ブラシレスモータスタータキット(RA2T1)
[ソフトウェア　チュートリアル編]
取扱説明書

ルネサス エレクトロニクス社　RA2T1(QFP-48 ピン)搭載
ブラシレスモータスタータキット

-本書を必ずよく読み、ご理解された上でご利用ください

株式会社北斗電子

REV.1.0.0.0

ー目　次ー

注意事項 1

安全上のご注意 2

CD 内容 4

注意事項 4

1.　チュートリアル 5

1.1.　マイコンボード初期設定 8
1.2.　モータに電流を流す 37
1.3.　A/D 変換と PWM を試す 55
1.4.　モータを回してみる 73
1.5.　ホールセンサの値をみる 83
1.6.　過電流・過熱保護の動作 91
1.7.　相電圧・相電流の観測 100

2.　チュートリアル（応用編） 104

2.1.　ハードウェアでの電流方向切り替え 104
2.2.　相補 PWM 信号での駆動 112
2.3.　相補 PWM 信号での駆動（ホールセンサ使用） 141
2.4.　センサレス駆動 148
2.5.　センサレス＋相補 PWM 駆動 159
2.6.　数値演算に関して 164
取扱説明書改定記録 169
お問合せ窓口 169

注意事項

本書を必ずよく読み、ご理解された上でご利用ください

【ご利用にあたって】

1. 本製品をご利用になる前には必ず取扱説明書をよく読んで下さい。また、本書は必ず保管し、使用上不明な点がある場合は再読し、よく理解して使用して下さい。
2. 本書は株式会社北斗電子製マイコンボードの使用方法について説明するものであり、ユーザシステムは対象ではありません。
3. 本書及び製品は著作権及び工業所有権によって保護されており、全ての権利は弊社に帰属します。本書の無断複写・複製・転載はできません。
4. 弊社のマイコンボードの仕様は全て使用しているマイコンの仕様に準じております。マイコンの仕様に関しましては製造元にお問い合わせ下さい。弊社製品のデザイン・機能・仕様は性能や安全性の向上を目的に、予告無しに変更することがあります。また価格を変更する場合や本書の図は実物と異なる場合もありますので、御了承下さい。
5. 本製品のご使用にあたっては、十分に評価の上ご使用下さい。
6. 未実装の部品に関してはサポート対象外です。お客様の責任においてご使用下さい。

【限定保証】

1. 弊社は本製品が頒布されているご利用条件に従って製造されたもので、本書に記載された動作を保証致します。
2. 本製品の保証期間は購入戴いた日から1年間です。

【保証規定】

保証期間内でも次のような場合は保証対象外となり有料修理となります

1. 火災・地震・第三者による行為その他の事故により本製品に不具合が生じた場合
2. お客様の故意・過失・誤用・異常な条件でのご利用で本製品に不具合が生じた場合
3. 本製品及び付属品のご利用方法に起因した損害が発生した場合
4. お客様によって本製品及び付属品へ改造・修理がなされた場合

【免責事項】

弊社は特定の目的･用途に関する保証や特許権侵害に対する保証等、本保証条件以外のものは明示・黙示に拘わらず一切の保証は致し兼ねます。また、直接的・間接的損害金もしくは欠陥製品や製品の使用方法に起因する損失金・費用には一切責任を負いません。損害の発生についてあらかじめ知らされていた場合でも保証は致し兼ねます。
ただし、明示的に保証責任または担保責任を負う場合でも、その理由のいかんを問わず、累積的な損害賠償責任は、弊社が受領した対価を上限とします。本製品は「現状」で販売されているものであり、使用に際してはお客様がその結果に一切の責任を負うものとします。弊社は使用または使用不能から生ずる損害に関して一切責任を負いません。
保証は最初の購入者であるお客様ご本人にのみ適用され、お客様が転売された第三者には適用されません。よって転売による第三者またはその為になすお客様からのいかなる請求についても責任を負いません。
本製品を使った二次製品の保証は致し兼ねます。

安全上のご注意

製品を安全にお使いいただくための項目を次のように記載しています。絵表示の意味をよく理解した上でお読み下さい。

表記の意味

警告	取扱を誤った場合、人が死亡または重傷を負う危険が切迫して生じる可能性がある事が想定される
注意	取扱を誤った場合、人が軽傷を負う可能性又は、物的損害のみを引き起こすが可能性がある事が想定される

絵記号の意味

	一般指示 使用者に対して指示に基づく行為を強制するものを示します		**一般禁止** 一般的な禁止事項を示します
	電源プラグを抜く 使用者に対して電源プラグをコンセントから抜くように指示します		**一般注意** 一般的な注意を示しています

警告

以下の警告に反する操作をされた場合、本製品及びユーザシステムの破壊・発煙・発火の危険があります。マイコン内蔵プログラムを破壊する場合もあります。

1. 本製品及びユーザシステムに電源が入ったままケーブルの抜き差しを行わないでください。
2. 本製品及びユーザシステムに電源が入ったままで、ユーザシステム上に実装されたマイコンまたはIC等の抜き差しを行わないでください。
3. 本製品及びユーザシステムは規定の電圧範囲でご利用ください。
4. 本製品及びユーザシステムは、コネクタのピン番号及びユーザシステム上のマイコンとの接続を確認の上正しく扱ってください。

発煙・異音・異臭にお気付きの際はすぐに使用を中止してください。

電源がある場合は電源を切って、コンセントから電源プラグを抜いてください。そのままご使用すると火災や感電の原因になります。

⚠注意

以下のことをされると故障の原因となる場合があります。

1. 静電気が流れ、部品が破壊される恐れがありますので、ボード製品のコネクタ部分や部品面には直接手を触れないでください。
2. 次の様な場所での使用、保管をしないでください。

   ホコリが多い場所、長時間直射日光があたる場所、不安定な場所、衝撃や振動が加わる場所、落下の可能性がある場所、水分や湿気の多い場所、磁気を発するものの近く
3. 落としたり、衝撃を与えたり、重いものを乗せないでください。
4. 製品の上に水などの液体や、クリップなどの金属を置かないでください。
5. 製品の傍で飲食や喫煙をしないでください。

ボード製品では、裏面にハンダ付けの跡があり、尖っている場合があります。

取り付け、取り外しの際は製品の両端を持ってください。裏面のハンダ付け跡で、誤って手など怪我をする場合があります。

CD メディア、フロッピーディスク付属の製品では、故障に備えてバックアップ(複製)をお取りください。

製品をご使用中にデータなどが消失した場合、データなどの保証は一切致しかねます。

アクセスランプがある製品では、アクセスランプ点灯中に電源の切断を行わないでください。

製品の故障や、データの消失の原因となります。

本製品は、医療、航空宇宙、原子力、輸送などの人命に関わる機器やシステム及び高度な信頼性を必要とする設備や機器などに用いられる事を目的として、設計及び製造されておりません。

医療、航空宇宙、原子力、輸送などの設備や機器、システムなどに本製品を使用され、本製品の故障により、人身や火災事故、社会的な損害などが生じても、弊社では責任を負いかねます。お客様ご自身にて対策を期されるようご注意ください。

CD 内容

添付「ソフトウェア CD」には、以下の内容が収録されています。

・チュートリアル
　TUTORIAL フォルダ以下　[本マニュアルで説明する内容]
・サンプルプログラム
　SAMPLE フォルダ以下
・プログラムのバイナリ（MOT ファイル）
　BIN フォルダ以下
・マニュアル
　manual フォルダ以下

注意事項

本マニュアルに記載されている情報は、ブラシレスモータスタータキットの動作例・応用例を説明したものです。

お客様の装置・システムの設計を行う際に、本マニュアルに記載されている情報を使用する場合は、お客様の責任において行ってください。本マニュアルに記載されている情報に起因して、お客様または第三者に損害が生じた場合でも、当社は一切その責任を負いません。

本マニュアルに、記載されている事項に誤りがあり、その誤りに起因してお客様に損害が生じた場合でも、当社は一切その責任を負いません。

本マニュアルの情報を使用した事に起因して発生した、第三者の著作権、特許権、知的財産権侵害に関して、当社は一切その責任を負いません。当社は、本マニュアルに基づき、当社または第三者の著作権、特許権、知的財産権の使用を許諾する事はありません。

1. チュートリアル

本チュートリアルでは、マイコンの基本的なプログラムを説明致しますが、ルネサスエレクトロニクスの Web から「RA2T1 グループ　ユーザーズマニュアル　ハードウェア編」を予めダウンロードして、手元に用意してください。マイコンの詳細な設定は、このマニュアル（以下「ハードウェアマニュアル」と記載する）に記載されています。

また、本マニュアルで説明するプログラムは、ルネサスエレクトロニクス e2studio+FSP の環境向けに作成されていますので、e2studio+FSP6.0.0 以降を PC にインストールしておいてください。なお、開発環境のインストール等は、ルネサスエレクトロニクス社のマニュアルを参照してください。（RA 向けに、e2studio と FSP がセットになっているインストーラを使用して e2studio をインストールする。もしくは、e2studio をインストール後に、RA 向けの FSP を追加でインストールする等して、FSP が使用できる状態としてください。）

本キットは、マイコンの基本的なプログラムは行えるが、モータ制御は初めてという方を想定した構成となっており、ホールセンサを使用してブラシレスモータを回転させる方法。モータが動作しているときの、電圧や電流を観測する方法。また、モータドライバボードの保護回路の使用方法の習得を目的にしています。

以下が、本チュートリアルで学べる事柄となります。

・マイコンボードを動作させるための初期設定（クロック設定やモジュールスタンバイの解除）
・モータのコイルに流す電流を制御する方法
・A/D 変換
・PWM（パルス幅変調）
・ホールセンサの値を取得する方法
・温度値の取得
・過電流検出の方法
・モータが動作しているときの相電圧・相電流の取得
・マイコンのハードウェアを使用してモータを回す方法
・ベクトル型制御（相補 PWM 駆動）
・疑似ホールセンサパターンの生成（ホールセンサレス制御）

－チュートリアル一覧－

チュートリアル	プロジェクト名	内容
チュートリアル 1	RA2T1_BLMKIT_TUTORIAL1	マイコンボードを動作させる初期設定 スイッチの読み取りと LED の制御
チュートリアル 2	RA2T1_BLMKIT_TUTORIAL2	モータへの通電方法 （モータは回転しません）
チュートリアル 3	RA2T1_BLMKIT_TUTORIAL3	A/D 変換と PWM 波形の生成方法
チュートリアル 4	RA2T1_BLMKIT_TUTORIAL4	実際にモータを回転させる方法
チュートリアル 5	RA2T1_BLMKIT_TUTORIAL5	ホールセンサの値を利用する方法
チュートリアル 6	RA2T1_BLMKIT_TUTORIAL6	過電流・過熱保護
チュートリアル 7	RA2T1_BLMKIT_TUTORIAL7	相電圧・相電流の観測

チュートリアル 1~7 までは、つながりのある内容となっており、例えばチュートリアル 6 はチュートリアル 5 の内容に「過電流・過熱保護」の機構を追加する内容になっています。一つ前のチュートリアルに少しずつ機能追加を行っていき、モータ制御プログラムを組み立てていく内容です。

チュートリアル	プロジェクト名	内容
チュートリアル A	RA2T1_BLMKIT_TUTORIAL_A	マイコンのハードウェア制御機構を用いたモータ制御

チュートリアル A は、チュートリアル 7 をベースに、マイコンが持っている機能であるブラシレスモータ用出力相切り替え機能を使ってモータを回す内容です。

チュートリアル	プロジェクト名	内容
チュートリアル B	RA2T1_BLMKIT_TUTORIAL_B	相補 PWM を使ったモータ制御（回転数固定）
チュートリアル B2	RA2T1_BLMKIT_TUTORIAL_B2	相補 PWM を使ったモータ制御（回転数可変）

チュートリアル B は、チュートリアル 7 をベースに、モータ制御の部分をベクトル制御とも呼ばれる相補 PWM 駆動に変えたものです。

チュートリアル	プロジェクト名	内容
チュートリアル C	RA2T1_BLMKIT_TUTORIAL_C	疑似ホールセンサパターンを使用した制御

チュートリアル C は、チュートリアル 7 をベースに、ホールセンサの部分を、疑似ホールセンサパターン（UVW 相の電圧からホールセンサパターンを生成、ホールセンサレス制御）を選択できるよう変えたものです。

チュートリアル	プロジェクト名	内容
チュートリアル BC	RA2T1_BLMKIT_TUTORIAL_BC	疑似ホールセンサパターンと相補 PWM を組み合わせた制御

チュートリアル BC は、チュートリアル B とチュートリアル C を組み合わせたもので、相補 PWM を使用して、疑似ホールセンサパターン（ホールセンサレス制御）を選択できるようにしたものです。

チュートリアル 1~7 は連続したチュートリアル。モータを回す上で基本的な内容を 1 ステップずつ追加していく内容。A と B と C はチュートリアル 7 の内容から枝分かれするイメージです。

サンプルプログラム(RA2T1_BLMKIT_SAMPLE)は、チュートリアル BC をベースに、相補 PWM 駆動とモータ内蔵ホールセンサ（もしくは、疑似ホールセンサパターン）を使った制御になっています。チュートリアルで学んだ内容をまとめたのがサンプルプログラムです。（サンプルプログラムは、別なマニュアルで内容を説明しています。）

1.1. マイコンボード初期設定

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL1

本キットに付属のマイコンボード(HSBRA2T1F48)は、RA2T1 グループのマイコンを搭載しており、クロック周波数 48MHz で動作させることが出来ます。(外付けの水晶振動子は、20MHz ですが、本キットでは使用していません。マイコンを最大動作周波数の 64MHz で動かす場合は、クロックソースを HOCO(マイコン内蔵オシレータ)に設定します。)

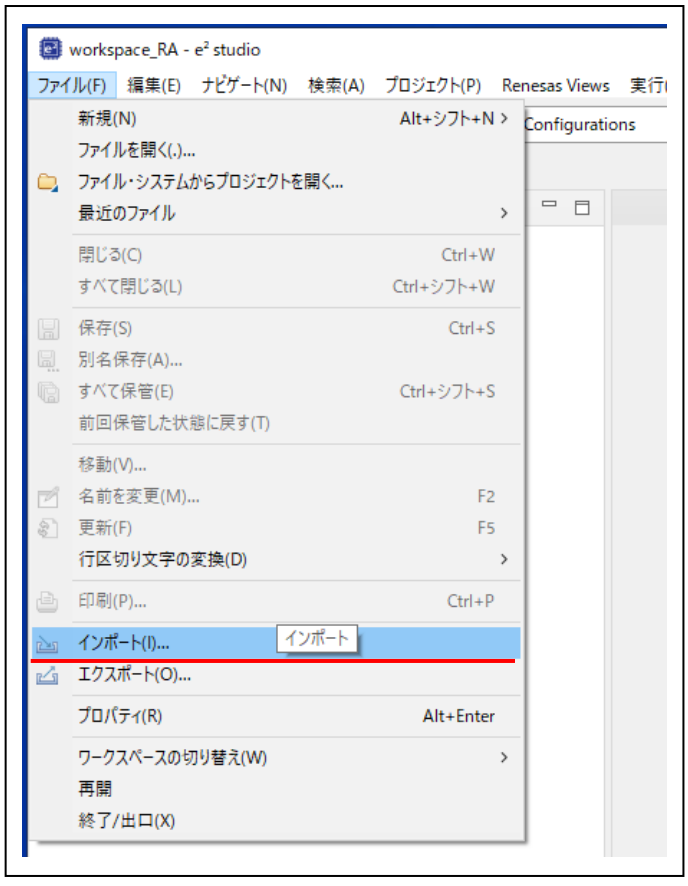
マイコンの動作モードやクロック周波数は、プログラムで設定する必要があります。

本プロジェクトでは、

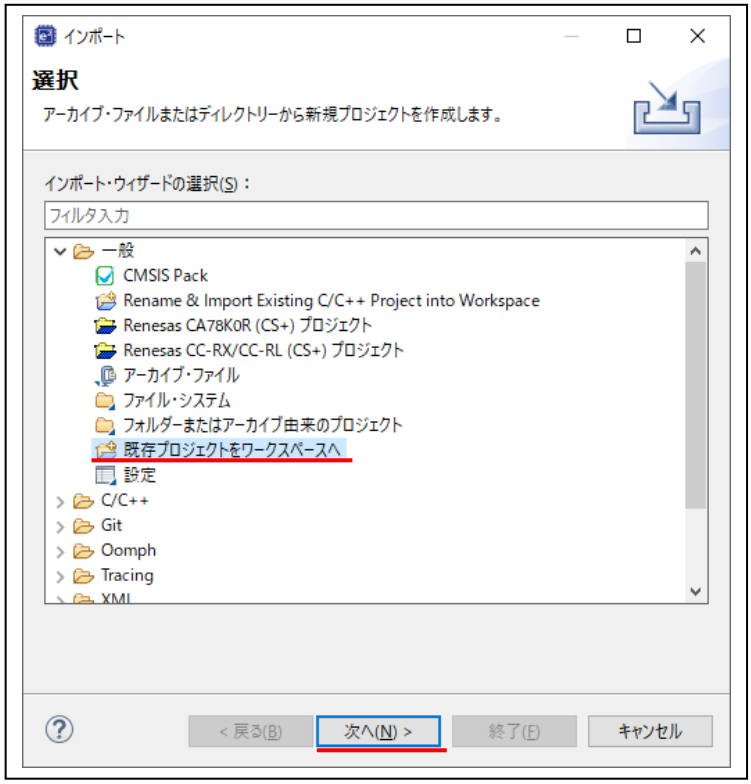
・マイコンボードの初期設定
・基本的な動作

を確認します。(※本プロジェクトは、モータドライバボード・接続ボードをつないだ状態で実行してください)

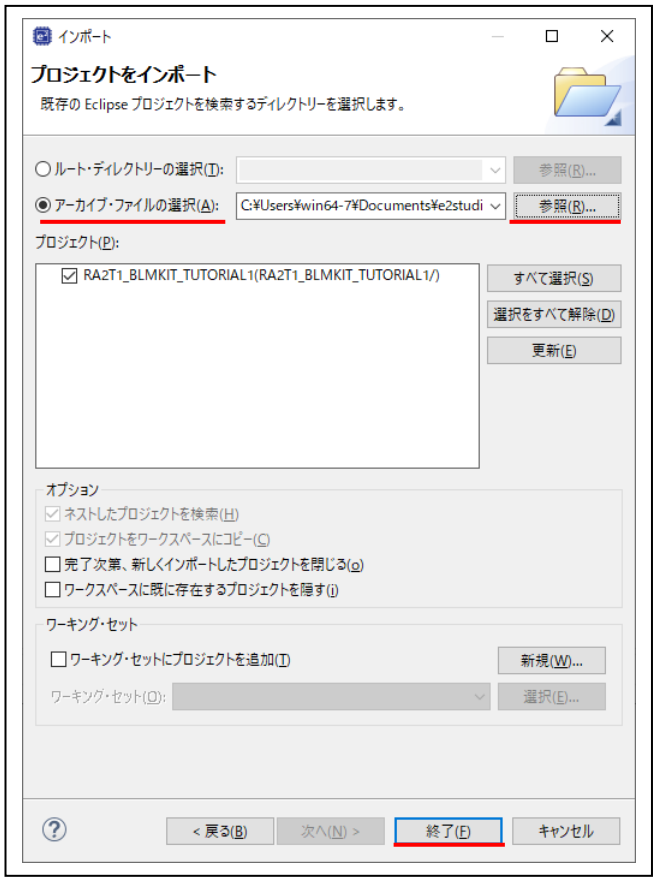
CD 内の TUTORIAL フォルダ内のアーカイブ(e2studio のプロジェクトをアーカイブした)ファイル、RA2T1_BLMKIT_TUTORIAL1.zip を、e2studio のプロジェクトにインポートしてください。



e2studio の「ファイル」メニューで「インポート」を選択します。



「一般」グループ内の「既存プロジェクトをワークスペースへ」を選択。「次へ」。

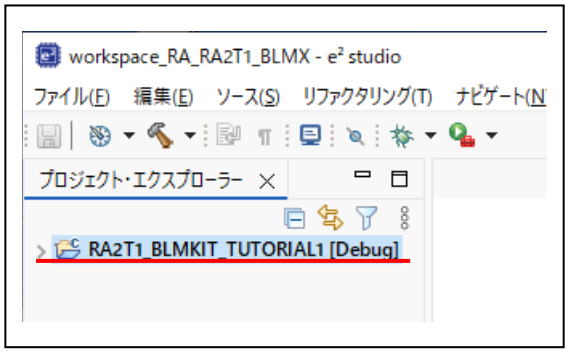


「アーカイブ・ファイルの選択」を選ぶ。

参照ボタンを押し、(RA2T1_BLMKIT_TUTORIAL1.zip)を選択。

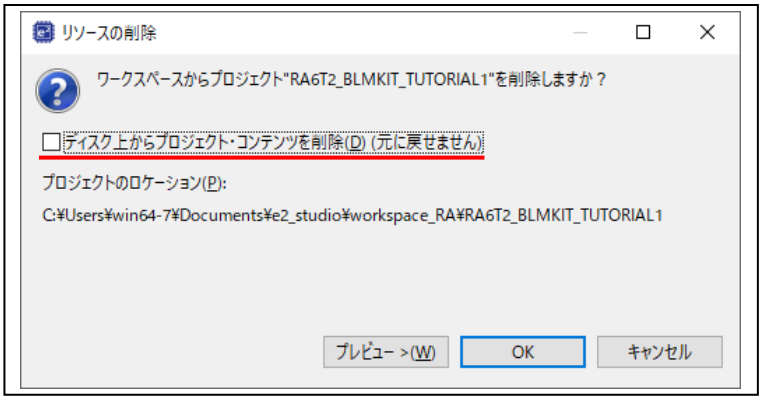
プロジェクト
RA2T1_BLMKIT_TUTORIAL1 がインポート対象に選択されている事を確認し、「終了」を押す。

プロジェクト・エクスプローラ上で、プロジェクト「RA2T1_BLMKIT_TUTORIAL1」が見えていればプロジェクトのインポートは成功です。



※同一名称のプロジェクトが存在している場合インポートが出来ませんので、ワークスペースを別に作成するか、既存のプロジェクトを削除後にインポートしてください

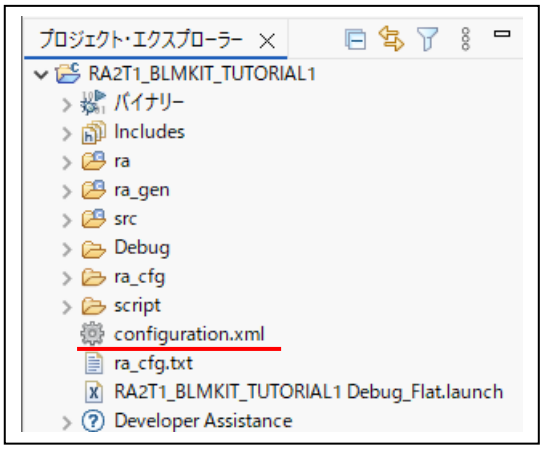
[参考]
※プロジェクトを削除する際に、「ディスク上からプロジェクト・コンテンツを削除」のチェックを入れないで削除した場合は、プロジェクトの実体（フォルダ、ファイル類）は残ったままとなりますので、一からプロジェクトのインポートをやり直したい場合は、「ディスク上からプロジェクト・コンテンツを削除」にチェックを入れてプロジェクトを削除してください



プロジェクトのインポート後、プロジェクトの基本的な設定を行う項目を次ページ以降に示します。
※RA2T1_BLMKIT_TUTORIAL1 プロジェクトをインポートした場合、次ページ以降の設定は済んだ状態となっています。プロジェクトを新規作成した場合、どのような設定を行う必要があるかの参考のために記載しています。
（RA2T1_BLMKIT_TUTORIAL1 プロジェクトを作成した手順を示しています）

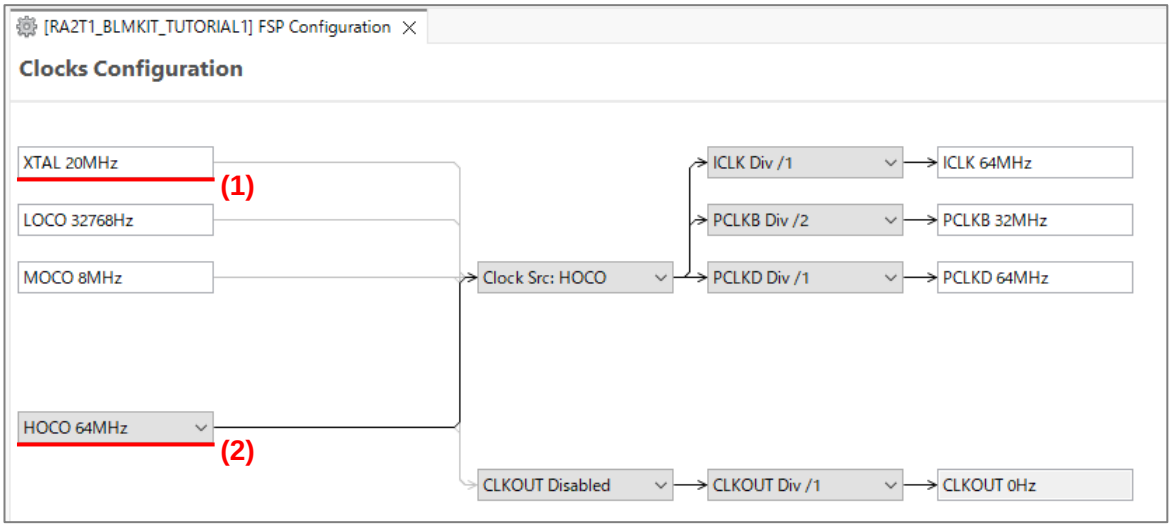
マイコンで動作するプログラムを作成する場合、クロック等の初期設定が必要です。ルネサスエレクトロニクス社がリリースしている FSP を使用すると、GUI 画面でクロックの設定を行い、コード生成ボタンを押す事で、初期設定のプログラムコードが出力されます。

プロジェクト・エクスプローラ内の、RA2T1_BLMKIT_TUTORIAL1 のフォルダアイコンをダブルクリックして、プロジェクトを開いた状態として、



configuration.xml（歯車のアイコンのファイル）をダブルクリック

・Clocks タブ



赤線部がデフォルトから変更している点になります。

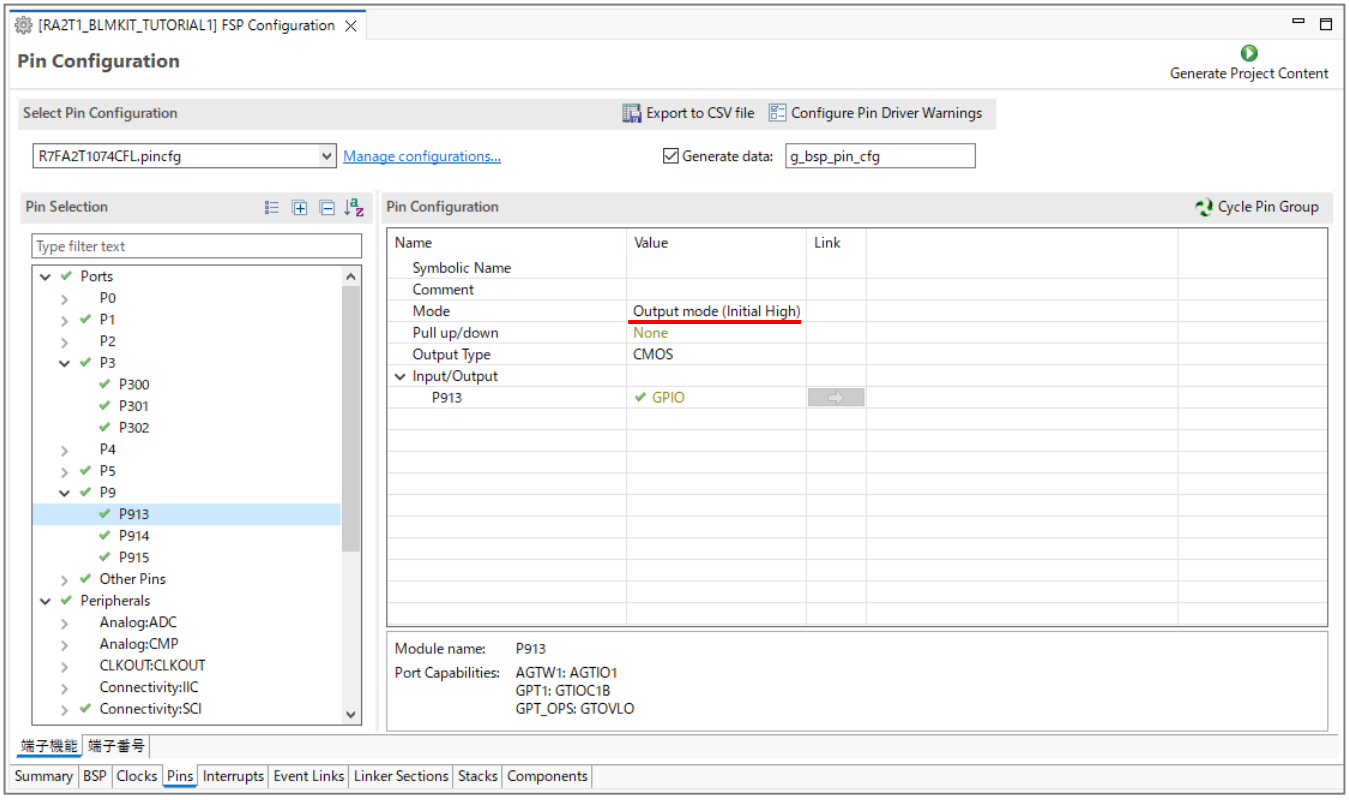
(1)XTAL **20000000** (20MHz)を入力
→ボード搭載 XTAL が 20MHz なので、その値に合わせる
(2)**HOCO 64MHz** を選択
→本マイコンは、最大動作周波数が 64MHz なので、64MHz 動作を選択

・主要なクロック設定値

	[MHz]	クロック種
ICLK	64	CPU コアクロック
PCLKB	32	周辺クロック
PCLKD	64	A/D 変換クロック
HOCO	64	内蔵オシレータ

設定後、Generate Project Content のボタンを押すと、設定した項目に従い、クロック設定のコードが自動生成されます。(他にも設定項目がありますので、ここでは Generate Project Content のボタンを押さずに、次へ進んで良いです。)

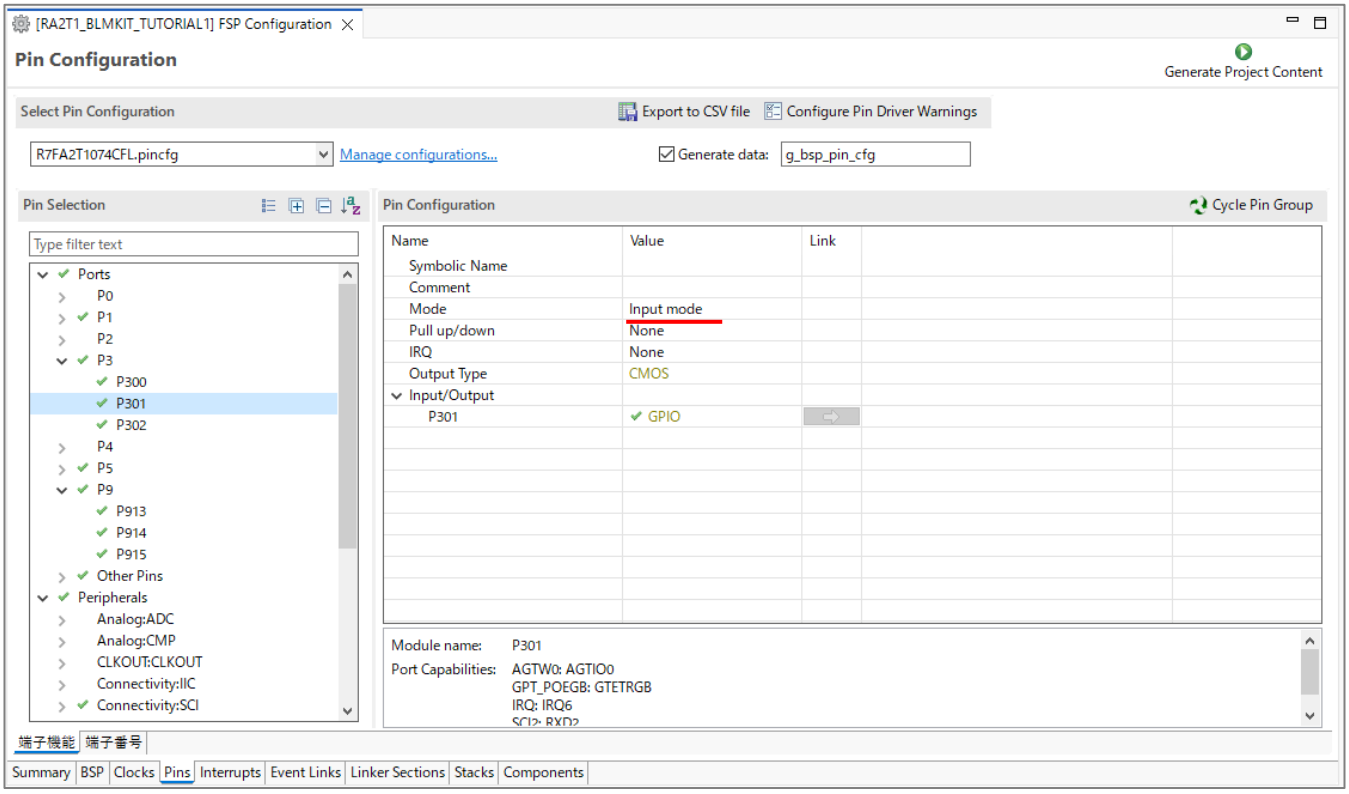
・Pins タブ(端子設定)



P913 を、Output mode(Initial High)に設定します。P913 は、接続ボード上の LED に接続されている端子なので、出力モード、初期値 H とします(LED 消灯状態)。

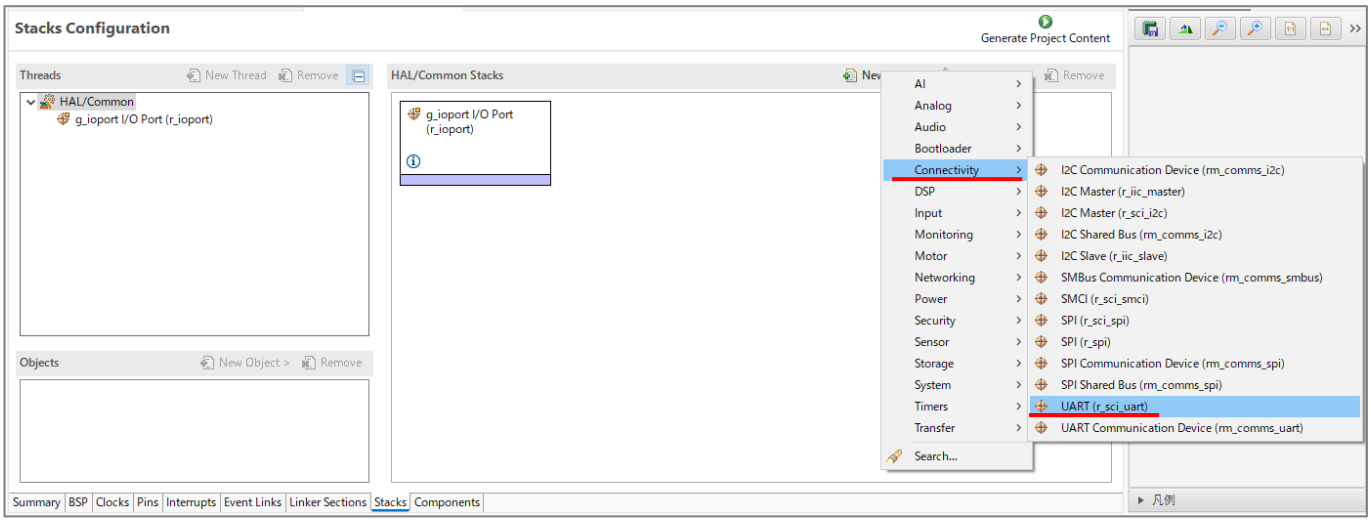
同様に、P914, P915 も、同じ設定とします。

(マイコンボード上の LED を使用する場合は、P500 を、Output mode(Initial Low)に設定します。)



P301 を Input mode に設定します。P301 は、接続ボード上のトグルスイッチに接続されている端子です。
同様に、P302 も同じ設定とします。

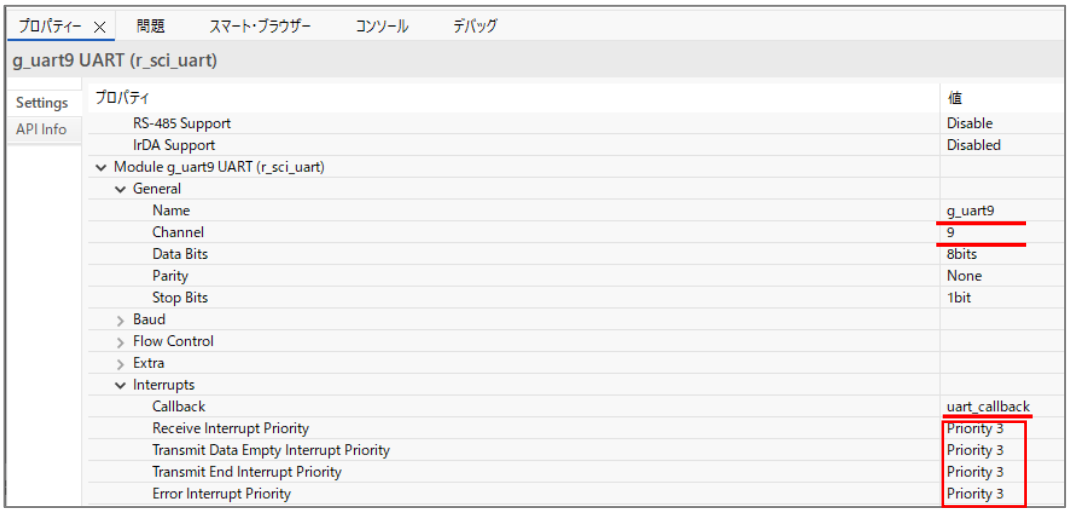
・Stacks タブ



Stacks タブから、

New Stack – Connectivity – UART(r_sci_uart)

を選択。UART は、PC との通信に使用する機能です。
UART(r_sci_uart)のプロパティとして、以下の設定が必要。



General – Name **g_uart9**

→名称は任意ですが、SCI9 を使用しているので、g_uart9 と設定しています

（※g_uart9 以外の名称とした場合は、ソースコードの変更が必要です）

General – Channel **9**

→SCI9 を使用していますので、9 を入力する必要があります

Interrupts – Callback **uart_callback**

→SCI9 の割り込みから呼ばれる関数名の指定

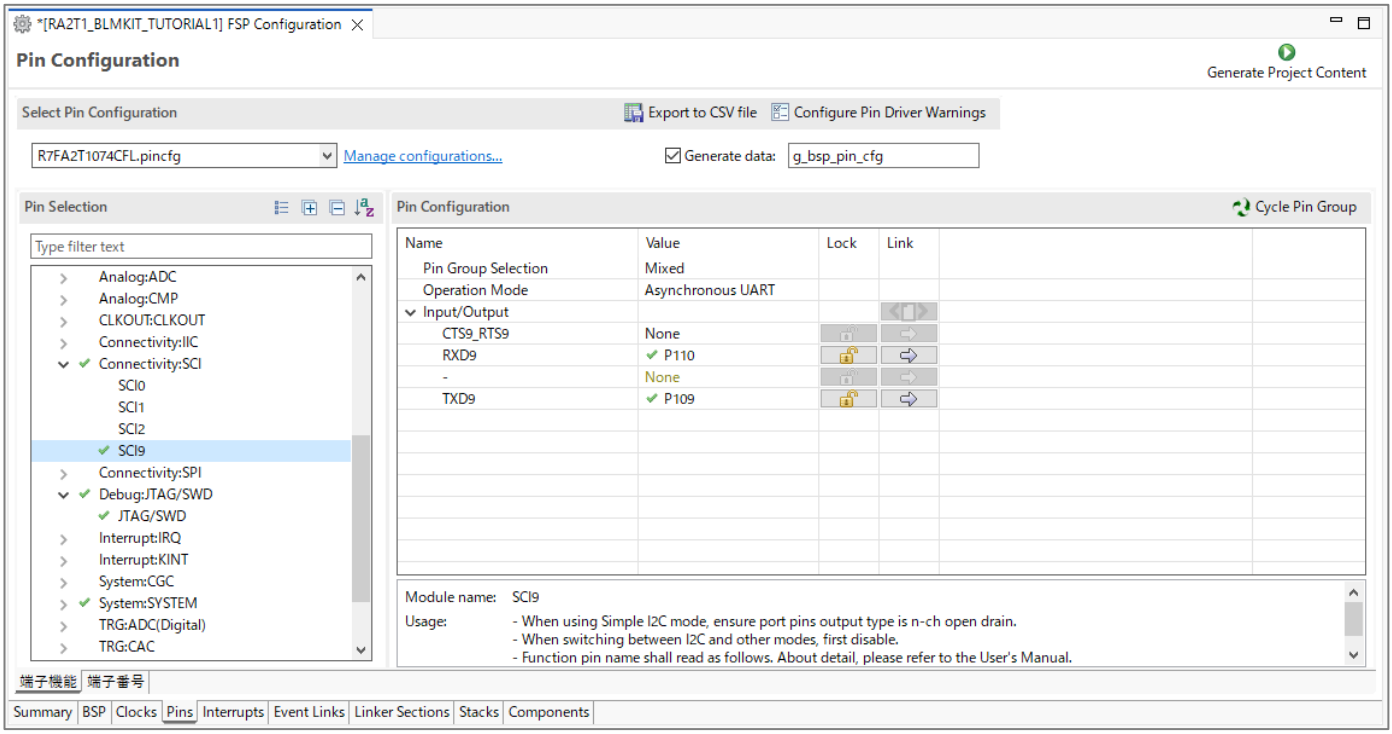
（※任意の関数名を指定可能ですが、uart_callback 以外の関数名を入力した場合は、ソースコードの変更が必要です）

Interrupts – xxx Priority **Priority 3**

→割り込みの優先度設定

（※優先度は任意ですが、モータ制御の場合モータ制御で使用する割り込みに高い優先度を割り当て、通信には低い優先度を設定する事が推奨です。Priority 0 ~ Priority 3 を設定可能で、3 が最低優先度です。）

・Pins タブ（UART, SCI9 の端子設定）

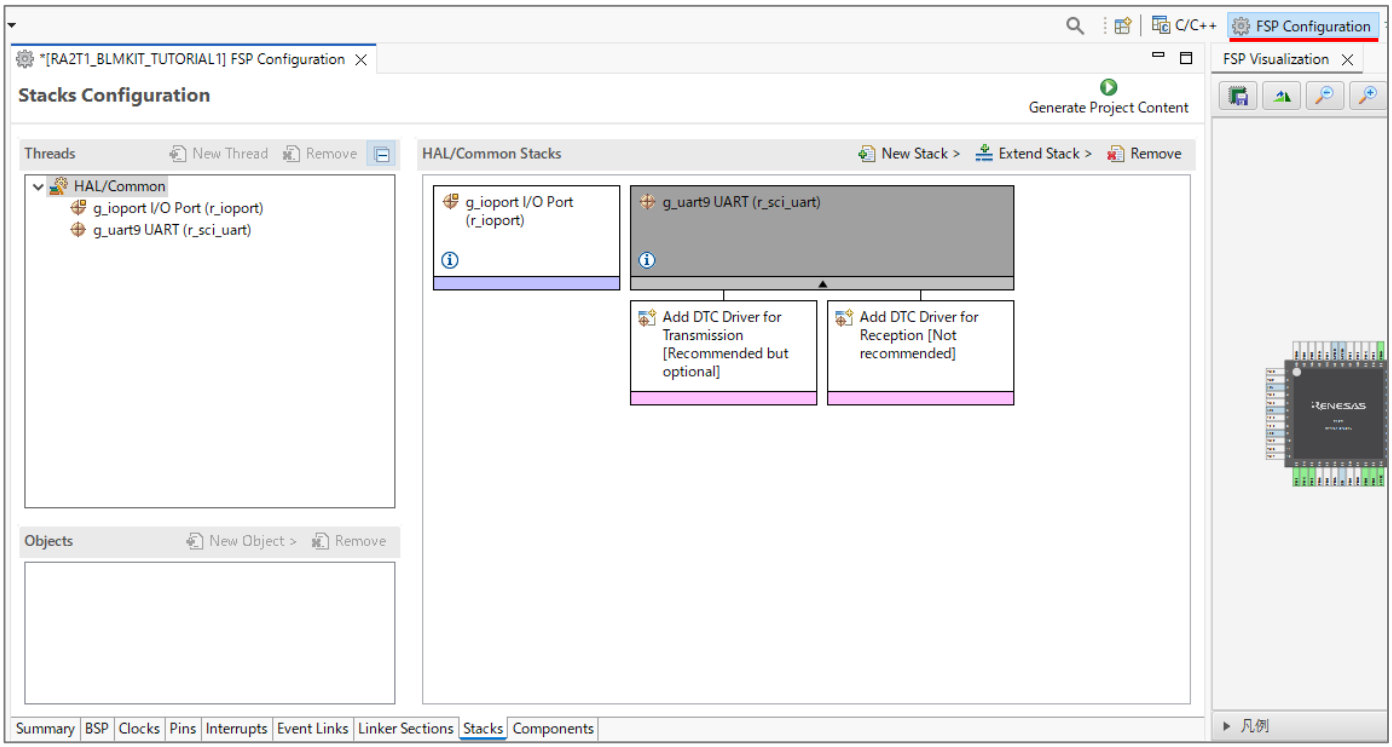


SCI9 で使用する端子を選びます。

RXD9 **P109**

TXD9 **P110**

とします。



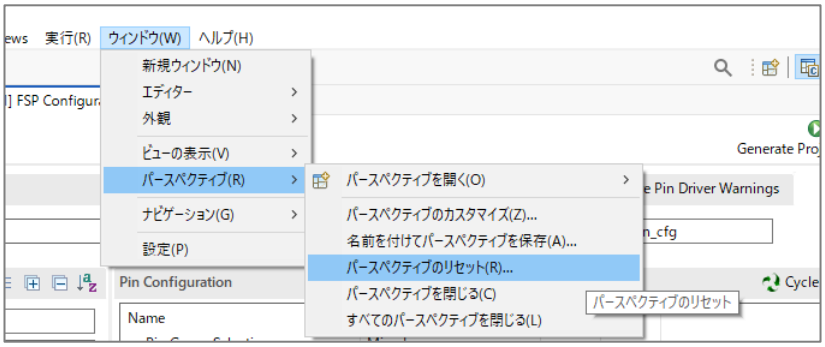
Stack に機能を追加した場合は、Stacks タブに追加した機能の箱が表示されます。

－FSP の設定変更を行う際のパースペクティブの設定－

FSP 上の設定や、FPS で追加した stack のプロパティの表示や編集を行う際には、上部のパースペクティブの設定で、「FSP Configuration」を選んでください。（上記画面の赤線部のボタンを押して選択した状態とする）

（ソースコードの編集時は「C/C++」、デバッグ時は「デバッグ」を選択してください）

－パースペクティブのリセット－



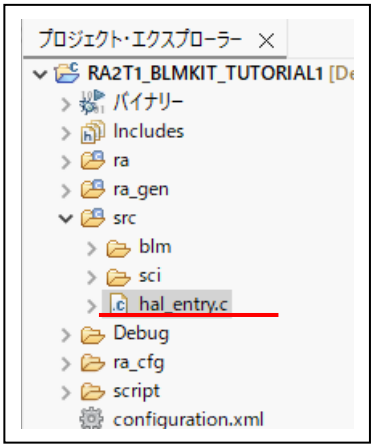
ウインドウの配列がごちゃごちゃになって、どこを開いているかよく判らなくなった場合は、
ウィンドウ － パースペクティブ － パースペクティブのリセット
で、ウィンドウの配置を初期化してください。

FSP での設定が一通り終わった後に

Generate Project Content のボタンを押します。そのタイミングで、GUI 画面で設定した項目が、ソースコードに反映されます。

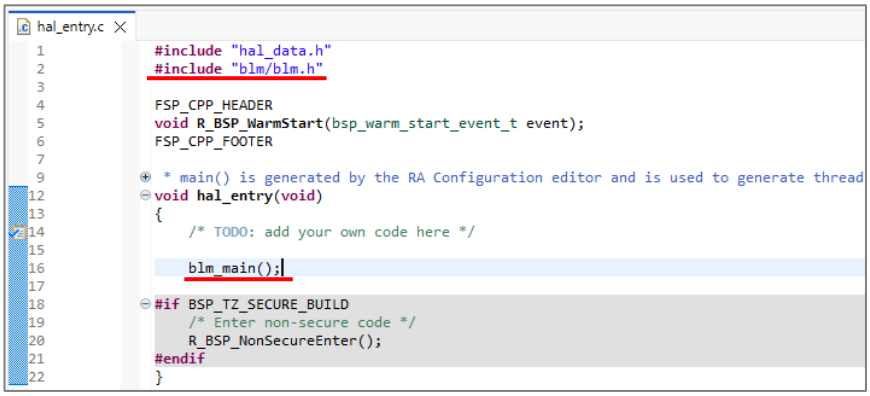
（RA2T1_BLMKIT_TUTORIAL1 のプロジェクトでは、上記で説明している設定は、設定済みとなっています。新規にプロジェクトを作成した場合は、上記の項目を設定してください。）

次に、ユーザプログラム本体を書き下します。



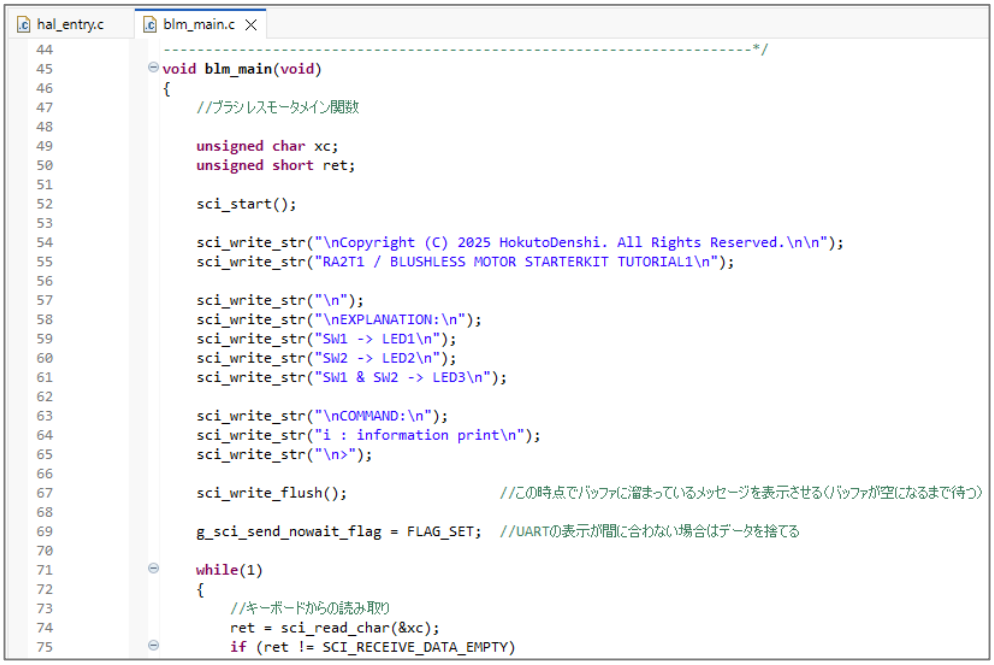
src 以下が、ユーザ側で作成したソースコードを格納するフォルダとなります。

src の下の hal_entry.c がユーザプログラムのエントリーポイント（プログラムのスタート地点）を記載するファイルとなります。



hal_entry.c には、赤線の 2 行を追加します。
#include "blm/blm.h"　blm_main()のプロトタイプを含むヘッダ
blm_main()　ブラシレスモータ制御のメイン関数

・blm_main.c



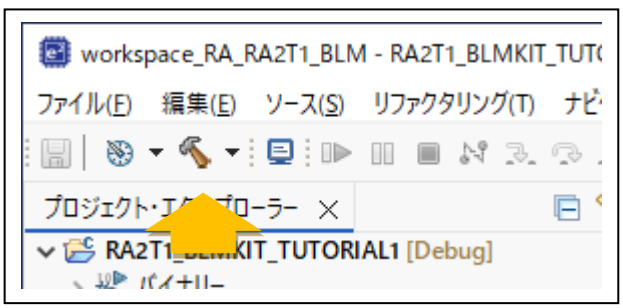
blm/blm_main.c
がプログラムの本体となります。

このチュートリアルでは、モータ制御は行っておらず、

・マイコンボードの初期設定
クロックや汎用 I/O 等の設定方法
・単純な SW と LED の操作
・UART 通信

を行うチュートリアルとなっています。

プログラムを記載後、



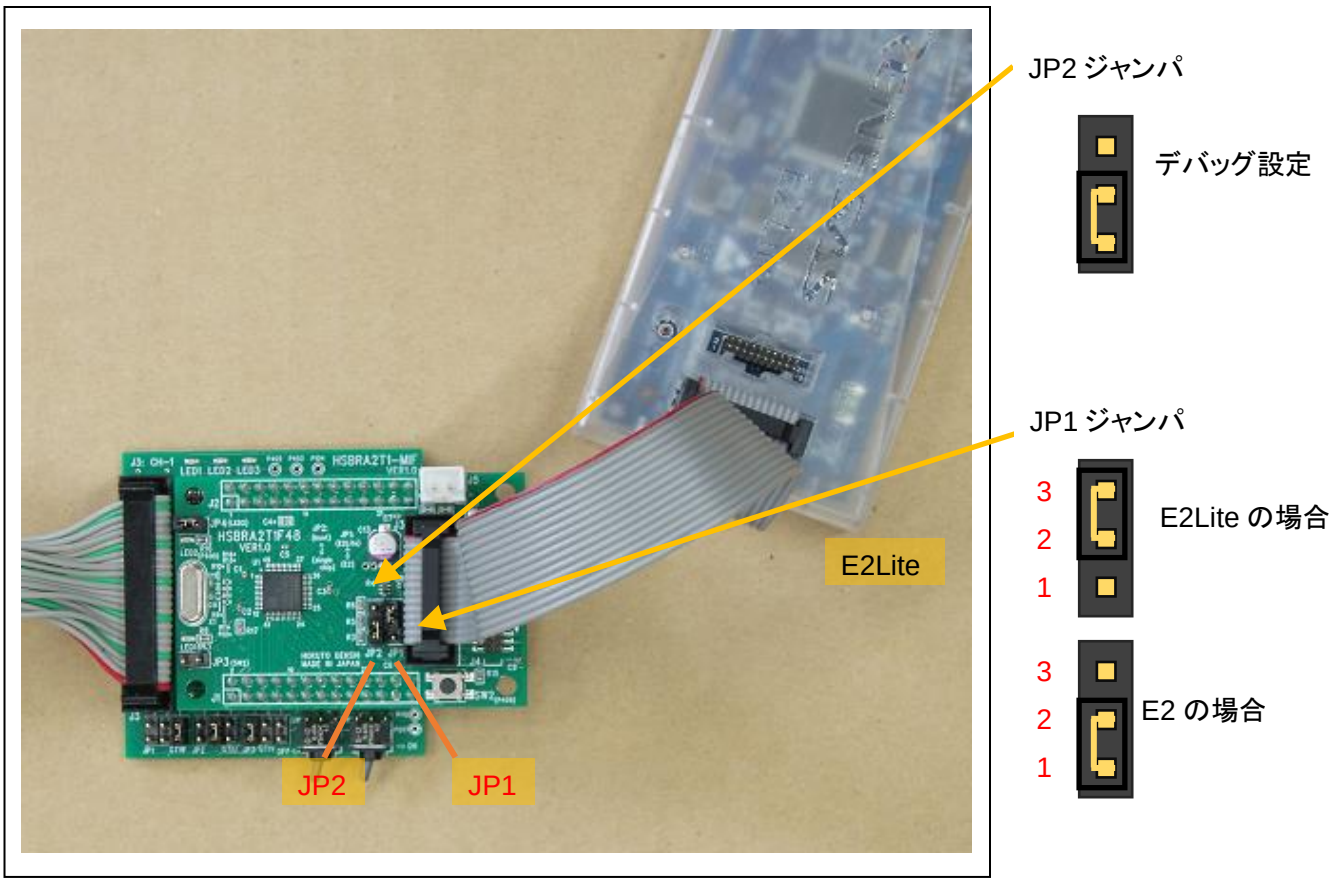
e2studio の Build(トンカチのアイコン)を押します。

```
プロパティー　問題　スマート・ブラウザー　コンソール　デバッグ
CDT ビルド・コンソール [RA2T1_BLMKIT_TUTORIAL1]
Building file: ../ra_gen/pin_data.c
Building target: RA2T1_BLMKIT_TUTORIAL1.elf
arm-none-eabi-objcopy -O srec "RA2T1_BLMKIT_TUTORIAL1.elf"  "RA2T1_BLMKIT_TUTORIAL1.srec"
arm-none-eabi-size --format=berkeley "RA2T1_BLMKIT_TUTORIAL1.elf"
   text    data     bss     dec     hex filename
   4056       4    3300    7360    1cc0 RA2T1_BLMKIT_TUTORIAL1.elf

14:06:26 Build Finished. 0 errors, 0 warnings. (took 2s.413ms)
```

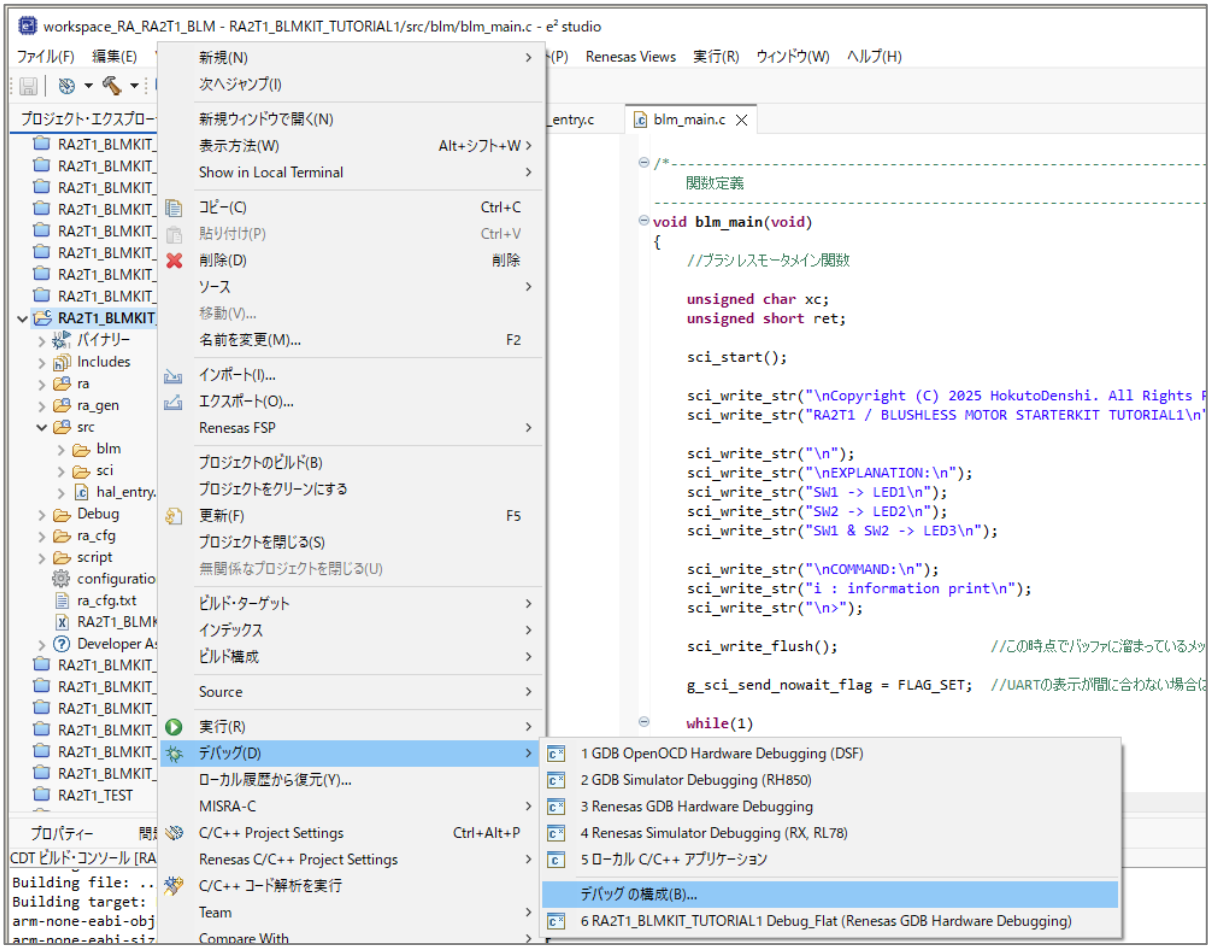
e2studio のコンソール上で、処理が進み、0 errors となれば問題ありません。

次に、ビルドしたプログラムをマイコンボード（正確には、マイコンボード上に搭載されているマイコンチップ）に書き込む必要があります。書き込みは、デバッガ（ルネサス E2, E2Lite）か USB-Serial 変換機器を使って行います。

ーデバッガ（E2, E2Lite）を使用してプログラムの書き込み（デバッグ実行）ー

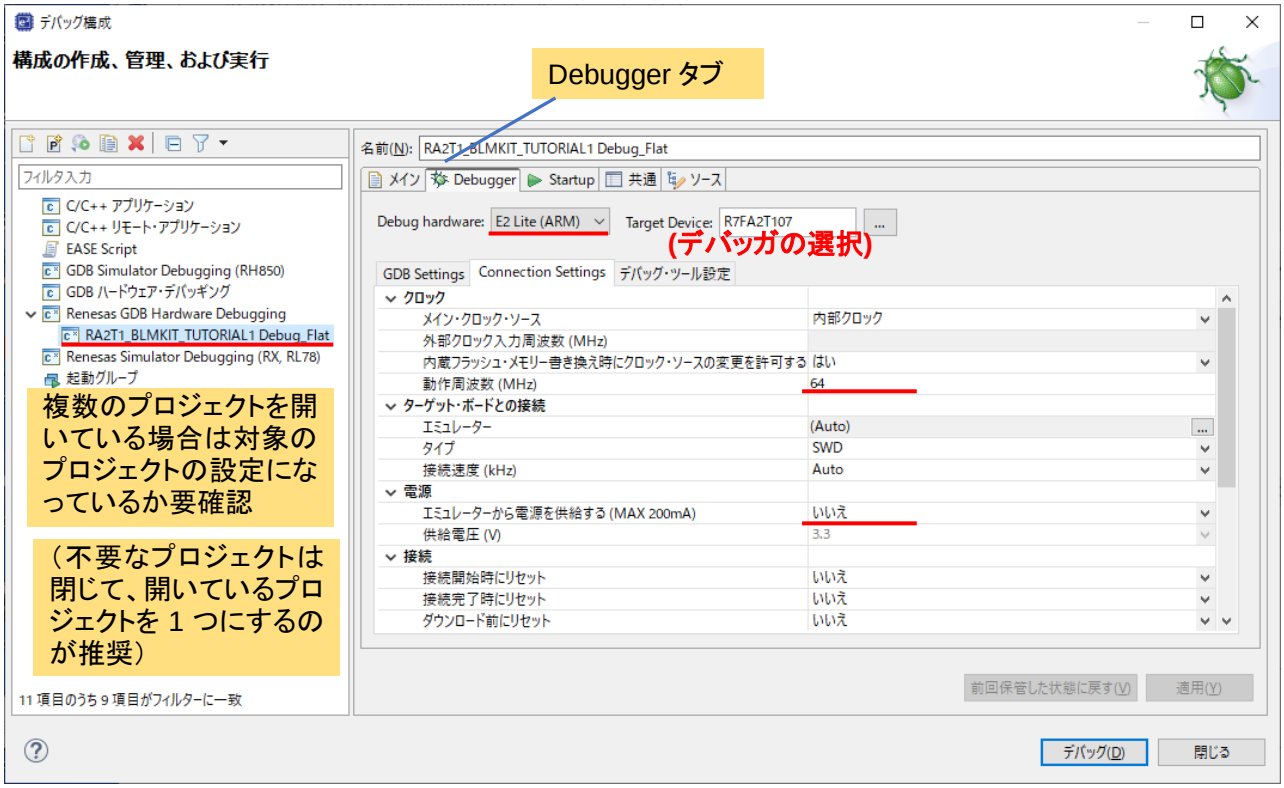


ルネサス E2 または E2Lite をお持ちの場合は、E2（もしくは E2Lite）をマイコンボード、J3（14P コネクタ）に接続します。JP1 ジャンパを、E2Lite の場合上側（2-3 ショート）、E2 の場合下側（1-2 ショート）に設定。



プロジェクトを右クリックして、

デバッグ – デバッグの構成



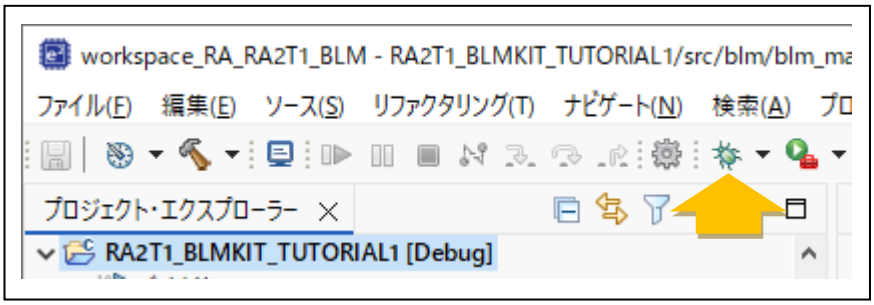
Renesas GDB Hardware Debugging の下の
　プロジェクト名　Debug_Flat

サンプルプロジェクトでは、E2Lite デバッガの設定となっています。
E2 デバッガを使用する際は、使用するデバッガを変更してください

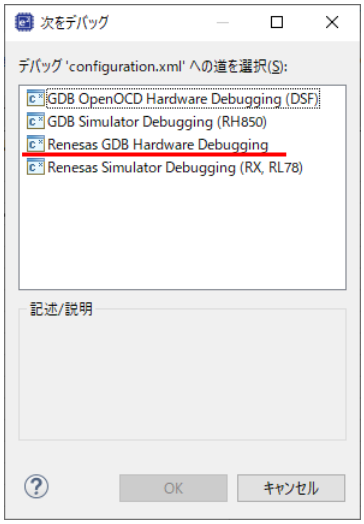
Debug Hardware　**使用するデバッガを選択**
動作周波数(MHz)　**64**
エミュレータから電源　を供給する(MAX 200mA)　**いいえ**

を選んでください。（デフォルトからの変更点）

「閉じる」を押す。

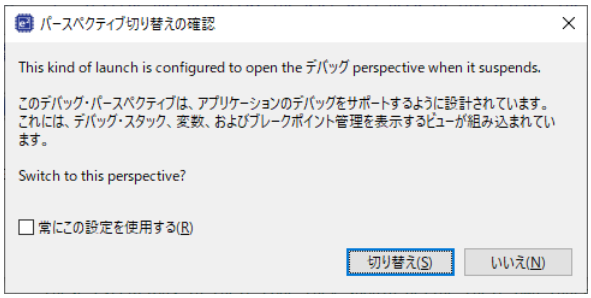


e2studio の Debug（虫のアイコン）を押します。

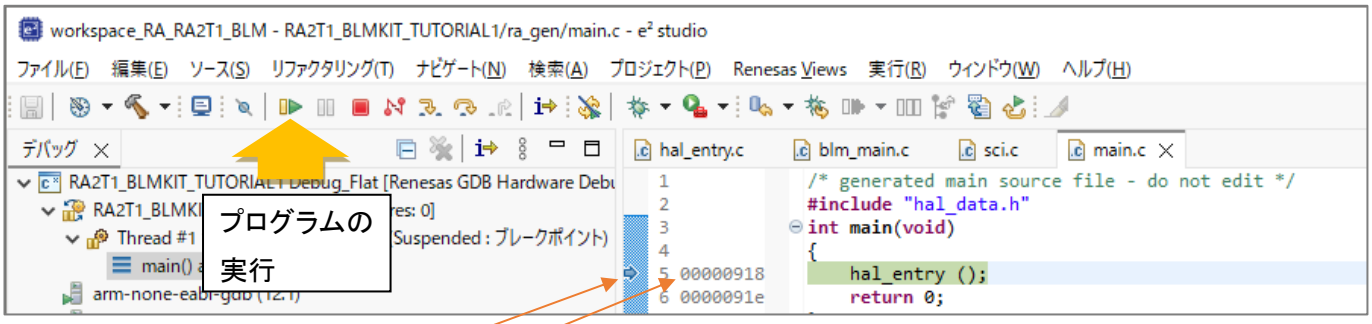


左記の画面が出た場合は、
Renesas GDB Hardware Debugging
を選択して、OK を押してください

configuration.xml（FSP　設定）がメインウィンドウで選択されている場合は、左記のダイアログが出ます。
（ソースコード（xx.c 等）が選択されている場合は出ません）



上記ダイアログが表示された場合は「切り替え」を押す。

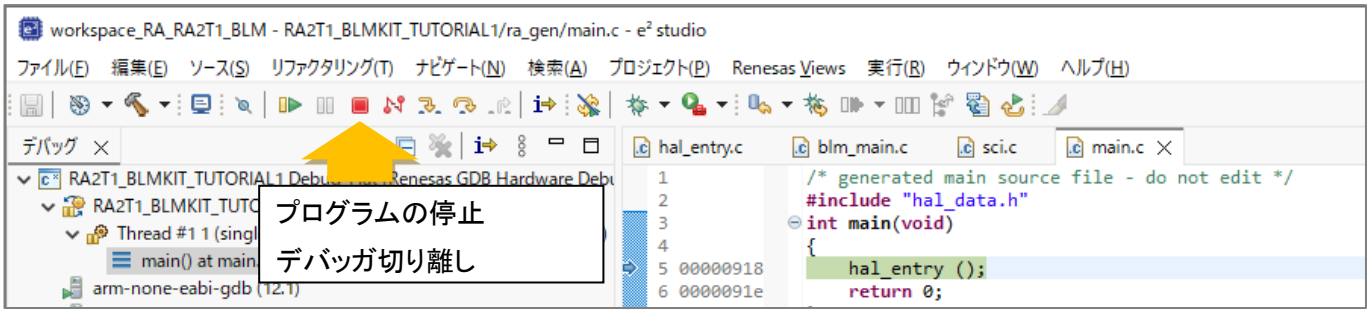


プログラムの停止位置
アドレスが表示される

プログラムのダウンロード（コンパイル・ビルドしたプログラムがマイコンチップの ROM に書き込まれる）が成功すると、上記の様な画面となります。

右三角のアイコンを押すとプログラムは実行されます。

（hal_entry()で一度停止しますので、もう一度プログラムの実行ボタンを押してください）



赤の四角のアイコンを押すと、プログラムは停止となり、デバッガ接続も切り離されます。E2/E2Lite の取り外しや、電源を落とすことが可能となります。

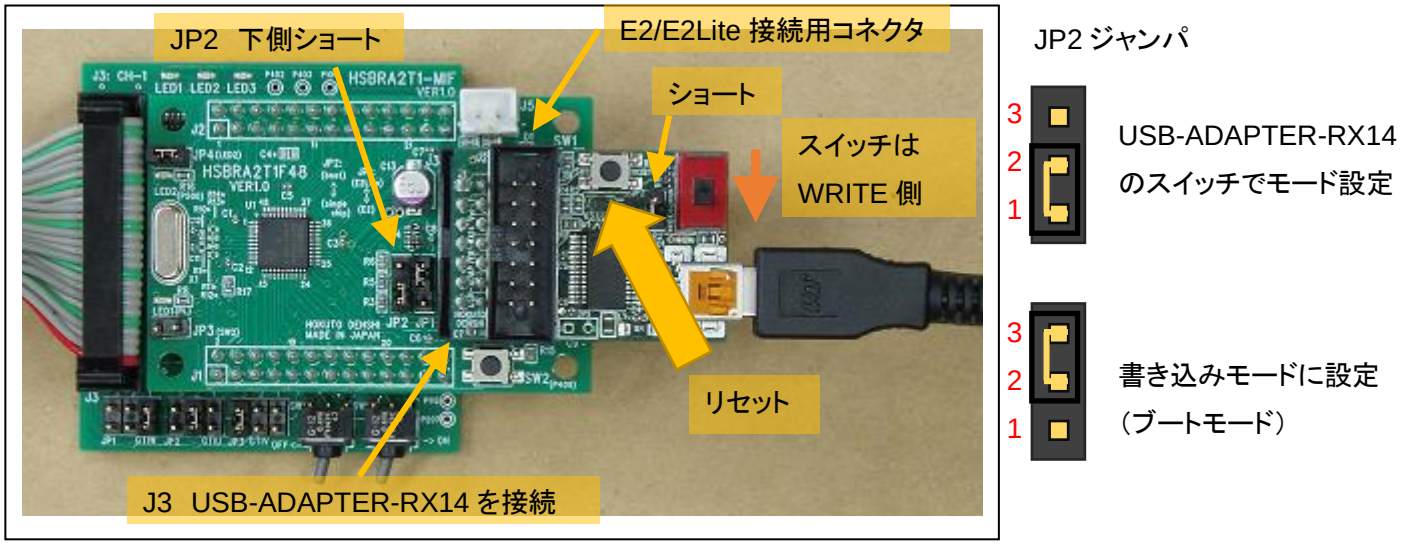
－RenesasFlashProgrammer を使用した書き込み－

デバッガでのダウンロードの他に、書き込みツールを使用して、マイコンの ROM にプログラムを書き込む方法があります。

(1)J3（14P コネクタ）に USB-ADAPTER-RX14 を接続
(2)USB-Serial 変換機器を J3 に接続
(3)E2（または E2Lite)を J3 に接続

(1)~(4)のいずれかで PC とマイコンボードを接続する。

(1)USB-ADAPTER-RX14 で接続

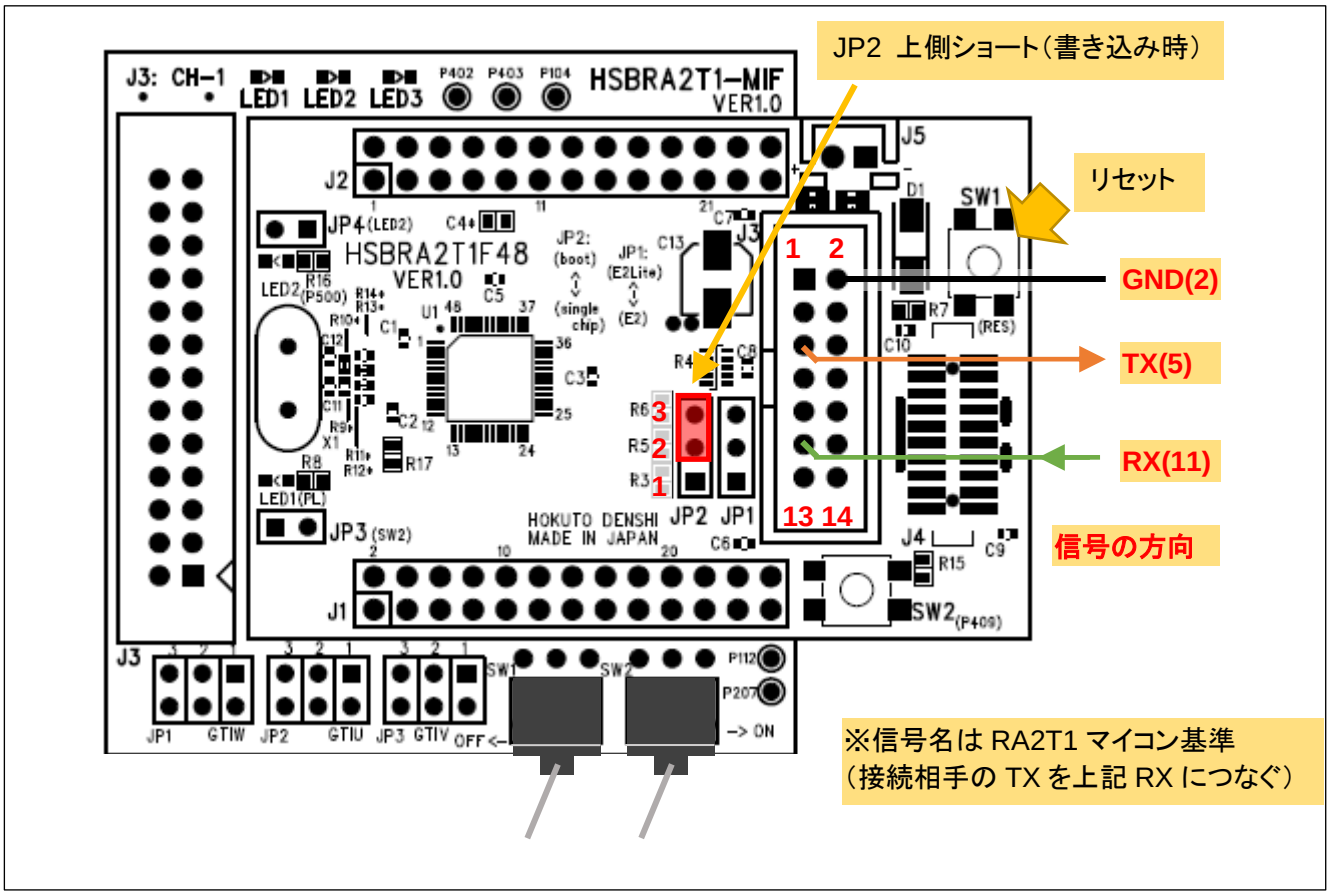


※USB-ADAPTER-RX14 は、別売品です

JP2 を 1-2 ショート（下側 2 ピンがショートされるように挿す）
USB-ADAPTER-RX14 の JP2 はショートに設定
USB-ADAPTER-RX14 のスイッチは WRITE 側（下側）に設定
※書込み後、プログラム実行時は、スイッチを RUN 側（上側）に切り替えてください）

マイコンボードの JP2 は、下側に設定すると、「書き込み」と「実行」モードが USB-ADAPTER-RX14 のスイッチで選択できるようになります。
（上側に設定した場合、USB-ADAPTER-RX14 の設定とは無関係に、「書き込み」モードとなります。）

(3)USB-Serial 変換機器を J3 に接続



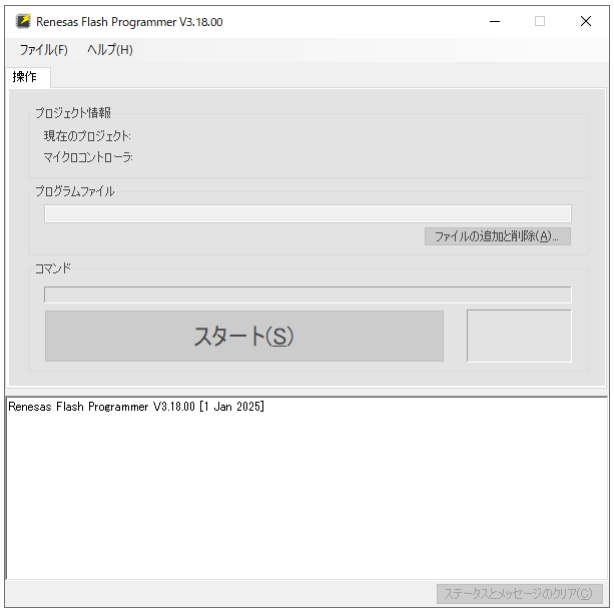
USB-Serial 変換機器は、5V の振幅で送受信できるものを使用してください。

J3（14P コネクタ）に、USB-Serial 変換機器の TX, RX, GND の信号線を接続してください。

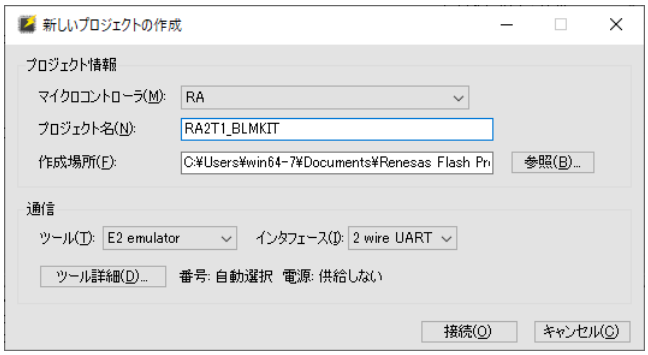
JP2 を 2-3 ショート（上側 2 ピンがショートされるように挿す）

※書込み後、プログラム実行時は、JP2 を「1-2 ショート（下側 2 ピンをショート）」、または「オープン（ジャンパピンを抜く）」としてください

RFP(RenesasFlashProgrammer)を起動。



ファイル – 新しいプロジェクトを作成



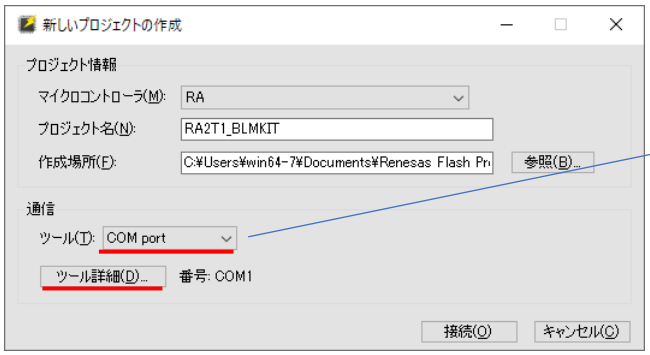
マイクロコントローラ　RA を選択

プロジェクト名　任意の名称を入力

ツール　デバッガを使用する場合は使用しているデバッガを選択

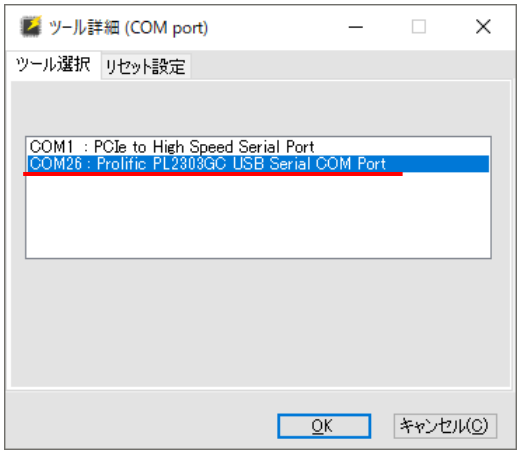
　　USB-ADAPTER-RX14 または USB-Serial 変換機器を使用する場合は COM port を選択

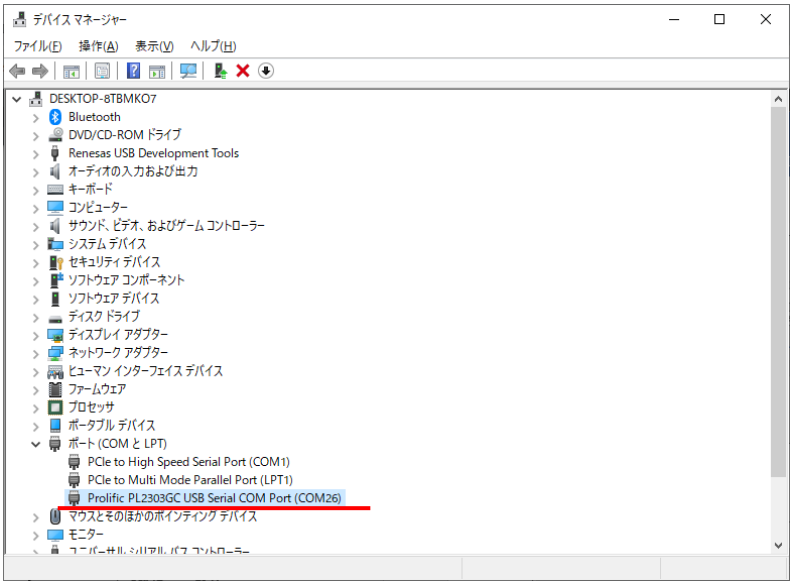
以下、USB-ADAPTER-RX14 を使う前提で説明します。



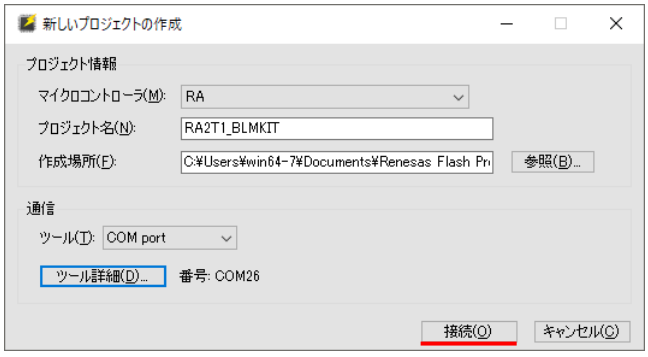
COM port を選択

ツール詳細 を押す

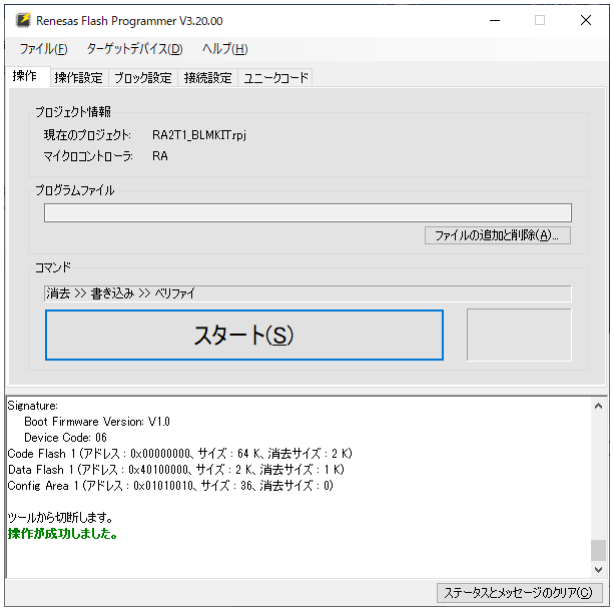




USB-ADAPTER-RX14 の場合は、「Prolific PL2303GC USB Serial Comm Port」として見えている、COM ポート番号を選択。（COM ポート番号か不明な場合は、USB ケーブルを抜いた際にデバイスマネージャ上で見えなくなるデバイスを選択してください。）



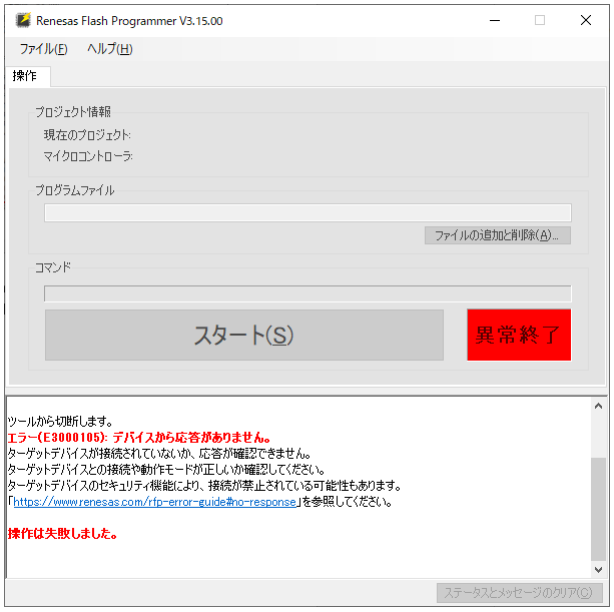
接続ボタンを押す



接続が成功しました　となれば問題ありません。

－エラーとなった場合－

・デバイスから応答がありません



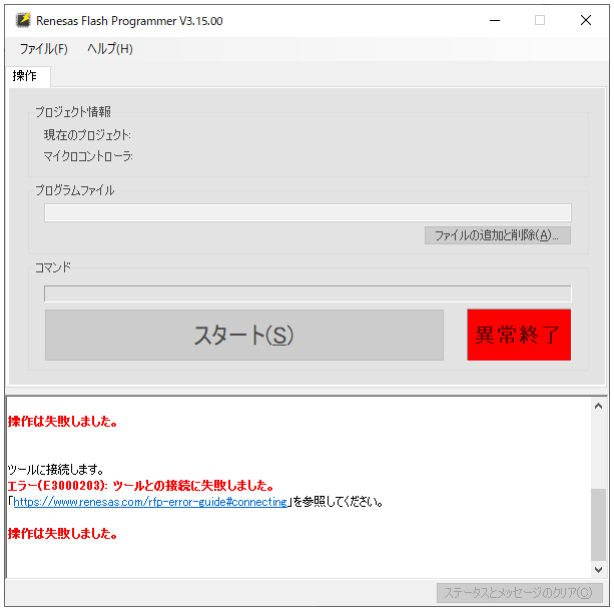
・USB-ADAPTER-RX14 のスイッチが WRITE 側になっている場合は、マイコンをリセット（USB-ADAPTER-RX14 上のプッシュスイッチを押す）してください
（電源を一度落として再投入する事でも可）

・USB-ADAPTER-RX14 のジャンパがショートに設定されている事を確認してください

・マイコンボードの JP2 の設定が前出の通りとなっている事を確認してください

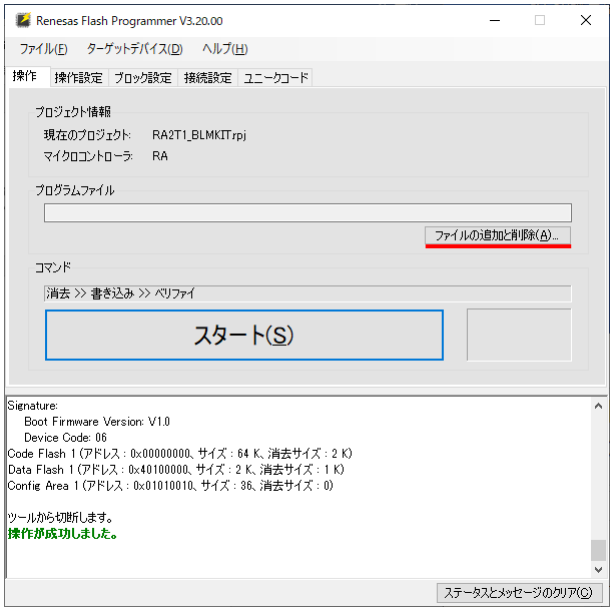
・COM ポート番号が間違えていないかを確認してください

・ツールとの接続に失敗しました

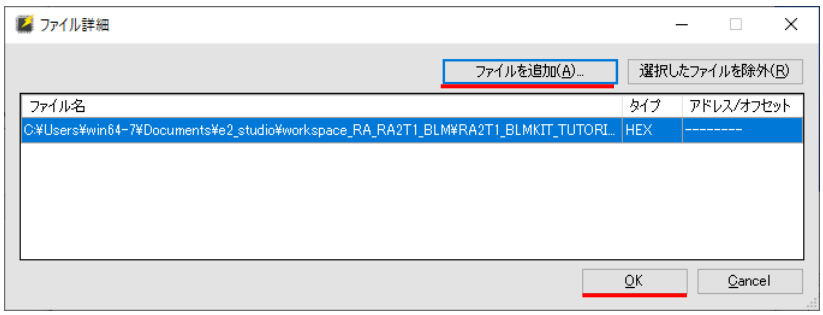


・COM ポート（この例では COM26）で、端末ソフト（teraterm 等）が開いていないか、COM26 を使用しているアプリケーションが存在しないかを確認してください（端末ソフトは閉じてください）

以下、接続が成功した場合の続きです。



ファイルの追加と削除　を押す。

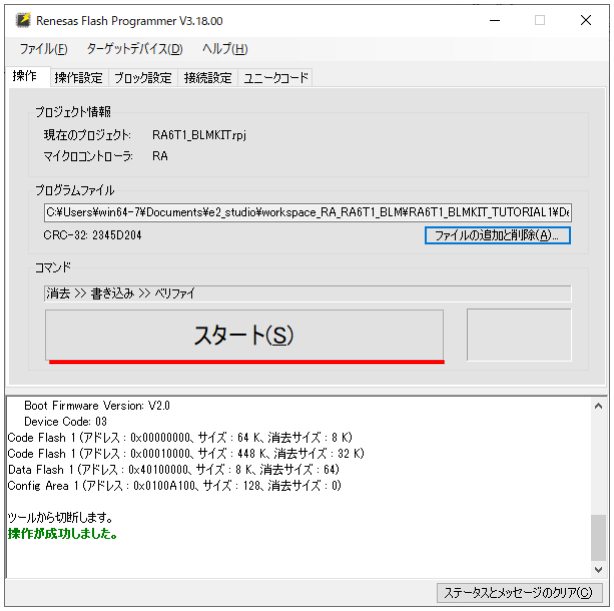


ファイルを追加 を押して、ビルドで生成した（Debug 以下の）srec ファイルを選択。
OK を押す

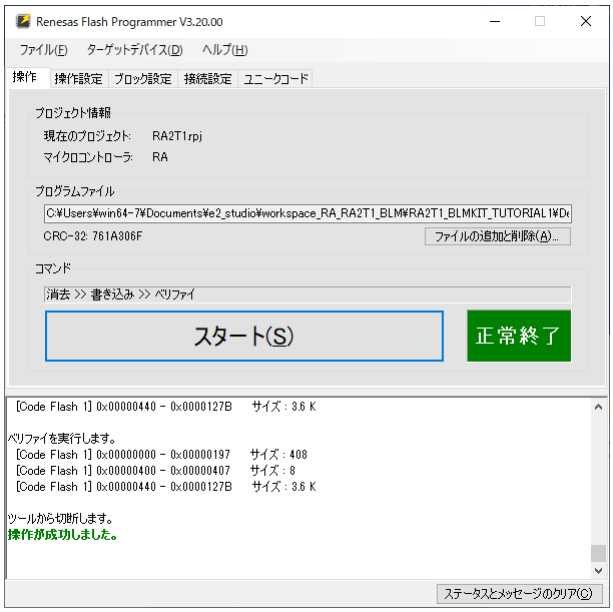
ここで、マイコンのリセットを行ってください。

・USB-ADAPTER-RX14 上のプッシュスイッチを押す
・マイコンボード上の SW1 を押す（USB-ADAPTER-RX14 の影となる位置にあるスイッチ）
・電源を一度落として再投入する

のいずれかを行う。（※デバッガをお使いの場合はリセットは不要です）



スタートを押す。



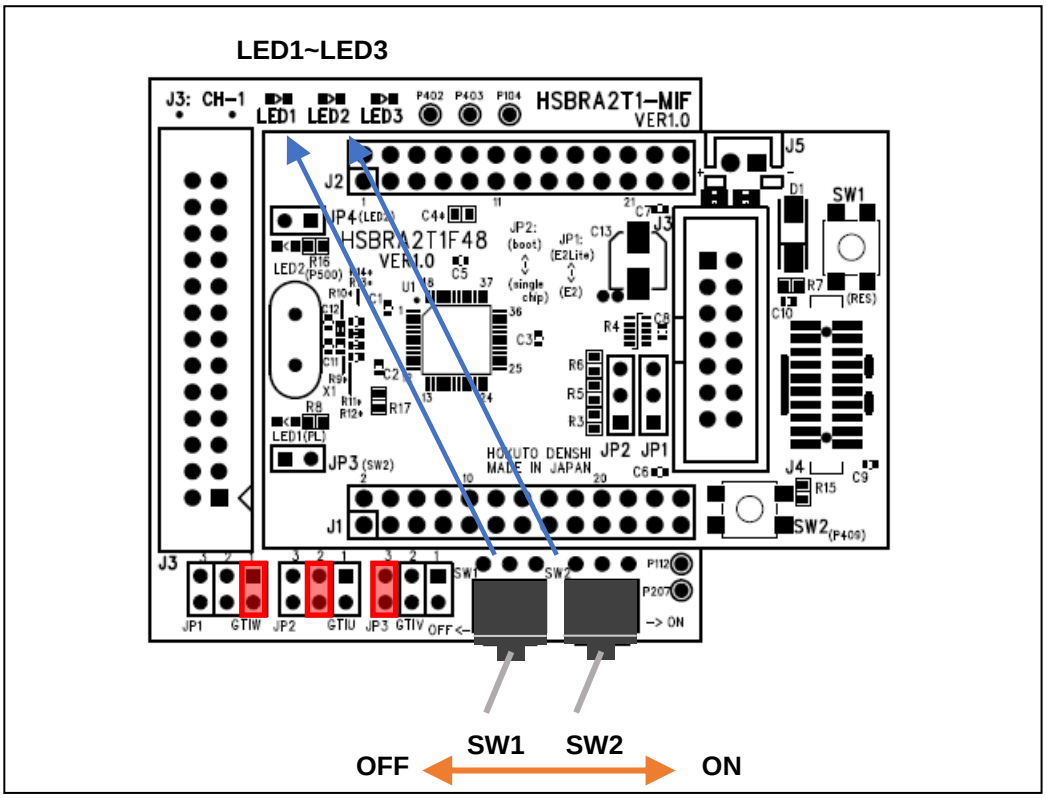
操作が成功しました、正常終了となれば問題ありません。

USB-ADAPTER-RX14 をお使いの場合は、スイッチを RUN 方向に切り替えてマイコンをリセットしてください。

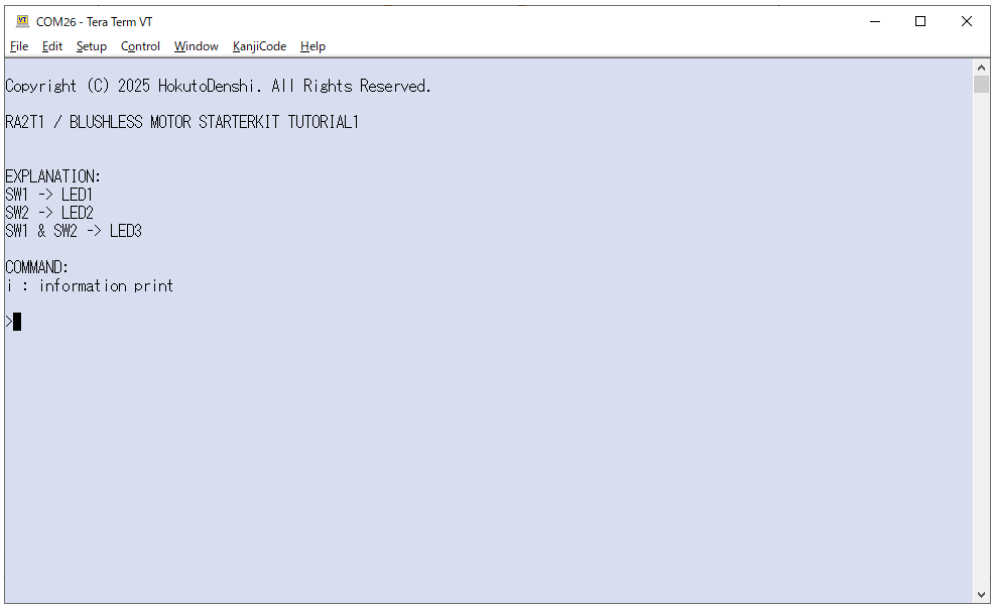
USB-Serial 変換機器を使用している場合は、JP2 のジャンパを下側に切り替えるか、ジャンパを抜いてマイコンをリセットしてください。

デバッガを使用して書き込みを行った場合は、デバッガを取り外してください。

デバッガでの実行時や、RenesasFlashProgrammer でプログラムを書き込んでリセットするとプログラムは実行されていますので、SW1~2 を切り替えてみてください。SW を ON 側に倒すと、対応する LED1~2 が点灯となれば、プログラムは期待通りの動作となっています。（LED3 は、SW1=SW2=ON の時に点灯）



USB-ADAPTER-RX14（もしくは USB-1S, USB-Serial 変換機器）をお使いの場合は、端末ソフト（teraterm 等）を開いて UART 通信の動作を確認してください。



端末は
速度　115,200bps, 8 ビット, パリティなし, 1 ストップビット
の設定で開いてください。

マイコンをリセットした際に、端末に上記表示が出力されれば、マイコン→PC 間の UART 通信は問題ありません。
端末からキーボードで i を入力してみてください。

```
>
SW1 -> OFF
SW2 -> OFF
```

i を入力する度に、SW の状態が表示されれば、PC→マイコンの UART 通信も問題ありません。

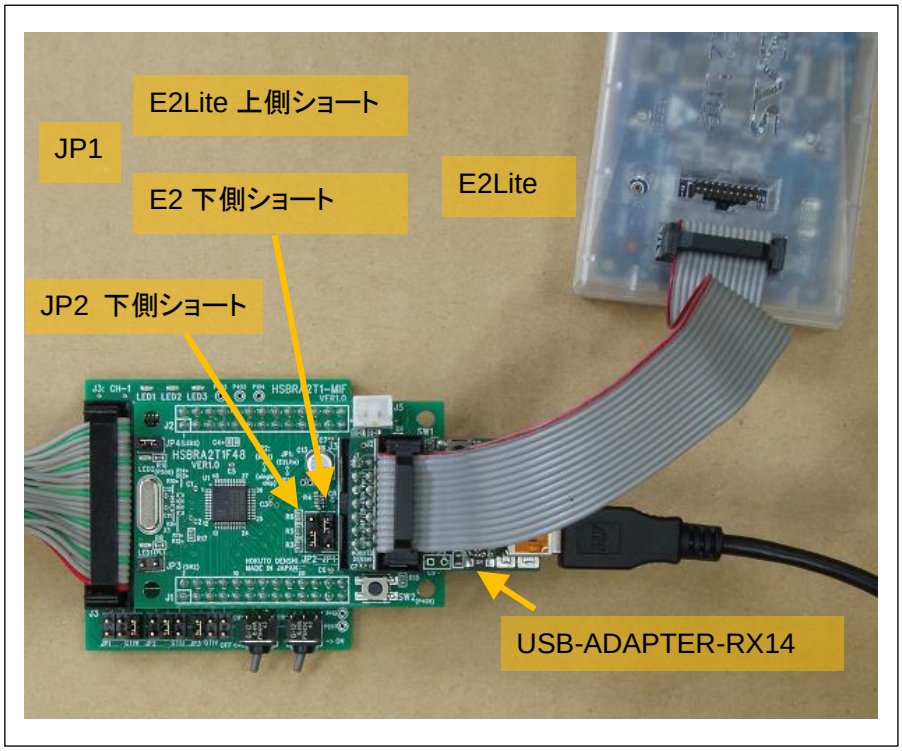
以降のチュートリアルでは、UART 通信を使用してモータの回転数を表示させたり、キーボードからの入力で動作を変えられたりするものがありますので、USB-ADAPTER-RX14（もしくは市販の USB-Serial 変換機器）が使える状態となっている事が望ましいです。

動作しない場合は、
・電源が入っているか（マイコンボード上の LED1(PL)が点灯しているか）
・マイコンボードが動作モードになっているか（JP2 のジャンパが抜けているか、下側ショートになっているか）
・USB-ADAPTER-RX14 のスイッチが RUN 側に切り替わっているか
・プログラムの書込みで失敗していないか
等を確認してください。

本チュートリアルの目的は、簡単なプログラムの作成と、実際にマイコンチップに書き込んで動作させるところまでとなります。

ソースコードを変えてみて、動作が変わるか等確かめてみてください。

・USB-ADAPTER-RX14 とデバッガの接続



USB-ADAPTER-RX14 をマイコンボードに挿したまま、USB-ADAPTER-RX14 上の 14P コネクタに E2, E2Lite を接続可能です。

デバッガ動作と、UART による通信は同時に行う事ができます。

（USB-ADAPTER-RX14 のスイッチは、RUN 側（上側）にセットしてください。）

※E2, E2Lite を使って書き込みやデバッグを行う際は、JP2 下側ショートとして下さい

TUTORIAL1 のプログラムの動作に関して簡単に説明致します。

src 以下の、blm/blm_main.c がメイン処理です。

blm_main.c

```
void blm_main(void)
{
    //ブラシレスモータメイン関数

    unsigned char xc;
    unsigned short ret;

    sci_start();

    sci_write_str("¥nCopyright (C) 2025 HokutoDenshi. All Rights Reserved.¥n¥n");
    sci_write_str("RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL1¥n");

    sci_write_str("¥n");
    sci_write_str("¥nEXPLANATION:¥n");
    sci_write_str("SW1 -> LED1¥n");
    sci_write_str("SW2 -> LED2¥n");
    sci_write_str("SW1 & SW2 -> LED3¥n");

    sci_write_str("¥nCOMMAND:¥n");
    sci_write_str("i : information print¥n");
    sci_write_str("¥n>");
```

先頭部分は、

・変数の定義

・UART 通信の開始　sci_start()

・メッセージの表示

を行っています。

```
    while(1)
    {
        //キーボードからの読み取り
        ret = sci_read_char(&xc);
        if (ret != SCI_RECEIVE_DATA_EMPTY)
        {
            switch(xc)
            {
                case 'i':
                    sci_write_str("¥nSW1 -> ");
                    if (BLM_SW_1_PORT == SW_ON)
                    {
                        sci_write_str("ON");
                    }
                    else
                    {
                        sci_write_str("OFF");
                    }

（中略）
                    sci_write_str("¥n¥n");

                    break;
            }
        }
```

次に、キーボードからの入力を読み取り、入力された文字が'i'であれば SW の状態を表示する様にしています。

```
        //SW と LED の連動
        if (BLM_SW_1_PORT == SW_ON) BLM_LED_1_PORT = LED_ON;
        else BLM_LED_1_PORT = LED_OFF;

        if (BLM_SW_2_PORT == SW_ON) BLM_LED_2_PORT = LED_ON;
        else BLM_LED_2_PORT = LED_OFF;

        if ((BLM_SW_1_PORT == SW_ON) && (BLM_SW_2_PORT == SW_ON)) BLM_LED_3_PORT = LED_ON;
        else BLM_LED_3_PORT = LED_OFF;
```

スイッチと LED の ON/OFF を連動させる部分です。

blm.h（定数定義）

```
/*----------------------------------------------------------------------
定数定義
----------------------------------------------------------------------*/
//LED, SW
#define BLM_LED_1_PORT      R_PORT9->PODR_b.PODR13
#define BLM_LED_2_PORT      R_PORT9->PODR_b.PODR14
#define BLM_LED_3_PORT      R_PORT9->PODR_b.PODR15
#define BLM_LED_4_PORT      R_PORT5->PODR_b.PODR0       //マイコンボード上の LED2

#define LED_ON      0
#define LED_OFF     1                                   //マイコンボード上の LED は逆

#define BLM_SW_1_PORT       R_PORT3->PIDR_b.PIDR1
#define BLM_SW_2_PORT       R_PORT3->PIDR_b.PIDR2

#define SW_ON       0
#define SW_OFF      1
```

TUTORIAL1 では、マイコンボードへのプログラムの書き込み及び、汎用 I/O の入出力とスイッチが押された場合の割り込みの処理、UART 通信（端末への情報表示と、端末からキーボードの読み取り）が行えるようになるのが目的です。

・汎用 I/O への出力

LED1 を点灯させる場合 （P913=L 出力）

→ BLM_LED_1_PORT = LED_ON;　(R_PORT9->PODR_b.PODR13 = 0;)と等価

LED1 を消灯させる場合 （P913=H 出力）

→BLM_LED_1_PORT = LED_OFF;　(R_PORT9->PODR_b.PODR13 = 1;)と等価

・SW1 が ON か OFF かを検出

if (BLM_SW1_PORT == SW_ON)　(if (R_PORT3->PIDR_b.PIDR1 == 0))と等価

{

　//SW1 が ON の時の処理を記載

}

else

{

```
  //SW1 が OFF の時の処理を記載
}
```

・端末への文字出力

```
sci_write_str("message¥n"); //¥n は改行
```

・端末からの文字入力

```
unsigned char xc;
ret = sci_read_char(&xc);  //関数の戻り値(ret)が SCI_RECEIVE_DATA_EMPTY の場合はキーボードからの入力なし
```

キーボードからの文字入力があると、xc に文字コード（i の場合は 0x69）が入ります。

以上で、最初のチュートリアルは終了となります。

FSP の設定からプログラムのビルド、書き込み、実行とプログラム開発の一通りのフローを経験するチュートリアルですので、RA でのプログラム開発を行った事があれば、本チュートリアルはスキップして頂いて問題ありません。

・チュートリアル 1 での端子設定

端子名	役割	割り当て	備考
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

・チュートリアル 1 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	

1.2. モータに電流を流す

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL2

モータドライバボードと、マイコンボードは接続してください。

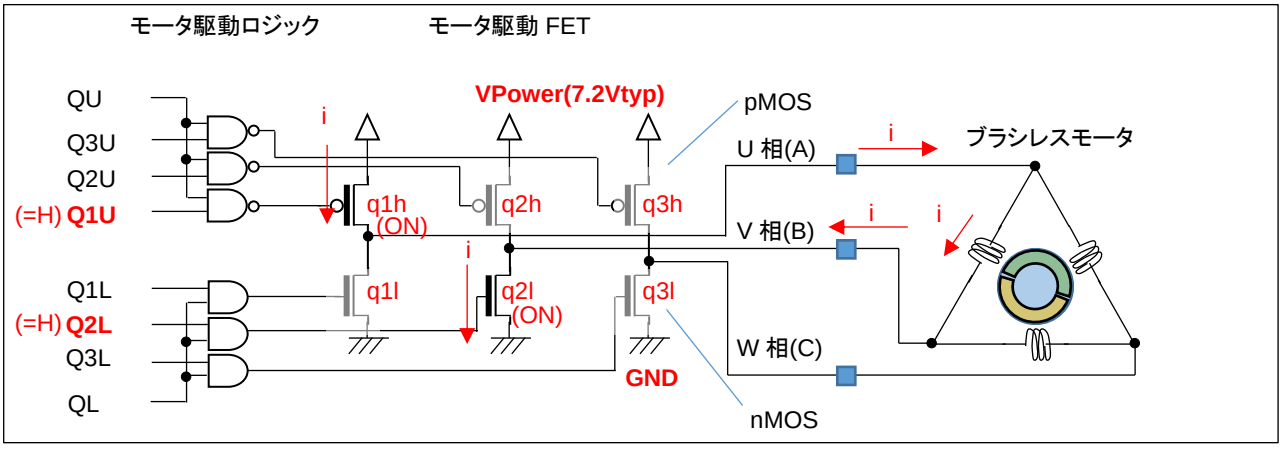
起動すると、0.5 秒毎に LED1-LED3 の点灯が切り替わります。SW1 を ON にすると、接続したモータドライバボードに、モータに電流を流す信号が送られます。SW1 を OFF にすると信号は止まります。

信号が送られている状態で、LED 点灯・消灯が切り替わるタイミングで、モータから「カチッ」という音が聞こえてくると思います。このとき、モータに電流を流す制御を行っています。モータは、U(A), V(B), W(C)の 3 本のワイヤでモータドライバボードとつながっています。

※モータ側では、端子に A, B, C と書かれていますが、以降の解説では U, V, W 相という記載を用います、U=A, V=B, W=C です。

3 本のワイヤに対し、モータドライバボード上で、U に H 側の電源(7.2V)を接続し、V に L 側の電源(GND=0V)を接続した場合モータ内部で、U 端子から V 端子に対して電流が流れます。単純に、3 本のワイヤ(UVW)のうち 2 本をアクティブ(片方を電源、もう一方を GND に接続)とする場合、電流の流れ方としては 6 通りあります。

・U 相から V 相に電流を流す設定



トランジスタの駆動パターンですが、モータに電流を流す際は、

H 側	U 相(q1h)	V 相(q2h)	W 相(q3h)
L 側	U 相(q1l)	V 相(q2l)	W 相(q3l)

合計 6 個のトランジスタの内、H 側 1 箇所、L 側 1 箇所を ON させます。

例えば、

q1h と q2l を ON させた場合、Vpower→モータの U 相端子→モータの V 相端子→GND に電流が流れます。…(a)

また、

q2h と q1l を ON させた場合、Vpower→モータの V 相端子→モータの U 相端子→GND に電流が流れます。…(b)

(a)と(b)では、電流が逆方向となります。

q1h と q1l（U 相の H 側と L 側）を ON させる制御は禁止です（モータには電流が流れず、電源間がショートする）。

禁止の組み合わせを省くと、下記の 6 通りとなります。

	(1)	(2)	(3)	(4)	(5)	(6)
H 側	q1h=ON	q1h=ON	q2h=ON	q2h=ON	q3h=ON	q3h=ON
L 側	q2l=ON	q3l=ON	q3l=ON	q1l=ON	q1l=ON	q2l=ON
電流の方向	U→V	U→W	V→W	V→U	W→U	W→V

上記(1)~(6)の様に制御する事により、U, V, W の 3 相の内 2 本にどちらの向きでも電流を流す事が可能です。

本チュートリアルプログラムでは、6 通りの電流を 500ms 毎に切り替えて流すようにしています。

なお、電流は 500ms 間流すわけではなく、LED の点灯パターンが変化した瞬間 50us の間流す様にしています。

モータに電流を流しているときに、モータの軸に触ると、「カチッ」と音がするタイミングで、わずかに動く感じが指に伝わってくるかと思います。電流が流れる時間は、一瞬のため、モータの軸が動くまではいきませんが、電流を流す時間を増やすとモータの軸の動きが大きくなるというイメージです。また、6 通りの電流の切り替えのタイミング（本チュートリアルでは 500ms）は、モータの回転数に影響するイメージです。

プログラムでは、
・電流を流す時間(50us)
を変更する事が出来ます。

blm_main.c 内で、

```
void blm_main(void)
{
//ブラシレスモータメイン関数

const float motor_on_time = 50.0e-6f;//[変更可]モータ通電時間 50[us]（デフォルト）
/*
モータの通電時間が長い場合、過大な電流が流れますので、最大でも 100[us]程度としてください
*/

const float motor_cycle_time = 500.0e-3f;       //[変更可]モータ回転周期 500[ms]（デフォルト） 回転周期の
1/6 の値（1/6 回転に掛かる時間）
```

赤字の部分で、電流を流す時間 50us と電流方向切り替わりタイミング 0.5 秒を定義しています。

・電流を流す時間：50.0e-6f の部分
・電流の切り替えタイミング：500.0e-3f の部分

上記部分を変えると、タイミングを変えることができますので試してみてください。

（motor_on_time の値は、あまり大きな値にしないでください。～100us 以下を目安に設定する事が推奨です。）
（※モータ内部はコイル（インダクタンス）で構成されており、長時間（＝DC 的に）電圧を印加すると、コイルのインピーダンスが下がり過大な電流が流れるためです）

本プログラムでは、2 つのタイマ（50us と 500ms）を使っています。50us の方は AGT0、500ms の方は AGT1 です。

タイマ	用途	設定時間	クロック源	カウンタ
AGT0	通電時間	50us	PCLKB	32bit
AGT1	電流切り替わりタイミング	500ms	PCLKB	32bit

AGT は、汎用的に使えるタイマで、カウンタは 32bit です。クロック源は選択可能ですが、PCLKB(=32MHz)（もしくは、PCLKB を分周したクロック）を選択しています。

PCLKB をベースにした場合、分周比は最大 8、カウンタは 32bit(2^{32}=4,294,967,296)ですので、設定可能な最大の周期は

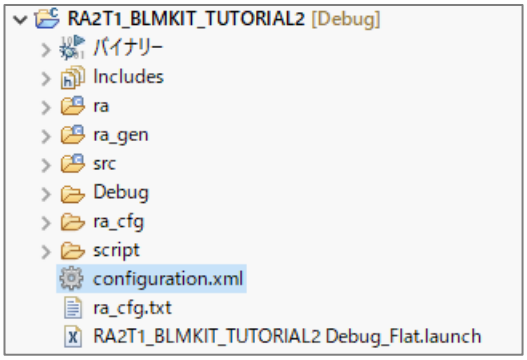
31.25ns × 8 × 2^{32} = 1073.7s (@PCLKB=32MHz)
となります。

モータ制御プログラムでは、タイマは必ず使用する機能となりますので、マイコンのハードウェアマニュアルを参照し、タイマの分解能（1 カウントの時間）や、設定範囲から、適切なタイマを選択していく事となります。
（本チュートリアルでは、AGT を使っていますが、以降のチュートリアルでは別なタイマも使用しています。）

モータに電流を流す処理は、電流の方向を設定後、モータに通電し、タイマ(AGT0=50us)時間経過後に電流を止めるというものです。

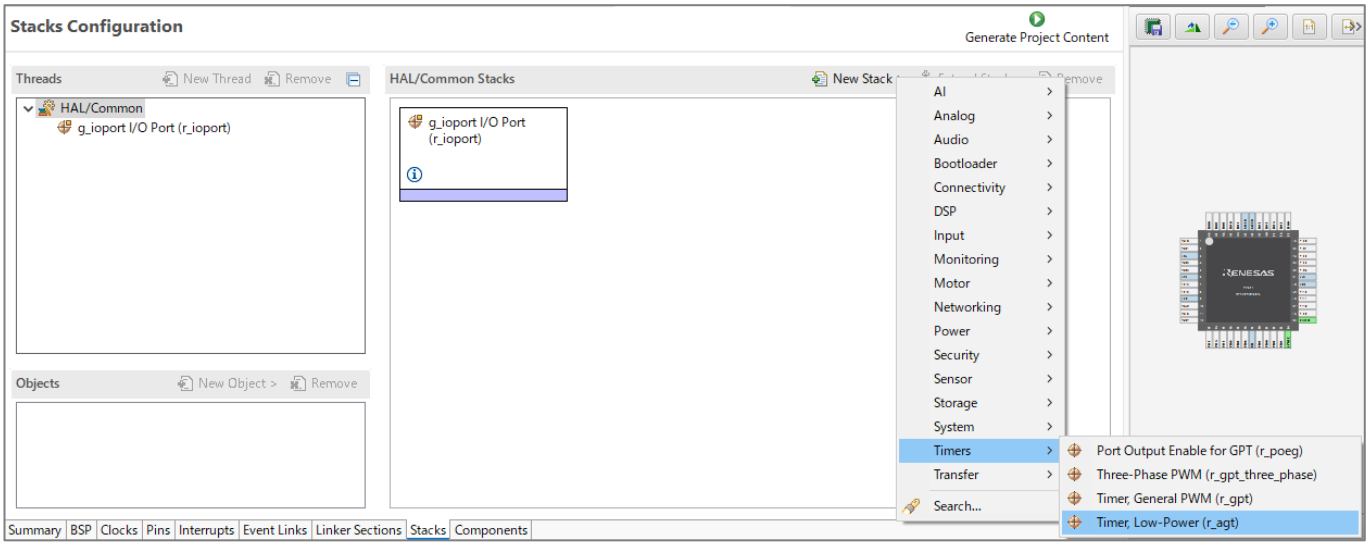
タイマ(AGT)の設定は、FSP を使用して行っています。スタックの追加で、

configuration.xml



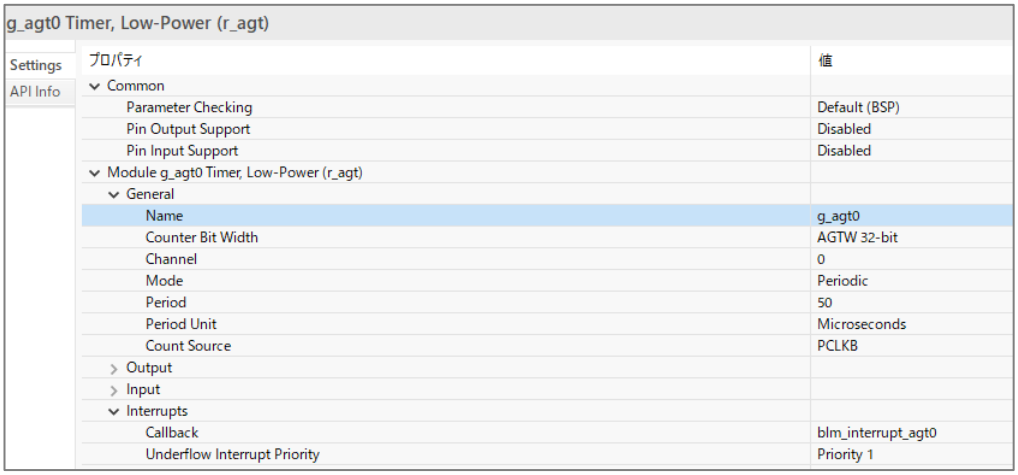
Stacks タブから、

New Stack – Timers – Timer Low-Power(r_agt)



を選択。

・プロパティ



		設定値	備考
General	Name	g_agt0	使用タイマ AGT0 と名称を合わせる
	Channel	0	AGT0 を使用
	Period	50	タイマ周期の設定
	Period Unit	Microseconds	単位[us]
Interrupts	Callback	blm_interrupt_agt0	タイマの周期終わりで呼び出される関数名の定義
	Underflow Interrupt Priority	Priority 1	割り込み優先度 1

Name は初期値は、g_timer0, g_timer1...の様な名称となります。ここでは、AGT を使っている事、どのタイマを起動しているのか等が判り易い様、周辺機能名(AGT0)と Name を合わせる様に設定しています。（Name は、プログラムする人が判れば、どのような名称でも問題ありません。）その他、タイマの使用チャネル番号や、周期時間(50us)等の設定を行っています。

Callback は、タイマ周期のタイマ起動後 50us 経過時に呼び出される関数名を指定します。プログラムソース内では、blm_interrupt_agt0 という関数を定義しています。Priority は割り込み優先度です。0~3, Disable の 5 つの選択肢の中から選択します。数値が小さい方が優先度が高く、優先度 1 の割り込み実行中に、優先度 0 の割り込みが掛った際には、割り込み 1 の割り込みルーチンの実行の途中で、先に割り込み 0 の割り込みルーチンが処理されます（多重割り込みが有効です）。

上記で、g_agt0(Timer Low-Power(r_agt))を追加した後、もう 1 つ、Timer Low-Power(r_agt)を追加します。2 つ目の AGT タイマは、

		設定値	備考
General	Name	g_agt1	使用タイマ AGT1 と名称を合わせる
	Channel	1	AGT1 を使用
	Period	500	タイマ周期の設定
	Period Unit	Milliseconds	単位[ms]
Interrupts	Callback	blm_interrupt_agt1	タイマの周期終わりで呼び出される関数名の定義
	Underflow Interrupt Priority	Priority 2	割り込み優先度 2

上記設定値としています。RA2T1 に搭載している AGT タイマは、2 チャンネルなので、Timer Low-Power(r_agt)は、2 つまで使用可能です。3 つ以上のタイマを使用したい場合は、別なタイマを使用する必要があります。

FSP でタイマを使用する場合、GUI で動作モードや周期等設定が行えますので、タイマ機能を制御するプログラムコードを書き下す必要はありません。

上記で設定しているのは初期値ですので、50us や 500ms という時間を変える場合、
・前出のプログラムコードを変更する
・初期値を FSP 上で変更する
のどちらでも有効です。

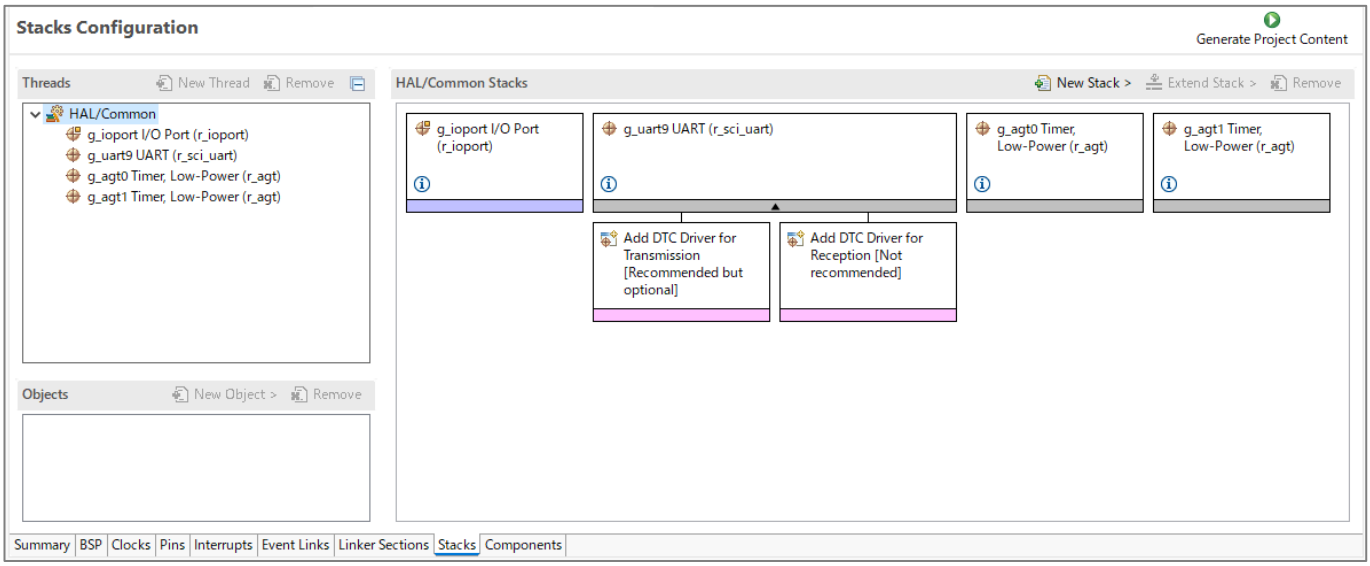
※FSP の設定を変更した場合「Generate Project Content」のボタンを押すのを忘れない様にしてください
（このボタンを押した際に、FSP で生成されるプログラムコードが更新されます）

※FSP で時間を設定する場合は、プログラムコード内の
blm_main.c 内

```
#if 1   //FSP 上で設定したタイマ設定時間を変えない場合は、#if 0 に変更

    timer_info_t agt0_info, agt1_info;

    //周期カウンタ値の取得
    (void) R_AGT_InfoGet(&g_agt0_ctrl, &agt0_info);
    (void) R_AGT_InfoGet(&g_agt1_ctrl, &agt1_info);

    //モータ通電時間（デフォルト 50us）の周期カウンタ値の変更
    (void) R_AGT_PeriodSet(&g_agt0_ctrl, (unsigned long)((float)agt0_info.period_counts *
(motor_on_time / agt0_setting_time)));

    //モータ回転周期（デフォルト 500ms）のカウンタ値の変更
    (void) R_AGT_PeriodSet(&g_agt1_ctrl, (unsigned long)((float)agt1_info.period_counts *
(motor_cycle_time / agt1_setting_time)));

#endif
```

#if 1 の部分を#if 0 に変えてください。

チュートリアル 2 でのスタックの構成は以下の様になります。



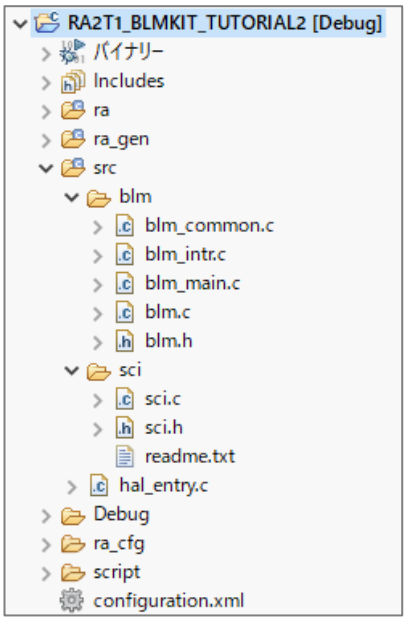
チュートリアル 1 同様に r_sci_uart(SCI9)を追加

r_agt(AGT0)

r_agt(AGT1)

の 2 つのタイマを追加。

src フォルダ以下ですが、



hal_entry.c（デフォルトで用意されているエントリポイントのソース）

に加え、

blm（ブラシレスモータ向けのソースコード格納フォルダ）

ファイル名	内容	備考
blm.c	モータ制御プログラム関数	チュートリアルによって変化
blm.h	モータ制御プログラム共通ヘッダ	
blm_common.c	モータ制御プログラム関数、共通部分	チュートリアル非依存の関数
blm_intr.c	モータ制御プログラム割り込み関数	
blm_main.c	モータ制御プログラムメインプログラム	

sci（UART(SCI)向けのソースコード格納フォルダ）

ファイル名	内容	備考
sci.c	UART(SCI)プログラム関数	
sci.h	UART(SCI)ヘッダ	
readme.txt	UART(SCI)関数の使い方	使い方を説明したテキストファイル

となっており、この構成は今後も共通です。

hal_entry.c 内で、blm_main()を呼ぶようにしているのも、チュートリアル 1 と同じで、今後のチュートリアルでも同様です。

タイマ	タイミング	処理内容
AGT0	50us 周期	モータの通電時間
AGT1	500ms 周期	モータの電流方向の切り替え

本チュートリアルでは、2 つの AGT タイマを使用しています。

AGT0 タイマ起動後、50us 経過後に、
blm_interrupt_agt0()
関数が実行されます。

同様に、AGT1 タイマ起動後、500ms 経過後に、
blm_interrupt_agt1()
関数が実行されます。これらは、FSP の Callback の設定で設定した関数名です。

AGT1 タイマは、起動後に止めることがないので、定期的（500ms 毎）に、blm_interrupt_agt1 が呼ばれます。

・blm_intr.c（モータ制御割り込み処理を記載したソース）

```
void blm_interrupt_agt0(timer_callback_args_t *p_args)
{                                          blm_interrupt_agt0 は、R_AGT_Start()の 50us 後
//50us 割り込み                            （AGT0 で設定した時間）に実行される

//既定の時間経過するとモータに流れる電流を止める

    if (TIMER_EVENT_CYCLE_END == p_args->event)
    {
        blm_drive[BLM_CH_1](BLM_OFF_DIRECTION);   通電終了

        (void) R_AGT_Stop(&g_agt0_ctrl);    //50us タイマは停止
    }
}

void blm_interrupt_agt1(timer_callback_args_t *p_args)
{                                          blm_interrupt_agt1 は、500ms
//500ms 割り込み                           （AGT1 で設定した周期）に 1 回実行される

//モータに印加する電流の方向を切り替える

    static unsigned short current_pattern = 1;  //初期値

    if (TIMER_EVENT_CYCLE_END == p_args->event)
    {

        //電流の向きに応じて、モータドライバボード上の LED(LED1-6)の点灯状態を切り替える
        switch (current_pattern)                                  1→2→3→4→5→6→1
        {                                                         の繰り返し
            case 1:
                blm_led_out(BLM_LED_1);           //LED1 を点灯
                break;

            case 2:
                blm_led_out(BLM_LED_2);           //LED2 を点灯
                break;

            case 3:
                blm_led_out(BLM_LED_3);           //LED3 を点灯
                break;

            case 4:
                blm_led_out(BLM_LED_1 | BLM_LED_4);//LED1 + LED4 を点灯
                break;

            case 5:
                blm_led_out(BLM_LED_2 | BLM_LED_4);//LED2 + LED4 を点灯
                break;

            case 6:
                blm_led_out(BLM_LED_3 | BLM_LED_4);//LED3 + LED4 を点灯
                break;
        }

        //切り替えるタイミングで 50us タイマを ON させる
        (void) R_AGT_Reset(&g_agt0_ctrl);
        (void) R_AGT_Start(&g_agt0_ctrl);    //50us タイマを起動    AGT0 タイマスタート

        //電流の流れる方向のパターンを設定
        blm_drive[BLM_CH_1](current_pattern);   通電開始

        //current_pattern を 1-6 の順番に切り替えてゆく
        current_pattern++;
        if (current_pattern > 6)
        {
            current_pattern = 1;//6 を超えたら 1 に戻る
        }
    }
}
```

blm_interrupt_agt1()は、500ms に 1 回定期的に実行されます。blm_interrupt_agt0()は、AGT0 タイマスタート後 50us 経過後に実行されます。AGT0（50us タイマ）は、blm_interrupt_agt0 内で停止されます。

blm_drive_[]()関数は、モータに引数に応じた方向に電流を流す制御を行う関数です。

・blm.c

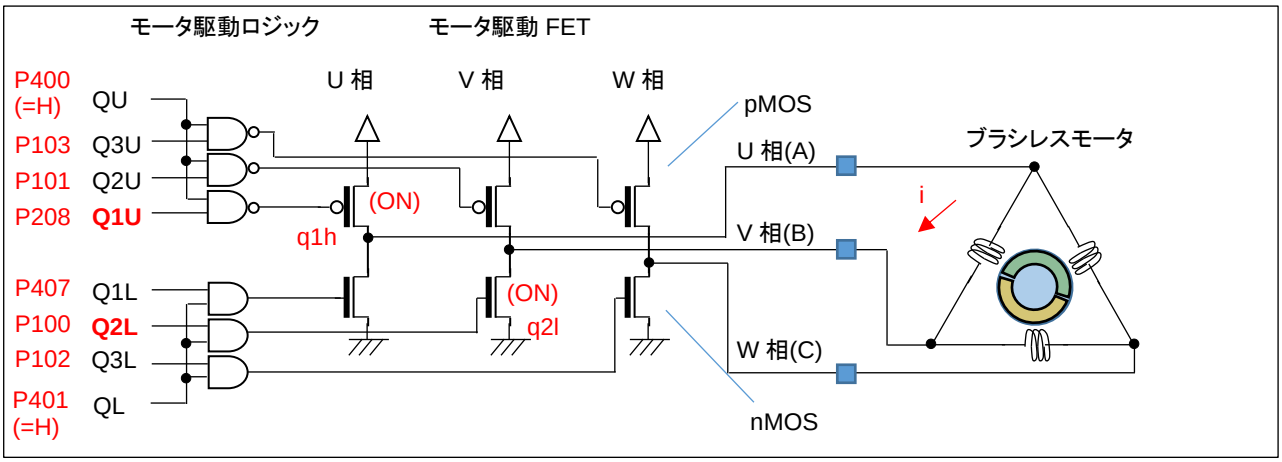
```
void blm_drive_ch1(unsigned short direction)
{
    //ブラシレスモータ CH-1 制御関数

    //引数
    // direction
    //  OFF_DIRCTION  : 電流 OFF
    //  U_V_DIRECTION : U→V に電流を流す様制御
    //  U_W_DIRECTION : U→W に電流を流す様制御
    //  V_W_DIRECTION : V→W に電流を流す様制御
    //  V_U_DIRECTION : V→U に電流を流す様制御
    //  W_U_DIRECTION : W→U に電流を流す様制御
    //  W_V_DIRECTION : W→V に電流を流す様制御

    //戻り値
    // なし

    //P208(Q1U)
    //P407(Q1L)
    //P101(Q2U)
    //P100(Q2L)
    //P103(Q3U)
    //P102(Q3L)

    switch(direction)
    {
        case BLM_OFF_DIRECTION:
            //P208, P407, P101, P100, P103 P102 = L
            R_PORT2->PCNTR3 = 0x01000000;
            R_PORT4->PCNTR3 = 0x00800000;
            R_PORT1->PCNTR3 = 0x000f0000;
            break;

        case BLM_U_V_DIRECTION:
            //電流を U→V に流す設定, P208(Q1U)=H, P100(Q2L)=H （他は L）
            R_PORT4->PCNTR3 = 0x00800000;   //OFF
            R_PORT2->PCNTR3 = 0x00000100;   //ON を後に持ってくる（Q1U と Q1L が同時 ON になる事が無い様に）
            R_PORT1->PCNTR3 = 0x000e0001;
            break;

        case BLM_U_W_DIRECTION:
            //電流を U→W に流す設定, P208(Q1U)=H, P102(Q3L)=H
            R_PORT4->PCNTR3 = 0x00800000;   //OFF
            R_PORT2->PCNTR3 = 0x00000100;   //ON
            R_PORT1->PCNTR3 = 0x000b0004;
            break;

        case BLM_V_W_DIRECTION:
            //電流を V→W に流す設定, P101(Q2U)=H, P102(Q3L)=H
            R_PORT2->PCNTR3 = 0x01000000;   //OFF
            R_PORT4->PCNTR3 = 0x00800000;   //OFF
            R_PORT1->PCNTR3 = 0x00090006;
            break;

        case BLM_V_U_DIRECTION:
            //電流を V→U に流す設定, P101(Q2U)=H, P407(Q1L)=H
            R_PORT2->PCNTR3 = 0x01000000;   //OFF
            R_PORT4->PCNTR3 = 0x00000080;   //ON
            R_PORT1->PCNTR3 = 0x000d0002;
            break;
```

```
//モータドライブ電流方向定義
#define BLM_OFF_DIRECTION 0
#define BLM_U_V_DIRECTION 1
#define BLM_U_W_DIRECTION 2
#define BLM_V_W_DIRECTION 3
#define BLM_V_U_DIRECTION 4
#define BLM_W_U_DIRECTION 5
#define BLM_W_V_DIRECTION 6
```

・blm.c（続き）

```
        case BLM_W_U_DIRECTION:
            //電流を W→U に流す設定, P103(Q3U)=H, P407(Q1L)=H
            R_PORT2->PCNTR3 = 0x01000000;   //OFF
            R_PORT4->PCNTR3 = 0x00000080;   //ON
            R_PORT1->PCNTR3 = 0x00070008;
            break;

        case BLM_W_V_DIRECTION:
            //電流を W→V に流す設定, P103(Q3U)=H, P100(Q2L)=H
            R_PORT2->PCNTR3 = 0x01000000;
            R_PORT4->PCNTR3 = 0x00800000;
            R_PORT1->PCNTR3 = 0x00060009;
            break;

        default:
            break;
    }
}
```

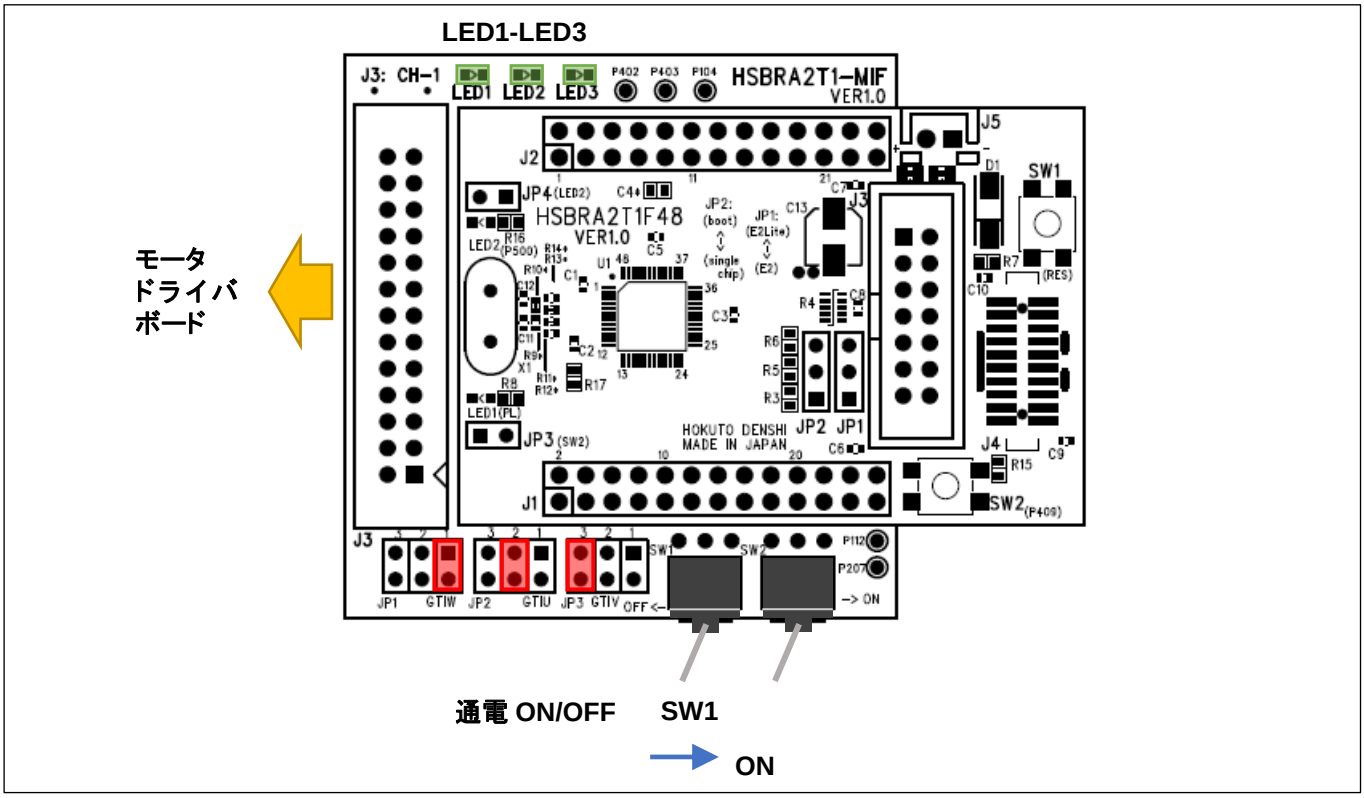
モータドライバボード側では、P208, P407, P101, P100, P103, P102 の 6 端子で電流を制御する方式です。P208 と P100 を H 制御すると、U→V の方向に電流を流す制御となるといった具合です。



P208 は Q1U につながっていて、U 相の H 側を制御しています。P100 は Q2L につながっていて、V 相の L 側を制御しています。残りも同様で、6 本の信号線が 6 個のモータ駆動 FET を 1 対 1 で制御する形となっています。P208 と P100 を H とすると、q1h, q2l の 2 つの FET が ON し、モータのコイルを経由して図の i の電流が流れます。U→V 以外の定義も同様に、3 相ある電極の 2 相間に電流を流す設定となります。

なお、SW1 を ON にすると、上図の QU=QL=H に制御され、6 本の信号（P208~P102）が有効になります。SW1 を OFF にすると、QU=QL=L に制御され、6 本の信号が無効化（P208~P102 の信号レベルに拘わらず 6 個のモータ駆動 FET が全て OFF 制御）となります。

本プログラムでは、モータの軸が一瞬振れる感覚がありますが、回転には程遠いと思います。モータを回転させる制御まで、あといくつかのステップがあります。ここでは、モータ内の任意のコイルの任意の方向に電流を流す事が、プログラムで行える事を理解してください。



上図で、J3, CH-1 のコネクタにモータドライバモード（その先にモータ）を接続します。このコネクタに接続されたモータを CH-1 と表記します。ブラシレスモータスタータキット(RA2T1)では、接続可能なモータは 1 つです。

TUTORIAL2 のプログラムを書き込んで起動すると、LED1-LED3 が 0.5 秒毎に点灯します

SW1 を ON 側に倒すと、モータが ON（本チュートリアルでは、モータに電流を流して、モータからわずかにカチカチ音がするだけですが）します。SW1 でモータの ON/OFF が切り替わります。SW2 は未使用です。

・チュートリアル 2 での端子設定

端子名	役割	割り当て	備考
P100~P103	Q2U~Q3L	出力（初期値 L）	モータ駆動端子
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P208	Q1U	出力（初期値 L）	モータ駆動端子
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P400	QU	出力（初期値 L）	
P401	QL	出力（初期値 L）	
P407	Q1L	出力（初期値 L）	モータ駆動端子
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

・チュートリアル 2 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	50us タイマ	
g_agt1	r_agt	AGT1	500ms タイマ	

※グレーの項目は前チュートリアルから変更なし

－モータ駆動関数の関数ポインタ化に関して－

モータ駆動関数は、
blm_drive_ch1()　[CH-1]
blm_drive_ch2()　[CH-2]　※本キットでは存在しない
という関数名で作成しています。ブラシレスモータスタータキット(RA2T1)は、1 モータ（を想定した作り）のキットなので、ch1 が付く関数のみ用意しています。

本キットのチュートリアルのプログラムでは、blm_drive_ch1()に対して、blm_drive[n]()という別名（関数ポインタ）を与えています。

関数の別名　　関数の実体
blm_drive[0]() → blm_drive_ch1()

上記の様に、関数に別名を設けている理由は、ループで処理を行うためです。

```
for (i=0; i<1; i++)
{
  blm_drive[i](current_direction);
}
```

blm_drive 以外の関数も、ループで処理したい関数は同様の構成を取っています。

関数の別名　=　CH 毎の関数名
blm_*xx*[0] = blm_*xx*_ch1　　（*xx* は drive や start, stop など）

という対応となっていて、プログラム内で呼び出す関数が

```
blm_xx_ch1();          //...(1)
```

の代わりに

```
blm_xx[BLM_CH1]();     //BLM_CH1=0    ...(2)
```

となっているだけで、(1)(2)のどちらでも動作は同じであると認識して頂きたく。

※モータ 1ch のみの本キットではループでの処理や関数ポインタ化は本来不要ですが、複数モータを取り扱うキットと共通化のために同じ構成としています。

例えばですが、処理関数に ch の引数を持たせる様に作成する手法（ループで処理するための関数の作り方の例）も考えられます。

```
void blm_drive(int ch)
{
    switch(ch)
    {
        case BLM_CH1:
          //blm_drive_ch1()相当の処理
          break;

         case BLM_CH2:
              //blm_drive_ch2()相当の処理
              break;
```

上記の様に、関数自体にチャネルの引数を持つ様に作るという形でも良いかと思いますが、本キットでは関数自体はチャネル独立で構成して、チャネル独立な関数を関数ポインタ（配列）でまとめる構成としてます。
（チャネルを引数として条件分岐させるより、関数ポインタ化した方がわずかながらオーバヘッドが小さくなるという意図です。それ程の違いはないとは思いますが。）

ーコラムー　PODR と PCNTR3 を使ったポート制御例に関して

モータ制御時は、モータ制御端子（6 端子）の ON/OFF を頻繁に切り替える必要があります。

例えば、U 相から V 相に電流を流す制御から、U 相から W 相に電流を流す制御に切り替える際、

	U から V		U から W
Q1U(P208)	H		H
Q1L(P407)	L		L
Q2U(P101)	L	→	L
Q2L(P100)	H		L
Q3U(P103)	L		L
Q3L(P102)	L		H

P208, P407, P100~P103 の端子を上記の様に変化させる必要があります。

この様なポート操作を行う方法をいくつか考えてみます。

(1)PODR_b を使った制御

```
    //U 相から V 相に電流を流す制御
    R_PORT2->PODR_b.PODR8 = 1;
    R_PORT4->PODR_b.PODR7 = 0;
    R_PORT1->PODR_b.PODR1 = 0;
    R_PORT1->PODR_b.PODR0 = 1;
    R_PORT1->PODR_b.PODR3 = 0;
    R_PORT1->PODR_b.PODR2 = 0;

    //↓

    //U 相から W 相に電流を流す制御
    R_PORT2->PODR_b.PODR8 = 1;
    R_PORT4->PODR_b.PODR7 = 0;
    R_PORT1->PODR_b.PODR1 = 0;
    R_PORT1->PODR_b.PODR0 = 0;
    R_PORT1->PODR_b.PODR3 = 0;
    R_PORT1->PODR_b.PODR2 = 1;
```

PODR は、Port Output Data Resistor で、ポートの L/H を制御するレジスタです。このレジスタは、1bit 単位でアクセスできるので、6 本の制御線を 6 行のプログラムコードで変更する手法です。単純ですが、冗長です。

(2)PODR を使った制御

```
    //U 相から V 相に電流を流す制御
    tmp = R_PORT1->PODR;
    tmp &= ~0x000e;           //b1,b2,b3 を 0 にする（P101, P102, P103 を L に設定）
    tmp |= 0x0001;            //b0 を 1 にする（P100 を H に設定）
    R_PORT1->PODR = tmp;

    tmp = R_PORT2->PODR;
    tmp |= 0x0100;            //b8 を 1 にする（P208 を H に設定）
    R_PORT2->PODR = tmp;

    tmp = R_PORT4->PODR;
    tmp &= ~0x0080;           //b7 を 0 にする（P407 を L に設定）
    R_PORT4->PODR = tmp;


    //U 相から W 相に電流を流す制御
    tmp = R_PORT1->PODR;
    tmp &= ~0x000b;           //b0,b1,b3 を 0 にする（P100, P101, P103 を L に設定）
    tmp |= 0x0004;            //b2 を 1 にする（P102 を H に設定）
    R_PORT1->PODR = tmp;

    tmp = R_PORT2->PODR;
    tmp |= 0x0100;            //b8 を 1 にする（P208 を H に設定）
    R_PORT2->PODR = tmp;

    tmp = R_PORT4->PODR;
    tmp &= ~0x0080;           //b7 を 0 にする（P407 を L に設定）
    R_PORT4->PODR = tmp;
```

PODR レジスタ(16bit)を読み込んで、ビット演算で必要なビットのみ操作を行った後で、PODR レジスタに 16bit 単位で書き戻すという処理です。リード・モディファイ・ライトという手法で、I/O ポートの操作において非常に良く用いられる手法です。

(3)PCNTR3 を使った制御

```
    //U 相から V 相に電流を流す制御
    R_PORT1->PCNTR3.PORR = 0x000e;  //b1,b2,b3 に相当する P101, P102, P103 を L にする
    R_PORT1->PCNTR3.POSR = 0x0001;  //b0 に相当する P100 を H にする

    R_PORT2->PCNTR3.POSR = 0x0100;  //b8 に相当する P208 を H にする

    R_PORT4->PCNTR3.PORR = 0x0080;  //b7 に相当する P407 を L にする


    //U 相から W 相に電流を流す制御
    R_PORT1->PCNTR3.PORR = 0x000b;  //b0,b1,b3 に相当する P100, P101, P103 を L にする
    R_PORT1->PCNTR3.POSR = 0x0004;  //b2 に相当する P102 を H にする

    R_PORT2->PCNTR3.POSR = 0x0100;  //b8 に相当する P208 を H にする

    R_PORT4->PCNTR3.PORR = 0x0080;  //b7 に相当する P407 を L にする
```

PCNTR3 は、POSR（ポートのセット）, PORR（ポートのリセット）という 2 つのレジスタで構成されています。POSR に 1 を書き込むと、ポートの出力が H になる、PORR に 1 を書き込むとポートの出力が L になるというレジスタです。

PCNTR3(PORR, POSR)を使った場合は、変更したいところを 1 にするというレジスタアクセスとなるので、上記の様に記載できます。さらに、PORR, POSR は PCNTR3 として、32bit アクセスも可能なので、下記の記載でも等価です。(変更したくないポートのビットは 0 にする。)

(4)PCNTR3 を使った制御(32bit 単位でのアクセス)

```
    //U 相から V 相に電流を流す制御
    R_PORT1->PCNTR3 = 0x000e0001;   //P101=P102=P103=L, P100=H
    R_PORT2->PCNTR3 = 0x00000100;   //P208=H
    R_PORT4->PCNTR3 = 0x00800000;   //P407=L

    //U 相から W 相に電流を流す制御
    R_PORT1->PCNTR3 = 0x000b0004;   //P100=P101=P103=L, P102=H
    R_PORT2->PCNTR3 = 0x00000100;   //P208=H
    R_PORT4->PCNTR3 = 0x00800000;   //P407=L
```

上位 16bit に PORR, 下位 16bit に POSR を設定すると、1 行でポートの出力を「0 に変更(L 出力)」「1 に変更(H 出力)」「変更しない」の処理が行えます。

PCNTR3 はポートの出力変更時に便利なレジスタですので、本チュートリアルでも積極的に使用しています。

※(1)の 6 行のコードが最終的に 3 行となっており、この例ではあまりすっきりした印象はないですが、使用しているポートが P1xx 等に固まっている場合、PCNTR3 を使うと 1 行で記載できるのですっきりとした感じになります
(RA2T1 は、ポート構成の都合上、P1xx 等の 1 系統のポートにモータ駆動端子をまとめられなかったので、それ程 PCNTR3 を使用するメリットが感じられません。)

1.3. A/D 変換と PWM を試す

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL3

このチュートリアルでは、マイコンの A/D 変換（Analog to Digital 変換）機能と、PWM（Pulse Width Modulation: パルス幅変調）を試してみます。モータを回す制御から一旦離れますが、どちらもモータを動かすのに必要な機能となります。

本プログラムでは、

・VR の回転角に応じたパルス幅の信号が、QL P401 から出力
・モータドライバボード上の温度センサ（サーミスタ, R54）の値を拾う

という動作を行います。本プログラムでは、情報をシリアル通信(UART)で出力します。USB-ADAPTER-RX14（別売オプション）を、J3 に挿す、または、市販の USB-Serial モジュールを J3 に接続してください（接続方法は、1.1 のプログラムの書き込みの箇所を参照してください）（シリアル通信のモニタは必須ではありませんが、接続した場合、プログラムの動作が判り易くなると思います）。

・起動時にシリアル端末から出力される情報

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL3


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)

>
Motor driver board connection check...
  CH-1 Connected.
```

PC 上では、teraterm 等のシリアル端末ソフトで表示してください

115,200bps, 8bit, none, 1bit の設定で表示できます

Motor driver board connection check... : CH-1 **NOT** Connected. になっている場合は、モータドライバボード側の接続を確認してください。（モータのホールセンサケーブルをモータドライバボードに接続していない場合などに NOT Connected.となります）（NOT Connected となった場合は動作が ON になりません。）

Active は、SW が OFF の時は、x になります。

ここで、SW1 を ON にします。

```
                        CH-1
Motor Driver Board     :  Connect
Active                 :     x
Temperature(A/D value) :  2101
Temperature(degree)    :    27
VR(A/D value)          :   994
QL duty[%]             :   0.0

  CH-1 START

                        CH-1
Motor Driver Board     :  Connect
Active                 :     o
Temperature(A/D value) :  2104
Temperature(degree)    :    27
VR(A/D value)          :   994
QL duty[%]             :  24.2
```

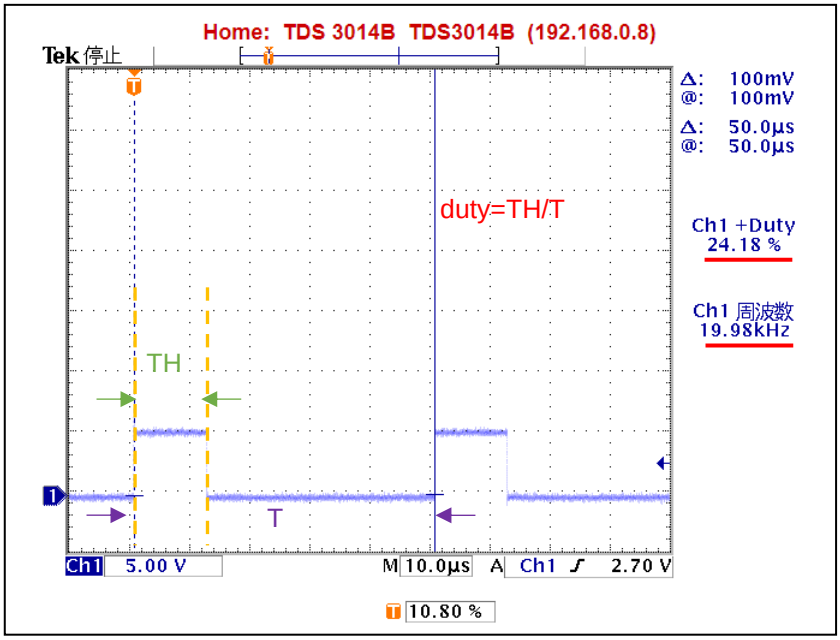
枠内の情報が
3 秒毎に表示されます

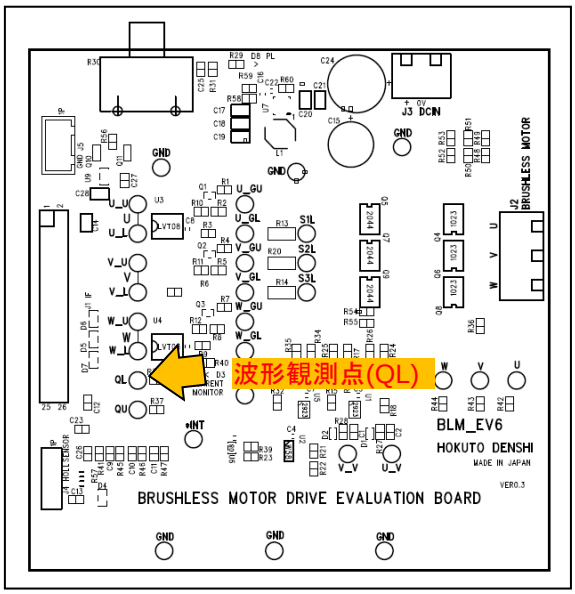
SW を ON にすると、QL duty が
0 からモータドライバボード上の
VR に応じた数値に変わります

→実際に PWM 波形が出力
されます

上記では、温度センサの A/D 変換値は 2104 で温度に変換すると、27℃。モータドライバボードの、VR の A/D 変換値は 994 で、duty は 24.2%に設定されているという情報が出力されています。

・オシロスコープで QL 端子（モータドライバボード QL 端子）を観測した波形





duty は、VR の回転角度に応じて、0~90%程度の範囲で変化します。QL 波形の周波数は、20kHz です。

※モータドライバボードが接続されていないと認識された場合、Active にはなりません（QL から波形が出力されません）

Temparature(A/D value)は、温度センサーの出力です。温度センサーの出力は、信号名では AD6、マイコンの P010/AN005 に接続されています。マイコンの当該端子を A/D 変換入力に設定し、A/D 変換機能を使って温度値を取得します。RA2T1 の A/D 変換機能は、12bit で 0~4095 までの値を取ります。(この値は、約 25℃のときに、2048 になります。温度が高いほど数値は大きくなります)

Temparature(degree)は、温度センサーの A/D 変換値を摂氏温度に変換したものです。温度の変換は、モータドライバボードの取扱説明書に計算式を示してあります。(計算式は、exp の計算を含む手間のかかるものとなっているので、本プログラムでは予め A/D 変換値と温度の 80 点のテーブルを作成しておき、テーブルから温度を換算しています。)ここでは、27℃となっていますが、ドライヤー等でモータドライバボードの温度センサ部(R54:黒いヒートシンクの下側付近にある)を暖めると、画面表示上の温度も上昇するかと思います。

VR(A/D value)は、モータドライバボード上の VR(ボリューム)を回すと、値が変わるはずです。時計回りに目一杯回すと 0。反時計回りに目一杯回すと、3724(4096×10/11)近傍になるはずです。VR は、プログラム上で値を拾いアナログ入力デバイスとして、任意の用途で使用する事が出来ます。

QL duty は、QL 端子の duty 値となります。この値は、VR の読み取り値に連動して変更を行っています。本プログラムでは(VR の読み取り値/4096×100=)0~90%程度まで設定可能です。この値と、実際に QL 端子から出力されるパルス波形は連動しています。

ここでは、VR の読み取り値をプログラムで処理して、QL 端子の duty 比を決めている事に注意願います。

QL 端子は、

CH-1
P401/GTIOC3B

上記の端子に接続されており、GPT タイマの出力が接続されています。

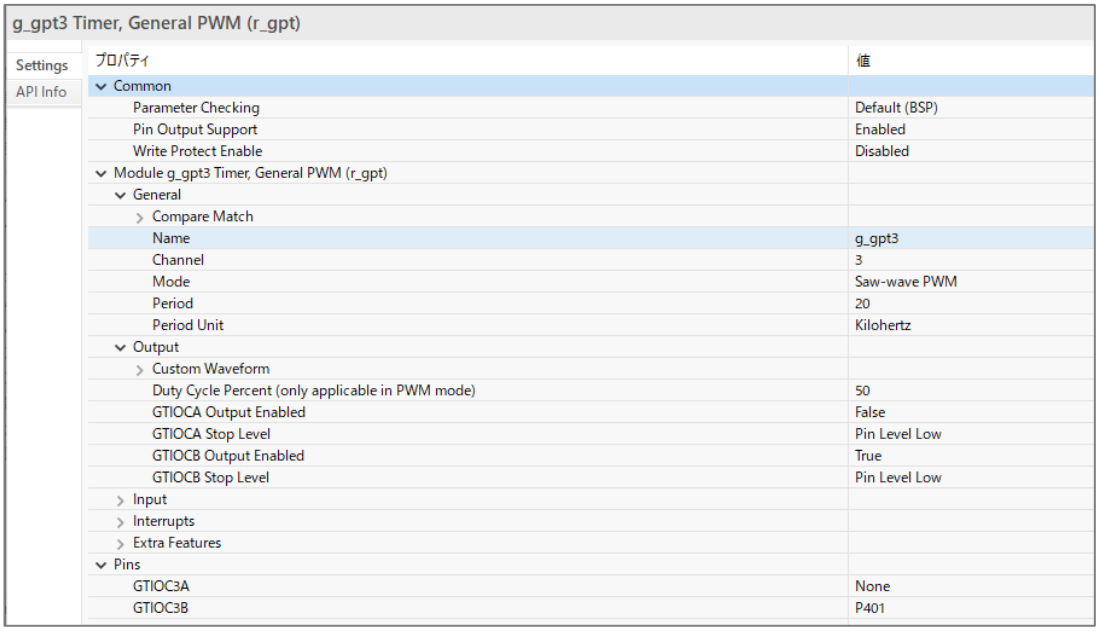
QL 端子に接続されているタイマのコンペアマッチレジスタの設定値を変えると、QL 端子に出力される、H パルスの幅が変わります。この様に、パルス幅を変える制御を、PWM(パルス幅変調)といい、モータの制御ではよく使用される方式です。

PWM 制御には、GPT タイマを使用しています。

GPT タイマは、AGT タイマ同様、FSP の stack として追加します。

New Stack > Timers > Timer, General PWM(r_gpt)

・プロパティ



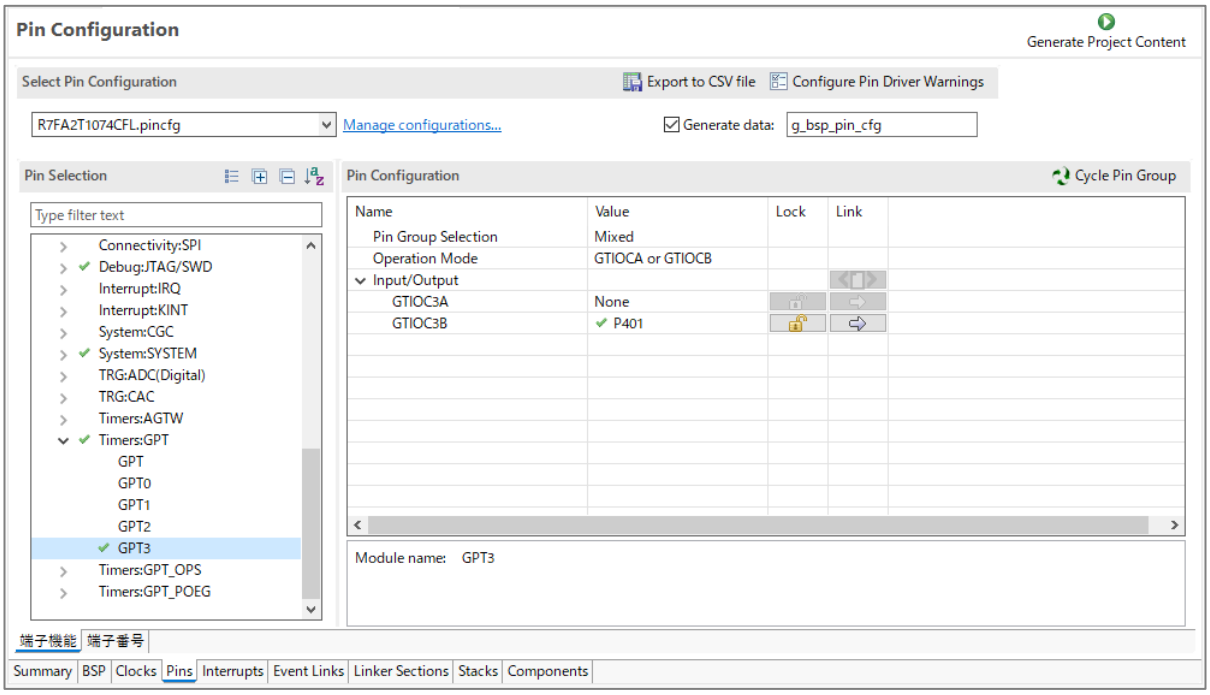
		設定値	備考
Common	Pin Output Support	Enabled	タイマで端子を駆動する場合設定必要
General	Name	g_gpt3	使用タイマ GPT3 と名称を合わせる
	Channel	3	GPT3 を使用
	Mode	Saw-wave PWM	PWM モード選択
	Period	20	タイマ周期の設定
	Period Unit	Kilohertz	単位[kHz]
Output	Duty Cycle Percent	50	プログラムで随時変更するのでここでは仮値
	GTIOCB Output Enabled	True	GTIOC3B(P401)の出力を有効化
	GTIOCB Stop Level	Pin Level Low	正極性を選択
Pins	GTIOC3A	None	
	GTIOC3B	P401	GTIOC3B の端子は複数から選択可能(*1)

Name は、タイマ追加順に、g_timer0, g_timer1, ...と割り振られるので、ここでは g_gpt3 に変更しています。（Name は任意の名称を付けられます。）

ここでは、PWM の周波数は、20kHz に設定しています。

(*1)Pins タブから端子を選んで設定

・端子設定（Pins タブ）

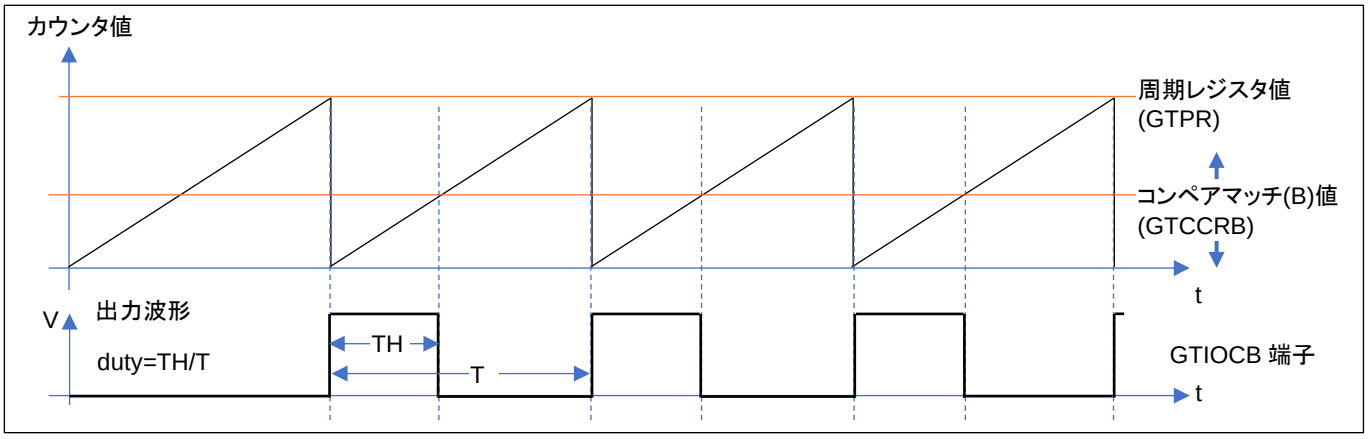


端子は、GPT3 の

Operatin Mode **GTIOCA or GTIOCB** を選ぶ

GTIOC3A None のまま

GTIOC3B **P401** を選ぶ （P401 は、CH-1 の QL 端子）



出力波形の duty を変更する場合は、コンペアマッチ値を変更します。この例（アップカウントのノコギリ波，コンペアマッチで L）の場合は、コンペアマッチ値を小さくすると出力される duty は小さくなります。

周期レジスタ値で、周期（PWM 周波数）が決まります。（TUTRIAL3 のプログラムでは、周期 50us, PWM 周波数 20kHz の設定で、固定です。）

（GTIOCnA 端子は GTCCRA レジスタ、GTIOCnB 端子は GTCCRB レジスタで duty が決まります。）

コンペアマッチレジスタ(GTCCRB)の値は、バッファリングされる設定となっており、GTCCRE レジスタに値を設定すると、適切なタイミングで GRCCRB レジスタに反映される様になります。(本チュートリアルでは使用していませんが、GTIOCnA 端子のタイミングを決める GTCCRA レジスタとバッファレジスタ GTCCRC の関係も同様です。)

PWM の duty を変更する際に、バッファレジスタである GTCCRE レジスタに値を設定しても良いのですが、FSP の環境では duty を変更する API 関数が用意されています。

```
void blm_dutyset_ch1(unsigned short duty)
{
//PWMduty 設定関数 CH-1

//引数：
// duty ： 設定する duty 比(0-255) (256->100%)

//戻り値：なし

    uint32_t duty_cycle_counts;

    duty_cycle_counts = (uint32_t)((g_gpt3_period_counts * (uint32_t)duty) >> BLM_DUTY_MULTIPLIER_SHIFT);

    //CH-1 の QL は GTIOC3B
    (void) R_GPT_DutyCycleSet(&g_gpt3_ctrl, duty_cycle_counts, GPT_IO_PIN_GTIOCB);
}
```

```
#define BLM_DUTY_MULTIPLIER_SHIFT   8
//duty は 0-1(100%)を 0-256 で取り扱う，8 ビットシフト
```

周期レジスタ値

R_GPT_DutyCycleSet()が API 関数で、この関数に計算したカウント値を与える事で、端子の出力 duty を変更する事が可能です。

API 関数に関しては、FSP のヘルプに記載があります。e2studio のインストールフォルダにも html 形式でヘルプファイルがインストールされていますので、参照ください。

なお、単純にコンペアマッチレジスタ値を設定した場合、

	期待値	実際の動作
duty を 0%に設定	端子は L 固定	タイマの 1 サイクル H が出力
duty を 100%に設定	端子は H 固定	端子は H 固定

(タイマがアップカウントで、周期終わりで H、コンペアマッチで L となる設定の場合)
上記の様な動作になります。(0 や 100%が期待値と異なるケースが出てきます)

(ダウンカウントではまた異なる結果。コンペアマッチでトグルとなる設定の場合、100%を超えるコンペアマッチ値を指定すると、現在の端子レベルが固定(L か H のどちらかで固まる)といった動作となります。コンペアマッチレジスタの値を設定する場合、カウント方向や、どのタイミングで波形が切り替わる設定なのかを意識する必要があります。)

API 関数を使った場合は、API 関数が 0 や 100%(＝この場合は引数に周期値を指定した場合)を、特別な値として取り扱うために、タイマの動作を深く考えなくても、期待値通りの出力が得られます。

これは、API 関数を使用するメリットかと思います。(但し、その分コンペアマッチレジスタを直接操作するより、処理のオーバヘッドがある。)

とにかく実行速度を稼ぎたい場合は、レジスタを直接書き換えた方が有利に働くケースもあると思われますが、まずは API 関数を使用するというスタンスで良いのではないかと思います。

・本チュートリアルでの duty 値の取り扱い

RX や他の RA のブラシレスモータスタータキットでは、duty は float 型で 0-100%を 0-1 で取り扱っています。RA2T1 マイコンは、FPU を持たないため float 型の計算は処理が重たくなるので、duty 値を unsigned short 型として、0-100%を 0-256 の数値で取り扱っています。

・blm_main.c

```
static void blm_duty_change(void)
{
    //duty 変更関数

    //引数
    // なし

    //戻り値
    // なし

    /*
     * VR（モータドライバボード上のボリューム）の回転角度を PWM の duty 比に反映させる
     */

    unsigned short i;

    const int bit_shift = BLM_ADC_BIT - BLM_DUTY_MULTIPLIER_SHIFT;
    //12bitADC, duty を 256 倍で取り扱う(8bit シフト) 12 - 8 = 4, 最終的に 4bit シフトさせる

    for (i=0; i<BLM_CH_NUM; i++)
    {
        if (g_state[i] == BLM_CH_STATE_ACTIVE)
        {
            g_duty[i] = (unsigned short)(g_adc_result[i].volume >> bit_shift);
            //単純に 4 ビットシフト（BLM_DUTY_MAX=100[%]の時は計算を単純化）
            blm_dutyset[i](g_duty[i]);
        }
    }
}
```

```
#define BLM_ADC_BIT                 12
#define BLM_DUTY_MULTIPLIER_SHIFT    8
```

duty を 0-1 で取り扱う場合は 12bit シフトだが ここでは 4bit シフトに収めている （8bit シフトの積み残しがある）

g_duty は 0-255 の値を取る

```
void blm_dutyset_ch1(unsigned short duty)
{
（中略）

    duty_cycle_counts = (uint32_t)((g_gpt3_period_counts * (uint32_t)duty) >> BLM_DUTY_MULTIPLIER_SHIFT);

（後略）
```

ここで 8bit シフトの積み残しを解消

VR の A/D 変換結果は 0-4095 になるので、単純にこの数値を 4bit 右シフトさせて、0-255 に変換した値を duty 値としています。

浮動小数点数演算ユニット(FPU)を持たないマイコンで、0~1(0~100%)の数値を取り扱う際は、何らかの定数を掛けた数値として取り扱うという方法が考えられます。単純に 100 を掛けて、0~1 を 0~100 で取り扱う等です。

定数値を 100 とすると、元に戻す際に÷100 の演算が必要ですが、定数値を 2^n しておくと、元に戻す際にビットシフト演算で対応できます。

分解能（表現可能な細かさ）は、定数値が大きいほど小さくなりますが、本チュートリアルでは 2^8(=256)としています。duty 値を取り扱う細かさとしては、1/256=0.39e^{-3} となるので、0.4%程度の分解能となります。

（数値演算において、2^n の定数を掛けて計算する手法は、相補 PWM の計算でも行っています。浮動小数点数を整数に置き換えて取り扱う代表的な手法かと思います。）

※なお、RA2T1 マイコンにおいても、float 型の変数や計算は使用可能です。但し、処理が重くなるので極力使用を避けています。

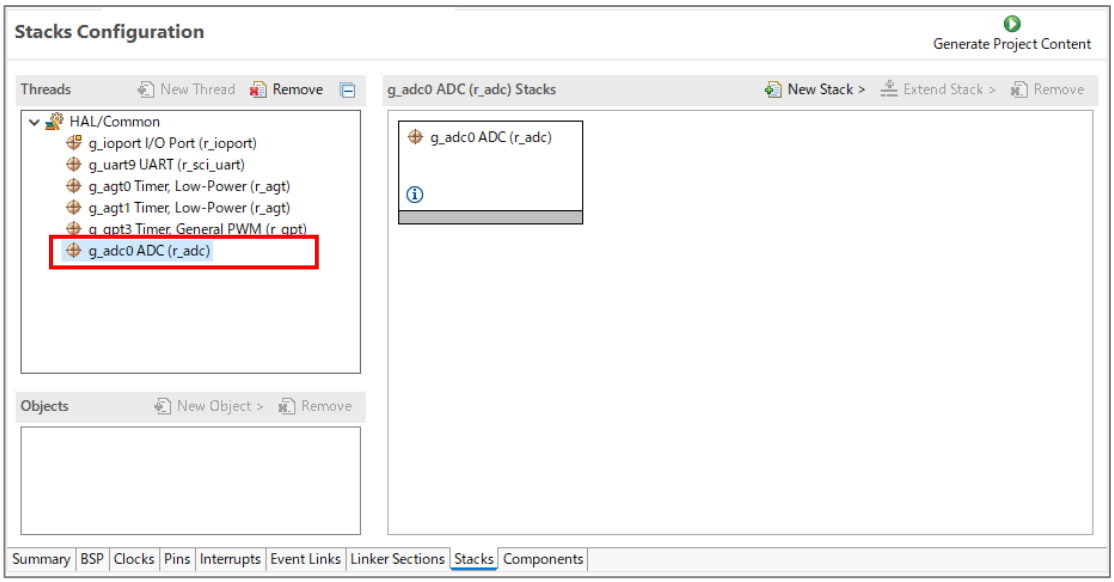
次に、本チュートリアルで使用している A/D 変換の部分に関して説明します。

接続信号名	内容	使用端子
AD0	U 相電圧	P013/AN000
AD1	V 相電圧	P014/AN001
AD2	W 相電圧	P015/AN002
AD3	U 相電流	P000/AN008
AD4	V 相電流	P001/AN009
AD5	W 相電流	P002/AN010
AD6	温度センサ	P010/AN005
AD003	電源電圧	P011/AN006
VR	ボリューム	P012/AN007

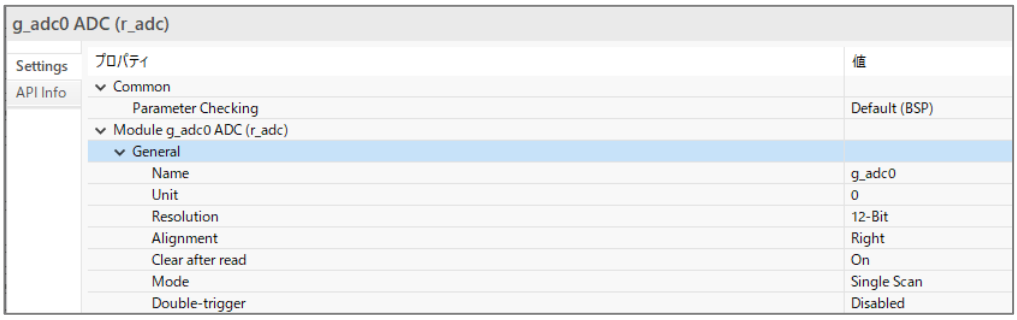
FSP では、

New stack > Analog > ADC(r_adc)

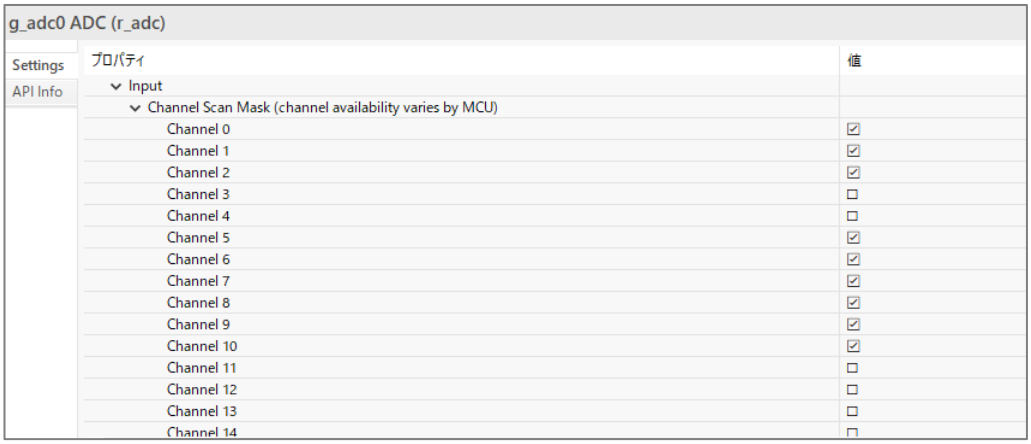
を追加します。



・プロパティ

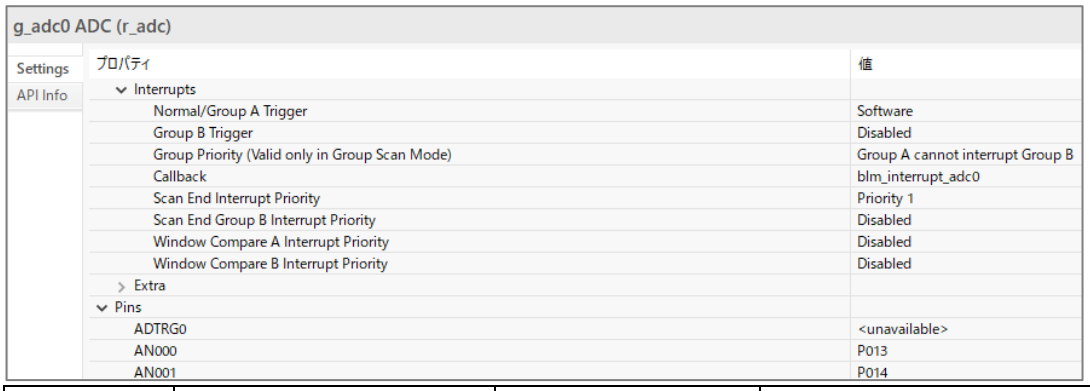


General　以下はデフォルトから変更点なし



Input		設定値	備考
Channel Scan Mask	Channel 0	チェック	AN000 に対応
	Channel 1	チェック	AN001 に対応
	Channel 2	チェック	AN002 に対応
	Channel 5	チェック	AN005 に対応
	Channel 6	チェック	AN006 に対応
	Channel 7	チェック	AN007 に対応
	Channel 8	チェック	AN008 に対応
	Channel 9	チェック	AN009 に対応
	Channel 10	チェック	AN010 に対応

Input の設定で、A/D 変換対象としたい端子(Channel)を選択します。



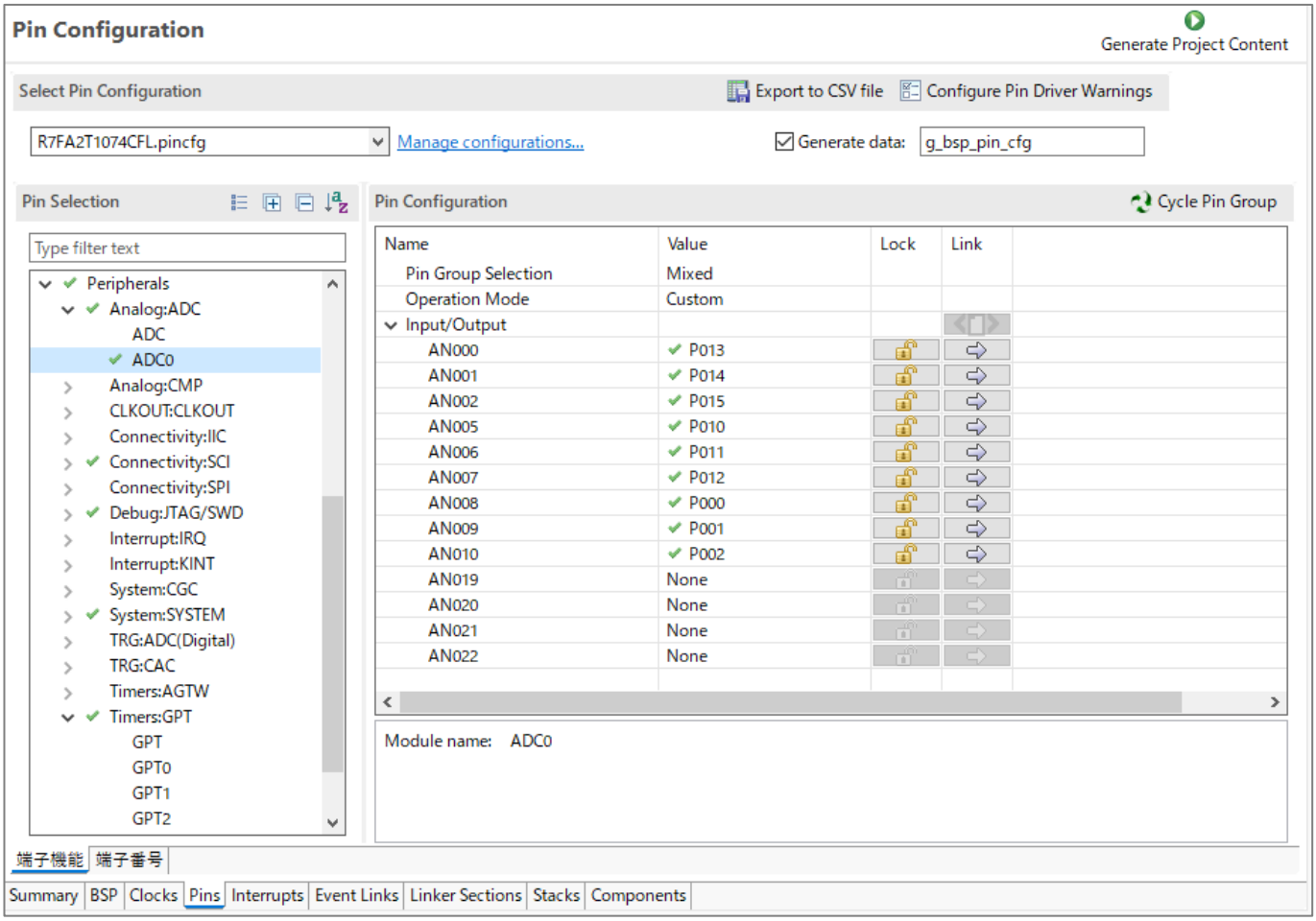
		設定値	備考
Interrupts	Callback	blm_interrupt_adc0	A/D 変換終了時に呼ばれる割り込み関数名を定義
	Scan End Interrupt Priority	Priority 1	A/D 変換終了割り込み

※デフォルトから変更点のみ記載

Scan End Interrupt Priority は、デフォルト Disabled で、その場合は A/D 変換終了時に割り込みは掛かりません。0~3 の Priority を設定する事で、A/D 変換が終了したタイミングで割り込みが掛ります。(ここでは、Priority 1 で設定していますが、Priority=割り込み優先度の設定値は任意です。数字の小さい方が優先度が高い割り込みとなります。)

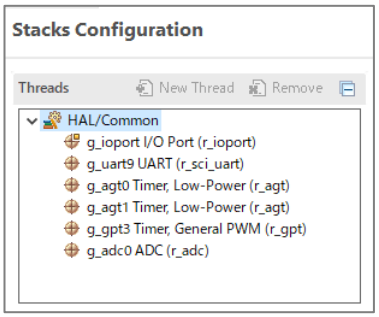
ADC の設定は、設定項目が多く様々なパラメータの設定が可能です。
(後のチュートリアルでは、A/D 変換値の平均化の設定なども行っています。)

次に、Pins タブの ADC の設定で、使用する端子を選んでおきます。



A/D 変換端子は、AN000→P013 に割り当て(他の端子には割り当てできない)と決まっているので、他の機能(GPT タイマの出力端子や、UART の通信端子)の様に、複数の端子から選ぶ訳ではなく、AN000 を有効化するかどうかの設定です。
(※ここで、AN000 を有効化しなくても、A/D 変換自体は実行されます。あえて設定しなくても、問題はないかも知れません。但し、AN000→P013 の設定を行っておくと、P013 を他の機能に割り当てようとした際にエラーが出るので、どの端子をどの機能で使用するかを明確にするために、設定を行っておくのが推奨です。)

本チュートリアルで使用してる stack（機能）は、以下となります。



一通り stack を追加したのが上記となります。

r_sci_uart(SCI9)

r_agt(AGT0)

r_agt(AGT1)

r_gpt(GPT3)

を使用しています。

AGT は、定期的な処理

・AGT0　50us タイマ　A/D 変換の起動とスイッチのチャタリング除去

・AGT1　10ms タイマ　VR→duty の更新, UART の表示更新タイミング

の用途で使用しています。これ以降のチュートリアルでも、AGT0, AGT1 は同様の使い方をしています。

AGT タイマと A/D 変換の割り込みコールバック関数は

機能	コールバック関数
AGT0	blm_interrupt_agt0()
AGT1	blm_interrupt_agt1()
ADC0	blm_interrupt_adc0()

としています、割り込み処理を記載した関数（blm_interrupt_xxx()関数）本体は、blm_intr.c 内にまとめています（UART の割り込みを除く）。

・blm_intr.c 内 AGT タイマ割り込み関数

```
void blm_interrupt_agt0(timer_callback_args_t *p_args)
{
    //100us 割り込み

    if (TIMER_EVENT_CYCLE_END == p_args->event)
    {
        //カウンタ変数
        g_agt0_counter++;   100us×n 回を観測するためのカウンタ変数

        //A/D 変換をキック
        if (g_adc_scan_flag == 0)   //前回の A/D 変換が完了している場合
        {
            g_adc_scan_flag = BLM_ADC_FLAG_0;
            (void) R_ADC_ScanStart(&g_adc0_ctrl);
        }
    }
}

void blm_interrupt_agt1(timer_callback_args_t *p_args)
{
    //10ms 割り込み

    if (TIMER_EVENT_CYCLE_END == p_args->event)
    {
        //カウンタ変数
        g_agt1_counter++;
        g_agt1_counter_2++;
    }
}
```

前回の A/D 変換が終わっていたら
A/D 変換開始指示

10ms×n 回を観測するためのカウンタ変数

AGT タイマの割り込みでは、

・カウンタ変数のインクリメント（メイン関数内でカウンタ変数を使用）

・100us の割り込みでは A/D 変換の開始

を行っています。

A/D 変換が終了すると、A/D 変換終了の割り込みが入ります。（入るように設定しています）

前回の A/D 変換が完了している場合（基本は完了しているはずです）、100us 毎に A/D 変換を起動します。

A/D 変換完了時の処理は、以下の様になっています。

・blm_intr.c 内 ADC 割り込み関数

```
void blm_interrupt_adc0(adc_callback_args_t *p_args)
{
    //A/D 変換割り込み

    //AN000 CH-1 AD0 U 相電圧
    //AN001 CH-1 AD1 V 相電圧
    //AN002 CH-1 AD2 W 相電圧
    //AN005 CH-1 AD6 温度センサ
    //AN006 CH-1 AD003 電源電圧
    //AN007 CH-1 VR ボリューム
    //AN008 CH-1 AD3 U 相電流
    //AN009 CH-1 AD4 V 相電流
    //AN010 CH-1 AD5 W 相電流

    //A/D 変換結果をグローバル変数に格納

    if (p_args->event == ADC_EVENT_SCAN_COMPLETE)
    {
        g_adc_result[BLM_CH_1].v_u_phase = R_ADC0->ADDR[0];
        g_adc_result[BLM_CH_1].v_v_phase = R_ADC0->ADDR[1];
        g_adc_result[BLM_CH_1].v_w_phase = R_ADC0->ADDR[2];
        g_adc_result[BLM_CH_1].temp = R_ADC0->ADDR[5];
        g_adc_result[BLM_CH_1].v_power = R_ADC0->ADDR[6];
        g_adc_result[BLM_CH_1].volume = R_ADC0->ADDR[7];
        g_adc_result[BLM_CH_1].i_u_phase = R_ADC0->ADDR[8];
        g_adc_result[BLM_CH_1].i_v_phase = R_ADC0->ADDR[9];
        g_adc_result[BLM_CH_1].i_w_phase = R_ADC0->ADDR[10];

        g_adc_scan_flag &= (unsigned short)~BLM_ADC_FLAG_0;     //フラグを落とす
    }
}
```

A/D 変換完了割り込み

AD 変換結果を
グローバル変数にコピー

フラグ変数のビットを落とす

A/D 変換結果レジスタ値を、グローバル変数に代入する処理となっています。

g_adc_result[ ].i_u_phase は、U 相の電流値を保存する変数です。ADDR[0]は、AN000 端子の A/D 変換結果が保存されているレジスタです。相電圧、相電流、電源電圧、VR（ボリューム），温度センサの値を g_adc_result（A/D 変換結果を格納する構造体）にコピーする処理を行っています。

A/D 変換割り込みは、ADC0(ADC-ch0, AN0xx 端子の A/D 変換）の A/D 変換終了のタイミング blm_interrupt_adc0()が呼び出されます。blm_interrupt_adc0()が呼ばれると、A/D 変換対象に設定した全ての端子（9 端子）の A/D 変換が終了した事となります。

AGT タイマの起動は、blm_init()関数で行っています。

・blm.c 内初期化関数(blm_init())

```
void blm_init(void)
{
    //ブラシレスモータ初期化関数

    //引数
    // なし

    //戻り値
    // なし

    //変数初期化
    g_agt0_counter = 0;
    g_agt1_counter = 0;
    g_agt1_counter_2 = 0;

    g_adc_scan_flag = 0;

    //タイマ
    (void) R_AGT_Open(&g_agt0_ctrl, &g_agt0_cfg);   //AGT0 初期化 100us タイマ
    (void) R_AGT_Open(&g_agt1_ctrl, &g_agt1_cfg);   //AGT1 初期化 10ms タイマ
    (void) R_GPT_Open(&g_gpt3_ctrl, &g_gpt3_cfg);   //GPT3 初期化 CH-1 QL PWM

 （中略）

//AGT タイマスタート
    (void) R_AGT_Start(&g_agt0_ctrl);
    (void) R_AGT_Start(&g_agt1_ctrl);        AGT タイマのスタート
```

blm_main()関数が、モータ制御の処理を行っているプログラムのスタート地点です。

・blm_main.c 内 blm_main()

```
void blm_main(void)
{
//ブラシレスモータメイン関数

    const unsigned long sw_read_interval = (unsigned long)(500e-6 / BLM_CONTROL_PERIOD);
//500us 毎にスイッチの状態をスキャン（100us:AGT0 で何カウントか）
    const unsigned long duty_change_interval = (unsigned long)(0.1 / BLM_COUNTER_PERIOD);
//0.1[s]毎（10ms:AGT1 で何カウントか）
    const unsigned long information_display_interval = (unsigned long)(3.0 / BLM_COUNTER_PERIOD);
//3 秒毎に画面に情報を表示（10ms:AGT1 で何カウントか）

    unsigned short prev_state[BLM_CH_NUM] = { BLM_CH_STATE_INACTIVE };

    unsigned short i;

    sci_start();  SCI(UART)の初期化

    sci_write_str("¥nCopyright (C) 2025 HokutoDenshi. All Rights Reserved.¥n¥n");
    sci_write_str("RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL3¥n");

 （中略）

blm_init();//初期化
```

sw_read_interval, duty_change_interval, information_display_interval は、それぞれスイッチの読み取りと duty の変更頻度と UART での画面表示頻度を決めている変数です。それぞれ、AGT0(100us), AGT1(10ms), AGT1(10ms)の何カウント毎に処理を行うかを決めています。

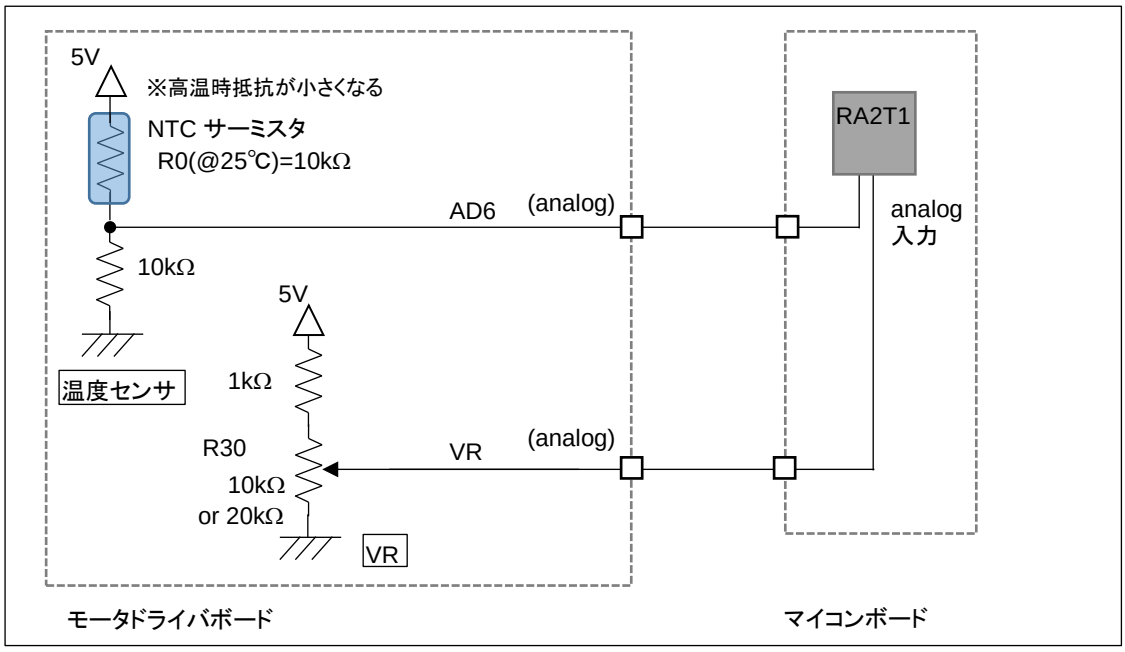
変数名	使用タイマ	呼び出し頻度	用途
sw_read_interval	AGT0(100us)×5	500us	トグルスイッチの読み出し
duty_change_interval	AGT1(10ms)×10	0.1s	VR の読み取り→duty 値の変更
information_display_interval	AGT1(10ms)×300	3s	画面表示

・プログラムのメインループ

```
//メインループスタート
while(1)
{
    //スイッチの読み取り(500us 毎)→出力 ON/OFF
    if (g_agt0_counter >= sw_read_interval)
    {
        //チャタリング防止
        //スイッチは、100us 毎×sw_read_interval(5)=500us 毎に読み取りを行う

        blm_sw_to_state();          SW を読み取りグローバル変数に代入

        //状態が変化した際スタート・ストップ
        for (i=0; i<BLM_CH_NUM; i++)
        {
            if (g_state[i] != prev_state[i])
            {
                if (g_state[i] == BLM_CH_STATE_ACTIVE)     SW の状態が変化した際スタート・ストップの処理
                {
                    blm_start[i]();
                    sci_write_str("¥n  CH-");
                    sci_write_uint16(i+1);
                    sci_write_str(" START¥n");
                }
                else
                {
                    blm_stop[i]();
                    g_duty[i] = 0;      //duty 設定変数は 0 にする
                    sci_write_str("¥n  CH-");
                    sci_write_uint16(i+1);
                    sci_write_str(" STOP¥n");
                }
                prev_state[i] =  g_state[i];//現在の状態を保存
            }
        }
        g_agt0_counter = 0;//カウンタ初期化
    }

    //VR を duty に反映（0.1 秒毎）
    if (g_agt1_counter >= duty_change_interval)
    {
        blm_duty_change();                          0.1 秒に 1 回 VR の読み取り値を duty に反映させる

        //コマンド入力
        blm_command_input();

        g_agt1_counter = 0;
    }

    //画面表示（3 秒に 1 回）
    if (g_agt1_counter_2 >=  information_display_interval)
    {
        if (information_display_flag == TRUE) blm_information_display();

        g_agt1_counter_2 = 0;                       3 秒に 1 回画面表示を行う
    }
}
```

RA2T1 は 1ch しか取り扱わないので、本来 ch 数のループ処理は不要であるが、他のキットに合わせてループを削除する事はしていません(BLM_CH_NUM=1)

本チュートリアルでは、温度センサと、VR（ボリューム）の読み取りを行っています。温度センサと VR の接続は、以下の様になっています。



温度センサは、25℃の時サーミスタは 10kΩとなりますので、AD6 端子の電位は 2.5V となります。高温時は電圧が上がる方向に変化します。VR は、軸を（軸方向から見て）反時計回りに回した際出力電位が上がり、最大 4.5V 程度（A/D 変換値で 3720 程度）、最小 0V（（A/D 変換値で 0）となります。

本チュートリアルでは、モータを駆動するという観点からは一旦離れましたが、PWM 波形を生成する、A/D 変換を行うというモータ制御においては重要なマイコン周辺機能を使うチュートリアルとなります。

次のチュートリアルでは、実際にモータを動かしてみます。

・チュートリアル 3 での端子設定

端子名	役割	割り当て	備考
P000~P015	AD0~AD6,VR AD003	アナログ入力(ADC)	
P100~P103	Q2U~Q3L	出力（初期値 L）	モータ駆動端子
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P206	HS2(*1)	入力（プルアップ）	モータドライバボード接続確認に使用
P208	Q1U	出力（初期値 L）	モータ駆動端子
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P400	QU	出力（初期値 L）	
P401	QL	周辺機能(GPT3)	GTIOC3B として設定
P407	Q1L	出力（初期値 L）	モータ駆動端子
P408, P409	HS1, HS3(*1)	入力（プルアップ）	モータドライバボード接続確認に使用
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

(*1)接続ボード上のジャンパ設定で HS1~HS3 の接続先は変更可能（ジャンパのデフォルト設定での接続先を記載）

・チュートリアル 3 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_gpt3	r_gpt	GPT3	QL PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	

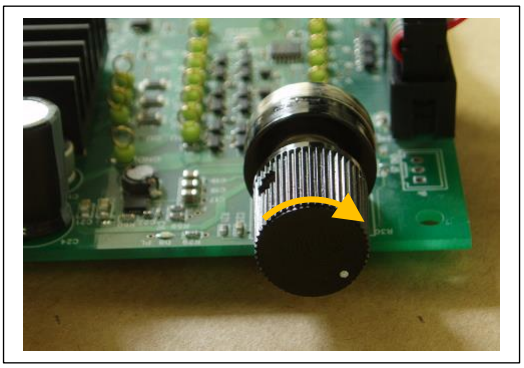
※グレーの項目は前チュートリアルから変更なし

1.4. モータを回してみる

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL4

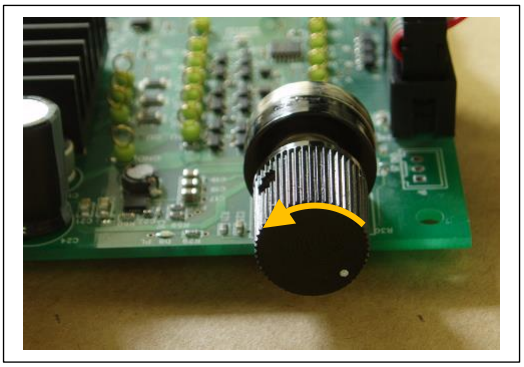
プログラムの動作としては、以下となります。

電源投入前に、VR を目一杯時計回り(軸を正面からみて)に回してください。



モータドライバボードに電源を投入してください。

SW1 を ON にして、徐々に VR を反時計回りに回していきます。



(オシロスコープがある場合は、QL 端子の duty を観測してください)
最初は変化がないと思いますが(ここで電源から流れ出る電流がモニタ出来る場合は、徐々に電流値が増していると思います)、ある程度 VR を回すとモータの軸が振動を始めるはずです。

VR をもっと回すと、モータの軸の振動が大きくなり、いずれスムーズに回転を始めると思います(このときの、duty は 15%程度で、電流は、0.15A~0.2A 程度です)。

回転前のモータの軸が振動している状態の時は、消費電流が大きく、回転を始めると消費電流は減ります。

VR をもっと回すと、消費電流は増加します。

VR を目一杯回すと、モータからブーンとうなる音が聞こえてくると思います。(消費電流は~1A 程度となります、本プログラムの duty は最大 25%程度に設定しています)

本プログラムでは、モータに流す電流の向き(＝印加磁界の向き)は、一定周期(6ms)で変化させ、モータに流す電流の大きさは、VR の回転角度に連動させています。

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL4


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
VR -> duty(0-25%)

COMMAND:
s : stop <-> start display information(toggle)

>
Motor driver board connection check...
  CH-1 Connected.
```

起動時、端末には上記メッセージが出力されます。本チュートリアルでは、端末からプログラムに指示を出すコマンドが用意されており、動作時に端末から"s"を入力すると画面表示を止めることが出来ます。(再度"s"を押すと、画面表示は再開)(s コマンドはチュートリアル 3 から実装)

・シリアル端末から出力される情報 3 秒に 1 回更新)

```
  CH-1 START

                        CH-1
Motor Driver Board     :  Connect
Active                 :     o
Temperature(A/D value) :  2123
Temperature(degree)    :    27
VR(A/D value)          :     0
QL duty[%]             :   0.0

                        CH-1
Motor Driver Board     :  Connect
Active                 :     o
Temperature(A/D value) :  2121
Temperature(degree)    :    27
VR(A/D value)          :  1256
QL duty[%]             :   7.4

                        CH-1
Motor Driver Board     :  Connect
Active                 :     o
Temperature(A/D value) :  2121
Temperature(degree)    :    27
VR(A/D value)          :  2423
QL duty[%]             :  14.5
```

SW1=ON

Active = o になります
かつ、画面にメッセージが
3 秒毎に表示される様にな
ります

VR を回して duty を増やしていく

duty が増えてくるといずれ回転を
開始します

QL duty の値と回転の様子に着目してください。スムーズに回っている状態ですと 15%程度の duty になるのではないかと思います。それより小さいと、軸が振動するだけで回らず。それより大きいと、モータの振動が大きくなり、スムーズに回る感じではなくなると思います。モータ制御においては、duty の制御（=モータに与える電力）が重要であると言えるかと思います。

本チュートリアルでは、TUTORIAL3 で使用している機能に加え、GPT0 タイマを使用しています。

リソース	内容	用途	備考
r_gpt	GPT タイマ (GPT0)	モータに印加する電流（磁界の方向）のシフト	6ms 周期

6ms 毎に流す電流方向を変えていき（印加磁界の方向を変えていき）、モータを回す力を生み出しています。

・blm_intr.c 内 GPT0 割り込み関数

6ms 毎に呼び出される関数

```
void blm_interrupt_gpt0(timer_callback_args_t *p_args)
{
    //6ms 割り込み，モータ回転周期タイマ

    //モータに与える磁界を回転させてゆく処理
    //6ms 毎に、電流を流す方向を変えてゆく

    static unsigned short loop = 0;
    unsigned short motor_phase_control;
    unsigned short i;

    //ポートデバッグ有効時割り込み処理の先頭でポートを H にして、割り込み処理を抜ける際にポートを L にする
    BLM_DEBUG_PORT_3_H

    if (TIMER_EVENT_CYCLE_END == p_args->event)
    {

        switch(loop)
        {
            case 0:
                motor_phase_control = BLM_U_V_DIRECTION;
                break;

            case 1:
                motor_phase_control = BLM_U_W_DIRECTION;
                break;

            case 2:
                motor_phase_control = BLM_V_W_DIRECTION;
                break;

            case 3:
                motor_phase_control = BLM_V_U_DIRECTION;
                break;

            case 4:
                motor_phase_control = BLM_W_U_DIRECTION;
                break;

            case 5:
                motor_phase_control = BLM_W_V_DIRECTION;
                break;

            default:
                motor_phase_control = BLM_OFF_DIRECTION;
                break;
        }

        loop++;
        if (loop >= 6) loop = 0;

        for (i=0; i<BLM_CH_NUM; i++)
        {
            if (g_state[i] == BLM_CH_STATE_ACTIVE)
            {
                //blm_drive[N] は関数ポインタで、blm_drive_chN()
                blm_drive[i](motor_phase_control);
            }
            else
            {
                blm_drive[i](BLM_OFF_DIRECTION);
            }
        }
    }

    BLM_DEBUG_PORT_3_L
}
```

6ms で次の状態に移行

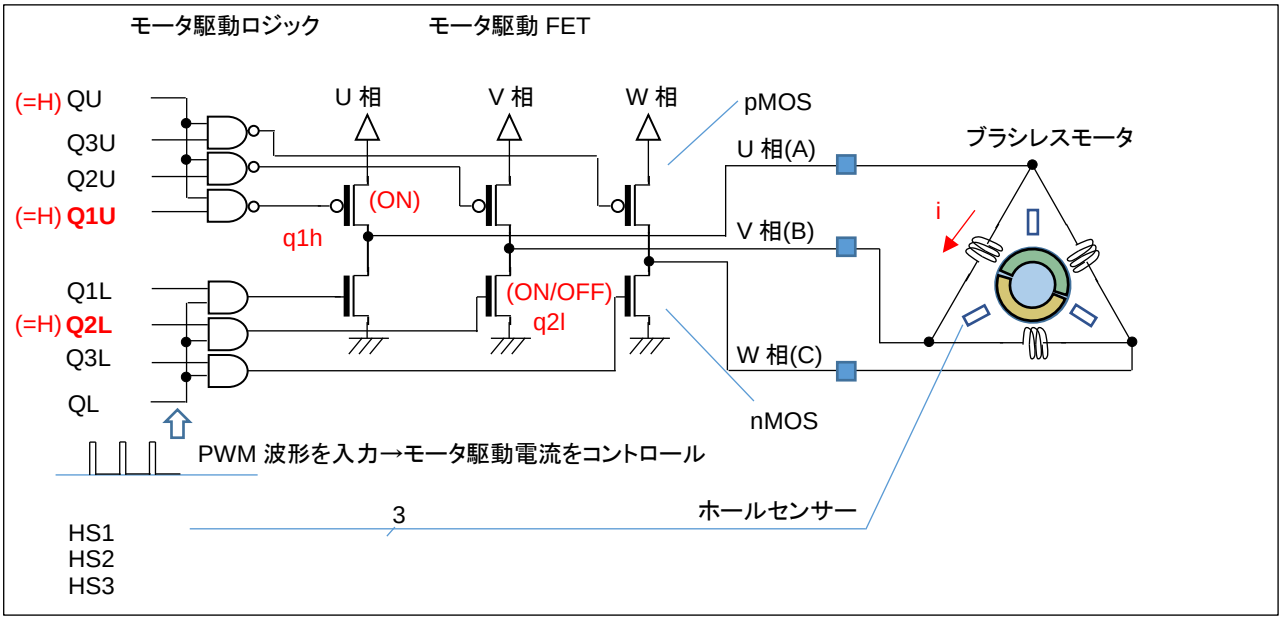
ループ変数をインクリメント

SW を倒して ON の状態の場合

実際にモータに流れる電流の向きを変更する

SW が OFF の時は駆動 OFF
（U, V, W 相の pMOS, nMOS の 6 素子全て OFF）

本プログラムでは、スイッチ(SW1)を ON にしてモータが ON ならば、モータに流す電流の向きを 6ms 毎に次の方向に切り替えて行きます。VR に連動した duty は、QL の信号に与えています。



QL には常に、(VR 回転角度に応じた)PWM 波形が入力されています。

プログラム的に g_motor_phase_control = U_V_DIRECTION のとき、q1h は ON(6ms の期間ずっと)していますが、q2l は、ON/OFF を断続的に繰り返しています。(ON している割合が duty 比)モータの U→V に流れる電流も、断続的に流れる・止まるを繰り返しています。duty 比が平均電流となりますので、duty 比を大きくした方がモータの軸を回す力も大きくなります。(pMOS, H 側は ON、nMOS, L 側を PWM 制御)

(U_V_DIRECTION の時は、U 相の H 側(pMOS)と、V 相の L 側(nMOS)のスイッチング素子(MOS FET)が ON します。その他の方向も同様に、H 側と L 側を 1 つずつ ON させます。電流を流す方向は、1.2 章で説明した 6 パターンとなります。)

本プログラムでは、6ms 毎に 1/6 回転する制御としています(回転数は固定です)。1 回転で、36ms なので、回転数としては、28 回転/s。1 分あたりの回転数は、1667rpm です。モータの回転方向は反時計回り(軸方向から見て)です。

これは、回転数を固定とし、モータに流れる平均電流を変化させモータを回転させる手法で、電流が小さいときはモータが回らず、電流が大きいときには(モータが発する音が大きくなり)無駄なエネルギーが消費されています(モータの軸を回す力が大きく、1667rpm より速く回す能力があるにも拘わらず、回転数がプログラムで 1667rpm に固定されているためです)。

特定の回転数でモータを回すためには、モータに流す平均電流を制御する必要があります。ここでは duty 比で電流を変化させていますので、回転数に応じた duty 比の制御が必要になると考えてください。
(なお、モータの軸に負荷をつなげている場合は、負荷の大きさに対応して duty を増やす必要があります。)

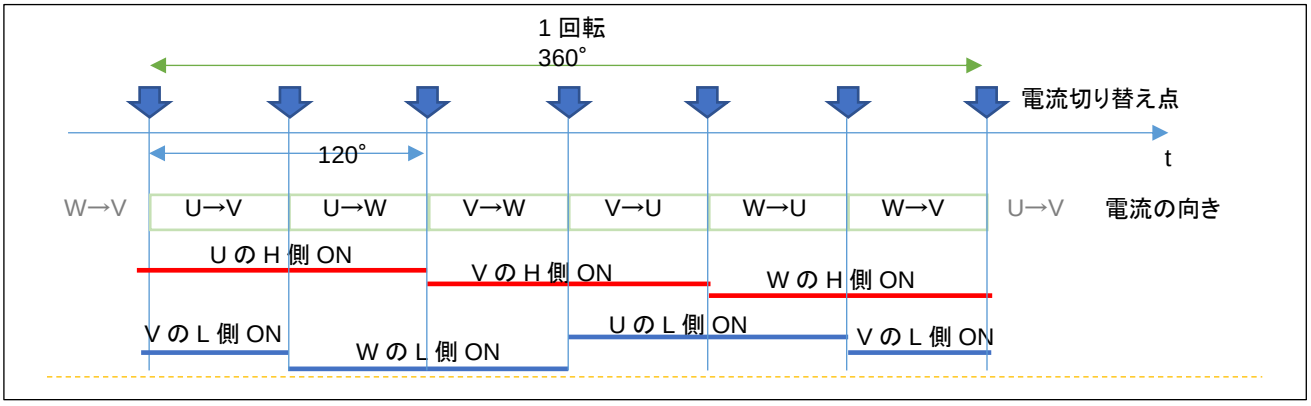
時系列	電流の向き	U 相		V 相		W 相		H 側	L 側
		Q1U (P208)	Q1L (P407)	Q2U (P101)	Q2L (P100)	Q3U (P103)	Q3L (P102)		
↓	U→V	○			○			U が ON	V が ON
	U→W	○					○		W が ON
	V→W			○			○	V が ON	
	V→U		○	○					U が ON
	W→U		○			○		W が ON	
	W→V				○	○			V が ON

○は MOS FET（トランジスタ）が ON（端子を H 制御）です。（空欄は OFF，端子は L 制御）

H 側の制御に着目すると、1 回転の間に U 相が ON するタイミング、V 相が ON するタイミング、W 相が ON するタイミングがそれぞれ 1 回訪れます。L 側の制御においても同様です。

1 回転を 360°とすると、120°単位で ON する素子が切り替わるので、この様な制御は「120 度制御」と呼ばれます。

・120 度制御



本チュートリアルでは、モータに流す電流方向を変化させ、適切な duty を与えればモータが回転する事を示しています。次のチュートリアルでは、モータに内蔵されたセンサを用い、モータが回っているのか、止まっているのか。また、現在の軸の絶対位置を読み取る方法を示します。

・チュートリアル 4 での端子設定

→チュートリアル 3 に同じ

・チュートリアル 4 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_gpt3	r_gpt	GPT3	QL PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	
g_gpt0	r_gpt	GPT0	6ms タイマ	印加電流の向きを変化させるのに使用

※グレーの項目は前チュートリアルから変更なし

ーポートデバッグに関してー

モータ制御プログラムでは UART に各種情報を出力できるようになっていますが、信号切り替わりや割り込みが入ったタイミング等、リアルタイム性が要求されるような情報は、UART 経由での出力には適していません。

そこで、本チュートリアルでは、I/O ポートをデバッグ用に使用できるよう設定しています。

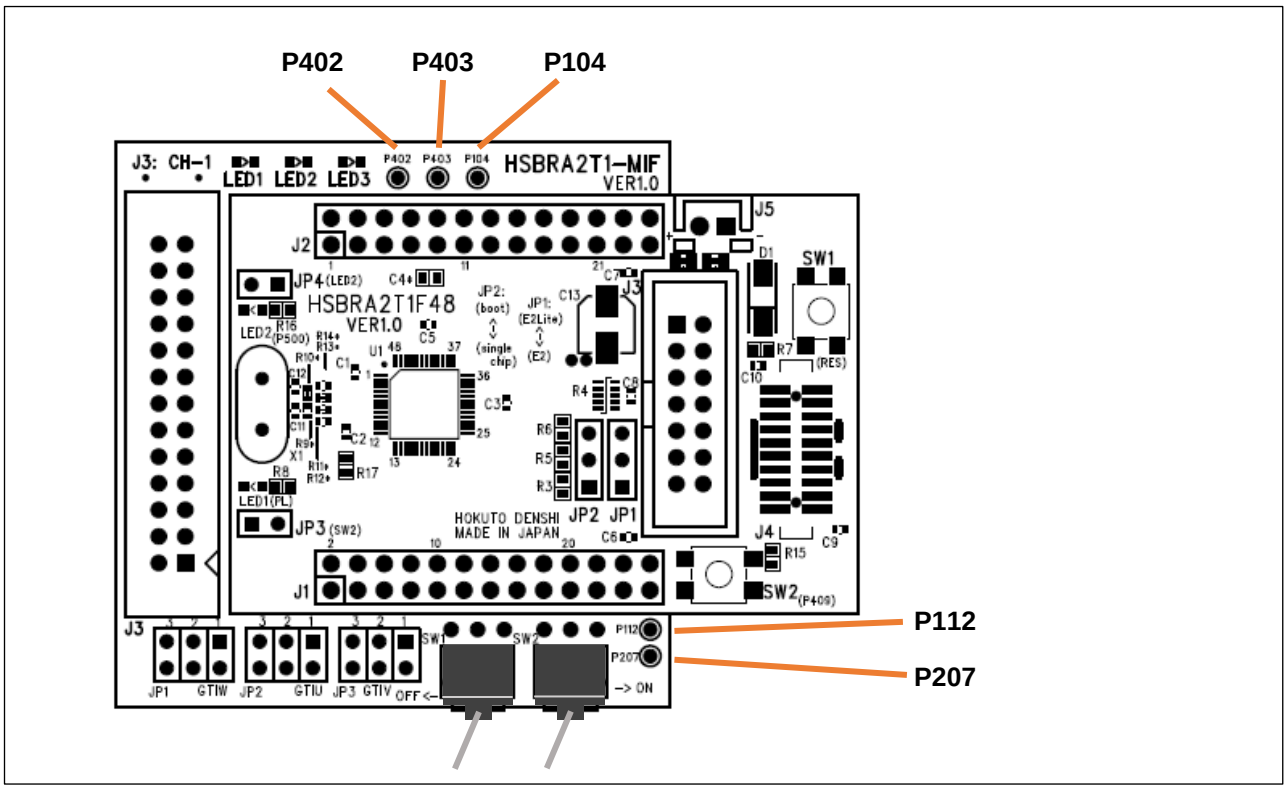
blm.h 内

```
//デバッグ用ポート
#define BLM_PORT_DEBUG      //定義時ポートによるデバッグを有効化する
```

上記定数を定義した場合は、（デフォルトで有効化しています）

端子名	ポートから L 出力	ポートから H 出力	ポートの L/H を 切り替える	備考
P402	DEBUG_PORT_1_L	DEBUG_PORT_1_H	DEBUG_PORT_1_T	
P403	DEBUG_PORT_2_L	DEBUG_PORT_2_H	DEBUG_PORT_2_T	
P104	DEBUG_PORT_3_L	DEBUG_PORT_3_H	DEBUG_PORT_3_T	
P112	DEBUG_PORT_4_L	DEBUG_PORT_4_H	DEBUG_PORT_4_T	
P207	DEBUG_PORT_5_L	DEBUG_PORT_5_H	DEBUG_PORT_5_T	

上表の端子をデバッグ端子に設定して、モニタする事が出来ます。



デバッグモニタ端子は、接続ボード上のスルーホールに接続されています。

例えば、割り込み処理の先頭で

DEBUG_PORT1_H

を実行し、割り込み処理の終わりに

DEBUG_PORT1_L

を入れておくと、P402 の H のパルス幅が（概ね）割り込み処理に掛かる時間という事で観測可能です。

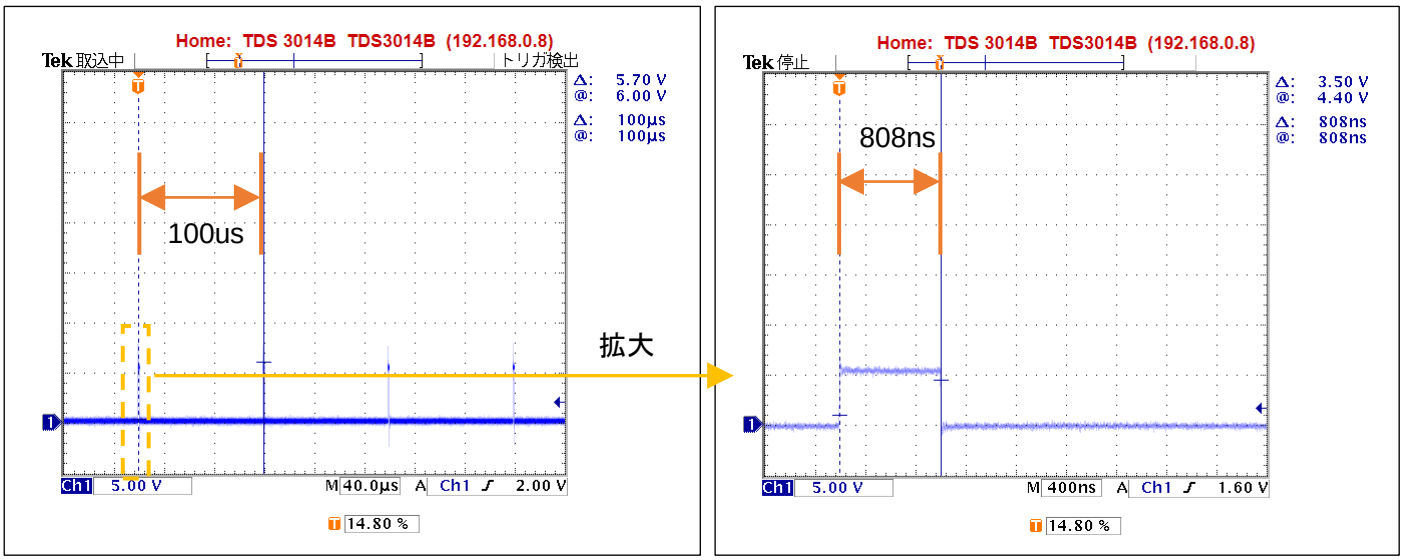
本チュートリアル以降、以下のポートデバッグを入れてあります。

・100us の割り込み処理(AGT0)
DEBUG_PORT_1_H ～ DEBUG_PORT_1_L

・10ms の割り込み処理(AGT1)
DEBUG_PORT_2_H ～ DEBUG_PORT_2_L

・6ms の割り込み処理(GPT0) ※本チュートリアル限定
DEBUG_PORT_3_H ～ DEBUG_PORT_3_L

・100us の割り込み処理(AGT0)をモニタした結果



100us 周期の割り込みを実行しており、パルスが 100us 毎に立っているので、周期設定等の誤りがないことが確認できます。

1 つのパルスを拡大すると、割り込み処理に掛かる時間は、0.8us 程度で 100us に対して十分にに短い時間内で割り込み処理が終わっているので問題がない事が判ります。

（もし、割り込み処理の実行時間が 100us を超える様であれば、そのような処理は 100us の割り込み外で実行する様にするなどの判断材料となります。）

1.5. ホールセンサの値をみる

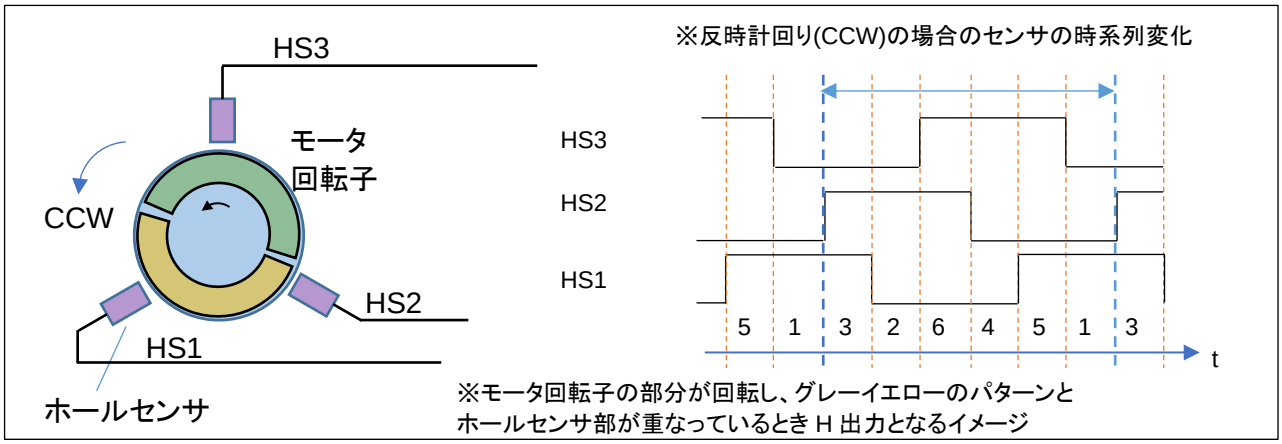
参照プロジェクト:RA2T1_BLMKIT_TUTORIAL5

本チュートリアルでは、ホールセンサの値を読み取っています。
TUTORIAL4 同様、必要に応じてシリアル端末を接続してください。

・シリアル端末から出力される情報

```
Motor driver board connection check...
   CH-1 Connected.

pos = 2                    ← ホールセンサ位置情報 2
                           (7 の場合はモータドライバボード未接続)
```

電源を投入すると、上記の表示がシリアル端末に出力されます。
表示は、0.1s 間隔で更新しています。ここで、モータの軸を手で回してみてください。数字が 1 から 6 まで変化すると思います(数値の変化の仕方は、一見順不同に見えると思います)。
この数値は、ホールセンサの位置を示しています。モータの軸は 1 回転あたり、6 回クリック(止まるところ)があると思います。モータの軸が 1 のとき
1　[時計回りに軸を回転] → 5
1　[反時計回りに軸を回転] → 3
となるはずです。この数値は、軸の絶対位置を表しています。



ホールセンサの値は、基本的には duty=50%の矩形波が 120°ずれて出力されるイメージです。

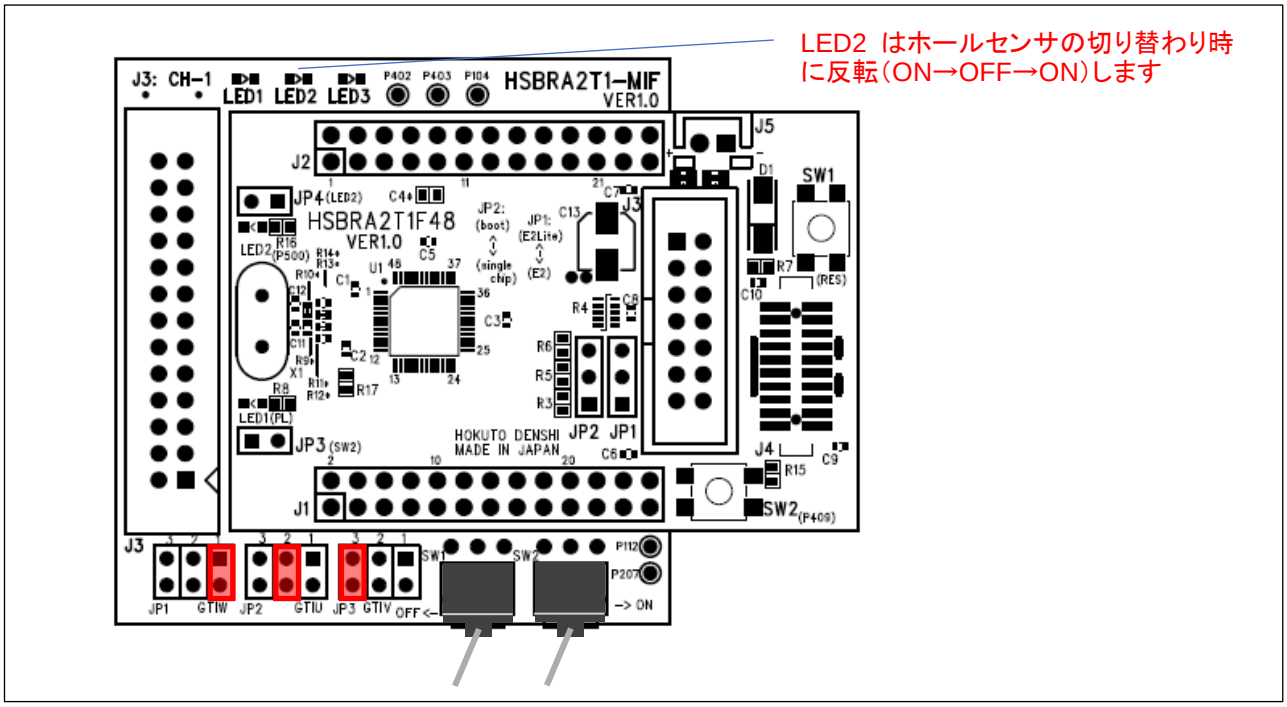
※軸が回る方向を、モータを軸方向から見て、CCW(反時計回り, CounterClockWise)、CW(時計回り, ClockWise)と表記します。

本プログラムで表示される数値は、

数値(pos)	[HS3] (bit2)	[HS2] (bit1)	[HS1] (bit0)
ホールセンサ端子	P409	P206	P409
3	0	1	1
2	0	1	0
6	1	1	0
4	1	0	0
5	1	0	1
1	0	0	1

0=L
1=H

pos = HS3×4 + HS2×2 + HS1
となります。(プログラムの処理を容易にするため、HS3~HS1 に重み付けを行い加算した数値)



		1	2	3
JP1	P408/GTIW	**HS1**	HS2	HS3
JP2	P206/GTIU	HS1	**HS2**	HS3
JP3	P409/GTIW	HS1	HS2	**HS3**

モータのホールセンサ出力(HS1~HS3)と、マイコンのホールセンサ入力(P408, P206, P409)は、JP1~JP3 のジャンパで接続関係を選択できるようになっています。本チュートリアル以降のホールセンサを使用するチュートリアル実行時は、ジャンパを上記設定としてください。

ホールセンサの出力は、接続されているマイコンのピンを単純に入力回路に設定して、デジタル的に読み取っています。

モータの軸を手で回すと、上記表の数値に従い変化 1→5 または 1→3（回転方向による）すると思います。これは、プログラムで「いまモータの回転軸がどの位置にあるのか」を把握できていることを示します。

本モータのホールセンサは、軸 1 回転につき、出力が 6 値変わりますので、軸位置が 60 度変化すると、ホールセンサの出力が変化するという事となります。

※モードライバボードやホールセンサ端子が接続されていない場合は、数値が 7 となります

ここで、SW1 を ON にしてみてください。
（スイッチを ON にすると、pos の画面表示は止まります）

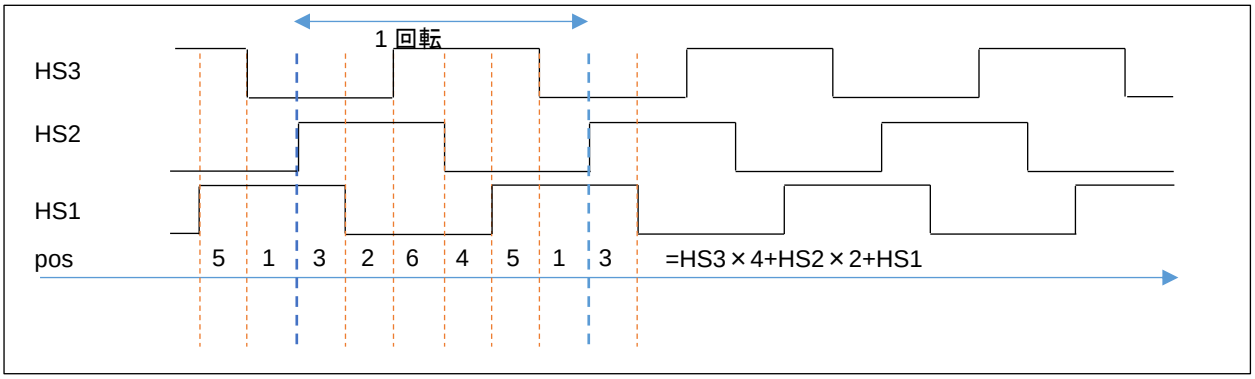
VR を回すと、TUTORIAL4 のプログラム同様、モータが回転を始めると思います。しかし、このプログラムでは、VR の回転に応じてモータの回転数が変わる事（モータの回転数は、音からある程度判断するしかないのですが）にお気付きでしょうか。TUTORIAL4 のプログラムは、回るか・回らないか。回ったときは常に 1670rpm ぐらいで回る（36ms で 1 回転）という動作でした。しかし、このプログラムでは、ホールセンサーの読み取り値が変化したときに電流の向きを変化させます。（正確には、電流の向きを変えるのは、タイマ割り込みの 100us 刻みのタイミングです。）

なお、VR を回す→QL 信号の duty 比を変えるという動作は、TUTORIAL4 と同じです。TUTORIAL4 と異なるのは、電流の向きを変えるタイミングです。TUTORIAL4 では、6ms という決まったタイミングでしたが、本チュートリアルでは、ホールセンサが切り替わったタイミングが電流の向きを切り替えるトリガとなります。

モータの軸が 1/6 回転して、ホールセンサの値が変わった時点で、モータの回転軸を引っ張る（押し出す）磁界を生む電流にスイッチできるため、流れる電流（このプログラムでは、VR に連動した duty）を大きくすると、モータの回転軸にかかる力が大きくなり、速く次のホールセンサ切り替え（1/6 回転）に達するため、VR(duty)とモータの回転数が連動する動作となります。

言い換えると、TUTORIAL4 のプログラムは、duty を大きくすると、モータの回転軸は速く次の 1/6 回転に達しますが（pos=5→1 等）、そこ（pos=1）で同じ場所（pos=1）に引っ張る力をキープしますので、エネルギーが無駄に消費され、電流は増加するものの回転数は変わらないという動作です。

・ホールセンサ位置と電流印加方向に関して



pos	反時計回り(CCW)	時計回り(CW)
3	U→V	W→U
2	U→W	W→V
6	V→W	U→V
4	V→U	U→W
5	W→U	V→W
1	W→V	V→U

※矢印の向きは時系列、本チュートリアルでは回転方向は「反時計回り(CCW)」固定です

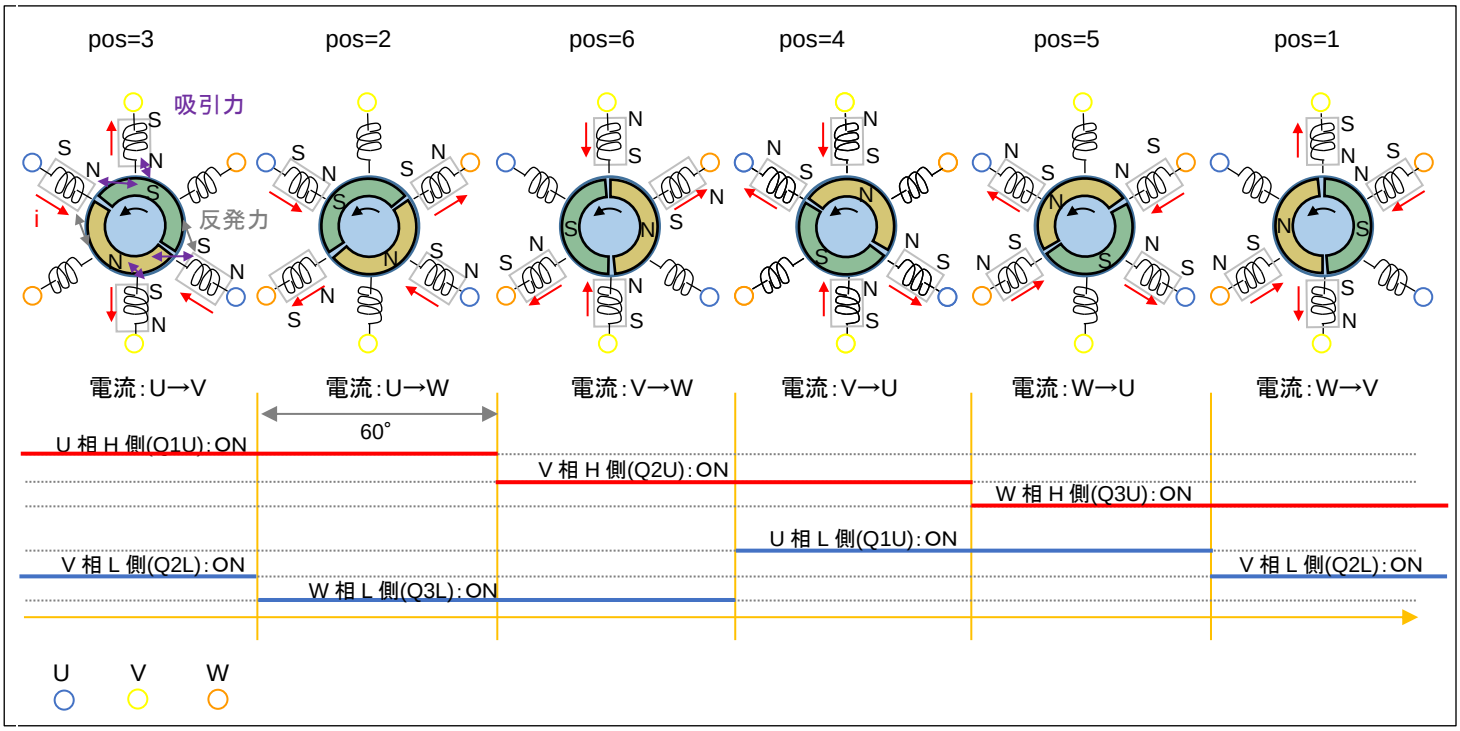
```
                              CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])     :   3660
Temperature(A/D value)    :   2073
Temperature(degree)       :     26
VR(A/D value)             :   3152
QL duty[%]                :   22.7
```

本チュートリアルでは、前チュートリアルと異なり、duty の値に応じてモータの回転数が変わります

本チュートリアルでは、回転数の表示が追加されています。

・VR の回転角に応じて duty が変わる：TUTORIAL4 と TUTORIAL5 で同じ動作
・duty に応じて回転数が変わる：TUTORIAL5 での新機構（TUTORIAL4 は回転数固定）

・反時計回り(CCW)

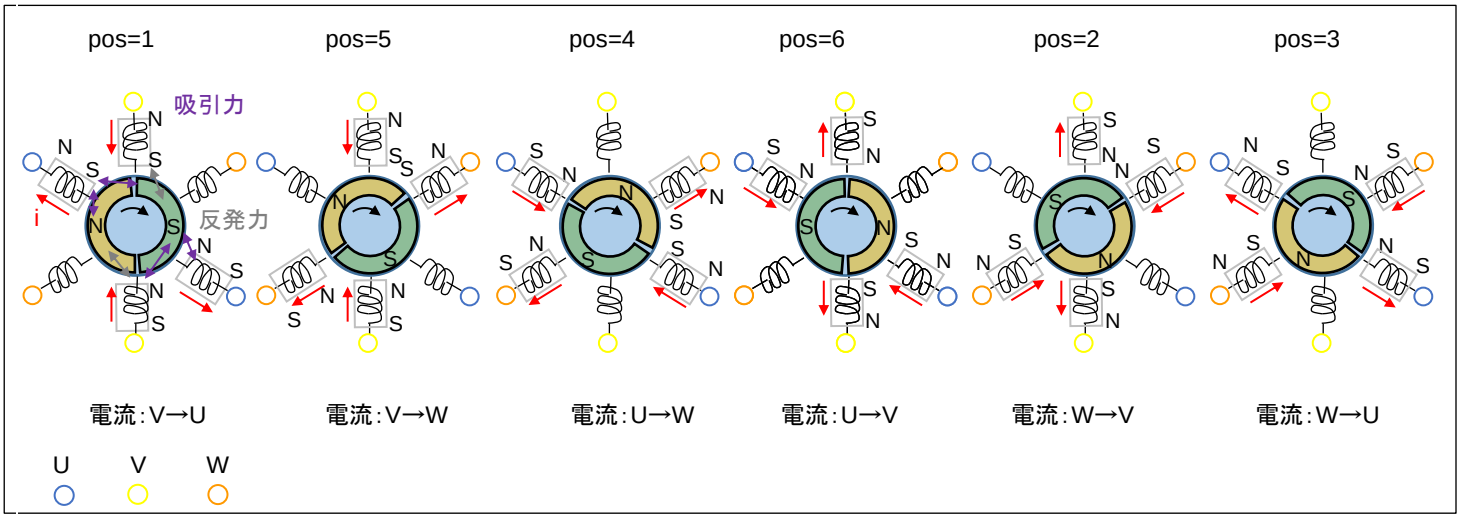


モータが 60 度回転した時点で(60 度回転した事はホールセンサで判断します)、次の電流パターンにシフトさせます。UVW の 3 相、H 側/L 側の計 6 本の制御信号を 120° 毎に ON/OFF を切り替えていく制御となりますので、この制御方法は TUTORIAL4 同様「120 度制御」です。
※コイルの数や配置はイメージです(実際のモータ内部の構成とは必ずしも同じではありません)

[参考]

本チュートリアルでは、回転方向は固定ですが、逆回転(時計回り(CW))させる場合は、以下の様になります。

・時計回り(CW)



・blm_intr.c 内 AGT0 割り込み関数 blm_interrupt_agt0　　100us 毎に呼び出される関数

```
    //モータ回転制御
    for (i=0; i<BLM_CH_NUM; i++)
    {
        g_sensor_pos[i] = blm_hall_sensor_pos[i]();

        if (g_state[i] == BLM_CH_STATE_ACTIVE)
        {
#if 1    //1 を 0 に変えると逆回転となる

            //回転方向:CCW
            switch(g_sensor_pos[i])
            {
                case 3:
                    blm_drive[i] (BLM_U_V_DIRECTION);
                    break;
                case 2:
                    blm_drive[i] (BLM_U_W_DIRECTION);
                    break;
                case 6:
                    blm_drive[i] (BLM_V_W_DIRECTION);
                    break;
                case 4:
                    blm_drive[i] (BLM_V_U_DIRECTION);
                    break;
                case 5:
                    blm_drive[i] (BLM_W_U_DIRECTION);
                    break;
                case 1:
                    blm_drive[i] (BLM_W_V_DIRECTION);
                    break;
                default:
                    blm_drive[i] (BLM_OFF_DIRECTION);
                    break;
    }
#else
            //回転方向:CW
            switch(g_sensor_pos[i])
            {
                case 3:
                    blm_drive[i] (BLM_W_U_DIRECTION);
                    break;
                case 2:
                    blm_drive[i] (BLM_W_V_DIRECTION);
                    break;
                case 6:
                    blm_drive[i] (BLM_U_V_DIRECTION);
                    break;
                case 4:
                    blm_drive[i] (BLM_U_W_DIRECTION);
                    break;
                case 5:
                       blm_drive[i] (BLM_V_W_DIRECTION);
                       break;
                case 1:
                    blm_drive[i] (BLM_V_U_DIRECTION);
                    break;
                default:
                    blm_drive[i] (BLM_OFF_DIRECTION);
                    break;
            }
#endif
```

ホールセンサ値の読み取り

回転方向:反時計回り(CCW)

ホールセンサ位置 3(HS3..HS1=0b011)の時
U 相から V 相に電流を流す

ホールセンサにより算出された
現在のモータ回転子の位置
(g_sensor_pos)に応じて
流す電流の向きを決める

回転方向:時計回り(CW)
→デフォルト無効

ホールセンサ位置 3(HS3..HS1=0b011)の時
W 相から U 相に電流を流す

ホールセンサの情報を使うことにより、最適なタイミングでモータの軸を引っ張る磁界を生成できますので、本プログラムでは、duty（平均電流:トルクに対応）を増やすと、モータの回転数が上がります。

なお、本チュートリアルでは、回転方向は固定です（軸方向から見て、反時計回り）。実際のアプリケーションで、回転方向を変える必要がある場合は、プログラムのテーブル（pos=3 のとき、U→V に電流を流す）を前ページの図の対応に変える必要があります。

・チュートリアル 5 での画面表示

```
                              CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])     :    3660
Temperature(A/D value)    :    2073
Temperature(degree)       :      26
VR(A/D value)             :    3152
QL duty[%]                :    22.7
```

本チュートリアルでは、回転数の表示が追加されています。

100us 毎に、カウンタ変数値をインクリメントしていき、1 回転した際、
・グローバル変数(g_rotation_counter)にカウンタ変数値を保存
・カウンタ変数値のリセット（0 代入）
しています。そして、画面表示のタイミングで、変数値を 1 分間あたりの回転数([rpm])に変換しています。

```
period = (float)g_rotation_counter[i] * BLM_CONTROL_PERIOD; //1 回転の周期
rpm[i] = (long)(1.0f / period) * 60L;                       //[rpm]変換
```

ホールセンサは、1/6 回転で値が変化しますが、pos=1 の時点（＝1 回転したタイミング）でカウンタ変数をリセットしているので、1 回転の周期は

1 回転するまで 100us(=BLM_CONTROL_PERIOD)で何回カウントされたか　×　100us　...(1)

で求まります。

回転数は、周期の逆数なので、(1)の逆数が 1 秒あたりの回転数。モータ等の回転数は、rpm, 1 分間あたりの回転数で表すことが多いため、1 秒間あたりの回転数×60 で計算しています。

※両方向の回転に対応させる場合、回転方向により電流を流すテーブルを変更します
※本チュートリアルでは、時計回り(CW)の回転方向のプログラムはコメントアウトで実装されています
※前ページの「回転方向:CW」の方を有効にすれば、モータは逆回転となります
（blm_interrupt_agt0()内の「#if 1」の部分を「#if 0」に変えると逆回転となります。）

・チュートリアル 5 での端子設定

端子名	役割	割り当て	備考
P000~P015	AD0~AD6,VR AD003	アナログ入力(ADC)	
P100~P103	Q2U~Q3L	出力（初期値 L）	モータ駆動端子
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P206	HS2(*1)	入力（プルアップ）	ホールセンサ入力として使用
P208	Q1U	出力（初期値 L）	モータ駆動端子
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P400	QU	出力（初期値 L）	
P401	QL	周辺機能(GPT3)	GTIOC3B として設定
P407	Q1L	出力（初期値 L）	モータ駆動端子
P408, P409	HS1, HS3(*1)	入力（プルアップ）	ホールセンサ入力として使用
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

・チュートリアル 5 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_gpt3	r_gpt	GPT3	QL PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	
~~g_gpt0~~	~~r_gpt~~	~~GPT0~~	~~6ms タイマ~~	本チュートリアルでは GPT0 は未使用

※グレーの項目は前チュートリアルから変更なし

1.6. 過電流・過熱保護の動作

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL6

本チュートリアルでは、スイッチを ON にし、VR を回していくとモータが回転し、回転数が上がっていきます。

基本的な動作は、TUTORIAL5 と同じです。TUTORIAL5 のプログラムは、duty の最大値を 25%弱に制限していますが、本プログラムでは 100%まで duty を上げられます。VR を回していくと、回転数が上がりますが、一定以上まで上げた場合、急にモータの回転が止まるはずです(*1)。このとき、LED3 が点灯していると思います。これは過電流保護機構が働いたためです。モータドライバボード側では、過電流検出機構が働くと、*INT が L になります(L パルスが出ます)。マイコンボード側で、この信号は、P111/IRQ4 につながっており、本プログラムでは以下の条件でモータに流れる電流を遮断し、モータを止める制御を行います。(過電流停止した場合、LED3 が点灯します)

(a)1 回でも*INT の信号が L になった場合
(b)100us 毎に*INT の信号をチェックし、10ms 間に 50 回(*2)以上過電流である場合
(c)100us 毎に*INT の信号をチェックし、1 秒間に 1000 回(*2)以上過電流である場合

過電流の判定基準は、(a)~(c)のどの条件を有効にするかは任意の組み合わせで設定可能です。(a)が一番厳しい判定基準です。(a)を有効にした場合は、(b)(c)の判定前にモータが止まりますので、実質的には

(1) (a)を有効にする　　※本チュートリアルでは、C コマンドで「(a)有効」「(a)無効」を切り替えます
(2) (b)及び(c)を有効にする
(3) (b)を有効化する
(3) (c)を有効化する

のいずれかの選択となります。(b)はある程度短期の判定基準。(c)は平均電流の判定基準のイメージです。(b)は 100 未満(10ms 間に 100us 毎に、100 回の判定となるので、100 以上の数値を指定すると、過電流エラーが検出される事がない)。(c)は、10,000 未満の任意の値を設定可能です。

(*1)電流リミットの掛けられる電源装置をお使いの場合は、電流リミットを 2A 程度に設定してください。
(電流リミットを 1A 程度に設定すると、電源装置側の電流リミットに引っかかり、過電流検出される電流まで達しません。)

(*2)50 回, 1000 回はデフォルトの設定値です。blm.h 内で数値を定義しているので、任意の値に変更可能です。

・起動時のメッセージ

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL6


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
LED3 : ERROR status
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)
C : Over current -> ONCE stop ENABLE <-> Over current -> ONCE stop DISABLE [toggle]
X : 10ms Over current -> stop ENABLE <-> 10ms Over current -> stop DISABLE [toggle]

>
Motor driver board connection check...
  CH-1 Connected.
```

起動時のメッセージは上記の様になっており、C と X のコマンドで過電流停止の条件を変更できるようになっています。キーボードから C, X を入力する度に ON/OFF がトグルで切り替わる様になっています。

コマンド	過電流停止の条件
C	1 回の過電流検出で停止(a)の有効・無効切り替え
X	10ms の規定回数で停止(b) の有効・無効切り替え
常に有効	1s の規定回数で停止(c)

C コマンド入力時は、(a)の 1 回の過電流停止を無効化。X コマンド入力時は、(b)の 10ms の条件を無効化します。（再度 C, X コマンドを入力した場合は、(a)の 1 回の過電流停止, (b)の 10ms の過電流停止を有効化します。）

・シリアル端末から出力される情報（過電流停止）

```
   CH-1 START

                                CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])     :   8940
Temperature(A/D value) :   2080
Temperature(degree)       :      26
VR(A/D value)             :   2113
QL duty[%]                :   50.0

                                CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])     : 10860
Temperature(A/D value) :   2086
Temperature(degree)       :      26
VR(A/D value)             :   2592
QL duty[%]                :   63.3
   CH-1 STOP

 *** OVER CURRENT ( COUNT = ONCE(interrupt) ) ***
```

過電流検出で停止

PC 上では、teraterm 等のシリアル端末ソフトで表示してください

115,200bps, 8bit, none, 1bit の設定で表示できます

VR を回して duty を上げていくと

上記は(a)の 1 回の過電流信号の割り込みで停止した場合です。過電流で停止した場合、一度 SW を OFF にすると、エラーはクリアされます。

次に、C コマンドを入力して、同様の duty を上げてみます。

・(a)の条件を無効化（C コマンドを入力）

```
Over current stop(ONCE) -> OFF

                                CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])     : 12720
Temperature(A/D value) :   2107
Temperature(degree)       :      27
VR(A/D value)             :   3068
QL duty[%]                :   74.6

   CH-1 STOP

 *** OVER CURRENT ( COUNT = 68 / 10[ms]) ***
```

そうすると、(b)の条件に引っかかってモータが停止します。

次いで、X コマンドを入力します。

・(b)の条件を無効化（C コマンドと X コマンドを入力）

```
10ms Over current stop -> OFF

                                CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])    : 13020
Temperature(A/D value) :    2149
Temperature(degree)       :      28
VR(A/D value)             :   3115
QL duty[%]                :   75.8

   CH-1 STOP

 *** OVER CURRENT ( COUNT = 1973 / 1[s]) ***
```

この場合は、(c)の条件に引っかかってモータが停止します。
（一般的には(b)より(c)の方が緩い条件となります。）

過熱保護機能に関しては、モータドライバボード上に搭載されているサーミスタ（1.3 章を参照）の温度のモニタリングを定期的に行い、設定した閾値を超えた場合に、モータを停止させるというものです。

プログラムでは、10ms 間隔で温度のモニタリングを行っており、1 回でも閾値を超えた場合モータが停止します。

・シリアル端末から出力される情報（過熱停止）

```
                                CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])    :   5880
Temperature(A/D value) :    2828
Temperature(degree)       :      49
VR(A/D value)             :   1406
QL duty[%]                :   34.0

   CH-1 STOP

 *** OVER TEMP ( TEMP = 51 [deg]) ***                    過熱検出で停止
```

過熱停止の場合の表示例です。

エラーとなった場合は、LED3 が点灯し動作が停止します、その後 SW1 を OFF するとエラーはリセットされます。

・blm.c, blm_init()内

```
//エラーチェックフラグ
g_error_check_flag = 0;

//過熱停止有効
g_error_check_flag |= BLM_ERROR_OVER_TEMP_STOP;  //(d)

//1回の過電流検出で停止
g_error_check_flag |= BLM_ERROR_OVER_CURRENT_STOP1;  //(a)

//10msの間規定回数以上の過電流検出で停止
g_error_check_flag |= BLM_ERROR_OVER_CURRENT_STOP2;  //(b)

//1sの間規定回数以上の過電流検出で停止
g_error_check_flag |= BLM_ERROR_OVER_CURRENT_STOP3;  //(c)
```

プログラム内では、

g_error_check_flag

変数の値で、(a)(b)(c)及び過熱停止(d)の有効化を行います。

(a) BLM_OVER_CURRENT_STOP1(=0x2)　1 回の過電流検出信号で停止
(b) BLM_OVER_CURRENT_STOP2(=0x4)　10ms 間に規定の回数（デフォルト 50 回）以上過電流検出で停止
(c) BLM_OVER_CURRENT_STOP3(=0x8)　1 秒間に規定の回数（デフォルト 1000 回）以上過電流検出で停止
(d) BLM_OVER_TEMP_STOP1(=0x1)　　過熱停止

過熱停止と 1 回の過電流検出で停止を有効にする場合。
g_error_check_flag = BLM_OVER_TEMP_STOP1 | BLM_OVER_CURRENT_STOP1;
(g_error_check_flag = 0x3)

g_error_check_flag には、有効にしたい停止方法を OR(|)で与えてください。

・blm.h

```
//過電流停止カウント回数
#define BLM_OVER_CURRENT_COUNT_10MS 50//100us 毎にチェックを行い 10ms あたり 50 回以上過電流検出で停止（最大 100）
#define BLM_OVER_CURRENT_COUNT_1S 1000//100us 毎にチェックを行い 1s あたり 1000 回以上過電流検出で停止（最大 10,000）

//過熱停止[℃]
#define BLM_OVER_TEMP   50
```

(b)(c)の電流検出の(d)の過熱停止の閾値は、上記で定義されていますので別な値に変える事も可能です。

・過電流検出で使用している端子

モータドライバボード側の信号名は *INT です。この信号がマイコンボード側では以下の端子に接続されています。

	端子名	備考
CH-1	P111/IRQ4	(a)の場合は IRQ4, (b)(c)の場合は汎用入力として使用されます

※モータドライバボード側の過電流検出は、U 相と V 相の電流が両方 8A ピークを越えた場合*INT=L となります

(a)の IRQ（端子割り込み）を使用する場合は立ち下がりエッジの検出。(b)(c)の場合は、100us 間隔で端子のレベルを読み取り、10ms, 1s 間の L の回数をカウントします。

・過熱保護で使用している端子

モータドライバボード側の信号名は AD6 です。この信号がマイコンボード側では以下の端子に接続されています。

	端子名	備考
CH-1	P010/AN005	A/D 入力として使用

(a)1 回でも *INT の信号が L になった場合停止させる処理

・blm_intr.c

IRQ4 は立下りエッジ検出に設定

```
void blm_interrupt_irq4(external_irq_callback_args_t *p_args)
{
    //CH-1 過電流割り込み

    (void)p_args;

    if (g_error_check_flag & BLM_ERROR_OVER_CURRENT_STOP_1)
    {
        blm_stop[BLM_CH_1]();
        g_error[BLM_CH_1].status |= BLM_ERROR_OVER_CURRENT_STOP_1;
        g_state[BLM_CH_1] = BLM_CH_STATE_INACTIVE;
        blm_led_change_on(BLM_LED_ERROR);       //LED（エラーステータス）を ON
    }
}
```

IRQ4 の割り込みが入った場合、即モータを停止させる処理です。

※グレーの部分は、本チュートリアル限定（他のチュートリアルでは、初期状態で有効／無効を選択、本チュートリアルでは、C コマンドの入力により有効／無効を切り替え）

(b)(c)100us 毎に過電流をチェックする処理

・blm_common.c

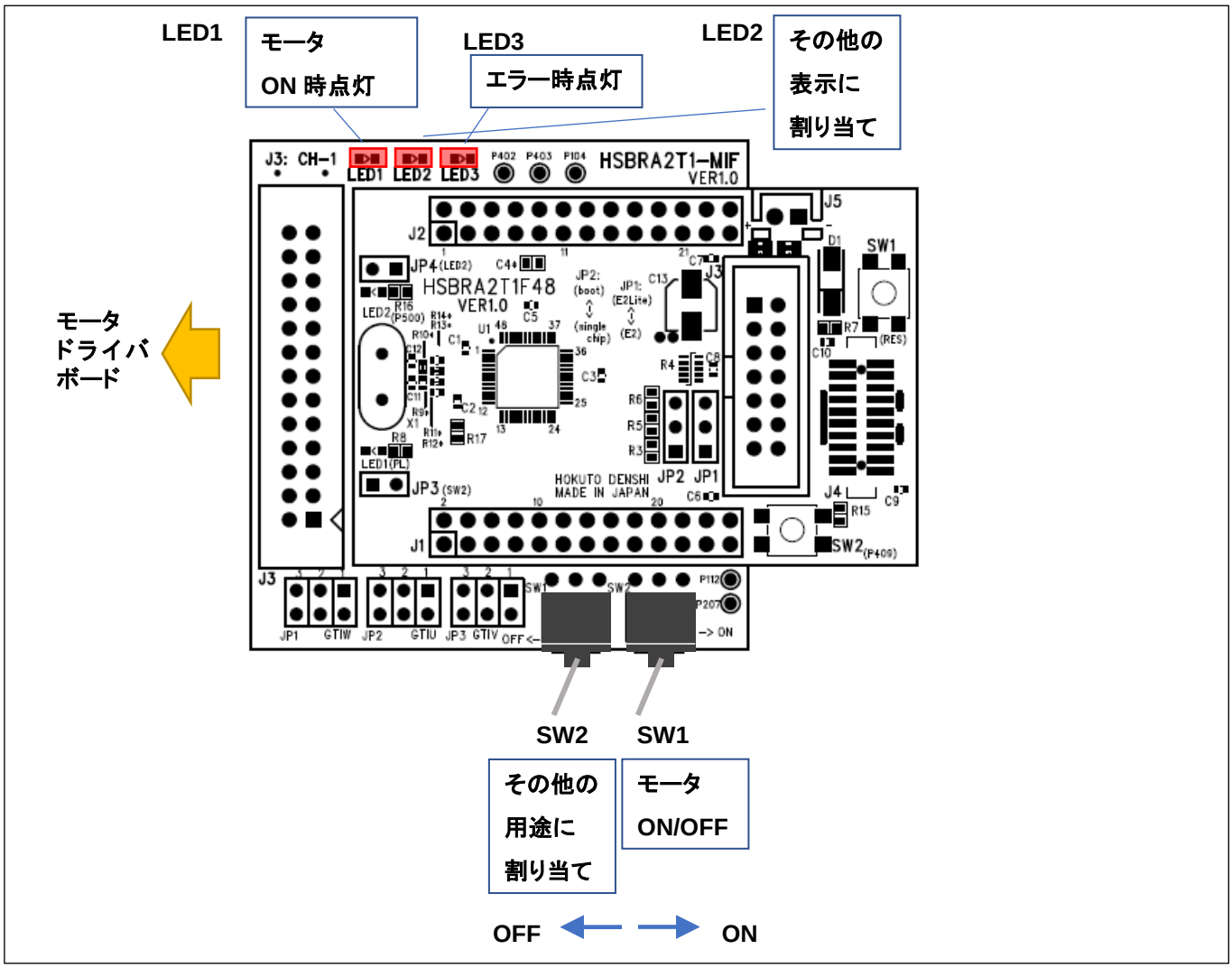
```
unsigned short blm_current_monitor_ch1(void)
{
    //過電流検出 CH-1

    //戻り値
    //  過電流検出なし : 1 (true)
    //  過電流検出あり : 0 (false)

    return R_PORT1->PIDR_b.PIDR11;
}
```

単純に端子のレベルを読む処理
（100us 毎に実行）

100us 毎に電流をチェックするのは、AGT0（100us タイマ）の割り込み処理内で実行しています。

10ms 毎のチェック(b)や 1 秒毎のチェック(c)、過熱停止のチェック(d)は、AGT1(10ms タイマ）の割り込み処理内で実行しています。

・基本的な LED と SW の役割



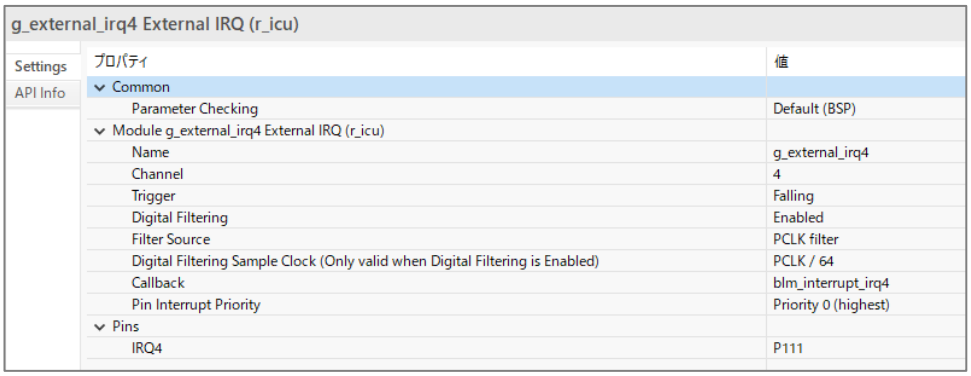
	割り当て	備考
SW1	CH-1 のモータ回転 ON/OFF	
SW2	その他の用途	本チュートリアルでは未使用

	割り当て	備考
LED1	CH-1 ON 時点灯（動作インジケータ）	
LED2	その他の用途	本チュートリアルでは未使用
LED3	エラーインジケータ	エラー（過電流、過熱検出）時に点灯

過電流停止（1 回の割り込みで停止）では、P111/IRQ4 の端子割り込みを使用しています。端子割り込み設定は、FSP では stack 追加で

New Stack > Input > External IRQ(r_icu)

を追加します。



		設定値	備考
Module	Name	g_external_irq4	使用する IRQ4 と名称を合わせる（任意）
	Channel	4	IRQ4 を使用
	Trigger	Falling	立下りエッジを検出
	Digital Filtering	Enabled	デジタルフィルタを有効化する
	Callback	blm_interrupt_irq4	割り込み時に呼び出される関数名を設定
	Pin Interrupt Priority	0	割り込み優先度
Pins	IRQ8	P111	IRQ4 の端子は複数から選択可能(*1)

(*1)Pins タブから ICU4 の端子設定で P111 を選びます

過電流停止は緊急性のある処理なので、割り込みの優先度は高く(0)設定しています。この場合、100us や 10ms の定期処理や、A/D 変換後の割り込み処理実行中でも、blm_interrupt_irq4()が優先して実行されます。

・チュートリアル 6 での端子設定

端子名	役割	割り当て	備考
P000~P015	AD0~AD6,VR AD003	アナログ入力(ADC)	
P100~P103	Q2U~Q3L	出力（初期値 L）	モータ駆動端子
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P111	*INT	IRQ4	過電流検出端子
P206	HS2(*1)	入力（プルアップ）	ホールセンサ入力として使用
P208	Q1U	出力（初期値 L）	モータ駆動端子
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P400	QU	出力（初期値 L）	
P401	QL	周辺機能(GPT3)	GTIOC3B として設定
P407	Q1L	出力（初期値 L）	モータ駆動端子
P408, P409	HS1, HS3(*1)	入力（プルアップ）	ホールセンサ入力として使用
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

・チュートリアル 6 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_gpt3	r_gpt	GPT3	QL PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	
g_external_irq4	r_icu	IRQ4	過電流検出	

※グレーの項目は前チュートリアルから変更なし

1.7. 相電圧・相電流の観測

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL7

本プログラムでは、A/D 変換の機能を使い、U, V, W の各相電圧および U, V, W の L 側の電流観測用抵抗に流れている電流を取得します。プログラムの動作としては、TUTORIAL6 と同様です。

・起動時のメッセージ

```
RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL7


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
LED3 : ERROR status
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display

>
Motor driver board connection check...
   CH-1 Connected.
```

'A'コマンド(キーボードから入力するコマンド)が追加されています。
(過電流停止の挙動を変える C, X コマンドは廃止されています。)

SW1 を ON にして、モータが回っている状態で、キーボードから'A'を入力してください。

・シリアル端末から出力される情報

```
                                CH-1
Motor Driver Board        :   Connect
Active                    :       o
rotation speed([rpm])     :   9960
Temperature(A/D value)    :   2103
Temperature(degree)       :     27
VR(A/D value)             :   2402
QL duty[%]                :   58.6

                キーボードから'A'を入力
   --- A/D information ---

CH-1 A/D Conversion result

serial V(U) V(V) V(W) I(U) I(V) I(W) V(Power) temp volume
0 2255 2735 2671 284 0 3 3768 2104 2397
1 2259 2730 2675 443 0 4 3763 2104 2402
2 2255 2713 2681 175 0 3 3765 2100 2401
3 2246 2694 2686 21 0 1 3761 2096 2401
4 2242 2681 2694 55 0 4 3760 2101 2400
```

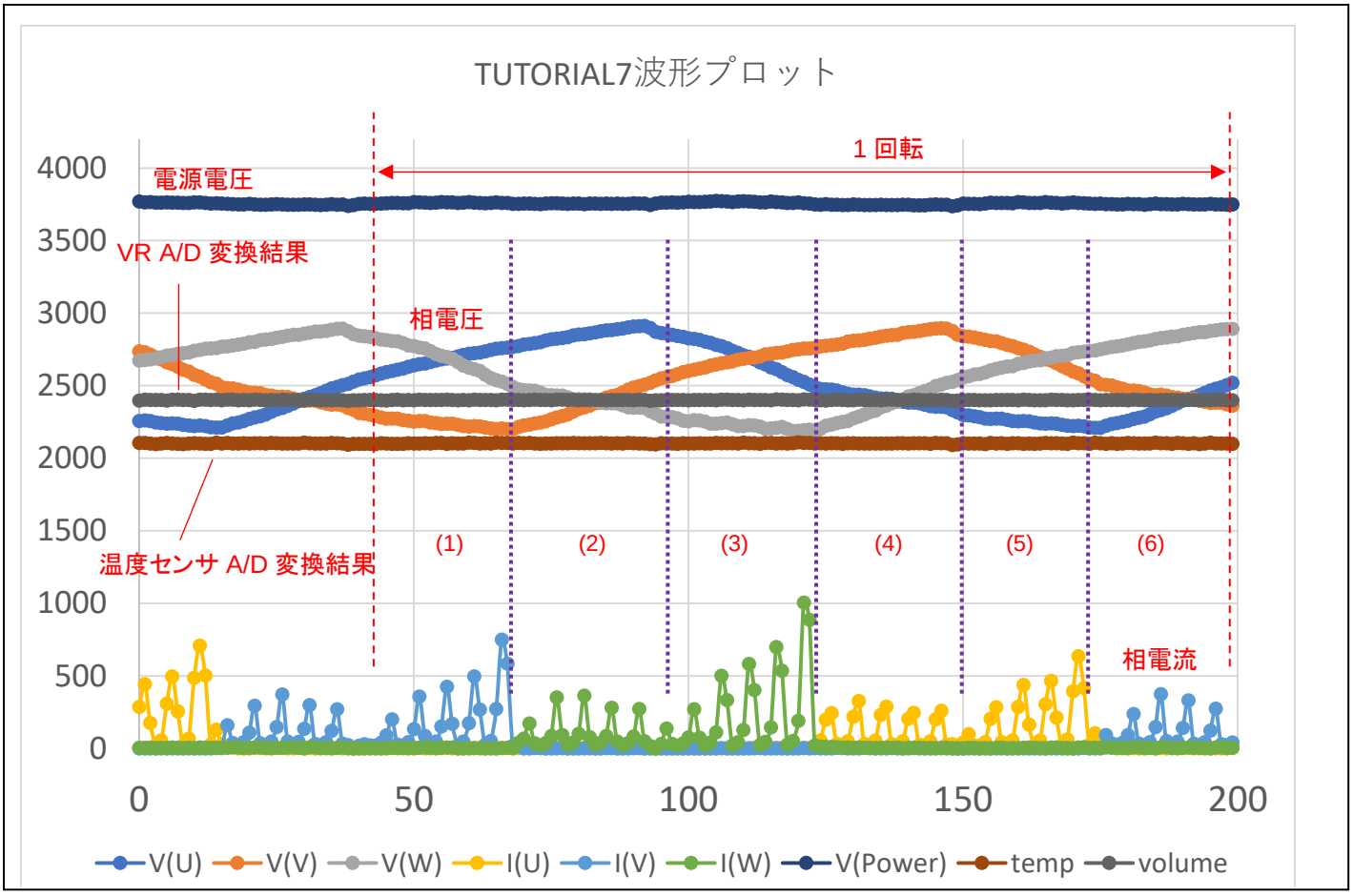
U 相電圧, V 相電圧, W 相電圧
U 相電流, V 相電流, W 相電流,
電源電圧, 温度, VR

合計 9 種のデータを出力します

データのサンプリングレートは 25kHz
(40us 間隔)です(本チュートリアル限定)

※データは、常時サンプリングされており、'A'を入力した時に、バッファリングされているデータ(200 点分)が表示されます

・シリアル端末から出力される情報を Excel でプロットした波形



波形は、端末に表示された数値をプロットしたものとなります。縦軸は、A/D 変換値となります。RA2T1 の A/D コンバータは、12bit 精度のため、値は 0~2^{12}-1(=0~4095)の範囲の値を取ります。相電圧は、モータの駆動端子を、RC でなだらかにした（LPF 通過後の）波形です。相電流は、GND 側に、電流センスの抵抗がある回路なので、I(U)がプラス方向に振れている＝他から U 相に電流が流れ込んでいる事を示しています。

本チュートリアルでは、前チュートリアル同様、1 回転で 6 回電流の向きを切り替えており、

	電流の流れる方向
(1)	U→V
(2)	U→W
(3)	V→W
(4)	V→U
(5)	W→U
(6)	W→V

に対応します。

※電流は PWM 駆動しているので、断続的に ON/OFF を繰り返しています。PWM のキャリア周波数と、A/D 変換周期(40us)の関係で電流波形の見え方は変わります。

・src¥blm¥blm_intr.c 内 CMT0 割り込み関数 blm_interrupt_cmt0

```
//回転方向:CCW
switch(g_sensor_pos[i])
{
    case 3:
        blm_drive[i] (BLM_U_V_DIRECTION);   (1)に対応
        break;
    case 2:
        blm_drive[i] (BLM_U_W_DIRECTION);   (2)に対応
        break;
    case 6:
        blm_drive[i] (BLM_V_W_DIRECTION);   (3)に対応
        break;
    case 4:
        blm_drive[i] (BLM_V_U_DIRECTION);   (4)に対応
        break;
    case 5:
        blm_drive[i] (BLM_W_U_DIRECTION);   (5)に対応
        break;
    case 1:
        blm_drive[i] (BLM_W_V_DIRECTION);   (6)に対応
        break;
    default:
        blm_drive[i] (BLM_OFF_DIRECTION);
        break;
}
```

時系列

※回転方向は反時計回りです

※制御プログラム次第で、波形は変わります

本プログラムでは、A/D 変換の生データを一旦メモリに格納し、それをシリアル端末経由で出力する処理を行っていますが、実際のモータ制御プログラムでは、得られたデータをリアルタイムに処理して、モータの制御に活用する事が出来ます。（チュートリアル（応用編）では、相電圧の変化によりモータを駆動するチュートリアルがあります。）

チュートリアル 7 までが、基本的にモータを回す上で必須となるであろうマイコンの機能を使ったチュートリアルとなります。

本チュートリアルでは、波形の観測精度をできるだけ高めるため、A/D 変換を起動するタイミングを GPT0 タイマを使い、40us 間隔としています。

・チュートリアル 7 での端子設定

→チュートリアル 6 に同じ

・チュートリアル 7 での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_gpt3	r_gpt	GPT3	QL PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	
g_external_irq4	r_icu	IRQ4	過電流検出	
g_gpt0	r_gpt	GPT0	A/D 変換起動	40us 周期

※グレーの項目は前チュートリアルから変更なし

2. チュートリアル（応用編）

チュートリアル 1~7 までの内容を踏まえ、チュートリアル 7 のプログラムに別な機能を加えたのが 2 章で説明するチュートリアル（応用編）となります。

2.1. ハードウェアでの電流方向切り替え

参照プロジェクト：RA2T1_BLMKIT_TUTORIAL_A

RA2T1 マイコンには、3 相ブラシレスモータ駆動機構が用意されており、ホールセンサの出力をマイコンの入力端子に接続する事により、出力の電流方向を切り替える事ができます。

制御方式としては、120° 制御となりますが、タイマ(GTP0)を組み合わせて 6 相の信号を PWM 制御しています。

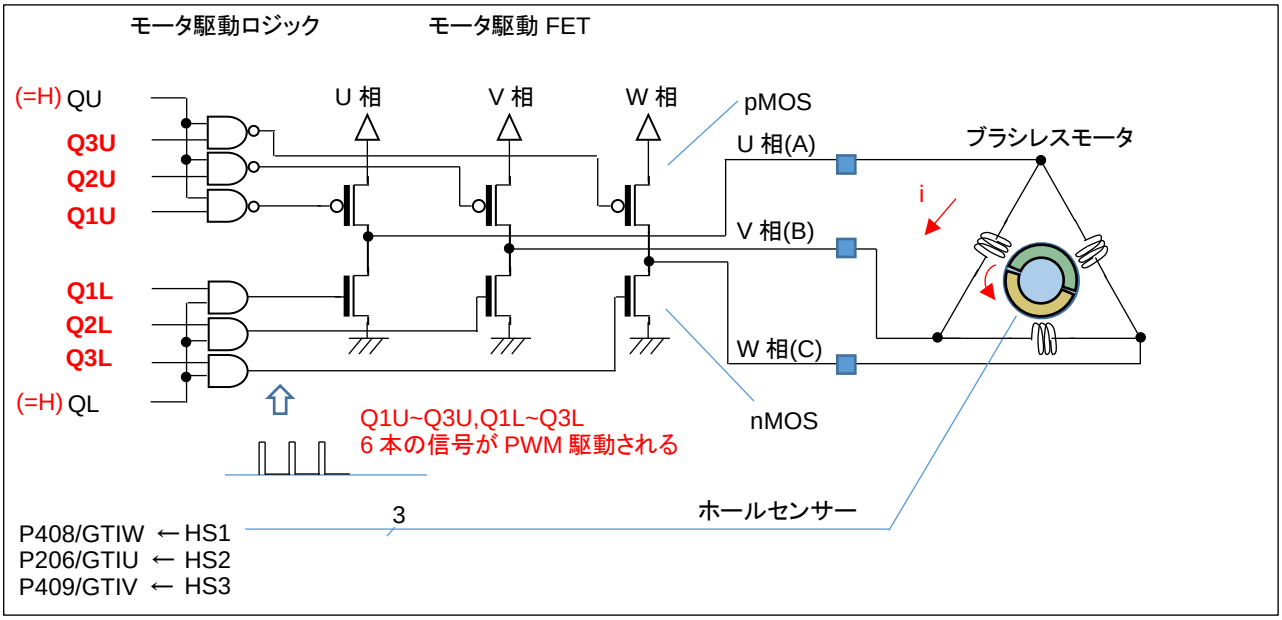
この方式の利点としては、マイコン側で電流方向の切り替えが行われる事です。（電流方向の設定をユーザプログラムで制御する必要がありません）

VR を絞った状態で、SW1 を ON にしてください。その後、徐々に VR を回してみてください。VR の回転角に応じて回転数が変わります。（VR を 1/3 ぐらいまで回さないと回転を始めませんが、一度回転が始まると、duty を絞っても回転を維持します。）

プログラムの動作は、TUTORIAL5~7 とそう変わらない動作ですが、本チュートリアルでは、P208, P407, P100~P103（モータ制御出力端子）の値をユーザプログラムで変更していない点が異なります。

TUTORIAL5~7 では、100us 毎にホールセンサの値を見て、それに応じた電流方法の切り替えをプログラム内で行っていましたが、本チュートリアルでは電流方向の切り替えを行ってる処理のプログラムコードは存在しません。（100us 毎にホールセンサの値を見ていますが、これは回転数の算出のためです。ホールセンサの値を見ている部分の処理を消しても、モータの回転には影響しません。）

ホールセンサ端子、P408/GTIW, P206/GTIU, P409/GTIV はホールセンサ入力端子(GTIU, GTIV, GTIW)として設定を行っており、端子レベルが変化した際、マイコンの機能として U,V,W のどの端子間に電流を流すか（＝P208, P407, P100~P103 の出力端子のレベル）を切り替えます。ホールセンサの 3 端子の L/H レベルの組み合わせで、出力の電流方向が決まります。



本チュートリアルでは、QU=QL=H とします。Q1U~Q3U, Q1L~Q3L の 6 本の信号がホールセンサ値(HS1~3)に応じて、アクティブ(H)になります。VR の回転角に応じて、PWM の duty は変わります。PWM は、Q1U~Q3U, Q1L~Q3L の 6 相に適用されます。（RA2T1 が持つブラシレスモータ駆動機能を使うと、GPT0 タイマの出力が、Q1U~Q3U, Q1L~Q3L の 6 本の信号を駆動する形となります。）

FSP では、

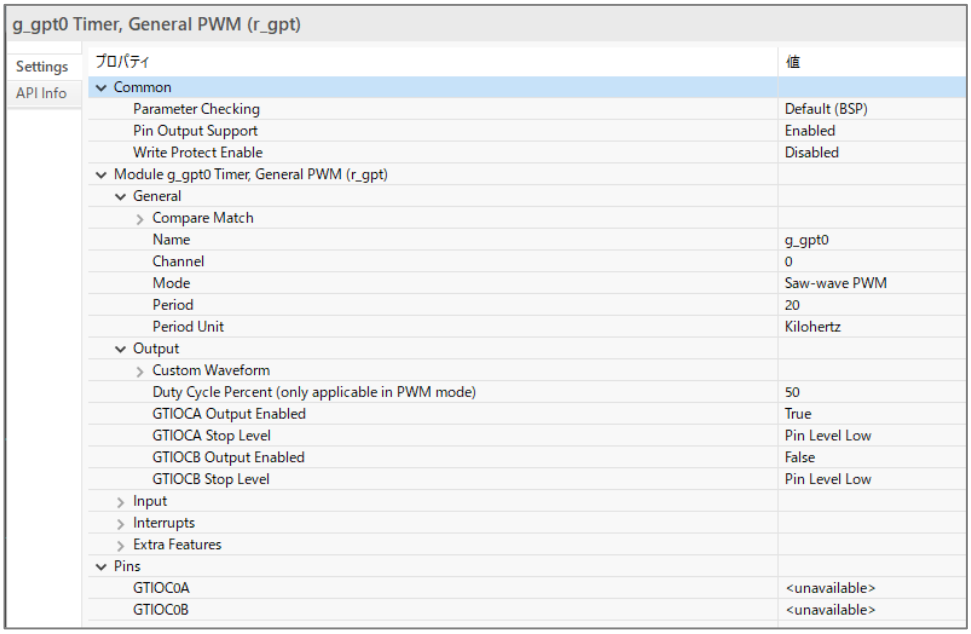
・GPT0 タイマの stack 追加

・GPT_OPS の端子設定

を行います。

・スタック追加

New stack > Timers > Timer, General PWM(r_gpt)

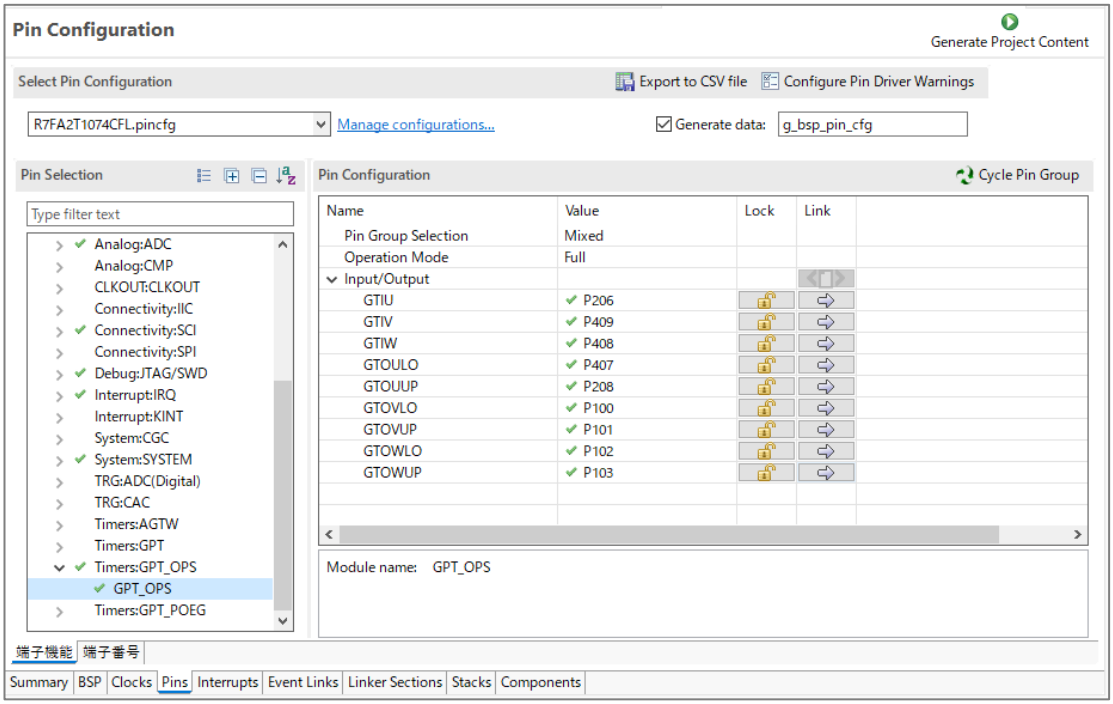


		設定値	備考
Common	Pin Output Support	Enabled	タイマで端子を駆動する場合設定必要
General	Name	g_gpt0	使用タイマ GPT0 と名称を合わせる
	Channel	0	GPT0 を使用
	Mode	Saw-wave PWM	PWM モード選択
	Period	20	タイマ周期の設定
	Period Unit	Kilohertz	単位[kHz]
Output	Duty Cycle Percent	50	プログラムで随時変更するのでここでは仮値
	GTIOCA Output Enabled	True	GTIOC0A の出力を有効化(*1)
	GTIOCA Stop Level	Pin Level Low	正極性を選択
	GTIOCB Output Enabled	False	GTIOC0B の出力は使用しない
Pins	GTIOC0A	<unavailable>	デフォルト値のままで OK(*2)
	GTIOC0B	<unavailable>	デフォルト値のままで OK

(*1)Q1U~Q3U, Q1L~Q3L の 6 端子を PWM 制御する動作となりますが、有効化するのは GTIOC0A の出力が有効となる様に設定します

(*2)GTIOC0A の出力は特定の端子に割り付けしない設定とします

・Pins タブ



GPT_OPS0 の端子設定を行います。

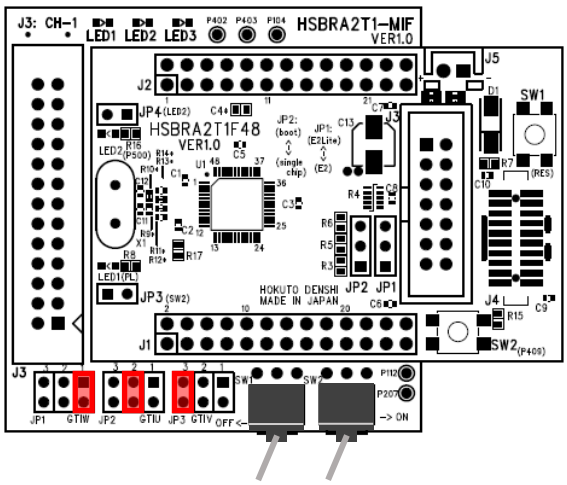
Operation Mode **Full** を選択

Input/Output	端子	モータドライバ ボード信号名	備考
GTIU	P206	HS2(*1)	ホールセンサ出力
GTIV	P409	HS3(*1)	ホールセンサ出力
GTIW	P408	HS1(*1)	ホールセンサ出力
GTOULO	P407	Q1L	U 相 L 側駆動端子
GTOUUP	P208	Q1U	U 相 H 側駆動端子
GTOVLO	P100	Q2L	V 相 L 側駆動端子
GTOVUP	P101	Q2U	V 相 H 側駆動端子
GTOWLO	P102	Q3L	W 相 L 側駆動端子
GTOWUP	P103	Q3U	W 相 H 側駆動端子

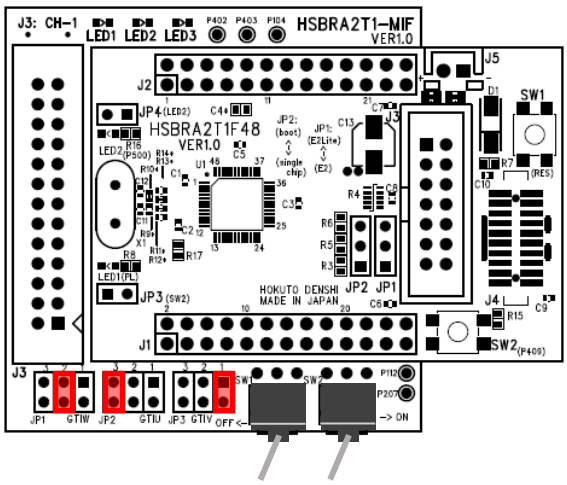
(*1)JP1~JP3 ジャンパが 1~3 設定の場合

・JP1~JP3 ジャンパの設定

デフォルト



逆回転



JP1	P408/GTIW	1(HS1)
JP2	P206/GTIU	2(HS2)
JP3	P409/GTIW	3(HS3)

2(HS2)
3(HS3)
1(HS1)

JP1~JP3 のジャンパは、1, 2, 3 を 1 つずつ選ぶ（HS1~HS3 の信号を、GTIU, GTIV, GTIW に 1:1 で接続する）必要があります。HS1~HS3 の信号を、GTIU, GTIV, GTIW に 1:1 で接続するジャンパの組み合わせは 6 通りありますが、モータが回転するのは 2 通りです。ジャンパ設定を右図の様に設定すると、モータは逆回転となります。

モータの正転・逆転はプログラム（レジスタ設定値）でも変更できますので、わざわざハード（信号線のつなぎなおし）で変更する必要はありませんが、マイコンのブラシレスモータ駆動機能（GPT_OPS 機能）の入力端子(GTIU, GTIV, GTIW)とブラシレスモータのホールセンサ出力(HS1~HS3)の関係を見られるように、JP1~JP3 のジャンパを用意しています。

なお、プログラムでのモータの回転方向変更は、後述の D コマンドで行えます。

・blm.c

```
void blm_start_ch1(void)
{
    //ブラシレスモータ CH-1 動作開始関数

    //引数
    // なし

    //戻り値
    // なし

    if (g_target_direction[BLM_CH_1] == BLM_CCW)
    {
        R_GPT_OPS->OPSCR_b.RV = 0;      //正回転
    }
    else
    {
        R_GPT_OPS->OPSCR_b.RV = 1;      //逆回転
    }

    //P400(QU) = P401(QL) = H
    R_PORT4->PCNTR3_b.POSR = 0x0003;      QU=QL=H 設定

    //Q1U-Q3U, Q1L-Q3L
    R_GPT_OPS->OPSCR_b.EN = 1;          //GPT OPS 出力有効          モータ動作開始
    (void) R_GPT_Start(&g_gpt0_ctrl);   //本チュートリアルでは GPT0 で QU1~3, QL1~3 の 6 相を PWM 駆動する
}
```

モータスタート時には、

・回転方向決定

・QU, QL 端子設定

・GPT OPS 機能の有効化

・GPT0 タイマスタート

を行っています。GPT OPS 機能は、RA2T1 マイコンが持つブラシレスモータ駆動機能です。

・起動時のメッセージ

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL A


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
LED3 : ERROR status
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display
D : rotation direction->CCW <-> rotation direction->CW (toggle)

>
Motor driver board connection check...
   CH-1 Connected.
```

'D'コマンド（回転方向変更コマンド）が追加されています。

D コマンドを入力すると、

```
Target direction -> CW
```

```
Target direction -> CCW
```

回転方向がトグルで切り替わります。

なお、回転方向はモータスタート時に反映されます。モータ回転時に変更しても、反映されるのは停止後、再スタートしたタイミングです。

D コマンドは、g_target_direction[BLM_CH_1]変数に反映され、前出の blm_start_ch1()関数内で使用されます。

マイコンのブラシレスモータ駆動機能を使うと、ほとんどマイコンのハードウェアのみでモータを回す事ができますので、マイコンがこのような機能を持っていて必要に応じて使うことができると認識して頂ければと思います。

・チュートリアル A での端子設定

端子名	役割	割り当て	備考
P000~P015	AD0~AD6,VR AD003	アナログ入力(ADC)	
P100~P103	Q2U~Q3L	周辺機能(GPT_OPS)	モータ駆動端子を GPT_OPS で駆動
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P111	*INT	IRQ4	過電流検出端子
P206	HS2(*1)	周辺機能(GPT_OPS)	GPT_OPS でホールセンサ入力端子に設定
P208	Q1U	周辺機能(GPT_OPS)	モータ駆動端子を GPT_OPS で駆動
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P400	QU	出力（初期値 L）	
P401	QL	出力（初期値 L）	汎用 I/O 端子に設定
P407	Q1L	周辺機能(GPT_OPS)	モータ駆動端子を GPT_OPS で駆動
P408, P409	HS1, HS3(*1)	周辺機能(GPT_OPS)	GPT_OPS でホールセンサ入力端子に設定
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

チュートリアル 7 では、P401 をタイマ出力端子に設定して、P100~P103, P208, P407 を汎用 I/O 出力に設定しているが、本チュートリアルでは逆（P401 は汎用出力、P100~P103, P208, P407 はタイマ（ここでは GPT_OPS）出力）としています。

・チュートリアル A での使用 stack

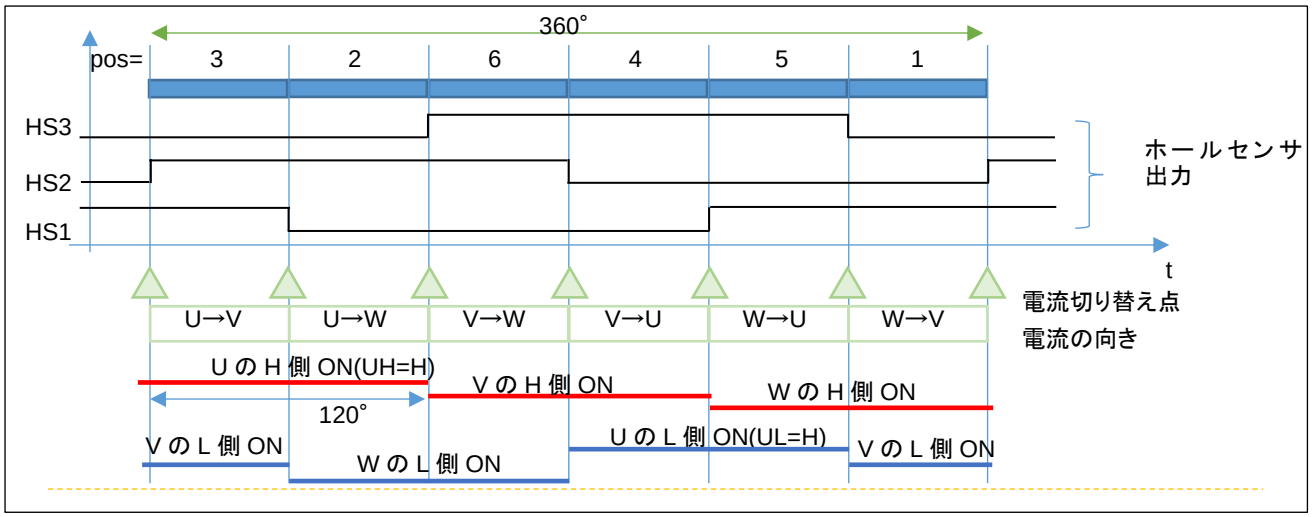
名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
~~g_gpt3~~	~~r_gpt~~	~~GPT3~~	~~QL PWM 波形生成~~	GPT3 は未使用
g_adc0	r_adc	ADC0	A/D 変換	
g_external_irq4	r_icu	IRQ4	過電流検出	
g_gpt0	r_gpt	GPT0	6 相 PWM 波形生成	20kHz

※グレーの項目はチュートリアル 7 から変更なし

2.2. 相補 PWM 信号での駆動

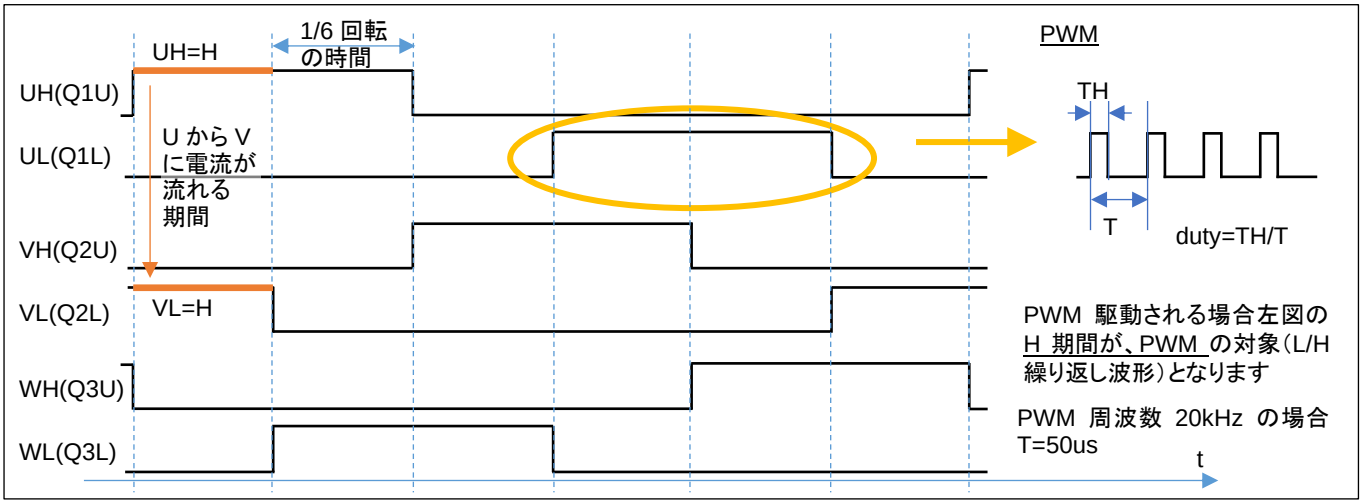
参照プロジェクト:RA2T1_BLMKIT_TUTORIAL_B

・120 度制御



～TUTORIAL7 でモータを駆動している方式は、H 側と L 側の ON 期間を 60°（1/6 周期）ずらし、H 側の ON の期間, L 側 ON の期間をそれぞれ 120°として、電流の方向を切り替えていく方式で、120°制御です。

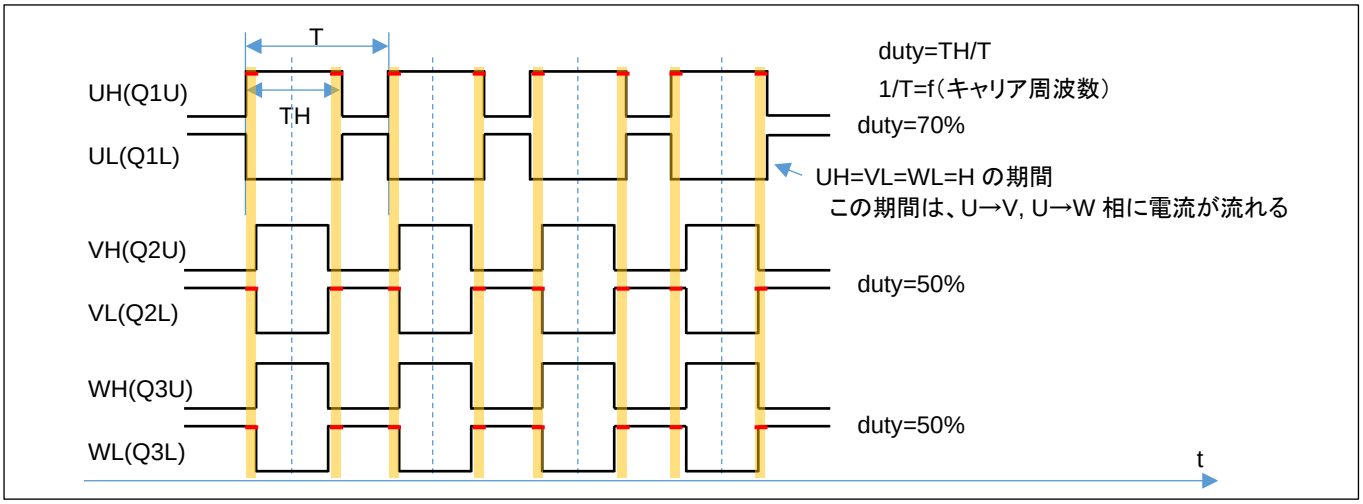
・120 度制御の駆動信号



120 度駆動では、6 本の信号線の内、H レベルになっている H 側と L 側の 1 ペアの信号線の間で電流が流れます。U 相の H 側と V 相の L 側が H レベルになっている場合は、U→V、V 相の H 側と U 相の L 側が H レベルになっている場合は、V→U の向きに電流が流れます。とある瞬間に着目すると、U→V, U→W, V→U, V→W, W→U, W→V の 6 通りある電流パスの内 1 つのパスに電流が流れているイメージです。

120 度駆動に対し、H 側と L 側の波形を反転信号で駆動する方式は、相補 PWM と呼ばれます。U 相 H 側(UH)と U 相の L 側(UL)は、常に逆相で駆動します。V 相と、W 相も同様です。

・相補 PWM 制御の駆動信号



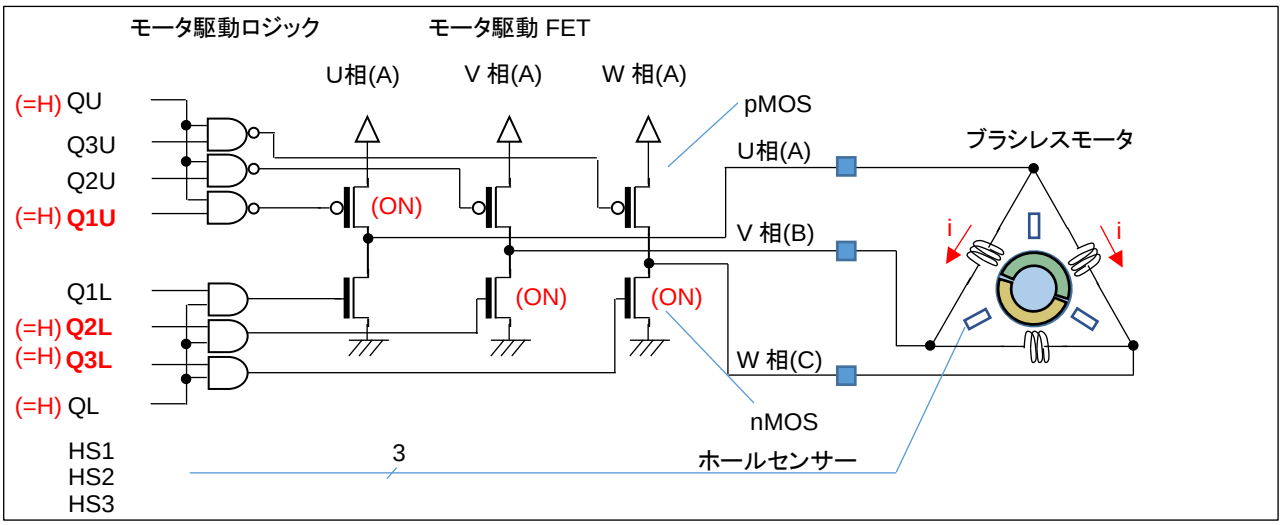
上図の相補 PWM では、

U 相　duty70%
V 相　duty50%
W 相　duty50%

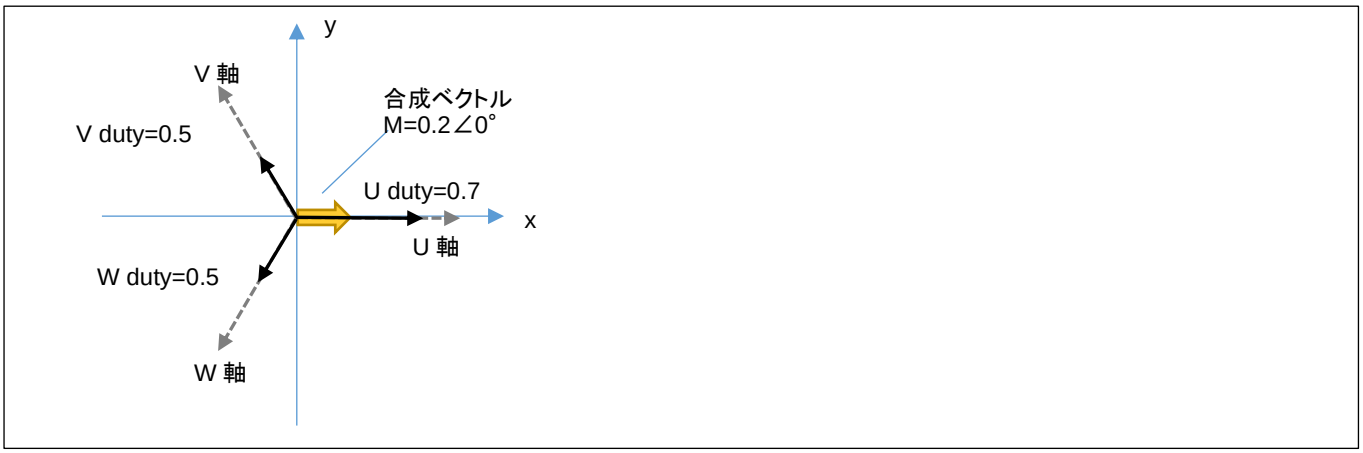
の、PWM 信号となっています。この例では、オレンジ塗りつぶしのタイミングで、（U 相の H 側と V 相の L 側と W 相の L 側が ON しており）

U 相 H　→　V 相 L
U 相 H　→　W 相 L

に電流が流れます。120°制御では、電流の流れるパスは 1 通りでしたが、相補 PWM では、基本的に 2 通りのパスがあります。（duty の高い相から duty の低い相への電流が生じる。）



ここで、duty の差分（70-50=20%）の時間だけ、U→V, U→W に電流が流れる事となります。



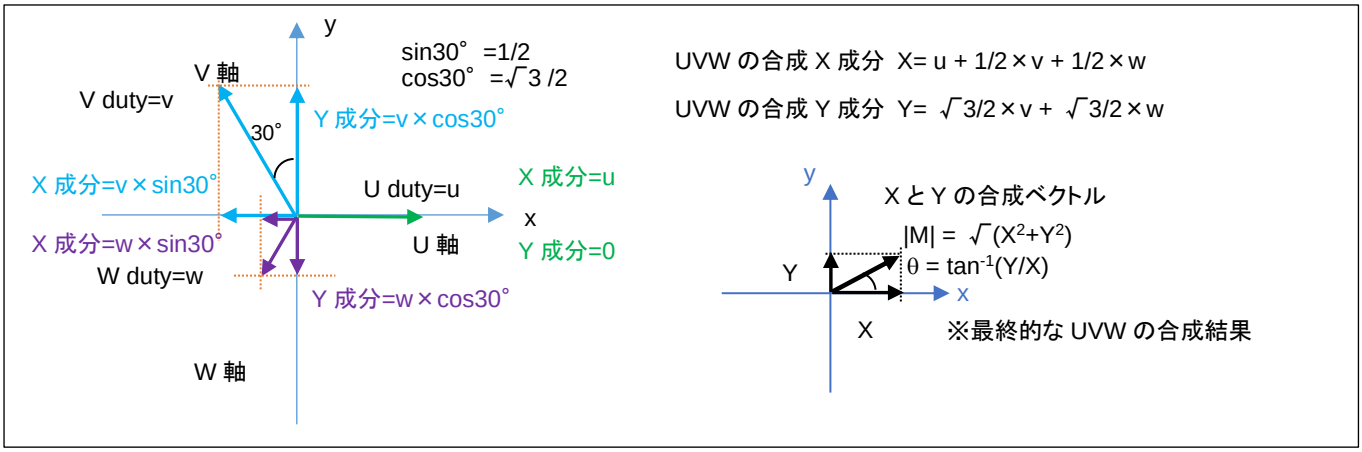
U 相に与える duty を 70%, V, W 相に与える duty を 50%とした場合、図で考えると、U,V,W 相は（上図 U,V,W 軸の方向に）120° の差分があり、それぞれの方向に 0.7, 0.5, 0.5 の強さで引っ張っているイメージです。

U=0.7, V=0.5, W=0.5 の強さで引っ張ると、この場合の合成ベクトルは、U 軸方向に 0.2 の力で引っ張る事となります。

U, V, W の 3 相の duty を調整する事により、「印加する磁界の大きさ（＝電流の大きさ）：黄色の矢印の長さ」と「どの向きに磁界を印加するか（＝UVW のどの方向に電流を流すか）：黄色の矢印の方向」を自由に設定できます。

U=0.7, V=0.5, W=0.5 の 3 本のベクトルの合成ベクトルは、U 軸方向に 0.2(20%)となります。

ここで、x 軸に U 軸を重ねる様にし、V 軸を 120°、W 軸を 240°の位置に取ります。



$$x = U - \frac{V}{2} - \frac{W}{2}$$

$$y = (V - W)\frac{\sqrt{3}}{2}$$

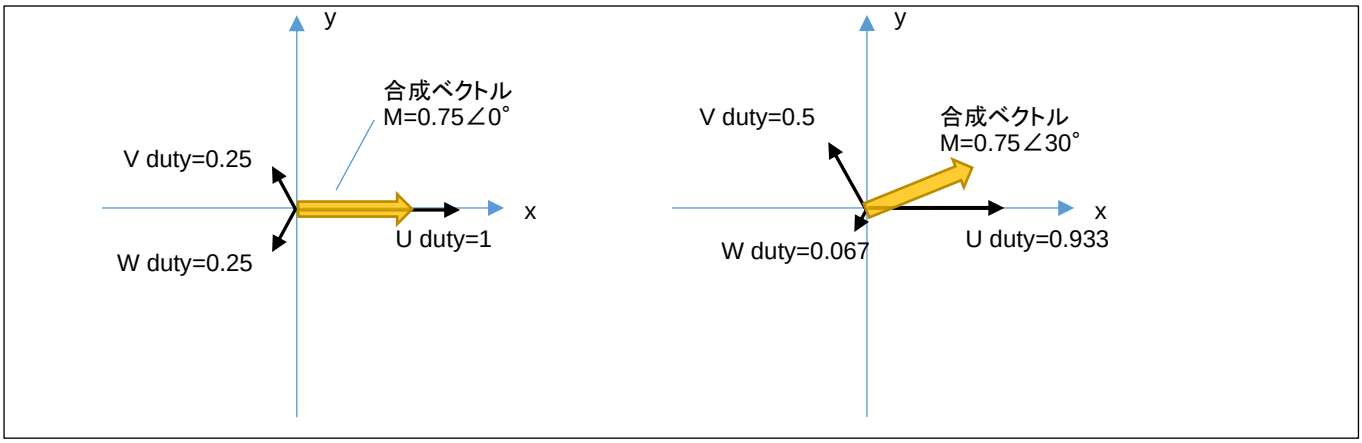
U, V, W の大きさから、x-y 軸（直交座標系）のベクトルに変換することができ、U,V,W の合成ベクトルの長さを|M|、x 軸を基準とした角度をθとすると、

$$|M| = \sqrt{x^2 + y^2}$$

$$\theta = tan^{-1}\frac{y}{x}$$

となります。

ここで、U 相の duty を 1 として、V, W 相の duty を 0.25 とすると、合成ベクトルとしては、U 軸方向に 0.75 となります。同様に V,V,W=(0.933, 0.5, 0.067)とすると、合成ベクトルは強さは 0.75 で角度は U 軸から 30°ずれた位置となります。



120 度制御の場合は、60°単位での切り替え（1 回転で 6 段階、電流の流れる方向＝磁界の方向は 6 パターンしか存在しない）となりますが、相補 PWM では磁界の向きを任意の角度で印加する事が可能です。

120 度制御：1 回転で電流の流れる方向 6 パターンの切り替え
相補 PWM：1 回転の間で合成ベクトルの向きを任意の刻み、角度で切り替えていく事が可能

本チュートリアルのプログラムの動作としては以下となります。

VR を絞った状態で SW を ON にしてください。その後、VR を上げてみてください。どこかでモータが回りだすタイミングがあるはずです。

プログラム動作としては、TUTORIAL4 と同じです。回転数は 1667rpm 固定で、VR により duty を変化させていくプログラムとなっています。（モータの制御に、ホールセンサの値は使っていません。）

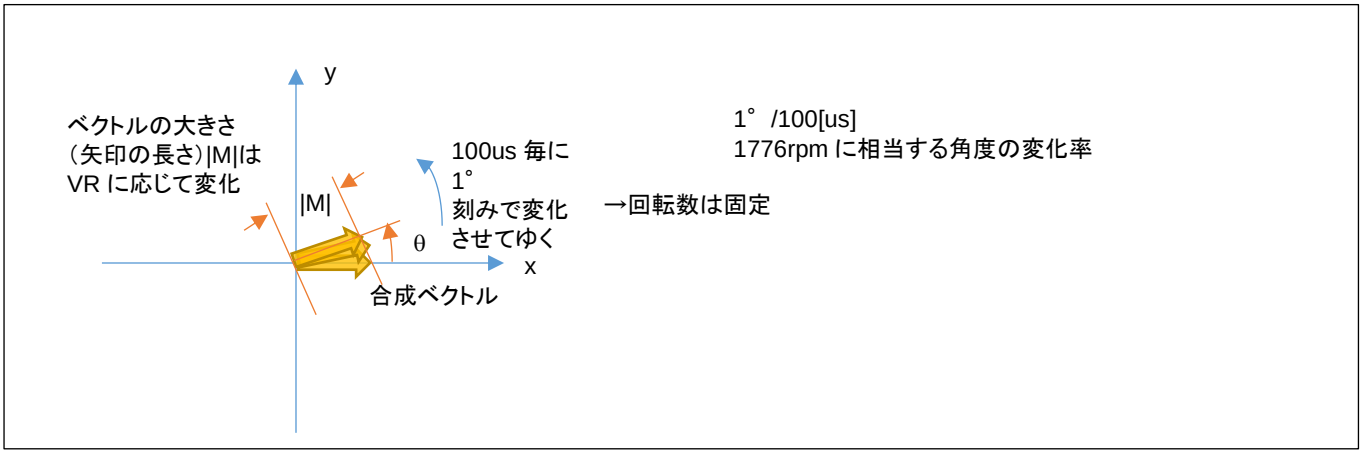
合成ベクトルの大きさ(|M|)=duty は、VR の回転角度に応じて変化します。合成ベクトルの角度(θ)は、100us 毎に動かしていきます。

1667rpm / 60 = 27.8 回転/s
1/27.8 = 36ms …1 回転（360°で 36ms かかる）
360° / (36ms / 100us) = 1°

本チュートリアルでは、100us(AGT0)の周期割り込みで、磁界の印加角度（合成ベクトルの角度）を変えていく処理としているので、100us 毎に印加角度を 1°ずつ変化させていけば良い（＝1667rpm の速度で回転するように印加する磁界を動かす）事となります。
→1 回転(360°)で、360 段階の細かさで磁界の印加方向を切り替える

回転数は固定で duty を VR のツマミの回転に応じて変化させる手法は、TUTORIAL4 と同様です、

duty（合成ベクトルの大きさ|M|）が小さいときはモータは回転せず、大きすぎても効率が下がります（このあたりの関係も TUTORIAL4 と同様です）。最適な duty のとき、モータはスムーズに回ります。



ここで、モータを制御する側からすると、

・合成ベクトルの大きさ|M|
・合成ベクトルの角度θ

を与えたいのですが、マイコン側からすると、

・U 相の duty(0~100%)
・V 相の duty(0~100%)
・W 相の duty(0~100%)

の 3 値を与える事となります。

|M|, θを与えた際、U,V,W のそれぞれの強さに分解する方式は一通りではないのですが、本チュートリアルのプログラムでのデフォルトは正弦波駆動（UVW の duty がそれぞれ、正弦波状で変化）としています。

U(duty) = (|M| ・cos θ) /2 + 0.5
V(duty) = (|M| ・cos (θ-120°)) /2 + 0.5
W(duty) = (|M| ・cos (θ-240°)) /2 + 0.5

（0-1 の範囲に正規化）

|M|, θの値から上式で、U,V,W のそれぞれの duty 値に変換しています。

－合成ベクトルの UVW への分解に関して－

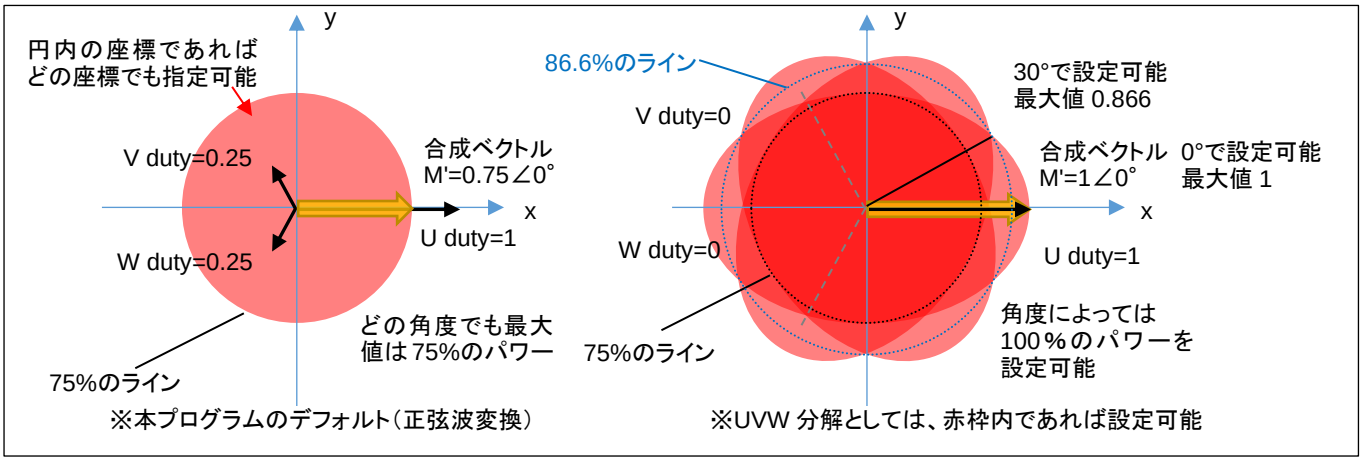
上記手法、計算式（正弦波駆動）を用いると、|M|=1, θ=0 を与えて各相の duty を計算すると、
(U, V, W) = (1.0, 0.25, 0.25)
となり、この合成ベクトルは、
0.75∠0°
となります。0°（U 軸）方向に最大限のパワーを与えたい場合、結果的には 75%のパワーで 0° 方向に力を印加する事となります。

同様に、|M|=1, θ=30° での各相の duty は、
(U, V, W) = (0.933, 0.5, 0.067)
となり、この合成ベクトルは
|M'|∠θ = 0.75∠30°
です。30° 方向に最大限のパワーを与えたい場合でも上記同様、結果的には 75%(=|M'|)のパワーとなります。
（入力として与えた、|M|が|M'|に変換されます。|M'|=0.75×|M|です。）

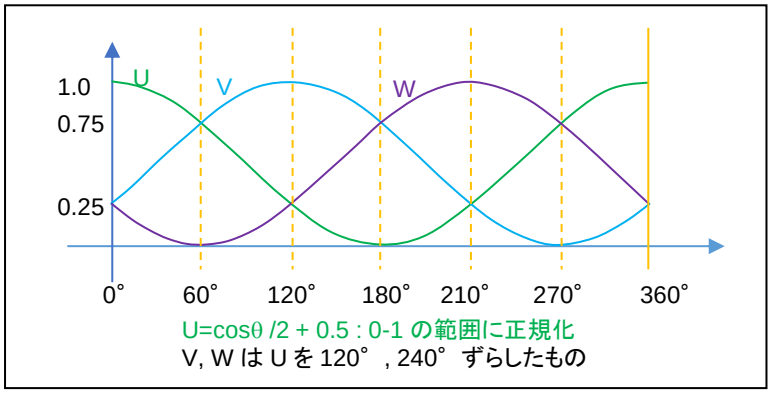
本手法で、UVW の分解を計算した場合、どの角度においても、最大値は 0.75 になります。（0.75 に制限されます）

例えば、3 相の duty を以下の様にすれば、
(U, V, W) = (1.0, 0.0, 0.0) → 1.0∠0°
となります。（|M'| = |M|とする事が可能です）

正弦波駆動ではない別な分解方法を用いると、0° 方向に 100%のパワーを与える事も可能です。但し、全角度に対して、100%のパワーを与える事は、UVW（0°，120°，240°）の 3 本のベクトルの合成では不可能です。（100%のパワーを与えることのできる角度は、0°，120°，240° とその反対側の 60°，180°，300° のみ。）
30° では、(U, V, W) = (1.0, 0.5, 0.0) → 0.866∠30°、86.6%が理論上の最大パワーとなります。



・本プログラムの UVW 分解（正弦波駆動）



横軸は角度（0~360° のモータ 1 回転）、縦軸は UVW の 3 相に分解したそれぞれの相に与える duty 比(0~1)を示しています

本プログラムの UVW 分解は、各相を正弦波で連続的に変化させる手法で、最大のパワーがどの角度においても、75%に制限されるが、UVW の変化に連続性があり、三角関数の計算は必要であるが、計算式は単純です。cosθの値は-1~1 の範囲の値を取りますので、PWM duty（設定可能なのは 0~100%, 0~1）に対応させるために、2 で割り 0.5 を加算して、0~1 の範囲にシフト（1 に正規化）させています。

duty=100%の設定を行った場合は、UVW の各相の duty の変化は、上記グラフの通り。duty=50%の設定を行った場合は、上記のグラフ×0.5 の値（UVW の各相の duty が 0-50%の範囲で変化）となります。

例えば、60° の方向に duty=100%の設定を行った場合は、
(U,V,W)=(0.75, 0.75, 0) →0.75∠60° （120° 制御で印加可能な最大パワーを基準にすると 75%）
となります。

60° の方向に duty=50%の設定を行った場合は、
(U,V,W)=(0.375, 0.375, 0) →0.375∠60°
となります。

設定した duty が 50%の場合、各相の duty は 0-50%の範囲で変化して、変化率は回転数に応じた値となります。例えば、2,000rpm の場合、1 回転が 30ms なので、30ms で 360° 分の変化（0→0.25→0.5→0.25→0 と変化）をします。本チュートリアルの制御周期は 100us なので、100us 毎に 1.2° ずつの変化となります。

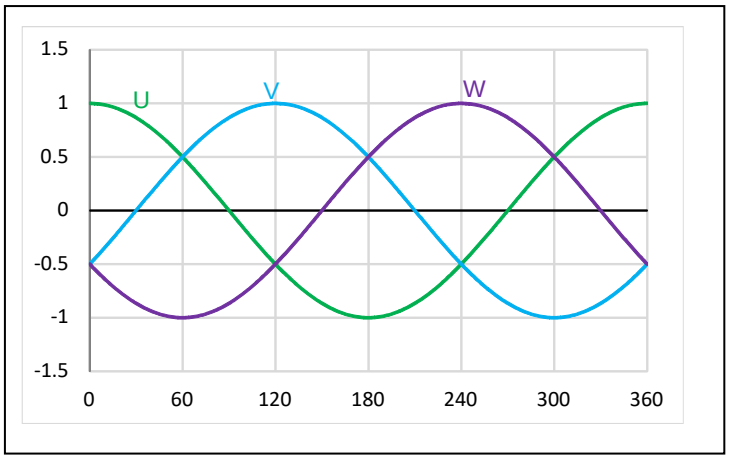
相補 PWM の場合、全体の duty は変化しなくても、UVW 各相の duty は常に変化している動作となります。

正弦波駆動は考え方や計算式はシンプルですが、印加可能なパワーが低いという欠点もありますので、その他の変換方法に関しても考えてみる事とします。

（印加可能なパワーが低い→120° 駆動等の別な駆動方式と同じ電源電圧を与えた場合、モータの回転数や出力パワーが小さくなってしまう。電源のエネルギーをモータ駆動のエネルギーに変換する効率が悪いという事を意味します。）

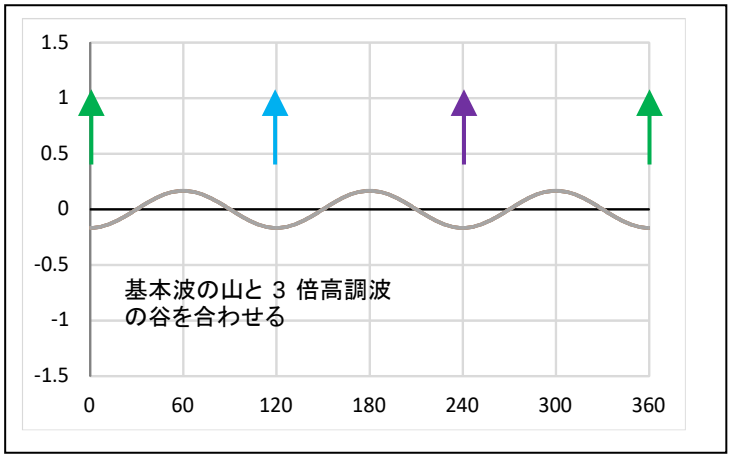
・正弦波駆動＋3 倍高調波の重畳

(1)正弦波駆動による分解（正規化前）



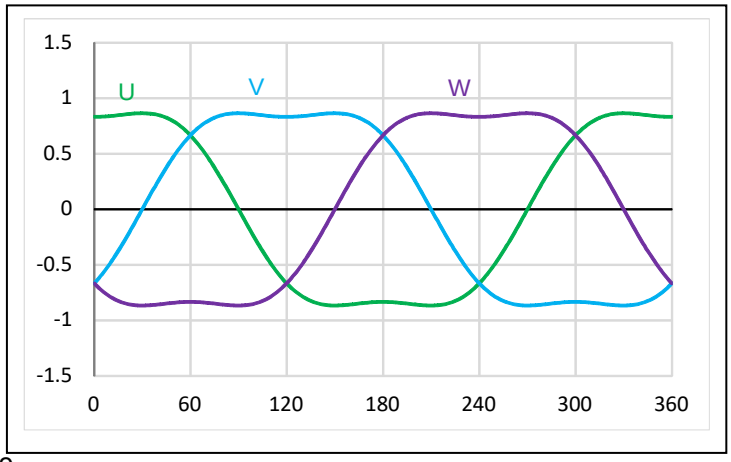
(1)は正弦波による分解の正規化前の UVW（120 度位相をずらした 3 相の単純な正弦波）の値です。

(2)3 倍高調波（振幅 1/6）



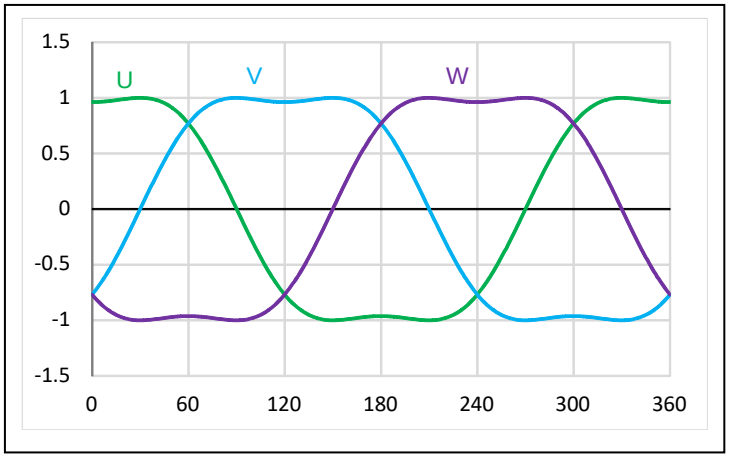
(2)は、周波数を 3 倍に、振幅を 1/6 にした正弦波の波形となります。
（sin で計算する場合は U/V/W 相と同位相、cos で計算する場合は逆位相）

(3)基本波＋3 倍高調波の重畳（(1)+(2)）



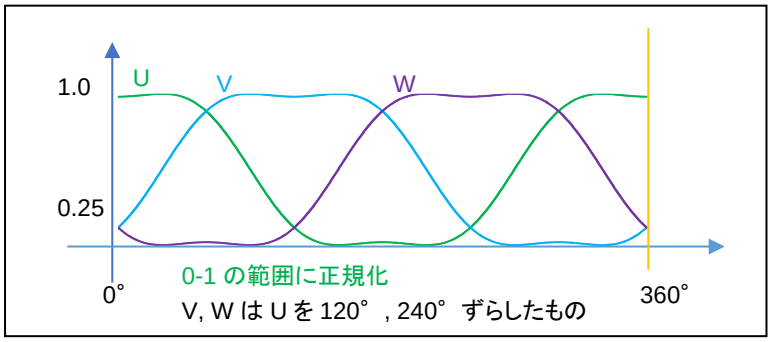
(1)と(2)を単純に足し合わせると、上記の様な波形となります。ここで、(1)の波形は-1～1 の値を取りましたが、(3)の波形は、(1)の波形のピークが抑えられている波形となります。具体的には、最大値が√3/2=0.866 になっています（-0.866～0.866 の値を取る）。3 倍高調波の重畳により、波形のピークが抑えられ、結果的に、全体を伸長する余力（0.866 を 1 に引き延ばす余地）が生じる結果となります。

(4)基本波＋3 倍高調波の重畳のスカラー倍（(3)×2/√3）



(4)は、単純に(3)をスカラー倍（×2/√3, 1.155 倍）したものです。

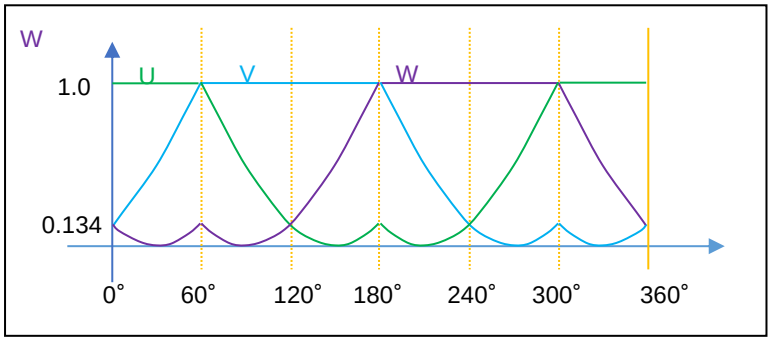
duty としては、0～1（このグラフでは正規化前なので、-1～1 の範囲）の設定が可能です。設定可能な duty をフルに活用する目的で(3)の波形を-1~1 の範囲に引き延ばす処理を行った結果です。



最終的に正規化した後の値（実際に U,V,W 相に設定する duty 値）は、上記の様になります（0-1 の範囲）。

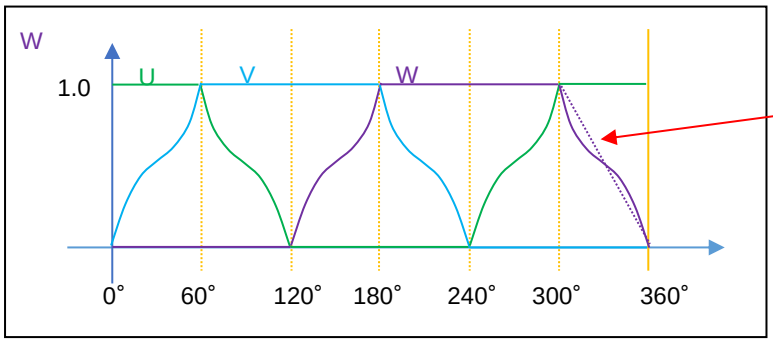
この、正弦波＋3 倍高調波駆動により、設定可能な duty は 0~360° どの角度でも、86.6%(=√3/2)となります。単純な正弦波駆動に対し、2/√3=1.155(+15.5%)設定可能な duty の引き上げが可能となります。

・UVW 分解（別バージョン 1）



どの角度でも設定可能な最大のパワーは、86.6%(=√3/2)であると考えます。例えばですが、上記の様な UVW のカーブとすると（かなり変則的なカーブですが）、設定可能な最大パワーが正弦波駆動の場合の 75%→86.6%に増加します。（但し領域で分けて三角関数と、1 固定で UVW 値を計算する必要があります。）（印加可能な duty としては「正弦波駆動＋3 倍高調波の重畳」と同じ）

・UVW 分解（別バージョン 2）



計算を簡単にするためにこのカーブを直線近似するという「別バージョン 2'」も考えられます
（直線近似した場合、入力の角度に対し、UVW 変換した結果の角度は最大 1.2° 程度ずれますが、それ程大きなずれにはなりません）

3 倍高調波を使用したバージョンや、上記別バージョン 1 でも、モータに 100%の電力を与える事はできません。（120 度制御では、刻みは 60° であるが、最大 100%の電力を与える事ができる。）前ページの右図の赤枠の外周をなぞるように UVW 分解（上記波形）を行うと、0,60,120,180,240,300° の位置では 100%の電力を与える事が可能です。

別バージョン 2 では、

1∠0° →1∠0°（0° 方向には 100%のパワーで引っ張る）

1∠15° →0.897∠15°（15° 方向には 89.7%のパワーになってしまう）

1∠30° →0.866∠30°（30° 方向には 86.6%のパワーになってしまう）角度により最大パワーが異なる変換です。

※「正弦波＋3 倍高調波」と「別バージョン 1」「別バージョン 2」は大体似たようなカーブとなっていると思います
相補 PWM でモータにより多くの電力を与える場合、大体このようなカーブとなる変換であると思います

別バージョン 2 では、三角関数の計算が必要になりますが、図のカーブの部分を直線近似しても傾向は変わらないのでは？というのが、「別バージョン 2'」です。別バージョン 2'では、U,V,W=1 の領域と、直線の領域で考えれば良く、UVW 分解の計算が単純化できます。

・UVW 変換方法のまとめ

入力（プログラムで与える|M|と角度）	UVW 分解の結果（|M'|と角度）			
	・正弦波変換（全角度に 75%のパワー）※本チュートリアルのデフォルト設定	・正弦波＋3 倍高調波変換・別バージョン 1（全角度に 86.6%のパワー）	・別バージョン 2（角度により上限を変える）	・別バージョン 2'（別バージョン 2 の直線近似版）
1∠0°	0.75∠0°	0.866∠0°	1∠0	1∠0°
1∠10°	0.75∠10°	0.866∠10°	0.922∠10	0.928∠**8.9°（*1)**
1∠20°	0.75∠20°	0.866∠20°	0.879∠20°	0.882∠**19.1°（*1)**
1∠30°	0.75∠30°	0.866∠30°	0.866∠30	0.866∠30°
0.1∠0°	0.075∠0°	0.0866∠0°	0.1∠0°	0.1∠0°
0.1∠30°	0.075∠30°（*2)	0.0866∠30°（*2)	**0.0866∠30°（*2)(*3)**	**0.0866∠30°（*2)(*3)**

(*1)別バージョン 2'のみ入力角度と実際に印加される角度に誤差が生じます（最大 1.2° 程度）

別バージョン 2 は角度により最大パワーが異なる（120 度駆動と PWM のハイブリッド？）の様な方式です。

(*2)この部分の変換は、0.1∠30° にする事も可能です。（考え方次第です）角度により設定上限が異なるだけで、上限に達しない範囲では入力の duty 値を変えずに変換するという考え方も取り得ます。

(*3)上限は、10° で 0.928, 15° で 0.897, 20° で 0.882, 30° で 0.866）

－合成ベクトルの UVW への分解に関して(2)－

正弦波駆動の UVW 分解で、

U=(cosθ×|M|) / 2 + 0.5　(1)

（V, W はθに 120°, 240°を加えて算出）

/ 2 + 0.5 は cos（-1~1 を取る）を 0~1（各相の duty として設定できる範囲）に変換する操作です（1 に正規化）。(1)式では、ベクトルの大きさ|M|を乗算した後で、正規化を行っています。（本プログラムのデフォルトで採用している計算式）

U={(cosθ×**1**) / 2 + 0.5}　×|M|　(2)

ここで、U の値は|M|=1 で計算し、正規化後に|M|を乗算するとどうなるか考えてみます。

|M|=1（入力の duty=100%）の時は、(1)と(2)で計算結果は変わりません。

|M|=0.5, θ=30°のケースで考えてみます。

(1)式で U, V, W を計算すると、(U, V, W) = (0.717, 0.5, 0.283)　(1')
(2)式で U, V, W を計算すると、(U, V, W) = (0.467, 0.25, 0.033)　(2')

となります。

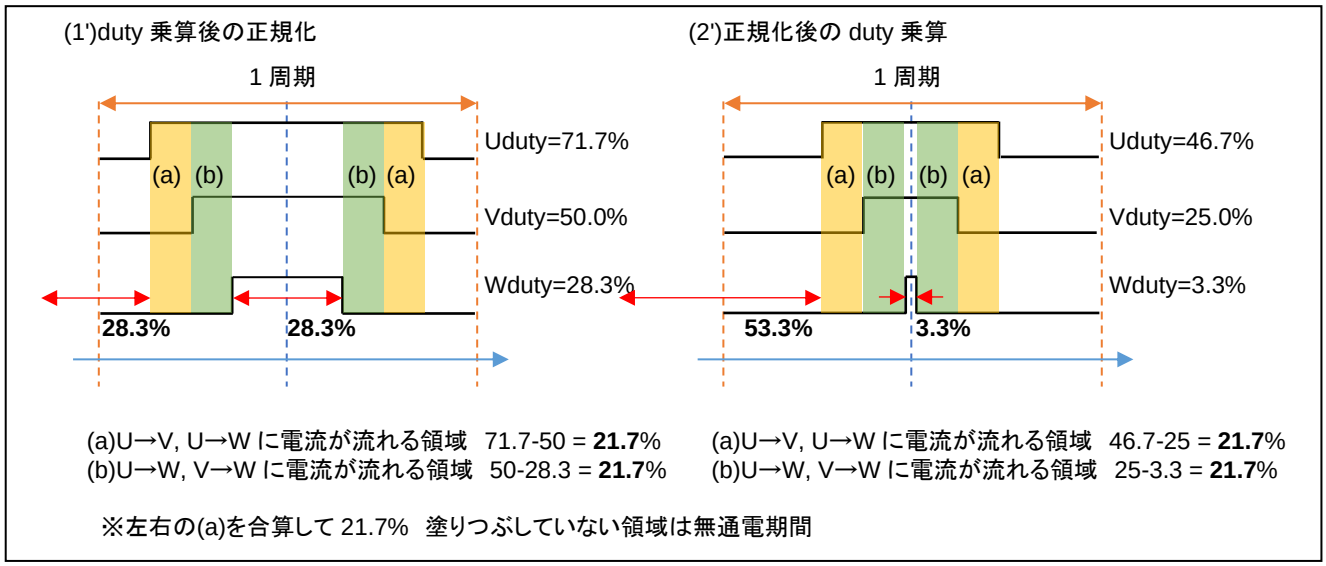
(1')と(2')で、UVW 各相の値は異なりますが、合成結果は

|M'| = 0.375
θ = 30°

で変わりません。

角度は、（当たり前ですが）入力の通り。ベクトルの大きさは、入力の 75%になるので、0.5×0.75=0.375 です。

この、(1')と(2')の違いは？



1 周期における電流の流れるタイミングが異なります。2 相間に流れる電流の大きさ（積算電流）は同じです。

※上図は、1 周期（キャリア周波数 20kHz の場合は、50us）を示しており、同じ波形が繰り返されるイメージです（100us 毎に角度を変えていくので、各相の duty は徐々に変化しますが、（ミクロな観点で見ると）基本的には 1 周期の左右に（ほぼ）同じ 1 周期の波形が来るイメージです。）

(1')(2')で電流が流れない時間（図の白い部分）は、
(1') : 100-71.7 + 28.3 = 56.6%
(2') : 100-46.7 + 3.3 = 56.6%
と同じですが、（赤矢印）
(1') : 28.3%と 28.3%で分割
(2') : 53.3%と 3.3%で分割
となります。(2')の場合は、(b)と(b)の間がほとんど空かない代わりに(a)と次の 1 周期の(a)の時間が空き、電流の流れない時間が長く続きます（この例だと 53.3%）。

この、(1')と(2')の違いが、モータのトルクや回転の滑らかさ、消費電力等に影響を与える事は考えられます。

しかし、基本的には、モータにどの程度の電力を与えるか、どの方向に引っ張るかという事が重要ですので、UVW 分解法における無電流期間の分布はそれ程大きな問題にはならないと考えます。

（PWM のキャリア周波数を変える事でも無電流期間と電流を流す期間の分布は変わります。…周波数を上げると全体が圧縮される。キャリア周波数の変更と、(1')と(2')どちらの式とするかは同じような影響かと思います。）

・通電期間の時系列(1') [duty を乗算した後に正規化]

	n-1 周期	←			1 周期			→	n+1 周期
	無通電	無通電	(a)	(b)	無通電	(a)	(b)	無通電	無通電
(1')	14.15%	14.15%	10.85%	10.85%	28.3%	10.85%	10.85%	14.15%	14.15%
	28.3%		21.7%		28.3%	21.7%		28.3%	
	→t								

周期の 28.3%無通電、21.7%通電の繰り返し。

・通電期間の時系列(2') [後で duty を乗算]

	n-1 周期	←			1 周期			→	n+1 周期
	無通電	無通電	(a)	(b)	無通電	(a)	(b)	無通電	無通電
(2')	26.65%	26.65%	10.85%	10.85%	3.3%	10.85%	10.85%	26.65%	26.65%
	53.3%		21.7%		3.3%	21.7%		53.3%	
	→t								

周期の 53.3%無通電、21.7%通電、3.3%無通電、21.7%通電、53.3%無通電の繰り返し。

※塗りつぶしが通電期間

通電時間と無通電期間のバランスが取れるのが(1')の方式。短い無通電期間が中央（＝三角波 PWM の頂点）に来るのが(2')の方式です。

なお、30° で duty を変える場合、

	(1')	(2')
0.1∠30°	(0.543, **0.5**, 0.457)	(0.093, 0.05, 0.007)
0.2∠30°	(0.587, **0.5**, 0.413)	(0.187, 0.1, 0.013)
0.3∠30°	(0.630, **0.5**, 0.370)	(0.280, 0.15, 0.02)

(1)式を使った場合、V 相の duty は常に 50%となります。この場合、0.5∠30° の場合同様に、無通電の期間が 1/2 分割されて分布する事となります。（バランスの関係は、上記 0.5∠30° と同様になります。）

本チュートリアル、サンプルプログラムのデフォルトでは、(1)式を使い(1')になる様な分解法ですが、duty, θを UVW 相に分解する方法は無数に存在します（そもそも正弦波変換なのか、他の変換方法を採るのか。正規化前に duty を乗じるのか、正規化後に乗じるのか、です）。どの手法が正解なのかは、一概にはなんとも言えません。

本キットでは、デフォルトが最大パワーが 75%に制限される UVW 分解法としてますが、この場合 120 度制御に比べて最高回転数が劣ります。最高回転数を稼ぐ事を目的とするのであれば、120 度制御とするか、相補 PWM では UVW の分解方法がポイントになるかと考えます。（UVW 分解方法は、モータ制御の目的に応じ色々なバリエーションが考えられます。）

※サンプルプログラム内には、
「正弦波駆動」(デフォルト) `blm_angle_to_uvw_duty_sin_faa` (1)
(2)式を使った正規化後に duty を乗じる方式(正弦波駆動) `blm_angle_to_uvw_duty_sin_post_faa` (2)
「正弦波+3 倍高調波」 `blm_angle_to_uvw_duty_sin_3harmonic_faa` (3)
「正弦波+3 倍高調波, duty を正規化後に乗じる」 `blm_angle_to_uvw_duty_sin_3harmonic_post_faa` (4)
「別バージョン 1」(全角度に 86.6%のパワー) `blm_angle_to_uvw_duty1_faa` (5)
「別バージョン 2」(60° 刻みで 100%のパワー) `blm_angle_to_uvw_duty2_faa` (6)
「別バージョン 2'」(別バージョン 2 の直線近似版] `blm_angle_to_uvw_duty2x_faa` (7)
の合計 7 種類の UVW 分解関数を用意しています。
(blm.c, blm_dutyset()関数内で、g_blm_angle_to_uvw_duty()関数を呼び出している部分を変更)
(サンプルプログラムでは、UVW 分解法の動作の違いを見ることができます。)

・blm.c

```
    //UVW 分解方法：(1)の正弦波駆動を指定 ※7 行の内いずれかのコメントアウトを外す（関数ポインタなのでプログラムの実行中に切り替えても OK）
    blm_angle_to_uvw_duty = blm_angle_to_uvw_duty_sin;//(1)正弦波駆動（デフォルト）
    //blm_angle_to_uvw_duty = blm_angle_to_uvw_duty_sin_post;//(2)正弦波駆動, duty を後から乗算
    //blm_angle_to_uvw_duty = blm_angle_to_uvw_duty_sin_3harmonic;//(3)正弦波 3 倍高調波重畳
    //blm_angle_to_uvw_duty = blm_angle_to_uvw_duty_sin_3harmonic_post;//(4)正弦波 3 倍高調波重畳, duty を後から乗算
    //blm_angle_to_uvw_duty = blm_angle_to_uvw_duty1;//(5)別バージョン 1, 全方向に 86.6%のパワー
    //blm_angle_to_uvw_duty = blm_angle_to_uvw_duty2;//(6)別バージョン 2, 60°で割り切れる角度では 100%のパワー
    //blm_angle_to_uvw_duty = blm_angle_to_uvw_duty2x;//(7)別バージョン 2' 別バージョン 2 を直線近似したもの
```

blm_angle_to_uvw_duty()関数が UVW 分解で呼び出される関数名(関数ポインタ)です。

(関数ポインタ) = (関数の実体)
blm_angle_to_uvw_duty = blm_angle_to_uvw_duty_sin_3harmonic;
を実行すると、blm_angle_to_uvw_duty()関数の実体が、blm_angle_to_uvw_duty_sin_3harmonic()になります。

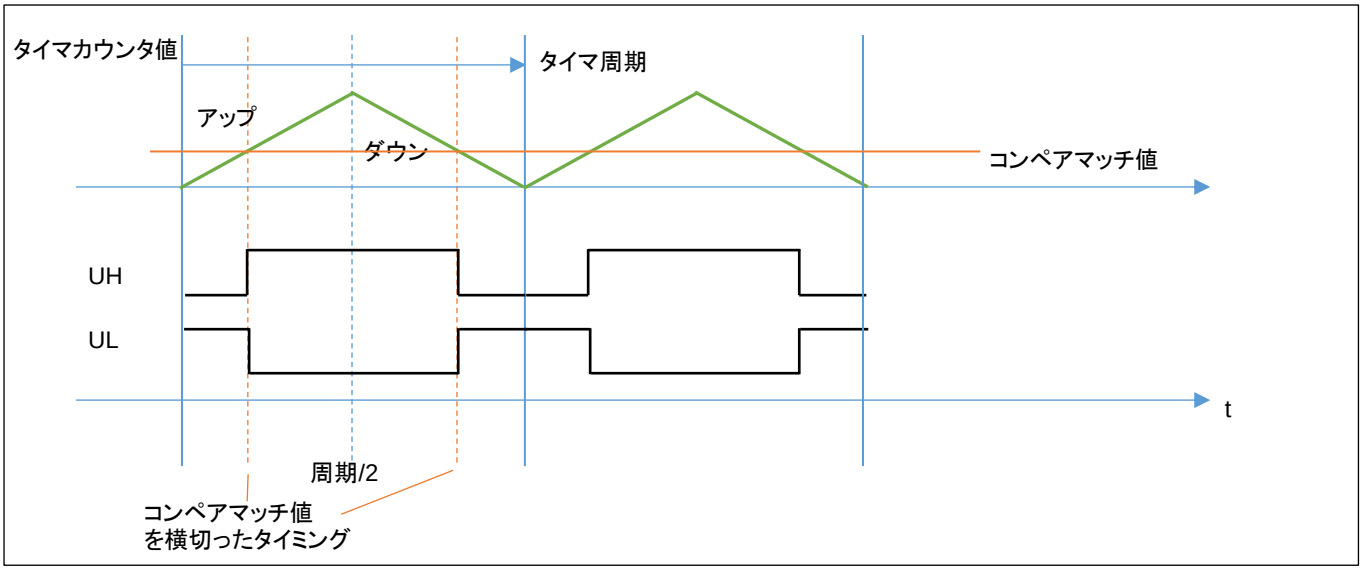
初期化時(blm_init()内で)指定しても良いですし、任意のタイミング(モータ駆動中でも)で UVW 変換関数の実体を変更する事が可能です。

・起動時のメッセージ

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL B


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
LED3 : ERROR status
VR -> duty(0-40%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display
D : rotation direction->CCW <-> rotation direction->CW (toggle)
1 : UVW calc -> sine
2 : UVW calc -> sine(2)
3 : UVW calc -> sine + 3harmonic
4 : UVW calc -> sine + 3harmonic(2)
5 : UVW calc -> another version
6 : UVW calc -> another version(100% power)
7 : UVW calc -> another version(100% power)(2)

>
Motor driver board connection check...
  CH-1 Connected.
```

本チュートリアルでは、キーボードからのコマンド入力により、UVW 分解法を 7 種類の内から変更可能です。モータ回転時にも変更は可能です。UVW 分解法と duty の関係（起動時のデフォルトの 1（正弦波：75%の変換）から 3（3 倍高調波：86.6%の変換）や 6（角度によっては 100%のパワー）に変更すると、同じ回転数でも duty を小さく設定できるはずなので確かめてみてください。

・相補 PWM 波形をカウンタを使って生成する方法



相補 PWM を構成するタイマは、周期の 1/2 まではアップカウント、周期の 1/2 から 1 周期まではダウンカウントとなる動作です。設定したコンペマッチ値を横切った際、出力は反転します。周期は固定とし、コンペアマッチ値を、U, V, W でそれぞれ別な値とすることで、U, V, W それぞれの相で別個の duty の矩形波を得ることができます。

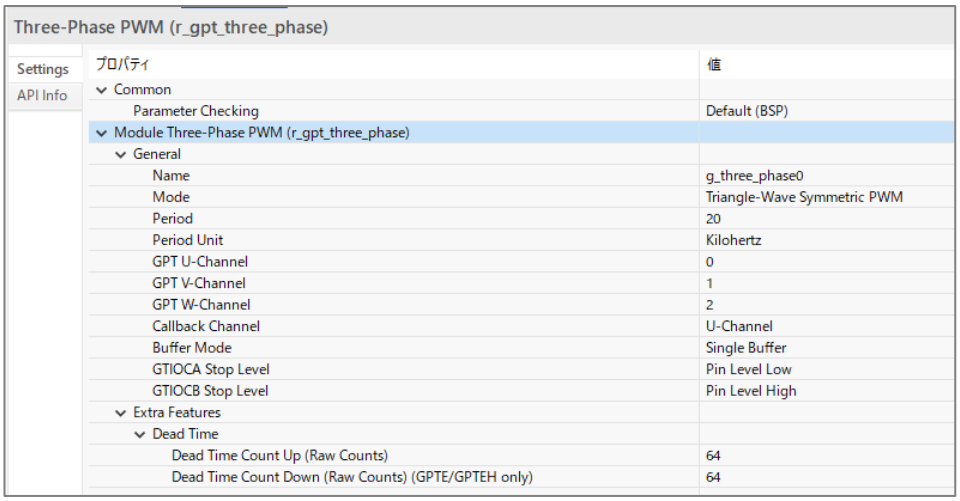
U_コンペアマッチ値 = (1.0 - U(duty)) × 周期/2 （U(duty)は 1 に正規化済みの値）
※V, W も同様

	割り当て(CH-1)
U 相	GPT0
V 相	GPT1
W 相	GPT2

※3 相の相補 PWM の生成には、タイマを 3 つ使用します

FSP では、以下の stack を追加します。（3 つのタイマがセットになっている stack です）

New Stack > Timers > Three-Phase PWM(r_gpt_three_phase)



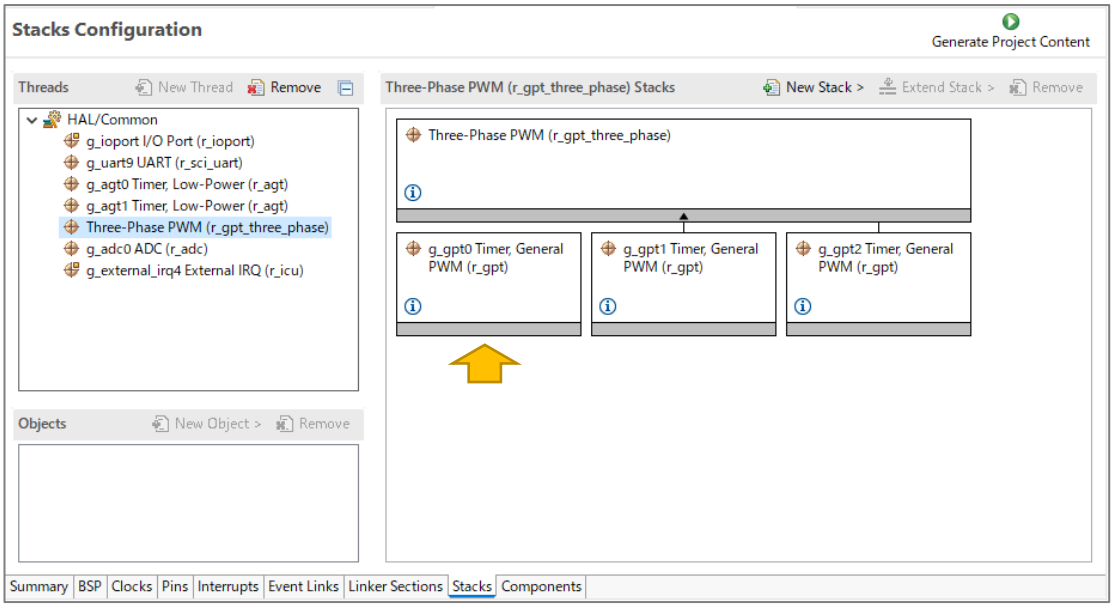
		設定値	備考
General	Name	g_three_phase0	デフォルトのまま
	Period	20	タイマ周期の設定
	Period Unit	Kilohertz	単位[kHz]
	GPT U-Channel	0	U 相制御に GPT0 を使用する設定
	GPT V-Channel	1	V 相制御に GPT1 を使用する設定
	GPT W-Channel	2	W 相制御に GPT2 を使用する設定
	GTIOCA Stop Level	Pin Level Low	初期値 L（QL 側端子は L から始める）
	GTIOCB Stop Level	Pin Level High	初期値 H（QU 側端子は H から始める）
Extra Features			
Dead Time	Dead Time Count Up	64	PCLKD=64MHz で 64 カウント(=1us)
	Dead Time Count Down	64	PCLKD=64MHz で 64 カウント(=1us)

3 相相補 PWM の Name は、g_three_phase0 として、上記設定とします。モータの CH あたり 1 つの Three-Phase PWM(r_gpt_three_phase)（3 相 PWM の stack）追加が必要です。

使用する GPT タイマは、GPT0/GPT1/GPT2 を選択していますが、マイコンの端子とモータドライバボードの接続関係で決まっています。((Q1U, Q1L) (Q2U, Q2L) (Q3U, Q3L)につながっている端子で決まります）

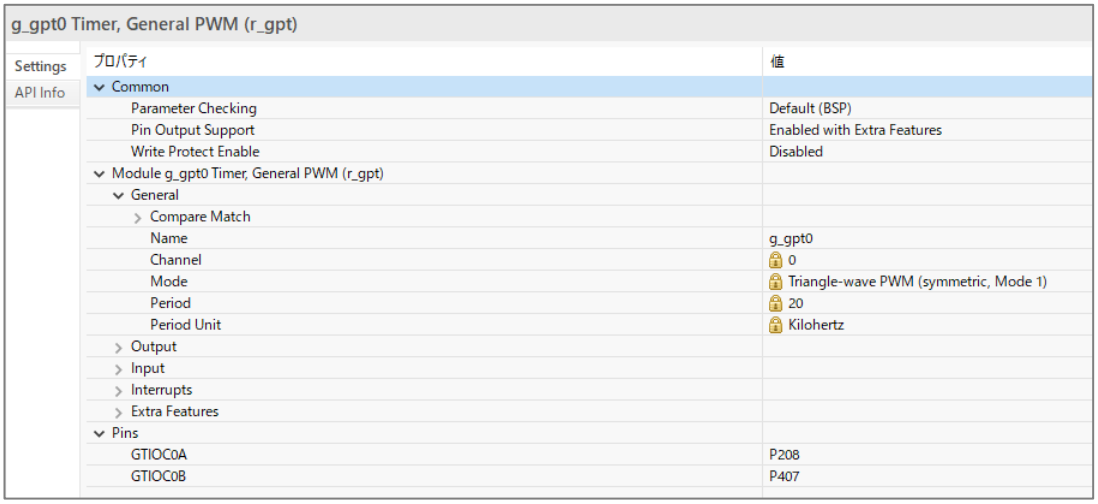
Dead Time は後述のデッドタイムの指定で、ここでは 1us の設定としています。（時間を短くし過ぎると、モータ駆動 FET 部で電流が流れて電力が無駄になります。長くし過ぎると長い無通電時間が生じて効率が下がります。）

Three-Phase PWM(r_gpt_three_phase)の stack は、階層構造となっており、Three-Phase PWM の stack（箱）の下に、GPT タイマの箱がぶら下がる形となります。



Three-Phase PWM の下には、3 つの「Timer, General PWM(r_gpt)」がぶら下がる形です。
（Three-Phase PWM の下には、GPT0, GPT1, GPT2 がぶら下がる。）
（r_gpt は、デフォルトでは、g_timer0, g_timer1, ...という Name で追加されます。デフォルトから変更しなくても問題ありませんが、ここではタイマの名称に合わせて、g_gpt0, g_gpt1, gpt2 に変更しています。）

次に、Three-Phase PWM の下にぶら下がる、g_gpt0 の設定を示します。（図の矢印のボックス）



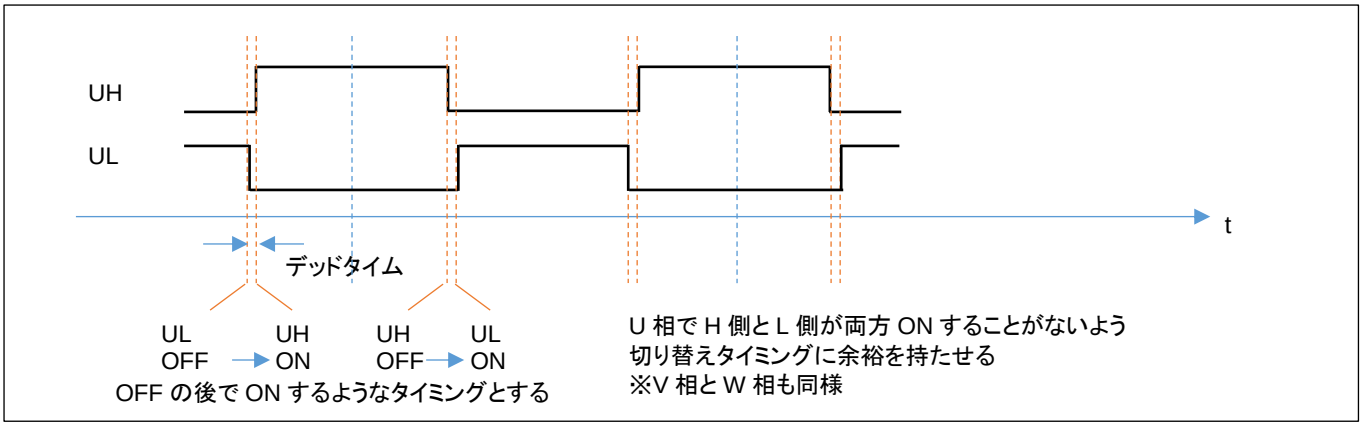
・g_gpt0

		設定値	備考
Common	Pin Output Support	Enabled with Extra Features	変更要
Module...			
General	Name	g_gpt0	使用タイマ名 GPT0 に合わせて変更（任意）
Pins	GTIOC0A	P208	Pins タブで設定
	GTIOC0B	P407	Pins タブで設定

Three-Phase PWM の下にぶら下がっている、r_gpt のプロパティで
Pin Output Support **Enabled with Extra Features**
に変更してください。

（※r_gpt は、トータルで 3 個使用していますが、そのうちのいずれか 1 箇所で変更すれば全てに反映されます。）
（設定項目で鍵マークのアイコンが付いている項目は、上位の stack(Three-Phase PWM)の設定値が反映されており、g_gpt の階層では変更できません。変更する場合は上位の stack で変更してください。）

相補 PWM では、デッドタイムを設定しています。
これは、相補 PWM で、Posi（=UH 側信号）と Nega（=UL 側信号）を、単に反転させる訳ではなく、時間差を付けて切り替える制御となります。



UH の信号と UL の信号が同時に ON すると、電流はモータのコイルではなく、出力回路部の MOS FET のみに流れます（電源が pMOS－nMOS を経由して GND にショート）ので、その様な状態を避けるための制御となります。

3 相の相補 PWM 駆動を行う場合は、使用するタイマは、1 個のモータあたり 3 つ必要ですが、マイコンから見るとタイマはハードウェア（CPU リソースを消費することなく、一定タイミングで切り替わる）ですので、マイコン側のプログラムとしては、適切なタイミングで PWM の各相 duty（UVW の 3 相の duty には、ベクトルの大きさ|M|と角度θの情報が含まれます）を設定すれば良い事となります。

ブラシレスモータの制御としては、単純な 120 度制御と、相補 PWM を用いたベクトル制御の 2 通りがメジャーな制御方式です。本チュートリアルのプログラムは、ホールセンサは回転数の算出に用いているだけで、回転の制御には使っていません。次の TUTORIAL_B2 では、相補 PWM とホールセンサを組み合わせて、モータを制御しています。（TUTORIAL5（回転制御にホールセンサを使用）に対応した相補 PWM 版が TUTORIAL_B2 となります。）

※デッドタイムを指定すると、設定 duty には誤差が生じますが、本プログラムではデッドタイム指定に伴う誤差の補正は行っていません。

・シリアル端末から出力される情報

```
                      CH-1
Motor Driver Board    : Connect
Active               :    o
UVW calculation method : (1)
target speed([rpm])   :  1670
target direction      :   CCW
rotation speed([rpm]) :  1680
Temperature(A/D value) :  2166
Temperature(degree)   :    28
VR(A/D value)         :  2086
duty[%]               :  19.9
```

(1)~(7)のどの UVW 分解法を選んでいるかが表示されます。

・起動時のメッセージ

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL B


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
LED3 : ERROR status
VR -> duty(0-40%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display
D : rotation direction->CCW <-> rotation direction->CW (toggle)
(後略)
```

	起動時のデフォルト	D コマンドで切り換え
	回転方向反時計回り(CCW)	回転方向時計回り(CW)

本チュートリアルでは、D コマンドで回転方向が変わります。回転方向は

CCW : 100us 毎に印加角度を 1° 増やす

CW : 100us 毎に印加角度を 1° 減らす

という違いです。(SW を ON にしたタイミングで、設定されている回転方向が適用されます。回転中に D コマンドを入力しても回転方向は変わりません。)

(D コマンドの動作は、チュートリアル A と同じですが、本チュートリアルでは D コマンドは、100us 毎にベクトルの角度を加算するか、減算するかの動作に反映されます。)

・チュートリアル B での端子設定

端子名	役割	割り当て	備考
P000~P015	AD0~AD6,VR AD003	アナログ入力(ADC)	
P100, P101	Q2L, Q2U	周辺機能(GPT1)	モータ駆動端子を相補 PWM で駆動
P102, P103	Q3L, Q3U	周辺機能(GPT2)	モータ駆動端子を相補 PWM で駆動
P108	デバッガ接続	周辺機能(SWDIO)	
P109	UART 通信（送信）	周辺機能(SCI9)	TXD9 として設定
P110	UART 通信（受信）	周辺機能(SCI9)	RXD9 として設定
P111	*INT	IRQ4	過電流検出端子
P206	HS2(*1)	入力（プルアップ）	ホールセンサ入力として使用
P208	Q1U	周辺機能(GPT0)	モータ駆動端子を相補 PWM で駆動
P300	デバッガ接続	周辺機能(SWCLK)	
P301	SW1	入力	ボード上でプルアップ
P302	SW2	入力	ボード上でプルアップ
P400	QU	出力（初期値 L）	
P401	QL	出力（初期値 L）	汎用 I/O 端子に設定
P407	Q1L	周辺機能(GPT0)	モータ駆動端子を相補 PWM で駆動
P408, P409	HS1, HS3(*1)	入力（プルアップ）	ホールセンサ入力として使用
P500	LED	出力（初期値 L）	マイコンボード上の LED2（チュートリアルでは未使用）
P913	LED1	出力（初期値 H）	初期状態で LED は消灯
P914	LED2	出力（初期値 H）	初期状態で LED は消灯
P915	LED3	出力（初期値 H）	初期状態で LED は消灯

モータ駆動端子（6 本）は、GPT タイマ 3 組。QU, QL 端子は、汎用 I/O で駆動する設定です。

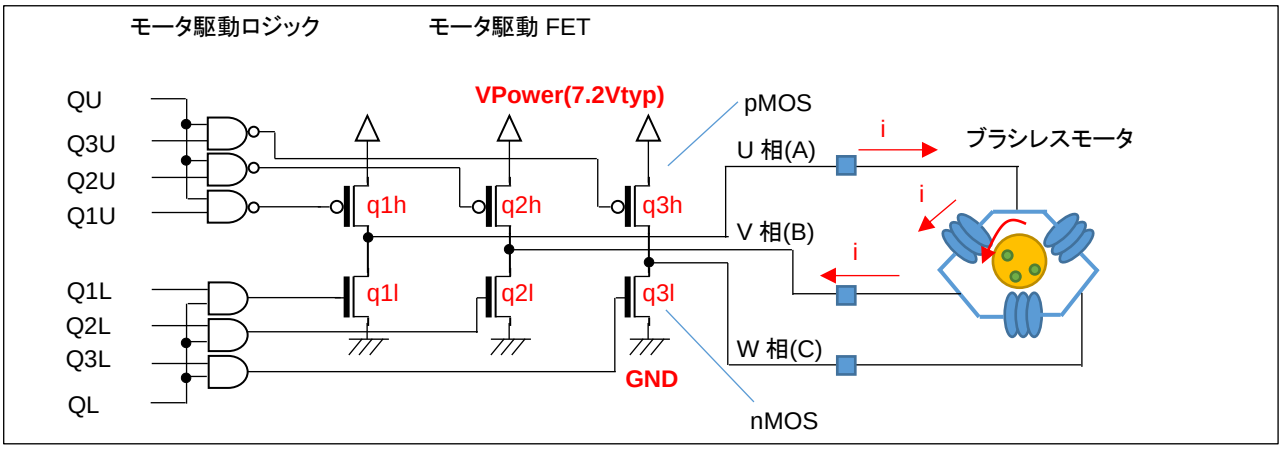
・チュートリアル B での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_three_phase0	r_gpt_three_phase	GPT0/1/2	相補 PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	
g_external_irq4	r_icu	IRQ4	過電流検出	

※グレーの項目はチュートリアル 7 から変更なし

モータ駆動用タイマとしては以下の様になっています。

チュートリアル	使用タイマ	備考
~チュートリアル 7	GPT3	のこぎり波 PWM モードで使用、1 相 PWM 制御
チュートリアル A	GPT0	1 つのタイマで 6 本の信号を PWM 制御（PWM は 1 相）
チュートリアル B	GPT0 GPT1 GPT2	3 つのタイマに別々な値を設定して、相補 PWM 制御



チュートリアル 7 までは、GPT を使い GTIOC3B を QL に接続。1 つの PWM 信号で、L 側(q1l, q2l, q3l)の 3 つの FET を PWM 駆動する方式です。（q1h, q2h, q3h の H 側の FET は、120° 単位でいずれかの FET が ON）

チュートリアル A では、QU=QL=H 固定。1 つのタイマで Q1L~Q3L, Q1U~Q3U の 6 本の信号が、アクティブ時に PWM 駆動される方式です。6 本の端子に同じ duty が適用されます。（PWM としては 1 相です）

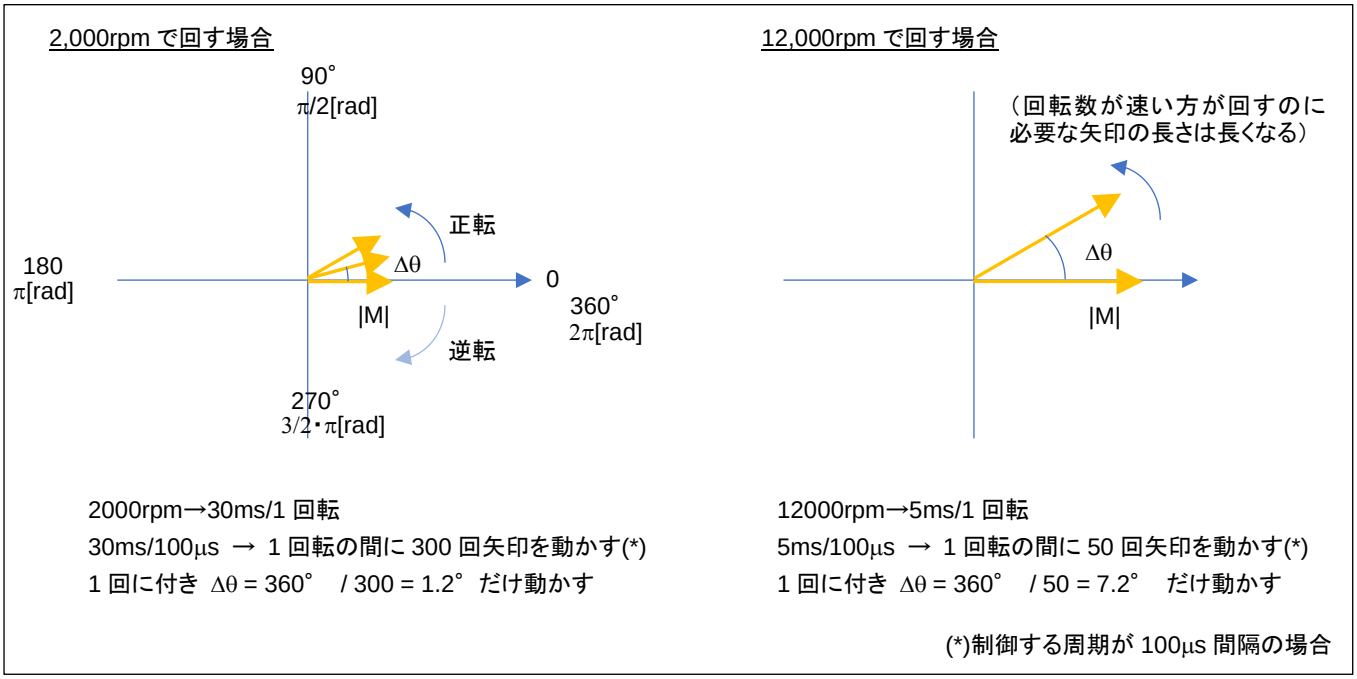
チュートリアル B では、QU=QL=H 固定。Q1U, Q2U, Q3U, Q1L, Q2L, Q3L の 6 本の信号が PWM 駆動されます。U 相 duty=(Q1U, Q1L), V 相 duty=(Q2U, Q2L), W 相 duty=(Q3U, Q3L)となり、duty 値は 3 値別々の値を取ります。Q1U と Q1L は、デッドタイムを持つので U 相 duty 値にデッドタイムを持たせた値（Q1U(PWM1)と Q1L(PWM1')は微妙にずれた duty 値）です。（結果的に、6 値の duty で 6 本の信号が駆動されます。）

	Q1U	Q2U	Q3U	Q1L	Q2L	Q3L	QU	QL	備考
～チュートリアル 7	120 度	120 度	120 度	120 度	120 度	120 度	H 固定	PWM	PWM duty は VR に対応
チュートリアル A	120 度 ＋ PWM	120 度 ＋ PWM	120 度 ＋ PWM	120 度 ＋ PWM	120 度 ＋ PWM	120 度 ＋ PWM	H 固定	H 固定	PWM duty は VR に対応 Q1U~Q3L の PWM duty は同値
チュートリアル B	PWM1	PWM2	PWM3	PWM1'	PWM2'	PWM3'	H 固定	H 固定	PWM duty は 3 値 （厳密には 6 値） duty のベクトルの長さは VR に対応 ベクトルの角度は一定値を一定時間でインクリメント

120 度は、1 回転の内 120°（1/3 の期間）ON(=H)、残りの 240° は OFF(=L)
120 度＋PWM は、1 回転の内 120°（1/3 の期間）PWM 波形、残りの 240° は OFF(=L)
PWM1 と PWM1'はデッドタイムの分のみずれがある、基本は、PWM1, PWM2, PWM3 の 3 値

本節で登場した、相補 PWM 制御に関してまとめると以下の様になります。

・相補 PWM



・矢印をΔθずつ動かしていく（矢印→モータに印加する磁界の方向）

・矢印を 1 回転させる＝モータが 1 回転

・θをどちらに動かすかでモータの回転方向を変えられる

・|M|の値は回転数に応じて制御する必要がある（|M|の値→モータに与える電力に相当）

・|M|, θの値は、前出の UVW 分解法にて 3 値の duty 値に変換する

→プログラム的には 3 値の duty 値を取り扱う、デッドタイム付与はマイコンの機能で行うのでプログラム的に 6 値の duty を計算する必要はない

ー浮動小数点の使用に関してー

本チュートリアルでは、duty 値の画面表示の際など限定的に浮動小数点の演算を使用しています。

100us 周期で実行されるモータの制御には、浮動小数点の演算は使用していません。RA2T1 マイコンは、FPU（浮動小数点ユニット）を持たないため、小数点を伴う演算は全て整数演算を使用したソフトウェアライブラリで処理されます。そのため、整数演算に対し、浮動小数点演算は非常に重たい処理となります。
通常であれば、浮動小数点で計算した方が素直な処理も、本チュートリアルのプログラムでは整数演算に置き換えているので、非常に処理内容が判り難い書き方になっています。

相補 PWM では、ある瞬間に
0.5∠30°
の磁界をモータに印加する場合、各相には
U 相→duty=0.717
V 相→duty=0.5
W 相→duty=0.283
の duty を与えますが（UVW 分解法に関しての考え方は本節を参照ください）、この値はタイマの周期値が 1000 の場合、最終的に欲しい値は
U 相タイマレジスタ値=1000×(1-0.717) = **283**
V 相タイマレジスタ値=1000×(1-0.5) = **500**
W 相タイマレジスタ値=1000×(1-0.283) = **717**
で、(283, 500, 717)となります。

一般的に UVW 分解の計算を行う場合、計算の過程で三角関数や×√3/2 等の計算を行うので、計算の途中では浮動小数点の演算を多用しますが、最終的な結果は整数値となります。（浮動小数点の演算をハードウェアで行う事ができる、RX や他の RA のキットでは浮動小数点の演算を使用しています。）

例えば、UVW 分解法に 3 倍高調波を使用した下式をプログラムコードで示します。

$$\left(\frac{\left(cos\theta - \frac{cos3\theta}{6}\right)\cdot\frac{2}{\sqrt{3}} + 0.5}{2}\right)\cdot duty\cdot cycle$$

・浮動小数点演算を使用した場合（RX, FPU を持つ RA で採用している計算式）

```
#define CONST_2_DIV_SQRT3 (1.154700538)
float harmonic;
float tmp;
harmonic = cosf(angle * 3.0f) / 6.0f;
tmp = ( (cosf(angle) – harmonic) * CONST_2_DIV_SQRT3 / 2.0f + 0.5f;
result = (usigned long)(tmp * duty * cycle);
```

angle は、ラジアン単位の角度。duty は、0-1 の duty 比。cycle はタイマの周期（整数）。最終的に 32bit タイマのレジスタ値を得る場合は、整数値にキャスト。result は、符号なし 32bit の値です。
（マイコンのタイマの仕様により、カウンタ値は 16bit または 32bit になります。）

RA2T1 では、以下の様に計算しています。

・浮動小数点を使用しない計算（例）

```
#define CONST_2_DIV_SQRT3 (295)        //2/√3×256
short harmonic;
long tmp;
harmonic = cos_t(angle * 3); //...(1)
tmp = cos_t(angle) * 6 – harmonic; //...(2)
tmp *= (long)CONST_2_DIV_SQRT3; //...(3)
tmp *= duty; //...(4)
tmp >>= 16; //...(5)
tmp += 1536; //...(6)
tmp *= (long)cycle; //...(7)
tmp >>= 10; //...(8)
result = (unsigned long)(tmp / 3); //...(9)
```

RA2T1 では、整数除算の演算もできるだけ使用しない様にしていますが、この計算では使用しています。
cos_t は、テーブル参照で cos 値を求める関数です。テーブル値は、起動後に計算を行っておきます。cos 値は、-1~1 の値を取りますが、テーブルを作成する段階で、256 倍してあります。よって、cos 値は整数値です。
angle は「ラジアン」ではなく、「度」で取り扱います。
(1)では、3 倍高調波の 1/6 を求める計算ですが、1/6 は除算となるのでここでは計算しません。そのため、(1)の段階で本来の値の 6 倍の数値で計算しています。
(2)では、harmonic を 6 倍の値で扱っているので、加減対象である cos 値も 6 倍しています。
(3)では、√3/2(=1.1547)を乗算する部分ですが、√3/2 は 256 倍して 295 を乗算します。
(4)では、duty を乗算する部分ですが、duty は 0-1 を 0-256 の 256 倍スケーリングした値で取り扱っています。よってこの部分も整数の乗算です。
ここまでの計算で、
cos : 256 倍
1/6 とする代わりに 6 倍
√3/2 を 256 倍
duty を 256 倍
で取り扱っているので、本来の計算値の $256 \times 6 \times 256 \times 256 = 100663296 = 3 \times 2^{25}$ の倍率が掛っています。このまま、サイクル値を乗算すると、long 型(32bit)のオーバフローとなるので、(5)で $1/2^{16}$ として扱う数値を小さくします。
（この時点で $3 \times 2^{25}/2^{16} = 3 \times 2^{9} = 1536$ の倍率。）
(6)では、÷2 + 0.5 の計算を全体的に 2 倍して、÷1 + 1.0 にした結果、+0.5 は、+1536 に変換され、その代わり倍率は 3×2^{10} に増加。

(7)では、サイクル数（元々整数）を掛けて、(8)では 3×2^{10} の倍率の内、2^{10} を元に戻す。
(9)残りの倍率は、3 なので 3 で割る。
（除算は避けたいところだが、整数の除算は Cortex-M23 の RA2T1 でも CPU 命令で処理されます。（sdiv 命令, 17-34 クロック…ちょっと重たいので、除算を乗算とビットシフトへの置き換えを行っても良いかと考えます。n÷3 → (n×341) >> 10 の様な置き換え。）（本チュートリアルのサンプルプログラムでは、/3 としています。）

浮動小数点演算を使用した場合の計算のプログラムコードは、前出の数式と対比させてもそれ程違和感のないものであると思います。それに対し、全て整数演算で処理している RA2T1 のプログラムコードは計算過程がかなり判り難いものになっていると考えます。もう少しシンプルに組み立てるか、計算する関数をライブラリ化してブラックボックス化してしまうなど、実際に使用する上では工夫が必要ではないかと思います。

ー三角関数の計算に関してー
前項で、三角関数は cos_t()としてテーブル参照としていると記載していますが、具体的には以下の様に使用しています。

・blm.c 内

```
//COSテーブル
short g_cos_table[360];//720bytes, cos値を256倍した値（-256~+256)をテーブル化する

//SINテーブル
short g_sin_table[360];//720bytes, sin値を256倍した値（-256~+256)をテーブル化する

//COS, SINテーブル計算
for (i=0; i<360; i++)
{                                                    #define PI 3.14159265358979f
    //0-359°の範囲のCOS値x256をテーブル化する
    g_cos_table[i] = (short)(cosf((float)i / 180.0f * PI) * 256.0f);

    //0-359°の範囲のSIN値x256をテーブル化する
    g_sin_table[i] = (short)(sinf((float)i / 180.0f * PI) * 256.0f);
}
```

テーブル計算は、単純に cosf, sinf（浮動小数点の演算を伴うライブラリ関数での計算）を使用しています。これは、非常に重たい計算となりますが、起動後に初期化のフローで計算させますので、モータ制御中に計算される訳ではありません。
また、cos, sin の計算結果を整数での取り扱いとするために、-1~1 の cos, sin 値を、256 倍して整数化しています。この場合、有効桁数は 2 桁以上 3 桁未満程度の粗さとなります。（1/256=0.4%程度の誤差要因になりますが、その程度の誤差は許容する事としています。）

三角関数で取り扱う角度は、一般的にはラジアンですが、本関数では度としています。角度も、整数で取り扱うためです。（角度は、ラジアンを 100 倍した値とか、256 倍した値とかで取り扱っても良いとは思います。）

テーブル化した cos, sin 値は単純に配列変数を参照する事で値を返す形としてます。

```
static short cos_t(short angle)
{
    //角度[°]をテーブル引きでCOSの値に変換

    //引数
    //  angle : 角度[°]

    //戻り値
    //  cos値を256倍した値

    //angleを0-359°の間に補正
    while (angle < 0)
    {
        angle += 360;
    }

    while (angle >= 360)
    {
        angle -= 360;
    }

    return g_cos_table[angle];
}
```

テーブルとしては、sin/cos 共通で 90° 分のデータを持たせておいて、sin→cos の 90° シフト、符号反転で値を返すやり方もありますが、速度重視で sin, cos 別々に 360° 分のデータをテーブル化しています。360° で 1° 刻みでは、1 データあたり 720bytes のデータとなります。(-256～256 のデータを 2 バイトの変数に代入しているので、メモリの使用効率はあまり良くないです。)sin, cos の他にも UVW 分解で使用する tan などもテーブル化しています。

RA2T1 では RAM が潤沢に使用できる訳ではありません(RAM は 8kB)が、

・cosθ

・sinθ

・tanθ

・√3/tanθ

の 4 つのテーブルで合計、2.88kB の RAM を使用しています。テーブル化するデータの種類やどのようにテーブル化するかが決まってしまえば、ROM 上にテーブルデータを持たせる方が良いかとも思います。(本プログラムでは、テーブルデータの持たせ方を容易に変更できるように、計算でテーブルデータを作成して RAM 上に展開する形としてます。)

2.3. 相補 PWM 信号での駆動（ホールセンサ使用）

参照プロジェクト：RA2T1_BLMKIT_TUTORIAL_B2

TUTORIAL_B では、VR のツマミを動かすとスムーズに回転する領域がありますが、duty を増やしても回転数が増加する事はありません（TURORIAL4 の相補 PWM 版です）。回転数は増加しないのに、消費電流だけ増える形です。

duty を増やすと回転数が duty に応じて増加する（TUTORIAL5 の相補 PWM 制御版）のが、本チュートリアルです。

ー制御方式に関してー

	制御方式	ホールセンサ （回転制御に使用しているか）
TUTORIAL4（1.4 節）	120°+PWM	未使用
TUTORIAL5（1.5 節）	120°+PWM	使用
TUTORIAL_B（2.2 節）	相補 PWM	未使用
TUTORIAL_B2（本節）	相補 PWM	使用

・チュートリアル B での blm_intr.c（100us 割り込み関数内）

```
if (g_state[i] == BLM_CH_STATE_ACTIVE)
{
    //UVWの磁界印加割合を設定
    blm_dutyset[i](g_angle[i], g_duty[i]);

    //印加磁界角度を進める
    if (g_target_direction[i] == BLM_CCW)
    {
        g_angle[i] += g_angle_diff[i];
        if (g_angle[i] > DEGREE_360)
        {
            g_angle[i] -= DEGREE_360;
        }
    }
    else if (g_target_direction[i] == BLM_CW)
    {
        g_angle[i] -= g_angle_diff[i];
        if (g_angle[i] < 0)
        {
            g_angle[i] += DEGREE_360;
        }
    }
```

g_angle_diff[i] = 64

100us 毎に決まった角度
（1667rpm に相当する 1[°]）
を加算する

角度が際限なく大きくなるといずれオーバフロ
ーするので、360°に制限する

逆回転の場合は、角度を減算する

angle は、「度単位」で取り扱っていますが、100us 毎の増分が1°の場合、1667[rpm]。2°の場合、3334[rpm]となります。回転数の分解能が、1667[rpm]では多少荒すぎるので、g_angle や g_angle_diff[]は、° ×64 の数値で取り扱っています。（g_angle=64 の時は、1°を表す）

チュートリアル B では角度の増分は、常に一定値（1667rpm に相当する角度）としています。そのため、回転した場合の回転数は設定した duty に拘わらず、大体 1667rpm です。

それに対し、本チュートリアルではホールセンサの値を見て、
(1)100us 毎の角度増分を決定する（回転が速いときは角度増分を増やす）[チュートリアル B では固定値]
(2)相補 PWM の印加角度(θ)をホールセンサの位置に合わせる
という処理を行っています。

※100us 毎に角度を加算するという、基本的な回転制御の部分はチュートリアル B での制御と変わりません

・チュートリアル B2 での blm_intr.c（100us 割り込み関数内）

```
if (g_state[i] == BLM_CH_STATE_ACTIVE)
{
    //理想位置を算出
    ideal_angle = blm_ideal_angle(g_sensor_pos[i], g_target_direction[i], g_angle_forward[i]);
                                                    ホールセンサが切り替わった際の理想的な角度を算出
    //速度のずれを g_angle_diff に反映
    g_angle_diff[i] = blm_angle_diff_calc(g_angle_diff[i], ideal_angle, g_angle[i],
g_target_direction[i]);
                                                    (1)理想的な角度と現在の角度を比較して
                                                    100us 毎の角度増分値を
                                                    理想的な角度となるように近づけていく
    //印加角度をセンサから算出される理想位置にずらす
    g_angle[i] = ideal_angle;
}                       (2)印加角度を理想値に上書き
```

blm_ideal_angle()関数はホールセンサの位置（チュートリアル 4 の pos=1~6）に応じた、そのタイミングでの理想的な角度を算出する関数です。

g_angle_diff[i]は、チュートリアル B では 1667rpm で算出した固定値でしたが、本チュートリアルでは、ホールセンサ切り替わり時の「理想角度」と「現在の角度」を比較して、100[us]毎の角度増分を理想値に近づける様に変更しています。印加角度の理想値は、以下で算出しています。

・blm.c（blm_ideal_angle()関数内）

```
if(direction == BLM_CCW)
{
    switch(pos)
    {
        case 3:
            ret = DEGREE_330;
            break;
        case 2:
            ret = DEGREE_30;
            break;
        case 6:
            ret = DEGREE_90;
            break;
        case 4:
            ret = DEGREE_150;
            break;
        case 5:
            ret = DEGREE_210;
            break;
        case 1:
            ret = DEGREE_270;
            break;
        default:
            return 0;
            break;
    }
```

DEGREE_330 = 330 × 64 = 21120
…330°に、角度取り扱い倍率の 64 を掛けた値

単純にホールセンサ位置と角度の対応テーブルとしています。(ホールセンサが 3 に切り替わった際は、印加角度が 330°となっているのが理想)

また、100us 毎に進める印加角度に関しては、下記で計算しています。

```
short blm_angle_diff_calc(short diff_angle_mul, short ideal_angle_mul, short angle_mul, short
target_direction)
{
    //制御周期(100us)毎の角度増分を計算する関数

    //引数
    // diff_angle_mul : 現状の角度差分
    // ideal_angle_mul : センサ切り替わり時の理想的な角度
    // angle_mul : 現状の角度
    // target_direction : 回転方向

    //戻り値
    //  計算後の diff_angle

    /*
     * ideal_angle_mul = angle_mul
     * となる様に、diff_angle_mul を微調整する
     *
     * ideal_angle_mul - angle_mul が
     *
     *  プラス : 現状の diff_angle_mul が遅い
     *  マイナス : 現状の diff_angle_mul が速い
     */

    short angle_sub;
    const long feedback = (long)(65536.0f * (float)BLM_ANGLE_DIFF_FEEDBACK);    //フィードバック係数
×65536

    angle_sub = ideal_angle_mul - angle_mul;

    //180[°]より大きな場合は diff_angle を変更しない
    if (angle_sub > DEGREE_180)
    {
        return diff_angle_mul;
    }

    //-360[°]より小さな場合は diff_angle を変更しない
    if (angle_sub < -DEGREE_360)
    {
        return diff_angle_mul;
    }

    //角度は、-180°~180°の範囲内に変換する
    if (angle_sub < -DEGREE_180) angle_sub += DEGREE_360;

    //理想との差分を BLM_ANGLE_DIFF_FEEDBACK の割合で埋めていく
    return diff_angle_mul + (short)((angle_sub * target_direction * feedback) >> 16);   //フィードバ
ック係数×65536 を元に戻す
}
```

=0.01f (1%)

feedback 変数は、コンパイル時に
値が定まる(都度、浮動小数点の演算を
行う訳ではない)

100us 毎の角度増分を決定する部分は、現状の角度増分(diff_angle)に対し、理想値とのずれ(angle_sub)を加算するのですが、1 回で理想値にしてしまうのではなく、BLM_ANGLE_DIFF_FEEDBACK(=0.01)(1%)ずつ差分を埋めていく(diff_angle を滑らかに変化させる)方式です。

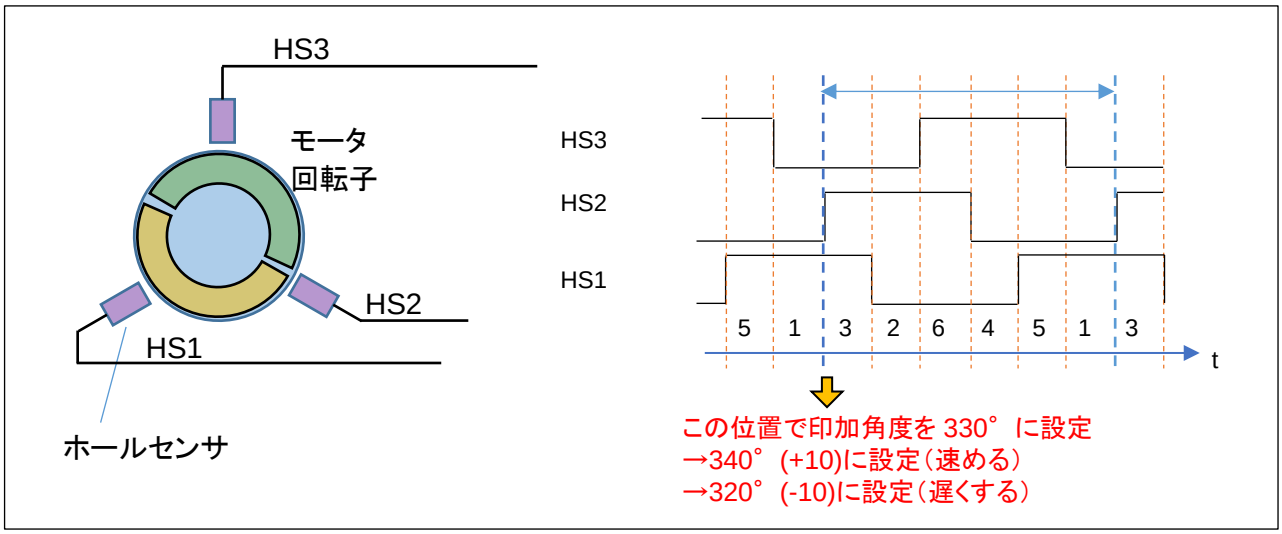
・シリアル端末から出力される情報

```
                          CH-1
Motor Driver Board        : Connect
Active                   :     o
UVW calculation method    : (1)
diff angle -> speed([rpm]):  2343
forward angle([deg])      :     0
target direction          :   CCW
rotation speed([rpm])     :  2400
Temperature(A/D value)    :  2128
Temperature(degree)       :    28
VR(A/D value)             :  1041
duty[%]                  :  25.4
```

diff angle -> speed　は、現在の磁界印加角度増分を回転数[rpm]に直したものです。(duty に応じた印加角度増分となる様、計算された値)

foward angle は進角調整値です。

基本的には、いままでのチュートリアルと大きくは変わりませんが、進角(forward angle)の表示、機能が追加されています。進角調整はホールセンサ位置と磁界の印加角度の関係を調整する機能です。

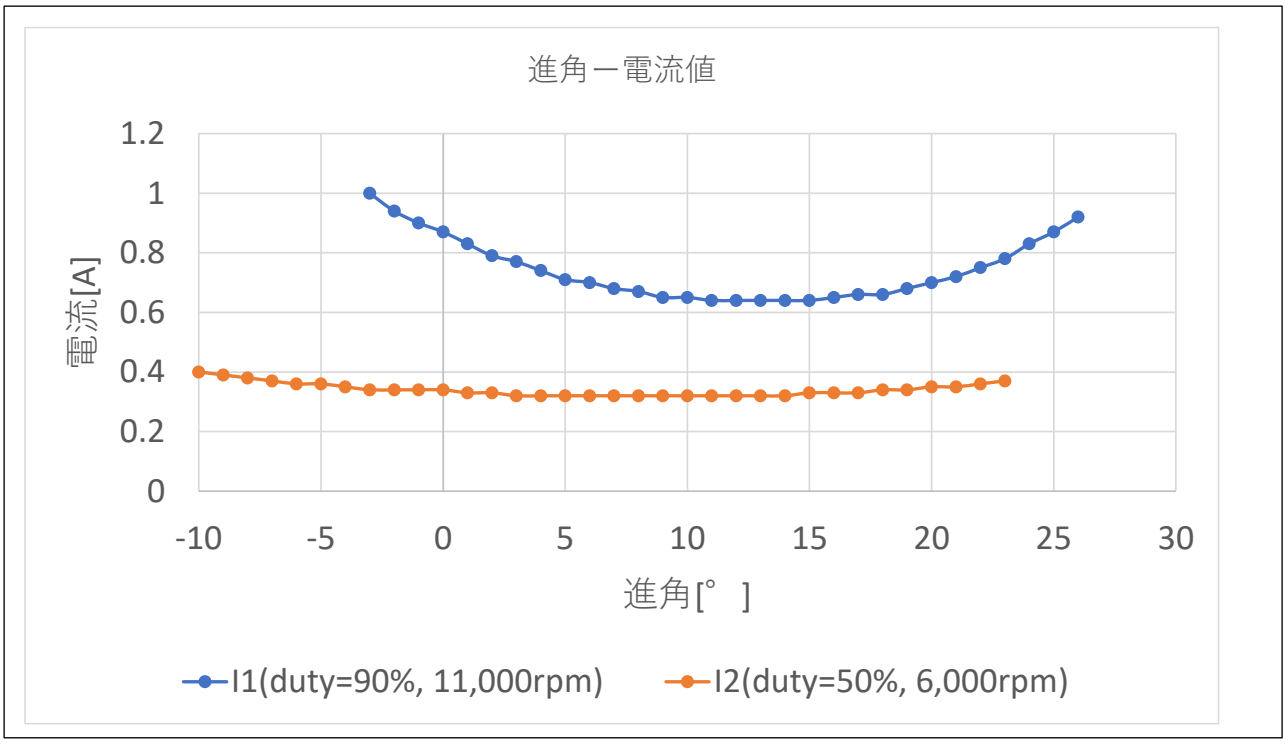


ホールセンサの出力が切り替わるタイミングで磁界の印加角度を設定していますが、この角度を速めたり遅くしたりする機能が進角調整です。この進角の値は、シリアル端末のキーボードで設定を行います。

	-1°	+1°	リセット(=0)
CH-1	q	w	e

キーボードから'q'を入力すると角度が 1° 遅くなります。'w'で 1° 速くする方向です。'e'で初期値(=0)に戻します。調整範囲は±45° の範囲です(blm.h 内で定義、変更は可能)。モータが停止しているときでも変更は可能です。一般に高速回転時は、進角を+の方向(エンジンでしたら点火タイミングを速めるイメージでしょうか、モータであれば磁界の引っ張る角度を少し前の方に設定する)にすると、消費電流が減る(効率が上がる)事が期待できます。

・進角設定値と電流値の変化



duty を 90%に設定し、11,000[rpm]程度でモータを回転させ、キーボードから q/w を入力し進角調整を行った場合の電流値を示します。この例では、進角を 13° 程度に設定した場合、一番電流値が減る事が観測されました。また、duty を 50%程度に設定し、6,000[rpm]程度でモータを回転させた場合は 8° 程度が電流値最小となりますが、ほぼフラットな電流値のカーブとなります。

高速回転時の方が

・電流が最小となる角度が大きい

・進角調整の効果が（電流の減少率）が大きい

という事が言えるかと思います。

本チュートリアルでは、デバッグ情報を追加で表示させる事が可能です。

・起動時のメッセージ

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL B2


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> NONE
LED1 : CH-1 Active ON/OFF
LED3 : ERROR status
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display
D : rotation direction->CCW <-> rotation direction->CW (toggle)
q : forward angle -1  [CH-1]
w : forward angle +1  [CH-1]
e : forward angle =0  [CH-1]
1 : UVW calc -> sine
2 : UVW calc -> sine(2)
3 : UVW calc -> sine + 3harmonic
4 : UVW calc -> sine + 3harmonic(2)
5 : UVW calc -> another version
6 : UVW calc -> another version(100% power)
7 : UVW calc -> another version(100% power)(2)
z : debug display(toggle)

>
```

q~e は、前出の進角調整の機能です。

'z'を入力するとホールセンサ切り替わり時の角度が表示される様になります。(もう一度'z'を入力すると表示されなくなります)

'z'を入力し、デバッグ出力を有効化すると、ホールセンサが切り替わったタイミングで

・シリアル端末から出力される情報

```
c1:pos:3:deg:332(2)
c1:pos:2:deg:27(-3)
c1:pos:6:deg:87(-3)
c1:pos:4:deg:152(2)
c1:pos:5:deg:208(-2)
```

c1: CH-1　(本キットでは、CH-1 のみ)

pos:3 ホールセンサの位置=3 に切り替わったタイミング

deg:332 その時の磁界印加角度

(2) 理想値 330°に対して 332 なので差分 2°(=現在値－理想値)

が表示されます。

'z'の入力に応じて表示、非表示は切り替わります。
（3 秒に 1 回表示される回転数等の情報は、's'コマンドで表示・非表示の切り替えが可能です。）

なお、非表示（起動時のデフォルト）にした場合でも、表示するかどうかの条件分岐のオーバヘッドはあります。（表示処理のために多少処理が重たくなります。）完全に表示する機能を無効化する場合は、

・blm.h

```
//デバッグ表示
#define BLM_DEBUG_PRINT_1  //定義時デバッグ情報を出力を可能とする
```

上記定義を未定義（コメントアウトや削除）とすると、表示に掛かる処理のオーバヘッドはなくなります。

※UART の表示速度より出力される情報量が多い場合は、表示用のバッファが溢れた時点で一部の表示は失われます（表示が途中で切れるケースもあります）

・チュートリアル B2 での端子設定
→チュートリアル B に同じ

・チュートリアル B2 での使用機能
→チュートリアル B に同じ

2.4. センサレス駆動

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL_C

本チュートリアルでは、モータの電流印加方向の切り替えをホールセンサを使用しないで制御する方式を試します。

制御方法は、TUTORIAL7 と同じ、120 度制御です。

	起動時のデフォルト	コマンドにより切り替え	用途
S コマンド	通常	始動制御	始動制御切り替え

	OFF	ON	用途
SW2	ホールセンサ使用	ホールセンサ未使用	電流切り替え方式選択

電源を投入した後(VR は絞った状態としてください、SW2 は OFF としてください)、SW1 を ON＝モータを ON にして VR を回していくとモータが回転を始めるはずです。

但し、この時の動作は TUTORIAL7 と変わりません。ホールセンサの切り替わりのタイミングで電流方向の切り替えを行っています。

モータが回転している状態で、SW2 を ON(右側)にしてみてください。(回転している状態で SW2 で動作を切り替えても、見た目上の変化は生じないと思います。SW2 を ON にすると、モータの電流方向切り替えは、疑似ホールセンサパターンで行われる様になります。SW2 を OFF にすると再度ホールセンサを使用する制御となります。

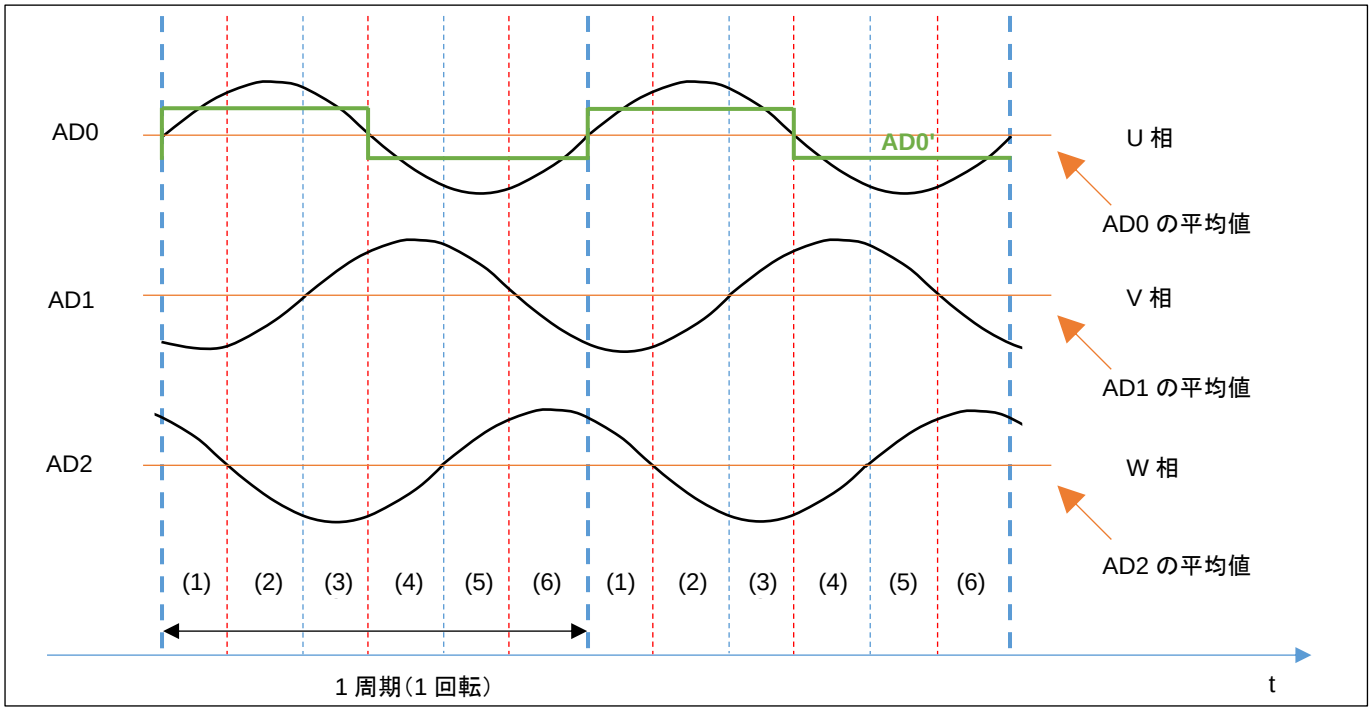
ホールセンサを使用しない場合は、ホールセンサの切り替わりを模擬した疑似ホールセンサパターンを使って電流の切り替えを行いますが、疑似ホールセンサパターンの取得には、モータが回転している事が条件となります。
→ホールセンサを使用しない場合、モータ停止時の軸の位置は判らない

そこで、最初はホールセンサを使用して回転を始めさせ、ある程度回転した状態で、SW2 を使用してセンサレス駆動へと移行します。

(モータの回転が止まった場合は、SW2 を OFF にしてホールセンサを使用する様に切り替えてモータを回転させてから再度 SW2 でホールセンサを使用しない状態に切り替えてください。)

疑似ホールセンサパターンの生成には、UVW の相電圧を使用しています。

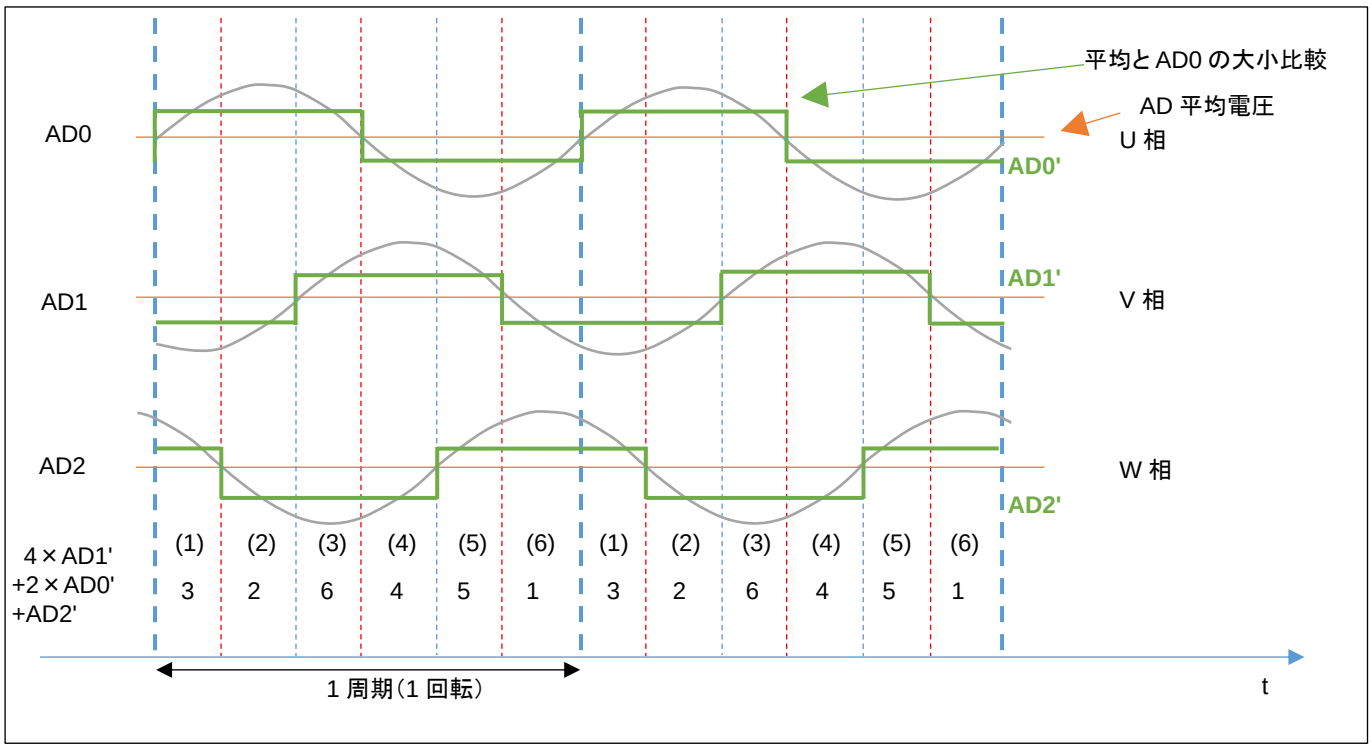
・各相電圧 LPF を通した波形



AD0~AD2 の信号は、UVW 各相電圧の LPF(Low Pass Filter, 低域通過フィルタ)通過後の波形です。PWM 制御を行った相電圧は、複雑な波形となりますが、LPF で信号処理を行うと、sin 波に近い波形となります。
AD0~AD2 は、マイコンの A/D 変換機能を使用して値を取得しているので、得られる値は電圧値ではなく、A/D 変換値(0~1023)です。ここでは、AD0 の平均値と現在の AD0 の値が判ればよいので、A/D 変換値のままで大小比較を行います。

AD0(U 相電圧)の平均値と AD0 の大小比較を行う事により、AD0'(デジタル的な値、0/1 値)を得ることが出来ます

この、AD0'の信号を使う事により、モータの軸の位置を特定して、モータに印加する電流の向きを切り替えます。

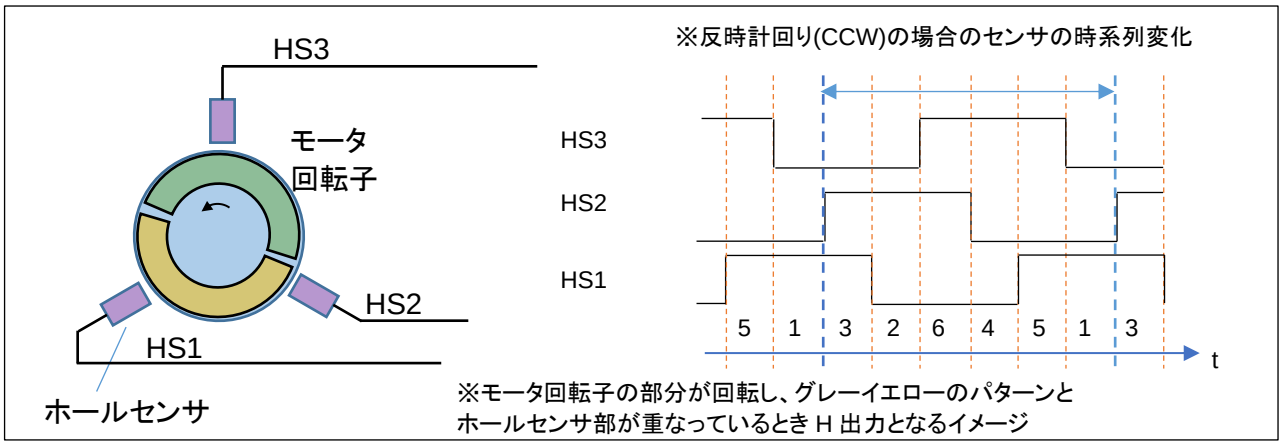


ホールセンサを使用した既存のプログラムをそのまま使う場合、AD0'~AD2'の 0/1 信号を生成して、重み付け

$4 \times AD1' + 2 \times AD0' + AD2'$

を行う事で、1~6 までの数値が得られます。この値を疑似ホールセンサパターンとします。

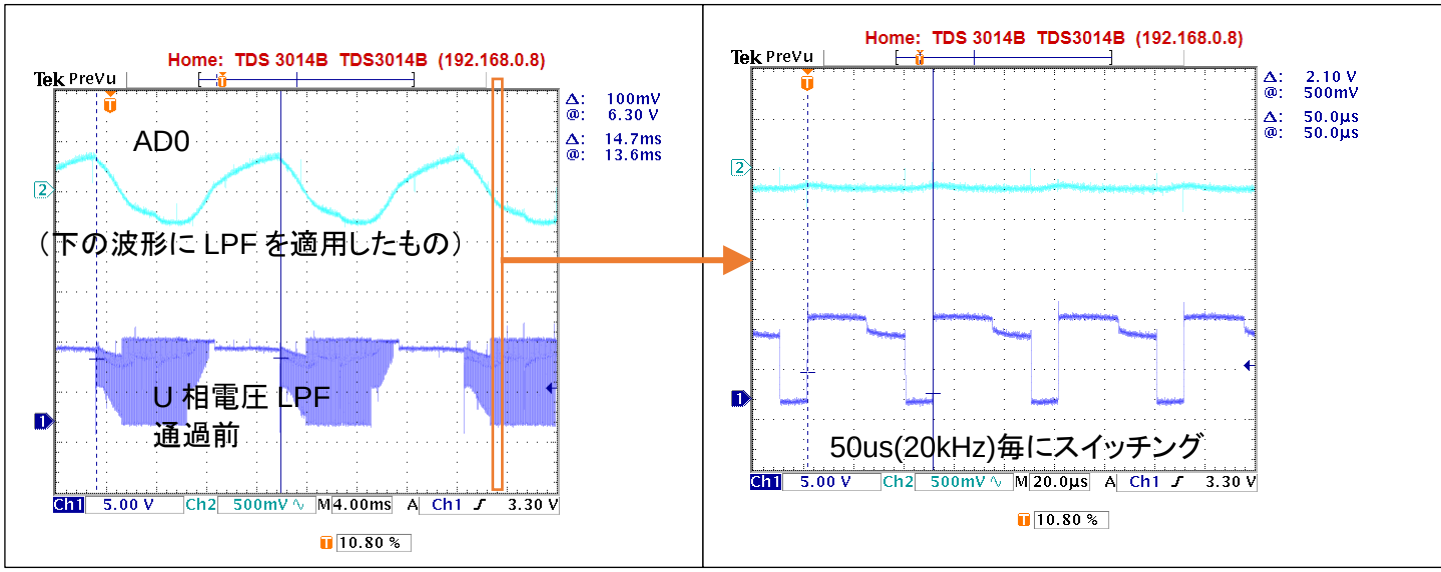
この、3, 2, 6, 4, 5, 1 の値、順番がホールセンサ



の出力と一致する様に重み付けを行ったので、プログラム的には、「ホールセンサの値(1~6)」と疑似ホールセンサパターン(1-6)」を同様に処理します。(センサ値と電流の印加方向の関係は、ホールセンサ，疑似ホールセンサパターンで同じです。)

※回転方向が CW(時計回り)の場合、モータのホールセンサと同じ出力を得る場合、重み付けの計算は、$4 \times \overline{AD2'} + 2 \times \overline{AD1'} + \overline{AD0'}$となります。本チュートリアルでは回転方向は、CCW のみ対応しています。サンプルプログラム(RA2T1_BLMKIT_SAMPLE)には、回転方向 CW に対応したコードが記載されています。

・AD0 電圧の A/D 変換値の取得

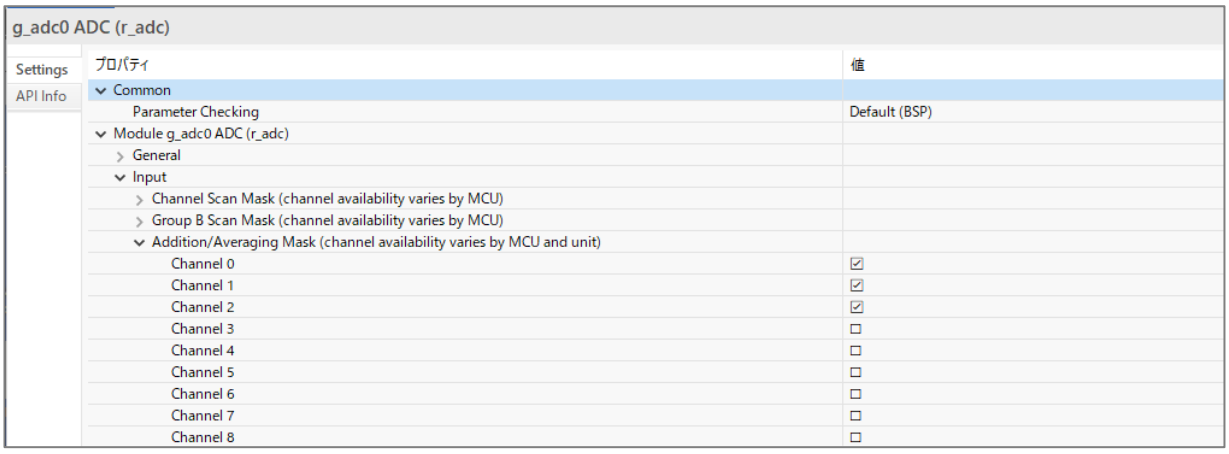


時間軸を拡大（左の 4ms/div に対し 20us/div）

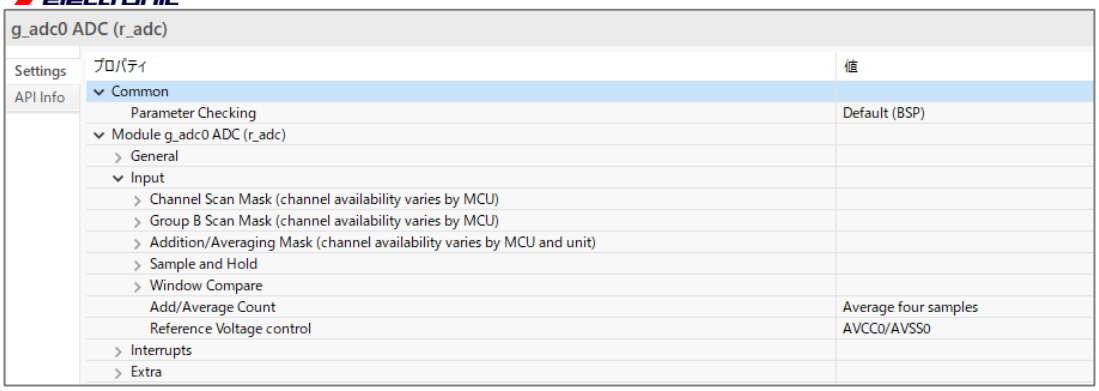
モータ端子の U 相電圧は LPF 通過前は（V→W に電流が流れているタイミング等で）パルス状の信号が発生していたり、（U 相が動かないタイミングでは）止まっていたり、一見意味のない信号に見えます。この信号に LPF を適用すると、AD0 の信号が得られます。LPF は、モータドライバボード上の回路で処理されています。

LFP 通過後の AD0 の信号でも、U 相電流が ON/OFF するタイミング等では、ノイズが乗るので本チュートリアルでは、A/D 変換値の平均化を取る様にしています。

FSP では、r_adc stack の設定で、



		設定値	備考
Input	Additon/Averaging Mask		平均化したい端子(Channel)にチェックを入れる
	Channel0	チェック	AN000→U 相電圧
	Channel1	チェック	AN001→V 相電圧
	Channel2	チェック	AN002→W 相電圧



		設定値	備考
Input	Add/Average Count	Average four samples	4 回の A/D 変換結果の平均化

上記の様に設定を行う事により、A/D 変換実行時間は増加しますが、ユーザ側はプログラム上で演算することなく、平均値が得られます。

接続信号名	内容	使用端子	平均化対象
AD0	U 相電圧	**P013/AN000**	○
AD1	V 相電圧	**P014/AN001**	○
AD2	W 相電圧	**P015/AN002**	○
AD3	U 相電流	P000/AN008	
AD4	V 相電流	P001/AN009	
AD5	W 相電流	P002/AN010	
AD6	温度センサ	P010/AN005	
AD003	電源電圧	P011/AN006	
VR	ボリューム	P012/AN007	

・Additon/Averaging Mask

	ADC0(g_adc0)
Channel 0	チェック AN000 を平均化
Channel 1	チェック AN001 を平均化
Channel 2	チェック AN002 を平均化
Channel 3	
Channel 4	
Channel 5	
Channel 6	
Channel 7	
Channel 8	
Channel 9	
Channel 10	
Channel 11	
…	

※なお、本チュートリアルでは、使用していませんが、PWM 波形を生成しているタイマに同期して A/D 変換をキックする事も可能です（その場合、波形切り替えのタイミングとずらして、A/D 変換をキックする事ができます）

・平均値（AD0 の DC 的な平均値）の算出

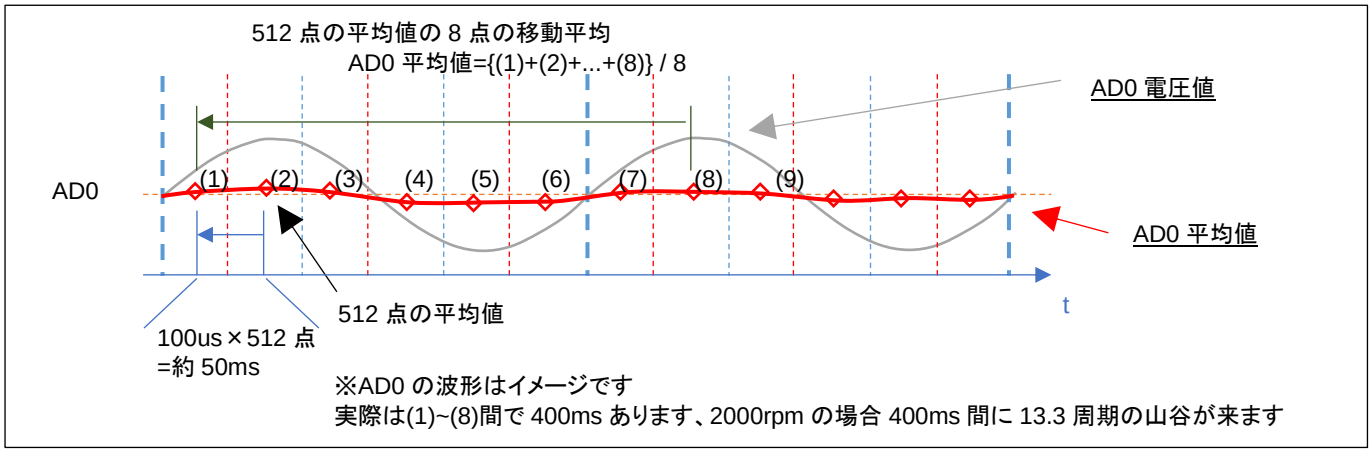
AD0 の信号は、LPF 通過後も sin カーブ状態ですので、AD0 との比較対象に使用する長期間の平均値を計算してします。本チュートリアルでは、512 点の平均値を取り、さらに 512 点の平均値 8 点の移動平均を求めています。

512 点の平均は、51.2ms に相当し、512 点×8 点は大体 400ms に相当します。

・blm.h

```
//ADC長期間の移動平均
#define BLM_ADC_LONG_AVERAGE 512    //512点の平均を求める（256, 512, 1024, 2048, 4096の値が有効）
（中略）
#define BLM_ADC_LONG_AVERAGE_HIST 8 //512点の平均値の8点の移動平均を取り最終的な平均値を求める
（2,4,8,16,32の値が有効）
```

512 点や 8 点は、blm.h 内の定数定義で変更可能です。



AD0 変換値は、100us 毎の A/D 変換値を 512 点平均を取り、その 512 点の平均値の移動平均を AD0 の平均値とします。

時間が t=(8)の時は、(1)~(8)の平均値を AD0 平均値とし、t=(9)の時は(2)~(9)の平均値を AD0 平均値とします。（約 50ms 毎に、最新の 512 点の平均値を取り込み、400ms 前の値を捨てる形で、平均値は更新されます。＝移動平均の考え方）

・AD0(,AD1, AD2)の移動平均と閾値のオプション

AD0(,AD1, AD2)は、
(1)移動平均値を取る(*1)
(2)ヒステリシス特性を持たせる
2 つのオプションを用意しています。

(*1)前出の移動平均の話は、「AD0 の平均値」の算出時の話で、AD0 の平均値の算出の際は 512 点のさらに 8 点の移動平均を取っています。この部分では、平均値ではなく、現在値をどう取り扱うかの話です。

・blm.h

```
//ADC短期間の移動平均
#define BLM_ADC_SHORT_AVERAGE_HIST 8 //移動平均のポイント数（2,4,8,16,32の値が有効）
```

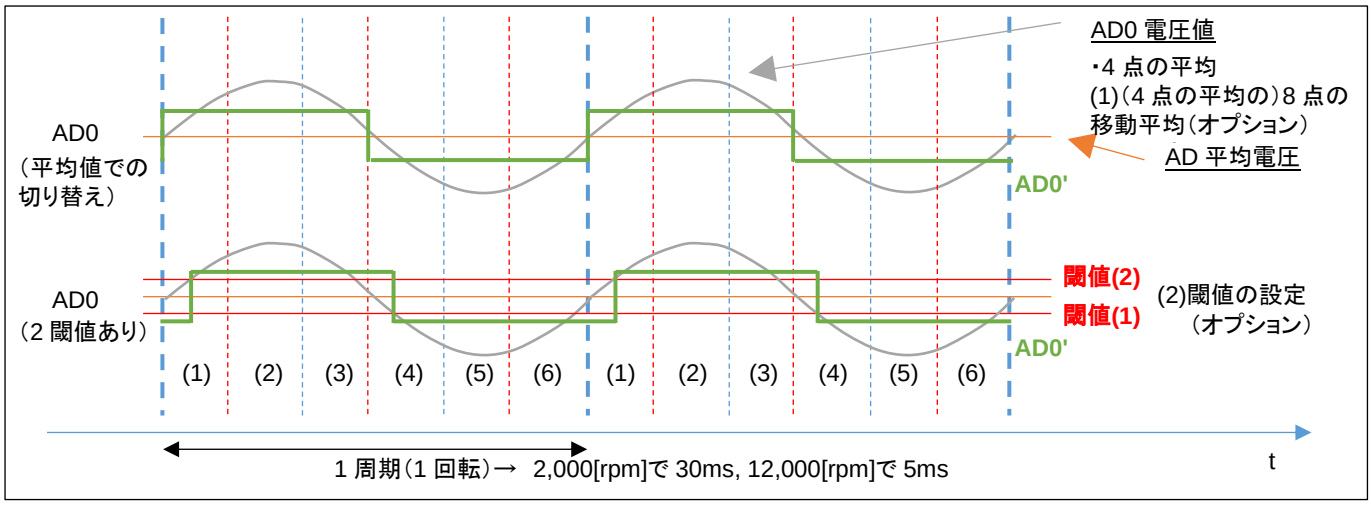
```
//疑似ホールセンサパターン
#define BLM_HALL_PSEUDO_SENSOR_AVERAGE 0x1 //b0=1:電圧の移動平均をホールセンサパターンとする，b0=0:その時の電圧(A/D値4点の平均）をホールセンサパターンとする
#define BLM_HALL_PSEUDO_SENSOR_HYS 0x2 //b1=1:ヒステリシスを有効にする，b1=0:ヒステリシス無効
#define BLM_HALL_PSEUDO_SENSOR_HYS_VAL 16 //16 = 20mV/5000mV*4096，20mV程度ヒステリシスを付ける
```

(1)移動平均の算出（プログラムでの平均化）

プログラムでは、AD0(,AD1, AD2)の A/D 変換値の移動平均を計算して使用できる様にしています。

マイコンの A/D 変換で、4 点の平均を取っていますが、4 点の平均値のさらに 8 点の移動平均を AD0 の値として使用します。

マイコンの A/D 変換機能での平均化に加え、ユーザプログラムでの移動平均化を行いノイズ対策を行える様にしています。



(2)ヒステリシスの有効化

AD0 の平均電圧との比較では、単純な大小比較（デフォルト）と、ヒステリシスを持たせた比較（オプション）が選べるようになっています。

ヒステリシスは、'h'コマンドで有効・無効を切り替えます。

※一般的にはノイズの多い信号を処理する場合はヒステリシス（0→1 に切り替わる閾値と 1→0 に切り替わる閾値に差を付ける）があった方が誤動作を防止できます

AD0 の移動平均を取る(1)とヒステリシス(2)は、どちらも低速回転時は有効／無効で大差なし。高速回転時は、有効にすると、（電流方向の切り替えが遅くなる分）消費電流が増えるイメージです。

・起動時のメッセージ

```
Copyright (C) 2025 HokutoDenshi. All Rights Reserved.

RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL C


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> OFF:Hall sensor use, ON:Pseudo hall sensor pattern use
LED1 : CH-1 Active ON/OFF
LED2 : ON:Starting operation, OFF:Normal operation
LED3 : ERROR status
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display
S : Normal operation <-> Starting operation(toggle)
a : pseudo hall sensor pattern <-> average voltage(toggle)
h : pseudo hall sensor pattern <-> hysteresis voltage(toggle)
z : debug display(toggle)

>
```

起動時は、平均化('a)とヒステリシス('h')は両方 OFF です。キーボードからのコマンド入力'a', 'h'で有効・無効がトグルで切り替わります。

・疑似ホールセンサパターンのデバッグ表示

キーボードから'z'を入力すると、疑似ホールセンサパターンのデバッグ表示を行います。

・シリアル端末から出力される情報

```
c1:pos:5,5h
c1:pos:1,1h
c1:pos:3,3h
c1:pos:2,2h
c1:pos:2,2h
c1:pos:6,6h
c1:pos:4,4h
c1:pos:5,5h
c1:pos:1,1h
c1:pos:1,3h
c1:pos:3,3h
c1:pos:2,2h
c1:pos:6,6h
c1:pos:4,4h
c1:pos:4,5h
c1:pos:5,5h
c1:pos:1,1h
c1:pos:3,3h
c1:pos:2,2h
c1:pos:6,6h
c1:pos:6,6h
c1:pos:4,4h
```

上記の様な出力が得られます。

c1: CH-1

pos: 5,5h

5 ホールセンサの位置情報

5 疑似ホールセンサパターンの位置情報

h 現在制御にホールセンサを使用

p 現在制御に疑似ホールセンサパターンを使用

（デフォルトでは、2ms 毎に表示）

この表示により、疑似ホールセンサパターンとホールセンサの値が合っているか、どちらが速く切り替わるかを確認可能です。

・モータの始動に関して

先に示したモータの始動方法は、「ホールセンサを使用して初期始動を行う」という方式です。

ホールセンサレスで制御する目的で、ホールセンサを使用するというのは、矛盾があると思います。

通常は、センサレス駆動の場合、始動時は「ホールセンサの値」「疑似ホールセンサパターンの値」どちらも使用できないので、一定回転数で電流の向きを変化させてモータを始動します。モータ始動後は、疑似ホールセンサパターンの値を使って回転を維持させます。

モータの始動にホールセンサを使わない手順を以下に示します。

(1)SW1=OFF の状態で SW2=ON にして疑似ホールセンサパターンを使う設定とします

(2)S コマンドを入力して始動モードにします
→モータに印加する電流を 6ms 毎に 1/6 回転(1667[rpm])する様に切り替える設定です

```
Operation mode -> StartUP
```

(3)VR を絞り SW1 を ON にします

(4)VR を回していきます
→この時の動作は 1.4 章の TUTORIAL4 のモータの回転数は一定、duty は可変という状態です
（電流の切り替えは一定時間間隔に行われ、ホールセンサ、疑似ホールセンサパターンのどちらも使用しない動作です。）

(5)モータが安定して回る様になったら、S コマンドを入力して始動モードをやめます
→モータは、疑似ホールセンサパターンでの制御となります

```
Operation mode -> Normal
```

本チュートリアルでは、(5)の手順（始動制御状態からセンサレス駆動への移行）は手動で行う事としていますが、一般的なセンサレス駆動の場合、この部分をプログラムで判断して自動的に移行させる事となります。

・シリアル端末から出力される情報（3 秒毎に表示される回転数等の情報）

```
                      CH-1
Motor Driver Board     : Connect
Active                :     o
hall sensor           :  Pseudo hall sensor pattern, phase voltage
  average    ->  OFF
  hysteresis ->  OFF
rotation speed([rpm])  :  3960
Temperature(A/D value) :  2147
Temperature(degree)    :    28
VR(A/D value)          :   946
QL duty[%]             :  23.0
```

ホールセンサ区分（モータ内蔵ホールセンサか、疑似ホールセンサパターンか）と、疑似ホールセンサパターンの際には、平均化とヒステリシスの ON/OFF が表示されます。

※本チュートリアルでは、回転数の計算やモータドライバボードの接続確認にホールセンサケーブルがつながっている事が前提となっています。（ホールセンサケーブルを接続しない状態では、モータドライバボード未接続となりモータに信号は送られません）

・チュートリアル C での端子設定

→チュートリアル 7 に同じ

・チュートリアル C での使用 stack

名称	リソース	周辺機能	用途	備考
g_ioport	r_ioport		汎用 I/O	最初から追加済み
g_uart9	r_sci_uart	SCI9	UART 通信	
g_agt0	r_agt	AGT0	100us タイマ	
g_agt1	r_agt	AGT1	10ms タイマ	
g_gpt3	r_gpt	GPT3	QL PWM 波形生成	
g_adc0	r_adc	ADC0	A/D 変換	
g_external_irq4	r_icu	IRQ4	過電流検出	
g_gpt0	r_gpt	GPT0	始動制御	6ms 周期

※グレーの項目はチュートリアル 7 から変更なし

2.5. センサレス＋相補 PWM 駆動

参照プロジェクト:RA2T1_BLMKIT_TUTORIAL_BC

本チュートリアルは、2.4 節のセンサレス駆動(TUTORIAL_C)（疑似ホールセンサパターン使用）のモータ駆動部を、2.3 節の相補 PWM 駆動(TUTORIAL_B2)に置き換え、2 つのチュートリアルを組み合わせたものです。

TUTORIAL_C 同様、モータの起動は 2 通り（ホールセンサで起動するか、一定回転数で起動）です。

・ホールセンサでの起動

(1)SW1 を OFF, SW2 を OFF, VR を絞った状態とします

(2)SW1 を ON にして、VR を回していき、モータを回転させます
→この時はホールセンサを使用しています、TUTORIAL_B2 と同じ動作です

(3)SW2 を ON にしてセンサレス動作に切り替える
→モータの回転制御に疑似ホールセンサパターンを使用する様に切り替わります

・一定回転数で起動

(1)SW1 を OFF, SW2 を ON, VR を絞った状態とします

(2)S コマンドを入力
→一定回転数（1667rpm で磁界印加角度を進めていく設定）

```
Operation mode -> StartUP
```

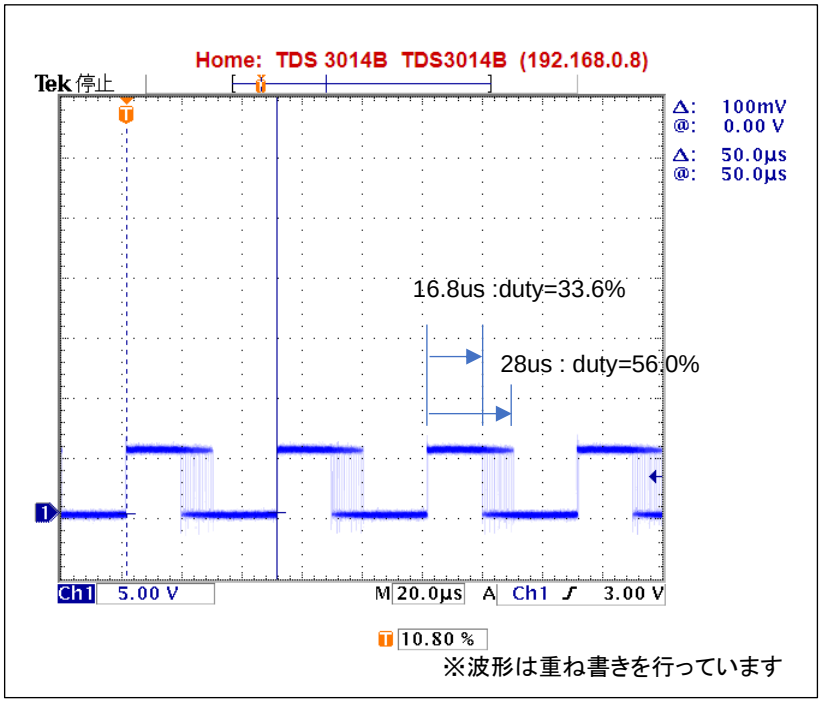
(3)SW1 を ON にして、VR を回していき、モータを回転させます
→この時の動作は 2.2 章の TUTORIAL_B のモータの回転数は一定、duty は可変という状態です

(4)モータが安定して回る様になったら、S コマンドで始動モードを切り替えます
→モータの回転制御に疑似ホールセンサパターンを使用する様に切り替わります

```
Operation mode -> Normal
```

※S コマンドはトグル動作なので画面表示が上記のメッセージとなる様に、場合によっては 2 回入力してください
本チュートリアルは、TUTORIAL_B2 と TUTORIAL_C の組み合わせなので、特に新しい要素はありません。

・PWM 波形イメージ（相補 PWM）



※波形は重ね書きを行っています

・矩形波の周期は 50us(20kHz)

・duty は、徐々に変化していく

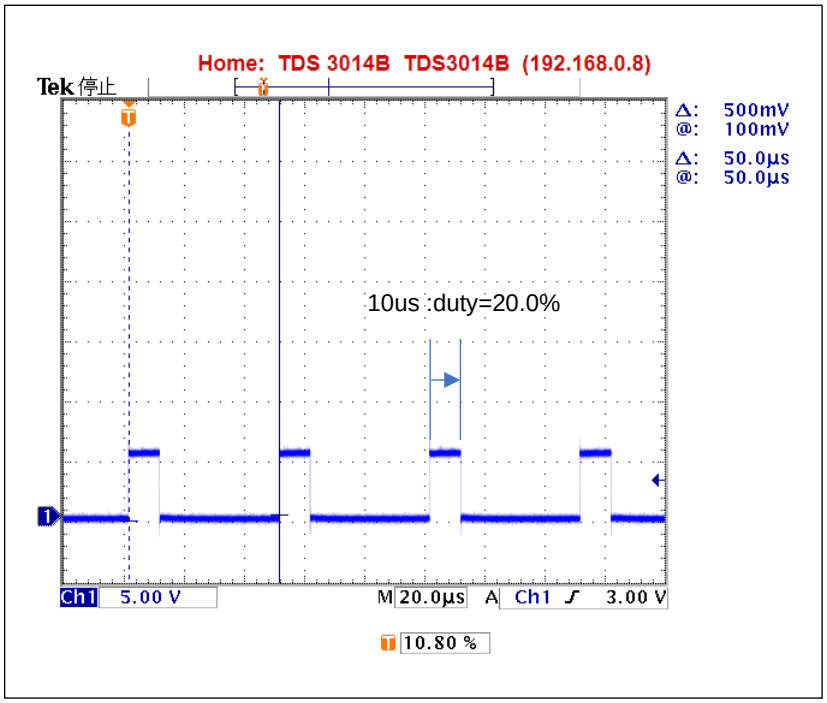
→上記の波形例では duty が約 34~56%の間で連続的に変化している

（波形取得時に、VR は回していません。モータに印加する duty は一定の状態でも、U 相, V 相, W 相の個々の波形の duty は連続的に変化する動作となります。）

→隣り合うパルスで徐々に duty が変化していく形となります

（上記の波形は、1 相の波形ですが、6 相全ての波形の duty が同じように動いています）

・PWM 波形イメージ（120 度制御）



・矩形波の周期は 50us(20kHz)
・duty は、VR を回さない限り変化なし
（上記の様な波形が、U, V, W 相の L 側の 3 相で出ているタイミングと出ていないタイミングがあります）
（チュートリアル 7 等では、H 側は duty=100%、ON のタイミングでは、H を維持、L 側のみ PWM 駆動）

相補 PWM の波形は、duty が連続的に変化するという点で、120 度制御に比べるとノイズが大きい形となります。（120° 制御では、決まった位置にスイッチングノイズが発生するのでスイッチングノイズが生じないタイミングで A/D 変換を行えば良いが、相補 PWM ではスイッチングノイズが生じるタイミングが常に移動）そのため、疑似センサパターンの判定時に平均化やヒステリシスを有効にした方が動作が安定する事は考えられます。

・起動時にシリアル端末から出力される情報

```
RA2T1 / BLUSHLESS MOTOR STARTERKIT TUTORIAL BC


EXPLANATION:
SW1 -> CH-1 motor ON/OFF
SW2 -> OFF:Hall sensor use, ON:Pseudo hall sensor pattern use
LED1 : CH-1 Active ON/OFF
LED2 : ON:Starting operation, OFF:Normal operation
LED3 : ERROR status
VR -> duty(0-100%)

COMMAND:
s : stop <-> start display information(toggle)
A : A/D convert data display
D : rotation direction->CCW <-> rotation direction->CW (toggle)
S : Normal operation <-> Starting operation(toggle)
a : pseudo hall sensor pattern <-> average voltage(toggle)
h : pseudo hall sensor pattern <-> hysteresis voltage(toggle)
q : forward angle -1  [CH-1]
w : forward angle +1  [CH-1]
e : forward angle =0  [CH-1]
1 : UVW calc -> sine
2 : UVW calc -> sine(2)
3 : UVW calc -> sine + 3harmonic
4 : UVW calc -> sine + 3harmonic(2)
5 : UVW calc -> another version
6 : UVW calc -> another version(100% power)
7 : UVW calc -> another version(100% power)(2)
z : debug display(LEVEL1)(toggle)
x : debug display(LEVEL2)(toggle)

>
```

2 種類のデバッグ表示（'z', 'x'で表示・非表示の切り替え）がある点が今までのチュートリアルとの相違です。

・シリアル端末から出力される情報（3 秒毎に表示される回転数等の情報）

```
                          CH-1
Motor Driver Board        : Connect
Active                   :     o
hall sensor              :  Pseudo hall sensor pattern, phase voltage
  average    ->  OFF
  hysteresis ->  OFF
rotation control(PHASE)   : Normal(2)
UVW calculation method    : (1)
diff angle -> speed([rpm]):  2005
forward angle([deg])      :     0
target direction          :   CCW
rotation speed([rpm])     :  2040
Temperature(A/D value)    :  2144
Temperature(degree)       :    28
VR(A/D value)             :   947
duty[%]                   :  23.0
```

rotation control は、
S コマンドで
StartUp(1)　回転数 1667rpm で磁界印加角度を動かす（始動制御）
Normal(2)　印加 duty に応じた回転数（通常）
が切り替わります。
diff angle -> speed は、現在の磁界印加角度増分から計算される回転数です。
（rotation speed は、ホールセンサの切り替わりタイミングから算出される回転数です。）

・'z'コマンドで表示される内容（チュートリアル C と同じ）

```
c1:pos:3,3p
c1:pos:3,2p
c1:pos:2,2p
c1:pos:2,6p
c1:pos:6,6p
```

z コマンドでは、モータ内蔵ホールセンサと疑似ホールセンサパターンの値を表示します（2ms 毎の読み取り）。
c1: CH-1 側である事を示す
pos:2,6p モータ内蔵ホールセンサ=2, 疑似ホールセンサパターン=6, 現在の制御=p 疑似ホールセンサパターンである事を示す（モータ内蔵ホールセンサを使用している場合は最後の文字は'h'）

・'x コマンドで表示される内容（CH-1 のみ）

```
c1:pos:2:deg:90(0)
c1:pos:6:deg:150(0)
c1:pos:4:deg:207(-3)
c1:pos:5:deg:268(-2)
c1:pos:1:deg:328(-2)
c1:pos:3:deg:27(-3)
c1:pos:2:deg:90(0)
c1:pos:6:deg:148(-2)
c1:pos:4:deg:207(-3)
c1:pos:5:deg:270(0)
c1:pos:1:deg:328(-2)
```

x コマンドは、センサ位置が切り替わった際の角度を表示します。

c1: CH-1 側である事を示す

pos:2 ホールセンサの切り替わり時の値（SW2 で選択した側のホールセンサ値）

deg:90(0) その時の角度が 90°，理想とのずれ 0°

'z', 'x'どちらのコマンドも、過去のチュートリアルで表示している内容です。

・チュートリアル BC での端子設定

→チュートリアル B2 に同じ

・チュートリアル BC での使用 stack

→チュートリアル B2 に同じ

以上で、チュートリアル編は終了となります。

チュートリアルで扱った内容をまとめたのが、サンプルプログラム(RA2T1_BLMKIT_SAMPLE)となります。サンプルプログラムに関しては、別の資料（「ソフトウェア　サンプルプログラム編」）に内容をまとめています。

2.6. 数値演算に関して

－三角関数(cos)の計算に関して－

相補 PWM のプログラムでは、制御周期(100us)毎に cos の計算を行って UVW 相の duty（UVW 分解）を計算しています。最近のマイコンでは TFU（三角関数をハードウェアで計算するユニット）を搭載しているものもあり、高速に cos, sin の計算ができます。RA2T1 には TFU も FPU も搭載されていませんので、RA2T1 のプログラムでは、初期化関数(blm_init)内で、0-360° の範囲で 1° 刻みで計算して cos, sin 値のテーブル化したデータを生成し、プログラム内で cos, sin の値はテーブル参照で利用しています。かつ、2.2 節で記載していますが、全て整数演算に置き換えて計算しています。

実際に三角関数の計算を使用している UVW 分解の関数でベンチマークを取ると以下の様な速度となりました。

関数名	RA2T1 テーブル化 整数演算 で計算	RA2T1 浮動小数 点を使用し た場合(*1)	RA2T1 浮動小数点 を使用した 場合(*2)	[参考] RX26T TFU で計算	備考
blm_angle_to_uvw_duty_sin	0.868ms	30.0ms	81.6ms	225us	デフォルトの正弦波駆動
blm_angle_to_uvw_duty_sin_post	0.868ms	30.0ms	81.6ms	225us	↑の後から duty を乗算
blm_angle_to_uvw_duty_sin_3harmonic	1.71ms	47.2ms	117ms	333us	3 倍高調波重畳
blm_angle_to_uvw_duty_sin_3harmonic_post	1.71ms	47.2ms	117ms	333us	↑の後から duty を乗算
blm_angle_to_uvw_duty1	0.628ms	22.8ms	59.0ms	184us	別バージョン 1
blm_angle_to_uvw_duty2	0.684ms	26.9ms	62.4ms	245us	別バージョン 2
blm_angle_to_uvw_duty2x	0.764ms	10.3ms	←	236us	別バージョン 2 の直線近似 （三角関数の計算未使用）

※ベンチマークは、上記関数を 0° ~360° まで 1° 刻みで 360 回計算した場合の実行時間

(*1)三角関数はテーブル化
(*2)三角関数は cosf 関数を使用

デフォルトの正弦波駆動(blm_angle_to_uvw_duty_sin)の場合

・テーブル化＋整数演算　0.868ms/360 回→1 回あたり 2.41us
・テーブル化＋浮動小数点演算　30ms/360 回→1回あたり 83.3us
・三角関数計算＋浮動小数点演算　108ms/360 回→1回あたり 266.7us

1 回あたりの変換時間は概ね上記となります。

100us 刻みで、相補 PWM の duty 値を更新する場合は、最大でも 100us 未満で UVW 分解値を算出する必要があります。A/D 変換や、角度の計算などもあるので、実際に UVW 分解に掛けられる時間はもっと短くなります。

RA2T1 マイコンでは、リアルタイムで三角関数の計算を行うのは現実的ではなく、また浮動小数点演算を使う場合はかなり最適化が必要であると思われます。

なお、TFU 搭載のマイコン（RX26T@120MHz）で TFU で計算した場合、RA2T1@64MHz でテーブル化＋整数演算よりも圧倒的に高速に三角関数の値を計算する事が出来ています。RA2T1 では、計算速度を考えるとテーブル化＋整数演算が現実的かと考えます。将来別なマイコンに置き換える場合は、TFU 搭載マイコンの使用を検討してみてください。

ー浮動小数点数の計算に関してー

FPU を持たない RA2T1 では、浮動小数点演算はそもそもできるだけ使用しない事が望ましいです。

使用する場合は、

```
float a;
...
a = a * 2.0;
```

上記の様なコードを書くと、コンパイル時

```
warning: conversion from 'double' to 'float' may change value [-Wfloat-conversion]
```

というワーニングが出ます。double 型の数値を float 型の変数に代入したので値が変わる（float で表現できない桁数の場合や double の精度が失われる）という内容で、気にしなければ、気にならない内容かもしれません。

ワーニングを消す目的で、

```
a = (float)(a * 2.0);       //基本的には違う
```

としたくなりますが、

```
a = a * 2.0f;
```

とするのが正解かと思います。

PC でのプログラムに慣れている方（組み込み系はあまり触らない方）なら、

```
2    （整数）
2.0   （浮動小数点数）
```

の使い分けは行うが、あえて

```
2.0f   （float 型の浮動小数点数）
```

を使うケースがあまりないかもしれません。

また、RX マイコンのプログラムに慣れている方では、RX では「RX231/RX651 等のマイコン」では、コンパイラ(CC-RX)オプションで、

double 型、及び long double 型の精度　単精度として扱う(-dbl_size=4)

がデフォルトになっています。

そのため、double と float どちらでも、4 バイト精度(float)で計算されます（float で扱うようなコードが生成されます）。
（a = a * 2.0　は float の演算として処理されます）

また、「RX72N」等のマイコンでは、倍精度浮動小数点命令を使用する(-dpfpu)がデフォルトで、double 型をハードウェアで計算する事ができるので、a= a * 2.0 の計算も高速に処理できます。

そのため RX では、2.0 と 2.0f を厳密に区別しなくても、それ程問題にならないケースが多いと思われます。

それに対し、RA マイコンは

float　4 バイト浮動小数点数
double　8 バイト浮動小数点数

となります。これは、double 型の変数を使った場合、計算が遅いというだけの話ではなく、float 型の変数 a を使った計算においても、

a = a * 2.0;

2.0 が double 型の定数として取り扱われるので、乗算の部分が double 型の演算となり、計算が遅くなります。

よって、上記の計算は

a = a * 2.0f;

である必要があります。

モータ制御の様なリアルタイム制御では、演算速度は重要な要素となりますので、double 型の精度が必要のない計算は、2.0f の様に f サフィックスを付ける様にしてください。

double 型を使用した処理は、単純な計算でもかなりのボリュームとなります。また、float 型を使った場合でも、相当なボリューム（命令数）となるので、

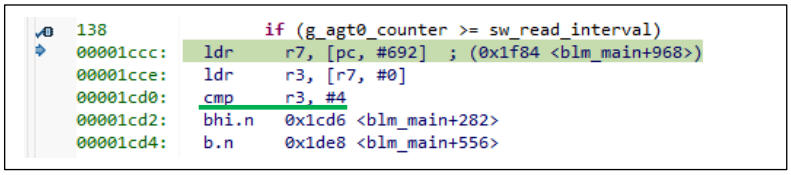
・double 型は特段の理由がない限り使用しない
・float 型も基本的には使用しない
事が求められるかと思います。
（FPU 搭載のマイコンだと、fadd の様な 1 命令で処理される内容が、RA2T1 での浮動小数点演算はかなり重たい処理になるという認識が必要です。）

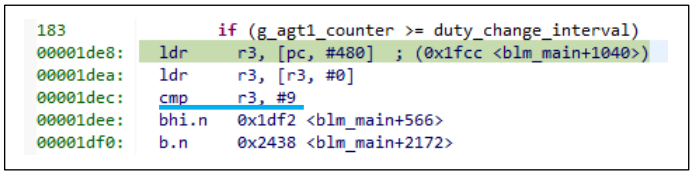
なお、プログラム内で double の演算を行っている箇所があります。

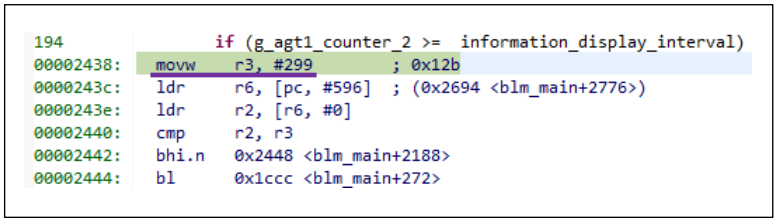
・blm_main.c

```
    const unsigned long sw_read_interval = (unsigned long)(500e-6 / BLM_CONTROL_PERIOD);
//500us 毎にスイッチの状態をスキャン（100us:AGT0 で何カウントか）                    計算結果=5
    const unsigned long duty_change_interval = (unsigned long)(0.1 / BLM_COUNTER_PERIOD);
//0.1[s]毎（10ms:AGT1 で何カウントか）                                          計算結果=10
    const unsigned long information_display_interval = (unsigned long)(3.0 / BLM_COUNTER_PERIOD);
//3 秒毎に画面に情報を表示（10ms:AGT1 で何カウントか）                              計算結果=300
```

上記の様なプログラムをコンパイルすると、生成される機械語のコード（RA の実行形式）は以下の様になります。







500e-6 / 100.0e-6 は、コンパイル時に PC 上で計算されます。計算結果のループ回数判定の"4"(=5-1)がプログラム内に埋め込まれて（＝ROM 上にデータが格納）、プログラムで使用される動作となります。RA マイコンが、500e-6 / 100.0e-6 を計算する訳ではありません。

（数値演算に関しては、リアルタイムで処理される部分では、極力浮動小数点の演算使用しない事が求められます。起動時に 1 回のみ実行される部分や、コンパイラで定数に置き換えられる部分に関しては、使用しても特に問題はないと考えます。）

（チュートリアルのプログラムでは、3 秒に 1 回実行される画面表示のタイミングでは、浮動小数点の演算を使用しています。100us や 10ms の周期で実行されるプログラムコード内では、浮動小数点の演算を使用していません。）

取扱説明書改定記録

バージョン	発行日	ページ	改定内容
REV.2.0.0.0	2025.9.30	—	初版発行

お問合せ窓口

最新情報については弊社ホームページをご活用ください。

ご不明点は弊社サポート窓口までお問合せください。

株式会社 北斗電子

〒060-0042 札幌市中央区大通西 16 丁目 3 番地 7

TEL 011-640-8800　FAX 011-640-8801

e-mail：support@hokutodenshi.co.jp (サポート用)、order@hokutodenshi.co.jp (ご注文用)

URL:https://www.hokutodenshi.co.jp

商標等の表記について

- 全ての商標及び登録商標はそれぞれの所有者に帰属します。
- パーソナルコンピュータを PC と称します。

ルネサス エレクトロニクス RA2T1(QFP-48 ピン)搭載
ブラシレスモータスタータキット

ブラシレスモータスタータキット(RA2T1)
[ソフトウェア　チュートリアル編]
取扱説明書

株式会社 北斗電子

©2025 北斗電子 Printed in Japan 2025 年 9 月 30 日改訂 REV.1.0.0.0 (250930)