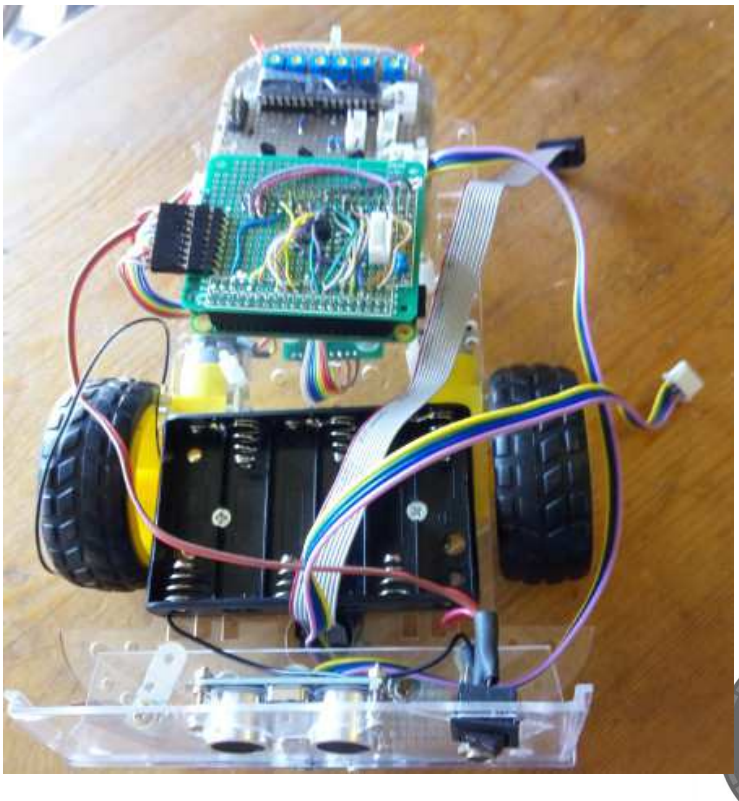
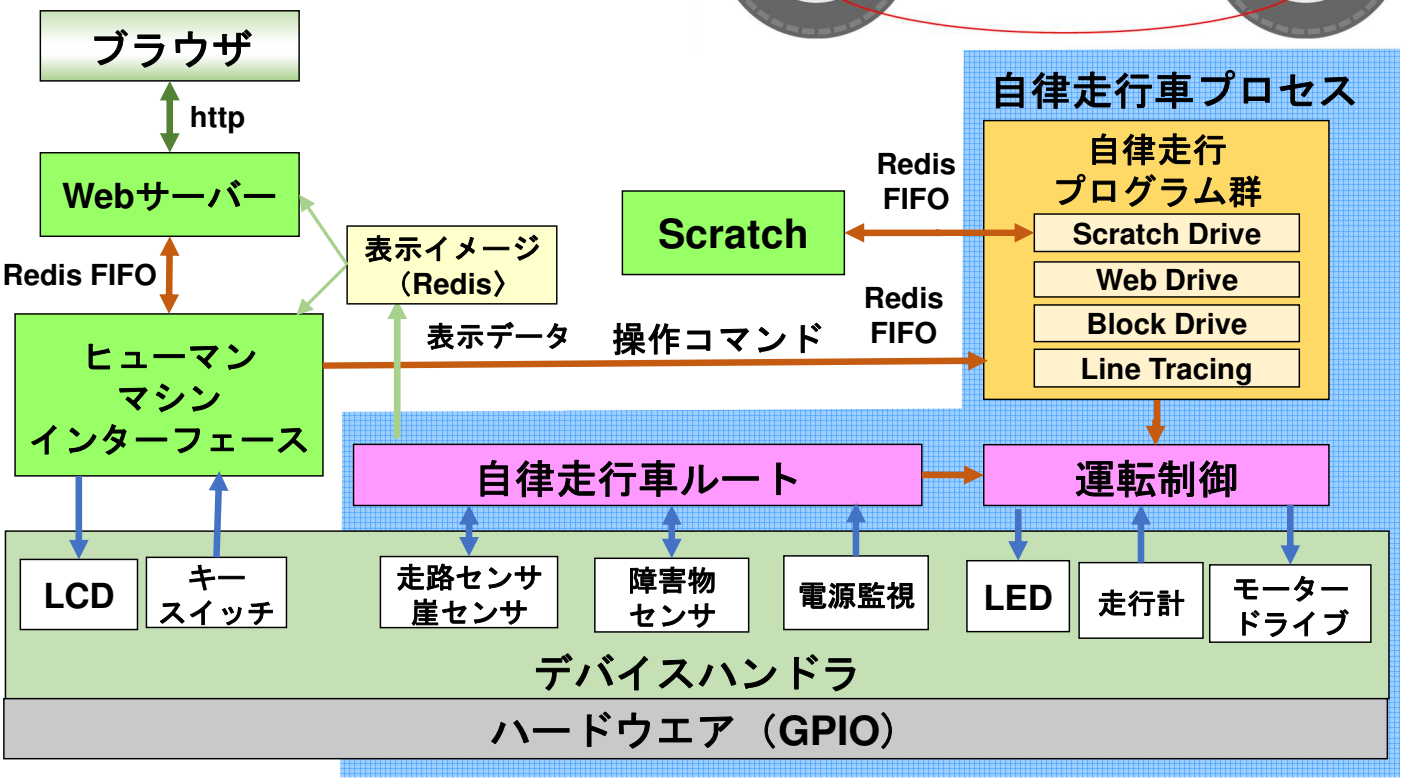
Raspberry Pi 中級電子工作

自律走行車

車両工作とマルチプロセス化





著者　秋山忠次

（非売品）

目次

コラム一覧

はじめに

2014 年の秋、書店で出版されたばかりの一冊の本[1]を見つけました。読んですぐに、「これはいい！」と思いました。Raspberry Pi は、小型 CPU カードとして非常に使いやすいと感じたからです。その理由は、

1. すぐに OS を立ち上げられる
2. emacs、cpp など、Linux のソフトウエア開発環境が活用できる
3. 自作ソフトウエアの大部分が Linux を搭載した PC 上でも走らせることができる
4. 周辺ハードウエアを駆動するライブラリが提供されているので、ハードウエアの詳細を知らなくても使える

などです。さっそく Raspberry Pi を手に入れ、最初のプロジェクトに取り掛かりました。

開発目標は、燻製などの調理に使える「温度コントローラ」でした。その経過と経験は『Raspberry Pi 中級電子工作：温度コントローラ　～設計から製作・検証・応用・保守まで』にまとめてあります。そこでは、設計手法から検証方法、さらに機能拡張の方法を説明しています。どちらかと言えば、ハードウエアの製作と駆動が主要なテーマでした。仕事の合間に進めたので、実用になるまで 1 年以上かかりました。ずっと使い続けており、改造は今でも続いています。

2020 年になって、次のプロジェクトとして、頼めばさまざまな業務を行ってくれる「秘書ロボット」の実現方法を探りました。音声の認識や合成は必須です。画像を扱ったり、感情表現を試みたり、メールを送ったり、検索エンジンを利用したりと、いろいろ試してみました。私自身の勉強も兼ねた、ソフトウエア中心のプロジェクトです。かなりの時間がかかると覚悟していたのですが、その頃「新型コロナウイルス」の感染が広がりました。外出自粛を余儀なくされ、思っていた以上に進んでしまいました。その過程は『Raspberry Pi 中（の上）級電子工作：対話型秘書ロボット～マルチプロセス・マルチプロセッサ化へのアプローチ』にまとめました。応答を良くするため、機能を PC やサーバーに分散させたら、Raspberry Pi を使う必然性が薄くなってしまったというのが、正直な反省点でした。

『対話型秘書ロボット』のあとがきで宣言したように、次のプロジェクトでは自律走行ができる車両を取り上げることにしました。開発対象は、温度コントローラより少し複雑ですが、ハードウエアの制御と、ハードウエアに依存しない抽象的な機能の組み合わせという点では、同程度の難易度なので、今回のレベルは「中級」としました。

実際に走行する車両なので、電池で動かします。電池の消耗を抑えるため、消費電流を減らすこと、パフォーマンスの低いプロセッサをあまり忙しくさせないことなどに留意しました。

『秘書ロボット』プロジェクトの報告書の最後に書いた、「実績のある手法は何度でも使いまわそう」という教訓に従っています。Redis を使ったプロセス間通信、Node.js を使ったサーバー設計、Socket.IO を使ったブラウザ・サーバー間通信などがそれにあたります。

このプロジェクトからは、『汎用測定チップ PICCOLO』の開発プロジェクトが派生しました。PIC をベースに、自律走行車の機能実現に必要なチップを開発しました。こちらは別の報告書があります。

なお、開発プロジェクトとは直接関係のない、趣味的な記事を、コラムとして掲載しています。楽しんでいただければ幸いです。

2024 年 4 月　著者

[1] 金丸隆志「Raspberry Pi で学ぶ電子工作 ― 超小型コンピュータで電子回路を制御する」講談社ブルーバックス；　2016 年に「最新 Raspberry Pi で学ぶ電子工作 ― 作って動かしてしくみがわかる」に改訂された。

1　Raspberry Pi プロジェクト（第 3 号）自律走行車

この本では Raspberry Pi を利用し、自律走行が可能な車両（一種のロボット）を作る過程を、一つのプロジェクトとして紹介します。

1.1　電子工作「入門」を卒業したら

Raspberry Pi を使った「電子工作入門」という書籍が多く出版されています。ひとむかし前の小型 CPU カードと比べると、Raspberry Pi は格段に扱いやすい素材です。とくに Linux と Python という、ソフトウエア開発に適した環境が後押しをしてくれます。気楽に電子工作を始めるきっかけになるので、入門書を活用して大いに経験を積みましょう。

でも、「LED がチカチカした」とか「温度センサの表示が読めた」という『体験』まで行きついた先はどうでしょう？　もう少しまとまったものを作って、その手法と経験を学ぶことが、何かの役に立つかもしれません。プロジェクトの達成感と反省は、単に役に立つことより、もっと有意義なものになると思います。今回のプロジェクトの対象は玩具っぽいものですが、それなりに興味深い点が多くあります。

1.2　自律走行車の構想を描く

開発に先立って、「何を実現するか」を最初に考えます。これをもとに開発仕様、必要なハードウエアの選択、ソフトウエアの設計と進めていきます。

自動車ではない

『自動車』は、道路交通法で「（第二条の九）原動機を用い、かつ、レール又は架線によらないで運転する車であって、原動機付自転車、軽車両…（一部省略）以外のものをいう」と定義されています。この規制を受けないよう、公道は走行せず、室内や敷地内などだけで走らせる「玩具」という位置づけにします。

電気で走る

走行車両は動力を必要とします。別に「エコカー」を気取るわけではありませんが、電池を動力源とすることにします。というより、他に選択肢がありません（内燃機関を搭載するのは大変です）。

入手性という点を重視し、乾電池で小型モーターを回すことにします。電子回路も同様に電池駆動なので、消費電流を低く抑えることが重要です。消費電流の少ない Raspberry Pi ZERO の出番ですね。

無線は使うが、ラジコンカーにはしない

自律走行車は、操作者の指令に従って走行します。走っている車両に有線で指示を送るわけにはいかないので。無線（赤外光通信も『無線』ですが、指向性があるので目的に合いません。電波を使います）無線 LAN（WiFi）を採用することにし、Bluetooth での指示は次の機会に回します。

似たような無線操縦車両の『玩具』として、ラジコンカーがあります。高速で走り回ったり、操縦技能を競ったりするわけですが、それは目指しません。自律性がない車両には興味がないのと、激しい衝突や横転に耐える設計が面倒だからです。

自律走行車のイメージは、指令を受けながら、自律的に移動するという点で、むしろ月面や惑星の探査車（NASA のマーズローバーや日本の SORATO など）に近いかもしれません。

自律走行車のイメージ（左：マーズローバー、右：SORATO）

コラム　月面車 SORATO のその後

2007 年に月面無人探査コンテスト（スポンサーは Google）が開催されました。当初の期限は 2015 年末でしたが、2018 年 3 月まで延長されました。参加登録は 34 チーム、そのうち 5 チームが最終フェーズまで進みました。そのなかに日本の HAKUTO チームが含まれています。自律走行車 SORATO を開発し、他チームのロケットに相乗りして、2018 年初頭に打ち上げる計画でした。しかしロケットの準備が整わず、打ち上げを断念しました（結局どのチームも達成できず）。グループは ispace 社を創設し、ロケットこそアメリカのスペース X 社のものですが、2022 年に自前の着陸船を打ち上げました。2023 年 4 月 26 日に月面着陸を試みたものの、残念ながら成功は叶いませんでした。ローバーを持ち込む次回の計画（2024～5 年）を応援したいと思います。

走行計画

自動車運転教習所では、実際の運転を始める前に、『運転計画』を立てるように指導されます。目的地や経由地はどこか、どういうルートで移動するのか、走行予定時間はどれくらいか、どこで休憩をとるかを考えてから、運転を始めるように言われます。もっとも最近は、ほとんどカーナビが代行してくれるので、あまり意識しない人が多いようです。

自律走行車両の場合、計画というほどではなくても、考慮する項目がいろいろあります。例えば、出発地と（経由地と）目的地、経路と走行速度、安全対策などが考えられますが、いちばん重要なのは走行を制御するアルゴリズムです。ラジコンのように命じられるままに猪突猛進する、走路を探して走行する、走行と回頭の組み合わせで走行する、状況判断を組み込んだプログラムで走行するなどが考えられます。このアルゴリズムを複数用意して、選べるようにすると面白いだろうと思います。

システム構成（を考えるときのヒント）

私たちが自動車を運転するときをイメージして、要素に分解してみます。下の図を見てください。

運転のイメージ

これから、運転は次のような要素に分解できることが分かります。

- 運転する目的と意思
- 知的な走行（運転）計画
- 身につけた運転操作
- 表示・指示・操作機構
- 走行するためのメカ（エンジン・車輪など）

このうち最初の 2 項目は知的な作業です。運転操作は繰り返しで身についたもので、ほとんど無意識で行うことができます。最後の 2 つは自動車の持つ機構です。あとで自律走行車のシステム構成を検討するとき、この図がヒントになります。

1.3 開発の準備

1.3.1 ソフトウエアの開発

自律走行車では、ソフトウエアの開発が重要な部分を占めます。実用的なソフトウエアの開発を手掛けたことのある人なら、モジュールの仕様設計と実装設計、プログラムの作成、出来上がったモジュールの検証という三項目にかかる時間は、ほぼ同程度だという実感があると思います。場合によっては、プログラムの作成にかける時間がいちばん短いこともあります。この本に許される分量のせいで、検証の説明を貧弱なものにせざるを得ませんでしたが、別途ダウンロードできるプロジェクトファイルには、検証用のプログラムが多く含まれています。

Raspberry Pi で GPIO 操作をプログラムするのに便利な言語は Python です。この本でも、できる限り Python（Python3）で記述するようにしています。ただし Web サーバーの設計などには JavaScript や HTML を使う必要があります。

Python でプログラムを書いていると、インタープリタの便利さから、使い捨て同然の扱いをするのを、よく見かけます。しかし、ここではそれを避け、Linux に備わったツールを使ってソフトウエアの保守性を良くしています。あまり一般的ではないかもしれませんが、とても役に立つ手法なので、最初のプロジェクトから採用しています。

1.3.2 プロジェクトに必要な機材

このプロジェクトに必要な機材（車両に部品として組み込むものは除く）を最初にまとめておきます。準備の参考にしてください。

PC

Raspberry Pi は部品として使い、最終的には車両に搭載します。ディスプレーやキーボードを取り付けたスタンドアローンなコンピュータとして、開発に使用することはできません。開発のために PC（パソコン）は必須です。

最低限必要になる機能は、Raspberry Pi に接続する SSH クライアントです。Windows PC の場合は、PuTTY や Tera Term などのフリーウエアを探してみてください。ファイルの編集には、フリーウエアの GNU emacs を使いました。Linux 用と Windows 用

の両方があるからです。もちろん、他のエディターを使っても構いません。

Ubuntu などの Linux OS を搭載した PC があれば、ソフトウエアモジュールの評価に使うと、結果の分析がやりやすくなります。これは必須要件ではありません。Windows の上で Linux を動作させる WSL (Windows Subsystem for Linux)というサブシステムも普及してきました。

最近は PC や Mac 用の Raspberry Pi Desktop という OS も配布されています。ここまで一致させる必要は、あまりないと思いますが。

WiFi 環境

Raspberry Pi ZERO にリモートログインするのに、WiFi（無線 LAN）環境は必須です。また、必要なパッケージをダウンロードするのにも使います。

自宅や開発場所での WiFi 設定を確認しておいてください。Raspberry Pi には、WiFi をサーチして接続するという機能がありません。誤った設定をすると、うんともすんとも言わなくなってしまいます。有線 LAN を装備しているモデルでは、そちらからログインして設定をなおすことができますが、Raspberry Pi ZERO には搭載していません。最悪の場合は、マイクロ USB インターフェースを持つネットワークアダプタを使えば有線 LAN に接続できますが、できれば避けたいものです。

屋外に出るときには、小型のルーターを使ったり、Raspberry Pi ZERO 自身がアクセスポイントになったりする（インターネットには出て行けない）ことが必要になります。これは、開発が一段落してから考えます。

工具と測定器

ハードウエアを組み上げるのには、それなりの工具が必要です。基本的なものは、ドライバ、ナイフ（カッター）、ニッパ、（車体加工用の）ドリルとやすり、半田ごて、ピンセットなどです。今回は信号の接続にコネクタを多用したので、ワイヤに端子を付けるための、カシメ工具も用意しました。

導通や電圧を調べるのはテスターで充分です。電子回路（とくにモーター駆動部）の動作を確認するときには、オシロスコープやロジックアナライザがあると便利です。今回は USB で PC につないで使う、簡易型オシロ兼ロジアナを使いました。PC のオーディオ入力を使うタイプでも良いと思います。このためだけに購入することはないので、この本の中では、オシロなしで済ませる方法も説明します。

コラム　自律走行と自動運転

日本の自動車業界では、自律走行する自動車（autonomous vehicle）を『自動運転車』と呼んでいます。これには次のようなレベルが定義され、現状は一部でレベル 3 まで実用化されており、2023 年には公道でレベル 4 の実証試験が始まりました。先進的（無謀？）な国では、運転者のいないタクシーも（試験）営業を始めました。

レベル	各レベルでの運転の自動化
1	運転者の運転を支援する（自動ブレーキなど）
2	運転者の責任で、運転の一部（追越など）を代行する
3	必要時に運転者が対応するが、基本的に自動で運転する
4	運転者が乗っているが、ほとんど自動で運転する
5	完全な自動運転で、人は乗客として乗る

自動運転のレベル

人によってはレベル 5 だけを自律走行と呼び、それ以外は補助的な機能や過渡的な段階だと位置づけているようです。最大の関心事は人（運転者、乗客、他の車両に乗っている人、歩行者など）の安全だからです。

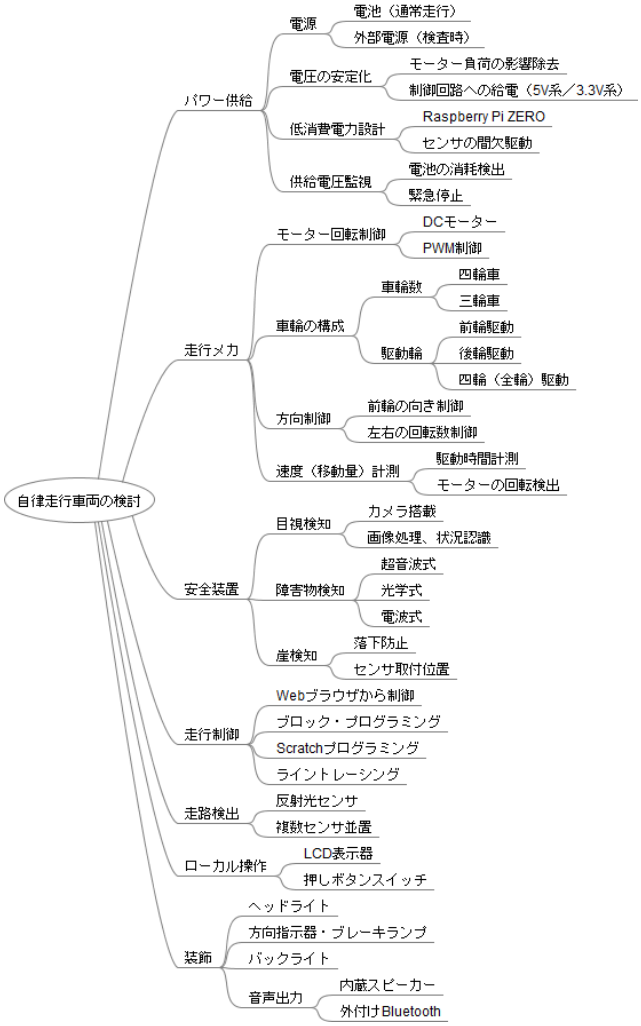
今回のプロジェクトの対象である『玩具』の車両には、人にぶつかってもあまり痛くないくらいの出力しかありません。もちろん、人に突っ込まないようにするのですが、むしろ重要な関心事は、車道に飛び出さない、壁にぶつからない、テーブルや段差から落ちないといった、車両自体の『安全（というより経済性）』です。これをもとに定義を文字どおりに解釈すれば、ブロックドライブやライントレースはレベル 4 か 5 と言えるのかもしれませんが、実際の機能はレベル 1 程度です。

2　開発仕様設計とハードウエアの選択

2.1　構成要素の検討

自律走行車両は、多くの要素から組み上げることになります。それぞれの要素として、どんなものを選ぶかを考えます。

このとき、下のようなマインドマップを使うと、整理しやすくなります。まず、考えられる要素をすべて書き出し、関連のある要素をまとめていきます。その過程で、新たな要素が追加されることもあります。これは、川喜多 二郎さんが考案した、KJ 法（考案者のイニシャル）というアプローチです。



構成要素を検討するマインドマップ

整理した構成を以下で説明していきます。

2.1.1 パワー供給

車両を動かすための電源は、基本的に電池（単三）です。電線を引っ張ったまま走行するわけにはいかないので。ただ開発・検証段階では静止していることが多いので、電池の消費を避けるため、外部（補助）電源も使えるようにします。

一般に、モーターと CPU の電源は別々の電池から取るべしと言われます。これは電池の内部抵抗のせいで、モーターの駆動時に CPU 電源が低下する（リセットされることがある）のを防ぐためです。そこで電源の出力インピーダンスを下げる目的で、降圧型の DC/DC コンバータを使うことにしました。一次（電池）側の電圧が 6V 以上になるようにします。車台に載る電池ボックスがあるので、電池は 6 本を直列で使います。9V から 5V 電源を作って供給し、Raspberry Pi が自分で 3.3V 電源を作ります。単三アルカリ電池の容量は 2000mA 時くらいです。DC/DC コンバータの効率が 90%あれば、200mA の回路を 15 時間以上動作させることができるはずです。

モーターの駆動電流を制限するのは難しいので、他で低消費電力化を図ることを考えます。CPU ボードはアイドル時の消費電流が 100mA 程度の Raspberry Pi ZERO を採用します。WiFi を使ったり、CPU 負荷を増やしたりすると、消費電流も増加します。2021 年には 64 ビット 4 コア CPU を搭載した Raspberry Pi ZERO 2W が発売されました。今回のようなマルチプロセスシステムに最適だと思います。しかし、あまり出回っていないし、消費電流も 30%ほど増加するということなので、当面はこのまま進めます。

コラム　飛行機の外部補助電源

飛行中の航空機は、すべての動力をエンジンから取り出しています。空港で駐機したら、エンジンを止め、地上電源装置（GPU）から照明や空調のための電気を受け取るようにします。排気を出さず、エネルギー効率を上げることが目的です。胴体や翼の下側に、写真のようなトラ模様の太いケーブルが接続されているのを空港などで見かけたことはありませんか？

今回の自律走行車では、電池で動力をまかないます。開発期間中に何度も電池を交換するのは、コストがかかるうえに面倒です。そこで、GPU のように外部電源（ＡＣアダプタ）を使えるようにしました。車輪を浮かせてモーターを回すなどの工夫で、実走行するまでは外部電源で開発することができました。

次に消費電流が多くなるのが、路面などを検出する光センサです。光源（LED）の駆動電流は 20mA が推奨されていますが、数が多いので消費電流も大きくなってしまいます。これを抑えるため、光源は間欠駆動（必要な時だけ光らせる）とし、光量も制限します。

電池の放電が進むと起電力が低下します。DC/DC コンバーターの最低入力電圧が 6V 程度なので、一本当たり 1V 以下になったら「電池切れ」として、モーターを止め、電池交換を促すことにします。電池電圧を（適当に分圧して）測定し、6V 以下で警報を出すようにします。

2.1.2 走行メカ

車両を動かすには、いちばん簡単に使えるブラシ付き DC モーターが適しています。ギアで回転数を落として推力を取り出します。回転速度は印加電圧（と負荷）で決まるので、パルス幅変調（PWM）で調整することにしました。

次に車輪の数と、どれを動輪（モーターで駆動する車輪）にするかを検討します。走行方向をどうやって制御するか（操舵、ハンドル操作）を含めて考えましょう。操舵には、サーボモーターを使って自動車のように前輪の向きを変える方法と、左右の動輪の回転数を変える方法が考えられます。それぞれ一長一短がありますが、今回は「その場で向きを変えること（回頭）」に適した、後者を採用しました。それに適した車輪の構造は、前に二輪（動輪）と後ろに一輪の三輪車です。

走行メカに付随するセンサとして、速度や走行距離の測定が考えられます。車輪と同時に回転する穴あき円盤（エンコーダ）の回転を測ることにします。

2.1.3 安全装置

自動車の安全な走行に責任がある運転者は、主に視覚を使って状況を確認します。レベル 2 以上の自動運転車では、カメラを使って車線や先行車を認識しますが、貧弱な CPU パワーでは無理な相談です。それはすっぱりあきらめ、車両の衝突や落下を防ぐことを考えました。

カメラ以外に進行方向の障害物を検知するには、電波、超音波、光などの反射が使われます。壁や家具を障害物として想定すると、材料の導電率によらない超音波が適しています。車両の前面に超音波センサ（距離計）を設置します。本当は後退用のセンサもあると良いのですが、長距離のバック走行はないとして諦めます。

机や上がり框からの落下を防ぐため、前輪のさらに前方にクリフ（崖）センサを取り付けます。赤外線が床で反射したのをとらえ、なくなったら崖だと認識させることにしました。車輪より前で検出できれば、停まることができます。ここでも、崖に平行に走って脱輪したり、後退するときに崖があったりしたときの対策は省略することにしました。

2.1.4 走行制御

車両の走行を制御する方法（アルゴリズム）は、何通りも考えられます。次の四種類から選んで実行できるようにし、追加方法も検討しておくことにします。

ラジコンもどき

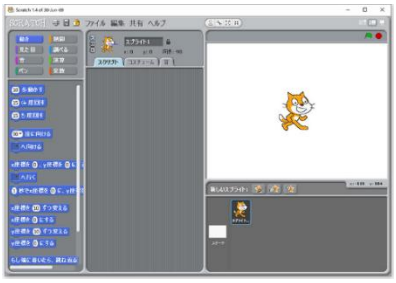
Web 画面を使って人が操縦する方法です。走行・加減速・停止・操舵などのボタンをクリックすることで操縦します。ジョイスティックや十字ボタンはないので、必ずしも操縦性は良くありません。「Web ドライブ」と呼ぶことにします。

シーケンス制御

「○○cm 前進する」とか、「右へ○○度回る」などのブロックを組み合わせて走行を制御できるようにします。子供向けのプログラミング学習に似た教材（超小型ロボットの運動をプログラムする）がヒントです。ブロックのシーケンスを編集したり、記憶したりできるようにすれば楽しいと思います。「ブロックドライブ」と呼ぶことにします。

プログラム制御

簡単なプログラミング言語で車両の動きを制御できるようにします。具体的には Scratch 言語を拡張し、そこで猫がステージ（猫の活動範囲）を動き回るように、移動範囲を走り回らせることを考えます。猫と同じように実車両が動けば、子どもたちも興味を抱いてくれると期待しています。「スクラッチドライブ」と呼ぶことにします。



Scratch1.4 の画面（右上のステージを猫が歩き回る）

走路認識

走行すべき経路を認識し、それに沿って走行できれば自律走行らしくなります。第一段階は、床面に描いた線をなぞるように走行（ライントレーシング）できるようにします。

2.1.5 走路検出

ライントレーシングのため、床面に描いた走路（白線または黒線）を検出できるようにします。動輪近くに複数の反射センサを（車幅方向に）配置し、走路を追尾できるようにします。

2.1.6 ローカル操作

自律走行車への指令は Web 経由で与えるのが基本ですが、WiFi 接続前や屋外での使用を考慮し、最低限のローカル操作機能を持たせます。ブラウザからも全く同じ操作ができるようにしておきます。

LCD 表示器

操作内容や現在の状態を小型 LCD に表示させます。表示内容はブラウザからも見えるようにします。

キースイッチ

キースイッチを押すことで表示内容の変更や操作を行えるようにします。ブラウザのクリックでも、同じことができます。

2.1.7 装飾

ここからは趣味的な内容ですが、前照灯（ヘッドライト：前方の左右)、赤色灯（方向指示灯・ブレーキ灯として共用：後方の左右)、後退灯（バックライト：後方の中央）を取り付けます。

前照灯

機能は実装しますが、いつヘッドライトを点灯するかは当面未定です。

方向指示灯

曲がる方向の赤色灯を点滅させます。

ブレーキ灯

減速時や停止時に赤色灯を同時に点灯させます。

後退灯

後退時に後退灯を点灯させます。

ハザード

異常時に二個の赤色灯を同時に点滅させます。障害物を回避するため後退することがあるので、後退灯は点滅させません。

音声出力

スクラッチドライブでは、猫の発言や報告などを（ふつうはテキストですが）音声で行えると面白いと思います。車両にオーディオ回路を搭載することも考えたのですが、GPIO ポートが足りません。それに、Scratch 端末から離れているかもしれないので、操作者の近くで音を出した方が良さそうです。

2.2 開発仕様

自律走行車両 Raspbuggy の開発仕様をまとめます。

2.2.1 ハードウエア仕様

項目	仕様
外形	自律走行型三輪車（前輪駆動）
電源	電池（単三 6 本）または AC アダプタ（9V/0.5A）
CPU	Raspberry Pi ZERO
動力系	ギア付 DC モーター（左右動輪に各 1） 個別に PWM 駆動
走行計	動輪ごとの回転を光学エンコーダで検出
障害物センサ	超音波センサ（検知範囲：15cm～1m 以上）
崖・走路センサ	赤外線反射式センサ
操作機能	LCD 表示器とキースイッチ（2 個）
点灯機能	前照灯、方向指示灯兼ブレーキ灯、後退灯

2.2.2ソフトウエア仕様

項目	仕様（機能項目）
OS	Linux (Raspberry Pi OS)
システム構成	マルチプロセスシステム プロセス間通信・同期機能を実現する
自律走行 サブシステム	自律走行制御プログラム（種類は複数） 運転制御機能 駆動メカのハンドラ 各種センサのハンドラ
HMI サブシステム	操作インターフェース（HMI）
Web サーバー サブシステム	走行プログラムの編集や指令を与える 現在の状態を表示する
プログラム開発 言語	Python3 Javascript/Html（Web サーバー、Scratch）

2.3　ハードウエアの選択

Raspberry Pi ZERO 以外のハードウエアユニットを選びます。個々の電子部品や具体的な回路図は第 4 章で説明します。

2.3.1 車台

自律走行車の土台となるのが車台です。構成要素の検討結果から、前輪駆動の三輪車を作るキットを探しました。モーターや車輪が付属していたので、お買い得でした。後輪はキャスターなどに使われるものですが、その場で回転する（回頭）と、車軸と垂直な横方向を向きます。そのまま直進しようとすると、後輪の軸が回転するので外乱になります。

採用した車台（写真左側が前）

車台は透明アクリル板（写真は保護紙を貼ったままの仮組状態）で、その上下に部品を取り付けます。なるべく最初から開いている穴を利用しますが、部品取付け用のねじ穴の追加が必要でした。前面には、超音波センサなどをマウントしたパネル（100 円ショップで購入したアクリル製写真スタンドを加工）と、センサ保護用のバンパー（色付きプラスチック板）を取り付けます。

コラム　三輪自動車

三輪自動車といえば、私の年代ではオート三輪とか、ダイハツミゼットを思い出します。今でも、東南アジアではタクシーとして使われています。これらは前輪が一つで、小回りがきくのが特徴です。

ドイツのメッサーシュミット社が戦後（航空機の製造が規制されていたころ）作っていた三輪車（後輪駆動）は、いまでも見かけることがありますね。イギリスのモーガン社は、前輪駆動三輪車を最近になって復刻しました。

三輪自動車（左：メッサーシュミット、右：モーガン）

部品の取り付け位置は、上面側と底面側でネジなどが干渉しないように注意して決めます。それでもナットを支える指が入らない場所ができてしまい、割りばしの先に両面テープで止めてからねじ止めした個所がありました。

車台の主なデータは以下のとおりです。

項目	仕様（推定）
外寸	全長 210mm、幅 98mm（最大幅 150mm）
板厚	3.8mm
前輪トレッド幅	124mm（車輪の中心の間隔）
前後輪間隔（ホイールベース）	前後方向に 125mm

車台のデータ

2.3.2モーターと車輪

車台キットに付属していたモーターの素性は不明でした。ギアを内蔵したブラシ付き CD モーターなのは確かです。中国製らしいので、外観がそっくりなものを調べてみると、次のような特性が分かりました。

項目	仕様（推定を含む）
印加電圧	3V～（5V以上で使えるが上限は不明）
無負荷電流	75～250mA
ギア比	48:1
最大回転数	100～120rpm
最大トルク	0.7～0.8 kg-cm

モーターの特性

電流のデータがバラバラなのは、測定条件がまちまちなせいのようです。

モーターと動輪、ロータリーエンコーダ（底面側から撮影）

動輪は直径 64mm、幅 25mm なので、無負荷時の車速は最大でも

$$64 \times \pi \times 100\text{rpm}/60\text{ 秒} = 33\text{cm/秒}$$

程度で、実機はもっと遅くなることが予測されます。

車輪には一周 20 穴のロータリーエンコーダが付いてきました。LED とフォトトランジスタからなる光インターラプタで挟み、パルスとして取り出します。1 パルス当たりの移動距離は

π x 64mm / 20 = 10mm/pulse

となります。最大パルス周波数は 33Hz です。

左右の動輪を逆方向に回転させると、動輪の中心の周りに回転します（トレッド幅を直径とする円を描く）。一周（360 度回頭）すると、走行距離は

π x 124mm = 390mm（39 パルス）

なので、1 パルス当たりの回転角は

360 度 / 39 = 9.3 度/pulse

です。つまり 90° 回頭するときの 10%以上あるので、分解能は十分とは言えません。いっぽう回転速度（ラジアン/秒）は動輪速度÷トレッド幅の半分です。速度 40%で回頭すると、100ms の間に

330mm x 40% x 0.1s x 180/ (62mm x π) = 12 度

も回ってしまうので、これも回転精度を落としてしまいます。回りすぎ防止を加える程度で我慢することにします。

2.3.3 モーター制御ユニット

モータードライブモジュール（購入）は、PWM のインターフェースが簡単になることを優先して選びました。モータードライバとしてよく例に挙げられている TA7291P（廃品種）や RV8835 といったチップは、2 つの制御信号を使って、正転・逆転・惰走・ブレーキを切り替えます。PWM 制御をしようとすると、正転と逆転では使う信号が変わってしまいます。ソフトウエア PWM なら問題ないのですが、Raspberry Pi ZERO のハードウエア PWM を使うのには不便です。

そこで PWM と方向制御が別信号になっているドライバを選びました。TB6612 はこういうインターフェースになっているばかりか、2 チャンネルのドライブ回路が集積されています。電圧・電流の使用範囲は定格内です。秋月電子通商の AE-TB6612 という小型ユニットを調達しました。おもな仕様は以下のとおりです。

項目	仕様（推定を含む）
コントローラ	TB6612FNG（東芝）
制御信号電源	2.7V～5.5V
モーター電源	2.5V～13.5V
ブリッジ回路	2 系統
最大出力電流	1.2A（各チャンネル）
出力オン抵抗	0.5Ω
制御信号（系統ごと）	速度制御用 PWM、動作選択 2 ビット
PWM 周波数	最大 100kHz

モーター制御ユニットの仕様

初期検討では、I2C バスインターフェースのドライバ RV8830 を採用したモジュール（秋月電子通商 AE-MOTOR8830）も候補にあげました。採用を見送ったのは、モーターごとに 1 枚のモジュールが必要だったせいです。制御信号とモーターの電源が共通なのも、使いにくい理由です。

モーターの動作範囲は 3V からとなっているので、5V を PWM で駆動すると、デューティ 60%以上でないと動かないことになります。実際には、無負荷状態で 30～40%から回転しました。電池出力（9V）を直接 PWM すれば制御可能範囲が広がりますが、モーターの定格電圧が分からないので止めました。

2.3.4 超音波センサ

前方の障害物検知には、定番の超音波センサ HC-SR04 を使います。一時期は受信信号端子が高インピーダンス状態になってしまうという不具合がありましたが、今は改善されているようです。

10μs 以上のパルスを与えると超音波を送信し、送信からエコー受信まで H レベルになる受信信号でインターフェースします。このモジュールの電源と論理レベルは 5V なので、3.3V 系の Raspberry Pi ZERO との間でレベル変換を行います。

コラム　かわいい自律走行車

ネットで情報をあさっていたら、写真のような模型を見つけました。前輪の角度を変える方式の四輪車で、顔も左右に振ります。目玉は超音波センサ、口のところにカメラがあって、人の顔を探して追尾するというかわいい機能があります。これをベースにしてもいいかと思ったのですが、Raspberry Pi ZERO に画像処理は重すぎます。Raspberry Pi 3 または 4 が必要だということでした。電源電流を食いすぎるので、しぶしぶ諦めました。

2.3.5 カウンタ

超音波センサのエコー信号のパルス幅や、ロータリーエンコーダのパルス数を計測するにはカウンタ機能が必要です。しかし Raspberry Pi のコアには、この目的で使えるカウンタがありません。ソフトウエア処理では間に合わないので、計測専用チップ PICCOLO を用意することにしました。

このチップの開発については別冊にまとめたので、この本では仕様と使いかただけを説明します。

項目	仕様
外形	14 ピン DIP
インターフェース	I2C 通信（スレーブ）
測定機能	パルス幅計測（1 入力） パルス周波数計測（1 入力） イベント計数（最大 4 入力） アナログ電圧（最大 4 入力）

PICCOLO チップのおもな仕様

2.3.6 走路センサ

赤外光の反射を使って走路を検出します。右のコラムにあるように、消費電流を少なくするのが重要です。そこで『間欠駆動』できる回路（センサ 5 素子）を自作することにしました。素子数が少ないので分解能が低く、あまり制御性はよくありませんが、走行はできました。

2.3.7 電源ユニット

電池の出力から 5V 電源を作るため、降圧型の DC/DC コンバータを使います。秋月電子通商が販売している AE-LXDC55-5V という小型モジュールを使うことにしました。下におもな仕様をまとめますが、目的に合致していることが分かります。

項目	仕様（推定）
入力電圧範囲	6V～14V
出力電圧・電流	5V　最大 1.5A
入出力間	非絶縁
寸法	11mm×25mm
制御チップ	LXC55（村田製作所）
制御周波数	2MHz

電源ユニットの仕様

外部電源は、出力電圧が DC/DC コンバータの入力電圧範囲内で、0.5A 以上取れることが必要です。

2.3.8 電源モニタ

一次（電池または外部電源の）電圧を測定し、DC/DC コンバータの入力範囲を外れたら警報を出せるようにしておきます。実は、この機能は汎用測定チップ PICCOLO で実現できます。それに気が付いたのは、下に示す A/D コンバータを購入し、組み付けてしまった後でした。変更するのも面倒なので、そのまま使うことにしました。

特性	仕様
供給電源	2.7V～5.5V
入力信号	差動型（±基準電圧以内）
変換方式	16 ビット ΔΣ 方式
インターフェース	I2C バス
基準電圧	2.048V±0.05%
内部増幅率	1、2、4、8
パッケージ	表面実装型（SOT-23）

A/D コンバータ MCP3425 の主な仕様

コラム　採用を諦めた走路センサの候補

10 年以上前だったら、CCD リニアイメージセンサを使って路面を『撮影』することを考えたかもしれません。一次元センサですが、横方向の解像度（分解能）は小さいものでも 1000 画素以上あるし、秋葉原で手に入れることができました。FAX などの用途があったので、大量に出回っていたのです。ところが、二次元 CMOS 撮像素子の性能が向上して、FAX よりスマホで撮影した画面をメールで送る方が一般的になってしまいました。今では高解像度であることを利用した検査工程などの用途に限られているようです。

CCD リニアイメージセンサの外形

今でも入手できないわけではありませんが、読み出しに電流（クロック周波数によるが 100mA 以上）を食うため、電池駆動の自律走行車では苦しいと判断しました。

秋月電子通商から走路検出用の AE-NJL5901AR-8CH というユニット（センサは 8 素子）が発売されています。基板のセンサと反対側に可変抵抗と表示 LED が搭載されているので、感度調整は楽そうです（車台の下に取り付けるのには向きませんが）。しかし、これも 200mA 弱の電流が必要です。電流ケチケチ作戦をとるため、ユニットを自作することにしました。

3　Raspberry Pi ZERO の準備

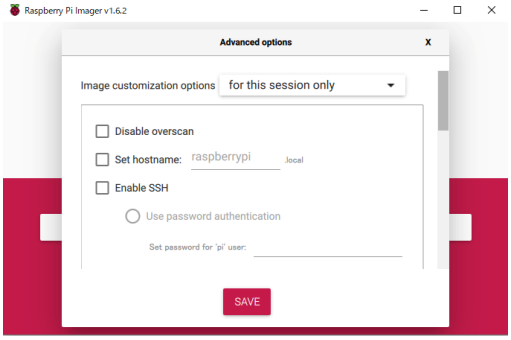
CPU カードである Raspberry Pi ZERO の準備をします。WiFi と GPIO コネクタの装備された、Raspberry Pi ZERO WH を用意しました。準備段階では机の上で動かしても構いませんが、絶縁物の上に置き、回路がショートしないよう注意してください。5V の USB 電源を PWR IN と表示されたコネクタ（端側）に接続します。

3.1　Raspberry Pi OS のインストール

まずオペレーティングシステム（Raspberry Pi OS）を立ち上げます。以前はイメージファイルをダウンロードしてから SD カードに書き込む必要がありましたが、2020 年 3 月に Imager というツールが公開され、格段に便利になりました。（https://www.raspberrypi.com/software/）から Raspberry Pi Imager を PC にインストールします。2021 年 11 月現在のバージョンは 1.6.2 でした。起動画面を下に示します。

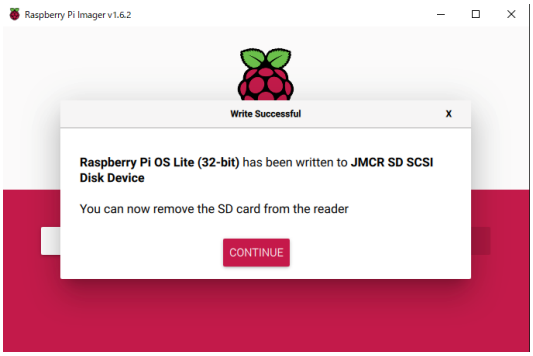
OS と SD カードを選ぶよう求めています。CHOOSE OS をクリックすると選択肢が示されますが、サーバー機能しか使わないので、Raspberry Pi OS (other)から Raspberry Pi OS Lite (32-bit)を選びました。次に CHOOSE STORAGE から SD カード（ふつうは一つしか候補が表示されません）を選ぶと、WRITE というボタンが表示されるようになります（まだクリックしないでください）。

以前は SSH 接続や WiFi の設定をファイルで与える必要がありましたが、Imager が面倒を見てくれるようになりました。キーボード上で Ctrl と Shift と x を同時に押すと、次のようなオプション設定画面が表示されます。



Set hostname にチェックを入れると Raspberry Pi のホスト名を変更できます。同じ環境で複数の Raspberry Pi を使うときは、それぞれの名前を設定しておきましょう。Enable SSH にチェックを入れ、パスワード認証（すでに選ばれている）にします。パスワードを入力する必要はありません。次に Configure wifi にチェックを入れ、SSID とパスフレーズを与えます（PC の WiFi 設定がコピーされているので、同じ環境で使う場合はそのままにしておきます）。国名が GB（英国）になっているので、JP（日本）を選びなおします（使える WiFi チャンネルが国ごとに違うため）。Set locale settings にチェックを入れ、時刻ゾーンを Asia/Tokyo にします。キーボードは接続しないので、どんな設定でも構いません。

SAVE をクリックしてから、元の画面に戻ります。ここで WRITE をクリックし、SD カードの内容が消去されるという警告を承認すると、書き込みが始まります。OS イメージをダウンロードしながら書き込むので、少し時間がかかります。下のようなメッセージが表示されれば完了です。



SD カードをアンマウントし、PC から取り出します。Raspberry Pi のマイクロ SD スロットに取り付け、電源を投入して、しばらくすると WiFi 上に Raspberry Pi が現れます。この段階の IP アドレスは、自動アドレス割り付けの範囲に設定されています。PC のコマンドプロンプトから `ping` コマンド

で、この範囲を探せば出てくるはずですが、もし出てこなかったら WiFi 設定を確認してください。arp -a コマンドでそれまでに見つかった IP アドレスを調べます。そのなかで、物理アドレスが b8:27:eb（Raspberry Pi Foundation のベンダーコード）で始まるアドレスが Raspberry Pi のものです。

IP アドレスが分かったら、PC から SSH 接続ができます。Windows なら PuTTY などの SSH クライアントを起動します。ユーザ名は pi、パスワードは raspberry です。Ubuntu の端末ソフトでは、Ubuntu のユーザ名を SSH 接続先でも使おうとするので、IP アドレスの前にユーザ名 pi@を付けてください。

```
$ ssh pi@IP アドレス
```

Raspberry Pi にログインできましたか？　すぐにパスワードを変更しろというメッセージが表示されるはずです。私は Ubuntu PC と同じユーザ名を使いたいので、自分の苗字と名前をグループ名、ユーザ名として登録しています。以下で最初のコマンドはグループ akiyama の追加、二番目はユーザ chuji をグループ akiyama に追加します。パスワードの設定を求められるので与えてやります。その他の情報は聞かれてもリターンを返すだけで十分です。ホームディレクトリは自動で作成されます。三番目は chuji に（システム管理者権限でコマンドを実行する）sudo 権限を与えています。説明の都合で、私の名前を使いましたが、皆さんは自由に選んでください。

```
$ sudo groupadd akiyama
$ sudo sudo adduser -ingroup akiyama chuji
 :
$ sudo usermod -G sudo chuji
$ exit
```

コマンドの最初にある sudo は、システム管理者権限で実行するという指示です。システムの変更やハードウエアの制御にはシステム管理者権限が必要です。パスワードを訊かれることがあるので、その時はログインパスワードを入力してください。

ここでいったんログアウトすれば、今度はユーザ名 chuji でログインできます。

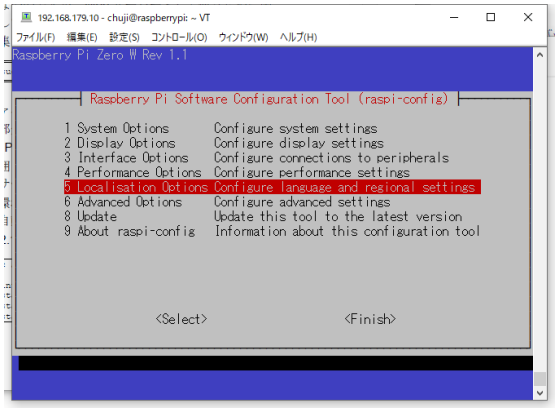
3.2　Raspberry Pi の基本的な設定

ログインしたら、ハードウエア構成ソフトである raspi-config を管理者権限で起動し、GPIO などの設定を行います。

```
Linux raspberrypi 5.10.63+ #1488 Thu Nov 18
16:14:04 GMT 2021 armv6l

The programs included with the Debian GNU/Linux
system are free software;
the exact distribution terms for each program are
described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY,
to the extent
permitted by applicable law.
Last login: Thu Dec  2 16:12:20 2021 from
192.168.179.116

SSH is enabled and the default password for the
'pi' user has not been changed.
This is a security risk - please login as the 'pi'
user and type 'passwd' to set a new password.

chuji@raspberrypi:~ $ sudo raspi-config
```

そうすると、次のような設定画面が表示されます。上下矢印で項目を選び、Enter キーを押してその項目の設定に移ります。画面下側の選択肢に移るには、Tab キーを押します。

なお、デフォルトで選択される bash を使っていると、上のように<ユーザ名>@<ホスト名>:<現在のディレクトリ> $というプロンプトが表示されます。ユーザ名やホスト名はこの例に限らないので、混乱を避けるために、以下では単に $ だけを表示します。



Raspi-config 画面

Imager でロケールを設定したはずなのに、反映されていなかった（設定し忘れたのかもしれない）ので、改めて設定します。5 Localisation Options から L1 Locale へ移り、リストのずっと下のほうにある jaJP.UTF-8 UTF-8 まで行ったら、スペースキーを押します。[*]と表示が変わるので、<OK>を選びます。デフォルトの選択では同じものを選んでおきます。設定にしばらく時間がかかります。そのあと L2 Timezone を選び、Asia→Tokyo と設定しておけば、時間表示が日本標準時になります。

次に 3 Interfacing Options から P5 I2C を選ぶと、I2C バスを動かすかどうか聞かれます。＜はい＞を選んで、I2C バスを動作可能にして、終了です。

Imager で hostname（ホスト名）を設定し忘れた場合は、デフォルトの Raspberrypi が設定されています。Raspberry Pi を使ったシステムをいくつも使う可能性を考慮して、ホスト名を Raspbuggy（他の名前を選んでも構いません）に変更しておきましょう。以下の二つのファイルを修正します。エディターとして vi を使った例を示します。nano でも構いません。

```
$ sudo vi /etc/hostname

$ sudo vi /etc/hosts
```

ホスト名として Raspberrypi と書かれているところ（各一か所）を Raspbuggy に変更します。

3.3 WiFi の設定

基本的な設定は終わりましたが、このままでは IP アドレスが自動設定のままです。同じルーターを使っていれば、IP アドレスが変わってしまうことはあまりないのですが、Web サーバーとして使うために固定しておきたいと思います。以下の設定ファイルを編集します。

```
$ sudo vi /etc/dhcpcd.conf
```

ファイルの最後に次の行を追加します。
192.168.179‘の部分は、自分の WiFi 環境に合わせ、Raspberry Pi の IP アドレスは、自動アドレス割り付けに使われる範囲（WiFi ルーターに設定されている）の外で、他のサーバーが使っていないアドレスを選びます。私の環境では 192.168.179.100 から 192.168.179.200 までが自動割り付けに使われているので、その範囲外の 192.168.179.36 にしました。

```
# added by chuji on 2020/2/20

interface wlan0
static ip_address=192.168.179.36/24
static routers=192.168.179.1
static domain_name_servers=192.168.179.1
```

ファイルをセーブしたら、Raspberry Pi を再起動（リブート）します。下のオプション -r はリブートするという指定です。sudo reboot でも同じ結果が得られます。

```
$ sudo shutdown -r now
```

しばらく待てば、新しい IP アドレスに SSH 接続することができるようになります。そのときホスト名も変更されていることを確認してください。

この節の最後として、Raspberry Pi の電源の切り方を説明します。いきなり電源を落とすことを避け、シャットダウンを実行します。オプション -h はシステムを停止するという指定です。

```
$ sudo shutdown -h now
```

しばらく待てば、Raspberry Pi がシャットダウンし、自動的に自分の電源を切ります。緑色 LED が消えたら、AC アダプタの電源も切っておきます。

3.4 ソフトウエアパッケージのインストール

あとで必要になるソフトウエアパッケージをインストールしておきます。Ubuntu 搭載の PC を使っているなら、ここにあるパッケージを PC にもインストールしておけば、開発ソフトウエアのかなりの部分を PC 上で検証できます。

最初に現在のパッケージ情報を更新し、最新の状態しておきます。

```
$ sudo apt-get update
  :
$ sudo apt-get upgrade
```

試しに I2C バス関連のツールをインストールしてみましょう。

```
$ sudo apt-get install i2c-tools
```

3.4.1 samba

Windows PC からもフォルダを操作できるようにするため、samba をインストールします。必要なディスク領域を表示して、インストールするかどうか聞かれるので、y と応えるとインストールが始まります。この手順は以下でも同じです。インストールが終わったら、バージョンも確認してみます。次に設定ファイルを編集します。

```
$ sudo apt-get install samba
  :
$ smbd -V
Version 4.13.13-Debian
$ sudo vi /etc/samba/smb.conf
```

smb.conf ファイルの先頭に近いところ（29 行目あたり）に以下のような行があります

```
Workgroup = WORKGROUP
```

これは Windows ワークグループ名のデフォルト値ですが、異なった名前を使っている場合には WORKGROUP の部分を合わせてください。

それから、ファイルの最後に以下の行を追加します。chuji のところは、自分のユーザ名にしてくださいね。

```
[chuji]
 comment = folder of chuji
 path = /home/chuji
 guest ok = yes
 read only = no
 browsable = yes
 force user = chuji
```

カギカッコで囲ったテキストがネットワークフォルダ名として表示されます。コメントには意味がありません。path はワークグループに開放するディレクトリ、force user は、他から接続したときに、そのユーザ名でログインすることを強制します。大事なのは read only = no で、外部からの書き込みを許すための指定です。設定をセーブしたら、samba をリスタートさせます。

```
$ sudo service smbd restart
```

これで Windows PC からでも、エクスプローラの「ネットワーク」から Raspbuggy→chuji で中身が表示されるようになります。PC で既にエクスプローラが起動されているときは、再起動してください。Ubuntu のファイルマネージャーでは、「他の場所」をクリックしたら、サーバーのアドレスに smb://<Raspberry Pi の IP アドレス>を入力し、「サーバーへ接続」をクリックします。PC のユーザ名が異なる時は、Raspberry Pi にログインするためのユーザ名とパスワードを聞かれます。

3.4.2 emacs

Linux の強力なエディターである emacs をインストールしました。これは私の好みのエディターというだけなので、必ずしも必要なわけではありません。Raspberry Pi ZERO には、ちょっと負荷が重いのですが、何とか使えます。

```
$ sudo apt-get install emacs
   :
$ emacs --version
GNU Emacs 27.1
Copyright (C) 2020 Free Software Foundation,
Inc.
GNU Emacs comes with ABSOLUTELY NO WARRANTY.
You may redistribute copies of GNU Emacs
under the terms of the GNU General Public
License.
For more information about these matters, see
the file named COPYING.
```

3.4.3 Python ツール

Python が使うツールとパッケージをインストールします。python3-pip は Python のパッケージ管理をするツールです。

```
$ sudo apt-get install python3-pip
```

以前と違って、python コマンドで Python3 が起動されるようになりました。前のプロジェクトで行っていた、シンボリックリンクの変更は不要です。

3.4.4 NODE.JS

Node.js は、Web サーバーを JavaScript で実現するためのパッケージです。残念ながら Raspberry Pi ZERO や初期の Raspberry Pi で使われている CPU（armv6l）向けのパッケージは、V.11 の後は更新されなくなりました。他のパッケージとの組み合わせが問題になることがあったので、『非公式』な最新版を使います。「何が起こっても知らないよ～っ」という版で、私的な用途なら自己責任で使えということですが、今のところ不都合は見つかっていません。

非公式ダウンロードサイト（https://unofficial-builds.nodejs.org/download/release/）から、最新バージョン（私がダウンロードしたのは 19.3.0）番号をクリックすると、CPU 毎のパッケージ一覧が表示されます。名前の最後の方が linux-armv6l.tar.gz となっているファイルをダウンロードします。なお、armv6 の次の文字は数字の 1 ではなく、アルファベットの l（エル）なので間違えないようにしてください。ダウンロードしたものを、smb 経由で Raspberry Pi（私の場合はホームディレクトリ）に転送します。または、Raspberry Pi に SSH 接続して次のコマンドを実行しても同じことができます。

```
$ wget https://unofficial-
builds.nodejs.org/download/release/v19.3.0/node-
v19.3.0-linux-armv6l.tar.gz
```

wget や tar の詳細な説明は省略します。

このファイルを tar コマンドで解凍し、/usr/local/bin に実行ファイルをコピーします。cp コマンドの-R オプションは、その下のディレクトリを含めてコピーするという意味です。

```
$ tar -zxvf node-v11.15.0-linux-armv6l.tar.gz
$ cd node-v19.3.0-linux-armv6l
$ sudo cp -R * /usr/local
```

まだ/usr/local/bin のファイルが実行できない（パスが通っていない）と思うので、以下のコマンドを実

行しておきます。~/.bashrc の末尾に同じ内容を追加しておけば、ログインしたときから node などが使えるようになります。

```
$ export PATH=$PATH:/usr/local/bin
```

念のため、バージョンを確認しておきましょう。node は Node.js 本体、npm は Node.js 専用のパッケージ管理ソフトです。

```
$ node --version
V19.3.0
$ npm -version
9.2.0
```

3.4.5 Socket.IO

Socket.IO パッケージは、ブラウザと Web サーバーとの間で通信をするためのパッケージです。通信に使えるプロトコル（通信規約）は何種類もあるのですが、Socket.IO は最適のプロトコルを選んでくれ、統一的なインターフェースから使うことができます。前のプロジェクトでもずっと採用してきました。

Socket.IO は npm を使ってインストールします。

```
$ sudo npm install socket.io
added 21 packages in 38s
```

3.4.6 Redis

Redis（REmote DIrectory Server）はメモリ上にデータベースを作るためのパッケージです。今までのプロジェクトでは 、そのごく一部である FIFO（First-In-First-Out）機能を使って、プロセス間で通信するために使います。同じコンピュータ上のプロセス同士で通信をする方法は、これ以外にもありますが、この方法が早くて便利なので、前のプロジェクトから採用しています。

まず、Redis サーバーをインストールします。この段階でサーバーは自動的に起動されるようになります。

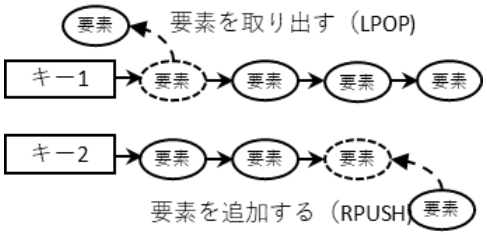
```
$ sudo sudo apt-get install redis-server
```

次に、JavaScript と Python から Redis を使うためのパッケージを、それぞれ pip と npm を使ってインストールします。

```
$ sudo pip3 install redis
:
Successfully installed deprecated-1.2.13 redis-
4.0.2 wrapt-1.13.3
$ npm install redis
```

```
+ redis@4.5.1
updated 1 package in 12.632s
```

メモリ上の Redis データベースを FIFO（First-in-First-out）として使う方法を説明します。



キー付きリストによる FIFO 実現

上の図のように、データベース上でキー（名前）ごとにリストを作り、右端から要素を追加（right-push）していき、取り出すときは左端から取り出（left-pop）します。取り出しはブロッキング型にします。空のリストから要素を取り出そう（BLPOP: blocking left-pop）とすると、その時点でプロセスの実行が休止（ブロック）し、リストに新しい要素が付け加わったときに再開されます。

いっぽう Node.js 用の JavaScript では、BLPOP の処理は要求を登録するだけで、次の命令に進んでいきます。受信があると割り込み処理が実行されるので、複数の FIFO を同時に待ち受けることができます。なお、このパッケージは、バージョン 4 から関数名に大文字（redis.rpush ではなく redis.RPUSH）を使うようになりました。小文字で呼び出そうとすると、「そんな関数はない」といって叱られます。

Redis サーバーはシステムと一緒に立ち上がるので、RPUSH する側が立ち上がっていなくても、BLPOP を行うことができる（データがないのでブロックします）ので、プロセスを立ち上げる手順や時間を考慮する必要がありません。実際の Redis サーバーへの要求は、TCP（ポート 6379）を通して行うのですが、Redis ライブラリを通すと、ずっと手軽に使えます。

Redis FIFO に馴染みのない方は、秘書ロボット・プロジェクトで作成した pusher.py と popper.py を使って、動作の感覚をつかんでみてください。

今回は、プロセス間の共有変数の保存場所としても Redis を使用します。一方のプロセスが書き込んだ（set）テキストを、他のプロセスが読み出す（get）ことで、最新の値を取り出すことができます。測定値の表示を更新するのに使いました。

3.4.7 pigpio

今回のプロジェクトから、GPIO ライブラリとして SMBUS モジュールより自由度の高い pigpio ライブラリ（http://abyz.me.uk/rpi/pigpio/index.html）を採用しています。実際の I/O 処理は高速で繰り返し実行されるプロセス（デーモン）が行い、他のプロセスは通信経由で命令を渡すようにします。その結果、どのポートでもソフトウエア PWM が実現できる、複数のプロセスから同時に GPIO が操作できるなどのメリットが発生しました。また、（スーパーユーザではなく）一般ユーザが GPIO 操作を行うことができるようになり、sudo 付きの命令は要らなくなります。

ライブラリは C 言語のソースコードとして配布されているので、make で実行型を作ります。以下では 2021 年 12 月現在の最新版 v79 をインストールしています。

```
$ wget
https://github.com/joan2937/pigpio/archive/refs/tags/v79.tar.gz
$ tar -zxvf v79.tar.gz
$ cd cd pigpio-79
$ make
$ make -n install >install.log
$ sudo make install
```

GPIO を使う前に、I/O 処理部（デーモン）を起動しておきます。/etc/rc.local に記述（sudo は不要）しておけば、システムが立ち上がったときに起動されるようになります。

```
$ sudo pigpiod -s 10 &
```

上の例では、I/O チェックの間隔を 10μs に設定しています（デフォルトは 5μs）。CPU パワーの貧弱な Raspberry Pi ZERO の負荷を重くしないためです。top コマンドで pigpiod の CPU 使用率を測ってみると、I/O 処理を何も行っていなくても、5μs の時には 8.5%を使っていました。10μs では 6.5%と、少しだけ改善します。自律走行車では 10μs で動かすことにします。

3.4.8 pysigset

Linux のシグナルによる割り込みを、一時的に待たせるライブラリをインストールします。Linux が持っている機能（sigprocmask(2)という OS コール）を Python から使いやすくするライブラリです。

```
$ sudo pip install pysigset
```

3.4.9 gauge.js

Web 画面に計器盤（ゲージ）を表示するためのライブラリです。類似の機能を提供するライブラリは何種類かありますが、使いやすさを重視して、ウクライナのスタドニク氏のプロジェクトから Canvas Gauge を借用しました。丸型と直線型の表示器が作れますが、自律走行車では速度計と操舵量を表示する丸型計器を使います。

```
$ sudo npm install canvas-gauges
```

コラム　開発手順について

この本で説明に使っているソフトウエアの開発手順は『ウオーターフォール（滝）』モデルといって、構想から検証までをトップダウンで順番に実行するものです。大規模な開発を大人数で行うときの手法で、スケジュールを守ったり、要員を確保したりするのに向いています。これに沿って、全体像から個々の部品に分解していく様子を説明すると分かりやすいと思ったからです。ただ、実際の作業が、この通りに進んでいたわけではありません。

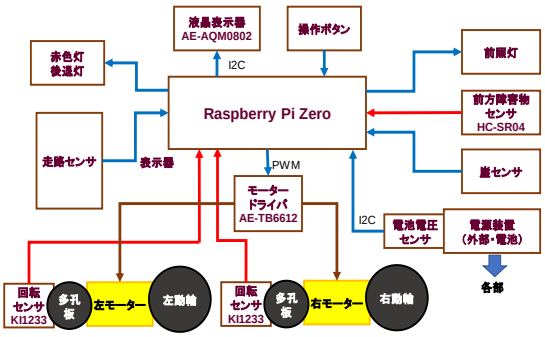
最近は『アジャイル』というアプローチがよく使われます。『機敏な』という意味の英語から来ており、小回りのきく開発手法です。小さな単位での実装と検証を繰り返すことで、開発期間を短縮したり、仕様変更に対応したりできます。

どちらも、チームでの開発に使われるものですが、趣味の一人プロジェクトでも意味があります。

ハードウエアの変更があったり、当初の目論見どおりにいかなかったりすると、せっかく検証を終えたモジュールでも改造が必要になることがあります。ときには、ソフトウエア構造を変えることすらあります。また、趣味という面では、ハードウエアをはやく動かしてみたいという欲望も大事です。こういう開発は、むしろアジャイル型に近いものになります。しかし、私としては「何をどういうアプローチ（考え方）でつくりあげたか」が大事で、「どういう過程で完成したか」はプライベートなものだと思っています。この本でウオーターフォール型に近い説明をしたのは、これが理由です。

4　ハードウエアの設計と組立

自律走行車のハードウエア設計について説明します。第 2 章で開発仕様と主要なハードウエアを決めました。その結果をまとめると、下の図のようになります。青い矢印は GPIO の信号を、赤い矢印は PICCOLO チップ経由で取り込むことを示しています。まだ機能ブロックの段階なので、購入するユニット以外は、どういう回路ユニットとして実装するか決まっていません。



ハードウエアの構成

4.1　処理時間の見積

ハードウエアの詳細を詰める前に、Raspberry Pi ZERO（Python）の処理時間を見積もっておきます。測定方法は右のコラムを参照してください。

測定する処理	処理時間	標準偏差	最短－最長
命令なし（測定機能）	14μs	1.1μs	12 - 16μs
代入	1μs	0.5μs	1 - 4μs
浮動小数点数の掛け算	3μs	0.6μs	2 - 5μs
平方根を求める	7μs	0.4μs	6 - 8μs
三角関数の演算	7μs	0.6μs	8 - 14μs
関数コール	9μs	0.4μs	8 - 10μs
GPIO ポート読み取り	1.1ms	0.4ms	0.8 - 2.5ms
GPIO PWM デューティ設定	1.3ms	0.4ms	1.0 - 2.6ms
GPIO I2C データ読み取り	1.8ms	0.4ms	1.4 - 2.8ms
Redis FIFO へプッシュ	3.2ms	0.3ms	2.9 - 3.9ms

処理時間の測定結果

かなりの数値演算を行っても、せいぜい 0.1ms 程度で処理できることが分かります。いっぽう pigpio ライブラリや Redis FIFO の処理は、OS を介して呼び出されるので、ms オーダーの時間がかかり、（OS の処理状態による）ばらつきも大きいことが判明しました。

このため、超音波センサの受信時間（10cm～1m で 0.6～6ms）をソフトウエアだけで処理する当初案は実用性が疑われる羽目になりました。30ms に一回 ON/OFF するロータリーエンコーダ（2 個なので、処理回数は 2 倍になる）は処理可能かもしれませんが、これにも対応できる測定チップ PICCOLO を使うことにしました。

後述するように、100ms 周期の処理内では、Redis へのアクセス（3～4ms かかる）回数を制限することにしました。

コラム　処理時間の見積

Raspberry Pi ZERO（Python）の処理時間を time ライブラリのパフォーマンスカウンタを使って調べました。測定プログラム test_pi.py は、プロジェクトファイルに含まれているので、参考にしてください。処理内容を一つだけ#define すれば、その処理時間を測定します。例えば

```
$cpp -DMULTIPLICATION test_pi.py | python
```

とすれば、浮動小数点数の掛け算にかかる時間を測定してくれます。

ここで問題になったのは、OS コールに伴う遅れでした。測定したい個所を time.perf_counter_ns のコール命令で囲み、出力の差を取ると、その間の経過時間を ns 単位で与えてくれます。コールの間に他の命令を挟まなければ、測定機能そのものに要する時間を測ったことになります。結果は 14μs 前後が得られますが、ときに異常に長くなる場合があります。これは、測定命令が OS（Linux）を呼び出すためです。OS をコールしてから、返ってくるまでの間に、他のプロセスを実行するときがあるので、時間がかかってしまったのです。これがいつ起こるかは、全く予測できません。

この影響を小さくする対策を取りました。処理時間が短いときは、同じ処理を繰り返すプログラムの実行時間を測り、結果を繰り返し回数で割ってやります。1000 回の繰り返し中に 1ms の遅れが出ても、影響は 1μs に抑えられます。for ループにかかる時間（2μs 程度）は別途測定しておき、結果から引いてやります。

OS コールを含む処理（GPIO と Redis）は、それ自体の処理中に遅れが入ってしまうので、測定結果を見ながら、「他の測定より異常に長い」結果は廃棄することで、影響を少なくしました。

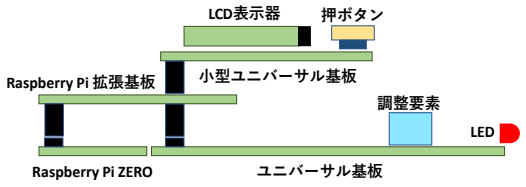
4.2　回路ユニットへの分解

購入ユニット以外の電子回路は、ユニバーサル基板にはんだ付けしてから、車台に搭載します。電子部品を選択する段階で、センサ以外の回路を 3 枚のユニバーサル基板に搭載することにしました。それは、

1. Raspberry Pi 用の拡張基板（秋月電子通商）
2. ユニバーサル基板（72×95mm）
3. 小型ユニバーサル基板（72×48mm）

1. は Raspberry Pi ZERO の 40 ピン拡張コネクタとインターフェースし、2. と 3. の基板と接続するのに使います。空いたスペースには、回路の一部を載せられます。3. には操作用に液晶ユニットとキースイッチを取り付け、回路の大部分は 2. に載せることにします。2. は車台を少しはみ出すので、角を丸くします。これ以外に、崖と走路のセンサは、車台の裏（下）側に取り付けるので、もっと小型のユニバーサル基板を使います。車輪の回転センサも裏側に取り付けますが、コネクタ付きなので基板は不要です。

3 枚のユニバーサル基板の接続は以下の側面図（左が前）のようにしようと考えました。3 枚の基板を結ぶのに、足の長い Raspberry Pi 用のスタッキングコネクタ（40 ピン）を使います。こちらを「システムコネクタ」、Raspberry Pi ZERO と接続する方を「GPIO コネクタ」と呼ぶことにします。基板を固定するには、プラスチック製のスペーサー（六角棒の両端に M3 のネジ穴とボルトが付いている）を使い、基板の間隔を調整します。



回路ユニット間の接続

4.2.1 GPIO の割り当て

Raspberry Pi ZERO の GPIO をどう使うか決めます。大部分の信号はハードウエアの構成から決められますが、一部は回路設計の段階で付け加えました。次の表のように割り当てました。ハードウエア PWM と I2C 以外は自由に指定できます。一時期はほとんどの GPIO 資源を使ってしまいましたが、PICCOLO チップを採用したことで、少し余裕ができました。*印をつけた GPIO#15 のシステムクロックは、当初 100ms の割り込みを発生させるつもりで割り当てましたが、後述するように、Linux のシステム割り込みで十分なことが分かったので、実際には使っていません。

GPIO		信号名	用途
0	ID_SD		（使用不可）
1	ID_SC		（使用不可）
2	I2C SDA	I2C SDA (I/O)	I2C 通信データ
3	I2C SCL	I2C SCL (I/O)	I2C 通信クロック
4	CPCLK	BEAMER (O)	前照灯
5		ACT_ OPTO(O)	反射センサ動作制御
6		SELECT (I)	SELECT キースイッチ
7	SPI CE1	LT5 (I)	走路センサ（左端）
8	SPI CE0	LT4 (I)	走路センサ（中左）
9	SPI_MISO	LT1 (I)	走路センサ（右端）
10	SPI MOSI	ACT_TRIP(O)	走行計の動作制御
11	SPI SCLK	LT3 (I)	走路センサ（中央）
12	PWM0	PWM_R (O)	右モーターPWM
13	PWM1	PWM_L (O)	左モーターPWM
14	UART TXD	US TRIG (I/O)	超音波送信
15	UART RXD	SYS_CLK (I/O)*	システムクロック
16		AIN2 (O)	右モーター制御 2
17		BACK (O)	後退灯
18	I2S CLK	RIGHT (O)	右折灯
19	I2S LRCLK	NEXT (I)	NEXT キースイッチ
20	I2S ADC	BIN1 (O)	左モーター制御 2
21	I2S DA	BIN2 (O)	左モーター制御 1
22		ISR (O)	割り込み処理確認用
23			
24		CLIFF (I)	崖センサ
25		LT2 (I)	走路センサ（中右）
26		AIN1 (O)	右モーター制御 1
27		LEFT (O)	左折灯

GPIO の割り当て

信号名は回路図で使っている記号で、(I) などの説明は、GPIO を入力（I）あるいは出力（O）端子として使っていることを示します。

同じグループに属する信号の GPIO 番号が一見バラバラなのは、GPIO コネクタ上で位置が近いピンを使っているせいです。実際の回路図を見ると納得できると思います。

4.2.2 システムコネクタのピン配置

3 枚の基板を結ぶシステムコネクタのピン配置を次の表のように決めました。

信号名	ピン番号		信号名
GND	1	2	GND
5V	3	4	5V
3.3V	5	6	3.3V
SDA	7	8	SCL
GND	9	20	GND
US_TRIG	11	12	US_ECHO
BACK	13	14	
RIGHT	15	16	LEFT
GND	17	18	GND
R_SPEED	19	20	L_SPEED
CLIFF	21	22	ACT_TACHO
LT1	23	24	LT2
LT3	25	26	LT4
LT5	27	28	ACT_OPTO
GND	29	30	GND
SELECT	31	32	NEXT
SW1	33	34	SW2
5V	35	36	5V
3.3V	37	38	3.3V
GND	39	40	GND

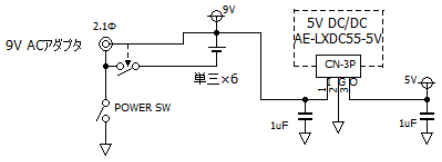
システムコネクタのピン配置

4.3　電子回路の設計

ここからは電子部品の仕様を示しながら、回路設計を行っていきます。回路の動作確認には、次章で設計するソフトウエア（デバイスハンドラとテストプログラム）が必要なので、使うプログラムと確認事項だけを説明します。各プログラムを使う前には、シミュレーションで検証しておくことが必要です。それから、ひとつずつ回路を動かしていきます。

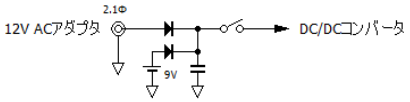
4.3.1 電源ユニット

電池と外部電源（どちらも 9V）から 5V を作ります。DC ジャックは、プラグを差し込むと電池側の接続が切れるタイプを使っています。DC/DC コンバータはユニバーサル基板に取り付けました。この回路の試験はテスター（電圧計）だけで行えます。



電源ユニット回路図

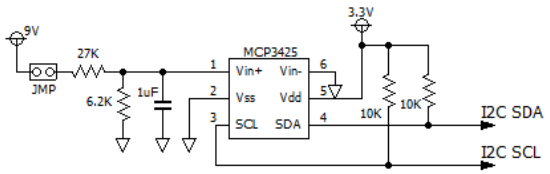
実はこの回路の問題に、後になって気が付いてしまいました。DC ジャックのコネクタは break-before-make といって、プラグを差し込みだすと電池との接続が切れ、それから AC アダプタに接続されます（逆も同様）。両方の電圧が同じでないときのための安全策なのですが、プラグを抜き差しすると電源が落ち、システムがリブートしてしまいます。これを防ぐには、出力電圧が電池（9V）より高く、DC/DC コンバータの最大入力電圧（14V）より低い AC アダプタ（例えば 12V）を使い、ダイオードで切り替える方法（下図参照）が一般的です（あとの祭り）。



4.3.2一次電圧センサ

A/D コンバータは、I2C インターフェースを持った MCP3425 を使いました。おもな仕様を下表に示します。表面実装型ですが、DIP 型プリント板に実装したモジュールが入手できます。差動入力として±基準電圧の範囲を、符号付き 16 ビットに変換してくれます。

作動入力のマイナス側は接地しますが、実際の測定値（6～9V）の範囲では、非線形性が問題になることはありません。最大 11V 程度まで測定したいので、抵抗で分圧します。図の分圧では、測定範囲 0～10.97V、16 ビット測定の場合 0.335mV/ビット（2.048/32768 × (27k+6.2k)/6.2k）になります。



一次電圧センサ回路図

入力側にあるジャンパは、電池以外の電圧を測定して、A/D コンバータの精度を確認するために設けました。普段はピンで短絡しておきます。

回路が正しく結線されていることは、A/D コンバータが I2C バス上（アドレス 0x68）に存在すれば確認できます。次のコマンド（Raspberry Pi OS に含まれている）を実行すれば、"68"のところに存在が表示されます。

```
$ sudo i2cdetect -y 1
```

一次電圧センサの動作確認は（ソフトウエアモジュールの検証を終えてから）test_adc.py を実行することで行います。

```
$ cpp -DBCM2835 test_adc.py | python cleanfile.py
| python
```

このプログラムは一次電圧センサから電圧を読み取って表示します。ジャンパーピンでジャンパをショートさせると 9V が、ADC 側のピンに 3.3V と 5V 電源を接続するとそれぞれの電圧が表示されます。デジタル電圧計でも測定して比較した結果を下に示します。抵抗のばらつき（1%）に起因すると考えられる、-0.7%程度の誤差で測定できており、期待どおりであることが分かります。

ジャンパ接続先	電圧実測値	ADC 指示値
短絡（9V 電源）	8.93V	8.88V
5V 電源	5.02V	4.98V
3.3V 電源	3.30V	3.27V
GND	0.00V	0.00V

一次電圧センサの試験結果

4.3.3 前方障害物センサ

超音波式障害物センサ HC-SR04 は電源電圧・入出力レベルとも 5V なので、3.3V 系の Raspberry Pi との間でレベル変換が必要です。他の製作記事の中には、3.3V 系の送信信号をそのまま HC-SR04 の入力に与え、出力（受信信号）を抵抗で 3/5 に分圧して GPIO ポートに与えているものもあります。MOS トランジスタ同士なので何とかなるのでしょうが、今回はトランジスタを使って、きちんとレベル変換することにします。

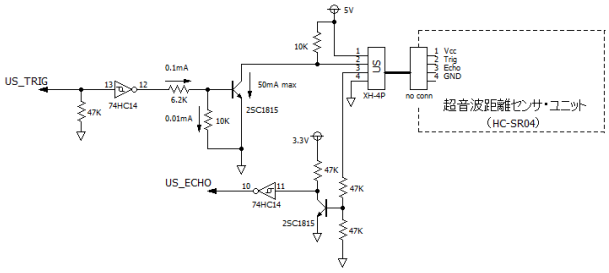
あまり高速の回路ではないので、小信号 NPN トランジスタであれば何でも使えます。私が使った 2SC1815 の主要特性を次の表に示します。

パラメータ	特性値
最大コレクタ・エミッタ電圧 V_{CEO}	50V
最大コレクタ電流 I_C	150mA
最大損失 P_D	400mW
直流ゲイン h_{FE}	120～700
トランジション周波数 f_T	80MHz

2SC1815 の主な特性

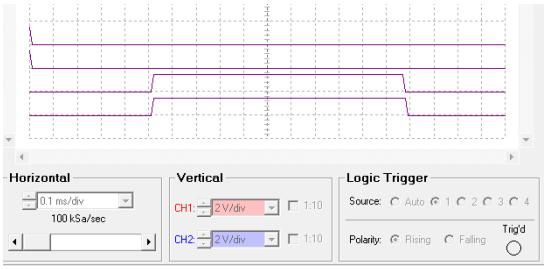
トランジスタをオンにするため、ベース電流を 0.1mA、その 10%をバイパスするくらいでベース抵抗を選びます。エコー側のコレクタ電流は 1mA としました。トリガ側のコレクタ抵抗は不要（センサユニット内 10kΩ でプルアップしている）のはずだったのですが、H レベルにならなかったので、追加しました。不良品だったのかもしれません。

電源投入時には GPIO ピンがハイインピーダンス状態になっているので、トリガ側をプルダウン（送信しない状態に）しています。シュミット回路は必ずしも必要ではありませんが、他の回路で使った残りのゲートがあったので、正論理になるようにしました。



前方障害物センサ回路

評価プログラム test_us.py を走らせ、各部の信号レベルを測定すると、以下のような結果が得られます。



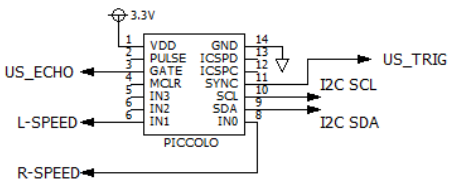
前方障害物センサの信号

信号は上から

- US_TRIG（3.3V レベル）
- モジュール Trig 入力（5V レベル）
- モジュール Echo 出力（5V レベル）
- US_ECHO（3.3V レベル）

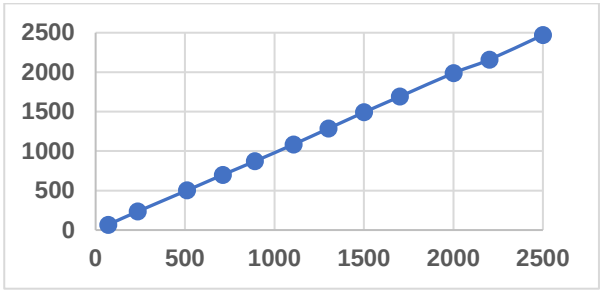
です。US_TRUG として 10μs 以上のパルスを与えると、500μs のむだ時間後に超音波が発生し、US_ECHO が立ち上がります。障害物からの反射波を検出すると US__ECHO が立下ります。この時間差（伝搬時間）を T とすると、T = 2×距離÷音速なので、音速を 340m/s として、伝搬時間から距離を求めることができます。

US_ECHO 信号が H レベルになっている時間を、PICCOLO チップで測定します。次のような回路図（走行計入力も含む）になります。



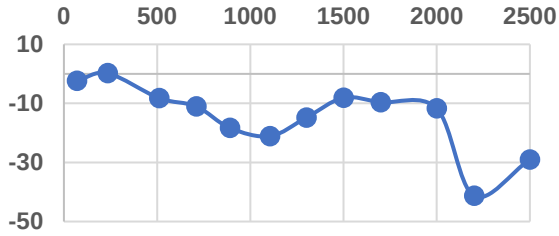
超音波伝搬時間測定回路（走行計への接続を含む）

巻尺で壁までの距離（横軸）を測った結果（精度10mm）と、テストプログラムの指示値（縦軸）とをプロットすると、次の図のようになりました（単位は mm）。



壁までの距離（巻尺測定値とセンサ指示値）

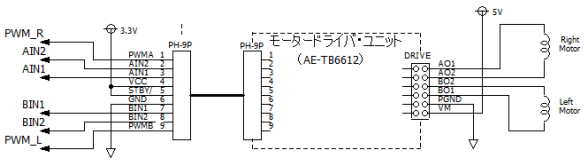
もう少し詳しく見るため、縦軸に「指示値 - 巻尺測定値」をプロットすると、次のようになります。このときの室温は 20.5℃（音速は 343m/s）だったので、1%程度マイナスの結果になっているのは納得できます。壁の接近を知るのには十分すぎる精度です。



センサ指示値と巻尺測定値との差（mm）

4.3.4 モータードライバ

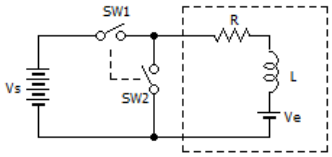
モータードライバ AE-TB6612 は、GPIO に直結できます。コントローラ TB6612 の入力ピンは内部でプルダウンされている（モーターはスタンバイ状態）ので、安心して使えます。直結できるので、拡張基板から制御信号を取り出すことにしました。



モータードライバ回路

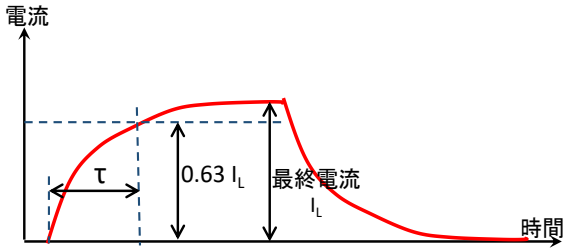
詳しい制御アルゴリズムは次章で説明しますが、ハードウエアとしてここで検討しておきたいのは、パルス幅制御（PWM）の周波数の決め方です。

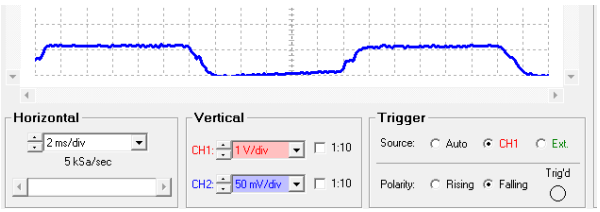
次の図の点線内はモーターの等価回路を示しています。R は抵抗成分、L はインダクタンス、V_e はモーターが回っていることによる（回転速度に比例した）起電力です。ドライバ回路は電源 V_s とスイッチで表せ、SW1 と SW2 は交互にオンになります。



モーターとドライバの等価回路

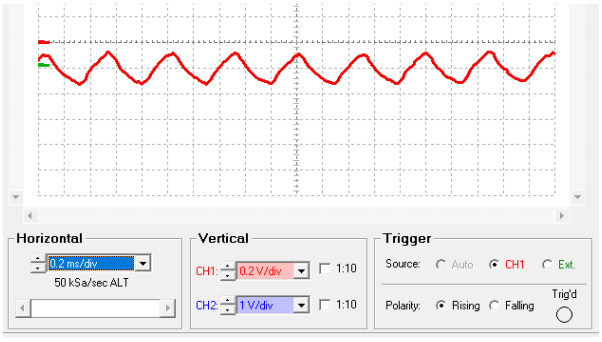
htest_motor1.py は、ドライバをゆっくり（例えば 1Hz）でオン・オフさせます。このときモーターに流れる電流（直列に挿入した抵抗（例えば 1Ω）の両端の電圧）を観測します。電流波形は次の図のようになります（V_e は無視している）。





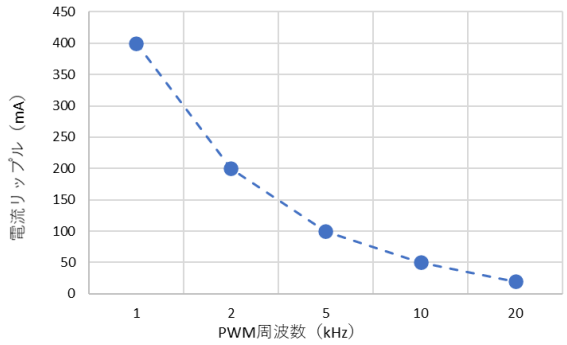
低速でスイッチング（上：説明図、下：波形例）

電流が増加しだしてから、最終的に安定した電流 I_L（$=V_s/R$）の 63%になる時間 τ（=L/R）を測ると、1.5ms でした。PWM 周期は、これより短くするのが一般的です。モーターに流れる電流は、この周期で増減を繰り返すので、短いほど変動（リップル）は少ないのですが、早すぎるとドライバのスイッチ損失が大きくなってしまいます。ドライバ素子の PWM 周波数は最大 100kHz なので、1/τ の 5 倍以上として周波数 5kHz 位が良さそうです。



モーター電流のリップル（PWM： 2kHz、デューティ 50%）

PWM 周波数をいろいろ変えて、デューティ 50%（いちばん振幅が大きい）のリップルを測ってみると、次の図のようになりました（周波数に逆比例）。



PWM 周波数と電流リップル

オシロスコープのような測定器がないときは、モーターの駆動音で推測します。1Hz では、回転と停止がはっきり聞こえます。1kHz まで上げると、あまり断続して聞こえない気がしますが、まだ駆動電流がゼロになる時間があります。さらに 5~10 倍にすればいいことが分かります。ちなみに、リップルが常に悪というわけではなく、トルクを取り出すために敢えて大きくすることもあるそうです。

DC/DC コンバータは 2MHz で制御しているので、10kHz 以下の PWM が 5V 電源に影響することはありません。ソフトウエア PWM の分解能は pigpiod の動作周期（10μs）なので、分解能が足りません。ハードウエア PWM を使用します。

4.3.5 走路センサ

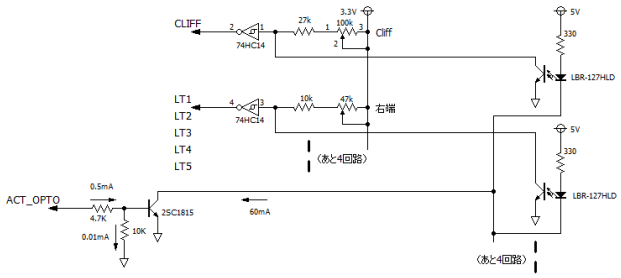
走路の検出には、赤外 LED とフォトトランジスタをプラスチック容器に入れた反射センサ（LBR-127HLD）を使います。

特性	仕様
LED 順方向電流	最大 60mA、推奨 20mA
LED 発光波長	940nm
LED 順方向電圧降下	1.2V_{typ}
指向性（半値角）	35 度
出力トランジスタ暗電流	100nA 以下
出力トランジスタ明電流	0.2mA 以上（LED 順電流 20mA）

反射式光センサ LBR-127HLD の主な特性

発光電流は少なめの 10mA にしました。LED を 5V で駆動するとして、直列抵抗は (5V-1.2V)/10mA→330Ω にします。

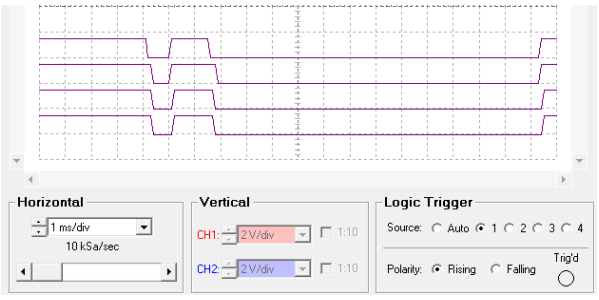
コレクタ抵抗は 3.3V/0.1mA=33kΩ 以上必要ですが、路面の反射率でも変わるので、調整できるように 10kΩ 固定抵抗+50kΩ 可変抵抗を使うことにします。これをシュミット回路付きのインバータで受けます。反射光を検出したとき、GPIO ポートの入力は H レベルになります。



走路センサと崖センサ回路

LED の消費電力を削減するため、走路の検出をするときにだけ光らせるようにしました。ACT_OPTO 信号でトランジスタをオンにしたときだけ、LED に電流を流します。この回路は崖センサと共用します（各自律走行プログラムは片方だけを使う）。トランジスタのベース抵抗は小さくして、6 個の LED を駆動できるようにしています。LED をすべて直列接続すれば、もっと消費電流を減らすことができたのですが、順方向電圧降下が積み重なってしまって、目的どおりの駆動電流にするのが困難になってしまいます。

htest_opto.py で動作を確認します。信号は上から ACT_OPTO、CLIFF、LT1（右端）、LT5（左端）です。前の長いパルスは走路センサを、その後のパルスは崖センサを検出しています。両方の処理に合計 7ms かかっていることが分かります。



走路センサと崖センサの信号波形（反射光のある場合）

車台をひっくり返して、走路センサの上に白い紙を近づけ、表示が黒（B）から白（W）に変わるように可変抵抗を調整します。紙を右側から近づけていくと LT1 から LT5 まで順番に変わっていくこと、可変抵抗でしきい値を調整できることを確認します。

ここでちょっとした問題が判明しました。日当たりのいい部屋で動作させると、走路や崖の有無にかかわらず「反射光あり」になってしまいます。フォトトランジスタが外部の光（『迷光』といいます）に反応してしまったのです。考えられる対策は

1. 光学的な方法で迷光を減少させる
2. 電子的な方法で照明光と迷光を区別する
3. 妥協する（センサが使えないときがある）

くらいです。1 はさらに

1.a. カメラのレンズフードのように筒状に被う
1.b. カメラのフィルタのように、波長を選ぶ

このうち 1.a は構造が複雑になってしまいます。1.b について考えると、フォトトランジスタの受光感度は 750nm～1300nm（中心波長 940nm）ですが、太陽光はこの範囲の赤外線が多く含まれているので、分離は困難です。蛍光灯や LED 照明下では、この問題はあまり起こりません。2.は LED の発光を変調し、受光信号から同位相の信号だけを取り出す手法ですが、そこまでやることもないと思います。簡単なフードをつける程度にし、直射日光下では崖センサが働かないこともあり得ると妥協することにしました。

4.3.6 崖センサ

崖センサも走路センサと同じ素子を使います。路面の反射率が低いこともありうるので、走路センサの半分の受光量でも検出できるように、抵抗値を 2 倍にしておきます。路面を検出している間の GPIO 入力は H、崖に出会うと L になります。動作試験は走路センサと一緒に行います。

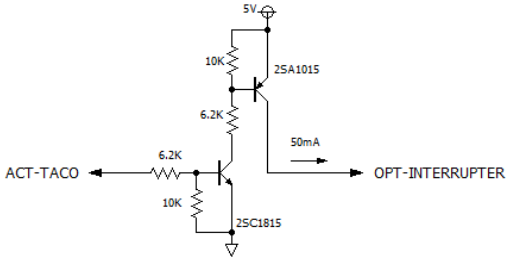
4.3.7 走行距離計（動輪回転センサ）

動輪の回転数を知るのには、同じ回転軸に取り付けた穴あき円盤（ロータリーエンコーダ）を使います。穴の両側に LED と光センサを取り付けた、光インターラプタを置き、光量の変化を電気パルスとして検出します。

特性	仕様
動作電源電圧	5.0V±10%
供給電流	最大 25mA
受光部スリット幅	0.5mm
出力形式	オープンコレクタ
オン時コレクタ電流	80mA 以下（推奨 15mA 以下）

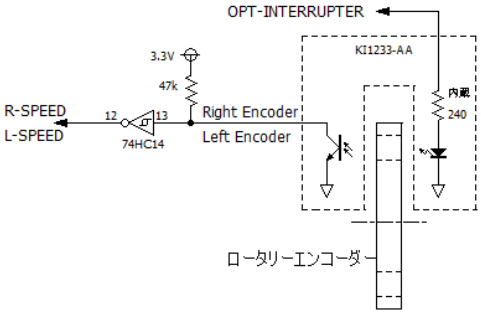
光インターラプタ KI1233 の主な仕様

左右で最大 50mA の消費電流は無視できないので、使うときだけ電源を供給できるようにします。まず NPN トランジスタで 5V にレベル変換し、PNP トランジスタ 2SA1015 を ON/OFF させることで電源スイッチを実現します。ベース回路の定数は障害物センサと同じです。2SA1015 の特性は、コンプリメンタリ NPN トランジスタ 2SC1815 とほぼ同じ（電圧・電流の向きは逆）です。



走行計センサ駆動回路

動作時もコレクタ電流を少なくして、シュミット回路で整形します。出力は PICCOLO チップの汎用入力につなぎます。



走行計回路（片輪分）

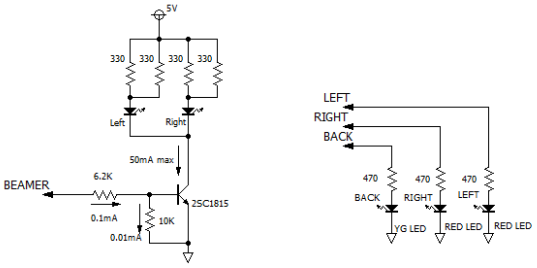
4.3.8 前照灯

前照灯（ヘッドライト）は高輝度の白色 LED（OSPW5111A-Z3）を使いました。

特性	仕様
最大順方向電流	30mA（パルス 100mA）
発光強度	25000m カンデラ（I_F=20mA）
順方向電圧降下	2.6V～3.6V（3.1V$_{typ}$）

白色 LED OSPW511A-Z3 の主な特性

この LED は順方向電圧降下が最小でも 2.6V と大きいので、3.3V 系で安定して光らせるのは困難です。超音波センサのインターフェースと同じ回路を使って、5V 系で駆動します。一灯あたり 10mA 流すことにして、負荷抵抗= (5V – 3.1V – 0.2V)/10mA = 170Ω（330Ω を 2 本並列接続）にします。



LED 駆動回路

回路図には赤色灯と後退灯も同時に示してあります。これらの動作確認には、BCM2835 を#define して test_led.py を実行します。

4.3.9 赤色灯、後退灯

自動車の後退灯は白色ですが、趣味的な装飾に白色 LED 駆動回路を設けるのは贅沢なので、赤色 LED（OSR5JA3Z74A）と同じシリーズの黄緑色 LED（OSG8HA3Z74A）を使って、GPIO で直接駆動することにしました。

特性	仕様
最大順方向電流	30mA（パルス 100mA）
順方向電圧降下	1.8V～2.6V（$2.1V_{typ}$）

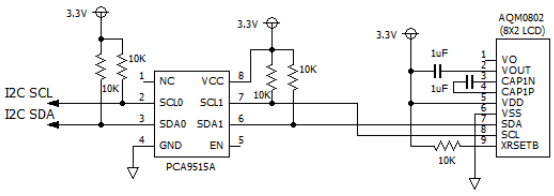
赤色 LED と黄緑色 LED の主な特性

これらの LED は、点灯したことが分かればいいので、順方向電流は 2mA と少なめにしました。負荷抵抗=（3.3V-2.1V）/2mA→470Ω にします。

動作確認は前照灯と同時に行えます。

4.3.10 液晶表示器

操作用の液晶表示器（LCD）を用意したいのですが、車両に載せるため、あまり大型のものは使えません。I2C インターフェースのある 8 文字×2 行の AQM0802A-RN-GBW（外形寸法は横 30mm×縦 19.5mm）を選びました。このシリーズの制御回路は Raspberry Pi ZERO の I2C SDA を直接駆動できないので、バスリピーターPCA9515A を間に挿入します。

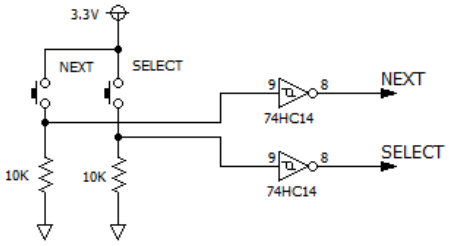


液晶表示器周辺回路

回路の動作確認には、BCM2835 を#define してから test_lcd.py を使います。Hello！ World!と二行にわたって表示できれば OK です。

4.3.11 操作用キースイッチ

操作のため、NEXT と SELECT という 2 つのキースイッチを用意し、LCD のそばに置きます。プルダウン抵抗とシュミット回路はもう一枚のユニバーサル基板に取り付けました。



キースイッチ回路

動作確認には、BCM2835 を#define して、test_keys.py を走らせます。ボタンを押して表示が一回だけ変われば OK です。すぐに放すと検出されないときがありますが、長押しは無視するので、しっかり押してください。

4.3.12 全回路図

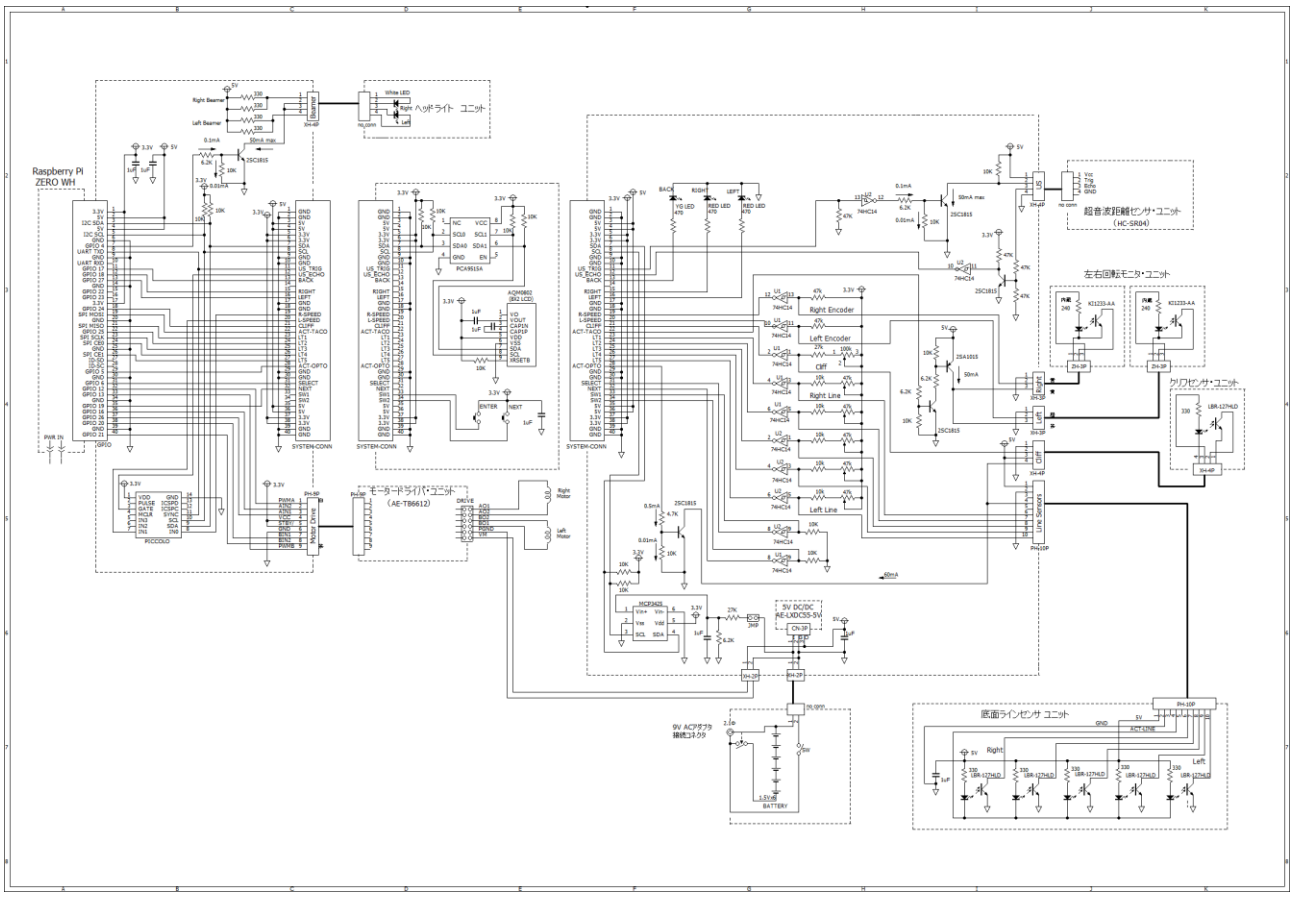
ここまでで、全部の回路ができ上がりました。基板間の接続を含めた全回路図を次ページにまとめます。

あとで後悔することになった失敗を告白します。回路図の左半分にある下側ユニバーサル基板の下側、電池からの入力（9V）とモーターへ供給する電源出力（5V）に同じコネクタを使っています。組付けたり外したりを繰り返しているうちに、誤って 5V の方に 9V を接続してしまったのです。Raspberry Pi ZERO が触れないほど熱くなり、壊れてしまいました。こういう個所には、違うコネクタを使って、誤接続を避ける工夫が必要でした。

コラム　海外の交差点 1：4-way stop

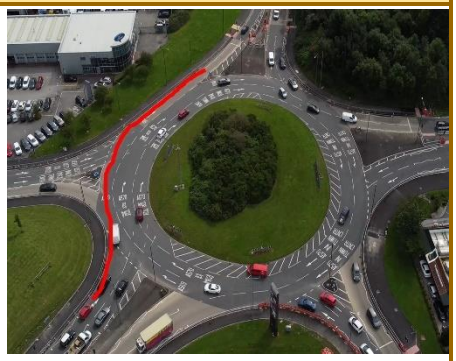
アメリカやカナダの、あまり交通量の多くない交差点で 4-way stop という標識を見かけます。交差する道路の数によっては 3-way だったり All-way だったりします。手前に予告標識がありますが、信号はありません。交差点直前で一時停止し、最初に到着した車から交差点に進入するルール（同時の場合は右側が優先するなど細かいルールもある）で、良く守られています。もっとも『自己責任』の国なので、安全が確認できれば減速するだけの人もありますが。円滑な交通と『早い者勝ち』精神が根底にあります。

信号機が要らないので日本でも普及すればいいと思うのですが、一時停止義務違反の取り締まりが好きな日本では、かえって円滑な走行を妨げかねないですね。



自律走行車の全回路図

コラム　海外の交差点 2：Roundabout



道路が交わる代わりに、円形の道路を回るようになっている場所をラウンドアバウトと言い、イギリスで特に多い交通制度です。信号機が不要、というより電気も自動車もなかった時代からありました。馬車は直角に曲がるのが難しいという制約もあったのでしょう。

ラウンドアバウト内にいる車両が優先するので、右側を確認しながらの進入（一時停止は必ずしも必要でない）には気を使います。また、大きなラウンドアバウトを回っていると、どの出口から出たらいいのか分からなくなることがあります。直前の標識を見て、何番目の出口なのか把握するには慣れが必要でした。間違ってしまっても、その先のラウンドアバウトで U ターンして戻ってくれば良いのですが、そこでも間違うと、もう迷子です。

片側一車線の田舎道では、交差点の中央に白丸を描いただけのミニ・ラウンドアバウトがあります。私の友人によると、昔の非舗装道路では交差点にマンホールの蓋のようなものを埋めて、button around と呼んでいたそうです。十字路で対向車と同時に右折するとき、日本では相手の直前を横切ります。しかしミニ・ラウンドアバウトでは（丸いマークがなくても）、相手の後ろに回り込むように右折します。知らずにいると事故に繋がりかねません。

4.4　ハードウエアの組み立て

回路図ができ上ったので、ハードウエアの組み立てを始めます。

4.4.1 電子回路のはんだ付け

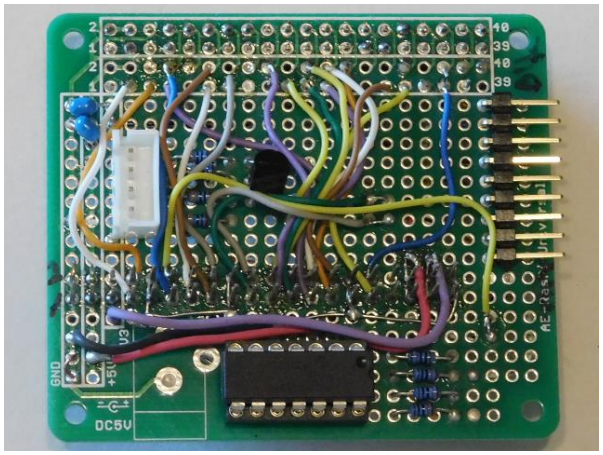
はんだ付けで組み立てる電子回路基板は 5 枚です。

1. Raspberry Pi 拡張基板
2. 下側ユニバーサル基板
3. 上側小型ユニバーサル基板
4. 走路センサ（小型基板）
5. 崖センサ（小型基板）

プリント板に電子部品を搭載するまえに、取り付け用の穴加工を行っておきます。配線が多いので、回路図を見ながら、ていねいにはんだ付けします。部品の位置を決めるときには、組み上げた後で他のユニットにぶつかったり、操作のじゃまになったりしないように気をつけます。

各基板の写真を次に示します。

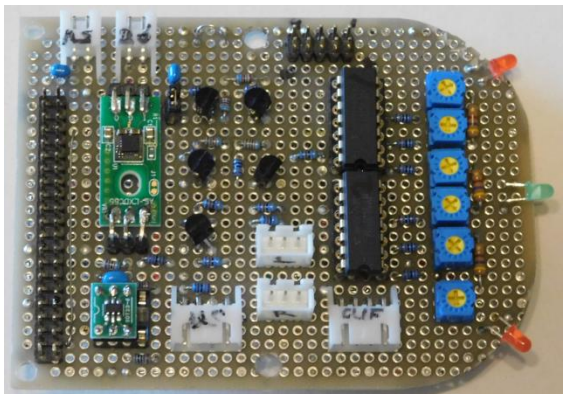
Raspberry Pi 拡張基板



Raspberry Pi 拡張基板

写真の左側にある白いコネクタは前照灯との接続用、右側の黒いピンヘッダーはモータドライブ・ユニットとの接続用です。写真下側の 14 ピン DIP IC が PICCOLO チップ（基板設計時にはなかったので、空いているところに後付け）です。その上側に 40 ピンのシステムコネクタです（基板の上側にピンが、下側にソケットが延びている）。

下側ユニバーサル基板



下側ユニバーサル基板

台車からはみ出す後ろ側の角を取りました。左側に見える小さな基板モジュールは、上側が DC/DC コンバータ、下側は A/D コンバータです。中央の黒い素子がトランジスタ、DIP 型 IC はシュミット・インバータです。

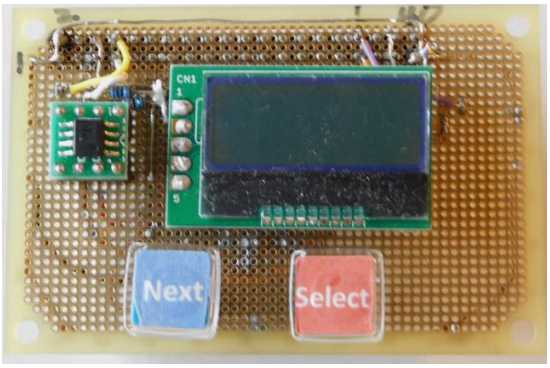
この基板は二階建ての下側になるので、コネクタはなるべく横向きのものにしました。調整用の半固定抵抗は、上側の基板と重ならない位置に置いています。

コラム　海外の交差点 3：Yield

優先道路と交わる信号のない交差点では、優先車両の走行を妨げないことと、安全を確認してから進入することが必要です。そのためアメリカやカナダでは YIELD（道を譲れ）という標識が優先道路との交差点手前にあります。イギリスの標識には GIVE WAY と書いてあります。先の 2 点が重要で、一時停車は必ずしも必要ありません。ここでも円滑な交通と自己責任が重要視されています。

これに慣れてしまい、帰国直後に失敗したことがあります。優先道路との T 字路を左折するときです。後ろに車が続いており、右側の見通しが悪かったので、這うような速度で右が見える位置まで進みました。曲がったとたんに隠れていたパトカーに捕まりました。「安全を確認するため前進した」と抗弁したのですが、藪蛇だったようです。「ブレーキランプが点いたのは見えたが、完全に停止しなかったから違反だ」と言われました。それから、一時停車するごとに「静止は 2 秒です」と、むかしの体操中継みたいにつぶやくようになりました。一時停車しても、ずるずると前進しないと安全確認はできません。外形的な規則重視に疑問を抱くのは私だけでしょうか？

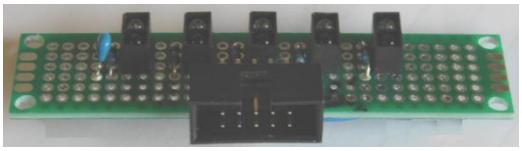
上側小型ユニバーサル基板



小型ユニバーサル基板

この基板には操作用のキースイッチと LCD 表示器を搭載しました。左側に見えるのはバスリピーターです。最初は LCD（ピン 間隔 1.5mm）を直付けするつもり（諦めた）だったので、目の細かい 1.27mm ピッチ基板を使っています。ランドの間隔が狭いので、短絡しないように注意しました。

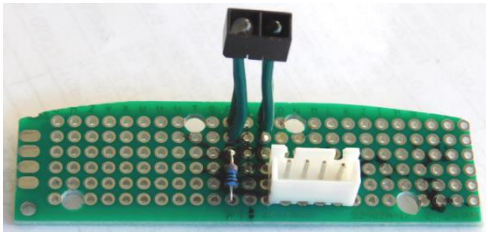
走路センサ基板



走路センサ基板

走路センサ基板全体を走路に近づけたいので、ピンヘッダーソケットは横向きを使いました。反射式光センサは 4 ピッチ間隔（10.16mm）にしました。

崖センサ基板



崖センサ基板

崖センサでは、センサ部だけを走路面に近付けるため、足が長いまま組付けています。車体の前面にガード板を取り付けるので、基板の前側を丸く削りました。

これ以外に、購入部品である超音波センサモジュール、モータードライバモジュール、光インターラプタの取り付け方法も考えておく必要があります。

前照灯は車台前面のパネルに穴を開けて取り付け、コネクタ付きワイヤを取り付けました。

4.4.2 ケーブルの組み立て

基板間を結ぶケーブルは、片側あるいは両端をコネクタ接続できるようにして、将来の修理や改造に備え、分解・再組立てできるようにしました。

コネクタは購入ユニット（光インターラプタ、モータードライバ）で決まっているものを除き、使い勝手のいい XH コネクタ（小型）とピンヘッダー（やや大きい）を使い分けています。ピンヘッダーを使う個所では、コネクタ付きのリボンケーブルを購入しました。あとはカシメ工具で必要な長さのケーブルを作っていきます。ひたすら忍耐のいる作業でした。

光インターラプタの接続に必要な ZH コネクタは超小型で、加工が大変です。細かい作業が苦手な人は、加工済ケーブル（3p）を探してみてください。

4.4.3 車台への取り付け

メカ部品と電子回路を車台に取り付けていきます。

車台の裏側に取り付ける基板からのケーブルは、車台の穴を使って取り出します。ケーブルの遊びが大きいと動輪に巻き込まれるおそれがあるので、スペーサーや電池ボックスに留めて固定しました。

三階建て基板

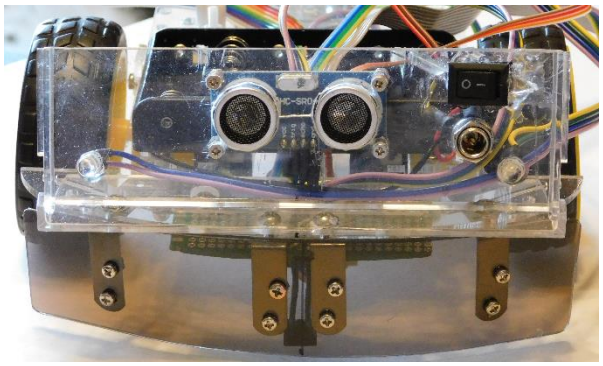
Raspberry Pi ZERO と 3 枚のユニバーサル基板は、重なるように取り付けます。キースイッチを操作するときに基板が変形しないように、スペーサー（上の写真にあるような樹脂ネジ）で支えます。

コラム　海外の交差点 4：Right turn on red

自己責任で『合理的』な運転を重視するアメリカやカナダでは、赤信号でも交差点に進入できるルールがあります。一時停車して安全を確認したら、右折できるというのです。交通量の多い道路では禁止されており、交差点に NO RIGHT TURN ON RED という標識があります。西部で始まったのが、オイルショックの時に全米に広がったのだそうです。右端の車線で止まっていると、後続車に叱られることもあります。左からくる車の速度を判断しそこなうと、怖い思いをするので、あまり好きではありません。アメリカでも見直しの議論があるそうです。

前面パネル

前面パネルに穴を開け、障害物センサ、前照灯、電源コネクタとスイッチを取り付けます。固定には 2mm のネジと接着剤を使いました。

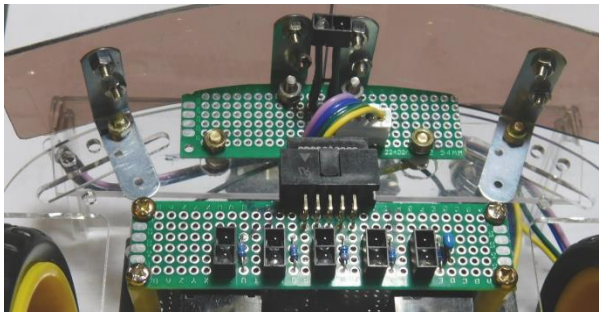


前面パネルとガード板

写真中央に見える二つ目は前方障害物センサ（超音波送受信器）、右側にあるのは電源スイッチと補助電源用コネクタです。パネルの下側、両端近くには前照灯を接着してあります。パネルの下に見える茶色いボードは、路面にある小さな障害物を押し出すためのガード板です。障害物センサの後方には電池ボックスが見えています。

走路･崖センサ

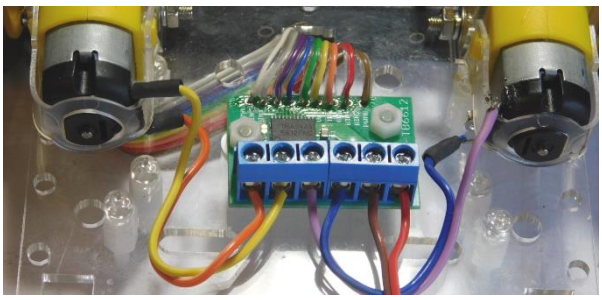
走路センサと崖センサは、走路に向けるため、車台の底側に取り付けます。取り付け位置は、走路センサはできるだけ前輪の車軸に近い位置に、崖センサはいちばん前側にします。走路センサは、スペーサーを使って路面に近い位置に固定します。崖センサーは、光センサの足を長く取ってあるので、ナット一個分だけ車台から浮かすだけにしました。



走路センサ（下）と崖センサ（上）の取り付け（裏側から撮影）

モータードライバ

モータードライバユニットは、モーターのすぐ近くに置きたいので、車台の裏面に取り付けます。プリント板の、回路のない個所に穴を開け、プラスチックネジで固定しました。

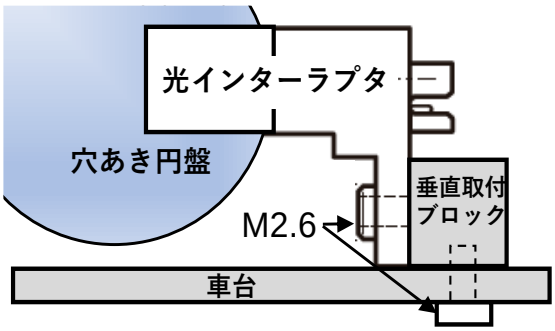


モータードライバの取り付け（裏側から撮影）

左右モーター側の端子が内向き（写真を参照）になるよう取り付けました。ドライバの出力#1 を写真の下側端子にしているため、左右の制御信号を逆（前進時には右モーターを右回転、左モーターを左回転）にする必要があります。これはデバイスハンドラのインクルードファイルで処理しました。

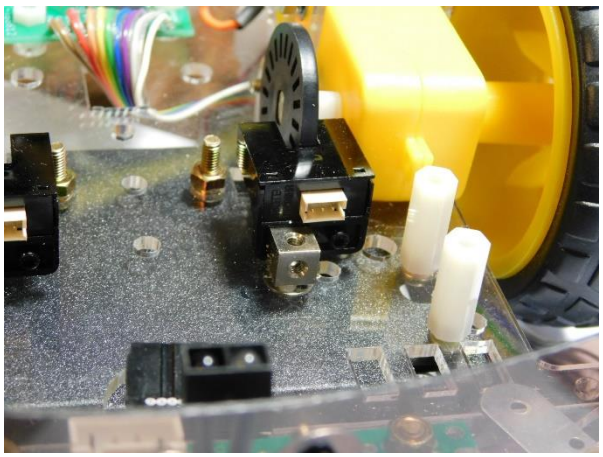
走行計

走行計のセンサ部の固定方法はだいぶ悩みました。光インターラプタ KI1233 は、2 本の腕の中に LED と受光器を入れ、対象物を挟み込むようにして動きを検出します。次の図のように立てて使うと、ちょうど光軸が円盤の中心とほぼ同じ高さになります。秋月電子通商で「垂直取り付けブロック」という部品（6mm□の金属立方体の 2 面に 2.6mm のネジ穴を切ってある）を見つけ、各面のネジ穴を使って光インターラプタと車台に固定しました。



光インターラプタの取り付け方法

実際に取り付けた様子を次の写真に示します。



穴あき円盤と光インターラプタの取り付け（裏側から撮影）

光インターラプタ（ZH コネクタが見える黒い部品）を垂直取り付けブロック（面に穴が開いた四角い金属）に組み付け、それを車台に固定します。動輪との間に見える白いポストは、走路センサを固定するためのスペーサーです。

走行計に ZH コネクタケーブルを取り付ける

電池ボックス

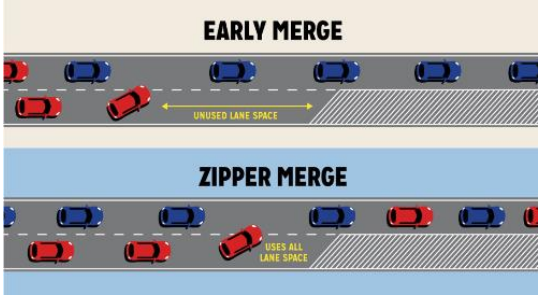
電池ボックスは（電池が入っているときは）いちばん重い部品です。動輪の近くに置いて、走路のグリップを良くすることと、後輪が浮き上がってしまわないことに気をつけて位置決めしました。ネジ穴を開ける前に部品を積み上げ、バランスを確認しておきます。

コラム　海外の交差点 5：Zipper merge

道路が立体交差している場所などには、合流車線があります。すぐに車線変更をしても構いませんが、多くの国では「目いっぱい合流車線を使って加速する」ことを推奨しています。道路の収容力を最大限利用するためです。その間に隣を走る車との『前後関係』を作るように、アクセルだけで速度を調整し、スムーズに合流できるようにします。

アメリカの合流車線は長く、終点近くで車線を区切る点線がなくなってしまう場所もあります（下の写真）。ウインカーも使わないので、合流したという実感がわきません。それでも大型トラックなどから「合流車線が短すぎる」という苦情が絶えないそうです。

事故や道路工事などで一部の車線が閉鎖されているところでは、右のような標識（Zipper Merge）を見かけます。「合流点でジッパーを閉じるように」一台ずつ合流せよという意味です。あまり手前で車線変更すると、スムーズに走れないし、走路スペースが無駄になるというわけです。州政府や自動車学校が繰り返し強調しているということは……そうしない運転者が多いということでしょう。万国共通の悩みかも知れませんね。



5　ソフトウエアの設計（1）システム構成と仕様

5.1　ソフトウエアの開発手法

自律走行車のソフトウエアを設計するため、全体の機能を分析して、複数の機能単位に分解していくことにします。それぞれの機能単位は、全体機能の一部を実現するもので、機能単位間で独立性が高い（他の機能と重複がなく、その内部だけで機能の説明ができる）ように設計します。必要なら、さらに小さい単位に分解し、全体を部品（機能単位）で組み立てられるようにします。この段階で、各部品の仕様も詳細化していきます。部品はできるだけ汎用的なものにし、他のプロジェクトでも利用するように考慮します。

5.1.1 ソフトウエアのモジュール化

こうやって詳細化された部品は、ソフトウエアのモジュールとして開発します。多くのモジュールは Python のオブジェクトとして定義します。オブジェクトには以下の 3 つの要素があります。

要素	説明
クラス	オブジェクトを生成するための定義
属性（プロパティまたはアトリビュート）	オブジェクトで使用する変数
操作（メソッドあるいはサービス）	オブジェクトに対して操作を行うための関数

オブジェクトモデル

各々のモジュールのオブジェクトモデルは、次章以降の各モジュールの最初に示しています。この本がなくてもプログラムを理解できるよう、同じか、より詳しい内容を各ファイルの先頭にコメントとして記述しています。

属性へのアクセスは、そのオブジェクトに特有の処理が必要なので、必ず操作関数を介して行います。ただ、属性の読み出しだけは、操作を介すると冗長なので、直接参照することもあります。一部の（オブジェクト内部でしか引用しない）属性と操作は、外部から参照できないようにします。Python のプログラムでは、このような属性と操作の名前を二個続きのアンダースコアで始まるように定義する（例えば`__state`）と、外部から参照できないようになります（`obj.__state` を参照するとエラーになる）。実は、他の言語と違って、その気になれば外部からこの属性にアクセスする方法はあるのですが、自分だけのプロジェクトであれば、この裏技を使わずにいることができます。

アンダースコアを多用すると、視認性が悪くなるので、この本の中でプログラムコード以外の場所ではアンダースコアを省略して説明します。

ソフトウエアを作りっぱなしで、使い捨てにするのは、たいへんな無駄です。せっかく Raspberry Pi を買い、ソフトウエアを作ったなら、何度でも、また長いこと使えるようにしたいものです。そのためには、ソフトウエアを把握しやすいくらい小さい単位で作っておき、他のソフトウエアと組み合わせられるようにします。ソフトウエアモジュールは、それぞれ一個のファイルにしておき、使うときに自動で結合させるようにします。

5.1.2 Linux ツールの利用

このプロジェクトでは Linux のツール cpp（C 言語プリプロセッサー）を活用しています。詳しい説明は前のプロジェクト報告「Raspberry Pi 中級電子工作：温度コントローラ　～設計から製作・検証・応用・保守まで」を参照してください。このプロジェクトでは、以下の命令を使っています。

```
#include
#define
#ifdef (#ifndef) ～ (#else ～) #endif
/* ～ */
-D～ オプション
```

ソフトウエアモジュールの応用範囲を広げるため、マクロを使ってパラメータや機能を選択できるようにしています。

5.2　プログラムの検証に備えて

作成したプログラムには、間違い（バグ）が含まれているので、実行しながら間違いを正す（デバグする）必要があります。プログラムの開発仕様に従って、それを満たしているかどうかを検証していきます。検証計画と実行は、プログラムの作成と同じくらい時間がかかるものです。この節では、検証のために、あらかじめ考慮しておくことをまとめます。

5.2.1 シミュレーション用コード

自分で作ったハードウエアとソフトウエアを組み合わせて使うと、問題が起こったときに、どちらに原因があるか調べるのが難しくなります。そこで、ソフトウエアのできるだけ多くの部分を、PC 上で検証します。ハードウエアへのアクセス直前まで動作させ、ハードウエア入出力のみをキーボードやスク

リーンに置き換えるようにします。プログラムは以下のように記述します。

```
#ifdef BCM2835
  :  実機を使うコード
#else
  :  シミュレーションコード
#endif
```

こうしておけば、普段はシミュレーション環境（実機を使わない環境）でプログラムを実行できます。BCM2835 を#define すれば、実機で走らせることができます。ハードウエア依存部分は、できるだけ小さくなるようにします。今回のプロジェクトでは pigpio パッケージを使うので、インクルードファイル use-pigpio.h でこの切り替えができるようにしました。

5.2.2 デバグ用コード

シミュレーション環境だろうと実機環境だろうと、動作を確認したくなることがあります。その時は、以下のように記述して、その時の変数をコンソールスクリーンに表示させるようにします。

```
#ifdef DEBUG
print('〇〇パラメータ', x, y, …)
#endif
```

5.2.3検証用代替プログラム（スタブ）

ソフトウエアの検証をするとき、別モジュールの動作を模擬する代替モジュールをスタブといいます。スタブ（stub）には切り株とか、ちびた鉛筆というような意味があります。それがないと目的とするモジュールの動作が確認できないところで使います。モジュール自身をスタブとして用いる（他モジュールの検証用コードを埋め込む）こともあります。

このプロジェクトでは、モジュール毎に評価用スタブ（test_XXX.py）を用意して、広い検証範囲を確保するようにしています。そのため、モジュールを単体で動作させるためにファイル末尾によく見かける、`if __name__ ==’__main__’:` 以下の部分は用意していません。

5.3　全体の構成手法

これから、自律走行車システムの構成を検討していきます。

1. まず全体を構成する手法について説明します。
2. 次に独立性の高いソフトウエアモジュールに分解し、それぞれの間の相互作用と実行タイミングを決めてやります。
3. 最後に各モジュールの仕様と検証方法を検討します。

実際のコーディングと検証は次章で行います。

5.3.1 マルチプロセス化

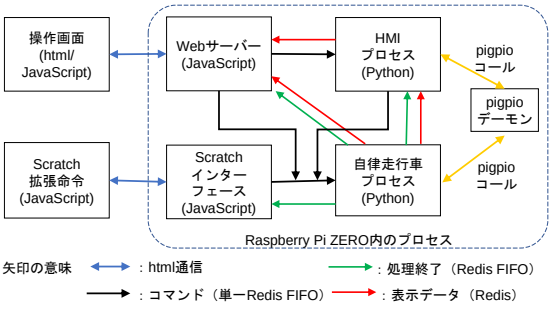
自律走行車のようなシステムを、単一プロセス（OS 上の実行単位）として構成しようとすると、プログラムが複雑になりすぎます。プログラミング言語を使い分けようとすると、コンパイルも面倒になります。

比較的独立性の高い機能は、独立したプロセスとして設計し、その間の相互作用を簡潔にすることで構成が容易になります。自律走行車というシステムの場合、以下のプロセスが考えられます（カッコ内は使用するプログラミング言語）。

- 自律走行車プロセス（Python）
- HMI（手動操作と表示）プロセス（Python）
- Web サーバー（JavaScript）
- Scratch インターフェース（JavaScript）
- pigpio デーモン（C）

最後の pigpio デーモンは、前にインストールした GPIO ライブラリを実行する既成プロセスです。

全体のプロセス構成とプロセス間のやり取り（データの交換）を次の図で示します。



マルチプロセス構成とプロセス間のやり取り

それぞれのプロセスは、一本のプログラムとして起動されますが、待ち行列や割り込み処理を含むので、実行順序は複雑です。プロセス内部のやり取りには変数が使えます。ローカル変数とグローバル変数を区別すること、割り込み処理で書き換えられることで起こりうる障害を避けること、オブジェクト内の属性を勝手に書き換えないことなど、通常の約束事に従います。

pigpio デーモンとの通信にはソケットあるいはパイプを使いますが、ライブラリの中に隠れているので、ここでは説明しません。

5.3.2プロセス間通信

プロセスの間では、通信機能を使って情報を交換します。前回までのプロジェクトで多用した、Redis データベースの FIFO 機能をここでも採用します。次の表にあるキーを指定してデータ（テキスト）を送ってやれば、受信側で取り出した順番に処理することができます。

このうち REDIS_TO_LCD_xxx は、ブラウザ上に表示された液晶（のイメージをコピーした）画面の更新データを、REDIS_TO_IFRAME は Web 画面内にはめ込む別の html ファイル名を渡すために使っています。

キーの名前	送信プロセス	受信プロセス
REDIS_TO_COMMAND	HMI Web サーバー Scratch	自律走行車 （単一 FIFO）
REDIS_TO_HMI	Web サーバー HMI（スイッチハンドラ）	HMI（単一 FIFO）
REDIS_TO_LCD_TOP	HMI	Web サーバー
REDIS_TO_LCD_BOTTOM	HMI	Web サーバー
REDIS_TO IFRAME	自律走行車	Web サーバー

Redis キーの名前（基本機能のみ）

今までのプロジェクトでは、Redis の FIFO（リスト）機能だけを使ってきましたが、今回はテキスト型変数の共有手段としても使用します。最新の値があればよくて、更新を知らせる必要がないところで使います。キーには REDIS_FOR_XXX という名前を付けることにします。

JavaScript は複数の FIFO からの受信を並列に記述できます。しかし Python プログラムでは単一の FIFO からの受信に限り、コマンドの内容によって処理を分けることにします。Python でも複数の FIFO を待つ機能はあるのですが、処理ルーチンの中でどこから受信したか判定する必要があるため、けっきょく同じようなプログラムになってしまうからです。

Redis へのアクセスにかかる時間は前に測定してあります。Raspberry Pi ZERO の場合、一度のアクセスに数 ms かかることが分かりました。安全を見込んで、100ms 周期の高速処理中でのアクセスは多くても 3~4 回に抑えるように設計します。

自律走行車プロセスが Redis からデータ（コマンド）を受け取るのは REDIS_TO_COMMAND だけですが、送る方のプロセスはいくつもあります。コマンドの処理結果を返すのはバックグランド処理なので、上の制限回数に数えません。100ms の割り込み処理中に発生する送信要求は以下のとおりです。

キーの名前	受信プロセス	用途
REDIS_TO_ALARM	サーバー	警報発生・解除を知らせる
REDIS_TO_SPEED	サーバー	走行速度を表示する
REDIS_TO_WHEEL	サーバー	操舵量を表示する

100ms 割り込み処理で使用する Redis キー

このうち REDIS_TO_ALARM は重要な情報なので、発生したら直ちに送信します。ライントレーシング走行中は操舵量が毎回変わります。走行速度はバックグランド処理か緊急停車時で変化するだけなので上の制限を満たせます。

HMI の表示用に電池電圧などを送る以下のキーは、自律走行車が走行していない（100ms 割り込みが起こらない）ときなので、この制限にかかりません。

キーの名前	受信プロセス	用途
REDIS_FOR_BATTERY	HMI	電池電圧を表示する
REDIS_FOR_OBSTACLE	HMI	障害物の存在を表示する
REDIS_FOR_SENSORS	HMI	走路と崖の状態を表示する

走行中でないときの表示に使用する Redis キー

5.4　自律走行車プロセスの構造化設計

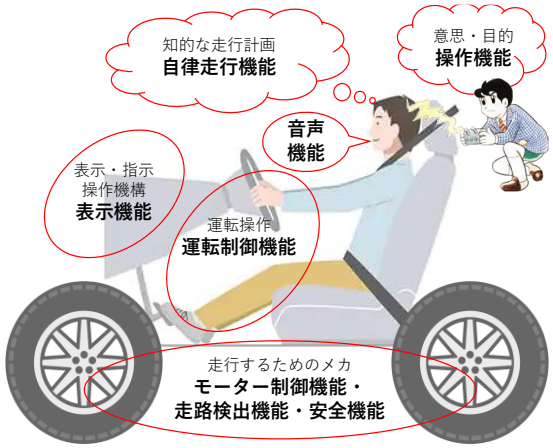
自律走行車プロセスをソフトウエアモジュールに分解（細分化）していきます。

最初に、自律走行システムに必要な機能を拾い出していきましょう。開発仕様として検討した内容をもとにまとめると、次の表のようになります。これがシステム構成の基本になります。

自律走行車プロセス機能	機能の説明
自律走行車ルート	自律走行車プロセスの実行を制御する 一定周期での実行制御 コマンド FIFO からの受信処理
自律走行機能 （自律走行プログラム）	経路・目標を決め走行する Web ドライブ ブロックドライブ Scratch ドライブ ライントレーシング
運転制御機能	自律走行機能の指示に従って運転する 目標地点に到達するまで走行する
モーター制御機能	運転制御機能の指示でモーターを駆動する
表示機能	方向指示、ブレーキランプなどを点灯する
走路検出機能	路面のガイドラインから偏差を検出する
安全機能 （安全センサ）	走行時の安全を確認する 崖への落下防止 障害物との衝突防止 電池残量の監視
HMI プロセスの機能	（参考：自律走行車とは別のプロセス）
操作機能 （ヒューマン・マシン・インターフェース）	キースイッチによる操作 Web ブラウザ上での操作 Scratch プログラムによる運転指示
操作読み取り機能	キースイッチの状態読み取り
表示機能	LCD と Web ブラウザへの表示 操作機能を補助する表示 走行状態の表示
音声機能	音声による表示機能

自律走行車に必要な機能

以前にシステム構成のヒントになると言った、運転のイメージ図と重ねてみると、こんな分類になると思います。



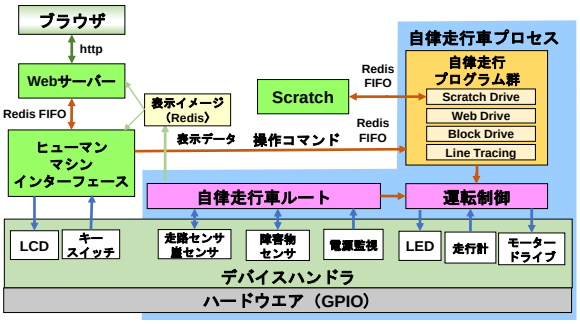
運転のイメージとシステム構成

各機能は次のような階層構造があり、上位機能は下位機能の実現方法を知らなくても実現できるようにします。表示機能と音声機能は操作機能を支援するためのものです。

操作機能を介してルートに走行条件を与える
→選んだ自律走行機能を実行する
→自律走行に基づいて運転操作をする
→指定条件に従ってモーターを制御する
←（必要なら）走路からの偏差を検出する
←安全な状態であることを確認する

5.4.1 階層化設計

各機能をブロック図にしたものを次に示します。



システムの階層化

緑色の Web サーバーと Scratch（インターフェース）、ヒューマン・マシン・インターフェース（HMI）は、それぞれが独立したプロセスとして、自律走行車プロセスに指示を出し、結果を受け取ります。この通信には Redis を使用します。これらの設計は、自律走行車プロセスの設計のあとで行います。

青色で囲った自律走行車プロセスは、単一のプログラムとして自律走行を実現します。自律走行プログラムは、操作機能からの指示に従い、下位の機能を使って自律走行を実現します。

ピンクのブロックは比較的早い応答を求められる機能で、上位機能とデバイスハンドラを繋ぐ役割を果たします。タイマー割り込みによって動作します。

白いデバイスハンドラは、ハードウエアを直接駆動するモジュールです。GPIO ポートやチップのレジスタを直接操作し、上位のブロックには（電圧とか距離といった）抽象的な情報を提供します。

5.4.2 処理間隔/周期の分析

実装方法を検討するため、自律走行車プロセスの機能に求められる処理の時間間隔（あるいは周期）を次の表にまとめました。キースイッチの検出と LCD への表示は自律走行車プロセスには含まれませんが、処理時間のグループ化を検討するため、いっしょに載せています

処理間隔または周期	機能
0.1～0.2ms	モーター駆動 PWM（ハードウエア処理）
0.6～6ms	超音波送受信時間測定（ハードウエア処理）
16～33ms	走行計のパルス積算（ハードウエア処理）
100ms 程度	超音波センサによる障害物検知 崖センサによる安全確認 走行計の計数値による走行終了の検出
100ms～500ms	自律走行プログラムのアルゴリズム実行 《キー操作の検出（チャタリング防止）》
500ms 程度	方向指示器（LED）の点滅 表示内容の更新
100ms～1 秒	走路センサによるガイドライン検出 ライントレーシングのための操舵量計算
数秒から数分	電池電圧の監視
制限なし（バックグランド）	自律走行へのコマンド解釈 《LCD 表示・Web 更新と音声出力》

各機能の処理間隔または周期

これを整理すると、次の 4 つのグループに分類できます。

- モーター用 PWM（ハードウエア処理）
- 動輪回転数と超音波の測定（ハードウエア処理）
- 100ms 割り込み、またはその整数倍
- バックグランド処理

割り込み周期は自律走行車ルートが決めることにします。走行中は 100ms で走行の制御を、停車中は 500ms で表示の更新を行うことにします。より遅い周期のモジュールは、割り込み処理内で回数を数えることでタイミングを作るか、バックグランド処理にします。

初期検討段階では、割り込みの基本周期は pigpio のソフトウエア PWM で発生させるつもりでした。しかし、Linux のシステム割り込み *signal.setitimer()*（ソフトウエア割り込み）でも 0.1ms 以内の精度で発生させられることが分かり、こちらを使うことにしました。確認のためのプログラムを次に示します（一回目だけは python がバイトコンパイルをするため、測定値が長くなります）。

test_periodic.py

```
/* test program of linux periodic interrupts
2022/2/21
*/

#include "include/use-time.h"
#include "include/use-sys.h"

#define REPEAT  20   /* 試験の繰り返し回数 */
#define INTERVAL  0.1   /* 割り込み周期 */

class system_test:
  def __init__(self):
    self.dt1 = datetime.now()
    self.count = 0
    signal.signal(signal.SIGALRM, self.handler)
    signal.setitimer(signal.ITIMER_REAL, INTERVAL,
INTERVAL)

  def handler(self, signum, frame):
    dt2 = datetime.now()
    tim = (dt2 - self.dt1).total_seconds()
    print('#', self.count, ' interval = ', tim *
1000, ' ms')
    self.dt1 = dt2

    self.count += 1
    if (self.count >= REPEAT):
      sys.exit(0)

periodic = system_test()

sleep(REPEAT * INTERVAL)
```

コラム　割り込み処理

割り込みが起こると、コンピュータはそのとき実行していた機械語の命令（あるいは Python の実行単位である『バイトコード』）の終了後に、割り込み処理ルーチンの実行を始めます。割り込み処理が終わったら、次の命令から実行を再開します。このとき、こんな問題が起こり得ます：

1. メイン（バックグランド）処理と割り込み処理が同じ変数を使っているとき、変数を読み込んでいる間に割り込まれることで、変数の一部だけが変更されると、データのつじつまが合わなくなることがある
2. バックグランド処理がリエントラント（処理中にもう一度呼び出せること）でないサブルーチンを呼び出している間に、割り込みが起こり、そこでも同じサブルーチンを呼び出すと、前者が正しく処理されないことがある。共用の通信ソケットを使う場合も同様

こういった問題を避けるため、特に OS の上で走っているプログラムでは、割り込み処理はできるだけ単純で短いものにし、面倒な操作はしないようにと『教授』されます。例えば、「割り込み処理内で print（リエントラントでない）は行うな」という『指導』です。

今回の自律走行車プロジェクトでは、『共有』する必要のあるデータや IO 資源などをバックグランド処理が扱う間は、割込みを受け付けないようにしました。それでも（処置忘れがあるのかもしれませんが）うまくいかないときがあり、プログラムの構造を変える必要がありました。

走行中に実行する機能は、100ms に 1 回ずつ処理するものとして、当面の検討を進めます。この 100ms という時間は、超音波センサ HC-SR04 の送信間隔を 60ms 以上とする必要条件から決めたものです。この処理間隔なら、センサで危険を検知してから停車するまでの移動距離は 3cm 以内に抑えられます。検証段階で割り込み処理にかかる時間が長すぎることが分かったら、この周期を長くすることを考えます（上の移動距離は大きくなってしまいますが）。この間隔で実行するのは次のような機能です。

- 安全センサを読み込み、危険が検知されたら緊急停車する
- 走行計を読み出し、目標に到達したら停車する
- 走路センサを読み出し、操舵量を自動的に決める（ライントレース実行時のみ）
- 方向指示器の点滅や LCD の表示更新は 0.5 秒程度で更新すればいいので、順番に一つずつ処理する

この程度の処理ならあまり時間がかからないようにも見えますが、Redis や GPIO へのアクセスを含むので、油断は禁物です。OS を呼び出したとき、他のプロセスが実行されたりして、見かけ上処理時間が長くなることがあります。割り込み処理に時間がかかり過ぎると何が起こるのでしょう？

Linux のシステムタイマーは、割り込み処理にシグナルを送って処理を始めさせます。割り込み処理が終了すると、次のシグナルを受けるまで処理は『休眠』します。あるタイマー割り込みの処理中に次のタイマー割り込みが発生しても、前の処理が終了するまでシグナルは受け付けられません。だから割り込み処理時間を測定して、100ms より充分短いことを確認しておく必要があるのです。

5.4.3モジュールへの分解と相互作用

ここまでの検討をもとに、自律走行車プロセスを（基本的にオブジェクトごとの）ソフトウエアモジュール（ファイル）に分解します。同時に、それらの間の相互作用（オブジェクトの生成と引用）を定義します。オブジェクトの一部は、生成したモジュールから下位のモジュールに引き渡されることがあります。

分解した結果を次の図に示します。機能を階層化したときと同じ色をつけてあります。下位モジュールは上位モジュールで生成・引用されますが、運転制御機能 driver だけは、各自律走行プログラムにオブジェクトを渡すことで直接引用できるようにしました。

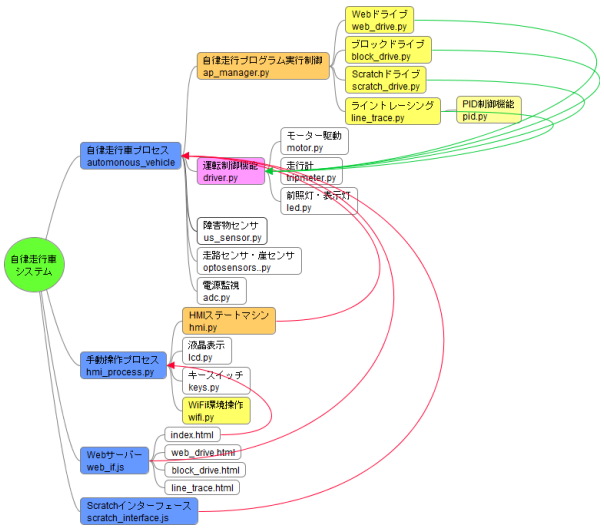
モジュールへの分解

設計段階になったら、モジュール名はファイル名としても使用します。オブジェクトのクラス名は、機能を示すために、もっと長くすることがあります。

Web サーバーや Scratch インタープリタ、ヒューマン・マシン・インターフェースとの通信には、Redis を使用します。コマンドは ap_manager が一括して受け付け、必要とするモジュールに割り振ります。表示する処理結果は各モジュールが返すことにして、各オブジェクトの生成時に Redis オブジェクトを渡すことにしました。

pid は自動制御で使用される PID 制御アルゴリズムを実行するモジュールで、温度コントローラ・プロジェクトで実績のあるものです。

モジュールの親子関係（灰色の線）、子でないオブジェクトの引用（緑の矢印）と Redis FIFO を介したコマンドの流れ（赤の矢印）を次の図に示します。



モジュールの親子関係とデータの流れ（全プロセス）

その他のモジュール

上の図には示していませんが、これ以外に次の下位モジュールを用意しました。

hmi_format は、センサから得られたデータなどを、LCD に表示するテキストに変換する関数です。ftos は浮動小数点数を、桁数を指定してテキストにする関数で、これも前のプロジェクト（温度コントローラ）で作成したものをそのまま使っています。

signal_conditioner は、走路センサの出力から、目的経路との偏差を計算して PID 制御に使うために用意しました。ライントレーシングのコーディングで詳しく説明します。

検証ストラテジー

ここでモジュールの検証に向けた準備をしておきます。これだけの数のモジュールを一度に検証することには無理があります。不具合があったとき、どのモジュールに起因するか分かりづらいからです。

私が採用したアプローチでは、まず最下位のモジュールの機能を検証し、次にそれを使ってすぐ上位のモジュールを検証します。こうやって順番に上位モジュールを検証していく、ボトムアップ型のアプローチをとりました。

下位モジュールの機能が検証できていれば、それを呼び出すモジュールの検証では、最下位の機能まで動作させる必要はありません。検証対象のモジュールが、下位モジュールをどういう順番で呼び出しているかを知れば、十分です。そのため、各モジュールを上位モジュールの検証用スタブとして使えるようにしました。こうしておけば、最下位モジュール（デバイスハンドラ）でハードウエア（GPIO）を使うとき以外の検証は、実機（Raspberry Pi ZERO）でなくても行えます。また、運転制御部にシミュレーション機能を組み込み、仮想的に走行してみることもできるようにしました。

ところで検証用スタブ機能を上手に使えば、今回のアプローチとは逆に、最上位モジュールから下位に向かって検証を進めること（トップダウン型アプローチ）も可能です。それでもボトムアップ型アプローチをとったのは、検証の途中でも実車両を動かしてみたいという（不純な）動機からです。

検証用スタブ機能を実現するため、次のようなマクロ定義を用意しました。次の表の先に出てくるものほど優先度（ファイルのインクルードや実コードで実現している）が高くなっています。

マクロ定義	検証時の動作
#ifndef BCM2835	GPIO を使わない（PC 上でも検証できる）
#ifdef APM_STUB	ap_manager の呼び出しを表示する
#ifdef AP_STUB	自律走行プログラムの呼び出しを表示する
#ifdef DRIVER_STUB	運転制御機能の呼び出しを表示する。運転制御機能が使っていないデバイスのハンドラも同様
#ifdef SIMULATED_DRIVE	運転制御機能を使って仮想走行を行う
#ifdef HANDER_STUB	各デバイスハンドラの呼び出しを表示する（入力デバイスはコンソールから入力値を与える）
#ifdef GPIO_STUB	GPIO 呼び出しを表示する（入力値には適当な値を使う）
#else （上のマクロがすべて未定義）	GPIO 動作をコンソール入出力で代替する
#else (#ifdef BCM2835)	Raspberry Pi ZERO の GPIO を使って動作する

検証用スタブを使うためのマクロ定義

5.4.4 単位系の選択

自律走行車プロセスの内部変数や演算に使う単位系を決めておきます。

数値処理を単純にするため、長さの単位は mm、時間の単位は秒、角度の単位はラジアン（弧度法）を使うことにします。ただし、HMI プロセスとのインターフェースや検証プログラムでは長さに cm、角度に『度』（度数法）を使うことにします。「右に 45 度回頭してから 25cm 前進する」とした方が、「右にπ/4（0.785398…）回頭してから 250mm 前進する」より分かりやすいからです。変換は自律走行プログラムの中と、検証プログラムの結果表示部で行います。

また、この本のなかでは 0.1 秒を 100ms などとして説明しています。

5.5　自律走行車プロセスの実行制御

各モジュールを実行するタイミングを整理します。

バックグランド処理

自律走行車プロセスが起動されると、各オブジェクトの生成と初期化を行い、コマンド待ち状態になります。Redis FIFO を介してコマンドが受信されると、それを実行します。

ap_manager は、操作機能（ヒューマン・マシン・インターフェース、Web サーバー、Scratch インタープリタ）からの指令（コマンド）を受け付け、順番に処理していきます。実際には下位のモジュールに処理を委ねます。処理が終わったら、次の指令待ち状態のなることで、バックグランドとして実行されます。

割り込み処理

システムからの定周期割り込みを使って、一定時間ごとに必要になる処理を行います。割り込みは自律走行車プロセスルートが受け付け、下位モジュールを呼び出します。

先に割り込み周期として 100ms を選びましたが、これは車両の制御性や安全性を確保するためでした。これは車両が走行しているときの話で、自律走行プログラムが動いていないときはそこまで早くする必要はありません。むしろ、センサの感度調整などなどの機能を充実させたいので、その時は周期を長くすることにしました。

割り込み処理の内容は、センサの駆動と入力読み込み、LED の点滅、PID 制御演算などです。センサの LCD 表示の更新は、『遅い』周期で行います

pigpio に起因する制限

当初の計画では、割込み処理は全て、OS タイマーからのシグナルを処理するタイミングで実行するつもりでした。バックグランド処理は Redis FIFO からデータを受け取りながら（データがなければ実行が止まる）行うことで、暇な間は CPU を使わずに済みます。

実際に動作させてみると、問題が起こりました。pigpio はデーモンとの通信にソケットを使いますが、その途中で割り込みが起こり、そこでも pigpio にアクセスしようとすると、ソケットを上書きしてしまいます。調べてみると、マルチスレッドへの対策は取っていましたが、割込みはお手上げのようです。バックグランド処理内で割り込まれたら困るところは、pysigset というパッケージで一時的に割り込みを止められます。この機能を簡単なプログラムでは確認できたのですが、うまくいきませんでした。保護し忘れている個所があるのかもしれませんが、次に説明する対策を取りました（保護の記述は残っている）。

pigpio の開発者自身も「割込みサービス内ではフラグを立てるだけにして、必要な I/O 操作はメインルーチンで行うのが良い」と説明した記事を見つけました。自律走行車プロセスをこのやり方に変更すると、確かに異常は起こりません。フラグを調べるためには、バックグランド処理をときどき行うことと、定周期処理の間隔がばらつくという代償が必要です。Redis FIFO のブロッキング読み出し（データがないときは入ってくるまで、プロセスの実行を止めて待つ）では、再開時に割込み処理から CPU を奪ってしまうようです。これを回避する方法は見つかっていません。

5.6　自律走行車プロセスの仕様設計

自律走行車プロセスを、ソフトウエアモジュールと各々の実行制御に分解できました。次は、各モジュールの設計仕様を、トップダウンで決めていきます。

ここで大切なのは、あるモジュールの設計をするとき、下位のモジュールの詳細にこだわらないことです。下位モジュールには、「これこれの機能を果たして欲しい」という定義だけを行い、それを使ってどんな機能を実現するかを設計します。

そのために有効なのが「オブジェクト」モデルです。オブジェクトとは、(『属性』などと呼ばれる）内部変数と、(『操作』などと呼ばれる）処理からなります。外部からは見えるのは属性と操作だけで、それ以上の詳細は内部に隠されています。例えば、あるデバイスのハンドラは、そのデバイスのアドレスや内部レジスタを知らなくても、上位モジュールがデバイスを使えるように設計します。設計したオブジェクトの定義名を『クラス』といいます。以下の仕様設計は、オブジェクトのクラス・属性・操作の定義として行っていくことにします。

属性はいつでも読み出し可能ですが、書き込みは必ず操作を介して行うようにしています。こうすることで、オブジェクトに固有の設定可能範囲などをチ

ェックしたり、変更した時の動作を指定したりできるからです。この章で説明するのは、そのオブジェクトの機能を定義するのに必要な属性だけで、単なる静的変数はモジュール設計時に追加します。ただし、機能を説明するうえで重要な属性は記載ました。

5.6.1 自律走行車ルート

自律走行車プロセスの最上位（ルート）autonomous_vehicle は、二つの直下オブジェクト ap_manager と driver（と警報関係のデバイスハンドラ）を生成（必要なデータ領域を確保して初期化すること）したら、低周期処理 SlowISR をタイマー割り込みハンドラとして登録して、コマンドを待つようにします。コマンドは Redis FIFO（キー：REDIS_TO__COMMAND）を介して受け取ってコマンド受信処理に渡し、次のコマンドが渡されるまで待ちます。また SlowISR が呼ばれたことを検出したら、操作 SlowPeriodicOps が以下のセンサを読み取って HMI 機能に送り、表示してもらいます。Redis を何回も使いますが、割り込み周期が長いので問題ありません。

- 電池電圧の測定結果
- 崖・走路センサの検出結果
- 障害物センサの検出結果

ルートのもう一つの機能は、自律走行プログラムが実行されているときに、一定周期（100ms）で運転制御などを実行することです。高速割り込み処理 FastISR が呼ばれたことを検出すると、操作 periodicOps が各種センサの読み取り、緊急停止処理、表示灯のフラッシング、運転機制御能や自律走行プログラムの実行タイミング発生をまとめて行います。

運転制御機能 driver の警報処理はルートから直接操作しますが、走行の制御は自律走行プログラムが行うので、ap_manager を通して運転機能オブジェクトを渡してやります。このおかげで、自律走行プログラムを検証するときに、ルートがなくても定周期処理を実行できるようになりました。

前にも説明したように、高速処理中の Redis へのアクセスは 3 回以内に制限します。速度や操舵量は、変更があった時にだけ更新（平均すれば 100ms に一回以下）すればいいので、Redis アクセスの回数制限を満たすことができます。

割り込み周期の変更は、コマンド受信処理 ap_manager が現在実行中の自律走行プログラムを知らせることで実現します。自律走行プログラムが実行中は高速、それ以外は低速処理を選ぶようにします。

ファイル名	automonous_vehicle.py
クラス	system_timer
属性	説明
	（外部から見える属性はない）
操作	説明
FaseISR	高速の割り込み処理
SlowISR	低速の割り込み処理
periodicOps	高速の周期的処理を実行する
SlowPeriodicOps	低速の周期的処理を実行する
command	受理したコマンドを ap_manager に渡す
ap_change	ap_manager が実行状態を知らせる

自律走行車プロセスルート

高速の周期的処理での GPIO 操作（一度に 1ms 強かかる）回数を数えておきます。緑欄はライントレーシング時、黄色欄はブロック/Scratch ドライブ時だけなので、同時に操作することはありません。非常停止後は走行しないので、そのときだけ時間がかかっても問題ありません。通常は最大 17 回です。また Redis アクセス（3ms 強）は 3 回以内に制限するので、時間のかかる処理は 30ms 強（プラス OS コールによる遅延）となり、何とか周期内に収まりそうです。

操作	コール回数	説明
崖センサの操作	3	LED の ON/OFF、センサ読取
障害物センサの操作	3	超音波送信、伝搬時間読取
電源電圧の読取	1	ADC の読み出し
走路センサの操作	7	LED の ON/OFF、センサ読取
走路追従の操舵	2 (+1)	モーター回転制御、方向指示
走行距離計の読取	2	左右の計数値読取り
非常停止	8	ドライバモードの変更

高速の周期的処理中の GPIO 操作回数

5.6.2コマンド受信処理

コマンド受信処理 ap_manager は、自律走行プログラム群を制御します。自律走行プログラムが起動されると、それが使う Web ページを Web サーバーに伝え、ブラウザに表示させます。また、自律走行車ルートに伝えることで、割り込み周期を変更してもらいます。

ファイル名	ap_manager.py
クラス	application_manager
属性	説明
current_app	現在実行中の自律走行プログラム（内部専用）
modules	自律走行プログラム群（内部使用）
操作	説明
command	HMI から与えられたコマンドを解釈・実行する
control	自律走行プログラムに 100ms 毎の処理を行わせる

コマンド受信処理オブジェクト

ヒューマン・マシン・インターフェースやブラウザから受け取るコマンドは、ルートを介して操作 command に与えられます。

コマンドは指令（多くは一文字）とパラメータをつなげたテキストで、パラメータが次のレベルへの指令とパラメータになっています。次のレベルには、最初の指令部を除いたコマンドを渡すことにします。

次の表に自律走行車プロセス（コマンド受信処理）へのコマンド構成を示します。指令は分かりやすい名前で記載していますが、実装時に一文字（または数文字）で置き換えます。各自律走行プログラムへのコマンドは、それぞれのプログラムの節で説明します。

用途で Sim とあるのは、仮想走行など検証時に使うために追加したもので、実機上で動作させるときは使いません。コマンド受信処理（あるいはそれを代替する検証スタブ）が解釈します。

プログラム ID は、Run 指令には必須ですが、他の指令の解釈時には（実行中のプログラムは分かっているから無視する）ので、付随させなくても構いません。

用途	指令	パラメータ	説明
Sim	Control	回数	高速周期の処理を実行する
	Callback	終了要因	走行終了のコールバックを行う
AP	Run	プログラム ID	自律走行プログラムを実行させる
	Pause	プログラム ID	実行中のプログラムを休止させる
	Resume	プログラム ID	休止中のプログラムを再開させる
	Terminate	プログラム ID	実行中のプログラムを終了させる
	Command	コマンド	実行中のプログラムにコマンドを与える
Sim AP	Exit		検証プログラムを終了する 自律走行車プロセスを終了する

自律走行車プロセスへのコマンド

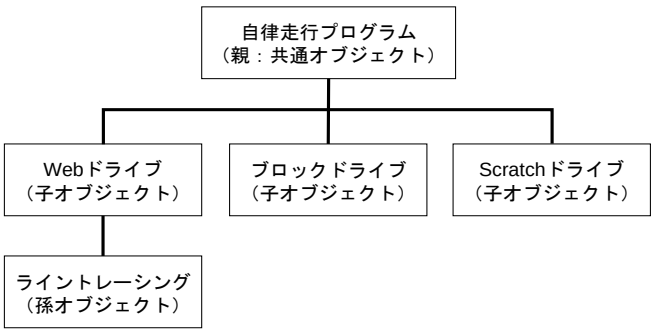
5.6.3自律走行プログラム

自律走行プログラムは 4 種類用意しましたが、オブジェクト構造のかなりの部分は共通です。これを『親オブジェクト』として、各自律走行プログラムはそれを継承（inherit）する『子（あるいは孫）オブジェクト』として設計しました。

ファイル名	drive_program.py
クラス	Drive_program
属性	説明
myID	自律走行プログラムの ID
state	自律走行プログラムの実行状態
drive_state	運転状態
drive_before	停車前の運転状態
avoiding	障害回避中フラグ
speed	走行速度
wheel	ハンドル操作量
alarm	警報の発生状況
操作	説明
start	自律走行を始める
pause	自律走行を中断する
resume	自律走行を再開する
terminate	自律走行を終了する
stopped	運転停止時に呼ばれるコールバック関数
update	速度、操舵量の表示を更新する
command	自律走行プログラムに命令を与える
control	自律走行プログラムの定周期実行を行う
my_alarm	警報処理を行う

自律走行プログラム（親オブジェクト）

command, control, my_alarm の 3 つの操作は、「何もしない」ように定義しておきます。各自律走行プログラムで、必要に応じて定義（上書き）します。自律走行プログラムの親子関係（オブジェクトの継承）を次の図に示します。



自律走行プログラムの親子関係

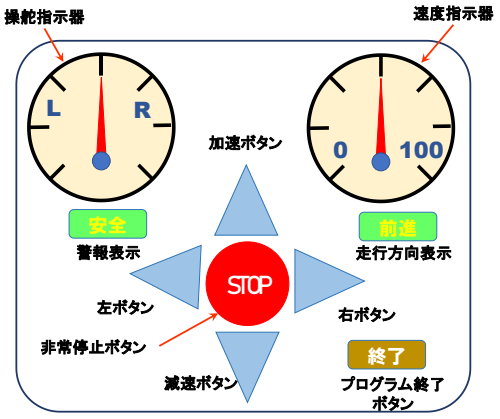
自律走行プログラムごとの動作指定は、command 操作のパラメータとして与えられます。Web ブラウ

ザから自律走行プログラムを使うことを想定しているので、ブラウザ上のイメージも同時に検討しておきます。

Web ドライブ

Web ドライブでは、ブラウザ上の矢印をクリックすることで前進・後退やハンドル操作を実行させます。Web ブラウザ上の画面イメージを次に示します。

他の自律走行プログラムと共通の機能もここで仕様設計しておきます。



Web ドライブの操作画面（イメージ）

安全警報が発生しているとき警報表示ボタンをクリックすると、アラームを解除して障害回避モードになり、運転を再開できます。停車は加減速ボタンの操作でもできますが、緊急時には STOP ボタンでも停車させることができます。プログラム終了ボタンをクリックすると、Web ドライブを終了し、他の自律走行プログラムを選べるようになります。この三つのボタンは、Scratch ドライブ以外の自律走行プログラムすべてで使えるようにします。イメージ図にはありませんが、その場で右あるいは左に回頭するボタンも設けます。回頭を停止するには、同じボタンをもう一度クリックするか、停止ボタンをクリックします。

自律走行プログラムが把握している走行速度と操舵量もブラウザに表示できるよう（適宜）更新します。

ファイル名	web_drive.py
クラス	web_driver
コマンド	説明
Go forward	静止時は前進、前進時は加速、後退時は減速
Go backward	静止時は後退、後退時は加速、前進時は減速
Steer right	右に一定量だけハンドルを切る
Steer left	左に一定量だけハンドルを切る
Right roll	右回りに回頭を始める
Left roll	左回りに回頭を始める
Stop	停車する（緊急停車）
Disarm	安全センサのアラームを解除して走行する
Exit	Web ドライブを終了する

Web ドライブオブジェクトへのコマンド

ここで、安全警報を例に、自律走行プログラムと Web ブラウザとの相互作用方法を決めておきます。原則としてブラウザは『状態』を持たず、自律走行プログラムから受け取るメッセージをもとに警報表示ボタンの表示を変えるようにします。メッセージは以下の 4 つです。

- 安全（初期化時）
- 安全警報（障害物と崖センサアラームの OR）
- 障害物回避中
- 電池交換要求

ブラウザはメッセージ毎の表示を定義した『辞書』を持ち、それに従って表示色やテキストを変更します。警報表示ボタンをクリックすると障害物回避（Disarm）コマンドを発行しますが、それに対する反応は自律走行プログラムが決めます。

ブロックドライブのブロックをクリックしたときや、ライントレーシングでガイドラインの色を変更したときなども、同様の相互作用を行うように設計します。

ブロックドライブ

ブロックドライブでは、個々の動作を実行したり、順番に指定した動作を手順として記録・保存・再生したりします。進行状況は Web に表示します。

ブロックドライブや Scratch ドライブでは、指定距離の走行など、実行に時間がかかる命令があります。そのため、Web サーバーや Scratch インターフェースに、実行の終了を知らせる必要があります。ここでは Redis FIFO を介してサーバーに終了を知らせる（報告する）ことにします。また、ある走行コマンドを処理中に次のコマンドを受信したら、次

を除いて新しい方のコマンドを廃棄することにします。

- 終了（Exit）命令と緊急停止（Stop）命令はいつでも受け付ける
- 手順記録開始命令の実行中は次のコマンド（一度に一つずつ）を受け付ける

Web 表示イメージを次に示します。ブロックを押すと、その処理中はブロックの色が変わり、処理が終了すると元の色に戻ります。このために Redis FIFO（キーは REDIS_TO_RESPONSE）を介して報告（一文字＋ブロック名）を Web サーバーに返すようにします。

ブロックドライブの操作画面（イメージ）

ブロックドライブでは、走行命令を手順として記録し、保存したり、再実行したりできるようにします。今回は、保存できる手順は 1 つだけに限定しました。

警報解除と終了の操作は Web ドライブと同じです。ブロックドライブが受け付けるコマンドを次ページの表にまとめます。このうち薄い黄色で着色したコマンドは実行に時間がかかる命令、薄い青で着色したコマンドは実行がすぐに終わる命令で、まとめて『ブロック実行コマンド』として設計に使います。

ファイル名	block_drive.py
クラス	block_driver
属性	説明
sequence	ブロックの実行手順
seq	順次実行するための手順コピー
seq_state	手順ステートマシンの状態
executing_block	現在実行中のブロック
from_first_step	手順を最初から実行するためのフラグ
driving_speed	指定走行速度（現在の走行速度ではない）
コマンド	説明
Clear Sequence	走行手順をクリアする
Start Recording	走行手順の記録を始める
Stop Recording	走行手順の記録を止める
Step Sequence	走行手順を 1 ステップだけ実行する
Replay Sequence	走行手順を最初から実行する
Save Sequence	走行手順を保存する
Load Sequence	走行手順を呼び出す
Slow Speed	走行速度を微速に設定する
Medium Speed	走行速度を半速に設定する
Fast Speed	走行速度を最大速に設定する
Forward 10	10cm 前進する
Forward 30	30cm 前進する
Backward 10	10cm 後退する
Backward 30	30cm 後退する
Right 45	右に 45 度回頭する
Right 90	右に 90 度回頭する
Left 45	左に 45 度回頭する
Left 90	左に 90 度回頭する
Wait 3	3 秒間静止する
Stop	停車する
Disarm	安全センサのアラームを解除して走行する
Exit	ブロックドライブを終了する

ブロックドライブオブジェクトへのコマンド

Scratch ドライブ

Scratch ドライブでは、Scratch プログラムが指定する（既定）動作を行い、動作開始時と終了後に報告を返します。操作には Scratch の画面を使うので、ここでは説明を省略します。他の自律走行プログラムで使うような警報ボタンと緊急停止ボタンは設けません（Scratch プログラムが処理する）が、プログラム終了ボタンだけは残すことにします。

Scratch プログラムからの要求が完了したときは、Redis FIFO（キーは REDIS_TO_SCRATCH）を介して報告（結果＋Scratch コマンド名）を Scratch インターフェースに返すようにします。

プログラムの実行制御はブロックドライブとよく似ています。違うのは、(Scratch プログラムがあるから）走行手順の記録が不要な点と、一命令でブロック類似の動作を連続して二つ以上行うことがある点、それに移動距離などの動作パラメータが指定できる点です。

ファイル名	scratch_drive.py
クラス	scratch_driver
属性	説明
cmd	Scratch からの命令（終了報告用）
busy	走行中（次のコマンドを受け付けない
next_move	組合せ走行後半の動作
next_parm	組合せ走行後半の移動距離/回転角度
X, y	現在の(x,y)座標（単位は mm）
θ	現在の方位（北を中心に±π）
操作	説明
travel	指定距離だけ直進する
roll	指定角度だけ回頭する
コマンド	説明
Initialize	設定を初期化する
Calibrate	現在の位置と向きを座標の基準にする
Set coordinate (x, y)	現在の位置を指定座標(x, y)とする
Set orientation(θ)	現在の向きを指定方位θとする
Go forward(d)	現在の向きに指定距離 d だけ前進する
Go backward(d)	現在の向きに指定距離 d だけ後退する
Turn(θ)	指定角度θ（右が正）だけ回頭する
Align(θ)	指定方位（北から±180°）を向く
Turn around	少し後退してから 180° 向きを変える
Go to (x, y)	指定座標(x, y)に移動する
Speed(s)	速度を全速の s%に設定する
Disarm	安全センサの警報を解除して走行
Exit	Scratch ドライブを終了する

Scratch ドライブオブジェクトへのコマンド

コマンドのパラメータ（x, y, θ）の単位は cm と度で、scratch_drive.py の中で内部単位系に変換します。コマンドの種類はもっと増やしていきたいと思っています。

安全警報が発生あるいは解除すると、Scratch インターフェースに報告を返しますが、その時実行中あるいは最後に実行されたコマンドを返すことにしました。Scratch インターフェースで処理することを期待しています。

ライントレーシング

ライントレーシングのコマンドと表示イメージは Web ドライブとよく似ている（ライントレーシングの開始とガイドラインの色を指定するボタンを追加する）ので、表示イメージの説明は省略します。Web ドライブの操作を継承するため、その子プロセス（共通部の孫）として設計します。

走路に沿って走行するには、5 個のセンサを使って走路からのずれを検出し、ずれを補正するようにハンドルを切ります。自動操縦には現代制御理論を使う例が多いのですが、今回は古典的な PID 制御を使ってみることにしました。入出力（走路からの偏差と操舵量）の数が少ないときは簡単かつ十分な性能を発揮してくれるからです。制御に使うパラメータを車両の特性に合わせる（チューニングといいます）のですが、毎回変える必要はないので、後で説明する方法を使って決めます。

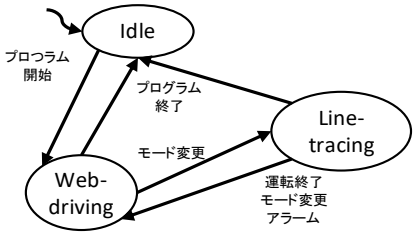
化学プラントなどの PID 制御では、安全な操業を行うため、最初は手動でゆっくりと動作を始めます（制御系はその状態をトラッキングしています)。そうやって最終的な運転状態に近づけてから、自動運転を開始します。前者を手動運転（Manual）モード、後者を自動運転（Automatic）モードと呼びます。

コラム　面舵いっぱい

船舶の操縦（操舵）には、面舵（おもかじ：右方向へ進む）と取舵（とりかじ：左方向）という用語が使われます。「面舵一杯」は船尾の舵板をいっぱいに傾けることですが、その角度は一般船舶で 30°、タンカーや軍艦では 35°以上だそうです。その時の回転半径は船舶の長さの 2 倍程度で、かなり急な旋回になります。映画などでは船が傾いたり、乗客が倒れたりしますね。

このプロジェクトでは、include/driver.h のなかで、「面舵いっぱい」はハンドルを最大限右に切ることと定義しました。船舶と比べると、かなりおとなしい操舵です。「取舵 10 度」は（最大が 30 度なので)、左に 33%などと決めました。

自律走行では、最初は Web-driving モード（PID 制御は Manual）になり、Web ドライブと同じインターフェースで走行し、スタート地点に移動させます（手で持ち上げて動かしても構いません）。それから Line-tracing モード（自動制御は Automatic）で走行を始めるようにします。プログラムが停止している Idle 状態を含めて、以下のようなステートマシンを実装します。



ラントレーシングプログラムの状態遷移図

ラントレーシングを始めたら、次の条件のいずれかが満たされて停車するまで、自動運転を続けます。

1. プログラム終了コマンドを受信した
2. 走路の終点に到達した
3. 走路（ガイドライン）を見失った
4. 安全警報か電池電圧低下警報を検出した

このうち、走路の終点はガイドラインを（走路センサ幅より広い）T 字型することで設定します。車両が走路と垂直に交差したときも停車することになりますが、その時も道に迷ったとみなします。

ファイル名	line_trace.py
クラス	line_tracer
line_sensor	ガイドラインセンサオブジェクト
sc	偏差計算オブジェクト
pv	PID 制御に与える現在の偏差
pid	PID 制御アルゴリズム
color	追従するガイドラインの色
first_action	Web ドライブに戻った直後を示すフラグ
操作	説明
guide_line	走路センサのデータを取り出す（HMI 用）
コマンド	説明
Line color	走路の色を変更する（デフォルトは黒）
Start Trace	Automatic モードでライントレース走行を開始する
Stop Trace	ライントレース走行を終了し、Manual モードになる

ラントレーシングオブジェクトへのコマンド

5.6.4運転制御

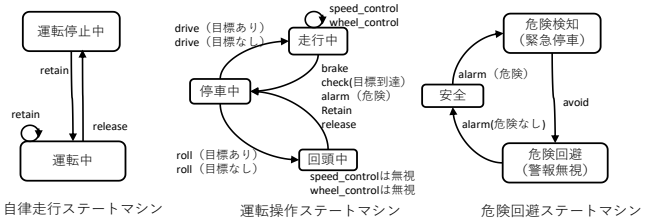
運転制御機能は、自律走行プログラムの指令に基づいて、アクセル・ブレーキ・ハンドル・方向指示器などを制御します。新しい指令を受けるまで同じ走行を続けますが、目標に到達したり、安全センサが警報を発したりしたときは停止します。多様な自律走行を実現するため、操作の数が多くなっています。

現時点で 4 つの自律走行プログラムがあるので、排他制御（ある自律走行プログラムが運転中は、他は運転できないようにすること）が必要です。操作 acquire で運転制御を始めます。操作 release で運転制御を解放し、他の自律走行プログラムが運転を制御できるようにします。自律走行プログラムは、acquire 時に自分の ID と、（指定距離の走行終了時などに）報告を受け取るためのコールバック関数（引数は下のような報告内容を示すコードで、自律走行プログラムが利用する）を渡すようにします。運転制御機能は、現在使用している自律走行プログラム以外の ID を持つ命令は拒否します。ただし、新しい ID での acquire は受け付けて、それまでの自律走行プログラムは自動的に release することにします。

コード名	報告内容
DRIVER_RELEASED	当該自動走行プログラムの運転制御が終了した（これ以上操作できない）
STOPPED_BY_BATTERY	電池電圧低下警報を検出したので停車した
STOPPED_BY_ALARM	安全警報を検出したので停車した
ALARM_CLEARED	安全警報が解除された
REACHED_GOAL	設定した走行距離に達したので停車した

コールバック関数に付随させる報告コード

動作を三つのステートマシンとして定義しておきます。状態遷移図を下に示します。左端は上に説明した排他制御、中央は運転操作、右端は安全確認状態です。



運転制御機能の状態遷移図

運転操作ステートマシンで、停車中状態からの運転開始手順を整理しておきます。

1. （必要なら）目標移動量を設定する
2. 速度と操舵量を設定する

3. 走行あるいは回頭を始める
4. 指令に従って速度と操舵量を変更する
5. 目標に到達したり、警報を検出したり、停車指令を受けたりしたら、停車する

走行中に速度（前進方向がプラス）や操舵量（任意単位で右向きがプラス。モーター回転速度に変換するときは、後で調整する係数を掛ける）を設定すれば、ただちに反映されます。回頭中には無視します（専用のモーター回転速度を使うため）。

走行あるいは回頭を始めれば、以下の要因のいずれかが発生するまで、動作を継続します。このうち、移動量と安全警報のチェックは周期 100ms の割り込みを使って実行します。その他の運転動作変更は自律走行プログラムから指令があったときに行います。

- 停車操作が行われた
- 設定されている目標移動量に到達した
- 安全警報が発生した
- 自律走行プログラムが acquire または release を行った

走行・回頭中に警報（崖、障害物、電源電圧）が発生すると緊急停車します。回避命令を与えると、電源電圧を除く警報を無視して、走行・回頭が行えるようになります。この回避命令は、警報がなくなれば、自動的に解除されます。

このモジュールには、シミュレーション機能を組み込んでおきます。シミュレーション機能とは、GPIO を使わずに位置や速度などを計算で求めることです（『仮想走行』と呼ぶことにします）。SIMULATED_DRIVE を#define すれば、下位モジュール（モータードライブ、走行計と LED）を組み込まずに、シミュレーションモデルに従って車両の位置と向きを刻々計算していきます。シミュレーションモデルの説明は付録を参照してください。

ファイル名	driver.py
クラス	driver
属性	説明（全て内部変数）
status	走行状態
speed	目標走行速度（%）
wheel	目標操舵量（%）
trip	走行距離目標（mm またはラジアン）
distance_to_go	残り走行距離（走行距離目標 - 実走行距離）
controller	運転を制御している自律走行プログラム（ID）
report_to	走行終了時の報告先
操作	説明
acquire	自律走行プログラムが運転制御を始める
release	自律走行プログラムが運転制御をやめる
speed_control	速度を設定する（走行中は即実行）
wheel_control	操舵量を設定する（走行中は即実行）
set_trip	目標移動量を設定する
drive	走行を開始する
roll	回頭を開始する
stop	停車する
get_status	現在の走行状態と残り走行距離を得る
get_speed	現在の速度と操舵量を得る
avoid	障害の回避を始める（安全警報を無視する）
control	安全確認をし、危険なら緊急停止や表示をする
simulation	車両の位置と向き、走行量を計算する
update	Web 上の速度表示と操舵量表示を更新する
beamer	前照灯を操作する

運転制御オブジェクト

方向指示器は、ほんとうは運転者が意図的に操作するものですが、ここでは操舵量がある程度大きくなった時（面舵 15 度以上）に点滅するようにします。後退灯とブレーキランプ、ハザードランプは走行状態に従って運転制御機能が操作します。

5.6.5 モータードライブ（デバイスハンドラ）

モータードライブ（AE-TB6612）のデバイスハンドラは、左右の動輪（A および B）ごとに 3 ビット（IN1、IN2、PWM）の信号で制御します。モーター駆動出力は、以下のように変化します。これ以外に STBY という制御信号がありますが、この信号を L レベルにすると、IN1=IN2=L と同じ出力になるので、回路上で H レベルにプルアップしています。OUT1 と OUT2 は（スイッチング MOS トランジスタを介して）駆動電源（VM = 5V）と GND に接続されますが、絶縁状態（OFF）にすることもできます。

IN1	IN2	PWM	OUT1	OUT2	モーターの状態
H	H	H/L	GND	GND	ショートブレーキ
L	H	H	GND	VM	右回転
		L	GND	GND	ショートブレーキ
H	L	H	VM	GND	左回転
		L	GND	GND	ショートブレーキ
L	L	H/L	OFF	OFF	ストップ

TB6612 の動作

「ショートブレーキ」状態では、モーターの両端が短絡されるので、モーターには回転を止める力が働きます。いっぽう「ストップ」状態では電流が全く流れず、車輪は自由に回転する（実際にはギアの抵抗があるので、自由回転ではない）ので、オートマ車の「ニュートラル」に近い状態になります。

実際のデバイスハンドラでは、GPIO を次表のように制御します。左右の制御信号が逆転しているのは、（モーターの接続端子を車台の内向きに取り付けたことによる）端子接続の違いを処理するためです。

モード	制御信号	AIN1	AIN2	BIN1	BIN2
	GPIO ポート	26	16	20	21
停車（ブレーキ）		H	H	H	H
ニュートラル		L	L	L	L
前進		L	H	H	L
後退		H	L	L	H
右へ回頭		H	L	H	L
左へ回頭		L	H	L	H

モータードライバの制御

デバイスハンドラの外部仕様を次に示します。

ファイル名	motor.py
クラス	motor_drive
属性	説明（全て内部変数）
mode	モーターの回転モード（前進、後退など）
操作	説明
set_speed	左右モーターの回転速度（%PWM）を設定する
drive	モーターを回転させる（始動する）
brake	モーターを止めてブレーキをかける（停車する）
neutral	モーターの入力を解放する（モーターは自由回転）
control	モーター制御信号を出力する（内部使用）

モーターハンドラオブジェクト

回転速度は PWM のデューティ（%）で、正の値は前進方向、負の値は後退方向の回転とします。また、回転速度はいつでも変更できることにします（停車時でも PWM は変えられるが、モーターは回転しない）。ただし、前進から後退、あるいは後退から前進といった急激な変更（危険な運転です！）は許さず、いったん停車と始動を行う必要があります。いまのところニュートラルの使いみちはありません。

5.6.6 走行計（デバイスハンドラ）

走行計は動輪の回転から移動距離（単位は mm）あるいは回頭角度（単位はラジアン）を求めるものです。ほんらい、エンジンの回転計はタコメーター、走行距離計はトリップメーターという、別々の装置です。しかし、この自律走行車には可変型の変速機がないので、両者にハードウエアとしての違いはありません。用途からトリップメーターという名前にしました。

ファイル名	tripmeter.py
クラス	tripmeter
属性	説明（全て内部変数）
mode	走行計の動作状態
pi	PIGPIO オブジェクト
piccolo	PICCOLO への通信チャンネル
操作	説明
enable	走行計を動作させる
disable	走行計を停止させる
retrieve	移動距離または回頭角度を読み出す
clear	測定値をクリアする

走行計オブジェクト

ロータリーエンコーダの回転は PICCOLO チップがカウントするので、ハンドラはそれを読み出して距離や角度に換算します。

PICCOLO チップの仕様は、PICCOLO 開発プロジェクト資料の付録を参照してください。走行計では、2 つの汎用入力#0 と#1 をイベント計数に使用します。

分解能を稼ぐため、光インターラプタ出力の立ち上がりと立下りの両方をカウントするようにしました。ロータリーエンコーダの穴が開いている部分とそうでない部分の大きさが等しいという保証はない（『ほぼ』ではある）ので、精度はあまり良くありません。計数値はパルス数の二倍になるので、走行距離は 5mm/カウント、回頭角度は 0.08 ラジアン（4.6°）/カウント（ともに滑りのない場合）となります。

5.6.7 LED（デバイスハンドラ）

LED は単純な回路ですが、いろいろな使い方をするので、操作が多くなっています。内部変数（属性）として、4 種類の LED（前照灯、左右の赤色 LED、後退灯）の点灯状態に加え、赤色 LED の点滅フラグを持つことで、曲がる方向とハザードを実現します。これらの属性だけでは、ハザードを解消した時ブレーキランプが消えてしまいますが、その時は回避走行をしているはずなので、問題ありません。

ファイル名	led.py
クラス	led_control
属性	説明
	（全て内部変数）
操作	説明
beamer	前照灯を操作する
backup_led	後退灯を操作する
right_led	右折灯を操作する
left_led	左折灯を操作する
right_turn	右折灯の点滅を操作する
left_turn	左折灯の点滅を操作する
hazard	ハザードランプの点滅を操作する
brake	ブレーキランプを操作する
blink	点滅のタイミングを知らせる

LED オブジェクト

5.6.8 センサのデバイスハンドラ

障害物センサ（デバイスハンドラ）

超音波式障害物センサの送受信を処理します。

ファイル名	us_sensor.py
クラス	obstacle_sensor
属性	説明
	（全て内部変数）
操作	説明
trigger	超音波を送信する (パルス幅 20μs)
distance	障害物までの距離（mm 単位）とコードを返す

障害物センサオブジェクト

PICCOLO チップの計数クロックを 250kHz(4μs)にして、計数値から障害物までの距離を、15℃の音速を使って求めます：

距離 = 4μs × 計数値 × 340.5m/s ÷ 2

空気中の音速は 1℃あたり約 0.2%変化するので、距離計として使うときは、気温を測定して補正するのが普通です。自律走行車では、近くに障害物があるかどうかを知るだけなので、温度補正はしません。

操作 distance は、距離測定結果を取り出し、障害物までの距離と、次の表に示すコードのタプルを返します。

距離コード	コードの意味と障害物までの距離
Clear	安全（500mm 以上あるいは障害物なし）
Marginal	ほぼ安全（250mm～500mm）
Warning	警告レベル（100mm～250mm）
Alert	警報発生（100mm 以内）
Not sure	距離が測定できない（距離データは使用不可

検出した障害物までの距離コード

trigger 実行から distance 実行までには、最大距離を 10m として往復時間 60ms 以上取る必要があります。割り込み処理の始めに distance で前回の結果を取り込み、そのあと trigger を実行するようにします（1 回目だけは割り込み開始前に trigger を実行しておく）。

走路センサ・崖センサ（デバイスハンドラ）

走路センサと崖センサは同種の光センサを使っており、発光制御回路が共通なので、ひとつのオブジェクトにまとめました。

ファイル名	optosensors.py
クラス	reflectometer
属性	説明
	（全て内部変数）
操作	説明
read_line	走路センサを読み取り、ビット列で返す
read_cliff	崖センサを読み取る
activate	反射光センサを動作させる
deactivate	反射光センサの動作を止める

崖センサ・走路センサオブジェクト

検出感度は半固定抵抗で調整できます。検出結果を LCD あるいは Web に表示させながら行うことにします。

走路センサの出力は、左端のセンサが最上位（ビット#4）の、右端が最下位（ビット#0）のビット列とします。光センサが黒線を検出したとき 1 を、白（反射光が強い）のときに 0 を返すことにします。

ライントレーシング走行（走路センサを使用する）時には、黒いガイドラインを『崖』と誤認する可能性があります。これに対応するため、どの自律走行プログラムが走っているかを知っている上位（具体

的にはルート）に、read_cliff を呼ぶかどうか決めさせることにします。

電源監視（デバイスドライバ）

電源監視オブジェクトの外部インターフェースはごく単純です。

ファイル名	adc.py
クラス	voltage_meter
属性	説明
	（外部から見える属性はない）
操作	説明
read	電池電圧を読み取る
close	I2C 通信を終了する（これ以降は使えない）

電源監視オブジェクト

ここで使った A/D コンバータ MCP3425 は内部レジスタ用のアドレスがないので、I2C アドレスの機器へ（内部アドレスを指定せず）直接アクセスします。

1 バイトとして書き込む制御レジスタの構造は以下のとおりです。16 ビットの連続変換、内蔵アンプ 1 倍で使用します。

7	6	5	4	3	2	1	0
RDY/	C1	C0	M	S1	S0	G1	G0

RDY/ ：データ更新ステータス（0：更新済、1：未更新）
C1-C0 ：未使用
M ：変換モード（0：ワンショットモード、1：連続モード）
S1-S0 ：分解能と変換レート（00：16bit/240sps、01：14bit/60sps、 10：16bit/15sps）
G1-G0 ：内蔵アンプ利得（00：1 倍、01：2 倍、10：4 倍、11：8 倍）

制御レジスタ

同じ I2C アドレスを読み出すと、次のような順番で 3 バイトのデータが返ってきます。

測定値上位バイト	測定値下位バイト	制御レジスタ

読み出しデータ

これを pigpio の i2c_read_device で受信すると、バイト数と受信バイト列のタプルを返してくれます。デバイスドライバで 16 ビットデータ（符号付きなので実際には 15 ビット）に再構成し、係数をかけて電圧を求めます。

データ／テキスト変換関数

LCD や Web ブラウザに表示するテキストを作る関数 hmi_format.py を用意しました。テキストは Redis FIFO を通して、HMI プロセスに渡します。

コラム　複数のプロセスから GPIO を使う

今回のプロジェクトでは、二つのプロセス（自律走行車と HMI）から GPIO を使います。インターネット上には「複数の C 言語プロセスから pigpio にアクセスしようとするとエラーになる」という『体験談』が多くあります。どうもこれは、それぞれの C プログラムが自分でライブラリ（一つしか存在できない）を立ち上げようとしていたのが原因のようです。最初に（ライブラリを実行する）デーモンを走らせておけば、各プロセスはデーモンとの通信を使って GPIO を使えるはずです。

それを確認するため実験を行いました。プロジェクトファイルにある test_pigpio.py は、車両の右折表示灯を 1 秒ごとに点滅させるプログラムです。cpp 処理するときに-D オプションで LEFT を指定すると左折表示灯が、SLOW を指定すると点滅間隔が 3 秒になります。

そのまま実行（右折灯が点滅します）してから、LEFT を指定して別プロセスを立ち上げると、左折灯も点滅するようになります。これで、複数プロセスから PIGPIO へのアクセスが可能であることが確認できました。

```
$cpp -DBCM2835 test_pigpio.py |python &
$cpp -DBCM2835 -DLEFT test_pigpio.py |python

（動作を確認したら両方のプロセスを停める）
```

今度は SLOW を指定したプロセスを走らせて（右折灯が 3 秒ごとに点滅します）から、何も指定せずに別プロセスを立ち上げると、（ほぼ）1 秒ごとに点滅するようになります。あとの方のプロセスを停めると、また 3 秒間隔に戻ります。これは、複数のプロセスが同じ GPIO を『共有』できる（排他制御していない）ことを示しています。ちょっとトリッキー（というより危険）なことができそうですが、今回のプロジェクトでは行いません。

```
$cpp -DBCM2835 -DSLOW test_pigpio.py |python &
（右折灯が 3 秒周期で点滅する）
$cpp -DBCM2835 test_pigpio.py |python
（点滅周期が 1 秒になる）
(Crtl-C で後のプロセスを停めると、点滅周期が 3 秒に戻る）
```

このように、複数のプロセスから GPIO を使えるので、システム設計がやりやすくなります。デーモンとの通信に使うソケットは、プロセス毎に用意されるので、プロセス間で矛盾が起こる心配はありません。ただ「同じプロセス内」で不用意にソケットを使うと問題が起こるので、注意が必要です。

ファイル名	hmi_format.py
関数	説明
format_voltage	電圧データを 19.9V 形式テキストに変換する
format_sensors	走路・崖センサの出力をテキストに変換する
ファイル名	ftos.py
関数	説明
ftos	浮動小数点数データをテキストに変換する

データ／テキスト変換関数

エンディアン補正関数

通信時のエンディアン不整合を補正する関数です。

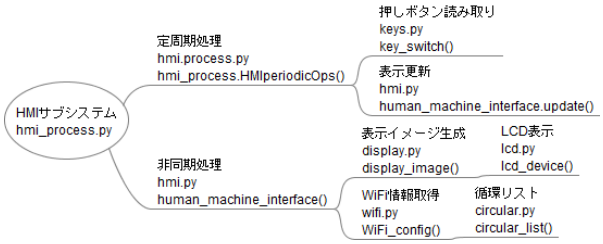
ファイル名	endian.py
関数	説明
fix_endian	ビッグエンディアンで受信したデータを内部用のリトルエンディアンに変換する

エンディアン補正関数

5.7　手動操作（HMI）プロセスの仕様設計

5.7.1 ファイル（オブジェクト）の階層構造

手動操作（HMI）プロセスは、自律走行車プロセスとは独立して並列に動作させます。プロセスを構成するファイル（オブジェクト）の階層（生成・コールする順番）を次の図のように構成します。機能、ファイル名、オブジェクト名を三行に表現してあります。



HMI サブシステムの構成

5.7.2 Redis インターフェースと定周期処理

HMI は他のプロセスと Redis を介してデータを交換します。キーと用途は以下のとおりです。このうち黄色に着色したのは HMI プロセスがデータを受けるキー、緑色はデータを送るキーです。「アクセス」欄で FIFO とあるのはリスト（push/pop）として、DATA とあるのは共有テキスト変数（set/get）として使うことを示しています。

キー	アクセス	説明
REDIS_TO_HMI	FIFO	ボタン操作等のコマンド受信
REDIS_FOR_BATTERY	DATA	電池電圧
REDIS_FOR_OBSTACLE	DATA	障害物までの距離を表すテキスト
REDIS_FOR_SENSORS	DATA	反射光センサの出力表示
REDIS_TO_COMMAND	FIFO	自律走行車へのコマンド
REDIS_TO_LCD__TOP	FIFO	Web 上の LCD1 行目表示更新
REDIS_TO_LCD_BOTTOM	FIFO	Web 上の LCD2 行目表示更新
REDIS_TO_IFRAME	FIFO	自律走行プログラム用ページ変更

HMI の Redis インターフェース

hmi_process.py の機能は以下の 3 つです：

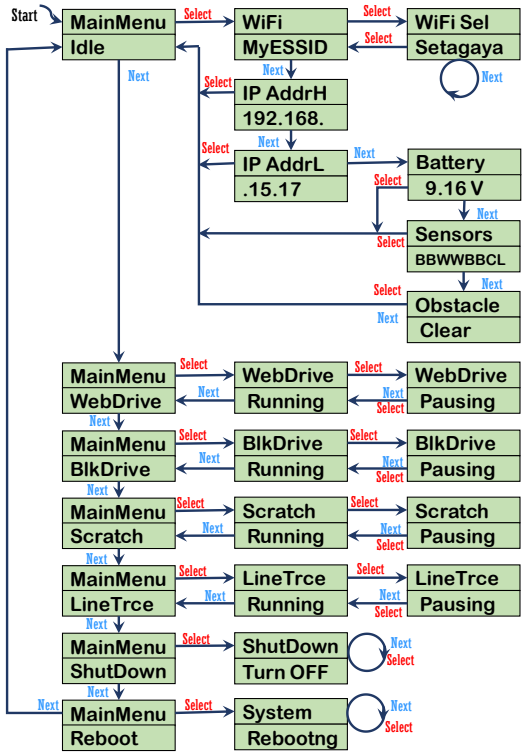
1. Redis FIFO の入力を取り出して HMI ステートマシンに渡す
2. キースイッチを読み取り、押下を検出したら自分あての Redis FIFO に入れる
3. LCD にセンサデータを表示しているときは、更新タイミングを HMI に渡す

最初のうちは、2 と 3 は定周期割り込みで処理していたのですが、pigpio の割り込み問題に対処するため、全体を「適当な間隔でチェックする」ように変更しました。キースイッチの検出周期は、それほど正確である必要がないため、割り込みは使いません。キースイッチの押下はそのまま処理せず、Web 画面でのクリックと同じ処理ができるように、自分で Redis FIFO に入れることにします。

5.7.3 HMI ステートマシン

ヒューマン・マシン・インターフェース（HMI）機能は、キースイッチなどを介したヒトの操作コマンドにより、表示やプログラムの実行を行う機能です。

おもな操作は、『NEXT』と『SELECT』という 2 個のキースイッチ入力で行います。その順番によって、2 行の液晶表示器（LCD）に表示されるテキストを、次の図のように変えていきます。表示内容は内部で記憶しているので、こういう機能は「ステートマシン」（あるいは「有限オートマトン」、「状態遷移機械」）と呼ばれています。おのおのの LCD 表示が「状態」を、入力（ボタン操作）によってどの状態に移るかを示しているので、次ページの図を「状態遷移図」といいます。



ボタン操作による HMI の状態遷移

状態遷移図の上半分は、WiFi の SSID と IP アドレス、センサの出力を表示させるためのものです。センサ出力を表示するときは Redis（FIFO ではなく共有テキスト変数）に書き込まれたテキストを使います。

中ごろの部分は 4 つの自律走行プログラムを選んで実行・停止させます。下側の 2 行は、シャットダウンと再起動（リブート）を行います。

キー入力による状態遷移と、それに伴う動作（作用といいます）を表にまとめたものが、次ページの状態遷移表（基本部）です。

作用欄に「自動更新」とある状態では、定周期処理のなかで Redis にあるテキストを LCD の 2 行目に転送します。

次ページの状態遷移表では基本部としてキー入力による遷移だけを示しています。これ以外に、Web サーバーからのコマンド（Update, 自律走行プログラムの実行命令と終了命令、Reboot, Shutdown）と自律走行車プロセスからのコマンド（電池電圧低下によるシャットダウン要求）を処理します。Update 以外のコマンドは、作用の記述を繰り返すのを避けるため、現在の状態を終状態直前にしてから、ステートマシンに遷移キーコマンドを与えるようにしました（再帰呼び出し）。

始状態	コマンドによる作用	終状態
Any	Update: LCD イメージを Web サーバーに再送する	始状態と同じ
Any	Reboot: システムを再起動する	System Rebooting
Any	Shutdown: システムをシャットダウンする	ShutDown Turn OFF
Any	Battery Low: 電池交換を求めて、システムをシャットダウンする	ShutDown Turn OFF
Any	自動走行プログラムの名前: 自律走行車プロセスに当該プログラ自律走行ムの実行命令を送る	当該プログラム Running
自律走行プログラムが実行中／休止中	Exit: 自律走行車プロセスに自律走行プログラムの終了命令を送る	MainMenu 当該プログラム

HMI ステートマシンの状態遷移表（例外処理）

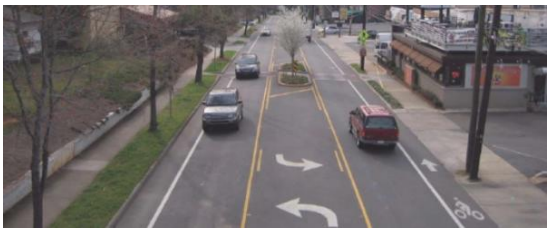
ステートマシンは次のファイルとして作成します。

ファイル名	hmi.py
クラス	human_machine_interface
属性	説明（全て内部変数）
state	操作表示の状態
display	表示機能オブジェクト
wifi	WiFi 環境オブジェクト
操作	説明
state_machine	HMI ステートマシンを実行する
sensor_update	LCD 上のセンサ指示値を更新する

HMI ステートマシンオブジェクト

コラム　海外の交差点 6: 左折レーン

日本の道路を走っているとき、右側にあるコンビニに入ろうとして、後続車を渋滞させてしまったことはありませんか？アメリカでは右側通行なので、交差点でないところで安全に左折する方法についてです。



道路の中央に黄色の線で囲まれた「左折レーン」があります。左折サインを出しながら、いったんこのレーンに入ります。対向車があれば停車して待ち、途切れたときに曲がります。対向車も左折することがあるので、このレーンを走り続けてはいけません。

始状態	上段：Next キー押下による作用 下段：Select キー押下による作用 （作用が同じ場合は 1 段で記載する）	終状態
電源投入	LCD1 行目に”MainMenu”と表示 LCD2 行目に”Idle”と表示	MainMenu Idle
MainMenu Idle	LCD2 行目に”WebDrive"と表示	MainMenu WebDrive
	LCD1 行目に”WiFi”と表示 現在の ESSID を LCD2 行目に表示	WiFi MyESSID
WiFi MyESSID	LCD1 行目に”IP AddrH”と表示 IP アドレス上位を LCD2 行目に表示	IPAddrH 192.168
	LCD1 行目に”WiFi Sel”と表示 現在の WiFi 環境を LCD2 行目に表示	WiFi Sel Setagaya
WiFi Sel Setagaya	次の WiFi 環境を LCD2 行目に表示	WiFi Sel Setagaya
	WiFi 環境ファイルを書き換える LCD1 行目に”WiFi”と表示 現在の ESSID を LCD2 行目に表示	WiFi MyESSID
IPAddrH 192.168	LCD1 行目に”IP AddrL”と表示 IP アドレス下位を LCD2 行目に表示	IPAddrL .15.17
	LCD1 行目に”MainMenu”と表示 LCD2 行目に”Idle”と表示	MainMenu Idle
IPAddrL .15.17	LCD1 行目に”Battery”と表示 電源電圧を LCD2 行目に表示 電源電圧の表示は自動更新する	Battery 9.16V
	LCD1 行目に”MainMenu”と表示 LCD2 行目に”Idle”と表示	MainMenu Idle
Battery 9.16V	LCD1 行目に”Sensors”と表示 光センサ入力を LCD2 行目に表示 光センサ入力の表示は自動更新する	Sensors BBWWBBCL
	LCD1 行目に”MainMenu”と表示 LCD2 行目に”Idle”と表示	MainMenu Idle
Sensors BBWWBBCL	LCD1 行目に”Obstacle”と表示 超音波センサ入力を LCD2 行目に表示 超音波入力の表示は自動更新する	Obstacle Clear
	LCD1 行目に”MainMenu”と表示 LCD2 行目に”Idle”と表示 走路センサの動作停止を要求する	MainMenu Idle
Obstacle Clear	LCD1 行目に”MainMenu”と表示 LCD2 行目に”Idle”と表示	MainMenu Idle
MainMenu WebDrive	LCD2 行目に”BlkDrive”と表示	MainMenu BlkDrive
	WebDrive 実行コマンドを発行 LCD1 行目に”WebDrive”と表示 LCD2 行目に”Running”と表示 Web 画面を Web_drive 用に変更	WebDrive Running
WebDrive Running	WebDrive 終了コマンドを発行 LCD1 行目に”MainMenu”と表示 LCD2 行目に”WebDrive”と表示 Web 画面をデフォルト用に変更	MainMenu WebDrive
	WebDrive 休止コマンドを発行 LCD2 行目に”Pausing”と表示	WebDrive Pausing
WebDrive Pausing	WebDrive 再開コマンドを発行 LCD2 行目に”Running”と表示	WebDrive Running

始状態	上段：Next キー押下による作用 下段：Select キー押下による作用 （作用が同じ場合は 1 段で記載する）	終状態
MainMenu BlkDrive	LCD2 行目に”Scratch”と表示	MainMenu Scratch
	BlockDrive 実行コマンドを発行 LCD1 行目に”BlkDrive”と表示 LCD2 行目に”Running”と表示 Web 画面を block_drive 用に変更	BlkDrive Running
BlkDrive Running	BlockDrive 終了コマンドを発行 LCD1 行目に”MainMenu”と表示 LCD2 行目に”BlkDrive”と表示 Web 画面をデフォルト用に変更	MainMenu BlkDrive
	BlockDrive 休止コマンドを発行 LLCD2 行目に”Pausing”と表示	BlkDrive Pausing
BlkDrive Pausing	BlockDrive 再開コマンドを発行 LCD2 行目に”Running”と表示	BlkDrive Running
MainMenu Scratch	LCD2 行目に”LineTrce”と表示	MainMenu LineTrce
	ScratchDrive 実行コマンドを発行 LCD1 行目に”Scratch”と表示 LCD2 行目に”Running”と表示 Web 画面を scratch_drive 用に変更	Scratch Running
Scratch Running	ScratchDrive 終了コマンドを発行 LCD1 行目に”MainMenu”と表示 LCD2 行目に”Scratch”と表示 Web 画面をデフォルト用に変更	MainMenu Scratch
	ScratchDrive 休止コマンドを発行 LLCD2 行目に”Pausing”と表示	Scratch Pausing
Scratch Pausing	ScratchDrive 再開コマンドを発行 LLCD2 行目に”Running”と表示	Scratch Running
MainMenu LineTrce	LCD2 行目に”ShutDown”と表示	MainMenu ShutDown
	LineTrace 実行コマンドを発行 LCD1 行目に”LineTrce”と表示 LCD2 行目に”Running”と表示 Web 画面を line_trace 用に変更	LineTrce Running
LineTrce Running	LineTrace 終了コマンドを発行 LCD1 行目に”MainMenu”と表示 LCD2 行目に”LineTrce”と表示 Web 画面をデフォルト用に変更	MainMenu LineTrce
	LineTrace 休止コマンドを発行 LLCD2 行目に”Pausing”と表示	LineTrce Pausing
LineTrce Pausing	LineTrace 再開コマンドを発行 LCD2 行目に”Running”と表示	LineTrce Running
MainMenu ShutDown	LCD2 行目に”Reboot”と表示	MainMenu Reboot
	LCD1 行目に”Shutdown”と表示 LCD2 行目に”TurnOFF”と表示 システムコマンド Shutdown を実行	ShutDown Turn OFF
ShutDown Turn OFF	システムコマンド実行中なのでキー入力を受け付けない	ShutDown Turn OFF
MainMenu Reboot	LCD2 行目に”Idle”と表示	MainMenu Idle
	LCD1 行目に”System”と表示 LCD2 行目に”Rebootng”と表示 システムコマンド Reboot を実行	System Rebooting
System Rebooting	システムコマンド実行中なのでキー入力を受け付けない	System Rebooting

HMI ステートマシンの状態遷移表（基本部）

5.7.4 表示イメージ生成

表示イメージ生成機能は、テキストを LCD と Web 画面の指定した位置にはめ込みます。動作シミュレーション（#ifdef HMI_SIMULATION）時には、二行の LCD 表示イメージを PC のコンソール画面に表示します。また、表示イメージを Web サーバーに転送します。

ファイル名	display.py
クラス	display_image
属性	説明
	（全て内部変数）
操作	説明
fill_field	テキストを指定位置にはめ込む
show	上の簡易版マクロ（行のみ指定）
update	現在の表示イメージをブラウザに送る

表示イメージ生成オブジェクト

Web サーバーは、新しいブラウザが接続してきたら、LCD イメージの更新を要求してきます（サーバーは覚えていない）。操作 update がこれに対応します。

5.7.5 LCD（デバイスハンドラ）

温度コントローラ・プロジェクト以来使っている LCD のデバイスハンドラです。pigpio ライブラリ用に書き換えましたが、基本設計は変えていません。一行の文字数や行数の異なる多くの表示器をカバーしていますが、今回は 8 文字×2 行（#define LCD8）を選んで使います。

ファイル名	lcd.py
クラス	lcd_device
属性	説明
	（全て内部変数）
操作	説明
put_string	指定位置に表示テキストを送信する
close	LCD との I2C 通信を終了させる

LCD オブジェクト

LCD のレジスタ構造や書き込み手順は、温度コントローラ・プロジェクトの説明書を参照してください。LCD への書き込み中はタイマー割り込みを禁止するようにしましたが、実際には使っていません（HMI プロセスでは割り込みを使わない）。

5.7.6 WiFI 環境情報の取得と設定

自律走行車が使っている WiFi 環境情報を取得・表示します。SSID や IP アドレス（8 文字 LCD に表示するため、上位 2 バイトと下位 2 バイトに分ける）は、ブラウザに接続先を知らせるのに必要な情報です。

Raspberry Pi に直接ディスプレーとキーボードを繋げれば、PC のように、周りの WiFi 環境を調べて接続することができます。しかし自律走行車のような構成では望むべくもありません。第 3 章で説明した設定ファイル wpa_supplicant.conf と dhcpcd.conf をあらかじめ用意しておく必要があります。走行する場所（WiFi ルーター）にあわせて、あらかじめ何通りかのファイルを用意しておき、キースイッチと LCD だけを使って選べるようにしてあります。HMI 状態遷移図の一番上の行は、そのために用意しました。接続するルーターを変更したときは、WiFi サービスを再起動します。

ファイル名	wifi.py
クラス	WiFi_config
属性	説明（全て内部変数）
router	現在使用中の WiFi ルーター
ssid	現在使用中の WiFi SSID
ip_address	現在使用中の IP アドレス
操作	説明
get_router	現在のルーター名を取得する
next_router	次のルーター候補を取得する
set_router	新しいルーターを設定する（再起動する）
get_ssid	現在使用中の SSID を取得する
get_ip_high	IP アドレスの上位 2 バイトを取得する
get_ip_low	IP アドレスの下位 2 バイトを取得する

Wifi 環境情報の取得・設定オブジェクト

ルーターの候補を管理するため、循環型のリストとしてルーター名を保存しておきます。「次の候補」を取り出すとき、リストの最後から最初に戻るためです。

ファイル名	circular.py
クラス	circular_list
属性	説明（全て内部変数）
clist	循環リスト
pos	リスト中の現在位置
操作	説明
next	次の要素を取り出す（内部使用）
find	指定した要素の位置を返す
set	指定した要素の位置を現在位置にする

循環型リストオブジェクト

5.7.7 キースイッチ（デバイスハンドラ）

キースイッチが押されているかを調べます。約 300ms 毎に読み取って状態変化を調べるので、チャ

タリング防止処理は行いません。押されたボタン名は一度だけ返し、（自律走行車では不要な）長押しには対応しません。

ファイル名	keys.py
クラス	key_switch
属性	説明
	（外部から見える属性はない）
操作	説明
key_status	キーが押されているか調べる
status_next	NEXT キーを読み取る
status_select	SELECT キーを読み取る
raw_key_status	キー接点の状態を読み取る（HMI では使わない）

キースイッチオブジェクト

5.8　共通インクルードファイル

5.8.1 インポート代替

Python のライブラリをインポートするためのファイルです。複数のモジュールで同じライブラリを使う必要がある時、重複インポートを避けるために用意しました。具体的な内容はプロジェクトファイルを見てください。インクルードファイルによっては、実際には使わないライブラリも同時にインポートしてしまうものもありますが、気にしないことにします。

use-redis.h では、検証時に Redis に書き込む代わりに、何を書き込もうとしたかを表示するダミーオブジェクトを定義しています。

インクルードファイル	使えるようにするライブラリ
use-math.h	math（数値演算）
use-redis.h	redis（Redis データベース、FIFO）
use-RegularExpression.h	re（正規表現によるテキストの解析）
use-sys.h	sys, os, signal, subprocess
use-time.h	time, sleep, datetime
use-itimer.h	signal, suspended_signals

インポートを代替するインクルードファイル

5.8.2 pigpio インターフェース

use-pgpio.h は、pigpio ライブラリを使用するための宣言です。ファイルが大きいので、ここでは掲載しません。プロジェクトファイルを見てください。

BCM2835 を#define すると、すべての IO 動作は pigpio のライブラリをコールするようになります。#define しないと、すべての IO 動作を検証用のコンソール入力と表示で代替します。その時、GPIO_STUB を#define しておくと、キーボード入力を行わず、適当な入力を返します。

use-pigpio.h

（プロジェクトファイルを参照）

5.8.3 GPIO のアサイン

GPIO ピンのアサインを定義します。

gpio.h（抜粋）

```
/* BCM2835 GPIO 定義
  初版:  2014/12/28 Chuji
  最新版:2021/7/23 PICCOLO チップを採用
  履歴：2020/9/3 車両ロボット用の定義を追加
  注：pigpio デーモンの使用を前提とする
  その場合は、このファイルの前に use-pigpio.h をインクルードすること
*/
#ifndef __GPIO
#define __GPIO

/* GPIO ポートの用途は以下のとおり
  GPIO0 (pin27):  ID_SD --- EEPROM は使わない
  GPIO1 (pin28):  ID_SC --- EEPROM は使わない
  GPIO2 (pin03):  I2C SDA *** I2C バス
  GPIO3 (pin05):  I2C SCL *** I2C バス
  GPIO4 (pin07):  BEAMER *** ヘッドライト駆動（Active H）
  GPIO5 (pin29):  ACT_OPTO *** 反射光センサ駆動（Active H）
  GPIO6 (pin31):  SELECT *** Select ボタン（Active H）
  GPIO7 (pin26):  LT5 *** ライントレース入力 #5
  GPIO8 (pin24):  LT4 *** ライントレース入力 #4
  GPIO9 (pin21):  LT1 *** ライントレース入力 #1
  GPIO10(pin19):  ACT_TRIP *** 回転計 LED 駆動信号（Active H）
  GPIO11(pin23):  LT3 *** ライントレース入力 #3
  GPIO12(pin32):  PWM_R *** 右モーター用 PWM
  GPIO13(pin33):  PWM_L *** 左モーター用 PWM
  GPIO14(pin08):  US_TRIG *** 超音波センサ駆動信号（Active H）
  GPIO15(pin10):  SYS_CLK *** システムクロック
  GPIO16(pin36):  AIN2 *** 右モーター制御信号 #2
  GPIO17(pin11):  BACK *** バックライト照明信号（Active H）
  GPIO18(pin12):  RIGHT *** 右折灯（Active H）
  GPIO19(pin35):  NEXT *** Next スイッチ（Active H）
  GPIO20(pin38):  BIN1 *** 左モーター制御信号 #1
  GPIO21(pin40):  BIN2 *** 左モーター制御信号 #2
  GPIO22(pin15):  ISR *** 割り込み処理確認用
  GPIO23(pin16):   *** 未使用
  GPIO24(pin18):  CLIFF *** クリフセンサ（Active L）
  GPIO25(pin22):  LT2 *** ライントレース入力 #2
  GPIO26(pin37):  AIN1 *** 右モーター制御信号 #1
  GPIO27(pin13):  LEFT *** 左折灯（Active H）
*/
/* GPIO 信号名の指定 */
#define GPIO_I2C  1 /* I2C チャンネル #1 */
#define GPIO_SDA  2
#define GPIO_SCL  3
#define GPIO_SYS_CLK  15
#define GPIO_US_TRIG  14
#define GPIO_PWM_R  12
#define GPIO_PWM_L  13
#define GPIO_AIN1  26
#define GPIO_AIN2  16
#define GPIO_BIN1  20
#define GPIO_BIN2  21
#define GPIO_SELECT  6
（後略）
```

5.8.4 Redis インターフェース

Redis のキーと、各言語からの使用方法を定義します。C 言語インターフェースは秘書ロボット・プロジェクトで使ったものですが、次のプロジェクトでも使う予定なので残しています。

redis.h（抜粋）

```
/* REDIS データベースの定義
   初版： 2018/2/14 Chuji
   最新版：2022/12/25 --- JavaScript 用インターフェースを追加
                  2020/9/8 ... バギー用キーを追加
   C 言語で使う定義のみ文字列を二重クオーテーションで囲む
*/
#ifndef __REDIS
#define __REDIS
/* クライアント用 Redis インターフェース */
/* Python コード */
#define REDIS_SERVER  host = 'localhost'
#define REDIS_PORT  port = 6379
（中略）
/* HMI プロセスへの操作命令キー */
#define REDIS_TO_HMI    'Hmi'
/* HMI 向けの表示キー */
#define REDIS_FOR_BATTERY  'Battery'
#define REDIS_FOR_OBSTACLE  'Obstacle'
#define REDIS_FOR_SENSORS  'Sensors'
/* Web ブラウザ向けの更新キー */
/* HMI からの更新要求 */
#define REDIS_TO_LCD_TOP  'Lcd0'
#define REDIS_TO_LCD_BOTTOM  'Lcd1'
#define REDIS_TO_IFRAME    'Iframe'
#define REDIS_TO_INDICATOR  'Led'
/* 自律走行プログラムから Web ページへの共通更新キー */
#define REDIS_TO_SPEED    'Speed'
#define REDIS_TO_WHEEL    'Wheel'
#define REDIS_TO_ALARM    'Alarm'
/* 自律走行プログラムからの応答キー */
#define REDIS_TO_RESPONSE  'Response'
/* 自律走行プログラムから Scratch プログラムへの応答キー
*/
#define REDIS_TO_SCRATCH  'Scratch'
/* 温度コントローラのトレンド表示用 */
#define REDIS_TO_TREND    'Trend'
（後略）
```

5.8.5 数値演算

my_round,.py は数値演算結果の四捨五入を定義します。プロジェクトファイルを参照してください。

5.8.6 PID 制御

pid.h は PID 演算に使うパラメータを定義します。また、process_value.h は、PID 演算に使うデータ型（値と値の有用性）を定義します。温度コントローラで使ったものなので、そちらの報告書あるいはプロジェクトファイルを参照してください。

5.8.7 PICCOLO チップ

PICCOLO チップのレジスタ構成や、コマンド/ステータスレジスタのビットパターンを定義します。

piccolo.h（抜粋）

```
/* piccolo.h PICCOLO チップの使用法に関する定義
   初版：2021/7/22 Chuji
   最新版：
*/
#ifndef __PICCOLO
#define __PICCOLO
/* PICCOLO チップの I2C アドレス */
#define PICCOLO_ADDRESS    0x38
/* 内部レジスタ番号 */
#define PICCOLO_T_STATUS  0
#define PICCOLO_A_STATUS  1
#define PICCOLO_T_COMMAND  2
#define PICCOLO_A_COMMAND  3
#define PICCOLO_WIDTH    4
#define PICCOLO_COUNT    6
#define PICCOLO_DATA0    8
#define PICCOLO_DATA1    10
#define PICCOLO_DATA2    12
#define PICCOLO_DATA3    14
（中略）
/* コマンドレジスタのビットパターン定義 */
#define PICCOLO_WIDTH_DISABLE  0
#define PICCOLO_WIDTH_ENABLE  1
#define PICCOLO_WIDTH_SYNC  0
#define PICCOLO_WIDTH_ASYNC  2
#define PICCOLO_WIDTH_GATE_H  0
#define PICCOLO_WIDTH_GATE_L  4
#define PICCOLO_WIDTH_CLOCK_8us  0
#define PICCOLO_WIDTH_CLOCK_2us  8
/* デフォルトは同期式、H レベルゲート、クロック 8us */
#define PICCOLO_WIDTH_DEFAULT
PICCOLO_WIDTH_ENABLE
（後略）
```

コラム　通学児童の保護

通学児童の安全には気を使います。スクールバスは黄色に塗装されている国が多いのですが、日本は例外的なのでしょうか？　バスを降りた後、陰から道路を横断しようとする子がいるとドキッとしますね。アメリカのスクールバスは降車中を知らせるため、赤色灯を点けたり、STOP マークを横に張り出したりします。このとき追い越す車だけでなく、対向車も停止して待たなければなりません（片側 2 車線以上のときは左車線のみ）。

学校の近く（スクールゾーン）では、登下校時間帯（右にように信号が点滅する。時間帯が表示されているだけの場所もある）の制限速度が厳しく制限されます（この地区では時速 30km）。いずれも違反したときの罰金は増額（2 倍になるところも）されます。それだけ事故が多かったということです。

6　ソフトウエアの設計（2）コーディングと検証

前章で検討したシステム構成に基づき、プログラムのコーディングと検証を行います。上位モジュールの検証に下位モジュールを使うことがあるので、前章とは逆に下位モジュールから順番に説明していきます。

なお、この本でファイルを掲載するとき、スペースの都合でタブを空白 2 文字に変え、空白行を削除しました。プロジェクトファイルの方が見やすいので、ダウンロードしてご覧ください。

検証目的でプログラムの動作を調整するため、マクロ定義を使用します。下位モジュールの STUB 化については前章で説明しました。自律走行車プロセスでは、それ以外に次のようなマクロ定義を使っています。HMI プロセスでのマクロ定義は後で説明します。

マクロ名	#ifdef
DEBUG	デバグ目的でパラメータを表示する
ADC_TEST	A/DC から読んだレジスタ値を表示する

その他のマクロ定義

まず自律走行車プロセスを、前章で検討したオブジェクト仕様に従って設計します。外部から見えない属性と操作の説明は、プログラム冒頭にまとめてあるので、詳しい説明は省略します。設計どおりにコーディングした箇所はいちいち説明しません。

各モジュールの検証プログラム test_xxx.py の大部分は、この本のページ数の制約で掲載していません。プロジェクトファイルを参照してください。

6.1　デバイスハンドラと下位ルーチン

6.1.1 データ／テキスト変換関数

最下位ルーチンとして、各種データをテキストに変換する関数群です。デバイスドライバの検証にも使うので、最初に説明します。変換機能は以下の三種類です。

関数名	入力パラメータ	出力テキスト
format_voltage	電圧	99.9 V
format_sensors	走路・崖センサ出力	WWBBW Ok

データ／テキスト変換関数

プログラムファイル

format_voltage()は、浮動小数点数／テキスト変換関数 ftos()（温度コントローラで検証済）を呼び出すだけです。format_sensors()は走路センサのデータを 5 文字（白い部分は W、黒い部分は B。左側センサから順番）に、崖センサデータをそれに続く 2 文字（安全時には Ok、危険時には No）に変換しています。

hmi_format.py

```
/* hmi_format.py バギー用書式設定
  初版：2020/9/9 Chuji
  最新版：
関数：
  format_voltage  電圧をテキスト'99.9 V'に変換
  format_sensors  光学センサ入力レベルをテキスト'WWBBW
Ok'に変換
*/
#ifndef __HMI_FORMAT  /* 複数のモジュールで使う可能性が
あるので一度だけインクルードする */
#define __HMI_FORMAT
#include "include/use-math.h"
#include "include/process_value.h"
#include "include/display.h"
#include "include/adc.h"
#include "include/sensor.h"
#include "include/web_page.h"
#include "ftos.py"
#define DATA_INVALID  ' Unknown'
def format_voltage(x):
  if not BATTERY_BAD_DATA(x):
    s = ftos(x, FORMAT_99_9)
    s += ' V  '
  else:
    s = DATA_INVALID
  return s
/* ラインセンサ出力を 5 文字のテキストに変換する */
def format_line(l):
  s = ''
  /* l[0](右端)を最下位にするため、左端から調べていく */
  for i in range(LINE_NUM_SENSORS):
    if Get_MSB(l) == LINE_MSB:
      s += DISP_BLACK
    else:
      s += DISP_WHITE
    l <<= 1
  return s
/* 反射光センサの出力をテキストに変換する */
def format_sensor(l, c): /* l: ラインセンサ配列、c:
クリフセンサ */
  /* まずラインセンサデータを変換する */
  s = format_line(l)
  /* 崖センサの状態を表す文字を最後に付ける */
  if c == DISP_SAFE:
    s += DISP_CLEAR
  else:
    s += DISP_CLIFF
  return s
#endif
```

モジュール検証

検証スタブ test_hmi_format.py は、引数を変えて各関数を呼び出し、出力テキストを表示します。

format_voltage()が四捨五入して表示できること、format_sensors()が 1（反射あり）と 0（反射なし）を W と B、Ok と No に変換し、走路センサの左右が正しく表示されていることが確認できます。

```
$ cpp test_hmi_format.py |python cleanfile.py
|python
*** voltage conversion ( 6.35 ) ***
<  6.4 V   >
*** voltage conversion ( 6.349 ) ***
<  6.3 V   >
*** voltage conversion ( -2.0  as a BAD data) ***
<  Unknown >
*** optosensors (0, 1, 1, 0, 1), cliff=0 ***
< WBBWB No >
*** optosensors (1, 1, 0, 0, 0), cliff =1 ***
< BBWWW Ok >
```

6.1.2 LED ハンドラ

4 種の LED（前照灯、後退灯、左折灯、右折灯）のデバイスハンドラをコーディング・検証します。他のデバイスハンドラでも同様の手順をとるので、少し詳しく説明します。手順とは、

1. 詳細な仕様を決める
2. インクルードファイルで必要な定義をする
3. プログラムを作成する
4. 模擬 GPIO でプログラムの動作を検証する
5. 上位の検証用スタブとして使えるようにする
6. GPIO を動かしてハードウエアを検証する

です。このうち、詳細仕様とは、前章で検討した外部仕様に加え、オブジェクトの設計上必要な属性と操作を定義したものです。私は習慣として、この内容をプログラムファイルの冒頭にコメントとして記述しています（最初にコメントだけのファイルを作ってからコーディングを始める)。それを見れば分かる内容は、この本では必要以上に説明していません。

インクルードファイル

LED を使用するのに必要なのは GPIO だけです。プログラムファイルの記述を単純にするためのマクロを定義しました。

led.h（抜粋）

```
/* led.h  LED 制御の定義
  初版：2022/2/17 Chuji --- gpio.h から独立
  最新版：
*/
#ifndef __LED
#define __LED
#include "use-pigpio.h"
#include "use-sys.h"
#define LED_ON  GPIO_HIGH
#define LED_OFF  GPIO_LOW
/* LED を操作する */
#ifndef SIMULATED_DRIVE
#define CREATE_LED(pi)  self.__lamps =
led_control(pi)
#define RIGHT_LED_ON()
self.__lamps.right_led(LED_ON)
#define RIGHT_LED_OFF()
self.__lamps.right_led(LED_OFF)
 (中略)
#else
#define CREATE_LED(pi)  pass
 (後略)
```

プログラムファイル

各 LED の点灯状態と点滅状態は内部属性として記憶しておきます操作の引数（オンまたはオフ）に従って属性を変更し、GPIO ポートに出力しています。500ms 毎に呼ばれる操作を使って、点滅をおこないます。

上位モジュールから使いやすくするため、後退灯と左右の赤色灯を別々の操作で点灯制御しています。大した手間ではないので、それぞれを記述しましたが、もっとスマートなやり方もあると思います。

led.py（抜粋）

```
/* led.py LED ランプ制御モジュール
  初版：  2020/9/4 Chuji
  最新版：2022/5/19 内部変数を外から見えなくした
          2021/1/24 --- 操作名変更
Class: led_control
属性：
  pi     PIGPIO オブジェクト
  front     前照灯の点灯状態
  back     後進灯の点灯状態
  right     右折灯の点灯状態
  left     左折灯の点灯状態
  right_blink   右折灯点滅フラグ
  left_blink   左折灯点滅フラグ
操作：
  beamer   前照灯を操作する
  backup_led   後進灯を操作する
  right_led   右折灯の点灯を操作する
  left_led   左折灯の点灯を操作する
  right_turn   右折灯の点滅を操作する
  left_turn   左折灯の点滅を操作する
  hazard   ハザードランプの点滅を操作する
  brake     ブレーキランプの点灯を操作する
  blink     0.5 秒毎に呼ばれて、点滅を制御する
 */
#include "include/use-pigpio.h"
#include "include/use-sys.h"
#include "include/led.h"
class led_control:
  def __init__(self, pi):
    self.__front = LED_OFF
    self.__back = LED_OFF
    self.__right = LED_OFF
    self.__right_blink = LED_OFF
    self.__left = LED_OFF
    self.__left_blink = LED_OFF
#ifdef HANDLER_STUB
    print(' LED lamps are initialized.', TO_ERROR)
#else
#ifndef BCM2835
    print('**** LED Lamps Initialization ***',
TO_ERROR)
#endif
    self.__pi = pi
    /* LED 駆動ポートを出力用に設定し、消灯状態にする */
    self.__pi.GPIO_MODE(GPIO_BEAMER, GPIO_OUT)
    self.__pi.GPIO_WRITE(GPIO_BEAMER, LED_OFF)
    self.__pi.GPIO_MODE(GPIO_BACK, GPIO_OUT)
    self.__pi.GPIO_WRITE(GPIO_BACK, LED_OFF)
    self.__pi.GPIO_MODE(GPIO_RIGHT, GPIO_OUT)
    self.__pi.GPIO_WRITE(GPIO_RIGHT, LED_OFF)
    self.__pi.GPIO_MODE(GPIO_LEFT, GPIO_OUT)
    self.__pi.GPIO_WRITE(GPIO_LEFT, LED_OFF)
 (中略)
  /* 右折灯の点灯制御  */
  def right_led(self, op):
    if (op == LED_ON) and (self.__right ==
LED_OFF):
      self.__right = LED_ON
```

```
#ifdef HANDLER_STUB
      print('Right Lamp is ON', TO_ERROR)
#else
      self.__pi.GPIO_WRITE(GPIO_RIGHT, LED_ON)
#endif
    elif (op == LED_OFF) and (self.__right ==
LED_ON):
      self.__right = LED_OFF
#ifdef HANDLER_STUB
      print('Right Lamp is OFF', TO_ERROR)
#else
      self.__pi.GPIO_WRITE(GPIO_RIGHT, LED_OFF)
#endif
 (中略)
  /* ハザード操作 */
  def hazard(self, op):
#ifdef HANDLER_STUB
    print('Hazard Lamp Operation!')
#endif
    self.right_turn(op)
    self.left_turn(op)
  /* 右折灯の点滅操作 */
  def right_turn(self, op):
    if (op == LED_ON) and (self.__right_blink ==
LED_OFF):
      self.__right_blink = LED_ON
#ifdef HANDLER_STUB
      print('Right Turn is ON', TO_ERROR)
#endif
      self.right_led(LED_ON)
    elif (op == LED_OFF) and (self.__right_blink
== LED_ON):
      self.__right_blink = LED_OFF
#ifdef HANDLER_STUB
      print('Right Turn is OFF', TO_ERROR)
#endif
      self.right_led(LED_OFF)
 (後略)
```

モジュール検証

検証スタブ test_led.py を、このモジュールの上位モジュール drive.py の代わりに使い、各 LED を順番に点灯、消灯、点滅させることで led.py を検証します。この検証は必ずしも Raspberry Pi で行う必要はなく、Linux の走っている PC 上でも行えます。もろん Raspberry Pi ZERO で検証しても構いません。

広範囲の検証項目をプログラム化しておくことで、バグが見つかりやすくなります。また、繰り返すことができるので、モジュールを変更したときや、ハードウエアの不具合があったときなどに、再現性のある検証ができます。この検証スタブだけは、以下に掲載しておきます。

test_led.py

```
/* test program for led.py on workstation
2020/10/4
*/
#include "include/use-time.h"
#include "include/use-pigpio.h"
#include "led.py"
#ifndef TIME_UNIT
#define TIME_UNIT  1
#endif
#ifndef INTERVAL
#define INTERVAL  3
#endif
#ifndef BLINK
#define BLINK    0.5
#endif
#ifndef BLINK_REPEAT
#define BLINK_REPEAT  5
#endif
```

```
pi = GPIO_OPEN()
led = led_control(pi)
print('*** Beamer Control --- ON ***')
led.beamer(LED_ON)
sleep(INTERVAL * TIME_UNIT)
print('*** Beamer Control --- OFF ***')
led.beamer(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Backlight Control --- ON ***')
led.backup_led(LED_ON)
sleep(INTERVAL * TIME_UNIT)
print('*** Backlight Control --- OFF ***')
led.backup_led(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Right Control --- ON ***')
led.right_led(LED_ON)
sleep(INTERVAL * TIME_UNIT)
print('*** Right Control --- OFF ***')
led.right_led(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Left Control --- ON ***')
led.left_led(LED_ON)
sleep(INTERVAL * TIME_UNIT)
print('*** Left Control --- OFF ***')
led.left_led(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Braking Control --- ON ***')
led.brake(LED_ON)
sleep(INTERVAL * TIME_UNIT)
print('*** Breaking Control --- OFF ***')
led.brake(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Right Turn Control --- ON ***')
led.right_turn(LED_ON)
sleep(BLINK * TIME_UNIT)
print('*** Blink ', BLINK_REPEAT, ' times ***')
for i in range(2 * BLINK_REPEAT - 1):
  led.blink()
  sleep(BLINK * TIME_UNIT)
led.right_turn(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Left Turn Control --- ON ***')
led.left_turn(LED_ON)
sleep(BLINK * TIME_UNIT)
print('*** Blink ', BLINK_REPEAT, ' times ***')
for i in range(2 * BLINK_REPEAT - 1):
  led.blink()
  sleep(BLINK * TIME_UNIT)
print('*** Left Turn --- OFF ***')
led.left_turn(LED_OFF)
sleep(BLINK * TIME_UNIT)
led.blink()
sleep(BLINK * TIME_UNIT)
led.blink()
sleep(INTERVAL * TIME_UNIT)
print('*** Hazard Control --- ON ***')
led.hazard(LED_ON)
sleep(BLINK * TIME_UNIT)
print('*** Blink ', BLINK_REPEAT, ' times ***')
for i in range(2 * BLINK_REPEAT - 1):
  led.blink()
  sleep(BLINK * TIME_UNIT)
led.hazard(LED_OFF)
sleep(INTERVAL * TIME_UNIT)
print('*** Breaking Control --- ON *** --- hazard
should stop')
led.brake(LED_ON)
sleep(BLINK * TIME_UNIT)
led.blink()
sleep(BLINK * TIME_UNIT)
led.blink()
sleep(INTERVAL * TIME_UNIT)
print('*** Breaking Control --- OFF ***')
led.brake(LED_OFF)
sleep(BLINK * TIME_UNIT)
led.blink()
sleep(BLINK * TIME_UNIT)
led.blink()
print('*** End of LED test procedure ***')
pi.GPIO_CLOSE()
```

最初に何も#define せずに実行すると、***で囲った検証項目のあとに、GPIO への設定と出力が表示されていきます。ポート番号や設定、出力が数値で表

されるので、設計と合っているか確認していきます。

```
$ cpp test_led.py |python cleanfile.py | python
**** LED Lamps Initialization ***
Mode at GPIO#  4  is set to:  1
Data:  0  is wrttten to GPIO#  4
Mode at GPIO#  17  is set to:  1
Data:  0  is wrttten to GPIO#  17
Mode at GPIO#  18  is set to:  1
Data:  0  is wrttten to GPIO#  18
Mode at GPIO#  27  is set to:  1
Data:  0  is wrttten to GPIO#  27
*** End of LED Initialization ***
*** Beamer Control --- ON ***
Data:  1  is wrttten to GPIO#  4
:
:
```

次に HANDLER_STUB を#define して実行すると、IO 操作は行わず、「どういう操作が行われたか」が表示されます。一種のスタブとして、上位モジュールの動作を検証するのに有効です。

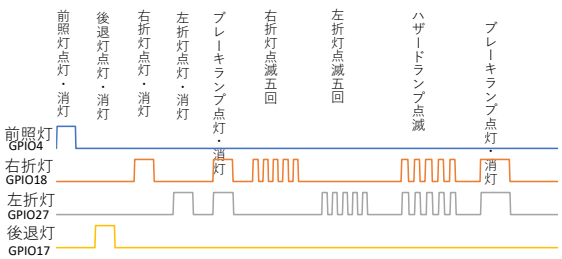
```
$ cpp -DHANDLER_STUB test_led.py |python
cleanfile.py | python
LED lamps are initialized.
*** Beamer Control --- ON ***
Beamer is ON
*** Beamer Control --- OFF ***
Beamer is OFF
:
:
```

ハードウエア検証

今度は Raspberry Pi ZERO 上で GPIO を動かしながら（BCM2835 を#define して）動作を確認します。PC でモジュールの検証を行ったときは、検証済のモジュールを Raspberry Pi ZERO に転送しておきます。コンソール画面上に検証項目が表示されるのを見ながら、実際の LED の点灯状況を確認します。

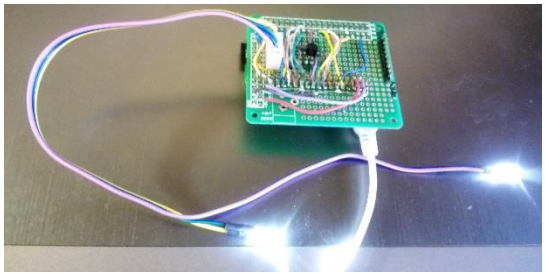
```
$ cpp -DBCM2835 test_led.py |python cleanfile.py |
python
```

最初に Raspberry Pi ZERO だけを使って、点灯信号を確認しました。この節の始めにやったシミュレーションによる検証では、GPIO 番号が表示されるだけでした。直感的に分かりづらかったので、GPIO ピンを直接観察します。cpp のオプションとして-DTIME_UNIT=0.01（時間設定の単位を 1 秒から 10ms に変更）してロジックアナライザで記録したものを、Excel で（データ量が多いので、ロジックアナライザでは表示しきれない）波形表示した結果を次に示します。この方が LED を制御している様子がよく分かりますね。



$ cpp -DBCM2835 -DTIME_UNIT=0.01 test_led.py | python

次は自作ハードウエアの評価です。私のハードウエアでは、前照灯を点灯させる回路と、モータードライバ回路へのインターフェースを Raspberry Pi 拡張基板に実装しています。まず拡張基板だけを接続してから、前照灯の動作を確認しました（検証プログラムの後半で他の LED を操作しても、何も起きません）。



前照灯の点灯操作を検証中

次にモータードライバ回路のハードウエア検証を行ってから、2.54mm ピッチユニバーサル基板を接続して、残りの LED の動作を確認するという手順を取りました。TIME_UNIT をデフォルト（1 秒）のままで実行すると、コンソールの表示を見ながらそれぞれの LED の点滅が確認できます。結果は掲載しませんが、評価プログラムどおりの動作でした。

ここまで検証できれば、このモジュール（デバイスハンドラ）がハードウエアを意のままに動かす操作を提供していることが確認できます。以下のデバイスハンドラの検証も、同じ流れで行えます。

6.1.3 モーターハンドラ

次にモーターハンドラのコーディングと検証を行います。回路が Raspberry Pi 拡張基板に搭載されているため、この基板の評価をしておくためと、そのあとで走行計モジュールの検証に使うからです。

インクルードファイル

インクルードファイルでは、どの GPIO を制御に使うかを最初に定義します。次にモーターの動作モード（前進、後退、右回頭、左回頭、停車、停止）と、モードごとの制御信号（H または L）の組み合わせをきめ、固定リスト（タプル）DRIVE_MODES

から取り出せるようにしました。回転数の制御は、ハードウエア PWM に渡してやります。

ハードウエア PWM の繰り返し周波数は 5kHz に設定しています。Raspberry Pi ZERO では 250MHz のクロックで PWM 信号を作っているので、PWM の分解能は 5 万分の 1（5kHz/250MHz）になります。pigpio ライブラリのインターフェースでは、PWM 設定値 0～100 万がデューティ 0～100%になっています。自律走行車では、回転速度 100%時の PWM 設定値が 900000（5V 電源で平均 4.5V）になるようにしました。左右のモーターの特性差を補償するため、回転速度を 100%以上にする可能性があるからです。

motor_drive.h（抜粋）

```
/* motor_drive.h DC モーター制御情報
  初版：2020/9/5 Chuji
  最新版：2021/1/24 --- 「ストップ」と「ブレーキ」の誤解釈を修正
*/
#ifndef __MOTOR_DRIVE
#define __MOTOR_DRIVE
/* モーター制御 GPIO ピンの定義 */
#define DRIVE_R1  GPIO_AIN1
#define DRIVE_R2  GPIO_AIN2
#define DRIVE_L1  GPIO_BIN1
#define DRIVE_L2  GPIO_BIN2
#define DRIVE_CONTROL  (DRIVE_R1, DRIVE_R2, DRIVE_L1, DRIVE_L2)
#define DRIVE_PARMS  4
#define MOTOR_R    GPIO_PWM_R
#define MOTOR_L    GPIO_PWM_L
/* モータードライバ TB6612FNG の制御情報 */
/* 制御信号 (right, left) = (AIN1, AIN2, BIN1, BIN2) */
/* 前進 */
#define DRIVE_FORWARD_POSITION  (GPIO_LOW, GPIO_HIGH, GPIO_HIGH, GPIO_LOW)
 （中略）
#define DRIVE_MODES  (DRIVE_FORWARD_POSITION, DRIVE_BACKWARD_POSITION, DRIVE_RIGHTROLL_POSITION, DRIVE_LEFTROLL_POSITION, DRIVE_BRAKE_POSITION, DRIVE_STOP_POSITION)
/* 上の配列の指標(mode) */
#define DR_FORWARD  0  /* 前進 */
#define DR_BACKWARD  1  /* 後退 */
#define DR_RIGHTROLL  2  /* 右回頭 */
#define DR_LEFTROLL  3  /* 左回頭 */
#define DR_BRAKE  4  /* 停車（パーキング） */
#define DR_STOP         5       /* 停止（ニュートラル） */
 （中略）
#define DRIVE_PWM_FREQ  5000  /* 5kHz */
#define DRIVE_PWM_FULL  1000000  /* 1M clock */
#define DRIVE_PWM_RANGE  9000  /* 出力 100%で 4.5V となるデューティ(Vcc=5V) */
#define DRIVE_DUTY(x)  int(DRIVE_PWM_RANGE * abs(x)) /* x in % (-100～100) */
 （中略）
/* ドライブの左右バランス係数 */
#define DRIVE_BALANCE  1.052  /* 実機にあわせて調整 */
 （後略）
```

DRIVE_BALANCE は、モーターや荷重のばらつきを補正するための係数です。最初は 1 にしておき、実際に走行テストをしながら、値を調整します。この値が 1 より大きくなる場合を考慮して、上のように最大回転速度（100%）時の PWM のデューティを 90%にしました。

プログラムファイル

モーターコントローラ AE-TB6612 を駆動するため、制御用信号とハードウエア PWM を発生させます。

motor.py

```
/* motor.py  モータードライブ制御
  初版：2020/9/9 Chuji
  最新版：2022/5/19 --- ローカル関数を見えなくした
          2021/1/24 --- ブレーキ操作を修正
Class motor_drive
属性
  pi     PIGPIO オブジェクト
  r_speed  右モーターの駆動速度設定値(%単位)
  l_speed  左モーターの駆動速度設定値
  mode     モーターの回転モード（前進、後退など）
  controls  制御 GPIO 信号リスト(AIN1, ...)
  drive_modes  運転モード毎の制御 GPIO 出力リスト(HIGH, LOW, ...)
操作
  control  ドライブ回路への制御出力を実行する（内部使用）
  (以下の操作は、すべて現在の回転モードを返す)
  set_speed  速度（%）を設定）する
  drive     モーターを駆動する
  brake     モーターを止めてブレーキをかける
  neutral  モーターをニュートラル状態にする
*/
#include "include/use-pigpio.h"
#include "include/use-sys.h"
#include "include/motor_drive.h"
class motor_drive:
  def __init__(self, pi):
    self.__pi = pi
    self.__r_speed = DRIVE_STOP
    self.__l_speed = DRIVE_STOP
    self.__controls = DRIVE_CONTROL
    self.__drive_modes = DRIVE_MODES
    self.__mode = DR_BRAKE
#ifdef HANDLER_STUB
    print('I/O ports for motor drive circuit is initialized.', TO_ERROR)
#else
    self.__pi.GPIO_MODE(DRIVE_R1, GPIO_OUT)
    self.__pi.GPIO_MODE(DRIVE_R2, GPIO_OUT)
    self.__pi.GPIO_MODE(DRIVE_L1, GPIO_OUT)
    self.__pi.GPIO_MODE(DRIVE_L2, GPIO_OUT)
#endif
    self.__control()
  /* モーターへの制御出力をコントローラに渡す……内部でのみ使用する */
  def __control(self):
#ifdef HANDLER_STUB
    print('Control data for motor drive circuit is applied: mode = ', self.__mode, TO_ERROR)
#else
    for i in range(DRIVE_PARMS):
      self.__pi.GPIO_WRITE(self.__controls[i], self.__drive_modes[self.__mode][i])
#endif
  /* モーターの回転速度(%) を設定し、停止状態を解除する */
  /* 現在のモーターの状態を返す */
  def set_speed(self, r_speed, l_speed):
    /* 回転方向の反転を指定なかったら、回転速度を設定する */
    if (self.__r_speed * r_speed >= 0)  and (self.__l_speed * l_speed >= 0):
      self.__r_speed = r_speed
      self.__l_speed = l_speed
      /* 回転速度は上下限の範囲内に限定する */
      if self.__r_speed > MAX_SPEED:
        self.__r_speed = MAX_SPEED
      elif self.__r_speed < MIN_SPEED:
        self.__r_speed = MIN_SPEED
      if self.__l_speed > MAX_SPEED:
```

```
        self.__l_speed = MAX_SPEED
      elif self.__l_speed < MIN_SPEED:
        self.__l_speed = MIN_SPEED
      /* 回転速度を PWM として渡す */
#ifdef HANDLER_STUB
      print('PWM data are transferred. Right: %d,
Left:%d' % (self.__r_speed, self.__l_speed *
DRIVE_BALANCE), TO_ERROR)
#else
      self.__pi.RIGHT_PWM(self.__r_speed *
DRIVE_BALANCE)
      self.__pi.LEFT_PWM(self.__l_speed)
#endif
    /* 回転方向を反転する指示を受けたら緊急停止する */
    else:
      self.brake()
    return(self.__mode)
  /* 設定に従ってモータの制御を行う */
  def drive(self):
    if (self.__r_speed == 0) and (self.__l_speed
== 0):
      self.__mode = DR_BRAKE
    elif IS_FORWARD(self.__r_speed):
      if IS_FORWARD(self.__l_speed):
        self.__mode = DR_FORWARD
      else:
        self.__mode = DR_LEFTROLL
    else:
      if IS_FORWARD(self.__l_speed):
        self.__mode = DR_RIGHTROLL
      else:
        self.__mode = DR_BACKWARD
#ifdef HANDLER_STUB
    print('Motor starts driving')
#endif
    self.__control()
    return(self.__mode)
  /* モーターの回転駆動を止めるが、車輪は自由に回る */
  def neutral(self):
    self.__mode = DR_STOP
    self.__r_speed = DRIVE_STOP
    self.__l_speed = DRIVE_STOP
#ifdef HANDLER_STUB
    print('Motor Neutral!! ')
#endif
    self.__control()
    return(self.__mode)
  /* モーターの回転を止め、ブレーキをかける */
  def brake(self):
    self.__mode = DR_BRAKE
    self.__r_speed = DRIVE_STOP
    self.__l_speed = DRIVE_STOP
#ifdef HANDLER_STUB
    print('Motor Brake!! ')
#endif
    self.__control()
    return(self.__mode)
```

内部で使う属性の説明をしておきます。GPIO のピン番号（controls）や、走行モード別の設定（drive_modes）は固定リストとして記憶しておき、必要なものを選んで使います。この設定は操作 control が行います。

速度設定操作 set_speed は、設定値の上下限をチェックしたうえで、ハードウエア PWM に渡します。モーターの回転方向を変えるような設定をしたときは、停車操作を行うようにしました。ただし上位モジュール（driver.py）でも同じチェックをしているので、この停車操作が実行されることはありません。

走行開始操作 drive、停車操作 brake、停止操作 neutral では、モードを決め、モード毎に決まっている制御信号を GPIO に送ります。set_speed 以降の操作は、すべて現在の動作モードを返すことにします。

モジュール検証

検証スタブ test_motor.py は、左右の動輪を意図どおりに回せるか、順番に検証します（プロジェクトファイルを参照）。

GPIO ポートとハードウエア PWM に対する設定が表示されます。動作モード毎の設定は、AIN1、AIN2、BIN1、BIN2 の順番に行われているので、意図どおりか確認します。

```
$ cpp test_motor.py |python cleanfile.py |python
Mode at GPIO#  26  is set to:  1
Mode at GPIO#  16  is set to:  1
Mode at GPIO#  21  is set to:  1
Mode at GPIO#  20  is set to:  1
Data:  1  is written to GPIO#  26
:
!!! Forward drive with 40\%
Hardware PWM at GPIO#  12  is set with frequency
5000  Hz and duty  360000
Hardware PWM at GPIO#  13  is set with frequency
5000  Hz and duty  360000
Data:  0  is written to GPIO#  26
Data:  1  is written to GPIO#  16
Data:  1  is written to GPIO#  21
Data:  0  is written to GPIO#  20
Motor drive mode is:  0
:
```

ハードウエア検証

ハードウエアの検証のため、BCM2835 を#define して test_motor.py を実行します。車台の下に箱などを置くか、車台をひっくり返して、車輪が宙に浮いている（無負荷で自由に回転できる）状態で検証します。

なおモータードライバユニットの電源は、2.54mm ピッチユニバーサル基板から供給する設計になっていますが、この段階では別の電源からとるようにしておきます。この検証が終わったら 2.54mm ピッチユニバーサル基板を接続し、そちらからモータードライバユニットの電源を取るようにしておきます。

```
$ cpp -DBCM2835 test_motor.py |python cleanfile.py
|python
:
```

動輪の回転方向を注意してみながら、モードの設定が正しく行われていることを確認します。回転方向などが間違っていたら、モータードライバ回路とモーターの結線を確認します。

次に、PWM への追従範囲を調べるため、デューティを徐々に上げていってみます。検証スタブ

test_motor_1.py（プロジェクトファイルを参照）を実行します。

0%に設定したら内部的には『停車状態』になってしまいます。0 以外の値（ここでは 10%）を設定して操作 drive を実行すると動輪を回転させようとします。以後は速度の設定だけで回転数が変わるはずです。

```
$ cpp -DBCM2835 test_motor_1.py |python cleanfile.py
|python
!!! Speed is set to  0 %
!!! Speed is set to  10 %
!!! Speed is set to  20 %
!!! Speed is set to  30 %
!!! Speed is set to  40 %
:
```

PWM デューティ 30%未満では無負荷でも回転しませんでした。30%では右車輪しか回転しません。実際の走行では 40%以上で行うことにし、motor_drive.h で指定します。

これでモータードライブの検証が済んだので、2.54mm ピッチのユニバーサル基板を接続し、LED ハンドラの残りの検証を完了させます。

6.1.4 走行計

走行計の読取りは PICCOLO チップが行ってくれるので、デバイスハンドラは PICCOLO チップとの通信が大部分です。

インクルードファイル

PICCOLO チップの再定義と、光インターラプタの駆動情報が大部分です。1 パルスあたりの移動距離・回頭角度も定義していますが、最終的には実験で値を調整します。

tripmeter.h

```
/* tripmeter.h 走行計に関する定義
   初版：2020/9/4 Chuji
   最新版：2021/7/22 --- PICCOLO チップ使用版に改造
*/
#ifndef __TRIPMETER
#define __TRIPMETER
#include "piccolo.h"
#define TRIP_ADDRESS   PICCOLO_ADDRESS
/* 走行計の動作モード */
#define TRIP_IDLE   0 /* 動作停止中 */
#define TRIP_DISTANCE   1 /* 距離計として動作中 */
#define TRIP_ROTATION   2 /* 回頭角度計として動作中 */
/* PICCOLO チップの使用方法定義 */
/* レジスタ定義 */
#define TRIP_COMMAND   PICCOLO_A_COMMAND
#define TRIP_RIGHT   PICCOLO_DATA0
#define TRIP_LEFT   PICCOLO_DATA1
/* 走行計動作制御 */
#define TRIP_ENABLE   PICCOLO_A_ENABLE
#define TRIP_DISABLE   PICCOLO_A_DISABLE
/* 1 パルス当たりの移動距離と回転角度
      --- 左右のエンコーダ計数値を足すので、2 で割ってある
*/
/*    理論値なので、実際の動作にあわせて調整すること */
#define GAIN_TRIP   2.5   /* 5mm/pulse */
/*
#define GAIN_ROLL   0.0406   */ /* 4.65°deg/pulse x
pi / 180 */
#define GAIN_ROLL   0.045
/* 距離・角度を指定したときの目標値補正量
    （「設定値を超えたら」停止するので、1 目盛分と停車遅れを補正する） */
#define TRIP_CORRECTION   2 * GAIN_TRIP
#define ROLL_CORRECTION   2 * GAIN_ROLL
/* シミュレーション/実走行用マクロ定義 */
#ifdef SIMULATED_DRIVE   /* シミュレーションでは走行計は使用しない */
#define CREATE_TRIP(x)   pass
#define ENABLE_TRIP(m)   pass
#define DISABLE_TRIP()   pass
#define CLEAR_TRIP()   pass
#define RETRIEVE_TRIP() 0.0
#else   /* 実走行では走行計を使用する */
#ifdef STEP_TUNING   /* チューニングのため、運転制御の上位から
          走行計を使えるようにする ---
          --- 運転制御と同時に使わないこと */
/* 運転制御用マクロ --- 速度計オブジェクトを隠さない */
#define CREATE_TRIP(x)   self.tripmeter =
tripmeter(x)
#define ENABLE_TRIP(m)   self.tripmeter.enable(m)
#define DISABLE_TRIP()   self.tripmeter.disable()
#define CLEAR_TRIP()   self.tripmeter.clear()
#define RETRIEVE_TRIP() self.tripmeter.retrieve()
/* 上位から使うためのマクロ */
#define ENABLE_TRIP_FOR_TUNING(m)
tripmeter.enable(m)
#define DISABLE_TRIP_FOR_TUNING()
tripmeter.disable()
#define CLEAR_TRIP_FOR_TUNING()   tripmeter.clear()
#define RETRIEVE_TRIP_FOR_TUNING()
tripmeter.retrieve()
#else
#define CREATE_TRIP(x)   self.__tripmeter =
tripmeter(x)
#define ENABLE_TRIP(m)   self.__tripmeter.enable(m)
#define DISABLE_TRIP()   self.__tripmeter.disable()
#define CLEAR_TRIP()   self.__tripmeter.clear()
#define RETRIEVE_TRIP()
self.__tripmeter.retrieve()
#endif   /* #ifdef STEP_TUNING の終わり */
#endif
#endif
```

プログラムファイル

初期化では、イベント計数機能を禁止します。PICCOLO チップにコマンドを送り、光インターラプタの電源を切ります。計数を開始するときは、逆の操作をします。計数値をクリアするには、PICCOLO チップの仕様に従い、動作禁止・計数開始を順番に行います。読み出し操作では、計数値を取り出し、左右の平均値に 1 パルス当たりの係数をかけてから返します。

tripmeter.py

```
/* tripmeter.py 回転センサによる移動距離/回頭角度の測定
   初版：2020/9/4 Chuji
   最新版：2022/5/19 --- ローカル変数を見えなくした
           2021/7/22 --- PICCOLO チップ用に改造
Class: tripmeter
属性：
  mode     走行計の動作状態
  pi     pigpio ライブラリ
  piccolo   PICCOLO チップへの通信チャンネル
操作：
  enable   走行計を動作させる
  disable   走行計を停止させる
```

```
  retrieve  パルス積算値を読み出す(距離/角度に換算する)
  clear     パルス積算値をクリアする
*/
#include "include/use-pigpio.h"
#include "include/use-sys.h"
#include "include/gpio.h"
#include "include/tripmeter.h"
#include "endian.py"
#ifdef HANDLER_STUB
#include "include/use-math.h"  /* 三角関数を計算するため */
#endif
class tripmeter:
  def __init__(self, pi):
#ifdef HANDLER_STUB  /* 走行計が呼ばれたことだけを表示する */
    print('*** Tripmeter Initialization ***', TO_ERROR)
#else
    self.__pi = pi
    self.__piccolo = self.__pi.OPEN_I2C(TRIP_ADDRESS)
    self.__pi.GPIO_MODE(GPIO_ACT_TRIP, GPIO_OUT)
#endif
    self.disable()
  /* 走行計の動作を開始させる */
  def enable(self, mode):
    if (mode == TRIP_DISTANCE) or (mode == TRIP_ROTATION):
      self.__mode = mode
#ifdef HANDLER_STUB
      print('*** Tripmeter is activated in mode #', mode, '***', TO_ERROR)
#else
      /* 光インターラプタの電源を投入する */
      self.__pi.GPIO_WRITE(GPIO_ACT_TRIP, GPIO_HIGH)
      /* PICCOLO チップのイベント計数を開始する */
      self.__pi.I2C_WRITE_BYTE(self.__piccolo, TRIP_COMMAND, TRIP_ENABLE)
#endif
  /* 走行計の動作作を停止させる */
  def disable(self):
    self.__mode = TRIP_IDLE
#ifdef HANDLER_STUB
    print('*** Tripmeter is deactivated ***', TO_ERROR)
#else
    /* PICCOLO チップのイベント計数を終了させる */
    self.__pi.I2C_WRITE_BYTE(self.__piccolo, TRIP_COMMAND, TRIP_DISABLE)
    /* 光インターラプタの電源を切る */
    self.__pi.GPIO_WRITE(GPIO_ACT_TRIP, GPIO_LOW)
#endif
  /* 走行計の積算値をクリアする */
  def clear(self):
#ifdef HANDLER_STUB
    print('*** Tripmeter is cleared ***', TO_ERROR)
#else
    self.__pi.I2C_WRITE_BYTE(self.__piccolo, TRIP_COMMAND, TRIP_DISABLE)
    self.__pi.I2C_WRITE_BYTE(self.__piccolo, TRIP_COMMAND, TRIP_ENABLE)
#endif
  /* Piccolo チップの積算値を読取り移動距離/回転角度として返す */
  def retrieve(self):
#ifdef HANDLER_STUB
    print('--- Reading Tripmeter right and then left', TO_ERROR)
    x = float(input('Tripmeter output (cm or deg) = '))
    if self.__mode == TRIP_DISTANCE:
      x *= 10  /* cm -> mm */
    elif self.__mode == TRIP_ROTATION:
      x *= math.pi/180  /* deg -> rad */
    else:  /* self.__mode == TRIP_IDLE */
      x = -1 /* 異常値 */
    return x
#else
    right = self.__pi.I2C_READ_WORD(self.__piccolo, TRIP_RIGHT)
    right = fix_endian(right)
    left  = self.__pi.I2C_READ_WORD(self.__piccolo, TRIP_LEFT)
    left  = fix_endian(left)
#ifdef DEBUG
    print('Right = ', right, 'Left = ', left)
#endif
    if self.__mode == TRIP_DISTANCE:
      dat = GAIN_TRIP * (right + left)
    elif self.__mode == TRIP_ROTATION:
      dat = GAIN_ROLL * (right + left)
    else:
      dat = 0.0
    return dat
#endif
```

モジュール検証

モジュールの検証用プログラム test_tripmeter.py（プロジェクトファイルを参照）では、走行計の各種モード設定を行い、走行距離測定結果と回頭角度測定結果を読み出します。

BCM2835 を#define せずに実行すると、初期化の様子と計数値を距離/角度に換算した結果を表示しました。

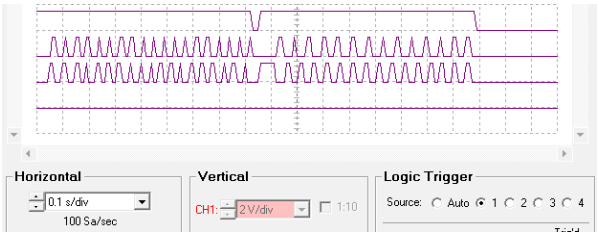
```
$ cpp test_tripmeterpy >tmp.py; python tmp.py
!!! Initialization
I2C communication with device address 0x38 is initiated
Mode at GPIO#  10  is set to:  1
A byte data 0x 0 is written to register # 3 of device at 0x38
Data:  0  is written to GPIO#  10
:
!!! Retrieve distance
Read a word data from Register #8 of device at 0x38 ? 200
Read a word data from Register #10 of device at 0x38 ? 300
distance =  5.0
:
```

次に HANDLER_STUB を#define して実行すると、検証プログラムの意図が表示され、距離計が出力するデータを cm または度（°）で聞かれます。

ハードウエア検証

ハードウエアの検証には BCM2835 を#define して test_tripmeter.py を実行します。実際に数秒間だけ動輪を回して、光インターラプタからの信号から測定値が得られることを確認します。DEBUG を #define すると、PICCOLO チップの計数値を表示してくれます。駆動時間（DRIVE_TIME）に比例して計数値が増加すること、左右の計数値の差は 2～3%程度であることが分かりました。この差はモーターのばらつきが主な要因で、実際に走行させると車輪の寸法や摩擦も影響してきます。最終的には直進させてモーターハンドラの補正係数 DRIVE_BALANCE を決めます。

もし可能なら、ロジックアナライザを信号 R-SPEED と L-SPEED につないで、発生したパルスを観測してみてください。次の図では、一連の動作をロジックアナライザの画面に収めるため、DRIVE_TIME=0.8、 INTERVAL=0.03 に設定しています。信号は上からフォトインタラプタの駆動信号、右側と左側のセンサ出力です。



走行計の駆動信号と光センサの出力

6.1.5 電源監視

電源監視は、A/D コンバータ MCP3425 の設定を行い、測定値を（分圧回路の入力）電圧に変換します

インクルードファイル

インクルードファイルでは制御レジスタのビットパターン（実際には使っていません）と設定データ、読み取りデータを電圧（単位は V）に変換する手順を定義しています。

adc.h

```
/* adc.h I2C インターフェース A/D コンバータ MCP34245 制御情報
   初版：2020/9/6 Chuji
   最新版：2021/1/24 --- i2c_read_device の誤使用訂正 */
#ifndef __MCP3425
#define __MCP3425
#include "use-pigpio.h"
/* 測定値のステータス */
#define ADC_GOOD   GPIO_OFF
#define ADC_BAD    GPIO_ON
/* I2C アドレス */
#define ADC_I2C_ADDR  0x68 /* b'1101xxx', x は特に指定がない時 0 */
/* 内部レジスタ */
/* MCP3425 との通信ではレジスタを指定しない。I2C アドレスだけでアクセスする */
/* 制御レジスタのビットパターン */
#define ADC_NO_UPDATE(x)  (x & 0x80) >> 7
#define ADC_CHANNEL(x)  (x & 0x60) >> 5 /* channel は実際には使用されていない */
#define ADC_MODE(x)  (x & 0x10) >> 4 /* 1:連続測定, 0:ワンショット */
#define ADC_SAMPLE(x)  (x & 0x0c) >> 2 /* 0:240sps, 1:60sps, 2:15sps */
#define ADC_GAIN(x)  (x & 0x03) /* 0: x 1, 2: x 2, 3: x 4, 3: x 8 */
/* 制御レジスタへの設定内容 */
#define ADC_CONFIG  0x18
/* 連続測定モード、毎秒 15 回測定（16 ビット分解能）、内部増幅率=1 */
/* 読み取りデータ：(測定値上位バイト) ー (測定値下位バイト) ー (制御レジスタ) */
#define ADC_READ_LEN  3
/* 測定値計算 (16 ビット分解能の場合) */
#define ADC_STATUS(x)  x[2]
#define ADC_OUT(x)  float((x[0] << 8) + x[1])
#define ADC_DATA_BAD(x)
(ADC_NO_UPDATE(ADC_STATUS(x)) != 0)
#define ADC_MAX    0x7FFF /* = 32767 (signed 16bit) */
#define ADC_Vref  2.048 /* internal reference voltage */
#define ADC_RESOLUTION  62.5e-6 /* = ADC_Vref / ADC_MAX */
/* アナログ回路の分圧 */
#define ADC_DIV    6.2/(6.2 + 27) /* = 0.18675 (分圧比) */
#define ADC_FACTOR  334.7e-6 /* = ADC_RESOLUTION / ADC_DIV */
#define ADC_VOLTAGE(x)  ADC_FACTOR * ADC_OUT(x)
/* 低電圧警報 */
#define BATTERY_NORMAL    9.0 /* 正常時の電池電圧 */
#define BATTERY_BAD    -2.0 /* ありえない電圧 */
#define BATTERY_BAD_DATA(x)  (x < 0)
#define BATTERY_LOW_LIMIT  6.0 /* 最低電池電圧 */
#define BATTERY_LOW(x)    (x < BATTERY_LOW_LIMIT)
#define BATTERY_OK    0  /* 以上回数カウンタ初期値 */
#define BATTERY_BAD_COUNT  3  /* 連続して電圧が低かったらシャットダウンを要求する回数 */
#endif
```

プログラムファイル

A/D コンバータのドライバ adc.py の基本部分は、初期設定と読み取りデータを電圧に変換することです。前回の読み取り以来、測定値が更新されていないときは、ありえない値を返すことで上位に知らせます。

HANDLER_STUB が#define されているときは処理内容を表示するだけです。

マクロ ADC_TEST は ADC から転送されてくるバイト列をそのまま表示するためのもので、デバイスのデータブックを読み間違えていないことを確認するために用意しました。開発初期以外には使用していません。

adc.py

```
/* adc.py  A/D コンバータ MCP34245 による電池電圧の測定
   初版：2020/9/7 Chuji
   最新版：2022/5/19 --- ローカル変数を見えなくした
Class voltage-meter:
属性：
  pi  PIGPIO オブジェクト
  i2c  I2C 通信ハンドル
  active  I2C 通信が使用可能かを示すフラグ
  output  電圧測定値（値とステータスからなる）
操作：
  read  電池電圧を読み取る
    （戻り値は電圧とデータステータスのタプル）
  close  I2C 通信を終了する  （これ以後は read 不可）
*/
#include "include/use-pigpio.h"
#include "include/use-sys.h"
#include "include/adc.h"
class voltage_meter:
  def __init__(self, pi):
#ifndef HANDLER_STUB
#ifndef BCM2835
    print('*** A/D Converter Initialization ***', TO_ERROR)
#endif
#endif
    self.__pi = pi
    self.__active = True
#ifdef HANDLER_STUB
```

```
    print('I2C channel to ADC is opened ',
TO_ERROR)
#else
    /* 通信チャンネルと ADC の初期設定 */
    self.__i2c = self.__pi.OPEN_I2C(ADC_I2C_ADDR)
    self.__pi.I2C_WRITE_DEVICE(self.__i2c,
ADC_CONFIG)
    /* いちどシグマデルタ回路を動かしておく */
    self.read()
#ifndef BCM2835
    print('*** A/D Converter Initialization
finished ***')
#endif
#endif
  def read(self):
    if self.__active == True:
#ifdef HANDLER_STUB
      output = float(input('ADC voltage = '))
#else
      dat_len, dat =
self.__pi.I2C_READ_DEVICE_BLOCK(self.__i2c,
ADC_READ_LEN)
#ifdef ADC_TEST
      print('A/D C output = 0x%2x, 0x%2x, 0x%2x' %
(dat[0], dat[1], dat[2]))
#endif
      /* 電圧の計算 */
      if dat_len == ADC_READ_LEN:
        output = ADC_VOLTAGE(dat)
        if ADC_DATA_BAD(dat):
          output = BATTERY_BAD
      else:
        output = BATTERY_BAD
    else:
      output = BATTERY_BAD
#endif
    return output
  def close(self):
#ifdef HANDLER_STUB
    print('I2C channel to ADC is closed')
#else
    self.__pi.CLOSE_I2C(self.__i2c)
    self.__active = False
#ifndef BCM2835
    print('*** I2C communication with A/D
converter is closed ***')
#endif
#endif
```

<u>モジュール検証</u>

検証スタブ test_adc.py（プロジェクトファイルを参照）は、A/D コンバータの測定値とステータス（0 が正常）を表示します。

まず何も#define せずに実行すると、A/D コンバータを読み取る代わりに、データ入力を求めてきます。適当な 16 進数（0x を付けても、付けなくてもよい）3 バイトをコンマで区切って与えれば、電圧とステータスに変換して表示します。1 バイト目がデータの上位バイト、2 バイト目が下位バイト、3 バイト目がステータスです。上位バイトの MSB は 0（正符号）にしてください。ステータスの MSB が 1 のときは「測定値が更新できていない」というフラグが立ちます。

```
$cpp test_adc.py |python cleanfile.py >tmp.py;
python tmp.py
*** A/D Converter Initialization ***
I2C communication with device address 0x68 is
initiated
A byte data 0x18 is written to device at 0x68
*** A/D Converter Initialization finished ***
Read 3 bytes from device at 0x68 ? 0x20, 0x34, 0x68
data from ADC:  2.7592668000000002 V with flag:  0
Read 3 bytes from device at 0x68 ? 0x70, 0x20, 0x86
data from ADC:  9.6072288 V with flag:  1
Read 3 bytes from device at 0x68 ?
:
```

次に HANDLER_STUB を#define して実行すると、測定値を V 単位で与えることができます。

```
$ cpp -DHANDLER_STUB test_adc.py >tmp.py; python
tmp.py
I2C channel to ADC is opened
ADC voltage = 6
ADC status (Good = 0, Bad = 1)0
data from ADC:  6.0 V with flag:  0
ADC voltage = 9
ADC status (Good = 0, Bad = 1)1
data from ADC:  9.0 V with flag:  1
:
```

<u>ハードウエア検証</u>

次に Raspberry Pi ZERO 上で BCM2835 を#define してから実行します。電圧測定値が表示されます。

```
$ cpp -DBCM2835 test_adc.py |python cleanfile.py |
pytho
n
data from ADC:  8.8568314 V with flag:  0
data from ADC:  8.8836074 V with flag:  0
data from ADC:  8.8836074 V with flag:  0
:
```

ハードウエア・ジャンパーを別の電圧源に接続すれば、その電圧を表示することを確認できます。結果は『電子回路の設計』の節に載せてあります。

ΔΣ方式の問題なのか、それとも python のバイトコンパイルのせいなのか分かりませんが、設定を書き込んだ直後の一回だけは、測定値が少し（約 0.4%）低めになるときがあります。

6.1.6 走路センサ、崖センサ

走路センサと崖センサは同じ回路（感度だけ異なる）なので、ひとつのファイルにまとめました。

<u>インクルードファイル</u>

インクルードファイルでは、使用する GPIO ポートとレベル、測定結果の表示文字の定義をしています。それに加え、走路センサのポートを右端から並べたリストを定義しています。

sensor.h

```
/* sensor.h 光学式センサの定義（バギー用）
   初版：2020/9/9 Chuji
   最新版：2020/9/13 ... シミュレーション時に使う定義を追
加
*/
#ifndef __OPTICAL_SENSOR
#define __OPTICAL_SENSOR
#ifdef SIMULATED_DRIVE
```

```
#include "use-math.h"
#else
#include "use-pigpio.h"
#endif
/* 崖センサ用の定義 */
#define CLIFF_SENSOR  GPIO_CLIFF
/* 入力信号レベル */
#define CLIFF_SAFE  GPIO_HIGH
#define CLIFF_DANGER  GPIO_LOW
/* 走路センサ用の定義 */
#define LINE_NUM_SENSORS  5
#define LINE_MSB  0b10000
#define Get_LSB(x)  (x & 1)
#define Get_MSB(x)  (x & LINE_MSB)
/* センサの GPIO ポートを左端から処理する。右端が最下位になるようにするため */
#define LINE_SENSORS  (GPIO_LT5, GPIO_LT4, GPIO_LT3, GPIO_LT2, GPIO_LT1)
/* 測定制御 */
#define SENSOR_CONTROL  GPIO_ACT_OPTO
#define SENSOR_ACTIVE  GPIO_HIGH
#define SENSOR_INACTIVE  GPIO_LOW
/* 走路センサの出力ビットエンコーディング */
#define LINE_WHITE  0
#define LINE_BLACK  1
/* 走路センサの検出データ */
#define LINE_OUT  {GPIO_LOW:LINE_BLACK, GPIO_HIGH:LINE_WHITE}
/* LCD 上の表示イメージ(走路センサ) */
#ifndef LINE_SYMBOL_WHITE
#define LINE_SYMBOL_WHITE  'W'
#endif
#ifndef LINE_SYMBOL_BLACK
#define LINE_SYMBOL_BLACK  'B'
#endif
/* ダミー用の走路センサ出力（実際には使用しない） */
#define LINE_DEFAULT_DATA  0b00100
/* LCD 上の表示イメージ（崖センサ） */
#define CLIFF_SYMBOL_SAFE  ' Ok'
#define CLIFF_SYMBOL_DANGER  ' No'
/* 光センサデータの可用性 */
#define LINE_GOOD  GPIO_ON
#define LINE_BAD  GPIO_OFF
/* シミュレーション時のセンサ出力 */
/* #define CASE_PARALLEL  (0.01 * math.pi) */ /* ほぼ指定の方向の範囲 */
/* #define COURSE_WIDTH  20 /* 走路マーカーの幅（の 1/2） */
#define SENSOR_GAP  10.16 /* センサの間隔  2.54mm x 4 ピッチ */
#define SENSOR_ALL_ON  0b00011111 /* センサは全て走路を検出している */
#define SENSOR_ALL_OFF  0b00000000 /* センサは全て走路を検出していない */
#define SENSOR0_ON  0b00000001 /* センサ#0（右端）が走路を検出している */
#define SENSOR1_ON  0b00000010 /* センサ#1 が走路を検出している */
#define SENSOR2_ON  0b00000100 /* センサ#2 が走路を検出している */
#define SENSOR3_ON  0b00001000 /* センサ#3 が走路を検出している */
#define SENSOR4_ON  0b00010000 /* センサ#4（左端）が走路を検出している */
#endif
```

プログラムファイル

センサの入力を読み取る前に、赤外 LED を光らせ（activate）、読み取ったら LED の発光を止める（deactivate）ことで、省電力化をはかっています。走路センサの読み取りでは、右端のセンサから順番に読み出して、結果をビット列（右端が最上位）として返します。

optosensors.py

```
/* optosensors.py 反射光式センサのドライバ
  初版：2020/9/11 Chuji
  最新版：2022/9/13 --- シミュレーション時のラインセンサ出力を追加
          2022/5/19 --- ローカル変数を見えなくした
Class: reflectometer
属性：
  pi    PIGPIO オブジェクト
  lines    ライセンサの入力ポートのリスト
操作：
  activate  ライセンサの動作を開始する（赤外灯を点灯させる:内部使用）
  deactivate  ライセンサの動作を停止する（赤外灯を消灯させる:内部使用）
  read_line  ライセンサの入力（右端が最下位のビット列）を読み取る
  read_cliff  クリフセンサの入力を読み取る
*/
#include "include/use-pigpio.h"
#include "include/use-sys.h"
#include "include/sensor.h"
#ifdef SIMULATED_DRIVE
#include "simulated_track.py"
#endif
class reflectometer:
  def __init__(self, pi):
    self.__pi = pi
    self.__lines = LINE_SENSORS
#ifdef HANDLER_STUB  /* ドライバーが呼ばれたことだけを示す */
    print('Optical sensors are initialized', TO_ERROR)
#else
    /* センサ駆動ポートの初期化 */
    self.__pi.GPIO_MODE(SENSOR_CONTROL, GPIO_OUT)
    self.__pi.GPIO_WRITE(SENSOR_CONTROL, SENSOR_INACTIVE)
    /* 走路センサポートの初期化 */
    for i in range(LINE_NUM_SENSORS):
      self.__pi.GPIO_MODE(self.__lines[i], GPIO_IN)
    /* 崖センサポートの初期化 */
    self.__pi.GPIO_MODE(CLIFF_SENSOR, GPIO_IN)
#endif
  /* 反射光式センサの動作開始 */
  def activate(self):
#ifdef HANDLER_STUB
    print('Line sensors are activated', TO_ERROR)
#else
    self.__pi.GPIO_WRITE(SENSOR_CONTROL, SENSOR_ACTIVE)
#endif
  /* 反射光式センサの動作停止 */
  def deactivate(self):
#ifdef HANDLER_STUB
    print('Line sensors are deactivated', TO_ERROR)
#else
    self.__pi.GPIO_WRITE(SENSOR_CONTROL, SENSOR_INACTIVE)
#endif
  /* 反射光センサの読み取り */
#ifndef SIMULATED_DRIVE
  def read_line(self):
    s = 0
    self.activate()

    /* 左端のセンサから調べ、右端が最下位になるようにする */
    for i in range(LINE_NUM_SENSORS):
#ifdef HANDLER_STUB
#ifdef GIVE_LINE  /* センサ入力をコンソールから与える */
      x = input('LINE ', i, '-th sensor? (Black = 1):')
      l = int(x)
#else      /* 適当な入力を与える */
      l = LINE_BLACK
#endif
#else      /* センサ入力を読み取る（黒=1、白=0） */
      l = LINE_OUT[self.__pi.GPIO_READ(self.__lines[i])]
```

```
#endif
      /* ビット列にエンコーディングする */
      s = (s << 1) + l
    self.deactivate()
    return s
#else  /* シミュレーション座標を使ったセンサ出力の計算結果
を返す */
#define  read_line(x, y, t)  track_sensor(x, y, t)
/* (x, y, t) 車両の座標と向き */
#endif
/* 崖センサを読み取る */
  def read_cliff(self):
    self.activate()
#ifdef HANDLER_STUB
#ifdef GIVE_CLIFF
    s = int(input('Cliff sensor (Safe = 0, Danger
= 1) ?'))
#else
    s = CLIFF_SAFE
#endif
#else
    s = self.__pi.GPIO_READ(CLIFF_SENSOR)
#endif
    self.deactivate()
/* NO_CLIFF を#define していたら、常に”安全”を返す。
   処理時間を計測するときのために、IO 動作は行うようにして
いる */
#ifdef NO_CLIFF
    s = CLIFF_SAFE
#endif
    return s
```

モジュール検証

評価スタブ test_opto.py（プロジェクトファイルを参照）では、赤外 LED のオン・オフを直接制御し、次に各センサの入力を読み取ります。

何も#define せずに実行すると、ポートへの設定と出力が表示されます。次に入力が要求されるので、ポートの状態を与えると、結果が表示されます。

キーボード入力を必要とするので、cpp の出力をいったん tmp.py に入れ、それを実行しています。

```
$ cpp test_opto.py |python cleanfile.py >tmp.py ;
python tmp.py
Mode at GPIO#  5  is set to:  1
Data:  0  is written to GPIO#  5
Mode at GPIO#  9  is set to:  0
:
Read line sensors... including
Activation/Deactivation
Data:  1  is written to GPIO#  5
GPIO# 9 input? 0
GPIO# 25 input? 1
GPIO# 11 input? 1
GPIO# 8 input? 0
GPIO# 7 input? 0
Data:  0  is written to GPIO#  5
[0, 1, 1, 0, 0]
:
```

HANDLER_STUB を#define して実行すると、IO は行われず、すべてのセンサ入力として 1 を返してきます。

ハードウエア検証

test_opto.py でもハードウエアを動かすことができますが、走路の状態を変えながら結果を調べたいので、専用スタブ test_opto1.py（プロジェクトファイルを参照）を用意しました。反射光センサの入力をまとめて表示し続けます。止めるときはキーボードの Ctrl-C を押します。

```
$ cpp -DBCM2835 test_opto1.py |python
WWWWW Ok
WWWWW Ok
WWWWW No
WBBBB No
:
```

6.1.7 障害物センサ

超音波センサ HC-SR04 を駆動して、障害物への距離とコードを返します。

インクルードファイル

インクルードファイルでは、送信パルスの極性とパルス長（20μs）、PICCOLO チップ 1 カウントあたりの障害物までの距離、距離コードのしきい値と LCD に表示するテキストを定義しています。

us_sensor.h

```
/* us_sensor.h 超音波式距離センサの制御情報
  初版：2020/9/11 Chuji
  最新版：
*/
#ifndef __US_SENSOR
#define __US_SENSOR
#include "use-pigpio.h"
#include "piccolo.h"
/* GPIO ポートの定義 */
#define US_TRIGGER  GPIO_US_TRIG
/* 超音波トリガのパルス幅と極性 */
#define US_PULSE_WIDTH  20 /* microsecond */
#define US_PULSE_POLARITY  GPIO_HIGH
/* PICCOLO チップの設定 */
#define PIC_TC_SET  PICCOLO_WIDTH_DEFAULT  /* パル
ス幅計測、同期モード */
        /* 時間分解能 8us */
#define PIC_DIST_UNIT  1.36  /* 1 クロックの距離
（8us/2 x 340m/s）単位は mm */
/* 距離コードのしきい値と異常距離 */
#define DIST_MARGINAL    500  /* 500mm */
#define DIST_WARNING    250  /* 250mm */
#define DIST_ALERT    100  /* 100mm */
#define DIST_DEFAULT    1200  /* 1.2m */
#define DIST_TOO_FAR    10000  /* 10m */
#define DIST_UNKNOWN    0  /* 距離不明 */
/* 設定値の辞書 --- タプルの順番は最初が「遠い」側 */
/* 距離コードを判定するしきい値 */
#define US_DELAYS  (DIST_MARGINAL, DIST_WARNING,
DIST_ALERT, DIST_ALERT)
/* 距離コード */
#define US_NOT_SURE  0
#define US_CLEAR  1
#define US_MARGINAL  2
#define US_WARNING  3
#define US_ALERT  4
#define US_DANGER  4
#define US_NUM_CODES  5
/* 距離に関する演算 */
#define IS_NOT_NEAREST(x)  (x < US_FACING)
#define IS_NOT_FARTHEST(x)  (x > US_CLEAR)
```

```
#define NEARER(x)  (x + 1)
#define FARTHER(x)  (x - 1)
/* 測定結果を示すテキスト */
#define TEXT_NOT_SURE  ' Error! '
#define TEXT_CLEAR  ' Clear  '
#define TEXT_MARGINAL  ' Far    '
#define TEXT_WARNING  ' Near   '
#define TEXT_ALERT  ' Danger!'
/* テキストの辞書 */
#define US_TEXTS  (TEXT_NOT_SURE, TEXT_CLEAR,
TEXT_MARGINAL, TEXT_WARNING, TEXT_ALERT)
#endif
```

プログラムファイル

初期化と超音波送信についての詳しい説明は不要でしょう。

距離は PICCOLO チップのカウント値に係数を掛けて求めます。ステータスを使った例外処理として、計数値が更新されていないときは『距離不明』に、オーバーフローがあったときは『非常に遠距離（障害物はない)』にします。後は測定した距離に応じて、距離コードを返します（上位プログラムでは、距離コードしか使いません)。

us_sensor.py

```
/* us_sensor.py 超音波式障害物検知センサー制御
  初版：2020.9.11 Chuji
  最新版：2022/5/19 --- ローカル変数を見えなくした
          2021/7/14 PICCOLO チップ対応版に改造
Class: obstacle_sensor
属性：
  pi     PIGPIO オブジェクト
  piccolo  PICCOLO チップとの通信チャンネル
操作：
  trigger  超音波パルスを送信する
  distance  前回測定した障害物までの距離を mm 単位の数値と
距離コードで返す
*/
#include "include/use-sys.h"
#include "include/us_sensor.h"
#include "endian.py"
class obstacle_sensor:
  def __init__(self, pi):
    self.__pi = pi
#ifdef HANDLER_STUB
    print('US obstacle sensor is initialized',
TO_ERROR)
#else
    /* 送信ポートの初期化 */
    self.__pi.GPIO_MODE(US_TRIGGER, GPIO_OUT)
    self.__pi.GPIO_WRITE(US_TRIGGER, GPIO_LOW)
    /* PICCOLO チップの初期化 */
    self.__piccolo =
self.__pi.OPEN_I2C(PICCOLO_ADDRESS)
    self.__pi.I2C_WRITE_BYTE(self.__piccolo,
PICCOLO_T_COMMAND, PIC_TC_SET)
#endif
    /* 距離を表すテキスト */
    self.__texts = US_TEXTS
    /* 距離測定を開始する */
    self.trigger()
  /* 超音波を発信する */
  def trigger(self):
#ifdef HANDLER_STUB
    print('US pulse is generated', TO_ERROR)
#else
    self.__pi.GENERATE_PULSE(US_TRIGGER,
US_PULSE_WIDTH, US_PULSE_POLARITY)
#endif
  /* 超音波の受信状態確認 --- 戻り値は mm 単位の距離データ
と区間コード */
  def distance(self):
    dist = DIST_UNKNOWN
    pos = US_NOT_SURE
#ifdef HANDLER_STUB
    print('US detection:', TO_ERROR)
    dist = float(input('Distance to obstacle
(mm)'))
#else
    try:
      status =
self.__pi.I2C_READ_BYTE(self.__piccolo,
PICCOLO_T_STATUS)
    except pigpio.error as e:
      print ("error: %s"%(e))
    if PICCOLO_WIDTH_READY(status): /* 測定値が更新
されているとき */
      count =
self.__pi.I2C_READ_WORD(self.__piccolo,
PICCOLO_WIDTH)
      count = fix_endian(count)
#ifdef DEBUG
      print('US pulse count = ', count)
#endif
      if not PICCOLO_WIDTH_OVERFLOW(status):
        /* オーバーフローなし */
        dist = PIC_DIST_UNIT * count
      else:
        dist = DIST_TOO_FAR
    /* 測定値が更新されていないとき: dist =
DIST_UNKNOWN */
#endif
    if (dist != DIST_UNKNOWN):
      if (dist >= DIST_MARGINAL):
        pos = US_CLEAR
      elif (dist >= DIST_WARNING):
        pos = US_MARGINAL
      elif (dist >= DIST_ALERT):
        pos = US_WARNING
      else:
        pos = US_ALERT
    return dist, pos
  /* 距離コードをテキストに変換する */
  def string(self, pos):
    return self.__texts[pos]
```

モジュール検証

検証用スタブ test_us.py（プロジェクトファイルを参照）は、1 秒ごとに（TRIGGER が#define されているときは）超音波を送信し、伝搬時間の測定結果を読み取って表示します。

HANDLER_STUB が#define されているときは、測定値を mm 単位で与えることができます。

何も#define せずに実行すると、PICCOLO チップのステータス（レジスタ#0）と計数値（レジスタ#4）を読み取ろうとするので、手動でデータを与えてやります。16 ビットデータはバイト逆順で与える必要があるので、ちょっと面倒ですね。与えるデータと結果を次の表に示します。

ステータス		計数値		検証結果	
OVT	RDP	(距離)	入力値	距離表示	区間表示
0	1	10cm	0x4900	99.28	Danger!
		20cm	0x9300	199.92	Near
		50cm	0x6f01	499.12	Far
		1.5m	0x4f04	1500.08	Clear
1	1	使わない	任意の値	10000	Clear
x	0	読まない	-	0	Error!

障害物センサの検証データ

ハードウエア検証

ハードウエアの検証には TRIGGER と BCM2835 を #define して test_us.py を実行します。センサから反射面（壁）までの距離を巻尺で測り、表示と比較すると、『電子回路の設計』の節にあるような直線になりました。ハードウエアの検証はひとまず終わりです。小型ユニバーサル基板が組付けられていたら、HMI 下位モジュール、それにキースイッチと LCD の検証を済ませておきましょう。

6.2　運転制御

運転制御機能は、ここまでのデバイスハンドラを使い、自律走行プログラムの指示に従って運転を行います。

インクルードファイル

少し趣味的ですが、操作量を船舶用語の「全速前進」とか「面舵いっぱい」といった表現でも設定できるようにしました。日本語と英語の表現はプロジェクトファイルを見てください。

運転制御では、多くの情報（状態）によって運転のしかたが変わります。別々の変数にすると分かりにくいので、次に示すような 1 ビットごとのステータスとして扱うことにしました。

警報の有無	障害物の有無	崖検知の有無	電源電圧
走行/停車中	前後/回頭	走行目標の有無	障害回避中

運転制御で使うステータス

唐突ですが、ここで『ハンドルの効き』を決めておきます。ハンドルをいっぱいに切ったとき（操舵量 100%）に外側の車輪の描く円の半径を『最小回転半径』といいます。国産小型乗用車の場合 4～6m ですが、これをトレッド幅で割ると 3～4 でした。付録の『仮想走行』で説明するように、この値は $(R+d)/2d$ で表現できます。これに内輪と外輪の速度 $V-\Delta v$、$V+\Delta v$ を代入すると、

$$\Delta v/V = 1 / (2 \times (R+d)/2d - 1)$$

となり、その値は 0.15～0.2 です。つまりハンドルをいっぱいに切ったとき、走行速度 V の 15～20%だけ外輪を加速、内輪を減速すれば、乗用車と同じような感覚で運転できることが分かりました。実際の自動車では、ホイールベースやタイヤの最大角度なども影響しますが、今回の構造ならトレッド幅だけで充分近似できていると思います。

driver.h（抜粋）

```
/* driver.h 走行制御用の定義
   初版：2020/9/14 Chuji
   最新版：2022/4/14  シミュレーション用パラメータを追加
*/
#ifndef __DRIVE_CAR
#define __DRIVE_CAR
#include "motor_drive.h"
#include "ap_manager.h"
#include "tripmeter.h"
/* 自動走行プログラムのコールバック関数に返すコード */
#define REACHED_GOAL     0  /* 走行目標に達して停車した
*/
#define STOPPED_BY_BATTERY  1   /* 電池電圧が低下した
ので停車した */
#define STOPPED_BY_ALARM  2  /* 安全警報が発生したので
停車した */
#define ALARM_CLEARED     3  /* 安全警報が解除された
*/
#define DRIVER_RELEASED  4  /* 走行機能の制御ができな
くなった */
#define DRIVER_AVOIDING  5  /* 障害物回避中 --- コー
ルバックでは使わない */
/* 操作要求の結果 */
#define DRIVE_REQ_ACCEPTED  'Ok'
#define DRIVE_REQ_ERROR     'Error'
（中略）
/* 運転ステータスをビットエンコードする
      7    6    5    4    3    2    1    0
   +----+----+----+----+----+----+----+----+
   | Rp | Dv | Rl | Tp | Av | BL | US | CL |
   +----+----+----+----+----+----+----+----+
     Rp: 1:警報発報中        0:発報中の警報なし
     Dv: 1:走行中            0:停車中
     Rl: 1:回頭走行          0:前進・後退走行
     Tp: 1:走行目標値あり    0:走行目標値なし
     Av: 1:障害回避中        0:障害回避なし
     BL: 1:電池電圧低下      0:電池電圧正常
     US: 1:前方障害あり      0:前方障害なし
     CL: 1:崖を検知          0:崖を検知せず
*/
（後略）
```

プログラムファイル

このプログラムは長いので掲載していません。プロジェクトファイルを参照してください。簡単に説明しておきます。

仮想走行するときは、モーター、LED それに距離計のハンドラは組み込まず、simulation()で 100ms 毎の位置と向きを計算で求めます。

最初に、運転する自律走行プログラムを acquire()と release()で決定します。次に speed_control()と wheel_control()で速度と操舵量を指定します。移動目標がある時は、set_trip()で目標値を設定し、定期

的に check_trip()を読み出して、目標に到達したら停車します。走行するときは drive()を、回頭するときは roll()を呼び出します。stop()を呼び出すと停車します。alarm()で安全警報を確認し、必要に応じて停車とハザード点滅を行います。定期で呼び出される control()は、警報と距離計の確認を行い、方向指示器の点滅を制御します。

```
driver.py
```
（プロジェクトファイルを参照）

モジュール検証

運転制御モジュールは機能が多いので、次のような機能グループごとに検証することにします。この機能は GPIO を使わずに（BCM2835 を#define せずに）検証できます。その代わり、デバイスハンドラをスタブで置き換え（HANDLER_STUB を#define し）て、煩雑な GPIO 操作は表示させません。

1. 自律走行プログラムの排他制御
2. 単純走行
3. 走行計を使った走行
4. 安全機能
5. シミュレーション走行

検証プログラム test_driver_xxx.py は内部で HANDLER_STUB を#define して、どういう意図で各デバイスハンドラを呼び出したかということだけを表示させています。検証プログラムの意図と、デバイスハンドラの呼び出し表示は、すべて標準エラー出力に出しています。これは、シミュレーション走行時に走行データ（標準出力）だけを取り出すための用意です。

1. 自律走行プログラムの排他制御の検証

検証シナリオは以下の 4 項目です。AP は自律走行プログラムを示しています。

- 存在しない AP が acquire しようとする→失敗する
- 4 種類の AP が acquire 後に release する→それぞれの AP で、acquire も release も成功する
- ある AP が acquire した後、他の AP が acquire しようとする→元の AP は release したという通知を受け、新しい AP が acquire する
- ある AP が acquire した後、他の AP が release しようとする→失敗する

検証プログラムを実行すると、設計どおりの排他制御ができていることが分かります。

```
test_driver_user.py
```
（プロジェクトファイルを参照）

```
$ cpp test_driver_user.py >tmp.py; python tmp.py
 LED lamps are initialized.
*** Tripmeter Initialization ***
I/O ports for motor drive circuit is initialized.
Control data for motor drive circuit is applied:
mode =  4
##### Invalid controller ID tries to acquire #####
### ACQUIRE result =  Error
:
```

2. 単純走行の検証

前進と後退、速度の変更、ハンドル操作、停止、回頭などの単純な走行を検証します。検証のシナリオは以下のとおりです。

- 停車状態からの前進、加速、減速、停車
- 前進中にハンドルを切る。右折・左折灯を確認
- 前二項を後退で行う
- 左右の回頭。右折・左折灯の確認
- 前照灯の点灯と消灯

検証スタブはシナリオどおりにコーディングします。

```
test_driver_simple.py
```
（プロジェクトファイルを参照）

シナリオどおりの動作になっているか確認します。特に、モーターのモード、LED の点灯状況に注意してください。ちなみに停車操作を行うと、PWM の設定が二回呼ばれます。これは、速度と操舵量を別個に設定しているせいですが、実害はありません。

```
$ cpp test_driver_simple.py >tmp.py; python tmp.py
 LED lamps are initialized.
:
### ACQUIRE result =  Ok

##### Drive forward #####
### set speed slow-ahead (40%)
PWM data are transferred. Right: 40, Left:40
PWM data are transferred. Right: 40, Left:40
:
Motor starts driving
Control data for motor drive circuit is applied:
mode =  0
:
```

3. 走行距離計を使った走行の検証

走行距離あるいは回転角度を指定して走行・回頭させ、指定走行量に達した時に停車することを確認します。検証のシナリオは以下のとおりです。

- 距離を 30cm に設定して走行する。距離計の距離が設定値未満なら継続、以上なら停車する

- 角度を 180° に設定して回頭する。距離計の距離が設定値から換算した距離未満なら継続、以上なら停車する

走行計の測定値を与えるため、cpp の処理結果をいったんファイルにしまってから、そのファイルを実行します。走行計の測定値を聞かれるので、目標値以下なら走行を継続、目標値以上なら停車して報告（コード 0（目標到達））が上がってくることを検証します。

test_driver_targeted.py
（プロジェクトファイルを参照）

検証プログラムを実行すると、走行計の出力を聞いてくるので、cm あるいは° で与えてやります。目標値までは動作を繰り返し、目標値に達すれば（越えれば）停車することが確認できます。

```
$ cpp test_driver_targeted.py >tmp.py ; python
tmp.py
LED lamps are initialized.
*** Tripmeter Initialization ***
:
### ACQUIRE result =  Ok

##### Drive 30cm forward #####
:
*** Tripmeter is activated in mode # 1 ***
*** Tripmeter is cleared ***
:
Motor starts driving
Control data for motor drive circuit is applied:
mode =  0
--- Reading Tripmeter right and then left
Tripmeter output (cm or deg)  = 20     （走行距離を cm 単位で与える）
:
```

4. 安全機能の検証

危険な速度変更を許さないこと、安全警報を発生させると停車すること、警報を解除するまで走行しないこと、回避命令を与えると警報があっても走行すること、回避中に警報が解除されると通常走行に戻ることを確認します。検証のシナリオは以下のとおりです。

- 前進を始め、警報がないときは継続する
- 停車せずに後退しようとすると失敗する
- 警報が発生すると緊急停車する
- 回避運転を始めると後退ができる
- 警報が解除されるとハザードも消える
- 表示灯の点滅を確認する

test_driver_safety.py
（プロジェクトファイルを参照）

検証プログラムを実行すると、前進から停車せずに後退ができないこと、アラームなしの状態から安全警報が発生すると停車して発車できないこと、安全警報が解除されると走行が再開できることを確認します。回避命令を実行すると、安全警報が発生中でも走行できること、安全警報が解除されると通常走行状態になることを確認できます。最後に点滅が 5 回のコール（0.5 秒）ごとに行われることが確認できれば終了です。

```
$ cpp test_driver_safety.py |python
 :
##### Try to acquire driver #####
 :
##### Drive forward #####
 :

##### Alarm handling #####
## No alarms ##

## Cliff alarm ##
Motor Brake!!
 :
Hazard Lamp Operation!
Right Turn is ON
Left Turn is ON
### Report from driver.py:  2

Try to start driving while alarm is on --- should
fail
Result =  Error
 :
```

これで運転制御機能の基本的なところは検証できました。最後に一連の運転操作を行い、時々刻々の車両位置と向きをシミュレーションできることを検証します。この機能は、自律走行プログラムの検証に使います。

5. シミュレーション走行の検証

SIMULATED_DRIVE を#define して実行すれば、デバイスハンドラの代わりにシミュレーションモデルが呼び出されます。この検証は、単純走行で行えば十分です。シナリオは 2 本です。

- 単純走行：前方（Y 軸方向）に進み、右にハンドルを切って障害物を回避し、元のコースに戻る。一旦停止後、後退してから左にハンドルを切って円を描く
- 目標付き走行：前方に 300mm 進んだら、一旦停止し右へ 90° 回頭する。さらに 300mm 前進したら、一旦停止し左へ 90° 回頭する。今度は後退と回頭を繰り返して元に戻る

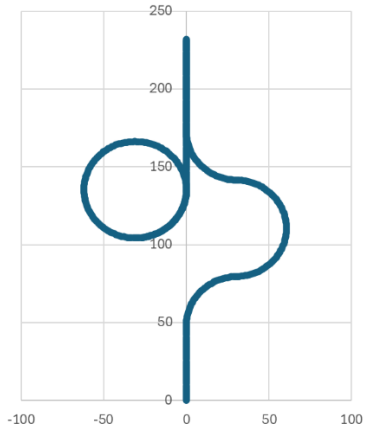
最初のシナリオをコーディングした結果を示します。

test_driver_simulation.py
（プロジェクトファイルを参照）

運転変更の前には'****'で始まるメッセージを標準エラー出力に表示します。シミュレーション結果は標準出力に表示されるので、下のようにリダイレクトすることができます。

```
$ cpp test_driver_simulation.py |python >tmp.csv
**** half-S shape drive ***
0 0
**** Reverse ***
-40.0 0
**** Circle ***
### Report from driver.py:  4
$
```

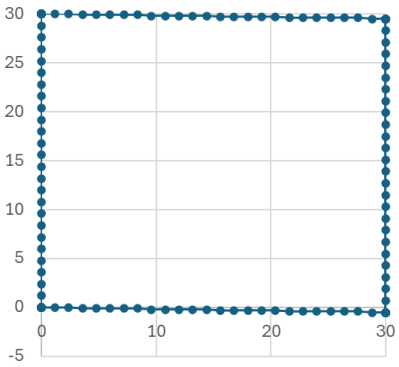
リダイレクトしたファイルを Excel で散布図にすると、八分音符のような「軌跡」が得られます。



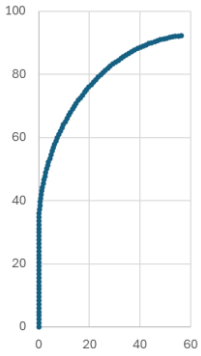
次のシナリオでは、ちょっと「細工」をしました。include/driver.h で回頭速度を実際の 1/10 にして、100ms の制御周期の間に回頭する角度を小さくなっています。走行計の分解能不足を補おうとしました。

test_driver_simulation1.py
（プロジェクトファイルを参照）

最初のシナリオと同じように「軌跡」を描かせると、次のような四角が現れてきました。回頭時の角度が「90 度を超えた」ことを停止条件にしているので、少し回りすぎています。



もうひとつ、仮想走行を試してみましょう。30cm 前進したら、ハンドルを右に切って 90 度方向を変えるまで進むテストが次の test_drive_simulation2.py です。トリップメーターを回頭角度測定用に設定し、ハンドルを切って走行した結果を右の図に示します。



test_driver_simulation2.py
（プロジェクトファイルを参照）

6.3　自律走行プログラム

ここから 4 種類の自律走行プログラムとそれらを管理する ap_manager を設計、検証していきます。検証ではコマンドを手で与え、実機での動作は、安全のため行いません（BCM2835 を#define しない）。

インクルードファイル

まず自律走行プログラムに共通する定義をインクルードファイルにまとめておきます。

drive_program.h

```
/* drive_program.h 自律走行プログラム共通定義
  初版：2020/9/17 Chuji
  最新版：
*/
#ifndef __DRIVE_PROGRAMS
#define __DRIVE_PROGRAMS
/* 自律走行プログラムのオブジェクト名 */
#define WEB_DRIVE_OBJECT  web_driver
#define BLOCK_DRIVE_OBJECT  block_driver
#define SCRATCH_DRIVE_OBJECT  scratch_driver
#define LINE_TRACE_OBJECT  line_tracer
/* 自律走行プログラムの識別名 */
#define NO_CONTROLLER           'Null' /* 自律走行プログラムの制御を受けていない */
#define WEB_CONTROLLER          'Web' /* Web drive */
#define BLOCK_CONTROLLER        'Blk' /* Block drive */
#define SCRATCH_CONTROLLER      'Scr' /* Scatch drive */
#define TRACE_CONTROLLER        'Lin' /* Line Tracing */
#define CONTROLLERS     (WEB_CONTROLLER, BLOCK_CONTROLLER, SCRATCH_CONTROLLER, TRACE_CONTROLLER)
/* プログラム状態 */
#define PROGRAM_INACTIVE  'Inactive'
#define PROGRAM_RUNNING    'Running'
#define PROGRAM_PAUSING         'Pausing'
#define PROGRAM_NO_ALARM        'No_Alarm'
#define PROGRAM_ALARM           'Alarm'
#define PROGRAM_LOW_BATTERY  'Battery_Low'
#define PROGRAM_AVOIDING        'Avoiding'
/* 認識している運転制御機能の状態 */
#define PROGRAM_IDLE    'Idle'  /* 自プログラムの制御を受けていない */
#define PROGRAM_PARKING         'Parking'
#define PROGRAM_DRIVING    'Driving'
#define PROGRAM_ROLLING    'Rolling'
#define PROGRAM_ROLLING_RIGHT  'RollRight'
#define PROGRAM_ROLLING_LEFT  'RollLeft'
```

```
#define PROGRAM_PARK_TO_AVOID  'ParkToAvoid'
/* AP Manager と自律走行プログラムへのコマンド構造 */
/* Python */
#define Cmd(x)  x[0]
#define OpCode(x)  x[1:]
/* JavaScript ... str.CodeField などとして使う */
#define CodeField  substr(0,1)
#define ParmField  substr(1)
/* レスポンスコード */
#define Activated  'A'
#define Deactivated  'D'
#define Activate(x)  Activated + x
#define Deactivate(x)  Deactivated + x
  /* Line Tace 専用 */
#define Reached    'R'
#define Lost    'L'
#define Reach(x)  Reached + x
#define GetLost(x)  Lost + x
/* Redis FIFO に速度、操舵量、アラームを送信する */
/* Web に通知するアラーム辞書 --- 不要になった
#define ALARM_MESSAGES
{ALARM_CLEARED:PROGRAM_NO_ALARM,
STOPPED_BY_ALARM:PROGRAM_ALARM,
STOPPED_BY_BATTERY:PROGRAM_LOW_BATTERY,
DRIVER_AVOIDING:PROGRAM_AVOIDING}
*/
#ifdef AP_STUB
#define SHOW_SPEED(fifo, speed)  print('Speed is
set to: ', speed)
#define SHOW_WHEEL(fifo, wheel)  print('Wheel is
set to: ', wheel)
#define ISSUE_ALARM(x)  print('Alarm <', x, '> is
issued.', TO_ERROR)
#define CLEAR_ALARM(x) print('Alarm is cleared.',
TO_ERROR)
#else
#define SHOW_SPEED(fifo, speed)
fifo.rpush(REDIS_TO_SPEED, str(speed))
#define SHOW_WHEEL(fifo, wheel)
fifo.rpush(REDIS_TO_WHEEL, str(wheel))
#define ISSUE_ALARM(x)
self.fifo.rpush(REDIS_TO_ALARM, x)
#define CLEAR_ALARM(x)  ISSUE_ALARM(x)
#endif
#endif
```

6.3.1 自律走行プログラム共通部

自律走行プログラム群の親オブジェクトを作成します。機能の大部分はプログラムの起動・休止・再開・終了に加え、運転制御からのコールバック関数からなります。それ以外の操作は、「何もしない」ようになっています。各自律走行プログラムでの操作が必要になる時には、上書きします。

drive_program.py

```
/* drive_program.py  自律走行プログラム（共通部）オブジェクト
  初版：2022/4/11 Chuji
  最新版： 2024/3/13 --- 共通上位オブジェクトとして再定義
Class Drive_program
属性
  myID    自律走行プログラムの ID
  state    自律走行プログラムの実行状態
  drive_state   運転状態
  drive_before  停車前の運転状態
  avoiding  障害回避中フラグ
  speed         走行速度
  wheel         ハンドル操作量
  driver  運転制御機能オブジェクト
  fifo    Redis FIFO（HMI と Web ブラウザへの報告用）
  alarm    警報の発生状況
操作
  start    自律走行を始める
  pause    自律走行を中断する
  resume  自律走行を再開する
  terminate  自律走行を終了する
  update  Web 画面の速度と操舵量表示を更新する
  command  自律走行プログラムに命令を与える
  control  自律走行プログラムの定周期実行を行う
  stopped  自動運転停止時のコールバック関数
  my_alarm  警報処理を行う
*/
/* 何回インクルードしても取り込まれるのは一度だけ */
#ifndef __DRIVE_PROGRAM_OBJECT
#define __DRIVE_PROGRAM_OBJECT
#include "include/drive_program.h"
#include "include/hmi.h"
#include "driver.py"
class Drive_program:
  def __init__(self, myID, sensor, driver, fifo):
    self.myID = myID
    /* 走路センサ(sensor)はライントレースでしか使わない
*/
    self.line_sensor = sensor
    self.driver = driver
    self.fifo = fifo
    self.state = PROGRAM_INACTIVE
    self.drive_state = PROGRAM_IDLE
    self.init()
  /* 内部変数の初期化 */
  def init(self):
    self.drive_before = PROGRAM_PARKING
    self.avoiding = False
    self.speed = STOP_ENGINE
    self.wheel = MID_SHIP
    self.alarm = PROGRAM_NO_ALARM
  /* 自律走行を始める*/
  def start(self):
#ifndef AP_STUB
    if self.state == PROGRAM_INACTIVE:
      self.state = PROGRAM_RUNNING
      self.driver.acquire(self.myID, self.stopped)
      self.init() /* 内部変数の初期化 */
      self.drive_state = PROGRAM_PARKING
      self.driver.stop(self.myID)
#else
    print('Automonous Program ' +  self.myID + '
started and now has Driver control.', TO_ERROR)
#endif
#ifdef DEBUG_AP
    print('Automonous Program ' +  self.myID + '
started and now has Driver control.', TO_ERROR)
#endif
  /* 自律走行を中断する */
  def pause(self):
#ifndef AP_STUB
    if self.state == PROGRAM_RUNNING:
      self.state = PROGRAM_PAUSING
      self.drive_before = self.drive_state
      self.drive_state = PROGRAM_PARKING
      self.driver.stop(self.myID)
#else
      print('Automonous Program ' +  self.myID + '
is pausing .', TO_ERROR)
#endif
  /* 自律走行を再開する */
  def resume(self):
#ifndef AP_STUB
    if self.state == PROGRAM_PAUSING:
      self.state =  PROGRAM_RUNNING
      if self.drive_before ==  PROGRAM_DRIVING:
        self.drive_state = PROGRAM_DRIVING
        self.driver.speed_control(self.myID,
self.speed)
        self.driver. wheel_control(self.myID,
self.wheel)
        self.driver.drive(self.myID)
      elif self.drive_before ==
PROGRAM_ROLLING_RIGHT:
        self.driver.roll(self.myID, ROLL_RIGHT)
        self.drive_state = PROGRAM_ROLLING_RIGHT
      elif self.drive_before ==
PROGRAM_ROLLING_LEFT:
        self.driver.roll(self.myID, ROLL_LEFT)
        self.drive_state = PROGRAM_ROLLING_LEFT
      self.drive_before = PROGRAM_PARKING
#else
      print('Automonous Program ' +  self.myID + '
has resumed.', TO_ERROR)
#endif
  /* 自律走行を終了する */
```

```
  def terminate(self):
#ifndef AP_STUB
    if self.state != PROGRAM_INACTIVE:
      self.state = PROGRAM_INACTIVE
      self.drive_state = PROGRAM_PARKING
      self.drive_before = PROGRAM_PARKING
      self.speed = STOP_ENGINE
      self.wheel = MID_SHIP
      self.alarm = PROGRAM_NO_ALARM
      /* 既に Release されていたら、これ以降の処理は失敗する */
      self.driver.stop(self.myID)
      self.driver.release(self.myID)
#else
      print('Automonous Program ' +  self.myID + ' has terminated.', TO_ERROR)
#endif
  /* Web 画面の速度と操舵量表示を更新する */
  def update(self):
    self.driver.update()
  /* 自動運転停止時のコールバック関数 --- 共通部分のみ */
  def stopped(self, cause):
#ifndef AP_STUB
    self.drive_state = PROGRAM_PARKING
    if cause == DRIVER_RELEASED:
      self.terminate()
    elif cause == STOPPED_BY_ALARM:
      self.alarm = PROGRAM_ALARM
      self.my_alarm()
    elif cause == STOPPED_BY_BATTERY:
      self.alarm = PROGRAM_LOW_BATTERY
      self.my_alarm()
    elif cause == ALARM_CLEARED:
      self.alarm = PROGRAM_NO_ALARM
      self.my_alarm()
#else
     print('Automonous Program ' +  self.myID + ' received message that the driver has stopped by cause of ', cause, TO_ERROR)
#endif
  /* 自律走行を終了する --- デフォルトは何もしない、または表示のみ */
  def control(self):
#ifndef AP_STUB
    print('drive program.control is called ...', TO_ERROR)
#else
    pass
#endif
  /* 警報処理を行う --- デフォルトは何もしない、または表示のみ */
  def my_alarm(self):
#ifndef AP_STUB
    print('alarm method is called ... current alarm: ', self.alarm, TO_ERROR)
#else
    pass
#endif
  /* 自律走行プログラムに命令を与える --- デフォルトは何もしない、または表示のみ */
  def command(self, cmd):
#ifndef AP_STUB
    print('Command method is called with: ', cmd, TO_ERROR)
#else
    pass
#endif
#endif
```

このオブジェクトを検証するためのスタブも用意しました。これから開発する自律走行プログラムの検証を練習するためです。

test_drive_program.py

（プロジェクトファイルを参照）

```
$ cpp test_drive_program.py >tmp.py; python tmp.py
>>> Driver is initialized
:
### starting an AP
>>> Driver is acquired by  Web
>>> Driver stops and parks

### setting speed
Command method is called with:  Set Speed
:
```

自律走行プログラムには、それぞれの走行に合わせた Web ページを用意しますが、説明はこの本の後半まで待ってください。

6.3.2 Web ドライブ

インクルードファイル

Web ドライブが使う定義と、それに対応する Web ページの定義をしておきます。このファイルは Web ページの html/JavaScript と共通して使えるようにしてあります。

web_drive.h

```
/* web_drive.h Web drive 用の定義
  初版：2020/9/17 Chuji
  最新版：
*/
#ifndef __WEB_DRIVE_DEFINITIONS
#define __WEB_DRIVE_DEFINITIONS
#include "drive_program.h"
#include "driver.h"
/* ページタイトル */
#define WEB_DRIVE_TITLE  Web 走行（矢印で加減速と方向制御ができます）
/* アラーム発生状態 */
#define WD_NO_ALARM  PROGRAM_NO_ALARM
#define WD_ALARM  PROGRAM_ALARM
#define WD_BATTERY  PROGRAM_LOW_BATTERY
#define WD_AVOIDING  PROGRAM_AVOIDING
/* コマンド取得関数 */
#define WD_COMMAND(x)  Cmd(x)  /* x[0] */
/* Web Drive が処理するコマンド */
#define WD_GO_FORWARD  'F'
#define WD_GO_BACKWARD  'B'
#define WD_STEER_RIGHT  'R'
#define WD_STEER_LEFT  'L'
#define WD_RIGHT_ROLL  'G'
#define WD_LEFT_ROLL  'T'
#define WD_MODE_TOGGLE  'M'  /* Line Trace 用 */
#define WD_LINE_COLOR  'C'  /* Line Trace 用 */
#define WD_STOP    'S'
#define WD_DISARM  'D'
#define WD_EXIT    'E'
/* Line Trace で自動走行中には拒否するコマンド */
#define PROHIBITED_COMMANDS  (WD_GO_FORWARD, WD_GO_BACKWARD, WD_STEER_RIGHT, WD_STEER_LEFT, WD_RIGHT_ROLL, WD_LEFT_ROLL)
/* 速度と操舵量 */
#define WD_STOPPING  STOP_ENGINE
#define WD_STRAIGHT  MID_SHIP
#define WD_SPEED_STEP  4
#define WD_INC_FORWARD  SLOW_AHEAD/WD_SPEED_STEP
#define WD_INC_BACKWARD  SLOW_ASTERN/WD_SPEED_STEP
#define WD_MAX_FORWARD  FULL_AHEAD
#define WD_MIN_FORWARD  SLOW_AHEAD
#define WD_MAX_BACKWARD  FULL_ASTERN
#define WD_MIN_BACKWARD  SLOW_ASTERN
#define WD_STEER_STEP  10
#define WD_INC_STEER  STARBOARD_5/WD_STEER_STEP
#define WD_MAX_STEER_RIGHT  HARD_STARBOARD
#define WD_MAX_STEER_LEFT  HARD_PORT
/* Web ページへの表示 */
#ifdef AP_STUB
#define WD_SET_SPEED(x) pass /* driver は使わない */
#define WD_SET_WHEEL(x) pass
#define WD_SHOW_SPEED(x)        print('Speed = ', x)
```

```
#define WD_SHOW_WHEEL(x)        print('Wheel = ',
x)
#define WD_ROLL(x)              print('Roll to ',
x, '(0 = Right, 1 = Left)')
#define WD_SHOW_ROLLING(x)      print('Rolling
button receives: ', x)
#define WD_SHOW_COLOR(x)        print('Color
button receives: ' x)
#define WD_SHOW_MODE(x)         print('Mode button
receives: ', x)
#define WD_SHOW_ALARM(x)        print('Alarm = ',
x)
#else
#define WD_SET_SPEED(x)
self.driver.speed_control(self.myID, x)
#define WD_SET_WHEEL(x)
self.driver.wheel_control(self.myID, x)
#define WD_SHOW_SPEED(x)  SHOW_SPEED(self.fifo, x)
#define WD_SHOW_WHEEL(x)  SHOW_WHEEL(self.fifo, x)
#define WD_ROLL(x)    self.driver.roll(self.myID,
x)
#define WD_SHOW_ROLLING(x)
self.fifo.rpush(REDIS_TO_RESPONSE, x)
#define WD_SHOW_COLOR(x)
self.fifo.rpush(REDIS_TO_RESPONSE, x)
#define WD_SHOW_MODE(x)
self.fifo.rpush(REDIS_TO_RESPONSE, x)
#define WD_SHOW_ALARM(x)  SHOW_ALARM(self.fifo, x)
#endif
#endif
```

プログラムファイル

親オブジェクトを継承する形で Web ドライブの走行プログラムを作ります。ボタン操作による加減速と回頭、それに安全警報の処理を行うだけなので、比較的単純な機能です。

web_drive.py

```
/* web_drive.py  Web Drive 自律走行プログラム
  初版：2022/6/2 Chuji
  最新版：2024/3/13 --- 共通オブジェクトを継承するように
変更
Class web_driver
属性（継承した属性は含まない）：

操作（継承した操作は含まない）：
  command  自律走行プログラムに命令を与える
  my_alarm  アラーム処理
*/
#ifndef __WEB_DRIVE_OBJECT
#define __WEB_DRIVE_OBJECT
#include "include/web_drive.h"
#include "drive_program.py"
class web_driver(Drive_program):
  def __init__(self, ID, sensor, driver, fifo):
    super().__init__(ID, sensor, driver, fifo)
    /* この自律走行プログラム固有の属性定義と初期化 */
    self.should_drive = False
#ifdef STEP_TUNING      /* チューニング用に走行計を使う
*/

self.driver.ENABLE_TRIP_FOR_TUNING(TRIP_DISTANCE)
    self.driver.CLEAR_TRIP_FOR_TUNING()
#endif
  /* 目的到着のコールバックは起こらないので、継承どおり */
  /* アラームの処理 */
  def my_alarm(self):
    if self.alarm == PROGRAM_NO_ALARM:
      CLEAR_ALARM(self.alarm)
    else:
      ISSUE_ALARM(self.alarm)
   /* 運転プログラムにコマンドを与える */
  def command(self, cmd):
#ifdef AP_STUB
    print('Web Drive/Line Trace received command:
', cmd, TO_ERROR)

#else
    if self.state != PROGRAM_INACTIVE:
      /* 緊急停止ボタン */
      if WD_COMMAND(cmd) == WD_STOP:
        self.driver.stop(self.myID)
        self.speed = WD_STOPPING
        if self.drive_state ==
PROGRAM_ROLLING_LEFT:
          self.wheel = WD_STRAIGHT
          WD_SET_WHEEL(self.wheel)

WD_SHOW_ROLLING(Deactivate(WD_LEFT_ROLL))
        elif self.drive_state ==
PROGRAM_ROLLING_RIGHT:
          self.wheel = WD_STRAIGHT
          WD_SET_WHEEL(self.wheel)

WD_SHOW_ROLLING(Deactivate(WD_RIGHT_ROLL))
        self.drive_state = PROGRAM_PARKING
      /* 前方向きボタン */
      elif WD_COMMAND(cmd) == WD_GO_FORWARD:
        if self.drive_state in (PROGRAM_IDLE,
PROGRAM_PARKING):
          self.speed = WD_MIN_FORWARD
          self.drive_state = PROGRAM_DRIVING
          /* 停車しているので走行を始める */
          self.should_drive = True
        elif self.drive_state == PROGRAM_DRIVING:
          if self.speed > WD_STOPPING:
            self.speed += WD_INC_FORWARD
            if self.speed > WD_MAX_FORWARD:
              self.speed = WD_MAX_FORWARD
          else:  /* i.e., self.speed < WD_STOPPING
*/
            self.speed -= WD_INC_BACKWARD
            if self.speed > WD_MIN_BACKWARD:
              self.speed = WD_STOPPING
              self.drive_state = PROGRAM_PARKING
              self.driver.stop(self.myID)
        if self.drive_state == PROGRAM_DRIVING:
          if (self.alarm != WD_ALARM) and
(self.alarm != WD_BATTERY):
            WD_SET_SPEED(self.speed)
            if self.should_drive == True:
              self.driver.drive(self.myID)
              self.should_drive = False
      /* 後方向きボタン */
      elif WD_COMMAND(cmd) == WD_GO_BACKWARD:
        if self.drive_state in (PROGRAM_IDLE,
PROGRAM_PARKING):
          self.speed = WD_MIN_BACKWARD
          self.drive_state = PROGRAM_DRIVING
          /* 停車しているので走行を始める */
          self.should_drive = True
        elif self.drive_state == PROGRAM_DRIVING:
          if self.speed > WD_STOPPING:
            self.speed -= WD_INC_FORWARD
            if self.speed < WD_MIN_FORWARD:
              self.speed = WD_STOPPING
              self.drive_state = PROGRAM_PARKING
              self.driver.stop(self.myID)
          else:  /* i.e., self.speed < WD_STOPPING
*/
            self.speed += WD_INC_BACKWARD
            if self.speed < WD_MAX_BACKWARD:
              self.speed = WD_MAX_BACKWARD
        if self.drive_state == PROGRAM_DRIVING:
          if (self.alarm != WD_ALARM) and
(self.alarm != WD_BATTERY):
            WD_SET_SPEED(self.speed)
            if self.should_drive == True:
              self.driver.drive(self.myID)
              self.should_drive = False

      /* 右向きボタン */
      elif WD_COMMAND(cmd) == WD_STEER_RIGHT:
        if (self.drive_state == PROGRAM_DRIVING)
or (self.drive_state == PROGRAM_PARKING):
          self.wheel += WD_STEER_STEP
          if self.wheel > WD_MAX_STEER_RIGHT:
            self.wheel = WD_MAX_STEER_RIGHT
          if (self.alarm != WD_ALARM) and
(self.alarm != WD_BATTERY):
            WD_SET_WHEEL(self.wheel)
      /* 左向きボタン */
      elif WD_COMMAND(cmd) == WD_STEER_LEFT:
        if (self.drive_state == PROGRAM_DRIVING)
or (self.drive_state == PROGRAM_PARKING):
```

```
          self.wheel -= WD_STEER_STEP
          if self.wheel < WD_MAX_STEER_LEFT:
            self.wheel = WD_MAX_STEER_LEFT
          if (self.alarm != WD_ALARM) and
(self.alarm != WD_BATTERY):
            WD_SET_WHEEL(self.wheel)
      /* 右回頭ボタン */
      elif WD_COMMAND(cmd) == WD_RIGHT_ROLL:
        if self.drive_state in (PROGRAM_IDLE,
PROGRAM_PARKING):
          self.drive_state = PROGRAM_ROLLING_RIGHT
          WD_ROLL(ROLL_RIGHT)
          WD_SHOW_ROLLING(Activate(WD_RIGHT_ROLL))
        /* もう一度同じボタンを押したら、回頭を停止する
*/
        elif self.drive_state ==
PROGRAM_ROLLING_RIGHT:
          self.command(WD_STOP)
      /* 左回頭ボタン */
      elif WD_COMMAND(cmd) == WD_LEFT_ROLL:
        if self.drive_state in (PROGRAM_IDLE,
PROGRAM_PARKING):
          self.drive_state = PROGRAM_ROLLING_LEFT
          WD_ROLL(ROLL_LEFT)
          WD_SHOW_ROLLING(Activate(WD_LEFT_ROLL))
        /* もう一度同じボタンを押したら、回頭を停止する
*/
        elif self.drive_state ==
PROGRAM_ROLLING_LEFT:
          self.command(WD_STOP)
      /* 障害物回避ボタン */
      elif WD_COMMAND(cmd) == WD_DISARM:
        if self.alarm == WD_ALARM:
          self.alarm = WD_AVOIDING
          self.avoiding = True
          ISSUE_ALARM(self.alarm)
          self.driver.avoid(self.myID)
       /* 終了ボタン */
      elif WD_COMMAND(cmd) == WD_EXIT:
        self.terminate()
#endif  /* end of #ifdef AP_STUB */
#ifdef STEP_TUNING  /* チューニング用に走行状態を表示す
る */
  /* 定周期割り込みで運転プログラムを実行する */
  /* プログラムの実行状態を返す */
  def control(self):
    print('Speed = ', self.speed, 'Distance = ',
self.driver.RETRIEVE_TRIP_FOR_TUNING(), TO_ERROR)
#endif
#endif
```

モジュール検証

検証プログラム test_web_drive.py は Web ブラウザからのコマンドと同じものを解釈して自律走行プログラムを制御します。AP_STUB を#define すれば、自律走行プログラムのどの機能を呼び出したかが表示され、DRIVER_STUB を#define すれば、運転制御機能をどのように使っているかが表示されます。また SIMULATED_DRIVE を#define すると仮想走行が行え、自律走行車がどんな走行をするか確認できます。

コマンドは標準入力から与えるので、コンソールか、ファイルかを指定できます。コマンドは、休止（P）、再開（S）、終了（T）、AP コマンド（C に続いて一文字）が有効です。なお、'#'の次に数字を与えると、100ms の割り込みを指定回数だけ実行します。'%'を与えると検証プログラムを終了させることができます。

test_web_drive.py
（プロジェクトファイルを参照）

次の実行例では、加速ボタンを 2 回、右ボタンを 1 回クリックした後、休止と再開を行っています。

```
$ cpp -DDRIVER_STUB test_web_drive.py >tmp.py ;
python tmp.py
Initializing Web drive program...
>>> Driver is initialized
:
Done.  Program shall be tested by giving commands
>>> Driver is acquired by  Web
>>> Driver stops and parks
Current Mode =  Parking
Command from Web? R
Current Mode =  Parking
Command from Web? CF
>>> Driver received Speed  40.0 %
>>> Driver starts driving
Current Mode =  Driving
Command from Web? CF
>>> Driver received Speed  50.0 %
Current Mode =  Driving
Command from Web? CR
>>> Driver received Wheel control  2 degrees
Current Mode =  Driving
Command from Web? P
>>> Driver stops and parks
Current Mode =  Parking
Command from Web? S
>>> Driver received Speed  50.0 %
>>> Driver received Wheel control  2 degrees
>>> Driver starts driving
Current Mode =  Driving
Command from Web?
:
```

論理的な動作が検証できたら、SIMULATED_DRIVE を#define して、自律走行車の仮想走行を試してください。プロジェクトファイル test_web.txt から入力を与えると、運転制御機能の評価で見た音符型の軌跡（速度などの動作パラメータが違うので、全く同じではない）が得られます。

```
$ cpp -DSIMULATED_DRIVE test_web_drive.py >tmp.py;
python tmp.py <test_web.txt 2>test_web.csv
:
```

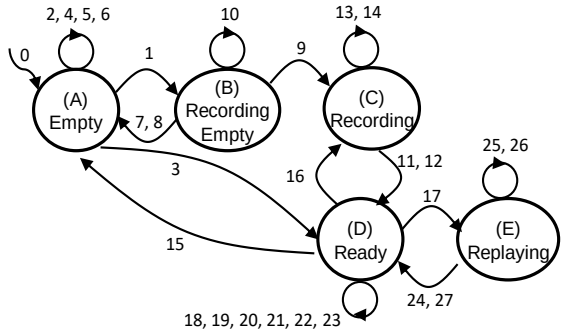
6.3.3ブロックドライブ

ブロックドライブでの運転は直進と回頭の二種類で、方向と走行距離／回転角度は運転制御機能が面倒を見てくれるから、自律走行プログラムで処理することはあまり多くありません。むしろ、その組み合わせで走行を実現することが重要です。そこで複数の走行ブロックを続けて実行する手順を記録・再生できるようにしました。記録中でも走行ブロック（コマンド）実行できるようにしたいので、動作がちょっと複雑になります。ステートマシンとして設計しました。次の 5 つの状態を設けます。

記号	状態名	説明
(A)	Empty	手順が何も記録されていない
(B)	Recording(Empty)	手順の記録を始めたばかりで、まだ空
(C)	Recording	手順を記録中(一つ以上の記録がある)
(D)	Ready	手順が再生可能
(E)	Replaying	手順を再生中

手順に関する状態定義

このステートマシンの状態遷移図は次のようになりました。



手順に関するステートマシンの状態遷移図

ブラウザからのコマンドや内部処理によって動作が変わる様子を、次の状態遷移表で記述します。表が長くなるのを防ぐため、状態は上で決めた記号で表現しています。なお終了（Exit）命令は、状態によらず常に最優先で処理され、ステートマシンが停止します。また作図上 Stop 命令と Disarm 命令は同じ遷移として扱っていますが、Stop 命令も優先して処理するため、実装コードでは遷移番号の後ろに s をつけて、8s などと別扱いしています（Disarm 命令の遷移は 21d などと記載）。

遷移番号	始状態	コマンド	副作用	終状態
0	-	ブロックドライブ起動	初期化	(A)
1	(A)	記録開始	（記録ブロック Active）	(B)
2		記録ロード	手順をロード。手順が空ならそのまま。	(A)
3			手順をロード。手順が空でないなら遷移（クリアブロック Active）	(D)
4		Block 実行コマンド	コマンド実行	(A)
5		Stop/Disarm	コマンド実行	(A)
6		その他の無効コマンド	-	(A)
7	(B)	記録停止	（記録ブロック Inactive）	(A)
8		Stop/Disarm	コマンド実行	(A)
9		Block 実行コマンド	コマンド記録、コマンド実行（クリアブロック Active）	(C)
10		その他の無効コマンド	-	(B)
11	(C)	記録停止	（記録ブロック Inactive）	(D)
12		Stop/Disarm	コマンド実行	(D)
13		Block 実行コマンド	コマンド記録、コマンド実行	(C)
14		その他の無効コマンド	-	(C)
15	(D)	手順クリア	手順をすべて削除（クリアブロック Inactive）	(A)
16		記録開始	（記録ブロック Active）	(C)
17		手順実行	手順の最初のコマンドを実行（手順実行ブロック Active）	(E)
18		手順保存	手順をファイルに保存	(D)
19		ステップ実行	遷移後はじめてなら、最初のコマンドを実行	(D)
20			二回目以降なら、次のコマンドを実行	(D)
21		Stop/Disarm	コマンド実行	(D)
22		Block 実行コマンド	コマンド実行	(D)
23		その他の無効コマンド	-	(D)
24	(E)	Stop/Disarm	コマンド実行	(D)
25		その他の無効コマンド	-	(E)
26		（手順中のブロック実行終了）	次のコマンドがあれば実行	(E)
27			次のコマンドがなければ手順実行ブロックを Inactive にする	(D)

手順に関するブロックドライブの状態遷移表

コマンド実行中は、該当するブロックの表示色を変え、終了すると元の色に戻すため、Web サーバーに情報を渡すようにします。この情報は、すべて同じ Redis FIFO（キーは'Response'）を通し、実行中／実行終了を表す文字の後にコマンドをそのまま伝えるようにしています。ただし、障害物回避ブロックだけは、他の自律走行プログラムと統一するため、警報用の Redis FIFO を通すようにしました。

手順関連のブロックは、状態によって以下のように表示色を変えることにします。手順クリア（手順が記録されているときに Active）を除き、ブロック機能の実行中に Active 表示をします。

ブロック	Inactive 表示	Active 表示
手順クリア	(A), (B)	(C), (D). (E)
記録開始	(A), (D), (E)	(B), (C)
ステップ実行	(A), (B), (C), (D)	(E)
手順実行	(A), (B), (C), (D)	(E)
手順保存	すべて	手順保存中のみ
手順ロード	すべて	手順ロード中のみ

手順に関するブロックの表示色

インクルードファイル

ブロックドライブの状態名やコマンドなどをインクルードファイルで定義しておきます。このファイルは Web ブラウザと共用して、同じ定義を使います。

block_drive.h（抜粋）

```
/* block_drive.h BLOCK drive 用の定義
   初版：2020/9/17 Chuji
   最新版：
*/
#ifndef __BLK_DRIVE_DEFINITIONS
#define __BLK_DRIVE_DEFINITIONS
#include "to_text.h"
#include "drive_program.h"
#include "driver.h"
/* ページタイトル */
#define BLOCK_DRIVE_TITLE  ブロック運転（ブロックを実行して運転します）
/* アラーム発生状態 */
#define BLK_NO_ALARM      PROGRAM_NO_ALARM
#define BLK_ALARM         PROGRAM_ALARM
#define BLK_BATTERY       PROGRAM_LOW_BATTERY
#define BLK_AVOIDING      PROGRAM_AVOIDING
#define BLOCK_STEP_PROCEDURES  ブロックを実行する手順のメニュー
#define BLOCK_BUTTONS     運転ブロック
#define BLOCK_MESSAGE_COLUMN  実行中のブロック
#define BLOCK_EMPTY_TEXT  ''
（中略）
/* Block Drive が処理するコマンド */
#define BLK_STOP  'Stop'
#define BLK_CLEAR  'Clear'
#define BLK_RECORD  'Record'
#define BLK_NO_RECORD  BLK_RECORD
#define BLK_STEP  'Step'
#define BLK_REPLAY  'Replay'
#define BLK_SAVE  'Save'
#define BLK_LOAD  'Load'
#define BLK_SLOW  'SlowSpeed'
#define BLK_MEDIUM  'MediumSpeed'
#define BLK_FAST  'FastSpeed'
#define BLK_FORWARD_10  'Forward10'
#define BLK_FORWARD_30  'Forward30'
#define BLK_BACKWARD_10  'Backward10'
#define BLK_BACKWARD_30  'Backward30'
#define BLK_RIGHT_45  'Right45'
#define BLK_RIGHT_90  'Right90'
#define BLK_LEFT_45  'Left45'
#define BLK_LEFT_90  'Left90'
#define BLK_DISARM  'Disengage'
#define BLK_WAIT_3  'Wait3'
#define BLK_EXIT  'ExitBlock'
/* コマンド一覧表 */
#define BLK_BLOCK_COMMANDS  (BLK_SLOW, BLK_MEDIUM,
BLK_FAST, BLK_FORWARD_10, BLK_FORWARD_30,
BLK_BACKWARD_10, BLK_BACKWARD_30, BLK_RIGHT_45,
BLK_RIGHT_90, BLK_LEFT_45, BLK_LEFT_90,
BLK_DISARM, BLK_WAIT_3)
（中略）
/* Web ページへの更新要求 */
#define BLK_ON  'A'
#define BLK_OFF  'D'
#define BLK_EXEC  'E'
（後略）
```

プログラムファイル

記録した運転手順の実行には、ひとつ注意することがあります。普通のプログラミングでは、while 文で手順を一つずつ取り出して実行すればいいのですが、ブロックドライブでは各手順の実行を下位の運転制御機能に任せているので、割り込みで対応しなければなりません。そのため走行終了のコールバック関数中で、手順の実行中かどうかを判断し、次のコマンドがある場合はそれを実行するようにしました。

block_drive.py（抜粋）

```
/* block_drive.py  ブロック型自律走行プログラム
  初版：2022/4/11 Chuji
  最新版： 2024/3/13 --- 共通オブジェクトを継承するように変更
Class block_driver
属性（継承した属性は省略した）：
  block_commands  有効なコマンドの一覧
  sequence  ブロックの実行手順
  seq     手順を順次実行するためのイテレータ
  seq_state  手順ステートマシンの状態
  from_first_step  手順を最初から実行するフラグ（実行中は False）
  executing_block  現在実行中のブロック
  driving_speed  指定された走行速度（走行速度の現在値ではない）
操作（継承した操作は省略した）：
  command  自律走行プログラムに命令を与える
  stopped  自律運転停止時のコールバック関数（共通部で一部コーディング済）
*/
#include "include/use-math.h"
#include "include/block_drive.h"
#include "include/use-time.h"
#include "drive_program.py"
/* Web ページの表示更新 --- ブロックボタンの表示を変える */
（中略）
#define BLOCK_ON(x)
self.fifo.rpush(REDIS_TO_RESPONSE, BLK_ON + x)
#define BLOCK_OFF(x)
self.fifo.rpush(REDIS_TO_RESPONSE, BLK_OFF + x)
#define BLOCK_EXEC(x)
self.fifo.rpush(REDIS_TO_RESPONSE, BLK_EXEC + x)
#endif
（中略）
/* ブロックドライブを親オブジェクトを継承する形で定義する */
class block_driver(Drive_program):
  def __init__(self, ID, sensor, driver, fifo):
    super().__init__(ID, sensor, driver, fifo)
    /* この自律走行プログラム固有の属性定義と初期化 */
    /* 有効なコマンドの一覧 */
    self.block_commands = BLK_BLOCK_COMMANDS
    self.sequence = [] /* ブロックの実行手順 */
    self.seq = iter(self.sequence) /* 手順を取り出すため */
    self.seq_state = SEQ_EMPTY /* 遷移番号 0 */
    self.executing_block = None /* 現在実行中のブロックはない */
    self.from_first_step = True /* 手順は最初から実行する */
    self.driving_speed = BLK_STOP_SPEED /* 最初の走行速度指定値 */
  /* 運転制御機能からのコールバック処理の追加--- ブロック運転実行の終了 */
  def stopped(self, cause):
    super().stopped(cause) /* 共通処理 */
#ifndef AP_STUB
    if cause == REACHED_GOAL:
      if self.executing_block != None:
        /* 実行中のブロックの表示をオフにする */
        BLOCK_OFF(self.executing_block)
        self.executing_block = None
      /* 手順実行の 1 ステップが終わった */
      if self.seq_state == SEQ_REPLAYING:
        next_step = next(self.seq, None)
        /* 状態遷移図の遷移番号 26
        --- 手順実行中で、次に実行するブロックがある */
        if next_step != None:
          self.interprete(next_step)
        else: /* 同遷移番号 27 */
```

```
        /* 手順の最後まで実行した */
          self.seq_state = SEQ_READY
          self.from_first_step = True
          BLOCK_OFF(BLK_ID_REPLAY)
      /* 単独コマンドあるいはステップ実行の終わり */
      elif self.seq_state == SEQ_READY:
        BLOCK_OFF(BLK_ID_STEP)
#endif
  /* コールバック関数のアラーム処理部 */
  /* アラーム処理（driver のコールバック処理から呼ばれる）
*/
  def my_alarm(self):
    if self.alarm == BLK_NO_ALARM:
      CLEAR_ALARM(self.alarm)
    else:
      ISSUE_ALARM(self.alarm)
      /* 実行中のブロック、手順のクリア */
      if self.seq_state == SEQ_REPLAYING: /* 遷移番
号 27 */
        self.seq_state = SEQ_READY
        self.from_first_step = True
        BLOCK_OFF(BLK_ID_REPLAY)
      if self.executing_block != None:
        BLOCK_OFF(self.executing_block)
        self.executing_block = None
（中略）
  /* コマンド処理 */
  def command(self, cmd):
（中略）
  /* 手順のうち、即時実行命令の終了処理 --- 次のブロック実
行 */
  def test_next_step(self):
    if self.seq_state == SEQ_REPLAYING:
      self.executing_block = None
      self.stopped(REACHED_GOAL)
  /* 障害物回避命令の実行 */
  def disarm(self):
    self.driver.avoid(self.myID)
    self.alarm = BLK_AVOIDING
    ISSUE_ALARM(self.alarm)
  /* ブロックコマンドの解釈 */
  def interprete(self, cmd):
    if cmd == BLK_WAIT_3:  /* 3 秒間停止 */
      self.executing_block = cmd
      BLOCK_ON(BLK_ID_WAIT_3)
      sleep(3)
      self.stopped(REACHED_GOAL)
      self.test_next_step()
    elif cmd in BLK_SPEEDS: /* 速度設定 */
      for s in BLK_SPEEDS:
        BLOCK_OFF(s)
      self.driving_speed = BLK_GET_SPEED[cmd]
      BLOCK_ON(cmd)
#ifdef AP_STUB
      print('Speed is set to ', self.speed,
TO_ERROR)
#endif
      self.test_next_step()
    elif cmd in BLK_TRIPS: /* 指定距離の走行 */
      self.executing_block = cmd
      BLOCK_ON(cmd)
      distance = BLK_GET_TRIP[cmd]
#ifndef AP_STUB
      if IS_POSITIVE(distance): /* 前進 */
        self.driver.speed_control(self.myID,
self.driving_speed)
      else:  /* 後退 */
        self.driver.speed_control(self.myID, -
self.driving_speed)

      self.driver.set_trip(self.myID,
DRIVE_TRIP_SET, abs(distance))
      self.driver.drive(self.myID)
      self.drive_state = PROGRAM_DRIVING
（後略）
```

モジュール検証

検証プログラム test_block_drive.py は Web ドライブ用とほぼ同じ構造をしています。

AP_STUB か DRIVER_STUB を#define すれば、ブロックドライブが運転する意思が表示されます。DRIVER_STUB で距離・角度の目標値をもって走行させるときには、走行終了のコールバックを呼びます（指示'$0'を与える）。ブロックドライブへのコマンドは、冒頭に'C'を付ける必要があります。

test_block_drive.py
（プロジェクトファイルを参照）

また SIMULATED_DRIVE を#define すると仮想走行が行え、自律走行車がどんな走行をするか確認できます。このとき、コンソール入力を文字ファイル test_block.txt（プロジェクトファイルに含まれています）にリダイレクトすると、運転制御機能の検証で行ったような四角形の仮想走行が行えます。標準出力をファイルにリダイレクトすると表計算ソフトで軌跡をプロットできます。

Web ブラウザへの表示（ブロックの表示色変更）は、別の SSH 窓でキー'Response'の Redis FIFO を読み出す（$ python popper.py Response）ことで確認できます。'A'+コマンドは該当するブロックの表示オン（色を濃くする）、'D' +コマンドは表示オフ（色を薄くする）ように html ファイルを設計します。

手順の記録・再生の検証は test_block1.txt（プロジェクトファイルに含まれています）を参考にしてください。

6.3.4 Scratch ドライブ

Scratch ドライブの実行制御は、ブロックドライブから手順処理を省いて（コマンド数は多いが）簡略化したものです。コマンドによっては、目標あり走行を二回（方向転換と一定距離走行）行う必要があるので、ブロックドライブのコールバック処理で手順を実行したのと類似の方法で二回目の走行をするようにしました。走行データは next_move と next_parm という変数に記憶しておきます。将来三回目以降の走行が必要になったら、ブロックドライブのようにリストにすることを考えます。

ブロックドライブでは、コマンド実行中に次のコマンドが来ても無視するようにしました。しかし Scratch ドライブでこれをやってしまうと、Scratch プログラムが進んでいく段階で、一部のブロックが実行されないことになりかねません。そこで、Scratch プログラムと Scratch ドライブとの間で、ハンドシェークをさせることにしました。すべてのコマンドの終了時には、終了通知（走行終了または

コマンドエラー）を返すことにます。Scratch の各ブロックには、終了通知を受けるまで待機させる（次のブロックには進まない）ことにします。際限なく待つとプログラムが永久に終わらないので、適当なタイムアウト時間が経っても通知が届かないときは、エラー処理をするようにします。

Scratch ドライブでは、座標を使って指示することがあるので、自分の居場所としての座標を記憶しておき、移動するたびに更新します。この座標は『自分がそうだと思っている』位置で、実際の位置とは（走行計の分解能と誤差、それに読み出し周期の（100ms）間に目標を行き過ぎてしまうせいで）動くたびにずれていってしまいます。

インクルードファイル

scratch_drive.h

```
/* scratch_drive.h SCRATCH drive 用の定義
  初版：2020/9/17 Chuji
  最新版：
*/
#ifndef __SCRATCH_DRIVE_DEFINITIONS
#define __SCRATCH_DRIVE_DEFINITIONS
#include "driver.h"
#include "drive_program.h"
#include "hmi.h"
#define SCRATCH_DRIVE_TITLE     Scratch ドライブ(外部プログラムの指示で走行します)
#define SC_ID   SCRATCH_CONTROLLER  /* 運転制御するときの自分の ID */
/* 状態定義 */
#define SC_INACTIVE  PROGRAM_INACTIVE
#define SC_IDLE    PROGRAM_IDLE
#define SC_DRIVING  PROGRAM_DRIVING
#define SC_PARKING  PROGRAM_PARKING
#define SC_PAUSING  PROGRAM_PAUSING
/* コマンド取得関数 */
#define SC_COMMAND(x)  x[0]
#define SC_GET_NUMS(x)  re.findall('[-+]?[0-9]+', x)
#define SC_PARMS(x)  SC_GET_NUMS(x)
#define SC_SCALE  10 /* cm -> mm */
/* Scratch Drive が処理するコマンド */
#define SC_INITIALIZE  'I'
#define SC_CALIBRATE  'C'
#define SC_SET_COORDINATE  'X'
#define SC_SET_ORIENTATION  'O'
#define SC_GO_FORWARD  'F'
#define SC_GO_BACKWARD  'B'
#define SC_TURN    'T'
#define SC_ALIGN  'A'
#define SC_TURN_AROUND  'R'
#define SC_GO_TO  'G'
#define SC_SPEED  'S'
#define SC_DISARM  'D'
#define SC_EXIT    'E'
#define SC_DELIMITER  ',' /* 数値表現の区切り記号 */
/* 内部命令キューで使うコマンド */
#define SC_STRAIGHT_DRIVE  's'
#define SC_ROLL_DRIVE  'r'
/* コマンド実行結果 */
#define SC_FINISHED  'C' /* コマンドが正常終了した */
#define SC_MET_ERROR  'E' /* コマンドのパラメータが不正で処理されなかった */
#define SC_EMERGENCY  'A' /* アラームが発生したため、コマンド処理が中途で終わった */
/* 座標・移動量（単位は mm）と方位（単位はラジアン） */
#define SC_BACK_STEP    80 /* 車両の長さの半分だけ後退する */
#define SC_TURNAROUND_ANGLE  math.pi /* 反転する角度 */
#define MAX_X_COORDINATE  1000 /* 単位は mm なので 1m */
#define MIN_X_COORDINATE  -MAX_X_COORDINATE
#define MAX_Y_COORDINATE  MAX_X_COORDINATE
#define MIN_Y_COORDINATE  -MAX_Y_COORDINATE
#define MAX_DRIVE_DISTANCE  MAX_TRIP
#define MIN_DRIVE_DISTANCE  0
#define ORIGIN_X    0
#define ORIGIN_Y    0
#define ORIGIN_THETA    0
#define MAX_ORIENTATION    math.pi
#define MIN_ORIENTATION    -MAX_ORIENTATION
#define ORIENTATION_PERIOD  (2 * math.pi) /* -180 度から 180 度の範囲外の補正 */
#define SC_TRIP    DRIVE_TRIP_SET
#define SC_ROLL    DRIVE_ROLL_SET
/* 速度と操舵量 */
#define SC_STOPPING  STOP_SPEED
#define SC_STRAIGHT  STRAIGHT_WHEEL
#define SC_ROLLING  HARD_STARBOARD
#define SC_RIGHT_90  math.pi/2
#define SC_LEFT_90  -SC_RIGHT_90
#define SC_COMPLEMENTARY(x)  (math.pi - x) /* 補角 */
/* 後退・方向転換量 */
#define SC_STEP_BACK  50 /* 50mm */
#define SC_FULL_TURN  MAX_ORIENTATION
/* コマンド作成関数 */
#define SC_CREATE_COMMAND(c, x)  c + str(x)
/* 処理結果を Scratch プログラムに知らせる */
#ifdef SC_DEBUG
#define SC_COMPLETED(x)  print('Scratch command <', x, '> is completed', TO_ERROR)
#define SC_ERROR(x)  print('Scratch command <', x, '> met an error', TO_ERROR)
#define SC_ALARM(x, y)  print('Scratch command <', x, '> is interrupted by an alarm <', ALARM_MESSAGES[y], '>.', TO_ERROR)
#define SC_TERMINATE_AP  print('Scratch drive program is terminated.')
#else
#define SC_COMPLETED(x)
self.fifo.rpush(REDIS_TO_SCRATCH, SC_FINISHED + x)
#define SC_ERROR(x)
self.fifo.rpush(REDIS_TO_SCRATCH, SC_MET_ERROR + x)
#define SC_ALARM(x, y)
self.fifo.rpush(REDIS_TO_SCRATCH, SC_EMERGENCY + x + ALARM_MESSAGES[y])
#define SC_TERMINATE_AP
self.fifo.rpush(REDIS_TO_HMI, HMI_TERMINATE)
#endif
/* 速度・操舵量の表示更新 --- ブラウザには表示欄がないかもしれないが */
#define SC_SHOW_SPEED(x)
SHOW_SPEED(self.fifo, x)
#define SC_SHOW_WHEEL(x)
SHOW_WHEEL(self.fifo, x)
#endif
```

プログラムファイル

プログラムファイル scratch_deive.py は長いので抜粋を掲載します。

scratch_drive.py（抜粋）

```
/* scratch_drive.py SCRATCH による運転制御
  初版：2020/9/17 Chuji
  最新版：2024/3/13 --- 共通オブジェクトを継承するように変更
          2022/6/21 --- テンプレート方式に変更
Class: scratch_driver
属性（継承した属性は含まない）：
  busy     走行中（次のコマンドを受け付けない）
  cmd     Scratch からの命令（終了報告用）
  next_move   組み合わせ走行後半の動作
  next_parm   組み合わせ走行後半の移動距離/回転角度
```

```
  x, y     現在の(x,y)座標（単位は mm）
  t(theta)  現在の方位（北を中心にプラス・マイナス 180
度、実際の単位はラジアン）
操作（継承した操作は含まない）:
  command  自律走行プログラムに命令を与える
  stopped  自律運転停止時のコールバック関数
  travel   指定距離だけ直進する
  roll     指定角度だけ回頭する
  コマンドで角度を渡すのは度単位。内部ではすべてラジアンで処
理する。
*/
#include "include/use-time.h"
#include "include/use-RegularExpression.h"
#include "include/use-math.h"
#include "include/scratch_drive.h"
#include "include/hmi.h"
#include "drive_program.py"
class scratch_driver(Drive_program):
  def __init__(self, ID, sensor, driver, fifo):
    super().__init__(ID, sensor, driver, fifo)
 (中略)
  def start(self):
    super().start()
    /* 初期化で運転速度がゼロにされてしまうので、再設定 */
    self.speed = SLOW_AHEAD
  /* 運転制御機能からのコールバック処理の追加--- プログラ
ム運転の終了 */
  def stopped(self, cause):
    super().stopped(cause) /* 共通処理 */
#ifndef AP_STUB
    if cause == REACHED_GOAL:
      if self.__next_move != None: /* 次の走行がある
*/
        if self.__next_move == SC_STRAIGHT_DRIVE:
          self.travel(self.__next_parm)
 (中略)
    /* 現在の向きに指定距離だけ前進する */
    elif self.__cmd == SC_GO_FORWARD:
      parms = SC_PARMS(cmd)
      if len(parms) < 1:
        SC_ERROR(self.__cmd)
        return
      distance = SC_SCALE * float(parms[0])
      x = self.__x + distance * MY_SIN(self.__t)
      y = self.__y + distance * MY_COS(self.__t)
      if (x > MAX_X_COORDINATE) or (x <
MIN_X_COORDINATE) or (y > MAX_Y_COORDINATE) or (y
< MIN_Y_COORDINATE):
        SC_ERROR(self.__cmd)
        return
      self.__x = x
      self.__y = y
      self.travel(distance)
 (中略)
  /* 指定距離だけ直進する */
  def travel(self, d):
#ifdef SHOW_SC
    print('Travel ', d, 'mm with speed: ',
self.speed)
#endif
#ifndef SC_DEBUG
    self.driver.set_trip(self.myID, SC_TRIP,
abs(d))
    if d >= 0: /* 前進 */
      self.driver.speed_control(self.myID,
self.speed)
      SC_SHOW_SPEED(self.speed)
 (中略)
  def roll(self, t):
#ifndef SC_DEBUG
    self.driver.set_trip(self.myID, SC_ROLL,
abs(t))
    self.__busy = True
    SC_SHOW_SPEED(ROLLING_SPEED)
    if t > 0: /* 右回頭 */
      self.driver.roll(self.myID, ROLL_RIGHT)
      self.drive_state = PROGRAM_ROLLING_RIGHT
      SC_SHOW_WHEEL(ROLL_RIGHT_WHEEL)
    else:     /* 左回頭 */
      self.driver.roll(self.myID, ROLL_LEFT)
      self.drive_state = PROGRAM_ROLLING_LEFT
      SC_SHOW_WHEEL(ROLL_LEFT_WHEEL)
 (後略)
```

モジュール検証

検証モジュールは、掲載していません。Scratch ドライブはブロック型（何かを始めると終わるまでプログラムを占有する）処理なので、仮想走行で検証します。

test_scratch_drive.py
（プロジェクトファイルを参照）

最初に別の窓でキー'Scratch'の Redis FIFO を読み取るようにしておきます。SIMULATED_DRIVE を #define して、仮想走行を実行します。走行コマンドの後にコマンド'#20'などで周期ごとの処理を呼び出せば、車両の位置と角度が刻々と変化していきます。目標に到達すると、FIFO に完了メッセージが表示されます。

コマンドを与えて、座標の変化を確認します。走行計の分解能が悪いせいで、走行を続けると座標誤差が蓄積して、どんどん表示がずれていってしまいます。実走行では「自分が把握している座標」が基準になるので、それに従って走行します。

```
$ python popper.py Scratch
Text: CF from: Scratch
Text: CB from: Scratch
:
```

```
$ cpp -DSIMULATED_DRIVE test_scratch_drive.py
>tmp.py; python tmp.py
Initializing Scratch drive program...
0.0,  0.0,  0.0,  0.0
Done.  Program shall be tested by giving commands
Command from Scratch program? CF10
Command from Scratch program? #10
0.1,  0.0,  1.2,  0.0
0.2,  0.0,  2.4,  0.0
0.3,  0.0,  3.6,  0.0
0.4,  0.0,  4.8,  0.0
0.5,  0.0,  6.0,  0.0
0.6,  0.0,  7.2,  0.0
0.7,  0.0,  8.4,  0.0
0.8,  0.0,  9.6,  0.0
0.9,  0.0,  10.8,  0.0
1.0,  0.0,  10.8,  0.0
Command from Scratch program? CB10
Command from Scratch program? #10
1.1,  0.0,  9.6,  0.0
:
```

6.3.5ライントレーシング

この自律走行プログラムは自動運転（Line-tracing）と手動運転（Web-driving）という二つのモードがあります。手動運転モードは Web ドライブと同じ動作をします（親オブジェクト）。

自動運転モードでは、走路に引かれた曲線（『ガイドライン』と呼ぶことにします）を追従するように走行します。その時『走路からのずれ』を、車体下に設置した 5 個の走路センサを使って測定します。

ガイドラインの幅が走路センサの間隔（10.16mm）より狭いと、センサの間に入ったときに見失ってしまうので、それより広くします。実際の走路では、幅 18mm のビニルテープを使いました。

この『ずれ（偏差と呼ぶことにします)』を走路センサの検出パターン（右端が最下位のバイナリー表示）から決めます。ここでは黒いガイドラインの場合を用意し、白いガイドラインでは検出パターンを反転してから調べるようにします。

隣り合う走路センサの間隔を 1 とした時、以下のように偏差を決めることにします。表では、ガイドラインの幅が 1 と推測できる場合を茶色、2 の場合を赤、3 の場合をオレンジ、4 の場合を黄色で示しています（車体がガイドラインに対して斜めになっている場合もある)。検出パターンのうち、薄い青色とピンクはノイズだと（勝手に）推測して、偏差を決めました。Uncertain と書いたところは偏差を推測できないパターンで、制御には使わないことにします。End と書いたパターンはガイドラインと直角な向きにいるか、終点に T 字型のパターンを見つけたことを想定し、走行を停止することにします。Lost と書いたパターンはガイドラインを見失ったことを示すので、やはり走行を停止します。

Code	検出パターン	センサ出力	Code	検出パターン	センサ出力	Code	検出パターン	センサ出力	Code	検出パターン	センサ出力
0		Lost	8		-1	16		-2	24		-1.5
1		+2	9		-1	17		Uncertain	25		-1.5
2		+1	10		0	18		+1	26		-1.5
3		+1.5	11		+0.5	19		+1.5	27		End
4		0	12		-0.5	20		0	28		-1
5		0	13		-0.5	21		Uncertain	29		-1
6		+0.5	14		0	22		+0.5	30		-1.5
7		+1	15		+1.5	23		+1	31		End

走路センサの検出パターンから偏差を求める辞書

コラム　世界最大の･･･

ロンドンを周回する高速 25 号線（M25）は、世界最大級の環状高速道路です。1986 年の開通式にはサッチャー首相（当時）が誇らしげにテープカットを行いました。

同時に、世界一混雑した道路としても知られています。開通当時から「容量不足」が指摘され、それから絶え間なく車線拡張工事が続いています（一部で片側 6 車線)。2003 年にはヒースロー空港周辺で一日 20 万台の通行が記録されました。過去最高の交通量は、空港入り口と M4 ジャンクション間で一日 26 万台だったそうです（2014 年)。

交通量が多いことに加え、相次ぐ拡張工事のせいで渋滞が激しく、『世界一混雑する道路』と言われています。誇り高き英国人の友人は、自嘲的に『世界最大の駐車場』と呼んでいました。日本のゴールデンウイークもすごいけど、これはとんでもない規模……

この偏差を PID 制御モジュールに与えるため、signal_conditioner.py というモジュールを追加しました。出力は偏差と、その可用性（使える・使えないのフラグ）です。Lost の場合は警報停止、End の場合は走行終了のコールバックを呼んで停車させます。Uncertain の場合は、しばらく前回値を使い、パターンが続いたら停車します。実際には表の値を二倍して使います。

PID 制御は、以前に温度コントローラで作成したモジュールのアルゴリズムをそのまま使います。ただし、目標値（SP）は常に『ガイドラインの中央』なので、0 です。また、温度コントローラの制御出力は電熱器を ON/OFF するための PWM 信号（0～100%）でしたが、自律走行車では操舵量（-100～+100%）なので、制御出力と積分項・微分項の下限（PID_MIN_PARM, PID_MIN_INTEGRAL）を-100 に変更します。これは、cpp 処理の-D オプション（-DBIPOLAR_PID）で指定することにします。

操舵量を積分したものが車両の向き（の変化）で、向きと速度を積分したものが車両の位置になります。これは二階の微分方程式で記述されるので、応答は振動しやすくなります。安定した走行には微分項が必須で、P 制御や PI 制御ではなく、PID 制御を採用します。

インクルードファイル

インクルードファイルでは、上の偏差辞書をタプル（変更できないリスト）ERROR_TABLE として用意します。このファイルは、Web ブラウザの表示に使う定義も含んでいますが、それについては次章で説明します。

line_trace.h（抜粋）

```
/* line_trace.h ラインとレーシング用の定義
   初版：2020/9/17 Chuji
   最新版：
*/
#ifndef __LINE_TRACE_DEFINITIONS
#define __LINE_TRACE_DEFINITIONS
#include "autonomous_vehicle.h"
#include "driver.h"
/* PID 制御の範囲を両極性にする */
#define BIPOLAR_PID
（中略）
/* 動作モード定義（PID 制御の拡張） */
#define LINE_INACTIVE  PROGRAM_INACTIVE  /* Out Of Service に相当 */
#define LINE_MANUAL  PID_MANUAL
#define LINE_AUTOMATIC  PID_AUTO
/* 拡張モード（Web 表示に使う */
#define LINE_GOALED  'GOAL'
#define LINE_LOST_GUIDE  'LOST'
#define LINE_NO_PID  'NoPID'
/* PV 計算結果の status として、pocess value の status (0 or 1) を拡張して使う */
#define STATUS_GOAL     2
（後略）
```

プログラムファイル

signal_conditioner.py は、走路センサの検出パターンから、走路の色を指定して偏差を決定します。出力は偏差（表の 2 倍の値）value と、偏差の可用性 status からなる構造体です。可用性は GOOD と BAD の他に、LOST と END を追加しました。表を引くだけなので、単独の検証はしません。

signal_conditioner.py

（プロジェクトファイルを参照）

自動運転モードでは、走路の偏差を PID 制御機能に与えて操舵量を計算します。このモジュールは温度コントローラのプロジェクトで検証済なので、説明を省略します。

pid.h, pid.py, test_pid.py, test_pid.xlsx

（プロジェクトファイルを参照）

自律走行プログラムは、これらの下位モジュールと、手動運転モード時に使う Web ドライブ機能を組み込んで使います。このとき、偏差のステータスとして LOST あるいは GOAL が返された場合は、自動走行を止めて停車し、自動運転を解除します。

line_trace.py

```
/* line_trace.py ラインとレースによる自動運転
   初版：2020/9/17 Chuji
   最新版：2024/3/13 --- 共通オブジェクトを継承するように変更
           2022/6/21 --- テンプレート方式に変更
Class: line_tracer
属性（継承した属性は省略した）：
   sc      偏差計算オブジェクト
   pid     PID 制御アルゴリズム
   pv      PID 制御に与える現在の偏差
   last_pv   最後に有効だった PV の値
   lost_count   連続してガイドラインを見失った回数
   default_pv   手動運転時の PV（ダミー）
   pid_modes   PID 制御モード表示テキスト辞書
   color      追従するガイドラインの色
   line_sensor   ガイドラインセンサオブジェクト
   first_action   Web ドライブモードに戻った直後を示すフラグ
操作：（継承した操作は省略した）
   command   自律走行プログラムに命令を与える
   control   自律走行プログラムの定周期実行を行う
   stopped   自律運転停止時のコールバック関数
   guide_line   走路センサのデータを取り出す（HMI 表示用）
*/
#include "include/use-sys.h"
#include "include/sensor.h"
#include "hmi_format.py"
#include "include/line_trace.h"
#include "web_drive.py"
#include "signal_conditioner.py"
#include "pid.py"
#define MY_AP_ID  TRACE_CONTROLLER
class line_tracer(web_driver):
  def __init__(self, ID, sensor, driver, fifo):
    super().__init__(ID, sensor, driver, fifo)
    /* この自律走行プログラム固有の属性を初期化 */
#ifndef AP_STUB
#ifndef SIMULATED_DRIVE
    self.line_data = self.line_sensor.read_line()
#endif
    self.color = BLACK_COLOR  /* デフォルトは黒線 */
```

```
    WD_SHOW_COLOR(Deactivate(WD_LINE_COLOR))
    self.sc = signal_conditioner()
    self.last_pv = MID_SHIP
    self.lost_count = NOT_LOST
    self.pid = pid_block()  /* PID 演算モジュール */
    self.pid.set_target(PID_MANUAL)
    self.default_pv = process_value()
    self.default_pv.value = MID_SHIP
    self.default_pv.status = STATUS_GOOD
    self.pid.execute(self.default_pv)
    self.mode = LINE_MANUAL  /* 手動運転モード */
    self.first_action = True
    WD_SHOW_MODE(LINE_MODES[self.mode])
#endif
  /* アラームの処理は Web Drive と同じ */
  /* 定周期割り込みで運転プログラムを実行する */
  /* プログラムの実行状態を返す */
  def control(self):
#ifdef AP_STUB
    print('Line_trace.control is called.',
TO_ERROR)
#endif
    /* 自動運転中でなければ何もしない */
    if (self.state != PROGRAM_INACTIVE) and
(self.mode == LINE_AUTOMATIC):
#ifndef SIMULATED_DRIVE  /* 実走行 */
      self.line_data =
self.line_sensor.read_line()
#ifdef DEBUG
      print('{0:05b}\t'.format(self.line_data),
end='')
#endif
#else  /* シミュレーション走行 */
      x, y, t = self.driver.sim_coordinate()
      self.line_data = read_line(x, y, t)
#ifdef DEBUG_LINE
      print('{0:05b}\t'.format(self.line_data),
end='', TO_ERROR)
#endif
#endif
      /* センサ入力からガイドラインからの偏差を計算する */
      pv = self.sc.deviation(self.line_data,
self.color)
#ifdef DEBUG_LINE
      print(' pv = ', pv.value, pv.status,
TO_ERROR)
#endif
      /* 走路を見失ったときのためにデータを取っておく */
      if pv.status == STATUS_GOOD:
        self.last_pv = pv.value
        self.lost_count = NOT_LOST
      /* 迷ったら停車する */
      if pv.status == STATUS_LOST:
        self.lost_count += 1
        /* 連続して走路を見失ったら自動走行を止める */
        if self.lost_count >= DETECT_LOST:
          pv.status = STATUS_BAD
          self.return_to_WD_mode(LINE_LOST_GUIDE)
          return self.state
        /* 見失っても何回かは回復を試みる */
        else:
          if self.last_pv > 0:
            pv.value = self.last_pv +
ADDITIONAL_ERROR
          else: /* last_pv < 0 ...ゼロはあり得ない */
            pv.value = self.last_pv -
ADDITIONAL_ERROR
          pv.status = STATUS_GOOD
       /* 終端を検出したら停車する */
      elif pv.status == STATUS_GOAL:
        pv.status = STATUS_BAD
        self.return_to_WD_mode(LINE_GOALED)
        return self.state
      /* そうでなければ PID 制御で走行する */
      mv = self.pid.execute(pv)
      mv.value *= -1
/*      print(mv.value, mv.status) */
      /* PID 制御ができなくなったら停車する */
      if mv.status == STATUS_BAD:
        self.return_to_WD_mode(LINE_NO_PID)
      else:   /* 制御は正常 */
        self.wheel = mv.value
#ifdef SHOW_PID
        print('A= %2.4f %s pv= %2.1f
Wheel= %4.3f' % (PID_P_INIT,
format_line(self.line_data), pv.value,
self.wheel), TO_ERROR)
#endif
        self.driver.wheel_control(MY_AP_ID,
self.wheel)
      return self.state
  /* 表示のために走路ガイドラインのデータを読み取る */
  def guide_line(self):
    /* プログラム実行中は使えるデータが存在する */
    if self.state == PROGRAM_RUNNING:
      return self.line_data
    /* 実行中でなければセンサを読み取ってからデータを返す
*/
    else:
#ifndef SIMULATED_DRIVE
      return self.line_sensor.read_line()
#else
      return LINE_DEFAULT_DATA
#endif
  /* アラーム停車時の動作 */
  def return_to_WD_mode(self, reason):
    /* 停車状態になる */
    self.mode = LINE_MANUAL
    self.drive_state = PROGRAM_PARKING
    WD_SHOW_MODE(LINE_MODES[reason])
    self.first_action = True
    /* pid 制御機能を手動動作にする */
    self.pid.set_target(PID_MANUAL)
    self.pid.execute(self.default_pv)
    /* 運転制御機能に停車させる */
    self.driver.stop(MY_AP_ID)
    self.speed = STOP_ENGINE
    self.wheel = MID_SHIP
   /* 運転プログラムにコマンドを与える */
  def command(self, cmd):
    if self.state != PROGRAM_INACTIVE:
      /* 自動走行中は受け付けないコマンド */
      if (self.mode == LINE_AUTOMATIC) and (cmd in
PROHIBITED_COMMANDS):
        return
      /* 何らかの操作が行われたので、前回の停車理由の Web
表示をクリアする */
      if self.first_action:
        WD_SHOW_MODE(LINE_MODES[self.mode])
        self.first_action = False
    /* Web Drive と共通のコマンド処理 */
    super().command(cmd)
    /* Line Trace 固有のコマンド処理 */
    if self.state != PROGRAM_INACTIVE:
      /* 動作モードを変更する */
      if LINE_COMMAND(cmd) == WD_MODE_TOGGLE:
        /* 自動運転から手動運転に移行 */
        if self.mode == LINE_AUTOMATIC:
          self.mode = LINE_MANUAL
          /* 停車状態から始める */
          self.driver.stop(MY_AP_ID)
          self.speed = STOP_ENGINE
          self.wheel = MID_SHIP
          self.drive_state = PROGRAM_PARKING
          WD_SET_WHEEL(self.wheel)
          self.pid.set_target(PID_MANUAL)
          self.pid.set_mv(self.wheel)
        /* 手動運転から自動運転に移行 */
        else:
          self.mode = LINE_AUTOMATIC
          self.speed = LINE_SPEED
          WD_SET_SPEED(self.speed)
          self.wheel = MID_SHIP
          WD_SET_WHEEL(self.wheel)
          self.pid.set_target(PID_AUTO)
          self.pid.bumpless()
          self.drive_state = PROGRAM_DRIVING
          self.driver.drive(MY_AP_ID)
        WD_SHOW_MODE(LINE_MODES[self.mode])
      /* 走路の色を変更する */
      elif LINE_COMMAND(cmd) ==  WD_LINE_COLOR:
        if self.color == BLACK_COLOR:
          self.color = WHITE_COLOR
          WD_SHOW_COLOR(Activate(WD_LINE_COLOR))
        else:  /* e.g. self.color == WHITE_COLOR
*/
          self.color = BLACK_COLOR
          WD_SHOW_COLOR(Deactivate(WD_LINE_COLOR))
```

走路シミュレーション

ライントレーシングには、走路のガイドラインが必要です。実際の走路では床に着色テープなどでガイドラインを設置しますが、検証はシミュレーションで行えます。下位ルーチンとして、(driver.py の属性になっている）現在の車両の位置と向きから、走路センサの出力を計算する関数を用意しました。このファイル simulated_track.py は掲載していません。計算の仕方はこの本の付録を、実際のコードはプロジェクトファイルを参照してください。

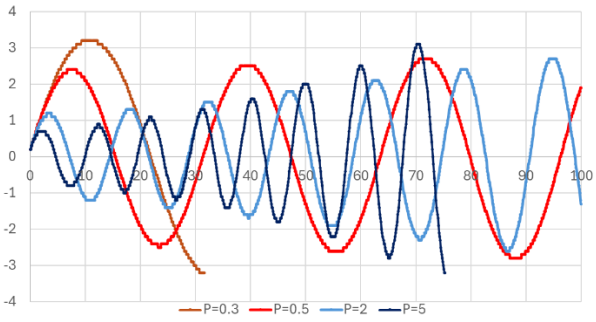
次の二つの走路を選んで検証できます。

- Y 軸方向に伸びる直線走路（STRAIGHT_TRACK）
- 原点を中心とした円環状走路（CIRCUIT_TRACK）

PID 制御のパラメータも、このシミュレーションで決めてみます。走路幅 12mm の直線走路（車両位置の初期値は中央から 2mm ずれた位置で、走路から右に 2 度傾いている）を比例制御（I = D = 0）で P パラメータを変えながら仮想走行を行ってみます。

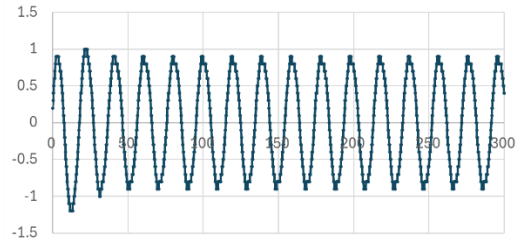
```
$ cpp -DSHOW_PID -DSIMULATED_DRIVE
-DPID_P_INIT=2.0 -DPID_I_INIT=0 -DPID_D_INIT=0
-DCOURSE_WIDTH=12 -DSIM_X_ORIGIN=2
-DSIM_THETA_ORIGIN=0.04 -DLINE_SYMBOL_WHITE=\'-\'
-DLINE_SYMBOL_BLACK=\'o\' test_line_trace.py
>tmp.py; python tmp.p
```

次の図ではガイドライン中央からの偏差（cm）を縦方向に表示しています。一部のデータが途中までしかないのは、ガイドラインがセンサから外れ、見失ったと判断したためです。



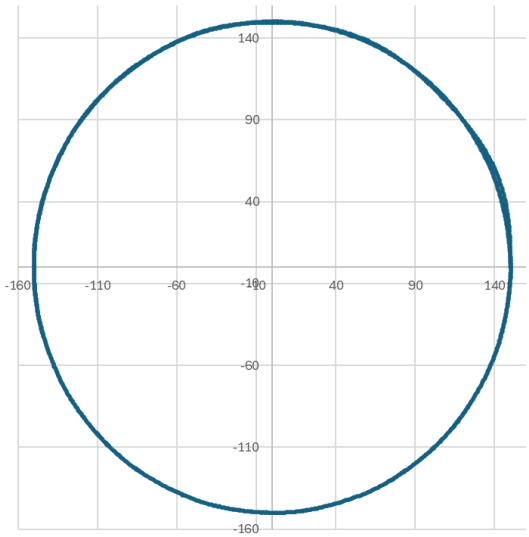
直線走路で P を変えながら仮想走行（横軸は秒）

P=5 (=Kc)以上では急激に不安定（周期 Tc=10 秒）になりました。これをもとに、ジーグラとニコルスの限界感度法でパラメータ（P (=0.6Kc) = 3、I (=0.5Tc) = 5 秒、D (=0.125Tc) = 1.2 秒）を決めました。このパラメータで 300 秒間 PID 制御を行ってみた結果は次のとおりです（縦軸の単位は cm)。走路センサの分解能（±1cm）以内に制御できており、車両の向きは 20 秒弱の周期で左右に揺れていることが分かります。



PID 制御で直線走路を仮想走行したときの偏差

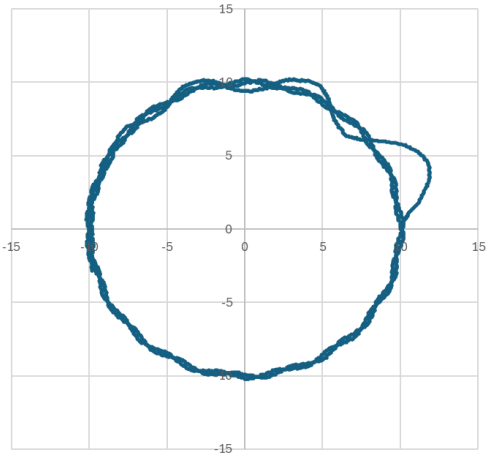
同じパラメータを使い、半径 150cm の円形走路で仮想走行（左回り）した結果を次に示します。



円形走路での仮想走行結果

右端の少し上側が膨らんで見えます。、中心からの距離を 10 倍に誇張した結果が次の図です。

シミュレーショ



円形走路での仮想走行結果（偏差を 10 倍に拡大）

走行開始点（コースの右端）から真上に走行を始めた直後だけ、偏差が大きくなっていることが分かります。これは PID 制御の積分項が初期値 0 から始ま

っているためでした。積分時間を過ぎたあたりから安定化し、その後は何周回っても良く追従できています。直線走路の場合でも、最初の 10 秒間は偏差が大きいことが分かります。自動走行は、直線部分から始めるのが良さそうです。

この不安定な時期にセンサがガイドラインを外れると、走路を見失ったとして停車してしまいます。しばらく（制御周期の数倍）の間、直前の偏差を使って制御を続けると、元に戻る可能性があるので、この対策を盛り込むことにしました。

モジュール検証

ライントレーシングモジュールの検証スタブは、以上のシミュレーション走行で使いました。

test_line_trace.py
（プロジェクトファイルを参照）

6.4 コマンド受信処理

コマンド受信処理 ap_manager は、HMI プロセスやブラウザなどからのコマンドを受けて、自律走行プログラム群を制御します。

インクルードファイル

自律走行プログラムに共通する定義は、既に drive_program.h として定義しました。

プログラムファイル

コマンド受信処理 ap_manager の原型は、各自律走行プログラムの検証スタブにあります。これに複数の自律走行プログラムを処理する機能を追加することで実現できます。

まず、各自律走行プログラムを初期化し、そのオブジェクトを modules という辞書に登録します。コマンドを受信したら、この辞書から必要な操作を呼び出します。プログラム実行の開始（start）あるいは終了（terminate）を行ったときは、現在実行中のプログラム名を ap_report()という操作で上位（ルート）に知らせます。

上位で定周期処理を行わない（仮想走行）ときは、次の 2 つのコマンドを追加で受け付けます。この機能は各自律走行プログラムの検証スタブにも組み込まれています。

- Control コマンドでは、定周期処理（ライントレーシングまたは運転制御）を指定回数だけ行います。
- Callback コマンドでは、driver.py が自律走行プログラムをコールバックするのを模擬します。

ap_manager.py

```
/* ap_manager.py 自律走行プログラム管理
  初版：2022/2/12 Chuji
  最新版： 2024/2/23 LCD への表示コマンド処理を止めた
Class: application_manager
属性：
  pi     PIGPIO オブジェクト
  current_app   現在実行中の自律走行プログラム
  modules   自律走行プログラム群（内部使用）
操作：
  command   上位から与えられたコマンドを解釈・実行する
  control   制御周期毎の処理
*/
#include "include/ap_manager.h"
#include "include/line_trace.h"
#include "include/driver.h"
#include "include/use-redis.h"
#include "include/use-sys.h"
/* 自律走行プログラム群のインクルード */
#include "web_drive.py"
#include "block_drive.py"
#include "scratch_drive.py"
#include "line_trace.py"
class application_manager:
  def __init__(self, ap_change, sensor, driver,
pi, fifo):
    self.ap_report = ap_change  /* 実行中プログラムの
報告先 */
    self.driver = driver
    self.current_app = NO_CONTROLLER  /* 実行中の自
律走行プログラム */
    /* 自律走行プログラムのリストを辞書として作成する */
    self.__modules = {}
    self.__modules[WEB_CONTROLLER] =
web_driver(WEB_CONTROLLER, sensor, driver, fifo)
    self.__modules[BLOCK_CONTROLLER] =
block_driver(BLOCK_CONTROLLER, sensor, driver,
fifo)
    self.__modules[SCRATCH_CONTROLLER] =
scratch_driver(SCRATCH_CONTROLLER, sensor, driver,
fifo)
    self.__modules[TRACE_CONTROLLER] =
line_tracer(TRACE_CONTROLLER, sensor, driver,
fifo)
  /* HMI 操作機能から AP_Manager へコマンドを与える */
  def command(self, cmd):
#ifdef SHOW_APM_COMMAND
    print('AP Manager received a command: ', cmd)
#endif
    /* 指定した自律走行プログラムを走らせる */
    if APM_DIRECTIVE(cmd) == APM_RUN:
      if APM_PROGRAM(cmd) in CONTROLLERS:
        self.current_app = APM_PROGRAM(cmd)
        self.__modules[self.current_app].start()
        /* 実行中の自律走行プログラムを報告する */
        self.ap_report(self.current_app)
    /* 実行中の自律走行プログラムを一時休止させる */
    elif APM_DIRECTIVE(cmd) == APM_PAUSE:
      if self.current_app != NO_CONTROLLER:
        self.__modules[self.current_app].pause()
    /* 休止中の自律走行プログラムを再開する */
    elif APM_DIRECTIVE(cmd) == APM_RESUME:
      if self.current_app != NO_CONTROLLER:
        self.__modules[self.current_app].resume()
    /* 実行中の自律走行プログラムを終了させる */
    elif APM_DIRECTIVE(cmd) == APM_TERMINATE:
      if self.current_app != NO_CONTROLLER:

self.__modules[self.current_app].terminate()
        self.current_app = NO_CONTROLLER
        /* 自律走行プログラムの終了を報告する */
        self.ap_report(self.current_app)
    /* 実行中の自律走行プログラムにコマンドを送る */
    elif APM_DIRECTIVE(cmd) == APM_COMMAND:
      if self.current_app != NO_CONTROLLER:

self.__modules[self.current_app].command(AP_CMD(cm
d))
    /* Web 画面上での速度/操舵量の表示を更新する */
```

```
    elif APM_DIRECTIVE(cmd) == APM_UPDATE:
      /* 実行中の自律走行プログラムに更新を依頼する */
      if self.current_app != NO_CONTROLLER:
        self.__modules[self.current_app].update()
    /* 自律走行車プロセスを終了する */
    elif APM_DIRECTIVE(cmd) == APM_EXIT:
      sys.exit()
/* シミュレーション用コマンド処理 -- 実行中プログラムのチェックは省略する */
#ifdef SIMULATED_DRIVE
    /* 回数を指定して AP の周期処理を実行する */
    elif APM_DIRECTIVE(cmd) == APM_CONTROL:
      /* 指定回数だけ実行する */
      for i in range(int(APM_PARM(cmd))):
        self.__modules[self.current_app].control()
    /* Driver からのコールバック処理をシミュレートする */
    elif APM_DIRECTIVE(cmd) == APM_CALLBACK:
      /* 指定されたパラメータ文字列を整数として渡す */

self.__modules[self.current_app].stopped(self.stoi(APM_PARM(cmd)))
  /* テキストを整数に変換する */
  def stoi(self, x):
    if x[:1] == '0b':
      return int(x[2:], 2)
    elif x[:1] == '0x':
      return int(x[2:], 16)
    else:
      return int(x)
#endif  /* ここまでは SIMULATED_DRIVE が#define されたときのみ実行される */
  /* システムタイマーから自律走行プログラムの実行タイミングを与える */
  /* 今のところライントレーシング以外の用途はない */
  def control(self):
#ifdef STEP_TUNING
    if (self.current_app == TRACE_CONTROLLER) or (self.current_app == WEB_CONTROLLER):
#else
    if self.current_app == TRACE_CONTROLLER:
#endif
      self.__modules[self.current_app].control()
```

モジュール検証

このモジュールの検証は、各自律走行プログラムを呼び出す（操作毎に何が呼ばれたか表示する）ことが分かれば充分なので、検証プログラムでは AP_STUB を#define しています。コマンドを与えて、各自律走行プログラムが応答を返すのを確認すれば、検証は終了です。

test_ap_manager.py
（プロジェクトファイルを参照）

6.5　自律走行車プロセスルート

自律走行車プロセスルートは、定周期割り込み制御しながら、必要な処理を実行していきます。また、Redis FIFO からコマンドを取り出して、コマンド受信処理に渡します。

直下のデータ変換モジュール

自動走行車ルートが使う、直下のモジュールを説明しておきます。HMI プロセスが LCD に表示できるよう、障害物センサ、光センサ（走路と崖)、電源電圧センサの測定値をテキストに変換するためのモジュール hmi_format.py を用意しました。

hmi_format.py
（プロジェクトファイルを参照）

このモジュールの検証プログラムは、走路センサと崖センサのデータが文字表現できていることを確認します。

test_hmi_format.py
（プロジェクトファイルを参照）

インクルードファイル

インクルードファイル autonomous_vehicle.h では、システム割り込みの周期（2 種類×3 条件）を定義しています。

走行方法	高速処理	低速処理	実行条件
実走行	100ms	500ms	#ifdef BCM2835
シミュレーション	1s	2s	#ifdef SIMULATED_DRIVE
センサ動作確認	10s	20s	上記のどちらも定義しない

システム割り込み周期

ファイルの掲載は省略します。

autonomous_vehicle.h
（プロジェクトファイルを参照）

プログラムファイル

自律走行車ルートは、システムタイマーオブジェクトを生成し、システムに定周期割り込みを要求します。どれかの自律走行プログラムが走っているときの割り込み周期は 100ms（高速処理)、何も走っていないときは 500ms（低速処理）に設定します。

Redis FIFO のブロッキング読み出しと、pigpio 呼び出しは干渉するため、割込み処理 SlowISR と FastISR では変数 interrupt を設定するだけにし、バックグランド処理でこの変数を調べるようにしました。Redis FIFO の読み出しは非ブロッキング型にして、コマンドが読み出せたときだけ処理するようにしています。

高速処理（走行中）では、安全センサを読み込んで運転制御（driver）に渡し、緊急停止の要否を決めさせます。また、

低速処理（停車中）では、HMI に表示用データを（Redis を介して）送ります。

Redis FIFO からコマンドを受け取ったら、コマンド受信処理（ap_mamager）に渡して、自律走行プロ

グラムを実行させます。速度と操舵量は各自律走行プログラムが更新します。

autonomous_vehicle.py（抜粋）

```
/* autonomous_vehicle.py 自律走行車プロセス
  初版：2020.9.11 Chuji
  最新版：2021/7/27 --- システム全体との関係を整理
オブジェクト部:
Class: system_timer  定周期処理
属性：
  pi    pigpio オブジェクト
  fifo    REDIS FIFO オブジェクト
  driver          運転制御部オブジェクト
  ap_manager  コマンド受信処理オブジェクト
  obstacle  障害物センサオブジェクト
  sensors  反射光式センサオブジェクト
  supply  電源電圧監部オブジェクト
  ap     現在起動されている自律走行プログラム
操作：
  SlowISR     割込み処理(500ms)
  FastISR     割込み処理(100ms)
  SlowPeriodicOps  定周期処理(500ms)
  periodicOps     定周期処理(100ms)
  command     受信したコマンドを ap_manager に渡す
  ap_change     運転している自律走行プログラムを受け取る
下位操作（内部から使用する）：
  set_system_interrupt  システムに定周期割り込みを要求する
  battery_alarm     電池電圧の低下時に HMI にシャットダウンを要求する
システム本体部：
  下位オブジェクトの生成とコマンド待ち
*/
/* 定義ファイルの読み込み */
（一部省略）
/* 割り込み処理中を示すパルスの発生処理 */
#define MONITOR_ISR  self.__pi.GPIO_MODE(GPIO_ISR, GPIO_OUT)
#define START_ISR  self.__pi.GPIO_WRITE(GPIO_ISR, GPIO_HIGH)
#define END_ISR    self.__pi.GPIO_WRITE(GPIO_ISR, GPIO_LOW)
#endif
/* 時間測定用 */
#ifdef ISR_THROUGHPUT
from time import perf_counter as at_this_moment
#define MICROSEC        1000000
#endif
/* 下位モジュールの組み込み */
（一部省略）
/* 定周期処理オブジェクト */
class system_timer:
  def __init__(self, fifo):
    self.__fifo = fifo
    /* 下位モジュールの準備 */
  （一部省略）
    /* システムタイマーを起動するので、これ以降はタイミングに注意 */
    self.set_system_interrupt()
#ifdef DEBUG
    print('*** Timer interrupt services are initialized ***', TO_ERROR)
#endif
  /* 定周期割り込みを指定する */
  def set_system_interrupt(self):
    if self.__ap == NO_CONTROLLER:
      /* 自律走行プログラムは起動されていない時はゆっくり */
      Disable_iTimer_Interrupt
      self.interrupt = NO_INTERRUPT
      Register_iTimer_Handler(self.SlowISR)
      Enable_iTimer_Interrupt(SYSTEM_MONITOR)
    else:  /* どれかの自律走行プログラムが起動されている時は早く */
      Disable_iTimer_Interrupt
      self.interrupt = NO_INTERRUPT
      Register_iTimer_Handler(self.FastISR)
      Enable_iTimer_Interrupt(SYSTEM_PERIOD)
  /* システムタイマーの割込み処理（1）長い周期 */
  def SlowISR(self, signum, frame):
    self.interrupt = SLOW_INTERRUPT
  /* システムタイマーの割込み処理（2）短い周期 */
  def FastISR(self, signum, frame):
    self.interrupt = FAST_INTERRUPT
  /* システムタイマーによる定周期処理（1）遅い処理---LCD表示更新 */
  def SlowPeriodicOps(self):
#ifdef SHOW_ISR
    /* 割り込み処理開始を表示する */
    START_ISR
#endif
#ifdef DEBUG
    print('!!! Slow Periodic System Interrupt!', TO_ERROR)
#endif
    /* HMI に表示するセンサ出力の更新 */
    /* 電池電圧 */
    battery = READ_BATTERY
    SHOW_DATA(REDIS_FOR_BATTERY, format_voltage(battery))
    /* 電池電圧が低下していたらシャットダウンを要求する */
    if BATTERY_LOW(battery):
      self.__battery_low += 1
      if self.__battery_low >= BATTERY_BAD_COUNT:
        self.__battery_alarm()
    else:
      self.__battery_low = BATTERY_OK
    /* 障害物センサ */
    distance, echo = READ_OBSTACLE  /* 前回の結果を取り出したら */
    RETRIGGER_US       /* 次回のために超音波を発射する */
    SHOW_DATA(REDIS_FOR_OBSTACLE, self.__obstacle.string(echo))
    /* 崖・走路センサ */
    cliff = READ_CLIFF
    line_sensors = READ_LINE
    SHOW_DATA(REDIS_FOR_SENSORS, format_sensor(line_sensors, cliff))
#ifdef SHOW_ISR
    END_ISR     /* 割り込み処理終了を表示する */
#endif
  /* システムタイマーによる定周期処理（2）走行制御 */
  def periodicOps(self):
#ifdef SHOW_ISR
    START_ISR   /* 割り込み処理開始を表示する */
#endif
#ifdef DEBUG
    print('!!! Fast Periodic System Interrupt!', TO_ERROR)
#endif
#ifdef ISR_THROUGHPUT
    start_time = at_this_moment()
#endif
    /* 非常停止要件の検出を開始する */
    emergency = NO_ALARM
  /* GPIO を使わないときはアラームをすべてクリアすること */
    /* 前方障害物があったら非常停止する
      （超音波センサの測定結果を読み取る） */
    distance, echo = READ_OBSTACLE
    if echo == US_DANGER:
      SET_OBSTACLE_ALARM(emergency)
    /* 超音波センサを再度駆動する */
#ifndef SIMULATED_DRIVE
    RETRIGGER_US
#endif
    /* 崖を検出したら非常停止する */
    /* ただしライントレーシング走行時はガイドラインを崖と判断
       する可能性があるので、この検出は行わない */
    if self.__ap != TRACE_CONTROLLER:
      cliff = READ_CLIFF
      if cliff == CLIFF_DANGER:
        SET_CLIFF_ALARM(emergency)
    /* 電池電圧が低下したら非常停止する */
    battery = READ_BATTERY
    if BATTERY_LOW(battery):
      self.__battery_low += 1
      if self.__battery_low >= BATTERY_BAD_COUNT:
        SET_BATTERY_ALARM(emergency)
        /* 同時にシャットダウンを要求する */
        self.__battery_alarm()
```

```
    else:
      self.__battery_low = BATTERY_OK
    /* 実行中の運転プログラムに制御を実行させる */
    self.__ap_manager.control()
    /* 走行制御プログラムに非常停止を知らせ、走行を制御させる */
    self.__driver.control(emergency)
#ifdef SHOW_ISR
    END_ISR    /* 割り込み処理終了を表示する */
#endif
#ifdef ISR_THROUGHPUT
    end_time = at_this_moment()
    /* 時間の単位は秒なので、ms 単位で表示する */
    isr_time = (end_time - start_time) * 1000
    print('ISR service time = ', isr_time, 'ms.', TO_ERROR)
#endif
  /* 受信したコマンドをコマンド解釈部に渡す */
  def command(self, cmd):
    self.__ap_manager.command(cmd)
  /* AP Manager からのコールバック関数。実行中の自律走行プログラムを知らせる */
  def ap_change(self, ap):
    While_iTimer_Interrupt_disabled:
      self.__ap = ap
      self.set_system_interrupt()
  /* 電池電圧が低下した時には HMI にシャットダウンを要求する */
  /*   注：SIMULATED_DRIVE 時には電圧異常は起こらない */
  def __battery_alarm(self):
    self.__fifo.rpush(REDIS_TO_HMI, HMI_BATTERY_LOW)
    /* LCD にメッセージが表示されるまで待つ */
    sleep(WAIT_BEFORE_SHUTDOWN)
    SYS_EXIT(SYS_NO_ERROR)
/* システム 起動 */
fifo = open_FIFO()  /* Redis データベースを使えるようにする */
vehicle = system_timer(fifo)
try:  /* バックグランド処理 */
  while True:
    /* タイマー割込みが起こっていたら、処理手続きを実行する */
    if vehicle.interrupt == FAST_INTERRUPT:
      vehicle.interrupt = NO_INTERRUPT
      vehicle.periodicOps()
    elif vehicle.interrupt == SLOW_INTERRUPT:
      vehicle.interrupt = NO_INTERRUPT
      vehicle.SlowPeriodicOps()
    /* FIFO にコマンドが入っていたら解釈部に渡す */
    cmd = fifo.lpop(REDIS_TO_COMMAND)
    if cmd != None:
      vehicle.command(cmd.decode('utf-8'))
    sleep(REPEAT_BACKGROUND)
except KeyboardInterrupt:
  /* ここだけは必ずプロセスを終了させる */
  sys.exit(SYS_NO_ERROR)
```

モジュール検証

ルートの検証に特別なスタブは要りません。。最初に SIMULATED_DRIVE を#define して実行し、割り込み周期と処理が切り替わることを確認します。2 秒ごとに表示が更新されたというテキストが出力されます。別のターミナルから pusher.py を実行し、コマンド RWeb を与えると、走行を始めたというメッセージが出力され、1 秒ごとに車両の位置が更新されます。コマンド'CF'を与えると前進を始めます。Web ドライブの動作が確認できれば充分です。

```
$ cpp -DSIMULATED_DRIVE autonomous_vehicle.py
>tmpAv.py ;python tmpAv.py
US obstacle sensor is initialized
US pulse is generated
Optical sensors are initialized
I2C channel to ADC is opened
0.0,  0.0,  0.0,  0.0
Data < Hello World! > is sent to Buffer( Dummy )
Data <  9.0 V   > is sent to Buffer( Battery )
Data <  Far     > is sent to Buffer( Obstacle )
Data < WWBWW Ok > is sent to Buffer( Sensors )
Data <  9.0 V   > is sent to Buffer( Battery )
Data <  Far     > is sent to Buffer( Obstacle )
Data < WWBWW Ok > is sent to Buffer( Sensors )
Data <  9.0 V   > is sent to Buffer( Battery )
Data <  Far     > is sent to Buffer( Obstacle )
Data < WWBWW Ok > is sent to Buffer( Sensors )
0.1,  0.0,  0.0,  0.0
0.2,  0.0,  0.0,  0.0
0.3,  0.0,  0.0,  0.0
0.4,  0.0,  0.0,  0.0
0.5,  0.0,  0.0,  0.0
0.6,  0.0,  0.0,  0.0
0.7,  0.0,  0.0,  0.0
0.8,  0.0,  1.2,  0.0
0.9,  0.0,  2.4,  0.0
1.0,  0.0,  3.6,  0.0
1.1,  0.0,  5.1,  0.0
1.2,  0.0,  6.6,  0.0
1.3,  0.0,  8.1,  0.0
1.4,  0.0,  9.6,  0.0
1.5,  0.0,  11.1,  0.0
1.6,  0.0,  12.6,  0.0
```

```
$ python pusher.py Command
RWeb
Pushed: RWeb
CF
Pushed: CF
```

次にセンサを読み取って必要な処理を行えることを確認します。何も#define しないで実行すると、センサの入力を求めながらゆっくりと動作します。電圧として 10（V)、距離として 1000（mm）を入力していきます。上と同じコマンドを与え、周期が変化すること、異常値（電圧 9V 未満、あるいは距離 100mm 未満）を入力すれば、停車することを確認すればこの検証も終了です（入力を与えるのが遅いと、エラーを起こして止まってしまいますが、異常動作ではありません)。

ハードウエア検証

ルートでは GPIO を使いません。使うのは Linux のシステム割り込みと Redis だけなので、ハードウエア検証は必要ないように思えます。

しかし、100ms の周期内に割り込み処理が完了するかという検証は、すべてのハードウエアが動作しているこの時点で行う必要があります。そのため、割り込み処理の開始時点から終了時点までの時間を測定しておきます（実際の割り込み処理 FastISR ではなく、100ms 毎に実行される periodicOps にかかる時間)。ISR_THROUGHPUT と BCM2835 を#define して実行することで測定します。このときの割り込み周期は 2 秒と遅くとっているので、処理に予想外の時間がかかっても、これ以内には確実に終わります。

```
$ cpp -DISR_THROUGHPUT -DBCM2835
autonomous_vehicle.py | python
```

```
ISR service time =  9.535152999887941 ms.
ISR service time =  9.268147999591747 ms.
ISR service time =  9.444151000025158 ms.
:
```

```
$ python pusher.py Command
RWeb
Pushed: RWeb
```

自律走行プログラムが走っているときだけ処理時間が表示されます。実行するプログラムを変えながら測定した結果は次のとおりです。余裕をもって処理できていることが分かりました。

自律走行プログラム	処理時間
Web ドライブ、Block ドライブ、Scratch ドライブ	10±1ms
ライントレーシング	25±5ms

100ms 定周期処理にかかる時間

6.6 走行検証

仮想走行

すでに前節で、自律走行車全体の仮想走行を検証してあります。このとき、本来は 100ms で実行される処理の周期が 1 秒に延長されているので、コンソール上で走行状態を確認することができます。Redis FIFO から（あるいは Web ブラウザから）コマンドを与えて、走行させてみてください。

安全機構の検証

それでは実際に走行させてみます。最初に安全装置の確認を行うため、車輪を浮かした状態で動作させます。BCM2835 を#define して走行させ（車輪が空回りをしている）状態で以下の動作を確認します。

1. 車両の前に本などで壁を作り、10cm 以内に近づける
2. 崖センサの下に黒い紙を置くか、机の端からはみ出させる

車輪の回転が止まり、ハザードランプが点滅するのを確認します。危険状態を解消させると、ハザードランプが消えます（停車しているのでブレーキランプが点灯するべきですが、消えたままです。実際には危険回避走行をしているはずなので、許容することにしていました)。ハザードランプが消えるか、回避指令を与えると、走行できる状態になります。

走行テスト

やっと走行テストに移れます。補助電源（AC アダプタ）を接続していたら、電池動作に切り替えます。幅 1m 程度の周囲が開けている場所で走行（Web ドライブ）させてみてください。まだ直進補正前なので、思った方向に進まないかもしれません。直進しようとすること、ハンドル操作で曲がること、その場で回頭できることを確認します。

ちょっと怖いけれど、壁や崖に向かって走行して、ちゃんと停まることも確認できます。

直進補正

左右の駆動系は、モーターの特性（印加電圧／回転数）や車輪の寸法ばらつきなどが原因で、同じ PWM を印加しても速度が同じになりません。このままでは直進しないので、補正が必要です。車体をまっすぐにして走行させ、移動距離と左右へのブレから補正値を決めることにします。方法は付録で説明してあります。

試行	L (cm)	x (cm)	g	$1/g$
1	102	17.5	0.9563	1.046
2	96	17	0.9520	1.050
3	96	16	0.9549	1.047
4	91	16.5	0.9482	1.055
5	76	13	0.9415	1.062

移動距離とブレから直進補正値を求める

結果は上表のとおりで、motor_drive.h で定義している DRIVE_BALANCE の値（$1/g$：初期値は 1）を 1.052 に変更した結果、「ほぼ直進（路面の状態による)」するようになりました。

6.7 総合検証

ここまでは HMI プロセスと Web サーバーなしでも検証できます。それらの個別検証がまだだったら、先にそちらを済ませてください。それが済んだら、いよいよ自律走行車システムとして、全部のプロセスを同時に稼働させることができます。

次章の最後にある、三者をすべて起動するシェルスクリプト start_raspbuggy を実行してください。自律走行プログラムをそれぞれ実行し、Web ブラウザから走行を確認します。表示設定を#define するときは、cpp の-D オプションを使います。

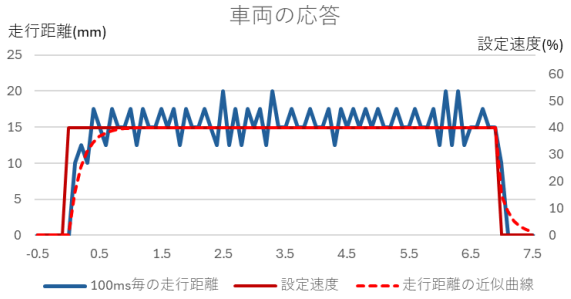
その場で円走行

Web ドライブでハンドルを最大まで切ると、その場で円を描くように走行します。速度 40%時の半径は 50cm 程度でした。

車体の応答

車体には慣性があるので、モーターを回転させても、すぐには速度が変わりません。自律走行車プロセスを走らせるときに、STEP_TUNING を#define すると、Web ドライブの実行中は（100ms 毎に）

走行計の距離積算値を標準エラー出力に表示するようになります（ほんらい走行計を使うのはブロックドライブと Scratch ドライブだけなので、干渉はしない）。2>でファイルにリダイレクトし、100ms 毎の増分を単位時間ごとの走行距離として、Excel でグラフ化したものが下図です。安定時の速度（設定値は 40%）は 15cm/s（15mm/100ms）でした。



車両の応答性評価結果

走行計の分解能（ロータリーエンコーダの 1 目分）による 2.5mm のばらつきがありますが、次のことが分かります。

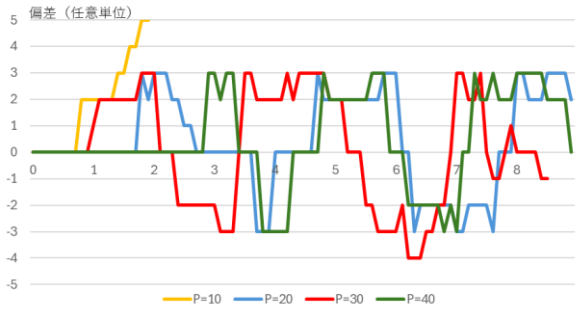
1. 100%駆動時の速度は 38cm/s（15cm/40%）で、初期検討時の見積り（33cm/s）より早め
2. モーターを駆動しても 100ms 程度のむだ時間（速度が変化しない）と、200ms 弱の応答時間（定常速度の 63%になるまでの時間）がある。停車するときはむだ時間が観測されない（図の点線は、上記のむだ時間と応答時間を使った近似）

特に 2.は、車両の応答が仮想走行と違ってくることを意味します（応答を組み込んだシミュレーションを設計すればいいのですが、そのためだけに面倒なことをする気はありません）。しかし、ライントレーシングでは、制御性に直接影響してしまいますしてしまいます。

ライントレーシング

ライントレーシングの動作を確認するため、制御のチューニングを実機で行ってみます。車両の位置や向きを測定するのは難しいので、ガイドラインの検出具合だけで行う必要があります。

自律走行車プロセスを起動するとき、SHOW_PID を#define すると、走路センサの表示値である偏差（signal_conditioner.py の出力）を標準エラー出力に表示します。比例制御（I=D=0）で、P パラメータを変えながら直線コースを走らせて解析しました。データを次のグラフに示します。



比例制御で直線のライントレーシングを行った結果

走路センサの分解能が低いため、結果はステップ状に変化します。P=10 の場合は、軌道を修正しきれず迷子になってしまいました。P=20 と P=50 の間では、ガイドラインの近くを走行しますが、P が大きくなるにつれて車体の振れが大きくなっていきました。微分項を入れて、抑える必要があります。ジーグラとニコルスの方法でパラメータを決めると、P=30、I=1.25s、D=0.3s となり、これを出発点として、さらにパラメータを調整していきます。ちなみに、仮想走行の場合と比べると、P は 10 倍、I と D は 1/3 になってしまいました。慣性の影響です。

パラメータを調整して、実際に走行した動画を公開しています。最初は屋内で走行させていたのですが、床板の合わせ目を黒線の一部として検出していることが分かり、白いシート上で走らせるようにしました。広い場所が要るので屋外で走行させようとしたのですが、太陽光に含まれる赤外線の影響で、走路センサが誤動作してしまいます。曇天でも状況は改善しないので、室内を片づけて広い走路を作りました。

走路センサの分解能が低いこと、車両が大きめなこと、制御周期が 100ms と長いことなどから、あまり制御性はよくありません。それでも、ヒトが運転するときのような、ゆっくり曲がるコースには追従できました。ハンドルの切れ具合を、そのように設定したのだから、当然と言えば当然の結果です。

走行を楽しむ

あとは楽しむだけです。もちろん、意図どおりに動かないなどの不具合が見つかれば、原因を調べて改善することになります。実際の走行を撮影した動画をプロジェクトページ（https://www.akiyama-tokyo.net/electronics/raspbuggy.html）で公開しています。

6.8 HMI 下位モジュール

次に HMI プロセスのコーディングと検証を行いますが、例によって下位モジュールの説明と検証から始めます。

6.8.1 キースイッチ

インクルードファイル

ハードウエアではキースイッチの状態を、インバータ型シュミット回路を介して読み取っているので、スイッチが ON の時の GPIO 入力は L になります。使いやすくするため、KEY__ON を定義していますが、GPIO と論理が反転しているので気をつけてください。

keys.h

```
/* keys.h キースイッチに関する定義
  初版：2020/9/11 Chuji
  最新版：
*/
#ifndef __KEY_SWITCH
#define __KEY_SWITCH
#include "use-pigpio.h"
/* スイッチ番号の定義 */
#define NEXT_KEY  GPIO_NEXT
#define SELECT_KEY  GPIO_SELECT
#define NO_KEY    0
/* スイッチの状態  */
#define KEY_ON    GPIO_LOW
#define KEY_OFF    GPIO_HIGH
#endif
```

プログラムファイル

キースイッチは割り込みではなく、一定時間（0.3 秒）ごとのポーリングで読み込みます。入力状態が変化したときにスイッチが押されたと判断します。こうすれば、操作時に接点が短時間で繋がったり切れたりする現象（チャタリングといいます）の影響を受けません。押してすぐ放す「ちょい押し」は見逃すことがあります。「長押し」は無視します。

keys.py

```
/* keys.py キースイッチ操作ハンドラ
  初版：2020/9/4 Chuji
  最新版： 2022/5/19 --- ローカル変数を見えなくした
           2021/1/24 --- 簡略化・チャタリング処理不要
           2020/9/11 --- 割り込みからポーリングに変更
Class: key_switch
属性：
  pi     PIGPIO オブジェクト
  prev_next   前回の NEXT キーの押下状態
  prev_select   前回の SELECT キーの押下状態
操作：
  key_status   キーが押されているか調べ、状態をタプルとして返す(一操作のみ)
  status_next   NEXT キーが押されているか調べる
  status_select   SELECT キーが押されているか調べる
  next_key   NEXT キー接点の状態を読み取る（実体は内部で使用するマクロ）
  select_key   SELECT キー接点の状態を読み取る（実体は内部で使用するマクロ）
  raw_key_status   キーの状態を調べ、状態をタプルとして返す
*/
#include "include/keys.h"
class key_switch:
  def __init__(self, pi):
    self.__pi = pi
    /* ソフトウエアによるフィルタのため、過去の結果を保存 */
    self.__prev_next = KEY_OFF
    self.__prev_select = KEY_OFF
#ifndef HANDLER_STUB
    self.__pi.GPIO_MODE(NEXT_KEY, GPIO_IN)
    self.__pi.GPIO_MODE(SELECT_KEY, GPIO_IN)
#endif
  /* ポーリング方式のキー入力読み取り */
#ifdef HANDLER_STUB
#define next_key()  KEY_OFF
#define select_key()  KEY_OFF
#else
#define  next_key()  self.__pi.GPIO_READ(NEXT_KEY)
#define select_key()
self.__pi.GPIO_READ(SELECT_KEY)
#endif
  /* NEXT キーが押されているか調べる */
  def status_next(self):
    /* 現在のキー接点の状態を読み取る */
    next_in = next_key()
    /* OFF から ON になったときだけ知らせる */
    if (next_in == KEY_ON) and (self.__prev_next
== KEY_OFF):
      next_out = KEY_ON
    else:
      next_out = KEY_OFF
    /* 次回に備えて現在の状態を記憶する */
    self.__prev_next = next_in
    return next_out
  /* SELECT キーが押されているか調べる */
  def status_select(self):
    /* 現在のキー接点の状態を読み取る */
    select_in = select_key()
    /* OFF から ON になったときだけ知らせる */
    if (select_in == KEY_ON) and
(self.__prev_select == KEY_OFF):
      select_out = KEY_ON
    else:
      select_out = KEY_OFF
    /* 次回に備えて現在の状態を記憶する */
    self.__prev_select = select_in
    return select_out
  def key_status(self):
    /* キーが押されているか調べる */
    next_out = self.status_next()
    select_out = self.status_select()
    return next_out, select_out
  def raw_key_status(self):
    /* 現在のキーの状態を返す */
    return next_key(), select_key()
```

モジュール検証

検証スタブ test_keys.py では約 0.5 秒ごとにキースイッチの状態を読み取り、押していないときは"----"を、押したときにはキー名を表示します。

test_keys.py

（プロジェクトファイルを参照）

最初に BCM2835 を#define せずに実行します。キーボードから入力を受け付けるため、cpp の出力をいったんファイル tmp.py に収納してから、python で実行します。

```
$ cpp test_keys.py |python cleanfile.py >tmp.py;
python tmp.py
```

```
Mode at GPIO#  19  is set to:  0
Mode at GPIO#  6  is set to:  0
GPIO# 19 input? 1
GPIO# 6 input? 1
  ----    ------
GPIO# 19 input? 0
GPIO# 6 input? 1
  NEXT    ------
GPIO# 19 input?
:
```

GPIO ポート入力は 1 が OFF です。0 にしたときに一度だけキースイッチの押下を検出していることを確認します。

ハードウエア検証

次に Raspberry Pi ZERO 上で BCM2835 を#define してから実行します。押したキースイッチにより NEXT または SELECT が表示されます。うまく同時に押せば、両方とも表示されます。

```
$ cpp -DBCM2835 test_keys.py |python cleanfile.py |
python
  ----    ------
  NEXT    ------
  ----    ------
:
```

6.8.2 LCD

LCD は最初のプロジェクト（温度コントローラ）から使っているシリーズのハードウエアなので、説明は簡単にします。

インクルードファイル

インクルードファイルは同一シリーズ LCD に共通にしています。自律走行車では 8 文字 LCD を使用するため LCD8 を定義します。

lcd.h

（プロジェクトファイルを参照）

プログラムファイル

LCD ハンドラは、以前からのモジュールがそのまま使えますが、I2C バスへのアクセスに pigpio を使ったものに変更しました。

lcd.py

（プロジェクトファイルを参照）

モジュール検証

モジュールの検証は以前のプロジェクトで済んでいるので、実際に書き込みを行う pigpio 部の検証だけにします。I2C バスに送り出すデータが表示されるので、LCD の仕様書どおりであることを確認すればじゅうぶんです。

test_lcd.py

掲載を省略

ハードウエア検証

BCM2835 を#define してから上の検証プログラムを実行すれば、LCD の一行目に Hello!、二行目に World!と表示されます

6.8.3 LCD 表示イメージ生成

このモジュールも以前のプロジェクトで使っていたものです。その時には、LCD 上の任意の位置の表示内容だけを変更できるようにしていました。自律走行車では、この機能は使いませんが、モジュールでは残しています。

インクルードファイル

8 文字 LCD を使っているので、表示するテキストをこのファイルで定義しています。WiFi 環境に関する定義は削除しています。

display.h（抜粋）

```
/* display.h
   初版: 24, December, 2014 Chuji
   最新版: 2020/9/8 --- バギー用に変更
*/
/* LCD 表示イメージの生成に関する情報 */
#ifndef __DISPLAY
#define __DISPLAY
/* 表示フィールドの定義 --- 2 行 x 8 文字を想定している */
/*   フィールド:   フィールドの割り付け
        +-----+          +----------+
行 #0   | #0 |           |  Item    |
        +-----+          +----------+
行 #1   | #1 |           |  Value   |
        +-----+          +----------+
*/
/* フィールドの定義 */
#define LCD8   /* 8 文字/行を使うという宣言 */
（後略）
```

プログラムファイル

プログラムでは、8 文字 LCD に特化した簡易型インターフェースと、接続してきた Web ブラウザに現在の表示イメージを送る機能を追加しました。

display.py

```
/* display.py  LCD 表示イメージの構築
   初版： 24 December 2014 Chuji
   最新版： 2022/5/16 --- ローカル変数を見えなくした
            2021/2/25 --- Buggy 用に改造
変更履歴
   23 January 2019 - LCD ドライバから独立
   8 June 2017 - リモート操作機能の追加
```

```
Class: display_image
属性:
  lcd:  LCD ドライバ
  lines:  表示イメージ（シミュレーション用とリモート操作用）
  display_fifo: イメージをリモート表示に送る Redis FIFO
操作:
  fill_field:  表示イメージを生成して LCD に送る（2 行モデル）
  show:  fill_field の簡易版マクロ（行とテキストのみを指定する）
  update: 現在の表示イメージを Web サーバーに再送する
*/
#include "include/use-redis.h"
#include "include/use-sys.h"
#include "include/display.h"
#include "include/my_round.h"
#include "ftos.py"
#include "lcd.py"
/* 表示イメージ生成オブジェクト */
class display_image:
  def __init__(self, pi, fifo):
    self.__lcd = lcd_device(pi)
    /* 表示画面イメージの初期化 */
    self.__lines = [DISP_ALL_SPACES,
DISP_ALL_SPACES]
    /* 表示行を Redis FIFO に結び付ける辞書 */
    self.__fields = (REDIS_TO_LCD_TOP,
REDIS_TO_LCD_BOTTOM)
    self.__display_fifo = fifo
/* テキストを表示する操作
       fill_field(row: 表示する行（0 から 1）
                  pos: 表示を始める文字位置（左端から）
                  s: 表示するテキスト
                  len: 表示する文字数 */

  def fill_field(self, row, pos, s, length):
    /* 行の範囲チェック */
    if (row < DISP_MENU_ROW) or (row >
DISP_VALUE_ROW):
      return
    /* 開始位置のチェック */
    if (pos < 0) or (pos > DISP_STRING_LEN):
      return
    /* 表示範囲に収まりきらないときは後ろを切り捨てる */
    if pos + length > DISP_STRING_LEN:
      length = DISP_STRING_LEN - pos
    /* 文字列が表示文字数に満たないときは後ろに空白を付け加
える */
    while(len(s) < length):
      s = s + DISP_SPACE
    /* シミュレーションと Web 表示用のバッファに書き込む
*/
    self.__lines[row] = self.__lines[row][:pos] +
s[0:length] + self.__lines[row][pos + length:]
    /* Web に表示イメージを渡す */
    self.__display_fifo.rpush(self.__fields[row],
self.__lines[row])
    /* LCD に文字を表示する */
#ifndef HMI_SIMULATION
    self.__lcd.put_string(row, pos, s, length)
#else
    print(DISP_SEPARATOR, TO_ERROR)

print("|"+self.__lines[DISP_NAME_ROW][:DISP_STRING
_LEN]+"|", TO_ERROR)
    print(DISP_SEPARATOR, TO_ERROR)

print("|"+self.__lines[DISP_VALUE_ROW][:DISP_STRIN
G_LEN]+"|", TO_ERROR)
    print(DISP_SEPARATOR, TO_ERROR)
    print(' ', TO_ERROR)

#endif
/* 一行に指定テキストを埋め込むマクロ */
#define show(x, y)  fill_field(x, DISP_LEFT_MOST,
y, DISP_STRING_LEN)
/* 現在の表示イメージを Web サーバーに再送する --- 新しく接
続してきた Web ブラウザに現在の状態を伝える） */
  def update(self):
    self.__display_fifo.rpush(REDIS_TO_LCD_TOP,
self.__lines[DISP_MENU_ROW])
    self.__display_fifo.rpush(REDIS_TO_LCD_BOTTOM,
self.__lines[DISP_VALUE_ROW])
```

モジュール検証

検証プログラムは以前からのものを使います。

test_display.py
（プロジェクトファイルを参照）

6.8.4 WiFi 環境情報取得・設定

自律走行車は、いろいろな場所で走行させたいと思います。そのとき、Web ブラウザから Web サーバーに接続するためには、現在の WiFi 環境（ESSID と IP アドレス）を知る必要があります（ポート番号はいつも同じ）。hostname と iwconfig コマンドで各々を表示できます。ESSID は'wlan0'を含む行のなかで、'ESSID'の先頭から 7 文字先の文字から最終文字'"'の前までを取り出します。

```
$hostname -I
192.168.179.36

$ iwconfig wlan0
wlan0     IEEE 802.11  ESSID:"Wxxxxxxxx"
          :
```

走行場所の WiFi 環境を変更したら、設定ファイルを書き換えます。書き換えには to_xxxx という手順（プロジージャ）を用意しておき、それぞれを実行することにします。例として、to_setagaya（自宅へ移動）を下に示します。コマンド chmod a+x で、このファイルが実行できるようにしておきます。このファイルの実行には sudo が必要なので、HMI プロセスも sudo で起動しておきます（/etc/rc.local に書いておき、OS が立ち上がったときに起動するようにしておけば、自動的に sudo したことになる）。

to_setagaya
#Wifi の設定を世田谷自宅のものに変更する cp -f /etc/network/interfaces.home /etc/network/interfaces cp -f /etc/wpa_supplicant/wpa_supplicant.conf.setagaya /etc/wpa_supplicant/wpa_supplicant.conf cp -f /etc/rc.local.setagaya /etc/rc.local

場所名（使用する WiFi ルーターの名前）と移動手順ファイル名は辞書 WIFI_PROCEDURES に定義しておき、場所が変更されたら実行できるようにしておきます。WiFi 環境がない場所では、自律走行車の Raspberry Pi ZERO がルーターと DHCP（アドレス割り当て）サーバーになる必要があります。この指定は移動先用の/etc/rc.local ファイルに記載し

ておきます。現在地名は専用ファイル WIFI_ENVIRONMENT に書いておきます。

以下の検証中に WiFi 環境が変わってしまっては面倒です。そこで、STAY_ON_WIFI を#define（wifi.h で記述している）したときは、検証用のシェルスクリプト dummy_to_xxx ファイルを実行するようにしました。

dummy_to_setagaya（評価用）

```
echo I am going to Setagaya ...
```

インクルードファイル

インクルードファイルでは、場所名と移動手順ファイル名を定義しておきます。

wifi.h（抜粋）

```
/* wifi.h WiFi 環境の定義
   初版： 2022/4/3 Chuji
   最新版：
*/
#ifndef __WIFI
#define __WIFI
#include "use-sys.h"
/* WiFi 環境ファイル名 */
#define WIFI_ENVIRONMENT  "wifi_environment.txt"
/* WiFi 環境（場所名） */
#define WIFI_SETAGAYA  'setagaya'
#define WIFI_MIURA  'miura   '
#define WIFI_KITAKARU  'kitakaru'
#define WIFI_KAWASAKI  'kawasaki'
#define WIFI_MOBILE  'mobile  '
#define WIFI_MYSELF  'raspbery'
#define DEFAULT_ROUTER  WIFI_SETAGAYA
（後略）
```

プログラムファイル

このモジュールの中心は、使用可能な WiFi 環境を順番に取り出すことと、新しい WiFi 環境を設定することです。後者はこの節の始めに説明しました。

「順番に取り出す」のに適したデータ構造は「循環リスト（リストの最後の要素の次は、最初の要素が取り出される）」ですが、標準ライブラリの queue や deque は大げさ（内部的にリストを書き換えたりすることがある）なので、簡単なモジュールを用意しました。この検証プログラム test_circular.py はプロジェクトファイルを参照してください。

circular.py

```
/* circular.py 循環リスト
   初版： 2022/4/3 Chuji
   最新版：
テキストのリストから順番に要素を取り出す。最後に行き着いたら先頭に戻る。
Class circular_list
属性：
  clist   :循環リスト
  pos   :現在の位置
操作：
  next   :次の要素を返す
  set   :指定した要素の位置を現在の位置にセットする
  find   :指定した要素の位置を返す
        （指定した要素が見つからなければデフォルト位置を使う）
*/
#define INITIAL_POSITION  0
#define DEFAULT_POSITION  INITIAL_POSITION
class circular_list:
  def __init__(self, clist):
    self.clist = clist
    self.pos = INITIAL_POSITION
  def next(self):
    self.pos += 1
    if self.pos >= len(self.clist):
      self.pos = INITIAL_POSITION
    return self.clist[self.pos]
  def find(self, c):
    for i in range(len(self.clist)):
      if c == self.clist[i]:
        return i
    return DEFAULT_POSITION
  def set(self, c):
    self.pos = self.find(c)
```

WiFi モジュール wifi.py は、現在の WiFi 環境を取り出します。IP アドレスは上位と下位に分けて表示できるようにしました。長い ESSID を全部 LCD に表示することはできませんが、最初の 8 文字が分かれば、大抵は問題ありません。

現在地をファイルから取り出し、操作 next_router で次々に候補地を取り出します。操作 set_router では、使用場所を変更したら、移動シェルスクリプトを実行したあと、WiFi と Web サーバーを再起動して（OS のリブートは必要ない）、新しい環境で使えるようにします。

wifi.py

```
/* wifi.py --- WiFi 環境の取得と設定を行う
   初版： 2017/2/20 Chuji
   最新版： 2022/4/3 --- Raspbuggy 用に改造
Class: WiFi_config
属性：
  router   :現在使用している WiFi 環境
  disp_router   :表示されている WiFi 環境
  ssid    :現在使用している WiFi の ESSID
  ip_address   :現在使用している IP アドレス
操作：
  get_router   :現在使用している WiFi 環境（ルーター）を得る
  next_router   :次の WiFi 環境を得る
  set_router   :WiFi 環境を設定する
  get_ssid   :現在使用している WiFi の ESSID を得る
  get_ip_high   :現在使用している IP アドレスの上位 2 バイトを得る
  get_ip_low   :現在使用している IP アドレスの下位 2 バイトを得る
*/
#include "include/wifi.h"
#include "include/use-sys.h"
#include "circular.py"
/* IP アドレスの分解 */
#define IP_SEPARATER  '.'
#define IP_SPLIT(x)  split(IP_SEPARATER)
#define IP_SPACES  '        ' /* テキストを 8 文字以上にする埋め草 */
class WiFi_config:
  def __init__(self):
    self.__router = DEFAULT_ROUTER /* 仮の値 */
    self.__ip_address = SHELL_COMMAND('hostname -I')
    ssid = SHELL_COMMAND('iwconfig wlan0 | grep ESSID')
```

```
    /* テキスト中、ESSID: の後に ssid が表示されている */
    essid = ssid.find('ESSID:')
    self.__ssid = ssid[essid + 7: -1]
    /* 使用する可能性のあるルーターのリスト */
    self.__routers = circular_list(WIFI_ROUTERS)
    /* 現在の WiFi 環境を示すファイルがあれば読み取り、 */
    if os.path.exists(WIFI_ENVIRONMENT):
      f = open(WIFI_ENVIRONMENT, 'r')
      self.__router =  f.read()
#ifdef DEBUG
      print('Router = ', self.__router, TO_ERROR)
#endif
      f.close()
      if self.__router not in WIFI_ROUTERS:
        self.__router = DEFAULT_ROUTER
    /* なければファイルを作成し、デフォルト環境を書き込む
*/
    else:
      self.__router = DEFAULT_ROUTER
      f = open(WIFI_ENVIRONMENT, 'w')
      f.write(self.__router)
      f.close()
    /* 環境リストの位置を指定する */
    self.__routers.set(self.__router)
    self.__disp_router = self.__router
  /* 現在使用している WiFi 環境（ルーター）を得る */
  def get_router(self):
    return self.__disp_router
  /* 次の WiFi 環境を得る */
  def next_router(self):
    self.__disp_router = self.__routers.next()
    return self.__disp_router
  /* WiFi 環境を設定する */
  def set_router(self):
    if self.__router != self.__disp_router:
      self.__router = self.__disp_router
      f = open(WIFI_ENVIRONMENT, 'w')
      f.write(self.__router)
      f.close()
      /* 設定手続きのリストから必要なファイル名を取り出す
*/
      move_procedure =
WIFI_PROCEDURES[self.__disp_router]
#ifdef HANDLER_STUB
      print('WiFi Environment is changed to ',
self.__router, 'by', move_procedure, TO_ERROR)
      print('WiFi Restart!!! ', TO_ERROR)
#else
      /* 設定手続きを実行し、WiFi と Web サーバーを再起動
する */
      SHELL(move_procedure)
      RESTART_WIFI
      RESTART_WEB_SERVER
#endif
  /* 現在使用している WiFi の ESSID を得る */
  def get_ssid(self):
    return self.__ssid
  /* 現在使用している IP アドレスの上位 2 バイトを得る */
  def get_ip_high(self):
    ip =
self.__ip_address.IP_SPLIT(self.__ip_address)
    return ip[0] + IP_SEPARATER + ip[1] +
IP_SPACES
  /* 現在使用している IP アドレスの下位 2 バイトを得る */
  def get_ip_low(self):
    ip =
self.__ip_address.IP_SPLIT(self.__ip_address)
    return ip[2] + IP_SEPARATER + ip[3] +
IP_SPACES
```

モジュール検証

モジュールの検証プログラムでは、現在地をファイルから取り出し、順番に変更していきます。最後には、現在表示している場所名を保存した後、移動手順ファイルを実行して再起動します（実際には表示するだけです)。

test_wifi.py

（プロジェクトファイルを参照）

```
$ cpp  test_wifi.py |python
SSID =  xxxxxxxxxx
IP Address (high) =  192.168
IP Address (low) =  179.36
Current Router =  miura
Next Router =  kitakaru
:
Next Router =  kawasaki
I am going to Kawasaki.
WiFi is being restarted...
Web Server is being restarted.
```

6.9　ヒューマン・マシン・インターフェース

ヒューマン・マシン・インターフェースは、仕様設計を行った時に、コマンドを解釈・実行するステートマシンと定周期処理・コマンド受信処理モジュールに分解しました。検証の都合で後者を先に説明します。

6.9.1 定周期処理・コマンド受信処理

定周期処理・コマンド受信処理は HMI プロセスのルート（最上位）として実装します。

インクルードファイル

定周期割り込みの設定パラメータを定義します。検証時にキー入力を与えるとき、割り込みに間に合わないとエラーになってしまうので、割り込み周期を長くするようにしています。

hmi_process.h

```
/* hmi_process 手動操作部の制御情報
   初版：2020/9/11 Chuji
   最新版：2022/2/25 --- 自律走行車プロセスのコピーから改
造
*/
#ifndef __BUGGY_HMI
#define __BUGGY_HMI
#include "use-pigpio.h"
/* キースイッチを読み取る間隔 */
#define KEY_POLL_PERIOD  0.3
/* データ表示更新用のシステムタイマー設定 */
#define SYSTEM_DELAY  2  /* クロック発生までの待ち時間
（秒） */
#ifdef BCM2835
#define SYSTEM_PERIOD  0.49  /* データ更新周期（秒）
*/
#else
#define SYSTEM_PERIOD  10
#endif
#endif
```

プログラムファイル

このプログラムの前半でステートマシンを初期化します。後半では、キースイッチの状態を調べて自律走行車プロセスに伝え、Redis FIFO にコマンドが入っていたらステートマシンに渡します。割り込みは

使わず、キースイッチと FIFO を適当な周期でチェックすることで、割り込み問題を回避しました。

hmi_process.py

```
/* hmi_process.py 手動操作プロセス
   初版：2020.9.11 Chuji
   最新版：2022/2/25 --- 自律走行車プロセスから分離
オブジェクト部:
Class: hmi_process
属性：
   pi     pigpio オブジェクト
   fifo     REDIS FIFO オブジェクト
   hmi     手動操作解釈部オブジェクト
   key     キースイッチ読み取りオブジェクト
操作：
   HMIperiodicOps   定周期処理の制御
システム本体部：
   下位オブジェクトの生成とコマンド待ち
*/
#include "include/use-pigpio.h"
#include "include/use-sys.h"
#include "include/use-time.h"
#include "include/use-redis.h"
#include "include/hmi_process.h"
#include "keys.py"
#include "hmi.py"
pi = GPIO_OPEN()
keysw = key_switch(pi)
fifo = open_FIFO()
fifo.flushdb()  /* Redis FIFO/DB をクリアする（すべてのキーを削除する） */
/* HMI ステートマシンを生成する */
hmi = human_machine_interface(pi, fifo)
/* 主ループ（300ms に一回処理を行う） */
try:
  while True:
     /* キースイッチが押されていたら、FIFO を介して HMI に渡す */
    next, select = keysw.key_status()
    if next == KEY_ON:
      fifo.rpush(REDIS_TO_HMI, HMI_NEXT_KEY)
    if select == KEY_ON:
      fifo.rpush(REDIS_TO_HMI, HMI_SELECT_KEY)
    /* センサデータを表示中だったら更新する */
    hmi.sensor_update()
    /* FIFO からデータを受け取る（ブロックしない） */
    cmd = fifo.lpop(REDIS_TO_HMI)
    /* コマンドを受信していたらステートマシンに渡す */
    if cmd != None:
#ifdef TEST_HMI_PROCESS
      print('HMI command is received: ',
cmd.decode('utf-8'))
#else
      hmi.state_machine(cmd.decode('utf-8'))
#endif
    sleep(KEY_POLL_PERIOD)
except KeyboardInterrupt:
  sys.exit(0)
```

キースイッチの押下を検出したときは、(そのままステートマシンを呼び出さずに）自プロセスあての Redis FIFO に入れてやります。こうすることで、Web ブラウザ上のクリック操作と区別することなく処理できます。。

モジュール検証

このモジュールには自分を検証する機能が組み込んであります。BCM2835 と TEST_HMI_PROCESS を #define して実行すれば、0.3 秒ごとにキースイッチと Redis FIFO を調べ、キー操作を含むコマンドが確認できます。

```
$ cpp -DBCM2835 -DTEST_HMI_PROCESS hmi_process.py
>tmp.py; python tmp.py

HMI command is received:  NEXT
HMI command is received:  NEXT
HMI command is received:  SELECT
HMI command is received:  SELECT
HMI command is received:  SELECT
HMI command is received:  NEXT
HMI command is received:  SELECT
HMI command is received:  Hello
HMI command is received:  World!
:
```

```
$ python pusher.py Hmi
SELECT
Pushed: SELECT
Hello
Pushed: Hello
World!
Pushed: World!
```

6.9.2HMI ステートマシン

インクルードファイル

ステートマシンのステート名とコマンドを定義します。

hmi.h（抜粋）

```
/* hmi.h Human Machine Interface */
/* 初版：2014/12/26 Chuji（温度コントローラ用）
   最新版: 2022/2/25 --- 自律走行車用に改造 */
#ifndef __HMI
#define __HMI
/* HMI States */
#define HMI_IDLE  'Idle'
#define HMI_SSID  'Ssid'
#define HMI_WIFI_SEL  'Wifi'
#define HMI_IP_HIGH  'High'
#define HMI_IP_LOW  'Low'
#define HMI_BATTERY  'Battery'
#define HMI_SENSORS  'Sensors'
#define HMI_OBSTACLE  'Obstacle'
#define HMI_WEB_STOP  'Web'
#define HMI_WEB_RUN  'Web_run'
#define HMI_WEB_PAUSE  'Web_pause'
#define HMI_BLK_STOP  'Block'
#define HMI_BLK_RUN  'Blk_run'
#define HMI_BLK_PAUSE  'Blk_pause'
#define HMI_SCR_STOP  'Scratch'
#define HMI_SCR_RUN  'Scr_run'
#define HMI_SCR_PAUSE  'Scr_pause'
#define HMI_LIN_STOP  'Line'
#define HMI_LIN_RUN  'Lin_run'
#define HMI_LIN_PAUSE  'Lin_pause'
#define HMI_SHUTDOWN  'Shutdown'
#define HMI_REBOOT  'Reboot'
#ifndef NO_SHUTDOWN  /* シャットダウン／リブートを実行中 */
#define HMI_TURNOFF     'Turnoff'
#define HMI_REBOOTING  'Rebooting'
#else  /* シミュレーションでシャットダウン／リブート実行中は
      メインメニューでシャットダウン／リブートを選択する状態に戻る */
#define HMI_TURNOFF     HMI_SHUTDOWN
#define HMI_REBOOTING   HMI_REBOOT
#endif
/* スイッチ押下とコマンド */
#define HMI_NEXT_KEY  'NEXT'    /* NEXT スイッチが押下された */
#define HMI_SELECT_KEY  'SELECT'  /* SELECT スイッチが押下された */
#define HMI_UPDATE  'UPDATE'  /* LCD 表示の更新要求 */
#define HMI_TERMINATE  'Exit'  /* 実行中の自律走行プログラム終了通知 */
```

```
#define HMI_BATTERY_LOW   'LOW BATTERY'   /* 電池電圧低下通知 */
（後略）
```

プログラムファイル

このプログラムは非常に長いので、この本には掲載しません。プロジェクトファイルを参照してください。設計したステートマシンの上の状態遷移表どおりにコーディングしてあります。

```
hmi.py
（プロジェクトファイルを参照）
```

モジュール検証

このモジュールの基本部の検証には、検証済の LCD とキースイッチを使います。表示は LCD 上で、出力は Redis FIFO（キーは 'Command', 'Lcd0', 'Lcd1', 'Iframe'）を監視することで確認します。

まず、キースイッチだけで状態遷移図（基本部）どおりの動作を確認します。シャットダウンとリブートが実行されると、それ以上のキー入力を受け付けなくなるので、いったんプロセスを停止させてから、実行しなおす必要があります。

例外処理の検証は、Redis FIFO（キーは'Hmi'）にコマンド（具体的なテキストは hmi.h で定義されている）を与えてやることで行います。

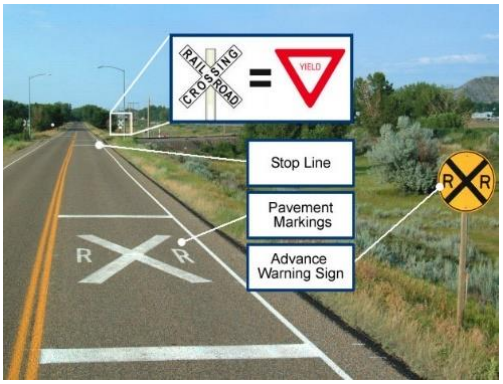
```
$ cpp -DBCM2835 -DSTAY_ON_WIFI hmi_process.py | python
$ python popmulti.py Command Lcd0 Lcd1 Iframe
$ python pusher.py Hmi
```

Web サーバーと html 文書の検証が終わったら、同じ検証を行います。キースイッチではなく、Web 画面画面上のボタンをクリックして、LCD 表示やインフレーム画面が変化することと、自律走行車プロセスへ適切なコマンドが送りだされることを確認します。

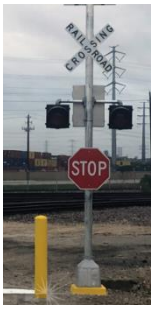
ステートマシンは OS のシャットダウンやリブートを含みますが、検証中に実行してしまうと面倒です。そこで、NO_SHUTDOWN を#define しておけば、シャットダウンやリブートは行わず、コンソールに表示するだけで、検証を継続できるようにしました。実際には use-sys.h と hmi.h で細工をしています。前にも説明しましたが、STAY_ON_WIFI を #define すれば、WiFi 環境の変更を禁止できます。他のすべての検証が終わるまでは、両方を#define していました。

コラム　列車は急に止まれない

海外ドライブ事情として「アメリカでは踏切手前で停まらない。そんなことすると追突される」といった、誤解されやすい記事をよく見かけます。「一時停止義務がある踏切は少ない」を理解し損なったようです。NHTSA（道路交通安全局）の報告によると、全米で年間 500 件ほどの踏切衝突事故が発生し、100 人以上が亡くなっています（日本の踏切事故は年間 10 件程度で、半分以上が歩行者）。政府や地方の安全委員会、自動車学校などは、安全啓蒙に躍起になっているのです。



上の写真の説明が分かりやすいと思います。踏切は、先のコラムに出てきた Yield と同じ（横切る線路が優先する）で、むやみに進入してはいけません。手前にある黄色い「踏切あり」標識や路面表示を見つけたら、後続車に注意しながら減速し、「安全を確認」してから、踏切に進入します。後続車も減速の意味を理解してくれますが、踏切直前での急停車は、やはり危険です。



見通しの悪い状況では、一時停止が義務付けられています。右のような標識で明示している踏切もあります。もちろん、停止信号や遮断機を無視してはいけません。

大陸の列車（とくに貨物列車）は長く慣性が大きいので、急ブレーキをかけても、停まるまでに 1 マイル以上進んでしまいます。まさに、「列車は急に止まれない」のです。

7　Web サーバーと Web ページの設計と検証

前章までに、自律走行車プロセスと、キースイッチ・LCD を使った HMI（操作）プロセスの設計と検証を（ほとんど）終えました。残っているのは、HMI プロセスと Web サーバー間の通信、それに各自律走行プログラムの操作です。これはブラウザあるいは Scratch プログラムから行いますが、操作コマンドとその応答は Redis FIFO を通して伝えられます。すでに自動走行車プロセスと HMI プロセス側の動作は検証済みなので、この検証は Redis FIFO のデータを調べる、あるいは Redis FIFO にデータを与えるだけで行うことができます。検証が終わったら、HMI プロセスと Web サーバーを立ち上げ、両者の通信と自律走行プロセスの最終検証を行ってください。

Web サーバーと Scratch オリジナルブロックは JavaScript で、Web ページは HTML と JavaScript で記述します。あまり詳細に説明すると『電子工作』の範囲を超えてしまうので、この本では設計の概要についてだけまとめ、詳細なコーディングの説明は省略します。プロジェクトファイルを参照してください。次に予定しているプロジェクトの報告書で手法を詳しく説明しようと思っています

また Scratch オリジナルブロックについては、拡張の仕方から説明しなければならないので、すべて省略します。ブロックドライブの動作を参考にしてください。個々のブロックは自律走行車プロセスにコマンドを与え、応答が返ってくるのを待つだけで、比較的単純なプログラムになります。

7.1　基本設計

7.1.1 二画面からなる Web ページ

Web ページは、次の二画面を組み合わせて使うことにします。

- 共通画面（LCD のイメージを表示し、二つのボタンで操作する）
- 自律走行プログラム用画面（仕様設計で描いた操作画面イメージ）

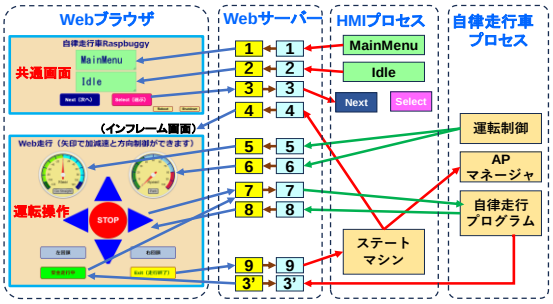
この二つの画面を同時に表示・操作できるようにするため、HTML のインラインフレームという機能を使うことにします。

7.1.2 Web ページとの通信

登場するのは以下の 4 つのプロセスです。Scratch ドライブを実行する場合は、Web ブラウザと Web サーバーが、それぞれ Scratch ブロックと Scratch インターフェースになりますが、プログラムの構造や通信手段はほぼ同じです。

- Web ブラウザ（外部）
- Web サーバー
- HMI プロセス
- 自律走行車プロセス

四者の間の（プロセス間）通信は次の図のようになります。サーバーは Redis FIFO と Socket.IO との間の橋渡しをするだけなので、（数は多いが）比較的単純な機能です。ブラウザからのコマンドの大部分は自律走行車プロセスに渡されますが、自律走行プログラムを実行・停止するコマンドなどは、HMI プロセスに渡されます。LCD 上の表示を変更するためです。



プロセス間通信チャンネル

番号で示した通信チャンネル（データの送受信経路）の用途を次の表に示します。Redis FIFO のキーはチャンネル名の前に `REDIS_TO_` を、IO イベント名はチャンネル名の前に `IO_EVENT_` を付けて使用します。

自律走行プログラムの終了ボタンの操作チャンネルは、番号を 3'としていますが、実際にはキースイッチ操作チャンネル 3 と同じ経路を通ります。

番号	チャンネル名	通信方向	通信データ
1	LCD_TOP	←HMI	LCD 一行目の表示データ
2	LCD_BOTTOM	←HMI	LCD 二行目の表示データ
3	HMI	→HMI	ボタンのクリック
4	IFRAME	←HMI	AP 別操作ページの指定
5	WHEEL	←運転制御	操舵量
6	SPEED	←運転制御	速度
7	COMMAND	→自律走行プログラム	運転操作コマンド
8	RESPONSE	←自律走行プログラム	運転操作の結果
9	ALARM	←自律走行車ルート	警報データ
3’	HMI	→HMI	AP 終了指令

Web ページの表示と操作に使う通信チャンネル

Web サーバーは、この表に従って Socket.IO と Redis FIFO の間でデータの受け渡しを行います。この通信チャンネルはプロジェクト毎に異なるので、Web サーバーを起動するときには、自律走行車用であることを示すため、BUGGY_SERVER を#define します。

```
$ cpp -DBUGGY_SERVER web_server.js | python
cleanfile.py | node
```

実装を簡単にするため、ここで一つの『手抜き』をします。それは、自律走行プログラムを操作するのは、その自律走行プログラムを起動するときに接続していたブラウザだけに限るという、使用上の制限です。自律走行プログラムの実行が進むにつれ、刻々と変わってきた画面を、後から接続してきたブラウザに反映するのが面倒だからです。その代わり、新しいブラウザからの接続があると、共通画面と速度・操舵量の表示だけは最新のものにする手順を用意しました。自律走行プログラムが起動されたときに接続しているブラウザが複数あるときは、どのブラウザからでも操作できるし、まったく同じ画面が表示されます。こうすることで、Web サーバーは要求を右から左へ転送するだけでよくなります。サーバーとしては邪道のような実装なので、あまりお勧めしません。

サーバーの記述を簡潔にするため、どのサーバーでも使う機能を web_server.h としてインクルードファイルにし、サーバー本体は短くて住むようにしました。Javascript では、改行は空白文字としてしか認識しないので、何行にも渡るマクロ定義が長い 1 行に変換されても処理できます。

_web_server.h

（プロジェクトファイルを参照）

サーバー本体 web_server.js は短い記述になりました。今後のプロジェクトなど用途ごとに変わる部分は、別に記述できるようにしています。ブラウザがソケットに接続してきた時の処理（OnSocketConnet）と、Redis FIFO からデータを受信したときの処理（SystemDependentReceiveFIFO）がそれです。今後のプロジェクトでも、この二か所（と通信チャンネル）だけを変更するだけで使えます。

_web_server.js

```
/* Raspberry Pi Project: Web サーバー
   2017/6/5   初版
   2019/5/21  - Web レコーダーを追加
   2019/6/5   - index.html 以外のファイル転送要求に対応
   2019/7/10  - 温度コントローラの選択機能追加
   2020/10/16 - Buggy 用に inframe を加える
   2022/12/3  - 非同期処理を明確に記述した
   2023/12/13 - 定型的記述をマクロにし、web_server.h として独立させた
*/
#include "include/web_server.h"        /* Web サーバーに必要なマクロ群の定義 */
/* 自律走行車プロセスへのコマンド定義を読み込む */
#include "include/hmi.h"       /* HMI に対するコマンド定義 */
#include "include/ap_manager.h"        /* AP Manager に対するコマンド定義 */
/* *** 冒頭の処理 *** */
/* Web サーバーを用意する */
PrepareHttpService;
/* Socket.IO サーバーを用意する */
PrepareSocketService;
/* Redis FIFO を使えるようにしておく */
PrepareRedisService;
/* Web サービス開始を準備する */
PrepareServices;
/* *** このシステムに固有の処理 *** */
/* 現在のフレームページを覚えておく */
let FramePage = WEB_EMPTY_PAGE;
/* ブラウザが接続してきた時に行う、このシステム固有の処理 */
function OnSocketConnet(new_socket){
   /* 現在の表示をブラウザ画面に反映する */
   Web_Send_FIFO(REDIS_TO_COMMAND, APM_UPDATE);
   /* 速度と操舵量を更新 */
   Web_Send_FIFO(REDIS_TO_HMI, HMI_UPDATE);    /* LCD 表示を更新 */
   new_socket.emit(IO_EVENT_IFRAME, FramePage);
   /* 現在のフレームページを表示する */
}
/* FIFO からデータを受信した時に行う、このシステム固有の処理 */
function SystemDependentReceiveFIFO(id, dat){
   /* iframe だけは受信したデータを記憶する */
   if (id == REDIS_TO_IFRAME){
      FramePage = dat;
   }
};
/* Redis と Socket.IO を含む Web サービスを開始する */
StartWebServer;
```

7.1.3 Web ブラウザ上の更新処理

ブラウザ上でボタンをクリックすると、それに対応するコマンドを自律走行車あるいは HMI に送りつけます。イベントを発生するサブルーチンを呼び出すとき、各ボタンに対して決めたコマンドをパラメー

タとして渡すようにしました。比較的単純な実装です。

逆にイベントを受信したときは、対応するボタンの属性（色や表示テキストなど）を変更してやります。そのために、ボタン（またはイベント）ごとの辞書を用意しておき、受信したメッセージに応じて新しい属性をセットすることで実現します。実装を簡単にするため、画面上のオブジェクト（ボタンなど）を指定する ID はイベント（と Redis FIFO のキー）と同じにしておきます。

例外として、自律走行プログラム操作では、画面上に配置された複数のオブジェクトを、唯一のチャンネル（RESPONSE）を通して変更します。そのために、変更するオブジェクトの ID を決定するための情報をメッセージに含ませておきます（それをもとに ID を決定する辞書を用意する）。表示変更に使う属性辞書も ID から探し出します。

上で説明したように、画面上のオブジェクトは、走行状態によって表示色やテキストが変更されます。『現在の状態』を Web 画面が持っていることになるため、後から接続したブラウザに表示されるオブジェクトには、最新の状態が反映されません。

Web ページ上に速度計や操舵計を表示するライブラリ Gauge.js を使います。それを使いやすくするためのマクロも用意しました。この表示だけは接続直後に、サーバーが最新の状態を与えます。

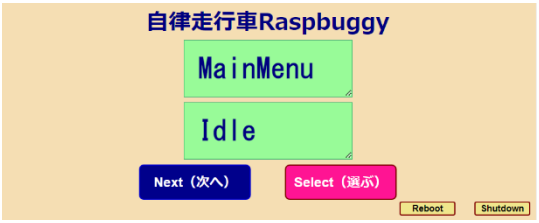
7.2　画面イメージの設計

Web ページを表示するファイルは、マクロを使って設計し、xxx.html.source というファイル名にしておきます。C プリプロセッサ cpp で処理したものを、xxx.html として使います。この処理は事前に行っておきます。なお、この節で表示している画面イメージは、すべて実際の Web ブラウザ画面をキャプチャしたものです。

```
$ cpp -DXXX xxx.html.source | python cleanfile.py |
python RecoverJapanese.py > xxx.html
```

7.2.1 共通画面

Web 画面 index.html.source には、LCD と操作用のボタンを表示させ、クリックすると、自律走行車上のキースイッチを押したのと同じ動作をさせます。システムの停止や再起動をさせるボタンを（右隅に小さく）追加しました。



共通画面

自律走行プログラムが実行されると、それ用の操作・表示画面を下側に表示します。

7.2.2 デフォルト操作画面

自律走行プログラムが実行されていないときは、Raspbuggy のロゴマークと各種自律走行プログラムの起動ボタンだけを表示しています。

自律走行プログラムが起動されると、その操作画面を表示するようにします。自律走行プログラムが終了すれば、デフォルト画面に戻ります。どの操作画面を表示するかは、HMI プロセスから知らされます。

デフォルト操作画面

7.2.3 Web ドライブ操作画面

操作画面のイメージに従い、操作と応答を定義していきます。

- 速度・操舵量を受信したらメーター表示を更新する。速度が正／ゼロ／負であるとき、進行方向表示を Drive/Park/Reverse にする。
- 4 方向の矢印をクリックしたら、減速・加速などのコマンドを発行する。
- 左右の回頭ボタンをクリックすると、その場で回頭を始める（初期イメージへの追加）
- STOP をクリックしたら、停車コマンドを発行する。
- 安全警報を受信したら警報ランプを点灯する。警報ランプをクリックしたら、回避コマンドを発行する。自律走行プログラムが警報の解除を検出したとき、消灯指示が送られてくる。
- 終了スイッチをクリックしたら、プログラム停止コマンド（HMI に対して）を発行する。

以上の動作を実現する JavaScript プログラムを HTML ファイル内に記述します。

Web ドライブ操作画面

7.2.4 ブロックドライブ操作画面

ブロックドライブ操作画面では、それぞれのブロックを四角形で表示します。安全警報、非常停止、終了ボタンも表示し、Web ドライブと同じ処理を行います。それに加え、

- ブロックをクリックすると、相当する実行コマンドを発行する。
- 自律走行プログラムから、ブロックの「実行中」あるいは「実行終了」報告が届いたときは、相当するブロックの表示色などを変更する

ことにします。

ブロックドライブ操作画面（低速で 30cm 前進中）

7.2.5 Scratch ドライブ操作画面

Scratch ドライブの操作は Scratch プログラムから行うので、ブラウザ上にはそのことだけを表示します。自律走行プログラムの停止だけが操作できます。

Scratch ドライブ操作画面

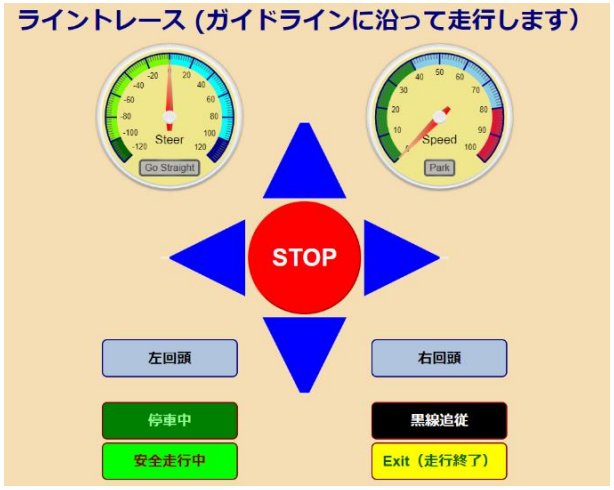
7.2.6 ラインとレース操作画面

ラインとレースの操作は、手動運転（Web-driving）と自動運転（Line-tracing）の二つのモードを持っています。Web-driving モードは、最初の位置に移動したり、障害物を回避したりするために使うもので、Web ドライブと同じ動作と操作を行います。表示と操作は Web ドライブと同じです。自動運転モードでは、次の操作を追加します。

- 自動運転ボタンをクリックすると運転モードを切りかえるコマンドを発行する
- 自律走行プログラムから自動運転の実行中／実行終了報告が届いたら、それに従って自動運転ボタンの表示色を変更する
- 色ボタンをクリックすると、ガイドラインの色を変えるコマンドを発行する。現在の色を受け取ってボタンの色を変更する

追加分は小さいので、Web ドライブと同じソースファイルに記述し、LINE_TRACE を#define することでラインとレース用の html ファイルを作り出します。

```
$ cpp -DLINE_TRACE web_drive.py | python
cleanfile.py | python RecoverJapanese.py
>line_trace.html
```



ライントレース操作画面

7.3 Web 画面のコーディング

用意する Web 画面（HTML ファイル）は以下の 5 つです。

	操作画面	ソースファイル (.html.source)	HTML 文書ファイル
1	共通画面	index	index.html
2	デフォルト	empty	empty.html
3	web ドライブ	web_drive	web_drive.html
4	ブロックドライブ	block_drive	block_drive.html
5	scratch ドライブ	scratch_drive	scratch_drive.html
6	ライントレース	web_drive	line_trace.html

Web 画面ファイル

前節にあるように、画面記述のかなりの部分が共通です。共通部の保守性を良くするため、それらを次のインクルードファイルに記述しました。インクルードするのは web_page.h だけで、他のファイルは自動的にインクルードされます。

ファイル名	記述内容
web_page.h	Web 画面の記述に必要なマクロ
web_color.h	Web 画面に表示する色の名前
socketIO.h	Socket.IO ライブラリの在処とイベント名
web_gauge.h	ゲージ描画用の定義とマクロ
web_project.h	プロジェクト毎に固有の定義とマクロ
to_text.h	マクロ定義したテキストを""で囲む

Web ページ記述用インクルードファイル

このうち to_text.h は分かりにくいと思います。C プリプロセッサは、マクロ名を引用符（"など）で囲むと、中身をマクロ展開してくれません（次の表の左側）。マクロ定義した名前など（引用符で囲まない）を引用符付きにするには NtoS というマクロを使います（下の右側）。これはスタイルシートやオブジェクト名を扱うときに重宝します。

test.txt（普通の記述）	test1.txt（to_text.h 使用）
#define ABC hello "ABC"	#include "to_text.h" #define ABC hello NtoS(ABC)
$ cpp test.txt ：(略) "ABC"	$ cpp test1.txt ：(略) "hello"

HTML 文書には xxx.html.source という名前を付け、cpp と整形プログラムで処理した結果を xxx.html として保存しておきます。cpp は、html で使うようなカッコで囲んでいない日本語テキストを処理すると、Unicode に変換してしまいます。日本語に戻すための整形プログラム RecoverJapanese.py を用意して対応しました。

```
$ cpp web_drive.html.source | python cleanfile.py
| python RecoverJapanese.py >web_drive.html
```

以上の準備を前提に、Web 画面の HTML 文書を作成します。ページの都合で、この本には掲載できませんでした。プロジェクトファイルを見てください。

7.4 Web 画面の検証

Web 画面（Web サーバーと HTML 文書）は、他のプロセスが動いていなくても検証できます。他のプロセスとの相互作用は Redis FIFO を介して行うので、それらの監視と送信を行えばいいのです。出力されるのは REDIS_TO_COMMAND と REDIS_TO_HMI の 2 つだけなので、同時に監視してやります。入力の方は、Redis FIFO のキーとデータを組にして与えてやります。

まず Web サーバーを起動して、ブラウザから http:(サーバーの IP アドレス):50020 にアクセスすると、基本操作画面とデフォルト操作画面が表示されます。

```
$ cpp -DBUGGY_SERVER web_server.js |python
cleanfile.py |node
```

この他に 2 つの SSH ウィンドウを開きます。一方（次に示す上側の画面）で Redis FIFO へデータを送るプログラム pushmulti.py を実行すると、サーバーを介してブラウザにデータを送ることができます。次の例では、下側のインラインフレーム画面を切り替えています。もう一方のウィンドウ（下側）では、自律走行車プロセスと HMI プロセス向けの Redis FIFO を監視します。

```
$ python pushmulti.py
Redis & Data? :  Iframe block_drive.html
Pushed:  block_drive.html   into:  Iframe
Redis & Data? :
```

```
$ python popmulti.py Hmi Command
```

仕様に従ってすべてのボタン操作と表示を確認します。ボタンの色や表示テキストを変更するには、'Response' FIFO を通して応答（'A'または'D'の後ろにボタン ID をつける）を送ってやります。web_server.h ファイルの中にある console.log() のコメントを外すと、サーバーの受け取った Redis データとイベントを表示してくれます。

すべての検証が終わったら、自律走行車プロセスと HMI プロセスを組み合わせて動作を検証します。自律走行車プロセス、HMI プロセスと Web サーバーを起動するシェルスクリプト start_raspbuggy を用意しました。

start_raspbuggy

```
# 自律走行に必要なプロセスを起動する
cpp -DBCM2835 hmi_process.py | python cleanfile.py
| python &

cpp -DPID_P_INIT=3 -DPID_I_INIT=5 -DPID_D_INIT=1.2
-DBCM2835 autonomous_vehicle.py | python
cleanfile.py | python &

cpp -DBUGGY_SERVER web_server.js | python
cleanfile.py | node &
```

システム起動時にこのまま実行しても良いのですが、デバグなどの都合で自律走行車が動いていない方がいい場合に備えて、下のランチャープログラムを用意しました。キースイッチを押しながらシステムを起動すると、LCD に'Vehicle Inactive'と表示するだけで終了します。キーを押さずにいると、上のシェルスクリプトを実行するようにしました。

start raspbuggy.py

```
（プロジェクトファイルを参照）
```

検証が済んだら、/etc/rc.local の末尾に追記して、システム起動時に実行できるようにしておきます。

/etc/rc.local（ファイルの最後に追記する）

```
:
/usr/local/bin/pigpiod -s 10 &
cpp /home/chuji/raspbuggy/start_raspbuggy.py
| python /home/chuji/raspbuggy/cleanfile.py |
python &
```

これで自律走行車『システム』の開発と検証が終わりました。楽しんでいただければ幸いです。

コラム　自律走行車のメリット

自律走行ができる自動車（いわゆる「自動運転車」）には、何が期待されているのでしょう？

1. 運転者のミスによる事故を防ぐ……よそ見運転をして急停車し、「止まってよかった」と言う不謹慎な CM はさすがに少なくなりましたが、自動走行レベル 5 が達成されるまでは、絵に描いた餅ではないかと思います。
2. 高齢者の移動手段を確保する……いわゆる「移動弱者」を救済するというたい文句ですが、公共交通網の発達した都会部でのメリットはあまり多くないと思います。むしろ交通手段の少ない地方に増えている「買物弱者」にとってありがたい存在になるかもしれません。ただし、田舎の舗装されていない道路を走る必要があります。
3. 物流の人手不足を解消する……これまで猶予されていた運転手の労働時間規制が、2024 年 4 月に発効しました。交代要員を確保できない運送会社が長距離便を減らしたり、撤退したりする事態が懸念されています。とはいえ、大型無人トラックが高速道を爆走する姿を想像するのは、あまりゾッとしません。初期には自動追尾機能を持った車両を先導する方式のほうが安心できますね。交代要員を乗せても、輸送量を倍以上にできれば、価値があります。列車のように長いトラックを運転するわけではないので、運転手の負担は増えません。
4. 宅配の駐車問題が起こらなくなる……すでに実証実験が始まっていますが、小型の自律型移動体に玄関先まで配達させるというアイディアです。あまり困難な技術というわけではないので、すぐに街角の風景に溶け込んでいく可能性があります。ただし、コロナ禍を機に急増した（料理）配達員はフリーランスの人が多いので、仕事がなくなるという危機に直面するかもしれません。

ちょっと悲観的な見方かもしれませんが、山積する課題を解決できれば、急速に普及していくと思います。

8　プロジェクトを振り返って

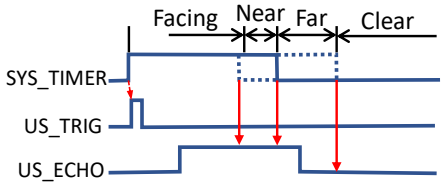
Raspberry Pi を使った自律走行車プロジェクトは、当初の目標をほぼ達成できました。Scratch プログラムからの制御と、「喋る自律走行車」については、未検証や未完の部分が多いので報告から外しました。『電子工作』としては、カバーする範囲が広くて面白いプロジェクトだったと思います。

8.1　脱線ぎみの進行

今回のプロジェクトでは、順調に進んでいたときに問題点や課題点が現れ、脱線ばかりしていました。急遽べつのプロジェクトを始めたり、構想外の開発が発生したり、完成間際のモジュールの構造を変更したりしました。さらに自宅の引っ越しまであって、予定が大幅に遅れてしまいました。

障害物センサ

超音波送信から反射波検出までの時間を測るのに、当初は次の図のような簡易方式（ソフトウエア処理）を考えていました。



超音波センサの信号検出方法（当初案）

GPIO の SYS_TIMER ピンにパルスを発生させ、立ち上がりで超音波を送信します。立ち下がりで割り込みをかけ、反射波を受信しているかどうかを検出します。障害物までの距離を複数の領域に分け（上図の場合は 4 つの領域）、その境界になるタイミングで割り込みを発生させるようにパルス幅を変更します。一回の送信毎にパルス幅を変えながら確認し、挟み込むようにして領域を決めることができます（上図の場合は”Far”領域）。

この方法だと、時間信号（SYS_TIMER）と受信信号入力（US_ECHO）の 2 ポートを使うだけで、簡単に障害物の有無が分かるという目論見でした。ところが pigpio を呼び出すのに 1ms 程度かかることが分かり、ハードウエアカウンタでの計測が必要になってしまいました。

しかし Raspberry Pi のコアには、この目的で使えるカウンタがありません。走行計にもカウンタ機能が必要です。外付けチップを探しましたが、入手が困難だったり、カウンタ数が少なかったりで、適当なものが見つかりません。そこで、ワンチップマイコン PIC で汎用計測チップ PICCOLO を自作することにしました。詳しくは、そちらの報告書を見てください。

このチップ開発のため、自律走行車プロジェクトを半年以上休眠させることになってしまいました。

仮想走行

自律走行プログラムの仕様設計をしているあいだ、「これをどうやって検証したらいいか」悩み続けました。実機を動かす前に、アルゴリズムを検証しておかないと、危険極まりないことになりかねません。

DRIVER_STUB を#define すれば、自律走行プログラムが運転制御部に何をさせようとしているか分かるので、論理を追いかけることはできます。また、BCM2835 を#define しなければ、入出力をコンソールで代行することもができます。しかし、それから車両がどう動くのか、想像するのは簡単ではありません。ライントレーシングに至っては、まったくお手上げでした。

そこで思いついたのが、「計算機の中で車両を走らせる」ことでした。思いついたというより、多くの実プロジェクトの常とう手段だと、思い出したというのが正しいでしょう。モーターや車輪などの機構は分かっているので、短い時間後の位置や向きがどう変わるか、モーター駆動信号から計算することができます。これを積み上げていって、車両の走行状態を描こうというわけです。ガイドラインとの位置関係から、どのセンサがガイドラインを捉えているか計算することもできました。

実時間で計算する必要はないので、「このコマンドを与えてから何秒間だけ走行する」といった操作をくり返すと、本当に自律走行しているかのように見えます。

理論立てて計算し、プログラムにしました。計算ミスがあったりして、結構時間をとられました。でも、それに助けられたこともあります。思いがけない車両の動きがあったときも、現象の再現が可能だし、その時の内部変数を調べることで、何が起こっ

ているのか理解することができました。ライントレーシングの PID チューニングを、実際に走行することなく行えました。Raspberry Pi や、他のハードウエアがなくてもできるので、旅行先や移動中にこっそり楽しんでいました。もっとも、けっきょく実機でのチューニングが必要でしたが。

オブジェクトの継承

自律走行プログラム群には、データや操作に多くの共通点があります。最初はコピーを使って作っていたのですが、同時にバグも再生産することになります。そこで共通部は別ファイルにして、各々に #include していました。

検証がほぼ終わったころになって気が付きました。「共通部を親オブジェクトとして継承すればいいのでは」と。コードを見直すと、確かにこれで記述できます。親子のオブジェクトを定義しなおし、ファイルを書き直しました。検証後の大幅な変更はリスクが大きいのでためらいがちですが、ここは『プログラムの美しさ』を重視することにしました（第 1 号プロジェクト報告書 51 ページ、コラム「プログラムは簡潔に美しく」参照）。

ところがやってみると、再設計後の書き直し作業（4 つのプログラム）は半日もかかりませんでした。共通部の取り込みを `super()` に置き換え、親オブジェクトの宣言をすれば、後はほとんど元のままです。それでもファイルは見違えるほど読みやすくなりました。なまじ cpp で処理することに慣れ過ぎていたことが足かせになってしまったようです。

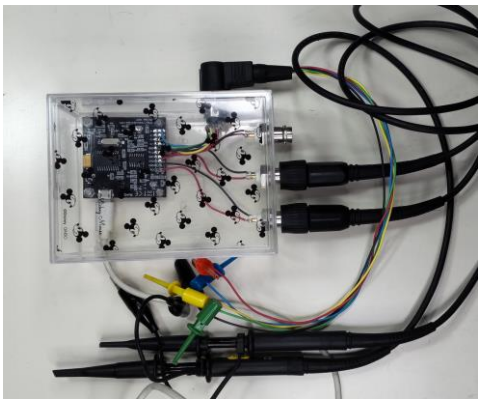
計器盤

自動車には様々な情報を表示する計器盤があり、Web 表示にも取り入れたいと思っていました。最初はスペースと指針を表す文字の組み合わせ（'0___▲_____100'）を表示し、指針となる文字（'▲'）を移動させていました。

もっと本物らしい例を探していたら、Canvas Gauge に出会いました。カスタマイズもできるし、使いやすいライブラリです。今回は丸形メーターだけを使いましたが、次のプロジェクトではバー表示器を考えています。

測定器

PICCOLO チップの検証にロジックアナライザがあると便利なので、安価な簡易型（2 現象オシロ＋4ch ロジアナ）キットを手に入れました（秋月電子通商。もう売っていない）。測定部だけで、操作や表示は USB 接続した PC 上で行います。



簡易型オシロ／ロジアナ

あまり高速の現象を捉えることはできませんが、今回のようなプロジェクト（10μs より遅い現象）では十分に役立ってくれます。特に PICCOLO チップの検証で多くの出番がありました。

8.2　開発途中のやり直し

今回のプロジェクトでは事前検討に時間をかけたのですが、開発途中で「しまった！」と思ったことがずいぶんありました。ハードウエア関連では、次のようなことが判明しました（本文で説明している）

1. 超音波センサのソフトウエア検出ができない
2. LCD の 1.27mm ピッチ基板直付けを諦めた
3. A/D 変換器は PICCOLO チップで代用できる
4. 電池と AC アダプタをホットスワップできない
5. 反射光センサの迷光対策が十分でない
6. 電源コネクタの誤挿入対策がとれていない

このうち、1 と 2 は回路を変更した（変更しないと動作しない）ものの、あとは放置しました。変更が面倒で、開発を停滞させたくなかったからです。

いっぽうソフトウエアは多くの変更をしました。

A. 自律走行プログラムの共通部を継承に変更した
B. 割込みとバックグラウンドの干渉対策をとった
C. 検証のため、仮想走行を組み込んだ
D. 自律走行プログラムの選択などをブラウザから直接指定したくなったので、HMI ステートマシンに例外処理を加えた
E. 電源投入時に、自律走行車システムを自動起動する／しないを選べるようにした

このうち B は必須でしたが、あとは「美しさ」や「使い勝手の良さ」を追求したものです。工具を握るよりキーボートをたたく方が楽だと思っているわけではないのですが……。

8.3　積み残した課題

今回のプロジェクトでやりきれなかったことと、発展の方向を整理しておきます。

pigpio と割込み

pigpio はユーザ権限で（sudo せずに）実行できるし、複数のプロセスから呼び出せるなど、とても使い勝手が良いライブラリです。今後も利用していきたいのですが、割込みやブロッキング型 OS コールとの相性が悪いことが分かりました。

デーモンとの通信資源が一つしかないことが原因らしいのですが、いちいち資源の獲得と解放を繰り返すのは効率が悪いのでしょう。BLPOP を使ったプロセスの実行制御はこれからも使いたいので、プロセスの設計を工夫する必要があります。次のプロジェクトでは、初期の設計段階からこの対応を検討したいと思います。

メカの駆動

自律走行車では、100ms 周期のメカ制御を試み、なんとか動作させることができました。もともと Linux はリアルタイム OS ではないし、これ以上早い処理を Python で行うのはしんどそうです。

自律走行プログラムの機能を増やしていくことなどを考えると、『作業分担』を視野に入れる必要があります。「単純だが早く処理する必要がある（IO 点数が多い場合もある）」機能と、「早い応答は必要ない（人間並み）が、複雑な処理や判断が求められる」機能に分類するのがいいと思います。自律走行車の場合は、初期構想で分析した、（身につけた）運転制御機能から下側と、（知的な）計画と意思を担う上側という分担です。それぞれを別のプロセッサに分担させ、通信を使って協調動作させます。

前者は「IO プロセッサ」などと呼ばれるもので、ハードウエアの能力を最大限引き出す役目があります。ライントレーシングの実行部を含めます。PICCOLO チップの機能を拡張した形で、2 チャンネルの PWM と 3 つのカウンタ、それに多くのポート（モーター制御（4）、崖センサ（2）、表示灯（4）、停車信号（1）など）を使う IO プロセッサを、I2C バスで Raspberry Pi とつなぐという構成です。14 ピンのベースチップ（PIC16F15325-I/P）では GPIO ピンが不足するので、パッケージが大きいものか、上位 PIC が候補になります。インターネット上の製作記事も、PIC などを使った『早い制御』を実現したものが多いですね。

ロボット言語

IO プロセッサが用意できれば、その上位のプログラムはメカの詳細を知らなくても設計できるようになります。いわゆる『高級言語』でメカを動かせるようにしたり、ネットワーク上の資源を使ったりすることが考えられます。

秘書ロボット『カオナシ』プロジェクトで引用した、Rapiro（https://www.rapiro.com/ja/）や、講談社などが販売した『週間鉄腕アトム』（https://pc.watch.impress.co.jp/docs/news/1045585.html）も、IO プロセッサと Raspberry Pi を組み合わせる構成でした。

今回はロボット言語として、ブロック組み立て方式と Scratch からの操作を試しました。メカを分離しておけば、Python などのライブラリとしてロボット操作ができるようになります。こういう方向での発展もアリだと思います。もっとも、それはもう『電子工作』ではないのかもしれません。

Web 設計の詳細

今回のプロジェクトで、は Web 設計について詳しく説明できませんでした。次には、第 1 号プロジェクトで製作した温度コントローラの大幅改造を予定しています。そこで Web の設計についても報告しようと思っています。

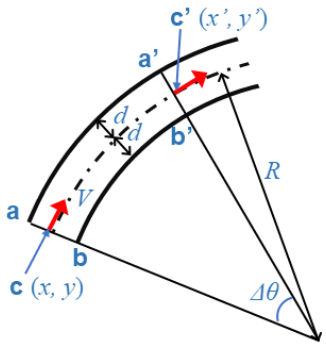
8.4　不安な世界

Web ブラウザで速度計などを表示するのに使わせてもらった Canvas Gauge は、非常にきれいで、使いやすいツールです。これからのプロジェクトでも活用できると思っています。

開発者のミハイロ・スタニク（Mykhailo Stadnyk）氏はウクライナのキーウ在住で、ソフトウエアエンジニア／アーキテクトとして 20 年以上のキャリアがある人です。数年前にキプロス（税金が安く、現地に行かなくても法人が設立できる）でのベンチャー立ち上げに参加したところまでは知っていました。しかしロシアのウクライナ侵攻から半年後の 2022 年の秋以降、ほとんど活動が見えなくなりました。SNS への投稿も少なく、数語の簡単なものばかりです。戦時下で、思うような活動ができていないのではないかと危惧しています。早く平和を取り戻すことを祈念しながら、自分にできることをやっていこうと思います。

仮想走行モデル

走行中の車両の位置と向きのシミュレーションを、短時間内の変化が蓄積したものとして計算していきます。最初に、左右の動輪の回転速度が異なる場合を考えます。このとき車両は円弧に沿って移動します。



左右の動輪の間隔を $2d$、車両の走行速度を V で表します。右図では動輪の軌跡を太線で、車両の位置（動輪の中央の点）の軌跡を一点鎖線で表わし、短い時間 Δt の間に動輪が **a** から **a'**、**b** から **b'**へ動くものと考えます。左右の動輪の走行速度は異なっていて、それぞれ $V + \Delta v$、$V - \Delta v$ だったとします。車両は Δt の間に半径 R の円弧（中心角を弧度法で $\Delta\theta$ とする）を描き、左右の動輪はそれとの同心円上にあります。

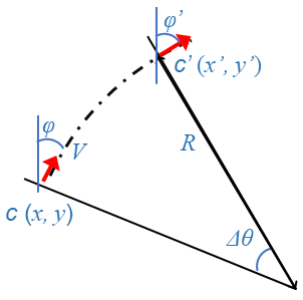
弧 **aa'**と **bb'**の長さはそれぞれ、

$$\mathbf{aa'} = (R + d)\,\Delta\theta = (V + \Delta v)\,\Delta t$$
$$\mathbf{bb'} = (R - d)\,\Delta\theta = (V - \Delta v)\,\Delta t$$

と表されます。これを R と $\Delta\theta$ について解くと、

$$R = V\,d / \Delta v$$
$$\Delta\theta = \Delta v\,\Delta t / d$$

となります。



車両の位置が **c** (x, y) から **c'** (x', y') に移動したとします。車両の向き（速度 V の方位角: Y 軸から測り、右側を正とする）を φ で表すと、

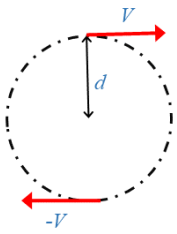
$$x' = x + V \sin\varphi\,\Delta t$$
$$y' = y + V \cos\varphi\,\Delta t$$

となります（回転運動なので、V の大きさは一定）。その時、速度ベクトル V は $\Delta\theta$ だけ回転しているので、新しい位置での車両の向きは、$\varphi' = \varphi + \Delta\theta$ となります。

この微小な変化を Δt ごとに積算していけば、車両の位置と向きが求められます。

左右の動輪の回転速度が同じ（$\Delta v = 0$）ときは直線走行で、上の式で $\Delta\theta = 0$（$R = \infty$）とした場合として含まれています。

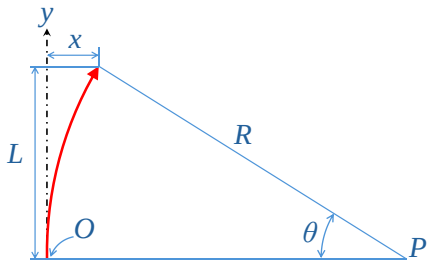
左右の動輪の回転が逆向きで、速度が同じ場合は、その場所で車体が回転（回頭）します。回転の半径は d なので、左車輪の移動速度を V（右車輪は$-V$）とすると、短い時間 Δt の間に、車両の向きは ϕ から $\varphi' = \varphi + V\,\Delta t / d$ に変化します。



直進補正

仮想走行モデルの解析は、駆動系の直進補正にも応用できます。

原点 O から上（y 軸方向）に直進しようとするとき、左右の車輪の速度が同じでないと、円弧（赤線）を描くようになります。遠方にある弧の中心を P、半径を R とします。車両が y 軸方向に L だけ進んだ時、横方向に x だけ外れていました。このときの弧の角度を θ で表わします。



$$L = R \sin\theta$$
$$x = R\,(1 - \cos\theta)$$

左右の動輪の間隔を d、左車輪の速度を V、右車輪の速度を $g\,V$（$g \leqq 1$）、その時の走行時間を t とすると、

$$\theta = V t / (R + d) = g V t / (R - d)$$

これを R と θ について解くと、

$$R = (1 + g)\, d / (1 - g)$$
$$\theta = (1 - g)\, V t / 2d$$

これを x と L に代入します。θ は小さいので、$\sin\theta \approx \theta$、$\cos\theta \approx 1 - \theta^2 / 2$ と近似できます。

$$L \approx V t$$
$$x \approx R\,\theta^2 / 2 \approx (1 - g)\,(V t)^2 / 4d$$

最後の近似は、$1 + g \approx 2$ を使っています。これを g について解くと、

$$g = 1 - 4\,d\,x / L^2$$

となり、測定値 L と x からこの車両の g の値を推定できました。補正として、右車輪を g の逆数だけ加速してやればいいことになります。

仮想ライントレーシングモデル

走路センサ（車両の横方向に 5 個が配置されている）のうち、どれが走路を検出しているかをシミュレーション上で調べ、ライントレーシングの動作を模擬します。これは、車両の位置と向き、それにガイドラインの形状によって決まります。

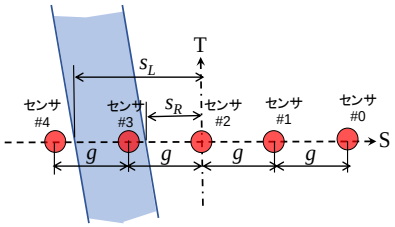
まず、計算のシナリオを説明します。走路に固定した座標系 X-Y を考え、車両の位置を(x_0, y_0) で、車両の向きを Y 軸からの角度 φ（右回りが正）で表します。このとき車両の向きに垂直に取り付けられた走路センサ群を延長した直線の傾き m は、$-\tan\varphi$ で、直線の方程式は、$y - y_0 = m(x - x_0)$ です。

この直線と、ガイドラインの境界線（の片側）との交点を　(x_1, y_1)とします（交点が 2 つある場合は、もうひとつを(x_2, y_2)とする。交点が存在しない場合もある）。この交点の座標を、車両に固定した座標系（車両の向き方向を T 軸、センサ軸方向を S 軸とする）表示に変換（座標軸を$-\varphi$ だけ回転）すると、

$$s = (x_1 - x_0)\cos\varphi - (y_1 - y_0)\sin\varphi$$
$$t = 0$$

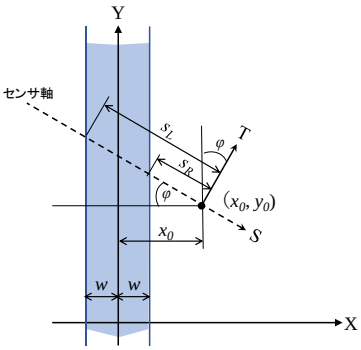
となります、s は車両中心からセンサ軸に沿った（符号つき）距離です。ガイドライン両側の境界線との交点のうち、他方の左にある方との距離を s_L、右にある方との距離を s_R とします（$s_L < s_R$）。

走路センサは間隔 g で 5 個が配置されています（中央のセンサが車両の中央になる）。右図のようにガイドラインに（青い領域）と重なったセンサがオン（検出）になり、外れたセンサはオフになります。それぞれがオンになる条件は次の表のようになります。オンになる範囲は、実際にはセンサの直径分だけ広くなりますが、走行シミュレーションでは無視してもいいでしょう。



センサ#4	センサ#3	センサ#2	センサ#1	センサ#0
$s_L <= -2g$, $s_R >= -2g$	$s_L <= -g$, $s_R >= -g$	$s_L <= 0$, $s_R >= 0$	$s_L <= g$, $s_R >= g$	$s_L >= 2g$, $s_R >= 2g$

直線走路の場合

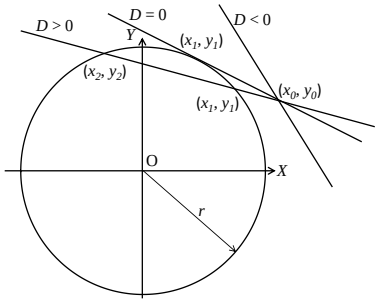


最初に、Y 軸に沿った幅 $2w$ の帯（図の着色した部分）状のガイドラインを考えます。ガイドラインの境界線は $x = \pm w$ です。これとセンサ軸との交点は $(\pm w\,,\ y_0 + m(\pm w - x_0))$です。これを (s, t) に座標変換し、$s_L < s_R$ となるように s_L と s_R を決めます。

円環状走路の場合

原点を中心にした半径 r の円環状走路を考え、幅 $2w$ のガイドラインをシミュレートします。

まず、幾何学的な解析をしておきます。点 (x_0, y_0) を通る傾き m の直線（センサ軸を表わす。車両の向きを φ とすると $m = -\tan\varphi$）と、原点を中心とした半径 r の円との交点を求めます。



直線の方程式 $y - y_0 = m\,(x - x_0)$ を y について解いてから、円の方程式 $x^2 + y^2 = r^2$ に代入します。

$$\begin{aligned} x^2 + y^2 &= x^2 + \{m\,(x - x_0) + y_0\}^2 \\ &= (1 + m^2)\,x^2 - 2m\,(mx_0 - y_0)\,x + (mx_0 - y_0)^2 \\ &= r^2 \end{aligned}$$

この解は、$\eta \equiv mx_0 - y_0$ と置いて、

$$\begin{aligned} x_1 &= \{2m\,\eta \\ &\quad \pm \sqrt{4m^2\eta^2 - 4\,(1 + m^2)\,(\eta^2 - r^2)}\} \\ &\quad / 2\,(1 + m^2) \\ &= \{m\,\eta \pm \sqrt{D}\} / (1 + m^2) \end{aligned}$$

となります。ただし、

$$D \equiv (1 + m^2)\,r^2 - \eta^2$$

としました。次に y について解くと、

$$y_1 = \{-\eta \pm m\sqrt{D}\} / (1 + m^2)$$

となります。$D > 0$ のとき、直線は円と 2 点で交わり、$D = 0$ のときは 1 点で接し、$D < 0$ のときは円との共通部がありません。点 (x_0, y_0) が円の内側にあるときは、必ず 2 点で交わります。

これから s_L と s_R を決めるのですが、交点がなかったり、2 個あったりするので、次の表のように、場合分けして考えます。交点が 1 個（接している）場合は、交点がないのと同じことにします。また、内円・外円との交点が 2 個ずつあるときは、近い方の走路のみを検出することにします。

(x_0, y_0)	交点数	端点 1	端点 2
ガイドラインの内側	外円 2 内円 2	内円との交点の近い方	外円との交点の近い方
ガイドライン上	外円 2 内円 2	内円との交点の近い方	外円との交点の近い方
	外円 2 内円 1	外円との交点の一方	外円との交点の他方
	外円 2 内円 0		
ガイドラインの外側	外円 2 内円 2	内円との交点の近い方	外円との交点の近い方
	外円 2 内円 1	外円との交点の一方	外円との交点の他方
	外円 2 内円 0		
	外円 1	走路センサ検出なし	
	外円 0		

内円との交点が近くに 2 つあるとき、遠い方の交点の先のガイドラインに走路センサがのっている可能性がありますが、無視することにします。走路センサの検出パターンから偏差を計算するときも、そのようなパターンは『ノイズ』として処理するようにしています。

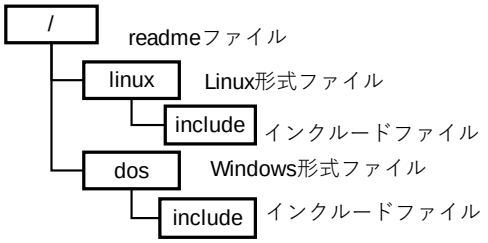
走路シミュレーションの検証

走路シミュレーションの検証プログラム test_simulated_track.py は掲載していません。プロジェクトファイルを参照してください。検証方法と使い方はプログラムの冒頭で説明しています。

プロジェクトファイル

このプロジェクトで作成したファイルをまとめてダウンロードできるようにしてあります。学習や研究目的であれば、自由に改造や再配布をしてもらって構いません。ただし著者は、プログラムの実行によって起こった、どのような事態にも責任を負いません。

ファイルは ZIP 圧縮されており、以下のようなディレクトリ構成になっています。linux ディレクトリには、そのまま Raspberry Pi で使えるファイルが、dos ディレクトリには、port2dos で変換したファイル（Windows 環境で内容を読むのが目的です。）が収納されています。タブを空白 2 文字に置き換える t2s.c や port2dos などのツール、テストパターンやデータ・試験結果は linux ディレクトリにしか入っていません。各々のソースディレクトリの下には、インクルードファイルを収容するディレクトリ include があります。



配布ファイルのディレクトリ構成

プロジェクトファイルのダウンロードは以下から:

https://www.akiyama-tokyo.net/electronics/raspbuggy.zip

本文中に掲載しているソフトウエア等は、dos ディレクトリのファイルから空白行を除いたものですが、手違いで更新できていないかもしれません。プロジェクトファイルが最新版です。

コラム　こんな自律移動（飛翔）体はイヤだ

地上を走る自動車に限らず、空中・水上・水中を自律的に移動できる機材に関する技術は、このところ長足の進歩を遂げています。安全確保が必須条件ではありますが、私たちの生活に溶け込んでいくことになるでしょう。ドローンを使って離島に物資を送ったり、被災地上空で通信を確保したりするには、自律性が欠かせません。

いっぽう人類には、新しい技術をおぞましい用途に使ってきた過去があります。ウクライナの戦場では、空や海を自律航行するドローン兵器による双方の攻撃が続いています。「従来の自動追尾ミサイルと、どこが違うというのか？」と、居直られそうですが、悲劇をエスカレートさせているのです。

ミサイル一発は数億円しますが、ドローン一機は十数万円で調達できます（最近、米軍が紅海でドローン迎撃にミサイルを使った『非経済性』が問題になった）。『低コスト』兵器は、大量に、また安易に使用される可能性をはらんでいます。ウクライナでは、家庭で自爆型ドローンを組み立てる女性が出てきました。「男は戦地へ、女は銃後で」という古めかしい考えが残っているのでしょうか？　日本でも戦争中に女学校生を動員して、飛行機部品や風船爆弾を作らせたことがあります。自律性のない風船爆弾の戦果はゼロでしたが、アメリカでは日曜学校の野外活動に来ていた児童 5 人と引率の女性が爆弾に触れて亡くなりました。それを知って長く苦しんだ女学生もいます。ウクライナの彼女も、戦争が終わった後で辛い思いをしなければよいのですが。

Raspberry Pi 中級電子工作

自律走行車

2024 年 4 月

著者・発行者　秋山忠次 (chuji@akiyama-tokyo.net)

非売品

いろいろな走行方法があります

Web Drive　Block Drive　Scratch Drive　Line Trace

ブロック運転（ブロックを実行して運転します）

緊急停車

手順は空	記録停止中	ステップ実行	最初から実行	手順保存
低速走行選択中	中速走行	高速走行		手順ロード
10cm前進	30cm前進中	10cm後退	30cm後退	3秒休止
45度左回転	90度左回転	45度右回転	90度右回転	

安全走行中　Exit（走行終了）

ライントレース（ガイドラインに沿って走行します）

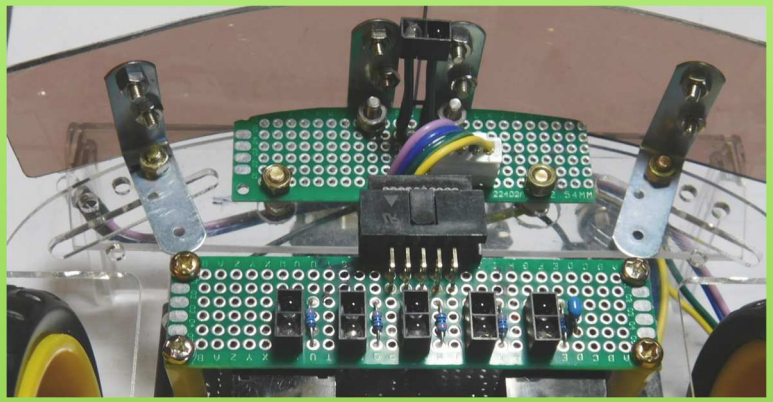
Steer　Go Straight　Speed　Park

STOP

左回頭　右回頭

停車中　黒線追従

安全走行中　Exit（走行終了）



非売品