ゼロから始める
mruby デバイス作り (Light 版)

— L チカのその先へ —

Kishima Craft Works　著

無料頒布
2025 年 7 月 19 日　Light 版 ver 1.0

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、®、© などのマークは省略しています。

Light版公開に際して

本書は、2019 年に執筆した同人誌になります。

現在は 2025 年 7 月。執筆した当時は Ruby でマイコンを触ったりする人はあまり多くありませんでしたが、hasumikin さんによる PicoRuby の登場をきっかけに、Rubyist の皆さんの中にも電子工作に興味を持たれる方が増えました。

LED を光らせたり、チュートリアルにある部品を動かしたり、まではすぐできるかと思いますが、その後何をしたらよいのか、わからなくなってしまう、ということがよくあるように思います。

私も最初は何をしたら良いのかわからなかったですが、色々な方のアプローチの仕方を見て、楽しみ方を知っていくことができました。

せっかく電子工作に興味を持ってもらえたのであれば、最初で止まらずにどんどん創意工夫で広がる世界に羽ばたいてもらいたい、という思いで、昔執筆した本書を一部内容を削って公開することにしました。

mruby を使うことを前提に書かれていますが、設計段階の話などは、PicoRuby を使ったプロジェクトでも参考になると思います。

書いたのがだいぶ前なので、使っている SDK やツールのバージョンが今となっては古く、そのままでは試すことが難しい部分はあるかと思いますが、大まかな使い方の部分は変わらないと思うので、適宜現状に合わせて読み替えていただければ幸いです。

自分が作りたいものを作る、本書がその助けになればと願っています。

はじめに

こんにちは！ 本書を手にとっていただきありがとうございます。kishima と申します。

今回はこれまでの電子工作の集大成として、オリジナルデバイスを作ってみようとした過程で得た知見を本にまとめてみようと思います。

想定する読者

本書では、本格的な電子工作はやったことはないけど自分で考えたハードウェアを作ってみたい、という方や、mruby をそういったオリジナルのデバイス上で動かすための方法を学んでみたい、という方を対象にしています。

筆者は組み込みソフト開発に関連した仕事に携わってきたので、どうやってそういったハードが作られているのか、なんとなく知っていますが、自分でハード関連の設計をした経験はまったくないので、自分のこれまでの経緯を踏まえて、初心者視点で解説していきたいと思います。ネットや入門書を参考に独学で学んだ内容が大半なので、もしかすると専門の設計者の方からすると不正確な記述もあるかもしれませんが、どうかご容赦頂ければと思います（特に電気回路や基板設計まわり）。

自作キーボードや、RaspberryPi や Arduino の開発キットなどをはんだごてを使って組み立てて動かしてみたことはあるけど、その先にどうしたらよいのかわからない、といった方にも理解してもらえるようにコンテンツを選びました。

本書では設計ツールの使い方も含めて解説していきますが、本格的にそれらの解説を書こうとするとそれだけで膨大になってしまうので、最低限必要な部分と筆者がつまづきやすいと感じた部分に絞って解説していきます。

読者の方も本書の中で示しているリファレンスを読みつつ、自分で手を動かしつつ使い方を覚えていくことを期待しています。

あったほうが良い知識、経験

- 基本的な電子工作の経験
  - RaspberryPi や Arduino のような一般的な開発ボードに LED やセンサーなどを繋いでみたり、はんだごてをつかって自作キーボードを組み立ててみた経験はあったほうがよいです。本書ではその周辺の解説は省いています。
- 初歩的な電気回路の知識
  - オームの法則やキルヒホッフの法則のような中学校で学ぶ程度の知識を前提としています。
- mruby の基本的知識
  - mruby を自分でビルドしてみて、Hello World してみたり、mrbgem を導入してみた程度の知識はあったほうが良いと思います。
- C 言語の知識
  - mruby は C 言語で実装されているので、移植にあたって C 言語はそれなりに理解している必要があります。

なくてもよい知識

- 深い電子回路の知識
  - 設計対象のボードの開発において電子回路的に高度な技術は使いません（そもそも筆者にその能力がないです・・・）。できるだけ平易に説明してみたつもりです。
- 深い組込みソフトの知識
  - mruby は組込み向けの言語だ、という話を聞いたことがある方もおられるかもしれませんが、本書の範囲では純粋に C/C++ 言語で書かれたプログラムについて語るだけなので、組込みソフト固有の知識は不要です。

本書の開発環境

自宅では Windows のデスクトップを使ってますが、執筆によく使っているのは MacBook なので、執筆作業の都合もあり、今回は MacBook で開発してみることにしました。

- MacBook Pro (Late2013)
- macOS10.13

開発に使用しているソフトはそれぞれ Windows にも対応しているので、Windows の開発にも適用できると思います。

本書で利用するソフトウェアのバージョン

- KiCad
  - KiCad 5.1.2 macOS10.13 向けを利用します。
- ツールチェイン/コンパイラ
  - Xtensa Toolchain version: crosstool-ng-1.22.0-80-g6c4433a
  - Compiler version: 5.2.0
- ESP-IDF
  - v3.2.2 を利用します。
- mruby
  - mruby 2.0.1 を利用します。
- FabGL
  - v0.6.0 を利用します。

本書の構成

第 1 章「オリジナルデバイスの設計の道」

導入として、執筆の背景、開発対象、設計の流れについて

第 2 章「設計図を作ろう」

仕様決め、デバイス選定、ブロック図について

第 3 章「初心者でも回路は怖くない（はず）」

KiCad を使った回路の設計について

第 4 章「自分だけの基板を」

KiCad を使った基板の設計、発注について

第 5 章「mruby を思いのままに！」

オリジナル基板でのビルド環境の構築および mruby の移植について

第 6 章「mruby を自分のデバイス色に染める」

オリジナル基板のための mrbgem の開発方法について

サポートページ

筆者のブログで誤記等の訂正や、関連ソースコードへのリンクなどを貼ったりしています。本書に関する最新の情報は、こちらを参照ください。

`https://silentworlds.info/my-books/`

巻末の著者紹介に記した Twitter 等の SNS で話しかけて頂いても大丈夫です。

免責事項

本書の内容を参照してのプログラムの作成、電子工作等は、必ずご自身の責任と判断によって行ってください。本書の内容を参照することによって生じた結果に対して、著者はいかなる責任も負いません。

目次

第1章　オリジナルデバイスの設計の道

本章では、まず本書の目的と、本書の解説の流れについてまとめてみたいと思います。

1.1　本書の目的

本書は、筆者が開発しているデバイスの開発過程を順番に解説することで、自分の思い描いたデバイスを形にして、その上でオリジナルのシステムを動かすまでの一連の流れを理解してもらうを目的としています。

説明対象の範囲が広いので、それぞれの解説は不十分な点もあるかもしれないですが、まずは全体の見通しを持ってもらえることを目標としています。概要がわかれば、自分に何が足りていないのか分かり、次のステップに進みやすくなると思っています。

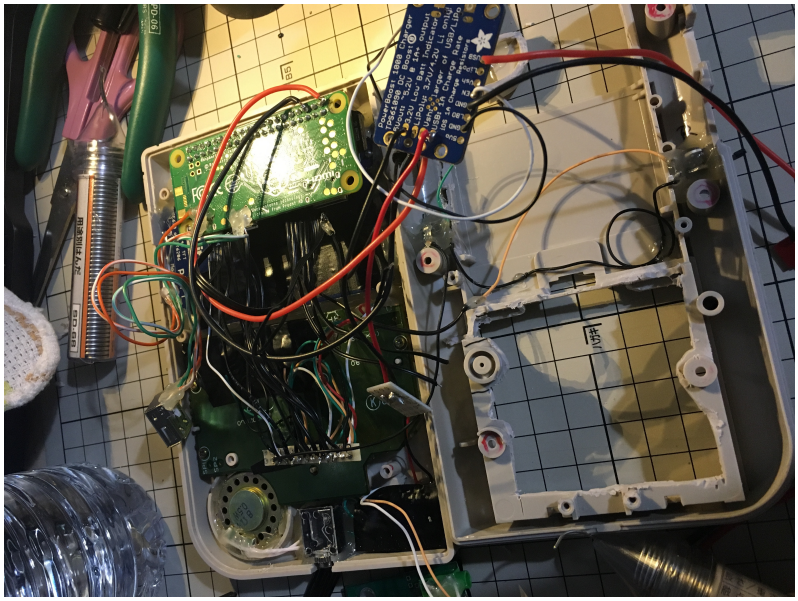
1.2　執筆の経緯

筆者は 2017 年ごろから趣味で電子工作を本格的にやるようになりました。色々なものを作っていく中で、少しづつ経験値が増えてきましたが、最初はどこから手を付けたらよいのかもよくわかっておらず、戸惑いながらあれこれネットの情報を漁ったりしていました。ハード（回路周辺）関係の仕事をされている方にとっては当たり前のようなことでも、知らない人間にとってはハードルが高く感じるものです。そんな過去の自分のために、オリジナルデバイスを作るにあたって、知っていればよかったと思える情報を整理しようと思ったのが執筆のきっかけです。

最近は自作キーボードが流行していて、電子工作に馴染みのなかったソフト系のエンジニアの方もはんだごてを握ったりする機会が、増えたりしてきました。そんな方はきっと過去の自分と同じように、漠然と作ってみたいものは頭の中にあっても、どうやって作ればよいのか分からない状態だと思うので、そういう方の助けになればと思っています。

本書では説明の題材として、mruby によるゲーム開発＆実行環境を取り上げています。これは子供の頃に思った、ファミコンとかスーパーファミコンはどうやって動いているんだろうか？ 自分で作れたらカッコいいな、とか、Family BASIC 欲しかったなあ、といった記憶がフラッシュバックしているためです (^^)

結果、図 1.1 のように初代ゲームボーイを手改造して、ラズパイを搭載してみたりしましたが、自分で１から作ってみたい！ という欲求は収まることを知りませんでした。



▲図 1.1: 初代ゲームボーイを改造して Raspberry Pi zero を搭載したもの

1.3　本書の解説対象範囲

本書では以下のような技術要素について解説しています。

- オリジナルデバイスの仕様決めの考え方
- 電気回路の初歩
- KiCAD による回路図の書き方
- KiCAD による基板データの作り方
- 基板を海外業者（Elecrow）に製造委託する
- EPS32 でのソフト開発
- FabGL on ESP32 による映像音声出力
- mruby の ESP32 への移植
- EPS32 向け mruby のライブラリ (mrbgem) の作り方

1.4　設計の流れ

本書では以下のような行程でオリジナルのデバイスを作る過程を追っていきます。

- デバイスの設計（第 2 章）
- 回路図を書く（第 3 章）
- プリント基板を CAD でデザインする（第 4 章）
- プリント基板を製造委託する（第 4 章）
- mruby を利用したファームウェアを開発する（第 5 章、第 6 章）

1.4.1 基板をどう作る？

本書では、将来的に基板をコミケのような即売会で頒布することも視野に入れて、基板を中国の基板製造メーカーに委託して製造してもらう前提で、設計を進めます。

１枚ものであれば、ユニバーサル基板に手配線してもよいかもしれないですが、手配線は手配線で中々骨の折れる作業なので、今まで基板を委託製造してもらったことがない方も一度試してみるのも楽しいと思います。図 1.1 のような複雑な配線を基板上できれいにまとめられるのは、手配線の面倒さを知っている方であればありがたみが分かるのではないでしょうか。

プリント基板とは部品間の配線をある基板の銅薄膜のパターンで行ったものです。各部品は通常はんだで基板に電気的にも物理的にも接着されます。基板を複数階層で作ることで複雑な配線も可能になります。本書では製造コストも考慮して２層基板（基板の表と裏面に配線できる）を前提として設計しています。

第2章　設計図を作ろう

あなたは何を作りたいのか？

そこがすべての原点となります。あなたのはんぱないパッションを設計図に落とし込んでいきましょう！

2.1　作りたいものを決める

前提として、何かを作りたいという気持ちは、すべてあなたの心から生まれるものなので、他人がとやかく口を出すものではないと思います。でも、漠然と何かを作ってみたいけど、何を作ればいいのかわからない、どの程度まで素人で手が届くのかわからない、というところで次の一歩が踏み出せないという方も多いのではないかと思います。

そんなときは、普段の生活や趣味の活動から「こういうのあったらいいなあ」というものを考えてみたり、イベントに足を運んで、誰かの作品を見て「自分もこういう感じの作ってみたい！」とか「自分ならここをこうしてみたい！」というインスピレーションを得たりするのがネタ出しの方法として良いと思います。

筆者も Maker Faire Tokyo に足を運んでみたり、電子工作好きな方が集まるオフ会に参加してみたりして、すごい人のすごい活動に大いに触発されました。

2.1.1　インスピレーションを受けられそうな場所

筆者が参加した or 参加してみたい場をあげてみます。

技術書典

言うに及ばず、これだけ幅の広い技術について、本を書いてみたいと思っている人が一同に集まる場というのは、そこに参加するだけで心が高揚します。

Maker Faire Tokyo

毎年ビックサイトで開催されている、Maker の祭典です。創意工夫に満ちた作品が展示されていて、見ているだけで楽しいですし、自分も何か Make したい！ という気持ちにさせてくれる場です。

コミックマーケット

あのコミケです。二次創作同人誌が有名ですが、技術書や各種の考察本、手芸品など色々な一次創作物も多くとてもおもしろいです。同人ソフト、デジタル（その他）、評論・情報、あたりのジャンルが好きです。

NT*

ニコニコ技術部の有志の方によるイベントです。最近では地方各地で開催されているようです（NT 金沢、NT 京都など）。なかなかタイミングが合わず参加できていないのですが、ニ

コニコの名を冠するだけあってユニークな作品が多く、とてもおもしろそうです。[*1]

hwoff

@pao5656 さんが毎年都内で開催している、HW や組み込みや電子工作が好きな人が集まる宴会です。ディープな方々とお話できる楽しい場です。[*2]

2.1.2 筆者の場合：Family mruby

以下、自分語りで恐縮です。

筆者が中学生のころ最初に触ったプログラミング言語は、富士通の FM-TOWNS 上で動く、F-BASIC386 という BASIC でした。この BASIC のすごいところは、当時流行りのマルチメディアに強く、FM-TOWNS の機能を生かした描画関連の命令が充実しており、スプライトを動かしたり、音楽の再生等も簡単にできました。N88-BASIC などでは難しいような見栄えの良い画面が簡単に作れるので、これを使って簡単なゲームを自作しながら、プログラミングの楽しさにはまっていき、最終的には自分が今の職業を選んだきっかけにもなりました。最近では小学校でプログラミング教育も始まるようで、親戚の子にプログラミング教えて欲しい！ なんて頼まれる出来事もありました。

ここで思ったのは、もし未経験者にプログラミングの楽しさを知ってもらおうとするのであれば、気軽に画像や音を動かせるインタフェースを備えた環境が必要、ということでした。最近は色々なプログラミング言語が無料で利用できるようになり、選択の幅は無限にあると思います。でも、モダンな言語で画面で画像を動かしたり音を出したりするには、OS や HW の環境に合わせたライブラリのインストールが必須だったりするので、少しハードルが高いように思います。

ならば、図 2.1 のような、特定のプログラミング言語に特化して、画や音を制御する機能を標準で盛り込んだ開発環境を HW 一体として作ったら良いのでは！ と思いました。

▲図 2.1: 作ってみたいものイメージ図

そして調べてみると、同様のコンセプトのものとしては、以下のようなものがありました。

- IchigoJam(株式会社 jig.jp より発売) [*3]

[*1] http://wiki.nicotech.jp/nico_tech/

[*2] 2019 年の hwoff: https://atnd.org/events/102504

[*3] 筆者が最近はじめた mruby meetup という勉強会で、IchigoJam を開発された福野さんが参加されて直接お話できて光栄でした。何でもやってみるものですね！

    - BASIC プログラミング環境を 1 マイコンで実現している（モノクロ NTSC ビデオ出力、PS/2 キーボード対応）
      `https://ichigojam.net/`
- MachiKania
    - BASIC プログラミング環境を 1 マイコンで実現している（256 色 NTSC ビデオ出力、PS/2 キーボード対応）
      `http://www.ze.em-net.ne.jp/~kenken/machikania/`
- Uzebox
    - C 言語で開発したゲームをロード＆実行する環境を 2IC で実現している（256 色 NTSC ビデオ出力、スーパーファミコンゲーム PAD 対応）
      `http://belogic.com/uzebox/index.asp`

IchigoJam は量産もしており、スタンドアローンなプログラミング環境のコンセプトの大先輩にあたります。機能はシンプルですが、手作りもできるくらい簡単な構成が入門用にとてもよいと思います。

Maker Faire Tokyo2018 で出会った MachiKania は、カラーで高解像度の映像出力を PIC32 の 1 マイコンで実現しており、技術的な面で感銘を受けました。筆者はこれまで mruby 推しの活動をしてきた経緯もあり、mruby でも同じようなことをやってみたい思うようになりました。

お家で気軽に BASIC を学べる環境の代表が、任天堂のあの Family Basic なのであれば、自分は「**Family mruby**」を作ってみようと思ったわけです。

――というのは理由の一部で、実際のところ、筆者自身が子供のころにこんなのがあったら良かったなあ！ というごくごく個人的な想いがモチベーションの大半を占めていたりもします。何であれ、大切なのは作るその人自身のパッションです！

2.2 必要な機能の整理

さて作りたいものが決まったところで、それを実現するためにどんな機能が必要なのか整理してみましょう。いわゆる要件定義というやつです。

洗い出した機能のうち、明らかに自力で実現不可能なものが含まれていれば、それは諦めざるをえませんが、諦めずに英語でググったりしてみると世界の誰かがその課題を解決してくれていたりするかもしれないので、諦めない心も大切です。

では Family mruby に必要な機能を挙げてみましょう。

- mruby によるプログラミング
    - 一番欠かせないポイントです。そのデバイス一つで mruby のプログラミングを楽しめるようにしたいです。
- 映像出力
    - 解像度は多少低くてもよいので、カラーで表示できるようにしたいです。ある程度一般的なモニタに出力したいです。
- キーボード入力
    - プログラミングをするにはやはりキーボードも必要です。
- 音声出力
    - ゲームも作れるようにするなら画像だけでは寂しいので、アナログの音声信号を出力したいです。

- プログラムの保存
  - 書いたプログラムは保存して、あとで呼び出せるようにしたいです。SD カードあたりが適当そうです。
- WiFi による無線通信
  - できれば WiFi 通信機能があれば、色々な外部機能との連携が可能になりそうです。
- 個人でも安く入手できる部品で作れる
  - これは機能の要求ではないですが、将来的に他の人にも使ってもらいたいので、安く作れることが前提となります。

2.3 実現方法を考える

ではそれぞれの機能をどうやって実現するか考えてみましょう。

やりたいことよって求められる技術的な経験や知識のレベルが変わってきますので、地道に一つずつ調べていく必要があります。HW に詳しい方が身近に居れば、その方に相談してみるのが早いかと思いますが、たとえ聞ける相手がいなくても、自分がやりたいと思ったことは、だいたい世界の誰かが同じようなことを考えていることが多いです。ググれば何かしらの知見が見つかるものです。

また先に紹介したイベントで先駆者の皆さんの作品を見ていると、どんなことが個人レベルで可能なのか、発想の引き出しにもなると思います。

注：これまでに挙げた要件は実は、ラズパイを使えばすべてサクッと実現できてしまったりもするのですが、Linux 上で実現するのは何となく負けた気がするので、ここはあえて１マイコンでの実現にこだわりたいと思います。

Linux を使用しない他の理由としては、起動を素早くしたい、抱えるストレージは最小限にしたい、低レイヤの HW の制御も自分で理解ししたい、といった点もあります。

Family mruby 実現のために開発するハードのコードネームを、「Narya（ナルヤ）ボード」とすることにします[*4]。

2.3.1 mruby によるプログラミング

mruby を使用する際の課題は CPU の性能と ROM と RAM の容量です。その中でも動的にオブジェクトをどんどん生成していく mruby の場合、特に RAM の容量が重要になってきます。mruby の実行に必要な RAM は数百 KB 程度と言われています。この値はあくまで目安なので、当然プログラム中で生成するオブジェクトの数によって、1MB 以上のメモリを消費する可能性もあります。サーバ上で動く Ruby で書かれた Web アプリでは数百 MB 以上のメモリを消費することもあると思います。

一方で、電子工作でよく使われるボードの ROM と RAM について表 2.1 に簡単にまとめてみます。

表を眺めてみると分かりますが、よく電子工作で用いられる ArduinoUno では mruby を動かすにはかなりメモリが不足していることがわかると思います。一方で Linux を動かすことが前提の Raspberry Pi Zero ではメモリの容量が圧倒的に多いことも分かります。今回は Linux に頼らずやっていきたいと考えているので、その他の選択肢では ESP32-WROOM-32 が候補となりそう

[*4] ナルヤは筆者が好きな指輪物語に出てくる三つの指輪の一つ、ルビーが嵌められた炎の指輪です。ガンダルフが付けていたりします。mruby つながりと、周囲を鼓舞する力を持っていることから名前を借りました

▼表 2.1: 開発ボードの ROM と RAM の比較

Board(MCU)	ROM size	RAM size
Arduino Uno(ATmega328P)[*5]	32KB	2KB
ProMicro(ATmega32U4)[*6]	32KB	2.5KB
IchigoJam(LPC1114)	32KB	4KB
MachiKania(PIC32MX370F512H)	512KB	128KB
ESP32-WROOM-32	4MB	520KB
Raspberry Pi Zero	>1GB	512MB

です。

ROM の 4MB は十分そうですが、RAM の 512KB は mruby をばりばり動かすにはちょっと不安が残ります。

マイコン向けとして、"mruby/c"という、言語仕様を限定して、より軽量にした mruby の VM 実装も存在していますが、今回はできるだけリッチな Ruby の言語仕様を実現したいので、mruby をなんとかそのまま動かしたいところです。

検討を始めた当初はどうしたものかと思っていましたが、2018 年になって、ESP32 シリーズに ESP32-WROVER-B というラインナップが加わりました。これは RAM として SPI 接続の PSRAM 8MB が増設されたもので、日本の技適マークもついている代物でした。「それを聞きたかった」と内心舞い上がったのは言うまでもありません。これだけの RAM があればかなり余裕を持って mruby のプログラムを走らせることができそうです。

2.3.1.1 ESP32-WROVER-B について

Family mruby に、ESP32-WROVER-B を採用すると決めたわけですが、その仕様を確認しておきます。ESP32-WROVER-B の主なスペックは以下のとおりです。

▼表 2.2: ESP32-WROVER-B のスペック

Type	Spec
CPU	ESP32-D0WD(dual core)
CPU clock	80MHz-240MHz
RF function	WiFi, Bluetooth
Module interfaces	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM,I2S, IR, pulse counter, GPIO, capacitive touch sensor,ADC, DAC
On-chip sensor	Hall sensor
Integrated crystal	40 MHz crystal
Integrated SPI flash	4 MB
Integrated PSRAM	8 MB
Operating voltage/Power	supply 2.7 V ~ 3.6 V
Minimum current delivered by power supply	500 mA
Recommended operating temperature range	– 40 ° C ~ 65 ° C

[*5] 趣味の電子工作では最もよく使われているボードといってもよいのではないでしょうか

[*6] USB との接続機能があるので、自作キーボードでよく使用されています

Module interface については、コラムを参照ください。

【コラム】ESP32-WROVER-B のインタフェース

ESP32 はたくさんのインタフェースを備えています。電子工作でよく用いられるものはほぼ網羅していると言ってもよいのではないでしょうか。各機能について、略称に馴染みが無い方もいると思うので、ここで簡単に説明します。多数のインタフェースを備えている ESP32 ではありますが、入出力に使えるピンの数はすべての機能を同時にカバーするには全然足りません。そのために ESP32 ではソフトウェアの指定で、どのピンがどの機能を持つのか「ある程度」動的に設定できます。

SD card:
いわゆる SD カードとの接続機能です。

UART: "Universal Asynchronous Receiver/Transmitter"
大抵のマイコンにはついている双方向のシリアル通信機能です。制御やデータ通信用以外にも、デバッグ用の文字列の送受信にもよく使われます。相手の機器とは一対一で接続します。

SPI: "Serial Peripheral Interface"
I2C と並んでよく見かける汎用的なシリアル通信を行う機能です。複数のピンを組み合わせて動作します。I2C より高速な通信を行うことが可能な場合が多いです。ホストは一つである必要があります。

SDIO: "Secure Digital Input/Output "
SD カード互換のプロトコルでデータの入出力を行う機能です

I2C: "Inter-Integrated Circuit"
こちらもよく見かける、汎用的なシリアル通信を行う機能です。読むときには、"I squared C"という読み方をします。単にアイツーシーとも読むこともあります。通信速度は遅めですが、通信線が 2 本だけで、複数の IC をぶら下げることができるので、扱いやすいです。ホストは一つである必要があります。

LED/Motor PWM: "Pulse Width Modulation"
PWM 制御ができるピンです。PWM 制御とは、高速に切り替わる電圧の高低の比率を変えることで、出力の大小を表現する方法です。モーターや LED の明るさの制御などに用いられます。

I2S: "Inter-IC Sound"
デジタル音声データの入出力によく用いられるシリアル通信機能です。Family mruby の実装では大活躍することになります。

IR: "Infrared"
赤外線通信の機能です。

pulse counter
電圧の高低によるパルスをカウントすることができるピンです。ロータリエンコーダの入力を捉えたりするために用いられれます。

GPIO: "General Purpose Input/Output"
ソフトウェアの制御で電圧の高低を制御できるピンです。

capacitive touch sensor:
読んで時のごとく、タッチセンサの入力として使えるピンです。内部でそのピンの静電

気容量を測って、人体の接触を判断しているようです。

ADC: "Analog Digital Convertor"

電圧を一定のしきい値によるデジタル値としてではなく、電圧値をそのまま読み取ることができる機能を持ったピンです。アナログな電圧値で測定結果を出力するセンサとの接続などに用います。

DAC: "Digital Analog Converter"

ADC とは逆で、所定の電圧値を出力することができるピンです。連続的に制御することで音声波形を作ることもできます。

JTAG

デバッグ用の通信規格です。ESP32 の場合 OpenOCD というツールと接続することで、gdb によるデバッグが可能になります。

参 照 元：`https://www.espressif.com/sites/default/files/documentation/esp32-wrover-b_datasheet_en.pdf`

2.3.2 映像出力、キーボード入力、音声出力

多機能な SoC であれば、映像出力、キーボード入力、音声出力に対応した機能を備えていて、それらを使いこなせばよい話でありますが、今回使おうとしている ESP32-WROVER-B にはそのような便利な機能は（音声出力を除いて）付いていません。そのためそれらを自力でなんとかする必要があります。

特に映像信号については、スクリーンバッファから C 言語で実装したロジックで映像信号を生成しているのでは、追いつかないくらい大量の情報を高速に処理する必要があります。例えば、１ピクセル１バイトと仮定した場合、320x240 の画像を 60fps で出力するには、単純計算で毎秒 4MB 超の画像データを処理して、かつ映像信号の規格に則って、所定のタイミングで IO を制御する必要があります。これはマイコンの CPU にはちょっと荷が重い処理です。IchigoJam や MachiKania では HW の機能を活用して NTSC の信号をリアルタイムに生成しています。NTSC とは、よく昔のテレビやビデオデッキの映像入力に用いられていた黄色の端子のケーブルの中を流れていた信号です。１本の信号線でアナログな電圧の高低でカラー映像信号を伝達しています。

当初は類似のオープンソースなライブラリを参考にして自分用にカスタマイズして搭載しようかと考えていましたが、検討を始めたのと同じくらいのタイミングでまさに自分のニーズに完全に一致するライブラリを作られている方を発見しました。

それは、「FabGL」[*7]というローマ在住の Fabrizio Di Vittorio さんという方が開発したライブラリです。この方のライブラリがなければ、Family mruby の開発にはもっと時間が掛かり、スペックダウンも避けられなかったと思います。

2.3.2.1 FabGL の機能

ここで、FabGL の機能を簡単に説明します。

VGA 映像出力

[*7] `https://github.com/fdivitto/fabgl`

I2S 機能を活用して、VGA 信号を出力することができる機能です。最近は HDMI に取って代わられつつありますが、D-Sub15 ピンの端子が付いているモニタに出力可能です。色数は 8 色または 64 色、解像度は最大 800x600 まで対応しているようです。

PS/2 キーボード、マウス入力

IchigoJam も PS/2 キーボード入力をサポートしていましたが、FabGL も同様にサポートしています。加えて同時に PS/2 マウスにも対応しています。最近ではキーボード、マウスは USB 接続が一般的ですが、デジタル信号として扱いが容易なので PS/2 を採用しています。PS/2 接続には Mini DIN 6pin のコネクタを用います。店頭では見かけませんが、Amazon 等ならまだ購入可能です。

内部では、ESP32 のコプロセッサで信号を処理することで、メインの CPU に負荷を与えずに処理を行っています。

アナログオーディオ出力

音声情報をアナログの電圧波形で出力します。端子にはイヤホンでよく見かけるミニプラグが刺さるようにすれば、アンプ付きスピーカーで再生できるでしょう。内部では I2S と DAC の機能を利用して、CPU の負荷を最小限にしながら出力を行っています。

FabGL は、このように欲しい機能を過不足なく提供してくれている素晴らしいライブラリです。

2.3.3 プログラムの保存

SD カードには SPI による接続モードがあり、これを使うと SPI の 4 本の信号線で SD カードにアクセスできますので、これを使用することにします[*8]。

2.3.4 WiFi による無線通信

WiFi は ESP32 の目玉機能でもあるので、活用することにします。しかし WiFi 機能を使用する場合は、処理の負荷が高まることもあり得るため、同時に使う機能に影響がでないかどうか注意が必要です。

2.3.5 個人でも安く入手できる部品で作れる

ESP32-WROVER-B 単体でほぼすべての機能が実現できているので、その他に必要になる部品として比較的高価なのは USB-UART 変換の IC や、各種コネクタ、電源用の IC などぐらいです。

もっとも単価が高い部品となるであろう ESP32-WROVER-B の価格は、秋月電子のサイトでは以下のとおりです。

- ESP32-WROVER-B の価格
  - 530 円（2019/8/26 現在）

これは ESP32-WROOM-32 よりも更に安い価格で、正直なんでこんなに安いのか訳がわかりません。いずれにせよ活用しない手はありません。

2.3.6 暗黙の要件

必要な機能の部分で挙げていなかったですが、実際に開発するにあたって必要になる項目を追加しました。

[*8] ESP32 でどんなことができるのかについては、Arudino のサンプルプログラムを一通り眺めてみるとよく理解できると思います

2.3.6.1　ESP32 へのプログラムの書き込み、デバッグ方法

SD カード上のスクリプトを実行するには、SD カードにパソコンで書き込めば大丈夫ですが、ESP32 には最初は何もソフトが書き込まれていないので、ESP32 へのソフト書き込み手段が必要です。プログラムを ESP32 に直接書き込む場合、GPIO を所定の状態にセットした上で、UART を用いて書き込みます。そのために UART が必要になりますが、UART はそのまま開発用 PCB には接続できません。UART を PC に接続するため、USB シリアルコンバーターの IC を搭載することにします。

また念のために JTAG のピンを利用できる形で残しておけるような構成を考慮します。

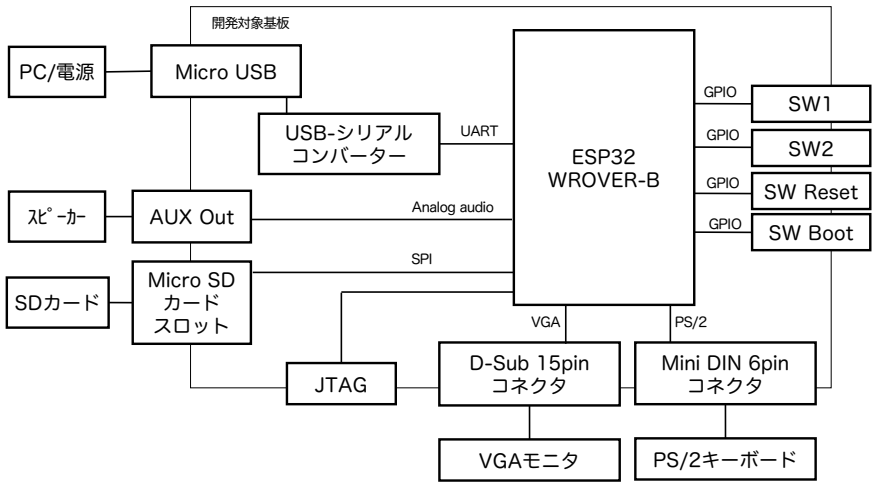
2.3.6.2　電源

当たり前ではありますが、電源がないと電子回路は動きません。電源をどこから入力するか考える必要があります。今回は準備が容易な USB を電源とすることにします。

2.4　ブロック図を描いてみる

これまで Family mruby に必要な機能と、それを実現する大まかな方針を確認してきました。各種部品と機能がどんな感じで繋がっているのか、脳みそを整理するために基板（Narya ボード）のブロック図を書いてみましょう。実際に書いてみると検討もれや課題に気づくきっかけにもなったりします。

これまでで検討を整理して描いたブロック図を図 2.2 に示します。



▲図 2.2: Narya ボードのブロック図

ブロック図を書くときには、まずメインのマイコンを中心に描いて、それから必要な機能ブロックと入出力の箱を置いていくとよいと思います。

2.5 ピンの配置を決める

ブロック図を回路図に落とし込んで行く前に、ブロック図の各 IC 間の接続を具体的に考えてみましょう。そのために各 IC の仕様の詳細を確認していきます。

2.5.1 使用する部品の仕様を理解する

電子部品の仕様は通常、データシートと呼ばれるドキュメントにまとまっています。データシートにはハード的にどのようにその部品を扱うべきか、どんなことをしてはいけないのか、が書かれています。初めて見るととても難しく感じますが、電子工作レベルではある程度おおざっぱでもなんとかなるので、あまり怖がらずに読んでいきましょう。今回は必要最低限のポイントに絞って把握するべきポイントを説明します。今回のプロジェクトでは、ほとんどの機能は ESP32-WROBER-B で実現していますので、ESP32-WROVER-B のデータシートを読んでみます。

2.5.1.1 ESP32-WROVER-B のデータシートを読む

ESP32-WROVER-B のデータシートは以下にあります。

`https://www.espressif.com/sites/default/files/documentation/esp32-wrover-b_datasheet_en.pdf`

ESP32-WROVER-B は、ESP32 という IC に周辺部品を足してパッケージにしたものなので、必要に応じて ESP32 のデータシートも参照必要です。

`https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`

ESP32-WROVER-B のデータシートは、以下のような章構成になっています。その意味を簡単に説明してみます。

1. Overview
   概要です。
2. Pin Definitions
   各ピンの機能の割り当ての説明です。基板を作るにはよく理解しておく必要があります。
3. Functional Description
   機能の説明です。必要な部分は一読しましょう。
4. Peripherals and Sensors
   ESP32-WROVER-B が提供する基本的な機能の説明です。内容は ESP32 と同じなので、そちらのデータシートを参照しています。
5. Electrical Characteristics
   電気的な特性です。入出力の電圧の範囲などを確認します。特に既定値を超える電圧を掛けると部品が壊れたり、発火したりする可能性があるので、ちゃんと確認しておいた方がよいです。
6. Schematics
   ESP32-WROVER-B の金属製パッケージの中身に隠れている回路です。通常あまり意識しなくても大丈夫ですが、特殊な使われ方をしているピンについて実装を確認するときに役に立ちます。

7. Peripheral Schematics
   ESP32-WROVER-B を利用した場合の、基本的な回路設計例です。回路図を作成するときに参考になります。
8. Physical Dimensions
   物理的な寸法図です。今は意識しなくても大丈夫です。
9. Recommended PCB Land Pattern
   基板を作るための接点端子の配置例です。第 4 章に関連しています。
10. U.FL Connector Dimensions
    アンテナコネクタの寸法図です。通常は利用しないと思います。
11. Learning Resources
    関連ドキュメントです。最終的には、一通り目を通しておくのが良いと思います。

2.5.1.2 電源について

データシートの５章から仕様を確認します。
まず、"5.1 Absolute Maximum Ratings"を参照して、禁止事項を確認します。

- Power supply voltage
  - 最低電圧：-0.3V　最大電圧：3.6V
  - -0.3V 以下、3.6V 以上の電圧を掛けると、壊れてしまう、ということを示しています。
- Cumulative IO output current
  - IO 出力電流の総和の最大値： 1100mA
  - 出力ピンにつながるデバイスに供給する電流が全体で 1.1A を超えたら壊れる、ということを示しています。

次に"5.2 Recommended Operating Conditions"を参照して、通常に使用する場合の条件を確認してみます。

- Power supply voltage
  - Typical:3.3V
  - Typical は典型的な、という意味です。基本的に 3.3V を電源として使用することが要求されています。
- Current delivered by external power supply
  - 最低：0.5A
  - 外部電源から供給される電流がこの程度は最低必要、ということを示しています。ポイントは最大値が規定されていない、ということです。電源投入時、瞬間的には 1A を超えるような電流が流れるそうです[*9]。
- Operating temperature
  - 最低：-40 度　最大：65 度
  - 使用する環境の推奨温度範囲です。-40 度はなかなかないとは思いますが、65 度は夏の車の中とか、強いライト等が当たる場所など場合によってはありえるので、注意必要です。

[*9] 参考：https://www.mgo-tec.com/blog-entry-esp-wroom-32-usb-inrush-current.html

【コラム】電子部品の電源電圧

電子部品の動作に必要な電圧は部品によって様々ですが、ある程度標準的な電圧が決まっています。電子工作でよく見かける電源電圧は以下のようなものかと思います。マイコンの動作電圧はマイコンの GPIO の入出力電圧の基準にもなるので、よく確認して部品を選択する必要があります。

- 12V
  - 自動車のバッテリーの電圧として見かけることがあります。車載機器の基本的な動作電圧で、シガーソケットから電源をとったりするデバイスでは 12V が電源電圧になったりします。また、RS232C というシリアル通信の基準の電圧としても見かけることがあります。
- 5V
  - ArduinoUno の動作電圧はこれです。一昔前は 5V が標準的なマイコンの動作電圧だったそうです。USB ポートから電源を取る場合も 5V なので、手軽な電源として利用できます。
- 3.3V
  - RaspberryPi の動作電圧はこれです。最近は 3.3V で動作するものが多くなっているそうです。電子工作では、5V を入力として、電圧を専用の部品で下げて 3.3V を内部の電源とするようなケースが多いようです。前述のとおり ESP32 も 3.3V が動作電圧です。
- 1.8V
  - 電子工作で見かけることはあまり多くはないですが、高速な CPU 用の電源などで使用されています。これは低電圧の方が省電力＋低発熱に効果があるためです。パソコンの CPU も 1.8V 以下で動いているようですね。

2.5.1.3 UART で使用するピン

ソフトウェアの書き込みおよび、デバッグのために用います。UART では信号を受信と送信それぞれの専用の線を用いて行います。

2.5.1.4 SPI で使用するピン

今回は SD カードの接続のために用います。ESP32 では 3 系統の SPI が利用できます。それぞれ、SPI、HSPI、VSPI と呼ばれており、所定のピンに割り当てされています（標準の SPI 通信を行う場合は、他のピンへの割り当ても可能です）。ESP32-WROVER-B では、そのうちひとつはフラッシュメモリと、PSRAM の接続で専有されているので、実際に使用できるのは残りの 2 つの HSPI、VSPI となります。

2.5.1.5 I2C で使用するピン

今回は用いませんが、SPI よりも低速な通信でよく用いられるインタフェースです。SDA（データ）と SCL（クロック）の 2 本の信号線だけで通信できるので、UART と同様に使いやすいインタフェースです。ESP32 では I2C もピンを自由に割り当て可能です。

2.5.2 ピンの割り当てを決める

ESP32 の大まかなインタフェースの仕様を理解したところで、次に ESP32-WROVER-B の各ピンに接続する信号の割り当てを決めます。この割り当てをベースに次のステップで回路図を書いていくことになります。

2.5.2.1 ESP32 のピン配置機能

ESP32 は多くの機能を持っていますが、機能の数に対して入出力のピンの数が足りていません。そのため、ソフトの設定によって、機能ごとに使用するピンを個別に設定できます。例えば ArduinoUno では基本的にある機能に対応するピンは固定です。

すべての機能やピンが自由に割り当てできるわけではないですが、自由度が高いのは大変ありがたいです。

2.5.2.2 FabGL の IO

ピンの割り当てを決めるにあたって、FabGL でどのようなピンを使用する必要があるのか確認します、

VGA 信号

VGA 信号には RGB の各種色信号と、VSYNC、HSYNC の計 5 本の信号が必要になります。64 色の場合、各 RGB 信号は 2bit で表現できるので、各々 2 本の出力ピンを使います。

PS/2 信号

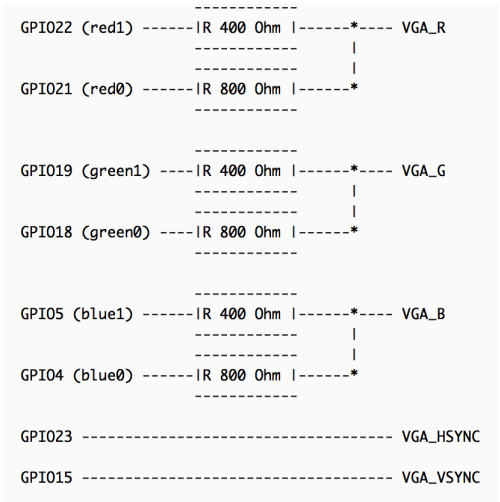
PS/2 は 2 本の信号線が必要になります。各々入出力が必要になるので、 2 本の入出力可能なピンを使います。

オーディオ信号

モノラル出力で十分なので、 1 本のピンを使用しますが、そのピンが DAC に対応している必要があります。

接続方法は、基本的に FabGL のサイト[*10]の案内に従うことにします。例えば VGA のピンの標準割り当ては図 2.3 のようになっています。

[*10] http://www.fabglib.org/



▲図 2.3: FabGL-VGA のピン配置

2.5.3 ピンの配置

最終的に図 2.4 のような配置としてみました。ESP32 は、入力限定のピンが複数あり、出力機能を割り与えられるピンは見た目ほど多くはないので、注意が必要です。配置のポイントとしては、IO15 が JTAG で使用するピンなので、VSYNC を IO27 に変更しています。

My Poject	General	ESP32-WROVER-BのIO			General	My Poject
GND		GND		GND		GND
電源		VDD33		IO23	VSPI	VGA-HSYNC
Pull Up & SW	Reset	EN		IO22		VGA-Red1
未使用	入力専用	SENSOR_VP/IO36	IN	TX	UART-TX	UART-TX
未使用	入力専用	SENSOR_VN/IO39	IN	RX	UART-RX	UART-RX
Switch	入力専用	IO34	IN	IO21		VGA-Red0
Switch	入力専用	IO35	IN	NC	-	-
PS/2 DAT		IO32		IO19	VSPI	VGA-Green1
PS/2 CLK		IO33		IO18	VSPI	VGA-Green0
未使用	(DAC)	IO25		IO5	VSPI	VGA-Blue1
DAC or I2C audio	(DAC)	IO26		NC	-	v
VGA-VSYNC		IO27		NC	-	-
HSPI(SDカード)	JTAG	IO14		IO4		VGA-Blue0
HSPI(SDカード)	Config LDO/JTAG	IO12		IO0	Boot mode	Pull Up & SW
GND		GND		IO2	Pull Down	Pull Down
HSPI(SDカード)	JTAG	IO13		IO15	Config TX/JTAG	HSPI(SDカード)
使用不可	SPI Flash用	SD2/IO9		SD1/IO8	SPI Flash用	使用不可
使用不可	SPI Flash用	SD3/IO10		SD0/IO7	SPI Flash用	使用不可
使用不可	SPI Flash用	CMD/IO11		CLK/IO6	SPI Flash用	使用不可

▲図 2.4: ピンの割り当て

第3章　初心者でも回路は怖くない（はず）

前章では、作りたいもののイメージをまとめて、大まかな完成形のブロック図を描いてみました。本章では、そのブロック図をベースに電気回路を設計してみましょう。

3.1　回路設計を始めよう

電気回路を設計すると言っても、未経験の方にとっては、ものすごく難しい専門的な世界に見えるのではないでしょうか。実際高度な回路の設計は専門的な世界なので、素人には手出しが難しい領域もあることは確かですが、ちょっとしたものであれば素人でもなんとかなると思います。本書で触れているのは中学理科の電気回路の知識でカバーできる範囲が大半と思います。本章の例を見ながらあまり構えずにやっていきましょう。

家庭用電源で使われている交流の回路や、電磁波の送受信に関わる高周波信号の回路はまた違った考え方が必要になるので、筆者は最初は手を出さずに置くのがよいと思っています。そのため本書では触れていません。

きちんと理解しないと頭がもやもやして先に進めない、という方は大学の工学部の授業で使われているような電気回路の教科書を時間を掛けて読んでみることをおすすめします。筆者も入門書だけを読んでも中々腑に落ちず、応用や基礎の本を読んでやっと理解できることが多いので、そういう方にとっては遠回りも近道かもしれません。特に高周波の回路設計の基礎としては必修と思います。

ESP32-WROOM-32 や、ESP32-WROVER-B では無線通信関連の回路がパッケージされているので、そのあたりのセンシティブで難易度の高い回路を気にすることなく開発できるのが嬉しいポイントです。

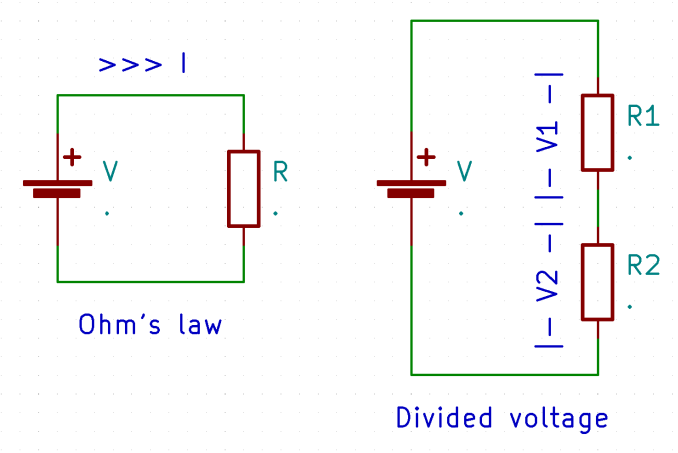
3.1.1　回路の公式

3.1.1.1　オームの法則、分圧の計算

回路図を眺める前に、オームの法則と合成抵抗の計算方法について複数しておきましょう。

オームの法則

以下に、オームの公式と対応する回路図のイメージを示します。



▲図 3.1: オームの法則

$$V = RI$$

V が電圧、R が抵抗、I が電流です。それぞれの単位は V（ボルト）、Ω（オーム）、A（アンペア）です。同じ抵抗であれば、電圧が高いほど、たくさんの電流が流れるということを意味しています。モーターなどのパワー系部品を扱わないデジタルメインの電子工作の場合、3〜12V 程度、0〜数百 k Ω、0〜数百 mA あたりの値を扱うことが多いと思います。

分圧の計算例

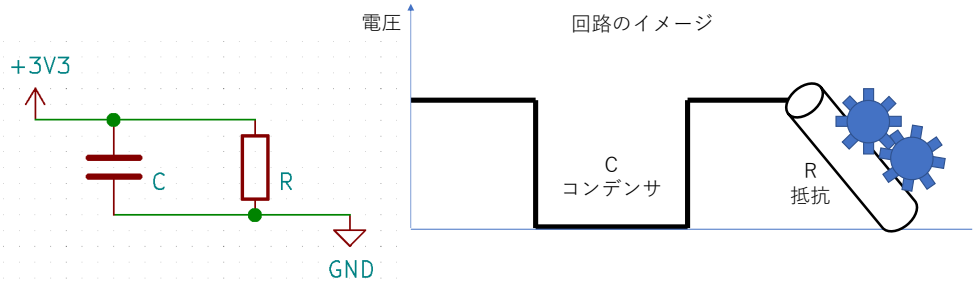
以下に、分圧の計算式を示します。図 3.1 の右側の図が対応します。

$$V2 = V\frac{R2}{R1 \cdot R2}$$

抵抗の組み合わせによって、0V〜電源電圧の範囲内で任意の電圧値を得られることが分かると思います。信号の取りうる電圧の範囲を狭めるためなどに用います。

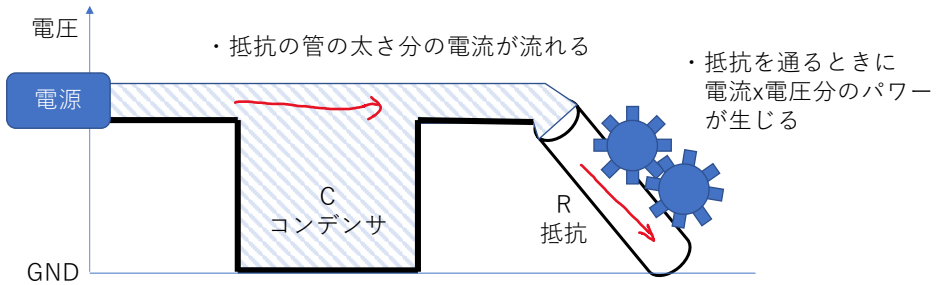
3.1.1.2　回路部品と電圧電圧の関係のイメージ

最初は、式だけ見てもいまいちピンとこないこともあると思います。そんなときは物理的な形あるものに置き換えてイメージするのが良いと思います。よくオームの法則を高いところから低いところへ流れる水で表した図を見かけます。水の流れの例をベースに、筆者が頭の中で描いている回路部品のイメージを図 3.2 と図 3.3 に示しました。



▲図 3.2: 回路と部品のイメージ

図 3.2 には、コンデンサと抵抗を並列に接続した回路図と、それに対応する水路のイメージ図を示しています。コンデンサは水桶で、抵抗は歯車の付いたパイプのイメージです。



▲図 3.3: 部品と電流の関係のイメージ

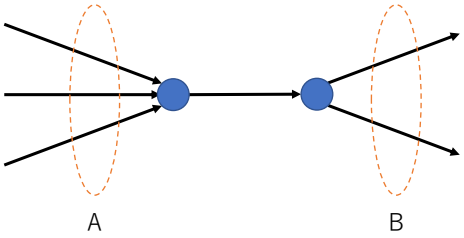
図 3.3 には、電源を接続して水流（電流）が流れているイメージを表しています。コンデンサは水桶なので、一度いっぱいになると溢れて、先の抵抗に水が流れるようになります。抵抗はパイプなので、一度に流せる水の量に制限があります。太ければ太いほど抵抗が低く、一度にたくさんの水を流せます。また電圧が高いほど勢いよく GND へ向けて水が流れ込みます。抵抗に付いている歯車は水流を受けて回り、その力で仕事をします。ただの抵抗器の場合、発生した力は熱に変換されます。

電流がたくさん流れている抵抗値を持つ部品は、電流 x 抵抗値の分の発熱をしていることは知っておきましょう。例えばパソコンの CPU 等もたくさんの電流を使って内部で仕事している抵抗器と見ることもできます。CPU チップという小さな部品の中でたくさん電流を使って激しく計算している分、発熱も激しくてヒートシンク等の冷却が必要になるわけです

電気部品にはそれぞれ耐久温度の上限があるので、それを超えないように設計しないといけません。もし超えると部品が壊れ、動かなくなるだけであれば良いですが、部品内部で短絡が生じて大量の電流が流れ込み、発火や爆発したりする、などの危険な状況にも繋がります。

3.1.1.3　キルヒホッフの法則

そしてもう一つ欠かせないのはキルヒホッフの法則です。図 3.4 に示すように、同じだけの電流が回路から出ていく、ということを示しています。



Aの 3 本の配線を流れる電流の総和と、
Bの 2 本の配線を流れる電流の総和は等しい

▲図 3.4: キルヒホッフの法則

ネット上に公開されている回路図を見てみて、どんな電圧や電流で動いているのか、頭を働かせてみると練習になると思います。

3.1.2 ブロック図から回路図へ

ブロック図を描く際に各ピンの機能を整理しました。その結果に従って部品間を接続していけばよいのですが、IC のピンの入出力をそのまま外部接続のコネクタに接続してよい、とは限りません。電圧の調整、電流の調整のために、抵抗やコンデンサを取り付ける必要があります。

3.1.3 電気回路で考慮するべき要素

回路図を作るにあたって考慮するべき観点を挙げてみます。

3.1.3.1 電源

電子回路は大まかに、各部品への電源供給のための回路と、電子部品間、外部接続端子間の通信のための回路の 2 系統に分けることができます。

電子工作で使用する電源は、数 V 程度の直流（DC）が一般的です。家庭用コンセントから取れる電源は 100V 50 or 60 Hz の交流（AC）なので、DC にするためには、AC/DC アダプタを使います。よくノートパソコンなどの電子危機の外側にある、あの黒い箱が AC と DC の変換を行っています。基本的に安全面価格面どちらの観点からも、AC/DC アダプタは自作せずにきちんとした品質のものを購入するのがよいと思います。また、DC5V であれば、USB 充電器の出力を利用することもできます。

デバイスの電源を考えるときには、以下のようなポイントについて整理する必要があります。

電圧

使用する部品によって、その部品を動作させるために必要な電圧や電流が決まっています。まずそれらを整理して、主電源から何種類の電圧を作る必要があるのか整理しましょう。またマイコンの電源の電圧がそのまま出力する信号の基準値になることが多いです。マイコンの仕様をよく確認しましょう。特に高い電圧の出力を、低い電圧で動いている電子部品の端子に繋ぐ場合は、要確認です。通常は電圧を同じレベルに合わせることが必要です。データシートに書かれた絶対定格値を超えてしまうと、部品が壊れてしまいます。

供給する電力

部品ごとに必要とする電力も決まっています。電源に関する部品を選定する際は、それを供

給できるだけの容量をもった部品を選択しましょう。電圧を固定で考えると、供給できる電力の大小は電源が供給できる電流の大きさの大小に比例することになります。マイコンを用いた電子工作であれば、通常の消費電流は 1A を超えることはそうないと思います。電流を大量に消費する回路はその分どこかで発熱しているので、消費電流が大きい場合は発熱についても気にするようにした方が良いでしょう。

Ground

すべての電源から流れた電流は最終的に Ground に戻ってきます。回路図を書く際は電流がどこからどこへ向かって流れているのかを意識しましょう。

3.1.3.2 GPIO

GPIO とは汎用の入出力のピンで、ソフトの設定で電圧の高低を 0、1 で設定できます。出力が HIGH のときに実際に出力される電圧は、その IC の電源電圧に依存することが多いです。入力として使う場合、ボタンや回路の状態から ON、OFF を検知したり、デジタル信号を受信するような用途で使われます。出力として使う場合、スイッチのように周辺回路のオンオフをしたり、高速に HIGH、LOW を制御してデジタル信号を生成したりできます。

3.1.3.3 デジタル信号

何らかのデジタル情報を外部に出力するような基板であれば、内部の IC 部品間の通信もデジタル信号で行う場合が多いでしょう。

デジタル信号の場合、高周波信号は特に取り扱いに注意が必要です。高周波帯では、単なる信号線も理想的な伝達経路とは見なすことはできなくなり、回路の形状、部品の配置等によって大きな影響を受けることになります。結果として単に部品を繋いだだけでは上手く動かない、といった問題が生じます。高周波信号を考慮した設計には専門的な知識、経験が必要になるので、本書では取り扱いません（筆者も経験がない）。ただ、趣味の工作をしていく上で高速な信号を取り扱う必要があった場合は、完成されたモジュールを購入して使う、などするのが良いと思います。

例えば数 Mbps を超えるような信号が流れている配線は無駄に長くしないように注意必要です。

UART

数 kbps〜数百 kbps の通信に用いられます。デバッグ用にターミナルを接続したり、ファームウェアの転送、AT コマンドや NMEA など文字列ベースの通信で制御するデバイス等で使用されます。

I2C

数百 kbps〜数 Mbps の通信。センサやキャラクタディスプレイなどの比較的低速な周辺デバイスとの接続にもちいられています。

SPI

数百 kbps〜数十 Mbps の通信。ESP32 ではフラッシュメモリや PSRAM の接続に、より高速なモードを用いています。

HDMI

数 Gbps〜の通信。高解像度なデジタル映像を通信するために非常に高速です。ここまで高速になると、素人が動作が安定した回路を設計するのは困難と思います。高速な信号を扱うことが難しい例として、短い HDMI ケーブルでは OK なのに、長い HDMI ケーブルでプロジェクターに PC を接続するとうまくうつらない、ということがたまにあります。これは信号が高速であるために、長いケーブルによる物理的な悪影響を受けてしまっているためと思

われます。

3.1.3.4 アナログ信号

アナログ信号とは、連続した電圧の高低に意味がある信号のことです。例えばアナログ音声信号があげられます。アナログ波形の電圧の高低をそのままスピーカーの振動の強弱に変換すれば、それは音声として認識できます。またセンサーの測定結果をアナログな電圧値で出力するような場合もあります。

フィルタ

音声や無線信号などの電圧波形について、抵抗、コンデンサ、コイルの組み合わせによってローパスフィルタやハイパスを実装することができます。本書の例では、アナログ音声出力信号について、ハイパスフィルタを通して、直流成分をカットしています。フーリエ変換の知識があればある程度イメージができるかと思います。知らない場合でもとりあえず飛ばしてもらって大丈夫です。

AD コンバータへの入力

IC がアナログ信号を処理するためには、アナログ信号をデジタル値に変換する必要があります。そのために用いるのが AD（Analog to Digital）コンバータです。ADC とも呼びます。ESP32 もいくつかのピンで ADC に対応しています。その逆として、デジタル値をアナログの電圧に変換する機能を、DAC と呼びます。

アンテナへの入出力

WiFi や BLE などで飛ばす電波の周波数は数 GHz 程もあります。ESP32-WROOM-32 や ESP32-WROVER-B などでは、アンテナを基板上の銅線のパターンとして実現しており、ESP32 とアンテナ間の接続はユーザが触ることを想定していません。

3.1.3.5 見えない抵抗値

回路図上で配線は抵抗 0 Ωの理想的な導体とみなされていますが、実際はそういうわけではありません。銅などの導体も電流が流れば電圧が微小ながら降下します。（更にはコンデンサやコイル相当の回路要素も含んでおり、高周波回路では無視できなくなります）そのため基板デザイン時は、不必要に配線を長くするのは避けるべきです。

3.1.4 ブロックとして回路を見る

回路図の全体図を見るととても複雑に見えますが、細かく分割していくと、ある程度パターン化されたブロックの組み合わせで成り立っていることが分かります。電気回路の教科書ではそういった点は教えてくれないので、筆者も最初は戸惑いましたが、色々作例を見ていくうちになんとなく分かってきました。

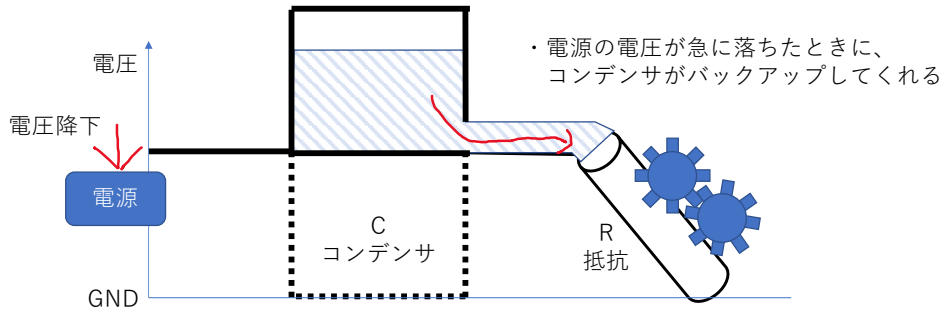
そういったパターンのいくつかを説明してみたいと思います。

3.1.4.1 バイパスコンデンサ

既存の回路図を見ていると、よく電源のラインがコンデンサを介して GND に繋がっていることがあります。これはバイパスコンデンサ（パスコン）と呼ばれていて、電圧の安定化やノイズの除去を目的として存在しています。

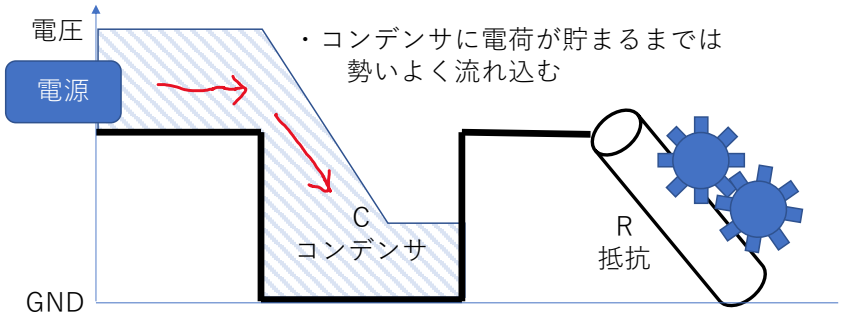
図 3.5 には、電源の電圧が低下したときのパスコンのイメージを図示しています。回路は図 3.2 の回路と同じものです。抵抗が電源を供給する先の IC 等の回路ブロックと見立てることができま

す。電源の電圧が低下したときに、パスコンは図にあるように、貯めた電荷を利用して電圧を一定に保とうとする一時的な給水源（バッテリー）となります。コンデンサの容量には限度があるので、すぐに枯渇しますが、電源が瞬間的に低下したようなケースで役に立ちます。電圧の急低下に即応するために、パスコンは電源供給先の回路ブロックのすぐそばに設置する必要があります。電子工作で使うような IC でのパスコンには、0.1〜数十μ F 程度のコンデンサを使うことが多いようです[*1]。



▲図 3.5: バイパスコンデンサのイメージ

バッテリーは大きければ大きいほどよいのであれば、大容量のパスコンをあちこちに設置したくなりますが、必ずしもそれが良いわけではありません。パスコンに電源を接続した瞬間のイメージ図を図 3.6 に示します。



▲図 3.6: 突入電流のイメージ

電源を接続した瞬間電源から見ると、パスコンは抵抗に水流が届く前にどばどば水を注げる桶のように見えます。そのためパスコンが充電できるまで、電源からは瞬間的に大きな電流が瞬間的に流れます。これを突入電流（Inrush Current）と呼びます。電源にも一度に流せる電流の上限があるので、過剰に容量の大きいパスコンを付けるのは避けるべきです。

またパスコンにはノイズを除去する効果もあります（ローパスフィルタ）。コンデンサの容量が小さい方が高い周波数のノイズを除去する効果があります。カットする周波数の帯域の違いを考慮

[*1] 数 F のとても容量の大きいコンデンサ（Supercapacitor）を搭載して、基板全体の電源喪失時のバックアップ電源として使用するようなこともあります

して、複数の容量のコンデンサを並列に IC の電源に接続したりします。

3.1.4.2　電源ブロック

回路には電源が必要不可欠です。電子機器は通常直流で動作するので、100V の交流（AC）を使用する場合は、直流（DC）を取り出すために AC/DC アダプタを使うことが多いです。各部品によって動作する電圧が異なることも多いので、基板上で入力された直流から必要な電圧を作り出します。その際は、LDO（Low Drop Out）レギュレータや DC/DC コンバータといった部品を用います。

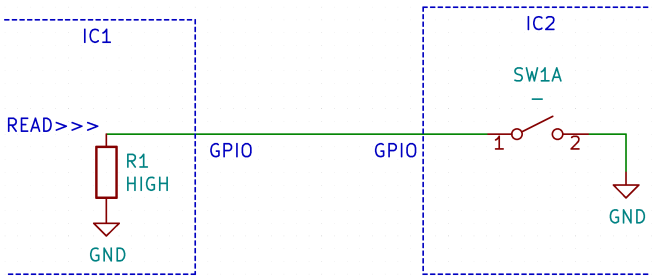
Narya board では LDO レギュレータを用いて、5V から 3.3V を作ります。

3.1.4.3　信号線の接続とプルアップ、プルダウン

電子工作をしていると GPIO から出力されている信号線についてプルアップ、プルダウンするという話を見かけるかと思います。GPIO が制御されていない場合のデフォルトの電圧値を決める意味があります。開放された入力端子は電気的に不安定なので、プルアップまたはプルダウンすることで電気的に安定させる意味もあります。

図 3.7 には、プルアップされていない場合の回路の例を示しています。IC1 が IC2 の GPIO の状態を READ のポイントで読んでいるような状態です。IC1 の R1：HIGH というのは、IC1 の端子がハイインピーダンスであることを示しています。ハイインピーダンスというのは、言い換えると、とても抵抗値が高く実質絶縁状態であることを示しています。このとき、IC2 のスイッチが ON であれば、READ のポイントの電圧は GND になります。

しかし IC2 のスイッチが OFF の場合は、READ の電圧値が定まらずとても不安定になります。

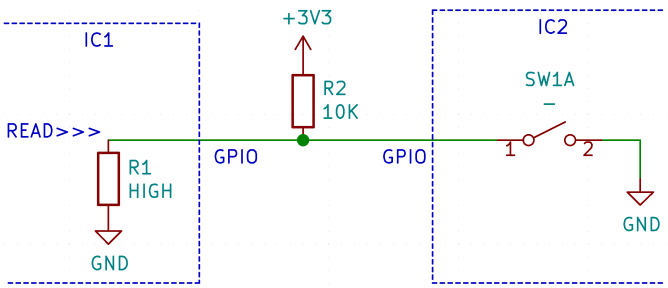


▲図 3.7: プルアップされていない場合の回路

一方、図 3.8 には、プルアップした場合の回路の例を示しています。このとき、IC2 のスイッチが ON であれば、READ のポイントの電圧は同様に GND になります。プルアップしていない場合との違いとして、R2 経由で電流（3.3/10,000=0.33mA）が IC2 に流れ込みます。

IC2 のスイッチが OFF の場合は、READ の電圧値は 3.3V になります。R1 を絶縁ではなく、非常に抵抗値の大きい抵抗（R1>>R2）として分圧を考えると理解しやすいと思います。

$$V(read) = 3.3\frac{R1}{R1 + R2}$$



▲図 3.8: プルアップされた合の回路

IC に流れる電流とノイズへの耐性の兼ね合いを見て、プルアップ、プルダウンの抵抗は 10K〜50K Ωの間くらいで選ばれることが多いようです。

3.1.4.4 USB-シリアル変換

ESP32 はソフトの書き込みに UART を使用しています。PC は UART で直接通信ができないので、USB シリアル変換のマイコンを使って、USB で PC と ESP32 を接続します。

3.1.4.5 音声出力

ESP32 から出力される音声信号の電圧は、0〜3.3V の範囲です。音声の信号は 0V を基準とした正負の電圧の波形なので、波形の基準を 0V にシフトさせないといけません。そのため、ハイパスフィルタ（直列接続のコンデンサ）を通して直流成分を除去しています。

3.1.4.6 ESP32-WROVER-B のデータシートを読む 2

データシートにはその部品の回路設計に必要な基本的な情報が書かれています。回路のブロックについての説明を踏まえて、ESP32-WROVER-B のデータシートをもう一度読み返してみましょう。

電源の回路

ESP32-WROVER-B に電源を供給するための回路について、データシート 7 章の"Peripheral Schematics"を参照して確認してみます。"Figure 4: Peripheral Schematics of ESP32-WROVER-B"には、ESP32-WROVER-B の回路例が載っています。3V3 に接続されている箇所が電源供給部の回路です。リセットの機能を持つ EN という端子に、抵抗＆コンデンサが繋がっている箇所は、EN のためのプルアップとリセット解除遅延の機能を持っています。こちらは本体への電源供給とは直接関係ないです。電源としてのポイントは、C1 と C2 というコンデンサが 2 つ接続されている点です。これらが前述したバイパスコンデンサです。

起動時の仕様

ESP32 は起動時の所定のピンの電圧によって、起動後の挙動が変わります。その仕様について、"2.3 Strapping Pins"を参照して確認してみましょう。Table4 に各ピンの初期電圧値と起動後の挙動について説明があります。特に影響が大きいのは、Voltage of Internal LDO (VDD_SDIO) と Booting Mode かと思います。Voltage of Internal LDO については、ESP32-WROVER-B の内部で使われているフラッシュメモリへの電源供給電圧の設定です。デフォルトでチップ内部でのプルダウンとなっています。注釈によればフラッシュメ

モリへの供給電圧は 3.3V でなくてはいけないので、起動中にプルアップしてはいけないことが分かります。。ただし ESP32 をパッケージした他のシリーズでは、1.8V であるものもあります。データシート 6 章の"Schematics"の、MTDI に対応する GPIO12 の回路を見ると、R9 が付いており、一見プルアップされているように見えますが、NC となっています。NC は Non Connection の略で、その部品は実際は搭載されないことを示しています。SD カードや IO として使用するときに、プルアップしようとするとフラッシュメモリにアクセスできず、全く起動しないという状況になるので要注意です[*2]。

Booting Mode については、GPIO0 によって、フラッシュメモリからのブートなのか、ソフト書き込みモードで起動するかが切り替わることが分かります。通常はプルアップで良いですが、ソフトを書き込み時だけプルダウンします。

3.2 回路設計ツールでの回路設計

では、回路図の書くために必要な要素の知識を確認したところで、実際に回路を設計してみましょう。

でも、そもそも回路の設計とは何でしょうか？

筆者の理解では、回路とは、電気的な入力（電源、デジタル信号、アナログ信号等）に処理を加えて、出力を得るための部品と配線の集合です。回路設計は回路において、期待する結果の出力を得るために、部品をどう繋ぐのかを決める作業です。要は点と線を繋ぐ作業の繰り返しなので、あまり難しく考えずにトライしてみましょう。

これから使っていくことなる回路設計ツールは、その点と線を繋ぐ作業をアシストして、電子データとして表現できるようにしてくれます。

3.2.1 回路設計ツールの選択

基板の実装をブレッドボードやユニバーサル基板で行う場合は、回路図は紙とペンで行うのが一番簡単かもしれませんが、基板を製造するためには最終的には所定のデータフォーマットでの出力が必要です。そのために PC ツールを使います。

業務用のツールには色々あるようですが、個人的な開発で、回路の設計に無償で使えるツールとしては、Eagle、KiCad が挙げられることが多いです。筆者が認識している違いを上げてみます。

- Eagle
  - 歴史が長く、部品ライブラリ等が充実。無償の場合は設計できる基板規模に制限あり。
- KiCad
  - 比較的最近オープンソースで開発されているもの。設計できる基板規模に制限はない。CERN が支援しており、最近利用者が増えている。部品ライブラリはまだ多くはない？

筆者は OSS という点と、制限なく利用できる点に惹かれたので、KiCad を使っています。開発環境としては、Windows 以外にも Mac、Linux もサポートしています。

[*2] 筆者は一度ハマりました・・・。ドキュメントを面倒臭がらずにちゃんと読むことの重要性を学習する良い経験でした

3.2.2 KiCad を使う

では KiCad を実際に使ってみましょう。基本的な使い方を説明していきます。本書だけで使い方のすべてを説明することはできませんが、使い方の概要を簡潔にまとめつつ、筆者が使っていて躓いた点についてまとめています。

3.2.2.1 KiCad でできること

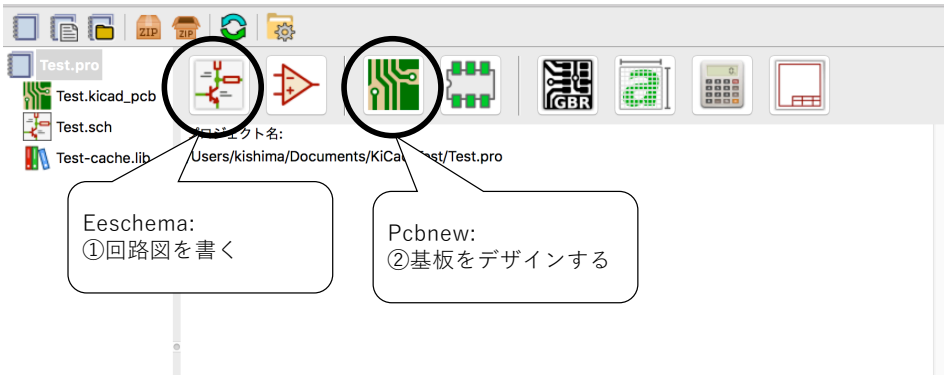
KiCad では、以下のような機能を持っています。

- 回路設計
- PCB レイアウト

それぞれが別のアプリを用いて編集を行い、最終的にはガーバー (Gerber) データという、基板や配線の形状を表す標準的なフォーマットのファイルを出力することになります。基板製造を請け負う会社にこのデータを渡すことで、基板を製造してくれます。
本章では回路設計の部分について説明しています。

3.2.2.2 KiCad のインストール

以下の公式サイトから環境に合わせてダウンロードして、インストールします。インストールに特別なソフトは不要と思います。
`http://www.kicad-pcb.org/download/`
筆者の環境では、MacOS 版を用いています。
インストールして「KiCad」を起動すると、図 3.9 のようなランチャー画面が表示されます（この例ではすでにプロジェクトを読み込んだ状態です)。

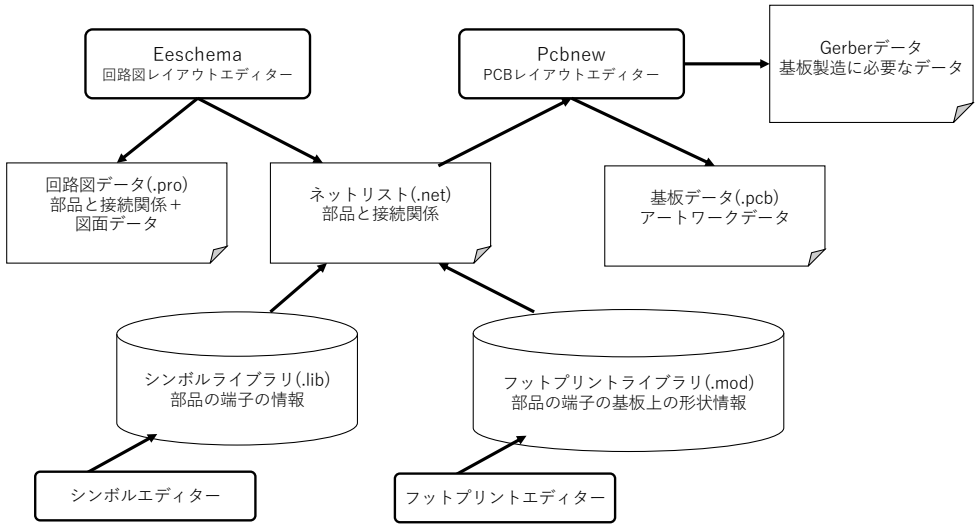


▲図 3.9: KiCad 起動時のメニュー

回路設計は 1 のアイコンの Eeschema というアプリをで行います。回路設計ができたところで、2 のアイコンの Pcbnew というアプリで基板データを仕上げて行きます。

3.2.3 KiCad を使った回路設計の流れ

KiCad を使った設計の流れと参照する情報を図 3.10 にまとめてみました。



▲図 3.10: KiCad を使った開発の流れ

まずは、Eeschema で回路図を作成して、ネットリストファイルを出力するのが目標となります。回路図上に設置する部品は、シンボルライブラリから読み込みます。電源や抵抗やコンデンサなどの基本的な部品および、一般的なスイッチ、IC 等の部品はデフォルトのシンボルライブラリにあらかじめ登録されています。もしシンボルライブラリに存在しない部品でもあっても自分で作成することができます。

回路図よって、対象の基板にどんな部品がどのようなトポロジーで接続されているのか[*3]を表現した結果が、ネットリストファイルに格納されて、Pcbnew に渡されることになります。

3.2.4 KiCad の基本的な使い方

では、KiCad を実際に使っていきましょう。本書では KiCad の使い方のイメージをまず掴んでもらえるように最小限の例を紹介したいと思います。その他の基本的な使い方については、公式サイトで公開されている「KiCad ことはじめ」[*4]を一読するのが良いと思います。

3.2.4.1 プロジェクトを作る

まず基板開発のプロジェクトを [ファイル][新規][プロジェクト] で作成します。試しに Test というプロジェクトを作成してみると、図 3.9 のような構造が表示されると思います。(Test-cache.lib はプロジェクト作成直後は表示されないです)

```
Test.pro
  |- Test.kicad_pcb
  |- Test.sch
```

プロジェクト名.kicad_pcb が基板のレイアウト情報で、Pcbnew で編集します。プロジェクト

[*3] 言い換えると、ネットリストファイルに格納されるのは、回路図図面の情報まるごとではなくて、部品間の接続関係の情報ということになります。Eeschema 図面上の部品の形や位置は論理的なのもので、実際の基板上の物理的配置とは無関係であることに注意しましょう

[*4] http://docs.kicad-pcb.org/5.1.2/ja/

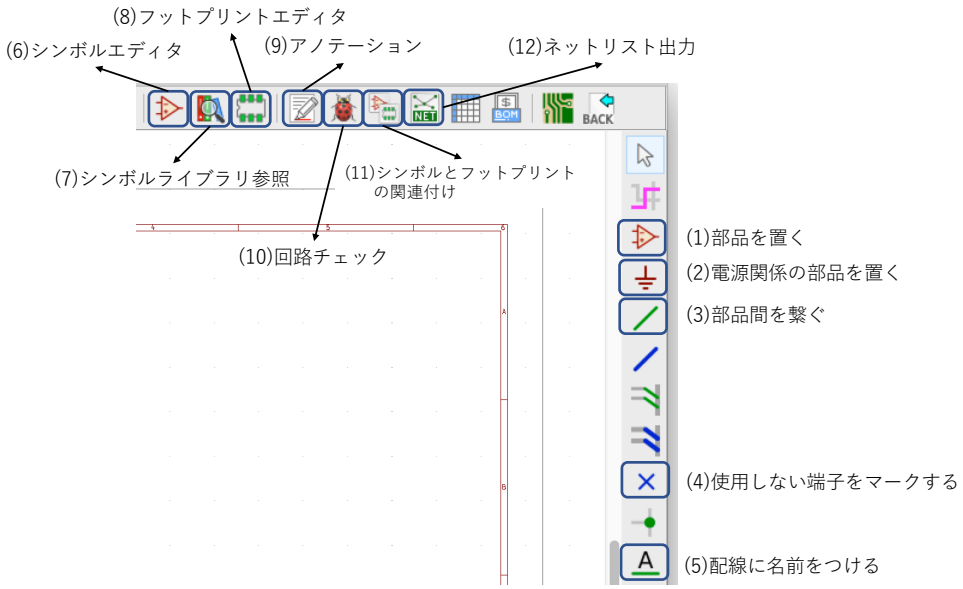
名.sch が回路図情報で、Eeschema でこれから編集します。

では、Test.sch をダブルクリックするか、Eeschema のアイコンをクリックして、回路図を編集してみましょう。

3.2.4.2 Eeschema の作業の流れ

Eeschema を起動すると、回路図の図面のシートとツールバーが表示されます。画面に表示されている枠が、回路図の図面の範囲です。実際に見たままの形で印刷することも可能です。この枠の中に部品を置いて、部品間を配線していきます。

図 3.11 には、よく使う機能のボタンとその意味を示しています。



▲図 3.11: Eeschema のよく使う機能

基本的には、以下のような流れで作業していきます。

1. 電源関係の部品と、一般部品を置く (1)(2)
2. 部品間を接続する (3)
3. 部品に通し番号を付ける (9)(アノテーション)
4. 回路チェックする (10)
5. 部品のシンボルとフットプリントの関連付けをする (11)
6. ネットリストを出力する (12)

3.2.4.3 部品を置く

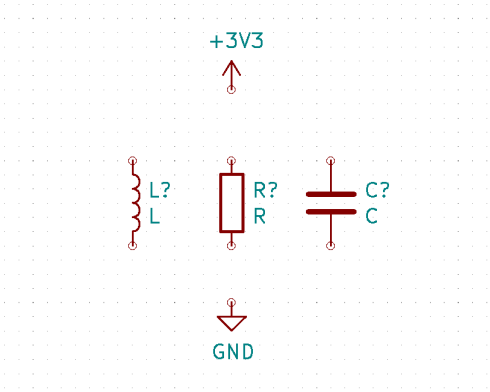
では、部品を置いてみましょう。(1) のボタンをクリックすると、シンボルを選択する画面がでます。シンボルというのが、抵抗などの電気部品のことを指しています。通常だとメニューには大量の部品が並んでいますが、この中から自分が置きたい部品を選択していきます。フィルターの欄に部品の名前を入れて絞り込むのが早いですが、最初はどんなキーワードを入れると良いのか分か

りにくいと思います。以下に代表的な汎用部品の絞り込み方を挙げてみます。

- 抵抗：r
- コンデンサ：c
- コイル：l
- ダイオード：diode
- LED：led
- スイッチ：button
  - "SW_Push"が一番シンプルなスイッチだと思います。
- スルーホールのコネクタ：conn
  - "Conn_01x03_Female"といった名前で見つかります。ホールの間隔等はフットプリントとの関連付けの段階で確定します。

ブレッドボードで使用できるような、2.54mm ピッチのピンで接続する部品などは、シンボルライブラリ上に存在していなくても、回路シミュレーションなどを行わず、基板を作るだけであればConn_** のコネクタで回路図上で表現しておけば OK なので気楽です。

図 3.12 には、電源、GND、抵抗、コンデンサ、コイルの部品を配置した状態を例に挙げています。



▲図 3.12: 部品を回路図上に配置した状態

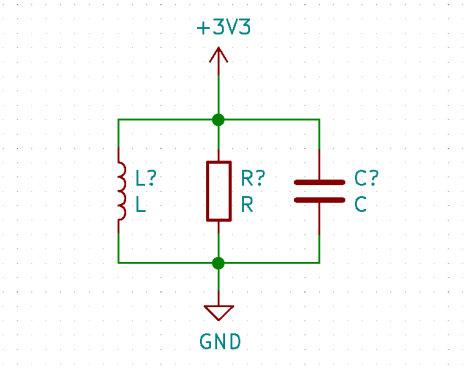
電源の部品は、図 3.11(2) のボタンを押して選択することができます。使用する電圧に合った部品を選択しましょう。(3V3 は、3.3V の意味です)

IC の中には電源としての機能を持ったピンを備えた IC があり、それを電源として扱うこともできます。

GND にも複数種類がありますが、よく分からなければ、とりあえず"GND"で見つかるものを選んでおけば大丈夫です。

3.2.4.4 部品を接続する

回路図上に配置した部品を接続してみましょう。配線には、図 3.11(3) の緑色の線のボタンを使用します。各部品の○で示された端子間を接続していきます。図 3.13 には部品を接続した状態を示しています。配線は互いに交差させることもできます。交差ではなく接続させる場合は、配線上を端点として検索すると、接続している部分が「●」で表示され区別されます。

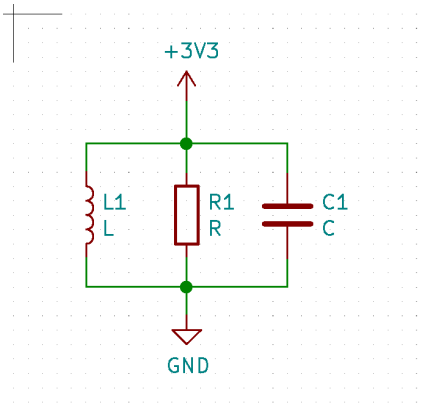


▲図 3.13: 部品間を接続した状態

この時点ではアプリケーション上各部品を一意に識別できない状態なので、各部品に互いに識別可能な名前を付ける作業を行います。これは図 3.11(9) のボタンで行います。とりあえずは特に設定は何もいじらず、右下の「アノテーション」ボタンをクリックすると、以下のようなログが表示されて、各部品に名前が付けられたことが分かります。

```
+3V3 をアノテート #PWR01 として
GND をアノテート #PWR02 として
C をアノテート C1 として
L をアノテート L1 として
R をアノテート R1 として
アノテーション完了。
```

以降の作業はこのアノテートされた名前が使用されるので、新しく部品を追加した場合は、毎回アノテーションを行う必要があります。図 3.14 には、アノテーション後の回路図の状態を示しています。今まで「？」だった場所に番号が振られていることが分かります。



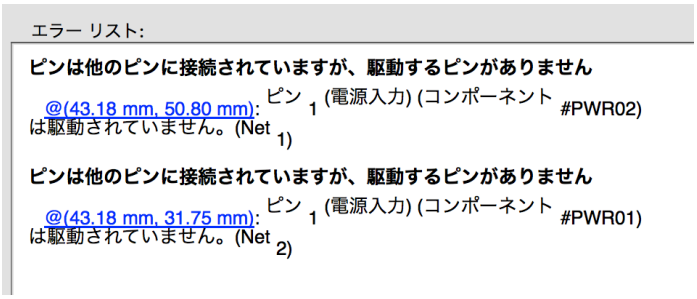
▲図 3.14: アノテーションを行った状態

3.2.4.5 検証してみる

これで回路図として体裁は整ったように見えますが、先に進む前に電気的な接続の確認（ERC）を行います。確認には、図 3.11(10) のてんとう虫マークをクリックして行います。ここでは電源

や GND が正しく部品に接続されているか、未処理の端子がないかどうか最低限のポイントを確認します。IC 間の論理的な接続関係が仕様通りか、アナログ回路的に問題がないかどうかなど、高度な回路シミュレーションのようなことを行う訳ではないことに注意してください。

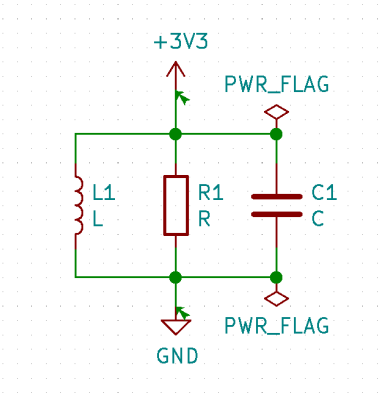
そして「実行する」をクリックするとチェックが行われます。図 3.14 の状態の回路図でチェックを行うと、図 3.15 のようなエラーメッセージが表示されると思います。



▲図 3.15: ERC のエラーメッセージ

これは、チェックを行う上で、回路上の電源と GND のピンを明示する必要があるため表示されているエラーです。このエラーを解除するためには、図 3.11(2) のボタンをクリックして選択できる、PWR_FLAG という部品を電源が供給されている配線と、GND が接続されている配線に接続します。

接続したあとの状態が、図 3.16 です。



▲図 3.16: PWR_FLAG を付けた状態

このような状態でもう一度 ERC を行うと、エラーが表示されなくなります。エラーが表示されなくなったら、次のステップに進みましょう。

3.2.4.6 フットプリントとの関連付け

Eeschema で行う作業も残り僅かとなりました。次は実際の物理的な基板の設計に必要な情報との紐付け作業です。

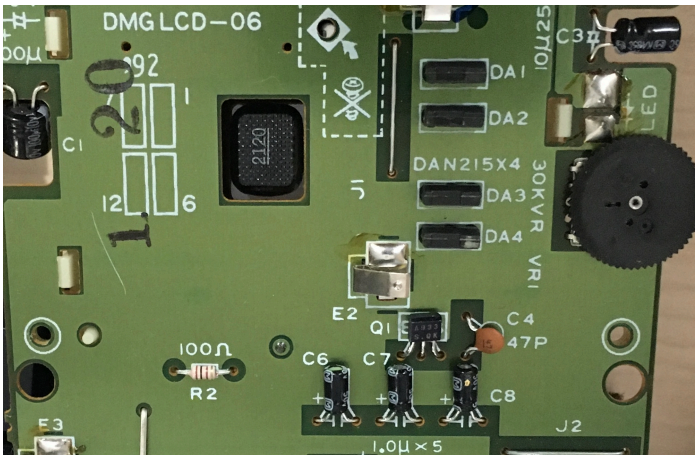
部品を基板に取り付けるためには、大まかに 2 つの手段があります。

- スルーホール実装 (THT:Through-Hole Technology)

- 表面実装 (SMT:Suface Mount Technology)

スルーホールは基板に穴が空いていて、そこに部品の端子の足を差してはんだ付けするものです。人の手ではんだ付けするにはこちらの方がやりやすいですので、電子工作ではこちらのほうがメジャーと思います。

一例として初代ゲームボーイを分解したときの写真を図 3.17 に載せてみます。昔は人の手で部品をはんだ付けしていたであろう形跡が見て取れます。

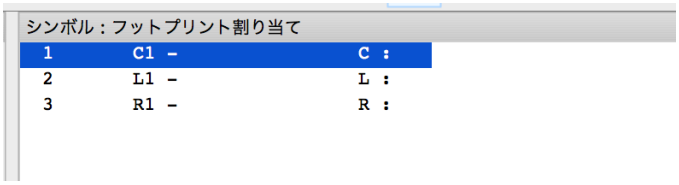


▲図 3.17: THT の例（初代ゲームボーイ）

表面実装は基板には穴が空いておらず、基板上のむき出しの金属面の端子部に部品をはんだ付けすることで、部品を固定しつつ配線に接続するものです。パソコンのマザーボードなどでイメージするような近年の工業製品の電子回路基板はこちらがほとんどと思います。部品も小さくすることができ、機械で自動的にはんだ付けもできるので、大量生産に向いています。

同じ部品でも THT か、SMT かで基板上での取り付けの端子の形状（フットプリント）が大きく異なります。そこを踏まえて、どのような部品を利用するのか意識しながら部品とフットプリントの対応付けを行います。

フットプリントの対応付けは、図 3.11(11) のボタンをクリックして行います。ボタンをクリックすると、図 3.18 のような表が表示されます。こちらは、各部品とそれに対応するフットプリントの名称を表示しています。最初はフットプリントは何も割り当てされていないので、回路図上の部品名だけ表示されています。



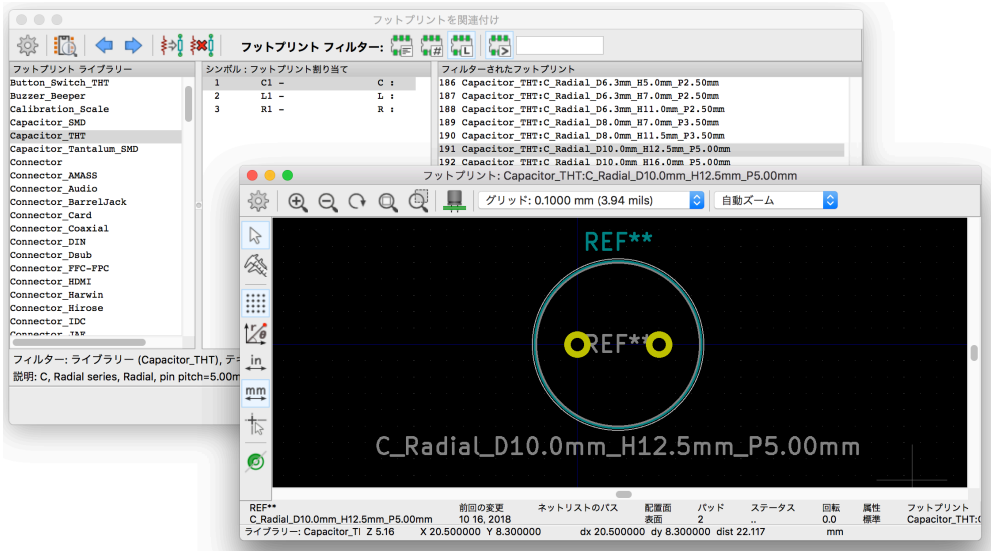
▲図 3.18: フットプリントの割り当て

次にウィンドウの左の方のフットプリントライブラリーから、部品に対応するフットプリントを見つけて選択していきます。例えば、コンデンサであれば、Capacitor_X というライブラリーを

選択します。

ここで先程触れた、THT と SMT の違いが現れてきます。ライブラリーには Capacitor_THT、Capacitor_SMD という二種類が存在することが分かります。THT はスルーホール実装のフットプリントで、SMD は Surface Mount Device の略で、表面実装部品のフットプリントであることを示しています。

実際に THT を選択して、「フィルターされたフットプリント」に表示されるフットプリントから適当なものを一つ選択して右クリックで「フットプリントを表示」を選択した結果が、図 3.19 です。



▲図 3.19: フットプリントの参照

実際の部品形状と、端子のスルーホールの穴が表示されていることが分かります。

自分が実際に使用する部品の物理的形状に合わせてフットプリントを選択します。

3.2.4.7 ネットリストの出力

全部品のフットプリントの対応付けまで完了すると、ネットリストの出力が可能になります。図 3.11(12) のボタンをクリックしてメニューを開き、「ネットリストを生成」ボタンで出力を行います。パラメータは特に触る必要ないです。

ここまで行うことで、Pcbnew による基板デザインに進むことができます。

もし、回路に変更が生じた場合は、ここまでの手順を繰り返して、再度ネットリストを出力し直すことなります。その際、アノテーションで指定した部品名がずれてしまうと、Pcbnew と Eeschema 間のトレーサビリティがなくなってしまうので、一度アノテーションした部品の名前を修正する場合は気を付けましょう。

3.3 本格的に回路設計

では、本格的に前章で作った Narya ボードのブロック図を回路図に落とし込んで行きましょう。まずすべての中心である ESP32-WROVER-B を回路図においてみよう！ と思ってシンボルライ

ブラリを覗いてみると、困ったことに部品がライブラリに登録されていません。
どうしたものでしょうか？

3.3.1 部品の追加

KiCad の標準シンボルライブラリに登録されていない部品については、自分でデータシートを参照しながら、部品を追加することができます。入出力端子が多い IC 部品ではそれも一苦労です。

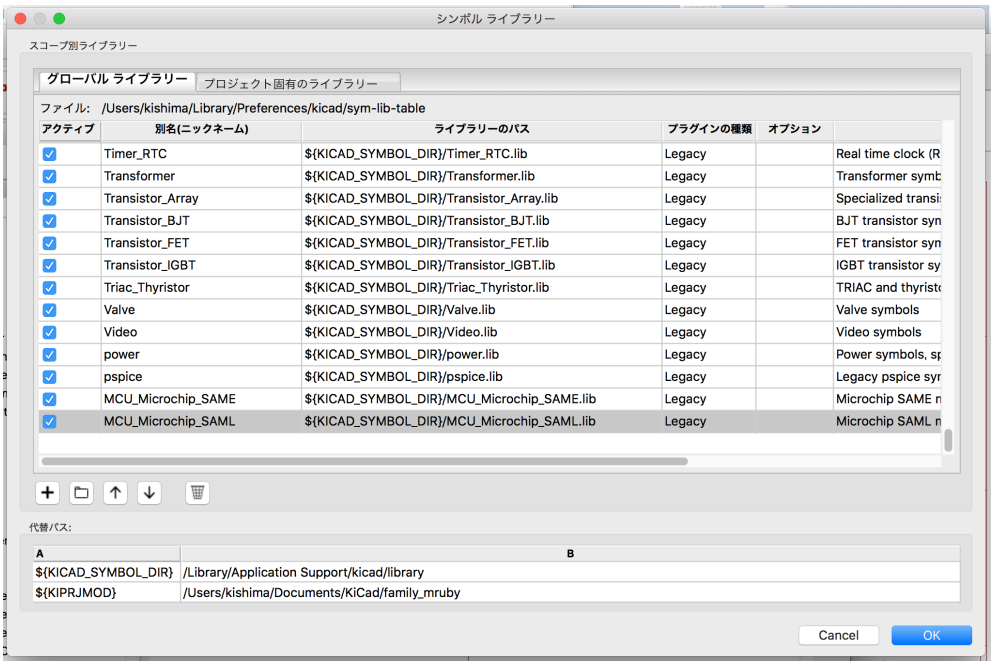
そこで無償で公開されている部品ライブラリデータを利用したいと思います。今回は、snapEDA を利用しました。

`https://www.snapeda.com/`

データは" Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA) "のライセンスの元配布されています。詳しくは以下を参照ください。

`https://www.snapeda.com/about/FAQ/#license`

取得したシンボルデータをシンボルライブラリに登録します。メニューの [設定][シンボルライブラリーを管理] からシンボルを追加できます。図 3.20 にはシンボルライブラリーの登録画面を示しています。こちらのフォルダマークをクリックして、ダウンロードした*.lib ファイルをグローバルライブラリーに登録すると、回路図上で部品として選択できるようになります。



▲図 3.20: シンボルライブラリの編集

シンボルと同様にフットプリントについても、フットプリントエディタを起動して、[設定][フットプリントライブラリを管理] から、ダウンロードした*.kicad_mod ファイルを読み込むことで、フットプリントとして選択できるようになります。

シンボルとフットプリントについて、部品のデータシートがあれば自作することができます。本書では紙面の都合で作り方は解説できませんが、シンボルエディタまたはフットプリントエディタ上で、自分が使いたい部品に似た既存部品のデータを見てみると、どのように作ればいいのか分か

ると思います。

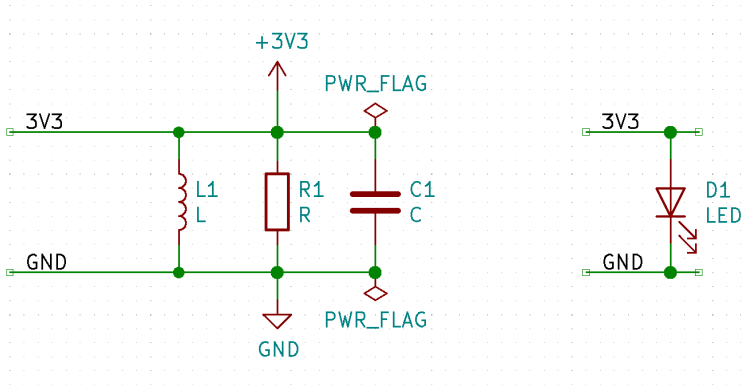
3.3.2 回路の描画

ここからは回路図を実際に書いていきます。回路図を書く際には、全体をいっぺんに書くのではなく、機能や部品のブロックごとに書いていくと頭が混乱せずにすむと思います。以降ブロックごとに Narya ボードの主な回路を説明していきます。

3.3.2.1 ブロック分割

まず回路をブロックに分けるための方法について説明します。KiCad では配線にラベルを付けることができます。同じラベルの付いた配線は、回路図上、直接接続されていなくても接続されていると見なされます。

図 3.21 には、図 3.16 の回路をベースにして、配線にラベルを付けることで回路のブロックを分けた例を示しています。



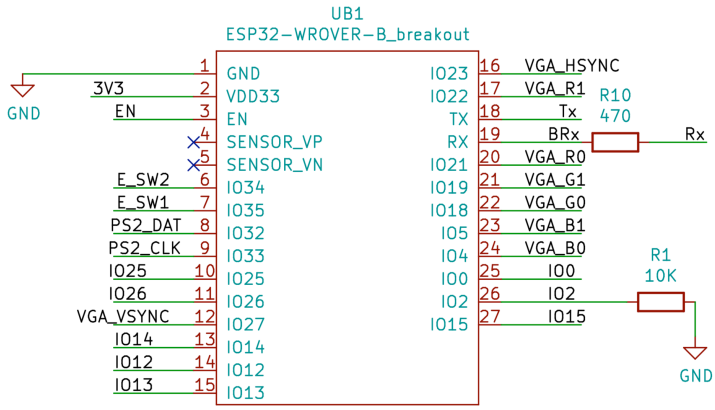
▲図 3.21: 配線にラベルを付ける例

この例では、3.3V の電源のラインに、「3V3」という名前を、GND のラインに「GND」というラベルを付けています。そして LED を追加し、「3V3」と「GND」を接続しています。このようにすることで、回路図上で配線を直結しなくても、ラベルを介して配線を自由に配置できるようになります。

この機能を利用して、回路の機能ブロックごとに回路をまとめると全体を把握しやすくなります。

3.3.2.2 主要部品の設置

まず回路の主要な部品を配置します。Narya Board では ESP32-WROVER-B がメインの部品となります。図 3.22 には、回路図上に ESP32-WROVER-B を配置して、前章で決めたピン配置に合わせて各ピンの配線にラベルを付けた状態を示しています。



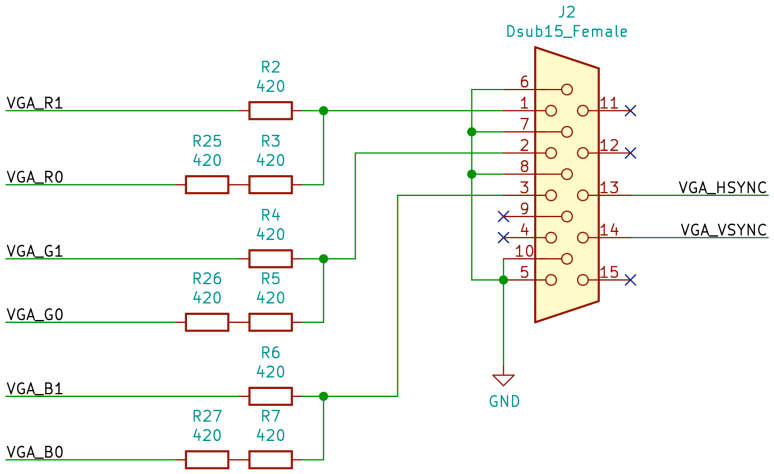
▲図 3.22: メイン部品

UART の Rx に接続されている R10 は類似の開発ボードで配置されていたため、まねしています。ESP32 の HW デザインガイドラインでは、Tx に 499 Ωの抵抗を接続することを要求しているので、それの対称の意味かと想像していますが、筆者も正確に理解できていないです。

3.3.2.3 外部との入出力部の設置

VGA 出力

図 3.23 には、FabGL のメイン機能である VGA 出力部の端子の回路を示しています。ESP32 側の配線については、FabGL の案内に従って配置しています。D-Sub15 ピンの各ピンの役割については、ネットで検索して確認しています。6、7、8 番は RGB 信号の帰還端子で、回路図のように GND を接続する必要があります（最初の試作で接続を忘れました・・・）。



▲図 3.23: VGA 出力（D-Sub15 ピン端子）

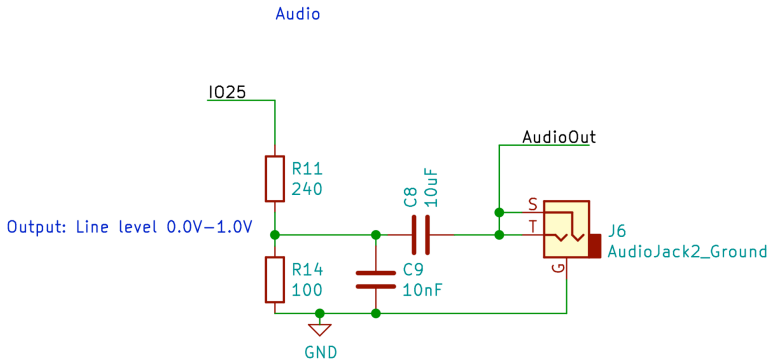
オーディオ出力

図 3.24 にはオーディオ出力の回路を示しています。こちらも FabGL の案内に従って回路を

作っています。出力の端子として、イヤホンジャックの端子を用いることにしています。ここからアンプ付きのスピーカーの LINE IN に接続することを想定しています。詳しい説明は少し専門的になってしまいますが、わからないときはサンプル通りに組んでみる、でも OK です。（ただ電源など電流がたくさん流れる回路は危ないので、ちゃんと原理を理解した上で設計するべきと思います）

IO25 の DAC から 0V〜3.3V の音声の波形が出力します。R11 と R14 で IO25 の出力電圧を分圧して、1V 程度にしています。Wikipedia によれば[*5]、オーディオの LINE レベルは Peek to Peek で 0.894V のようなので、これに合わせていると思われます。

そのままでは波形の範囲が 0V〜1V なので、これをハイパスフィルタ[*6]を通すことで、-0.5V〜+0.5V の範囲にします。LINE に接続されている先のオーディオ回路は 600 Ωと仮定すると、カットオフ周波数が 26Hz となって、だいたい人間の可聴域の下限にマッチします。C8 を直列に繋いでいる時点で直流の電流がそのまま先には流れない、ということはイメージができるかと思います。電流ではなくて、電圧の高低のみが伝わるとイメージしています。C9 はノイズ除去のためにあると理解しています。FabGL のサンプルでは 100nF となっていますが、これだと可聴域の波形もカットしてしまいそうだったので、10nF に変更しています。

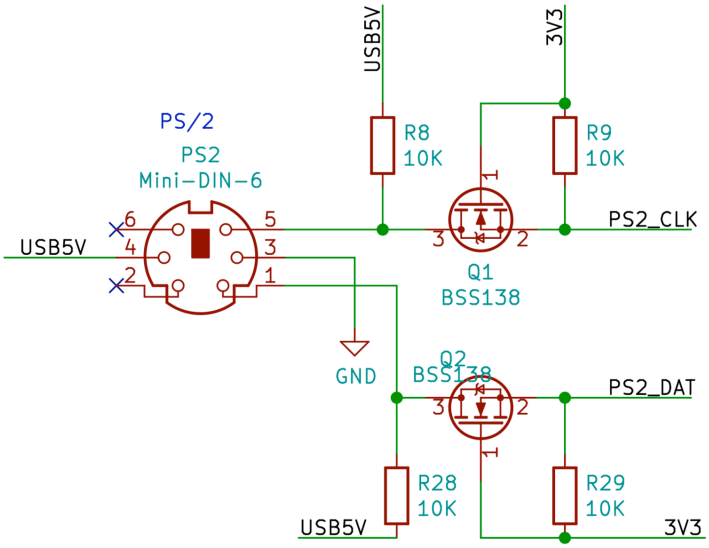


▲図 3.24: オーディオ出力（ステレオ AUX 端子）

PS/２キーボード

図 3.25 には PS/2 キーボードの接続のためのミニ DIN9 ピンの端子の接続部を示しています。ESP32 が 3.3V で動作するのに対して、PS/2 デバイスは 5V で動作するので、信号の電圧レベル変換が必要です。FabGL の案内では、抵抗だけで簡易的に変換を行っています。そのままでも動作するのですが、少し気になったので、MOSFET を使用したレベル変換回路を導入しています。仕組みは本書では詳しく説明できませんが、「MOSFET レベル変換」といったキーワードで検索すると解説しているサイトが見つかると思います。PS/2 程度の通信速度であれば、MOSFET の応答速度で十分対応できると思います。

[*5] LINE level： https://en.wikipedia.org/wiki/Line_level
[*6] 回路によるフィルタ： https://ja.wikipedia.org/wiki/フィルタ回路

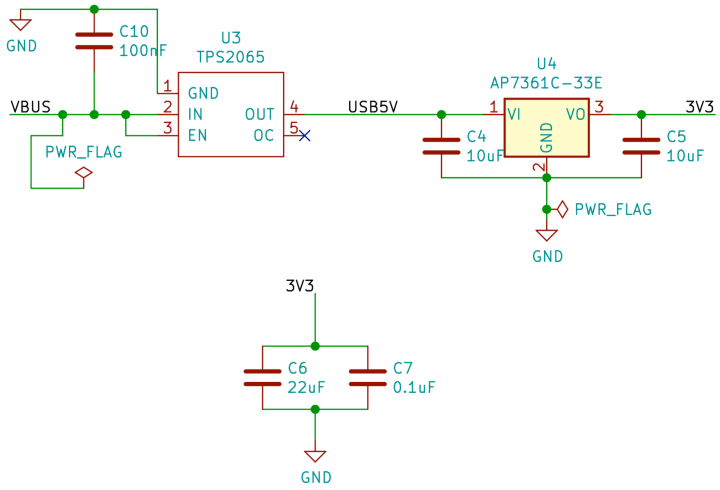


▲図 3.25: PS/2 キーボード（ミニ DIN9 ピン端子）

3.3.2.4 電源回路の配線

電源の回路について、図 3.26 に示しています。外部入力は USB なので 5V なので、ここから 3.3V を作るために、LDO（AP7361）を利用しています。出力電圧固定の LDO は入力と出力と GND の 3 端子で動作するので、部品の入出力の意味が理解しやすいです。ESP32 を利用する場合、データシートによれば電源に 500mA は最低限必要なので、その倍の 1A 程度は安定して出力できる部品を使う必要があるようです。mgo-tec さんの解析記事[*7]を読ませて頂いた結果、USB から電源を取るときには、電流制限をした方が良さそうだったので、Switch Scienece の開発ボードを参考に、TPS2065 を利用しています。この部品を使うことで突入電流を抑えて、大電流による瞬間的な電圧変動を抑制することができます。周辺に接続しているコンデンサの容量はそれぞれのデータシートを参考に設定しています。

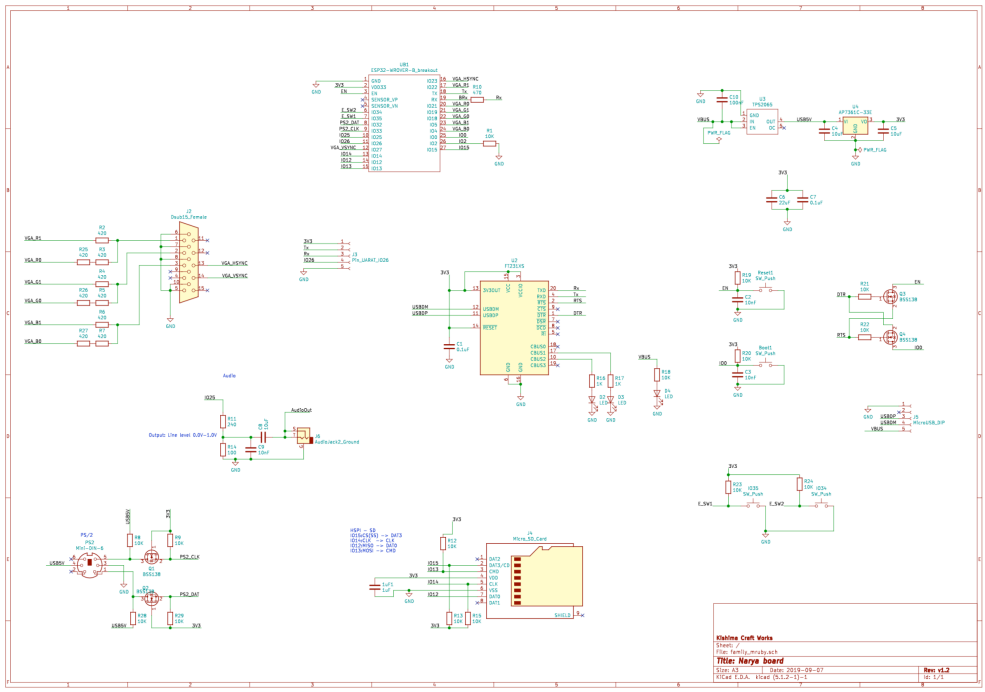
[*7] mgo-tec 電子工作：https://www.mgo-tec.com/blog-entry-esp-wroom-32-usb-inrush-current.html



▲図 3.26: 電源回路

3.3.2.5 回路全体図

各機能ごとに回路を組んでいき、ESP32-WROVER-B のすべての端子を処理が完了すると、全体の回路が完成です。図 3.27 に全体の回路図を示します。各機能ブロックにテキストで名前を付けたりすると、後で見返したときに分かりやすいと思います。



▲図 3.27: 回路図全体

3.3.2.6 基板設計に入る前に

基板設計に入る前に、制作経験のない回路については、機能ブロック単位で期待した動作をするかどうかブレッドボード上で確認しておくことをおすすめします。

3.4 部品の選定と調達

自分用に作るだけであれば、ジャンク品でも何でも最低限の数の部品を調達すればよいだけですが、ある程度まとまった数を作るのであれば、確実に入手ができそうなお店で買えるものを使って構成したいところです。

3.4.1 国内通販ストア

使ってみたことがあるお店を挙げています。

- 秋月電子
  - 色々安い掘り出し物がありますが、商品の入れ替わりもあるので、マイナーな部品は在庫が補充されないこともあるかもしれません。
- 千石電商
  - 各種表面実装のチップ抵抗なども小分けで購入できることを最近知って、助かっています。
- Switch Science
  - 部品よりもボードとかキットのほうがメインかと思います。ちょっと高めですが、即日発送してくれるのはありがたいです。
- マルツ
  - 部品を購入するなら、こちらも良いかもしれないです。
- StrawberryLinux
  - かゆいところに手が届くキットを扱っているイメージです。

3.4.2 海外通販ストア

- Aliexpress
  - 中国の EC サイトです。部品問屋のようなお店が多数出店しており、驚くほど安い値段で部品を購入できます。安いのはいいですが素性がよくわからない部品も多いので、品質は自分で担保する必要あります。
- eBay
  - アメリカのオークションサイトです。Aliexpress と同じような商品や、アメリカで出回っているキットなどが手に入ります。日本には発送してくれない商品も多いので要確認です。
- Digi-Key
  - アメリカの電子部品通信販売会社です。品揃えが豊富で素性はしっかりしているはずなので、日本で手に入りにくい部品をまとまった数購入するなら、こちらを利用することになると思います。

【コラム】コイルってどこにいる？

学校で習った電気回路の基本部品と言えば、抵抗、コンデンサ、コイルでしたが、これまでコイルは全然出てきませんでした。たまたま使っていないだけなのでしょうか？

抵抗とコンデンサを比較すると、プリント基板上でコイル部品を見かける頻度は多くありません。ノイズ除去や、高周波回路、電源回路のために利用するようなケースで見かけることが多いです。

電子回路で信号のやり取りをしている分には、IC と抵抗、コンデンサの組み合わせが主となると思います。電源回路でも、あらかじめ直流のアダプタを使えば、自分でコイルを使った回路を作る機会は多くないと思います。意外と登場人物が少ないな、と思ってみると、電子回路も少しは簡単に思えてきませんか？

第4章　自分だけの基板を

回路図は机上の世界のお話でしたが、次は回路に現実世界の体を与えるための作業となります。電子工作ではブレッドボードやユニバーサル基板による実装を行ったりしますが、本書ではちょっと背伸びをしてプリント基板を作ってみたいと思います。回路図までできていれば意外と簡単なので、臆せず挑戦してみましょう！

何よりも面倒なワイヤーの手配線を避けられますし、きれいに仕上がった本格的な基板を手にしたときの充実感はなかなかのものですよ。

4.1　プリント基板を使う理由

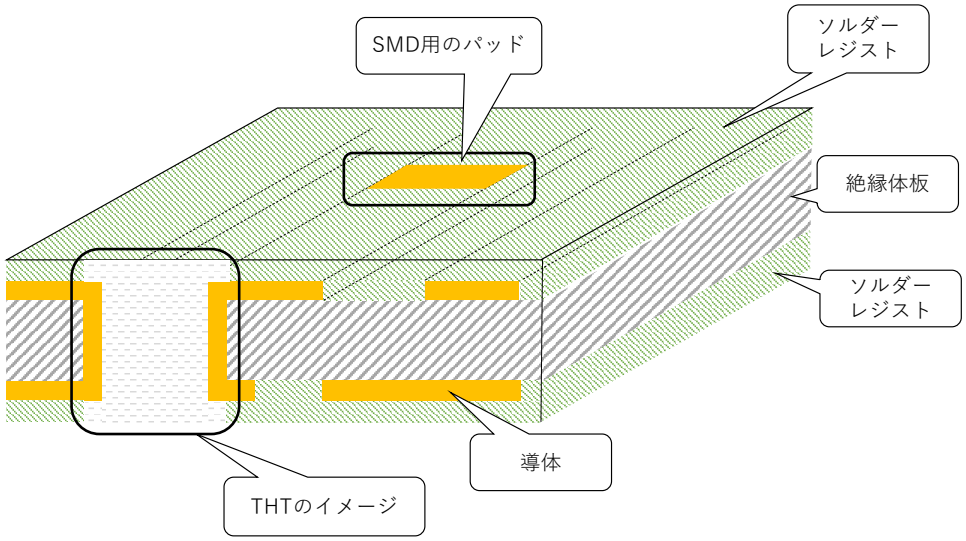
プリント基板とは、パソコンのマザーボードや、電子機器のケースを開けると出てくる、小さな部品がたくさん載った板のことです。あの板の上には銅の膜で作った配線が張り巡らされており、工作ではワイヤーを使って行っていた配線を、板の上で行っています。プリント基板を使うことで小さな部品を高密度で配置して、安定した品質で生産できるようになります。

個人向けのプリント基板の製造には、銅膜を張った専用の板にパターンを転写して、不要な銅膜を溶液で溶かして配線パターンを作るようなやり方がありましたが、なかなか手間が掛かるものでした。昔は業者に依頼するにも最低でも数万円掛かってましたが、最近では中国の業者に委託すると５枚から数ドルで製造してくれます。これだけ安いと、将来量産する予定がなくても、個人でもちょっと手の混んだ回路をしっかり作りたいときには、業者でプリント基板を製造してもらうことが現実的になります。

プリント基板のことを PCB と言ったりもしますが、PCB と言う場合、厳密には部品が実装されたプリント基板のことを指すようです。

4.1.1　プリント基板の構造

プリント基板にも色々種類がありますが、中国業者に発注するときに一番お手軽なのは二層基板だと思います。図 4.1 には二層プリント基板のイメージを示しています。



▲図 4.1: 二層プリント基板のイメージ

絶縁体の板に対して導体（主に銅）の薄い膜で作った配線を貼り付けて、その上から SMD 部品のはんだ付けなどで表に出す必要のある導体面以外をソルダーレジストを塗布してマスクしています。ソルダーレジストは導体を腐食から守りつつ、はんだも弾くのではんだ付けしやすくなります。よく緑色の基板を見かけるのは、ソルダーレジストの色によるものです。

二層と呼ぶのは板の表裏双方に配線が可能だからです。基板の表と裏の配線を繋ぐためには、基板に穴を空けて穴の側面に導体を貼り付けることで接続しています。配線用の穴のことはビアホールと呼んでおり、単にビアと言ったりもします。

THT の部品の取り付けにはビアよりも物理的に大きい穴を空けます。またはんだ付け用に穴の周囲のレジストは行わず、導体面が見えるようになっています。

このような構造を前提に基板のレイアウトを行っていきます。

より高度なプリント基板では、４層、６層といったものもあります。そのような基板では導体が基板の表裏だけでなく、基板の内部にも存在しており、表面から見えないところで配線を行うことができます。薄い銅の配線パターンを基版でサンドイッチしているような構造になっています。最近のスマホのような高度な電子機器ではこういった多層基板が使われています[*1]。

4.2 KiCad での基板デザインを覚える

プリント基板の構造を理解したところで、実際に KiCad を使って基板をデザインしていきましょう。基板のデザインのことはアートワークと呼んだりもします。

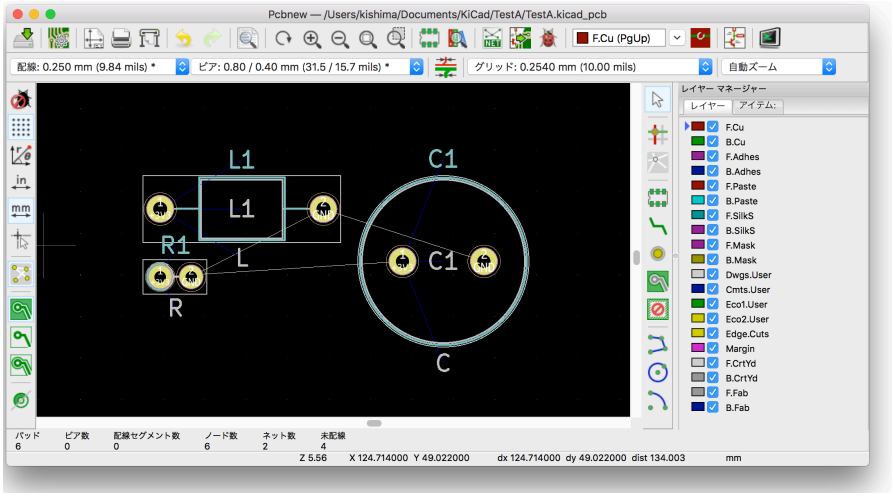
4.2.1 ネットリストのロード

まず Eeschema で出力したネットリストを読み込みます。メニューの [ツール][ネットリストのロード] から読み込みができます。ネットリストファイルに*.net ファイルを指定して読み込みま

[*1] もし手元に最近の電子機器の不要基板があれば、切断砥石などでカットしてみると、銅箔がサンドイッチされている様子がよくわかります。え、普通そんなことしないって？

す。設定は特に触らなくても大丈夫です。

図 4.2 には「3.2.4 KiCad の基本的な使い方」で作成したネットリストを読み込んだ直後の Pcbnew の編集画面の様子を示しています。



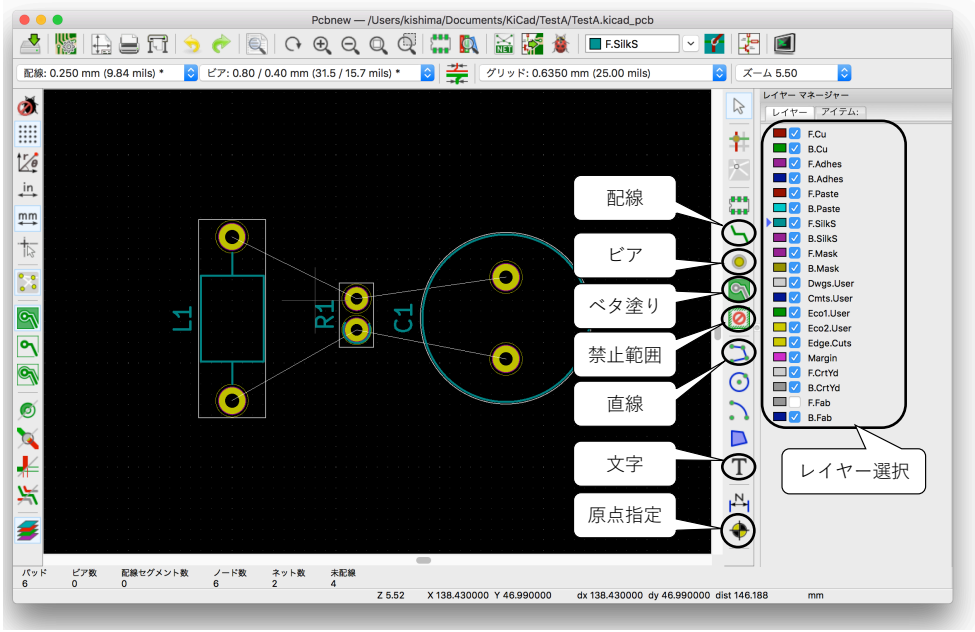
▲図 4.2: ネットリストロード後の状態

各部品のフットプリントが適当に配置されており、その端子間を白い線が結んでいることが分かると思います。白い線はラッツネストという名前で、部品の端子間の接続関係を示しています。

Pcbnew で行うほとんどの仕事は、部品を適切に並べて、ラッツネストで示された部品の接続関係を確認し、試行錯誤しつつ物理的な配線のパターン形状を描いていくことです。

4.2.2 ツールメニュー

配線を行う前に Pcbnew の機能のうちよく使いそうなものを簡単に説明します。図 4.3 には、各機能のボタンを示しています。



▲図 4.3: Pcbnew のよく使う機能

各機能の内容は以下のとおりです。

- レイヤー選択
  - 操作を行う対象のレイヤーを選択します。
- 配線
  - 端子間を接続する配線パターンを所定の太さで描きます。太さはあとからでも変更できます。
- ビア
  - ビアホールを空ける位置を指定します。配線中でもショートカット (v) でビアによる反対面への移行ができます。
- ベタ塗り
  - 配線ツールで描いたパターン以外の面を一様に塗りつぶすパターンを描きます。GND のベタパターンを描くためによく用いられます。
- 禁止範囲
  - ベタ塗り禁止の範囲を指定します。
- 直線
  - 直線を描きます。描いた線の意味は操作対象のレイヤーによって異なります。
- 文字
  - 文字を描きます。描いた文字の意味は操作対象のレイヤーによって異なります。
- 原点指定
  - 出力ファイル上で基板上の位置を示す際の原点を示します。通常は基板外形の左下を指定することが多いようです。

4.2.3 配線

では、メインの作業である配線を行ってみましょう。しかしその前にレイヤーの意味を把握しておく必要があります。

4.2.3.1 レイヤーの意味

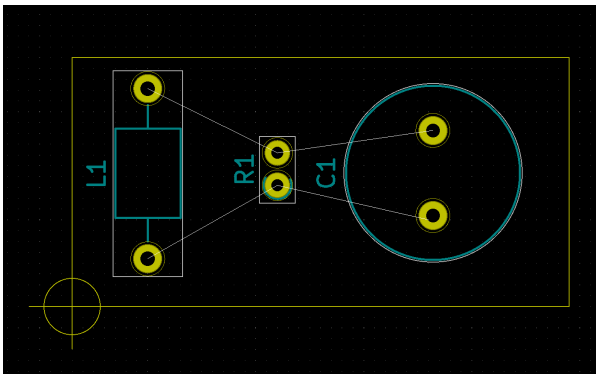
Pcbnew では基板をいくつかのレイヤーに分けてデザインしていきます。作業する際は、自分がどのレイヤーに対して処理をしているのかに注意しましょう。以下には各レイヤーの意味をまとめています。一見数が多くて大変そうですが、実際に編集する必要のあるレイヤーは多くはないので、安心してください。

F(Front) は基板表面を、B(Back) は基板裏面を表しています。最終的に基板製造のためガーバーデータとして出力されるレイヤーには（★）でマークを付けました。主に（★）マークの付いたレイヤーに着目して編集していけば OK です。

- F/B.Cu（★）
  - 導体面を表しています。配線機能で配線を行うのはこのレイヤーです。２層以上の多層基板では更に導体レイヤーが増えます。
- F/B.Adhes（未使用）
  - 接着剤のレイヤ、だそうです。部品をはんだ付け前に固定するボンドを塗る位置を指定するためのものです。
- F/B.Paste（未使用）
  - はんだペーストを載せる面を示します。SMD 部品をリフローで実装する場合に関係してきますが、今回は関係しません。
- F/B.Silks（★）
  - 白色のマーキングを表しています。部品名や基板名などを印刷するために用いられています。シルク、というのは昔はシルクスクリーンで印刷していた名残のようです。
- F/B.Mask（★）
  - ソルダーレジストを塗らない面を示しています。はんだ付けするための導体面が対応しています。フットプリントデータに情報が含まれているので、通常ユーザが手作業で指定する必要はないです。
- (Dwgs/Cmts/Eco1/Eco2).User（未使用）
  - ユーザ定義のレイヤーです。基板の補足情報的なものを書き込むために用いることができます。
- Edge.Cuts（★）
  - 基板の外形を表すレイヤーです。ここで描いた線にそって基板が切り抜かれます。
- Margin（未使用）
  - 部品と周囲の要素とのマージン距離を示すレイヤーのようです。
- F/B.CrtYd
  - 部品の外形の大きさを描くためのレイヤーです。このレイヤーで線が重なるようだと、実際に実装した際に部品同士が干渉してしまうと思われます。
- F/B.Fab
  - 製造（Fabrication）用の補足情報を書き込むためのレイヤーです。シルクレイヤーではないので、ここのレイヤーで表示されている内容は基板に反映はされません。

4.2.3.2 基板の外形

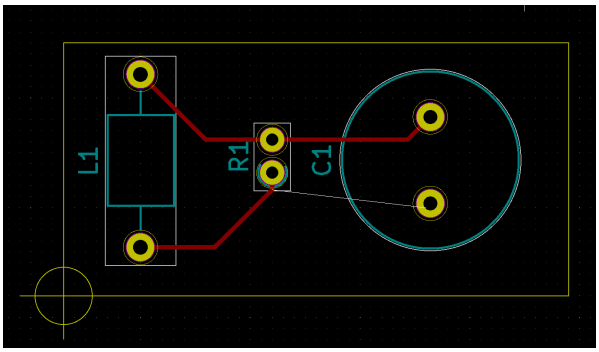
次に基板の形を決めてみましょう。まずは一番簡単な長方形の基板にしてみます。Edge.Cuts レイヤを選択した状態で、図 4.4 のように直線機能で部品を囲むように長方形を描いてみます。これで基板の形が決まりました。曲線機能で角を丸くしたり、斜めに切れ込みを入れたりするなど、物理的に無理がない範囲で自由に設定できます。



▲図 4.4: 基板外形の指定

4.2.3.3 繋いでみる

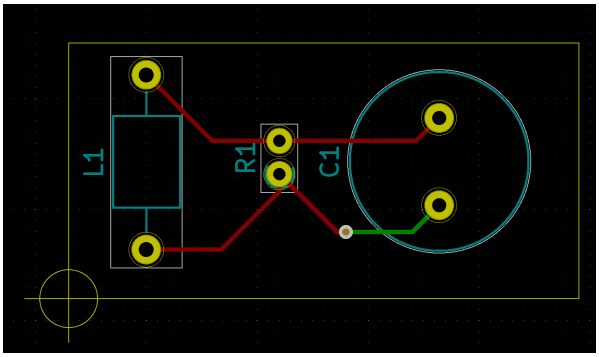
そして配線機能で、図 4.5 のように各部品の端子を繋いでみます。操作を行うときは、F.Cu を選択してください。端子間を繋ぐと、対応するラッツネストの白い線が消えて、接続が完了したことが分かります。ラッツネストの線がなくなるまで部品間を接続していきます。



▲図 4.5: 配線例

4.2.3.4 基板の表と裏

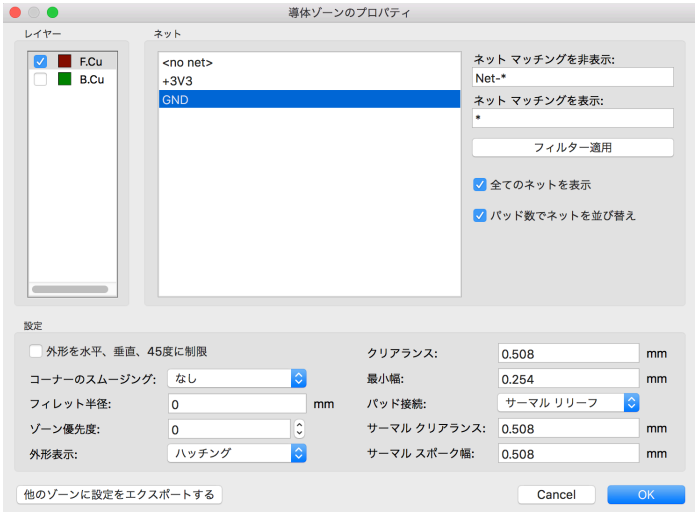
二層基板の使い方の練習として、R1 から C1 への接続にビアを使った配線を行ってみます。F.Cu を選択した状態で、R1 の端子から線を伸ばし、途中で v キーを押すとビアを打つ状態に切り替わります。適当なところでクリックすると、そのポイントでビアが打たれ、配線が裏面に回ります。C1 は THT で裏面からも接続できるので、そのまま C1 の端子につなげます。そうすると、図 4.6 のような状態になります。



▲図 4.6: ビアを利用した配線

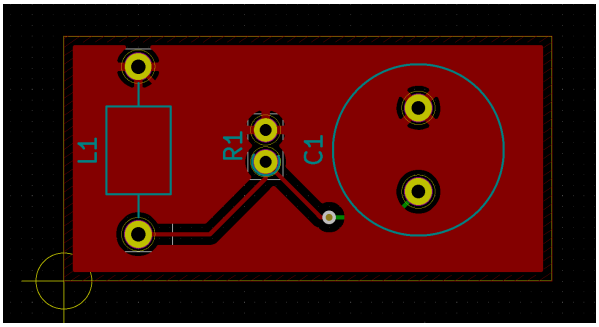
4.2.3.5 ベタ GND

次に配線の周囲を GND で囲ってみましょう。ベタ塗りのボタンをクリックして、基板外形の角をクリックすると、図 4.7 のような設定画面が表示されます。ベタ塗りを何と接続するかを、「ネット」から選択します。通常は GND を選択して、配線を囲むようなベタを塗ります。その他のオプションは今はそのままで大丈夫です。



▲図 4.7: ベタ塗りの設定

OK を選択したあと、基板の四隅をクリックしてベタ塗りの範囲を確定させると、図 4.8 のような状態になります。3V3 の配線を避けるような形でベタ GND が塗られていることが分かります。GND の配線はベタと同じ GND なので、配線を避けずに上塗りされています。



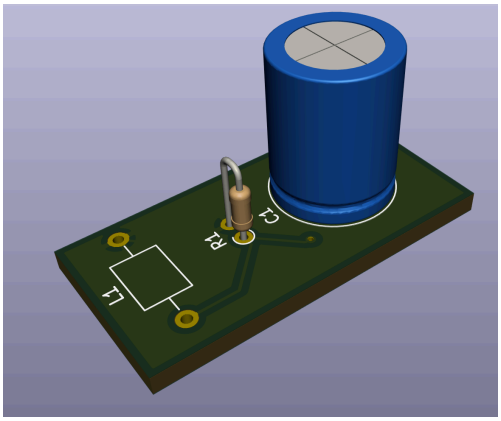
▲図 4.8: ネットリストロード後の基板

4.2.3.6　シルク

基板上に自由に文字や図形を書き込むことも可能です。Silks レイヤーを選択して、直線機能や、文字描画機能で自由に必要な情報を書き込めます。標準では部品番号をシルクに印字するような設定になっています。不要であれば削除も可能です。

4.2.3.7　確認

これでお試しの回路は完成ですが、基板が実際どんな見た目になるのか、3D で確認することもできます。[表示][3D ビューアー] を選択すると、図 4.9 のような表示になります。フットプリントに 3D モデルが設定されていると、R1 は C1 のように部品の形状が表示されます。L1 には 3D モデルがないので何も表示されていません。自作のフットプリントを用いた場合は 3D モデルまで準備するのは難しいですが、シルクの位置など 3D ビューアーじゃないと気づきにくい点もあるので、基板が完成したら一度 3D で確認してみると良いと思います。



▲図 4.9: 3D 表示した基板

そして最後にデザインルールチェック（DRC）を行います。DRC では、部品の配線漏れがないか、配線や部品間の距離が所定の範囲に収まっているかをチェックします。もしここで配線間の距離が近すぎて、基板の製造装置のスペックを超えたりすると、配線同士が繋がってしまったり期待しない結果になってしまいます。てんとう虫ボタンをクリックして、「配線のエラーをすべてレポート」「塗り潰した導体の領域に対して配線を検査」にもチェック入れて「DRC を実行」をク

リックします。問題がなければ何も中央の「問題/マーカー」欄には表示されません。問題があった場合は個別に見直して、配線し直すなどして微調整することになります。

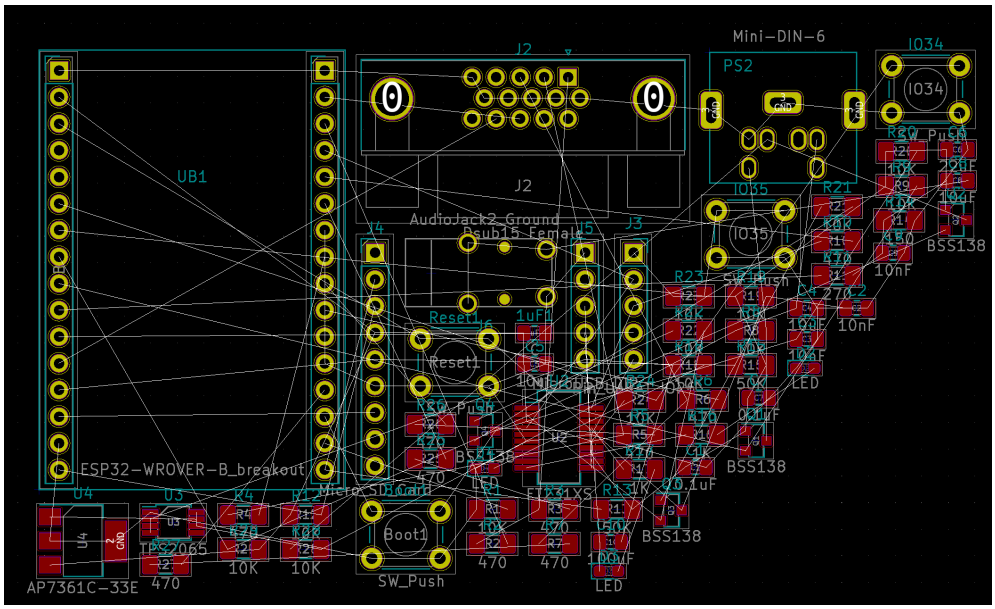
ここまで完了すると、あと最後にデータをエクスポートして、基板業者に渡せるようになります。

4.3 本格的に基板を作ってみよう

前節で Pcbnew の基本的な使い方を確認しました。では、Narya Board の配線を実際に行ってみましょう。

4.3.1 ネットリストのロード（本番）

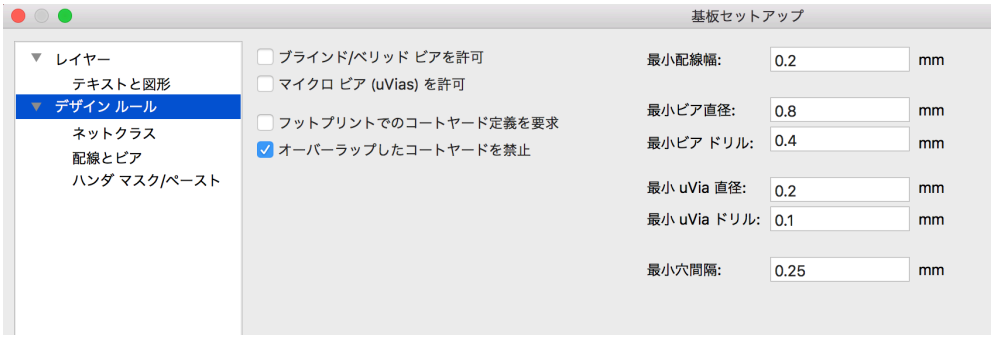
Narya Board のネットリストを読み込んだ状態が、図 4.10 です。練習で試した回路と違って、大量の部品が並んでいて、ラッツネストもかなり複雑なことになっていることが分かります。これを地道に配線していくことになります。



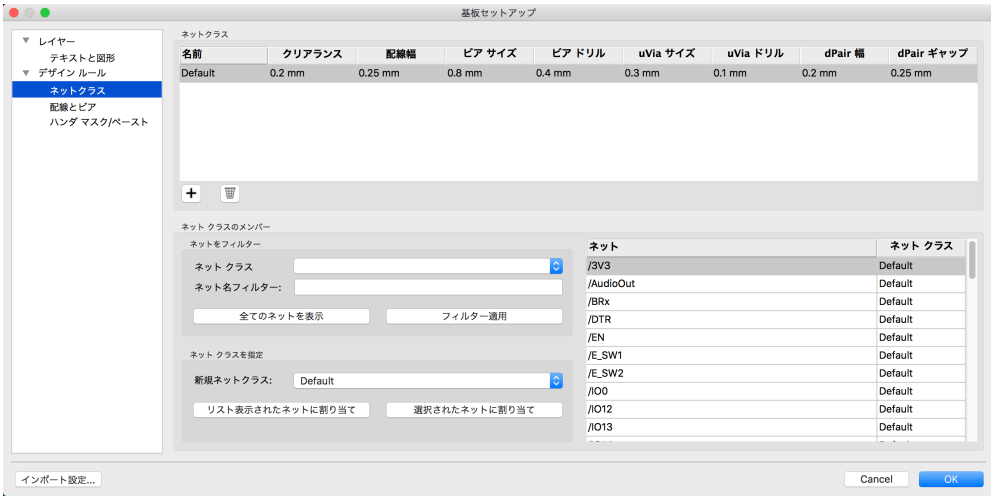
▲図 4.10: ネットリストロード後の基板

4.3.2 設定

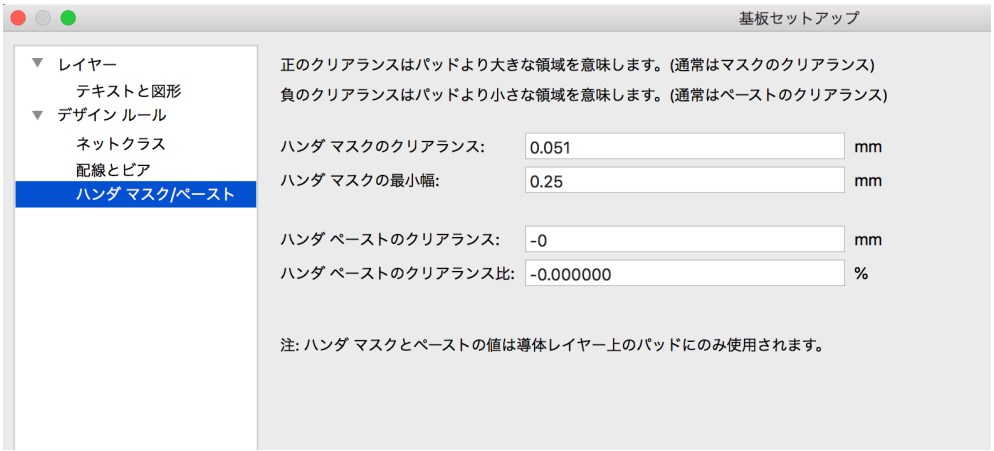
最終的に基板を製造する際に、その基板メーカーの持っている装置のスペックによって、どれだけ細かいパターンの基板を作れるかが決まってきます。線の幅の上限等を設定値で決めることができます。配線を始める前に自分の委託先のスペックに合わせておく必要があります。基板メーカーによって細かいスペックは異なってくるのですが、今回は筆者が Elecrow というメーカーに製造を委託した際に使用した設定値を図 4.11、図 4.12、図 4.13 に示します。ほぼデフォルト値で、きちんと確認して調整したわけではないので、あくまで参考値として参照ください。[ファイル][基板セットアップ] から設定できます。



▲図 4.11: デザインルール



▲図 4.12: ネットクラス



▲図 4.13: はんだマスクペースト

4.3.3 配線の注意点

配線の仕方については、色々と考慮するべきことがあり、職人的な世界でもあると聞いています。筆者も素人なので分からないことだらけですが、配線をしていく過程で、気に掛けていることを挙げます。

4.3.3.1 電源

電流が多く流れる配線は太くするのが基本です。太くすることで配線自身が持つ抵抗値が下がり、電圧を下げずに電流を運ぶことができます。

4.3.3.2 GND

信号線の周りを GND で囲むことでノイズの影響を受けにくくなると聞いているので、そのようにしています。しかし一方で細長い GND のパターンはノイズを拾うアンテナにもなるそうなので、そういった箇所ができるだけ生じないようにしています。また GND は裏表で同じ電位なるように、適宜ベタ GND 上にビアを打って、表裏の GND を連結しています。筆者ももう少し勉強して、定量的に語れるようになりたいです。

4.3.3.3 パッド

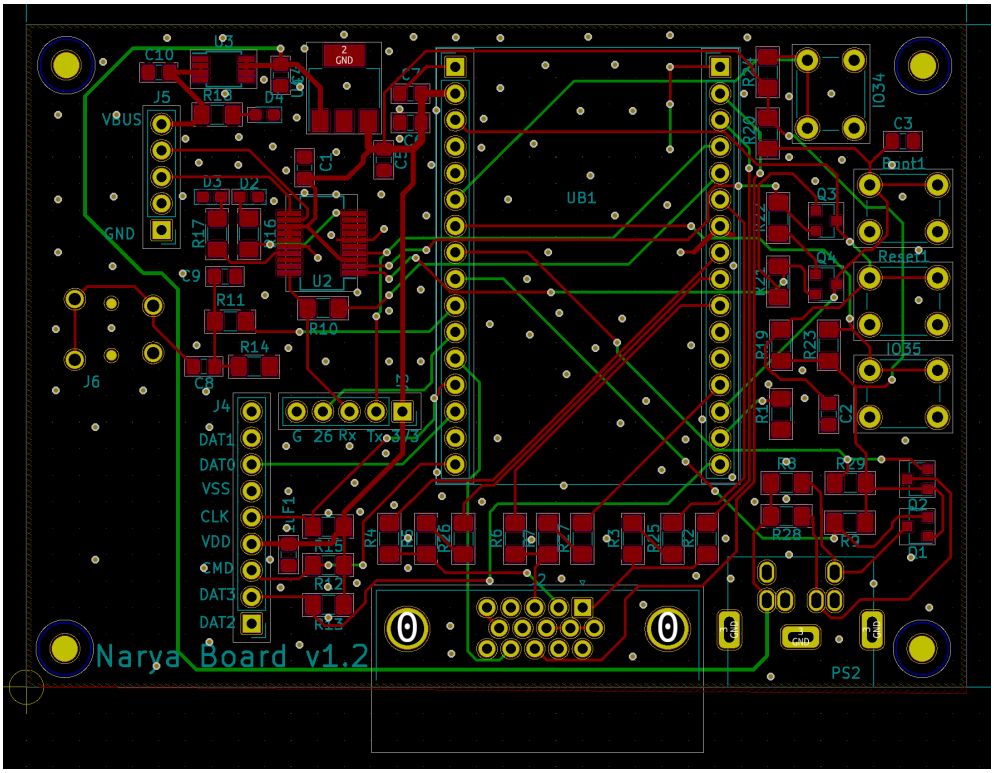
SMD 部品を手作業ではんだ付けする場合は、パッドが小さすぎないか確認しています。もし小さすぎる場合は、フットプリントライブラリを修正して、パッドの大きさを大きくしたりしています。

4.3.3.4 自動配線

個人プロジェクトと言えど、この程度の規模でも配線作業は半日以上掛かりました。実は配線を自動的にやってくれるツールもあります。今回は配線の練習も兼ねてすべて手作業で行いましたが、freerouting という自動配線ツールも KiCad と組み合わせて使うことができます。「freerouting kicad」というキーワードで検索してみると、使い方が多数見つかるかと思います。

4.3.4 配線を終えて

配線を頑張って終えた結果が図 4.14 です。（ベタ GND は非表示にしています）これで基板のデザインは完了です。おつかれさまでした。

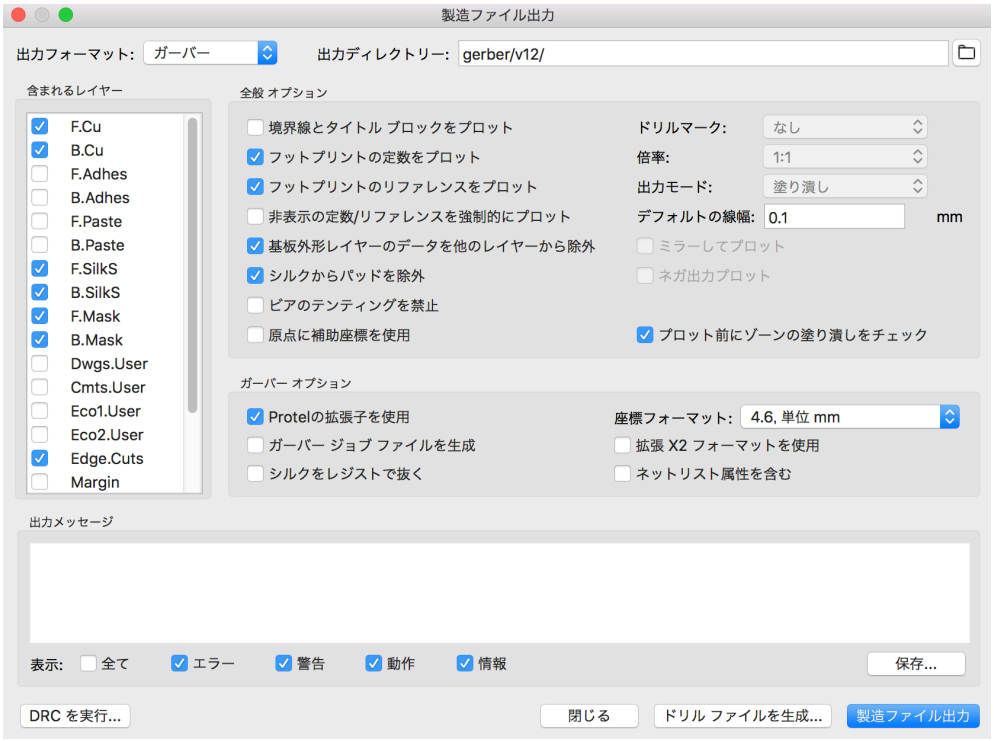


▲図 4.14: 配線を完了したあとの基板

4.4 ガーバーデータの出力

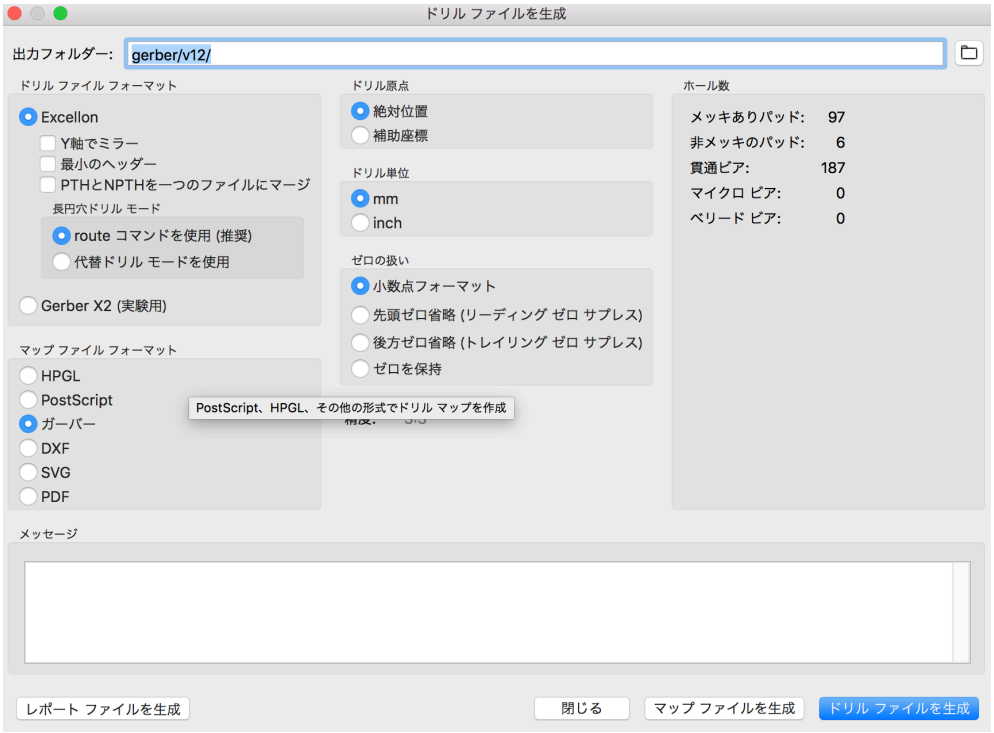
仕上がった基板データを基板製造業者に製造委託するために、汎用の基板製造用フォーマットに変換出力します。汎用の基板製造用フォーマットのことをガーバー（Gerber）フォーマットと呼んでいます。

KiCad では、[ファイル][プロット] で出力できます。図 4.15 には、プロットの設定を示しています。二層基板の場合は、「含まれるレイヤー」に F/B.Cu、F/B.SilkS、F/B.Mask、Edge.Cuts を指定します。出力先フォルダも指定したのち、「製造ファイル出力」ボタンをクリックして、ガーバーファイルを出力します。



▲図 4.15: KiCAD の画面

そして基板への穴あけの情報は別途ドリルファイルで指定します。プロットの画面の「ドリルファイルを生成」ボタンをクリックします。図 4.16 のように設定して、「ドリルファイルを生成」ボタンをクリックしてドリルファイルを出力します。



▲図 4.16: KiCAD の画面

"プロジェクト名.drl"がメッキありで、"プロジェクト-NPTH.drl"がメッキなしのドリル穴の位置を示すファイルです。

4.5 プリント基板メーカーへ製造依頼

ガーバーデータがあれば、あとはプリント基板メーカーに製造を依頼するだけです。最近では色々なメーカーが個人向けにも製造してくれるようになっていますが、今回は比較的使用している方が多そうだった、中国の Elecrow というサービスを利用してみようと思います。

4.5.1 ファイル名の変換

Elecrow にガーバーファイルを提出する際には、ファイル名と拡張子を所定の形に揃える必要があります。ドリルファイルと合わせて下記のようなシェルスクリプトでまとめて変換してしまうのが簡単です。引数（$1）にはプロジェクトファイル名が入ります。

```
#!/bin/bash
mv $1-B_Cu.gbl $1.GBL
mv $1-B_Silks.gbo $1.GBO
mv $1-B_Mask.gbs $1.GBS
mv $1-Edge_Cuts.gm1 $1.GML
mv $1-F_Cu.gtl $1.GTL
mv $1-F_Silks.gto $1.GTO
mv $1-F_Mask.gts $1.GTS
mv $1-NPTH.drl $1-NPTH.TXT
```

```
mv $1-PTH.drl $1.TXT
```

変換したファイルを zip で固めると準備完了です。

4.5.2 Web サイトで登録

以下の Elecrow の受付ページを開いてみます。

`https://www.elecrow.com/pcb-manufacturing.html`

まずは画面右上のリンクからアカウントを作成しておきます。支払い方法には PayPal が選択できるので安心です。送付は国際配送になるので、住所をローマ字で記入します。住所の書き方は「国際 住所 書き方」といったキーワードで検索すると色々例が出てくると思います。

すると図 4.17 のような画面が表示されます。（英語で表示される場合もあるみたいです）こちらで製造の依頼を行います。

初期値では二層基板が選択されています。最低限設定必要なのは、寸法と製造枚数と国指定です。寸法が 100mm*100mm 以内で、5〜10 枚程度だととても安く製造できます。最低価格で 5$を切るのはすごいですよね。寸法が 100mm を超えると標準のサイズを超えるためちょっと高くなります。生産時間については割増料金を払うと特急で製造してもらえます。筆者の経験では、通常生産でも長期休み前の繁忙期でなければ一週間以内に製造完了するので、十分速いと思います。配送方法も選択できますが、あとの画面でももう一度選択することになります。

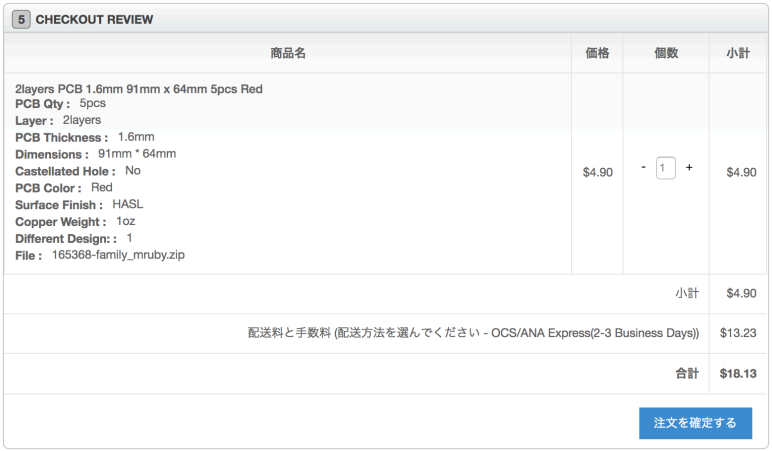
zip で固めたガーバーデータも画面上の方のボタンをクリックしてアップロードします。

▲図 4.17: Elecrow の注文画面

そして一通り設定が終わったら、「カートに入れる」ボタンをクリックして注文をカートに入れます。入れたあとはチェックアウトを行うと実際の支払い画面に進みますので、あとは画面に従って PayPal 等の操作を進めていくと、最終的には図 4.18 のような画面が表示されます。この際複

数の基板の注文をカートに入れてまとめて発注すると、１回分の配送費で済みます。基板の製造費と比較すると配送費が馬鹿にならないので、いくつか作りたい基板がある場合、まとめて発注するのがよいと思います。

比較的安くて配送が早いので、配送方法には OCS/ANA を利用しています。



▲図 4.18: 発注確認の画面

発注が完了すると、図 4.19 のようなメッセージが表示されます。メールボックスにも注文完了のメッセージが届いているはずです。

▲図 4.19: 注文完了の画面

4.6 基板の完成

Elecrow のユーザーアカウントページを開き、対象の発注を確認すると、製造状態が分かります。「In Production」となっていれば製造中で、一週間以内には製造完了して、「Shipped」という状態に変わるはずです。送付したガーバーデータに不備があれば、連絡のメールが届くこともあるようですが、筆者はまだ経験したことはないです。特に不備がなければ、英語でのコミュニケーションも不要です。Web サイト上でのデータのアップロードと仕様の指定だけで完了します。

状態が「Shipped」となったタイミングで、メールで製造した基板の写真と共に発送完了した旨の英文メールが届きます。こちらに書かれている OCS トラッキング情報で配送状態が分かります。筆者の場合は、OCS を使用した場合、メールが届いたタイミングで荷物がすでに通関済みで佐川急便に引き渡されていました。

あとは荷物を受け取るだけです。届いた荷物には納品書と共に、図 4.20 のようにビニールで脱

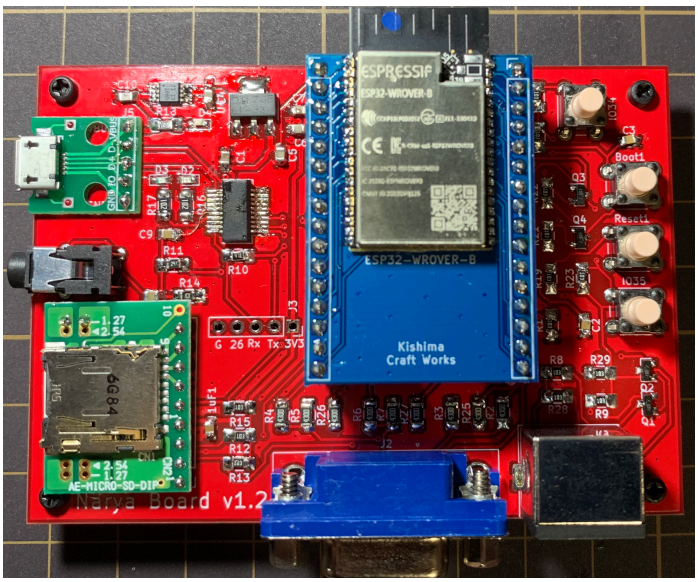
気密封された基板が入っています。

▲図 4.20: 到着したボード

4.6.1 動作確認

お楽しみの基板の組み立てです。部品を一つ一つはんだ付けしていくことになりますが、その前にまず基板の表面を確認して、パターンにずれや繋がってはいけない部分が繋がっていないか確認します。細かいシルクやレジストのズレ等はあるようです。またテスターで主要な各配線が期待通り繋がっているか確認していきます。

そしてはんだ付けが完了した結果が図 4.21 です。初めて電源を入れる前は最低限、電源と GND がショートしていないか確認しておきましょう。

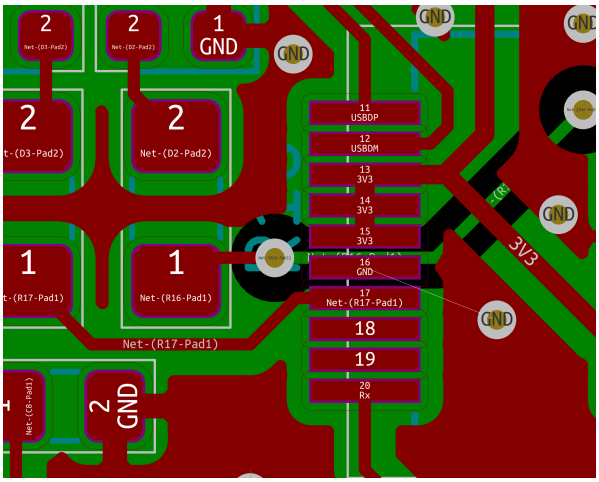


▲図 4.21: はんだ付けが終わった Narya Board の写真

動作確認用に基本的なソフトの準備も終えておくのが望ましいです。ソフトの開発について次章以降で解説しています。

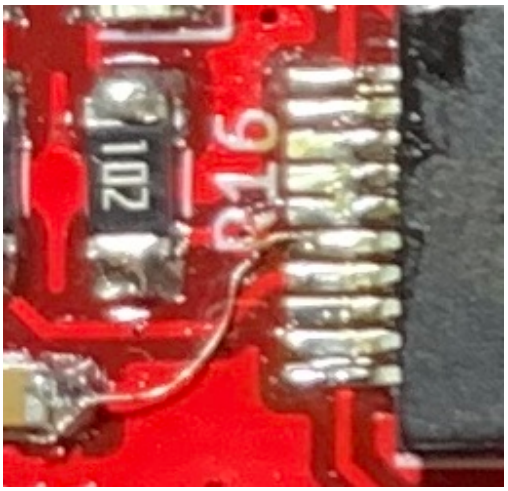
4.6.1.1　ミスの例

実装していく過程で、失敗に気づくこともあります。例えば、図 4.22 では、USB シリアル変換 IC の 16 番の GND ピンの一つがどこにも接続されていません。おそらく最初はベタ GND と接続していたのですが、ベタ塗りの領域を見直しているうちにうっかり接続を切ってしまっていたと思われます。ガーバーデータを出力する前の最終チェックが不十分でした。



▲図 4.22: GND の接続漏れ

対策として、図 4.23 のように IC の近くの GND のパッドから配線して修正しています。このような作業をリワークと呼んでいます。



▲図 4.23: リワークの結果

人間のやることなので、ある程度の規模の場合、このようにミスが発生することもあらかじめ視野にいれておいた方がよいと思います。

4.6.2 組み立ての知恵

基板が届いたところで、はんだ付けを行っていくことになるのですが、SMD 部品を多用すると細かい作業が増えて大変です。スムーズに作業を進めるためには、道具の準備も大切です。

4.6.2.1 ほしい用具

筆者が揃えてよかったと思う道具を以下に挙げてみます。

- それなりのはんだごてとこて台
  - 温度調節付きのこて先を取り替えられるものが作業が安定します。筆者は白光の FX600 を使っています。あまり安いものは本当に作業しづらいので、はんだごては、しっかりしたものを買った方がよいと思います。こて台も簡易的なものではなくて、重しが入っていて安定して使えるものがあるとうっかり転げたりしないので安心です。
- 細いこて先
  - 細かい作業には細いこて先が必要です。はんだも細めのものがあった方がベターです。
- フラックス、フラックスクリーナー
  - IC の足のはんだ付けにはフラックスがないと辛いです。フラックスはずっと残しておくと基板を腐食するそうなので、クリーナーも買ってみました。
- 拡大鏡
  - まだ老眼じゃないから大丈夫だろう、と思ってましたが、IC の細かい足の部分は流石にルーペがないとはんだ付けの確認が無理でした。最終的には液晶モニタ付きの実体顕微鏡も購入したりしています。
- 手元を照らせるライト
  - 細かい作業をするために、手元をピンポイントで照らせる LED ライトを買って重宝しています。
- 短絡確認できるテスター
  - 配線がショートしているかどうか音で確認できるテスターがあると、基板の配線のチェックや、はんだ付けの結果確認に役立ちます。

【コラム】部品の固定

大型の部品の固定にはホットボンドを用いたりしますが、絶縁性があまり良くはないようなので、筆者は回路に直接触れるようなところにはあまり用いないようにしています。

代わりに筆者が USB の端子などもげやすい部品の補強に用いているのは、紫外線硬化レジンです。100 円ショップでも手芸向けに売っており、絶縁性も高いようなので、部品の固定に時々使っています。

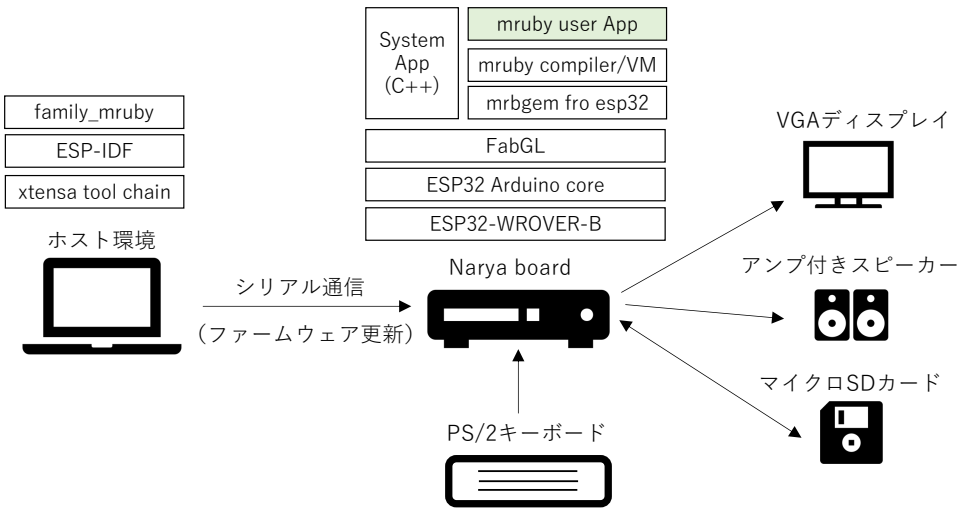
第5章　mrubyを思いのままに！

前章までで mruby を動かすためのハードウェアの準備が整いました！ ここからはソフトウェアがメインのパートになります。まずは mruby を ESP32 向けにビルドして動かしてみましょう。

Linux や Mac であれば make 一発で済んだ mruby のビルドも、ESP32 で実現するためには一工夫必要になります。

5.1　Family mruby のシステム構成

mruby を Narya ボード上で動かすためのソフトウェア観点のシステム構成について少し考えてみましょう。Family mruby では、mruby で書かれたアプリを実行する環境を構築することを目標としています。そのため、ESP32 上で、mruby のコンパイラと VM を実行します。ソフトウェアのスタック構成とシステムの構成図を図 5.1 に示しました。Narya ボード上で動かす基本ソフト（ファームウェア）はホスト PC 上でクロスコンパイルによりビルドして、基板上にあらかじめシリアル通信で書き込みます。使用者は、基本ソフトが動いている状態で、mruby のコードを PS/2 キーボードと VGA モニタを使ってプログラミングし、ホスト PC 要らずでそのまま Narya ボード上で実行可能とするような構成を想定しています。



▲図 5.1: Family mruby のシステム構成

図 5.1 にはいくつか見慣れない単語も登場していますが、本章と次章で追って説明していきます。

5.2 ESP32 向けのソフト開発環境の選択

まずは ESP32 で自作のソフトウェアをビルドするための環境について確認していきましょう。ESP32 向けの公式開発環境としては、ESP-IDF と ArduinoIDE があります。それら 2 つについてざっくりと見てみましょう。

5.2.1 ESP-IDF

ESP32 の最も基本的な開発環境です。公式サイトの導入手順に従って、ツールチェインの導入、ESP-IDF のダウンロードという手順を踏みます。コマンドライン上でのビルドを行うので、一般的なソースコードのビルドに慣れている方であれば、問題なく導入できるかと思います。環境を最新に保つことで、最新のバグフィックスや機能追加の恩恵を受けることができます。特に Mac、Linux であればスムーズに利用できるかと思います。

本書では mruby のビルドとの相性の良さから、こちらを選択することにします。具体的な利用手順は後述します。

5.2.2 ArduinoIDE

ArduinoIDE を使用する場合、ボードマネージャから開発環境をインストール可能です。執筆時点の最新安定版は v1.0.2 となっています。インストールが容易で、Arduino 向けのライブラリ資産も活用できるので、一番使用されている環境なのではないでしょうか。新機能の開発や不具合修正は ESP-IDF で先行して進んでいるので、若干反映が遅いという点と、Arduino ライブラリ以外の資産のビルドが難しいという点がデメリットだと思います。本書では、mruby をビルドする都合もあり、ESP-IDF の方を利用するため、詳細には触れないですが、Web 上にも情報は豊富なので、利用を始めるのは容易かと思います。

5.3 評価用基板

自作の基板に ESP32 を搭載することは決めていますが、自作の基板でいきなり動作確認をする前に、汎用の評価用基板があれば、それを使って環境構築や基本的な動作の確認をするのがベターです。いきなり自作基板での評価を行うと、問題があったときにそれが自分の開発した基板の問題なのか、ソフトの実装の問題なのか切り分けが難しくなります。

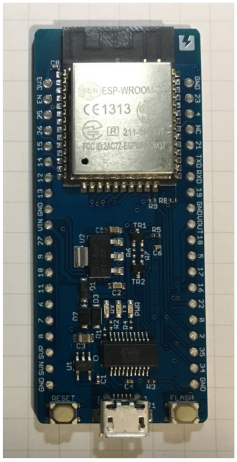
ここでは ESP32 を搭載した評価用に使える開発用基板を紹介したいと思います。

ESP32 は、アンテナパターンや Flash メモリをあわせて金属でシールドしたパッケージで販売されています。日本で使用する場合、それらの中から日本の認証を取得したものを選ぶことになります。パッケージ上に技適マークが付いているので、それで見分けが付きます。

特に有名なのは ESP32-WROOM-32 です。

今回は ESP32-WROVER-B を利用しますが、こちらも認証を取得しており、SPI に PSRAM が接続されている以外は、ESP32-WROOM-32 とほぼ同じなので、ESP32-WROOM-32 搭載基板も十分評価に使用できます。

ESP32-DevkitC



ESPr Developer 32

M5Stack Basic

▲図 5.2: ESP32 を搭載した開発用基板

以降、図 5.2 に掲載した基板について簡単に紹介します。

5.3.1 ESP32-DevKitC

ESP32 の開発元の Espressif Systems が提供している開発用基板です。国内でも購入可能ですし、中国の通販サイトではより安く購入可能ですが、動作が不安定なものも混じっているようなので、不安な方は日本の店舗から購入した方が良いと思います。

また日本のお店では取り扱いは無いようですが、ESP32-WROVER-B を搭載した ESP32-DevKitC-VB というのもあります。

`http://akizukidenshi.com/catalog/g/gM-11819/`

5.3.2 ESPr Developer32

ESP32-DevKitC とよく似ていますが、日本の Switch Science が開発した基板です。少し高めですが、電源周辺を強化しており、より安定した動作が可能と思われます。

`https://www.switch-science.com/catalog/3210/`

5.3.3 M5 Stack

M5Stack 社が開発している ESP32 をコアとして、液晶ディスプレイやボタン、バッテリー等を統合したデバイスです。コアは ESP32 なので、ESP32 の開発環境としても利用できます。

`https://www.switch-science.com/catalog/3647/`

5.4 ESP-IDF ビルド環境の構築

では、ESP-IDF 開発環境を導入して、mruby をビルドできるところまでやってみましょう。今回は執筆時点の最新安定版（v3.2.2）を利用して Mac にインストールしていきます。[*1]細かい手順

[*1] v4 以降では CMake を使用するようになり、手順が変わります。

は公式サイトにも書かれています。
`https://docs.espressif.com/projects/esp-idf/en/v3.2.2/get-started/macos-setup.html`

5.4.1 xtensa ツールチェイン[*2]のインストール

ESP32 用のクロスコンパイラを導入しましょう。ESP32 のコアは Xtensa LX6 というプロセッサが使われています。
以下にざっくりと手順をまとめてみます。

- python のパッケージをインストールするために、pip が使える状態にします。
- `https://dl.espressif.com/dl/xtensa-esp32-elf-osx-1.22.0-80-g6c4433a-5.2.0.tar.gz` をダウンロードします。
- ダウンロードしたファイルを以下のように展開します。

```
$ mkdir -p ~/esp
$ cd ~/esp
$ tar -xzf ~/Downloads/xtensa-esp32-elf-osx-1.22.0-80-g6c4433a-5.2.0.tar.gz
```

- `~/.profile` に以下の行を追加して、ツールチェインを PATH に加えます。

```
export PATH=$HOME/esp/xtensa-esp32-elf/bin:$PATH
export IDF_PATH=~/esp/esp-idf
```

5.4.1.1 ESP-IDF のインストール

ESP-IDF は github で管理されています。以下のような手順で、ESP-IDF を`~/esp/esp-idf` にインストールします。ESP-IDF は、submodule を従えているでの、`--recurisve` を忘れないようにしましょう。

```
cd ~/esp
git clone -b v3.2.2 --recursive https://github.com/espressif/esp-idf.git
python -m pip install --user -r $IDF_PATH/requirements.txt
```

以上で準備は完了です。
ESP-IDF を更新したいときは、`~/esp/esp-idf` を git で更新するか、全部消して git clone からやり直してもよいかと思います。

5.4.2 ESP32 アプリプロジェクトのディレクトリ構成

アプリケーションをビルドするための最小のディレクトリ構成は以下のような形です。

```
your_app/
    |-- main/
    |   |-- component.mk    #ビルドのオプション設定。空でも OK
    |   |-- your_app_main.c #アプリの実装
```

[*2] コンパイル環境のツールセットをツールチェインと呼びます。

```
    |
    |-- Makefile            #ESP32 プロジェクト設定を include した Makefile
```

Makefile には以下のように設定します。

▼リスト 5.1: Makefile の中身

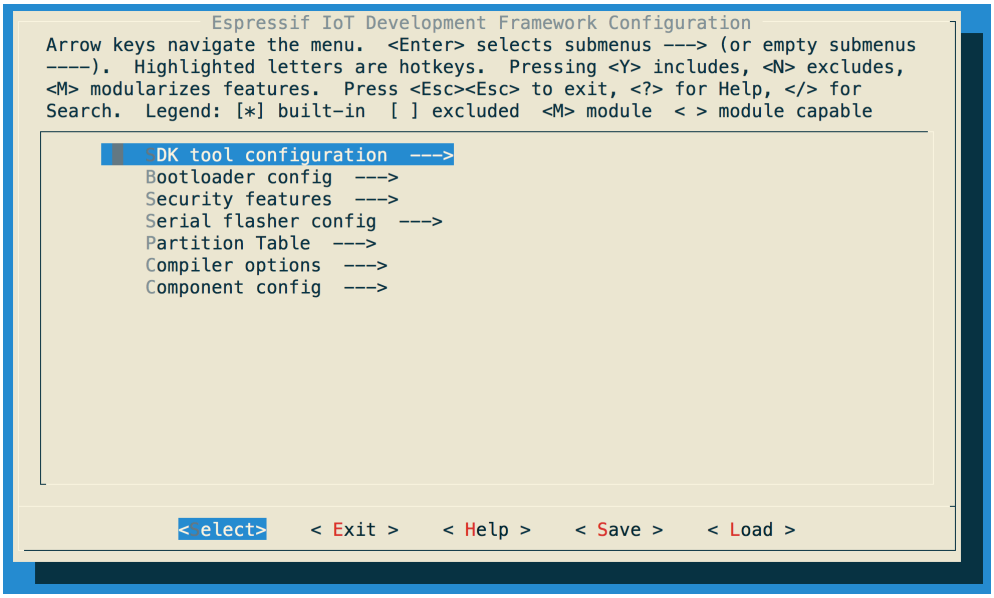
```
PROJECT_NAME := your_app
include $(IDF_PATH)/make/project.mk
```

`$(IDF_PATH)/make/project.mk` には、基本的な make 設定が書かれていますが、通常は意識しなくても大丈夫です。

`your_app_main.c` にアプリケーション処理をプログラミングしていきます。アプリケーションは処理の開始地点は、`main()` ではなくて、`void app_main()` です。ESP-IDF では、OS として FreeRTOS が使用されています。諸々の起動時の処理はアプリケーション開発者が気にしなくても済むようになっています。

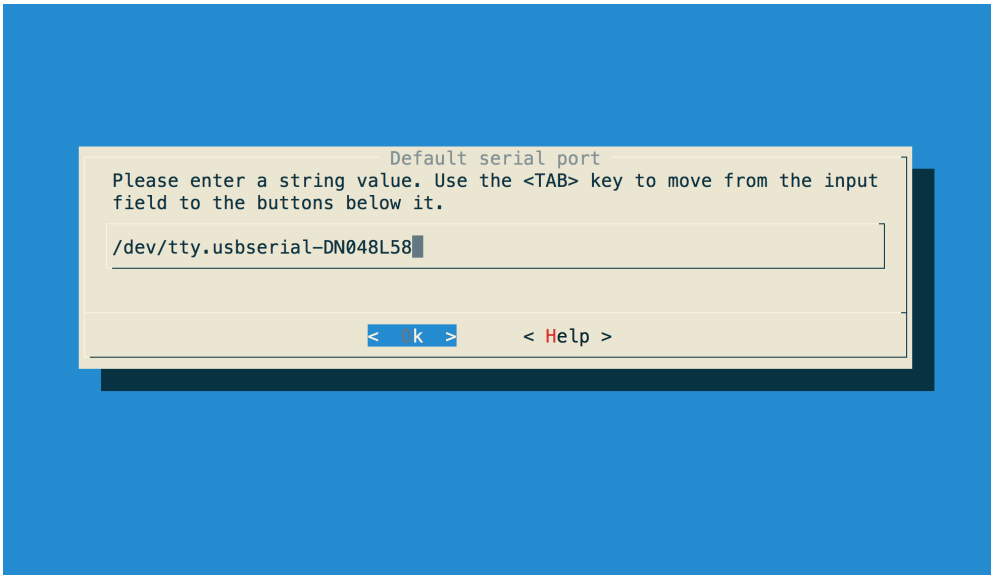
5.4.3 ESP32 アプリケーションのビルド

ESP32 の各種設定や、FreeRTOS の初期設定等、各種のビルドに必要な設定は、`make menuconfig` で生成します。ターミナルを開き、`your_app/`に移動して、`make menuconfig` を実行すると、図 5.3 のような設定が表示されます。



▲図 5.3: make menuconfig による設定編集画面

特にソフトウェアをシリアルで書き込む場合、シリアルポートの設定が必須となります。メニューから"Serial flasher config"を選択して、図 5.4 のようにシリアルポートの名前を設定します。



▲図 5.4: ソフト書き込み用のシリアルポート設定

MacOS に USB で接続する場合、`ls /dev/tty.*`で表示されるデバイスから、serial、USB といった名前のデバイスを見つけて設定します。USB コードを抜き差しすると、消えたり現れたりするので、見つけられるかと思います。

設定を終えて、exit を選択して、設定を保存すると `sdkconfig` というファイルと、`build/`というディレクトリが追加されます。`sdkconfig` には、先程設定した結果がテキストとして格納されており、ビルド時に参照され、コンパイルオプションに反映されます。

以下のようなコマンドを使って、ビルドしたり、ボードに書き込みしたりすることできます。

- `make`：ビルドしてバイナリを生成する
- `make flash`：ビルド＆バイナリをボードに転送する
- `make monitor`：シリアルポートのログをモニターする

ESP32-DevkitC などの開発ボードがあれば、

`~/esp/esp-idf/examples/get-started/hello_world` のサンプルを適当なところにコピーして、ビルドと書き込みを一度試してみると良いと思います。

`make menuconfig` で設定できるオプションは非常にたくさんあります。だいたいはそのままで大丈夫ですが、詳細は以下の公式サイトの資料を参照すると（ざっくりとした）説明が載っています。

`https://docs.espressif.com/projects/esp-idf/en/v3.2.2/api-reference/kconfig.html`

5.5 mruby を ESP32 へ

ESP32 の基本的な開発環境が整ったところで、本題の mruby の移植に取り掛かりましょう。

色々と準備することも多そうで、大変そうに思えますが、・・・実はすでに移植済みの環境が github に公開されております。

"mruby-esp32/mruby-esp32"というリポジトリが mruby を ESP32 環境に移植したコンポーネントです[*3]。こちらには、先程説明した ESP32 のプロジェクトに mruby が最初から取り込まれており、手順に従ってビルドするだけで、簡単に mruby を ESP32 上で動かすことができます。

`https://github.com/mruby-esp32/mruby-esp32`

5.5.1 mruby-esp32 の使い方

mrubt-esp32 の使い方について説明したいと思います。基本的には github の Readme の通りにやれば大丈夫です。

まずは github よりリポジトリを clone します。

```
git clone --recursive https://github.com/mruby-esp32/mruby-esp32.git
```

mruby-esp32 は以下のような構成になっています。

```
mruby-esp32/
    |-- main/
    |   |-- component.mk  #ビルドのオプション設定
    |   |-- mruby_main.c  #mruby の組み込みと実行
    |   -- examples/
    |         |-- simplest_mrb.rb      #Ruby スクリプト例
    |         |-- system_mrb.rb
    |         |-- wifi_example_mrb.rb
    |
    |-- components/
    |   |-- mruby_component/             #mruby コンポーネント
    |         |-- component.mk           #ビルドのオプション設定
    |         |-- esp32_build_config.rb  #mruby のビルド設定
    |         --- mruby/                 #mruby のソースコード
    |
    |-- Makefile           #ESP32 プロジェクト設定を include した Makefile
    |-- sdkconfig          #ESP-IDF ビルド設定ファイル
    |-- その他ファイル
```

検証対象の評価ボードを接続してシリアル通信できるようにした状態で、mruby-esp32 ディレクトリに移って、以下のようなコマンドを打ちます。

```
make menuconfig
・・・ここでシリアルポートの設定をする
make MRUBY_EXAMPLE=simplest_mrb.rb
make MRUBY_EXAMPLE=simplest_mrb.rb flash monitor
```

これで開発用基板に mruby を含んだバイナリが転送され、リスト 5.2 のように"simplest_mrb.rb"が実行された結果がシリアルモニタに表示されます。

"simplest_mrb.rb"の部分を別のファイルに変更することで、実行する対象のスクリプトを変更することができます。

[*3] 筆者もちょっとだけコントリビュートしています

▼リスト 5.2: simplest_mrb.rb の実行結果

```
1234
4321
```

5.5.2 mruby-esp32 をビルドするための仕組み

mruby-esp32 を使って mruby を ESP32 の上で動かすことはできましたが、ESP32 を活用するのであれば、どのようにして mruby が組み込まれているのかを理解しておきたいところです。mruby のようなライブラリの組み込み方を mruby-esp32 の実装を参照に勉強してみましょう。

5.5.2.1 ESP32 のコンポーネントのビルドについて

ESP-IDF では、ビルド時の動作の指示は、"component.mk"ファイルで行います。あるコンポーネントに対して一つの、"component.mk"がセットになってビルドを行うイメージです。

詳しい説明は以下を参照ください。

`https://docs.espressif.com/projects/esp-idf/en/v3.2.2/api-guides/build-system.html`

5.5.2.2 main のビルド

main もビルド時はコンポーネントと似たような存在なので、main の方から覗いてみましょう。main/component.mk の中身はリスト 5.3 のようになっています。

▼リスト 5.3: main/componet.mk の中身

```
COMPONENT_DEPENDS = mruby_component

COMPONENT_EXTRA_CLEAN := example_mrb.h

mruby_main.o: example_mrb.h

example_mrb.h: $(COMPONENT_PATH)/examples/$(MRUBY_EXAMPLE)
        $(COMPONENT_PATH)/../components/mruby_component/mruby/bin/\
mrbc -B example_mrb -o $@ $^

.PHONY: example_mrb.h
```

ぱっと見た感じでは makefile と同じように見えますが、色々と省略されています。もしファイルの中身が空っぽの場合は、以下のような動作になります。

- 同じディレクトリに存在する (*.c、*.cpp、*.cc、*.S) ファイルがコンパイル対象になる。
- include ディレクトリがあれば、そこに配置されたヘッダファイルはコンパイル時の参照対象になる。
- コンパイルされたオブジェクトはアプリケーションに自動的にリンクされる。

この基本動作に対して足りない処理を付け足していくイメージです。例の中ではいくつか見かけない変数が使用されています。これらはビルド時に自動的に設定されるもので、変数の内容を置き換えることでビルドの動きを変えることもできます。main/componet.mk で使用されている変数を以下に挙げます。

COMPONENT_DEPENDS

依存関係のあるコンポーネントを指定します。"components"ディレクトリに配置したコン

ポーネントのディレクトリ名が対応します。この例では components/mruby_component を参照することを示しています。

COMPONENT_EXTRA_CLEAN

make clean したときに削除する対象ファイルを追加します。

COMPONENT_PATH

コンポーネントの絶対パスを示します。この例では/Users/USER_NAME/YOUR_WORKING_DIRECTORY/mruby-esp32/main が入っており、相対パスから mruby のバイトコードコンパイラ (mrbc) を参照しています。

この main での component.mk の主な目的は、mrbc -B example_mrb -o $@ $^ という部分で、main/examples/以下の.rb ファイルをコンパイルして、example_mrb.h というファイルに C ヘッダ形式のバイトコードを出力することにあります。

このように component.mk を編集することで、ビルド工程に手を入れることが可能になります。

5.5.2.3 components のビルド

components のビルドも基本的には main と同じような流れで行われます。main との違いとしては、components ディレクトリの配下に複数のコンポーネントのディレクトリが並べられるという点です。

mruby_component の componet.mk の内容は以下のとおりです。

```
COMPONENT_OWNBUILDTARGET := true
COMPONENT_OWNCLEANTARGET := true

COMPONENT_ADD_INCLUDEDIRS := mruby/include/

CFLAGS += -Wno-char-subscripts -Wno-pointer-sign

build:
        cd $(COMPONENT_PATH)/mruby && MRUBY_CONFIG=../esp32_build_config.rb $(MAKE)
        cp $(COMPONENT_PATH)/mruby/build/esp32/lib/libmruby.a $(COMPONENT_LIBRARY)

clean:
        cd $(COMPONENT_PATH)/mruby && MRUBY_CONFIG=../esp32_build_config.rb \
$(MAKE) clean
```

"COMPONENT_OWNBUILDTARGET := true "とすることで、ビルドのデフォルト動作をキャンセルしています。代わりに"MRUBY_CONFIG=../esp32_build_config.rb $(MAKE)"で、mruby のビルドを実行しています。mruby のビルドでは、MRUBY_CONFIG という環境変数を使って、mruby 標準の"build_config.rb"の代わりに、”esp32_build_config.rb”を指定しています。そしてビルドの結果、出力された"libmruby.a"を、$(COMPONENT_LIBRARY) にコピーして、ビルド時にリンクされるようにしています。

esp32_build_config.rb のポイントは、MRuby::CrossBuild.new(’esp32’) do |conf| end というブロックで、クロスコンパイルの設定をしている箇所です。リスト 5.4 に抜粋して示しています。

- ツールチェインの設定
  - コンパイル時は変数 CC にセットされた xtensa のコンパイラを使うので、toolchain には":gcc"をセットしています。

- インクルードするヘッダのパス設定
  - 環境変数 COMPONENT_INCLUDES にセットされたパスを、インクルード対象としてセットしています。
- mruby のチューニング
  - mruby が確保するリソースをデフォルトよりも少なめにするために、cc.defines でパラメータを定義しています。各パラメータの意味は mrbconf.h"に記載があります。esp32_build_config.rb で使用しているものについては、リスト 5.4 にコメントを加えています。
- mrbgem
  - mruby の機能を拡張するために、mruby は mrbgem という機能を持っています。これを使うことで、ローカルや github 等からソースコードを取得して、mruby のビルド時にリンクすることが可能になります。組み込む対象の mrbgem は、conf.gem に対して設定しています。

▼リスト 5.4: esp32_build_config.rb のクロスコンパイル設定

```
MRuby::CrossBuild.new('esp32') do |conf|
  toolchain :gcc

  conf.cc do |cc|
    cc.include_paths << ENV["COMPONENT_INCLUDES"].split(' ')

    cc.flags << '-Wno-maybe-uninitialized'
    cc.flags.collect! { |x| x.gsub('-MP', '') }

    #筆者注：ヒープ１ページに格納するオブジェクト数
    cc.defines << %w(MRB_HEAP_PAGE_SIZE=64)

    #筆者注：インスタンス変数の管理方法の指定。
    #       mruby2.0 ではセグメントリストに統一されたので削除してもよいはず。
    cc.defines << %w(MRB_USE_IV_SEGLIST)

    #筆者注：khash のデフォルトサイズの設定
    cc.defines << %w(KHASH_DEFAULT_SIZE=8)

    #筆者注：文字列バッファの最小サイズ
    cc.defines << %w(MRB_STR_BUF_MIN_SIZE=20)

    #筆者注：メモリをすぐ回収するために、GC をメモリ確保時に毎回実行
    cc.defines << %w(MRB_GC_STRESS)

    cc.defines << %w(ESP_PLATFORM)
  end

  conf.cxx do |cxx|
    cxx.include_paths = conf.cc.include_paths.dup

    cxx.flags.collect! { |x| x.gsub('-MP', '') }

    cxx.defines = conf.cc.defines.dup
  end

  conf.bins = []
  conf.build_mrbtest_lib_only
  conf.disable_cxx_exception #筆者注：C++ の例外を無効化
```

```
  #筆者注：puts や print メソッドを有効にする
  conf.gem :core => "mruby-print"

  #筆者注：mrbc をビルドする
  conf.gem :core => "mruby-compiler"

  #筆者注：ESP32 のシステム関連機能の API
  conf.gem :github => "mruby-esp32/mruby-esp32-system"

  #筆者注：ESP32 の WiFi 関連の API
  conf.gem :github => "mruby-esp32/mruby-esp32-wifi"
end
```

このようにクロスコンパイルの設定を行うことで、ESP32 向けに mruby のビルドが行われます。

5.5.3 mruby を C 言語実装から実行する方法

mruby は mruby_main.c 内で実行されています。mruby_main.c はこのアプリケーションのエントリポイントとなるファイルです。mruby_main.c の実装を眺めながら、mruby を C 言語から呼び出す方法を確認してみましょう。

▼リスト 5.5: mruby_main.c の実装

```
 1: #include <stdio.h>
 2: （中略）
 3: #include "mruby.h"
 4: （中略）
 5: #include "example_mrb.h"
 6: （中略）
 7:
 8: void mruby_task(void *pvParameter)
 9: {
10:   mrb_state *mrb = mrb_open();
11:   (中略)
12:   mrbc_context *context = mrbc_context_new(mrb);
13:   int ai = mrb_gc_arena_save(mrb);
14:   mrb_load_irep_cxt(mrb, example_mrb, context);
15:    （中略）
16:   mrb_gc_arena_restore(mrb, ai);
17:   mrbc_context_free(mrb, context);
18:   mrb_close(mrb);
19:    （中略）
20:   while (1) {
21:   }
22: }
23:
24: void app_main()
25: {
26:   nvs_flash_init();
27:   xTaskCreate(&mruby_task, "mruby_task", 8192, NULL, 5, NULL);
28: }
```

リスト 5.5 に mruby_main.c のコードを示しました。実装を見てみると、main 関数が無いことに気づきます。main 関数は ESP-IDF が準備しており、ユーザアプリケーションは app_main 関数がエントリポイントとなります。

app_main 関数の中では、nvs_flash_init 関数で nvs(non volatile storage) 領域という設定値

などを書き込む領域を初期化したあと、xTaskCreate 関数で FreeRTOS のタスクを生成しています。

ESP-IDF で開発するアプリケーションは FreeRTOS というリアルタイム OS がベースとなっています。RTOS はリアルタイム OS の意味です。RTOS 一般的に Linux などで言うところのスレッドに相当するタスクを厳密なタイミングでスケジューリングすることができるように設計されており、XX msec の周期で確実に処理を行わないといけない、といった要求に応えないといけない比較的小規模なプロセッサ上で使用されています。

タスクについては、POSIX スレッドに馴染みがあれば、まずはスレッドと同じようなイメージで捉えれば OK です。xTaskCreate 関数では、mruby_task 関数を"mruby_task"という名前のタスクとして生成して実行しています。

mruby_task 関数では、mruby のバイトコードを読み込み、実行する処理を行っています。

以下に使用されている関数の意味をまとめています。

mrb_open()

mruby の処理環境のセットアップを行い、`mrb_state` という構造体のポインタを返します。`mrb_state` には mruby で使用するすべての状態やメモリが格納されています。`mrb_state` のポインタは mrb という変数名であることを前提に、mruby VM の中でも各所で参照されています。

mrbc_context_new()

`mrbc_context` 構造体のポインタを返します。通常 `mrbc_context` には mruby の実行情報が格納されますが、VM 実行時の挙動を調整するためにも使用できます、

mrb_gc_arena_save()

C 言語での Ruby オブジェクトを生成する際に、GC の挙動を調整するために使用しています。

mrb_load_irep_cxt()

バイトコードを読み込んで実行します。この例では、example_mrb.h に定義されている char 型の配列である example_mrb を読み込んで実行します。

mrb_gc_arena_restore()

`mrb_gc_arena_save()` の対となる関数です。

mrbc_context_free()

`mrbc_context` の開放処理を行います。

mrb_close()

`mrb_state` の開放処理を行います。

example_mrb.h には、main をビルドする際に mrbc によってコンパイルされた結果のバイトコードのバイト列の配列が以下のような形で格納されています。

```
/* dumped in little endian order.
   use `mrbc -E` option for big endian CPU. */
#include <stdint.h>
extern const uint8_t example_mrb[];
const uint8_t
#if defined __GNUC__
__attribute__((aligned(4)))
#elif defined _MSC_VER
__declspec(align(4))
#endif
```

```
example_mrb[] = {
0x45,0x54,0x49,0x52,0x30,0x30,0x30,0x36,0x08,0x68,0x00,0x00,0x00,0x6a,0x4d,0x41,
(中略)
};
```

5.6 ESP32 上での mruby 環境開発

ESP32 上で mruby 環境を構築する際に気をつけるべき点について説明します。

5.6.1 メモリの管理

ESP32 は使用可能なメモリが比較的多いとはいえ、PC と比べたら雀の涙ほどしかありません。メモリの管理はとても重要です。PC 用の OS と無理やり比較してみるならば、メモリ空間の高度な管理機能を搭載していないようなマイコンでは、カーネルもユーザランドも同じメモリ空間に存在しているようなものです。DMA（Direct Memory Access）のようなハードウェアが直接メモリを操作するような機能を利用する場合、メモリの番地にも意味があり、注意が必要です。ESP32 では開発環境が提供する関数を利用して、番地などは意識しなくても済むようになっています。

ESP32 は内蔵しているメモリは 512KB しかありません。しかも FabGL を使用した場合、画面出力用のバッファなどのコアの機能で大部分を使い切ってしまうので、mruby で使用できる分は多くはありません。そのため、ESP32-WROVER-B の特徴である、SPI-PSRAM を活用します。こちらは SPI によってアクセスするので、内蔵メモリよりはアクセス速度に劣りますが、大容量を生かして mruby の活用範囲を広げることができます。

5.6.1.1 メモリアロケータ

mruby ではメモリ管理の事情が環境によって大きく異なることを想定して、メモリアロケータの関数を独自に指定することができます。メモリアロケータを指定する場合、`mrb_open()` の代わりに `mrb_open_allocf()` を使用します。`mrb_open_allocf()` は、引数に `mrb_allocf` という型のメモリの取得開放を行う関数を設定できます。

`mrb_allocf` が mruby が使用する全てのヒープメモリの確保と開放を行います。機能としては `realloc()` と同じような機能を提供することが求められています。

デフォルトは以下のように `free()` と `realloc()` が利用されています。

▼リスト 5.6: デフォルトのメモリアロケータ

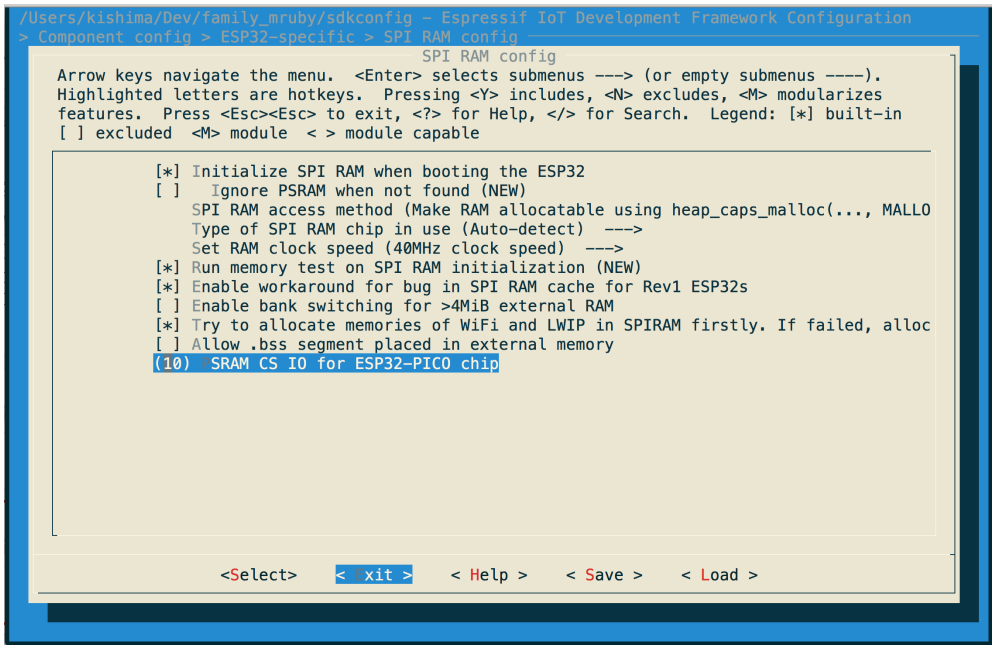
```
void*
mrb_default_allocf(mrb_state *mrb, void *p, size_t size, void *ud)
{
  if (size == 0) {
    free(p);
    return NULL;
  }
  else {
    return realloc(p, size);
  }
}
```

ESP32 では `realloc()` と `free()` はサポートされていますが、SPI-PSRAM を利用するために、`heap_caps_realloc()` を `MALLOC_CAP_SPIRAM` というオプション付きで利用します。

▼リスト 5.7: ESP32-WROVER-B 用のメモリアロケータ

```
void*
mrb_esp32_psram_allocf(mrb_state *mrb, void *p, size_t size, void *ud)
{
  if (size == 0) {
    free(p);
    return NULL;
  }
  else {
    return heap_caps_realloc(p, size, MALLOC_CAP_SPIRAM);
  }
}
```

またビルド設定も変更が必要です。設定画面から、[Component config][ESP32-specific] に入り、[Support for external, SPI-connected RAM] にチェックを入れて、[SPI RAM config] を図 5.5 のように設定します。



▲図 5.5: SPI-PSRAM の設定例

SPI-PSRAM の設定については、まだ十分検証できていないので、今後も確認を進めていくつもりです。

【コラム】ESP32 のメモリ確保方法のオプション

ESP-IDF では、heap_caps_malloc()/heap_caps_realloc() を利用したヒープメモリの確保において、いくつかの方法を選択することができます。開放するときは、確保方法に関わらず free() を用います。

- MALLOC_CAP_8BIT
  - 8 ビットアライメントでメモリを確保する。
- MALLOC_CAP_32BIT
  - 32 ビットアラインメントでメモリを確保する。
- MALLOC_CAP_DMA
  - DMA（Direct Memory Access）でアクセス可能な領域にメモリを確保する。FabGL の映像、音声のバッファで使用されています。
- MALLOC_CAP_INTERNAL
  - ESP32 内部のメモリを確保する
- MALLOC_CAP_SPIRAM：
  - SPI-PSRAM 上でメモリを確保する

https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/mem_alloc.html 参照

第6章　mrubyを自分のデバイス色に染める

前章では、mruby を ESP32 の上で動かすための準備をしてきました。

でも本番はここからです。mruby はそれだけでは現実世界との接点を何ももっていません。ハードウェアを制御するためには、誰かが mruby との仲介をしてあげる必要があります。もしかするとすでに誰かがやってくれたオープンソースのライブラリを利用できるかもしれませんが、必ずしもそれが期待できるとは限りません。

本章では、mruby に自分のデバイス上で動く手足を生やして、mruby からデバイスを自由に制御できるようにするための方法を解説します。

6.1　mruby とハードウェア

Ruby に限らず一般的な PC 上で利用するプログラミング言語の場合は、Linux や、MacOS、Windows などの一般的な OS 上で動かすことを前提としています。

mruby は"stdio.h"も使えないようなフリースタンディング環境で使用することも想定しているので、最小構成の場合、純粋に Ruby のコードを読んで計算をするだけで、文字の出力も何もできません。

一般的な OS 上で動くアプリに組み込む際はあまり気にする必要はないですが、一般的ではないハードウェア上で mruby を動かすためには、ハードウェアとの接続部分は自分でケアする必要があります。

一般的ではないハードウェアと言っても、大抵は供給元の会社やユーザから何らかの C 言語や C++ のライブラリが提供されていると思います。mruby の C 言語拡張としてそれらのライブラリと mruby のクラスやメソッドを繋ぐことで、mruby からそのハードウェアの機能を利用できるようなります。

これまで説明してきた Narya ボードの場合の設計方針を説明していきたいと思います。

6.2　mruby と HMI

ゲーム用のデバイスなので、ここが一番ポイントと言ってよいと思います。通常ならば RasbpberryPi のように標準で映像出力機能を搭載しているようなチップを利用するところですが、開発環境も Linux が前提となったり、ちょっと大げさになりがちです。

映像出力、キーボード入力、音声出力の機能を提供しているのが、FabGL という ESP32 Arduino 環境向けのライブラリです。ESP32 は通信機能がメインの汎用のマイコンであり、映像出力専用の機能を持っていないため、映像出力はソフトウェアと I2S というハードウェアの機能を利用して実現しています。これがなかなかすごいです。

汎用ポートを利用したモニタへの映像出力を実現した事例として、筆者の知っているものを以下

に上げてみます。

- IchigoJam(株式会社 jig.jp より発売)
  - LPC1114 で、NTSC ビデオ出力（296 × 216px、2 色）を実現しています。
    `https://ichigojam.net/`
- MachiKania
  - PIC32 で、NTSC ビデオ出力（最大 384 × 216px、256 色）を実現しています。
    `http://www.ze.em-net.ne.jp/~kenken/machikania/`
- Uzebox
  - ATmega644 と RGB/NTSC 変換を行う AD725 を利用して、NTSC ビデオ出力（最大 240x224px、256 色）を実現しています。
    `http://belogic.com/uzebox/index.asp`
- arduino-tvout
  - Arduino Uno その他の Arduino シリーズで、NTSC ビデオ出力（128x96px、2 色）を実現しています。
    `https://code.google.com/archive/p/arduino-tvout/`
- ESP32Lib
  - bitluniESP32 で、VGA ビデオ出力（最大 800x600px、16384 色 (14bit)）を実現しています。
    `https://github.com/bitluni/ESP32Lib`
- FabGL
  - ESP32 で、VGA ビデオ出力（最大 800x600px、64 色 (6bit)）を実現しています。
    `https://github.com/fdivitto/fabgl`

Narya ボードでは前述の通り、FabGL を採用しています。色数だけでみると ESP32lib も良さそうに見えますが、14bit カラーの実現のために 14 本の GPIO を使用しているため、キーボードや音声その他の機能に使える IO がほとんど残っていません。そのため FabGL を利用しています。

6.2.1 FabGL とは

FabGL は Fabrizio Di Vittorio という方が開発されている ESP32 向けの HMI ライブラリです。以下のような機能を提供しています。

- VGA ビデオ出力
- PS/2 キーボード＆マウス入力
- DAC 音声出力
- 描画ライブラリ（スプライト含む）
- GUI ライブラリ
- ANSI/VT ターミナルライブラリ

筆者が必要としている機能を完璧に提供してくれていて、これを見つけたときは運命かな？ と思ったほどです。これらは ArduinoIDE 開発環境向けのライブラリとして提供されています。

6.2.2 mruby への FabGL の組み込み

FabGL は ArduinoIDE 開発環境向けのライブラリであると触れましたが、これを mruby と連動させるには以下のような点が課題となります。

- mruby を ArduinoIDE でビルドするのが難しい
  - ArduinoIDE はビルド方法詳細が隠蔽されており、minirake 使用が前提の mruby をビルドするには色々と修正が必要になる
- FabGL は Arduino ESP32 のライブラリに依存している
  - FabGL は Arduino 向けの ESP32 API を利用しているので、ESP-IDF にコピーしてそのままでビルドすることはできない
- FabGL は C++ で書かれている
  - 大きな問題ではないですが、mruby は C 言語で書かれているのに対して、FabGL は C++ で書かれているので、コンパイル時のオプションに注意が必要です

以下のような方針で対応することにします。

- mruby を ArduinoIDE でビルドするのが難しい
  - FabGL の規模が mruby と比較して小さいこともあり、ESP-IDF でビルドする前提で進めることにします。
- FabGL は Arduino ESP32 のライブラリに依存している
  - ESP-IDF も各種 API を提供していますが、Arduino 向けの API とは異なっています。そのため、ESP-IDF ではビルドするために、ESP-IDF に Arudino 用のライブラリをコンポーネントとして導入します。
- FabGL は C++ で書かれている
  - mruby のビルド設定に若干のオプションを加えることでビルドできるようになります。

6.2.3 ESP-IDF で Arudino 向けのコードをビルドする

Arduino 向けのコードを ESP-IDF でビルドするには、"Arduino core for the ESP32"というコンポーネント[*1]を利用します。

以下のようにリポジトリを git clone して配置します。他のコンポーネントからも参照するので、コンポーネント名は"arduino"で固定します。

```
mruby-esp32/
    |-- main/
    |-- components/
    |   |-- mruby_component/   #mruby コンポーネント
    |   |-- arduino/           #Arduino core for the ESP32 コンポーネント
    |
    |-- その他ファイル
```

このように配置すると、`make menuconfig` を実行した際に、Arduino 関連のメニューが増え、`Arduino.h` ヘッダファイル等を参照できるようになります。設定項目が多数あり、ArduinoIDE でビルドしたときと同じ設定を再現するのがなかなか難しいです。そのため今回は、ArduinoIDE で ESP32 のボード設定を読み込みこんだ結果の sdkconfig ファイルをコピーして参照しました。

例えば、筆者の環境の MacOS の場合、sdkconfig は以下に配置されていました。

`~/Library/Arduino15/packages/esp32/hardware/esp32/1.0.2/tools/sdk/sdkconfig`

コンポーネントを配置した状態でビルドすると、Arduino と同様に `setup()` と `loop()` が自動的に呼び出されます。ArduinoIDE との違いは、メインのソースコードの拡張子が".ino"ではなく

[*1] `https://github.com/espressif/arduino-esp32`

て、".cpp"となる点です。

6.2.4 FabGL のコンポーネント化

FabGL を組み込む際に、アプリのコードにコピーして組み込んでも大丈夫ですが、将来のバグフィックスのマージ等のためにコンポーネントとして独立させたいと思います。

FabGL コンポーネントは以下のように配置します。

```
mruby-esp32/
    |-- main/
    |-- components/
    |   |-- mruby_component/   #mruby コンポーネント
    |   |-- arduino/           #Arduino core for the ESP32 コンポーネント
    |   |---FabGL_component/   #FabGL コンポーネント
    |
    |-- その他ファイル
```

Arduino のライブラリである FabGL をコンポーネントするためのポイントを以下に挙げます。ソースコード本体には手を加えずに、コンパイル設定に手を入れるだけでいけます。

- 不要なファイルの削除
  - ビルドに必要なのは、src と tools フォルダのファイルです。
- component.mk の追加
  - 以下のような内容の component.mk を追加します

```
SDKPATH = $(COMPONENT_PATH)/../arduino/tools/sdk/include
COREPATH = $(COMPONENT_PATH)/../arduino/cores

COMPONENT_EXTRA_INCLUDES := $(SDKPATH)/esp32 $(SDKPATH)/driver $(SDKPATH)/soc
$(SDKPATH)/freertos $(PROJECT_PATH)/build/include $(SDKPATH)/log $(SDKPATH)/ulp
$(SDKPATH)/esp_adc_cal $(SDKPATH)/heap $(COREPATH)/esp32
$(COMPONENT_PATH)/../arduino/variants/esp32

COMPONENT_SRCDIRS := src

CPPFLAGS += -std=gnu++11 -fno-rtti
```

FabGL が参照している Arduino 固有のヘッダファイルが見えるように `COMPONENT_EXTRA_INCLUDES` に相対パスを追加します。

最新の FabGL コンポーネントは github（`https://github.com/kishima/FabGL_component`）で管理しています。Arduino core for the ESP32 コンポーネントと組み合わせることで、ESP-IDF で FabGL を使ったアプリケーションが作れるようになります。

これでビルドに最低限必要なソースコードが揃いました。

6.3 mrbgems による機能拡張

ここからが mruby のカスタマイズの本番です。mruby に独自のライブラリを追加するには、mrbgem という機能を使用します。mruby はデフォルトでは、`require` といった外部のソースコードを動的に読み込む機能を持っていないので、通常は mruby が起動した時点ですべてのライ

ブラリは VM に読み込まれた状態であることが前提となります。

6.3.1 mrbgem とは

前章でも少し触れましたが、mruby には、Ruby における RubyGem のように、パッケージを取り扱うための機能として、mrbgem という機能があります。mrbgem は、Ruby で書かれたコードもしくは C 言語拡張として書かれたコードをパッケージ化して、mruby のビルド時に取り込むための機能です。

ライブラリとして用いるようなコードは mrbgem として実装しておくのがよいと思います。

広く使ってほしい mrbgem は `https://github.com/mruby/mgem-list` に必要な情報を添えて pull request を送るとよいと思います。（本書では詳細は説明しません）

6.3.2 mrbgem の作り方

mrbgem を作ることはそんなに難しくはありませんので、順を追って作り方を理解していきましょう。

6.3.2.1 必要なファイルと配置

C 言語拡張と Ruby のコードで構成される mrbgem を作るために必要なファイルとその構造は以下のとおりです。ホスト環境で動作することを想定している場合、test ディレクトリも作成してテストコードも登録するのがよいと思います。クロスコンパイル前提の場合は、スタブ等の準備なしにホスト環境でテストを行うことができないので、以下の例では test ディレクトリを除外しています。

```
mrbgem 名/
  |- mrblib/
  |   |- (Ruby のコード)
  |
  |- src/
  |   |- (C のコードとヘッダファイル)
  |
  |- mrbgem.rake
```

mrbgem.rake の中身の例は、リスト 6.1 のようになっています。`"spec.add_dependency 'mrbgem 名'"`という書式で mrbgem 間の依存関係を示すこともできます。ビルドの動作に手を加える必要がなければ、spec の設定だけで問題ありません。

▼リスト 6.1: mrbgem.rake の例

```
MRuby::Gem::Specification.new('mrbgem 名') do |spec|
  spec.license = ' ライセンス名'
  spec.authors = ' 開発者名'
end
```

mrbgem 名は、接頭語として`"mruby-"`を付けるのが慣例のようです。

6.3.2.2 C++ のソースコードをビルドするための注意点

FabGL をビルドするためには、C++ のコードをコンパイルする必要があります。mrbgem では*.cpp のファイルも自動でビルド対象になりますが、コンパイル時にエラーが出ないように一部編集する必要がありました。

現在の mruby では、.cpp が混じっている mrbgem は.c のファイルも C++ のソースとしてコンパイルされるようです。そのため、`extern "c"{}`の代わりに、`MRB_BEGIN_DECL` と `MRB_END_DECL` を使用します。

また、esp32_build_config.rb のコンパイル設定で、`conf.enable_cxx_exception` としています。例外処理だけに影響しているように見えますが、C++ のソースビルド全体に影響を及ぼしているようです。

6.3.3 最低限実装必要な関数

C 言語拡張を実装した場合、mruby は自動的に所定の関数を VM の起動時と終了時に呼び出します。最低限以下の関数を実装する必要があります。

- `void mrb_[mrbgem 名]_gem_init(mrb_state *mrb)`
  - VM 起動時、アプリのコードが実行される前に呼び出されます。C 言語によるクラスやメソッドの定義が必要であれば、ここで行います。
- `void mrb_[mrbgem 名]_gem_final(mrb_state *mrb)`
  - VM 終了時に呼び出されます。

注：mrbgem 名に"-"が含まれる場合、関数名上は"_"に自動変換されます。

6.3.4 クラスや、メソッドの定義

Ruby の `class` や `def` に相当する処理を C 言語でも実装することができます。その方法を確認してみましょう。

C 言語拡張の実装で使用できる主要な関数は、"mruby.h"に定義されているので、このヘッダを自分の mrbgem のソースコードにインクルードして、使用します。その他のヘッダの関数も使用可能ですが、VM の動作に直結するような機能は VM の動作を十分把握した上で、使用するべきでしょう。

6.3.4.1 クラスの定義

クラスを定義するには以下の関数を用います。

通常のクラス定義

```
struct RClass *mrb_define_class
  (mrb_state *mrb, const char *name, struct RClass *super);
```

- `mrb`：mruby の実行状態
- `name`：定義するクラス名
- `super`：親クラスのクラスオブジェクトへのポインタ
- 戻り値：生成されたクラスオブジェクトへのポインタ

mrb は mruby の VM の実行状態を全て格納している構造体なので、実行状態に影響するような処理を行う関数には無条件で引き渡すべき変数です。super については、特に特別なクラスを継承する必要がない場合は、オブジェクトクラスを設定します。Object クラスオブジェクトへのポインタは、mrb_state 構造体から `mrb->object` のように参照できます。Object クラス以外の基本的なクラスやモジュールへのポインタも定義されているので、"mruby.h"の内容を参照ください。

モジュール内のクラス定義

モジュールに属するクラスを定義する場合は以下の関数を利用します。

```
struct RClass * mrb_define_class_under
  (mrb_state *mrb, struct RClass *outer, const char *name, struct RClass *super);
```

- mrb：mruby の実行状態
- outer：定義するクラスが属するモジュールへのポインタ
- name：定義するクラス名
- super：親クラスのクラスオブジェクトへのポインタ
- 戻り値：生成されたクラスオブジェクトへのポインタ

6.3.4.2 メソッドの定義

メソッドの定義には以下の関数を使用します。

```
void mrb_define_method
  (mrb_state *mrb, struct RClass *cla, const char *name,
   mrb_func_t func, mrb_aspec aspec);
```

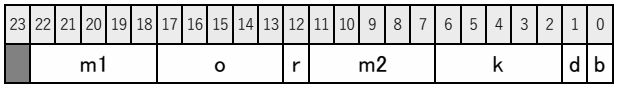
- mrb：mruby の実行状態
- cla：メソッドが属するクラスオブジェクトへのポインタ
- name：定義するメソッド名
- func：メソッド実体への関数ポインタ
- aspec：メソッド引数の定義

func には、メソッドの処理を行う mrb_func_t 型の関数ポインタを設定します。mrb_func_t の定義は以下のとおりです。

```
typedef mrb_value (*mrb_func_t)(struct mrb_state *mrb, mrb_value);
```

mrb_value には、レシーバのオブジェクトが設定されます。mruby のメソッドは基本的にこの関数で表現することができます。

aspec には、引数のパターンをビットで示します。例えば引数の数が 3 つのメソッドの場合、MRB_ARGS_REQ(3) というようなマクロを使って表現します。その他、キーワード引数、rest 引数、ブロック引数等の表現もあります。図 6.1 に各ビットの意味を示しています。



m1 ： 必須引数の数(mandatory)
o ： 初期値付き引数の数(option)
r ： rest引数の有無(rest)
m2 ： 後置必須引数の数(mandatory2)
k ： キーワード引数の数(keyword)
d ： 末尾のハッシュ引数有無(dictionary)
b ： ブロック引数の有無(block)（未参照）

▲図 6.1: aspec のビットの意味

6.3.4.3 その他の定義関連の関数

定義に関する主な関数を以下に挙げます。詳細は"mruby.h"を参照ください。

- モジュールの定義
  - `mrb_define_module()`
- クラスメソッドの定義
  - `mrb_define_class_method()`
- モジュール関数の定義
  - `mrb_define_module_function()`
- 定数の定義
  - `mrb_define_const()`

6.3.5 mruby のオブジェクトの C 言語での表現

Ruby ではすべてのクラスは Object クラスを継承しているので、オブジェクトはすべて Object クラスを継承したクラスのインスタンスとして表現することができます。そのため汎用のオブジェクトは、C 言語でも Object クラスをベースとした実装になっているかと推測できますが、実際はリスト 6.2 に示すように、muby の汎用オブジェクトは mrb_value という構造体として表現されています。

C 言語拡張においても、メソッドの引数や戻り値などで頻繁に扱うことになります。

▼リスト 6.2: mrb_value 構造体

```
typedef struct mrb_value {
  union {
#ifndef MRB_WITHOUT_FLOAT
    mrb_float f;
#endif
    void *p;
    mrb_int i;
    mrb_sym sym;
  } value;
  enum mrb_vtype tt;
} mrb_value;
```

mrb_value では、表 6.1 のような種別のオブジェクトを格納しています。Ruby のオブジェクトだけではなく、ユーザからは見えない VM の内部処理のための情報も格納しています。これは、クラス名の探索などを行わず、よく使うクラスをフラグで区別できるようにして、処理を高速化するためと思われます。

mrb_value を操作するための関数は多数用意されているので、代表的なものの使い方を後述します。

▼表 6.1: mrb_vtype の種別

種別名	意味
MRB_TT_FALSE	false オブジェクト
MRB_TT_FREE	内部用：メモリ開放したオブジェクトにセット
MRB_TT_TRUE	true オブジェクト
MRB_TT_FIXNUM	Fixnum オブジェクト
MRB_TT_SYMBOL	Symbol オブジェクト
MRB_TT_UNDEF	内部用：未定義の意味
MRB_TT_FLOAT	Float オブジェクト
MRB_TT_CPTR	文字列オブジェクト
MRB_TT_OBJECT	その他もろもろの一般オブジェクト
MRB_TT_CLASS	Class オブジェクト
MRB_TT_MODULE	Module オブジェクト
MRB_TT_ICLASS	内部用：Mixedin 管理用
MRB_TT_SCLASS	Singleton クラスオブジェクト
MRB_TT_PROC	Proc オブジェクト
MRB_TT_ARRAY	Array オブジェクト
MRB_TT_HASH	Hash オブジェクト
MRB_TT_STRING	String オブジェクト
MRB_TT_RANGE	Range オブジェクト
MRB_TT_EXCEPTION	Exception オブジェクト
MRB_TT_FILE	File クラスオブジェクト？
MRB_TT_ENV	内部用：REnv 用
MRB_TT_DATA	内部用：C 言語のポインタを扱うために使用
MRB_TT_FIBER	Fiber クラスオブジェクト
MRB_TT_ISTRUCT	内部用：Inline Structure
MRB_TT_BREAK	内部用：RBreak 用

mrb_value に関する注意点として、C 言語で書かれたメソッドの戻り値は mrb_value 構造体のポインタではなく、mrb_value 構造体の実体であるという点があります。mrb_value には、Fixnum、Float、Symbol のように mrb_value の中に値を直接格納できるタイプと、mrb_value 構造体にオブジェクトの実態を指すポインタを格納するタイプの 2 種類があることを覚えておくと、GC 中のメモリ解放で実際にどんなことが起きているのか理解がしやすくなると思います。

6.3.6 メソッド引数

Ruby は柔軟な引数の受け取り方が可能ですが、それを C 言語の関数でも受け取ることができるように、`mrb_int mrb_get_args(mrb_state *mrb, mrb_args_format format, ...);` と

いう関数を用いて、メソッドの引数を参照します。
例えば、文字列を受け取る場合は、次のように行います。

```
mrb_value string;
mrb_get_args(mrb,"S",&string);
```

表 6.2 には、mrb_get_args 関数のフォーマットを示しています。

▼表 6.2: mrb_get_args 関数のフォーマット

文字	Ruby の型/種別	C の型	コメント
‘o‘	Object	mrb_value	すべての型の引数の参照に使用できる
‘C‘	Class/Module	mrb_value	
‘S‘	String	mrb_value	‘¡が続く場合、‘nil‘が与えられる可能性がある
‘A‘	Array	mrb_value	‘¡が続く場合、‘nil‘が与えられる可能性がある
‘H‘	Hash	mrb_value	‘¡が続く場合、‘nil‘が与えられる可能性がある
‘s‘	String	char *, mrb_int	２つの変数で受け取る。 ‘s¡は ‘nil‘の場合 (‘NULL‘,‘0‘) を与える
‘z‘	String	char *	‘NULL‘終端の文字列を受け取る。 ‘z¡は ‘nil‘の場合に ‘NULL‘を与える
‘a‘	Array	mrb_value *, mrb_int	２つの変数で受け取る。 ‘a¡ は ‘nil‘の場合、(‘NULL‘,‘0‘) を与える
‘f‘	Float	mrb_float	
‘i‘	Integer	mrb_int	
‘b‘	ブール値	mrb_bool	
‘n‘	Symbol	mrb_sym	
‘&‘	ブロック	mrb_value	‘&¡ はブロックが与えられていない場合、 例外を発生させる
‘*‘	残りの引数	mrb_value *, mrb_int	残りの引数を配列として受け取る。 *! はスタックのコピーを回避する
‘|‘	オプション		‘|‘に続くフォーマット文字はオプションである。
‘¿	オプション有無	mrb_bool	先行する引数がある場合、‘TRUE‘となる。 オプション引数の有無の確認に用いられる

6.3.7 よく使う API

よく使うクラスの API の例を以下に挙げています。他にも多数存在しているので、ヘッダファイルを確認してみてください。関数名からだいたいの機能は推測がつくかと思います。

6.3.7.1 mrb_int

mruby の C 言語実装では整数型は mrb_int で表現されています。ビット長は MRB_INT64、MRB_INT32、MRB_INT16 のいずれかをビルド時のオプションで宣言することで選択できます。

6.3.7.2 Fixnum,true,false,nil

mrb_value は直値でいくつかのオブジェクトを表現することができます。オブジェクトを生成する関数を以下に示します。内部ではヒープメモリを新たに確保することなく、mrb_value 構造体に所定の値をセットしています。

- `mrb_fixnum_value(mrb_int i)`
  - mrb_int を元に Fixnum のオブジェクトを返します。
- `mrb_value mrb_float_value(struct mrb_state *mrb, mrb_float f)`
  - mrb_float を元に Float のオブジェクトを返します。
- `mrb_value mrb_true_value(void)`
  - true オブジェクトを返します。
- `mrb_value mrb_false_value(void)`
  - false オブジェクトを返します。
- `mrb_value mrb_nil_value(void)`
  - nil オブジェクトを返します。

6.3.7.3 String

文字列に関する関数は、`mruby.h` の他に、ヘッダ `mruby/string.h` に定義されています。

- `mrb_value mrb_str_new_cstr(mrb_state*, const char* str)`
  - C の文字列 str を Ruby の String オブジェクトに変換した結果を返します。
- `mrb_value mrb_str_new_static(mrb_state*, const char* str)`
  - `mrb_str_new_cstr()` と似ていますが、str のポインタがそのまま String オブジェクト内部に保持されます。
- `void mrb_str_concat(mrb_state*, mrb_value str1, mrb_value str2)`
  - str1 に str2 を連結します。

6.3.7.4 Array

配列に関する関数は、ヘッダ `mruby/array.h` に定義されています。

- `void mrb_ary_push(mrb_state *mrb, mrb_value array, mrb_value value)`
  - オブジェクト `value` を、配列 `array` に push します。
- `mrb_value mrb_ary_pop(mrb_state *mrb, mrb_value ary)`
  - 配列 `ary` から pop したオブジェクトを返します。

6.3.7.5 Hash

ハッシュに関する関数は、ヘッダ `mruby/hash.h` に定義されています。

- `void mrb_hash_set(mrb_state *mrb, mrb_value hash, mrb_value key, mrb_value val)`
  - ハッシュ `hash` にキー `key` に対応する値 `val` をセットします。
- `mrb_value mrb_hash_get(mrb_state *mrb, mrb_value hash, mrb_value key)`
  - ハッシュ `hash` からキー `key` に対応する値を取得して返します。

6.3.7.6 Ruby メソッドの呼び出し

C 言語からも Ruby のメソッドを呼び出すことができます。

- `mrb_value mrb_funcall(mrb_state *mrb, mrb_value self, const char`

*name, mrb_int argc, ...)
  - self にメソッドのレシーバーを指定して、name のメソッドを呼び出します。内部では mrb_run() が実行されています。

6.3.7.7 インスタンスの生成

その他のクラスに属するオブジェクトについても、C 言語の関数を用いて、操作することができます。

- mrb_value mrb_obj_new(mrb_state *mrb, struct RClass *c, mrb_int argc, const mrb_value *argv)
  - クラス c のインスタンスを生成します。引数は mrb_value の配列で渡します。initialize のために内部では mrb_funcall_argv() が実行されています。

6.4 ガベージコレクション（GC）

mruby 自身はマルチタスク/マルチスレッド機能をもっていないので、C 言語拡張を実行している最中は VM の他の処理が割り込んできたりはしませんが、mruby オブジェクトを生成するなどしてヒープメモリをアクセスする API を読んだタイミングで、API 内部ではガベージコレクションが発生する可能性があります。

意識しないところで、重い処理が動いている可能性があるので、mrbgem を開発するときには、GC の基本的な動きについてもよく理解しておく必要があります。

6.4.1 mruby のガベージコレクション

mruby のガベージコレクションは世代別インクリメンタルアルゴリズムで実装されています。まつもとさんのコメントによれば、メモリ効率を多少犠牲にしてもソフトウェアリアルタイムを実現するため、GC でロックする時間を一定に近づくように設計されているそうです。

まとまった解説はネット上にはあまり見かけませんが、@syu_cream さんの「mrubook: mruby の実装を探検する薄い本」という kindle 本にわかりやすい解説があるので、気になる方は、そちらを読んでみることをおすすめします。

また C 言語でオブジェクトを生成する場合に考慮する必要があるアリーナの仕組みについては、まつもとさんの記事が参考になります。（英訳したものが mruby にも添付されています）

http://matz.rubyist.net/20130731.html

6.4.1.1 gc.c にある説明書きの翻訳

gc.c には mruby の GC に関する説明書きが英文でコメントされています。オフィシャルのドキュメントとしてはもっとも詳しい説明かもしれません。本書では実装の詳細まで追えていませんが、翻訳だけでも以下に掲載してみます。ある程度筆者の意訳も入っていますので、細かい点はソースコードを参照下さい。

■三色インクリメンタルガベージコレクション

mruby の GC は三色インクリメンタルガベージコレクションとマーク＆スイープを組み合わせたものです。アルゴリズムの詳細は省きますが、実装に関する点を以下で説明しています。

○オブジェクトの色

それぞれのオブジェクトは 3 色に塗り分けられます。

- 白色 - マークされていない
- 灰色 - マークされているが、子オブジェクトはマークされていない
- 黒色 - 子オブジェクトと共にマークされている

○ 2 種類の白色

白色にはフリップフロップ的に入れ替わるの 2 種類の白色（白 A と白 B）があります。これはそれぞれ、現在の白色（現在の GC サイクル中に生成されたオブジェクト）、スイープ対象の白色（スイープされる予定の死んだオブジェクト）を意味しています。

A と B は、次の GC サイクルの開始時点に入れ替わります。現在の GC サイクル中に生成されたオブジェクトで白色であるものは、（新しい GC サイクルの開始時点で）死んだオブジェクトと見なされます。その一方、（GC サイクル前に生成されていた）すべての死んだオブジェクトはスイープされていることになります。（スイープ対象の白色は新しい GC サイクル開始時点ではもう残っていないはずなので）すべての白 A のオブジェクトを白 B に塗り直すよりも、白 A と白 B の意味を交換するほうが、簡単です。

この結果、現在の GC サイクルでスイープするオブジェクトは常に前回の GC サイクルから残されていたものになります。こうやって（GC のサイクル中に）新しく生成されらオブジェクトによる混乱を回避しつつ、インクリメンタルにオブジェクトをスイープできるようにしています。

○実行タイミング

GC の実行時間と処理の間隔は生存しているオブジェクトの数によって決まります。以下はその調整のための API です。

- `gc_interval_ratio_set`
- `gc_step_ratio_set`

詳細は、それぞれの関数のコメントを参照下さい。

○ライトバリア

（GC による誤回収を防ぐため）mruby の実装者および C 拡張ライブラリの開発者は、オブジェクトのフィールドからの参照を更新する際には、必ずライトバリアを挿入しなくてはいけません。オブジェクト A->オブジェクト B の参照を更新するときには、2 つの異なる種別のライトバリアが使えます。

- `mrb_field_write_barrier` - オブジェクト B をマークする
- `mrb_write_barrier` - オブジェクト A をマークする

○世代別 GC モード

mruby の GC は 3 色 GC の構造を使用しつつ世代別 GC モードを提供しています。このモードでは、スイープフェーズのあと、黒色オブジェクトを、白色ではなくて、「古い」オブジェクトとして扱います。基本的な考え方は伝統的な世代別 GC と同じです。

- マイナー GC
  - GC のマークフェーズにおいて「若い」オブジェクト（灰色オブジェクト）を通り過ぎるだけです。こうすることで、新しく生成されたオブジェクトだけがスイープされ、古いオブジェクトはそのままになります。

- メジャー GC
  - 通常のフル GC と同じです。

「伝統的な」世代別 GC モードとの違いは、メジャー GC が三色のアルゴリズムに沿って、インクリメンタルに起動される点です。詳細は、それぞれの関数のコメントを参照下さい。

6.4.1.2 GC のトリガー

GC 開始の判定は、`mrb_obj_alloc()` にあります。オブジェクト生成時に live のオブジェクトの数がしきい値を超えたときにインクリメンタル GC が始まります。

```
if (gc->threshold < gc->live) {
  mrb_incremental_gc(mrb);
}
```

オブジェクトの生成を行うような関数を呼んだ場合は、常に GC が割って入る可能性があることを意識しておく必要があります。

＊ `MRB_GC_STRESS` の定義が有効の場合、フル GC が毎回実行されます。

6.4.2 C の言語拡張内でオブジェクトを生成する場合

mruby では、C 言語拡張の中で生成したオブジェクトは全て利用中とみなして GC で開放されないようにマークされます。そのオブジェクトを記録するための領域が、GC arena です[*2]。ただしその arena にはデフォルトで 100 個までしかオブジェクトを記録できないので、C 言語拡張内でたくさんオブジェクトを生成すると、arena が溢れて例外が発生してしまいます。

実際に開放するべきではないオブジェクトは多くはないことが多いため、`mrb_gc_arena_save()`、`mrb_gc_arena_restore()` の 2 つの関数を用いて、arena に記録されない区間を作ることができます。

save、restore 間で生成したオブジェクトのうち、開放されたくないオブジェクトは `mrb_gc_protect()` で個別に保護する必要があります。

使い方のイメージは以下のとおりです。

```
function(mrb_state *mrb, mrb_value self){
   (ここで生成したオブジェクトは GC されない)

  int ai = mrb_gc_arena_save(mrb);
   (ここで生成したオブジェクトは GC される)
  mrb_gc_arena_restore(mrb, ai);

  mrb_gc_protect(mrb,obj);//開放されたくないオブジェクト
}
```

6.5 mruby によるハードウェア制御

mrbgem の作り方を把握したところで、実際に mruby から FabGL を介してハードウェアを制御してみましょう。

[*2] Matz にっき：`https://matzdiary.herokuapp.com/20130731.html`

6.5.1 描画 API を作る

これまでに知識をもとに FabGL を mruby から呼び出してみましょう

mruby-esp32-narya という mrbgem をリスト 6.3 のように実装してみました。ここでは、VGA ディスプレイに円を描くメソッドを実装しています。詳細は `https://github.com/kishima/mruby-esp32-narya` を参照ください。

▼リスト 6.3: 画面制御のための自作 mrbgem のコード（抜粋）

```
mrb_value mrb_narya_display_draw_circle(mrb_state *mrb, mrb_value self)
{
  mrb_int x;
  mrb_int y;
  mrb_int r;
  mrb_int col;
  mrb_get_args(mrb, "iiii", &x,&y,&r,&col);

  Color color;
  switch(col){
    case 0:
      color = Color::Black;
      break;
   (中略)
  }

  Canvas.setBrushColor(color);
  Canvas.fillEllipse(x, y, r, r);
  return self;
}

mrb_value mrb_narya_display_clear(mrb_state *mrb, mrb_value self)
{
  Canvas.setBrushColor(Color::Black);
  Canvas.clear();
  return self;
}

mrb_value mrb_narya_display_swap(mrb_state *mrb, mrb_value self)
{
  Canvas.swapBuffers();
  return self;
}

void mrb_mruby_esp32_narya_gem_init(mrb_state *mrb)
{
  struct RClass *narya_module = mrb_define_module(mrb, "Narya");
  struct RClass *display_module = mrb_define_module_under(mrb,
narya_module, "Display");
  mrb_define_module_function(mrb, display_module, "draw_circle",
mrb_narya_display_draw_circle, MRB_ARGS_REQ(4));
  mrb_define_module_function(mrb, display_module, "clear",
mrb_narya_display_clear, MRB_ARGS_NONE());
  mrb_define_module_function(mrb, display_module, "swap",
  mrb_narya_display_swap, MRB_ARGS_NONE());
}
```

そして mruby のコードをリスト 6.4 のように実装します。mrbgem で実装した API を利用し

て、円がランダムに流れていく画面を描画しています。詳細は https://github.com/kishima/family_mruby を参照ください。

▼リスト 6.4: mruby による VGA ディスプレイ制御

```
class Ball
  def initialize(x,y,r,col,speed)
    @x = x
    @y = y
    @r = r
    @color = col
    @speed = speed
  end
  attr_accessor :x, :y, :r, :color, :speed
  def move(x,y)
    @x += x
    @x = 0 if @x > 320
    @x = 320 if @x < 0
    @y += y
    @y = 0 if @y > 200
    @y = 200 if @y < 0
  end
end

def draw(ball)
  Narya::Display::draw_circle(ball.x,ball.y,ball.r,ball.color)
end

def load_balls
  balls = []
  15.times do
    balls << Ball.new(0, rand(200)+20, rand(20)+10, rand(6)+1,
rand(13)+2)
  end
  balls
end

balls = load_balls
count = 0
loop do
  Narya::Display::clear
  balls.each do |ball|
    ball.move(ball.speed,ball.speed/2)
    draw ball
  end
  Narya::Display::swap
  ESP32::System::delay(1)
  count += 1
  if count > 500
    count = 0
    balls = load_balls
  end
end
```

リスト 6.3 を実行した結果の画面がリスト 6.4 です。

▲図 6.2: サンプルスクリプトの実行結果

写真では動きは分からないですが、高速に円が流れていく様子が確認できます。これで、自分だけのデバイス上で、mruby を使って VGA ディスプレイの出力を制御できました！

あとはこの調子で、mruby から使える FabGL と ESP32 の API を増やしていくことで、より使いやすいプラットフォームになっていくことでしょう。

付録A　参考情報

A.1　Ruby/mruby の解説書籍

A.1.1　「まつもとゆきひろ直伝　組込 Ruby「mruby」のすべて 総集編」 まつもとゆきひろ（著）

まつもとさんの日経 Linux 上の連載をまとめたものです。mruby の特徴や実装が簡潔にまとまっているので、概要を把握するためにちょうど良いと思います。

Kindle 版が購入可能です。

A.1.2　「Ruby のしくみ -Ruby Under a Microscope-」 Pat Shaughnessy（著）島田浩二・角谷信太郎（共訳）

mruby ではないですが、Ruby の実装について詳しく解説された本です。かなりのボリュームで詳細にわかりやすく解説されているので、言語の実装を学ぶ上で大変勉強になります。

Kindle 版、書籍版それぞれ購入可能です。

A.1.3　「mrubook: mruby の実装を探検する薄い本」 大久保諒（著）

mruby の実装を解説した本です。本書の先輩にあたる本です。（筆者の頭が追いついていないため）本書のターゲットから外しているコンパイラやガベージコレクションについての解説がありますので、そちらが気になる方におすすめです。

Kindle 版が購入可能です。

A.2　参考 URL

- 「KiCad ことはじめ」
  - KiCad の基本的な使い方がまとまっています。本書で説明していない箇所も含めて最初はこの内容を一通り追ってみるのがよいと思います。
  - `http://docs.kicad-pcb.org/5.1.2/ja/getting_started_in_kicad/getting_started_in_kicad.pdf`
- 「KiCad で雑に基板を作る チュートリアル」
  - `https://www.slideshare.net/soburi/kicad-53622272`
- 「Electrical Information」
  - `https://detail-infomation.com/category/%e3%81%9d%e3%81%ae%e4%bb%96/%e5%9f%ba%e6%9d%bf/`
- 「KiCad Pcbnew 各レイヤの役割」

    - http://uchan.hateblo.jp/entry/2016/02/07/153224
- 「最近の mruby で C++ を使った mrbgem を作る時の注意事項的な何か」
    - https://qiita.com/kjunichi/items/ec87ca81557cb93e3b4f
- ESP32 official documents
    - https://github.com/espressif/esptool/wiki/ESP32-Boot-Mode-Selection
    - https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/sd_pullup_requirements.html
    - https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
    - https://www.espressif.com/sites/default/files/documentation/esp32_hardware_design_guidelines_en.pdf
    - https://www.espressif.com/sites/default/files/documentation/ESP32_FAQs__EN.pdf
    - https://www.espressif.com/sites/default/files/documentation/esp32-wrover-b_datasheet_en.pdf

A.3 Narya board の技術資料

本書の題材である Family mruby と Narya board の技術資料の参照先をまとめています。

A.3.1 基板関連

開発中ではありますが、Narya board の以下の情報を下記に格納しています。もし開発基板サンプルを入手した方や、自作された方の参考になればと思います。一応 GPL3.0 をライセンスとしていますが、一般的な設計図データには著作権は通常発生しないという話もあるので、製作者の気持ちとして受け取ってもらえればと思います。

- 回路図 (pdf)
- ガーバーデータ
- BOM
- KiCad 用自作シンボル/フットプリントデータ

https://github.com/kishima/narya_board/tree/master/dev_board/dev_v1.2

A.3.2 ソフトウェア

Narya board 上で動作するソフトウェアの参照先です。OSS ライセンスにて公開しています。

A.3.2.1 Family-mruby

ESP32 に焼くメインソフトです。今後も改修続けていきます。内部では、mruby、FabGL-component、mruby-esp32-narya、mruby-esp32-system を参照しています（本書執筆時点)。

https://github.com/kishima/family_mruby

A.3.2.2 mruby 2.0.1

https://github.com/mruby/mruby/releases/tag/2.0.1

A.3.2.3 FabGL-component

```
https://github.com/kishima/FabGL_component
```

A.3.2.4 mruby-esp32-narya

```
https://github.com/kishima/mruby-esp32-narya
```

A.3.2.5 mruby-esp32-system(Fork)

```
https://github.com/kishima/mruby-esp32-system
```

付録B　技術的補足

B.1　FreeRTOS の補足情報

B.1.1　app_main 関数はどこからやってくる

ESP-IDF アプリケーションの入り口は、app_main 関数ですが、その app_main 関数はどこからやってくるのでしょうか？ 本編の中で示した例では app_main 関数でタスクを生成したあとそのまま関数を抜けてしまっていますが、その後はどうなっているのでしょうか？

正解は、`esp-idf/esp32/cpu_start.c` です。cpu_start.c の中の main_task 関数がタスクとして実行されており、その中で app_main 関数を以下のような流れの中で呼んでいます。

```
static void main_task(void* args)
{
  //FreeRTOS の初期化完了待ち
  （中略）

  //起動時に使用していたヒープメモリ領域の有効化
  heap_caps_enable_nonos_stack_heaps();

  //SPIRAM が有効な場合の内部処理用メモリの予約
  （中略）

  //Watchdog の初期化
  （中略）

  app_main();

  //タスクの削除
  vTaskDelete(NULL);
}
```

この実装を見ると、app_main 関数を抜けたあと、main_task は自発的に終了することが分かります。またこのタスクが終了することでその他のタスクの処理を妨げるようなこともなさそうであることが分かります。

さいごに

謝辞

本書で実現しようとした各種機能を ESP32 一つで実現できたのは、FabGL という素晴らしいライブラリのおかげです。FabGL を見つけたときは、筆者がこんな感じのライブラリほしいな、と思っていたらそれを完全に実現しており、心を読まれているのか！？ というレベルで驚きました。FabGL を開発した Fabrizio Di Vittorio さんに大感謝です。Grazie mille!!

また、mruby を ESP32 に移植するための環境を構築してくれていた、Carson McDonald さん、YAMAMOTO Masaya さんにも感謝します。おかげで変なところで躓かず、導入がとてもスムーズになりました。

著者紹介

・kishima

組み込み系サラリーマン。Ruby でバリバリ仕事しているわけではないけれど、なんとなく Ruby（特に mruby）が好きなひと。

- 連絡先: `kishima@silentworlds.info`
- ブログ: `https://silentworlds.info`
- GitHub: `https://github.com/kishima`
- Twitter: `https://twitter.com/kishima`

ゼロから始める mruby デバイス作り (Light 版)

L チカのその先へ

2025 年 7 月 19 日　Light 版 ver 1.0

著　者　Kishima Craft Works
