

## パターン ライブラリー

## 目的

ET ロボコン競技には、毎年必ず使用する技術要素が数多くあります。基本ルールは変わらず、難所が変化するためです。

私たちは「**定型パターンを毎年実装することは無駄である**」と考え、「ETロボコン特化フレームワーク」の内部にそれら定型パターンの実装を貯蔵したライブラリーを設計しました。

**抽象度の高いパターン部品を提供することで、アプリケーションでは走行戦術を考える事に集中できるようになります。**

## 機能

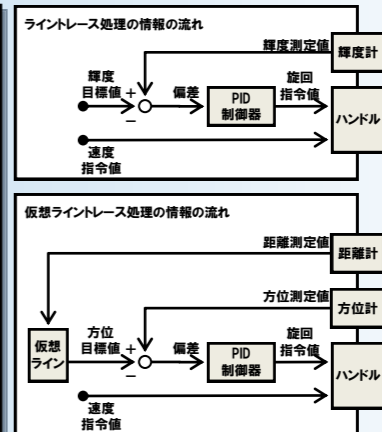
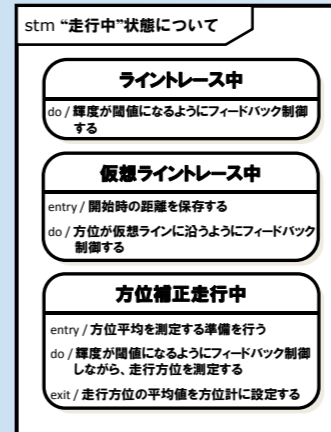
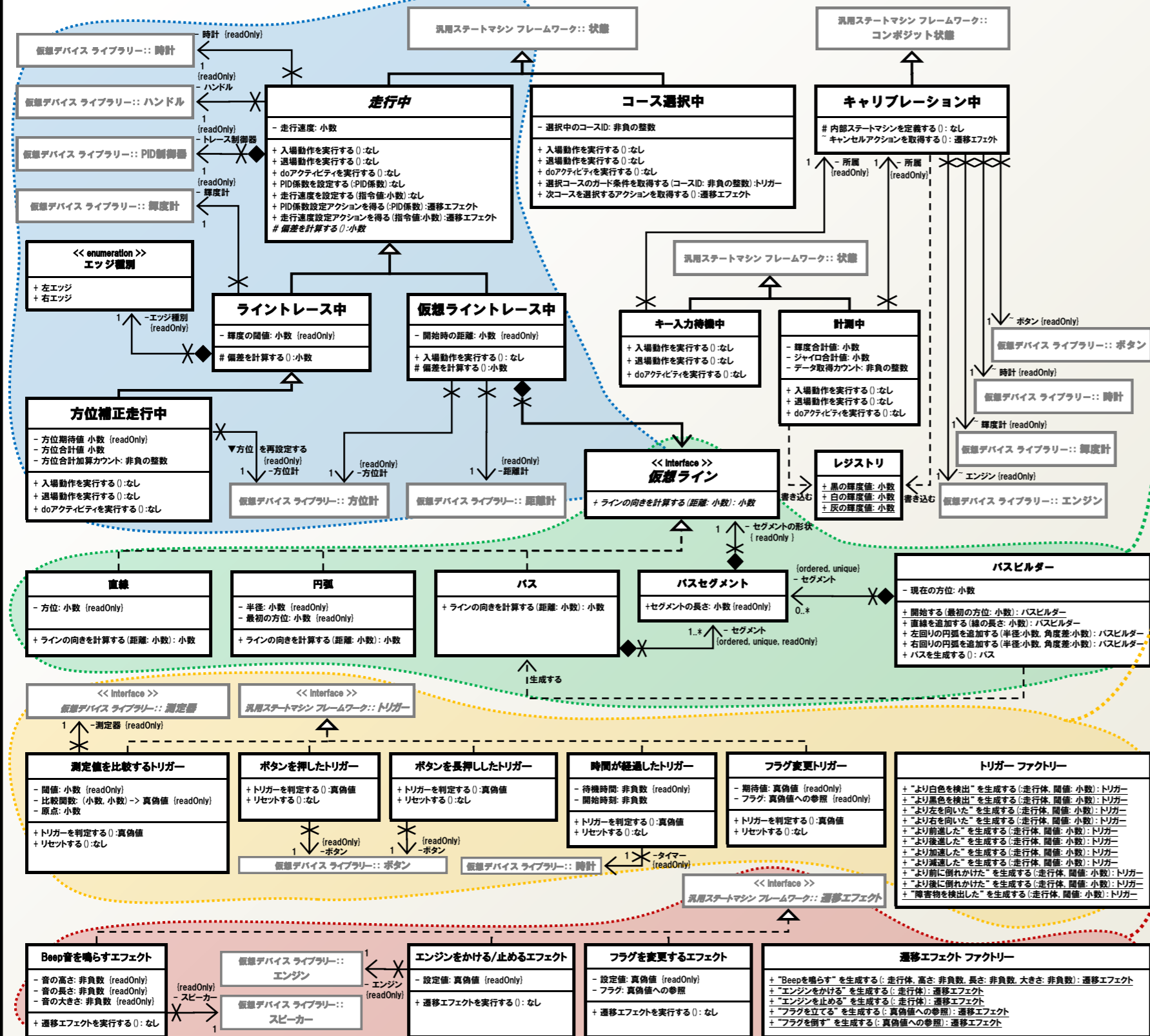
“パターン ライブラリー”は、C++ 言語の上で、次に挙げる 4 つのフィーチャを提供します。

- ① “ラインに沿って走る”などの定型な「状態」を提供する
- ② “指定距離走りしたら通知する”などの定型な「トリガー」を提供する
- ③ “Beepを鳴らす”などの定型な「遷移エフェクト」を提供する
- ④ “直線”“円弧”などの仮想ラインを扱う仕組みを提供する

直接走行と関係のない、ログ収集やコマンド受信に関するクラスは、紙面の都合で省略しています。

実際に、フレームワークは開発の初期に完成し、以降は非常に安定していました。走行戦術を考える段階に入ってからフレームワークの中を書き換えたのは、ジャイロ値の測定器とトリガーを追加した1回だけでした。

## package パターン ライブラリー



“走行中”状態は、走行パターンを考える上で、最も基本となる状態です。“走行中”状態はラインに追従するようフィードバック制御を行います。追従するラインの種類によって 2 つに大別されます。

## ◆“ライントレース中”状態

コースに描かれた黒いラインを追従します。  
目標値: 白と黒の境目の輝度値  
測定値: 輝度計の値

## ◆“仮想ライントレース中”状態

データ上のラインを追従します。ラインは、後述する「仮想ライン」クラスのインスタンスで表されます。  
目標値: 仮想ラインが計算した方位  
測定値: 方位計の値

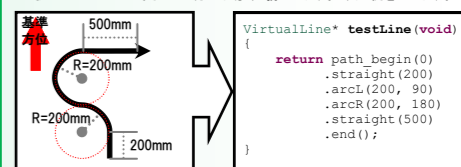
## ◆“方位補正走行中”状態

特別な“ライントレース中”状態です。コースに描かれたライン(直線部分!)を走りながら、そのラインの向きで方位計をリセットします。

仮想ラインと仮想ライントレースは、コース上のラインを無視して自由自在に走行するための仕組みです。仮想ライン自体は、現在の走行距離を元に、走行すべき方位の目標値を計算します。そして、“仮想ライントレース中”状態は、方位計の値が目標値に近づくように制御します。フレームワークが提供する仮想ラインの実装は、次の 3 種類です。

- ◆ 直線(d) := 方位
- ◆ 円弧(d) := 最初の方角 + K \* d / 半径 (※ K = タイヤ径や車幅から算出した係数)
- ◆ パス(d) := { 直線(d) と 円弧(d) とのシーケンス }

フレームワークは、仮想ラインを定義するための小規模な DSL を用意しました。特に、パスの定義に Fluent Interface Pattern を用いたことで、直線と円弧のシーケンスであることが分かり易く定義できます。次に例を示します。



前ページで紹介した測定器などから情報を取得し、閾値を超えた (or 下回った) 時に励起するトリガーの実装を提供します。個々は非常にシンプルですが、(前ページで紹介したように)トリガーは AND, OR 等で合成できるため、簡単に複雑な条件のトリガーを生成できます。

例えば「強制終了 := 100mm 走行した後で、ボタンを押下するか長押しされた」というのは、コード上では右のように表せます。

```
Trigger tShutdown(void) {
    return tForwardThan(100) && (tButtonPressed() || tButtonLongPressed());
}
```

前ページで紹介した仮想デバイスを用いた処理や、フラグを操作する処理を行う遷移エフェクトの実装を提供します。

去年はフレームワークの使い方を理解してもらうのに時間がかかった

開発の基本はステートマシン定義  
今年の主力は新人

フレームワークを使う敷居が下がる

調整スピードが速くなる!

実装と実験の回転が速くなる!

開発スピードが速くなる!

この ET ロボコン特化フレームワークでは、宣言型のプログラミング手法を採用しています。走行パターンを表すステートマシンを宣言すると、フレームワークが、その通りに走行体を動かしてくれます。

## 命令型のプログラミング手法

目的を達するためのアルゴリズムや手順を記述する手法

## 宣言型のプログラミング手法

目的そのものを記述する手法

宣言型のプログラミング手法は、命令型の手法に比べて **抽象度がとても高く、分かりやすいコードを書くことができます**。さらに、“目的”と“手段”を分離することによって裏側で **使える最適化技法の幅が大きく広がり**、「技術的に詳しい人が“手段”を記述し、そうでない人が“目的”を記述する」というような分業ができるようになります。

(例えば、SQL は宣言型の RDB 操作に関するドメイン特化言語として、その特性を存分に活かしています)

他方、フレームワークや言語そのものが“手段”の情報を持っていないため、複雑なフレームワークや言語になってしまいます。また、“手段”が特定できないほど汎用的な事柄に関しては、そもそも宣言型のプログラミング手法は使えません。

※ 目的と手段とは、相対的な言葉であることに注意が必要です。

以上の特性から、ドメイン特化言語には宣言型のプログラミング手法こそが相応しいと言えます。

このフレームワークでは、「左に180度曲がったらシーソー攻略に移る」等といった粒度で“やりたいこと”を宣言していきます。右に、簡単な例を示します。

```
// まずラインの左側をライントレースする。
// 左に90度曲がったところで Beep を鳴らして、ラインを無視して帰ってくる。
// (backLine() の戻り値は帰ってくるための仮想ラインを返す)
SRunning* sLineTracing = sEdgeTracing(1.0F);
SRunning* sComingBack = sVirtualLineTracing(backLine(), 1.0F);

StateMachine sub = defInitial( sLineTracing )
    .def( sLineTracing to sComingBack when tLeftThan(90) )
    .def( sComingBack to sFinal() when tButtonPressed() with aBeep() )
    .toStateMachine();
```

また、コンポジット状態の定義を上手に使うことで、さらに宣言の抽象度を増すことができます。

## ET ロボコン 2010 走行戦術



走行タイムを減らす為に…

- ★ 必要のない停止は行わない！
- ★ 実ラインより仮想ライン！

方位計 誤差抑制の為に…

- ★ 急旋回を行わない！
- ★ 直線の実ラインなるべく方位補正を行う！

リザルトタイムを減らす為に…

- ★ 難所にすて挑戦する！

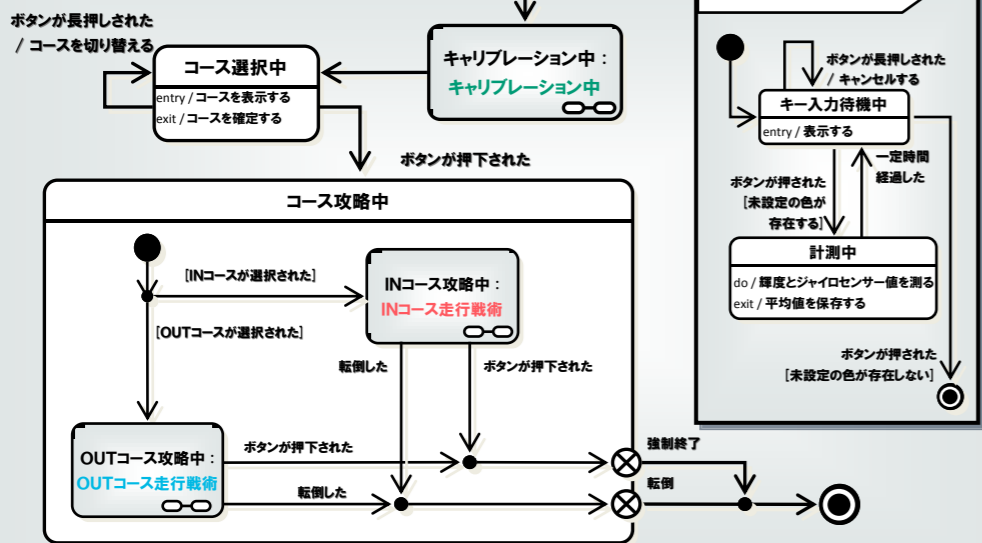
## 機能

“ET ロボコン 2010 走行戦術” が提供する機能は、ただひとつ。

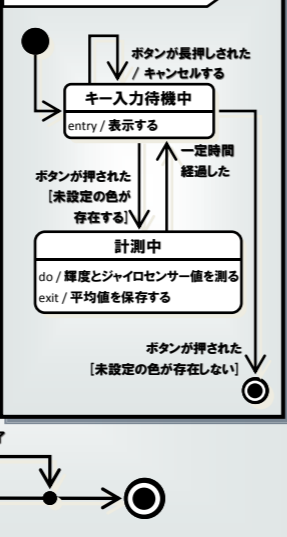
即ち、

2010年度のETロボコン コースを走破する！

## stm 走行戦術



## stm キャリブレーション中



## stm INコース走行戦術

**エニグマ解読 要点**

走行速度向上ポイント

- ★ パリティビットは読まない。(探索精度向上で検出失敗を防ぐ。)
- ★ ショートカットし、走行距離を減らす。

探索精度向上ポイント

- ★ 衝立の面に対して斜めに探索しないような仮想ラインを選定。
- ★ 面に対して正面から探索することで検出失敗を防ぐ。
- ★ 衝立付近にいるときのみ衝立の有無の検出を実行する。
- ★ 衝立付近に検出範囲を絞ることで誤検出を減らす。

調整スピード向上ポイント

- ★ 衝立の有無に関係なく同じコース(仮想ライン)を走る。

パターン	衝立a	衝立b	衝立c	通過ゲート
1	なし	なし	あり	A→C
2	なし	あり	なし	A→D
3	あり	なし	なし	B→C
4	あり	あり	あり	B→D



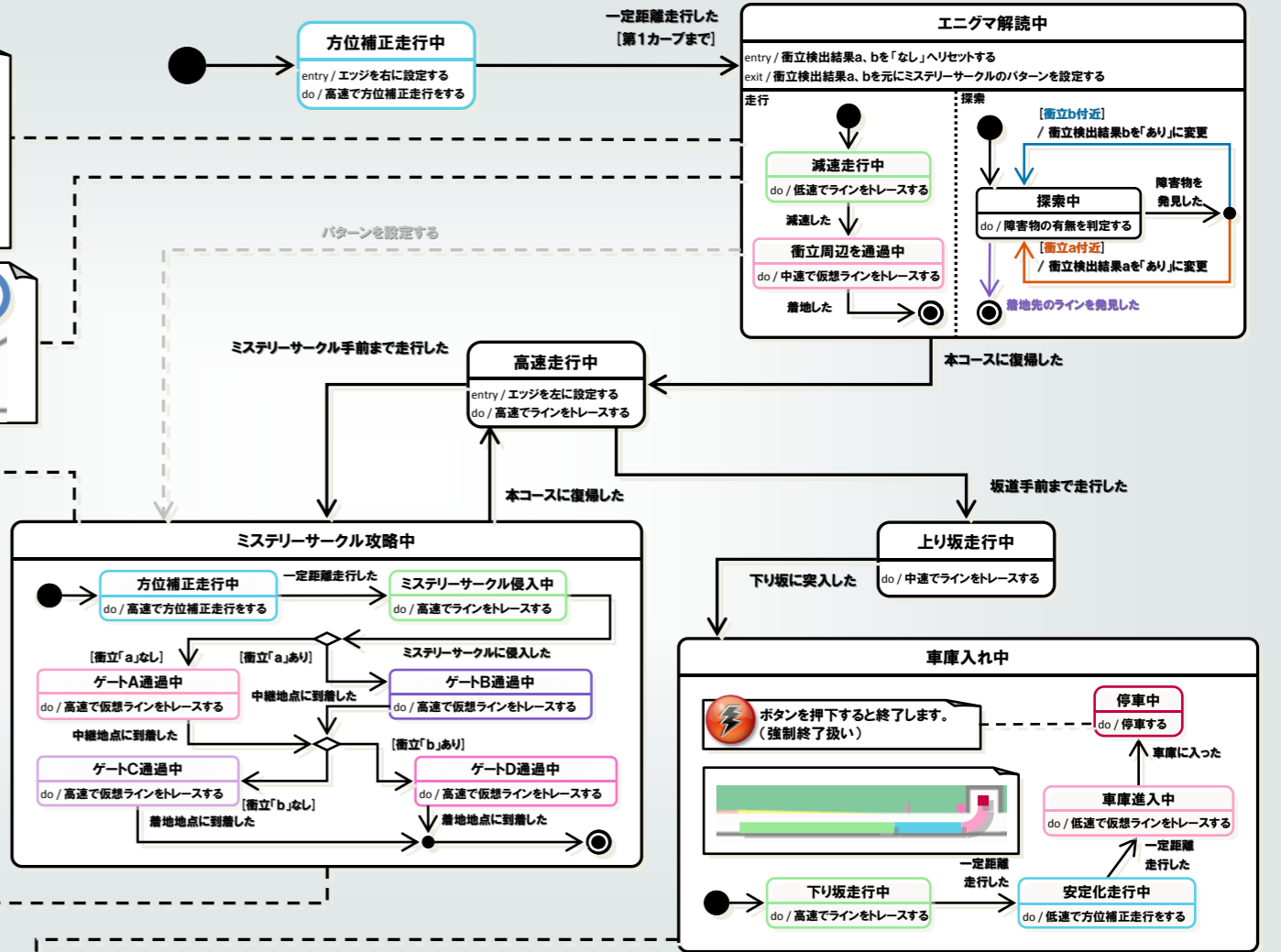
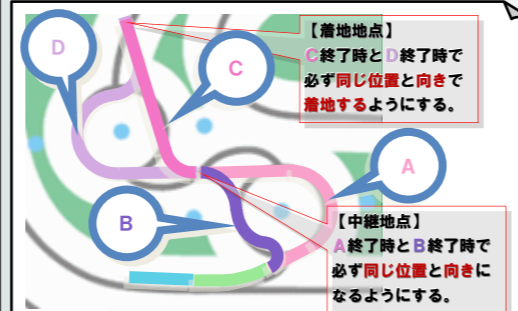
**ミステリーサークル攻略 要点**

走行速度向上ポイント

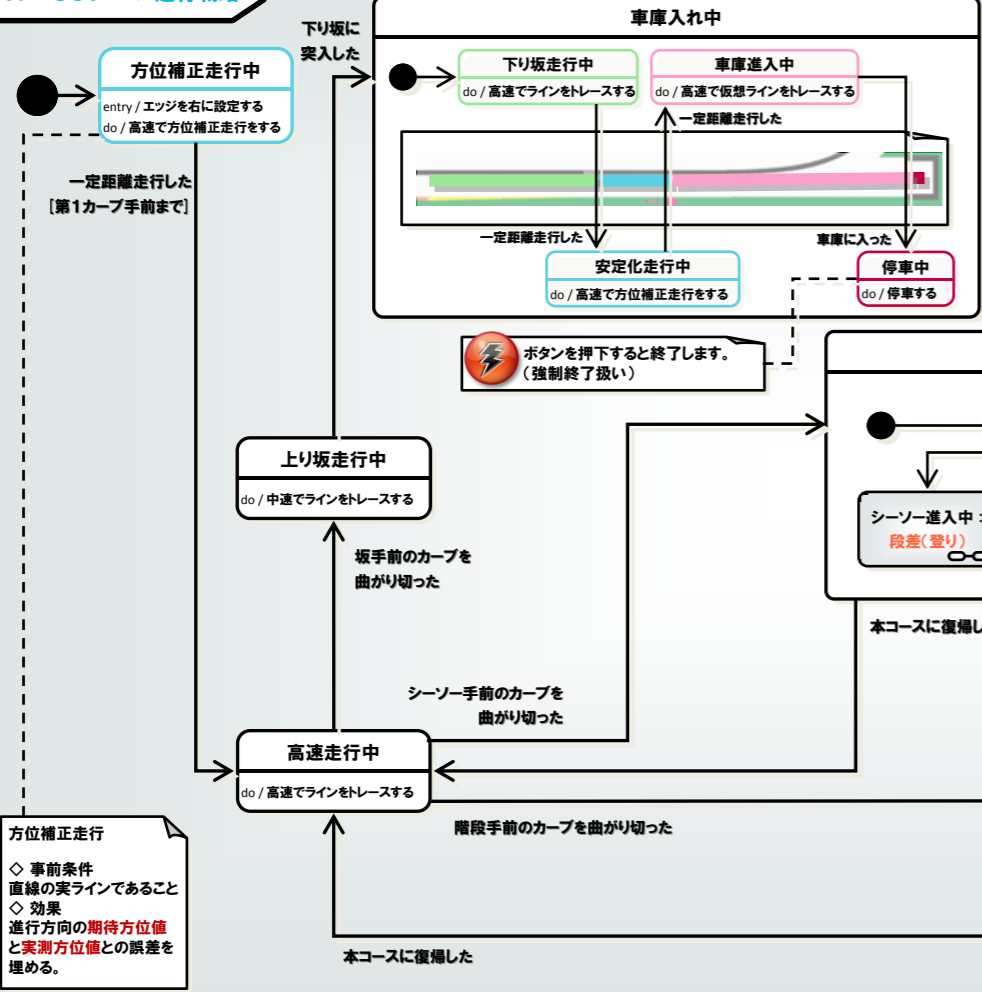
- ★ 仮想ラインをトレースする。
- ★ サークル部分は実ラインをトレースすると、コースアウトを防ぐ為に低速で走行しなければならない。

成功率向上ポイント

- ★ ゲートの中央を通るような仮想ラインを選定。
- ★ 中央を通るようにすることで多少左右にずれても大丈夫。
- ★ 調整スピード向上ポイント
- ★ 各地点(中継、着地)での車体の位置と向きを同じにする。
- ★ 同じにすることで調整項目を減らしている。



## stm OUTコース走行戦術



**車庫入れ 要点**

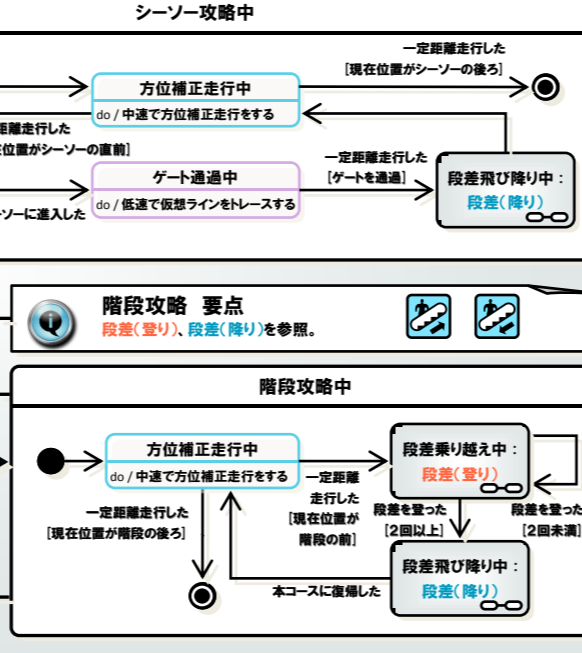
成功率向上ポイント

- ★ 坂の頂上を検知する。
- ★ 正確に検知できる坂の頂上からの距離でガレージの位置を推定する。
- ★ 最後の直線で方位補正走行を行う。
- ★ 最後の直線の向きを基にガレージに向かうことで、制度を向上する。

**シーソー攻略 要点**

成功率向上ポイント

- ★ 実ラインより仮想ライン。
- ★ 急な坂でのライトレースは、傾度値が白寄りになるなどの理由から走行が非常に不安定になるので、方位を基準とする仮想ラインをトレースする。
- ★ タイヤの空転を避ける工夫をしているため、方位計は信頼できる。

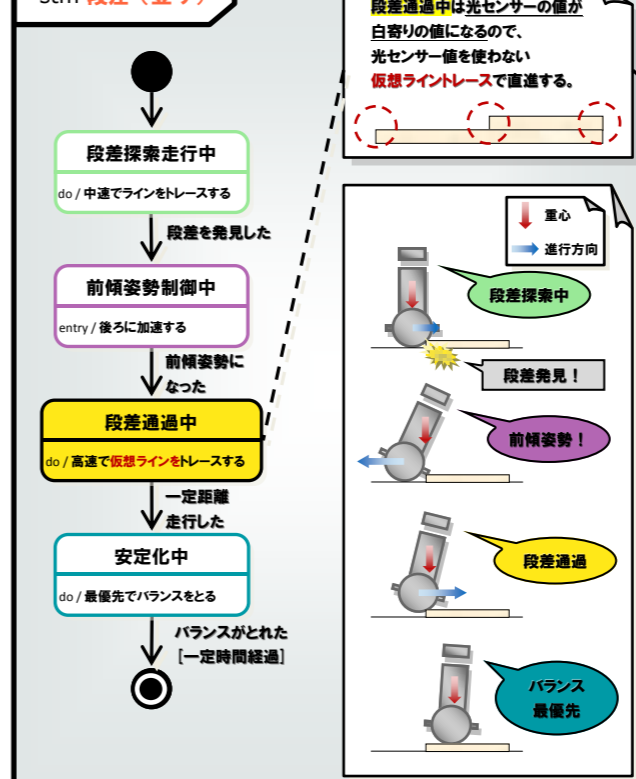


## stm 段差(登り) 要点

★ 段差に対してタイヤが直角にあたるように乗り上げる。

- ★ 片方の車輪が浮いてしまうと、方位計算、走行距離計算に支障が出るため。
- ★ 左右のバランスに対して制御できないため。
- ★ 車体を前傾させて、重心を前へ。
- ★ 倒立状態を保つには車輪とコースの接触点上付近に重心がある必要があるため。
- ★ 乗り上げた後はすぐにスピードを落とし、バランスをとることを優先する。
- ★ 意図的に前傾状態にしているため転倒を防止するため。

## stm 段差(登り)

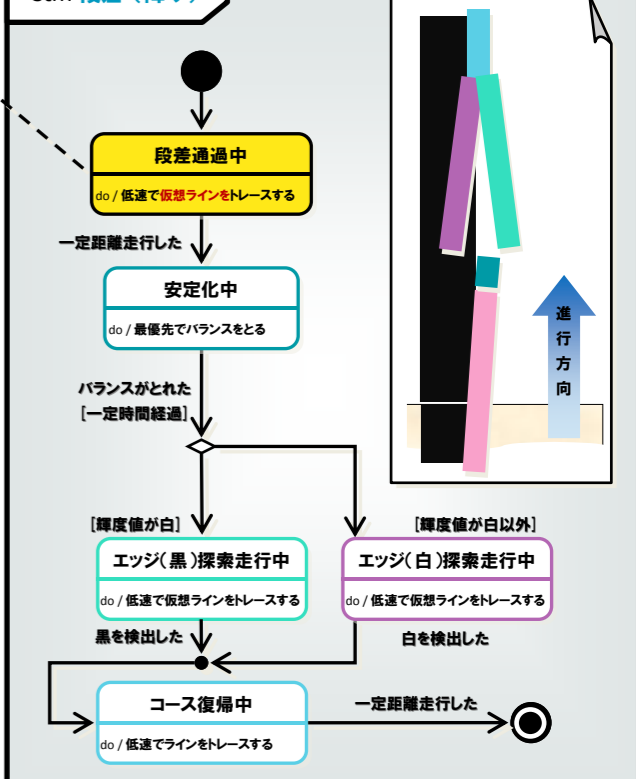


## stm 段差(降り) 要点

★ 段差通過中はやや右前方へと前進する。

- ★ 逆エッジ側(ラインよりも左側)に降りないようにするため。
- ★ 段差通過後、右エッジに確実に乗る。
- ★ 「白」を検出したらラインを突き抜けないように浅い角度で左前方に向かう。
- ★ 「黒」を検出したらコースアウトしないように浅い角度で右前方に向かう。
- ★ エッジを見つけた後は走行を安定させるために低速でラインをトレースする。

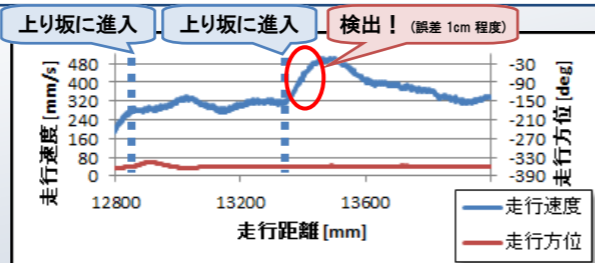
## stm 段差(降り)



## 坂の頂上を検出する

ガレージインを確実に成功させるためには、2つの技術が必要です。

- ① 目的の軌道を正確に走行すること。  
⇒ 仮想ラインを用いることでクリアしました。
- ② ↑ の基準となる位置を正確に見つけること。  
⇒ **坂の頂上を検出することでクリア**しました。  
頂上を越えた直後に発生する“速度”の特徴的な波形を検出します。

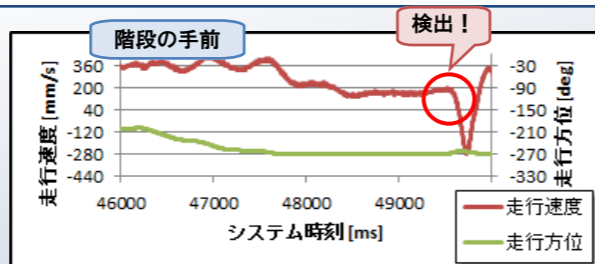


## 段差を検出する

今年度最大の難所は、“段差を登る”ことです。そして、段差を登るための制御を正確に行うためには、段差があることを知らなければなりません。

⇒ **速度を監視することで、段差に触れたことを検出します。**

段差に接触した直後に発生する“速度”の特徴的な波形を検出します。

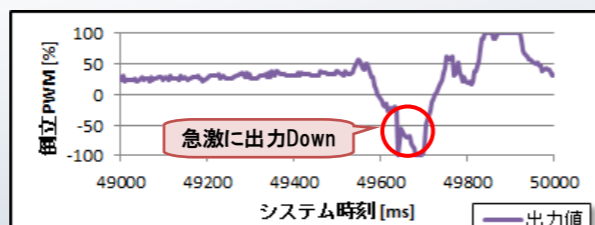


## ジャイロ オフセット操作による急加減速

車体を前傾／後傾にするためには、加速度を与える必要があります。特に、段差を登るために、段差を検出後、即座に車体の傾斜を制御します。

しかし、速度指令値には、バランサー API 内で強力な LPF が掛けられており、指令値を変えてから加減速するまで時間がかかります。

そこで、バランサー API の制御ロジックを逆手にとって、**ジャイロ オフセットを操作することで、即座に加速度を与えます。**

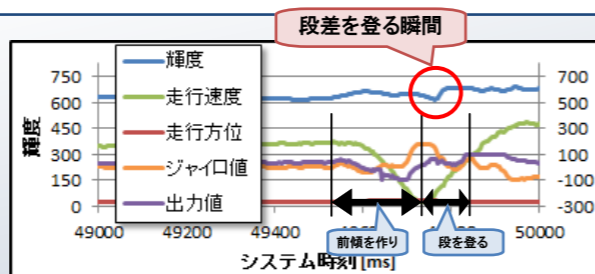


## 段差を登る／降りる瞬間に仮想ラインを使う

光センサーが車輪よりも前方についているため、段差を登るとき／降りるときには、光センサーと地面との距離が変化します。

これが原因で、**段差の登り降りを行う瞬間、光センサーの値が大きく変動してしまいます。**

その対策として、**光に頼らない仮想ラインを利用して走行します。**



## 追加課題: 並行性設計について

私たちは、並行性について 2 種類に大別できると考えました。

- 1) **ハードウェア特性に対応するための並行性**
- 2) **走行戦術を考える上で必要な並行性**

そして、これら並行性を、次のようなコンセプトで設計に反映していきました。

- 1) 「ハードウェア特性に対応するための並行性」は、“仮想デバイス ライブラリ”が隠蔽する  
⇒ See also right in p.2
- 2) 「走行戦術を考える上で必要な並行性」を実現する仕組みを“ステートマシン フレームワーク”が提供する  
⇒ See also left in p.2
- 3) **可能な限りタスクを分割しない**  
⇒ 結果として、アプリケーションではタスクを 1 つしか定義していません

## ◆「可能な限りタスクを分割しない」とは？

NXT はシングルコア CPU が動いているため、タスク分割によるメリットはかなり少ないです。

メリット	デメリット
● 複数の直交する処理が存在し、かつそれらが必要な時間内に完了しない場合、自動的に優先度の高いタスクが実行されるようにスケジュールされる。	● タスク間同期や共有データアクセスに関する問題を考える必要が追加される。 ● タスク切り替えのオーバーヘッドが追加される。

各センサーやモーターの制御を個々に見るとそれらの処理は直交していますが、全体を見ると「センサー値の取得 ⇒ 状態遷移の処理 ⇒ モーターへの出力更新」という流れがあります。つまり、どこかをタスク分割しても、全体の流れを変えないためにタスク間同期が必要になります。それでは、逐次実行と変わりありません（もちろん、マルチコアなら直交なロジックを分割統治することで性能向上のメリットがあります）。

また、メインループの 1 サイクルの処理は、じゅうぶんに時間内に収まっています。

**使うメリットがなく、被るデメリットが非常に多いため、私たちはタスク分割を行いません。**

## ハードウェア特性に対応するための並行性

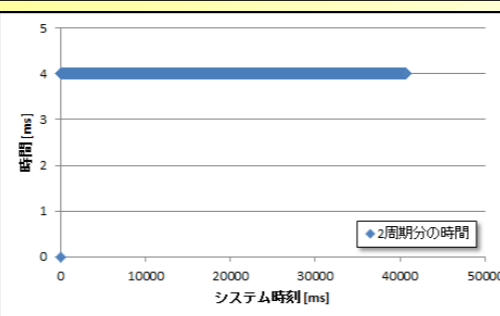
例えば、競技で用いる走行体には超音波センサーと光センサーが接続され、並行して稼働しています。超音波センサーは「超音波を発して、反射した波が返ってくるまでにかかった時間」を測定するため、40ms 程度の周期でセンサー値を問い合わせる必要があります。他方、光センサーの値は 1ms の間に複数回、自動的に更新されます。

このように、センサーによって異なる周期で並行して扱う必要があります。

## 走行戦術を考える上で必要な並行性

例えば、エンジマデコーディング部を走行する際、指定した仮想ライン上を走行すること、ペットボトルを検出することを並行して行います。

このように、難所によっては複数の処理を並行して実行したいことがあります。



↑ 収集したログの各レコード (2 周に 1 回送信) の tick の差を計算してグラフにしました。レコード間の時差がバラけていないので、すべての処理が時間内に収まっています。

## ET ロボコン走行体 監視制御システム

## 目的

デバッグや走行戦術の妥当性検証に必要なもの、それは情報です。／また、走行体へのプログラムアップロードは手間のかかる作業です。

私たちは「“これで良い”と確信に足る、あるいはダメだと確認できるだけの情報を収集する仕組みが必要だ」「数多い制御パラメーターを調整する毎に再コンパイルしたくない」「試走中に迷走した車体をすぐに回収したい」と考え、これらを実現する監視制御システムを構築しました。

## 機能

ETロボコン走行体 監視制御システムは、次に挙げる 7 つのフィーチャを提供します。

- ① 走行中の走行体から、内部状態に関するログデータを受信／記録する
- ② 記録しているログデータをリアルタイムにグラフやアニメーションで表示する
- ③ 記録されているログをさまざまな計算やグラフを用いて解析する

④ ログデータと解析結果を注釈つきで履歴に残す

⑤ 履歴に残された情報を多数の PC で共有する

⑥ 制御パラメーターを走行体に送信／変更する

⑦ 走行体の走行をコントローラーで制御する

dep

開発 PC

監視制御システム

分析用ログデータ (Excel 形式)

Web サーバー

ログデータベース

クラウド風味に  
全てのデータはサーバー側で  
管理します

ワンクリックで、様々な分析結果が  
たくさんのグラフやデータとして  
Excel 上に表示!  
そのままさらに詳しい分析を!

いつでも  
パラメーター変更や走行制御が  
可能!

注釈とともに 皆で共有!  
振り返りの確認にも役立った!

ログ  
とったよ

OK, モデル  
に載せるよ

ログは即座に全員で共有できる!  
モバイルルーターで**試走会**上でも!



## Column: C++言語による開発について

私たちは今回、C++ 言語を用いて開発を行いました。これが地区大会の審査の中で物議を醸したそうなので、少し掘り下げてお話ししたいと思います。

## Q1. なぜ C++ 言語で開発しようと思ったの？

昨年度までは C 言語で実装していましたが、やはりオブジェクト指向で設計したモデルを C 言語で実装するというのは無理があって、いろんなところが歪んでいました。特に昨年度は継承（あるいはジェネリクス）に相当することを実現しようとして、関数ポインタが乱舞し、void\* データが飛び交う素敵なコードになっていました（汗）。これではいけないと、今年は C++ 言語で実装することにしました。

## Q2. C++ 言語で開発してどうだった？

開発はずいぶん楽になりました。先ほど言ったような酷いコードは、即ちバグの温床です。それが、C++ の型システムを有効に使うことによって、コンパイラが殆どすべてのバグを弾いてくれるようになりました。おかげで、開発中にバグで悩むことは一切無かったです。

## Q3. メモリーが足りなくなったりしなかった？

“あっ”という間に足りなくなりました（笑）。1 週間ももたなかったです。ステートマシン フレームワークの実装だけで拡張ファームウェアには乗らなくなってしまって、NXT BIOS を使って開発を続けました。でも、NXT BIOS の上では、まだまだ余裕がありますよ。半分も使ってません。

## Q4. 具体的にはいくつぐらい？

実行イメージが IN/OUT 合わせて 92KB です。このうち、76KB くらいがフレームワークです。RAM は調べていませんが、使用する RAM が可能な限り少なくて済むように、フレームワーク内で工夫しています（2 ページ左側）。

それについて、懇親会で「こんなところで new/delete して大丈夫なのか」と質問されました。その場では「もちろん大丈夫です」と答えた気がしますが、本当は「大丈夫でした」と答えるべきでした。私には「大丈夫じゃないかもしれない」という意識がなかったのです。

## Q5. C++ 言語で開発して一番良かったことは？

良かったことはたくさんあります。1 番は、その強力な型システムを使ったことでしょう。メンバー関数や演算子のオーバーロードのおかげで直感的で分かりやすいフレームワークにできました。デストラクタと情報隠蔽を使ってリソース管理をフレームワークに閉じ込めることもできました。「ポインタが危険なら安全なポインタを作れば（使えば）いい。配列が危険なら安全な配列を作れば（使えば）いい。」という普通の C++er の思想を体現できたと思います。これは C 言語ではできなかったことです。

## Q6. C++ 言語で開発して一番大変だったことは？

今回は実行イメージのサイズが心配で、テンプレート機構を封印していました。C++ 言語だから大変だったという訳じゃないですが、テンプレートが使えなかったのは大変でした。例えば、コンテナなどを自分で作ることにしたわけです。今にして思えば、最初からテンプレートをちゃんと使って、問題が出たら対策を考えればよかったですね。