

**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

ルネサスエレクトロニクス製 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、組み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、評価ボード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。

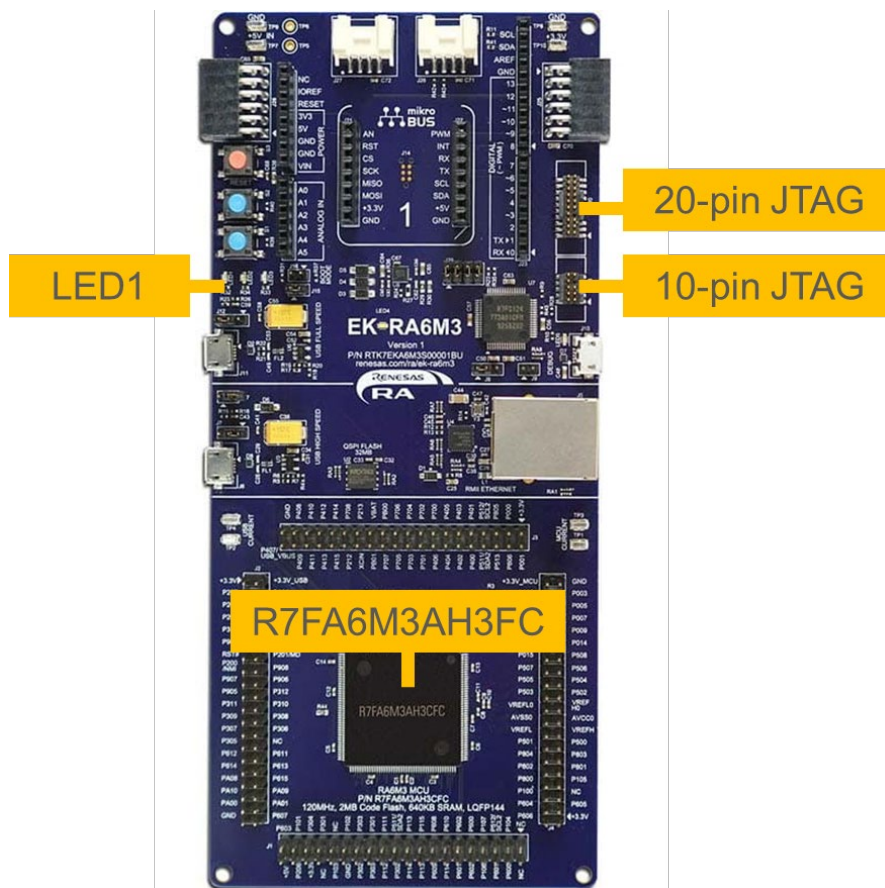
3. 本スタートアップガイドの目的

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

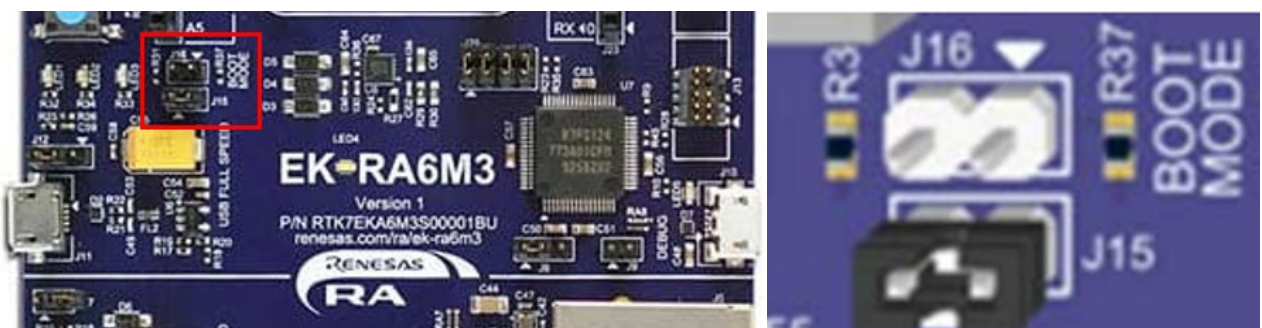
EK-RA6M3 評価キットには、評価ボード本体とマイクロ USB ケーブルが同梱されています。



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

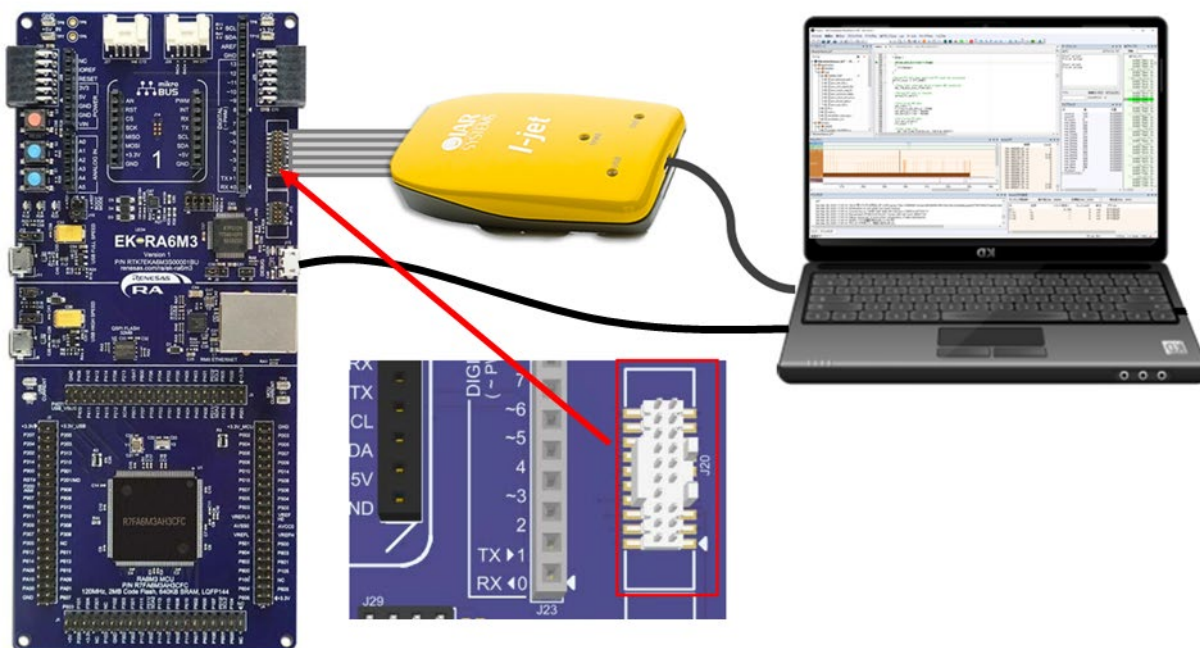
BOOT CONFIG が INTERNAL FLASH の向きでクローズとなっていることを確認してください。



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**

ヘッドでは 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。

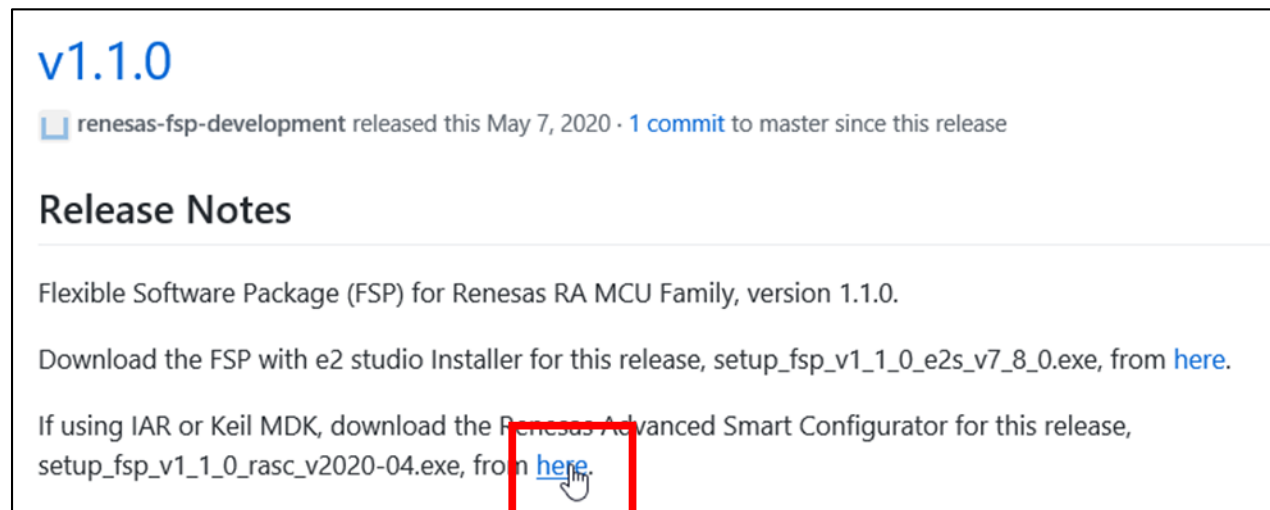


5. RA Smart Configurator のインストール

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>



v1.1.0

renesas-fsp-development released this May 7, 2020 · 1 commit to master since this release

Release Notes

Flexible Software Package (FSP) for Renesas RA MCU Family, version 1.1.0.

Download the FSP with e2 studio Installer for this release, `setup_fsp_v1_1_0_e2s_v7_8_0.exe`, from [here](#).

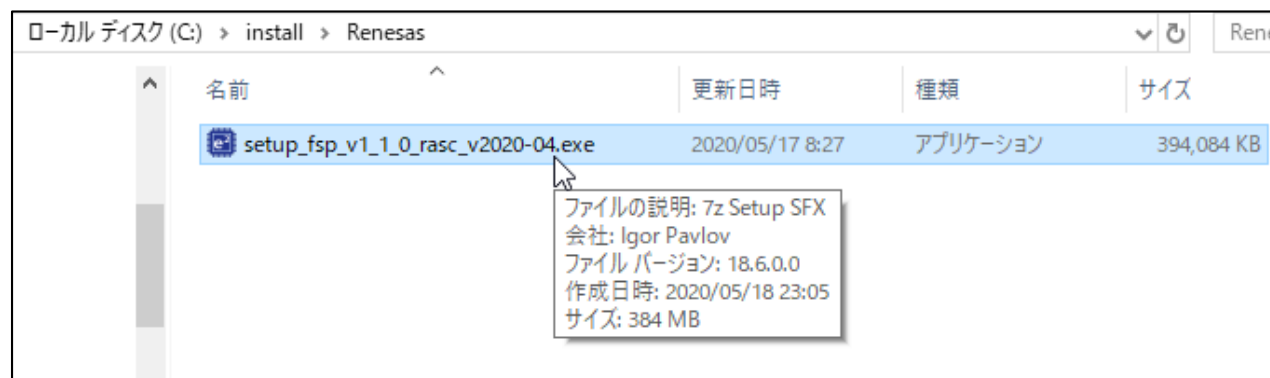
If using IAR or Keil MDK, download the Renesas Advanced Smart Configurator for this release, `setup_fsp_v1_1_0_rasc_v2020-04.exe`, from [here](#).

「`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

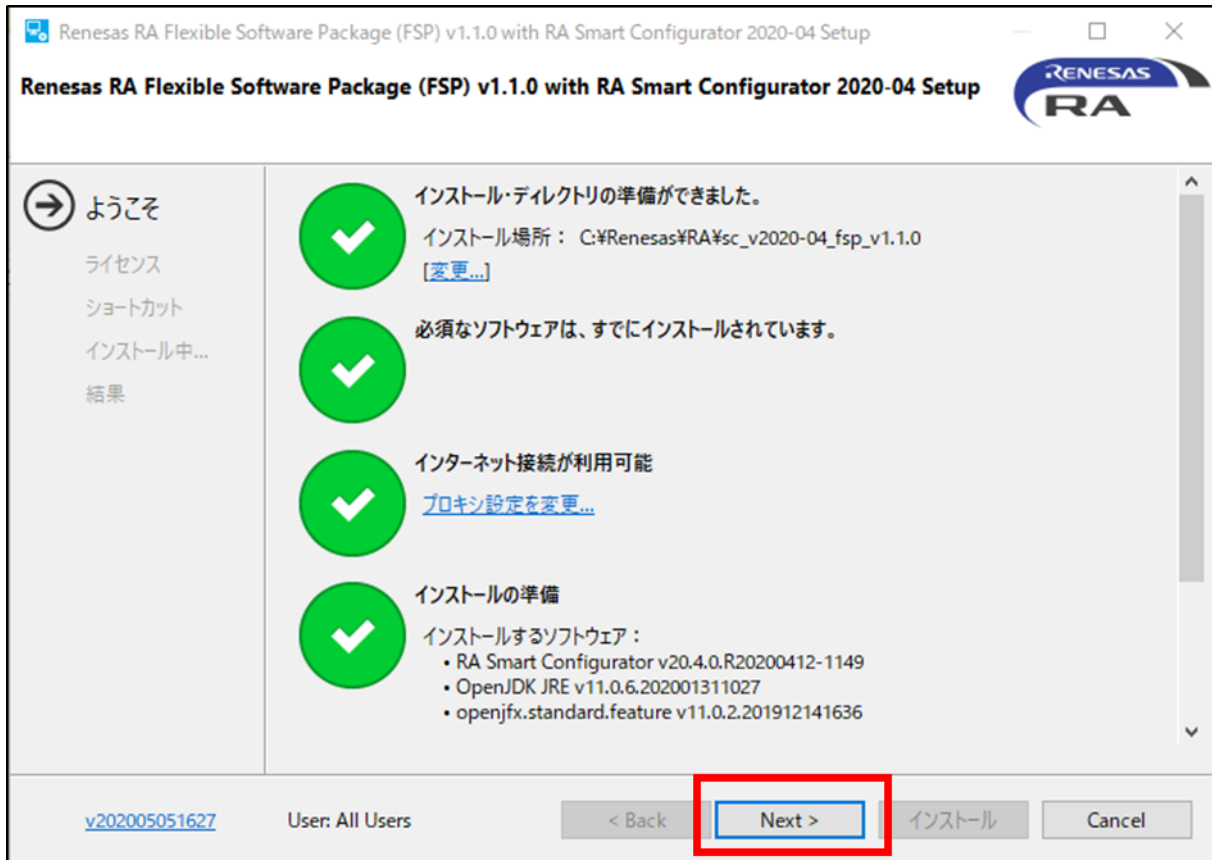


名前	更新日時	種類	サイズ
setup_fsp_v1_1_0_rasc_v2020-04.exe	2020/05/17 8:27	アプリケーション	394,084 KB

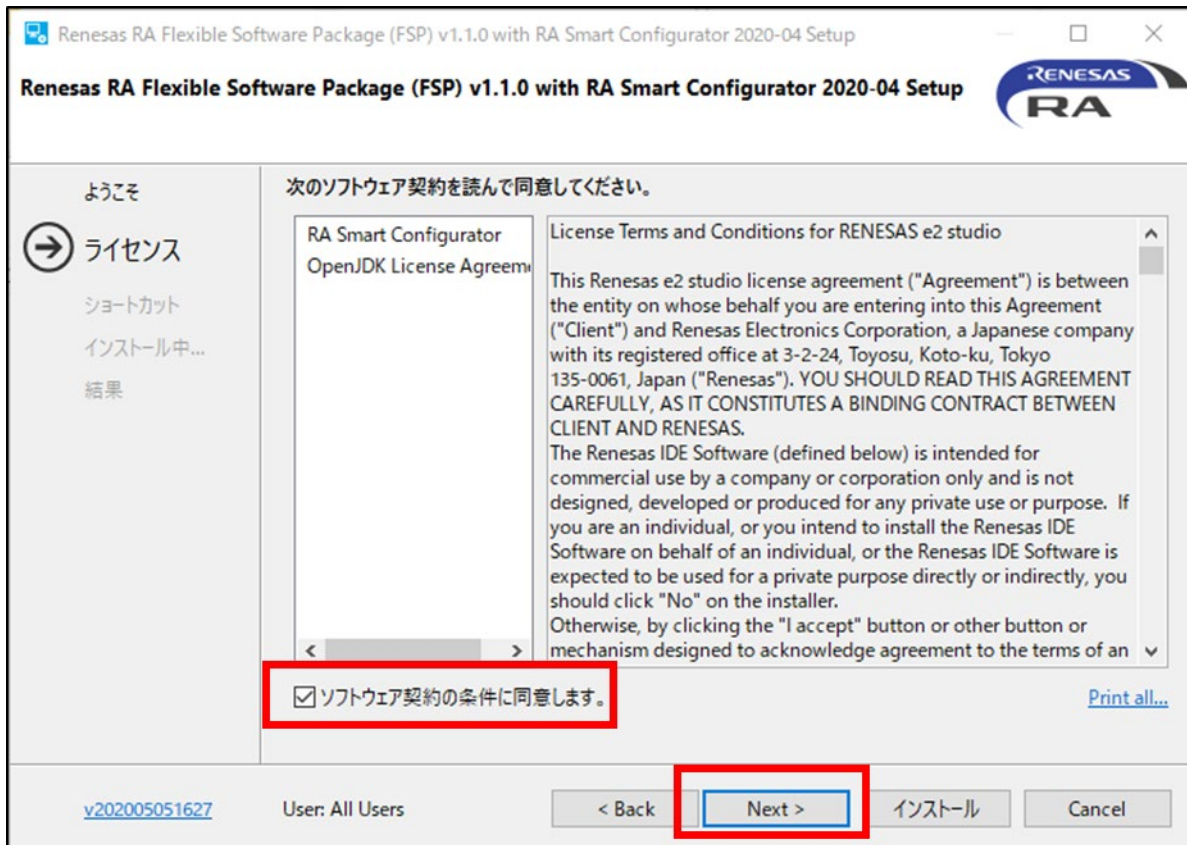
ファイルの説明: 7z Setup SFX
会社: Igor Pavlov
ファイルバージョン: 18.6.0.0
作成日時: 2020/05/18 23:05
サイズ: 384 MB

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。

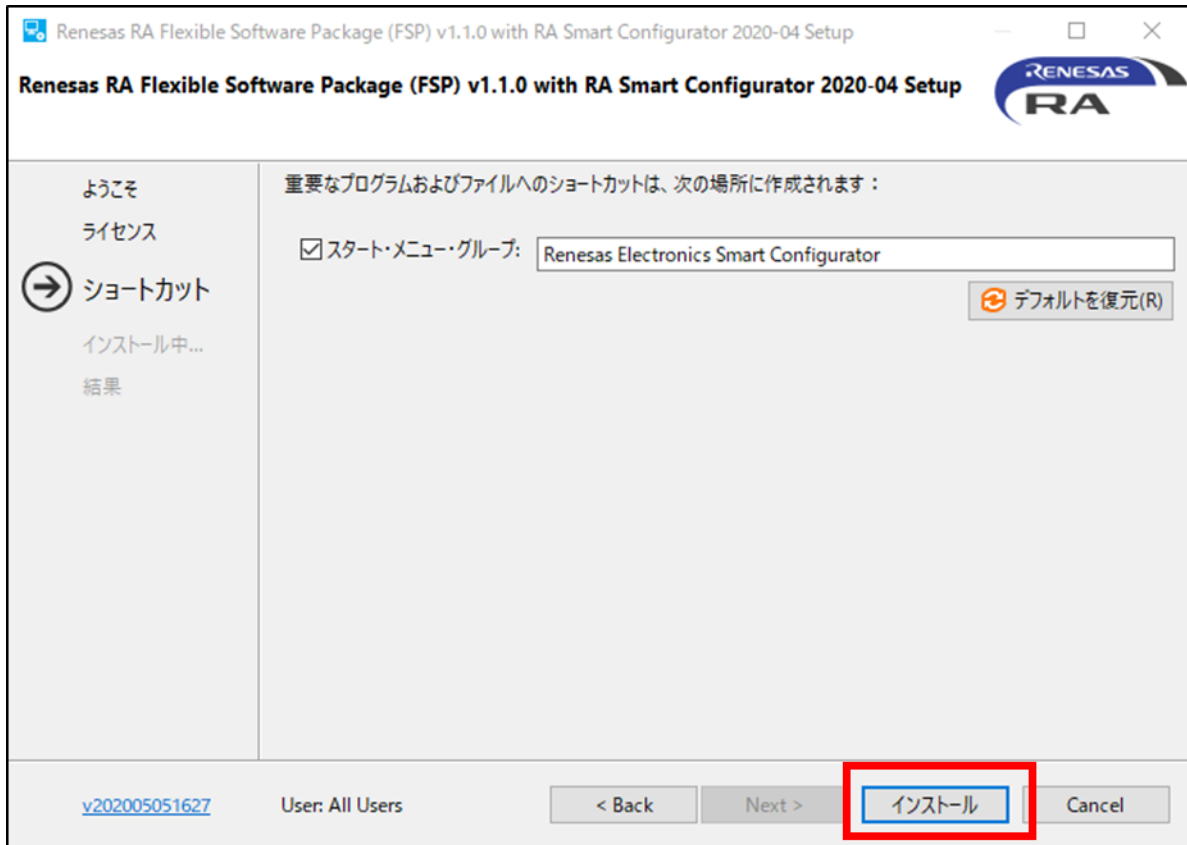
ホストコンピュータの状況によっては、Visual Studio のランタイムライブラリの追加インストールが要求されることがあります。



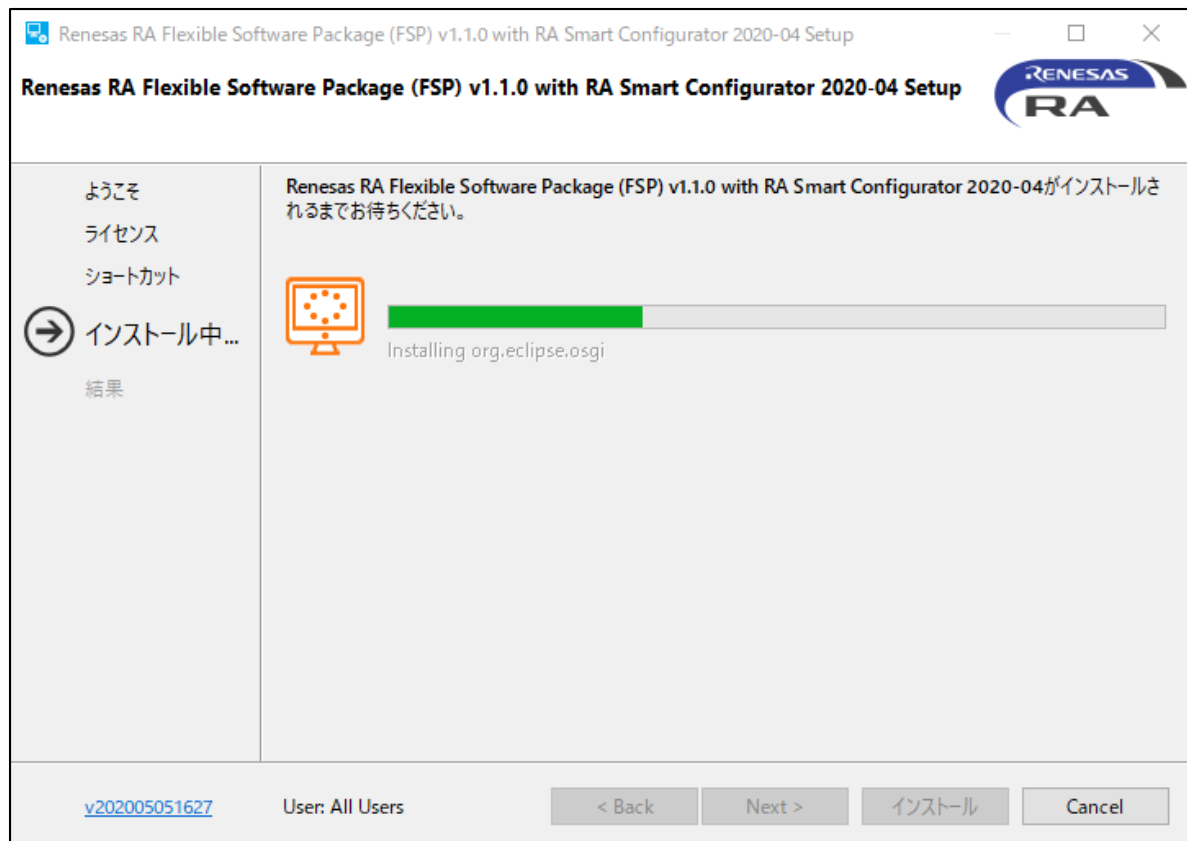
Next をクリックしてください。



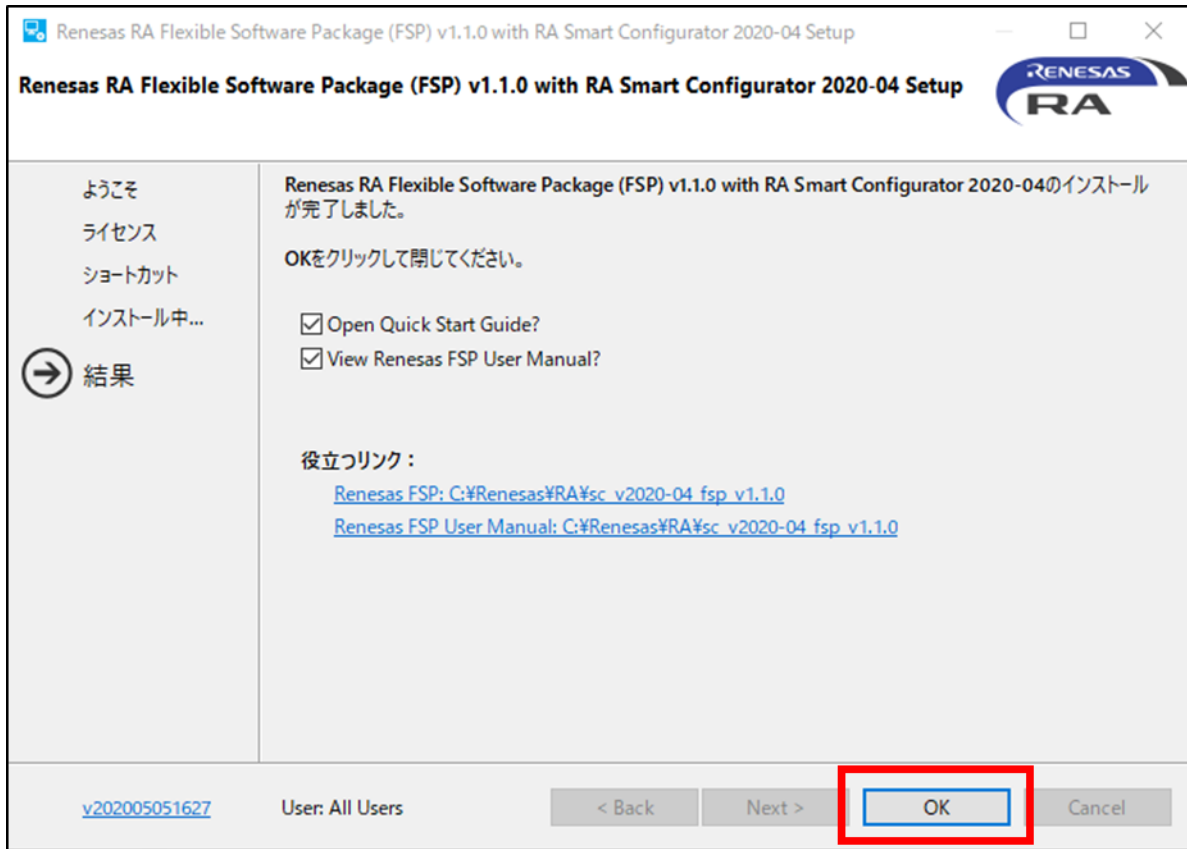
ソフトウェア契約の条件を読んで同意にチェック後 **Next** をクリックしてください。



インストール をクリックしてください。



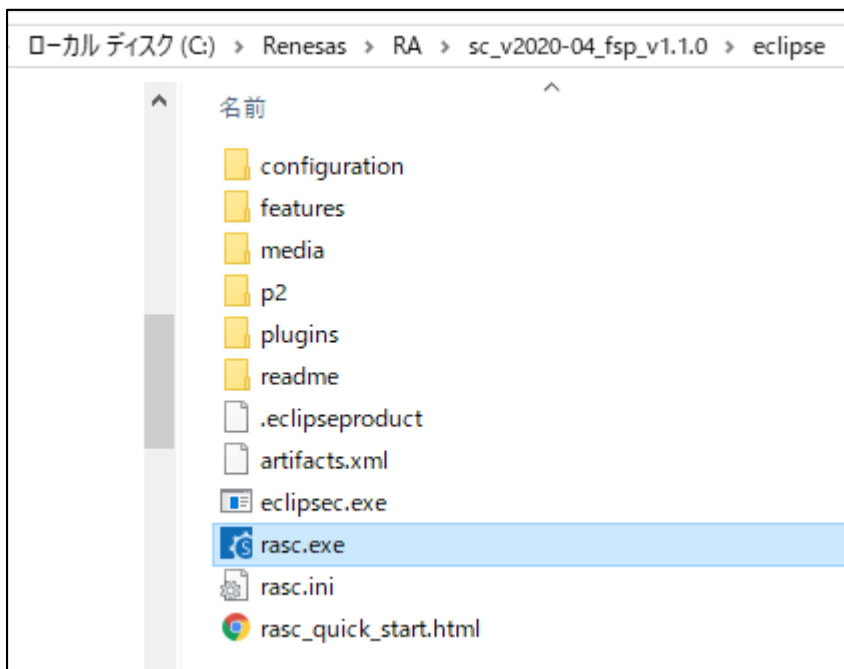
インストールが終了するのを待ちます。



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

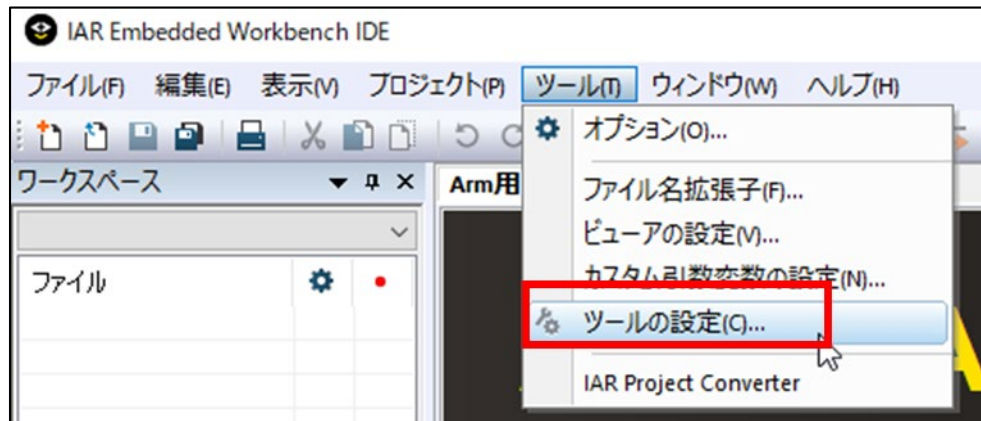
EWARM のメニューから RA Smart Configurator を起動し、プロジェクトと連携できるように設定を行います。

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。

ツールの設定

メニュー内容(M):

メニューテキスト(T):

コマンド(C):

引数(A):

初期ディレクトリ(D):

出力ウィンドウにリダイレクト(O)

コマンドラインのプロンプト(P)

ツール使用可能時(V):

常時

OK

キャンセル

新規作成(N)

削除(D)

参照(B)...

各項目にパラメータを指定します。

ツールの設定

メニュー内容(M):

RA Smart Configurator

メニューテキスト(T):

RA Smart Configurator

コマンド(C):

%RA%sc_v2020-04_fsp_v1.1.0#eclipse#rasc.exe

引数(A):

--compiler IAR configuration.xml

初期ディレクトリ(D):

\$PROJ_DIR\$

出力ウィンドウにリダイレクト(O)

コマンドラインのプロンプト(P)

ツール使用可能時(V):

常時

OK

キャンセル

新規作成(N)

削除(D)

参照(B)...

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。

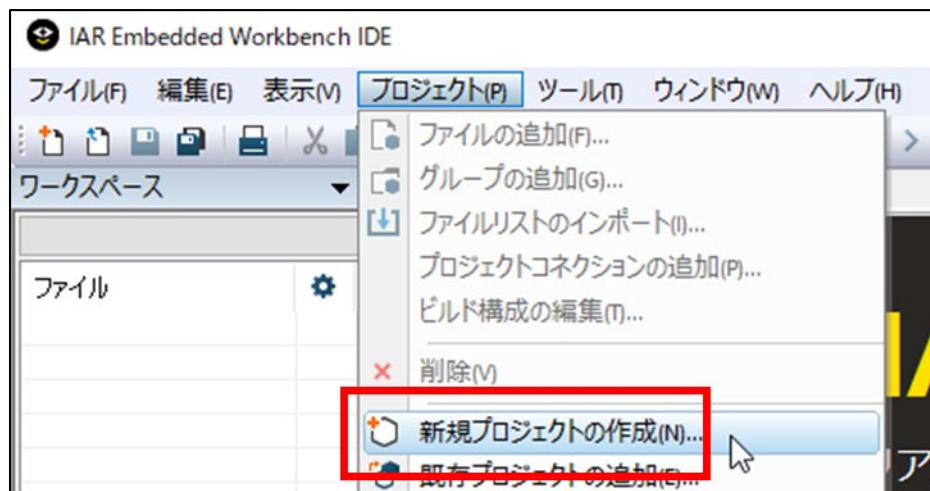
この手順を繰り返していくことで、アプリケーションを完成させていきます。

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

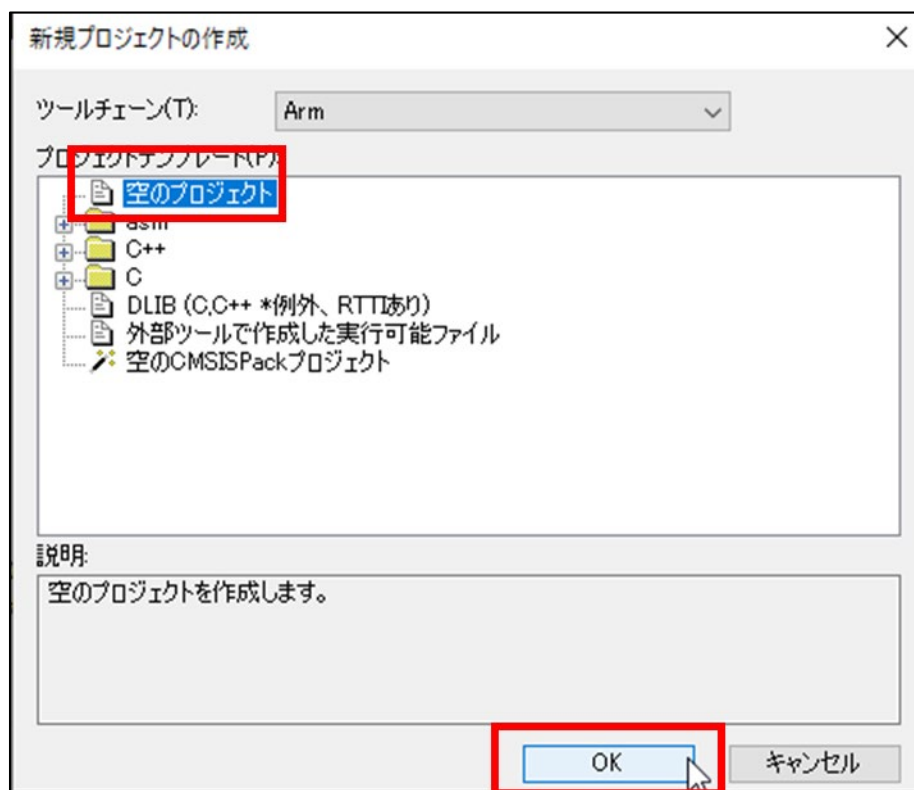
上記ワークフローに従ってプロジェクトを作成していきます。

8.1. EWARM で新規ブランクプロジェクトを作成

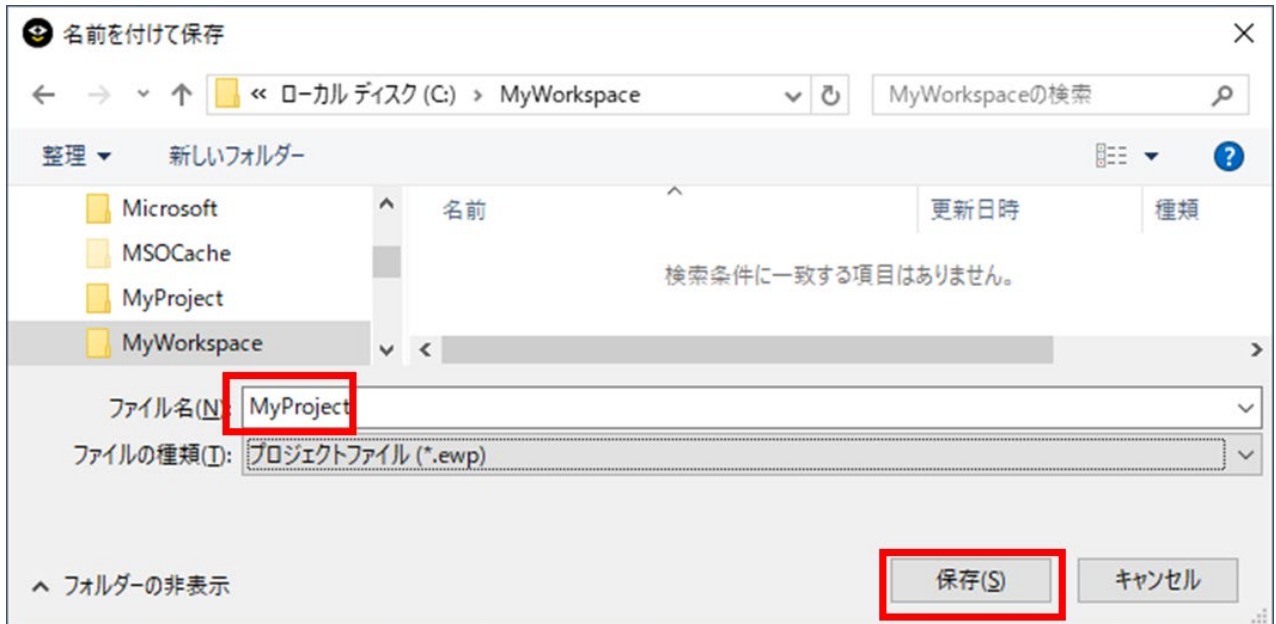
プロジェクトメニューから新規プロジェクトの作成を選択します。



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。



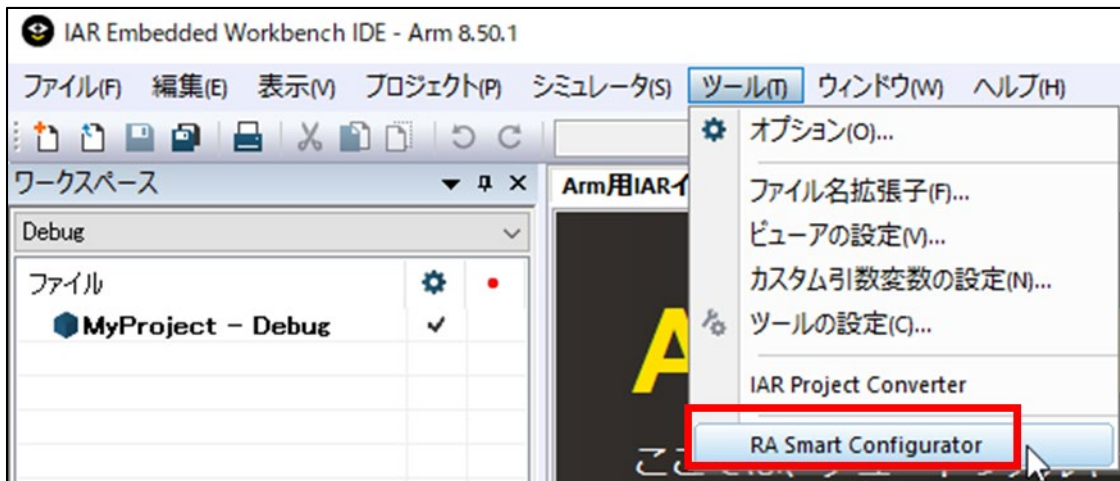
新規プロジェクトが作成されました。

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

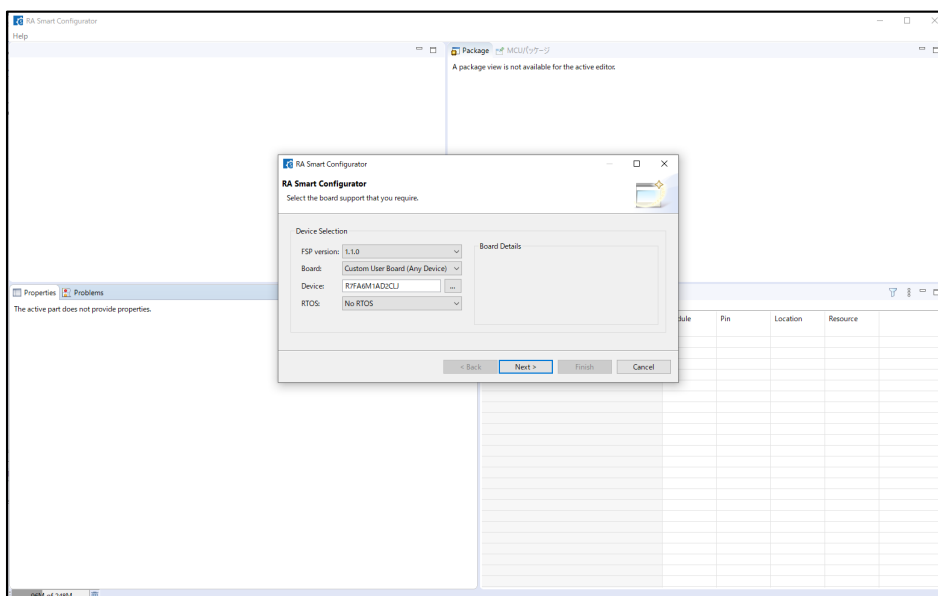
メニューバーの **ツール** から先ほど登録をした **RA Smart Configurator** を起動します。



スタートアップ画面が表示され、



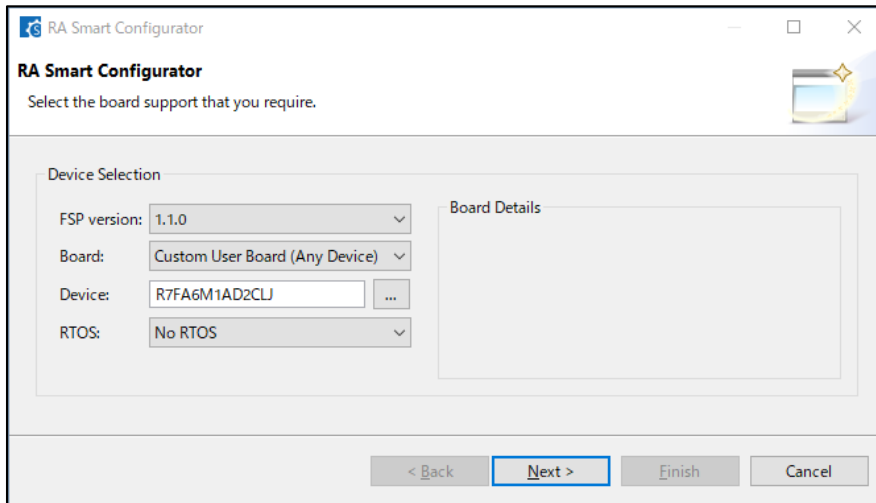
初期ウィザードが表示されます。



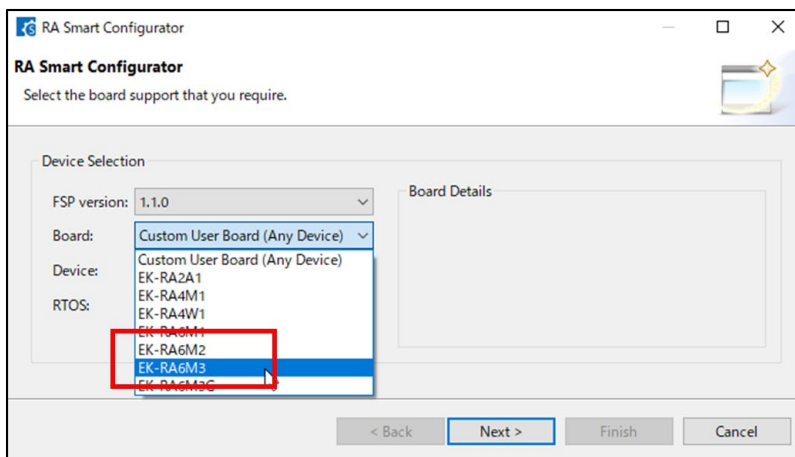
起動しない場合には、**ツール > ツールの設定** パラメータを確認してください。

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

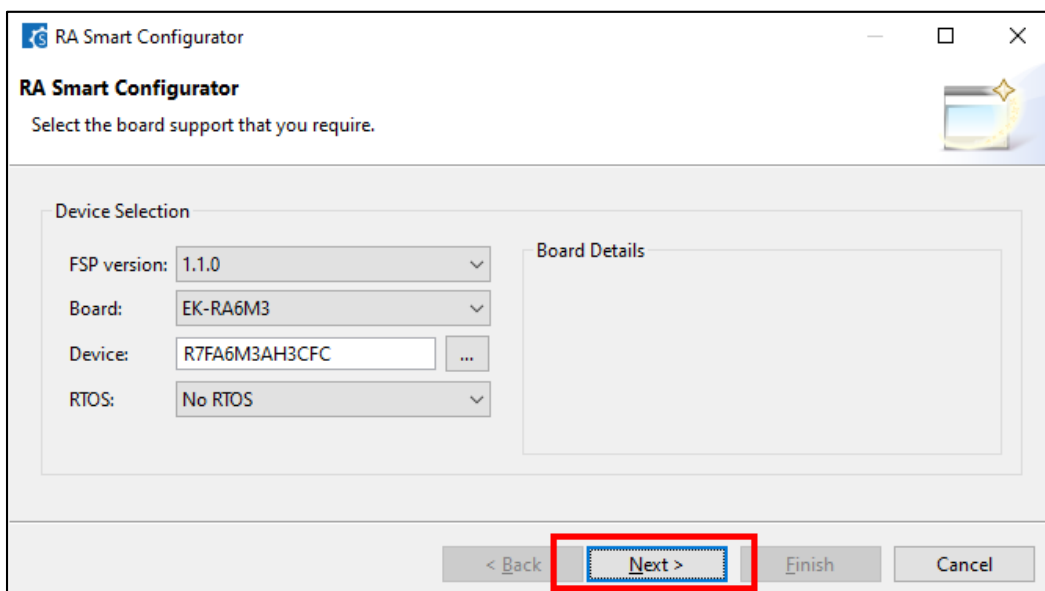
最初に、使用するボードを評価ボードか、あるいはカスタムボードか選択します。評価ボードを選択すると、自動的に Device も選択されます。



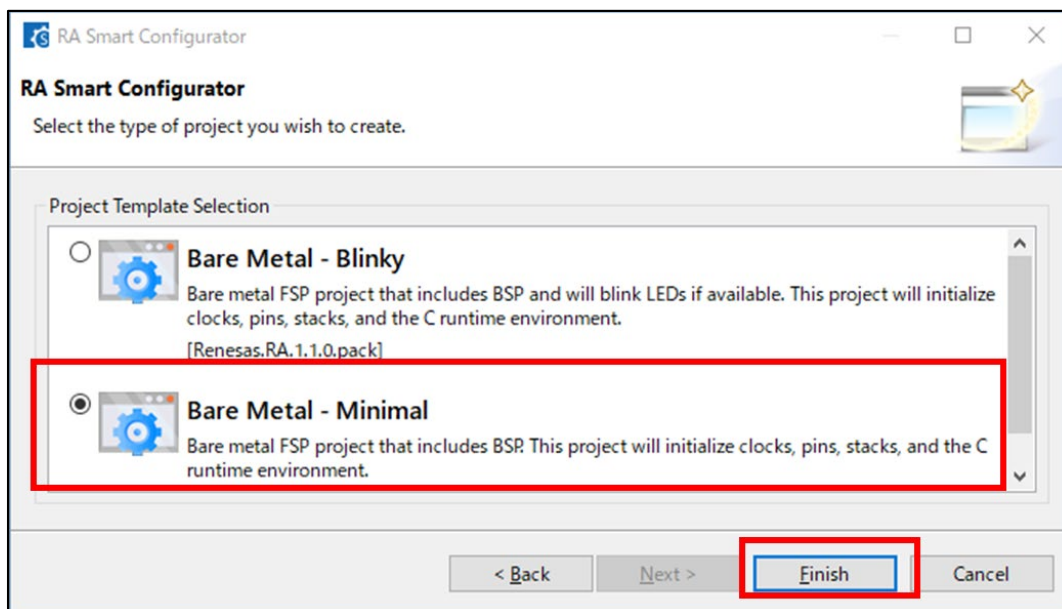
Board に使用する評価ボード **EK-RA6M3** を指定します。



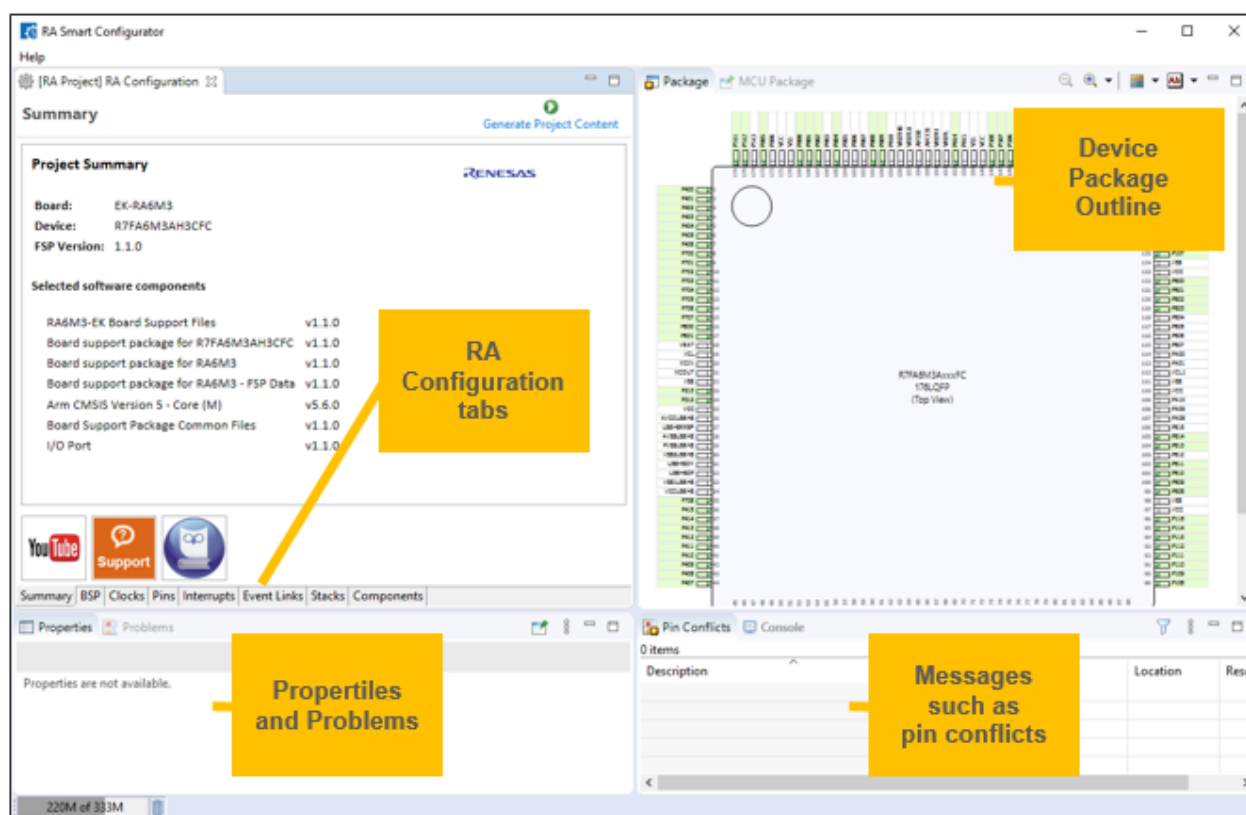
Device も R7FA6M1AD3CFP に切り替わります。本ドキュメントでは RTOS は使用しないので **No RTOS** のままにして **Next** をクリックします。



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。

The screenshot shows the 'Summary' window in the RA Configuration tool. The title bar reads '[RA Project] RA Configuration'. The main content area is titled 'Project Summary' and includes the Renesas logo. Below the logo, the following information is displayed:

- Board: EK-RA6M3
- Device: R7FA6M3AH3CFC
- FSP Version: 1.1.0

Under the heading 'Selected software components', a list of components and their versions is shown:

Component	Version
RA6M3-EK Board Support Files	v1.1.0
Board support package for R7FA6M3AH3CFC	v1.1.0
Board support package for RA6M3	v1.1.0
Board support package for RA6M3 - FSP Data	v1.1.0
Arm CMSIS Version 5 - Core (M)	v5.6.0
Board Support Package Common Files	v1.1.0
I/O Port	v1.1.0

At the bottom of the window, there are icons for YouTube, Support, and a help icon. A navigation bar at the very bottom contains the following tabs: Summary, BSP, Clocks, Pins, Interrupts, Event Links, Stacks, and Components.

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。

The screenshot shows the 'Board Support Package Configuration' window in the RA Configuration tool. The title bar reads '[RA Project] RA Configuration'. The main content area is titled 'Board Support Package Configuration' and includes the 'Generate Project Content' button and a 'Restore Defaults' button. The 'Device Selection' section contains the following fields:

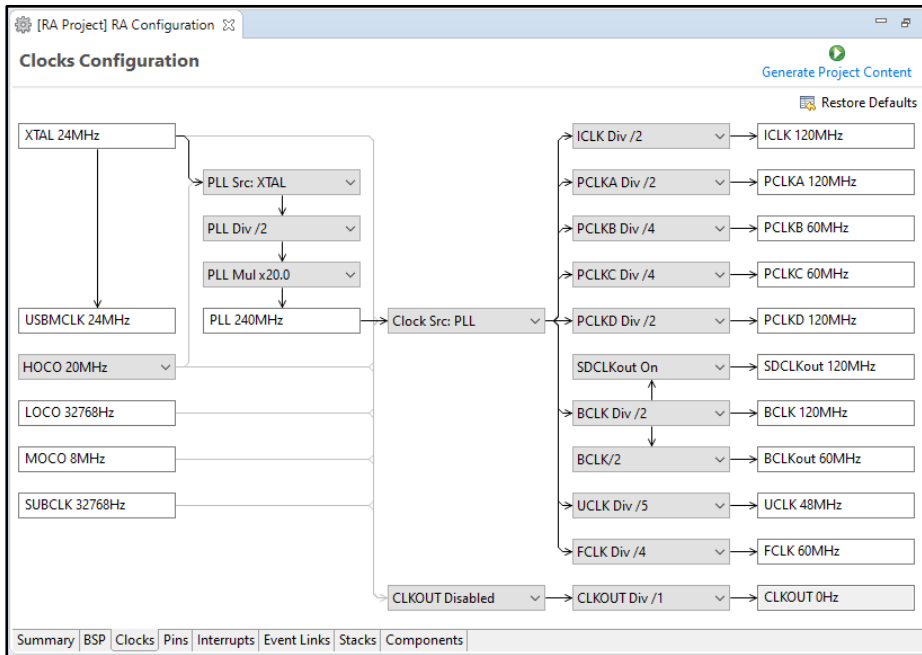
- FSP version: 1.1.0
- Board: EK-RA6M3
- Device: R7FA6M3AH3CFC
- RTOS: No RTOS

The 'Board Details' section is currently empty. Below the configuration window, the 'Properties' window is open, showing the 'EK-RA6M3' properties. The table below lists the properties and their values:

Property	Value
▼ R7FA6M3AH3CFC	
part_number	R7FA6M3AH3CFC
rom_size_bytes	2097152
ram_size_bytes	655360
data_flash_size_bytes	65536
package_style	LQFP
package_pins	176
▼ RA6M3	
series	6

Clocks

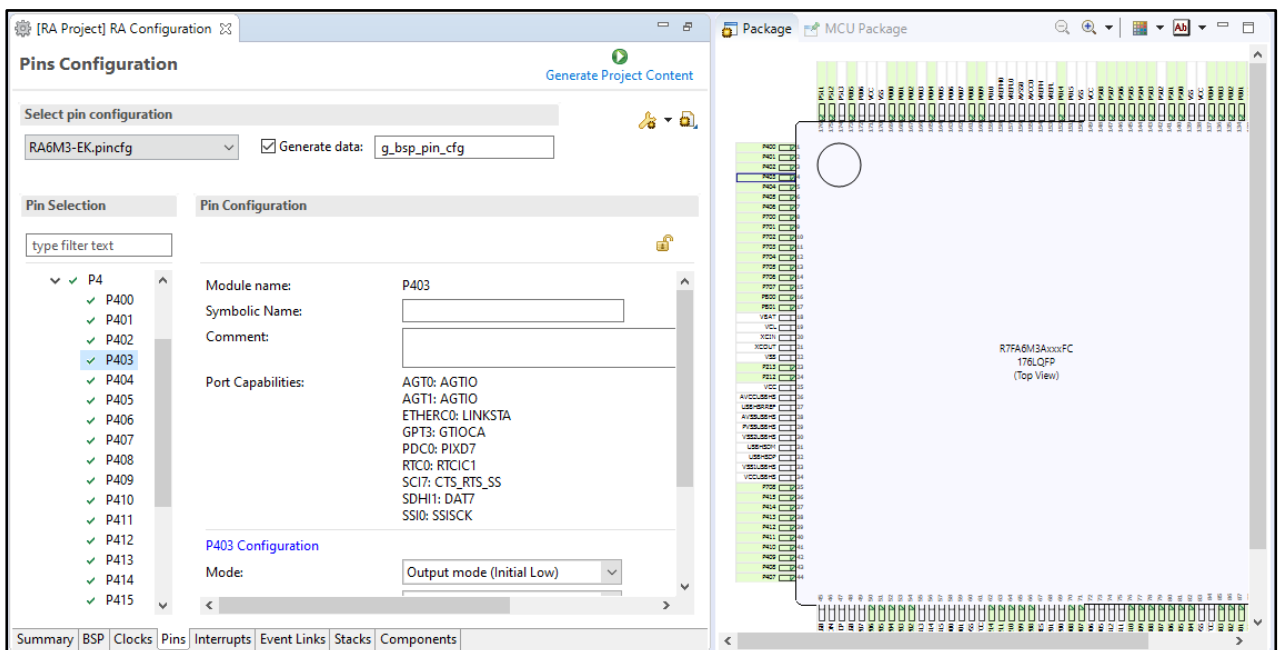
システム全体のクロック構成の確認・変更をすることができます。マイコン内ではコンポーネントごとに異なる分周のクロックを供給する必要がありますが、この画面を使って視覚的に設定を行うことができます。



この設定では外部の 24MHz のクリスタルから PLL で通倍したクロックを各モジュールに供給しています。CPU のメインクロックが一番上にある ICLK です。

Pins

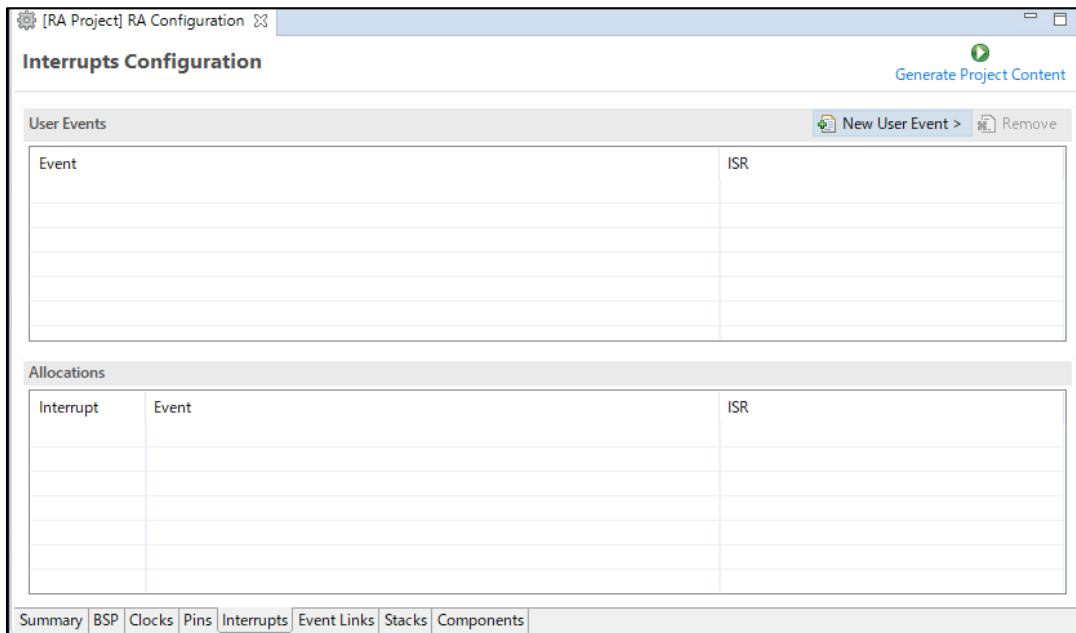
MCU の各ピンの割り当てを設定・確認することができます。



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。

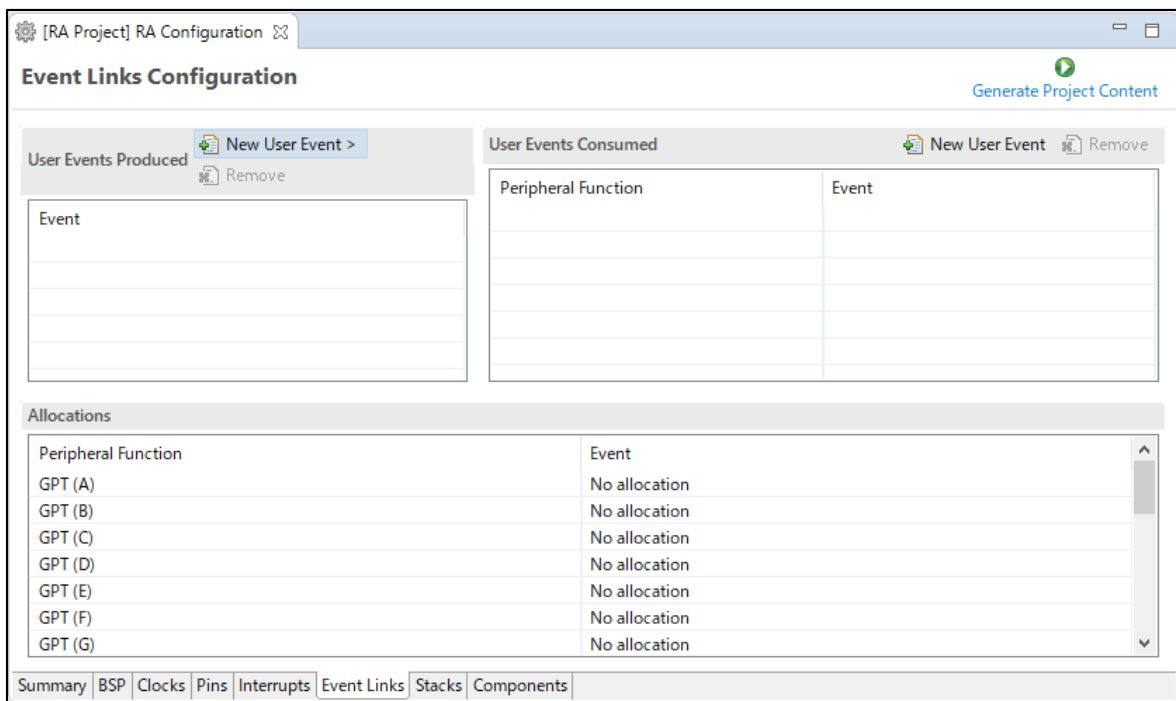
Interrupt Configuration

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。



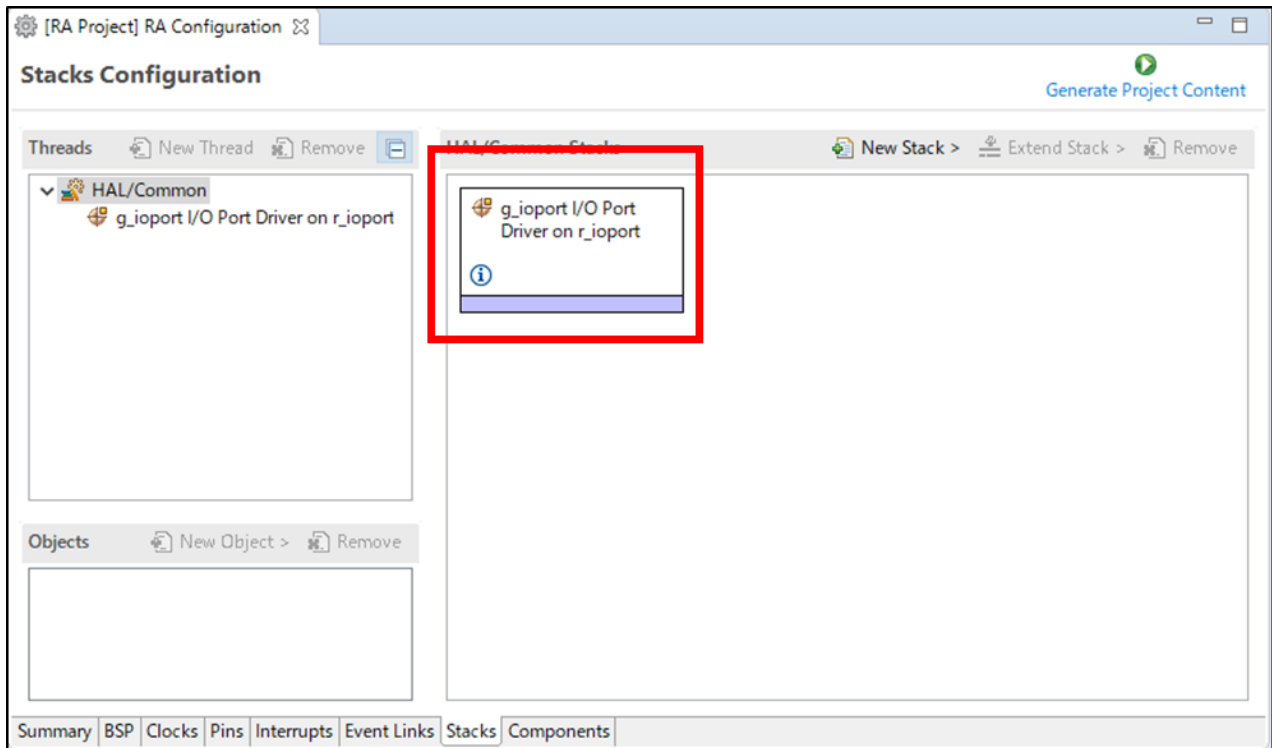
Event Links

RA ファミリー独自の 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

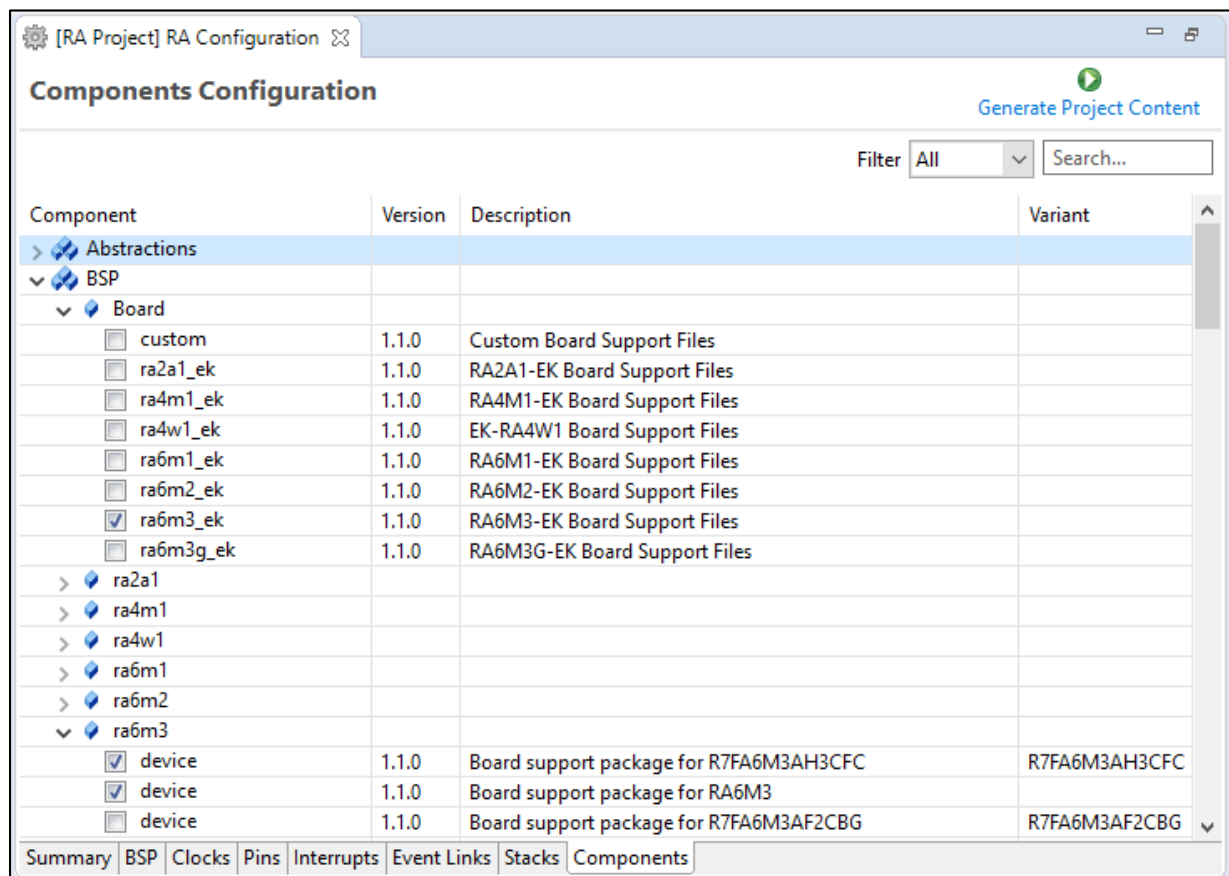
ソフトウェアブロックを Stack というモジュールで配置をすることができます。FSP および RA Smart Configurator の非常に独特なコンセプトです。スタックによっては親子関係を持つこともあり、必要な (依存する) スタックが配置されていない場合にはエラーが表示されます。スタックにはペリフェラルのドライバのほか、ミドルウェアなどの大規模なコンポーネントもあります。



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

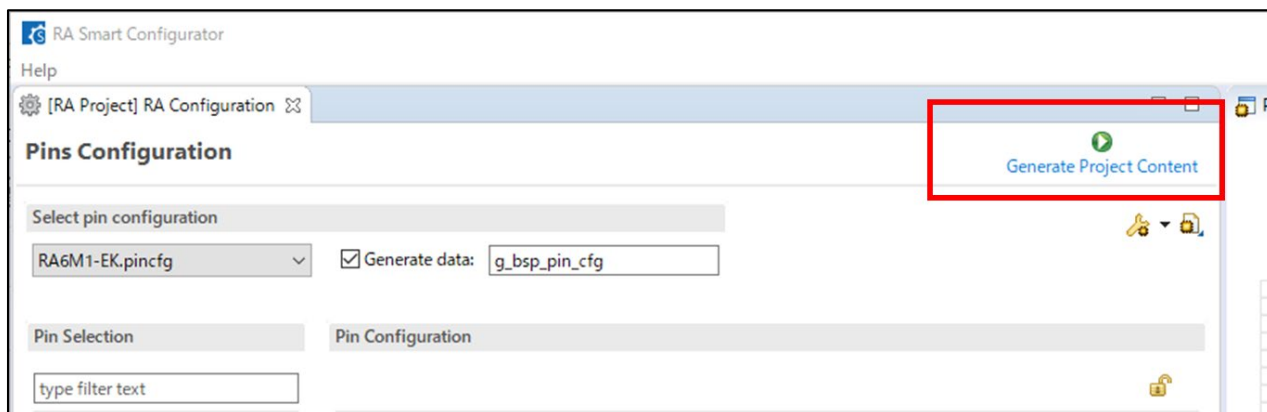
これまでの設定によってプロジェクトに取り込まれる（コード生成される）ファイルが確認できます。基本的にはこの画面は確認用で、手動でチェックを追加することはありません。



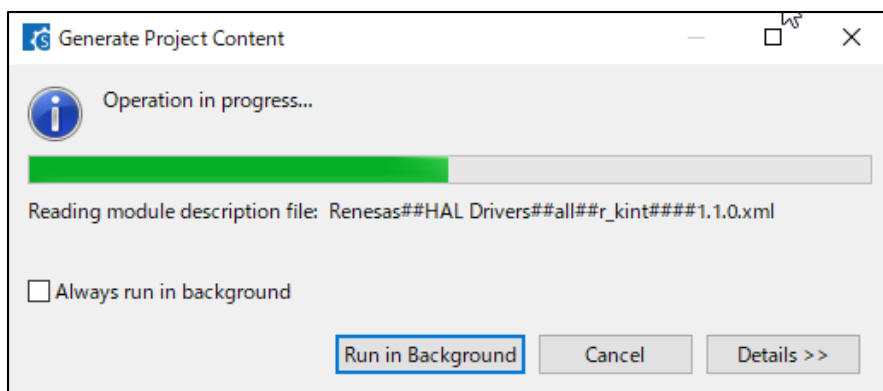
EK-RA6M3 用の最低限の設定は既に行われていることがわかりました。まずはこの状態でプロジェクトを生成します。

8.4. RA Smart Configurator でファイルを生成

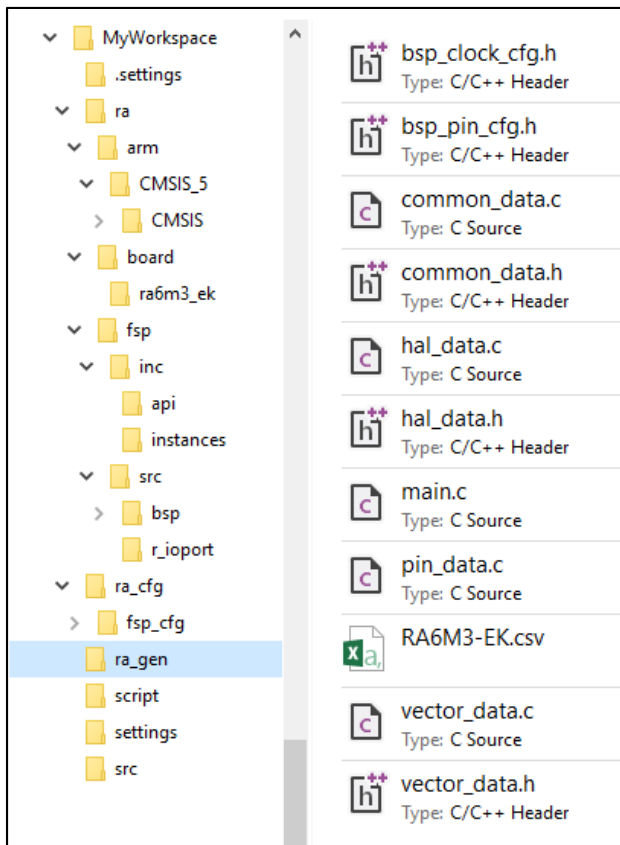
RA Smart Configurator で設定が終わったらファイルの生成を行います。各種エラーがないことを確認したら、**Generate Project Content** ボタンをクリックします。



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。



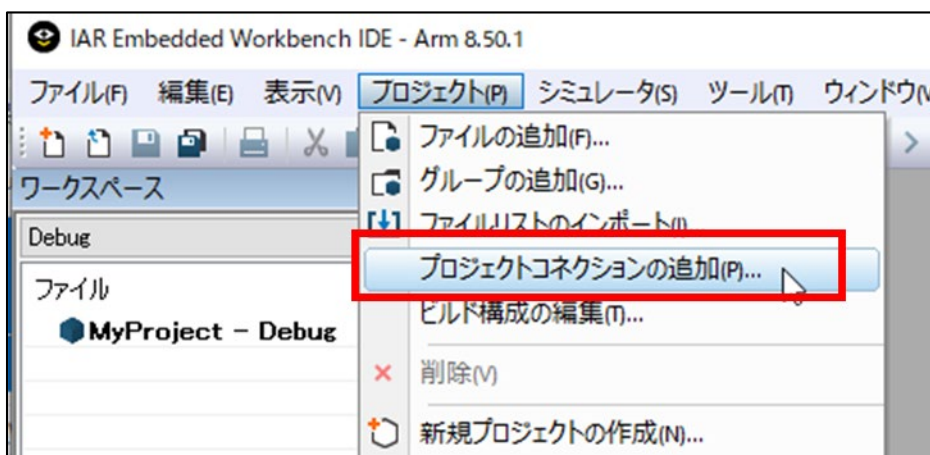
RA Smart Configurator で設定を変更して、Generate をするごとにこれらのファイルが更新されます。

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

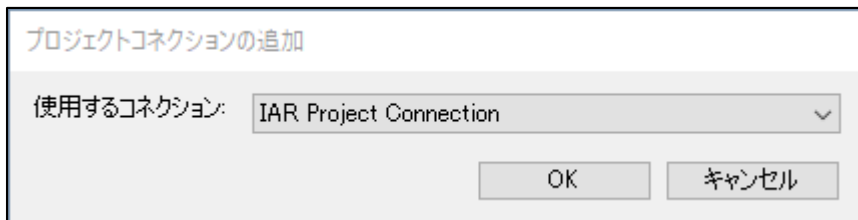
EWARM ではフォルダ内のファイルを自動で取り込む動作はしません。プロジェクトコネクションという仕組みを使用して、RA Smart Configurator の生成ファイル群と連携します。

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。

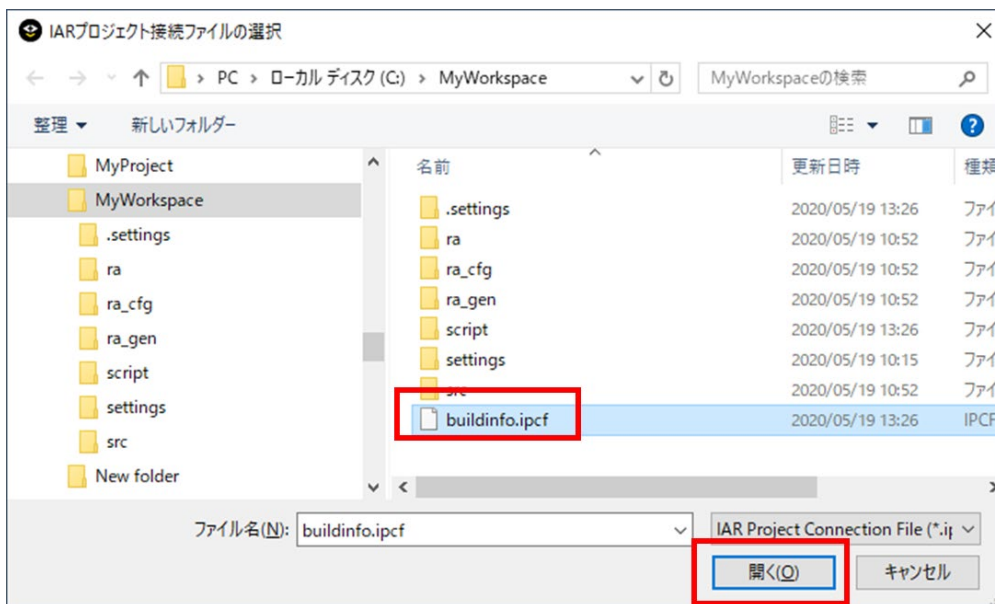
EWARM の画面に戻り、**プロジェクト > プロジェクトコネクションの追加** を選択してください。



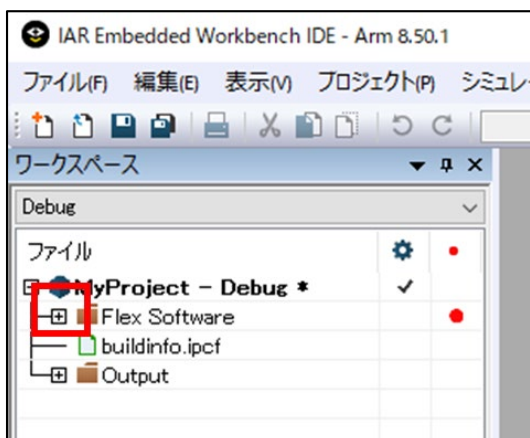
IAR Project Connection を選択し **OK** をクリックしてください。



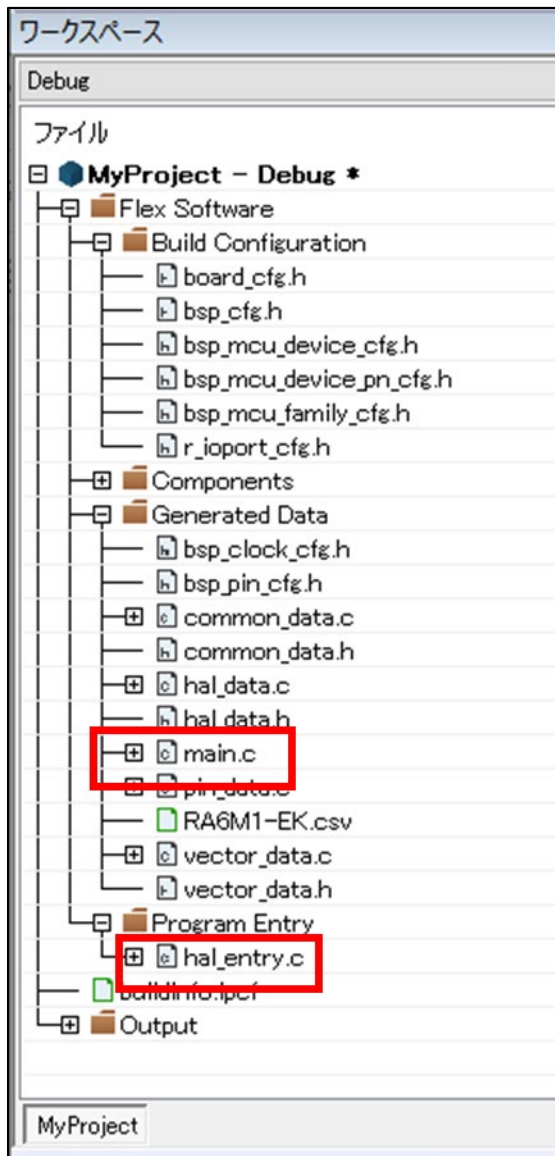
buildinfo.ipcf ファイルを選択して、開く を実行してください。



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。

```

main.c x
1  /* generated main source file - do not edit */
2  #include "hal_data.h"
3  int main(void) {
4      hal_entry();
5      return 0;
6  }
7

hal_entry.c x
1  #include "hal_data.h"
2
3  FSP_CPP_HEADER
4  void R_BSP_WarmStart(bsp_warm_start_event_t event);
5  FSP_CPP_FOOTER
6
7  /* main() is generated by the RA Configuration editor and is used to generate threads if an RTOS is used. This function
8   * is called by main() when no RTOS is used.
9   */
10 void hal_entry(void)
11 {
12     /* TODO: add your own code here */
13 }
14
15
16 /* This function is called at various points during the startup process. This implementation uses the event that is
17 * called right before main() to set up the pins.
18 */
19
20 /* @param[in] event Where at in the start up process the code is currently at
21 */
22 void R_BSP_WarmStart (bsp_warm_start_event_t event)
23 {
24     if (BSP_WARM_START_RESET == event)
25     {
26         #if BSP_FEATURE_FLASH_LP_VERSION != 0
27             /* Enable reading from data flash. */
28             R_FACI_LP->DFLCTL = 1U;
29
30             /* Would normally have to wait tDSTOP(6us) for data flash recovery. Placing the enable here, before clock and
31              * C runtime initialization, should negate the need for a delay since the initialization will initially take more than 6us.
32              */
33         #endif
34     }
35
36     if (BSP_WARM_START_POST_C == event)
37     {
38         /* C runtime environment and system clocks are setup. */
39
40         /* Configure pins. */
41         R_IOPORT_Open(&g_ioport_ctrl, &g_bsp_pin_cfg);
42     }
43 }

```

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。

```

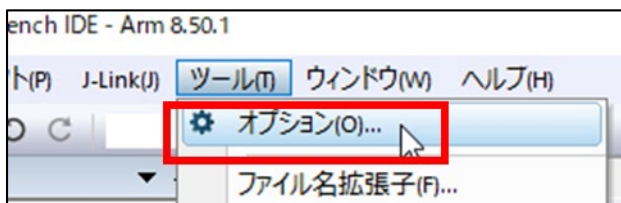
main.c x
1  /* generated main source file - do not edit */
2  #include "hal_data.h"
3  int main(void) {
4      hal_entry();
5      return 0;
6  }
7

```

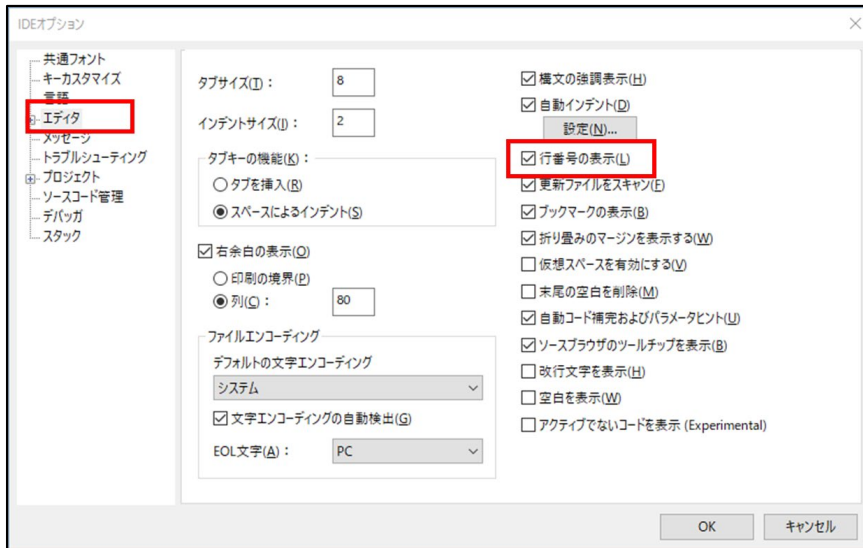
該当のコメントのあるソースコードは、RA Smart Configurator により上書きされる可能性があるので、編集せずに使用してください。

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。



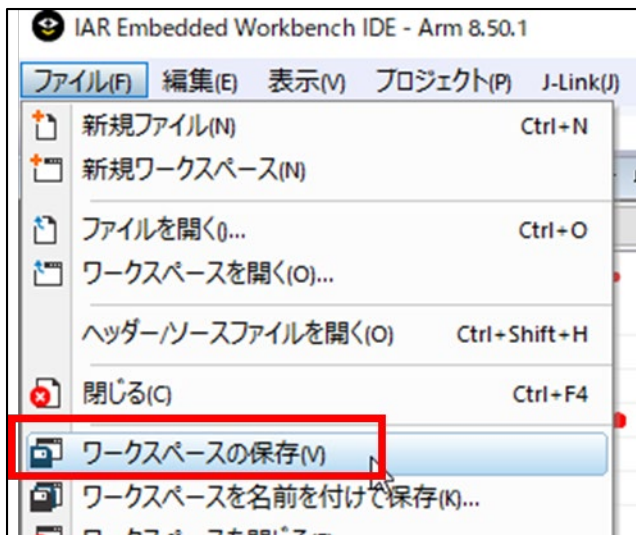
エディタ カテゴリの 行番号の表示 にチェックをつけてください。



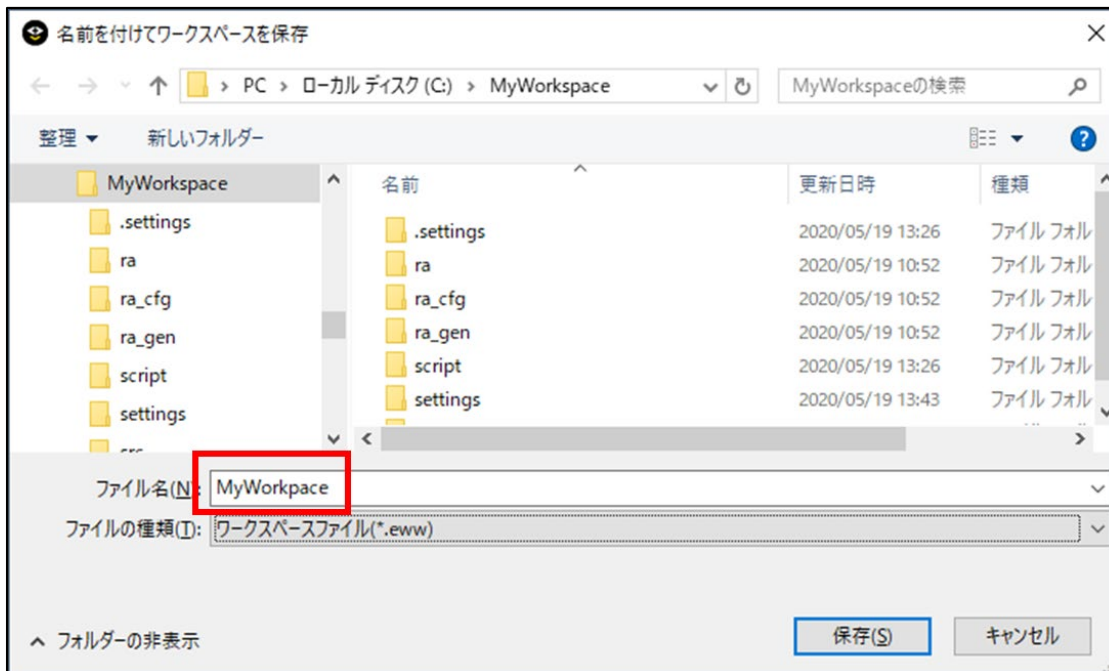
8.6. ワークスペースファイルを保存する

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ため、ここで保存をしておきます。

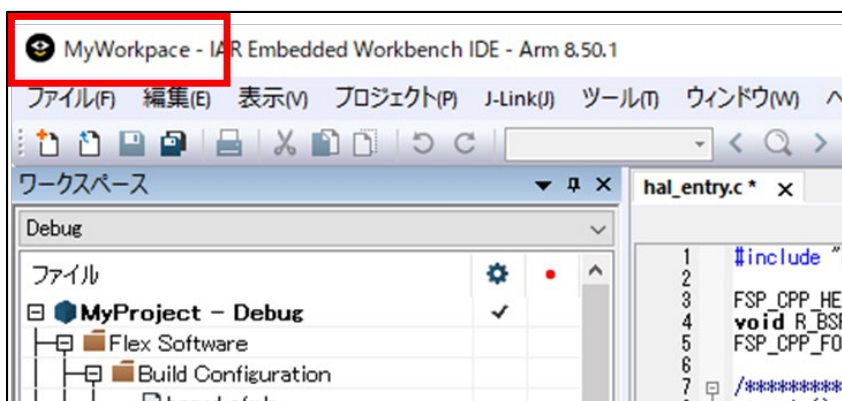
メニューバーの **ファイル > ワークスペースの保存** を選択します。



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。

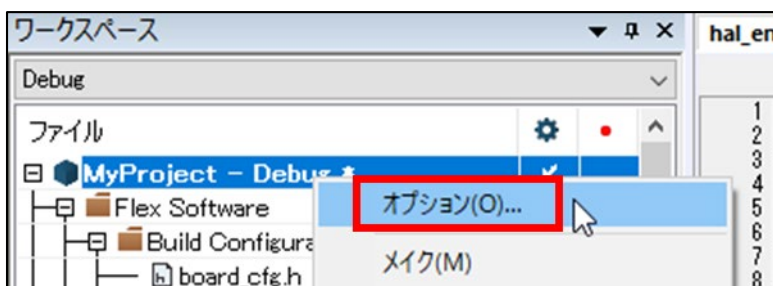


今後複数のプロジェクトを同時に立ち上げているときなど、ワークスペースの識別をするのに便利です。

8.7. EWARM のプロジェクトオプションを変更

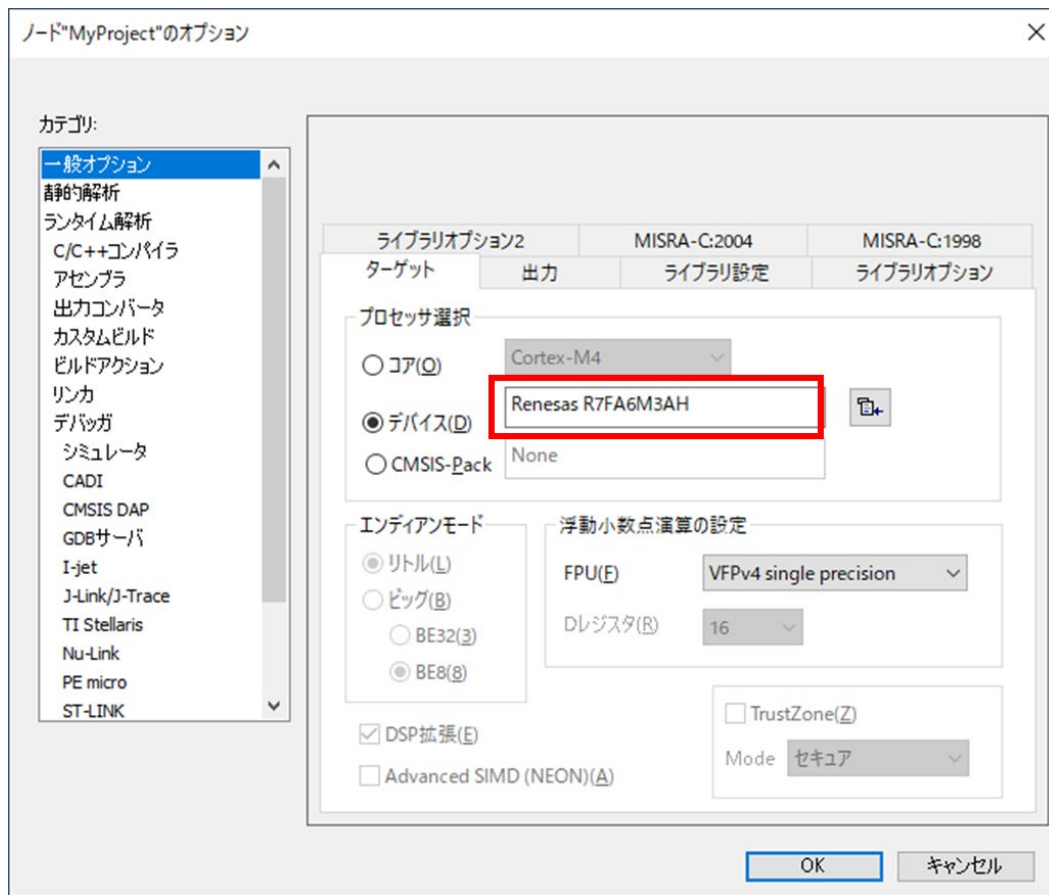
ソースコードが取り込まれたので、プロジェクトのオプション設定を確認・変更していきます。

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

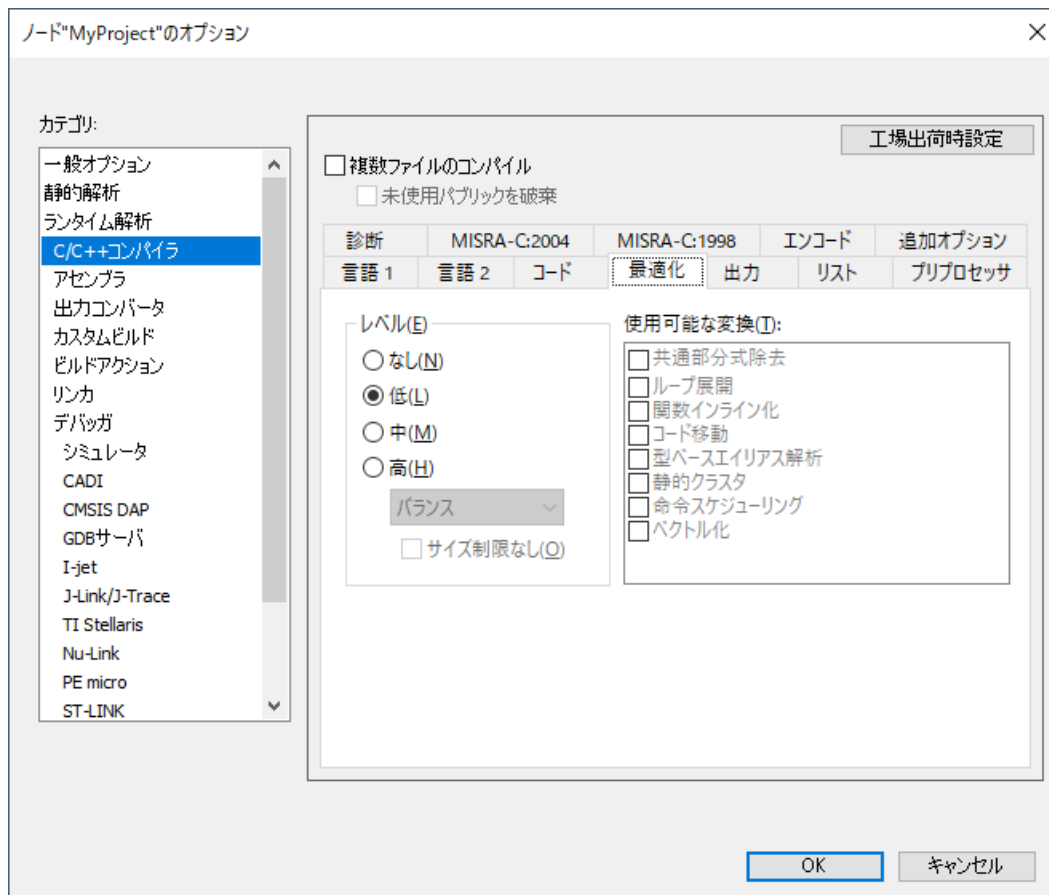


最初のターゲット画面で、**R7FA6M3AH** デバイスが選択されていることがわかります。

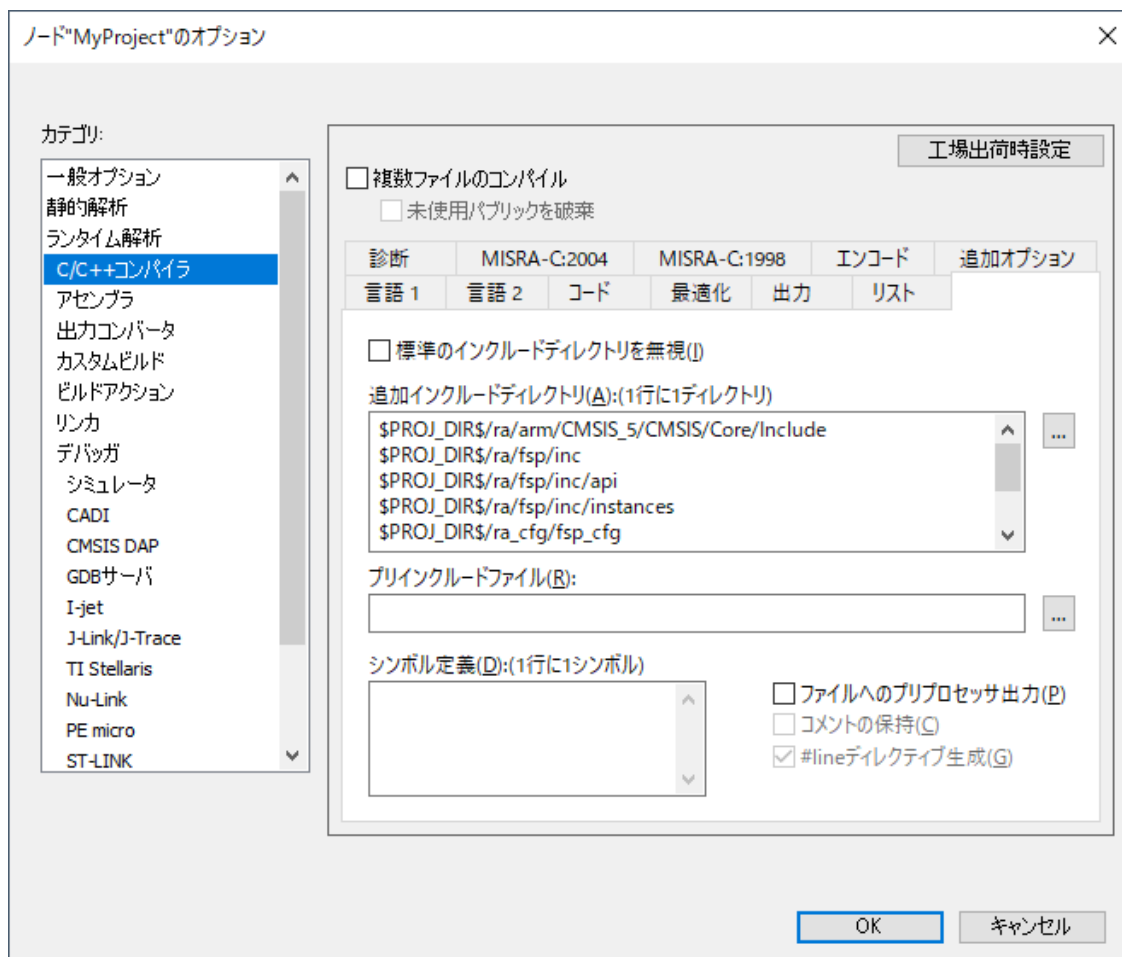
プロジェクトコネクションで取り込んだ際に、ビルドをするための最低限の設定も行われています。



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。

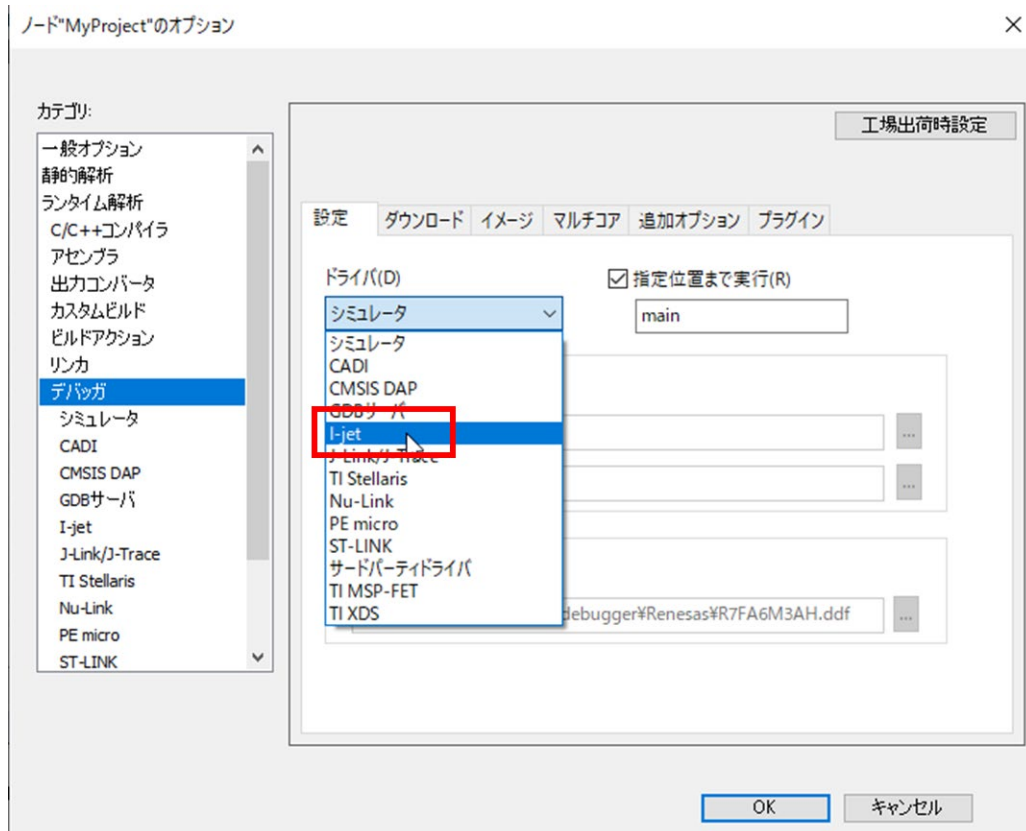


プリプロセッサ タブではプロジェクトで参照しているヘッダファイルのインクルードパスも確認ができます。



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

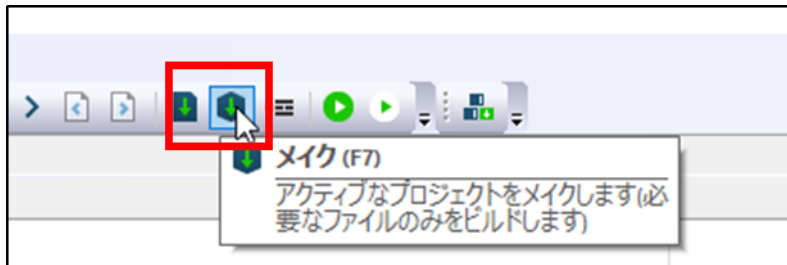
先ほど見た通りアプリケーションのエントリーとなる `hal_entry` 関数は空なので、テスト処理を記述します。

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

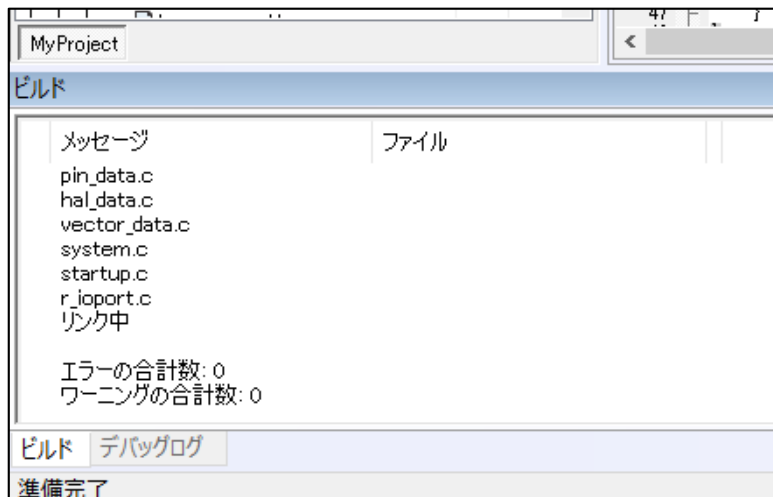
`while (1)` の無限ループの中で、ソフトウェアディレイで待ちながら `count` 変数をインクリメントしていく、というシンプルなコードです。

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。



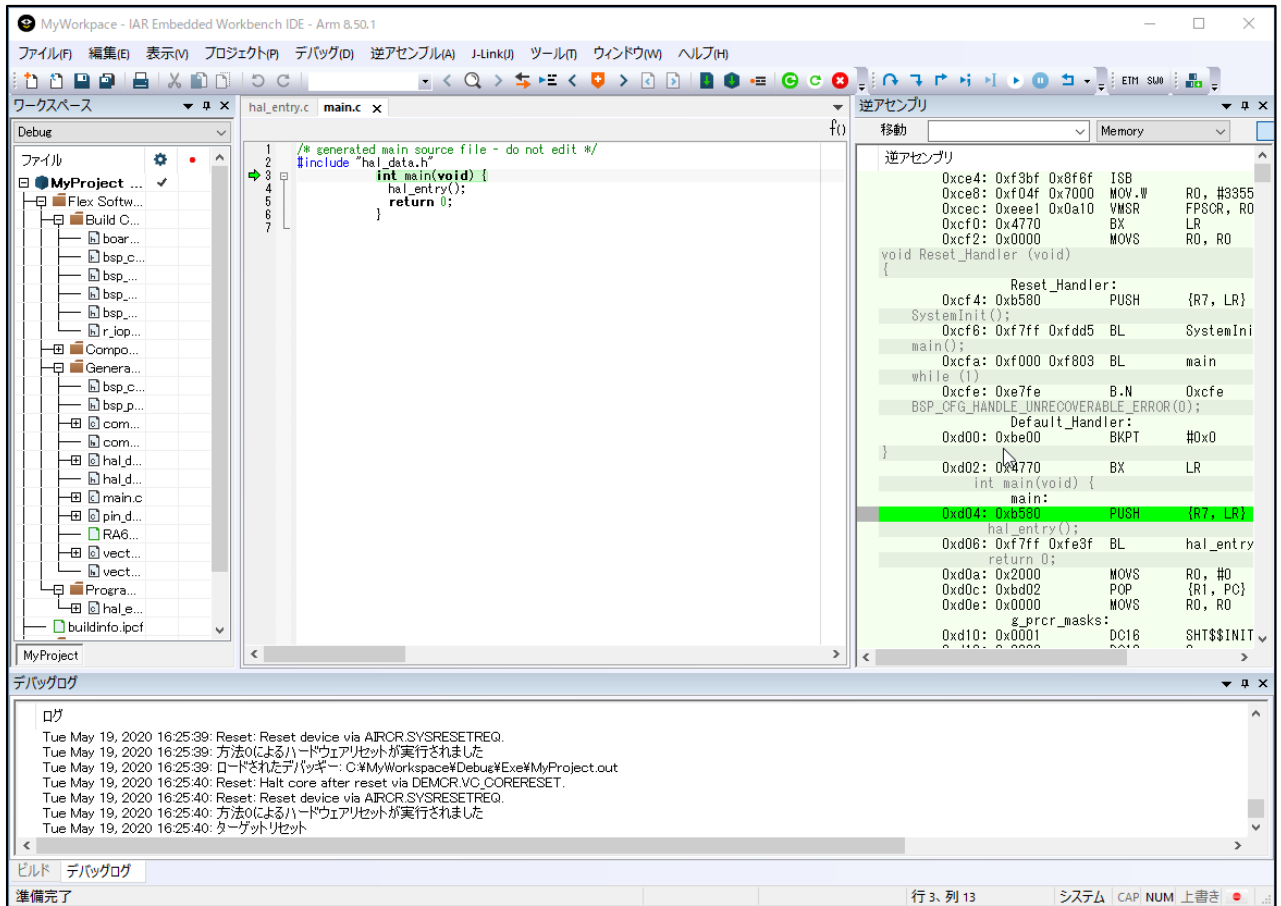
デバッグを開始する前に、以下の設定・接続を確認してください：

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

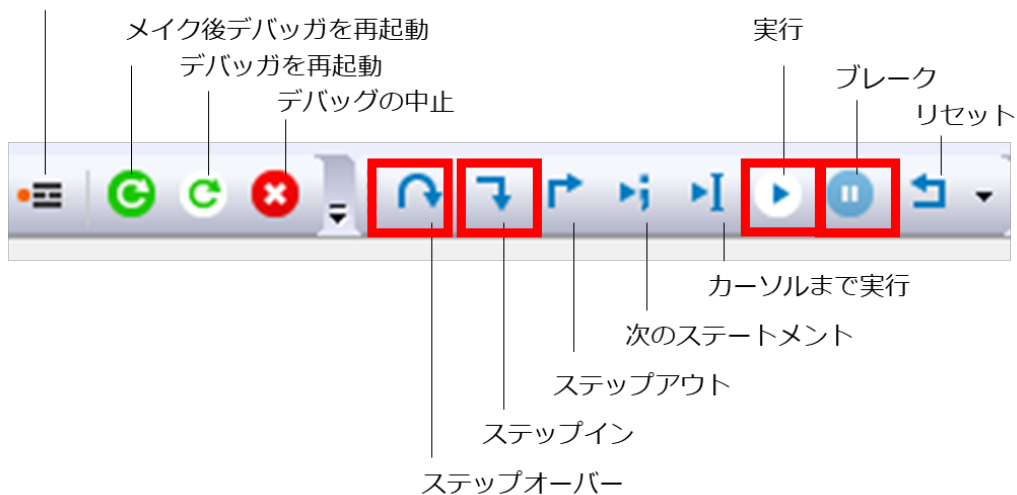




main 関数の先頭で止まっていることがわかります。

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。

ブレークポイントの切り替え



実行 し、



ブレーク します。



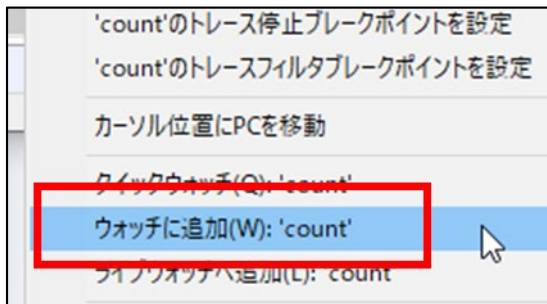
hal_entry 関数のソフトウェアディレイを実行中だということが分かります。

```

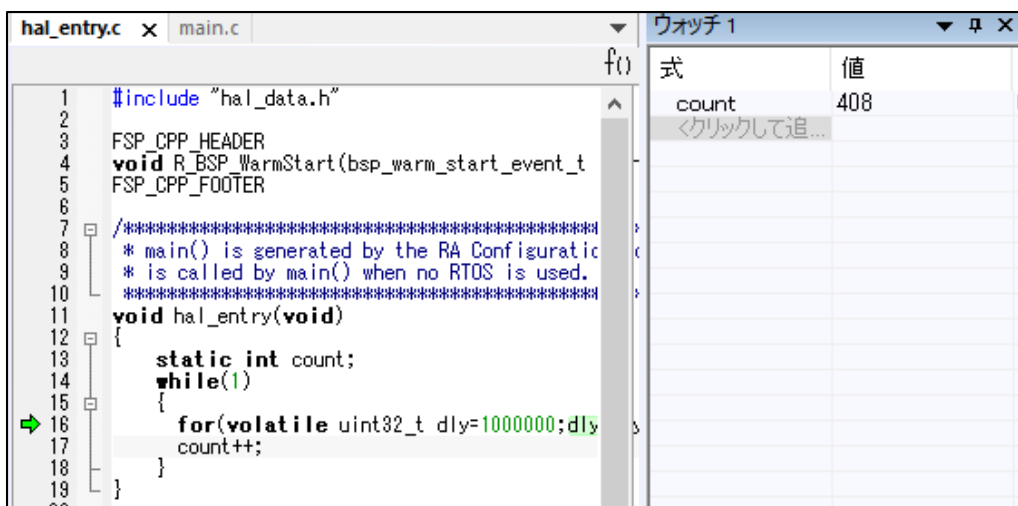
7  |  /*****
8  |  * main() is generated by the RA Configuration editor and
9  |  * is called by main() when no RTOS is used.
10 |  *****/
11 |  void hal_entry(void)
12 |  {
13 |      static int count;
14 |      while(1)
15 |      {
16 |          for(volatile uint32_t dly=1000000;dly;dly--){};
17 |          count++;
18 |      }
19 |  }

```

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、ウォ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プングされており、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)
    {
        /* Access 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。

```

hal_entry.c x
9  * is called by main() when no RTOS is used.
10 *****
11 void hal_entry(void)
12 {
13     static int count;
14     bsp_io_level_t pin_level = BSP_IO_LEVEL_LOW;
15     uint32_t pin = BSP_IO_PORT_01_PIN_12;
16     while(1)
17     {
18         /* Access to PFS register */
19         R_BSP_PinAccessEnable();
20         /* Write to this pin */
21         R_BSP_PinWrite(bsp_io_port_pin_t) pin, pin_level);
22         /* Protect PFS registers */
23         R_BSP_PinAccessDisable();
24         /* Toggle level for next write */
25         if (BSP_IO_LEVEL_LOW == pin_level)
26         {
27             pin_level = BSP_IO_LEVEL_HIGH;
28         }
29         else
30         {
31             pin_level = BSP_IO_LEVEL_LOW;
32         }
33         for(volatile uint32_t dly=1000000;dly;dly--){};
34         count++;
35     }
36 }
37
38
39
40
41
42
43
44 *****

bsp_io.h x
304 *****
305 STATIC_INLINE uint32_t R_BSP_PinRead (bsp_io_port_pin_t pin)
306 {
307     /* Read pin level. */
308     return R_PFS->PORT[pin >> 8].PIN[pin & BSP_IO_PRIV_8BIT_MASK].PmnPFS_b.PIDR;
309 }
310
311 *****
312 * Set a pin to output and set the output level to the level provided
313 * @param[in] pin The pin
314 * @param[in] level The level
315 *****
316 STATIC_INLINE void R_BSP_PinWrite (bsp_io_port_pin_t pin, bsp_io_level_t level)
317 {
318     /* Set output level and pin direction to output. */
319     uint32_t lvl = (uint32_t) level;
320     R_PFS->PORT[pin >> 8].PIN[pin & BSP_IO_PRIV_8BIT_MASK].PmnPFS = BSP_IO_PFS_POR_OUTPUT | lvl;
321 }
322
323 *****
324 * Enable access to the PFS registers. Uses a reference counter to protect against interrupts that
325 * via multiple threads or an ISR re-entering this code.
326 *****
327 STATIC_INLINE void R_BSP_PinAccessEnable (void)
328 {
329     #if BSP_CFG_PFS_PROTECT
330     #if BSP_CFG_PFS_PROTECT
331     /* Get the current state of interrupts */
332     FSP_CRITICAL_SECTION_DEFINE;
333     FSP_CRITICAL_SECTION_ENTER;
334     /* If this is first entry then allow writing of PFS. */
335     if (0 == g_protect_pfswe_counter)
336     {
337         R_PMISC->PWPR = 0; // Clear BOWI bit - writing to PFSWE bit enables
338         R_PMISC->PWPR = 1U << BSP_IO_PWPR_PFSWE_OFFSET; // Set PFSWE bit - writing to PFS register enables
339     }
340     /* Increment the protect counter */
341     g_protect_pfswe_counter++;
342     /* Restore the interrupt state */
343     FSP_CRITICAL_SECTION_EXIT;
344     #endif
345     #endif
346 }
347
348
349

```

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。

```

hal_entry.c x
9  * is called by main() when no RTOS is used.
10 *****
11 void hal_entry(void)
12 {
13     static int count;
14     bsp_io_level_t pin_level = BSP_IO_LEVEL_LOW;
15     uint32_t pin = BSP_IO_PORT_01_PIN_12;
16     while(1)
17     {
18         /* Access to PFS register */
19         R_BSP_PinAccessEnable();
20         /* Write to this pin */
21         R_BSP_PinWrite(bsp_io_port_pin_t) pin, pin_level);
22         /* Protect PFS registers */
23         R_BSP_PinAccessDisable();
24         /* Toggle level for next write */
25         if (BSP_IO_LEVEL_LOW == pin_level)
26         {
27             pin_level = BSP_IO_LEVEL_HIGH;
28         }
29         else
30         {
31             pin_level = BSP_IO_LEVEL_LOW;
32         }
33     }
34 }
35
36
37
38

bsp_io.h x
320 uint32_t lvl = (uint32_t) level;
321 R_PFS->PORT[pin >> 8].PIN[pin & BSP_IO_PRIV_8BIT_MASK].PmnPFS = BSP_IO_PFS_POR_OUTPUT | lvl;
322 }
323
324 *****
325 * Enable access to the PFS registers. Uses a reference counter to protect against interrupts that could
326 * via multiple threads or an ISR re-entering this code.
327 *****
328 STATIC_INLINE void R_BSP_PinAccessEnable (void)
329 {
330     #if BSP_CFG_PFS_PROTECT
331     #if BSP_CFG_PFS_PROTECT
332     /* Get the current state of interrupts */
333     FSP_CRITICAL_SECTION_DEFINE;
334     FSP_CRITICAL_SECTION_ENTER;
335     /* If this is first entry then allow writing of PFS. */
336     if (0 == g_protect_pfswe_counter)
337     {
338         R_PMISC->PWPR = 0; // Clear BOWI bit - writing to PFSWE bit enables
339         R_PMISC->PWPR = 1U << BSP_IO_PWPR_PFSWE_OFFSET; // Set PFSWE bit - writing to PFS register enables
340     }
341     /* Increment the protect counter */
342     g_protect_pfswe_counter++;
343     /* Restore the interrupt state */
344     FSP_CRITICAL_SECTION_EXIT;
345     #endif
346     #endif
347 }
348
349

```

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。


```

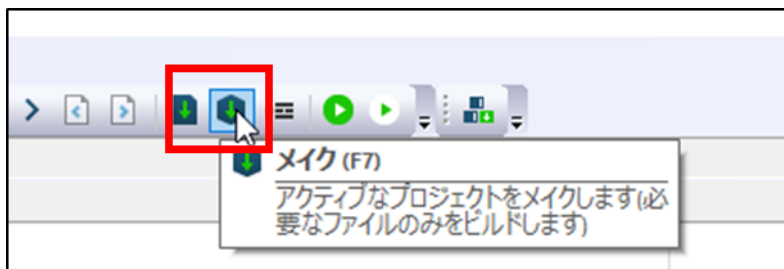
hal_entry.c x      bsp_io.h x
9  * is called by main() when no RTOS is used.
10 *****
11 void hal_entry(void)
12 {
13     static int count;
14
15     bsp_io_level_t pin_level = BSP_IO_LEVEL_LOW;
16     uint32_t pin = BSP_IO_PORT_01_PIN_12;
17
18     while(1)
19     {
20         /* Access to PFS register */
21         R_BSP_PinAccessEnable();
22
23         /* Write to this pin */
24         R_BSP_PinWrite(bsp_io_port_pin_t) pin, pin_level);
25
26         /* Protect PFS registers */
27         R_BSP_PinAccessDisable();
28
29         /* Toggle level for next write */
30         if (BSP_IO_LEVEL_LOW == pin_level)
31         {
32             pin_level = BSP_IO_LEVEL_HIGH;
33         }
34         else
35         {
36             pin_level = BSP_IO_LEVEL_LOW;
37         }
38
39         for(volatile uint32_t dly=1000000;dly;dly--){};
40         count++;
41     }
42 }
43
44 *****
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

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。

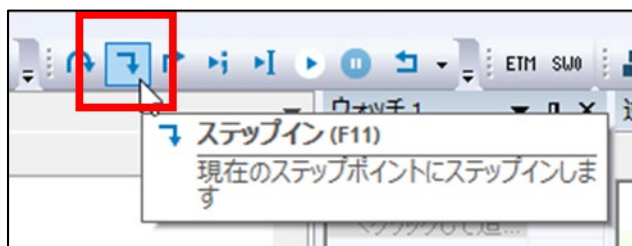
実は、ピン初期化は main 関数が実行される前、hal_entry.c の R_BSP_WarmStart 関数内で実行されています。


```

hal_entry.c x
38
39     for(volatile uint32_t dly=1000000;dly;dly--){};
40     count++;
41 }
42 }
43
44
45 /******
46  * This function is called at various points during the startup process. This
47  * is called right before main() to set up the pins.
48  * @param[in] event Where at in the start up process the code is current
49  * *****
50 void R_BSP_WarmStart (bsp_warm_start_event_t event)
51 {
52     if (BSP_WARM_START_RESET == event)
53     {
54         #if BSP_FEATURE_FLASH_LP_VERSION != 0
55             /* Enable reading from data flash. */
56             R_FACI_LP->DFCTL = 1U;
57
58             /* Would normally have to wait tDSTOP(6us) for data flash recovery.
59              * C runtime initialization, should negate the need for a delay since
60              * C runtime environment and system clocks are setup. */
61         #endif
62     }
63
64     if (BSP_WARM_START_POST_C == event)
65     {
66         /* C runtime environment and system clocks are setup. */
67
68         /* Configure pins. */
69         R_IOPORT_Open(&g_ioport_ctrl, &g_bsp_pin_cfg);
70     }
71 }
72

```

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。

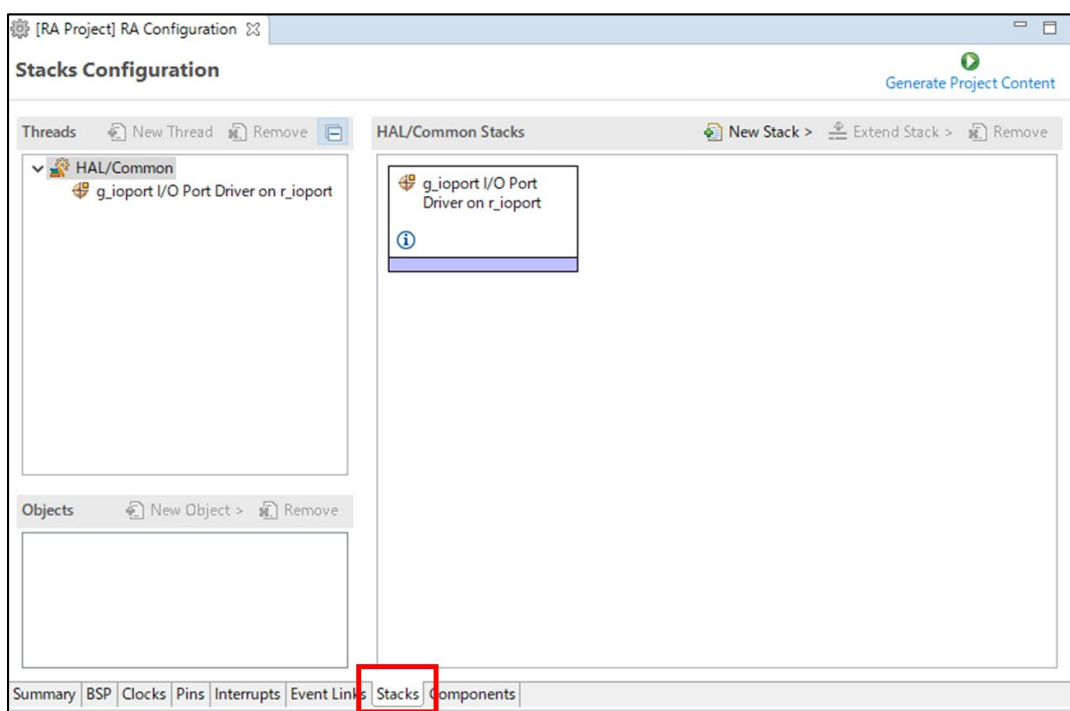
ソフトウェアディレイは、他の処理ができなかったり、不要に電力を消費したり、というデメリットがあるため、タイマーで周期処理を作成することが推奨されます。

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。

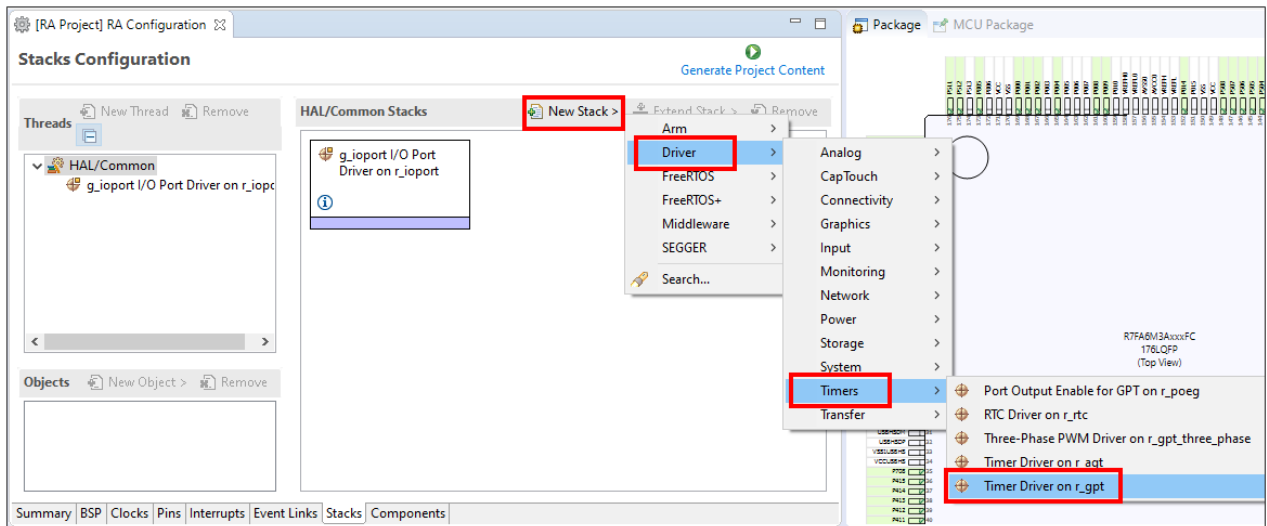
10.1. RA Smart Configurator で Timer スタックを追加

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。



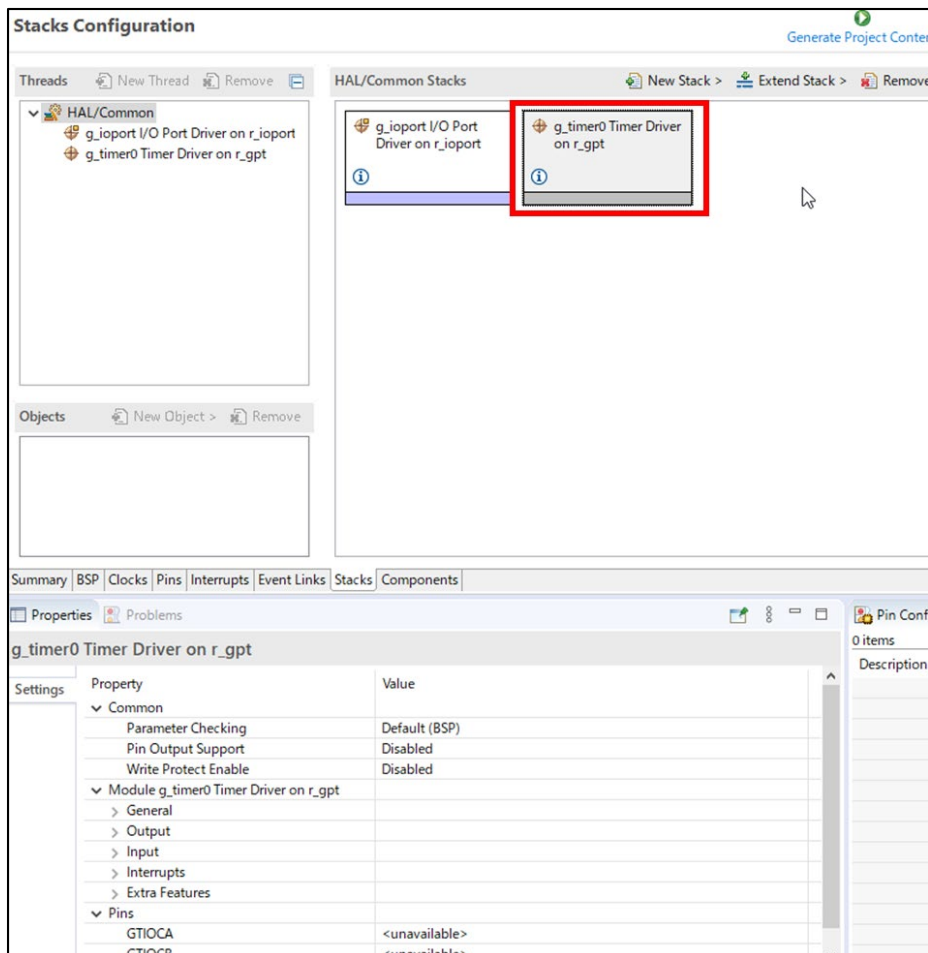
New Stack > Driver > Timers > Timer Driver on t_gpt を選択します。



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。

Property	Value	Property	Value
▼ Common		▼ Common	
Parameter Checking	Default (BSP)	Parameter Checking	Default (BSP)
Pin Output Support	Disabled	Pin Output Support	Disabled
Write Protect Enable	Disabled	Write Protect Enable	Disabled
▼ Module g_timer0 Timer Driver on r_gpt		▼ Module g_timer0 Timer Driver on r_gpt	
▼ General		▼ General	
Name	g_timer0	Name	g_timer0
Channel	0	Channel	0
Mode	Periodic	Mode	Periodic
Period	0x10000000	Period	100
Period Unit	Raw Counts	Period Unit	Milliseconds
> Output		> Output	
> Input		> Input	
▼ Interrupts		▼ Interrupts	
Callback	NULL	Callback	g_timer0_callback
Overflow/Crest Interrupt Priority	Disabled	Overflow/Crest Interrupt Priority	Priority 1
Capture A Interrupt Priority	Disabled	Capture A Interrupt Priority	Disabled
Capture B Interrupt Priority	Disabled	Capture B Interrupt Priority	Disabled
Trough Interrupt Priority	Disabled	Trough Interrupt Priority	Disabled
> Extra Features		> Extra Features	

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 **Milliseconds**、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 秒となるよう運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。

The screenshot shows the RA Configuration tool interface. The 'Interrupts Configuration' window is open, displaying the 'Allocations' section. A red box highlights the following entry in the table:

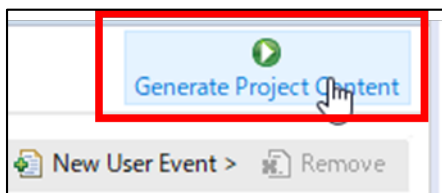
Interrupt	Event	ISR
0	GPT0 COUNTER OVERFLOW (Overflow)	gpt_counter_overflow_isr

At the bottom of the window, the 'Interrupts' tab is selected and highlighted with a red box.

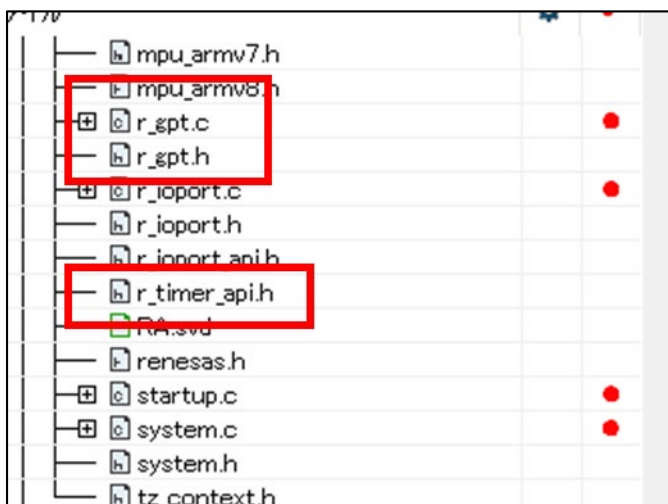
GPT0_COUNTER_OVERFLOW イベントに対して、gpt_counter_overflow_isr が呼ばれることが確認できました。_ISR は RA Smart Configurator が生成する割り込みハンドラです。実際にはここから先ほど設定したコールバック関数が呼ばれます。

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。

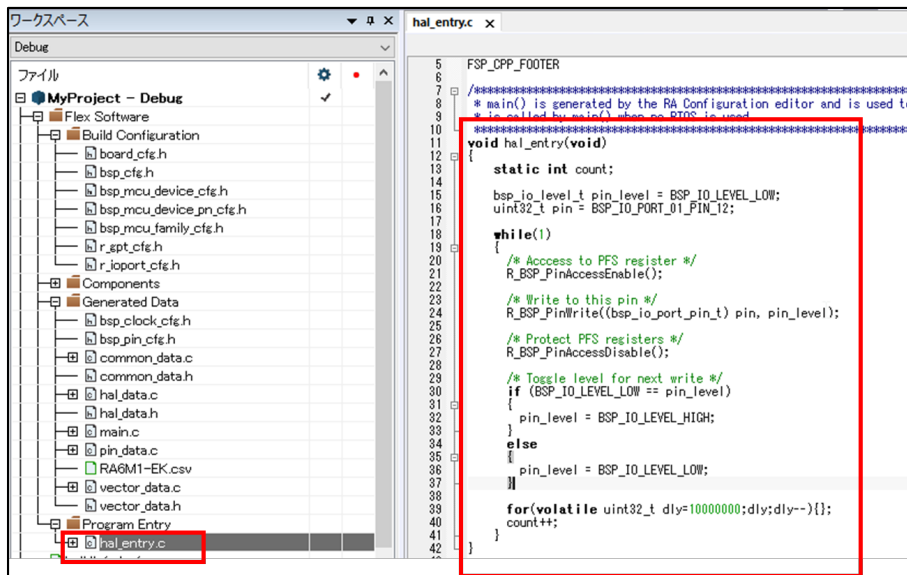


EWARM をアクティブにするとプロジェクトが更新され低レイヤードライバの r_gpt.c/h やラッパーとなる r_timer_api.h などが追加されていることがわかります。



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。

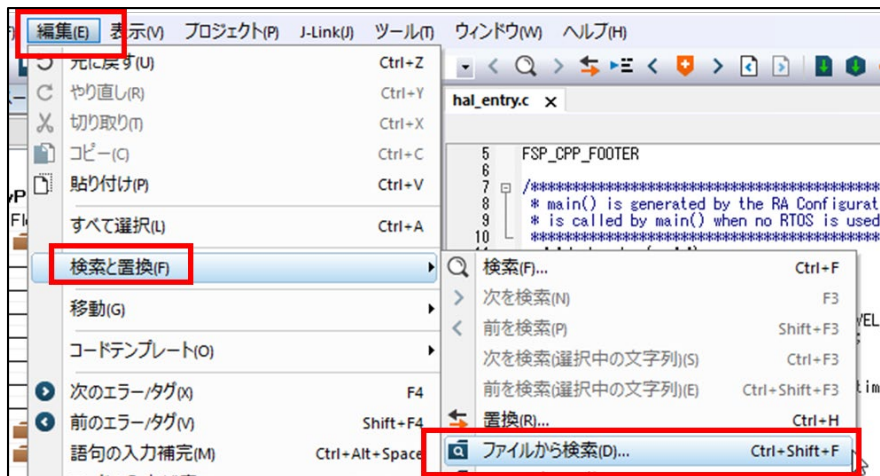


アプリケーション開発を継続することができます。Generated Data Group のファイルは再生成すると上書きされることがあるので、編集をしないようにしてください。

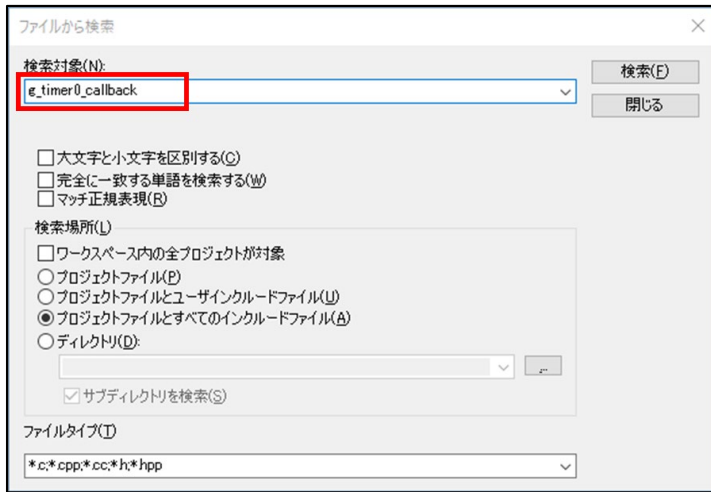
10.5. タイマー割り込みコールバック関数の実装

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

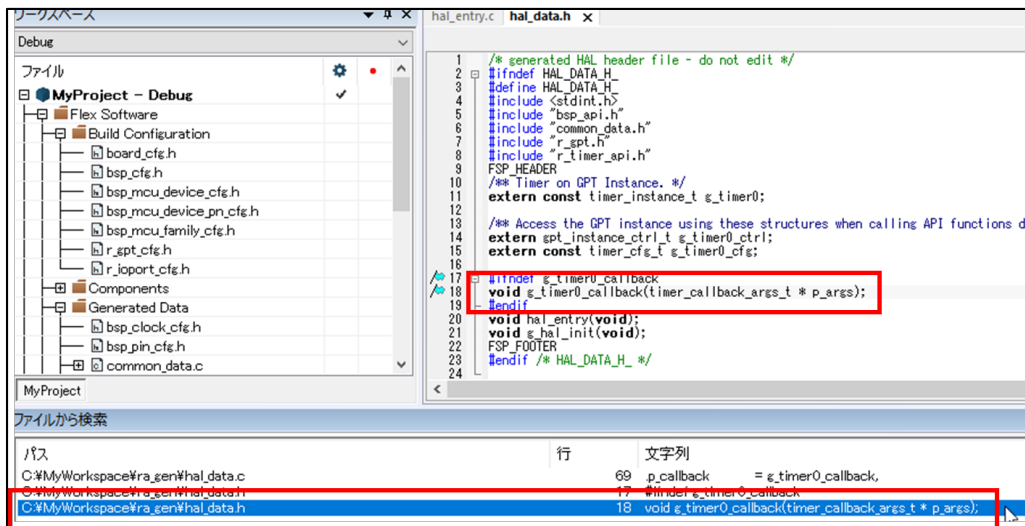
関数の宣言を調べたいときに便利な機能が **編集 > 検索と置換 > ファイルから検索(CTRL + SHIFT + F)** です。



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++;
    }
}

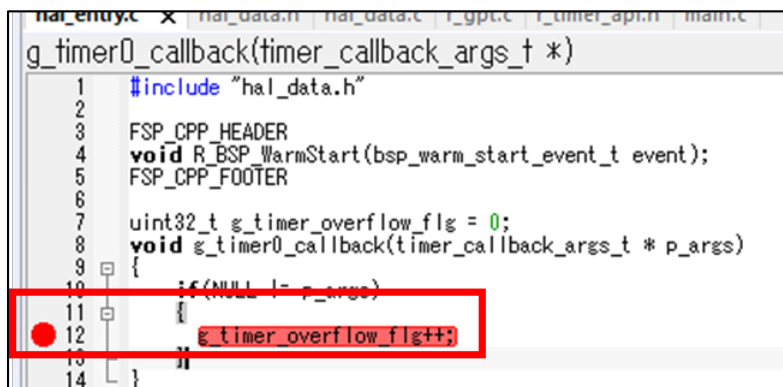
```

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

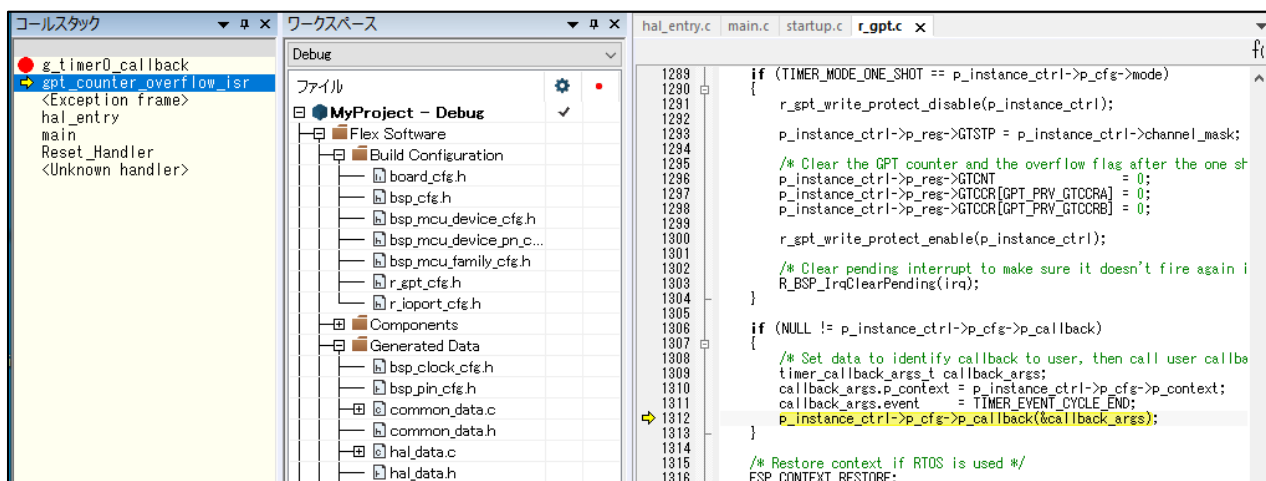
設定したコールバック関数がどのように呼ばれているのかデバッグをします。

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。



```
hal_entry.c x hal_data.h hal_data.c r_gpt.c r_timer_api.h main.c
g_timer0_callback(timer_callback_args_t *)
1  #include "hal_data.h"
2
3  FSP_CPP_HEADER
4  void R_BSP_WarmStart(bsp_warm_start_event_t event);
5  FSP_CPP_FOOTER
6
7  uint32_t g_timer_overflow_flg = 0;
8  void g_timer0_callback(timer_callback_args_t * p_args)
9  {
10     if (NULL != p_args)
11     {
12         g_timer_overflow_flg++;
13     }
14 }
```

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
● g_timer0_callback
  ↳ gpt_counter_overflow_isr
    ↳ hal_entry
      ↳ main
        ↳ Reset_Handler
          ↳ <Unknown handler>

ワークスペース
Debug
MyProject - Debug
  Flex Software
  Build Configuration
  board_cfg.h
  bsp_cfg.h
  bsp_mcu_device_cfg.h
  bsp_mcu_device_pin_c...
  bsp_mcu_family_cfg.h
  r_gpt_cfg.h
  r_joport_cfg.h
  Components
  Generated Data
  bsp_clock_cfg.h
  bsp_pin_cfg.h
  common_data.c
  common_data.h
  hal_data.c
  hal_data.h

hal_entry.c main.c startup.c r_gpt.c x
1289     if (TIMER_MODE_ONE_SHOT == p_instance_ctrl->p_cfg->mode)
1290     {
1291         r_gpt_write_protect_disable(p_instance_ctrl);
1292
1293         p_instance_ctrl->p_reg->GTSTP = p_instance_ctrl->channel_mask;
1294
1295         /* Clear the GPT counter and the overflow flag after the one st
1296         p_instance_ctrl->p_reg->GTCONT = 0;
1297         p_instance_ctrl->p_reg->GTCCR (GPT_PRIV_GTCCRA) = 0;
1298         p_instance_ctrl->p_reg->GTCCR (GPT_PRIV_GTCCRB) = 0;
1299
1300         r_gpt_write_protect_enable(p_instance_ctrl);
1301
1302         /* Clear pending interrupt to make sure it doesn't fire again i
1303         R_BSP_IrqClearPending(irq);
1304     }
1305
1306     if (NULL != p_instance_ctrl->p_cfg->p_callback)
1307     {
1308         /* Set data to identify callback to user, then call user callba
1309         timer_callback_args_t callback_args;
1310         callback_args.p_context = p_instance_ctrl->p_cfg->p_context;
1311         callback_args.event = TIMER_EVENT_CYCLE_END;
1312         p_instance_ctrl->p_cfg->p_callback(&callback_args);
1313     }
1314
1315     /* Restore context if RTOS is used */
1316     FSP_CONTEXT_RESTORE;
```

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 ファミリー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。