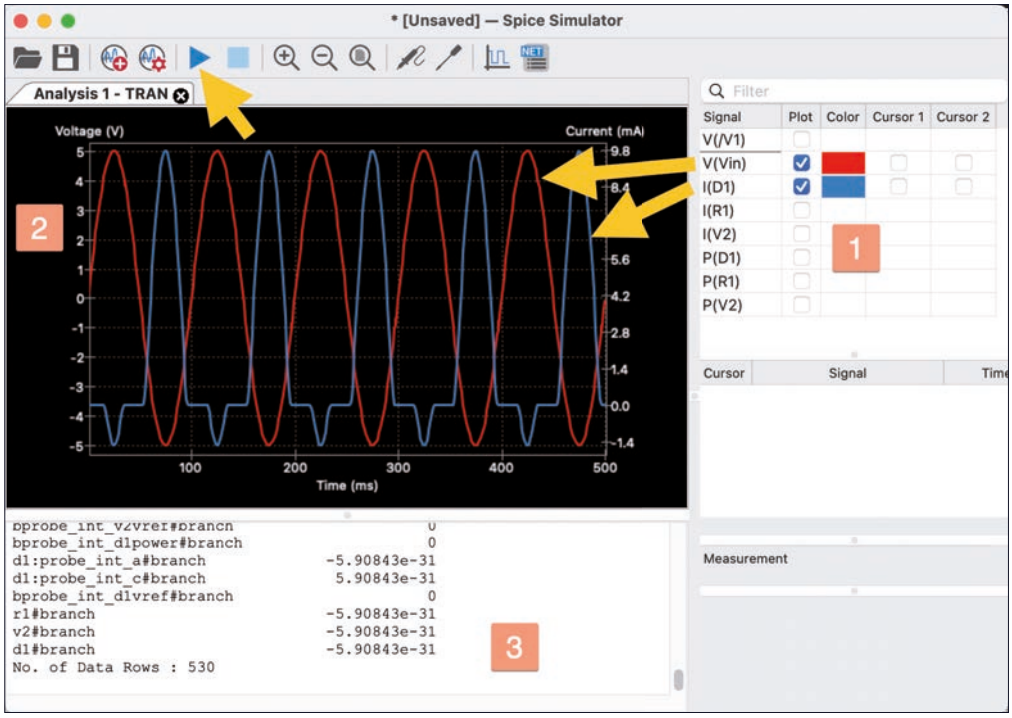


5.21.2.3: The simulator will capture the signal "I(D1)".

The simulator is now ready to run. Let's do that in the next segment of this chapter.

5.21.3. Simulate

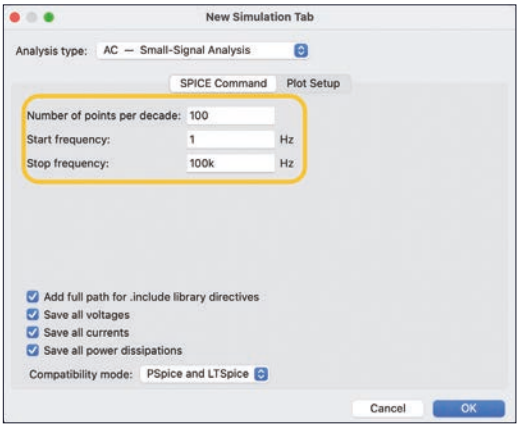
It's time to simulate! If the simulator isn't open, open it and check that the V(Vin) and I(D1) signals are checked for plotting. Then, click the blue play button to run the simulation (see below).



5.21.3.1: The simulation results.

My simulation only contains 500 points, so it is rapid to finish. In the screenshot above, you can see the simulated captured signals (1), the time plot (2), and the results (3).

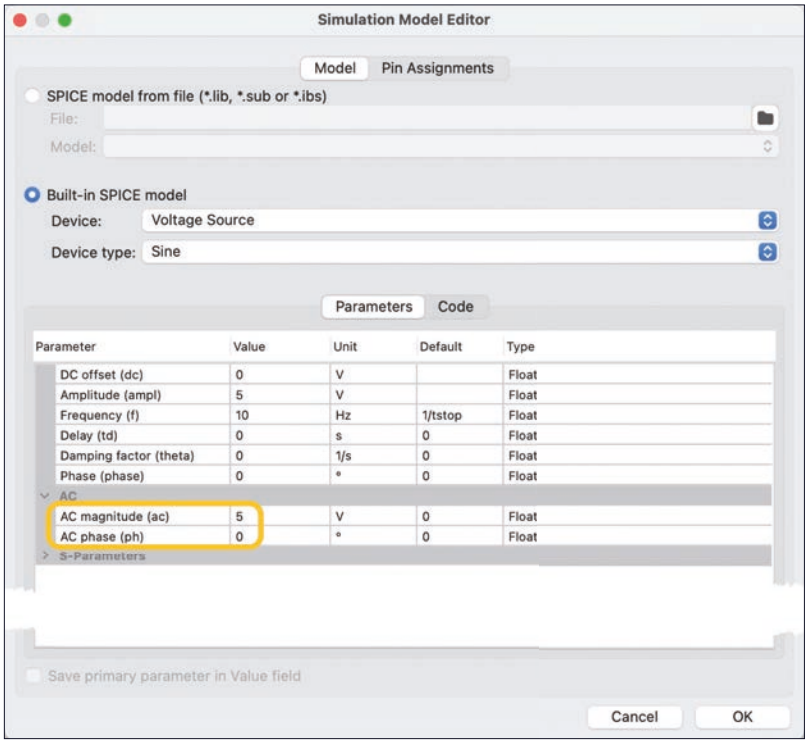
You can create additional plots as needed. With the simulation window active, click the "New Analysis Tab" button in the toolbar. A new simulation tab configuration window will appear. In this tab, let's do an AC analysis. You can see my setup below:



5.21.3.2: AC analysis SPICE settings.

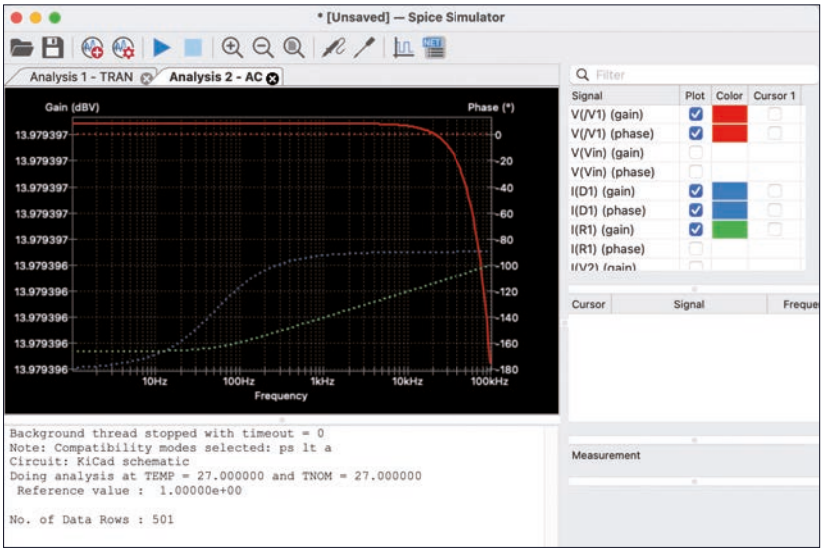
Click OK to close the window. You will see a new tab in the simulator. If you click the play button, you will see a flat line. This is not what you expect in an AC circuit analysis.

An AC analysis requires at least one AC source in the circuit. This could be a voltage or current source. I already have a voltage source in my circuit that is set as a "Sine" device type, but I must configure its AC magnitude and phase parameters before using it with the AC analysis type. You can see my voltage source simulation model parameters below.



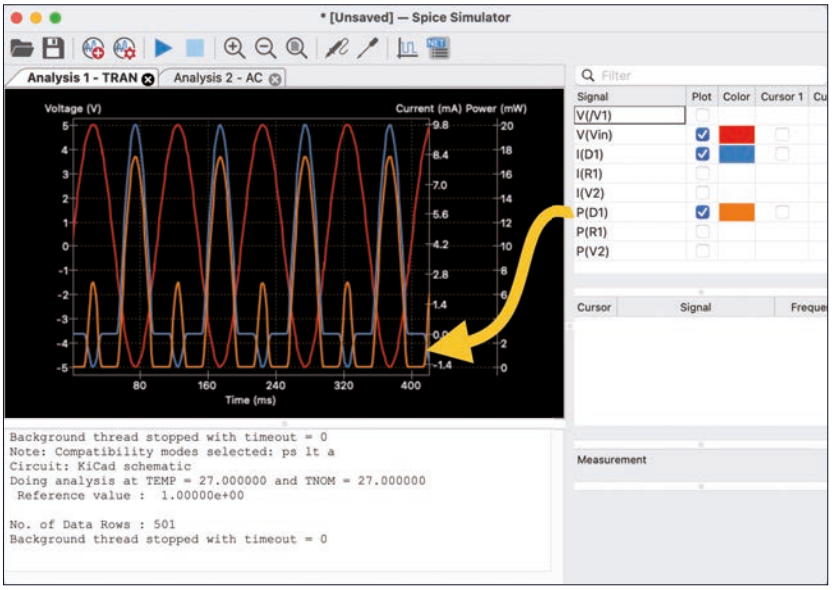
5.21.3.3: Setup the voltage source for AC analysis.

In the voltage source, I set the AC magnitude to 5V and the AC phase to 0°. Now, my circuit is ready for AC analysis. Return to the simulator window and click the play button to run the AC analysis. Notice my selected signals and the plots in the screenshot below.



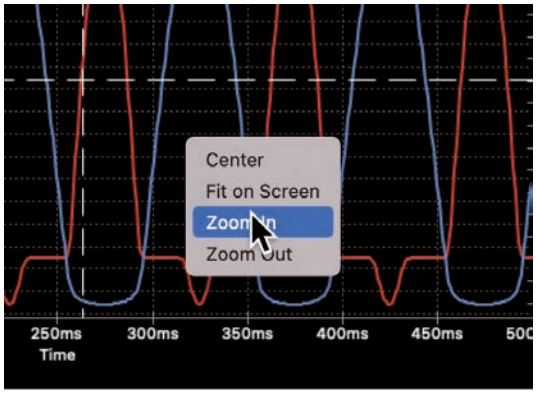
5.21.3.4: The result of the AC analysis.

Let's return to the transient analysis and add a new signal. With the new plot tab selected, check the P(D1) box from the signals list. The simulator will immediately draw the plot for the three signals (Figure 5.21.3.5):



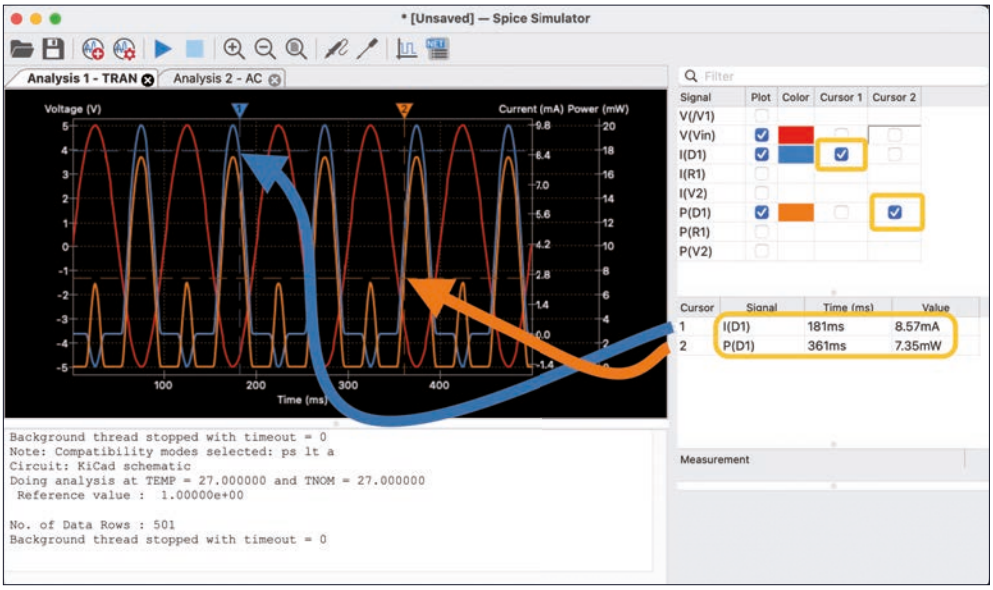
5.21.3.5: Plot for signal P(D1) in a new tab.

The simulator plots are interactive. Use your mouse to click and define a boxed area to zoom in. You can also use the mouse scroll wheel to zoom in and out and the middle button to pan.



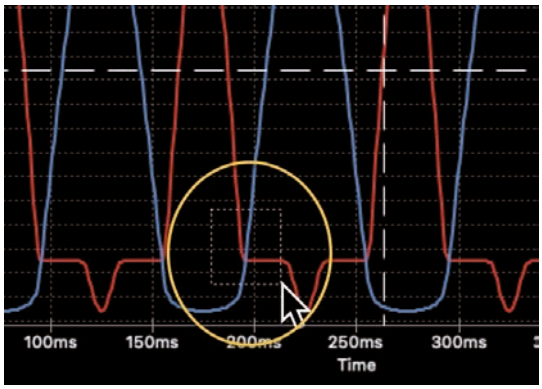
5.21.3.6: Context menu options for the plot.

Like an oscilloscope, the simulator has cursors. You can assign two cursors to any of the enabled signals and move the cursors with your mouse to get precise readings along any part of the plot.



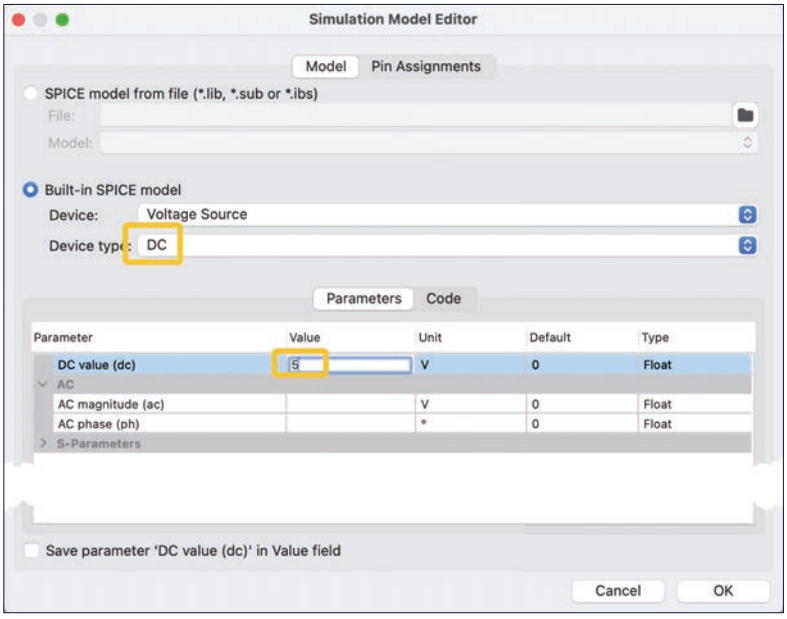
5.21.3.7: Moving the time line for cursor Vin.

You can also zoom into a specific region of the plot by drawing a rectangle with your mouse:



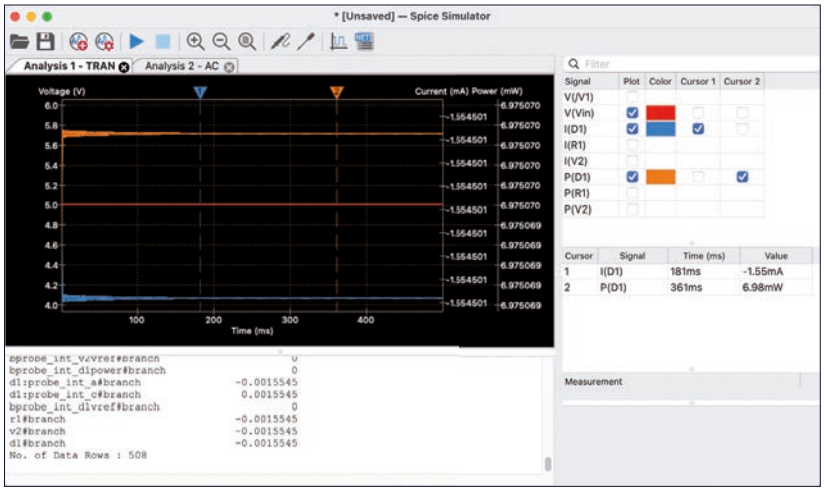
5.21.3.8: Zoom in to a region of the plot.

Let's make a change to our circuit and rerun the simulation. I will change the voltage source configuration to a 5 V stable source for the new experiment. Make the change in the voltage source symbol like this (see DC field in "Source"):



5.21.3.9: Changed the voltage source to 5 V stable.

Return to the simulator window and click the blue play button to run the simulation. There is no need to change any of the simulator settings. The new plot looks like this:



5.21.3.10: A new plot with the new voltage source settings.

With these basic simulator skills, you can try out a variety of scenarios. For example:

- What would happen to the voltage and current on the diode if the source voltage is 3V instead of 5V?
- What if it is a small AC voltage at 10Hz?
- What if you used a different type of LED?
- What if you double the value of the resistor?

These questions and many more are now easy to answer with the help of the simulator.

5.22. KiCad project templates

KiCad project templates can save you a lot of time when you start a new project. With a KiCad project template, your new project is pre-configured. Its schematic layout editors are already populated with standard components that can form the basis for your project.

You can choose one of several templates that KiCad ships with or create your own. You can also create a user template from your existing KiCad projects. In this chapter, I'll show you both options.

Template locations

KiCad system and user templates are stored in specific directories. You can find and/or change those directories in your instance of KiCad in the "Configuration Paths" window (Figure 5.22.1). While in the KiCad project window, click on Preferences and Configure Paths.

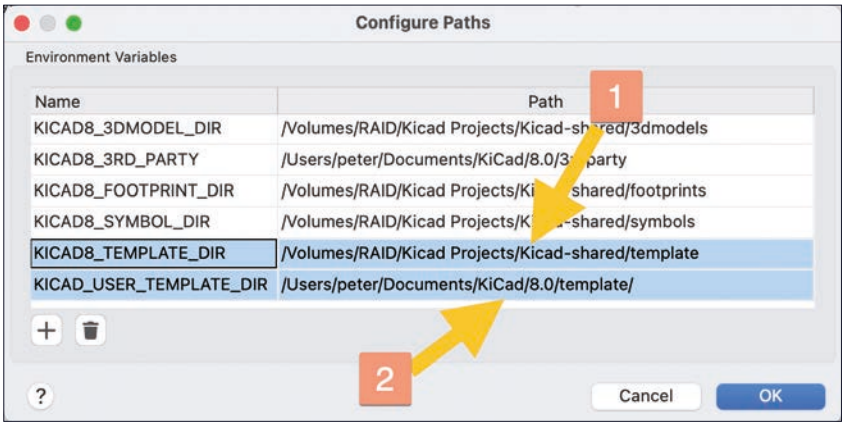


Figure 5.22.1: The Configure Paths window.

Notice that there are two template directories:

- KICAD8_TEMPLATE_DIR, where you can find the system templates.
- KICAD_USER_TEMPLATE_DIR, where you can store your custom templates.

The source of the system project templates is Gitlab. You can periodically look at Gitlab for new or updated templates and copy them to your KiCad instance system template directories.

Later in this chapter, I will show you how to create a custom project template. I will save mine in the specified KICAD_USER_TEMPLATE_DIR directory.

5.22.1. Using a system project template

A "System template" is a template that KiCad provides. For example, system templates contain Arduino Mega, Beaglebone Black, Raspberry Pi, or STM32 projects. Let's look at an example. I will create a new project using the Arduino Pro Mini system template.

Start KiCad and open the main project window. From the File menu, select "New Project from Template…" (Figure 5.22.1.1).

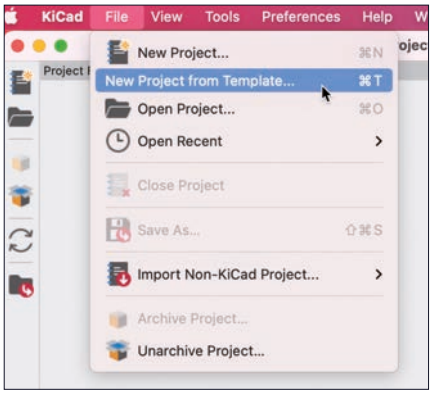


Figure 5.22.1.1: Create a new project from a template.

The Project Template Selector window will appear (Figure 5.22.1.2). Select the "System Templates" tab (1), use the scroller to pan the project templates left and right, and find the Arduino Pro Mini template. Click to select it (2).

In the information pane (3), you will see a description of the template, including a view of its footprints and schematics. The information you see on the information page (3) is a simple HTML file. The template's creator can provide as much or as little information as they want.

Notice the location of the selected template in the folder text box (4). Click OK (5) to continue.

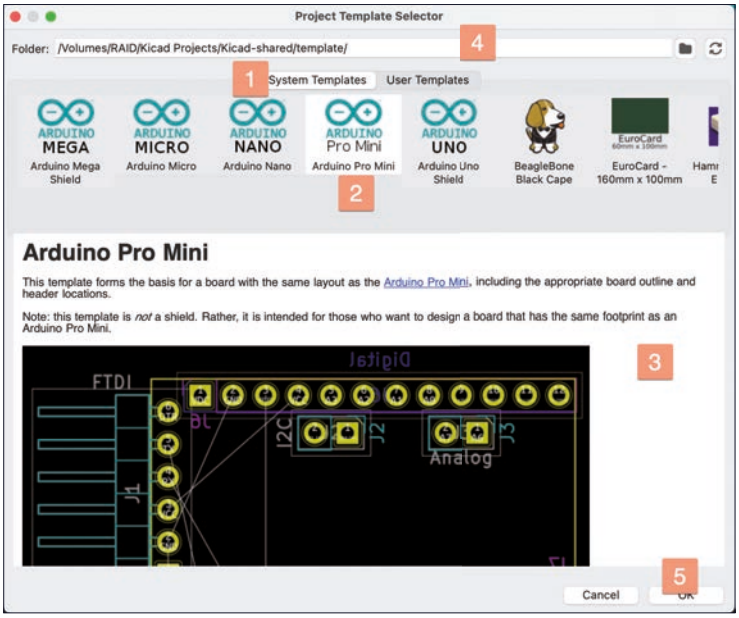


Figure 5.22.1.2: The project template selector.

KiCad will ask you to choose a location for the new (Figure 5.22.1.3). Give the project a name (1), check the new folder box (2), and click "save" (3).

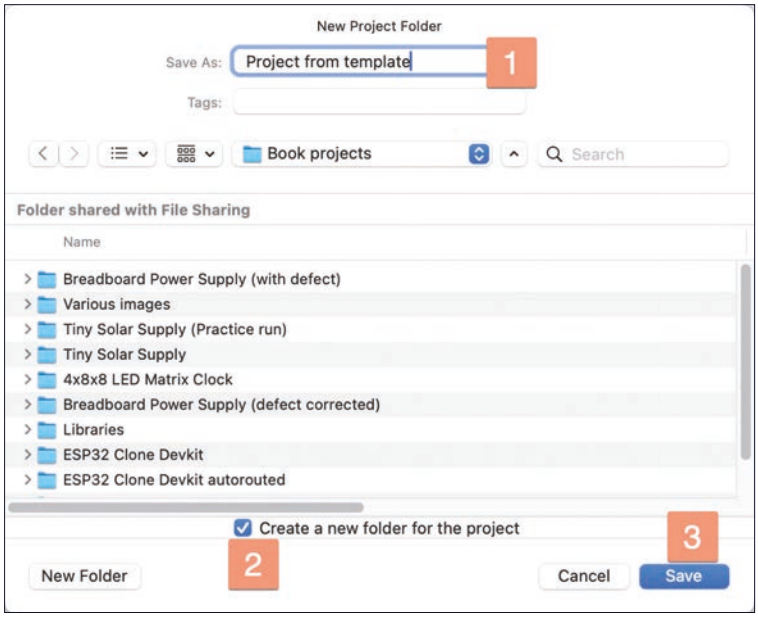


Figure 5.22.1.3: The location for the new project.

The new project is ready using the settings and starting schematic and layout from the project template. Open the schematic and layout editors to see what the project looks like at the moment (Figure 5.22.1.4):

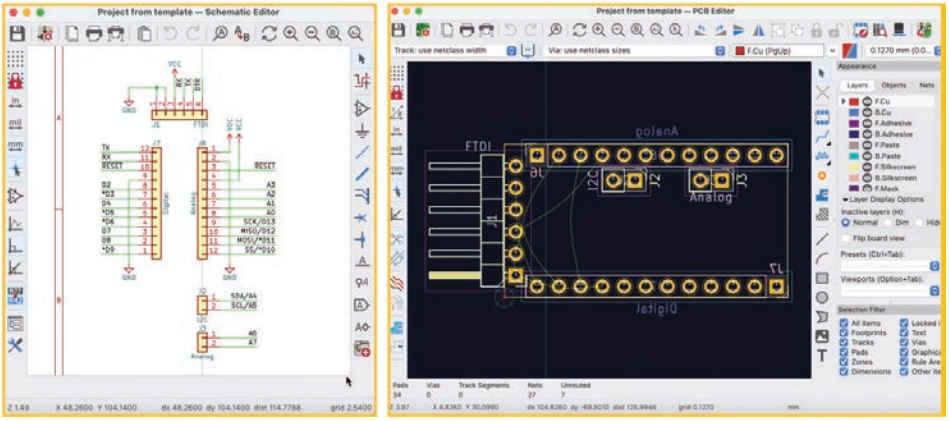
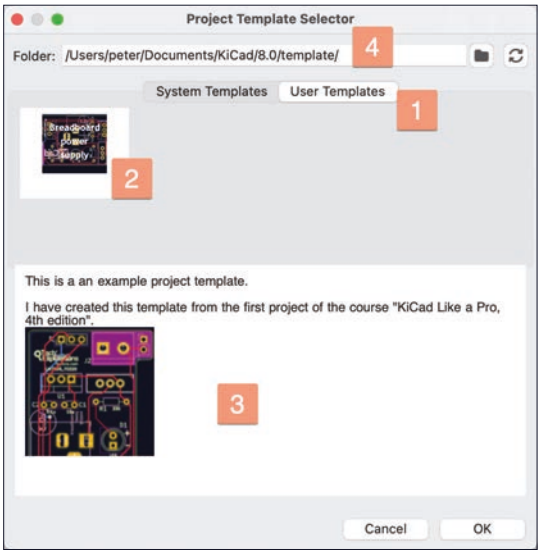


Figure 5.22.1.4: The new project, with the two editors populated.

As you can see, the new project is not blank. It contains the starting schematic and layout inherited by the Arduino Mini template.

5.22.2. Create a user project template

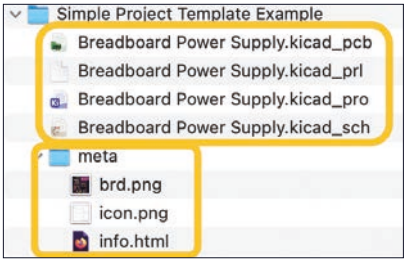
It is possible to create custom user templates from your existing KiCad projects. When you have created a user project template, it will be available for use in the User Templates tab of the Project Template Selector window:



5.22.2.1: The User Templates tab.

In the example above, I have selected the User Templates tab (1) to see a user project template (2). The information pane (3) contains a simple HTML document with rudimentary formatting. Also, notice the location of the template (4).

Let's look inside the user template directory for this template. Doing so will give us clues about the necessary template files and their structure. See Figure 5.22.2.2 (below).



5.22.2.2: The contents of a user project template directory.

Two groups of files compose a user project template:

1. Project meta.
2. Project files.

The project metafiles are stored in a folder named "meta." At a minimum, the meta folder must contain "icon.png" and "info.html." The PNG file is used to create an icon for the template (see item "2" in Figure 5.22.2.1). The PNG image must rectangular. I use a resolution of 128x128 pixels.

The HTML file composes the contents of the information pane (see item "3" in Figure 5.22.2.1). You can include additional files in the meta folder if you wish. For example, I have added a file named "brd.png" to display a small layout image in the information window. I use the file "brd.png" in my HTML code. The project files are the three primary files of any KiCad project:

- The project file, ".kicad_pro".
- The schematic file, ".kicad_sch".
- The layout file, ".kicad_pcb".

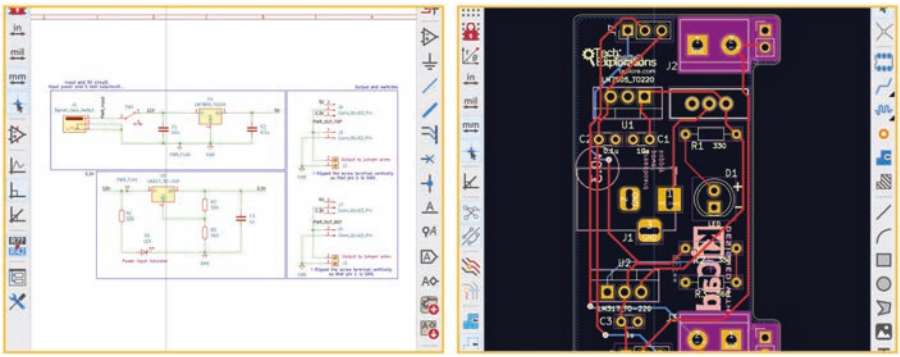
You can get those files from your source KiCad project and copy them into the project template folder. To create this project template for this example, I copied the files I listed above from my LED torch project. I did not make any changes to those files. Below, you can see the contents of the HTML file:

```
<html>
<head></head>
<body>
<p>This is a an example project template.</p>
<p>I have created this template from the
first project of the course "KiCad Like a
Pro, 3rd edition".</p>
<br>
<img src="brd.png">
<br>
<img src="icon.png">
</body>
</html>
```

The paths are all relative to the location of the HTML file. You can also use regular HTTP links to external web pages.

Once you have created the new project template and stored it in the appropriate location, you can use it. In the KiCad project window, click File, and select "New Project from Template…". The Project Template Selector will appear (see Figure 5.22.2.1 above). Select the "User Templates" tab, and click on the project template icon. Then click "OK" and choose a name and location for the new project.

Your new project is now ready to use, with its schematic and layout editor populated with the symbols and footprints inherited from the template (Figure 5.22.2.3):



5.22.2.3: The new project, with the PCB and schematic editor contents inherited from my user project template.

System and user project templates can save you a lot of time when starting new projects. I often capture my new projects early in their development process to use them as templates in the future.

5.23. Share projects and configurations with others

KiCad has a project archiving feature that allows you to create a self-contained ZIP archive of your project. Once you have archived your KiCad project, you can store it for later use or share it with other people. The project archiver will keep all project dependencies (like symbols or footprints) in the archive so that your collaborators (or you) will not need to spend time fixing library issues.

In this chapter, I will show you how to archive and un-archive a project using an example. I will archive one of my projects on my Mac OS computer and open it on my Windows 11 virtual machine. I installed the same version of KiCad on both machines (KiCad 8.0.2, the newest version available when I wrote this chapter).

First, open a KiCad project. With the KiCad project window showing, click on File, then Archive Project (Figure 5.23.1).

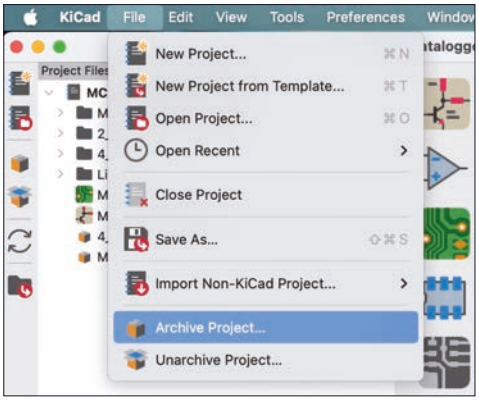


Figure 5.23.1: Archive a KiCad project.

KiCad will ask for a name and save location for the archive and then create a new ZIP file (Figure 5.23.2):

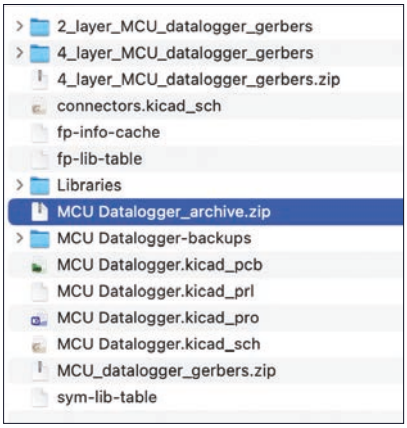


Figure 5.23.2: The new project archive in the project directory.

Next, copy the project ZIP archive to your target machine. Below, I have copied it to my Windows 11 virtual machine (or a collaborator that used KiCad on Windows). See "1" in Figure 5.23.3 below:

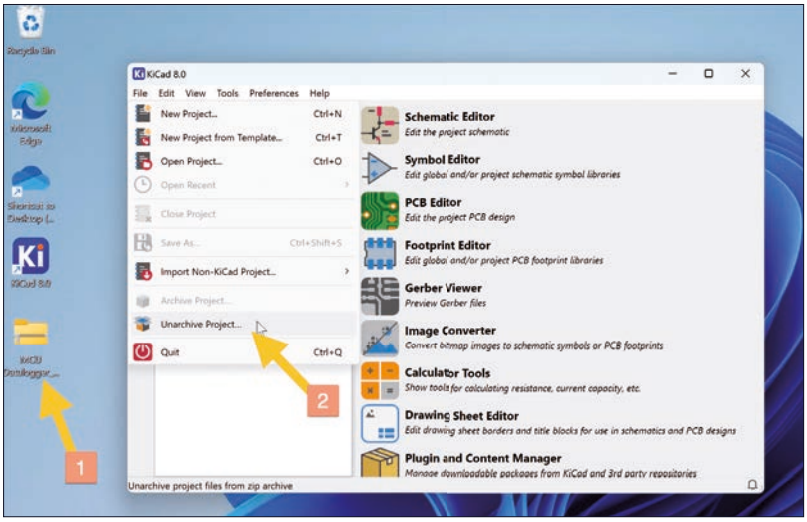


Figure 5.23.3: Unarchviging this archive in KiCad running on Windows 10.

In the KiCad project window, click on File and "Unarchive Project…". KiCad will ask you to select the archive file, so use the file browser to locate the ZIP file and click Open (Figure 5.23.4).

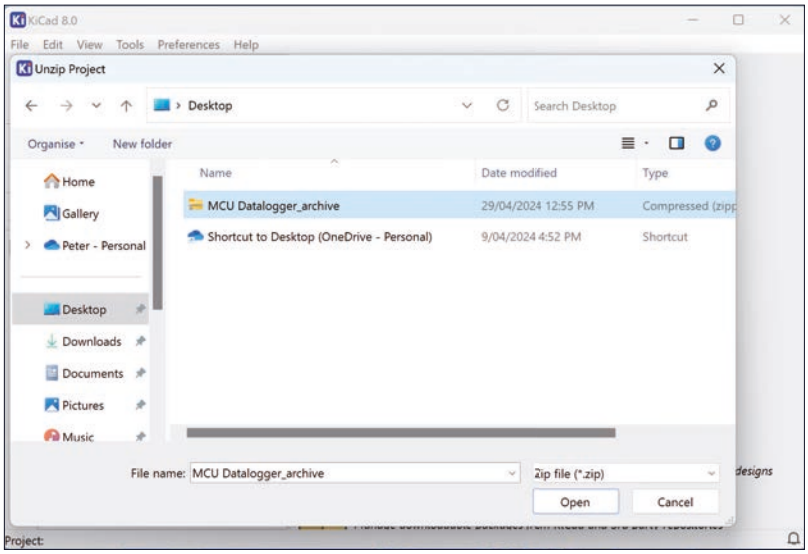


Figure 5.23.4: Navigating to the archive ZIP file.

Next, KiCad will ask you for a destination location for the unarchived project. I'll save mine to the desktop (Figure 5.23.5):

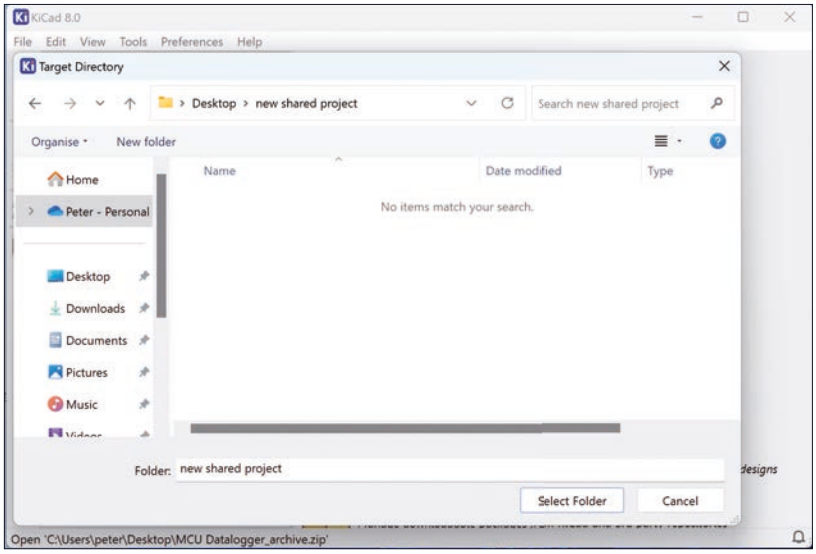


Figure 5.23.5: Select a destination for the unarchived project.

Navigate to the new folder and look inside. You will see the contents of the project that I archived on my MacOS computer (Figure 5.23.6):

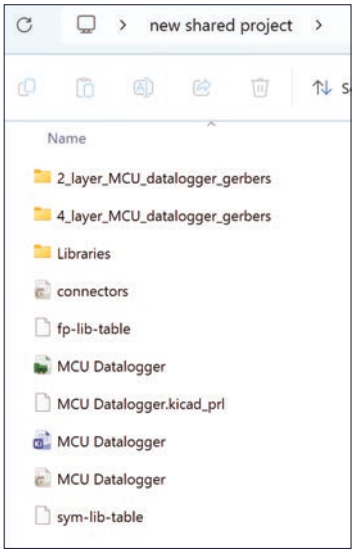


Figure 5.23.6: The contents of the unarchived project.

Still working in Windows 11, go to KiCad and open the project. You will see the project settings and the editor contents as they were when the archive was created (Figure 5.23.7).

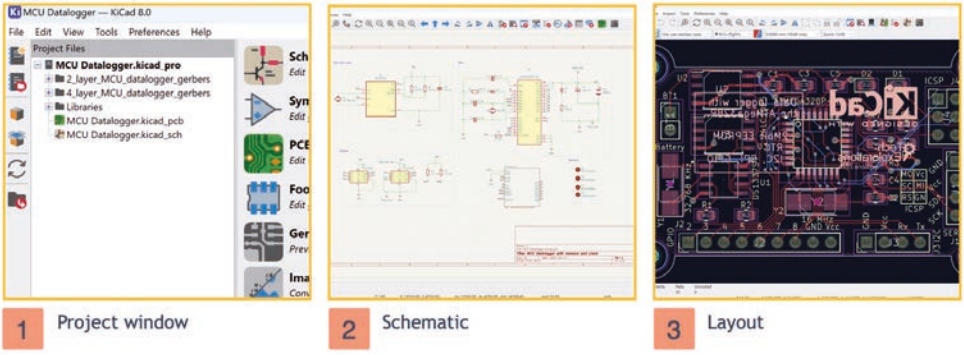


Figure 5.23.7: The project, unarchived in Windows 11.

Note, however, that the project archive does not contain any KiCad customizations. For example, if you have set custom editor themes, those will not come through with the project. It is possible to share such customizations. For example, if you want to share a custom color theme, follow this process:

1. Open the Preferences window and select the Schematic or PCB editor Colors pane.
2. Click the "Reveal Themes in Finder" button at the top right of the window. This will open the directory containing the theme JSON files.
3. Copy the file with your theme to the same directory in the target KiCad instance.

4. Restart the target KiCad instance to make the new theme available.

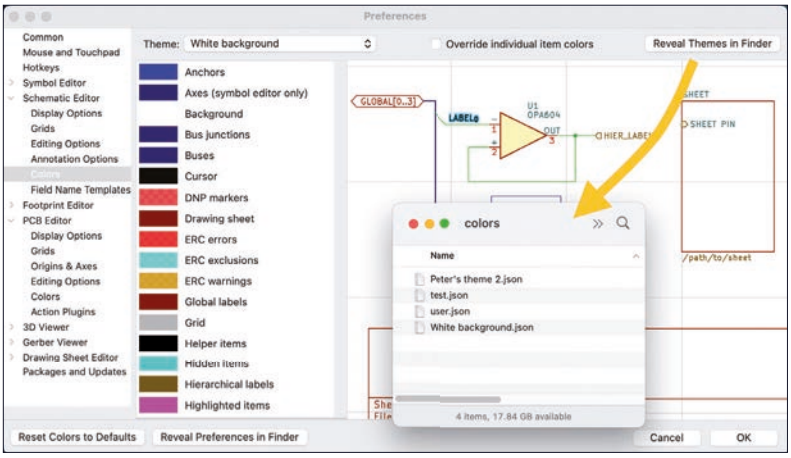


Figure 5.23.8: You can share the JSON files in the "colors" directory.

Aside from the colors, you can share many other aspects of your KiCad configuration. Look inside the Preferences directory for a complete list of sharable configurations (click "Reveal Preferences in Finder at the bottom right of the Preferences window).

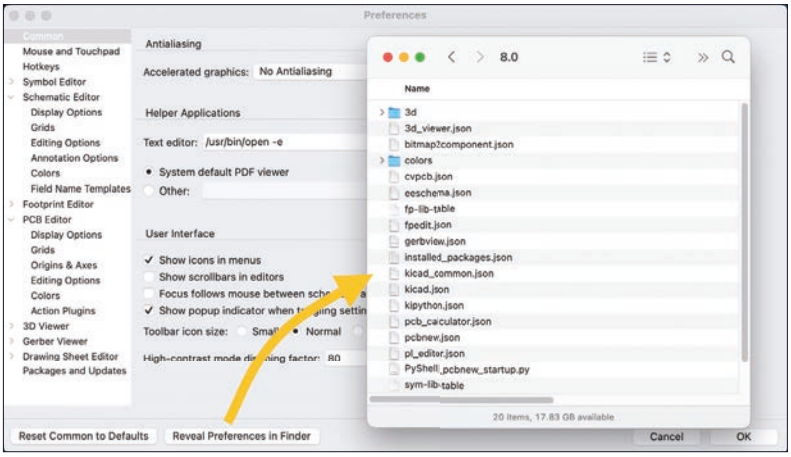


Figure 5.23.9: You can share many of your KiCad's custom configurations.

KiCad has made progress in making it easy to share entire projects and application configurations with others. You can also use these sharing capabilities to back up your KiCad work by backing up the contents of your Preferences directory.

5.24. Calculate the width of a trace

KiCad includes a calculator that you can use to precisely calculate the width of a track according to various parameters, like the current you wish to convey through the trace, its total length, and the maximum temperature rise when that current is flowing through it. You can use this calculator to find the minimum trace width, or you can rely on your experience and choose a much larger width than the standard width of signal traces.

To use the calculator, open the KiCad launcher window and click on the calculator icon(Figure 5.24.1).

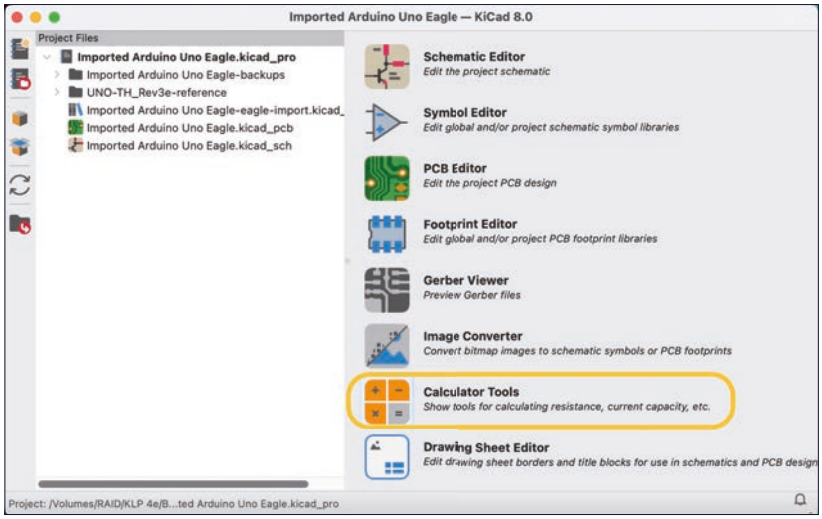


Figure 5.24.1: The calculator is available via the KiCad project window.

The calculator app contains multiple calculators. One of them is the Track Width calculator. Select it by clicking on its tab. Fill in the values that best describe your power track requirements. For a typical Arduino gadget, the values that you see in Figure 5.24.2 are reasonable. I have only altered the conductor length value to 20mm to match better the power trace length of one of my PCB projects. I overshoot these values to ensure that the trace width the calculator returns can comfortably cover the requirements.

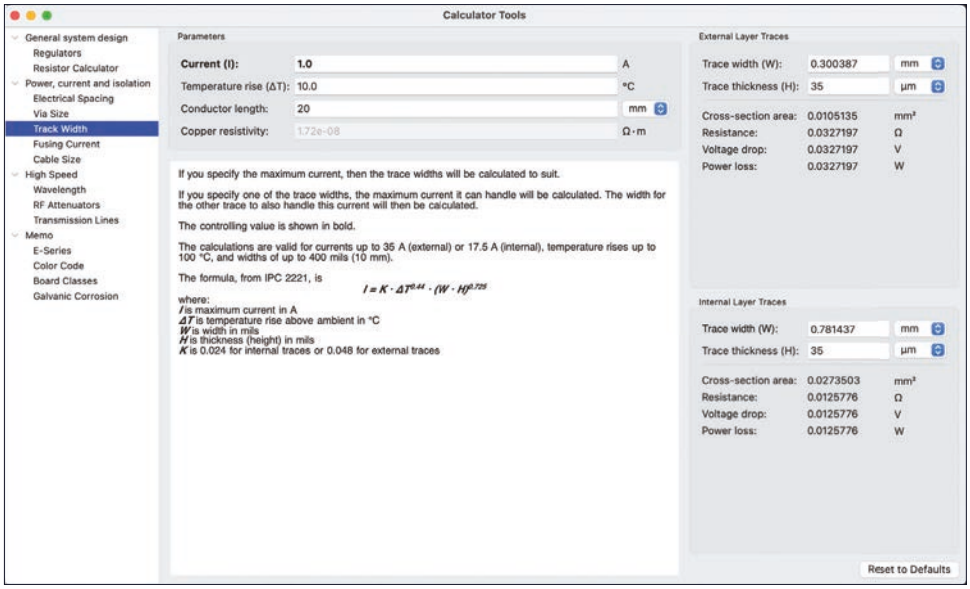


Figure 5.24.2:The Track Width calculator.

At the top right corner of the calculator, there is a field to provide the trace thickness. This value you don't have control over is defined by the manufacturer's specifications (some manufacturers allow you to select the weight of your copper trace, but for simplicity, let's assume here that this is fixed). The default value, 0.035 mm, seems to be an industry standard. Manufacturers typically make their boards with that trace thickness. To be sure, search your preferred manufacturer's website for their trace thickness or ask them.

The calculator returns the suggested trace width as you type in the parameters. In the example of 5.24.3, the suggested width is 0.30 mm.

5.25. Design a custom schematic sheet

You can create a custom version of the sheet layout. You can use your custom sheet to replace the default one in the schematic editor. A popular reason for doing that is to include a logo on every project page.

This recipe teaches you how to create a simple custom sheet layout. The only difference between the default layout and the one we will create is that the custom one contains a logo graphic. Once you have your custom sheet layout file, you can use it in all your schematics.

To create and edit sheet layouts, KiCad provides a helper application called "Drawing Sheet Editor." To open this editor, go back to the KiCad project manager and click on the Drawing Sheet Editor button at the bottom of the right pane (Figure 5.25.1).

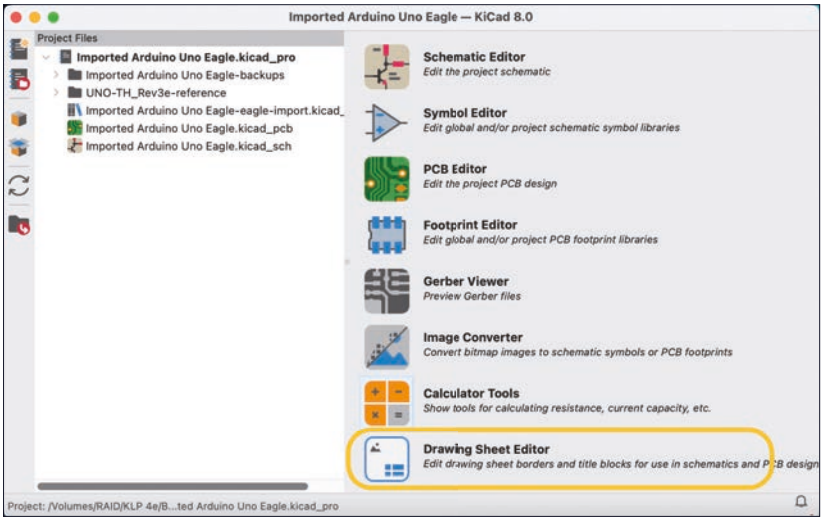


Figure 5.25.1: Starting the Drawing Sheet Editor.

The editor will start showing the default layout. You can work and edit the default layout, create a new one, or load and edit an existing one.

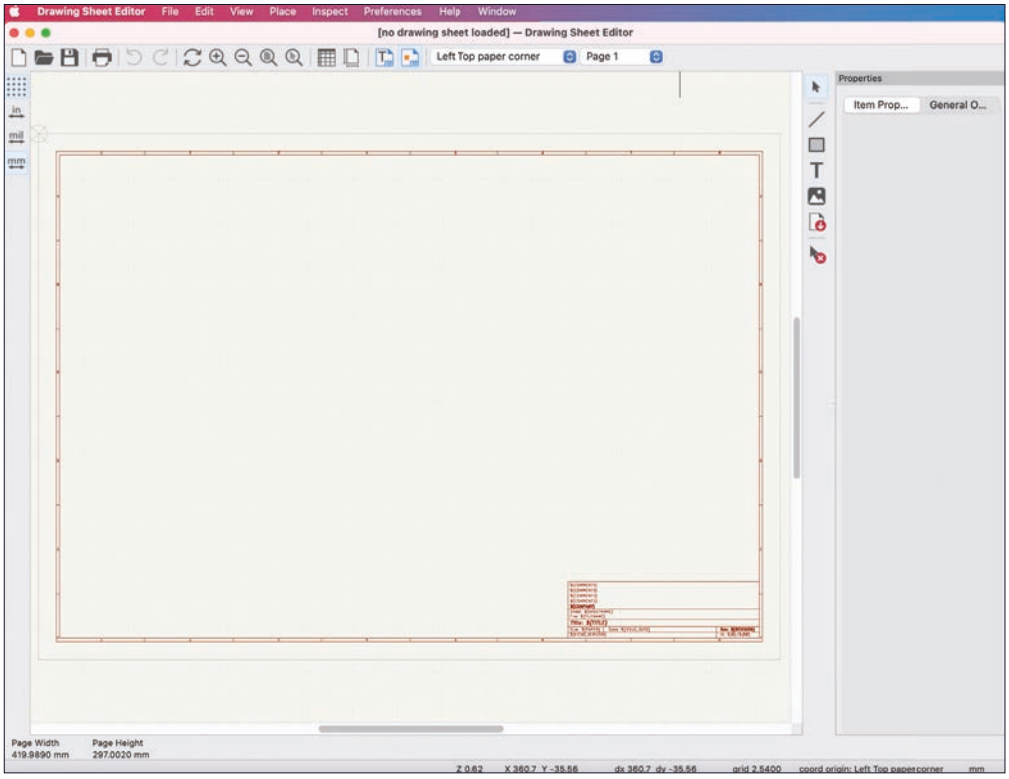


Figure 5.25.2: The Drawing Sheet Editor showing the default sheet layout.

A blank layout is empty of borderlines and text placeholders, giving you maximum freedom to design. You can create a blank layout by choosing "New" from the File menu or clicking the New Document button in the top toolbar. You can also open an existing layout to modify it to your requirements—KiCad ships with several sheets you can use as-is in your schematics or change them. To open a sheet layout, click the Open button from the top toolbar and browse for the sheet you want to work on.

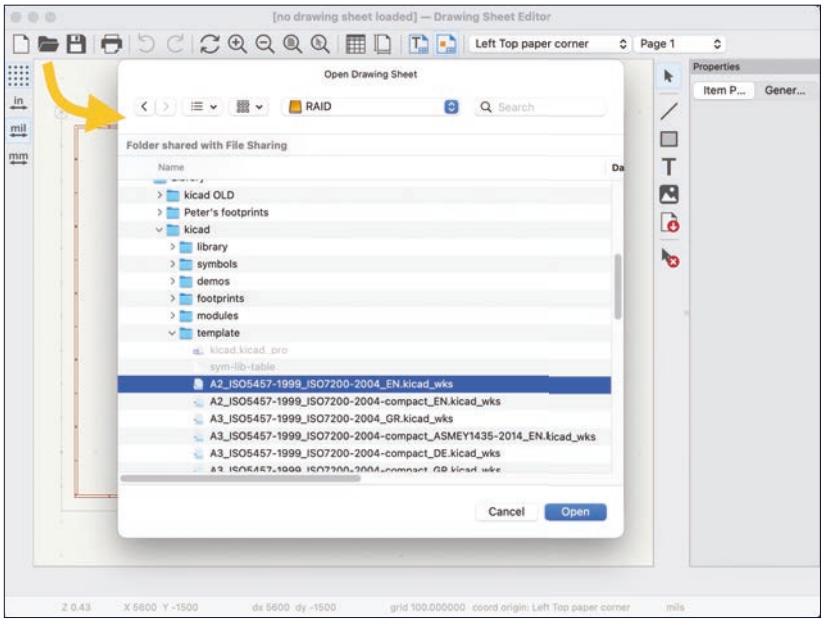


Figure 5.25.3: Open an existing sheet layout to edit.

If you don't remember where to find the schematic sheet layout files in your KiCad instance, go back to the KiCad project window and select "Configure Paths" from the Preferences menu. In the Environment Variables table, look for the path stored in the KICAD8_TEMPLATE_DIR variable.

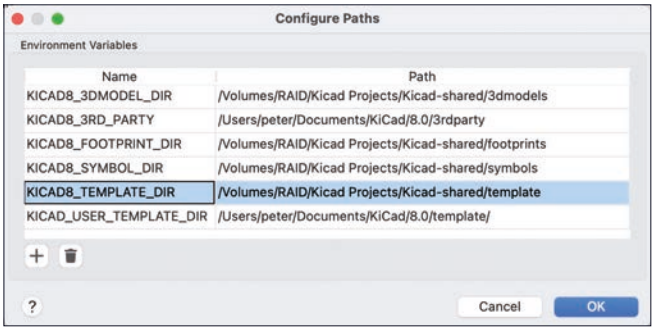


Figure 5.25.4: The path to my KiCad schematic editor sheet layout files.

I will use the sheet layout with the filename "A2_ISO5457-1999_ISO7200-2004_EN.kicad_wks" for this example. You can see the template below:

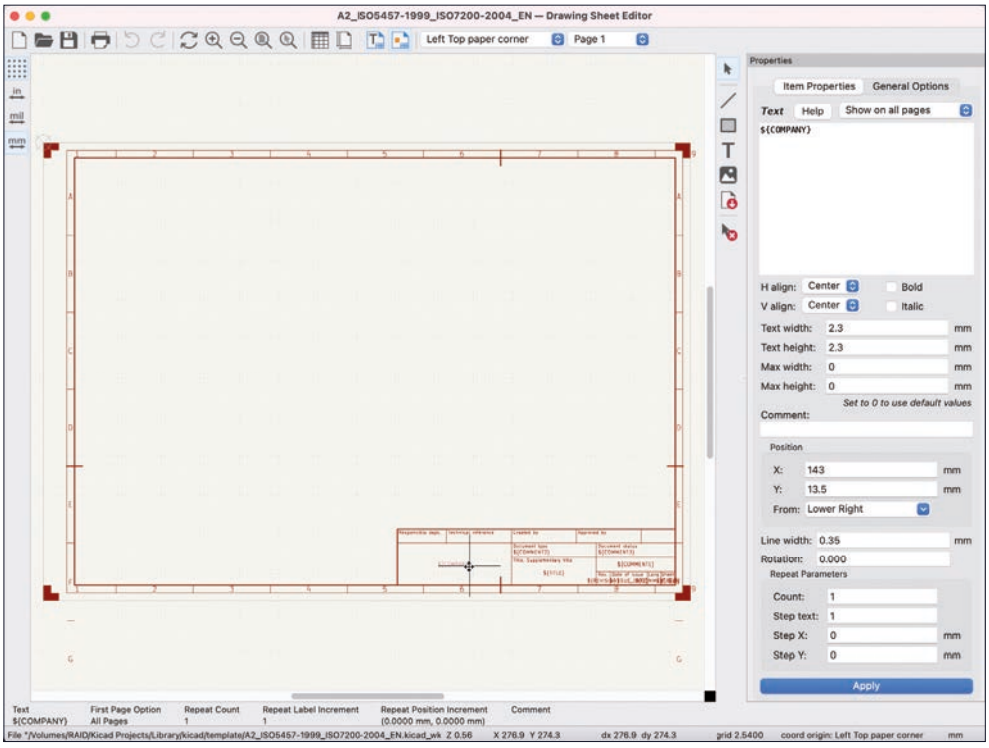


Figure 5.25.5: I will modify this sheet layout.

The editor provides drawing tools in the right toolbar and a properties pane on the right side. You can draw lines and boxes and add text. You can also insert images.

Each element has properties that you can edit in the properties pane. For example, below, I have used the line tool to draw a single line and then changed the line width to 5 mm in the properties pane:

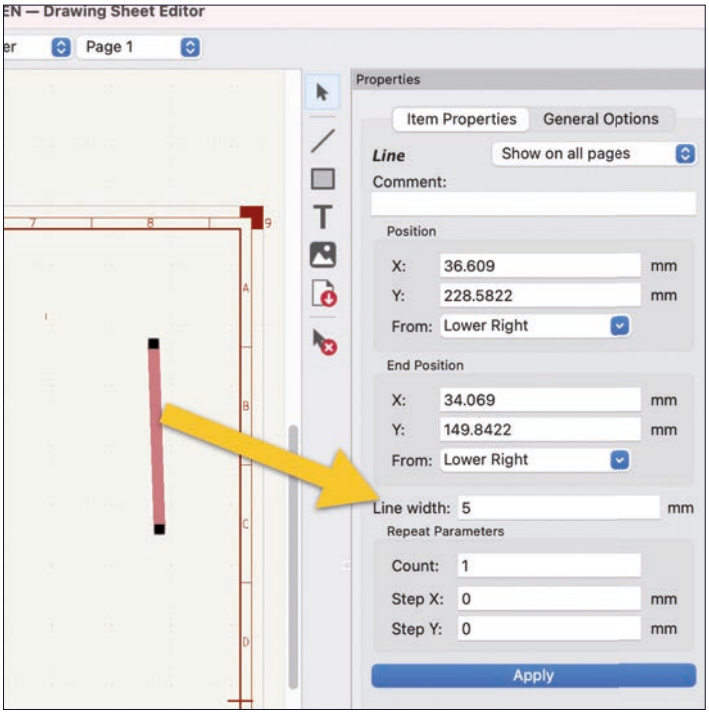


Figure 5.25.6: Every element has properties.

The most interesting and useful part of the sheet is the information box that contains the text placeholders. In the example below, I have selected the placeholder "${COMPANY}" to display its properties in the right pane (see below).

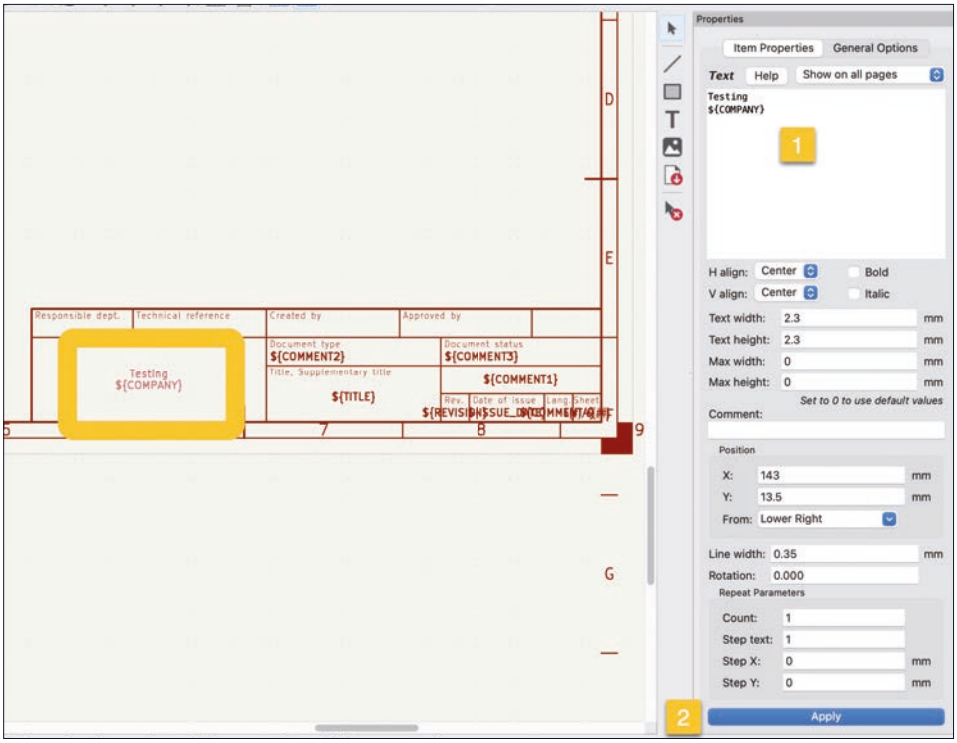


Figure 5.25.7: Editing the "COMPANY" placeholder.

In the Properties pane, I have added the word "Testing" in the text box, followed by the "${COMPANY}" variable in the following line ("1" in the figure above). Click "Apply" ("2") to finish editing this text placeholder, and then save the sheet editor.

Return to the schematic editor and open the Page Settings window (File —> Page Settings). Use the folder button to navigate and select the sheet layout you edited in the previous step in the File field. Notice that the "Company" placeholder has a field where you can type text. In other words, the placeholders you define in the sheet editor become text fields in the Page Settings window of the schematic editor. I have typed some text in the example below before clicking "OK."

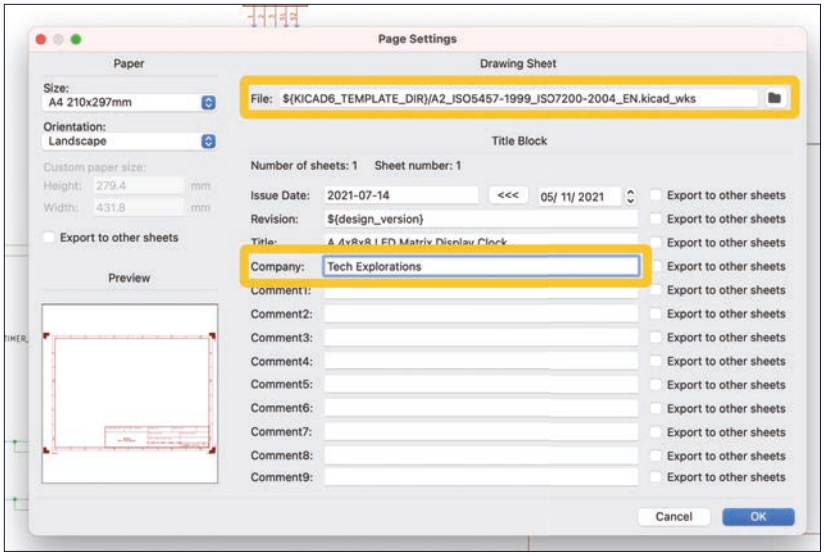


Figure 5.25.8: Setting a value for the Company text field.

In the schematic editor, the information box shows the text you provided in the Page Settings text fields using the edited sheet layout. You can see the result of this work below:

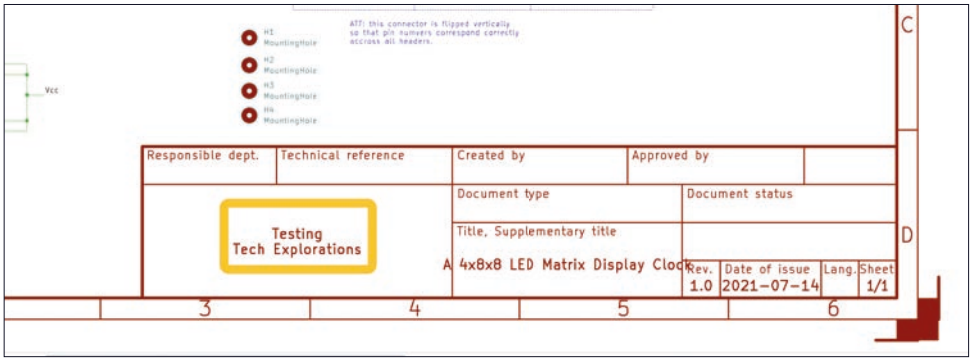


Figure 5.25.9: Using the customised sheet layout in this schematic.

You can also use formatting symbols to automatically display information such as the KiCad version used, current date, sheet number, etc. The table below lists the available format symbols.

Symbol	Description
%K	KiCad version
%Z	Paper format name
%Y	Company name
%D	Date

%R	Revision
%S	Sheet number
%N	Number of sheets
%Cx	Comment (x = 0 to 9 to identify the comment)
%F	Filename
%P	Sheet path
%T	Title

Table 5.25.1: A list of format symbols available in KiCad.

Another modification you can make is to add an image, such as a logo, to decorate the information box. Below, I have used the image tool from the right toolbar to add a PNG image in the Company text box:

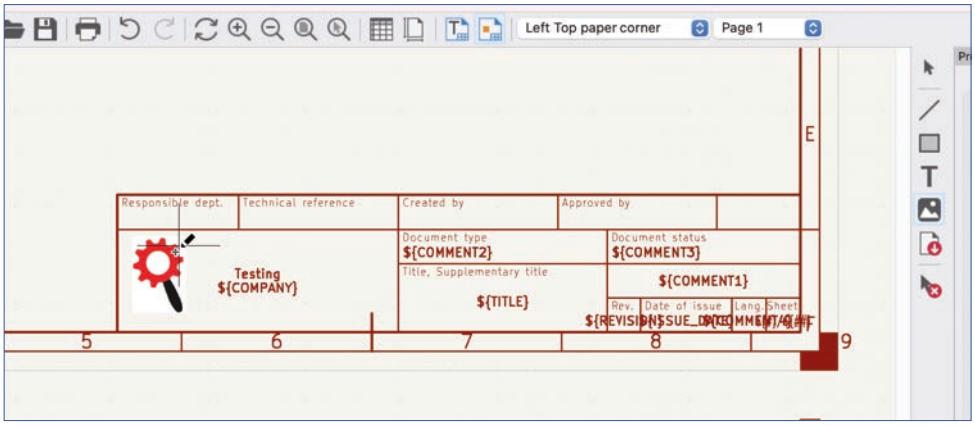


Figure 5.25.10: Added an image to the project information box.

Save the sheet layout and return to the schematic editor. The schematic information box will now display the logo:

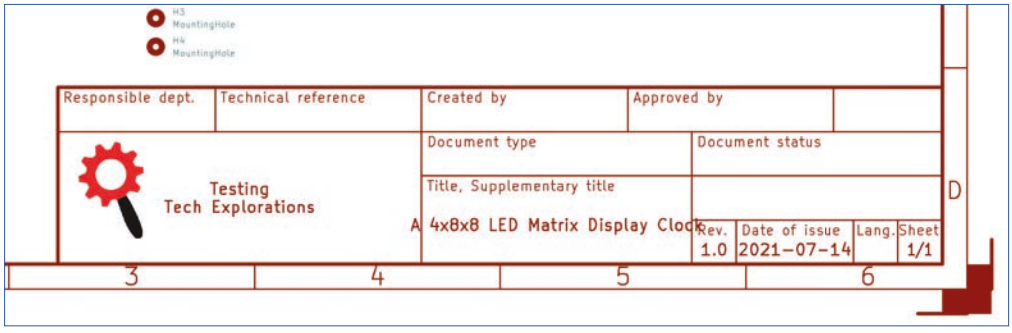


Figure 5.25.11: The image appears in the schematic information box.

You can use the same sheet layout in any of your projects by loading the sheet file to the project through the 'File' field in Page Settings.

5.26. Python scripting API

KiCad Python scripting extends the capabilities of the KiCad electronic design automation suite, enabling users to automate repetitive tasks, enhance functionality, and integrate with other tools through scripts written in Python. By leveraging Python, one of the most popular programming languages, users can programmatically manipulate various aspects of their PCB designs, from modifying design elements to generating custom reports. Python scripting in KiCad is particularly powerful for tasks such as:

- Automation: Streamline processes like board layout adjustments and batch modifications to design elements.
- Custom Tools: Create bespoke tools tailored to specific project requirements, such as custom design rule checks or automatic component placement.
- Data Extraction: Generate detailed reports like bills of materials, project statistics, or layer-specific information.
- Integration: Connect KiCad workflows with other software tools or databases to enhance the design of the manufacturing pipeline.

I did not cover this topic in previous editions of this book as it felt like an experimental feature used by power users with strong Python skills and the ability to understand documentation, more appropriate for programmers rather than PCB designers.

With KiCad 8, the situation has improved by an order of magnitude. KiCad Python Scripting has matured along with the rest of the platform, and it can be used to code a variety of operations on PCBs, from simple, such as changing track widths of a specific net in bulk, to more complicated, such as extracting a BOM with a single click. The documentation situation has also improved, with a lot of useful and straightforward information being available as part of the official KiCad documentation, in addition to the Doxygen auto-generated documentation, which generates the software reference documentation directly from the KiCad source code.

In this chapter, I will show you two simple examples of how to use KiCad Python Scripting. Along the way, you will learn how to use the KiCad Python shell to interact with your PCB in real time, how to create KiCad Plugins using the code that you tested in the shell, and how to navigate the PCBNew API to find the functions you need for your scripts.

Understanding the structure of KiCad's Python API is crucial as you begin. The API revolves primarily around modules such as "pcbnew", which manages PCB layout functionalities. By diving into this module, you can access classes and methods to manipulate virtually every aspect of your PCB design, from layers and tracks to footprints and vias. The KiCad documentation online is a great resource, offering detailed descriptions of these modules to help you understand how they interact with different components of your projects.

Starting with small scripting tasks can lead to a deeper understanding of KiCad's scripting environment. For instance, you could begin by writing scripts that automate the creation of backups for your PCB designs or generate a list of all the components on your board with their properties. These initial scripts are practical and teach you how to navigate the API and manipulate board data effectively.

Another key aspect is incorporating robust error handling. When scripting, unexpected behavior can occur, especially with complex designs or when updates to the API occur. Always ensure your scripts check for potential errors and handle them gracefully. For example, confirming the existence of a component before trying to modify it can prevent your script from crashing and aid in troubleshooting.

Don't overlook the power of Python's external libraries. While KiCad's API is powerful, sometimes using an external Python library for certain tasks like data manipulation or generating reports can be more efficient. The Python ecosystem includes libraries like "pandas" for data analysis or "matplotlib" for plotting, which can be seamlessly integrated with your KiCad scripts. You can combine powerful Python libraries with Kicad's to create powerful Python scripts for KiCad.

Finally, tapping into the KiCad community can be incredibly beneficial. Many users have likely faced and overcome similar challenges, sharing their scripts and solutions online. Engaging with platforms like the KiCad forums, GitHub repositories, or blogs dedicated to KiCad scripting can provide valuable insights, help solve specific problems, and inspire new ideas for what you can accomplish with scripting.

By following these guidelines, you'll enhance your efficiency in using KiCad and your overall approach to electronic design. Scripting allows you to streamline and extend your design workflows significantly, making them more powerful and less tedious.

5.26.1. A simple example

Let's begin with a simple example demonstrating using the KiCad Python console to list all footprints in an active PCB project. This example will teach you how to use the basic features of the KiCad Python scripting with the Python shell. You will also learn how to find the functions (and information about the functions) that you will want to include in your Python programs. When scripts run in the Python shell, they do so outside the KiCad environment. This is evident from the fact that any changes you make via the script cannot be undone in the PCB editor via the Undo button. For this reason, such scripts are often used to extract information from a PCB, and not for making changes.

This first example is very simple. It aims to print information about your PCB's footprints in the shell. In the second example of this chapter, you will write a script that changes the PCB, which is a typical use case of Python scripts in KiCad. Furthermore, once you have written and tested this Python script in the shell, you will learn how to convert it into an Action Plugin that you can invoke from the PCB editor Tools menu. Action Plugins extend the functionality of KiCad within the KiCad environment.

Let's begin with the first example. I suggest you open a copy of one of your existing KiCad projects, which you will change in the next example. Once the project is open, start the PCB editor and follow these steps.

1: Open the Python Console
In KiCad, navigate to Tools > Scripting Console. This opens an interactive Python shell linked directly to your active KiCad session.

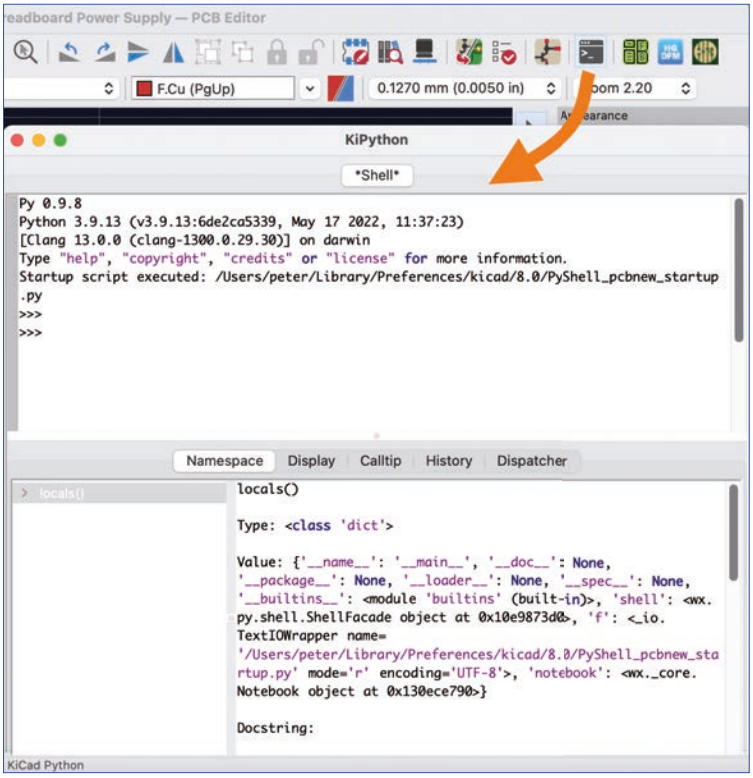


Figure 5.26.1: Started a new KiPython (KiCad Python) shell.

2: Write and Run a Script
You now have a Python shell connected to the active PCB in the PCB editor. The shell window is the large pane at the top of the KiPython window. We'll work in the shell in this chapter. Below the shell are several tabs: Namespace, Display, Calltip, History, and Dispatcher. As you progress through this example, you will learn about some of those tabs.

Let's start with the very first line of code. In the Python console, copy the following code into the shell after the ">>>" prompt and hit the Enter key:

```
import pcbnew
```

The command "import pcbnew" is used to access the "pcbnew" module, part of KiCad's Python API. This module allows you to interact programmatically with the PCB design features in KiCad. When you import "pcbnew", you gain access to a wide range of classes and functions that can be used to manipulate PCB layouts directly from Python scripts. By importing the "pcbnew" class, you can use functions such as:

1. Access and modify PCB designs: You can create, open, and save PCB files, manipulate layers, and modify properties of various elements such as tracks, pads, vias, and footprints.
2. Query PCB data: The module allows you to retrieve information about the current PCB, such as the list of all components, their positions, and other attributes.
3. Control graphical representation: You can manage graphical elements like drawings and texts on the PCB.
4. Automate tasks: Common tasks such as updating footprints, rerouting connections, or adjusting design rules can be automated using scripts that call functions from the pcbnew module.

You can explore the functions available in the "pcbnew" class in the Namespace tab. See the screenshot below:

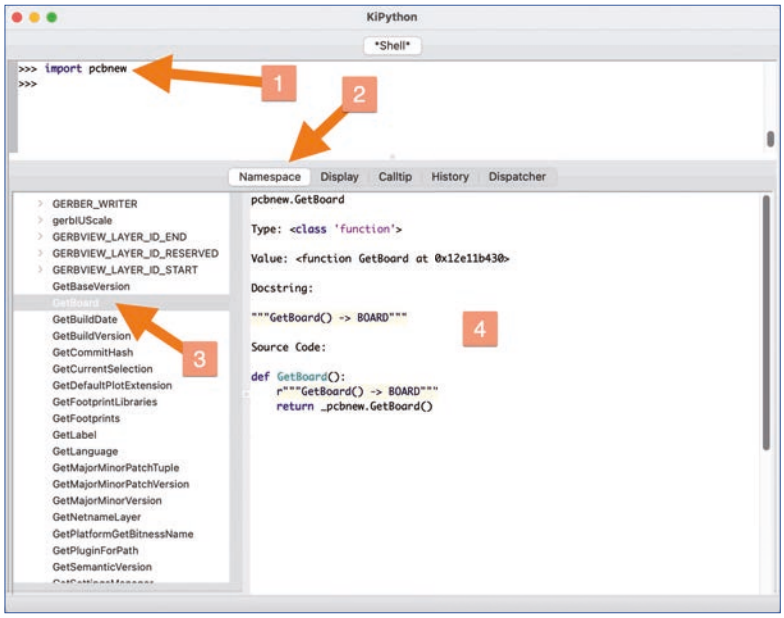


Figure 5.26.2: You can explore the KiCad Python API namespace in the Namespace tab.

After entering the import command (1), select the Namespace tab (2). You can scroll up and down to view the available functions and classes in the left pane. We'll be using the "GetBoard()" function next, so find it and click (3) on it to reveal documentation information in the right pane (4). The information is rather cryptic, but you learn that this function

returns an object of type "Board." You can use the Namespace tab to find information about the modules loaded in the shell quickly. I recommend you do this regularly as you progress with this example to become accustomed to it.

Let's continue. Back in the shell, copy this line of code:

```
board = pcbnew.GetBoard()
```

This code retrieves the currently loaded PCB board from the KiCad application. "GetBoard()" is a function within the "pcbnew" module. This function is responsible for fetching the active board currently open in KiCad. The result of pcbnew.GetBoard() is assigned to the variable board. This variable now holds a reference to the PCB board object. This object represents the entire PCB design and allows the script to interact with and manipulate various aspects of the PCB, like its layers, tracks, footprints, and other components. Click the History tab to see the commands you have issued in the session thus far:

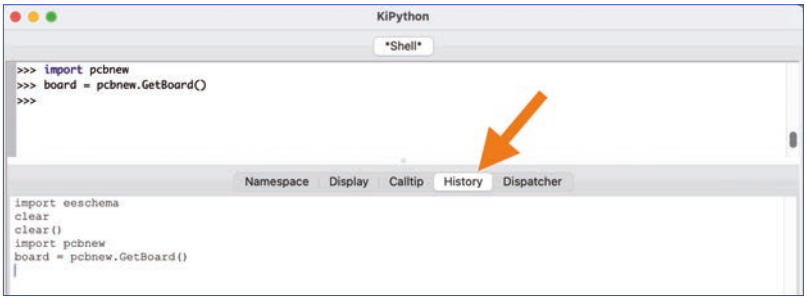


Figure 5.26.3: The History tab contains the command issues in this session.

Let's finish the script with the following two lines. Copy these lines one at a time and observe Python's indentation requirements for code inside a loop. Notice that after you copy the first line, the prompt will change to "...". Copy the second line after that prompt:

```
for footprint in board.GetFootprints():
    print(f"{footprint.GetReference()}: {footprint.GetFPID().
GetUniStringLibItemName()}")
```

After the second line, hit Enter again to commit the code and run it by Python. The output of the program will appear in the shell. It looks like this:

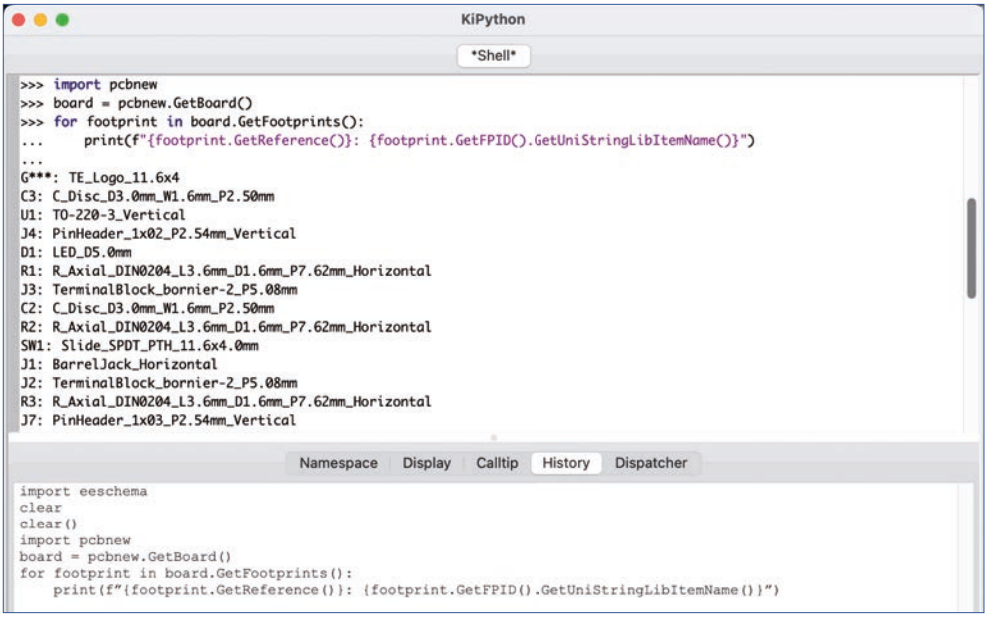


Figure 5.26.4: Completion of the program, and its output.

Let's look closely at the last two lines. The first line, "`for footprint in board.GetFootprints():`" uses KiCad's "pcbnew" module to create a loop that iterates over all the footprints present on the currently loaded PCB board. Remember the "`board`" object previously obtained using "`board = pcbnew.GetBoard()`"? This object represents the active PCB design in KiCad.

In the same line, the method "`GetFootprints()`" is invoked on the board object. This method retrieves a list (or similar iterable collection) of all the footprints currently part of the PCB design. A footprint, in this context, refers to the physical representation of a component on the PCB, which includes pads, silkscreen, and other associated graphics and attributes. The "`for`" loop is used to iterate through each element in the iterable returned by "`GetFootprints()`." In each iteration of the loop, the variable "`footprint`" is assigned to one of the footprints from the list. You can then use this "`footprint`" variable within the loop to access or modify properties specific to each footprint.

Inside the loop, you can execute various operations on each footprint. For example, read or modify the footprint's position, orientation, reference designator, or any assigned custom attributes. You could also execute checks or make changes, such as repositioning footprints, updating their metadata, or checking for compliance with specific design rules.

In the second line, the code "`print(f"{footprint.GetReference()}: {footprint.GetFPID().GetUniStringLibItemName()}")`" uses Python's f-string formatting to print details about each footprint in a KiCad PCB project. This is typically used within a loop that iterates over all footprints on a board. The method call "`footprint.GetReference()`" retrieves the reference designator of the footprint. The reference designator is a label used

to identify components on a printed circuit board (PCB), such as "R1" for resistor 1 or "C2" for capacitor 2. Then, "`GetFPID()`" is a method called on the footprint object that fetches the footprint ID, an instance of a class encapsulating the unique identifiers for the footprint. This footprint ID object contains several pieces of information related to identifying the footprint within the KiCad system. Next, The "`GetUniStringLibItemName()`" method is called on the footprint ID object returned by "`GetFPID()`". It retrieves the name of the library item as a Unicode string. This name is typically the name given to the footprint in the library, such as "SOT-23" for a specific type of transistor package. This lets users know exactly what kind of footprint (in terms of physical layout and pin configuration) is referenced.

Finally, the output of this script is produced by the "`print()`" function with f-string formatting. Python's f-string formatting creates a string that combines the reference designator and the library item name. The curly braces {} in the f-string include expressions evaluated at runtime, and their results are inserted into the string. In this case, the results of "`GetReference()`" and "`GetUniStringLibItemName()`" are inserted into the string. The formatted string is constructed with each footprint's reference designator followed by a colon and the name of the library item, providing a clear and readable output of the form "R1: SOT-23".

This concludes the first example I wanted to show you. Before continuing with the second example, I'd like to examine the KiCad Python shell more closely, particularly its interactivity.

Interactive shell and code completion

The interactive shell in KiCad, often called the "Python Console" or "Scripting Console," is a built-in tool that provides a direct access point for executing Python code. It behaves similarly to other Python interactive interpreters. Still, it is integrated within KiCad, providing the unique advantage of direct interaction with the KiCad projects and its components through the KiCad Python API.

As you have already seen, when you open the Python Console in KiCad, you are presented with a prompt where you can type Python commands. The commands you enter are executed immediately, and the results or output are displayed in the console. This immediate feedback makes it an excellent tool for learning and experimenting with the KiCad Python API and debugging and developing scripts.

Because of the relatively "unfriendly" documentation of the KiCad Python API, I find that the code completion feature of the interactive shell is a reliable way to find the functions I need for my scripts quickly. The code completion feature in the KiCad Python shell enhances the scripting experience by helping you write code more efficiently and accurately. This feature assists in quickly finding available methods, variables, and properties, reducing errors and learning time. In the screenshot below, you can see code completion in action.

Figure 5.26.5: Code-completion in action.

In the example above, I started with "`pcbnew.`" After the full stop, the code-completion bubble appeared and offered available options. The options narrowed down as I typed additional characters. You can use your mouse scroller to scroll through the list.

Generally, to use code completion in the KiCad Python shell, as in other text editors, you begin typing a command or object name. When you reach a point where you need suggestions, you can press the Tab key. The shell will then display a list of completions matching the partially typed input. If the input is ambiguous or incomplete, multiple suggestions might be shown. Select the desired completion from the list and continue with your coding.

5.26.2. Make a change to a PCB

In this example, you will learn how to write a simple Python script that makes bulk modifications to the active PCB. First, you will write and test the script in the KiCad Python shell. After that, you will convert the script into a KiCad Action Plugin to invoke it from the PCB editor's Tool menu without opening the Python shell. I am running this example using a clean Python shell by restarting KiCad.

The process starts with the Python shell, where you will write and test the program. Start KiCad and open your test projects using the preview example. Open the PCB editor, and then click on the Python Shell button. The script in this example is based on the original presented by Maciej 'Orson' Suminski at KiCon 2019. I have added comments, variables, and some auxiliary code to make reading and learning easier. The objective of the script is to change the width of all tracks that belong to the "12V" net. In the shell, copy and paste the following program (best to do this line-by-line):

```
from pcbnew import *

# Kicad track widths are in nm.
# This variable will hold the value in mm
# and we'll use a function to convert it to nm.
desired_track_width = 0.25

# Load the current board
board = GetBoard()

# Check if the board is loaded correctly
```

```
if board is None:
    print("No active board loaded. Please open a PCB file and try again.")
else:
    # Iterate through all tracks on the board and set their width
    for track in board.GetTracks():
        if track.GetNetname() == "/12V":
            track.SetWidth(FromMM(desired_track_width))

# Refresh to update the board display and save the changes
Refresh()

print("All track widths have been updated.")
```

Here's my shell window:

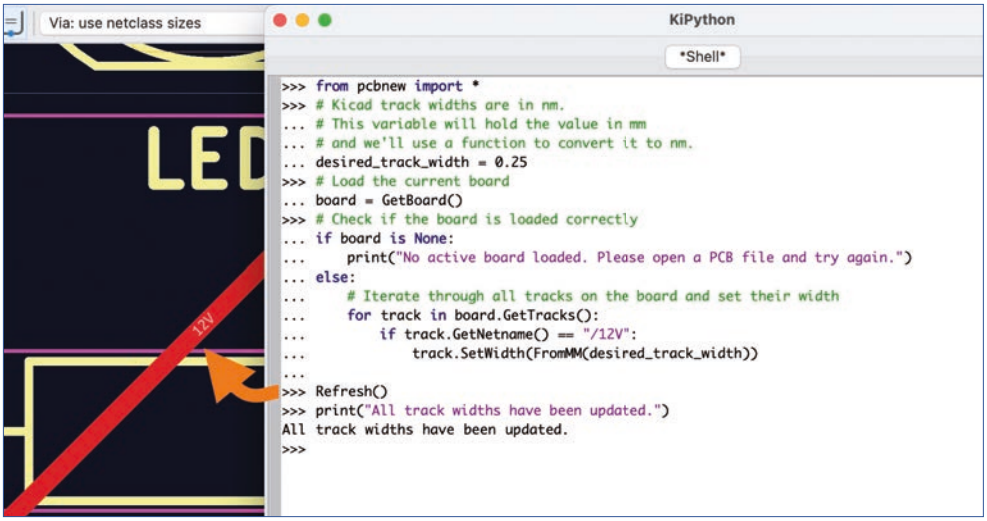


Figure 5.26.6: This script has modified the width of the routes that belong to the "12V" net.

After calling the "Refresh()" function, the PCB is updated so that all tracks that belong to the "12V" net have a width of 0.25 mm. Let's focus on this script and learn how it works.

The statement "`from pcbnew import *`" imports all public objects from the pcbnew module directly into the namespace of the script or module where it is called. As you know, this module, pcbnew, is part of KiCad's Python API and provides access to numerous functions, classes, and constants that allow you to manipulate and interact with the PCB layout elements in KiCad. By importing everything from the module into the local namespace, you can directly use the functions and classes without the need to prefix them with "`pcbnew.`". For example, after such an import, you can directly use "`GetBoard()`" instead of "`pcbnew.GetBoard()`". The syntax from pcbnew import * is known as a wildcard import. This import style adds all the public names from pcbnew to the local namespace, which

can lead to conflicts if other parts of the program or other imported modules use the same names, so you should be careful and consider using it in small scripts or script that you understand well.

The line "`desired_track_width = 0.25`" declares and initializes a variable named `desired_track_width` with a value of `0.25`, representing the width in millimeters (mm). Given that KiCad uses nanometers as the unit of measurement for track widths and other dimensions, the value stored in `desired_track_width` must be converted from millimeters to nanometers before it can be applied to the track properties in KiCad. Luckily, there's an easy way to do this conversion. In KiCad's Python API, the function `FromMM()` is used to convert measurements from millimeters (mm) to the internal unit system used by KiCad, which is typically nanometers (nm). This function is particularly useful because it provides a straightforward way to handle unit conversions, allowing PCB designers to work in millimeters—more familiar and often more meaningful in practical terms—while ensuring that the values are correctly converted to the scale that KiCad uses internally for precision and operations.

The code line "`board = GetBoard()`" retrieves the currently active PCB from KiCad and assigns it to the variable board. You saw this code in the previous example, but in the form "`board = pcbnew.GetBoard()`" as we only imported the pcbnew module (not *).

Next, the line "`if board is None:`" checks whether the variable board is None. This common pattern is used to verify that an object or variable has been successfully initialized or assigned a meaningful value before proceeding with operations that depend on that object. If the condition is true, a board has not been loaded; the program will print a message to the shell and exit.

The line of code "`for track in board.GetTracks():`" iterates over all the tracks on the currently loaded PCB board. This is a common pattern for accessing and potentially modifying each track in a board design. The method call to "`board.GetTracks()`" retrieves a list of all the tracks in the PCB layout. And the code "`for track in`" is a Python for-loop, which iterates over each item in the iterable returned by "`GetTracks()`". For each iteration of the loop, the variable track is assigned to one of the tracks from the list. You can then use this track variable within the loop to access or modify properties specific to each track.

Next, the code "`if track.GetNetname() == "/12V":`" functions as a conditional check within the broader context of operations on tracks in a PCB design. The method "`track.GetNetname()`" is called on a track object, which represents an individual conductive path (or track) on the PCB. The `GetNetname()` function retrieves the name of the electrical net of which the track is part. Then, we use "`== "/12V`" to check whether the name of the net returned by `GetNetname()` is equal to "`/12V`".

If it is equal, then the last line of code in the block runs. The line of code "`track.SetWidth(FromMM(desired_track_width))`" performs two main operations to set the width of a PCB track. The function "`FromMM(desired_track_width)`" is part of KiCad's Python API and is used to convert a measurement from millimeters (mm) to KiCad's

internal unit system, which is in nanometers (nm). The argument desired_track_width is expected to be a numerical value representing the width of the track in millimeters. Then, the method "`track.SetWidth(...)`" is called on a track object. The argument passed to `SetWidth` must be in the internal unit system of KiCad (nanometers), which is why the `FromMM` function is used first.

Finally, we use the "`Refresh()`" to refresh or update the graphical representation of the PCB layout in KiCad's user interface. This function is particularly important after making programmatic changes to the PCB via scripts, as it forces the PCB editor to redraw the board, reflecting any modifications made to the layout. If you don't use "Refresh()" at the end of the script, any changes will not be rendered in the PCB editor until you force a refresh through the Refresh option in the View menu.

Suppose you changed your mind and want to restore the changes tracks back to the original. If you try to use the Undo function, you will notice that it is unavailable for changes introduced via the Python shell. This is an important limitation of the shell and a reason why it is normally used for testing and development and not in production settings. To run Python scripts fully integrated with KiCad, you will want to develop a KiCad Action Plugin.

5.26.3. Create a KiCad Action Plugin

Suppose you could run the script successfully and verify that the PCB has been updated to show the new width for the "12V" net. In that case, you might wonder if this was too much work for something simple that can be achieved by existing functionality in KiCad. I agree that this example is trivial. However, this is only the start. The real value of KiCad Python Scripting is in the ability to create Plugins that can add custom and advanced functionality to KiCad. For example, you can create plugins that automatically optimize PCB layouts for signal integrity, thermal performance, and electromagnetic compatibility, which can be time-consuming and error-prone when done manually. Another example of an advanced action plugin and application is for customized design rules check and reporting, such as the one from NextPCB I discuss in My book Fundamentals and Projects.

Let's examine a simple action plugin by building on the work from the previous segment. Regardless of the plugin's complexity, the components are the same. What you will learn in this segment can be applied to all action plugins. To develop a KiCad action plugin, you will need a tested and working script and a text editor. I will use the script from the previous segment and the Atom text editor. You will not need the KiCad Python shell, so feel free to close it.

An Action Plugin Skeleton

Creating a KiCad action plugin involves structuring a Python class that extends KiCad's ActionPlugin class. Below is a skeleton or template for a KiCad action plugin. This structure includes essential components such as defining the plugin's metadata and the main functionality that will be executed when the plugin is activated.

```
import pcbnew
from pcbnew import ActionPlugin

class MyCustomPlugin(ActionPlugin):
    def defaults(self):
        """Setup the basic properties of the plugin."""
        self.name = "My Custom Plugin"
        self.category = "A Category"
        self.description = "A detailed description of what my plugin does."
        self.icon_file_name = ""  # Optionally add a path to an icon file here
        self.show_toolbar_button = True  # Optionally show a button on the
toolbar

    def Run(self):
        """Execute the functionality of the plugin."""
        # Your code here
        pcbnew.wxLogMessage("Hello from My Custom Plugin!")

# Register the plugin into KiCad
MyCustomPlugin().register()
```

Let's drill into the components of an action plugin script:

1. Import Necessary Modules: Import the `pcbnew` and, specifically, `ActionPlugin` from pcbnew, which are necessary for creating the plugin.
2. Plugin Class Definition: Define a class that extends `ActionPlugin`. This class will encapsulate all the functionality of your plugin.
3. defaults Method: This method is used to set up the basic properties of the plugin:
    1. self.name: The name of the plugin as it will appear in KiCad's menu.
    2. self.category: A category under which the plugin will be grouped in KiCad's menu.
    3. self.description: A short description of what the plugin does.
    4. self.icon_file_name: Path to an icon that will be displayed alongside the plugin in the menu (optional).
    5. self.show_toolbar_button: Whether to show a button on the toolbar for this plugin (optional).
4. Run Method: This is where you define the plugin's actual functionality. Whatever you code here will be executed when the user selects your plugin from the menu.
5. Plugin Registration: At the bottom of the script, an instance of your plugin class is created and registered with KiCad. This registration makes the plugin available within the KiCad environment.

For a plugin to be able to be registered, it must be saved in the "scripting/plugins" directory of your KiCad data folder with the ".py" extension. The easiest way to find the correct plugins directory is to click "Revel Plugin Folder in Finder" under Tools —> External Plugins.

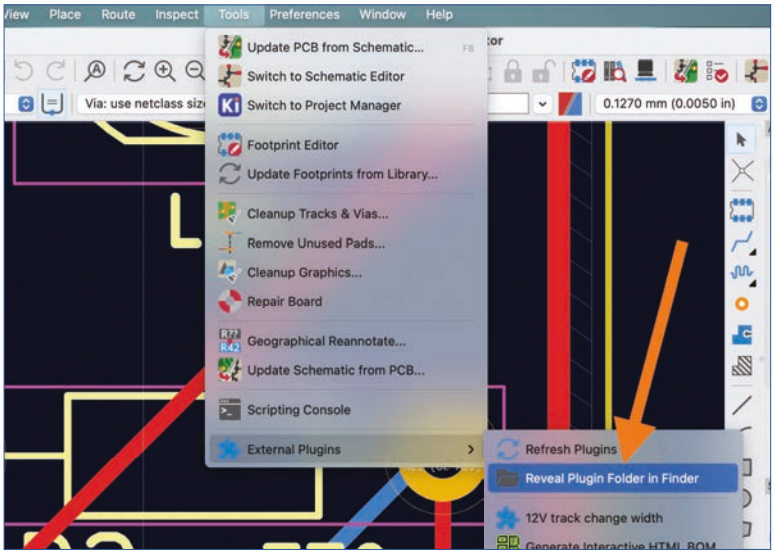


Figure 5.26.3.1: How to find the plugins directory.

After saving the plugin file in the plugins folder, you must Refresh the registry. You can do this by restarting KiCad or, better, by clicking Refresh Plugins under Tools —> External plugins in the PCB editor.

Convert a KiCad Python Script into an Action Plugin

Now that you know the Action Plugin basics, let's convert the script from the previous segment into a plugin. Because this script will work as a plugin integrated into Kicad, we will adapt the basic code into the Action Plugin skeleton you learned about in the previous segment. We will use a simple dialog box to interact with the user. Below is the new script with all the changes implemented. I have highlighted the original code with bold text.

```
import pcbnew
from pcbnew import ActionPlugin, GetBoard, FromMM, Refresh
import wx

class TrackWidthAdjuster(ActionPlugin):
    def defaults(self):
        """Set up the basic properties of the plugin."""
        self.name = "Adjust 12V Track Widths"
        self.category = "Modification Tools"
        self.description = "Sets all 12V track widths to a specified width in mm"
        self.icon_file_name = ""  # Optional: Add the path to an icon file here
```

```
    def Run(self):
        """Execute the adjustment of the track widths for 12V nets."""
        # Desired track width in mm, converted to KiCad's internal units (nm)
        desired_track_width = 0.25
        track_width_nm = FromMM(desired_track_width)

        # Load the current board
        board = GetBoard()

        # Check if the board is loaded correctly
        if board is None:
            wx.LogMessage("No active board loaded. Please open a PCB file and try
again.")
        else:
            # Iterate through all tracks on the board and set their width if they
belong to the "12V" net
            for track in board.GetTracks():
                if track.GetNetname() == "/12V":
                    track.SetWidth(track_width_nm)

            # Refresh to update the board display and save the changes
            Refresh()

            wx.LogMessage("All 12V track widths have been updated.")

# Register the plugin into KiCad
TrackWidthAdjuster().register()
```

Copy this code to your text editor and the KiCad plugin directory. Refresh the plugins (Tools —> External Plugins —> Refresh Plugins), and then run the plugin (Tools —> External Plugins —> Adjust 12V Track Widths). The plugin will change the width of the traces and show a confirmation message, as in the example below:

Figure 5.26.3.2: The plugin completed succesfully.

Click OK to exit. Remember that the plugin runs fully integrated in the KiCad environment. This means that functions such as undo and redo work. If you prefer the original track widths, you can use the undo function to revert the changes made by the plugin.

Code Walkthrough

Let's zoom into the code to understand how it works. The import statements are at the script's header. These import key modules and functions to develop an Action Plugin for KiCad using Python. Aside from the pcbnew module, we need several more in this script. The `pcbnew` module provides the "`ActionPlugin`" class. It is used to create new plugins by extending this class. Action plugins can be executed from the KiCad tools menu, allowing users to extend KiCad's functionality with custom actions. You have used the "`GetBoard`" function previously. This function retrieves the current board being edited in KiCad and is essential for scripts that need to interact directly with the board, as it provides the script with a handle for the board object. You have also seen the "`FromMM`' and "Refresh" functions before, so I will not repeat them here. The "import wx" function is new. "`Wx`" refers to wxPython, a cross-platform GUI toolkit for Python, used extensively in KiCad for its user interface. Importing `wx` is essential for creating custom dialogs, frames, or other GUI components within the plugin or interacting with KiCad's GUI elements. In this script, we use wx to create the notification window for user notifications.

The code class "`TrackWidthAdjuster(ActionPlugin):`" defines a new class named `TrackWidthAdjuster` that extends ActionPlugin from KiCad's Python API. This line initiates the creation of a custom plugin for KiCad using Python. In that class, you will find "`default`" and "`run`" functions.

In the "`default`" function, we set up the plugin's basic properties, which determine how it will be presented and categorized within the KiCad user interface. The names used for the various properties are self-explanatory. The name is the most important because it appears in the Plugins menu. You can also specify the path to an icon to represent your plugin. I have left that path blank. The icon can be rectangular in PNG format with around 16x16, 24x24, or 32x32 pixels.

In the "`run`" function, I have copied the original code with a few simple changes. The most important change is the replacement of the regular Python "`print()`" functions with calls to the "`wx.LogMessage()`" function. This function utilizes the wxWidgets library for its graphical user interface, which displays messages to the user through KiCad's message panel. When used in the plugin, it creates a small window with a custom message for the user to see and acknowledge with an "OK" button.

Finally, the code "`TrackWidthAdjuster().register()`" registers an Action Plugin with KiCad. "`TrackWidthAdjuster():`" creates an instance of the TrackWidthAdjuster class, and "`.register()`" registers the plugin instance with KiCad, making it available to be activated through the KiCad UI. Registration typically involves adding the plugin to the available tools or actions a user can execute directly from the KiCad menus. As you have probably guessed, this line aims to integrate the `TrackWidthAdjuster` plugin into KiCad so that it becomes part of KiCad's operational framework. Registering the plugin becomes accessible

to users under the menus designated for plugins, often under Tools > External Plugins in KiCad.

5.26.4. Learning More About KiCad Python Scripting API

This was an introduction to KiCad Python Scripting, the Python Shell, and KiCad Action Plugins. To deepen your knowledge of KiCad Python scripting, consider the following resources:

1. Official Documentation: Visit the KiCad Documentation website, which offers comprehensive guides and API references. You will find the documentation for KiCad 8 at https://txplo.re/kicad_8_scripting
2. Community Forums: Engage with other KiCad users and developers in forums such as the KiCad.info forums or the subreddit r/KiCad. These communities can provide support, share scripts, and offer practical advice. The main KiCad forum is https://forum.kicad.info/
3. Pcbnew Namespace Reference: This reference is extracted from the source code. It is useful if you can find your way around the Python source code. You can find it here: https://docs.kicad.org/doxygen-python-6.0/namespacepcbnew.html

Index

2D Barcode 395
3D model 65
3D models 63, 148, 222
3D shapes 144, 151
3D STEP files 63
3D viewer 116, 150, 217, 222
4x8x8 LED Matrix Clock 78
8x8 LED display 84
8x8 LED matrix display 73, 74
0402 257, 285
".gitignore" file 166
.STEP 148
".step" file 361

A

AC analysis 448
Action Plugin 482, 484
ActionPlugin 482, 486
Align/Distribute 106
Align items to Grid 36
Altium 432
amplitude 388
analysis report 220
annotation 34, 39
annotator 173
antenna 253, 308
AP3015 24
API 473
Appearance 126, 215, 246
archive 458
Arduino 75, 77, 83, 84
Arduino Pro Mini 773, 5, 77, 82, 114, 117, 139
Arduino Pro Mini footprint 120
Associations 87
Atmega328P 73, 240, 352, 354
Atmega328P-AU 158, 185
autorouter 362
auto-routing algorithm 369

B

B3S-1000P 259
backup 165
Barcode 395
barrel jack 114
Basic Git commands 410
Berkley University 436
BGA 392
Bill of Materials 76, 161
Board has malformed outline 139
Board Setup 101
BoM 167
BOM table 173
boost converter 24
branch 412
breadboard power supply PCB 100
bug 151, 237
bulk-editing tools 330
buttons 114

C

cache 165
Cadstar 432
calculator 463
caliper 287
capacitors 24
CIDER 436
circuit simulator 435
classifier 341
Clearance Report 380
clearance resolution 378
Clemens 34
Clemens Valens 23
code completion 479
color scheme 428
commit 221, 411
configuration string 444
Configure the simulator 444
Constraints 43
Constraints Resolution 380
copper fills 51, 201, 307, 308, 310
copper fill zones 126
copper traces 248
copper zone 53
courtyard 294, 301
courtyards 342
CP2102N 259, 322
custom and advanced functionality 482

Custom Rule 348
Custom Rules 344, 345, 347
custom schematic sheet 464
Customs Rules 345
custom user templates 456

D

data logger 158
DC/DC converter 24
defects 341
Design for Manufacturing 70
Design Rules 102
Design Rules and net classes 337
Design Rules Check 61, 136, 216, 315, 348
Design Rules Checker 341, 342, 377
design rule violations 344
DFM 70
DFM checklist 70
DFM report 321
diff 414
differential pair 305, 381, 388
differential pair routes 305
differential pair tracks 384
Differential skew 389
Digikey 81
DIP 392
DRC 54, 75, 123, 128, 136, 142, 144, 200, 215, 216, 228, 230, 248, 298, 301, 310, 327, 342, 347, 348
drill files 317
Drill Holes 375

E

Eagle 364, 432
EasyEDA 432
Edge.Cuts 104, 106, 115, 116, 117, 286
Edge.Cuts layer 116
Edit Text and Graphic Properties 301
EEPROM 193, 224
EEPROMs 158
Electrical Rules Check 38, 94, 183, 275
Electrical Rules Checker 348
Elektor 23, 34
Elektor Labs 23
Elektor Magazine 23
ERC 91, 94, 96, 184, 186, 275, 348, 350
ERC errors 207
Error 350
ESP32 252, 272, 281, 285, 290, 298, 304, 322
ESP32 development kit 253, 256
ESP32 dev kit 252
ESP32-DevKitC-v4 256
"ESP32-DevKitC-V4 269
ESP32-SOLO-1 269
ESP32WROOM-32 260
Espressif 252, 270

F

"Fill All Zones" 204
Find and Replace 330, 331
Footprint Editor 391, 392
footprint generator 393
footprint placement 107
footprints 342
Footprint Wizard 391
Fortran 436
four-layer 227
Freerouting 362, 363, 367, 369
f-string formatting 478

G

Gerber 63, 67, 104, 142, 217, 222, 316, 327
Gerbers 165, 225
Gerber Viewer 67, 142, 143, 218, 231, 319
Gerber ZIP 219
GetBoard 476, 477, 480
GetFootprints 477
git 422
Git 158, 162, 167, 188, 189, 222, 225, 230, 398, 403, 404, 413
gitattributes 234, 235
git branch 233, 414, 419, 422
git checkout 416
Git commit 234
git diff 413
GitHub 403, 419
gitignore 408, 409
git log 416

git push 424
Git repository 164, 171, 173, 175, 179, 189, 201, 206, 227
git status 412, 415
git user settings 404
Global Deletions 228
GND 37
GNDA 37
GNDD 37, 54
GPIO 285
graphics 129
grid size 116
GUI 486
guidelines 36

H

HEAD 416
hierarchical labels 181, 183
hierarchical sheet 170
Highlight wires 396
HSPICE 436

I

import 432
inductor 24, 29
Inspect 372
Interactive Delete 329, 330
Interactive Router 197
interactive shell 478
ISPICE 436

J

Java JRE 365
JSON 461

K

keep-out area 308
KiCad 364
KiCad Library for Arduino Modules 75

L

layout design workflow 41
LayoutEditor 364
LED matrix display 106, 108, 193
LED matrix displays 103
length tuning 388
Lisp 344
lock 124
Locked tracks 124
log 412
logo 62
LTSpice 436

M

Manufacture 142, 217
manufacturer 142
manufacturing analysis 220
MAX7819 74, 154
MCU 193
MCU Datalogger 162
meander 388
Measure Tool 376, 377
merge 225, 234, 400, 417
merge changes 234, 242
Micro Power Step-Up DC-DC Converter 29
Micro USB 272
MMSS8050-H-TP 260
mounting holes 63, 114
Move 123
MSC1212Y5PAGT 354, 355, 356

N

Net Classes 183, 371
Net Inspector 373
netlabels 382
NextPCB 67, 144, 199, 218, 219, 283, 299, 317
NextPCB Active Manufacturing 319
nextpcb.com 43
NextPCB online Gerber viewer 319
NextPCB's DFM tool 72
NextPCB's KiCad plugin 319
NextPCB's online Gerber viewer 232
ngspice 436
Not in schematic 215

O

origin 421
oscilloscope 450
Oshpark 144, 218
oshpark.com 43
outline 103

P

PCB editor 142
PCB Manufacture 221
pcbnew 475, 476
PCB version numbers 337
PCM 365
Pin Conflicts Map 351, 353, 354
Pin not connected 350
placement technique 106
Plot 142
plotter 142
plugin 365
Plugin and Content Manager 365
Plugin Class Definition 483
plugin directory 485
Plugins 482
Pomodoro 73
POWER_FLAG 276
Preserve 400
project archiver 458
project templates 452
Project Template Selector 454
PWR_FLAG 37, 91, 186
Python 472, 486
Python API 472, 478, 480, 486
Python Console 474, 478
Python scripting 473

Q

QFN 392
QFN28 259
QRCode 395

R

ratnest 246
real-time clock 158
rechargeable battery 24
reference BOM 257
reference designators 131
Reference designators 57, 312
Refresh 480
Refresh Plugins 485
Replace All 331
repository 398, 406
resistors 114
rough outline 108
route 48, 388
routing 122, 228
RTS 279
Rule Area Properties 309

S

schematic design workflow 26, 88
Schematic Hierarchy 179
Schematic Setup 335
Schottky diode 30
Schottky rectifiers 29
SCK 240
SCL 240
scripting 472
Scripting Console 478
S-Expressions 344, 345, 348
share your project 426
silkscreen 56, 129, 134, 139, 143, 210, 217, 301, 312, 313, 337
simulation probes 444
simulation settings 445
simulator plots 449
single-track 381
skew tuning 389
slide switch 114
SMD 158
Snapeda 28, 63, 77, 81, 162, 167, 259, 322, 354, 355, 357
Snapmagic 260
SOIC 392
solar panel 24
solar power 23
spacing 388
SPI 84, 240
SPICE 435, 436, 438, 439, 444
SPICE code 441
Spice Model 443
SPICE simulation settings 444
SS12D07VG4 147
SS14 30
status 413
step 63
STEP file 64
STEP files 63
Symbol Fields Table 174, 263
Symbol Libraries 81

System template 453

T

Texas Instruments 8051
355
text label 301
Text properties 335
text variable 80, 334, 335, 336
Text Variables 332
theme 429
Tiny Solar Supply 23, 27
TQFP 223
Track length tuning 385
tracks 122
Tracks and Vias overrides 48
Track Width calculator 463
transient analysis 449
transistor 24
Transistors 322
Tune skew 390
two-layer board 122

U

UART 74, 259
Ultralibrarian 28, 63
un-archive 458
Unconnected 278
USB 74, 259, 285
USB connector 322

V

version control 398
violations 62, 230, 315
Violation Severity 43, 342, 341, 344, 348
violation severity classifier 341
Violation Severity settings 341, 342

W

Walk around 197, 304
warnings 55, 140, 350
Warning: Text thickness out of range 139
wire tool 37
wiring 36

X

XSPICE 436

KiCad Advanced Projects and Recipes

Mastering PCB design with real-world projects

This book builts on *KiCad Like a Pro Fundamentals and Projects* and aims to help you practice your new KiCad skills by challenging you in a series of real-world projects. The projects are supported by a comprehensive set of recipes with detailed instructions on how to achieve a variety of simple and complex tasks. Design the PCBs for a solar power supply, an LED matrix array, an Arduino-powered datalogger, and a custom ESP32 board. Understand the finer details of the interactive router, how to manage KiCad project teams with Git, how to use an autorouter on 2 and 4-layer PCBs, and much more.

KiCad 8 is a modern, cross-platform application suite built around schematic and design editors. This stable and mature PCB tool is a perfect fit for electronic engineers and makers. With KiCad 8, you can create PCBs of any complexity and size without the constraints associated with the commercial packages.

Here are the most significant improvements and features in KiCad 8, both over and under the hood:

- Modern user interface, completely redesigned from earlier versions
- Improved and customizable electrical and design rule checkers
- Theme editor allowing you to fully customize the look of KiCad on your screen
- Ability to import projects from Eagle, CADSTART, and more
- An improved and tightly integrated SPICE circuit simulator
- Autorouting with the Freerouting plugin
- Filters define which elements of a layout are selectable
- Enhanced interactive router helps you draw single tracks and differential pairs with precision
- New or enhanced tools to draw tracks, measure distances, tune track lengths, etc.
- Enhanced tool for creating filled zones
- A customizable coordinate system facilitates data exchange with other CAD applications
- Realistic ray-tracing capable 3D viewer
- Differential pair routing
- Rich repositories of symbol, footprint, and 3D shape libraries
- Python scripting API for programmatic customization and extensions.
- Improved footprint wizard for fast custom footprints

Dr. Peter Dalmaris, PhD is an educator, an electrical engineer and Maker. Creator of online video courses on DIY electronics and author of several technical books. As a Chief Tech Explorer since 2013 at Tech Explorations, the company he founded in Sydney, Australia, Peter's mission is to explore technology and help educate the world.

Tech Explorations
txplore.com

Elektor International Media
www.elektor.com

ISBN 978-3-89576-628-2
9 783895 766282

elektor
design > share > earn