R4年度次世代技術活用人材育成事業
技術修得コース　実習座学（IoT）
第1回

茨城県産業技術イノベーションセンター
IT・マテリアルグループ

プログラムは下のURLでも確認できます。
https://github.com/Sangise/IoT_R4/tree/main/%E7%AC%AC1%E5%9B%9E

目次

＜前半＞
〇Raspberry Piとは？
〇Raspberry Piの特徴
〇Raspberry Piの構成
〇Raspberry PiによるLEDの点灯
〇プログラミングによるLEDの点滅
〇GPIO制御について
〇LED点滅プログラムの警告への対応
〇タクトスイッチでLEDの点灯

＜後半＞
〇AD変換とは？
〇mcp3208の仕様
〇分解能について
〇逐次比較法について
〇SPI通信
〇SPIモジュールの設定
〇AD変換器の仕様と回路構成
〇AD変換器の仕様と動作説明
〇AD変換器の実動作波形
〇AD変換読み取りプログラム
・実行・結果
・制御ビット部
・データ読込部
・GPIO初期設定、メイン処理

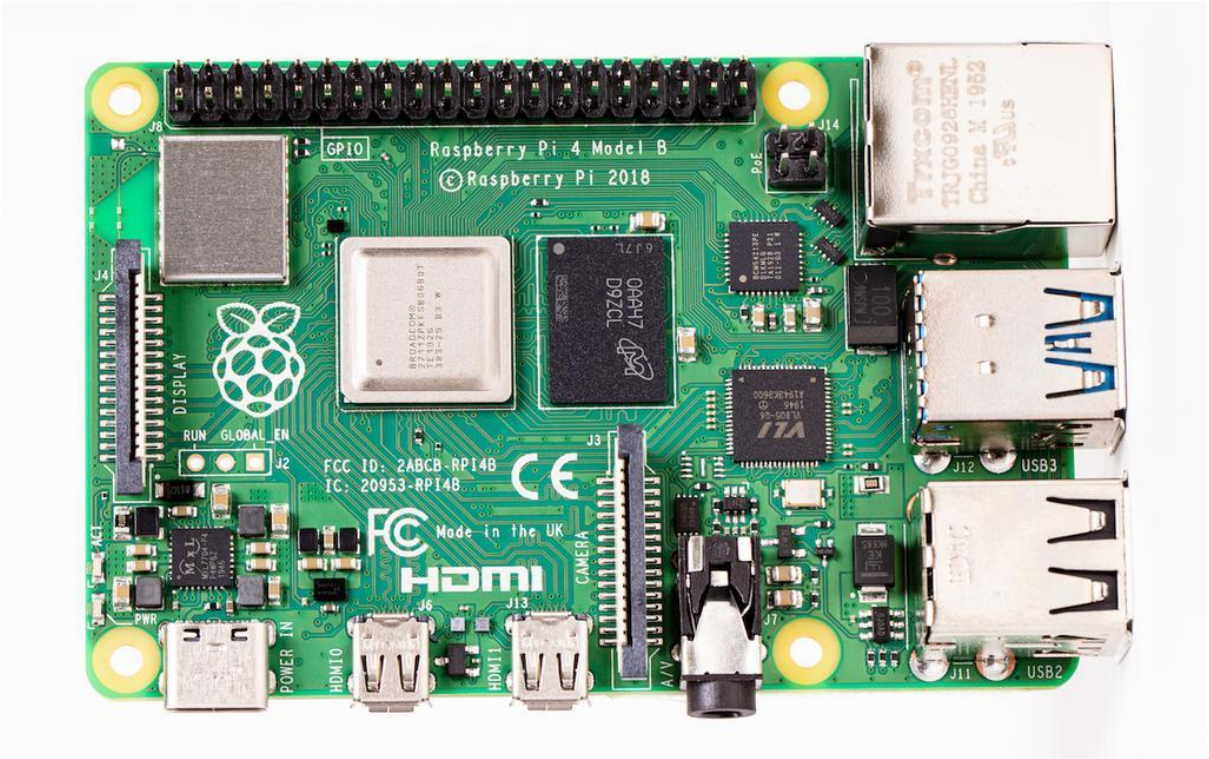
第一回　前半

ラズベリーパイでLEDを光らせる

Raspberry Piとは？

Raspberry Pi（ラズベリーパイ）は，10,000円から20,000円程度で購入できる小型のコンピュータ。キーボードやマウス，ディスプレイを接続することにより通常のコンピュータと同じように扱える。

2012年にRaspberry Pi財団から発売されて以降、販売台数が累計2000万台を超えたと言われている。



Raspberry Piの特徴

①ハードウェアとつなげやすい

「GPIO（General-Purpose Input/Output）」と呼ばれるピンがあり，ここにセンサやLEDなど様々な電子部品を接続して，プログラムにより直接制御できる。

②OSはLinuxベースのものが多く，プログラミングの学習環境として適している

Linux系OSは，最初からプログラミングの開発環境が導入されていることが多いため，プログラミングの環境に適している。インターネット上からOS(Linux)を無料でインストールできる。

③安価で小型・省電力であること

Raspberry Pi本体のみの価格は5000～6000円前後であり，壊れた場合でも代替品を購入することが比較的容易である。

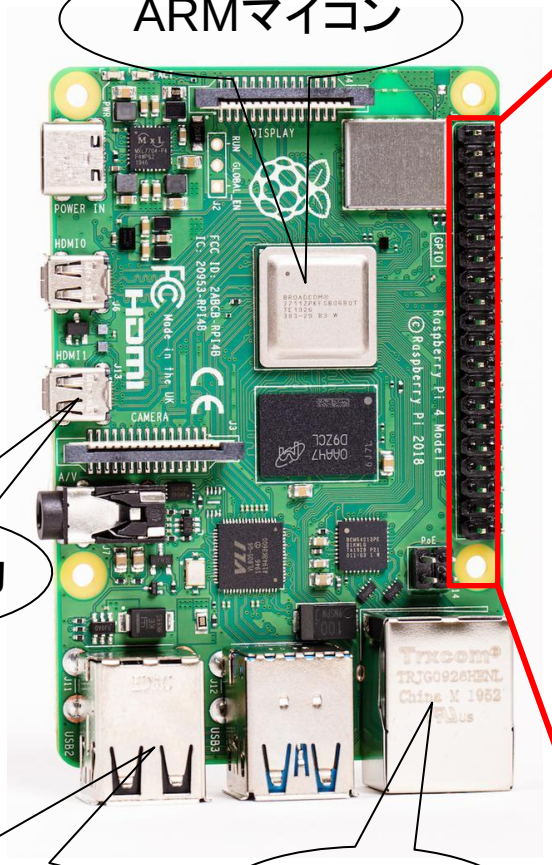
Raspberry Piのバージョンによるが消費電力は数W程度である。通常のコンピュータは数十Wの電力を消費するため，消費電力が1桁小さい。

Raspberry Piの構成

ワンボードマイコン→一枚のプリント基板にマイコンや入出力装置を取り付けたもの

Wi-Fi, Bluetoothを搭載しているシリーズも有

Raspberry Pi自体にOSは搭載されていないが，micro-SDを通じてインストール可能。



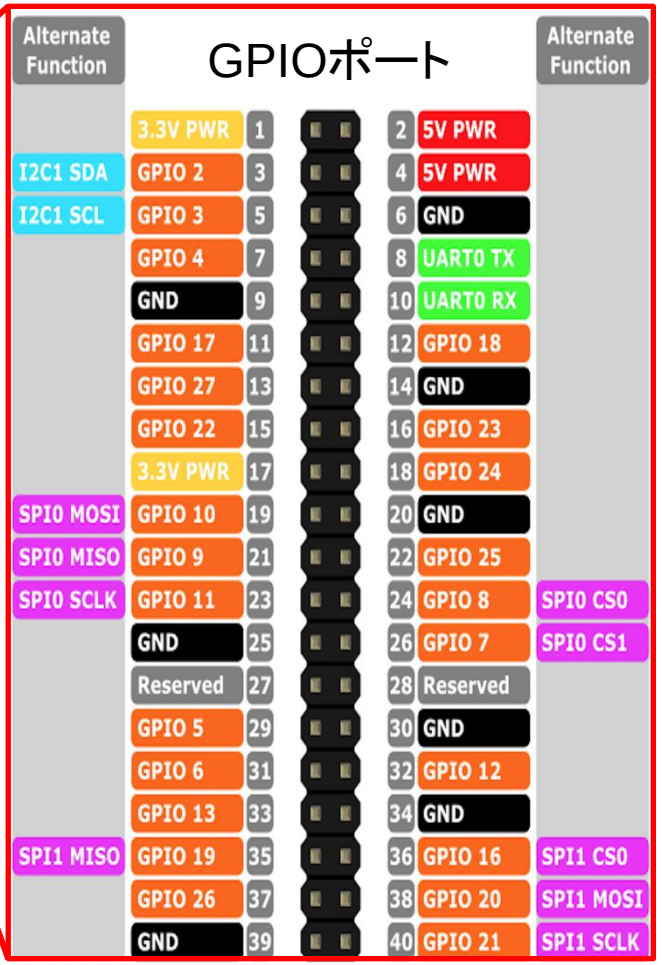


図2　Raspberry Pi3の構成

Raspberry Piの起動とシャットダウン

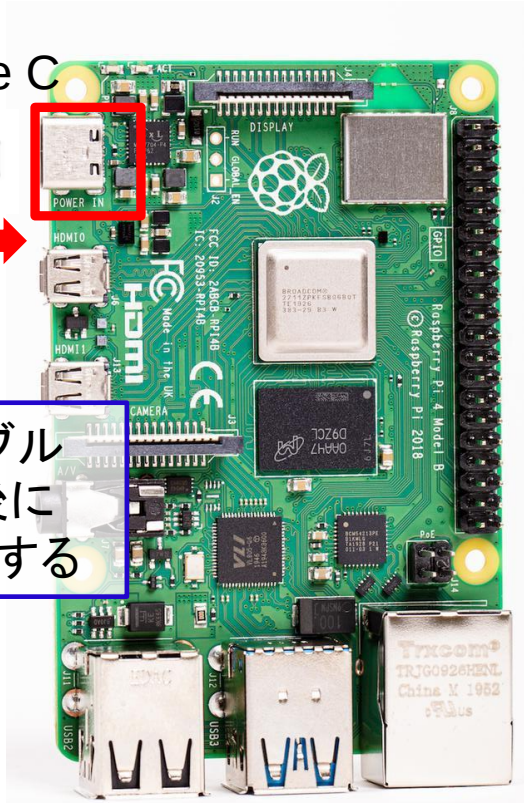
起動

Micro USBへ電源供給することで
Raspberry Piを起動することができる。
（OSイメージが書き込まれているSD
カードが挿入されているかを確認）

USB　Type C

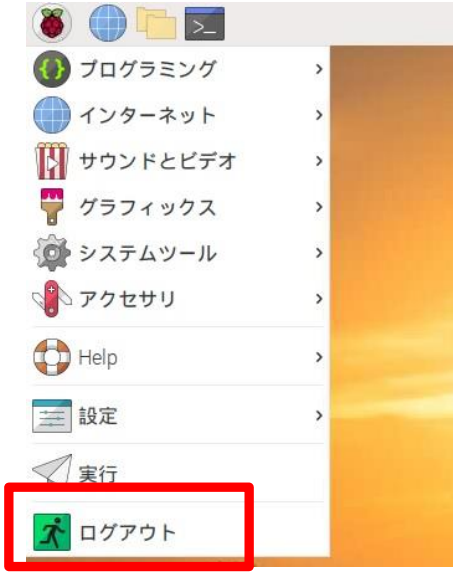
5V
2.5A以上推奨

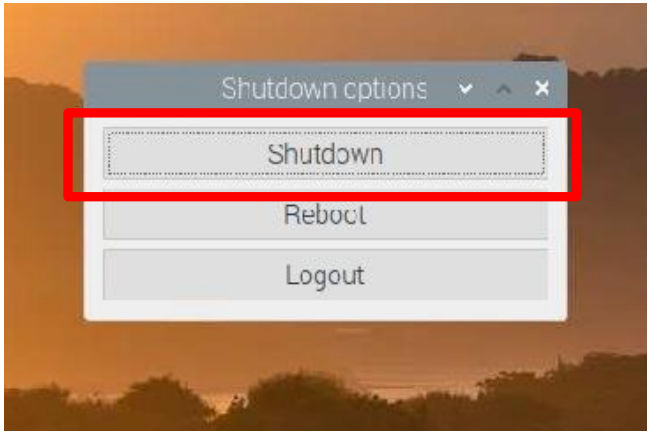
スイッチ付ケーブル
の場合は接続後に
スイッチをONにする



シャットダウン

- 画面左上のRaspberry Piのマーク を押すと，プルダウンメニューが表れる
- 一番下の「ログアウト」（機種によっては「Shutdown」）を選択する
- 「Shutdown options」が表示されるので「Shutdown」を選択する





- 完全に電源が落ちた後に，電源供給している
Micro USBを抜く（スイッチ付きケーブルの場合はスイッチをOFFにする）

Raspberry PiによるLEDの点灯

●必要なもの
330 Ωの抵抗（1本），LED（1個），
ブレッドボード，ジャンパー線

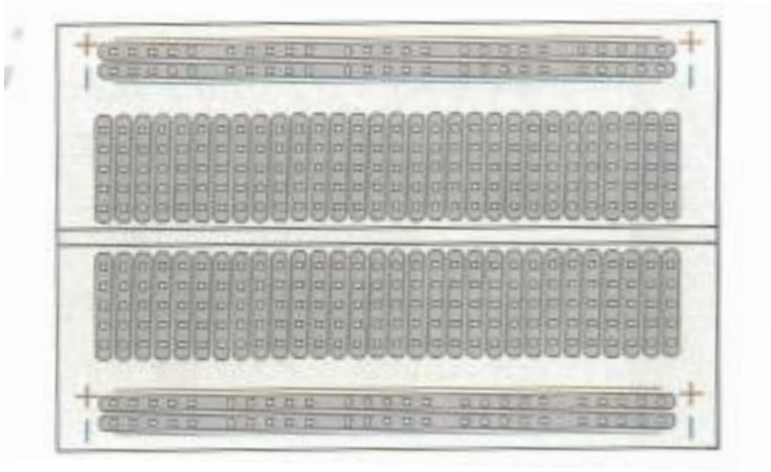


図3　ブレッドボードの内部接続

●ブレッドボードから接続を外す際の注意
回路作成時と同様，電源が入った状態で行って問題ないが，3.3 Vのピンに接続したジャンパー線とGNDに接続したジャンパー線の端子部分を接触させないように注意。
　接触させてしまうと，3.3 VピンとGNDピンの間で大きな電流が流れ，Raspberry Piが強制的の再起動する。故障に繋がる可能性あり。

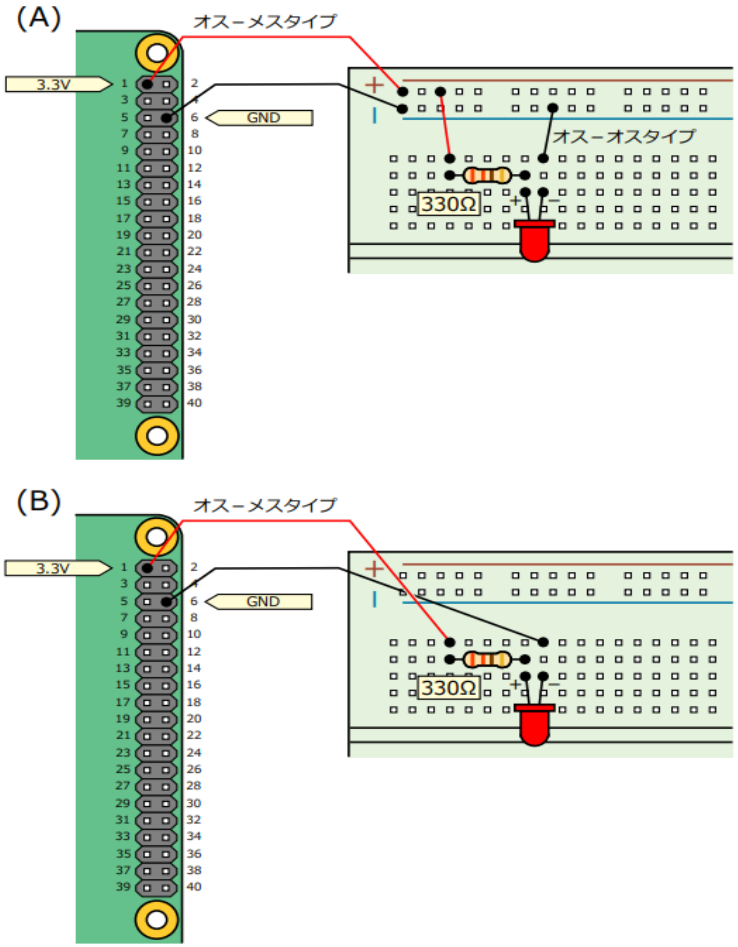


図4　ブレッドボードによる，LED点灯回路

プログラミングによるLEDの点滅①

①ファイルマネージャをクリック

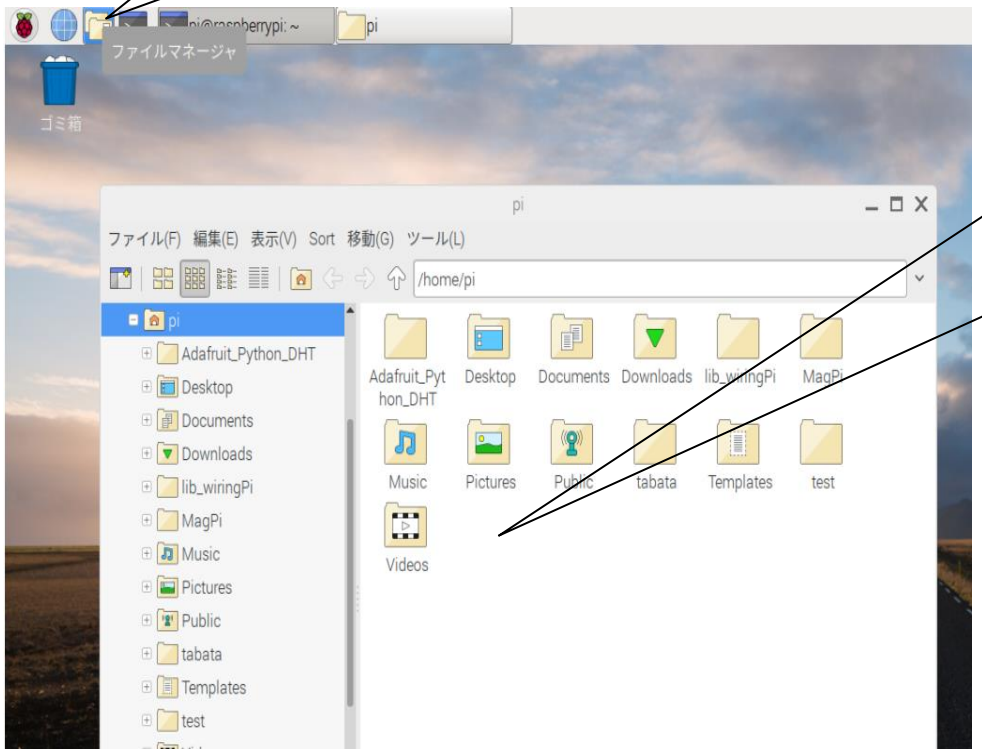


図5　フォルダ＆ファイル作成

②ここで右クリック→新規作成→フォルダ→「test」と入力

③作成した「test」フォルダ内で右クリック→新規作成→空のファイル→「test1.py」と入力

勉強会では「Python（パイソン）」と呼ばれるプログラミング言語を使用する。特徴として記述量が少なく，読みやすいプログラムを書くことができる。
なお，Raspberry Piの「Pi」はPythonの「パイ」から取られており，Raspberry Pi財団が推奨する学習用言語である。

プログラミングによるLEDの点滅②

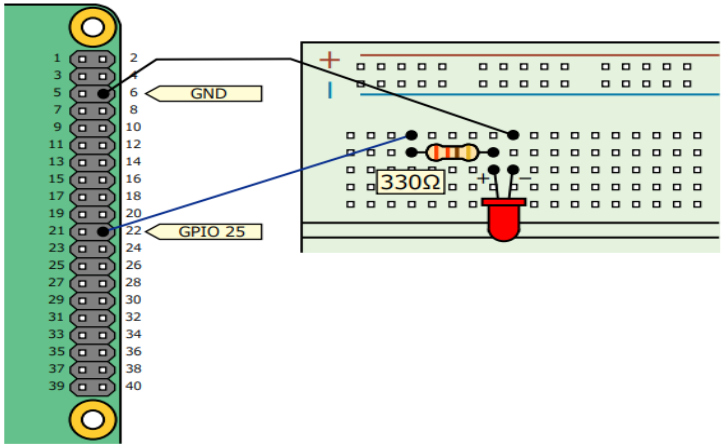


図6　ブレッドボードによる，LED点滅回路

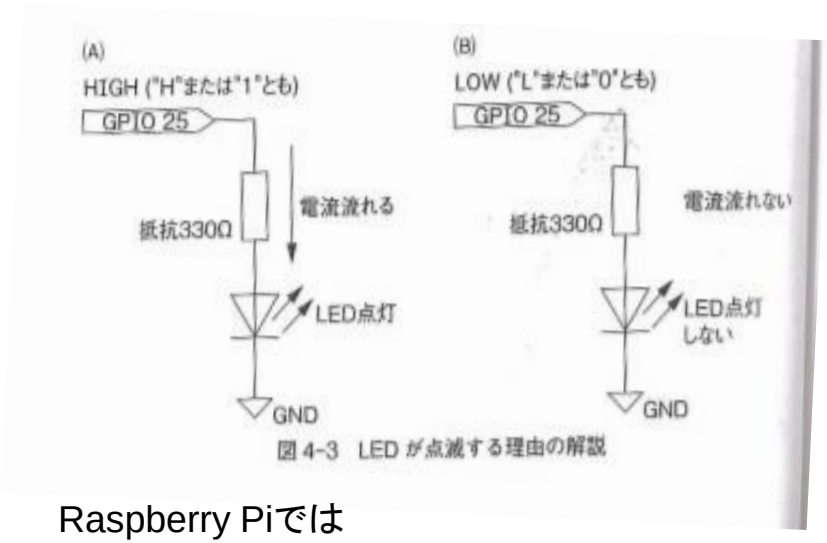
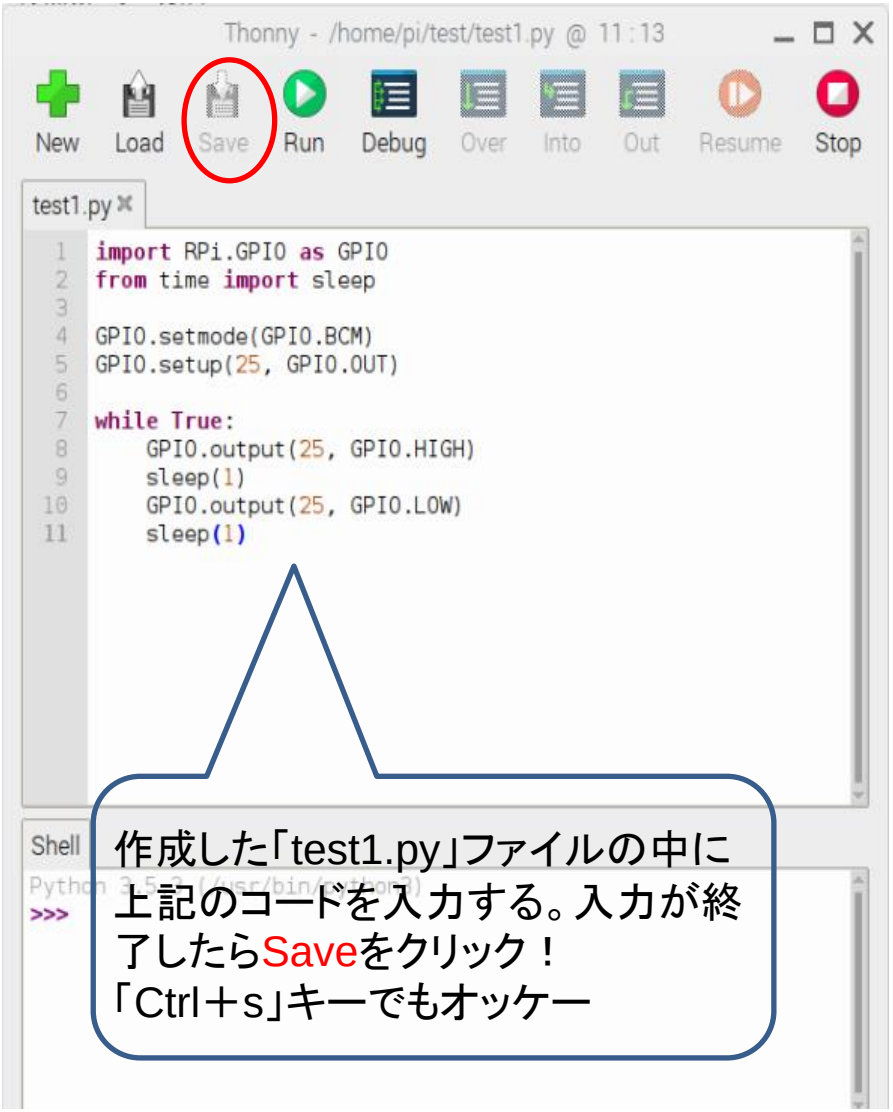


図 4-3　LED が点滅する理由の解説

Raspberry Piでは
HIGH → 3.3 V　LOW → 0 V



GPIO制御について

Raspberry PiではGPIO（入出力ピン）をpythonから操作するライブラリが標準で入っている。

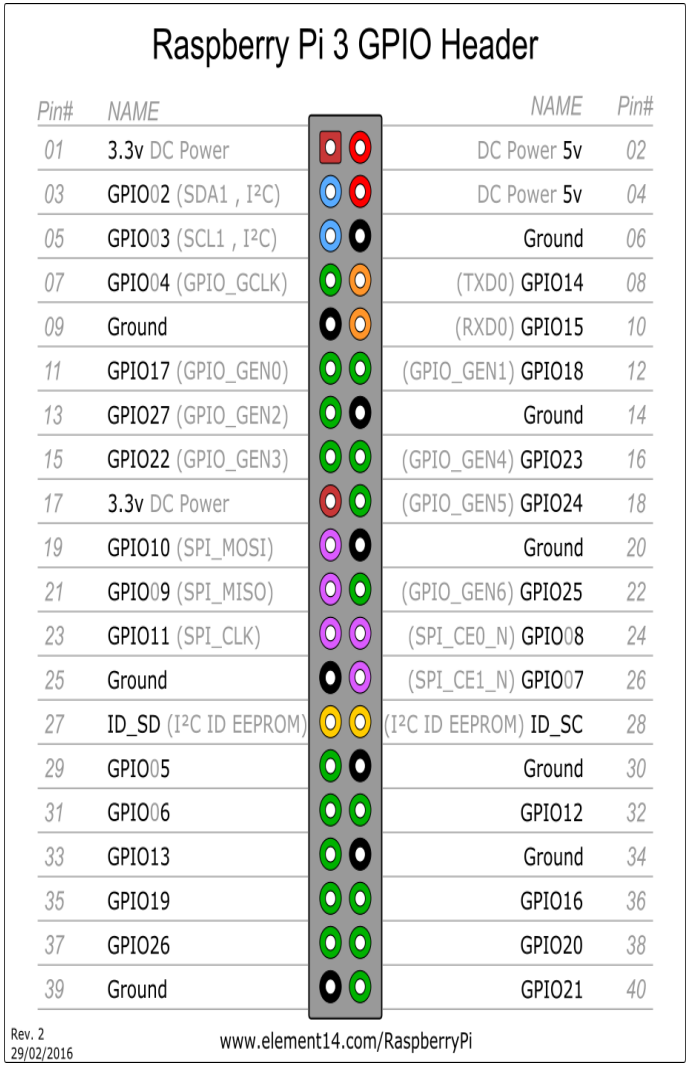
1. 準備： import RPi.GPIO as GPIO

2. モードの指定： GPIO.setmode(GPIO.BCM)
   - GPIO.BOARD：物理ピン番号（左上からの連番）
   - GPIO.BCM：役割ピン番号（broadcomが命名しているもの）

3. ピンの設定： GPIO.setup(ピン名， GPIO.OUT)
   - GPIO.IN ： 入力ピン
   - GPIO.OUT ： 出力ピン

4. ピンの操作： GPIO.output(ピン名， GPIO.HIGH)
   - GPIO.HIGH ： HIGH状態（3.3 V）
   - GPIO.LOW ： LOW状態（0 V）

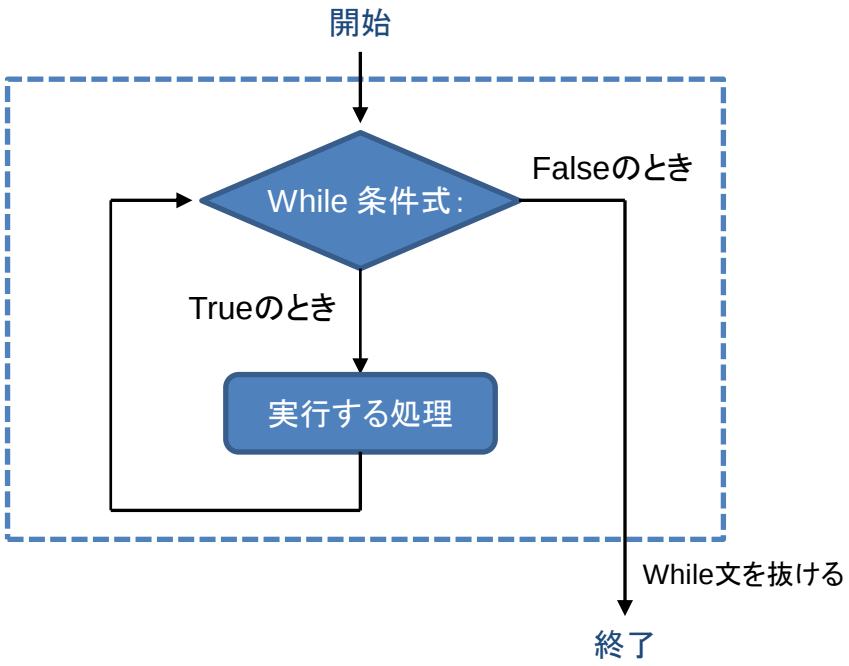
ピン名で指定したピンの出力をHIGHまたはLOWに設定する。



while文（構文）

ある条件を満たす間，処理を繰り返すループ処理の構文

＜フローチャート＞



[書式]
while 条件式：
　　# インデントの開始
　　ステートメント1
　　ステートメント2
　　ステートメント3
　　条件式がTrueの間，繰り返し実行される
インデントの終了(while文の終了)

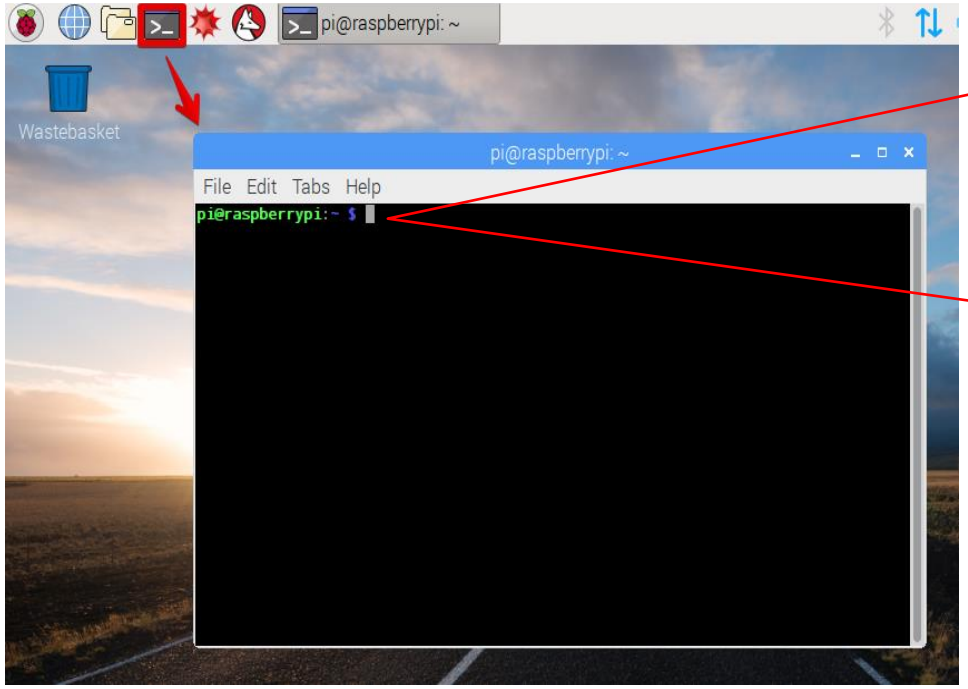
インデント：行頭に空白を入れて字下げを行うこと。

pythonにおけるインデントは重要

pythonは同じ数の空白でインデントされたまとまりを一つのブロックと認識するため。

プログラムの実行方法

LXTerminalを起動



①「pwd」と入力
現在いるディレクトリのことをカレント
ディレクトリというが、これを確認できる。

②「cd test」と入力
testディレクトリ内に移動する。

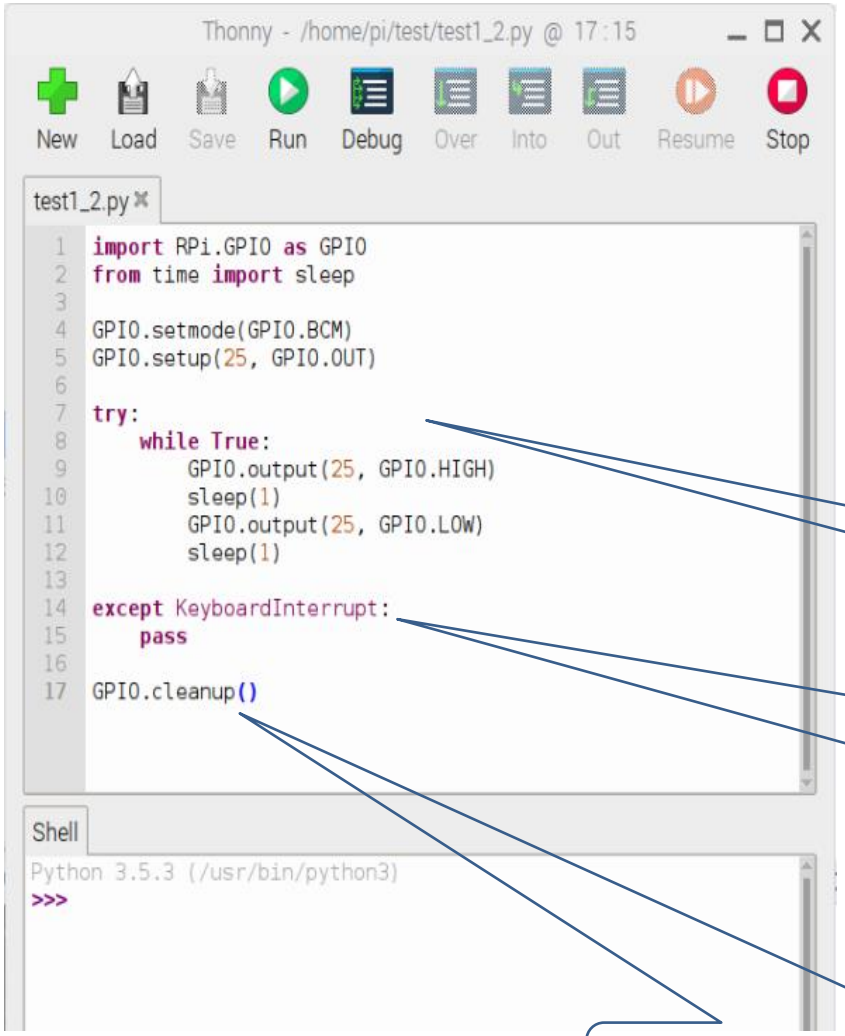
③「ls」と入力
testディレクトリ内のファイルを一覧できる。

④「python test1.py」と入力
プログラムを実行できる。

⑤「Ctrl」キーを押しながら「c」キーを入力
プログラムを終了できる。
（これを「Ctrl-c」と呼ぶ）

LEDが1秒おきに点滅すれば成功！

LED点滅プログラムの警告への対応



もう一度「test1.py」を実行すると「RuntimeWarning」（警告）が表示される。
この警告は「GPIO25を出力に設定しようとしているが，このピンは既に使用済み」という内容である。

1回目の終了時に，GPIOの出力設定を解除することで解消できる。

〇try
例外処理を行う。（詳細は次のスライドにて）

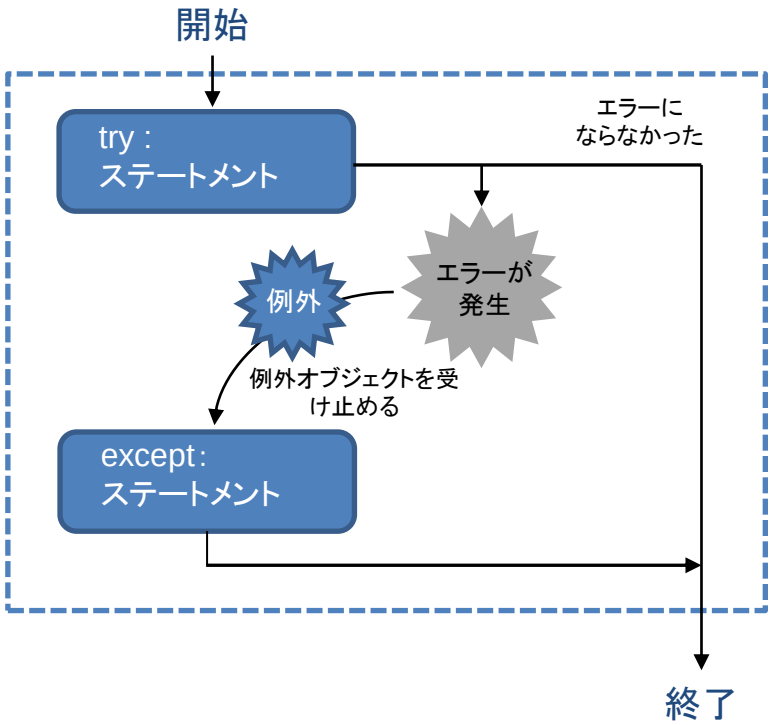
〇Except KeyboardInterrupt:
Ctrl-cでこの処理はこのブロックに移る。
〇pass
何もせずに次の命令に移る。

GPIOの後始末をしてプログラムを終了。

try ~ except文（構文）

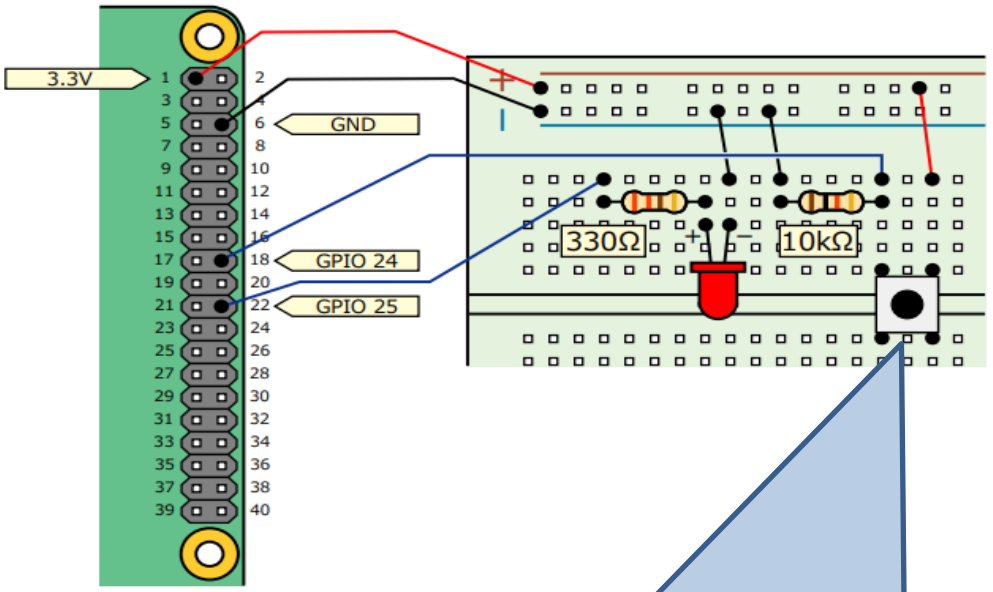
実行時にエラーが発生したとき，プログラムが止まらないように対応するコードを組み込んだ構文を例外処理と呼ぶ。

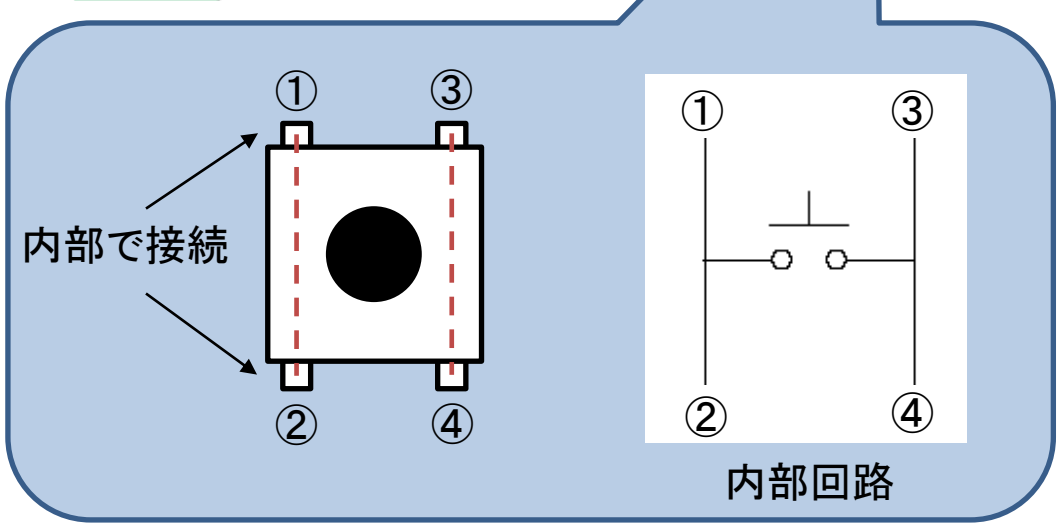
＜フローチャート＞



[書式]
try 条件式:
　# インデントの開始
　例外が発生する可能性がある処理
except (例外の種類):
　例外を受けて実行する処理

タクトスイッチでLEDを点灯





Thonny - /home/pi/test/test2.py @ 19:15

New Load Save Run Debug Over Into Out Resume Stop

test2.py

```
1  import RPi.GPIO as GPIO
2  from time import sleep
3
4  GPIO.setmode(GPIO.BCM)
5  GPIO.setup(25, GPIO.OUT)
6  GPIO.setup(24, GPIO.IN)
7
8  try:
9      while True:
10         if GPIO.input(24) == GPIO.HIGH:
11             GPIO.output(25, GPIO.HIGH)
12         else:
13             GPIO.output(25, GPIO.LOW)
14         sleep(0.01)
15
16 except KeyboardInterrupt:
17     pass
18
19 GPIO.cleanup()
```

GPIO 24を入力に設定。

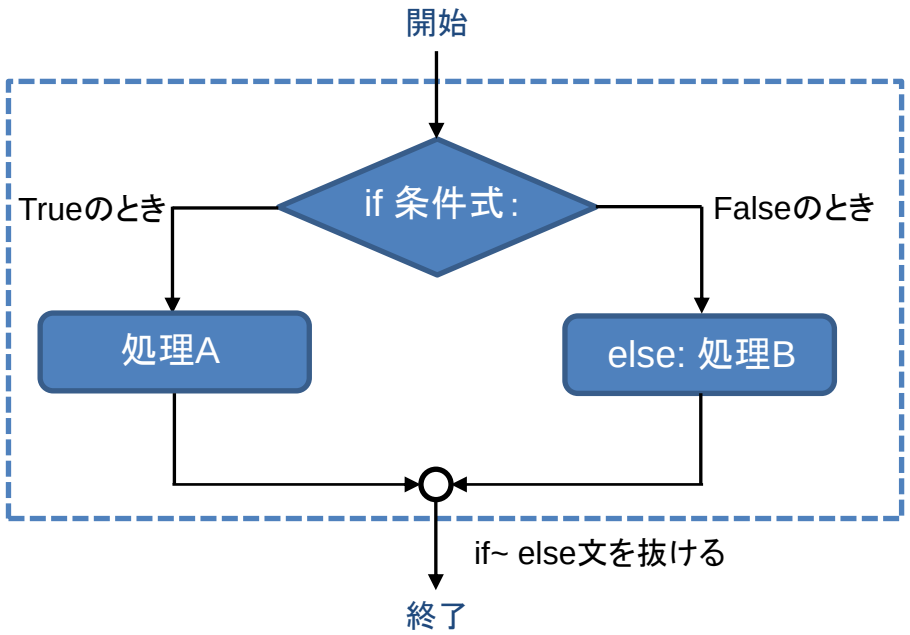
Shell

```
Python 3.5.3 (/usr/bin/python3)
>>>
```

if ~ else文（構文）

条件に合う場合と合わない場合の処理

＜フローチャート＞




```
[書式]
if 条件式：
    # 処理A
    ステートメント a1
    ステートメント a2
else:
    # 処理B
    ステートメント b1
    ステートメント b2
# インデントの終了
```

条件式がTrueのときに実行する

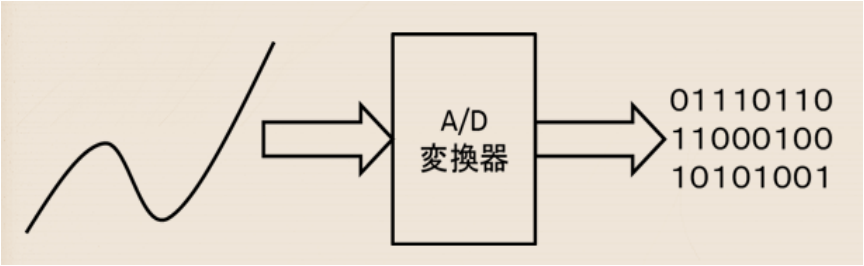
条件式がFalseのときに実行する

第一回　後半

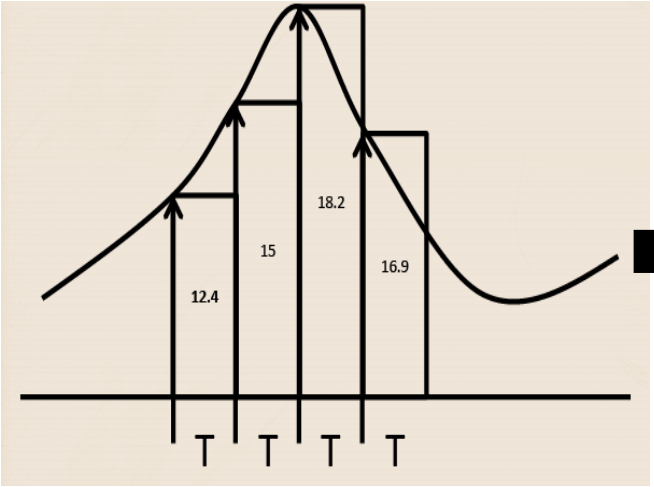
AD変換とSPI通信

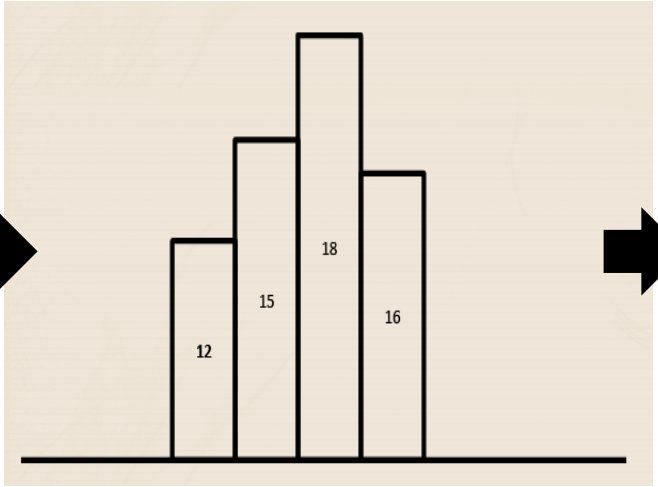
AD変換とは？

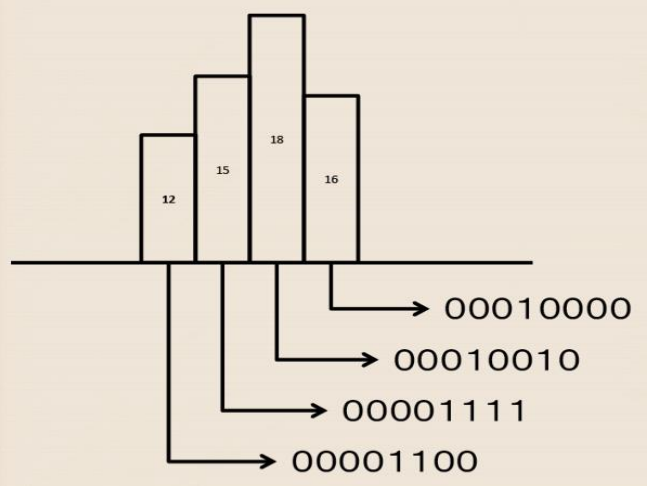
AD変換の概要



AD変換とは，左図のようにアナログ信号をディジタル信号に変換することである。







①標本化
アナログ信号を一定時間ごとに区切り，その値を読み込むこと（サンプリングとも呼ぶ）

②量子化
標本化し読み込んだ値をディジタル信号に変換できるように加工すること（量子化による誤差を量子化誤差と呼ぶ）

③符号化
量子化された値を指定された2進数の桁数で表現すること（この2進数の桁数を分解能と呼ぶ）

mcp3208の仕様

秋月電子通商

マイページ | 注文書 | お問い合わせ | かごの中身 | トラ技広告 | 回路図集

クイック注文：通販コードを入力
(アルファベット+数字)
数量 注文
通販コード一括入力フォーム

トップページ | 商品カタログ | 新商品 | お知らせ | 注文方法 | 振込先 | よくある質問 | ダウンロード | 配送状況確認 | ログイン 検索

トップ > 半導体 > インターフェースIC > ADコンバータ > 12bit 8ch ADコンバータ MCP3208-CI/P

AA

商品画像

RoHS2

12bit 8ch ADコンバータ MCP3208-CI/P

[MCP3208-CI/P]
通販コード I-00238
発売日 2002/08/28
メーカーカテゴリ Microchip Technology Inc.(マイクロチップ)/Atmel Corporation(アトメル)

マイクロチップの12ビットADコンバータ

Cグレード（INL ±2LSB）

※フラットタイプもあります→I-05813

MCP3208 PDFデータシート

この商品を友達に教える
お気に入りに追加する

店舗情報

関連商品»PIC(12F 16F 18F 24F) / H8 / RX / LCD / ﾒﾓﾘ / ICｿｹｯﾄ / LED(3mm 5mm 角型 SMD 点滅 ﾊﾟﾜｰ ﾊﾞｰ ﾌﾟｾﾞﾌﾞ) / TR(TO-220 TO-92 TO-3P SMD) / FET(Nch Pch SMD) / ﾀﾞｲｵｰﾄﾞ / ｺﾝﾃﾞﾝｻ / SW / 抵抗 / 水晶 / ｾﾝｻ / 基板(ﾕﾆﾊﾞｰｻﾙ SMD DIP化 変換 電力 ｱﾙﾐ) / ﾌﾞﾚｯﾄﾞﾎﾞｰﾄﾞ / ﾈｼﾞ / ﾋﾟﾝﾍｯﾀﾞ / ﾋﾟﾝｿｹｯﾄ / 線材 / ﾀｰﾐﾅﾙ / ｺﾈｸﾀ / 電池ｹｰｽ / 電池一般 / 工作用品 / ﾊﾟｰﾂ

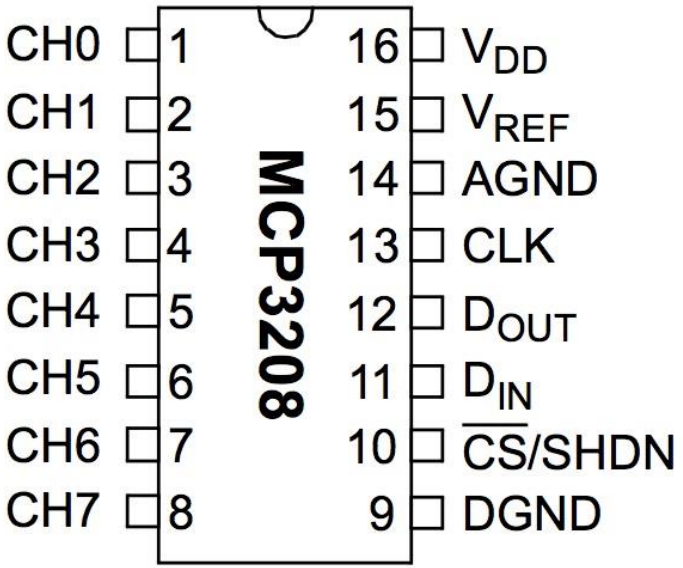
[I-00238] 12bit 8ch ADコンバータ MCP3208-CI/P AA

1個 ￥310（税込）

購入数量 1 個 かごに入れる かごの中身を見る

＜主な特徴＞

- 分解能　12bit（4095）
- 入力数　8チャンネル
- 動作電圧　2.7～5.5V
- 動作温度範囲　-40℃～85℃
- 変換方式　逐次比較法
- インターフェース　SPI



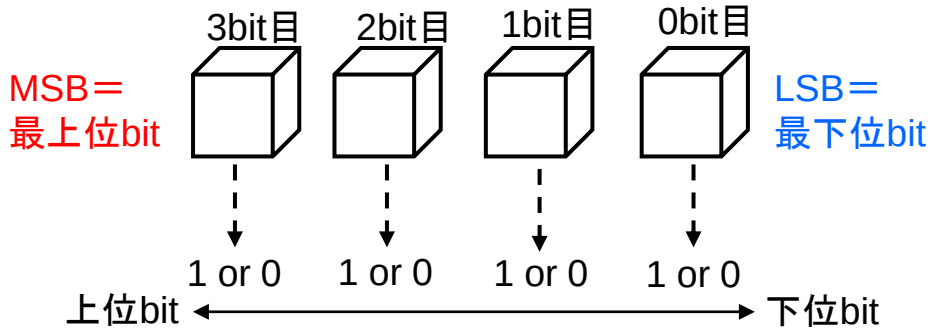
茨城県産業技術イノベーションセンター
Ibaraki Prefecture

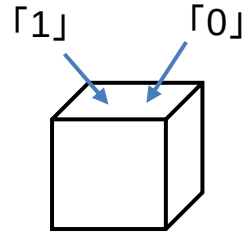
分解能について

分解能･･･アナログ信号をどの程度の細かさでデジタル表現（近似）できるかを示す。分解能が高いほど，アナログ値をより正確にデジタル値に変換できる。

コンピュータ上では，すべてのデータを2進数で表している。
2進数⇒「0」と「1」の2つの数字だけで，すべての数を表す方法。

（例）4bitの場合のイメージ
箱を4つ並べて考える。bitを増やすと扱える数が大きくなる。





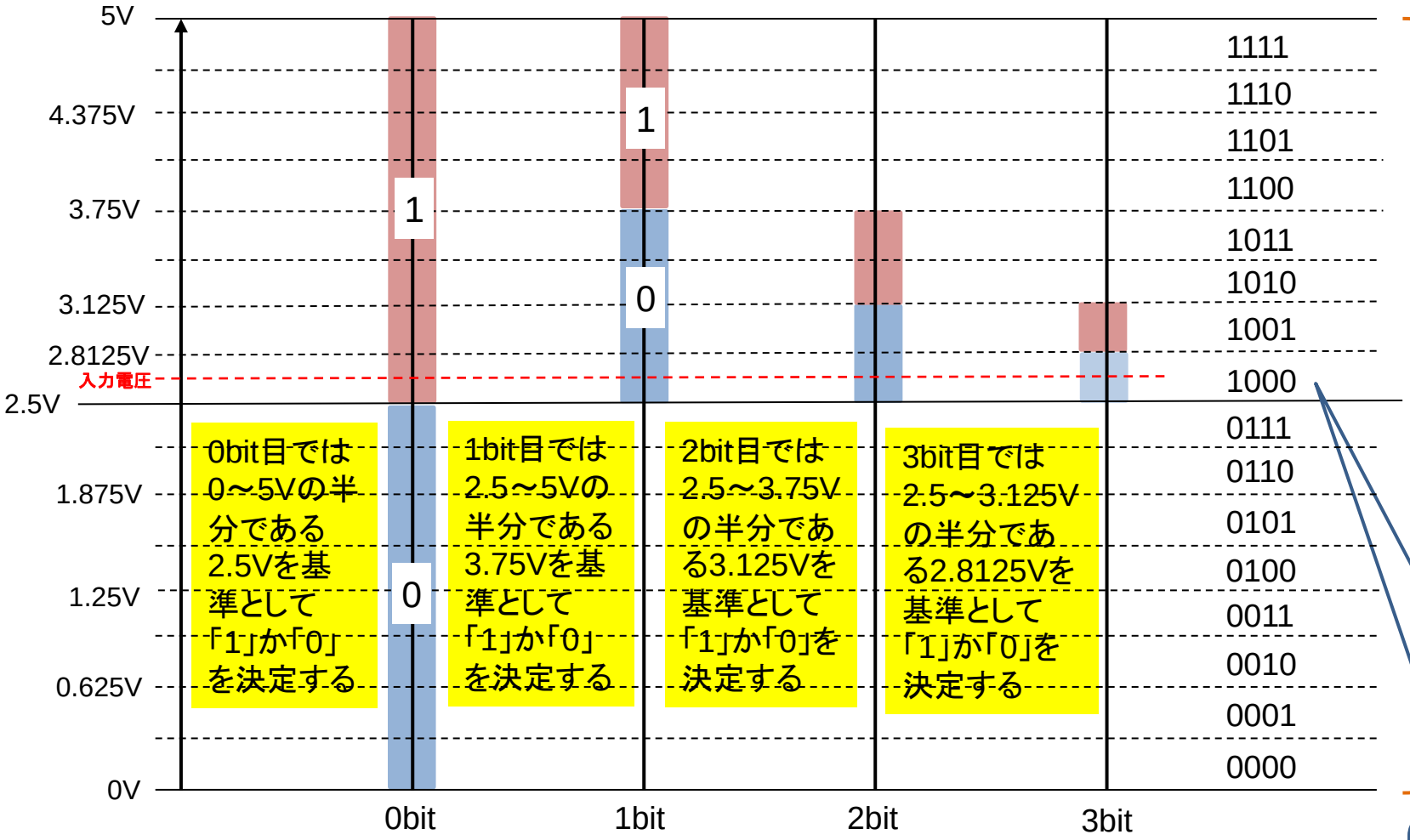
bitとは「1」か「0」が入る箱のようなイメージでデータの最小単位。
1bitでは1か0の2通りのデータしか扱えない。

表1 相互関係

2進数	10進数	16進数
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

◎n個のbitで表現できる数は2^n通りになる。
4bit → 2^4＝16通り
8bit → 2^8＝256通り
12bit → 2^{12}=4096通り

逐次比較法について



16分割するようなイメージ

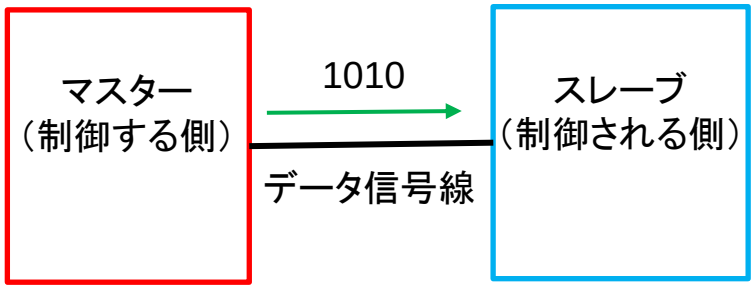
この例のAD変換値は“1000”になる。

4bitの分解能では2.5～2.8125Vの間に入力電圧があった場合，A/D変換結果はすべて“1000”となり誤差が大きい。

通信システム

シリアル通信

下図のように一本のデータ信号線を使ってデータを1bitずつ順番に通信する方式。「シリアル」とは「連続」や「一列」という意味。

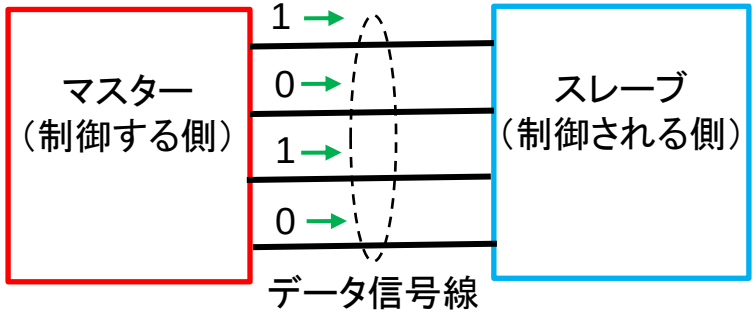


＜メリット＞
・配線が簡単。
・クロック（タイミング）ずれが発生しにくく，ノイズに強い。

＜デメリット＞
・通信速度がパラレル通信よりも遅い。

パラレル通信

下図のように複数のデータ信号線を使って何bitかのデータを一度に通信する方式。「パラレル」とは「並列」や「平行」という意味。



＜メリット＞
・一度に多くのデータを送れる。
・通信速度がシリアル通信より速い。

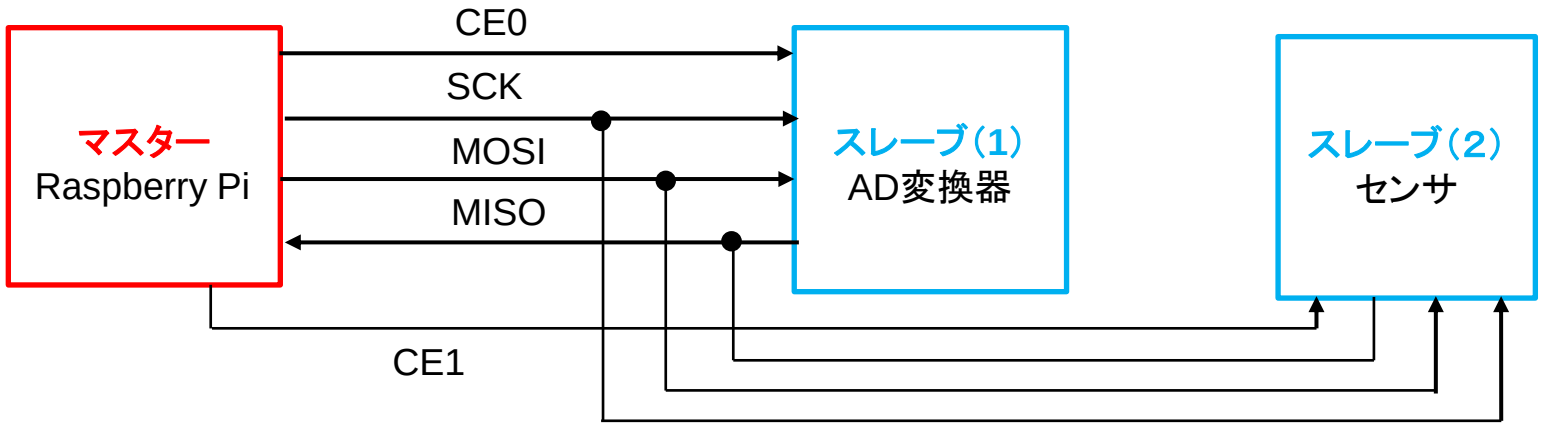
＜デメリット＞
・マイコンのI/Oポートは使える本数に制限があるため組み込みソフトウェアではあまり使われない。
・配線が複雑になる。（高コストになる）

SPI通信①

SPIとは，異なるデバイス間で通信するための規格。今回の例では，Raspberry Pi ～ mcp3208（AD変換器）間でSPI通信をさせる。

機能名	制御するラズパイのピン	意味
MOSI（Master Out Slave In）	#19（ポート10），#38（ポート20）	マスター → スレーブにデータを送信
MISO（Master In Slave Out）	#21（ポート9），#35（ポート19）	スレーブ → マスターにデータを送信
SCKL（Serial Clock）	#23（ポート11），#40（ポート21）	データ転送のタイミングを制御する
CE0/CE1（SS，CS）	#24（ポート8）/ #26（ポート7）	マスターとスレーブを決定する信号を送信する

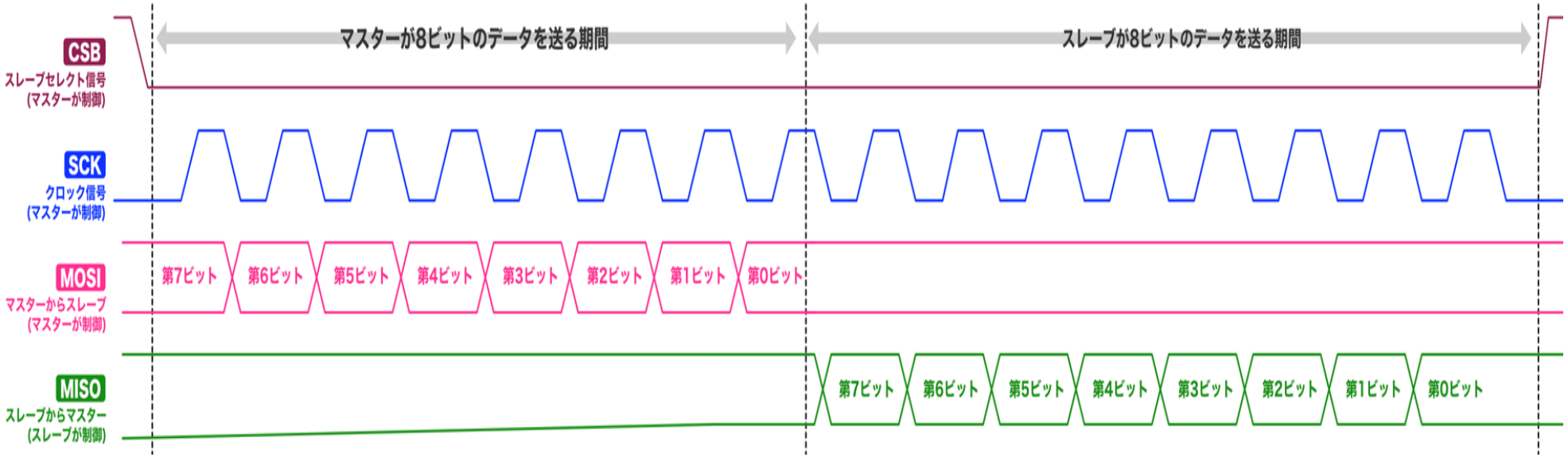
マスター（主人）：Raspberry Pi
スレーブ（奴隷）：AD変換器



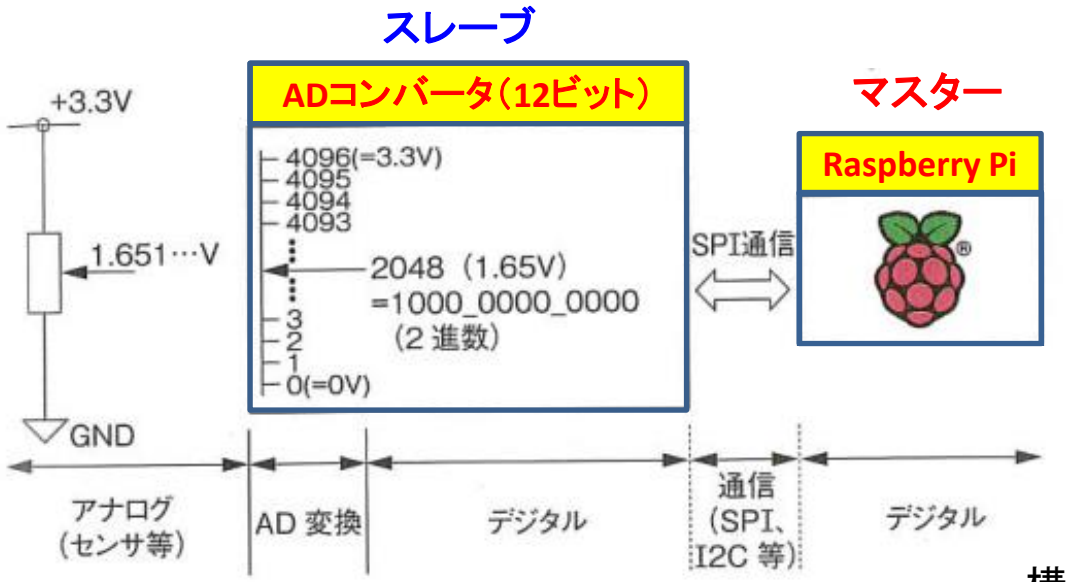
SPI通信参考ページ：https://tool-lab.com/make/pic-practice-23/

SPI通信②

順番	マスター動作	スレーブ動作
1	通信する相手のスレーブセレクト信号を0にして，通信相手を指定する	スレーブセレクト信号が0になったスレーブは，これからクロック信号に従ってデータ通信を行う
2	クロックを発生させて8ビットのデータを送信する（最上位ビットから送信）	クロック信号の立ち上がり/立ち下がりでMOSI信号を読み取り，8ビットデータを受信する
3	引き続きクロックを発生させて，クロックの立ち上がり/立ち下りでスレーブからの送信されるMISO信号を読み取り，8ビットデータを受信する	クロック信号に合わせて8ビットデータを送信する（最上位ビットから送信）
4	データの送受信が終わったら，スレーブセレクト信号を1にする	スレーブセレクト信号が1になったら通信処理終わり



Raspberry PiでAD変換



構成

■ AD変換されたデジタル量をRaspberry Piで読み取るために

① Raspberry Piのインターフェース設定で SPIを有効にする

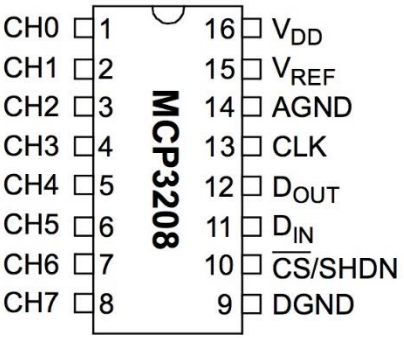
② ADコンバーター（MCP3208）の仕様を理解しょう

③ SPI（ **Serial Peripheral Interface**）通信の仕様を理解しよう

④ 回路工作して Pythonプログラムで動かしてみよう

⑤ Pythonプログラムを理解しよう

ADコンバータ（12ビット）



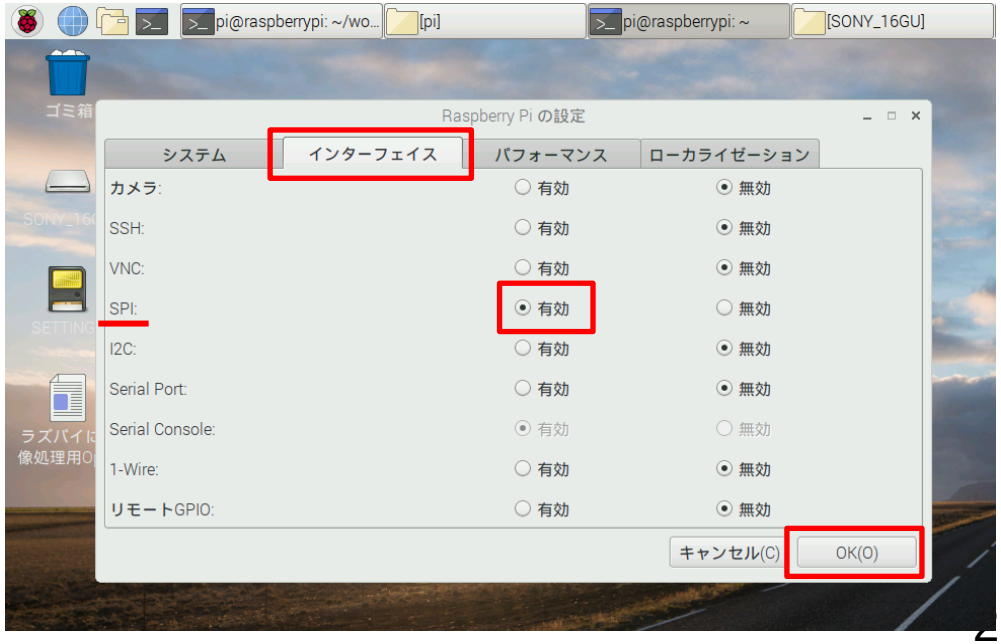
SPIモジュールの設定

まず、始めに・・・
Raspberry Piの設定からSPIモジュールを
読み込ませるための設定を行う。

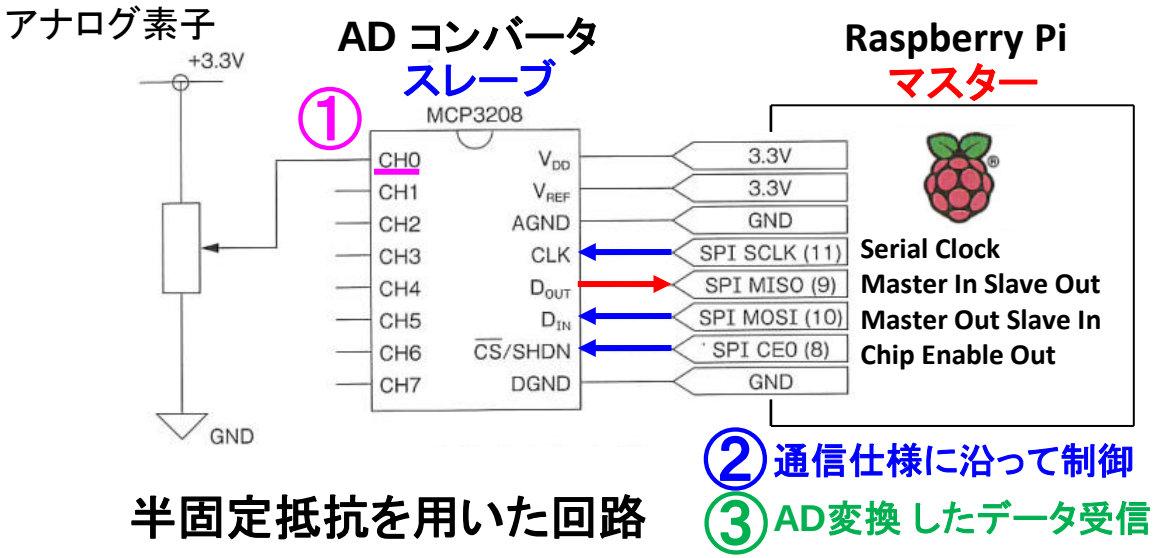
右図に示すように
左上のRaspberry Piのメニューボタンから、
「設定」＞「Raspberry Piの設定」を
クリックし、システム設定を開く。

次に「インターフェイス」タブをクリックし、
「SPI」が「有効」になっていることを確認する。
「無効」になっていた場合は「有効」を選択し、
「OK」ボタンをクリックする。
変更を有効にするため再起動する。
再起動：LXTerminalで「reboot」を実行する。

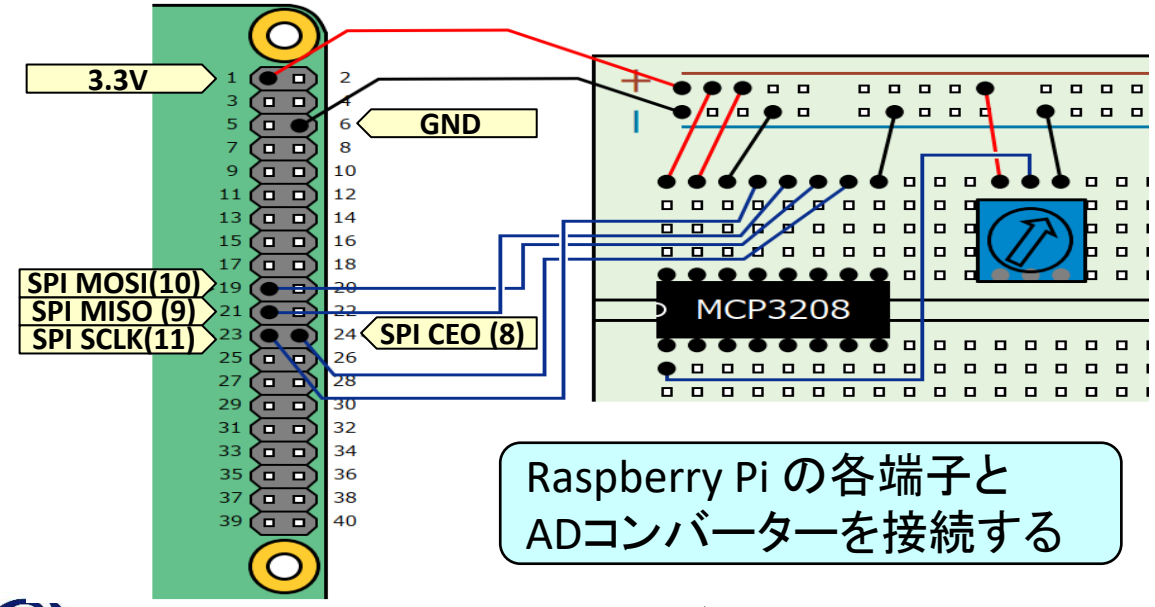
注）今回はSPIモジュールは使用しない。



ADコンバータの仕様と回路構成



半固定抵抗を用いた回路



Raspberry Pi の各端子と
ADコンバーターを接続する

半固定抵抗を用いた回路のブレッドボード上での回路

MCP3208の構成ビット

	制御ビット選択				入力構成	チャンネル選択
	SINGLE/DIFF	D2	D1	D0		
①	1	0	0	0	シングルエンド	CH0
	1	0	0	1	シングルエンド	CH1
	1	0	1	0	シングルエンド	CH2
	1	0	1	1	シングルエンド	CH3
	1	1	0	0	シングルエンド	CH4
	1	1	0	1	シングルエンド	CH5
	1	1	1	0	シングルエンド	CH6
	1	1	1	1	シングルエンド	CH7
	0	0	0	0	差動	CH0 = IN+ CH1 = IN-
	0	0	0	1	差動	CH0 = IN- CH1 = IN+
	0	0	1	0	差動	CH2 = IN+ CH3 = IN-
	0	0	1	1	差動	CH2 = IN- CH3 = IN+
	0	1	0	0	差動	CH4 = IN+ CH5 = IN-
	0	1	0	1	差動	CH4 = IN- CH5 = IN+
	0	1	1	0	差動	CH6 = IN+ CH7 = IN-
	0	1	1	1	差動	CH6 = IN- CH7 = IN+

AD変換器の仕様と動作説明（SPI通信のしくみ）

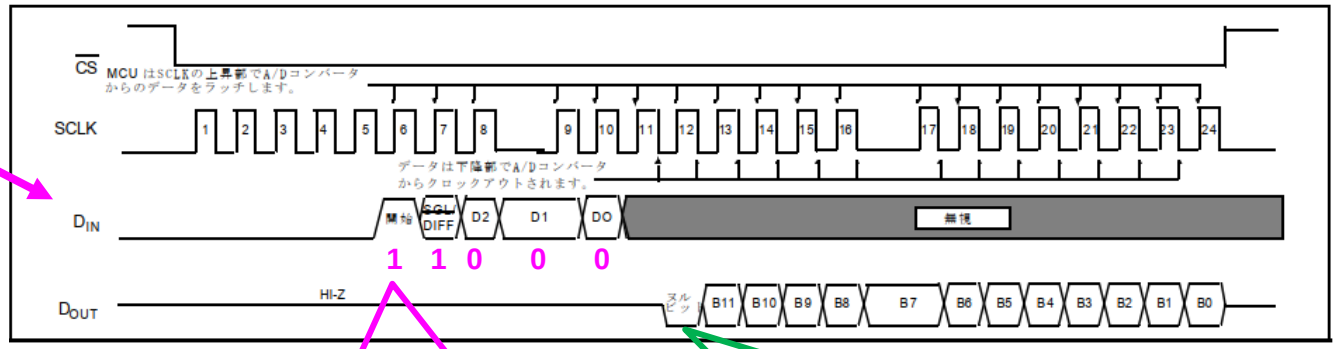
■ AD変換器（MCP3208）の**SPI通信仕様**の説明

Raspberry Pi からADに向かって **$\overline{\text{CS}}$（CEO）**を Lowにすることで ADを有効にする。
Raspberry Pi からADに向かって **クロック（SCLK）**を供給する。ADはこのクロックで動作する。
Raspberry Pi からADに向かって **Dinで AD変換器の制御ビット**を設定する。
★1　5ビットを設定する。「開始=**1**」「シングルエンド・入力=**1**」「CH0を選択：D2=**0**,D1=**0**,D0=**0**」
★2　クロックの立下りで確定させる。ADはクロックの立上がりでデータを受け取る。

AD変換器は、**Dout** を使いAD変換したデータをRaspberry Piに送信する。
D0を受け取って1クロックサイクル後、AD変換器は「ヌルビット=**0**」に続き、D11～D0の12ビットのデーターを、MSBのD11からクロックの立下りに同期して出力する。
Raspberry Pi は、クロックの立上りで12ビットのデータを受信する。

■ AD変換器の制御ビット仕様

制御ビット選択				入力構成	チャンネル選択
SINGLE/DIFF	D2	D1	D0		
1	0	0	0	シングルエンド	CH0
1	0	0	1	シングルエンド	CH1
1	0	1	0	シングルエンド	CH2
1	0	1	1	シングルエンド	CH3
1	1	0	0	シングルエンド	CH4
1	1	0	1	シングルエンド	CH5
1	1	0	0	シングルエンド	CH6
1	1	1	1	シングルエンド	CH7



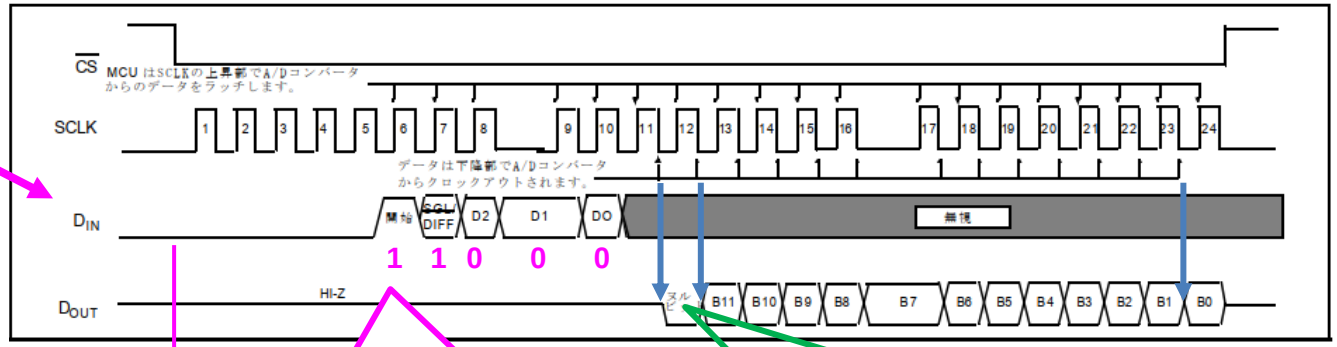
AD変換器をシングルエンドで
CH0 を使うように設定する

ヌルビットと AD変換した12ビットの
データがMSBから出力される

AD変換器の実動作波形

■ AD変換器の制御ビット仕様

制御ビット選択				入力構成	チャンネル選択
SINGLE/DIFF	D2	D1	D0		
1	0	0	0	シングルエンド	CH0
1	0	0	1	シングルエンド	CH1
1	0	1	0	シングルエンド	CH2
1	0	1	1	シングルエンド	CH3
1	1	0	0	シングルエンド	CH4
1	1	0	1	シングルエンド	CH5
1	1	0	0	シングルエンド	CH6
1	1	1	1	シングルエンド	CH7



AD変換器をシングルエンドで
CH0 を使うように設定する

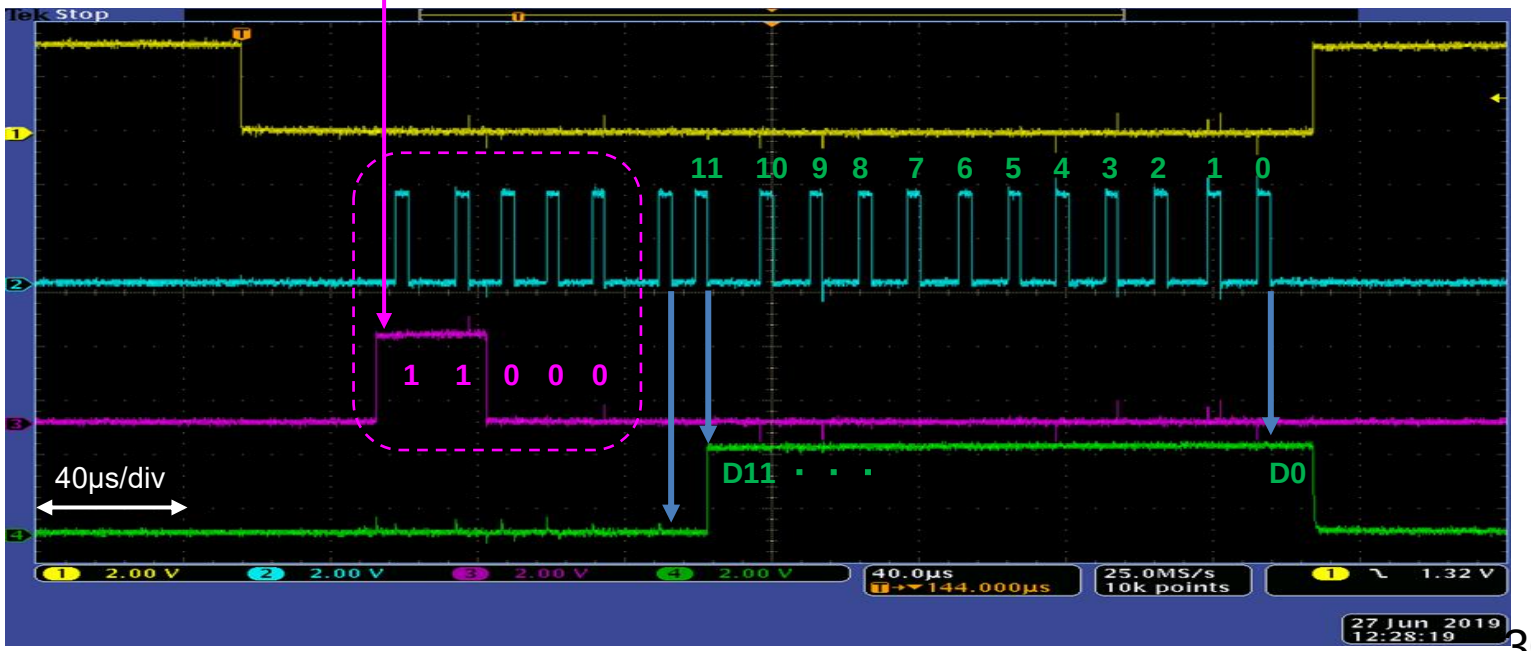
ヌルビットと AD変換した12ビットの
データがMSBから出力される

SPI_CEO（GPIO 8）

SPI_SCLK（GPIO 11）

SPI_MOSI（GPIO 10）

SPI_MISO（GPIO 9）



茨城県産業技術イノベーションセンター
Ibaraki Prefecture

AD変換読み取りプログラム

New Load Save Run Debug Over Into Out Resume Stop

ADC.py

```
# -*- coding: utf-8 -*-
import RPi.GPIO as GPIO
from time import sleep

def readadc(adcnum, clockpin, mosipin, misopin, cspin):
    if adcnum > 7 or adcnum < 0:
        return -1
    GPIO.output(cspin, GPIO.HIGH)
    GPIO.output(clockpin, GPIO.LOW)
    GPIO.output(cspin, GPIO.LOW)

    commandout = adcnum
    commandout = commandout | 0x18
    commandout = commandout << 3
    for i in range(5):
        if commandout & 0x80 == 0x80:
            GPIO.output(mosipin, GPIO.HIGH)
        else:
            GPIO.output(mosipin, GPIO.LOW)
        commandout = commandout << 1
        GPIO.output(clockpin, GPIO.HIGH)
        GPIO.output(clockpin, GPIO.LOW)

    adcout = 0
    for i in range(13):
        GPIO.output(clockpin, GPIO.HIGH)
        GPIO.output(clockpin, GPIO.LOW)
        adcout = adcout << 1
        if i>0 and GPIO.input(misopin)==GPIO.HIGH:
            adcout = adcout | 0x1

    GPIO.output(cspin, GPIO.HIGH)
    return adcout
```

New Load Save Run Debug Over Into Out Resume Stop

ADC.py *

```

GPIO.setmode(GPIO.BCM)
SPICLK = 11
SPIMOSI = 10
SPIMISO = 9
SPICS = 8

GPIO.setup(SPICLK, GPIO.OUT)
GPIO.setup(SPIMOSI, GPIO.OUT)
GPIO.setup(SPIMISO, GPIO.IN)
GPIO.setup(SPICS, GPIO.OUT)

try:
    while True:
        inputVal0 = readadc(0, SPICLK, SPIMOSI, SPIMISO, SPICS)
        print(" ADC = " + str(inputVal0))
        sleep(1.0)

except KeyboardInterrupt:
    pass

GPIO.cleanup()

```

AD変換読み取りプログラム（概要）

New Load Save Run Debug Over Into Out Resume Stop

ADC.py

```
# -*- coding: utf-8 -*-
import RPi.GPIO as GPIO
from time import sleep

def readadc(adcnum, clockpin, mosipin, misopin, cspin):
    if adcnum > 7 or adcnum < 0:
        return -1
    GPIO.output(cspin, GPIO.HIGH)
    GPIO.output(clockpin, GPIO.LOW)
    GPIO.output(cspin, GPIO.LOW)

    commandout = adcnum
    commandout = commandout | 0x18
    commandout = commandout << 3
    for i in range(5):
        if commandout & 0x80 == 0x80:
            GPIO.output(mosipin, GPIO.HIGH)
        else:
            GPIO.output(mosipin, GPIO.LOW)
        commandout = commandout << 1
        GPIO.output(clockpin, GPIO.HIGH)
        GPIO.output(clockpin, GPIO.LOW)

    adcout = 0
    for i in range(13):
        GPIO.output(clockpin, GPIO.HIGH)
        GPIO.output(clockpin, GPIO.LOW)
        adcout = adcout << 1
        if i>0 and GPIO.input(misopin)==GPIO.HIGH:
            adcout = adcout | 0x1

    GPIO.output(cspin, GPIO.HIGH)
    return adcout
```

ADからデータを読み取る関数
関数名：redadc

New Load Save Run Debug Over Into Out Resume Stop

ADC.py *

```
GPIO.setmode(GPIO.BCM)
SPICLK = 11
SPIMOSI = 10
SPIMISO = 9
SPICS = 8

GPIO.setup(SPICLK, GPIO.OUT)
GPIO.setup(SPIMOSI, GPIO.OUT)
GPIO.setup(SPIMISO, GPIO.IN)
GPIO.setup(SPICS, GPIO.OUT)

try:
    while True:
        inputVal0 = readadc(0, SPICLK, SPIMOSI, SPIMISO, SPICS)
        print(" ADC = " + str(inputVal0))
        sleep(1.0)

except KeyboardInterrupt:
    pass

GPIO.cleanup()
```

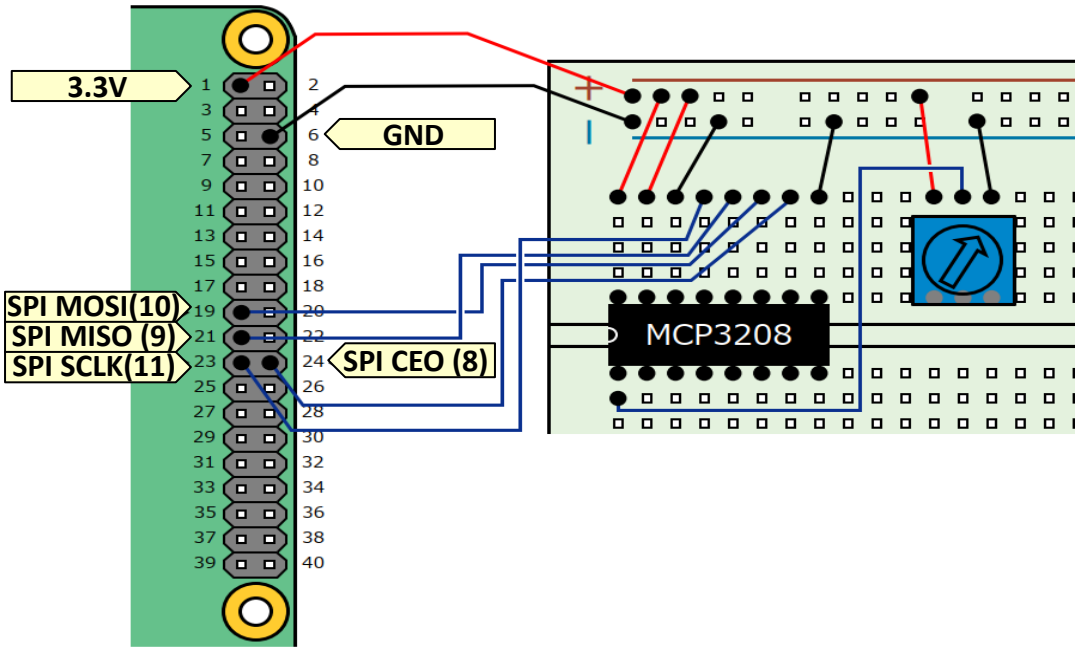
GPIOの設定

メイン処理
AD読込関数（readadc）を起動し
データを inputVal0 に格納、表示

今回は、時間が不足する可能性があるので
work フォルダに ADC.py ファイルを準備。
test フォルダにコピーして使用して下さい。

プログラムの実行・結果

■ ADC.py を実行してみましょう




```
pi@raspber
ファイル(F)  編集(E)  タブ(T)  ヘルプ(H)
^Cpi@raspberrypi:~/work $ python ADC.py
 ADC = 4095
 ADC = 4092
 ADC = 4094
 ADC = 4092
 ADC = 4094
 ADC = 4095
 ADC = 4090
 ADC = 3856
 ADC = 3388
 ADC = 2806
 ADC = 2452
 ADC = 2131
 ADC = 1450
 ADC = 845
 ADC = 567
 ADC = 404
 ADC = 279
 ADC = 272
 ADC = 283
 ADC = 0
 ADC = 0
pi@raspberrypi:~/work $
```

このプログラムを実行すると、AD変換で読み取った電圧に対応する値が Python Shell に表示される。
ボリウム（半固定抵抗）のつまみを回すことで、値が変化する。12bit 階調 4095～0。

プログラムの説明（GPIO初期設定、メイン処理）

New Load Save Run Debug Over Into Out Resume Stop

ADC.py *

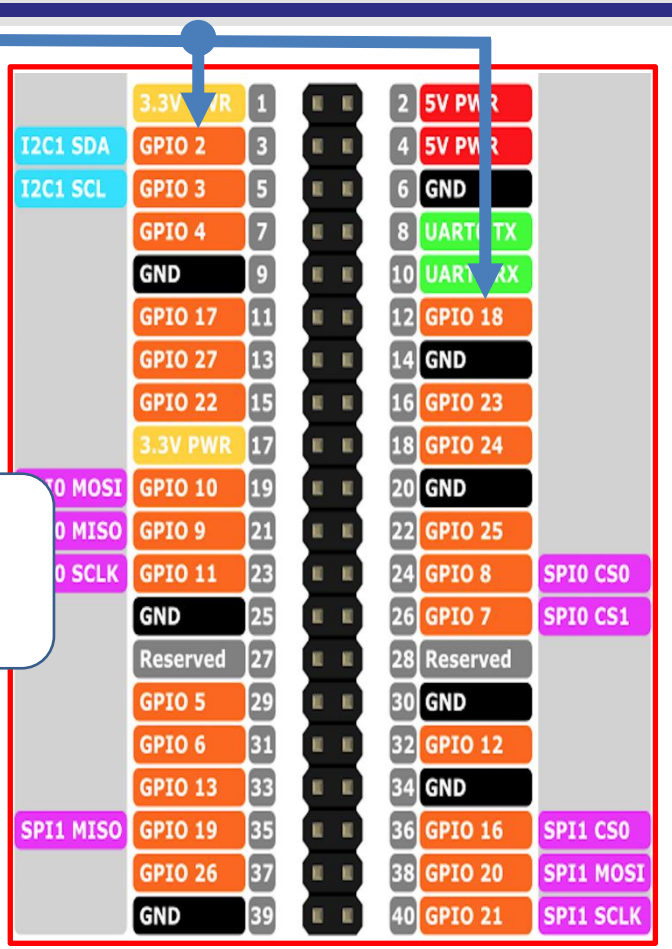
```
34
35  GPIO.setmode(GPIO.BCM)
36  SPICLK = 11
37  SPIMOSI = 10
38  SPIMISO = 9
39  SPICS = 8
40
41  GPIO.setup(SPICLK, GPIO.OUT)
42  GPIO.setup(SPIMOSI, GPIO.OUT)
43  GPIO.setup(SPIMISO, GPIO.IN)
44  GPIO.setup(SPICS, GPIO.OUT)
45
46  try:
47      while True:
48          inputVal0 = readadc(0, SPICLK, SPIMOSI, SPIMISO, SPICS)
49          print(" ADC = " + str(inputVal0))
50          sleep(1.0)
51
52  except KeyboardInterrupt:
53      pass
54
55  GPIO.cleanup()
56
```

GPIO（General-purpose input/output）
汎用入出力 を BCM番号で定義する

GPIOのBCM番号での設定
ピンの名前を変数として定義

初期設定
同じ

GPIO.setup (11, GPIO.OUT)
GPIO.setup (10, GPIO.OUT)
GPIO.setup (9, GPIO.IN)
GPIO.setup (8, GPIO.OUT)



メイン処理。
AD読込関数（**readadc**）を起動し、データを inputVal0 に格納。
データ（ inputVal0 ）を表示。
1.0秒ウエイト。
割り込み（KeybordInterrupt）が入るまで、繰り返す（約1.0 秒おきにデータ更新）

プログラムの詳細説明（制御ビット）

redadc：引数がある関数　5～33行

引数　adcnum：チャンネル選択 0～7

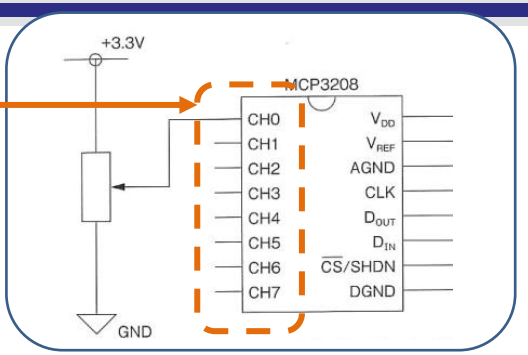
clockpin : SPICLK = 11

mosipin : SPIMOSI = 10

misopin : SPIMISO = 9

cspin : SPICS = 8

37～46行で設定



New　Load　Save　Run

ADC.py

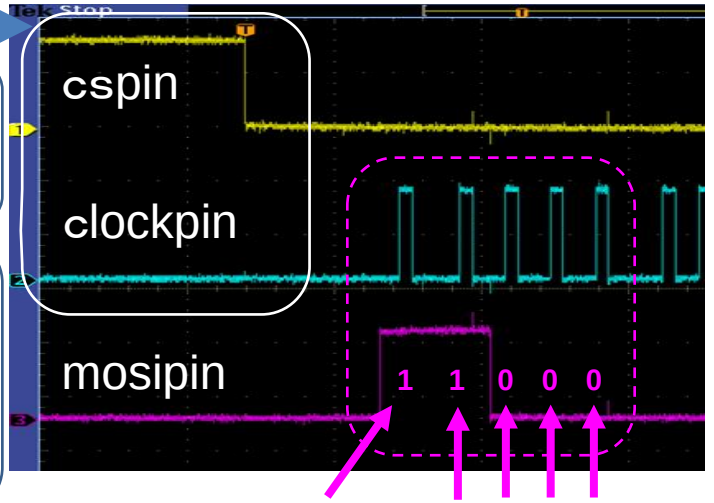
```
1  # -*- coding:
2  import RPi.GP  as GPIO
3  from time  import sleep
4
5  def readadc(adcnum, clockpin, mosipin, misopin
6      if adcnum > 7 or adcnum < 0:
7          return -1
8      GPIO.output(cspin, GPIO.HIGH)
9      GPIO.output(clockpin, GPIO.LOW)
10     GPIO.output(cspin, GPIO.LOW)
11
12     commandout = adcnum
13     commandout = commandout | 0x18
14     commandout = commandout << 3
15     for i in range(5):
16         if commandout & 0x80 == 0x80:
17             GPIO.output(mosipin, GPIO.HIGH)
18         else:
19             GPIO.output(mosipin, GPIO.LOW)
20         commandout = commandout << 1
21         GPIO.output(clockpin, GPIO.HIGH)
22         GPIO.output(clockpin, GPIO.LOW)
23
24
25
26
27
28
29
30
31
32
33
```

ADチャンネル 0～7 以外が設定されたら -1 を返して終了

初期設定：CSをLowにしてデバイス選択

commandout に adcnum (0 ～ 7)を入れる
commandout = 00000000 OR 00011000 ⇐ 0x18
= 00011000 ⇐ ビット毎のOR した結果
= 11000000 ⇐ 3ビット左シフト した結果

range(5)：下記操作を5回繰り返す
if commandout のMSBが1だったら
mosipin を Highにする
else 上記条件がFalseなら
mosipin を Lowにする
1ビット左シフトする
クロックを Hight、Low する



1回目　11000000 & 0x80 (10000000) ⇒ 10000000 ⇒ 0x80　0x80 と等しいので mosipin をHigh
2回目　10000000 & 0x80 (10000000) ⇒ 10000000 ⇒ 0x80　0x80 と等しいので mosipin をHigh
3回目　00000000 & 0x80 (10000000) ⇒ 10000000 ⇒ 0x00　0x80 と等しくない　mosipin をLow
4回目　00000000 & 0x80 (10000000) ⇒ 10000000 ⇒ 0x00　0x80 と等しくない　mosipin をLow
5回目　00000000 & 0x80 (10000000) ⇒ 10000000 ⇒ 0x00　0x80 と等しくない　mosipin をLow

プログラムの詳細説明（データの読み込み）

AD読込データの初期値を 0 にしD13のヌルビット（0）とデータ12ビットの計13ビットを読込む

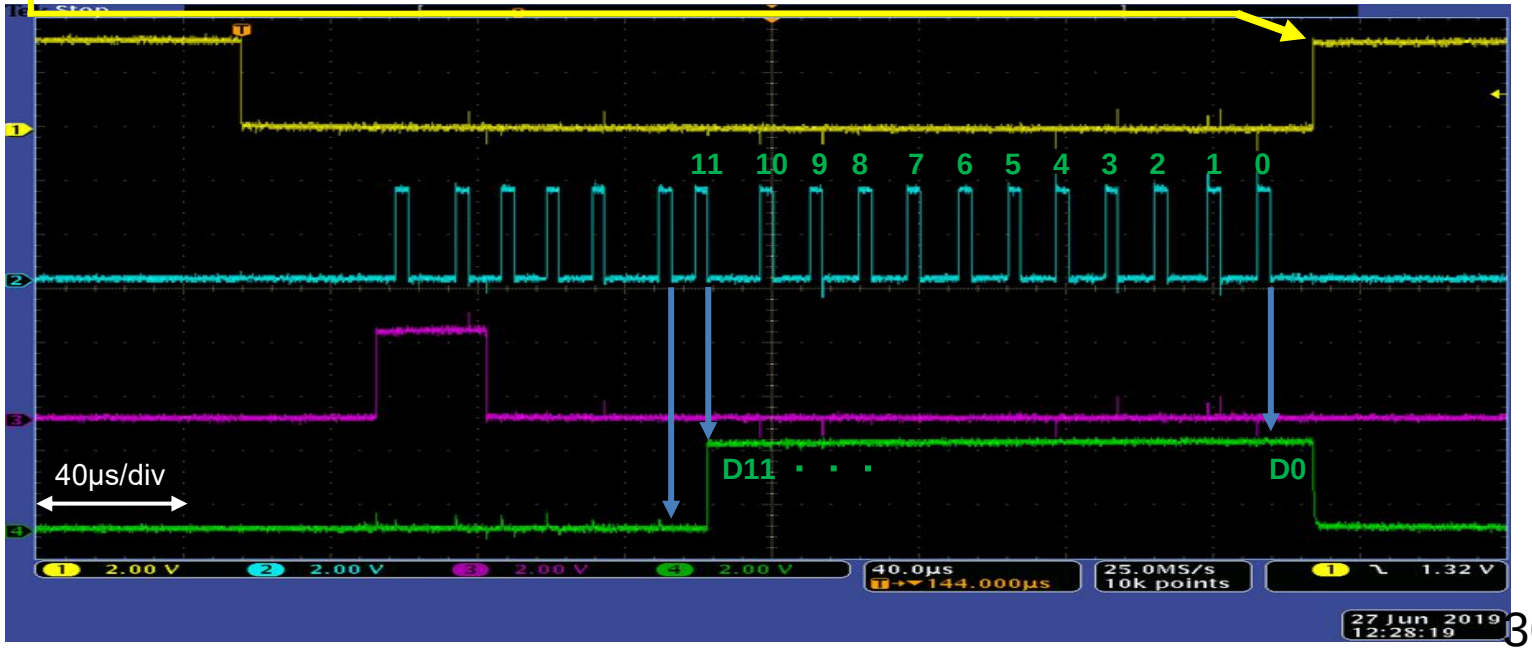
```
23
24      adcout = 0
25      for i in range(13):
26          GPIO.output(clockpin, GPIO.HIGH)
27          GPIO.output(clockpin, GPIO.LOW)
28          adcout = adcout << 1
29          if i>0 and GPIO.input(misopin)==GPIO.HIGH:
30              adcout = adcout | 0x1
31
32      GPIO.output(cspin, GPIO.HIGH)
33      return adcout
```

range(13)：下記操作を13回繰り返す。最初はヌルビットで 値は Low である。
クロックを Hight、Low する。
★クロックの立下りでADがMSBから順にデーターを出力する
Raspberry Pi はクロックの立上りでデータを1ビット毎に受信していく
adcout を1ビット左シフトする
if MISO のデータが1だったら、adcout の LSB を1にする（ adcout | 0x1 ）　↓

最後にCSピンをHighにする
ADCへのアクセスを終了する

adcout [11:0]

											ヌル	1回目
										ヌル	D11	2回目
									ヌル	D11	D10	3回目
									・	・	・	N回目
								ヌル	D11	D10	D9	12回目
							ヌル	D11	D10	D9	D8	13回目



cspin（GPIO 8）

clockpin（GPIO 11）

mosipin（GPIO 10）

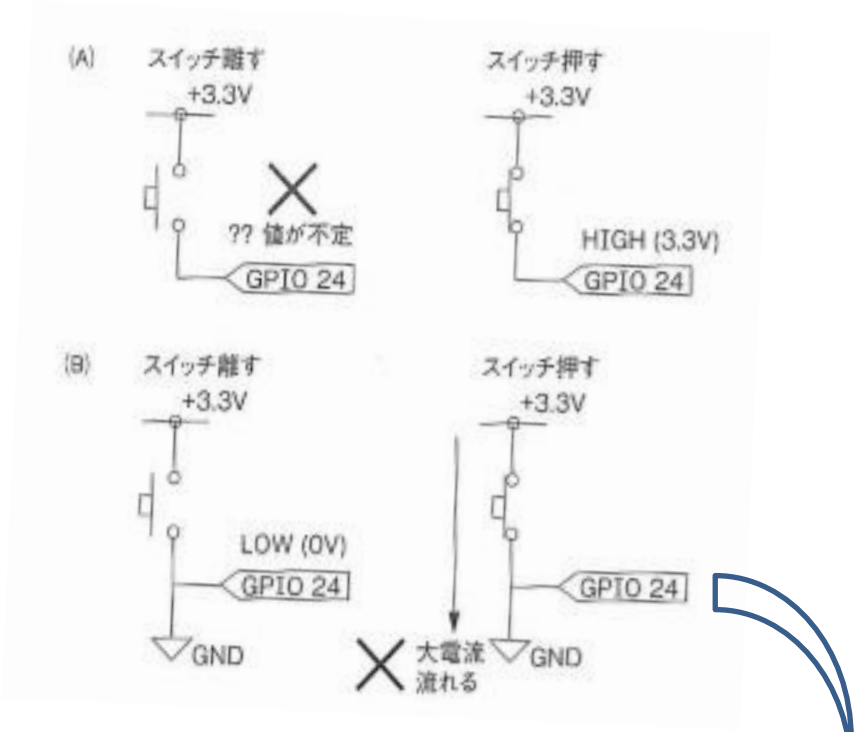
misopin（GPIO 9）

茨城県産業技術イノベーションセンター
Ibaraki Prefecture

R4年度次世代技術活用人材育成事業
技術修得コース　実習座学（IoT）
第1回
参考資料

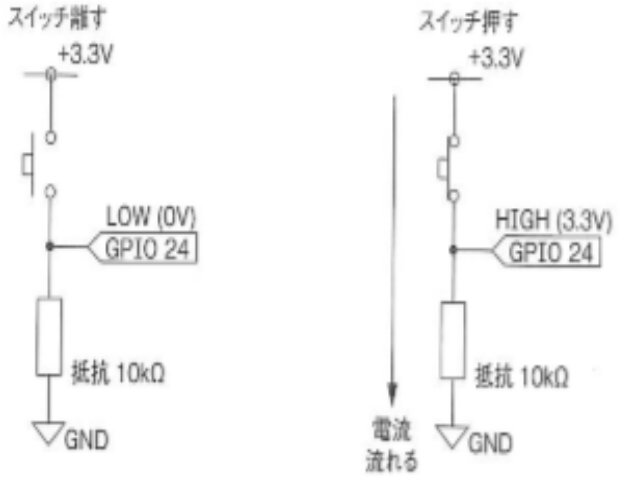
茨城県産業技術イノベーションセンター
IT・マテリアルグループ

(参考)プルダウン抵抗について

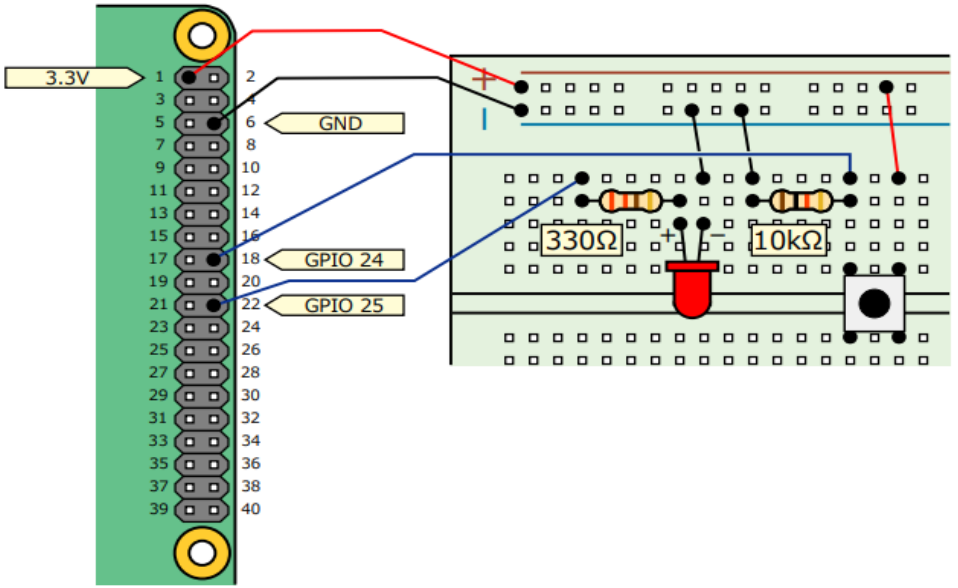


①タクトスイッチの誤った使用例

※大電流が流れるとRaspberry Piが強制的に再起動してしまい故障の原因になるので注意！



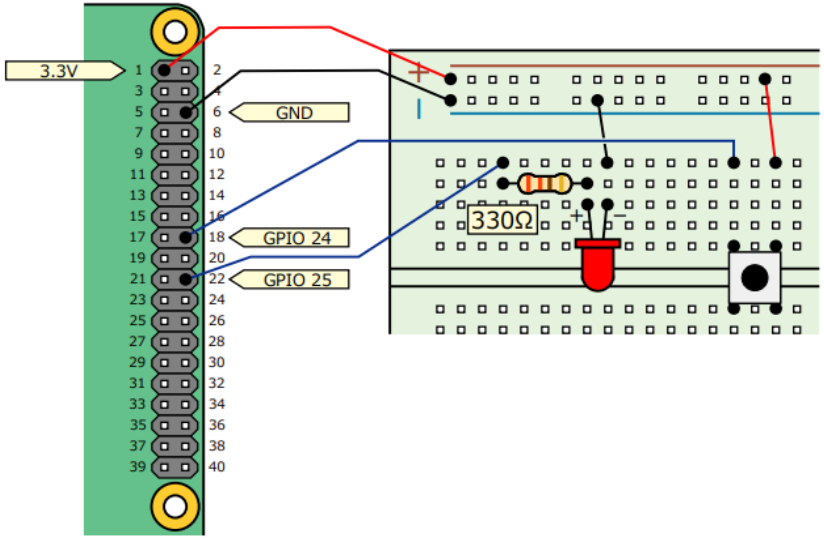
②タクトスイッチとプルダウン抵抗を用いた使用例



（参考）Raspberry Pi内部のプルダウン抵抗の利用

プルダウン抵抗やプルアップ抵抗はRaspberry Pi内部にあらかじめ用意された抵抗を有効にすることで実現できる。

「タクトスイッチでLEDの点灯」で記述したプログラムの6行目を下記のように変更する。



プルダウン抵抗を取り除いた回路

6行目：GPIO.setup(24, GPIO.IN)　　この行を下記のように変更する

▼

6行目：GPIO.setup(24, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

Raspberry Pi内部のプルダウン抵抗を有効にするための変更

プルアップ抵抗を用いたい場合
「pull_up_down=GPIO.PUD_UP」

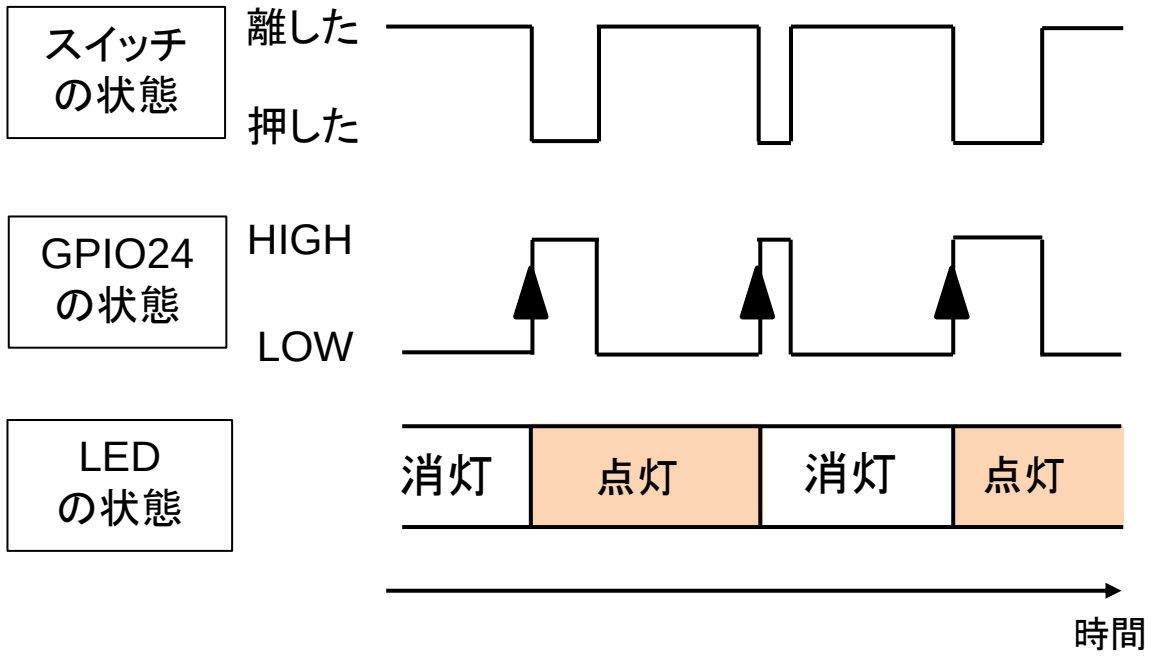
課題

①最初，LEDは消灯している状態

②タクトスイッチを1回押して離す→LEDは点灯状態を保持

③タクトスイッチをもう一度押して離す→LEDは消灯状態

④以降，②と③を繰り返す



課題プログラムの構造

import文

def 関数名:
　必要に応じて呼ばれる関数
　(関数は複数あってもよい)

初期化処理

try:
　ここにメインの処理

except KeyboardInterrupt:
　pass

GPIO.cleanup()

プログラムの構造

●defによる関数定義
def文を用いて関数を定義できる。

```
1   #Filename <def.py>
2   from time import sleep
3
4   def mile2meter(mile):
5       meter = mile*1609.344
6       return meter
7
8   try:
9       while True:
10          print('Please enter a number.')
11          num = float(input())
12          distance = mile2meter(num)
13          print('Calculation result.')
14          print(distance)
15          print('--------------------')
16          sleep(0.05)
17
18  except KeyboardInterrupt:
19      pass
20
21
```

イベント検出機能の追加

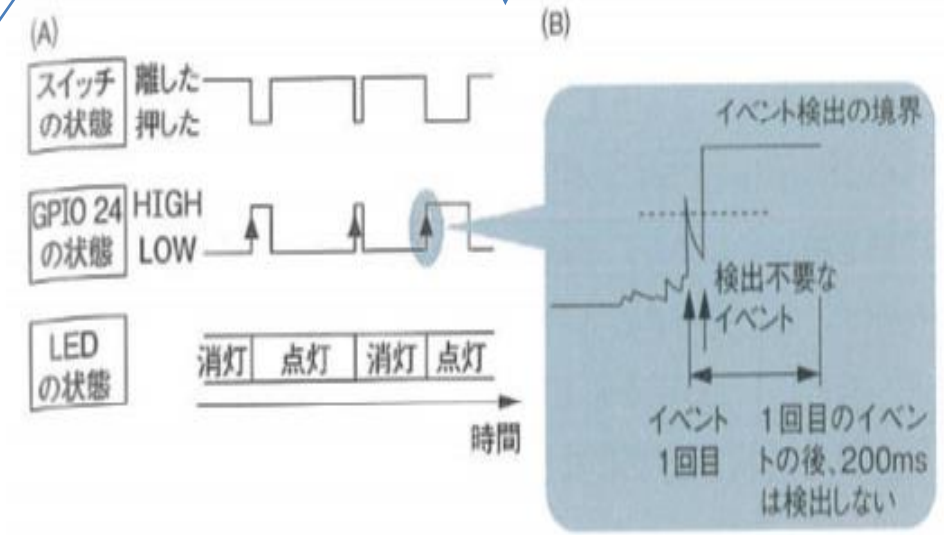
GPIO.add_event_detect(SWITCH, GPIO.RISING, callback=my_callback, bouncetime=200)

「チャンネル」
イベント検出をしたいチャンネル

「GPIO.RISING」
これはInputPin（GPIO 24）のLOWからHIGHへの上昇（RISING）を検出するポジティブエッジである。

「callback=my_callback」
エッジを検出したときに実行したい関数を指定しており，「my_callback」という名前の関数が定義されている。

ポジティブエッジ検出のタイミングで呼ばれる

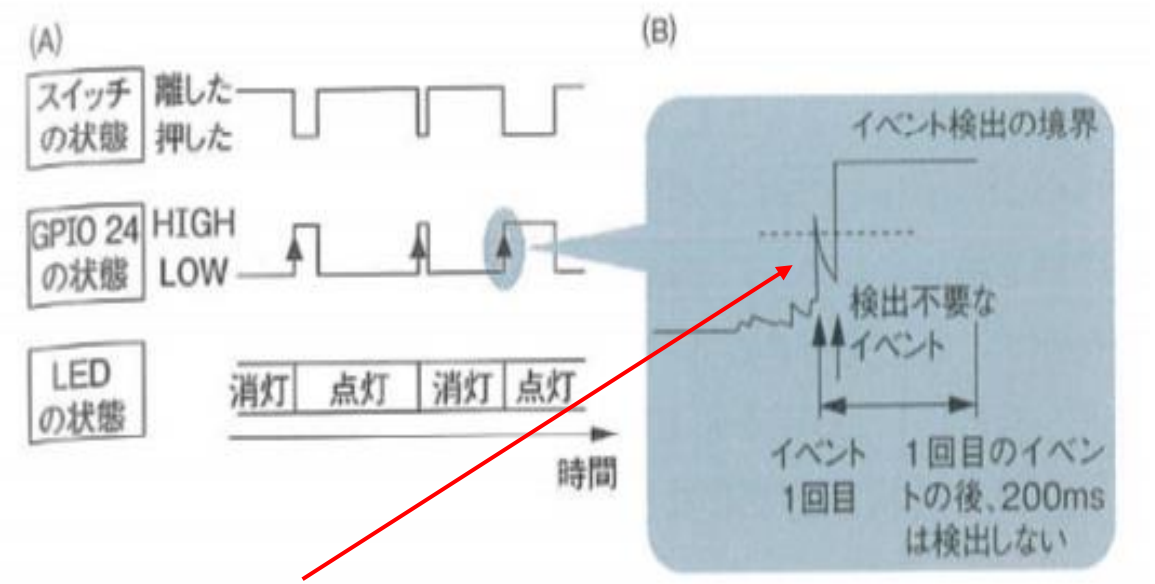


GPIO 24における電圧の変化

※人間がタクトスイッチを押して得られる電圧変化は，図のようにノイズ状に上下しながらHIGHに到達する。

不要なイベントの除外

○不要なイベントの除外
　人間がタクトスイッチを押した際の電圧の変化は，きれいにLOWからHIGHに変化するとは限らず，図のようにノイズ状に上下しながらHIGHに到達することがある。



図の場合，点線でLOWとHIGHが切り替わるとすると，ポジティブエッジが2回検出され，LEDの状態が2回変更されてしまう。

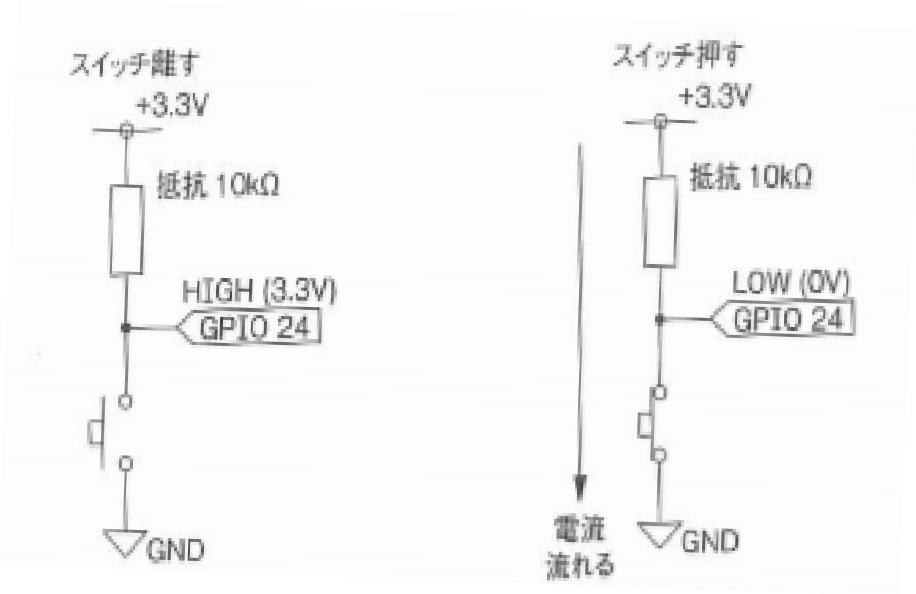
bouncetime=200とイベント文の中で設定することで一度のイベントを検出してから200 ms（0.2秒）はイベントを検出しない，という処理ができる。

課題プログラム

```
#<Filename: question.py>
#ライブラリのインポート
import RPi.GPIO as GPIO
from time import sleep

#ポート番号の定義
SWITCH = 24
LED = 25
led_value = GPIO.LOW

#GPIOの初期化
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(SWITCH, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

#swith切り替え関数
def my_callback(ch):
    global led_value
    if ch == SWITCH:
        #① ____________ = not _____________
        if led_value == GPIO.HIGH:
            #② ____________________________
            print("LED ON")
        else:
            #③ ____________________________
            print("LED OFF")

#イベントの設定
GPIO.add_event_detect(SWITCH, GPIO.RISING, callback=my_callback, bouncetime=200)

try:
    while True:
        sleep(0.01)

except KeyboardInterrupt:
    pass

GPIO.cleanup()

```

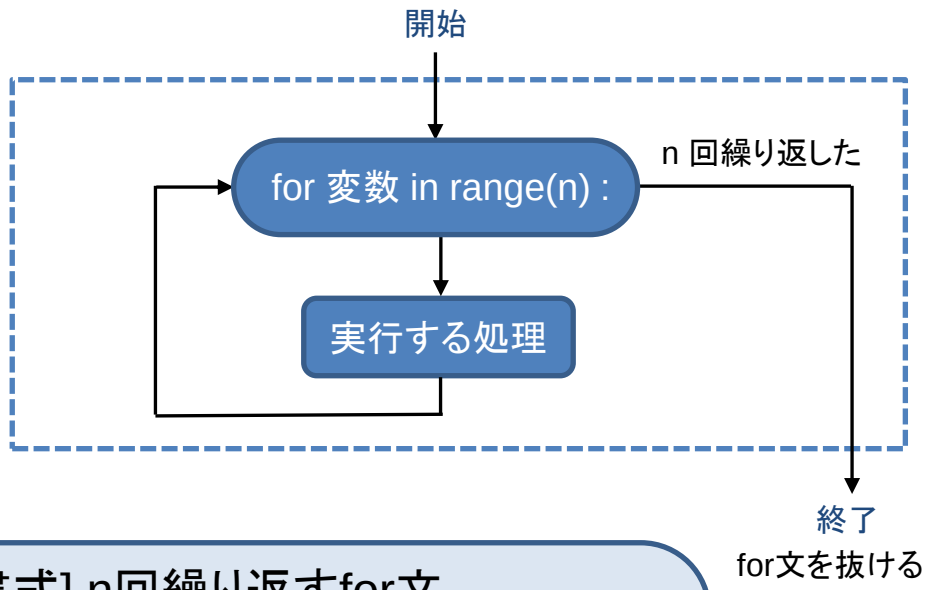
参考

プルアップ抵抗



for文

＜フローチャート＞



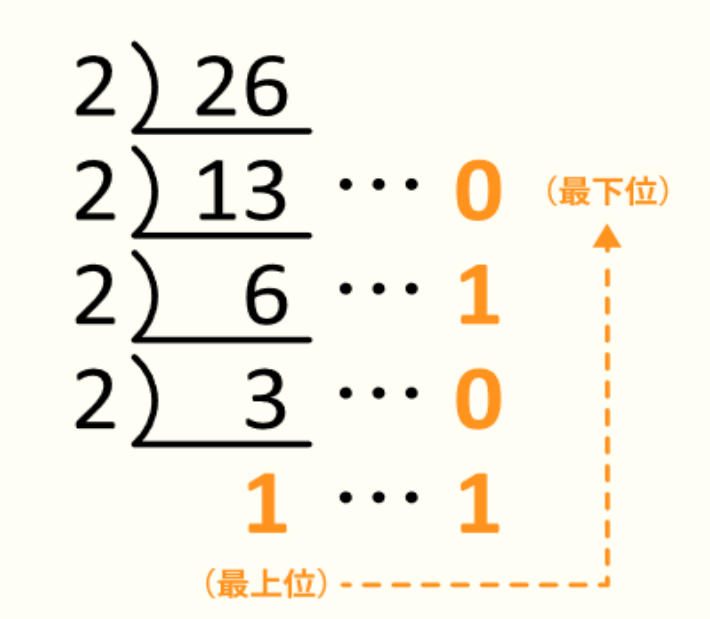
[書式] n回繰り返すfor文

```
for i in range(n):
    # インデントの開始
    ステートメント1
    ステートメント2
# インデントの終了（for文の終了）
```

2進数⇔10進数

＜10進数→2進数＞
変換する10進数を2で繰り返し割っていき，その余り（0か1になる）を下位から上位へ順に並べる。

（例1）10進数$(26)_{10}$を2進数に変換



変換結果は「11010」

＜2進数→10進数＞
2進数の各位の数字にその位の2^{n-1}を掛けて，すべての桁について足し合わせる。
※n:桁数

（例2）2進数$(11010)_2$を10進数に変換

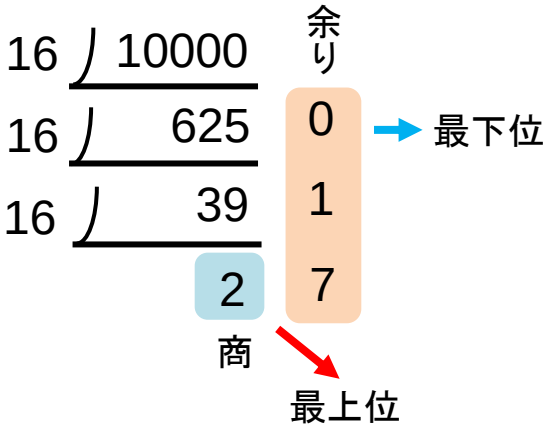
2進数	1	1	0	1	0
2^{n-1}	2^4	2^3	2^2	2^1	2^0

$$\underbrace{1\times2^4}_{5桁目}+\underbrace{1\times2^3}_{4桁目}+\underbrace{0\times2^2}_{3桁目}+\underbrace{1\times1^1}_{2桁目}+\underbrace{0\times2^0}_{1桁目}$$
$$=26$$

10進数⇔16進数

＜10進数→16進数＞
変換する10進数を商が0になるまで16で繰り返し割っていき，商と余りを求める。

（例1）10進数$(10000)_{10}$を16進数に変換



変換結果は「2710」

＜16進数→10進数＞
16進数の各位の数字にその位の16^{n-1}を掛けて，すべての桁について足し合わせる。
※n:桁数

（例2）16進数$(2710)_{16}$を10進数に変換

16進数	2	7	1	0
16^{n-1}	16^3	16^2	16^1	16^0

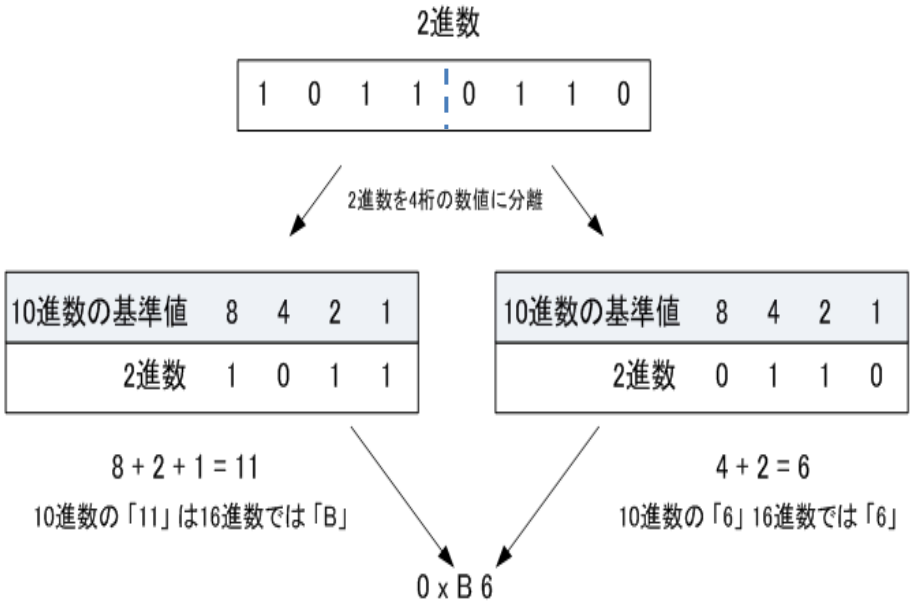
$$\underbrace{2\times16^3}_{\text{4桁目}}+\underbrace{7\times16^2}_{\text{3桁目}}+\underbrace{1\times16^1}_{\text{2桁目}}+\underbrace{0\times16^0}_{\text{1桁目}}$$

=10000

2進数⇔16進数

＜2進数→16進数＞
変換する2進数を4桁の数値に分離する。分離した数から10進数の値を求めて最後にその10進数の値を16進数に変換する。

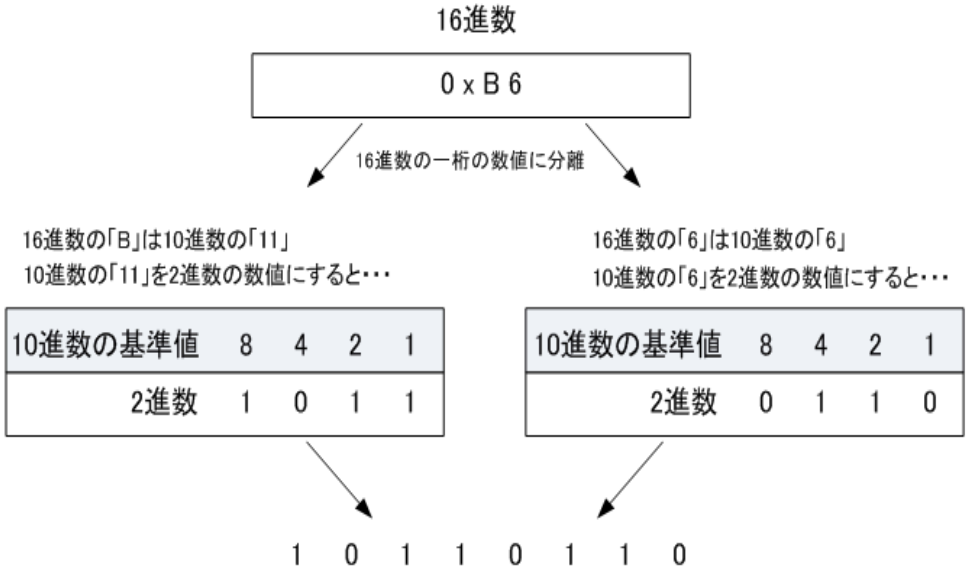
（例1）2進数$(10110110)_{10}$を16進数に変換



※変換表が暗記できていれば2進数→10進数への変換は不要。

＜16進数→2進数＞
左図と逆のことをする。16進数の1桁を2進数の4桁に変換する。そして，その4桁の数値を並べるだけ。

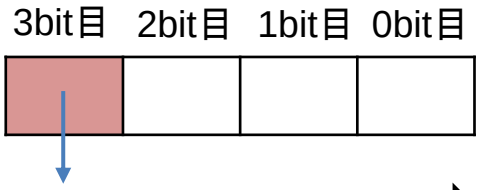
（例2）16進数0xB6を2進数に変換



※変換表が暗記できていれば16進数→10進数への変換は不要。

2進数における負の数表現①

2進数で負の数を表す場合は2の補数の考え方を使う。



◆符号ビットを取り入れる ⇨

左端を符号ビットと呼んで，
0 正の数
1 負の数
を表すという約束に変更する。

2の補数作成の方法
①すべてのビットを反転させる
②反転させた結果に1を加える
①及び②の結果を2の補数と呼ぶ。

一般に，nビットの2の補数系で扱うことのできる数Mの範囲は
$-2^{n-1} \leqq M \leqq 2^{n-1}-1$
この例では4ビットなので
$-8 \leqq M \leqq 7$となる。

符号ビットあり

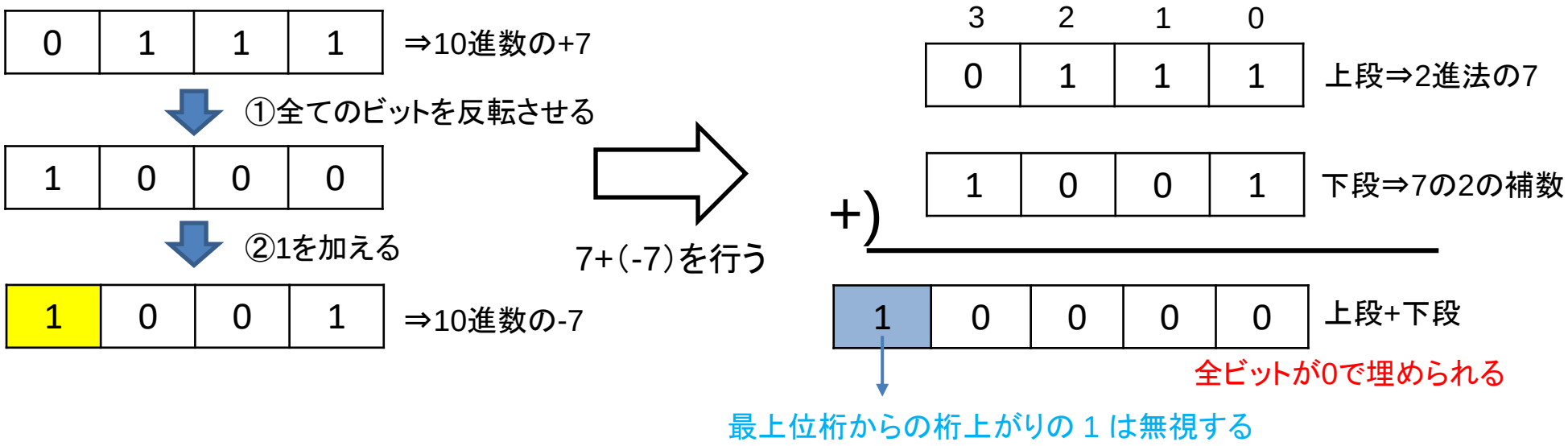
2進数	10進数
0 111	7
0 110	6
0 101	5
0 100	4
0 011	3
0 010	2
0 001	1
0 000	0
1 111	-1
1 110	-2
1 101	-3
1 100	-4
1 011	-5
1 010	-6
1 001	-7
1 000	-8

符号ビットなし

2進数	10進数
1111	15
1110	14
1101	13
1100	12
1011	11
1010	10
1001	9
1000	8
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0

2進数における負の数表現②

（例）4bitでの7+(-7)＝0（普通計算）の確かめ



（注）オーバーフロー
4bitの2の補数系（符号あり）で，$7_{(10)}+6_{(10)}$ を実行すると
$0111_{(2)}+0110_{(2)}=1101_{(2)}=-3_{(10)}$となり明らかに誤った結果が得られる。これは，4bitの2の補数系で表せる数の範囲（-8～7）を越えた数を扱ったために発生。

例題

(1) $110010_{(2)}$を10進数に変換せよ。

(2) $16_{(10)}$を16進数に変換せよ。

(3) 10進法で示された下記の減算を，5bitの2の補数を用いて加算により実行せよ。
①14-9=5
②9-14=-5

(4) 符号あり8bitで表される$11100110_{(2)}$の絶対値を10進数で求めよ。

ビット演算子①

2進数の値をビットごとに演算する。ビット演算子を使えば，ビットの合成や打消しなどが可能になる。

演算子	例	説明
&	a & b	論理積（AND。両ビットともに1のとき1）
|	a | b	論理和（OR。どちらかのビットが1ならば1）
^	a ^ b	排他的論理和（XOR。比較したビットの値が異なるとき1）
~	~a	ビット反転（NOT。ビットの1，0を反転させる。）

（例）a=0b0101，b=0b0011のとき

```
  0101
&)0011
  0001
```

AND
両方とも1の桁だけ1

```
  0101
|)0011
  0111
```

OR
どちらかが1の桁は1

```
  0101
^)0011
  0110
```

XOR
一致しない桁は1

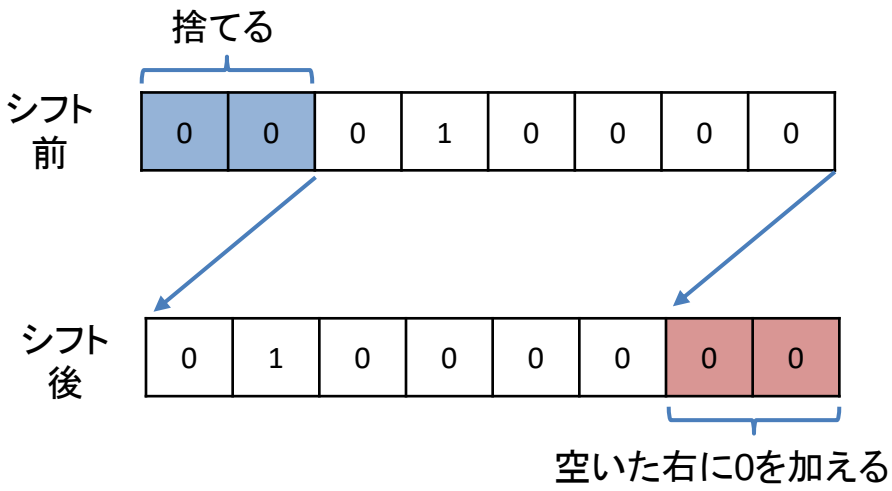
NOT
ビットの反転なので，
~aは0b1010になる

ビット演算子②

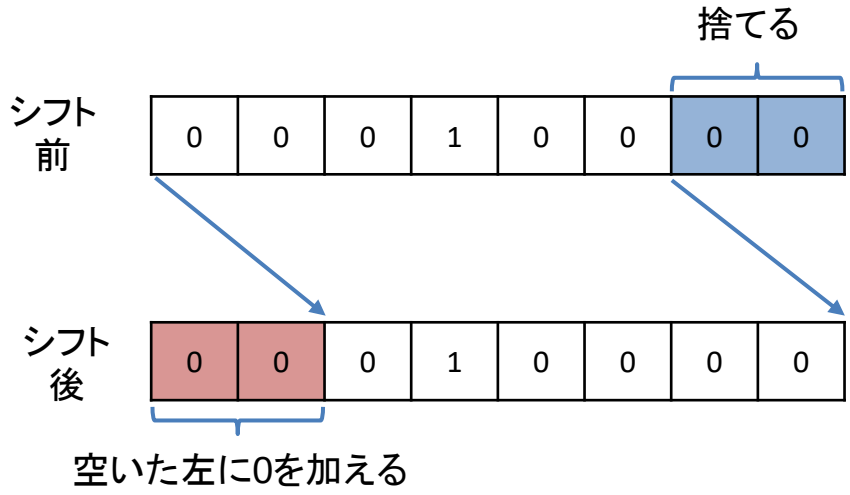
名称	演算子	例	説明
左シフト	<<	a << 1	左シフト。ビットを左にずらす。
右シフト	>>	a >> 1	右シフト。ビットを右にずらす。

（例）10進数の16を2進数（8bit）で表すと0b00010000になる。これを左右に2bitシフトさせる。（符号なしの場合）

（1）00010000<<2（左シフト）



（2）11110000>>2（右シフト）



※符号ありの場合のシフト演算は参考資料参照。

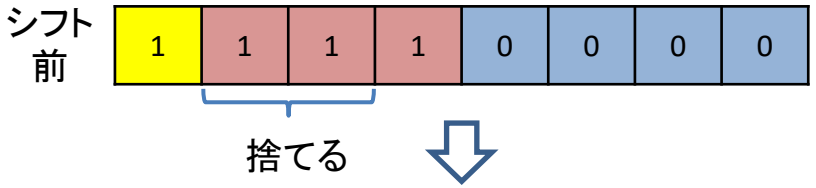
参考資料①：符号ありのシフト

名称	演算子	例	説明
左シフト	<<	a << 1	左シフト。ビットを左にずらす。
右シフト	>>	a >> 1	右シフト。ビットを右にずらす。

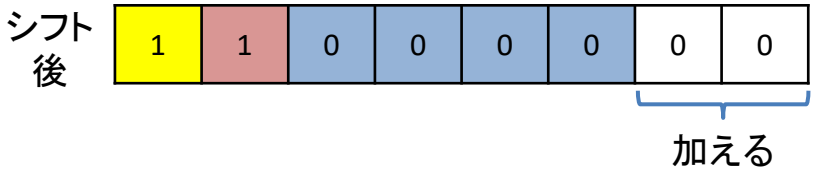
$16_{(10)}$=0b00010000
（ビットを反転して+1すると・・・）
-16=0b11110000

（例）10進数の-16を8bitの2の補数系で表すと11110000になる。これを左右に2bitシフトさせる。（符号ありの場合）

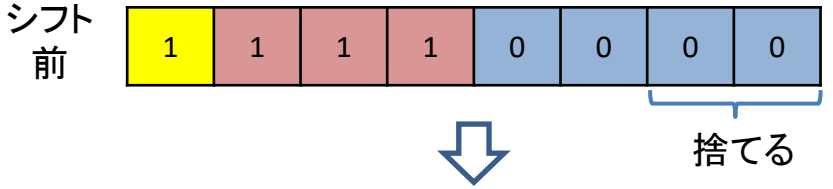
（1）11110000<<2（左シフト）



①一番左の符号ビットを1に固定して，残りの111 0000（7bit分）を左に2bitずらす。
②あふれた桁（左から1番目と2番目）を捨てて100 0000とする。
③固定していた符号ビット1をくっつける。



（2）11110000>>2（右シフト）



①一番左の符号ビットを1に固定して，残りの111 0000（7bit分）を右に2bitずらす。
②あふれた桁（右から1番目と2番目）を捨てて□□1 1100とする。
③空いた桁（□部分）には符号と同じ数字を埋める。今回は1で埋める。
④固定していた符号ビット1をくっつける。