

ふぁうむうえあの歩き方

さくしゃ：う p 主

1. 趣旨

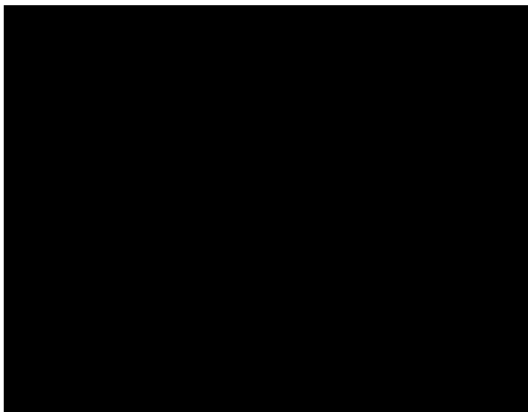
後に「**文書**」と呼ばれる本書の趣旨は、うp主製ファームウェアの設計意図を記したものになります。オブジェクト指向での設計となっていますが、オブジェクト指向云々に関しては後々説明できたら説明してきたいと思います。

また、ソースコードの中身の細かい処理まで説明する事は少ないと思います。まずは設計方針、設計概念を理解して頂くことを第一としています。このメソッドを理解できれば向かう所敵なしです。しかし、敵は味方の中にいるのも事実！

2. オブジェクト指向って？

うp主はC言語しか使った事ありませんが、概念はどの言語でも同じでしょう。**モノ (Object) に処理を実装する**。導入編はそんな感じで入った方が良いでしょう。色々なモノを詰め込みすぎると分けわからなくなります。

まずは下みたいな構成を考えてみましょう。絵がテキトーですみません。



これには以下の登場人物が存在します。

- バッテリー
- シリンダー
- リモコン
- 挟み込みセンサー

2.1. バッテリー

電源の供給の他、バッテリーの状態を I2C 通信で教えてくれるデキル奴。

2.2. シリンダー

電気を与えて、伸びる、縮む信号を与えてやると、棒が伸び縮みする不思議な棒。音がうるさいのがたまにキズ。

その他、これ以上、伸びない、縮まない場合に EndStop 信号で自動停止した事を教えてくれたり、動作中には棒が伸び縮みしている事を示す HallOut 信号（パルス）も出力してくれる。

2.3. リモコン

霊長類にバッテリー残量を教えてくれる LED や、シリンダーを伸ばすボタン、縮ませるボタンなどが搭載されているリモコン。

2.4. 挟み込みセンサー

シリンダー動作中に本センサーに反応が合った場合、シリンダーを停止させなければならない面倒くさい仕掛け。

2.5. ではコイツの仕様について

文字列による箇条書きで行くぜ。こう言うのを読み解いて、論理的に落とし込むのもお仕事の一つ。こう言うベンチャー的な仕事は案外外資で役に立つ事もある。と思う。あちらは上長から指示とかされないみたいだからな。では本題に戻ろう。

- バッテリーから取得した残量をリモコンについている LED を使って表示する
- リモコンの伸ばすボタンを押し続けている間、シリンダーが伸びる
- リモコンの縮むボタンを押し続けている間、シリンダーが縮む
- シリンダーが縮んでいる間に挟み込みセンサーが反応したらシリンダーを停止しなければならない

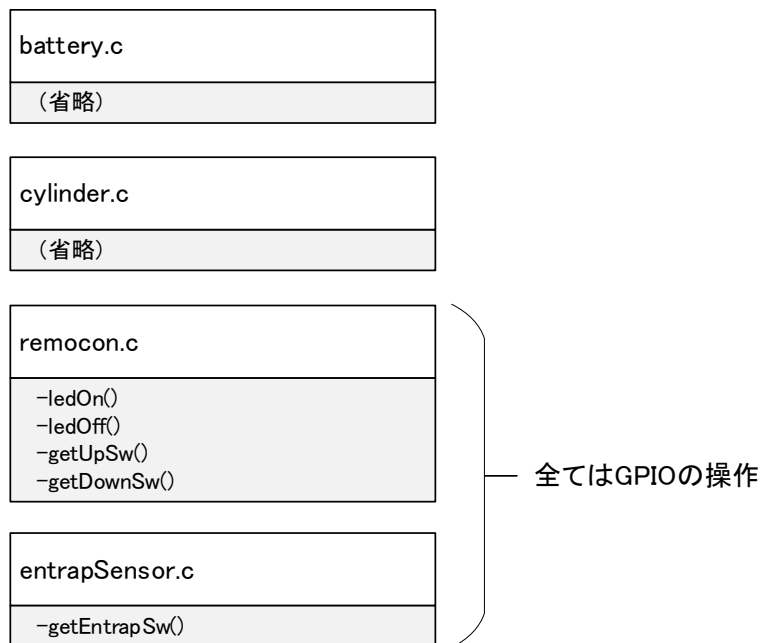
敢えてこんなモンだ。

登場人物の説明で上に書いてない機能を保有している人物（シリンダー）が居るが、それらの存在も忘れずに、詳細はロジックに落とし込む段階で自ら詰めて行かないとならない。

ただしこれは**ベンチャー流の場合だけ**な。車屋さんとかそういう所では仕様書に忠実に従うこと。あちらの世界ではスタンドプレーは厳禁だな。マジ怒られる。

2.6. じゃあどう実装する？

オブジェクト指向とはモノに処理を実装する事です。まあ深く考えずにやってみよう。まずは登場人物をクラスに仕立て上げる。う p 主は C 言語派なので battery.c cylinder.c remocon.c entrapSensor.c と
言う感じだな。う p 主の場合 ***.c がクラスに相当** するぜ。



でも上記のクラス分けには難点があると思うんだぜ。敢えて言うと remocon.c entrapSensor.c だな。

リモコンには LED1 つとボタンが 2 つ、挟み込みセンサーはセンサー1 つだけ。でも、ここでもう一つ考えなければならない事がある。ソースコードが**流用される**とした場合、どういうクラス分けにすれば良いのか？ 流用先にリモコンがあるとは限らない。挟み込みセンサーもあるかどうか分からんな。極端な話、冷蔵庫にでもエアコンにでも流用できる事を前提としたアーキテクチャだ。そういう事を考えて設計するとした場合、あまりに具体的過ぎるクラス設計では駄目だと思うぜ。オブジェクト指向の基本は具体化ではなく抽象化だからな。

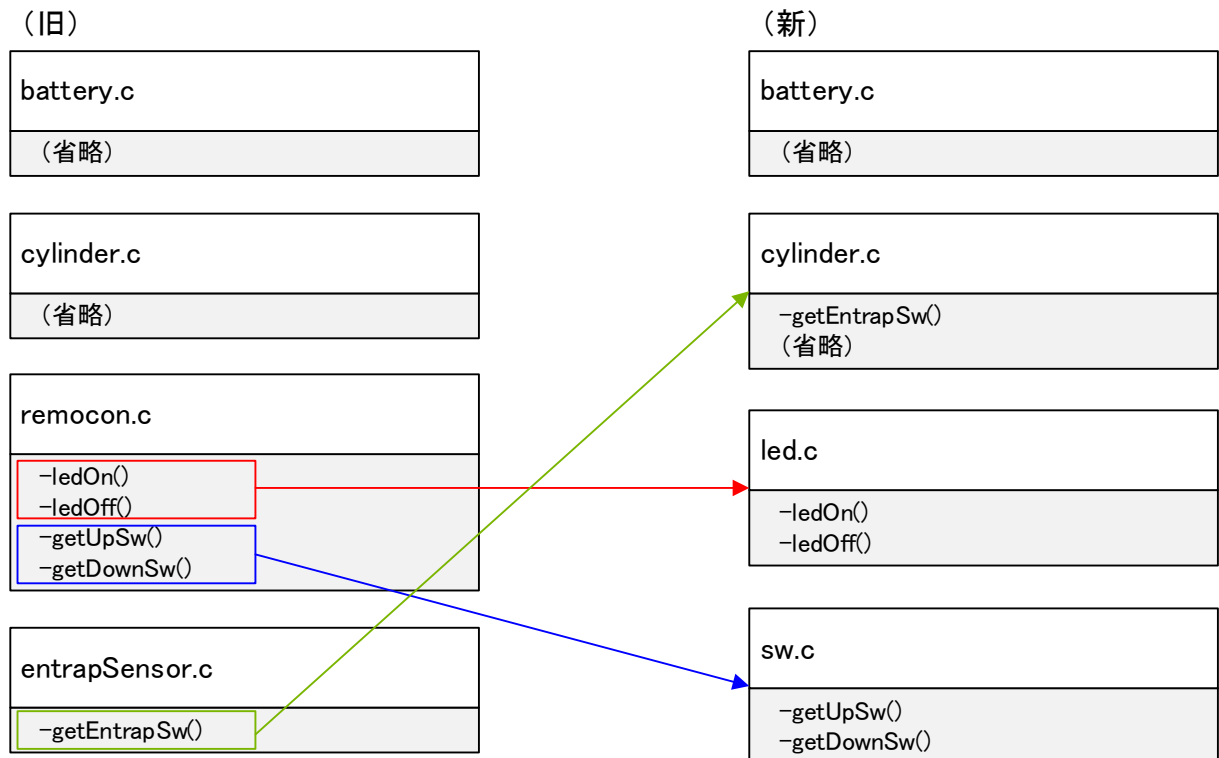
2.6.1. なぜ remocon.c entrapSensor.c が駄目なの？

この辺の考え方は十人十色だぜ。オブジェクト指向の設計は余程外れていない限り正解というモノも無い。皆が皆違う答えを出す。そういうモノだぜ。

ではなぜ駄目かと思ったかだが、流用先にリモコンが無かったり、挟み込みセンサーが無かった場合、当然、当該するクラスは不要になるので削除となる訳だが、remocon.c の中身の処理自体は GPIO の処理のみと仮定すれば、中身の処理自体は流用できる。しかし、クラス自体は削除されるため、他のクラスにコピーが必要だったり、ポーティングが必要になったりするわけだな。綺麗にポーティングできれば良いが、出来ない場合はクラス分け自体を根本から考え直す事になりかねない。

これでは、面倒くさい。

そういう訳なので、ちょっと直してみたぜ。



まず led.c と sw.c が増えたな。この理由は抽象化とカプセル化だ。LED なら他の機種として流用するにも違和感がない。LED って聞いて分からない人は少ないでしょ？ **LED に関する処理は全て led.c に詰まっていると考えろー。他の別のところに LED 処理が飛散している事は一切ない。** sw.c も同じ理由。

前の remocon.c の場合、LED の処理とスイッチの処理が混じっていて、見た感じ小汚いぜ。リモコンに何が搭載されているのかなど、イメージしづらい部分もあるからな。結局、流用の際は remocon.c の中身を見て必要な処理、不要な処理を選別する必要がある。こう言うのを事前に綺麗に部品化することで複雑な制御もシンプルに見える様になってくるんだぜ。

entrapSencsor.c に関してはだな、仕様のにも完全にシリンダーの動作に直結している。シリンダーが無くなれば要らない奴だ。なら、cylinder.c に入れてしまった方がシンプルだぜ。シリンダー駆動に関する処理は cylinder.c に詰まっていると考えろー。そして、シリンダーが不要な場合はクラスごとポイ〜。

要は**モノ (オブジェクト) に注目せよ**って事だよ。従来の手続き型設計とは考え方が全然違うよ。従来方法はモノではなく機能ごとに別れている感じだな。機能ごとにまとめてしまうと、必要なもの、不要なものがゴツチャゴツチャになっちゃうぜ。

2.7. その他、配慮しなければならないこと

[2.2 シリンダー](#)で説明している通り、このシリンダーさんは EndStop と HallOut の出力信号を持っています。その信号を受け取る側がどう使うかは別として、少なくとも [2.5](#) ではコイツの仕様についてではその信号を使用する事には触れられてはいません。

こう言う事に関する是非は回路も絡む話になってくるので、ファーム屋さんだけではどうしようもない話になりますが、製品をより良くするために有効なアイテムなのであれば、要求仕様になくても積極的に活用していく様な思考をオススメしたい。ただ、二度目になりますが、これはベンチャー流です。大手でも要求仕様書に手が回っていない場合はこの思考は重宝されますが、カッチカチの要求仕様書を書いてくる車屋さんとかで仕様書に記載の無い処理実装すると後が怖いことになります。

2.8. オブジェクト指向なんたらのがき

うp主がここで伝えたかった事は、

あっちこっちに影響の出る様な糞コードは書かないでね！

糞コードは書くは簡単。

しかし、流用性は無く、可読性も無く、リアルタイム性も無く、スループットも悪い。

それと忘れてはならない事を一つ、

その糞コードの尻ぬぐいをする人が必ず居ると言う事。

これを忘れてはならない。

（お前が言うなってやっだな。）

という訳で、オブジェクト指向うんたらってのはこれで終わりなー。

結局、最初の方にあった写真は何だったの？ とかには触れないでおくぜ。デリケートな問題ぜ。
うp主はよく布石を回収せずに次に行ってしまう癖があるぜ。

3. ようやくファームウェアの歩き方

前置きが長かったな。書かねーとかいいつつ、思った以上にオブジェクト指向について語ってしまった。まあここからも地味に続いて行くぜ。ここからは、具体的なコーディングに於けるテクニク的なものになっていきます。どういう概念の基に設計されているかを説明するぜ。

3.1. コーディング時に必要な気概、心構え

前述の通り、流用性、可読性、リアルタイム性が第一、スループットは第二として考えましょう。第一を心がければ、第二のスループットは勝手に備わってきます。そういうものです。

うp主も以前、あまりの糞コードのメンテに携わった時、当時の上長に作り直す旨の直談判をした事があります。作った本人が目の前に居るのに。本人曰く、

「スループットを最重要視したため、結果的に糞になってしまった。」(棒)
「やれるもんならやってみせろー」(棒)

などと語っており、たいへん嫌味も言われたモンです。

そして、まずは可読性を最重要視して作ってみた結果、スループットが **8 倍** にもなりグウの音も出さなかった事があります。が、相手が粘着質だったため、その後、事あるごとに粘着され続けることに・・・社会的にも問題になったものですよ。その後、その人は目下を辞めさせる事に注力していく様になりました。

ここで分かる事は、この業界は**常在戦場、背水の陣**でやらねば自分がやられる側になります。そういう心構えで仕事をしましょう。ちなみに、常在戦場、背水の陣はうp主の座右の銘です。嫌いな言葉は精神論です。

兎にも角にも、新参や弱小は小さい事から**ルールを変えて行く**しか勝ち目は無いのです。中には無能なんだけど声だけ大きい人(発言力のある人)も居るので、ルール改変も命がけ。正に常在戦場、背水の陣。

3.2. 設計書を書く際に必要な気概、心構え

前項を参考にすれば分かると思いますが、可読性の良いソースコードは設計書とイコールと言っても過言ではありません。書くのは理想だけど、必要なければ要らないと思います。必ず書けという話ではれば旧態依然の環境です。そういう新しい考え方にシフトできない環境は淘汰されていくと思うぜ。

これはうp主の考えですが、設計書は、今後コーディングを行う上での指針、設計方針、その他概要が記載されていれば十分との考えです。理由は、詳細まで詰めすぎるとコードに修正が入るたびに設計書へフィードバックしなければならないから。最後にまとめるとは良く聞くものの、大体はまとめられないまま、誰も読まない設計書の出来上がり。後任者は設計書など読まずにコードを解析した方が速いという訳です。

最後に大事なこと一つ、

- **見られないドキュメントなら書かない方がいいぞ。見せるんじゃない。魅せるんだ！**
- **まずは常識を疑え！**

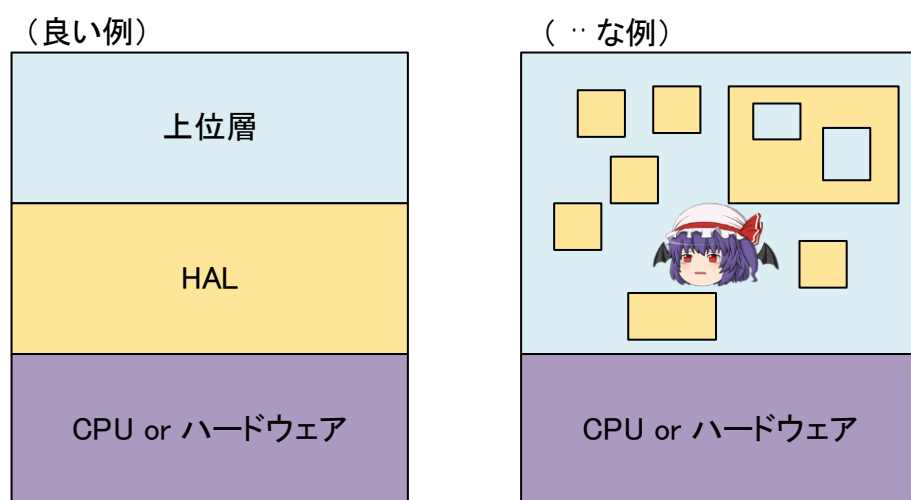
二つ目は特に重要だ。設計書に関係なく、常識を疑ってみるのも良いかもしれない。

3.3. 組み込みモノを作るに当たってのp主の矜持

これは組み込みモノに関わらない話にもなると思いますが、組み込みモノはしばしばマイコンが変わります。マイコンが変われば過去の資産（ソースコード）が全く使い物にならなくなる様な事では話になりません。

要は、如何にマイコン周りの制御と、上位層の制御をカッチリと分離できるかが鍵になる訳です。カッチリと分離できているのであれば、マイコンが変わっても変わるのはマイコン周りの制御のみで、上位層に影響を与える事はありません。

まず、マイコン周りの制御の事を **HAL (Hardware Abstract Layer)** と呼びます。分離方法について良い例と🐱（ダメ）な例を挙げてみます。



お分かり頂けたでしょうか・・・？

では、もう一度。

きゃあああ ああああ。
友達にまんじゅうをつもりが、なぜか
糞コードの迷路に迷ってしまったわ。(被害者)

初動で作りやすいのはダントツで🐱な例の方です。こちらは上位層、HAL 関係なしにメチャクチャやっています。ただし、中盤で行き詰まり、終盤ではデグレ[※]との戦いになります。また、移植性も可読性もあったものではありません。

対して良い例の方は、上位層と HAL がカッチリと分離しています。こちらの場合は初動にとっても時間が掛かりますが、言わずもがな、最終的に良い成果物が出来上がるのはこちらの方です。また、HAL はマイコンが変わらない限り流用が可能なため、流用する際は初動にも時間が掛からずにジョリーグッド！

※ デグレとは、不具合を修正してみたら、もともと不具合ではなかった部分が不具合化してしまう現象。モグラ叩き状態。車屋さんなどでやっちゃった場合は最低な扱いを受ける。もともと発生しづらい不具合が新規に見つかった場合と比べ、情状酌量の余地は少ない。

3.3.1. HAL についてもっと詳しく！

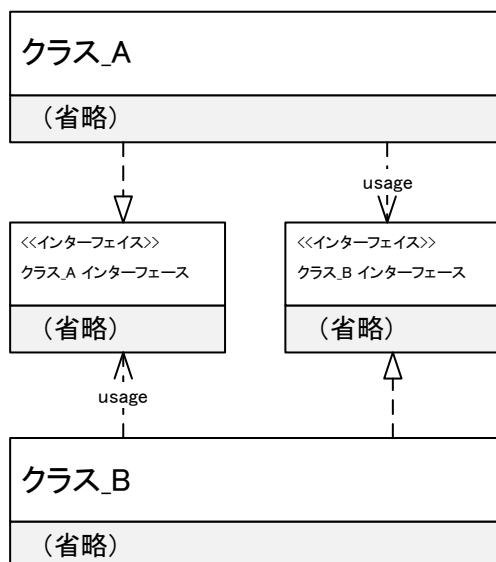
マイコンやハードウェアと直接関わっているリソースの事です。ファームウェアはマイコンに内蔵されているリソースを動作させるため、マイコン側で用意してある**レジスタ**を操作してマイコン、又は、ハードウェアのリソースを動作させます。この一連のレジスタを操作する処理をまとめたものを HAL と呼びます。

また、HAL はハードウェアと上位層を仲介するクッションの様な存在で、CPU or ハードウェアと上位層の互換性の違いを吸収するレイヤーでもあります。

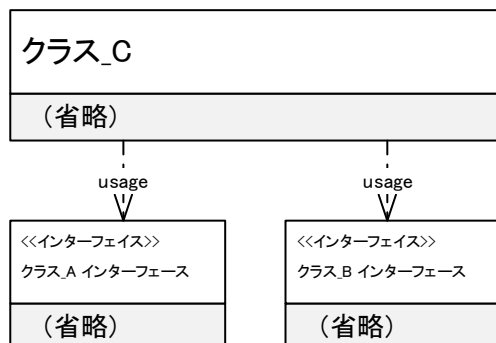
尚、ラmp主はアドレスのマッピングも HAL の一部と捉えています。上位層にはマッピングすら意識させたくありません。とにかく、上位層にはハードウェアを何一つ意識しなくても良い設計にしなければならないと思っています。

3.3.2. 各クラス間を繋ぐためのラmp主ルール

クラス間の繋がりはインタフェースを使用してリンクさせます。インタフェースの良い所は、継承とは別の概念だからです。下記例で言うと、クラス A とクラス B はお互いのインタフェースを用いてリンクさせていますが、例えばクラス B をクラス C が継承した場合、クラス C から見えるのはクラス B までです。クラス A は見えません。これは C++ ではなかなか出来ない概念です。できるのか？（震え声）



ちなみに、継承させたい場合はこんな感じにすれば、クラス A もクラス B も両方継承できます。



両方のインタフェースとリンクすればいいだけだね。実現の線は省略しているけど、一つ上で説明しているからモーマンタイ。こういう書き方は普通にアリです。

また、このインタフェースには他にも隠し技を残しているので楽しみに！

3.4. ようやくソースコードの構成

下のコピペの様に感じになっているね！

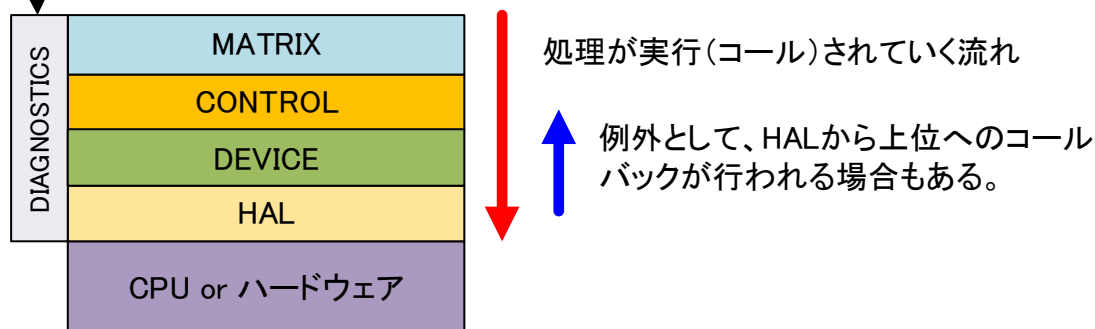
DIAGNOSTICS とかはかなりテキトーに名前付けてしまった。でも、この辺りからかなり重要なんだよな～。この辺りからしっかりしていかないと、糞コード街道まっしぐら。

名前	更新日時	種類	サイズ
COMMON	2017/12/21 10:07	ファイル フォルダー	
CONTROL	2017/12/21 10:07	ファイル フォルダー	
DEVICE	2017/12/21 10:07	ファイル フォルダー	
DIAGNOSTICS	2018/02/16 14:59	ファイル フォルダー	
HAL	2017/12/21 10:23	ファイル フォルダー	
INTERFACE	2018/02/15 9:31	ファイル フォルダー	
MATRIX	2017/12/21 10:07	ファイル フォルダー	
typedef.h	2017/11/24 11:28	H ファイル	1 KB

構成のブロック図的には以下の様になっているぜ。

COMMON が無いのは、COMMON はアルゴリズムであって実態が無いのよ。誰かにインスタンスとして生成されるまでは命が宿っていないので、下のブロック図には書かない事にしたぜ。

危険な成分



ちなみに、ハードウェアの差分を吸収する HAL を仲介しているため、HAL より上の DEVICE、CONTROL、MATRIX に関しては CPU の変更などがあっても、そのまま流用する事ができます。極端な話、パソコンのアプリに流用も可能です。

それぞれ接点がある部分に関わり合いながら動く構造になります。接点が無い者同士（例えば、MATRIX と HAL）の関連付けはできるけど、**とりあえず駄目です！**

一度許すと糞コード街道まっしぐらになるため、許したくありません。とは言え、上記も完成されたものではないので、良くなる方向への改変はアリです。**場当たり的な対応ならばナシ**です。

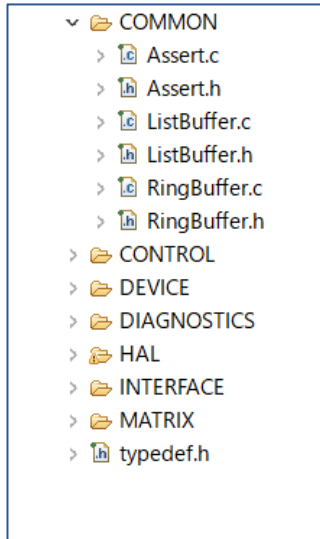
ちなみに、**例外とされているコールバック**について説明します。これは HAL での割り込み要因を上位側で受け付けるための仕組みです。上位側が割り込み待ちで停止してしまう事は良い事ではありません。従って、コールバック関数の登録、コールバック関数の呼び出しの流れに関しては、例外として上記ブロック図通りに従わなくても良いです。

それでは、ひとつひとつフォルダの中身を覗いていきましょう。

3.4.1. COMMON

役割としては、各機種共通に利用できる資産であり、組み込みモノでも PC のアプリでも関係なく、共通で利用できる親クラスに相当するクラスが並んでいます。だもんで、このクラス群単品では役に立ちません。ここにある親クラスは継承されて初めて機能するものです。ちなみに、この継承もルールから漏れずインタフェースでの実現となります。

共通で利用できる、なんかいい感じのアルゴリズムのコードを書いたのなら、ここに放り込んでおくといいぜ。



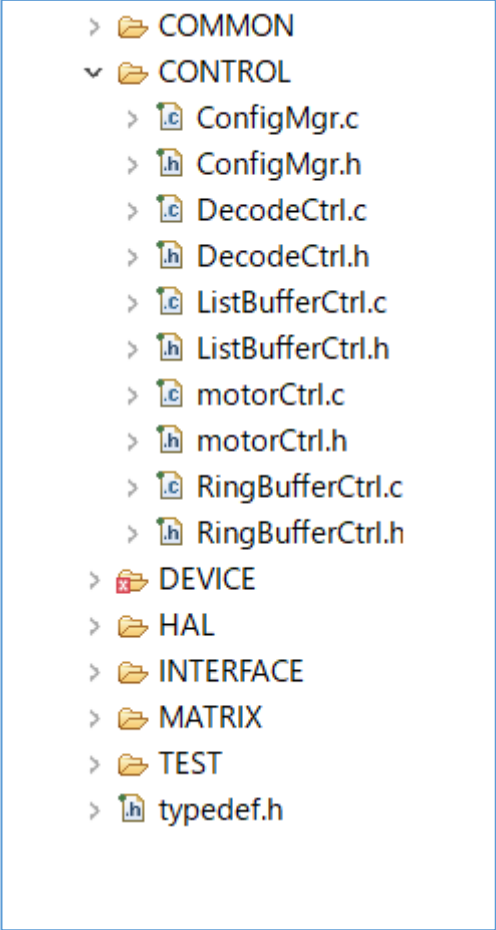
各ソースファイルの説明は省略するよー。ソースコード直接見てちょんまげ。



















ちなみに COMMON はアルゴリズムの実態のみなので、[3.4](#) ようやくソースコードの構成のブロック図のどこで利用しても問題ナシとの認識です。大体は CONTROL で使用される事になると思いますが。

3.4.2. CONTROL

[3.4](#) ようやくソースコードの構成のブロック図を見てくれだぜ。CONTROL は DEVICE を呼ぶ側に位置している。従って、DEVICE のクラスとの繋がりを持つファイル構成となっているぜ。

COMMON にあったアルゴリズムはここでインスタンスが作られる事になる場合が多くなると思います。COMMON から派生したっぽい名前のファイルがチラホラありますね。



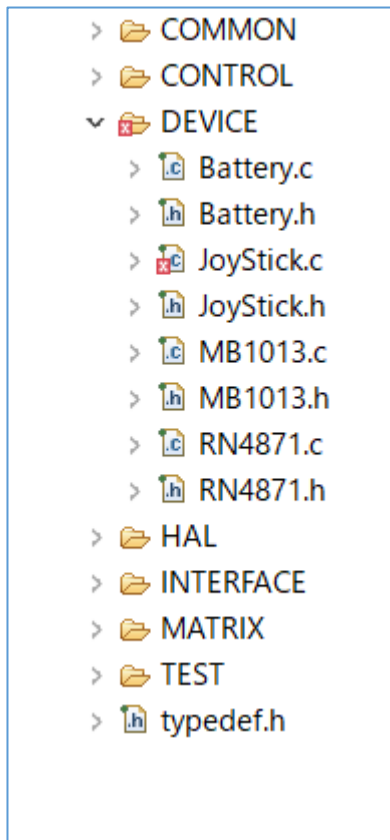
- >  COMMON
- ▼  CONTROL
 - >  ConfigMgr.c
 - >  ConfigMgr.h
 - >  DecodeCtrl.c
 - >  DecodeCtrl.h
 - >  ListBufferCtrl.c
 - >  ListBufferCtrl.h
 - >  motorCtrl.c
 - >  motorCtrl.h
 - >  RingBufferCtrl.c
 - >  RingBufferCtrl.h
- >  DEVICE
- >  HAL
- >  INTERFACE
- >  MATRIX
- >  TEST
- >  typedef.h

3.4.3. DEVICE

DEVICE にはその名の通り、デバイス制御、部品の制御を行うレイヤーになっているぜ。

ファイル名は抽象化された名前や、具体的な部品名などにしてある。本来ここには具体的なファイル名になっているのが望ましいが、デバイスそのものがどの部品でも制御が同じ様な場合は抽象的なファイル名にしてある。

例えば、MB1013.c は超音波センサーの制御、RN4871.c は Bluetooth LE の制御という感じである。ファイル名は部品の品番です。部品に変更があれば新しい品番の制御を作ります。HAL との癒着は無い仕組みなので、一度作った品番の制御は別のモデルにそのまま流用する事ができます。



3.4.4. HAL

ハードウェアと上位層の間に存在する緩衝地帯。これの良し悪しが流用性に大きく関わってきます。DEVICE に提供する API の引数は今後を見越した上での柔軟性の高い引数にしておく事が肝要です。しかし、先の事は分からないため、void 型ポインタとしておいて、後から上位と HAL で調整できる引数を作っておくのもアリだと思います。

HAL の直下には CPU の型番のフォルダを作り、そこに CPU のリソースと用途にちなんだ名前のソースファイルを作ります。CPU が変更になった場合は、HAL の直下に新しい CPU のフォルダを作り、そこに HAL を実装していくスタイル。

```
>  📁 COMMON
>  📁 CONTROL
>  📁 DEVICE
>  📁 HAL
    >  📁 STM32L073xx
        >  📄 ADC.c
        >  📄 ADC.h
        >  📄 EEPROM.c
        >  📄 EEPROM.h
        >  📄 GPIO.c
        >  📄 GPIO.h
        >  📄 I2C1.c
        >  📄 I2C1.h
        >  📄 TIM21_InputCapture.c
        >  📄 TIM21_InputCapture.h
        >  📄 TIM6_CyclicTimer.c
        >  📄 TIM6_CyclicTimer.h
        >  📄 TIM7_FreeRunTimer.c
        >  📄 TIM7_FreeRunTimer.h
        >  📄 USART.c
        >  📄 USART.h
    >  📁 INTERFACE
    >  📁 MATRIX
    >  📁 TEST
    >  📄 typedef.h
```

3.4.5. INTERFACE

3.3.2 各クラス間を繋ぐための p 主ルールの通り、クラス間の繋がりにはインタフェースの概念を使用します。

```
> COMMON
> CONTROL
> DEVICE
> HAL
▼ INTERFACE
  > Adc_INTERFACE.h
  > Battery_INTERFACE.h
  > ConfigMgr_INTERFACE.h
  > CyclicTimer_INTERFACE.h
  > DataLinkLayer_INTERFACE.h
  > DecodeCtrl_INTERFACE.h
  > DistanceSensor_INTERFACE.h
  > Eeprom_INTERFACE.h
  > Event_INTERFACE.h
  > FreeRunTimer_INTERFACE.h
  > Gpio_INTERFACE.h
  > I2C_INTERFACE.h
  > InputCapture_INTERFACE.h
  > JoyStick_INTERFACE.h
  > ListBuffer_INTERFACE.h
  > ListBufferCtrl_INTERFACE.h
  > MotorCtrl_INTERFACE.h
  > RingBuffer_INTERFACE.h
  > RingBufferCtrl_INTERFACE.h
  > Usart_INTERFACE.h
> MATRIX
> TEST
> typedef.h
```

このインタフェースは API 関数の型を定義したポインタの集まりです。Adc_INTERFACE.h を参考にしてみましょう。

```
typedef struct
{
    void (*init)(void);
    void (*stop)(void);
    void (*start)(void);
    unsigned short (*getADC)(const unsigned long ch);
} Adc_INTERFACE;

extern Adc_INTERFACE AdcCtrl;
```

上記の様に全てポインタを介しています。**これは p 主が強い拘りを持っているところです。**ポインタを介すことで、C 言語でも文法がオブジェクト指向言語の様に振る舞えることと、動的にポインタ自体を書き換える事で、通常とは別の振る舞いをさせる事も可能なのです。オブジェクト指向ではポリモーフィズムとも呼びますね。

この辺は集中力続いていたら後々説明いたします・・・

ちなみに、上記例の AdcCtrl の実現元は HAL の ADC.c に存在します。上位層で ADC を使いたい場合は、使いたいクラスで Adc_INTERFACE.h をインクルードするだけで AdcCtrl を使用する事ができます。

3.4.6. MATRIX

大事な要素なのにフォルダ順に説明して居たら、結構最後の方に来てしまった。うp主流アーキテクチャはマトリクスでのイベント・ドリブンで動作します。ちなみに、このマトリクスはバックグラウンドで動作します。

うp主流とか言ってるけど、はっきり言ってありきたりな手法だぜ。

大抵の事はバックグラウンドで実行する事になります。割り込みは必要最低限のみです。

3.4.6.1. イベントってどうするんですか？

イベントは定義したイベントをリングバッファにアタッチすれば、後はバックグラウンド処理でリングバッファをチェックし、イベントが格納されていたら、マトリクスにイベントぶち込んで実行させる感じなんです。

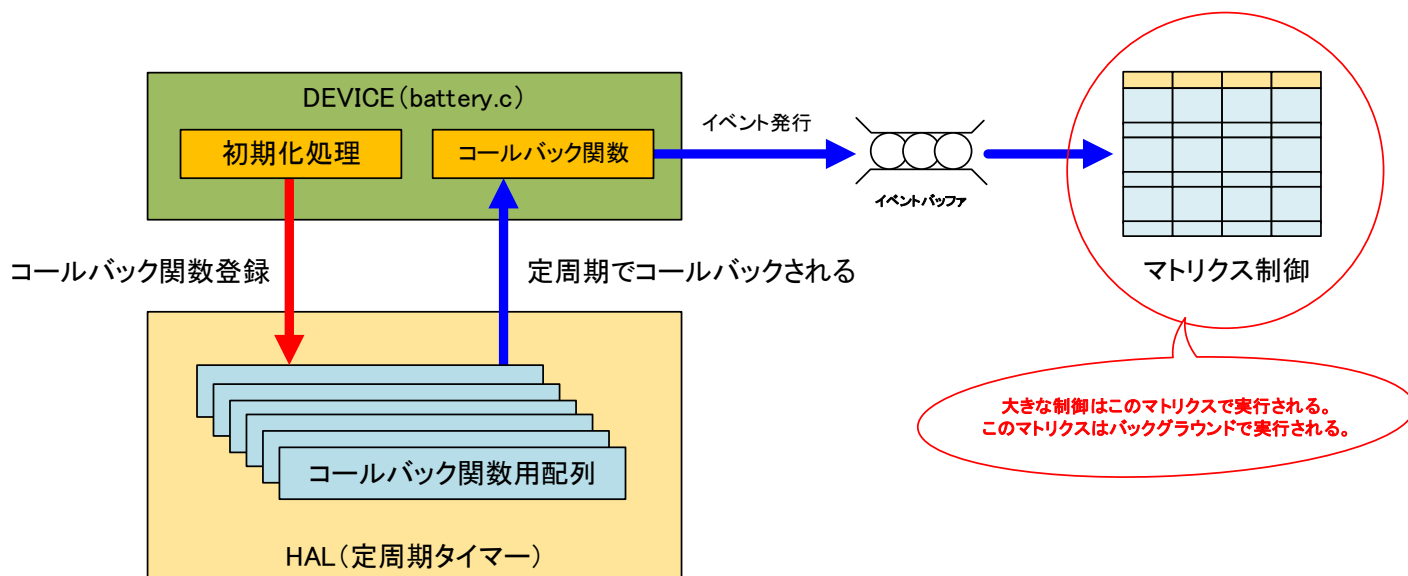
じゃあイベントはどうやって発行するの？

イベントを発行するにも、タイミングがなければ発行できませんよね？

基本このマトリクスは定周期タイマを利用してイベントを作っていくスタイルです。**イベントはそのイベントに関わるクラスが自分自身で発行するルール**です。同じイベントをいたる所で発行させてしまうと、これもまた糞コード街道まっしぐらなので。**処理やイベントやその他が、関係する一つのクラスにまとまっている事**こそが可読性を良くさせます。

例えば、バッテリーの情報を取得する/DEVICE/battery.cがありますが、これは1秒周期でバッテリー情報を取得しに行きます。ではこの1秒はどうやって作るのか？

これは3 [よう](#)やくファームウェアの歩き方の図で触れたコールバックを利用しています。各クラスの初期化処理にて、定周期タイマ割り込み発生時に実行して欲しいコールバック関数を登録します。登録したコールバック関数は定周期でコールされる事になるため、このタイミングを利用してイベントの発行を行います。



ちなみに、バックグラウンドとは空いた時間の事を言いますね。要は、大きい処理はCPUの空いた時間で実行しているのです。大きい処理を割り込みで実行していたらリアルタイム性損ないますもんね。

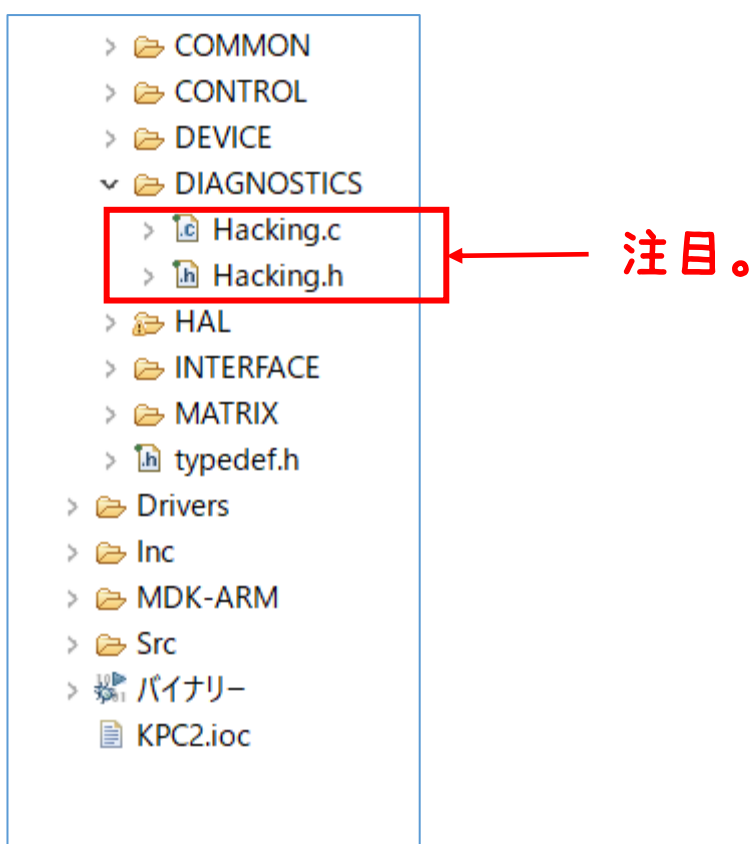
3.5. 危険な成分って結局なに？

DIAGNOSTICS の事です。ファイル名もヤバ目ですね。自己診断の事なんですが、方法がヤバ目なだけです。上の方でポリモーフィズムについて触れました。要は API のポインタを書き換える事で振る舞いを強制的に変えてしまう事です。自己診断にも使えるし、自動耐久試験にも使えるでしょう。

任意の条件下で Hacking.c の持つハッキングプログラムのアドレスで、ターゲットの API のポインタを書き換える事で実現します。

一部の API の振る舞いを変えるだけなので、周辺への影響は極めて軽微。また、他の API は通常通りに動作するため、自己診断モードに入った事すら分かせない所に**趣を感じます**。いいね！👍

ファームウェア的にも自己診断時の特別なルートが動作する訳ではないので、ファームウェアの耐久試験にもなり得るわけです。



うp主はまだ確立されていないこの方法について無限の可能性を感じるのです。オブジェクト指向言語のそれと違い、動的に変更可能。言わば Hacking.c は DLL みたいなものですよ。

この DLL みたいな感じを大事にしていきたいと思います。

おわり

4. APPENDIX

本書には以下のサイトよりフォントを無償使用させて頂いております。

JK ゴシック M

<http://font.cutegirl.jp/jk-font-medium.html>

うずらフォント

<https://fontmeme.com/jfont/uzura-font/>

本書には以下のサイトよりゆっくり画像を無償使用させて頂いております。

<http://www.nicotalk.com/charasozai.html>