FCの作り方

（やさしめ）

0. 前提知識と注意書き

- FCを作る際に最初に読む資料を想定しています
- なんとなくの流れがわかれば〇（難しい書き方は知らなくてOK）

```
// PreArmStateへの遷移
manager.changeState(std::make_unique<PreArmingState>());
```

- コメントのない部分は気にしなくて大丈夫！

```
void InitState::update(FlightManager& manager)
```

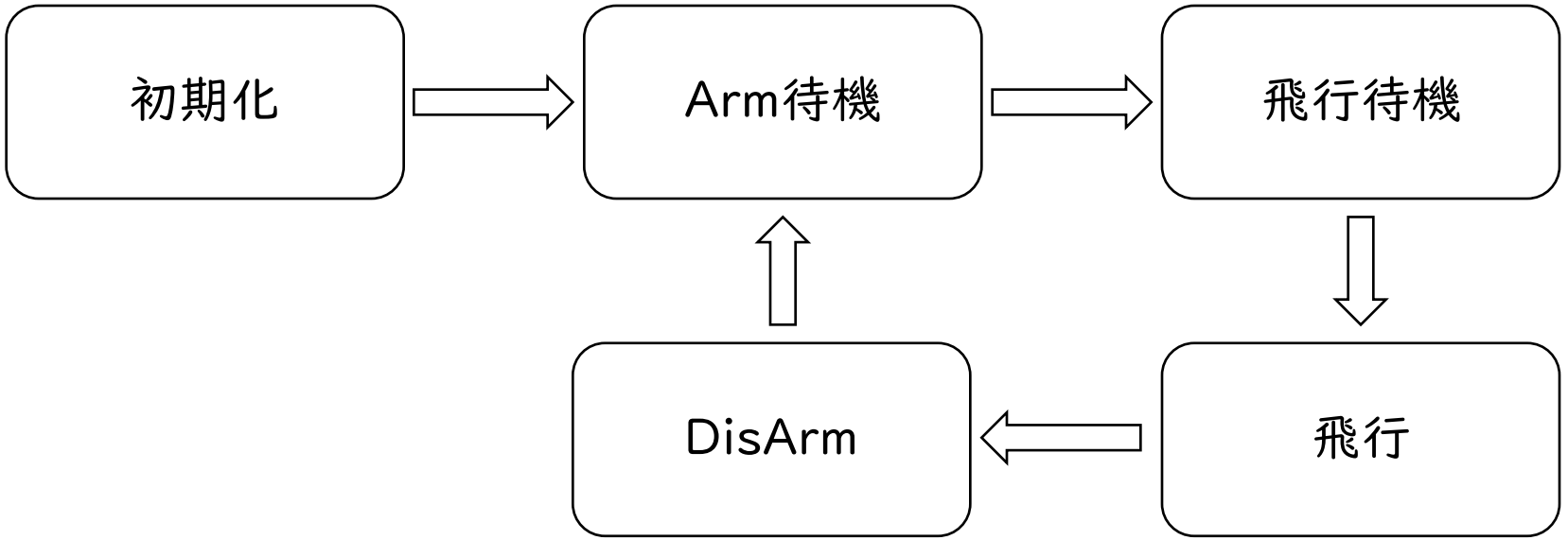
- 見たことないコードや全体の設計は、次の講座で説明します

- 元のコードは、ここにあります（読んでみてもいいかも？）

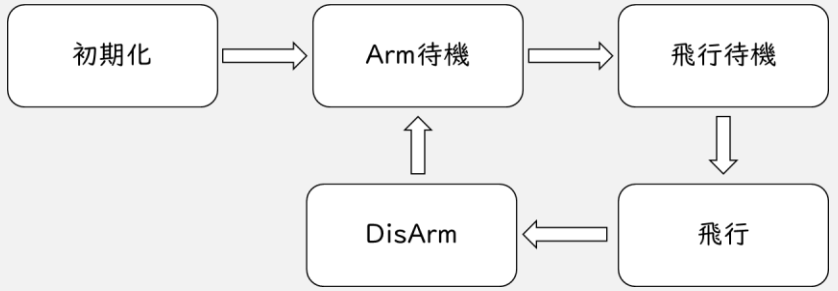
https://github.com/NOKOLat/PFLIGHT/tree/master

1. FCの全体像

・基本的な処理の流れ



ファイルの場所：Core/Src/wrapper.cpp（下側）



2. SBUSデータの取得

・SBUSデータの取得

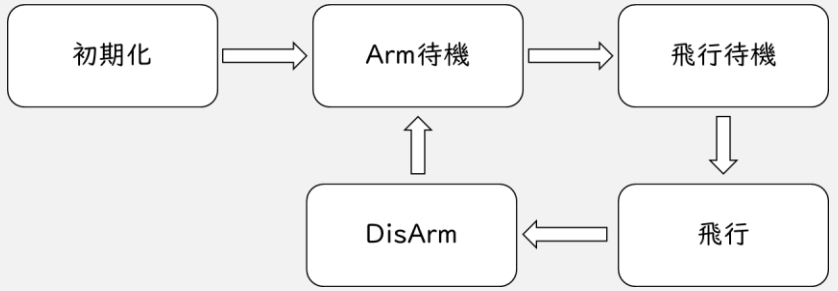
・データを受信したら割り込み処理を行う
→ほかの処理を止めて処理をする

```cpp
// UART割り込み
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){

    // UART5(DMA) SBUS受信用
    if(huart == &huart5){

        // 各チャンネルのデータを取得
        sbus.parse();
        sbus_data = sbus.getData();

        // スイッチ判定などの処理を実行
        decoded_sbus_data = nokolat::Decode(sbus_data);

        // 判定結果をFlightManagerに渡す
        flightManager.sbusUpdate(decoded_sbus_data);

        // 受信の再開
        HAL_UART_Receive_DMA(&huart5, sbus.getReceiveBufferPtr(), sbus.getDataLen());
    }
}
```

0. データの受信
1. データの解析
2. スイッチやスロットルデータの取り出し・判定
3. 次の受信への準備

ファイルの場所：Core/Src/State/InitState.cpp

3. 初期化処理



・飛行や制御に必要なチェックをする

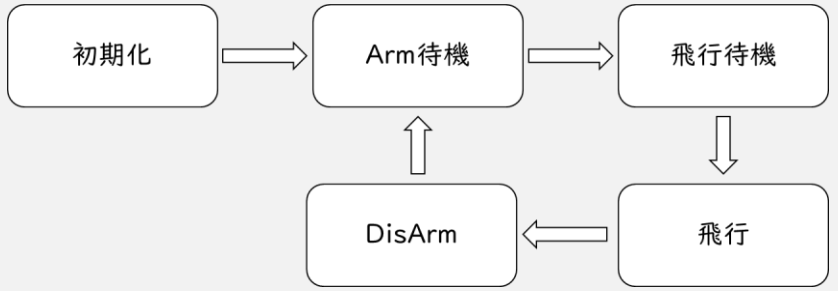
1. SBUSの受信チェック
2. IMUの通信チェック
3. モーターのチェック（モータ数が適切か）
4. サーボモーターの初期化

・成功したらLEDをつけてArm待機へ

```
void InitState::update(FlightManager& manager) {

    // SBUSの受信チェック
    if(!manager.sbus_data.is_receive){

        printf("SBUS_ERROR \n");
        return;
    }

    // IMUの通信チェック
    if (manager.imuUtil) {

        if (manager.imuUtil->Init() != 0) {

            printf("IMU_ERROR \n");
            return;
        }
    }
    // Motorの設定チェック
    if(manager.pwm.CheckMotorSetting(motor_count)){

        printf("MotorSetting_Error\n");
        return;
    }

    // Servoの初期化
    manager.pwm.InitServo();

    // 赤LEDをつける
    manager.red_led.Set(PinState::on);

    // PreArmStateへの遷移
    manager.changeState(std::make_unique<PreArmingState>());
}
```

失敗したら次のループで再度判定

ファイルの場所：Core/Src/State/PreArmingState.cpp



4. Arm待機処理

・Armの判定と処理

0. Armスイッチのチェック
1. ESCの初期化
2. センサーのキャリブレーション

・LEDをつけて飛行待機へ

```
void PreArmingState::update(FlightManager& manager) {

    // Armスイッチをチェック
    if(manager.sbus_data.arm){

        //ESCの初期化をすませておく
        manager.pwm.InitMotor();

        // センサーのキャリブレーション
        manager.imuUtil->Calibration(UserSetting::calibration_count);

        //黄LEDをつける
        manager.yellow_led.Set(PinState::on);

        //PreFlightStateに遷移
        manager.changeState(std::make_unique<PreFlightState>());
    }

    // Servo判定とPwm出力(abc_value = 0)
    manager.pwm.CalcServo(manager.sbus_data, 0, manager.control_data.servo_pwm);
    manager.pwm.GenerateServo(manager.control_data.servo_pwm);
}
```

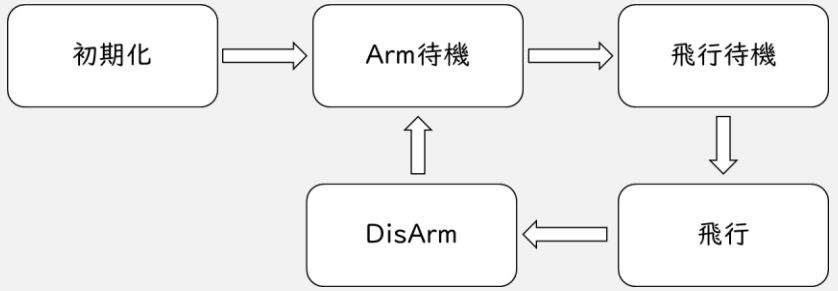
失敗したら次のループで再度判定

判定成功時の処理

サーボモーターの開閉処理（以後省略）

ファイルの場所：Core/Src/State/PreFlightState.cpp

5. 飛行待機処理



・飛行指示の待機

0. Arm、飛行スイッチのチェック
1. PID計算の初期化
2. Madgwickフィルターの初期化

・LEDをつけて飛行状態へ

Armが解除されたら、DisArmする

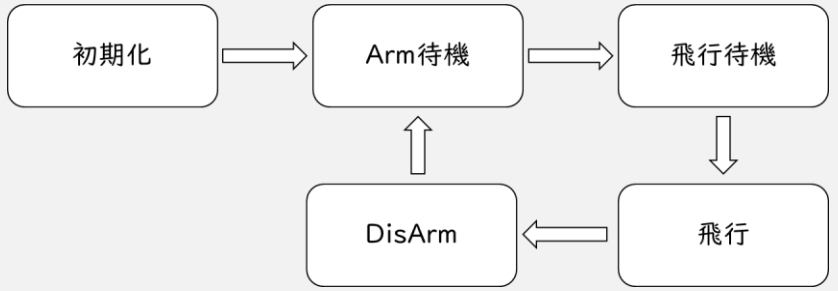
失敗したら次のループで再度判定

判定成功時の処理

```
void PreFlightState::update(FlightManager& manager) {

    // Armスイッチの判定
    if(!manager.sbus_data.arm){

        // Disarm状態に遷移
        manager.changeState(std::make_unique<DisarmingState>());
    }

    // Servo判定とPwm出力(abc_value = 0)
    manager.pwm.CalcServo(manager.sbus_data, 0, manager.control_data.servo_pwm);
    manager.pwm.GenerateServo(manager.control_data.servo_pwm);

    // Flightスイッチの判定
    if(manager.sbus_data.fly){

        //PIDの初期化
        // 各PIDインスタンスのリセット
        manager.angle_pitch.reset();
        manager.angle_roll.reset();
        manager.rate_pitch.reset();
        manager.rate_roll.reset();
        manager.rate_yaw.reset();

        //Madgwickフィルターの初期化(400hz)
        manager.madgwick.begin(UserSetting::MadgwickSampleFreq);

        //飛行用LEDをつける
        manager.green_led.Set(PinState::on);

        // 飛行状態に遷移
        manager.changeState(std::make_unique<FlyingState>());
        return;
    }
}
```

ファイルの場所：Core/Src/State/FlyingState.cpp

6. 飛行処理(1)



・センサーデータ取得部分

0. Armのチェック
1. 加速度・角速度データの取得
2. Madgwickで角度に変換
3. （センサー向きの調整）

次のページに続く

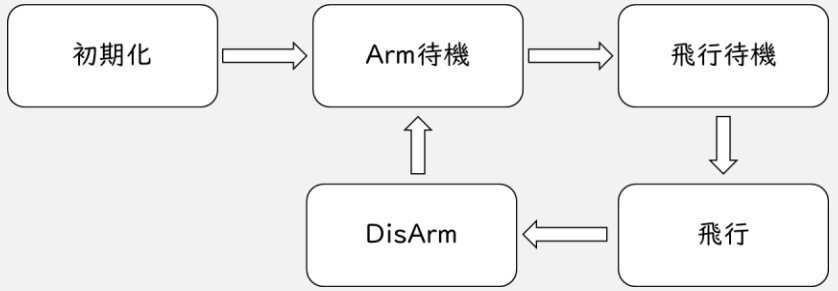
```
void FlyingState::update(FlightManager& manager) {

    // ループカウント（PIDの処理をするかを決定）
    static uint8_t loop_count = 0;
    loop_count++;

    // Armのチェック
    if(!manager.sbus_data.arm){
        manager.changeState(std::make_unique<DisarmingState>());
    }

    // センサーデータの取得
    manager.imuUtil->GetData(manager.sensor_data.accel, manager.sensor_data.gyro);

    // Madgwickフィルターでの姿勢推定
    manager.madgwick.updateIMU(
        manager.sensor_data.gyro[0], manager.sensor_data.gyro[1], manager.sensor_data.gyro[2],
        manager.sensor_data.accel[0], manager.sensor_data.accel[1], manager.sensor_data.accel[2]
    );

    // 推定データの取得
    manager.sensor_data.angle[0] = manager.madgwick.getPitch();
    manager.sensor_data.angle[1] = manager.madgwick.getRoll();
    manager.sensor_data.angle[2] = manager.madgwick.getYaw();

    // センサー向きの調整
    float buf = manager.sensor_data.gyro[0];
    manager.sensor_data.gyro[0]  = manager.sensor_data.gyro[1];
    manager.sensor_data.gyro[1]  = buf;
    manager.sensor_data.gyro[2] *= -1;
```

Armが解除されたら、DisArmする

Madgwickの計算とデータ取得

ファイルの場所：Core/Src/State/FlyingState.cpp

7. 飛行処理(2)



・センサーデータ取得部分(2)

時間がかかるため1ループ分ずれる

1. ADC（赤外線）の読み取り
2. ADCの停止
3. ADCの再開

次のページに続く

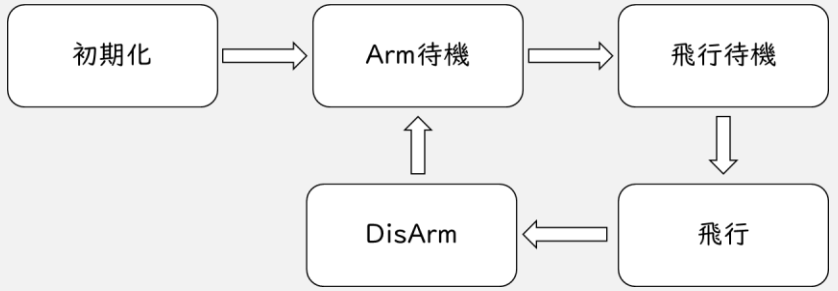
```
// 推定データの取得
manager.sensor_data.angle[0] = manager.madgwick.getPitch();
manager.sensor_data.angle[1] = manager.madgwick.getRoll();
manager.sensor_data.angle[2] = manager.madgwick.getYaw();

// センサー向きの調整
float buf = manager.sensor_data.gyro[0];
manager.sensor_data.gyro[0]  = manager.sensor_data.gyro[1];
manager.sensor_data.gyro[1]  = buf;
manager.sensor_data.gyro[2] *= -1;

//ADCX値の読み取り
manager.sensor_data.adc_value = HAL_ADC_GetValue(&hadc1);

//ADCのストップ
HAL_ADC_Stop(&hadc1);

//ADCのスタート
HAL_ADC_Start(&hadc1);
```

前ループの結果を読み取り

次ループ用の取得開始

ファイルの場所：Core/Src/State/FlyingState.cpp

8. 飛行処理(3)



・制御計算部分

1. 角度制御
スロットルの入力と現在角で計算
→計算結果は目標角速度

2. 角速度制御
センサー値と目標角速度で計算
→計算結果はモーター出力

```
//ADCX値の読み取り
manager.sensor_data.adc_value = HAL_ADC_GetValue(&hadc1);

//ADCのストップ
HAL_ADC_Stop(&hadc1);

//ADCのスタート
HAL_ADC_Start(&hadc1);


// 100hz 角度制御(pitch, roll)
if(loop_count % 4 == 0){
    // 目標角と現在角から目標角速度を計算
    manager.angle_pitch.calc(manager.sbus_data.target_value[0], manager.sensor_data.angle[0]);
    manager.angle_pitch.getData(&manager.control_data.target_rate[0]);
    manager.angle_roll.calc(manager.sbus_data.target_value[1], manager.sensor_data.angle[1]);
    manager.angle_roll.getData(&manager.control_data.target_rate[1]);

    // yaw軸はセンサーデータを使用
    manager.control_data.target_rate[2] = manager.sbus_data.target_value[2];
}
```

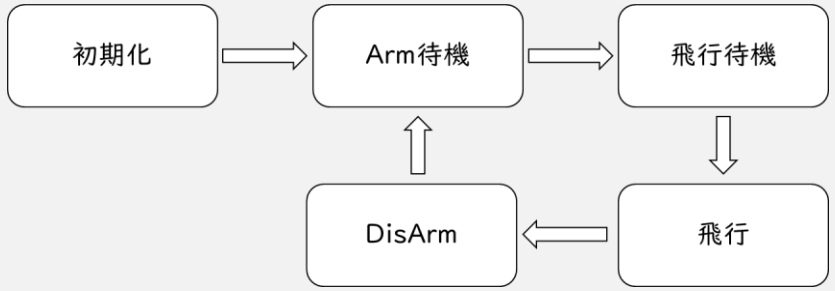
4回に1回角度制御の計算

```
// 400hz 角速度制御
//目標角速度と現在角速度(センサーデータ）から制御量を計算
manager.rate_pitch.calc(manager.control_data.target_rate[0], manager.sensor_data.gyro[0]);
manager.rate_pitch.getData(&manager.control_data.pid_result[0]);

manager.rate_roll.calc(manager.control_data.target_rate[1], manager.sensor_data.gyro[1]);
manager.rate_roll.getData(&manager.control_data.pid_result[1]);

manager.rate_yaw.calc(manager.control_data.target_rate[2], manager.sensor_data.gyro[2]);
manager.rate_yaw.getData(&manager.control_data.pid_result[2]);
```

毎回行う角速度制御の計算

ファイルの場所：Core/Src/State/FlyingState.cpp

9. 飛行処理(4)



・モーター出力部分

1. モーター出力の計算
2. サーボ出力の計算
3. 結果の出力

3軸PIDの結果とスロットルの足し算

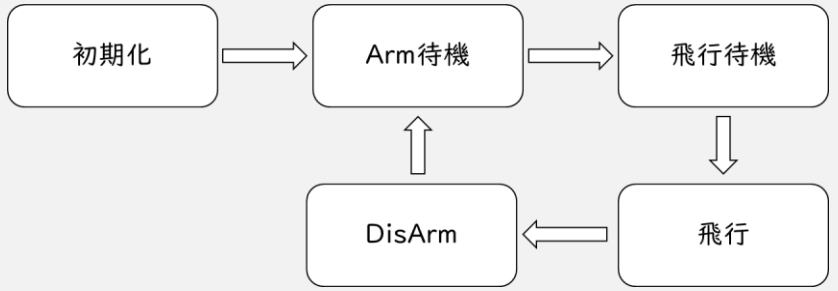
スイッチと赤外線からサーボの開閉を決定

モーターとサーボのpwmを出力

```
// PID結果を各モーターに分配
manager.pwm.CalcMotor(manager.sbus_data.throttle, manager.control_data.pid_result, manager.control_data.motor_pwm.data());

// Servoのpwmを生成
manager.pwm.CalcServo(manager.sbus_data, manager.sensor_data.adc_value, manager.control_data.servo_pwm);

// PWMを生成
manager.pwm.GenerateMotor(manager.control_data.motor_pwm.data());
manager.pwm.GenerateServo(manager.control_data.servo_pwm);
}
```

ファイルの場所：Core/Src/State/FlyingState.cpp

10. 飛行処理(5)

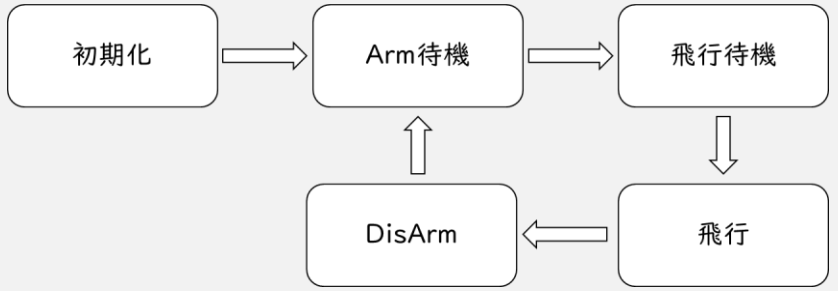


・飛行処理のまとめ

1. Armスイッチの判定（飛行終了を判定）
2. センサーデータの取得
3. センサーデータから角度を計算
4. 角度と角速度にPID制御
5. 制御結果をもとにモーターのPWMを計算
6. PWMを出力

ファイルの場所：Core/Src/State/DisArmingState.cpp

11. DisArm処理



・DisArm処理

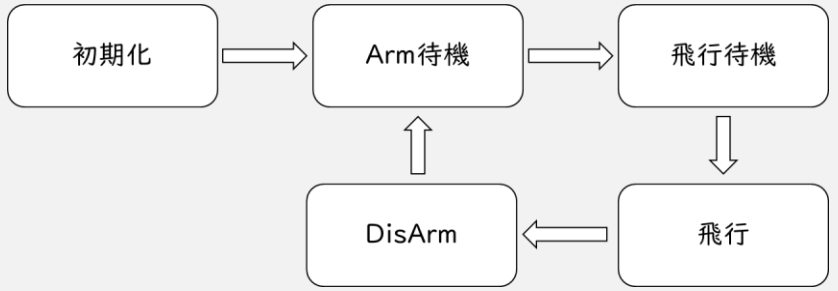
1. モーターの停止
2. PIDのリセット
3. Arm、飛行用のLEDを消す

```
void DisarmingState::update(FlightManager& manager) {

    // Pwmの停止
    manager.pwm.MotorStop();

    // 各PIDインスタンスのリセット
    manager.angle_pitch.reset();
    manager.angle_roll.reset();
    manager.rate_pitch.reset();
    manager.rate_roll.reset();
    manager.rate_yaw.reset();

    // LEDを消す
    manager.yellow_led.Set(PinState::off);
    manager.green_led.Set(PinState::off);

    // PreArmingへの遷移
    manager.changeState(std::make_unique<PreArmingState>());
}
```

・終了後Arm待機に戻る

ファイルの場所：Core/Src/State/FailSafeState.cpp

12. FailSafe処理



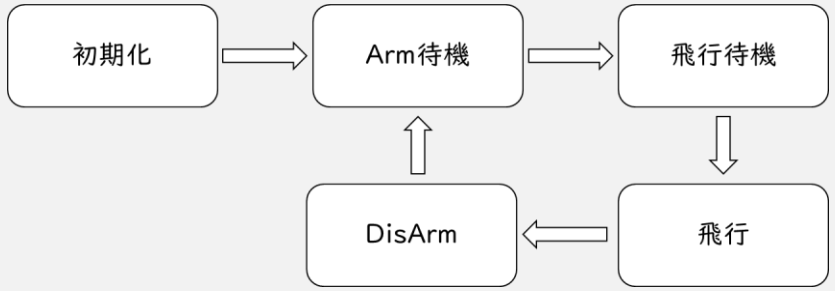
・フェイルセーフ処理

0. 通信の切断などの非常時に実行
1. モーターを止める
2. LEDを点滅

・復帰できたら危ないので無限ループ

```
void FailSafeState::update(FlightManager& manager) {

    // Pwmの停止
    manager.pwm.MotorStop();

    // PIDのリセット
    manager.angle_pitch.reset();
    manager.angle_roll.reset();
    manager.rate_pitch.reset();
    manager.rate_roll.reset();
    manager.rate_yaw.reset();

    // LEDの点滅
    while(1){

        for(volatile uint32_t i=0; i<1000000; i++);

        // Pwmの停止
        manager.pwm.MotorStop();

        manager.red_led.Set(PinState::toggle);
        manager.yellow_led.Set(PinState::toggle);
        manager.green_led.Set(PinState::toggle);
    }
}
```

13. おわりに



・処理の流れがわかった（かも）

・各処理で具体的なコードをどう書いたらいいかを考える
（LEDをつける、センサーとの通信、SBUSの受信などなど）

```
//LEDをつける
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
```

```
// I2C通信（書き込み）
HAL_I2C_Mem_Write(i2c_pin, i2c_addr, uint8_t(reg_addr), 1, tx_buffer, len, 1);
```

FCの作り方

おわり

次は、状態遷移の設計について勉強してみよう！