

# EAPOL-<sup>TM</sup> Trainer

Embedded Application development Trainer

**mruby**





### 注意事項(必ず確認してください)

- 本書は著作権上の保護を受けています。本書の一部あるいは全部について、株式会社アイ・エル・シーから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

また、本書付属の CD-ROM(内部のソフトウェアなどを含みます)のご使用にあたり、以下の点に注意してください。

- CD-ROM は、特定のコンピュータでのみ使用可能です。
- CD-ROM に含まれるソフトウェアによって生成されたソースコードおよびソースコードを利用し作成した実行モジュールを、商用利用を目的に使用することはできません。
- 株式会社アイ・エル・シーは、CD-ROM 内のソフトウェアのご使用に関する Q&A 等のサポートは原則行いません。
- CD-ROM の著作権は株式会社アイ・エル・シーが有するものであり、日本国著作権法により保護されています。
- お客様は、CD-ROM を複製することはできません。
- お客様は、株式会社アイ・エル・シーに無断で、第三者に対する CD-ROM の転載、配布、公開、公衆送信、譲渡、貸与、使用許諾その他一切の行為を行うことはできません。
- お客様は CD-ROM 内のソフトウェアについてリバース・エンジニアリング、逆コンパイル、逆アンセンブルすることはできません。
- 株式会社アイ・エル・シーは、いかなる場合においても、CD-ROM の使用または使用不可能から生ずる損害に関しての責任の一切を負わないものとします。
- CD-ROM の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、株式会社アイ・エル・シーは一切その責を負いません。
- 本書の内容に基づいて作成したアプリケーション及びサンプル提供アプリケーションを、商用利用を目的に使用することはできません。

本書に登場する製品名などは、一般に、各社の登録商標、または商標です。

なお、本文中に TM, ®, ©マークなどは特に明記しておりません。

# 目次

|                      |    |
|----------------------|----|
| はじめに                 | 4  |
| 第 1 章 付属マイコンボードの準備   | 9  |
| 1.1 準備するもの           | 10 |
| 1.2 接続方法             | 13 |
| 1.3 最初のプログラムの実行      | 14 |
| 第 2 章 プログラム作成：基礎編    | 17 |
| 2.1 マイコンボードの基本       | 18 |
| 2.1.1 マイコンボードの各端子と機能 | 18 |
| 2.2 プログラム動作の説明       | 19 |
| 2.2.1 プログラムの作成       | 19 |
| 2.2.2 動作を変えてみる       | 22 |
| 2.3 mruby プログラムの基本   | 23 |
| 2.3.1 メソッド           | 23 |
| 2.3.2 変数             | 24 |
| 2.3.3 コメント           | 27 |
| 2.3.4 制御構造           | 28 |
| 2.3.5 配列             | 30 |
| 2.3.6 クラス            | 32 |
| 2.4 タッチパネルを使ったプログラム  | 37 |
| 2.4.1 LED 点灯プログラム    | 37 |
| 2.4.2 LED 輝度変更プログラム  | 41 |
| 2.5 画面表示部品クラス        | 47 |
| 2.5.1 画面に表示可能な部品     | 47 |
| 2.5.2 ボタン            | 48 |
| 2.5.3 スイッチ           | 50 |
| 2.5.4 縦スライダー         | 52 |
| 2.5.5 横スライダー         | 54 |
| 2.5.6 デジタル数値         | 56 |
| 2.5.7 テキストボックス       | 58 |
| 2.5.8 ピクチャ           | 60 |
| 2.5.9 デジタルパッド        | 62 |
| 2.6 画面描画メソッド         | 64 |

|       |                            |     |
|-------|----------------------------|-----|
| 2.6.1 | 画面描画機能                     | 64  |
| 2.6.2 | 画像表示                       | 64  |
| 2.6.3 | 矩形表示                       | 65  |
| 2.7   | 写真表示プログラム                  | 66  |
| 2.7.1 | プログラムの動作内容                 | 66  |
| 2.7.2 | 画像データの作成                   | 66  |
| 2.7.3 | プログラムの作成                   | 66  |
| 2.7.4 | プログラム内容の説明                 | 68  |
| 2.8   | タッチパネルを使ったゲーム              | 69  |
| 2.8.1 | プログラムの動作内容                 | 69  |
| 2.8.2 | 画像データの準備                   | 70  |
| 2.8.3 | プログラムの作成                   | 70  |
| 2.8.4 | プログラム内容の説明                 | 74  |
| 第 3 章 | プログラム作成：応用編                | 85  |
| 3.1   | 本章の使い方                     | 86  |
| 3.2   | ブレッドボードの使い方                | 86  |
| 3.2.1 | ブレッドボードとは                  | 86  |
| 3.3   | マイコンボードの端子と変数名の対応          | 88  |
| 3.4   | マイコンボードの端子の使い方             | 90  |
| 3.5   | LED を点滅させる                 | 95  |
| 3.6   | 電子ピアノを作る                   | 98  |
| 3.7   | 電子オルゴールを作る                 | 104 |
| 3.8   | 傾きを検知する                    | 112 |
| 3.9   | 最後に                        | 118 |
| 第 4 章 | 付録                         | 119 |
| 4.1   | プログラムが動かないときは              | 120 |
| 4.1.1 | プログラムがコンパイルできない場合          | 120 |
| 4.1.2 | 画像ファイルが変換できない場合            | 120 |
| 4.1.3 | マイコンボードでプログラムが実行できない場合     | 121 |
| 4.1.4 | マイコンボードでプログラムの実行中に動作停止する場合 | 122 |
| 4.2   | マイコンボードのシステムプログラムの書き込み方法   | 123 |
| 4.2.1 | システムプログラムについて              | 123 |
| 4.2.2 | システムプログラムの書き込み方法           | 124 |

# はじめに

---

## (1) mrubyとは

日本発の純国産プログラム言語「Ruby」をベースとして、組み込み開発の分野にも適用できるように作られた言語が「mruby」です。

Ruby は他の言語に比べて非常に開発効率が高いことが特長ですが、これまではパソコンやネットワークサーバーなどで主に使われてきました。

mruby は Ruby の特長を引き継ぎつつ、組み込み機器でも動作できるように軽量・コンパクトに作られた言語です。

これからは、組み込み開発の現場でも開発効率の高い mruby を使うことが可能になります。

EAPL-Trainer mruby（以下、本製品）は、日本発祥の組み込み開発言語 mruby を習得するための学習教材です。

## (2) 本製品の対象ユーザー

本製品では、同梱の液晶付きマイコンボードを使って組み込みプログラムの基本を学ぶことができます。何らかのプログラム言語の経験がある人を対象として想定していますが、掲載しているプログラムは簡単なものですのでプログラムの初心者の方でも自分で動かすことができると思います。

## (3) 本書の構成と使い方

### 第1章 付属マイコンボードの準備

付属マイコンボードを使うための準備の手順を記載します。

### 第2章 プログラムの作成：基礎編

付属マイコンボードを使って mruby プログラムを作成するために必要な基礎知識を記載します。

マイコンボードの基本的な使い方、mruby の基本的な文法、グラフィックを使ったプログラムの作り方、等、付属マイコンボードだけを使って実践できる内容が記載されています。

mruby を使ったマイコンプログラム作成の基礎を習得できます。

### 第3章 プログラムの作成：応用編

プログラム作成の応用として、外付けの基板と電子部品を使ったプログラムの作成について記載し

ます。

マイコンボードの入出力端子に色々な部品を接続して制御を行うことができます。

#### 第4章 付録

プログラムがうまく動かない場合の対処方法を記載します。何らかの問題が発生したときにはこの章を参照してください。

また、マイコンボードに元々書き込まれているシステムプログラムを書きかえる場合の手順を記載します。

### (4) 本書の表記法

本書で使う書体は以下の法則に従っています。

#### 等幅(Courier)

プログラムコードやプログラムコードに使用する語や出力結果などに使います。

#### 等幅の太字(**Courier Bold**)

コマンドラインで入力して実行するコマンド名やオプションなどに使います。

#### 斜体(*Italic*)

プログラム中に記述する項目の説明などに使います。

## (5) 本製品の付属ファイルとインストールについて

本書に添付されている CD-ROM には以下が含まれます。

```
CD-ROM
├ emrb_1_00_000.zip ... プログラム/データ 圧縮ファイル
└ Readme.txt      ... リリースノート
```

上記の プログラム/データ 圧縮ファイルを展開すると下記のフォルダが生成されますので、ハードディスク上の任意のフォルダにコピーしてください。

※ただし、コピー先のフォルダ名に空白文字が含まれると正しく動作しない場合がありますので、「デスクトップ」や「マイ ドキュメント」にはコピーしないでください。

コピーしたフォルダを本書では「インストールフォルダ」と呼びます。

```
emrb_1_00_000
├ binary          ... マイコンボード用システムプログラム
├ document
│   └ EAPL-Trainer_mruby_1_00_000_JP.pdf ... ユーザーズガイド(本書)
├ redistpackage ... mruby 関連ツール 補助プログラム
├ sample          ... サンプルプログラム
└ tool            ... mruby 関連ツール
```

インストールに必要なシステム要件

| 要件      | 内容  |
|---------|---|
| OS      | Windows XP Professional (SP3 以上)、<br>Windows 7 Professional (32 ビット版、64 ビット版)               |
| ハードウェア  | PC/AT 互換機   |
| CPU     | 1.0GHz 以上の Intel PentiumⅢ相当以上のプロセッサ   |
| メモリ     | ・ Windows XP : 512MB 以上<br>・ Windows 7 (32 ビット版) : 1GB 以上<br>・ Windows 7 (64 ビット版) : 2GB 以上 |
| USB ポート | 1 つ以上   |
| ハードディスク | 10MB 以上の空き容量(インストール用およびプログラム作成用)  |
| ディスプレイ  | XGA(1024×768)以上推奨、32 ビットカラー以上   |
| ディスク装置  | CD-ROM ドライブ   |



## (6) 本製品の内容のアップデートなどについて

付属ツールのバージョンアップや新しいサンプルの提供などを下記の WebPage で行う予定です。

軽量 Ruby フォーラム

<http://forum.mruby.org/>

上記フォーラム内でサポートを実施する予定です。2012 年 11 月現在では準備中です。

株式会社アイ・エル・シー 商品ページ

<http://www.ilc.co.jp/commodity/eapl-trainer-mruby/index.html>

本製品の開発元である株式会社アイ・エル・シーのページです。軽量 Ruby フォーラムが発足するまでの間は上記ページにてアップデート等を行う予定です。



# 第1章 付属マイコンボードの準備

---

本章では、付属のマイコンボードの使い方と学習に使う PC の準備の手順を説明します。  
マイコンボードを使える状態にして、最も簡単なプログラムを作って動かします。

## 1.1 準備するもの

付属のマイコンボードを使うためには以下のものを準備します。

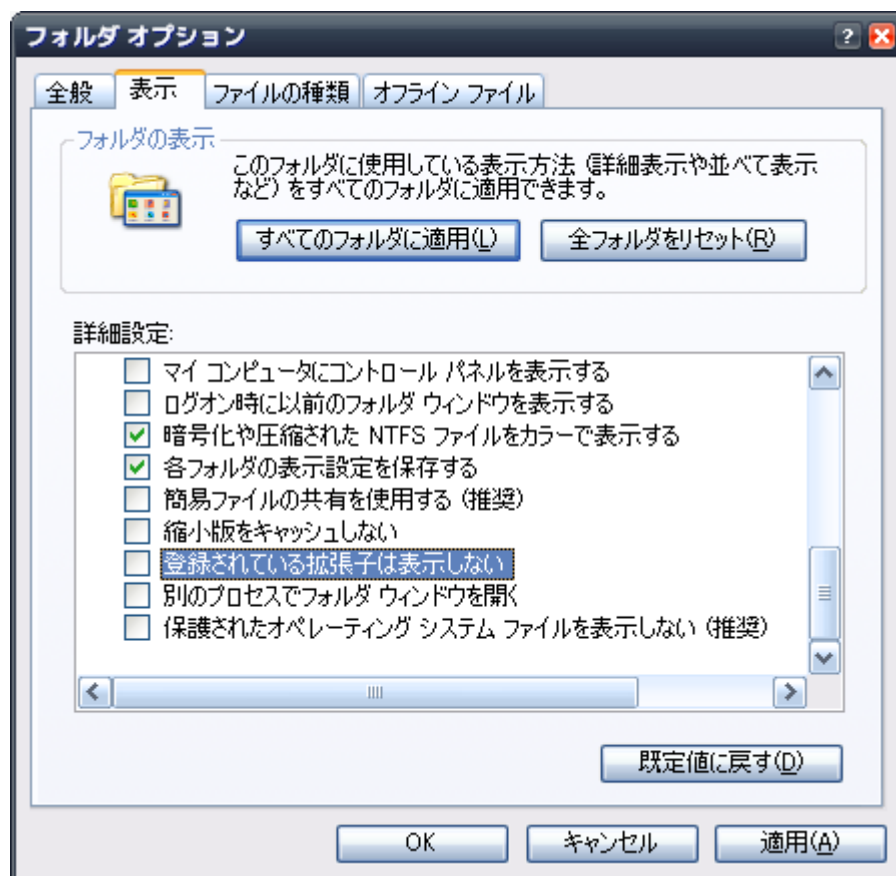
### (1) プログラム開発用PC

Windows XP/Windows 7 が動作する PC です。

本書での説明はファイルの拡張子を表示した設定に合わせていますので、あらかじめファイルの拡張子を表示する設定としておいてください。

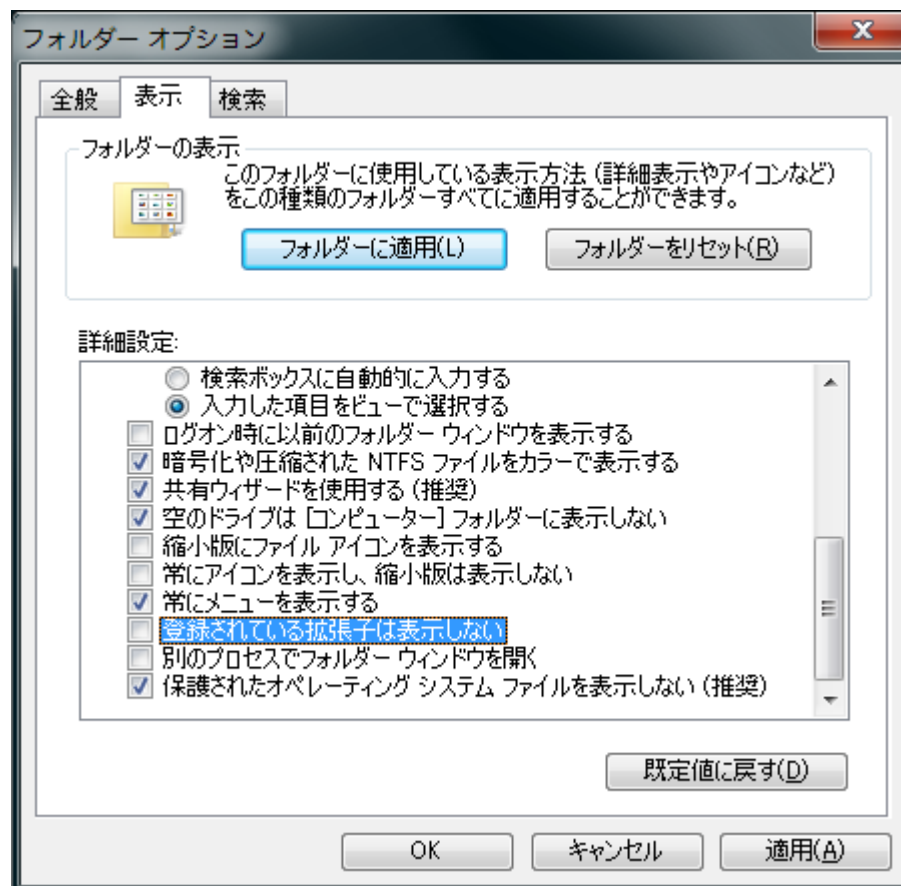
#### ■Windows XP の場合の設定方法

エクスプローラのメニュー[ツール(T)] - [フォルダ オプション(O)]を選択してください。下記のダイアログが表示されますので、下記の「登録されている拡張子は表示しない」という項目のチェックボックスを外してください。



## ■Windows 7 の場合の設定方法

エクスプローラのメニュー[ツール(T)] - [フォルダー オプション(O)]を選択してください。下記のダイアログが表示されますので、下記の「登録されている拡張子は表示しない」という項目のチェックボックスを外してください。



## (2) SDメモリカード

付属マイコンボードで動作させるプログラムを書きこみます。

対応する規格は下記となりますので、下記の表で“○”となっているものを用意してください。

|      | SD カード | miniSD カード | microSD カード |
|------|--------|------------|-------------|
| SD   | ○      | ○(*1)      | ○(*1)       |
| SDHC | ×      | ×          | ×           |
| SDXC | ×      | ×          | ×           |

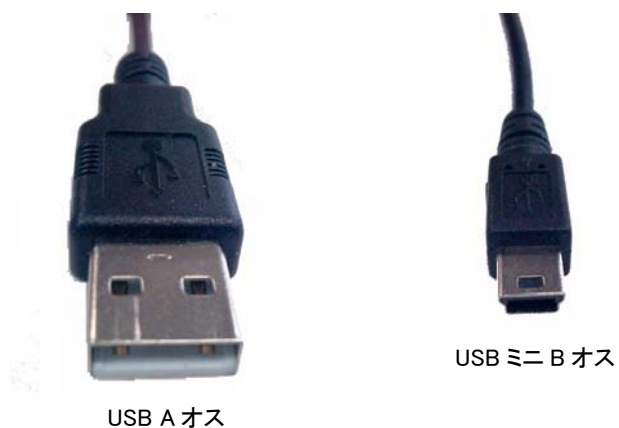
(\*1)SD カードサイズのアダプタが必要です。

SDHC/SDXC 規格のカードには対応していませんのでご注意ください。

## (3) USBケーブル

PC と接続して付属マイコンボードに電源を供給します。

USB A オス <-> USB ミニ B オス のケーブルを用意してください。

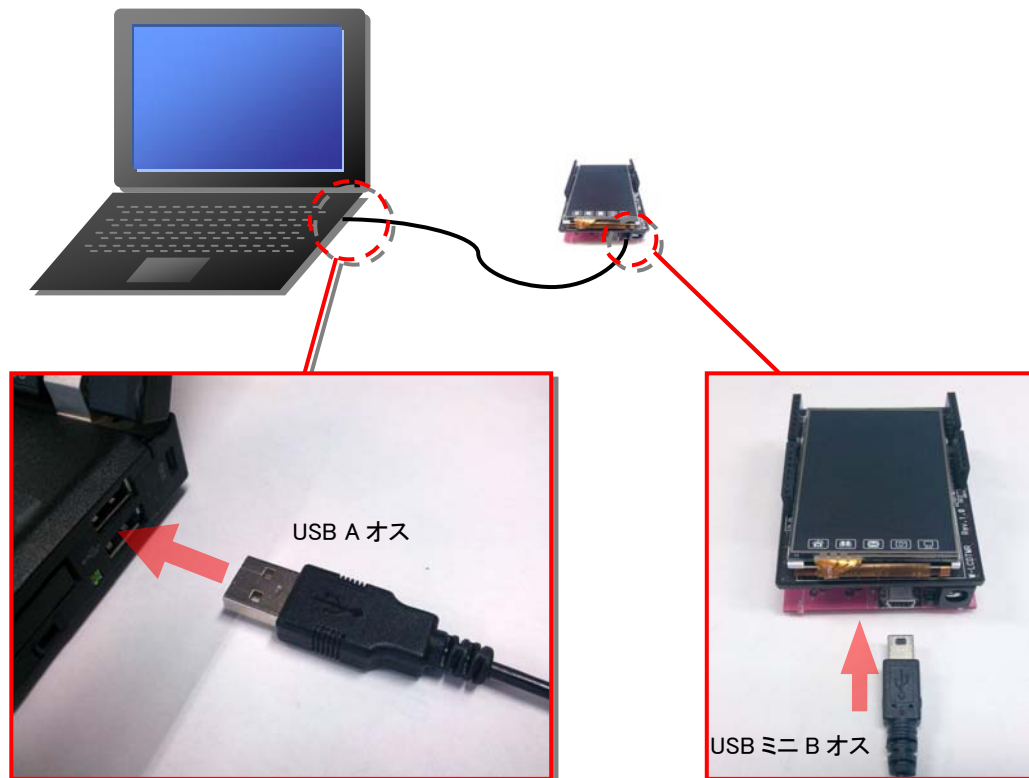


## 1.2 接続方法

付属のマイコンボードの電源としてPCのUSBポートが使えます。

1.1項で用意したUSBケーブルのUSB Aオスコネクタを動作中のPCに、USBミニBオスコネクタをマイコンボードに接続することによって、マイコンボードに電源が供給されてマイコンボードが起動します。

ケーブルの接続順序はPC側とマイコンボード側のどちらが先でも構いません。



電源を切る場合は、マイコンボードから USB ケーブルを外してください。

## 1.3 最初のプログラムの実行

### (1) プログラムの作成

マイコンボード上にある LED を点滅させるだけの簡単なプログラムを作りましょう。

Windows PC 上で、メモ帳などを使用して下記のプログラムを作成してください。

```
def setup()
  gr_pinMode($PIN_LED0, $OUTPUT)
end

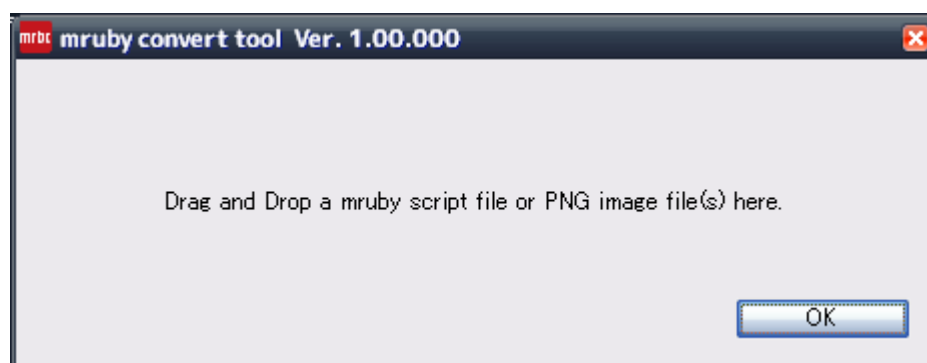
def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(100)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(100)
end
```

上記のプログラムを“**sample\_01.rb**”という名前で保存します。

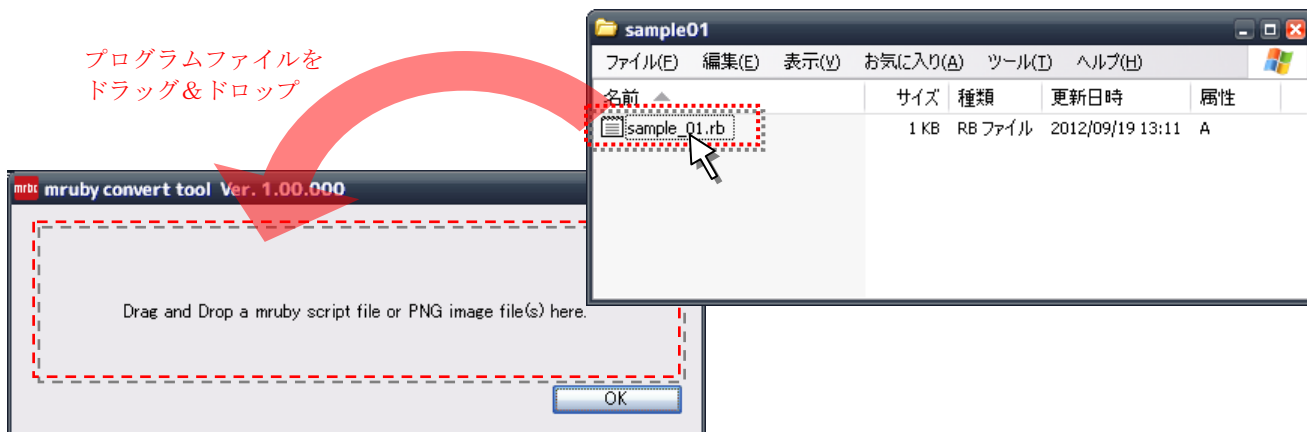
また、インストールフォルダ内の **sample¥sample01** フォルダに同じ内容のファイルが格納されています。

### (2) プログラムのコンパイル

インストールフォルダ内の **tool** フォルダに格納されている “**mrbcConvert.exe**” というプログラムを実行してください。実行するとプログラム変換ツール(下記の画面)が表示されます。



(1) で作成したプログラムファイルを上記の画面にドラッグ&ドロップしてください。



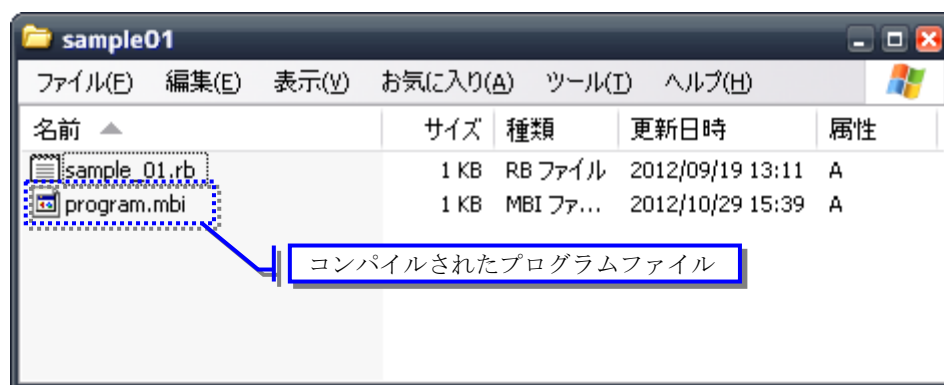


正常にプログラムがコンパイルされると、下記のようにプログラムファイルと同じフォルダに“**program.mbi**”というファイルが生成されます。

何か問題が発生するとエラーダイアログが表示される場合があります。その場合は第4章 4.1.1を参照してください。

インストールフォルダ内の **sample¥sample01** フォルダに変換結果の“**program.mbi**”ファイルも格納されています。

プログラム変換ツールが表示されている間は、何度でもプログラムファイルをドラッグ&ドロップすると“**program.mbi**”ファイルが生成されます。ただし、ファイル名は固定ですので同じプログラムをドラッグ&ドロップすると“**program.mbi**”ファイルは上書きされます。



プログラム変換ツールの[OK]ボタンを押すと、プログラム変換ツールは終了します。

### (3) プログラムの書き込みと実行

(2)で生成された“**program.mbi**”というファイルをSDカードにコピーしてください。

その際にSDカードにはフォルダを生成せずSDカードの直下にファイルをコピーしてください。また、ファイル名は“**program.mbi**”から変更しないでください。

※対応するSDカードの種別は「第1章 1.1(2)SDメモ리카ード」の項目を参照ください。

上記のプログラムをコピーしたSDカードを、マイコンボード液晶基板に搭載されているSDカードスロットに挿入して、マイコンボードの電源を入れるとプログラムを自動的に実行します。

既にマイコンボードの電源が入っている場合は、一度USBケーブルを外して電源を切断した後でSDカードを挿入して、マイコンボードの電源を入れてください。

※電源の入れ方は、「第1章 1.2 接続方法」の項目を参照ください。

プログラムが正常に実行されるとマイコンボードの下図の部分のLEDが点滅します。



これで、最初のプログラムの動作確認は終わりです。

以後、本書で説明しているプログラムのコンパイル・書き込み・実行は同様の手順で行ってください。

## 第2章 プログラム作成：基礎編

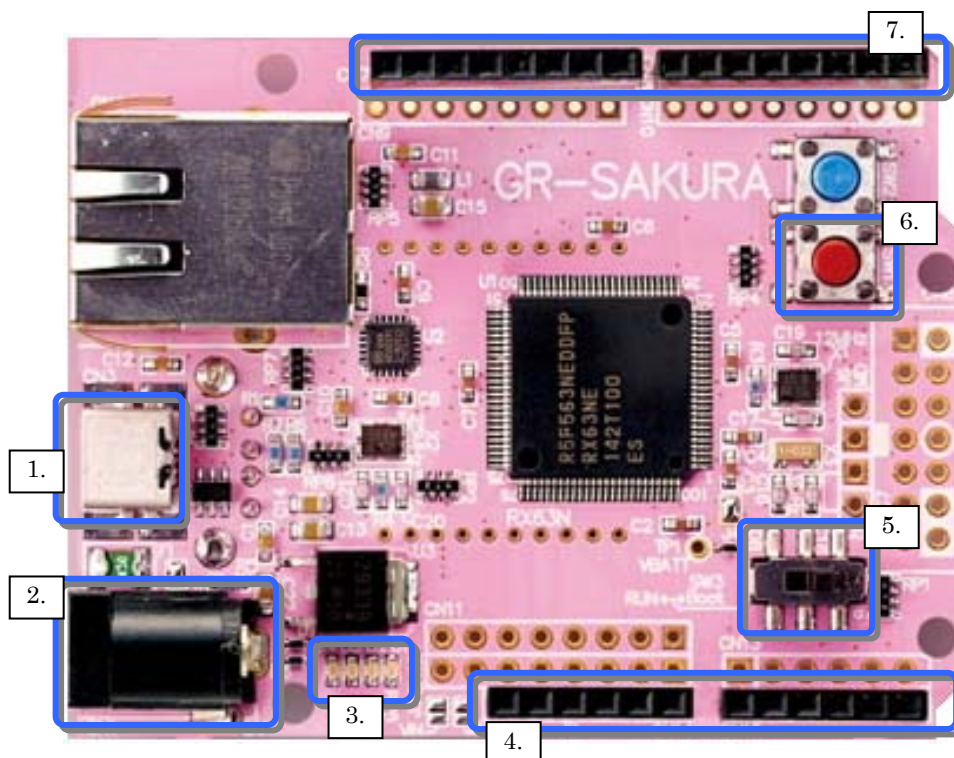
---

本章では、付属のマイコンボードで動作する簡単なプログラムの説明を行います。  
mruby を使ってマイコンボードを動かすプログラムの作り方を習得できます。

## 2.1 マイコンボードの基本

### 2.1.1 マイコンボードの各端子と機能

マイコンボードの外観を以下に示します。



| 番号 | 説明                                       |
|----|--|
| 1. | USB 接続コネクタ。PC と接続してプログラムの転送、または、電源供給を行う。 |
| 2. | 電源コネクタ。USB 接続コネクタに PC を接続している場合は、接続不要です。 |
| 3. | LED。ユーザ作成プログラムから点灯・消灯ができます。              |
| 4. | 各種入出力端子。詳細は第3章で説明します。                    |
| 5. | モード切換えスイッチ。常に"RUN"側にスイッチを入れてください。        |
| 6. | 赤いボタンはリセットボタンです。                         |
| 7. | 各種入出力端子。詳細は第3章で説明します。                    |

## 2.2 プログラム動作の説明

### 2.2.1 プログラムの作成

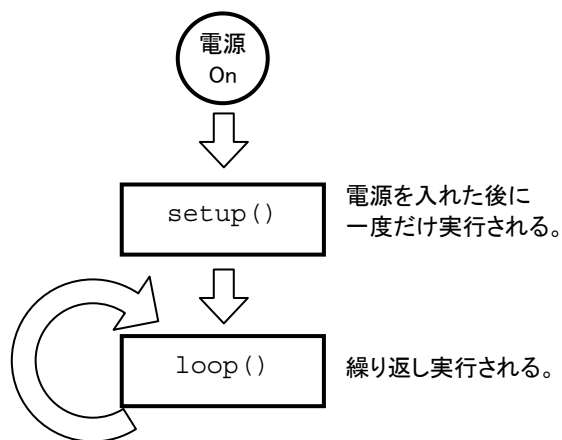
第一章で作ったプログラムについて説明します。

```
def setup()
  gr_pinMode($PIN_LED0, $OUTPUT)
end

def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(100)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(100)
end
```

setup と loop という2つのキーワードを書いています、この2つのキーワードは必ず書く必要があります。

setup は電源を入れた後に一度だけ実行される処理です。loop は繰り返し実行される処理です。setup と loop の関係は以下のようになります。

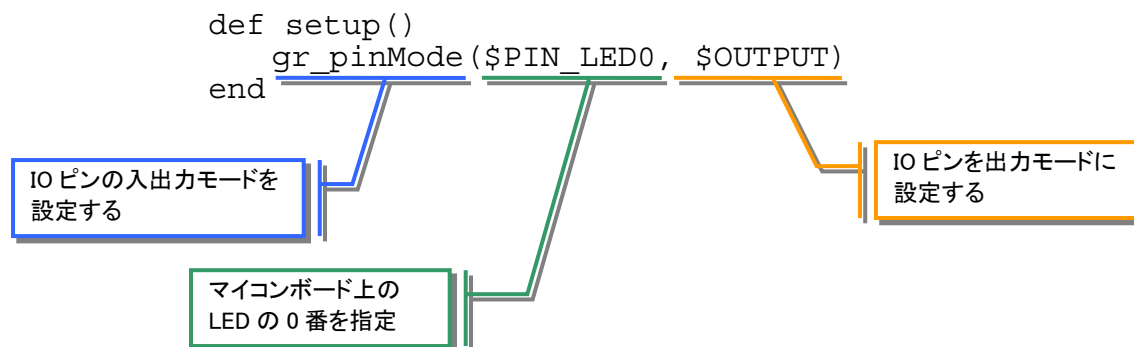


## (1) setup処理

setup 処理の内容は下記となっています。

```
def setup()
  gr_pinMode($PIN_LED0, $OUTPUT)
end
```

setup 処理は電源 On の後に 1 回だけ実行されます。上記の setup 処理の意味は、「電源 On 後にマイコンボード上の LED の 0 番を出力モードに設定する」となります(下図参照)。

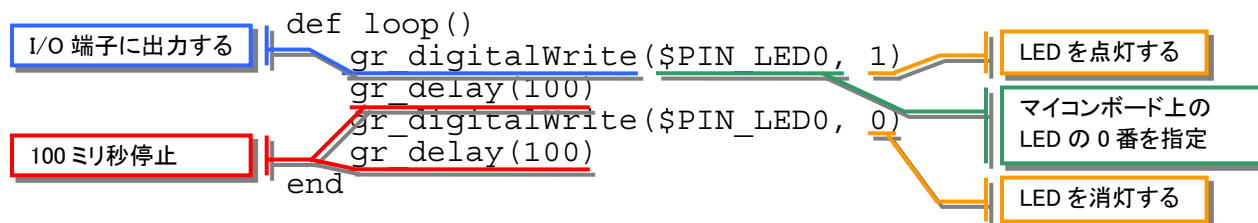


## (2) loop処理

loop 処理の内容は下記となっています。

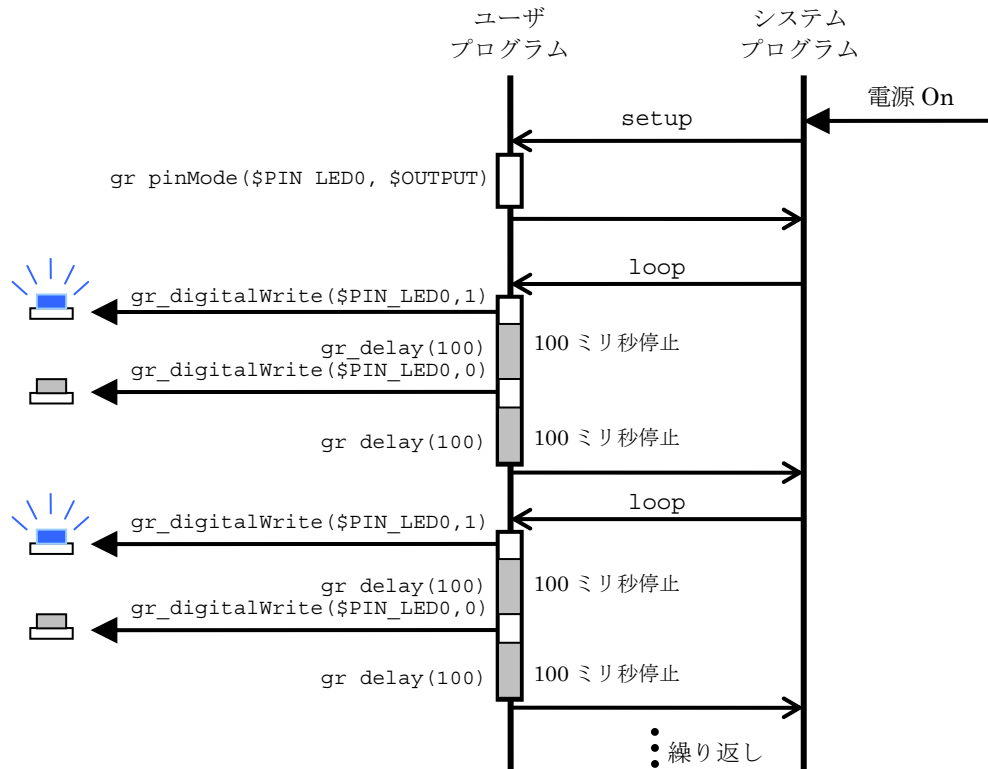
```
def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(100)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(100)
end
```

loop 処理は繰り返し実行されます。上記の loop 処理の意味は、「マイコンボード上の LED の 0 番を点灯して 100 ミリ秒停止、LED の 0 番を消灯して 100 ミリ秒停止」となり、それを繰り返し実行することによって、LED を点滅させます(下図参照)。



### (3) 全体の処理の流れ

全体の処理の流れは下記となります。



loop 処理が繰り返し実行されるため、LED が繰り返し点滅する動作となります。

## 2.2.2 動作を変えてみる

第1章で作成したプログラムを修正して下記のように変更します。

変更したプログラム：灰色の部分を変更

```
def setup()
  gr_pinMode($PIN_LED0, $OUTPUT)
end

def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(50)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(50)
end
```

インストールフォルダ内の **sample¥sample02** フォルダに **sample\_01\_modify.rb** という名前で同じ内容のファイルが格納されています。

第1章と同じ手順でプログラムを実行してください。LEDの点滅速度が速くなります。

このように、プログラムを修正することによってマイコンボードの動作を簡単に変えることができます。



## 2.3 mrubyプログラムの基本

ここまでで作成したプログラムは mruby というコンピュータ言語を使っています。本項では mruby の基本的な文法を説明します。

### 2.3.1 メソッド

mruby では、他の言語の関数やサブルーチンに相当するものをメソッドと呼びます。第1章で作成したプログラムの `setup` や `loop` もメソッドです。メソッドの書式は下記となります。

#### メソッドの書き方

```
def メソッドの名前 ( メソッドの引数 )  
  メソッドで実行する処理  
end
```

メソッドの引数が不要な場合は、引数の記述を省略することが可能です。

例) 足し算を行うメソッド

```
def add(a, b)  
  return a + b  
end
```

上記の例では、メソッド名は `add` となります。引数は `a` と `b` の2つであり、`a` と `b` を足して返すという処理となります。

メソッドの使い方は下記となります。

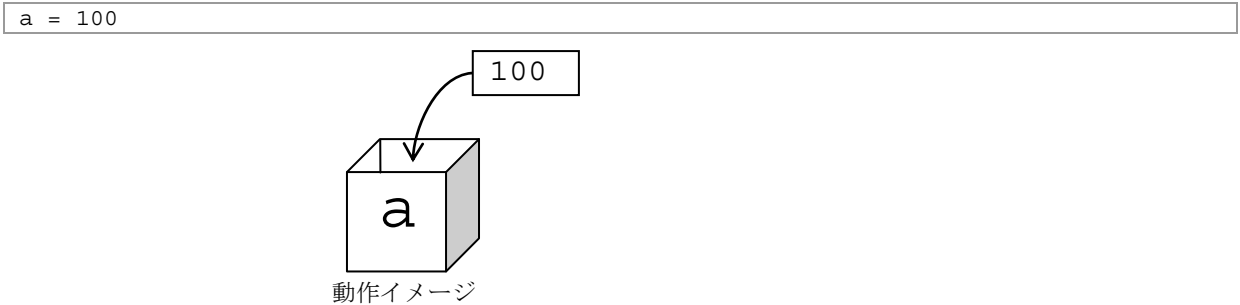
```
def add(a, b)  
  return a + b  
end  
  
c = add(10, 20)
```

上記のプログラムを実行すると、`add` という名前のメソッドに引数として `10` と `20` を渡して呼び出します。`add` メソッドの内部でそれらを足し合わせて返し、その結果が `c` に入ります。結果として、`c` の値は `30` となります。

### 2.3.2 変数

変数とは、名前の付いた箱のようなものです。その箱の中に数値などを入れることができます。

例) a という名前の変数に 100 という数値を入れるプログラム。



第1章で出てきたLEDを点滅させるプログラムを題材に変数の使い方を説明します。

変更前：

```
def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(100)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(100)
end
```

変更後：灰色の部分を変更

```
def loop()
  delayTime = 100
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(delayTime)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(delayTime)
end
```

LEDを点灯/消灯した後に100ミリ秒停止していました。この100という数値をdelayTimeという変数に置き換えています。

それによって、以下の2つの利点があります。

#### 1. 処理の修正箇所が減る

停止時間を変更する場合、delayTimeに入れている数値(100)を修正するだけで済みます。変更前だと、100という数値が2か所ありますので2か所修正する必要があります。

#### 2. 数値の意味が明確になる

delayTimeという変数に100という数値を入れて使うと、数値の意味がわかりやすくなります。100という数値がそのままプログラムに書いてあるとそれが何を意味するものかわかりにくいプログラムになります。

mruby で使うことのできる変数の中で基本的なものを以下に示します。

## (1) ローカル変数

メソッドの中で使うことができる変数です。変数の名前の先頭がアルファベットの小文字または“\_”で始まるものがローカル変数となります。

ローカル変数を使える範囲はメソッドの中だけです。メソッドの外に同じ名前のローカル変数が存在する場合、それは別の変数として扱います。

例)

```
def method1
  a = 123
end

def method2
  a = 456
end
```

method1 と method2 という 2つのメソッドの中で使っている a という変数は別の箱になります。それぞれ異なる値を持ちます。

## (2) グローバル変数

プログラムの中のどこで使っても同じ内容を扱うことができる変数です。変数の名前の先頭に“\$”を付けるとグローバル変数という扱いになります。

第1章で出てきた LED を点滅させるプログラムを題材にグローバル変数の使い方を説明します。

変更前：

```
def setup()
  gr_pinMode($PIN_LED0, $OUTPUT)
end

def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay(100)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay(100)
end
```

変更後：灰色の部分を変更

```
def setup()
  gr_pinMode($PIN_LED0, $OUTPUT)
  $delayTime = 100
end

def loop()
  gr_digitalWrite($PIN_LED0, 1)
  gr_delay($delayTime)
  gr_digitalWrite($PIN_LED0, 0)
  gr_delay($delayTime)
end
```

停止時間を意味する変数`$delayTime`を追加しました。変数の名前の先頭に"\$"という文字が付けられているので、グローバル変数となります。

`setup` メソッドで`$delayTime` に 100 という数値を入れて、`loop` メソッドで同じ`$delayTime` という変数に入っている 100 という数値を使うことができます。

上記のプログラムで使っている、`$PIN_LED0` や`$OUTPUT`などもグローバル変数です。マイコンボードを使うために必要な数値はあらかじめグローバル変数で設定してあります。

マイコンボード用に設定済みのグローバル変数の一覧は 第3章3.3 に記載します。

### 2.3.3 コメント

コメントとは、プログラムを読みやすくするために書き込む注釈文です。プログラムの実行内容には影響しませんが、適切なコメントを書くことによってプログラムが読みやすくなります。

#### コメントの書き方

# コメント記述

プログラム中に # 文字を記述すると、そこから1行の行末までが全てコメント扱いとなります。  
# は行の頭に書いてもよいですし、行の途中に書いてもよいです。

例) 下記の灰色の部分がコメントとなります。

```
# 初期化処理
def setup()
  # LED0 を出力モードに設定する
  gr_pinMode($PIN_LED0, $OUTPUT)
end

# 繰り返し処理
def loop()
  gr_digitalWrite($PIN_LED0, 1)  # LED0 を点灯する
  gr_delay(100)                 # 100 ミリ秒停止
  gr_digitalWrite($PIN_LED0, 0)  # LED0 を消灯する
  gr_delay(100)                 # 100 ミリ秒停止
end
```

## 2.3.4 制御構造

### (1) 条件分岐

条件によって処理を分岐する場合は if 文を使います。if 文の書式は下記となります。

#### if 文の書き方

```
if 条件 then
    条件が成り立つ場合に実行する処理
end
```

または

#### if 文の書き方

```
if 条件 then
    条件が成り立つ場合に実行する処理
else
    条件が成り立たない場合に実行する処理
end
```

条件が数値の大小関係の比較の場合、以下のような記号を使います。

a と b が等しい     : a == b  
a は b より大きい   : a > b  
a は b 以上         : a >= b

例)

```
if a >= 10 then
    result = 1
else
    result = 0
end
```

a が 10 以上の場合、result の値は 1 となります。そうでない場合、result の値は 0 となります。

## (2) 繰り返し

mruby では繰り返し処理を行うための方法がいくつかあります。ここでは、基本的な while 文の使い方を説明します。while 文の書式は下記となります。

### while 文の書き方

```
while 繰り返しを続ける条件  
  繰り返し行う処理  
end
```

例)

```
i = 1  
while i <= 10  
  a = a * 2  
  i = i + 1  
end
```

i の値が 1 から始まり、i が 10 以下の間は  $a = a * 2$  という処理を繰り返します。

また、繰り返しの回数が最初から決まっている場合は times というメソッドを使うとも可能です。times メソッドの書式は下記となります。

### times メソッドの書き方

```
繰り返し回数.times {  
  繰り返し行う処理  
}
```

例)

```
10.times {  
  a = a * 2  
}
```

$a = a * 2$  という計算を 10 回繰り返します。

### 2.3.5 配列

複数のデータをひとまとめにして扱うための機能として配列があります。

配列を作成するときの書式は下記となります。

#### 配列の書き方

*配列の名前* = [ データ, データ, ... ]

例) 10, 20, 30 の 3 つの値を持つ配列の作成

```
testArray = [ 10, 20, 30 ]
```

配列に格納したデータには、先頭からの位置を示すインデックス番号が付きます。インデックス番号は 0 から始まります。

配列に格納したデータの読み書きを行うときの書式は下記となります。

#### 配列の読み書き

*配列の名前* [ インデックス番号 ]

例) 配列の 2 番目を読み出す

```
testArray = [ 10, 20, 30 ]  
value = testArray[1]
```

変数 `value` の値は 20 になります。

例) 配列の 1 番目に書き込む

```
testArray = [ 10, 20, 30 ]  
testArray[0] = 15
```

配列 `testArray` の内容は [15, 20, 30] になります。



配列の持つデータの個数を取得するときの書式は下記となります。

#### 配列の大きさ取得

*配列の名前.size*

例) 配列の大きさを取得する

```
testArray = [ 10, 20, 30 ]
value = testArray.size
```

変数 `value` の値は 3 になります。

配列に対して繰り返し処理を行うときの書式は下記となります。

#### 配列に対する繰り返し

*配列の名前.each { | 繰り返し変数の名前 |  
配列の各データに対して行う処理  
}*

例) 配列の各データの合計を計算する

```
testArray = [ 10, 20, 30 ]
sum = 0
testArray.each { |val|
  sum = sum + val
}
```

変数 `sum` の値は 60 になります。

`for` 文と組み合わせて配列の繰り返し処理を行うこともできます。その場合の書式は下記となります。

#### for 文による繰り返し

*for 繰り返し変数の名前 in 配列の名前 do  
配列の各データに対して行う処理  
end*

例) 配列の各データの合計を計算する

```
testArray = [ 10, 20, 30 ]
sum = 0
for val in testArray do
  sum = sum + val
end
```

変数 `sum` の値は 60 になります。

## 2.3.6 クラス

クラスとは、関連するデータと処理を一か所にまとめて記述するための機能です。

クラスを使うことによって以下のような利点が生れます。

- ・外部に対して処理内容を隠すことができる
- ・プログラムの書き方がシンプルになる

### (1) クラスとインスタンス

クラスにはデータと処理を記述しますが、それらを使用するためにはインスタンスというものを作成する必要があります。

クラスはデータと処理の設計図で、インスタンスはその設計図を基に作られる実体のようなものです。

#### クラスの記述

```
class クラスの名前  
end
```

例：

```
class SampleClass  
end
```

#### インスタンスの作り方

```
インスタンスの名前 = クラスの名前.new
```

例：

```
sampleInstance = SampleClass.new
```

クラスとは設計図のようなもので、インスタンスはそこから作られる実体ですので、1つのクラスから複数のインスタンスを作成することができます。従って、下記のようなことも可能です。

```
sampleInstance1 = SampleClass.new  
sampleInstance2 = SampleClass.new  
sampleInstance3 = SampleClass.new
```

### (2) インスタンス変数

インスタンス変数とは、インスタンスの中で扱うことができる変数です。

インスタンス毎に個別のインスタンス変数を持つため、同じ名前のインスタンス変数であってもイ

インスタンス毎に異なる値を格納することができます。

変数の名前の先頭に"@"を付けるとインスタンス変数という扱いになります。

### (3) initializeメソッド

クラスの中にメソッドを記述することができますが、その中で `initialize` という名前のメソッドは特別な意味を持ちます。

`initialize` という名前のメソッドを作成しておく、インスタンスを生成したときに自動的に呼ばれます。`initialize` メソッドの中に初期化を行う処理を記述することによって、インスタンスを初期化することができます。

#### initialize メソッドの書き方

```
class クラスの名前
  def initialize (初期化引数)
    初期化処理
  end
end
```

例：

```
class SampleClass
  def initialize (a, b)
    @a = a
    @b = b
  end
end
```

@a と @b という記述が前述のインスタンス変数となります。`initialize` メソッドの初期化引数はインスタンスを生成するときに指定する引数となります。

上記の例では初期化時の引数として a と b という 2つの引数を持ちます。それらの引数で与えられた値をそれぞれインスタンス変数の@a と@b に格納しています。

a と@a、b と@b は別の変数です。

```
sampleInstance1 = SampleClass.new(10,20)
sampleInstance2 = SampleClass.new(30,40)
```

と記述した場合は、`sampleInstance1` という名前のインスタンスのインスタンス変数@a に 10 が格納され、@b に 20 が格納されます。`sampleInstance2` という名前のインスタンスのインスタンス変数@a に 30 が格納され、@b に 40 が格納されます。

### (4) インスタンスメソッド

`initialize` 以外の名前で作成したメソッドはインスタンスメソッドとなります。インスタンスメソッドは他の処理からインスタンスに対して呼び出すことができるメソッドとなります。

**インスタンスメソッドの書き方**

```
class クラスの名前
  def インスタンスメソッドの名前(引数)
    インスタンスメソッドの処理
  end
end
```

例：

```
class SampleClass
  def initialize (a, b)
    @a = a
    @b = b
  end

  def getSum          # 引数が不要な場合は省略可能です
    return @a + @b
  end

  def changeValue(a, b)
    @a = a
    @b = b
  end
end
```

上記のクラスに対して下記のプログラムを書いた場合、

```
sampleInstance1 = SampleClass.new(10,20)
sum = sampleInstance1.getSum
```

sum には  $10+20 = 30$  が格納されます。

```
sampleInstance1.changeValue(30, 40)
sum = sampleInstance1.getSum
```

sum には  $30+40 = 70$  が格納されます。

**(5) クラスの作成例**

ここでは、クラスの例として図形の円を扱うクラスを作ります。

円を扱うクラスは半径を保持し、直径と面積を計算できるものとします。

クラスの記述例

```
class Circle
  # 生成初期化処理
  def initialize( r )
    @radius = r
  end

  # 直径の取得
```

```
def getDiameter
  return @radius * 2
end

# 面積の取得
def getArea
  return 3.14 * @radius * @radius
end

# 半径の設定
def setRadius( newRadius )
  @radius = newRadius
end
end
```

円を扱うクラスは以下のインスタンス変数とインスタンスメソッドを持ちます。

インスタンス変数

@radius : 円の半径を格納します

インスタンスメソッド

getDiameter : 円の直径を取得します

getArea : 円の面積を取得します

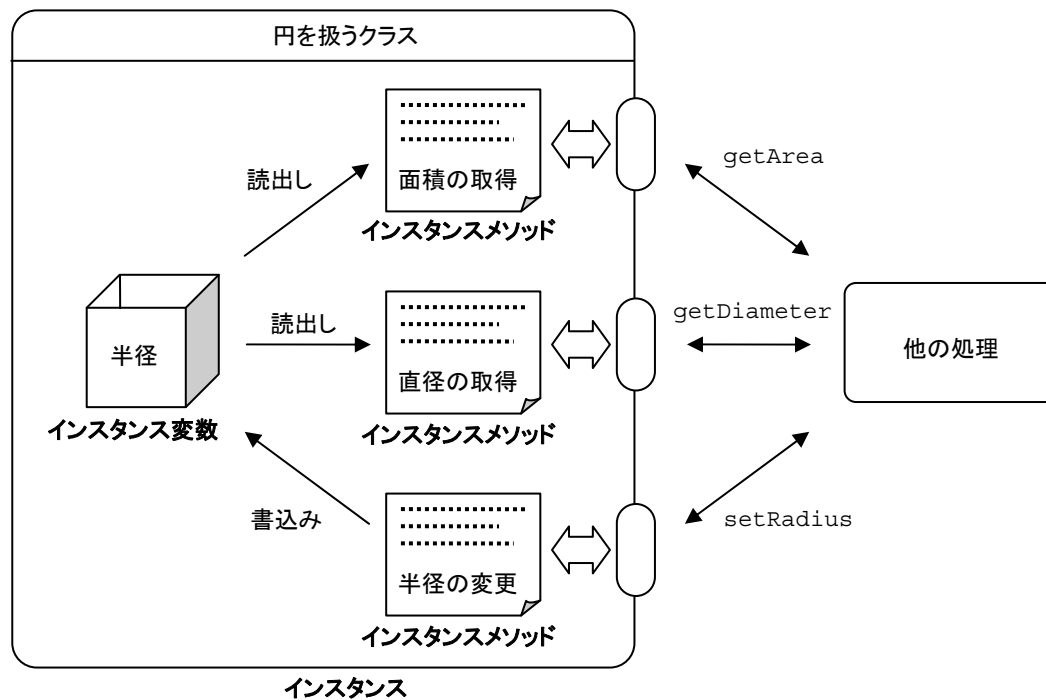
setRadius : 円の半径を設定します

円を扱うクラスはこれらのデータと処理を内部で管理していて、他の処理からはインスタンスメソッドの呼び出しを行うことによって各種の処理を行うことができます。

これは、他の処理からはインスタンスメソッドの入口しか見えないということを意味しており、クラスの内部のインスタンス変数の構成やインスタンスメソッドの処理内容を変更しても他の処理に影響を与えないという利点があります。

円を扱うクラス概念図を以下に示します。インスタンス変数やインスタンスメソッドの処理内容がクラスの中に隠されていることがわかります。

隠されているということは、他の処理からはクラスの中身を気にしなくてもよいということであり、クラスの中身を変更しても他の処理の処理内容の変更は不要ということを意味します。



### 円を扱うクラスの使用例

```
circle1 = Circle.new(10)
diameter = circle1.getDiameter
area = circle1.getArea
```

diameter の値は  $10 * 2 = 20$  となり、area の値は  $3.14 * 10 * 10 = 314$  となります。

```
circle1.setRadius(20)
diameter = circle1.getDiameter
area = circle1.getArea
```

diameter の値は  $20 * 2 = 40$  となり、area の値は  $3.14 * 20 * 20 = 1256$  となります。

## 2.4 タッチパネルを使ったプログラム

これまでは使用しなかったタッチパネル液晶画面を使ったプログラムの作り方を説明します。

### 2.4.1 LED点灯プログラム

#### (1) プログラムの動作内容

画面上にボタンを表示して、ボタンをタッチすると LED が点灯するプログラムを作ります。

#### (2) プログラムの作成

下記のプログラムを作成して実行してください。

```
def setup()
  # ボタンを作成
  $button1 = GNButton.new(10, 10, "onTouch1")

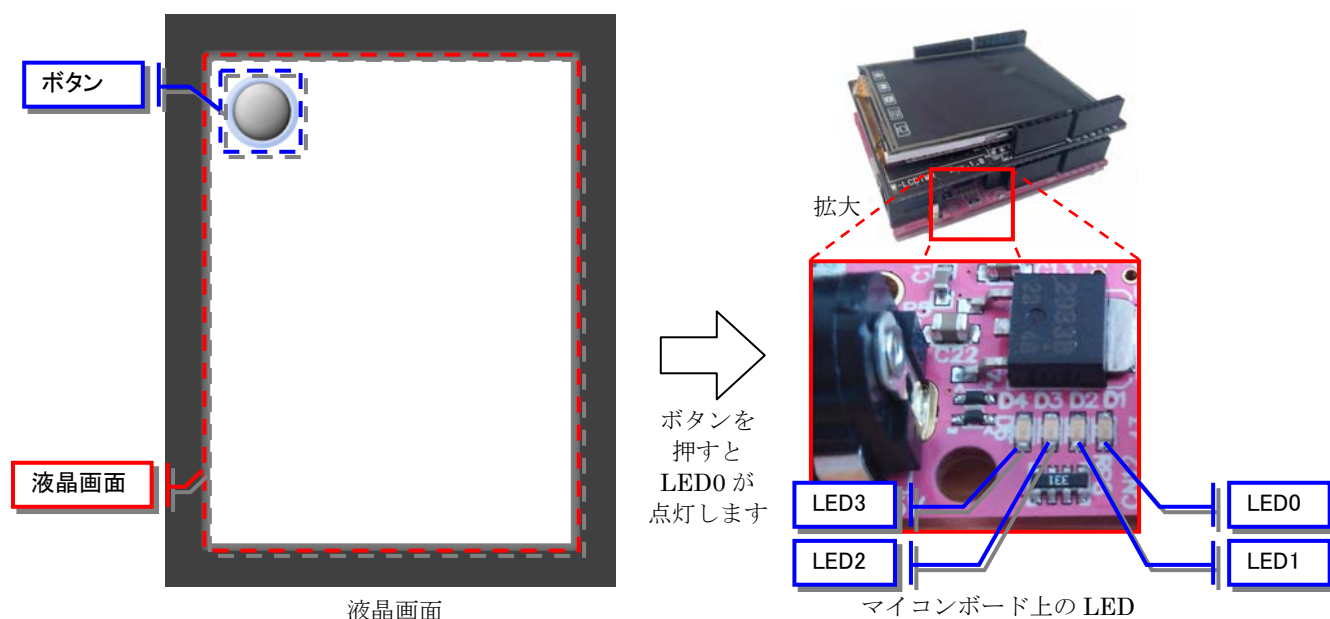
  gr_pinMode($PIN_LED0, $OUTPUT)
end

def onTouch1(param1, param2)
  # LED を On/Off
  gr_digitalWrite($PIN_LED0, param1)
end

def loop()
end
```

上記のプログラムを“**sample\_03.rb**”という名前で保存します。インストールフォルダ内の **sample¥sample03** フォルダに同じ内容のファイルが格納されています。

上記のプログラムをコンパイルして SD カードに書き込み、実行すると下記のような画面が液晶に表示されます。



液晶画面に表示されたボタンを押すとマイコンボード上の LED が点灯して、離すと LED0 が消灯します。

### (3) 処理内容の説明

処理内容の概要は下記となります。

```
def setup()
  # ボタンを作成
  $button1 = GNButton.new(10, 10, "onTouch1")
  gr_pinMode($PIN_LED0, $OUTPUT)
end

def onTouch1(param1, param2)
  # LED を On/Off
  gr_digitalWrite($PIN_LED0, param1)
end

def loop()
end
```

液晶画面に配置する部品の設定。

ボタンを押したときに実行する処理。

loop 処理は不要なので、メソッドの中身を空にしておきます。  
loop メソッドそのものを削除すると正常に動作しません。

setup メソッドの中で、\$button1 という名前のボタンを作成しています。

ボタンの作成には位置とメソッド名を設定します。この例では、画面上の(10,10)の位置にボタンを配置して、ボタンを押したときには onTouch1 という名前のメソッドを実行する、という設定を行っています。

```
def setup()
  # ボタンを作成
  $button1 = GNButton.new(10, 10, "onTouch1")
  gr_pinMode($PIN_LED0, $OUTPUT)
end
```

ボタンを押したときに onTouch1 という名前のメソッドを実行する。

(10, 10)の位置にボタンを作成する。

作成するボタンの名前

ボタンを押した場合に実行するメソッドには param1 と param2 という名前の引数があります。ボタンを押した場合には param1 の値は 1 となり、押したボタンを離した場合には param1 の値は 0 となります。

param1 の値を使って、押した場合と離した場合の処理を分けることが可能です。

この例では、param1 の値をそのまま LED 用のピンの出力値に使っているため、ボタンを押した場合は 1 を出力して LED が点灯します。押したボタンを離した場合は 0 を出力して LED が消灯します。

```
def onTouch1(param1, param2)
  # LED を On/Off
  gr_digitalWrite($PIN_LED0, param1)
end
```

ボタンを押したときと離したときに自動的に実行するメソッド。

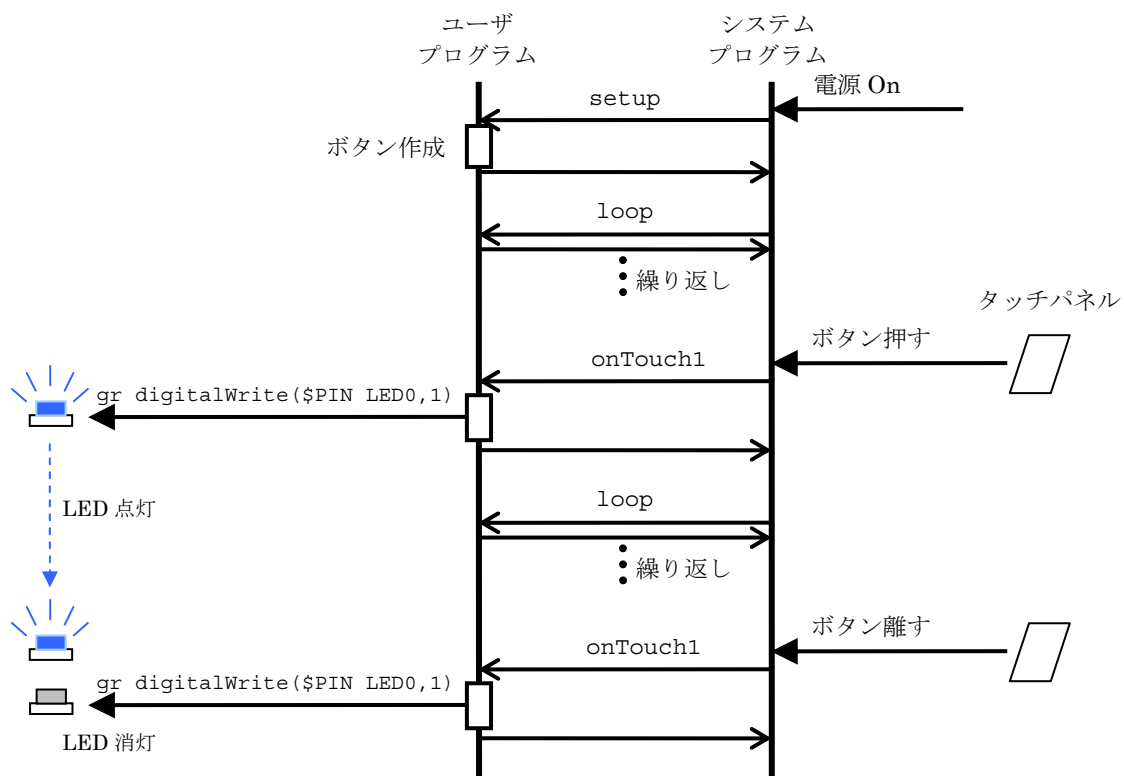
ボタンを押したときに LED を点灯して、  
離したときに LED を消灯します。



#### (4) 処理の流れ

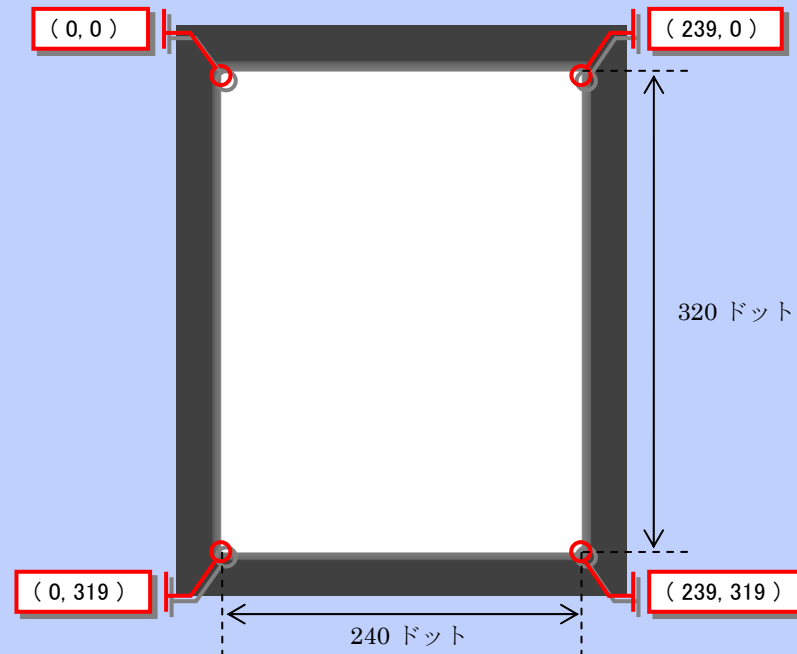
全体の処理の流れは下記となります。loop メソッドが繰り返し実行されます。その途中で、ボタンを押したときと離れたときだけ、onTouch1 メソッドが実行されます。

onTouch1 メソッドの中でボタンを押したときは LED0 を点灯、ボタンを離したときは LED0 を消灯します。それによって、ボタンを押している間は、LED0 は点灯したままとなります。



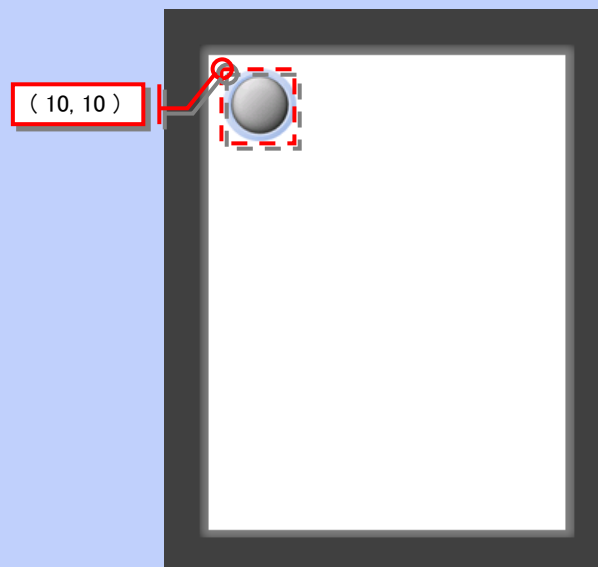
**Memo：液晶画面の座標について**

マイコンボードに搭載している液晶の表示画素数は横 240 ドット、縦 320 ドットです。  
左上の座標が (0, 0) となりますので、画面上の座標は下記となります。



画面上に配置する部品的位置を指定するには、部品の左上の座標を設定します。

`$button1 = GNButton.new(10, 10, "onTouch1")` と記述した場合、下記のようにボタンの左上が画面上の (10, 10) の位置になります。



## 2.4.2 LED輝度変更プログラム

### (1) プログラムの動作内容

前項ではLEDを点滅させるだけでしたが、付属のマイコンボードのLEDは明るさを256段階で調整することができます。

ここでは、LEDの明るさを連続的に変化させるプログラムを作成します。

### (2) プログラムの作成

下記のプログラムを作成して実行してください。

```
def setup()
  $button1 = GNButton.new(10, 10, "onTouch1")
  $numbmp1 = GNDigitalNum.new(170, 10)
  $slider1 = GNVSlider.new(180, 30, 0, 255, "onTouch2")
  gr_pinMode($PIN_LED0, $OUTPUT)
  gr_pinMode($PIN_LED3, $OUTPUT)
end

def onTouch1(param1, param2)
  # LEDをOn/Off
  gr_digitalWrite($PIN_LED0, param1)
end

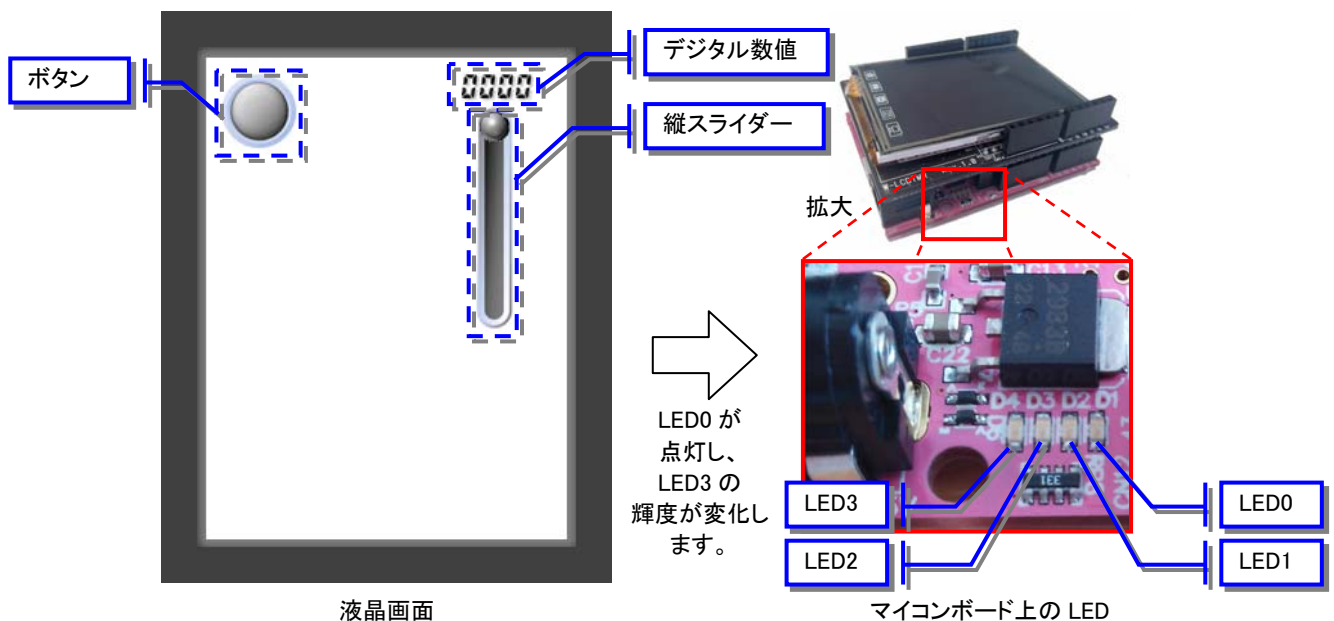
def onTouch2(param1, param2)
  gr_analogWrite($PIN_LED3, param1)
  $numbmp1.setInteger(param1)
end

def loop()
end
```

上記のプログラムを"sample\_04.rb"という名前で保存します。

インストールフォルダ内の sample¥sample04 フォルダに同じ内容のファイルが格納されています。

上記のプログラムをコンパイルしてSDカードに書き込み、実行すると下記のような画面が液晶に表示されます。



液晶画面に表示されているボタンをタッチすると、マイコンボード上の LED0 が点灯します。縦スライダーをタッチして上下にスライドさせると LED3 の輝度を変更することが可能です。また、輝度の値が数値表示されます。

### (3) 処理内容の説明

処理内容の概要は下記となります。

```
def setup()
  $button1 = GNButton.new(10, 10, "onTouch1")
  $numbmp1 = GNDigitalNum.new(170, 10)
  $slider1 = GNVSlider.new(180, 30, 0, 255, "onTouch2")

  gr_pinMode($PIN_LED0, $OUTPUT)
  gr_pinMode($PIN_LED3, $OUTPUT)
end

def onTouch1(param1, param2)
  # LED を On/Off
  gr_digitalWrite($PIN_LED0, param1)
end

def onTouch2(param1, param2)
  gr_analogWrite($PIN_LED3, param1)
  $numbmp1.setInteger(param1)
end

def loop()
end
```

液晶画面に配置する部品の設定。

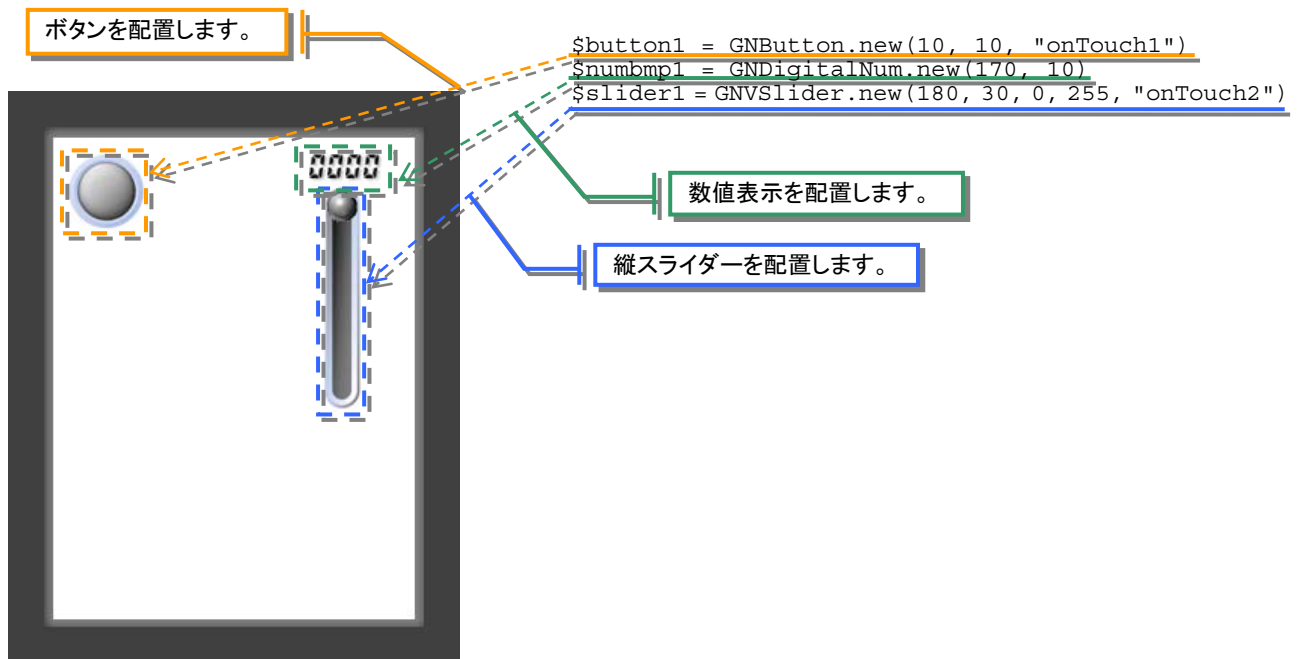
ボタンを押したときに実行する処理。

スライダーを操作したときに実行する処理。

loop 処理は不要なので、メソッドの中身を空にしておきます。

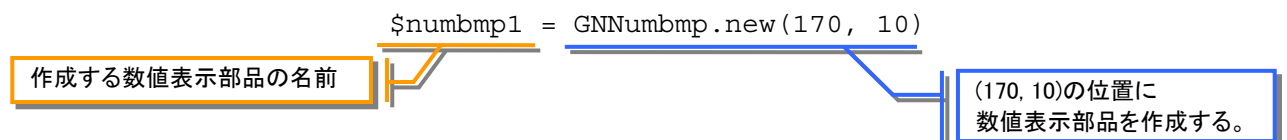
setup メソッドの中で画面に配置する部品を設定することができます。上記の例では、ボタンと縦スライダーとデジタル数値の部品を画面に配置しています。

液晶画面に配置する部品の設定と画面表示の対応は下記となります。

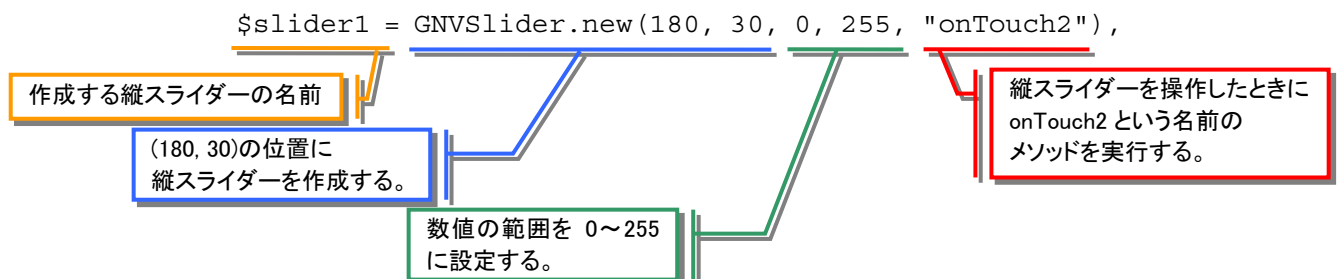


上記のように、画面に配置したい部品を作成すると、それらの部品が画面に表示されます。

デジタル数値は、画面上に4桁のデジタル数値を表示するための部品です。設定方法は下記となります。

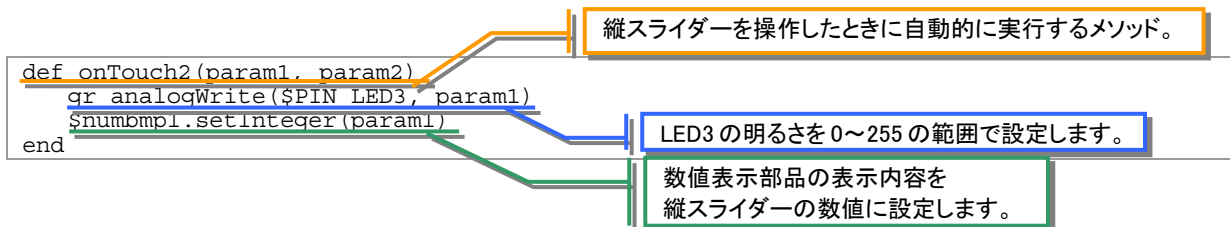


縦スライダーは、連続して数値を変化させることができる部品です。設定方法は下記となります。縦スライダーを操作した場合に変化する数値の範囲と、実行するメソッドの名前を設定することができます。

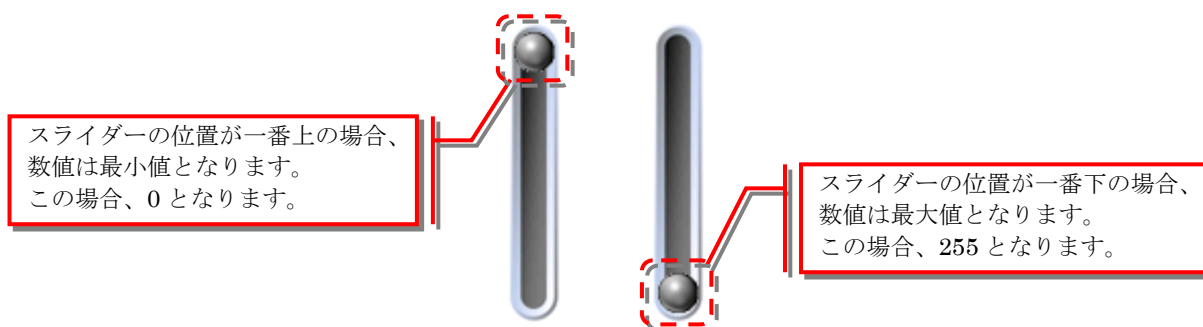


縦スライダーを操作したときに実行するメソッドの内容は下記となります。

縦スライダーを操作したときに実行するメソッドにはparam1とparam2という名前の引数があります。引数param1の値は縦スライダーに設定した範囲の間で変化します。この場合、0～255の範囲で変化します。



#### ■縦スライダーの表示内容と設定数値の関係



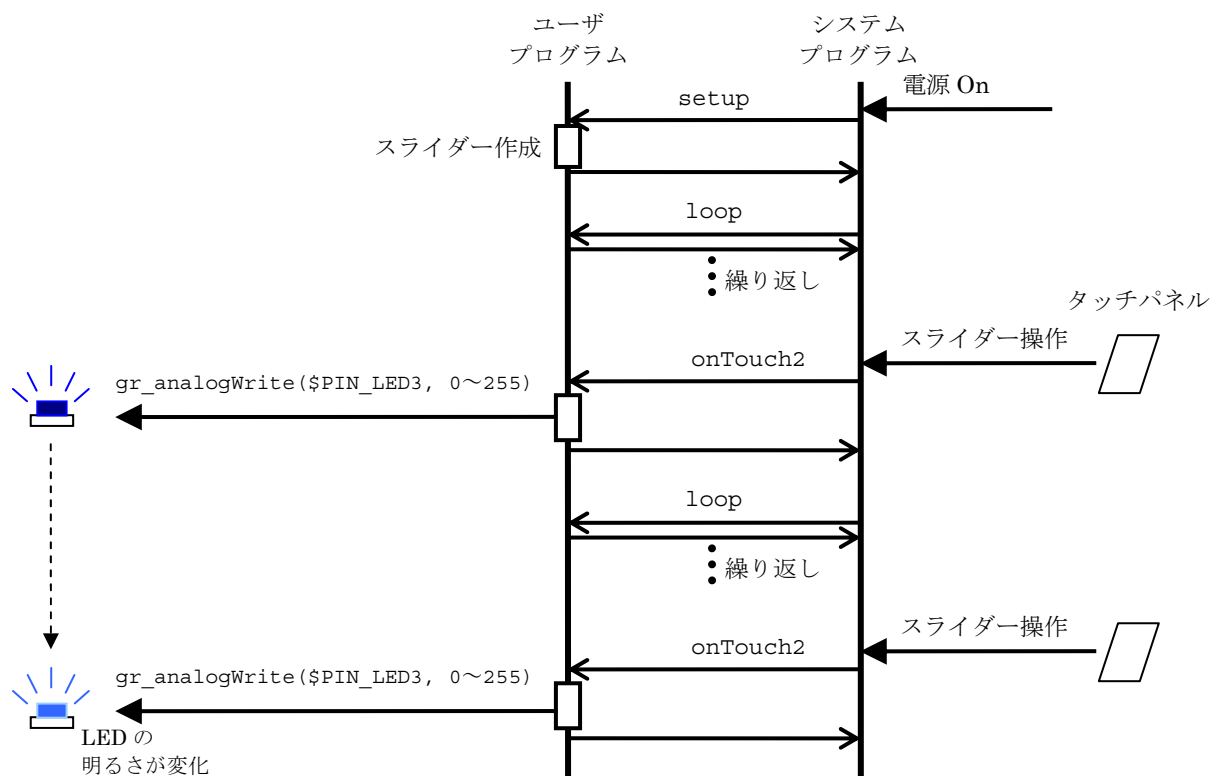
縦スライダーを操作して数値が変化すると上記の onTouch2 というメソッドが実行されます。メソッドの引数param1に縦スライダーの操作によって連続的に変化する数値が入りますので、その値をLED3の明るさに設定します。また、数値を画面上で確認するためにデジタル数値に設定します。

それによって、縦スライダーの操作によってLED3の明るさを連続的に変化させることができ、画面上でその値を確認することができます。

#### (4) 処理の流れ

全体の処理の流れは下記となります。loop メソッドが繰り返し実行されます。その途中で、縦スライダーを操作して数値が変化するときだけ、onTouch2 メソッドが実行されます。

onTouch2 メソッドの中でLED3 の明るさとデジタル数値の表示内容を変更します。それによって、縦スライダーの操作に連動して LED3 の明るさとデジタル数値の内容が連続的に変化します。



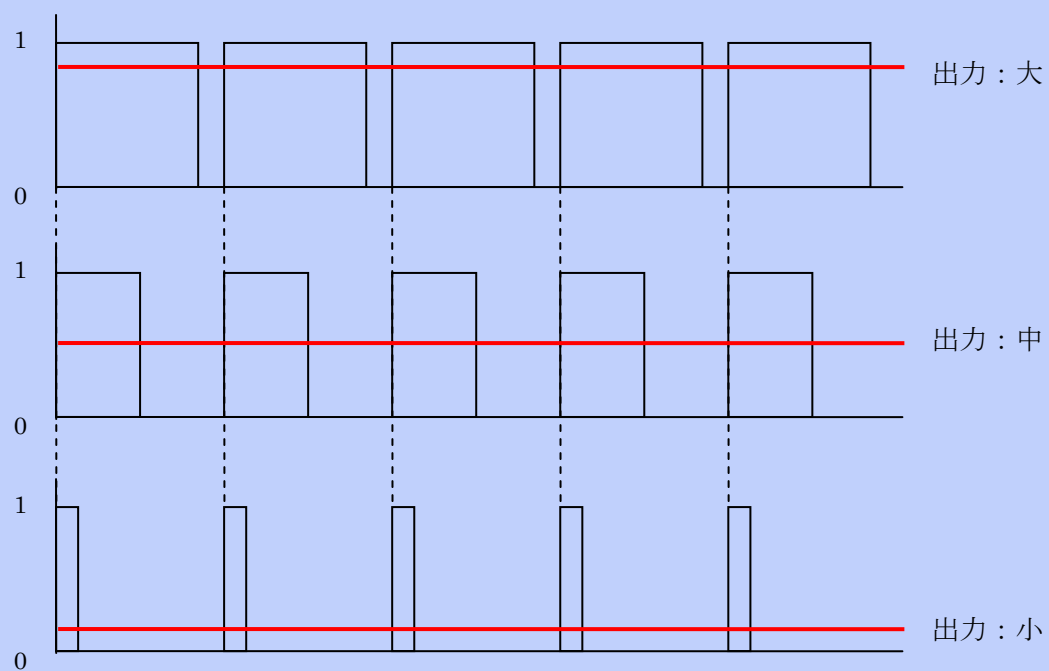
### Memo : PWM 制御について

LED0～LED3 を含めてマイコンボードに搭載しているデジタル I/O ピンの出力値は基本的には 0 か 1 のデジタル値です。しかし、本項の例のように LED の明るさをアナログ値で設定することもできます。

これは、PWM 制御と呼ばれる方式で出力値を変化させています。

PWM(Pulse Width Modulation)とは、デジタル値出力を小刻みに 0 と 1 で切り換えて出力電圧の平均値を変化させるという方式です。

出力を 1 にする時間を長くすれば出力の平均値は高くなり、短くすれば出力の平均値が低くなります。





## 2.5 画面表示部品クラス

### 2.5.1 画面に表示可能な部品

これまで、ボタンや縦スライダーなどの部品を液晶画面に表示して使ってきました。

ここでは、画面に表示して使うことができる部品について説明を行います。表示部品は mruby のクラス機能を使って実現しています。

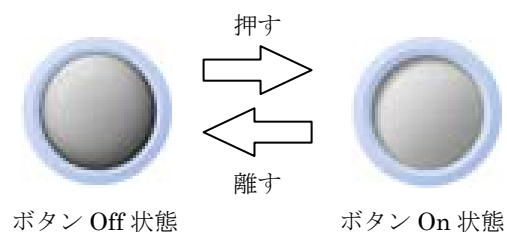
従って、部品を使用する場合には表示部品クラスをもとにインスタンスを生成する必要があります。液晶画面に表示して使用できる部品の一覧は下記となります。

| 表示部品名    | クラス名         | 表示  | 機能                          |
|----------|--------------|---|-----------------------------|
| ボタン      | GNButton     |    | 押している間だけ何かを実行したい場合に使います。    |
| スイッチ     | GNSwitch     |    | 押す度に On と Off を切換えることができます。 |
| 縦スライダー   | GNVSlider    |  | 縦方向にスライドさせて連続的に数値を変化させます。   |
| 横スライダー   | GNHSlider    |  | 横方向にスライドさせて連続的に数値を変化させます。   |
| デジタル数値   | GNDigitalNum |  | 4桁の数値を表示します。                |
| テキストボックス | GNTextbox    |  | 任意の英数字を表示します。               |
| ピクチャ     | GNPicture    |   | SD カード上の画像を表示することができます。     |
| デジタルパッド  | GNDigitalPad |  | 4方向のボタン入力を行うことができます。        |

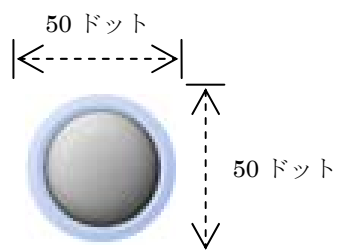
## 2.5.2 ボタン

### (1) 表示

画面上に押しボタンを表示します。タッチパネルを押すとボタンが On になり、離すと Off に戻ります。



画面上の大きさは下記となります。



### (2) ボタンの作り方

ボタンの作り方は下記となります。

#### ボタンの作り方

```
ボタンの名前 = GNButton.new(x 座標, y 座標, "メソッド名")
```

|        |                          |
|--------|--------------------------|
| ボタンの名前 | : 作成するボタンの名前             |
| x 座標   | : ボタンの表示位置 x 座標          |
| y 座標   | : ボタンの表示位置 y 座標          |
| メソッド名  | : ボタンを操作したときに実行するメソッドの名前 |

例) (40, 40) の位置に \$button1 という名前のボタンを作成して、ボタン操作時に onTouch1 というメソッドを実行します。

```
$button1 = GNButton.new(40, 40, "onTouch1")
```

**(3) ボタンを操作したときの処理**

ボタンを押したときと離れたときに実行します。ボタンを押し続けている間は実行しません。  
 ボタンを操作したときに実行するメソッドは以下の形となります。

**ボタンを操作したときの処理の書き方**

```
def メソッド名 (param1, param2)
end
```

|        |   |
|--------|---|
| メソッド名  | : ボタンを作成したときに設定したメソッド名                                |
| param1 | : ボタンの操作情報1<br>ボタンを押した場合は1 となります<br>ボタンを離した場合は0 となります |
| param2 | : ボタンの操作情報2<br>常に0 となります                              |

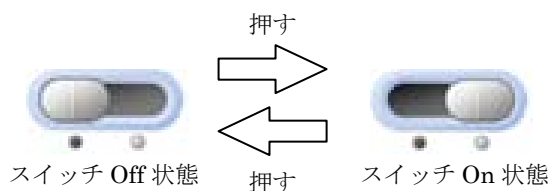
例) ボタンを押している間だけ LED を点灯します。

```
def onTouch1(param1, param2)
  # LED を On/Off
  gr_digitalWrite($PIN_LED0, param1)
end
```

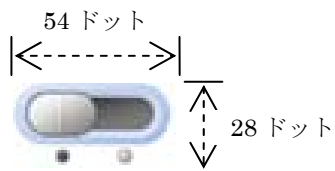
## 2.5.3 スイッチ

### (1) 表示

画面上に切換えスイッチを表示します。タッチパネルを押すたびにスイッチの状態が On と Off を繰り返します。



画面上の大きさは下記となります。



### (2) スイッチの作り方

スイッチの作り方は下記となります。

#### スイッチの作り方

```
def スイッチの名称 = GNSwitch.new(x 座標 , y 座標, "メソッド名" )
```

|         |   |                         |
|---------|---|-------------------------|
| スイッチの名称 | : | 作成するスイッチの名称             |
| x 座標    | : | スイッチの表示位置 x 座標          |
| y 座標    | : | スイッチの表示位置 y 座標          |
| メソッド名   | : | スイッチを操作したときに実行するメソッドの名称 |

例) (80, 40) の位置に \$switch1 という名称のスイッチを作成して、スイッチ操作時に onTouch2 というメソッドを実行します。

```
$switch1 = GNSwitch.new(80, 40, "onTouch2")
```

### (3) スイッチを操作したときの処理

スイッチを押したときと離れたときに実行します。スイッチを押し続けている間は実行しません。スイッチを操作したときに実行するメソッドは以下の形となります。

#### スイッチを操作したときの処理の書き方

```
def メソッド名 (param1, param2)
end
```

|        |  |
|--------|--|
| メソッド名  | : スイッチを作成したときに設定したメソッド名                              |
| param1 | : スイッチの状態<br>スイッチがOffの場合は0となります<br>スイッチがOnの場合は1となります |
| param2 | : スイッチの操作情報2<br>常に0となります                             |

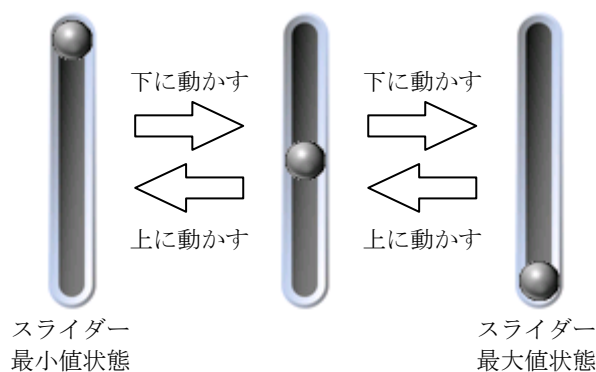
例) スイッチがOnのときはLEDを点灯して、スイッチがOffのときはLEDを消灯します

```
def onTouch2(param1, param2)
  # LEDをOn/Off
  gr_digitalWrite($PIN_LED0, param1)
end
```

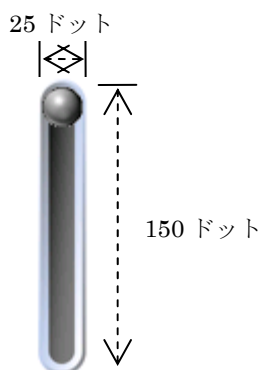
## 2.5.4 縦スライダー

### (1) 表示

画面上に縦方向のスライダーを表示します。タッチパネル押しながら滑らかに上下に動かすことができます。



画面上の大きさは下記となります。



### (2) 縦スライダーの作り方

縦スライダーの作り方は下記となります。

#### 縦スライダーの作り方

縦スライダーの名前 = `GNVSlider.new(x 座標, y 座標, 最小値, 最大値, "メソッド名")`

縦スライダーの名前 : 作成する縦スライダーの名前

x 座標 : ボタンの表示位置 x 座標

y 座標 : ボタンの表示位置 y 座標

最小値 : スライダー最小状態の数値

最大値 : スライダー最大状態の数値

メソッド名 : 縦スライダーを操作したときに実行するメソッドの名前

例) (60, 40)の位置に\$slider1 という名前の縦スライダーを作成して、最小値を 0、最大値を 255 とする。スライダー操作時に onTouch3 というメソッドを実行します。

```
$slider1 = GNVSlider.new(60, 40, 0, 255, "onTouch3")
```

**(3) 縦スライダーを操作したときの処理**

縦スライダーを操作して数値が変化したときに実行します。縦スライダーを操作したときに実行するメソッドは以下の形となります。

**縦スライダーを操作したときの処理の書き方**

```
def メソッド名 (param1, param2)
end
```

|        |   |
|--------|---|
| メソッド名  | : 縦スライダーを作成したときに設定したメソッド名                                       |
| param1 | : 縦スライダーの数値(最小値と最大値の間の値をとります)                                   |
| param2 | : 縦スライダーのタッチ状態<br>縦スライダーを押している間は1 となります<br>縦スライダーを離した場合は0 となります |

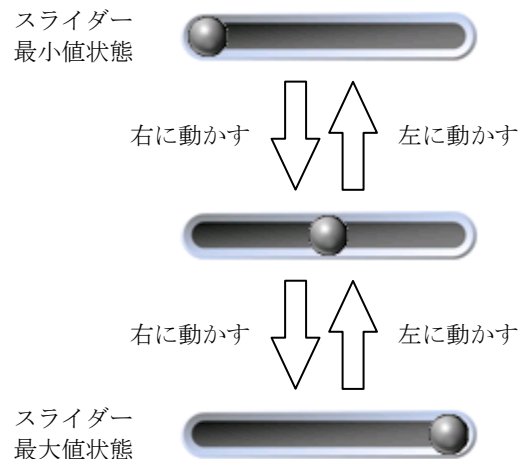
例) 縦スライダーの操作に連動して LED の明るさを変化させます。

```
def onTouch3(param1, param2)
  # LED の明るさを変える
  gr_analogWrite($PIN_LED0, param1)
end
```

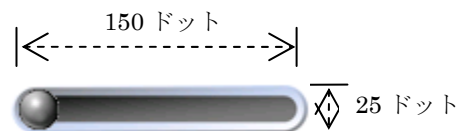
## 2.5.5 横スライダー

### (1) 表示

画面上に横方向のスライダーを表示します。タッチパネル押しながら滑らかに左右に動かすことができます。



画面上の大きさは下記となります。



### (2) 横スライダーの作り方

横スライダーの作り方は下記となります。

#### 縦スライダーの作り方

横スライダーの名前 = GNHSlider.new(x 座標 , y 座標, 最小値, 最大値, "メソッド名" )

|        |                          |
|--------|--------------------------|
| ボタンの名前 | : 作成するボタンの名前             |
| x 座標   | : ボタンの表示位置 x 座標          |
| y 座標   | : ボタンの表示位置 y 座標          |
| 最小値    | : スライダー最小状態の数値           |
| 最大値    | : スライダー最大状態の数値           |
| メソッド名  | : ボタンを操作したときに実行するメソッドの名前 |

例) (40, 60) の位置に \$slider2 という名前の横スライダーを作成して、最小値を 0、最大値を 255 とします。また、スライダー操作時に onTouch4 というメソッドを実行します。

```
$slider2 = GNVSlider.new(40, 60, 0, 255, "onTouch4")
```



### (3) 横スライダーを操作したときの処理

横スライダーを操作して数値が変化したときに実行します。横スライダーを操作したときに実行するメソッドは以下の形となります。

#### 横スライダーを操作したときの処理の書き方

```
def メソッド名 (param1, param2)
end
```

|        |   |
|--------|---|
| メソッド名  | : 横スライダーを作成したときに設定したメソッド名                                       |
| param1 | : 横スライダーの数値(最小値と最大値の間の値をとります)                                   |
| param2 | : 横スライダーのタッチ状態<br>横スライダーを押している間は1 となります<br>横スライダーを離した場合は0 となります |

例) 横スライダーの操作に連動して LED の明るさを変化させます。

```
def onTouch4(param1, param2)
  # LED の明るさを変える
  gr_analogWrite($PIN_LED0, param1)
end
```

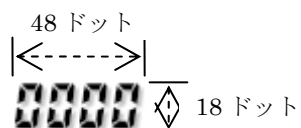
## 2.5.6 デジタル数値

### (1) 表示

画面上に4桁のデジタル数値を表示します。0～9999の数値を表示できます。



画面上の大きさは下記となります。



### (2) デジタル数値の作り方

デジタル数値の作り方は下記となります。作成した直後はデジタル数値の表示は"0000"です。

#### デジタル数値の作り方

```
デジタル数値の名前 = GNDigitalNum.new(x 座標 , y 座標 )
```

|                           |                    |
|---------------------------|--------------------|
| デジタル数値の名前 : 作成するデジタル数値の名前 |                    |
| x 座標                      | : デジタル数値の表示位置 x 座標 |
| y 座標                      | : デジタル数値の表示位置 y 座標 |

例) (80,50)の位置に\$digitalNum1 という名前のデジタル数値を作成します。

```
$digitalNum1 = GNDigitalNum.new(80, 50)
```

### (3) デジタル数値の表示数値変更

デジタル数値の表示内容をプログラムで変更できます。数値を変更するときには以下のメソッドを使用します。

#### デジタル数値の数値を変更するときの書き方

デジタル数値の*名前*.setInteger(*表示数値*)

デジタル数値の*名前* : 数値を変更するデジタル数値の*名前*  
*表示数値* : 表示する値(0～9999)

表示数値の値として0より小さい数値を設定した場合の表示は“0000”、9999より大きい数値を設定した場合の表示は“9999”となります。

例) デジタル数値の値を3000に変更します。

```
$digitalNum1 = GNDigitalNum.new(80, 50)  
$digitalNum1.setInteger(3000)
```

## 2.5.7 テキストボックス

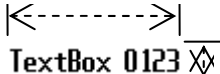
### (1) 表示

画面上に英数字文字列を表示します。

**TextBox 0123**

英数字文字列を表示

画面上の大きさは下記となります。

文字列の長さに依存  
  
 TextBox 0123 14 ドット

### (2) テキストボックスの作り方

テキストボックスの作り方は下記となります。

#### テキストボックスの作り方

テキストボックスの名前 = GNT textbox.new(x 座標, y 座標, 表示幅, 表示文字列)

|             |                      |
|-------------|----------------------|
| テキストボックスの名前 | : 作成するテキストボックスの名前    |
| x 座標        | : テキストボックスの表示位置 x 座標 |
| y 座標        | : テキストボックスの表示位置 y 座標 |
| 表示幅         | : 画面上の表示幅            |
| 表示文字列       | : 表示する文字列            |

表示文字列に設定できる文字の長さは 64 文字までです。1 文字の幅は文字によって異なりますので、表示文字列の内容によって表示幅を調整してください。表示幅を超えた文字は表示されません。表示文字列として使用できる文字一覧は ASCII コードの 32~125 までの文字となります。

使用可能文字一覧)

```
空白文字 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ¥ ] ^ _
a b c d e f g h i j k l m n o p q r s t u v w x y z { | }
```

ただし、“¥”の文字はバックスラッシュ文字として表示されます。

例) (30, 50) の位置に表示幅が 90 の \$textbox1 という名前のテキストボックスを作成します。表示する文字列は「TextBox 0123」とします。

```
$textbox1 = GNT textbox.new(30, 50, 90, "TextBox 0123")
```

### (3) テキストボックスの表示文字列変更

テキストボックスの表示内容をプログラムで変更できます。表示文字列を変更するときには以下のメソッドを使用します。

#### テキストボックスの表示文字列を変更するときの書き方

テキストボックスの*名前*.setString(*表示文字列*)

|                     |   |                            |
|---------------------|---|----------------------------|
| テキストボックスの <i>名前</i> | : | 数値を変更するテキストボックスの <i>名前</i> |
| <i>表示文字列</i>        | : | 表示する文字列                    |

例) テキストボックスの表示文字列を"textbox ABC"に変更します。

```
$textbox1 = GNTtextbox.new(30, 50, 90, "TextBox 0123")  
$textbox1.setString("textbox ABC")
```

## 2.5.8 ピクチャ

### (1) 表示

マイコンボード上に搭載した SD カード上の画像データを画面上に表示します。

### (2) ピクチャの作り方

ピクチャの作り方は下記となります。

#### ピクチャの作り方

ピクチャの名前 = `GNPicture.new(x 座標 , y 座標 , 画像ファイル名 )`

|         |                     |
|---------|---------------------|
| ピクチャの名前 | : 作成するピクチャの名前       |
| x 座標    | : ピクチャの表示位置 x 座標    |
| y 座標    | : ピクチャの表示位置 y 座標    |
| 画像ファイル名 | : SD カード上にある画像ファイル名 |

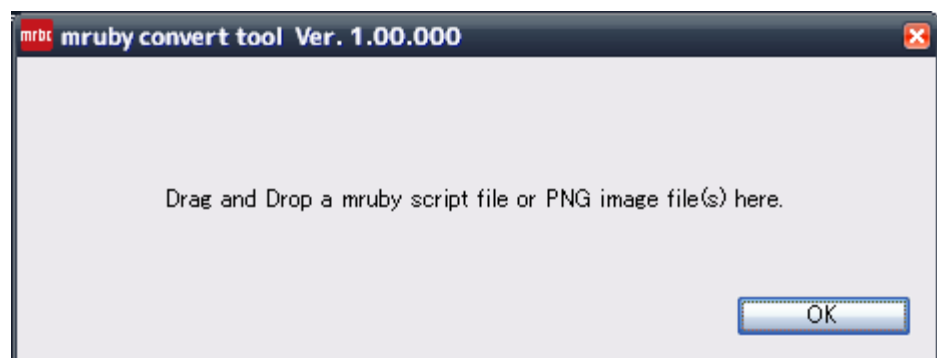
表示可能な画像ファイルはあらかじめ PC 上で作成しておいて SD カードに書き込んでおく必要があります。

画像形式は独自のものですので、PC 上で変換する必要があります。

### (3) 画像データの変換

画像データの変換には、mruby プログラムの変換に使用したツールを使用します。

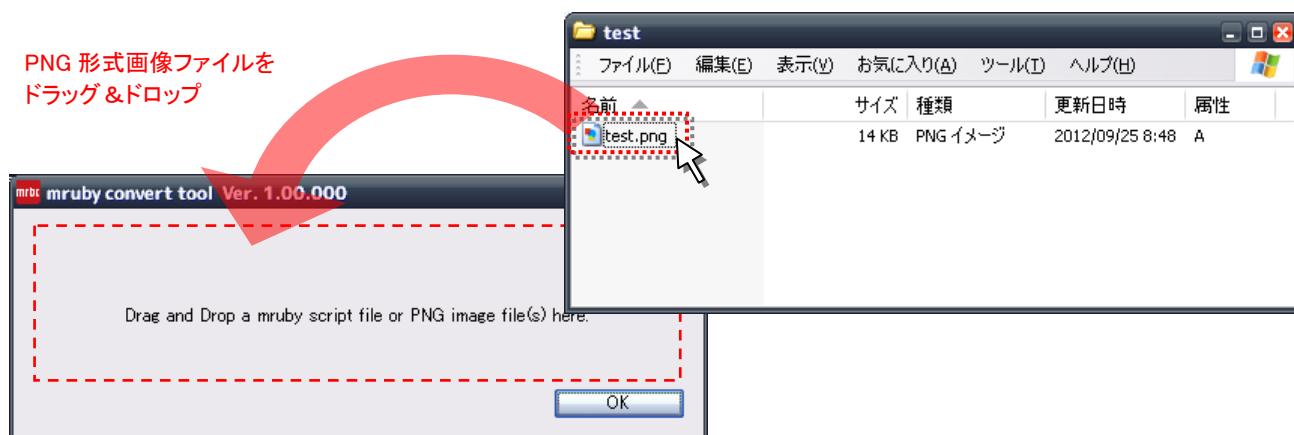
インストールフォルダ内の `tool` フォルダに格納されている "`mrbcConvert.exe`" というプログラムを実行してください。実行すると下記の画面が表示されます。



PNG 形式の画像ファイルを上記の画面にドラッグ&ドロップしてください。

ただし、PNG 形式の画像ファイルのファイル名の長さは拡張子 ".png" を除いて 8 文字以下である必要があります。

"abcdefgh.png" というファイル名の画像は変換できますが、"abcdefghi.png" というファイル名の画像は変換できません。



正常に画像データが変換されると、下記のように画像ファイルと同じフォルダにPNG形式画像ファイルと同じ名前で拡張子が“.gbf”となっているファイルが生成されます。

何か問題が発生するとエラーダイアログが表示される場合があります。その場合は第4章4.1.2を参照してください。



上記で生成された拡張子が“.gbf”というファイルをSDカードにコピーしてください。

その際にSDカードにはフォルダを生成せずSDカードの直下にファイルをコピーしてください。

SDカードに画像ファイルをコピーした状態で下記のようにピクチャを作成すると液晶画面上に画像が表示されます。

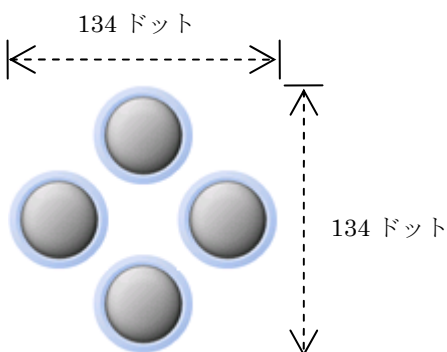
例) (20, 50)の位置に\$picture1という名前のピクチャを作成します。表示する画像はSDカード上の“test.gbf”という名前の画像ファイルとします。

```
$picture1 = GNPicture.new(20, 50, "test.gbf")
```

## 2.5.9 デジタルパッド

### (1) 表示

画面上に4方向の操作ボタンを表示します。ゲーム等、4つの方向をタッチ操作で指示したい場合に使います。



### (2) デジタルパッドの作り方

デジタルパッドの作り方は下記となります。

#### デジタルパッドの作り方

```
デジタルパッドの名前 = GNDigitalPad.new( x 座標 , y 座標, "メソッド名" )
```

デジタルパッドの名前： 作成するデジタルパッドの名前  
 x 座標                   : デジタルパッドの表示位置 x 座標  
 y 座標                   : デジタルパッドの表示位置 y 座標  
 メソッド名               : デジタルパッドを操作したときに実行するメソッドの名前

例) (20,20)の位置に\$digitalpad1 という名前のデジタルパッドを作成して、デジタルパッド操作時に onPad というメソッドを実行します。

```
$digitalpad1 = GNDigitalPad.new(20, 20, "onPad")
```



### (3) デジタルパッドを操作したときの処理

デジタルパッドを操作したときに実行します。

デジタルパッドを操作したときに実行するメソッドは以下の形となります。

#### デジタルパッドを操作したときの処理の書き方

```
def メソッド名 (param1, param2)
end
```

|        |   |
|--------|---|
| メソッド名  | : デジタルパッドを作成したときに設定したメソッド名  |
| param1 | : デジタルパッドの操作情報 1<br>上方向を押した場合は \$DPAD_UP となります<br>下方向を押した場合は \$DPAD_DOWN となります<br>左方向を押した場合は \$DPAD_LEFT となります<br>右方向を押した場合は \$DPAD_RIGHT となります |
| param2 | : デジタルパッドの操作情報 2<br>常に 1 となります  |

例) デジタルパッドを押した方向を dir という変数に取得します。

```
def onPad(param1, param2)
  # 方向を取得(変数 dir にデジタルパッドの操作方向が格納されます)
  dir = param1
end
```

## 2.6 画面描画メソッド

### 2.6.1 画面描画機能

表示部品を使わずに直接液晶画面にグラフィック表示を行う機能もあります。

画面表示部品との違いは、クラスを使用しないためインスタンスを生成する必要はありません。メソッドを呼ぶだけで画面描画を行うことができます。

液晶画面に描画を行うメソッドの一覧は下記となります。

| 名前   | メソッド名       | 機能                    |
|------|-------------|-----------------------|
| 画像表示 | GNDrawImage | SD カード上の画像ファイルを表示します。 |
| 矩形表示 | GNDrawRect  | 画面上に指定した色の矩形を表示します。   |

### 2.6.2 画像表示

#### (1) 機能概要

マイコンボード上に搭載した SD カード上の画像データを画面上に表示します。

#### (2) 画像データの変換

表示可能な画像データはピクチャ部品で使用するものと同じものとなります。画像データの作成方法は 2.5.8(3) 項を参照してください。

#### (3) メソッドの使い方

メソッドの使い方を下記に示します。

##### 画像表示メソッドの書き方

```
GNDrawImage ( 画像表示 x 座標, 画像表示 y 座標, 画像データファイル名 )
```

例) 画面上の (50, 100) の位置に "test.gbff" という名前の画像ファイルを表示する。

```
GNDrawImage(50, 100, "test.gbff")
```

## 2.6.3 矩形表示

### (1) 機能概要

画面上の任意の位置に指定した色の矩形を表示します。

### (2) メソッドの使い方

メソッドの使い方を下記に示します。

#### 矩形表示メソッドの書き方

`GNDDrawRect` (矩形表示 x 座標, 矩形表示 y 座標, 矩形幅, 矩形高さ, 表示色 R, 表示色 G, 表示色 B)

表示色の指定は、光の三原色である赤(R), 緑(G), 青(B)の値で行います。それぞれ 0～31 の 32 段階の数値を指定可能です。

例) 画面上の (50, 100) の位置に幅 30、高さ 20 の青色の矩形を表示する。

```
GNDDrawRect(50, 100, 30, 20, 0, 0, 31)
```

## 2.7 写真表示プログラム

### 2.7.1 プログラムの動作内容

液晶画面を使って自分の好きな写真を表示するプログラムを作成します。

### 2.7.2 画像データの作成

自分で用意した画像ファイルを液晶画面に表示するためには、まず最初に画像データの作成処理を行います。

画像データを作成するためには、PNG 形式の画像ファイルを用意する必要があります。ここでは、下記の名前の PNG 形式画像ファイルを用意します。

```
bluesky.png
flower.png
green.png
leaf.png
tile.png
```

上記の PNG 形式画像ファイルは、インストールフォルダ内の **sample¥sample05** フォルダに格納されています。

画像データを変換するためにインストールフォルダ内の **tool** フォルダに格納されている **"mrbcConvert.exe"** というプログラムを実行してください。起動した画面に、上記のファイルを全てドラッグ&ドロップすると拡張子が **".gbf"** 形式のファイルに変換されます。

変換した結果のファイルも、インストールフォルダ内の **sample¥sample05** フォルダに格納されています。

### 2.7.3 プログラムの作成

下記のプログラムを作成して実行してください。

```
#=====
# メイン初期化
#-----
def setup()
  $counter = 0
  $photoNo = 0
end

# メインループ
#-----
def loop()
  if $counter == 0 then
    case $photoNo
      when 0 then
        GNDrawImage(0, 0, "tile.gbf")
      when 1 then
        GNDrawImage(0, 0, "bluesky.gbf")
      when 2 then
        GNDrawImage(0, 0, "flower.gbf")
      when 3 then
        GNDrawImage(0, 0, "green.gbf")
    end
  end
end
```

```

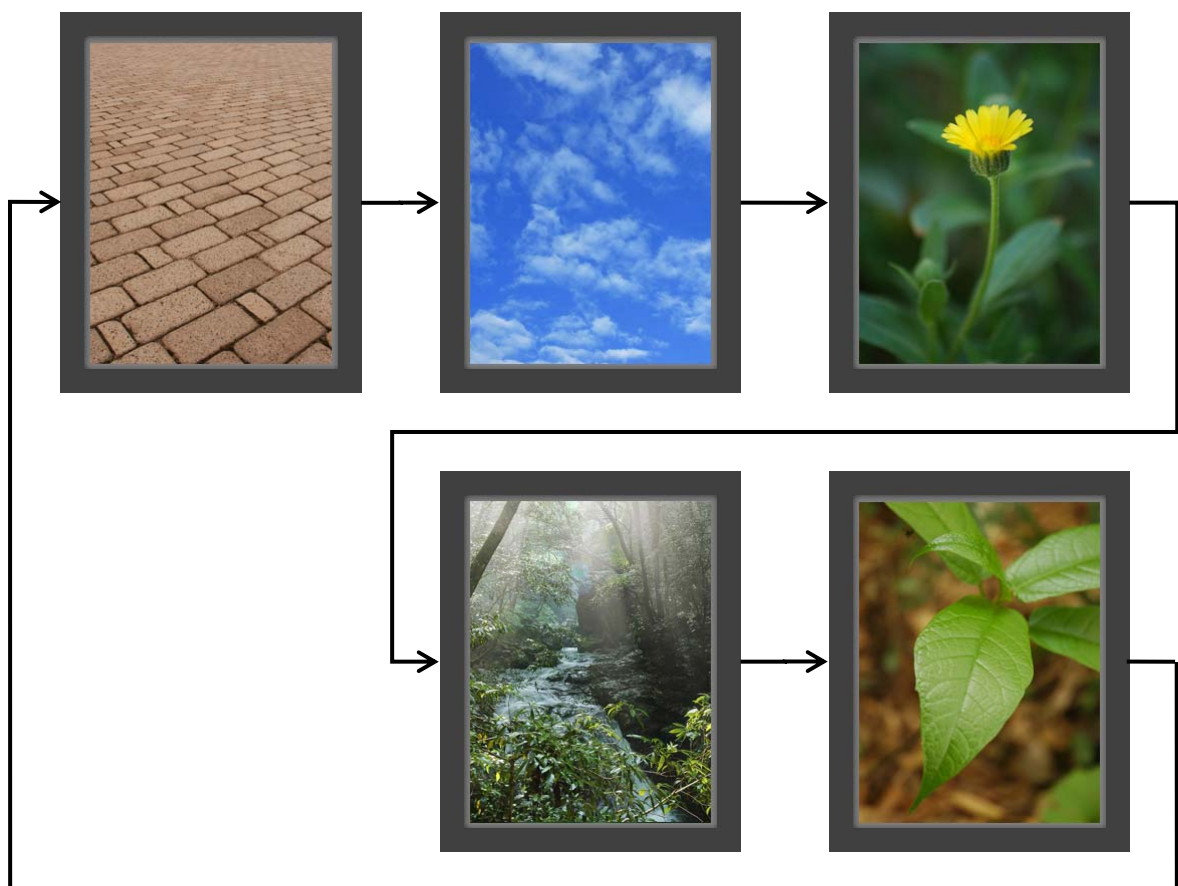
        when 4 then
            GNDrawImage(0, 0, "leaf.gbf")
        end
    end
end

$counter += 1
if $counter >= 50000 then
    $counter = 0
    $photoNo += 1
    if $photoNo >= 5 then
        $photoNo = 0
    end
end
end
end
end

```

インストールフォルダ内の **sample¥sample05** フォルダに“**photoFrame.rb**”という名前で同じ内容のファイルと、そのコンパイル結果のファイル“**program.mbi**”が格納されています。

実行すると下記のような画面が順番に液晶画面に表示されます。

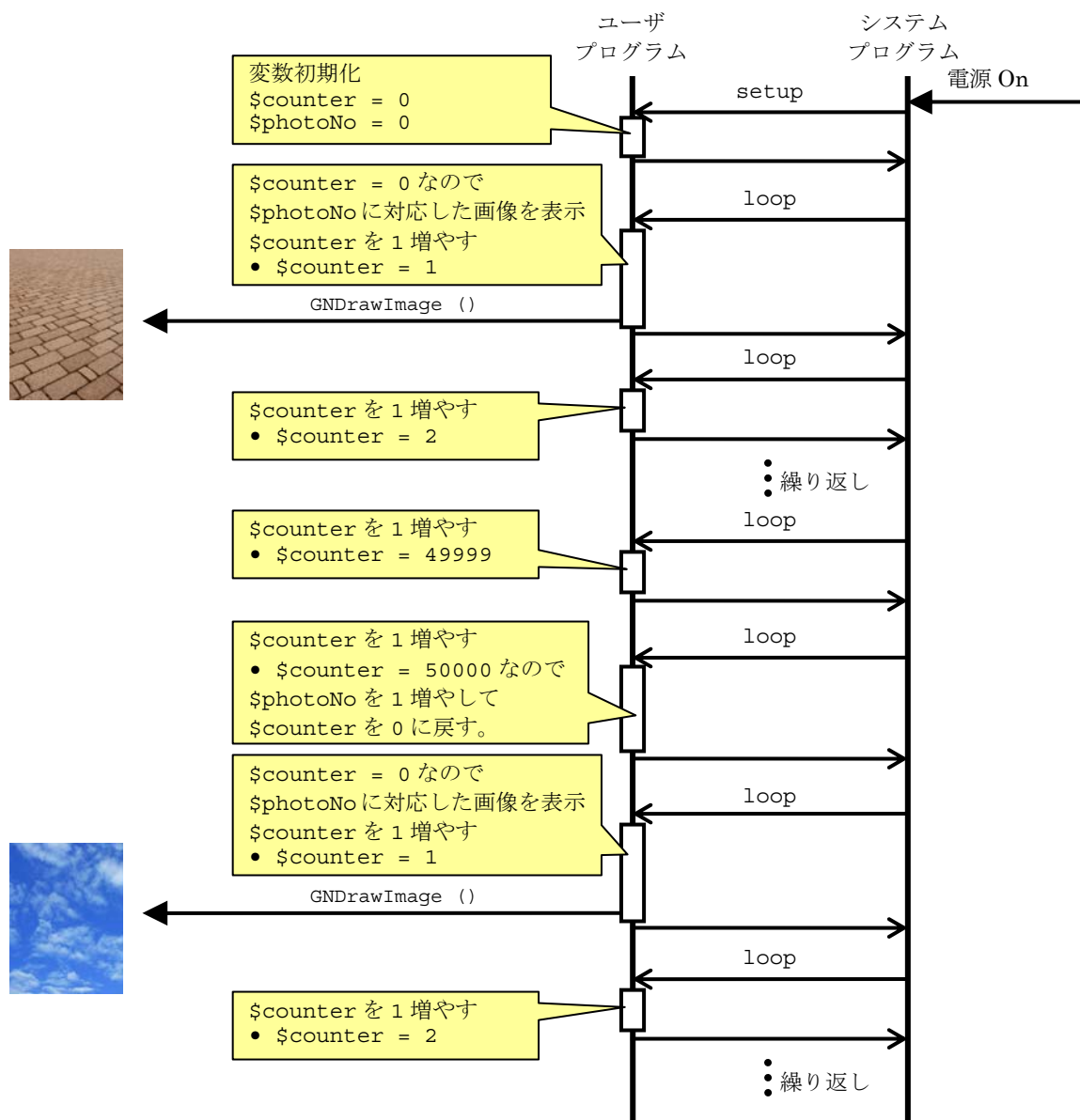


## 2.7.4 プログラム内容の説明

時間経過とともに5種類の画像を順番に表示します。表示する画像を指定するために\$photoNo という変数を使用します。

また、一定時間毎に\$photoNo を切替えるために周期処理の中でループの回数をカウントします。loop メソッドを実行するたびに変数\$counter を1ずつ増加させます。\$counter の値が 50000 以上になると\$photoNo の値を1増やして、\$counter を0に戻します。

これを繰り返すことによって、一定時間ごとに写真を繰り返して表示します。



## 2.8 タッチパネルを使ったゲーム

### 2.8.1 プログラムの動作内容

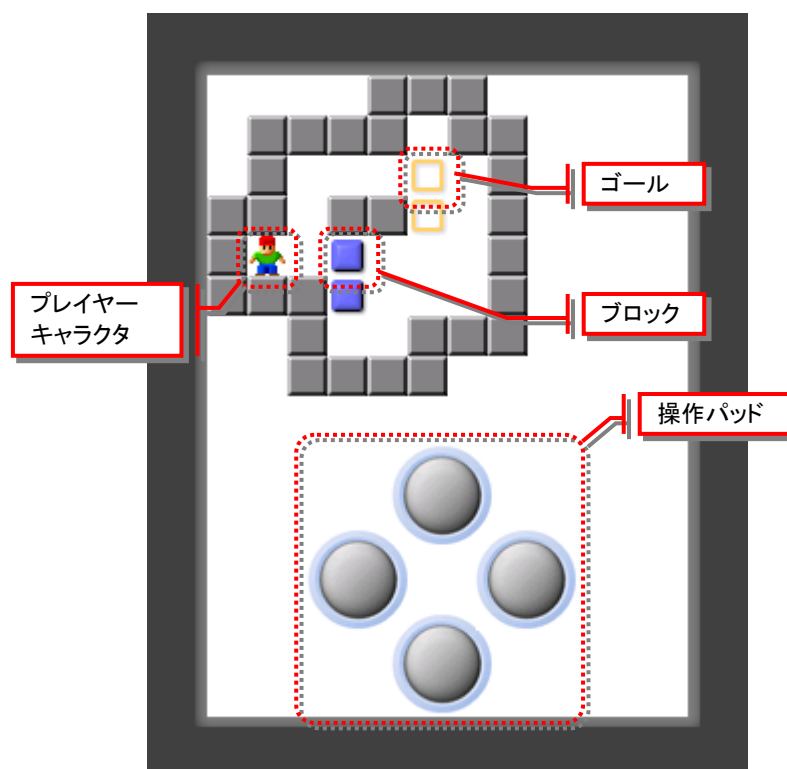
本項では、LCD 画面のタッチパネルを使った簡単なゲームを動かす例を説明します。

画面上に4方向ボタンとパズル画面が表示されます。

4方向ボタンを操作してプレイヤーキャラクタを移動させて、画面上の青いブロックをゴールまで運びます。全てのブロックをゴールまで運ぶとクリアとなります。

プレイヤーキャラクタは青いブロックを押すことはできますが、引くことはできません。また、一度に押すことができるブロックは1つだけです。

全てのブロックをゴールまで移動させるとステージクリアとなり、次のステージに進みます。今回はサンプルですのでステージ数は2とします。2つめのステージをクリアすると最初のステージに戻ります。



## 2.8.2 画像データの準備

画面に表示するキャラクタの画像データを準備します。ここでは、下記の名前の PNG 形式画像ファイルを用意します。

|            | ファイル名      | 画像  | 大きさ   |
|------------|------------|---|-------|
| プレイヤーキャラクタ | player.png |          | 20x20 |
| ブロック       | block.png  |          | 20x20 |
| ゴール        | goal.png   |          | 20x20 |
| 壁          | wall.png   |          | 20x20 |
| 床          | floor.png  |  (白色の画像) | 20x20 |
| クリア時メッセージ  | clear.png  |       | 80x20 |

上記の PNG 形式画像ファイルは、インストールフォルダ内の **sample¥sample06** フォルダに格納されています。

また、画像変換した結果の拡張子が **“.gbf”** のファイルも、同じフォルダに格納されています。

## 2.8.3 プログラムの作成

下記のプログラムを作成して実行してください。

同じ内容のプログラムファイルがインストールフォルダ内の **sample¥sample06** フォルダに **“blockgame.rb”** という名前で格納されています。

```
# 状態遷移の定義
$ST_CREATE = 0
$ST_PLAY   = 1
$ST_CLEAR  = 2

# ステージデータの定義
$DT_FLOOR = 0
$DT_WALL  = 1
$DT_PLAYER = 2
$DT_BLOCK = 3
$DT_GOAL  = 4

# 画面上の表示物の大きさ
$UNIT_SCALE = 20

# ステージデータ
$totalStage = 2
$stageData = [
  [
    0, 0, 1, 1, 1, 1, 0, 0,
```



```

1, 1, 1, 0, 0, 1, 0, 0,
1, 2, 0, 3, 0, 1, 0, 0,
1, 1, 1, 1, 0, 1, 0, 0,
0, 0, 1, 0, 0, 1, 1, 0,
0, 0, 1, 0, 0, 4, 1, 0,
0, 0, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0,
],
[
0, 0, 0, 0, 1, 1, 1, 0,
0, 1, 1, 1, 1, 0, 1, 1,
0, 1, 0, 0, 0, 4, 0, 1,
1, 1, 0, 1, 1, 4, 0, 1,
1, 2, 0, 3, 0, 0, 0, 1,
1, 1, 1, 3, 0, 0, 0, 1,
0, 0, 1, 0, 0, 1, 1, 1,
0, 0, 1, 1, 1, 1, 0, 0,
],
]
# ステージ情報参照メソッド
def getStageData(x, y)
  return $stageData[$stage_no][y*8+x]
end

# ステージ生成メソッド
def createStage()
  for y in 0..7
    for x in 0..7
      case getStageData(x, y)
      when $DT_FLOOR
        # 床を描画
        drawFloor(x, y)
      when $DT_WALL
        # 壁を描画
        drawWall(x, y)
      when $DT_PLAYER
        # プレイヤーキャラクタを描画
        $player.setPos(x, y)
      when $DT_BLOCK
        # ブロックを生成して描画
        $blockList.push(DisplayObject.new(x, y, "block.gbf"))
      when $DT_GOAL
        # ゴールを描画
        drawGoal(x, y)
      end
    end
  end
end

# 壁描画メソッド
def drawWall(x, y)
  GNDdrawImage(x * $UNIT_SCALE, y * $UNIT_SCALE, "wall.gbf")
end

# 床描画メソッド
def drawFloor(x, y)
  GNDdrawImage(x * $UNIT_SCALE, y * $UNIT_SCALE, "floor.gbf")
end

# ゴール描画メソッド
def drawGoal(x, y)
  GNDdrawImage(x * $UNIT_SCALE, y * $UNIT_SCALE, "goal.gbf")
end

# ブロック有無判定メソッド
def isBlockExist(x, y)
  $blockList.each { |block|
    if (x == block.posX) && (y == block.posY) then
      return block
    end
  }
  return nil
end

```

```

# 画面表示物クラス
class DisplayObject
  attr_accessor :posX, :posY

  # 初期化处理
  def initialize(x, y, fName)
    @fName = fName
    setPos(x, y)
  end

  # 位置設定
  def setPos(x, y)
    @posX = x
    @posY = y
    GNDDrawImage(@posX * $UNIT_SCALE, @posY * $UNIT_SCALE, @fName)
  end

  # 移動
  def moveTo(x, y)
    # 移動元を描画
    if getStageData(@posX, @posY) == $DT_GOAL then
      drawGoal(@posX, @posY)
    else
      drawFloor(@posX, @posY)
    end

    # 現在位置を移動、移動先に画像を描画
    setPos(x, y)
  end
end

#-----
# メイン初期化
#=====
def setup()
  # 状態初期化
  $state = $ST_CREATE

  # ステージ番号初期化
  $stage_no = 0

  # ブロック配列生成
  $blockList = []

  # デジタルパッド生成
  $digitalpad = GNDigitalPad.new(50, 185, "onPad")

  # プレイヤー生成
  $player = DisplayObject.new(0, 0, "player.gbf")
end

# メインループ
#-----
def loop()
  if $state == $ST_CREATE then
    # ブロック配列クリア
    $blockList.clear

    # ステージ生成
    createStage()

    # プレイ開始
    $state = $ST_PLAY
  end
end

# デジタルパッド操作時処理
#-----
def onPad(param1, param2)
  # ステージクリア状態の場合はパッド操作で次のステージへ
  if $state == $ST_CLEAR then

```

```

$stage_no += 1
if $stage_no >= $totalStage then
    $stage_no = 0
end

# 次のステージの生成を開始する
$state = $ST_CREATE

return
end

if $state == $ST_PLAY then
    # ゲームプレイ中状態の場合はプレイヤーの移動を行う
    dx = 0 # プレイヤーキャラクタ x 移動量
    dy = 0 # プレイヤーキャラクタ y 移動量
    isBlockMove = 1 # ブロック移動判定フラグ
    isStageClear = 1 # ステージクリア判定フラグ

    # デジタルパッド入力に対する移動処理
    case param1
    when $DPAD_UP
        dy = -1
    when $DPAD_DOWN
        dy = 1
    when $DPAD_LEFT
        dx = -1
    when $DPAD_RIGHT
        dx = 1
    end

    # プレイヤーの移動先にブロックがあるかどうか
    block = isBlockExist($player.posX + dx, $player.posY + dy)
    if block != nil then
        # ブロックの移動先の算出
        blockMoveX = $player.posX + dx * 2
        blockMoveY = $player.posY + dy * 2

        # ブロックの移動先に何かあるか
        if (isBlockExist(blockMoveX, blockMoveY) == nil) &&
            (getStageData(blockMoveX, blockMoveY) != $DT_WALL) then
            # ブロックの移動先にブロックも壁もなければ移動可能
            block.moveTo(blockMoveX, blockMoveY)
        else
            # ブロックが移動できない
            isBlockMove = 0
        end
    end

    # ゴール判定
    $blockList.each { |block|
        if getStageData(block.posX, block.posY) != $DT_GOAL then
            # ブロックがゴールの上に無ければステージクリア判定フラグを 0 にする
            isStageClear = 0
        end
    }

    # プレイヤーの移動
    if (getStageData($player.posX + dx, $player.posY + dy) != $DT_WALL) &&
        (isBlockMove == 1) then
        $player.moveTo($player.posX + dx, $player.posY + dy)
    end

    # クリア判定
    if isStageClear == 1 then
        GNDrawImage(80, 70, "clear.gbf")
        $state = $ST_CLEAR
    end
end
end
end

```

## 2.8.4 プログラム内容の説明

### (1) クラス定義

画面上で移動する物体を管理するための画面表示物クラスを作成します。

画面表示物クラスは下記を管理するために使用します。

- ・プレイヤーキャラクタ
- ・ブロック

```
# 画面表示物クラス
class DisplayObject
  attr_accessor :posX, :posY
  # 初期化处理
  def initialize(x, y, fName)
    @fName = fName
    setPos(x, y)
  end

  # 位置設定
  def setPos(x, y)
    @posX = x
    @posY = y
    GNDrawImage(@posX * $UNIT_SCALE, @posY * $UNIT_SCALE, @fName)
  end

  # 移動
  def moveTo(x, y)
    # 移動元を描画
    if getStageData(@posX, @posY) == $DT_GOAL then
      drawGoal(@posX, @posY)
    else
      drawFloor(@posX, @posY)
    end

    # 現在位置を移動、移動先に画像を描画
    setPos(x, y)
  end
end
```

attr\_accessor に続けて :インスタンス変数名 を記述することによって、インスタンス変数を直接外部から読み書き可能となります。

画面表示物クラスに関する説明を以下に記述します。

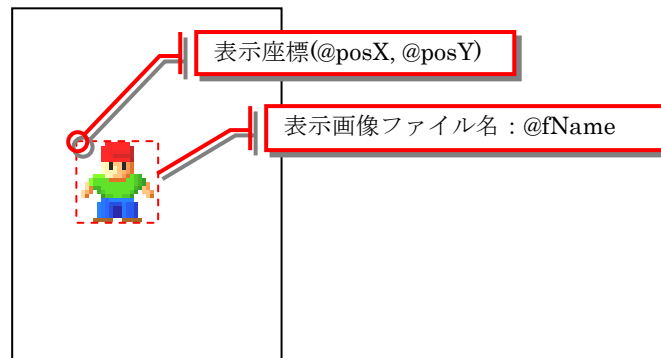
### ■インスタンス変数

インスタンス変数として下記の3つを持ちます。

@posX : 表示 x 座標

@posY : 表示 y 座標

@fName : 表示画像ファイル名



### ■インスタンスメソッド

インスタンスメソッドは下記となります。

#### initialize メソッド

```
initialize(x, y, fName)
    x      : 表示 x 座標
    y      : 表示 y 座標
    fName  : 表示画像ファイル名 (gbf 形式)
```

初期化処理です。インスタンス生成時に引数として渡された x, y 座標と表示画像ファイル名をインスタンスメソッドに格納します。

#### 表示位置設定メソッド

```
setPos(x, y)
    x      : 表示 x 座標
    y      : 表示 y 座標
```

表示位置設定処理です。引数で指定した座標に移動し、そこに画像ファイルの内容を表示します。

#### 移動処理メソッド

```
moveTo(x, y)
    x      : 表示 x 座標
    y      : 表示 y 座標
```

移動処理です。移動前の位置の表示を消去して、移動後の位置に表示を行います。  
移動前の位置にゴールがあればゴールを表示して、そうでなければ床を表示します。

## (2) 状態遷移

パズルゲームを作るにあたって、ゲームの状態を管理します。

その時のゲームの状態によって処理内容を決めます。また、条件によって状態を切換えます。これを、状態遷移設計と呼びます。

今回のゲームでは、下記の3つのゲーム状態を持つことにします。

- ・ **ステージ生成状態**

パズルゲームのステージを生成する状態です。ステージの生成が終わるとゲームプレイ中状態に切換えます。

- ・ **ゲームプレイ中状態**

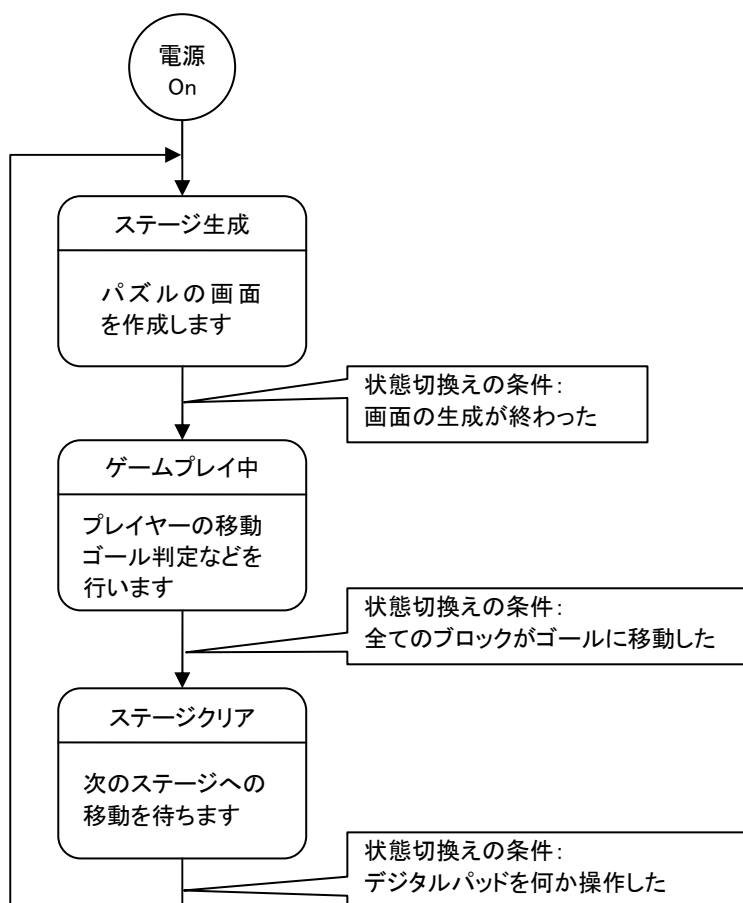
パズルゲームをプレイ中の状態です。デジタルパッドを操作してプレイヤーキャラクタを移動させることができます。

すべてのブロックをゴールまで移動させると、ステージクリア状態に切換えます。

- ・ **ステージクリア状態**

ステージクリア時の状態です。ステージクリアメッセージを表示し、デジタルパッドを操作するとステージ生成状態に切換えます。

上記の状態の関連図を以下に示します。



プログラムの中ではゲーム状態を以下の変数に格納して管理します。

```
$state
```

ゲーム状態の値は以下の変数を使います。

```

$ST_CREATE = 0    ステージ生成状態
$ST_PLAY   = 1    ゲームプレイ中状態
$ST_CLEAR  = 2    ステージクリア状態

```

### (3) ステージ生成状態の処理

ステージ生成状態での処理内容を説明します。

```

# メインループ
#-----
def loop()
  if $state == $ST_CREATE then
    # ブロック配列クリア
    $blockList.clear

    # ステージ生成
    createStage()

    # プレイ開始
    $state = $ST_PLAY
  end
end
end

```

ステージ生成処理を行います

ステージ生成処理が終わったら、  
ゲーム状態をゲームプレイ中状態に切換えます

まず、メインループではブロックを管理する配列をクリアしています。

プレイヤーキャラクタのインスタンスは1つだけ作ればよいのですが、ブロックは各ステージで生成する数が変わりますのでインスタンスの配列で管理します。

後述するステージ生成処理の中でブロックの配列に要素を追加していくため、ここでは配列の中身を空にします。

次にステージ生成処理を行い、全て完了するとゲーム状態をゲームプレイ中状態に切換えます。



次にステージ生成処理の説明を行います。

```
# ステージデータ
$totalStage = 2
$stageData = [
  [
    0, 0, 1, 1, 1, 1, 0, 0,
    1, 1, 1, 0, 0, 1, 0, 0,
    1, 2, 0, 3, 0, 1, 0, 0,
    1, 1, 1, 1, 0, 1, 0, 0,
    0, 0, 1, 0, 0, 1, 1, 0,
    0, 0, 1, 0, 0, 4, 1, 0,
    0, 0, 1, 1, 1, 1, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
  ],
  [
    0, 0, 0, 0, 1, 1, 1, 0,
    0, 1, 1, 1, 1, 0, 1, 1,
    0, 1, 0, 0, 0, 4, 0, 1,
    1, 1, 0, 1, 1, 4, 0, 1,
    1, 2, 0, 3, 0, 0, 0, 1,
    1, 1, 1, 3, 0, 0, 0, 1,
    0, 0, 1, 0, 0, 1, 1, 1,
    0, 0, 1, 1, 1, 1, 0, 0,
  ],
]

# ステージ情報参照メソッド
def getStageData(x, y)
  return $stageData[$stage_no][y*8+x]
end

# ステージ生成メソッド
def createStage()
  for y in 0..7
    for x in 0..7
      case getStageData(x, y)
      when $DT_FLOOR
        # 床を描画
        drawFloor(x, y)
      when $DT_WALL
        # 壁を描画
        drawWall(x, y)
      when $DT_PLAYER
        # プレイヤーキャラクタを描画
        $player.setPos(x, y)
      when $DT_BLOCK
        # ブロックを生成して描画
        $blockList.push(DisplayObject.new(x, y, "block.gbf"))
      when $DT_GOAL
        # ゴールを描画
        drawGoal(x, y)
      end
    end
  end
end
```

ステージ 1 のデータ

ステージ 2 のデータ

現在のステージ番号に対応するステージデータ配列の x, y 座標で指定した位置の数値を取得します。

ステージデータを参照します

ステージデータが床の場合  
床を描画

ステージデータが壁の場合  
壁を描画

ステージデータがプレイヤーキャラクタの場合  
プレイヤーキャラクタの位置を設定

ステージデータがブロックの場合  
ブロックのインスタンスを生成して配列に追加

ステージデータがゴールの場合  
ゴールのインスタンスを生成して配列に追加

メソッド createStage() の中でステージデータを生成しています。

ステージデータは配列 \$stageData に格納しています。\$stageData に格納されている数値とステージデータの対応は下記となります。

- 0 : 床
- 1 : 壁
- 2 : プレイヤーキャラクタ
- 3 : ブロック
- 4 : ゴール

ステージの大きさは縦 8 × 横 8 です。メソッド createStage() の中で 8 × 8 の 2 重 for ループ処理を行いつつステージデータを読み込み、それぞれのステージデータに対応した処理を行います。

#### (4) ゲームプレイ中状態の処理

ゲームプレイ中状態の場合の処理を説明します。

```
# デジタルパッド操作時処理
#-----
def onPad(param1, param2)
  (中略)

  if $state == $ST_PLAY then
    # ゲームプレイ中状態の場合はプレイヤーの移動を行う
    dx = 0          # プレイヤーキャラクタ x 移動量
    dy = 0          # プレイヤーキャラクタ y 移動量
    isBlockMove = 1  # ブロック移動判定フラグ
    isStageClear = 1 # ステージクリア判定フラグ

    # デジタルパッド入力に対する移動処理
    case param1
    when $DPAD_UP
      dy = -1
    when $DPAD_DOWN
      dy = 1
    when $DPAD_LEFT
      dx = -1
    when $DPAD_RIGHT
      dx = 1
    end

    # プレイヤーの移動先にブロックがあるかどうか
    block = isBlockExist($player.posX + dx, $player.posY + dy)
    if block != nil then
      # ブロックの移動先の算出
      blockMoveX = $player.posX + dx * 2
      blockMoveY = $player.posY + dy * 2

      # ブロックの移動先に何かあるか
      if (isBlockExist(blockMoveX, blockMoveY) == nil) &&
        (getStageData(blockMoveX, blockMoveY) != $DT_WALL) then
        # ブロックの移動先にブロックも壁もなければ移動可能
        block.moveTo(blockMoveX, blockMoveY)
      else
        # ブロックが移動できない
        isBlockMove = 0
      end
    end
  end

  # ゴール判定
  $blockList.each { |block|
    if getStageData(block.posX, block.posY) != $DT_GOAL then
      # ブロックがゴールの上に無ければステージクリア判定フラグを 0 にする
      isStageClear = 0
    end
  }

  # プレイヤーの移動
  if (getStageData($player.posX + dx, $player.posY + dy) != $DT_WALL) &&
    (isBlockMove == 1) then
    $player.moveTo($player.posX + dx, $player.posY + dy)
  end

  # クリア判定
  if isStageClear == 1 then
    GNDrawImage(80, 70, "clear.gbf")
    $state = $ST_CLEAR
  end
end
```

①デジタルパッド入力に従ってプレイヤーキャラクタの移動方向を決めます。

②プレイヤーキャラクタの移動先にブロックがあるかどうか判定します。移動先にブロックがある場合は、そのブロックも移動させます。

③配列に入ったブロックを順番に判定を行い、一つでもゴールに到達していなければゴール判定フラグを 0 にします。

④プレイヤーを移動させます。

⑤全てのブロックがゴールに到達していれば、ゲーム状態をステージクリアに切替えます。

```

        end
    end
end

```

ゲームプレイ中状態の処理は全てデジタルパッド操作時の処理の中に含まれます。

①デジタルパッドの入力値に従ってプレイヤーキャラクタの移動量を決めます。

現在位置に対する  $x, y$  座標の増加減少分をそれぞれ  $dx, dy$  という変数に設定します。

②プレイヤーキャラクタの移動先にブロックがあるかどうかを判定します。

現在のプレイヤーキャラクタの座標に①で求めた  $dx$  と  $dy$  を加えた位置がプレイヤーキャラクタの移動先の座標になるため、そこにブロックが存在するかどうかを判定しています。

ブロックが存在する場合はそのブロックも移動させる必要がありますが、さらにその先にブロックまたは壁が存在する場合は移動できないのでその判定を行います。

現在のプレイヤーキャラクタの座標に①で求めた  $dx$  と  $dy$  を2倍したものを加えた位置がブロックの移動先の座標になります。その座標に対してブロックまたは壁が存在するかどうかの判定を行います。

③ゴールの判定を行います。

あらかじめゴール判定フラグを1に設定しておき、配列に入っているブロックを順番に判定してゴールに到達していないブロックがあればゴール判定フラグを0に変更します。すなわち、全てのブロックがゴールに到達しているときのみゴール判定フラグは1のままとなります。

④プレイヤーキャラクタの移動先に壁もブロックも存在しなければ移動します。

⑤全てのブロックがゴールに到達していれば、ゲーム状態をステージクリア状態に切換えます。

## (5) ステージクリア状態の処理

ステージクリア状態の場合の処理を説明します。

```
# デジタルパッド操作時処理
#-----
def onPad(param1, param2)
  # ステージクリア状態の場合はパッド操作で次のステージへ
  if $state == $ST_CLEAR then
    $stage_no += 1
    if $stage_no >= $totalStage then
      $stage_no = 0
    end
  end

  # 次のステージの生成を開始する
  $state = $ST_CREATE

  return
end

(中略)

end
```

ステージクリア状態の処理は全てデジタルパッド操作時の処理の中に含まれます。

デジタルパッドを操作したときに、ゲーム状態がステージクリア状態ならば上記の処理を行います。処理内容としては、ステージ番号を1つ増加させます。ステージ番号がステージ総数を越えた場合はステージ番号を0に戻します。

これらの処理が終わったら、ゲーム状態をステージ生成状態に切替えます。

## (6) その他のメソッド

setup メソッド

```

#-----
# メイン初期化
#=====
def setup()
  # 状態初期化
  $state = $ST_CREATE

  # ステージ番号初期化
  $stage_no = 0

  # ブロック配列生成
  $blockList = []

  # デジタルパッド生成
  $digitalpad = GNDigitalPad.new(50, 185, "onPad")

  # プレイヤー生成
  $player = DisplayObject.new(0, 0, "player.gbf")
end

```

setup メソッドでは各種初期化処理を行います。

ゲーム状態をステージ生成状態で初期化して、ステージ番号を 0 で初期化します。

ブロックは各ステージで存在する数が変化するため配列を使って管理します。初期化処理ではブロックを管理するための配列を中身が空の状態で作成します。

また、デジタルパッドとプレイヤーキャラクタは常に 1 つだけ存在するものですので、初期化処理の中でインスタンスを 1 つだけ生成しておきます。

画面描画メソッド

```

# 壁描画メソッド
def drawWall(x, y)
  GNDrawImage(x * $UNIT_SCALE, y * $UNIT_SCALE, "wall.gbf")
end

# 床描画メソッド
def drawFloor(x, y)
  GNDrawImage(x * $UNIT_SCALE, y * $UNIT_SCALE, "floor.gbf")
end

# ゴール描画メソッド
def drawGoal(x, y)
  GNDrawImage(x * $UNIT_SCALE, y * $UNIT_SCALE, "goal.gbf")
end

```

壁、床、ゴールを画面上に描画するメソッドをそれぞれ用意します。`$UNIT_SCALE` は画面上の表示要素の大きさを定義した変数です。今回はゲーム画面の表示画像の大きさが  $20 \times 20$  なので `$UNIT_SCALE` の値も 20 を設定します。

ゲーム内の座標(x, y)をそれぞれ 20 倍した位置に画像を描画することで、画面上の壁や床等を隙間なく表示します。

### ブロック有無判定メソッド

```
# ブロック有無判定メソッド
def isBlockExist(x, y)
  $blockList.each { |block|
    if (x == block.posX) && (y == block.posY) then
      return block
    end
  }
  return nil
end
```

指定した(x, y)座標にブロックが存在するかどうかを判定します。

ブロックは配列に格納して管理しているため、配列に含まれる全てのブロックの座標をチェックして指定した座標と一致するブロックが存在する場合はメソッドの戻り値としてそのブロックを返します。

指定した座標と一致するブロックが一つも存在しない場合はメソッドの戻り値として `nil` を返します。

`nil` とは、値を持たないという意味です。

## 第3章 プログラム作成：応用編

---

本章では、付属のマイコンボードと色々な電子部品を組み合わせで動作するプログラムの例を説明します。必要な電子部品の一覧も記載しますので、別途それらの部品を購入いただければ実際に動作させることができます。

## 3.1 本章の使い方

本章では、マイコンボードの各端子と色々な電子部品を接続して電子回路を作成する方法を説明します。

色々な電子回路について、必要な部品一覧と接続方法、プログラムソースコードを記載しますので気に入ったものがあれば実際に作ってみてください。

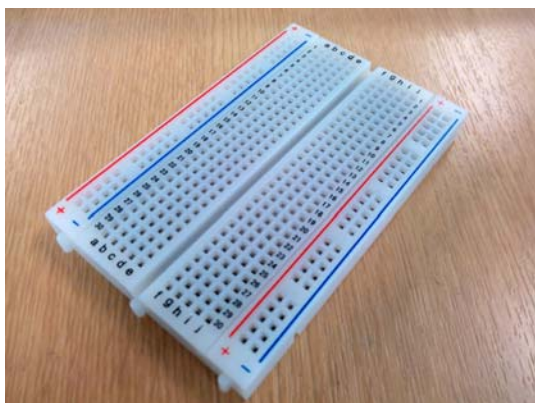
また、それらを応用して自分だけの電子回路を作って動かしてみましょう。

## 3.2 ブレッドボードの使い方

### 3.2.1 ブレッドボードとは

ここでは手軽に電子回路を作るためにブレッドボードというものを 사용합니다。

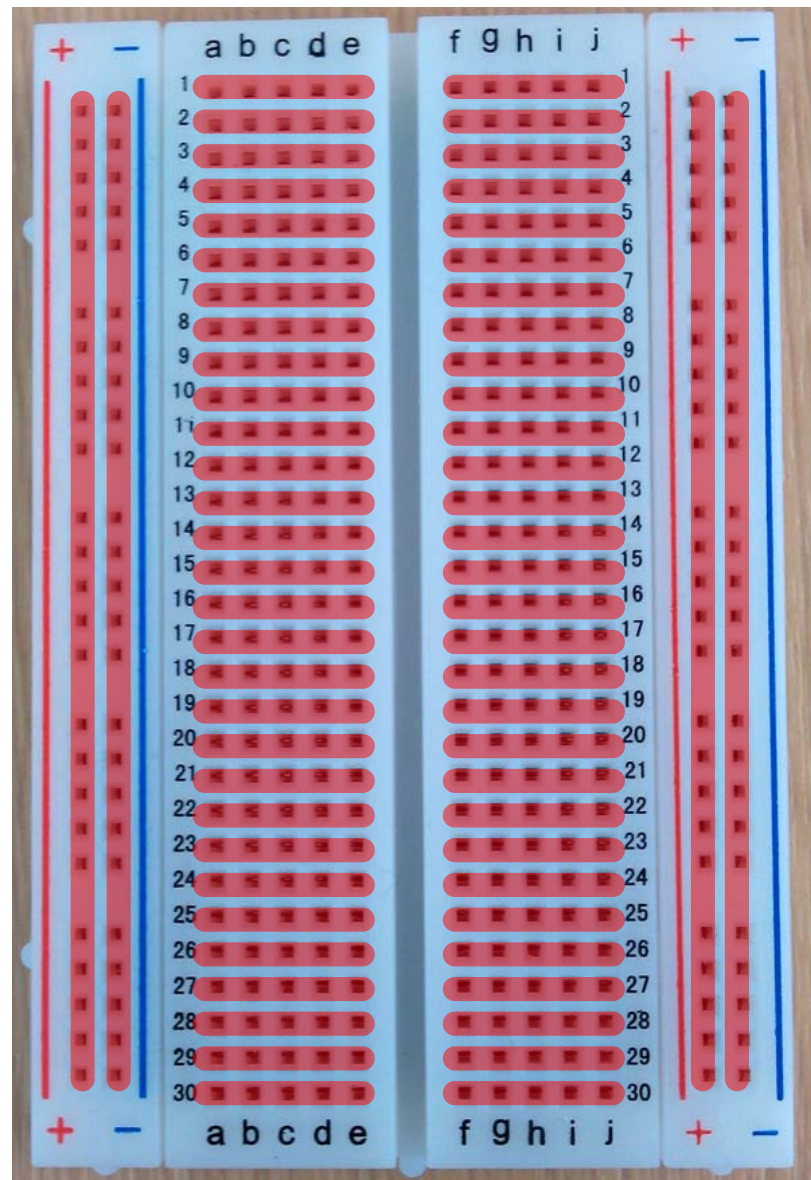
ブレッドボードとは、たくさんの穴があいたボードです。電子部品やリード線を直接挿し込むことによって電子回路を作ることができるので、ハンダ付けが不要です。



表面にある穴はブレッドボード内部で電氣的に接続されています。規則的に接続されているので、それを考慮した上で電子部品を配置することで電子回路を作成することができます。

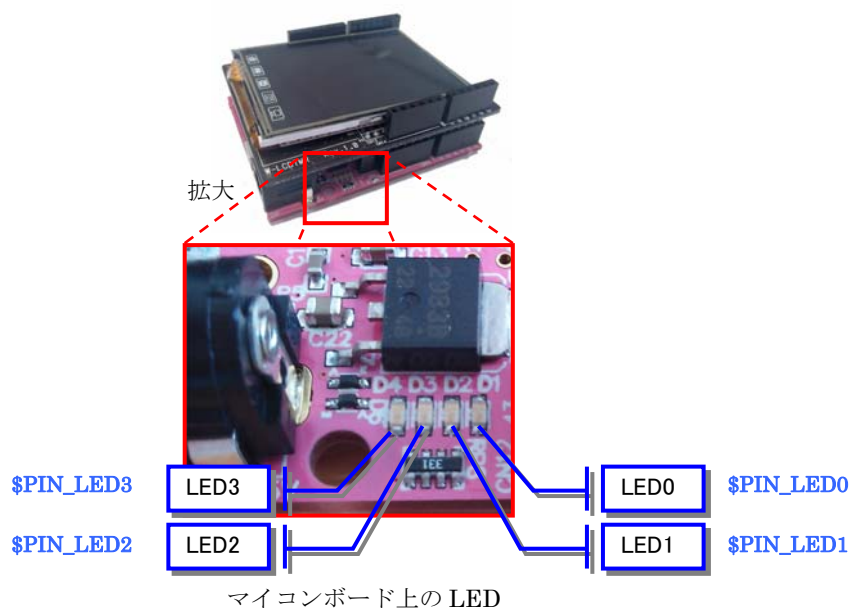
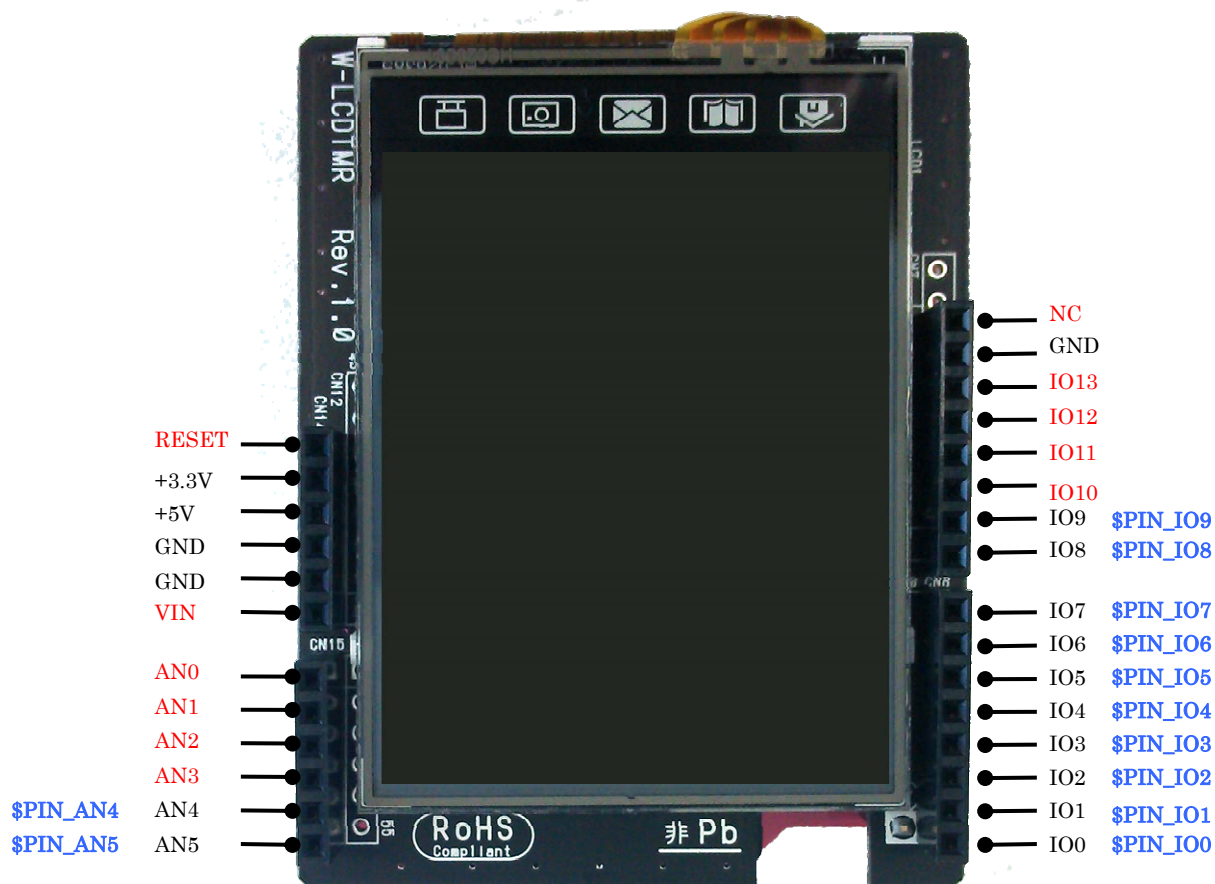


下図の赤い線が内部で電氣的に接続されています。



### 3.3 マイコンボードの端子と変数名の対応

入出力端子の詳細を以下に示します。赤字で表示している端子は使用しません。  
プログラムから使用するときには青い文字のグローバル変数名で端子を指定します。



| 端子名称      | 用途                              |
|-----------|---------------------------------|
| RESET     | 使いません                           |
| +3.3V     | 3.3V の電源が得られます                  |
| +5V       | 5V の電源が得られます                    |
| GND       | グラウンドとなります                      |
| VIN       | 使いません                           |
| AN0～AN5   | アナログ入力端子です。AN0～AN3 は使用できません。    |
| IO0～IO13  | デジタル入出力端子です。IO10～IO13 は使用できません。 |
| NC        | 使いません                           |
| LED0～LED3 | LED です。点灯させることができます。            |

## 3.4 マイコンボードの端子の使い方

マイコンボード上の端子 I00～I09 および AN4～AN5 をプログラムから使うことができます。

### (1) デジタル入出力端子

I00～I09 の 10 本の端子はデジタル入出力端子です。設定によって入力にも出力にも使用することができます。

また、出力を行う場合はデジタル(On/Off)の出力と PWM 制御によるアナログ出力を行うことができます。

デジタル入出力端子に関して使用可能なメソッドの一覧は下記となります。

| 名前      | メソッド名           | 機能  |
|---------|-----------------|---|
| 入出力切換え  | gr_pinMode      | 指定した端子の入出力動作を設定します。                       |
| デジタル入力  | gr_digitalRead  | 端子からデジタル値を読み込みます。                         |
| デジタル出力  | gr_digitalWrite | 端子へデジタル値を出力します。                           |
| アナログ出力  | gr_analogWrite  | 端子へ PWM 制御によるアナログ値を出力します。                 |
| 矩形波出力   | gr_tone         | 端子へ指定周波数の矩形波を出力します。<br>圧電ブザーで音を鳴らすのに便利です。 |
| 矩形波出力停止 | gr_noTone       | 矩形波出力を停止します。                              |

## ■入出力切換え

デジタル入出力端子を使う前に、その端子を入力として使用するか出力として使用するかを指定する必要があります。

### 入出力切換えメソッドの書き方

```
gr_pinMode(端子番号, 入出力モード)
```

端子番号 : 端子番号(端子の対応は3.3項の図を参照してください)  
 入出力モード : 入力の場合 : \$INPUT  
                   出力の場合 : \$OUTPUT

例) 端子 IO4 を出力モードで使用する場合

```
gr_pinMode($PIN_IO4, $OUTPUT)
```

端子 IO3 を入力モードで使用する場合

```
gr_pinMode($PIN_IO3, $INPUT)
```

LED0 を出力モードで使用する場合(LED を入力モードで使用することはできません)

```
gr_pinMode($PIN_LED0, $OUTPUT)
```

## ■デジタル入力

入出力モードを入力に設定した端子からの入力値を取得します。

### デジタル入力メソッドの書き方

```
gr_digitalRead(端子番号)
```

端子番号 : 端子番号(端子の対応は3.3項の図を参照してください)  
 戻り値 : HIGH の場合 : 1  
           LOW の場合 : 0

例) 端子 IO3 の入力値を取得する場合

```
value = gr_digitalRead($PIN_IO3)
```

変数 value に入力値が格納されます。

## ■デジタル出力

入出力モードを出力に設定した端子に対して、デジタル出力を行うことができます。

### デジタル出力メソッドの書き方

```
gr_digitalWrite(端子番号, 出力値)
```

端子番号 : 端子番号(端子の対応は 3.3 項の図を参照してください)

出力値 : HIGH の場合 : 1  
LOW の場合 : 0

例) 端子 IO4 の出力を HIGH にする場合

```
gr_digitalWrite($PIN_IO4, 1)
```

LED0 を点灯する場合

```
gr_digitalWrite($PIN_LED0, 1)
```

## ■アナログ出力

入出力モードを出力に設定した端子に対して、PWM 制御によるアナログ出力を行うことができます。

### アナログ出力メソッドの書き方

```
gr_analogWrite(端子番号, 出力値)
```

端子番号 : 端子番号(端子の対応は 3.3 項の図を参照してください)

出力値 : 0~255 の範囲で設定します

例) 端子 IO4 の出力を 50%にする場合

```
gr_analogWrite($PIN_IO4, 127)
```

LED0 の出力を 50%にする場合

```
gr_analogWrite($PIN_LED0, 127)
```

## ■矩形波出力

入出力モードを出力に設定した端子に対して矩形波を出力することができます。端子に圧電ブザー等を接続して音を鳴らす場合に便利です。

### 矩形波出力メソッドの書き方

```
gr_tone(端子番号, 周波数, 時間)
```

|      |                    |
|------|--------------------|
| 端子番号 | : 矩形波を出力する端子の番号    |
| 周波数  | : 矩形波の周波数[Hz]      |
| 時間   | : 音を鳴らす時間[ms]      |
|      | 0を指定すると無制限に鳴り続けます。 |

例) I08 番端子に接続した圧電ブザーで 300[Hz]の音を 500[ms]鳴らす場合

```
gr_tone($PIN_I08, 300, 500)
```

## ■矩形波出力停止

gr\_tone メソッドを使って出力した矩形波を停止します。

### 矩形波出力停止メソッドの書き方

```
gr_noTone(端子番号)
```

|      |                   |
|------|-------------------|
| 端子番号 | : 矩形波出力を停止する端子の番号 |
|------|-------------------|

例) I08 番端子に接続した圧電ブザーの音を止める場合

```
gr_noTone($PIN_I08)
```

(2) アナログ入力端子

AN4～AN5 の 2 本の端子はアナログ入力端子です。

アナログ入力端子に関して使用可能なメソッドの一覧は下記となります。

| 名前     | メソッド名         | 機能                |
|--------|---------------|-------------------|
| アナログ入力 | gr_analogRead | 端子からアナログ値を読み込みます。 |

■ アナログ入力

アナログ入力端子からの入力電圧値を取得します。

アナログ入力メソッドの書き方

gr\_analogRead(端子番号)  
端子番号 : 端子番号 (端子の対応は 3.3 項の図を参照してください)  
戻り値 : 0～675  
入力可能な電圧値は最大 3.3V です。電圧値が 3.3V のときに戻り値は 675 となります。

例) 端子 AN5 の入力を取得する場合

```
value = gr_analogRead($PIN_AN5)
```

変数 value に入力値が格納されます。



## 3.5 LEDを点滅させる

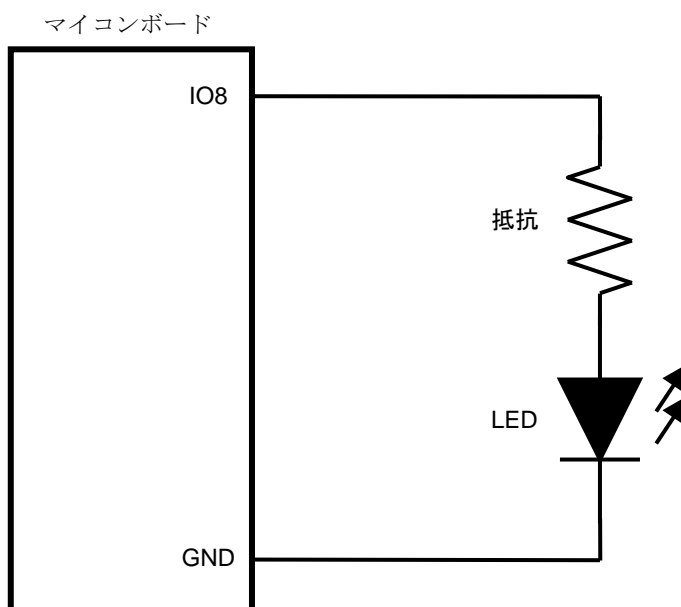
実際にブレッドボードを使って簡単な電子回路を作ってみます。

マイコンボードを使って最初に作った LED 点滅プログラムを応用して、ブレッドボード上に配置した LED を点滅させます。

必要な部品一覧

- ・ブレッドボード 1 個
- ・LED 1 個
- ・抵抗 (220～470  $\Omega$  程度) 1 個
- ・ジャンプワイヤ線 2 本

### (1) 回路図

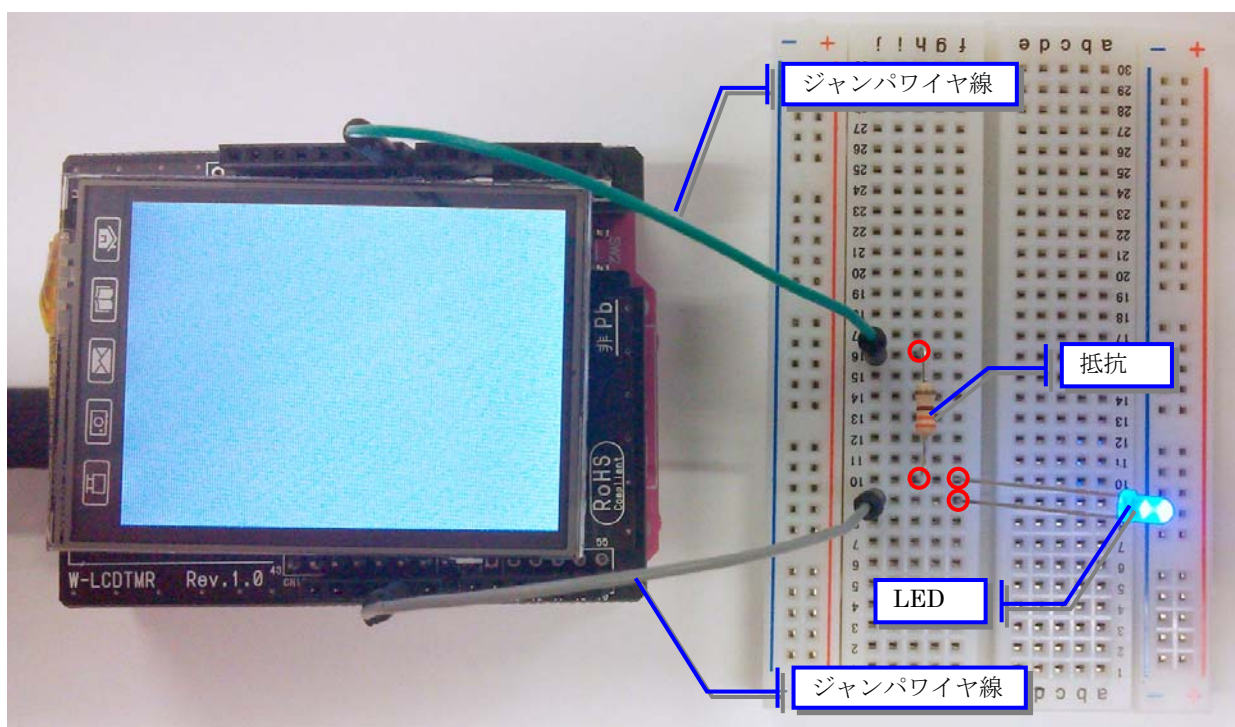


マイコンボードの IO8 番ピンの出力を On/Off 切り換えを行って LED を点滅させます。

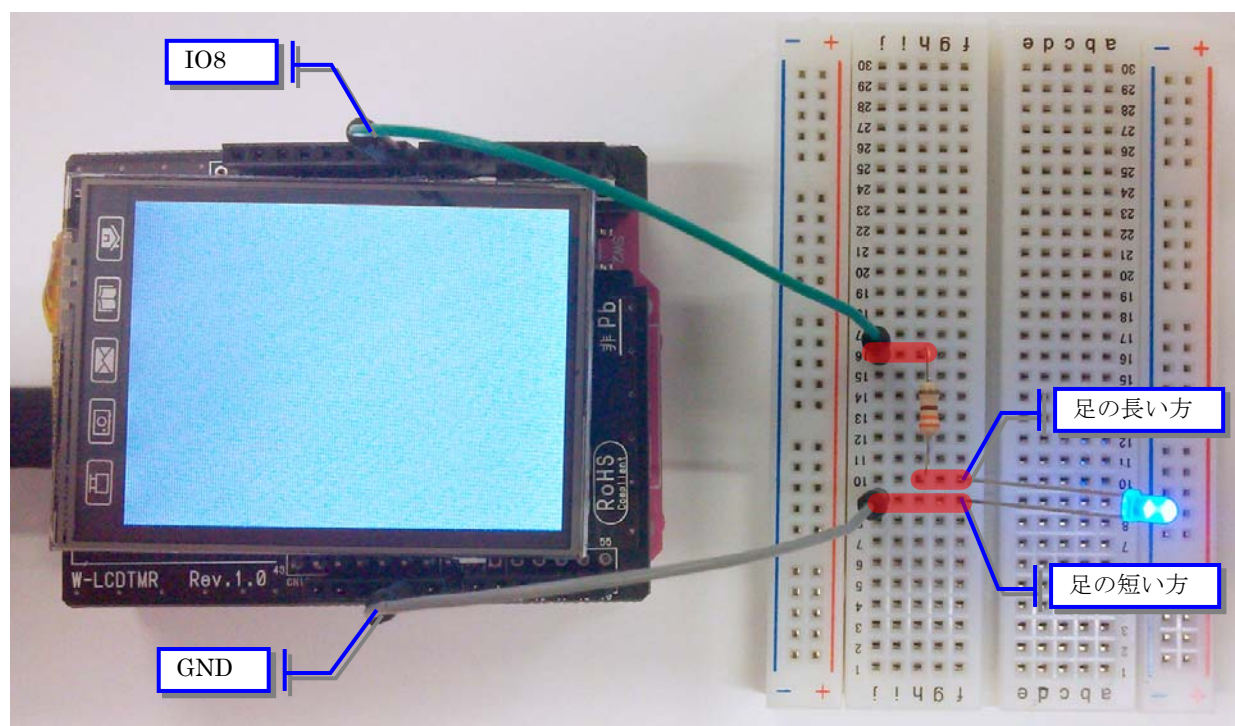
マイコンボードのピン出力を直接 LED に接続すると電圧が高すぎて LED が壊れる可能性があるため、抵抗を間に接続します。

## (2) 接続図

赤丸部分に抵抗と LED の足を差し込んでいます。



赤い線の部分がブレッドボード内で接続されています。  
LED の 2 本の足は長い方を+電源(PIN8)側に接続してください。



### (3) プログラムの作成

下記のプログラムを作成して実行してください。

同じ内容のプログラムファイルがインストールフォルダ内の **sample¥sample07** フォルダに **"led.rb"** という名前で格納されています。

```
# メイン初期化
#-----
def setup()
  # IO8 を出力設定とする
  gr_pinMode($PIN_IO8, $OUTPUT)
end

# メインループ
#-----
def loop()
  # IO8 の出力を On とする
  gr_digitalWrite($PIN_IO8, 1)
  gr_delay(100)

  # IO8 の出力を Off とする
  gr_digitalWrite($PIN_IO8, 0)
  gr_delay(100)
end
```

プログラムの動作内容は第1章で作成した最初のプログラムと同様となります。

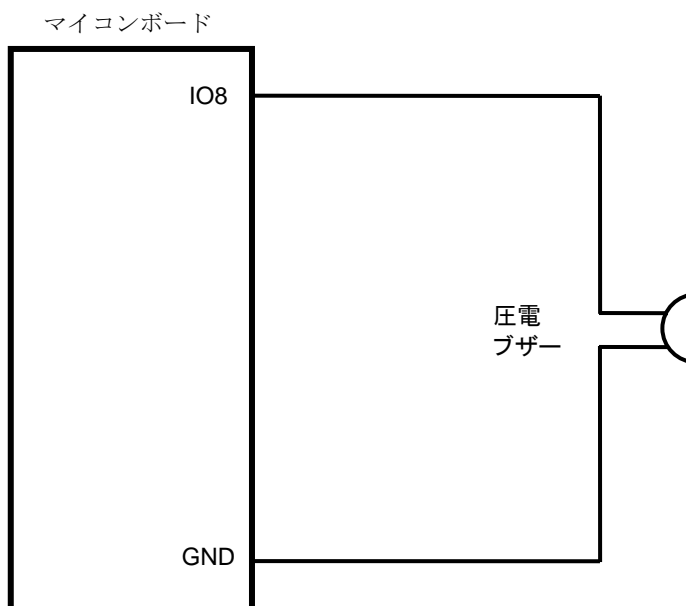
## 3.6 電子ピアノを作る

本項では、圧電スピーカーを接続して音を鳴らす例を説明します。LCD 画面上のボタンを押すとボタンに対応した音階の音が鳴ります。

必要な部品一覧

- |           |     |
|-----------|-----|
| ・ブレッドボード  | 1 個 |
| ・圧電ブザー    | 1 個 |
| ・ジャンプワイヤ線 | 2 本 |

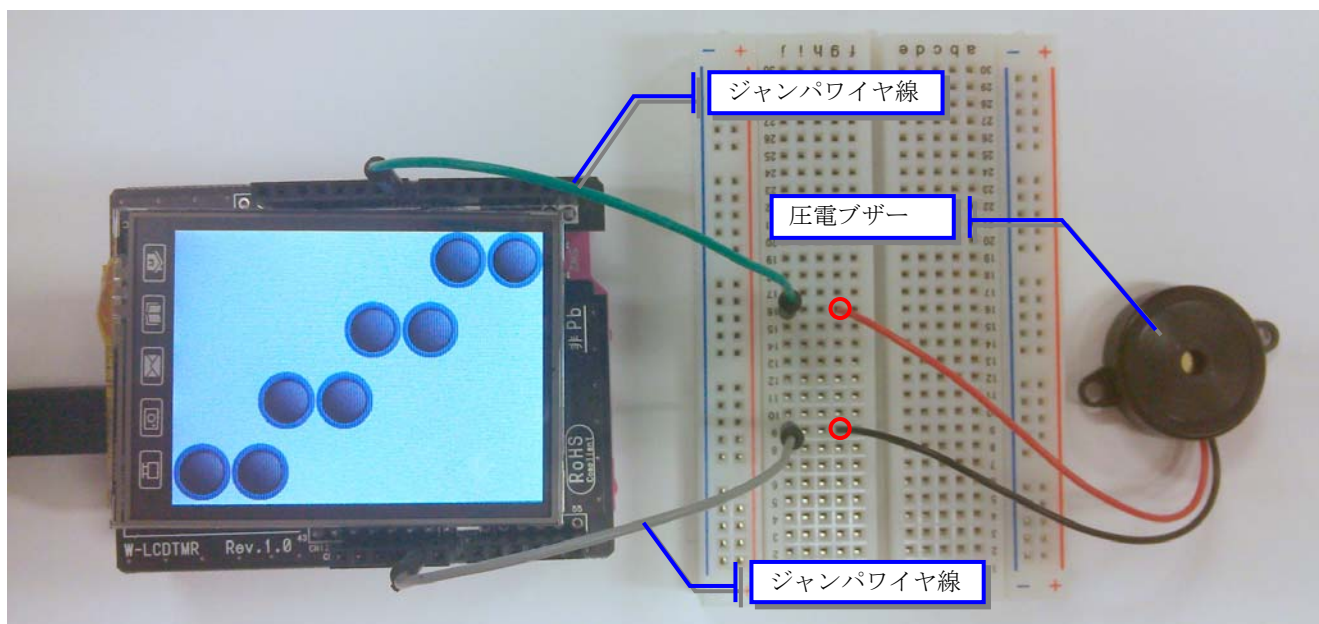
### (1) 回路図



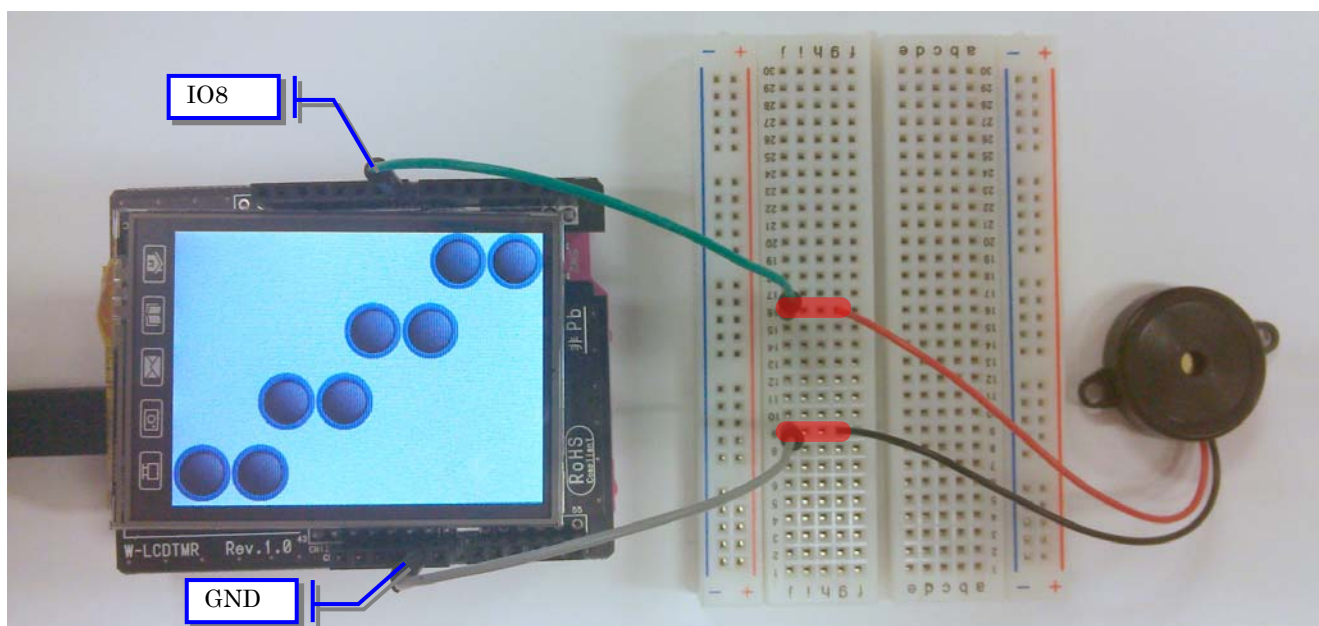
マイコンボードの IO8 番ピンの出力を PWM によるアナログ出力を行って圧電ブザーを鳴らします。液晶画面に表示したボタンによって圧電ブザーを鳴らす周波数を決めます。

## (2) 接続図

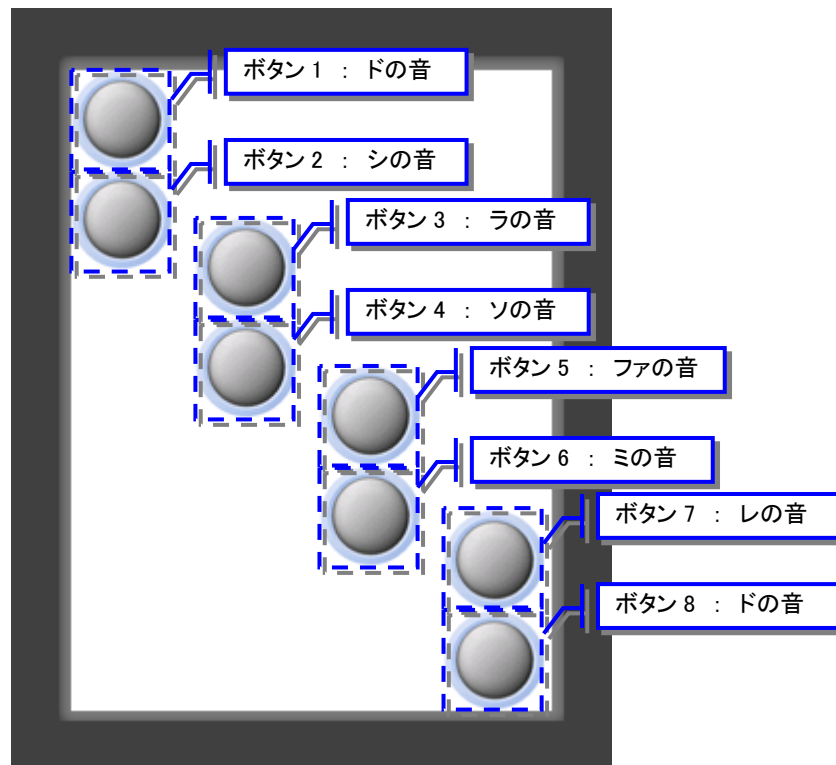
赤丸部分に圧電ブザーの線を差し込んでいます。



赤い線の部分がブレッドボード内で接続されています。



### (3) 液晶画面表示



#### (4) プログラムの作成

下記のプログラムを作成して実行してください。

同じ内容のプログラムファイルがインストールフォルダ内の **sample¥sample08** フォルダに **"piano.rb"** という名前で格納されています。

```
# メイン初期化
#-----
def setup()
  $button1 = GNButton.new( 0, 0, "onTouch1")
  $button2 = GNButton.new( 0, 50, "onTouch2")
  $button3 = GNButton.new( 62, 74, "onTouch3")
  $button4 = GNButton.new( 62, 124, "onTouch4")
  $button5 = GNButton.new(124, 148, "onTouch5")
  $button6 = GNButton.new(124, 198, "onTouch6")
  $button7 = GNButton.new(186, 220, "onTouch7")
  $button8 = GNButton.new(186, 270, "onTouch8")

  gr_pinMode($PIN_IO8, $OUTPUT)
end

# メインループ
#-----
def loop()
end

def onTouch1(param1, param2)
  if param1 == 1 then
    # ドの音
    gr_tone($PIN_IO8, 1046, 0)
  else
    gr_noTone($PIN_IO8)
  end
end

def onTouch2(param1, param2)
  if param1 == 1 then
    # シの音
    gr_tone($PIN_IO8, 986, 0)
  else
    gr_noTone($PIN_IO8)
  end
end

def onTouch3(param1, param2)
  if param1 == 1 then
    # ラの音
    gr_tone($PIN_IO8, 880, 0)
  else
    gr_noTone($PIN_IO8)
  end
end

def onTouch4(param1, param2)
  if param1 == 1 then
    # ソの音
    gr_tone($PIN_IO8, 783, 0)
  else
    gr_noTone($PIN_IO8)
  end
end

def onTouch5(param1, param2)
  if param1 == 1 then
    # ファの音
    gr_tone($PIN_IO8, 698, 0)
  else
    gr_noTone($PIN_IO8)
  end
end
```

```
def onTouch6(param1, param2)
  if param1 == 1 then
    # ミの音
    gr_tone($PIN_IO8, 659, 0)
  else
    gr_noTone($PIN_IO8)
  end
end

def onTouch7(param1, param2)
  if param1 == 1 then
    # レの音
    gr_tone($PIN_IO8, 587, 0)
  else
    gr_noTone($PIN_IO8)
  end
end

def onTouch8(param1, param2)
  if param1 == 1 then
    # ドの音
    gr_tone($PIN_IO8, 523, 0)
  else
    gr_noTone($PIN_IO8)
  end
end
```



### (5) 圧電ブザーを鳴らす

画面上に8つのボタンを配置して、それぞれのボタンを押したときにドレミの音階を鳴らします。圧電ブザーを鳴らす場合は `gr_tone` というメソッドを使うと簡単に指定した音階を出すことができます。音を止める場合には `gr_noTone` というメソッドを使います。

音を鳴らすメソッドの使い方については、3.4(1)項を参照してください。

## 3.7 電子オルゴールを作る

本項では、光センサを使って明るさを検知する例を説明します。ここでは光センサとして CdS と呼ばれるセンサを使います。

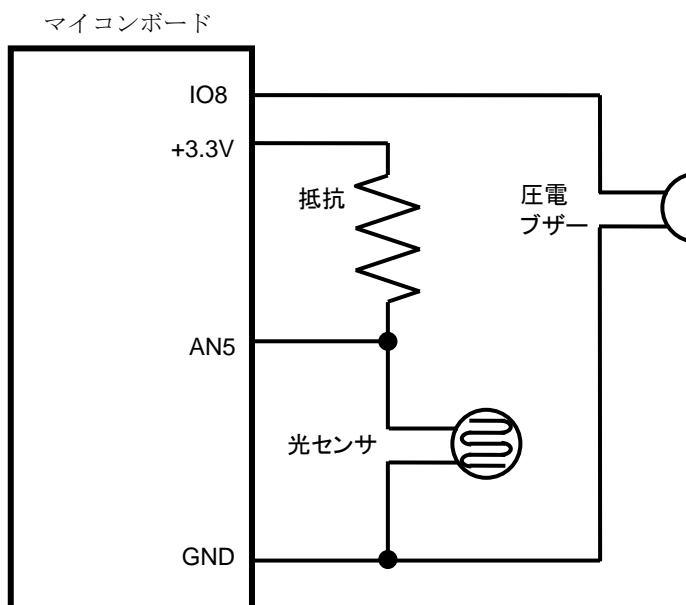
CdS とは周囲の明るさによって抵抗値が変化する素子です。抵抗値の変化に伴って変化する電圧の値をマイコンボードで読み取ることによって周囲の明るさを知ることができます。

それを応用して、周囲が明るくなったら音楽を鳴らして周囲が暗くなったら音楽を止めるオルゴールを作ってみます。

必要な部品一覧

|                      |     |
|----------------------|-----|
| ・ブレッドボード             | 1 個 |
| ・圧電ブザー               | 1 個 |
| ・抵抗(10K $\Omega$ 程度) | 1 個 |
| ・CdS(光センサ)           | 1 個 |
| ・ジャンプワイヤ線            | 6 本 |

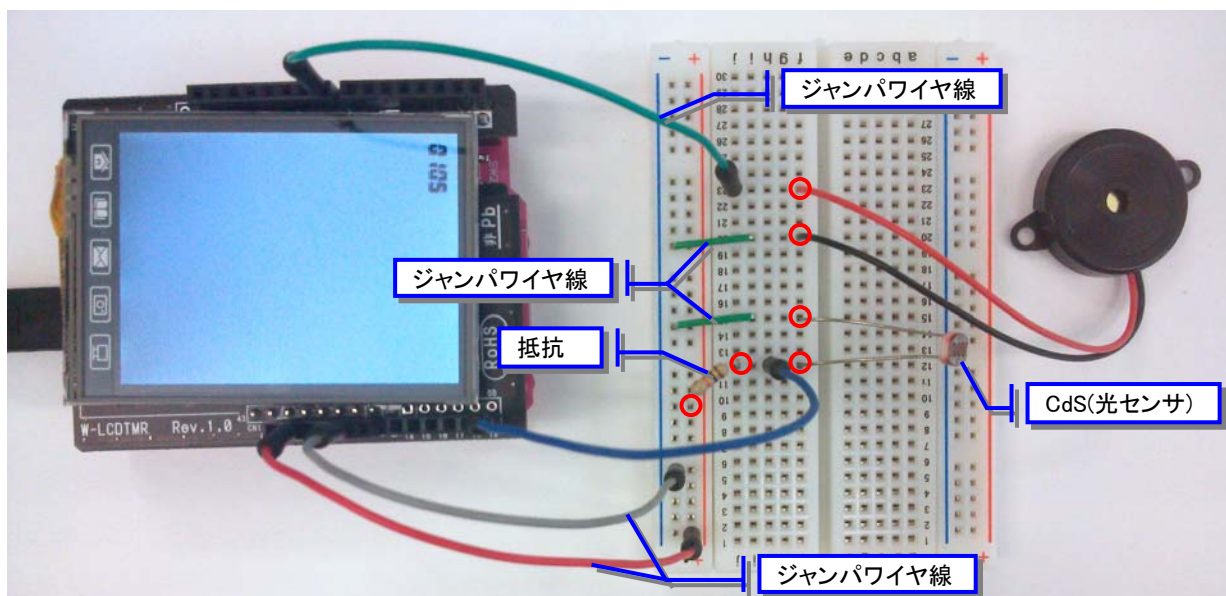
### (1) 回路図



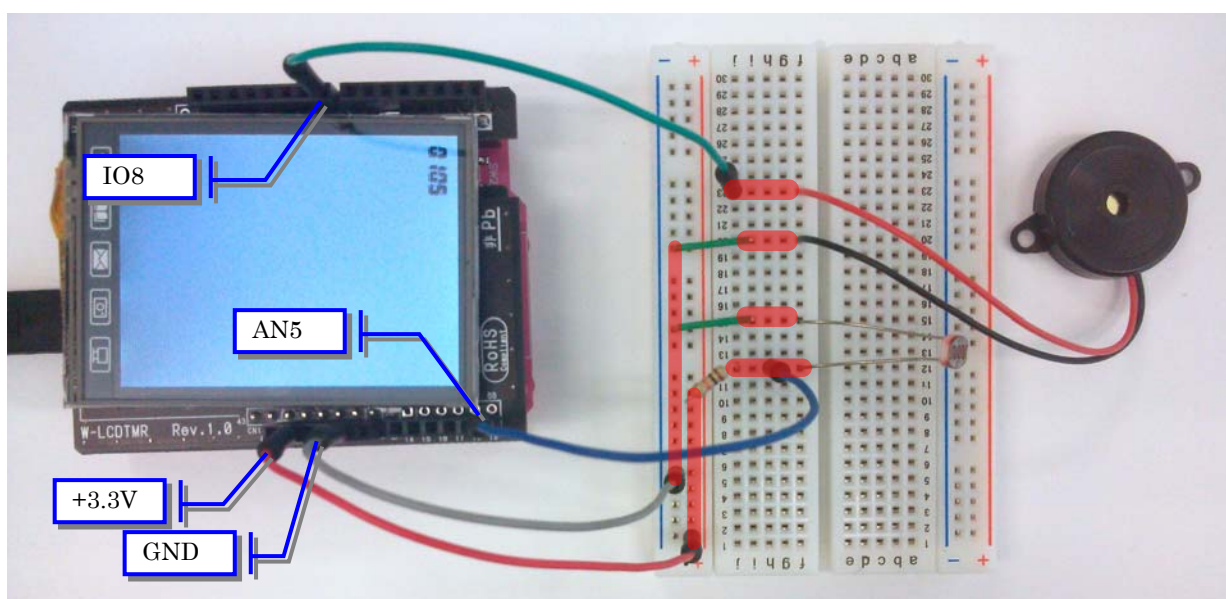
CdS は明るさによって抵抗値が変化しますが、マイコンボードの入力ポート(上記 AN5)が検出できるのは電圧値の変化です。抵抗値の変化を電圧値に変換するために、固定抵抗と CdS の直列回路を作成しています。

(2) 接続図

赤丸部分に抵抗と圧電ブザーと光センサの足を差し込んでいます。



赤い線の部分がブレットボード内で接続されています。



### (3) 液晶画面表示



#### (4) プログラムの作成

下記のプログラムを作成して実行してください。

同じ内容のプログラムファイルがインストールフォルダ内の **sample¥sample09** フォルダに **"musicbox.rb"** という名前で格納されています。

```
# 状態の定義
$ST_STOP = 0
$ST_PLAY = 1

# 1:ド 2:レ 3:ミ 4:ファ 5:ソ 6:ラ 7:シ 8:ド
$musicData = [
  1, 1, 5, 5, 6, 6, 5, 0, 4, 4, 3, 3, 2, 2, 1, 0
]
$musicLength = $musicData.size

# 再生と停止を切替える値(周囲の明るさによって調整してください)
$light_val = 150

# メイン初期化
#-----
def setup()
  # 状態の初期化
  $state = $ST_STOP

  # カウンタの初期化
  $tic = 0
  $playCount = 0

  # 明るさ表示部品の生成
  $numbmp1 = GNDigitalNum.new(10, 10)

  gr_pinMode($PIN_IO8, $OUTPUT)
end

# メインループ
#-----
def loop()
  # 明るさの取得
  light = gr_analogRead($PIN_AN5)
  $numbmp1.setInteger(light)

  case $state
  when $ST_STOP
    # 停止状態の場合
    if light < $light_val then
      $state = $ST_PLAY
      $tic = 0
      $playCount = 0
    end
  when $ST_PLAY
    # 再生状態の場合
    if light >= $light_val then
      $state = $ST_STOP
    end
  end

  # 再生状態ならばメロディの再生を行います
  if $state == $ST_PLAY then
    $tic += 1
    if $tic >= 300 then
      $tic = 0

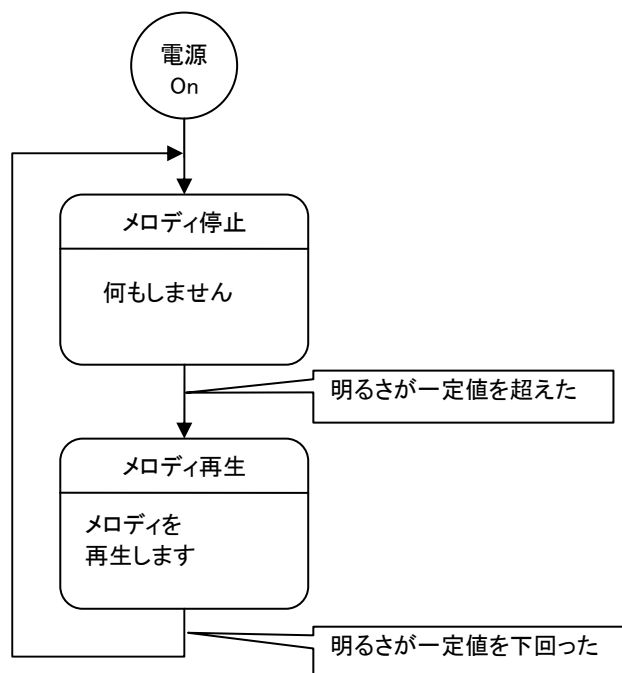
      playMelody($musicData[$playCount])
      $playCount += 1
      if $playCount >= $musicLength then
        $playCount = 0
      end
    end
  end
end
```

```
end

# 音階再生
def playMelody( tone )
  case tone
  when 0
    # 音を止める
    gr_noTone($PIN_IO8)
  when 1
    # ドの音
    gr_tone($PIN_IO8, 523, 500)
  when 2
    # レの音
    gr_tone($PIN_IO8, 587, 500)
  when 3
    # ミの音
    gr_tone($PIN_IO8, 659, 500)
  when 4
    # ファの音
    gr_tone($PIN_IO8, 698, 500)
  when 5
    # ソの音
    gr_tone($PIN_IO8, 783, 500)
  when 6
    # ラの音
    gr_tone($PIN_IO8, 880, 500)
  when 7
    # シの音
    gr_tone($PIN_IO8, 986, 500)
  when 8
    # ドの音
    gr_tone($PIN_IO8, 1046, 500)
  end
end
```

## (5) 状態遷移

周囲の明るさによって状態遷移を行います。



プログラムの中ではゲーム状態を以下の変数に格納して管理します。

`$state`

ゲーム状態の値は以下の変数を使います。

`$ST_STOP = 0`      メロディ停止状態

`$ST_PLAY = 1`      メロディ再生状態

## (6)明るさの取得と判定

明るさの取得はメインループで行っています。

```

# 再生と停止を切替える値(周囲の明るさによって調整してください)
$light_val = 150

(中略)

# メインループ
#-----
def loop()
  # 明るさの取得
  light = gr_analogRead($PIN_AN5)
  $numbmp1.setInteger(light)

  case $state
  when $ST_STOP
    # 停止状態の場合
    if light < $light_val then
      $state = $ST_PLAY
      $tic = 0
      $playCount = 0
    end
  when $ST_PLAY
    # 再生状態の場合
    if light >= $light_val then
      $state = $ST_STOP
    end
  end
end

(中略)

end

```

電圧値を端子 AN5 から取得して、取得した値を画面に表示します。

メロディ停止状態の時に、電圧値が一定値を下回った(明るくなった)ならば、状態をメロディ再生状態に切替えます。

メロディ再生状態の時に、電圧値が一定値を上回った(暗くなった)ならば、状態をメロディ停止状態に切替えます。

状態を切替えるための判定値は\$light\_val という変数に入っています。この値を境界値としてメロディの再生と停止の判定を行います。

再生と停止がうまく切り換わらない場合は\$light\_val の値を調整してください。

画面上に現在の電圧値が表示されていますので、明るくした場合と暗くした場合に表示される値を確認して、それらの中間の値を\$light\_val に設定するようにプログラムを修正してください。



**Memo : CdS とは**

CdS 光導電セルとは、光を当てることで半導体内の電流の流れに変化が生じ抵抗値が下がる、という性質をもった素子です。



## 3.8 傾きを検知する

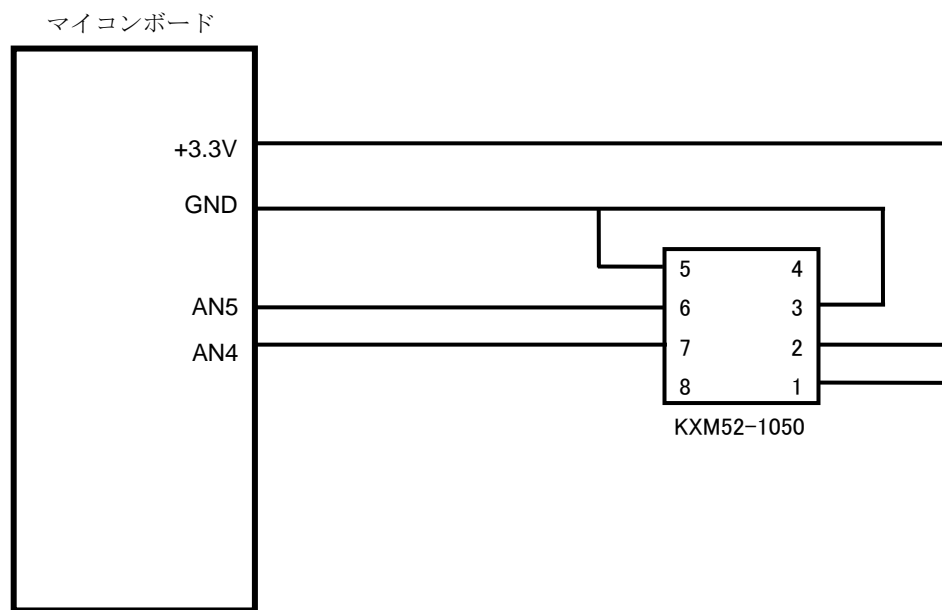
本項では、加速度センサーを使って傾きを検知する例を説明します。ここでは、加速度センサーとして KXM52-1050 (秋月電子通商 製) という名前のセンサーを使います。

加速度センサーは X, Y, Z の 3 軸の傾きを検出することができます。ここでは、X, Y の 2 軸の傾きを検出して、液晶画面上のグラフィックを動かすプログラムを作成します。

必要な部品一覧

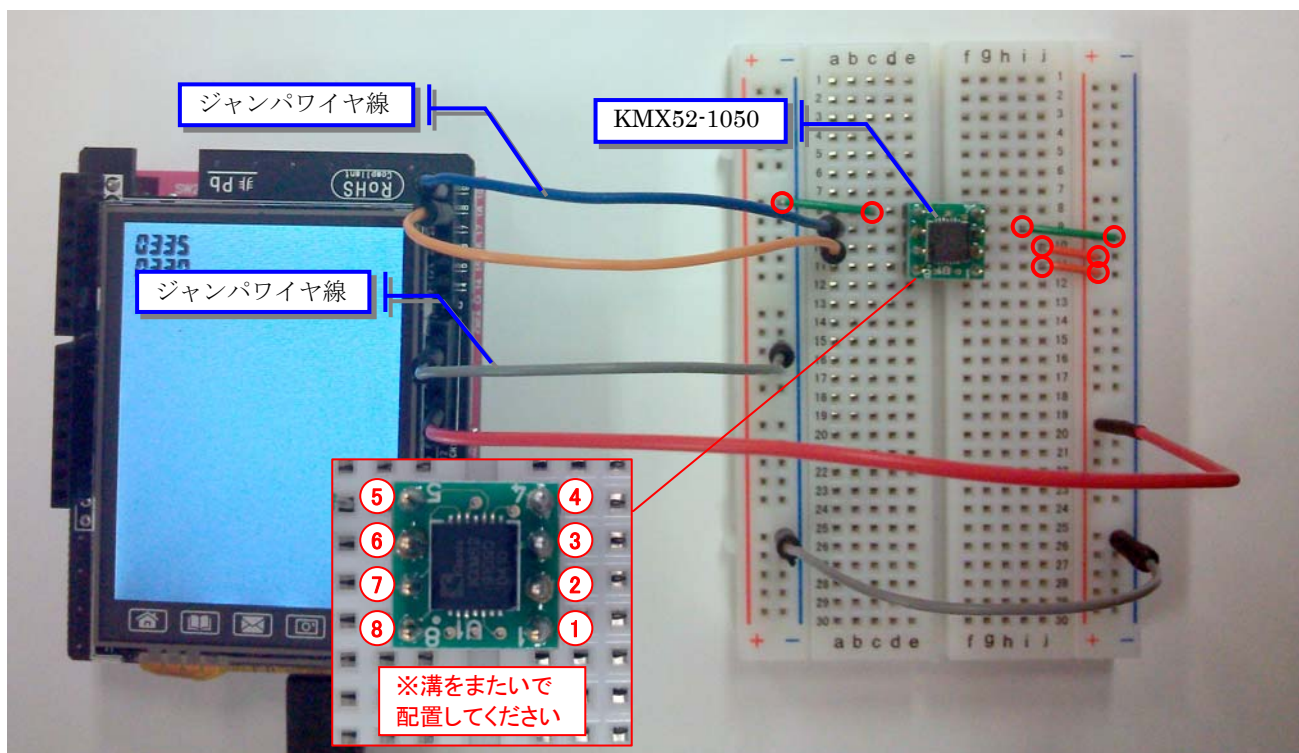
- |                       |     |
|-----------------------|-----|
| ・ブレッドボード              | 1 個 |
| ・加速度センサー (KXM52-1050) | 1 個 |
| ・ジャンプワイヤ線             | 9 本 |

### (1) 回路図

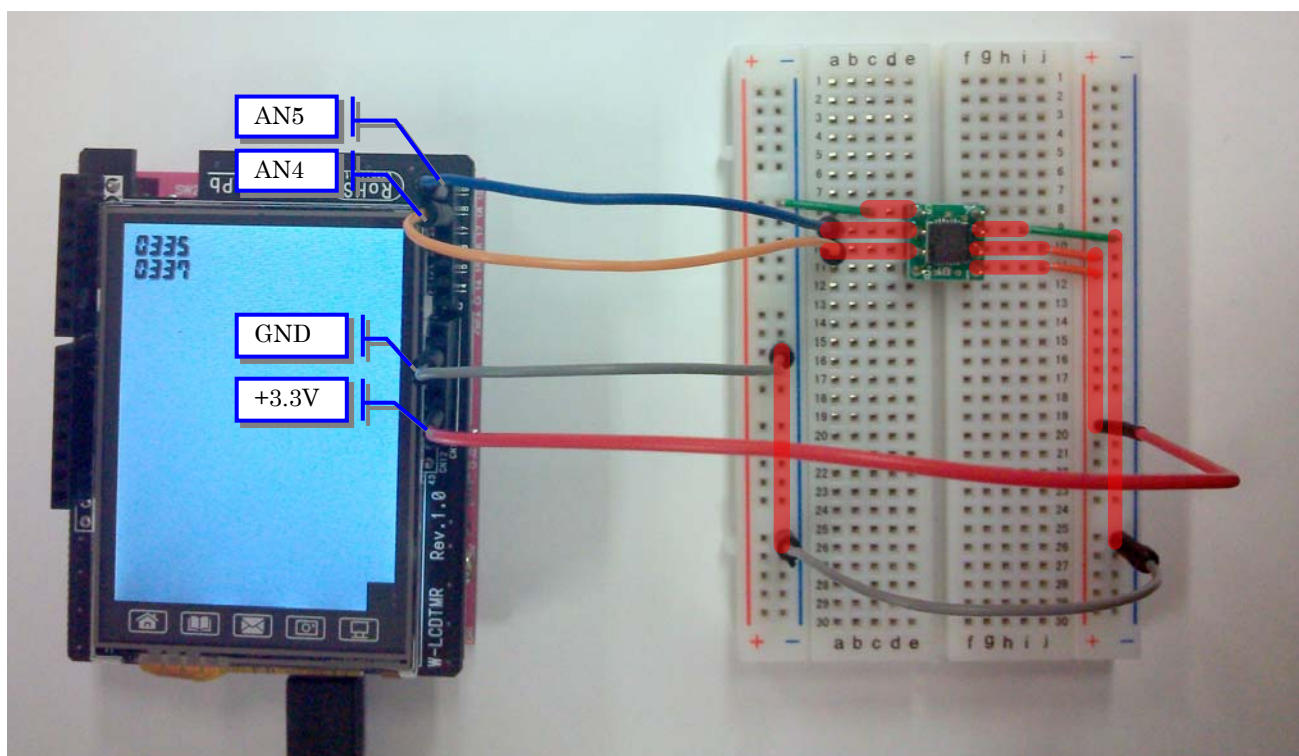


## (2) 接続図

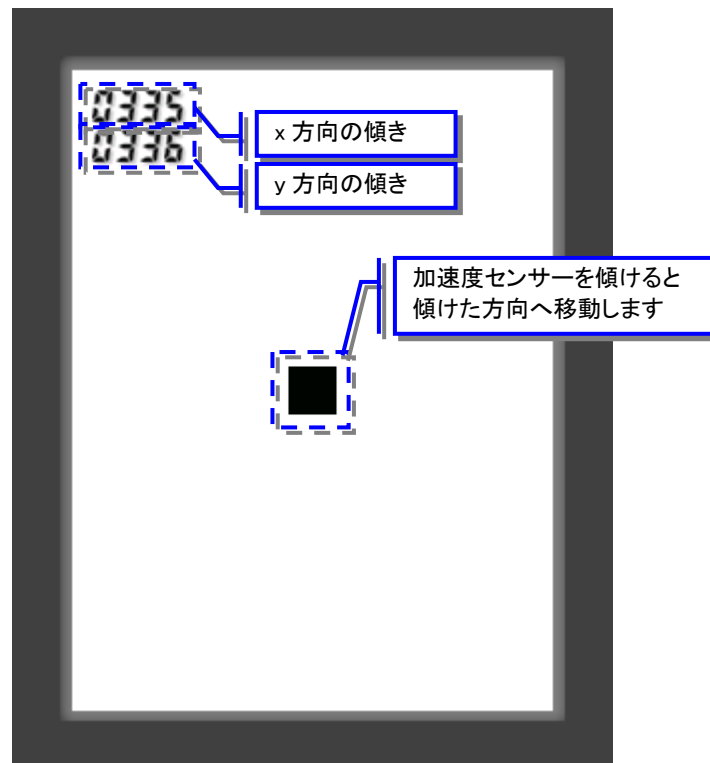
赤丸部分にジャンプワイヤ線を差し込んでいます。KMX52-1050 の配置と向きに注意してください。



赤い線の部分がブレッドボード内で接続されています。



### (3) 液晶画面表示



#### (4) プログラムの作成

下記のプログラムを作成して実行してください。

同じ内容のプログラムファイルがインストールフォルダ内の `sample¥sample10` フォルダに `"accelometer.rb"` という名前で格納されています。

```
# 傾きの中心値の設定 (センサを水平な場所に置いて調整してください)
$centerX = 336
$centerY = 336
$margin = 4

# 表示する矩形の大きさ
$sizeX = 24
$sizeY = 24

# メイン初期化
#-----
def setup()
  # 傾きの値を表示する部品を配置
  $numb1 = GNDigitalNum.new(10, 10)
  $numb2 = GNDigitalNum.new(10, 30)

  # 表示位置
  $posX = 120
  $posY = 160
  $newX = 120
  $newY = 160

  # 移動範囲
  $posXmin = 0
  $posYmin = 0
  $posXmax = 240 - $sizeX
  $posYmax = 320 - $sizeY
end

# メインループ
#-----
def loop()
  # x 方向傾きの取得
  axisX = gr_analogRead($PIN_AN4)
  $numb1.setInteger(axisX)

  # y 方向傾きの取得
  axisY = gr_analogRead($PIN_AN5)
  $numb2.setInteger(axisY)

  # 移動量を一旦 0 にします
  velX = 0
  velY = 0

  # x 方向の傾きによって x 方向の移動量を決めます
  if axisX > ($centerX + $margin) then
    velX = 1
  end
  if axisX < ($centerX - $margin) then
    velX = -1
  end

  # y 方向の傾きによって y 方向の移動量を決めます
  if axisY > ($centerY + $margin) then
    velY = 1
  end
  if axisY < ($centerY - $margin) then
    velY = -1
  end

  # 新しい x 座標を計算します
  $newX = $posX + velX
```

```
# 画面の端では止まります
if $newX > $posXmax then
    $newX = $posXmax
end
if $newX < $posXmin then
    $newX = $posXmin
end

# 新しいy座標を計算します
$newY = $posY + velY

# 画面の端では止まります
if $newY > $posYmax then
    $newY = $posYmax
end
if $newY < $posYmin then
    $newY = $posYmin
end

# 移動している場合は矩形の表示を更新します
if $posX != $newX || $posY != $newY then
    # 現在位置を消去
    GNDDrawRect($posX, $posY, $sizeX, $sizeY, 31, 31, 31)

    # 新しい位置に描画
    GNDDrawRect($newX, $newY, $sizeX, $sizeY, 0, 0, 0)

    # 描画位置の更新
    $posX = $newX
    $posY = $newY
end
end
```

## (5)傾きの取得と判定

傾きの取得はメインループで行っています。

```
# 傾きの中心値の設定 (センサを水平な場所に置いて調整してください)
$centerX = 336
$centerY = 336
$margin = 4
傾きを検知するための基準となる値です。
傾きセンサーを水平に置いた場合の数値を設定します。

(中略)

# メインループ
#-----
def loop()
  # x方向傾きの取得
  axisX = gr_analogRead($PIN_AN4)
  $numbmp1.setInteger(axisX)
  電圧値を端子 AN4 から取得して、
  取得した値を画面に表示します。

  # y方向傾きの取得
  axisY = gr_analogRead($PIN_AN5)
  $numbmp2.setInteger(axisY)
  電圧値を端子 AN5 から取得して、
  取得した値を画面に表示します。

  # 移動量を一旦 0 にします
  velX = 0
  velY = 0

  # x方向の傾きによって x 方向の移動量を決めます
  if axisX > ($centerX + $margin) then
    velX = 1
  end
  if axisX < ($centerX - $margin) then
    velX = -1
  end
  x方向の傾きが中心値より大きければ移動量を1に、
  中心値より小さければ移動量を-1にします。

  # y方向の傾きによって y 方向の移動量を決めます
  if axisY > ($centerY + $margin) then
    velY = 1
  end
  if axisY < ($centerY - $margin) then
    velY = -1
  end
  y方向の傾きが中心値より大きければ移動量を1に、
  中心値より小さければ移動量を-1にします。

  (中略)

end
```

水平な状態からの傾きを検知するために、基準となる数値を\$centerX, \$centerY という変数に格納しています。これらは、センサー部品によって値が変わる可能性がありますので、うまく動かない場合は調整してください。

加速度センサーを水平な台の上に置いたときに画面上に表示される数値を確認して、上記プログラムの\$centerX と\$centerY の初期設定値を変更してください。

## 3.9 最後に

これで、本書の説明は終わります。

第3章でいくつかの電子回路を作成して、センサーやブザーの基本的な使い方が理解できたならば、色々な部品を組み合わせで新しい電子回路を作ることができるのではないかと思います。

本書では扱いませんでしたが、入力に使えるセンサーとしては距離を計測するものや温度を計測するもの等様々な種類があります。また、出力先も LED やブザーだけではなくモーター等を接続することも可能です。

自分だけのアイデアで世界に一つしかない電子回路を作ってみると、もっと楽しく電子工作をすることができると思います。



## 第4章 付録

---

プログラムがうまく動かない場合の対策方法やマイコンボードのシステムプログラムの書き換え方法を説明します。

何か問題が発生した場合に参照してください。

## 4.1 プログラムが動かないときは

### 4.1.1 プログラムがコンパイルできない場合

■プログラムコンパイル時に“Error : compile error occured.”というダイアログが表示される

mruby プログラムに何らかの記述の間違があるとコンパイル結果の“program.mbi”のファイルサイズが0KBとなり、上記のエラーダイアログが表示されます。mruby プログラムにタイプミスなどの間違がないか確認してください。

一度に大きなプログラムを作成するとプログラムの書き間違を見つけることが大変となります。プログラムの動作を確認しながら少しずつプログラムを大きくしていくことをお勧めします。

■プログラムコンパイル時に“Error : too large program.”というダイアログが表示される

同梱のマイコンボードは記憶容量が小さいため、あまり大きな mruby プログラムを実行することができません。上記のダイアログが表示された場合、プログラムサイズが限界を超えています。プログラムサイズを小さくする必要があります。

### 4.1.2 画像ファイルが変換できない場合

■PNG 形式画像の変換時に“Error : too large image.”というダイアログが表示される

表示可能な画像の大きさは縦 320 ドット、横 240 ドットまでとなっています。それより大きな画像を変換した場合に上記のダイアログが表示されます。画像の大きさを小さくする必要があります。

■PNG 形式画像の変換時に“Error : too long image filename.”というダイアログが表示される

表示可能な画像のファイル名の長さは拡張子“.png”を除いて8文字以下である必要があります。それより長いファイル名を付けた場合に上記のダイアログが表示されます。ファイル名を短くする必要があります。

■PNG 形式画像の変換時に“Error : unknown image format.”というダイアログが表示される

変換する PNG 形式画像の形式が非対応のものの場合に表示されます。変換可能な形式は RGB 24bit 形式の画像と ARGB 32bit 形式の画像です。ARGB 32bit 画像の場合、アルファ値の情報は無視されます。

■PNG 形式画像の変換時に“Error : illegal image data.”というダイアログが表示される

画像ファイルの拡張子が“.png”ですが、ファイルの内容が PNG 形式ではない場合などに表示されます。画像ファイルの内容を確認してください。

■PNG 形式画像の変換時に“Error : the program can't start.”というダイアログが表示される

microsoft 製の補助プログラムのインストールが必要です。

インストールフォルダ内の `redistpackage%vc%credist_x86.exe` というプログラムを起動して Microsoft Visual C++ 2008 Redistributable をインストールしてください。

### 4.1.3 マイコンボードでプログラムが実行できない場合

マイコンボードでプログラムがうまく動かないときは以下のことを確認してみましょう。

#### ■本書付属のツールのインストール先は正しいか

付属のツールやサンプルプログラムのコピー先のフォルダ名に空白文字が含まれていると mruby プログラムのコンパイルが正常に行われず、可能性があります。“C:¥work¥mruby¥”等、フォルダ名に空白文字を含まない場所にコピーしてください。

#### ■SD カードが刺さっているか

マイコンボードに電源を入れると SD カード上のプログラムを実行します。SD カードが刺さっているかどうか確認してください。

#### ■SD カードに必要なプログラムが書かれているか

SD カードの中にプログラムファイル“`program.mbi`”がコピーされているか確認してください。

#### ■“`program.mbi`”ファイルのファイルサイズが 0KB ではないか

mruby プログラムに何らかの記述の間違いがあるとコンパイル結果の“`program.mbi`”のファイルサイズが 0KB となり、正常に動作しません。mruby プログラムにタイプミスなどの間違いがないか確認してください。

一度に大きなプログラムを作成するとプログラムの書き間違いを見つけることが大変となります。プログラムの動作を確認しながら少しずつプログラムを大きくしていくことをお勧めします。

#### ■mruby プログラムが大きすぎないか

同梱のマイコンボードは記憶容量が小さいため、あまり大きな mruby プログラムを実行することができません。mruby プログラムのコンパイルを行った場合に、“`Error : too large program.`”というメッセージが表示された場合、プログラムサイズが限界を超えています。プログラムサイズを小さくする必要があります。

#### ■画面に配置した部品が表示されない

同時に作成できる部品の最大数は 32 です。それより多くの部品を作成することはできません。

#### ■ピクチャ部品の画像が表示されない

SD カード上に設定したファイル名の画像がコピーされているかどうか確認してください。

#### 4.1.4 マイコンボードでプログラムの実行中に動作停止する場合

プログラムの実行途中で動作が停止する場合、以下の可能性があります。

##### ■プログラム実行用のメモリが足りない

同梱のマイコンボードは記憶容量が小さいため、あまり大きな mruby プログラムを実行することができません。プログラムの実行時に画面に“Error: program load failed.”と表示されて停止する場合はメモリ使用量が限界を超えている可能性があります。その場合、プログラムサイズを小さくする必要があります。

##### ■メソッド名などが間違えている

プログラム実行中に“undefined method '**メソッド名**'”と表示されて動作を停止した場合は mruby プログラム内に存在しないメソッド名を記述している可能性があります。メソッド名の書き間違いなどがないか確認してください。

##### ■プログラムが意図した通りに動かない

プログラムが意図した通りに動かない場合、プログラムのどの部分を実行しているか、変数の値がどうなっているか、等を調べることによって解決する可能性があります。

そのような場合は、一時的に液晶画面にデジタル数値部品やテキストボックス部品を配置して、プログラムの動作している場所や変数の内容を表示することでプログラムの問題を調べることができます。

## 4.2 マイコンボードのシステムプログラムの書き込み方法

### 4.2.1 システムプログラムについて

SD カードに書き込んで動かすプログラムとは別に、マイコンボード上にシステムプログラムが書き込まれています。

通常はシステムプログラムの書き換えは不要ですが、以下のような場合に書き込みを行う必要があります。

- ・マイコンボードに他のプログラムを作成して動かした場合

付属のマイコンボードに他のプログラムを書きこんで動かした場合、本書に載っているサンプルプログラムは動作しなくなりますが、インストールフォルダ内の `binary\mrbsys_1_00_000.bin` というファイルをマイコンボードに書き込むことによって出荷時の状態に戻すことができます。

- ・システムプログラムの更新を行う場合

今後、システムプログラムの機能拡張等でシステムプログラムを更新する予定です。更新に関してはWebPage経由でダウンロード可能とする予定です。WebPageのURLに関しては、1.1.1(6)」を参照してください。

## 4.2.2 システムプログラムの書き込み方法

上記の `mrbsys_1_00_000.bin` ファイル、または、ダウンロード提供された更新ファイルをマイコンボードに書き込む手順は下記となります。

1. マイコンボードと PC を USB ケーブルで接続します。

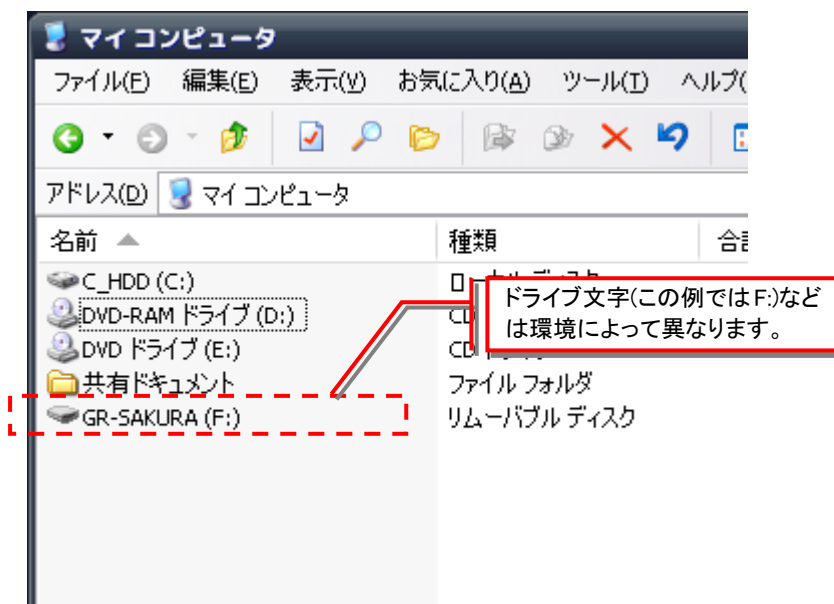
「1.2 接続方法」の説明と同じ方法で接続してください。

2. マイコンボード上のリセットボタンを押してください

「2.1.1 マイコンボードの各端子と機能」の図の 6 番の赤いボタンがリセットボタンです。

3. マイコンボードが PC の外付けドライブとして認識されます

“GR-SAKURA”という名前のドライブが PC で認識されます。Windows の「マイコンピュータ」画面で確認してください。下図は Windows XP の場合の画面です。



4. “GR-SAKURA”ドライブにシステムプログラムファイルをコピーします

Windows エクスプローラを使って“GR-SAKURA”という名前のドライブを開いて、その中にシステムプログラム(`mrbsys_1_0_000.bin`)をコピーしてください。通常のファイルコピーと同様にファイルをドラッグ&ドロップ操作によってコピー可能です。

5. 完了

以上で書き込み手順は完了です。4. のファイルのコピーが完了するとマイコンボードは自動的に起動します。

---

著者            株式会社アイ・エル・シー  
                 〒732-0824  
                 広島市南区的場町1丁目3番6号 広島の場ビル9F  
                 <http://www.ilc.co.jp>

---

# EAPL-Trainer™

Embedded Application development Trainer

mruby

