倒立振子ライブラリをモデルに組込むには

ETロボコン実行委員会 – er-info@etrobo.jp – 2.0, 2016-05-21 12:56:41 | 2016年用

sample_c4（倒立振子のサンプル）を試す

倒立振子を使ったサンプル「sample_c4」を動かしてみましょう。

sample_c4は、次のサイトから入手できます:

GitHubのetroboEV3のページ

次の手順でビルドします:

1. sample_c4フォルダを、EV3RTインストールディレクトリの `hrp2/sdk/beginners` の中にコピーします
2. コピーしたら、コピー先のディレクトリの内容を確認しましょう

```
$ cd（ev3rtのインストールディレクトリ）/hrp2/sdk/beginners
$ ls
Makefile  sample00/  sample01/  sample03/  sample04/  sample05/  sample_c4/

$ ls sample_c4
Makefile.inc  app.c  app.cfg  app.h  balancer.c  balancer.h  balancer_param.c
balancer_private.h
```

3. `beginners` ディレクトリでビルドします

```
$ cd hrp2/sdk/beginners
$ make app=sample_c4
```

4. ビルドに成功すると `app` という実行ファイルが生成されます

実行する手順は次の通りです:

1. プログラムをEV3本体に転送します
2. 尻尾を一番上に向けてプログラムを起動します
3. 尻尾が完全停止位置に動きます
4. 画面に「EV3way-ET sample_c4」と表示されます
5. 走行体をラインの左エッジに配置します
6. タッチセンサを押すと走行を開始します
7. 走行を終了するには、EV3本体の「Back」ボタンを押します

sample_c4の構造をクラス図に表す

astah* で「sample_c4」プロジェクトを作成し、sample_c4のコードをクラス図を使って整理してみましょう。

次のような方針でクラス図を使った図に直してみます

- ev3apiの関数群は「ev3api」クラスのメソッドと考えてクラスにまとめます
- `main_task` はクラスではありませんが、この演習ではクラスを使って表すことにします（この演習では `bt_task` については気にしないでおきます）
- `app.c` で定義している他の関数は `app` クラスのメソッドと考えましょう
- `balancer_init` と `balancer_control` は `balancer` クラスのメソッドと考えましょう
- グローバルは定数や変数は `app` クラスの属性と捉えることができますが、今回の作図では割愛しましょう

作成したsmaple_c4の構造を表したクラス図

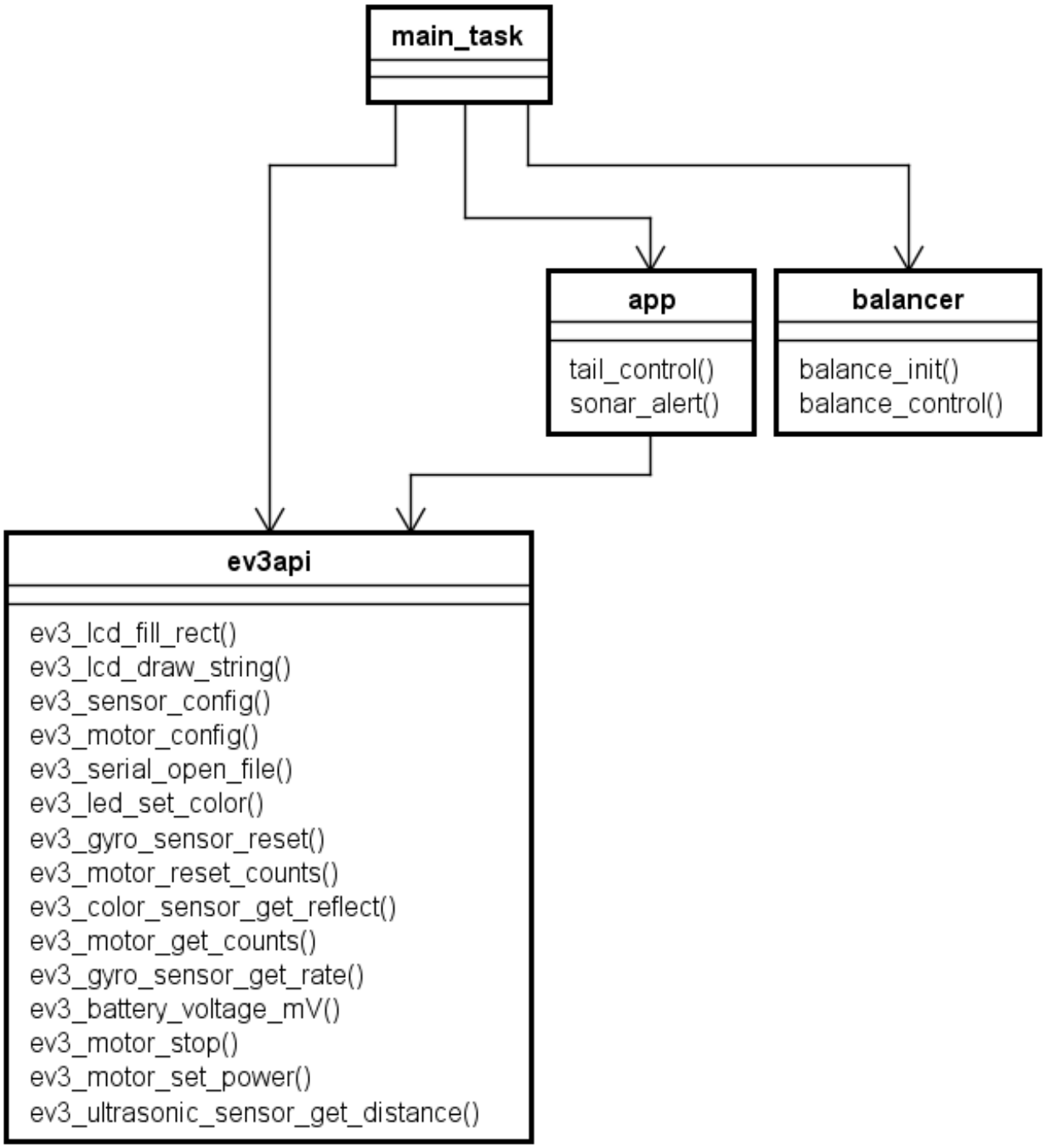

図 1. sample_c4のクラス図

倒立振子ライブラリのパラメータを調べる

操作量を算出する関数（`balance_control`）のパラメータをみると、その更新時期・間隔には違いがあるのがわかります。

これは、使う側からすると、呼び出すのと設定するのは同じタイミングでなくてもよいという意味になります。

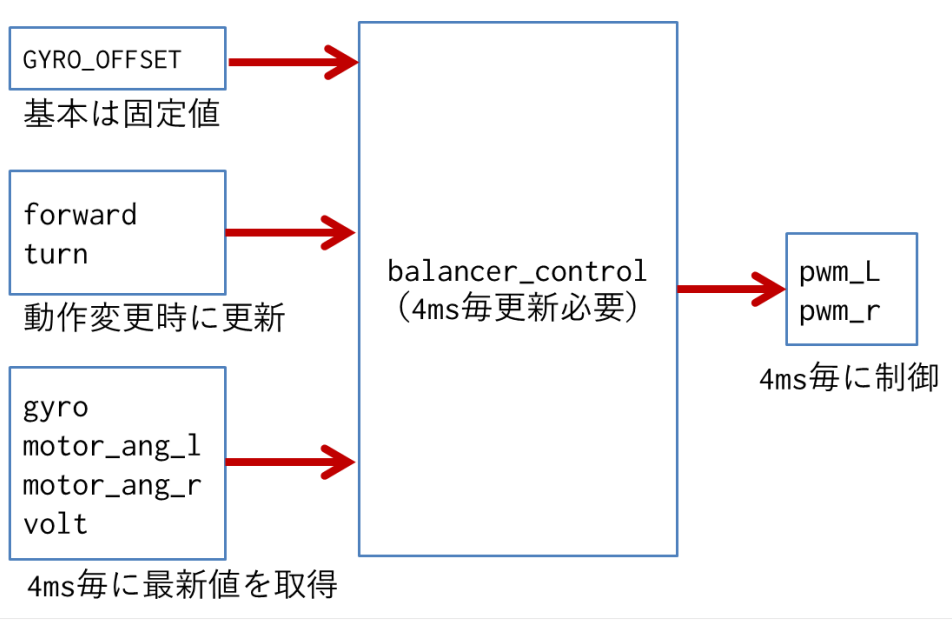

図 2. balacne_controlのパラメータの更新タイミング

倒立振子ライブラリの呼び出し方を考える

更新タイミングの違いについて、次のように整理してみました。

- `GYRO_OFFSET` は、一度だけ取得して覚えておけばよいので、`init` の時に渡して覚えさせておけばよさそうです
- `forward` , `turn` は、動作を変更したい時にしか変更しないので、変更時に覚えさせておけばよさそうです
- `gyro` , `motor_ang_l` , `motor_ang_r` , `volt` は、4msの更新のたびに最新値を取得して渡すことが必要そうです

つまり、**倒立振子ライブラリのために覚えておく変数を属性** 、 **ライブラリの関数を操作** と考えて、セットにしてクラスとみなせば、倒立振子ライブラリをクラスとして仕立てることができることになりそうです。

> では、どのようなクラスになりそうでしょうか？

整理した結果から得たBalancerクラス

検討の結果、次のようなクラスを作ることにしました。

属性やメソッド説明は次のページにあります。

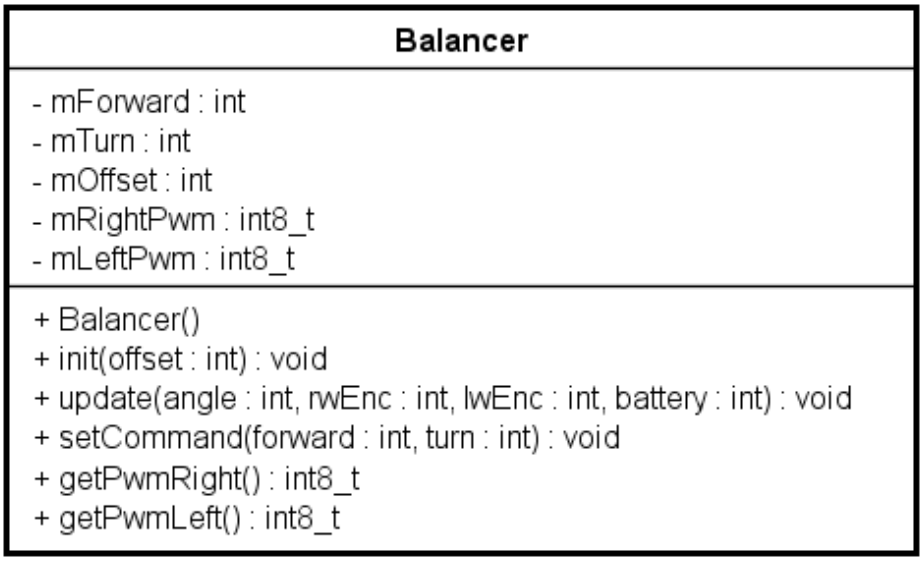

図 3. 整理した結果から得たBalancerクラス

Balancerクラスの説明

- `gyro` , `motor_ang_l` , `motor_ang_r` , `volt` は `update` メソッドに渡し、このメソッドを呼び出すと `balance_contol` が呼び出されて操作量を計算します
- 操作量の計算結果は、属性 `mLeftPwm`、 `mRightPwm` に保存しておきます
  - この属性がることで、このクラスの利用者は、最後に `update` メソッドを呼び出した時の計算結果を `getPwnLeft` 、 `getPwmRight` メソッドを使って参照できるようになります
- `GYRO_OFFSET` は `init` メソッドに渡して、属性に保存しておきます
- `forward`, `turn` は `setCommand` メソッドに渡して、属性に保存しておきます

sample_cpp4の構造をクラス図に表す

作成した `Balancer` クラスを使用したクラス図を描きましょう。

1. `sample_c4.asta` をコピーして `sample_cpp4.asta` を用意します
2. 「sample_c4のクラス図」を「sample_cpp4のクラス図」に変更します
3. `Balancer` クラスを追加します。
4. 関連を引き直します

作成したsample_cpp4のクラス図

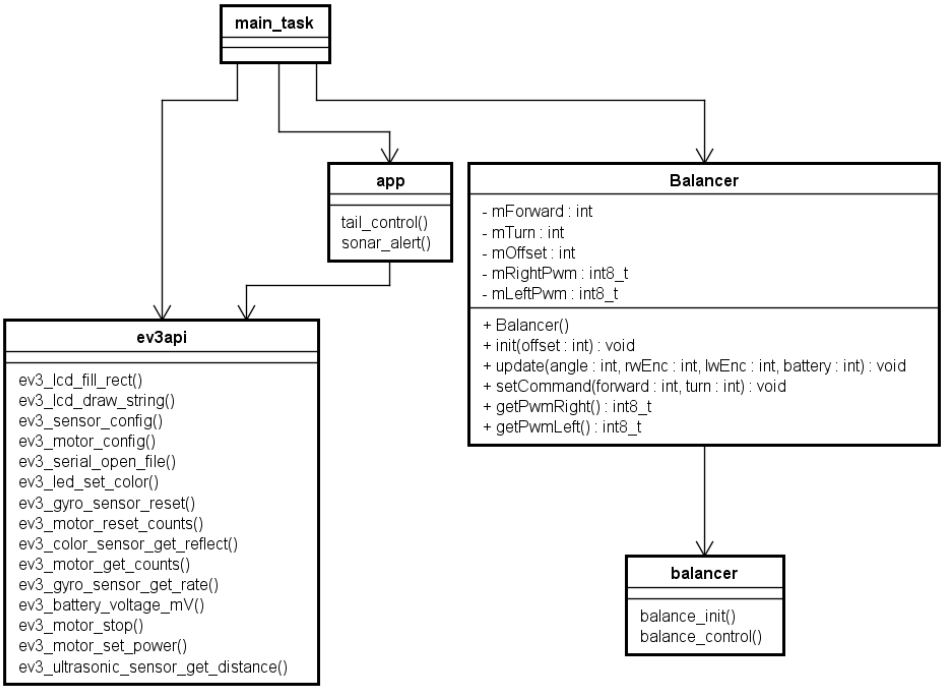

図 4. Balancerクラスを追加したsample_cpp4のクラス図

sample_cpp4のコードを作成する準備

C++のクラスを追加するので、コードを修正するため前に少し準備しましょう。

1. 「sample_c4」ディレクトリをそのままコピーして「sample_cpp4」ディレクトリを作成します
2. `app.c` は、C++のコードを扱うので `app.cpp` に変更します
3. `Makefile.inc` を次のように編集します

Makefile.incの抜粋

```
APPL_COBJS += balancer.o balancer_param.o

APPL_CXXOBJS += BalancerCpp.o ①

SRCLANG := c++ ②
```

① C++オブジェクトとして「BalancerCpp.o」を追加します

② C++用の言語設定を追加します

修正できたら、ビルドしてみて、修正前と同じように動作するか確認しておきましょう。

BalancerCpp.hを追加する

`sample_cpp4` ディレクトリに `Balancer` クラスのヘッダファイルを作成します。

既存の `balance` 関数のファイルと区別するため、ファイル名を `BalancerCpp.h` としました。

BalancerCpp.h

```
#ifndef EV3_UNIT_BALANCERCPP_H_
#define EV3_UNIT_BALANCERCPP_H_

#include "ev3api.h"

class Balancer {
public:
    Balancer();

    void init(int offset);
    void update(int angle, int rwEnc, int lwEnc, int battery);
    void setCommand(int forward, int turn);
    int8_t getPwmRight();
    int8_t getPwmLeft();

private:
    int mForward;
    int mTurn;
    int mOffset;
    int8_t mRightPwm;
    int8_t mLeftPwm;
};

#endif  // EV3_UNIT_BALANCERCPP_H_
```

BalancerCpp.cppを追加する

`sample_cpp4` ディレクトリに `Balancer` クラスのcppファイルを作成します。
既存の `balance` 関数のファイルと区別するため、ファイル名を `BalancerCpp.cpp` としました。

BalancerCpp.cpp

```
#include "balancer.h"

#include "BalancerCpp.h"

/**
 * コンストラクタ
 */
Balancer::Balancer()
    : mForward(0),
      mTurn(0),
      mOffset(0),
      mRightPwm(0),
      mLeftPwm(0) {
}

```

```cpp
/**
 * バランサを初期化する
 * @param offset ジャイロセンサオフセット値
 */
void Balancer::init(int offset) {
    mOffset = offset;
    balance_init();  // 倒立振子制御初期化
}

/**
 * バランサの値を更新する
 * @param angle   角速度
 * @param rwEnc   右車輪エンコーダ値
 * @param lwEnc   左車輪エンコーダ値
 * @param battety バッテリ電圧値
 */
void Balancer::update(int angle, int rwEnc, int lwEnc, int battery) {
    // 倒立振子制御APIを呼び出し、倒立走行するための
    // 左右モータ出力値を得る
    balance_control(
        static_cast<float>(mForward),
        static_cast<float>(mTurn),
        static_cast<float>(angle),
        static_cast<float>(mOffset),
        static_cast<float>(lwEnc),
        static_cast<float>(rwEnc),
        static_cast<float>(battery),
        &mLeftPwm,
        &mRightPwm);
}

/**
 * PWM値を設定する
 * @param forward 前進値
 * @param turn    旋回値
 */
void Balancer::setCommand(int forward, int turn) {
    mForward = forward;
    mTurn    = turn;
}

/**
 * 右車輪のPWM値を取得する
 * @return 右車輪のPWM値
 */
int8_t Balancer::getPwmRight() {
    return mRightPwm;
}

/**
 * 左車輪のPWM値を取得する
 * @return 左車輪のPWM値
 */
int8_t Balancer::getPwmLeft() {
    return mLeftPwm;
}
```

app.cppを修正する

作成したBalancerクラスを使って、app.cppのコードを修正しましょう

app.cpp（抜粋その１）

```
/* 関数プロトタイプ宣言 */
static int sonar_alert(void);
static void tail_control(signed int angle);

#include "BalancerCpp.h"          (1)
Balancer balancer;                (1)

/* メインタスク */
void main_task(intptr_t unused)
{
```

(1) 追加したBalancerクラスのインスタンスを作成しています

app.cpp（抜粋その２）

```
    /* ジャイロセンサーリセット */
    ev3_gyro_sensor_reset(gyro_sensor);
    balancer.init(GYRO_OFFSET);                  (1)

    ev3_led_set_color(LED_GREEN); /* スタート通知 */
```

(1) BalancerクラスのinitメソッドをＯ呼び出しています
（GYRO_OFFSETを渡しています）

app.cpp（抜粋その３）

```
        /* 倒立振子制御API に渡すパラメータを取得する */
        motor_ang_l = ev3_motor_get_counts(left_motor);
        motor_ang_r = ev3_motor_get_counts(right_motor);
        gyro = ev3_gyro_sensor_get_rate(gyro_sensor);
        volt = ev3_battery_voltage_mV();

        /* 倒立振子制御APIを呼び出し、倒立走行するための */
        /* 左右モータ出力値を得る */
        balancer.setCommand(forward, turn);    (1)
        balancer.update(gyro, motor_ang_r, motor_ang_l, volt); (2)
        pwm_L = balancer.getPwmRight();        (3)
        pwm_R = balancer.getPwmLeft();         (3)
```

(1) forward, turnのコマンドを設定しています
（ここでは毎回呼んでいますが変更のあった時だけでよいです）

(2) センサの最新値をbalancerライブラリへ渡して、操作量を再計算しています

(3)

算出した操作量をモータに渡す変数に設定しています

修正が済んだら、ビルドして、動作を確認しましょう。

ここまでのまとめ

要素技術を検討して得られたC言語で作成した既存のライブラリ（この演習では倒立振子ライブラリ）をシステムのモデルに組込むため、クラスに仕立てる方法を演習しました。

- パラメータの更新機会に着目して、同時に必要なものをまとめました
- 更新機会が少ないパラメータは、クラスの属性に保持することで覚えておけます
- 更新機会に応じてメソッドを分けることで、一度の呼び出に必要なパラメータを減らすことができます
- パラメータを属性に保持することで、メソッドのインターフェースが必要最小限に抑えられます

トレーニングのまとめ

このトレーニングでは、モデルとコードの対応づけ、要素技術をモデル図に反映する方法について演習しました。

1. コードのありのままの状況を整理するのにもモデル図が使えることがわかりました
2. モデルとコードを対応づけていれば、双方向に利用できることがわかりました
    - モデルで考えることもコードから設計（モデル）に戻って検討することもできます
3. 要素技術を利用するときは、実験と組込みを区別して考えましょう
4. 要素技術をシステムに組込むには、モデル図の要素としての名前（クラスや操作など）をつけて、モデル図から読み取れるようにしましょう
    - 倒立振子ライブラリも要素技術のひとつとして、モデル図に組み込めることがわかりました

このトレーニングの結果を活かして、より見通しのよいシステムやモデル図が作成できるようになることを期待しています。

本資料について

レーニングのスライドです。

2.0, 2016-05-21 12:56:41, 2016年用

2.0
Last updated 2016-05-21 02:19:46 JST