CAN マルチネットワークボード
取扱説明書　開発環境編

ルネサス エレクトロニクス社　RX231, RL78/F15, RA2L1 搭載
HSB シリーズ応用キット

-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

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。

1. 概要

本書では、CAN マルチネットワークボードのソフトウェア開発環境に関して記載します。

2. プログラムの書き込み

2.1. HSB_CAN_MULTI ボード

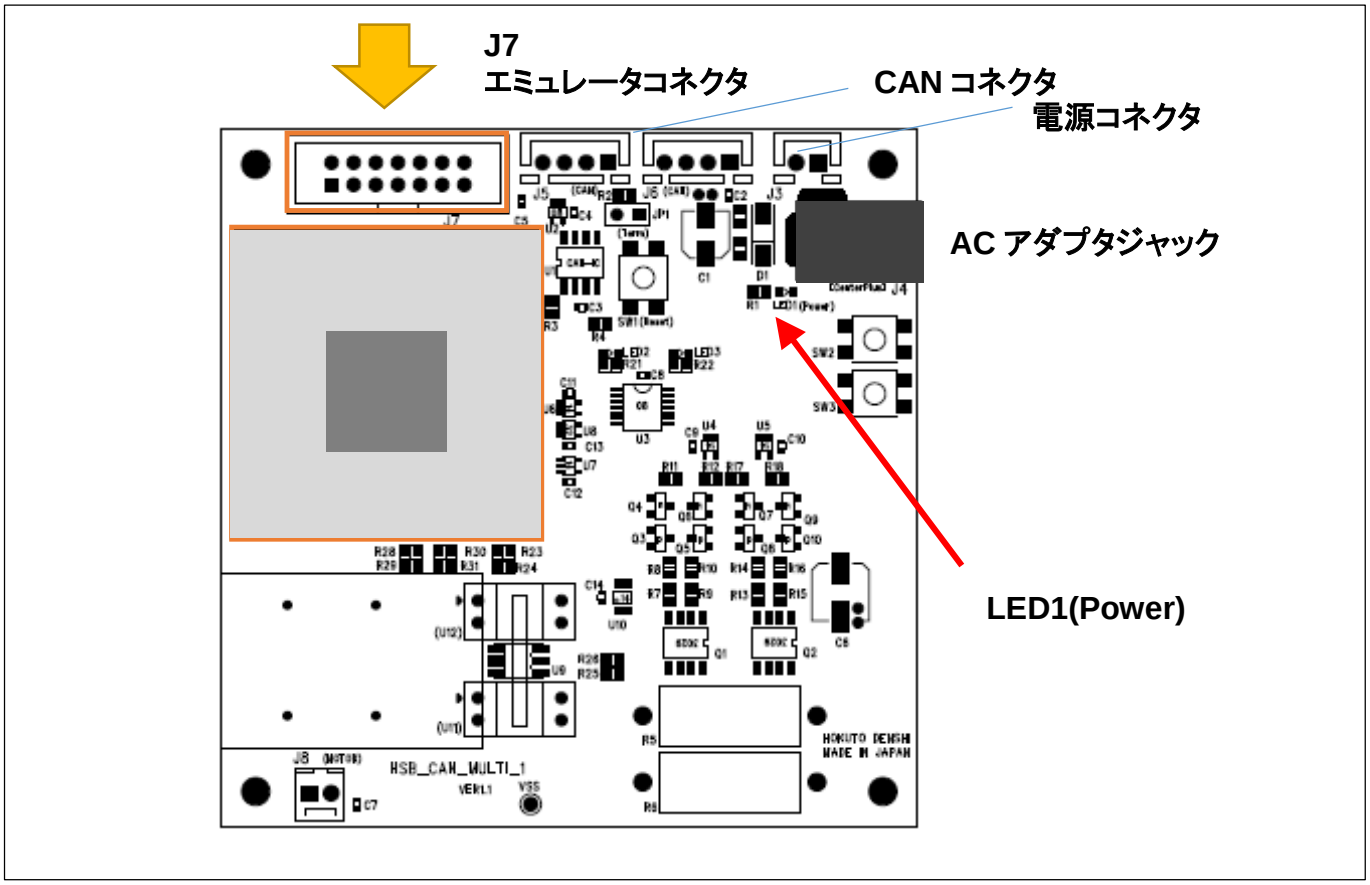
プログラムの書き込みは、

・ルネサス製 E2 エミュレータ
・ルネサス製 E2Lite エミュレータ
・ルネサス製 E1 エミュレータ　※ターゲットマイコンが RX231, RL78/F15 の場合
・ルネサス製 E20 エミュレータ　※ターゲットマイコンが RX231, RL78/F15 の場合　※電源が別途必要
・MULTI_WRITER（当社製品）

のいずれかの機器が必要となります。

HSB_CAN_MULTI ボードに差し込まれている CPU カードが RA2L1 の場合は、E1, E20 は使用できません。

プログラムの書き込みは、ボードの J7（エミュレータコネクタ）経由で行います。



（上記ボードは、HSB_CAN_MULTI_1 ですが、他のボードも同様です）

書き込み時は、ボードに対して電源を印加する必要があります。

・AC アダプタを接続
・電源コネクタに電源印加
・CAN コネクタ経由で電源印加
・USB 経由で電源印加（HSB_CAN_MULTI_4 の場合）

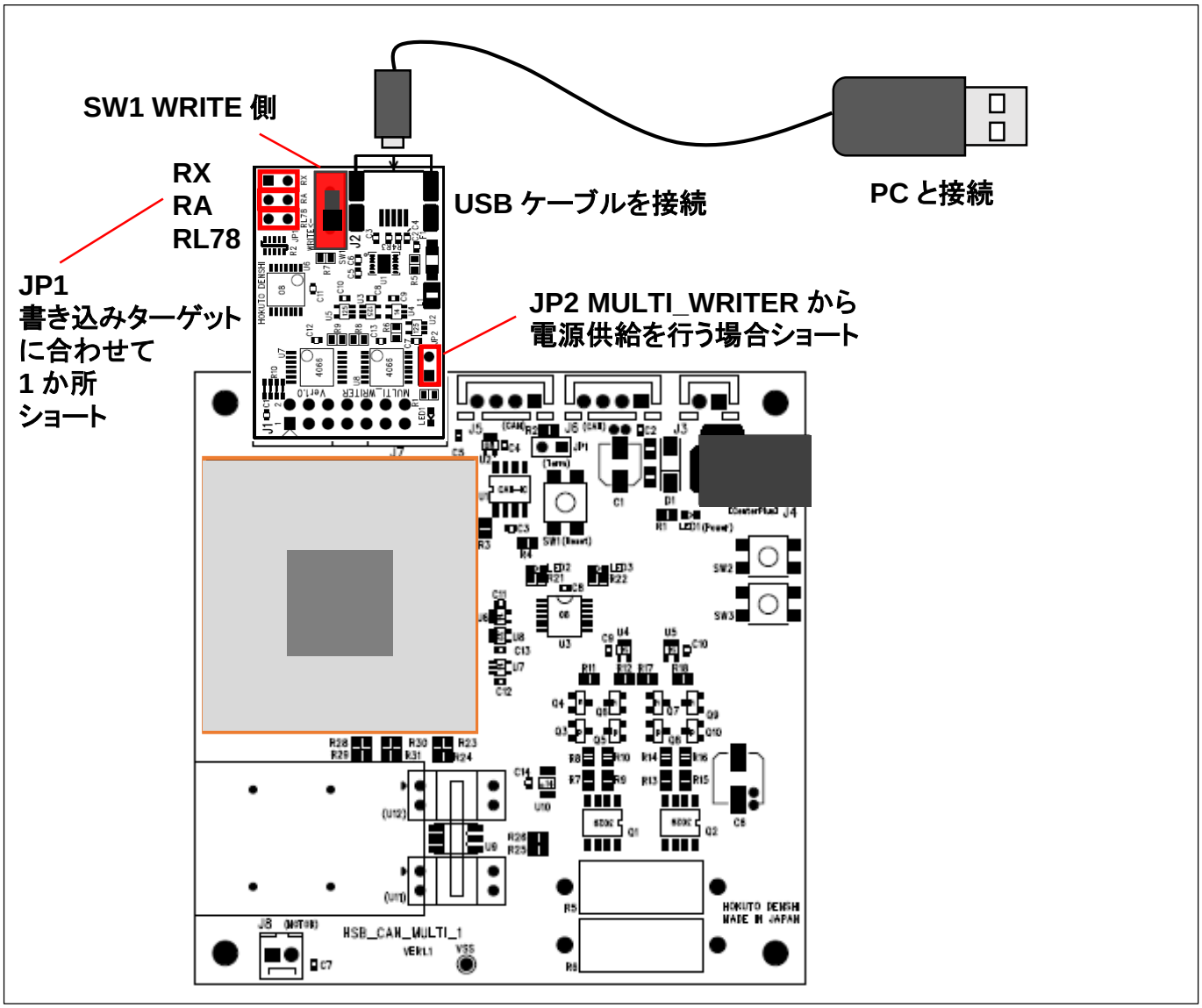
上記、いずれかの経路でボードに電源が印加されている場合は、書き込み機器からの電源供給は行わない状態で書き込みを行います。

ボードに対して電源供給の経路が無い場合は、書き込み機器から電源を印加する設定とします。

書き込み機器接続前に、ボード上の LED1 が点灯している場合は、ボードに電源が供給されています。LED1 の点灯状態を確認して、書き込み機器から電源供給を行うかどうかを決めてください。

なお、E20 エミュレータ使用時は、外部から電源供給を行う必要があります。

－MULTI_WRITER を接続する場合－

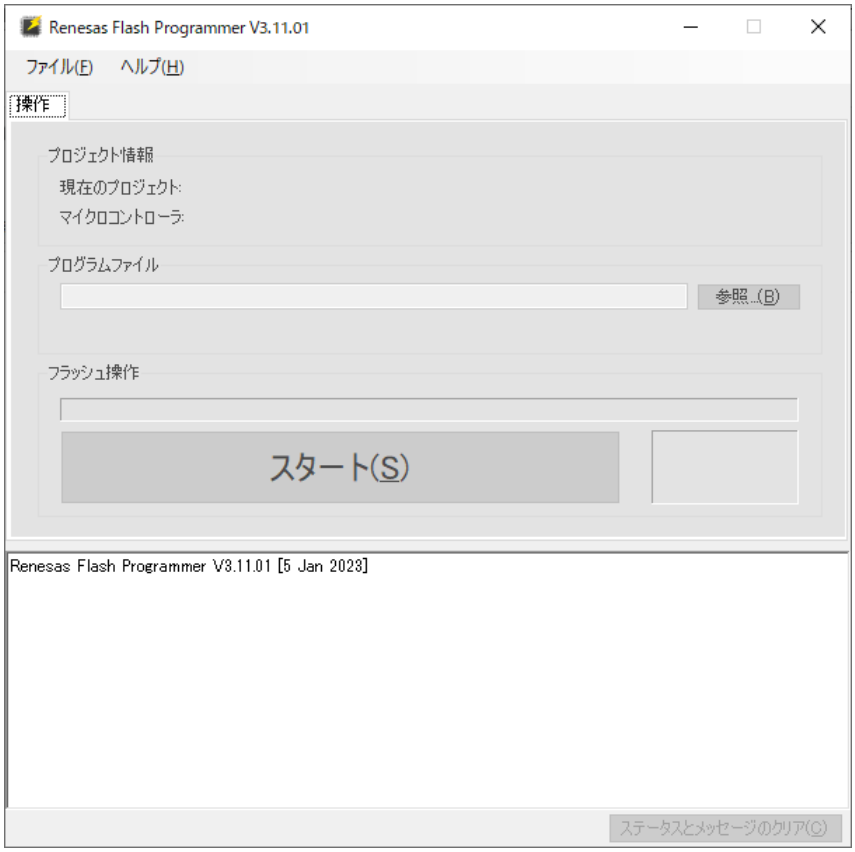


MULTI_WRITER を接続する場合で、MULTI_WRITER から電源供給を行う場合は、JP2 をショートに設定してください。

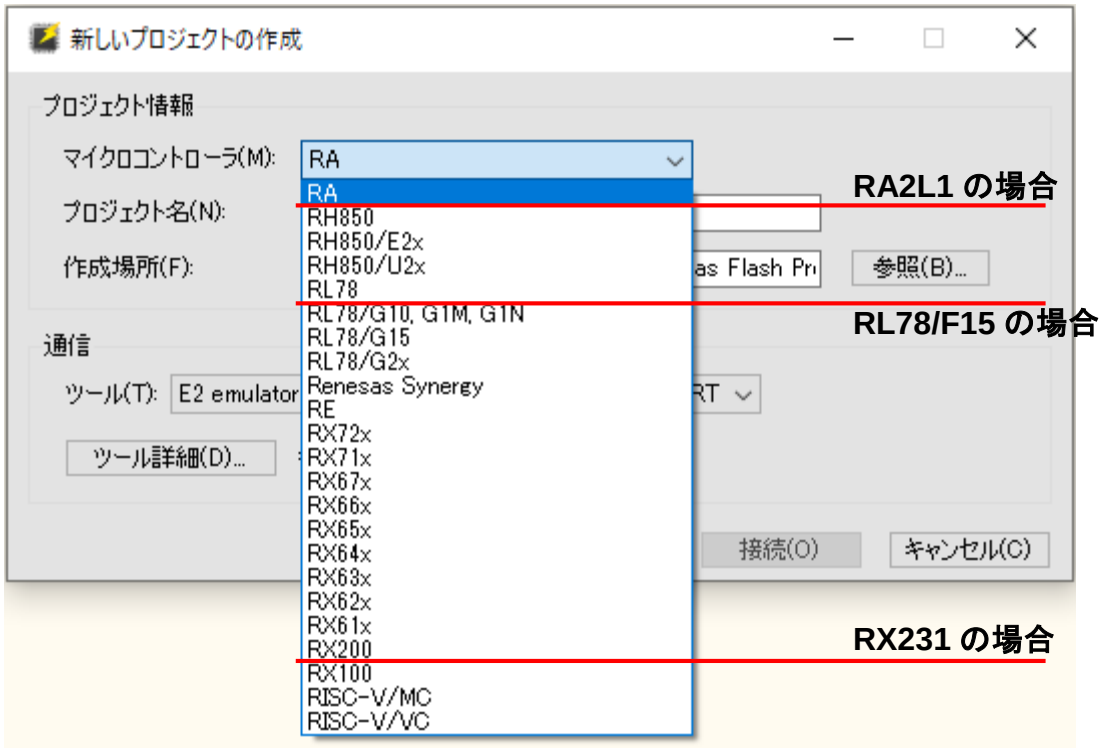
SW1 は、WRITE 側（下側）、JP1 は 3 つあるジャンパの内、書き込みターゲットのジャンパのみショート（6 ピンあるジャンパの内、1 か所のみショートジャンパを挿す）としてください。

ルネサス製エミュレータを接続する場合は、J7 の 14 ピンコネクタに、エミュレータのケーブルを挿してください。E2 エミュレータの場合は、E2 付属の 14 ピン変換コネクタ経由で接続してください（変換コネクタのスイッチは、ターゲットが RL78/F15 の場合は RL78 側、それ以外は Other 側）

書き込みターゲットボード(HSB_CAN_MULTI_n)と、PC を接続した後、RenesasFlashProgrammer を起動してください。



ファイルー新しいプロジェクトを作成



マイクロコントローラの選択画面となりますので、HSB_CAN_MULTI ボードに差し込まれている、CPU カードのタイプに応じて、

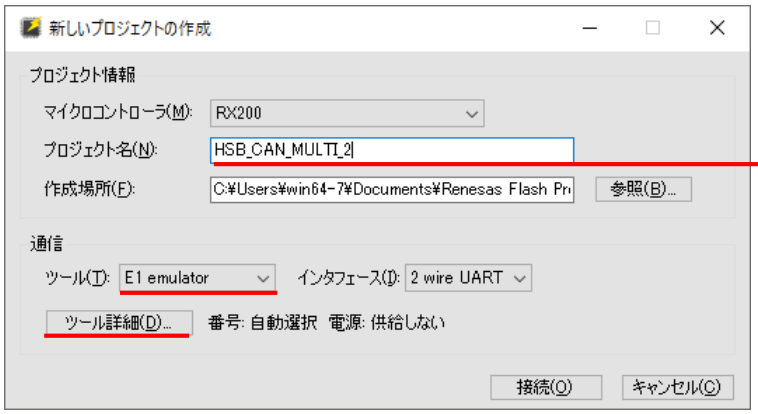
「RA」

「RL78」

「RX200」

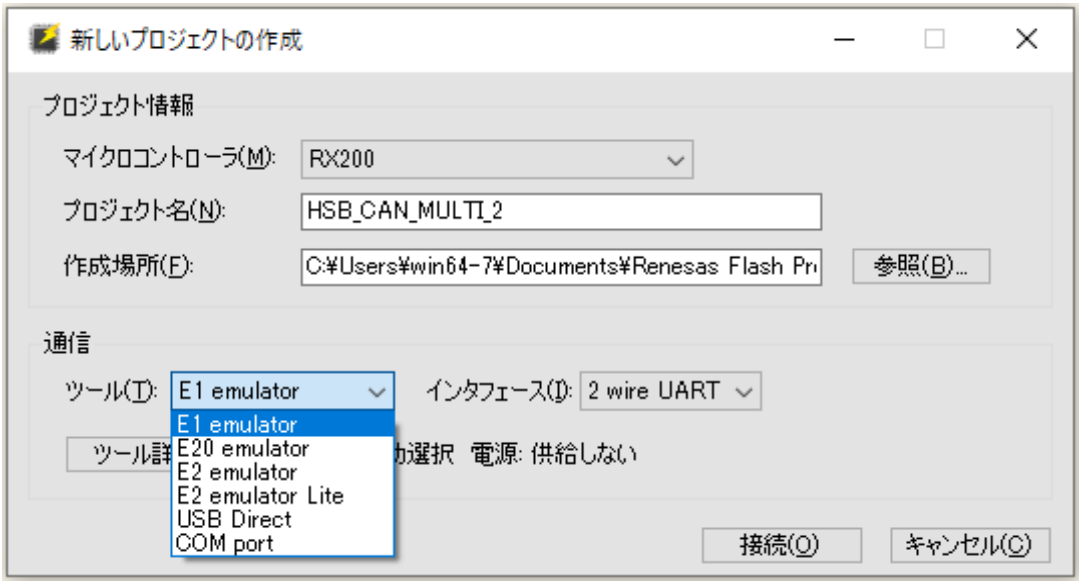
のいずれかを選択してください。

（以下、RX231 をターゲットにした場合の画面キャプチャを示します）



適当なプロジェクト名を入力

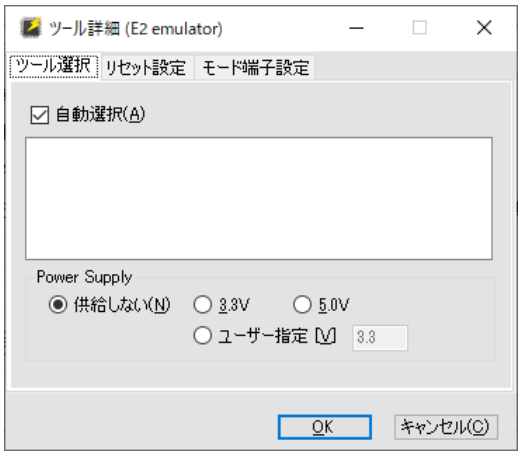
次にボード接続した書き込み機器に応じて、ツールのところを選択してください。



E1/E20/E2/E2Lite の場合は、Ex emulator の項目を選択してください。MULTI_WRITER を使用する場合は、「COM port」を選択してください。

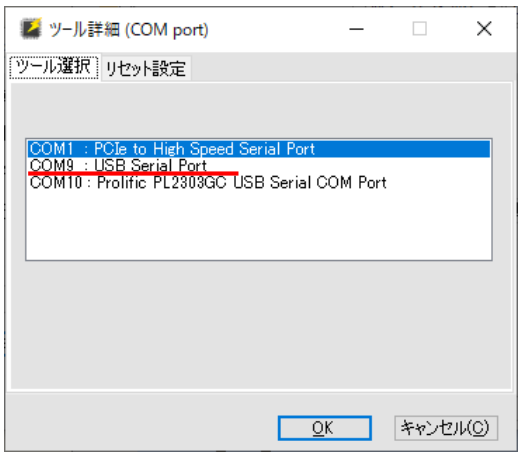
次に、ツール詳細ボタンを押してください。

E2 emulator を選択した場合は、

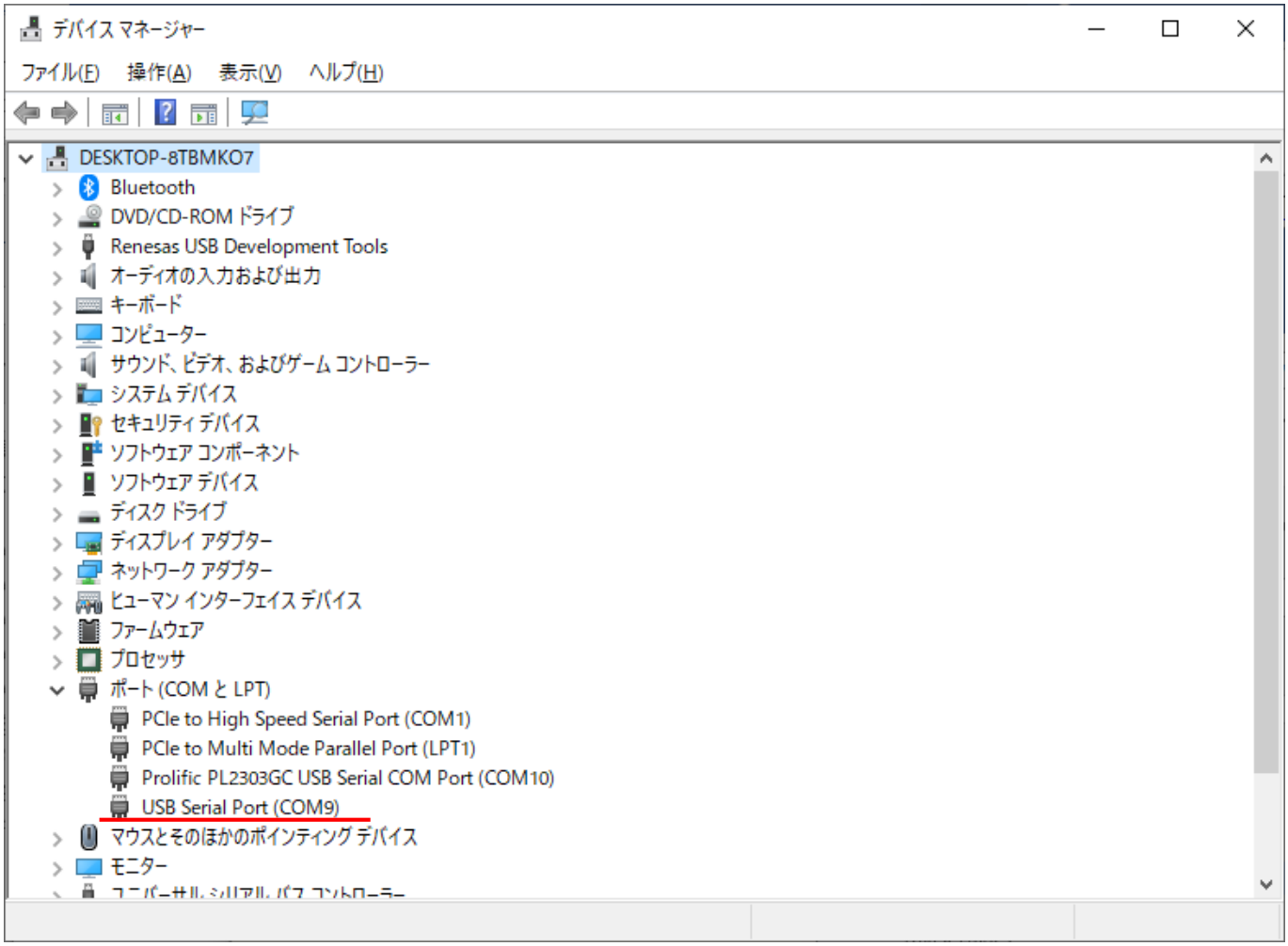


上記の画面となりますので、Power Supply の欄を選択してください。ボードに外部から電源が供給されている場合は「供給しない」（デフォルト）です。エミュレータから電源供給を行う場合は、「5.0V」を選択してください。
（E2 Lite では、3.3V を選択してください）

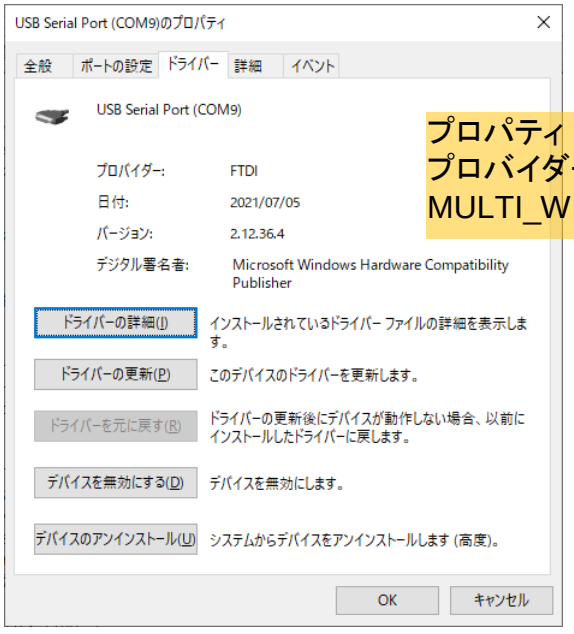
[MULTI_WRITER を使用する場合] COM Port を選択した場合、



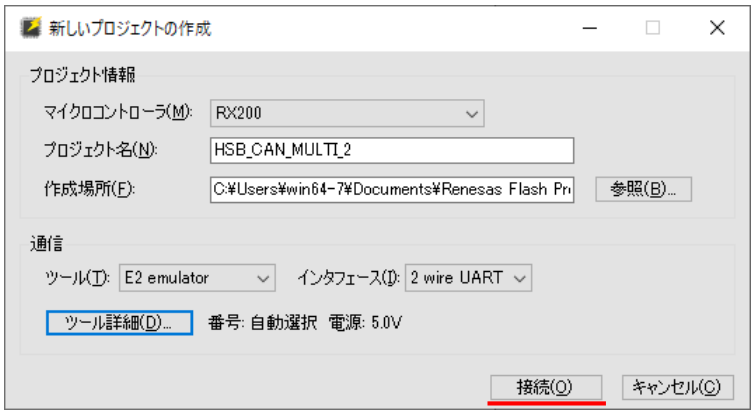
COM 番号の選択画面となりますので、この例では COM9(USB Serial Port)を選択して、OK を押してください。



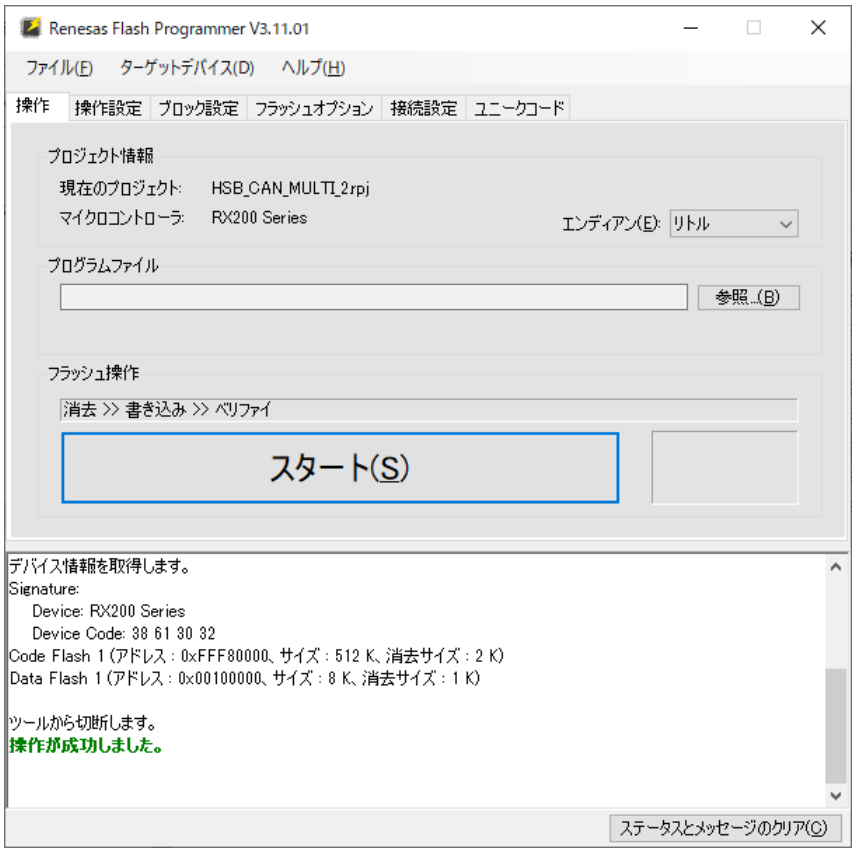
MULTI_WRITER は、PC 上では COM ポートと認識されますが、番号は環境により異なります。デバイスマネージャを起動して、MULTI_WRITER を抜き差しした際に、見えなくなるデバイスが MULTI_WRITER の COM ポート番号です。表示される名称は、Windows10/11 の場合(USB Serial Port)となります。



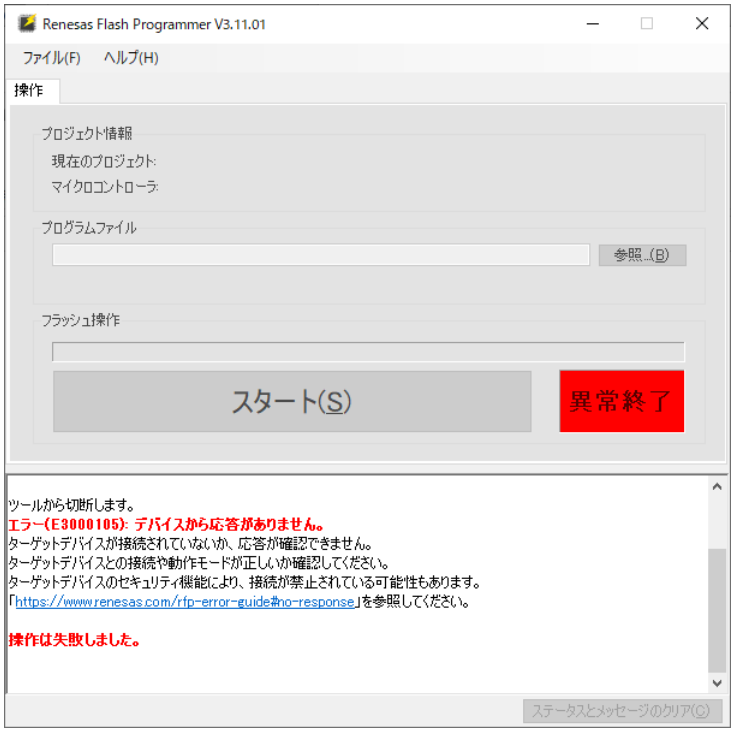
プロパティ「ドライバ」タブで見るとプロバイダー「FTDI」となっているデバイスが、MULTI_WRITER です



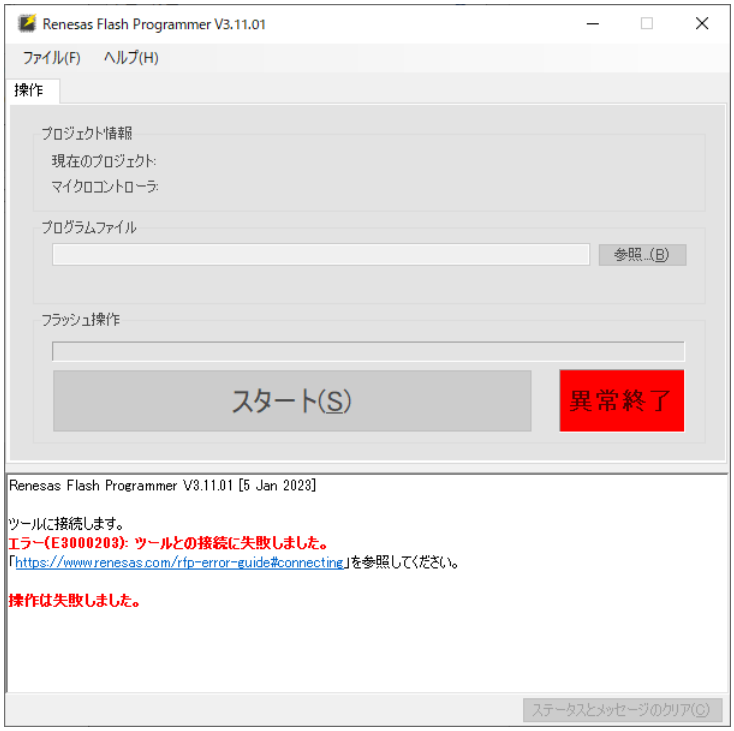
ツール詳細の設定が終わったら、次に「接続」ボタンを押します。



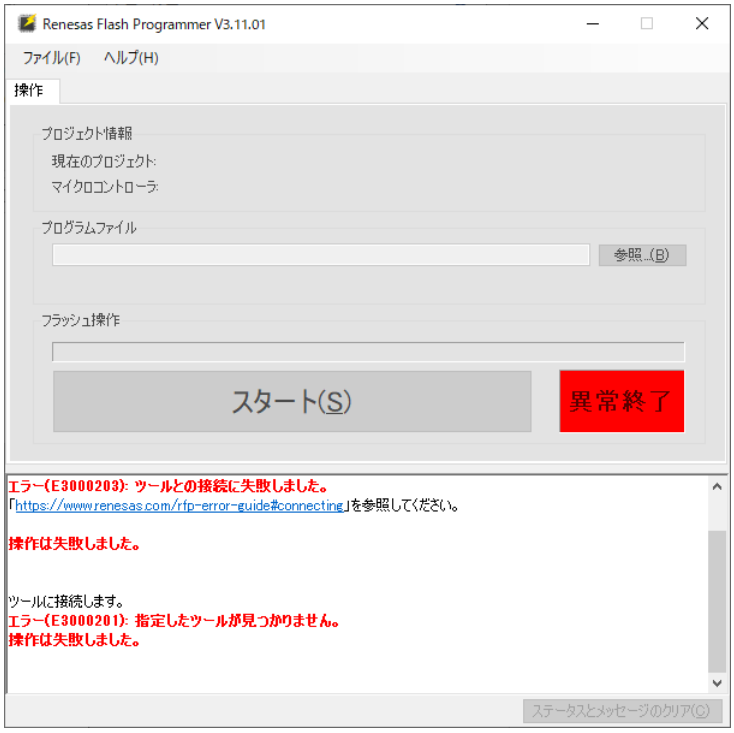
RenesasFlashProgrammer がマイコンと通信を行い、デバイス情報取得の後、「操作が成功しました」という表示となれば OK です。



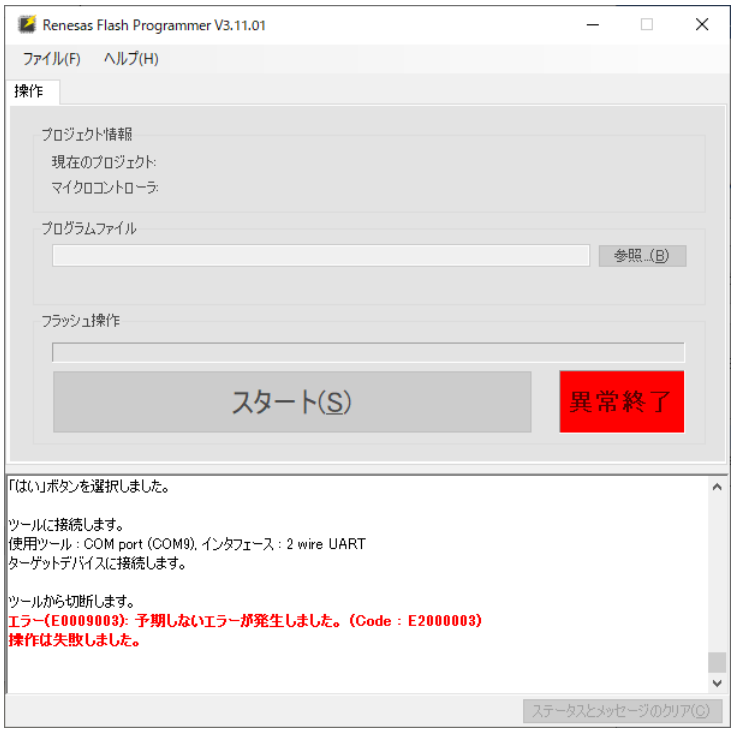
デバイスから応答がありませんというエラーとなった場合は、「マイクロコントローラの選択が正しいか」「MULTI_WRITER の COM ポートが間違えていないか」「エミュレータや MULTI_WRITER が PC, HSB_CAN_MULTI ボードと接続されているか」を確認してください。



ツールとの接続に失敗しましたというエラーになった場合は、MULTI_WRITER の COM ポート（この例では COM9）で Teraterm 等の端末が開いていないかを確認してください。

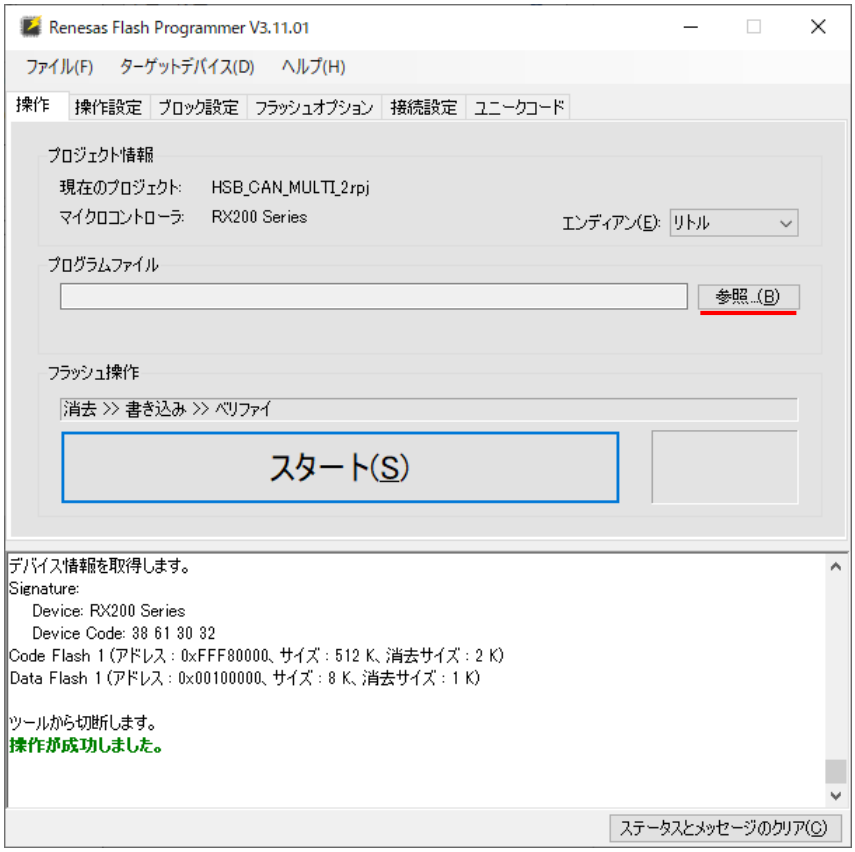


指定したツールが見つかりませんというエラーの場合は、接続しているエミュレータと選択したエミュレータが異なっていないかを確認してください。

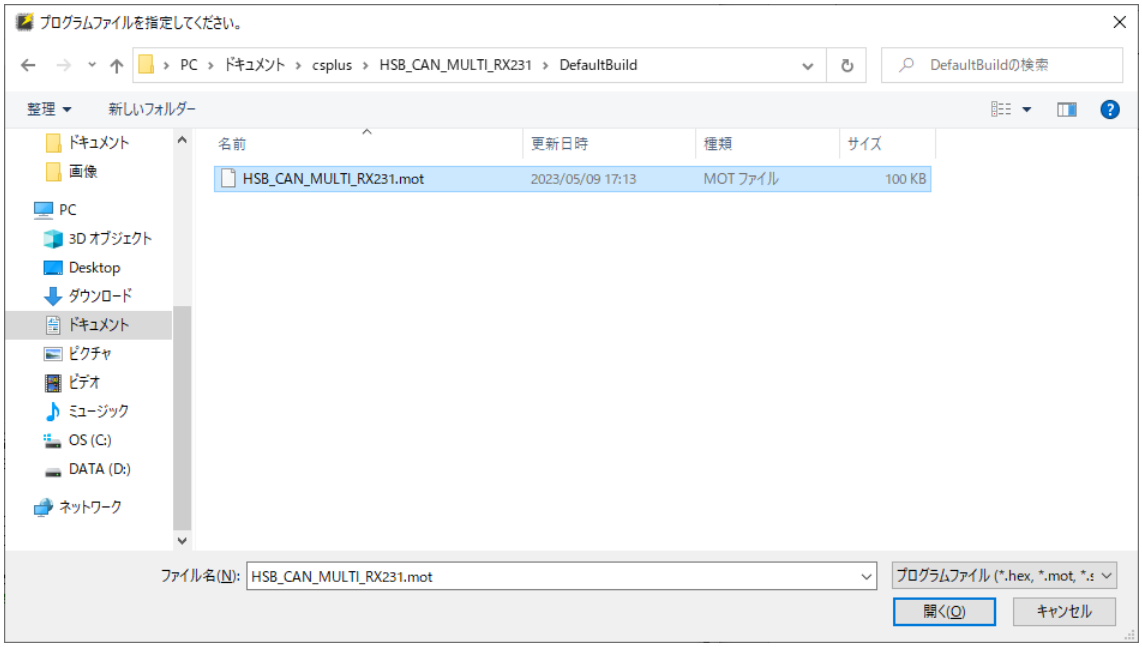


予期しないエラーが発生しました。もしくは、接続中にフリーズ（接続試行が終わらない）場合は、マイクロコントローラの選択に誤りがある事が考えられます。
（接続中にフリーズした場合は、USB ケーブルを抜いてください）

接続に成功した場合、

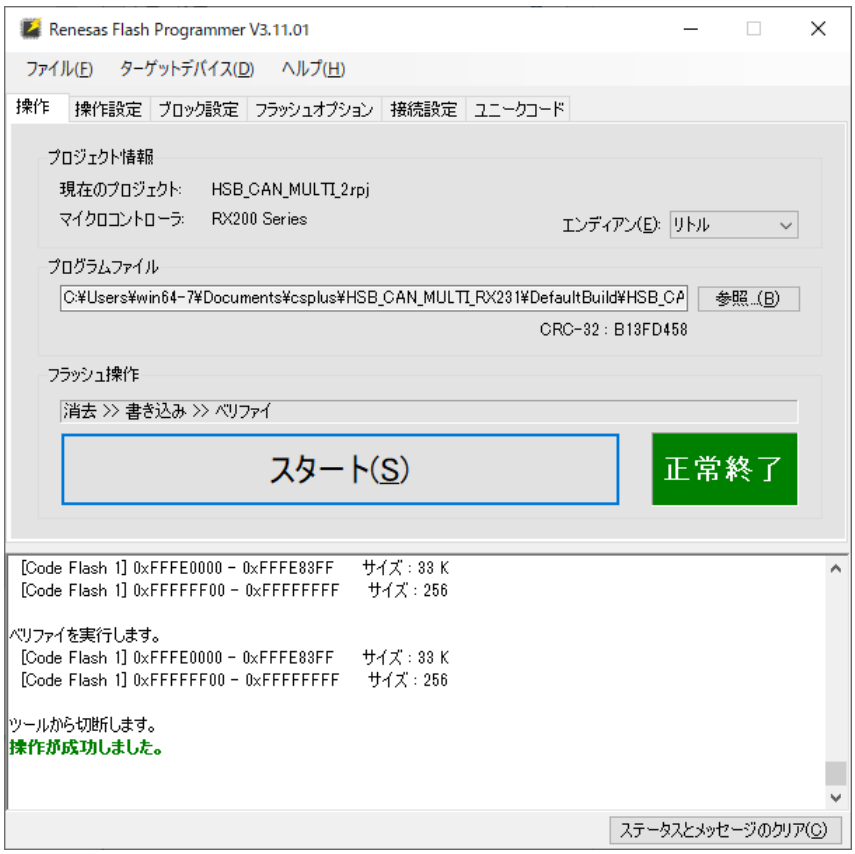


プログラムファイル「参照」のボタンを押してください。



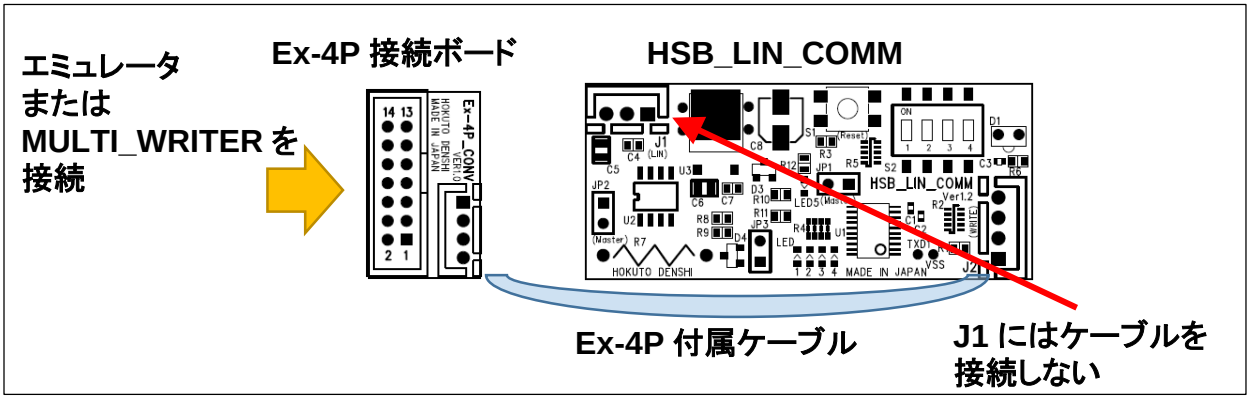
書き込む mot ファイル（srec ファイル）を選択します。

その後、「スタート」ボタンを押してください。



書き込み、ベリファイが実行され、「操作が成功しました。」というメッセージが表示されれば、プログラムの書き込みは成功です。

2.2. HSB_LIN_COMM ボード

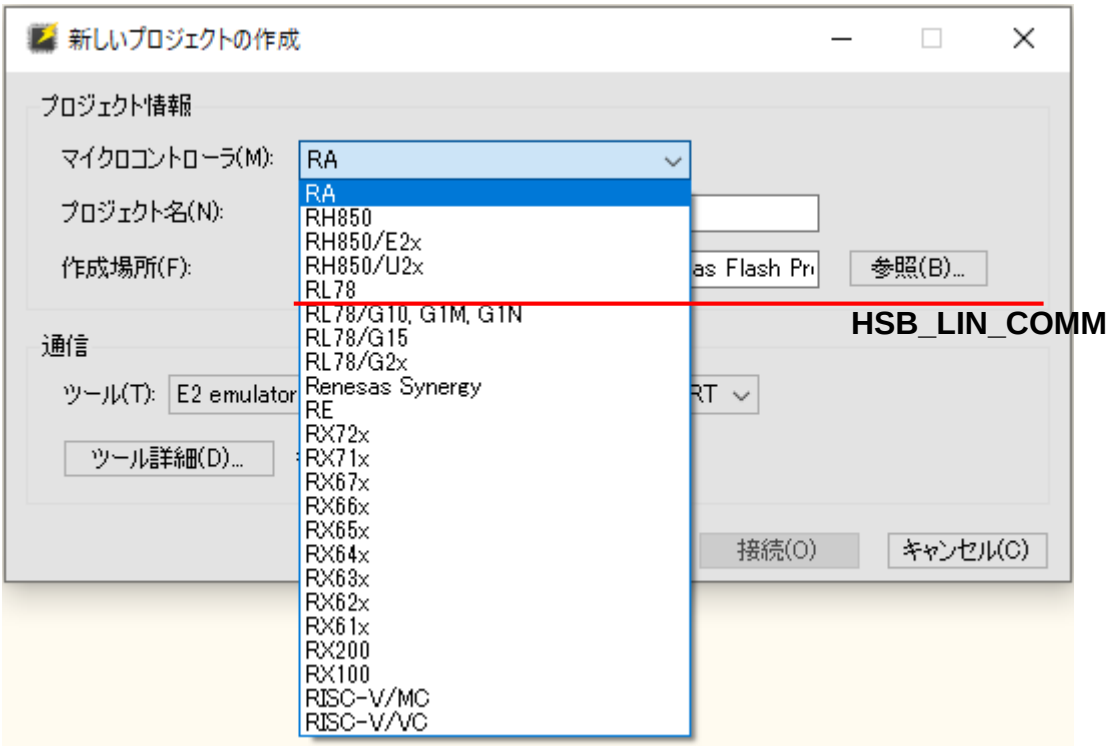


HSB_LIN_COMM ボードに対して書き込みを行う場合は、別売の Ex-4P 接続ボードが必要です。Ex-4P 接続ボードには、14 ピンコネクタが搭載されていますので、E2/E2Lite/E1 もしくは MULTI_WRITER を接続してください。

HSB_LIN_COMM に対する電源供給は、基本的にはエミュレータか MULTI_WRITER からの供給になります。（MULTI_WRITER を使用する場合は、JP2 をショートで使用してください）

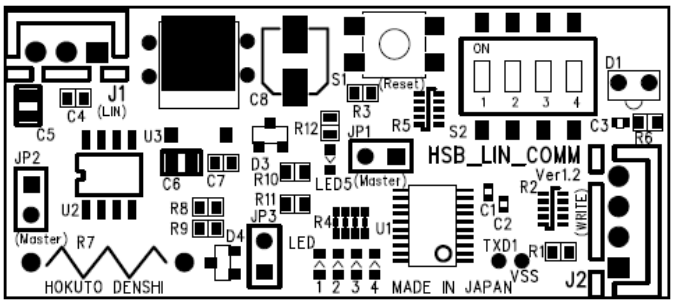
RenesasFlashProgrammer を使用して書き込むのは HSB_CAN_MULTI ボード同様です。

ファイルー新しいプロジェクトを作成



マイクロコントローラは「RL78」を選択してください。それ以降の操作は、HSB_CAN_MULTI ボードと同様です。

－HSB_LIN_COMM の書き込み端子の信号接続に関して－



HSB_LIN_COMM 側の書き込み端子(J2)のピン配置は上記となっています。

No	信号名	No	信号名
1		2	VSS
3		4	
5	TOOL0	6	
7		8	VDD
9	VDD	10	*RESET
11		12	VSS
13	*RESET	14	VSS

ルネサスのエミュレータの端子接続は上記の様になっていますので、同じ信号名の端子を接続すれば、Ex-4P 変換ボードを使用しなくても書き込みが可能です。（ケーブルやコネクタを自作する方は、参考にしてください。）

※E20 はエミュレータからの電源供給ができませんので、E20 を使用しての書き込みは基本的には不可です
（E20 しか手元にないという場合は、HSB_LIN_COMM の J1 に VSUP=7~18V を印加すれば書き込みは可能です）

3. プログラムのビルド

HSB_CAN_MULTI_n(n=1~4), HSB_LIN_COMM 向けのソースコードに関しては、「CAN マルチネットワーク ソースコード CD」として販売しております。当該 CD 内には、

(1)HSB_CAN_MULTI(CPU_CARD_RX231-48)向け CS+プロジェクト
(2)HSB_CAN_MULTI(CPU_CARD_RL78F15-64)向け CS+プロジェクト
(3)HSB_CAN_MULTI(CPU_CARD_RA2L1-64)向け e2studio プロジェクト（アーカイブ）
(4)HSB_LIN_COMM 向け CS+プロジェクト
(5)PC 向けデモプログラム(HSB_CAN_MULTI_DEMO.exe) VisualStudio(C#)向けプロジェクト
(6)PC 向けデモプログラム(HSB_CAN_MULTI_DEMO2.exe) VisualStudio(C#)向けプロジェクト

が含まれます。

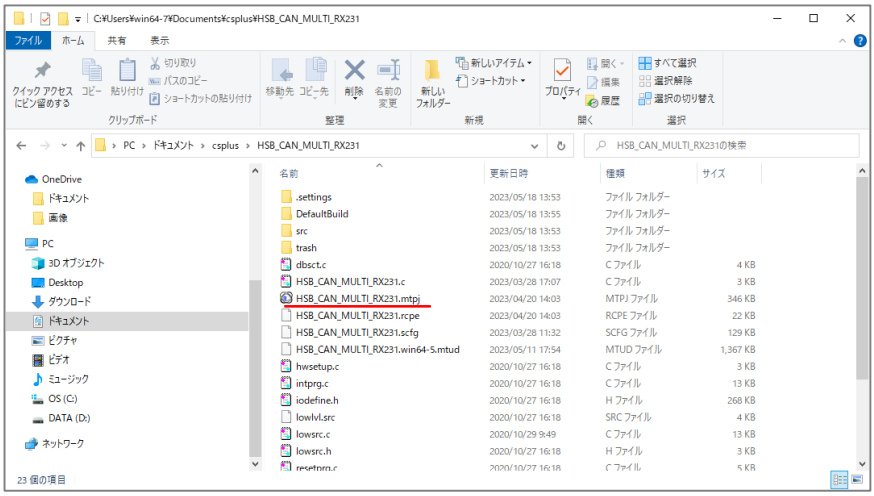
－CD 構成－

フォルダ		
HSB_CAN_MULTI¥	HSB_CAN_MULTI_RX231¥	(1) CS+プロジェクトフォルダ
	HSB_CAN_MULTI_RL78F15¥	(2) CS+プロジェクトフォルダ
	HSB_CAN_MULTI_RA2L1.zip	(3) e2studio アーカイブファイル
HSB_LIN_COMM¥	RL78_G11_LIN_COMM¥	(4) CS+プロジェクトファイル
PC¥	HSB_CAN_MULTI_DEMO¥	(5) VisualStudio 向けプロジェクトフォルダ
	HSB_CAN_MULTI_DEMO¥	(6) VisualStudio 向けプロジェクトフォルダ

3.1. プロジェクトの開き方

－CS+プロジェクト－

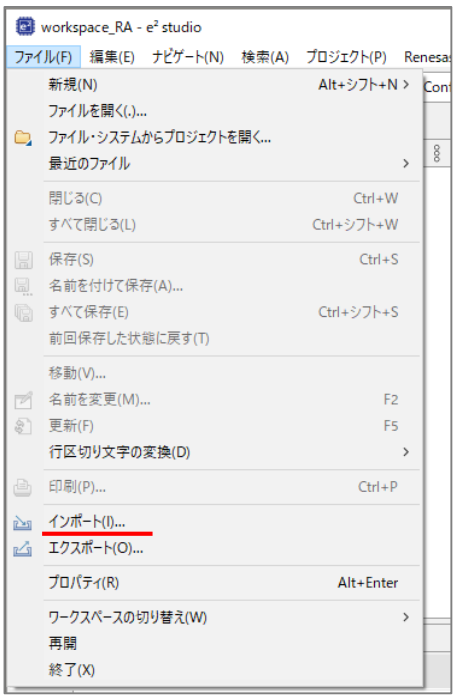
CS+プロジェクトフォルダは、フォルダごと PC のストレージ上にコピーを行い、mtpj ファイルをダブルクリックして、プロジェクトを開いてください。CS+forCC（Ver8.09 以降）が必要になりますので、予めインストールをお願い致します。



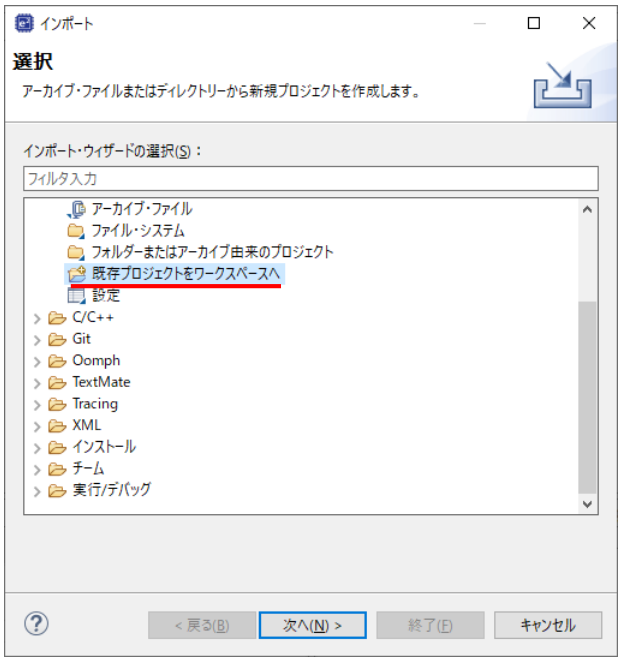
ーe2studio アーカイブー

予め、RA マイコン向け FSP がセットになった e2studio をインストール願います。（FSP4.3 以降）

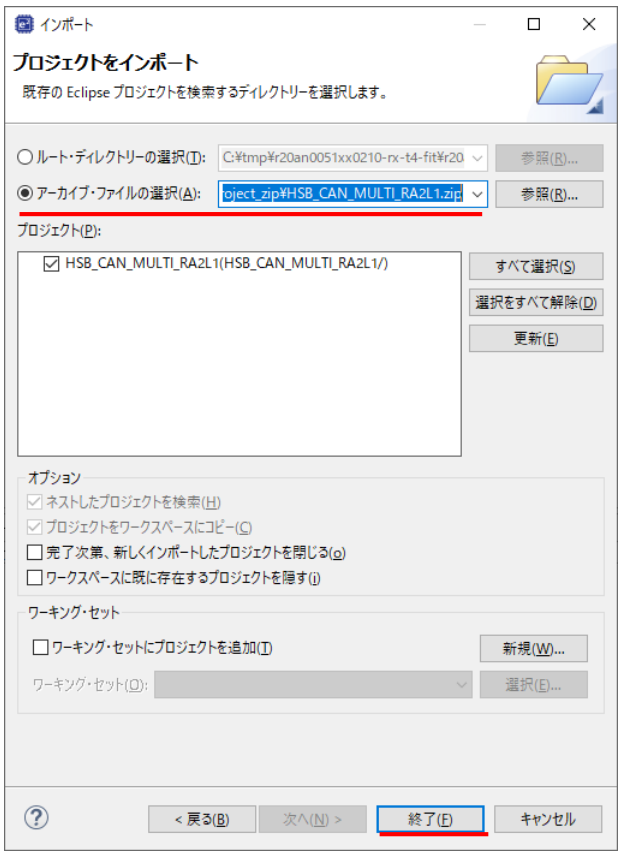
e2studio 向けのアーカイブファイルは、zip 形式となっているので、ワークスペースにインポートして、プロジェクトを開いてください。



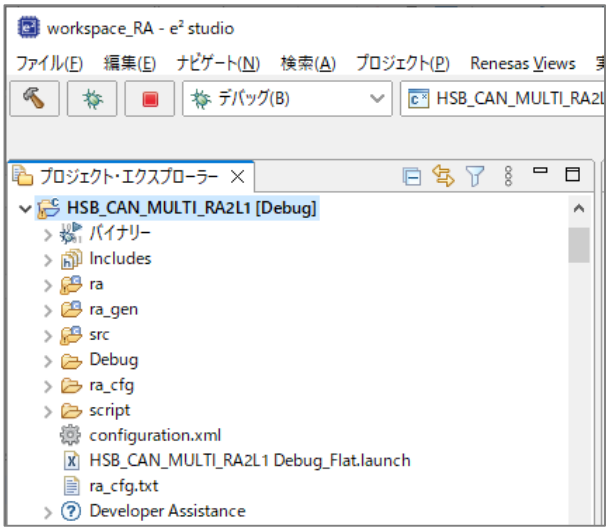
ファイルメニュー「インポート」



「既存プロジェクトをワークスペースへ」を選択。



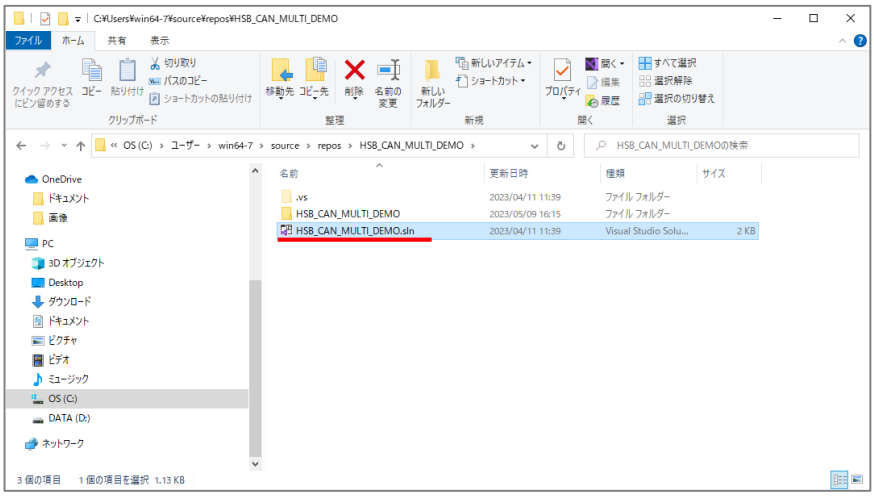
「アーカイブ・ファイルの選択」の方を選択して、ファイルとして、CD に収納されている
HSB_CAN_MULTI_RA2L1.zip
を選択して、「終了」を押す。



ワークスペース上に、HSB_CAN_MULTI_RA2L1 というプロジェクトができていれば、インポートは成功しています。

ーVisualStudio(C#)向けプロジェクトー

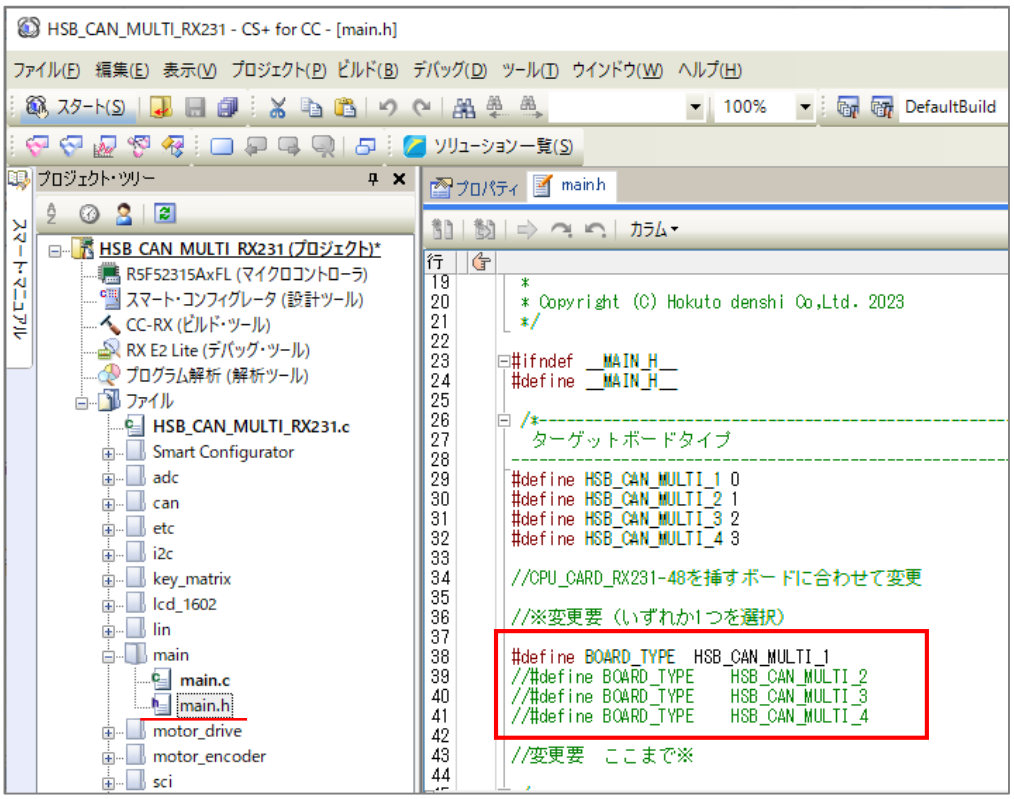
VisualStudio プロジェクトフォルダは、フォルダごと PC のストレージ上にコピーを行い、sln ファイルをダブルクリックして、プロジェクトを開いてください。VisualStudio（2019 以降）が必要になりますので、予めインストールをお願い致します。



3.2. ファイル変更箇所

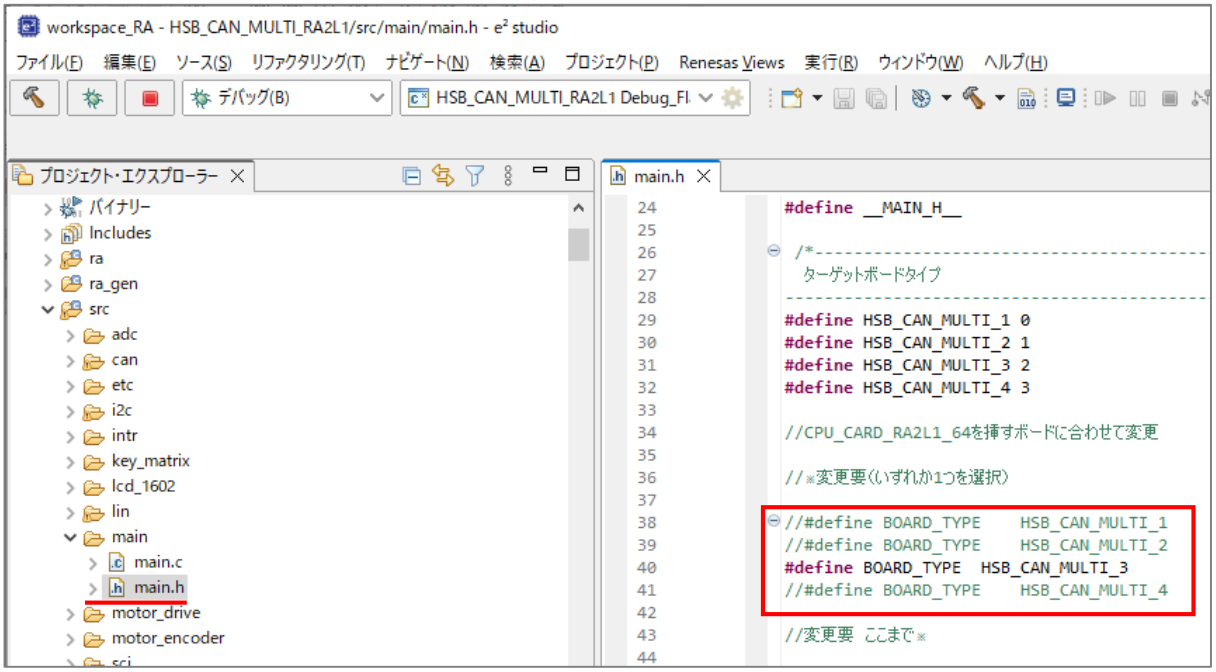
HSB_CAN_MULTI ボードは、4 枚のボードが 1 つのプロジェクトとなっていますので、ターゲットボードを指定する箇所があります。

ーCS+プロジェクトー



main フォルダ内の main.h 内に、ターゲットボードの選択箇所があります。赤枠で囲まれている部分の 4 行の内 1 行のコメントアウト（行頭の//）を外して、ターゲットボードを選択してください。

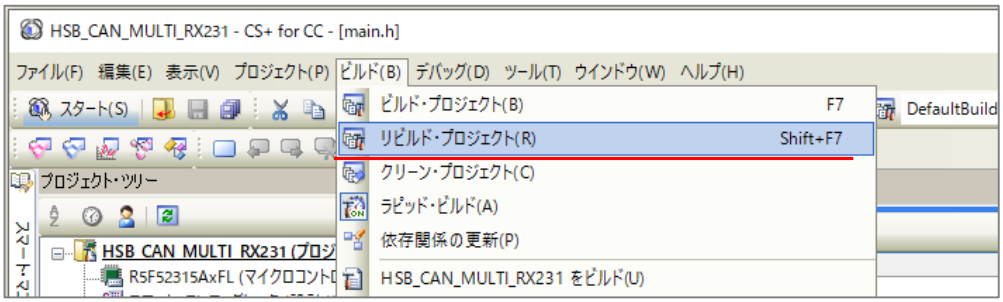
－e2studio プロジェクト－



e2studio(RA2L1)の場合、src フォルダ内の main¥main.h の変更が必要です。

3.3. プログラムのビルド

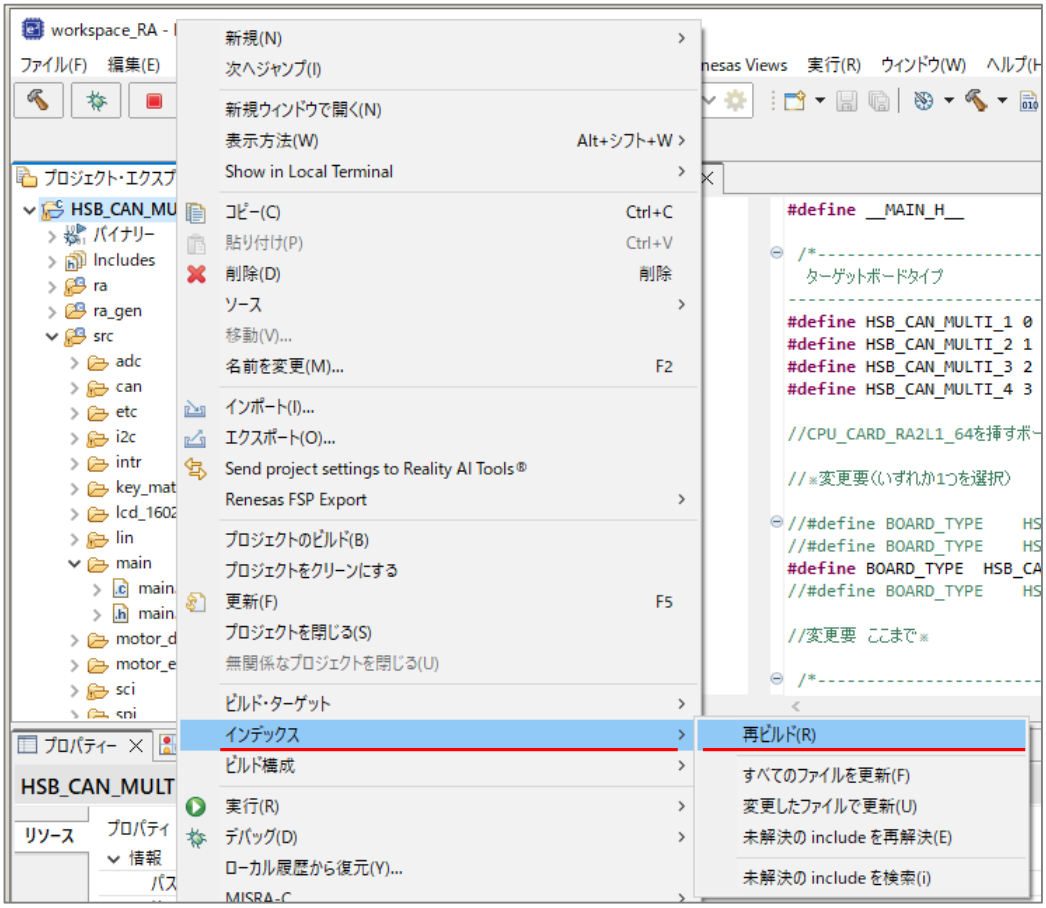
－CS+プロジェクト－



メニューの「ビルド」－「リビルド・プロジェクト」を実行してください。

※ソースコードの変更であれば「ビルド・プロジェクト」の実行でも問題ありません。ボード定義を変更した場合は、「リビルド」が無難です。

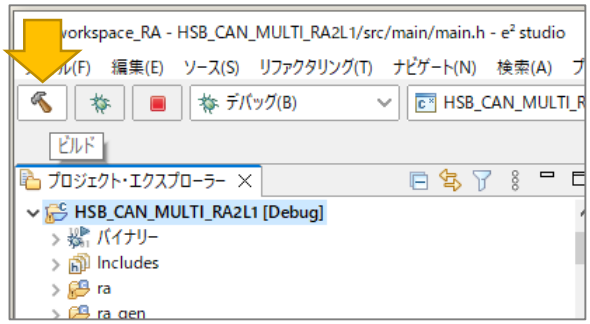
－e2studio プロジェクト－



プロジェクト名（HSB_CAN_MULTI_RA2L1)を右クリック

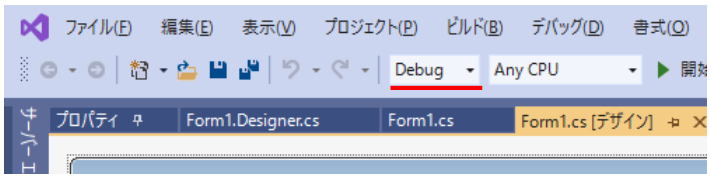
インデックス－再ビルド

※ターゲットボードタイプの変更や、定数定義を変更した場合に実行してください。ソースコードの変更のみであれば、本操作は不要です。

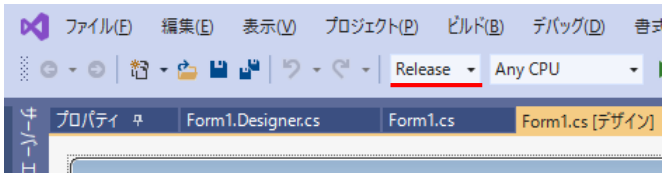


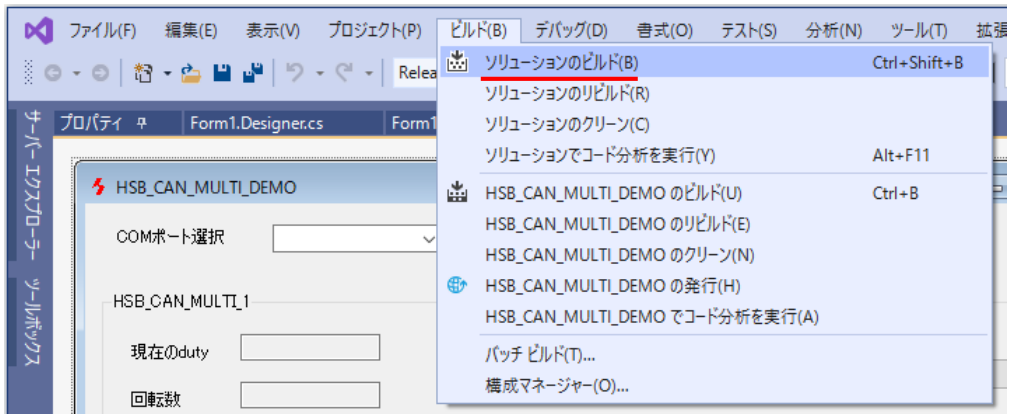
トンカチのアイコンを押す。

ーVisualStudio プロジェクトー



構成の「Debug」を「Release」に変更する

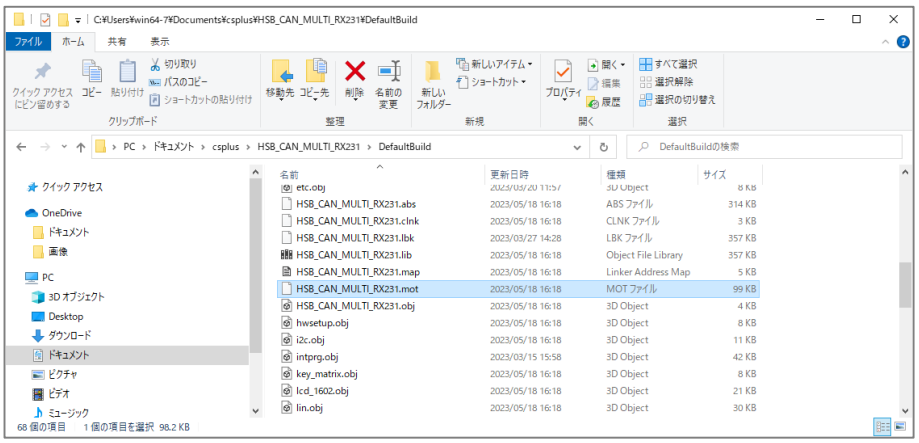




「ビルド」ー「ソリューションのビルド」を実行する

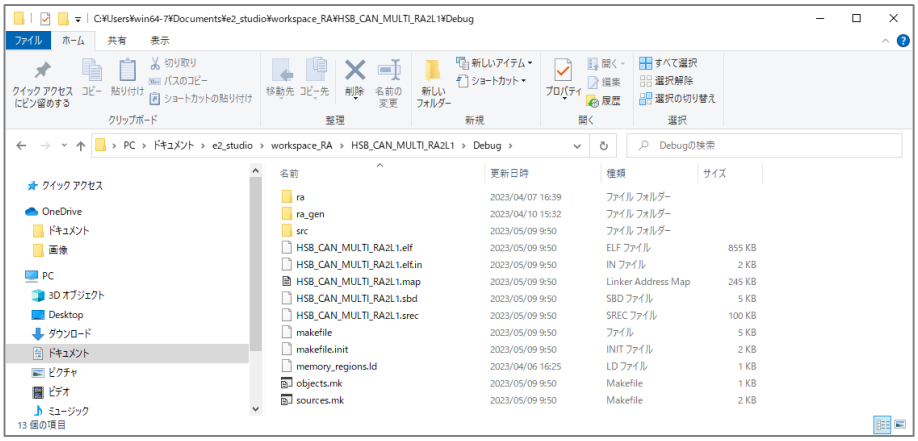
3.4. ビルド結果の生成物

ーCS+プロジェクトー



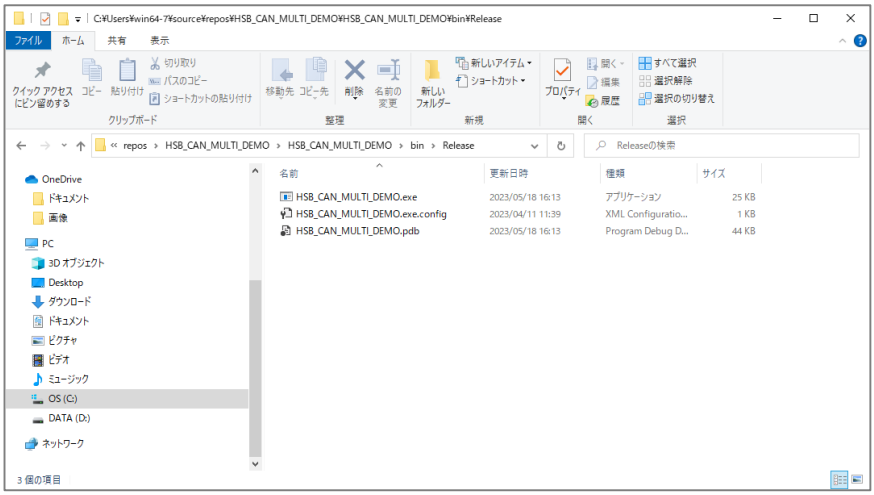
プロジェクトフォルダの下の
DefaultBuild¥
フォルダ内に、
プロジェクト名.mot
（HSB_CAN_MULTI 向け、RX231 では HSB_CAN_MULTI_RX231.mot）
が出力されます。このファイルを、2 章を参照して、マイコンボードに書き込みを行ってください。

－e2studio プロジェクト－



プロジェクトフォルダの下の
Debug¥
フォルダ内に、
プロジェクト名.srec
（HSB_CAN_MULTI 向け、RA2L1 では HSB_CAN_MULTI_RA2L1.srec）
が出力されます。このファイルを、2 章を参照して、マイコンボードに書き込みを行ってください。

－VisualStudio プロジェクト－



プロジェクトフォルダの
bin¥Release¥
フォルダ内に、
プロジェクト名.exe
（HSB_CAN_MULTI_DEMO プロジェクトでは、HSB_CAN_MULTI_DEMO.exe）
が出力されます。このファイルを、ダブルクリックで実行してください。

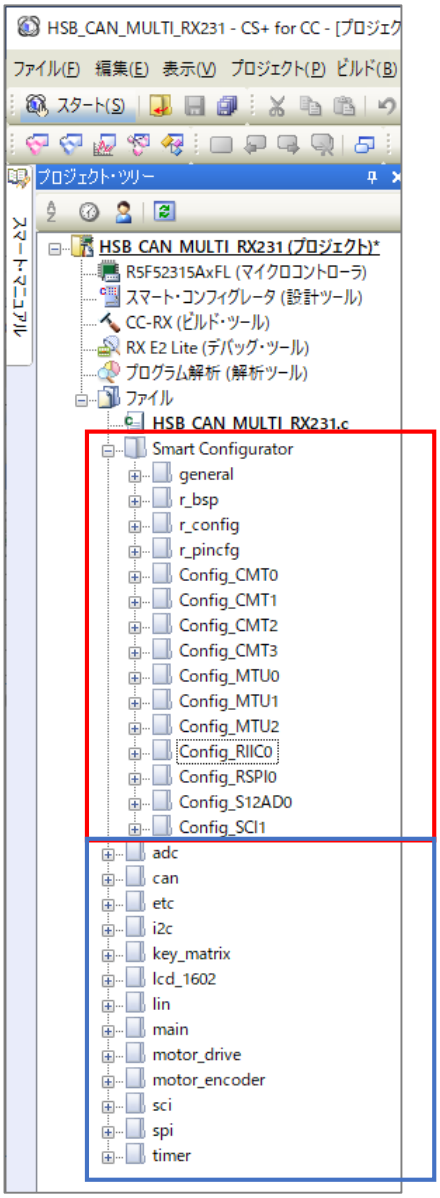
4. コード生成機能の使用に関して

マイコンボード向けのプロジェクトに関しては、ルネサスエレクトロニクス社が用意している、コード生成の機能を使って一部のプログラムコードを出力させています。

RX231 向けのプロジェクトでは、「RX スマート・コンフィグレータ」。
RL78/F15 向けのプロジェクトでは、「コード生成」。
RA2L1 向けのプロジェクトでは、「FSP」。
RL78/G11 向けのプロジェクトでは、「コード生成」。
を使用しています。これらのツールをどの部分で使用しているかを記載します。

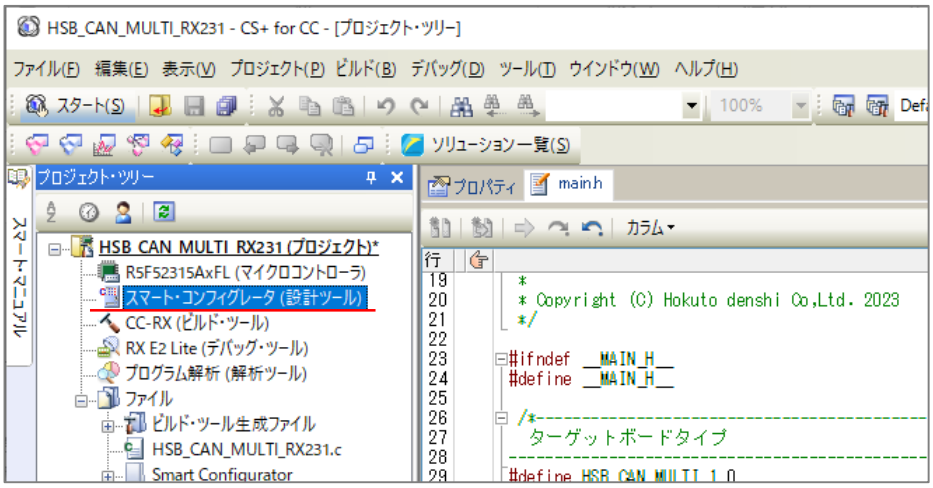
4.1. HSB_CAN_MULTI_RX231（スマート・コンフィグレータ）

プロジェクトツリーでは、



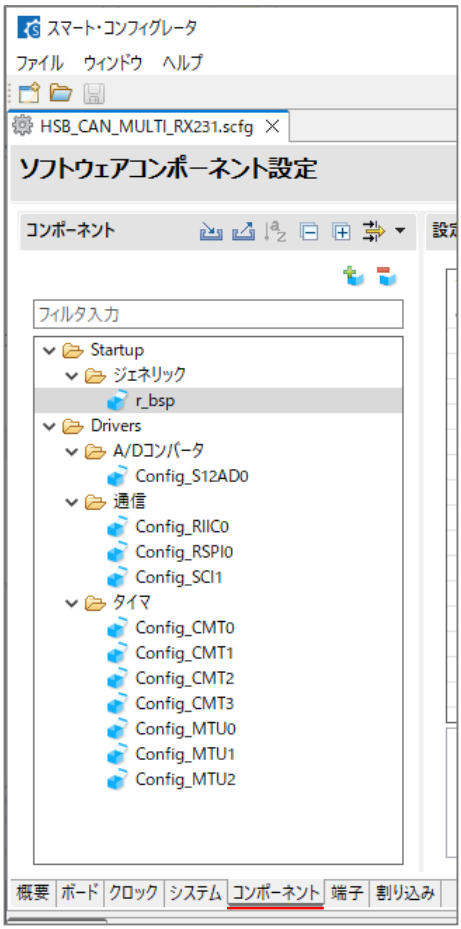
赤線で囲まれた部分が、スマート・コンフィグレータで生成されたプログラムコードです。（スマート・コンフィグレータでは、バイナリ化されたライブラリを使用する訳ではなく、ソースコードの形でプログラムコードが生成されます。）

青線で囲まれた部分が、ユーザ作成コードです。（ユーザ作成コード部分は、「取扱説明書　ソフトウェア編」を参照してください。）



スマート・コンフィグレータは、CS+には含まれませんので、別途インストールが必要です。（RX スマート・コンフィグレータをインストールしてください。）

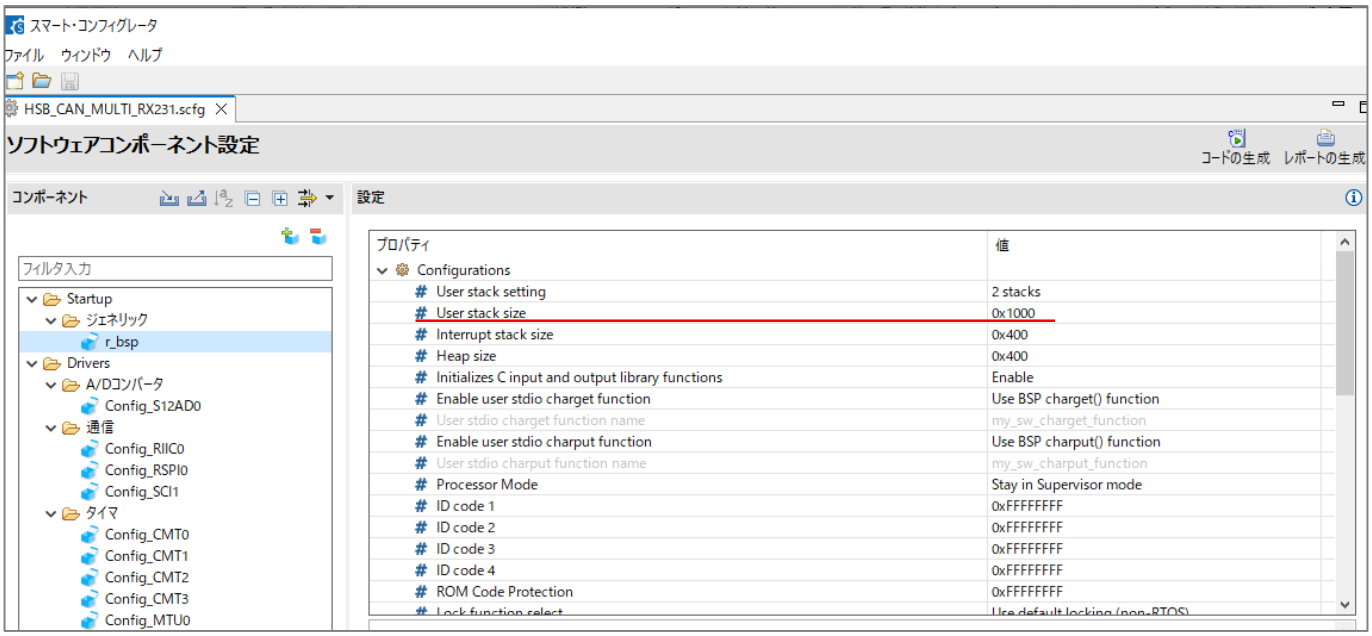
スマート・コンフィグレータ（設計ツール）の部分をダブルクリックすると、スマート・コンフィグレータが起動します。



「コンポーネント」タブを選択すると、現在どのような機能を使用しているかが判ります。

コンポーネント名	内容	使用ボード
r_bsp	クロック等の基本設定	全てのボード
Config_S12AD0	A/D 変換	HSB_CAN_MULTI_3
Config_RIIC0	I2C 通信	HSB_CAN_MULTI_3
Config_RSPI0	SPI 通信	HSB_CAN_MULTI_3
Config_SCI0	SCI(UART)通信	HSB_CAN_MULTI_4
Config_CMT0	タイマ(500ms), CAN 通信周期	全てのボード
Config_CMT1	タイマ(1ms), スイッチ読み取り	HSB_CAN_MULTI_1,HSB_CAN_MULTI_2
Config_CMT2	タイマ(10ms), モータ回転数	HSB_CAN_MULTI_1
Config_CMT3	タイマ(50ms), SCI コマンド処理等	全てのボード
Config_MTU0	タイマ, モータ駆動	HSB_CAN_MULTI_1
Config_MTU1	タイマ, モータエンコーダ	HSB_CAN_MULTI_1
Config_MTU2	タイマ, モータ駆動	HSB_CAN_MULTI_1

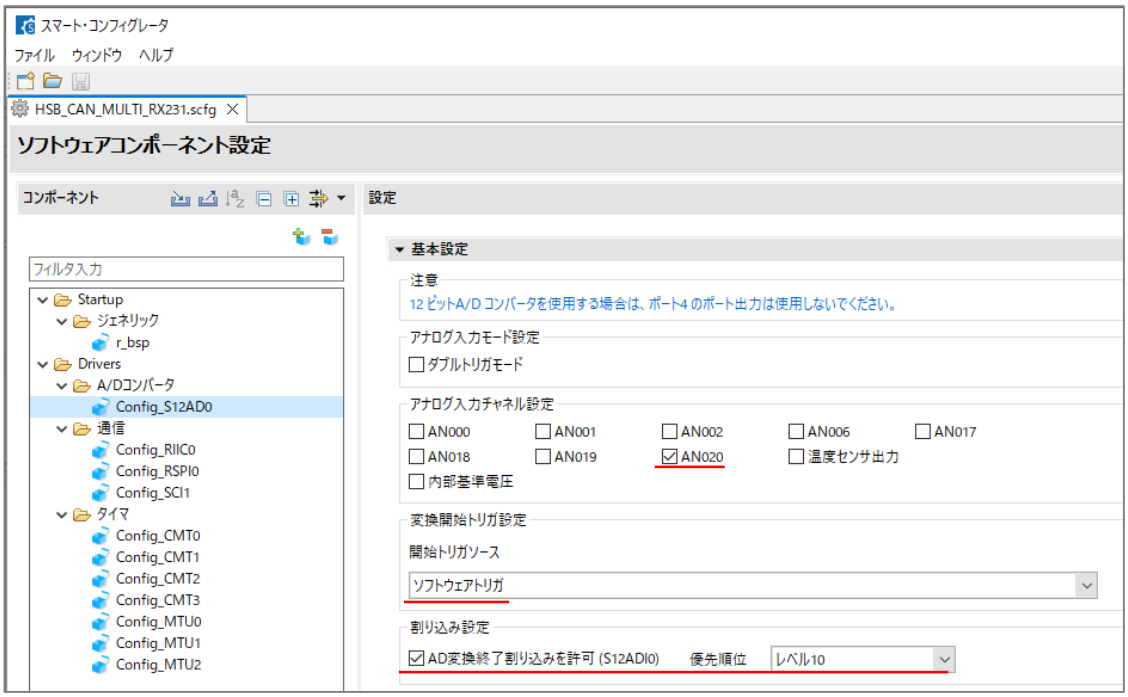
4.1.1. r_bsp



r_bps では、User stack size をデフォルト(0x400)から変更しています。SPI フラッシュのアクセス等でメモリを使用する部分があり、デフォルトから増やしています。(設定値 0x1000=4kB)

※ユーザスタックは、関数内で定義した変数(自動変数)を確保する際に消費されるメモリ領域です

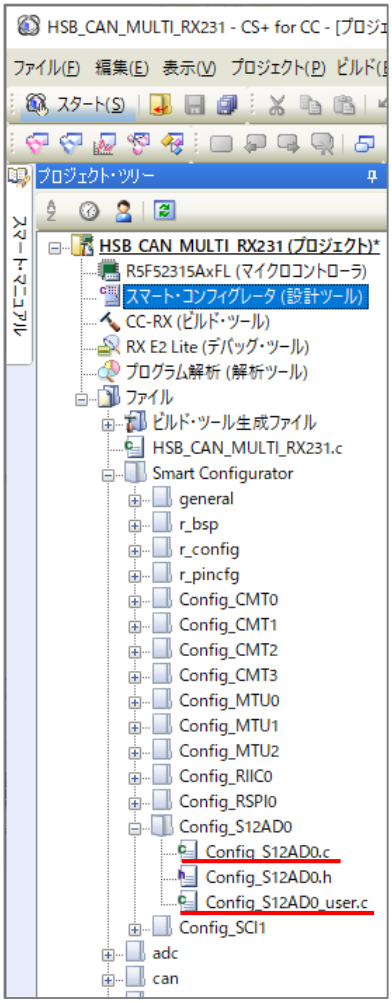
4.1.2. A/D コンバータ(Config_S12AD0)



A/D 変換で使用する端子（フォトダイオード接続端子）AN020
ソフトウェアトリガ（50ms タイマ内で、A/D 変換開始コードを実行）
A/D 変換割り込み許可

の設定を行っています。ユーザ記載のコード内には、A/D 変換の初期化等のコードは含まれていません。上記、GUI の設定のみで、A/D 変換の実行が行えています。

スマートコンフィグレータで設定した項目は、



ソースツリーでは、

Config_S12AD0.c　　API 関数が記載されている（基本的にはユーザが編集するものではない）

Config_S12AD0_user.c　割り込み関数が記載されている（ユーザが処理を追記するファイル）

に反映されます。

Config_S12AD0.c 内には、

void R_Config_S12AD0_Create(void)

void R_Config_S12AD0_Start(void)

void R_Config_S12AD0_Stop(void)

void R_Config_S12AD0_Get_ValueResult(ad_channel_t channel, uint16_t * const buffer)

という 4 つの API 関数が生成されており、

R_Config_S12AD0_Create() 初期化関数、初期処理で実行されているので、基本ユーザが実行する必要はない
R_Config_S12AD0_Start() A/D 変換を開始する関数
R_Config_S12AD0_Get_ValueResult() A/D 変換の結果を回収する関数

それぞれの関数は、上記の様な働きをします。

Config_S12AD0_user.c 内には、

void R_Config_S12AD0_Create_UserInit(void)
static void r_Config_S12AD0_interrupt(void)

が定義されており、

R_Config_S12AD0_Create_UserInit() 初期化時にユーザで追加したい処理を記載（中身は空）
r_Config_S12AD0_interrupt() A/D 変換終了のタイミングで実行される、割り込み関数

となっています。r_Config_S12AD0_interrupt()内には、コードを追加しており、

```
static void r_Config_S12AD0_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
    unsigned short val;

    R_Config_S12AD0_Get_ValueResult(ADCHANNEL20, &val);
    g_adc_val = val;
    g_adc_flag = ADC_DONE;
    /* End user code. Do not edit comment generated here */
}
```

上記の記載としています。

A/D 変換終了のタイミングで、
・変換結果をグローバル変数にコピー(g_adc_val)
・A/D 変換実行中フフラグを落とす
処理を行っています。

スマートコンフィグレータ生成コード内にユーザコードを追加する場合は、必ず

/* Start user code
/* End user code.

の間に記載する必要があります。
（それ以外の場所に記載すると、プログラムコードの自動生成の際に、消されてしまいます。）

－使用している API 関数－

R_Config_S12AD0_Start()

A/D 変換の開始

ユーザコードの adc¥adc.c 内の adc_kick()関数内で使用

R_Config_S12AD0_Get_ValueResult()

A/D 変換結果の回収

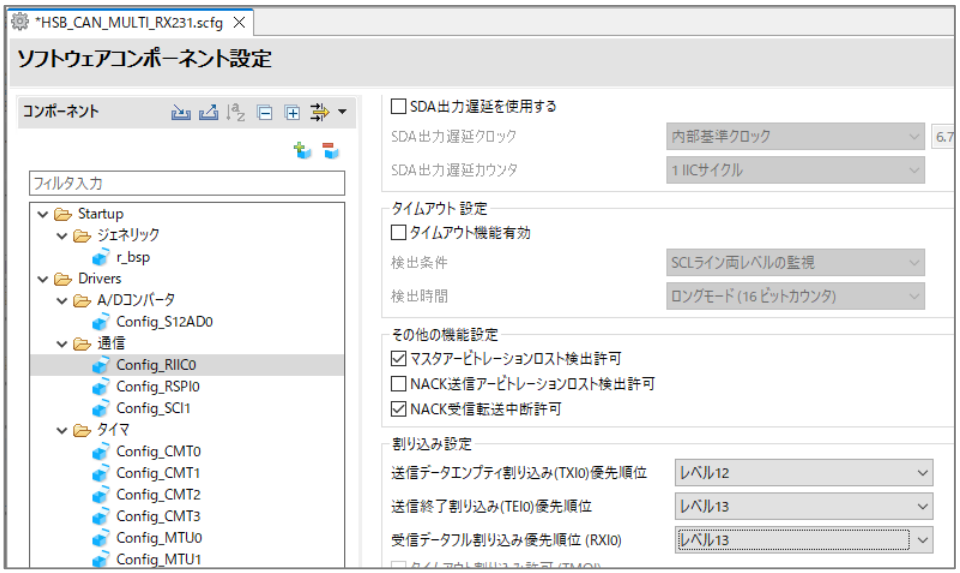
A/D 完了割り込み関数(r_Config_S12AD0_interrupt())内で使用

－ユーザ側で記載を追加している関数－

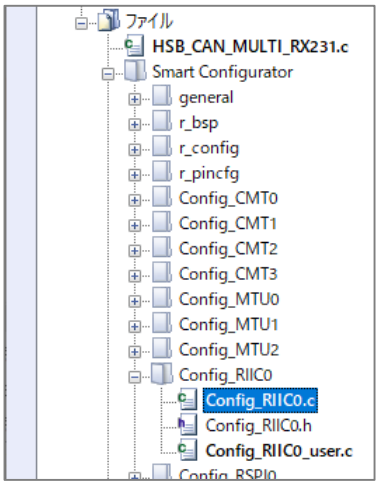
r_Config_S12AD0_interrupt()

A/D 変換結果の回収、A/D 変換中フラグを落とす

4.1.3. I2C(Config_RIIC0)



通信速度(100kbps)と割り込みレベル（12,13）の設定を行っています。



Config_RIIC0.c が API 関数、Config_RIIC0_user.c がユーザで追加するソースです。

－使用している API 関数－

R_Config_RIIC0_Start()

I2C 通信前に実行

R_Config_RIIC0_Master_Send()

データ送信 API 関数

I2C デバイス（温度センサ）のレジスタ書き換えに使用

R_Config_RIIC0_Master_Receive()

データ受信 API 関数

I2C デバイス（温度センサ）のレジスタ（温度値）読み出しに使用

－ユーザ側で記載を追加している関数－

r_Config_RIIC0_callback_transmitend()

送信完了割り込み時に呼ばれる関数

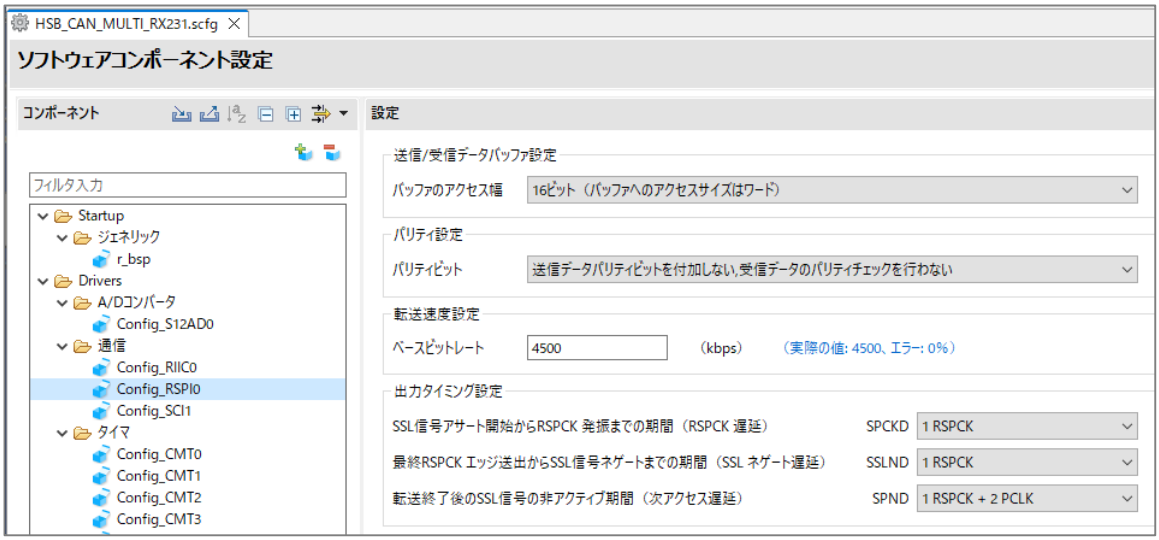
送信中フラグを落とす処理を記載

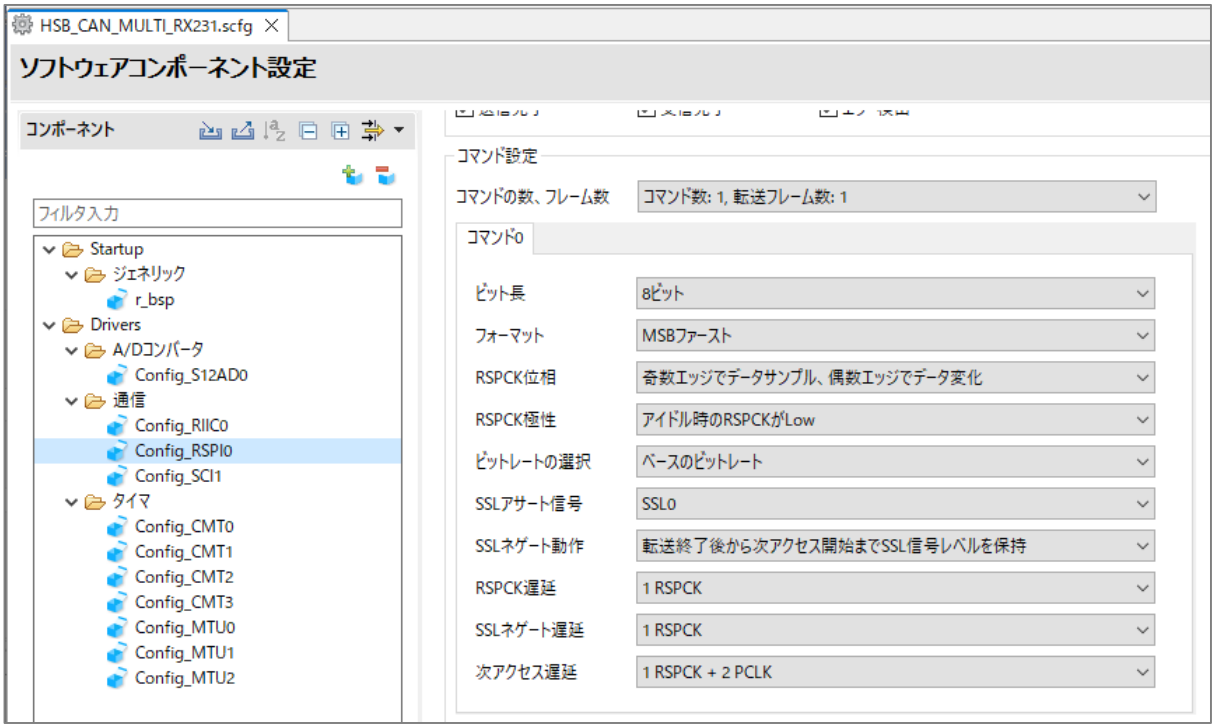
r_Config_RIIC0_callback_receiveend()

受信完了割り込み時に呼ばれる関数

受信中フラグを落とす処理を記載

4.1.4. SPI(Config_RSPI0)



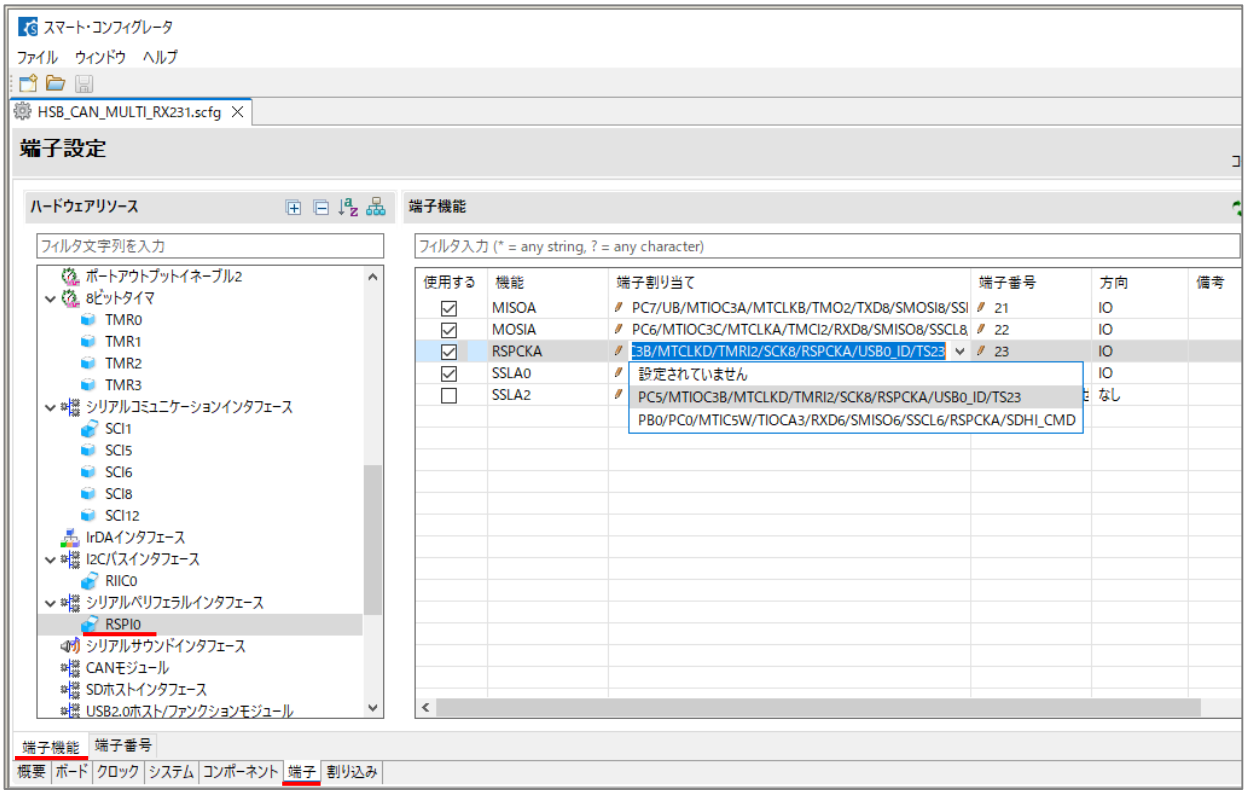


通信速度や、通信条件を設定。

SPI(RSPI0)は、使用する端子を選べるようになっています。例えば、クロック端子 RSPCKA は、PC5 か PB0 が割り当てられており、どちらの端子を使用するか設定が必要です。

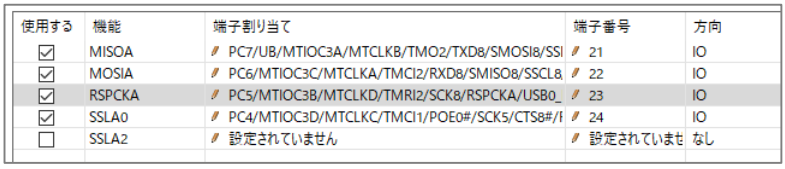
信号名	マイコン	備考
SCK	**PC5/RSPCKA**(23)	
MOSI	PC6/MOSI(22)	
MISO	PC7/MISO(21)	
*CS	PC4/SSLA0(24)	
*CS1	PA1(34)	接続先 J11-2
*CS2	PB5/PC3(25) [IO1]	
*CS3	PB0/PC0(29) [IO4]	
OUT1	PB1/PC1(27) [IO3]	汎用出力

信号表（取扱説明書）では、PC5 が SPI のクロックとして使用されているので、



「端子」タブ、「端子機能」タブ RSPI0

上記で、RSPCKA は、PC5 と PB0 のどちらを使用するか選択できるようになっています。



端子設定を取扱説明書の信号表と合わせると、上記の設定となります。

マイコンでは、マルチファンクションピンコントローラ（マイコン内蔵の機能をどの端子に割り当てるか）の設定が必要ですが、スマート・コンフィグレータでは、「端子」タブからこの設定を行う事となります。

生成されるコードでは、Config_RSPI0.c が API 関数、Config_RSPI0_user.c がユーザで追加するソースです。

－使用している API 関数－

R_Config_RSPI0_Start()

SPI 通信前に実行

R_Config_RSPI0_Send_Receive()

データ送受信 API 関数

SPI デバイス（フラッシュメモリ）のアクセスに使用

－ユーザ側で記載を追加している関数－

r_Config_RSPI0_callback_transmitend()

送信完了割り込み時に呼ばれる関数

送信中フラグを落とす処理を記載

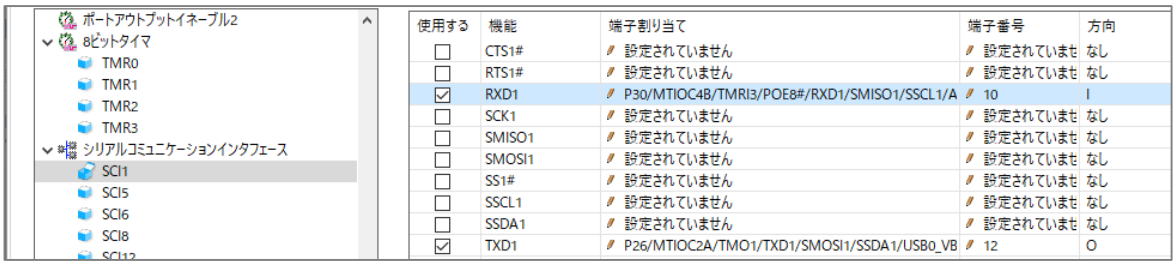
r_Config_RSPI0_callback_receiveend()

受信完了割り込み時に呼ばれる関数

受信中フラグを落とす処理を記載

4.1.5. SCI(Config_SCI)

通信速度(115,200bps)、8 ビット、パリティなし、1 ストップビットの通信条件と割り込みレベル(8)の設定。



使用端子は、TXD1(P26), RXD1(P30)の設定です。

生成されるコードでは、Config_SCI1.c が API 関数、Config_SCI1_user.c がユーザで追加するソースです。

－使用している API 関数－

R_Config_SCI1_Start()

UART(SCI)通信前に実行

R_Config_SCI1_Serial_Receive()

データ受信 API 関数

UART 経由でのコマンド受信に使用

R_Config_SCI1_Serial_Send()

データ送信 API 関数

PC に情報を送る用途で使用

－ユーザ側で記載を追加している関数－

r_Config_SCI1_callback_transmitend()

送信完了時に呼ばれる関数

送信データを再度 API 関数に渡す処理で使用

r_Config_SCI1_callback_receiveend()

受信完了時に呼ばれる関数

受信データをリングバッファにコピーする処理を実行

r_Config_SCI1_callback_receiveerror()

受信エラー時に呼ばれる関数

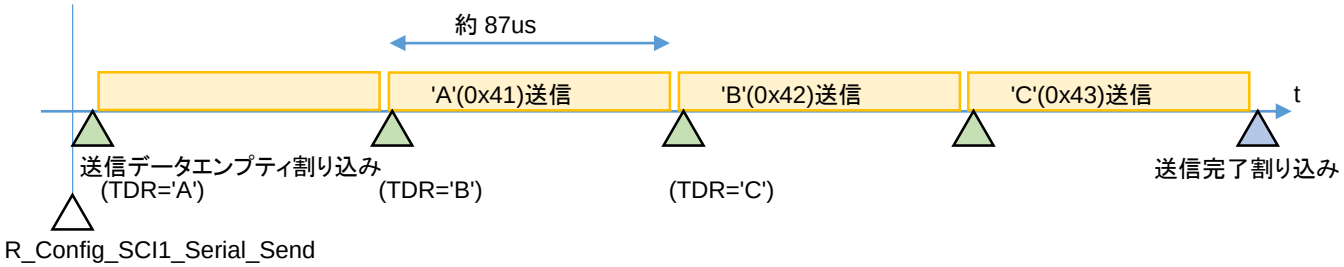
エラーフラグを立てる処理で使用

sci.c を使用する場合、
Config_SCI1_user.c 内に
これらのコードを追加してください

```
/***********************************************************************************************
***************************
* Function Name: r_Config_SCI1_callback_transmitend
* Description  : This function is a callback function when SCI1 finishes transmission
* Arguments    : None
* Return Value : None
***********************************************************************************************
*************************/
static void r_Config_SCI1_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI1_callback_transmitend. Do not edit comment generated
here */
    intr_sci_send_end();
    /* End user code. Do not edit comment generated here */
}

/***********************************************************************************************
***************************
* Function Name: r_Config_SCI1_callback_receiveend
* Description  : This function is a callback function when SCI1 finishes reception
* Arguments    : None
* Return Value : None
***********************************************************************************************
*************************/

static void r_Config_SCI1_callback_receiveend(void)
{
    /* Start user code for r_Config_SCI1_callback_receiveend. Do not edit comment generated
here */
    intr_sci_receive_end();
    /* End user code. Do not edit comment generated here */
}

/***********************************************************************************************
***************************
* Function Name: r_Config_SCI1_callback_receiveerror
* Description  : This function is a callback function when SCI1 reception encounters error
* Arguments    : None
* Return Value : None
***********************************************************************************************
*************************/

static void r_Config_SCI1_callback_receiveerror(void)
{
    /* Start user code for r_Config_SCI1_callback_receiveerror. Do not edit comment generated
here */
    intr_sci_receive_error();
    /* End user code. Do not edit comment generated here */
}
```

intr_sci_send_end(), intr_sci_receive_end(), intr_sci_receive_error()は、sci.c 内に記載しているユーザコードです。これらの関数を、スマート・コンフィグレータが生成した割り込み関数（正確には、割り込み関数から呼ばれる通常の関数）内に記載しています。

```
unsigned char str[3]={"ABC"};

R_Config_SCI1_Start();
R_Config_SCI1_Serial_Send(str, 3);
```



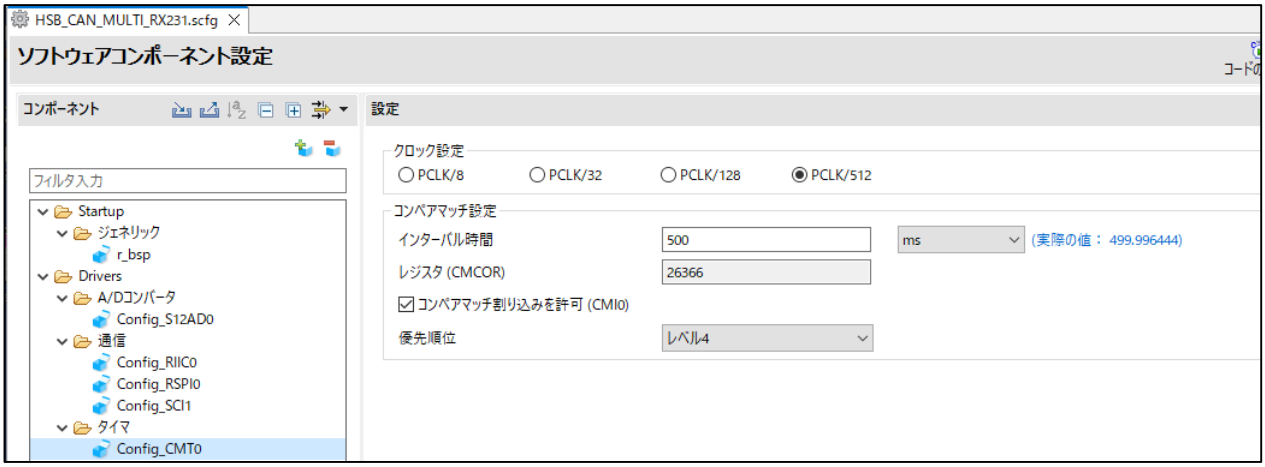
SCI の送信割り込みには 2 種類あり、最後のデータの送信が終わった場合は、送信完了割り込み。最後のデータ以外は、送信レジスタに空きができた時点（実際にデータの送信を始めたタイミング）で送信データエンプティ（送信バッファが空きました、次のデータ送ってください）割り込みが掛かります。

r_Config_SCI1_callback_transmitend()は、送信完了割り込みで API 関数が処理を行った後で呼ばれる関数になります。

ユーザ側のコードでは、r_Config_SCI1_callback_transmitend()内で、intr_sci_send_end()を実行し、その時点でバッファに溜まっているデータを R_Config_SCI1_Serial_Send()に渡す処理を行っています。

データの受信に関しては、
R_Config_SCI1_Serial_Receive(&g_sci_recv_data, 1);
としておき、1 バイト受信する毎に、intr_sci_receive_end()が呼ばれる様にしています。intr_sci_receive_end()内で呼ばれる intr_sci_receive_end()の中で、受信データをリングバッファに格納した後、再度
R_Config_SCI1_Serial_Receive(&g_sci_recv_data, 1);
を実行し、次に 1 バイト受信したタイミングで、intr_sci_receive_end()が呼ばれる様にしています。
PC との通信において、決まったパケットサイズでやり取りをするわけではないため、1 バイト単位での処理としています。
（R_Config_SCI1_Serial_Receive(xxx, 3) とすると、受信データが 3 バイト溜まらないと、ユーザプログラム側で処理ができない事となります。1 バイト単位の処理では、処理効率という点では、効率が悪いのですが、許容しています。なお、送信側の R_Config_SCI1_Serial_Send()は、1 度に複数バイトを API 関数に渡しています。）

4.1.6. タイマ(Config_CMTn)



CMT は定期的に実行されるタイマの処理です。CMT0~CMT3 の 4 つのタイマを定義しています。

	周期	割り込みレベル	用途
CMT0	500ms	4	CAN 定期送信
CMT1	1ms	4	スイッチ読み取り
CMT2	10ms	4	エンコーダセンサカウント計測
CMT3	50ms	4	センサ, LIN, SCI 処理等

生成されるコードでは、Config_CMT*n*.c が API 関数、Config_CMT*n*_user.c がユーザで追加するソースです。(n=0~3)

－使用している API 関数－

R_Config_CMT*n*_Start()

タイマスタート API 関数

タイマ動作開始時に実行

R_Config_CMT*n*_Stop()

タイマ停止 API 関数

タイマ停止時に実行

－ユーザ側で記載を追加している関数－

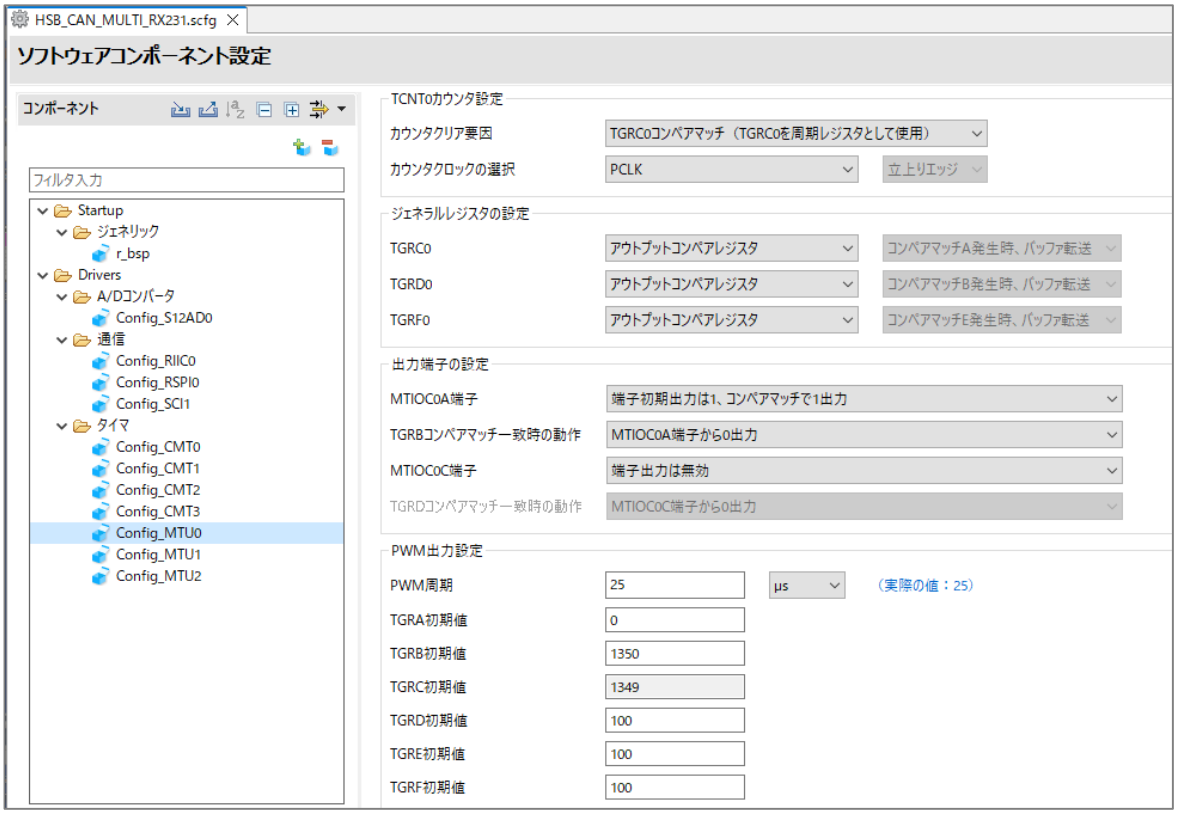
r_Config_CMT*n*_cmi*n*_interrupt()

周期的に呼ばれる関数

```
static void r_Config_CMT0_cmi0_interrupt(void)
{
    /* Start user code for r_Config_CMT0_cmi0_interrupt. Do not edit comment generated here */
    timer_cmt0();
    /* End user code. Do not edit comment generated here */
}
```

timer_cmt0()は、timer.c 内に記載している関数です。500ms 毎に timer_cmt0()が実行されます。

4.1.1. PWM タイマ(Config_MTU0, Config_MTU2)



HSB_CAN_MULTI_1 のモータ駆動時に、MTU タイマを使用して PWM 信号を生成しています。

PWM の周期は、25us(40kHz)としています。

生成されるコードでは、Config_MTU*n*.c が API 関数、Config_MTU*n*_user.c がユーザで追加するソースです。(n=0,2)

－使用している API 関数－

R_Config_MTU*n*_Start()

タイマスタート API 関数

タイマ動作開始時に実行

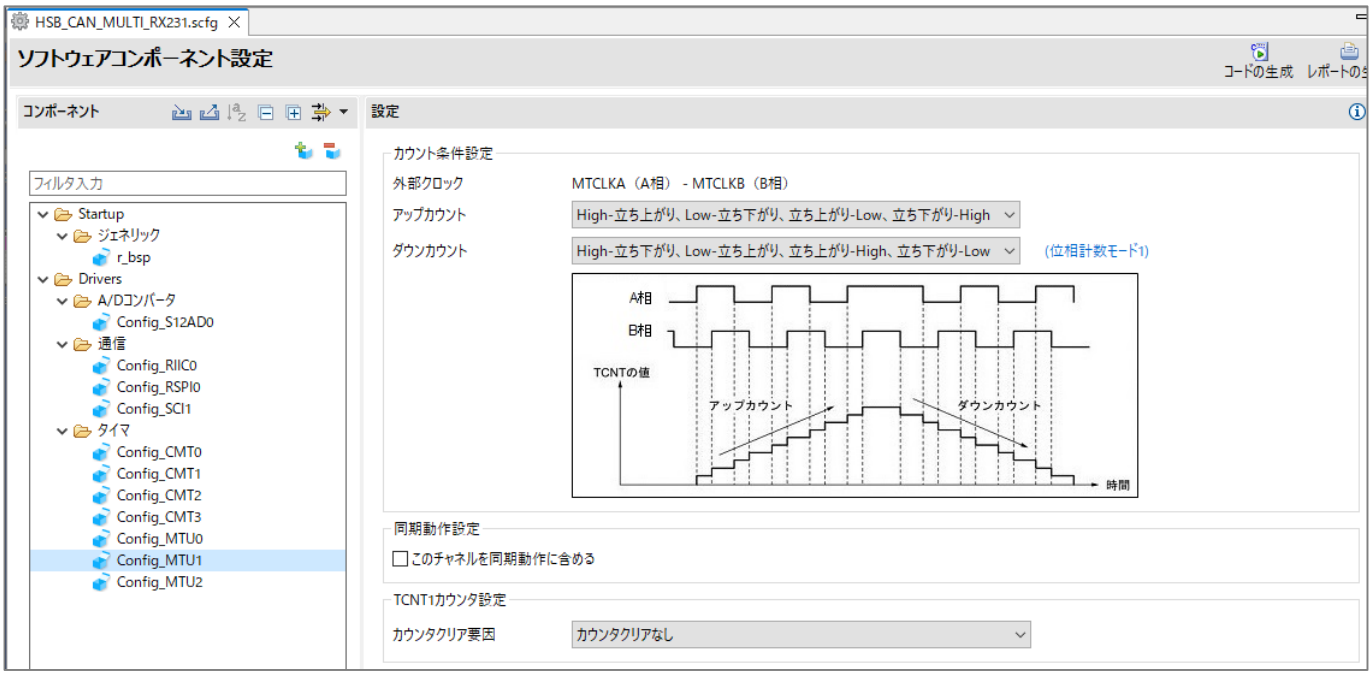
R_Config_MTU*n*_Stop()

タイマ停止 API 関数

タイマ停止時に実行

Config_MTU*n*_user.c 側にはコードを追加していません。

4.1.2. 位相計数タイマ(Config_MTU1)



HSB_CAN_MULTI_1 のモータエンコーダセンサの計測に、MTU1 タイマを使用しています。

PWM の周期は、25us(40kHz)としています。

生成されるコードでは、Config_MTU1.c が API 関数、Config_MTU1_user.c がユーザで追加するソースです。

ー使用している API 関数ー

R_Config_MTU1_Start()

タイマスタート API 関数

タイマ動作開始時に実行

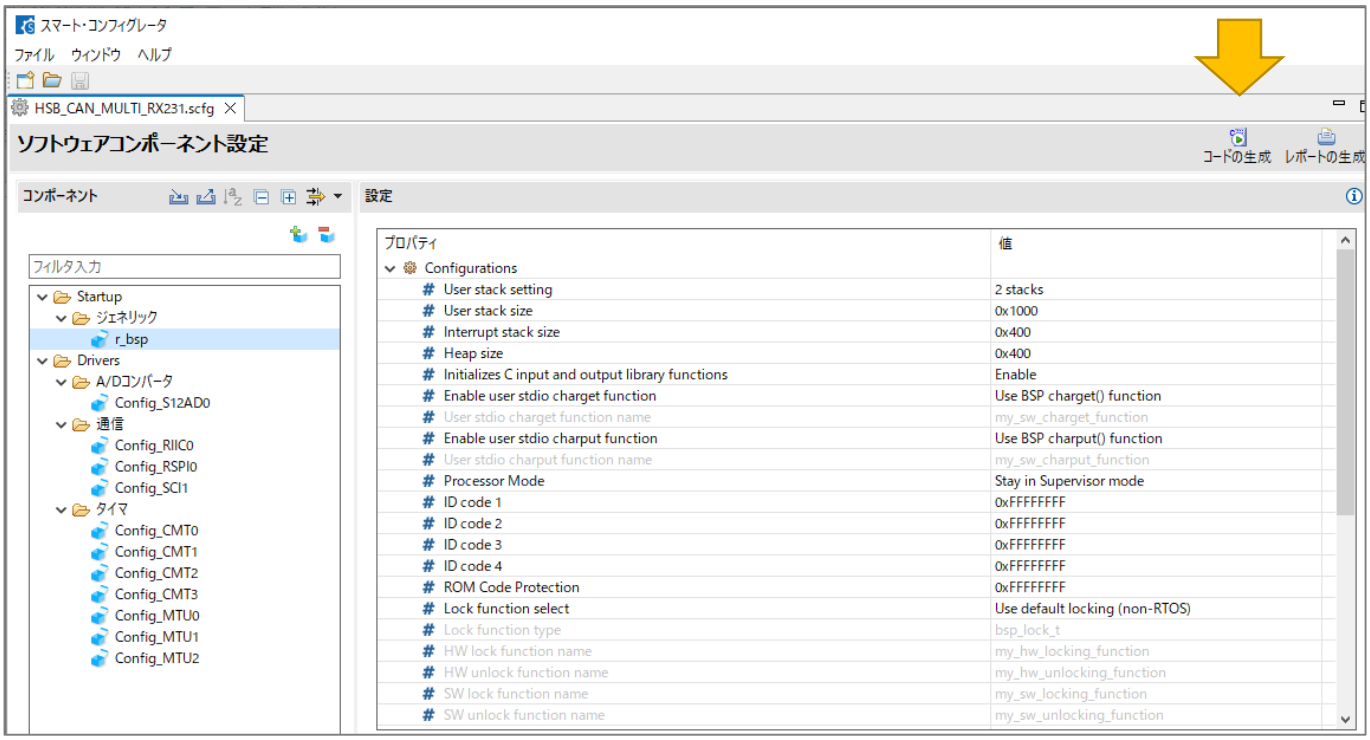
R_Config_MTU1_Stop()

タイマ停止 API 関数

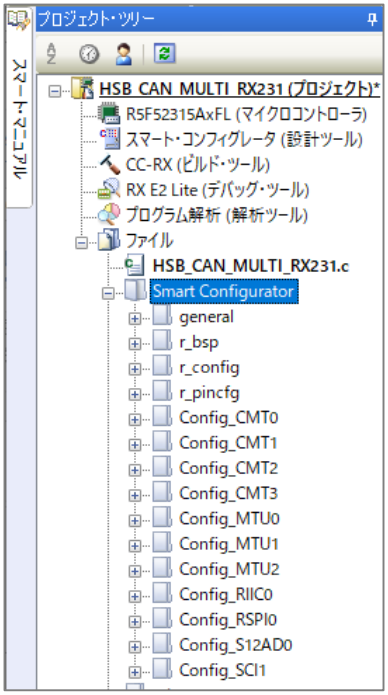
タイマ停止時に実行

Config_MTU1_user.c 側にはコードを追加していません。

4.1.3. プログラムコードの出力

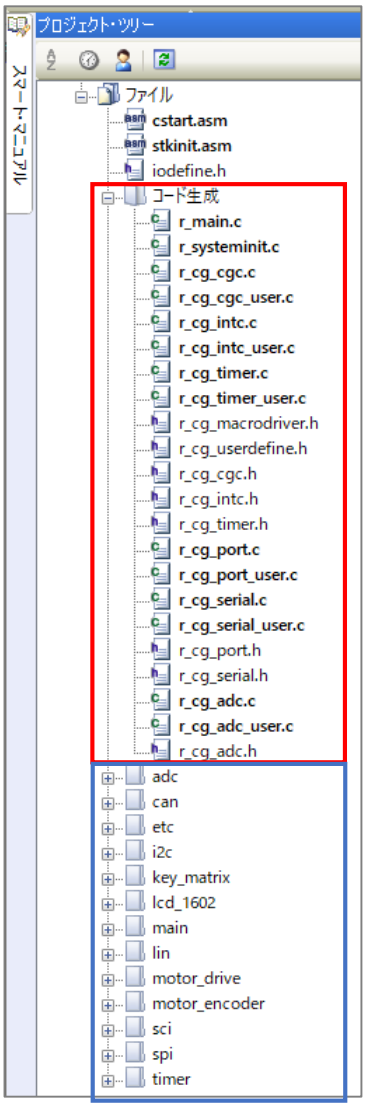


GUI 上で一通りの設定を行った後、「コード生成」のボタンを押すと、



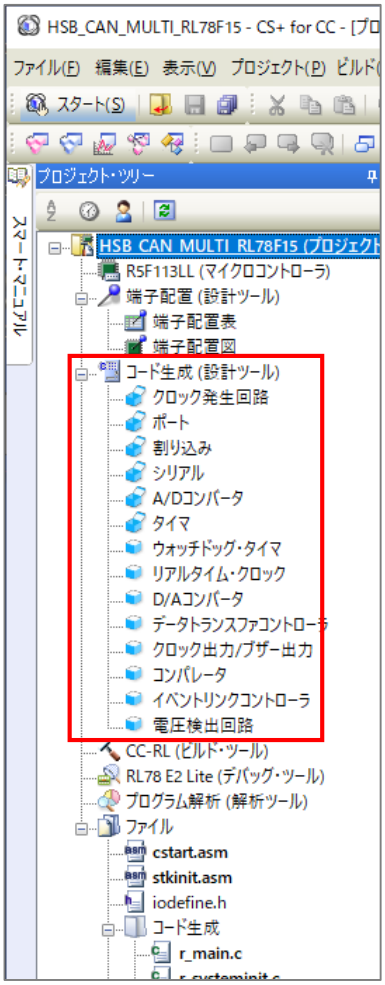
プログラムコードの出力が行われ、SmartConfigurator 以下のソースコードが生成・更新されます。
（その後、3 章の手順でプログラムのビルドを行ってください。）

4.2. HSB_CAN_MULTI_RL78/F15（コード生成）



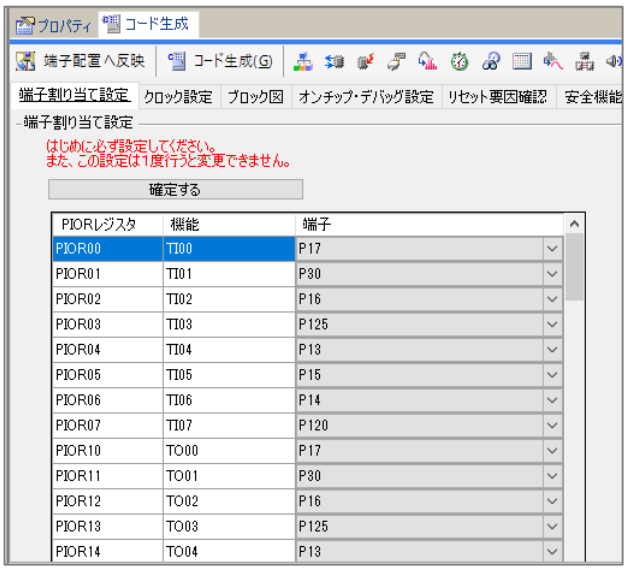
赤線で囲まれた部分が、コード生成で生成されたプログラムコードです。（コード生成では、バイナリ化されたライブラリを使用する訳ではなく、ソースコードの形でプログラムコードが生成されます。）

青線で囲まれた部分が、ユーザ作成コードです。（ユーザ作成コード部分は、「取扱説明書　ソフトウェア編」を参照してください。）

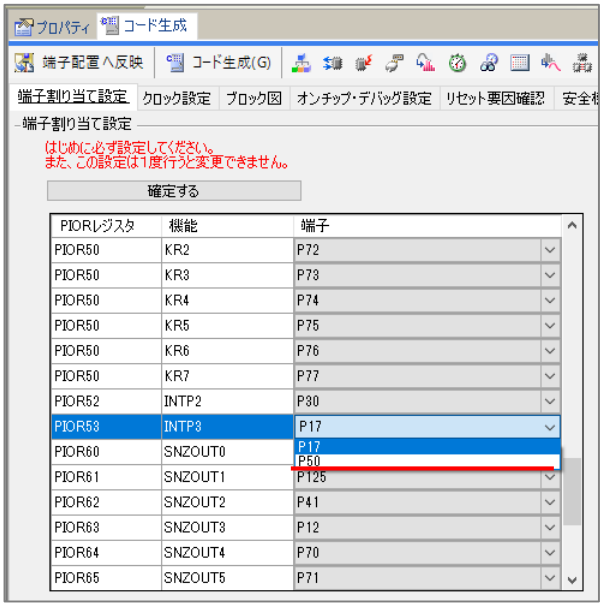


RL78 のコード生成は、RX と異なり CS+の中で完結しています。
（RX のスマート・コンフィグレータは、CS+とは別なツール上で実行されます。）

4.2.1. 端子割り当て設定



RL78 では、最初に端子割り当ての設定があります。「この設定は一度行うと変更できません。」という記載があり、この設定を間違えると、プロジェクトの作り直しとなります。



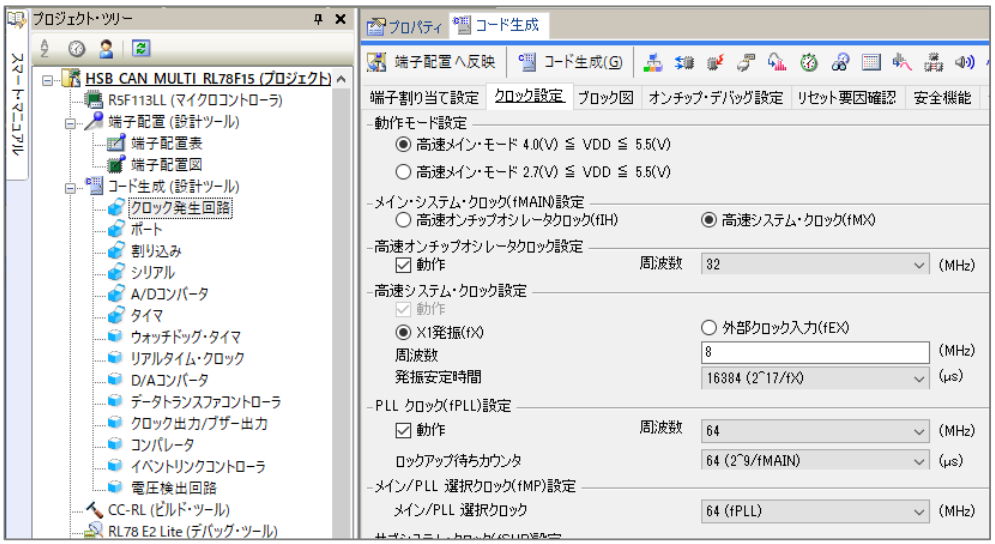
この設定で、1 か所デフォルトから変更する必要があります。INTP3 が初期値 P17 ですが、これを P50 側に変更してください。

エンコーダ回路	マイコン	備考
ENC1	P50/(INTP3)(38) [IO5]	INTP3 を使用する場合 PIOR の設定要
ENC2	P51/INTP11(39) [IO6]	

エンコーダ回路の ENC1 側は、マイコンの P50 に接続されています。INTP3 は、P17（デフォルト）で、P50 がサブの様な設定となります。

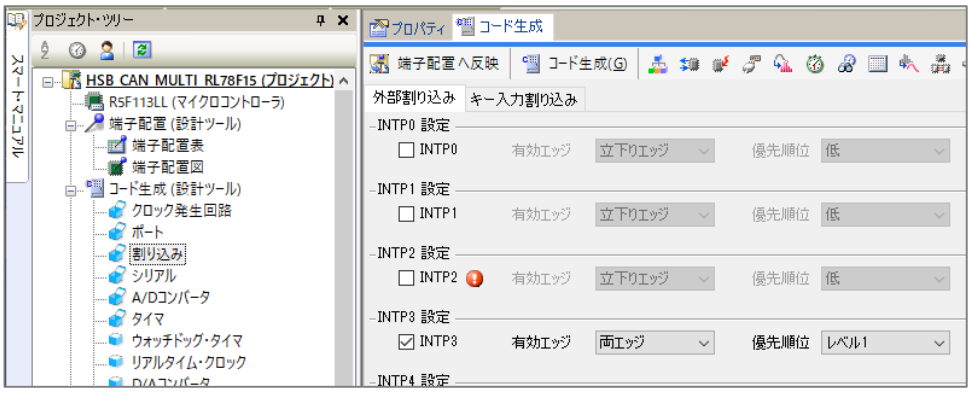
PIOR53 INTP3 を P50 側選択後、「確定する」のボタンを押してください。

4.2.2. クロック発生回路



クロック発生回路は、クロックの設定を行っているところになります。ボード(CPU_CARD_RL78F15-64)には、8MHz の水晶振動子が搭載されていますので、X1 発振(8)、PLL を動作させて 64MHz, CPU クロック 32MHz を生成します。

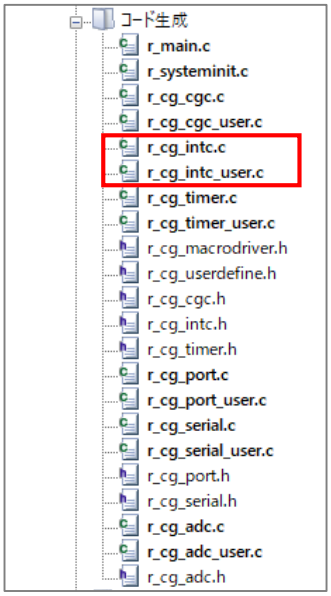
4.2.1. 割り込み



割り込みの設定では、どの割り込みを使用するか、割り込みのエッジ、割り込み優先度の設定を行います。この「割り込み」の設定は、割り込みにも色々ありますが「端子」割り込みです。端子レベルが変化した際に割り込みを掛ける機能です。

	有効エッジ	優先順位	用途
INTP3	両エッジ	レベル 1	モータエンコーダセンサ
INTP11	両エッジ	レベル 1	モータエンコーダセンサ

RL78/F15 では、位相計数タイマが無いので、HSB_CAN_MULTI_1 のエンコーダセンサの読み取りに割り込みを使用しています。



割り込みの設定が反映されるファイルは、

r_cg_intc.c　　API 関数が記載されている（基本的にはユーザが編集するものではない）

r_cg_intc_user.c　割り込み関数が記載されている（ユーザが処理を追記するファイル）

になります。

r_cg_intc.c 内には、

void R_INTC_Create(void)

void R_INTC3_Start(void)

void R_INTC11_Start(void)

void R_INTC3_Stop(void)

void R_INTC11_Stop(void)

という 5 つの API 関数が生成されており、

R_INTC_Create()　初期化関数、初期処理で実行されているので、基本ユーザが実行する必要はない

R_INTC3_Start(), R_INTC11_Start()　端子割り込みを開始する関数

R_INTC3_Stop(), R_INTC11_Stop()　端子割り込みを停止する関数

それぞれの関数は、上記の様な働きをします。

r_cg_intc_user.c 内には、

static void __near r_intc3_interrupt(void)
static void __near r_intc11_interrupt(void)

が定義されており、

r_intc3_interrupt() INTP3(P50)の端子レベルが変化した際に呼び出される割り込み関数
r_intc11_interrupt() INTP11(P51)の端子レベルが変化した際に呼び出される割り込み関数

となっています。r_intc3_interrupt()内には、コードを追加しており、

```
static void __near r_intc3_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    if (IO5_IN == 0)
    {
        //P50(INTP3) fall

        if (IO6_IN == 0)
        {
            //P51=L
            g_motor_encoder_value++;
        }
        else
        {
            //P51=H
            g_motor_encoder_value--;
        }
    }
    else
    {
        //P50(INTP3) rise

        if (IO6_IN == 0)
        {
            //P51=L
            g_motor_encoder_value--;
        }
        else
        {
            //P51=H
            g_motor_encoder_value++;
        }
    }

    /* End user code. Do not edit comment generated here */
}
```

上記の記載としています。

P50 端子の変化時、P51 が L なのか H なのかで、カウンタ変数(g_motor_encoder_value)を「カウントアップ」または「カウントダウン」する事としています。
（INTP11, r_intc11_interrupt()側は、P51 端子の変化時、P50 の L/H でカウンタを増減させる処理）

コード生成で出力されたコード内にユーザコードを追加する場合は、必ず

/* Start user code
/* End user code.

の間に記載する必要があります。
（それ以外の場所に記載すると、プログラムコードの自動生成の際に、消されてしまいます。）

－使用している API 関数－
R_INTC*n*_Start_Start()　(n=3, 11)
端子割り込み開始

R_INTC*n*_Start_Stop()　(n=3, 11)
端子割り込み停止

－ユーザ側で記載を追加している関数－
r_intc*n*_interrupt()　(n=3,11)
2 端子のエッジ、L/H の関係によりカウンタを増減

4.2.2. シリアル

・UART0

シリアルは、UART, SPI, I2C の設定が含まれます。

UART は、UART0 を 115,200bps, 8 ビット，パリティなし、ストップビット 1 で設定しています。

※RL78 の UART は、「送信」と「受信」で速度やビット数、パリティ設定などを別々に設定できる点が注意が必要です
上記の設定は「送信」の設定

受信設定は、別なタブになっています。

一般的には、「送信」と「受信」で別な速度に設定したり、パリティ有無を変える事はありません。（通信相手が、RL78 の場合は、送信と受信で別な値としても通信可能ですが、通常の通信相手は送信と受信で別な値を設定できないのが普通です。RL78 では、送信と受信で別なモジュールを使うため設定が独立していて、別な値を設定可能であるという点が、注意すべき点です。送信は通るが、受信ができない等の動作の場合、送信と受信で設定値が同じかどうかを確認してください。）

r_cg_serial.c が API 関数、r_cg_serial_user.c がユーザで追加するソースです。

－使用している API 関数－

R_UART0_Start()

UART 通信前に実行

R_UART0_Receive()

データ受信 API 関数

UART 経由でのコマンド受信に使用

R_UART0_Send()

データ送信 API 関数

PC に情報を送る用途で使用

－ユーザ側で記載を追加している関数－

r_uart0_callback_receiveend()

受信割り込み時に呼ばれる関数

受信データをリングバッファにコピーする処理を実行

r_uart0_callback_sendend()

送信完了割り込み時に呼ばれる関数

送信データを再度 API 関数に渡す処理で使用

r_uart0_callback_error()

受信エラー割り込みで呼ばれる関数

```
/***********************************************************************************************
***************************
* Function Name: r_uart0_callback_receiveend
* Description  : This function is a callback function when UART0 finishes reception.
* Arguments    : None
* Return Value : None
***********************************************************************************************
*************************/
static void r_uart0_callback_receiveend(void)
{
    /* Start user code. Do not edit comment generated here */
    intr_sci_receive_end();
    /* End user code. Do not edit comment generated here */
}
/***********************************************************************************************
***************************
* Function Name: r_uart0_callback_sendend
* Description  : This function is a callback function when UART0 finishes transmission.
* Arguments    : None
* Return Value : None
***********************************************************************************************
*************************/
static void r_uart0_callback_sendend(void)
{
    /* Start user code. Do not edit comment generated here */
    intr_sci_send_end();
    /* End user code. Do not edit comment generated here */
}

/***********************************************************************************************
***************************
* Function Name: r_uart0_callback_error
* Description  : This function is a callback function when UART0 reception error occurs.
* Arguments    : err_type -
*                    error type value
* Return Value : None
***********************************************************************************************
*************************/
static void r_uart0_callback_error(uint8_t err_type)
{
    /* Start user code. Do not edit comment generated here */
    intr_sci_receive_error();
    /* End user code. Do not edit comment generated here */
}
```

sci.c を使用する場合、
r_cg_serial_user.c 内に
これらのコードを追加してください

・CSI11(SPI)

CSI は、SPI の設定です。通信速度(4Mbps)や、信号条件、割り込みの設定等を行っています。

r_cg_serial.c が API 関数、r_cg_serial_user.c がユーザで追加するソースです。(UART0 と同じファイルです)

ー使用している API 関数ー

R_CSI11_Start()

SPI 通信前に実行

R_CSI11_Send_Receive()

送受信 API

SPI デバイス(フラッシュメモリ)のアクセスに使用

ーユーザ側で記載を追加している関数ー

r_csi11_callback_receiveend()

受信割り込み時に呼ばれる関数

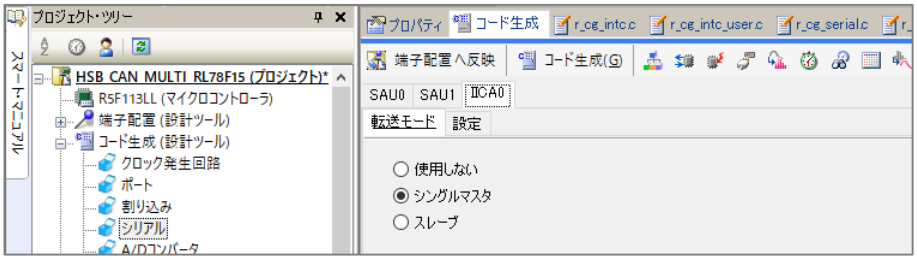
受信中フラグを落とす処理を記載

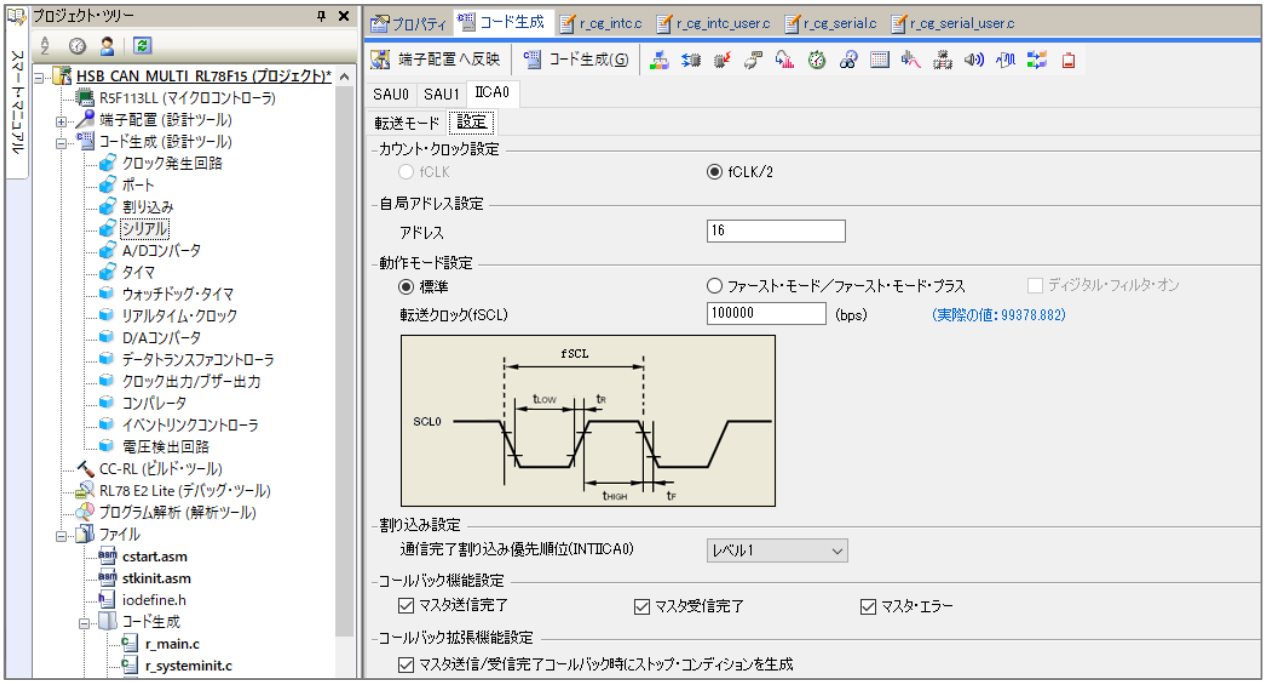
r_csi11_callback_sendend()

送信割り込み時に呼ばれる関数

送信中フラグを落とす処理を記載

・IICA0(I2C)





IICA は、I2C の設定です。通信速度(100kbps)や、信号条件、割り込みの設定等を行っています。

r_cg_serial.c が API 関数、r_cg_serial_user.c がユーザで追加するソースです。(UART0 と同じファイルです)

ー使用している API 関数ー

R_IICA0_Master_Send()

送信 API

I2C デバイス(温度センサ)のアクセスに使用

R_IICA0_Master_Receive()

受信 API

I2C デバイス(温度センサ)のアクセスに使用

－ユーザ側で記載を追加している関数－

r_iica0_callback_master_receiveend()

受信割り込み時に呼ばれる関数

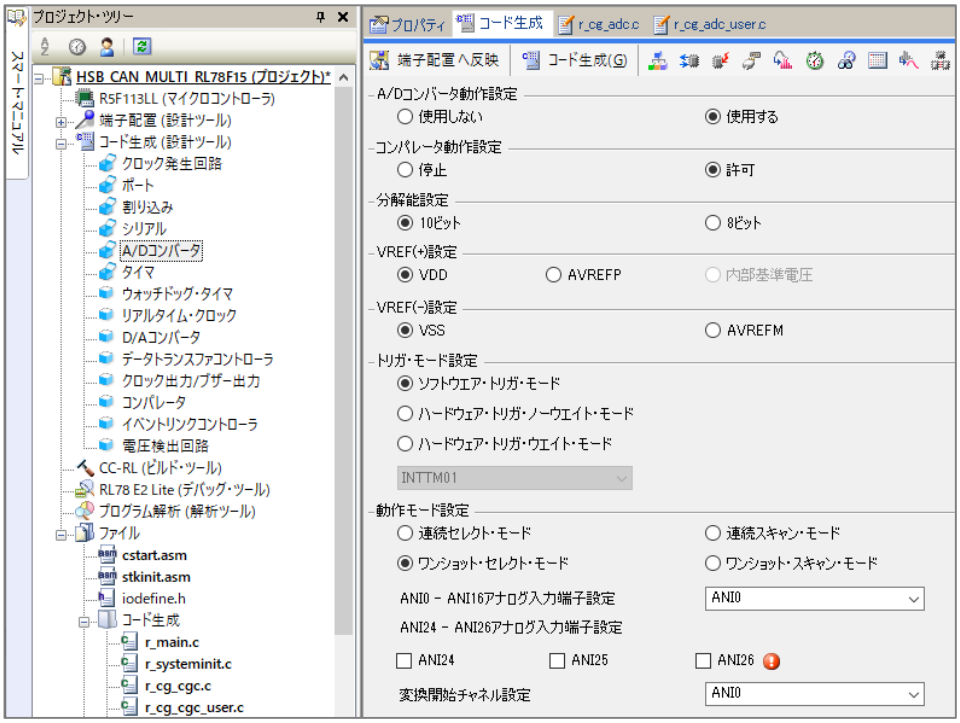
受信中フラグを落とす処理を記載

r_iica0_callback_master_sendend()

送信割り込み時に呼ばれる関数

送信中フラグを落とす処理を記載

4.2.3. A/D コンバータ



ANI0(P33)をアナログ入力に設定（他、割り込みの設定）。

r_cg_adc.c が API 関数、r_cg_adc_user.c がユーザで追加するソースです。

－使用している API 関数－

R_ADC_Start()

A/D 変換を開始する API 関数

タイマ内で定期的に実行

R_ADC_Get_Result()

A/D 変換結果を取得する API 関数

A/D 変換終了割り込み内で使用

－ユーザ側で記載を追加している関数－

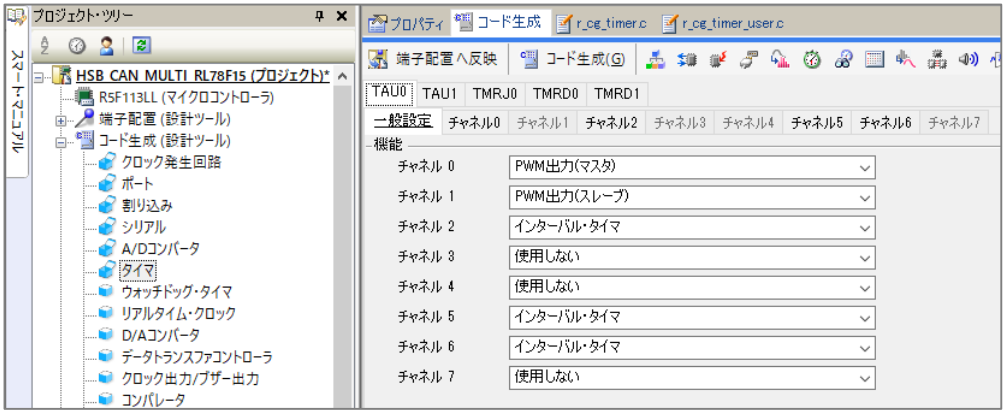
r_adc_interrupt()

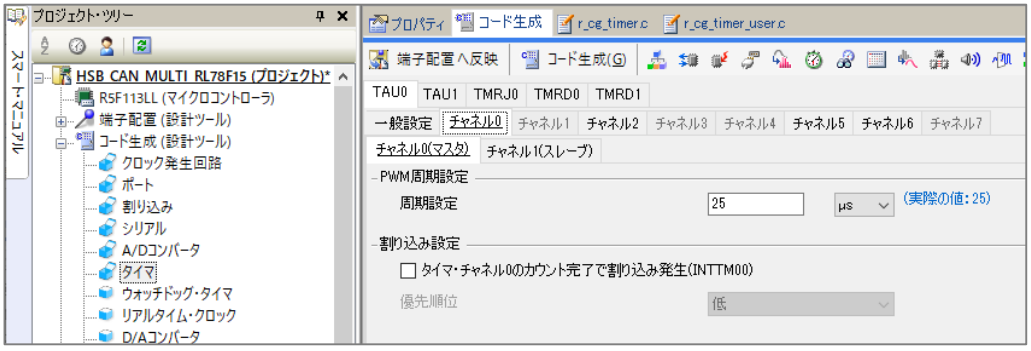
A/D 変換終了時に呼ばれる関数

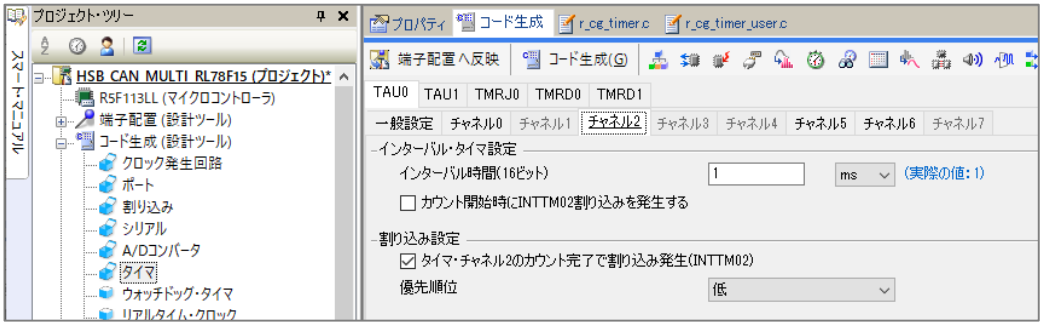
A/D 変換中フラグを落とす処理を記載

4.2.1. タイマ

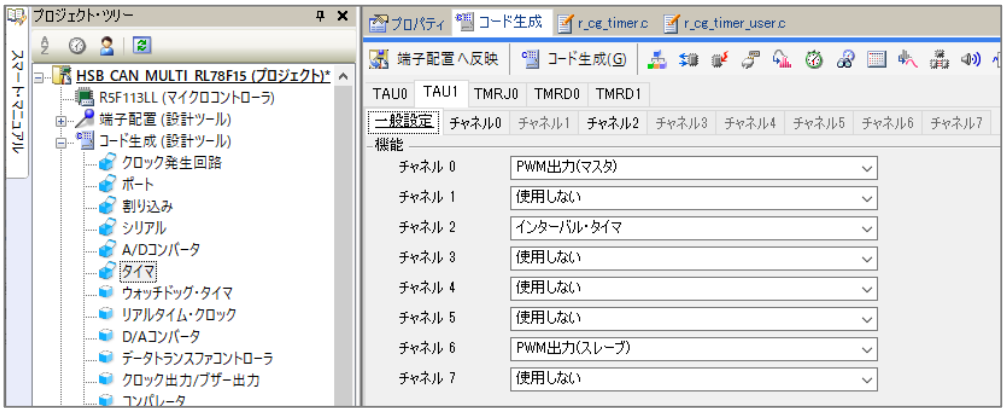
・TAU0







・TAU1



TAU	ch	周期	用途
TAU0	ch0/ch1	25us	HSB_CAN_MULTI_1 モータ駆動 PWM
	ch2	1ms	HSB_CAN_MULTI_1, HSB_CAN_MULTI_2 スイッチ読み取り[割り込み]
	ch5	50ms	センサ読み取り、LIN、UART 定期処理[割り込み]
	ch6	500ms	CAN 定期送信[割り込み]
TAU1	ch0/ch6	25us	HSB_CAN_MULTI_1 モータ駆動 PWM
	ch2	10ms	HSB_CAN_MULTI_1 エンコーダセンサ[割り込み]

r_cg_timer.c が API 関数、r_cg_timer_user.c がユーザで追加するソースです。

－使用している API 関数－

R_TAU*n*_Channel*m*_Start() (n=0~1, m=0~7)

タイマを開始する API 関数

R_TAU*n*_Channel*m*_Stop) (n=0~1, m=0~7)

タイマを停止する API 関数

－ユーザ側で記載を追加している関数－

r_tau*n*_channel*m*_interrupt() (n=0~1, m=0~7)

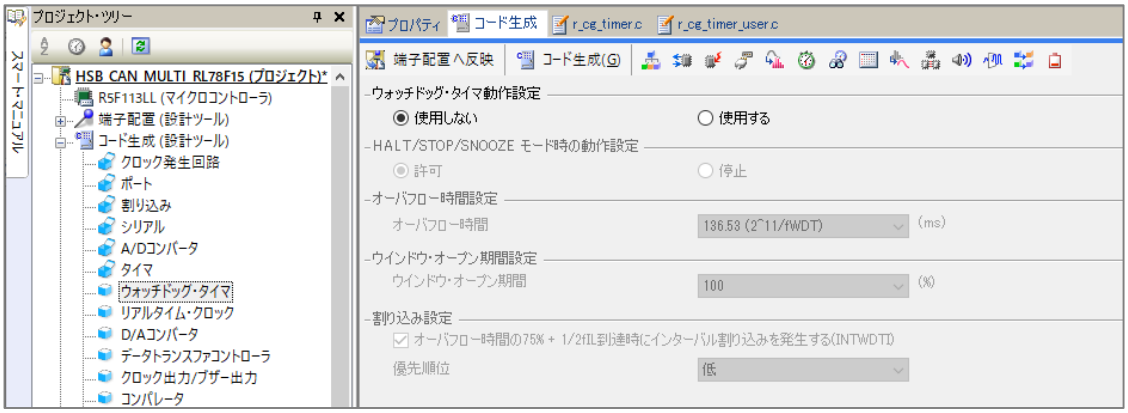
タイマ割り込みで呼ばれる関数

定期処理を記載

```
static void __near r_tau0_channel2_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    timer_tau02();
    /* End user code. Do not edit comment generated here */
}
```

timer.c 内に、timer_tau02()関数を記載しています。（割り込みを有効化している、timer_tau05(), timer_tau06(), timer_tau12()も同様。）

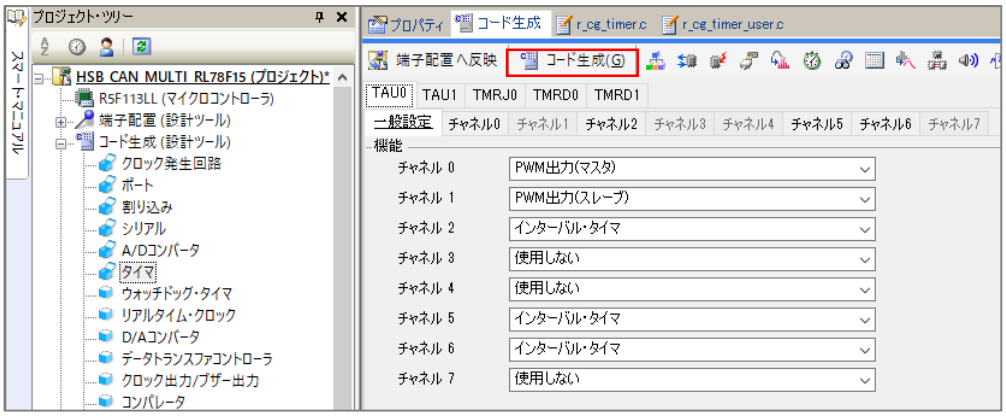
4.2.2. ウオッチドッグ・タイマ

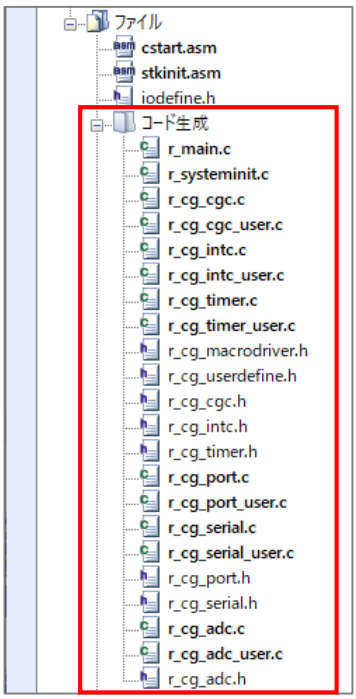


「使用しない」にチェックを入れてください。

ユーザプログラムでは、ウオッチドッグ・タイマをクリアするコードを入れていません。デフォルトの「使用する」を選択すると、マイコンが一定時間毎にリセットが掛る動作となります。

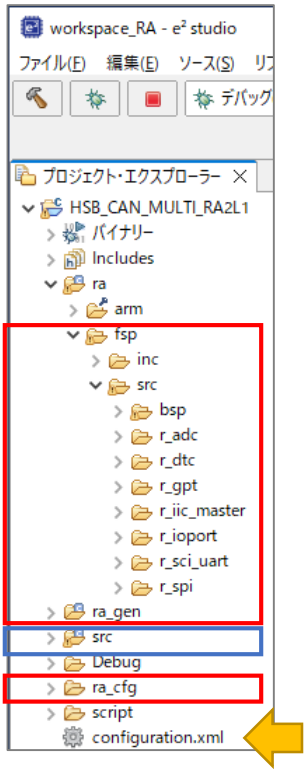
4.2.3. プログラムコードの出力





一通り設定が終わった後で、コード生成のボタンを押すと、プログラムコードが出力されます。
（その後、3 章の手順でプログラムのビルドを行ってください。）

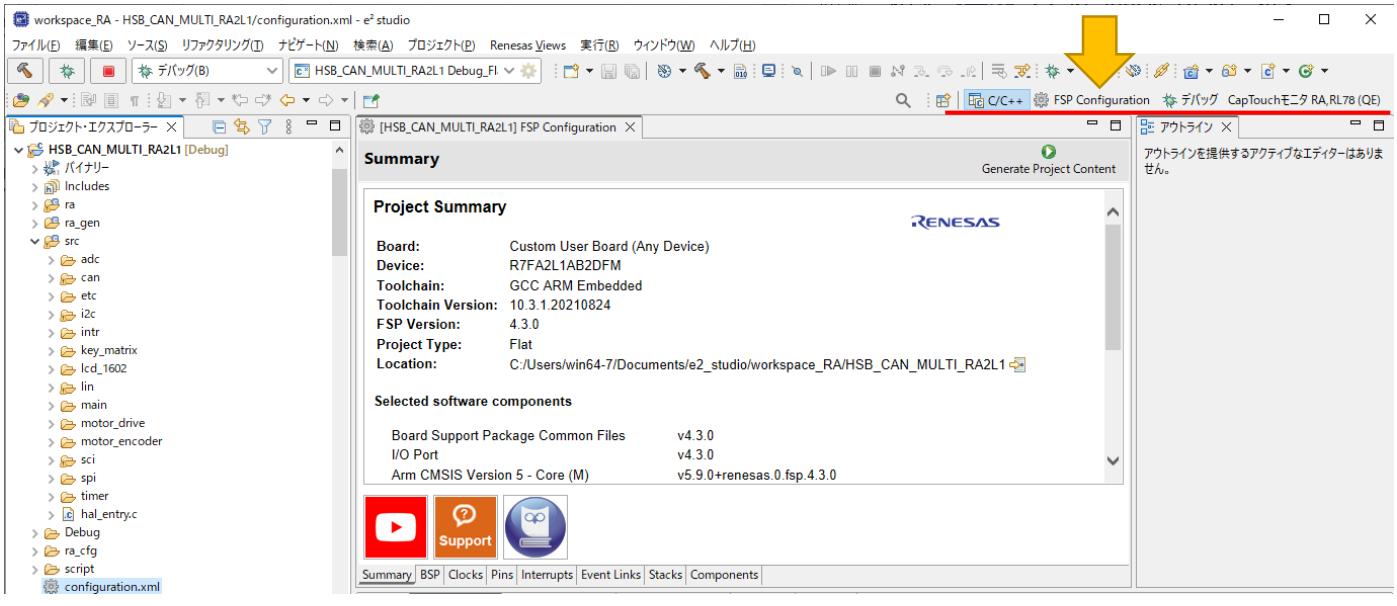
4.3. HSB_CAN_MULTI_RA2L1（FSP）



プロジェクト・エクスプローラでは、赤線で囲まれた部分が、FSP で生成されたプログラムコードです。（FSP では、バイナリ化されたライブラリを使用する訳ではなく、ソースコードの形でプログラムコードが生成されます。）

青線で囲まれた部分が、ユーザ作成コードです。（ユーザ作成コード部分は、「取扱説明書　ソフトウェア編」を参照してください。）

Configuration.xml が FSP の設定ですので、ダブルクリックして、Configuration.xml の画面を開いてください。



e2studio の右上のタブですが、

・「C/C++」 ソースコードを編集する場合に選択

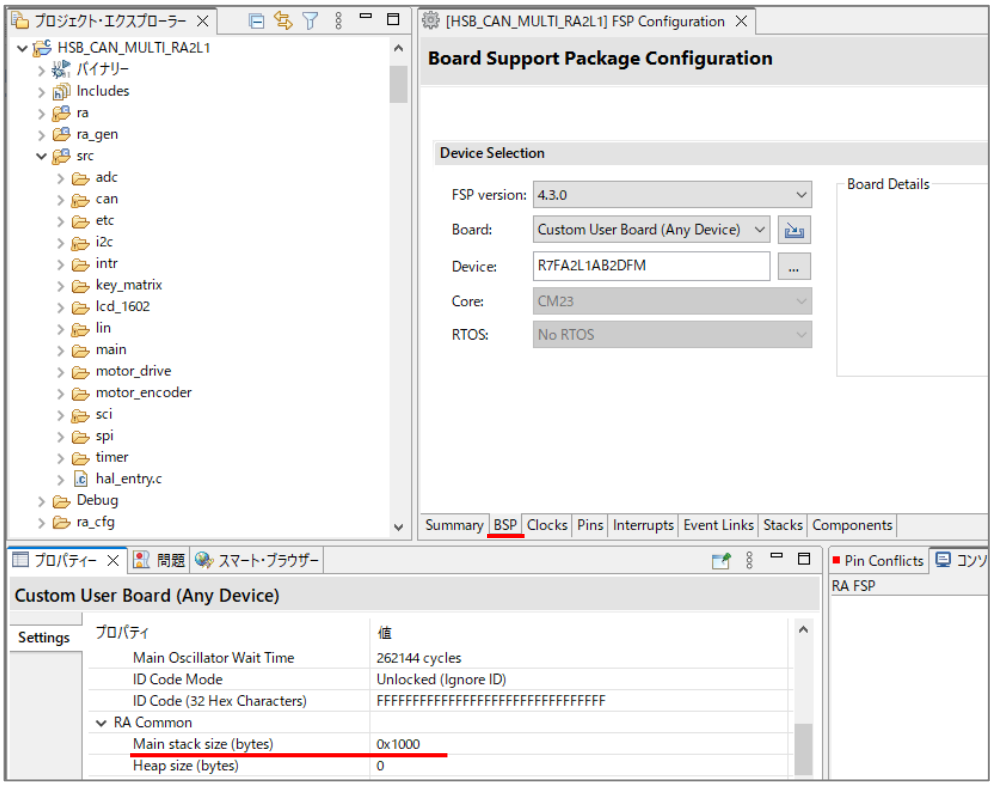
・「FSP Configuration」 FSP の設定を行う場合に選択

・「デバッグ」 デバッグ時に選択

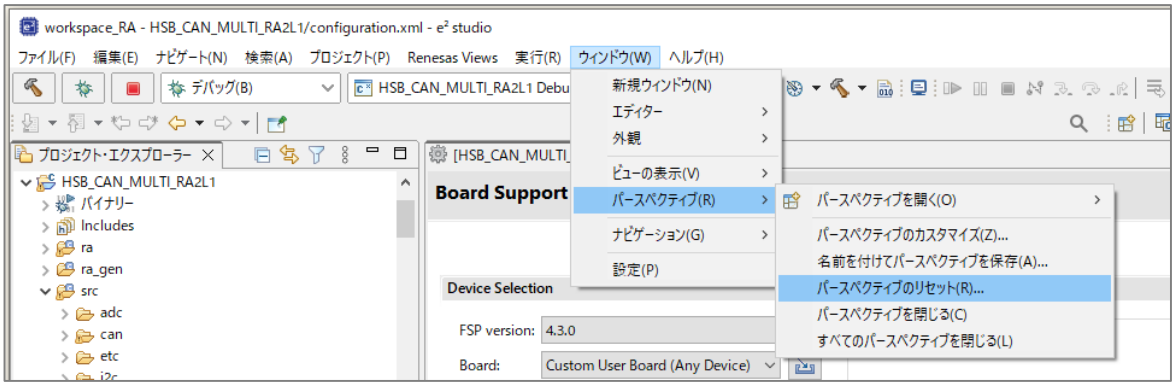
FSP Configuration のボタンを押して、FSP の編集画面とします。

（右上のタブでどれを選択するかで、表示内容が異なりますので、FSP の設定を行う場合は、必ず「FSP Configuration」タブを選択してください。

4.3.1. BSP



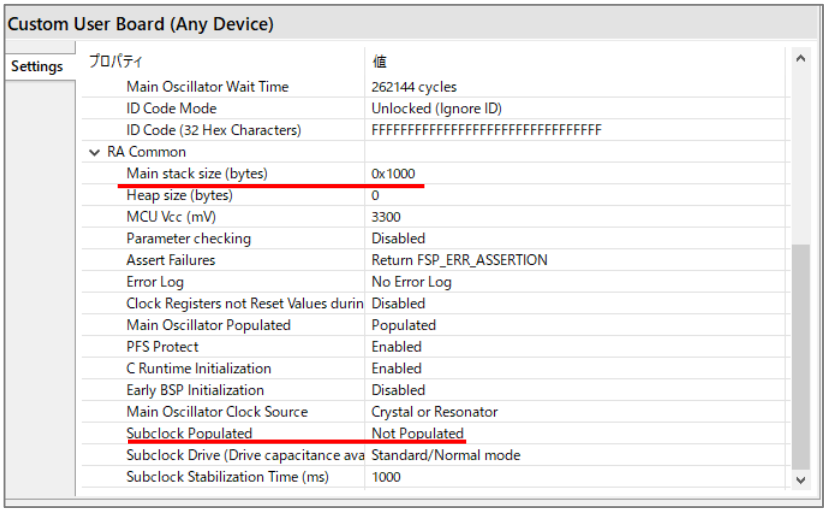
BSP タブを選択し、プロパティの「Main stack size」を 0x1000(4kB)に変更します。



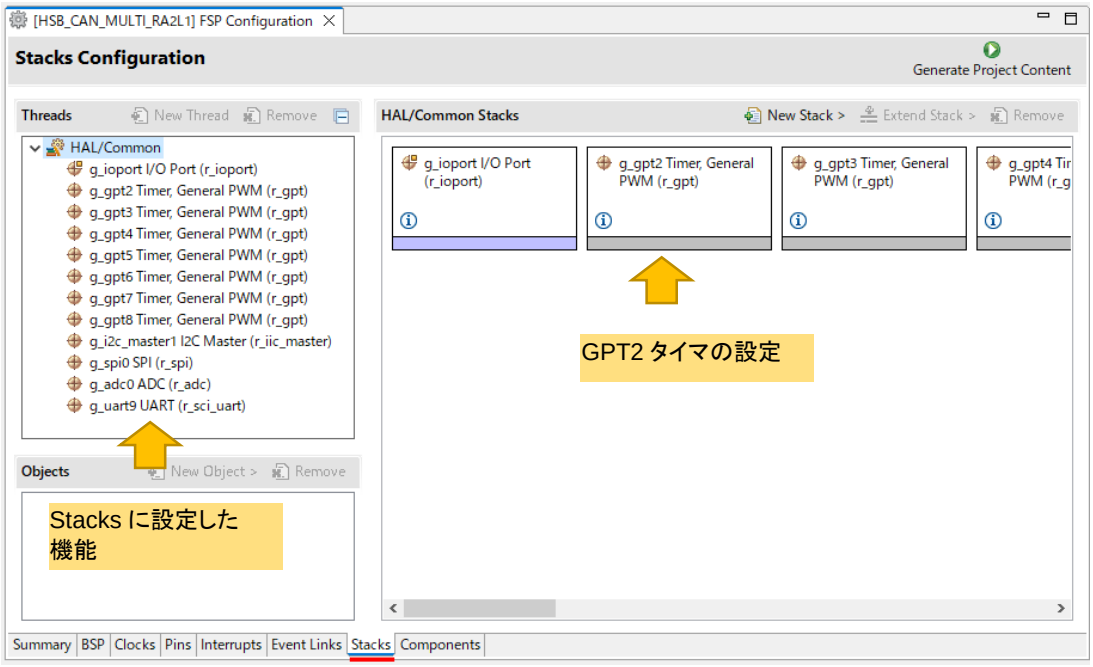
なお、プロパティが表示されない等、画面レイアウトが上記と異なる場合は、

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

を実行して、ウィンドウ配置を初期化してください。（ウィンドウの表示を変更して、必要なものが出てこないという場合は、「パースペクティブのリセット」が有効です。



メインスタックサイズの変更に加え、「Subclock Populated」の項目を「Not Populated」に変更してください。CPU_CARD_RA2L1 には、サブクロックは搭載されていませんので、この項目をデフォルト（Populated，サブクロック発振）とすると、プログラムが発振待ちで進まなくなります。



FSP では、Stacks のタブが、各種周辺機能の設定となります。

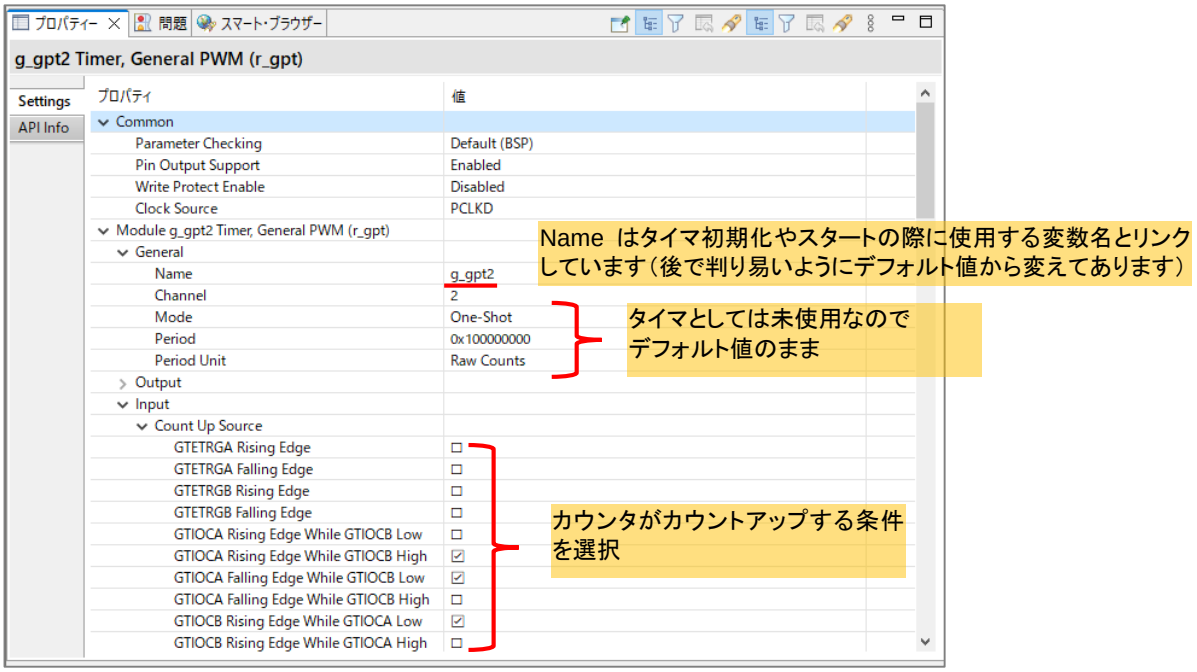
Stacks の中に、使用する周辺機能を追加していきます。本プロジェクトでは上記の機能を Stacks に設定しています。

4.3.1. タイマ(g_gpt*n* Timer (n=2~8))

GPT はタイマの設定です。

タイマ	周期	用途
GPT2		HSB_CAN_MULTI_1 のエンコーダセンサ
GPT3	25us	HSB_CAN_MULTI_1 のモータ制御 PWM
GPT4	10ms	HSB_CAN_MULTI_1 のエンコーダセンサタイマ[割り込み]
GPT5	1ms	HSB_CAN_MULTI_1, HSB_CAN_MULTI_2 のスイッチ読み取り[割り込み]
GPT6	14ms	HSB_CAN_MULTI_4 の LIN 受信[割り込み]
GPT7	500ms	CAN の定期送信[割り込み]
GPT8	50ms	センサ値取得、LIN 等定期処理[割り込み]

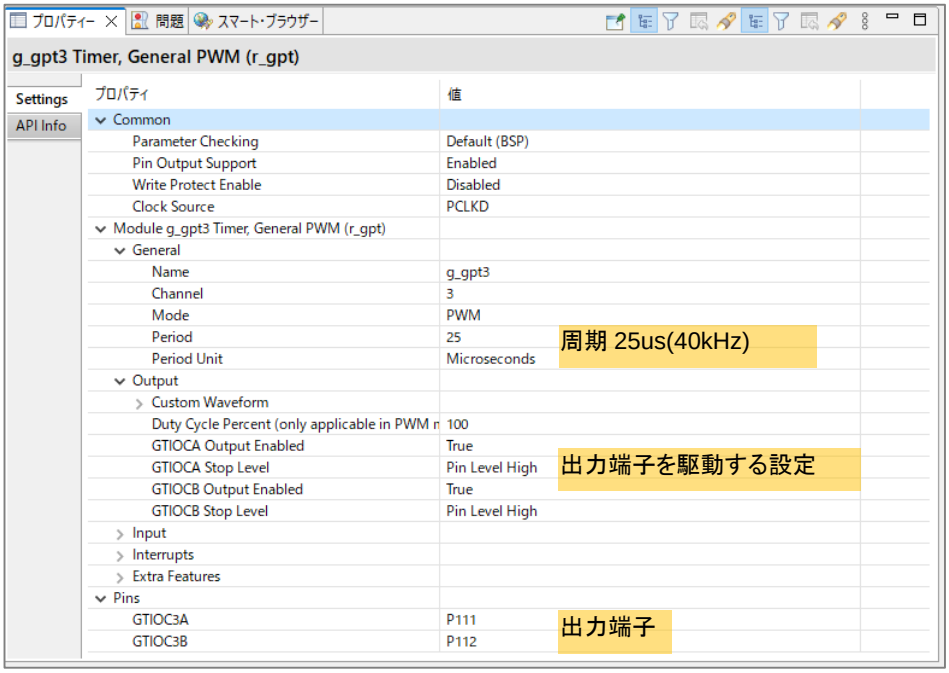
・GPT2



GPT2 は、位相計数カウンタとして使用しています。

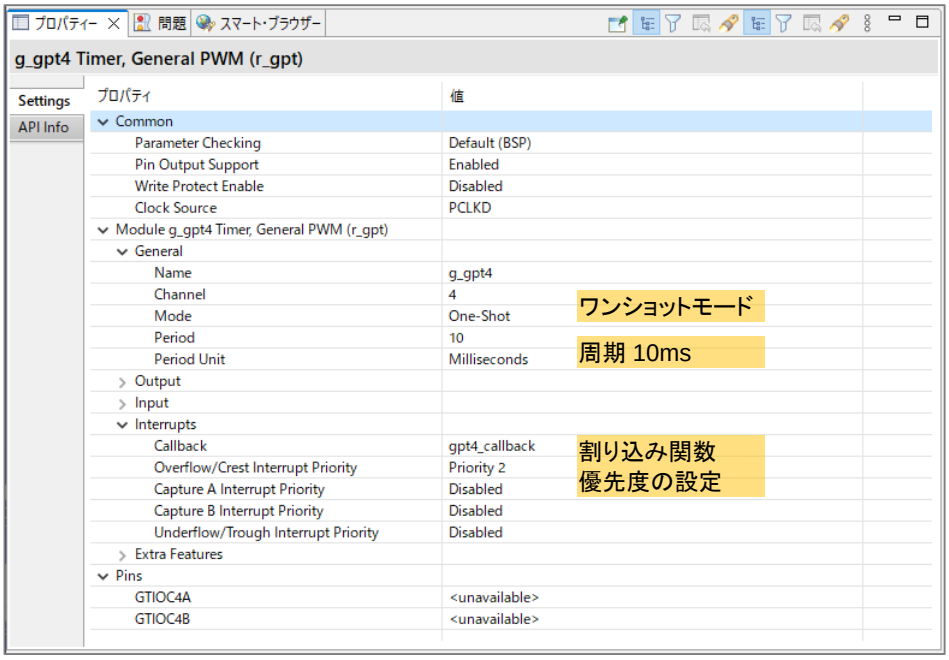
Count Up Source として、GTIOCA(P500)が↑(Rising Edge)の時、GTIOCB(P501)が H レベルであれば、カウントアップ、等の設定を行っています。Count Down Source も同様に設定しています。

・GPT3



GPT3 は、モータを駆動する PWM 波形を生成しています。

・GPT4



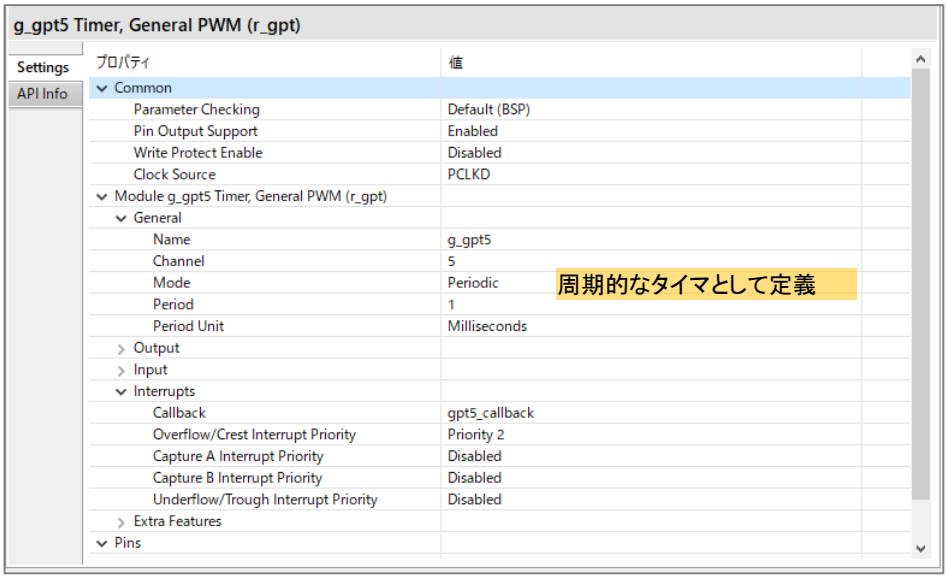
GPT4 は、ワンショットモード（起動すると、周期終わりでタイマが止まるモード）に設定しています。また、周期終わりのタイミングで割り込みを実行したいので、割り込み関数(gpt4_callback())と、Overflow で優先度 2 で割り込みが掛る設定を行っています。

```
//GPT4(10ms)
void gpt4_callback(timer_callback_args_t *p_args)
{
    //10mタイマ割り込み

    //10ms間のエンコーダカウント値を取得

    if (TIMER_EVENT_CYCLE_END == p_args->event)  周期終わりで実行する場合
    {
        g_motor_encoder_value = R_GPT2->GTCNT;

        motor_encoder_stop();

        g_motor_encoder_flag = 0;
    }
}
```

FSP 上で定義した割り込み関数は、この例では motor_encoder.c 内（ユーザ作成プログラムコード内）に記載しています。

RX のスマート・コンフィグレータや RL78 のコード生成では、ツールが生成したファイル(xxx_user.c)内に、割り込みで実行される処理を記載しましたが、RA の FSP では関数本体をユーザ作成コード内に記載します。

・GPT5



GPT5 は、Mode を Periodic で設定しており、この場合は一度タイマをスタートすると 1ms（設定値）毎に、gpt5_callback()関数が実行されます。

・GPT6

Mode One-Short

Period 14ms（仮設定値）

GPT6 は GPT4 同様ワンショットモードとしています。周期は、14ms 設定ですが、この値はタイマ使用時に変更する様にしています。

・GPT7
Mode Periodic
Period 500ms

・GPT8
Mode Periodic
Period 50ms

GPT7, GPT8 は、どちらも周期的な実行としているタイマです。

－使用している API 関数－
R_GPT_Open()
タイマを初期化する API 関数

R_GPT_Start()
タイマを開始する API 関数

R_GPT_Stop()
タイマを停止する API 関数

R_GPT_InfoGet()
タイマの設定値を取得する API 関数

R_GPT_DutyCycleSet()
タイマの duty 設定値を変更 API 関数
モータ駆動 PWM 波形を生成する GPT3 で使用

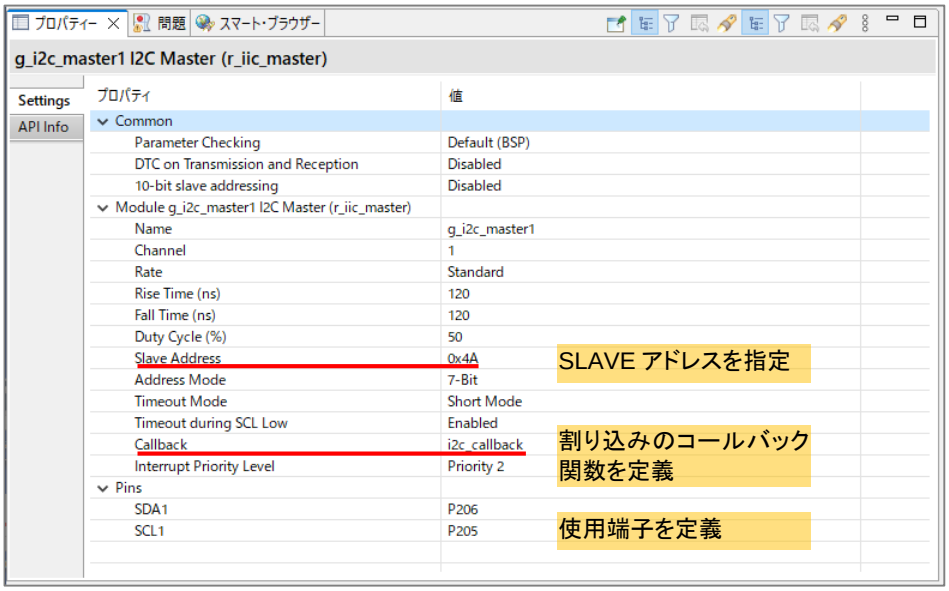
GPT7 を初期化、タイマスタートする場合、下記の様なコードとなります。

```
(void) R_GPT_Open(&g_gpt7_ctrl, &g_gpt7_cfg);
(void) R_GPT_Start(&g_gpt7_ctrl);
```

g_gpt7 は、FSP の Name で設定した名称です。

－割り込み関数－
GPT4 の割り込み関数(gpt4_callback())は、motor_encoder.c 内に、GPT5, GTP7, GPT8 の割り込み関数(gpt*n*_callback(), n=5,7,8)は、timer.c 内に、GPT6 の割り込み関数(gpt6_callback())は、lin.c 内に定義されています。

4.3.2. I2C(g_i2c_master1)



I2C の設定は、上記を定義しています。

－使用している API 関数－

R_IIC_MASTER_Open()

I2C を初期化する API 関数

R_IIC_MASTER_Write()

データ送信 API 関数

R_IIC_MASTER_Read()

データ受信 API 関数

－割り込み関数－

```
void i2c_callback (i2c_master_callback_args_t * p_args)
{
    if (p_args->event == I2C_MASTER_EVENT_TX_COMPLETE)
    {
        g_i2c_tx_flag = 1;
    }
    else if (p_args->event == I2C_MASTER_EVENT_RX_COMPLETE)
    {
        g_i2c_rx_flag = 1;
    }
}
```

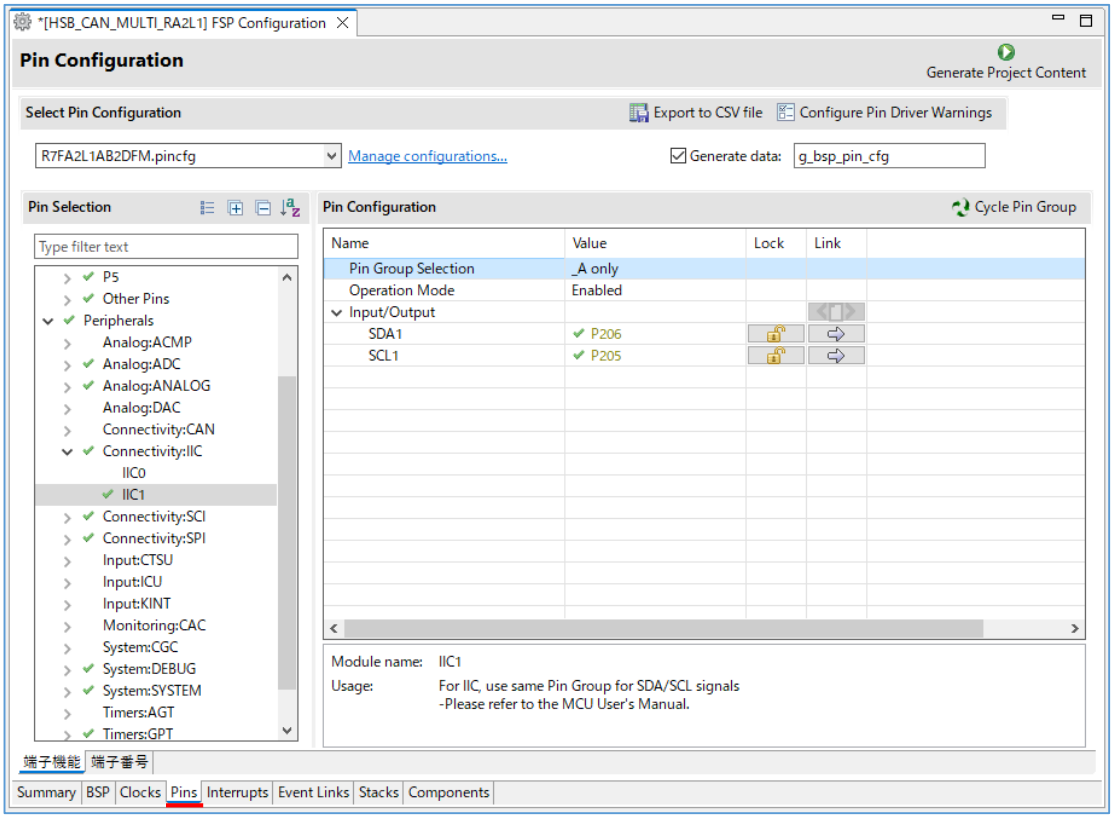
送信完了時の割り込み

受信完了時の割り込み

割り込み関数に関しては、RX, RL78 では、xxx_user.c 内に、送信完了時に呼び出される関数と、受信完了時に呼び出される関数が定義されていましたので、その中にユーザ処理を追加しました。

RA の FSP では、割り込み関数(i2c_callback())自体を、ユーザ側で定義（i2c.c 内に記載）しており、イベントフラグ(p_args->event)の内容により、処理を分ける内容となっています。

－使用端子割り当て－



Pins タブでどの端子を使用するかの設定が行えます。

信号名	マイコン	備考
SCL	P205/SCL1(23)	
SDA	P206/SDA1(22)	
OUT1	P112(37) [IO2]	
IN1	P001/IRQ7(63) [IO17] P409/IRQ6(14) [IO19]	※マイコン側は 2 端子に接続
ALERT	P502(51) [IO7] P015/IRQ7(52) [IO8]	温度センサの出力信号 ※マイコン側は 2 端子に接続

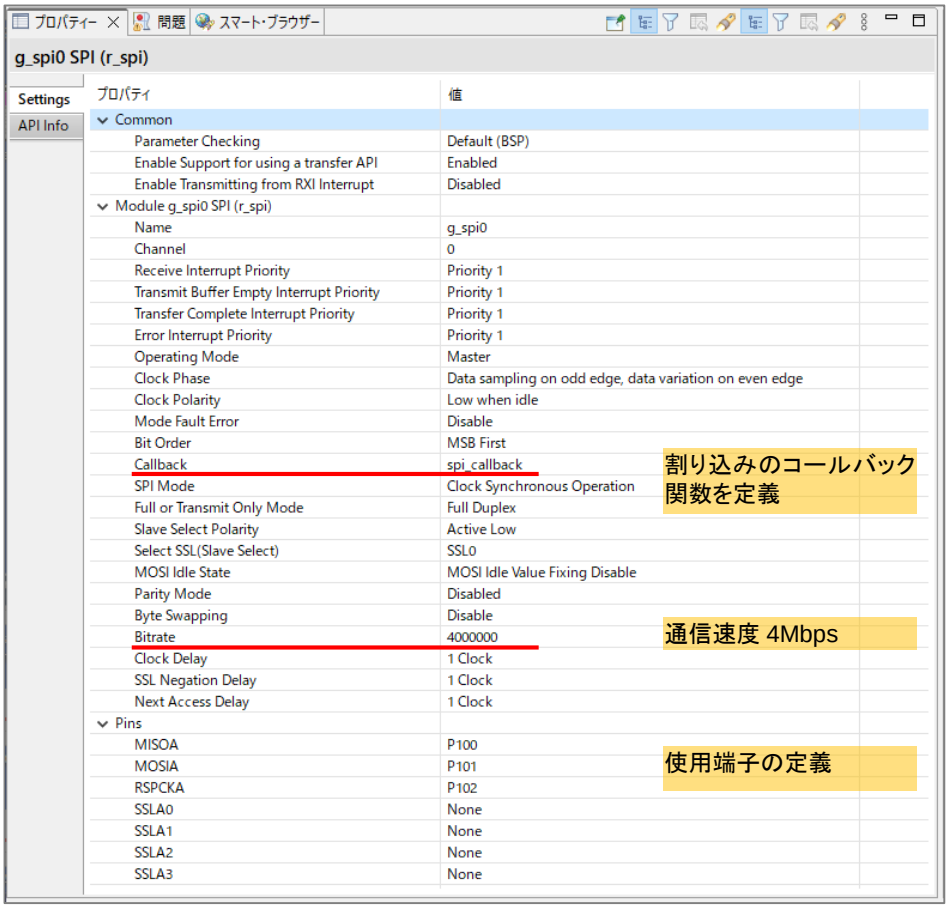
SCL と SDA は、P205, P206 なので、これらの端子を割り当てます。その際、

・「Pin Group Selection」を「_A only」と設定する

・「Pin Group Selection」を「Mixed」に設定し、SCL1 を P205, SDA1 を P206 を選択

のどちらでも問題ありません（最終的に SCL1:P205, SDA1:P206 になっていれば OK です）。

4.3.3. SPI(g_spi0)



I2C の設定は、上記を定義しています。通信速度、極性や通信条件、割り込み関数や使用端子を定義しています。

ー使用している API 関数ー

R_SPI_Open()

SPI を初期化する API 関数

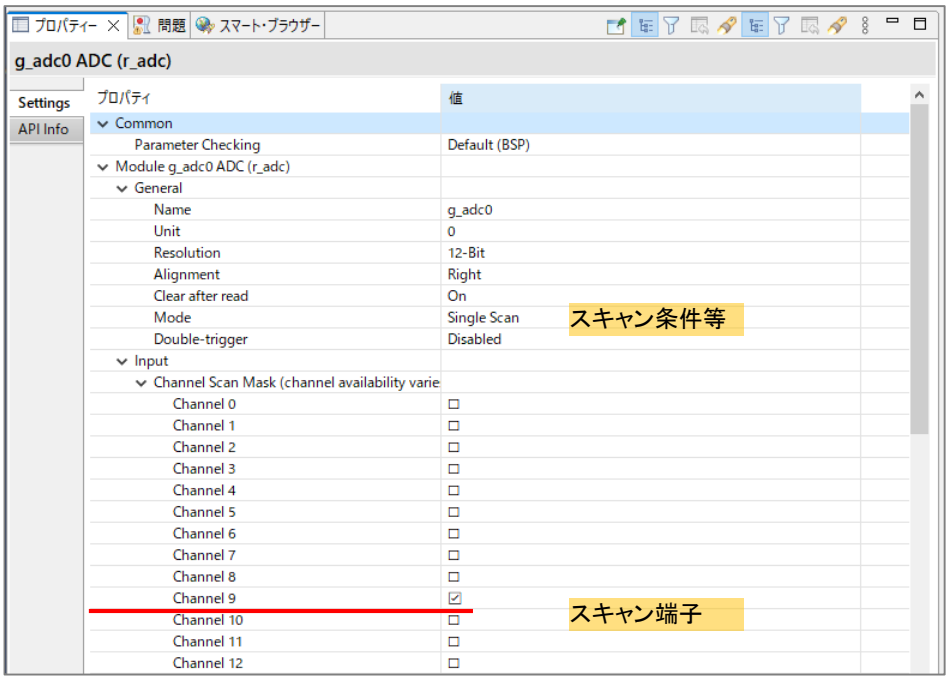
R_SPI_WriteRead()

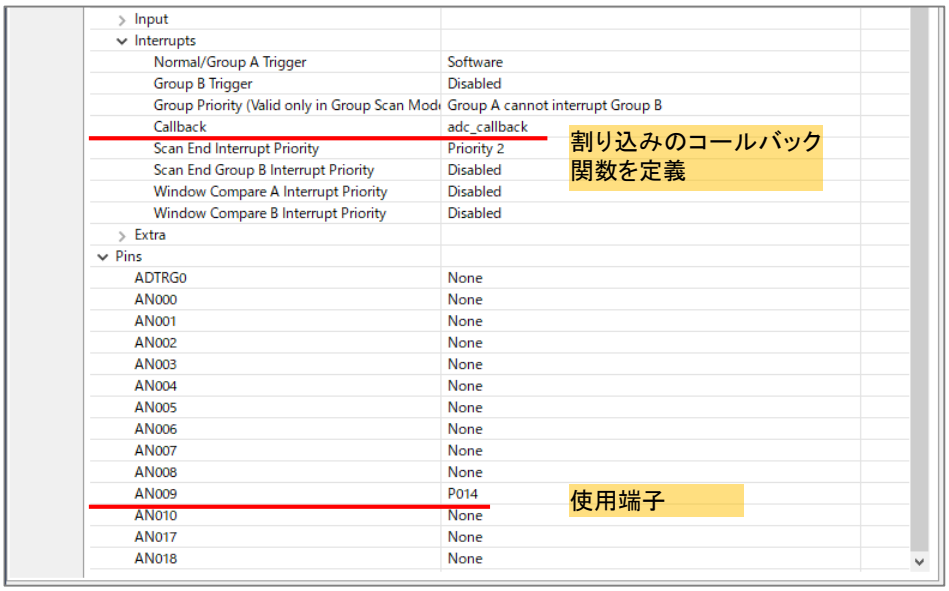
データ送受信 API 関数

ー割り込み関数ー

spi_callback()を、spi.c 内に定義しています。(SPI 通信中フラグを落とす処理)

4.3.4. A/D コンバータ(g_adc0)





ADC では、A/D 変換対象端子の設定や、割り込み関数を定義しています。

－使用している API 関数－

R_ADC_Open()

ADC を初期化する API 関数

R_ADC_ScanCfg()

ADC 初期設定 API 関数

R_ADC_ScanStart()

A/D 変換を実行する API 関数

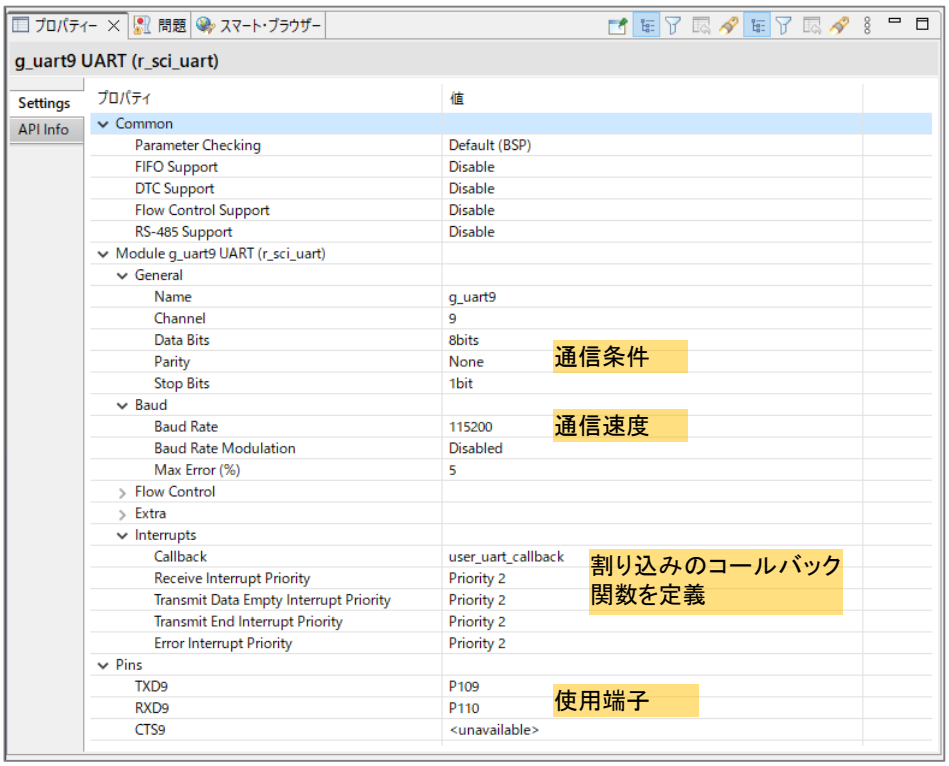
R_ADC_Read()

A/D 変換結果を取得する API 関数

－割り込み関数－

adc_callback()を、adc.c 内に定義しています。（A/D 変換結果の回収と、A/D 変換中フラグを落とす処理）

4.3.1. UART(g_uart9)



UART では、通信条件、通信速度、割り込み関数、使用端子を定義しています。

－使用している API 関数－

R_SCI_UART_Open()

UART(SCI)を初期化する API 関数

R_SCI_UART_Close()

UART(SCI)の使用を終了する API 関数

R_SCI_UART_Write()

送信 API 関数

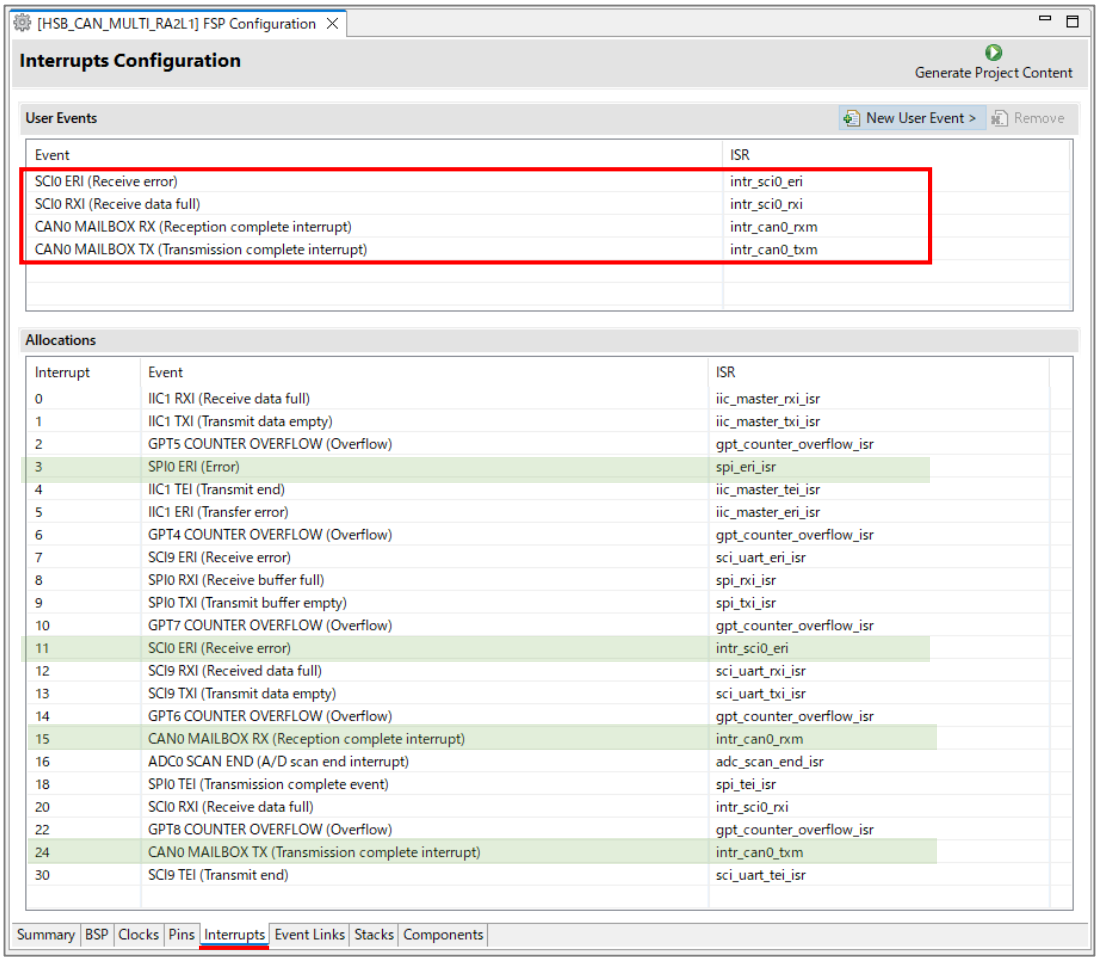
R_SCI_UART_Read()

受信 API 関数

－割り込み関数－

user_uart_callback()を、sci.c 内に定義しています。（ユーザ側で用意しているバッファの内容を送信 API に渡す。受信のタイミングで、受信データをリングバッファに保存する処理。）

4.3.2. 割り込み



Interrupts タブが割り込みの設定です。赤枠で囲まれている(User Events)のが、ユーザ側で追加した割り込みです。Allocations の欄は、割り込みの一覧です。この中で、緑で塗りつぶしされているのが、ユーザ側で追加したものです。（割り込み番号、3, 11, 15, 24）（緑塗りつぶしはツール側が行うのではなく、本図の説明のために追加。）

FSP の Stacks で追加して、割り込みを有効にしたものは、自動的に Allocations に反映されます。

SCI0 と CAN0 に関しては、FSP のコード生成を使用せずにコードを書き下しているので、これらの機能で割り込みを使用したい場合は、New User Ivent のボタンから、割り込みを追加する事となります。

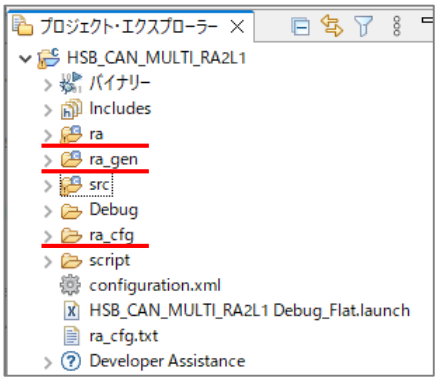
－追加した割り込み－

割り込み番号	割り込み名	割り込み関数名	
3	SCI0 ERI	intr_sci0_eri	SCI0(LIN)のエラー割り込み
11	SCI0 RXI	intr_sci0_rxi	SCI0(LIN)の受信割り込み
15	CAN0 MAILBOX RX	intr_can0_rxm	CAN の受信割り込み
24	CAN0 MAILBOX TX	intt_can0_txm	CAN の送信割り込み

割り込み関数名は、ユーザ側で定義した名称です（任意）。割り込み番号は、RA2L1 の場合は、割り込み元モジュールと割り込み番号には制約がありますので、割り込み元に応じた番号が適切に割り振られます。（割り込みを追加すると、番号がずれることがあります）割り込み番号は、割り込みの有効化や優先度設定の際に使用しています（intr.c 内の関数呼び出しの際に引数として指定）。

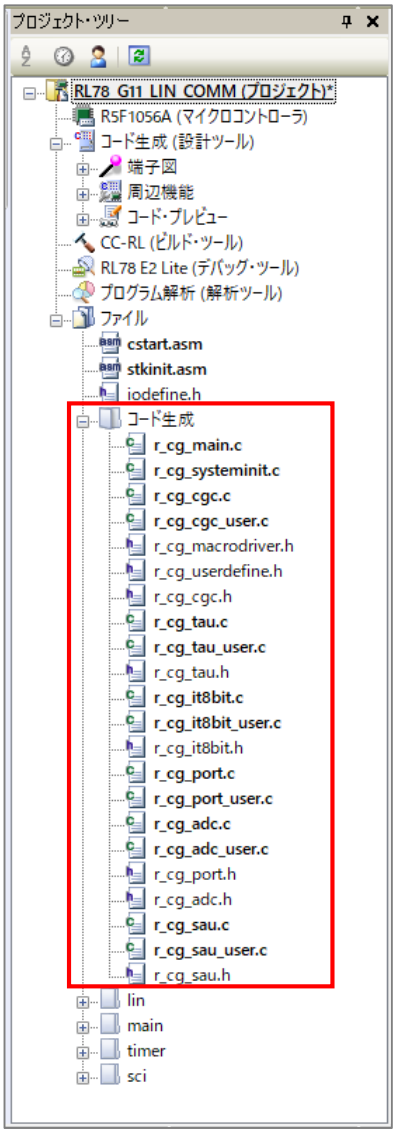
4.3.3. プログラムコードの出力

Generate Project Content のボタンを押してください。



FSP 生成コードが出力、更新されます。

4.4. RL78_G11_LIN_COMM（コード生成）

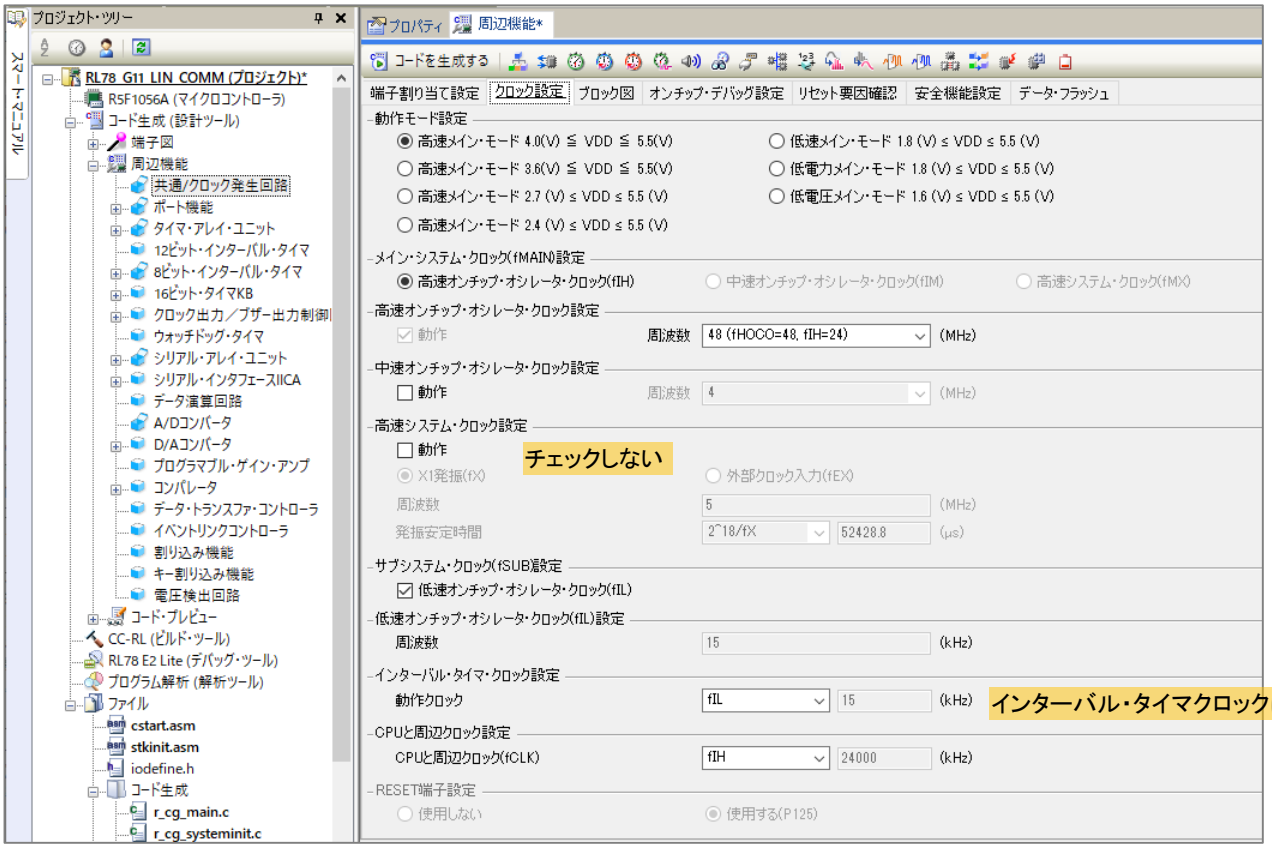


赤線で囲まれた部分が、コード生成で生成されたプログラムコードです。

4.4.1. 端子割り当て設定

デフォルトから変更の必要はありません。

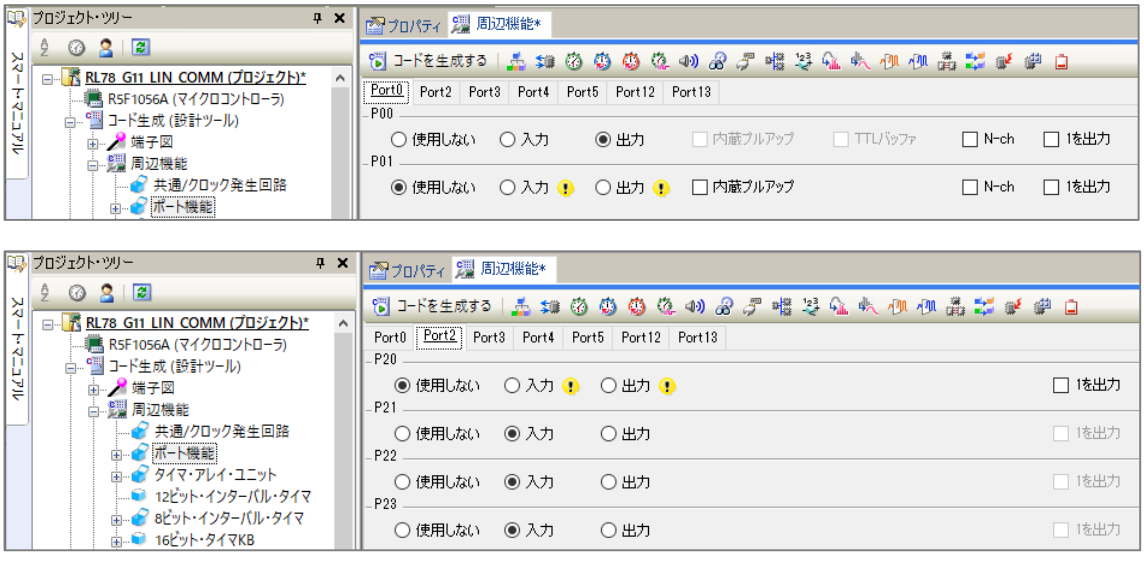
4.4.1. 共通/クロック発生回路



HSB_LIN_COMM には、メインクロック（水晶振動子や発振器）は搭載されていませんので、「高速システムークロック設定」の「動作」チェックボックスのチェックは外してください。

インターバル・タイマは使用していますので、「インターバル・タイマクロック設定」fiL を選択してください。

4.4.2. ポート機能

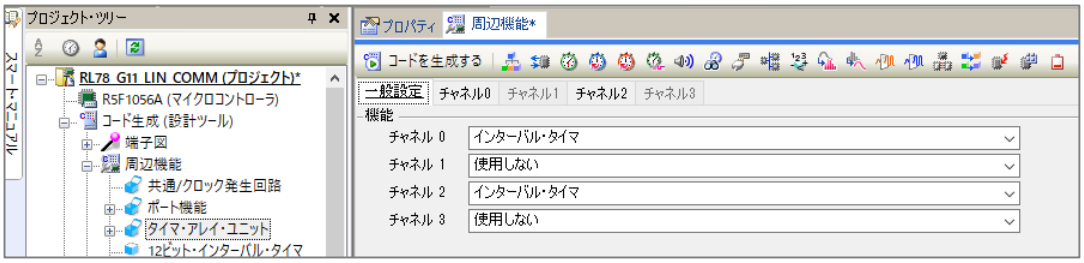


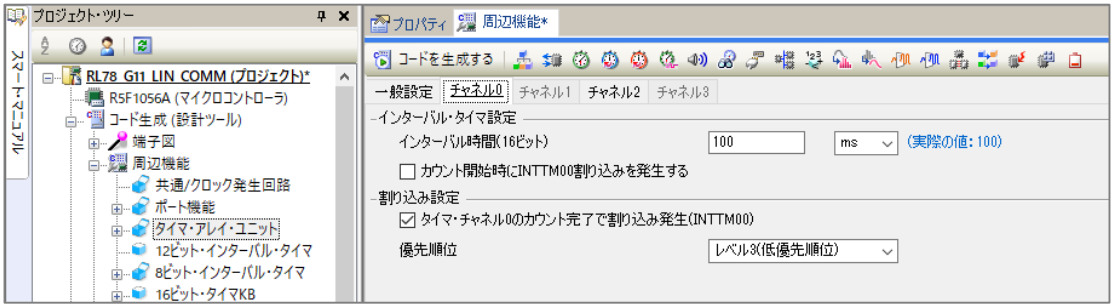
－設定ポート－

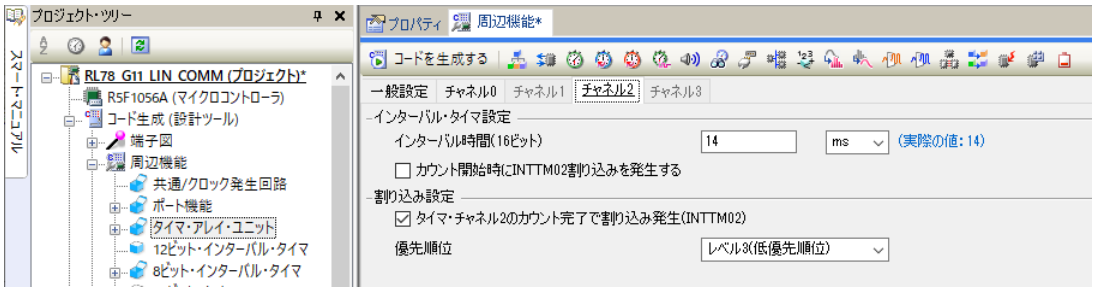
P00	出力	LED2
P01	使用しない(*)	LED1
P21	入力	S2-4(DIP-SW)
P22	入力	S2-3(DIP-SW)
P23	入力	S2-2(DIP-SW)
P30	出力	LED4
P31	入力	JP1（MASTER/SLAVE モード設定）
P33	入力	S2-1((DIP-SW)
P56	出力	LED3

(*)LED 駆動に使用しているので、本来出力設定とするものですが、デバッグ用途で UART1 の出力に割り当てているので、ここでは「使用しない」を選択しています
（デバッグモードに設定しない場合は、ユーザプログラム内で、汎用ポート、出力に設定しています）

4.4.3. タイマ・アレイ・ユニット







タイマは、ch0:100ms, ch2:14ms でどちらも、カウント完了時に割り込みを掛ける設定としています。

ch	周期	用途
ch0	100ms	LIN のヘッダ送信間隔
ch2	14ms	LIN の受信完了タイミング

r_cg_tau.c が API 関数、r_cg_tau_user.c がユーザで追加するソースです。

ー使用している API 関数ー

R_TAU*n*_Channel*m*_Start() (n=0, m=0,2)

タイマを開始する API 関数

R_TAU*n*_Channel*m*_Stop() (n=0, m=0,2)

タイマを停止する API 関数

ーユーザ側で記載を追加している関数ー

r_tau*n*_channel*m*_interrupt() (n=0, m=0,2)

タイマ割り込みで呼ばれる関数

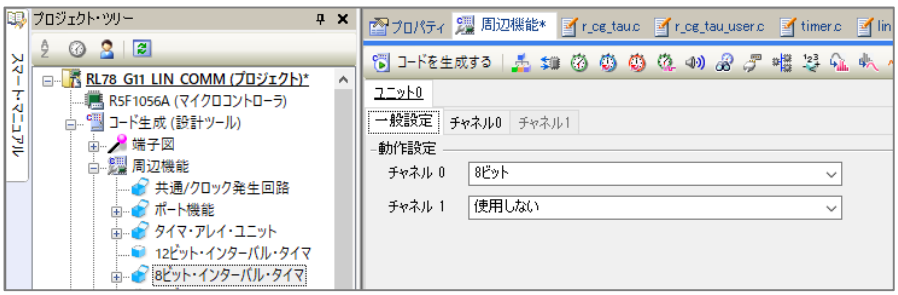
```
static void __near r_tau0_channel0_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    interval_operation();
    /* End user code. Do not edit comment generated here */
}
static void __near r_tau0_channel2_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    lin_post_receice();
    /* End user code. Do not edit comment generated here */
}
```

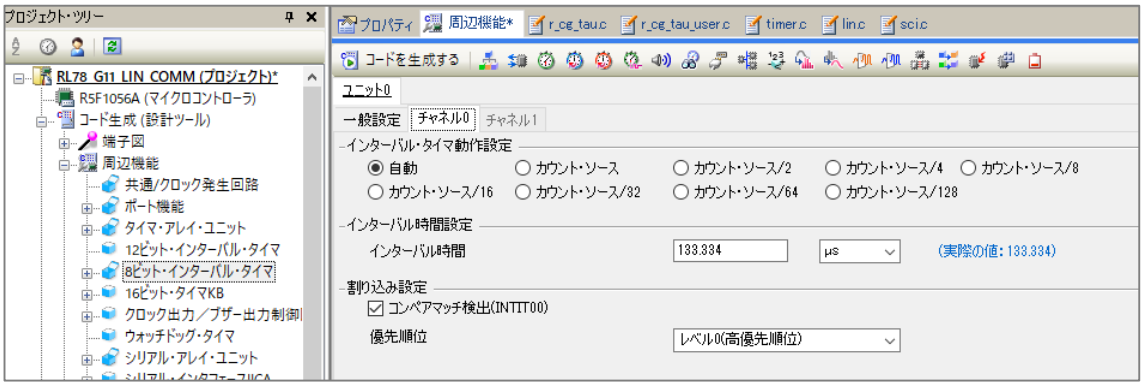
100ms 毎にヘッダ送信を行う処理

MASTER:ヘッダ送信から 12ms 後に
SLAVE:ブレーク検出から 14ms 後に
受信の後処理を行う

timer.c 内に、interval_operation()関数を記載しています。lin.c 内に、lin_post_receive()関数を記載しています。

4.4.4. 8 ビット・インターバル・タイマ





8 ビット・インターバル・タイマは、レスポンス送信のウェイトに使用しています。

r_cg_it8bit.c が API 関数、r_cg_it8bit_user.c がユーザで追加するソースです。

－使用している API 関数－

R_IT8Bit0_Channel0_Start()

タイマを開始する API 関数

R_IT8Bit0_Channel0_Stop()

タイマを停止する API 関数

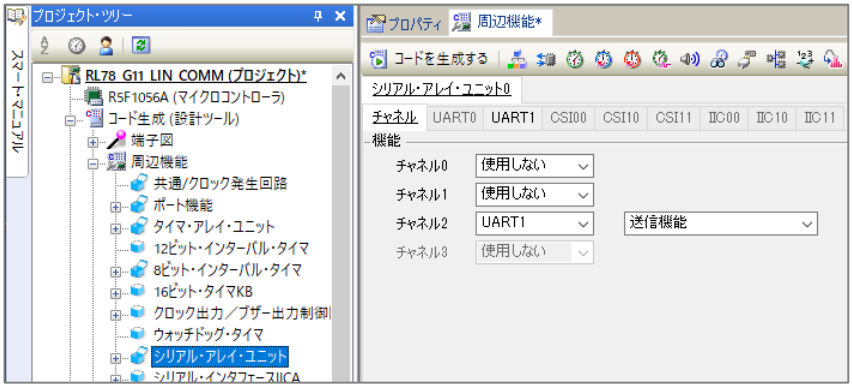
－ユーザ側で記載を追加している関数－

r_it8bit0_channel0_interrupt()

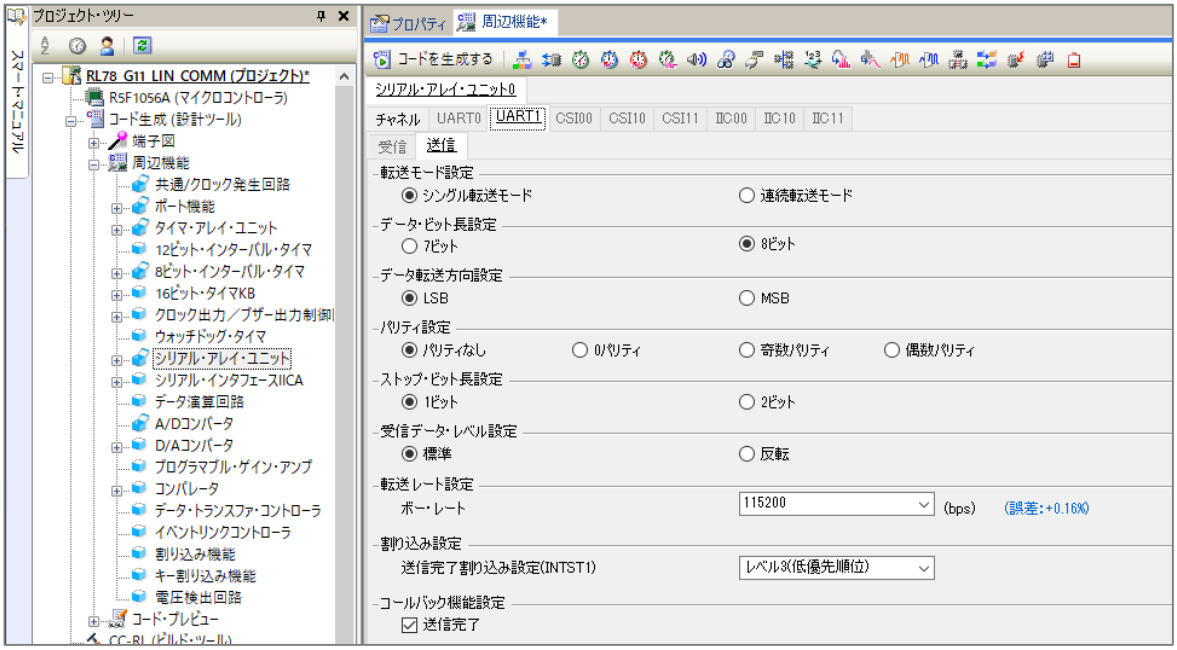
タイマ割り込みで呼ばれる関数

フラグ変数を落とす処理

4.4.5. シリアル・アレイ・ユニット



シリアル・アレイ・ユニットは、UART1 の送信のみ有効化しています。



通常は、UART1 は未使用です。デバッグ出力を有効化(*)してプログラムをビルドした場合に、TXD1 スルーホールから UART のデバッグ出力が有効になります。（その代わり、LED1 が外部の LIN パケットで制御されなくなります。）

(*)main.h の

```
/*----------------------------------------------------------------------------
    定数定義
----------------------------------------------------------------------------*/
//#define SCI_DEBUG//TXD1 でのデバッグメッセージ出力を有効化する
```

上記の//（コメントアウト）を外して、SCI_DEBUG 定数を定義した場合、デバッグ出力が有効となります。

r_cg_sau.c が API 関数、r_cg_sau_user.c がユーザで追加するソースです。

ー使用している API 関数ー

R_UART1_Start()

UART を開始する API 関数

R_UART1_Send()

UART 送信 API 関数

ーユーザ側で記載を追加している関数ー

r_uart1_callback_sendend()

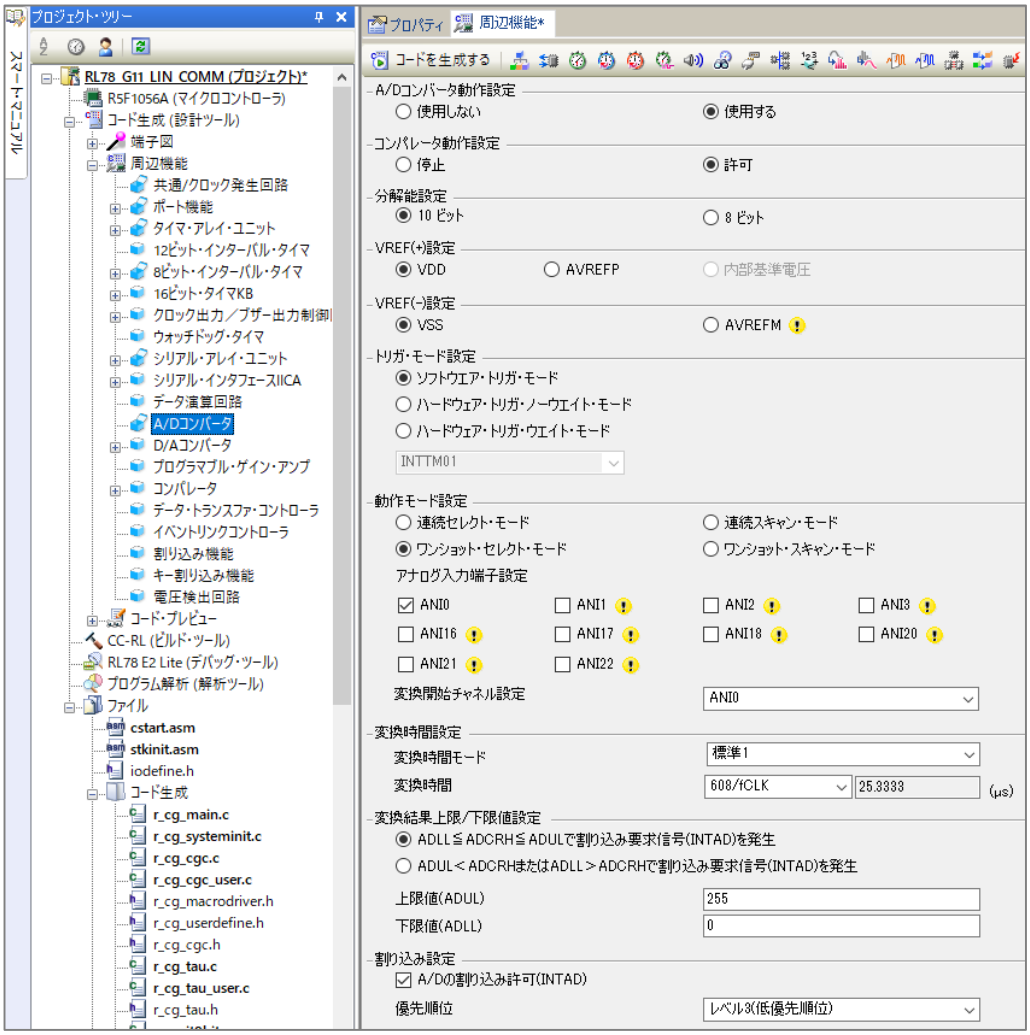
送信完了割り込み時に呼ばれる関数

送信データを再度 API 関数に渡す処理で使用

```
/***********************************************************************************************
***************************
* Function Name: r_uart1_callback_sendend
* Description  : This function is a callback function when UART1 finishes transmission.
* Arguments    : None
* Return Value : None
***********************************************************************************************
***************************/
static void r_uart1_callback_sendend(void)
{
    /* Start user code. Do not edit comment generated here */
    intr_sci_send_end();
    /* End user code. Do not edit comment generated here */
}
```

sci.c を使用する場合、
r_cg_sau_user.c 内に
このコードを追加してください

4.4.1. A/D コンバータ



ボード上のフォトダイオード（明るさセンサ）の値を読み取るのに、A/D コンバータを使用しています。
ANI0(P20)をアナログ入力端子に設定。

r_cg_adc.c が API 関数、r_cg_adc_user.c がユーザで追加するソースです。

－使用している API 関数－

R_ADC_Start()

A/D 変換を開始する API 関数

タイマ内で定期的に実行

R_ADC_Get_Result()

A/D 変換結果を取得する API 関数

A/D 変換終了割り込み内で使用

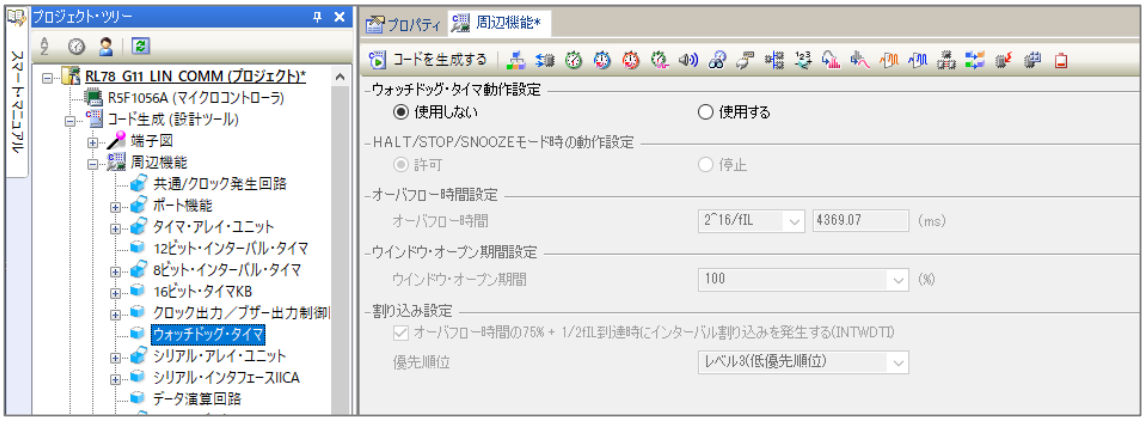
－ユーザ側で記載を追加している関数－

r_adc_interrupt()

A/D 変換終了時に呼ばれる関数

A/D 変換中フラグを落とす処理を記載

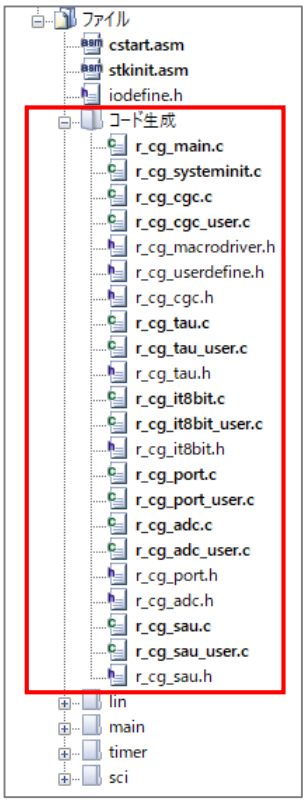
4.4.2. ウオッチドッグ・タイマ



ウオッチドッグ・タイマは、使用しない設定です。

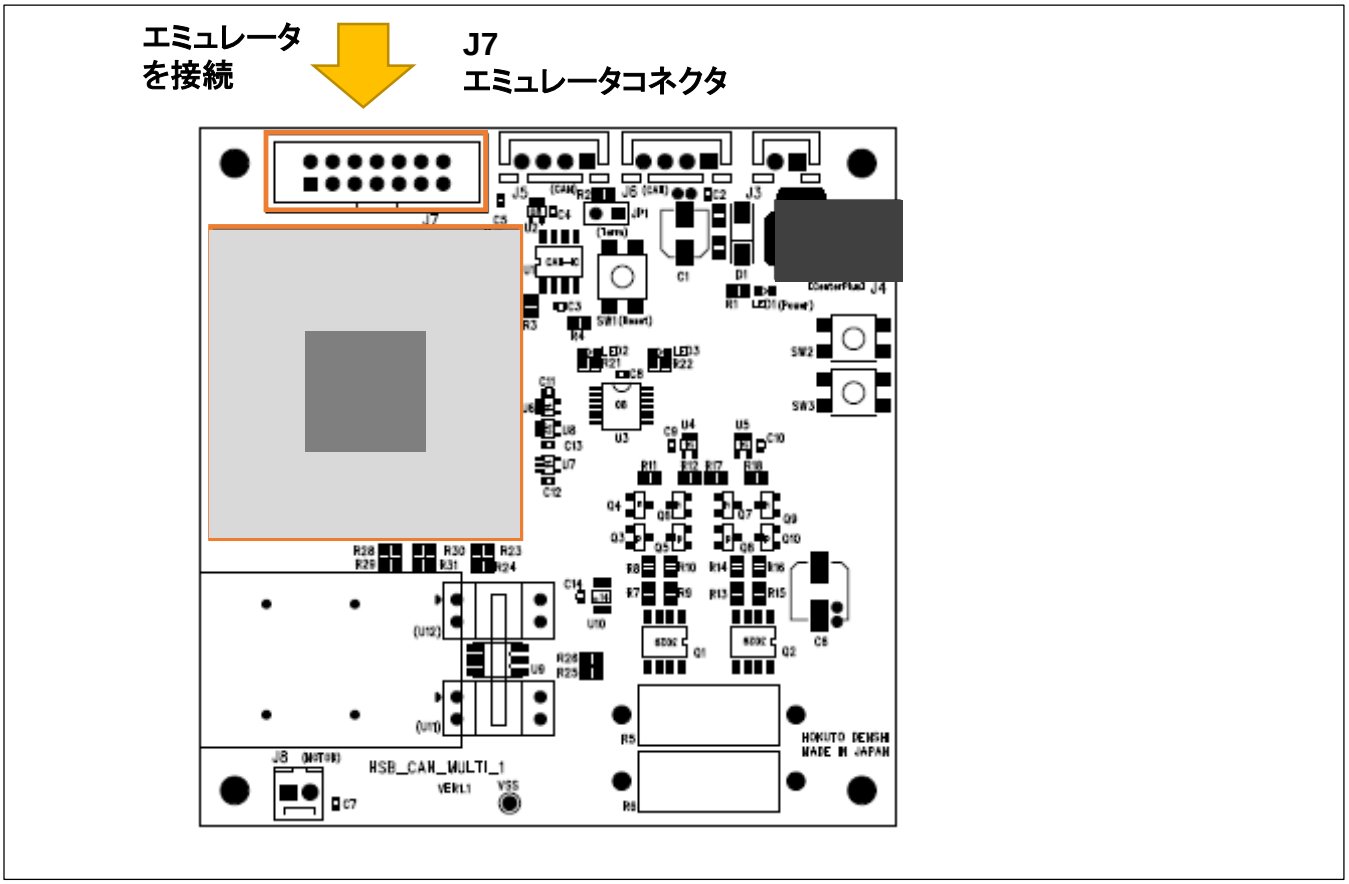
ユーザプログラムでは、ウオッチドッグ・タイマをクリアするコードを入れていません。デフォルトの「使用する」を選択すると、マイコンが一定時間毎にリセットが掛る動作となります。

4.4.3. プログラムコードの出力

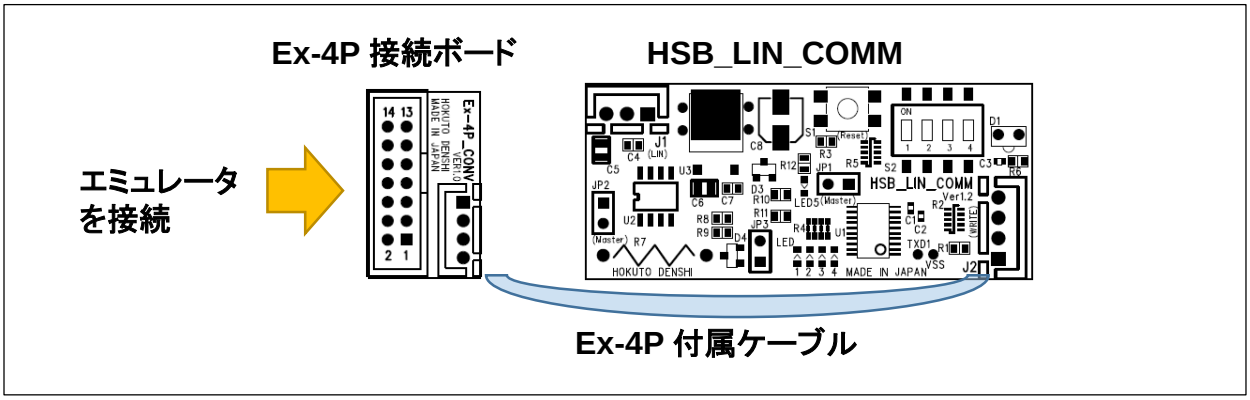


一通り設定が終わった後で、コード生成のボタンを押すと、プログラムコードが出力されます。
（その後、3 章の手順でプログラムのビルドを行ってください。）

5. プログラムのデバッグ



（上記ボードは、HSB_CAN_MULTI_1 ですが、他のボードも同様です）



エミュレータ（デバッガ）接続は、ボードの 14P コネクタに接続してください。E2, E20 の場合は、エミュレータ付属の 14P 変換コネクタ経由で接続してください。

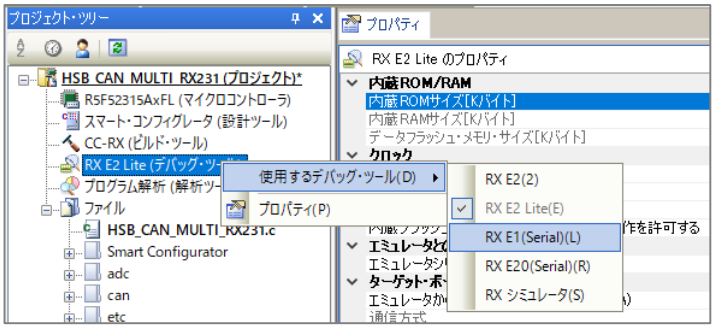
5.1. CS+

CS+プロジェクト、
・HSB_CAN_MULTI_RX231
・HSB_CAN_MULTI_RL78F15
・RL78_G11_LIN_COMM
プロジェクトに関して。

プログラムのデバッグには、ルネサスエレクトロニクス製、E2, E2Lite, E1, E20 が使用可能です。

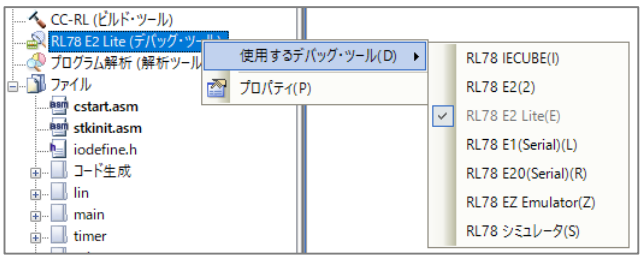
CS+プロジェクトを開き、

・HSB_CAN_MULTI_RX231

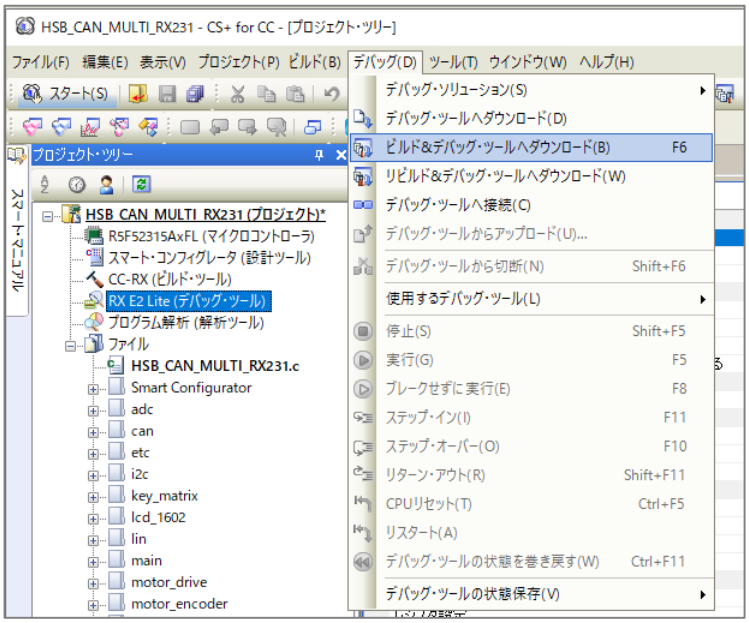


RX E2 Lite（デバッグ・ツール）を右クリックすると、使用エミュレータの変更が行えます。
（CD に含まれるプロジェクトの初期値は、E2 Lite が選択されています。）
RX シミュレータは、PC 上のソフトウェアで RX マイコンの動作を模擬するモードでえすので、実機デバッグは行えません。その他の、「RX E2」「RX E2Lite」「RX E1」「RX E20」のいずれかを選択してください。

・HSB_CAN_MULTI_RL78F15
・RL78_G11_LIN_COMM

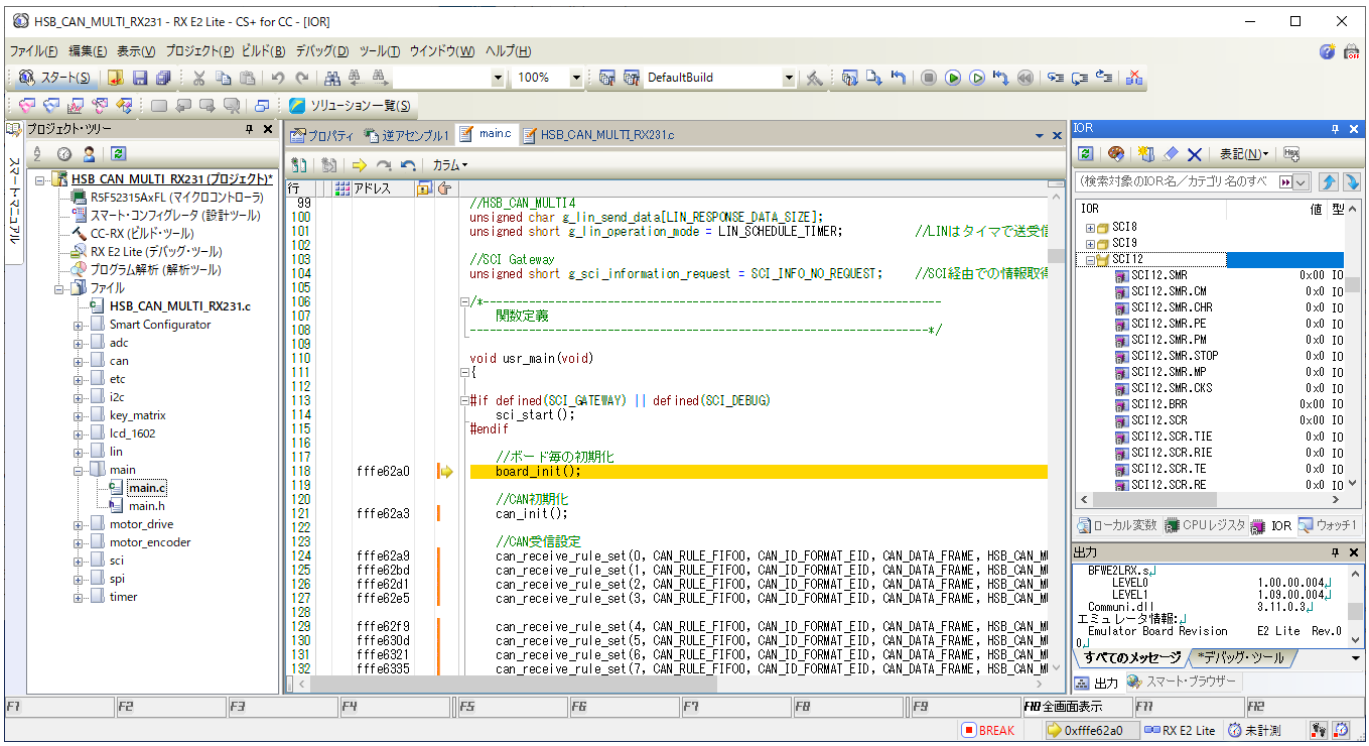


ターゲットマイコンが RL78 の場合も同様に、（デバッグ・ツール）を右クリックすると、使用エミュレータの変更が行えます。



デバッグ－ビルド＆デバッグ・ツールへダウンロード

を選択すると、作成したプログラムが、デバッガ経由でマイコンに書き込まれて実行されます。



デバッガ接続を行うと、

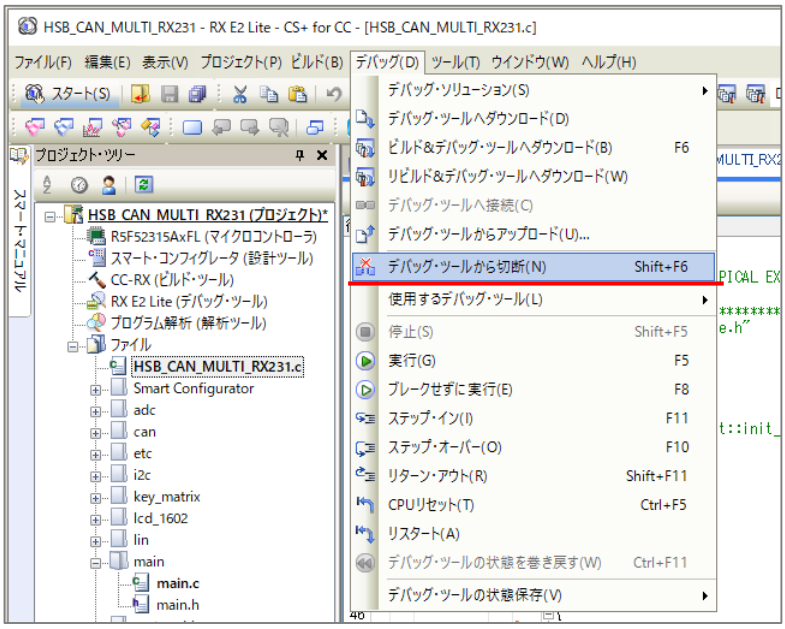
・変数のモニタ

・レジスタ値のモニタ

・メモリ値のモニタ

・ステップ実行

等を行う事ができます。



デバッグの終了時は、「デバッグ・ツールから切断」ボタンを押してください。

※エミュレータを外したり、ボードの電源を切る前に「デバッグ・ツールから切断」を押してエミュレータとボードの切り離しを行ってください

・HSB_CAN_MULTI_RX231

デバッグ時は、「CC-RX(ビルド・ツール)」の「最適化レベル」の設定を「0(-optimize=0)」とすると、
・変数がモニタしやすい
・ユーザが書いたプログラムコード通りに実行される
様になります。

デフォルトでは、「2(-optimize=2)」となっており、コンパイル時に最適化が入り、変数が CPU レジスタ割り付けとなったり、プログラムコードの実行順の変更や組み換えが行われます。「0(-optimize=0)」とした場合、最適化 OFF となり、変数がメモリ割り付け、プログラムコードの実行順がソースコード通りとなります。

・HSB_CAN_MULTI_RL78F15

・RL78_G11_LIN_COMM

RL78 でも同様、「CC-RL（ビルド・ツール）」の「最適化レベル」の設定を「デバッグ優先(-Onothing)」とすると、デバッグが容易となります。

※場合によっては、最適化 OFF と最適化 ON（デフォルト）で動作が変わる事もあります

5.2. e2studio

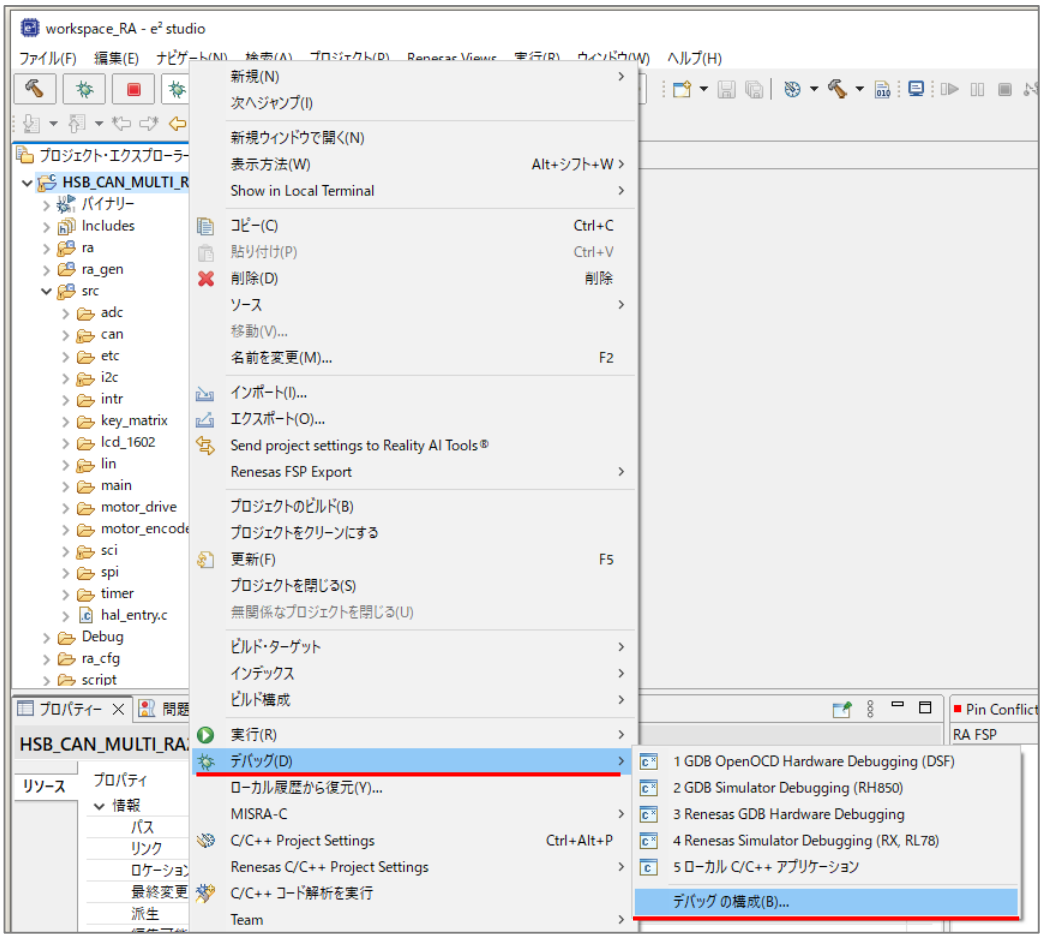
e2studio プロジェクト、

・HSB_CAN_MULTI_RA2L1

プロジェクトに関して。

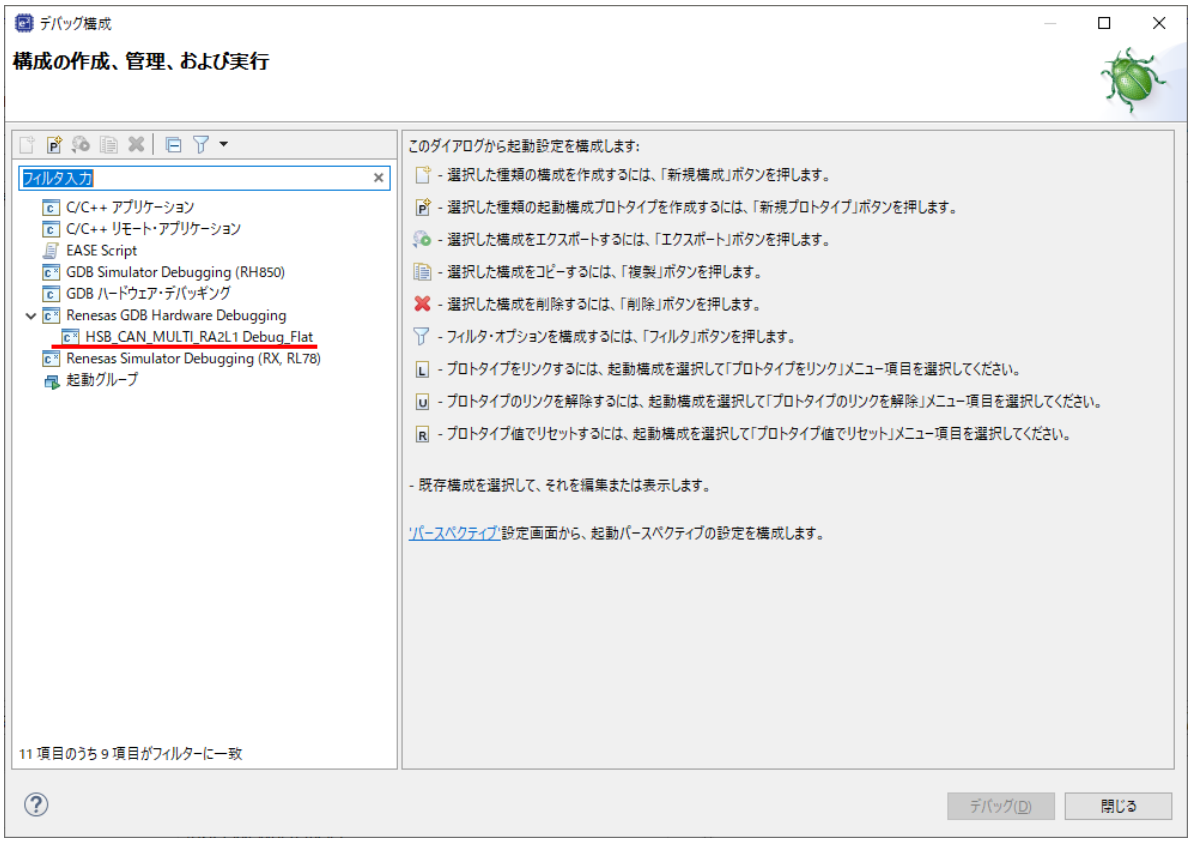
また、CS+プロジェクトを、e2studio にインポートして使用する場合も、こちらを参照ください。

プロジェクト(HSB_CAN_MULTI_RA2L1)を開いた状態で、



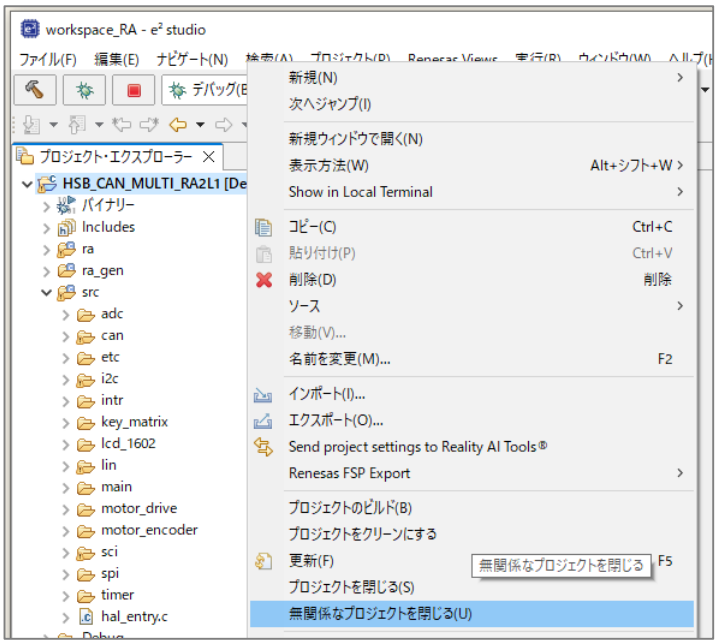
プロジェクト名で右クリック。

デバッグ－デバッグの構成を開く。

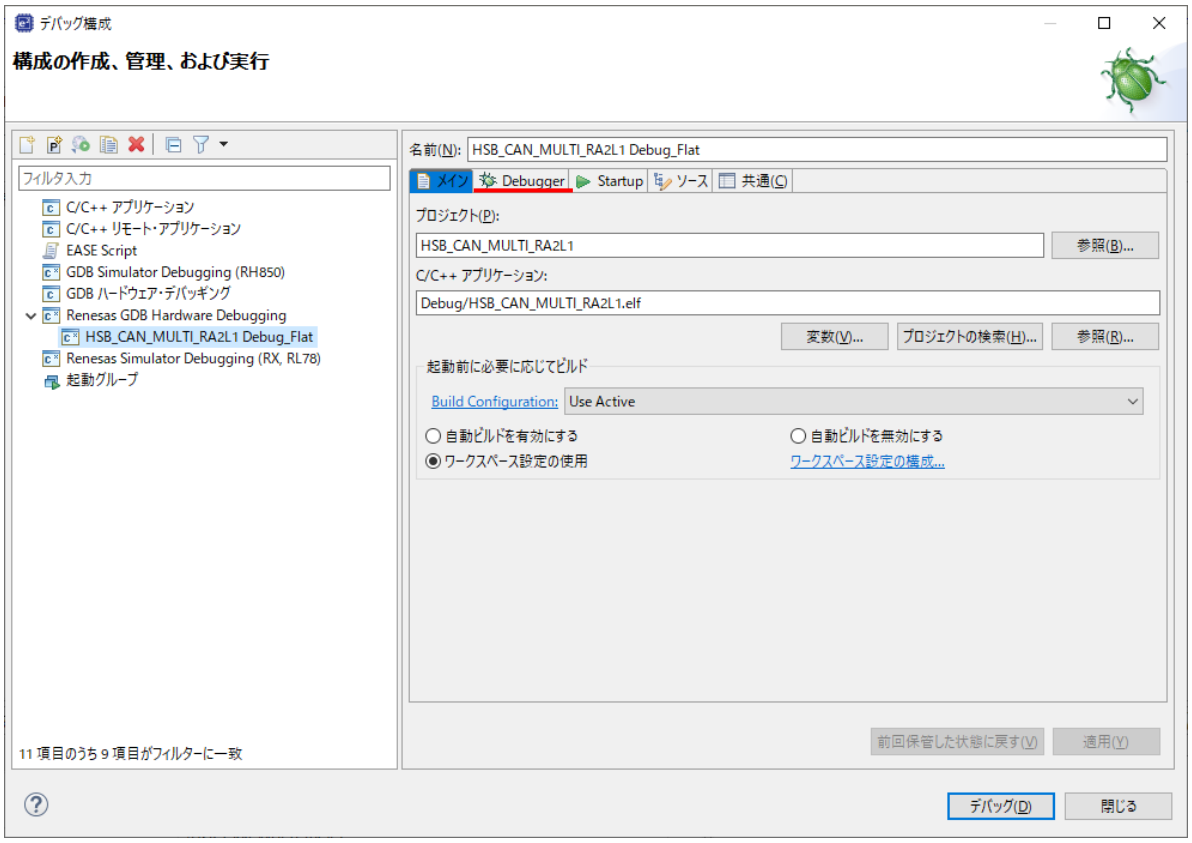


「HSB_CAN_MULTI_RA2L1_Debug_Flat」をクリック

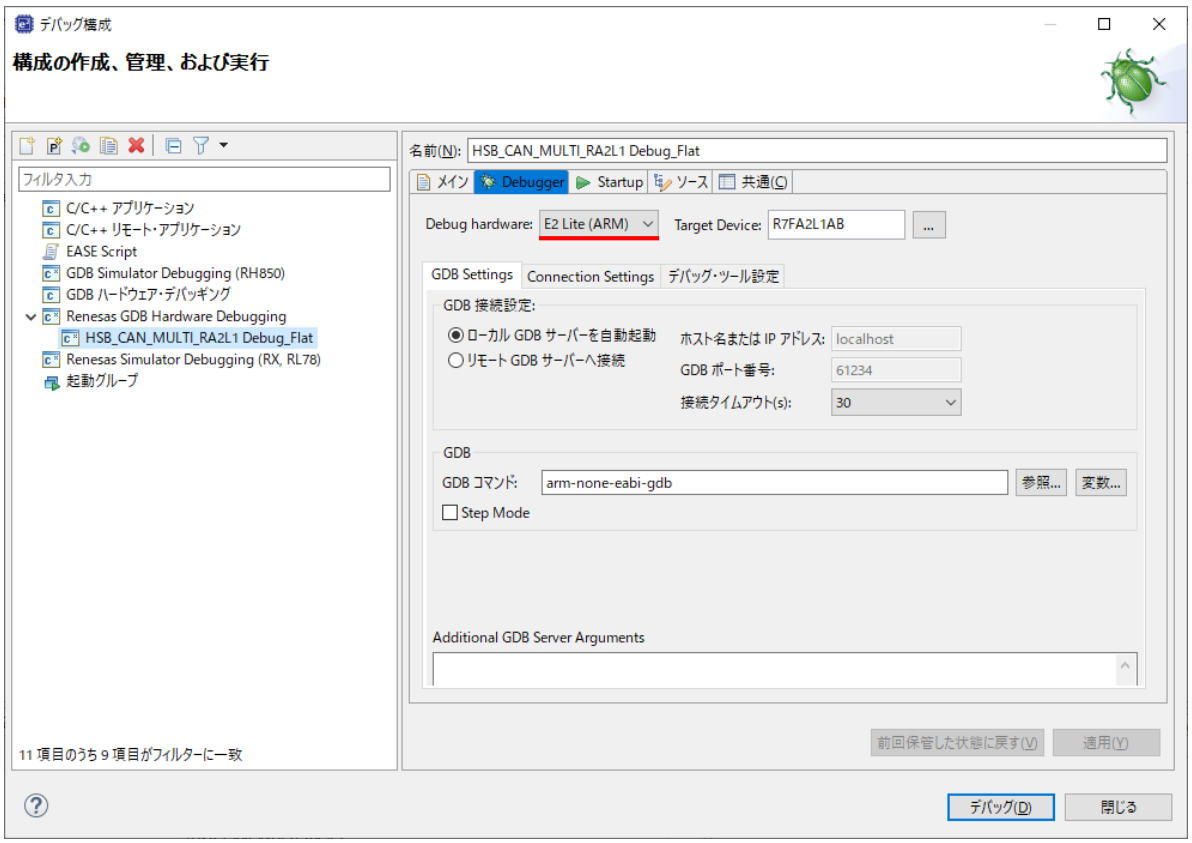
※Renesas GDB Hardware Debugging の下に複数の構成がぶら下がっている場合は、複数のプロジェクトが開いた状態になっていますので、



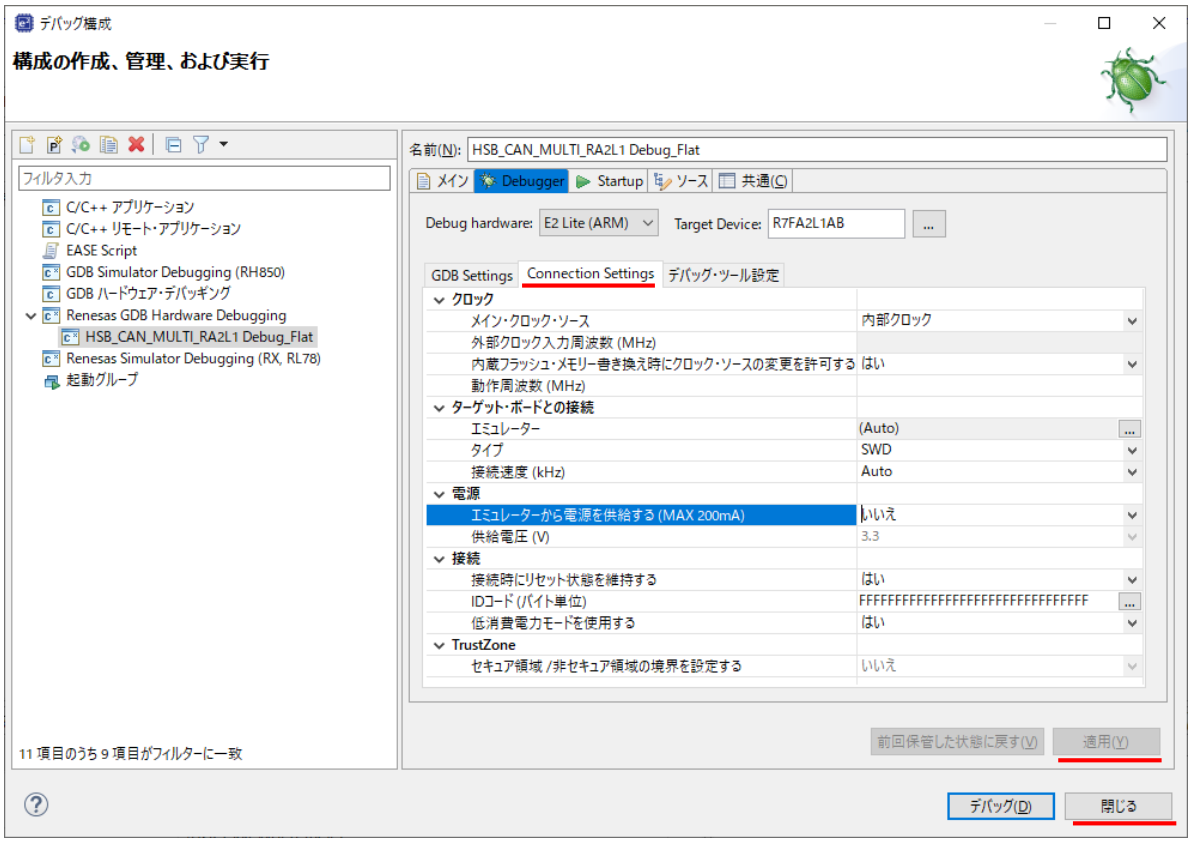
プロジェクト名を右クリックー「無関係なプロジェクトを閉じる」を実行して、開いているプロジェクトを 1 つだけ(HSB_CAN_MULTI_RA2L1)にしてください。(「無関係なプロジェクトを閉じる」がグレーアウトしていれば問題ないです)



Debugger タブを選択



使用するエミュレータを選択（赤線部分）

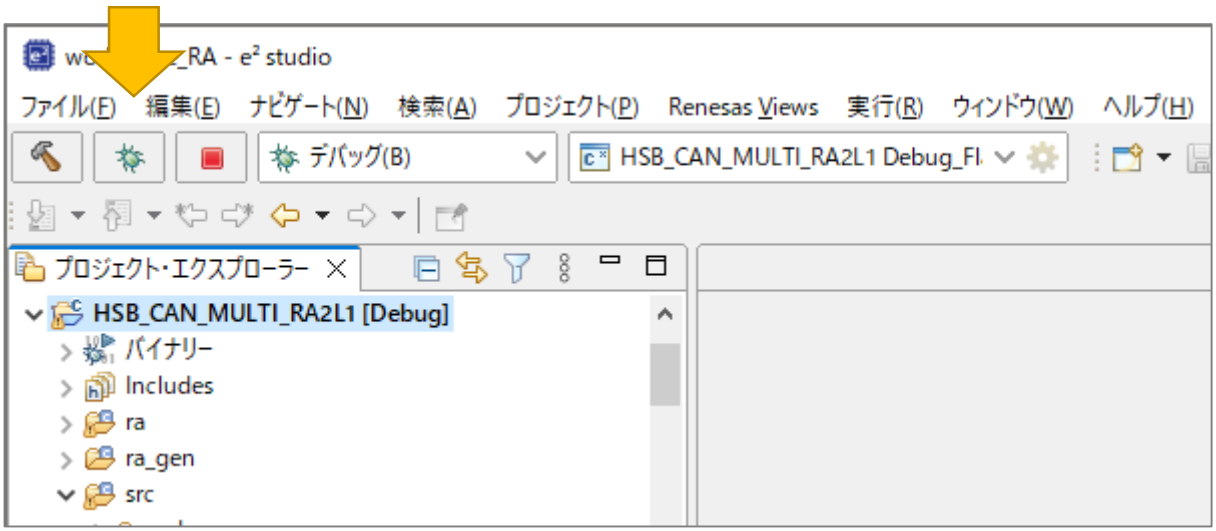


Connection Settings タブを選択

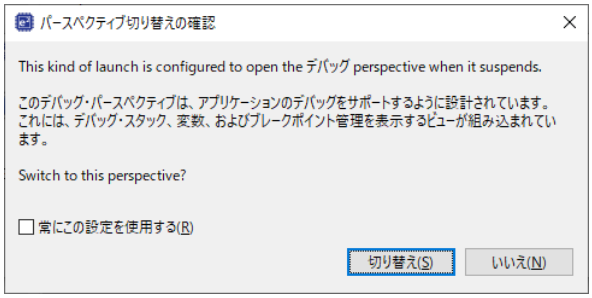
「エミュレータから電源を供給する」「いいえ」

を選択（CD 内のプロジェクトでは、「いいえ」を選択していますが、プロジェクトを新規に作成した場合のデフォルトは「はい」になっています。）

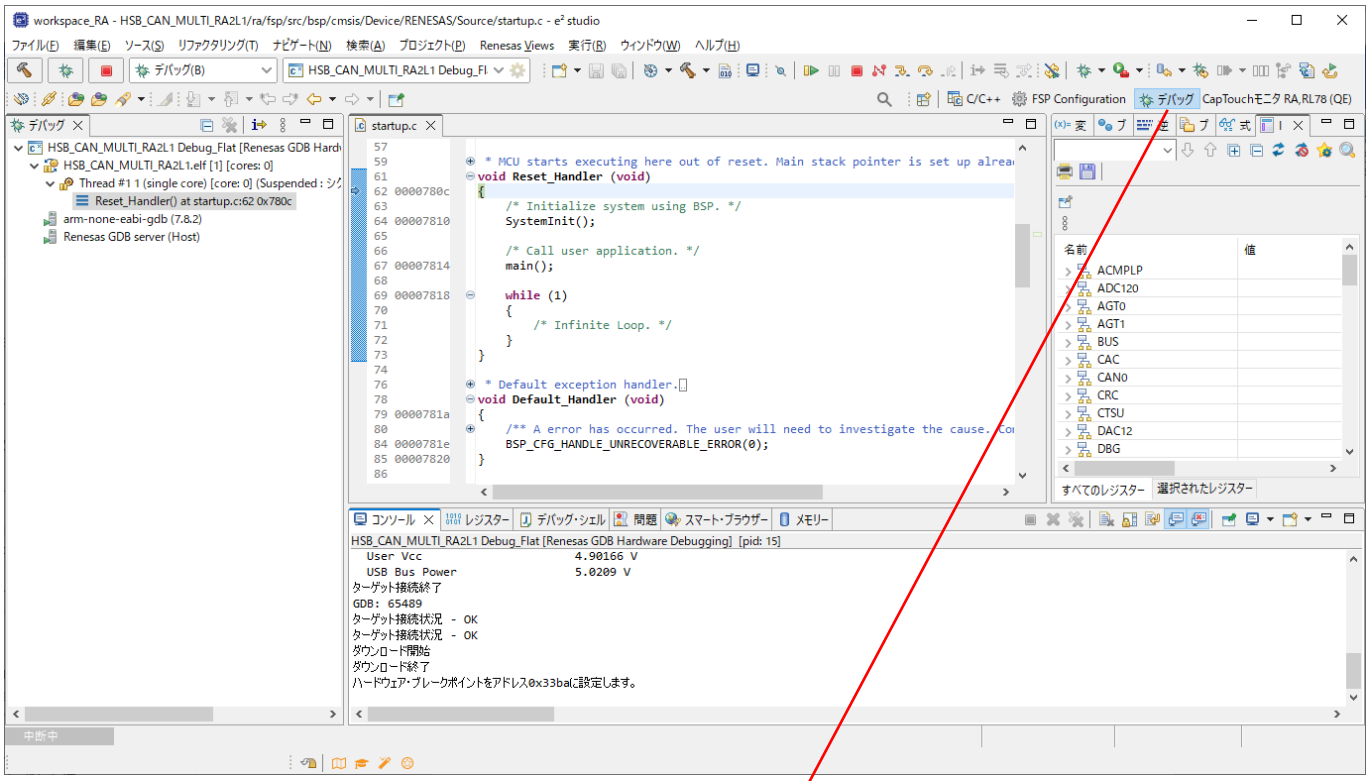
※設定を変更した場合は、「適用」「閉じる」



虫のアイコン（デバッグ）を押す。



上記画面が出た場合は、「切り替えを」押す。



CS+同様、

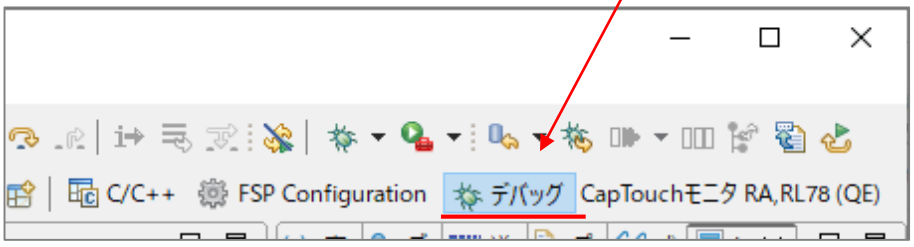
・変数のモニタ

・レジスタ値のモニタ

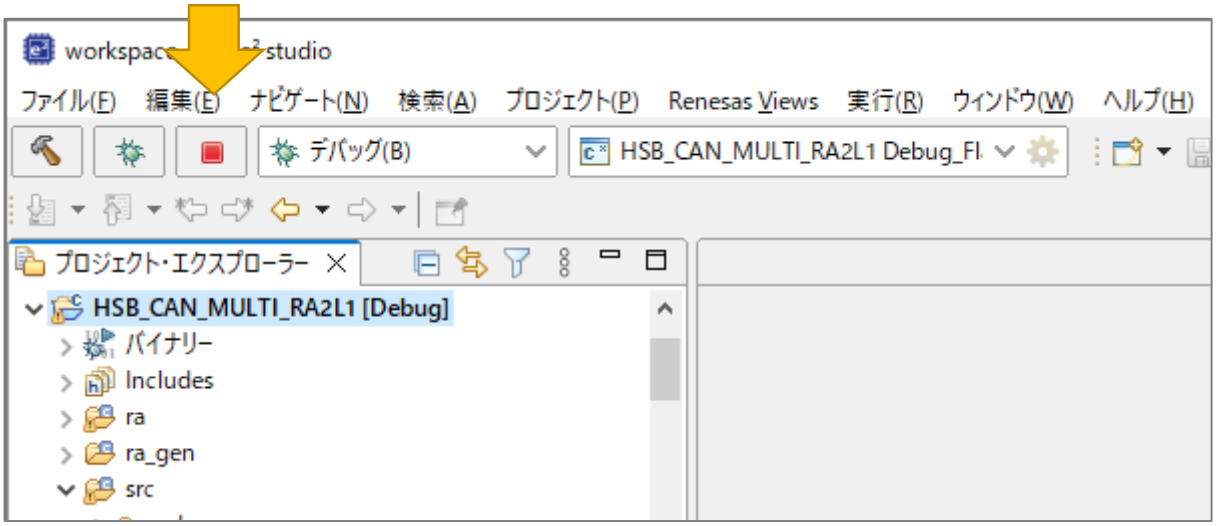
・メモリ値のモニタ

・ステップ実行

等が行えます。

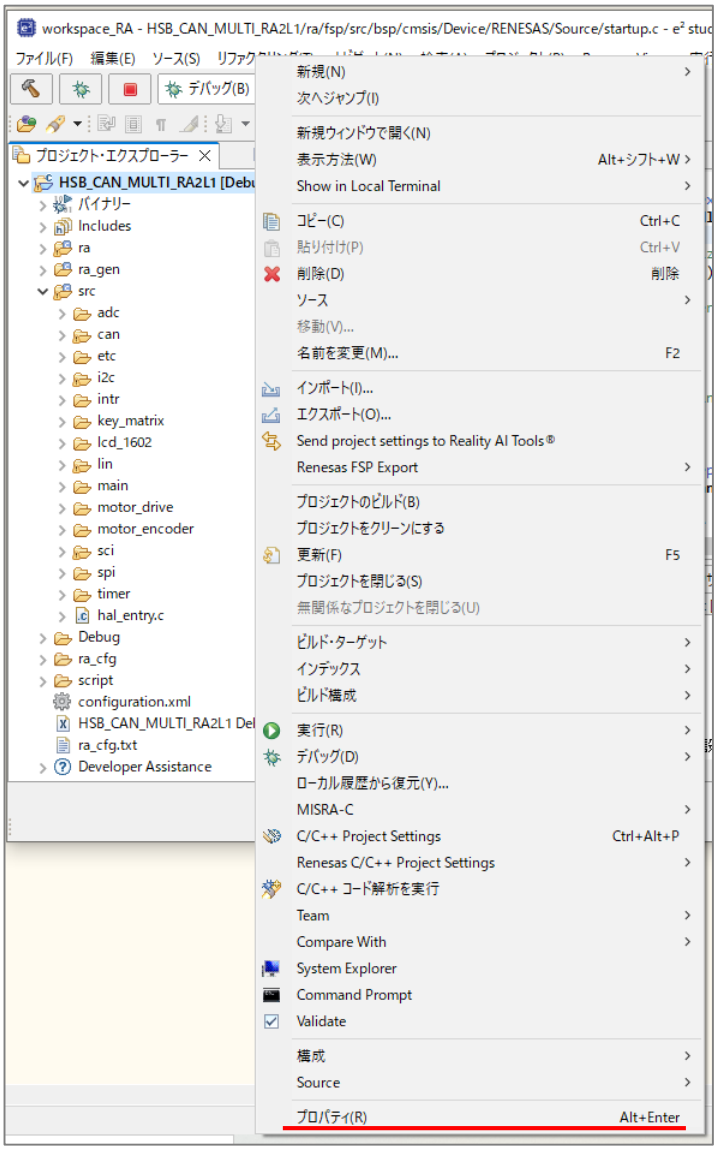


デバッグ時は、表示領域の設定を「デバッグ」を選択してください。
「C/C++（ソースコードの編集）」等を選択すると、デバッグに必要なウィンドウが表示されません

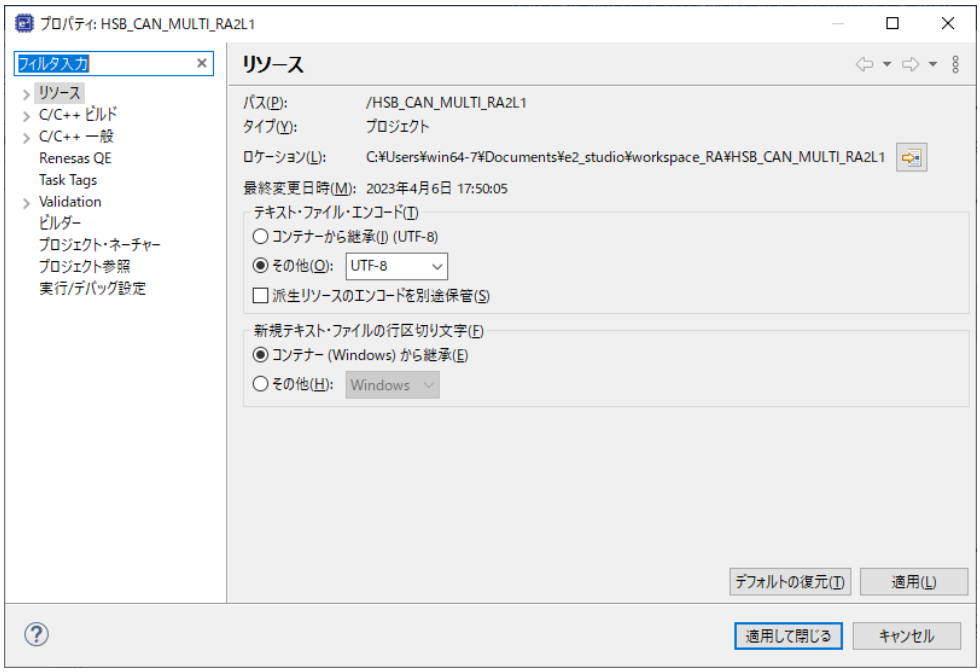


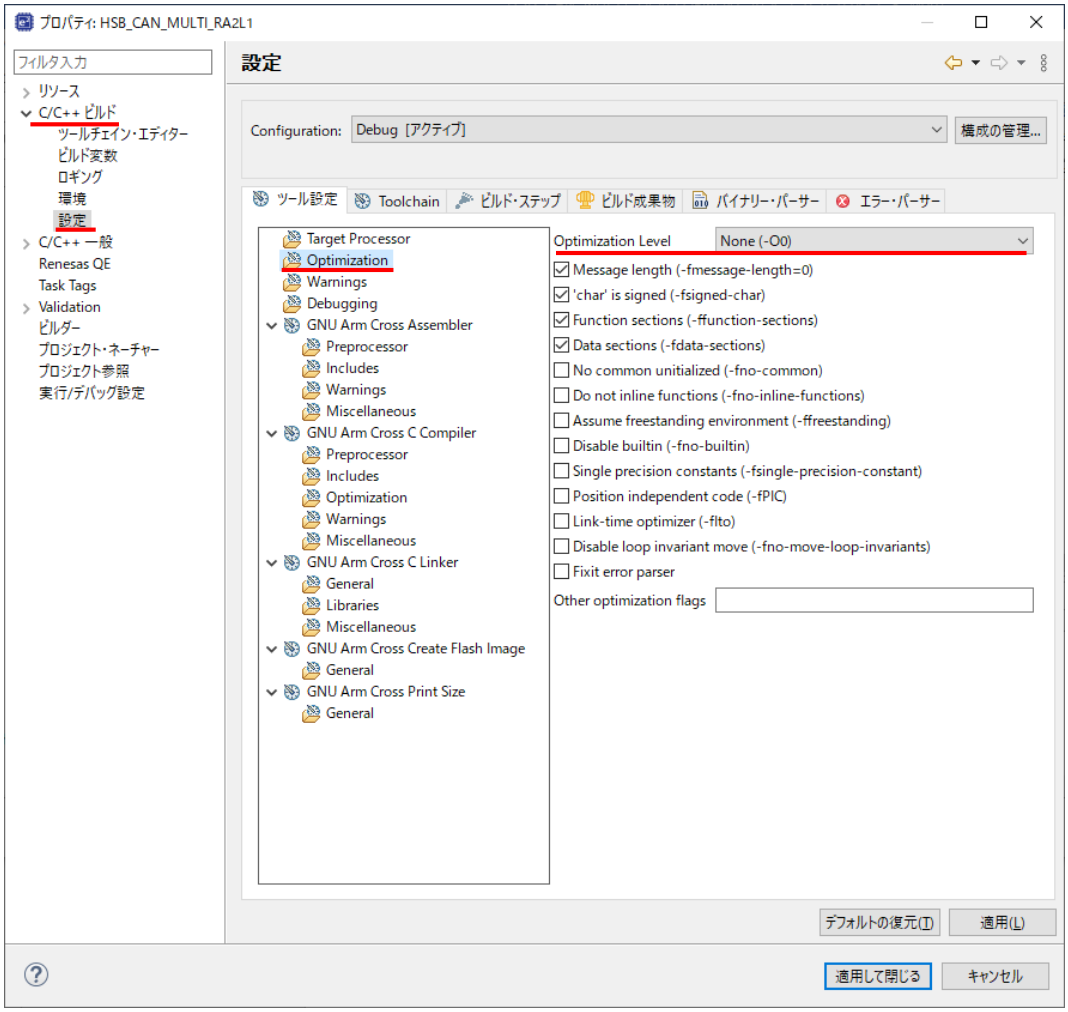
デバッグの終了時は、「停止」ボタンを押してください。

※エミュレータを外したり、ボードの電源を切る前に「停止」を押して切り離しを行ってください



プロジェクトを右クリックープロパティ





C/C++ビルドー設定ーOptimizationーOptimization Level

で、最適化の設定が行えます。デバッグ向けなのは「None(-O0)」です（デフォルトは-O2）。

デバッグ時は、必要に応じて最適化を OFF にしてください。

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.1.0.0.0	2023.6.6	—	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

ルネサス エレクトロニクス社　RX231, RL78/F15, RA2L1 搭載
HSB シリーズ応用キット

CAN マルチネットワークボード取扱説明書
開発環境編

株式会社 北斗電子
 2023 年 6 月 6 日改訂 REV.1.0.0.0 (230606)