Nichiden Documentation

リリース ***0.1***

Kenichi Ito

3 月 07, 2017

目次:

第 1 章

東京大学地文研究会天文部 日電引き継ぎ

- 更新: 2017/03/04 Kenichi Ito(nichiden_27)

日電の引き継ぎ文書です。ソースは全て公開しますので、ご自由に修正・追記してください。

1.1 冒頭部の情報について

各資料の冒頭部分には、「書いた人」「更新日時」「実行に必要な知識・技能」「タスクの重さ or 難易度」「タスクの必須度 or 情報の必須度」といった項目が示されています。

1.2 実行に必要な知識・技能

資料で解説しているタスクや知識を実践するのに必要な知識などです。自分が理解できそうな資料を探したり、作業をするために何を身につけるべきか判断したりする際に参考にしてください。

1.3 タスクの重さ

タスク関連の記事に付いています。

- 1: 数時間で可能
- 2: 数日かかる
- 3: 数週間

- 4: 一月はかかる
- 5: 準備だけで数ヶ月

の 5 段階です。作業計画を立てる際などにご覧ください。

ただし、各代の方針によって、個々のタスクの重さは変動することに注意しましょう。

1.4 難易度

技術情報などをまとめた記事に付いています。

- 1: 常識の範囲
- 2: 少しやれば可能
- 3: 練習・勉強が必要
- 4: 分野に慣れてればできる
- 5: 得意分野ならどうぞ

学んでおきたい分野の大枠を掴むのに利用してください。

1.5 タスクの必須度

タスク関連の記事に付いています。

- 1: よほど暇なら
- 2: たまにやるべき
- 3: 年による
- 4: 毎年やるべき
- 5: しないとプラネ終了

今年の日電で何をやるか決める手助けになるかもしれません。

1.6 情報の必須度

技術情報などをまとめた記事に付いています。

- 1: 趣味レベル
- 2: 知ってると楽
- 3: 必要な場合がある
- 4: 担当者には必須
- 5: 全員必須

第 2 章

プロジェクト マネジメント

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/02/18
- 実行に必要な知識・技能: 特になし
- タスクの重さ: 3: 数週間
- タスクの必須度: 4: 毎年やるべき

2.1 概要

日電長向けの内容です。日電の仕事をどう進めていくかなどを書きます。

2.2 チーム編成

2.2.1 連絡手段を決める

27 では Slack を使用した。日電は作業内容が多岐にわたるため、一つのチャットグループでは話題が混乱してしまう。LINE でも多数のグループを立てることで対応はできるが、管理が困難になるので最初からチームチャットを導入しておくことをお勧めする。

以下は 27Slack の最終的なチャンネル構成である。

```
#acrab ... 星座絵アプリ
#dome ... ドームコンソール
#general ... 総合
#ginga ... ぎんとう
```

```
#haisen ... 配線
#hojyotou ... 補助投
#ittou ... いっとう
#log ... 活動記録
#nissyu-idohen ... 日周緯度変
#notifications ... 通知を流す用
#parts ... 部品管理
#piscium ... 星座絵無線機
#random ... 雑談
#seizae ... 星座絵
```

Slack の特徴の一つに、他の Web サービスとの豊富な連携がある。27 では、ToDo リストを共有できる Trello や GitHub を使用した。通知が Slack で一覧できるので、進捗の共有が簡単になる。

余裕があれば、進捗を煽る Bot などを作成しても面白いかもしれない。

2.2.2 目標を設定する

日電のやることは毎年そう変わらないが、できれば先年までと違うことに挑戦するとよい。ただし、新しい製品や機能には不具合や故障が付き物 であり、早期に目標を設定して動き始めることが重要だ。

前年度までに壊れてしまったものや、部員・観客へのインタビューで不満が多く寄せられた部分などが、新たな目標の候補となるだろう。

2.2.3 担当を決める

日電の仕事の特質は、種類が多く、それぞれの作業は個人作業になりやすい ことである。また人数も限られているので、各自の責任感を高める意味でも早期に担当を振っておくべきだろう。これには、日電員が作業に向けて勉強するにあたり、個々の課題を明確にするという利点もある。

2.2.4 スケジュールを決める

進捗は必ず遅れるものだが、だからと言ってスケジュールを切らずに闇雲に進めるのは大変危険である。最低限、どの期間に何をするかの大枠は示しておきたい。27 日電での例は以下の通り。

- 準備期間 ～8/1(S セメ試験)
- 試作期間 8/2～9/25(夏休み終了)
- 製作期間 9/26～10/31

- 動作試験 11/1～11/24(駒場祭前日)

もちろん、実際の作業に入った後の進捗の把握や再検討は、日電長が随時やっておく必要がある。

2.3 作業が始まったら

2.3.1 夜間

日電は個人プレーが多いので夜間への参加は必須ではないが、他の投影機は随時日電のサポートを必要とするので最低一人は出ておくのが望ましい。

また、作業で得られた進捗は、箇条書きなどにまとめて日電全員に報告 するとよい。作業に参加していないメンバーが現状を把握できる上、日電長自身も進捗を目に見える形で確認できる。

2.3.2 卒検・リハ

卒検やリハでは、ドームを膨らませて投影機を配置し、点灯させる。この間の配線は日電が全て行うので、大変仕事が多くかつ責任が重い。この二日間だけは日電メンバーを全員揃える のが重要だ。前々から日程は判明しているはずなので、各自の予定を確認しておこう。

第 3 章

日周緯度変 (日周緯度変換機構)

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/03/03
- 実行に必要な知識・技能: 歯車機構
- タスクの重さ: 5: 準備だけで数ヶ月
- タスクの必須度: 2: たまにやるべき
- 元資料
  - `saitama.pdf` by 荒田 実樹 (nichiden_23)

3.1 概要

日電の正式名称「日周緯度変・電源パート」から分かる通り、日電の本来の任務は日周緯度変装置を動作させることです。この記事では、日周緯度変装置自体に関しての情報をまとめます。

3.2 構成

日周緯度変の役割は、プラネタリウムにおいて日周運動を再現する、あるいは (仮想的な) 観測地の緯度を変更することである。なお、天文部の現在のプラネタリウムには歳差軸はない。

日周緯度変は大きく

- 日周ギヤボックス
- 緯度変ギヤボックス

の 2 つに分かれ、それぞれに対応するステッピングモーターが搭載されている。

このステッピングモーターは一個数万円するものだが、本番中に劣化するので数年おきに買い換えられている。高価な割に**ケーブルが切れやすい** ので、取り扱う際は十分な注意を要する。

ギヤボックスはどちらも **10 年以上前に製作された** ものである。製作年度に関して確かと言える情報は見つかっていないが、緯度は 1994 年の 05 主投、日周は 2000 年前後で製作されたようだ。

これほどの機構を再現できる技術が失われたため、かなりの長期間作り変えられていない。ただし、いつかは必ず壊れてしまうものであり、**なるべく早期に予備を製作すべき** だろう。日周緯度変の製作に成功すれば、天文部の歴史に名が残ることは間違いない。

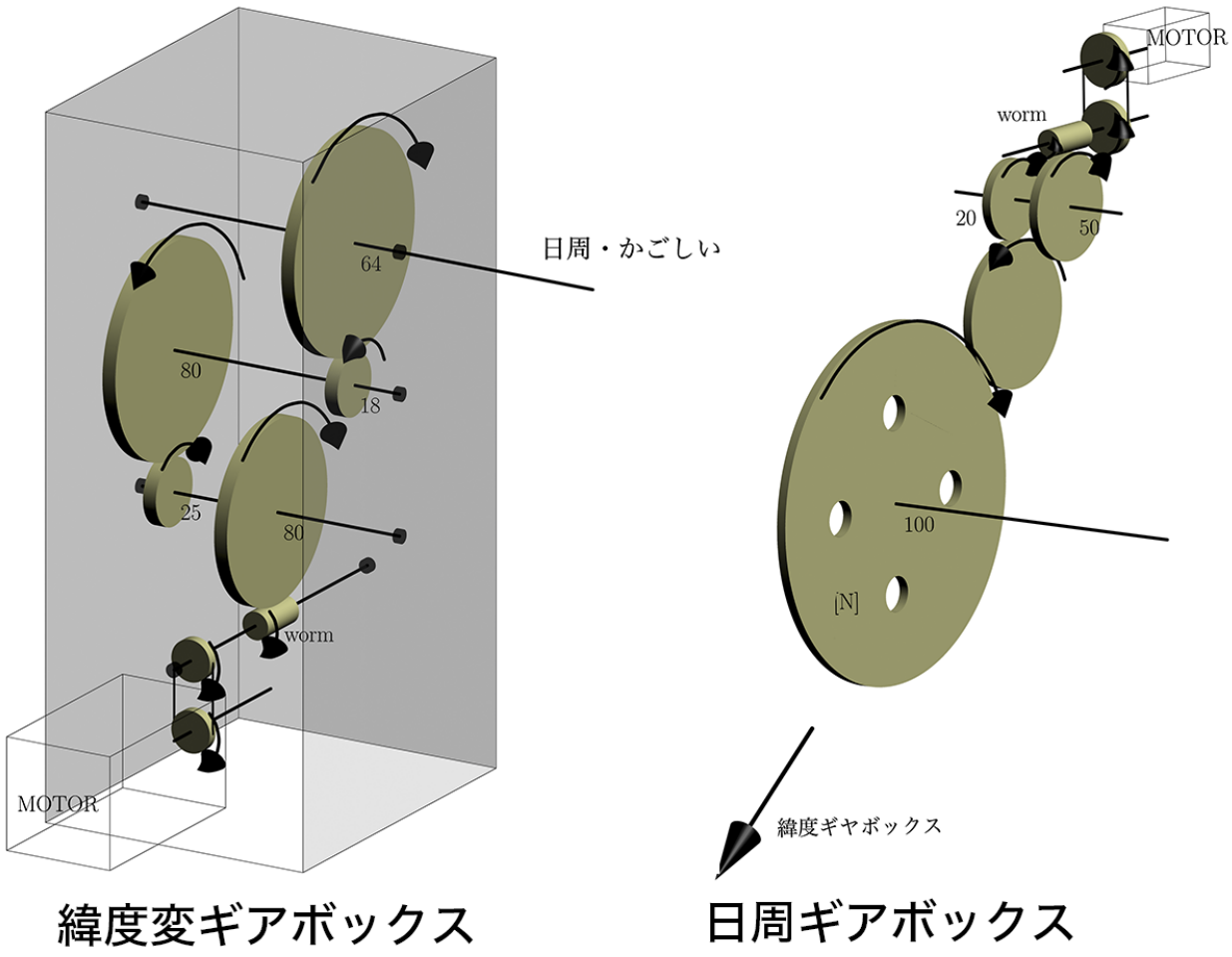
特に緯度変に関しては、軸が摩耗していたり、ギアの噛み合わせが悪かったりして、緯度を動かし続けるとある角度でかごしい全体が「揺れる」ことがある。この点は早急に改善が必要である。

ギヤボックス周辺の可動部分には**定期的にグリスを塗るか、潤滑油を差しておく** ことが望ましい。ただし 2017 年現在、機構部分の保守作業の大部分は、**かごしいの作業の過程で行われている。** ごきぶりやしいたけとの接続も彼らの方がより把握しているはずなので、任せておいて問題はないだろう。

3.3 歯車機構と減速比

日周緯度変を自在に操るには各ギヤ比を知っておく事が必要である。

もちろん実物を見れば調べられるのだが、それなりに面倒だったので同じ手間を繰り返さないよう図に各ギヤの歯の数を載せておく。また、「正の向き (時計回り)」にモーターを回した時に各ギヤがどちら向きに回るかも載せておく。



図に書いてあるギヤの歯の数からギヤ比を計算すると、

$$\frac{1}{80} \times \frac{25}{80} \times \frac{18}{64} = \frac{9}{8192} \quad \text{緯度変}$$

$$\frac{1}{50} \times \frac{20}{100} = \frac{1}{250} \quad \text{日周}$$

となる。なお、これらのギヤの歯の数は動かしながら手で数えたので、もしかすると数え間違いがあるかもしれない。あまり過信しないよう。

つまり、各軸をある角速度で回したい場合、緯度変の場合は `8192/9` を、日周の場合は `250` をそれぞれ掛ければモーターが出力すべき角速度が求まる。日周緯度変関連のソースコードで 8192 や 250 といった定数が登場するのは、この減速比の変換をするためである。

3.4 モーターの角速度

ステッピングモーターは速度を自由に設定できるが、もちろん上限はある。緯度変更モジュール単体で試したところ、緯度モーターの最高角速度は $3500deg/s$ 程度であった。

この速度をかごしいの速度に換算すると $3500/(8192/9) = 3.84deg/s$ となる。1 度動かすのに 0.26 秒、南北の反転に 46.8 秒ほどという計算だ。

ただし、この $3500deg/s$ という速度は速度を徐々に上げていった場合の最高速度であり、停止した状態から回転させる場合は $1800deg/s$ 程度が限度だと思われる。この場合かごしいの速度は $1800/(8192/9) = 1.98deg/s$ となり、1 度動かすのに 0.5 秒、南北反転には 1 分半かかることになる。

また、実際にかごしいを載せた場合や、主投影機を全て設置した場合など、回すものの重量によってさらに実際の速度は低下するので注意すること。

第 4 章

日周緯度変 (さいたま)

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/02/26
- 実行に必要な知識・技能: AVR マイコン、電子回路
- タスクの重さ: 3: 数週間
- タスクの必須度: 3: 年による
- 元資料
  - **日周・緯度変資料**.pdf by 岩滝宗一郎 (nichiden_22)
  - saitama.pdf by 荒田 実樹 (nichiden_23)

4.1 概要

日周緯度変コントローラ「さいたま」の使い方や仕組みなどを説明します。壊れて部品交換が必要になった・新しいさいたまを作りたい際に参照してください。

4.2 沿革

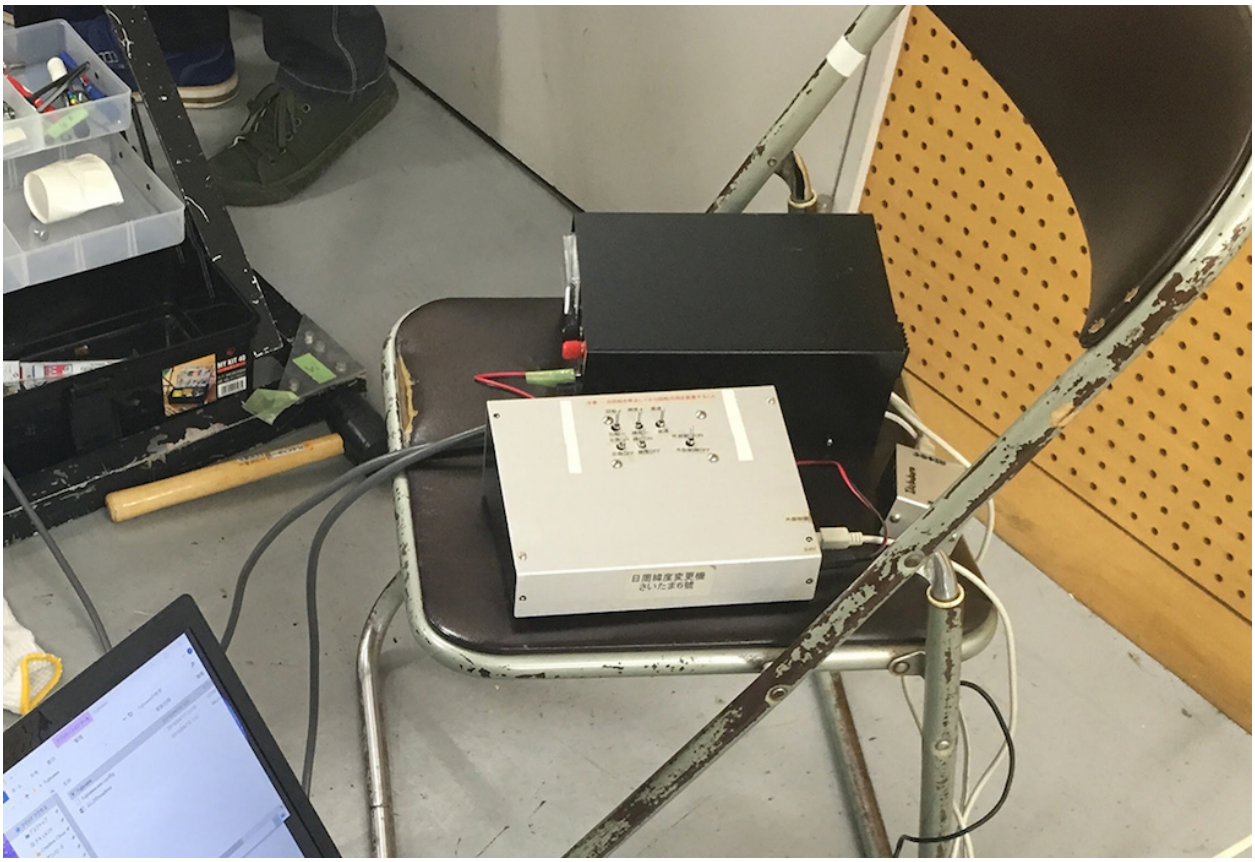
日周緯度変のステッピングモーターを回すには、モータドライバという回路が必要になる。駒場祭プラネではモータドライバと制御基板を内蔵した専用コントローラーを使っており、2002 年の 13 日周以降今日に至るまで「さいたま」と呼ばれている。

「さいたま」や他の回路に使用するマイコンは長らく PIC であったが、22 日電で **AVR** が採用され、23 日電、24

日電でもこれを踏襲した。PIC 時代はほとんどのプログラムがアセンブラで書かれていたが、AVR を使うようになってからは C 言語が使われるようになった。

さいたまはたびたび作り変えられてきたが、現在使用しているのは 22 が製作した**さいたま 6 號** である。23 は、さいたまを PC から制御するための装置「Ikebukuro」を導入した (Ikebukuro の資料参照)。24 では基板を作り直し、その際マイコンを **ATmega** に変更したようだ。

4.3 使い方



背面に日周・緯度用に二つのコネクタがあるので、専用ケーブル※を使ってそれぞれのモータに繋ぐ。次に、モータの電力を確保するため、電源装置から **24V** を右面の DC ジャックから供給しよう。

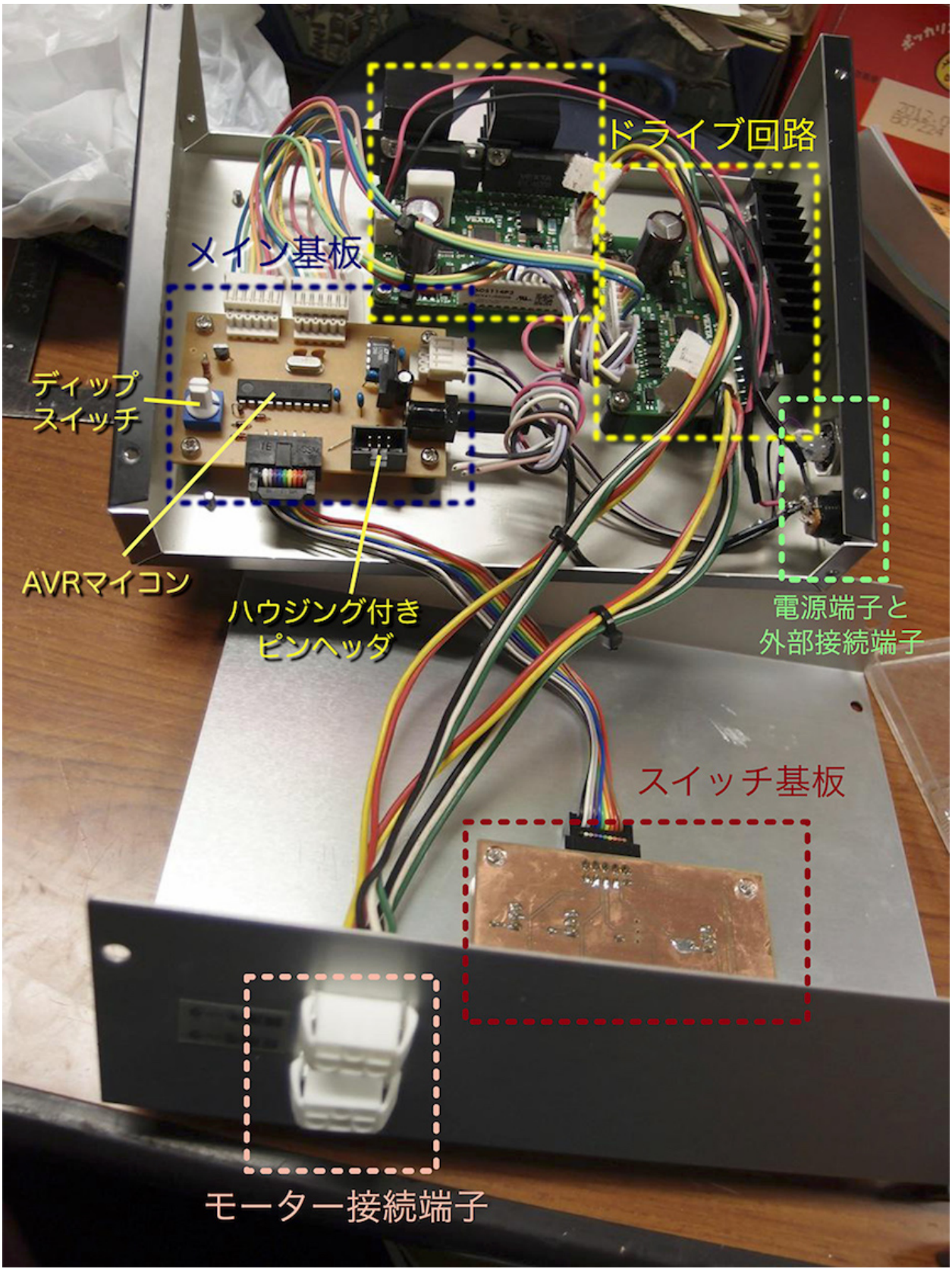
※コネクタは EL コネクタ (日本圧着)6 極、ケーブルは VCTF0.5sq 5 芯

コントローラ前面には 6 つのスイッチがある。現在では外部制御で PC から動かす方がはるかに便利だが、PC が使えない緊急時や急いでいるときはこのスイッチを使えばよい。

ラベルを参照すれば大体の意味は分かることだろう。回転の ON/OFF と方向はそれぞれの軸で独立しているが、回転速度は 2 軸で変えることはできない。**外部制御** `ON/OFF` スイッチは、PC から操作したい場合に ON にする。これが ON になっているときは、さいたま側のスイッチに反応しなくなるので気をつけよう。

4.4 電装

ケースの両側に付いているねじを全て外すと、さいたまの中身を確認できる。


ドライブ回路
メイン基板
ディップ
スイッチ
AVRマイコン
ハウジング付き
ピンヘッダ
電源端子と
外部接続端子
スイッチ基板
モーター接続端子

- メイン基板
  - 画像は 23 引き継ぎのものだが、この翌年に作り変えられているので注意
  - マイコンがスイッチの状態を読取り、モータードライバに CW・CCW パルスを送信する
  - スイッチはあまり重要でなかったため、基板作り変えの際取り除かれた模様
  - 2 × 3 のピンヘッダは、AVR マイコンにプログラムを書き込むためのもの
- モータードライバ基板
  - CW・CCW パルスに応じてステッピングモータに電流を流す
  - 既製品。割と高価だが、部室に未使用の買い置きがある
- スイッチ基板
  - トグルスイッチが配置されている
  - 普通に使っている場合一番壊れやすい部分なのでたまに異常がないか見てあげよう

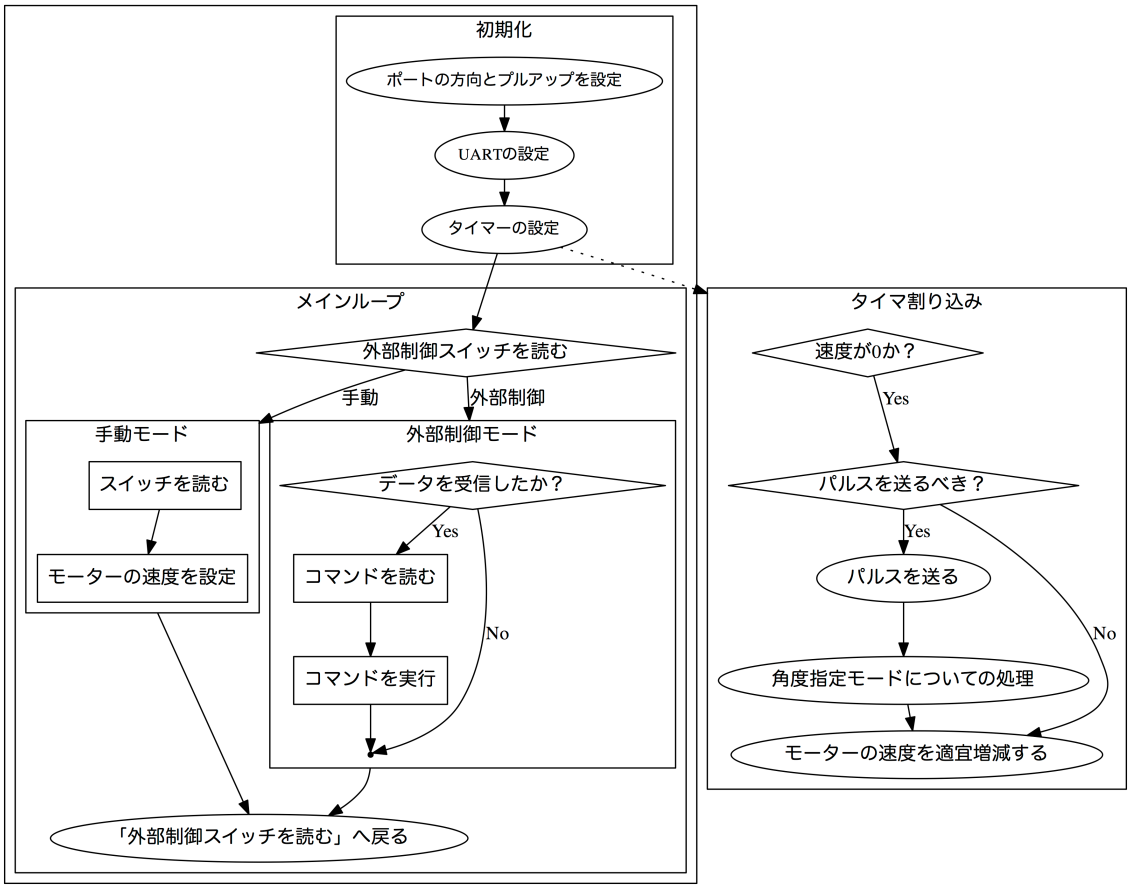
4.5 プログラム

23 代が `nissyuido` ディレクトリ以下に使用したソースコードを残している。`WinAVR` 環境があれば `default` ディレクトリに移動して`$ make` コマンドを打てばビルドされるとのことだ。

ただし、前述の通り **24 代でマイコンを `ATmega328P` に変更** しており、以前のコードを使用できるかは定かでない。作り変えをする際は、24 代の方に問い合わせてみることも検討されたい。

以下のプログラム解説は、23 荒田氏の作成されたドキュメントと同内容である。必要なら氏の PDF や、コード中のコメントなども併せて確認すると良いだろう。

4.5.1 main.c



プログラムの起点となる `main` 関数が入っている。プログラムの大まかな流れは図を参照。

プログラムは初期化 (`main.c` の `init` 関数) の後、メインループ (無限ループ) に入る。ただ、初期化の時にタイマ割り込みを設定しているため、100 マイクロ秒ごとに現在メインループで実行されている内容に関係なく「タイマ割り込み」の内容が実行される。

メインループの処理内容は `main.c` を、「タイマ割り込み」の処理内容は `motordrive.c` を参照されたい。

外部制御コマンドのフォーマットはここで処理している。

4.5.2 motordrive.c

ここが最も重要な部分である。ステッピングモーターのドライブ回路に定期的にパルスを送り、指定した速度でモーターを回す。

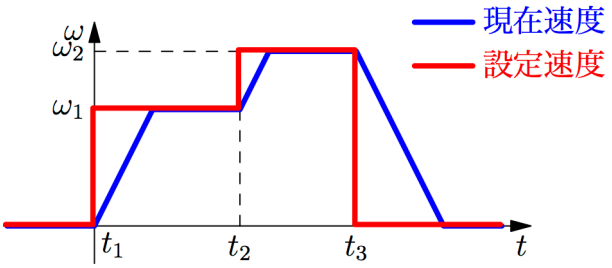
モーターを角速度 $Speed$(コード中では `m->current_speed` $[deg/s]$) で回すにはどうすればよいか考えてみよう。一回のパルスでモーターが回転する角度はモーターとドライブ回路によって決まっており、MotorStep $=$ 0.72deg(コード中では `MOTOR_STEP` $[10^{-2}deg]$) である。

AVR のタイマ機能により、`motordrive` 関数は ControlPeriod $= 100\mu$s (コード中では `CONTROL_PERIOD` $[\mu s]$) 間隔で呼ばれる。 前回パルスを送ってからの経過時間を $n \cdot$ ControlPeriod とする (前回パルスを送ってから n 回目の `motordrive` の呼び出し; n はコード中では `m->count`)。

このとき、簡単な考察により、$n \cdot$ ControlPeriod $\cdot$ Speed $\geq$ MotorStep の時に次のパルスを送ればよいことが分かる。

実際には、速度指定モード・角度指定モードがあったり、速度を徐々に変化させる処理を行っているので、もう少し複雑なプログラムになっている。

「速度を徐々に変化させる処理」であるが、今のところ、現在の速度と目標速度が一致しなければ一定の加速度を加えるという、比較的単純な制御になっている。時間と速度をグラフで表すと図のようになる。赤線が指定した速度、青線が実際の速度である。このように速度を徐々に変化させるようにプログラムの改修を行ったので、もはや**「一旦回転を停止してから回転方向を変更する」必要はない。**



角度指定モードの、速度を下げ始めるタイミングについて。

プログラムが実行されている時点を時刻 t_0 とし、時刻 t_1 にモーターが停止するとする。 モーターの t_0 における角速度を ω_0 とする。単位時間当たりのモーターの角速度の変化を α とする。現在のプログラムでは、`motordrive` が呼ばれるたびに速度を 1deg/s ずつ増減するので、

$$\alpha = \pm \frac{1\mathrm{deg/s}}{\mathrm{ControlPeriod}}$$

である。モーターの時刻 t における角度 (位置) を $\theta(t)$ とする。

時刻 t におけるモーターの角速度と角度はそれぞれ

$$\omega(t) = \omega_0 + \alpha(t - t_0)$$
$$\theta(t) = \theta(t_0) + \omega_0(t - t_0) + \frac{1}{2}\alpha(t - t_0)^2$$

となる。 時刻 t_1 にモーターが停止、すなわち $\omega(t_1) = 0$ より、

$$t_1 - t_0 = -\frac{\omega_0}{\alpha}$$

である。これを $\theta(t_1)$ に代入すると、

$$\begin{aligned}\theta(t_1) &= \theta(t_0) - \frac{\omega_0^2}{\alpha} + \frac{1}{2}\alpha\left(-\frac{\omega_0}{\alpha}\right)^2 \\ &= \theta(t_0) - \frac{1}{2\alpha}\omega_0^2\end{aligned}$$

を得る。

求めたい量は、現在角がどういう値になったら速度を下げ始めるか、その角度である。つまり、$\theta(t_1) - \theta(t_0)$ の値である:

$$\begin{aligned}\theta(t_1) - \theta(t_0) &= -\frac{1}{2\alpha}\omega_0^2 \\ &= \frac{1}{2\left(\frac{1\mathrm{deg/s}}{\mathrm{ControlPeriod}}\right)}\omega_0^2 \\ &= \frac{\mathrm{ControlPeriod}}{2(1\mathrm{deg/s})}\omega_0^2\end{aligned}$$

ステップ数に換算するために両辺を MotorStep で割ると、

$$\frac{\theta(t_1) - \theta(t_0)}{\mathrm{MotorStep}} = \frac{\mathrm{ControlPeriod}}{2(1\mathrm{deg/s})\mathrm{MotorStep}}\omega_0^2$$

を得る。

なお、コード中では $\frac{\mathrm{MotorStep}}{\mathrm{ControlPeriod}}$ に `MOTOR_MAX_SPEED` という名前を与えている。

4.5.3 uart.c

外部とシリアル通信するための関数が記述されている。通信データのバッファリングを行っているが、この仕組みが正常に動いているかは検討の余地がある。

4.5.4 timer.c

`motordrive` 関数を $100\mu s$ 間隔で呼び出すための設定を行う。ぶっちゃけ、`motordrive` 関数をそのままタイマ割り込みハンドラにしても良い気がする。

4.5.5 コンパイルするには

ソースコードを編集したら、書き込む前にコンパイルする必要がある。`default` ディレクトリ以下に `Makefile` が入っているので、`Makefile` の意味が分かる人は利用すると良いだろう。

Makefile の意味が分からない人は、Atmel Studio だか何だか知らないが、適当にプロジェクトを作ってファイルを放り込んでコンパイルすればよろしい。その際、

- マイコンの種類は ATmega328P
- クロック周波数は 16MHz: プリプロセッサの設定をいじって、`F_CPU=16000000UL` が predefined になるようにする。(コンパイラオプションとして`-DF_CPU=16000000UL` が渡されれば OK)
- 言語規格は C99+GNU 拡張 (コンパイラオプションとして`-std=gnu99` が渡されれば OK)

となるように注意する。

4.6 今後の展望

もしもさいたまを作り替えるようなら、もう少し強力なマイコンを搭載すること、センサー (後述) 対応にすること、**通信経路** についてもっとしっかり考えること (`RS-485` にするのか、全部 `RS-232` と USB シリアル通信で統一するのか) が望ましい。

23 代で相対角度指定が実装されたが (現在使われていない)、**絶対角度指定** があると良いだろう。つまり、投影される星空を見ながら日周緯度変を操作するのではなく、「緯度は何度、日周は何月何日何時」という形で指定できるようにする。

絶対角度指定のためには、角度センサーを設置して現在位置を取得するか、一ヶ所にフォトインタラプタなどを設置して初期位置を判別できるようにして、後はステッピングモーターのステップ数で現在位置を把握する、などの方法が考えられる。

いずれにせよ、さいたまを作り直す際に日周緯度変に設置したセンサーを接続することを考慮しておくとよいだろう。

PC 側のソフトウエアだが、PC で操作する以上何らかのメリットが欲しい。速度を柔軟に調節できるようにはなったが、操作性にはまだまだ改善の余地がある。リアルタイムで操作するには、マウスよりもキーボード、欲を言えばタッチパネルでの操作の方がいい。いろいろ工夫してみると良いだろう。

もう一つの方向性として、操作の記録・再生が考えられる。ボタン一つで一本のソフトをまるまる上映できると楽だろう。ただし、ソフトウエアを実装する手間、操作を記録しておく手間に見合うメリットがあるかよく考える必要がある。

第 5 章

日周緯度変 (Ikebukuro)

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/02/26
- 実行に必要な知識・技能: 電子回路
- タスクの重さ: 2: 数日かかる
- タスクの必須度: 3: 年による
- 元資料
  - **日周・緯度変資料**.pdf by 岩滝宗一郎 (nichiden_22)
  - saitama.pdf by 荒田 実樹 (nichiden_23)

5.1 概要

さいたま 6 號には、外部機器 (PC など) を接続して制御することが可能です。これをさいたまの「**外部制御**」と呼んでいます。

埼玉 6 號の外部制御端子は `RS-485/Mini-DIN6` という規格・形状ですが、残念ながらこのままでは PC に直接挿して外部制御することはできません。そこで、`RS-485/Mini-DIN6` と `USB` の変換を行うために製作されたのが、**変換モジュール `Ikebukuro`** (埼玉と繋がっているからしい...) です。

実際に日周緯度変を動かす際に関わりの多い部分なので、使い方や構造などを解説します。

5.2 沿革

外部制御は、埼玉 6 號を製作した 22 代の時点で既に仕様に盛り込まれていた。ただし使用はされておらず、テストもされていない状態だった。続く 23 代で `Ikebukuro` が導入され、**PC を使用しての日周緯度変制御を初めて行った。**

その後の 24〜27 代では日周緯度変を操作するアプリの開発を行なっており、通信には `Ikebukuro` が使われ続けている。

5.3 使い方

さいたま 6 號側面に通信用の端子 (Mini-DIN6) があるので、ケーブルで Ikebukuro の同じ端子に接続する。次に、Ikebukuro の USB 端子 (miniB) と PC の USB 端子を USB ケーブルで接続すれば準備完了である。

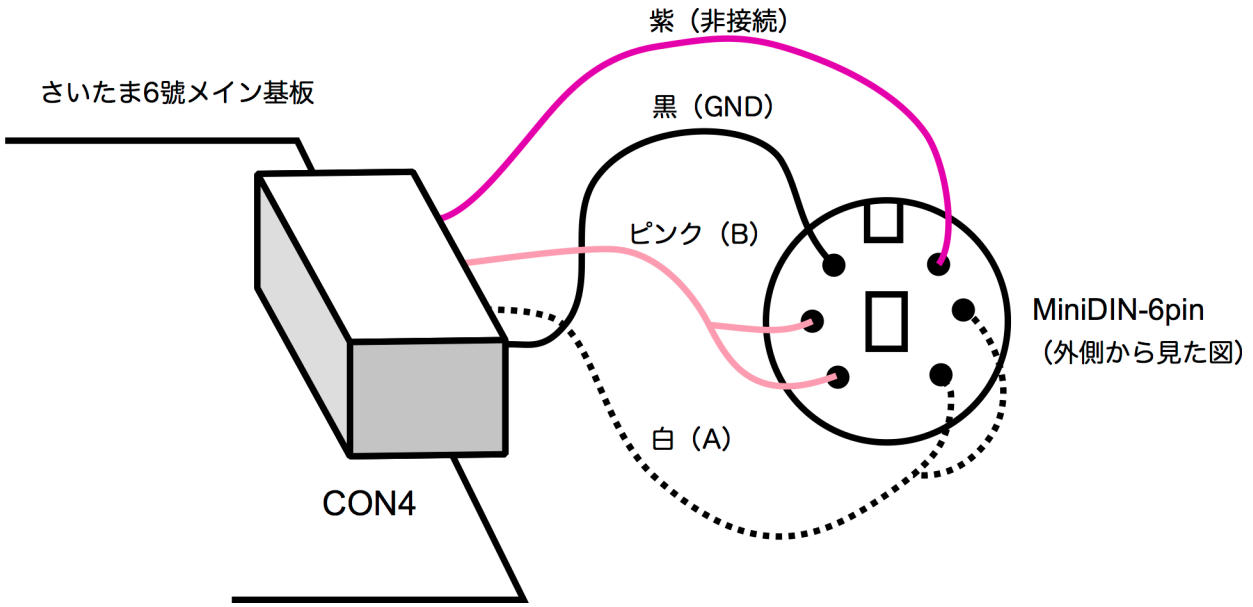
さいたま 6 號の外部制御スイッチを ON にすれば、PC からの信号を待ち受ける状態になる。あとは日周緯度変用のアプリなどを起動してシリアルポートに接続すれば良い。

5.4 電装

5.4.1 通信方式

埼玉 6 號の外部制御端子は、電気的には RS-485 という規格に従っている。RS-485 では、3 本の線 (A,B,GND) で半二重通信 (双方向の通信ができるが、同時に両方向の通信は不可) ができる。ノイズに強く、長距離にも耐えられるらしい。

埼玉 6 號の側部にある Mini-DIN6 コネクタのピンとの対応は、図のようになっている。



5.4.2 内部基板

Ikebukuro の中身は単なる変換モジュールであり、RS-485 と UART の変換を行う IC(LTC485) と、USB と UART の変換を行うモジュール (AE-UM232R) が載っているだけである。

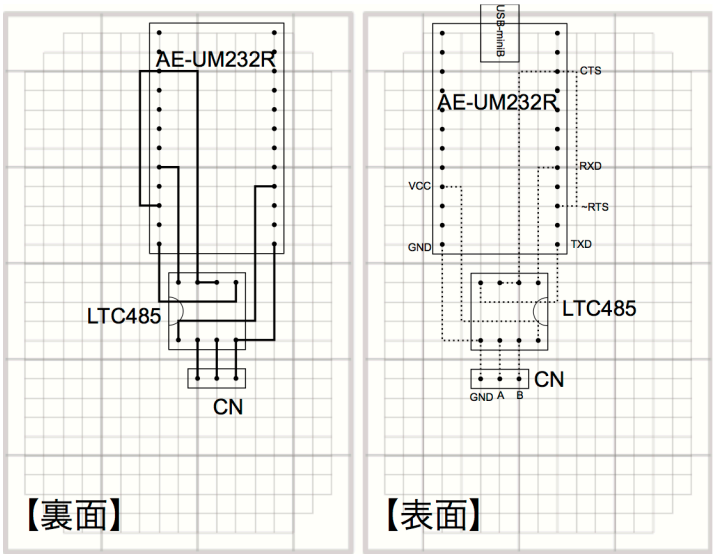
AE-UM232R に搭載されている USB-UART 変換チップ (FTDI 社の FT232R という、定番チップ) の関係で、利

用するには PC 側にドライバが必要な場合がある。FTDI 社の Web ページから入手できるので、必要なら使用する PC にインストールしておこう。

また、このチップは初期状態から設定が書き換えられており、`InvertRTS=1` となっている。この設計は FTDI 社謹製の `FT_PROG`(Windows 用) というツールで設定できる。

`RS-485` は半二重なので、通信方向の切り替えが必要である。これは、シリアルポートの RTS ピンを使用して行っている。

`Ikebukuro` の基板の配線を図に載せる。なお、図には反映していないが、RTS の状態確認用に LED を実装している。



`AE-UM232R` は IC ソケットに差さっているので、容易に `Ikebukuro` から取り外すことができる。他の回路のテストで USB シリアル変換モジュールを使いたい時は、`Ikebukuro` から `AE-UM232R` を取り外して使うと良いだろう (実際、23 代で惑星投影機の基板のテストに利用した。最終的には XBee で通信するのだが、デバッグ時は「信頼と安定の」FT232R を利用しようというわけだ)。

5.5 プロトコルとコマンド

外部制御モードでは PC からさいたま 6 號に指令 (コマンド) を送る。ここでは、コマンドを表す文字列のフォーマットを述べる。

このフォーマットを変更したいと思ったら、さいたま側のプログラムの `main.c` と PC 側のプログラムをいじればよい。

5.5.1 各コマンドに共通するフォーマット

位置	0	1	2	3	4	...
内容	$	W	*addr*	*len*	*cmd*	...

- *addr*: 機器のアドレス (16 進 1 桁; ‘0’ ～ ‘9’, ‘A’ ～ ‘F’)
  - メイン基板上のディップスイッチで設定した値と一致させる
  - 現在は 0 が使われている
  - 複数の機器を繋いだ時用のアドレスだが、使うことはないだろう
- *len*: 以降のバイト数 (16 進 1 桁; ‘0’ ～ ‘9’, ‘A’ ～ ‘F’)
- *cmd*: コマンドの種類

5.5.2 速度指定コマンド

位置	...	3	4	5	6	7..10	...
内容	...	7	V	*motor*	*dir*	*speed*	...

- *len*: 以降のバイト数 = ‘7‘ (7 byte)
- *cmd*: コマンドの種類 = ‘V‘
- *motor*: モーター (日周: ‘D’, 緯度: ‘L’)
- *dir*: 回転方向 (時計回り: ‘+’, 反時計回り ‘–’)
- *speed*: 回転速度 [deg/s] (16 進 4 桁; ‘0’ ～ ‘9’, ‘A’ ～ ‘F’)

5.5.3 角度指定コマンド

位置	...	3	4	5	6	7...12	13...16
内容	...	D	P	*motor*	*dir*	*angle*	*speed*

- *len*: 以降のバイト数 = ‘D‘ (13 byte)
- *cmd*: コマンドの種類 = ‘P‘

- *motor* モーター (日周:‘D’, 緯度:‘L’)
- *dir* 回転方向 (時計回り:‘+’, 反時計回り ‘−’)
- *angle* 回転角度 [10^{-2}deg] (16 進 6 桁; ‘0’ 〜 ‘9’, ‘A〜 ‘F’)
- *speed* 回転速度 [deg/s] (16 進 4 桁; ‘0’ 〜 ‘9’, ‘A’ 〜 ‘F’)

5.5.4 コマンド実例集

以上がさいたま外部制御コマンドの仕様だが、これだけでは少々分かりにくいはずなのでいくつか実例を挙げておく。

- 緯度モーターを 3000deg/s で時計回りに回す -> `$W07VL+0BB8`
- 日周モーターを 1800deg/s で時計回りに 90000deg 回す -> `$W0DPD+8954400708`

5.5.5 PC 側のプログラム

PC 側からは、Ikebukuro が仮想 COM ポートに見えるので、その COM ポートに対して上に述べたコマンドを書き込めば良い。この辺りの具体的な話は外部制御アプリの資料に譲る。

第 6 章

日周緯度変 (外部制御アプリ)

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/03/04
- 実行に必要な知識・技能: シリアル通信、C#
- タスクの重さ: 3: 数週間
- タスクの必須度: 4: 毎年やるべき

6.1 概要

日周緯度変の外部制御には、大きな可能性があります。PC を使える以上かなり複雑な操作も自動化できるためです。

ただし、本番で日周緯度変を動かすのは人間である部員たち。それも、プログラムの挙動を知っている日電員だけが操作するとは限りません (26・27 と、かごしいにも日周緯度変操作を手伝って貰っています)。

誰にでも使いやすい日周緯度変外部制御アプリを作るために注意すべきことを書きます。

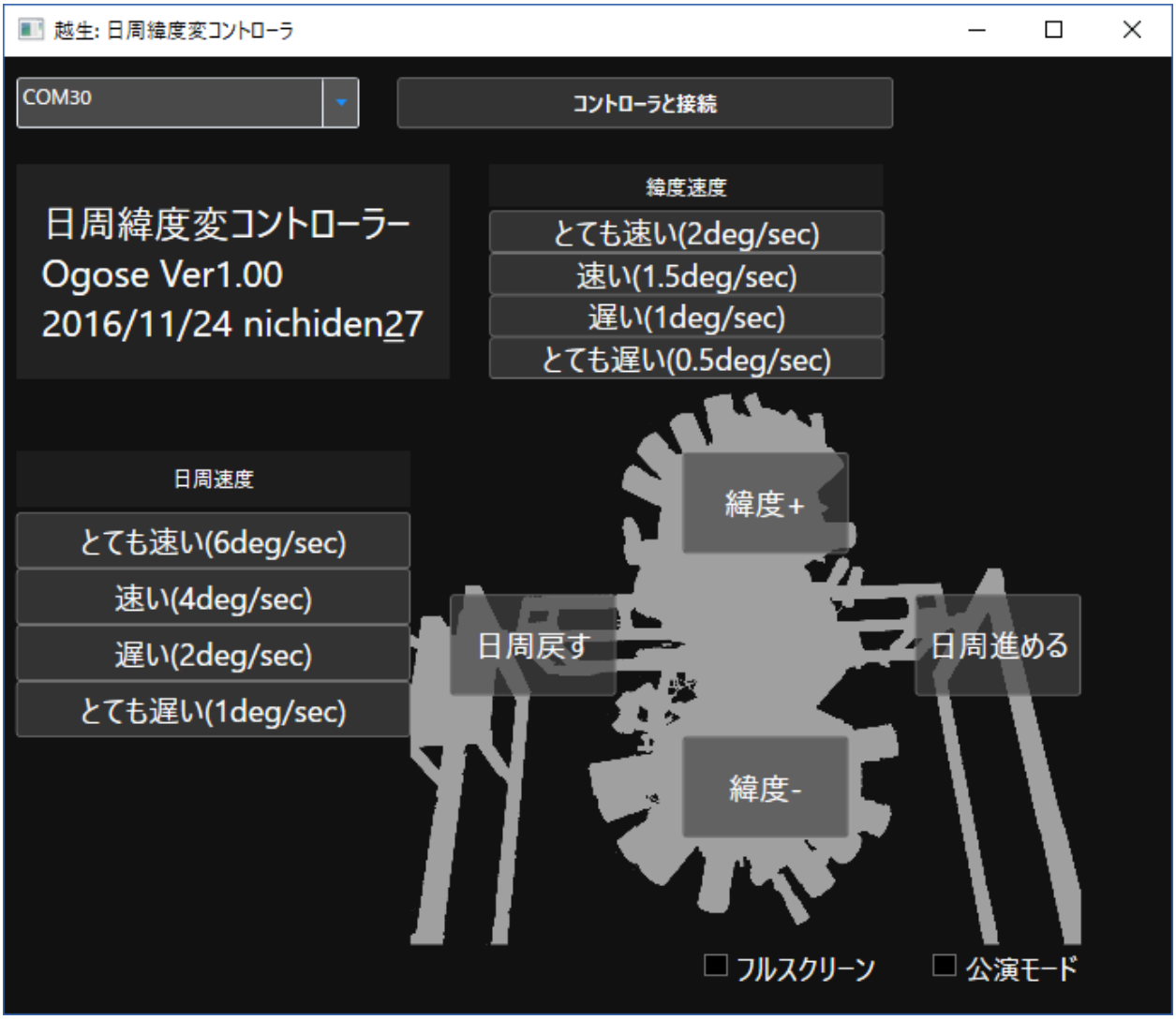
6.2 沿革

過去に開発されたアプリケーションが多数にのぼるため、外部制御アプリの歴史に分離する。

6.3 Ogose の特徴と使い方

`Ogose` は、27 代日電で作成した、2017/03/01 現在最新の外部制御アプリである。

特徴は沿革の方に文章で書いたので、ここでは簡単に箇条書きするにとどめる。



- 各軸 4 段階で速度を指定
- 回転中の速度変更が可能
- 回転の開始/停止はトグルボタン (押すたびに ON/OFF が切り替わる)
- フルスクリーン切り替えボタン
- 誤操作防止用の「公演モード」
- キーボードでも操作可能
- フリーソフトを使ってゲームコントローラに操作を割り当て可能

6.3.1 基本操作

起動したら、まずは最上部からシリアルポートを選んで**接続ボタン** を押す。さいたまを Ikebukuro を介して繋いだだけの状態ならポートは一つしか出ないことがほとんどだが、複数出たとしても一つずつ試してモーターが動くかどうかで判断すればよい。

未接続のままコマンドを送る操作をした場合は、エラーメッセージとともに送信するコマンドが表示される。接続していないことに気づけるようにする一方、わざと未接続にすることでコマンド送信のデバッグも可能だ。

実際に日周緯度変を動かすときは、左側と上側にある**速度切り替えボタン** から速度を選び、動かしたい方向のボタンをクリックする。起動後に一度も速度を選んでいないときは、各軸とも「速い」速度が選択される。

回転開始ボタンは押すと「停止」に表示が切り替わり、再度押すと回転が停止する。反対方向のボタンは「停止」するまでグレーアウトしてクリックに反応しない。これは、ボタンの表示がおかしくなるのを防ぐためである。

フルスクリーンボタン にチェックするとフルスクリーンになる。画面全体が黒背景になるので、本番中はフルスクリーンが望ましい。**公演モード** ボタンは、ON にすると警告が表示され日周を進めることしかできなくなる。

モーターの回転はキーボード操作でもできる。

- W: 緯度 +
- A: 日周戻す
- S: 緯度-
- D: 日周進める

と対応している。PC ゲーマーには馴染みのある配置かと思う。

Ogose のコードの解説は、たいへん長いので別記事とする。

-> *Ogose の実装解説*

6.4 通信プログラム

6.4.1 シリアル通信の基本

Ikebukuro の記事にある通り、パソコン側からは ` はシリアルポートとして見える。よって、シリアルポートにアクセスするプログラムを書けばよい。

ポート名は、Windows であれば COM1 や COM4 のように、「'COM'+ 数字」の形式である。Mac OS X や Linux のような UNIX 環境であれば、/dev/tty.usbserial-A5017ABT である。

シリアルポートの設定は、さいたま 6 號側のプログラム中に記述してある設定と一致させなければならない。現時点での設定は、

- baud rate: 2400
- parity: none
- char bits: 8
- stopbit: 1

である。baud rate はこの値である必然性はないので、変えても良い (さいたま 6 號のプログラムも一緒に変更すること！)。

通信経路の途中で RS485 を使っている関係上、シリアルポートには読み取りと書き込みのいずれか一方しかできない。

RS485 の通信方向の切り替えには、シリアルポートの RTS 端子を使う。RTS が HIGH になるとパソコン側から日周緯度変側への通信ができ、LOW になると逆方向の通信ができる。コマンドを送信する直前に RTS を HIGH にし、終了したら LOW に戻すといいだろう。

6.4.2 .Net でのシリアル通信

.Net Framework を使う場合は、System.IO.Ports 名前空間以下にある SerialPort クラスを利用すると便利である。Mac OS X や Linux では .Net のオープンソース実装である Mono を利用できるが、Mono においても同様である。

以下では C#のコード例を示す。

まず、ソースコードの冒頭に以下の文を書いておくと便利であろう:

```
using System.IO.Ports;
```

通信を開始する前に、まずポートを開く：

```
SerialPort port = new SerialPort(
  portName:portName,
  baudRate:2400,
  parity:Parity.None,
  dataBits:8,
  stopBits:StopBits.One
);
port.Open();
```

この後、port.IsOpen により、ポートを開くのに成功したか確認しておこう。

実際に通信するには SerialPort.Write メソッドを使う。プログラムを終了する際には、port.Close() によりポートを閉じておく。

6.4.3 他の言語でのシリアル通信

日電では 23 から C#で外部制御アプリを開発してきたが、他の各種の言語でもシリアル通信を扱えるので記しておく。

- C, Lua: librs232
- Lua: LuaSys
- Python: pySerial
- Ruby: ruby-serialport
- Java: Java Communications API / RXTX(Windows)
- JavaScript(Node.js): serialport

それぞれの使い方やインストール方法についてはググって欲しい。

6.4.4 ユーザインターフェース

日周緯度変をある程度誰でも動かせるようにするには、GUI は欠かせない。歴代日電では、GUI 開発に Windows Form アプリケーションや WPF を用いてきた。もちろん他にも GUI のフレームワークは星の数ほどあるが、時間も限られている以上、定番で枯れた技術を使うのが無難ではなかろうか。

一つの可能性としては、最近イケイケの Web アプリがある。Chrome と JavaScript をベースにデスクトップアプリを実現する Electron など、ここ最近急速にシェアを伸ばしている技術もある。もしあなたがそういった技術を得意としているなら、乗り換える価値はあるかもしれない。

6.5 今後の展望

コントロールの操作性 にはまだまだ改善の余地がある。本番でリアルタイムの操作をするには、マウスよりもキーボードがいいし、タッチパネルやゲームコントローラーといった馴染みのある操作系も役に立つだろう。ゲームコントローラーは 27 ではフリーソフトでキー操作と無理やり関連付けしたが、DirectInput を使えば自前でも利用できる。

さらなる方向性として、**操作の記録・再生** が考えられる。23 や 25 で行ってきたことを発展させ、ボタン一つで一本のソフトをまるまる上映できるようになれば楽だろう。

ただし、当然ソフトの指示は毎年変わるので、開発の負担は増加する。ライブ解説ではタイミングを人力で判断せねばならず、想定外の事態も起こりうる以上、全自動化への道は平坦ではない。コストに見合うメリットが得られるような仕組みを、ぜひ考案してほしい。

第 7 章

日周緯度変 (外部制御アプリの歴史)

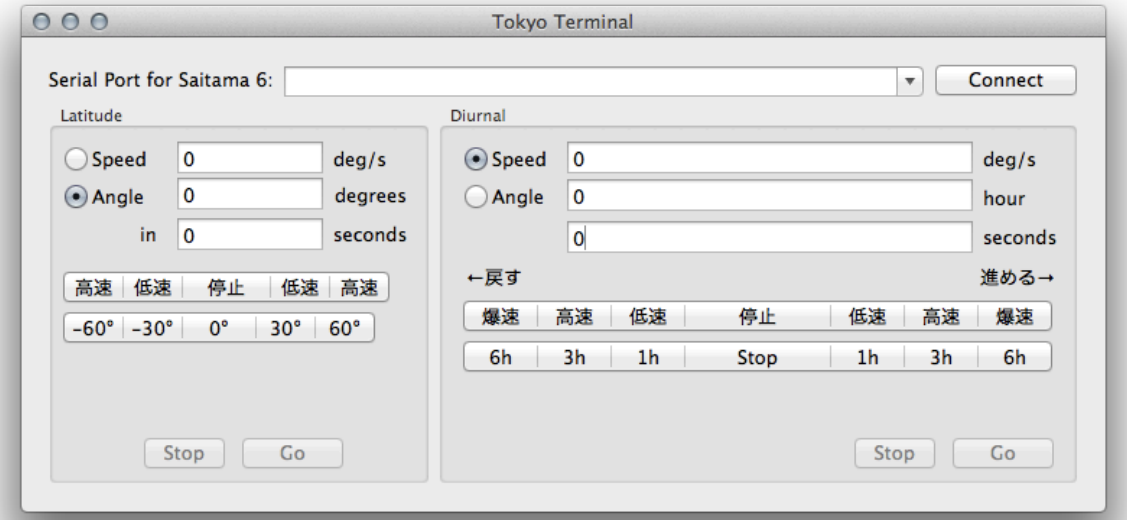
- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/03/04
- 実行に必要な知識・技能: 特になし
- 難易度: 2: 少しやれば可能
- 情報の必須度: 2: 知ってると楽
- 元資料
  - 引き継ぎと技術的補足：日周緯度変外部制御ユーザインタフェース.docx by 紺野雄介 (nichiden_23)
  - ふじさわ readme.docx by 池下 氏 (nichiden_24)
  - 25 の日周・緯度変について.docx by 伊藤太陽 (nichiden_25)

7.1 概要

日周緯度変 (外部制御アプリ) から歴代外部制御アプリケーションの紹介を分離した記事です。

現在は実装されていない機能があったりと、それぞれに特徴があるので、把握しておくと今後の改善に繋がるかもしれません。

7.2 Tokyo Terminal(2012)



23 代荒田氏が制作した。外部制御は 23 代で初使用されたが、`Tokyo Terminal` はそのテスト用に書かれたものである。

氏の環境が Mac OS だったため、Mac でしか動作しない。23 代の作業データの**日周緯度変外部制御/**`Saitama6Brain/`以下にソースファイルやアプリ本体があるが、最新の Mac OS では対応していない旨が表示され起動しなかった。

これ以降の外部制御アプリは全て Windows 向けに開発されたものだ。Mac 向けに開発する必要に迫られることがもしあれば、`Tokyo Terminal` のコードが参考になるかもしれない (Mac 開発の経験があれば一から書いた方が早い可能性も大いにあるが...)。

7.3 NisshuidohenController2(2012)

23 代紺野氏の作。`Tokyo Terminal` が Mac 専用だったため、Windows 版として開発された。開発言語は C#で、Windows Form アプリケーション (`System.Windows.Forms` 名前空間を使用) である。

`Tokyo Terminal` と共通の機能が多いが、大きな違いは**操作の記録・再生ができる** ことである。「速度指定」か「角度指定」かを選択して「記録」ボタンを押すと右のスペースに送るコマンドが表示され、同時に `Instruction.txt` というファイルにも保存される。

あらかじめ必要なコマンドを記録しておいて面倒な操作なしに再実行できる画期的な機能である... と言いたいところだが、**表示されるのはコマンドだけ** (数値は 16 進数) なので、肝心の再生部分が手軽に利用できるとは言い難い。

「高速」「低速」のボタンで出る角速度は以下の数値に固定されている。なお、上半分で入力する角速度はモーターのもの、以下の角速度はギアによる減速後のかごしいのものなので注意。

- 日周高速ボタン: 4 deg/s
- 日周低速ボタン: 1 deg/s
- 緯度高速ボタン: 1 deg/s
- 緯度低速ボタン: 0.5 deg/s

この数値は後発の `Fujisawa`、`Chichibu` でも同じものが使われている。

当時の日周緯度変は相応のスキルのある日電員が操作しており、自分たちで使うための最小限の機能を盛り込ん

だという印象だ。実際、本番中に使用したのは「高速」などのボタンだけだったという。

将来これを使うことはなさそうだが、速度・角度・方向に対応するコマンドを表示してくれるので、デバッグ用の計算機にはなるかもしれない。

7.4 Fujisawa(2013)

24 池下氏によるもの。引き続き C#による Windows アプリだが、UI のフレームワークに `Windows Presentation Foundation(WPF)` を使用している。

WPF の詳細についてはググって欲しいが、デザインとコードを分けて書くことができるというのが大きな利点である。つまり、内部の動作を崩さずに見た目だけをいじり倒せるのだ。その甲斐あってか、23 の UI に比べデザイン面が大きく改善した。コマンド文字列を生成するコードは `fujisawa/NisshuidohenController.cs` でクラスにまとめて定義されている。

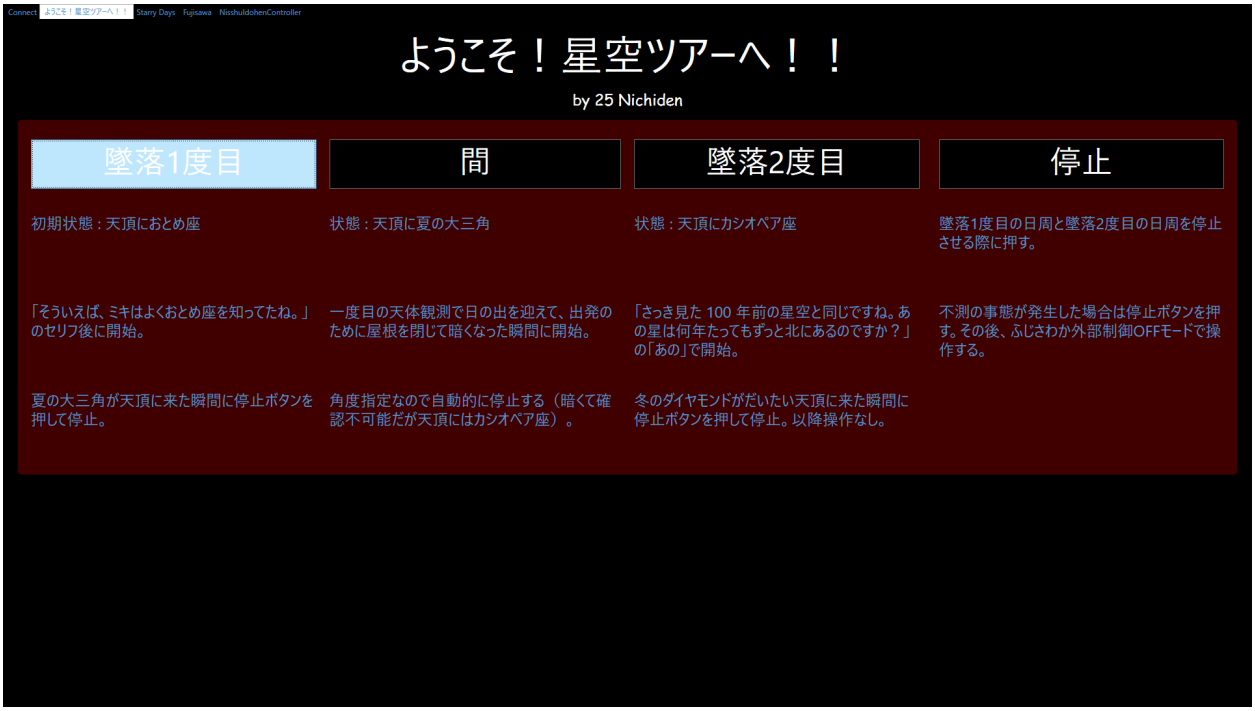
日周緯度変に PC を使うとき、一番問題になるのは画面が光ることだ。PC は画面も大きいし、そのために星がかき消されてしまいかねない。`Fujisawa` は、**黒基調の画面** にすることで PC の光害を抑制している。

使い方は見れば分かると思うが、「びゅーん」が高速回転、「のろのろ」が低速回転だ。また、本番で画面を暗くしているとマウス操作が大変なので、日周はキーボードでも操作できる。`C`: 高速逆転、`V`: 低速逆転、`B`: 停止、`N`: 低速順回転、`M`: 高速順回転 である。

総じて 24 日電の雰囲気がよく出ており大変分かりやすいものの、回転方向が少し把握しにくい。なお、アプリ名

称は往時の天文部の「備品などに駅名をつける」習慣に則ったもので、「ふじさわ」は制作者の地元であるらしい。

7.5 Chichibu(2014)

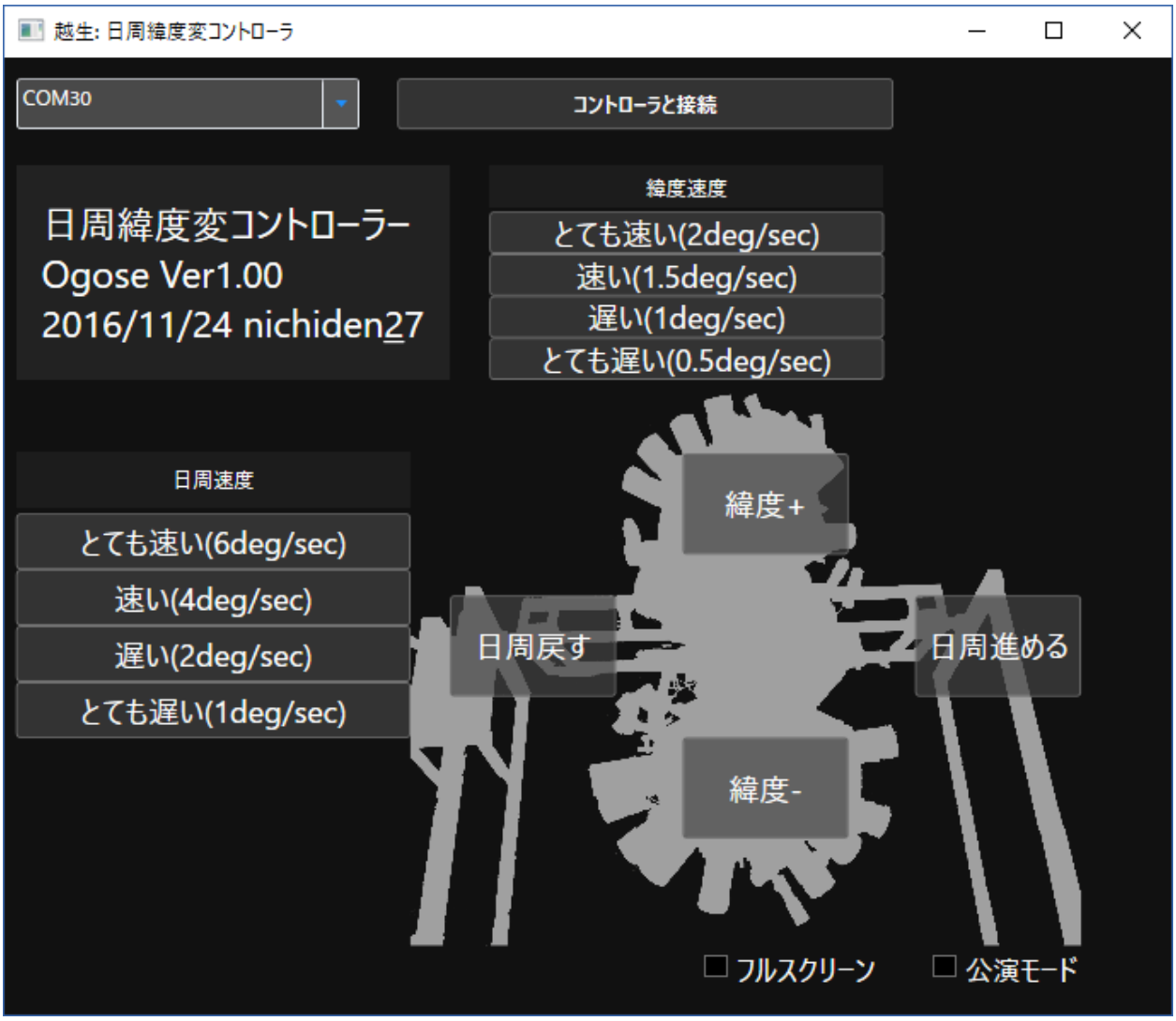


25 伊藤氏が開発したアプリ。他にない特色として、25 ソフト専用のモードが用意され、ボタンを押すだけでシナリオで要求された動きを実現できることがある。

また、23 の `NisshuidohenController2` と 24 の `Fujisawa` の画面もそのまま移植され、それぞれの機能が利用できる。本番でも、ライブ解説時には `Fujisawa` を使っていたようだ。

フルスクリーン状態で起動することで、以前より更に画面の光漏れを抑えている。ただし、ソフト内に終了ボタンがないうえタイトルバーも見えないので、プログラムを終了させる際は `Alt+F4` を押すなどショートカットキーを使うしかない。

7.6 Ogose(2016)



27 日電の伊東が開発した。これまでの UI の問題点を洗い出した上で、改善すべく様々な変更を加えている。デザイン部分 (`MainWindow.xaml`) は一から作り直したが、コマンド文字列生成は `NisshuidohenController.cs` を継続使用した。

指定できる速度が各軸 4 段階に増えたことで、多彩な演出が可能となった。また、速度指定のボタンを回転スタート/ストップボタンとは別に用意したため、回転を止めずとも速度を変更できる。

ウィンドウモードとフルスクリーンモードをボタンで切り替えられる機能も実装した。また、フルスクリーンボタンの横にある「公演モード」ボタンは、使用できる機能を「日周進める」に限定し誤操作を防止する機能である。キーボード操作などにより意図しない状態になるバグが存在するので注意。

思いがけないことに、ボタンを十字に配置したことで、ゲームコントローラーのボタンと同様の配置となった。本番ではゲームコントローラーを実際に使用し、慣れていない人でも操作ができるという恩恵があった。

実装についてなど、詳細は外部制御アプリの記事に示すこととする。

第 8 章

日周緯度変 (Ogose の実装解説)

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/03/06
- 実行に必要な知識・技能: Windows GUI 開発、C#、WPF、Visual Studio の操作
- 難易度: 3: 練習・勉強が必要
- 情報の必須度: 3: 必要な場合がある

8.1 概要

日周緯度変 (外部制御アプリ) から Ogose の実装解説を分離した記事です。

Ogose のソースコードを読む、あるいは書き換える際に参考にしてください。

8.2 ファイル構成

Ogose のソースファイル等は、Ogose フォルダ内に入っている。いかにファイル・ディレクトリ構成の抜粋を示す。

```
Ogose
├- Ogose
|    ├- App.config
|    ├- App.xaml
|    ├- App.xaml.cs
|    ├- MainWindow.xaml
|    ├- MainWindow.xaml.cs
```

```
|   ├- NisshuidohenController.cs
|   ├- Ogose.csproj
|   ├- Properties
|   |   └- (省略)
|   ├- bin
|   |   ├- Debug
|   |   |   ├- Ogose.exe
|   |   |   └- (省略)
|   |   └- Release
|   |       ├- Ogose.exe
|   |       └- (省略)
|   ├- main_projector_27_w.png
|   └- obj
|       └- (省略)
└- Ogose.sln
```

見た目には複雑で身構えてしまうかもしれない。ただ、Visual Studio(以下 VS) でプロジェクトを作成すると自動で生成されるファイルがほとんどで、実際に開発者が触るべきファイルは多くない。

Ogose 直下には Ogose.sln がある。これは「ソリューション (開発プロジェクトをまとめたもの)」の状態を管理している。sln ファイルをダブルクリックするか、VS 内の読み込みメニューで選択してあげれば Ogose の各ファイルを閲覧できる。

Ogose の下に更に Ogose ディレクトリがあり、この中にソースコードなどが収められている。このうち、開発で実際に触ったのは App.xaml MainWindow.xaml MainWindow.xaml.cs NisshuidohenController.cs の四つのみである。

Ogose/Ogose/bin/以下には、ビルドで生成された.exe ファイルが格納される。Debug と Release は適当に使い分ければいい。exe の他にも様々なファイルが吐き出されるが、基本的には Ogose.exe 単体で動作する。

以下、ソースコードを簡単に解説する。WPF 開発の基本的な知識全てに触れるとは限らないので、よく理解できない部分はググるなどして補完してもらいたい。

8.3 App.xaml

App.xaml や App.xaml.cs の内容は、GUI のみならずアプリケーション全体に適用される。何も書かなくても問題ないが、Ogose では画面デザインに関する記述を全てこちらに分離した。*MainWindow.xaml* が長くなりすぎないようにするのが目的である。

XAML(ざむる) は、XML をベースとした GUI の記述言語である。XML のタグを用いて階層状に指示を書けるようになっている。なお、<>で囲まれた単位は「タグ」とも「要素」とも言うが、GUI の要素と混同する危険があるので、ここでは「タグ」に統一する。

<Application>タグには色々とおまじないが書いてあるが、気にする必要はない。その下の<Application.Resources>内からがコードの本番だ。

8.3.1 ブラシ

```
<!--  App.xaml -->
<SolidColorBrush x:Key="WindowBackground" Color="#FF111111"/>
<SolidColorBrush x:Key="ButtonNormalBackground" Color="#AA444444"/>
<SolidColorBrush x:Key="ButtonHoverBackground" Color="#FF334433"/>
<SolidColorBrush x:Key="ButtonNormalForeground" Color="White"/>
<SolidColorBrush x:Key="ButtonDisableBackground" Color="#AA222222"/>
<SolidColorBrush x:Key="ButtonDisableForeground" Color="SlateGray"/>
<SolidColorBrush x:Key="ButtonNormalBorder" Color="#FF707070"/>

<LinearGradientBrush x:Key="TextBoxBorder" EndPoint="0,20" MappingMode="Absolute"␣
↪StartPoint="0,0">
    <GradientStop Color="#ABADB3" Offset="0.05"/>
    <GradientStop Color="#E2E3EA" Offset="0.07"/>
    <GradientStop Color="#E3E9EF" Offset="1"/>
</LinearGradientBrush>
```

ブラシ は、色などのデザインに名前 (x:Key) をつけて使い回せるようにしたものである。各色の役割が明確になるし、後からの変更も楽なので積極的に利用した。SolidColorBrush は単色のブラシ、LinearGradientBrush はグラデーションのブラシである。

配色が気に入らなければ、ここの色指定を変えれば良い。色は名称で指定しても良いし (色一覧)、Web などでお馴染みの 16 進数で更に細かく決めることもできる。ここでは ARGB という RGB に加えアルファ値 (透過度) も指定する方式で書いているので注意。例えば#FF111111 なら、不透明で{R,G,B} = {17,17,17}の色を指す。

8.3.2 コントロールテンプレート

コントロールテンプレート は、コントロール (ボタンやテキストエリアなど) のテンプレートである。この中にボタンなどの見た目を書いておくと使い回しが効く。今あるコントロールテンプレートとその用途は以下の通り。

- “NormalToggleButton” ... 日周緯度変回転用のトグルボタン
- “ComboBoxToggleButton” ... 接続するシリアルポートを選択するコンボボックス

また、<ControlTemplate.Triggers>タグ内で「トリガー」を指定できる。トリガーは、特定のイベントが起きたら動的にコントロールの見た目を変更する機能だ。マウスでポイントした時やクリックした時に色が変わると、操作の結果がユーザーに視覚的に伝わる。

```
<!--  App.xaml -->
<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="true">
        <Setter TargetName="InnerBackground"  Property="Fill" Value="#FF222288" />
    </Trigger>
    <Trigger Property="IsChecked"  Value="true">
        <Setter Property="Content" Value="停止" />
        <Setter TargetName="InnerBackground"  Property="Fill" Value="#FF111144"/>
    </Trigger>
    <Trigger Property="IsEnabled" Value="false">
        <Setter TargetName="Content" Property="TextBlock.Foreground" Value="
↪{StaticResource ButtonDisableForeground}"  />
        <Setter TargetName="InnerBackground" Property="Fill" Value="{StaticResource␣
↪ButtonDisableBackground}"  />
    </Trigger>
</ControlTemplate.Triggers>
```

例として、"NormalToggleButton"のトリガー定義を紹介する。マウスポインタが乗った時、Checked(ON) 状態になった時でそれぞれ”InnerBackground” の色を変更するようになっている。Property="IsEnabled"は、ボタンが有効 (=操作できる) かを示しており、これが false の時は、文字・背景の色をグレー調にしてクリックできないことをアピールする。

8.3.3 スタイル

スタイル には、要素の外観を定義できる。前項のコントロールテンプレートに比べ機能が制限され、より個別の要素に対して用いる。

スタイルの適用の仕方はいくつかある。**TargetType** に要素の種類を入れると、同じ種類の要素全てに適用される。以下は Window の見た目を指定している例。

```
<!--  App.xaml -->
<Style TargetType="Window">
    <Setter Property="Background" Value="{StaticResource WindowBackground}" />
    <Setter Property="Height" Value="600" />
    <Setter Property="MinHeight" Value="600" />
    <Setter Property="Width" Value="700" />
    <Setter Property="MinWidth" Value="700" />
</Style>
```

<Setter>タグはプロパティを操作するために使う。Property にプロパティの名前、Value に値を入れるだけである。Value は実際の値でもいいし、ブラシなど他で定義したリソースを与えてもよい。

<Setter>の中には更に様々な機能を持ったタグを入れられる。<ControlTemplate>が入っていることもあ

るし、<Style.Triggers>タグでトリガーを設定することもできる。複雑な使い方は筆者もよく把握していないので、頑張ってググって貰いたい。

もう一つのスタイル適用方法は、**x:Key プロパティ** を用いることだ。<Style>タグに x:Key="hogefuga"のように分かりやすい名前をつけておく。

```
<!--  App.xaml -->
<Style x:Key="DiurnalPlusButton" TargetType="ToggleButton" BasedOn="{StaticResource␣
↪ToggleButton}">
    <Setter Property="Content" Value="日周戻す" />
    <Setter Property="FontSize" Value="18" />
</Style>

<Style x:Key="DiurnalMinusButton" TargetType="ToggleButton" BasedOn="{StaticResource␣
↪DiurnalPlusButton}">
    <Setter Property="Content" Value="日周進める" />
</Style>
```

そして、適用したいボタンなどに Style="{StaticResource hogefuga}"などと指定すれば該当する x:Key を持つスタイルが適用される。

```
<!--  MainWindow.xaml -->
<ToggleButton x:Name="diurnalPlusButton" Style="{StaticResource DiurnalPlusButton}"␣
↪Grid.Row="2" Grid.Column="0"
              Command="{x:Static local:MainWindow.diurnalPlusButtonCommand}" />
```

上の App.xaml のコードでは、**スタイルの継承** という機能も活用している。BasedOn プロパティに基にしたいスタイルの x:Key を指定すると、そのスタイルの中身を引き継いだり、部分的に書き換えたりできる。

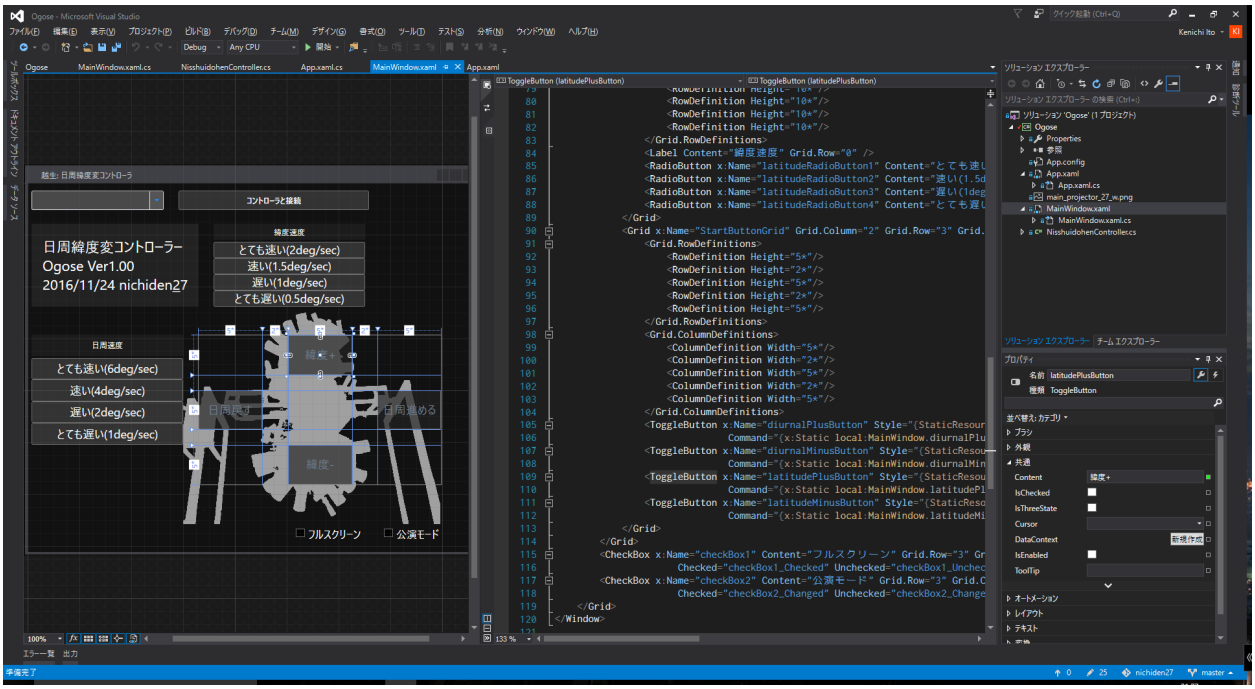
例えば、"DiurnalMinusButton"スタイルは"DiurnalPlusButton"スタイルを継承したので、FontSizeについて再度記述する必要がない。一方で、ボタンに表示する文字は変更したいので、Content を書き換えている。

8.4 MainWindow.xaml

メインのウィンドウの構造を記述する。といっても Ogose には一つしかウィンドウがないので、配置を変えたい場合はこれを編集すればいい。UI のデザインについてもこの中に書けるが、たいへん長いので *App.xaml* に移した。

8.4.1 編集方法について

ウィンドウの見た目は XAML のコードだけで自在に操れるが、VS ではより便利に、実際の画面をプレビューしながらドラッグ&ドロップで編集することもできる。



GUI での編集は手軽で初心者にも扱いやすいが、コードが自動生成されるので手で書くよりも読みにくくなりがちだ。また、数値を細かく決めたい場合はコードを直接編集した方が早い。図のように画面プレビューとコードは並べて表示できるので、双方の利点を使い分けるとよかろう。

8.4.2 グリッド

WPF のレイアウト要素はいくつかあるが、Ogose では<Grid>タグを使ってレイアウトしている。**グリッド** は、画面を格子状に分割してその中に要素を配置していくことができる。いちいち行や列を定義せねばならず面倒だが、サイズを相対的に決められるので、ウィンドウを大きくしたときボタンも拡大されるというメリットがある。

```
<!-- MainWindow.xaml -->
<Grid x:Name="MainGrid">
    <Grid.RowDefinitions>
        <RowDefinition Height="1*"/>
        <RowDefinition Height="30"/>
        <RowDefinition Height="40*"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="1*"/>
```

```
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*"/>
        <ColumnDefinition Width="60*"/>
        <ColumnDefinition Width="20*"/>
        <ColumnDefinition Width="20*"/>
        <ColumnDefinition Width="1*"/>
    </Grid.ColumnDefinitions>
    <Grid x:Name="HeaderGrid" Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="9*"/>
            <ColumnDefinition Width="1*"/>
            <ColumnDefinition Width="13*"/>
            <ColumnDefinition Width="7*"/>
        </Grid.ColumnDefinitions>
```

上のコード片は、グリッドを定義している例である。一意の x:Name を付けて<Grid>を宣言したら、<Grid.RowDefinitions>で行を、<Grid.ColumnDefinitions>で列を定義する。

グリッドの使い方

それぞれの中に行・列を欲しいだけ並べれば良いのだが、**高さや幅の指定** にポイントがある。数値のみを書くとピクセル数を表すが、**数値*** とすると相対サイズを表せるのだ。例えば、Height="1*"の行と Height="2*"の行だけがある場合、グリッドは 1:2 の比率で分割される。

また、コード例では使っていないが Auto を指定すると、中に配置した子要素のサイズに合わせてくれる。ピクセル指定、相対指定、Auto 指定は混ぜて書いても問題ない。画面プレビューで行や列を分割した場合、サイズが単純な数値にならないので適宜コード側で修正するといいだろう。

グリッドの中に要素を置く 時は、画面プレビュー上で設置したい場所に動かすだけで良い。ただし、グリッドは入れ子にすることもでき (コード例では MainGrid の下に HeaderGrid を入れてある)、意図した階層に置けないことも多々ある。その場合は、望みの階層に要素の定義をコピペした上で、Grid.Row と Grid.Column プロパティに何行何列目かを指定する。両プロパティは **0 始まり** なので要注意。Grid.Row="1" Grid.Column="1"なら 2 行 2 列目だ。

要素が横に長く、**複数の列に渡って配置** したいーそんな時は、Grid.RowSpan や Grid.ColumnSpan を使おう。それぞれに指定した数だけ要素が占める場所が下方向・右方向に伸びる。これは、画面プレビューで操作している時に勝手に追加されていることもあるので、やはりコード側で直してあげよう。

8.4.3 UI 要素

個別の UI 要素については実際にコードを見ていただく方が早い。Ogose では ComboBox、ToggleButton、RadioButton、CheckBox などを使い分けている。それぞれの動作を規定するコードについては、*MainWindow.xaml.cs* の項で扱う。

少し説明が必要なのは、RadioButton についてだ。**ラジオボタン** というと、◎ 選択肢 1 ◎ 選択肢 2 のようなデザインが普通だ。

しかし、Ogose では縦に並べたり横に並べたりするので、横の二重丸がなく/普通のボタンと同じ見た目で/全体がクリック可能 である方が都合がよい。実は、これには複雑なコーディングは必要なく、トグルボタン用のスタイルを適用してやるだけで済む。

```
<!--  App.xaml -->
<Style TargetType="RadioButton" BasedOn="{StaticResource ToggleButton}">
```

これは、RadioButton クラスが ToggleButton クラスを継承しているため、共通のスタイル指定が使えることによる (参考にした記事: RadioButton な ToggleButton を実現する)。

8.5 MainWindow.xaml.cs

MainWindow.xaml のコードビハインドである。C#で書かれている。日電の Windows アプリケーションは代々 C#なので、宗教上やむを得ない事情がなければ C#を読み書きできるようになろう。

とはいえ、VS のコード補完 (IntelliSense) が凄く優秀なので、コードを書いていて苦労することはあまりなさそうだ。筆者も C#経験はないが、言語使用については for 文を少しググったくらいで不便を感じることは少なかった。

コード中にやたら<summary></summary>で囲まれたコメントを目にすると思うが、これは VS のドキュメント自動生成機能の推奨 XML タグらしい。ドキュメントを作るかは別として、面倒でなければこの形式のコメントにして損はなさそうだ。

400 行近いコードの全てを解説することはしないので、コードだけでは分かりにくいと思われる項目のみを以下に掲載する。

8.5.1 コマンド

コマンド とは、ユーザの操作を抽象化したものである。例えば、Word で編集していてペースト操作をしたいとき、どうするか考えてみよう。ショートカットキーを知っていれば Ctrl (Command) +V を叩くだろうし、右クリックしてペーストを選ぶ人もいるだろう。メニューバーからペーストメニューを選択してもペーストできる。

操作はいろいろだが、結果として呼ばれる処理は同一なのだ。この仕組みがコマンドで、WPF では ICommand というインターフェースで実現される。

無理にコマンドを使わずともアプリは作れるのだが、Ogose のキーボード操作を実装する際、必要に迫られて導入した。これまでと違い Ogose の回転/停止ボタンはトグル式で、色やラベルが状態により変化する。25 まで Click イベントを用いる方式では上手く行かなくなったのである (キー操作だと、外観を変えるべきボタンの名称を関数内で取得できないため... だった気がする)。

そこで、ICommand を使うようにプログラムを書き直した。時間がない中でやったので、かなり汚いコードになってしまった。今後書き換える際はぜひ何とかして欲しい。

コマンドの使い方

コマンドは高機能の代わりに難解なので、使い始めるときはこの記事あたりを参考にした。

âŚă まず、RoutedCommand クラスを宣言する。絶賛コピペなので意味はよく知らない。diurnalPlus は日周を進めるという意味だ。

```
/// <summary> RoutedCommand </summary>
public readonly static RoutedCommand diurnalPlusButtonCommand = new RoutedCommand(
↪"diurnalPlusButtonCommand", typeof(MainWindow));
```

âŚą この状態ではまだコマンドとボタン・処理が結びついていない。CommandBinding という操作でこれらを紐付けする。これもコピペ。

```
/// <summary>
/// MainWindow に必要なコマンドを追加する。コンストラクタで呼び出して下さい
/// </summary>
private void initCommandBindings()
{
    diurnalPlusButton.CommandBindings.Add(new CommandBinding(diurnalPlusButtonCommand,
 diurnalPlusButtonCommand_Executed, toggleButton_CanExecuted));
    /// (省略)
}
```

これをボタンの数だけ書き連ねる。new CommandBinding() に与えている引数は順に、コマンド・実行する関数・実行可能かを与える関数である。三番目のコマンド実行可否は、コマンドを実行されては困る時のための仕組みだ。

```
/// <summary> 各ボタンが操作できるかどうかを記憶 </summary>
private Dictionary<string, bool> isEnabled = new Dictionary<string, bool>()
{
    {"diurnalPlusButton", true},
    {"diurnalMinusButton", true},
```

```
    {"latitudePlusButton", true},
    {"latitudeMinusButton", true}
};
```

```
private void toggleButton_CanExecuted(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = isEnabled[((ToggleButton)sender).Name];
}
```

上手い方法が全然思いつかなかったので、isEnabled という連想配列を作っておいて、呼び出し元ボタンの名前をもとに参照するようにした。呼び出し元は、引数 sender に与えられて、ToggleButton など元々のクラスに型変換するとプロパティを見たりできる。

âŚć さて、private void initCommandBindings() をプログラム開始時に実行しなければバインディングが適用されない。MainWindow のコンストラクタ内で呼び出しておく。

```
public MainWindow()
{
    InitializeComponent();
    initCommandBindings();
}
```

考えてみれば大したことはしてないので、コンストラクタの中に直接書いてしまっても良かったかもしれない。

âŚč あとは XAML 側でコマンドを呼び出せるようにするだけである。<Window>タグ内にローカルの名前空間 (xmlns:local="clr-namespace:Ogose") がなければ追加しておこう。各コントロールの Command プロパティにコマンドをコピペする。

```
<!-- MainWindow.xaml -->
<ToggleButton x:Name="diurnalPlusButton" Style="{StaticResource DiurnalPlusButton}"␣
↪Grid.Row="2" Grid.Column="0"
              Command="{x:Static local:MainWindow.diurnalPlusButtonCommand}" />
```

これでクリック操作でコマンドが使えるようになる。

キー操作でコマンドを実行する

ここまできたら、キー操作でもコマンドが実行されるようにしたい。XAML で<KeyBinding>タグを使えば実現できるのだが、なんとこれではボタンが sender にならない。色々調べても対処法が見つからないので、結局キー操作イベントから無理やりコマンドを実行させるしかなかった。

```
private void Window_KeyDown(object sender, KeyEventArgs e)
{
    var target = new ToggleButton();
    switch (e.Key)
    {
        case Key.W:
            latitudePlusButtonCommand.Execute("KeyDown", latitudePlusButton);
            break;
        case Key.A:
            diurnalPlusButtonCommand.Execute("KeyDown", diurnalPlusButton);
            break;
        case Key.S:
            latitudeMinusButtonCommand.Execute("KeyDown", latitudeMinusButton);
            break;
        case Key.D:
            diurnalMinusButtonCommand.Execute("KeyDown", diurnalMinusButton);
            break;
    }
```

(コマンド名).Execute() メソッドの第一引数は ExecutedRoutedEventArgs e の Parameter、第二引数は object sender として渡される。結局、sender は第二引数に人力で指定した。

e.Parameter というのは、仕様では「コマンドに固有の情報を渡す」とされていて、要は自由に使っていいようだ。キーボード操作によるものかどうか、コマンドの処理で判定するために”KeyDown” という文字列 (勝手に決めた) を渡している。

コマンドごとの処理

最後に、CommandBinding でコマンドと紐付けた関数について書く。日周を進めるボタンのものは以下のようになっている。

```
private void diurnalPlusButtonCommand_Executed(object sender, ExecutedRoutedEventArgs␣
↪e)
{
    if (e.Parameter != null && e.Parameter.ToString() == "KeyDown")
    {
        ((ToggleButton)sender).IsChecked = !((ToggleButton)sender).IsChecked;
    }
    if (sender as ToggleButton != null && ((ToggleButton)sender).IsChecked == false)
    {
        emitCommand(nisshuidohenController.RotateDiurnalBySpeed(0));
    }
    else
    {
```

```
        emitCommand(nisshuidohenController.RotateDiurnalBySpeed(diurnal_speed));
    }
    if (sender as ToggleButton != null) toggleOppositeButton((ToggleButton)sender);
}
```

どうしてこのような汚いコードになったのか弁解しておこう。

8.5.2 イベントハンドラ

コマンド関連以外の残りのコードの大部分は、GUI おなじみのイベントハンドラである。どんな時にどの関数が呼ばれるかは `MainWindow.xaml` を見れば書いてあるので、ここでは触れない。

8.5.3 NisshuidohenController.cs

TODO

第 9 章

部品の買い方

- 書いた人: Kenichi Ito(nichiden_27)
- 更新日時: 2017/02/17
- 実行に必要な知識・技能: 秋葉原の土地勘
- 難易度: 2/少しやれば可能
- 情報の必須度: 3/必要な場合がある

9.1 概要

回路を設計したら、部品を買いましょう。日電は重要なので予算はある程度優遇されますが、無限ではないのでできれば安く手に入れたいところです。通販は便利ですが、回数が重なると送料が無視できません。せっかく東京にいることだし、(2 年秋に本郷に通う人は特に) 秋葉原に通うことをオススメします。

9.2 リアル店舗で買う

9.2.1 全体の注意

- ほぼ大体 10 割くらい秋葉です
- 自分の経験と勘に絶対の自信がある人以外は、買うものの**型番**をメモすべし
  - 名称が似ていても仕様が違ったりするので、型番で！！！
- 予備を絶対に買うこと

  - 壊れやすい部品 (IC とか) は特に注意

9.2.2 秋葉原電気街へのアクセス

- 駒場から
- 山手線に乗り、秋葉原駅で降りる
- 銀座線に乗り、末広町駅で降りる
- 本郷から
  - 徒歩で
    - キャンパス内の位置にもよるが、30 分足らずで移動できる
    - 湯島や外神田を散策できるので、余裕のある日は是非
  - 電車で
    - 根津駅から千代田線で新御茶ノ水駅まで移動
    - 御茶ノ水からは徒歩
    - 電気街は秋葉原駅より西側なので、思いのほか早く着く
  - バスで
    - 正門前や赤門前から**茶** 51 系統で万世橋まで
    - 本郷二食前から**学** 07 系統で御茶ノ水駅前まで
    - 徒歩が少なくて楽

9.2.3 秋月電子通商

迷ったらここ。大抵のものは揃う上価格も安い。

ただ店内が狭く、平日昼間とか以外はすごく混雑するので注意。あと初見だと棚の配置が分からない。店内に地図があるはずなのでまずはゲットしよう。

ネット通販もあるので、店員に質問するときは通販のページを見せながらだとスムーズかと。とにかく部品の種類が凄いので見つからないと思ったら臆せず聞くべし。

ここの 1000 円のお楽しみ袋は伝説。初心者は勉強のために一回は買ったほうがいいだろう。

9.2.4 千石電商

第二選択肢。秋月より広く三号店まであるものの、値段が全体的に高い。

秋月では揃わないコネクタや工具、切り売りの線材などはここで入手しよう。会計は各フロアで行わないといけないので注意。

9.2.5 西川電子

総武線の高架近くにある店。「ネジ」と「コネクタ」に強いとされる。特にネジは一通り揃っているのは秋葉でもここくらいで、ハ◯ズなんかよりずっと安価で手に入るので近くを訪れた際はぜひ寄ろう。また、二階のヒロセや日圧のコネクタもたいへん充実している。

鈴商亡き今、秋葉の部品屋の「人情」みたいなものを感じられる店の一つかも。秋月や千石からすると閑散としていて心配だからみんな買いに行こうな。

9.2.6 akibaLED ピカリ館

秋月、千石と同じ並びにある LED 専門店 (!)。夕方に通ると大変まぶしい。

秋月がパワー LED の店頭在庫を切らしていた時に助けてもらった。値段は秋月と同じくらいだったと思う。とにかく LED 関連の品揃えが凄いので時間が余ったら寄ると面白いかも。

なお、これを運営している（株）ピースコーポレーションは LED パラダイスという LED 通販サイトも手がけている。

9.2.7 マルツ秋葉原本店/2 号店

言わずと知れた通販大手だが店舗も全国に点在している。秋葉には本店と 2 号店があり、後者は秋月のはす向かいのブロックにあるので行きやすい。

近くに秋月千石の二強がある以上、ここで買い物する機会は多くないが、実は夜に真価を発揮する。秋葉の部品屋は 19 時くらいまでに軒並み閉まってしまうのに、ここは 20:00 まで開いている。今すぐに欲しい部品がある場合は駆けこもう。

また 2 号店の入り口近くには怪しい電源装置が特価で置いてあったりする。

9.2.8 鈴商

今はないお店。千石〜秋月〜マルツの並びにあった老舗だが、2015 年 11 月に閉店。跡地には秋葉原神社という謎の施設が入った。

9.2.9 ラジオデパート

総武線の高架沿い、西川電子と同じ並びにある。

行ったことがないので不明だが、多数の部品屋が入居しており、珍しい部品があるかもしれない。

9.3 ネットで買う

9.3.1 全体の注意

- まともなサイトは仕様をまとめたデータシートを PDF などで配布している
  - 必ず確認してから買おう!
  - データシートなしの部品には手を出さないのが無難
- 送料・納期を確認すべし
- 領収書は、納品書 + 支払いの証拠 (振込の明細書など) とするのが基本
  - 電子部品は店の選択肢が少ないのでこうするしかないようです

9.3.2 秋月電子通商

送料一律 500 円。お店に行く時にもここで予習しておくと便利。そのうち本店での商品位置情報が分かるアプリがリリースされるらしい。

9.3.3 千石電商

これも予習に使える。送料 432 円。

9.3.4 マルツ

大手だけどここで買ったことがないのであまり分からず。法人/官公庁向けの販売もしているのでサイト構成はちょっと敷居が高い感じ。

LED の使い方をはじめ、初心者~脱初心者向けの情報を提供していたりする。回路を組んでいて迷ったら探してみよう。

ケース加工やプリント基板加工もしており、しっかりとしたものを作りたい場合はお世話になるかも? 大学生協との提携も盛んで、学科や研究室で電子回路を扱う時に関わることになるのかもしれない。

9.3.5 スイッチサイエンス

10000 円以上で送料無料。海外向け製品など秋月で扱ってない商品もあるようだ。

電子工作や IoT のイベントを運営していたり、ブログに解説記事を載せたりと活動は活発。

9.3.6 オリエンタルモータ

モータ専門の企業。日周・緯度変のモータとモータドライバは代々ここで買っている。

買い替える場合は型番を見て同じものを買えばいいだろう。ドライバはかなり高額だが、部室に予備があるしそもそも既製品なのであんまり壊れない。

9.3.7 maxon

同じくモータ製造大手。データシートが詳細で、モータの勉強ができる。マクソンアカデミーという解説記事群もある。

9.3.8 ミスミ

金属系の部品なら買えないものはない、神のような存在。しかも送料無料 (ネジ一本でも)。ただし法人格のアカウントがないと買えない。

日電で本格的に機械部品を扱うことは稀なので、無理に使うことはないかもしれない。

第 10 章

索引

- genindex
- search