RA0L1 グループ

FPB-RA0L1 チュートリアル

要旨

本書ではルネサス製 RA0L1 グループ MCU を搭載した FPB-RA0L1 ボードを対象に、統合開発環境 e² studio を用いてプロジェクト新規作成からタッチセンサの設定、プログラム作成、およびデバッグまでの操作手順を説明します。

動作確認デバイス

RA0L1 グループ

参考ドキュメント

[1] RA0L1 グループ ユーザーズマニュアル ハードウェア編(R01UH1143)
[2] RA0L1 MCU グループ用評価キット FPB-RA0L1 v1 ユーザーズマニュアル(R20UT5601)
[3] 統合開発環境 e² studio 2023-10 以上 ユーザーズマニュアル クイックスタートガイド ルネサスマイクロコントローラ RA ファミリ(R20UT5210)

目次

1. 開発環境

本アプリケーションノートでは、下記の開発環境を用いて説明します。

1.1 ハードウェア環境

以下のハードウェアを使用します。

- ボード：FPB-RA0L1(製品型名：RTK7FPA0L1S000010BJ、搭載デバイス：R7FA0L1074CFL)
  ボードと PC を製品付属の USB Type A - Type C ケーブルで接続して下さい。

1.2 ソフトウェア環境

本書では以下のソフトウェアを使用します。あらかじめソフトウェアをインストールしてください。

- 統合開発環境
  — e² studio : 2025-07 以降
- コンパイラ
  — GNU ARM Embedded : 13.3.1.arm-13.24 以降
- Flexible Software Package
  — Version : 6.1.0 以降
- 静電容量式タッチセンサ対応開発支援ツール QE for Capacitive Touch
  — Version : 4.2.0 以降

2. ソフトウェア概要

本アプリケーションノートで作成するプログラムの仕様を説明します。

2.1 作成するプログラム

RA0L1 グループのタッチセンサ機能を使用して以下の制御を行うプログラムを作成します。

- Touch Button のタッチ検出状態に応じて LED5 と LED6 の点灯状態を制御する
- Touch Button1 のタッチ検出により LED1 と LED2 の点滅動作の開始/停止を制御する
- Touch Button2 のタッチ検出により点滅周期を変更する

使用するボードと LED とタッチセンサの位置を示します。

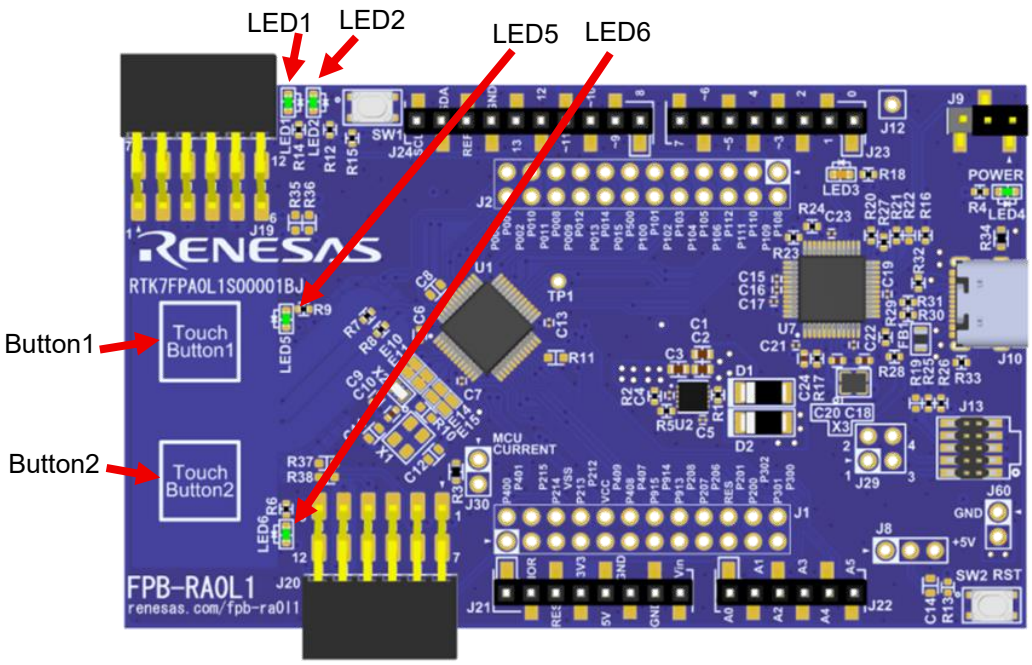


図 2-1. ボード上の LED 位置

2.2 使用するリソース

プログラム作成に当たり、RA0L1 グループの以下のリソースを使用します。

2.2.1 クロック発生回路

HOCO クロック周波数: 32MHz

HOCO クロック分周: 1 分周

ICLK 選択ソース：HOCO

ICLK 周波数：32MHz

TAU CK00 供給クロック ：32MHz

2.2.2 低電圧検出回路(LVD)

使用する低電圧検出回路: 電圧監視 0

検出電圧: 1.86V

2.2.3 静電容量式センシングユニット(CTSU2SLa)

使用するタッチセンサ機能：TouchButton1、TouchButton2

使用する FSP のソフトウェアスタック：rm_touch、r_ctsu

2.2.4 タイマアレイユニット(TAU)

使用するタイマ機能：	TAU チャンネル 0
チャンネル 0 に供給する TAU クロック：	CK00
チャンネル 0 の動作モード：	インターバルタイマモード
インターバルタイマ周期：	1ms
使用する FSP のソフトウェアスタック：	r_tau

2.2.5 I/O ポート

使用するポート ：	P002(LED1) / P104(LED2) / P401(LED5) / P400(LED6)
使用する FSP のソフトウェアスタック ：	r_ioport

3. プロジェクト新規作成

この章ではプロジェクトの作成方法を説明します。

3.1 プロジェクト起動

1. e² studio のメニューバーから
「新規」→「Renesas C/C++ Project」→「Renesas RA」を選択します。

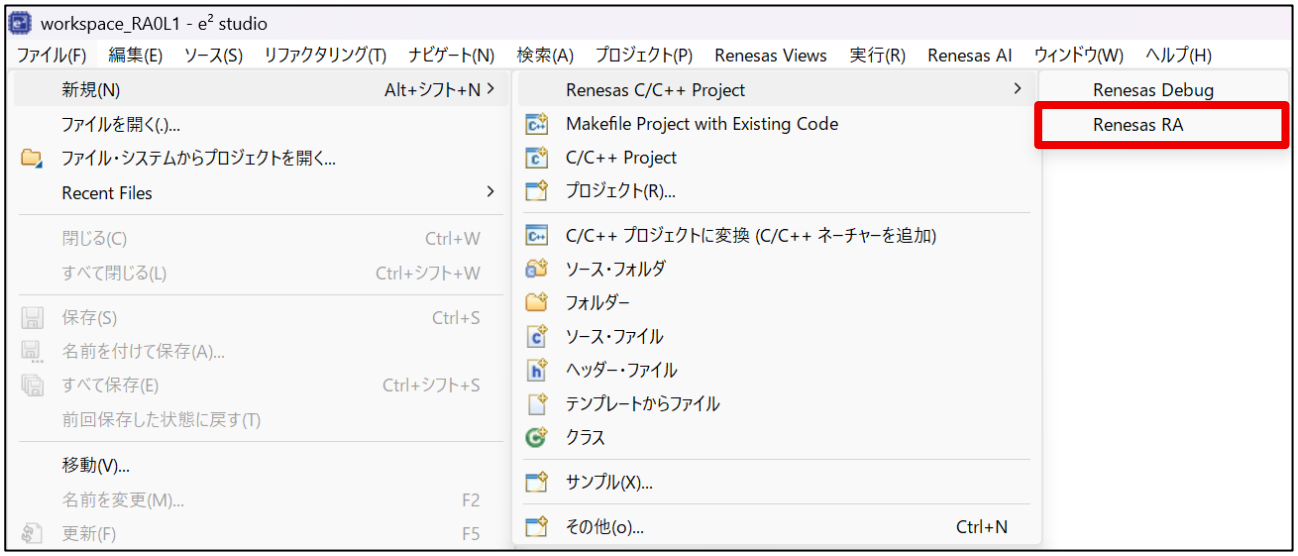


図 3-1. プロジェクト起動

2. 「Renesas RA C/C++ Project」を選択し、「次へ」をクリックします。

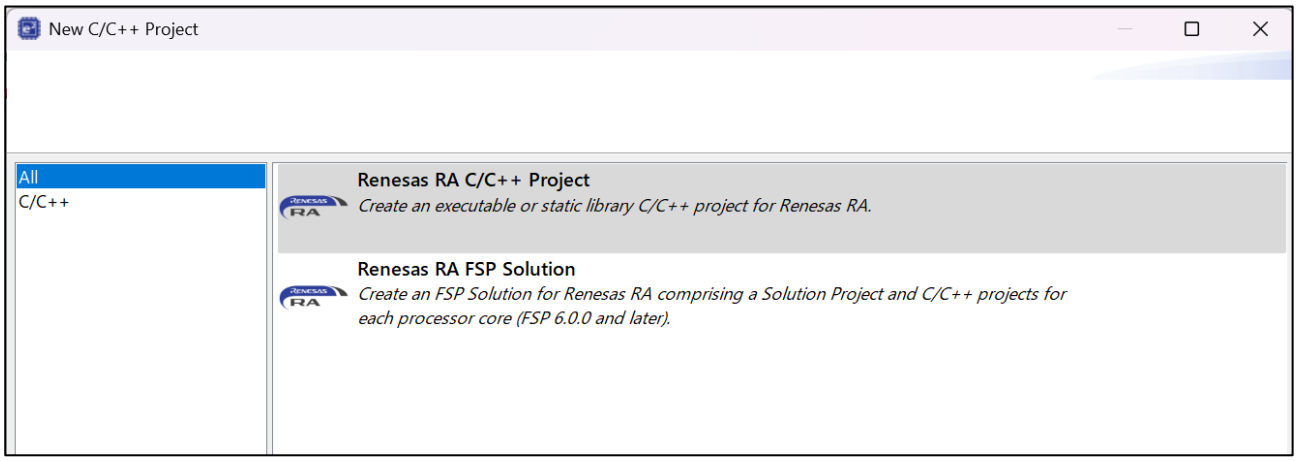


図 3-2. プロジェクト起動

3.2 プロジェクト名の設定

Project name に任意の名前を記述し、「次へ」をクリックします。(本書では FPB_RA0L1_Tutorial とします)

”デフォルト・ロケーションの使用”にチェックを入れた場合、プロジェクトの作成場所は下に表示されているパスに作成されます。任意の場所に作成したい場合は、チェックを外してパスを設定してください。

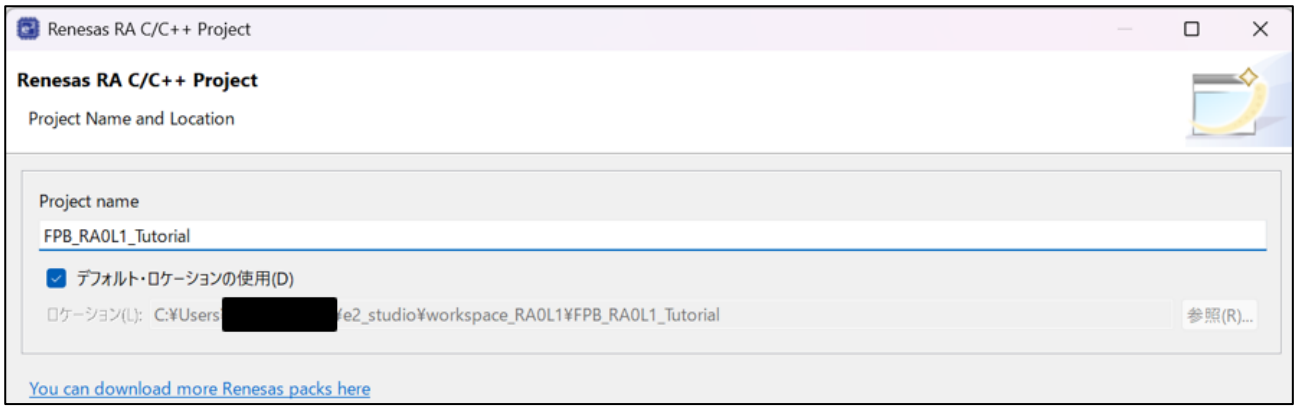


図 3-3. プロジェクト名の設定

3.3 デバイス・ツール設定画面

Device and Tools Selection 画面では以下のように設定します。

1. FSP Version：最新版を選択します。
2. Board：一覧から「RA0」→「RA0L1」→「FPB-RA0L1」を選択します。
3. Device：Board を選択すると自動で設定されます。
4. Language：「C」を選択します。

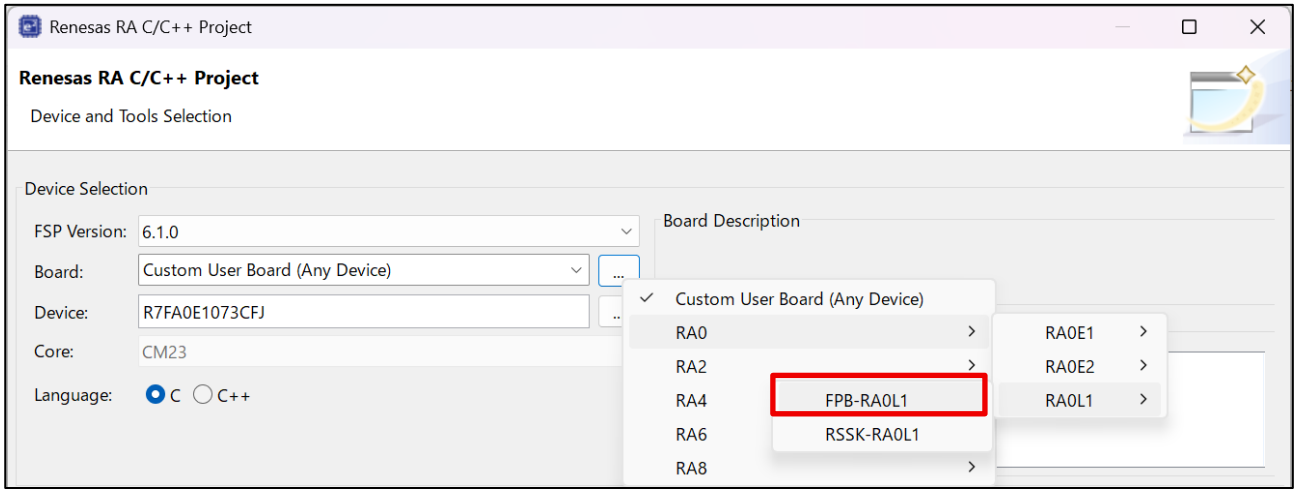


図 3-4. デバイス・ツール設定画面

5. Toolchains の設定：“GNU ARM Embedded”を選択します。バージョンの一覧より、最新版を選択します。

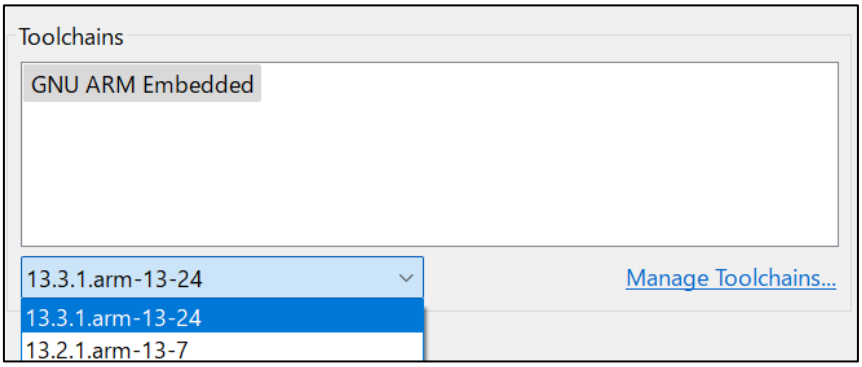


図 3-5. デバイス・ツール設定画面

6. Debugger の設定：J-Link ARM を選択します。

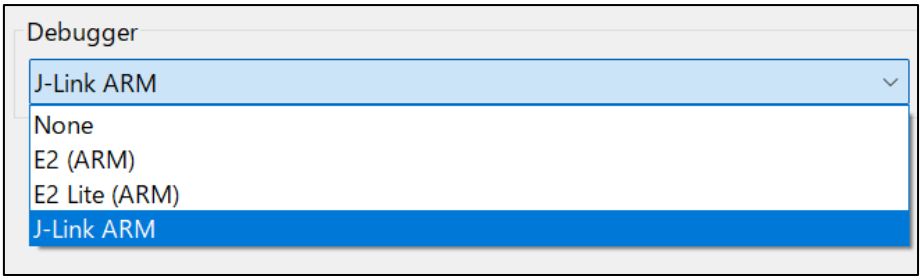


図 3-6. デバイス・ツール設定画面

設定は完了です。

7. 赤枠部分の設定を確認して「次へ」をクリックします。

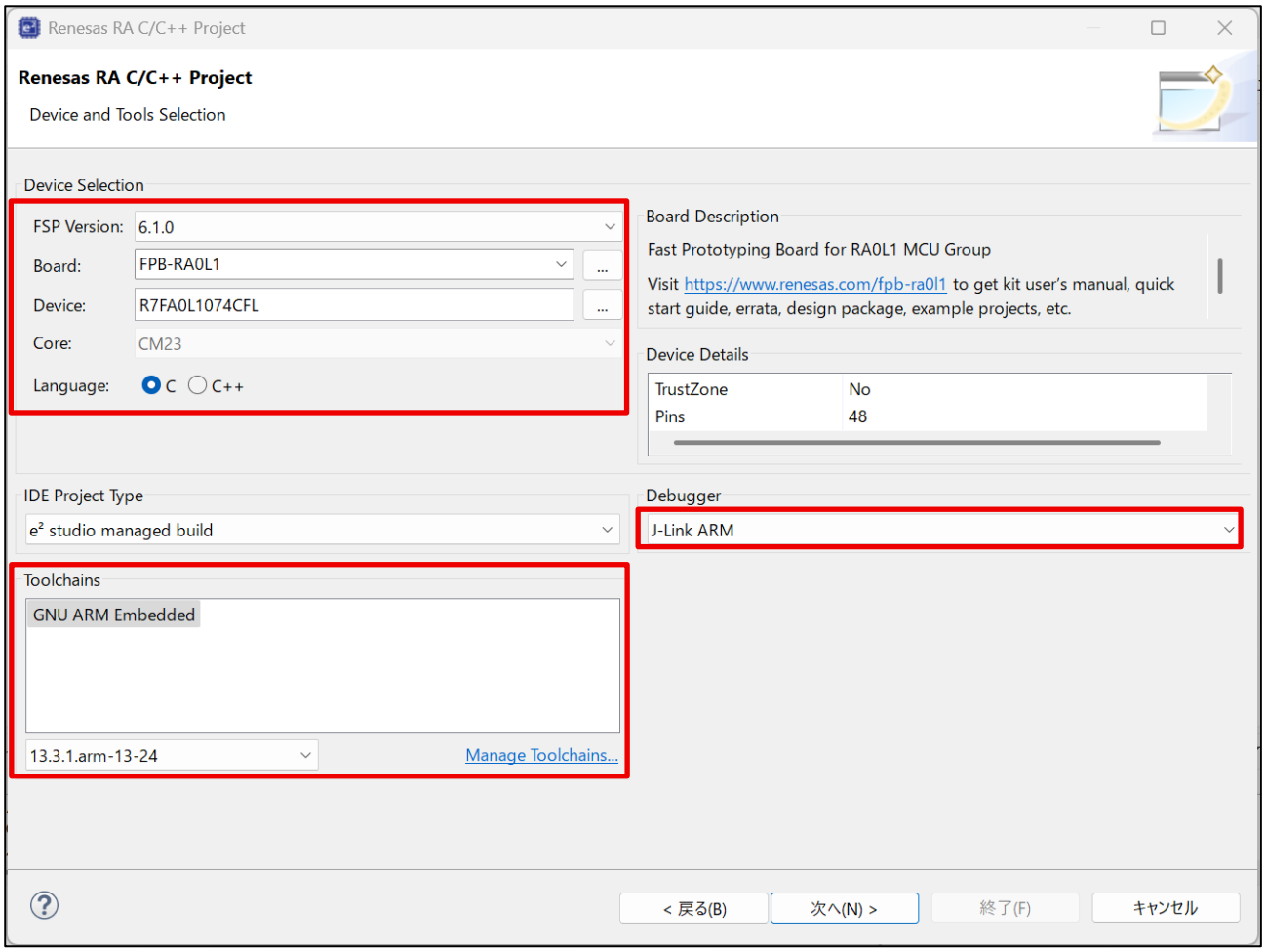


図 3-7. デバイス・ツール設定画面

3.4 前のプロジェクト・スマートバンドルの設定

「None」が選択されていることを確認して「次へ」をクリックします。

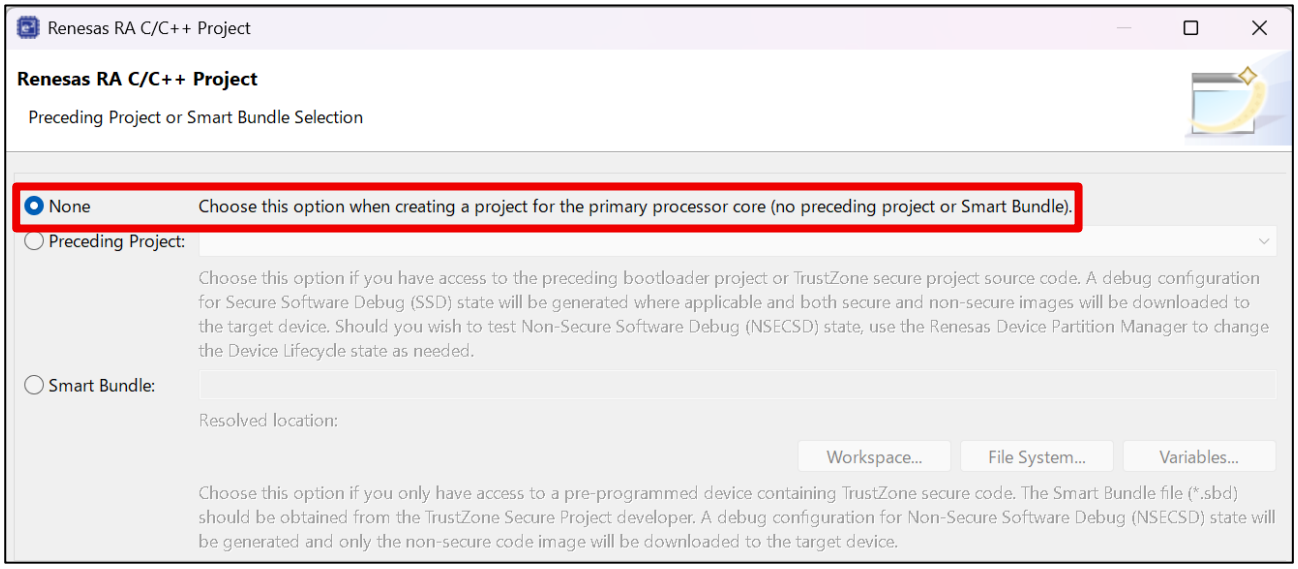


図 3-8. 前のプロジェクト・スマートバンドルの設定

3.5 ビルドアーティファクトの設定

本アプリケーションノートではプログラムから実行ファイルを生成しますので「Executable」を選択します。

RTOS は使いませんので No RTOS を選択します。2 つ選択したら「次へ」をクリックします。

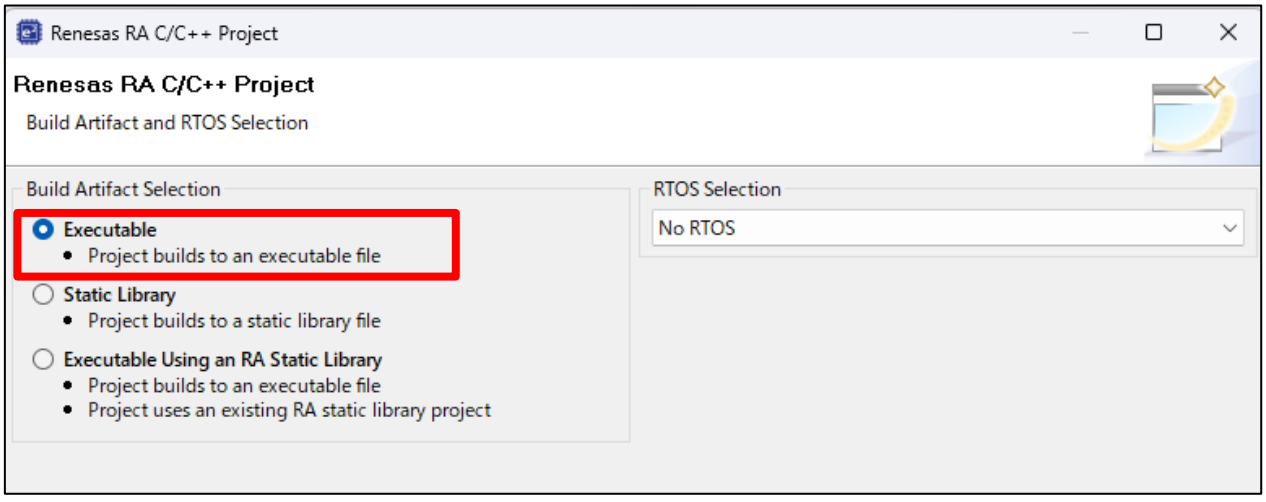


図 3-9. ビルドアーティファクトの設定

3.6 テンプレートタイプの設定

1. 本アプリケーションノートでは FSP の使い方を交えて説明しますので今回は「Bare Metal – Minimal」を選択します。その後、「終了」をクリックしてください。

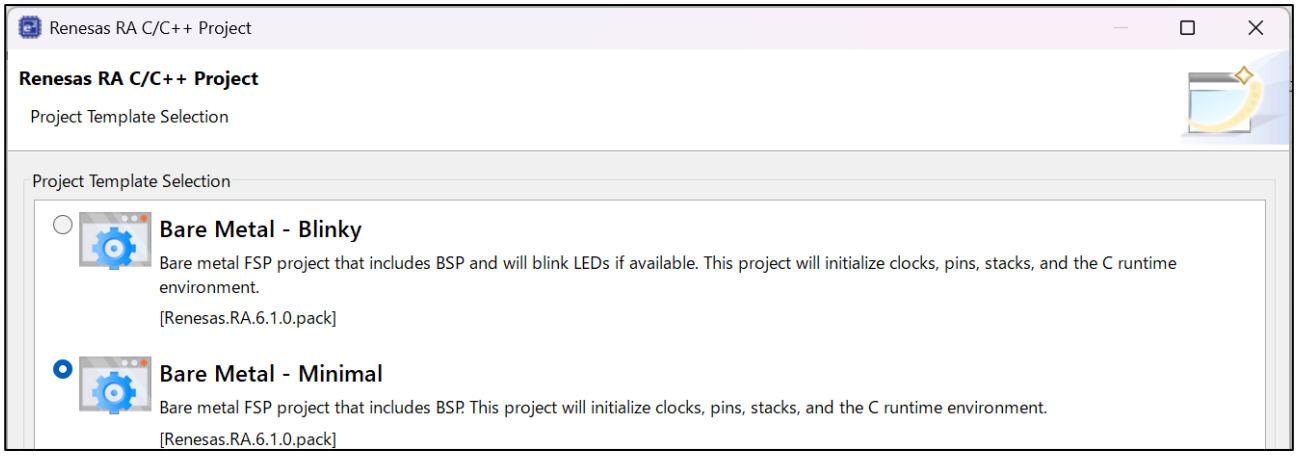


図 3-10. テンプレートタイプの設定

2. パースペクティブ画面表示のポップアップが開きますので、「パースペクティブを開く」をクリックします。

※「いいえ」をクリックしても問題ありません。

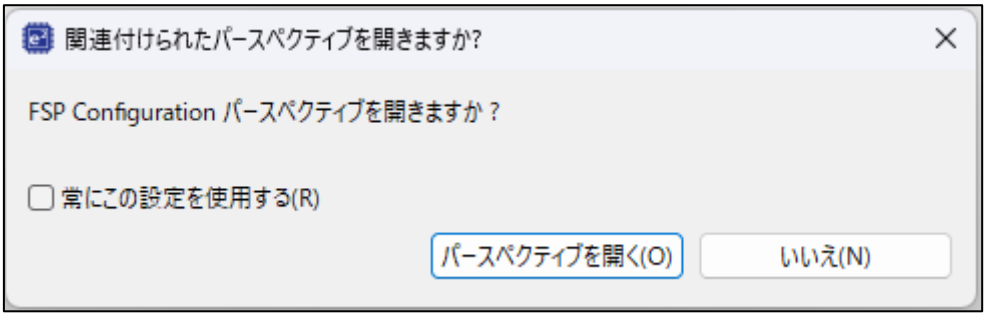


図 3-11. テンプレートタイプの設定

3. プロジェクトの作成が完了しました。

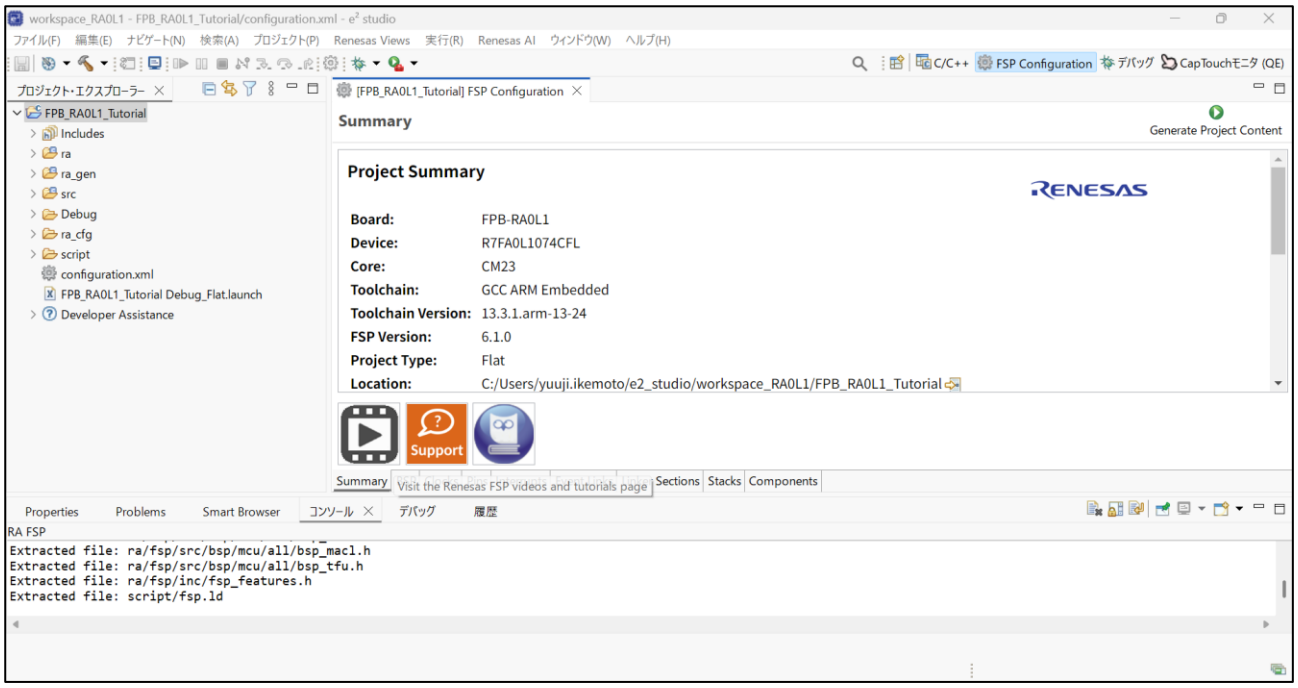


図 3-12. テンプレートタイプの設定

4. QE Touch を用いたタッチセンサのチューニングとモニタリング

4.1 タッチ IF で使う FSP Configurator 設定

FSP Configurator を用いて、システムのクロック設定や、周辺機能の初期設定を行います。

4.1.1 FSP Configurator 画面の呼び出し方法

プロジェクト・エクスプローラーの configuration.xml ファイルをダブルクリックして、設定画面を表示します。

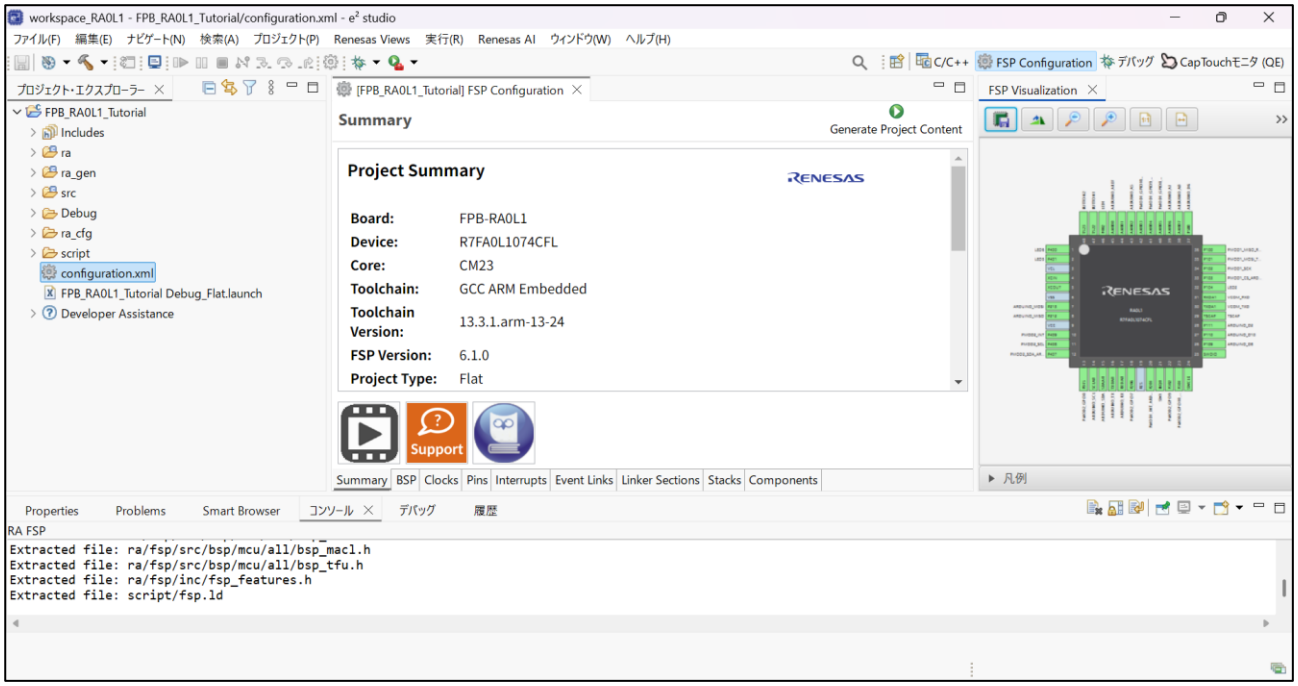


図 4-1.FSP Configurator 画面の呼び出し方法

4.1.2 クロック設定画面

“Clocks”タブを選択してクロック設定画面を開きます。デフォルト画面は以下のようになっていることを確認します。今回は 32MHz そのまま使用するので変更しません。TAU CK00 の供給クロックが 32MHz であることを確認します。

各クロックの経路は太い矢印で示されています。

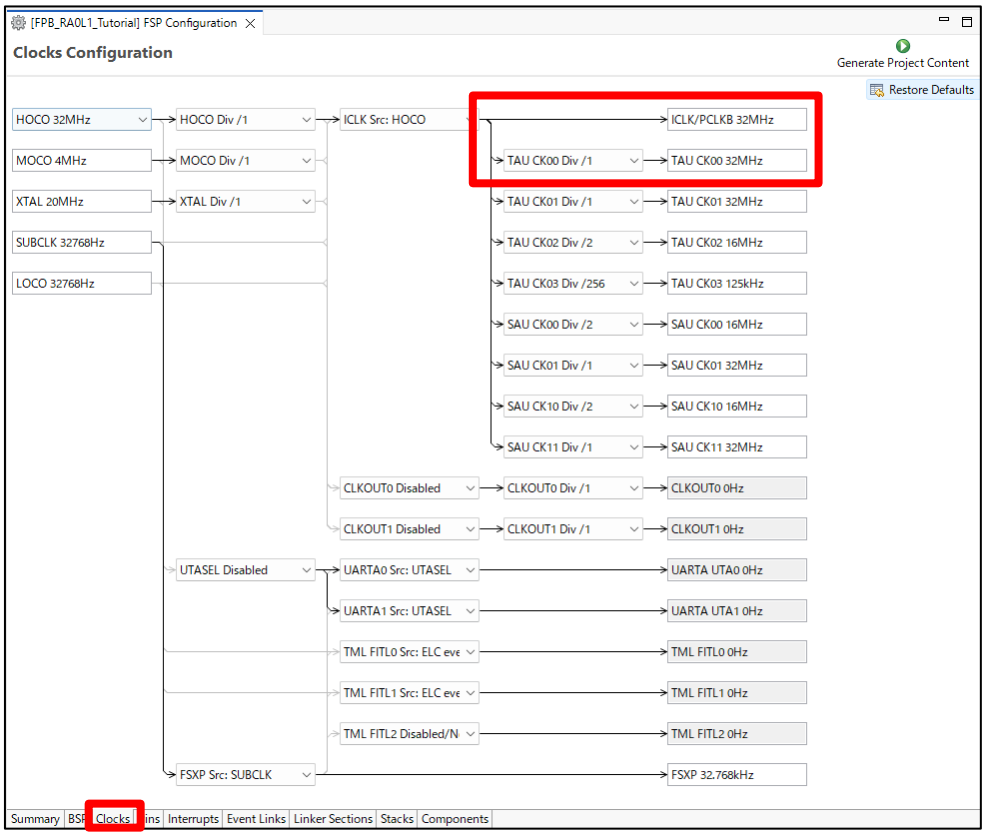


図 4-2. クロック設定画面

4.1.3 低電圧検出回路の設定

1. “BSP”タブを選択して Board Support Package Configuration 画面を開きます。そして下のパネルの”プロパティー”タブを選択し、プロパティー画面を表示します。

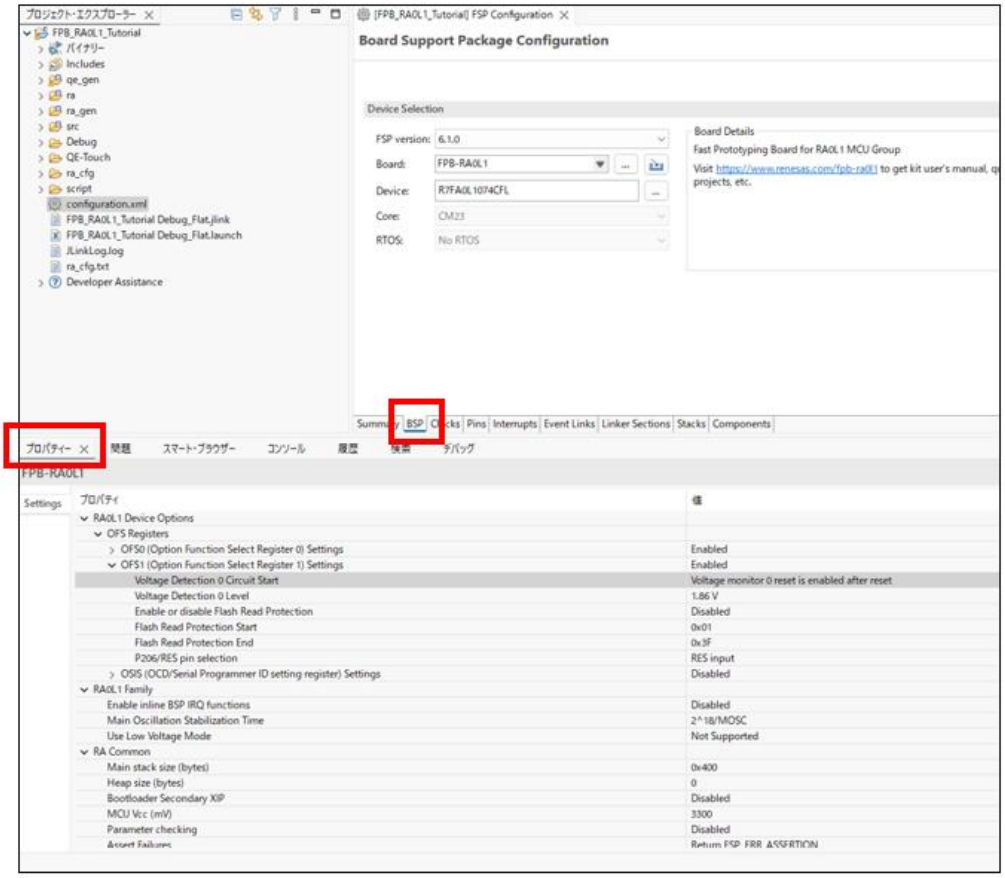


図 4-3. プロパティー画面の表示

2. プロパティー画面にて、”Voltage Detection 0 Circuit Start”を”Voltage monitor 0 reset is enabled after reset”に、”Voltage Detection 0 Level”を”1.86V”に設定します。

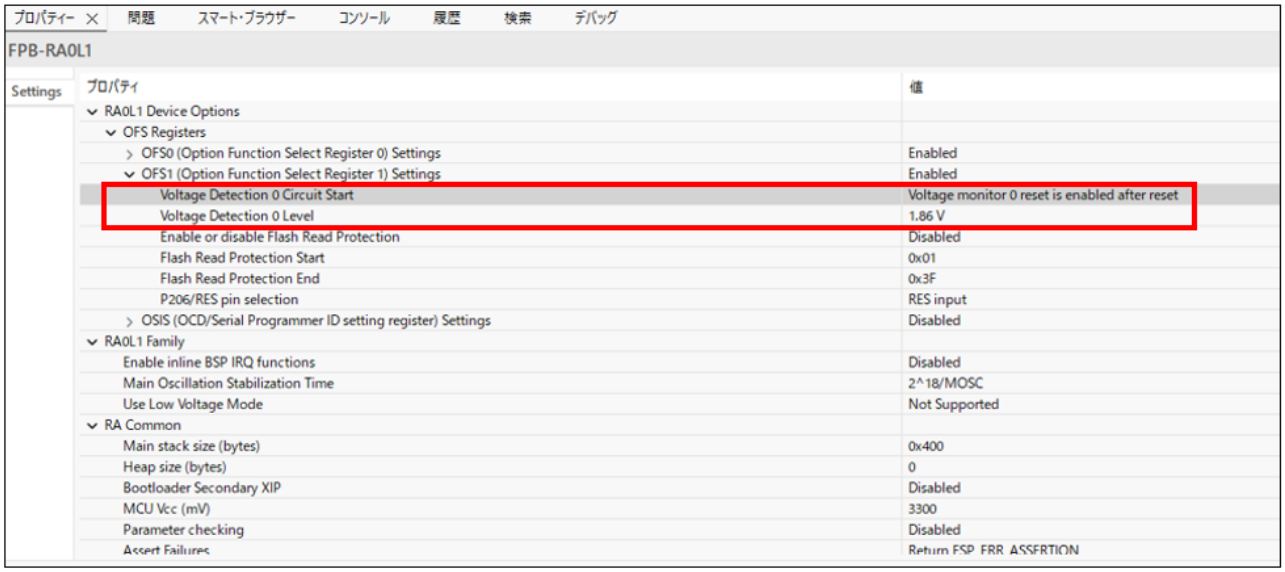


図 4-4. 低電圧検出回路の設定画面

4.1.4 Touch ドライバの設定

1. FSP Configurator 画面の「Stack」タブを開き、「New Stack」→「CapTouch」→「Touch（rm_touch）」を追加します。

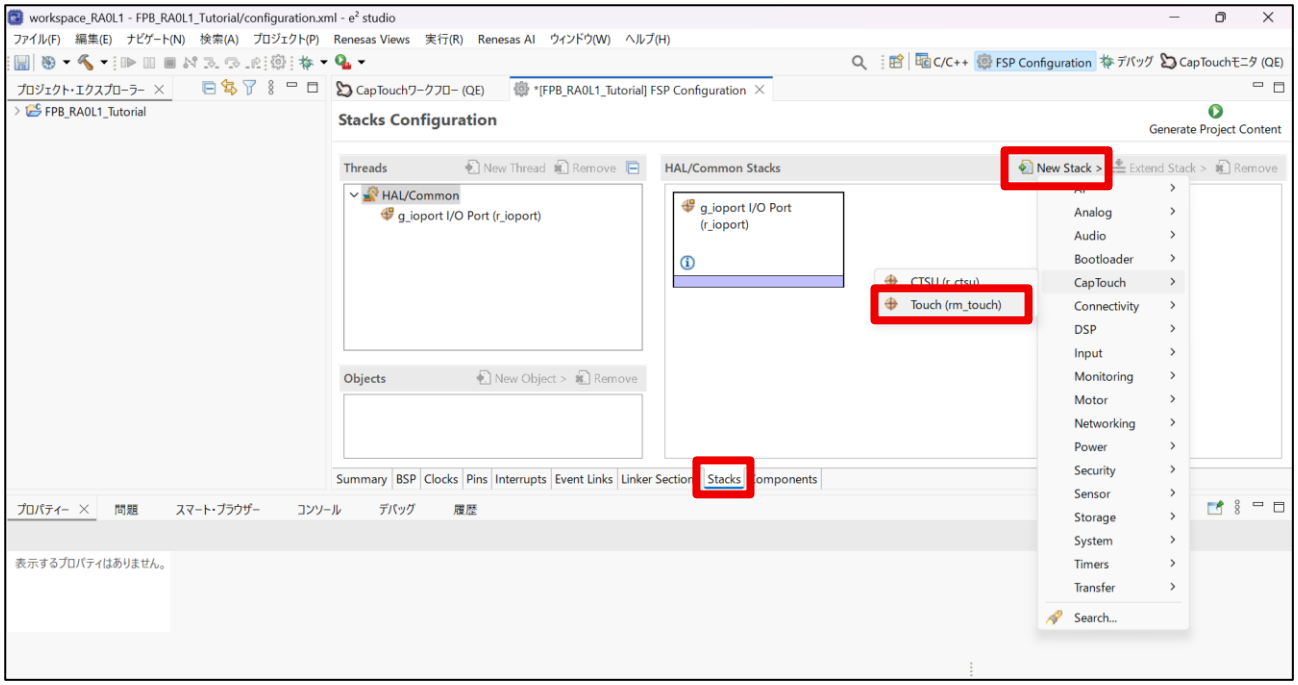


図 4-5. Touch ドライバの設定

2. スタックの追加を確認できたら Touch ドライバの設定は完了です。

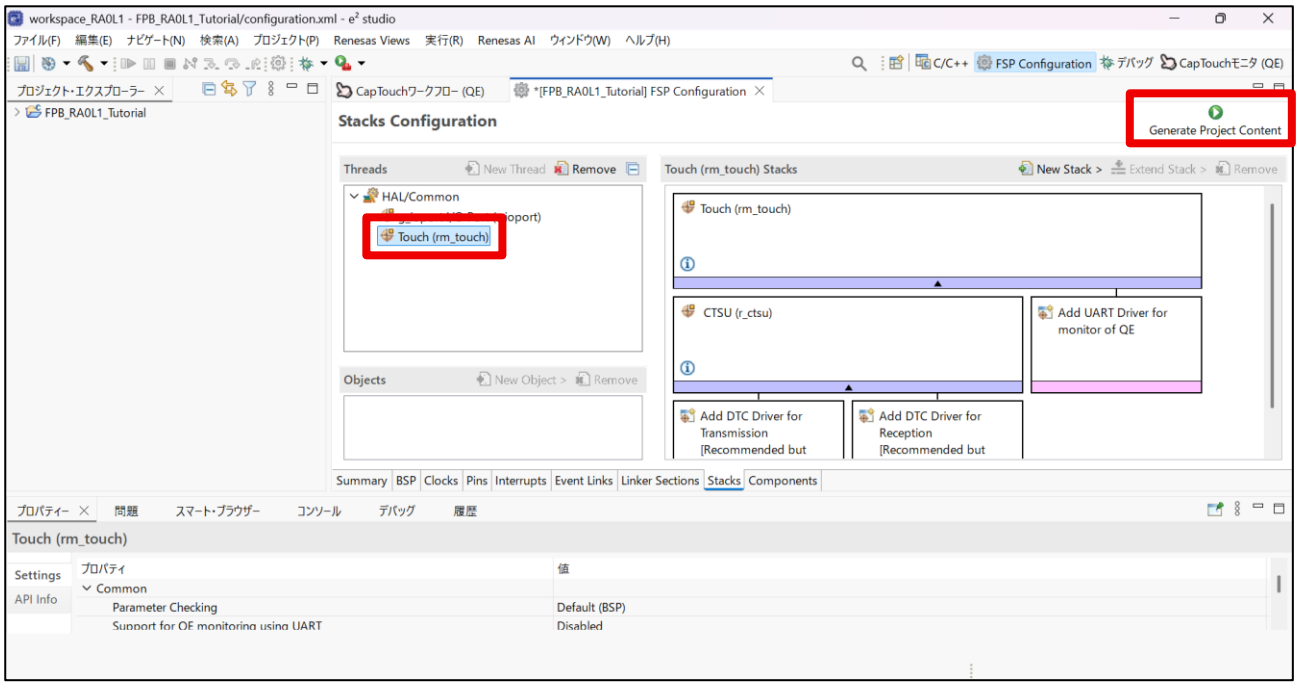


図 4-6. Touch ドライバの設定

4.1.5 ピン設定の確認

1. FSP Configurator 画面の「Pins」タブを開き、「Pin Selection」→「Peripherals」→「HMI:CTSU」→「CTSU」を開きます。
「TS22」「TS23」「TSCAP」にチェックが入っていること(※)が確認出来たら「Generate Project Content」をクリックしてコード生成を行います。
※3.3 デバイス・ツール設定画面でボードを選択したことにより、自動的に設定されます。

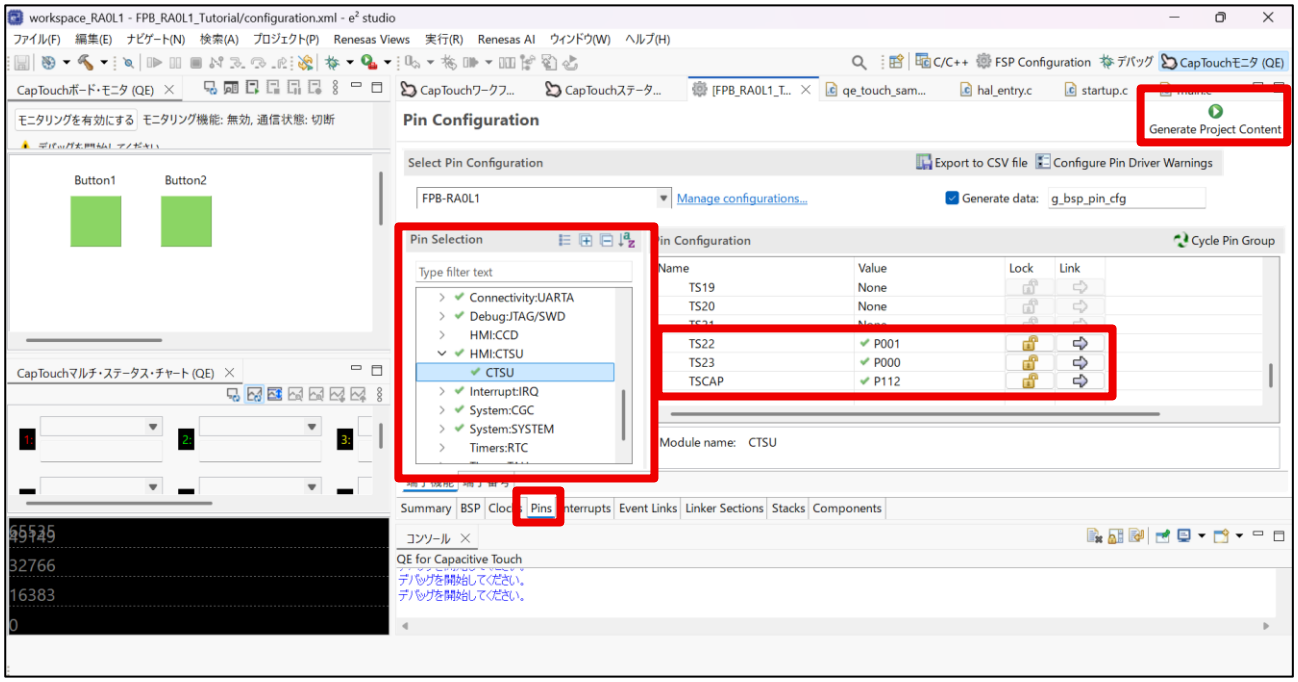


図 4-7. ピン設定の確認

2. コード生成を行う前に設定を保存するポップアップが出るので「続行」で進めてください。

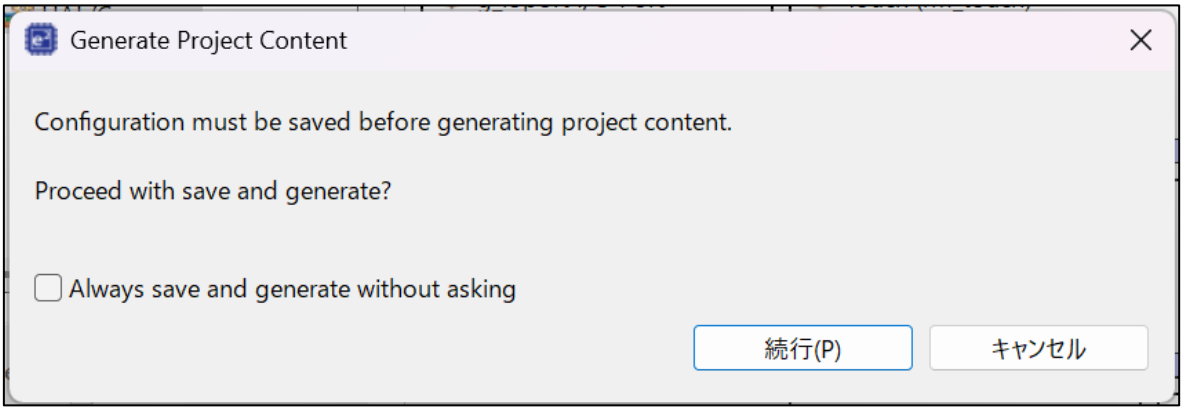


図 4-8. ピン設定の確認

4.2 QE Touch を使ったタッチセンサのチューニング

4.2.1 静電容量タッチの準備

(1) プロジェクトの選択

e² studio のメニューバーから「Renesas Views」→「Renesas QE」→「CapTouch ワークフロー」をクリックします。

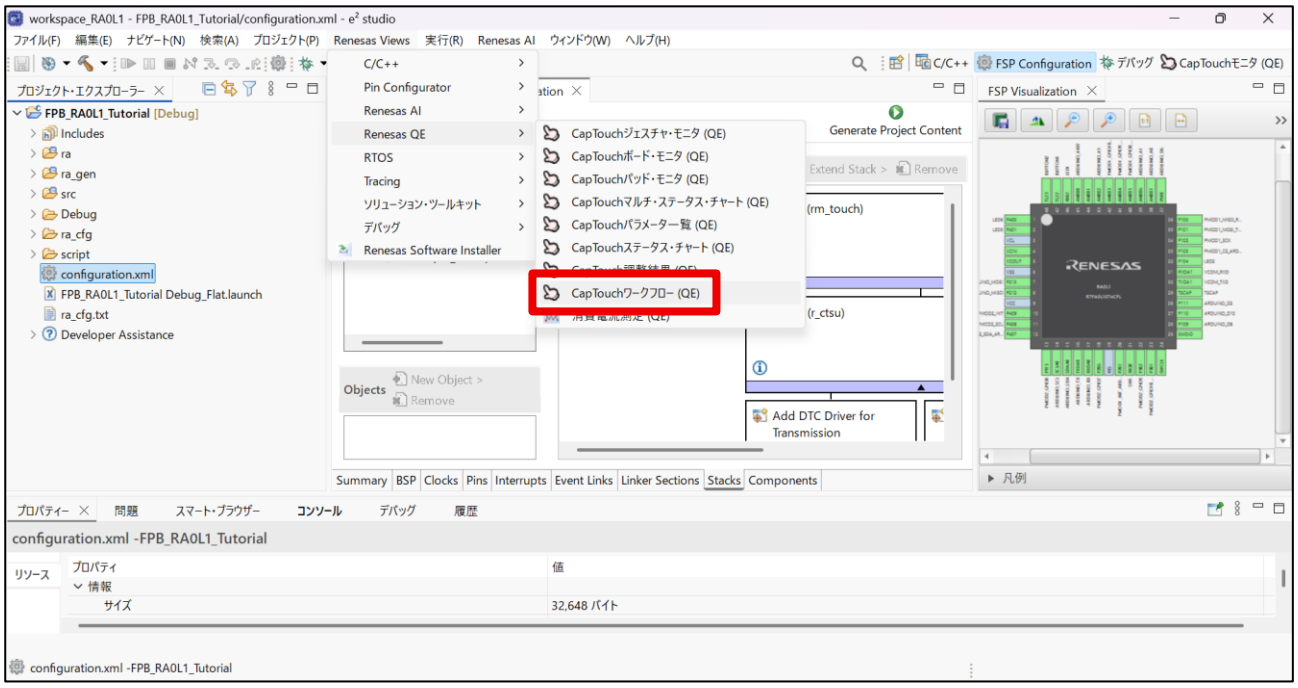


図 4-9. CapTouch ワークフローの選択

「プロジェクトの選択」から今回のプロジェクトを選択します。

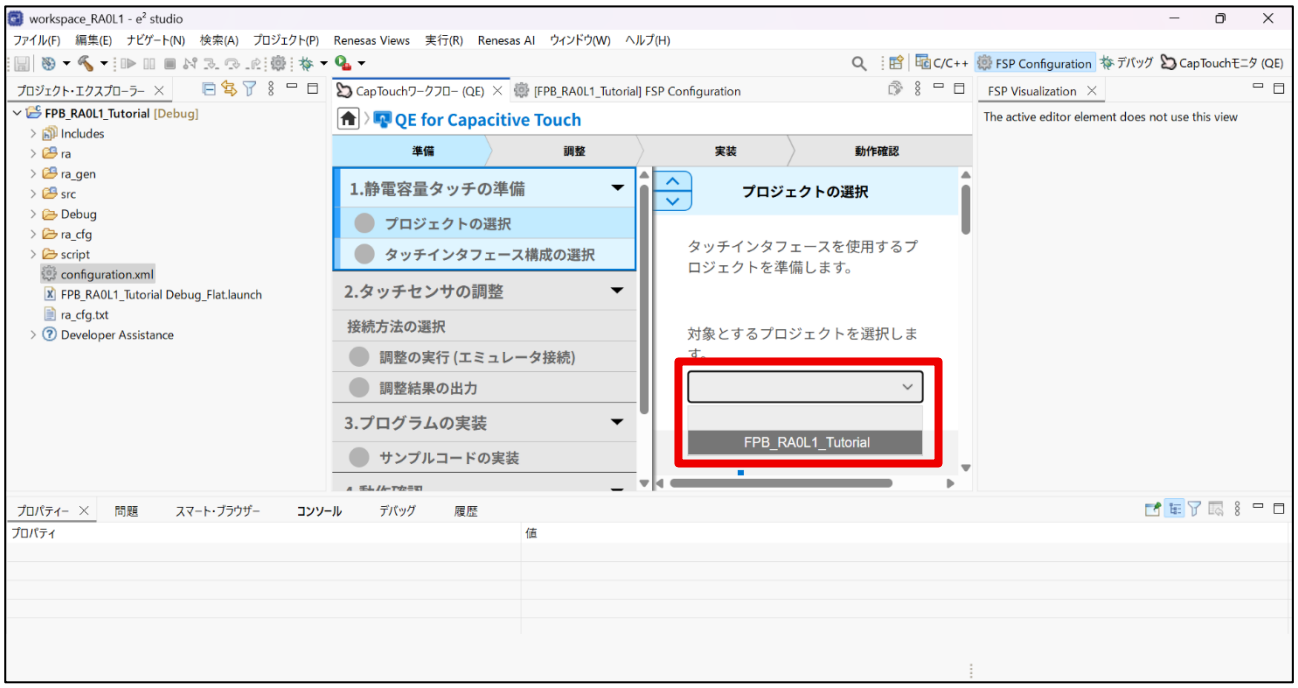


図 4-10. プロジェクトの選択

(2) タッチインタフェース構成の選択

「タッチインタフェース構成の選択」から「タッチインタフェース構成の新規作成」をクリックします。

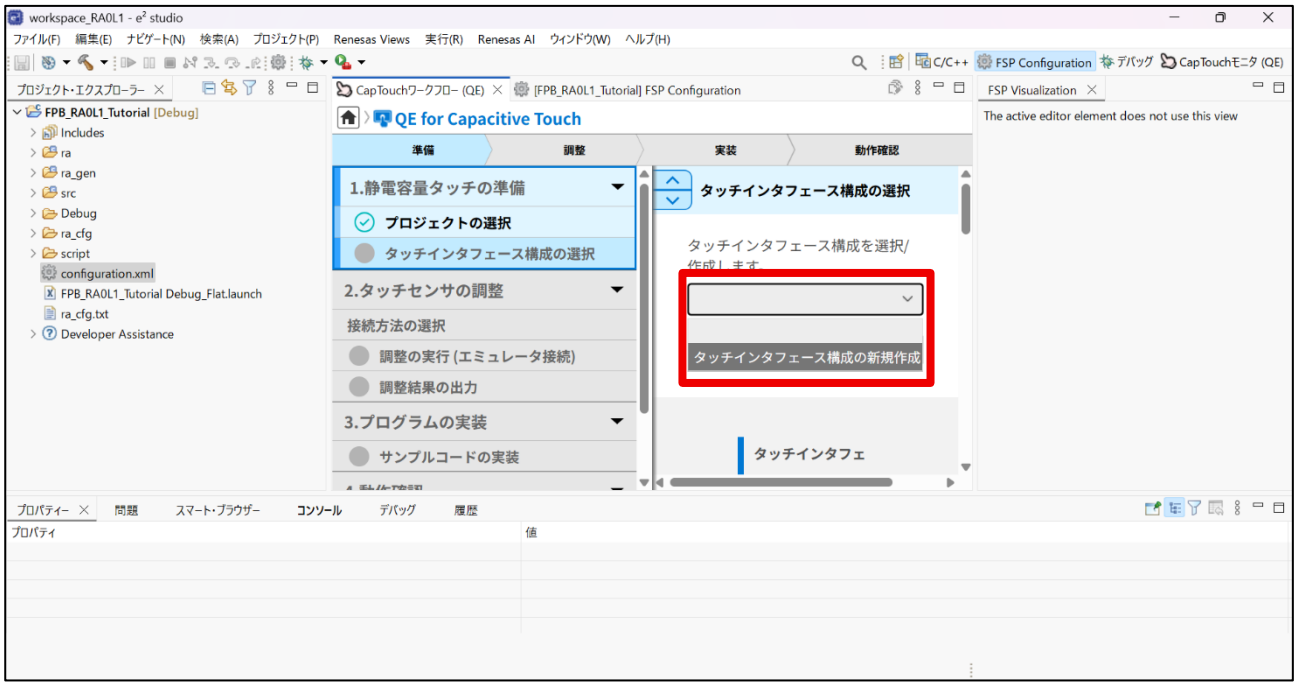


図 4-11. タッチインタフェース構成の選択

「タッチインタフェース構成の作成」画面がでます。右側のタッチ I/F の欄から使用するインタフェースを選択し、赤枠内のフィールドでクリックすることで構成を設置できます。

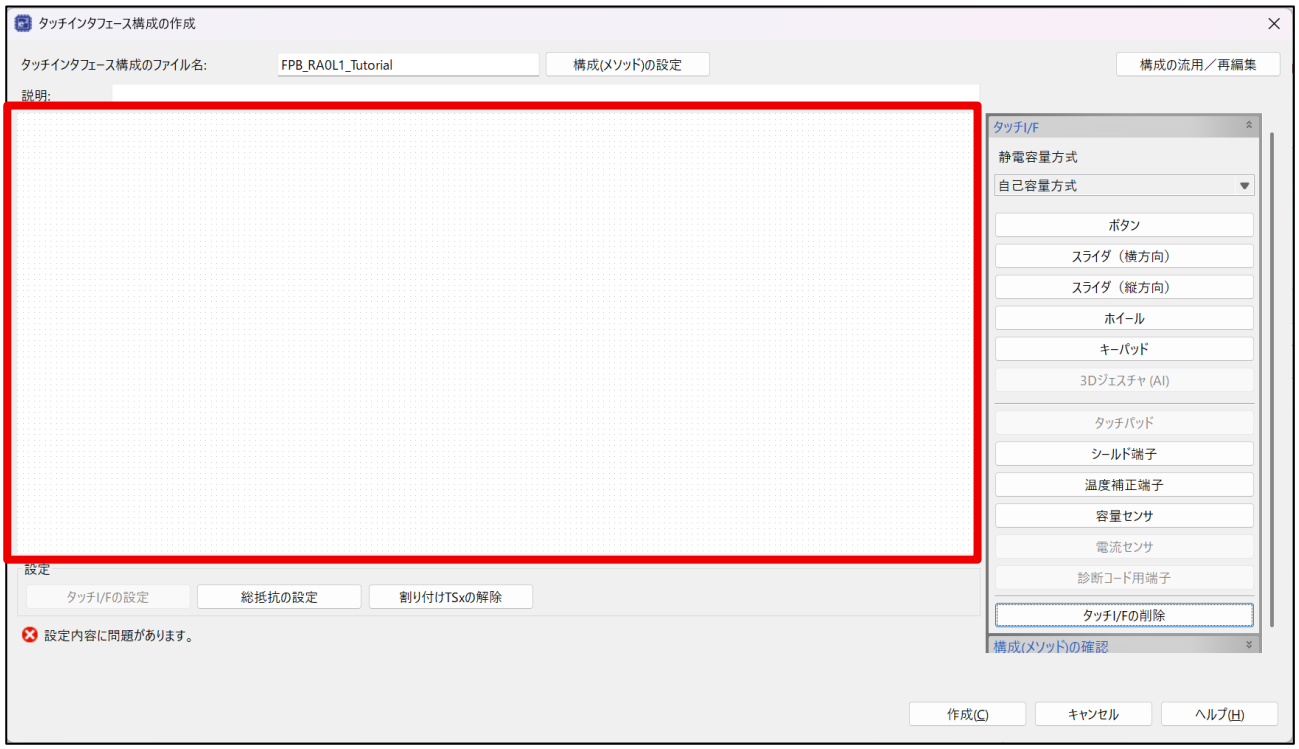


図 4-12. タッチインタフェース構成の選択

今回はボタンを 2 つ使用しますので右側のタッチ I/F からボタンを選択しフィールド上に 2 つ設置します。

ボタンを 2 つ設置した後、再度タッチ I/F のボタンを選択するか esc キーを押して選択解除します。

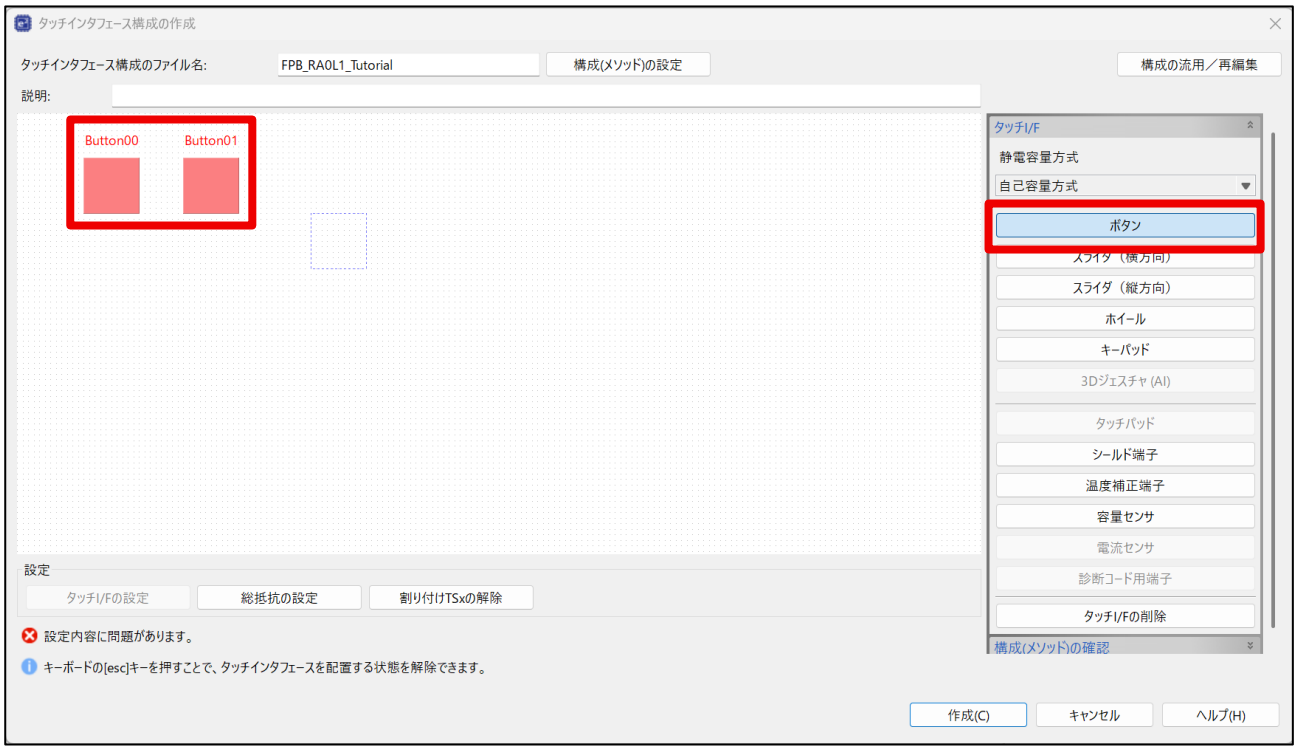


図 4-13. タッチインタフェース構成の選択

設置したボタンをダブルクリックするとタッチインタフェースの設定ポップアップが表示されます。

1 つ目のタッチインタフェースの設定を下記のように設定してください。

- 名前：Button1
  ※ここで記述した名前は後述のサンプルプログラムのマクロ名の一部となります。
- タッチセンサ：TS22

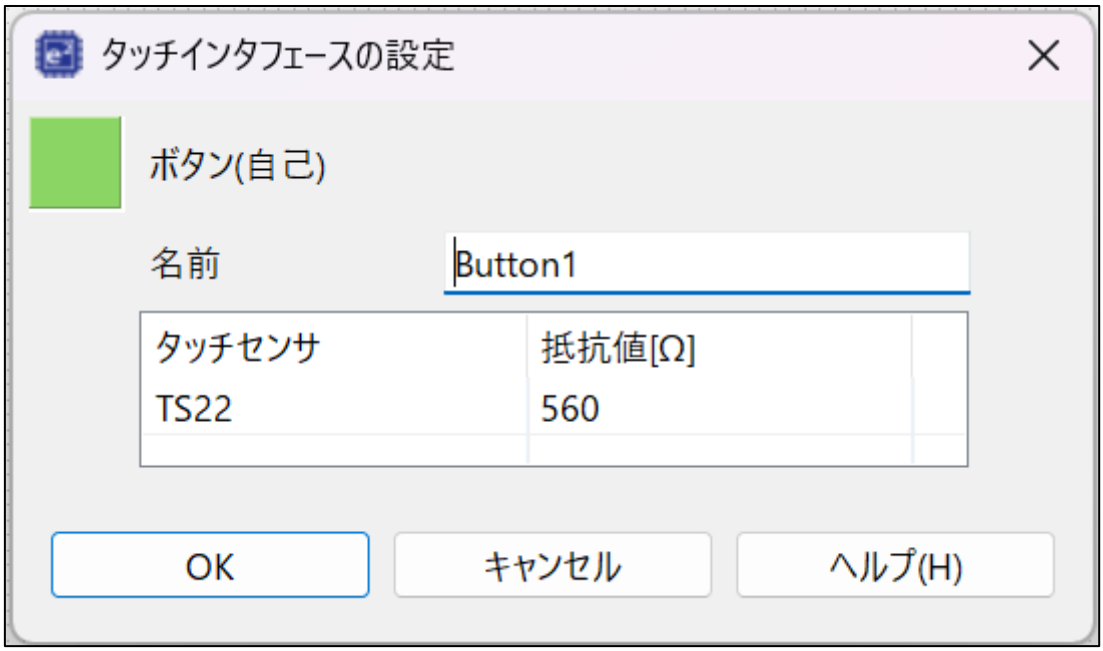


図 4-14. タッチインタフェース構成の選択

2 つ目のタッチインタフェースの設定を下記のように設定してください。

- 名前：Button2
- タッチセンサ：TS23

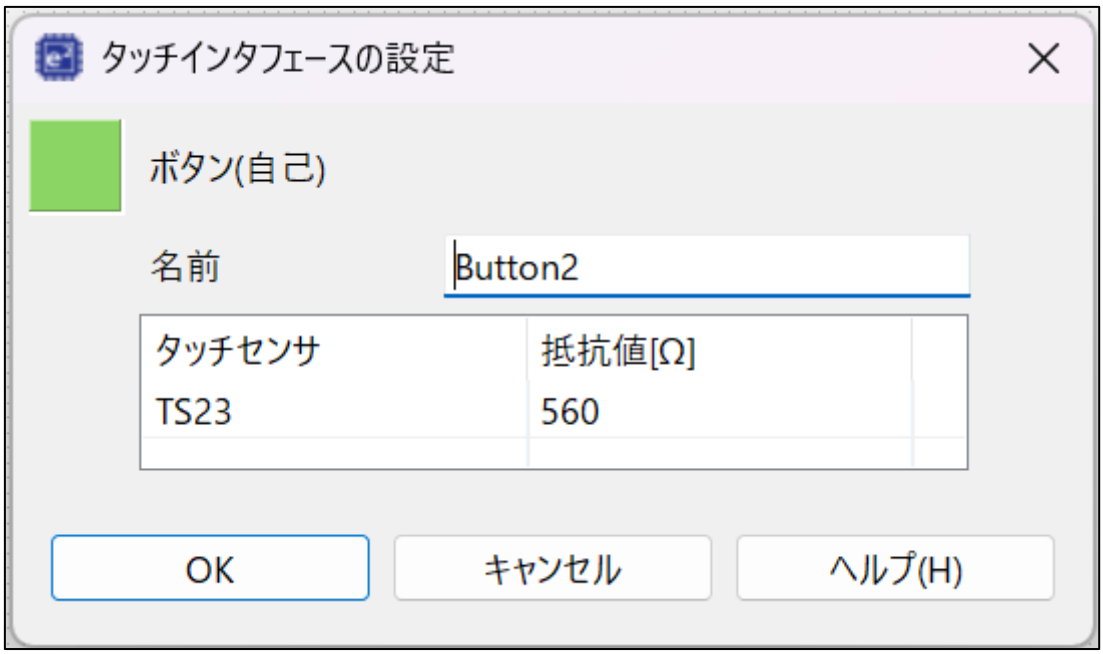


図 4-15. タッチインタフェース構成の選択

使用する TS 端子は、ボードのマニュアルや回路図を確認して特定できます。

ボタンの位置は「図 2-1. ボード上の LED 位置」を確認してください。

表 4.1 FPB-RA0L1 静電容量式タッチボタンの制御ポート

部品番号	RA MCU 制御ポート
Touch Button1	P001/TS22
Touch Button2	P000/TS23

タッチボタン設定が完了すると緑色表記になります。左下のエラーメッセージに何も表示されなければ設定完了です。「作成」をクリックしてタッチインタフェース構成を終了します。

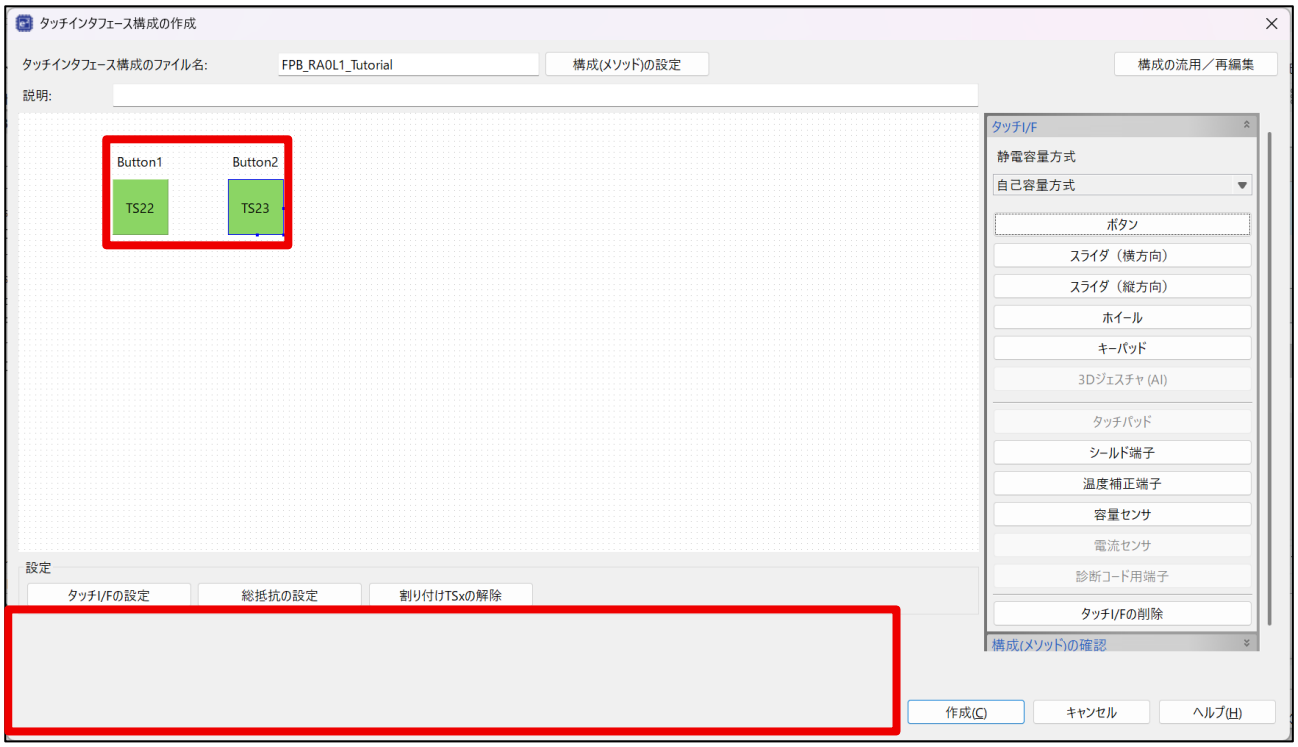


図 4-16. タッチインタフェース構成の選択

4.2.2 タッチセンサの調整

(1) 調整の実行 (エミュレータ接続)

付属の USB ケーブルでボードと PC を繋ぎます。その後、「調整を開始する」をクリックします。

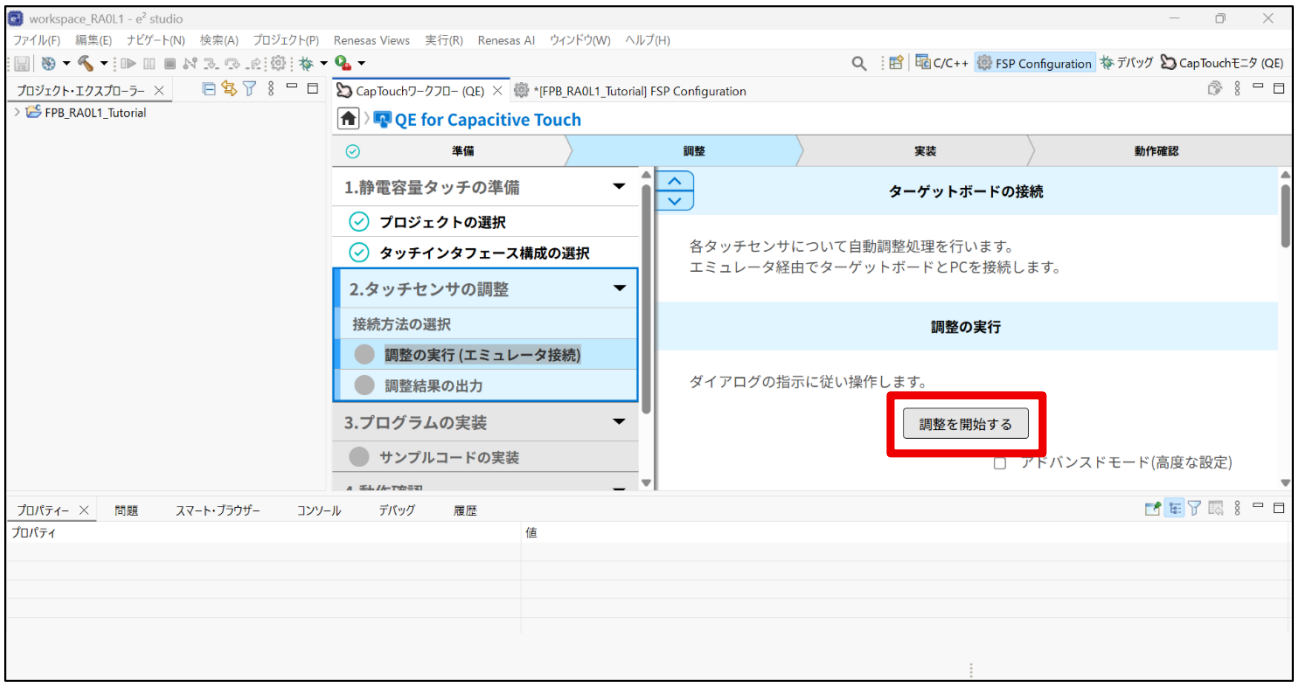


図 4-17. 調整の実行 (エミュレータ接続)

使用する周波数は 32MHz に設定し OK をクリックします。

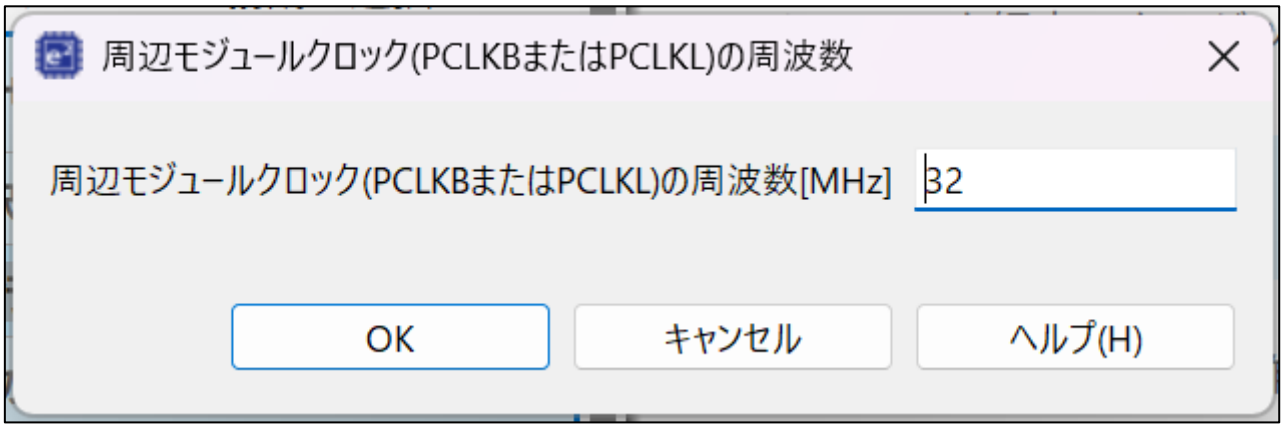


図 4-18. 調整の実行 (エミュレータ接続)

保管するリソースの選択する画面では今回のプロジェクトの configuration.xml にチェックして「OK」を押します。

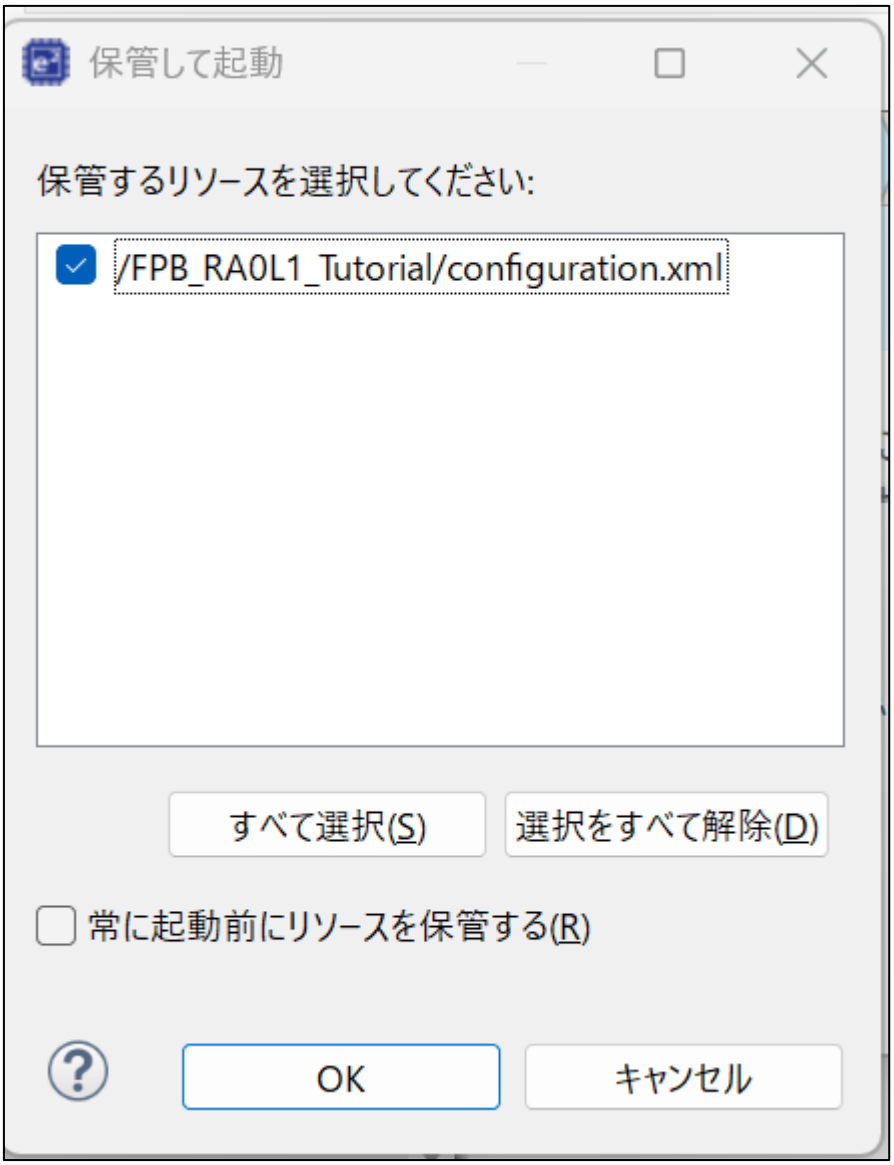


図 4-19. 調整の実行 (エミュレータ接続)

パースペクティブ切り替えの確認表示のポップアップが開きますので、「切り替え」をクリックします。
※「いいえ」をクリックしても問題ありません。

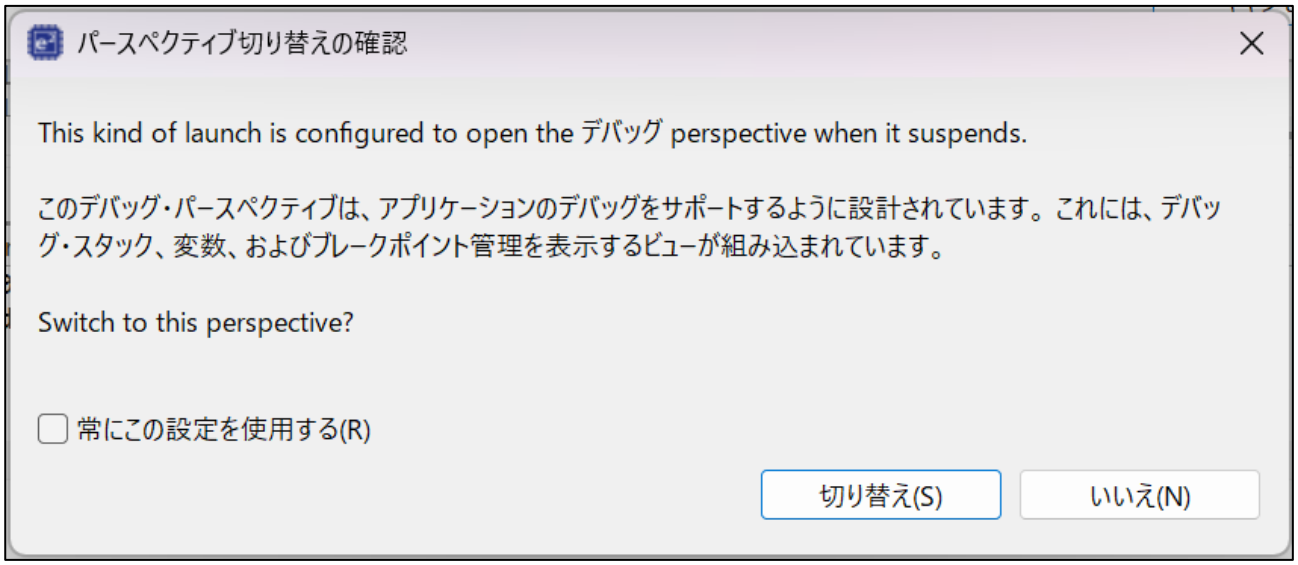


図 4-20. 調整の実行（エミュレータ接続)

タッチセンサの感度調整を行うポップアップが出ます。

このポップアップでは Button1 の感度調整を行うのでボードの Button1 を指で触れます。
数値が安定したらキーボードで何かのキーを押します。Button1 の感度調整は完了です。

【Button1 を押して無い時】

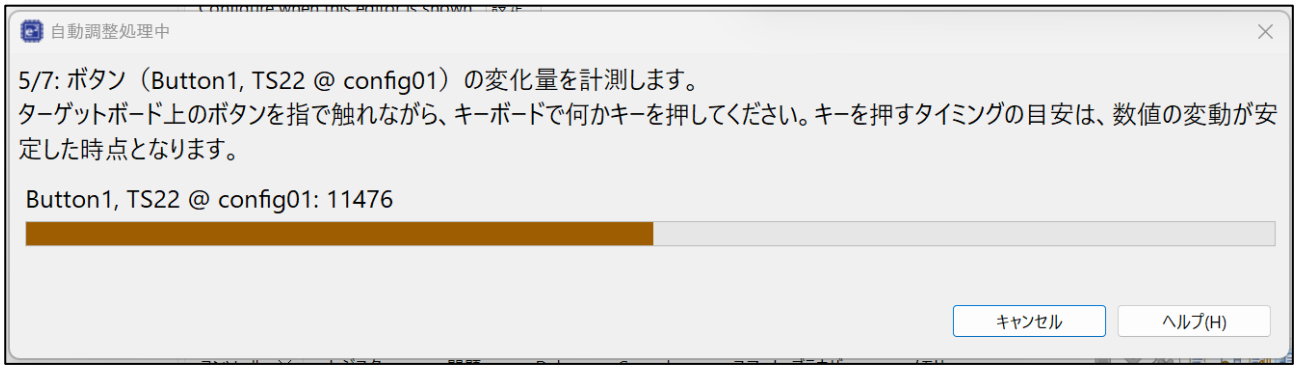


図 4-21. 調整の実行（エミュレータ接続)

【Button1 を押している時】

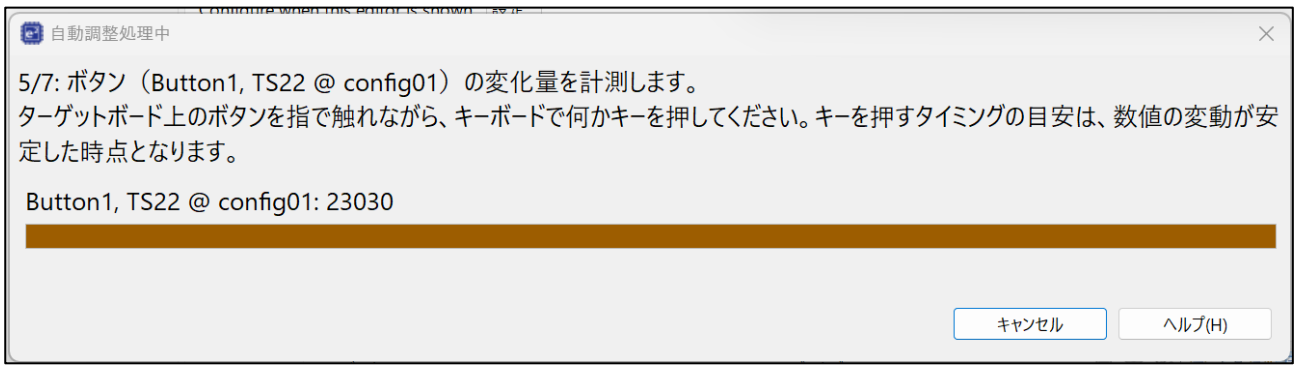


図 4-22. 調整の実行（エミュレータ接続)

次に Button2 の感度調整を行うポップアップが表示されるので、Button1 同様に感度調整を行います。

【Button2 を押して無い時】

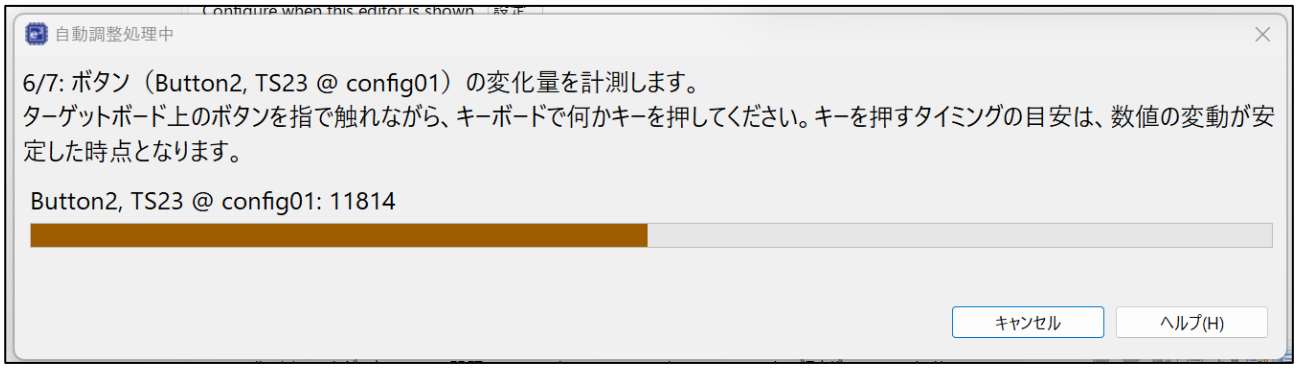


図 4-23. 調整の実行（エミュレータ接続）

【Button2 を押している時】

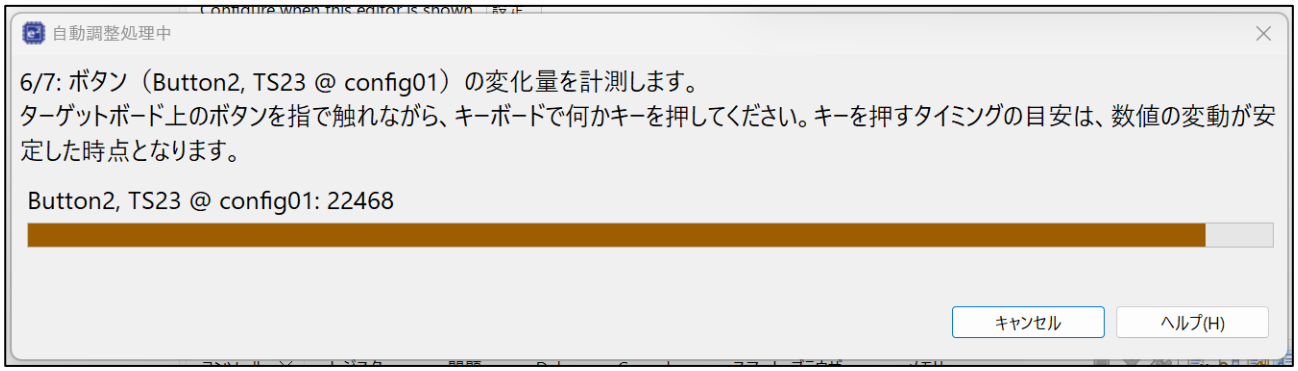


図 4-24. 調整の実行（エミュレータ接続）

感度調整が完了するとチューニング結果のポップアップが表示されます。

オーバーフロー欄または警告／エラー欄を確認し、メッセージがある場合、リトライ対象の選択にチェックしてチューニングをやり直してください。

メッセージがない場合、正常に調整出来ていますので「調整処理の継続」をクリックします。

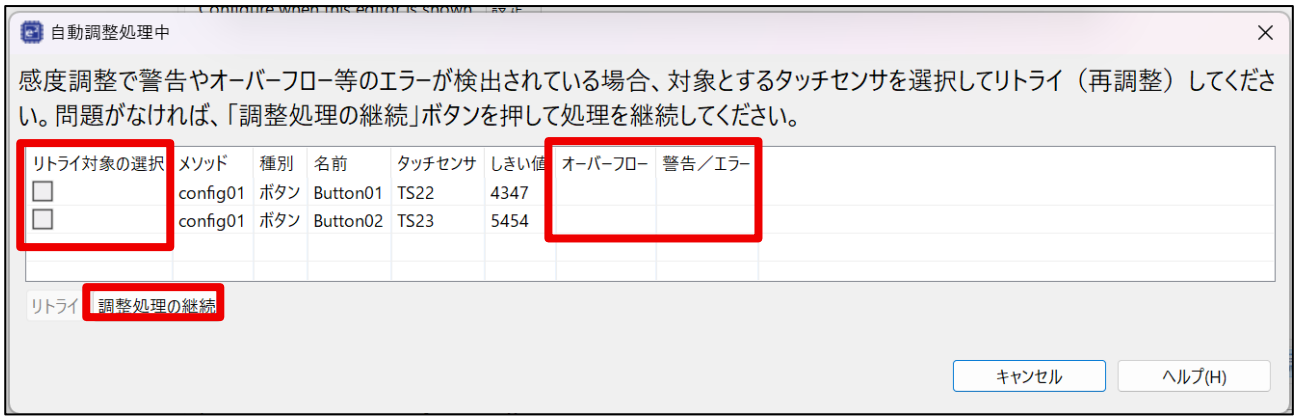


図 4-25. 調整の実行 (エミュレータ接続)

(2) 調整結果の出力

調整結果の出力から「ファイルを出力する」をクリックします。

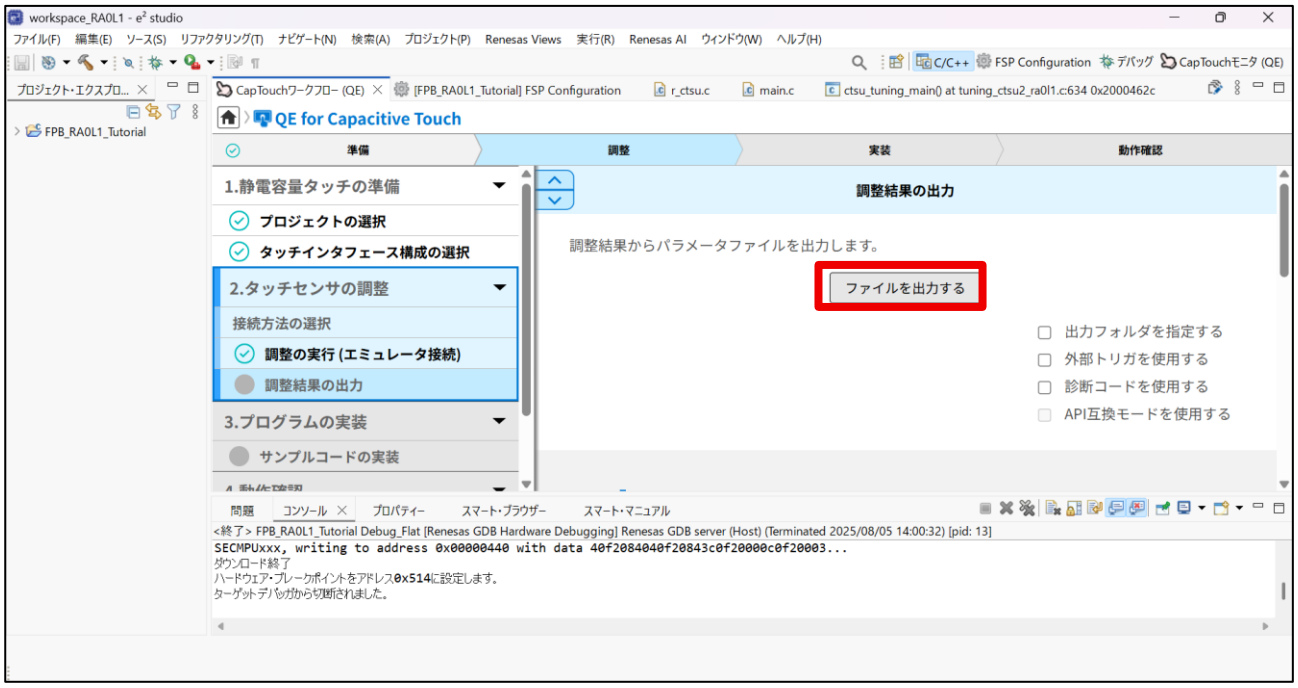


図 4-26. 調整結果の出力

CapTouch ワークフローの調整結果の出力にチェックが付いていることを確認して、調整作業は完了となります。

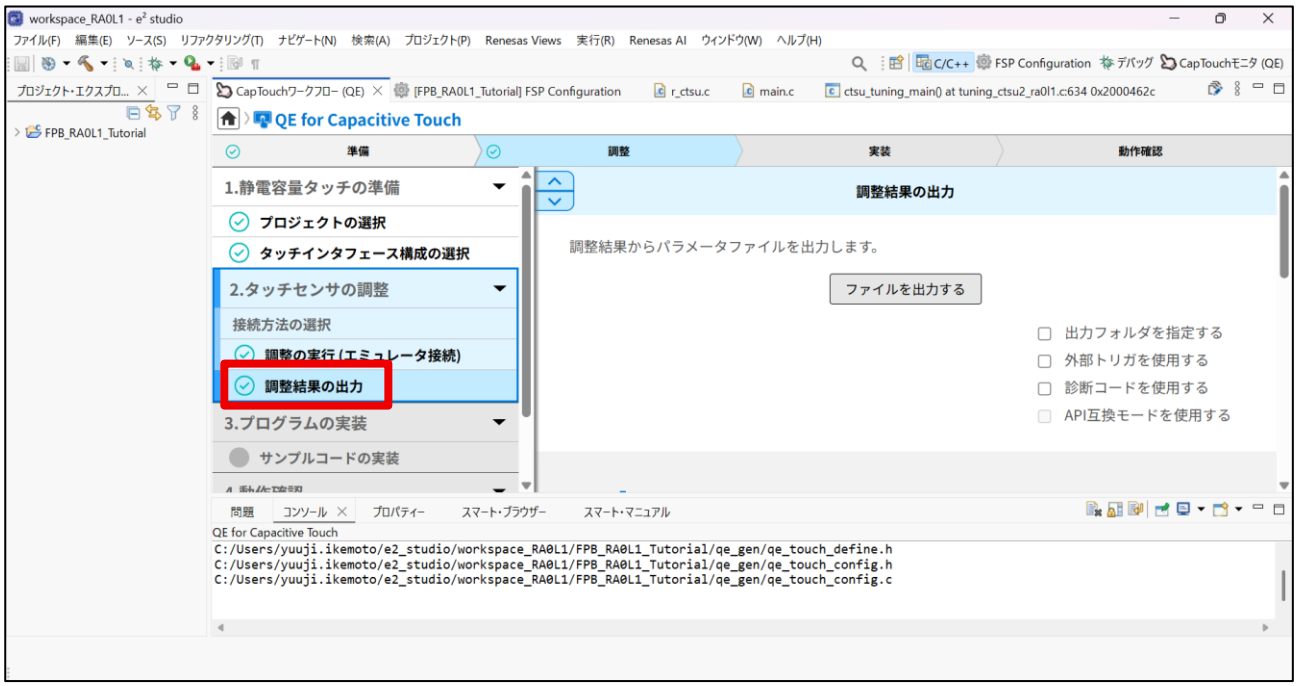


図 4-27. 調整結果の出力

4.2.3 プログラムの実装

(1) サンプルコードの実装

サンプルコードの実装から「例を表示する」をクリックします。

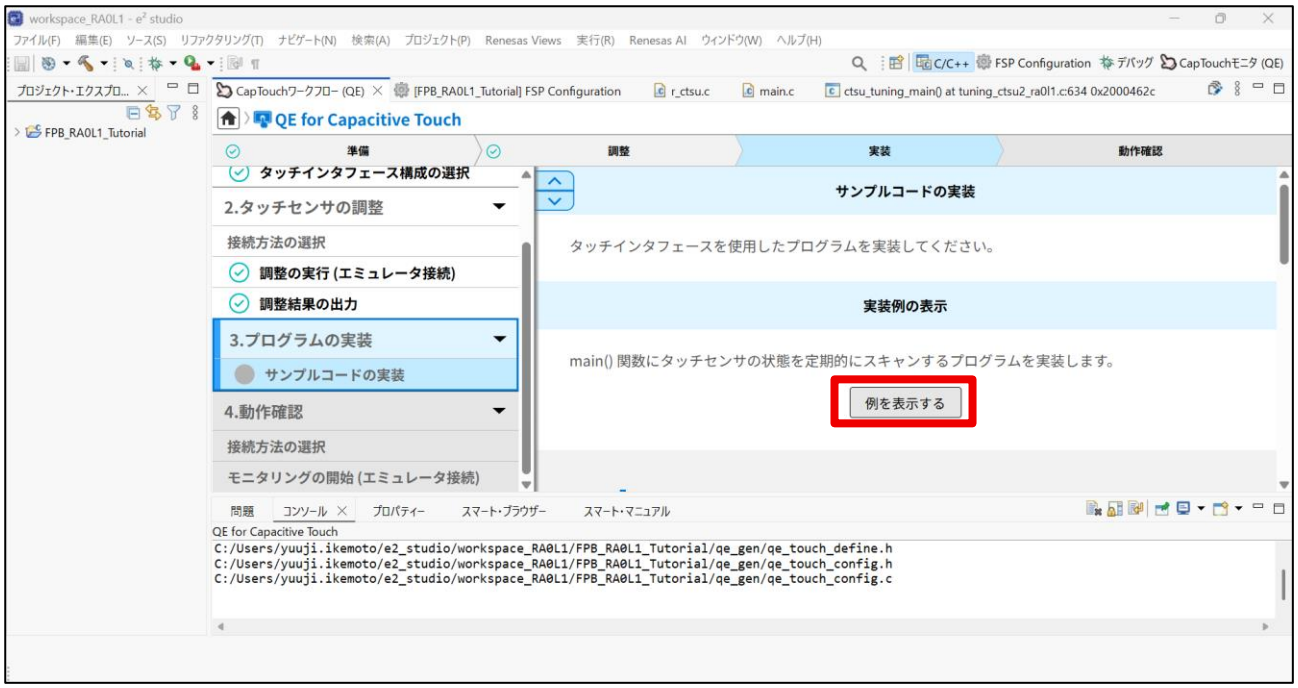


図 4-28. サンプルコードの実装

サンプルコードが表示されるので「ファイルに出力」をクリックし、続けて「OK」をクリックします。

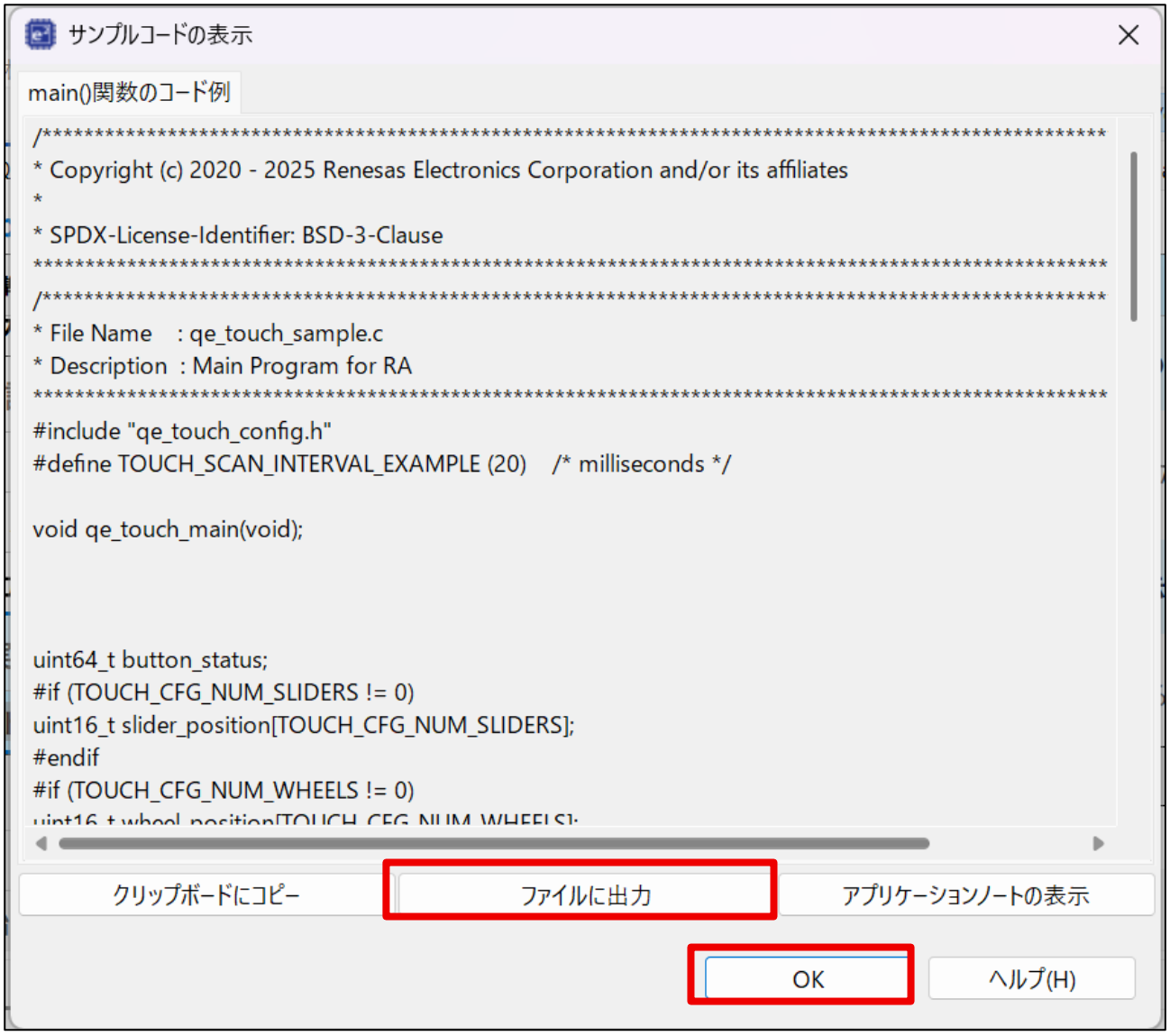


図 4-29. サンプルコードの実装

プロジェクト・エクスプローラーから「FPB_RA0L1_Tutorial」→「qe_gen」→「qe_touch_sample.c」が確認出来れば出力は完了です。

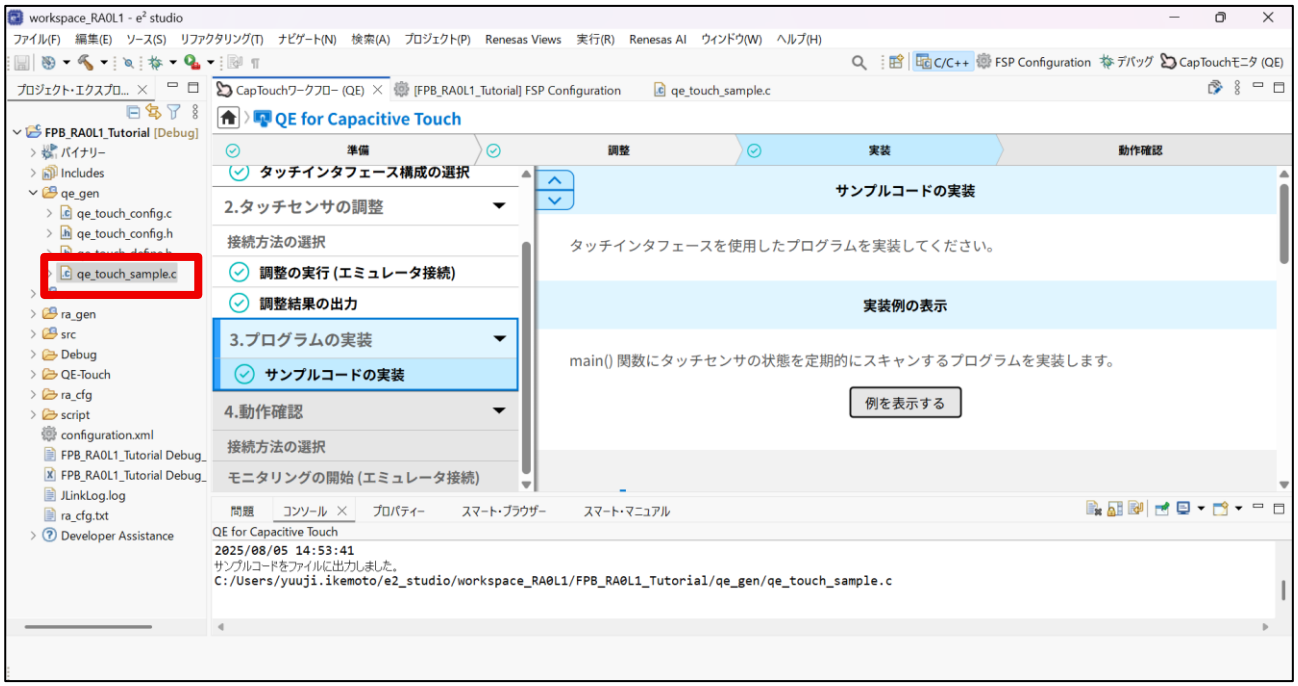


図 4-30. サンプルコードの実装

(2) サンプルコードの関数呼び出しの追加実装

プロジェクト・エクスプローラーから「FPB_RA0L1_Tutorial」→「src」→「hal_entry.c」を開き

hal_entry 関数に「qe_touch_sample.c」の中にある qe_touch_main 関数をコールするコードを追加します。

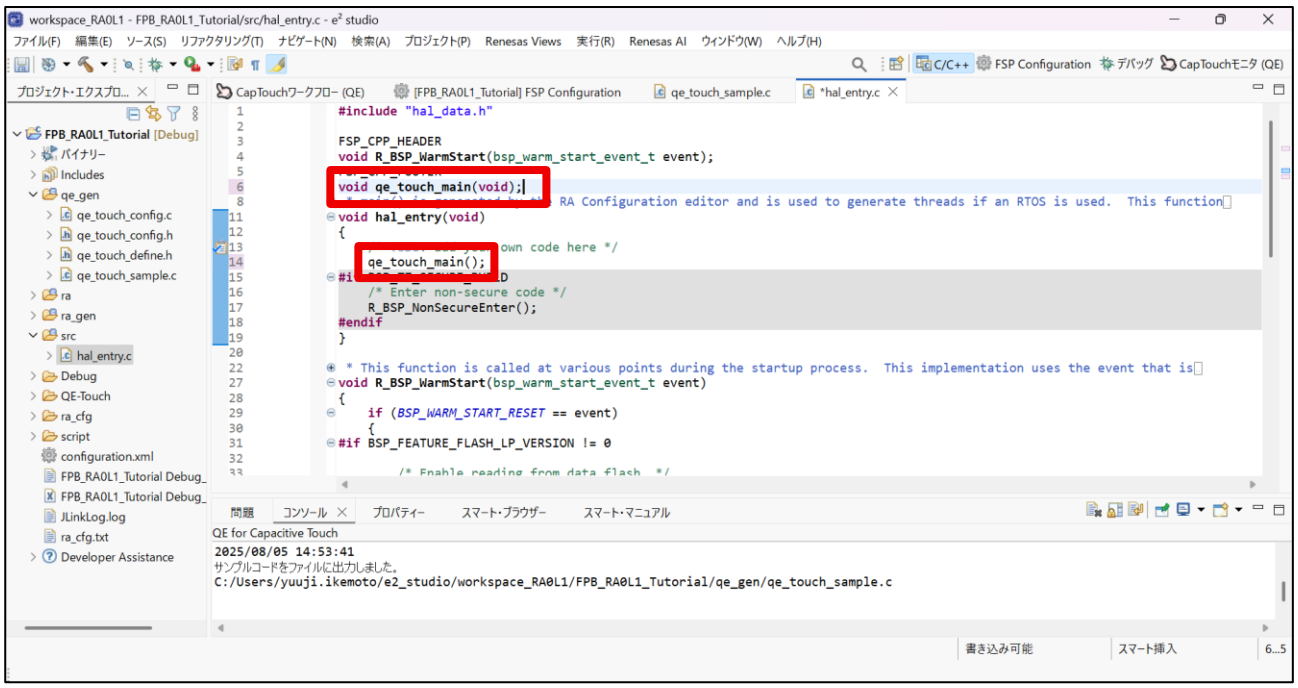


図 4-31. サンプルコードの実装

レンチアイコンをクリックするとビルドを開始します。

「コンソール」ウインドウにビルドログが出力されます。

最後に “Build Finished. 0 errors, “と表示されていればビルドが正常終了したことを意味します。

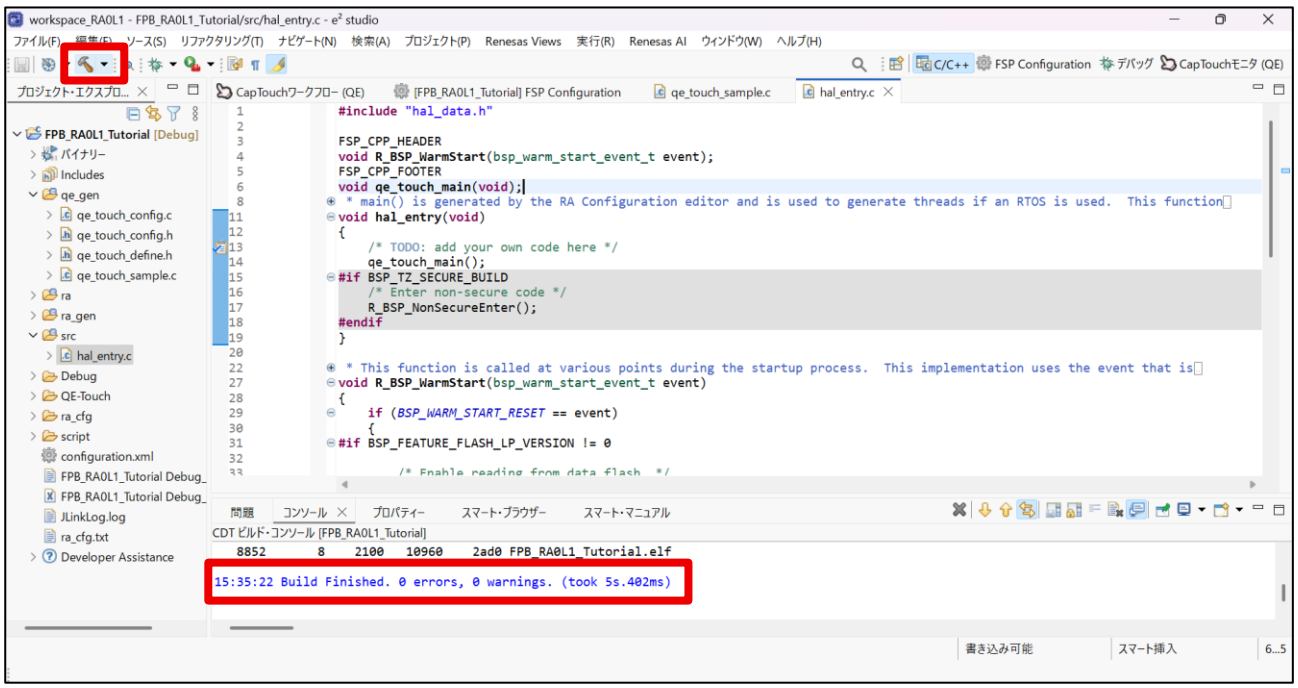


図 4-32. サンプルコードの実装

4.3 QE Touch を使ったタッチセンサのモニタリング

4.3.1 デバッグ設定と起動

(1) デバッグの構成

e² studio のメニューバーから「実行」→「デバッグの構成」をクリックします

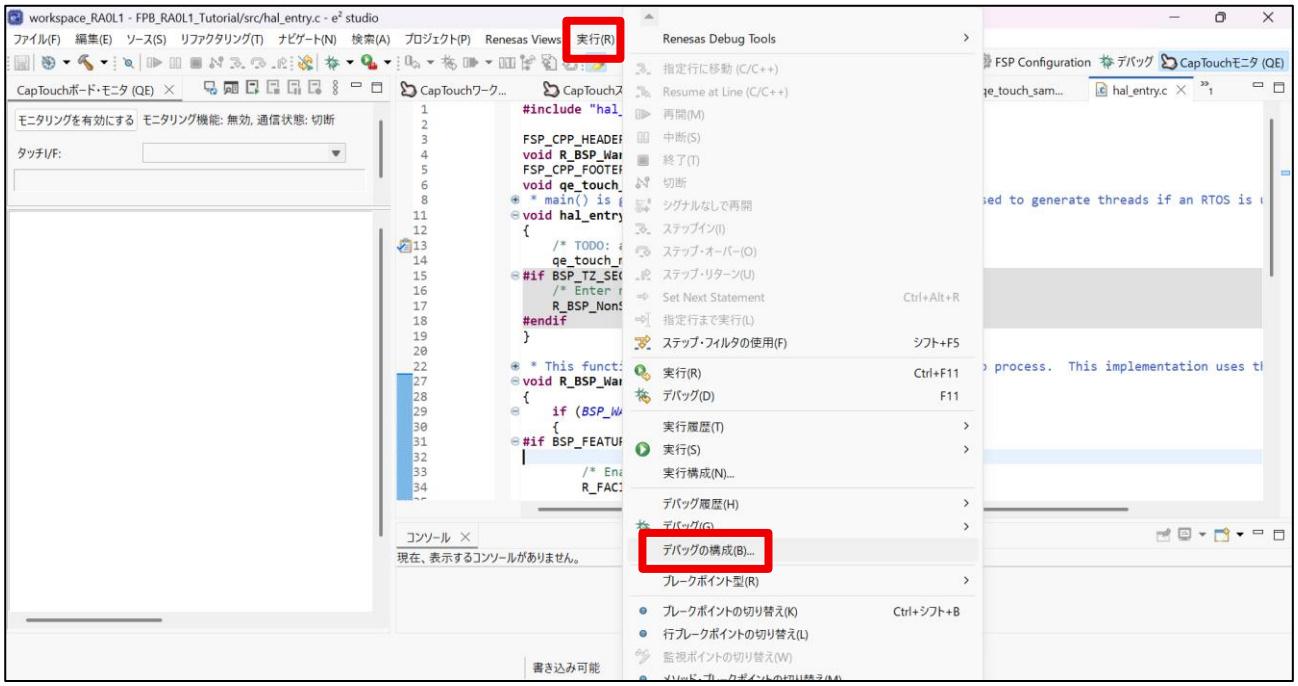


図 4-33. デバッグの構成

Debugger を開き、Debug hardware が「J-Link ARM」、Target Device が「R7FA0L107」であることを確認します。

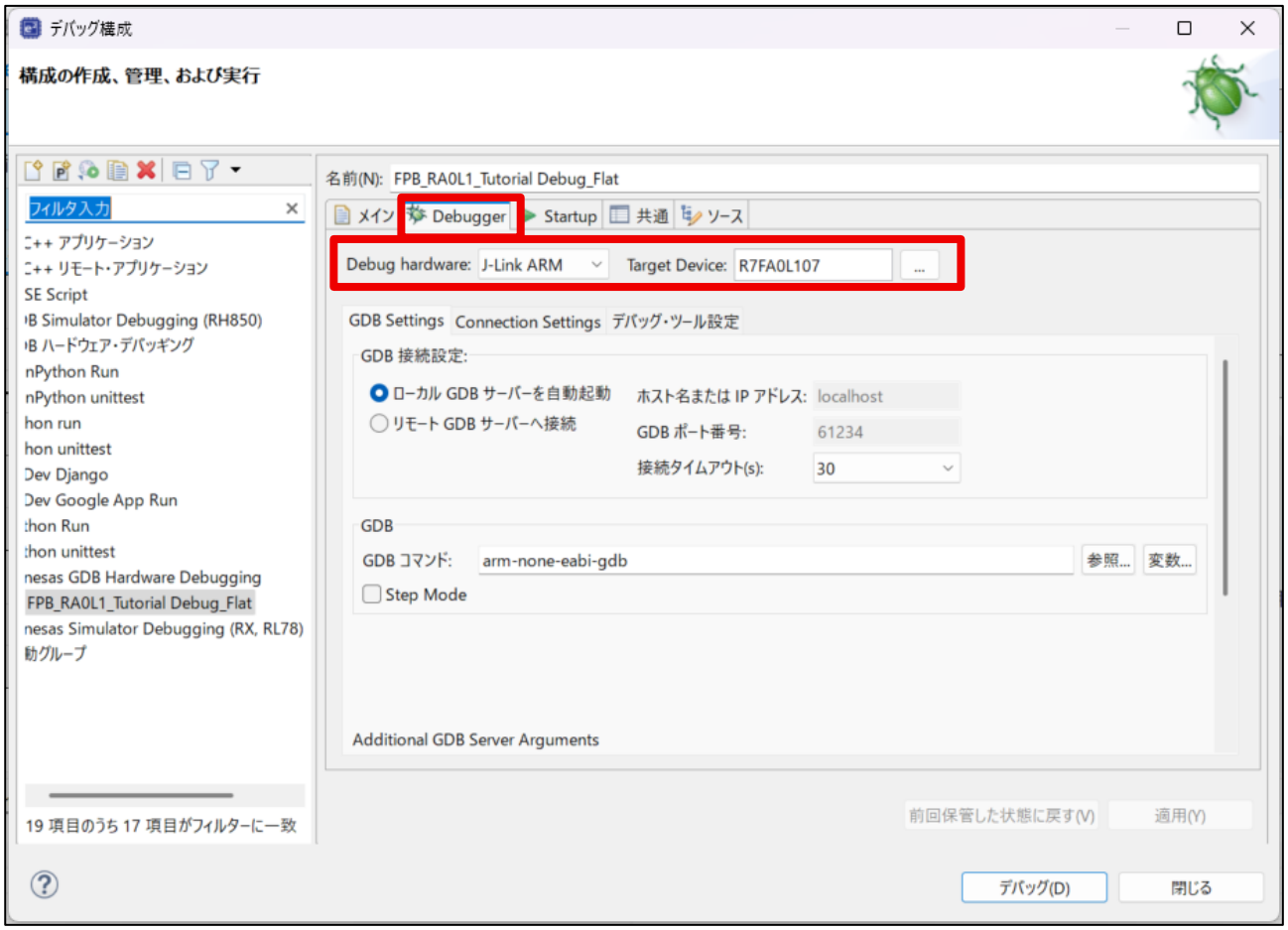


図 4-34. デバッグの構成

Startup タブを開き、ランタイム・オプションの「再開」にチェックを付けます。

デバッグの設定はこれで完了です。「適用」をクリック後、「デバッグ」をクリックします。

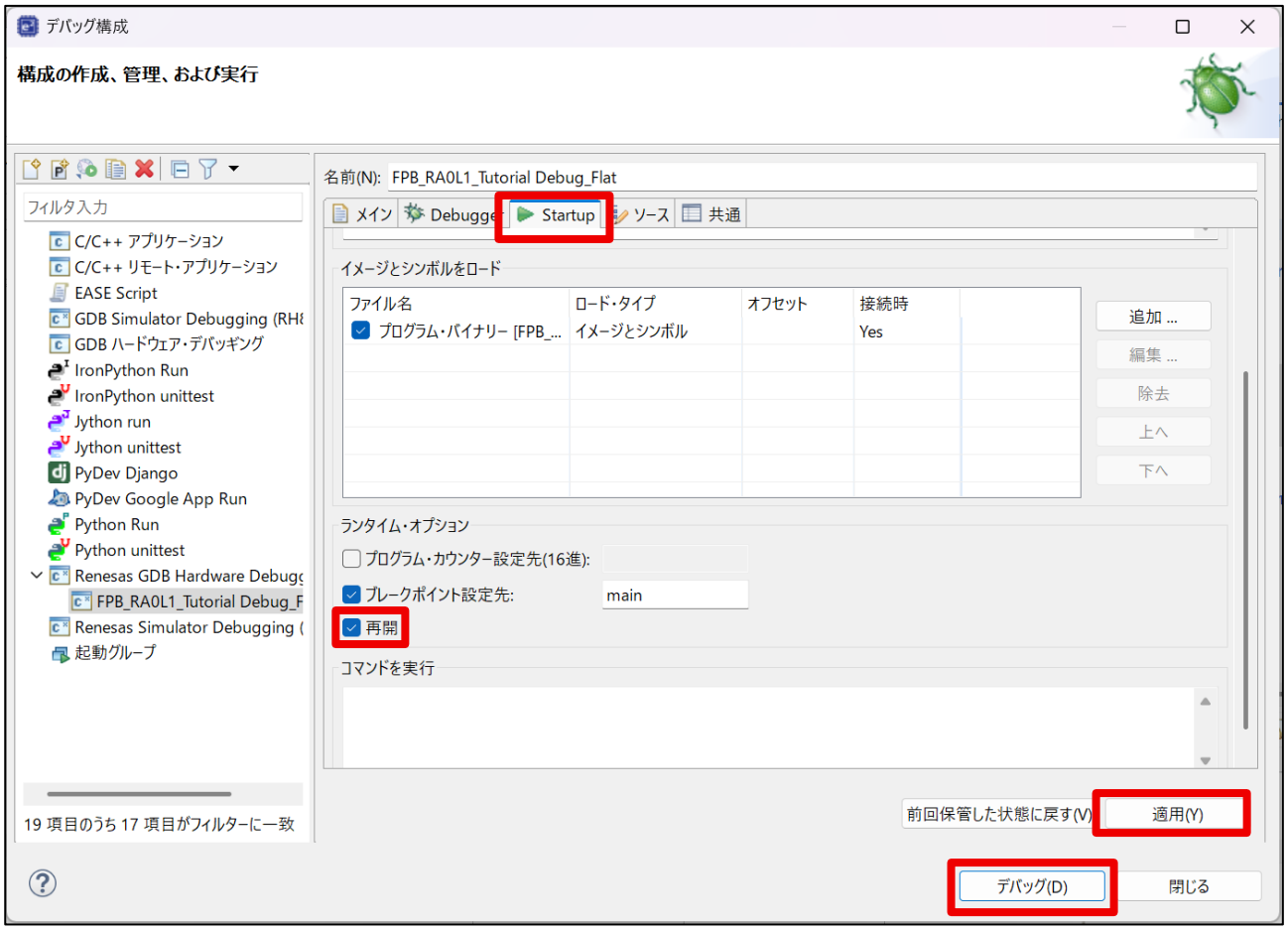


図 4-35. デバッグの構成

パースペクティブ切り替えを行うポップアップが出ます。切り替えをクリックして先に進んでください。

※「いいえ」をクリックしても問題ありません。

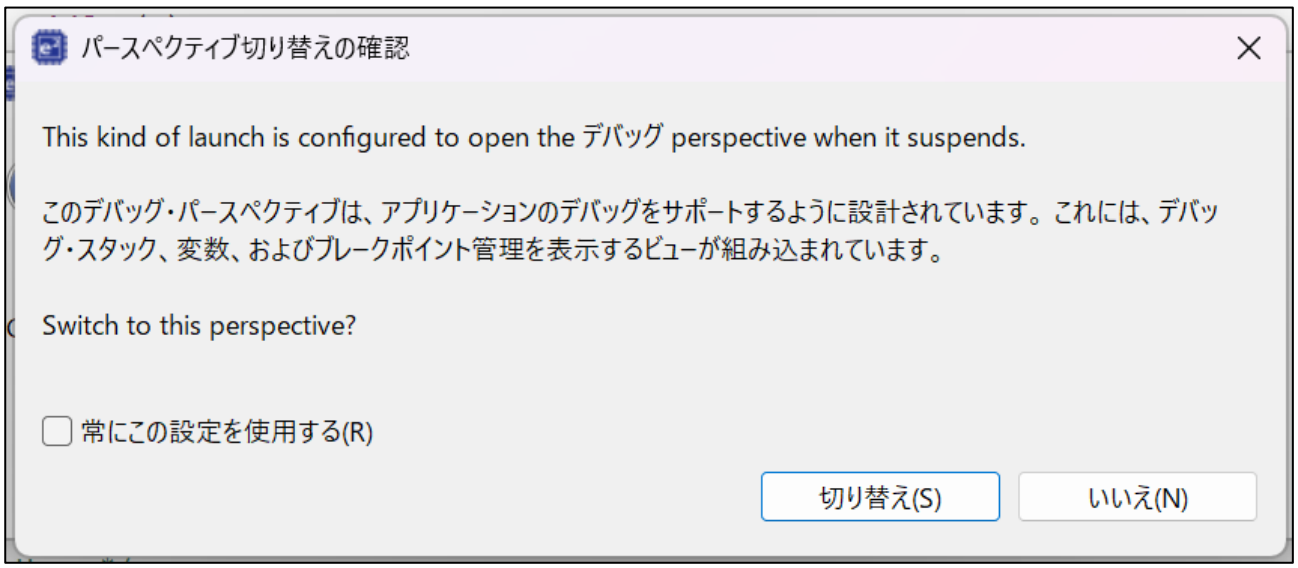


図 4-36. デバッグの構成

(2) 実行

デバッグ終了後プログラムが main 関数の先頭で一時停止します。(e² studio プロジェクトのデフォルトが main 関数で一時停止するよう設定されているためです。)

この状態は、SystemInit()が完了して、初期設定が完了した状態です。

再開ボタンを押して、プログラムを続行してください。

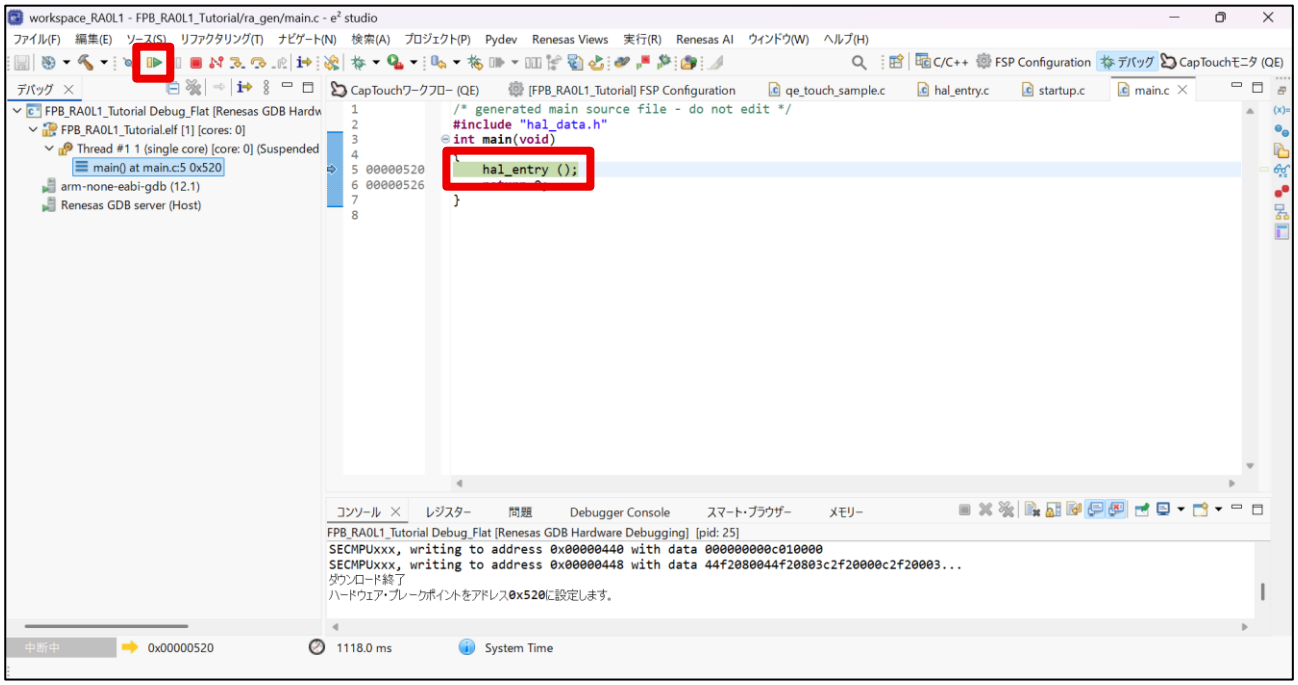


図 4-37. 実行

4.3.2 モニタリング

CapTouch ワークフローの動作確認を開き、モニタリングに使用する接続方法をエミュレータに設定します。

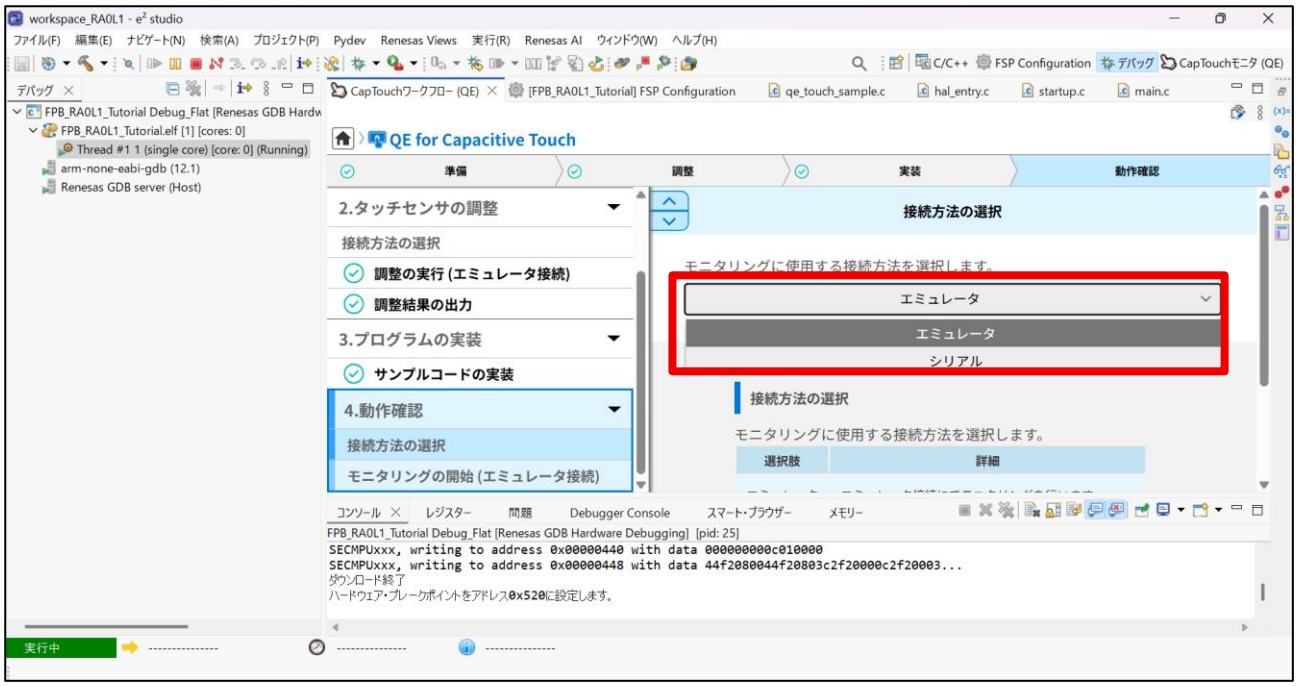


図 4-38. モニタリング接続方法の選択

CapTouch ワークフローのモニタリングの開始（エミュレータ接続）から「ビューを開く」をクリックします。

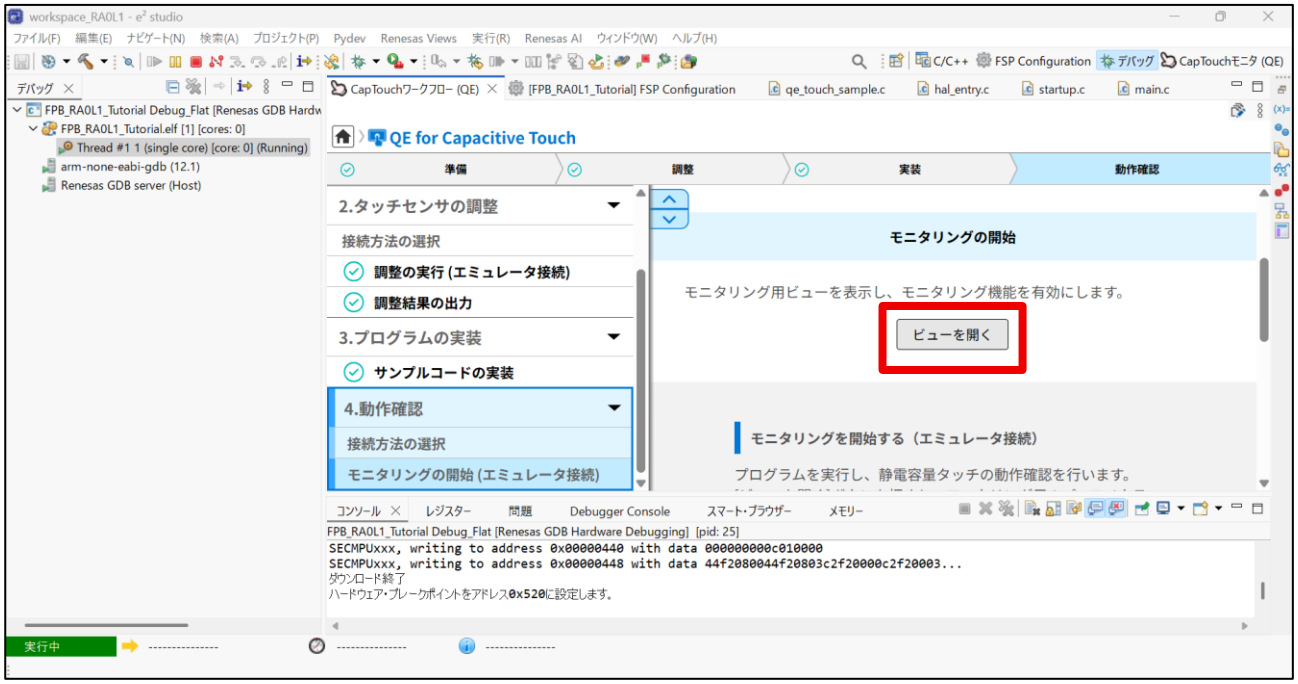


図 4-39. モニタリングの開始

「CapTouch ボード・モニタ」、「CapTouch ステータス・チャート」、「CapTouch マルチ・ステータス・チャート」画面が表示されることを確認します。

まだこの状態では「モニタリング機能：無効」になっています。

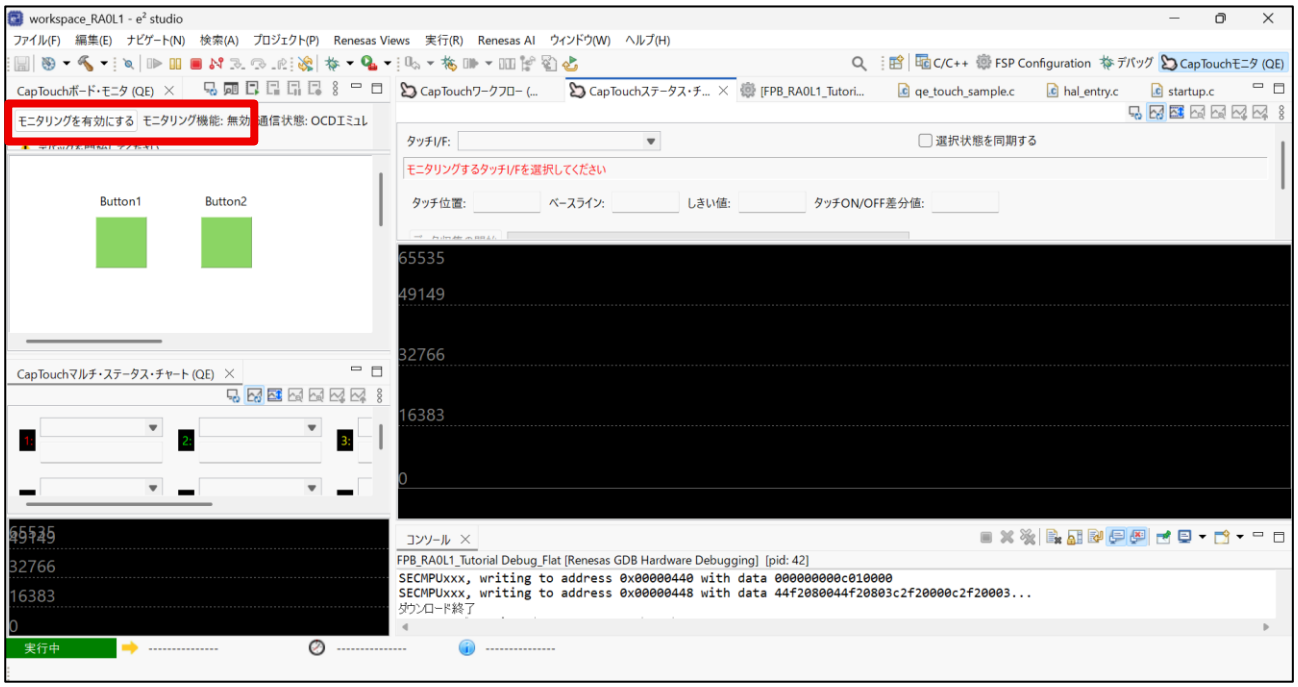


図 4-40. モニタリング機能：無効

この状態で「モニタリングを有効にする」をクリックすると「モニタリング機能：有効」になります。

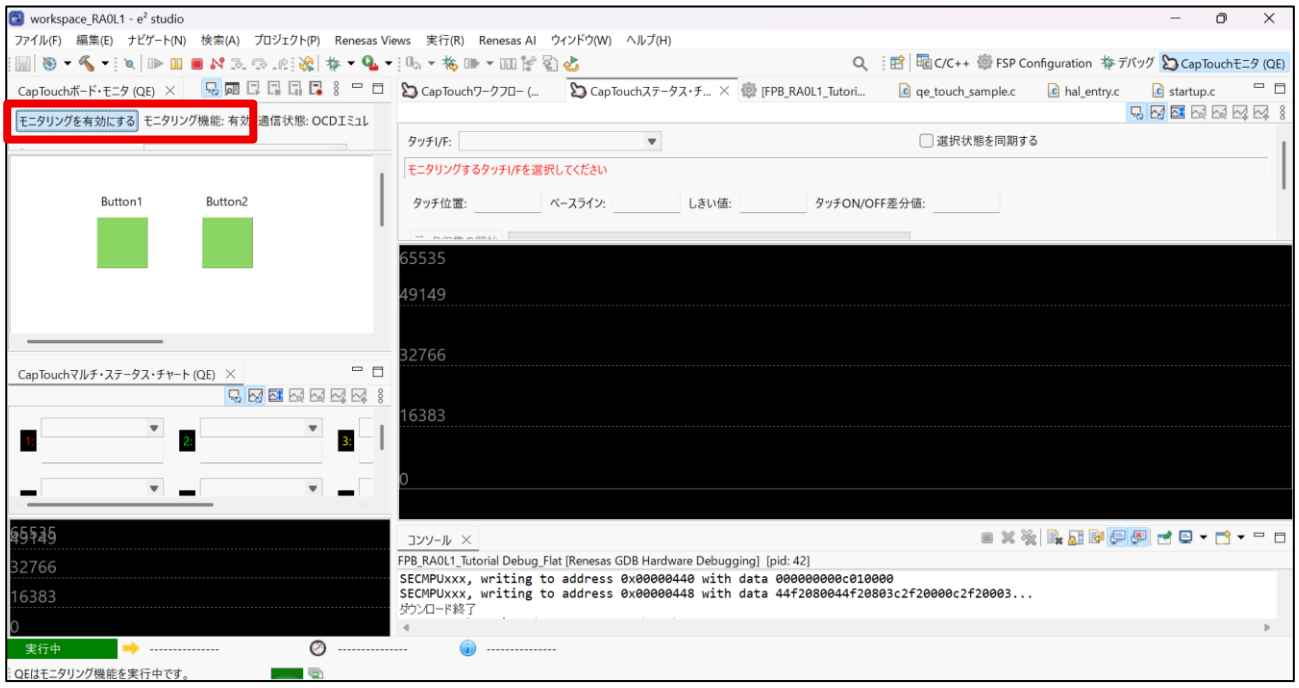


図 4-41. モニタリング機能：有効

CapTouch ボード・モニタではタッチセンサの ON/OFF を確認できます。

Button1 をタッチ検出したとき、指アイコンが表示されます。

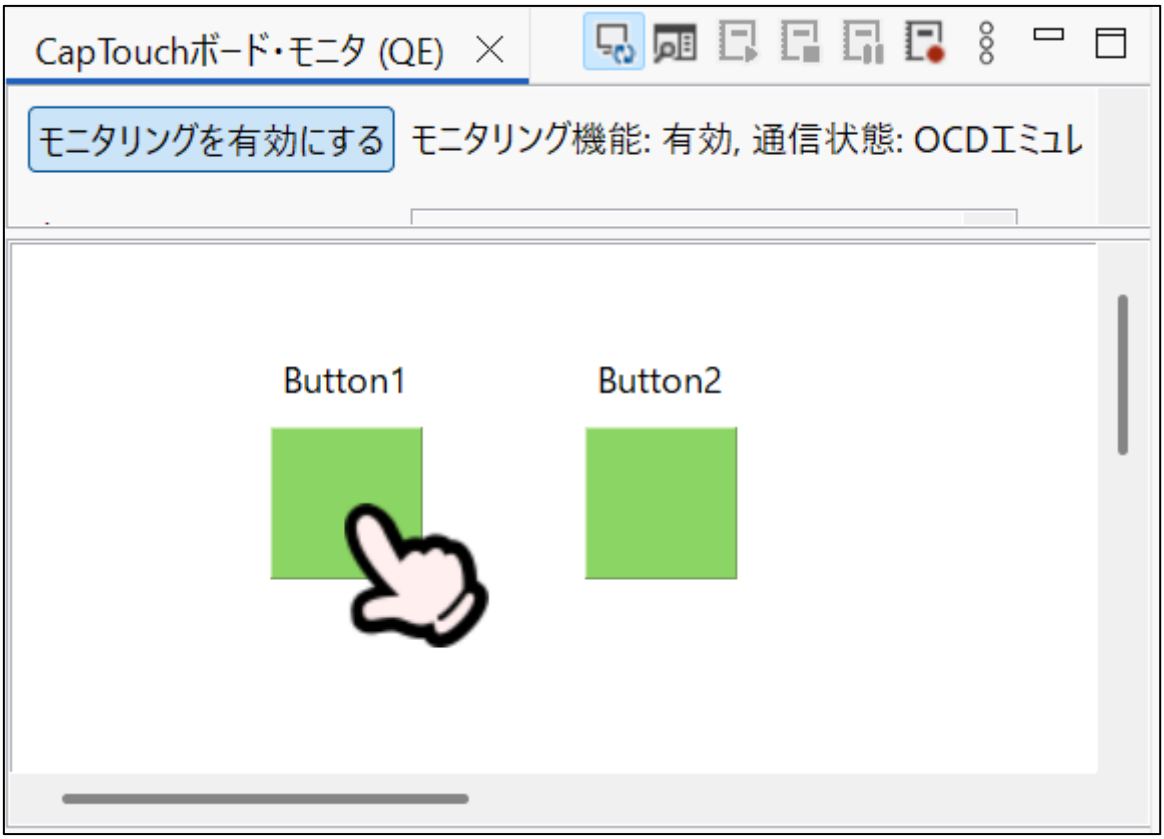


図 4-42. モニタリング Button1 検出

Button2 をタッチ検出したとき、指アイコンが表示されます。

図 4-43. モニタリング Button2 検出

CapTouch ステータス・チャートではタッチ I/F から 1 つ選択したインタフェースを数値としてモニタリングできます。

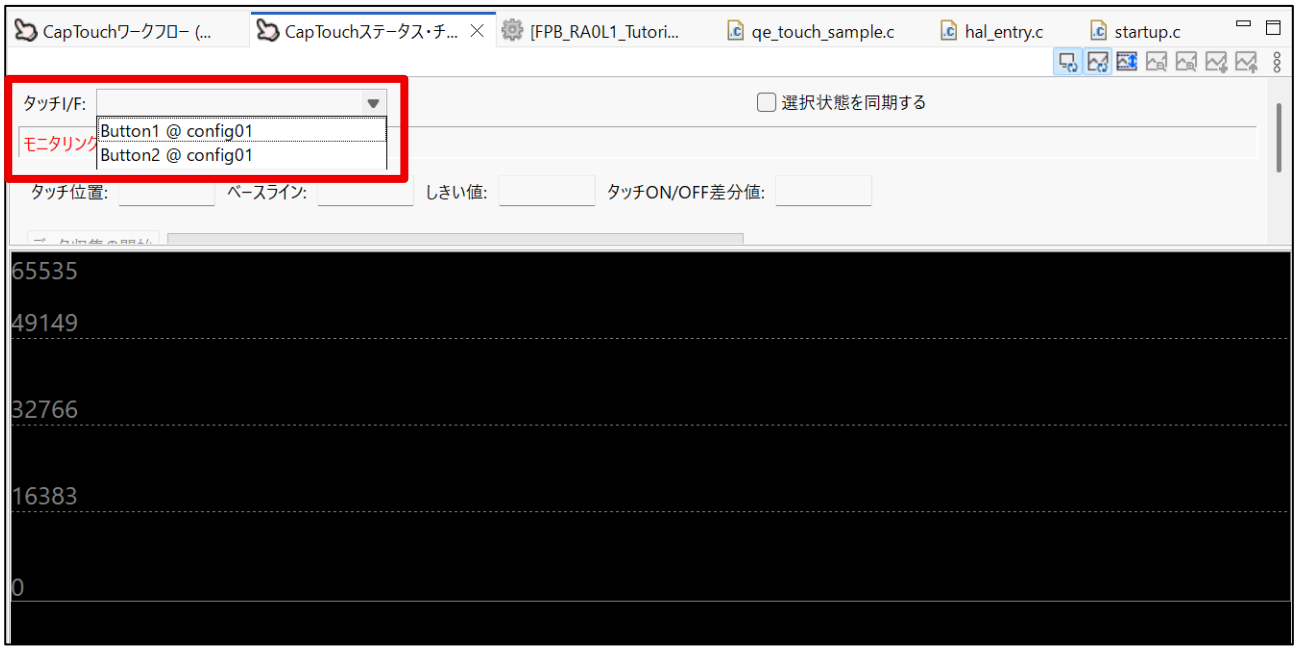


図 4-44. CapTouch ステータス・チャート タッチ I/F 選択

Button1 をタッチ検出したときの表示です。

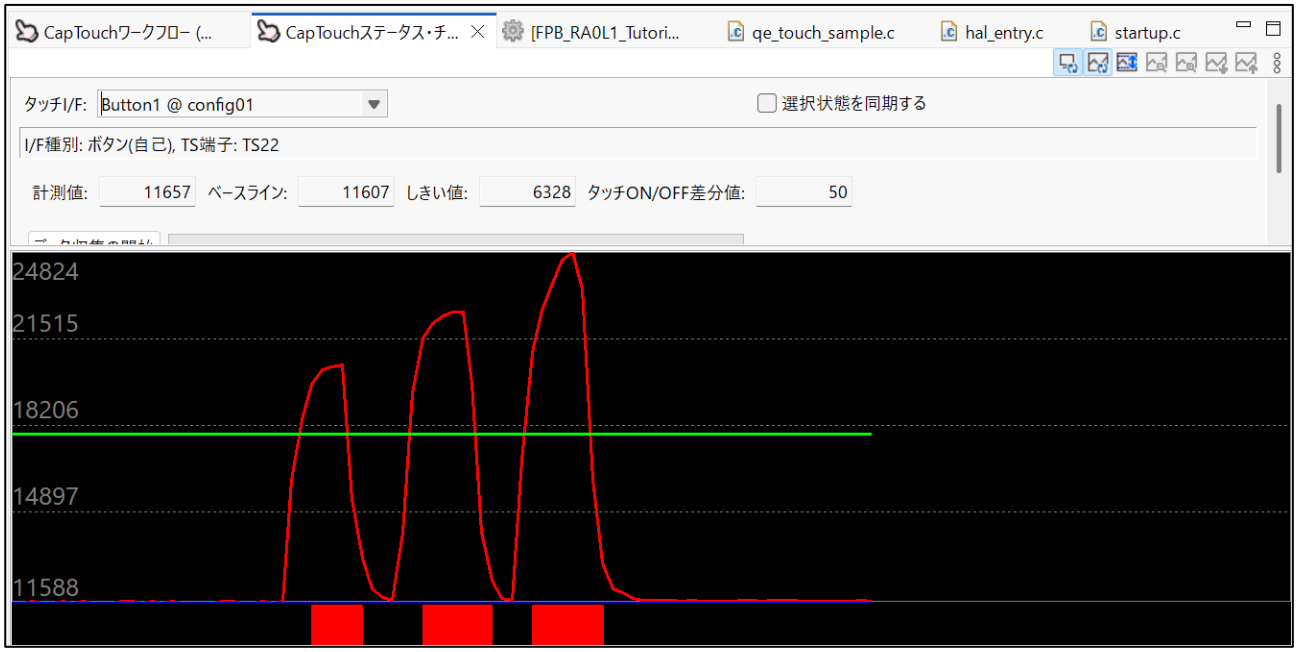


図 4-45. CapTouch ステータス・チャート Button1 検出

Button2 をタッチ検出したときの表示です。

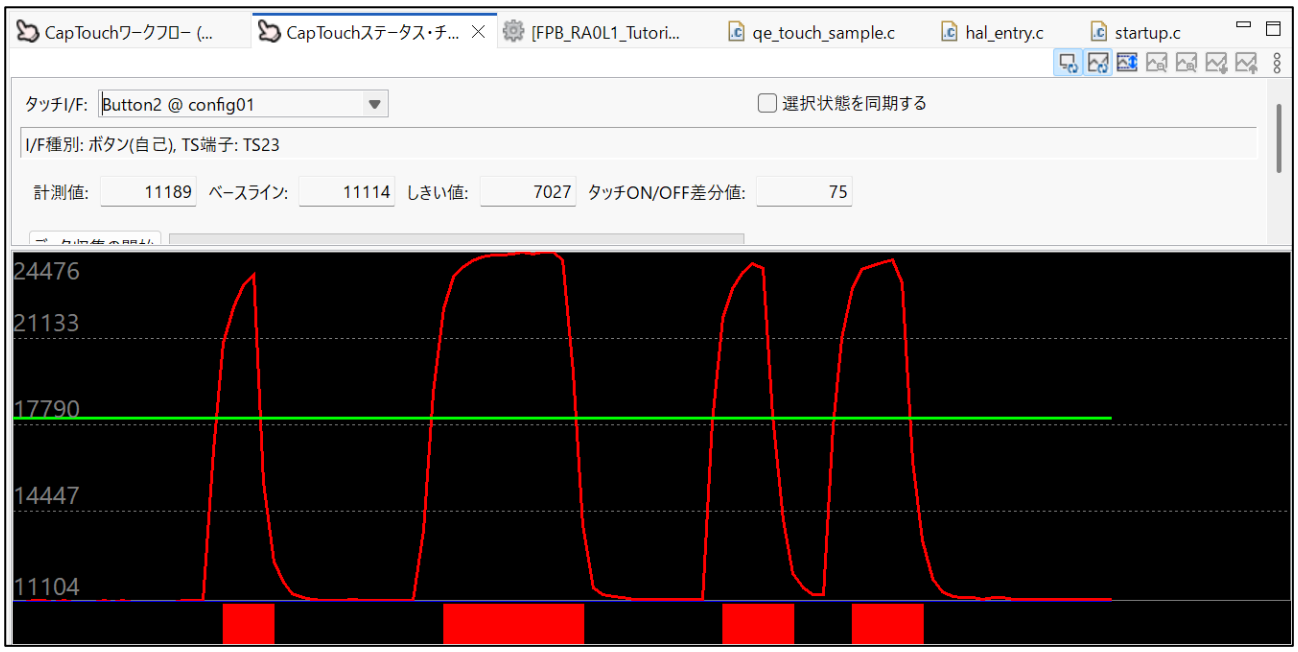


図 4-46. CapTouch ステータス・チャート Button2 検出

CapTouch マルチ・ステータス・チャートでは CapTouch ステータス・チャートと同様にインタフェースを数値としてモニタリング出来ますがこちらは最大 9 つまで同時にモニタリングできます。

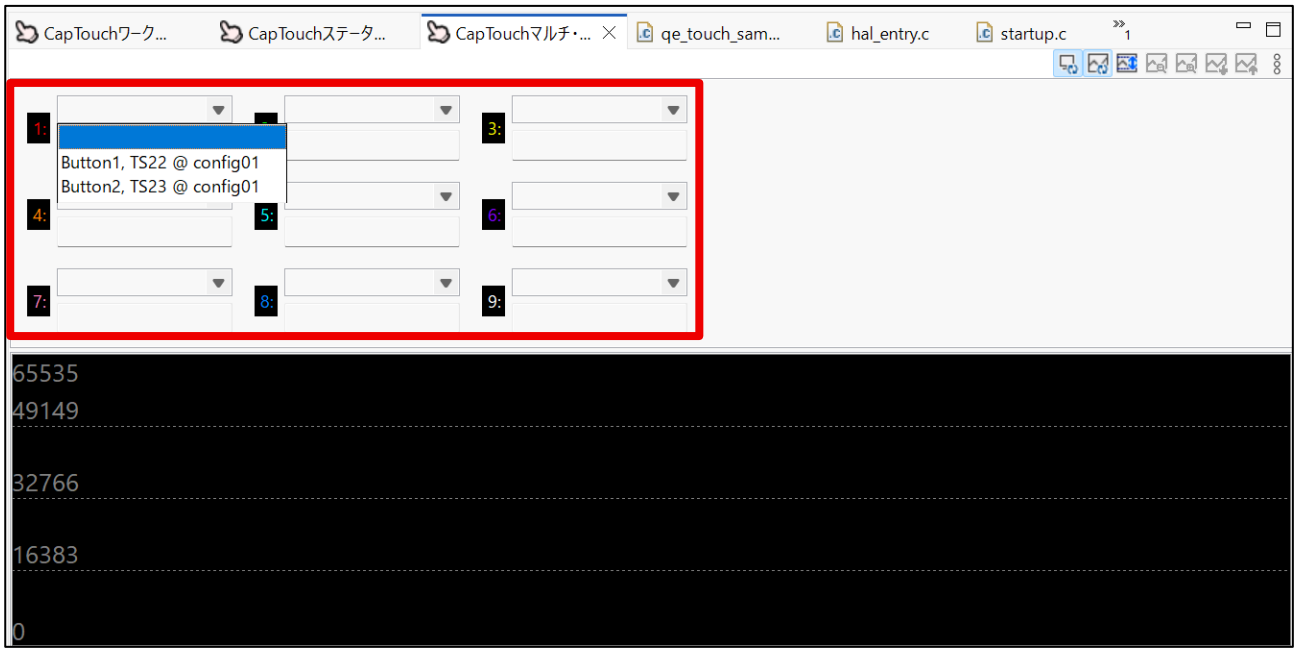


図 4-47. CapTouch マルチ・ステータス・チャート タッチ I/F 選択

Button1 と Button2 をタッチ検出したときの表示です。

Button1：赤線

Button2：緑線

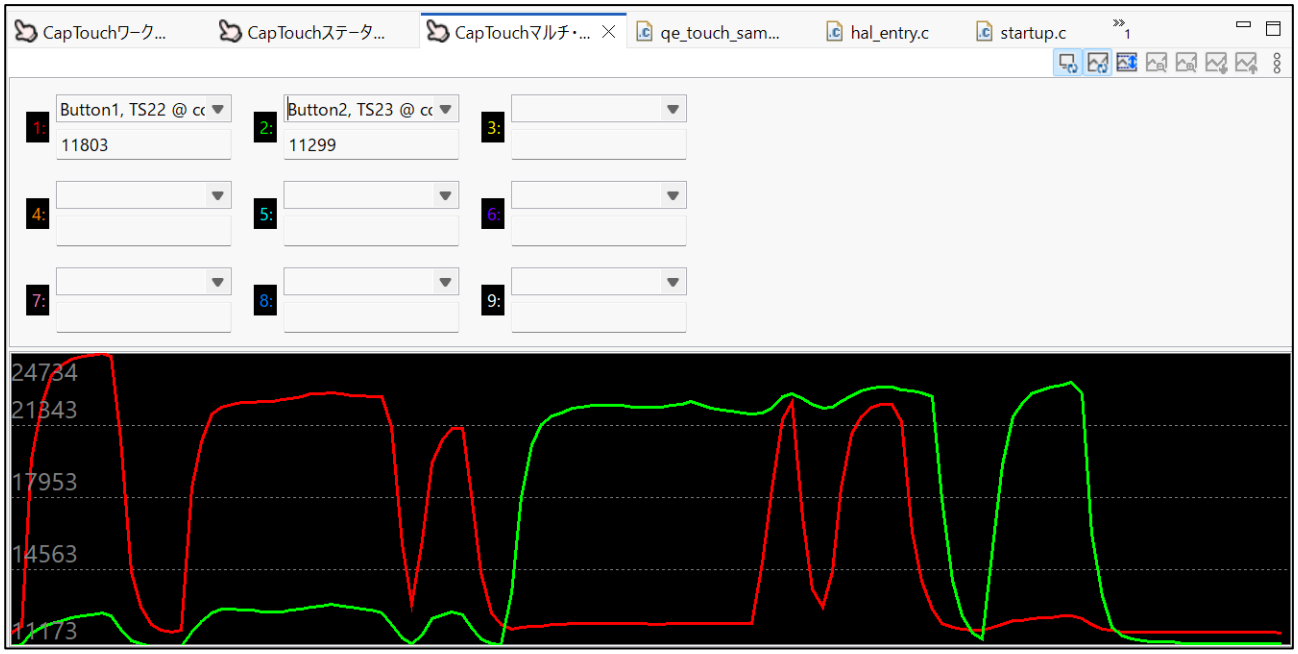


図 4-48. CapTouch マルチ・ステータス・チャート タッチ検出

5. タッチ操作を用いたサンプルプログラムの作成

5.1 サンプルプログラムで使用する機能の FSP Configurator 設定

5.1.1 タイマ機能の設定

FSP Configuration の Stacks タブを開きタイマ機能を追加します。

「New Stack」→「Timer」→「Timer, Independent Channel, 16-bit and 8-bit Timer Operation (r_tau)」をクリックして追加します。

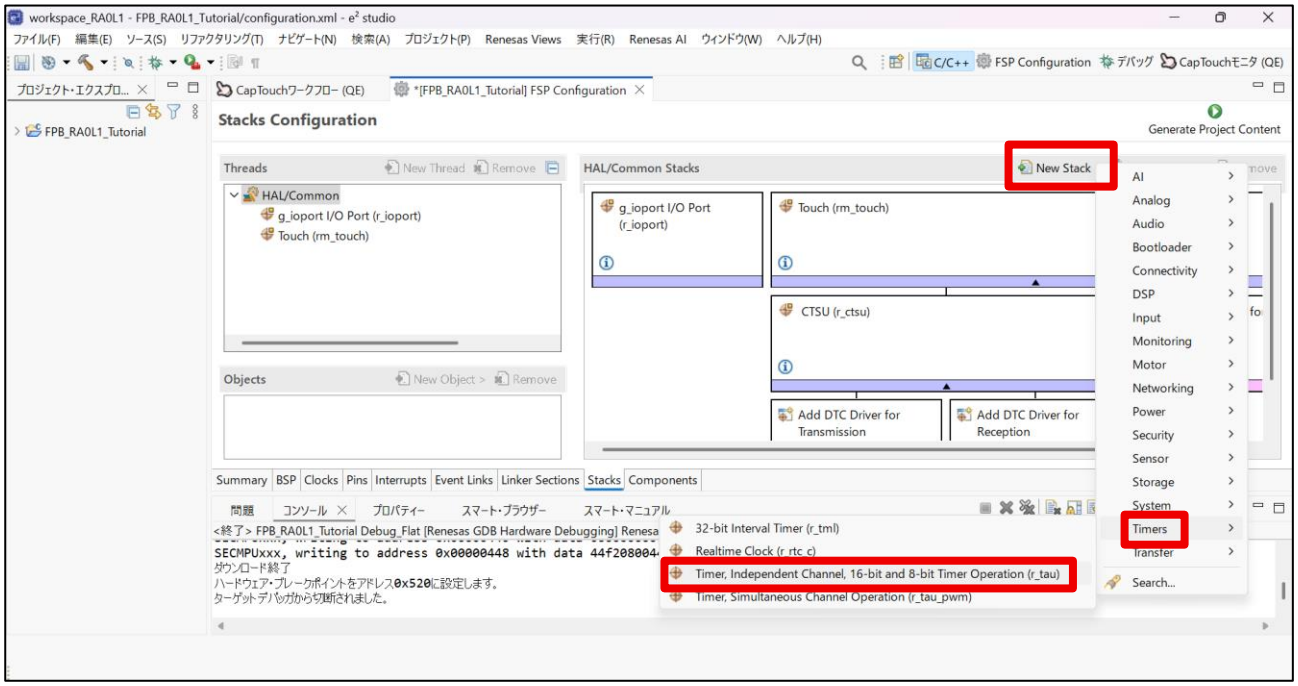


図 5-1. タイマ機能の設定

g_timer0 という名前のスタックが追加されていることを確認します。

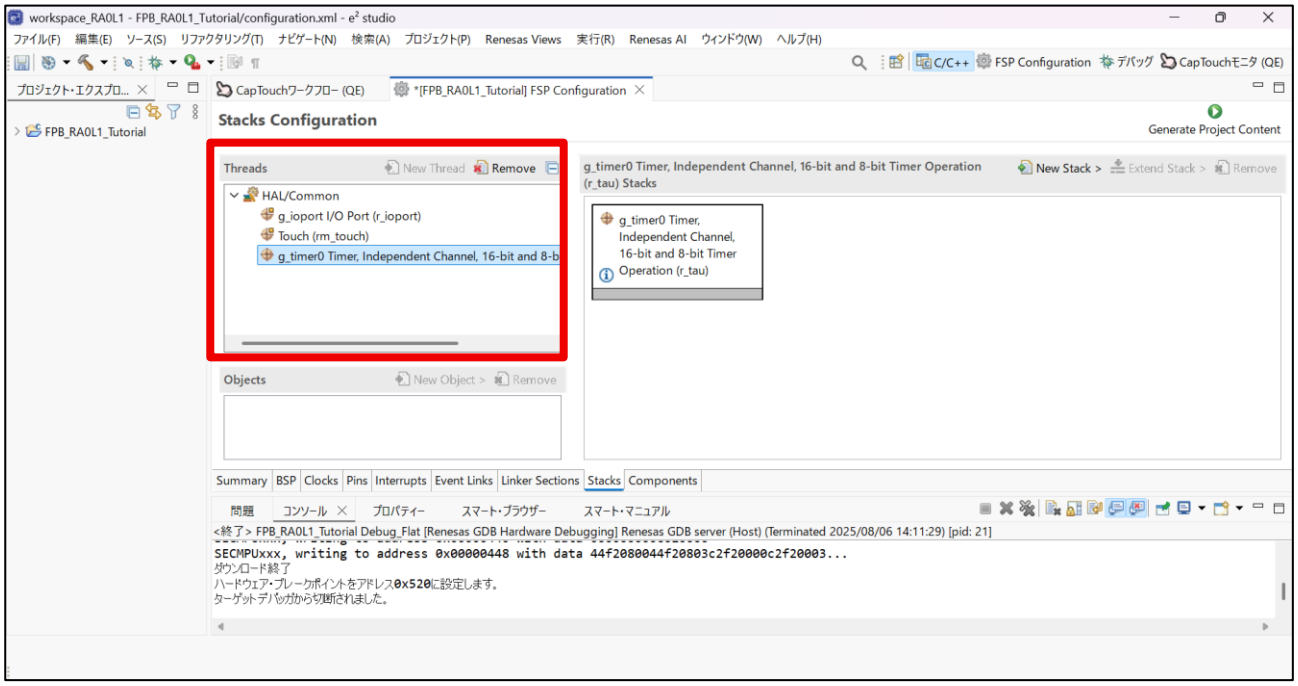


図 5-2. タイマ機能の設定

「プロパティー」ウインドウを表示した状態で g_timer0 のブロックをクリックすると、タイマの詳細な設定項目を表示します。

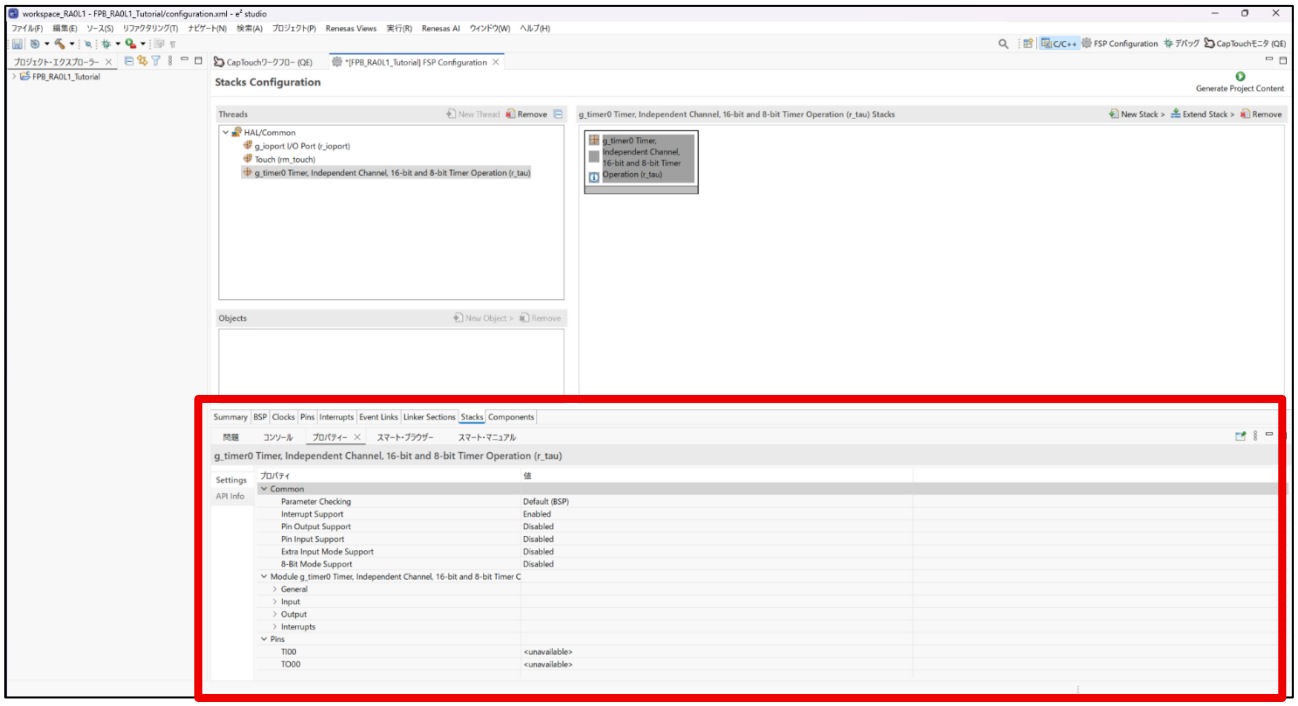


図 5-3. タイマ機能の設定

「プロパティー」ウインドウが見つからない場合は、e² studio のメニューバーから「ウィンドウ」→「ビューの表示」→「プロパティー」を選択すると表示します。

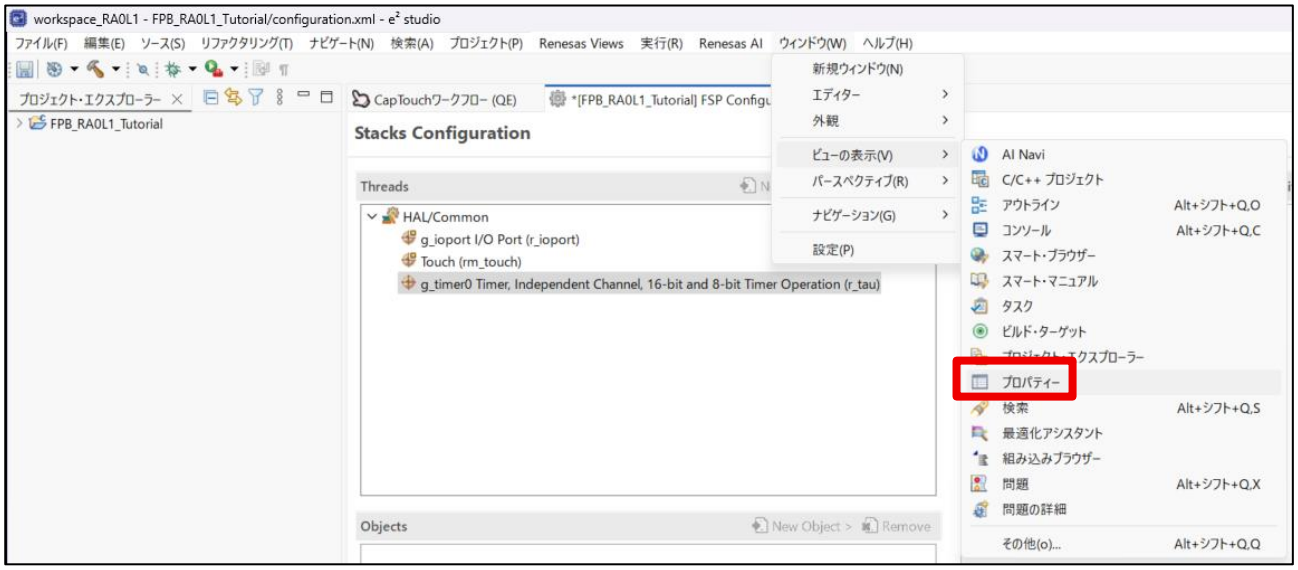


図 5-4. タイマ機能の設定

General のリストを表示します。この画面で基本的なタイマの設定を行います。

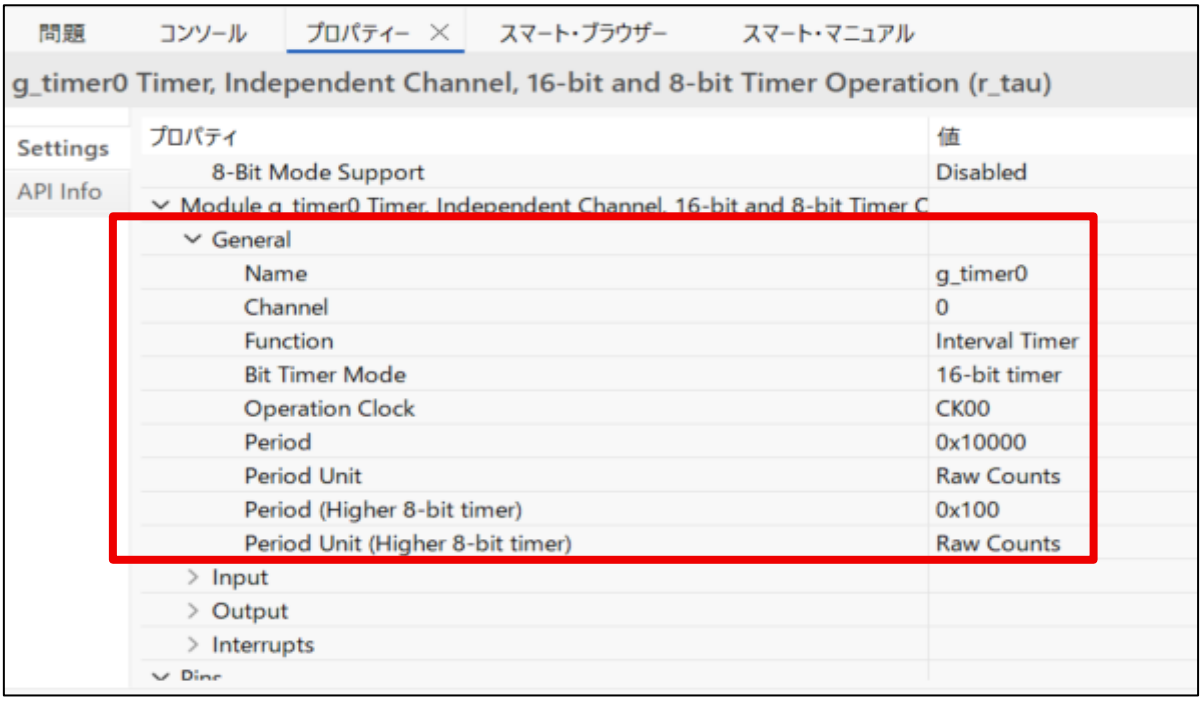


図 5-5. タイマ機能の設定

General の下記の項目を設定します。

- Name : タイマモジュールの名前を設定します。
  今回は「MyTimer」という名前で作成します。
- Channel : 使用する TAU のチャンネル番号。
  今回はチャンネル 0 を使用します。「0」を設定します。
- Function : 使用する TAU チャンネルの機能。
  今回はインターバルタイマを使用します。「Interval Timer」を選択します。
- Bit Timer: 使用する TAU チャンネルのカウンタビット幅。
  今回は 16 ビット幅を使用します。「16-bit timer」を選択します。
- Operation Clock: 使用する TAU チャンネルに供給するクロックを設定します。
  今回は CK00 を使用します。「CK00」を選択します。
- Period / Period Unit : 16bit タイマを使用する際の周期を設定します。
  今回は 1ms 周期を作るため「Period = 1」「Period Unit = Milliseconds」を設定します。

以上で General の設定は終了です。

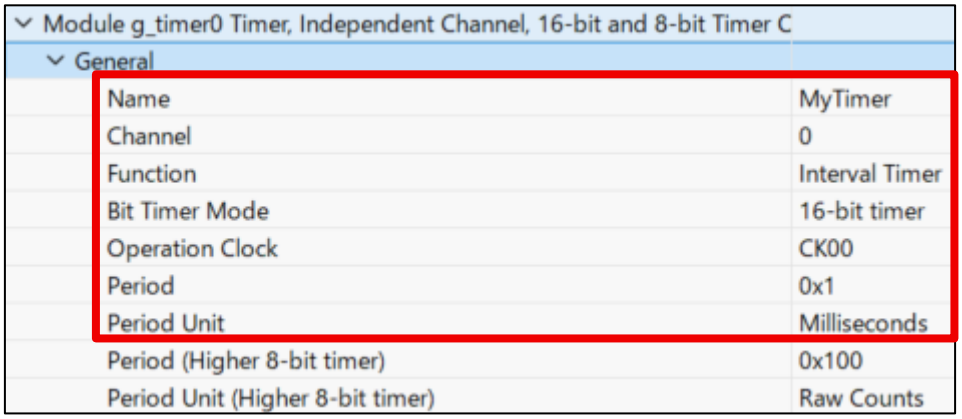


図 5-6. タイマ機能の設定

Interrupts のリストを表示します。この画面では割り込みの設定を行います。

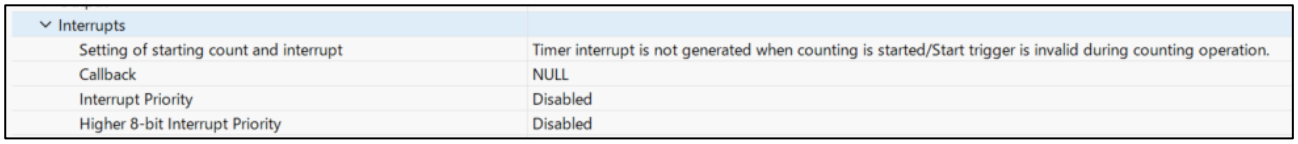


図 5-7. タイマ機能の設定

Callback にユーザが実装可能な割り込み処理関数を設定します。

デフォルト設定の「NULL」は実装する関数がないことを意味します。

今回は「MyCallback」 という名前のコールバック関数を作成します。

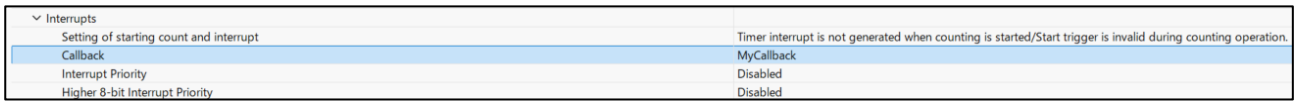


図 5-8. タイマ機能の設定

Interrupt Priority に割り込み優先度を設定します。

デフォルトの「Disabled」は割り込み禁止を意味します。

今回は割り込みを使用しますので、Priority 0 ~ 3 のいずれかに設定します。

本プロジェクトでは「Priority3」で進めます。

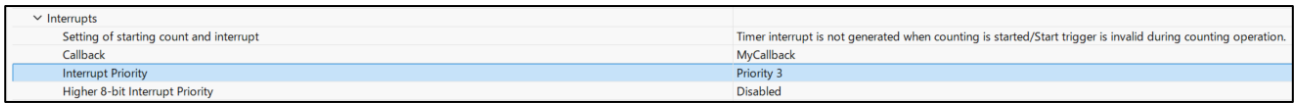


図 5-9. タイマ機能の設定

Timer の設定は以上です。

5.1.2 ピンの設定

LED 制御に使用する端子を以下に示します。

表 5.1 FPB-RA0L1 LED の制御ポート

部品番号	カラー	RA MCU 制御ポート
LED1	緑	P002(High 点灯)
LED2	緑	P104(High 点灯)
LED5	緑	P401(Low 点灯)
LED6	緑	P400(Low 点灯)

FSP Configuration の Pins タブを開き端子設定を確認します。

LED の対応する端子の Mode が Output mode であることを確認してピンの設定は完了です。

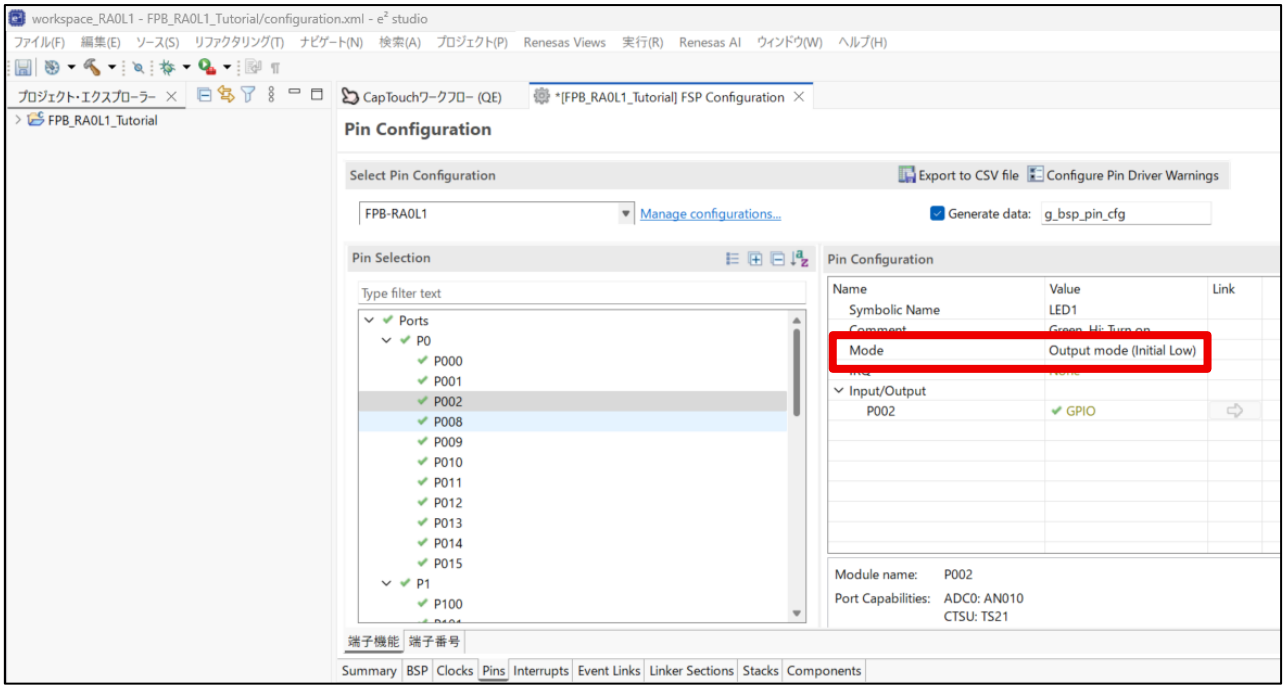


図 5-10. LED1 のピン設定

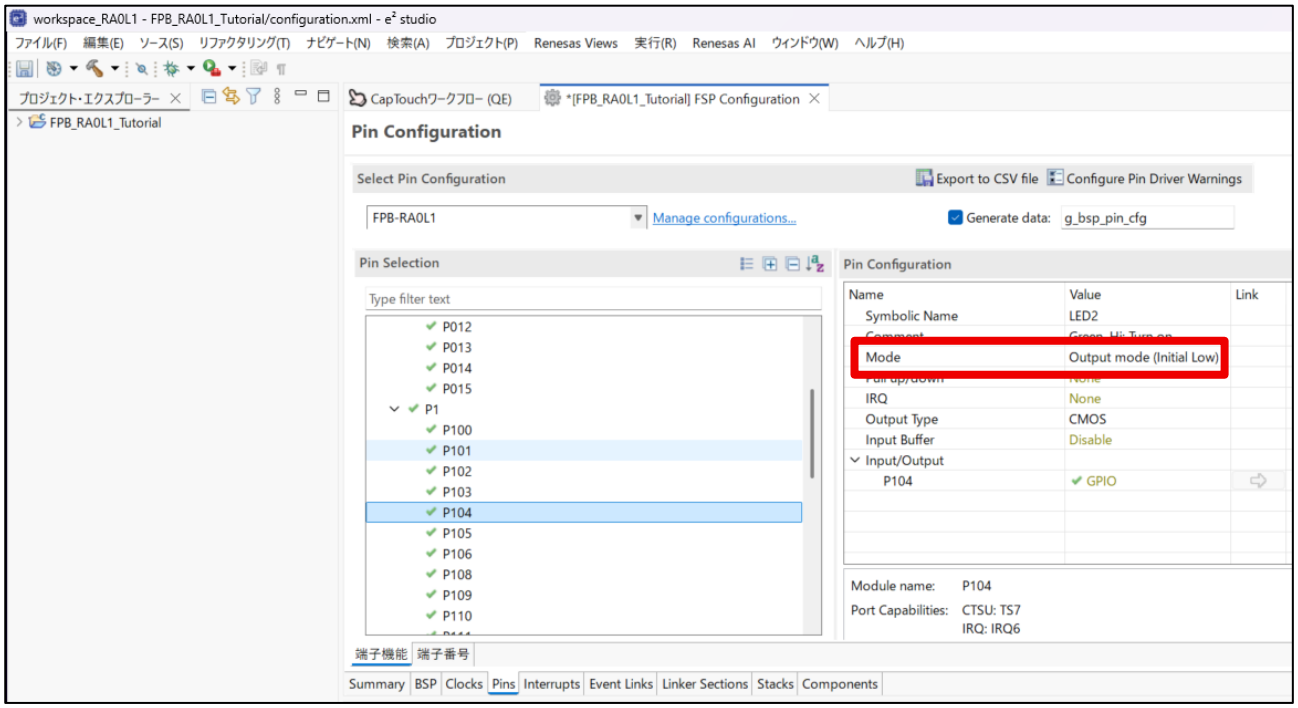


図 5-11. LED2 のピン設定

今回のプログラム作成で必要な設定は完了しましたので、「Generate Project Content」ボタンを押してコード生成します。

図 5-12. ピンの設定

Configuration.xml ファイルに保存するかどうかの確認画面がでますので「続行」ボタンを押して保存します。

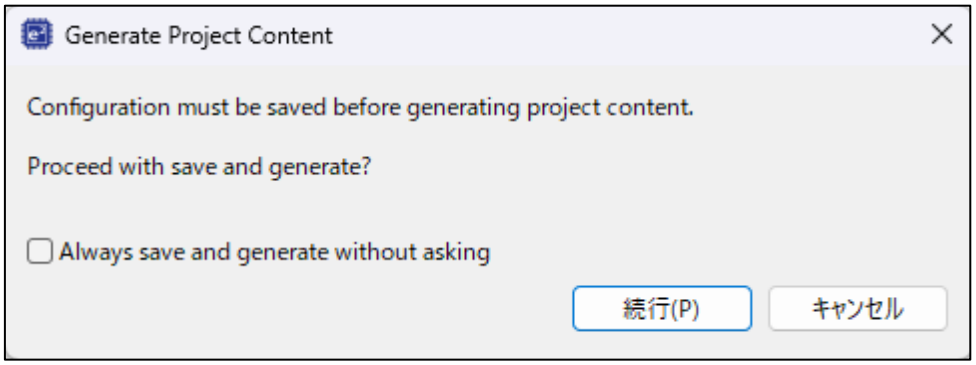

Generate Project Content
Configuration must be saved before generating project content.
Proceed with save and generate?
Always save and generate without asking
続行(P)
キャンセル

図 5-13. ピンの設定

5.2 サンプルプログラムのコーディング

main プログラム、割り込み関数にコードを実装します。

実装する内容は以下の通りです。

- main プログラム
  - タイマの起動と開始
  - Touch Button タッチ検出時の LED 点滅開始と停止、点滅速度の変化
- 割り込みプログラム
  - 時間計測用変数のカウント

ユーザが記述するファイルは プロジェクトフォルダ¥qe_gen¥qe_touch_sample.c です。

qe_touch_sample.c をダブルクリックして、ファイルを開きます。

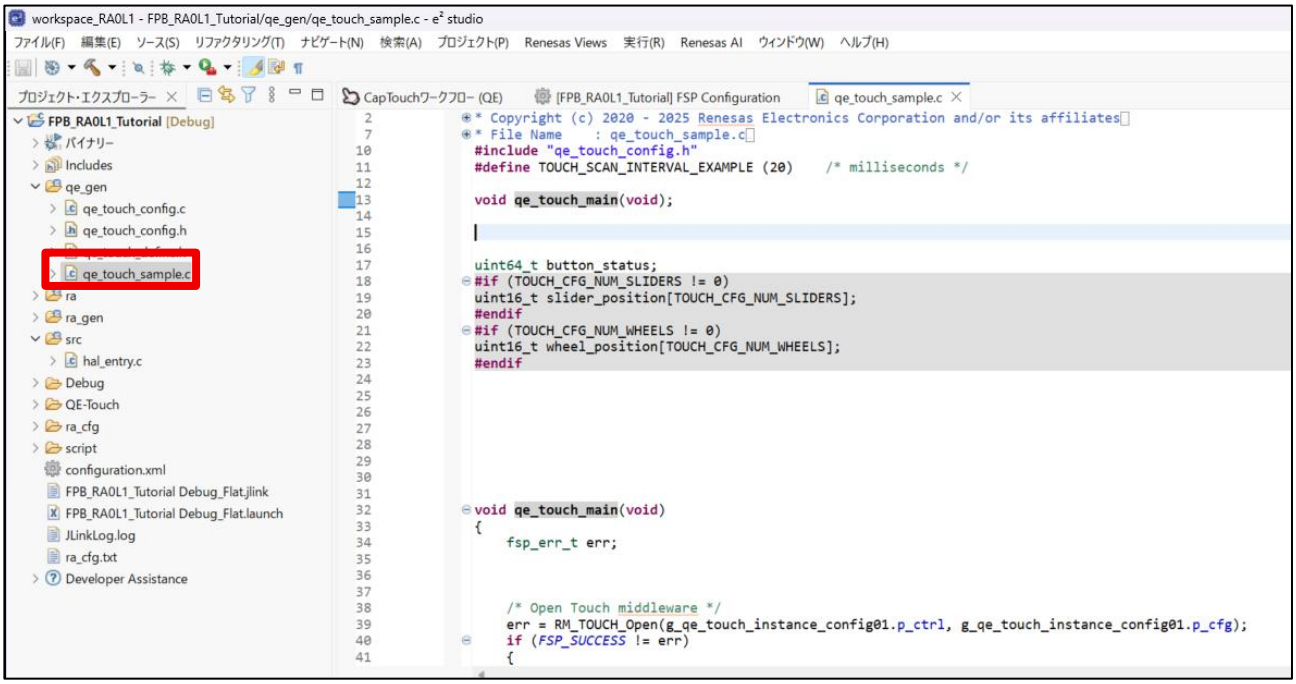


図 5-14. main プログラムの実装

qe_touch_sample.c の void qe_touch_main(void)の中にソースコードを記述していきます。

I/O ポートの初期設定は qe_touch_main 関数が呼び出される前に実行されますので、記述不要です。

プログラム作成は下記の段階を踏んで説明します。

(1) タイマ機能を用いて 20ms ごとにタッチスキャンを行い、タッチ情報を取得する
(2) Touch Button のタッチ検出状態に応じて LED5 と LED6 の点灯状態を制御する
(3) Touch Button1 のタッチ検出により LED1 と LED2 の点滅動作の開始/停止を制御する
(4) Touch Button2 のタッチ検出により点滅周期を変更する

5.2.1 タイマ機能を用いて 20ms ごとにタッチスキャンを行い、タッチ情報を取得する

(1) タイマのオープン関数の実装

「プロジェクト・エクスプローラー」ウインドウにある「Developer Assistance」のリストを開くと、「5.1.1 タイマ機能の設定」で設定したタイマモジュール MyTimer が表示されます。

さらにリストを開くと、関数一覧が表示されます。

タイマをオープンする関数は R_TAU_Open()です。マウスでこの関数をソースファイルにドラッグアンドドロップします。この関数は main loop を行う前に呼び出します。

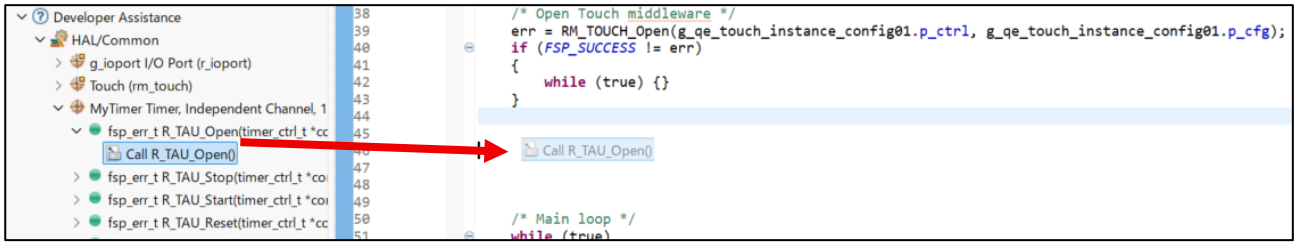


図 5-15. main プログラムの実装

オープン関数が追加されました。

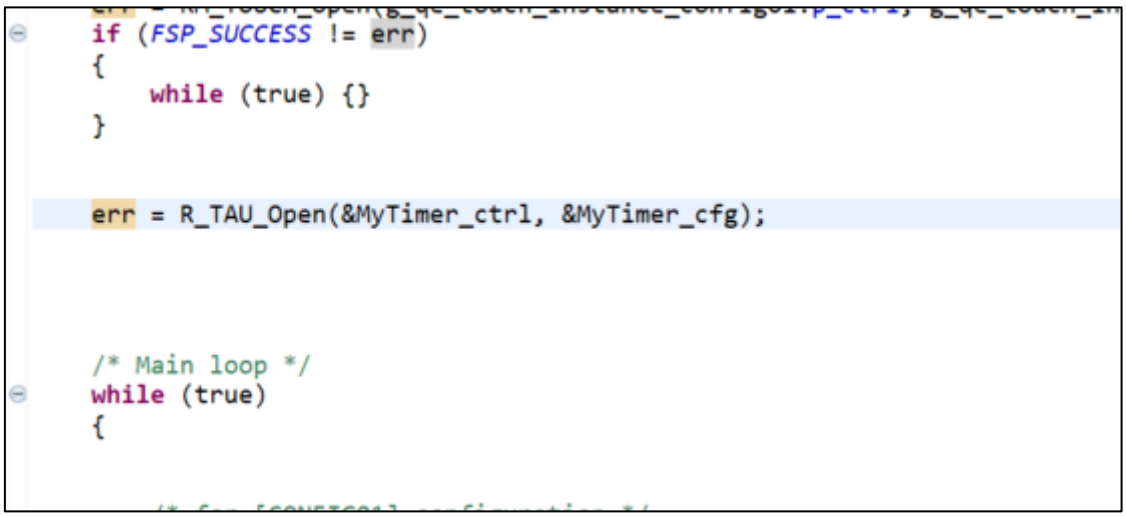


図 5-16. main プログラムの実装

(2) タイマのスタート関数の実装

スタート関数は R_TAU_Start()です。先程と同様に、マウスでソースファイル内の R_TAU_Open()関数呼び出しの下にドラッグアンドドロップします。

図 5-17. main プログラムの実装

スタート関数が追加されました。

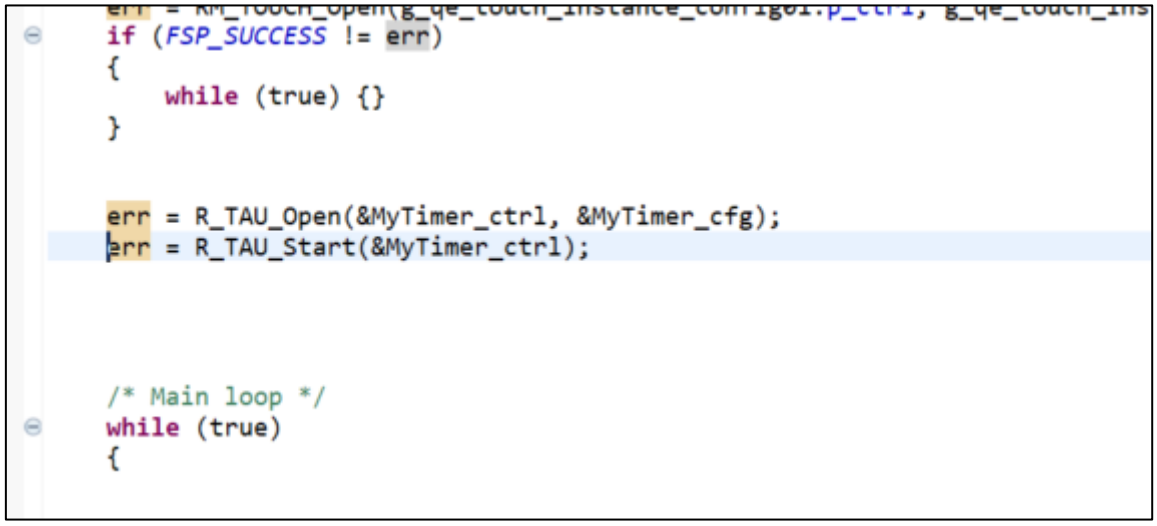


図 5-18. main プログラムの実装

(3) 戻り値の確認

オープン関数、スタート関数が異常終了を返した場合を検出するために、関数呼び出し後に while(err);を実装しておきます。異常終了した場合はこの文で無限ループします。

関数のソースコードは以下のようになります。

CapTouchワークフロー (QE) [FPB_RA0L1_Tutorial] FSP Configuration qe_touch_sample.c

```
void qe_touch_main(void)
{
    fsp_err_t err;



    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }


    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);




    /* Main loop */
    while (true)
    {


        /* for [CONFIG01] configuration */
        err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
        if (FSP_SUCCESS != err)
        {
            while (true) {}
        }
        while (0 == g_qe_touch_flag) {}
        g_qe_touch_flag = 0;

        err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
        if (FSP_SUCCESS == err)
        {
            /* TODO: Add your own code here. */
        }









        /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
        R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS);
    }
}
```

図 5-19. main プログラムの実装

(4) タイマ割り込みのコールバック関数の実装

タイマ割り込みから呼ばれるコールバック関数を記述します。記述するファイルは引き続き ge_touch_sample.c とします。

Developer Assistance のリストからタイマモジュールの「Callback function definition」をソースファイルの最終行にドラッグアンドドロップします。

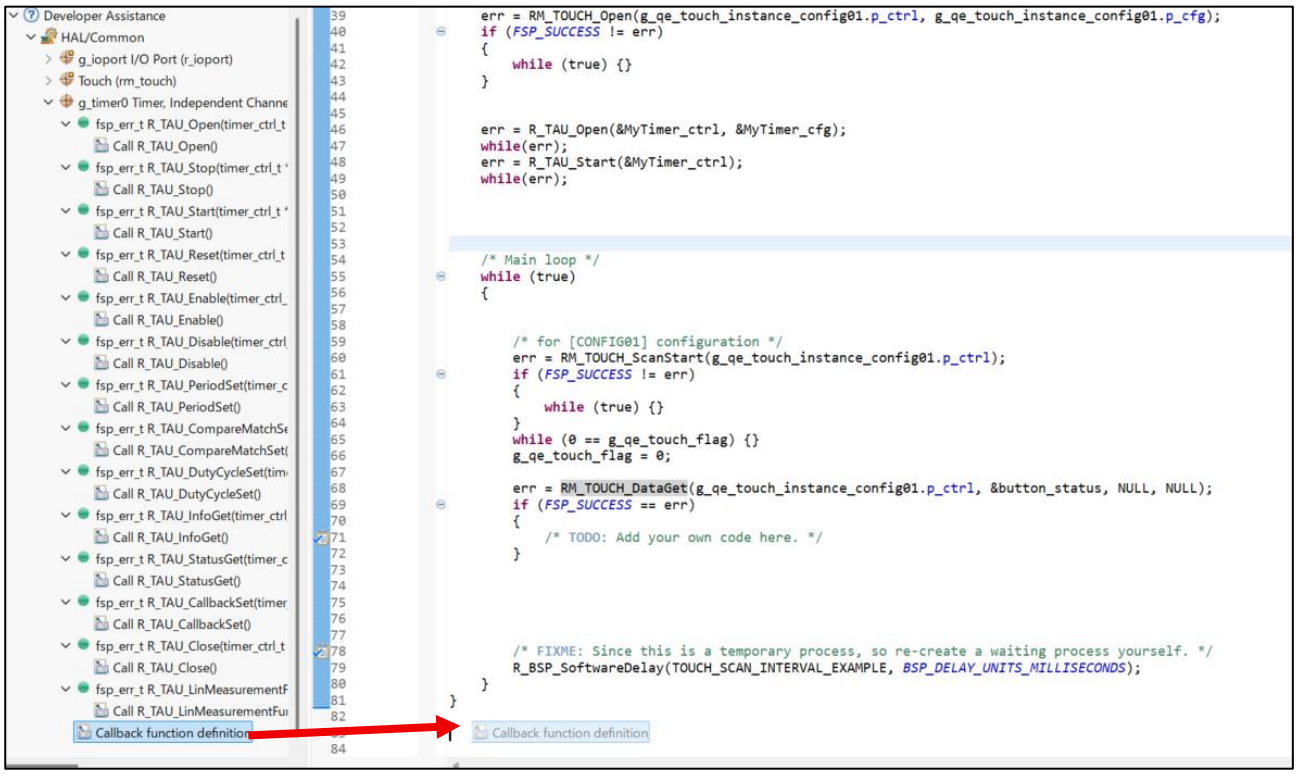


図 5-20. main プログラムの実装

コールバック関数が追加されました。

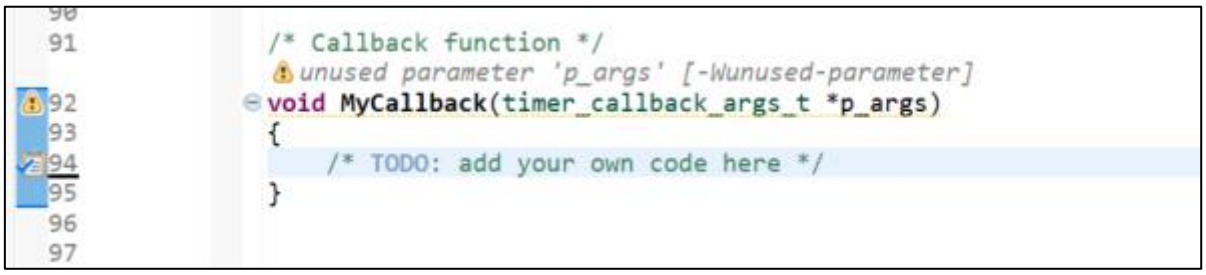


図 5-21. main プログラムの実装

インクリメント用の変数を宣言します。

関数外に volatile uint16_t g_touch_info_get_interval;を宣言します。

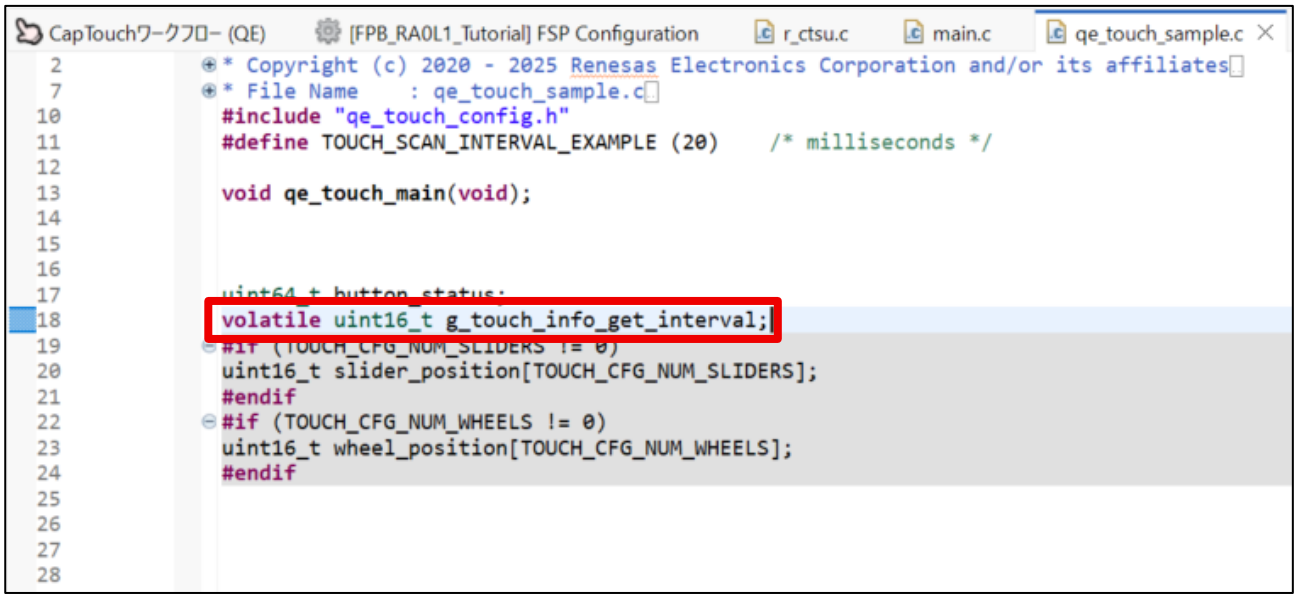


図 5-22. main プログラムの実装

先ほど作成したコールバック関数内で g_touch_info_get_interval 変数をインクリメントするようにコーディングします。これで 1ms 毎にこの変数の値がインクリメントされます。

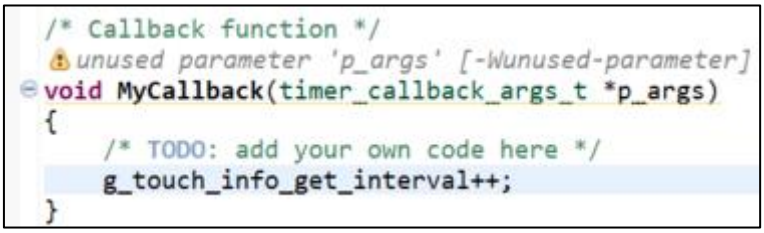


図 5-23. main プログラムの実装

(5) タッチ関数の実行タイミングの調整

(d)で準備した変数を用いて、20ms 毎にタッチスキャンとタッチ情報リードを行うよう条件文を追加します。

タッチスキャンを行う関数は RM_TOUCH_ScanStart 関数、

タッチ情報リードを行う関数は RM_TOUCH_DataGet 関数です。

上記 2 関数が呼び出される前に条件文を追加します。

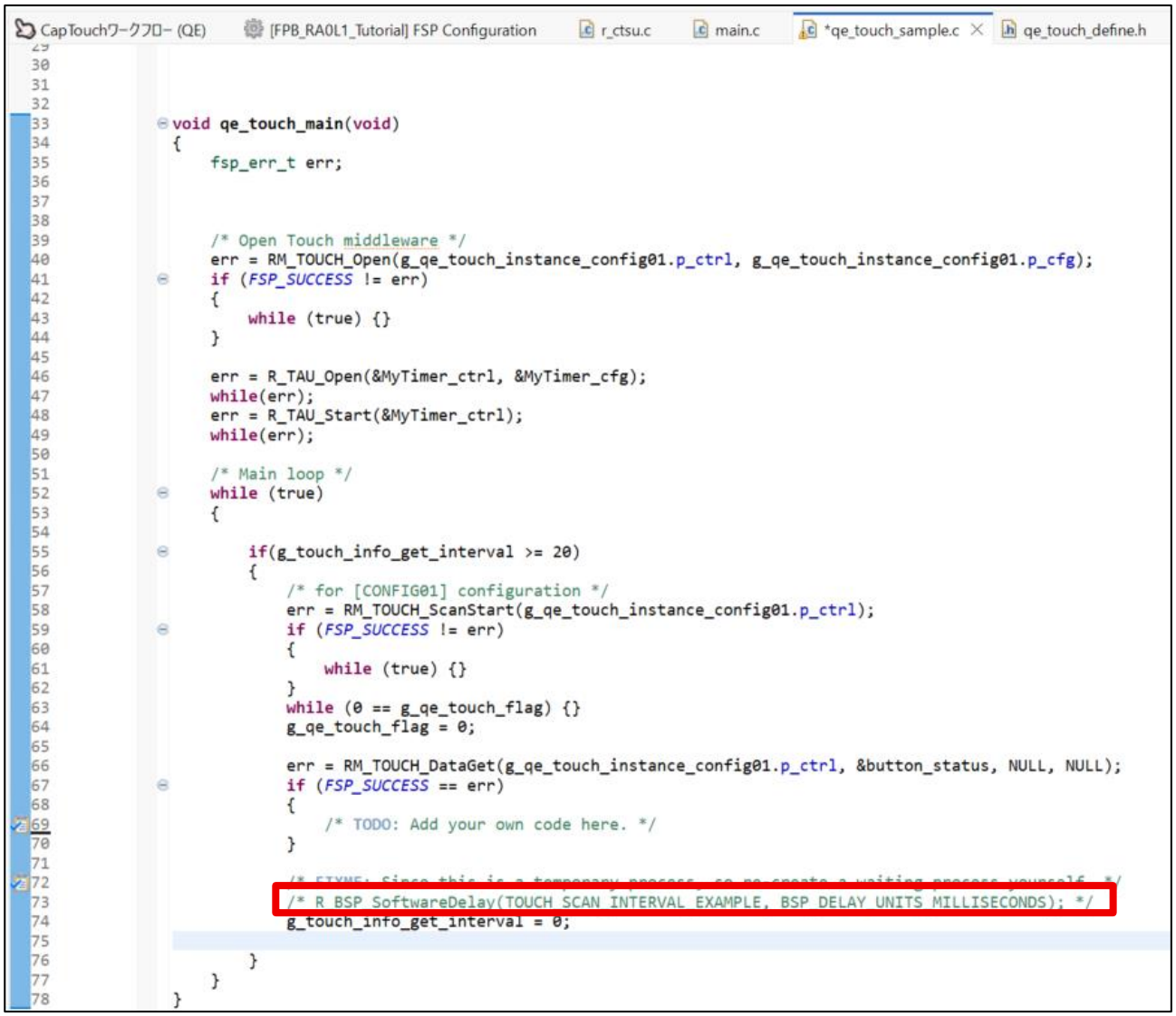
CapTouchワークフロー (QE) [FPB_RA0L1_Tutorial] FSP Configuration r_ctsu.c main.c *qe_touch_sample.c qe_touch_define.h

```
void qe_touch_main(void)
{
    fsp_err_t err;


    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS);
            g_touch_info_get_interval = 0;

        }
    }
}
```

図 5-24. main プログラムの実装

(6) ソフトウェアディレイ関数のコメントアウト

R_BSP_SoftwareDelay()関数は(e)の実装により不要となったためコメントアウトします。




```
void qe_touch_main(void)
{
    fsp_err_t err;


    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
            }

            /* FIXME: Since this is a temporary process, so no create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}
```

図 5-25. main プログラムの実装

以上で 20ms ごとにタッチスキャンとタッチ情報リードを行いタッチ情報を取得するプログラムは完成です。

ここまでのソースコードをテキストで掲載します。qe_touch_sample.c を生成した時点から追記した箇所を青字で示しています。

```
void qe_touch_main(void)
{
    fsp_err_t err;



    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
 g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
 &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
            }

            /* FIXME: Since this is a temporary process,
 so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
 BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
}
```

5.2.2 Touch Button のタッチ検出状態に応じて LED5 と LED6 の点灯状態を制御する

タッチ情報リードを行う RM_TOUCH_DataGet()関数を実行すると、第 2 引数に指定した button_status 変数に TouchButton1 と 2 の情報が格納されます。

例えば TouchButton1 のみの情報を参照したい場合、プロジェクトフォルダ¥qe_gen¥qe_touch_define.h に記述されている下記のマクロ定義(赤枠内)を用いて論理積を行う事で参照できます。

赤枠で示したマクロ定義名の一部である"BUTTON1"、および"BUTTON2"は「4.2 QE Touch を使ったタッチセンサのチューニング」で設定した名前に紐づいています。

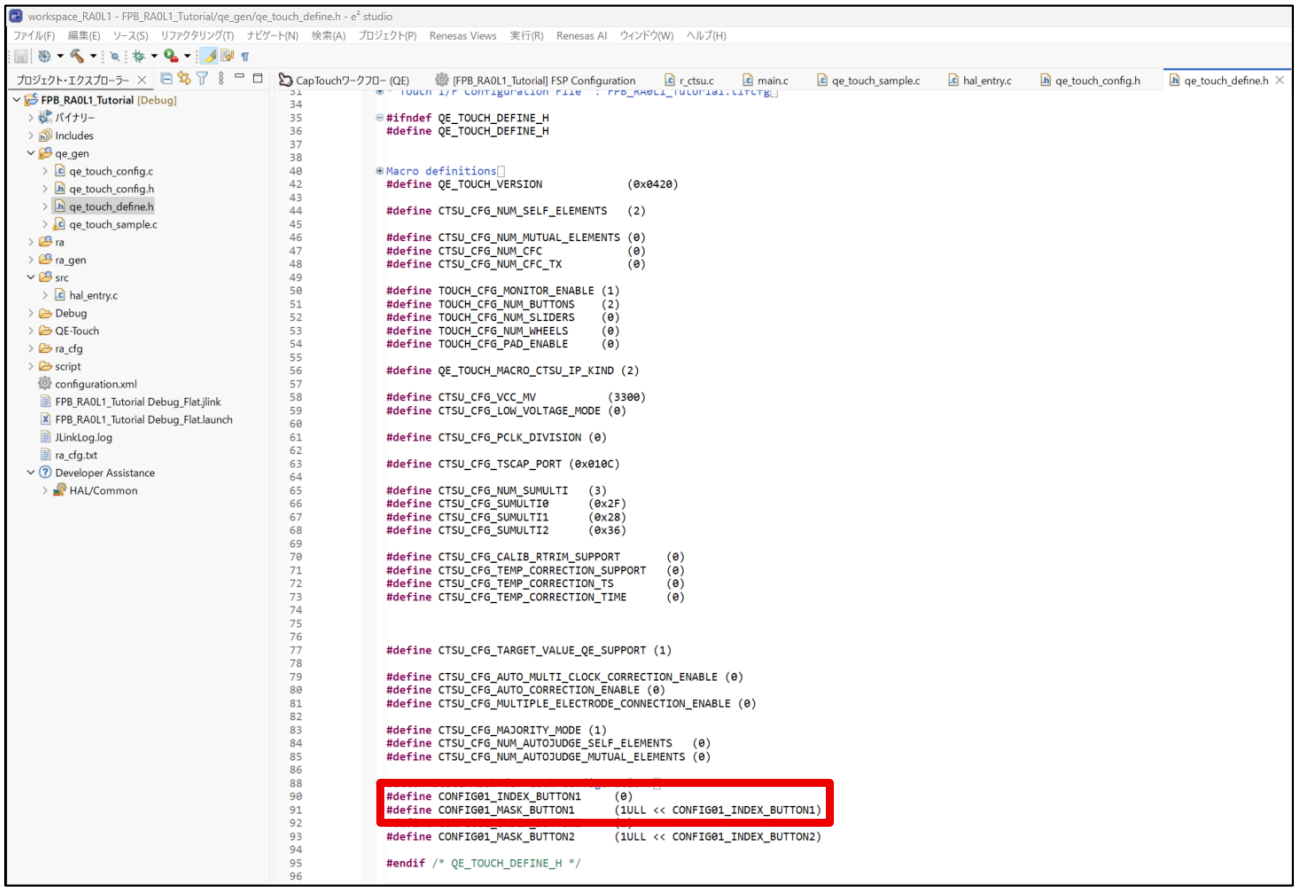


図 5-26. main プログラムの実装

(1) Touch Button 1 をタッチ検出する間は LED5 が点灯するプログラム

下記のように条件文を追加します。

```
69                  err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
70                  if (FSP_SUCCESS == err)
71                  {
72                      /* TODO: Add your own code here. */
73                      if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
74                      {
75
76                      }
77                      else
78                      {
79                          |
80                      }
81                  }
```

図 5-27. main プログラムの実装

条件文の中で LED5 の点灯/消灯を設定するプログラムをコーディングします。

Developer Assistance のリストから R_IOPORT_PinWrite()関数をソースファイルの条件内にドラッグアンドドロップします。

図 5-28. main プログラムの実装

R_IOPORT_PinWrite()関数が追加されました。

```
69                  err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
70                  if (FSP_SUCCESS == err)
71                  {
72                      /* TODO: Add your own code here. */
73                      if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
74                      {
75                          err = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
76                      }
77                      else
78                      {
79                          err = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
80                      }
81                  }
```

図 5-29. main プログラムの実装

R_IOPORT_PinWrite 関数の引数を変更し LED の ON/OFF を設定します。

LED5 を設定したい場合、マクロ定義はプロジェクト・エクスプローラーから ra_cfg¥fsp_cfg¥bsp_pin_cfg.h の下記赤枠にて記述されています。

このマクロ定義を R_IOPORT_PinWrite()関数の引数に使用して LED5 の ON/OFF を設定できます。

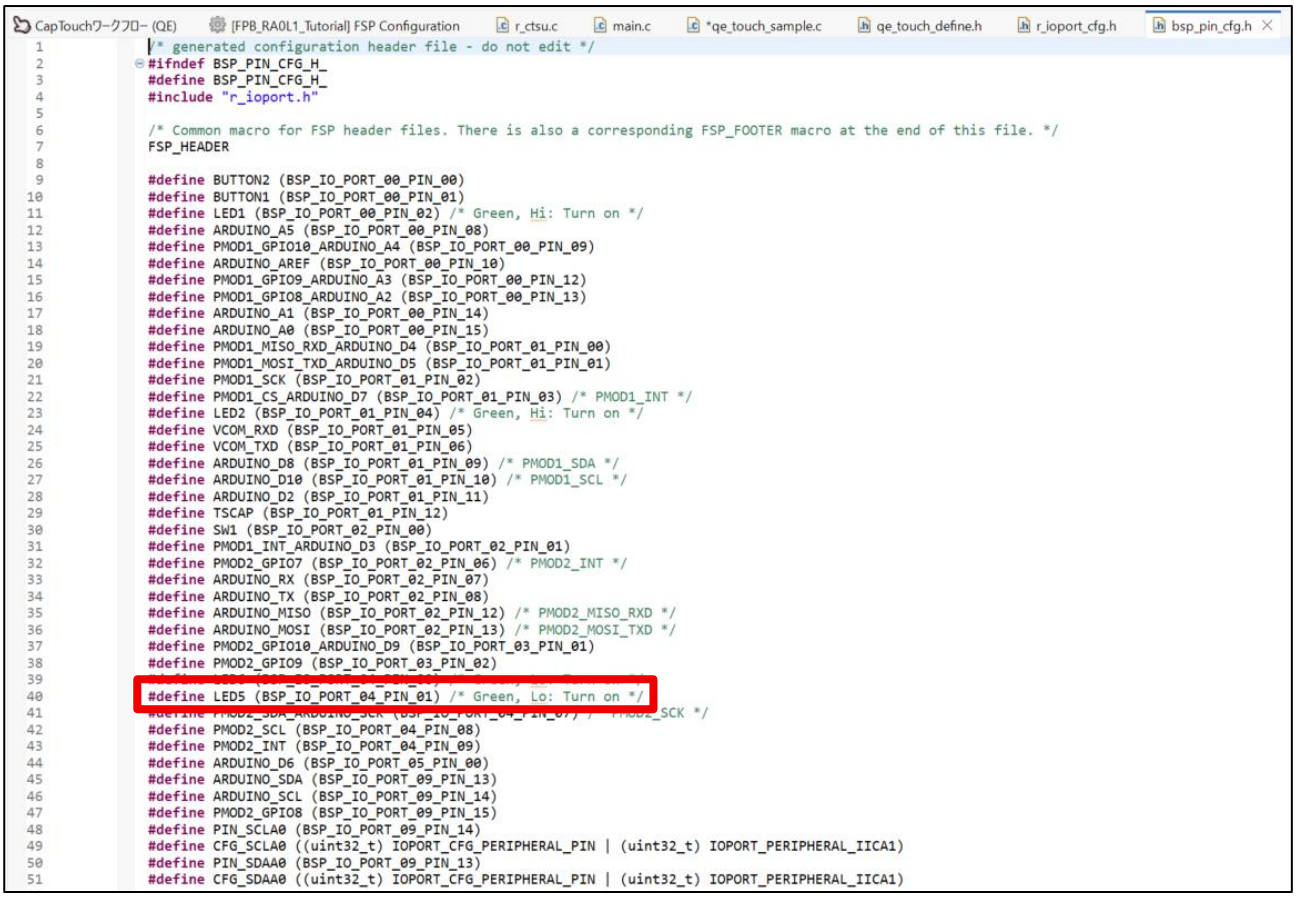


図 5-30. main プログラムの実装

R_IOPORT_PinWrite()関数の第 2 引数には設定するピン番号 = "LED5"、 第 3 引数には出力内容を記入します。

Touch Button 1 をタッチ検出する間は LED5 が点灯するように設定したいので、下記のようにソースコードを編集します。

CapTouchワークフロー (QE) [FPB_RA0L1_Tutorial] FSP Configuration r_ctsu.c main.c *qe_touch_sample.c qe_touch_define.h

```
void qe_touch_main(void)
{
    fsp_err_t err;



    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }


    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);



    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
                }
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}
```

図 5-31. main プログラムの実装

(2) Touch Button 2 をタッチ検出する間は LED6 が点灯するプログラム

前述した手順で LED6 の実装を行います。

下記のように条件文を追加します。

```
67          if (FSP_SUCCESS == err)
68          {
69              /* TODO: Add your own code here. */
70              if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
71              {
72                  err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
73              }
74              else
75              {
76                  err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
77              }
78
79              if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
80              {
81
82              }
83              else
84              {
85                  |
86              }
87          }
```

図 5-32. main プログラムの実装

Developer Assistance のリストから R_IOPORT_PinWrite 関数をソースファイルの条件内にドラッグアンドドロップします。

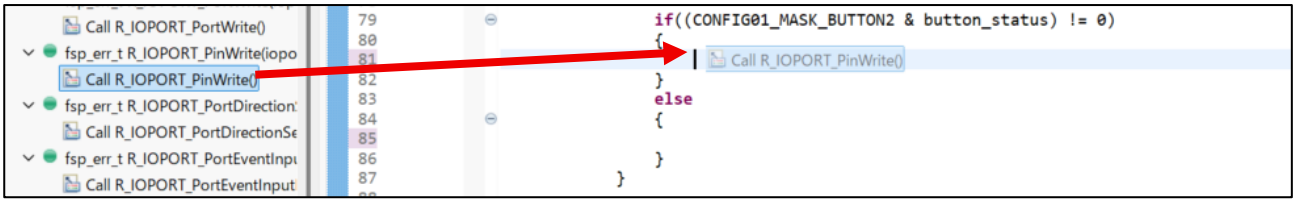


図 5-33. main プログラムの実装

R_IOPORT_PinWrite()関数が追加されました。

```
78
79              if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
80              {
81                  err = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
82              }
83              else
84              {
85                  err = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
86              }
87          }
```

図 5-34. main プログラムの実装

LED5 と同様に bsp_pin_cfg.h に記述されている下記のマクロ定義を R_IOPORT_PinWrite()関数の引数に使用して LED6 の ON/OFF を設定できます。

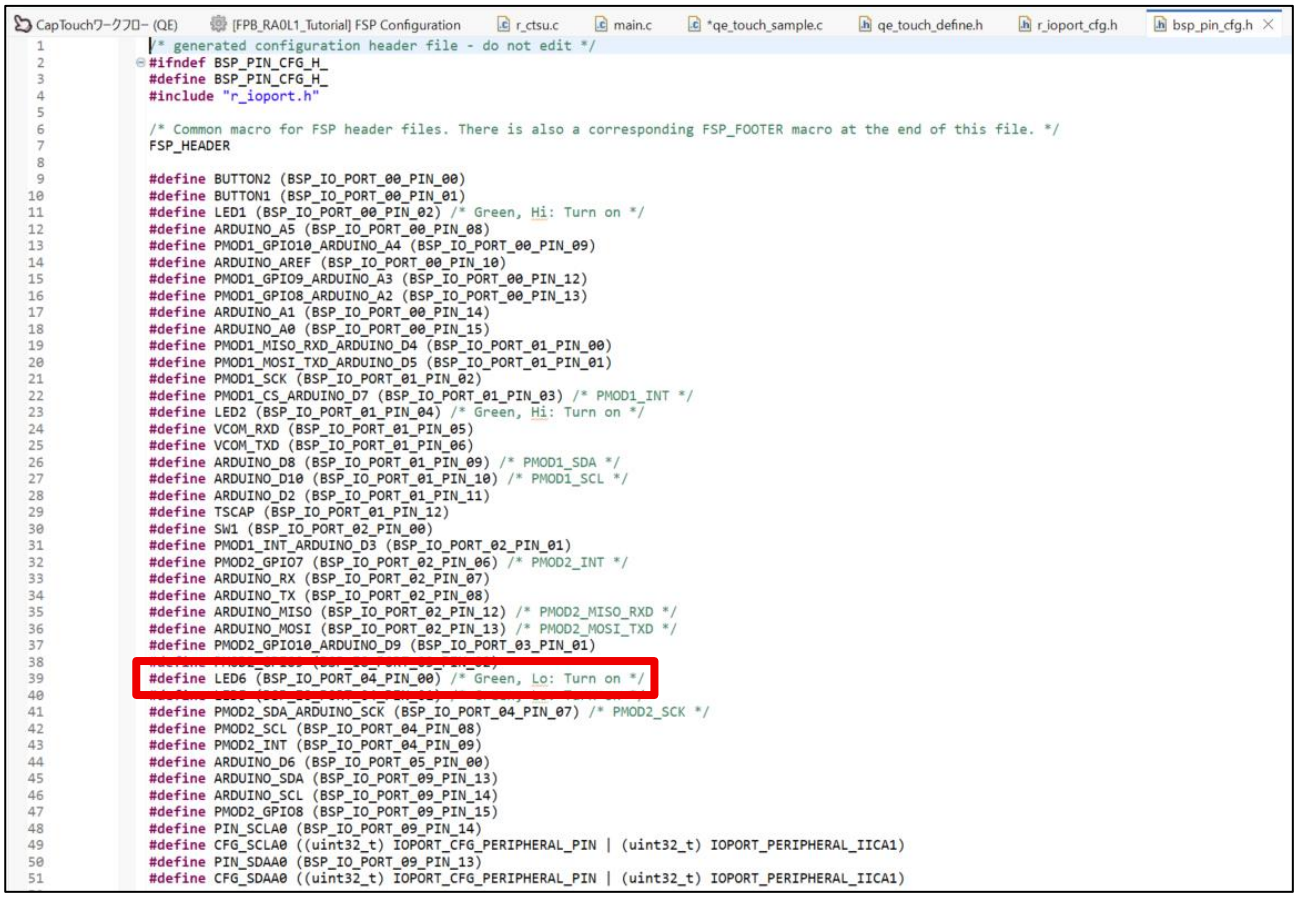


図 5-35. main プログラムの実装

R_IOPORT_PinWrite()関数の第 2 引数には設定するピン番号 = "LED5"、 第 3 引数には出力内容を記入します。

Touch Button 2 をタッチ検出する間は LED6 が点灯するように設定したいので、下記のようにソースコードを編集します。

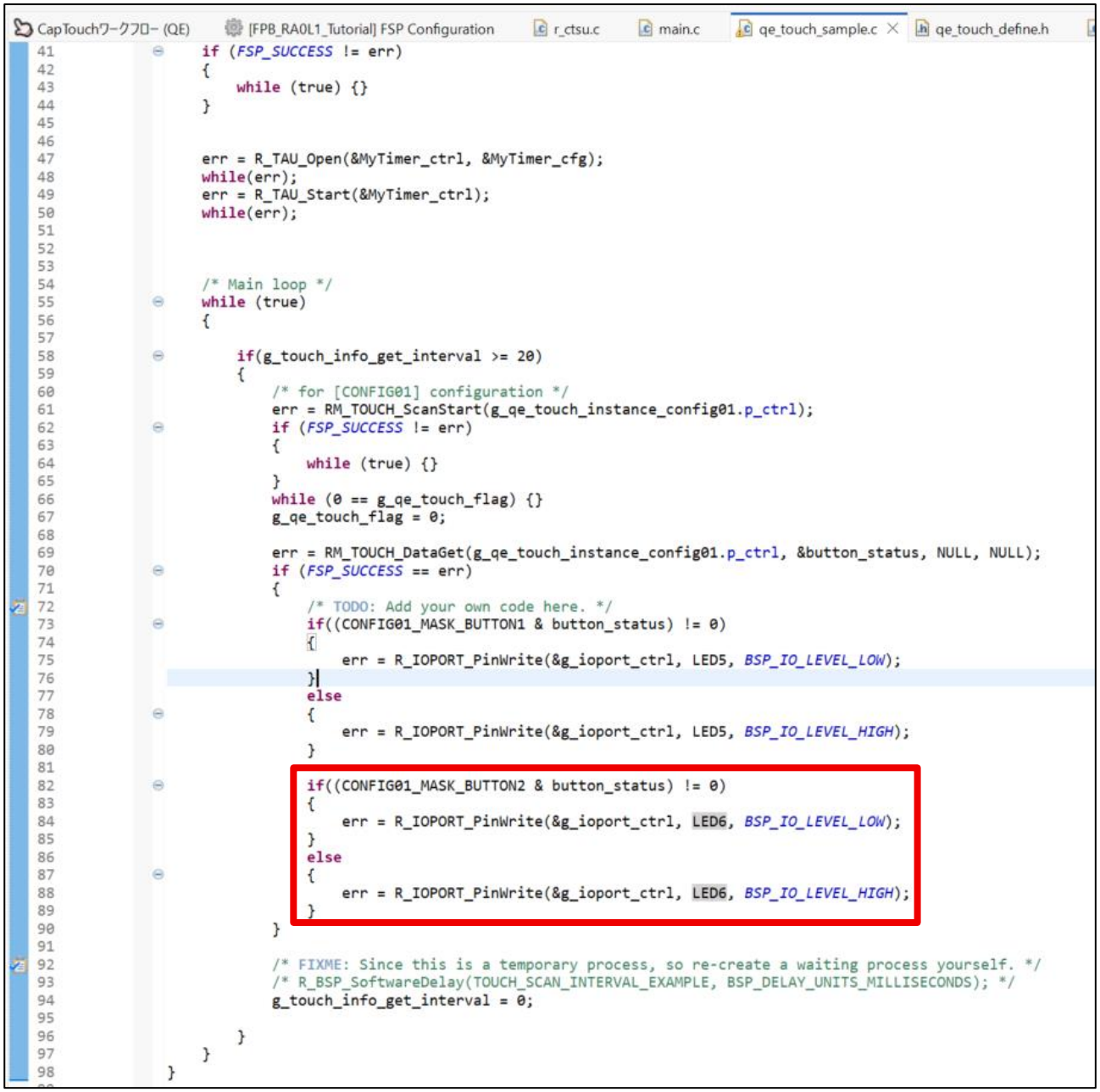
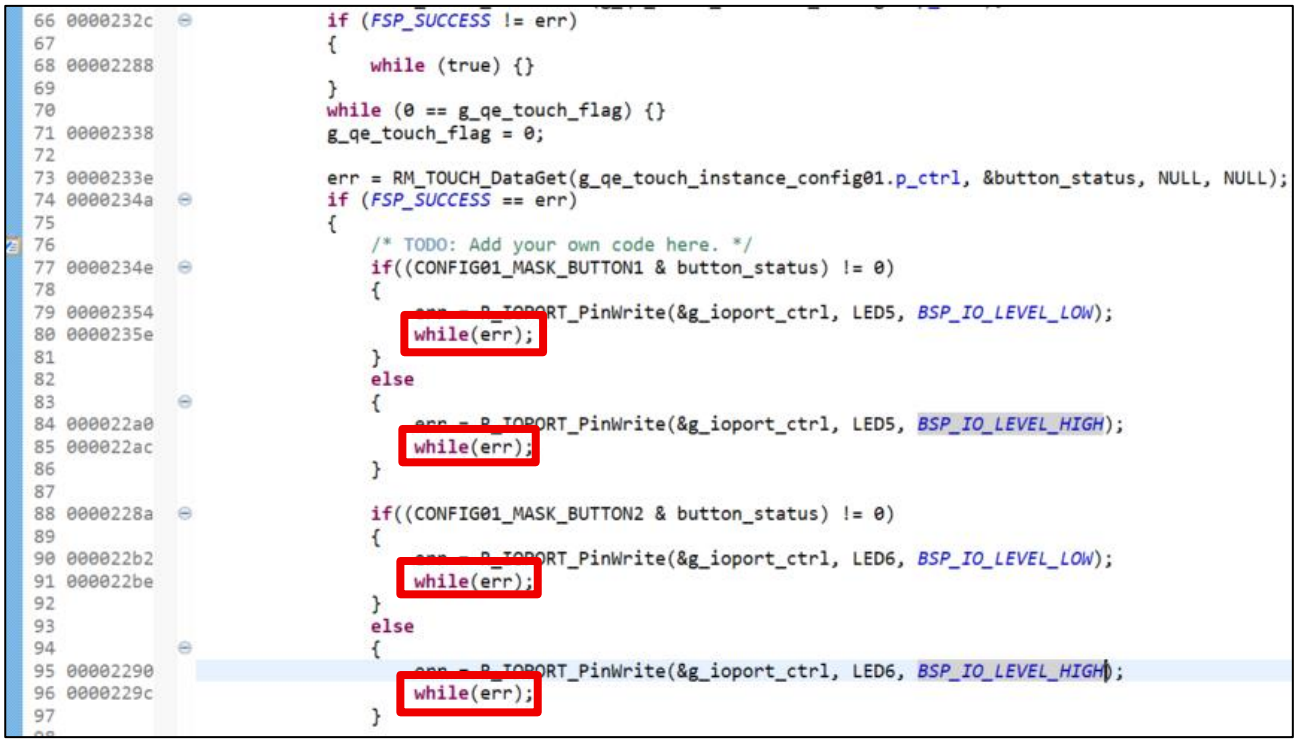


図 5-36. main プログラムの実装

R_IOPORT_PInWrite()関数が異常終了を返した場合を検出するために、関数呼び出し後に while(err);を実装しておきます。



```
66 0000232c             if (FSP_SUCCESS != err)
67                      {
68 00002288                 while (true) {}
69                      }
70                      while (0 == g_qe_touch_flag) {}
71 00002338             g_qe_touch_flag = 0;
72
73 0000233e             err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
74 0000234a             if (FSP_SUCCESS == err)
75                      {
76                          /* TODO: Add your own code here. */
77 0000234e                 if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
78                          {
79 00002354                     err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
80 0000235e                     while(err);
81                          }
82                          else
83                          {
84 000022a0                     err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
85 000022ac                     while(err);
86                          }
87
88 0000228a                 if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
89                          {
90 000022b2                     err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
91 000022be                     while(err);
92                          }
93                          else
94                          {
95 00002290                     err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
96 0000229c                     while(err);
97                          }
```

図 5-37. main プログラムの実装

ここまでのソースコードをテキストで掲載します。qe_touch_sample.c を生成した時点から追記した箇所を青字で示しています。

```
void qe_touch_main(void)
{
    fsp_err_t err;



    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
 g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;

            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
 &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                            BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                            BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
```

```
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
                }
            }

            /* FIXME: Since this is a temporary process,
 so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
 BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
}
```

5.2.3 Touch Button1 のタッチ検出により LED1 と LED2 の点滅動作の開始/停止を制御する

ここでは Touch Button1 のタッチ検出タイミングを検出し、LED1 と LED2 の点滅動作の開始/停止を制御します。この段階では点滅周期は 2 秒固定とします。

(1) LED1 と LED2 の初期値設定

LED1 と LED2 の初期値は LED1 が ON、LED2 が OFF になるように設定します。

メインループが始まる前に設定するので while (true) より前に追記します。

Developer Assistance のリストから R_IOPORT_PinWrite 関数をソースファイルにドラッグアンドドロップします。

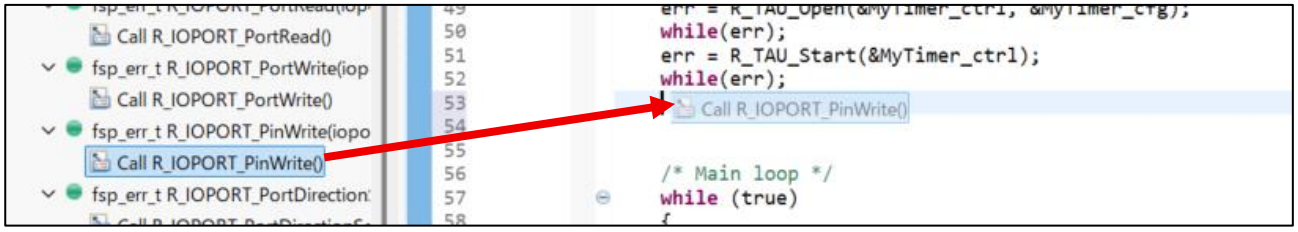


図 5-38. main プログラムの実装

R_IOPORT_PinWrite()関数が追加されました。

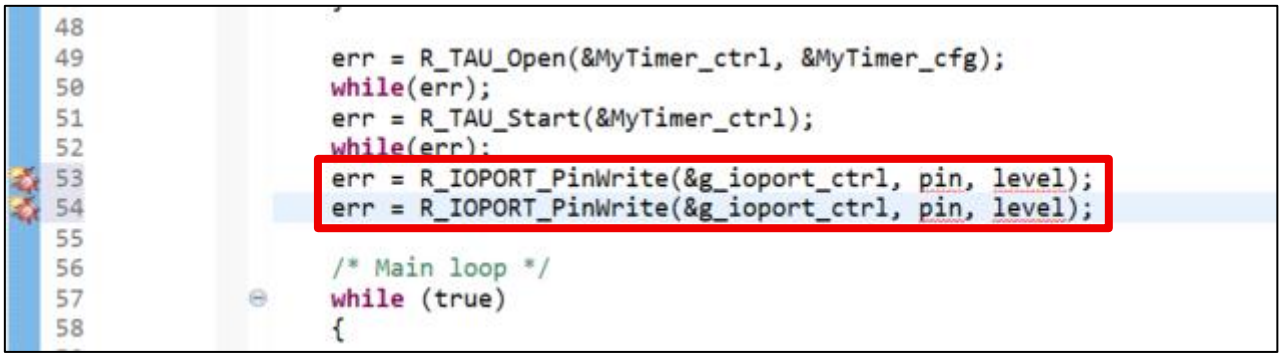


図 5-39. main プログラムの実装

bsp_pin_cfg.h に記述されている下記のマクロ定義を R_IOPORT_PinWrite()関数の引数に使用して LED1,LED2 の ON/OFF を設定できます。

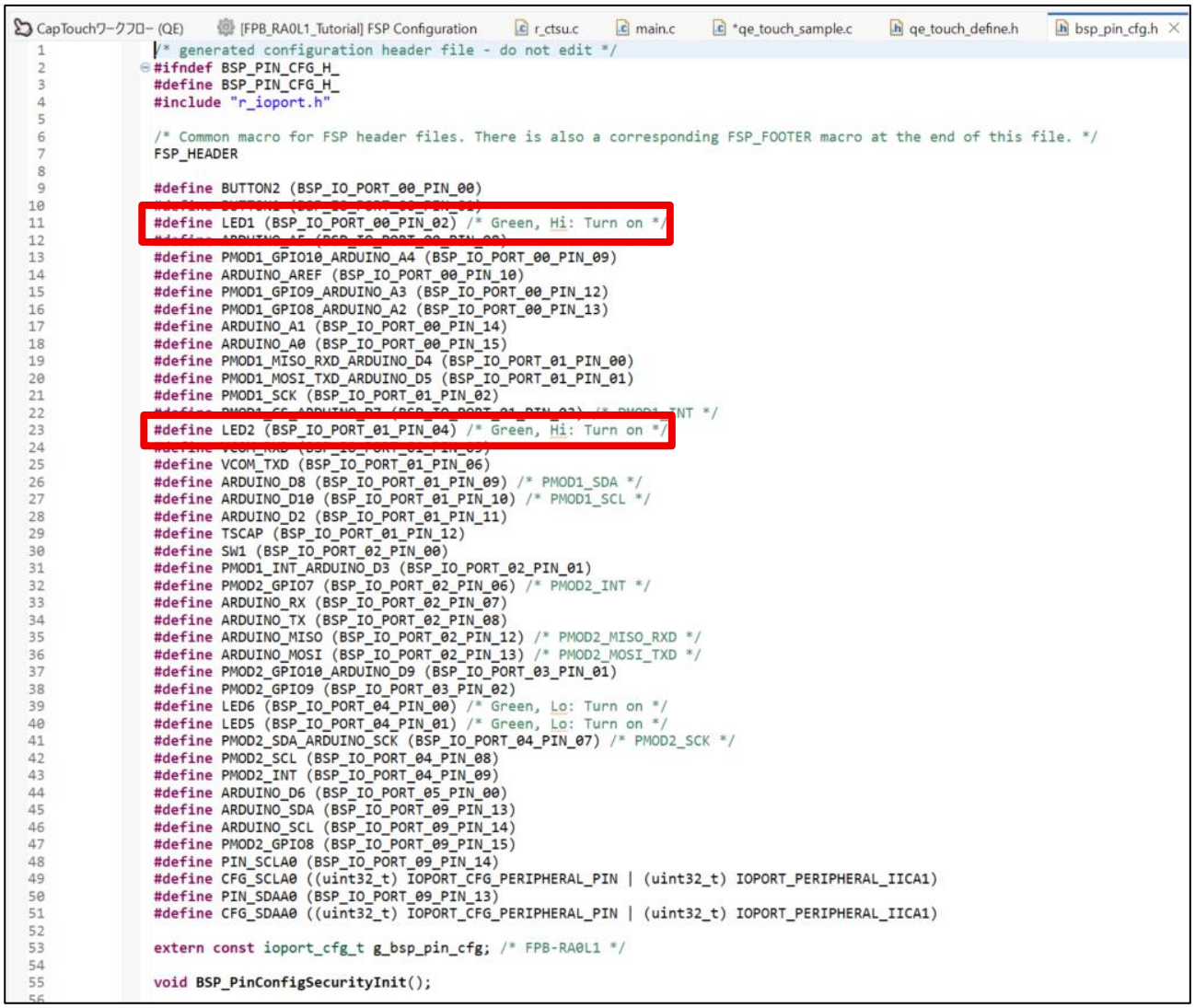
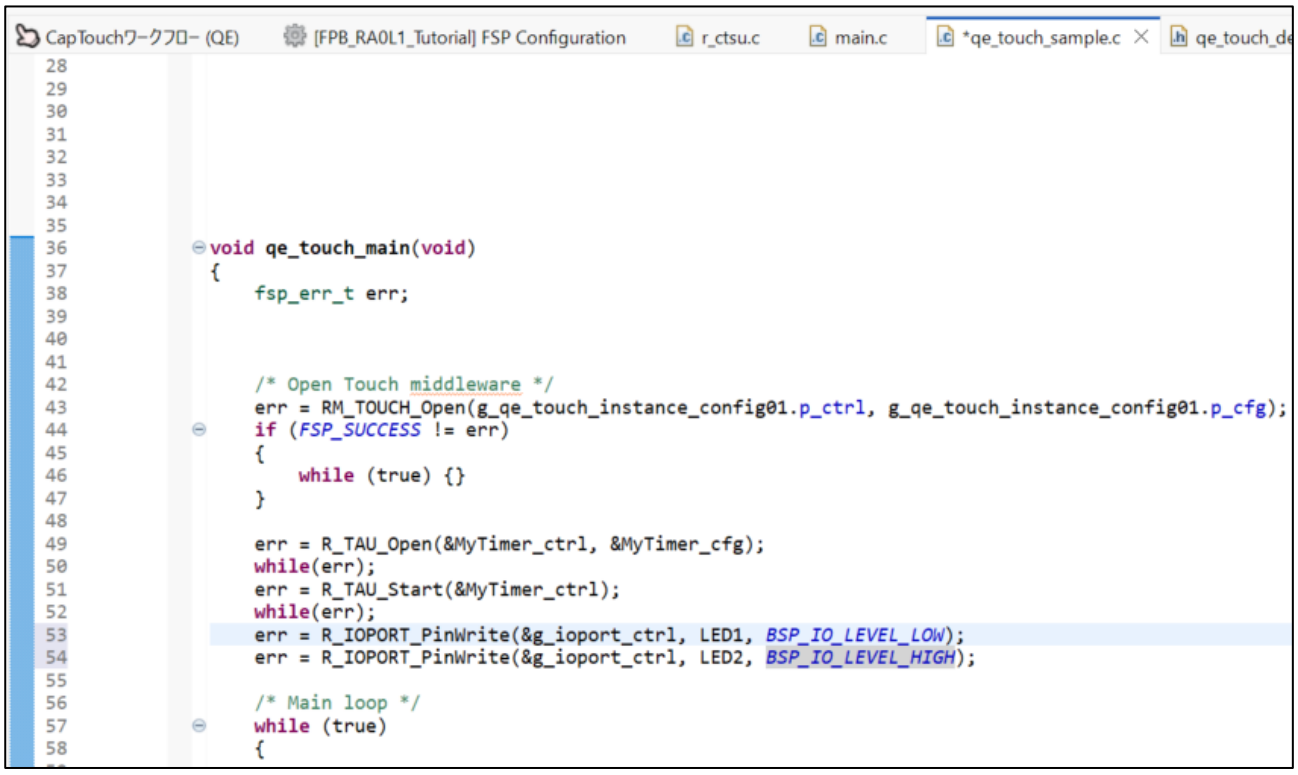


図 5-40. main プログラムの実装

R_IOPORT_PinWrite()関数の第 2 引数には設定するピン番号 = "LED1" および "LED2"、 第 3 引数には出力内容を記入します。

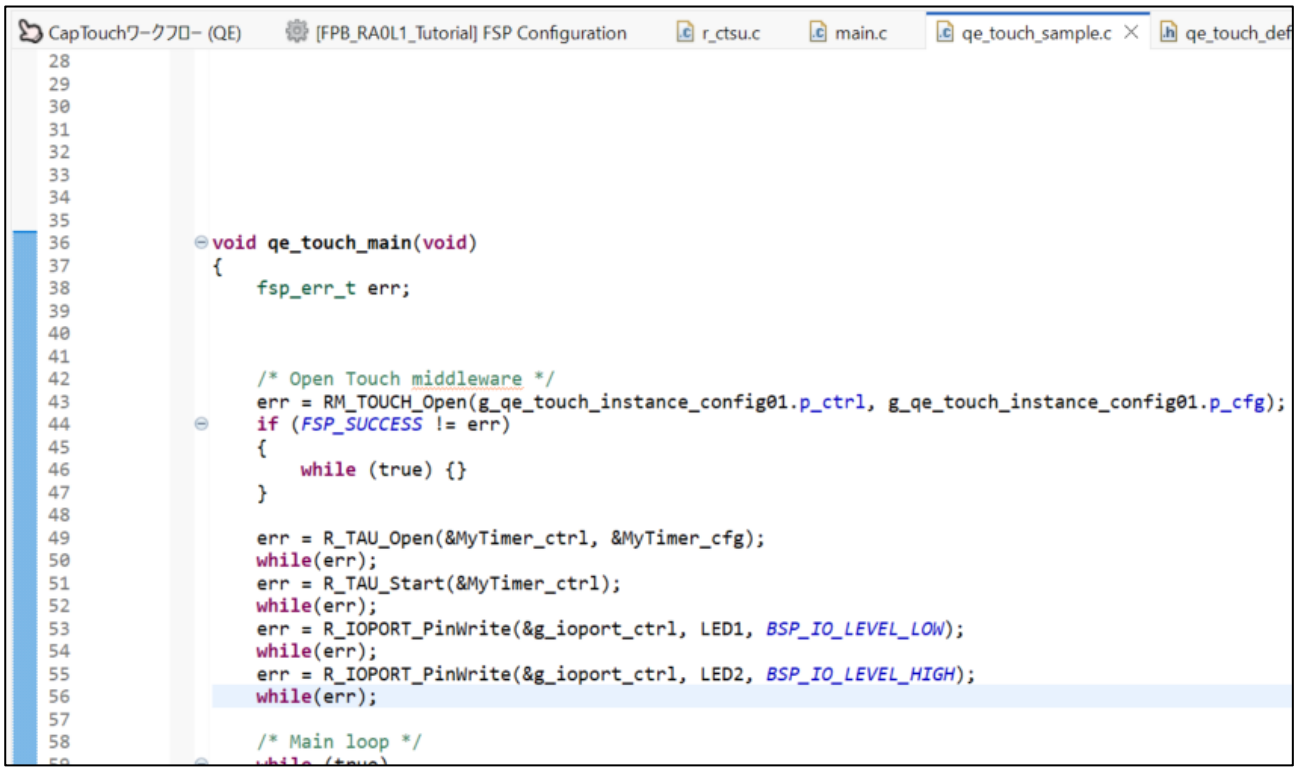
下記のように LED1 と LED2 の初期設定処理を編集します。




```
28
29
30
31
32
33
34
35
36  void qe_touch_main(void)
37  {
38      fsp_err_t err;
39
40
41
42      /* Open Touch middleware */
43      err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
44      if (FSP_SUCCESS != err)
45      {
46          while (true) {}
47      }
48
49      err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
50      while(err);
51      err = R_TAU_Start(&MyTimer_ctrl);
52      while(err);
53      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
54      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
55
56      /* Main loop */
57      while (true)
58      {
```

図 5-41. main プログラムの実装

異常終了を返した場合を検出するために、関数呼び出し後に while(err);を実装しておきます。




```
28
29
30
31
32
33
34
35
36  void qe_touch_main(void)
37  {
38      fsp_err_t err;
39
40
41
42      /* Open Touch middleware */
43      err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl, g_qe_touch_instance_config01.p_cfg);
44      if (FSP_SUCCESS != err)
45      {
46          while (true) {}
47      }
48
49      err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
50      while(err);
51      err = R_TAU_Start(&MyTimer_ctrl);
52      while(err);
53      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
54      while(err);
55      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
56      while(err);
57
58      /* Main loop */
```

図 5-42. main プログラムの実装

(2) 前回の Touch Button 情報の記憶

20ms ごとに Touch Button の情報を取得し button_status に格納しますが、タッチ検出したタイミングを検出するため、前回の Touch Button 情報を格納する変数を新規に作成します。

uint64_t g_button_status_previous;

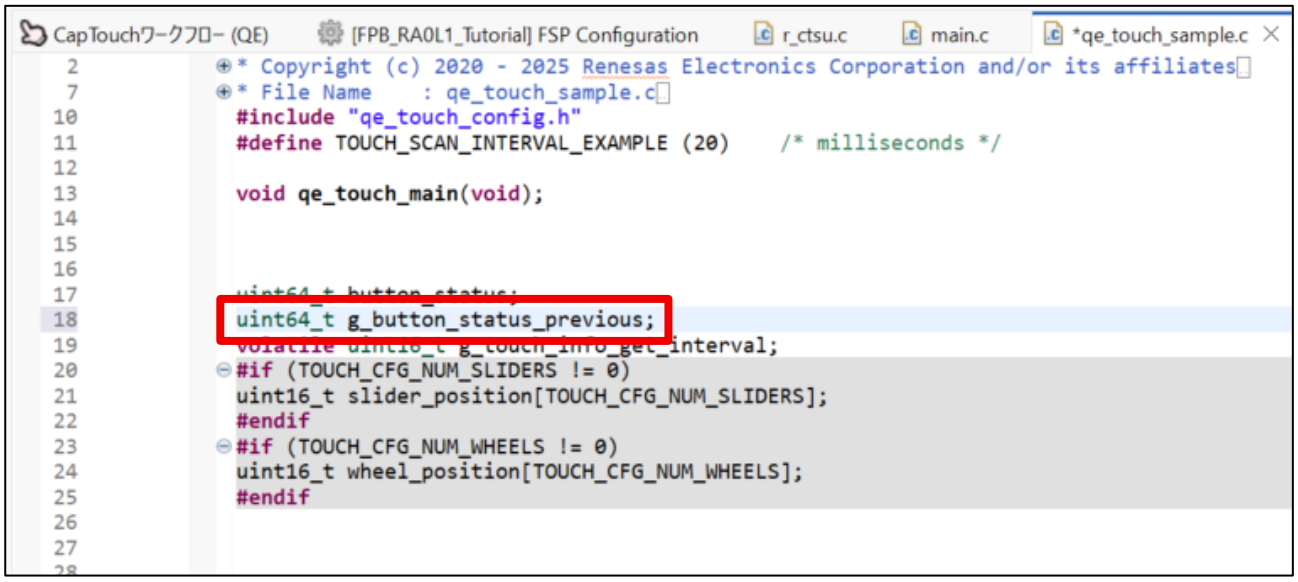
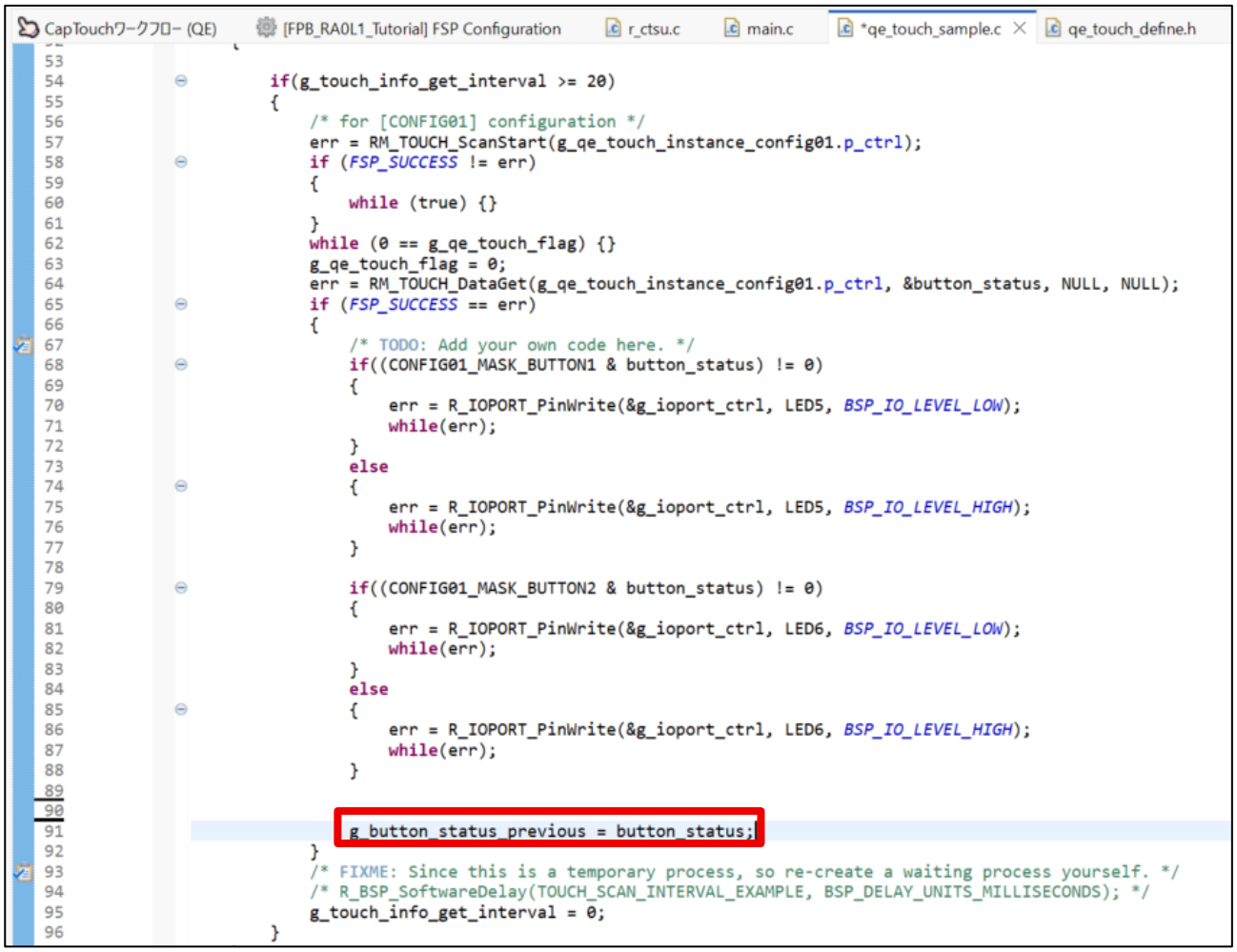


図 5-43. main プログラムの実装

g_button_status_previous = button_status;を if (FSP_SUCCESS == err)条件の最終行に配置することで 20ms 毎に前回値を更新し続けることが出来ます。




```
53
54          if(g_touch_info_get_interval >= 20)
55          {
56              /* for [CONFIG01] configuration */
57              err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
58              if (FSP_SUCCESS != err)
59              {
60                  while (true) {}
61              }
62              while (0 == g_qe_touch_flag) {}
63              g_qe_touch_flag = 0;
64              err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl, &button_status, NULL, NULL);
65              if (FSP_SUCCESS == err)
66              {
67                  /* TODO: Add your own code here. */
68                  if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
69                  {
70                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
71                      while(err);
72                  }
73                  else
74                  {
75                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
76                      while(err);
77                  }
78
79                  if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
80                  {
81                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
82                      while(err);
83                  }
84                  else
85                  {
86                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
87                      while(err);
88                  }
89
90
91                  g_button_status_previous = button_status;
92              }
93              /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
94              /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
95              g_touch_info_get_interval = 0;
96          }
```

図 5-44. main プログラムの実装

(3) Touch Button1 のタッチ検出プログラムの実装

Touch Button1 を押したタイミングを検出するための条件文を作成します。

タッチ検出条件は以下２つの論理積となります。

- 今回 Touch Button1 が押されたか？
  ((CONFIG01_MASK_BUTTON1 & button_status) != 0)
- 前回 Touch Button1 が押されていないか？
  ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0)

```
65              if (FSP_SUCCESS == err)
66              {
67                  /* TODO: Add your own code here. */
68                  if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
69                  {
70                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
71                      while(err);
72                  }
73                  else
74                  {
75                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
76                      while(err);
77                  }
78
79                  if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
80                  {
81                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_LOW);
82                      while(err);
83                  }
84                  else
85                  {
86                      err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6, BSP_IO_LEVEL_HIGH);
87                      while(err);
88                  }
89
90                  if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
91                  {
92
93                  }
94                  g_button_status_previous = button_status;
95              }
96              /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
97              /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
98              g_touch_info_get_interval = 0;
99          }
100     }
101 }
```

図 5-45. main プログラムの実装

(4) Touch Button1 タッチ検出時のインクリメント処理の実装

Touch Button1 を押した際にインクリメントされる変数を作成し、条件文に組み込みます。

uint8_t g_button1_push_count; を宣言します。

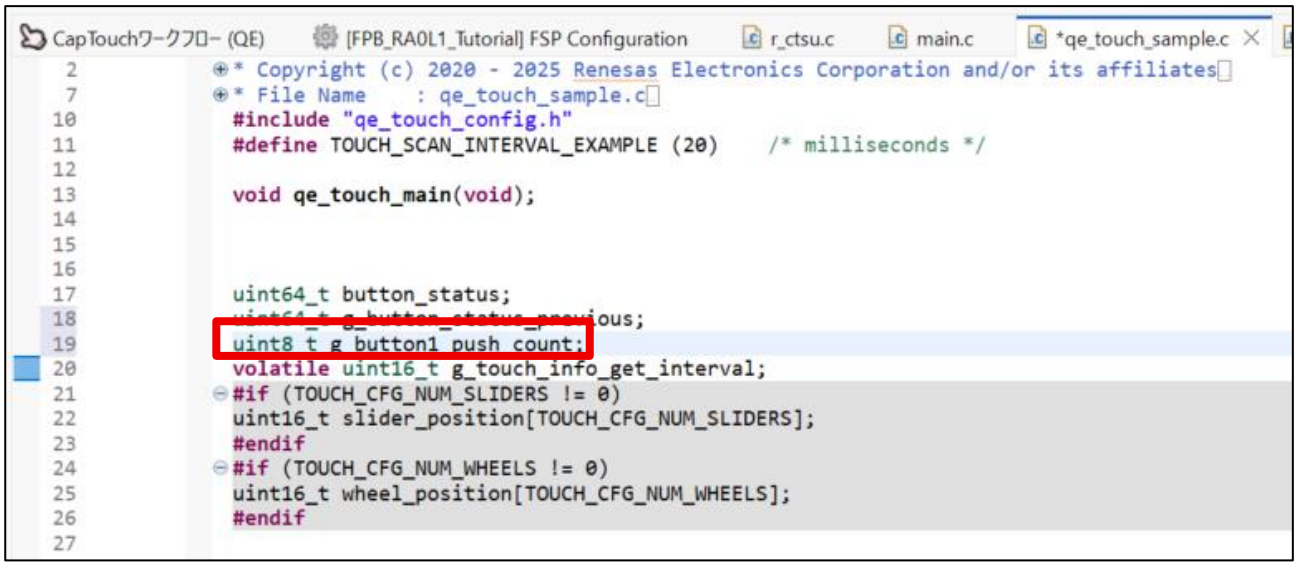


図 5-46. main プログラムの実装

Touch Button1 を押したタイミングでインクリメントします。

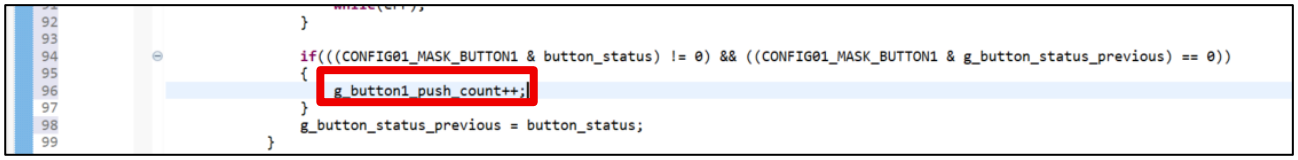


図 5-47. main プログラムの実装

(5) Touch Button1 の再タッチ検出によるカウント変数クリア

Touch Button 1 をもう 1 度タッチ検出すると g_button1_push_count 変数を初期化するように条件文を作成します。

ここまでが点滅動作の開始/停止を実現するプログラムとなります。変数の値が 1 の場合、点滅開始状態を意味します。

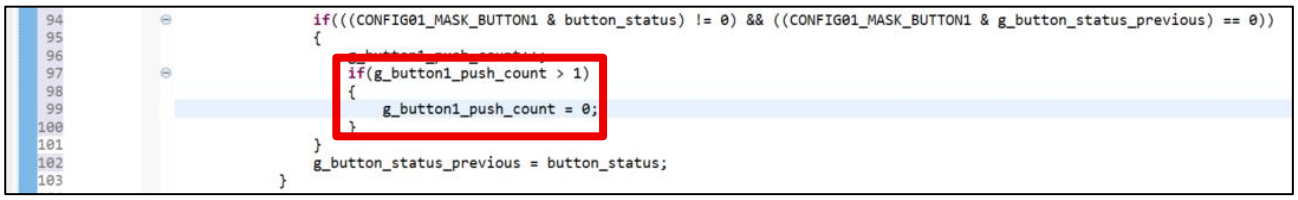


図 5-48. main プログラムの実装

(6) LED 点滅開始状態の条件文追加

(d)で実装した変数を用いて、点滅開始状態で制御するための条件文を追加します。

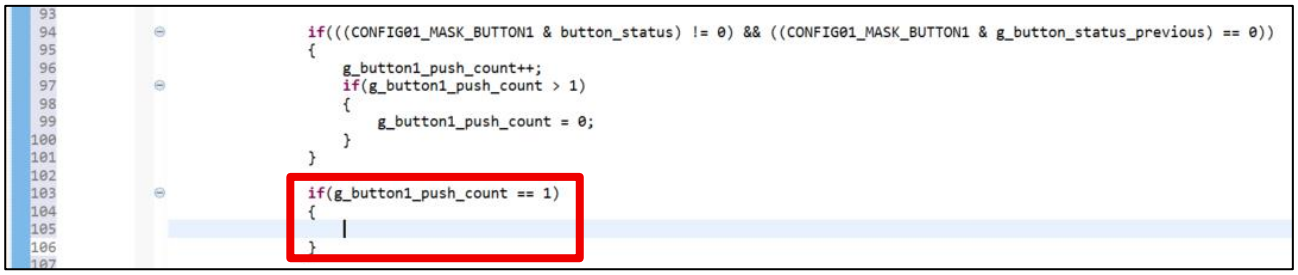


図 5-49. main プログラムの実装

(7) LED 点滅周期制御用の変数追加と実装

LED の点滅周期 2 秒を制御するため変数を新規に作成し、条件文に組み込みます。

volatile uint16_t g_led_blink_interval;

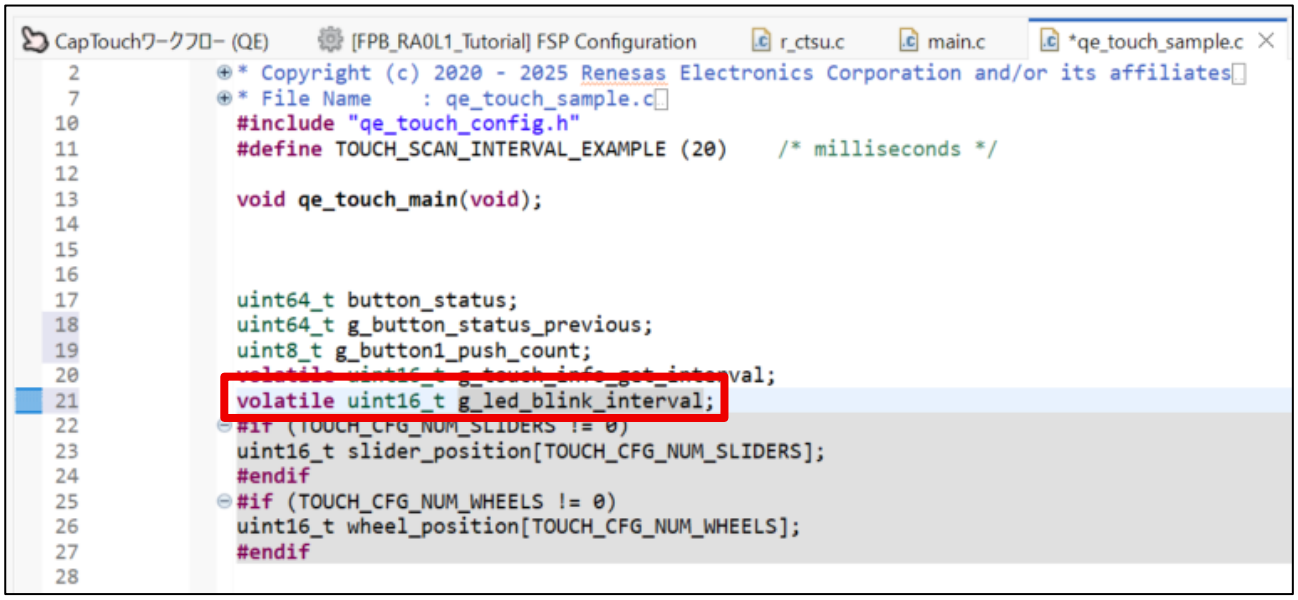


図 5-50. main プログラムの実装

MyCallback 関数内で g_led_blink_interval 変数をインクリメントさせます。

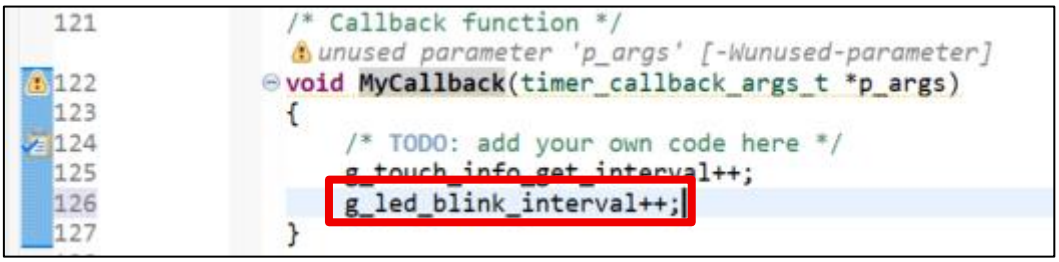


図 5-51. main プログラムの実装

g_led_blink_interval 変数を用いて 2 秒経過を判定する条件を(e)の中に作成します。

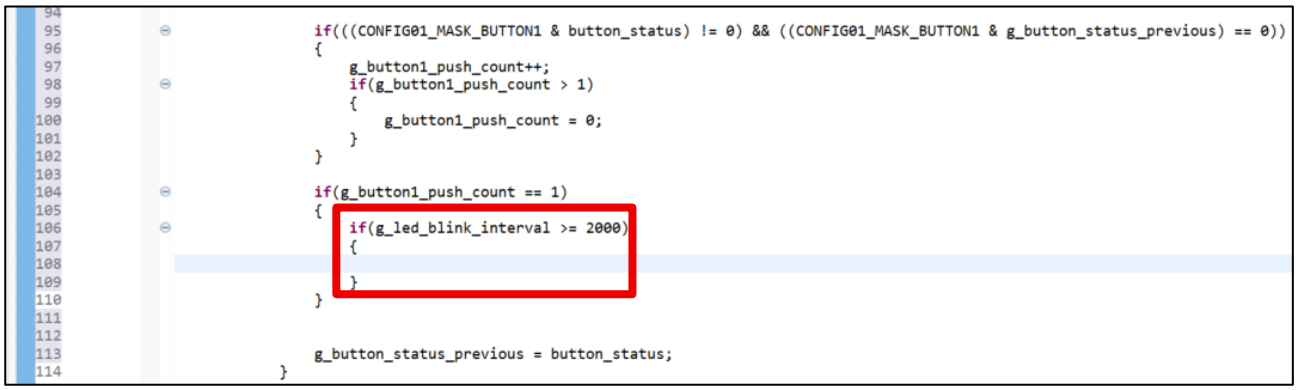


図 5-52. main プログラムの実装

2 秒経過の条件成立時に g_led_blink_interval 変数を初期化します。

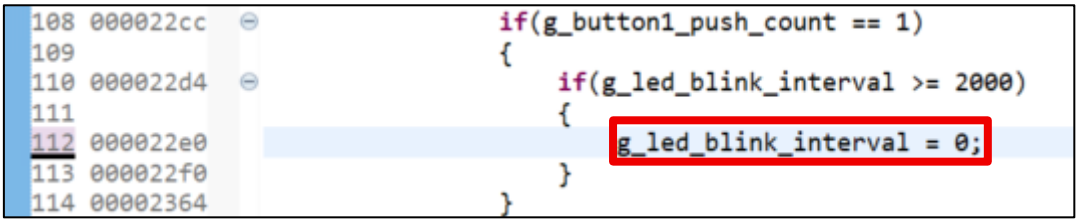


図 5-53. main プログラムの実装

また、Touch Button1 が押されたタイミングでも g_led_blink_interval 変数を初期化します。

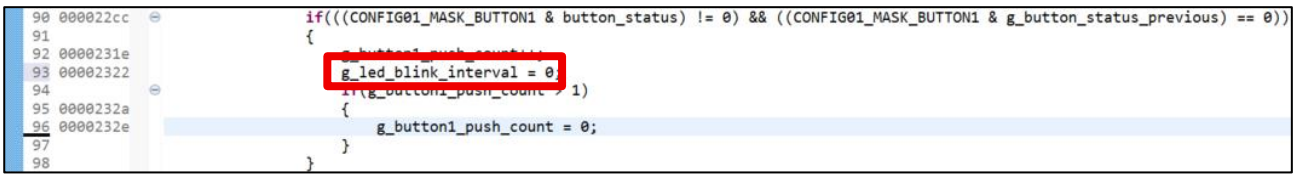


図 5-54. main プログラムの実装

(8) LED1 と LED2 の点灯状態を切り替えるプログラムの作成

Developer Assistance のリストから R_IOPORT_PinRead()関数をソースファイルにドラッグアンドドロップします。

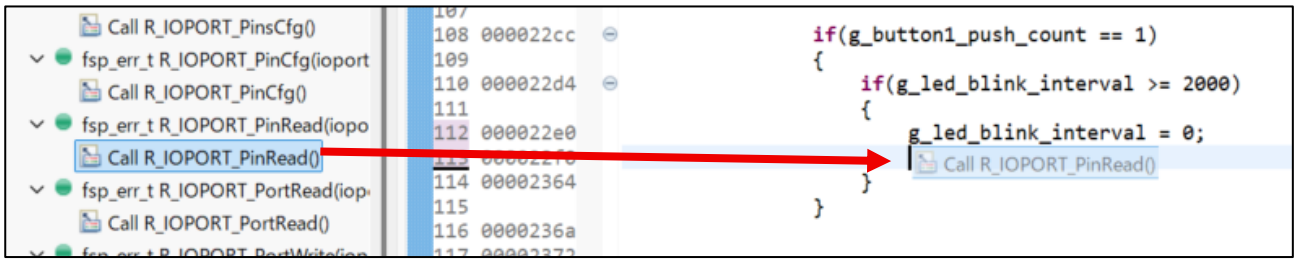


図 5-55. main プログラムの実装

R_IOPORT_PinRead()関数が追加されました。

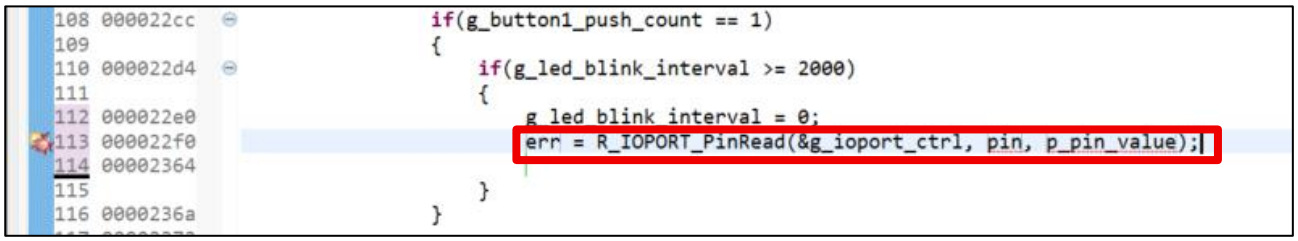


図 5-56. main プログラムの実装

R_IOPORT_PinRead()関数は、第 2 引数で指定した Pin の情報を第 3 引数に格納します。第 3 引数に指定するための変数を作成します。

bsp_io_level_t value;

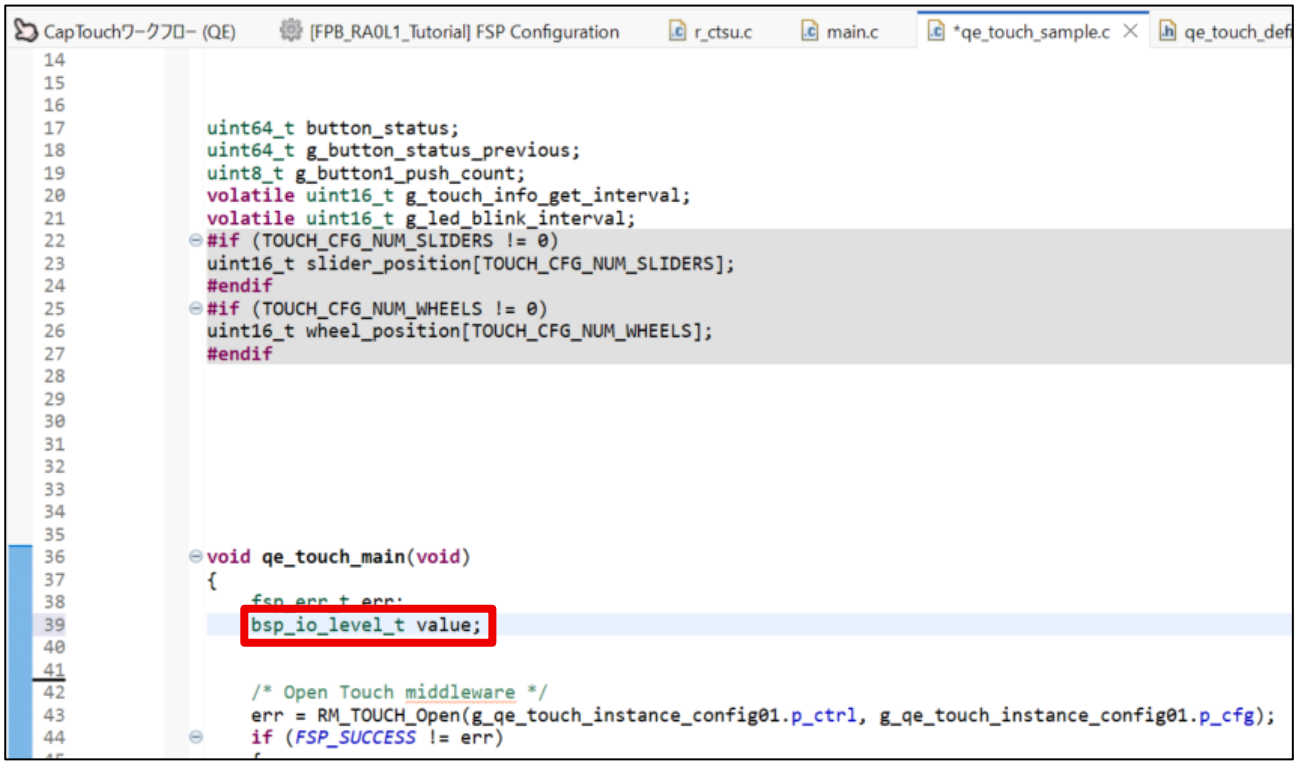


図 5-57. main プログラムの実装

LED1 の状態を取得したいので下記のように編集します。


```
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_LOW);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5, BSP_IO_LEVEL_HIGH);
                    while(err);
                }

                if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, pin, level);
                    while(err);
                }

                if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
                {
                    g_button1_push_count++;
                    if(g_button1_push_count > 1)
                    {
                        g_button1_push_count = 0;
                    }
                }

                if(g_button1_push_count == 1)
                {
                    if(g_led_blink_interval >= 2000)
                    {
                        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                    }
                }


                g_button_status_previous = button_status;
            }

            /* FIXME: Since this is a temporary process, so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE, BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;

        }
    }
}
```

図 5-58. main プログラムの実装

異常終了を返した場合を検出するために、関数呼び出し後に while(err);を実装しておきます。

```
108 000022cc                if(g_button1_push_count == 1)
109                         {
110 000022d4                    if(g_led_blink_interval >= 2000)
111                             {
112 000022e0                        g_led_blink_interval = 0;
113 000022f0                        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
114 00002364                        while(err);
115                             }
116 0000236a                }
```

図 5-59. main プログラムの実装

R_IOPORT_PinRead 関数の第 3 引数を使用して LED1 と LED2 の点滅切り替えを行うプログラムを作成します。

条件文は下記のようになります。

```
108 000022cc                if(g_button1_push_count == 1)
109                         {
110 000022d4                    if(g_led_blink_interval >= 2000)
111                             {
112 000022e0                        g_led_blink_interval = 0;
113 000022f0                        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
114 00002364                        while(err);
115                                 if(value == BSP_IO_LEVEL_HIGH)
116 0000236a                        {
117 00002372
118 00002376                        }
119 00002382                        else
120                                 {
121
122                                 }
123 00002388                    }
124 00002394                }
125 00002398
```

図 5-60. main プログラムの実装

LED1 の出力状態を判定し、LED1 と LED2 の出力を以下のように設定します。

- LED1 が HIGH の場合、LED1 を LOW、LED2 を HIGH に設定する
- LED1 が LOW の場合、LED1 を HIGH、LED2 を LOW に設定する

```
108 000022cc                if(g_button1_push_count == 1)
109                         {
110 000022d4                    if(g_led_blink_interval >= 2000)
111                             {
112 000022e0                        g_led_blink_interval = 0;
113 000022f0                        err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
114 00002364                        while(err);
115                                 if(value == BSP_IO_LEVEL_HIGH)
116 0000236a                        {
117 00002372                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
118 00002376                            while(err);
119 00002382                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
120                                     while(err);
121                                 }
122                                 else
123 00002388                        {
124 00002394                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
125 00002398                            while(err);
126 000023a2                            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
127                                     while(err);
128                                 }
129                             }
130                         }
```

図 5-61. main プログラムの実装

ここまでのソースコードをテキストで掲載します。qe_touch_sample.c を生成した時点から追記した箇所を青字で示しています。

```
void qe_touch_main(void);

uint64_t button_status;
uint64_t g_button_status_previous;
uint8_t g_button1_push_count;
volatile uint16_t g_touch_info_get_interval;
volatile uint16_t g_led_blink_interval;
#if (TOUCH_CFG_NUM_SLIDERS != 0)
uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
#endif
#if (TOUCH_CFG_NUM_WHEELS != 0)
uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];
#endif

void qe_touch_main(void)
{
    fsp_err_t err;
    bsp_io_level_t value;

    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
 g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;
            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
 &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
BSP_IO_LEVEL_LOW);
```

```
                while(err);
            }
            else
            {
                err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
BSP_IO_LEVEL_HIGH);
                while(err);
            }

            if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
            {
                err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
BSP_IO_LEVEL_LOW);
                while(err);
            }
            else
            {
                err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
BSP_IO_LEVEL_HIGH);
                while(err);
            }

            if(((CONFIG01_MASK_BUTTON1 & button_status) != 0)
&& ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
            {
                g_button1_push_count++;
                g_led_blink_interval = 0;
                if(g_button1_push_count > 1)
                {
                    g_button1_push_count = 0;
                }
            }

            if(g_button1_push_count == 1)
            {
                if(g_led_blink_interval >= 2000)
                {
                    g_led_blink_interval = 0;
                    err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                    while(err);
                    if(value == BSP_IO_LEVEL_HIGH)
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
BSP_IO_LEVEL_LOW);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
BSP_IO_LEVEL_HIGH);
                        while(err);
                    }
                    else
                    {
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
BSP_IO_LEVEL_HIGH);
                        while(err);
                        err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
BSP_IO_LEVEL_LOW);
                        while(err);
                    }
                }
```

```
                }
                g_button_status_previous = button_status;
            }
            /* FIXME: Since this is a temporary process,
 so re-create a waiting process yourself. */
            /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
 BSP_DELAY_UNITS_MILLISECONDS); */
            g_touch_info_get_interval = 0;
        }
    }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
    g_led_blink_interval++;
}
```

5.2.4 Touch Button2 のタッチ検出により点滅周期を変更する

ここでは、LED 点滅動作中に Touch Button 2 をタッチ検出するたびに、点滅周期を 2 秒→1 秒→0.5 秒のローテーションで変更するようにプログラムを作成します。

(1) Touch Button2 のタッチ検出プログラムの実装

Touch Button2 を押したタイミングを検出するための条件文を作成します。

LED 点滅動作中の操作に限定するため、if(g_button1_push_count == 1)条件内に記述します。

(3)-(c)と同様に Touch Button2 の条件文を作成します。

条件文は下記のようになります。

```
108
109             if(g_button1_push_count == 1)
110             {
111                 if(g_led_blink_interval >= 2000)
112                 {
113                     g_led_blink_interval = 0;
114                     err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
115                     while(err);
116                     if(value == BSP_IO_LEVEL_HIGH)
117                     {
118                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
119                         while(err);
120                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
121                         while(err);
122                     }
123                     else
124                     {
125                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
126                         while(err);
127                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
128                         while(err);
129                     }
130                 }
131
132                 if(((CONFIG01_MASK_BUTTON2 & button_status) != 0) && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
133                 {
134
135                 }
136             }
```

図 5-62. main プログラムの実装

(2) Touch Button2 タッチ検出時のインクリメント処理の実装

Touch Button2 を押した際にインクリメントされる変数を作成します。

uint8_t g_led_blink_table_index;

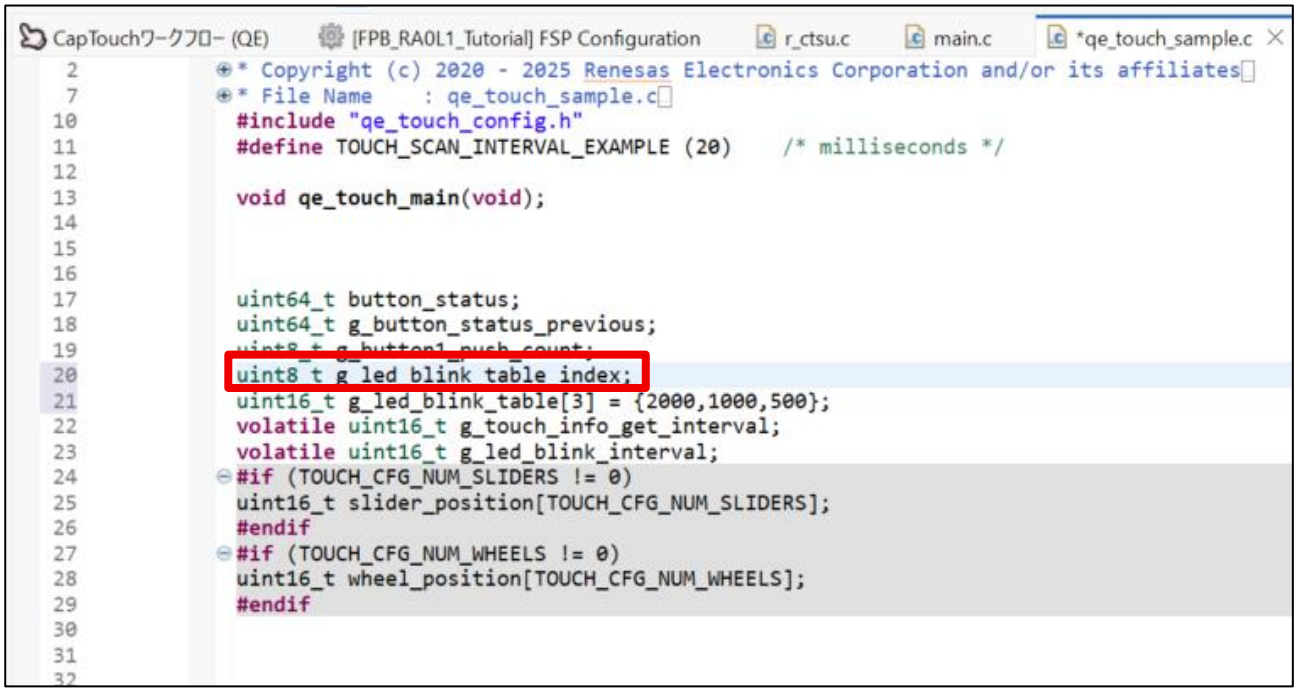


図 5-63. main プログラムの実装

(a)で作成した条件文の中にインクリメント処理を追加します。

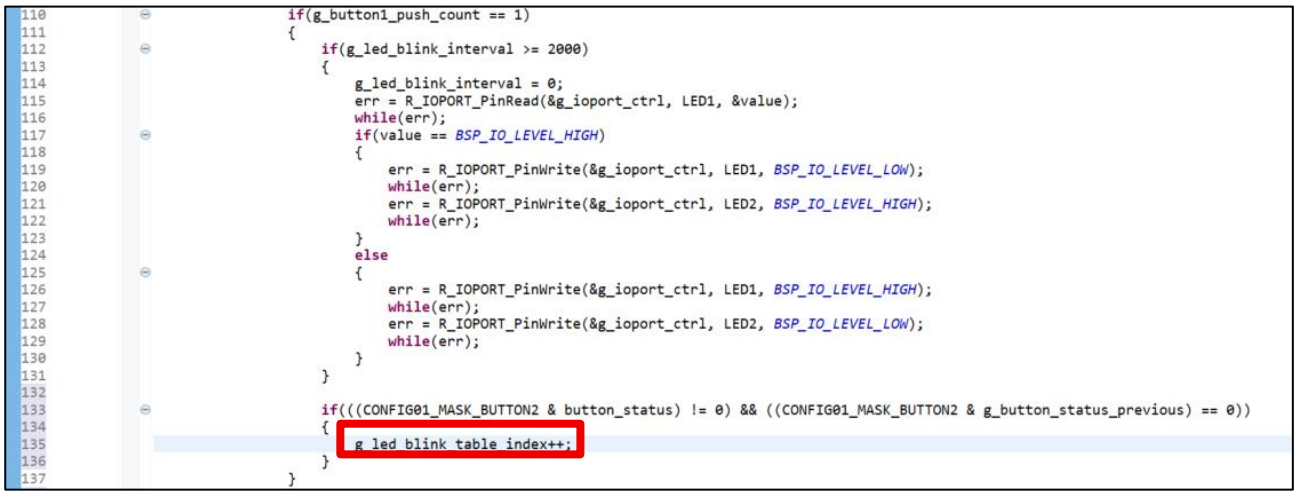


図 5-64. main プログラムの実装

(3) TouchButton2 タッチ検出による点滅周期のローテート制御の実装

Touch Button2 をタッチ検出するごとに点滅周期が 2 秒→1 秒→0.5 秒のローテーションとなるように、3 度押されたタイミングでインクリメントされた変数を初期化するように条件文を作成します。

```
110             if(g_button1_push_count == 1)
111             {
112                 if(g_led_blink_interval >= 2000)
113                 {
114                     g_led_blink_interval = 0;
115                     err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
116                     while(err);
117                     if(value == BSP_IO_LEVEL_HIGH)
118                     {
119                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
120                         while(err);
121                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
122                         while(err);
123                     }
124                     else
125                     {
126                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
127                         while(err);
128                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
129                         while(err);
130                     }
131                 }
132
133                 if(((CONFIG01_MASK_BUTTON2 & button_status) != 0) && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
134                 {
135                     g_led_blink_table_index++;
136                     if(g_led_blink_table_index > 2)
137                     {
138                         g_led_blink_table_index = 0;
139                     }
140                 }
141             }
```

図 5-65. main プログラムの実装

(4) 点滅周期を定義するためのテーブル変数の追加

(3)-(g)では点滅周期を 2 秒固定としていましたが、点滅周期を 2 秒→1 秒→0.5 秒のローテーションに変更するため、周期を定義する配列変数を作成します。

uint16_t g_led_blink_table[3] = {2000,1000,500};

CapTouchワークフロー (QE) | [FPB_RA0L1_Tutorial] FSP Configuration | r_ctsu.c | main.c | *qe_touch_sample.c

```
 2  * Copyright (c) 2020 - 2025 Renesas Electronics Corporation and/or its affiliates
 7  * File Name    : qe_touch_sample.c
10  #include "qe_touch_config.h"
11  #define TOUCH_SCAN_INTERVAL_EXAMPLE (20)    /* milliseconds */
12
13  void qe_touch_main(void);
14
15
16
17  uint64_t button_status;
18  uint64_t g_button_status_previous;
19  uint8_t g_button1_push_count;
20  uint16_t g_led_blink_table[3] = {2000,1000,500};
21  volatile uint16_t g_touch_info_get_interval;
22  volatile uint16_t g_led_blink_interval;
23  #if (TOUCH_CFG_NUM_SLIDERS != 0)
24  uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
25  #endif
26  #if (TOUCH_CFG_NUM_WHEELS != 0)
27  uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];
28  #endif
29
30
31
```

図 5-66. main プログラムの実装

(5) 点滅周期判定文の修正

(3)-(f)で作成した 2 秒経過判定文を (4)-(d)で作成した配列に変更します。

```
110             if(g_button1_push_count == 1)
111             {
112                 if(g_led_blink_interval >= g_led_blink_table[g_led_blink_table_index])
113                 {
114                     g_led_blink_interval = 0;
115                     err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
116                     while(err);
117                     if(value == BSP_IO_LEVEL_HIGH)
118                     {
119                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
120                         while(err);
121                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
122                         while(err);
123                     }
124                     else
125                     {
126                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_HIGH);
127                         while(err);
128                         err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_LOW);
129                         while(err);
130                     }
131                 }
132
133                 if(((CONFIG01_MASK_BUTTON2 & button_status) != 0) && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
134                 {
135                     g_led_blink_table_index++;
136                     if(g_led_blink_table_index > 2)
137                     {
138                         g_led_blink_table_index = 0;
139                     }
140                 }
141             }
```

図 5-67. main プログラムの実装

(6) 点滅開始時の点滅周期の固定化

点滅動作開始時の点滅周期が常に 2 秒周期となるように、点滅の開始タイミングで g_led_blink_table_index 変数の初期化を行います。

そのため初期化処理は if(g_button1_push_count > 1)条件内に追加します。

```
101 000022cc            if(((CONFIG01_MASK_BUTTON1 & button_status) != 0) && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
102                     {
103 0000231e                g_button1_push_count++;
104 00002322                if(g_button1_push_count > 1)
105                         {
106 0000232a                    g_button1_push_count = 0;
107                             g_led_blink_table_index = 0
108                         }
109                     }
```

図 5-68. main プログラムの実装

以上でサンプルプログラムのコーディングは完了です。

ここまでのソースコードをテキストで掲載します。qe_touch_sample.c を生成した時点から追記した箇所を青字で示しています。

```
void qe_touch_main(void);

uint64_t button_status;
uint64_t g_button_status_previous;
uint8_t g_button1_push_count;
uint8_t g_led_blink_table_index;
uint16_t g_led_blink_table[3] = {2000,1000,500};
volatile uint16_t g_touch_info_get_interval;
volatile uint16_t g_led_blink_interval;
#if (TOUCH_CFG_NUM_SLIDERS != 0)
uint16_t slider_position[TOUCH_CFG_NUM_SLIDERS];
#endif
#if (TOUCH_CFG_NUM_WHEELS != 0)
uint16_t wheel_position[TOUCH_CFG_NUM_WHEELS];
#endif

void qe_touch_main(void)
{
    fsp_err_t err;
    bsp_io_level_t value;

    /* Open Touch middleware */
    err = RM_TOUCH_Open(g_qe_touch_instance_config01.p_ctrl,
                        g_qe_touch_instance_config01.p_cfg);
    if (FSP_SUCCESS != err)
    {
        while (true) {}
    }

    err = R_TAU_Open(&MyTimer_ctrl, &MyTimer_cfg);
    while(err);
    err = R_TAU_Start(&MyTimer_ctrl);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1, BSP_IO_LEVEL_LOW);
    while(err);
    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2, BSP_IO_LEVEL_HIGH);
    while(err);

    /* Main loop */
    while (true)
    {

        if(g_touch_info_get_interval >= 20)
        {
            /* for [CONFIG01] configuration */
            err = RM_TOUCH_ScanStart(g_qe_touch_instance_config01.p_ctrl);
            if (FSP_SUCCESS != err)
            {
                while (true) {}
            }
            while (0 == g_qe_touch_flag) {}
            g_qe_touch_flag = 0;
            err = RM_TOUCH_DataGet(g_qe_touch_instance_config01.p_ctrl,
                                   &button_status, NULL, NULL);
            if (FSP_SUCCESS == err)
            {
                /* TODO: Add your own code here. */
                if((CONFIG01_MASK_BUTTON1 & button_status) != 0)
                {
```

```
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                    BSP_IO_LEVEL_LOW);
            while(err);
        }
        else
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED5,
                                    BSP_IO_LEVEL_HIGH);
            while(err);
        }

        if((CONFIG01_MASK_BUTTON2 & button_status) != 0)
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                    BSP_IO_LEVEL_LOW);
            while(err);
        }
        else
        {
            err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED6,
                                    BSP_IO_LEVEL_HIGH);
            while(err);
        }

        if(((CONFIG01_MASK_BUTTON1 & button_status) != 0)
          && ((CONFIG01_MASK_BUTTON1 & g_button_status_previous) == 0))
        {
            g_button1_push_count++;
            if(g_button1_push_count > 1)
            {
                g_button1_push_count = 0;
                g_led_blink_table_index = 0;
            }
        }

        if(g_button1_push_count == 1)
        {
            if(g_led_blink_interval
                >= g_led_blink_table[g_led_blink_table_index])
            {
                g_led_blink_interval = 0;
                err = R_IOPORT_PinRead(&g_ioport_ctrl, LED1, &value);
                while(err);
                if(value == BSP_IO_LEVEL_HIGH)
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
                                            BSP_IO_LEVEL_LOW);
                    while(err);
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
                }
                else
                {
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED1,
                                            BSP_IO_LEVEL_HIGH);
                    while(err);
                    err = R_IOPORT_PinWrite(&g_ioport_ctrl, LED2,
                                            BSP_IO_LEVEL_LOW);
```

```
                        while(err);
                    }
                }

                if(((CONFIG01_MASK_BUTTON2 & button_status) != 0)
                 && ((CONFIG01_MASK_BUTTON2 & g_button_status_previous) == 0))
                {
                    g_led_blink_table_index++;
                    if(g_led_blink_table_index > 2)
                    {
                        g_led_blink_table_index = 0;
                    }
                }
            }
            g_button_status_previous = button_status;
        }
        /* FIXME: Since this is a temporary process,
                  so re-create a waiting process yourself. */
        /* R_BSP_SoftwareDelay(TOUCH_SCAN_INTERVAL_EXAMPLE,
                               BSP_DELAY_UNITS_MILLISECONDS); */
        g_touch_info_get_interval = 0;
    }
  }
}

/* Callback function */
void MyCallback(timer_callback_args_t *p_args)
{
    /* TODO: add your own code here */
    g_touch_info_get_interval++;
    g_led_blink_interval++;
}
```

5.3 リビルド

コーディング完了後に、プロジェクトをビルドします。

レンチアイコンをクリックしてビルドを開始します。

「コンソール」ウインドウにビルドログが出力されます。

最後に “Build Finished. 0 errors, “と表示されていればビルドが正常終了したことを意味します。

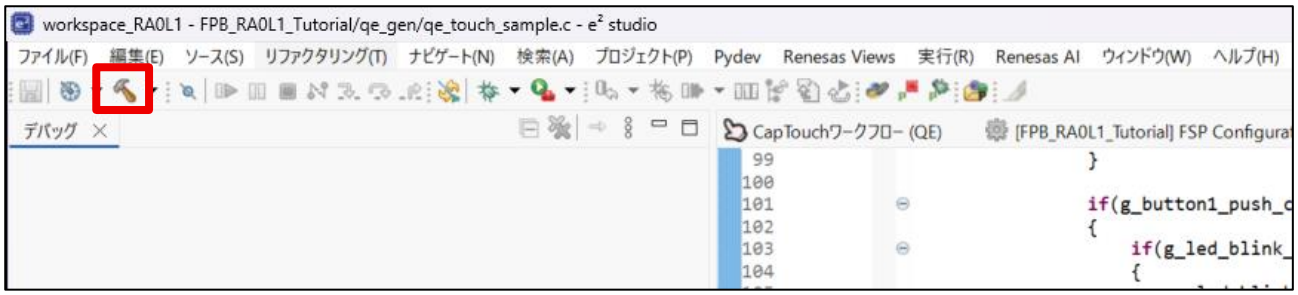


図 5-69. リビルド

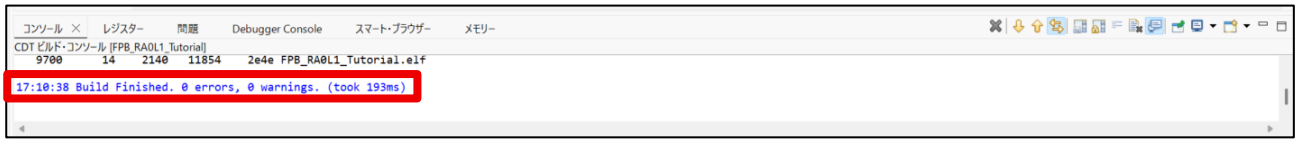


図 5-70. リビルド

5.4 実行

ビルドが完了した後にプログラムをボード上の MCU に書き込みます。

「4.3.1 デバッグ設定と起動」で書き込みの設定がされていると、赤枠の虫マークを押すとデバッグができます。

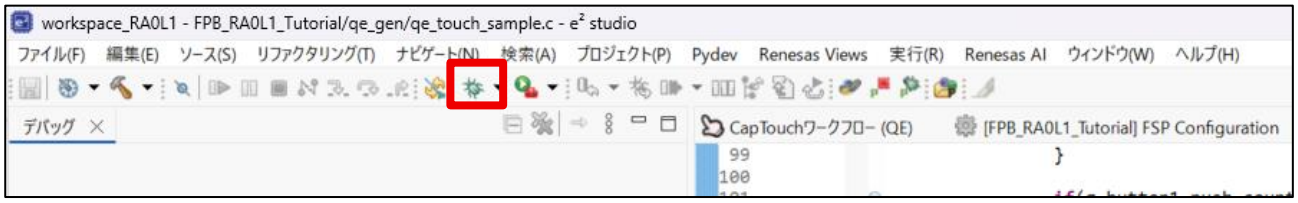


図 5-71. 実行

プログラムがダウンロードできたら SystemInit();で止まっていることを確認します。

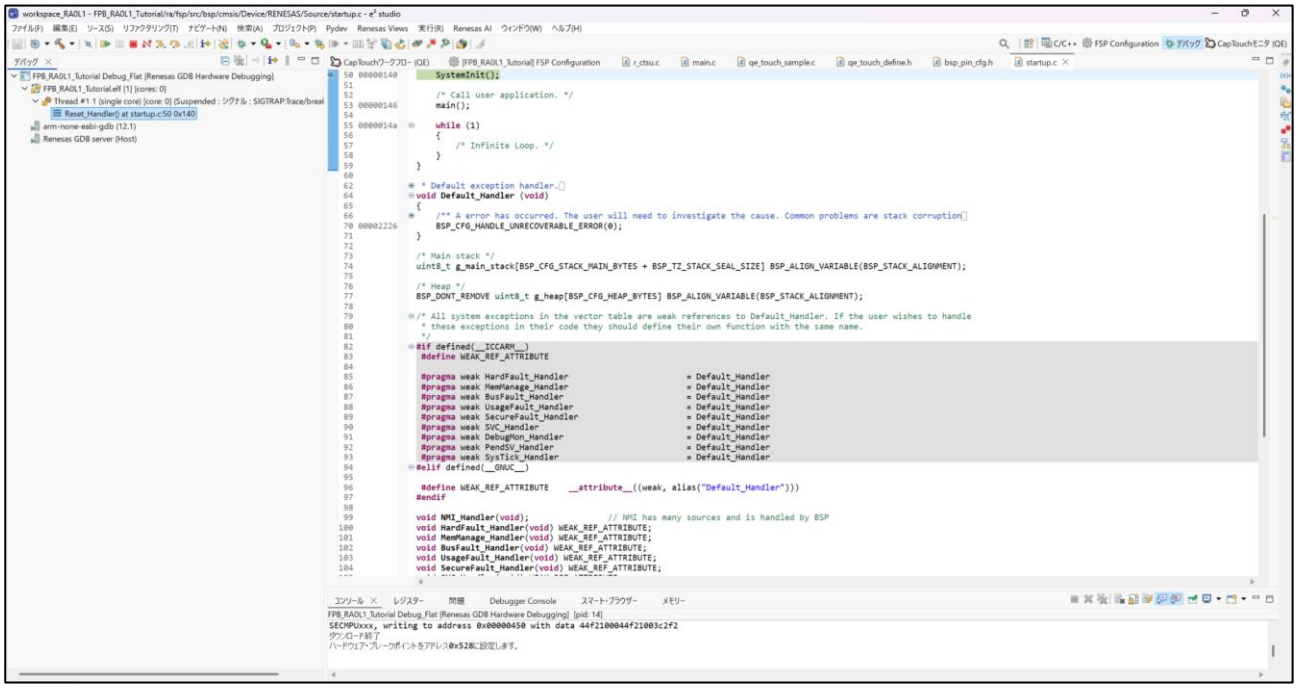


図 5-72. 実行

赤枠の再開アイコンをクリックします。

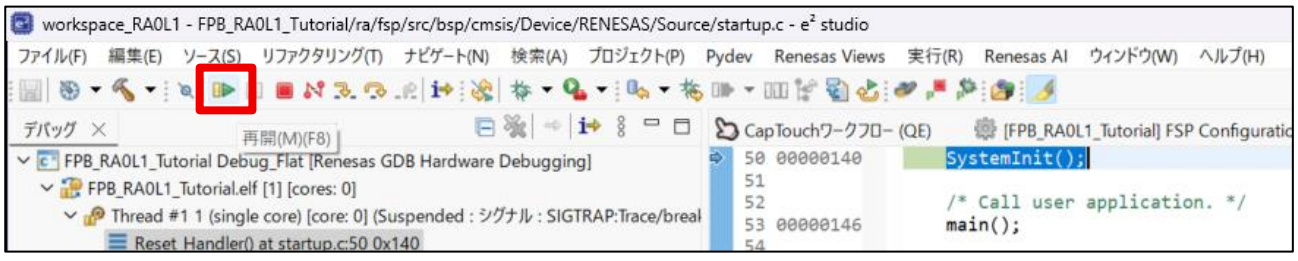


図 5-73. 実行

プログラムが main 関数の先頭で一時停止します。(e² studio プロジェクトのデフォルトが main 関数 で一時停止するよう設定されているためです。)

この状態は、SystemInit()が完了して、初期設定が完了した状態です。

再度、再開アイコンをクリックして実行します。

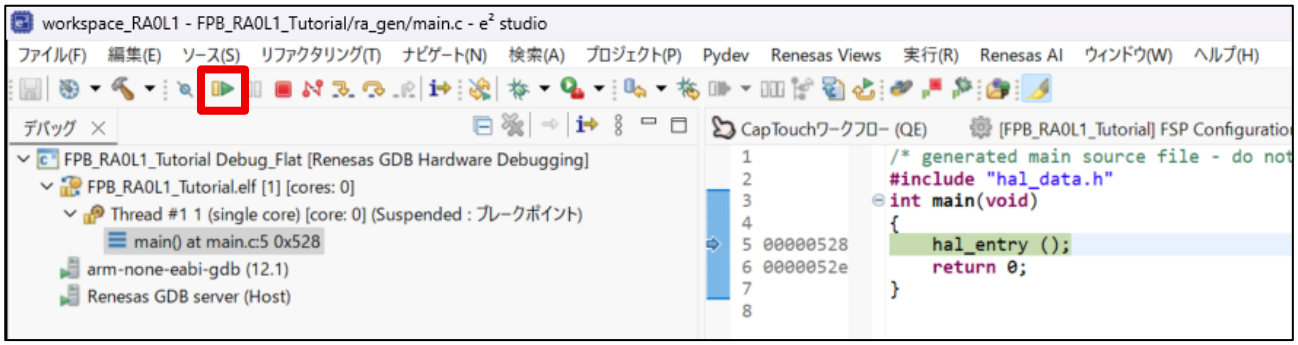


図 5-74. 実行

画面左下に実行中と表示されます。この状態ではプログラムが実行されています。実際にボード上の Touch Button をタッチ検出することで、LED5 と LED6 が点灯し、LED1 と LED2 の点滅動作が確認できます。

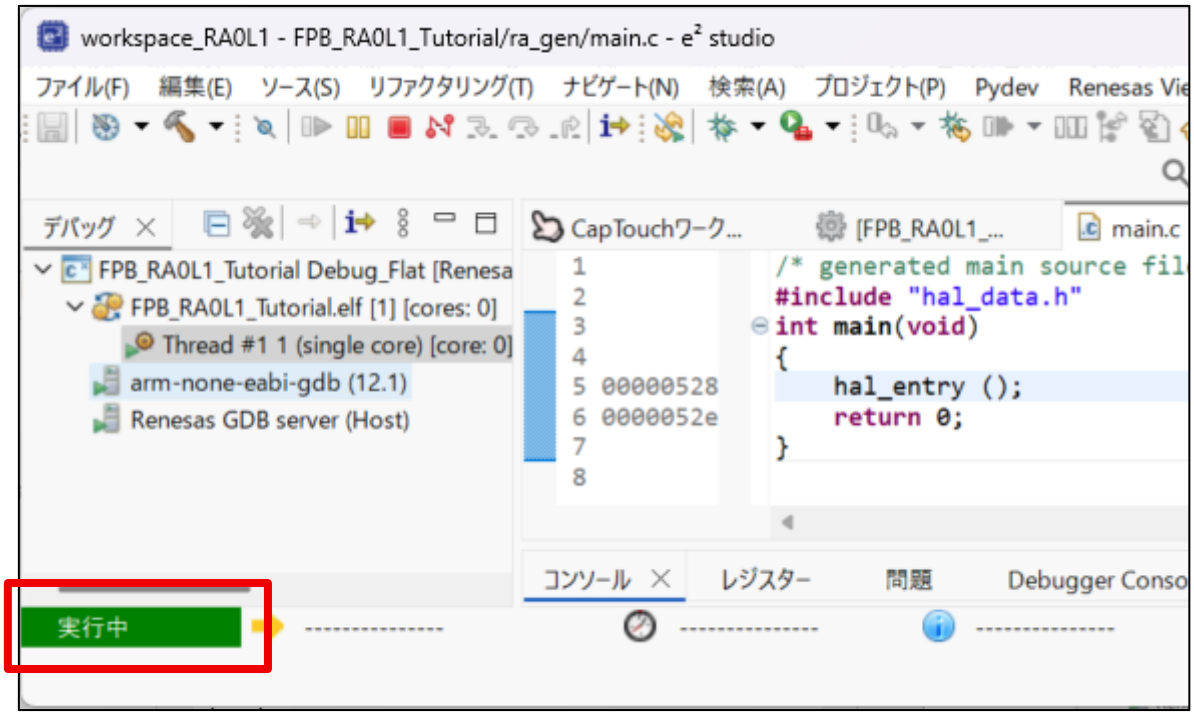


図 5-75. 実行

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2025.09.17	-	初版発行

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策
   CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。
2. 電源投入時の処置
   電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. 電源オフ時における入力信号
   当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。
4. 未使用端子の処理
   未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。
5. クロックについて
   リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
6. 入力端子の印加波形
   入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。
7. リザーブアドレス（予約領域）のアクセス禁止
   リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
8. 製品間の相違について
   型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。回路、ソフトウェアおよびこれらに関連する情報を使用する場合、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。
   標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等
   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等
   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

(Rev.5.0-1　2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。