実習 2 日目：トピックス選択

T. Obina (KEK, Accelerator Division 7)

2018/11/02(金) 12:00 - 16:30

概要

EPICS 入門セミナー第 2 日目; 午後の実習。最初に全員でできる実習 (GPIO の割り込みで EPICS レコードを動かす) を行う。その後、各自でトピックスを選んで実習を行う。Arduino は 11 個しかないので各テーブル 1 個ずつとして、終わったら隣の人と交代。その後時間があれば Software　Sequencer あるいは CSS の残りトピックスをおこなう。

目次

1　共通：GPIO からの割り込みで制御

1.1　はじめに

このトピックスでは GPIO ピンからの電圧入力で EPICS レコードをプロセスすることを目指す。ボタン入力を検出するなど、外部ハードウェアとのインターフェースの例としてもっとも単純なものであるので、まずは昨日の復習を兼ねて GPIO についてもう少し説明する。ここでは図 1 のように「プッシュスイッチを押すと GPIO ピンに電圧がかかり、それを検出する」という動作を目指したい。しかし、実際の回路ではこのように配線すると問題が起きる

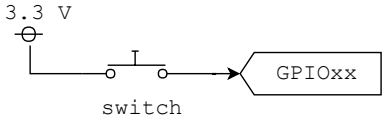

図 1　なんとなく、目指している動作

場合があります。スイッチが閉じているときは良いのですが、オープンのときに端子の電圧は不安定となり、状況によって 0 V だったり、中途半端な電圧になったりします。そこで、オープン状態のときに High/Low のどちらかにきちんと定まるように RasPi 内部では「プルアップ」または「プルダウン」回路 (図 2 参照) のどちらかに接続されています。使いたいピンがプルアップなのかプルダウンなのかは特にデジタル入力として使うときに注意が必要です。外部にスイッチをつなぐ場合、左のように入力ピンがプルダウンされている場合は正論理 (スイッチが OFF の時に LOW, ON の時に HIGH)、右図のようにプルアップされている場合は負論理 (スイッチが OFF の時に HIGH, ON のときに LOW) として構成します。実際に Raspberry Pi ではどのようになっているのか確認しましょう。例えばコ

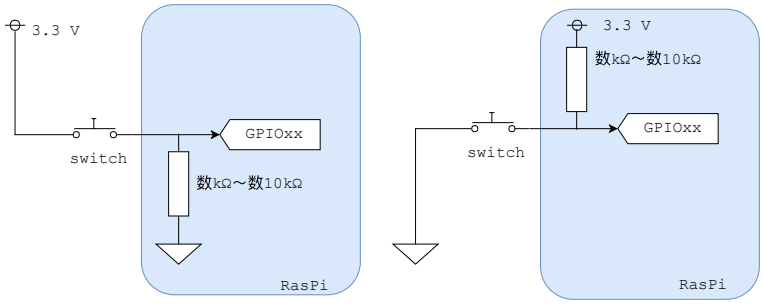

図 2　プルアップ回路とプルダウン回路

マンドラインで `gpio readall` コマンドを実行すると全てのピンステータスを読むことができます。図 3 に電源投入直後に実行した出力を示します。V の列が現在の電圧 (1/0 が High/Low) を示しており、BCM の列が Broadcom 定義のピン番号、wPi/Name の列は wiring Pi 定義のピン番号や名前です。wiring Pi は Raspberry Pi 用のインターフェースライブラリのひとつですが、今回の講義では使用せず、全て BCM 番号で説明します (例えばこれまで GPIO17 等と指定してきたのは BCM 番号です)。wiring Pi に興味のある方は web[4] などで調べてください。gpio コマンドのオプション類は `man gpio` で確認できます。電源投入時の電圧 (表で"V"の列) をみると、デフォルトでは

- `GPIO01 - 08` まではプルアップ
- `GPIO09 - 26` まではプルダウン

となっていることがわかります。ハードウェアの詳細は [1] などを参照してください。

1.2　ハードウェア

今回は、図 1 のように「プッシュスイッチを押すと、ピンに電圧がかかる」という構成 (正論理) にしたいので、プルダウン状態になっているピンを使えばスイッチ解放時に 0 V に確定することが分かると思います。ここでは例として GPIO18 を使います。使用するスイッチの画像例は下図の通り。どのピン同士が接続しているのか確認して、図 1

```
$ gpio readall
 +-----+-----+---------+------+---+---Pi 3+---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 | ALT0 | 1 |  3 || 4  |   |      | 5v      |     |     |
 |   3 |   9 |   SCL.1 | ALT0 | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 0 | IN   | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI | ALT0 | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO | ALT0 | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK | ALT0 | 0 | 23 || 24 | 1 | OUT  | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | OUT  | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3+---+---+------+---------+-----+-----+
```

図 3　gpio readall コマンド出力例。各ピンの High/Low 状態の他、モード IN/OUT の状態も一覧表示できる。

の回路になるようにブレッドボード上に組んでください。(昨日使用したブレッドボードに十分スペースはあると思います: この章の最後に参考写真を掲載しています)

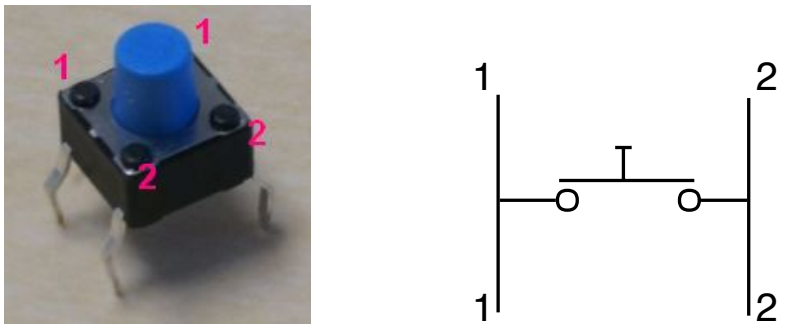

図 4　Push switch の写真 (左) と回路図 (右)。4 本足が出ており、同じ側面から出ている足 2 本のピン間は押している間だけ導通する。

1.3　Db 作成

EPICS データベース作成。先日作成した `<TOP>/gpio1App/Db` をもとにして test.db に追加するとする。

```
record(bo, "$(head):GPIO17:OUT") {
  field(DTYP, "devgpio")
  field(OUT,  "@17 H")
  field(ZNAM, "OFF")
  field(ONAM, "ON")
}
record(bi, "$(head):GPIO18:IN") {
  field(DTYP, "devgpio")
  field(INP,  "@18 H")
  field(SCAN, "I/O Intr")
  field(ZNAM, "OFF")
  field(ONAM, "ON")
}
```

ここで肝心なのは `SCAN, "I/O Intr"`のフィールドです。入力ステータスを定期的 (例えば 0.1 秒毎) に読み込むことも可能ですが、それでは「押した瞬間にできるだけ速く応答する」ことが困難になります。この指定により外部

割り込みに応答して EPICS レコードをプロセスすることが可能になっています。<TOP>/gpio1App/src ディレクトリでは特に操作は必要ありませんので、あとは make をかけて実行し、st.cmd を実行してください。別端末を開き、camonitor でレコードをモニターしながら、push switch を押してみてください。意図した動作が実現できているでしょうか？

1.4 GUI 作成

- 今回追加したレコードを GUI で表示し、ボタンを押すのに同期して表示が変わることを確認する。
- 自分の Raspberry Pi だけでなく、ほかの人のレコードも表示してみる。

1.5 時間のある人へ

- 負論理のスイッチを作成する。プルアップしているピンを使うこと。
  - プルアップしているピンを選ぶ。例えば GPIO02
  - 図 2 を参照し、スイッチは GND に向けて接続する
- 単にロジックを反転したいだけならば PU/PD どちらでも構わず、ソフトで反転するので十分。実際には (どちらかといえば) プルアップ入力の方が使われることが多い。理由を考える。
- 現在は信号の立ち上がり/立下りエッジの両方を検出してレコードが動いています。場合によっては立ち上がりエッジだけを検出したいこともある（例えばリモコンのボタンとか）ので、ソフトウェア（レコードのリンク) で対処する方法を考えてみる。
- デバイスサポートに手を入れてどちらかのエッジだけで割り込みをかける方法を考えてみる
- コマンドラインから LED を制御する”だけ”ならば、gpio コマンドで十分。わざわざ EPICS IOC にする意義（御利益）は何なのかを考える。

参考写真

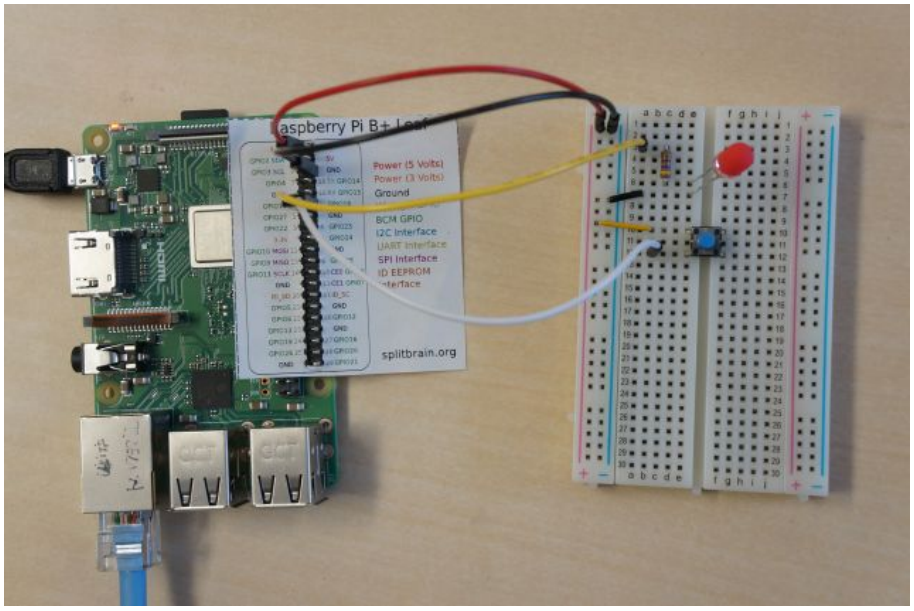

図 5　GPIO のサンプル。ブレッドボード上に LED とスイッチによる割り込みを実装した例。

※ さらに進んだトピックス

- 割り込みを I/O Intr にしてどこまで高速に応答できるか、応答時間のばらつきはどうなるかを検証 (信号発生機, オシロスコープ必要)。リアルタイム性, タスク優先度、コンテクストスイッチなど。

2　11 名限定：Arduino でアナログ入力

2.1　はじめに

Raspberry Pi にはアナログ入力がありません。しかし、加速器制御にかぎらず一般的にアナログ入力はとても肝心かつ面白いトピックスで、測定したい対象の特性に気を付けてきちんとした測定器を準備することは非常に重要です。例えば、Keysight や Tektronix など、メーカー製のデジタルマルチメーター (DMM) を使うならば午前中のセッションでも登場した Stream Device を使って Ethernet 経由でアクセスするのが適切です。

一方で、場合によっては精度は不要で傾向だけ安価かつ手っ取り早く測定したいこともあります。そんなときに、なんらかの方法で Raspberry Pi でアナログ値を取り込むことができると便利です。例えば

- `Analog --> ADC on Arduino <==[UART(RS232/USB)]==> RPi`
- `Analog --> ADC + I2C ==[I2C]==> RPi`
- `Analog --> ADC + SPI ==[SPI]==> RPi`
- `Analog --> ADC + I2C ==[I2C]== [マイコン] <==[UART(RS232/USB)]==> RPi`
- ...

などが良くつかわれる構成でしょう。I2C や SPI は高速のシリアル通信規格で、主に基板内やごく近くにあるデバイス間の通信で使用されます。温度や湿度が測定したいならば、I2C インターフェースを持った温度センサーチップなども発売されていて容易に入手可能です。ここでは Arduino を使ってアナログ値を取り込む方法を説明します。I2C 経由で取り込む例は別の実習トピックスとして紹介します。

2.2　Arduino

Arduino(アルデュイーノ or アルドゥイーノ と発音)[2] とは、いわゆるワンボードマイコンの一種であり、主な特徴には以下のようなものが挙げられる。

- AVR マイコン搭載（Atmel 社が製造しているマイクロコントローラー）
- Analog 入出力、Digital 入出力を装備
- 統合開発環境 (Win/Mac/Linux 等) があり、プログラムをコンパイルしてしてダウンロード・実行する
- Arduino 言語は C 言語ライクな構文。初心者でも容易に製作できるように設計。
- オープンソースハードウェア

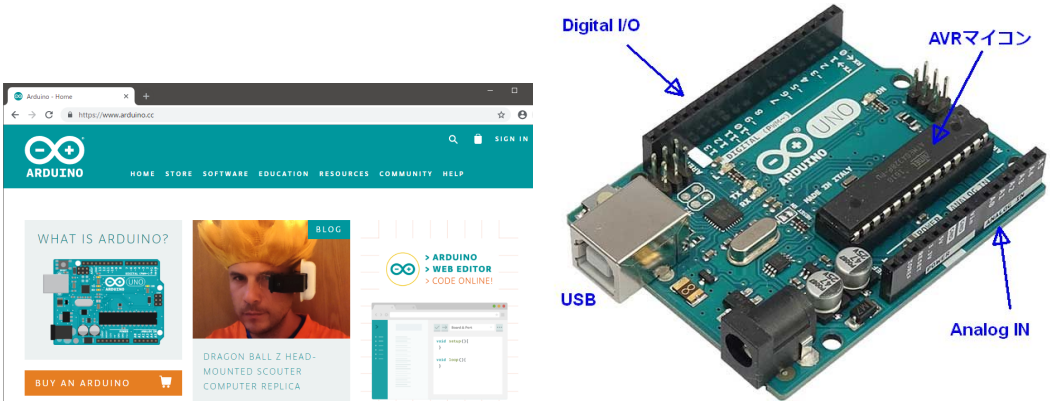

図 6　左：arduino web サイト [2], 右：ハードウェア写真

Arduino には回路の規模や価格、インターフェースの種類などによって多くの派生ハードウェアがある。今回の講習で使用する Arduino Uno の代表的なスペックは以下の通り

item	spec
プロセッサ	ATmega328P
Digital IO	14 本 (このうち 6 本は PWM 使用可)
Analog IN	6 本, 10 bit(1024), 0 - 5 V, max 10kSPS

2.3 アナログ照度センサー

CdS(硫化カドミウム) センサーを使用して明るさを検知します。CdS は非常に安価な半導体光センサーで、フォトレジスターとも呼ばれる素子です。暗いときには数 MΩ 程度、明るいときには数 100 MΩ 程度なので間違ってショートする危険が無く誘導性や容量性を持たないので使いやすいのですが、カドミウムが RoHS 規制対象物質なので EU では使用できないという欠点はあります。明るさを測定するには適当な電圧をかけて流れる電流から抵抗値の変化を測定すれば良く、例えば図右のような回路を作って Arduino で電圧を測定するのが簡単でしょう。

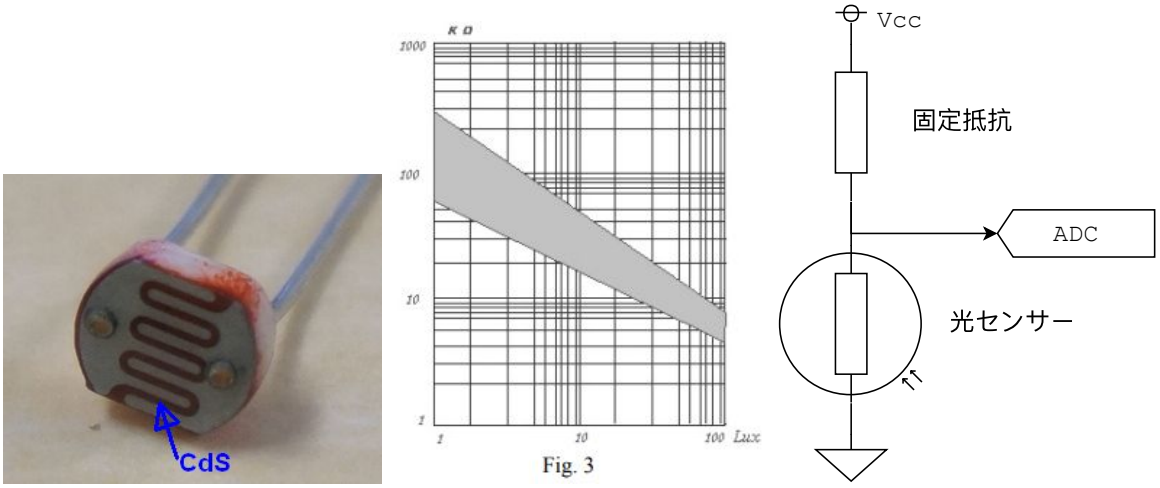

図 7　CdS センサーの写真と特性, 回路例。

2.4 Arduino の中で動いているものは?

実習用の Arduino には、定期的にアナログ入力値やデジタル入出力を読んでシリアル経由で値を送るプログラムが既に書き込まれています。EPICS IOC からみると、シリアル経由でアナログ値取得コマンドを送って返信を読み込むだけで良いことを意味します。これはいままでやってきた Stream Device と同じです。大きな違いは、Ethernet 経由ではなく Raspberry Pi のシリアルポートを使うことだけです。主に使用するコマンドや通信条件は以下の通り：

speed	115200 bps
parity	none
func	command
delimiter	*
version	?
ain	S
aout	s
din	R

アナログ入力を読むには S コマンドに続いてチャンネル番号を送る。(例：S0 など)

2.4.1 USB 接続、ID 確認

まずは Arduino が接続しているポート番号や、デバイス ID を知る必要があります。Arduino を Raspberry Pi の USB ポートに差し込み、 `dmesg` コマンドを入力すると Linux システムでどのように検知されているか分かります：

```
$ dmesg
....

[ 687.051600] usb 1-1.1.3: new full-speed USB device number 5 using dwc_otg
[ 687.195516] usb 1-1.1.3: New USB device found, idVendor=2341, idProduct=0043
[ 687.195531] usb 1-1.1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=220
[ 687.195541] usb 1-1.1.3: Manufacturer: Arduino (www.arduino.cc)
[ 687.195549] usb 1-1.1.3: SerialNumber: 6493633303735131F030
[ 687.240259] cdc_acm 1-1.1.3:1.0: ttyACM0: USB ACM device
[ 687.241217] usbcore: registered new interface driver cdc_acm
[ 687.241225] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
```

のような出力が出ているはず。”SerialNumber”以下がこれがデバイス固有の ID であり、このデバイスが ttyACM0 として認識されていることが分かります (CDC: Communication Device Class, ACM: Abstract Control Model)。つまり RasiPi のユーザープログラムからこれにアクセスするには `/dev/ttyACM0` を使えば良いことになります。[注：] 複数の USB デバイスを接続したときにこの番号は変わってしまう可能性がありますのでそんなトラブルを予防するにはシリアル ID をもとにアクセスする方が望ましいです。USB デバイスを刺したあとで

```
$ ls /dev/serial/by-id/
usb-Arduino__www.arduino.cc__0043_6493633303735131F030-if00
```

というデバイスファイルが出来上がっているので、これに対して read/write するのが一番確実です (ID は個体によって違うので、各自調べて記録すること)。文字列は長いですがコピー＆ペーストすれば手間はかかりません。

2.4.2 コマンドラインで確認

まずは EPICS を使わずに、正常に通信できているかどうか確認する。(うまく通信できないときにデバッグする手段としても有効) Linux 上で使える端末ソフトウェアはいくつかありますが、ここでは `GNU screen`[5] を使います。

```
screen /dev/ttyACM0 115200
```

あるいは

```
screen /dev/serial/by-id/usb-Arduino..(省略)..F030-if00 115200
```

通信速度をオプションとして指定するだけで、あとはデフォルトで大丈夫です。

```
Ready:0.0.1.03
```

という表示が出るはずです。次にキーボードから 1 文字`"?"`を入力してください。以下の応答文字列が表示されるでしょうか？

```
0.0.1.03*
```

最後についている*が Arduino から送られて来るコマンドの終了 (デリミタ) です。次に `S0` あるいは `S1` などと入力すると、なにがしかの数値が返ってきます。これはアナログチャンネル 0 や 1 の ADC 値を読むコマンドです。ここまで確認できたら、screen を終了して EPICSIOC に進みます。

GNU screen を終了するには CTRL-a を押してからキーを離し、 k キーを押します。これで kill コマンドを送ったことになり、画面左下に図のように Really kill this window [y/n] と表示されますので、y を押してください。これで Screen 終了です。画面上部に [screen is terminating] と表示されていることと思います。あるいは

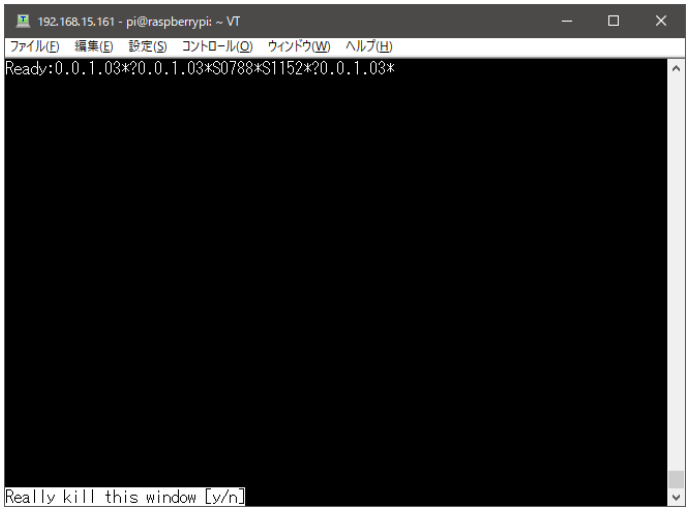

図 8　GNU Screen でのデバッグ例と、CTRL-a k (kill コマンド) を送った時の表示

（少し乱暴ですけど）Arduino がつながっている USB ケーブルを引き抜けば接続中の Screen は終了します。これで Arduino が壊れることはありませんので恐れずに抜いても大丈夫です。

慣れないと GNU Screen の操作は少しわかりにくいかもしれませんが、Raspberry Pi にシリアルデバイスを接続するときにはとても便利ですし、Unix を使う際の「仮想端末」として使うことが出来ます。覚えておくときっと役立つことと思います [私見]。(※余談ですが GNU Screen のコマンドキーはデフォルトで CTRL-a になっています。通常のシェルではこれは「行頭に戻る」ですので、気に入らない場合は割り当てを変えましょう) また、念のため screen が生き残っていないか確認するには以下のコマンドが適切です。

```
$ screen -ls
No Sockets found in /run/screen/S-pi.
$
```

2.4.3　EPICS IOC 作成

今回は既に出来上がっているテンプレートを使います (ここでは例としてアプリケーション名を arduino としていますので、適当に変更して作成してください)。テンプレートの名前は Noboru Yamamoto's Arduino Kit からとっています。

```
$ makeBaseApp.pl -t nykit arduino
$ makeBaseApp.pl -i -t nykit arduino
```

configure/RELEASE ファイルを編集し、 ASYN, STREAM を有効化

```
ASYN=/opt/epics/R315.6/modules/soft/asyn/4-31
STREAM=/opt/epics/R315.6/modules/soft/stream/2-7-7
```

<AppTop>/src/Makefile の中で、asyn, stream, drvAsynSerialPort, および該当するライブラリが入っていることを確認

```
arduino_DBD += asyn.dbd
arduino_DBD += stream.dbd
arduino_DBD += drvAsynSerialPort.dbd

arduino_LIBS += asyn
arduino_LIBS += stream
```

その後、make する。

iocBoot/ioc<APPNAME>/st.cmd を編集し、header マクロを置き換えること、およびシリアルポート名として ls /dev/serial/by-id/ した出力をコピペする

```
#epicsEnvSet("header", "ET_your_name_here")
epicsEnvSet("STREAM_PROTOCOL_PATH", ".:../../arduinoApp/Db")
dbLoadRecords("db/devArduinoProto.db","head=$(header):Arduino,R=0,TTY=PS1,ascan=.1 second")
drvAsynSerialPortConfigure("PS1", "/dev/serial/by-id/usb-Arduino..(省略)..F030-if00")
asynSetOption("PS1", 0, "baud", "115200")
```

その後、ioc 実行し、エラーが出ないことを確認。 dbl を実行してどのようなレコードがあるか確認する。

```
$ chmod +x st.cmd
$ ./st.cmd
```

照度モニターの値を読み込む。手をかざしてみて、明るさに応じて出力が変化することを確認する。

```
camonitorpi@raspberrypi:~ $ camonitor ET_kektaro:Arduino:LUM
ET_kektaro:Arduino:LUM         2018-10-26 09:07:46.175368 3.80254
ET_kektaro:Arduino:LUM         2018-10-26 09:07:46.375620 3.79765
ET_kektaro:Arduino:LUM         2018-10-26 09:07:46.474368 3.78788
....
```

以下の項目を確認、作成する

- GUI 作成. 照度データをプロットしながら、センサーに手をかざす。
- 対応するレコードの db ファイルを確認
- protocol ファイルを読んで、どのような通信を行っているのか確認する。
- デバッグ用に TraceMask を設定してどのような通信を行っているのか確認する (この際には startup script でスキャン時間を 1 sec 程度に指定する方が良い)
- その他のレコードにアクセスしてみる。温度センサーなど。

2.5 将来：さらに進んだトピックス

- Arduino のスケッチを自作する。大まかな流れは以下の通り。
  - 開発環境 (Arduino IDE) をダウンロード [3]、PC にインストール
  - PC に Arduino を接続 (USB 経由)
  - スケッチを書く
  - Arduino に書き込む
  - USB 経由で色々とメッセージを表示してデバッグ

3　(時間があれば)SNL/Sequencer で LED 点滅

3.1　はじめに

ここでは EPICS 標準配布物に含まれる State Notation Language (SNL) および Sequencer [6] について簡単に説明し、一定周期で LED を点滅させるために使用する。

SNL は状態記述言語と呼ばれる言語の 1 つで、プログラマは「状態 (state)」および「状態間の遷移条件」記述することで制御プログラムを見通し良く、また信頼性・保守性の高いプログラムを書くことができる。SNL で記述されたソースは State Notetion Compiler によって C 言語ソースに変換され、EPICS IOC 内のプロセスあるいは単独のプログラムとして実行できる。sequencer とは SNL コードを IOC 内で実行するツールであり、ioc shell の中で seq コマンドとして実行可能。

3.2　簡単な例

makeBaseApp.pl のテンプレートとして example を使用するとサンプルコードが作成される。これは昨日の講義で使用したテンプレートなので、exampleApp 以下にソースが展開されているはず。後で見てほしい。この状態遷移図と、該当するソースコードは図 10 の通り。”Light ON”と”Light OFF”の 2 つの状態を定義し、v の値が 5 より大きいか小さいかという”条件”によってそれぞれの間を遷移するという条件が記載されている。プログラムは単純なのですべて記載する。

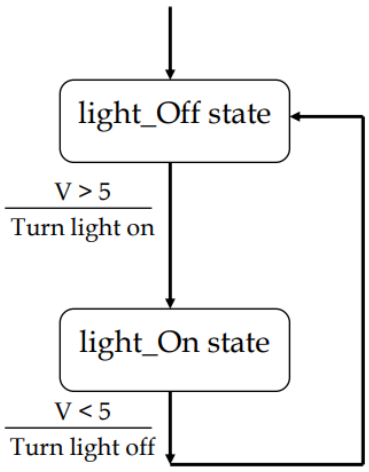

図 9　状態遷移図。

```
program sncExample
double v;
assign v to "{user}:aiExample";
monitor v;

ss ss1 {
  state low {
    when (v > 5.0) {
      printf( “Changing to high\n");
    } state high
  }
  state high {
    when (v <= 5.0) {
      printf("Changing to low\n");
    } state low
  }
}
```

非常に単純なので、何をやろうとしているのかは分かりやすいと思う。この程度のプログラムであればふつうに C

言語の条件分岐だけで記述しても問題無いと思うが、もっと複雑な状態遷移が必要な場合も多くあるので、そんなときに SNL は便利な記述方式です。例えば、インターロック機器の状態や、機器立ち上げ・立ち下げシーケンスなど。

その他にも、例えば 1 秒おきに ON/OFF を繰り返すケースでは以下のような SNL プログラムが適切だろう。これを EPICS Application として実装する。(実際にはもう少し変数 ON/OFF の周期や幅も変数として入れてしまう)

```
assign out to "{head}:GPIO17:OUT";

ss ss1 {
    state LEDoff {
        when (delay(0.5)) {
            out = 1;
            pvPut(out);
        } state LEDon
    }
    state LEDon {
        when (delay(0.5)) {
            out = 0;
            pvPut(out);
        } state LEDoff
    }
}
```

3.3 EPICS application 作成

ここでは既に作成済みの rpiGpio テンプレートを使用する。この中には GPIO17, 18 を使った入出力設定と、サンプルの SNL プログラムが用意されている。

```
$ mkdir -p ~/epics/app/pwm
$ cd ~/epics/app/pwm
$ makeBaseApp.pl -l
Valid application types are:
        rpiGpio
        support
        caServer
        caClient
        stream
        nykit
        example
        ioc
Valid iocBoot types are:
        ioc
        example
        nykit
        rpiGpio
        stream

$ makeBaseApp.pl -t rpiGpio test
$ makeBaseApp.pl -i -t rpiGpio test
```

<appTop>/configure/RELEASE を編集し、SNCSEQ, RPIGPIO を有効化

```
SNCSEQ=/opt/epics/R315.6/modules/soft/seq/2.2.4
RPIGPIO=/opt/epics/R315.6/modules/soft/gpio/20160308
```

念のため<appTop>/<appName>/src ディレクトリに移動し、Sequencer プログラム sncProgram.stt の中身を確認する。(短いので全部掲載する)

```
program sncPwm

short  out;
assign out to "{head}:GPIO17:OUT";
```

```
double  freq;
assign  freq to "{head}:GPIO17:PWM_FREQ";
monitor freq;

double  duty;
assign  duty to "{head}:GPIO17:PWM_DUTY";
monitor duty;

ss ss1 {
    state LEDoff {
        when (delay(1/freq*(100-duty)/100.0)) {
            out = 1;
            pvPut(out);
        } state LEDon
    }
    state LEDon {
        when (delay(1/freq*(duty)/100.0)) {
            out = 0;
            pvPut(out);
        } state LEDoff
    }
}
```

その後、make し、 st.cmd を実行する。

```
$ ./st.cmd
.....
iocRun: All initialization complete


epics>
epics> dbl
ET_kektaro:GPIO18:IN
ET_kektaro:GPIO17:PWM_FREQ
ET_kektaro:GPIO17:PWM_DUTY
ET_kektaro:GPIO17:OUT
```

GPIO17 に 1/0 を書いて LED が点灯/消灯することを確認する。

これで問題なく動作することを確認してから、sequencer を実行する。st.cmd の最後の行

```
## Start any sequence programs
seq sncPwm,"head=ET_kektaro"    <--- コメント外す
```

のコメントを外してから、再び st.cmd を実行する。

```
$ ./st.cmd
......
iocRun: All initialization complete


## Start any sequence programs
seq sncPwm,"head=ET_kektaro"
sevr=info Sequencer release 2.2.4, compiled Wed Oct 17 23:30:53 2018
sevr=info Spawning sequencer program "sncPwm", thread 0x167a7c0: "sncPwm"
sevr=info sncPwm[0]: all channels connected & received 1st monitor
```

実習項目

- PWM_FREQ, PWM_DUTY の現在値を確認。
- PWM_FREQ の値を 1 Hz から 0.5 Hz に変更してみる。

- PWM_DUTY の値を変更 (0 - 100) して LED の点滅間隔が変わることを確認

※参考：PWM による調光 LED の明るさを変えるには流れる電流値を変えるのが簡単ですが、デジタル出力には ON/OFF の 2 種類しかないので電流量を変えることはできません。しかし、ON/OFF を高速で切り替えて、その ON 時間と OFF 時間を調整することによって平均的な明るさを変えることが出来ます。これが PWM(Pulse Width Modulation) による明るさ調整です。以下はその他の実習項目：

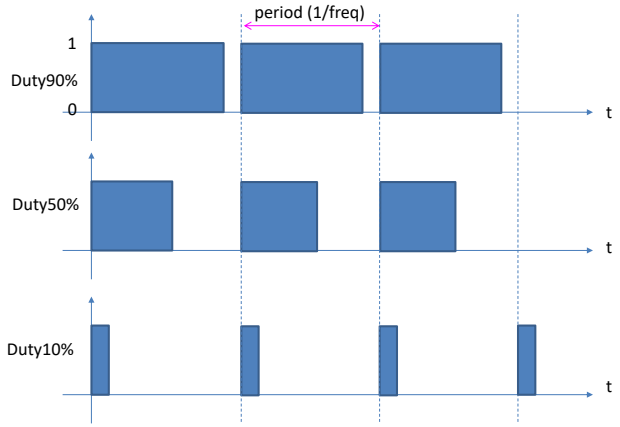

図 10　PWM による調光原理

- database を編集し、PWM_DUTY の値範囲を 0 ～ 100 % に限定する（DRVH,DRVL)
- database を編集し、PWM_FREQ の値範囲を 0 以上に限定する（DRVL)
- PWM_FREQ, PWM_DUTY の値を調節する GUI を作成 (CSS).
- PWM_FREQ の値を 1 Hz から 10 Hz に変更。同様に DUTY を変えてみる
- PWM_DUTY の値を変更 (0 - 100) して LED の点滅間隔や明るさが変わることを確認
- PWM_FREQ の値をさらに増やし、どのような応答になるかをみる

このサンプル（テンプレート）は実際に明るさ調整用の PWM として使うにはいくつか問題がある。例えば DUTY 0 % に設定しても、完全に消灯しない。この理由そして対策を考えること。一方で RPi にはハードウェア PWM 機能をピンに持たせることも可能である (次節参照)。使い方については Web に多くのドキュメントがあるのでそれを参照するのが良いだろう。コマンドラインからの設定、さらに将来的には EPICS からの制御に挑戦してほしい。

3.4　参考: PWM 専用ピンを使う (not EPICS)

RasPi には PWM チャンネルは 2 つあり、4 本のピンに接続されている。すなわち、1 つのチャンネルは GPIO12 と 18 に接続しており、もう 1 つは GPIO13 と 19 に接続している。そのため、PWM 設定をおこなうと 12 と 18 は同じ出力が出る (別々には設定できないとも言えるし、同じ出力が出せるともいえる)。LED をつなぐピンを変更 (例えば GPIO13 に変更) し、以下のコマンドを入力すればどのような動作をするか分かるだろう。

```
$ gpio -g blink 13         <-- 点滅
$ gpio -g mode  13 pwm     <-- PWM モードに設定 (GPIO12 or 18, 13 or 19),
$ gpio -g pwm 13 1024      <-- PWM 最大
$ gpio -g pwm 13  100
$ gpio -g pwm 13   50
$ gpio -g pwm 13   10
$ gpio -g pwm 13    0      <-- PWM 最小  これならば完全に消灯する
```

4　2 名限定：I2C 経由でデータ取得

RPi に I2C(Inter-Integrated Circuit) デバイスを接続して制御する。I2C はアイ・スクエアド・シー、アイ・アイ・シーと呼ぶのが正しいという説もあるが、アイ・ツー・シーと呼ぶことも多い。主に比較的低速の IC と、プロセッサーやマイコンとの間を接続するときに使用されるバス規格で、ボード内や隣接するボードなど、比較的短距離の通信に使用される。シリアルデータ (SDA) とシリアルクロック (SCL) の 2 本の線で通信する。標準的にはアドレス空間は 7bit(最大ノード数 112)、通信速度は 100 kbps 程度。今回使用している Raspberry Pi のイメージ（Raspbian）では標準でサポートされている。実習では

- ADT7410 温度センサー
- その他、ADC, DAC など

を使用する。詳細は `http://cerldev.kek.jp/trac/EpicsUsersJP/wiki/epics/raspberrypi/setup_epics_i2c` を参照

4.1　ADT7410 温度センサー

まずはデータシート `https://www.analog.com/jp/products/adt7410.html#product-overview` を確認する。今回は秋月のキットを使っているのでそちらの説明 `http://akizukidenshi.com/catalog/g/gM-06675/` も確認する。(図は秋月の資料より引用）I2C に使用する 2 つのピン (SDA,SCL) の 2 本は RaspberryPi 上でプルアップさ

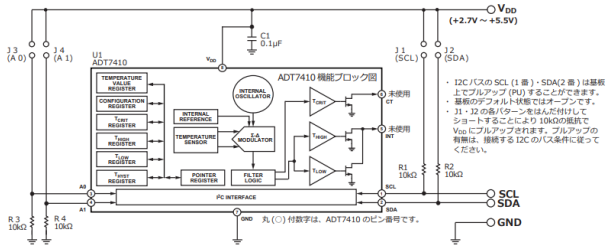

図 11　ADT7410 温度センサー; 秋月電子のキット資料より

れているので、特に外付け回路無しでそのままつないで大丈夫。ブレッドボード上に GND と Vss と合わせて合計 4 本の線を接続したら以下のコマンドを実行する。

```
$ i2cdetect  -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

デフォルト設定では 0x48 のアドレスに I2C デバイスを検出出来ている。次に、i2cdump や i2cget コマンドでデータを確認。どのアドレスに何のデータがあるのかはスペックシートを確認すること。

```
$ i2cdump -y 1 0x48
$ i2cget -y 1 0x48 0x00 w    <-- 温度データ
$ i2cget -y 1 0x48 0x03 b    <-- 設定パラメータ
```

configure/REEASE では

```
STREAM=/opt/epics/R315.6/modules/soft/stream/2-7-7_I2C
```

をしていること。また、protocol file として (まだ少しおかしい部分があるが) とりえず以下の設定で読めるようだ。要調査。

```
pi@raspberrypi:~/epics/app/i2cTest1/protocol $ more adt7410.proto
Terminator   = "";
LockTimeout  = 500;
ReplyTimeout = 100;
ReadTimeout  = 100;
WriteTimeout = 100;
MaxInput     = 3;
ExtraInput   = Error;


get {
  out 0x48;
  in  "%03r";
}
```

Database は以下の通り：

```
record(mbbiDirect, "ADT7410:GET") {
  field(DTYP, "stream")
  field(INP,  "@adt7410.proto get() I2C")
  field(SCAN, ".2 second")
}
```

startup script では以下のパラメータを設定する

```
epicsEnvSet( "STREAM_WORKAROUND", "1")
## Load record instances
dbLoadRecords("db/adt7410.db")
drvAsynI2CConfigure("I2C", "/dev/i2c-1", 1)
```

その他の i2c デバイスも同様にチェックして、制御できる。Raspberry Pi から直接 I2C をつなぐべきか、Arduino などのマイコンで I2C を受けてある程度処理してから RPi に渡すなど、いくつか構成は考えられる。目的に応じて構成を検討する。

5 時間の余った方へ

- 実習に使用している GPS を見てみる
- CSS DataBrowser の使い方をもう少しやってみる
  - waveform データをアーカイブからとってきて (alan) 表示するには？
  - カーソルの表示
  - グラフに注釈 (annotation) を付けたい (例えば室温のデータ 2 日分取ってきて、どの時刻に何が起きたか記述)。plt ファイルとして保存した場合にもその時刻に注釈が残っていることを確認。
  - 水平軸を動かす：マウスで軸をつかんで左右に動かす
  - 複数のデータをプロットしておき、そのうち 1 つのデータの縦方向位置を動かしてみる: マウスで軸をつかむ
  - プロットしたデータを保存する。保存形式やオプションを変えてみる。
- CSS OPI の色々なウィジェットを試してみる
  - OPI Sample をいくつか触ってみる
  - Intensity Graph の例
- もう少し実用的な GPIO の使い方：直接 LED や負荷をつなぐのではなく、トランジスタや FET のスイッチ経由で制御する。
- Python で EPICS/Channel Access

次回以降のトピックス（中級・上級, システム管理他)。ご意見募集。

- デバイスサポートの書き方
- AA セットアップ、設定など（やまだ）
- Portable Archiver (路川）

参考文献

[1] https://elinux.org/RPi_BCM2835_GPIOs

[2] https://www.arduino.cc/

[3] https://www.arduino.cc/en/Main/Software

[4] http://wiringpi.com/

[5] https://www.gnu.org/software/screen/

[6] https://www-csr.bessy.de/control/SoftDist/sequencer