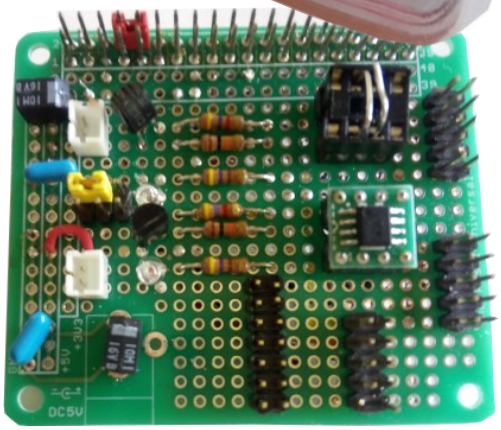
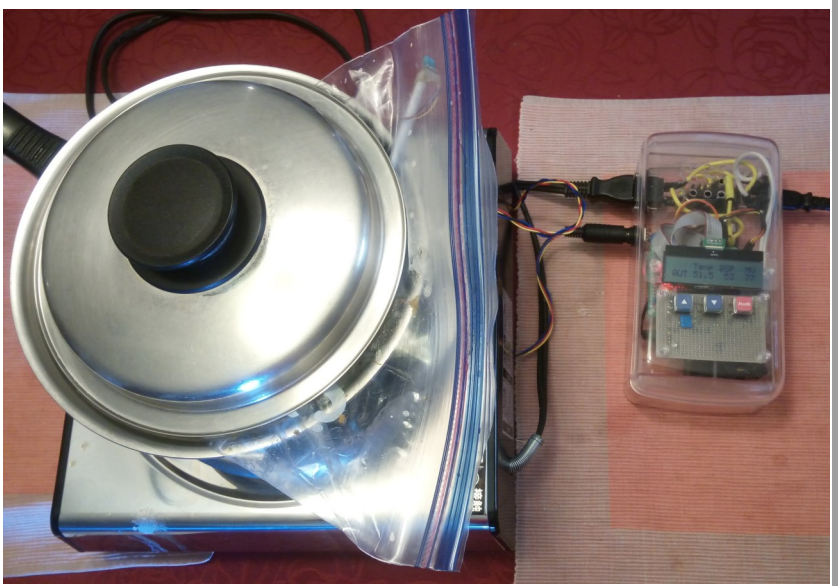
Raspberry Pi　中級電子工作

温度コントローラ

設計から製作・検証・応用・保守まで





著者　秋山忠次

（非売品）

目次

コラム一覧

はじめに

2014 年の秋、出版されたばかりの本（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）を書店で見つけました。読んですぐに、「これはいい！」と思いました。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 が立ち上げられるよう、Raspbian が提供されている。
2. 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 上で検証できる。
4. GPIO を駆動するためのライブラリが提供されているので、システム LSI である BCM2835 の詳細を知らなくても使える。
5. I2C バス、パルス幅変調、タクトスイッチ検出の実例が載っている。

などです。さっそく Raspberry Pi（当時はモデル B）を手に入れるとともに、温度コントローラを念頭に設計を始めました。必要なハードウエアをどうするか、ソフトウエアをどう構成するかを仕様書にまとめていきました。

仕事の合間に進めたので、実際に動作しだしたのは 1 年以上経ってからです。それまで Raspberry Pi にはプリンターサーバーとして「出向」してもらいました。

使っていくうちに、あれこれ改造したくなって、仕様書を何度も改訂しながら、機能を強化していきました。

その内容を公開しようと思ったのは、長らく積読状態だった本（木村英紀「ものつくり敗戦」日経プレミアシリーズ）を読んでからです。

勝手な解釈だとのそしりを恐れずに言うと、「ものつくり敗戦」の要旨は次のようなものです。産業革命に続く新しいパラダイムシフト（現著者は「第三の科学革命」と呼んでいる。ここでは、第一と第二は省略）が世界中で進むなか、技術は産業財を、道具→機械→システムと変化させてきた（最初の矢印が産業革命、二つ目の矢印が第三の科学革命）。日本は産業革命のなかで、「機械を道具のように使いこなす」逆向きの道をとり、多数の優秀な労働力にもとづく産業を発展させた。産業革命の結果として生まれた、大量生産・大量消費社会では、複雑で不確実な対象（生産工程、市場など）を扱う必要が生じ、システム化が不可避となった。そこで必要になるのは、普遍性を持ち標準化できる技術だが、ヒト依存の日本では産業革命のときのような対応ができず、世界に後れを取りつつある。苦手分野である、理論と論理、システムとソフトウエアを克服しなければならない。

たしかに日本では、社会的地位の高い人が「わしは数学（あるいは理論、コンピュータ）が苦手でね」と自慢気に語ることがあります。理論を語ろうとすると、「ろくな経験もないくせに頭でっかちなことを言う」と揶揄されることもしばしば（私のひがみも少々入っている）。目に見えるものを仕上げる「匠（たくみ）の技」も大事ですが、複雑で目に見えないものを誰にも扱えるようにすることが、これから必要になると思います。

いまの日本では、昔のように多数の経験豊かで優秀な人材に依拠する産業は、先が見えています。いっぽうで、小学生にプログラミング教育を施そうという、付け焼刃になりかねない政策も始まりました。心ある担当官僚は、「国民皆プログラマー化を目指すつもりは毛頭なく、論理的な考え方を身に着けさせることが目的」と言っていますが、現場ではプログラミング技能を偏重する動きがあるようです。

「ものつくり敗戦」でも、日本人（あるいは日本企業）は何事も、「課題がある」→「いろいろやってみた」→「できた」→「よかった」で終わってしまうことが多いと述べています。せっかくの努力を普遍化しようとしない性向があるようです。

私自身はソフトウエアの専門家から程遠い存在ですが、基礎を学び、システム的なアプローチを取ろうとする態度は、それほど劣っていないと自負しています。その模索過程を公開することが、普遍化という視点で、多少は人の役に立つのではないかと考えた次第です。

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。技術者には「よそ見」が必要だし、それが視野を広げる契機になると考えているからです。楽しんでいただければ幸いです。

2019 年 5 月　著者

1　Raspberry Pi プロジェクト

この本では Raspberry Pi を利用して、実際に使える装置（作品）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」を卒業しよう！

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、入門書は大事だと思います。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「温度計の表示が読めた」まで行きついた先はどうでしょう？　何をしたいのか、自分の目標を持つといいと思います。せっかく作るのなら、何か役に立つものが良いかもしれません。それも、ある程度の期間、使い続けることができるものが。そんな思いから、自分が使うものを作り上げる過程を、プロジェクトとしてまとめました。電子工作入門を卒業して、中級編か応用編を目指しませんか？

何を作るかは、皆さんのアイディアしだい。ガーデニングの水やりを目指した人もあるし、フィルムカメラを改造して、自分のデジカメを作り上げた、つわ者もいます。本当に欲しいものがあれば、やる気になるというものです。よいアイディアが出たら、ベンチャーを起こしてビジネスにできるかもしれません。

私が最初に Raspberry Pi を利用したのは、自宅のプリンターサーバー用でした。一台のプリンターを家族と共有するのに便利に使えます。ところがスキャナ―機能を共用するのは難しいことでした。メーカーは Raspberry Pi の CPU である ARM 用のドライバーを提供してくれません。かなりアクロバティックな使いかたをしていましたが、結局インテル CPU を使った古いネットブックに置き換えました（いまのプリンターはサーバー機能を内蔵しています）。それで、制約なく Raspberry Pi を利用できるようになったので、改めてプロジェクトを始めました。

中級以上の工作では、長い間使い続けることを考えます。使っている間に問題が起こったり、もっと良くしたいという欲が出てきたりします。そのためには、修理や改造といった保守（メンテナンス）を考慮する必要があります。保守を視野に入れたアプローチをとったので、皆さんの参考にできる例としてまとめました。

1.2　温度コントローラを作る

やる気を出すには、自分の趣味や嗜好にあったものがいいと思います。燻製作りが私の趣味の一つです。50～70℃の煙の中に食材を数時間さらすと、ベーコンなどのおいしい燻製ができます。写真のような燻製室に 300W のニクロム線を張り、温度を調節します。最初のころは調光器を使って調整していたのですが、温度計を眺めながら、いちいちダイヤルを回すのは面倒です。

これを自動化しようというのが、最初の思い付きでした。ところで、そのための機材として、温度調節器とか温調計という名前の製品があり、安いものは一万円前後で手に入ります。それを自作しようというのは、記録を取ったり、遠隔操作したりという拡張をもくろんでいたからです。名前も温度コントローラにしました。

鶏肉を燻製する前に、コラーゲンが変性する 70℃で長時間調理すると、肉が柔らかくなります。コンフィというフランス料理は、カモ肉をラードで同じように調理します（今は、ジッパー付き袋にオリーブオイルと一緒に入れ、お湯で調理することが多い）。こういった用途にも使おうと思います。甘酒やヨーグルトを作るのにも役立ちそうです。

1.3　コントローラの仕様書を書く

温度コントローラを作るにあたって、目標を決めます。これは、「どうやって作るか」ではなく、「何を

実現するか」をまとめた仕様書（厳密には外部仕様書といいます）を作るということです。目標となる仕様が確定すれば、必要な要素（部品）が決められるし、それらをどう評価したらいいか、はっきりします（こちらは設計仕様書あるいは設計書）。後になっても、なぜそう作ったのか分かるし、改造する方法を探ることもできます。

まず、温度コントローラが何をするかを言葉で表現します。

項目	仕様
制御内容	温度制御（PID 制御 1 ループ）
制御対象	箱内の空気温度、鍋の水温
制御方法	加熱と自然冷却
制御出力	電気ヒーターを駆動する AC 電源（スイッチ出力）
可搬性	屋外、自宅以外での使用を考慮し、持ち運べること

次に、数値で表せる目標仕様を書き出します。無理をせず、実現可能で実用的な値を決めます。

項目	仕様
電源	AC100V 電源（ヒーター駆動電流＋0.1A）
ヒーター駆動	最大 AC100V/600W
周囲温度	0℃～40℃
目標設定温度	周囲温度+5℃～100℃
制御温度精度	±1℃
静定時の変動	±0.2℃
最速制御周期	1 秒（2 秒、5 秒、10 秒なども可能）

その次は実現する機能を定義します。オプションとなっているのは、なくても温度コントローラとして使えるが、より便利に使えるようにする機能です。

項目	仕様
表示機能	文字表示：制御モード、パラメータ名 数値表示：制御変数（温度、制御目標値、操作量）
操作機能	キースイッチによる操作 （制御変数設定、モード設定、チューニング、停止）
記録機能	コントローラ内記録、（オプション）ネットワーク記録
異常処理	温度読み取りエラーの影響を極力排除し、異常は表示する
リモート操作機能	（オプション）ブラウザへの制御状態表示とブラウザからの操作

表示と操作の仕様はもう少し詳しく説明します。操作するのは、増加（▲）キー、減少（▼）キー、モードキーの三つです。制御操作中は、▲キーと▼キーで、設定値や加熱量を増減させます。モードキーを押すと、▲キーで制御の運転モード（O/S、MANUAL、AUTO）を選べるようになります。再びモードキーを押すと、そのモードで制御が始まります。選択中に▼キーを押すと、PID 制御のチューニングパラメータが設定できるようになります。さらにモードキーを押すとプログラムの停止とシステムのシャットダウンが選べるようになります。さらにモードキーを押すと、制御状態表示に戻ります。

最後に外部とのインターフェースを定義します。

項目	仕様
外形	10cm×20cm×5cm に収まる絶縁ケースに収納
AC 電源	AC ケーブルをインレットプラグに接続
制御出力	ヒーターを AC コンセントに接続
温度センサ	コネクタで差し替え可能
通信	WiFi（無線 LAN）経由：ssh、（オプション）ブロードキャスト、ブラウザ経由操作

ここまでは仕様の概要です。ハードウエアとソフトウエアの詳しい仕様は、次章以降にまとめてあります。仕様書の初版に、ソフトウエアモジュールの構成と検証（手順と検証用データ）を含めたら、約 30 ページになりました。大部分はこの本のなかに転載してあります。

1.4　ソフトウエアの開発にむけて

温度コントローラの機能を作りこむ作業の大部分は、ソフトウエアの開発です。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。この本でかけている分量からも、それを感じ取ってください。

Raspberry Pi で GPIO の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 で記述するようにしています。オプション機能で Web サーバーを設計するときだけは、JavaScript と HTML を使わざるを得ませんでしたが。

この本で使用しているのは Python 2.7 です。サポートの終了が近づいているので、Python 3 に切り替えても良いと思います。掲載しているプログラムで影響を受けるのは、入出力（print と raw_input）とシステムコール（exec）くらいだと思いますが、確認はしていません。

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ことを、よく見かけます。しかし、ここではそれを、できるだけ避けるようにします。また、先に述べたソフトウエアの保守性を良くするため、Linux に備わったツールを使っています。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、あえて採用しました。じっさい、これなしには、こんな大きなソフトウエアは作れなかったというのが正直な感想です。

インターネット上では、Raspberry Pi に限らず、いろいろな製作（試作？）記事が見つかり、参考になるものが多くあります。いっぽう、本人には当たり前のこととして背景の説明がなかったり、なぜか分からないけど、こうしたら動いたとなっていたりといった、自分の理解に役立ってくれない記事もあります。この本では、できるだけ理解に役立つ記述を心がけました。理解していないと、保守ができないからです。それでもカバーしきれない個所は、そう書いてあるので、他の資料にあたってください。

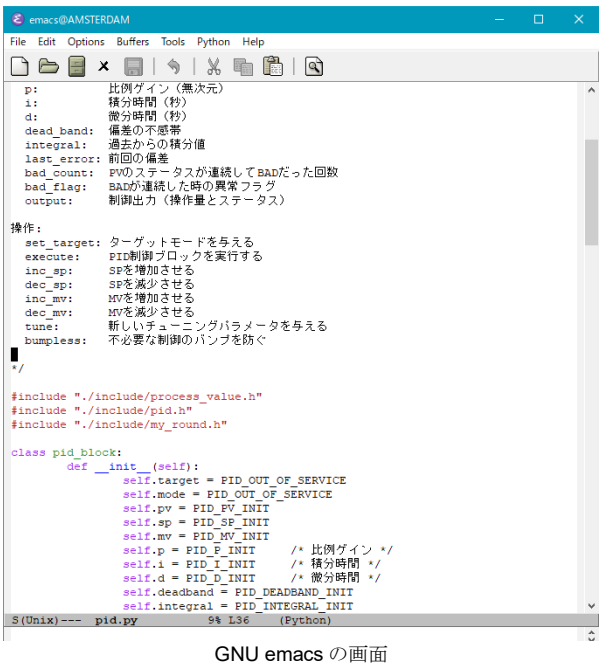
1.5　プロジェクトに必要な機材

このプロジェクトに必要な機材（温度コントローラ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、最終的にはケースに収納します。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。PC（パソコン）は必須です。

最初に Raspbian をダウンロードするときから、PC を使います。あまり高機能な機種である必要はありません。OS 以外で最低限必要になる機能は、プログラムの作成に使うエディターと、Raspberry Pi に接続する SSH クライアントです。Windows PC の場合は、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。



GNU emacs の画面

私の場合は、PC として、一万円ちょっとで購入した中古ノートに、Linux の一種である Ubuntu OS を載せたものを使いました。Raspberry Pi と同じ Linux なので、ソフトウエアの作成から、検証の大部分までを、この上で行えるのが最大の利点です。Raspberry Pi 上でも同じことができますが、SSH 接続したターミナル画面ですべてを行う必要があります。検証結果を表計算ソフトで処理したりするのは、PC 上の方が圧倒的に有利です。エディターとしては、emacs や vi があるし、コンソール画面から直接 SSH 接続ができます。

また、PC にソフトウエアのバックアップを取っておくことを、お勧めします。

Windows と Linux の間でプログラム（テキスト）ファイルをやり取りするときに、困ることが二つあります。一つ目は日本語の文字コードです。Linux は UTF-8、Windows は Shift-JIS を使っているので、異なる文字コードを表示できなくなる場合があります。二つ目の問題は一行の終わりのコードで、Linux では newline、Windows では Carriage-Return と Line-Feed の二文字になっており、これも読み取りにくい時があります。例えば Windows の notepad（メモ帳）は、UTF-8 の日本語を含むファイルを正しく表示できますが、newline（実は Line-Feed と同じ）だけのファイルは、改行で左に戻って表示してくれません。この対策は付録で説明します。

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。誤った設定をしてしまうと、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 B モデルなどでは、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。

測定器

オシロスコープやロジックアナライザがあれば、ハードウエアの試験には理想的ですが、それが必要になるのは、かなり重症な時です。それより、誤りなく配線することを心がけましょう。回路の短絡や断線を調べるのは、テスターで足ります。

温度センサがちゃんと動いているか知るのに、別の温度計があると便利です。気温計でもいいし、ガラス管に入った温度計があれば使います。

UNIX、Linux、 Raspbian

コンピュータのオペレーティングシステム（OS: 日本語では「基本ソフト」ともいう）のお話です。黎明期のコンピュータはあまり高性能でもなく、また高価なものだったので、できるだけ多くの処理を効率よく実行させる必要がありました。多くのユーザープログラムを（ほぼ）同時に実行する環境として OS（タイムシェアリングシステム）が登場しました。

1960 年代中ごろ、産学連携でメインフレーム（大型コンピュータ）の OS を開発するプロジェクトが、アメリカで始まりました。Multics という歴史的な OS ですが、あまりに多くの機能を組み込んだのと、ハードウエアの性能が追い付かなかったため、目論んでいたほど普及しませんでした。

産学連携プロジェクトから離脱したベル研究所の、ケン・トンプソンは同僚のデニス・リッチーと独自の OS である UNIX を開発しました。Multics の反省からミニコンピュータ（とは言っても、性能は現在のスマートフォンにも及ばない）でも使える OS を考えたのです。OS 自体の機能を絞り込み、ほかに必要となる機能は独立したソフトウエアで実現した単純な OS だったので、マルチ・ックスに対して、ユニ・ックスと名付けたということです。最初は他の OS と同様、アセンブリ語で開発されたのですが、すぐに C 言語で書き直されたので、他のコンピュータへ移植するのが比較的簡単にできるようになりました。

ベル研が、研究者向けにはソースコードを安価に提供したので、大学や研究所に普及していきました。特にカリフォルニア大学バークレー校で熱心な改良がされ、バークレー・ソフトウエア・配布版（BSD）として多くの研究者に使われました。しかし、ベル研の親会社であるアメリカ電信電話会社（AT&T）がライセンス料を要求したり、標準化をめぐって業界が対立したりして、普及が進まなくなってしまいました。

90 年代の初め、ヘルシンキ大学の学生だったリーナス・トーバルズは、教育用 UNIX の PC 移植を考えるなか、ライセンス問題などに嫌気を感じていました。そこで、外部仕様はそのままに、自前（カーネル部分）のコードを作ってしまったのです。これが Linux の始まりです。フリーウエアとして公開することで、多くの開発者が参加し、急激に成長していきました。

Linux のカーネルに、追加のソフトウエアと、デスクトップ環境などの様々なパッケージを付加し、使い勝手を良くしたうえで配布するものを、ディストリビューションといいます。カーネル自身は無償ですが、有償で提供するディストリビューションがあるのは、追加分が有償だからです。ディストリビューションの一つが Debian 系と呼ばれるグループで、パッケージを deb 形式で管理するのが特徴です。Ubuntu も Raspbian も、無償の Debian 系ディストリビューションです。

2　ハードウエアの準備

2.1　必要なハードウエア

最初に、温度コントローラを実現するのに必要なハードウエアをまとめておきます。購入する電子部品、回路ユニットは、東京の秋葉原にある秋月電子通商（http://akizukidenshi.com/）で調達しました。秋葉原に行くことのできない人は、通信販売で購入するか、この節で説明する仕様と同じようなものを調達してください。ケースなど、電子部品・ユニット以外のものは 100 円ショップでも入手できます。

名称	調達方法
Raspberry Pi ZERO WH	購入
マイクロ SD カード	購入
半導体リレーユニット	購入
液晶表示ユニット	購入
操作ユニット	自作
インターフェースユニット	自作
温度センサユニット	自作
部品・部材	購入

必要なハードウエア

部品の調達や組み立てには時間がかかるので、次章以降のソフトウエア開発と並行して行うといいでしょう。組み立てたハードウエアの試験には、開発したソフトウエアモジュールを使うと便利なので、いきなりハードウエアの電源を入れたりせずにソフトウエアモジュールの検証まで待ちます。ただし、Raspberry Pi の立ち上げだけは先に行えます。

それぞれのハードウエアに求められることを、以下に説明します。

Raspberry Pi ZERO WH

Raspberry Pi のなかでは、一番小型で安価な Raspberry Pi ZERO を使いました。温度コントローラのように、あまり早い制御が求められない用途には、十分な性能を持っています。もっと能力の高い B+などを持っている場合には、それを使っても問題ありません。

WiFi は必須です。GPIO コネクタが装着されている WH モデルが使いやすいのですが、W モデルに自分でピンヘッダーをはんだ付けすることもできます。

マイクロ SD カードとアダプター

Raspberry Pi にインストールして使う Raspbian（Linux の一種）の大きさは、最小モデル（Lite）で 2GB 弱、フルセット（Desktop+推奨ソフト）でも 5GB 強です。マイクロ SD カードは、容量が 8GB あれば十分です。しかし、容量の小さいものは、かえって手に入りにくくなっています。手持ちのものか、入手可能で一番安いものがいいでしょう。

PC から書き込むときには、外形が大きい SD カード型のアダプターに入れて使います。アダプターはマイクロ SD カードを買ったときに付属してきます。

半導体リレーユニット

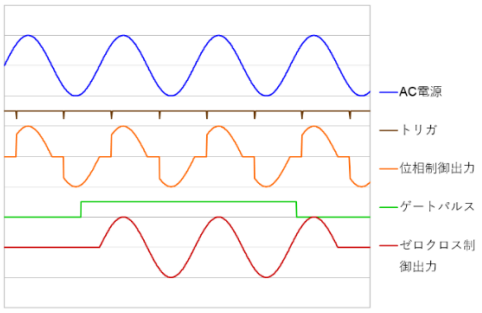
温度コントローラでヒーターを制御するには、AC 電源をオンオフする素子が必要です。この制御方法には、位相制御とゼロクロス制御の二つがあります。

位相制御とは、交流電圧がある電圧（位相）になったときにトリガを発生し、出力をオンにする方式です。次ページにある図の一番上が AC 電源波形、その下がトリガ、その下がスイッチング素子を介して負荷に伝わる電圧です。トリガが発生すると負荷側に電圧が現れ、AC 電圧がゼロになると消えることが分かります。電力を連続的に制御できるので、電球の明るさを調整するのに向いています。

いっぽうゼロクロス制御では、ゲートパルスが与えられてから AC 電源が 0 ボルトになるとオンになり、パルスが消えた後の 0 ボルトでオフになります。下から 2 番目がゲートパルス、一番下が負荷電圧です。パルス幅で制御できるので、Raspberry Pi から使いやすいのです。こういう回路を半導体リレーといいます。

制御方式	特徴
位相制御	連続的に変えられる
ゼロクロス制御	ゲートパルス幅でオンオフ

AC 電源の制御方式



位相制御とゼロクロス制御の波形

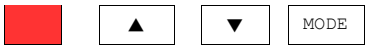
液晶表示ユニット

表示器には 4 つのパラメータとパラメータ名を表示するため、16 文字×2 行は欲しいところです。下のように横方向に 4 つのフィールドを設定し、上の行にパラメータ名を、下の行にパラメータの値を表示するようにしました。8 ビット並列バスに接続できるタイプもありますが、Raspberry Pi から扱いやすい I2C バス型を選びます。

	"PV"	"SP"	"MV"
PID モード	温度	目標値	操作量

操作ユニット

人が手で操作するのに使うもので、用途に合わせて自作します。入力数は多いほど便利ですが、操作するときに目移りしても困るので、最低限必要なものに限ります。普段使うのはパラメータの増減と、調整するパラメータを選ぶ入力だけなので、3 つのキーに絞ります。



何らかの異常が起こったとき、すぐ操作者に伝わるように、操作ユニットのなかに赤色警報灯をつけておきますキーと警告灯の配置を上図のようにしておくと、操作するときにすぐ目につきます。

ケースに入れて長期間使うので、ブレッドボードは使わず、ユニバーサル基板にはんだ付けしました。

インターフェースユニット

Raspberry Pi と他のユニットを繋ぐためのユニットを自作します。搭載する部品の多くはコネクタですが、I2C バスリピーターと LED ドライバーはここに入れておきます。

先日、Raspberry Pi で使える、液晶表示器とタッチスイッチが搭載されたユニットが発売されました。これを使用するときは、液晶表示器ドライバーと割り込み処理を差し替えるだけで済むはずです。

温度センサユニット

温度センサは、I2C バスでインターフェースできるものを選び、使う環境に合わせて二種類を自作しました。コネクタで交換できるようにしておきます。センサを裸で使うわけにはいかないので、環境からの保護と熱接触を目的とした工作をします。

部品・部材

各ユニットの説明には出てきませんでしたが、それ以外にも必要な資材があります。

まず、温度コントローラを収納するケースが要ります。表示器が外から見えること、ユニットの取り付け加工が簡単なこと、絶縁材でできていることが望ましいので、アクリル樹脂製の箱がいいでしょう。私は 100 円ショップで名刺入れとして売っていた、写真のような箱（200 円でした）を使いました。取り付け用のネジ（プラスチックネジも含む）やスペーサも忘れてはいけません。内部配線用の線材が必要ですが、フラットケーブルを使うと組み立てと分解が容易になります。

ケースの例

AC 電力を扱うので、電源コード、制御出力用コンセント、フューズボックスが必要です。AC を流す電線は、内部配線を含めて、耐圧 300V 以上、電流 10A 以上のものを使います。AC 電源の接続部が露出していると危険なので、ビニルテープと自己融着テープ、熱収縮チューブなどを使って絶縁します。Raspberry Pi に 5V の電源を供給する AC アダプター（500mA 程度）も用意しておきます。

温度センサ用に、コネクタ、放熱器、ガラス管、グリス、接着剤などを調達します。I2C バスの信号数（4 本）を扱えるなら、コネクタに制限はありません。私は、比較的大型で工作しやすい、PS2 マウス

用のコネクタを使いました。特殊な部材として、湯温センサに詰めるため、粉末シリカを DIY 陶芸用品店で手に入れました。

試験用恒温槽


試験用恒温槽

恒温槽というと大げさかもしれませんが、動作試験には温度をコントロールする対象が必要です。ヒーターを断熱材で囲ったものですが、試験用なので簡単に手に入るものがいいです。今回は 40W の電球を板に取り付け、牛乳パックを被せたものを用意しました。牛乳パックの底を切り開いて温度センサを挿入できるようにしています。

2.2 Raspberry Pi ZERO WH の準備

Raspberry Pi ZERO WH の準備をします。この準備は総合検証を始めるまでに済んでいればいいので、慌てて行う必要はありません。この段階では、AC アダプター、設定用の PC と WiFi 環境があれば十分です。Raspberry Pi を机の上で動かすときは、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。

インストールの方法には何通りかありますが、古くから使われているオーソドックスなやり方を採用しています。

2.2.1Linux のインストール

まずオペレーティングシステム（OS）を立ち上げます。Raspberry のダウンロードサイト（https://www.raspberrypi.org/downloads/）から Raspbian をクリックして、一番軽量の Raspbian Lite（ZIP ファイル）をダウンロードします。2019 年 2 月の時点の最新版は、Raspbian Stretch（カーネルバージョン 4.14）でした。解凍後のイメージは SD メモリ上で 1.9GB を占めます。

PC でイメージファイルを右クリックし、書き込みソフトを起動します。Ubuntu PC の場合はディスクイメージライター、Windows10 では「ディスクイメージの書き込み」を選びます。書き込み先には SD カードを指定します。間違えてもハードディスクに書き込まないようにしてください。

数分で書き込みが終わり、SD カードが PC にマウントされ、中身が見えるようになります。/boot ディレクトリに 2 つのファイルを書き込みます。あらかじめ PC 上で用意しておくと簡単です。最初のファイルは ssh という名前の空ファイルです。このファイルがあるとシステムは自動的に、PC からの SSH 接続をできるように設定します。Windows でファイルの拡張子を表示しない設定になっていると、ssh.txt などのファイルが ssh と表示されてしまうので注意してください。拡張子のない ssh が正しいファイル名です。

つぎに WiFi 環境を指定する wpa_supplicant.conf というファイルを作ります。WiFi のセキュリティが WEP 方式の場合は、次のような内容にします。セキュリティについては、WiFi 環境を設定した人に確認してください。なお、インデント（字下げ）にはスペースではなく、タブを使います。自分の SSID と WEP のパスワードは WiFi 環境に合わせた文字列にしてください。

```
ctrl_interface=DIR=/var/run/wpa_supplicant
GROUP=netdev
update_config=1
network={
   ssid="自分の SSID"
   key_mgmt=NONE
   wep_key0="WEP のパスワード"
   wep_tx_keyidx=0
   auth_alg=SHARED
}
```

WiFi のセキュリティが WPA 方式の場合は、次のような内容にします。自分の SSID と PSK パスフレーズは WiFi 環境に合わせた文字列にしてください。

```
ctrl_interface=DIR=/var/run/wpa_supplicant
GROUP=netdev
update_config=1
network={
   ssid="自分の SSID"
   proto=WPA
   pairwise=CCMP
   key_mgmt=WPA-PSK
   psk="PSK パスフレーズ"
   auth_alg=OPEN
}
```

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 の端末ソフトから 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
```

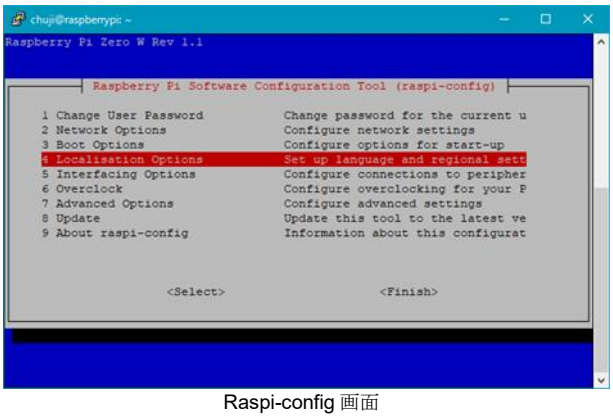
Raspberry Pi にログインできましたか？　すぐにパスワードを変更しろというメッセージが表示されるはずです。私は Ubuntu PC と同じユーザ名を使いたいので、自分の苗字と名前をグループ名、ユーザ名として登録しています。以下で最初のコマンドはグループ akiyama の追加、二番目はユーザ chuji をグループ akiyama に追加してホームディレクトリを作成、三番目は chuji のパスワード設定、四番目はログイン時のシェルに bash を選択、五番目は chuji に sudo を実行する権限を与えています。説明の都合で、私の名前を使いましたが、皆さんは自由に選んでください。

```
$ sudo groupadd akiyama
$ sudo useradd --create-home --gid akiyama chuji
$ sudo passwd chuji
Enter newUNIX password:
Retype new UNIX password:
passwd: password updated successfully
$ sudo usermod -s /bin/bash chuji
$ sudo usermod -G sudo chuji
$ exit
```

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。まずは raspi-config を起動して、表示を日本語に切り替えましょう。

```
login as: chuji
chuji@192.168.15.26's password:
Linux raspberrypi 4.14.79+ #1159 Sun Nov 4 17:28:08
GMT 2018 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: Wed Feb 20 16:51:35 2019 from
192.168.15.24

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、右上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 で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 を押します。



Raspi-config 画面

4 Localisation Options から I1 Change 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>します。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 I2 Change 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 5 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から、raspi-config を終了します。

なお bash を使っていると、左下の例のように(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 $ だけを表示しておきます。

2.2.2WiFi 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、サーバーとして使うためには固定しておきたいと思います。vi でも nano でも良いので、使い慣れたエディターで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。192.168.15 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以外で選びます。

```
# added by chuji on 2019/2/20

interface wlan0
static ip_address=192.168.15.16/24
static routers=192.168.15.1
static domain_name_servers=192.168.15.1
```

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 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。

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。LED が消えたら、AC アダプターの電源も切っておきます。

OS のカーネルとシェル

OS のなかで、カーネル（核）と呼ばれる部分は、次のような機能を提供することで、ユーザープログラムとハードウエアを仲介しています。

- マルチタスク（複数のタスクを走らせる）を実現する、プロセス管理
- 各タスクにメモリを割り当てる、メモリ管理
- ハードウエアのデバイスドライバー（とのインターフェース。割り込みを含む）
- ディスク管理
- TCP/IP などの通信

組込みシステムでは、ディスク管理と通信を必要としないものもあります。

カーネルに含まれないソフトウエアの代表格が、シェルと呼ばれる、ユーザの命令を実行するソフトウエアです。核を取り巻く殻（シェル）というイメージですね。いろいろなシェルが開発されましたが、いまは bash が一番使われていると思います。

2.2.3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。最初に現在のパッケージ情報を更新しておきます

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

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 4.5.16-Debian
$ sudo vi /etc/samba/smb.conf
```

ファイルの先頭（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。日本語の文字セットは以下のようになっているはずです。

```
Unix charset = UTF-8
dos charset = CP932
```

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に/home/chuji の中身が表示されるようになります。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。

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。python-smbus は I2C バスを使うためのライブラリ、pip は Python のパッケージ管理をするツール、python-dev は開発用のパッケージです。netifaces はネットワークインターフェース（つまり WiFi）の情報を得るためのパッケージで、pip を使ってインストールします。

```
$ sudo apt-get install python-smbus
$ sudo apt-get install python-pip
$ sudo apt-get install python-dev
$ sudo pip install netifaces
```

NODE.JS

これ以後のパッケージは、ブラウザからの操作を可能にする Web サーバーに必要なものです。後で必要になった時にインストールしても構いません。Web サーバーが必要ない方は読み飛ばしてください。wget や tar の詳細な説明は省略しています。

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 用のパッケージサイトで提供されているのは Raspberry Pi 2 以降用の CPU 向けなので、Raspberry Pi ZERO や Raspberry Pi 1 では、apt-get による標準的なインストール方法が使えません。直接 Node.js Foundation から最新版をダウンロードしました。他の CPU についても以下の armv6l を armv7l とか arm64 に置き換えれば、同じようにインストールできます。

なお、この方法はグローバルインストールといって、すべてのディレクトリから Node.js を使えるようにしています。この外にローカルインストールといって、自分のホームディレクトリ下にインストールして、同じコンピュータ上で異なるバージョンの Node.js を使うようにもできます。

まず https://nodejs.org で Node.js の安定バージョン（LTS: long term support）を調べます。この時点では、LTS ステータスにあったのは 10.16 でした。PC のブラウザで、https://nodejs.org/dist/にアクセスします。ディレクトリ一覧が表示されるはずです。ここで latest-v10.x/のディレクトリ下にあるファイルのうち、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。この本の執筆時点では node-v10.15.3-linux-armv6l.tar.gz が最新でした。ダウンロードしたものを、smb 経由で Raspberry Pi の controller 以外のディレクトリ（私の場合は~/tmp）に転送します。または、Raspberry Pi に SSH 接続して以下のコマンドを実行しても同じことです。

```
$ wget https://nodejs.org/dist/latest-v10.x/node-v10.15.3-linux-armv6l.tar.gz
```

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、以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-v10.15.3-linux-armv6l.tar.gz
$ cd node-v10.15.3-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 --version
V10.15.3
$ npm -version
6.4.1
```

Node.js のバージョンがあまり新しいと、次の Socket.IO のインストールができないときがあります。その時は、上の手順で Node.js のバージョンを古いものと交換してみてください。LTS なら大丈夫だとは思いますが。

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。

Socket.IO は npm を使ってインストールします。-g オプションは、グローバルインストールを指定しています。

```
$ sudo npm install -g socket.io
+ socket.io@2.2.0
added 45 packages from 33 contributers in 24.835s
```

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。ここでは 、そのごく一部である FIFO（First-In-First-Out）機能を使って、Web サーバーと Python プログラムとの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採用しました。

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 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インストールします。

```
$ sudo pip install redis
$ sudo npm install -g redis
```

ブレッドボードとはんだ付け

この本の電子工作では、部品やケーブルを穴に差すだけのブレッドボードは使わず、はんだ付けをしています。出来上がった回路を長く期間使うことと、移動もすることを考慮し、誤接触や接続不良を防ぐためです。また、AC 電源を扱う回路では、誤接触が起こると非常に危険です。金属露出部は、絶縁テープや熱収縮チューブで覆うようにしています。回路ユニットを、ネジなどでケースに固定するのも同じ理由です。フラットケーブルはピン数を多くして、抜けにくくしました。

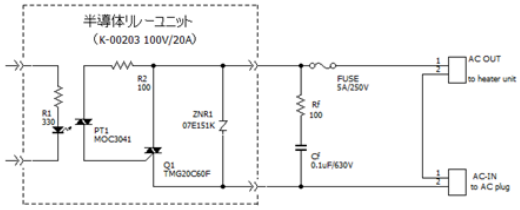
2.3　半導体リレーユニット

電気ヒーターの消費電力は最大 600 ワットなので、オン時の電流は 10 アンペア以下です。そこで 20～30A の電流を流せる半導体リレーを、秋月電子通商で探したら、以下の仕様のキット（K-00203）が見つかりました。もっと電流を流せるものもあります。実際に使ってみると、ほとんど発熱がなく、放熱器がなくても大丈夫そうでした。もちろん放熱器は取り付けましたが。

項目	仕様
V_{DRM}（尖頭阻止電圧）	600V
$I_{T(RMS)}$（実効オン電流）	25A
（非放熱時）	2A
入力	フォトカップラ

半導体リレーユニット仕様

この半導体ユニットの内部回路と、周辺回路を下図に示します。周辺回路はキットで推奨されている、ノイズノイズフィルタ用の抵抗とコンデンサ、保護用フューズです。半導体リレーが常時オンになり続けることはほとんどないので、フューズの溶断電流は少なめにしましたが、後に 10A に変えました。



半導体リレーユニット回路図

フォトカップラには 330Ω の抵抗が直列に接続されているので、5V で駆動したときの電流は

（5V – 0.6V) ÷ 330 = 13mA

程度になります。0.6V はダイオードの順方向電圧降下です。これを駆動するため、インターフェースユニットにトランジスタ回路を設けます。

ところで AC 電源が 0 ボルトになるのは、電源周波数が 50 ヘルツの地域では一秒間に 100 回、60 ヘルツの地域では 120 回しかありません。ゲートパルスが AC 電源と同期していると、制御分解能が悪くなるので、意図的に周期を変えてやるようにします。

AC 電源を扱う回路の電線は、絶縁耐圧（300V 以上）と電流容量（10A）を満たすものを使います。ノイズフィルタ（スナバ回路）はなくても動作しますが、抵抗は電力 1W、コンデンサの耐圧は 300V 以上のもの（ここでは 630V のフィルムコンデンサ）で作りました。

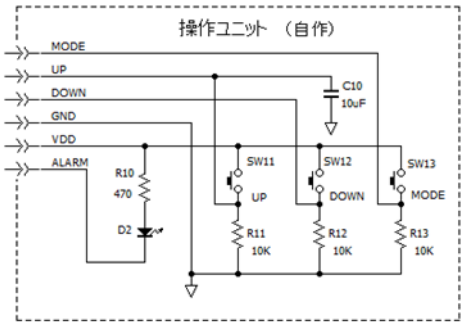
2.4　液晶表示ユニット

16 文字×2 行が表示でき、I2C バスで制御できる薄型の表示ユニット（AE-AQM1602A）を使うことにしました。このキットには、入出力ピンを 2.54mm ピッチに変換するプリント基板が付属しており、フラットケーブルで接続できます。

なお、このユニットは Raspberry Pi の I2C バスの終端抵抗（1.8kΩ）を直接ドライブできないので、インターフェースユニットに搭載した、推奨回路のバスリピーターPCA9515AD を介して接続します。

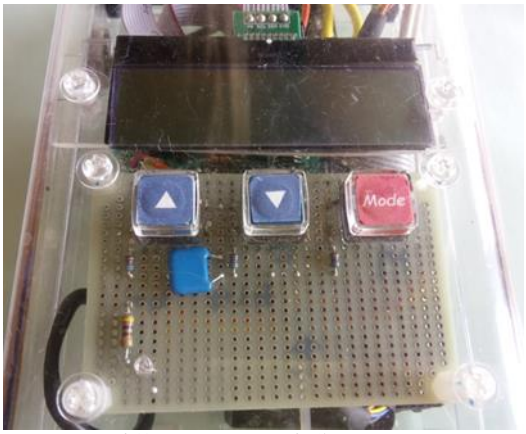
2.5　操作ユニット

人の手による操作を受け入れるため、3 個のキー入力を用意します。ユニバーサル基板にタクトスイッチを取り付け、ケースの前面に設置します。異常時のアラームを示す LED も取り付けておき、全体をフラットケーブルでインターフェースユニットに接続します。



操作ユニット回路図

液晶表示ユニットと操作ユニットは、ケースの前面に取り付けます。ケースが透明なので、液晶画面がよく見えます。操作ユニットのタクトスイッチには、▲と▼、それに MODE と印刷したキートップ（カバー付き）を取り付けました。



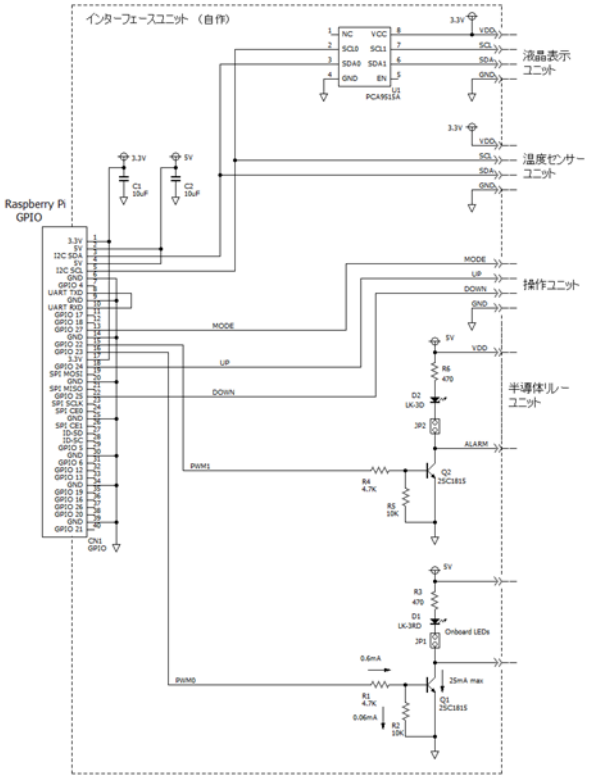
液晶表示ユニットと操作ユニット

2.6　インターフェースユニット

Raspberry Pi との接続を簡単にするため、秋月電子通商が設計した Raspberry Pi 用ユニバーサル基板（AE-RasPi-Universal）を利用しました。背の高い接続用の 40 ピンのソケット（MFH2X20SG-2）を取り付け、二階建てにすることができます。取り付けには、長さ 15mm の樹脂スペーサを使いました。普通のユニバーサル基板でも同じことができます。最近になって、秋月電子通商から Raspberry Pi ZERO 用のユニバーサル基板（AE-RasPi-Universal-Zero）が発売されました。外形が半分になるので、細かい細工ができる人には向いています。

その他のユニットとの接続用にピンヘッダーを立て、フラットケーブルでつなげるようにしました。

液晶との接続コネクタとの間に、I2C バスリピーター（PCA9515）を取り付けます。



インターフェースユニット回路図

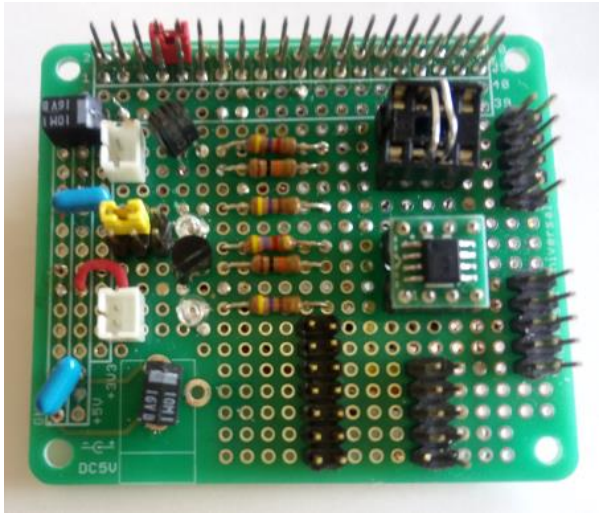
半導体リレーのフォトカップラや警報 LED を駆動するため、NPN トランジスタで 2 回路のドライバーを作ります。基板上の LED の駆動に 10mA、フォトカップラ駆動に 13mA 必要なので、トランジスタのコレクターには 25mA 以上流せるようにします。トランジスタは小信号用なら、ほとんど何でも使えますが、念のため必要な特性と、使用したトランジスタ（2SC1815GR）の特性を表にしておきます。

特性	必要な特性	2SC1815GR
耐圧（V_{CE}）	10V 以上	50V
定格電流（I_C）	50mA 以上	150mA
直流増幅率（h_{FE}）	100 以上	350

トランジスタの特性

ベース電流は、最大コレクター電流(25mA) ÷ h_{FE}(100) = 0.25mA 以上必要なので、余裕を見て 0.6mA を流します。ベース抵抗は、R1 = (駆動電圧(3.3V) – ベース・エミッタ電圧(0.6V)) ÷ 0.6mA = 4.7kΩ。駆動電圧がない時にトランジスタをオフにするため、ベースを抵抗でプルダウンしておきます。ベース電流の 10%を分流するとして、 R2 = ベース・エミッタ電圧(0.6V) ÷ 0.06mA = 10kΩ。

はんだ付けは、ショートやイモはんだを起こさないように注意して行います。Raspberry Pi 用ユニバーサル基板は、電源やグランドの配線ができているので、はんだ付け個所が少なくて済みました。実物を下の写真に示します。ほとんどピンヘッダーのかたまりですね。実験用に、温度コントローラでは使わないピンヘッダーも取り付けているためで、温度コントローラ専用ならもう少し空いたものになります。



インターフェースユニット

2.7　温度センサユニット

温度センサは二種類用意しました。一つは燻製などに使う、空気温度を測るものです。もう一つは、低温調理用の、湯温を測るものです。I2C バスを介してデータを得られる IC を使おうと思います。空気用は、多くの人が使いこなしている、アナログデバイス社製の ADT7410A（8 ピン SOIC パッケージ入り）で作りました。湯温用はガラス管に封入するので、内径に入る大きさの、ST マイクロエレクトロニクス社製 STTS751（6 ピン SOT パッケージ入り）を選びました。おもな仕様は下表のとおりです。

項目	ADT7410A	STTS751
外形	8 ピン SOIC	6 ピン SOT
動作電圧	2.2V~5.5V	2.2V~3.6V
測定範囲	-55℃~+150℃	-40℃~+125℃
分解能	13, 16 ビット	9-12 ビット
精度	±0.4℃	±1.0℃

温度センサ仕様

両者のピン割り当てを下表にまとめます。

ピン	ADT7410A	STTS751
1	SCL	Add/Term
2	SDA	GND
3	A0	V_{DD}
4	A1	SCL
5	INT	EVENT/
6	CT	SDA
7	GND	-
8	V_{DD}	-

温度センサのピン割り当て

ADT7410A の I2C バスアドレスは A0 ピンと A1 ピンで、STTS751 のアドレスは Addr/Term ピンをプルアップする抵抗値で設定します。STTS751 にはアドレス設定の異なる二種類がありますが、通常手に入るのは STTS-751-0 の方です。STTS751 - 1 では、下表のアドレスに 0x02 を足します。両者とも同じ I2C バスアドレス（0x48: センサをもうひとつ使う場合は 0x49）に設定します。

アドレス	A1	A0	Addr/Term
0x48	0	0	7.5kΩ±5%
0x49	0	1	12kΩ±5%
0x4A	1	0	-
0x4B	1	1	-
0x38	-	-	20kΩ±5%
0x39	-	-	33kΩ±5%

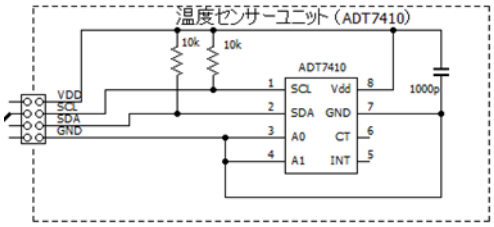
温度センサのアドレス設定表

信号用電線には、センサ付近の温度が 100℃近くなることから、耐熱温度 150℃のジュンフロン線を使います。IC のすぐ近くで SDA と SCL を終端（10kΩ でプルアップ）します。

高温部から十分離れたところからはポリプロピレン電線（耐熱温度 80℃）（あるいは耐熱ビニル電線（耐熱温度 75℃）でも可）で延長しました。SDA と GND、SCL と V_{DD} をそれぞれ撚り合わせ、それをまた撚り合わせておきます。

2.7.1空気温度センサ（ADT7410A）

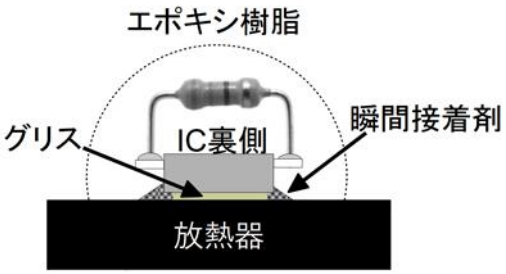
ADT7410A は Raspberry Pi の工作でもポピュラーなデバイスです。大事なのは、温度を測る IC チップの温度を周囲の空気温度に近づけることです。周囲との熱接触を良くするとともに、気流の影響を避けるだけの熱容量を確保するため、一辺 2~3 センチの放熱器に取り付けます。放熱器の温度が空気温度に近づくので、それに接触している IC チップの温度もそれに近づくというわけです。回路図は以下のとおりです。



温度センサ（ADT7410A）の回路図

まず樹脂に封止された IC の上（印刷のある）面を、放熱器に接触させます。熱接触を良くするため、上面には伝熱グリス（パワートランジスタや CPU を放熱器に取り付けるときに使うものです）をごく少量だけ塗ります。ここで「ごく少量」というのがみそで、多くては熱がよく伝わらないし、IC の周りにはみ出してしまいます。擦りつけないように気をつけながら、IC を放熱板に密着させ、縁に瞬間接着剤を垂らして仮固定します。つぎに空中配線で抵抗とジュンフロン線を接続します。IC の温度を上げすぎないよう、短時間ではんだ付けしてください。イモはんだになっていないことを確認したら、動作テストを行っておきます。

温度センサが動作することを確認したら、絶縁と固定を兼ねて、エポキシ樹脂（実際には二液型のエポキシ接着剤）で全体を被ってやります。エポキシ樹脂は熱に強いと言われていますが、標準品の使用温度は 80℃くらいです。高温型と呼ばれるものは 150℃まで使えるので、こちらを選びます。



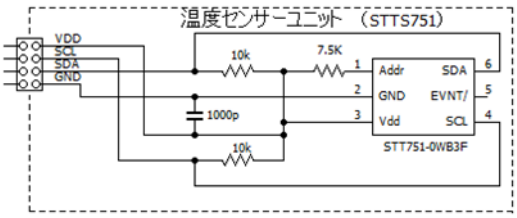
空気温度センサの組み立て

出来上がりの写真を下に示します。燻製の煙で樹脂が茶色くなってしまいました。

出来上がった空気温度センサ

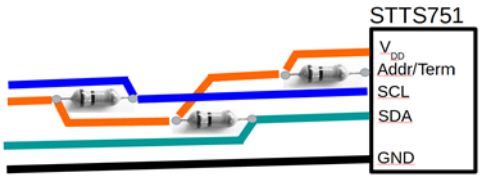
2.7.2湯温センサ（STTS751）

STTS751 は 7.5kΩ のプルアップ抵抗で I2C バスアドレスを 0x48 に設定します。回路図は以下のとおりです。



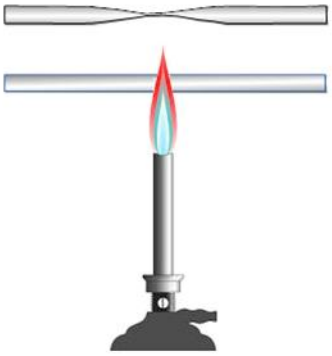
温度センサ（STTS751）の回路図

STTS751 は超小型のパッケージに入っているので、組み立てには気を使います。また、接続部の絶縁も難しいので、ジュンフロン線との接続部が、長さ方向に離れるようにします。次ページの図を参考にしてください。内径 4 ミリのガラス管に入れるので、はんだ付け個所などに出っ張りができないように注意します。この段階で動作テストを済ませておきましょう。



湯温センサの空中配線

センサ部を湯の中に浸けるので、ガラス管に封入します。まず外径 6 ミリ（内径 4 ミリ）のガラス管の中央付近をバーナーで加熱し、柔らかくなったら、両側に引いて伸ばします。えい、やあ、と引っ張ると、中央が細く伸びます。図の下の方が過熱している様子、上が引き伸ばしたガラス管です。



ガラス管の引き伸ばし加工

細い部分を折ってから、もう一度バーナーで加熱すると、穴がふさがります。柔らかいうちに反対からそっと吹くと、先端が少し丸くなります。風船のように膨らます必要はありません。バーナー加熱は火事につながりやすいので、ガラス加工の経験がない人は、決して一人ではやらないでください。経験者に教わるか、観光地のガラス細工体験コーナーで相談してみてください。ガラス加工ができないときは、細いガラス試験管を使うこともできます。

配線したセンサを先頭にして、ガラス管にゆっくり押し込んでいきます。はんだ付けした箇所がお互いに離れていないとショートしてしまうので、じゅうぶん気を付けて行います。その時の様子を下の写真（横倒しになっている）に示します

液温センサをガラス管に挿入する

このままではガラス管の外側とセンサ IC の間に空気が入っているので、熱接触が良くありません。充填材を詰める必要がありますが、充填材は熱伝導が良く、電気的には絶縁性のものを選びます。ところが、熱伝導度と電気伝導率は、一方が大きければ他方も大きい傾向があり、選定は難しいのです。右上の表に主だった材料の電気伝導率（大きいほど電気を良く通す）と熱伝導度（大きいほど熱が良く伝わる）を電気伝導率が大きい順にまとめました。特性は組成や含水量によっても変わるし、一部は他のデータから換算したものなので、数値は目安だと思ってください。電気伝導度は数値の範囲が広いで、指数表現（3E-5 = 3×10^{-5} など）を使っています。

材料	電気伝導率 (s/m)	熱伝導度 (W/(m・K))
銅	6E+7	400
ステンレス	1E+6	0.02
炭素	3E+4	150
水（液体）	4E-6	0.6
メタノール	2E-7	0.2
シリコンオイル	1E-10	0.1
シリコンゴム	1E-11	0.2
エポキシ樹脂	1E-13	0.3
酸化アルミニウム	1E-13	28
酸化マグネシウム	1E-13	50
空気	6E-14	0.02

電気伝導率と熱伝導度

データを見ると、酸化マグネシウムがよく両立できる材料であることが分かります。実際、工業用温度センサの充填材によく使われますが、入手は簡単ではありません。そこで次善のものとして、酸化アルミニウム（アルミナ）を使うことにします。陶磁器の釉薬（ガラス質のコーティング剤）や研磨剤に使う材料なので、陶芸用品店や工作材店などで手に入ります。細かい粉末なので、誤って吸い込まないようにしてください。マスクの着用をお勧めします。

アルミナ

アルミナの粉末は絶縁性が良いため、静電気を帯びやすく、ガラス管の中に入れるのは少々難しい。こぼれないように注意しながら、少しずつ入れていきます。そのとき空気を巻き込んでしまいます。空気は熱伝導が悪く、温度センサの特性に影響するので、抜いてやる必要があります。本格的な真空ポンプが使えれば理想的ですが、私はアルミナを詰めたガラス管をビンに入れ、ワイン用の簡易真空ポンプで減圧しました（写真を参

空気抜き

照)。ビン内の気圧が下がれば、アルミナ中の空気が抜けます。いちど空気が抜ければ、再び巻き込むことはないので、高温型エポキシ樹脂接着剤で蓋をしておきます。

二つの温度センサにケーブルと PS2 マウス用コネクタを取り付けた様子です。ジュンフロン線と耐熱電線のつなぎ目は熱収縮チューブで保護しています。

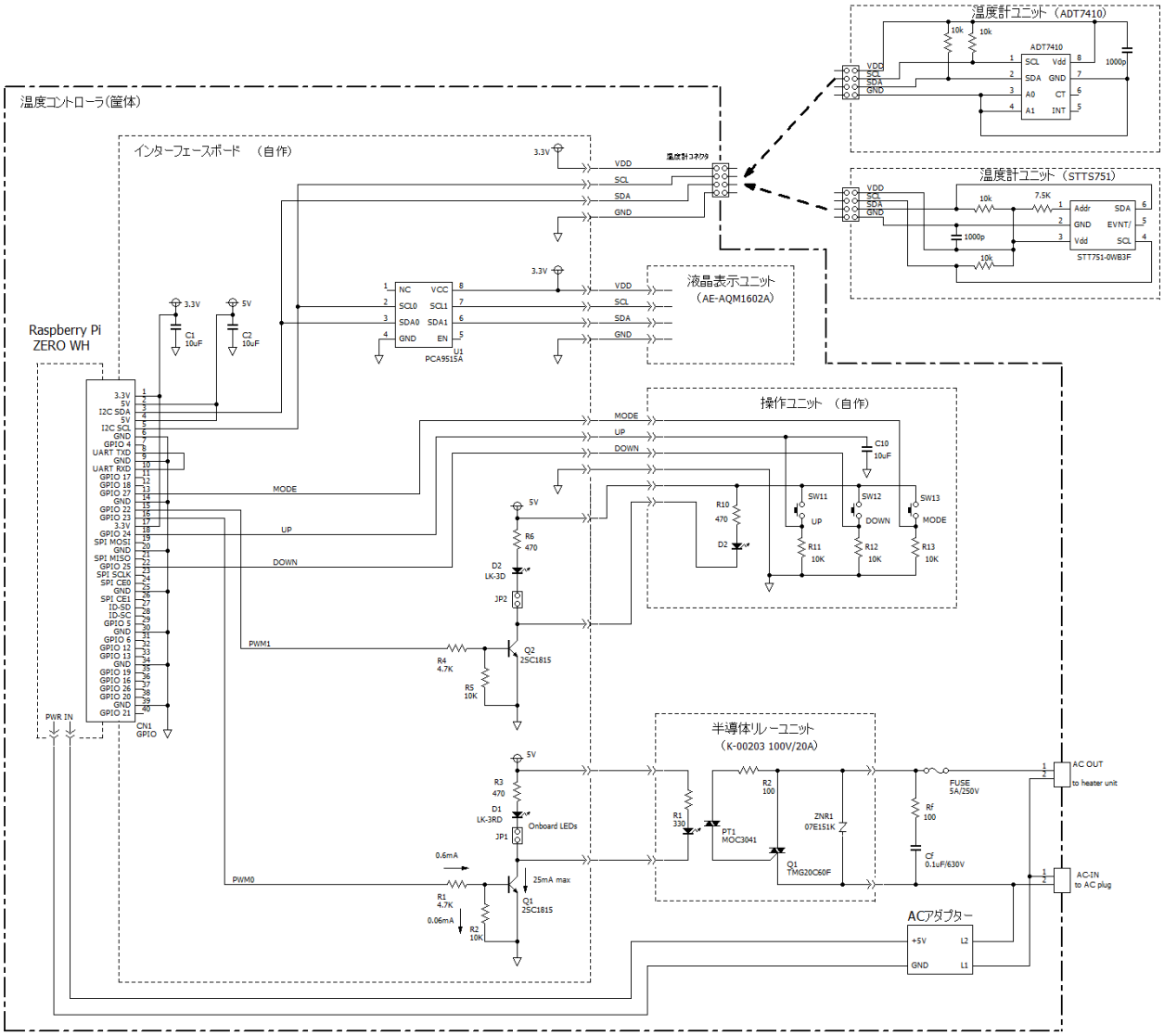
二種類の温度センサ（左：湯温用、右：空気用）

工業用の温度センサは、ステンレス製のケースに入っています。片方が塞がったステンレス管が手に入るなら、ガラスと違って割れる心配のない温度センサが作れます。ただし、管壁に触れてショートするのを防ぐため、熱収縮チューブなどで覆ってから、封入してください。

2.8　全回路図

ここまでで、全部のユニットができ上がりました。それらの接続を含めた全回路図を下にまとめます。ケースの中で AC 電源を引き回す際には、絶縁が保たれるように注意し、事故を防ぐため AC 電源接続部が露出しないようにしてください。ケースの外からは触れられないようにすることが特に重要です。

なお、タイマー割り込みを利用するため、TXD と RXD をジャンパーピンで接続しています。



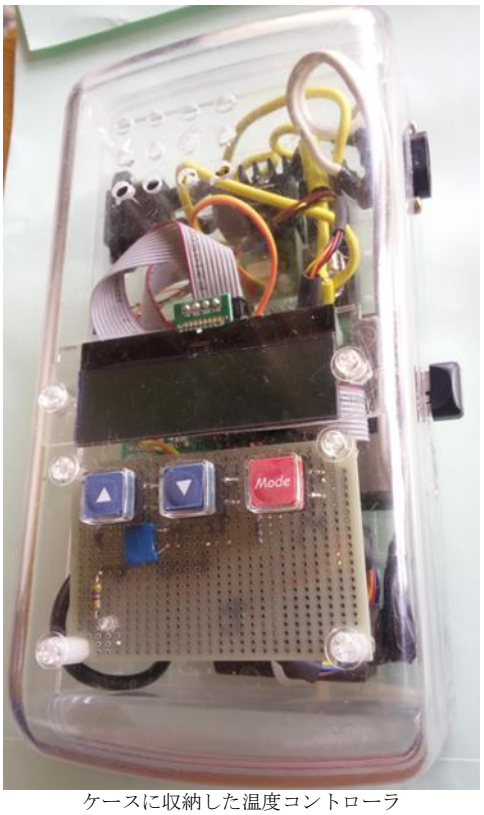
温度コントローラ全回路図

2.9　ケースの組み立て

全回路をケースに入れるための配置を考えます。ユニット同士が接触しないようにし、配線のためのスペースを確保します。半導体リレーユニットの発熱を考慮し、ケースには取り付ける場所に換気用の穴を開けておきます。コネクタとユニット固定用の穴も開けます。キースイッチを取り付ける場所では、キートップがケースと接触しない大きさの穴にします。温度センサ以外をケースに組み込みます。

ケースに各ユニットと外部コネクタを取り付け、その間を接続します。あとあとの保守を考えて、AC 電源部を除いて、分解できるようにしておきます。コネクタ付きフラットケーブルが便利ですが、接続方向を間違えないようにしましょう。ユニットの固定にはネジ（できればプラスチックネジ）を使っておきます。AC 配線をする際には、前にも説明したように、しっかりと固定し、はんだ付け部が露出しているところは、ビニルテープを巻いてから自己融着テープで覆うか、熱収縮チューブで被覆します。

出来上がりの写真を右に示します。ケースの上半分には、半導体リレーユニットとスナバ回路、それに電源入力とリレー出力用のコンセントが収納されています。操作ユニットの下には AC アダプターがあります。発熱が予想された半導体ユニットの近くには、通気用の穴を開けてあります。外部には AC が露出していないことに注意してください。これは最初のころから使っているケースです。当時の CPU は Raspberry Pi B だったので、有線 LAN や WiFi アダプターのコネクタにケース外からアクセスできるようになっています。



ケースに収納した温度コントローラ

他の有名な OS

Apple の MacOS は、BSD 版 UNIX の技術をもとに開発されました。特にウィンドウ（Windows ではない）を使ったグラフィックインターフェースが多くのユーザに受け入れられました。スマートフォン用の iOS も同じ技術に基づいています。

Google の Android は、スマートフォンやタブレット端末用の OS として、最も普及しています。Linux のカーネルを使い、その上に独自のライブラリやアプレットを載せています。同社の PC 向け OS である Chrome（専用 PC だけで使える）と Chromium（オープンソース）も、Linux カーネルの上に構築されています。

Microsoft の MS-DOS/Windows は、IBM 互換 PC のために開発され、UNIX とは別系統の技術（CP/M）に基づく OS として進化してきました。ただ、Microsoft は MS-DOS の前に UNIX の流れをくむ OS を製品化していたので、MS-DOS の使い心地は UNIX と似た感じがします。コマンド名こそ違っていますが、パイプなどの考え方は UNIX から受け継いでいます。さらに Mac OS（1984）や UNIX の X-Window（1984）の後からリリースされた Windows（1985）も先行 OS に似た印象です。

3　ソフトウエア開発環境

3.1　ソフトウエアの部品化

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。

温度センサや液晶表示、パルス幅変調などは、さまざまな本で取り上げる、かなり汎用的な機能です。その部分のソフトウエアを部品（モジュール）化しておけば、温度コントローラ以外でも使えるようになります。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手法を紹介します。

3.2　Linux ツールの利用

Raspberry Pi で使うオペレーティングシステム（OS あるいは基本ソフトウエアとも呼ばれます）は UNIX を起源とする Linux です。UNIX は、もともとソフトウエア開発をする人たちが便利に使える OS として開発されました。せっかくなので、その機能を開発環境として上手に使いましょう。

PC の OS に Ubuntu を使うと便利なのは、Raspberry Pi とほとんど同じ環境を使えるからです。ソフトウエアの作成を PC で行い、かなりの部分は、そこで検証（デバグ）できます。検証結果を表計算ソフトで処理するのも簡単です。この章では、Raspbian と Ubuntu の両方で使えるツールを考えます。

3.2.1cpp

cpp（C 言語プリプロセッサー）は Linux の前身である UNIX の C 言語用に開発されたもので、見やすい形のソースコードを、コンパイラが処理しやすい形に変換します。その主な機能は、(1) 他ファイルの取り込み、(2) 文字列の置き換え、(3) 条件付き処理、それに(4) コメントの除去です。その指示はすべて#で始まる命令（#は必ず行の先頭から始まり、インデントはできません）からなり、その他の部分は元のままで出力します。出力先は Linux の標準出力で、ファイルにリダイレクトしたり、パイプ（次節参照）につないで次のプログラムに渡したりできます。今回のプロジェクトでは次のような命令を使います。

#include（他のファイルを取り込む）

#include 命令は、その位置に他のファイルを取り込みます。取り込むファイル名は、命令のすぐ後（同じ行）に指定します。取り込むファイルが、いま開いているファイルと異なるディレクトリにある場合は、それへの相対パスで指定します。

```
#include "include/pid.h"
#include "pid.py"
```

この機能のおかげで、オブジェクト毎にそれぞれのファイルを作り、部品（モジュール）化することができます。

Python を使った経験のある人なら、ここで「ちょっと待って」と言うかもしれません。「それって、import と同じではないか？」というものです。例えば、thermo.py のなかに、こんな記述があります。

```
#include "i2c.py"
  :
    self.i2c = i2c_device(...)
```

これは、import を使って以下のようにも書けるはずです。わざわざ cpp を持ち出す必要はないように思えます。

```
import i2c
  :
    self.i2c = i2c.i2c_device(...)
```

上の考察は正しいのですが、ひとつ重要な制限があります。それは、import されるモジュールが、Python で実行できる形になっている必要があることです。以下に説明するような、#define を使って名前を定義する、#ifdef を使ってモジュールの機能そのものを切り替えるといった強力な操作は、import されるモジュールでは使えなくなってしまいます。上の例では i2c.py をシミュレーション環境で評価したり、実機上で動かしたりするために cpp を使っているのです。

最終的にモジュールの機能仕様が固定でき、検証が済んで、安定したモジュールとして使えるのであれば、import を使うことを考えても良いでしょう。

#define（名前を定義する）

#define 命令は、次に続く名前（マクロ名）をその後ろにある値（置換文字列）との関連付けを定義します。処理しようとするファイルの中に、その前に

定義された名前が出てくると、定義された値に置き換えます。名前も値も文字列として記述します。例として次のようなファイル cpp-test.txt を作ってみましょう。以下、この例のようにファイルの内容を示すときは、ファイル名を黄色欄に示すようにします。

cpp-test.txt

```
#define MY_STRING  hello!
#define MY_NUMBER  123

MY_STRING = MY_NUMBER
```

これを cpp に与えると、以下のように置き換えられます。

```
$ cpp cpp-test.txt
hello! = 123
```

名前には英数字とアンダースコアが使えます。プログラムコードの中では、「これはマクロ定義されたものだ」と分かるように、マクロ名をすべて大文字にすることがよく行われます。この本のなかでも、同じ習慣に従っておきます。

一度定義したマクロ名を、もう一度定義しようとすると、cpp は警告を表示します。

こんな定義をする理由として、以下の 4 つがあげられます。

1. 意味不明の数値をなくす
2. 保守性を良くする
3. モジュールの機能を切り替える
4. 関数も定義できる

1. 意味不明の数値をなくす

プログラムの中に数値を埋め込んでいると、それが何を表すのか分かりません。例えば以下のようなコードでは、

```
if mode == 3:
   :
```

この 3 が何かを数えた結果なのか、約束ごととして取り決めたものなのかよく分かりません。そのプログラムを書いた人には分かっているのでしょうが、他の人が読んでも意味が伝わりません。しばらく後になると、プログラムを書いた本人でさえ思い出せなくなります。こういう意味不明の数値を、「なぜか分からない数値だが、それを使うと動作する」という意味で、マジックナンバーと呼んだりします。マジックナンバーをなくすのが、第一の理由です。

2. 保守性を良くする

プログラムは一度書いたら終わりというわけではなく、使い込んでいくうちに手直しが必要になることがあります。この手続きを保守といいます。プログラムに問題が見つかっても、マジックナンバーがあると、どこに手を入れたらいいのか分かりません。後でプログラムを読み直すために、コメントを付けたりするわけですが、デバグにとり紛れたりして付け忘れることが、しばしばあります。

また、同じ意味で使っている数値が複数の個所に現れることもよくあります。特にモジュール間での情報伝達や、通信を行うときには、プログラム内の離れた場所で使われます。その値を変更する必要ができたとき、一方を変更し忘れたために苦労することになります。その点、定義ファイルに#define しておけば、それを取り込むすべてのプログラムが一度に修正できます。

こういった定義を使うプログラムは、一つのファイルとは限りません。そのため、独立した定義ファイルを作り、各プログラムの冒頭で#include するようにします。冒頭に組み込むことから、こういう定義ファイルは「ヘッダーファイル」と呼ばれます。共通して使えるよう、決まったディレクトリに置くことが多く、ファイル名の拡張子を.h とする習慣があります。

ヘッダーファイルのなかに、特定のプログラム言語のコードを入れるのは、望ましくないと言われています。ただ今回は、プログラムの流れをつかむのに関係ないコードは、ヘッダーファイルに含めることがあります。その場合のヘッダーファイルは、他の言語では使えません。

2. モジュールの機能を切り替える

プログラムモジュールは、できるだけ汎用に作りたいものです。用途が異なってもソースコードに手をつける必要がなければ、さまざまな用途に使えます。例えば温度センサのヘッダーファイルでは、以下のように定義して、ATD7410A（高分解能と低分解能）と STTS751 の設定を切り替えることで、同じモジュールを別々の温度センサで使えるようにしています（#ifdef については後で説明します）。

```
#ifndef STTS751
     /* 温度センサ ADT7410A の定義 */
#ifdef TEMP_HIGH_RESOLUTION
#define TEMP_CONFIG_DATA  0x80
    /* 16 ビット分解能 */
#else
#define TEMP_CONFIG_DATA  0x00
   /* 13 ビット分解能 */
#endif

#else
```

```
    /* 温度センサ STTS751 の定義 */
#define TEMP_CONFIG_DATA   0x4c
    /* 12 ビット分解能、ワンショット動作 */
#endif
```

この用途で、今回いちばん多く使うのが BCM2835 というマクロ名で、これが定義されていると Raspberry Pi のハードウエアを直接使うコードが出てきます。定義されていないと出力は print 文に、入力は input 文に切り替えられ、入出力をシミュレートします。このおかげで、プログラム機能の大部分を PC の上で検証できるようになります。

3. 関数を定義する

マクロ名にカッコと仮変数をつけると、関数（マクロ関数）が定義できます。cpp はマクロ関数を定義した式に置き換えてくれます。例えば、以下のようなファイル my_func.py

my_func.py

```
#define MY_SQUARE(x)  x*x
#define IS_LARGER(x, y)  x>y

if IS_LARGER(this, that):
  print MY_SQUARE(this)
```

を cpp で処理すると、

```
$ cpp my_func.py
if this>that:
   print this*this
```

と変換してくれます。これは実際に関数を作っているのではなく、見やすい名前を、実際の数式に置き換えるだけです。数式が複雑になればなるほど、また何回も使えば使うほど、この定義の効果が表れます。この定義を「マクロ定義」、置き換えを「マクロ展開」と呼びます。温度コントローラのソフトウエアでは、表示イメージを生成するところで、たくさんのマクロ展開が行われています。

#ifdef ～ (#else ～) #endif（条件付き処理）

この定義機能を使って、プログラムの一部を選択することができます。#ifdef（if defined）は、それに続くマクロ名が定義されているときだけ次のテキストを残し、定義されていない場合はそっくり削除します。

```
#ifdef OPTION1
  この文字列は、
  OPTION1 が定義されていると残る
  定義されていないと削除される
#else
  この文字列は、
  OPTION1 が定義されていると削除される
  定義されていないと残る
#endif
```

#else 以下はなくても構いませんが、最後は必ず #endif で終わります。定義されていない場合だけ使いたいときには、#ifdef の代わりに#ifndef（if not defined）を使うことができます。また、#ifdef のブロックの中で別の#ifdef を記述することもできます。

ちょっと変わった使いかたですが、一つの定義ファイルを複数のモジュールで#include する場合があります。それぞれのモジュールを評価するときには問題ないのですが、全体をまとめて動かすときには、重複定義を避ける必要があります。そこでインクルードファイルは、以下のような記述をしておきます。

```
#ifndef __THIS_FILE
  定義の記述
  :
#define __THIS_FILE
#endif
```

一回目に取り込んだ時には、定義が読み込まれます。二度目からは__THIS_FILE が定義されているので、空文しか残りません。__THIS_FILE の定義名は、インクルードファイルごとに変えます。先頭にアンダースコアを 2 回続けるのは、そういうマクロ名が他で使われることはないだろうという推測からです。

/ コメント */*

cpp は、”/* ”と” */”で囲まれたテキストを、コメントと認識して、すべて削除します。コメントが複数行にわたっていても、また”/* “が行の途中から始まっても構いません。

-D オプション

cpp には、-D というオプションがあります。このオプションを使えば、元のファイルに手を加えることなく、処理の最初に#define することができます。例えば、

```
$ cpp -DSTTS751 -DT_AMBIENT=30 thermo.py
```

とすると、thermo.py を開く前に、次のような記述があったとみなします。

```
#define STTS751
#define T_AMBIENT  30
```

この機能を使って、#ifdef で条件ごとの処理を記述しておけば、cpp 実行時に使うコードを指定してやることができます。モジュールの機能を切り替える

のに、この方法を使えば、ソースコードに手を加える必要がありません。

重複定義を避けるため、-D オプションで定義を変更する可能性のあるマクロ定義は、#ifndef と#endif で囲っておきます。

3.2.2リダイレクトとパイプ

Linux のプログラムで入出力先を指定しないときは、「標準入力」と「標準出力」が使われます。ふつうの状況では標準入力はキーボード、標準出力はコンソールのスクリーンになっています。プログラムを実行するときに、他の入出力先に切り替えることができます。ファイルから入力させるときは、<(ファイル名)とし、出力先をファイルにするには、>(ファイル名)と指定します。

```
$ echo hello!
hello!
$ echo hello! >tmp
$ echo <tmp
hello!
```

二番目のコマンドは echo の出力先を tmp というファイルにする、三番目の命令は入力先を tmp にするという命令です。三番目の命令は cat tmp と同じことです。

あるコマンドの出力を、別のコマンドの入力にする機能を「パイプ」といって、“|” で指定します。以下のような例をよく見かけますね。これは全てのプロセスのうち、tty という文字列を含むものをすべて表示します。

```
$ ps -aux | grep tty
```

リダイレクトとパイプは、比較的簡単なコマンド（プログラム）を組み合わせて、複雑な処理を行うのに有効です。

3.2.3クリーンアップ

すこし前に戻りますが、cpp が（標準出力へ）出力するテキストは、あまり読みやすいものではありません。空行が残っているし、処理した内容を#で始まるコメントとして残しています。そのあとをコンパイラなどで処理する分には問題ないし、#で始まる行は python でもコメントとして無視されます。

それでも人が読みやすいようにするため、きれいにするツールを用意しました。これはマクロ展開などが正しく行われているか確認するときなどに役立ちます。次のような python プログラム cleanfile.py を作っておきます。

cleanfile.py

```
# cleanup procedure for cpp output

while True:
  try:
    s = raw_input()
    if s != "":
      if s[0] != "#":
        print s
  except EOFError:
    break
```

空文と冒頭が#で始まる行を削除するだけですが、見やすさは格段に良くなります。実際に使うときは、

```
$ cpp program.py | python cleanfile.py > object.py
$ python object.py
```

とするか、あるいは

```
$ cpp program.py | python cleanfile.py | python
```

とすれば、複数のモジュールからなるプログラムを組み合わせて（#include して）実行することができます。

検証途中でキーボードからの入力を必要とするときがあります。このときは実行時のコードをパイプから受け取っていると、標準入力がそちらを向いているので、入力がないというエラーが起きてしまいます。一度実行コードをファイルに入れる、前の方の手順を取ってください。

3.2.4シェルスクリプト

Linux のコマンドひとつひとつは、比較的簡単な機能を実現する場合が多いのですが、コマンドオプションとパイプを使うと、こみいった処理も行えます。ただ、そのたびに長いコマンド列をタイプするのは効率が悪いし、ミスタイプも起こしやすくなります。そこで命令の列をファイルに記述して、このファイルを実行するのがシェルスクリプトです。例えば runpython というファイルに、以下のように記述します。

runpython

```
cpp $1 | python cleanfile.py | python
```

ここで$1 は、一つ目のパラメータを表します。まず、次のコマンドで、このファイルを誰にでも（a）実行できる（+x）ようにします。

```
$ chmod a+x runpython
```

そこで、以下のようなコマンドを与えて、program.py を実行することができます。

```
$ runpython program.py
```

cpp の-D オプションを変えたシェルスクリプトを作っておくと、いろいろな温度コントローラを実現できます。私は、チューニングを済ませたパラメータを-D オプションで与えるシェルスクリプトを、スモーカー用や低温調理器用などに用意しています。

もっと複雑なシェルスクリプトを書くこともできますが、それは専門書を参考にしてください。

3.3　プログラムの検証

作成したプログラムには、間違い（バグ）が含まれていることがあるので、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。

3.3.1シミュレーション用コード

温度コントローラのように、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。

3.3.2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 x, y, …
#endif
```

シミュレーション環境やデバグで犯しやすいミスを指摘しておきます。先に定義した runpython というシェルスクリプトでは、スクリーンへの出力は問題ないのですが、キーボードから入力を与えることができません。整形された Python プログラムの標準入力がパイプになっているからです。この場合には、いったん cpp の出力をファイルに入れて、それを実行させます。前のクリーンアップの節を見てください。

3.3.3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、下位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。検証の主眼ではないが、それがないと上位モジュールの動作が確認できないときに使います。温度コントローラの場合は、制御対象がなくても評価できるように、温度が一定法則で変化する数値モデルをスタブとして用意しました。

また、割り込み処理を検証するため、「いま、こういう割り込みを受けた」と表示するだけのスタブも用意しました。

ヘッダーファイル

この本で採用している、ヘッダーファイルの二重読み込みを避けるために、本体を#ifndef __〇〇 と #define __〇〇 #endif で囲むという手法が、どれだけ一般的に普及しているかは知りません。共同開発したドイツの会社の技術者たちが使っているのを見て、なるほどドイツ人は実践的だわいと思い、自分たちも使うようになりました。モジュール毎の検証と、総合検証が矛盾なく（定義がないとか、二重にあるとかいうメッセージに悩まされず）できたのは、このおかげです。彼らには感謝してもしきれません。

4　ソフトウエアモジュールの開発

ハードウエアの動作テストにも使えるよう、ソフトウエアモジュールは先に用意しておきます。

この章では、温度コントローラのソフトウエアをモジュール化し、実際に作成していきます。各々のモジュールの詳細な仕様を説明してから、プログラムコードを示しています。

最終的に温度コントローラとして動作させる Python のプログラムは、約 570 行にもなります。このくらいの大きさになると、全体のプログラムリストを眺めて考えることは、ほとんど不可能です。目の行き届く大きさのモジュールに分解し、それぞれの間の相互作用を設計するような、階層的アプローチが必要になります。

この章でヘッダーファイルとプログラムファイルを示すとき、ファイル名を黄色欄に示すようにします。本の印刷幅に入れる都合上、一部の行は右端で折り返して、複数行に印刷しています。また、印刷する行を短くするため、元ファイルのタブを 2 文字の空白で置き換えています。少し見にくい場合もありますが、そんなときは配布ファイルを参照してください。

4.1　全体設計

「温度コントローラ」というひとまとまりの「装置」は、多くの部品（モジュール）から組み上げるようにして設計していきます。その方法として、次のようなアプローチ（やり方）があります。

- 全体像から詳細化していくトップダウンアプローチ
- 部品を決めてから組み上げていくボトムアップアプローチ

ボトムアップアプローチは、個々の部品が決まっているから理解しやすいと言われることがあります。ハードウエアの設計が済んでいれば、それを組み上げていけば良いと考えがちだからです。しかし、一人でこれをやろうとすると、ほんとうは必要なソフトウエア部品を、しばしば入れ忘れてしまいます。また、部品の仕様を先に決めると、その影響で全体構成が歪んで、検証や保守に悪い影響を及ぼすこともあります。

最初に決まっているのは、温度コントローラの外部仕様ですから、これを部品に分解していくトップダウンアプローチを基本に、モジュール構成を設計することにします。

ただし、設計時はともかく、各モジュールの説明をトップダウンアプローチで行うと、非常に分かりにくくなってしまいます。そこで次節からのモジュール毎の説明は、ボトムアップの順番で行います。検証（動作テスト）も同様に、末端から順番に動作させていきます。

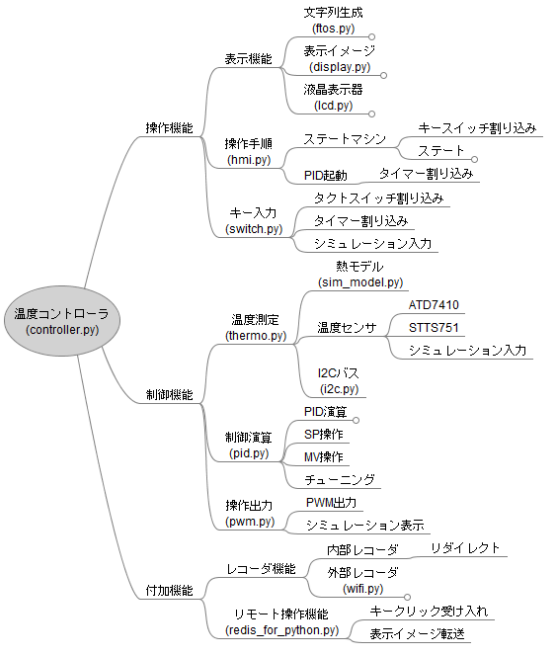
4.1.1モジュール構成

この節ではトップダウンアプローチをとります。仕様書をよく読むと、温度コントローラの主要な機能は、次の 3 つに分解できることが分かります。

- 温度を制御する機能
- 人の操作を支援する機能
- オプション（付加）機能

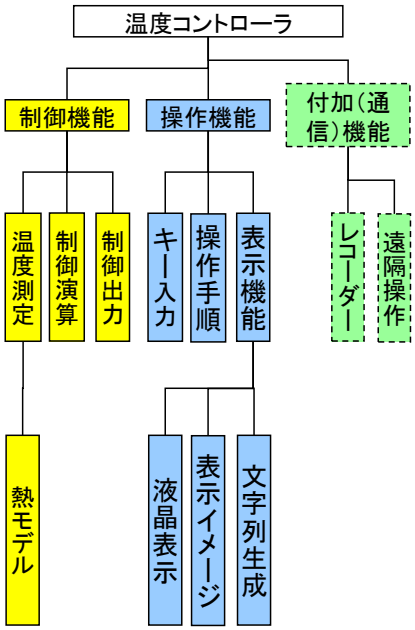
この 3 つの機能をつぎつぎに詳細化していくのがトップダウンアプローチです。ところが実際に進めていくと、詳細化した下位機能を眺めるうちに、上位機能の設計不備に気が付くことがあります。その時には、上位機能の設計に戻ってやり直します。

この過程で役立つのが、上位から下位にいたるツリー構造を描く、マインドマップという手法です。要素に下位の要素を取り付けたり、まとめて上位要素に繋ぐ（あるいは繋ぎ変える）ことで、トップダウンとボトムアップを行ったり来たりできます。作成途中のマインドマッを下に示します。末端の〇印は、下位要素を表示していないことを表します。



マインドマップによるトップダウン設計

できあがったトップダウン構成を下図に示します。



機能のトップダウン構成

制御機能

温度測定機能は温度センサの読み取りと、データの形式を Python で扱える数値に変換する機能です。

制御演算機能は一つのまとまったものだと考えても良さそうです。入出力は含まずに、PID 演算を行うブロックとして設計します。

制御出力機能は PID 演算結果を PWM 信号として出力します。

操作機能

操作機能は、人がキーを押したときにどう表示し、温度制御機能をどう変えていくかというものです。

キー入力機能はキースイッチの操作を読み取り、長押しなどの処理を行います。

操作手順機能はかなり複雑です。あるキーを押すと状態（ステート）が変化し、その後の反応に影響するようになります。こういう動作はステートマシンと呼ばれるものなので、その設計手法を取り込みます。

表示機能は、操作に必要な情報を液晶表示器に表示します。表示イメージ機能では、数値を文字列に変換する文字列生成機能を使いながら、表示内容を組み立てます。液晶表示機能では、その内容をハードウエアに送ります。

付加機能は、制御状態を記録・表示するためのレコーダー機能と、ブラウザによる操作を可能にする遠隔操作機能からなります。

抽象的なレベルの機能

制御演算と操作手順、表示イメージと文字列生成は、ハードウエアには依存しません。まとめて抽象レベルと呼ぶことにします。付加機能も同じレベルだと考えます。

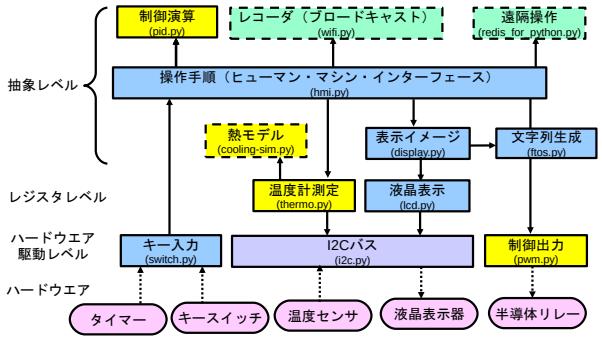
ハードウエア駆動機能

トップダウン構成には出てきませんでしたが、ソフトウエアが使用（駆動）するハードウエアは、キースイッチ、液晶表示器、LED、半導体リレー（SSR)、温度センサ、内部タイマーです。このうち液晶表示器と温度センサは I2C バスを介して駆動するので、バスの動作と、各 IC の内部レジスタ操作をレベルが異なるモジュールにしておきます。前者はハードウエア駆動レベル、後者をレジスタレベル（またはドライバー）と呼ぶことにします。I2C バス駆動機能は、二つのモジュールで使うので、独立したモジュールにします。

このようにソフトウエアモジュールを階層化するのは、個々のモジュールをオブジェクトとして定義するのに役立ちます。各モジュールは、そのオブジェクトに関する詳しい知識と、下位モジュール（オブジェクト）の操作仕様だけを知っていれば、設計も評価もできるからです。その時に必要な情報だけを気にすればいいので、設計が簡単になります。また、何人かで設計や評価を分業することも容易です。

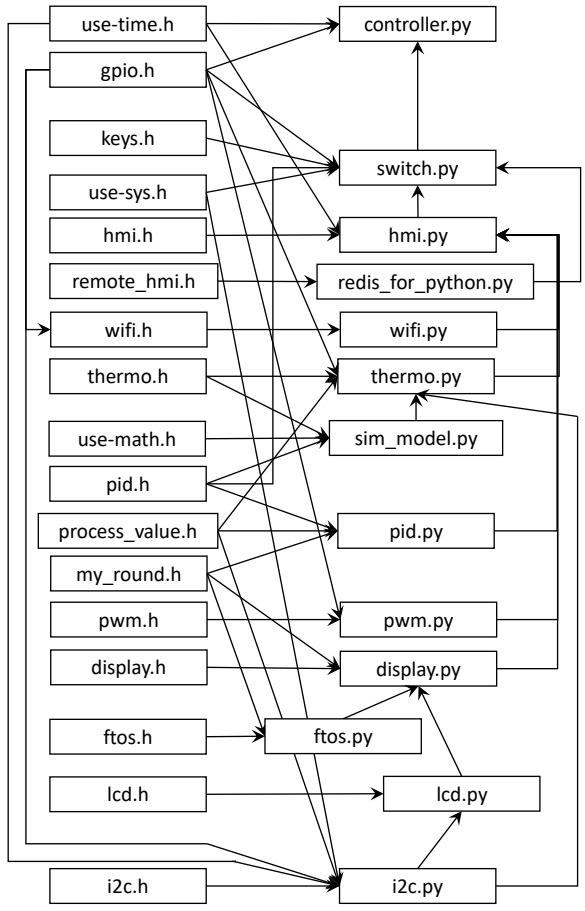
例えば、操作手順で表示を考えるときは、何という欄に何を表示するかという定義でことが足ります。下位の表示イメージは、液晶表示器のどのフィールドにどんな文字を表示するかを決めますが、液晶表示器のレジスタ構造は知らずに済みます。液晶表示（レジスタレベル）は、表示器制御用 LSI のどのレジスタに何を書き込めばいいかを知っているだけで、実際の書き込みが I2C バスを経由しているのか、8 ビット並列バスを経由しているのか知りません（実際に、レジスタ構造は同じで、他のバスインターフェースを使うデバイスが存在します)。

抽象／レジスタ／ハードウエアレベルに分類したモジュール構成を次ページの図に示します。矢印はオブジェクトを生成する向き、点線矢印はハードウエアとの信号の流れです。



モジュールのレベル毎構造

ここまで設計ができあがると、各モジュールを記述するファイルを決めることができます。下図の右側はオブジェクトを記述する Python プログラムファイル、左側は定義を記述するヘッダーファイルです。矢印はそれぞれを#include する向きをあらわしています。プログラムファイルは、必要とするインクルードファイルをすべて#include することで、単独でも動作させて検証ができるようにしています。#ifndef を上手に使って、何回も#include しても警告が出ないようになっています。



ソフトウエア全体のファイル構成

出来上がったモジュールは、ハードウエアに近いものから順番に動かすか、スタブを使って検証を進めていきます。

4.1.2オブジェクト指向

ソフトウエアモジュールは 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。

- クラス：オブジェクトを生成するための定義
- 属性（アトリビュート）：オブジェクトの内部変数
- 操作（サービス）：オブジェクトに対して操作をするための関数

各々のモジュールのオブジェクトとしての定義は、次節以降の各モジュールの最初に示しています。この資料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、直接参照することがあります。

4.1.3機能選択用#define

各モジュールから機能選択を行うために、以下の定義を使います。液晶表示器の要素数定義は、液晶表示器ドライバーの節を参照してください。

パラメータ	用途	#ifdef	#ifndef
BCM2835	実行環境の指定	Raspberry Pi 実機で実行	シミュレーション
STTS751	温度センサの指定	STTS751	ADT7410A
HIGH_RESOLUTION	ADT7410A の分解能	16 ビット	13 ビット
TEMP_SIMULATION	温度シミュレーション	熱モデル	センサまたはキーボード入力
LOCAL_RECORDER	PID 制御記録を指定	ローカルに出力	出力しない
NETWORK_RECORDER	PID データ送信を指定	ネットワークに送信	送信しない
HW_DEBUG	ハードウエアの評価	ハードウエアの動作をモニター	通常処理
LCD_SILENT	検証中の LCD	表示しない	表示する
REMOTE_OP	リモートオペコン機能の指定	リモートオペコンを実現	リモートオペコンを使用しない

機能選択用マクロ定義

上のマクロ定義以外に、先に説明したインクルードファイルの多重読み込み防止用の定義があります。

```
#ifndef 既読マーカー
    :
#define 既読マーカー
#endif
```

4.2　ヘッダーファイル

各ソフトウエアモジュールに固有なものを除き、共通して使えるヘッダーファイルを、ここにまとめておきます。

ヘッダーファイルのなかには、定義だけでなく、プログラム（cpp 処理で削除されない Python のコード）の一部を含むものがあります。インクルードファイルに定義以外の記述をするのを嫌う人もいますが、私はむしろ積極的に利用しています。プログラムモジュールに import 命令などを並べるより、「他から借りてくる」ことを明示したかったからです。

4.2.1GPIO

Raspberry Pi の GPIO を使うための情報を gpio.h で定義します。

gpio.h

```
/* BCM2835 GPIO 定義
  初版:  2014/12/28 Chuji
  最新版:  2017/2/26 複数チャンネルの PWM をサポート
*/

#ifndef __GPIO

/* ターゲットシステムでのみ有効な処理 */

#ifdef BCM2835
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)  /* ピン番号ではなくポート名で指定する */
#endif


/* 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
  GPIO3 (pin05):  I2C SCL
  GPIO4 (pin07):  CPCLK --- 使用しない
  GPIO5 (pin29):  使用しない
  GPIO6 (pin31):  使用しない
  GPIO7 (pin26):  SPI CE1 --- 使用しない
  GPIO8 (pin24):  SPI CE0 --- 使用しない
  GPIO9 (pin21):  SPI_MISO --- 使用しない
  GPIO10(pin19):  SPI MOSI --- 使用しない
  GPIO11(pin23):  SPI SCLK --- 使用しない
  GPIO12(pin32):  使用しない
  GPIO13(pin33):  使用しない
  GPIO14(pin08):  UART TXD *** 周期割り込み
  GPIO15(pin10):  UART RXD *** 割り込み検出
  GPIO16(pin36):  使用しない
  GPIO17(pin11):  使用しない
  GPIO18(pin12):  使用しない
  GPIO19(pin35):  使用しない
  GPIO20(pin38):  使用しない
  GPIO21(pin40):  使用しない
  GPIO22(pin15):  PWM 1
  GPIO23(pin16):  PWM 0
  GPIO24(pin18):  DI 0 *** UP キー
  GPIO25(pin22):  DI 1 *** DOWN キー
  GPIO26(pin37):  使用しない
  GPIO27(pin13):  DI 2 *** MODE キー
*/


/* GPIO ポート名の指定 */

#define GPIO_MODE_SW  27 /* GPIO ポート番号 */
#define GPIO_UP_SW  24
#define GPIO_DOWN_SW  25

#define GPIO_PWM0  23
#define GPIO_PWM        GPIO_PWM0
#define GPIO_PWM1  22

#define GPIO_CLOCK_OUT   14 /* TXD から周期割り込みを発生 */
#define GPIO_CLOCK_IN    15 /* RXD で周期割り込みを検出 */


/* I/O レベルの定義 */

#ifdef BCM2835 /* 実機の場合 */

#define GPIO_OUT GPIO.OUT
#define GPIO_IN  GPIO.IN
#define GPIO_LOW GPIO.LOW
#define GPIO_HIGH GPIO.HIGH

#define GPIO_setup(x,y)   GPIO.setup(x,y)
#define GPIO_output(x,y)  GPIO.output(x,y)

#else           /* シミュレーションの場合 */

#define GPIO_OUT 1
#define GPIO_IN  0
#define GPIO_LOW 0
#define GPIO_HIGH 1

#define GPIO_setup(x,y)   print "GPIO ポート指定:", x, "<-", y
#define GPIO_output(x,y)  print "GPIO ポート出力:", x, "<-", y

#endif

/* 表示灯のレベル定義 */
#define INDICATOR_ON GPIO_HIGH
#define INDICATOR_OFF GPIO_LOW


#define __GPIO

#endif
```

4.2.2システムコール

システムコール用の定義をいくつか用意します。まず Linux コマンドを実行するための use-sys.h です。宣言とコマンドが定義されています。

use-sys.h

```
/* システムコールの使用宣言
     初版： 2017/3/7 Chuji
     最新版： 2019/1/22 -- シャットダウン追加
*/

#ifndef __USE_SYS

import sys
import os

#define SYS_SHUTDOWN  "shutdown -h now"
#define SYS_REBOOT  "shutdown -r now"

#define __USE_SYS
```

```
#endif
```

次は時間関数 sleep を呼ぶための定義ファイル use-time.h です。

use-time.h

```
/* タイマー使用宣言
  初版：  2016/11/2 Chuji
  最新版：  2017/2/28
*/

#ifndef __USE_TIME

/* スリープタイマーのインポート */

from time import sleep
from time import time

#define __USE_TIME

#endif
```

4.2.3パッケージライブラリ

数値演算パッケージを利用するための定義ファイル use-math.h です。

use-math.h

```
/* math ライブラリの使用宣言
     初版： 2016/11/3 Chuji
     最新版： 2016/11/3
*/

#ifndef __USE_MATH

import math

#define __USE_MATH

#endif
```

4.2.4プロセス値の定義

温度センサの読み取りエラーや PID 制御の異常値を処理するため、これらを受け渡すときは、数値とステータス（GOOD または BAD）をペアにして行います。この構造体をクラスとして定義しておきます。

process_value.h

```
/* ステータス付き制御変数
  国際標準 IEC61499 とフィールドバス協会による

  初版： 2017/2/19 Chuji
  最新版： 2017/2/21
*/

#ifndef __PV_STRUCT

#define INIT_VALUE  0
#define  STATUS_GOOD  0
#define STATUS_BAD  1

class process_value:
  def __init__(self):
    self.value = INIT_VALUE
    self.status = STATUS_BAD
```

```
#define __PV_STRUCT

#endif
```

二重定義はほんとうにダメ？

前に、ヘッダーファイルの二重定義は避けるべきだと言いました。実は cpp は、全く同じ定義が出てくる限り、黙って処理してくれます。例えば、

```
#define ABC  123
#define ABC  123
```

は異常と認められません。ここで片方が 1234 にでもなっていると、警告を出したうえで、後の定義を採用します。二つ目の定義が出てくるまでは、先の定義を使っているので、プログラムのなかで矛盾が起こってしまいます。

いっぽう、以下の三つの記述は、同じ意味なのに、重複定義とみなされます。cpp は文字列を登録して、置換するだけなので、こういうことが起こるのです。

```
#define FUNC1(x)  (x*x)
#define FUNC1 (x)  (x * x)
#define FUNC1(y)  (y*y)
```

4.3　制御出力（パルス幅変調）

半導体リレー駆動とアラーム表示は、どちらも PWM 出力ポートを使用するので、同じファイル pwm.py に記述しました。実際には二つのオブジェクトを定義しています。コードはまとめて右欄に示します。

4.3.1半導体リレー駆動

オブジェクト定義

クラス：pulse_width_modulator

属性：pwm（PWM ハードウエア）

操作：output（デューテイ変更）

ヘッダーファイル

pwm.h でパルス幅変調のポートと設定を定義します。パルス周波数 PWM_FREQUENCY0 を 1 より少しだけ大きくしているのは、商用電源の周波数との同期を避けるためです。

なお、このファイルでは PWM 出力を 2 チャンネルまで持てるように定義していますが、実機では、2 チャンネル目をアラームとして使っています。

pwm.h

```
/* パルス幅変調の定義
   初版：2014/12/31 Chuji
   最新版：2016/12/03 複数ポートのパルス幅変調追加
*/

#ifndef __PWM

#include "gpio.h"

/* PWM 出力ポート */

#define PWM_PORT0  GPIO_PWM
#define PWM_PORT   PWM_PORT0
#define PWM_PORT1  GPIO_PWM1
#define STATUS_PORT PWM_PORT1

/* PWM パルス周波数（単位はヘルツ） */

#define PWM_FREQUENCY0  1/0.99
#define PWM_FREQUENCY  PWM_FREQUENCY0
#define PWM_FREQUENCY1  PWM_FREQUENCY0

#define PWM_MIN    0.0
#define PWM_MAX    100.0

/* アラーム表示 */
#define INDICATOR_ON  GPIO_HIGH
#define INDICATOR_OFF GPIO_LOW

#define __PWM

#endif
```

プログラムファイル

モジュール pwm.py です。オブジェクト生成時に GPIO ポートを PWM 出力に初期化します。操作が呼ばれたら、上下限のチェックを行ってから、デューティサイクルを書き込みます。実機を使わないときは、出力をスクリーンに表示します。

pwm.py

```
/* パルス幅変調出力
      初版： 2014/12/31 Chuji
      最新版： 2017/2/20  - アラーム灯表示を追加

Class pulse_width_modulator
属性:
   pwm:  PWM オブジェクト
操作:
   output:  PWM の出力デューティサイクル（%）を変える

Class status_indicator
属性:
   indicator: アラーム表示灯オブジェクト
操作:
   output: 表示灯の点灯/消灯させる
*/

#ifndef __PWM_PY

#include "./include/gpio.h"
#include "./include/pwm.h"

class pulse_width_modulator:
  def __init__(self, port):

#ifdef BCM2835  /* 実機の場合 */
    GPIO_setup(port, GPIO_OUT)
    self.pwm = GPIO.PWM(port, PWM_FREQUENCY)
    self.pwm.start(PWM_MIN)

#else    /* シミュレーション用 */
    print "PWM: ", "0.0%  at initialization."
#endif


/* PWM のデューティサイクルを設定する
     output(duty: duty cycle in 0-100%)  */
  def output(self, duty):
    if duty < PWM_MIN:
      duty = PWM_MIN
    elif duty > PWM_MAX:
      duty = PWM_MAX

#ifdef BCM2835  /* 実機の場合 */
    self.pwm.ChangeDutyCycle(duty)

#else    /* シミュレーション用 */
    print "PWM: ", duty, "%"
#endif

class status_indicator:
  def __init__(self, port):
    self.indicator = port
#ifdef BCM2835
    GPIO_setup(self.indicator, GPIO_OUT)
    GPIO_output(self.indicator, INDICATOR_ON)
#endif

  def output(self, flag):
#ifdef BCM2835
    GPIO_output(self.indicator, flag)
#else
    print "Status indicator:", flag
#endif

#define __PWM_PY

#endif
```

4.3.2アラーム表示

オブジェクト定義

クラス：status_indicator

属性：indicator（表示回路）

操作：output（出力制御）

ヘッダーファイル

アラーム表示の定義を前節の pwm.h で行います。

プログラムファイル

ソースコードは前節の pwm.py に含まれています。オブジェクト生成時に GPIO ポートを出力に初期化します。操作が呼ばれたら、表示出力を書き込みます。実機を使わないときは、出力をスクリーンに表示します。

プログラミング書法

ソフトウエアを仕事にしている友人から、「いい本だよ」と勧められたことがあります。UNIX を開発したリッチーの同僚である、カーニハンたちが書いた「プログラム書法」（共立出版。原著は The Elements of Programming Style: 1974/78）という本です。分かりやすくて間違いの少ないプログラムを書くための、具体例に富んだ指南書といったところでしょうか。例に挙げられている FORTRAN や PL/I を使った経験はありませんでしたが、「なるほど」とか、「あるある」と思わせる内容でした。

続いてカーニハンたちが書いた「プログラミング作法」（アスキー。原著は The Practice of Programming: 1999；カドカワから再版）では、デバグや検証のための気配りも説明してくれています。例に C 言語が使われているので、分かりやすいと感じました。

この本の説明の基礎になっている、マジックナンバーを避けること、検証時の境界値法、検証の自動化などは、カーニハンたちの本から学んだことです。これだけ昔の本なのに、主張が古びることもなく、いまだに示唆に富んでいるというのは驚きです。再版されたのも頷けます。なお、両書の英語版は PDF ファイルを自由にダウンロードできます。

4.4　I2C バス入出力

オブジェクト定義

クラス：i2c_device

属性：channel（I2C バスオブジェクト）、addr（I2C バスアドレス）、type（デバイスの型）、data（入出力データ）

操作：read_byte, read_word, write_byte, write_block, wait（読み込み操作の出力はプロセス値を表す構造体）

ヘッダーファイル

I2C バスのポートとデバイスアドレスを定義します。I2C デバイスには、読み書き可能なもの（温度センサ）と書き込み専用（液晶）があり、i2c.py のなかでチェックするようにしています。

i2c.h

```
/* I2C バスの定義とアドレス
  初版:  2016/10/29 Chuji
  最新版:  2016/12/3 複数の温度センサを使えるようにした
*/

#ifndef __I2C

#include "gpio.h"

#ifdef BCM2835  /* 実機のみ */
import smbus
#endif

/* I2C バスのピン指定 */
#define GPIO_I2C_SDA  2
#define GPIO_I2C_SCL  3

/* デバイスのタイプ定義 */
#define I2C_WRITE_ONLY  0
#define I2C_READ_WRITE  1


/* I2C バスのアドレス定義 */

#define I2C_LCD_ADDR     0x3E

#ifdef STTS751  /* 温度センサは STTS751 */

#define I2C_TEMP_ADDR    0x48 /* プルアップ抵抗 7.5 kΩ
*/
#define I2C_TEMP0_ADDR   I2C_TEMP_ADDR
#define I2C_TEMP1_ADDR   0x38 /* プルアップ抵抗 20 kΩ
*/

#else           /* 温度センサは ADT7410 */

#define I2C_TEMP_ADDR    0x48 /* アドレスピンは全て接地
*/
#define I2C_TEMP0_ADDR   I2C_TEMP_ADDR
#define I2C_TEMP1_ADDR   0x49 /* A0 ピンのみプルアップ
Vdd */
#endif

/* LCD の待ち時間 */
#define I2C_WAIT 0.001 /* 1ms */

#define __I2C

#endif
```

プログラムファイル

I2C バスへの読み書きを行うモジュールです。クラスを生成するとき、デバイスの I2C バスアドレスとタイプ（読み書き可能、または書き込み専用）を渡します。このモジュールは液晶表示ユニットと温度センサユニット両方のドライバーにサービスを提供します。検証途中では、それぞれのモジュールにインクルードする場面があります。多重定義を防ぐために、#ifndef __I2C …... #define __I2C #endif で囲って、一度だけコードが残るようにしています。

BCM2835 が#define されていれば、I2C バスにアクセスしようとします。#define されていない（シミュレーション）ときは、出力要求をコンソールに表示したり、読み取ったものとして処理を進めるためのデータ入力を求めたりします。

読み込み操作は、デバイスの内部レジスタアドレスを指定して行い、ステータスとペアになったプロセス値を返します。書き込み専用デバイスから読み込もうとすると、ステータス BAD を返します。ワード読み込みでは、指定したレジスタから連続した 2 バイトを返します。

書き込み操作は、デバイスの内部レジスタアドレスと書き込むデータを渡し、成功すれば GOOD を返します。複数バイト書き込みでは、同じレジスタアドレスに何回も書き込みます。これは、液晶の表示データなどを想定しています。別途設定する書き込み位置は、1 バイトを書き込むと次（右側）の書き込み位置に自動的に移るからです。

i2c.py

```
/* i2c.py  I2C バスの駆動
   初版：  14 February 2015 Chuji
   最新版：  23 January 2019

改訂履歴
  23 January 2019 余計な処理を除去して簡略化
  19 February 2017 I/O エラーに対応

Class: i2c_device
属性:
  channel  I2C バス
  addr     I2C バスアドレス
  type     動作モード(Write only or Read/Write)
  data     I2C から読み取ったデータ構造体(値とステータス)

操作:
  read_byte   デバイスから 1 バイト読み取る
  read_word   デバイスから 1 ワード(2 バイト)読み取る
  write_byte   デバイスに 1 バイトのデータを書き込む
  write_block   デバイスる複数バイトのデータを連続して書き込む
  wait     処理の終了待ちタイマー
*/


#ifndef __I2C_EMU  /* 複数回取り込み防止 */

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

#include "./include/i2c.h"

class i2c_device:
  def __init__(self, addr, device):
    self.addr = addr
    self.type = device
    self.data = process_value()
#ifdef BCM2835
    self.channel = smbus.SMBus(1)
#endif


  /* 1 バイトの読み込み */
  def read_byte(self, reg): /* 戻り値は構造体
*/

    if self.type == I2C_READ_WRITE:
#ifdef BCM2835
      try:
        self.data.value =
self.channel.read_byte_data(self.addr, reg)
        self.data.status = STATUS_GOOD
      except IOError:
        self.data.status = STATUS_BAD
#else
      print "I2C read byte data from
address", hex(self.addr),":(Reg)", hex(reg),
"in hex?"
      self.data.value = input("? ")
      self.data.status = STATUS_GOOD
#endif
    else:
      self.data.status = STATUS_BAD
    return self.data


  /* 1 ワードの読み込み */
  def read_word(self, reg): /* 戻り値は構造体
*/


    if self.type == I2C_READ_WRITE:
#ifdef BCM2835
      try:
        self.data.value =
self.channel.read_word_data(self.addr, reg)
        self.data.status = STATUS_GOOD
      except IOError:
        self.data.status = STATUS_BAD
#else
      print "I2C read word data from
address", hex(self.addr),":(Reg)", hex(reg),
"in hex?"
      self.data.value = input()
      self.data.status = STATUS_GOOD
#endif
    else:
      self.data.status = STATUS_BAD
    return self.data

  /* 1 バイトの書き込み */
  def write_byte(self, reg, val): /* 戻り値は結
果のステータス */
#ifdef BCM2835
    try:
      self.channel.write_byte_data(self.addr,
reg, val)
      self.data.status = STATUS_GOOD
    except IOError:
      self.data.status = STATUS_BAD
    self.wait(I2C_WAIT)

#else
    print "I2C byte write:", hex(val), "=>",
hex(self.addr), ":", hex(reg)
    self.data.status = STATUS_GOOD
```

```
#endif
    return self.data.status


  /* 複数バイトデータの連続書き込み */
  def write_block(self, reg, string): /* 戻り
値は結果のステータス */

    for char_pos in range(len(string)):
      put_char = string[char_pos]
      self.write_byte(reg, ord(put_char))
    return self.data.status

  /* 処理待ちタイマー */
  def wait(self, interval):
#ifdef BCM2835
    sleep(interval)
#else
    print "wait for", interval, "sec"
#endif

#define __I2C_EMU

#endif
```

4.5　温度測定（温度センサドライバー）

温度センサのレジスタ構成と設定

二種類の温度センサのレジスタ構成を次ページの表にまとめます。両者の構造は似ていますが、微妙に異なっています。大部分の設定はデフォルト値でいいので、必要なところだけ設定します。

2 バイトにわたる温度データの形式も少し異なっています。

ファジイ制御

エアコンなどに搭載されている「ファジイ制御」とはどんなものでしょうか？　もともとは人工知能（AI）に、人間に近い推論の基準（ルール）を教え込むために作り上げられたものです。たとえば、

- 室温がちょっと低かったら、すこし暖房を強める
- 室温が快適だったら、空調の動作をあまり変えない

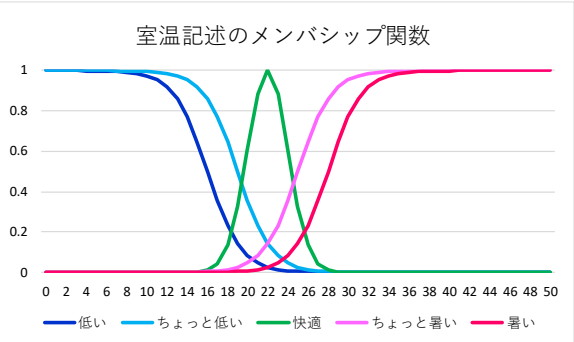
といったルールを設けます。ここで、『ちょっと』とか『快適』、あるいは『すこし』とか『あまり』といった、あいまいな表現を、コンピュータが分かる形にするのがファジイ論理です。

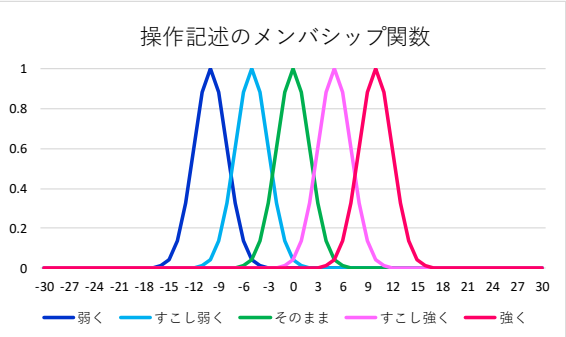
あいまいさのないルールでは、記述の正しさを、真か否かという二値のどちらかであるとします。たとえば、

- 室温が 21℃以下だったらヒーターを点け、23℃以上だったら切る

というルールはオンオフ制御のことです。

ファジイ論理では、記述がどのくらい正しいかを、0 から 1 までの数値（メンバシップ関数と言います）で表現します。室温が 15℃だったら「ちょっと低い」は 0.9 だけ正しいが、20℃だったら 0.3 になるといった具合です。下図の左側は横軸を室温に取った、ルール前半の室温記述のメンバシップ関数です。その右は暖房の操作記述のメンバシップ関数です。実際にメンバシップ関数の形を決めるには、『人間の知恵』が必要です。





ファジイ制御では、このルールから操作量を決めますが、その方法は人為的で、何通りもあり得ます。例えば、ある温度で、各ルールの室温記述メンバシップ関数から正しさを調べ、それから操作記述の加重平均を作ったら、その重心を操作量にします。こんな制御を使うと、たしかに室温は 22℃『付近』に落ち着くようになります。ただ、この程度のルールで行う制御品質は、オンオフ制御（43 ページ）と比例制御の中間『くらい』にしかなりません。ファジイ制御のご利益が得られるルールについては、後（47 ページ）で検討します。

レジスタアドレス	ADT7410A		STTS751-0	
	レジスタ名	デフォルト値	レジスタ名	デフォルト値
0x00 (R/-)	Temperature（上位バイト）	0x00	Temperature（上位バイト）	undefined
0x01 (R/-)	Temperature（下位バイト）	0x00	Status	undefined
0x02 (R/-)	Status	0x00	Temperature（下位バイト）	undefined
0x03 (R/W)	Configuration	0x00	Configuration	0x00
0x04 (R/W)	T_{HIGH}（上位バイト）	0x20	Conversion rate	0x04
0x05 (R/W)	T_{HIGH}（下位バイト）	0x00	T_{HIGH}（上位バイト）	0x55
0x06 (R/W)	T_{LOW}（上位バイト）	0x05	T_{HIGH}（下位バイト）	0x00
0x07 (R/W)	T_{LOW}（下位バイト）	0x00	T_{LOW}（上位バイト）	0x00
0x08 (R/W)	T_{CRIT}（上位バイト）	0x49	T_{LOW}（下位バイト）	0x00
0x09 (R/W)	T_{CRIT}（下位バイト）	0x80	-	-
0x0A (R/W)	T_{HYST}	0x05	-	-
0x0B (R/-)	ID	0xCX	-	-
0x0F (-/W)	-	-	One Shot	N/A
0x20 (R/W)	-	-	THERM limit (T_{CRIT})	0x55
0x21 (R/W)	-	-	THERM hysteresis (T_{HYST})	0x0A
0x22 (R/W)	-	-	SMBus timeout enable	0x80
0x2F (-/W)	Software reset	N/A	-	-
0xFD (R/-)	-	-	Product ID	0x00
0xFE (R/-)	-	-	Manufacturer ID	0x53
0xFF (R/-)	-	-	Revision Number	0x01

ADT7410A の温度データ

上位バイト								下位バイト							
Sign	128℃	64℃	32℃	16℃	8℃	4℃	2℃	1℃	1/2℃	1/4℃	1/8℃	1/16℃	1/32℃	1/64℃	1/128℃

注：16 ビット分解能設定時。13 ビット分解能設置時は下位 3 ビットをマスクして使うこと。

STTS751-0 の温度データ

上位バイト								下位バイト							
Sign	64℃	32℃	16℃	8℃	4℃	2℃	1℃	1/2℃	1/4℃	1/8℃	1/16℃	0	0	0	0

Status レジスタのビット構成は以下のとおりです。RDY/ビットと BUSY ビットは測定動作が終了したことを示しています。それ以下のビットは温度アラームですが、ここでは使いません。

温度センサ	b7	b6	b5	b4	b3	b2	b1	b0
ADT7410A	RDY/	T_{CRIT}	T_{HIGH}	T_{LOW}	-	-	-	-
STTS751-0	BUSY	T_{HIGH}	T_{LOW}	-	-	-	-	T_{CRIT}

RDY/=LOW、BUSY=LOW：温度測定終了。

ステータスレジスタ

Configuration レジスタは、温度センサの機能を次のように設定します。

ADT7410A のコンフィギュレーション

bit	ビット名称	設定方法	実際の設定
7	Resolution	測定分解能＝0：13 ビット、1：16 ビット	1 又は 0
6	Operation mode	動作モード＝00：繰り返し測定、01：ワンショット測定、 10：測定周期 1 秒固定、 11：測定停止	0
5			0
4	INT/CT mode	INT/CT ピンの動作モード（使用しない）	0
3	INT polarity	INT ピンの論理極性（使用しない）	0
2	CT polarity	CT ピンの論理極性（使用しない）	0
1	Fault queue	CT/INT ピンが動作するまでのアラーム回数-1 を設定（使用しない）	0
0			0

Configuration レジスタ（ADT7410A）

STTS751 のコンフィギュレーション

bit	ビット名称	設定方法	実際の設定
7	MASK1	EVENT ピンの動作＝0：動作禁止、1：動作許可	0
6	RUN/STOP	動作モード＝0：繰返し測定、1：動作停止（ワンショット）	1
5	0	常に 0	0
4	-	-	0
3	Resolution	測定分解能＝00：10 ビット、01：11 ビット、10：9 ビット、11：12 ビット	1
2			1
1	-	-	0
0	-	-	0

Configuration レジスタ（STTS751）

ADT7410A の動作モードは繰り返し測定を選びます。STTS751 は Conversion rate レジスタの下位 4 ビット（R=0～9）に変換レートを設定します。変換レートは毎秒 2^{R-4} 回となるので、12 ビット分解能で最速の R=0x3（毎秒 8 回）を採用しました。

ADT7410A の測定値レジスタは、上下バイトのアドレスが連続しているので、一度に読み出すことができます。いっぽう STTS751 の測定値読みだしには、ちょっと手間が必要です。アドレスの離れた二つのレジスタから、それぞれ読み出すことが問題のもとになっています。2 回の読み出し動作には、それなりの時間がかかるので、その間に測定値が更新されてしまうと、上下バイトの整合が取れません。温度が 0.1℃変化しただけで、最悪 1℃の誤差が出てきます。対策としては、

1. 下位バイトを使わない（分解能は 1℃になる）。
2. めったに起こることではないので無視する。
3. 読み取りの間に測定値が更新されないようにする。

の 3 つが考えられます。どれをとっても、あまり実用的な問題はないと思いますが、せっかくなので対策 3 を取ることにしました。STTS751 の動作モードをワンショット（トリガが加えられた時に一回だけ測定する）に設定しておきます。測定値を読み出すときには、まず One Shot レジスタにデータ（値は何でもいい）を書き込みます。変換中は Status レジスタの BUSY ビットがセットされているので、これがクリアされるまで待ってから、データの読み出しを行います。

オブジェクト定義

クラス： thermometer

属性：i2c（I2C バスオブジェクト）、pv（測定温度）、simulation（温度シミュレターオブジェクト）

操作：read_temp（温度読み取り）

ヘッダーファイル

温度センサ用定義ファイルです。二種類の温度センサに対して、同じ機能には同じ定義名を与えるようにしています。

thermo.h

```
/* 温度センサの定義
   初版：2014/12/31 Chuji
   最新版：2017/3/20 ADT7410 で 16 ビット分解能を可能にした
*/

#ifndef __TEMP

#define TEMP_BYTE  8
#define TEMP_SIGN_MASK  0x8000
#define TEMP_INT_MASK  0x7fff
#define TEMP_WORD_MASK  0xffff
#define  TEMP_HIGH_MASK  0xff00
#define TEMP_LOW_MASK   0xff
#define TEMP_BYTE_LEN   8

/* シミュレーションで使う周囲温度と初期温度 */

#ifndef AMBIENT_TEMP
#define AMBIENT_TEMP  20.0
#endif

#ifndef INITIAL_TEMP
#define INITIAL_TEMP  60.0
#endif

/* 複数センサを使うための定義 */

#define TEMP_SENSOR0  0
#define TEMP_SENSOR  TEMP_SENSOR0
#define TEMP_SENSOR1  TEMP_SENSOR+1

#ifndef STTS751  /* 温度センサ ADT7410 の定義 */

#define TEMP_CONFIG_REG  0x03
#define TEMP_VALUE_REG  0x00
#define TEMP_STATUS_REG  0x02
#define TEMP_RESET_REG  0x2F

#ifdef TEMP_HIGH_RESOLUTION
#define TEMP_CONFIG_DATA  0x80  /* 16 ビット分解能 */
#else
#define TEMP_CONFIG_DATA  0x00  /* 13 ビット分解能 */
#endif

#define TEMP_STATUS_MASK  0x80
#define TEMP_STATUS_READY  0x00
#define TEMP_RESET_DATA    0x00

#ifdef TEMP_HIGH_RESOLUTION  /* 上位有効バイトの定義 */
#define  TEMP_HIGH_BYTE    0xff00
#else
#define  TEMP_HIGH_BYTE    0xf800
#endif

#define TEMP_LOW_BYTE    0xff

/* ステータスレジスタの表示値 */
#define TEMP_BUSY    0x80
#define TEMP_READY    0x00
```

```
#ifdef TEMP_HIGH_RESOLUTION   /* 分解能 */
#define TEMP_FRACT    128.0 /* 分解能の逆数 */
#define TEMP_RESOLUTION    1/TEMP_FRACT /* 1/128℃
*/
#define TEMP_GET_HIGH    5 /* 上位バイトを得るための右
シフト回数 */
#define TEMP_GET_LOW    0 /* 下位バイトを得るための左シ
フト回数 */

#else
#define TEMP_FRACT    16.0 /* 分解能の逆数 */
#define TEMP_RESOLUTION    1/TEMP_FRACT /* 1/16℃
*/
#define TEMP_GET_HIGH    5 /* 上位バイトを得るための右
シフト回数 */
#define TEMP_GET_LOW    3 /* 下位バイトを得るための左シ
フト回数 */
#endif

#define HIGHEST_TEMP    150 /* 温度測定値の上限 */
#define LOWEST_TEMP    -55 /* 温度測定値の下限 */

#else  /* 温度センサ STTS751 の定義 */

#define TEMP_CONFIG_REG    0x03  /* 設定レジスタ */
#define TEMP_CONFIG2_REG  0x04  /* 変換レートレジスタ
*/
#define TEMP_VALUE_REG    0x00  /* 温度測定値上位バイ
ト */
#define TEMP_VALUE2_REG    0x02  /* 温度測定値下位バイ
ト */
#define TEMP_STATUS_REG    0x01  /* ステータスレジスタ
*/
#define TEMP_ONESHOT_REG  0x0f  /* ワンショット動作ト
リガレジスタ */

#define TEMP_CONFIG_DATA  0x4c  /* 12 ビット分解能、ワ
ンショット動作 */
#define TEMP_CONFIG2_DATA  0x07  /* 変換速度毎秒 8 回
*/
#define TEMP_STATUS_MASK  0x80
#define TEMP_STATUS_READY  0x00
#define TEMP_ONESHOT_CMD  0x00  /* 実際には何でもいい
*/

/* status register */
#define TEMP_BUSY    0x80
#define TEMP_READY    0x00

#define TEMP_FRACT    16.0 /* 分解能の逆数 */
#define TEMP_RESOLUTION    1/TEMP_FRACT /* 1/16℃
*/
#define TEMP_GET_HIGH    4 /* 上位バイトを得るための右
シフト回数 */
#define TEMP_GET_LOW    4 /* 下位バイトを得るための左シ
フト回数 */

#define HIGHEST_TEMP    125 /* 温度測定値の上限 */
#define LOWEST_TEMP    -40 /* 温度測定値の下限 */

#endif

#define __TEMP

#endif
```

プログラムファイル

プログラムモジュールは以下のとおりです。
thermometer オブジェクトを初期化すると、温度センサに各種の設定値を書き込みます。温度センサは同じ IC なら 2 個まで扱えるようにしてあります。

温度を読み取ろうとしたとき、TEMP_SIMULATION が定義されていると、シミュレーションモデルが起動されて、モデルの現在温度を返します。そうでないとき、ADT7410A の場合は、2 バイトの温度データを読み取ります。通信によってメモリ上に置かれた 2 バイトの、どちらを上位とするかは CPU に依存します。Raspberry Pi の場合は上下を交換する必要があります。

STTS751 の場合は、まずワンショットトリガをかけ、ステータスレジスタの BUSY ビットがクリアされるのを待ちます。次に上位バイトと下位バイトをそれぞれ読み出し、ワードデータにまとめます。

そのあとの処理はどちらの温度センサでも同じで、符号を除いた 2 バイトデータに分解能（最下位ビットが何℃になるか）を掛け、符号をつけます。最後に上下限のチェックを行ってから値を返します。ステータスが BAD になるのは、通信エラーが起きたときと、測定値が上下限を超えたときです。

温度センサによらず、処理が同じ手順で記述できているのは、レジスタの構造やシフト回数を表すマクロ定義を thermo.h で切り替えているからです。

thermo.py

```
/* 温度センサドライバー
      初版： 2014/12/31 Chuji
      最新版： 2019/1/23 - 簡略化

Class: thermometer
属性:
   i2c:    I2C バスオブジェクト
   pv:     温度（I2C バス受信値またはシミュレーション値）
   simulation: 温度モデルオブジェクト
操作:
   read_temp: 現在温度の読み取り


使用可能な温度センサー:
   ADT7410 アナログデバイス製（デフォルト）
   STTS751 STMicroelectronics 製 - "STTS751"  を
#define する
    これらのデバイスは、"BCM2835"を#define していないと、
I2C バスの読み取り値を聞かれる

   温度モデル "TEMP_SIMULATION"を#define する
*/

#ifdef BCM2835
#include "./include/gpio.h"
#endif

#include "./include/process_value.h"
#include "./include/thermo.h"

#include "i2c.py"

#ifdef TEMP_SIMULATION
#include "sim-model.py"
#endif

class thermometer:
  def __init__(self, device):
    if device == TEMP_SENSOR1:
      addr = I2C_TEMP1_ADDR
    else:
      addr = I2C_TEMP0_ADDR
```

```
    self.i2c = i2c_device(addr, I2C_READ_WRITE)
    self.pv = process_value()

    /* ターゲットハードウエアを設定する */
    self.i2c.write_byte(TEMP_CONFIG_REG,
TEMP_CONFIG_DATA)

#ifdef STTS751 /* STTS751 の場合は第二設定レジスタを操作
*/
    self.i2c.write_byte(TEMP_CONFIG2_REG,
TEMP_CONFIG2_DATA)
#endif

#ifdef TEMP_SIMULATION
    self.simulation = thermal_model()
#endif


/*温度の読み取り */

#ifdef TEMP_SIMULATION  /* シミュレーション時には加熱
（操作）量を与える */
  def read_temp(self, mv):
    self.pv.value = self.simulation.temperature(mv)
    self.pv.status = STATUS_GOOD
    return self.pv
#else
  def read_temp(self):

#ifndef STTS751    /* ADT7410 の場合 */
    self.pv = self.i2c.read_word(TEMP_VALUE_REG)
    if self.pv.status != STATUS_GOOD:
      return self.pv
    word_data = self.pv.value
      /* 上下バイトの交換 */
    data = ((word_data >> TEMP_BYTE_LEN) &
TEMP_LOW_MASK) | ((word_data << TEMP_BYTE_LEN) &
TEMP_HIGH_MASK)

#else    /* STTS751 の場合 */
               /* ワンショット変換をトリガする */
    self.i2c.write_byte(TEMP_ONESHOT_REG,
TEMP_ONESHOT_CMD)

               /* A/D 変換が終わるまで待つ */
    self.pv = self.i2c.read_byte(TEMP_STATUS_REG)
    if self.pv.status == STATUS_BAD:
      return self.pv

    while (self.pv.value & TEMP_STATUS_MASK) !=
TEMP_STATUS_READY:
      self.pv = self.i2c.read_byte(TEMP_STATUS_REG)
      if self.pv.status == STATUS_BAD:
        return self.pv

    self.pv = self.i2c.read_byte(TEMP_VALUE_REG)
    if self.pv.status == STATUS_GOOD:
      datah = self.pv.value
      self.pv = self.i2c.read_byte(TEMP_VALUE2_REG)
    if self.pv.status == STATUS_GOOD:
      datal = self.pv.value
      data = ((datah << TEMP_BYTE_LEN) &
TEMP_HIGH_MASK) | datal
#endif

    /* 負の数値をチェック */
    if data & TEMP_SIGN_MASK:  /* 温度が負の場合 */
      self.pv.value = (((~data & TEMP_INT_MASK) >>
TEMP_GET_LOW) + 1) * -TEMP_RESOLUTION
    else:
      self.pv.value =  (data >> TEMP_GET_LOW) *
TEMP_RESOLUTION

    /* 温度上下限のチェック */
    if (self.pv.value > HIGHEST_TEMP) |
(self.pv.value < LOWEST_TEMP):
      self.pv.status = STATUS_BAD

    return self.pv
#endif
```

4.6　液晶表示（液晶表示器ドライバー）

液晶表示器はさまざまなタイプがありますが、できるだけ多くの表示器に対応できるようにドライバーを汎用化しておきます。多くの液晶表示器はルネッサステクノロジ社の液晶コントローラ HD44780 か、その互換品を使っているので、制御方法は共通です。SunLike Display 社のデバイスはすべて、制御レジスタは共通で、文字の位置を示すデータアドレスのみが異なるので、#define するだけで同じインクルードファイルを使うことができます。具体的には以下のデバイスが使えます。実際に使うのは SC1602 なので、16 文字×2 行がデフォルトです。

デバイス	文字構成	#define
SC801	8 文字×1 行	LCD_8, LCD_1_LINE
SC802	8 文字×2 行	LCD_8
SC1004	10 文字×4 行	LCD_10, LCD_4_LINE
SC1601	16 文字×1 行	LCD_1_LINE
SC1602	16 文字×2 行	
SC1604	16 文字×4 行	LCD_4_LINE
SC2002	20 文字×2 行	LCD_20
SC2004	20 文字×4 行	LCD_20, LCD_4_LINE
SC4002	40 文字×1 行	LCD_40, LCD_1_LINE

液晶表示器を指定するマクロ

この液晶表示器のアドレスは、下表のように固定されています。

I2C アドレス	0x3E（ハードウエア固定）
コマンドレジスタアドレス	0x00
データレジスタアドレス	0x40

液晶表示器アドレス

コマンドレジスタには一つしかアドレスがありません。書き込むデータの上位から見て、最初に 1 が立っているのが何ビット目かでコマンドを認識します。次ページにある表の設定データ欄で、0x04+などとなっている部分がそれに相当します。その下のビットがどういう意味を持っているか、データ詳細欄に説明してあります。

文字コードを書き込むデータレジスタのアドレス AC は 7 ビットしかないので、アドレス範囲は 0x00~0x7F までに限られます。そのため SC4004（40 文字×4 行）はサポートできません。最初にデータアドレスを設定し、データレジスタに一文字（1 バイト）書き込むと、データアドレスは自動的に右へ移動するように設定しておきます。

Function Set コマンドの IS ビットにより、そのあとに書き込まれたコマンドは、以下の拡張命令あるいは非拡張命令の一方を選んで解釈します。今回は拡張命令のみ使用します。

マニュアルによると、液晶表示器の初期化は次の手順のとおりです。まず Function Set コマンドの IS ビットを立ててから、拡張命令を順次書き込み、最後にボルテージフォロアの増幅率を設定します。このあとボルテージフォロアが安定するまで 200ms 以上待ってから、IS ビットをクリアし、他のコマンドを書き込みます。オブジェクト生成（__INIT__ 部）のプログラムコードを見てください。

コマンド

コマンド	設定データ	データ詳細	採用する設定
Clear Display	0x01	データ初期化（処理に約 1ms かかる）	0x01
Return Home	0x02	アドレス初期化（処理に約 1ms かかる）	0x02
Entry Mode Set	0x04 +I/D+S	I/D: カーソル移動方向= 0x2:右、0x0:左 S: 全 DDRAM シフト= 0x1:許可、0x0:禁止	0x06
Display On/Off	0x08+D+C+B	D: データ表示＝0x4:オン、0x0:オフ C: カーソル表示＝0x2:オン、0x0:オフ B:カーソル点滅＝0x1:オン、0x0:オフ	0x0C
Function Set	0x20 +DL+N +DH+IS	DL: アクセスデータ長（bit）＝0x10:8、0x0:4 N: 表示行数＝0x8: 2 行 0x0: 1 行 DH: 0x4：二倍長文字表示、0x0:通常文字 IS: 0x1: 拡張命令が続く、0x0: 拡張なし	0x38 +IS
Set DDRAM address	0x80 +AC	AC（0x00～0x7F ）：次に書き込む DDRAM（文字データ）アドレスを設定	0x80 +AC

非拡張命令（IS=0x0）

コマンド	設定データ	データ詳細	採用する設定
Cursor/Display Shift	0x10 +SCRL	SCRL: カーソル・表示のシフト命令 0x0:カーソル左シフト 、0x4:右シフト 0x8:カーソル・表示左シフト、 0xC: 右シフト	使用しない
Set CGRAM	0x40 +AC	AC（0x00～0x3F ）：次に書き込む CGRAM（キャラクタジェネレータ）アドレスを設定	使用しない

拡張命令（IS=0x1）

コマンド	設定データ	データ詳細	採用する設定
OSC frequency	0x1+BS+F	BS：発信器バイアス＝0x4: 1/4、0x0: 1/5 F（0x0～0x7）: 内部クロック（122～347Hz）	0x14 （183Hz, 1/4）
ICON address	0x40 +AC	AC（0x0～0xF）：アイコンアドレス	使用しない
Power/ICON/contrast	0x50 +Ion +Bon +C54	Ion: アイコン表示（機能しない）→0x0 Bon: 電源ブースタ（VDD を選択）＝0x4: 3.3V、0x0: 5V C54: （0x0～0x3）：コントラストの上位ビット	0x56 (3.3V)
Follower	0x60 +Fon +Fab	Fon: ボルテージフォロア ON(0x8)/Off(0x0) Fab: （0x0～0x7）: フォロア増幅率	0x6C （2 倍）
Contast	0x70+C	C: （0x0~0xF）:	0x73 (0x13)

オブジェクトモデル

クラス： lcd_device

属性：lcd（液晶表示器の I2C モデル）

操作：put_string（文字列を表示する）

ヘッダーファイル

ヘッダーファイルでは、コマンドの具体的な内容と、表示行の左端のデータアドレス、表示可能な文字数と行数を定義しています。

lcd.h

```
/* lcd.h
   初版: 24, December, 2014 Chuji
   最新版: 20, Jan 2019 - イメージ生成部を分離
*/

/* SunLike Displays 社製 LCD SC1602C シリーズのドライバー
    4 行 x40 文字はサポート対象外（i2c バスの制限） */

#ifndef __LCD

/* LCD の論理モデル */

/* 表示アドレスの定義 */

/*   表示文字のアドレス（16 進表示）16 文字幅デバイスの場合
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |00|01|02|03|04|05|06|07|08|09|0A|0B|0C|0D|0E|0F|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |40|41|42|43|44|45|46|47|48|49|4A|4B|4C|4D|4E|4F|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |10|11|12|13|14|15|16|17|18|19|1A|1B|1C|1D|1E|1F|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |50|51|52|53|54|55|56|57|58|59|5A|5B|5C|5D|5E|5F|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

    注: 20 文字幅デバイスの場合、3 行目と 4 行目はそれぞれ$14 と$54 で始まる
        10 文字幅デバイスの場合、3 行目と 4 行目はそれぞれ $0A と$5A で始まる
*/


/* LCD の内部アドレス定義 */

#define FIRST_ROW  0x0
#define SECOND_ROW  0x40

#ifdef LCD20
#define THIRD_ROW  0x14  /* 20 文字幅デバイス SC2002/2004 */
#define FOURTH_ROW  0x54
#define LCD_LINE_LEN  20
#else
#ifdef LCD10
#define THIRD_ROW  0x0A  /* 10 文字幅デバイス SC1004 */
#define FOURTH_ROW  0x5A
#define LCD_LINE_LEN  10
#else        /* 8/16/40 文字幅デバイス */
#define THIRD_ROW  0x10  /* 10 文字幅デバイス SC1004 */
#define FOURTH_ROW  0x50
#ifdef LCD8
#define LCD_LINE_LEN  8  /* 8 文字幅デバイス SC0801/0802 */
#else
#ifdef LCD40
#define LCD_LINE_LEN  40  /* 40 文字幅デバイス SC4001/4002 */
#else
#define LCD_LINE_LEN  16  /* 16 文字幅デバイス(既定)SC1602/1604 */
#endif
#endif
#endif
#endif

Char_addr = [FIRST_ROW, SECOND_ROW, THIRD_ROW, FOURTH_ROW]

#define LCD_FIRST_ROW  0
#define LCD_FIRST_CHAR  0

/* 最大行数 */
#ifdef LCD_1_LINE
#define LCD_LAST_ROW  0
#else
#ifdef LCD_4_LINE
#define LCD_LAST_ROW  3
#else
#define LCD_LAST_ROW  1 /* 既定 */
#endif
#endif

/* 制御レジスタアドレス */
#define LCD_CMD_REG    0x00
#define LCD_DATA_REG    0x40

/* 制御コマンド */
#define LCD_CMD_CLEAR    0x01  /* 全表示のクリア */
#define LCD_CMD_HOME    0x02  /* カーソルをホーム位置に */
#define LCD_CMD_ENTRY_MODE  0x06  /* 文字を入力するとカーソルを移動 */
#define LCD_CMD_DISPLAY_ON  0x0c  /* カーソルを点滅させない */
#define LCD_CMD_CURSOR    0x14  /* カーソルは右に移動 */
#define LCD_CMD_FUNCTION  0x38  /* 表示フォントの選択 */
#define LCD_CMD_EXTENTION  0x39  /* 拡張コマンドの開始 */
#define LCD_CMD_OSC_FREQ  0x14  /* 内部クロックの設定 */
#define LCD_CMD_CONTRAST  0x73  /* コントラストの設定 */
#define LCD_CMD_POWER    0x56  /* 電源の設定 */
#define LCD_CMD_FOLLOWER  0x6C  /* フォロア回路をオン */
#define LCD_CMD_END_EXT    LCD_CMD_FUNCTION

#define LCD_CMD_CG_ADDR    0x40
#define LCD_CMD_CHAR_ADDR  0x80

/* LCD 初期化にかかる時間 */
#define LCD_INIT_TIME  0.2 /* 200ms */


#define __LCD

#endif
```

プログラムファイル

初期化は既に説明したとおりです。文字列の表示は、最初の文字アドレスを計算・設定したら、すべての文字列を書き込むだけです。ただし、表示可能範囲の外に出てしまわないか、事前にチェックします。

PC でのシミュレーション環境で検証するとき、LCD への表示メッセージをすべて出力すると、コンソール表示が非常に多くなり、結果を読み取りにくくなることがあります。そのときは LCD_SILENT を#define することで抑制します。

lcd.py

```
/* lcd.py  LCD 表示デバイスドライバー
   初版：  24 December 2014 Chuji
   最新版：24 January 2019 - display.py を分離

Class: lcd_device
属性:
  lcd:  LCD デバイスオブジェクト
操作:
  put_string:  文字列の表示

*/

#include "./include/lcd.h"
#include "i2c.py"


/* LCD オブジェクト */
class lcd_device:
  def __init__(self):
    self.lcd = i2c_device(I2C_LCD_ADDR,
I2C_WRITE_ONLY)
#ifndef LCD_SILENT
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_FUNCTION)
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_EXTENTION)
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_OSC_FREQ)
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_CONTRAST)
    self.lcd.write_byte(LCD_CMD_REG, LCD_CMD_POWER)
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_FOLLOWER)
    self.lcd.wait(LCD_INIT_TIME)
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_END_EXT)
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_DISPLAY_ON)
    self.lcd.write_byte(LCD_CMD_REG, LCD_CMD_CLEAR)
#endif
  /* LCD への文字列表示
          put_string(row: 表示する行番号（0-3）
                  pos: 表示開始位置（行の左端から）
                  s: 表示する文字列
                  len: 表示する文字数）*/

  def put_string(self, row, pos, s, len):
    if row < LCD_FIRST_ROW:
      row = LCD_FIRST_ROW
    if row > LCD_LAST_ROW:
      row = LCD_LAST_ROW
    if pos < LCD_FIRST_CHAR:
      pos = LCD_FIRST_CHAR
    if pos > LCD_LINE_LEN:
      pos = LCD_LINE_LEN
    if (pos + len) > LCD_LINE_LEN:
      len = LCD_LINE_LEN - pos

    addr = Char_addr[row] + pos
#ifndef LCD_SILENT
    self.lcd.write_byte(LCD_CMD_REG,
LCD_CMD_CHAR_ADDR + addr)
    self.lcd.write_block(LCD_DATA_REG, s[:len])
#endif
```

4.7　文字列生成（数値→文字変換）

液晶表示器に数値を表示するため、数値を文字列に変換します。関数として実装するので、オブジェクトモデルは存在しません。

関数 ftos(数値、書式）の戻り値は文字列で、書式は以下の 4 種類です。実際に使われているのは FORMAT_199_9（測定温度）と FORMAT_999（その他のパラメータ）だけで、あとは初期のバージョンで使っていました。文字列の左側はゼロサプレス（例えば 014 は空白 1 文字+14 になる）されます。

書式名	文字列
FORMAT_199_9	1xxx.x
FORMAT_9_9	x.x
FORMAT_999	xxx
FORMAT_99_9	xx.x

数値→文字列変換書式

ヘッダーファイル

ヘッダーファイルでは、書式名と最大・最小値、固定文字列を定義しています。ゼロ以下の数（マイナス）値には、ゼロと同じ結果を返します。

ftos.h

```
/* ftos.h
   初版: 24, December, 2014 Chuji
   最新版: 20, Jan 2019 - 独立ファイルとして分離

浮動小数点数を指定書式の文字列に変換する

*/


#ifndef __FTOS

/* 書式の定義 */
#define FORMAT_199_9  1  /* 199.9 */
#define  FORMAT_9_9  2  /*   9.9 */
#define FORMAT_999  3  /* 999 */
#define FORMAT_99_9     4       /* 99.9 */

#define MAX_199_9  199.9
#define  MAX_9_9    9.9
#define MAX_999    999
#define MAX_99_9        99.9

#define MIN_199_9  0
#define  MIN_9_9    0
#define MIN_999    0
#define MIN_99_9        0

/* 固定（既定）文字列 */

#define MAX_STRING_199_9  "199.9"
#define  MAX_STRING_9_9  " 9.9"
#define MAX_STRING_999  " 999"
#define MAX_STRING_99_9 "99.9"

#define MIN_STRING_199_9  "  0.0"
#define MIN_STRING_9_9  " 0.0"
#define MIN_STRING_999  "   0"
#define MIN_STRING_99_9 MIN_STRING_9_9

#define SPACE    " "
#define SPACE2    "  "
#define SPACE3    "   "
#define ZERO_POINT  " 0."
#define  DECIMAL_POINT  "."
#define SPACE2_ZERO_POINT  "  0."

#define ALT_SPACE  "_"

#define __FTOS

#endif
```

もう一つのヘッダーファイルは、四捨五入マクロを定義しています。

my_round.h

```
/* 数値の丸め */
/* 2017/3/8 Chuji */

#ifndef __MY_ROUND

/* 浮動小数点数を整数に四捨五入するマクロ */
#define MY_ROUND(x)       int(x + 0.5)

#define __MY_ROUND

#endif
```

プログラムファイル

コードは単純なので、特に説明の必要はないでしょう。書式ごとに str 関数で文字列に変換し、その長さを調べたら書式合わせの空白文字を付け加えます。

小数点以下一位まで表示する書式では、10 倍したものを四捨五入して整数にし、後ろから 2 文字目に小数点を挿入しています。

ftos.py

```
/* ftos.py  浮動小数点数から文字列への変換
   初版： 24 December 2014 Chuji
   最新版： 20 Jan 2019 - LCD ドライバから独立
*/

#include "./include/ftos.h"
#include "./include/my_round.h"

/* 浮動小数点数から指定した書式の文字列に変換する
      ftos(x: 変換する浮動小数点数
           format_def: 書式 (199.9, 999, 9.9) */

def ftos(x, format_def):
  if format_def == FORMAT_199_9: /* 199.9 */
    if x <= MIN_199_9:
      return (MIN_STRING_199_9)
    elif x >= MAX_199_9:
      return (MAX_STRING_199_9)
    else:
      s = str(MY_ROUND(10 * x))
      if len(s) == 4:  /* 1xxx */
        return (s[0:3] + DECIMAL_POINT + s[3])
      elif len(s) == 3: /* xxx */
        return (SPACE + s[0:2] + DECIMAL_POINT +
s[2])
      elif len(s) == 2: /* xx */
        return (SPACE2 + s[0] + DECIMAL_POINT +
s[1])
      elif len(s) == 1: /* x */
        return (SPACE2_ZERO_POINT + s)
      else: /* just in case */
        return (MIN_STRING_199_9)

  elif format_def == FORMAT_9_9: /* 9.9 */
    if x <= MIN_9_9:
      return (MIN_STRING_9_9)
    elif x >= MAX_9_9:
      return (MAX_STRING_9_9)
    else:
      s = str(MY_ROUND(10 * x))
      if len(s) == 2:  /* xx */
        return (SPACE + s[0] + DECIMAL_POINT +
s[1])
      elif len(s) == 1:  /* X */
        return (ZERO_POINT + s)
      else:  /* just in case */
        return (MIN_STRING_9_9)

  elif format_def == FORMAT_999:  /* 999 */
    if x <= MIN_999:
      return (MIN_STRING_999)
    elif x >= MAX_999:
      return (MAX_STRING_999)
    else:
      s = str(MY_ROUND(x))
      if len(s) == 3:  /* xxx */
        return (SPACE + s)
      elif len(s) == 2: /* xx */
        return (SPACE2 + s)
      elif len(s) == 1: /* x */
        return (SPACE3 + s)
      else:
        return (MIN_STRING_999)

  elif format_def == FORMAT_99_9:  /* 99.9 */
    if x <= MIN_99_9:
      return (MIN_STRING_99_9)
    elif x >= MAX_99_9:
      return (MAX_STRING_99_9)
    else:
      s = str(MY_ROUND(10 * x))
      if len(s) == 3:  /* xxx */
        return (s[0:2] + DECIMAL_POINT + s[2])
      elif len(s) == 2: /* xx */
        return (SPACE + s[0] + DECIMAL_POINT +
s[1])
      elif len(s) == 1: /* x */
        return (ZERO_POINT + s)
      else:
        return (MIN_STRING_99_9)
```

ガラス細工について

「アルコールランプでガラス細工はできるか？」という質問がよくあります。答えは「できなくはないが、難しい」です。

加工が容易な軟質ガラス（並ガラス、ソーダガラス）が軟化して、加工しやすくなる温度は、500～600℃と言われています。ガラスには固体が液体に変わるときの融点というものがなく、この範囲の温度で徐々に柔らかくなっていくのです。アルコールランプの炎で一番温度が高いところは約1000℃、ろうそくでも 900℃を越えるので、軟化は可能です。しかしそこは、内炎と外炎の境目で一番上のあたり。高温になる領域が狭いうえ、風で炎が揺らぐため、ガラス管をじゅうぶん加熱するのは、なかなか難しいのです。

逆に家庭用ガスコンロは高温部分が広すぎて危険なので、台所でのガラス細工は止めてください。

ガスバーナーが使える実験室か、ガラス工房の利用をお勧めします。

4.8　表示イメージ（イメージ生成）

液晶表示器に表示するイメージを生成します。

表示フィールド定義

16 文字×2 行に以下のフィールドを設定し、HMI の状態に応じて各フィールドを使い分けるようにします。上の行にパラメータ名、下の行にパラメータの値を表示します。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ラベル 0			ラベル 1					ラベル 2				ラベル 3			
モード			数値 1					数値 2				数値 3			

液晶表示器フィールド定義

表示する数値の書式は、温度測定値が 199.9（5 文字）、他のパラメータは 999（空白+3 文字）です。

各フィールドに、以下のパラメータ名、パラメータ値を表示します。手動設定可能なパラメータには、パラメータ名欄に表示するテキストの前に”@”を表示して、操作を分かりやすくします。

ラベル 0 とモードの欄は 3 文字分のスペースしかないので、”O/S”、”MAN”、”AUT”、”TUN”、”STP”と略記します。

動作状態／パラメータ		ラベルフィールド	数値フィールド
通常動作時	制御モード	ラベル 0 は空白-	モードに実効モード
	温度測定値	ラベル 1 に”Temp”	数値 1
	制御目標値	ラベル 2 に”SP”	数値 2
	制御操作量	ラベル 3 に”MV”	数値 3
PID モード変更	制御モード	ラベル 0 に設定モード	モードに実効モード
	温度測定値	ラベル 1 に”Temp”	数値 1
	制御目標値	ラベル 2 に”SP”	数値 2
	制御操作量	ラベル 3 に”MV”	数値 3
チューニング時	制御モード	ラベル 0 に”TUN”	モードに実効モード
	温度測定値	ラベル 1 に”Temp”	数値 1
	P パラメータ	ラベル 2 に”P”	数値 2
	I パラメータ	ラベル 2 に”I”	数値 2
	D パラメータ	ラベル 2 に”D”	数値 2
	制御操作量	ラベル 3 に”MV”	数値 3
停止要求時	制御モード	ラベル 0 に”STP”	モードに実効モード
	温度測定値	ラベル 1 に”Temp”	数値 1
	制御目標値	ラベル 2 に”SP”	数値 2
	制御操作量	ラベル 3 に”MV”	数値 3

各フィールドの表示内容

生成した表示イメージは、属性のなかに保存し、シミュレーション時に表示するほか、Web ブラウザにも送れるようにしておきます。

オブジェクト定義

クラス：display_image

属性：lcd（液晶表示器オブジェクト）、lines（表示イメージを保持する文字列）、display_fifo（ブラウザへの表示更新要求を送る FIFO）

操作：show（表示イメージを生成し、液晶表示器/ブラウザに送る）

ヘッダーファイル

display.h では個々の表示メージの場所と、表示する文字列を定義しています。

名前表示フィールドの開始位置は DISP_xx_NAME_FIELD、値表示フィールドの開始位置は、DISP_xx_FIELD というマクロ名を付けています。

display.h

```
/* display.h
   初版: 24, December, 2014 Chuji
   最新版: 20, Jan 2019 - LCD 定義ファイルから分離
*/

/* LCD 表示イメージの生成に関する情報 */

#ifndef __DISPLAY

/* 表示フィールドの定義 --- 2 行 x 16 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 | #1 | #2 | #3 |  |target mode| "Temp" |"SP"|"MV"|
      +----+----+----+----+  +----------+--------+----+----+
行 #1 | #4 | #5 | #6 | #7 |  |    mode  |   PV   | SP | MV |
      +----+----+----+----+  +----------+--------+----+----+
*/


/* フィールドの定義 */
#define LCD16  /* 16 文字/行を使うという宣言 */

#define DISP_STRING_LEN  16
#define DISP_ROW_LEN  2

#define  DISP_NAME_ROW  0
#define  DISP_VALUE_ROW  1

#define DISP_LEFT_MOST   0
#define DISP_RIGHT_MOST  DISP_STRING_LEN-1

#define DISP_TARGET_MODE_FIELD  0
#define DISP_PV_NAME_FIELD  4
#define DISP_SP_NAME_FIELD  8
#define DISP_MV_NAME_FIELD  12
#define DISP_MODE_FIELD  DISP_TARGET_MODE_FIELD
#define DISP_PV_FIELD  DISP_PV_NAME_FIELD-1
#define DISP_SP_FIELD  DISP_SP_NAME_FIELD
#define DISP_MV_FIELD  DISP_MV_NAME_FIELD

#define DISP_PRO_NAME_FIELD  DISP_SP_NAME_FIELD
#define DISP_INT_NAME_FIELD  DISP_SP_NAME_FIELD
#define DISP_DIF_NAME_FIELD  DISP_SP_NAME_FIELD
#define DISP_PRO_FIELD  DISP_SP_FIELD
```

```
#define DISP_INT_FIELD  DISP_SP_FIELD
#define DISP_DIF_FIELD  DISP_SP_FIELD

/* 文字列の定義 */

#define  DISP_FIELD_LEN  4
#define DISP_TEMP_FIELD_LEN  5
#define DISP_MARKER_FIELD_LEN  2

#define  DISP_OUT_OF_SERVICE  "O/S "
#define  DISP_MANUAL  "MAN "
#define DISP_AUTO  "AUT "
#define DISP_TUNING  "TUN "
#define DISP_SHUTDOWN  "STP "

#define DISP_TEMPERATURE  "Temp"
#define  DISP_SP    "  SP"
#define DISP_MV    "  MV"
#define DISP_NULL  "    "
#define DISP_SET_SP  " @SP"
#define DISP_SET_MV  " @MV"


#define  DISP_SPACE  "  "
#define DISP_ACTIVE  " @"

#define  DISP_PRO  "  @P"
#define DISP_INT  "  @I"
#define DISP_DIF  "  @D"

/* シミュレーション画面用 */
#define DISP_ALL_SPACES    "                "
#define DISP_EXIT_MESSAGE  " Controller off "
#define DISP_SHUTDOWN_MESSAGE  "System shut down"
#define DISP_GOODBYE    DISP_ALL_SPACES

#define DISP_SEPARATOR    "+----------------+"

#define __DISPLAY

#endif
```

プログラムファイル

display.py のコードは簡単で、表示要求をほとんどそのまま、液晶表示器ドライバーに渡しています。同時に各行の表示イメージを保持し、シミュレーション時には表示イメージをスクリーンに表示するようにしています。また、ブラウザから操作するとき（REMOTE_OP）は、表示イメージをブラウザに送ります。

このコードで重要なのは、後半にまとめた多数のマクロ定義です。この定義により、上位モジュールからは、「何行目のどこに」表示するのではなく、「どのフィールドに」表示するかを指定できます。

display.py

```
/* display.py  LCD 表示イメージの構築
   初版： 24 December 2014 Chuji
   最新版： 23 January 2019

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:
  show:   表示イメージを生成し LCD に送る（2 行モデル）

*/

#include "./include/display.h"
#include "./include/my_round.h"

#include "ftos.py"
#include "lcd.py"

#ifdef REMOTE_OP
#include "redis_for_python.py"
#endif

/* 表示イメージ生成オブジェクト */

class display_image:
  def __init__(self):
    self.lcd = lcd_device()
    self.lines = [DISP_ALL_SPACES, DISP_ALL_SPACES]

#ifdef REMOTE_OP
/* リモート操作用の FIFO を生成 */
    self.display_fifo = open_FIFO()
#endif


/* 文字列を表示する操作
       show(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: 表示する文字列
            len: 表示する文字数 */

  def show(self, row, pos, s, len):
    if pos + len > DISP_STRING_LEN:
      len = DISP_STRING_LEN - pos

    self.lcd.put_string(row, pos, s, len)

    self.lines[row] = self.lines[row][:pos] +
s[0:len] + self.lines[row][pos + len:]

#ifdef REMOTE_OP
    self.display_fifo.rpush(Redis_lines[row],
self.lines[row])
#endif

#ifndef BCM2835
    print DISP_SEPARATOR
    print "|"+self.lines[DISP_NAME_ROW]+"|"
    print DISP_SEPARATOR
    print "|"+self.lines[DISP_VALUE_ROW]+"|"
    print DISP_SEPARATOR
    print

#endif


/* 特定の表示をするマクロ（定義は display.h 参照） */

#define SHOW_TARGET(y) show(DISP_NAME_ROW,
DISP_TARGET_MODE_FIELD, self.modes[y],
DISP_FIELD_LEN)
#define SHOW_PV_NAME() show(DISP_NAME_ROW,
DISP_PV_NAME_FIELD, DISP_TEMPERATURE,
DISP_FIELD_LEN)
#define SHOW_SP_NAME() show(DISP_NAME_ROW,
DISP_SP_NAME_FIELD, DISP_SP, DISP_FIELD_LEN)
#define SHOW_MV_NAME() show(DISP_NAME_ROW,
DISP_MV_NAME_FIELD, DISP_MV, DISP_FIELD_LEN)
#define SHOW_SET_SP() show(DISP_NAME_ROW,
DISP_SP_NAME_FIELD, DISP_SET_SP, DISP_FIELD_LEN)
#define SHOW_SET_MV() show(DISP_NAME_ROW,
DISP_MV_NAME_FIELD, DISP_SET_MV, DISP_FIELD_LEN)
#define SHOW_PRO_NAME() show(DISP_NAME_ROW,
DISP_PRO_NAME_FIELD, DISP_PRO, DISP_FIELD_LEN)
#define SHOW_INT_NAME() show(DISP_NAME_ROW,
DISP_INT_NAME_FIELD, DISP_INT, DISP_FIELD_LEN)
#define SHOW_DIF_NAME() show(DISP_NAME_ROW,
DISP_DIF_NAME_FIELD, DISP_DIF, DISP_FIELD_LEN)
#define SHOW_MODE(y) show(DISP_VALUE_ROW,
DISP_MODE_FIELD, self.modes[y], DISP_FIELD_LEN)
#define SHOW_TUNING() show(DISP_VALUE_ROW,
DISP_MODE_FIELD, self.modes[PID_TUNING],
DISP_FIELD_LEN)
```

```
#define SHOW_PV(x) show(DISP_VALUE_ROW,
DISP_PV_FIELD, ftos(x,FORMAT_199_9),
DISP_TEMP_FIELD_LEN)
#define SHOW_SP(x) show(DISP_VALUE_ROW,
DISP_SP_FIELD, ftos(x, FORMAT_999), DISP_FIELD_LEN)
#define SHOW_MV(x) show(DISP_VALUE_ROW,
DISP_MV_FIELD, ftos(x, FORMAT_999), DISP_FIELD_LEN)
#define SHOW_PRO(x) show(DISP_VALUE_ROW,
DISP_PRO_FIELD, ftos(x, FORMAT_999),
DISP_FIELD_LEN)
#define SHOW_INT(x) show(DISP_VALUE_ROW,
DISP_INT_FIELD, ftos(x, FORMAT_999),
DISP_FIELD_LEN)
#define SHOW_DIF(x) show(DISP_VALUE_ROW,
DISP_DIF_FIELD, ftos(x, FORMAT_999),
DISP_FIELD_LEN)
#define SHOW_EXIT()  show(DISP_NAME_ROW,
DISP_LEFT_MOST, DISP_EXIT_MESSAGE, DISP_STRING_LEN)
#define SHOW_SHUTDOWN()  show(DISP_NAME_ROW,
DISP_LEFT_MOST, DISP_SHUTDOWN_MESSAGE,
DISP_STRING_LEN)
#define SHOW_GOODBYE()  show(DISP_VALUE_ROW,
DISP_LEFT_MOST, DISP_GOODBYE, DISP_STRING_LEN)
```

オンオフ制御

いちばん単純な温度制御として、オンオフ制御があります。たとえば測定温度が設定温度より 1℃高くなるとヒーターを切り、1℃低くなるとヒーターを入れます。反応に遅れがあるので、温度は±1℃より広い幅で変動します。チューニングのときに P パラメータを無限大にしたのと同じ動作です。

ひとむかし前の空調（今でも使われています）では、熱膨張率の異なる 2 種類の金属を貼り合わせたバネ（バイメタル）でできたスイッチひとつで、送風をオンオフしていました。それでも「室温が一定しない」という苦情はなかったそうです。先人の知恵は素晴らしいと思います。

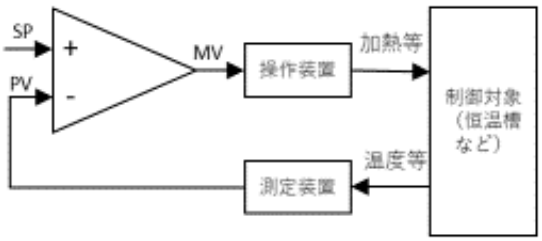
4.9　制御演算（PID 制御）

今度は、実際に温度を制御するための PID アルゴリズムの実装です。この制御は、長く工業分野で使われてきた実績があります。温度や液位、流量といった比較的単純な対象を制御するとき、とても強力なツールになります。

この自動制御は、3 つの基本的な変数を使います。慣例に従って、プロセス値（PV）、設定値あるいは目標値（SP）、操作量（MV）と呼ばれます。制御の目的は次のようなものです。『操作量を変えていくことで、制御対象のプロセス値を、できるだけ設定値に近づけるようにすること』。ここでは「できるだけ」という点が重要です。完全に一致させることが難しい場合でも、なるべく早く、なるべく長い間、許容範囲内に収めることが求められます。

PID 制御のあらまし

一番簡単な制御方法は、プロセス値と設定値の差（偏差と呼ばれています）に比例した操作量を作り出すことで、比例制御（P 制御）と呼ばれています。アナログ回路で使うオペアンプ（演算増幅器）は、+入力と - 入力の差を増幅して出力します。それをフィードバックすることで結果的に両方の入力が等しくなるようにするので、比例制御と同じことをしていることが分かります。フィードバックしている部分が制御対象というわけです。



PID 制御の働き

ところが、実際の制御対象はそれほど単純ではありません。温度コントローラの場合、電気ヒーターの加熱量を変えて操作するのですが、その熱はヒーター→ヒーター周りの空気→食品周りの空気→食材と伝わっていきます。熱が伝わる仕組みは、ヒーターと空気の間では熱伝導と輻射、空気内部は対流、空気と食材は熱伝導で、時定数の異なる遅れが重なった複雑なものです。そのため比例係数（利得）の小さい比例制御では加熱に時間がかかり、大きいと温度が不安定になってしまいます。この不安定化を防ぐのに、偏差を積分（デジタル演算では、制御周期

を掛けてから積算）した項（積分項）を操作量に加えることがあります。比例 - 積分制御（PI 制御）と呼ばれます。

PI 制御で、長い時間掛けて積算すれば、最終的には安定した状態に落ち着きます。しかし、安定するのに時間がかかったり、行き過ぎてしまったりという問題が現れることがあります。そこでプロセス値が急に変化したら操作量を変えてやれば安定になります。そのため、偏差の変化率（差分値を制御周期で割る）に比例した項（微分項）を操作量に加えてやります。これが PID 制御です。けっきょく操作量は次の式のようになります。

操作量 = 利得 × （偏差 + 偏差の積分値/積分時間 + 偏差の変化率 × 微分時間）

ここで利得、積分時間、微分時間をそれぞれ P パラメータ、I パラメータ、D パラメータといい、制御対象に合わせて最適化することで、応答が早くて安定した制御を実現します。

偏差がごく小さい値（不感帯といいます）に収まっているときは、積分項だけを操作量にすることで、さらに安定します。

制御モード

制御機能は「制御モード」によって動作を変えます。上で説明したように操作量を変えていくのは「自動制御（AUTO）モード」のときです。いきなりこのモードで動作させるのは望ましくないため、「測定値トラック（O/S）モード」と「手動操作（MANUAL）モード」を加えます。

O/S は Out of Service という意味で、測定値の表示のみを行います。操作量は安全な値（温度制御の場合加熱しない）に固定します。設定値には測定値をコピーし、常に偏差をゼロにします。温度コントローラの場合は、単なる温度表示器として動作することになります。

MANUAL モードでは、操作量を手動で変更できます。設定値には測定値をコピーし、常に偏差をゼロにします。水道の蛇口から流れ出る水量を、手で加減するイメージです。流れ出る水量は、圧力によって変化していしまいます。

電源投入時は O/S モードで、温度コントローラは温度計として機能します。つぎに MANUAL モードにして、操作量を変えて温度が変わるか調べます。それから AUTO モードにすると、自動制御が始まります。測定値のステータスが BAD（測定値が使い物にならない）のときは、前回の測定値を使いますが、BAD が続くと AUTO モードから MANUAL モードに変わります。このように意図と異なる動作を区別するため、コマンドとして与えるモードを「ターゲットモード」、実際の制御計算に使うモードを「実効モード」と呼ぶことにします。

液晶表示器では表示幅の都合から、モードを三文字の略号"O/S"、"MAN"、"AUT"と表示させます。

また、PID パラメータを変えたとき、操作量が急変（バンプと呼ばれます）することを防ぐため、積分項と微分項を初期化する操作を行います。

オブジェクト定義

クラス：pid_block

属性：mode, target, pv, sp, mv, p, i, d, dead_band, integral, last_error（上の説明に出てきた）、bad_count, bad_flag（PV のステータスを判定する）、output（mv とステータス）

操作：execute（PID 制御アルゴリズムの実行）、set_target（制御モードの目標を設定する）、inc_sp, dec_sp, inc_mv, dec_mv（設定値、操作量の増減）、tune（PID パラメータの設定）、bumpless（バンプを防ぐ）

ヘッダーファイル

各種変数・パラメータの初期値、上下限、ステップ値、ステータスが連続して BAD になる限界回数を定義しています。制御周期にデフォルト値は 1 秒です。制御周期と制御パラメータの初期値は、外部で #define することで変更できます。

pid.h

```
/* PID 制御ブロックの定義
   初版：2014/12/26 Chuji
   最新版：2017/05/05 to modify P パラメータの範囲を変更
*/

#ifndef __PID

/* PID 動作状態の定義 */

#define PID_OUT_OF_SERVICE  0
#define PID_MANUAL  1
#define PID_AUTO  2

#define PID_TUNING  3  /* for target mode display
only */
#define PID_NULL_TARGET  4  /* for target mode
display only */
#define PID_SHUTDOWN  5  /* for target mode display
only */

/* パラメータの初期値 */

#define PID_PV_INIT  0.0
#define PID_MV_INIT  0.0
```

```
#define PID_SP_INIT  0.0

#ifndef PID_P_INIT
#define PID_P_INIT  2.0
#endif

#ifndef PID_I_INIT
#define PID_I_INIT  10.0
#endif

#ifndef PID_D_INIT
#define PID_D_INIT  0.0
#endif

#define PID_P_STEP  1.0
#define PID_I_STEP  1.0
#define PID_D_STEP  1.0

#define PID_DEADBAND_INIT  0.1
#define PID_INTEGRAL_INIT  0.0
#define PID_LAST_ERROR_INIT  0.0

#ifndef PID_PERIOD
#define PID_PERIOD    1.0 /* 制御周期（秒） */
#endif

#define PID_PARM_STEP    1.0 /* パラメータの増減ステッ
プ */

/* 上下限の定義 */
/* 最高温度 */
#define PID_PV_MAX     100.0

/* PID パラメータの上下限 */
#define PID_P_MAX  800 /* 比例項は無次元 */
#define PID_P_MIN  1.0

#define PID_I_MAX  800  /* 積分項の次元は秒 */
#define PID_I_MIN  0
#define PID_P_ONLY      1    /* I がこの値以下になると
PID 制御ではなく P 制御 */

#define PID_D_MAX  200  /* 微分項の次元は秒 */
#define PID_D_MIN  0

/* 制御変数の上下限 */
#define PID_MAX_PARM  100  /* MV と SP の上下限 */
#define PID_MIN_PARM  0

#define PID_MAX_PROPORTIONAL  100  /* 積分項の上下限
*/
#define PID_MIN_PROPORTIONAL  -100

#define PID_MAX_INTEGRAL  100
#define PID_MIN_INTEGRAL  0
#define PID_NO_INTEGRAL         0

#define PID_MAX_DIFFERENTIAL  100  /* 微分項の上下限
*/
#define PID_MIN_DIFFERENTIAL  -100
#define PID_NO_DIFFERENTIAL     0

/* ステータスが BAD になるまでの回数 */
#define PID_BAD_COUNT_LIMIT  3

#define __PID

#endif
```

プログラムファイル

PID 制御の実行アルゴリズムです。アルゴリズムの説明が済んでいるので、ここでの説明は省略します。

実際の数値計算はほんの一部で、大部分のコードが異常処理に充てられていることが分かります。

pid.py

```
/* pid 制御ブロック
      初版： 2014/12/26 Chuji
      最新版：2019/1/22

改定履歴
      2019/1/22   属性を直接読み出せるようにする
      2017/3/8    P 制御を許す、MV/SP 設定時は整数にする
      2017/2/20   与えられた PV と制御出力を構造体にする

Class: pid_block
属性:
  mode:        ブロックの動作モード（O/S, MAN, AUTO）
  target:      ブロックのターゲットモード
  pv:          プロセス値
  sp:          セットポイント
  mv:          ブロックの出力（操作量）
  p:           比例ゲイン（無次元）
  i:           積分時間（秒）
  d:           微分時間（秒）
  dead_band:   偏差の不感帯
  integral:    過去からの積分値
  last_error:  前回の偏差
  bad_count:   PV のステータスが連続して BAD だった回数
  bad_flag:    BAD が連続した時の異常フラグ
  output:      制御出力（操作量とステータス）

操作:
  set_target: ターゲットモードを与える
  execute:    PID 制御ブロックを実行する
  inc_sp:     SP を増加させる
  dec_sp:     SP を減少させる
  inc_mv:     MV を増加させる
  dec_mv:     MV を減少させる
  tune:       新しいチューニングパラメータを与える
  bumpless:   不必要な制御のバンプを防ぐ

*/

#include "./include/process_value.h"
#include "./include/pid.h"
#include "./include/my_round.h"

class pid_block:
  def __init__(self):
    self.target = PID_OUT_OF_SERVICE
    self.mode = PID_OUT_OF_SERVICE
    self.pv = PID_PV_INIT
    self.sp = PID_SP_INIT
    self.mv = PID_MV_INIT
    self.p = PID_P_INIT  /* 比例ゲイン */
    self.i = PID_I_INIT  /* 積分時間 */
    self.d = PID_D_INIT  /* 微分時間 */
    self.deadband = PID_DEADBAND_INIT
    self.integral = PID_INTEGRAL_INIT
    self.last_error = PID_LAST_ERROR_INIT
    self.bad_count = 0
    self.bad_flag = STATUS_BAD
    self.output = process_value()
    self.output.value = self.mv
    self.output.status = self.bad_flag

  def bumpless(self):  /* 積分項と微分項を初期化しバンプ
を防ぐ */
    self.integral = PID_INTEGRAL_INIT
    self.last_error = PID_LAST_ERROR_INIT

  def set_target(self, targetmode):
    self.target = targetmode

  def tune(self, pgain, i_time, d_time):
    self.p = pgain
    self.i = i_time
    self.d = d_time
    self.bumpless()

  def inc_sp(self):
    if self.mode == PID_AUTO:
```

```
      self.sp = MY_ROUND(self.sp) + PID_PARM_STEP
      if self.sp > PID_MAX_PARM:
        self.sp = PID_MAX_PARM

  def dec_sp(self):
    if self.mode == PID_AUTO:
      self.sp = MY_ROUND(self.sp) - PID_PARM_STEP
      if self.sp < PID_MIN_PARM:
        self.sp = PID_MIN_PARM

  def inc_mv(self):
    if self.mode == PID_MANUAL:
      self.mv = MY_ROUND(self.mv) + PID_PARM_STEP
      if self.mv > PID_MAX_PARM:
        self.mv = PID_MAX_PARM

  def dec_mv(self):
    if self.mode == PID_MANUAL:
      self.mv = MY_ROUND(self.mv) - PID_PARM_STEP
      if self.mv < PID_MIN_PARM:
        self.mv = PID_MIN_PARM

/*  PID 制御ブロックの実行
     execute(pv: ステータス付きプロセス値)
     出力 <- ステータス付き操作量 */

  def execute(self, pv):
    if pv.status == STATUS_GOOD:
      self.pv = pv.value
      self.bad_count = 0
      self.bad_flag = STATUS_GOOD
      self.mode = self.target
    else:
      self.bad_count += 1
      if self.bad_count > PID_BAD_COUNT_LIMIT:
        if self.mode == PID_AUTO:
          self.target = PID_MANUAL
          self.mode = PID_MANUAL
        self.bad_flag = STATUS_BAD

    if self.mode == PID_OUT_OF_SERVICE:
      self.sp = self.pv
      self.mv = PID_MV_INIT
      self.bumpless()
      self.output.value = self.mv
      self.output.status = self.bad_flag
      return self.output

    elif self.mode == PID_MANUAL:
      self.sp = self.pv
      self.integral = self.mv
      self.last_error = PID_LAST_ERROR_INIT
      self.output.value = self.mv
      self.output.status = self.bad_flag
      return self.output

    elif self.mode == PID_AUTO:
      pterm = self.p * (self.sp - self.pv)
      if abs(pterm) > (self.p * self.deadband):
        if self.i < PID_P_ONLY:  /* P control
without I */
          self.integral = PID_NO_INTEGRAL
        elif (pterm > PID_MAX_PROPORTIONAL) |
(pterm < PID_MIN_PROPORTIONAL):
          self.integral = PID_INTEGRAL_INIT
        else:
          self.integral += PID_PERIOD * pterm /
self.i
          if self.integral > PID_MAX_INTEGRAL:
            self.integral = PID_MAX_INTEGRAL
          elif self.integral < PID_MIN_INTEGRAL:
            self.integral = PID_MIN_INTEGRAL
        if self.i < PID_P_ONLY:
          dterm = PID_NO_DIFFERENTIAL
        else:
                dterm = self.d * (pterm -
self.last_error) / PID_PERIOD

        result = pterm + self.integral + dterm
      else:
        result = self.integral

      self.last_error = pterm

      if result > PID_MAX_PARM:
        result = PID_MAX_PARM
      elif result < PID_MIN_PARM:
        result = PID_MIN_PARM

      self.mv = result

      self.output.value = self.mv
      self.output.status = self.bad_flag

      return self.output
```

4.10　HMI

HMI（ヒューマン・マシン・インターフェース）は人間の操作を解釈して、必要な処理を実行する機能です。温度コントローラの場合は、3 つのキースイッチ操作に応じて、液晶表示器の表示を更新するとともに、温度制御に反映します。

スイッチを操作する順番で動作を変えるため、HMI 自身が自分の「状態」を記憶し、状態毎に応答を定義していくことから始めます。このような動作を「ステートマシン」あるいは「有限オートマトン」と言います。日本語で「状態遷移機械」ということもありますが、かえって仰々しくて分かりづらいですね。ある状態（始状態）にあったとき、操作入力（イベント）に対する応答（どういうときに何をするか、次にどの状態にいるか）を指定することで、動作を定義していきます。以下のような状態遷移表にして整理すると、分かりやすくなります。状態ごとに、すべての入力に対して動作と終状態を決めてやります。

始状態	入力	条件	動作	終状態
状態名	入力名	○○のとき	○○をする	状態名

状態遷移表の記載項目

4.10.1ステートマシン

では HMI のステートマシンを定義しましょう。液晶表示器には、前出した表示イメージ生成の節で説明した各フィールドへの表示内容を指定します。次の表をデフォルトとし、状態ごとの表示はこれとの違いとして記述しています。

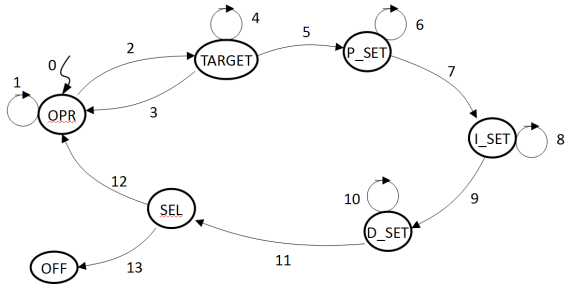
フィールド	表示内容
ラベル 0	空白
モード	実効モード
ラベル 1	" PV"
数値 1	現在の温度（℃）を 0.1℃単位で表示
ラベル 2	" SP"（AUT モードでは"@SP"と表示）
数値 2	目標温度（℃）を 1℃単位で表示
ラベル 3	" MV"（MAN モードでは"@MV"と表示）
数値 3	制御出力（%）を 1%単位で表示

まず、ステートマシンは次の表にある状態をとることにします。

状態名	状態	制御機能	表示機能
OPR	制御の操作を行う	実効モードで実行	デフォルト
TARGET	ターゲットモードを選択する	実効モードで実行	ラベル 0 にターゲットモードを表示
P_SET	P パラメータ指定する	実効モードで実行	ラベル 2 に"@P"、数値 2 に P パラメータを表示
I_SET	I パラメータ指定する	実効モードで実行	ラベル 2 に"@I"、数値 2 に I パラメータを表示
D_SET	D パラメータ指定する	実効モードで実行	ラベル 2 に"@D"、数値 2 に D パラメータを表示
SEL	シャットダウンするか指定する	実効モードで実行	ラベル 0 に"STP"を表示
OFF	停止	停止	モードに"STP"を表示

ステートマシンの状態一覧

状態とその間の遷移を下の状態遷移図に示します。図中の番号は、状態遷移表の遷移番号を示しています。遷移 0 は、電源投入あるいは温度コントローラ起動時の遷移を示します。



ステートマシンの状態遷移

最終的な状態遷移表を次ページに示します。始状態ごと、入力ごと、条件ごとに動作と終状態が定義され、漏れも重複もないことを確認してください。

上位モジュールには、キースイッチの連続押下を検出する機能があります。パラメータ値の増減には便利な機能ですが、ステートマシンの状態が変わる時などには、いったんリピート機能を解除する必要があります。これは HMI オブジェクトからの戻り値で教えるようにしています。同様に、プログラムの終了とシステムシャットダウンも戻り値に反映させて、上位モジュールに処理を委ねています。

ファジイ制御（続き）

前（32 ページ）のコラムでは、もっと役に立つルールを追加してはじめて、ファジイ制御が生きてくると言いました。では、どんなルールがありうるでしょうか？　文献には、しばしば偏差の変化速度に関するルールが載っています。でもそれでは、PID 制御とあまり代わり映えしません。もっとも、解析的に記述できないような制御対象には有効かもしれません。

もっと AI に教え甲斐のあるルールはないのでしょうか？　たとえばエアコンの場合に重要なのは、室内にいる人が快適に過ごせることと、できるだけ電力消費を抑えることでしょう。そうすると、

- 『暑い』屋外から入った直後は、冷房を『強め』に効かせるが、『ある程度』の時間が経ったら、このルールは適用しない。
- 設定温度が『かなり低め』になっているときは、『時間が経つ』につれて、『省エネ温度』に向けて、『少しずつ』設定温度を上げていく。
- 冷凍機やヒートポンプは、『なるべく』運転させない。

といったルールを追加すればいいでしょう。エアコンには、送風機の風量や風向を変える機能もあるので、

- 人がいる『付近の』温度を、『なるべく』快適に保つ（これは実際に製品化されている）。

とか

- 『赤ちゃん』が『眠って』いたら、冷たい風を『直接当てないよう』にする（まだ実現できているか知りませんが）。

といったルールがあってもいいでしょう。

もっとも、これらのルールの大半は、「あるといいな」という私の妄想でしかありません。技術者からは「そんなこと、どうやって実装するんだ？」という声が聞こえてきそうです。でも大事なのは、どうやって作るかではなく、何を実現しようとするか、という点だと思います。

遷移	始状態	入力	条件	動作	終状態
0	-	-	(動作開始)	Mode=O/S、表示を初期化	OPR(O/S)
1	OPR	▲	Mode=O/S	No action	OPR(O/S)
		▼	Mode=O/S	No action	OPR(O/S)
		▲	Mode=MAN	MV をインクリメント	OPR(MAN)
		▼	Mode=MAN	MV をデクリメント	OPR(MAN)
		▲	Mode=AUT	SP をインクリメント	OPR(AUT)
		▼	Mode=AUT	SP をデクリメント	OPR(AUT)
2	OPR	Mode		実効モードのコピーを、ターゲットモードとしてラベル 0 に表示。リピート解除	TARGET
3	TARGET	Mode		ラベル 0 をクリア。ターゲットモードを PID に与え、実効モードを決めさせる。リピート解除	OPR
4	TARGET	▲		ターゲットモードを O/S→MAN→AUT→O/S に順次変更。リピート解除	TARGET
5	TARGET	▼		ラベル 0 をクリア。ラベル 2 に"@P"、数値 2 に P パラメータを表示。リピート解除	P_SET
6	P_SET	▲		P パラメータをインクリメント	P_SET
		▼		P パラメータをデクリメント	P_SET
7	P_SET	Mode		ラベル 2 に"@I"、数値 2 に I パラメータを表示。リピート解除	I_SET
8	I_SET	▲		I パラメータをインクリメント	I_SET
		▼		I パラメータをデクリメント	I_SET
9	I_SET	Mode		ラベル 2 に"@D"、数値 2 に D パラメータを表示。リピート解除	D_SET
10	D_SET	▲		D パラメータをインクリメント	D_SET
		▼		D パラメータをデクリメント	D_SET
11	D_SET	Mode		PID パラメータを PID 制御に与える。ラベル 2 に" SP"、数値 2 に SP を表示。リピート解除	SEL
12	SEL	Mode		ラベル 0 をクリア。ターゲットモードを PID に与え、実効モードを決めさせる。リピート解除	OPR
13	SEL	▲		コントローラプログラムの停止（制御出力継続）	OFF
		▼		コントローラのシャットダウン（制御出力 0%）	OFF

HMI ステートマシンの状態遷移表

4.10.2実装

オブジェクト定義

クラス：human_machine_interface

属性：target（ターゲットモード）、parmP, parmI, parmD（PID パラメータ）、modes（モードと表示文字列をペアにした辞書）、temp, control, pid, lcd, indicator（下位オブジェクト）、my_socket, last_execution（WiFi ブロードキャスト用）

操作：state_machine（HMI ステートマシンの実行）、block_execution（PID 制御の実行）

ヘッダーファイル

HMI の状態名定義が大部分で、あまり大きなファイルではありません。

hmi.h

```
/* ヒューマン・マシン・インターフェース */
/* 2014/12/26 Chuji */

#ifndef __HMI

#include "gpio.h"

/* HMI の状態名定義 */

#define HMI_OPERATION  1
#define HMI_MODE_SELECT  2
#define HMI_P_SET  3
#define HMI_I_SET  4
#define HMI_D_SET  5
#define HMI_SHUTDOWN  6
#define HMI_EXIT  7
#define HMI_CONTINUE  8
#define HMI_KEY_RELEASE  9

/* HMI で使うキースイッチ名定義 */
#define HMI_MODE_SW  GPIO_MODE_SW
#define HMI_UP_SW  GPIO_UP_SW
#define HMI_DOWN_SW  GPIO_DOWN_SW

/* ステータス表示灯 */
#define HMI_ALARM       GPIO_HIGH
#define HMI_NO_ALARM    GPIO_LOW

#define __HMI
```

```
#endif
```

プログラムファイル

HMI モジュールのプログラムを説明します。

ステートマシン（state_machine）のプログラムは非常に長いのですが、状態遷移表を忠実にコード化しているだけです。特に難しいところはないと思います。

これだけ長いのは、コンパクトなプログラミングというポリシーに反するのですが、始状態ごとに見れば、30 行（キーごとなら 10 行）程度なので、if～elif～else～という構造さえ見失わなければ、プログラムの見通しはそう悪くないと思います。なお、コード記述の都合で、キーに対する if 文の順番が状態遷移表と異なっているところがあります。

コーディングには、構造化を意識したアプローチをとりました。まず、三つの def 部の冒頭を書き出し、状態遷移表の「始状態」ごとの処理を作る枠組みを用意します。

```
class human_machine_interface:
  def __init__(self):

  def state_machine(self, key):
    if self.state == HMI_OPERATION:
    elif self.state == HMI_MODE_SELECT:
    elif self.state == HMI_P_SET:
    elif self.state == HMI_I_SET:
    elif self.state == HMI_D_SET:
    elif self.state == HMI_SHUTDOWN:

  def block_execution (self):
```

次に、キー「入力」ごとの処理の枠組み（赤文字）を作り、「始状態」ごとにコピーします（青文字）。

```
  def state_machine(self, key):
    if self.state == HMI_OPERATION:
      if key == HMI_UP_SW:
      elif key == HMI_DOWN_SW:
      elif key == HMI_MODE_SW:
    elif self.state == HMI_MODE_SELECT:
      if key == HMI_UP_SW:
      elif key == HMI_DOWN_SW:
      elif key == HMI_MODE_SW:
    elif self.state == HMI_P_SET:
 :
```

OPR ステートだけは、「条件」として現在のモードによるケース分けの枠組みを作ります。

```
  def state_machine(self, key):
    if self.state == HMI_OPERATION:
     self.target = self.pid.mode
     if key == HMI_UP_SW:
        if self.target == PID_AUTO:
        elif self.target == PID_MANUAL:
      elif key ==  HMI_DOWN_SW:
        if self.target == PID_AUTO:
        elif self.target == PID_MANUAL:
      elif key == HMI_MODE_SW:
        if self.target == PID_AUTO:
        elif self.target == PID_MANUAL:
    elif self.state == HMI_MODE_SELECT:
 :
```

あとは状態遷移図の「動作」と「終状態」を書き込んでいけば、コーディングは終了です。

PID 制御の実行部（block_execution）では、まず温度を読み取って PID 制御に渡し、その出力を PWM に渡します。PID 演算結果のステータスが BAD のときはアラームを点灯し、PWM 出力を変えません。最後にモード、PV、SP、MV を液晶表示器に表示させます。

レコーダー機能（ローカル、ネットワークの一方あるいは両方）が指定されているときは、トレンド作成に使うすべてのパラメータをプリントし、または文字列 msg に変換してネットワークに送信します。ローカルレコーダーでは、標準出力への print をコンソールに表示するか、リダイレクト*>file_name* を使ってファイルに収納します。

hmi.py

```
/* ヒューマン・マシン・インターフェース
      初版： 2014/12/27 Chuji
      最新版： 2019/1/27

改定履歴
      2019/1/27 表示イメージ生成オブジェクトの採用
      2017/5/5 P パラメータの値域と増減ステップを変更
      2017/3/3 PID データをネットワークに広報
      2016/12/3 過大温度警報追加、温度センサと PWM の複数利用
      2018/9/4 PID ブロックの実行間隔をモニター

Class: human_machine_interface - HIM ステートマシン
属性:
  state:    HMI 操作モードを表す状態
  target:   PID ブロックのターゲットモード設定用
  parmP:    PID ブロックの P パラメータ設定用
  parmI:    PID ブロックの I パラメータ設定用
  parmD:    PID ブロックの D パラメータ設定用
  modes:    操作モードを表示する文字列の辞書

  temp:     温度センサオブジェクト
  control:  操作出力用 PWM オブジェクト
  pid:      PID 制御ブロックオブジェクト
  lcd:      表示機能オブジェクト（LCD デバイスではない）
  indicator: アラーム表示 LED オブジェクト

  my_socket:  ネットワーク記録計にデータを広報するソケット
  start:  PID ブロックを起動したシステム時刻

操作：
  state_machine:    HMI ステートマシン実行
  block_execution: PID 制御ブロックの実行

*/

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

#include "thermo.py"
#include "pwm.py"
#include "pid.py"
#include "display.py"

#ifdef NETWORK_RECORDER
#include "wifi.py"
```

```
#endif


class human_machine_interface:
  def __init__(self):
    self.state = HMI_OPERATION
    self.target = PID_OUT_OF_SERVICE
    self.parmP = PID_P_INIT
    self.parmI = PID_I_INIT
    self.parmD = PID_D_INIT

    /* 表示文字列辞書の生成*/
    self.modes =
{PID_OUT_OF_SERVICE:DISP_OUT_OF_SERVICE,
PID_MANUAL:DISP_MANUAL, PID_AUTO:DISP_AUTO,
PID_TUNING:DISP_TUNING, PID_NULL_TARGET:DISP_NULL,
PID_SHUTDOWN:DISP_SHUTDOWN}

                /* オブジェクトの生成 */
    self.temp = thermometer(TEMP_SENSOR)
    self.control = pulse_width_modulator(PWM_PORT)
    self.pid = pid_block()
    self.lcd = display_image()
    self.indicator = status_indicator(STATUS_PORT)

    /* 表示イメージの生成 */
    self.lcd.SHOW_TARGET(PID_NULL_TARGET)
    self.lcd.SHOW_PV_NAME()
    self.lcd.SHOW_SP_NAME()
    self.lcd.SHOW_MV_NAME()
    self.lcd.SHOW_MODE(self.pid.mode)
    self.lcd.SHOW_PV(self.pid.pv)
    self.lcd.SHOW_SP(self.pid.sp)
    self.lcd.SHOW_MV(self.pid.mv)

#ifdef NETWORK_RECORDER
    self.mysocket = wifi_socket()
#endif

    self.start = time()

  /* キー操作で実行される HMI ステートマシン */
  def state_machine(self, key):
    if self.state == HMI_OPERATION:
      self.target = self.pid.mode

      if key == HMI_UP_SW:
        if self.target == PID_AUTO:
          self.pid.inc_sp()
          self.lcd.SHOW_SP(self.pid.sp)
        elif self.target == PID_MANUAL:
          self.pid.inc_mv()
          self.lcd.SHOW_MV(self.pid.mv)
        return(HMI_CONTINUE)

      elif key == HMI_DOWN_SW:
        if self.target == PID_AUTO:
          self.pid.dec_sp()
          self.lcd.SHOW_SP(self.pid.sp)
        elif self.target == PID_MANUAL:
          self.pid.dec_mv()
          self.lcd.SHOW_MV(self.pid.mv)
        return(HMI_CONTINUE)

      elif key == HMI_MODE_SW:
        self.lcd.SHOW_TARGET(self.target)

        if self.target == PID_AUTO:
          self.lcd.SHOW_SP_NAME()
        elif self.target == PID_MANUAL:
          self.lcd.SHOW_MV_NAME()
        self.state = HMI_MODE_SELECT
        return(HMI_KEY_RELEASE)

          /* OPERATION ステートの処理はここまで */


    elif self.state == HMI_MODE_SELECT:
      if key == HMI_UP_SW:
        if self.target == PID_OUT_OF_SERVICE:
          self.target = PID_MANUAL
        elif self.target == PID_MANUAL:
          self.target = PID_AUTO
        elif self.target == PID_AUTO:
          self.target = PID_OUT_OF_SERVICE
        self.lcd.SHOW_TARGET(self.target)
        return(HMI_KEY_RELEASE)

      elif key == HMI_MODE_SW:
        self.pid.set_target(self.target)
        self.lcd.SHOW_TARGET(PID_NULL_TARGET)
        if self.target == PID_AUTO:
          self.lcd.SHOW_SET_SP()
        elif self.target == PID_MANUAL:
          self.lcd.SHOW_SET_MV()
        self.state = HMI_OPERATION
        return(HMI_KEY_RELEASE)

      elif key == HMI_DOWN_SW:
        self.pid.set_target(self.target)
        self.lcd.SHOW_TARGET(PID_TUNING)
        self.lcd.SHOW_PRO_NAME()
        self.parmP = self.pid.p
        self.lcd.SHOW_PRO(self.parmP)
        self.state = HMI_P_SET
        return(HMI_KEY_RELEASE)

    /* MODE  SELECT ステートの処理はここまで */


    elif self.state == HMI_P_SET:
      if key == HMI_UP_SW:
        self.parmP += PID_P_STEP
        if self.parmP > PID_P_MAX:
          self.parmP = PID_P_MAX
        self.lcd.SHOW_PRO(self.parmP)
        return(HMI_CONTINUE)

      elif key == HMI_DOWN_SW:
        self.parmP -= PID_P_STEP
        if self.parmP < PID_P_MIN:
          self.parmP = PID_P_MIN
        self.lcd.SHOW_PRO(self.parmP)
        return(HMI_CONTINUE)

      elif key == HMI_MODE_SW:
        self.parmI = self.pid.i
        self.lcd.SHOW_INT_NAME()
        self.lcd.SHOW_INT(self.parmI)
        self.state = HMI_I_SET
        return(HMI_KEY_RELEASE)

    /* P パラメータ設定ステートの処理はここまで */


    elif self.state == HMI_I_SET:
      if key == HMI_UP_SW:
        self.parmI += PID_I_STEP
        if self.parmI > PID_I_MAX:
          self.parmI = PID_I_MAX
        self.lcd.SHOW_INT(self.parmI)
        return(HMI_CONTINUE)

      elif key == HMI_DOWN_SW:
        self.parmI -= PID_I_STEP
        if self.parmI < PID_I_MIN:
          self.parmI = PID_I_MIN
        self.lcd.SHOW_INT(self.parmI)
        return(HMI_CONTINUE)

      elif key == HMI_MODE_SW:
        self.parmD = self.pid.d
        self.lcd.SHOW_DIF_NAME()
        self.lcd.SHOW_DIF(self.parmD)
        self.state = HMI_D_SET
        return(HMI_KEY_RELEASE)

    /* I パラメータ設定ステートの処理はここまで */

    elif self.state == HMI_D_SET:
      if key == HMI_UP_SW:
        self.parmD += PID_D_STEP
        if self.parmD > PID_D_MAX:
          self.parmD = PID_D_MAX
        self.lcd.SHOW_DIF(self.parmD)
        return(HMI_CONTINUE)

      elif key == HMI_DOWN_SW:
        self.parmD -= PID_D_STEP
        if self.parmD < PID_D_MIN:
          self.parmD = PID_D_MIN
        self.lcd.SHOW_DIF(self.parmD)
        return(HMI_CONTINUE)
```

```
      elif key == HMI_MODE_SW:
        self.lcd.SHOW_SP_NAME()
        self.lcd.SHOW_SP(self.pid.sp)
        self.lcd.SHOW_TARGET(PID_SHUTDOWN)
        self.pid.tune(self.parmP, self.parmI,
self.parmD)
        self.state = HMI_SHUTDOWN
        return(HMI_KEY_RELEASE)

    /* D パラメータ設定ステートの処理はここまで */

    elif self.state == HMI_SHUTDOWN:
      if key == HMI_MODE_SW:
        self.lcd.SHOW_TARGET(PID_NULL_TARGET)
        if self.target == PID_AUTO:
          self.lcd.SHOW_SET_SP()
        elif self.target == PID_MANUAL:
          self.lcd.SHOW_SET_MV()
        self.state = HMI_OPERATION
        return(HMI_KEY_RELEASE)

      /* leave SHUT_DOWN state */

      elif key == HMI_UP_SW:
        self.lcd.SHOW_EXIT()
        self.lcd.SHOW_GOODBYE()
#ifndef BCM2835
        print "program exit"
#endif
        return(HMI_EXIT) /* 温度コントローラの終了 */

      elif key == HMI_DOWN_SW:
        self.lcd.SHOW_SHUTDOWN()
        self.lcd.SHOW_GOODBYE()
#ifndef BCM2835
        print "system shutdown"
#endif
        return(HMI_SHUTDOWN) /* シャットダウン */

    /* シャットダウン選択ステートの処理はここまで */

    return(HMI_KEY_RELEASE) /* any other condition
*/

/* HMI ステートマシンはここまで */



/* PID 制御ブロックの周期的実行 */

  def block_execution (self):
    executed = time()

#ifdef TEMP_SIMULATION  /* シミュレーションには過熱量が
必要なため */
    pv = self.temp.read_temp(self.pid.mv)
#else
    pv = self.temp.read_temp()
#endif
    /* 温度が過大だったら O/S モードに移行する */
    if pv.value >= PID_PV_MAX:
      self.pid.set_target(PID_OUT_OF_SERVICE)

    mv = self.pid.execute(pv)
#ifdef HW_DEBUG
    print mv.status, mv.value
#endif
    self.control.output(mv.value)

    if mv.status == STATUS_BAD:
      self.indicator.output(INDICATOR_ON)
    else:
      self.indicator.output(INDICATOR_OFF)
      self.control.output(mv.value)

    self.lcd.SHOW_MODE(self.pid.mode)
    self.lcd.SHOW_PV(pv.value)
    self.lcd.SHOW_MV(mv.value)

    if self.state != HMI_P_SET and self.state !=
HMI_I_SET and self.state != HMI_D_SET:
      self.lcd.SHOW_SP(self.pid.sp)

#ifdef LOCAL_RECORDER
```

```
    print executed - self.start, self.pid.mode,
self.pid.pv, self.pid.sp, self.pid.mv
#endif

#ifdef NETWORK_RECORDER
    msg = '%f %d %f %f %f' % (executed -
self.start, self.pid.mode, self.pid.pv,
self.pid.sp, self.pid.mv)
    self.mysocket.publish(msg)
#endif

    self.last_execution = executed

/* PID 制御ブロックの実行はここまで */
```

プログラムは簡潔に美しく

学生時代、他学部のコースを受けて、単位にできる制度がありました。そこで、FORTRAN の大家と呼ばれた有名教授の、当時普及しだした PASCAL という言語を「使ってみる」というコースを受講しました。たぶん専修生以外では、日本で最初に PASCAL を使ったグループだったと思います。

実際に指導してくれたのは、いかにも頭の切れそうな新進気鋭の助手（今でいう助教）でした。あるとき、彼の眺めていたプログラムが非常に短いのに気がついて質問したら、こう言われました。「プログラムは簡潔で美しくなければいけない」

どうもプログラムというものは、一度に目が届く範囲に収まっていないと、論理が把握しきれないし、バグも多くなるということのようです。それを「美しい」と思う感性は素晴らしいと感じました。それまでは、プログラムは絡んだスパゲッティのようなものだと理解していたので、目からうろこが落ちる思いでした。目の届く範囲というのは、プリンター用紙一枚、多くても二枚以内が望ましいそうです。

それからは、美しいとまでは言えなくても、短く簡潔なプログラム作りを心掛けるようになりました。この本は二段組なので、各ソフトウエアモジュールは 1 ページ以内に収まるのが目安です。ヘッダーファイルを別にしている効果もあり、冒頭の長いコメントがなければ、短いプログラムになっています。唯一の例外は hmi.py ですが、本文で説明したように、論理を見誤ることはありません。

ちなみに、この助手は後に、コンピュータ科学の指導的立場で長く活動されました。

4.11　キー入力（割り込み処理）

このモジュールは実機での割り込みを受け付け、HMI に渡します。

オブジェクト定義

クラス：switches

属性：hmi（HMI オブジェクト）、pushed（最後に押下されたキー）、count（長押しタイマー）、int_timer（PID ブロック実行用タイマー）、exec_interval（PID 実行間隔）、click_fifo（ブラウザからのクリックの受付口）

操作：switch_isr_hander, timer_isr_handler（割り込みハンドラー）、get_remote_click（ブラウザからのクリック処理）、push, release, check_timer（シミュレーション用）

ヘッダーファイル

使用するキースイッチを GPIO で定義しています。またスイッチのチャタリングを除去するための不感時間も定義します。キースイッチの応答が思わしくないときは KEYS_BOUNCE を長くして見てください。

タイマーにより一秒間に 5 回（200ms ごとに）割り込みを発生させ、キースイッチの押下検出と PID 制御の実行タイミングを作ります。最後にスイッチの長押し検出のためのカウンタを定義します。

keys.h

```
/* キースイッチの定義 */
/* 初版：2014/12/31 Chuji
   最新版：2017/2/26 */

#ifndef __KEYS

# include "gpio.h"

/* キースイッチの論理名を GPIO で定義 */

#define KEYS_NOTHING  0
#define KEYS_MODE_SW  GPIO_MODE_SW
#define KEYS_UP_SW  GPIO_UP_SW
#define KEYS_DOWN_SW  GPIO_DOWN_SW

/* キースイッチの状態を読むマクロ定義 */

#define key_input(channel)  GPIO.input(channel)

/* キースイッチの状態を GPIO で定義 */

#define KEYS_ON  GPIO_HIGH
#define KEYS_OFF GPIO_LOW

/* チャタリング防止パラメータ */

#define KEYS_BOUNCE  100 /* in millisecond */
#define KEYS_TIMER_BOUNCE  1 /* バウンスなし */

/* 連続してキーを押下したことの検出 */

#define KEYS_TIMER_INIT    0  /* 割り込みカウンタの初期値 */
#define  KEYS_CHECK_INTERVAL  5  /* 一秒間のタイマーチェック回数 */

/* 定周期割り込みパルスの幅 */

#define KEYS_TIMER_DUTY          50

/* continued push detect parameters */

#define KEYS_INIT  0   /* 連続押下カウンタのクリア値 */
#define KEYS_FIRST  1  /* 最初の押下 */
#define  KEYS_CONTINUE  KEYS_CHECK_INTERVAL  /* 連続押下モードになる割り込み回数 */


#define __KEYS

#endif
```

プログラムファイル

キー割り込みハンドラーでは、押されたキースイッチを変数 pushed に記憶し、実際の処理はタイマー割り込みがあったときに行います。そのため、200ms 以下の早い操作は無視されることがあります。電動バルブを開閉するボタンなどで、インチングといって、ボタンをちょっとだけ押したらすぐ放す操作をしますが、これは行えません。ゆっくり操作します。

タイマー割り込みがあると、キースイッチの押下をチェックし、継続時間をカウンタで数えます。HMI にキー操作を伝えますが、長押しタイマーが所定値を超えると、タイマー割り込みごとに HMI に伝えるようにします。

同じタイマー割り込みを使って、（5 回/秒×制御周期）回ごとに PID 制御を実行します。

タイマー割り込みは、キースイッチと同じソフトウエア構造を採りたかったので、あえて外部割り込みを使っています。TXD ポートから 200ms のパルスを発生し、RXD ポートで検出しています。厳密なタイミングを必要とするときに、外部にリアルタイムカウンター・タイマーIC を置いて、そこからの割り込みを受ける設計を模擬しています。好みの問題なので、内部タイマー割り込みを使っても構いません。

switch.py

```
/* スイッチとタイマーの割り込みハンドラー
    初版： 2014/12/31 Chuji
    最新版： 2019/1/23

改定履歴
     2019/1/23  シャットダウンコマンドをマクロ定義
     2017/6/7  リモートキー操作の追加（Redis FIFO）
```

```
     2015/1/4  ハードウエアタイマーの追加

Class: switches - キースイッチとタイマーの割り込み処理
属性:
   hmi:         ヒューマン・マシン・インターフェース・オブジェクト
   pushed:      最後の押下されたキーの ID
   count:       長押しタイマー
   int_timer:   PID ブロック実行用タイマー
   exec_interval: PID ブロック実行間隔（タイマー割り込み回数）
   click_fifo: リモート操作受信用の Redis FIFO

操作:
   switch_isr_handler: キースイッチ割り込みサービス
   timer_isr_handler:  タイマー割り込みサービス
   get_remote_click: リモート操作サービス

   (以下の操作はシミュレーション専用)
   push:          スイッチ押下をシミュレートする
   release:       スイッチ解放をシミュレートする
   check_timer:   定周期割り込みをシミュレートする

*/


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

#include "./include/pid.h"
#include "./include/keys.h"

#ifdef HW_DEBUG
#include "hmi_stub.py"
#else
#include "hmi.py"
#endif

#ifdef REMOTE_OP
#include "redis_for_python.py"
#endif

/* 割り込みハンドラー */

class switches:
  def __init__(self):
    self.exec_interval = int(KEYS_CHECK_INTERVAL *
PID_PERIOD)
    self.int_timer = KEYS_TIMER_INIT
    self.pushed = KEYS_NOTHING
    self.count = KEYS_INIT
#ifndef BCM2835
    self.key_input = KEYS_OFF
#endif

    /* HMI オブジェクトの生成 */
    self.hmi = human_machine_interface()

#ifdef BCM2835
    /* 割り込み元ハードウエアの設定 */
    /* キースイッチ */
    GPIO.setup(GPIO_MODE_SW, GPIO.IN, pull_up_down =
GPIO.PUD_DOWN)
    GPIO.setup(GPIO_UP_SW, GPIO.IN, pull_up_down =
GPIO.PUD_DOWN)
    GPIO.setup(GPIO_DOWN_SW, GPIO.IN, pull_up_down =
GPIO.PUD_DOWN)

    /* タイマー */
    GPIO.setwarnings(False)
    GPIO.setup(GPIO_CLOCK_OUT, GPIO_OUT)
    self.clock = GPIO.PWM(GPIO_CLOCK_OUT,
KEYS_CHECK_INTERVAL)
    GPIO.setup(GPIO_CLOCK_IN, GPIO_IN)
    GPIO.setwarnings(True)

    /* 割り込みハンドラーの設定 */
    GPIO.add_event_detect(GPIO_MODE_SW, GPIO.BOTH,
callback = self.switch_isr_handler, bouncetime =
KEYS_BOUNCE)
    GPIO.add_event_detect(GPIO_UP_SW, GPIO.BOTH,
callback = self.switch_isr_handler, bouncetime =
KEYS_BOUNCE)
    GPIO.add_event_detect(GPIO_DOWN_SW, GPIO.BOTH,
callback = self.switch_isr_handler, bouncetime =
KEYS_BOUNCE)

    GPIO.add_event_detect(GPIO_CLOCK_IN, GPIO.RISING,
callback = self.timer_isr_handler, bouncetime =
KEYS_TIMER_BOUNCE)

    /* タイマーの起動 */
    self.clock.start(KEYS_TIMER_DUTY)
#endif

#ifdef REMOTE_OP
    self.click_fifo = open_FIFO()
#endif



/* 割り込みハンドラーの定義 */

  /* キースイッチの押下/解放 */
  def switch_isr_handler(self, channel):
#ifdef BCM2835
    if key_input(channel) == KEYS_ON: /* 押下 */
#else
    if self.key_input == KEYS_ON:
#endif
      self.pushed = channel
      self.count = KEYS_FIRST

#ifdef HW_DEBUG
      print "Switch #", channel, "is pushed."
#endif

    else: /* 解放 */
      self.pushed = KEYS_NOTHING
      self.count = KEYS_INIT
#ifdef HW_DEBUG
      print "Switch #", channel, "is released."
#endif

  /* タイマー処理 */
  def timer_isr_handler(self, channel):
    hmi_result = HMI_CONTINUE

    /* 処理中に他のキー操作割り込みがあった場合に備えて
                    押下されたキーの ID をローカルに保存する
*/
    pushed_key = self.pushed
    /* これ以後は self.pushed が変更されても大丈夫 */

    if pushed_key != KEYS_NOTHING:
      /* 最初の押下か、既に解放されているとき */
      if self.count <= KEYS_FIRST:
        hmi_result =
self.hmi.state_machine(pushed_key)
#ifdef BCM2835
      /* 解放が検出されなかった場合の備え、
         キーがまだ押下されているかチェックする */
      if key_input(pushed_key) == KEYS_OFF:
        self.pushed = KEYS_NOTHING
        self.count = KEYS_INIT
#endif
      if self.count != KEYS_INIT:
        self.count += 1
      if self.count >= KEYS_CONTINUE:
        hmi_result =
self.hmi.state_machine(pushed_key)
    else:
      self.count = KEYS_INIT

    /* PID 制御ブロックを実行するタイミングか調べる */
    self.int_timer +=1
    if self.int_timer >= self.exec_interval:
      self.int_timer = KEYS_TIMER_INIT
      self.hmi.block_execution()

    /* HMI ステートマシンのステートが変わったときはキーをクリア */
    if hmi_result == HMI_KEY_RELEASE:
      self.pushed = KEYS_NOTHING

    elif hmi_result == HMI_EXIT:
#ifdef BCM2835
      GPIO.cleanup()
```

```
      sys.exit()
#else
      print "Exit instruction"
#endif

    elif hmi_result == HMI_SHUTDOWN:
#ifdef BCM2835
      GPIO.cleanup()
      os.system(SYS_SHUTDOWN)
#else
      print "Shutdown instruction"
#endif


#ifndef BCM2835 /* シミュレーション専用 */

  def check_timer(self):
    print "--- Timer Interrupt ---"
    self.timer_isr_handler(GPIO_CLOCK_IN)

  def push(self, pushed_key):
    print "--- Key #", pushed_key, "is pushed ---"
    self.key_input = KEYS_ON
    self.switch_isr_handler(pushed_key)

  def release(self, released_key):
    print "--- Key #", released_key, "is released ---
"
    self.key_input = KEYS_OFF
    self.switch_isr_handler(released_key)

#endif

#ifdef REMOTE_OP
  def get_remote_click(self):
    /* リモート操作があるまでブロックする */
    tag, clicked_key =
self.click_fifo.blpop(REDIS_KEY)
    if clicked_key == REDIS_UP:
      self.pushed = KEYS_UP_SW
      self.count = KEYS_INIT
    elif clicked_key == REDIS_DOWN:
      self.pushed = KEYS_DOWN_SW
      self.count = KEYS_INIT
    elif clicked_key == REDIS_MODE:
      self.pushed = KEYS_MODE_SW
      self.count = KEYS_INIT
#endif
```

ガラス細工のついでに

ガスバーナーが使えないときは、観光地などのガラス工房に相談すると良いと言いましたが、ついでに浮沈子を作ったらどうでしょうか。温度センサと同じようにガラス管の一端を塞いだら、こんどはそこを十分に柔らかくして、ゆっくりと息を吹きこみます。風船のように直径 2cm くらいまで膨らませます。ガラス加工入門の定番なので、どのくらい膨らませたら良いか、工房の人が教えてくれると思います。十分冷えてから軸を短く切ります。少しだけ水を入れたら、口を下にして、水を満たしたペットボトルにそっと入れます。ちょうど浮くように水量を調整してください。蓋をしてからペットボトルを握りしめると、内部の圧力が上がって浮沈子が沈んでいきます。手を離せば浮き上がります。水に色をつけて楽しんだり、気圧計代わりにしたりするのも面白いですよ。

4.12　温度コントローラ（全体）

全てのソフトウエアモジュールをとりこんで、Raspberry Pi を温度コントローラとして動作させるためのモジュールです。

BCM2835 など、必要な定義を#define します。これは cpp の-D オプションで指定しても構いません。最初の版では、ブラウザからの操作を含めていないので、REMOTE_OP は#define していません。

最初に GPIO のクリーンアップをしておきます。これは、その前に GPIO を使ったプログラムを実行し、キーボード割り込みなどで強制終了したとき、GPIO を使いっぱなしになっていることがあるからです。そのまま GPIO 命令を実行すると、「そのポートは既に使われている」という警告が発生してしまいます。それを防ぐためのもの処理です。

割り込み処理 switches を生成すると、プログラムは実行状態になり、割り込みによって処理が進んでいきます。割り込みを待つ間、100ms の休眠を繰り返します。この時間は適当で構いません。なお、ブラウザ操作を受け入れるときは、ブラウザからのクリック待ちをするので、休眠は必要ありません。休眠あるいはクリック待ちを永久に繰り返すので、終了させるときはキーボードで Control-C を操作するか、他のプロセスから kill シグナルを送ります。

このモジュールは実機でのみ動作させます。

controller.py

```
/* Raspberry Pi を使った温度コントローラ

   初版：  2017/2/28  Chuji
   最新版： 2017/6/7 - リモート操作を追加
*/

#define BCM2835
#define LOCAL_RECORDER
#define NETWORK_RECORDER

/*
#define REMOTE_OP
*/

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

#define SLEEP_TIME  0.1  /* 休眠時間 */

#include "switch.py"

/* 前のプログラムで GPIO を使っていた場合に備えてクリーンアップ */

GPIO.setwarnings(False)
GPIO.cleanup()
GPIO.setwarnings(True)

RaspberryPi = switches()

try:
  while True: /* 割り込み待ちに入る手順 */
#ifdef REMOTE_OP    /* リモート操作では操作待ちでブロック
*/
```

```
    RaspberryPi.get_remote_click()
#else                   /* ローカル専用時には単なる待ち */
    sleep(SLEEP_TIME)
#endif
except KeyboardInterrupt:
  GPIO.cleanup()
  pass
```

衝撃の雑誌記事

雑誌の記事に衝撃を受けるという、滅多にない経験をしたことがあります。1978 年、イリノイ大学の大学院生、キン・マン・チュンとヘルベルト・ユエンが BYTE 誌（9 月号～11 月号）に連載した、PASCAL 言語のコンパイラを開発したという記事です。IBM PC の発表（1981）に先立つこと 3 年のことです。もっとも、私が最初に見たのは数年後に出た日本語の紹介記事でしたが。

CPU は 8080、メモリは 36K バイト（メガバイトでもギガバイトでもなく）、走るのは BASIC インタープリタだけという初期のコンピュータですよ。一度に機械語にせず、いったん仮想 CPU の機械語である P コード（P は「疑似」という意味の pseudo からきている）を生成してから、8080 の機械語に翻訳しています。メモリのせいで BASIC プログラムの大きさに制限があったのも事実ですが、中間言語を経由するというのは、応用性の高いアイディアでした。この応用として（中間言語を生成したりはしないが）、CPU ごとに翻訳部だけを作ることで、多種類の組込み用小型 CPU 向けの C コンパイラが、今でも作られています。

PASCAL という言語への思い入れ（51 ページ）もあり、じっくり読みこんで、自分でも移植を試みたものです。BASIC 言語でサブルーチンの再帰呼び出し（サブルーチンのなかから、それ自身を呼び出すこと）という離れ業を演じたのも驚きですが、BASIC だけで最終的に 8080 の機械語に翻訳することができるとは思ってもいませんでした。

実際に処理できたのは機能を制限した PASCAL のサブセットでしたが、これでコンパイラを書き直し、それでフルセットのコンパイラを処理するというブートストラップ法にも言及しています。力任せの処理と構想力を兼ね備えた記事を、学生が書くような時代が来たという感慨を深くしました。

4.13　レコーダー機能（オプション）

温度コントローラを使っていると、温度の変化していく様子を把握したくなってきます。時系列データをグラフ化すると便利です。

オプションとして、ネットワークを介して、制御状態を PC など他の装置に知らせる機能を実現します。WiFi を使って、パブリッシャ・サブスクライバモデルの通信を行うことにしました。

4.13.1WiFi ブロードキャスト

温度コントローラにはグラフィック機能を組み込んでいないので、PC を使うことにします。Raspberry Pi から WiFi を介して、データを送信します。受信する PC が何台あってもいいように、ブロードキャスト（広報）させます。

これは放送局をモデルにした通信方式です。パブリッシャ（発行者）はデータをネットワークにブロードキャスト（広報）します。サブスクライバ（購読者）はそのブロードキャストを受信して、記録したり、表示したりと自分の用途に使います。この方式は、サブスクライバが何台いても Raspberry Pi の負荷は変わらないことが利点です。一方、より一般的なクライアント・サーバーモデルのように通信エラーを回復する機能がないため、ノイズが入ったり、サブスクライバが多忙だったりすると、「受信もれ」が起こるという欠点があります。

温度コントローラとしての用途では、ときどき「データのとび（欠損）」が起こっても、制御状態は把握できるので、この方式にしました。サブスクライバで表示をする際に、「とび」の影響を受けないように工夫しています。

なお、ローカルレコーダーとして、ブロードキャストと同じデータをファイルに入れることもできます。これは後で取り出して、別の PC で表示させるためのものです。

ヘッダーファイル

WiFi を使うためのヘッダーファイルです。このファイルは Python 専用ですが、Raspberry Pi と PC の両方で使えるようになっています。WiFi のインターフェース名は、Raspbian と PC で異なっている場合があります。Raspberry Pi は”wlan0”、 Ubuntu では”wlo1”だったので、切り替えることにしました。Ubuntu PC の場合に UBUNTU64 を#define することで切り替えます。PC のインターフェース名が異なっているときは、ここを変更してください。

TCP/IP を使うため、socket をインポートします。netifaces はインターフェースカードの情報を得るためのパッケージです。このファイルの主要な機能は、パブリッシャ、サブスクライバそれぞれで使う IP アドレスを求めることです。

ポート番号は他で予約されていない、49152 から 65535 の範囲から選ぶことができます。ここでは 50001 にしました。

wifi.h

```
/* WiFi 通信定義
  初版： 2017/2/20 Chuji
  改定版：2017/3/31 PC の IP アドレス取得を更新
  最新版：2017/7/3  Raspberry 向けと Linux PC 向けを共通化
*/

#ifndef __WIFI

from socket import *
import netifaces

/* 自己の IP アドレスを得る */
#ifndef UBUNTU64
#define WIFI_IF  "wlan0"
#else
#define WIFI_IF  "wlo1"
#endif

#define WIFI_BROADCAST_MASK    '255'
#define WIFI_IP_SEPARATOR      '\.'

def wifi_get_my_ip_address():
    return
netifaces.ifaddresses(WIFI_IF)[netifaces.AF_INET][0][
]['addr']

def wifi_get_broadcast_address():
  ip = wifi_get_my_ip_address()
  host_field = ip.rfind(WIFI_IP_SEPARATOR)
  bip = ip[0:host_field - 1] + WIFI_BROADCAST_MASK
  return bip

#define WIFI_BROADCAST_ADDRESS
wifi_get_broadcast_address()

/* ブロードキャスト側のアドレスは空 */
#define WIFI_BROADCAST_HOST  ''

/* プライベートポート（49152 から 65535 の間）を選択 */
#define WIFI_BROADCAST_PORT    50001

/* ソケットに指定する IP アドレスとポート */
#define WIFI_BROADCAST_ENDPOINT
(WIFI_BROADCAST_HOST, WIFI_BROADCAST_PORT)

/* パブリッシャの IP アドレスとポート */
#define WIFI_PUBLISHER_ENDPOINT
(WIFI_BROADCAST_ADDRESS, WIFI_BROADCAST_PORT)

/* 受信バッファの大きさ */
#define WIFI_BUFSIZE 8192

#define __WIFI

#endif
```

プログラムファイル

このモジュールは Raspberry Pi と PC の両方で使います。Raspberry Pi では、このモジュールを使うときに NETWORK_RECORDER が#define されていることを利用し、#define されていれば Raspberry Pi 用にパブリッシャ、されていなければ PC 用のサブスクライバになるよう設計しました。

コードはそれほど複雑ではありません。通信に使う UDP ソケットを作成し、そこにパブリッシャあるいはサブスクライバ機能を関連付けます。Raspberry Pi ではパブリッシャ用のソケットを開き、文字列 msg を送信します。PC ではサブスクライバ用のソケットを開き、受信を行います。ソケットからデータを受け取ろうとしますが、受信するまで待ちになります（プログラムがブロックします)。

wifi.py

```
/* パブリッシャ/サブスクライバモデルによるデータ広報
   初版：  2017/2/20 Chuji
   最新版： 2017/3/4
*/

/* このモジュールはパブリッシャとサブスクライバで共用できる
     NETWORK_RECORDER が#define されていればパブリッシャ
     #define されていなければサブスクライバ

Class: wifi_socket - UDP ソケットオブジェクト
属性:
  soc:       UDP ソケット
  publisher: パブリッシャ用 IP ブロードキャストアドレス)
  addr     : サブスクライバ用 IP アドレス
  msg      : サブスクライバ用受信メッセージ

操作:
  publish  : 与えられたバイト列をブロードキャストする
  subscribe: 受信したバイト列を得る
  close    : パブリッシュ・サブスクライブを停止する

*/


#include "./include/wifi.h"

class wifi_socket:
  def __init__(self):
    self.soc = socket(AF_INET, SOCK_DGRAM)

#ifdef NETWORK_RECORDER
    /* パブリッシャがブロードキャストできるようにする */

    self.soc.setsockopt(SOL_SOCKET, SO_BROADCAST,
True)
    self.publisher = (WIFI_BROADCAST_ADDRESS,
WIFI_BROADCAST_PORT)

#else    /* サブスクライバが受信できるようにする */
    self.msg = ''
    self.addr = WIFI_BROADCAST_ADDRESS
#endif
    self.soc.bind(WIFI_BROADCAST_ENDPOINT)


#ifdef NETWORK_RECORDER /* パブリッシャ */
  def publish(self, msg):
#ifdef DEBUG
    print "publsh:", msg
#endif
    self.soc.sendto(msg, self.publisher)

#else                    /* サブスクライバ */
  def subscribe(self):
    self.msg, self.addr =
self.soc.recvfrom(WIFI_BUFSIZE)
#ifdef DEBUG
    print "received:", self.msg
#endif
    return self.msg
#endif
```

```
def close(self):
  self.soc.shutdown(SHUT_RDWR)
  self.soc.close()
```

4.13.2 レコーダー実装例

グラフ表示部の実装方法はいろいろあります。例えば、

- Python のライブラリを使って描く
- Visual Basic など別の言語で描く
- Excel などの表計算ソフトのグラフ機能を使う
- Raspberry Pi を Web サーバーにして、ブラウザでグラフが見えるようにする

などが考えられます。ここでは Python にこだわって、Ubuntu PC 上でグラフ表示プログラムを作りました。

必要なパッケージ

必要なパッケージは以下の 3 つです。

- 数値計算パッケージ numpy
- プロットパッケージ matplotlib
- 正規表現パッケージ re

数値計算パッケージはデータを配列として使うため、プロットパッケージはグラフを作成するため、正規表現パッケージはサブスクライブした文字列からデータを取得するために使います。以下の手順でインストールしますが、すでに Python のパッケージ管理ツール pip をインストールしている場合は、最初のコマンドは不要です。正規表現パッケージ re は既にインストールされていると思います。

```
$ sudo apt-get install pip
$ sudo pip install numpy
$ sudo pip install matplotlib
```

ヘッダーファイル

記録プロット用のヘッダーファイルを以下に示します。これ以外に、Raspberry Pi で使ったのと同じ、wifi.h と pid.h（制御周期を知るため）を include の下に置いておきます。

recorder.h では必要なモジュールの取り込み、matplotlib を使うための定義を行っています。グラフの時間軸表示単位は分、表示範囲は制御周期の NUM_POINTS（900）倍にしています。

recorder.h

```
/* ネットワークレコーダーのため定義
    初版： 2017/3/3 Chuji
    最新版： 2019/3/9
*/

#ifndef __RECORDER

/* このプログラムの実行環境は UBUNTU 16.04 LTS 64 bit
*/

#define UBUNTU64

/* 外部ファイルの読み込み */
#include "pid.h"

/* WiFi 受信モジュールを使う場合 */
#ifndef SHOW_LOCAL_FILE
#include "../wifi.py"
#endif

#ifdef SHOW_LOCAL_FILE
#define FRACTION_OF_TIME  0.05
#else
#define FRACTION_OF_TIME  0.01
#endif

/* 使用するソフトウエアパッケージ */

#define NUMERICAL_LIBRARY numpy
#define MATPLOT_LIBRARY   matplotlib.pyplot
#define REGULAR_EXPRESSION_LIBRARY  re
#define WARNING_LIBRARY    warnings
#define IGNORE_WARNINGS   'ignore'

import NUMERICAL_LIBRARY as np
import MATPLOT_LIBRARY as plt
import REGULAR_EXPRESSION_LIBRARY

import WARNING_LIBRARY
WARNING_LIBRARY.filterwarnings(IGNORE_WARNINGS)


  /* 描画領域の全体サイズ */
#define REC_FIG_WIDTH 12
#define REC_FIG_HEIGHT 5

  /* グラフ領域の大きさ（全体に対して） */
  /* 指定方法 [左端, 下端, 幅, 高さ] */
#define REC_TREND_SIZE  [0.05, 0.05, 0.9, 0.9]

  /* Y 軸の表示範囲 */
#define MIN_Y 0
#define MAX_Y 100

  /* トレンドデータの形式 */
  /* データ： 時刻 モード PV  SP  MV  */
#define REC_DATA_FIELDS 5
#define REC_POS_TIME  0
#define REC_POS_MODE  1
#define REC_POS_PV  2
#define REC_POS_SP  3
#define REC_POS_MV  4

  /* 表示色 */
#define REC_PV_COLOR 'r'
#define REC_SP_COLOR 'g'
#define REC_MV_COLOR 'b'
#define REC_NO_COLOR 'w'

  /* 表示を開始する最低サンプル数 */
#define MIN_POINTS 5

  /* 想定するサンプル間隔（秒）*/
#ifndef MONITOR_PERIOD
#define MONITOR_PERIOD 1.0
#endif

  /* 表示する測定点数*/
#define NUM_POINTS  900
  /* 表示単位は 60 秒＝1 分 */
#define TIME_UNIT  60
  /* サンプル間隔（分） */
```

```
#define UNIT_T MONITOR_PERIOD/TIME_UNIT

#define MIN_T 0
#define MAX_T NUM_POINTS * UNIT_T

  /* 作図用マクロ関数 */
  /* 必要点数が集まったら、トレンド作図を開始 */
#define START_TREND(tr, ti, x, c)
tr.plot(ti[0:MIN_POINTS], x, c)
  /* 表示データをトレンドに与える */
#define DRAW_TREND(tr,ti, x)
tr.set_data(ti[0:len(x)], x)


#define __RECORDER

#endif
```

プログラムファイル

記録プロット用のコード recorder.py を下に示します。Raspberry Pi でも使った wifi.py を同じディレクトリに、wifi.h を./include に置いておきます。

matplotlib と numpy の詳しい解説は省略して、プログラムの流れだけ説明します。最初に描画領域を指定し、縦横軸のスケールを設定します。t, pv, sp, mv の配列を用意したら、ファイル（#ifdef SHOW_LOCAL_FILE）または WiFi（#else）から一行分のデータ（時刻、モード、PV、SP、MV の順でプリントされた文字列）を読み込みます。正規表現パッケージを利用して、文字列から各変数の数値に変換し、それぞれの配列に追加します。データの数が MIN_POINTS 個になったら、グラフの最初の部分を表示します。新しいデータを次々に配列に入れながら、データ数分だけ描画（グラフが右に伸びる）していきます。データ数が NUM_POINTS を越えたら、古いデータを捨て（pop）、時間軸の右端と左端の時刻を進めます。こうすることで、時間軸とデータが左へシフトしていきます。

SAVE_DATA を定義しておけば、コンソールにデータを吐き出すので、リダイレクトしてファイルに格納できます。

recorder.py

```
/* 温度コントローラ用ネットワークレコーダー
    開始： 2017/3/3 Chuji
    初版完成： 2017/3/10
    最新版：  2019/3/9 --- タイムスタンプを表示に利用
*/

/* 動作指定：  ファイルの表示か、WiFi 受信データか */
/*
#define SHOW_LOCAL_FILE
*/

#define SAVE_DATA

#include "./include/recorder.h"


#ifdef SHOW_LOCAL_FILE /* ファイルの表示 */
f=open(SHOW_LOCAL_FILE)
lines = f.readlines()
f.close

#else /* WiFi からの受信準備 */
mysocket = wifi_socket()
#endif


/* 描画領域の指定 */
fig = plt.figure(figsize=(REC_FIG_WIDTH,
REC_FIG_HEIGHT))

/* グラフ部の大きさ定義 */
trend = plt.axes(REC_TREND_SIZE)

/* 横軸の両端 */
t_min = MIN_T
t_max = MAX_T

/* 縦横軸の両端定義 */
trend.set_xlim(t_min, t_max)
trend.set_ylim(MIN_Y, MAX_Y)


/* 表示するデータ配列を確保 */
pv = []
sp = []
mv = []
t  = []


/* 記録開始 */
print "start receiving"

try:

#ifdef SHOW_LOCAL_FILE
  for line in lines:
#else
  while True:
    line = mysocket.subscribe()
#endif
    /* 現在の時刻と PV, SP, MV を取得 */
    clusters = re.findall('\S+', line)
    if len(clusters) == REC_DATA_FIELDS:
      ctime = float(clusters[REC_POS_TIME])
      cmode = int(clusters[REC_POS_MODE])
      cpv = float(clusters[REC_POS_PV])
      csp = float(clusters[REC_POS_SP])
      cmv = float(clusters[REC_POS_MV])

#ifdef SAVE_DATA
      print ctime, cmode, cpv, csp, cmv
#endif
      ctime = ctime/TIME_UNIT
      t.append(ctime)
      pv.append(cpv)
      sp.append(csp)
      mv.append(cmv)

      /* 表示点数を越えたら、古いデータを削除 */
      if len(pv) > NUM_POINTS:
        t.pop(0)
        pv.pop(0)
        sp.pop(0)
        mv.pop(0)

      /* 横軸表示範囲の移動 */
      while ctime > t_max:
        t_min += UNIT_T
        t_max += UNIT_T
      trend.set_xlim(t_min, t_max)

      /* MIN_POINTS 点目から表示開始 */
      if len(pv) == MIN_POINTS:
        pv_dat, = START_TREND(trend, t, pv,
REC_PV_COLOR)
        sp_dat, = START_TREND(trend, t, sp,
REC_SP_COLOR)
        mv_dat, = START_TREND(trend, t, mv,
REC_MV_COLOR)
      elif len(pv) > MIN_POINTS:
        DRAW_TREND(pv_dat, t, pv)
        DRAW_TREND(sp_dat, t, sp)
        DRAW_TREND(mv_dat, t, mv)
```

```
      plt.pause(FRACTION_OF_TIME)

except KeyboardInterrupt:
#ifndef SHOW_LOCAL_FILE
  mysocket.close()
#endif
  pass
```

検証

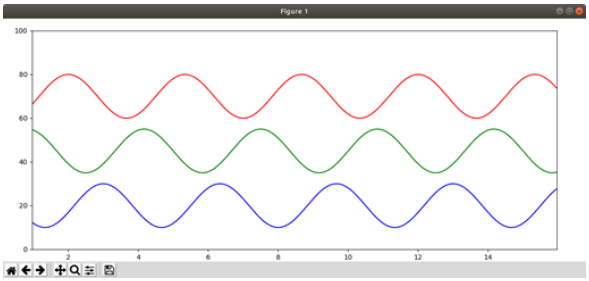
検証のため、プログラムで発生させた波形を表示させてみます。

gen-trend.py

```
import math

#ifdef NETWORK_RECORDER
#include "./include/use-time.h"
#include "wifi.py"
mysocket = wifi_socket()
#endif

omega = 0.01

for i in range(1000):
  pv = 70 + 10 * math.sin(math.pi * omega * (i -
70))
  sp = 45 + 10 * math.sin(math.pi * omega * i)
  mv = 20 + 10 * math.sin(math.pi * omega * (i +
70))
  print i, 2, pv, sp, mv

#ifdef NETWORK_RECORDER
  msg = '%f %d %f %f %f' % (i, 2, pv, sp, mv)
  mysocket.publish(msg)
  sleep(1)
#endif
```

このプログラムを（NETWORK_RECORDER を #define せずに）実行して作ったトレンドファイル（trend.csv）をレコーダーで描画してみます。

```
$ cpp gen-trend.py | python >trend.csv
$ cpp -DSHOW_LOCAL_FILE=\'trend.csv\'
recorder.py | python
```



トレンド記録例

横軸の単位は分で、トレンドが右端まで達すると、全体が左へシフトしていきます。PC が 2 台あればネットワークからの受信の検証もできるのですが、今回は実機での検証まで待つことにします。

パソコンこと始め

先のコラム（55 ページ）でチュンたちが使ったのは、個人用コンピュータの草分けと言われる、Altair 8800 でした。CPU は 8080、メモリは最大 64K バイト、外部記憶装置はフロッピーディスクまたはカセットテープで、BASIC インタープリタが走るという代物です。1975 年の発売当初、256 バイトのメモリを搭載した組み立てモデル（キーボード、ディスプレー、外部記憶を含まない、箱型）が 395 ドル、完成品が 660 ドルだったそうです（キットは不良部品が多く、ほとんど動作しなかったらしい）。4K バイトのメモリボード（完成品）は 275 ドルもしました。ちなみに、この年の為替レートは 1 ドル＝210 円です。

キーボード、CRT、カセットレコーダまたはフロッピードライブがまとまった、PET-2001（795 ドル）、Apple II（4K メモリモデルが 1298 ドル）、TRS-80（599.95 ドル）の「御三家」が発売されたのは 1977 年のことです。日本では 1975 年に、PC8001 が 168,000 円で発売されました。このころは my computer 略してマイコンと呼ばれていました。

満を持して IBM PC（1500～3000 ドル）が発売されたのは 1981 年のことです。Microsoft が納入した PC-DOS を搭載していました。Personal Computer という言葉は 1970 年代の初めからあったそうですが、これ以後は IBM PC（と互換機）を指すようになりました。PC-DOS を互換機用に外販したのが MS-DOS です。

5　モジュールの検証

ソフトウエアモジュールの検証（動作テストとデバグ）を行います。最初は PC 上で動作させ、GPIO はコンソール入出力やファイルで代用します。基本的にコンソールからの操作なので、Raspberry Pi に SSH 接続して行うこともできますが、いろいろなツールを使えるので PC 上の方が便利です。この章では Ubuntu PC 上での検証を前提に説明します。

自作を含めたハードウエアの動作試験には、用意したソフトウエアモジュールを使う方が試験範囲を広く取れるので、ソフトウエアモジュールの評価を先に済ませておきます。

実機でのみ動作する controller.py モジュールを除き、すべてのモジュールのできるだけ多くの部分を、PC 上で検証していきます。その中で大事なのは、同じ検証を何度でも繰り返すことができるようにするということです。ソフトウエアモジュールを改造したら、以前と同じ検証を行って、改造時の過ちで機能を変えてしまっていないか確認する必要があるからです。そのために各ソフトウエアモジュールを評価するための、検証用モジュールを用意しておきます。簡単な確認を以外は手動にせず、検証を自動化するように心がけています。そうすることで、何度でも同じ条件で検証できるからです。

5.1　検証に使うモジュール

ソフトウエアモジュールをひとつずつ検証していき、それを使って次のモジュールを検証するようにします。とはいえ、PC 上では実現が難しい機能が必要になる時は、代替用ソフトウエアモジュールを用意しなければなりません。

5.1.1温度モデル

制御対象となるスモーカーなどを使わずにソフトウエアモジュールを動作させるため、制御対象をモデル化しておきます。このモジュールがあれば、PWM による加熱や温度センサによる温度測定をしなくても、PC 上で制御を実現できるからです。

モデルの温度は、周囲との温度差に比例する速さで冷えて（あるいは熱くなって）いきます。これをニュートンの法則といい、この法則をもとにモデルを作ります。

加熱すれば温度が上がり、加熱しなければ周囲温度に近づくように冷えていくのをプログラムにすると、以下のようになります。おまけとして、周囲温度がゆっくり揺らいでいる状況を再現しました。

sim-model.py

```
/* ニュートンの法則による熱モデル
     初版： 2015/1/11 Chuji
     最新版：2019/1/23 - 単純化

Class: thermal_model
属性:
  t:    モデルの現在温度
  time: 揺らぎを与えるための時刻
操作:
  temperature: 現在の温度を得る

*/

#ifndef __SIMULATION

#include "./include/pid.h"
#include "./include/thermo.h"
#include "./include/use-math.h"

#define HEATING_COEFF  0.01
#define COOLING_COEFF  0.01

#ifndef FLUCT_AMP
#define FLUCT_AMP  2  /* 揺らぎの振幅（℃） */
#endif

#ifndef FLUCT_PERIOD
#define FLUCT_PERIOD  50  /* 揺らぎの周期（秒） */
#endif

#ifndef T_INITIAL
#define T_INITIAL  60  /* 初期温度 */
#endif

#ifndef T_AMBIENT
#define T_AMBIENT  25  /* 周囲温度 */
#endif

class thermal_model:
  def __init__(self):
    self.t = T_INITIAL
    self.time = 0.0

  def temperature(self, x):
    global math
    self.time += PID_PERIOD
    fluctuation = FLUCT_AMP * math.sin( 2 * math.pi
* self.time / FLUCT_PERIOD)
    ta = T_AMBIENT + fluctuation

    self.t += (HEATING_COEFF * x - COOLING_COEFF *
(self.t - ta)) * PID_PERIOD
    return self.t


#define __SIMULATION

#endif
```

5.1.2検証モジュール

ソフトウエアモジュールを検証するソフトウエアには、すべて test_xxx.py というファイル名を付けておきました。xxx は、検証対象となるモジュールです。こういうモジュールは使い捨てに近いと思って、マジックナンバーを埋め込みたくなりますが、間違えようのない数値以外は、ソースコードの冒頭で定義をするように努力しましょう。

次節からは、検証対象となるソフトウエアモジュール毎に用意した検証モジュールの説明と、結果を説明します。

5.2　シミュレーション環境での検証

実機（GPIO）を使わず（BCM2835 を#define せずに）にソフトウエアモジュールを検証していきます。モジュールは結果を標準出力（コンソール）に出すので、必要によってファイルにリダイレクトして後から調べます。Excel などの表計算ソフトで処理したり、グラフ化したりする場合もあります。

なお、PWM モジュール pwm.py はほとんど間違いようのないプログラムなので、特に検証モジュールは用意しません。HMI モジュールの検証のときに一緒に検証してしまいます。

5.2.1代替温度モデル

各モジュールの検証に使う、sim-model.py を最初に検証しておきます。温度モデルを 500 回呼んで、温度変化を調べます。加熱量 HEATING が 0 のときは、温度は徐々に周囲温度に近づいていくはずです。

検証モジュール

sim-model.py の検証モジュールを以下に示します。初期温度、周囲温度、加熱、繰り返し回数、揺らぎの振幅などは cpp のオプションとして変更できます。

test_sim.py

```
/* ニュートンの法則による熱モデルの検証
    初版：2016/11/76 Chuji */

#ifndef INITIAL_TEMP
#define INITIAL_TEMP 60.0
#endif

#ifndef AMBIENT_TEMP
#define AMBIENT_TEMP 20.0
#endif

#ifndef HEATING
#define HEATING 0.0
#endif

#ifndef REPEAT
#define REPEAT 500
#endif

#ifndef FLUCT_AMP
#define FLUCT_AMP 5.0
#endif

#include "sim-model.py"


model = thermal_model()

for i in range(0, REPEAT):
  print i, model.temperature(HEATING)
```

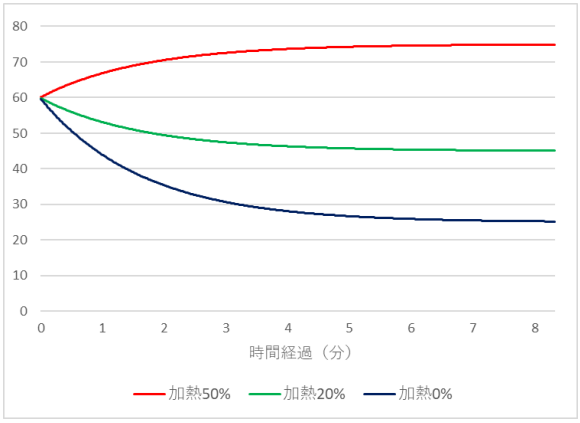
例えば、加熱 20%で試験するには、以下のようにコマンド入力します。bash を使っているので、上矢印キーで前のコマンドを呼び出せ、すこし修正して繰りかえすことができます。

```
$ cpp -DHEATING=20 test_sim.py |python cleanfile.py
| python >test_sim_result20.csv
```

結果をいれた csv ファイルを Excel などの表計算ソフトで開いて解析します。データを読み込むとき、データ区切りにスペースを使うように指定してください（通常はコンマまたはタブしか区切りに使わないため）。

結果

初期温度 60℃、周囲温度 25℃、揺らぎ振幅 0 で、加熱を 0%、20%、50%としたときの結果を下のグラフに示します。



温度モデルの検証結果

加熱しなければ、徐々に周囲温度 25℃に近づくこと、加熱すれば加熱と冷却が釣り合う温度に近づくことが分かります。期待どおりの動作なので、この後はこのモデルの温度制御をすることで、各ソフトウエアモジュールを検証していきます。

5.2.2I2C バス

ここでは、I2C バスへの読み書きを試します。I2C バスアドレスとレジスタ、データが正しく伝わるか、読み込み禁止の液晶表示器（LCD）から読み込もうとしたら何が起こるか検証します。

検証モジュール

このモジュールは GPIO を使わない検証用に用意したのですが、実は BCM2835 を#define すれば実機でも同じ評価ができます。しかし、レジスタへ書き込むデータを変更したり、読み出したデータを解釈したりするのは面倒なので、次節にある温度センサドライバを使った検証の方が簡単です。

まず温度センサのステータスレジスタと温度測定値を読み出し（実際にはキーボードから入力し）て、表示します。次に設定レジスタに設定を書き込みます。LCD の書き込みアドレスを一行目の左端に設定し、4 文字を書き込みます。最後に LCD からデータを読み出そうとしますが、失敗することを確認します。

test_i2c.py

```
/* I2C バスの動作テスト */
/* 2016/11/9 Chuji */

#define DUMMY_STRING "ABCD"

/*
#define BCM2835
*/

#include "./include/thermo.h"
#include "./include/lcd.h"

#include "i2c.py"

temperature = i2c_device(I2C_TEMP_ADDR,
I2C_READ_WRITE)
lcd = i2c_device(I2C_LCD_ADDR, I2C_WRITE_ONLY)

print "READ BYTE (status)"
dat = temperature.read_byte(TEMP_STATUS_REG)
print hex(dat.value), dat.status
print

print "READ WORD (temperature)"
dat = temperature.read_word(TEMP_VALUE_REG)
print hex(dat.value), dat.status
print

print "WRITE BYTE (config)"
result = temperature.write_byte(TEMP_CONFIG_REG,
TEMP_CONFIG_DATA)
print "Result = ", result
print

print "WRITE BYTE TO LCD"
result = lcd.write_byte(LCD_CMD_REG,
LCD_CMD_CHAR_ADDR)
print "Result = ", result
print

print "WRITE STRING TO LCD"
result = lcd.write_block(LCD_DATA_REG, DUMMY_STRING)
print "Result = ", result
print

print "ILLEGAL BYTE READ FROM LCD"
dat = lcd.read_byte(LCD_CMD_REG)
print "Result = ", hex(dat.value), dat.status

print "ILLEGAL WORD READ FROM LCD"
dat = lcd.read_word(LCD_DATA_REG)
print "Result = ", hex(dat.value), dat.status
```

結果

キーボードから入力を得なければならないので、cpp の結果をいったんファイルにしまってから、それを実行しています。

温度センサ（0x48）のアドレス 0x2 と 0x0 から読み出そうとしますが、キーボードから与えたデータが帰ってきています。

書き込み動作は［データ］=> ［I2C アドレス］:［レジスタ］と表示されます。この I2C アドレスを持つデバイスのレジスタにデータを書き込むという表示で確認できます。LCD からデータを読み出そうとすると、BAD（1）を返して、読み出しを行わないことが確認できます。

```
$ cpp test_i2c.py >python.py; python python.py
READ BYTE (status)
I2C read byte data from address 0x48 :(Reg) 0x2 in
hex?
? 0xAB
0xab 0

READ WORD (temperature)
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
0x1234
0x1234 0

WRITE BYTE (config)
I2C byte write: 0x0 => 0x48 : 0x3
Result =  0

WRITE BYTE TO LCD
I2C byte write: 0x80 => 0x3e : 0x0
Result =  0

WRITE STRING TO LCD
I2C byte write: 0x41 => 0x3e : 0x40
I2C byte write: 0x42 => 0x3e : 0x40
I2C byte write: 0x43 => 0x3e : 0x40
I2C byte write: 0x44 => 0x3e : 0x40
Result =  0

ILLEGAL BYTE READ FROM LCD
Result =  0x0 1
ILLEGAL WORD READ FROM LCD
Result =  0x0 1
```

5.2.3温度測定

温度センサのレジスタ値を読み取って、thermo.py が期待どおりの温度を返すか確認します。

テストパターン

検証する温度とレジスタ値のパターンは下表のとおりです。ADT7410A の測定温度レジスタはワードデータで、I2C バス通信によって上下バイトが逆転していることに注意してください。なお STTS751 の場合、-55℃と-50℃は仕様外ですが、可能なレジスタ値として評価しています。+150℃は仕様外だし、レジスタ値範囲外です。

テストパターンを決めるのに、境界値法を使います。これは入力パターンの「構造」が変わる前後でテストし、評価漏れを防ぐことです。具合的には、測定可能温度の上下限、符号が変わるところ、最下位ビットの反転、上下位バイトの変わるところのパターンを与えます。

温度（期待値）	ADT7410A	STTS751	
	ワード	上位バイト	下位バイト
-55℃	0x80E4	0xC9	0x00
-50℃	0x00E7	0xCE	0x00
-25℃	0x80F3	0xE7	0x00
-1℃	0x80FF	0xFF	0x00
-0.0625℃	0xF8FF	0xFF	0xF0
0℃	0x0000	0x00	0x00
0.0625℃	0x0800	0x00	0x10
+1℃	0x8000	0x01	0x00
+25℃	0x800C	0x19	0x00
+50℃	0x0019	0x32	0x00
+125℃	0x803E	0x7D	0x00
+150℃	0x004B	N/A	

温度センサのテストパターン

検証モジュール

検証モジュールは以下のとおりです。なお、TEMP_SIMULATION を#define して、このモジュールを走らせると、前に行った代替温度モデルの検証と同じ結果が得られます。

test_thermo.py

```
/* 温度センサドライバの試験
    2016/11/6 Chuji */

#define REPEAT 200
#define MV 0

/*
#define STTS751
*/


#include "thermo.py"
model = thermometer(I2C_TEMP_ADDR)


for i in range(REPEAT):
#ifndef TEMP_SIMULATION
  temp = model.read_temp()
#else
  temp = model.read_temp(MV)
#endif
  print "T =", temp.value,"degC"  print "T =",
temp.value,"degC"
```

まずキーボードから温度を入力することにして、検証モジュールを実行してみます。STTS751 の場合を下に示します。まず設定レジスタ（0x03）と変換レートレジスタ（0x04）に書き込んで、分解能 12 ビットでワンショット測定をするように設定しています。次にワンショットトリガ（0xF）に書き込みを行うことで測定を始めます。ステータスレジスタ（0x1）を読み取って、最上位ビットがクリアされる（測定が終了する）まで待ちます。一回目は BUSY（0x80）を返し、二回目で終了（0x00）と返答してみました。すると上位バイト（0x0）、次に下位バイト（0x2）のデータを読み出そうとするので、それぞれ 0xC9 と 0x00 を返してやると、測定値-55℃を表示しました。

```
$ cpp  -DSTTS751 test_thermo.py|python
I2C byte write: 0x4c => 0x48 : 0x3
I2C byte write: 0x7 => 0x48 : 0x4
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? 0x80
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? 0x00
I2C read byte data from address 0x48 :(Reg) 0x0 in
hex?
? 0xC9
I2C read byte data from address 0x48 :(Reg) 0x2 in
hex?
? 0x00
T = -55.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
?
```

I2C バスを介した読み取りデータをいちいち手で与えるのは面倒なので、温度センサモジュールが測定手順を実行しているのを確認したら、自動検証することにします。キーボードからの入力を与えるファイル thermo_test_pattern_7410 と thermo_test_pattern_751 を作成して、リダイレクトで検証モジュールに与えてやります。各ファイルの内容は、上で述べたレジスターパターンどおりです。なお、印刷の都合で thermo_test_pattern_751 は 3 行に分割して示してあります。

thermo_test_pattern_7410	thermo_test_pattern_751		
0x80E4	0x00	0x00	0x00
0x00E7	0xC9	0xFF	0x00
0x80F3	0x00	0xF0	0xFF
0x80FF	0x00	0x00	0x00
0xF8FF	0xCE	0x00	0x00
0x0000	0x00	0xC9	0xFF
0x0800	0x00	0x00	0xF0
0x8000	0xE7	0x00	0x00
0x800C	0x00	0xCE	
0x0019	0x00	0x00	
0x803E	0xFF	0x00	
0x004B	0x00	0xE7	

温度センサのテストパターンファイル

結果

ADT7410A の結果を下に示します。検証モジュールをいったん python.py に変換し、次のコマンドでテストパターンを与えています。初期設定はレジスタ（0x03）への書き込み一回だけです。結果は最初の表どおりになりました。なお、最後の Traceback (most recent call last):以降は、テストパターンファイルの最後まで行ったので、「入力がなくなった」という終了メッセージです。

```
$ cpp  test_thermo.py|python cleanfile.py >python.py
$ python python.py <thermo_test_pattern_7410
I2C byte write: 0x0 => 0x48 : 0x3
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = -55.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = -50.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = -25.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = -1.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = -0.0625 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 0.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 0.0625 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 1.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 25.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 50.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 125.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
T = 150.0 degC
I2C read word data from address 0x48 :(Reg) 0x0 in
hex?
Traceback (most recent call last):
  File "python.py", line 61, in <module>
    temp = model.read_temp()
  File "python.py", line 47, in read_temp
    self.pv = self.i2c.read_word(0x00)
  File "python.py", line 25, in read_word
    self.data.value = input()
  File "<string>", line 1
```

STTS751 の場合の結果を下に示します。終了メッセージは削除しています。結果はやはり最初の表のとおりでした。

```
$ cpp  -DSTTS751 test_thermo.py|python
cleanfile.py >python.py
$ python python.py <thermo_test_pattern_751
I2C byte write: 0x4c => 0x48 : 0x3
I2C byte write: 0x7 => 0x48 : 0x4
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = -55.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = -50.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = -25.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = -1.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = -0.0625 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = 0.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = 0.0625 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = 1.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = 25.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
```

```
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = 50.0 degC
I2C byte write: 0x0 => 0x48 : 0xf
I2C read byte data from address 0x48 :(Reg) 0x1 in
hex?
? I2C read byte data from address 0x48 :(Reg) 0x0
in hex?
? I2C read byte data from address 0x48 :(Reg) 0x2
in hex?
? T = 125.0 degC
```

5.2.4液晶表示器

液晶表示器ドライバーの検証モジュールを以下に示します。最初に上の行の左から 5 番目（左端はアドレス 0）から"ABC"を表示します。次に左端から"Overwritten"と上書きします。下の行の中央に"Middle"と表示し、次に 10 文字目から"Truncated"と表示しようとしますが、右端にぶつかるので途中で書き込みをやめるはずです。最後は"Hello Word!"という文字が左から右へ流れるようにしています。

test_lcd.py

```
/* 液晶表示器の試験
   2016/11/12 Chuji */

/*
#define BCM2835
*/

#define HELLO "Hello World! "

#ifdef BCM2835
#include "./include/use-time.h"
#endif

#include "lcd.py"
lcd = lcd_device()

row = 0
pos = 4
string="ABC"
print "display position on line", row, "at
postion", pos, "with", len(string),"chars"
lcd.put_string(row, pos, string, len(string))

#ifdef BCM2835
sleep(2)
#endif

row = 0
pos = 0
string="Overwritten"
print "display position on line", row, "at
postion", pos, "with", len(string),"chars"
lcd.put_string(row, pos, string, len(string))

#ifdef BCM2835
sleep(2)
#endif

row = 1
pos = 6
string = "Middle"
print "display position on line", row, "at
postion", pos, "with", len(string),"chars"
lcd.put_string(row, pos, string, len(string))

#ifdef BCM2835
sleep(2)
#endif
```

```
row = 1
pos = 10
string = "truncated"
print "display position on line", row, "at
postion", pos, "with", len(string),"chars"
lcd.put_string(row, pos, string, len(string))

#ifdef BCM2835
sleep(2)
#endif


print "display move"
for i in range(LCD_LINE_LEN):
  lcd.put_string(0, LCD_LINE_LEN - 1 -i, HELLO,
len(HELLO))

#ifdef BCM2835
  sleep(1)
#endif
```

結果

結果は長いので、最初の 3 文字書き込みまでを以下に示します。書き込みはコマンドレジスタ（0x0）とデータレジスタ（0x40）に行っています。初期化時のデータレジスタへの書き込みと 200ms 待ちなどは、液晶の仕様書どおりになっていることが確認できました。3 文字書き込みでは、まずコマンドレジスタにアドレス（0x80+0x04）を設定し、連続して"A"（0x41）、"B"（0x42）、"C"（0x43）とデータレジスタに書き込んでいることが分かります。

```
I2C byte write: 0x38 => 0x3e : 0x0
I2C byte write: 0x39 => 0x3e : 0x0
I2C byte write: 0x14 => 0x3e : 0x0
I2C byte write: 0x73 => 0x3e : 0x0
I2C byte write: 0x56 => 0x3e : 0x0
I2C byte write: 0x6c => 0x3e : 0x0
wait for 0.2 sec
I2C byte write: 0x38 => 0x3e : 0x0
I2C byte write: 0xc => 0x3e : 0x0
I2C byte write: 0x1 => 0x3e : 0x0
display position on line 0 at postion 4 with 3
chars
I2C byte write: 0x84 => 0x3e : 0x0
I2C byte write: 0x41 => 0x3e : 0x40
I2C byte write: 0x42 => 0x3e : 0x40
I2C byte write: 0x43 => 0x3e : 0x40
```

この後の結果をすべて読む必要はなくて、最初のアドレス設定のみ確認すれば十分です。左端から 11 文字は 0x00 に、下行中央から 6 文字は 0xC6（0x80+0x46）に、10 文字目から 9 文字は 0xCA（0x80+0x4A）に（実際の書き込みは右端にあたるまでの 6 文字のみ）、そのあとの移動は右端の 0x8F（0x80+0x0F）から始めてアドレスが減少していくのが確認できます。

この評価は、イメージ生成モジュールを介して表示イメージを見るか、実機で確認すると意図した表示パターンがよくわかります。

5.2.5数値→文字列変換

今まではハードウエアに近いレベルのモジュールでしたが、ここから先は抽象レベルなので、全ての機能が PC 環境で検証できます。

数値から文字列への変換の検証モジュールでは、値の範囲の外でどうなるか、範囲内で四捨五入ができているかを確認します。書式は 199.9（書式 1)、9.9（書式 2)、999（書式 3）について行いましたが、実際に温度コントローラで使っているのは書式 1 と書式 3 だけです。

test_ftos.py

```
/* 浮動小数点数から文字列への変換するルーチンの試験
   2016/11/12 Chuji */

#include "ftos.py"

def test_ftos(val, string):
  print val, "in format(", string, ") is
<"+ftos(val, string)+">"

test_ftos(-5, FORMAT_199_9)
test_ftos(0, FORMAT_199_9)
test_ftos(50.21, FORMAT_199_9)
test_ftos(120.55, FORMAT_199_9)
test_ftos(200.0, FORMAT_199_9)
test_ftos(-0.3, FORMAT_9_9)
test_ftos(0.51, FORMAT_9_9)
test_ftos(8.66, FORMAT_9_9)
test_ftos(10.2, FORMAT_9_9)
test_ftos(-2.2, FORMAT_999)
test_ftos(3, FORMAT_999)
test_ftos(40.1, FORMAT_999)
test_ftos(320.8, FORMAT_999)
test_ftos(1000.3, FORMAT_999)
```

結果

結果は以下のとおりです。書式の範囲外では最小値あるいは最大値になっていること、範囲内では四捨五入ができていることが確認できます。また数字の前にある空白を加えると、文字数が書式どおりになっています。

```
$ cpp test_ftos.py |python
-5 in format( 1 ) is <  0.0>
0 in format( 1 ) is <  0.0>
50.21 in format( 1 ) is < 50.2>
120.55 in format( 1 ) is <120.6>
200.0 in format( 1 ) is <199.9>
-0.3 in format( 2 ) is < 0.0>
0.51 in format( 2 ) is < 0.5>
8.66 in format( 2 ) is < 8.7>
10.2 in format( 2 ) is < 9.9>
-2.2 in format( 3 ) is <   0>
3 in format( 3 ) is <   3>
40.1 in format( 3 ) is <  40>
320.8 in format( 3 ) is < 321>
1000.3 in format( 3 ) is < 999>
```

5.2.6表示イメージ生成

表示イメージ生成の検証モジュールは、生成するオブジェクト以外は液晶表示器用と同じです。ここで LCD_SILENT を#define して、I2C バスの動作を止めることで、表示を単純にしています。

test_display.py

```
/* 表示イメージ生成の評価
   2016/11/12 Chuji */

/*
#define BCM2835
*/

#define HELLO "Hello World! "

#include "display.py"
lcd = display_image()


row = 0
pos = 4
string="ABC"
print "display position on line", row, "at postion",
pos, "with", len(string),"chars"
lcd.show(row, pos, string, len(string))


row = 0
pos = 0
string="Overwritten"
print "display position on line", row, "at postion",
pos, "with", len(string),"chars"
lcd.show(row, pos, string, len(string))

row = 1
pos = 6
string = "Middle"
print "display position on line", row, "at postion",
pos, "with", len(string),"chars"
lcd.show(row, pos, string, len(string))

row = 1
pos = 10
string = "truncated"
print "display position on line", row, "at postion",
pos, "with", len(string),"chars"
lcd.show(row, pos, string, len(string))

print "display move"
for i in range(DISP_STRING_LEN):
    lcd.show(0, DISP_STRING_LEN - 1 -i, HELLO,
len(HELLO))
```

結果

結果を下に示します。これも長いので、最初の方だけです。思っていた表示イメージが得られました。この後は、上の行の右から左に”Hello World!”という文字が動いていくのが確認できました。

```
$ cpp -DLCD_SILENT test_display.py |python |more
display position on line 0 at postion 4 with 3 chars
+----------------+
|    ABC         |
+----------------+
|                |
+----------------+

display position on line 0 at postion 0 with 11 chars
+----------------+
|Overwritten     |
+----------------+
|                |
+----------------+
```

```
display position on line 1 at postion 6 with 6 chars
+----------------+
|Overwritten     |
+----------------+
|      Middle    |
+----------------+

display position on line 1 at postion 10 with 9 chars
+----------------+
|Overwritten     |
+----------------+
|      Middtrunca|
+----------------+
```

5.2.7PID 制御

PID 制御モジュールは機能が複雑なので、2 段階に分けて検証します。

1. 外部からの操作への反応
2. PID 制御アルゴリズムの実行

1.の操作への反応には、異常処理（あまり起こらないような操作への反応）を含みます。2.の PID 制御は、実際に代替温度モデルに対して、制御を実行してみます。

操作への反応を検証

まず検証する操作手順を定義します。このテストパターンに従って、具体的な操作（PID オブジェクトの操作を実行すること）をプログラムし、モード、PV、SP、MV を表示して期待どおりか確認していきます。

No	操作	期待する反応
1	O/S で実行	PV を SP にコピー
2	ターゲットを MAN	MAN に移行。PV→SP
3	MV を増やす	MV が増加
4	MV を減らす	MV が減少
5	SP を増やす・減らす	PV は変わらない
6	ターゲットを AUT	AUT に移行
7	SP を増やす	SP が増加
8	SP を減らす	SP が減少
9	MV を増やす、減らす	MV は変わらない
10	PV のステータスを BAD	MV は追従していく
11	（上の繰り返し）	3 回目で MAN、MV→BAD
12	GOOD にする	MV→GOOD
13	ターゲットを AUT	AUT に移行
14	P, I, D をセット	セットされる
15	ターゲットを MAN	MAN に移行
16	ターゲットを O/S	O/S に移行、MV→0

PID 制御のテストパターン

テストパターンの番号順に操作します。自動実行するように評価モジュールをプログラムします。

test_pid1.py

```
/* 操作に対する PID の反応を検証する
   2016/11/13 Chuji */


#include "./include/process_value.h"
#include "./include/display.h"
#include "pid.py"

modes = [DISP_OUT_OF_SERVICE, DISP_MANUAL,
DISP_AUTO]
```

低温調理器

アメリカの家庭には、スロークッカーという電気調理器が普及しています。蓋のついた鍋型のもので、長時間の調理に使います。放っておけばいいので、楽ちんなんだそうです。いかにも大らかでアメリカ的な器具ですね。ヒーターの切り替えをするだけの簡単な機種もありますが、調理温度と時間を設定できる機種では、70℃で 3 時間といった低温調理ができます。日本にも輸入されていますが、むしろ電気炊飯器の保温機能を使って同じことをする人が多いようです。

ヨーロッパでは、太い棒の先にヒーターと撹拌機がついた、イマージョン（水没）クッカーというもののほうが、普及しています。鍋に放り込んで使います。日本には低温調理器とか真空調理器（ジッパー袋に密封して湯煎するため）という名称で輸入されていますが、あまり売れてはいないようです。

この本で取り上げた温度コントローラでも、同じことができます。炊飯器でも良いのではないかというご意見には、ご飯を炊いているときでも調理ができる、という言い訳をしています。

```
#define SHOW_ALL  print modes[pid.mode],
pid.pv, pid.sp, mv.value, ":", mv.status

pid = pid_block()

pv = process_value()
pv.value =  20
pv.status = STATUS_GOOD
mv = process_value()
mv.value = 0
mv.status = STATUS_GOOD

print "Test case 1"
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 2"
pid.set_target(PID_MANUAL)
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 3"
pid.inc_mv()
pid.inc_mv()
pid.inc_mv()
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 4"
pid.dec_mv()
pid.dec_mv()
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 5"
pid.inc_sp()
pid.inc_sp()
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL

pid.dec_sp()
pid.dec_sp()
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 6"
pid.set_target(PID_AUTO)
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 7"
pid.inc_sp()
pid.inc_sp()
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 8"
pid.dec_sp()
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 9"
pid.inc_mv()
pid.inc_mv()
```

```
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL

pid.dec_mv()
pid.dec_mv()
print "before execution"
SHOW_ALL
print "after execution"
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 10"
pv.status = STATUS_BAD
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 11"
mv = pid.execute(pv)
SHOW_ALL
mv = pid.execute(pv)
SHOW_ALL
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 12"
pv.status = STATUS_GOOD
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 13"
pid.set_target(PID_AUTO)
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 14"
print "PID parms:", pid.p, pid.i, pid.d
pid.tune(5, 20, 3)
print "PID parms:", pid.p, pid.i, pid.d
print

print "Test case 15"
pid.set_target(PID_MANUAL)
mv = pid.execute(pv)
SHOW_ALL
print

print "Test case 16"
pid.set_target(PID_OUT_OF_SERVICE)
mv = pid.execute(pv)
SHOW_ALL
print
```

実行結果は以下のとおりでした。期待どおりの結果だと確認できます。

```
$ cpp test_pid1.py |python
Test case 1
before execution
O/S  0.0 0.0 0 : 0
after execution
O/S  20 20 0.0 : 0

Test case 2
MAN  20 20 0.0 : 0

Test case 3
before execution
MAN  20 20 0.0 : 0
after execution
MAN  20 20 3.0 : 0

Test case 4
before execution
MAN  20 20 3.0 : 0
after execution
MAN  20 20 1.0 : 0
```

```
Test case 5
before execution
MAN  20 20 1.0 : 0
after execution
MAN  20 20 1.0 : 0
before execution
MAN  20 20 1.0 : 0
after execution
MAN  20 20 1.0 : 0

Test case 6
AUT  20 20 1.0 : 0

Test case 7
AUT  20 22.0 5.4 : 0

Test case 8
AUT  20 21.0 3.6 : 0

Test case 9
before execution
AUT  20 21.0 3.6 : 0
after execution
AUT  20 21.0 3.8 : 0
before execution
AUT  20 21.0 3.8 : 0
after execution
AUT  20 21.0 4.0 : 0

Test case 10
AUT  20 21.0 4.2 : 0

Test case 11
AUT  20 21.0 4.4 : 0
AUT  20 21.0 4.6 : 0
MAN  20 20 4.6 : 1

Test case 12
MAN  20 20 4.6 : 0

Test case 13
AUT  20 20 4.6 : 0

Test case 14
PID parms: 2.0 10.0 0.0
PID parms: 5 20 3

Test case 15
MAN  20 20 4.6 : 0

Test case 16
O/S  20 20 0.0 : 0
```

制御の実行を検証

では、実際に PID 制御を実行してみます。制御対象は代替温度モデルです。温度モデルから現在温度を読み取り、PV として PID 制御に渡します。出力の MV を温度モデルに与え、次の温度を計算していきます。その時、以下の操作パターンで行います。

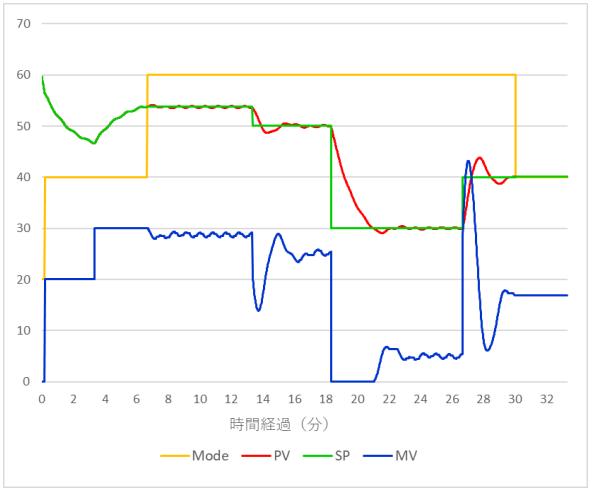
時刻（秒）	操作
0	制御開始
10	モードを O/S から MANUAL へ変更、MV を 20 に
200	MV を 30 に変更
400	モードを AUTO へ変更（SP は測定温度に追従）
800	SP を 50℃に設定
1100	SP を 30℃に変更
1600	SP を 40℃に変更
1800	PV のステータスを BAD にする

検証モジュールを下に示します。SP や MV を変更するのに、「禁じ手」である属性への直接書き込みを行っているのは、プログラムを単純化するためです。本当なら inc_sp()を 20 回呼ぶなどの操作が必要です。

test_pid2.py

```
/* 温度モデルを PID 制御する試験
   2016/11/13 Chuji */

#define TEMP_SIMULATION

#include "./include/display.h"

#include "thermo.py"
#include "pid.py"

modes = [DISP_OUT_OF_SERVICE, DISP_MANUAL,
DISP_AUTO]

model = thermometer(I2C_TEMP_ADDR)
pid = pid_block()

for i in range(2000):

  pv =  model.read_temp(pid.mv)
  if i>= 1800:
    pv.status = STATUS_BAD
  mv = pid.execute(pv)
  print i, modes[pid.mode], pid.pv, pid.sp,
pid.mv

  if i == 10:
    pid.set_target(PID_MANUAL)
    pid.mv = 20
  if i == 200:
    pid.mv = 30
  if i == 400:
    pid.set_target(PID_AUTO)
  if i == 800:
    pid.sp = 50
  if i == 1100:
    pid.sp = 30
  if i == 1600:
    pid.sp = 40
```

実行結果はコンソール出力なので、ファイルにリダイレクトしたものを表計算ソフトでグラフ化しました。モードは O/S を 20、MANUAL を 40、AUTO を 60 として数値化して重ねてあります。



PID 制御の検証結果

開始 10 秒後から 400 秒までは MANUAL モードなので、MV（青）は手動で設定しています。PV（赤）は平衡温度に近づいていきます（この変化を指数関数的といいます）。SP（緑）は PV に追従しているので重なって見えます。400 秒に AUTO モードに切り替えると、PV は SP に追従していきます。MV は加熱する必要がある時に増大し、温度が上がりすぎると減少しています。

よく見ると、PV の追従に遅れがあるばかりでなく、目標である SP を行き過ぎてしまう（オーバーシュートといいます）ときがあります。これは PID パラメータとして初期値を使ったためです。制御対象に合わせてパラメータをチューニングすると、応答はずっと良くなります。その方法は実機を使った制御のところで説明します。

1800 秒後から連続して PV のステータスが BAD になると、MV を一定にして制御は MANUAL モードになります。これは直前の MV を維持することが、たとえ温度が分からないときでも、比較的安全な運転につながるからです。BAD がさらに続くような事態になると、MV を最も安全な（温度コントローラの場合は加熱をやめる）状態にすることが求められるのですが、そこまでは実装していません。

小型 CPU カード

Linux が走る小型の CPU カードというのは、Raspberry Pi が最初というわけではありません。

まだ IoT という概念もなかったころのことです。Linux の専門家を目指していた若い同僚に誘われて（おだてられて）、アメリカで発売されたばかりの、ノート PC 用メモリに使われる、SO-DIMM モジュールと同型の CPU カードを購入しました。何枚かまとまらないと、日本から買うことができなかったからです。Linux を搭載できるという触れ込みだったのですが……。

メモリモジュール用のコネクタから有線 LAN や RS232C に繋ぐことから始めて、Linux のソースコードを入手し、PC にクロスコンパイラをインストールし、生成した OS イメージを不揮発メモリに RS232C 通信経由でダウンロードする必要がありました。PC と通信するところまでは行ったのですが、そこで挫折してしまいました。このカードで何をするかという目的があいまいだったから、あまり苦労する気にならなかったのです。Linux を走らせるだけなら、古い PC で十分だったので。同僚は、かなり楽しんだようです。

5.2.8HMI

手動操作と割り込みの発生を模擬して、HMI モジュールの動作を確認します。タイマーとキースイッチからの割り込みを模擬するため、キーボードまたはファイルから一文字のコマンドを得て、割り込みルーチンをコールしています。COMMAND_FILE にファイル名が定義されていれば、コマンドをそのファイルから取り出します。定義されていないと、キーボードからコマンドを入力します。

コマンドの一字目が"u"なら UP キーの押下、"d"なら DOWN キーの押下、"m"なら MODE キーの押下、"x"なら PID 制御の実行タイミングを意味しています。"-"は記録用のテキストをコンソールに出力します。

I2C バスに送るデータを表示させると画面が混雑するので、温度センサと液晶表示器のドライバーから I2C バスへのコマンドを LCD_SILENT を使って停止しています。

なお、この検証モジュールを使うと、PWM モジュールの実機によらない部分の検証もできてしまいます。

test_hmi.py

```
/* HMI ステートマシンを評価するためのタイミング生成
   2011/11/14 Chuji */

/* LCD と I2C バスの動作を止める */
#define LCD_SILENT
#define TEMP_SIMULATION

/* タイミング生成コマンドの一字目の定義 */
#define D_UP "u"  /* UP キーの押下 */
#define D_DOWN "d"  /* DOWN キーの押下 */
#define D_MODE "m"  /* MODE キーの押下 */
#define D_EX "x"  /* PID 制御の実行 */
#define D_COMMENT "-"  /* コメント */

#include "hmi.py"

hmi =  human_machine_interface()

#ifdef COMMAND_FILE
/* タイミングを生成するコマンドをファイルから得る */
f = open(COMMAND_FILE)
instructions = f.readlines()

for instruction in instructions:

#else  /* コマンドをキーボードから得る */
while True:
  try:
    instruction = raw_input("command: ")
#endif
    s = instruction[0]

    if s == D_UP:
      ret = hmi.state_machine(HMI_UP_SW)
      print "== UP KEY == Return: ", ret
      print
    elif s == D_DOWN:
      ret = hmi.state_machine(HMI_DOWN_SW)
      print "== DOWN KEY == Return: ", ret
      print
    elif s == D_MODE:
      ret = hmi.state_machine(HMI_MODE_SW)
```

```
      print "== MODE KEY == Return: ", ret
      print
    elif s == D_EX:
      print "==== BLOCK EXECUTION ===="
      hmi.block_execution()
      print
    elif s == D_COMMENT:
      print instruction
#ifndef COMMAND_FILE
  except KeyboardInterrupt:
    pass
#endif
```

まずは使用感を得るため、コンソールからの入力を行います。操作や割り込みによって、表示が変わっていくのが確認できます。

```
$ cpp test_hmi.py|python cleanfile.py >
python.py; python python.py

I2C byte write: 0x0 => 0x48 : 0x3
PWM:  0.0%  at initialization.

（一部省略）

+----------------+
|    Temp  SP  MV|
+----------------+
|O/S  0.0   0   0|
+----------------+

command: u
== UP KEY == Return:  8

command: m
+----------------+
|O/S Temp  SP  MV|
+----------------+
|O/S  0.0   0   0|
+----------------+

== MODE KEY == Return:  9

command: u
+----------------+
|MAN Temp  SP  MV|
+----------------+
|O/S  0.0   0   0|
+----------------+

== UP KEY == Return:  9

command: m
+----------------+
|    Temp  SP  MV|
+----------------+
|O/S  0.0   0   0|
+----------------+

+----------------+
|    Temp  SP @MV|
+----------------+
|O/S  0.0   0   0|
+----------------+

== MODE KEY == Return:  9

command: x
==== BLOCK EXECUTION ====
PWM:  0.0 %
Status indicator: 0
PWM:  0.0 %
+----------------+
|    Temp  SP @MV|
+----------------+
|MAN  0.0   0   0|
+----------------+

+----------------+
|    Temp  SP @MV|
+----------------+
|MAN 59.7   0   0|
+----------------+

+----------------+
|    Temp  SP @MV|
+----------------+
|MAN 59.7   0   0|
+----------------+

+----------------+
|    Temp  SP @MV|
+----------------+
|MAN 59.7  60   0|
+----------------+


command:
```

HMI は動作が複雑で、操作と割り込みの順番により動作が変わるので、いちいちコマンドを与えるのは面倒だし、誤操作や入力忘れが出やすいので、すべての検証操作をファイルに入れて与えます。入力ファイル hmi_test_sequence は行数が多い（200 行近い）のでここに掲載するのは止めておきます。

コマンド入力と結果の最初の部分を以下に示します。入力ファイル名は\"で囲っておきます。

```
$ cpp -DCOMMAND_FILE=\"hmi_test_sequence\"
test_hmi.py|python cleanfile.py > python.py; python
python.py

I2C byte write: 0x0 => 0x48 : 0x3
PWM:  0.0%  at initialization.

（途中省略）

--- test sequence for hmi state machine

==== BLOCK EXECUTION ====
PWM:  0.0 %
Status indicator: 0
PWM:  0.0 %

（途中省略）

+----------------+
|    Temp  SP  MV|
+----------------+
|O/S 59.7  60   0|
+----------------+

（以下省略）
```

結果は非常に長くなるので、ファイル hmi_result にリダイレクトしておきます。

```
$ python python.py >hmi_result
```

結果ファイル hmi_result は 4000 行近いファイルです。ここにはファイルの内容は表示しませんが、ファイルを読みながら HMI の動作を確認していきます。といっても全部を読む必要はありません。== UP KEY ==などとキー操作を示してある直前の表示イメージが、その操作の結果なので、ステートマシンどおりか確認するだけで済みます。

PWM モジュールは、渡ってきた PWM パルス幅とステータスを表示していることが確認できます。

5.2.9キースイッチ操作

switch.py でのキースイッチ操作を模擬して、割り込み処理を直接実行します。コマンドをキーボードまたはファイルから得るところは test_switch.py と同様です。この検証では、スイッチの押下ばかりでなく、長押しも確認しています。

test_switch.py

```
/* 割り込み処理の評価
   2015/1/11 Chuji */

/* キーボード入力からタイミングを作る：
   0: タイマー割り込みを模擬する
   1: UP キーの押下を模擬する
   2: DOWN キーの押下を模擬する
   3: MODE キーの押下を模擬する
   -: キーを放したことを模擬する
   x: 何もしない（コメントが付けられる）
*/

/* LCD と I2C バスの動作を止める */
#define LCD_SILENT
#define TEMP_SIMULATION

#include "switch.py"

/* コマンドとキー名称の辞書 */
key_dic ={"3":KEYS_MODE_SW, "1":KEYS_UP_SW,
"2":KEYS_DOWN_SW}

hw =  switches()

print "start loop!"

#ifdef INT_SEQ  /* コマンドをファイルから得る */
f = open(INT_SEQ)
instructions = f.readlines()

for action in instructions:
#else  /* コマンドをキーボードから得る */
while True:
  try:
    action = raw_input("Command: ")
#endif
    if action[0] == "0":
      hw.check_timer()
    elif action[0] == "-":
      hw.release(key_dic[action[1]])
    elif action[0] != "x":
      hw.push(key_dic[action[0]])

#ifdef INT_SEQ
print "end of file"
#else
  except KeyboardInterrupt:
    pass
#endif
```

次のコマンドで実行し、キーボードから割り込みを模擬するデータ（0~3、-、x）を与えれば、割り込みルーチンの検証ができます。タイマー割り込みは 200ms に一回起こるように設定しているので、PID 制御が実行されるのはタイマー割り込み 5 回ごとだということを忘れないでください。

```
$ cpp test_switch.py|python cleanfile.py
>python.py ; python python.py
```

割り込みに対する評価は複雑なので、いちいち手入力していては面倒だし、まちがいがおこりやすいので、割り込みシーケンスをファイルで与えるようにします。ファイル swich_test_sequence は 100 行以上あるので、内容は掲載していません。

```
$ cpp -DINT_SEQ=\'switch_test_sequence\'
test_switch.py|python cleanfile.py >python.py ;
python python.py >switch_result
```

実行が終わったら、出力ファイル switch_result の内容を確認します。ファイルは 900 行近いので掲載はしていませんが、割り込みごとに液晶表示画面の表示変化を調べていきます。長押ししていると、200ms ごとにキー押下が続くように見える点が肝心なチェックポイントです。

小型 CPU カード（その 2）

秋葉原の秋月電子通商からは、H8 という CPU を使ったボードが発売されています。かつて温度コントローラをつくろうという意図で、このボードを手に入れました。温度センサはアナログ入力でしたが、PWM や HMI の基本はこの頃に設計しています。

C 言語は使えたものの、RC232C 経由でダウンロードとデバグを行う点は、前と一緒です。最悪だったのは、ソフトウエアの設計です。ボトムアップ型の設計をしてしまったため、機能間のインターフェースがめちゃめちゃで、デバグも検証も思うように行きません。保守性が悪いので、長く使うことはできませんでした。

Raspberry Pi に出会って、ようやく思いどおりのものができました。すぐに Linux が立ち上げられるし、GPIO や I2C バスで周辺回路とつなぐのも簡単です。なにより、トップダウン型の設計をしたのと、PC の Linux 環境で検証できたのが、実現を容易にしました。実機では、配線間違いが一か所あったのと、cpp にオプションを付け忘れたことがあったくらいで、ほとんど苦労なく動かせました。設計内容を文書にしていたので、改造や拡張のときに困ることもありませんでした。

6　総合検証

ここからは、ハードウエアとソフトウエアモジュールを組み合わせ、温度コントローラとしての機能を総合的に検証していきます。

6.1　ハードウエア動作確認

前章でソフトウエアモジュールの検証が済んだので、その機能を使ってハードウエアの動作を順番に確認していきます。Raspberry Pi 単体で動作させることから始めて、ハードウエアをひとつずつ接続していきます。

動作を確認する順番は、必ずしも以下で示すとおりである必要はありません。組上がったハードウエアから評価していってもいいでしょう。ただし、以下の三つの順番だけは守ってください。これは、前段のハードウエアの動作を確認できないと、後段を動かすことができない（動作不良があっても、どちらか分からない）からです。

- 最初にインターフェースユニットの動作を確認する（WiFi を除く）
- PWM の確認をしてから半導体リレーを動かす
- 温度計で I2C バスの動作を確認してから、液晶表示器を繋ぐ

私は、ホームディレクトリである/home/chuji の下に controller というディレクトリを作って、ここで作業することにしました。PC から Raspberry Pi に SSH 接続し、そのコンソール画面から検証を行います。

```
$ mkdir controller
```

次に PC から Samba で Raspberry Pi に接続し、PC 上で検証の済んだソフトウエアモジュールを、Raspberry Pi に転送しておきます。include にあるヘッダーファイルも同様です。プログラムファイルは controller に、ヘッダーファイルは controller/include に転送します。

ハードウエア動作テスト専用のソフトウエアファイルには、htest_xxx.py という名前を付けてあります。ここで xxx は試験するハードウエアの名前です。事前に PC で作成しておいてから、Raspberry Pi に転送しておくといいです。ソフトウエアモジュール検証で使ったものを#define で切り替えて使うケースもあるので、test_xxx.py も全て転送しておきましょう。

以降の操作は、PC から Raspberry Pi に SSH 接続し、そのコンソール画面から行います。最初は作業ディレクトリに移ることから始めます。

```
$ cd controller
```

検証をひとつ進めるごとに、次のハードウエアを接続します。そのたびに、いったん Raspberry Pi をシャットダウンし、電源を切ってからハードウエアをつなぎ、再度電源投入と SSH 接続をすることを心がけてください。

6.2　WiFi ブロードキャストの確認

いささか唐突ですが、オプションであるレコーダー機能で使う、ネットワーク広報（WiFi ブロードキャスト）の検証から始めます。Raspberry Pi と AC アダプターがあればできることなので、他のハードウエアを接続していない、この時点で行うのがよいと思います。ネットワークレコーダーを実現する予定のない方は、この節は読み飛ばしても大丈夫です。

まず Raspberry Pi で、先にレコーダーの評価に使ったテストデータ発生プログラム gen-trend.py を走らせ、ネットワークにトレンドデータをパブリッシュします。

```
$ cpp -DNETWORK_RECORDER gen-trend.py | python
```

Raspberry Pi コンソールに正弦波のデータが一秒ごとに表示されていきます。このとき、同時に WiFi にも同じデータがパブリッシュされているはずです。

それを PC で受信してみます。PC のコンソールで、実装と検証を済ませたレコーダーを起動します。

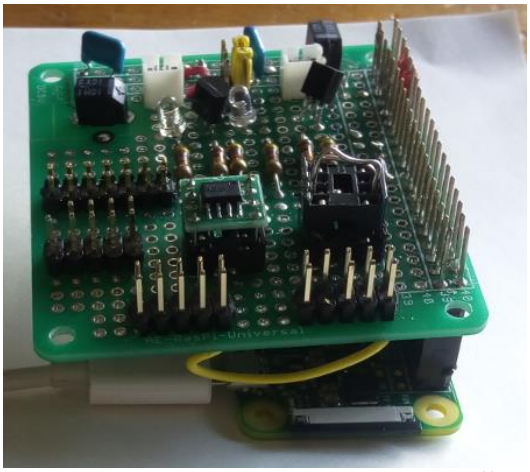
```
$ cpp -DSAVE_DATA recorder.py | python
```

コンソールに受信したデータが表示されるとともに、検証時に見たのと同じトレンド画面が表示されることを確認してください。コンソールに表示されるデータに飛びがない（各行の最初が一秒ごとの時刻なので、不連続になることがない）か、調べてください。たまに飛ぶくらいなら実用上の問題はありません。頻繁に起こるようなら、ノイズによる通信障害か、PC の負荷が高いことを疑ってください。

これ以後は、Raspberry Pi にハードウエアを繋いでいきます。

6.3 LED 駆動回路の確認

いったん Raspberry Pi をシャットダウンし、電源を切ったら、インターフェースユニットを接続します。Raspberry Pi の 40P ピンヘッダーにソケットを挿入していきます。この段階では、二枚のユニット同士が接触しないように注意すれば、机の絶縁物の上で試験しても構いません。



Raspberry Pi とインターフェースユニットを接続

LED ドライブは二種類あるので、それぞれのドライバーを使って動作させます。インターフェースユニットの回路図には、駆動トランジスタと LED の間にジャンパーを設けています。最終的に温度コントローラとして使うとき、内部の LED の点滅がじゃまになるなら、点灯を禁止するために設けたものです。ジャンパーを実装したときは、それが短絡するようになっていることを確認してから検証に入ってください。そうしないと、「点灯しない」ので、無駄に回路のチェックをする羽目になります。上の写真では、上方のピンヘッダーに黄色いショートバーが挿入されています。

半導体リレー駆動 PWM

PWM を駆動する検証モジュール htest_pwm.py は以下のとおりです。

htest_pwm.py

```
/* PWM 出力回路の動作テスト
    初版： 2016/11/9 Chuji
    最新版： 2016/11/9

PWM 出力パルス幅を 10%づつ変えていく
*/


#define BCM2835

#include "./include/use-time.h"
#include "pwm.py"

pwm = pulse_width_modulator(PWM_PORT)

for i in range (0, 11):
  print 10*i
  pwm.output(10*i)

  sleep(5)

GPIO.cleanup()
```

以下のように実行すると、インターフェースユニット上の LED が約 5 回点滅するごとに、点灯時間が長くなっていきます。最後のコマンドの前に sudo を入れるのを忘れないようにしてください。GPIO を使うにはルート権限（sudoer）が必要です。これを忘れると、Python のランタイムエラーが起こってしまいます。

```
$ cpp htest_pwm.py | sudo python
```

点滅する LED は、半導体リレーユニット駆動に使う側であることを確認してください。

もし LED が点灯しないときは、回路図とインターフェースを比較してください。ありがちな問題は、コネクタのピン番号間違い、トランジスタのエミッターを GND に接続していない、LED の方向が逆、ジャンパーの付け忘れなどです。

アラーム表示点灯

アラームを 3 秒ごとに点滅させて、ハードウエアの動作を確認します。検証モジュールは以下のとおりです。

htest_alarm.py

```
/* アラーム出力回路の動作テスト
    初版： 2016/11/9 Chuji
    最新版： 2016/11/9

3 秒ごとにアラームを点滅させる
*/

#define BCM2835

#include "./include/use-time.h"
#include "pwm.py"


alm = status_indicator(STATUS_PORT)
sleep(2)

for i in range (0, 5):
  print "CLEARED"
  alm.output(INDICATOR_OFF)
  sleep(3)
  print"ALARM"
  alm.output(INDICATOR_ON)
  sleep(3)

GPIO.cleanup()
```

実行方法は全く同じです。

```
$ cpp htest_alarm.py | sudo python
```

コンソールに CLEARED と表示されたらアラーム側の LED が消灯し、3 秒後にコンソールに ALARM と表示され LED が点灯します。3 秒後に消灯し、これを 5 回繰り返せば、動作確認は終わりです。

動作確認が終わったら、先に短絡させたジャンパーを取り除いても構いませんが、安全のため PWM 側は残しておいた方が良いです。半導体リレーの動作を目視することができるからです。

6.4 操作ユニットの確認

いったん Raspberry Pi をシャットダウンして電源を切ります。ここから先は、思わぬ短絡事故などを起こさないよう、ケースに入れて試験するのがいいです。各ハードウエアユニットを順番にコネクタで接続していきます。まず、操作ユニット（キースイッチと警報灯）を接続して試験します。

最初に、前項で確認したアラーム点灯試験をやり直します。今度は操作ユニットに組み込んだ警報灯が点滅することを確認します。

正しく点滅しないときは、コネクタやケーブルのピン番号と接続（接触不良も短絡もないこと）、LED の方向を確認してください。

次はキースイッチの試験ですが、全てのソフトウエアモジュールを動かすと、問題が起こったときに分析しにくいので、HMI 以降を代替モジュール hmi_stub.py で置き換えます。HMI の操作インターフェースである、state_machine と block_execution の処理を単なるコンソール表示に置き換えます。この置き換えは、HW_DEBUG が#define されているときに、switch.py のなかで#include するファイルを変えることで実現しています。

hmi_stub.py

```
/* キー操作評価に使う HMI ステートマシンの代替スタブ
      初版： 2017/2/25 Chuji
      最新版： 2017/2/25

押下したスイッチと PID 実行要求回数を表示する

*/

#include "./include/pid.h"
#include "./include/hmi.h"

class human_machine_interface:
  def __init__(self):
    self.counter = 0
    self.state = HMI_OPERATION
    print "HMI is initialized."

/* HMI ステートマシン */
  def state_machine(self, key):
    print "State machine received request on
Switch #", key
    return(self.state)

/* PID ブロック実行 */
  def block_execution (self):
    self.counter += 1
    print "Function Block execution #",
self.counter
```

キースイッチと割り込み処理の検証モジュール htest_switch.py は以下のとおりです。BCM2835 と HW_DEBUG を#define して、ハードウエアを動かしながらデバグしていきます。コンソールには、switch.py のキー割り込み情報とと、hmi_stub.py の操作情報が表示されます。

htest_switch.py

```
/* キースイッチの動作テスト*/

from time import sleep

#define BCM2835
#define HW_DEBUG

#include "switch.py"

RasberryPi = switches()

try:
  while True:
    sleep(0.1)
except KeyboardInterrupt:
  GPIO.cleanup()
  pass
```

結果を下に示します。PID 制御ブロックは 1 秒ごとに呼ばれています。最初に UP キー（24）を押下すると、ステートマシンに解釈要求が送られています。次に DOWN キー（25）、MODE キー（27）の操作を確認します。最後に DOWN キーを長押しすると、ステートマシンを 200ms ごとに呼び出していることが確認できます。キーを放すと呼出しが止まります。

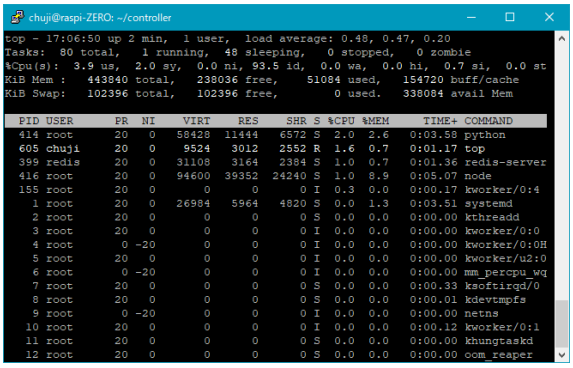
```
$ cpp htest_switch.py|sudo python
HMI is initialized.
Function Block execution # 1
Function Block execution # 2
Switch # 24 is pushed.
State machine received request on Switch # 24
Function Block execution # 3
Switch # 24 is released.
Function Block execution # 4
Function Block execution # 5
Switch # 25 is pushed.
State machine received request on Switch # 25
Switch # 25 is released.
Function Block execution # 6
Function Block execution # 7
Switch # 27 is pushed.
State machine received request on Switch # 27
Switch # 27 is released.
Function Block execution # 8
Function Block execution # 9
Function Block execution # 10
Switch # 25 is pushed.
State machine received request on Switch # 25
Function Block execution # 11
```

```
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
Function Block execution # 12
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
Function Block execution # 13
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
State machine received request on Switch # 25
Function Block execution # 14
State machine received request on Switch # 25
State machine received request on Switch # 25
Switch # 25 is released.
Function Block execution # 15
Function Block execution # 16
```

思ったとおりの動作にならないときは、回路図と実物を良く比較してください。よくある問題は、コネクタのピン番号間違い、操作ユニット内での VDD と GND の結線忘れなどです。

CPU 負荷はどのくらい？

温度コントローラはどのくらい Raspberry Pi ZERO に負荷をかけている（CPU 時間とメモリをどれだけ使う）のでしょうか？　Linux の top コマンドで確認してみました。SSH 接続した端末から起動すると、以下のような表示が出てきます。



CPU 時間のうち、一秒周期のコントローラ（Python）が 2%、Web サーバー（node）と redis サーバーが各 1%を使っています。メモリも同程度でした。このくらい遅い CPU でも、十分余裕があることが分かり、一安心です。

6.5　I2C バスと温度センサの確認

ここでは、I2C バスと温度センサの動作を同時に確認します。さらにコネクタと内部配線も検証対象になるので、慎重に行ってください。これらを別個に検証するには、オシロスコープなどの測定器が必要になるからです。

htest_temp.py

```
/* 温度センサの動作テスト
   2016/11/6 Chuji

温度を繰り返し読み取る
*/

#define REPEAT 200
#define BCM2835

#include "./include/use-time.h"
#include "thermo.py"

model = thermometer(TEMP_SENSOR)
for i in range (1, REPEAT):
  temp = model.read_temp()
  if temp.status == STATUS_GOOD:
    print "T =", temp.value, ":GOOD"
  else:
    print "T =", temp.value, ":BAD"
  sleep(1)
```

まずドライバーの動作が簡単な ADT7410A から始めます。この段階では、センサは放熱器に取り付けた状態で、エポキシ樹脂による封止はされていないと思います。短絡に注意しながら、次のコマンドを実行します。

```
$ cpp htest_temp.py | sudo python
```

ハードウエアが正しく組上がっていれば、室温と GOOD という表示が、一秒ごとに更新されるはずです。放熱器を手で握ってみてください。表示温度が少しずつ上がっていくはずです。今度は手を放してみてください。しばらくすると、表示温度がゆっくり室温まで下がっていきます。

もし BAD というステータスが表示されたら、I2C バスが正しく動いていないことを示しています。いったん電源を切り、回路図と実物をじっくり比較してください。もっともありそうな問題は、IC 周りの立体配線のイモはんだと短絡、IC やコネクタのピン番号の間違い、インターフェースボード上の結線やケース内配線の両端での接触不良と短絡です。

ADT7410A の動作が確認できたら、STTS751 の動作も確認しましょう。ガラス管に封入する前に行います。

```
$ cpp -DSTTS751 htest_temp.py | sudo python
```

今度も温度が表示されるようになったら、チップをつまんで、温度が変化することを確認します。

温度センサの動作が確認できたら、エポキシ樹脂（ADT7410A）やアルミナ（STTS751）で封止する加工を行ってください。封止が完了したら、もういちど同じ動作確認をして、工作中に回路を損傷していないか調べます。これで温度センサが使用可能な状態になりました。

6.6　液晶表示ユニットの確認

I2C バスの動作が確認できたら、液晶表示ユニットの動作を調べます。液晶表示ユニットのコネクタを接続したら、ソフトウエアモジュールの検証に使った test_lcd.py を、BCM2835 を#define して実行します。

```
$ cpp -DBCM2835 test_lcd.py |sudo python
```

上の行へ ABC と表示し、2 秒後に上書きされます。次に下の行の中央に Middle と表示し、2 秒後に dd のうしろに truncated と（実際には tranca より後ろは切り捨てられる）書き込まれます。次に、上の行の右端から Hello World!という文字列が左端まで流れていきます。

表示されないときは、回路図と実物を比較します。よくある問題は、I2C バスリピーター（PCA9515）周辺の結線間違い、液晶表示ユニットへのケーブルとコネクタのピン番号間違いなどです。

6.7　半導体リレーユニットの確認

最後に試験用恒温槽を使って、半導体リレーユニットの動作を確認します。

最初に試験用恒温槽の配線を確認しておきます。電球に繋がったプラグを、AC コンセントに差し込んで、電球が点灯することを確認します。

次にプラグを温度コントローラの電力供給コンセントに差し込みます。この時点では電球は消えたままです。

以前に PWM 出力回路の動作を調べた、htest_pwm.py を起動します。電球の温度が上がることと、点灯を直視すると、まぶしい場合があることに注意してください。

```
$ cpp htest_pwm.py | sudo python
```

点灯しないときは、回路図と配線を比較します。よくある問題は、インターフェースユニット上のコネクタのピン番号間違い、半導体リレーユニットのフォトカップラの極性の間違い、AC 回路の配線間違いなどです。

ここまででハードウエアの動作確認は完了しました。ケースの蓋を閉め、接触事故を防ぎましょう。

低温調理の化学

食肉のタンパク質の成分の話です。アクチンは球状のタンパク分子で、多数の分子が、真珠のネックレスを二本撚りあわせた紐の型に結合（重合）しています。もう一つの重要なタンパク分子はミオシンといって、カマキリのような二本の腕（頭という人も多い）を持っています。この腕がアクチンの紐を手繰るような力を出すことで、筋肉が収縮すると考えられています。腱のような部分では、コラーゲンという丈夫なタンパク質が、筋肉の構造を支持しています。筋肉タンパク量の半分以上はミオシンです。

肉を加熱すると、タンパク分子の立体構造が壊れ（変性といいます）ます。このとき、筋肉内の水分が出てしまうと、パサパサの固い肉になってしまいます。特にアクチンが固くなりやすい。シチューのように水の多い料理では、食感は悪くありませんが、焼いたときには固くならない工夫がほしいところです。

この変性がおこる温度はタンパク分子ごとに異なり、だいたいミオシンで 50～60℃、コラーゲンで 68℃以上、アクチンで 66～73℃です。ミオシンとコラーゲンだけが変性するのが望ましいので、アクチンの変性温度以下で調理するのが、低温調理です。コラーゲンの変性温度は高いのですが、幸い時間をかければ低温でも変性が進みます。何時間も調理するのはこのためです。ローストビーフなどに向いています。

調理温度は微妙です。細菌が繁殖する温度と死滅する温度の境目に近いからです。65℃以上でほとんどの細菌が死滅するといわれていますが、全部ではないし、毒素が残るものもあります。鮮度の良い食材を選ぶとともに、繁殖が進む温度に長く置かない（加熱後は水で冷ますなど）工夫が必要です。

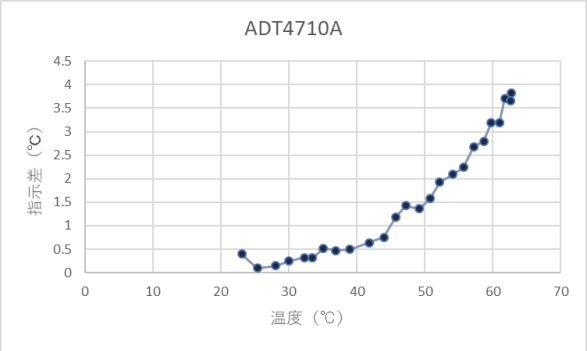
6.8　温度センサの較正

ハードウエア動作試験の最後として、自作した温度センサの表示がどの程度正しいか調べます。すでに、手で握ったら表示温度が上がるのは確認しています。


温度センサの較正

温度センサと市販のガラス管温度計を接触させて、テープで固定したものを、試験用恒温槽に入れます。次章のように温度コントローラを起動したら、MAN モードのまま温度を表示させ、ガラス管温度計の指示と比較します。MV を変え、温度が安定したら、両方の温度指示値を読み取っていきます。ガラス管温度計の指示は、目分量で 0.1℃まで読み取ります。

ガラス管温度計を基準に、自作温度センサの指示値と基準値との差をプロットすると、次の図のようになりました。



温度センサ較正データ

温度が上がるにしたがって、温度センサの方が高めの指示をすることが分かります。でもこれは、温度センサの精度が悪いからではなく、ガラス管温度計の使いかたに問題があったからです。

左の写真を見ると、ガラス管温度計の指示を読み取るため、測温液（アルコール）の上端が試験用恒温槽の外に出ています。実は、この温度計は、測温液が上端まで同じ温度になっていることを前提にしているのです。水温を測る時には、測温液部分をすべて水没させなければなりません。でもそうすると、試験用恒温槽の温度は読み取れませんよね。外に出ている部分の測温液は、槽内の部分より室温に近くなるので、温度が低めに測定されてしまっていたのです。この差は、温度が高くなるほど（恒温槽の外にある測定液の体積が大きくなるほど）、大きくなります。

ある人の実験によると、水温が 70℃のとき、測温液溜まりだけを水没させた状態での指示は、全体を水没させた状態より約 4℃低く表示したそうです。これを考慮すると、両者の温度測定値はほぼ一致し、指示差は±1℃以内だと解釈して良さそうです。液温センサについても同様の結果になりました。

さまざまな温度センサ

被測定物体に直接触れる温度センサの主なもの（電気的なもの）を以下にまとめます。

- 測温抵抗体： 白金線などの抵抗値が、ほぼ温度に比例することを利用
- 熱電対： 二種類の金属を接触させると、接点の温度差に依存した起電力が発生する
- サーミスタ： 半導体の抵抗が温度で変わることを利用
- IC センサ： トランジスタのベース・エミッタ間電圧の温度依存性を利用。

工業分野では測温抵抗体や熱電対が多く使用され、電子回路には安価なサーミスタが使われてきました。IC 温度センサは、精度や直線性が良く、周辺回路もほとんど不要なので、非常に便利です。いっぽう測定温度範囲が狭く、プラスチックパッケージのせいで熱接触や電気絶縁が難しい欠点があります。今回の温度コントローラでは温度範囲の問題がないので、熱接触と電気絶縁を確保するための工作を行ってから使いました。

7　温度制御試験と調整

温度コントローラのハードウエアとソフトウエアの検証が済んだので、実際に温度制御を行ってみます。

7.1　試験条件

試験用恒温槽を使って、牛乳パック内の空気温度を制御することにします。電球に牛乳パックを被せ、上端（実際には牛乳パックの底）に開けた穴から、空気用温度センサを入れます。ゼムクリップなどを使って、上端近くに吊るしておきましょう。

恒温槽の電源プラグを温度コントローラの出力コンセントにつなぎ、温度コントローラの電源を入れます。電球を点滅させるのは目に良くないので、PC の操作する姿勢では、恒温槽が直接見えないよう、物陰に隠しておきましょう。温度コントローラ自体は、操作できる場所に置きます。

7.2　温度コントローラの起動

PC から Raspberry Pi に SSH 接続します。controller ディレクトリに移ったら、以下のコマンドを実行します。

```
$ cpp controller.py |python cleanfile.py | sudo python
```

BAD ステータスの警告灯は一瞬で消え、液晶表示器に写真のようなデータが表示されます。モードは O/S、Temp 欄に室温が表示されます。SP は（小数点以下を四捨五入した）温度を、MV は 0 を示しています。

温度コントローラ起動表示

この状態では温度指示計になっています。▲キーや▼キーを押しても、何も起こりません。

7.2.1手動操作

MODE キーを押下すると、液晶表示器の左上にも O/S と表示されます。これは制御モードの設定目標を表しています。▲キーを押すたびに、MAN⇒AUT⇒O/S と変化します。表示を MAN にしたところで MODE キーを押すと、左上の表示が消え、すぐに左下の実効モードが MAN に変わります。

右上の表示が@MV になっているはずです。これは MAN モードでは、MV を手動設定できることを示しています。試しに▲キーを押すと、MV の指示値が 1 ずつ増加します。▼キーで減少します。電球が短時間点灯するので、覗いてみてください。

しばらくすると、温度表示がさっきより上昇していることに気が付きます。電球から発生した熱で、牛乳パック内の温度が上がったのです。

7.2.2トレンドの表示

この後はトレンド画面で見ると制御状態を把握しやすいので、オプションのレコーダーを実装していたら、PC 上に表示してみてください。

```
$ cpp recorder.py | python
```

7.2.3自動制御

温度表示が変化しなくなったら、自動制御を始めてみましょう。さっきと同様に MODE キーを押してモード設定に移り、▲キーで左上の表示を AUT にします。もう一度 MODE キーを押すと、左上の表示が消え、右下の実効モードが AUT になります。

MV の前にあった@が消え、こんどは SP の前に出てきました。SP の値は、直前の温度を表示しています。▼キーを押して、SP を減少させてみましょう。MV が 0 になって、電球が消えてしまうのが分かります。恒温槽の空気が、自然冷却で下がっていきます。PC のトレンド画面では、緑の SP が減少し、赤の PV がゆっくりと SP に近づいていくのが見えます。

PV が SP を下回ると、ふたたび MV が増加し、加熱を始めます。両者が同じになるのが望ましいのですが、まだチューニングができていないので、オーバーシュートが発生すると思います。

両者がほぼ一致したら、今度は SP を増やしてみましょう。再び追従動作が始まります。

7.3　チューニング

温度コントローラが動作を始めたら、PID 制御パラメータのチューニングをしてみましょう。

制御対象には、熱容量、伝熱と冷却のメカニズムなどによる固有の（主に遅れ）特性があります。それが原因で応答が悪くなったり、温度が設定値を越えてしまうオーバーシュートを起こしたりすることがあります。対象に合わせて PID パラメータを調整（チューニング）すると、制御への応答が改善されます。

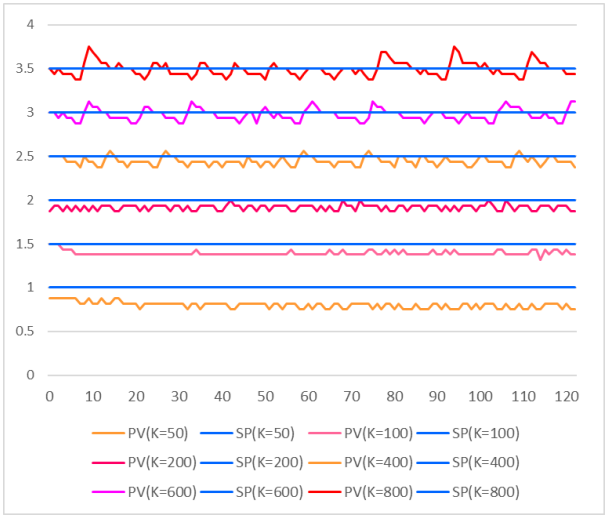
現代では、シミュレーションを活用してチューニングを実行したり、コントローラに自動チューニング機能を実装したりするのが一般的です。制御対象の運転中に試行錯誤したりするのは、危険だったり、生産物の品質を悪くしかねないからです。しかし、ここで作っているのは、それほど温度も高くならず、制御対象の応答も比較的単純なものです。だから、「古典的」とさえ言われる、実機を動作させてのチューニングをします。この手法は経験則に基づくもので、単純な制御対象に対して多くの実績があります。

代表的なチューニング法には、加熱量をステップ的に変えた時の応答を調べる「ステップ応答法」と、P 制御が不安定になる増幅率と振動周期からパラメータを決める「限界感度法」があります。ここでは、計算が分かりやすいことから、後者を使います。

7.3.1手順

試験用恒温槽のチューニングを、限界感度法で行ってみます。まず、P 制御にするため、I と D のパラメータを 0 に設定します。次に P パラメータを大きくしながら、制御状態を調べます。ローカル、またはネットワークレコーダーに記録しながら行うと分かりやすいです。

制御が本当に安定しているか確認するため、設定値をステップ的に変えて、その後の動作を見てみましょう。結果を次の図に示します。



図では、P パラメータ（図では K と表記）が下から 50、100、200、400、600、800 の時の動作です。室温より 5℃くらい高い設定値で制御しているので、加熱（温度上昇）の方が冷却（温度下降）より早い、三角波のような状態になります。この結果から、不安定になる P パラメータ K_c は 400～600、その振動周期 T_c は 10～12 秒と読み取れます。

7.3.2パラメータの決定

古典制御の経験則（ジーグラとニコルスの方法といいます）によると、この時 PID パラメータの値は、以下の表のとおりにするのが良いとされています。

制御方式	P パラメータ	I パラメータ	D パラメータ
P 制御	$0.5K_c$	0	0
PI 制御	$0.45K_c$	$0.83T_c$	0
PID 制御	$0.6K_c$	$0.5T_c$	$0.125T_c$

ジーグラ・ニコルス法

上の実験結果からパラメータを決めると、下表のようになります。P 制御は行わないので、省いてあります。

制御方式	P パラメータ	I パラメータ	D パラメータ
PI 制御	225	9	0
PID 制御	300	6	2

試験用恒温槽のチューニング結果

同じように、私が使っている燻製室と低温調理器のパラメータを求めてみました。下表に PID 制御の場合のパラメータをまとめてあります

制御対象	P パラメータ	I パラメータ	D パラメータ
燻製室	6	240	30
温熱調理器	35	250	62
試験用恒温槽	300	6	2
（デフォルト）	2	10	0

各種装置の PID パラメータ（PID 制御）

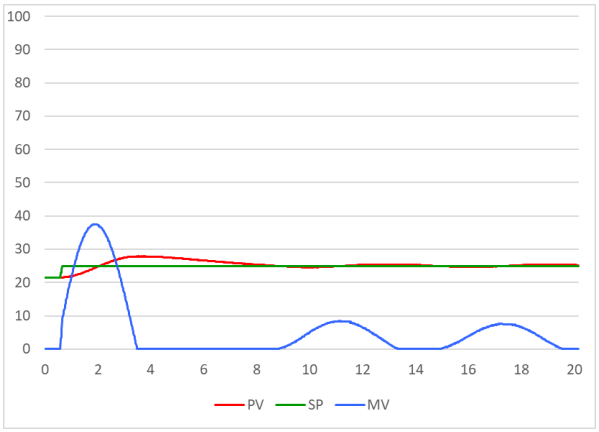
表でデフォルトとあるのは、pid.h で定義した値で、cpp の-D オプションで指定しないときは、この値が使われます。当たらずとも遠からずといったところですね。

7.3.3動作確認

では実際にこのパラメータを使って、試験用恒温槽の温度制御を行ってみましょう

デフォルトパラメータの場合

pid.h で定義しているデフォルトパラメータで制御した結果を下図に示します。設定値（SP：緑で表示）を上げたときに、オーバーシュート（温度（PV：赤で表示）が設定値を越えて上がること）が発生しているばかりか、10 分以上経過しても設定値の上下に振動しているのが分かります。あまり安定した制御とは言えません。



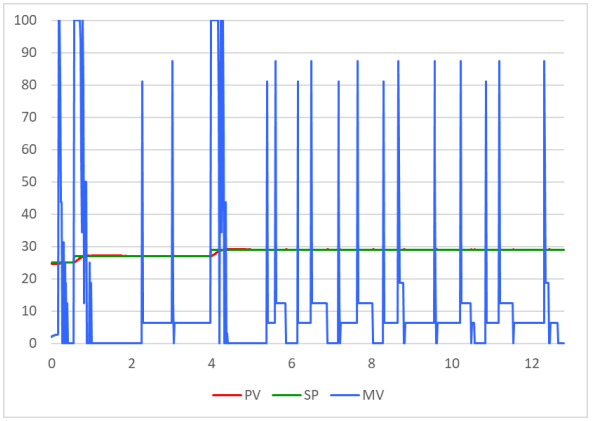
デフォルトパラメータによる PID 制御動作

いっぽう、操作量（MV：青で表示）の変化は滑らかです。加熱を重油バーナーなどで行っているときは、燃料の供給を調整するバルブがゆっくり動き、損耗が少なくなるので、あえてこういう制御を選ぶこともあります。

いまは、電気ヒーターのオンオフで加熱を調整しているので、もっと応答のよい制御が可能です。チューニングはそのために行いました。

PID 制御の場合

チューニングしたパラメータを使って PID 制御をしたデータを次の図に示します。操作量（MV）は目いっぱい振れていますが、応答性はずっと良く、温度（PV）はほとんど設定値（SP）と重なっています。設定値を上げたときに、追従に遅れが見られますが、これはヒーターを全力運転したときの加熱限界のせいなので、ヒーターを変えない限り改善できません。

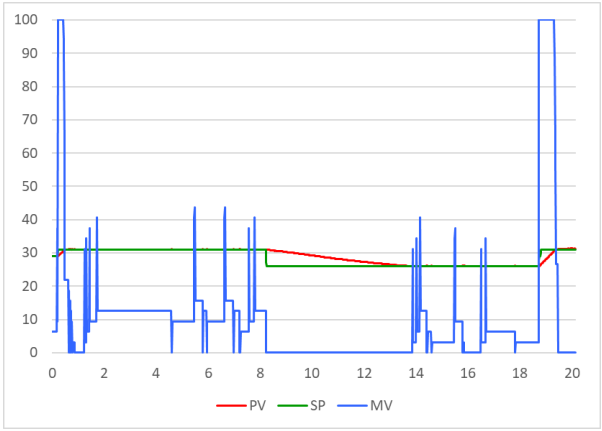


チューニングした PID 制御

操作量の平均値は、設定値によって変わります。これは、試験用恒温槽から逃げる熱量と加熱量が釣り合っていることを示しています。周囲の気流などの影響で冷え方が変わり、わずかに温度が揺らいだのを補償するように、操作量がスパイク状に変化しています。電気ヒーターを使ったから実現できた制御です。

PI 制御の場合

こんどは PI 制御の結果を見てみます。微分項がないので、操作量（MV）の急な変化は少なくなっています。応答性は PID 制御とあまり変わりません。設定値（SP）を下げた時の遅れが目立ちますが、これは自然冷却に頼っているせいで、制御パラメータによらず起こることです。



チューニングした PI 制御

8　機能拡張（Web インターフェース）

ここまでで温度コントローラの製作と評価は済んだのですが、機能的にちょっと物足りなく感じました。せっかく WiFi 経由で動作の記録を見ることができるのに、温度コントローラのキー操作をするには、温度コントローラの前まで行かなければなりません。燻製を作る時に燻煙室は屋外に置いています。そうしないと屋内に煙が充満してしまうからです。温度コントローラは燻煙室のヒーターのそばに置く必要があるので、とうぜん屋外にあります。いちいち屋外に出ていくのはおっくうなので、室内から操作できるように機能拡張しました。

私ほど怠惰ではない人や、とりあえず温度コントローラを実現したいだけの人は、この章を飛ばしても構いません。あくまで機能拡張なので。

機能拡張は、温度コントローラの前面にある液晶表示器と 3 つのキーが、ブラウザから見える／操作できるようにすることで実現します。欲張れば表示や操作を増やせるのでしょうが、ここでは簡単なものにとどめました。

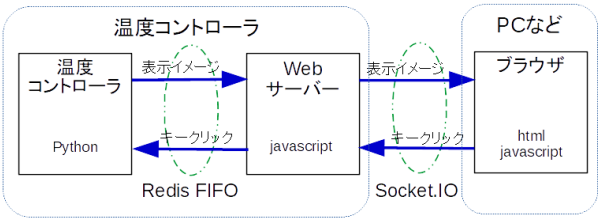
JavaScript や HTML の詳しい説明は、この本の目的に含まれないので省略し、何を実現しているかということに絞って説明します。

8.1　Web サーバーの設計

温度コントローラの各ソフトウエアモジュールの構造を見れば、Web インターフェース機能は、

1. display.py が（もともとはシミュレーション用に用意した）液晶表示イメージをそのままブラウザに送る
2. ブラウザのクリックを、switch.py の押下キーバッファに代入する

ことで実現できることが分かります。Web サーバーはそれを中継する機能だけで十分です。Web サーバーの機能を下図に示します。



8.1.1ブラウザとの通信

Web サーバーとブラウザとの通信は、JavaScript の通信 Socket.IO で行います。Socket.IO は通信の実装を詳しく知らなくても使えるので便利です。多くの機能がありますが、ここで使うのは事象（イベント）だけです。Web サーバーとブラウザがイベントを通知しあい、そこに情報を載せます。使用するイベントは下表に示す 5 種類です。

ブラウザから Web サーバーへのイベント	
IO_EVENT_UP	UP ボタンをクリックした
IO_EVENT_DOWN	DOWN ボタンをクリックした
IO_EVENT_MODE	MODE ボタンをクリックした
Web サーバーからブラウザへのイベント（データが付属する）	
IO_EVENT_LCD_1	1 行目の表示を更新した
IO_EVENT_LCD_2	1 行目の表示を更新した

普通の Web サーバーではポート番号に 80 あるいは 8080 を使いますが、ここではプライベートポート（49152 から 65535 の範囲）50000 を使うようにしました。ブラウザから接続要求があると、指定したディレクトリにある index.html というファイルを送り返します。

JavaScript（Node.js）と Socket.IO は、ハードウエアの準備の章でインストールしています。まだインストールを実施していないときは、前に戻ってインストールしてください。

8.1.2温度コントローラとの通信

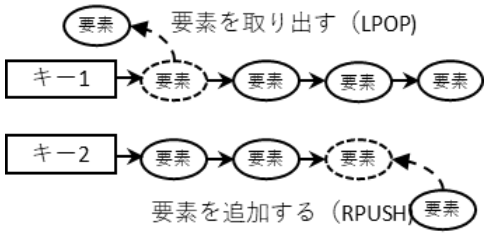
Web サーバーと温度コントローラとの装置内通信には、さまざまな方法があります。例をあげると、

- シグナル（非同期イベント）
- パイプ（3.2.2 節参照）
- キュー（FIFO（First-in-first-out））
- セマフォ（排他制御に使う）
- ソケット（プロセス間通信）
- 共有メモリ（双方が同じメモリにアクセス）

などが Linux のカーネルでサポートされています。

今回は、プロセス間の親子関係やプロセス ID を意識しないで使えて、かつ高速であることから、カーネルではなく、メモリ上のデータベース Redis を FIFO として利用しました。Redis は、動作の早いデ

ータベースとして知られていますが、FIFO はその機能の一部を使って実現できます。



キー付きリストによる FIFO 実現

上の図のように、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が停止（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。

FIFO は次の名前（キー）を持つ 3 本を使います。

Web サーバーから温度コントローラへの FIFO	
REDIS_KEY	クリックされたキーを表す ID
温度コントローラから Web サーバーへの FIFO	
REDIS_LINE_1	1 行目への表示データ
REDIS_LINE_2	2 行目への表示データ

Socket.IO では、Web サーバーとブラウザの間に確立させる相互通信チャンネル（ソケット）は一つだけなので、それに与えるイベントで操作や表示行を区別しました。一方、Redis は FIFO を何本でも使えるので、押下されたキーの名前（ID）や表示データをそのまま送れます。空の FIFO を読み出そうとすると、そこで待ちに入り、FIFO にデータが渡された段階で再開することができます。

しかし Python では、待ちに入るとプログラムの実行がブロックされてしまうので、扱える受信 FIFO は一つだけです。JavaScript では、受信した結果を処理する関数を与えるだけなので、受信 FIFO を二つにしても、そこでプログラムの実行が止まることはありません。

Redis は、ハードウエアの準備の章でインストールしています。まだインストールを実施していないときは、前に戻ってインストールしてください。

定義ファイル

Socket.IO と Redis で使う定義をヘッダーファイルにまとめました。定義だけなので、Web サーバーだけでなく、ブラウザ用 HTML ファイルと温度コントローラで共通して使えます。使用言語が変わっても、通信の約束事を一つのファイルで定義できるのが、この手法のいちばん強力なところです。

重要：JS_REQ_SOCKET_IO と JS_REQ_REDIS はそれぞれ Socket.IO と Redis がインストールされたディレクトリです。インストール方法によっては、違うディレクトリになる場合があるので、実機で確認しておいてください。JS_HTML_FILE は HTML ファイルのあるディレクトリです。これも実機のディレクトリに変更してください。URL_SOCKET_IO には温度コントローラの IP アドレスを使ってください。ポート番号 50000 は、そのままでも構いません。

remote_hmi.h

```
/* WiFi 経由のリモート操作用定義
  初版： 2017/6/7 Chuji
  最新版：2019/3/17 ...共通定義のみ抽出
*/

#ifndef __REMOTE

/* この定義は Python、html、Javascript のソースコード共用 */

/* Javascript、html 共用 SOCKET.IO 定義*/
/*  IP アドレスは実環境に合わせること */
#define URL_SOCKET_IO
'http://192.168.15.17:50000'
#define PORT_SOCKET_IO  50000

/* Javascript で記述した Web サーバー用 */

#define JS_FORK       'child_process'
#define JS_REQ_HTTP     'http'
#define JS_REQ_SOCKET_IO
'/usr/local/lib/node_modules/socket.io'
#define JS_REQ_FILE_SYSTEM  'fs'
#define JS_REQ_REDIS
'/usr/local/lib/node_modules/redis'

#define JS_HTTP_STATUS_OK  200
#define JS_CHAR_UTF8     'utf-8'
#define JS_HTTP_HEADER  {'Content-Type' :
'text/html'}

#define JS_HTML_FILE
'/home/chuji/controller/index.html'

/* Web サーバーと html に埋め込まれる Javascript 共用の Socket.io イベント */

#define IO_EVENT_CONNECT  'connection'  /* サーバー */
#define IO_EVENT_CONNECTED  "connect"  /* クライアント */
#define IO_EVENT_DISCONNECT  'disconnect'

#define IO_EVENT_UP  "UP KEY"
#define IO_EVENT_DOWN  "DOWN KEY"
#define IO_EVENT_MODE  "MODE KEY"

#define IO_EVENT_LCD_1  "lcd 1"
#define IO_EVENT_LCD_2  "lcd 2"
```

```
/* html ページのデザイン */

#define WEB_LCD_COLOR  size = "7" color="navy"
#define WEB_TITLE_FONT  size = "4" color="red"
#define WEB_TITLE  Raspberry Pi Temperature
Controller

#define WEB_STYLE_UP  "color:white; background-
color:blue; width:60px; height:30px; margin-
right:auto; margin-left:auto;"

#define WEB_STYLE_DOWN  "color:white;
background-color:blue; width:60px; height:30px;
margin-right:auto; margin-left:30px;"

#define WEB_STYLE_MODE  "color:yellow;
background-color:red; width:60px; height:30px;
font-weight:bold; margin-right:auto; margin-
left:30px;"

#define WEB_ID_LCD_1  IO_EVENT_LCD_1
#define WEB_ID_LCD_2  IO_EVENT_LCD_2
#define WEB_NO_DATA  ""

#define WEB_MESSAGE_RAW  "system message"
#define WEB_MESSAGE_RAW2  "Operation message"

/* ボタンの定義 */

  /* ボタンのトップ表示 */
#define SYMBOL_UP  "▲"
#define SYMBOL_DOWN  "▼"
#define SYMBOL_MODE  "Mode"

/* Redis FIFO の定義（Web サーバーと Python） */

/* Redis インターフェース */
#define REDIS_SERVER  host = 'localhost'
#define REDIS_PORT  port = 6379

/* データベース識別子 */
#define REDIS_FIFO  db=0 /* 内部通信用 FIFO */

/* Redis のコマンドオプション */
#define REDIS_ALL_ENTRIES  0 /* lrem コマンドです
べてをクリア */
#define REDIS_LAST_ENTRY  -1 /* リストの最後の要素
を指定 */

/* Javascript と Python で共通して使う Redis キー */
#define REDIS_KEY  "key in"
#define REDIS_LINE_1  IO_EVENT_LCD_1
#define REDIS_LINE_2  IO_EVENT_LCD_2

/* Web サーバーからの Redis メッセージ */
#define REDIS_UP  IO_EVENT_UP
#define REDIS_DOWN  IO_EVENT_DOWN
#define REDIS_MODE  IO_EVENT_MODE

/* Python 固有の定義 */

/* FIFO が空のときに読み込もうとするプログラムをブロック
*/
#define REDIS_NO_TIMEOUT  0

/* REDIS の BLPOP コール時に戻される配列の定義 */
#define REDIS_ID  0  /* キーID が戻される */
#define REDIS_DATA  1  /* FIFO から取り出されたデー
タ */

#define __REMOTE

#endif
```

Python から Redis を使うための手続きを redis_for_python.py にまとめました。実際に記述されているのは定義だけです。

DEBUG_REDIS を#define して使うと、Redis を使う関数 open_FIFO や rpush と blpop をコンソール画面とキーボードでシミュレートします。今回は使っていない set と get も用意して、他の用途でも使えるように考慮してあります。

redis_for_python.py

```
/* Python 用 REDIS データベース
  初版： 2018/2/14 Chuji
  最新版： 2019/3/17 ... Remoto_HMI との重複解消
*/

#ifndef __REDIS

/* javascript との共通定義取り込み */
#include "./include/remote_hmi.h"

import redis

/* Redis API の取得 */

#ifdef DEBUG_REDIS  /* Python デバグ用スタブ */
class open_FIFO:
  def __init__(self):
    self.dummy=1
  def set(key, msg):
    print "Put @key= "+key+" data= "+msg
  def rpush(key, msg):
    print "RPush @key= "+key+" data= "+msg
  def get(key):
    return raw_input("Get data from @key=?
"+key+" >> ")
  def blpop(key):
    return raw_input("Blpop @key= "+key+" ?>> ")

#else  /* 本番用 Redis 使用宣言 */

import redis

#define REDIS_API(x)
redis.StrictRedis(REDIS_SERVER, REDIS_PORT, x)
#define open_FIFO()  REDIS_API(REDIS_FIFO)

#endif

/* LCD 表示ラインキー */
Redis_lines = [REDIS_LINE_1, REDIS_LINE_2]

#define __REDIS

#endif
```

Web サーバー

Web サーバーを Python で書く試みもありますが、HTML に埋め込むのと同じ JavaScript で作成しました。ファイル名は web-if.js です。ブラウザと温度コントローラの間で、イベントとメッセージを中継するのが、主な機能です。動作については概要のみ説明します。

サーバー機能を立ち上げ、接続してきたブラウザに HTML ファイルを送り、Socket.IO からのイベントを待ちます。イベントを受信したら、そのキーID を FIFO に入れます。

いっぽう Redis FIFO からの受信を待ち、温度コントローラから受け取った表示データを Socket.IO に送っています。

DEBUG を#define すれば、Socket.IO と Redis からの受信を確認できるように記述しています。

Web-if.js

```
/* 温度コントローラ操作用 Web サーバー
   初版 2017/6/5
   最新版 2018/3/26 ... 機能単純化
*/

#include "./include/remote_hmi.h"

/* ライブラリの使用宣言 */
const fork = require(JS_FORK).exec;
var http = require(JS_REQ_HTTP);
var socketio = require(JS_REQ_SOCKET_IO);
var fs = require(JS_REQ_FILE_SYSTEM);
var in_queue1 =
require(JS_REQ_REDIS).createClient();
var in_queue2 =
require(JS_REQ_REDIS).createClient();
var out_queue =
require(JS_REQ_REDIS).createClient();

/* Redis データベースのクリア（1 回でよい） */
in_queue1.flushdb();

/* ブラウザとの相互作用定義 */
var shell = http.createServer(function(req, res)
{
  res.writeHead(JS_HTTP_STATUS_OK,
JS_HTTP_HEADER);
  res.end(fs.readFileSync(JS_HTML_FILE,
JS_CHAR_UTF8));
}).listen(PORT_SOCKET_IO);

/* Socket.IO からの受信開始 */
var io = socketio.listen(shell);

/* Socket.IO からの受信処理 */
io.sockets.on(IO_EVENT_CONNECT, function
(socket){
#ifdef DEBUG
  console.log("connected from a browser\n");
#endif

/* クリックしたボタンごとの処理 */
  socket.on(IO_EVENT_UP, function(){
#ifdef DEBUG
    console.log("UP key is pushed\n");
#endif
    out_queue.rpush(REDIS_KEY, REDIS_UP);
  });

   socket.on(IO_EVENT_DOWN, function(){
#ifdef DEBUG
    console.log("DOWN key is pushed\n");
#endif
    out_queue.rpush(REDIS_KEY, REDIS_DOWN);
  });

  socket.on(IO_EVENT_MODE, function(){
#ifdef DEBUG
    console.log("MODE key is pushed\n");
#endif
    out_queue.rpush(REDIS_KEY, REDIS_MODE);
  });

  socket.on(IO_EVENT_DISCONNECT, function(){});
});


/* 温度コントローラとの通信 */

/* ブラウザへの表示文字列送信 */
/* Socket.IO へのメッセージ送信 */
function create_message(socket, event_id, msg){
  socket.emit(event_id, msg);
}

/* 表示関数 */
function show(event_id, message){
#ifdef DEBUG
  console.log("received", event_id, message);
#endif
  create_message(io.sockets, event_id, message);
}

/* FIFO の受信処理 ... 最後に再び FIFO 受信要求をする */
/* 1 行目の表示 */
function show_title(err, msg){
  current_title = msg[REDIS_DATA];
  show(IO_EVENT_LCD_1, msg[REDIS_DATA]);
  in_queue1.blpop(REDIS_LINE_1, REDIS_NO_TIMEOUT,
show_title);
}
/* 2 行目の表示 */
function show_data(err, msg){
  current_data = msg[REDIS_DATA];
  show(IO_EVENT_LCD_2, msg[REDIS_DATA]);
  in_queue2.blpop(REDIS_LINE_2, REDIS_NO_TIMEOUT,
show_data);
}

/* Redis FIFO からの受信処理定義 */
/* javascript では FIFO 受信待ちでブロックしない */

in_queue1.blpop(REDIS_LINE_1, REDIS_NO_TIMEOUT,
show_title);
in_queue2.blpop(REDIS_LINE_2, REDIS_NO_TIMEOUT,
show_data);
```

8.2 Web ページの設計

Web ページを記述する HTML ファイルを直接作らず、index.html.source というファイルから cpp で生成するようにしました。これは、Socket.IO を介しての約束事を Web サーバーと共有するためです。

JavaScript で記述された<script>部では、Socket.IO の使用を宣言してから、イベントの処理関数を定義します。表示欄を指定して、そこのテキストを受信したものと交換します。HTML 部にある表示欄では液晶表示器と同じイメージにするため、等幅フォント（<tt>）を指定しています。

script 部のボタンクリック処理関数では、それぞれのボタンに該当するイベントを Socket.IO に発生させています。HTML 部のボタン表示では、クリックしたときにそれぞれの処理関数を呼ぶように指定しています。

index.html.source

```
/* ブラウザからの操作用 index.html ファイルのソース
   2016/6/20 Chuji
   2019/3/20 単純化

   使用する前に cpp で処理すること
   $ cpp index.html.source >index.html
*/


#include "./include/remote_hmi.h"

/* html 宣言部 */
<!DOCTYPE html>

<html lang="ja">
<meta charset="utf-8">
<head>
<script type="text/javascript"
src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
```

```
/* Socket.IO 用ソケット生成 */
var mysock =  io.connect(URL_SOCKET_IO);

/* Socket.IO からの受信処理 */
mysock.on(IO_EVENT_CONNECTED, function(){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="connected to the host";
#endif
});

/* 表示要求の処理 */
/* 1 行目 */
mysock.on(IO_EVENT_LCD_1, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="LCD 1 data is received";
#endif
  obj=document.getElementById(WEB_ID_LCD_1);
  obj.textContent=msg;
});

/* 2 行目 */
mysock.on(IO_EVENT_LCD_2, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="LCD 2 data is received";
#endif
  obj=document.getElementById(WEB_ID_LCD_2);
  obj.textContent=msg;
});

/* 画面上のボタンクリック処理 */
function on_click_up(){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW2);
  obj.textContent="UP is pushed";
#endif
  mysock.emit(IO_EVENT_UP);
}

function on_click_down(){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW2);
  obj.textContent="DOWN is pushed";
#endif
  mysock.emit(IO_EVENT_DOWN);
}

function on_click_mode(){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW2);
  obj.textContent="MODE is pushed";
#endif
  mysock.emit(IO_EVENT_MODE);
}

</script>


</head>

/* HTML 主要部 */
<body>

<center>
<font WEB_TITLE_FONT><b>WEB_TITLE</div></b></font>

/* デバグ用：Socket.IO 受信メッセージ表示 */
#ifdef DEBUG
<div id=WEB_MESSAGE_RAW> サーバーからの受信メッセージ表示
欄 </div>
<p>
#endif

/* 液晶表示部の初期表示 */
<font WEB_LCD_COLOR><tt><pre><b>
<span id=WEB_ID_LCD_1>Hello</span>
<span id=WEB_ID_LCD_2>World!</span>
</b></pre></tt></font>
<p><p>

/* デバグ用：ボタンクリック操作の表示 */
#ifdef DEBUG
<div id=WEB_MESSAGE_RAW2> ボタン操作の表示欄 </div>
<p>
#endif

/* 操作用ボタンの表示とクリック処理定義 */
<input type="button" value=SYMBOL_UP
style=WEB_STYLE_UP onclick="on_click_up();"/>
<input type="button" value=SYMBOL_DOWN
style=WEB_STYLE_DOWN onclick="on_click_down();"/>
<input type="button" value=SYMBOL_MODE
style=WEB_STYLE_MODE onclick="on_click_mode();"/>

</center>
</body>

</html>
```

このソースを以下のように処理して、index.html を作成し、remote_hmi.h 内の JS_HTML_FILE で指定した（ここでは Web サーバーと同じ）ディレクトリに置きます。

```
$ cpp index.html.source | python cleanfile.py
>index.html
```

でき上がった index.html ファイルを下に示します。確かに短くはなっていますが、記述は分かりにくいですね。

index.html

```
<!DOCTYPE html>
<html lang="ja">
<meta charset="utf-8">
<head>
<script type="text/javascript"
src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
var mysock =
io.connect('http://192.168.15.17:50000');
mysock.on("connect", function(){
});
mysock.on("lcd 1", function(msg){
 obj=document.getElementById("lcd 1");
 obj.textContent=msg;
});
mysock.on("lcd 2", function(msg){
 obj=document.getElementById("lcd 2");
 obj.textContent=msg;
});
function on_click_up(){
 mysock.emit("UP KEY");
}
function on_click_down(){
 mysock.emit("DOWN KEY");
}
function on_click_mode(){
 mysock.emit("MODE KEY");
}
</script>
</head>
<body>
<center>
<font size = "4" color="red"><b>Raspberry Pi
Temperature Controller</div></b></font>
<font size = "7" color="navy"><tt><pre><b>
<span id="lcd 1">Hello</span>
<span id="lcd 2">World!</span>
</b></pre></tt></font>
<p><p>
<input type="button" value="▲" style="color:white;
background-color:blue; width:60px; height:30px;
margin-right:auto; margin-left:auto;"
onclick="on_click_up();"/>
<input type="button" value="▼" style="color:white;
background-color:blue; width:60px; height:30px;
margin-right:auto; margin-left:30px;"
onclick="on_click_down();"/>
```

```
<input type="button" value="Mode"
style="color:yellow; background-color:red;
width:60px; height:30px; font-weight:bold; margin-
right:auto; margin-left:30px;"
onclick="on_click_mode();"/>
</center>
</body>
</html>
```

8.3　温度コントローラの機能追加

温度コントローラのソフトウエアモジュール display.py と switch.py で、REMOTE_OP を#define したときにだけ残る記述を追加します。さらに controller.py で割り込み待ち方法を変更します。4 章に掲載したコードには、既に追加されています。

display.py と switch.py では、まず redis_for_python.py をインクルードし、オブジェクトの初期化で Redis FIFO の使用を宣言（open_FIFO）します。

display.py では、表示イメージを更新したときに、同じイメージを文字列として FIFO にも送ります（rpush）。

switch.py では、割り込み待ち部で、FIFO からのデータ待ちをします（blpop）。イベントのキーを調べて、それぞれの割り込み処理ルーチンを呼ぶようにしています。

controller.py では sleep する代わりに、switch.py の待ち操作を実行します。

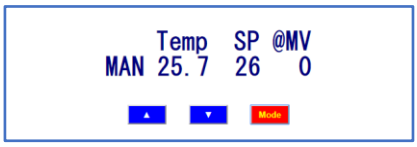
クリック割り込み処理のあと、次のタイマー割り込みでキースイッチの状態を調べますが、ハードウエアは操作していないので、自動的に押下状態を抜けてしまいます。つまりブラウザのボタンでは、長押しができないということです。マウスボタンの連打で対応するのですが、チューニングのとき以外に不便に感じることはありません。

8.4　検証

では Web サーバーを動かしてみましょう。先に、Web サーバーを子プロセスとして起動（コマンド列の最後に&があることに注意）してから、温度コントローラを起動します。

```
$ cpp web-if.js |python cleanfile.py | sudo node &
$ cpp -DREMOTE_OP controller.py| phython
cleanfile.py | sudo python
```

温度コントローラを起動する前に、ブラウザで http://192.168.15.17:50000（URL_SOCKET_IO で定義したアドレス）を開くと、Hello World!という文字とボタンが表示されるはずです。それから温度コントローラを起動すると、ここが液晶表示器のコピーに変わります。



赤字の Mode ボタンをクリックすると、1 行目の左端に O/S というターゲットモードが表示されます。▲ボタンをクリックして MAN に表示を変え、もう一度 Mode ボタンをクリックします。ターゲットモードが消え、しばらくしたら実効モードが MAN になり、MV という文字の前に@が現れたら動作は期待どおりです。

期待どおりにならない場合は、以下の手順で確認してください。まず、index.html.source から index.html を作る時に、-DDEBUG を追加します。ボタンをクリックしたときと、Web サーバーから更新イメージを受信したときに、そのことをブラウザ上に表示するようになります。

サーバーの実行を止め、-DDEBUG オプションをつけて再起動します。これでブラウザからのクリックイベントと、温度コントローラからの表示イメージ受信が表示できるようになります。

```
$ sudo killall -9 node
$ cpp -DEBUG web-if.js |python cleanfile.py | sudo
node
```

イベントが伝達されたかどうかを確認してください。ブラウザかサーバーのどこかで途切れている可能性があります。

温度コントローラそのものは、-DDEBUG_REDIS オプションを指定してやれば、イベントが伝わったときの動作を確認できます。

8.5　プログラムの自動起動

ここまでの動作確認が済んだら、温度コントローラと Web サーバーを自動で起動できるようにしておきます。温度コントローラはデフォルト構成（温度センサは ADT7410A、制御周期は 1 秒、制御パラメータはデフォルト値）にしておきます。

/etc/rc.local に記述を追加します。これはシステムが使うファイルなので、sudo 権限で編集します。

```
$ sudo vi /etc/rc.local
```

末尾に次の 2 行を追加します。ユーザ名とディレクトリは、実際の名前に合わせてください。システム起動時に実行されるので、ファイルのありかはルートディレクトリから記載します。

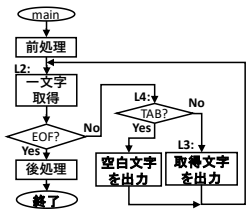
```
cpp /home/chuji/controller/web-if.js | python
/home/chuji/controller/cleanfile.py |
/usr/local/bin/node &

cpp -DREMOTE_OP
/home/chuji/controller/controller.py | python
/home/chuji/controller/cleanfile.py | sudo python
&
```

これで自動起動ができるようになりました。試しに再起動してみてください。液晶表示器に温度などが表示され、ブラウザからも見えるはずです。この状態で温度コントローラとして動作するので、SSH 接続する必要がなくなりました。

シャットダウンもブラウザから行えます。Mode→▼→Mode→Mode→Mode とボタンをクリックすると、ターゲットモードフィールドに STP と表示されるようになります。ここで▼ボタンをクリックすれば、シャットダウンが実行されます。Raspberry Pi の LED が消灯したら、電源を落としてください。

コンパイラの出力を読む

コンパイラへの興味（55 ページ）から、そのアセンブリ語出力を読むようになりました。付録に出てくる t2s.c をいろいろな環境で処理（cc -S t2s.c）した結果のアセンブリ語ソース（コメント部を省略）を下に示します。プログラムの構造やラベル（L2 など）は、右図のように全く同じです。構文解析部はすべて同じ（gcc）で、CPU 毎に別々のコードを生成していることが分かります。Cygwin では、Windows の上に UNIX 環境を作りこんでいるため、システムコールの部分が複雑になっています。



Raspberry Pi

```
main:
        @ args = 0, pretend = 0,
frame = 0
        @ frame_needed = 1,
uses_anonymous_args = 0
        stmfd   sp!, {fp, lr}
        add     fp, sp, #4
        b       .L2
.L4:
        ldr     r3, .L5
        ldr     r3, [r3, #0]
        cmp     r3, #9
        bne     .L3
        mov     r0, #32
        bl      putchar
        mov     r0, #32
        bl      putchar
        b       .L2
.L3:
        ldr     r3, .L5
        ldr     r3, [r3, #0]
        mov     r0, r3
        bl      putchar
.L2:
        bl      getchar
        mov     r2, r0
        ldr     r3, .L5
        str     r2, [r3, #0]
        ldr     r3, .L5
        ldr     r3, [r3, #0]
        cmn     r3, #1
        bne     .L4
        mov     r0, r3
        ldmfd   sp!, {fp, pc}
```

Ubuntu (X64)

```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        jmp     .L2
.L4:
        movl    c(%rip), %eax
        cmpl    $9, %eax
        jne     .L3
        movl    $32, %edi
        call    putchar@PLT
        movl    $32, %edi
        call    putchar@PLT
        jmp     .L2
.L3:
        movl    c(%rip), %eax
        movl    %eax, %edi
        call    putchar@PLT
.L2:
        call    getchar@PLT
        movl    %eax, c(%rip)
        movl    c(%rip), %eax
        cmpl    $-1, %eax
        jne     .L4
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
```

Cygwin64 (Window10)

```
main:
   pushq %rbp
   .seh_pushreg  %rbp
   movq  %rsp, %rbp
   .seh_setframe %rbp, 0
   subq  $32, %rsp
   .seh_stackalloc 32
   .seh_endprologue
   call  __main
   jmp   .L2
.L4:
   leaq  c(%rip), %rax
   movl  (%rax), %eax
   cmpl  $9, %eax
   jne   .L3
   call  __getreent
   movq  16(%rax), %rax
   movq  %rax, %rdx
   movl  $32, %ecx
   call  putc
   call  __getreent
   movq  16(%rax), %rax
   movq  %rax, %rdx
   movl  $32, %ecx
   call  putc
   jmp   .L2
.L3:
   call  __getreent
   movq  16(%rax), %rdx
   leaq  c(%rip), %rax
   movl  (%rax), %eax
   movl  %eax, %ecx
   call  putc
.L2:
   call  __getreent
   movq  8(%rax), %rax
   movq  %rax, %rcx
   call  getc
   movl  %eax, %edx
   leaq  c(%rip), %rax
   movl  %edx, (%rax)
   leaq  c(%rip), %rax
   movl  (%rax), %eax
   cmpl  $-1, %eax
   jne   .L4
   movl  $0, %eax
   addq  $32, %rsp
   popq  %rbp
   ret
```

9　プロジェクトは終わらない

Raspberry Pi を使った温度コントローラ開発プロジェクトは、いったん完了とします。じゅうぶん実用に耐えるだけの機能は実現できました。

9.1　初期モデルと改造の歴史

このプロジェクトのもとになる一号機の構想を始めたのは、2014 年の秋のことでした。休日に秋葉原に通ったり、仕事の合間にプログラムを書いたりして、温度コントローラとして動き出したのは一年後のことです。

それなりに便利なものができたのですが、使っていると、いろいろ不満や欲が出てきます。それに沿って、改造を重ねてきました。まず、現在の安定したソフトウエアのすべてのバックアップを取ります。PC を使うのが良いですね。開発仕様書を改訂して、ソフトウエアを改造しました。その過程は以下のようなものでした。

1. 制御対象ごとにチューニングした制御パラメータや制御周期を指定して、それぞれのシェルスクリプトを作り、すぐに実行できるようにした。
2. レコーダーで制御状態をリアルタイムに観測できるようにした。
3. センサ故障や接続不良が起きた時の処理を付け加えた。
4. Web インターフェースを実装して、リモート操作ができるようにした。

今回のプロジェクトで説明したのは、ここまでの全システムです。実は、もっと改造は進んでいるし、今後の改造構想もあるのですが、JavaScript の比重が増えてしまいます。この本のカバー範囲に収まりきらないので、以下では概要だけを説明します。

9.2　温度コントローラの発展形

制御対象の選択

制御対象ごとに、最適な制御パラメータや制御周期が異なります。それぞれ、シェルスクリプト化しているのですが、いちいち SSH 接続が必要でした。

そこで Web サーバーと HTML ファイルを改造して、実行するシェルスクリプトを選べるようにしました。これで、ブラウザさえあれば、任意の温度コントロールを選んで実行し、シャットダウンまでできるようになりました。PC なしで温度コントローラが使えます。もちろん、タブレット端末かスマートフォンがあれば、の話ですが、機動性は格段に向上しました。レコーダー用途の PC は、まだ必要でした。

WiFi 環境の選択

WiFi 環境のない外出先でも温度コントローラを使えるように、複数の wpa_supplicant.conf を用意し、切り替えるようにしました。最初は SSH 接続してから切り替えるしかなかったので、出かける前に切り替える必要がありました。忘れて出かけてしまい、往生したこともあります。

そこで、キースイッチで、wpa_supplicant.conf を選んで、切り替えることができるソフトウエアを作り、/etc/rc.local に登録して、温度コントローラの代わりに、システム起動時に立ち上がるようにしました。WiFi 環境を切り替えてからシステムをリブートし、前述の Web サーバーで温度コントローラを選んだら、WiFi 環境選択プログラムを停止（killall -9 python）させ、選択した温度コントローラを起動するようにしました。

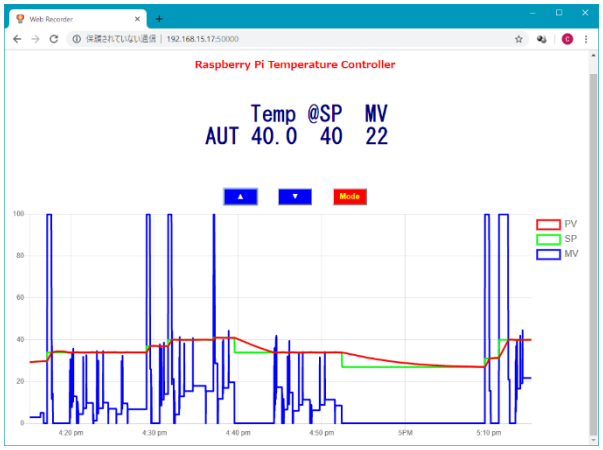
この WiFi 選択ソフトウエアは、switch.py や lcd.py をそのまま部品として使い、hmi.py を作り変えるだけで実現できました。

WiFi 環境が全くない場所でも使えるよう、Raspberry Pi に（ブラウザ等に IP アドレスを割り振る）DHCP サーバーを搭載しました。この WiFi 選

択ソフトウエアから、必要な時に起動できるようにしています。

ブラウザ上のレコーダー

レコーダーとして使うためだけに PC を持ち歩くのは不便です。タブレット端末やスマートフォンでも表示できるよう、Chart.js を使ってレコーダーを web 用に書き換えました。これで、温度コントローラとブラウザさえあれば、 完全に PC フリーで、どこでも調理ができるようになりました。



パラレル入出力液晶表示器

液晶表示器には、I2C バスインターフェースのものだけでなく、パラレルバスから駆動するものもあります。そちらの方が種類も豊富です。

ちょっと処理時間はかかりますが、GPIO ポートをまとめて、パラレルバスの動作を模擬させることができます。今回の公開範囲には含めていませんが、インターフェースユニットに必要な回路が搭載してあり、初期の i2c.py のなかに駆動ルーチンを加えていました。

2 ループコントローラ

ハードウエアの章で説明したように、温度センサと PWM 出力は 2 チャンネルまで使えるようになっています。PWM 出力の一つはアラーム表示灯用に使っていますが、必ずしも必要ではありません。液晶表示器にアラームを表示してもいいので。

というわけで、温度コントローラは 2 つの制御対象をいちどにコントロールできます。PID 制御部などはオブジェクトになっているので、2 つ生成すればいい。液晶表示器は 16 文字×4 行などを使うとすべてを表示できます。

温度センサドライバ thermo.py は、#ifdef でセンサを選択していますが、オブジェクト生成時に温度センサを指定するように改造すれば、異なる温度センサを併用することもできます。

プログラム可変式温度コントローラ

燻製では、「40℃で 2 時間乾燥後、温度を徐々に上げて 70℃で 3 時間燻煙する」といったレシピ（調理法）をよく見かけます。次節にも出てくるタイマーで、PID 制御の設定値を変えることで実現できます。温度コントローラとは別プロセスで起動し、Redis を介して、▲キー押下イベントを必要回数だけ送れば簡単に作れます。Redis では、二つ以上のプロセスが、同じ FIFO にイベントを入れられることを利用しています。

フライヤー

湯温センサは水が沸騰するまでは使えますが、天ぷらやフライのように 100℃を越える調理には使えません。もっと高温まで使える温度センサが必要です。ステンレス管入り熱電対が適しています。

熱電対は異種金属を二か所で接触させると、両接触点の温度差に依存した起電力を発生します。温度差なので、片方を氷水に浸けるか、常温側接点の温度を測って補正（冷接点補償と言います）します。また、起電力は温度差に比例しているわけではないので、線形化処理も必要です。

しかし今では、これらの補正を行う IC があるので、I2C あるいは SPI 通信を介して温度を測ることができます。温度センサモジュール thermo.py を改造するだけでいいのです。液晶表示の書式は、199.9 または 999 なので、100℃以上の温度制御もできるようになります。

9.3　ソフトウエアモジュールの活用

ここで開発したソフトウエアモジュールは、部品として他の目的に利用できます。前節の WiFi 環境変更ソフトウエアも、その例の一つです。外にもいろいろ考えてみました。Raspberry Pi ZERO は比較的安価なので、組込みシステムを作るのに便利です。

汎用タイマー

現在のハードウエアだけで、1 回路の汎用タイマーが作れます。半導体リレーユニットを PWM ではなく、ステータスで駆動すれば、500W までの電気器具が制御できるようになります。照明器具や、クリスマスの電飾、電熱器による煮込み料理などに使えます。わが家ではベランダの草花の水やりに使っています。

キースイッチと液晶表示器で操作しても良いし、ブラウザから操作するのも便利そうです。時刻は time.time()で取得できます。

環境モニタリング

今の温度コントローラのままでも、気温計として使うことができます。空気温度センサを繋いで、O/S モードのまま屋外に置けば、室内に居ながらにして、暑い外気温を知ることができます。Raspberry Pi ZERO に直接温度センサを接続し、半導体リレーや操作ユニットを外せば、小型の環境モニタリング装置ができます。

I2C バスで接続できる湿度や気圧センサ（あるいは温度・湿度・気圧を単一モジュール化した製品もある）を追加すれば、環境モニタリングは完ぺきになります。

インターネットラジオサーバー

インターネット上には、音楽などを四六時中配信してくれる、インターネットラジオ局が多くあります。受信したデータをオーディオ信号や PCM 信号に変換するカードを取り付け、mplayer をインストールすれば、さまざまな音源を再生することができます。温度コントローラ用の Web サーバーを改造すると、インターネットラジオサーバーが作れます。放送局の URL などは Redis データベースで管理できるので。選局はブラウザから行いましたが、キースイッチと液晶表示器でもできそうです。

自作したサーバーを使ってみたのですが、いちいち放送局を探して登録するのが面倒なのと、画面のデザインが武骨すぎて面白くありません。結局、人気ソフトの Volumio をインストールしてしまいました。

9.4　プロジェクトは続く

この章のはじめで、開発プロジェクトは「いったん」完了と言いました。それは、この本での説明を終えるという意味で、もう何もしないということではありません。むしろ、プロジェクトはずっと終わらないのです。

温度コントローラはずっと使い続けていくので、その間に不具合が発生することもあります。じっさい、電源ケーブルに足を引っかけて、コントローラを作業台から落としてしまったことがあります。幸い、内部配線用のフラットケーブルが抜けたくらいで済んだのですが、動作テストの手順が確立していたので、すぐに使えるようになりました。

前二節で説明したように、改造や機能拡張は今でも続いています。初号機は Raspberry Pi B でしたが、Raspberry Pi ZERO に置換したことで、使い勝手がずっと良くなりました。調理に使わないときは、環境モニタリングや水やりといった「副業」も、こなすようになりました。

プロジェクトには仕様書を用意し、必要に応じて増補・改訂しています。これは保守や改造のときに、大いに役に立ちました。資料を残しておくことは、とても大切です。

もっと大事なことは、この経験を普遍化して、今後の役に立てることです。この本を書いたのも、苦労したこと、うまくいったことを振り返り、自分の次のプロジェクト（電子工作に限らず、何かすること全体）への反映を目論んだのが、第一の理由です。公開することにしたのは、私のプロジェクトを追体験することで、皆さんにも何か得ることがあるかもしれないと考えたからです。

「ものつくり敗戦」で指摘された、日本の苦手種目である、理論と論理、システムとソフトウエアを身に着けることも大切です。デジタルネイティブと呼ばれる世代は、これらを使いこなすのが得意です。でも、使いこなすことと、理解することは、同じではありません。なぜか分からないけど、うまくいったからいいや、というだけでは、別の課題の役に立たないからです。ものごとの背景や仕組みを理解してはじめて、応用ができるのだと思います。あるプロジェクトで有効だったアプローチも覚えておくと、意外なときに役に立つものです。これらが、一人ひとりの懐を深くしてくれます。

誰もが、匠と呼ばれるようなソフトウエア技術者やプログラマーになれるわけではないし、その必要もありません。ただ、複雑で目に見えにくいものに、ためらわずに取り組む姿勢は大切にしたいと思います。この本が、それに役立つことがあれば、これに勝ることはありません。

燻煙室のヒーター

本論からは外れますが、燻煙室にヒーターを組み込む方法を紹介します。この方法は、秋葉原の坂口電熱さんで教わったものです。趣味用の小規模な買い物だったのに、実に親切に相談にのってくれました。

まず発熱材であるニクロム線を調達します。今回は300Wのものを調達しました。長さは15cmくらいあります。

これを均等に引き伸ばして、必要な長さにします。メジャーを見ながら引っ張りますが、力を抜くと元に戻ろうとするので、さらに引っ張ります。

目的の長さになったら、右下の写真のようなセラミック管に通していきます。ニクロム線の外径にあったものを一袋買ったら、2台作ってもおつりが来ました。針金でセラミック管の部分を縛り、台に固定します。針金がニクロム線に触れるとショートの原因になるので、気をつけましょう。

下の写真は木板で作った箱型燻煙室に取り付けた様子です。底には煙を入れる穴が開いていますが、その上に載せて使います。周りに立っている金属ネジは、この上にアルミ皿を置いて、油受けや燻煙材を燃やすのに使うためのものです。両脇の木材の上に金網を渡して、食材を載せます。

少量の燻製を作るなら、園芸店などで売っているような木製の樽を使うこともできます。古い電熱器のヒーターを骨董市（装飾用だったようです）で買ってきて、底に浮かせるように取り付けました。2列に並んだ釘は、下が燻煙材の皿を、上が食材を置く金網を支えるためのものです。

電子工作というより、木工と針金細工それに電気配線が中心ですが、こんなものを作ってはじめて、当初の目的が達成できるようになります。

Linux と Windows のファイル

先に Windows と Linux の間のファイル互換性問題を指摘しました。もう一度整理すると

1. 日本語コードが異なる。Linux では UTF-8 を、Windows では Shift-JIS を使っており、互いの日本語を表示できないことがある。

2. 一行の終わりのコードが異なる。Linux では newline（NL）一文字、Windows では Carriage-Return（CR）と Line-Feed （LF）の二文字を使う。

というものでした。この間の変換を行うツールとして Linux の nkf（Network Kanji Filter）というものがあります。Raspberry Pi や Ubuntu PC にインストールする手順は下のとおりです。

```
$ sudo apt-get install nkf
```

nkf にはいろいろオプションがありますが、とりあえず次の 6 つだけ知っていれば役に立ちます。

オプション	処理
--guess	ファイルの日本語コードと改行を表示する
--overwrite	変換した結果を、元ファイルに上書きする
-s	Shift-JIS に変換する
-w	UTF-8 に変換する
-Lw	改行を Windows 式（CR+LF）に変換する
-Lu	改行を Linux 式（NL）に変換する

nkf のオプション（一部）

--overwrite を指定しなければ、変換結果は標準出力に表示されるので、リダイレクトで別のディレクトリや別ファイルにすることもできます。

私のプログラムは区切りにタブを使うことが多いのですが、この本にプログラムリストを掲載すると横幅を取りすぎます。それでスペース 2 文字に置き換えるプログラム t2s.c を用意しました。

t2s.c

```
/* tool to replace a TAB with two SPACEs */

#include <stdio.h>

#define NL      '\n'
#define CR      '\r'
#define LF      '\n'
#define TAB     '\t'
#define SPACE   ' '

int c;

int main(){
        while ((c = getchar()) != EOF){
                if (c == TAB){
                        putchar(SPACE);
putchar(SPACE);
                }
                else putchar(c);
        };
}
```

使うコンピュータ上でコンパイルして使います。

```
$ cc -ot2s t2s.c
```

これをホームディレクトリ（~/）に置いて、以下のシェルスクリプト port2dos を用意しました。一番目のパラメータで指定した拡張子のファイルを変換して、二番目のパラメータで指定したディレクトリに同名で格納します。

port2dos

```
# tool to convert UTF-8+newline to SJIS+CRLF and
replace TAB with two SPACEs
# usage: port2dos file-extention directory-to-save

for f in *.$1;
do
        nkf -s -Lw $f | ~/t2s >$2/$f
done
```

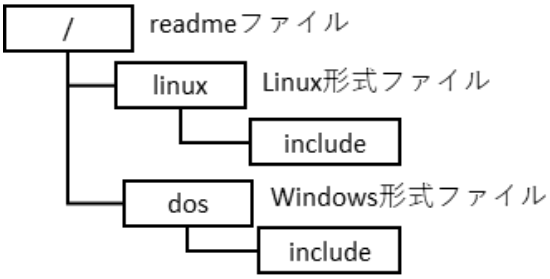
下の使用例では、カレントディレクトリにある*.py ファイルをすべて、Windows 形式にしてから、タブをスペースに変換して、結果を../dos に収納します。

```
$ port2dos py ../dos
```

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。ただし著作者は、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は負いません。

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、前出のシェルスクリプト port2dos で変換したファイルが収納されています。t2s.c と port2dos や、テストパターンと結果は linux ディレクトリにしか入っていません。



配布ファイルのディレクトリ構成

ダウンロード元:

https://www.akiyama-tokyo.net/electronics/TemperatureController.zip

改訂履歴：

Rev.1 全回路図（17 ページ）の誤記訂正（2020 年 7 月 12 日）

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

2019 年 5 月

著者・発行者　秋山忠次 (chuji@akiyama-tokyo.net)

非売品

非売品