IAR Embedded Workbench for Arm を使用した
ルネサスエレクトロニクス
RA6M3 スタートアップガイド

目次

1. はじめに 4
   1.1. 本ドキュメントに記載の情報について 4
2. 使用機材 4
   2.1. 評価ボード 4
   2.2. ルネサスエレクトロニクス製コード生成ツール 4
   2.3. IAR システムズ 統合開発環境 4
3. 本スタートアップガイドの目的 5
4. EK-RA6M3 評価キットの準備 6
   4.1. ジャンパ設定の確認 6
   4.2. I-jet をホストコンピュータに接続 6
5. RA Smart Configurator のインストール 8
   5.1. セットアップの実行 8
6. EWARM と RA Smart Configurator の連携 12
   6.1. EWARM ツールオプションへ登録 12
7. EWARM と RA Smart Configurator を使用したワークフロー 14
   7.1. ワークフロー概要 14
8. RA Smart Configurator を使って新規プロジェクトを作成 15
   8.1. EWARM で新規ブランクプロジェクトを作成 15
   8.2. RA Smart Configurator を起動 16
   8.3. RA Smart Configurator でプロジェクト構成を設定 18
   8.4. RA Smart Configurator でファイルを生成 24
   8.5. EWARM のプロジェクトコネクションで、RA Smart Configurator 生成ファイルを取り込み 25
   8.6. ワークスペースファイルを保存する 29
   8.7. EWARM のプロジェクトオプションを変更 30
   8.8. ユーザコードを記述 34
   8.9. ビルド→デバッグ 34
9. 評価ボード上の LED を制御 38
   9.1. LED 制御コード 38
   9.2. PFS レジスタの制御 39
   9.3. LED 点滅コードの実行 40
   9.4. ピン設定の初期化 40
10. タイマー割り込みで LED を点滅する 41
   10.1. RA Smart Configurator で Timer スタックを追加 42
   10.2. Timer スタックのパラメータ設定 43
   10.3. 割り込み割り当ての確認 44

10.4. コードの再生成 45
10.5. タイマー割り込みコールバック関数の実装 46
10.6. タイマー開始コードの記述 47
10.7. タイマー割り込みコールバック関数のデバッグ 49
11. おわりに 49

1. はじめに

ルネサスエレクトニクス製 RA ファミリは Arm Cortex-M 搭載の 32 ビット MCU です。優れたパフォーマンス、低消費電力動作、組込みセキュリティを兼ね備えています。
本ドキュメントでは IAR Embedded Workbench for Arm （以下:EWARM）と、高処理能力かつ各種通信インタフェースをサポートする RA6M3 搭載した評価ボードを使用する場合の開発手順を step by step で説明します。
RA ファミリの評価環境は互換性が高く、他の RA ファミリ評価ボードでも同様の手順で評価を行うことができます。

1.1. 本ドキュメントに記載の情報について

各種製品バージョンおよび Web ページの情報は、2020 年 6 月時点の情報に基づいています。バージョンアップや URL に変更がある場合には、適宜読み替えてください。

2. 使用機材

本ドキュメントで使用する、評価ボートと開発環境の説明を行います。

2.1. 評価ボード

EK-RA6M3 (RA6M3 MCU グループ評価キット: http://renesas.com/ra/ek-ra6m3)

- R7FA6M3AH3CFC MCU （120MHz, Arm Cortex®-M4, 2MB フラッシュメモリ、640KB SRAM）
- I-jet（または I-jet Trace）デバッグプローブ用 JTAG コネクタ
- USB コネクタから 5V 給電

2.2. ルネサスエレクトロニクス製コード生成ツール

RA Smart Configurator 1.x.x を使用します。
RA Smart Configurator は初期化コード自動生成ツールです。同梱されている Flexible Software Package（FSP）を使用して、ユーザが GUI で選択したコンポーネントに必要なコードを生成します。また、本ドキュメントでは使用しませんが、RTOS（FreeRTOS）を使用したプロジェクトも設定することもできます。
RA Smart Configurator は EWARM と簡単に連携をすることができます。開発の途中でも再度コンフィグレーション・コード生成のやり直し、といったワークフローが実現でき開発工数、コストの削減に貢献し、開発を簡略化します。

2.3. IAR システムズ 統合開発環境

IAR Embedded Workbench for Arm Ver8.50.x
RA ファミリ MCU はバージョン 8.50.1 以降でサポートされています。
製品ライセンスユーザの方はマイページより最新バージョンをインストールしてください。
評価版は IAR システムズ Web ページよりダウンロードすることができます。
https://www.iar.com/jp/iar-embedded-workbench/#!?architecture=Arm

インストーラをダウンロード後、ウィザードの指示に従ってインストール、ライセンスの取得・有効化を実施してください。

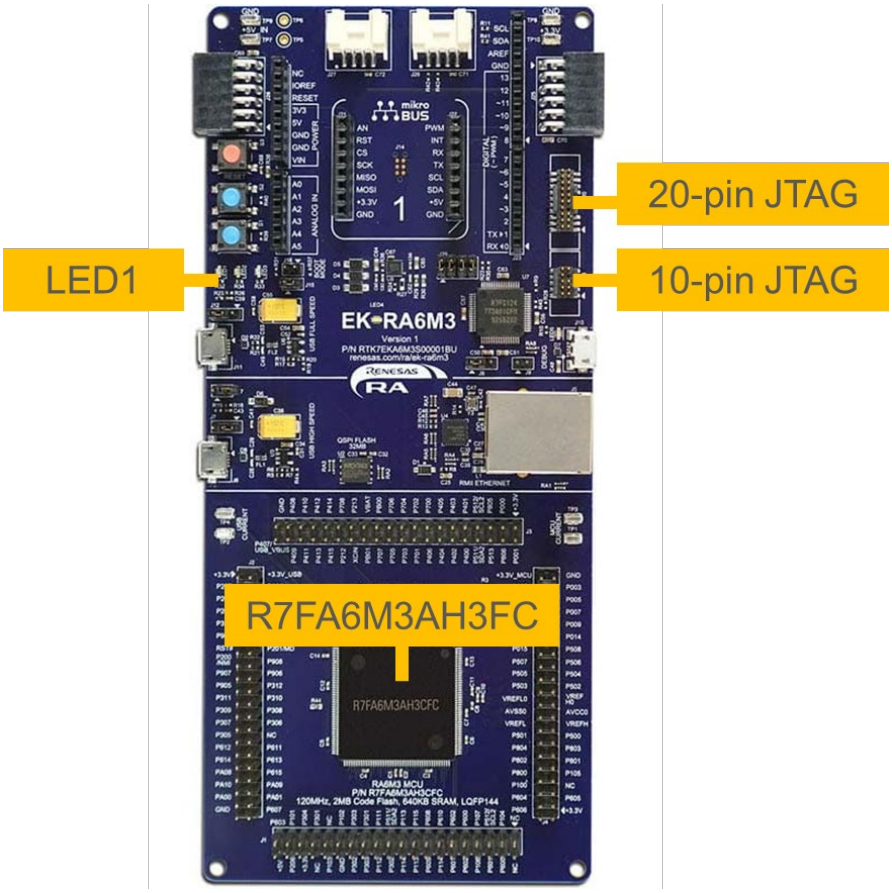
3. 本スタートアップガイドの目的

RA Smart Configurator を使用してコード自動生成を行い、EK-RA6M3 と EWARM を使用して動作確認を行います。

EK-RA6M3 にはユーザ LED が実装されているので、該当する GPIO 端子とタイマー割り込みを使用して周期的に LED 点滅を行うプログラムを作成します。

4. EK-RA6M3 評価キットの準備

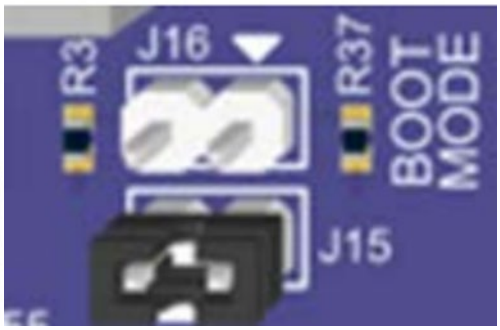
EK-RA6M3 評価キットには、評価ボード本体とマイクロ USB ケーブルが同梱されています。



ボード左側にはユーザ LED が実装されています。

4.1. ジャンパ設定の確認

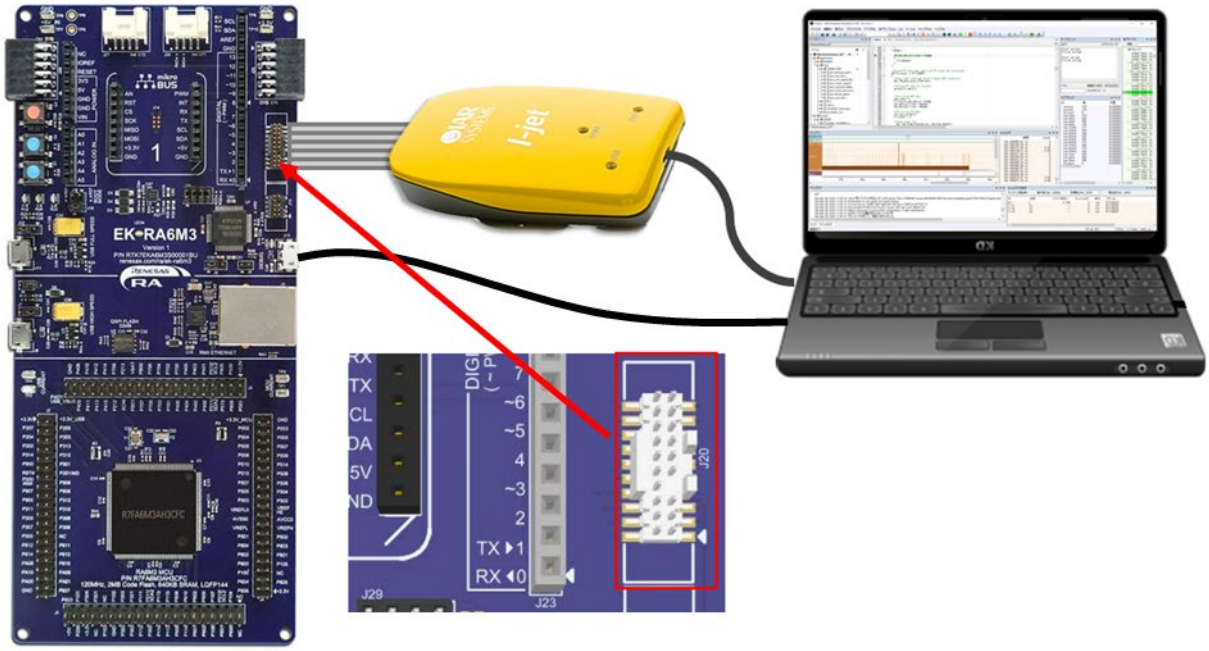
BOOT CONFIG が INTERNAL FLASH の向きでクローズとなっていることを確認してください。



4.2. I-jet をホストコンピュータに接続

EK-RA6M3 へは **J10** マイクロ USB コネクタから電源を供給します。デバッグプローブは I-jet、I-jet Trace またはオンボードの J-Link OB が使用可能です。I-jet(または I-jet Trace)を使用する際には **J20** MIPI20 ヘッダに I-jet を接続し、I-jet の USB ポートをホストコンピュータの任意のポートに接続してください。**J20**

ヘッダでは JTAG、SWD および ETM（トレース）をサポートしています。I-jet の代わりに J-Link OB を使用する際には J-Link OB の制御に使用されるため、**J10** USB が接続されていることを確認してください。ボードの電源オプション詳細については EK-RA6M3 ユーザマニュアルの section 5.1.1 を参照してください。

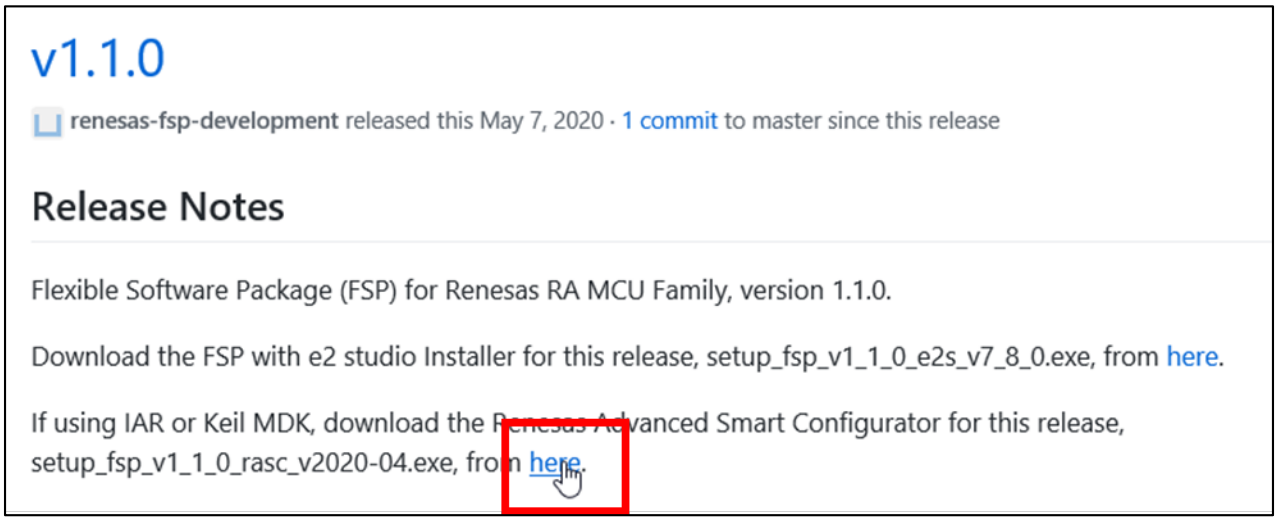


5. RA Smart Configurator のインストール

RA Smart Configurator は、ルネサスエレクトロニクス社の GitHub サイトからダウンロードします。

2020 年 6 月現在の最新版は、バージョン 1.1.0 になります。

https://github.com/renesas/fsp/releases

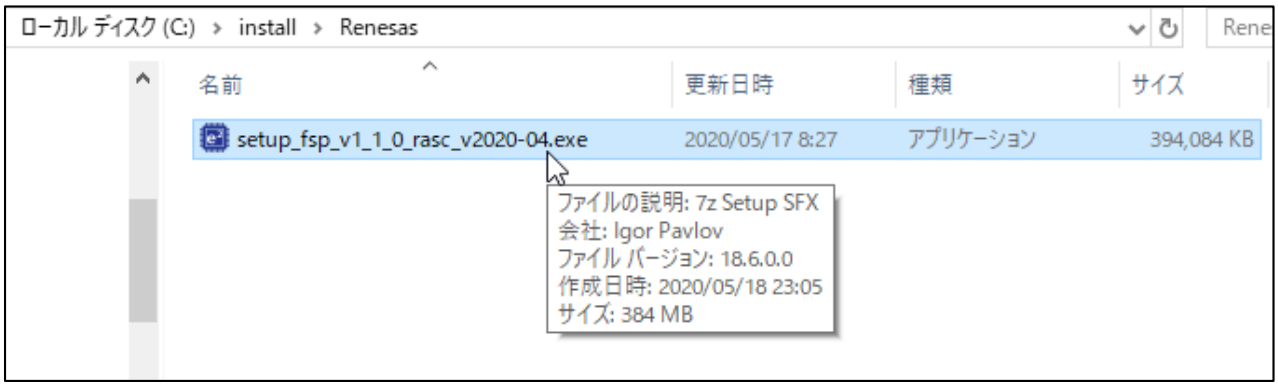


「setup_fsp_v1_1_0_rasc_v2020-04.exe」をダウンロードする事が出来ますので、任意のフォルダ（例：C:\Install\Renesas）に保存します。

*上の setup_fsp_v1_1_0_**e2s**_v7_8_0.exe と間違えないように注意してください。

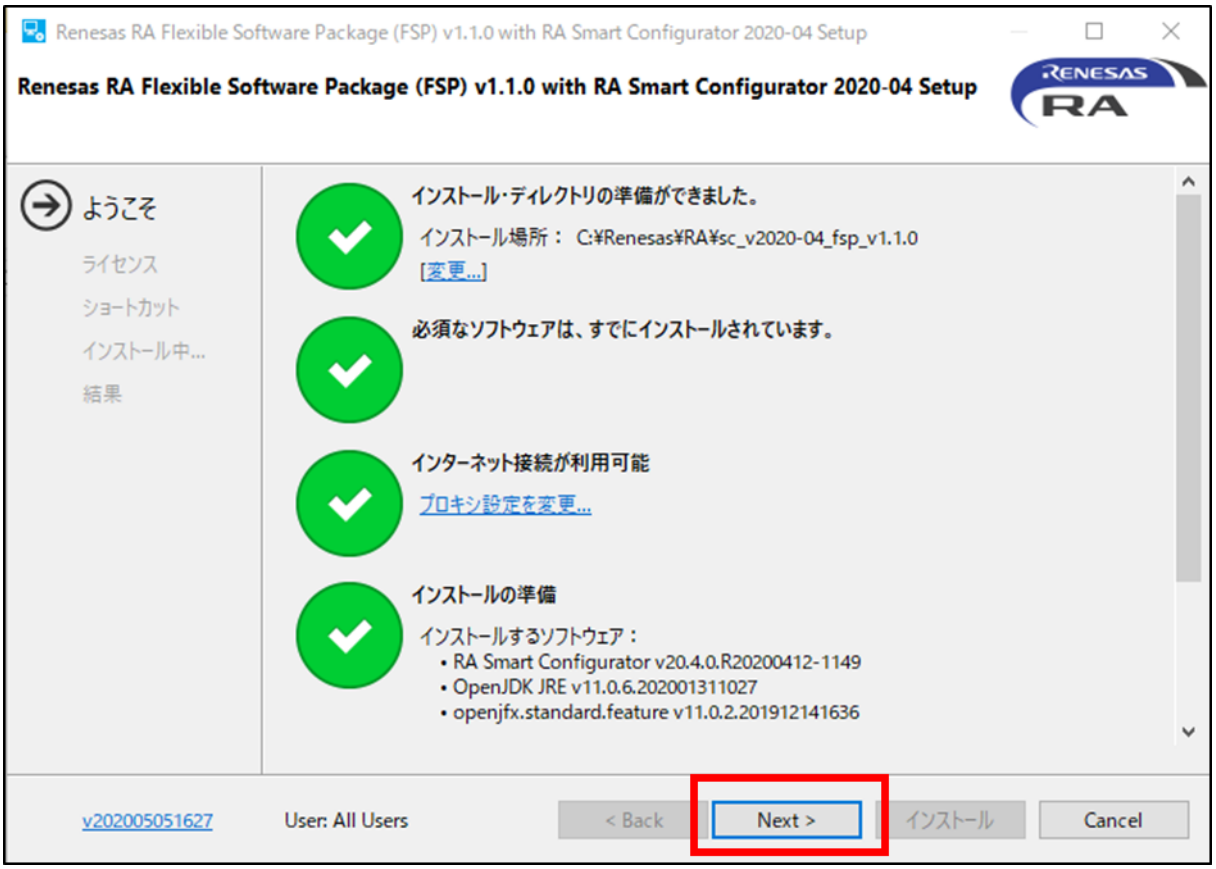
5.1. セットアップの実行

「setup_fsp_v1_1_0_rasc_v2020-04.exe」を実行します

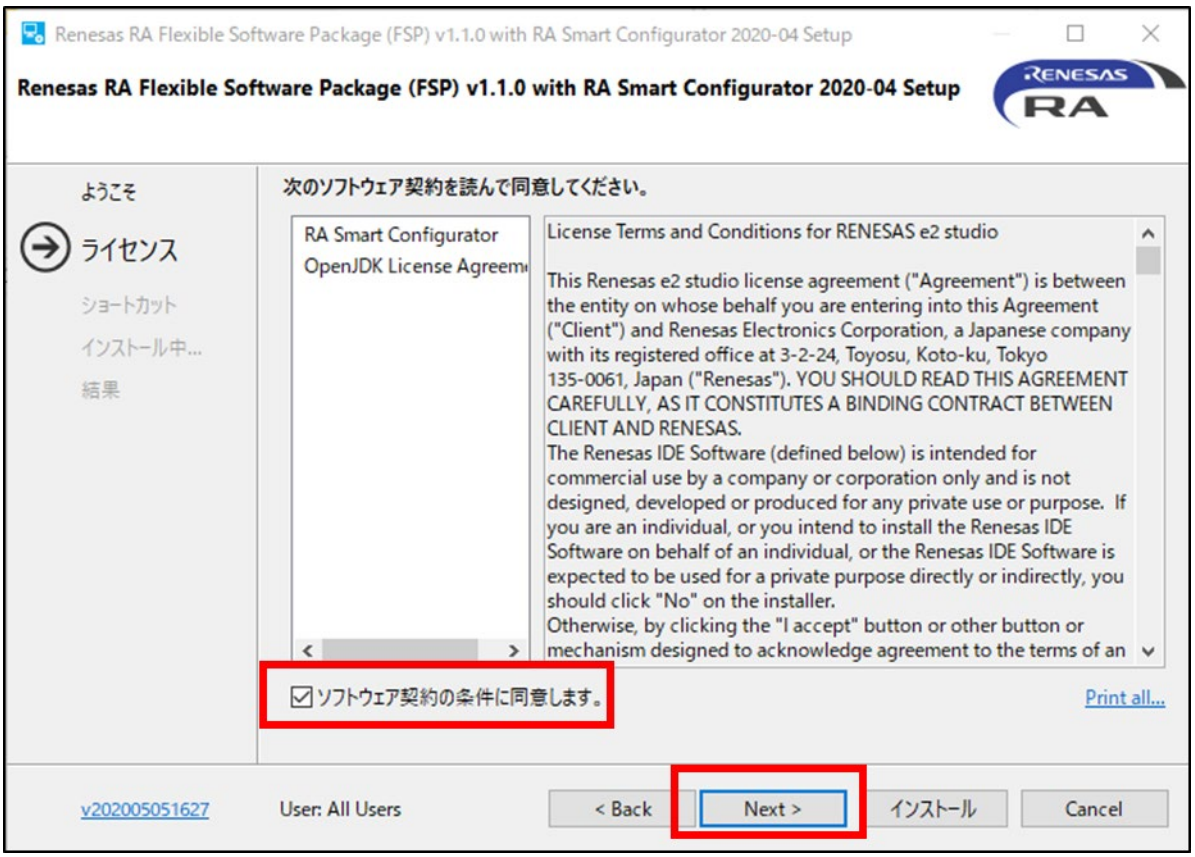


インストーラが起動するので、デフォルトでインストールを進めていきます。

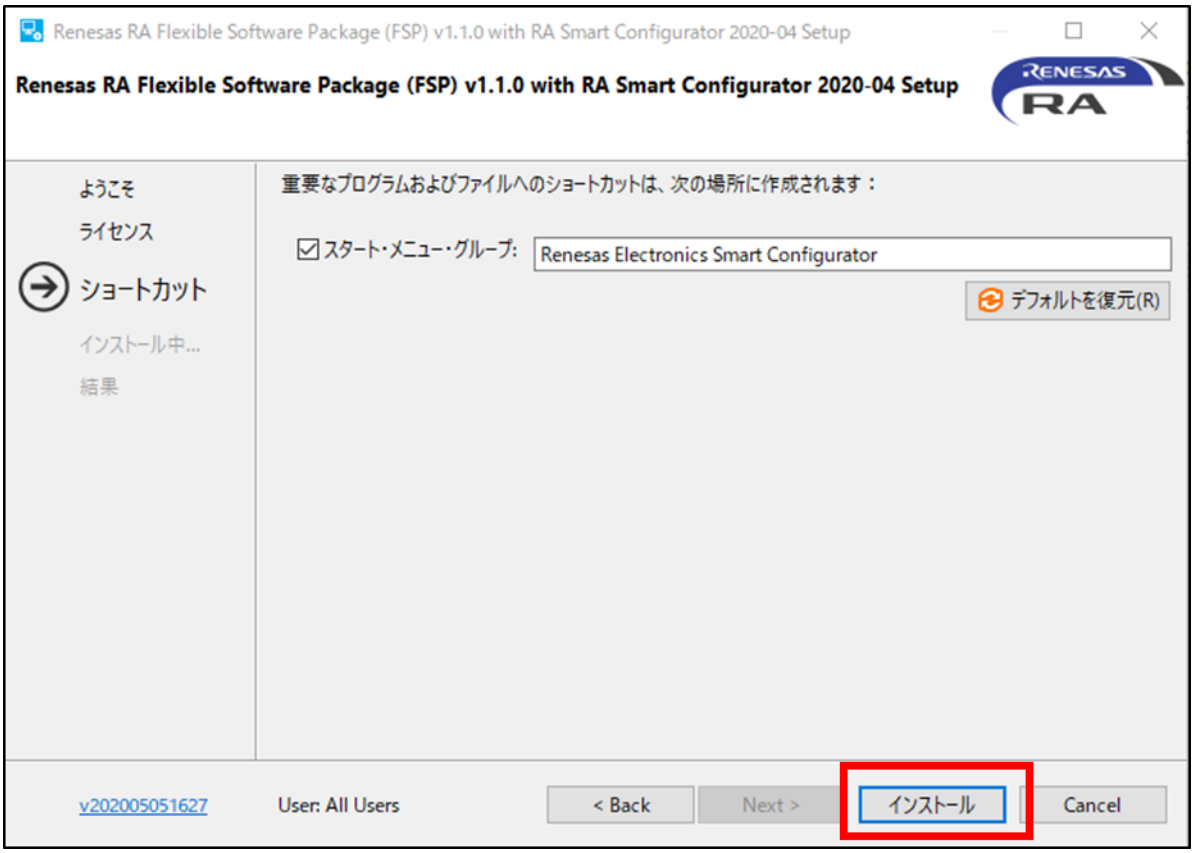
ホストコンピュータの状況によっては、Visual Studio のランタイムライブラリの追加インストールが要求されることがあります。



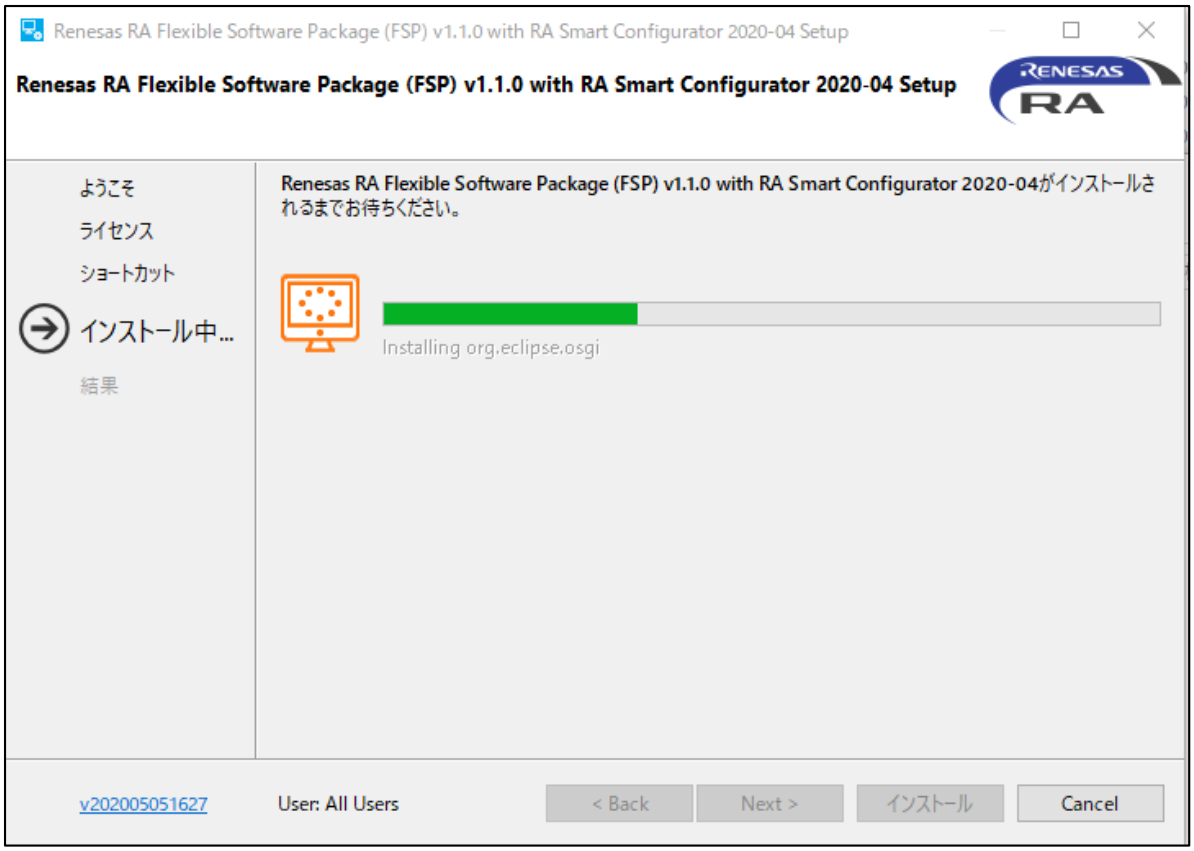
Next をクリックしてください。



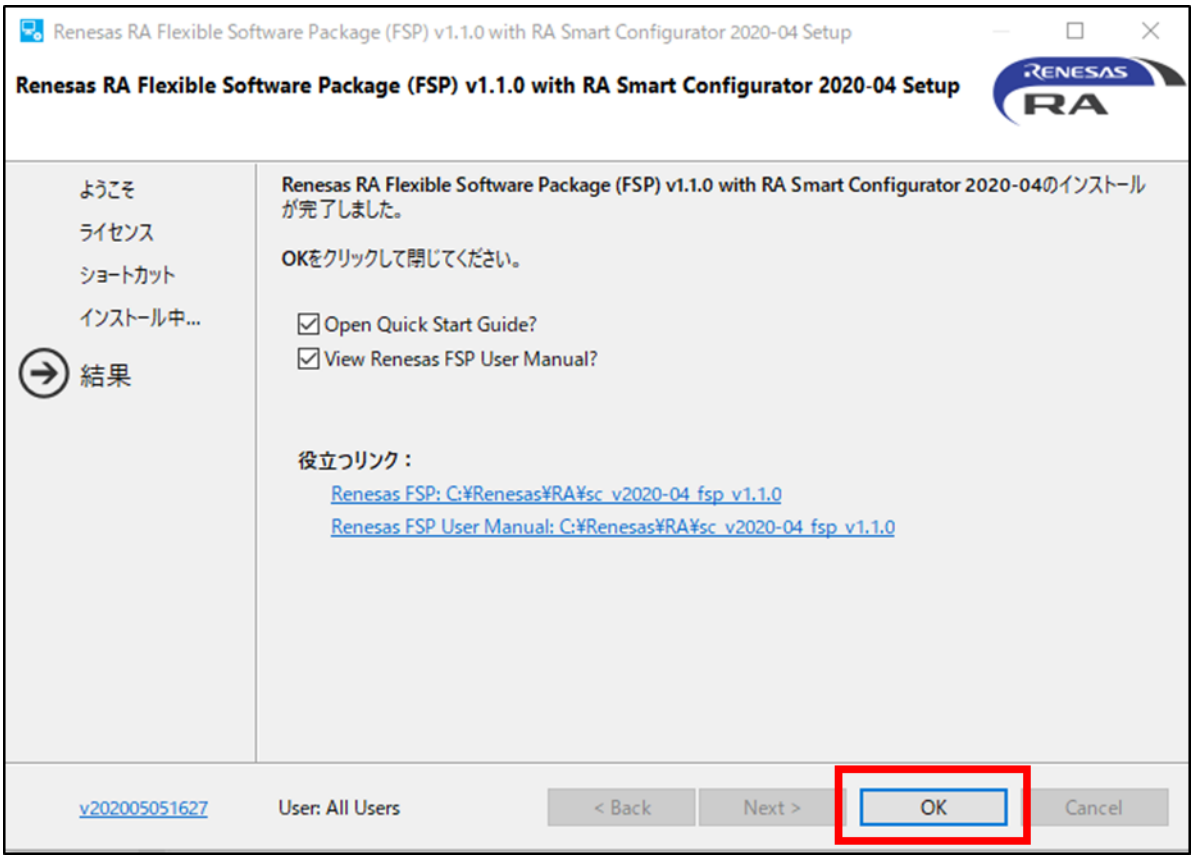
ソフトウェア契約の条件を読んで同意にチェック後 **Next** をクリックしてください。



インストール をクリックしてください。



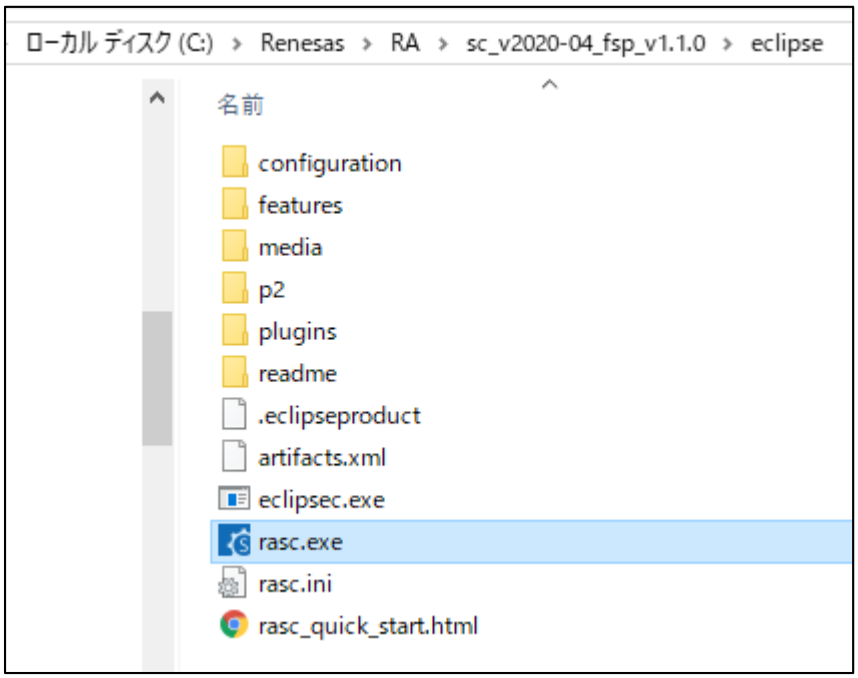
インストールが終了するのを待ちます。



OK をクリックして終了してください。

追加ソフトウェアのインストールが要求されたら、同様にデフォルトの設定でインストールを実施してください。

正常に完了すると、RA Smart Configurator（rasc.exe）が C:\Renesas\RA\sc_v2020-04_fsp_v1.1.0\eclipse にインストールされます。



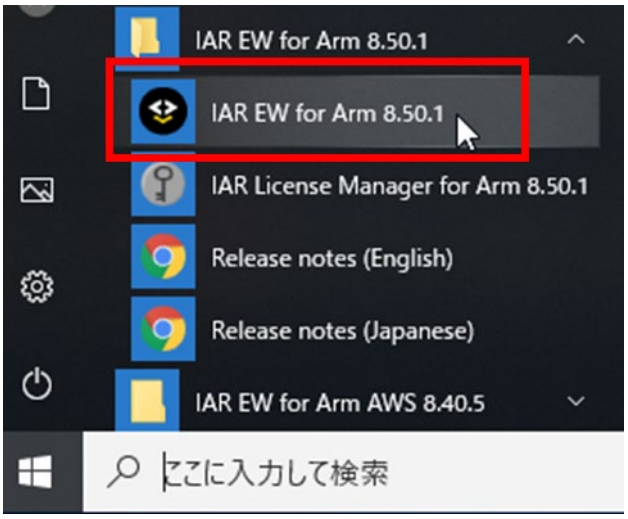
基本的に rasc.exe は直接起動することはありません。EWARM からのメニューから呼び出しを行います。

6. EWARM と RA Smart Configurator の連携

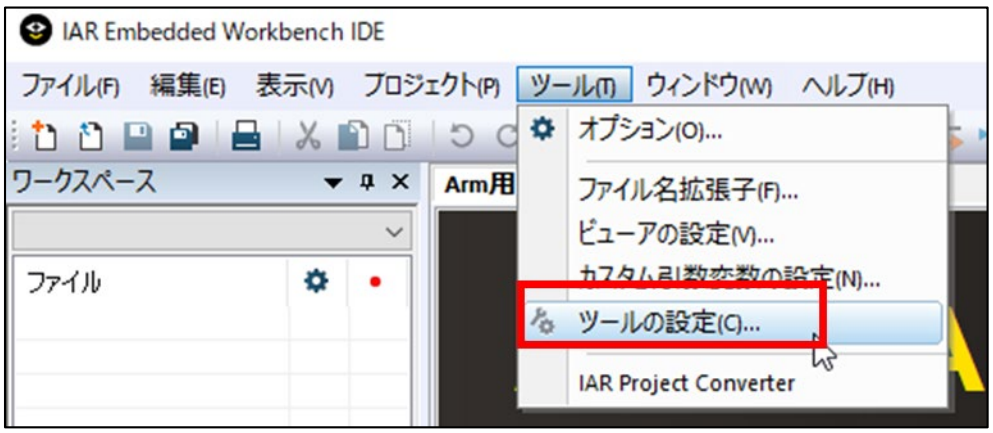
EWARM のメニューから RA Smart Configurator を起動し、プロジェクトと連携できるよう設定を行います。

6.1. EWARM ツールオプションへ登録

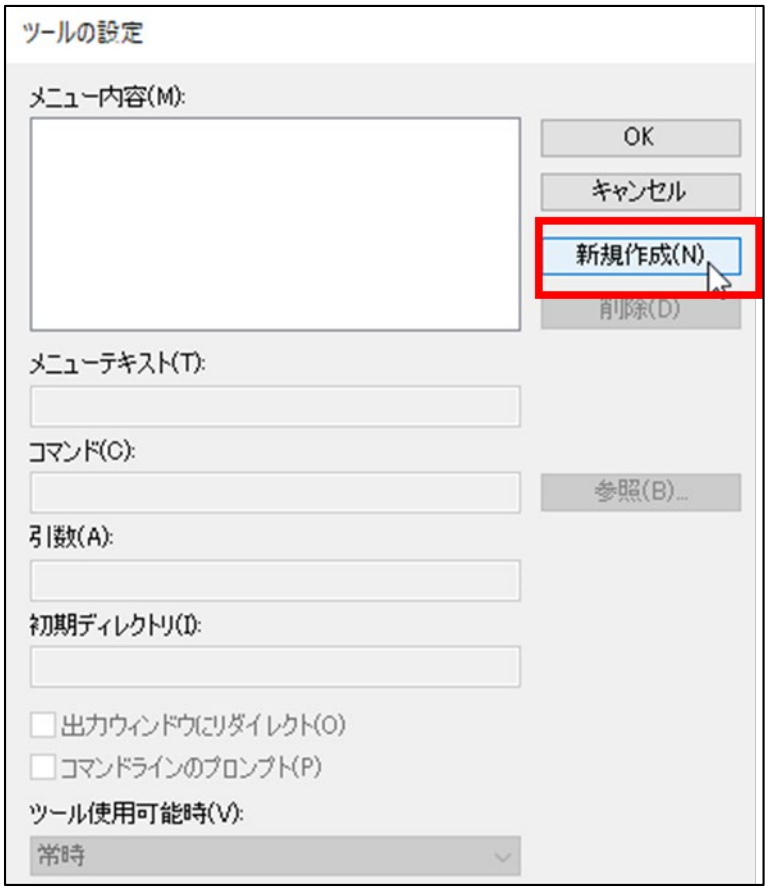
スタートメニューから IAR EW for Arm 8.xx を起動します。



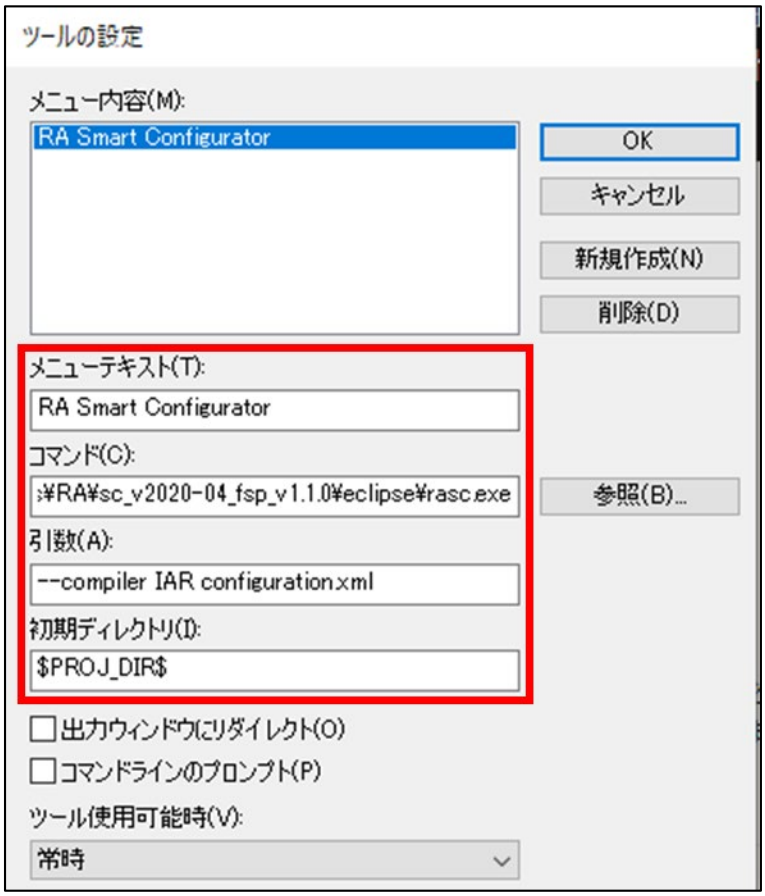
メニューバーから**ツール>ツールの設定**を選択します。



新規作成 をクリックします。



各項目にパラメータを指定します。



メニューテキスト：RA Smart Configurator　（任意）

コマンド：C:\Renesas\RA\sc_v2020-04_fsp_v1.1.0\eclipse\rasc.exe

※インストールバージョンが異なる場合は、インストールされた rasc.exe を指定してください。

引数：--compiler IAR configuration.xml

RA Smart Configurator に対して EWARM 用のコード生成することを指定します。

初期ディレクトリ：$PROJ_DIR$

EWARM のプロジェクトディレクトリとの連携を指定します。

ツール使用可能時 はデフォルトの **常時** のままにしてください。

設定が終わると**ツール** > **メニュー**から **RA Smart Configurator** が起動できるようになります。

7. EWARM と RA Smart Configurator を使用したワークフロー

EWARM と RA Smart Configurator を使用して開発を進める際のワークフローを説明します。

7.1. ワークフロー概要

- EWARM で新規ブランクプロジェクトを作成
- RA Smart Configurator を起動
- RA Smart Configurator でプロジェクト構成を設定
- RA Smart Configurator でファイルを生成
- EWARM のプロジェクトコネクションで、RA Smart Configurator 生成ファイルを取り込み
- EWARM のプロジェクトオプションを変更
- ユーザコードを記述
- ビルド→デバッグ

デバイス構成を変更する場合

- RA Smart Configurator を起動
- RA Smart Configurator でプロジェクト構成を変更
- RA Smart Configurator でファイルを生成

これを実施することで、EWARM のプロジェクト中のファイルが更新されます。

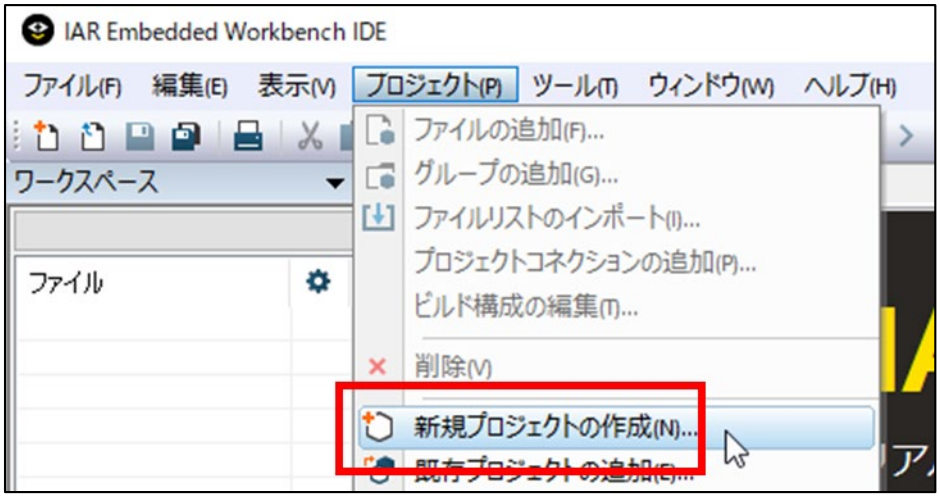
この手順を繰り返していくことで、アプリケーションを完成させていきます。

8. RA Smart Configurator を使って新規プロジェクトを作成

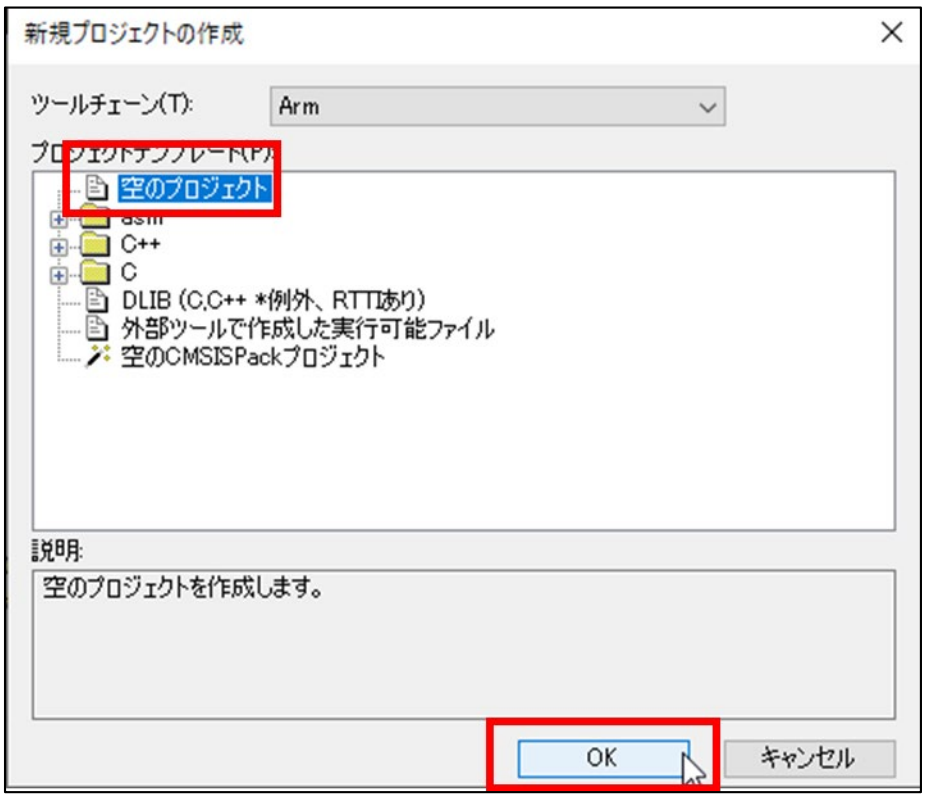
上記ワークフローに従ってプロジェクトを作成していきます。

8.1. EWARM で新規ブランクプロジェクトを作成

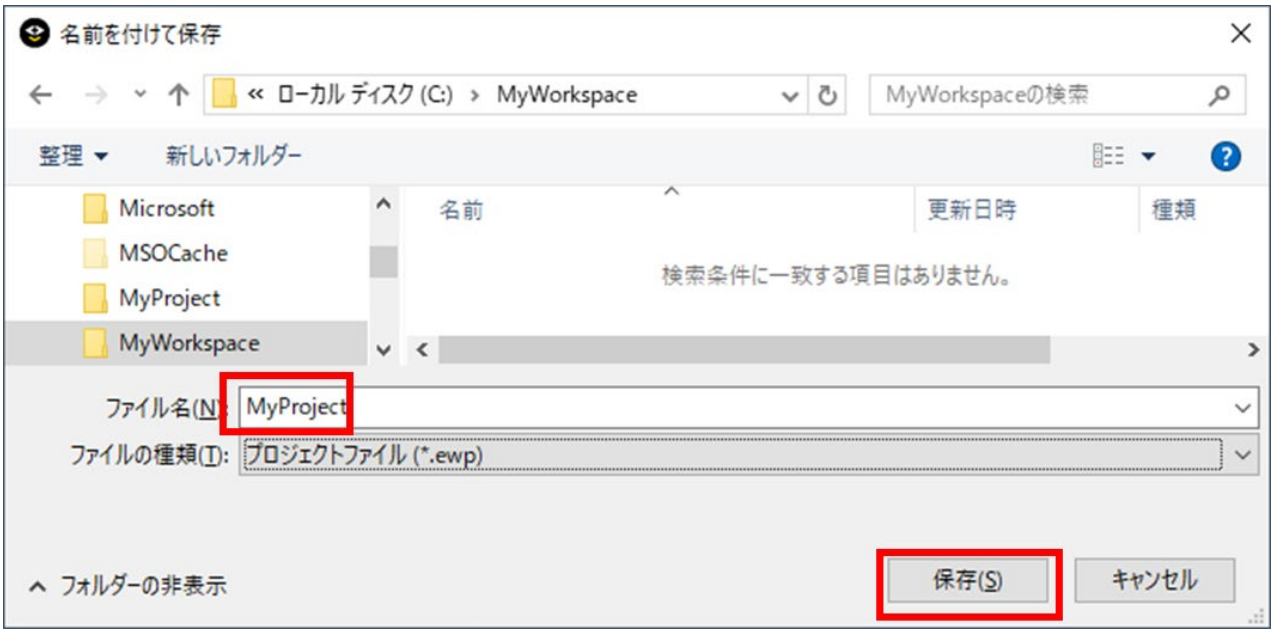
プロジェクトメニューから**新規プロジェクトの作成**を選択します。



空のプロジェクトを選択して **OK** をクリックします。



任意のフォルダにプロジェクトを保存します。ここでは C:\MyWorkspace フォルダに MyProject というプロジェクト名で保存することにします。フォルダ名、プロジェクト名を変更した場合には適宜読み替えてください。

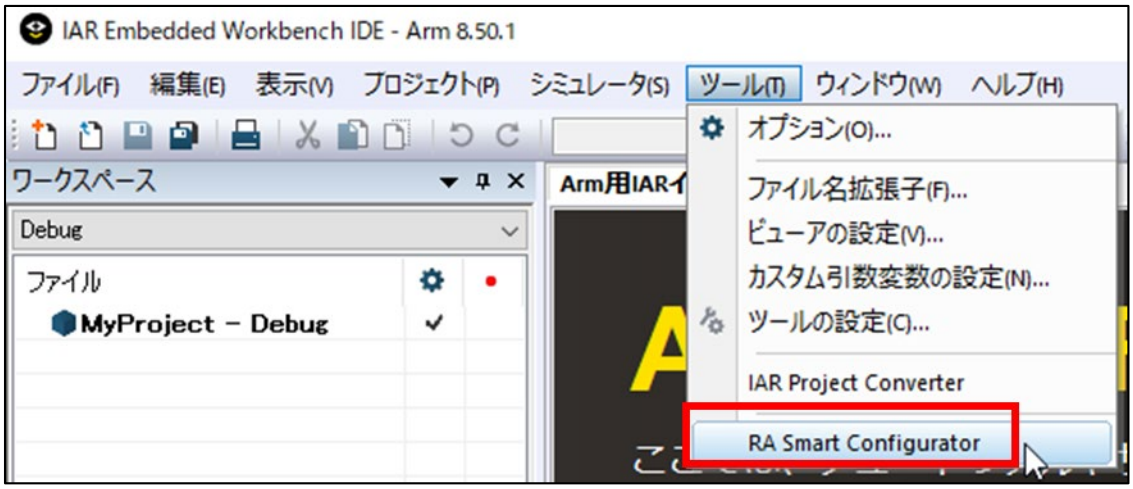


新規プロジェクトが作成されました。

Arm 用 IAR インフォメーションセンタ タブは閉じて構いません。

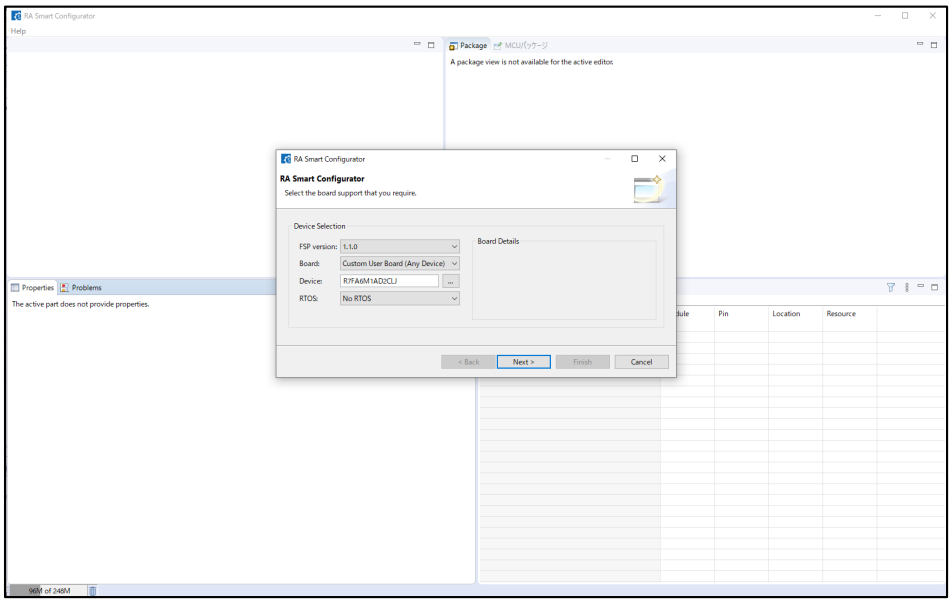
8.2. RA Smart Configurator を起動

メニューバーの **ツール**から先ほど登録をした **RA Smart Configurator** を起動します。



スタートアップ画面が表示され、

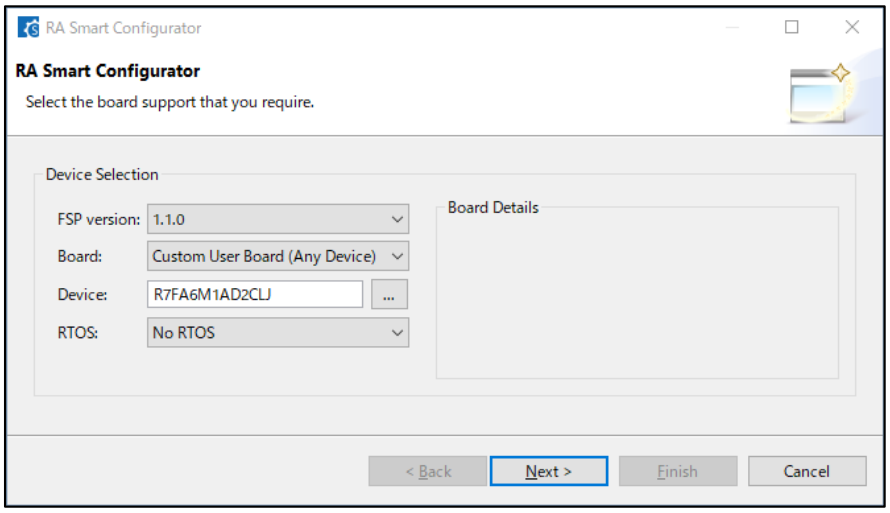
初期ウィザードが表示されます。



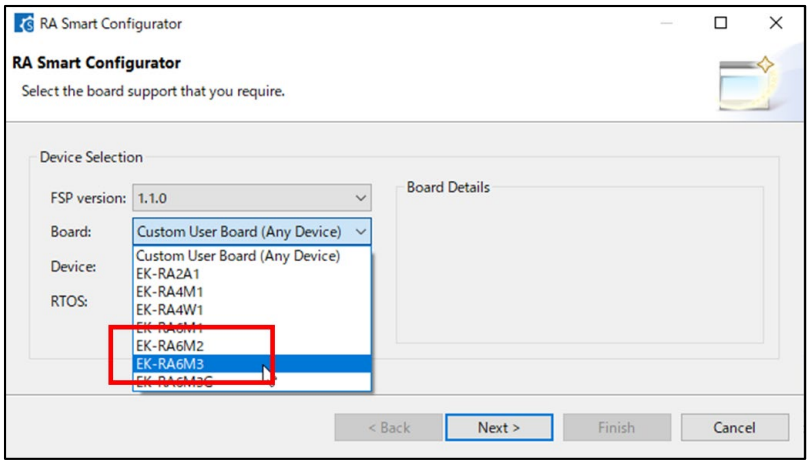
起動しない場合には、**ツール** > **ツールの設定** パラメータを確認してください。

8.3. RA Smart Configurator でプロジェクト構成を設定

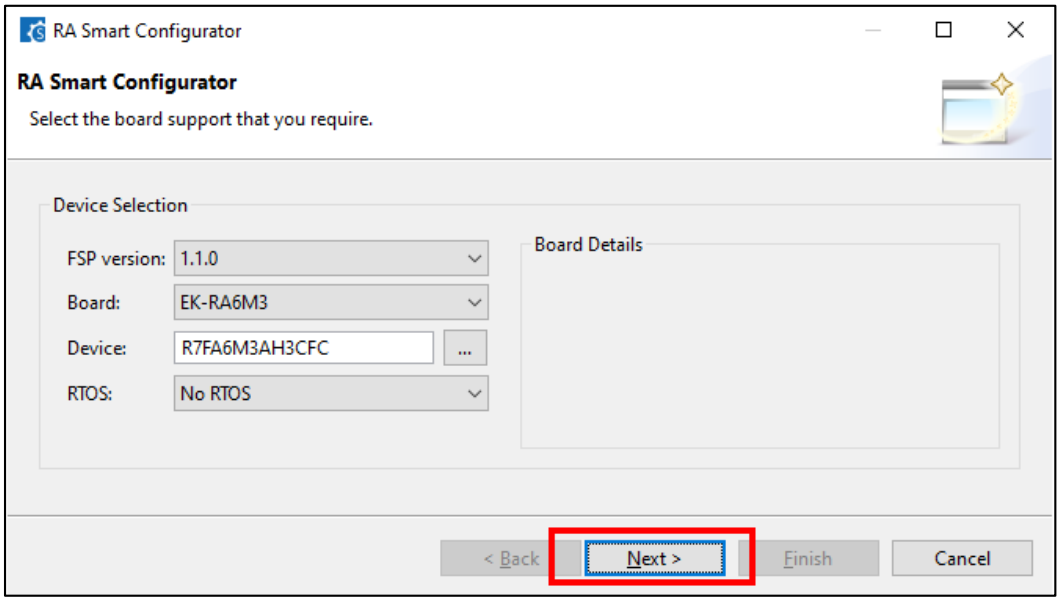
最初に、使用するボードを評価ボードか、あるはカスタムボードか選択します。評価ボードを選択すると、自動的に Device も選択されます。



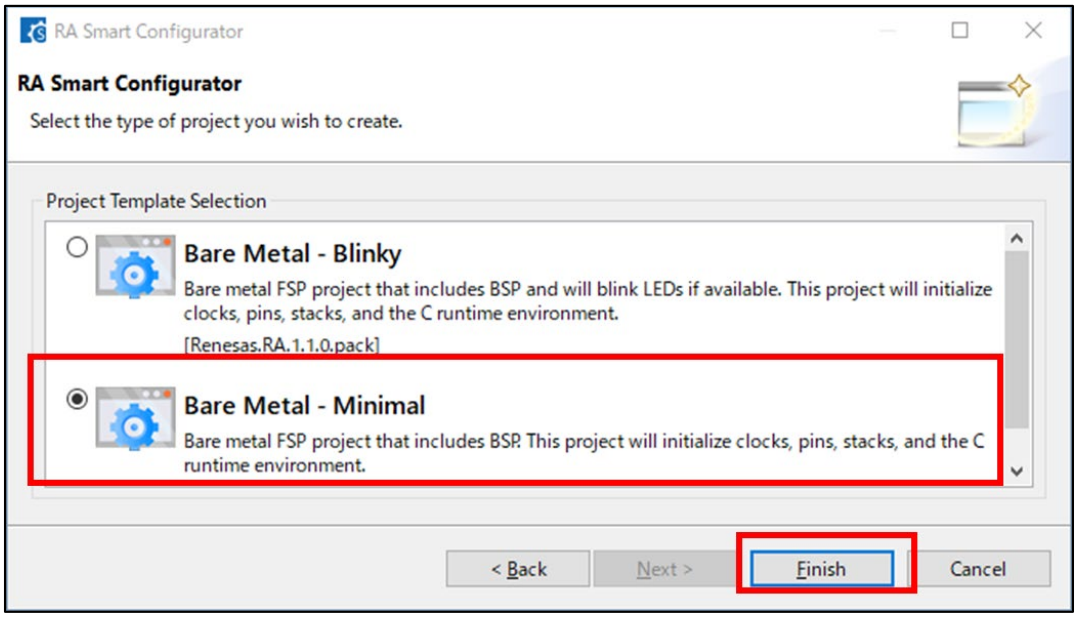
Board に使用する評価ボード **EK-RA6M3** を指定します。



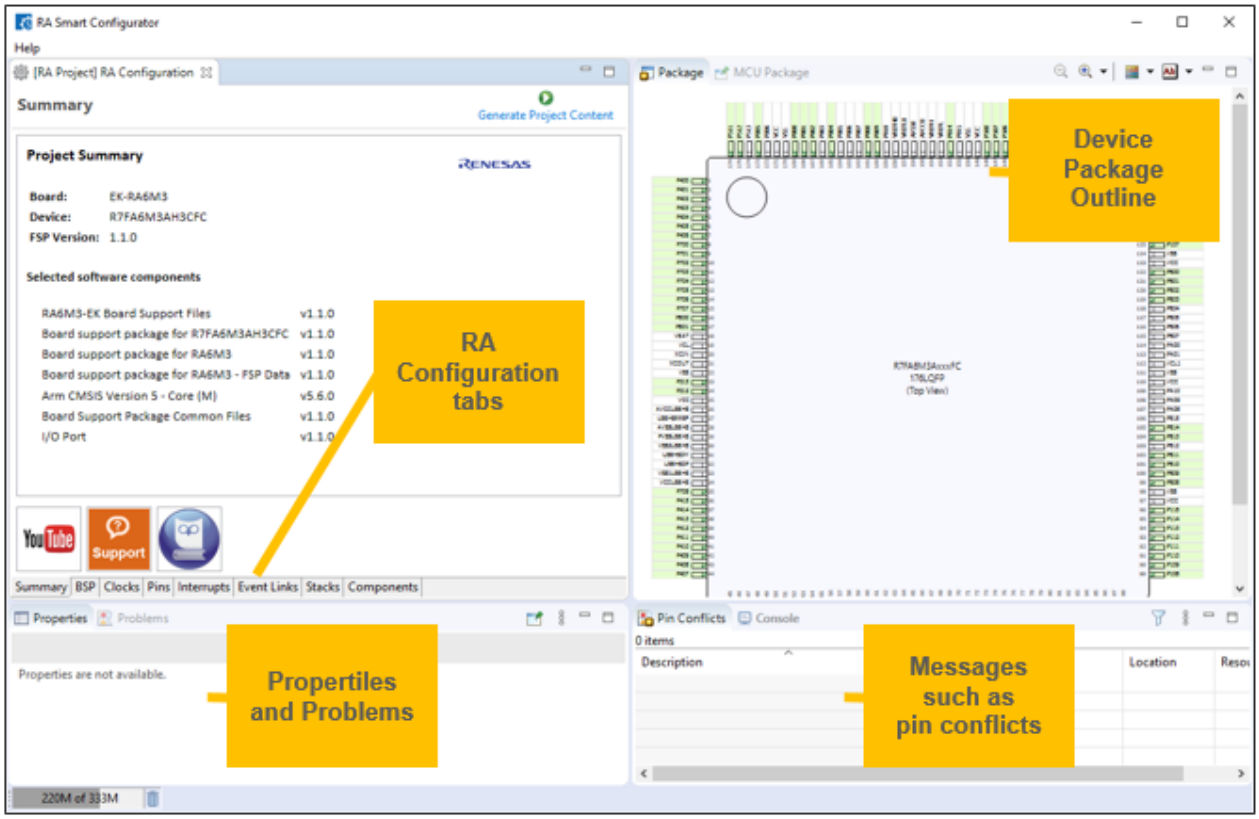
Device も R7FA6M1AD3CFP に切り替わります。本ドキュメントでは RTOS は使用しないので **No RTOS** のままにして **Next** をクリックします。



本ドキュメントでは、手動で LED 点滅プロジェクトを実装していきます。プロジェクトテンプレートとして、**Bare Metal - Minimal** 構成を選択して **Finish** をクリックしてください。



RA Smart Configurator のメイン画面が表示されます。

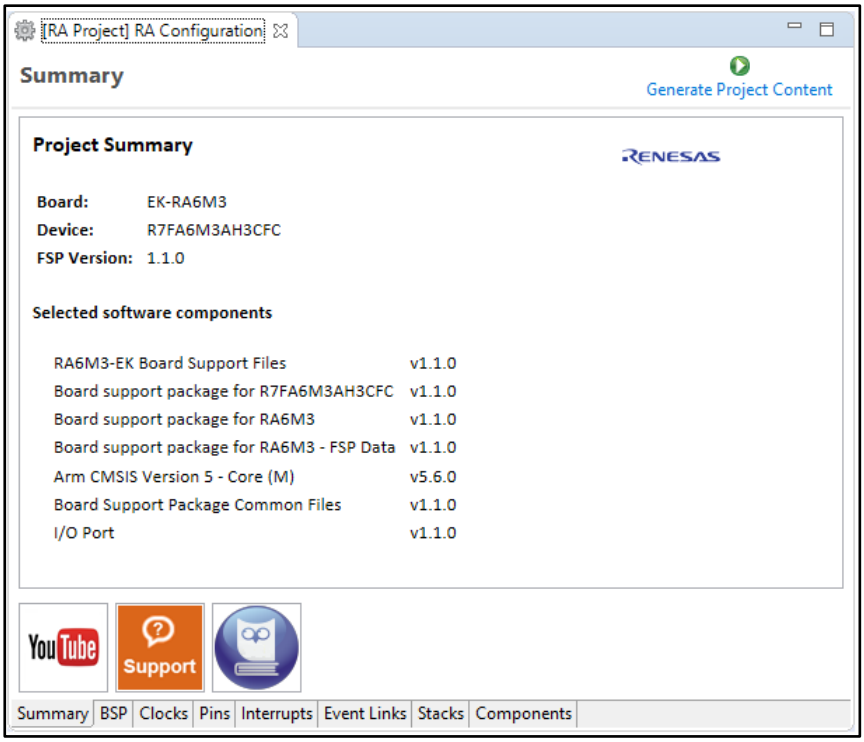


基本的な画面構成は以下の通りです。

- 左上：各種の情報や設定
- 右上：使用するデバイスのピン割り当て
- 左下：選択項目の詳細（プロパティ）
- 右下：構成の不整合などのエラー情報

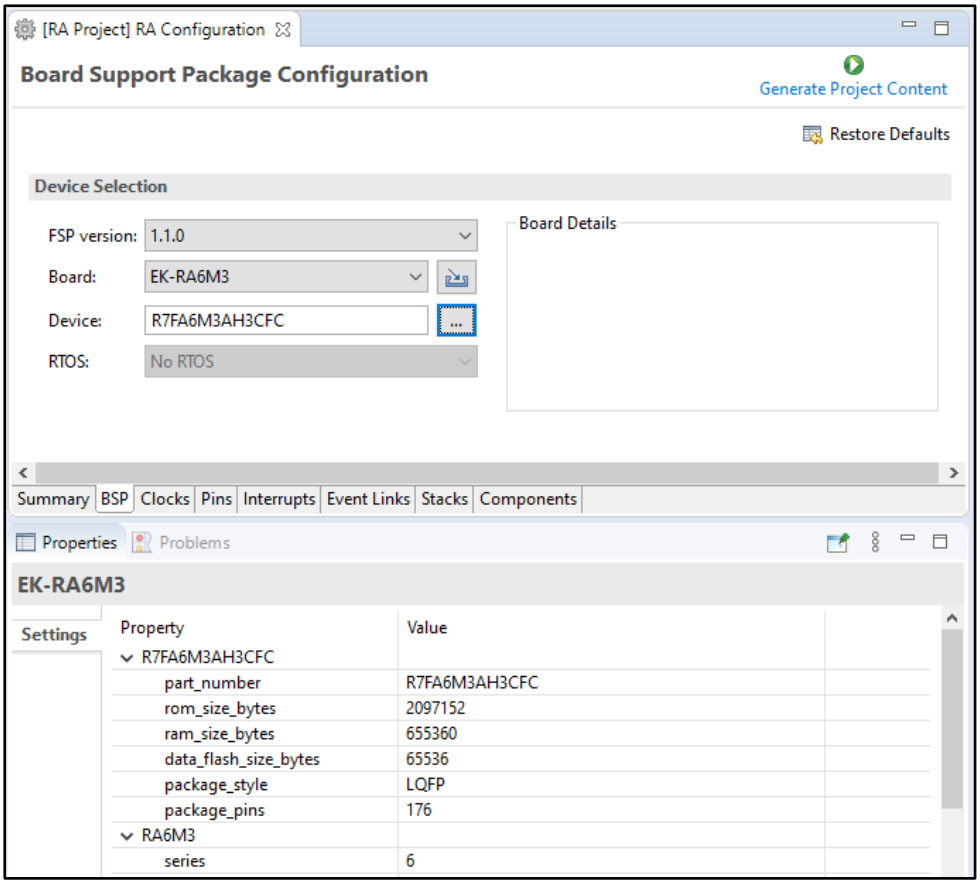
Summary

ボード情報や FSP のバージョンなど今回のプロジェクトの概要を表示しています。



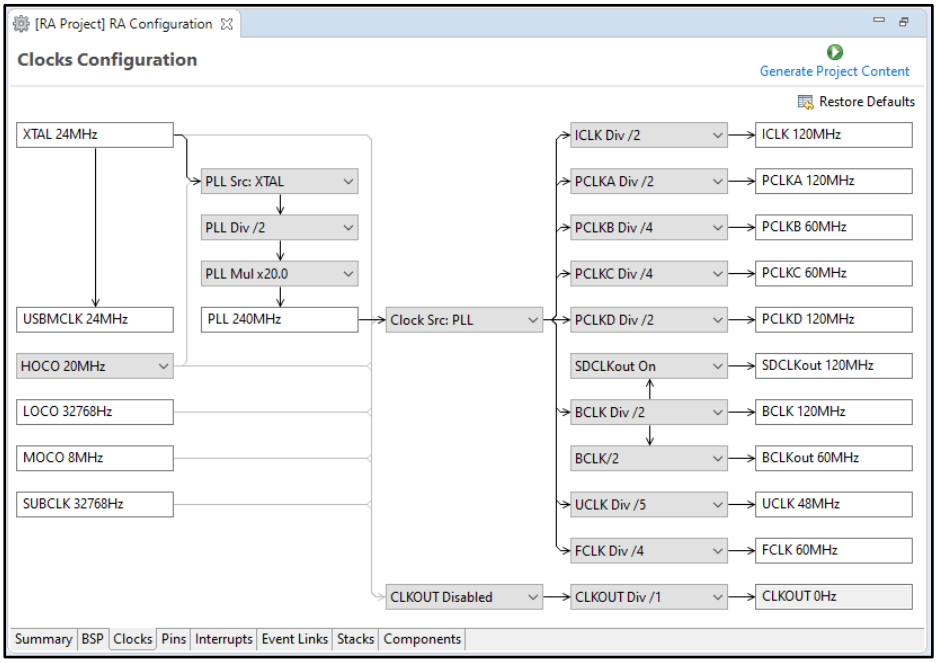
BSP

先ほど選んだボードの情報を Property ウィンドウで確認することができます。



Clocks

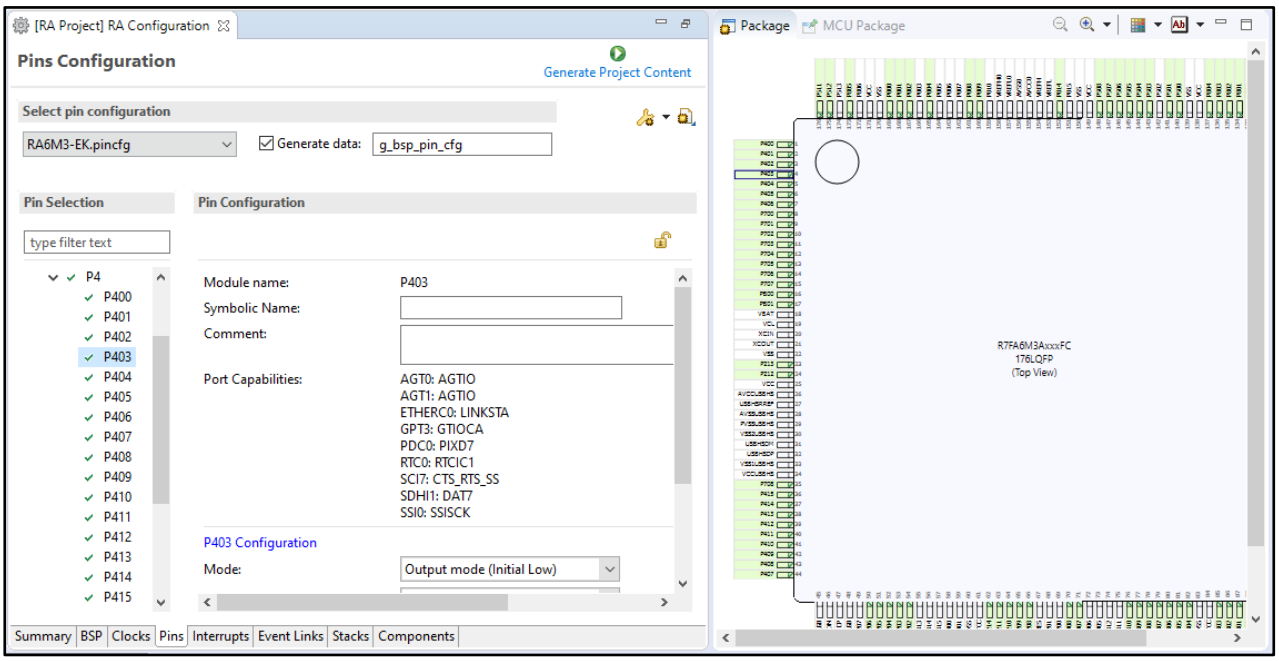
システム全体のクロック構成の確認・変更をすることができます。マイコン内ではコンポーネントごとに異なる分周のクロックを供給する必要がありますが、この画面を使って視覚的に設定を行うことができます。



この設定では外部の 24MHz のクリスタルから PLL で逓倍したクロックを各モジュールに供給しています。CPU のメインクロックは一番上にある ICLK です。

Pins

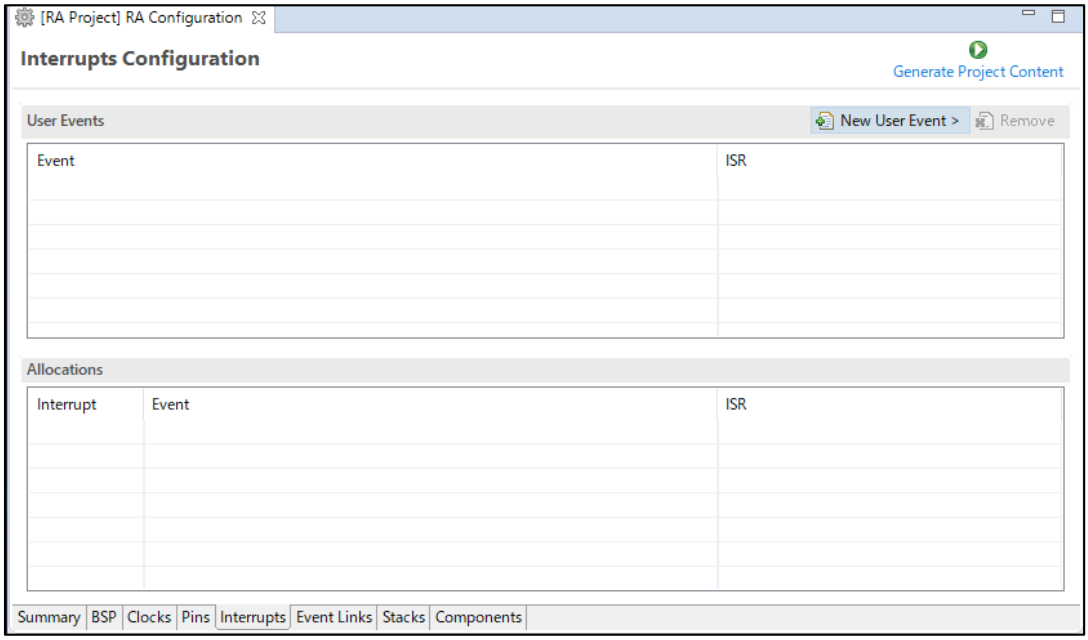
MCU の各ピンの割り当てを設定・確認することができます。



今回制御する LED1 は P1 ポートの P403 ピンなのですが、GPIO の出力モード（初期値 Low 出力）ということがわかります。薄いグリーンになっているピンが何らかの設定がされているピンです。また、この設定は **g_bsp_pin_cfg** というデータファイルで生成され、初期化コードの中で設定がされます。

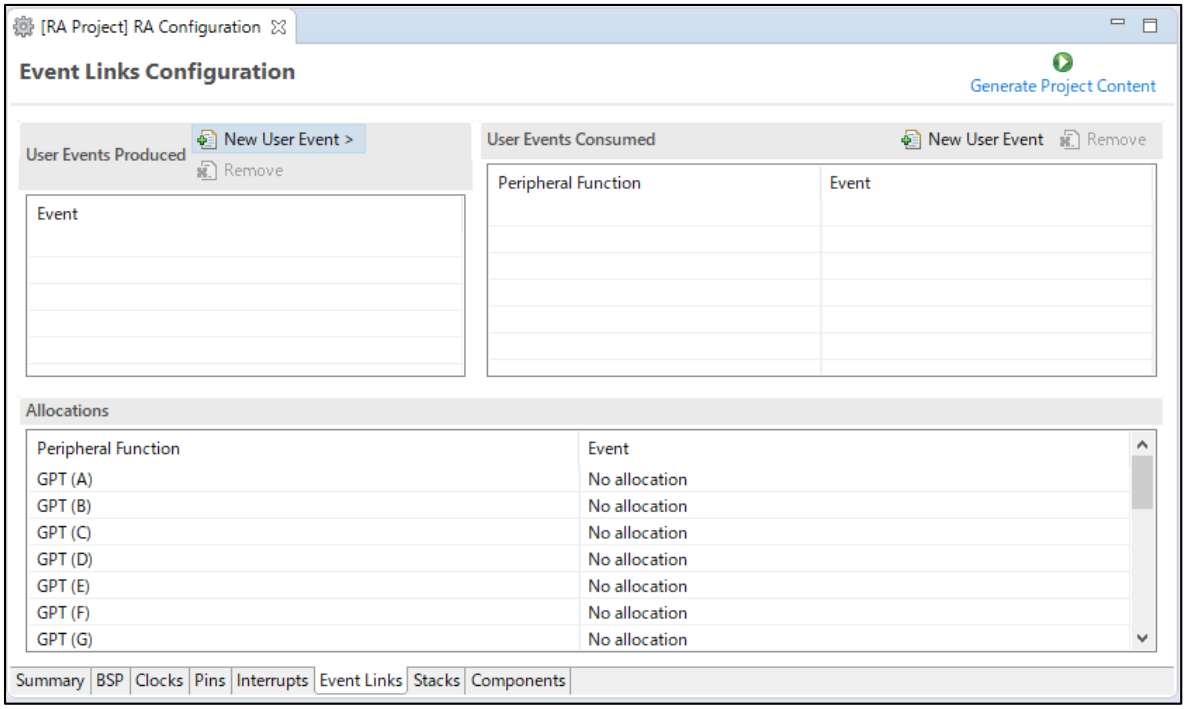
Interrupt Configuration

RA Smart Configurator で設定したペリフェラルの割り込みの確認などができます。また、ユーザ独自の Event を作成し、割り込みと紐づける、といった設定も可能です。



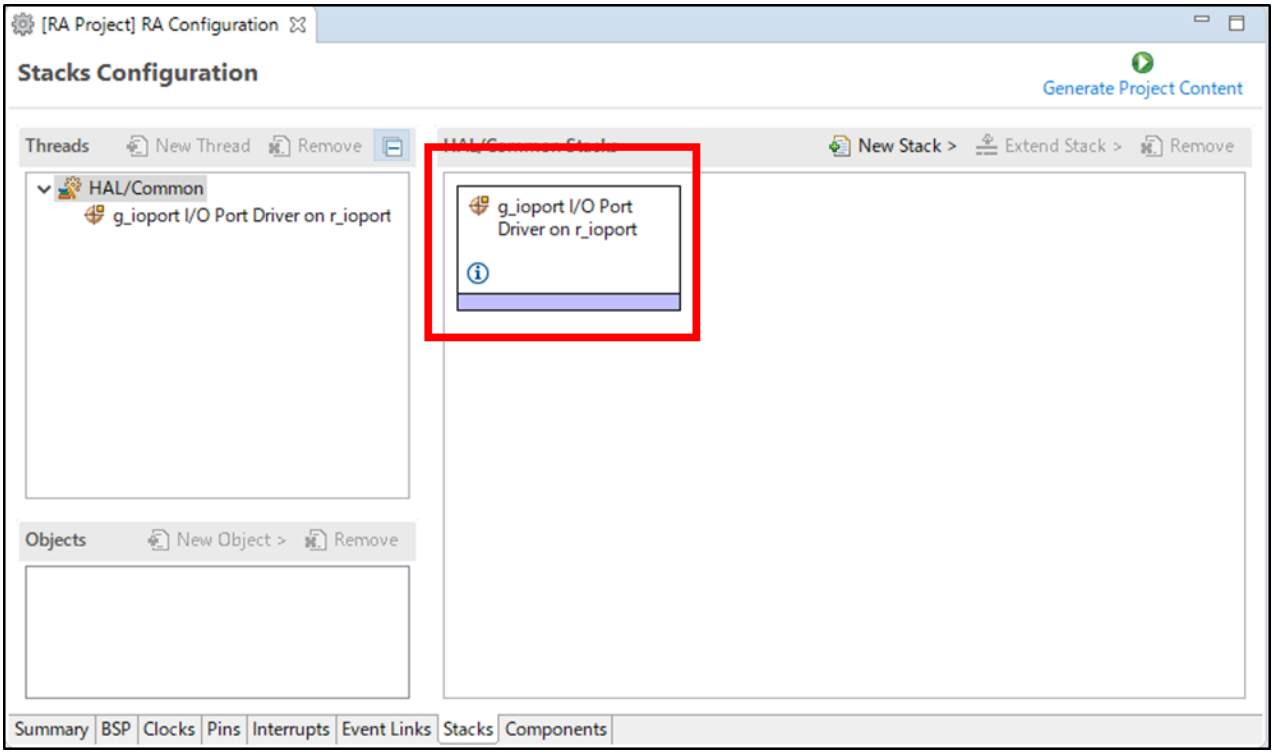
Event Links

RA ファミリ独自の ELC（Event Link Controller）機能を使用して、周辺モジュールで生成されたイベントを他のモジュールへのシグナルとしてリンクさせることができます。本ドキュメントでは使用しません。



Stacks

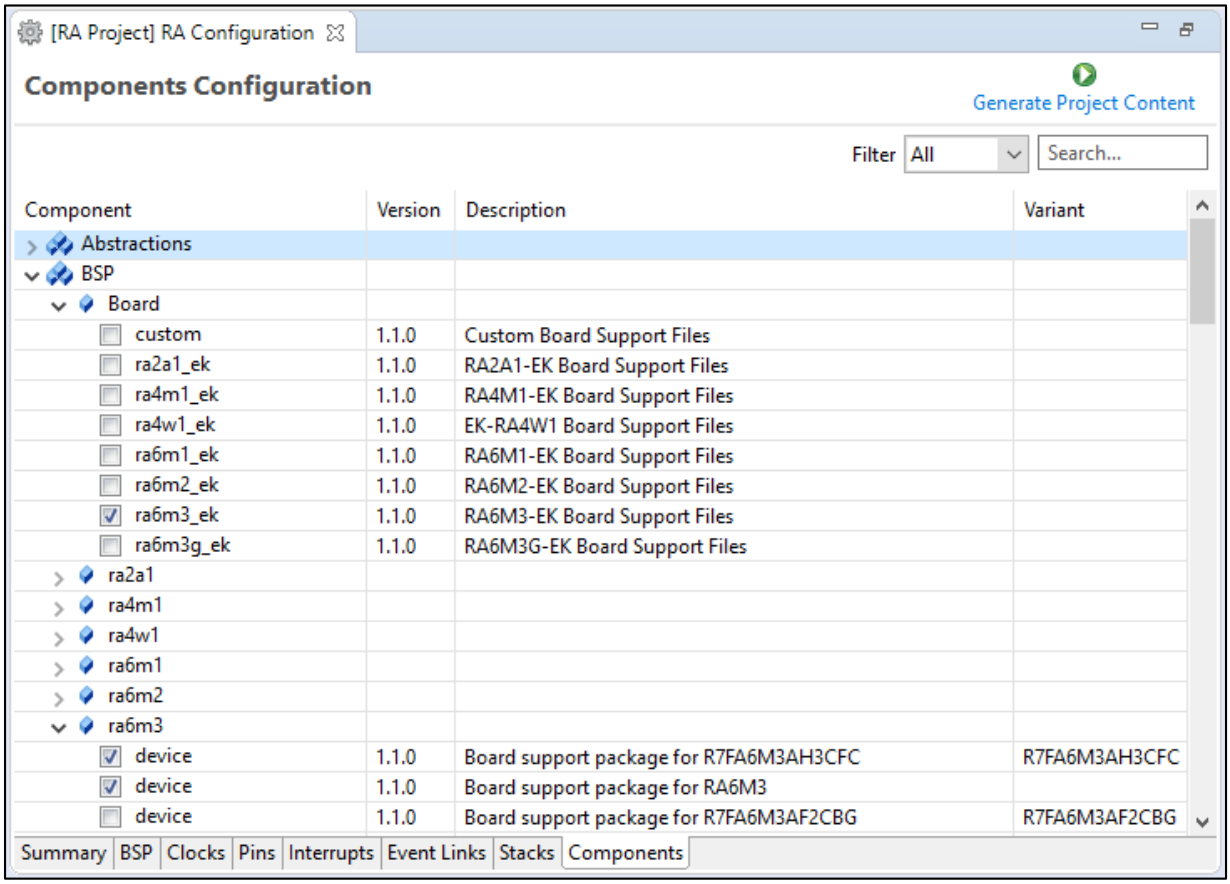
ソフトウェアブロックを Stack というモジュールで配置をすることができます。FSP および RA Smart Configurator の非常に独特なコンセプトです。スタックによっては親子関係を持つこともあり、必要な（依存する）スタックが配置されてない場合にはエラーが表示されます。スタックにはペリフェラルのドライバのほか、ミドルウェアなどの大規模なコンポーネントもあります。



デフォルトで、GPIO 制御用の **g_ioport** スタックがプロジェクトに組み込まれます。

Components

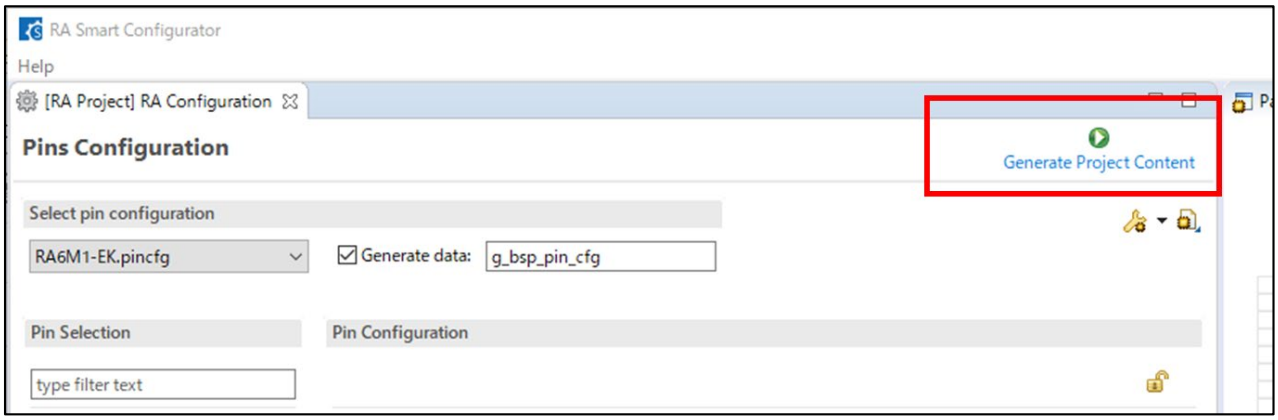
これまでの設定によってプロジェクトに取り込まれる（コード生成される）ファイルが確認できます。基本的にはこの画面は確認用で、手動でチェックを追加することはありません。



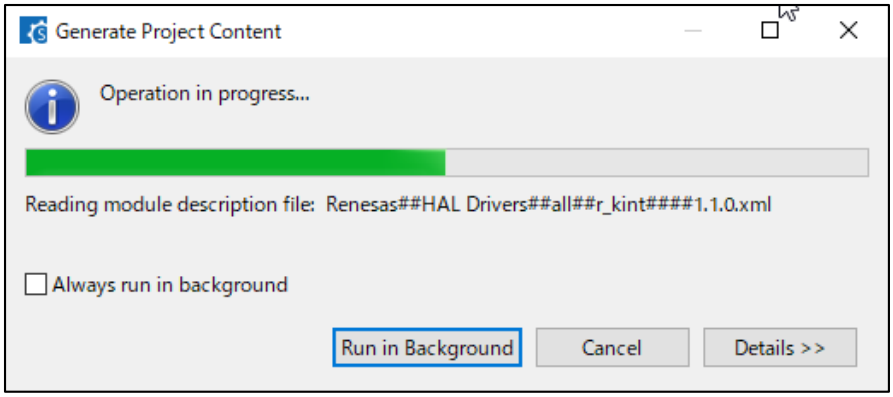
EK-RA6M3 用の最低限の設定は既に行われていることがわかりました。まずはこの状態でプロジェクトを生成します。

8.4. RA Smart Configurator でファイルを生成

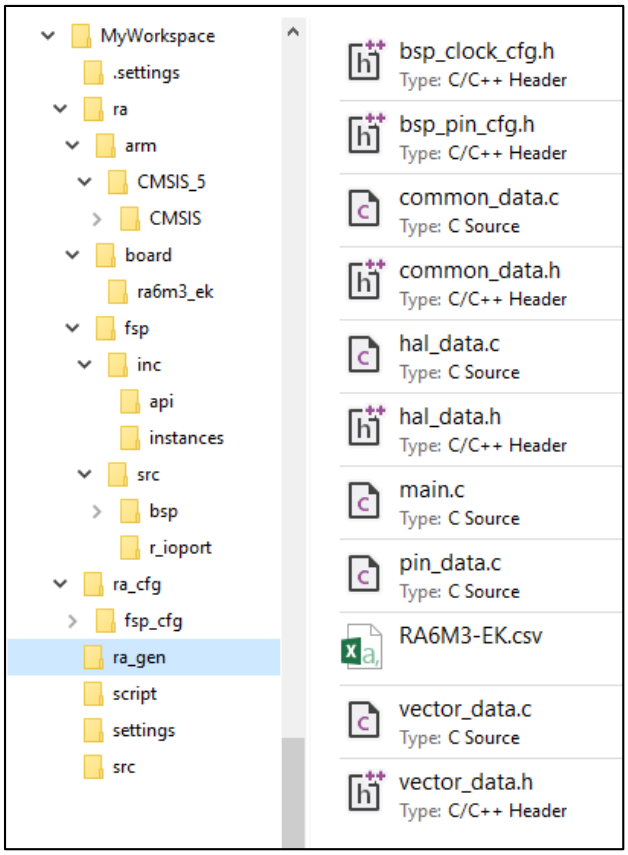
RA Smart Configurator で設定が終わったらファイルの生成を行います。各種エラーがないことを確認したら、**Generate Project Content** ボタンをクリックします。



ファイルの生成が実行されます。



先ほど EWARM で作成したプロジェクトフォルダに多くのフォルダ、ファイルが生成されます。



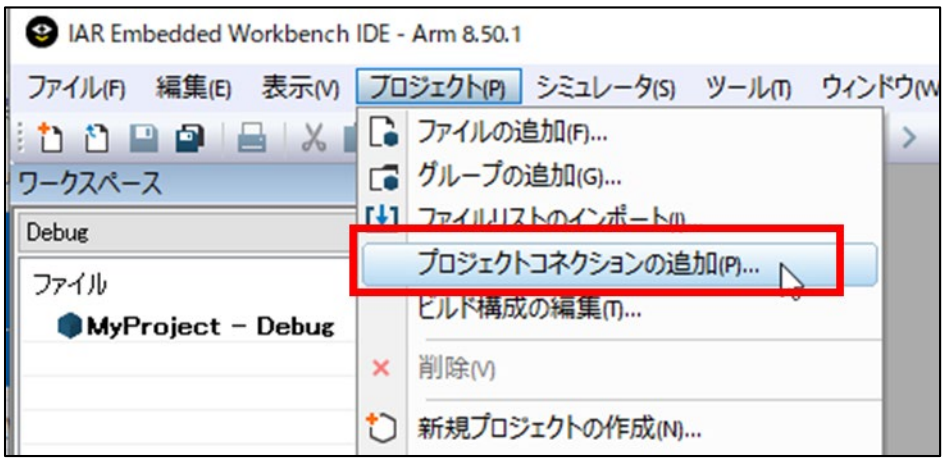
RA Smart Configurator で設定を変更して、Generate をするごとにこれらのファイルが更新されます。

8.5. EWARM のプロジェクトコネクションで、RA Smart Configurator 生成ファイルを取り込み

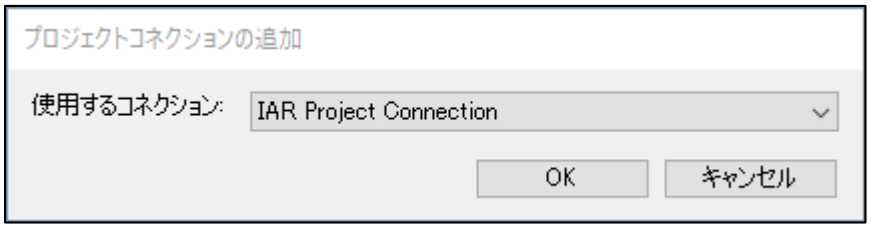
EWARM ではフォルダ内のファイルを自動で取り込む動作はしません。プロジェクトコネクションという仕組みを使用して、RA Smart Configurator の生成ファイル群と連携します。

RA Smart Configurator は後ほど再度使用するので、起動したままにしておいてください。

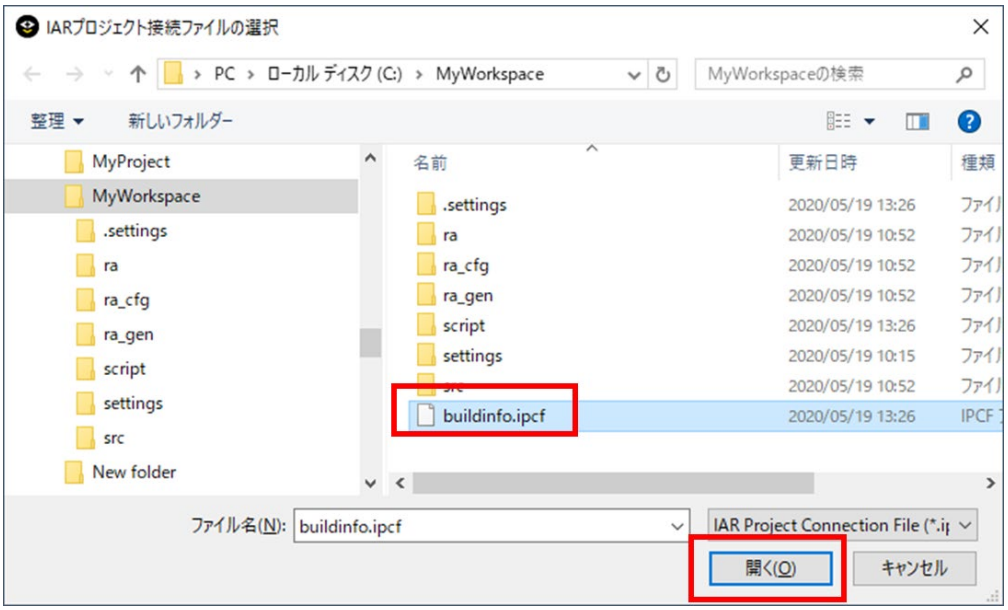
EWARM の画面に戻り、**プロジェクト** > **プロジェクトコネクションの追加** を選択してください。



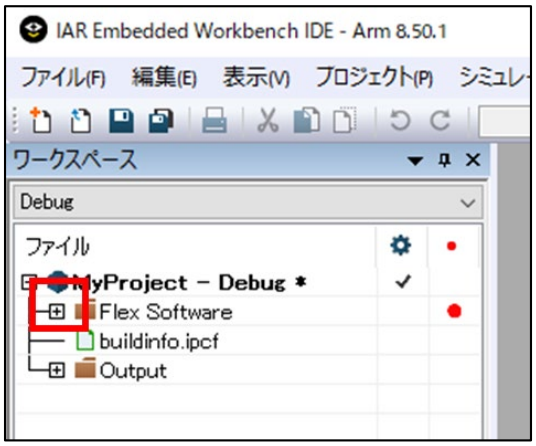
IAR Project Connection を選択し **OK** をクリックしてください。



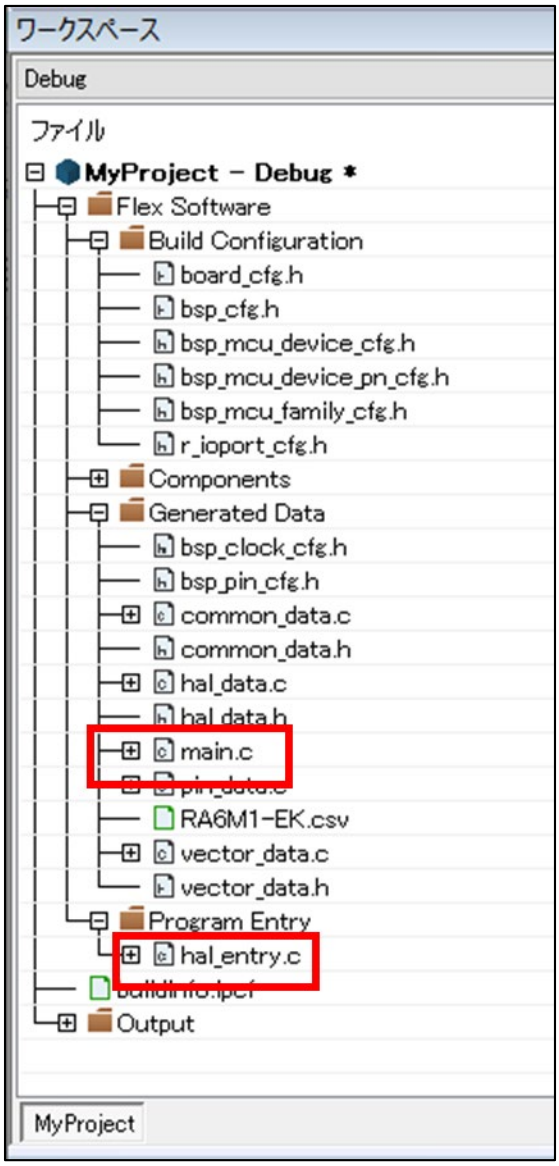
buildinfo.ipcf ファイルを選択して、**開く** を実行してください。



Flex Software グループがプロジェクトに追加されます。



Flex Software のグループを展開すると、生成されたファイル群が確認できます。

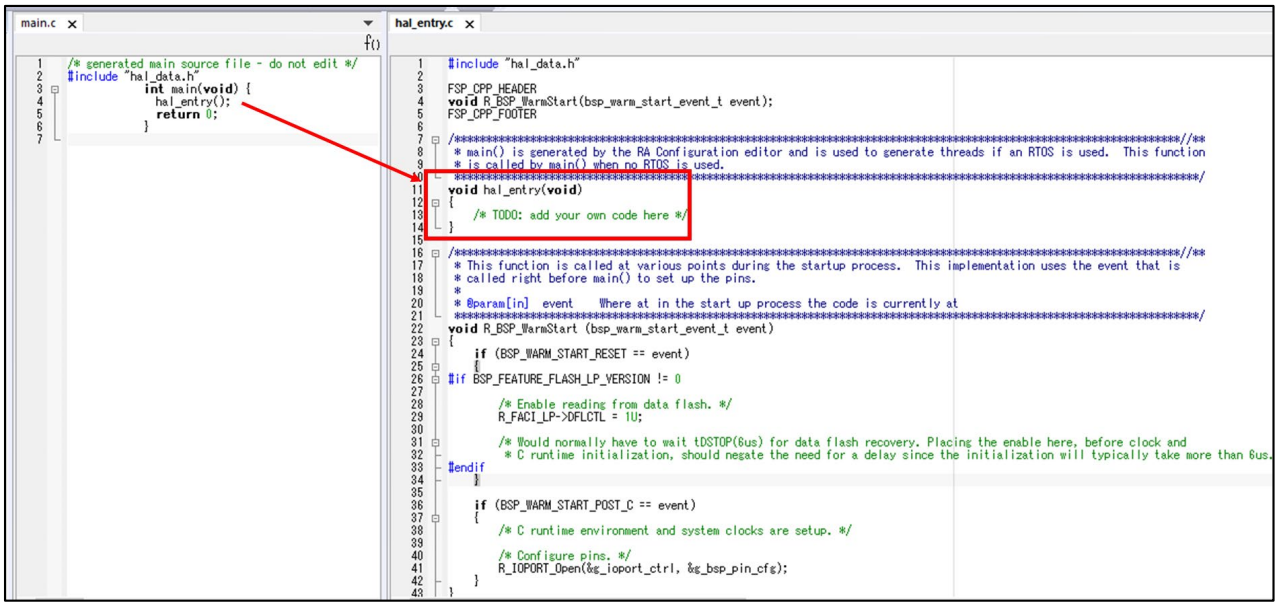


Components には HAL Driver など通常”As Is”で使用されるコードが生成されます。

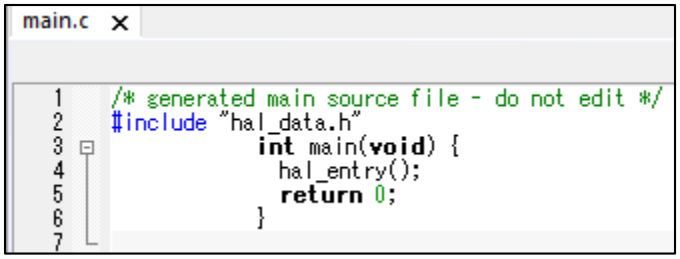
Build configuration、**Generated Data** に RA Smart configurator で行った設定に従ってコードが生成されます。

Program Entry グループは hal_entry.c ファイルを含んでいます。このファイルに記述されている ha_entry()関数からユーザコードは開始されます。

以下の画面で、**hal_entry()**関数が main()関数から呼ばれていることがわかります。



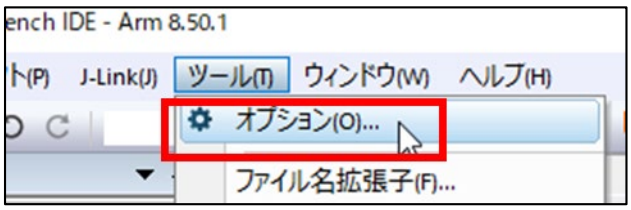
なお、main.c は Generated Data グループに存在するのですが、先頭行に do not edit というコメントが記載されています。



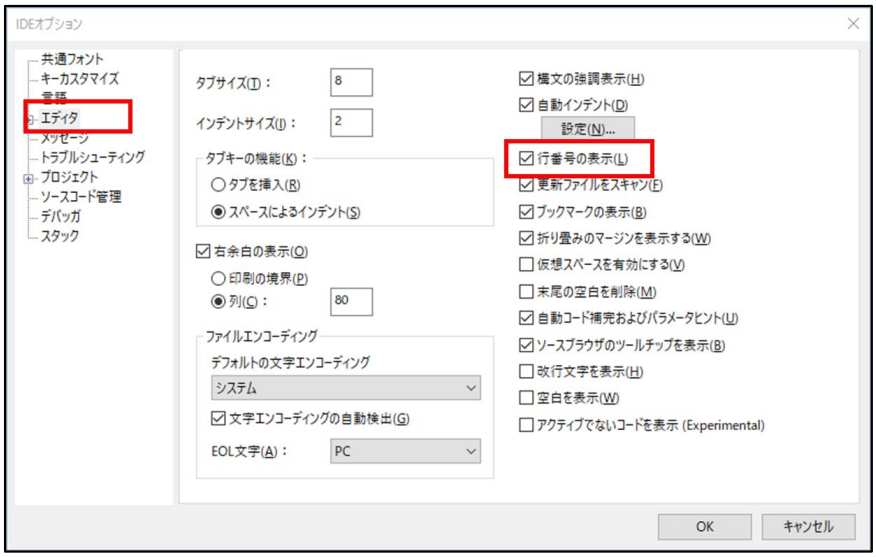
該当のコメントのあるソースコードは、RA Smart Configurator により上書きされる可能性があるので、編集せずに使用してください。

エディタの設定を変更することで行番号を表示できるようになります。

メニューバーの **ツール** >**オプション** を開きます。



エディタ カテゴリの **行番号の表示** にチェックをつけてください。

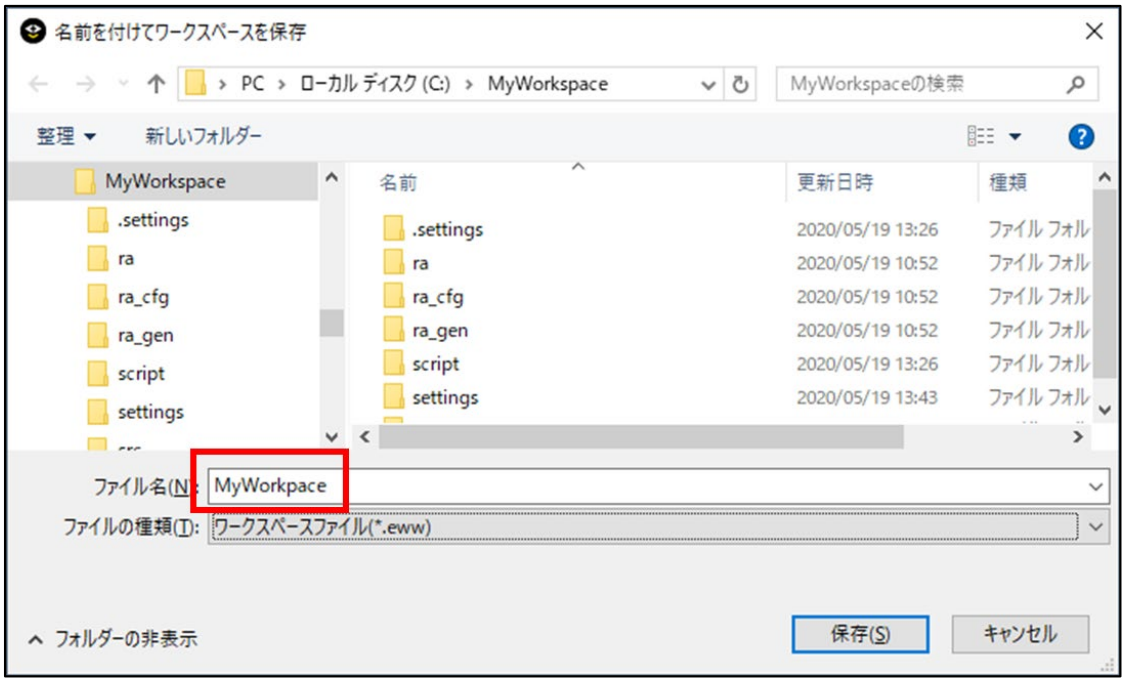


8.6. ワークスペースファイルを保存する

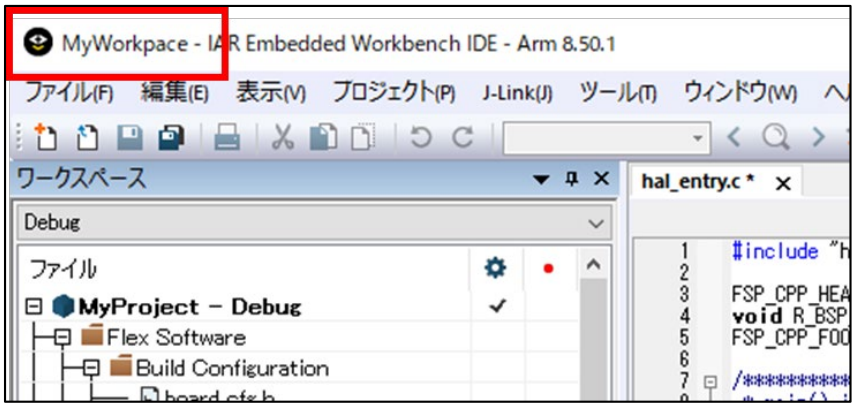
EWARM のプロジェクト管理構成は、ワークスペースがトップ階層で、ワークスペースに対してプロジェクトが所属をします。一つのワークスペースに対して、複数のプロジェクトを作成・登録することもできます。これまでの手順ではプロジェクトは作成していましたが、ワークスペースを保存していなかったので、ここで保存をしておきます。

メニューバーの **ファイル > ワークスペースの保存** を選択します。

ワークスペースの名前を聞かれるので、ここでは MyWorkspace と指定して保存します。



タイトルバーにワークスペース名が表示されるようになります。

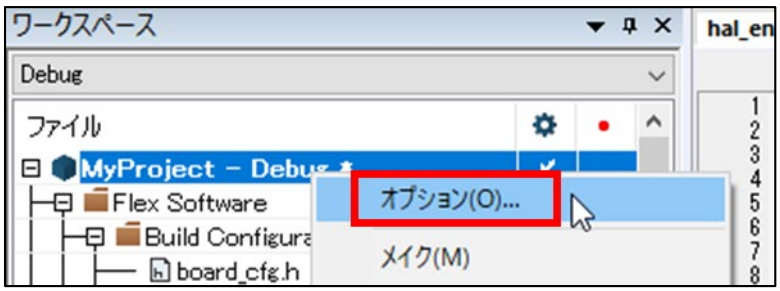


今後複数のプロジェクトを同時に立ち上げているときなど、ワークスペースの識別をするのに便利です。

8.7. EWARM のプロジェクトオプションを変更

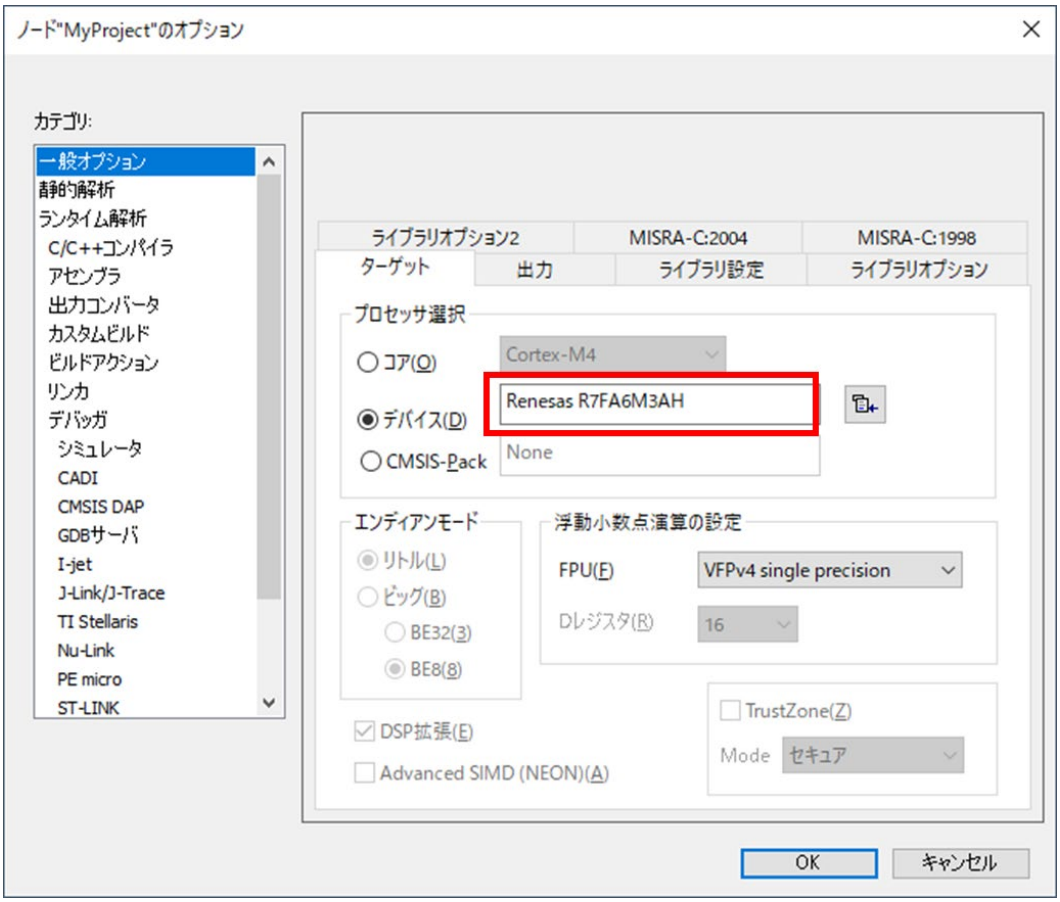
ソースコードが取り込まれたので、プロジェクトのオプション設定を確認・変更していきます。

MyProject アイコンを右クリックして、**オプション** を開きます。



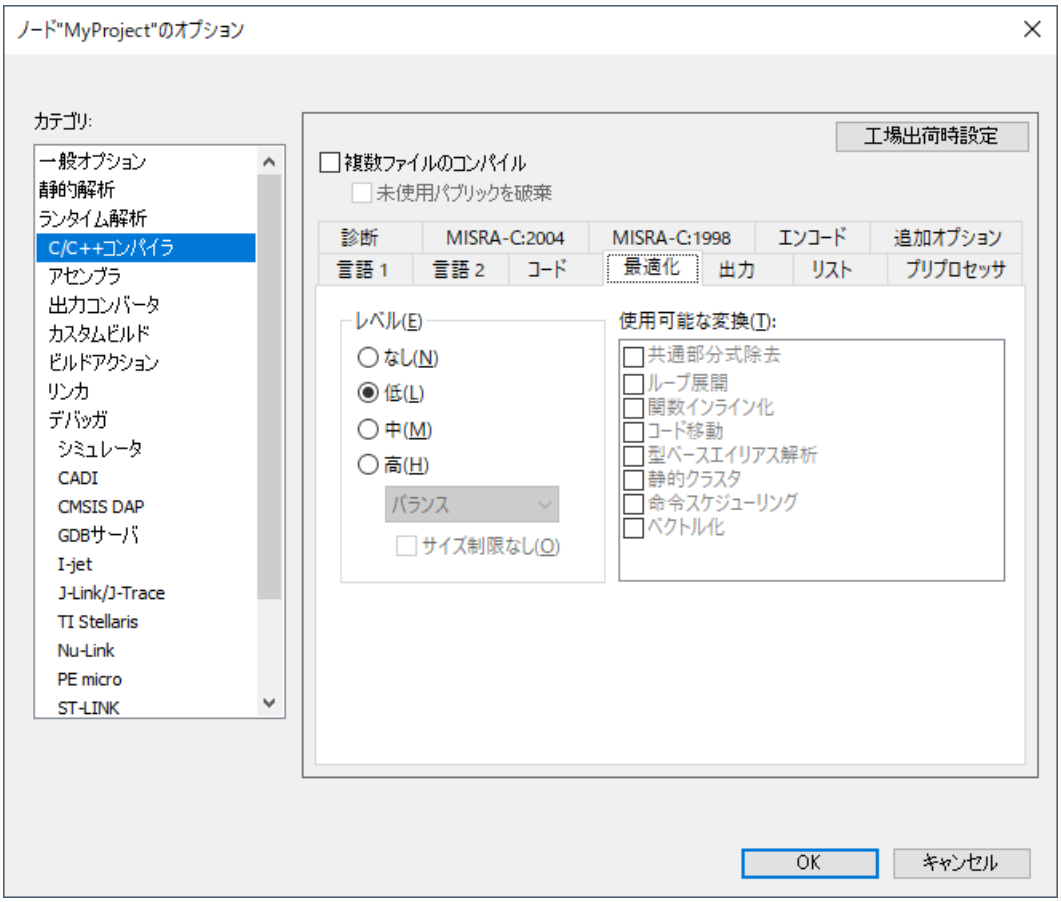
最初のターゲット画面で、**R7FA6M3AH** デバイスが選択されていることがわかります。

プロジェクトコネクションで取り込んだ際に、ビルドをするための最低限の設定も行われています。

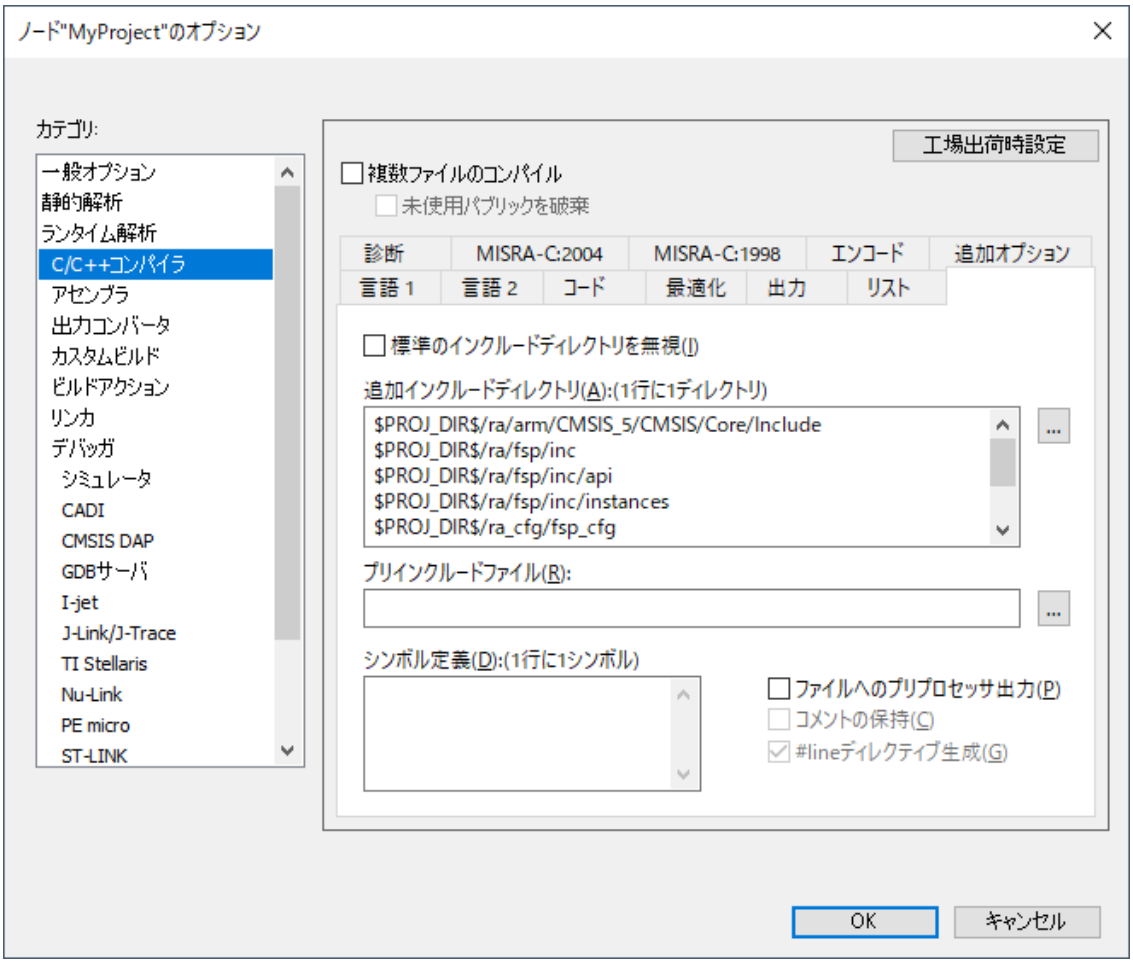


C/C++コンパイラ カテゴリの **最適化** タブを見るとデフォルトの最適化レベルは **低** になっていることがわかります。この設定はソースコード連携デバッグの効率を上げるための標準的な設定です。

今後アプリケーションの性能を高めたい、あるいはコードサイズを小さくしたい、という際には最適化レベルを上げてください。最適化レベルをあげた場合ステップ実行は引き続き可能ですが、変数のレジスタ割り当てなどによりデバッグがしづらくなることがあります。

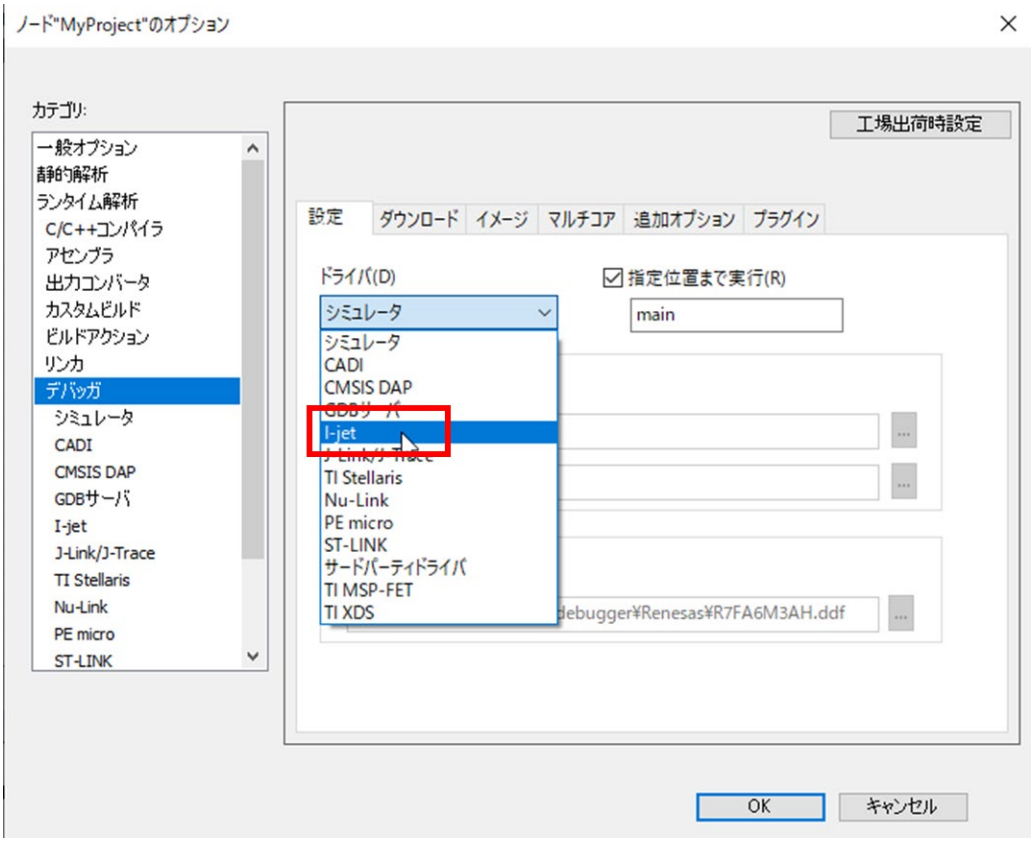


プリプロセッサ タブではプロジェクトで参照しているヘッダファイルのインクルードパスも確認ができます。



RA Smart Configurator の生成コードが構造化されていることがわかります。

デバッガの設定では変更が必要です。



デバッガ > **設定** > **ドライバ**がデフォルトの**シミュレータ** から **I-jet** に変更してください。J-Link OB を使用する際には **J-Link/J-Trace** を選択してください。

以上で設定の確認、変更は終了です。**OK** をクリックしてください。

8.8. ユーザコードを記述

先ほど見た通りアプリケーションのエントリーとなる hal_entry 関数は空なので、テスト処理を記述します。

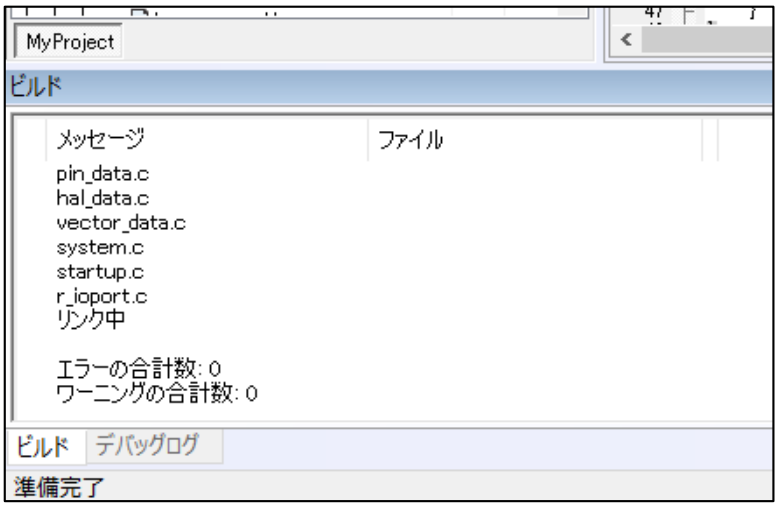
```
void hal_entry(void)
{
    static int count;
    while(1)
    {
      for(volatile uint32_t dly=10000000;dly;dly--){};
      count++;
    }
}
```

while（1）の無限ループの中で、ソフトウェアディレイで待ちながら count 変数をインクリメントしていく、というシンプルなコードです。

8.9. ビルド → デバッグ

プロジェクトをビルドしてきます。ツールバーの **メイク** ボタンをクリックするとメイクが行われます。

エラーの合計数:0 となればメイク成功です。

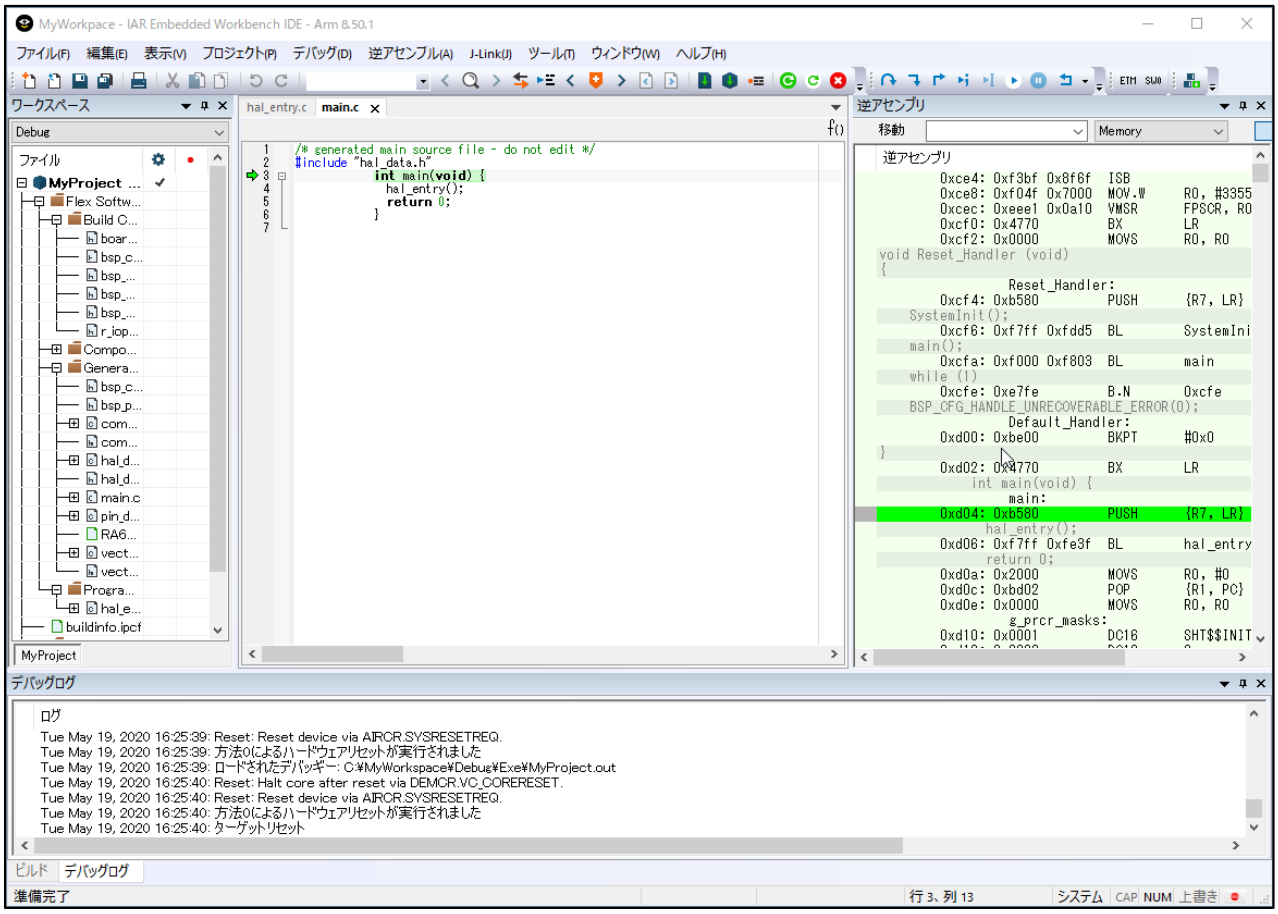


デバッグを開始する前に、以下の設定・接続を確認してください：

1. I-jet とコンピュータを USB ケーブルで接続
2. J20 ヘッダに I-jet を接続
3. J15 ジャンパが 1-2 をクローズ
4. J10 USB ポートがコンピュータと USB ケーブルで接続
5. J9 ジャンパが 1-2 をクローズ
6. J8 ジャンパが 1-2 をクローズ

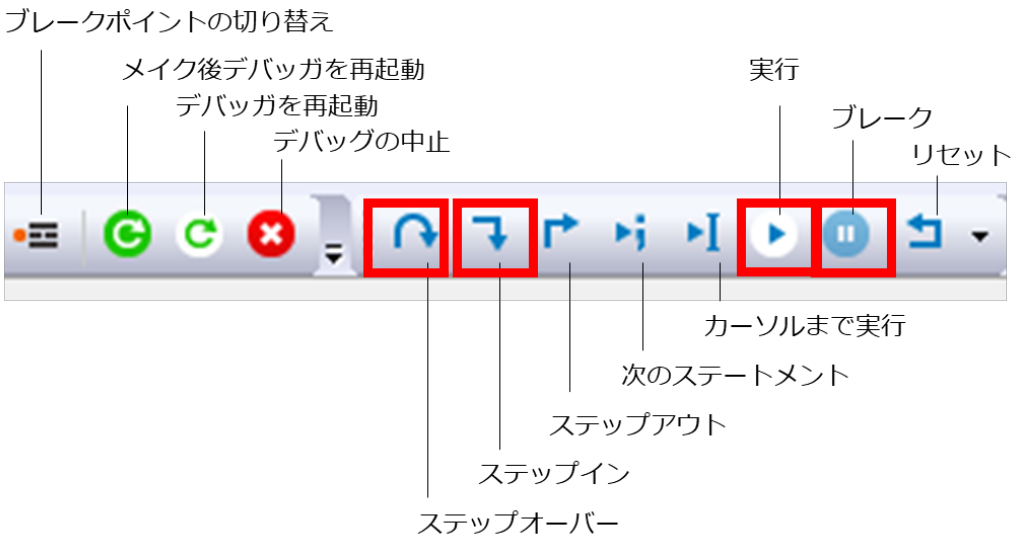
生成された実行ファイルを RA6M3 のフラッシュメモリにダウンロードし、デバッグ実行を開始します。

ダウンロードしてデバッグ (Ctrl+D) をクリックしてください



main 関数の先頭で止まっていることがわかります。

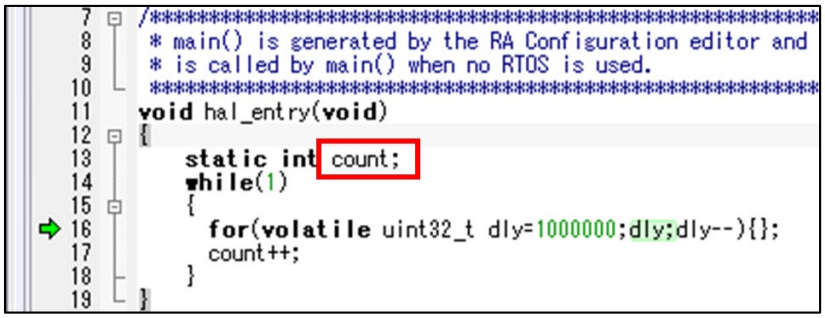
右上に表示される実行制御アイコンで、ステップ実行や Run 実行、ブレークなどの操作ができます。



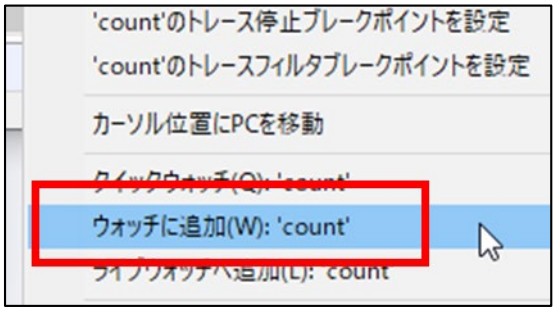
実行 し、

ブレーク します。

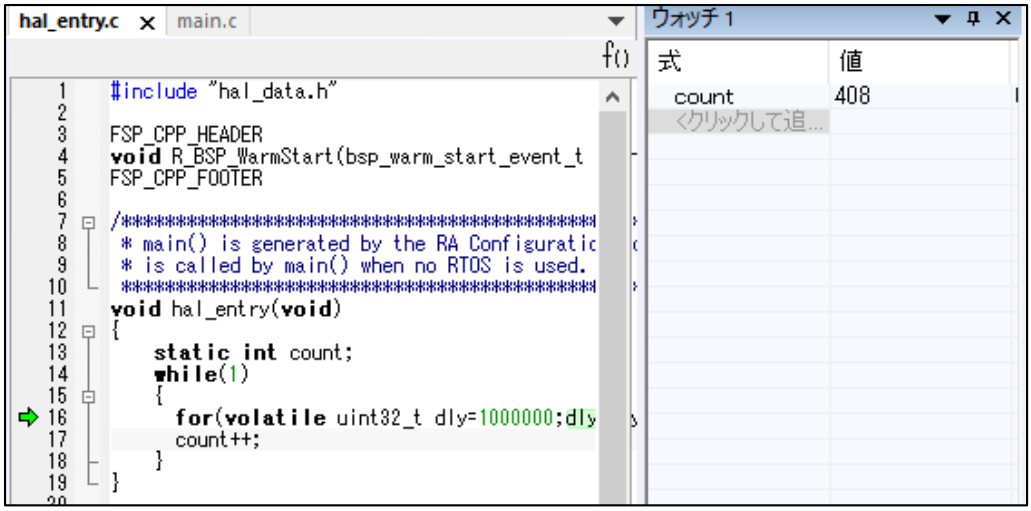
hal_entry 関数のソフトウェアディレイを実行中だということが分かります。



変数の値を確認するため count 変数を右クリックし、**ウォッチに追加** を選択してください。



ウォッチウィンドウが表示され、現在の値が確認できます。



なお、ウィッチウィンドウには手動で変数を登録することもできます。

基本的なビルド、デバッグ操作が確認できました。 **デバッグの中止** をクリックして、デバッグセッションを終了します。

9. 評価ボード上の LED を制御

続いて、評価ボード上の LED をコードから制御をしていきます。GPIO を含めた周辺レジスタはメモリマッピングされており、指定アドレスにアクセスをすることで制御ができるのですが、コーディング負荷が高くなるので、FSP で用意されている API を使用して制御を行います。

9.1. LED 制御コード

実際に LED を点滅するコードを見てみましょう。

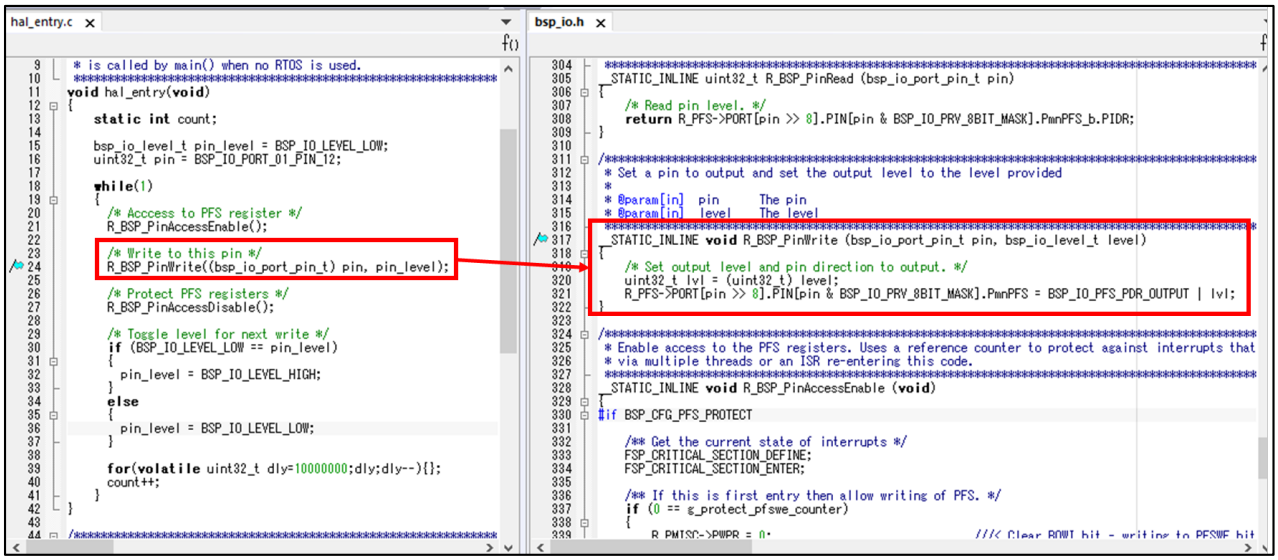
```
void hal_entry(void)
{
    static int count;

    bsp_io_level_t pin_level = BSP_IO_LEVEL_LOW;
    uint32_t pin = BSP_IO_PORT_04_PIN_03;

    while(1)
    {
      /* Acccess to PFS register */
      R_BSP_PinAccessEnable();

      /* Write to this pin */
      R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);

      /* Protect PFS registers */
      R_BSP_PinAccessDisable();

      /* Toggle level for next write */
      if (BSP_IO_LEVEL_LOW == pin_level)
      {
        pin_level = BSP_IO_LEVEL_HIGH;
      }
      else
      {
        pin_level = BSP_IO_LEVEL_LOW;
      }

      for(volatile uint32_t dly=10000000;dly;dly--){};
      count++;
    }
}
```

赤字にした箇所が元のコードに対して追記をした箇所になります。

実際に LED（GPIO）をトグルしているのは R_BSP_PinWrite 関数になります。

この制御で使用している各種定義は Components > bsp_io.h に記載されています。

R_BSP_PinWrite 関数は引数としてピン番号と、High か Low からのレベルを受け取ります。



BSP_IO_LEVEL_LOW、BSP_IO_LEVEL_HIGH、BSP_IO_PORT_04_PIN_03 のいずれも bsp_io.h で enum 定義されています。なお、BSP_IO_PORT_04_PIN_03 は LED が接続されている P4 03 ピンを示します。

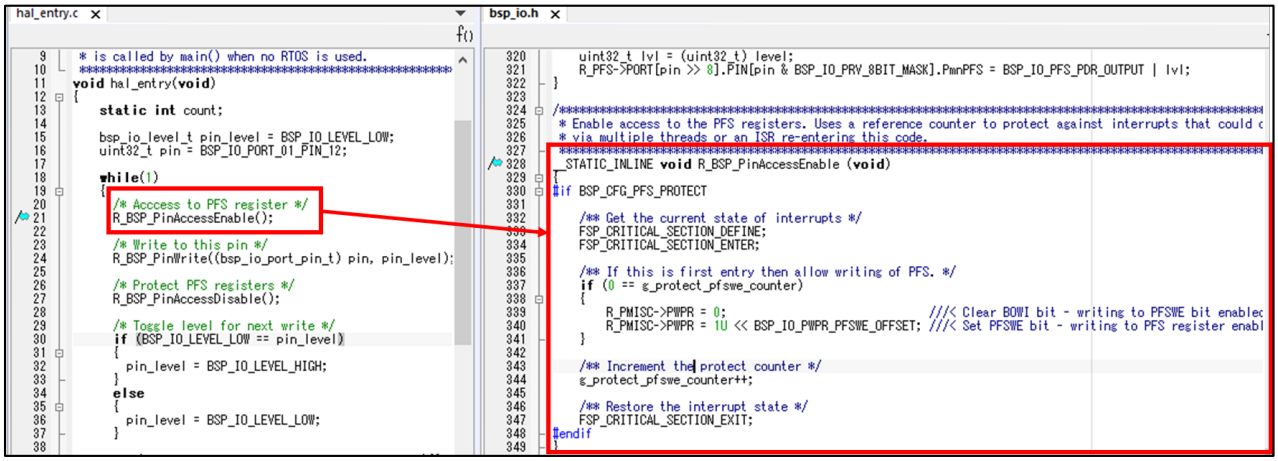
9.2. PFS レジスタの制御

R_BSP_PinWrite 関数の前後で、

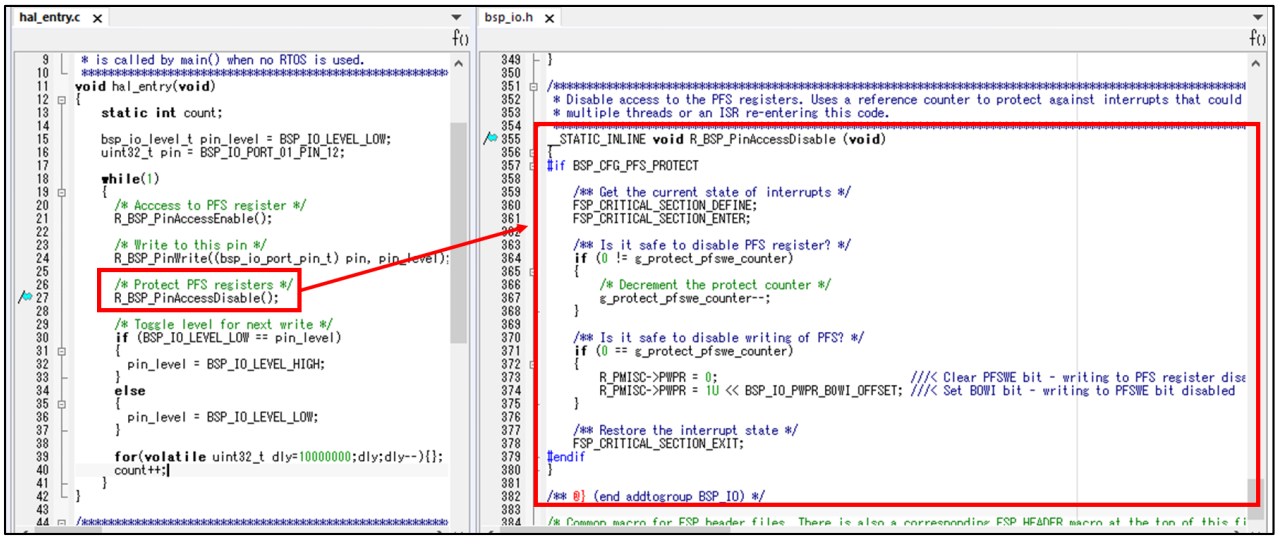
R_BSP_PinAccessEnable();と
R_BSP_PinAccessDisable();
が呼ばれています。

PFS は Pin Function Select モジュールです。Pin への設定をする際に、予期せぬ Pin 設定書き込みを防ぐために Write Enable に設定をする必要があります。

R_BSP_PinAccessEnable 関数では他の割り込みやスレッドが、制御をしようとしていないかフラグを使って確認をした上で、書き込み有効化設定を実行しています。



R_BSP_PinAccessDisable では逆に、PFS レジスタへの書き込みを無効にし、フラグを落としています。



これにより、ピン制御のコンフリクトなどを避けることができます。

9.3. LED 点滅コードの実行

再度 **メイク** し、

エラーが 0 であれば **ダウンロードしてデバッグ** を実行してください。

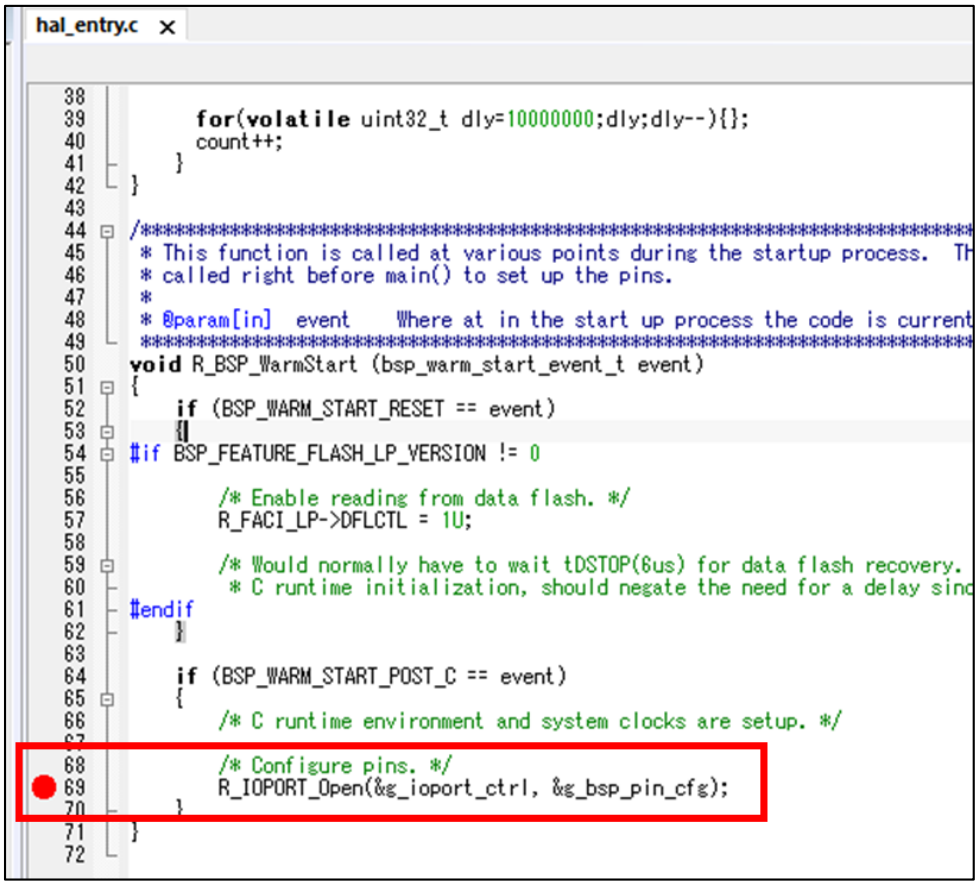
ボード上の LED が約 1 秒周期で点滅することができました。

なお、R_BSP_PinAccessEnable（）と R_BSP_PinAccessDisable（）をコメントアウトすると、LED 点滅しなくなります。ぜひ試して見てください。

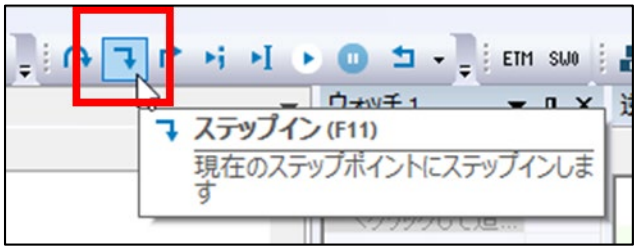
9.4. ピン設定の初期化

通常 GPIO の制御を行う際には、GPIO ピンの初期化処理が必要ですが、hal_entry 関数や main 関数には初期化処理の記述がありません。

実は、ピン初期化は main 関数が実行される前、hal_entry.c の R_BSP_WarmStart 関数内で実行されています。



RA Smart configurator で生成されたピン設定を元に LED 制御ピンではなく他の使用ピン全ての初期化を行っています。処理の詳細は、R_BSP_WarmStart にブレークポイントを設定してデバッグ実行し、**ステップイン（F11）**実行で追ってみてください。



10. タイマー割り込みで LED を点滅する

先ほどの例ではソフトウェアディレイで、LED の点滅を行いました。実は FSP でもソフトウェアディレイの API が用意されています。

```
const bsp_delay_units_t bsp_delay_units = BSP_DELAY_UNITS_MILLISECONDS;

/* Set the blink frequency  (must be <= bsp_delay_units */
const uint32_t freq_in_hz = 2;

/* Calculate the delay in terms of bsp_delay_units */
const uint32_t delay = bsp_delay_units / freq_in_hz;

/* Delay */
R_BSP_SoftwareDelay(delay, bsp_delay_units);
```

R_BSP_SoftwareDelay を使用することで、正確な時間ソフトウェアディレイを実行することができます。

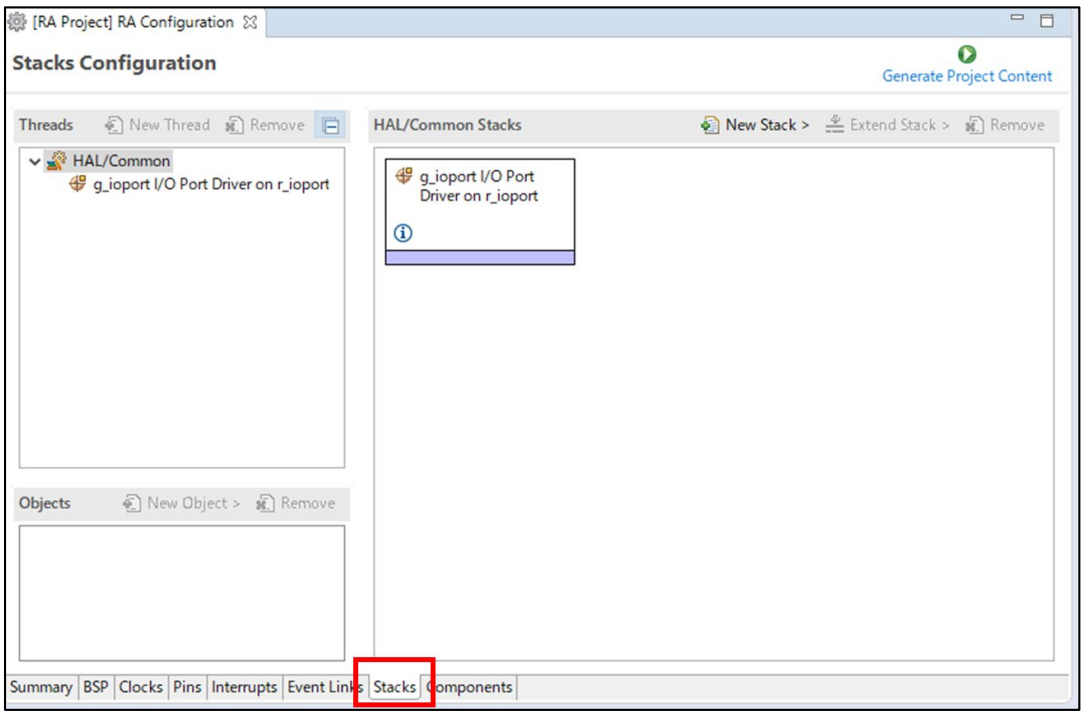
ソフトウェアディレイは、他の処理ができなかったり、不要に電力を消費したり、というデメリットがあるため、タイマーで周期処理を作成することが推奨されます。

今回は RA ファミリの General Purpose Timer の割り込みを使用して、LED の 100 m 秒点滅を実装します。

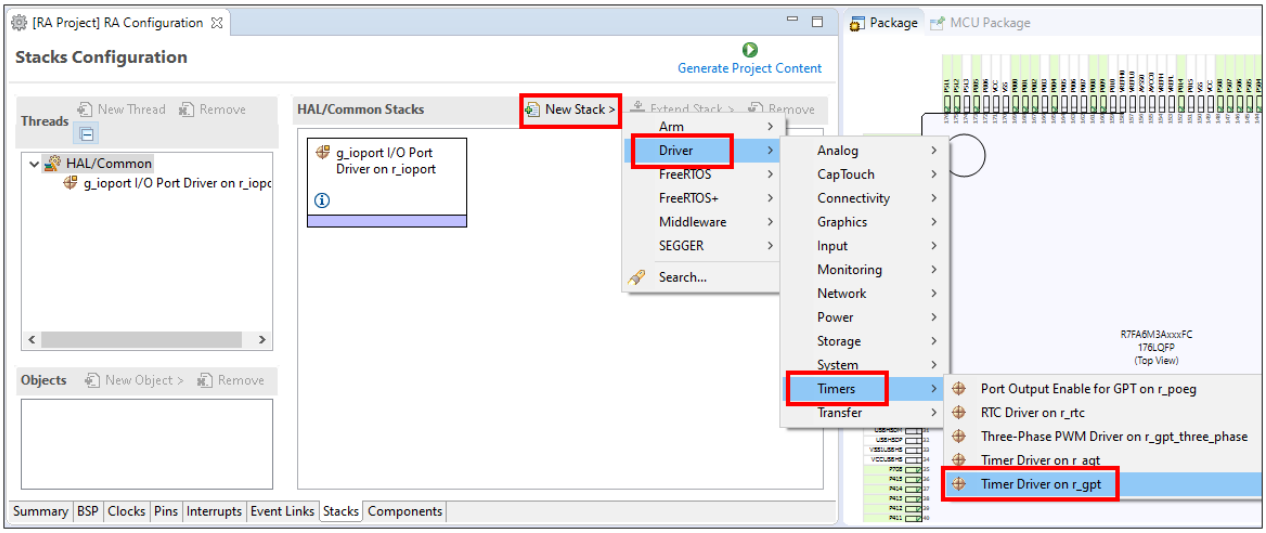
10.1. RA Smart Configurator で Timer スタックを追加

RA Smart Configurator 画面をアクティブにします。RA Smart Configurator が起動していなければ、再度 **ツール** > **RA Smart Configurator** を起動してください。2 回目以降に起動するときは、前回の設定が読み込まれた状態になるため、再設定は不要です。

Stacks タブをクリックします。



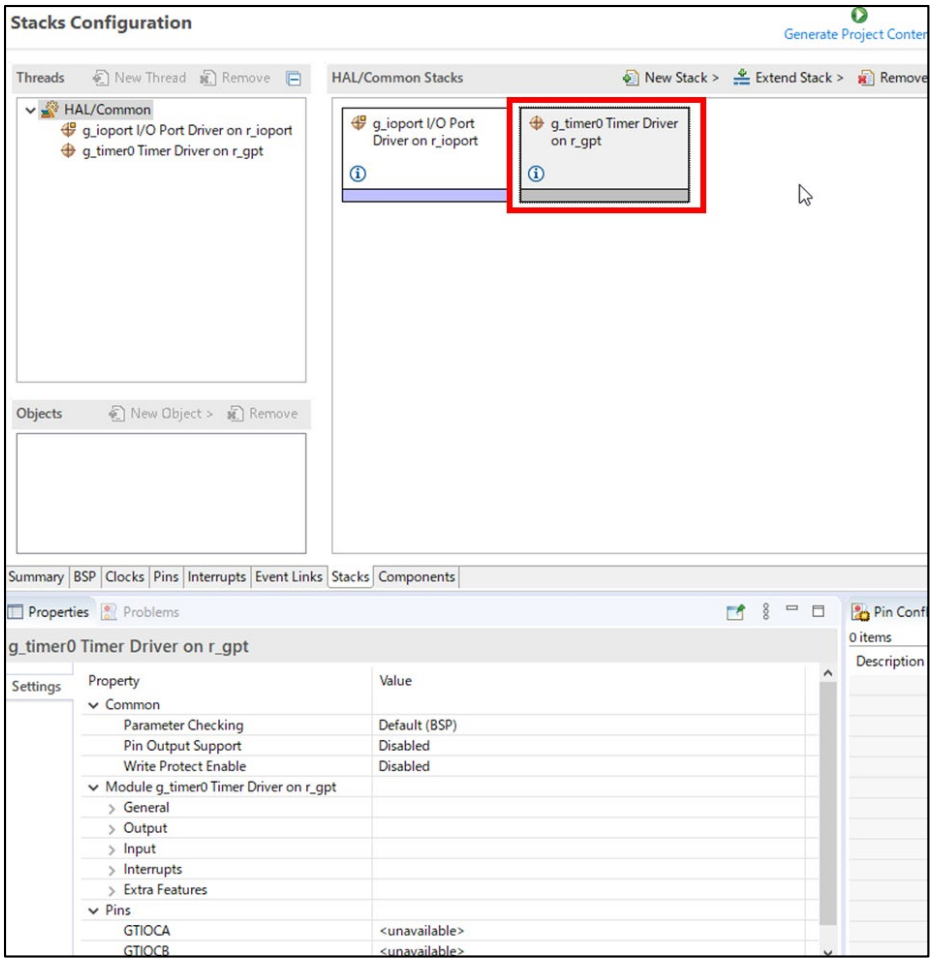
New Stack > **Driver** > **Timers** > **Timer Driver on t_gpt** を選択します。



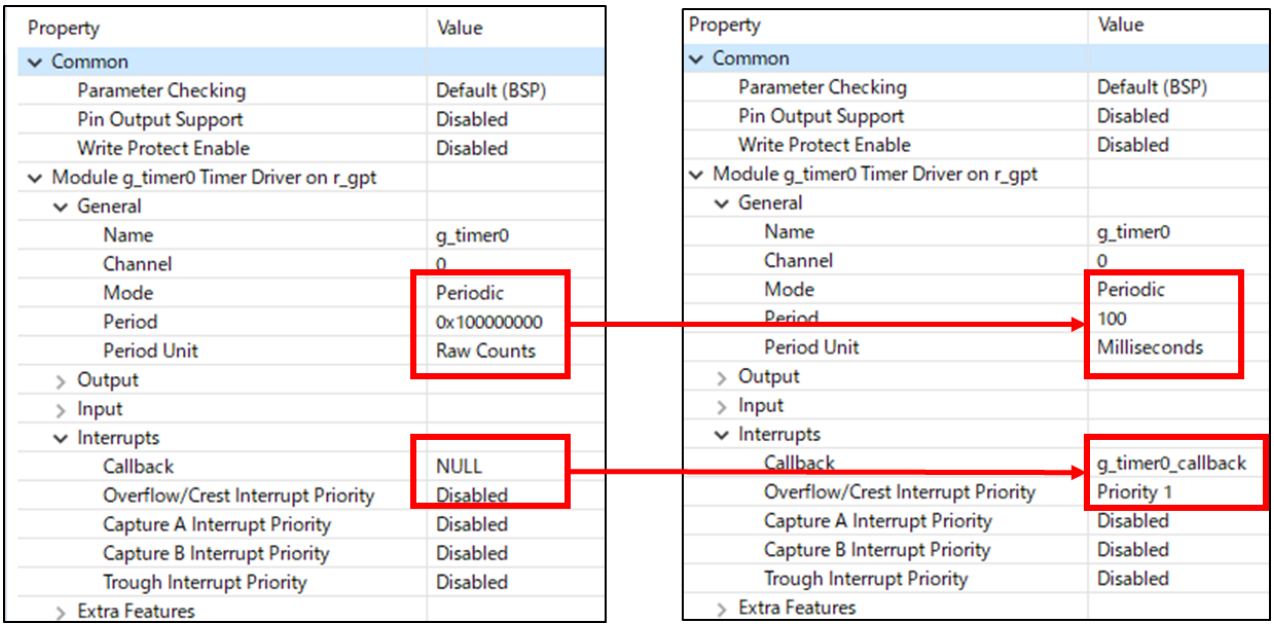
プロジェクトに Timer ドライバが追加されました。下のプロパティウィンドウに各種パラメータが表示されています。

10.2. Timer スタックのパラメータ設定

プロパティのパラメータを変更して 100ms タイマー割り込みを発生させます。



変更する箇所は以下です。



今回は繰り返しタイマーを使用するので、Mode は **Periodic** のままにします。

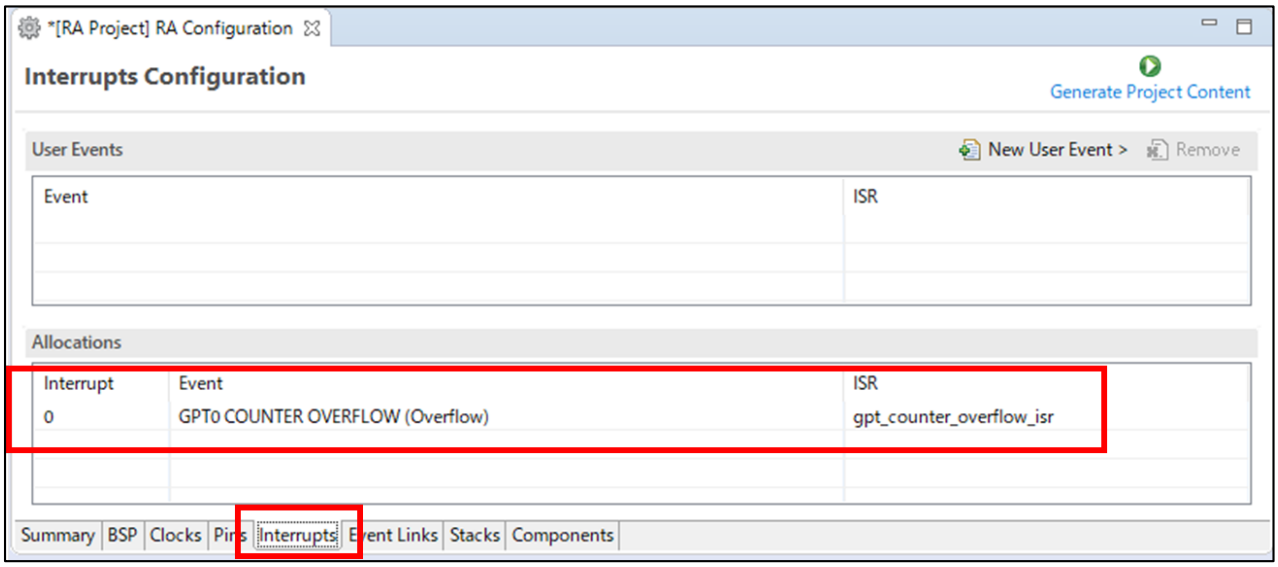
周期はデフォルトではサイクルカウントを指定するのですが、それではわかりにくいので、Period Unit を **Millseconds**、Period を **100** に指定します。これにより、仮に今後 CPU クロックを変更したとしても LED の点滅周期は 100m 秒となるよう連動してくれるようになります。

最後に **Interrupts** > **Callback** に **g_timer0_callback** と設定しました。この関数名は任意です。RA Smart Configurator と FSP を用いた開発の場合、割り込みハンドラで直接処理をするのではなくコールバック関数を登録して、実際の処理はコールバック関数で行います。

g_timer0_callback はユーザコードで実装をします。周期割り込みは Overflow が発生します。Overflow 割り込みの優先度を設定します。ここでは上から 2 番目の **Priority 1** を設定しました。

10.3. 割り込み割り当ての確認

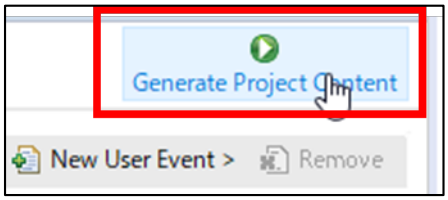
Interrupts タブで先ほど設定したタイマー割り込みの設定を確認します。



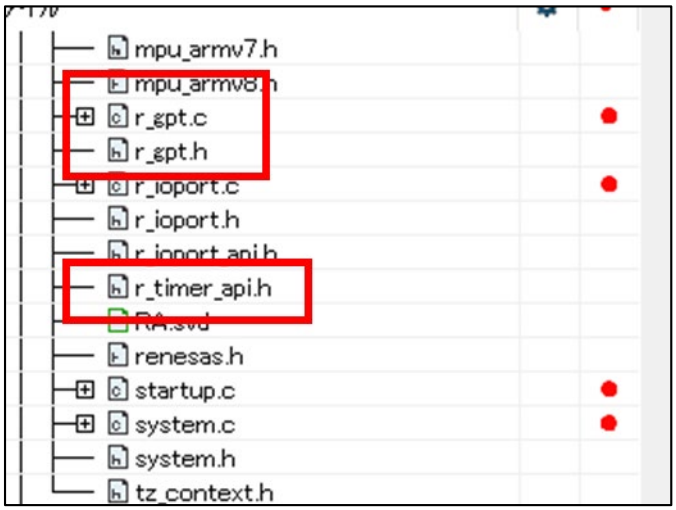
GPT0_COUNTER_OVERFLOW イベントに対して、gpt_counter_overflow_isr が呼ばれることが確認できました。**_isr** は RA Smart Configurator が生成する割り込みハンドラです。実際にはここから先ほど設定したコールバック関数が呼ばれます。

10.4. コードの再生成

EWARM がデバッグ中でないことを確認し、**Generate Project Content** をクリックします。

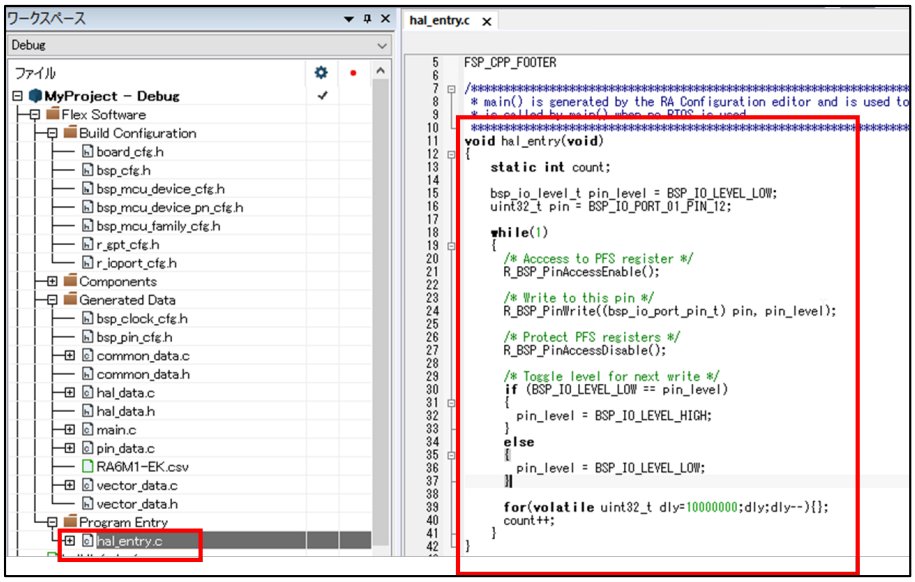


EWARM をアクティブにするとプロジェクトが更新され低レイヤードライバの r_gpt.c/h やラッパーとなる r_timer_api.h などが追加されていることがわかります。



タイマー処理は可読性、メンテナンス性を考慮して、基本的にラッパーAPI を使用して実装することを推奨します。

hal_entry.c は RA Smart Configurator で再生成を行っても更新されません。

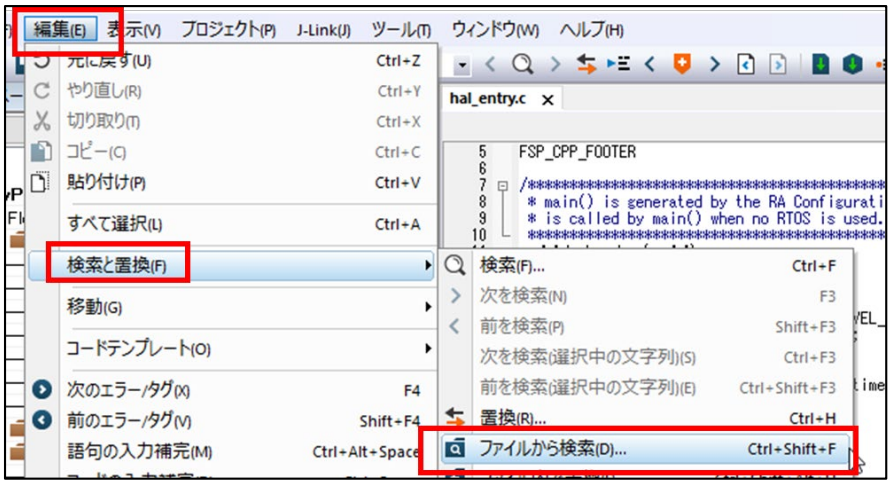


アプリケーション開発を継続することができます。Generated Data Group のファイルは再生成すると上書きされることがあるので、編集をしないようにしてください。

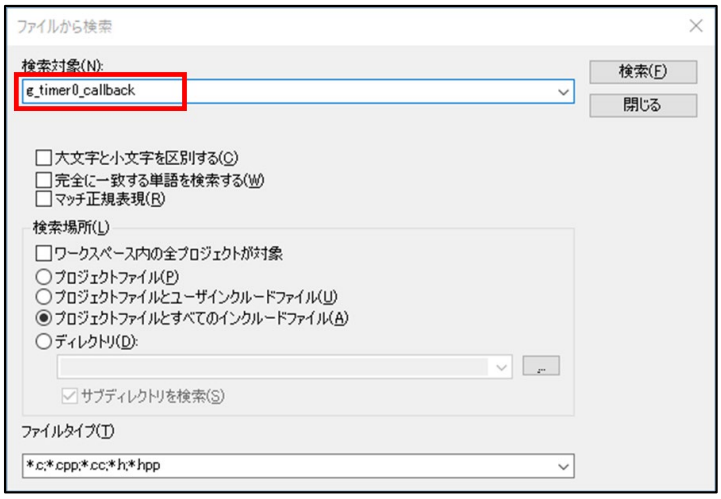
10.5. タイマー割り込みコールバック関数の実装

RA Smart Configurator で指定した g_timer0_callback をコードに実装します。

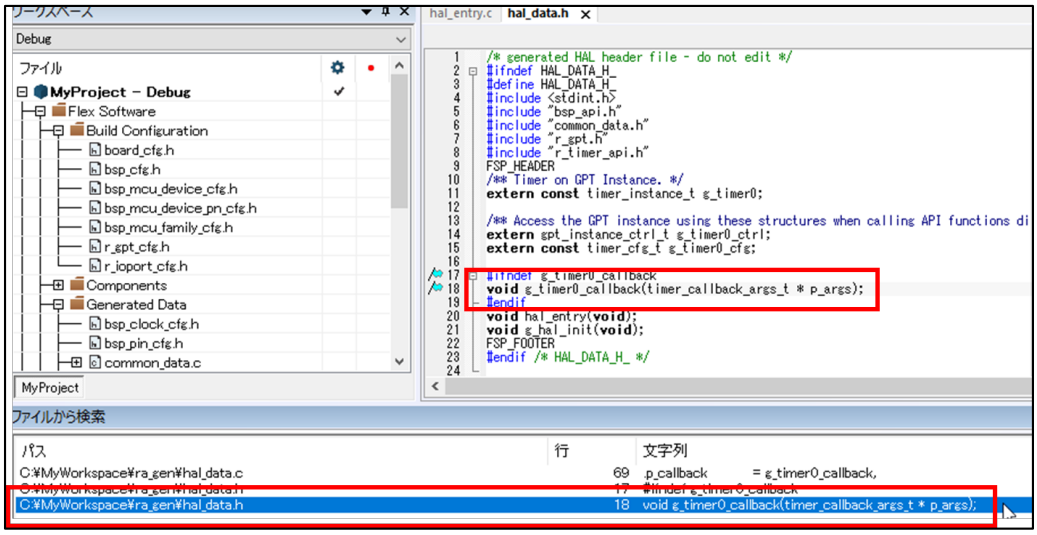
関数の宣言を調べたいときに便利な機能が**編集 > 検索と置換 > ファイルから検索(CTRL + SHIFT + F)**です。



関数名や変数名を入れて検索するとプロジェクト内の一致する記述を一覧表示できます。



hal_data.h で宣言されていることがわかりました。



割り込みによっては p_args には、パラメータを受け取ることができるのですが、今回はパラメータは不要なので、NULL チェックだけを行います。

```
uint32_t g_timer_overflow_flg = 0;
void g_timer0_callback (timer_callback_args_t * p_args)
{
    if (NULL != p_args)
    {
      g_timer_overflow_flg = 1;
    }
}
```

また、このコールバック関数が実行された、つまり、タイマー割り込みが発生したことを示すグローバルフラグを定義し、関数内で 1 に変えています。

10.6. タイマー開始コードの記述

タイマーモジュールは自動的には開始しないので、ユーザコードで記述する必要があります。

エラー検出も含めたコードは以下です。

```
fsp_err_t err  = FSP_SUCCESS;

//err = R_GPT_Open(&g_timer0_ctrl, &g_timer0_cfg);
```

```
err = g_timer0.p_api->open(&g_timer0_ctrl, &g_timer0_cfg);
if (FSP_SUCCESS != err)
{
    while(1);
}

//err = R_GPT_Start(&g_timer0_ctrl);
err = g_timer0.p_api->start(&g_timer0_ctrl);
if (FSP_SUCCESS != err)
{
    while(1);
}
```

このコードを hal_entry 関数内、while(1)の前で記述します。

なお、コメントアウトをした R_GPT_xxx 関数でも動作はします

タイマー割り込みフラグを使って LED 点滅を実装したユーザコード全体が以下です。

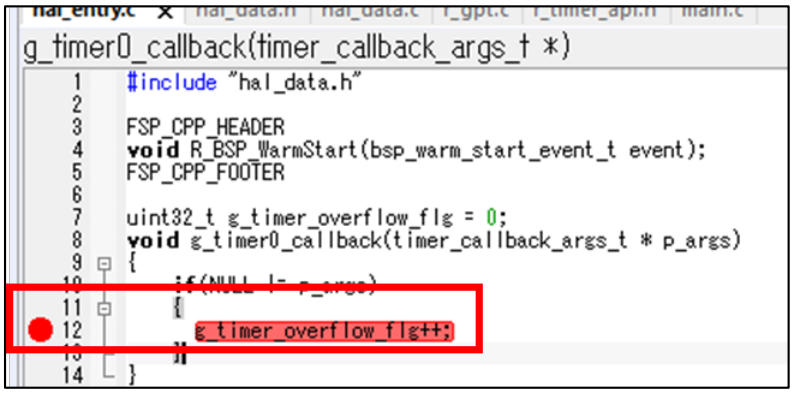
```
void hal_entry(void)
{
    static int count;

    bsp_io_level_t pin_level = BSP_IO_LEVEL_LOW;
    uint32_t pin = BSP_IO_PORT_04_PIN_03;

    fsp_err_t err  = FSP_SUCCESS;
    err = g_timer0.p_api->open(&g_timer0_ctrl, &g_timer0_cfg);
    if (FSP_SUCCESS != err)
    {
      while(1);
    }

    err = g_timer0.p_api->start(&g_timer0_ctrl);

    if (FSP_SUCCESS != err)
    {
      while(1);
    }

    while(1)
    {
      R_BSP_PinAccessEnable();
      R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);
      R_BSP_PinAccessDisable();

      if (BSP_IO_LEVEL_LOW == pin_level)
      {
        pin_level = BSP_IO_LEVEL_HIGH;
      }
      else
      {
        pin_level = BSP_IO_LEVEL_LOW;
      }

      while (g_timer_overflow_flg == 0);
      g_timer_overflow_flg = 0;
      count++;
    }
}
```

これでデバッグを行うとユーザ LED1 が 100m 秒周期で点滅します。

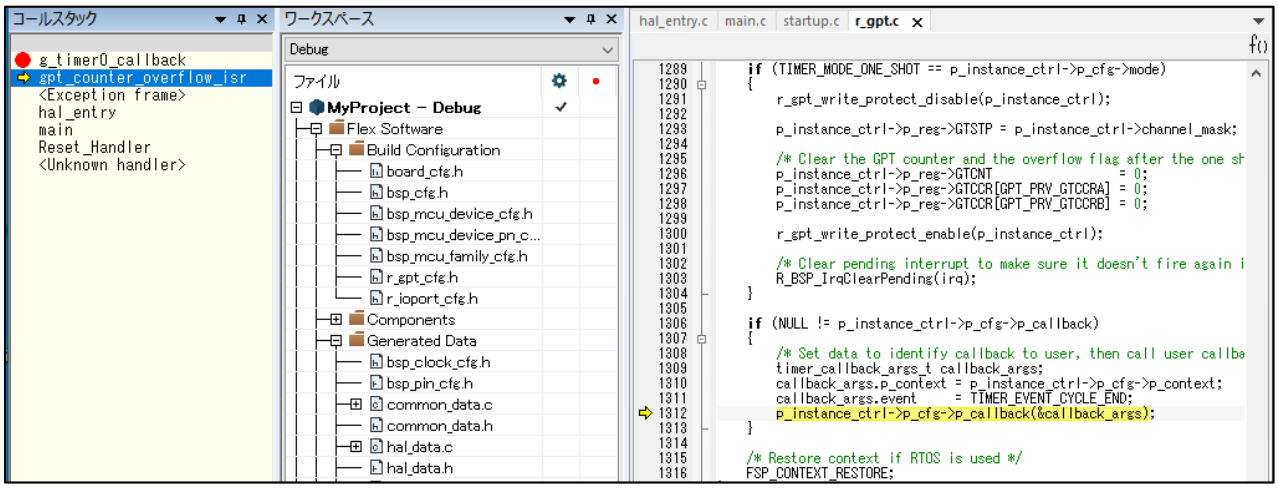
10.7. タイマー割り込みコールバック関数のデバッグ

設定したコールバック関数がどのように呼ばれているのかデバッグをします。

フラグ変数を設定している箇所にブレークポイントを設定して実行します。



ブレークポイントで止まったら、**表示>コールスタック**を選択します。



コールスタックウィンドウには、現在の停止箇所までの関数呼び出し階層が表示されます。

g_timer0_callback は割り込みハンドラである gpt_counter_overflow_isr から呼ばれた、割り込みは main 関数から呼ばれた hal_entry 関数を処理中に発生した、ということがわかりました。このコールスタックウィンドウは実行経路を追う、あるいはプロジェクトの構成を調査するときなどにも便利なので、ぜひ活用してみてください。

11. おわりに

本ドキュメントでは、EWARM と RA Smart Configurator を組み合わせて、RA6M3 評価ボード用で動作するプロジェクトの作成手順と EWARM の開発・デバッグ機能を紹介しました。

ルネサスエレクトロニクス社の RA ファミリは非常に高性能かつ多機能です。RA Smart Configurator を使用し GUI 上で必要な機能を設定することで、ユーザコードでは簡単な API 関数コールでモジュールを使用することができます。

今回は GPIO のポート制御とタイマー割り込みの使用方法について説明しました。他のモジュールについても同様の手順で実装をすることができます。