Renesas RAファミリ

RA8 デュアルコア MCU を用いたアプリケーション開発

要旨

本アプリケーションノートでは、1 GHz動作の Cortex®-M85（CM85）コアと 250 MHz 動作の Cortex®-M33（CM33）コアを搭載した Renesas RA8 デュアルコア MCUの性能上の特長を紹介します。

デュアルコアアーキテクチャの概要に加え、2つのコア間でタスクを効果的に分担するためのユースケースを説明します。さらに、デュアルコアシステム特有の開発手法やデバッグ手順についても解説します。

本アプリケーションノートは、Helium™ を活用する CM85 コアと CM33 コアを同時に使用することで、性能を最大限に引き出す RA8デュアルコアMCU 向けアプリケーションの作成方法をまとめています。

具体的には、以下の内容について説明します。

- アプリケーションの概要・特長
- ユースケースのブロック図
- ツールの設定方法
- サンプルプロジェクトの動作確認
- 既存アプリケーションを新しい RA8デュアルコアMCU へ移行する手順

必要なリソース

本アプリケーションノートでは、以下のリソースを使用します。

開発ツールおよびソフトウェア

- e² studio：Version 2025-10（25.10.0）
- LLVM Embedded Toolchain for Arm：v18.1.3
- Renesas Flexible Software Package（FSP）：v6.2.0

ターゲットデバイス

- RA8P1
- RA8D2
- RA8M2
- RA8T2

ハードウェア

- EK-RA8P1：RA8P1 MCU グループ評価キット（http://www.renesas.com/ek-ra8p1）
- EK-RA8D2（オプション）：RA8D2 MCU グループ評価キット（http://www.renesas.com/ek-ra8d2）
- 本アプリケーションノートでは、Windows® 10 を搭載した PCを例として使用しています。対応 OS の詳細は、各開発ツールおよびソフトウェアのユーザーズマニュアルを参照してください。
- 評価キットと PC の接続には、USB Type-C ケーブルが1 本必要です。

目次

1. サンプルプロジェクトの概要 4
2. RA8デュアルコアMCU 4
3. Renesas e² studioを使用したRA8デュアルコアアプリケーションの作成 5
3.1 RA8P1デュアルコアMCU向けソリューションプロジェクトの作成 5
3.2 マルチコアプロジェクトのデバッグおよび実行 9
4. RA8デュアルコアMCUを使用したアプリケーション開発 14
4.1 システムを分割し性能を最大化 14
4.2 アプリケーションにおけるプロセッサ間通信の利用 14
4.2.1 プロセッサ間割り込みの使用 14
4.2.2 プロセッサ間通信FIFOメッセージの使用 15
4.3 RA8デュアルコアMCUでの共有メモリとリソースの使用 16
4.3.1 FSPフラットプロジェクトにおける共有メモリとリソースの使用 16
4.3.2 RTOSベースのプロジェクトにおける共有メモリとリソースの使用 18
4.4 RA8デュアルコアアプリケーションでのキャッシュとTCMの活用 18
4.4.1 Tightly Coupled Memory(TCM) 18
4.4.1 ITCMを使用した性能の向上 19
4.4.2 DTCMを使用した性能の向上 22
4.4.3 CTCMを使用した性能の向上 23
4.4.4 Cortex®-CM85コアのデータキャッシュを活用した性能の向上 24
4.4.5 ニューラルプロセッシングユニット(NPU)の使用 25
5. アプリケーションプロジェクト 27
5.1 IPC – 共有メモリプロジェクト 27
5.1.1 アプリケーションにおけるプロセッサ間通信の実装 29
5.1.2 アプリケーションにおける2コア間の共有メモリの実装 30
5.2 RTOS/IPC/共有メモリ/TCMプロジェクト 34
6. マルチコアFlat Bare-Metalプロジェクトの検証 35
6.1 プロジェクトのインポート 35
6.2 プロジェクトのビルド 36
6.2.1 CM85コア上で開発されたプロジェクトのコンパイル 36
6.2.2 CM33コア上で開発されたプロジェクトのコンパイル 37
6.3 プロジェクトのダウンロードおよび実行 38
7. FreeRTOSベースプロジェクトの検証 43
7.1 プロジェクトのインポート 43
7.2 プロジェクトのビルド 43
7.3 プロジェクトのダウンロードおよび実行 44
8. デュアルコアサンプルプロジェクトの新しいデュアルコアMCUへの移行手順 47

9. リファレンス....58
ウェブサイトおよびサポート....59
改版記録....60

1. サンプルプロジェクトの概要

本サンプルプロジェクトでは、Renesas FSP を用いてRA8デュアルコアMCU上でアプリケーションを開発するための基本的な手順を解説しています。主な対象は RA8P1 MCUですが、ここで紹介する手法は他のRA8デュアルコアMCUにも対応しています。

このプロジェクトでは、IPC（Inter-Processor Communication）モジュールを活用してCPU コア間でタスクを分担する方法や、メモリおよび各種リソースを共有することで、デュアルコアMCU の性能を最大限に引き出す方法を示しています。また、TCM（Tightly Coupled Memory）やキャッシュの活用方法についても解説しています。

2. RA8デュアルコアMCU

RA8デュアルコアMCUは、1 GHz動作の高性能 Arm® Cortex®-M85 コアと、250 MHz動作のArm® Cortex®-M33コアを組み合わせた 非対称（Asymmetric）デュアルコアアーキテクチャを採用しています。
2つのコアを同時に活用することで複数のタスクを並列に実行でき、さらに図 1に示す特長により高い処理性能を実現します。

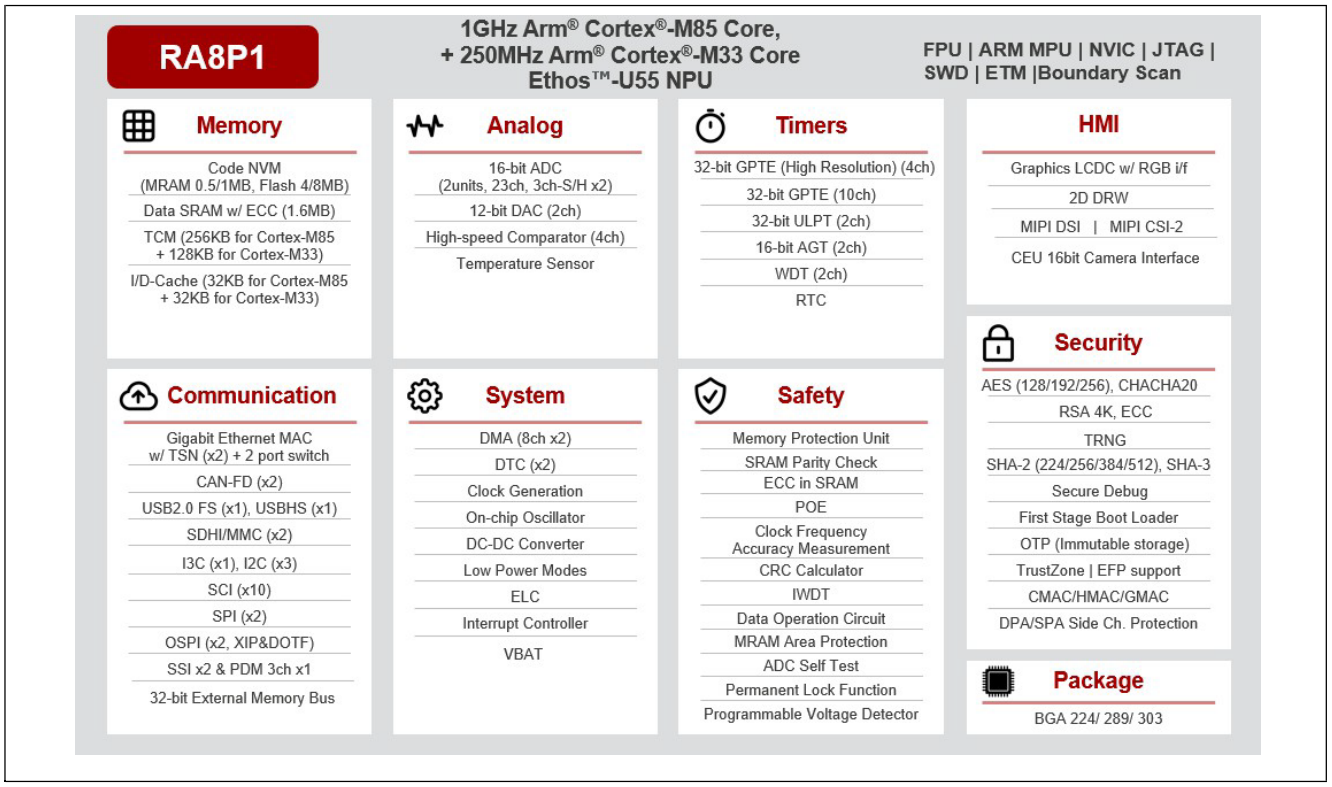


図1. RA8P1 MCUの機能

3. Renesas e² studioを使用したRA8デュアルコアアプリケーションの作成

プロジェクトを作成する際は、プロジェクトタイプの選択、プロジェクト名および保存先の指定、ならびに各種プロジェクト設定を行います。主な設定項目には、FSPのバージョン、ターゲットボード、ツールチェーンのバージョン、デバッガが含まれます（図3参照）。

本章では、**Bare Metal-Blinky**プロジェクトを例に、デュアルコアプロジェクトを作成する手順をステップごとに説明します。FSPのソリューションプロジェクトは、ソリューションプロジェクト本体に加え、CPU0用およびCPU1用の個別プロジェクトで構成されます。ソリューションプロジェクトでは、両CPU共通のメモリ構成やクロック設定を一元的に管理でき、ソリューション内の各プロジェクトに対してメモリ領域の分割や設定を行うことが可能です。

3.1 RA8P1デュアルコアMCU向けソリューションプロジェクトの作成

RA8P1 MCU向けアプリケーションでは、以下の手順に従ってe² studio上でマルチコアプロジェクトを作成します。本節では**Bare Metal-Blinky**プロジェクトテンプレートを使用しますが、プロジェクト作成時にはBare Metal-Minimalテンプレートを選択することも可能です。

e² studioを起動し、**ファイル > 新規 > Renesas C/C++ Project > Renesas RA**を選択します。
次に、**Renesas RA FSP Solution**を選択し、**次へ**をクリックします。

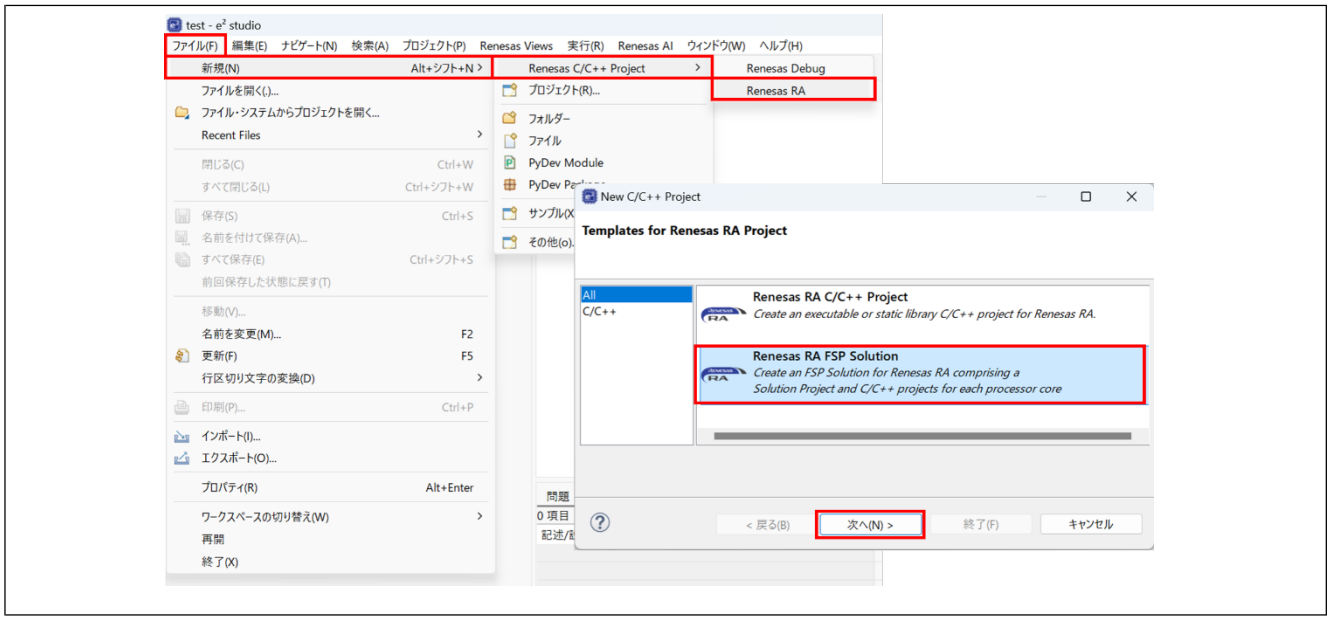


図2. RA FSP Solutionを使用したデュアルコアプロジェクト作成

新しいプロジェクトに **ek_ra8p1_blinky** などの名前を付け、**Next** をクリックします。

Device Selection 画面では、サポートされているデュアルコアRA評価キットの中からボードタイプを選択します（例：EK-RA8P1、EK-RA8D2）。

Solution Template Selection では、**Multicore > Flat > Bare Metal** の中から **Blinky** プロジェクトテンプレートを選択します。
続いて、**Toolchains** に **LLVM Embedded Toolchain for Arm**、**Debugger** に **J-Link ARM** を設定し、**終了** をクリックします。

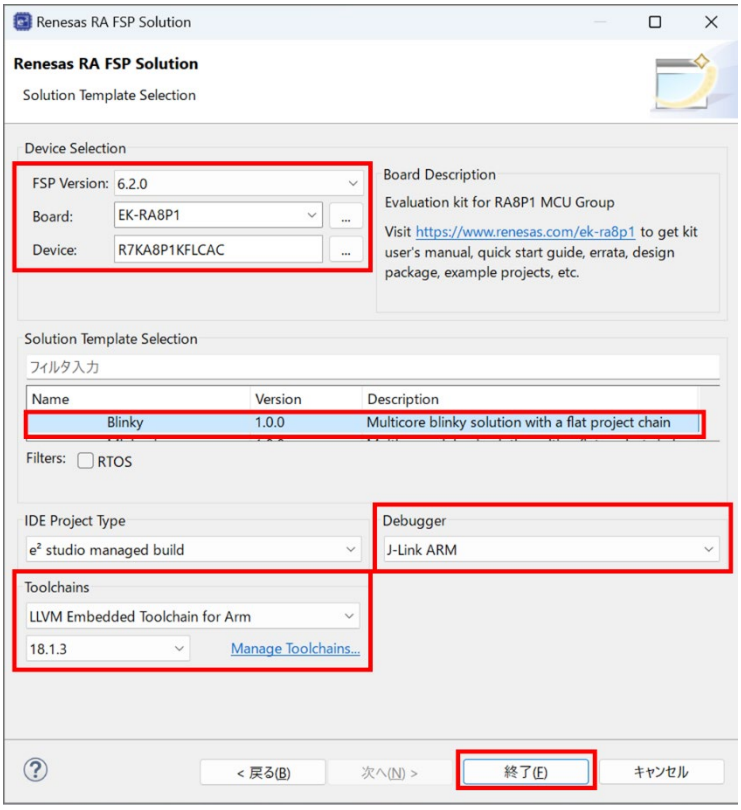


図3. プロジェクトの設定

この手順を完了すると、デュアルコアプロジェクトが正しく作成され、自動的にビルドされます。作成されるプロジェクトは、図4に示すように、3つのプロジェクトフォルダで構成されます。

- **ek_ra8p1_blinky** : CPU0およびCPU1向けのプロジェクトをまとめたソリューションプロジェクト。両コア共通のクロック設定およびメモリ設定を管理
- **ek_ra8p1_blinky_CPU0** : CPU0向けのアプリケーションプロジェクト
- **ek_ra8p1_blinky_CPU1** : CPU1向けのアプリケーションプロジェクト

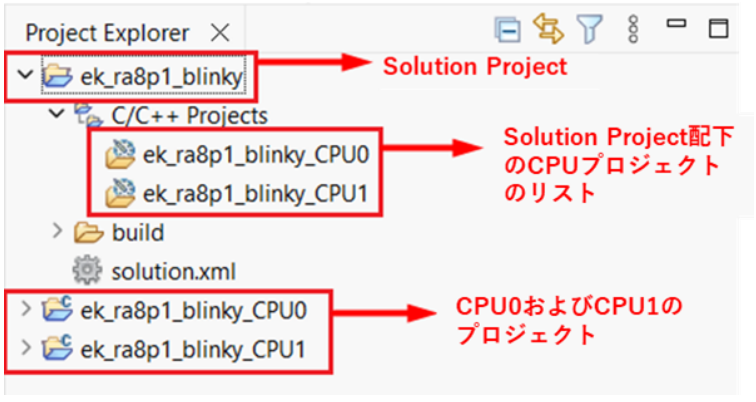
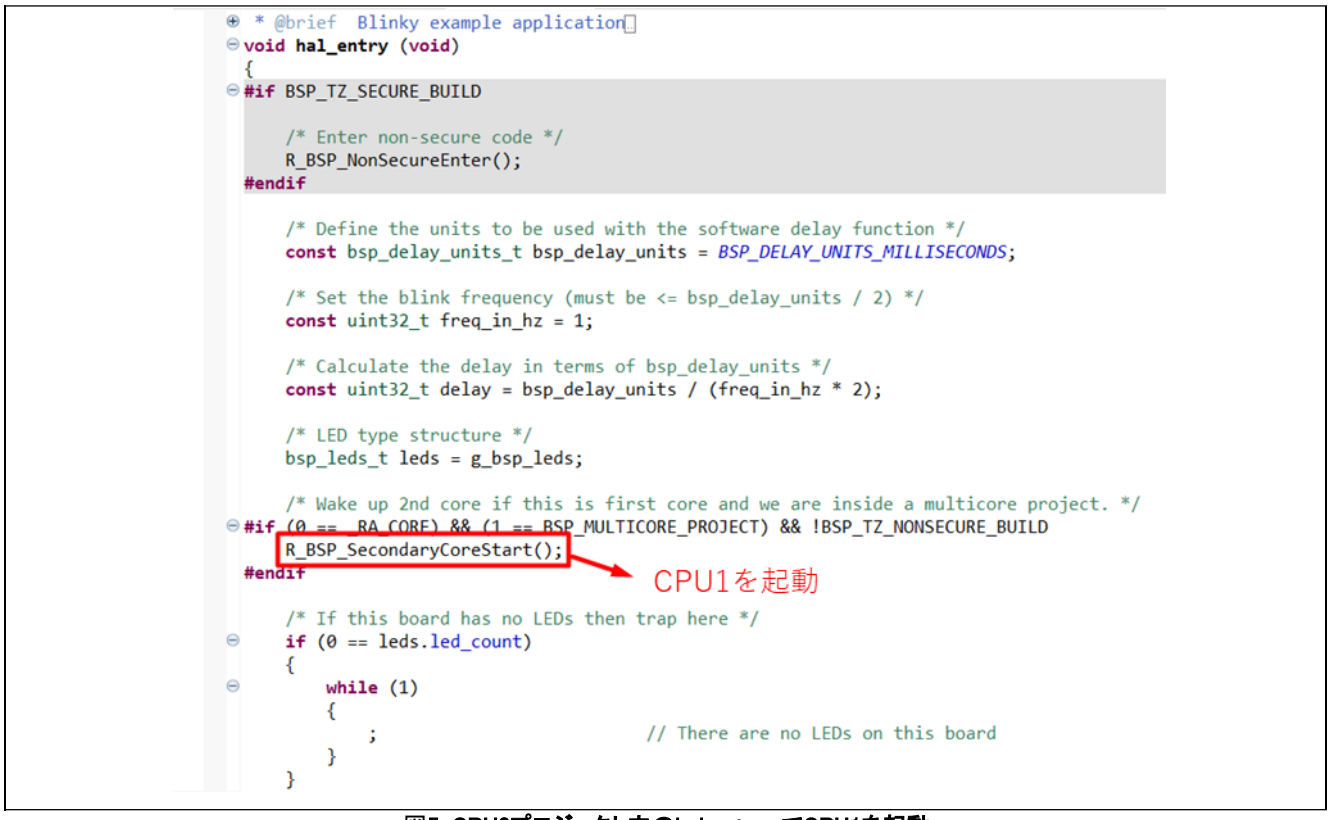


図4. デュアルコアプロジェクトの作成

CPU0のhal_entry()内でFSPのインライン関数 R_BSP_SecondaryCoreStart() を呼び出すことで、CPU1が起動します（図5参照）。




```
 * @brief  Blinky example application
void hal_entry (void)
{
#if BSP_TZ_SECURE_BUILD

    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif

    /* Define the units to be used with the software delay function */
    const bsp_delay_units_t bsp_delay_units = BSP_DELAY_UNITS_MILLISECONDS;

    /* Set the blink frequency (must be <= bsp_delay_units / 2) */
    const uint32_t freq_in_hz = 1;

    /* Calculate the delay in terms of bsp_delay_units */
    const uint32_t delay = bsp_delay_units / (freq_in_hz * 2);

    /* LED type structure */
    bsp_leds_t leds = g_bsp_leds;

    /* Wake up 2nd core if this is first core and we are inside a multicore project. */
#if (0 == BSP_CFG_RA_CORE) && (1 == BSP_MULTICORE_PROJECT) && !BSP_TZ_NONSECURE_BUILD
    R_BSP_SecondaryCoreStart();
#endif

    /* If this board has no LEDs then trap here */
    if (0 == leds.led_count)
    {
        while (1)
        {
            ;                           // There are no LEDs on this board
        }
    }
```

図5. CPU0プロジェクト内のhal_entry.cでCPU1を起動

本Blinkyプログラムでは、CPU0がLED1を点灯／消灯し、CPU1がLED2を点灯／消灯します。これにより、図6に示すように、どのコアがどのLEDを制御しているかを容易に確認できます。

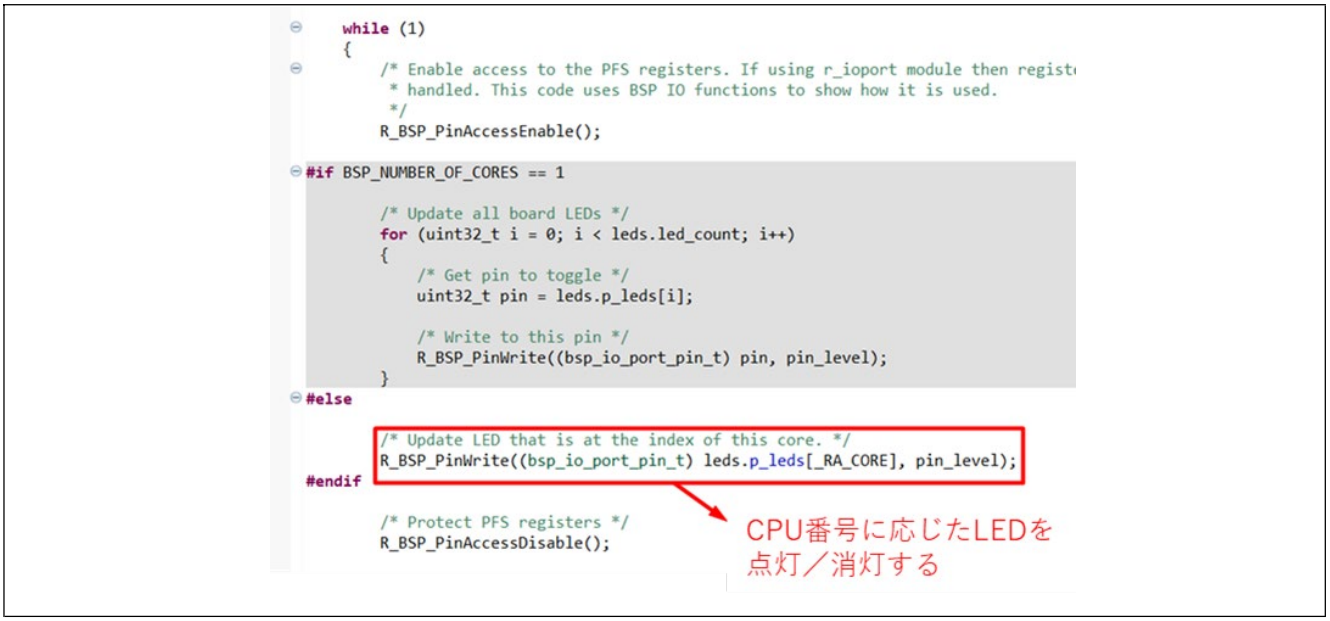


図6. Hal_entry.cにおけるBlinkyのLED制御実装

プロジェクトを再ビルドする場合は、以下の順序で実行してください。

- まずCPU0プロジェクトをビルドします。
- 次にCPU1プロジェクトをビルドします。

ソリューションプロジェクトで設定を変更した場合は、その変更をCPU0およびCPU1の両プロジェクトに反映させるため、ソリューションプロジェクトも再ビルドする必要があります。
また、デュアルコアのソリューションプロジェクトを右クリックし、**プロジェクトのビルド**を選択することで、ビルド手順を簡略化することもできます。この場合、自動ビルドは CPU0プロジェクト → CPU1プロジェクト の順で実行されます。
ソリューションベース、またはマルチコア構成のプロジェクトでは、両CPU間での設定や依存関係を一貫して管理するため、ソリューションプロジェクトからのビルドを推奨します。

図7. デュアルコアソリューションプロジェクトのビルド

CPU0およびCPU1の両方のプロジェクトが正常にビルドされ、アプリケーションイメージ（.elfファイル）がそれぞれのプロジェクトのDebug/ディレクトリに生成されていることを確認してください（例：Debug/「<プロジェクト名>.elf）。

3.2　マルチコアプロジェクトのデバッグおよび実行

デュアルコアを同時にデバッグするには、まずRenesas Flash ProgrammerまたはRenesas Device Partition Managerを使用して、MCUの初期化を行います。この初期化により、保護レベルが2に設定されていること、およびTrustZoneの境界が未設定な状態に変更します。すでにTrustZoneの境界が設定されている場合は、デバッグに適した環境を構築するため、境界をリセットする必要があります。

初期化が完了したら、デュアルコア設定を行い、デバッグを開始します。この手順を実施しない場合、プロジェクトの書き込みやデバッグが正常に行われない可能性があります。

以下に、Renesas Device Partition Managerを使用してデバイスを初期化する手順を示します。

1. 評価ボードとPCが接続されていることを確認します。
2. e² studio上で**実行 > Renesas Debug Tools**から**Renesas Device Partition Manager**を起動します。
3. **Action**欄で**Initialize device**を有効にします（チェックを入れる）。
4. **Target MCU Connection**および**Connection type**を選択します。
5. **Run**をクリックしてMCUを初期化します。

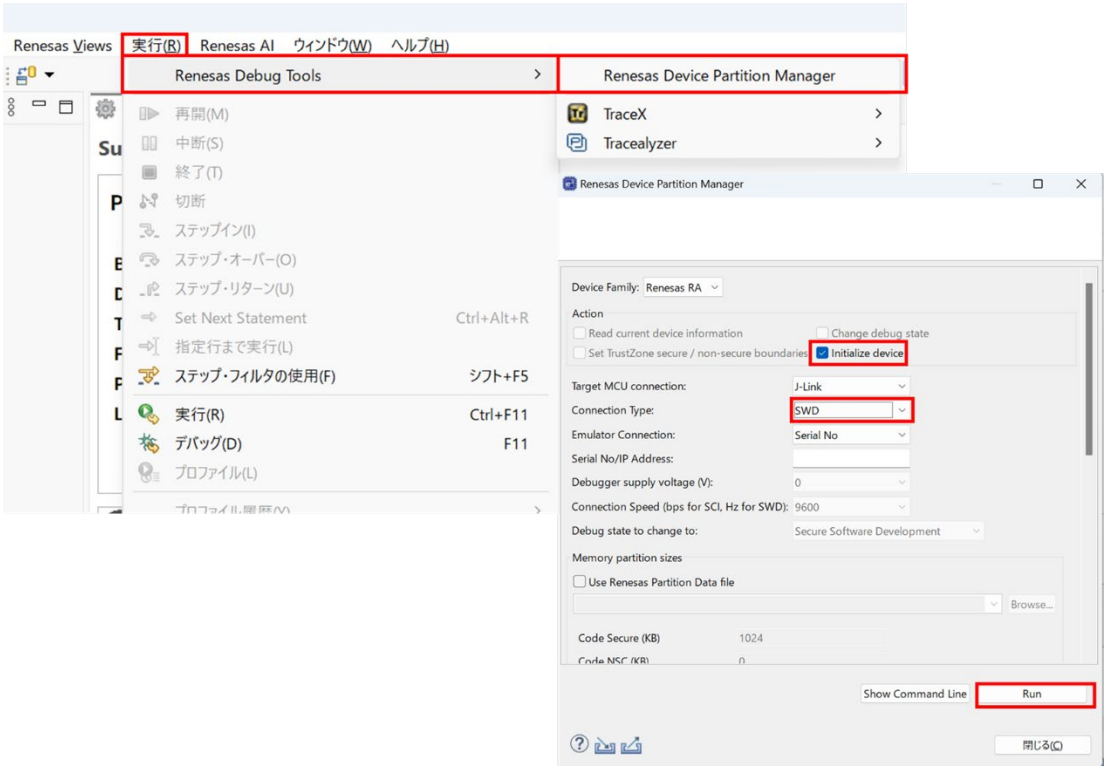


図8. RDPMによるMCUの初期化

初期化が成功すると、図9に示すメッセージが表示されます。Renesas Device Partition Manager画面をスクロールして確認することができます。

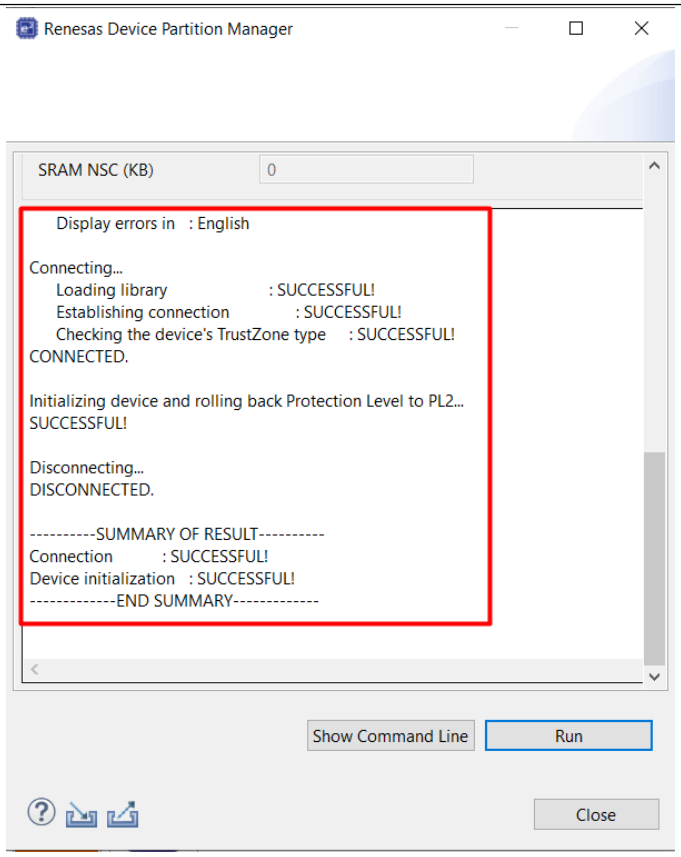


図9. RDPMでのデバイス初期化成功メッセージ

Renesas Flash Programmerを使用してデバイスを初期化するには:

1. Renesas Flash Programmerを開きます。
2. 新しいプロジェクトを作成し、ターゲットMCUへの接続を確立します。
3. **ターゲットデバイス**タブに移動します。
4. **デバイスを初期化する**をクリックして初期化を実行します。

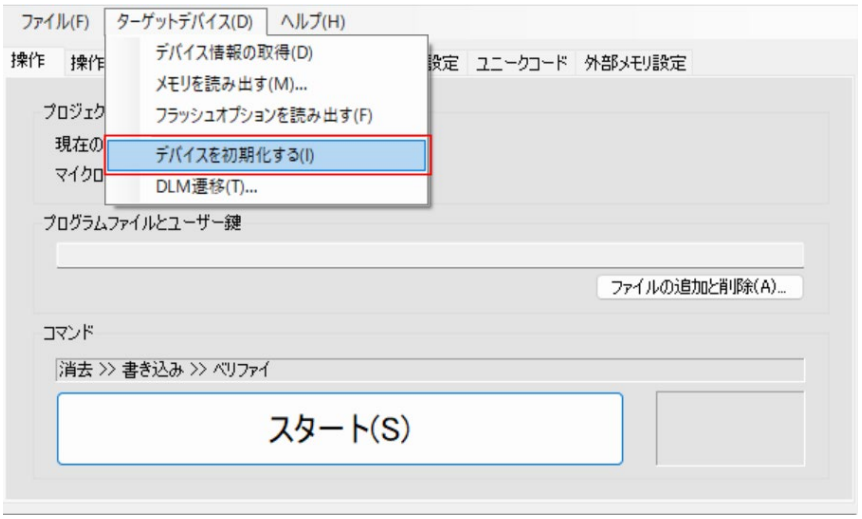


図10. RFPによるMCUの初期化

ステータスメッセージは図11のようにコンソールに表示されます。

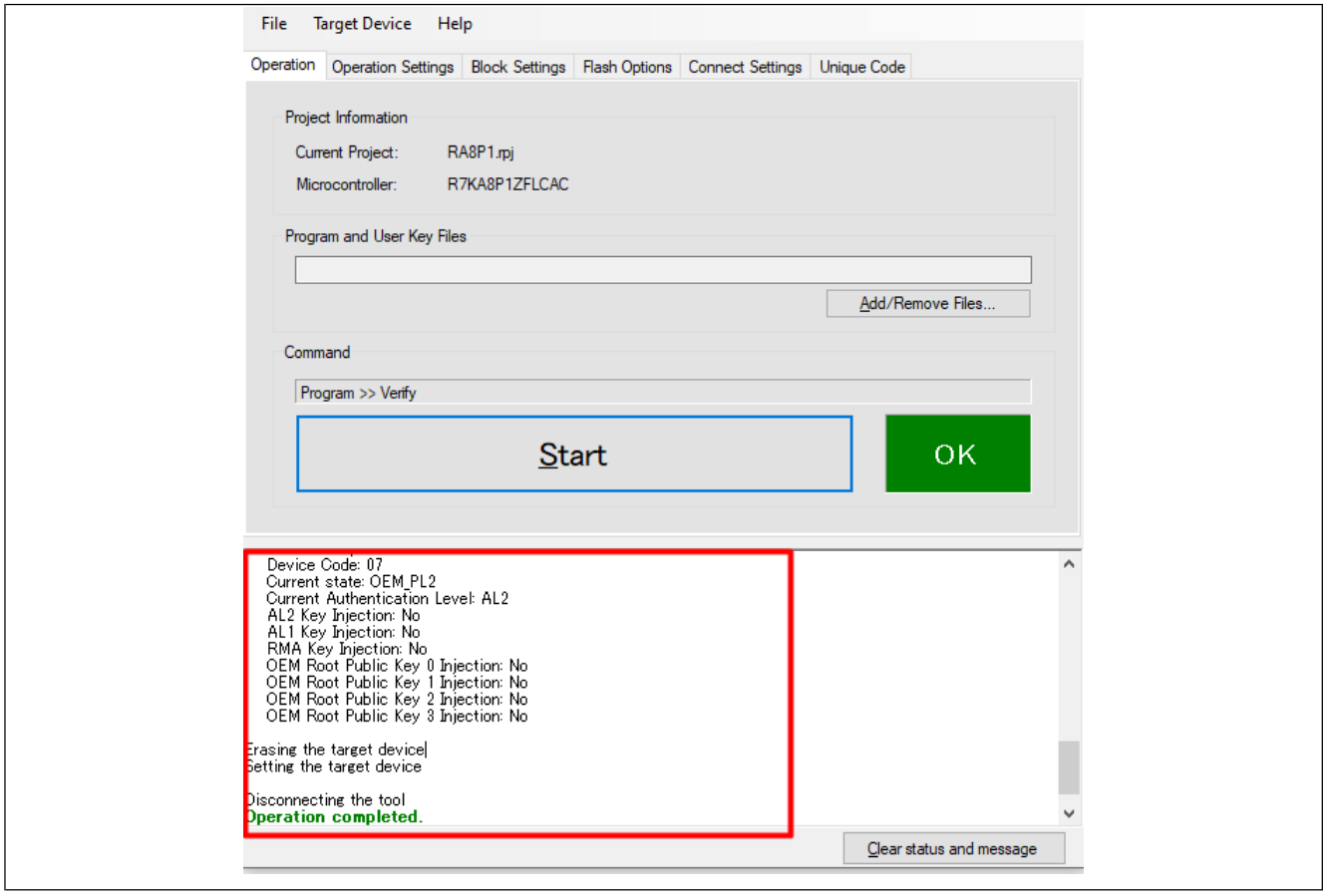


図11. RFPでのデバイス初期化成功メッセージ

RDPMまたはRFPでデバイスを初期化した後、**デバッグ構成**を開きます。**ek_ra8p1_blinky_CPU1 Debug_Multicore**を選択し、**Debugger**タブに移動して**Connection Settings**をクリックします。TrustZoneの境界設定が無効になっていることを確認してください。

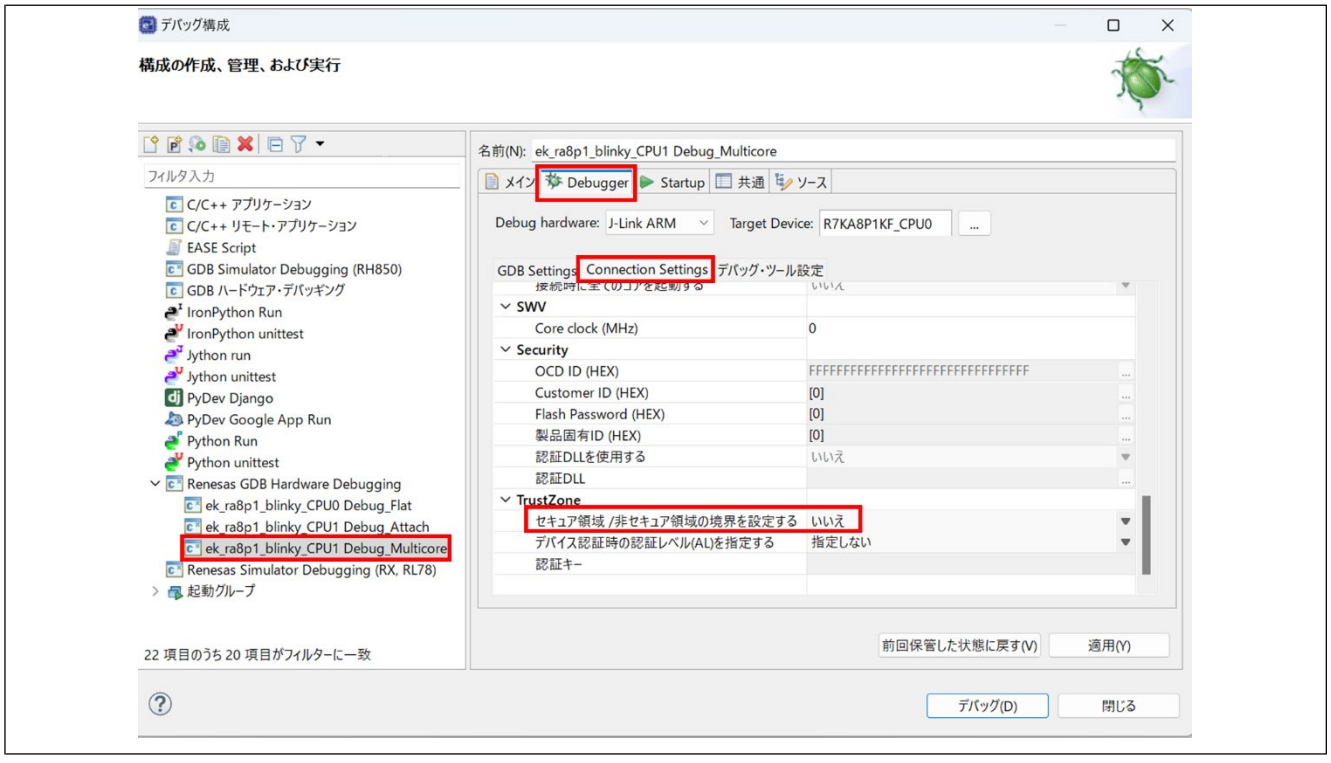


図12. TrustZoneの境界設定の無効化

e² studioには、両コアのプロジェクトを同時にデバッグできる効率的なデバッグ機能が用意されています。これは、各コアに用意された起動設定をまとめた**起動グループ**を使用することで実現できます。CPU1プロジェクトを生成すると、この起動グループは自動的に作成されます。

デバッグ構成を開き、作成された**Debug Multicore Launch Group**を選択して**デバッグ**をクリックすると、デバッグセッションが開始されます。

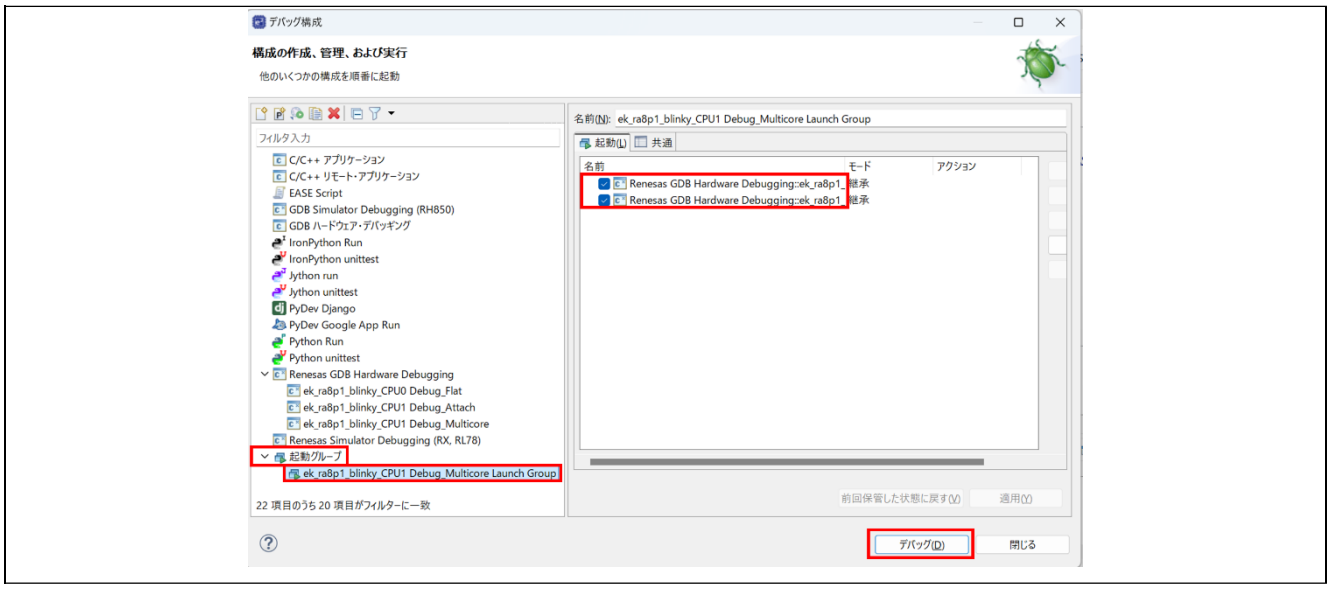


図13. Debug Multicore Launch Groupの例

デバッグタブでは、まずCPU0プロジェクトのデバッグセッションが起動・接続され、その後にCPU1プロジェクトが接続されます。デバッガはCPU0プロジェクトのReset_Handler()で停止します。

再開ボタンを2回クリックするとCPU0の動作が開始されます。main()関数を通過すると、ターゲットボード上のLED1が点滅を開始し、CPU0が正常に動作していることを確認できます。

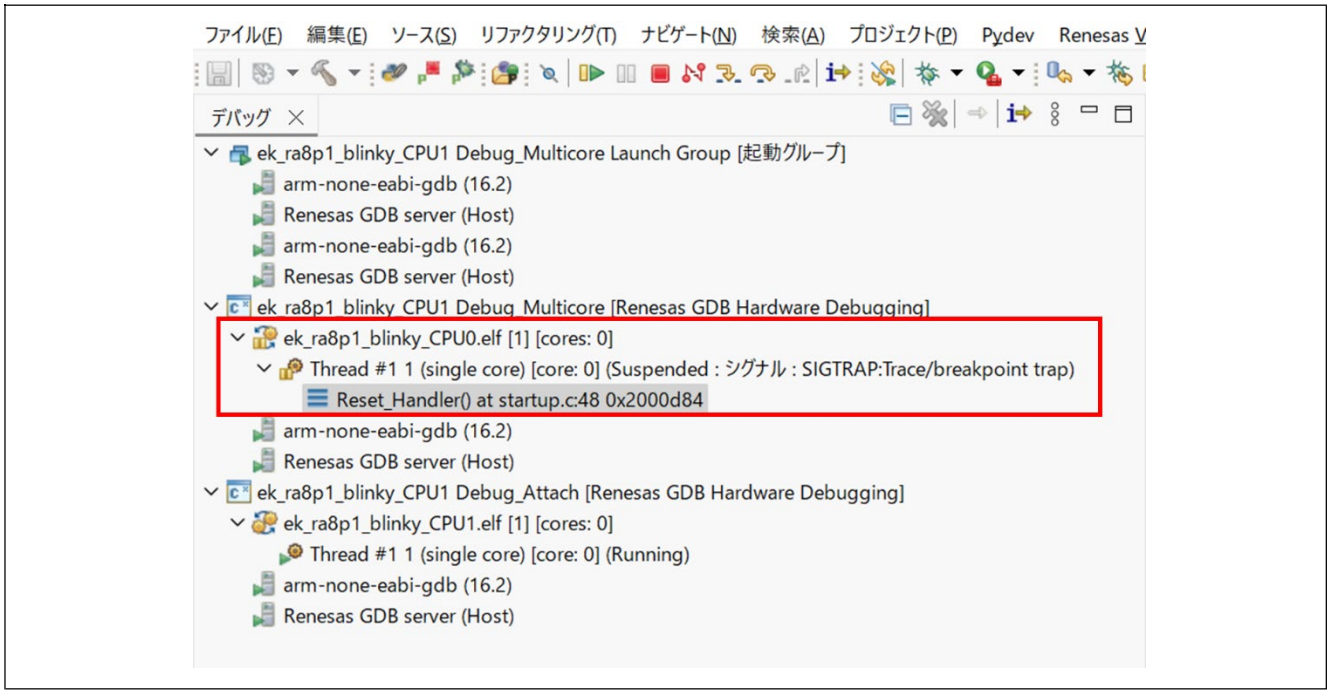


図14. デュアルコア実行時の起動直後のデバッグ状態

図15に示すように、CPU0がR_BSP_SecondaryCoreStart()を呼び出すと、CPU1のデバッグセッションが完全に有効化され、通常のデバッグ操作が可能です。

続けて再開ボタンをクリックするとCPU1の実行が開始され、LED2が点滅します。これにより、CPU1が正常に起動したことを確認できます。

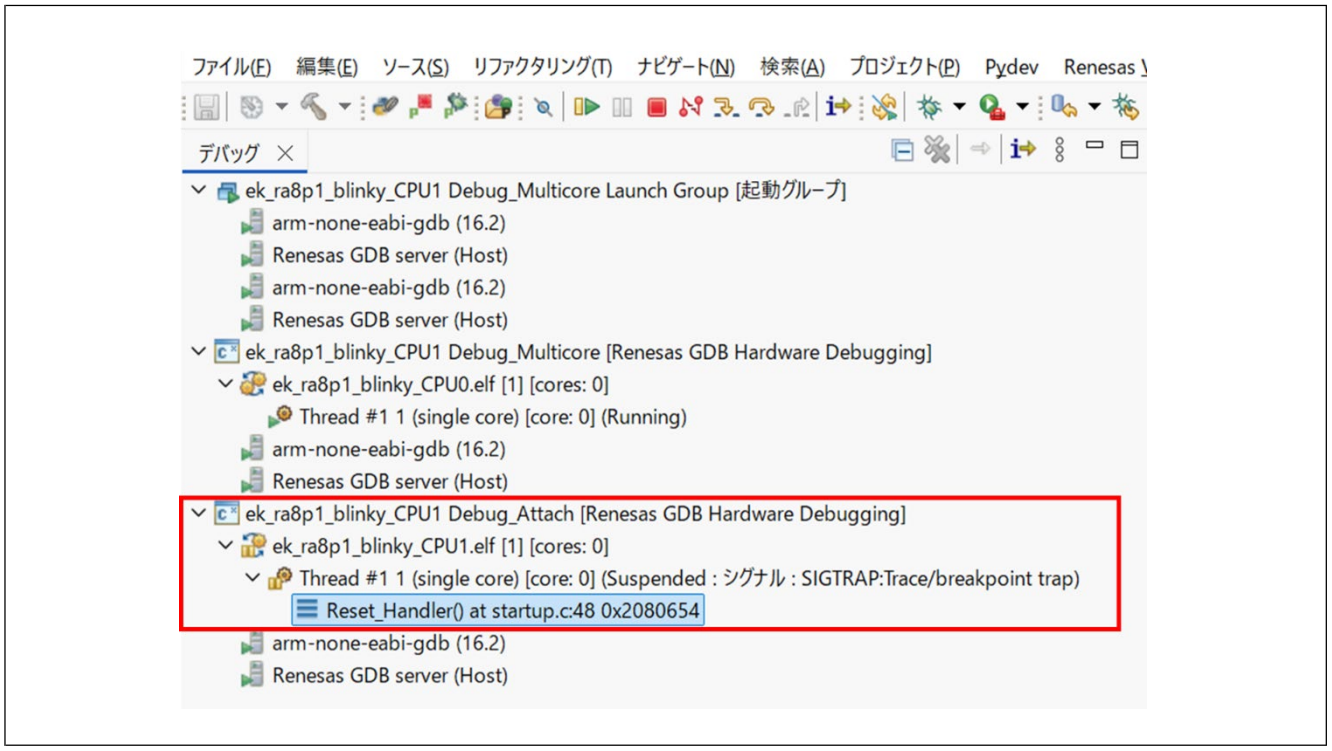


図15. R_BSP_SecondaryCoreStart()後のデバッグ状態

プロジェクトペアで最初のマルチコアデバッグを完了すると、以降はドロップダウンメニューから適切な起動グループを選択することができ、デバッグセッションを素早く開始できます。

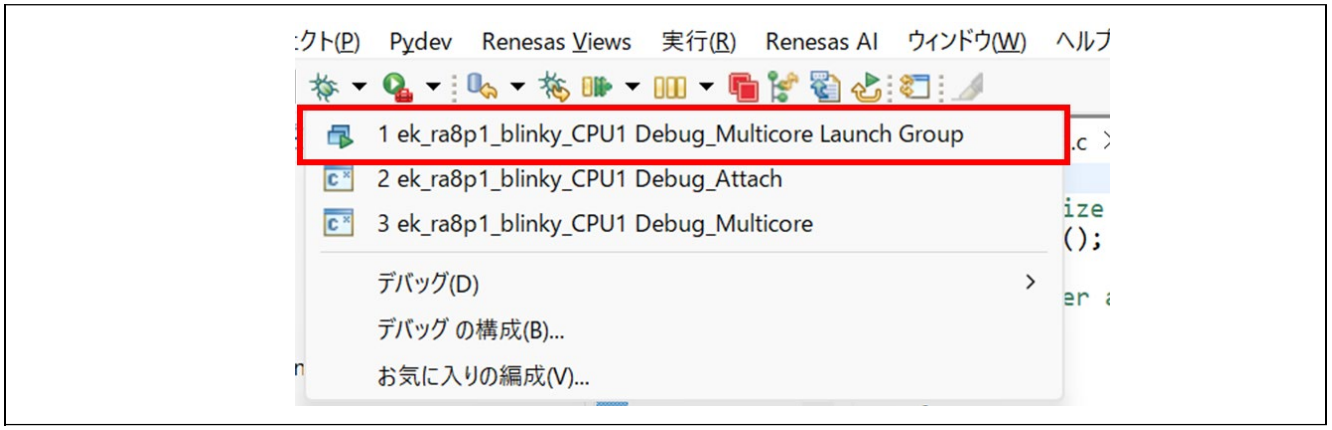


図16. デバッグドロップダウンメニューを使用したクイックアクセス

4. RA8デュアルコアMCUを使用したアプリケーション開発

RA8デュアルコアMCUを活用したアプリケーションを計画および設計する際には、性能を最大限に引き出すため、いくつかの重要なポイントを考慮する必要があります。

- アプリケーションを複数のタスクに分割し、各CPUが独立して処理できるように設計します。
- 異なるコアで動作するタスク間の通信には、IPC（Inter-Process Communication）モジュールを使用します。また、リソースの競合を防ぎ、データの整合性を保ちながら、アプリケーション全体の効率を高めるため、共有リソースは慎重に管理します。
- ITCM、DTCM、CTCM、STCMを活用することで、処理性能を向上させます。
- 命令キャッシュおよびデータキャッシュを有効に活用し、さらなる性能向上を図ります。
- グラフィックス処理、デジタル信号処理（DSP）、人工知能／機械学習（AI/ML）などの演算負荷の高いタスクはCM85コアに割り当てます。CM85コアは高い動作周波数に加え、M-Profile Vector Extension（MVE）を活用できます。一方、センサ入力などのデータ取得やUART入出力といった比較的軽量なタスクはCM33コアで処理します。

4.1 システムを分割し性能を最大化

デュアルコアアーキテクチャを活かすため、リアルタイム制御、AI/ML、グラフィックス処理などのタスクをCPUコア間で分担します。例えば、一般的なグラフィックスアプリケーションでは、次のように2つのCPUに役割を割り当てることができます。

- **CPU0**：グラフィックスモジュール、JPEGデコーダ、SDRAMアクセスを管理
- **CPU1**：データ取得、タッチコントローラ、出力制御などの入出力処理やユーザインタフェースを担当

このように役割を分担することで、システムバスや共有メモリにおけるリソース競合を低減でき、各CPUが互いに干渉することなく効率的に動作します。また、リアルタイム制御ループ、センサフュージョン、AI/ML推論といった追加の処理を、性能要件に応じていずれかのコアに割り当てるための拡張性の高い基盤を構築できます。

明確に定義されたタスク分割と、IPC割り込み、共有メモリ、メッセージキューなどのプロセッサ間通信機構を組み合わせることで、デュアルコアシステムは高い効率を維持しつつ、優先度の高いタスクが予測どおりに実行されることを保証します。

4.2 アプリケーションにおけるプロセッサ間通信の利用

プロセッサ間通信（IPC）を使用することで、MCU内の2つのプロセッサ間でハードウェアリソースの共有やデータのやり取りが可能です。また、IPCは割り込みイベントを生成することで、プロセッサ間の同期や動作の調整もサポートします。

本節では、IPCの主な要素として以下を取り上げます。

- データのやり取り
- タスク同期
- 効率的なリソース共有

4.2.1 プロセッサ間割り込みの使用

プロセッサ間割り込みは、共有データやシステムイベントに直ちに対応が必要な場合に、もう一方のCPUへ通知するための低レイテンシな手段です。RAデュアルコアアーキテクチャでは、マスカブルおよびノンマスカブルの両方のプロセッサ間割り込みをサポートしており、イベントの重要度に応じた柔軟な優先度制御が可能です。

図17に示す一般的な処理例では、一方のコアが周辺回路の動作、バッファの空き状況、または処理完了などのイベントを監視し、その後、もう一方のコアに対してプロセッサ間割り込みを発行します。この割り込みは軽量な通知メカニズムとして機能し、イベント発生時に受信側CPUが即座に対応できるようにします。

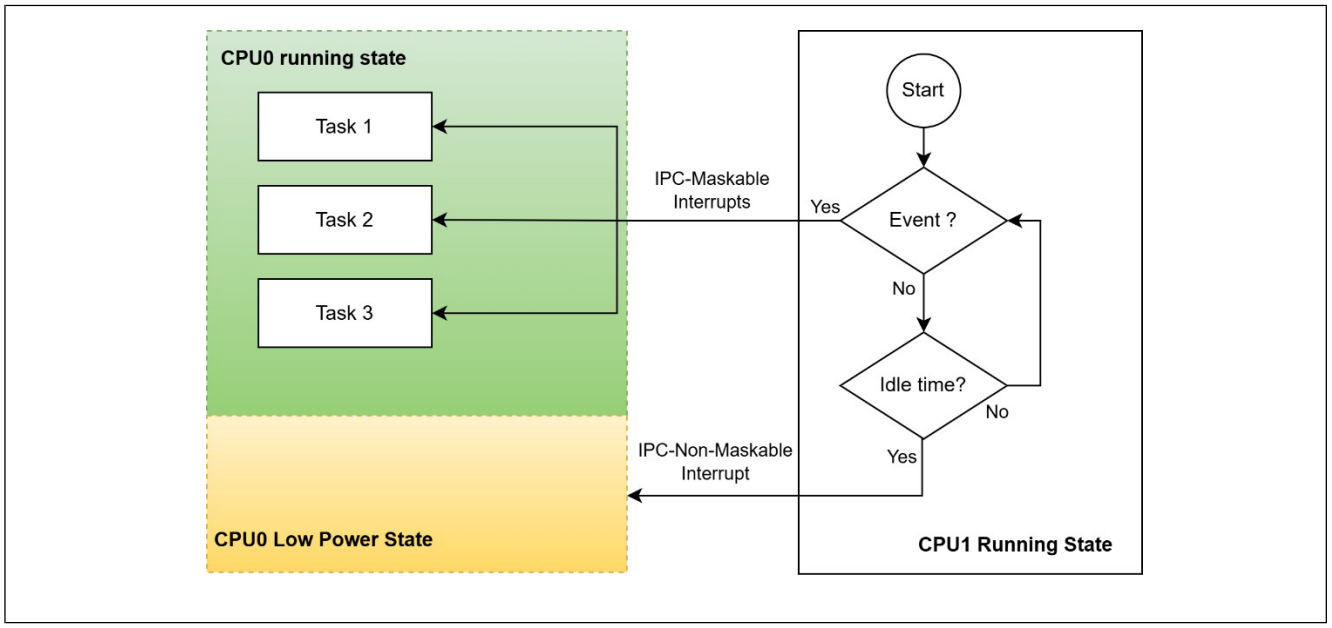


図17.IPC割り込みを使用したCPU0通知の例

共有メモリやステータスフラグを扱う際の競合状態（レースコンディション）を防ぐため、各割り込み通知は通常、明確に定義された通信プロトコルと組み合わせて使用されます。これには、専用の共有メモリ構造体への書き込み、状態フラグの更新、またはIPCメッセージキューを介したデータ送信などが含まれます。

通知を受け取ったCPUは、更新された情報を処理し、必要なタスクを実行します。また、双方向の同期が必要な場合には、応答用の割り込みを発行することもできます。これらの考え方については4.3節で詳しく説明しており、共有メモリへのアクセスにおける排他制御のフローは図20に示しています。

プロセッサ間割り込みは、セマフォ、メッセージキュー、共有メモリバッファと組み合わせることで、リアルタイム性やグラフィックス処理を重視するデュアルコアシステムにおける、信頼性の高いプロセッサ間連携の基盤となります。これにより、確定的なウェイクアップを実現し、ソフトウェアポーリングやタイマベースの通知ループと比べて、レイテンシを低減し、システムバスの競合を抑制できます。

アプリケーションにおけるプロセッサ間割り込みの実装方法については、5.1.1節を参照してください。

4.2.2　プロセッサ間通信FIFOメッセージの使用

IPCモジュールは、2つのCPUコア間で効率的かつ確定的にメッセージをやり取りするための、4つのハードウェアFIFOを提供します。各FIFOは単方向であり、メッセージの種類や優先度に応じて、個別の通信チャネルを構成できます。

- **IPC00およびIPC01：CPU1 → CPU0へのメッセージ送信用FIFO**
- **IPC10およびIPC11：CPU0 → CPU1へのメッセージ送信用FIFO**

各FIFOは4段構成で、32ビット幅のデータをサポートしています。これにより、共有メモリや複雑な同期処理を用いることなく、小規模なコマンド構造体、フラグ、メッセージトークンを高速に転送できます。FIFOはハードウェアで実装されているため、ソフトウェアによる調停を必要とせず、低レイテンシでデータ転送が行えます。

この仕組みは、以下の用途に適しています。

- コマンドやイベントの通知
- 軽量な同期処理
- 小さなデータトークンの送信
- もう一方のCPUでの状態遷移やタスク実行のトリガ

図18は、IPCメッセージFIFOを用いたデータのやり取りの仕組みを示しています。

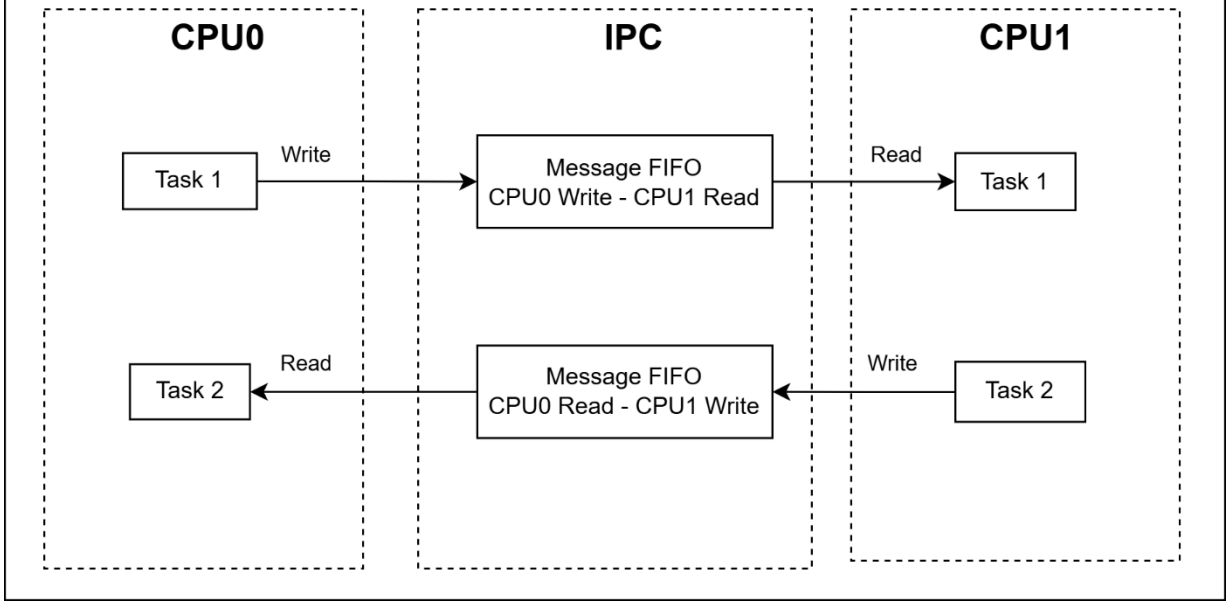


図18.IPC-Messages FIFOを使用したデータのやり取り

より高度な共有メモリ通信や、より大きなデータのやり取りが必要な場合は、5.1節を参照してください。同節では、IPC FIFO、共有メモリ構造体、RTOSキューを組み合わせたプロセッサ間データ交換の方法について説明しています。

4.3　RA8デュアルコアMCUでの共有メモリとリソースの使用

共有メモリは、デュアルコアシステムにおいて大量のデータを2つのコア間でやり取りするための、最も効率的な手法の一つです。この方式では、2つのコアが共通のメモリ領域に直接アクセスしてデータの読み書きを行いますが、競合状態やデータ破壊を防ぐための並行制御が不可欠です。

4.3.1　FSPフラットプロジェクトにおける共有メモリとリソースの使用

2つのコア間で、高速かつ大量のデータ（例：ビデオストリーミング）をやり取りする必要がある場合、共有メモリは他のIPC手法と比べて高速であるため、最適な選択肢となります。ただし、競合状態を回避するためには、適切な同期メカニズムを組み合わせて使用する必要があります。

図19は、共有メモリ領域への同期アクセスを実現するための、データ交換および通知方法の一例を示しています。

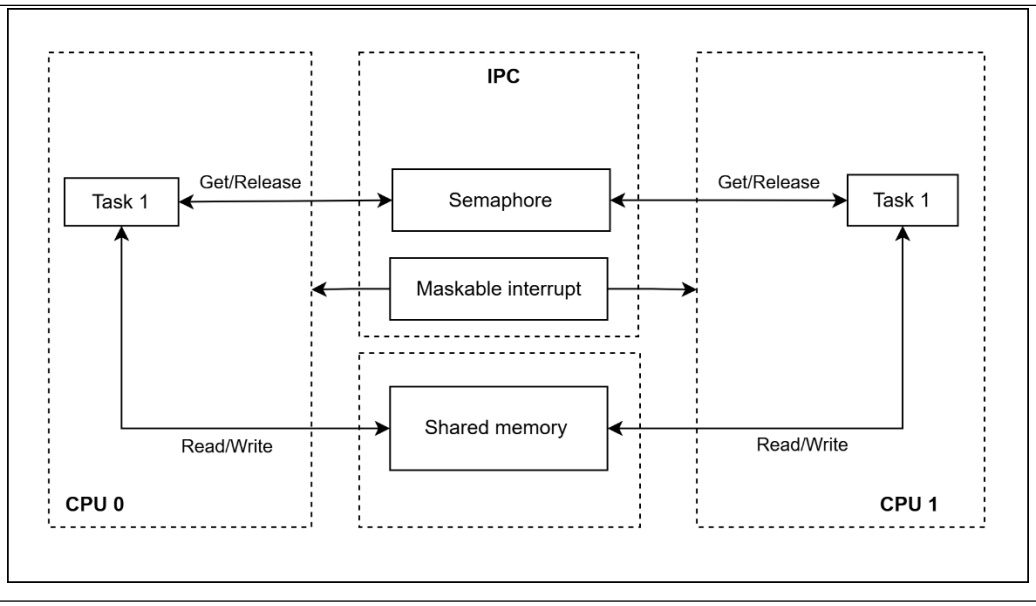


図19.共有メモリデータ交換と同期

デュアルコアシステムで共有メモリを実装する際には、いくつかの重要な点を考慮する必要があります。

- メモリ割り当て：両方のコアからアクセス可能な専用のメモリ領域を確保し、共有メモリとして設定します。
- 同期メカニズム：データ破損を防ぎ、正しいアクセス順序を保証するために、ミューテックス、セマフォ、またはハードウェアフラグなどの適切な同期方式が必要です。
- データ交換：。一方のコアが共有メモリ領域にデータを書き込み、もう一方のコアが必要に応じて読み出すことで、コア間の連携した通信を実現します。

図20は、両方のCPUがセマフォを実行し、プロセッサ間マスカブル割り込みを通じてコア通知を送信する際の排他制御フローを示しています。

アプリケーションで共有メモリを実装する手順については、5.1を参照してください。

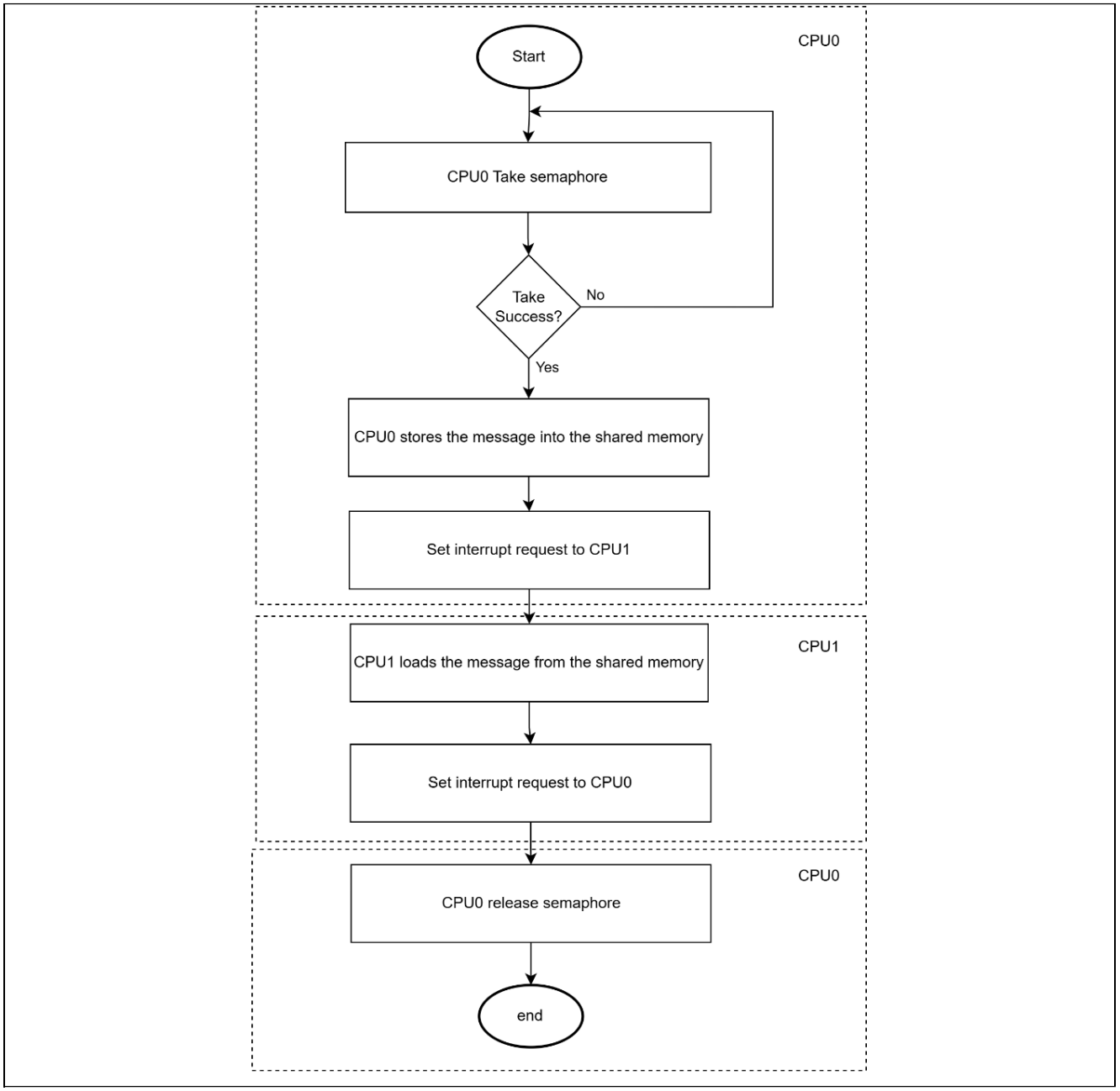


図20.アプリケーション排他制御フロー

デュアルコアシステムでは、一部の処理やサービスをどちらか一方のコアに集約して割り当てることで、性能の最適化やリソースの重複を防ぐことができます。この方式により、各コアは干渉を受けることなく割り当てられた処理に集中でき、システム全体の効率が向上します。

また、この手法はコードサイズの最適化やリソース共有に伴う課題の軽減にも有効です。タスクがこれらのサービスを必要とする場合には、プロセッサ間通信チャネルを介してリクエストを送信することで、必要な処理を依頼できます。
一部のサービスは2つのコアで共有されますが、実際の処理は一方のコアで集中的に行われます。

代表的な例として、周辺デバイス管理（UART、SPI、I2C）、ファイルシステム管理、ネットワークスタック管理などがあります。
図21は、UART周辺機能をCPU1が管理し、CPU0がIPC機構を通じてUARTサービスにアクセスする構成例を示しています。
このアーキテクチャは、付属のサンプルアプリケーションでも実際に確認できます。

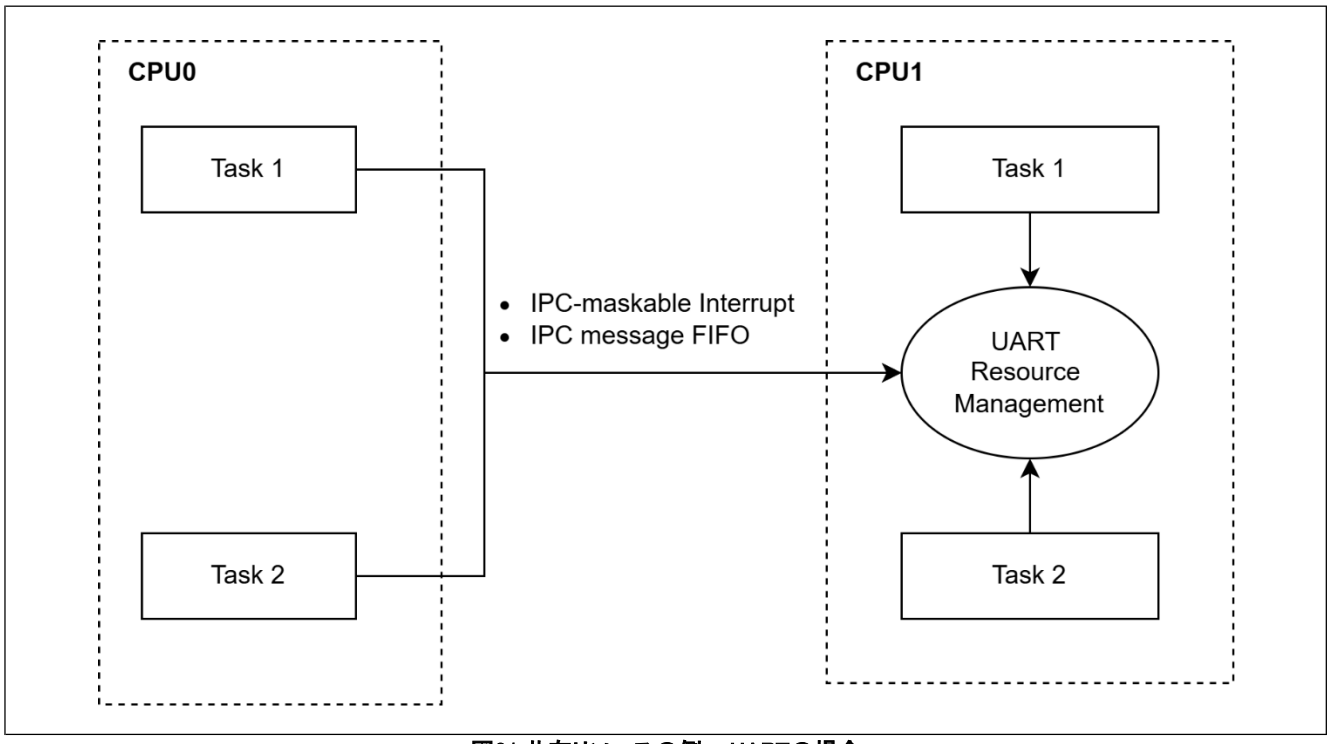


図21.共有リソースの例 – UARTの場合

4.3.2 RTOSベースのプロジェクトにおける共有メモリとリソースの使用

FreeRTOSを用いたマルチコアアプリケーションにおいても、共有メモリや共有リソースの考え方自体はBare-Metalの場合と同じです。両CPUが同じメモリ領域や周辺機能にアクセスする場合は、競合状態やバスの取り合いを防ぐための対策が必要です。

一方で、FreeRTOSには、セマフォ、ミューテックス、キュー、ストリームバッファ、メッセージバッファといった、マルチタスク・マルチコア開発を支援する同期メカニズムが用意されています。これらを活用することで、複雑な処理でも比較的シンプルに実装できます。

本プロジェクトでは、Renesas FSPのIPCモジュールをベースにした軽量なメッセージキューラッパーを使用して、2つのコア間でデータをやり取りしています。アプリケーションの要件に応じて、ストリームバッファやメッセージバッファなど、他のFreeRTOSプリミティブを選択することも可能です。

FreeRTOS環境における共有メモリやIPCの具体的な実装例については、5.2節を参照してください。

4.4 RA8デュアルコアアプリケーションでのキャッシュとTCMの活用

高い性能を引き出すためには、TCM（Tightly Coupled Memory）とキャッシュを、Helium™技術と組み合わせて活用します。
TCMは、1サイクルでの確定的なアクセスが可能なため、時間制約の厳しい処理に適しています。
処理負荷の高いコードや、頻繁にアクセスされるデータをTCMに配置することで、高速かつ予測可能な実行が可能です。

4.4.1 Tightly Coupled Memory (TCM)

RA8デュアルコアMCUでは、**CPUごとに専用のTCMが用意**されています。

- CPU0 ： 命令用TCM（ITCM）、データ用TCM（DTCM）
- CPU1 ： C-AHB TCM（CTCM）、S-AHB TCM（STCM）

CPU0には合計256KBのTCMが搭載されており、その内訳は128KBのITCMと128KBのDTCMです。

CPU1には合計128KBのTCMが搭載されており、64KBのCTCMと64KBのSTCMで構成されています。

※注：TCMは、CPUディープスリープモード、ソフトウェアスタンバイモード、ディープソフトウェアスタンバイモード

では使用できません。

図22は、MCUローカルサブシステムにおけるTCMの配置を示しています。

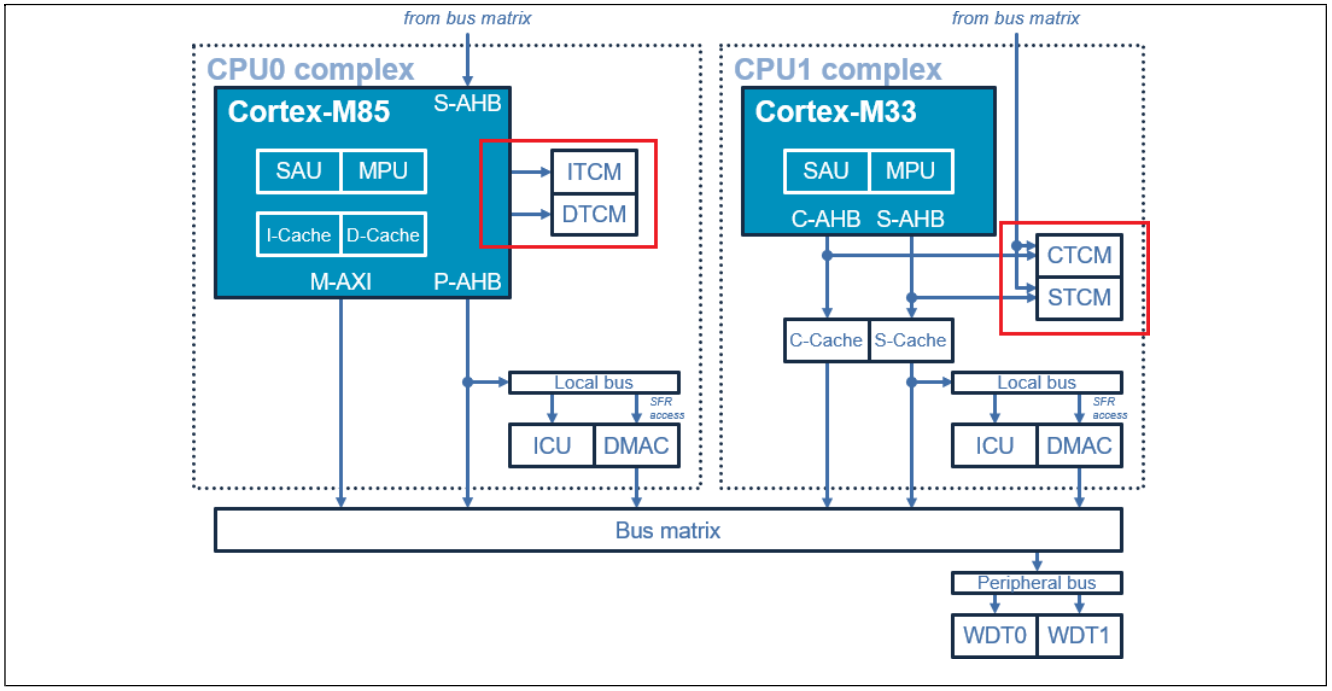


図22.ローカルMCUサブシステムにおけるTCMメモリ

4.4.1　ITCMを使用した性能の向上

時間制約の厳しい処理において性能を高めるには、特定の関数の命令コードをITCM（Instruction Tightly Coupled Memory）に配置します。この設定は、e² studioのC/C++プロジェクト用FSPコンフィギュレータにある**Linker Sections**の設定から行います。

以下では、本サンプルプロジェクトの一つであるRA8P1_DSP_exampleプロジェクトを例に、arm_cmplx_mag_f32関数をITCMに割り当てる方法を示します。

1. e² studioでプロジェクトを開きます。
2. Project Explorerで**configuration.xml**をダブルクリックします。
3. **Linker Sections**タブを開きます（図23参照）。
4. **arm_cmplx_mag_f32**関数を、指定されたITCMセクションに割り当てます（図24～図26参照）。

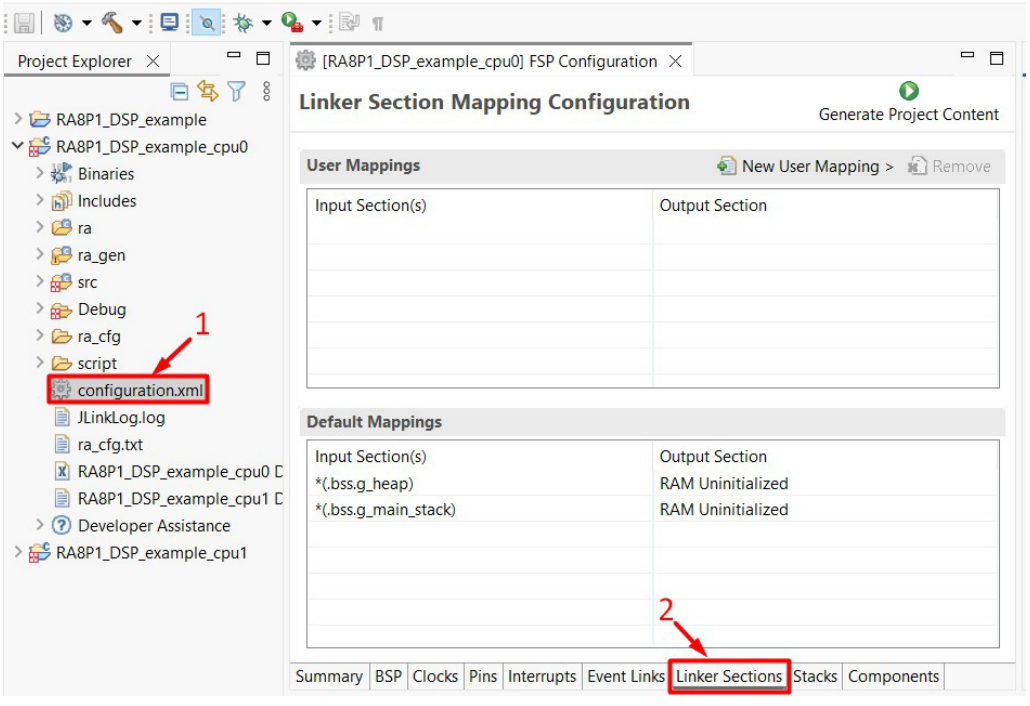


図23. Linker Sectionsを開く

New User Mapping > ITCM > Code initialized from > ITCM code from FLASH をクリックします。

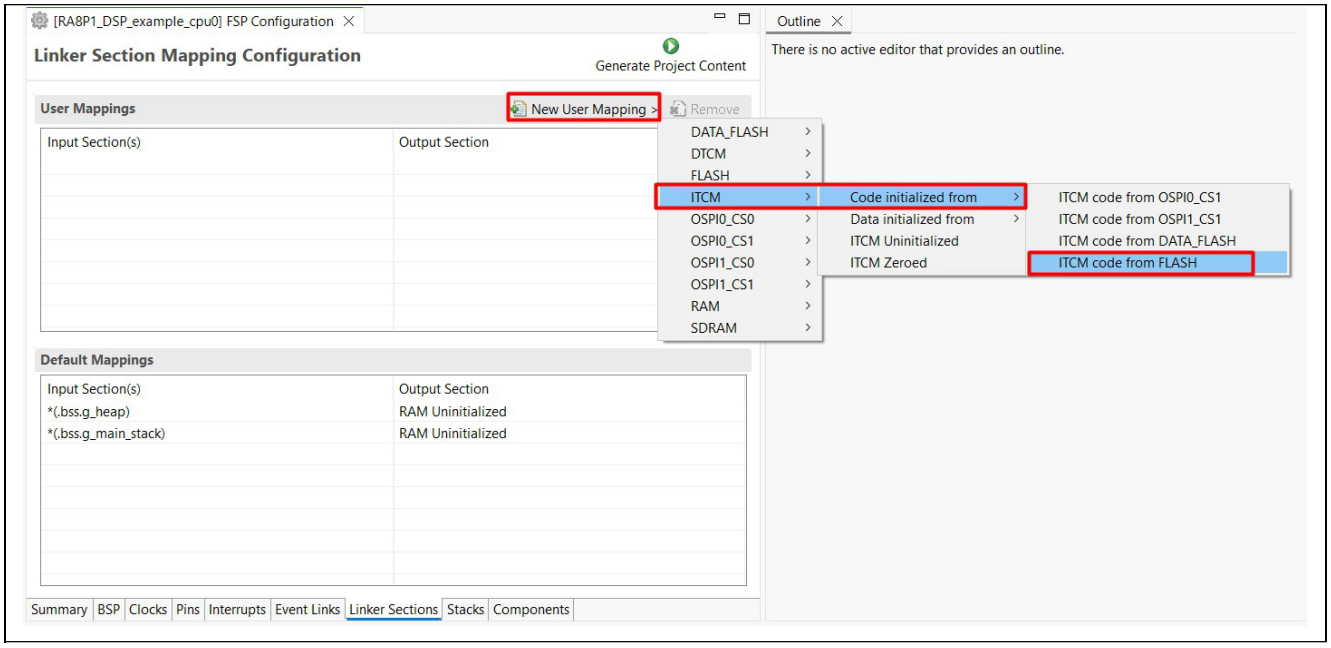


図24.ITCM向けの新しいセクションマッピングの定義

新しいユーザマッピングの設定ウィンドウが表示されます。入力するセクション名は特定のフォーマットに準拠する必要があります。命令コードを含むセクションの場合は、

.text.<関数名>

という命名規則に従って入力してください。図25は、arm_cmplx_mag_f32関数をITCMに配置する例を示しています。

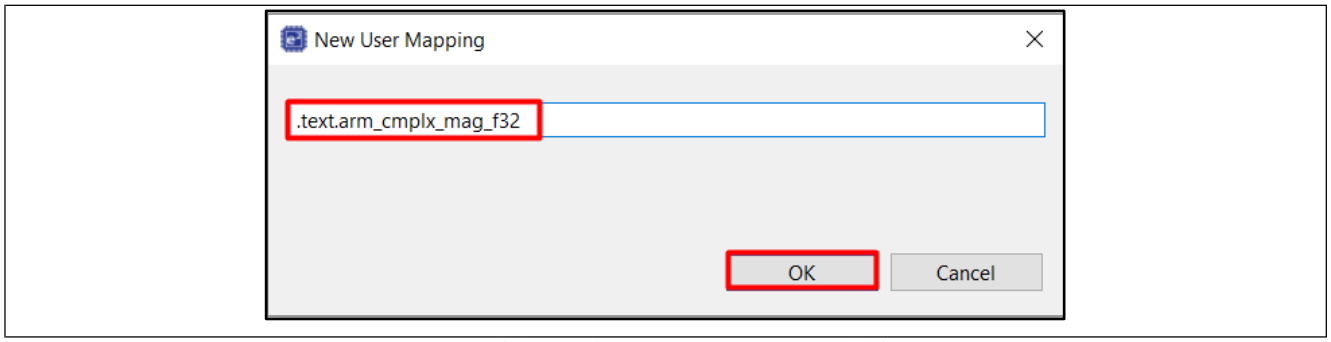


図25.命令コード用の入力セクション名の定義

図26に示す設定を完了した後、変更内容を反映するために次の操作を行います。**Generate Project Content**をクリックし、その後**Build Project(プロジェクトのビルド)**を選択して、更新されたプロジェクト設定を反映したコード生成とコンパイルを実行します。

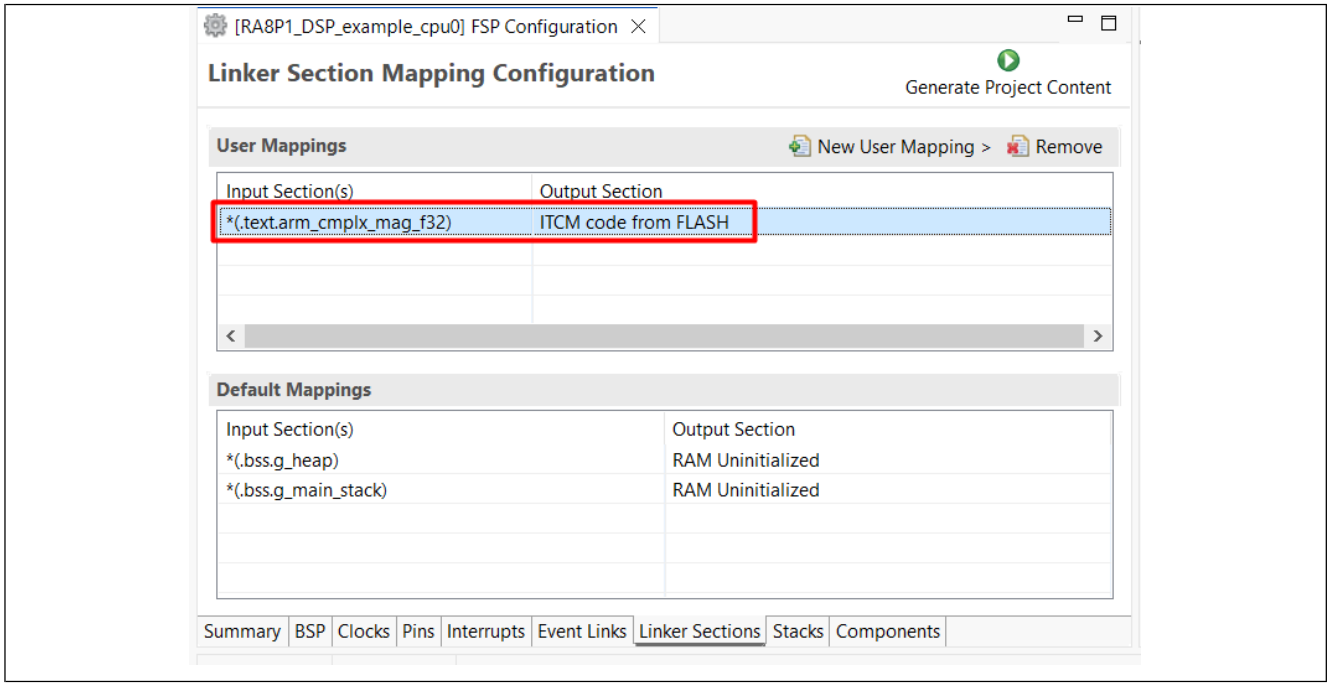


図26.ITCMセクションにおける関数の正常なセットアップ

ビルドが成功すると、**Debug/*.map**に生成されるマップファイルを確認することで、コードやデータが指定したメモリセクションに正しく配置されているかを確認できます。このファイルには、セクション割り当てを含む詳細なメモリ配置情報が記載されています。詳細は図27を参照してください。

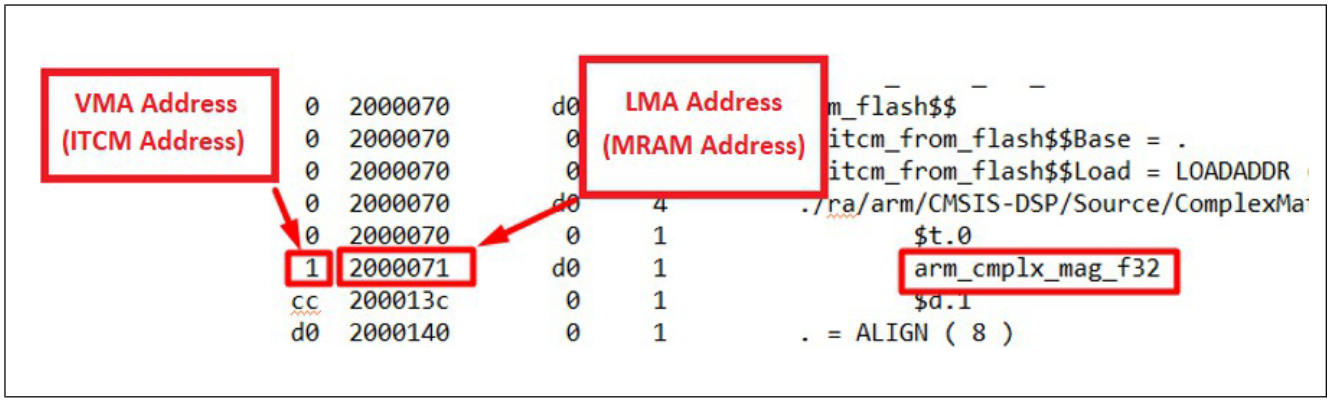


図27.ITCMにおけるコード配置の確認

4.4.2 DTCMを使用した性能の向上

実行時の性能を向上させるために、データバッファをDTCM（Data Tightly Coupled Memory）領域に割り当てることができます。これらのバッファは、e² studioのFSP Configuration設定にある**Linker Sections**タブから管理します。

以下では、RA8P1_DSP_exampleプロジェクトで使用されているtestInput_f32_10khzバッファ（図28参照）を、Linker Sections設定画面においてDTCMに配置する手順を示します。

Linker Sectionsタブから、**New User Mapping > DTCM > Data initialized from … > DTCM data from FLASH** を選択します（図29参照）。

この設定により、起動時にデータがMRAMからDTCMへコピーされ、実行中は高速なデータアクセスが可能になります。

```
/* -----------------------------------------------------------------------
Test Input signal contains 10KHz signal + Uniformly distributed white noise
** ----------------------------------------------------------------------- */

float32_t testInput_f32_10khz[2048] =
{
-0.865129623056441,     0.000000000000000,  -2.655020678073846,     0.0000000000
-2.899160484012034,     0.000000000000000,  2.563004262857762,  0.00000000000000
0.048366940168201,  0.000000000000000,  -0.145696461188734,     0.00000000000000
-1.176633086028377,     0.000000000000000,  3.690223557991855,  0.00000000000000
2.739754205367484,  0.000000000000000,  -0.062610410524552,     0.00000000000000
1.195039415434387,  0.000000000000000,  -2.177388969045026,     0.00000000000000
```

図28.testInput_f32_10khzバッファをDTCMに配置

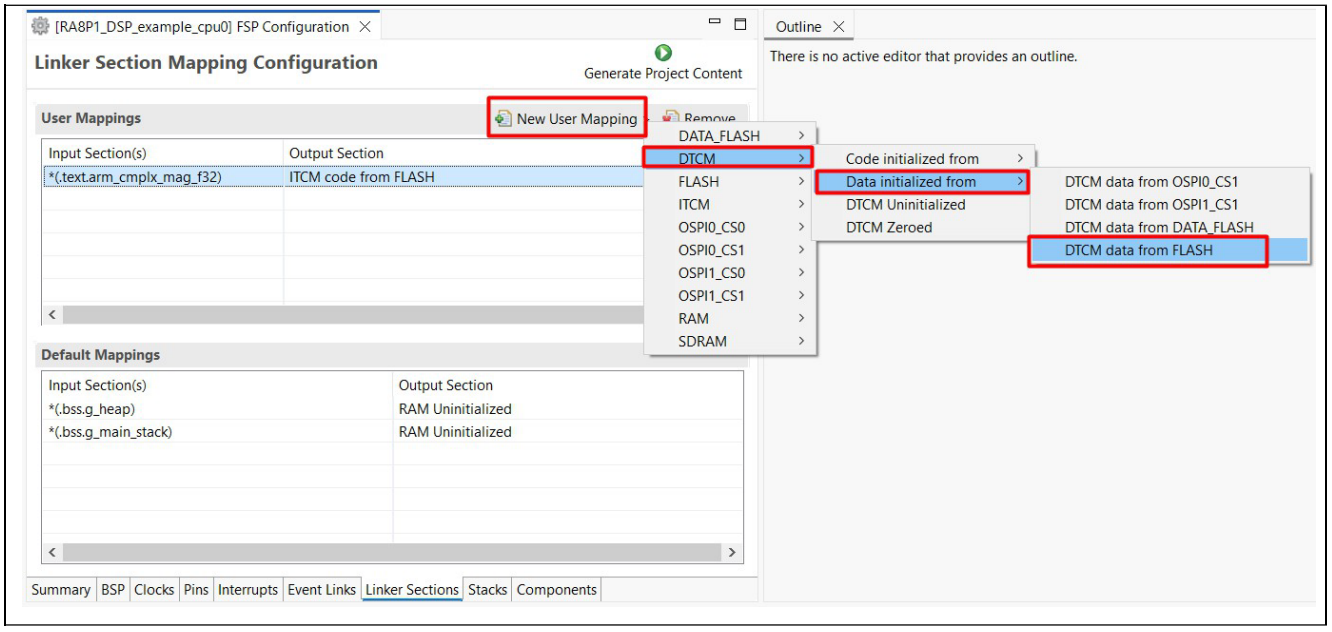


図29.DTCM用の新しいセクションマッピングの定義

次に、初期化データに対応する**Input section name**を設定します。セクション名は、以下の形式で指定してください。

.data.<バッファ名>

例えば、初期化されたバッファ**testInput_f32_10khz**を割り当てる場合は、次のように指定します。
.data.testInput_f32_10khz

この設定例を**図30**に示します。

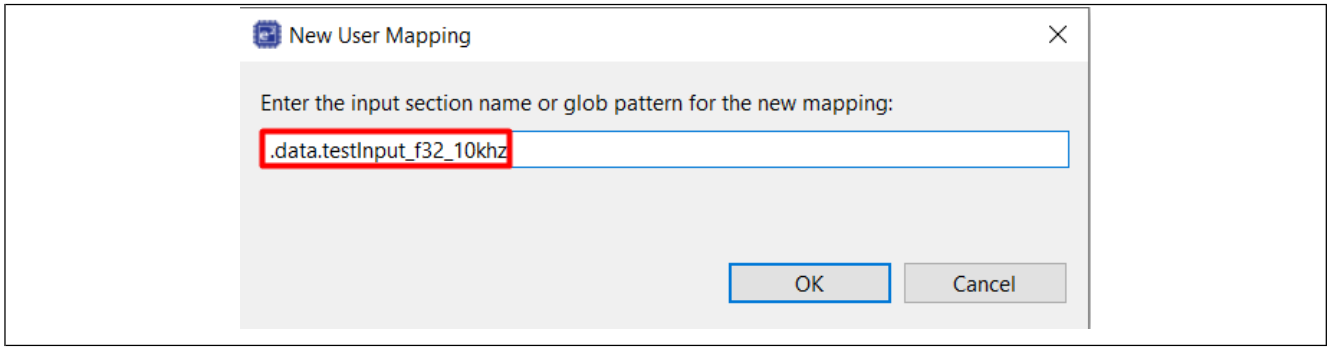


図30.初期化データ用の入力セクション名の定義

図31に示す設定を完了した後、**Generate Project Content**をクリックし、その後**Build Project**(**プロジェクトのビルド**)を選択して、変更内容を反映したコード生成とコンパイルを行います。

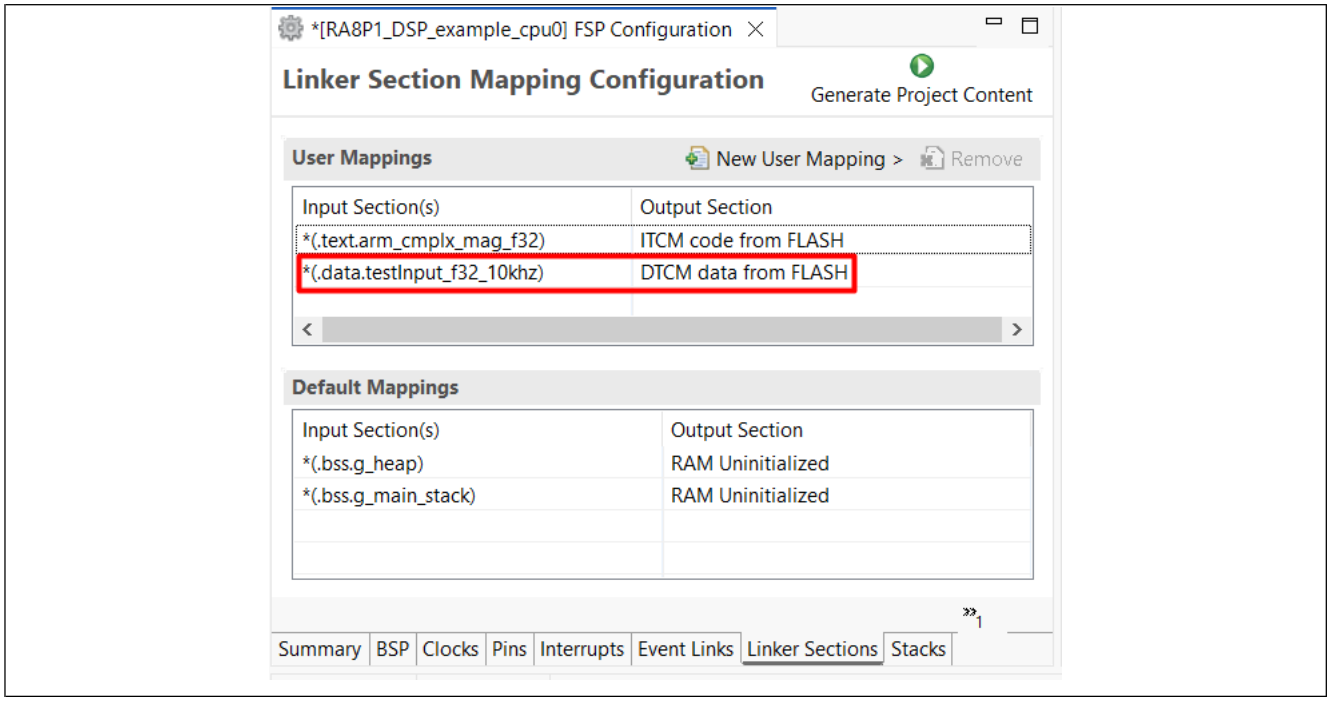


図31.DTCMセクションにおけるデータの正常なセットアップ

ビルドが成功すると、**Debug/*.map**に生成されるマップファイルを確認することで、コードやデータが指定したメモリセクションに正しく配置されているかを確認できます。このファイルには、セクション割り当てを含む詳細なメモリ配置情報が記載されています。詳細は図32を参照してください。

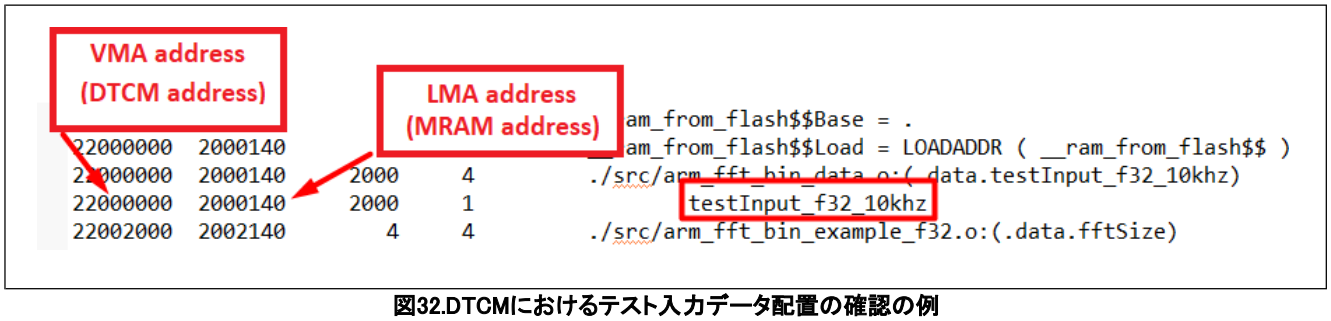


図32.DTCMにおけるテスト入力データ配置の確認の例

4.4.3　CTCMを使用した性能の向上

CPU1上で時間制約の厳しい処理の性能を高めるには、特定の関数の命令コードをCTCMに配置します。この設定は、e² studio上のCPU1プロジェクトに含まれるconfiguration.xmlの**Linker Sections**の設定から行います。

関数をCTCMに明示的に割り当てることで、命令フェッチのレイテンシが低減され、実行速度が向上します。以下では、e² studioのLinker Sections設定画面を使用して、CPU1の`ipc_callback()`関数をCTCMに配置する手順を示します。この方法により、コールバック処理は高速かつ決定論的なメモリアクセスで実行され、プロセッサ間通信イベントに対する応答性が向上します。

1. e² studioでプロジェクトを開きます。
2. **configuration.xml**をダブルクリックします。
3. **Linker Sections**タブを開きます。
4. `ipc_callback()`関数を、指定されたCTCMセクションに割り当てます（図33、図34参照）。

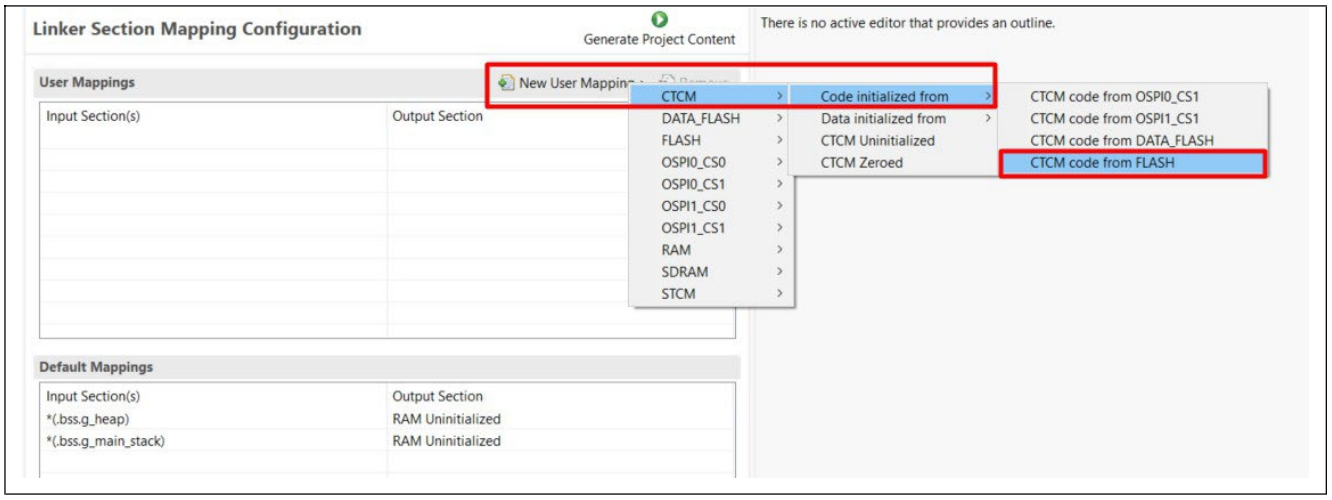


図33.CTCM用の新しいセクションマッピングの定義

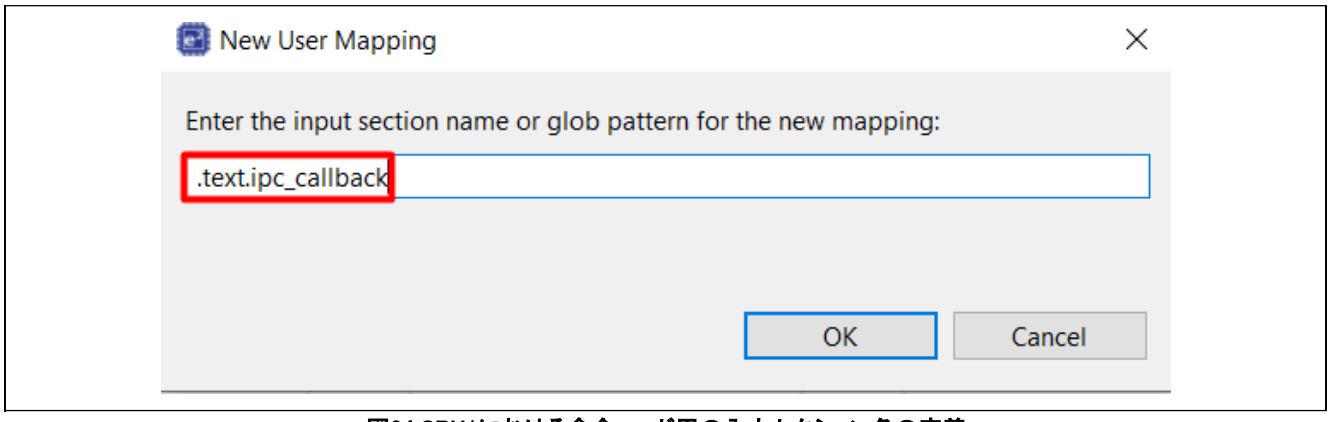


図34.CPU1における命令コード用の入力セクション名の定義

設定を完了した後、**Generate Project Content**をクリックし、続いて**Build Project（プロジェクトのビルド）**を選択して、変更内容を反映したコード生成とコンパイルを行います。
ビルドが正常に完了した後は、**Debug/*.map**に生成されるマップファイルを確認することで、コードやデータが指定したメモリセクションに正しく配置されているかを確認できます（図35参照）。

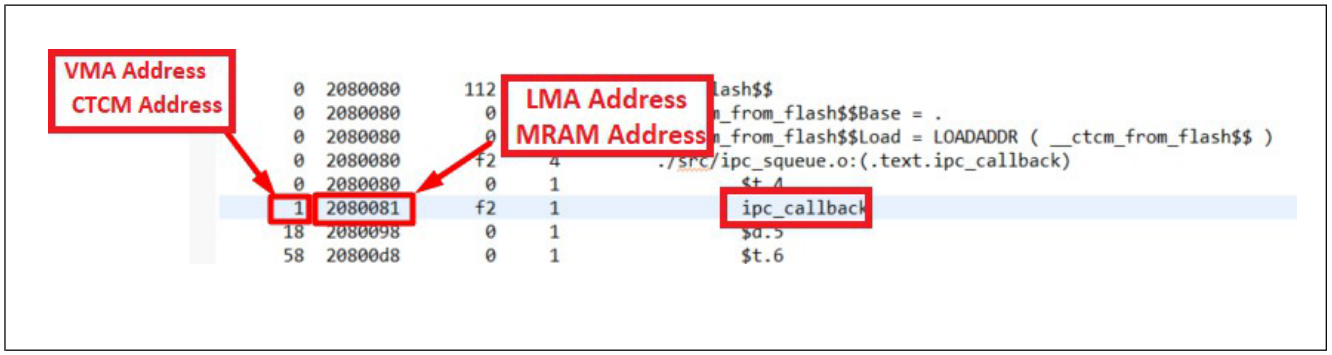


図35. ipc_callbackのCTCMへの割り当て成功例

4.4.4 Cortex®-CM85コアのデータキャッシュを活用した性能の向上

RA8デバイスのデフォルト設定では、FSPは常にCM85の命令キャッシュ（I-Cache）を有効化しており、必要に応じてキャッシュの整合性も管理します。また、CM85のデータキャッシュ（D-Cache）については、図36に示すように、BSP設定から任意に有効化することが可能ですが、デフォルトでは無効になっています。

システムでキャッシュを使用する場合は、キャッシュコヒーレンシについて考慮することが重要です。詳細については、Cortex-M85のキャッシュに関するドキュメントを参照してください。

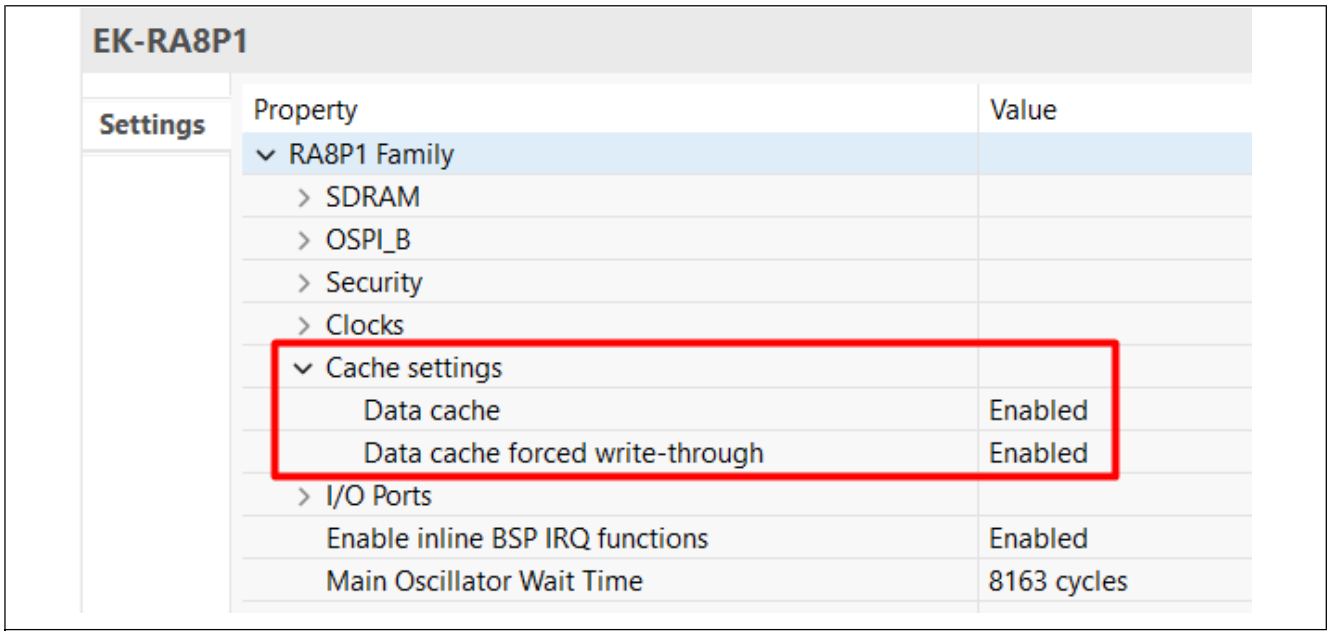


図36. CM85データキャッシュ(D-Cache)有効化

4.4.5　ニューラルプロセッシングユニット(NPU)の使用

RA8P1デュアルコアMCUには、Ethos-U55が内蔵されています。Ethos-U55は、近年の言語モデルやビジョンモデルの基盤となるトランスフォーマーベースのモデルをエッジ環境で扱えるNPUであり、32～256MACユニットまでスケール可能な構成を備えています。これにより、高性能なエッジAIアプリケーションを、消費電力を抑えながら実現できます。

Ethos-U55は、従来のEthos-Uシリーズと同一のツールチェーンに対応しているため、Armベースの機械学習（ML）ツールへの既存投資を活かしたスムーズな移行が可能です。

このNPUはホストCPUと連携して動作し、画像認識、音声認識、異常検知といった用途において、効率的なAI/MLアクセラレーションを提供します。MLの計算処理をCPUからEthos-U NPUへオフロードすることで、処理性能と電力効率の両立が可能となり、大きな電力増加を伴うことなくエッジAIを実装できます。

図37および図38には、NPUのブロック図と、Renesas FSPにおけるEthos設定例を示しています。

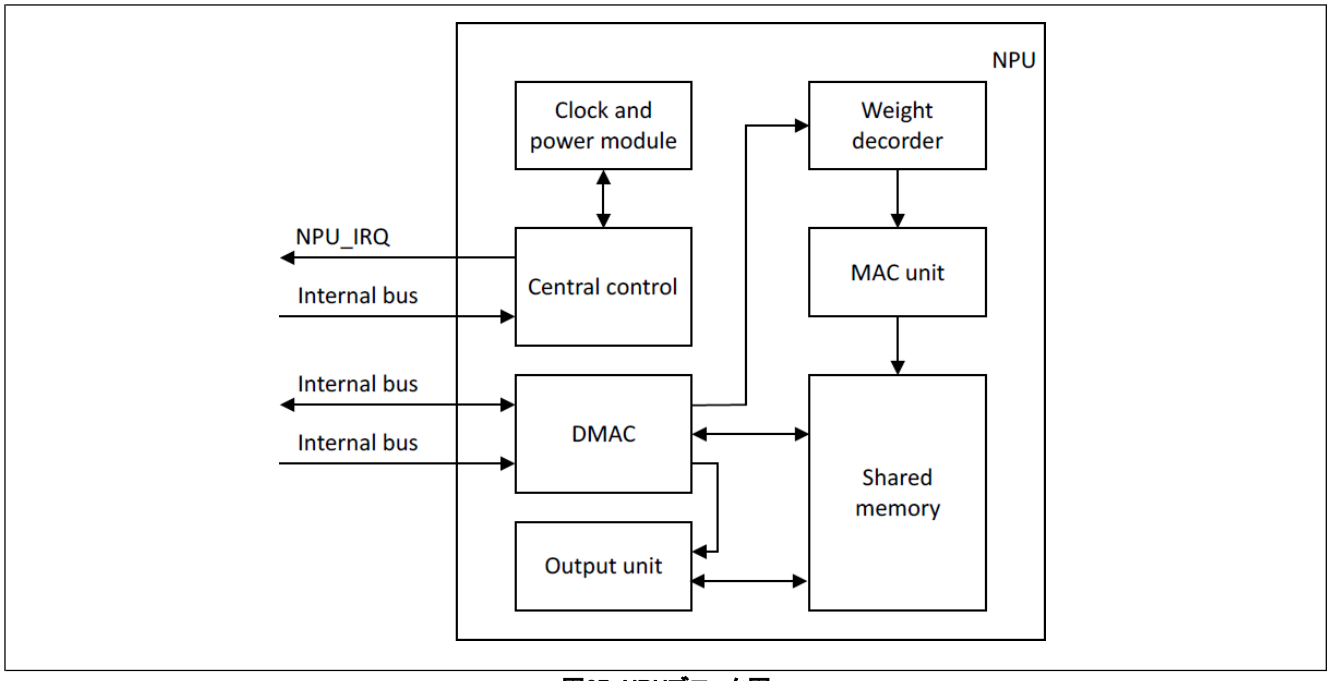


図37. NPUブロック図

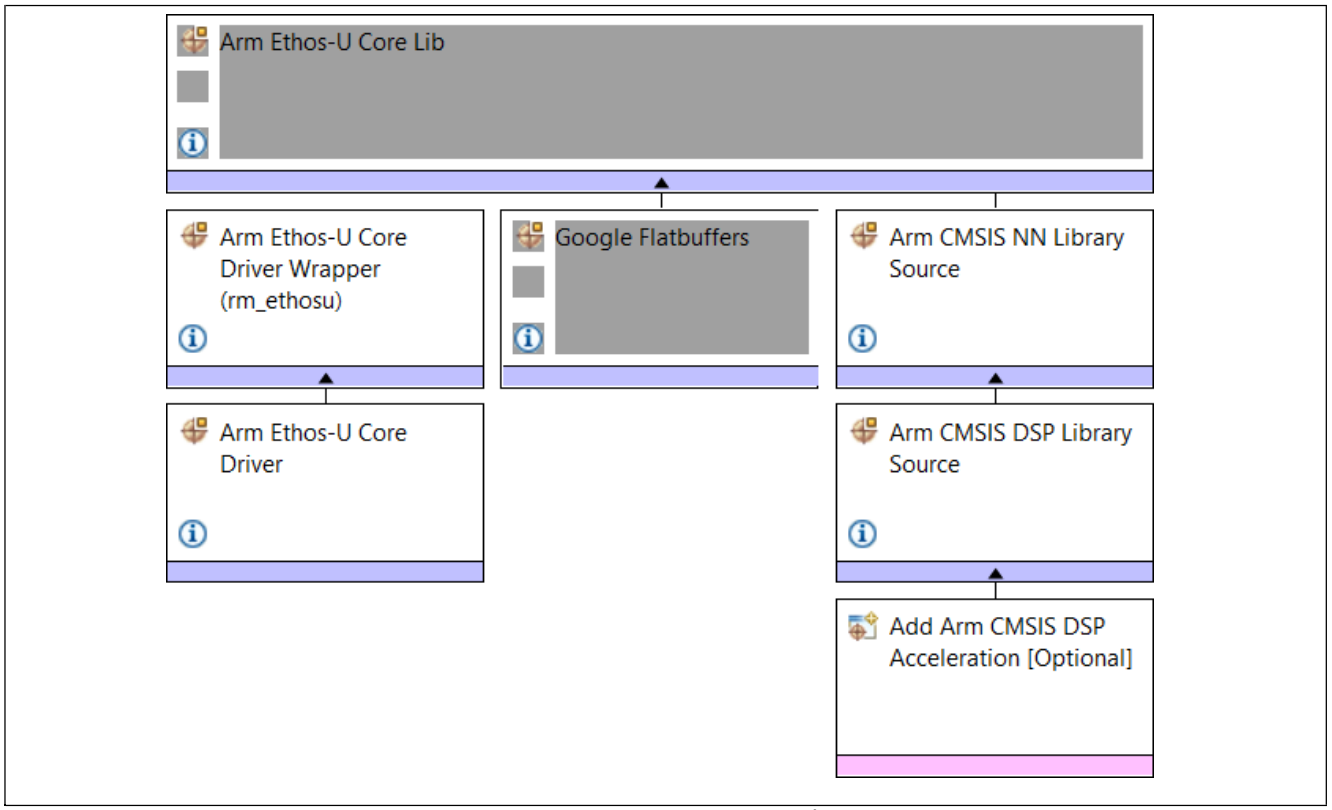


図38. Renesas FSPにおけるEthosサポートの例

RA8デュアルコアMCU向けAIアプリケーションの開発については、以下のアプリケーションノートを参照してください。

- 「Reference System Design for Vision AI Design using Ethos-U NPU」(ドキュメント番号:R11AN0995)
- 「Using the Ethos-U NPU with RA8 MCUs」(ドキュメント番号:R01AN7712)

5. アプリケーションプロジェクト

5.1 IPC – 共有メモリプロジェクト

本アプリケーションプロジェクトの実装は、以下の仕様に基づいて構成されています。
CPU1はユーザインタフェースの管理および端末へのログ出力を担当し、CPU0はリアルタイム制御およびセンサデータ処理を担当します。

両コア間でのタスク分担の概要を図39に示します。
また、本アプリケーションプロジェクトで使用される周辺リソースの一覧を表1に示します。

表1.本プロジェクトで使用されるリソース一覧

CPU0	CPU1
プロセッサ間通信(IPC0 & IPC1)	プロセッサ間通信(IPC0 & IPC1)
ハードウェアセマフォ	ハードウェアセマフォ
リアルタイムクロック制御	シリアル通信インタフェースUART
12ビットA/Dコンバータ – 温度センサ	I/Oポートの制御
I/Oポート制御 – ユーザLED	

本サンプルアプリケーションでは、すべての実行時の情報が、CPU1により管理されるSCI-UARTインタフェースを介して、Tera Termのターミナルコンソールに送信されます。
一方、CPU0はリアルタイムクロック(RTC)の制御、センサデータの取得、およびユーザLEDの制御を担当します。

本アプリケーションの機能概要を図40に示します。

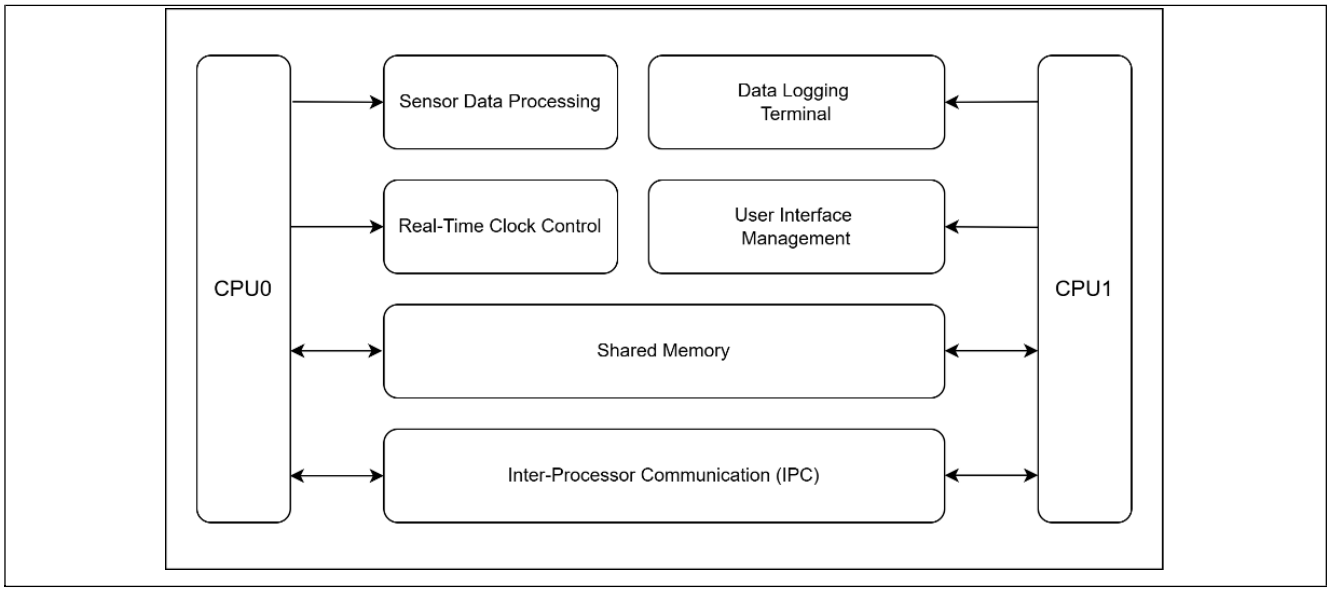


図39. デュアルコアBareMetalサンプルにおけるタスク分割

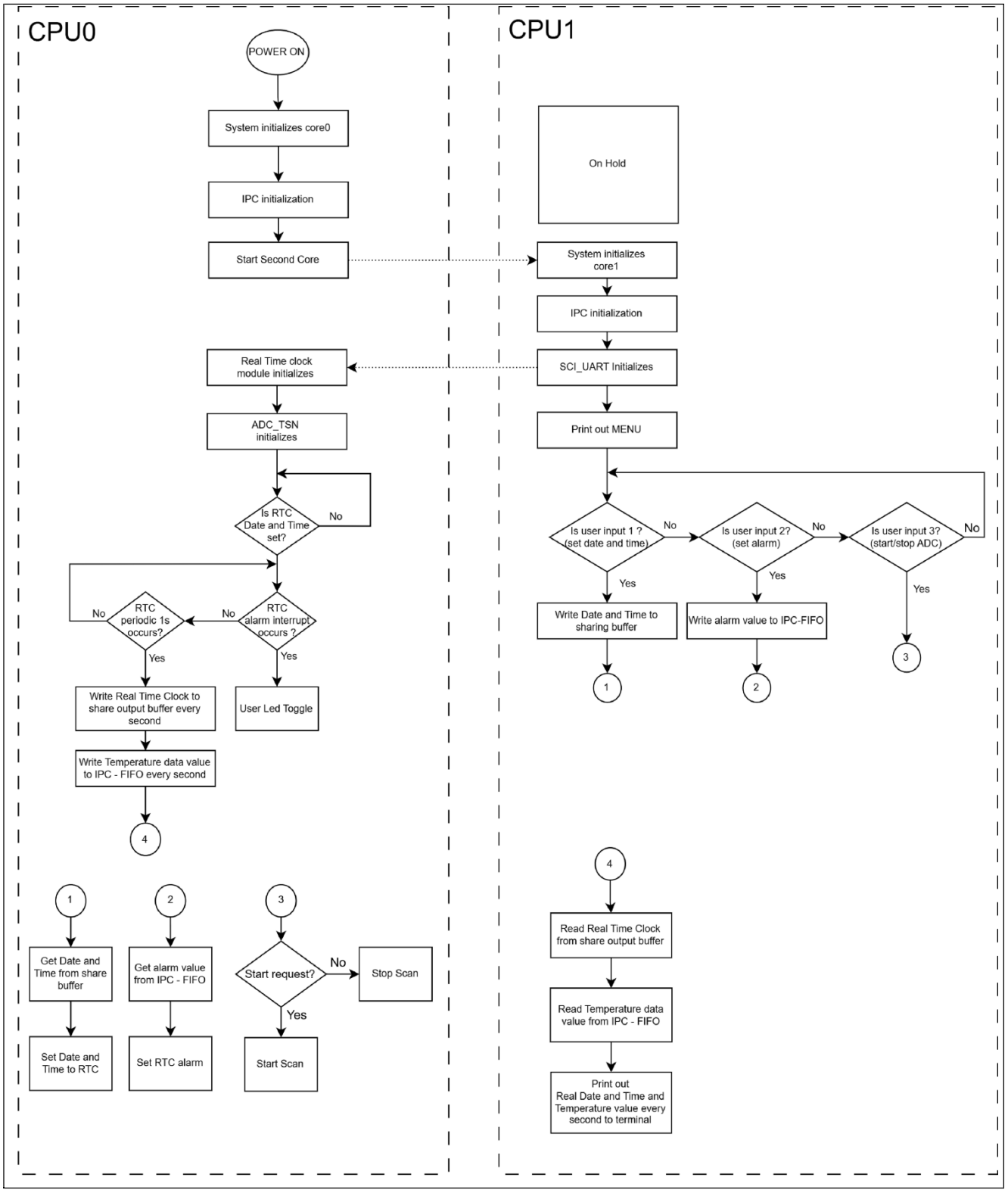


図40. IPC/セマフォおよび共有メモリ機能を使用したアプリケーション

表2に、本アプリケーションで使用されているIPCおよびハードウェアセマフォのAPIをまとめています。
これらのAPIは、デュアルコアシステムにおけるプロセッサ間通信および同期処理を実現するために使用されます。

表2.プロセッサ間通信API一覧

関数	説明
R_IPC_Open	IPCインスタンスの設定
R_BSP_IpcNmiEnable	IPC-NMI用のユーザコールバックを設定
R_IPC_MessageSend	IPCメッセージFIFOへメッセージを送信
R_IPC_EventGenerate	他コアに対してマスカブル割り込みを生成
R_BSP_IpcSemaphoreTake	ハードウェアセマフォを取得
R_BSP_IpcSemaphoreGive	ハードウェアセマフォを解放
R_BSP_IpcNmiRequestSet	他コアに対してノンマスカブル割り込み（NMI）を発生

5.1.1　アプリケーションにおけるプロセッサ間通信の実装

プロセッサ間通信（IPC）は、同一CPUシステム内の2つのプロセッサ間で、ハードウェアリソースの共有およびデータ交換を行うための仕組みです。

IPCモジュールは最大16個のハードウェアセマフォをサポートしており、コア間の効率的な同期処理を可能にします。
また、IPCは割り込み駆動型イベントの生成にも対応しており、マスカブル割り込みおよびノンマスカブル割り込み（NMI）を用いたプロセッサ間通知を実現します。

図41および図42に、アプリケーションにおけるIPCマスカブル割り込みおよびIPCノンマスカブル割り込みの実装例を示します。

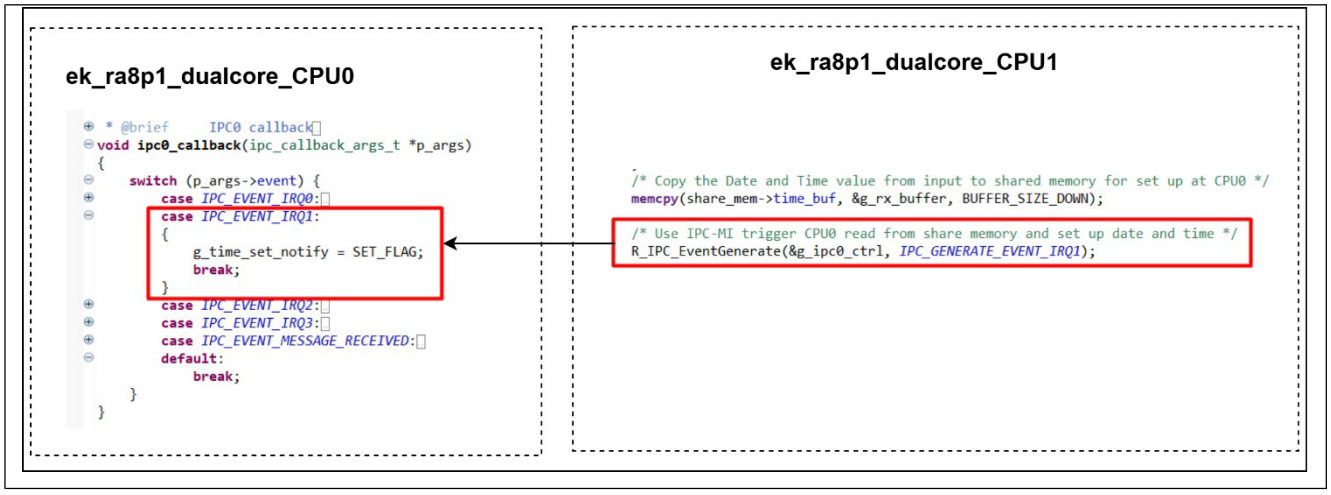


図41. IPCマスカブル割り込みサンプルアプリケーションの例

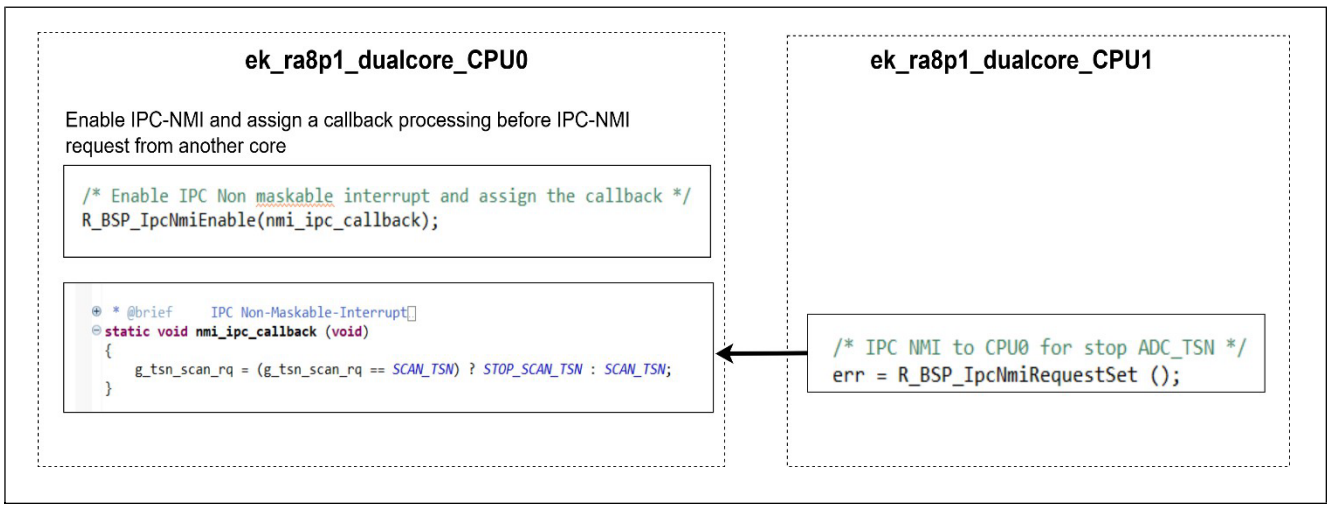


図42. IPCノンマスカブル割り込みサンプルアプリケーションの例

本アプリケーションでは、2つのCPU間で簡単なデータをやり取りするために、一方向FIFOも使用しています。具体的には、温度データをCPU0からCPU1へ送信し、ユーザのアラーム設定値をCPU1からCPU0へ送信します。このデータフローの例を図43に示します。

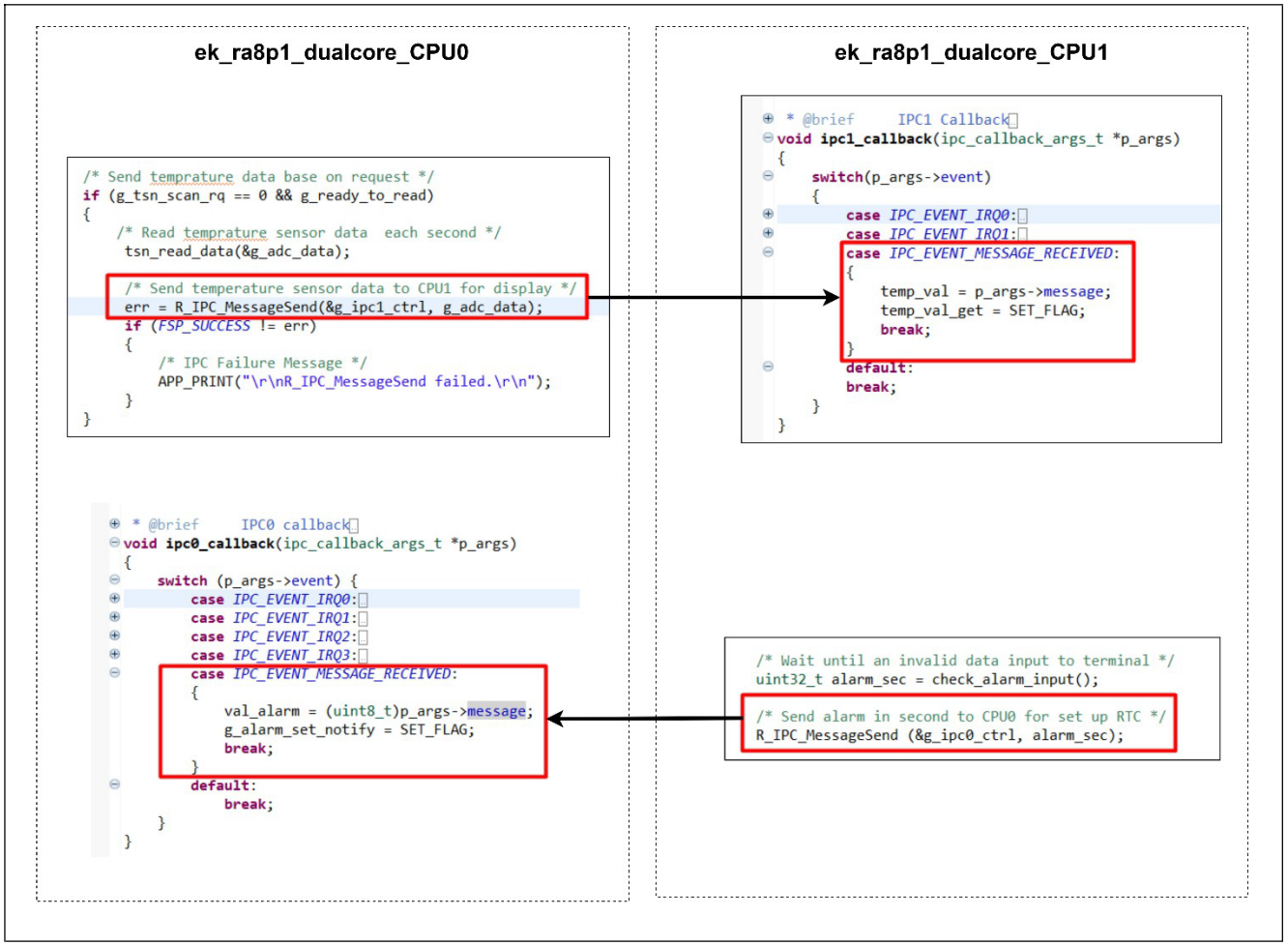


図43. IPCメッセージFIFOを用いたサンプルアプリケーション例

5.1.2　アプリケーションにおける2コア間の共有メモリの実装

2つのコア間で共有メモリのメカニズムを実装するには、ソリューションプロジェクト内に専用のメモリ領域を定義する必要があります。本アプリケーションでは、共有メモリをCPU0のRAMの最終32KB領域に割り当てています。このメモリ領域を定義するための設定手順を以下に示します。

まず、RAM_CPU0_Sのサイズを0xE2000に変更します。

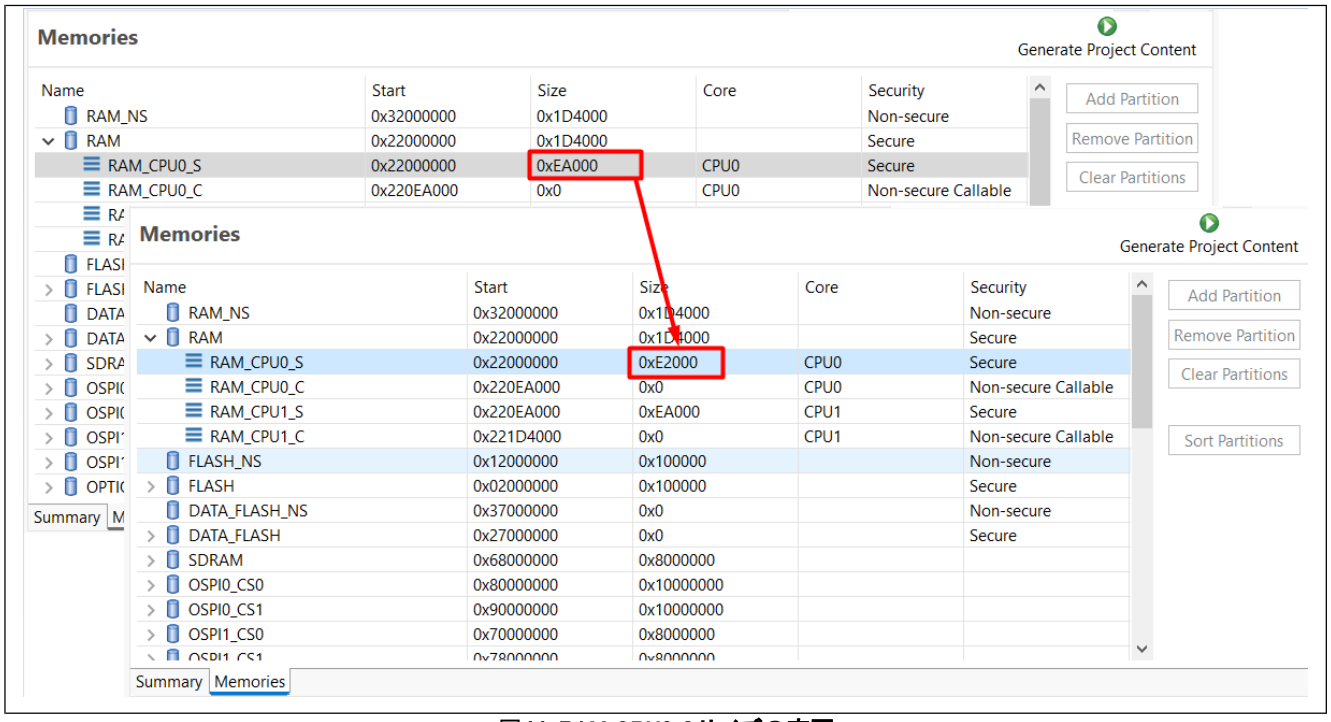


図44. RAM_CPU0_Sサイズの変更

RAM > **Add Partition**をクリックします。

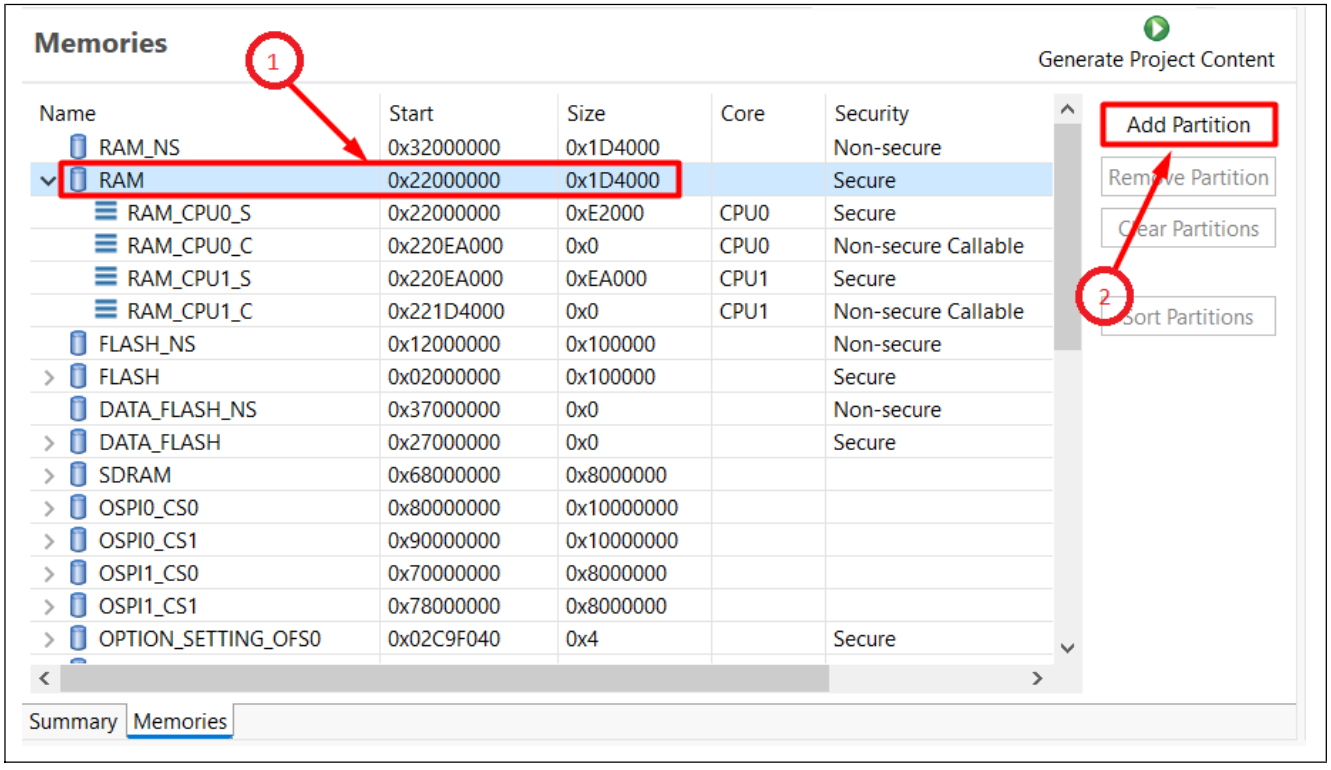


図45. 共有メモリ領域のパーティションの追加

図46に示すNew Partitionポップアップウィンドウで、必要な項目をすべて入力し、OKをクリックします。Multicore Flat Bare-Metal Projectの場合、各項目は以下のように設定します。

- **Name**：SHARED_MEM
- **Start**：0x220E2000
- **Size**：0x8000
- **Core**：CPU0

・**Security**：Secure

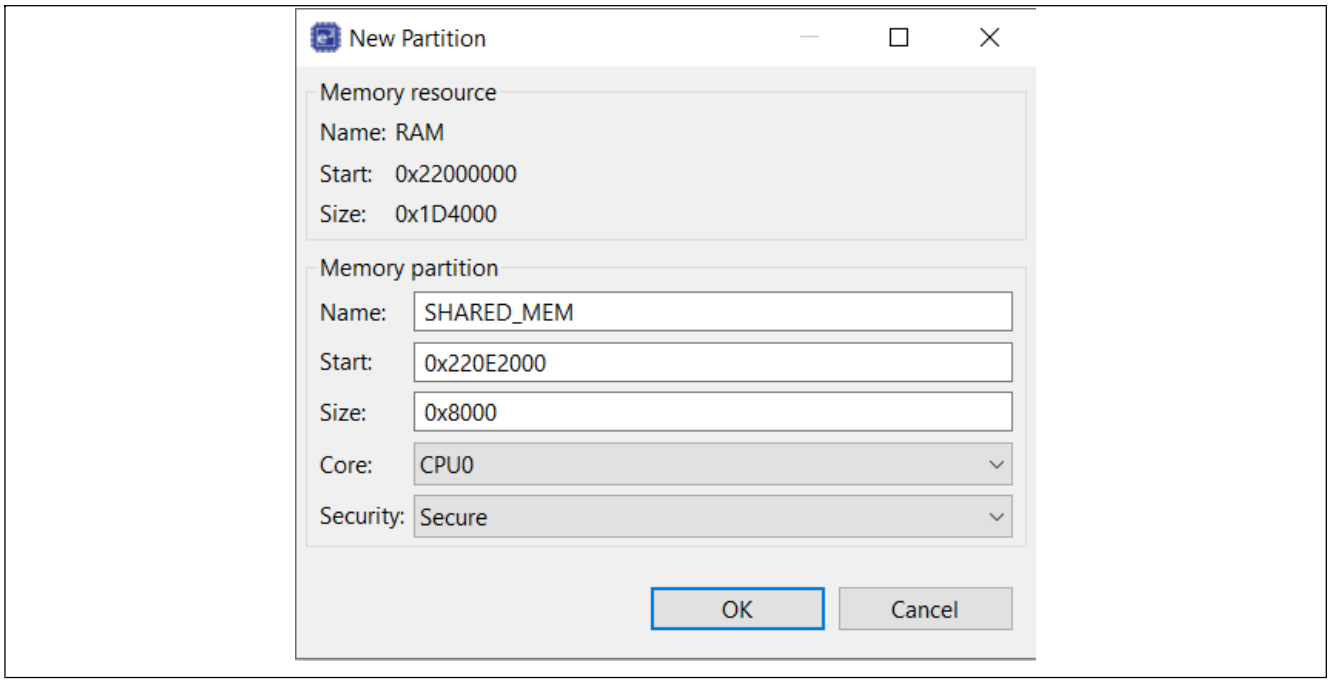


図46. 共有メモリパーティションの定義

「Partition does not follow address order」というエラーメッセージが表示された場合、メモリパーティションがアドレス順に正しく並んでいないことを示しています。この場合、図47に示すようにSort Partitionsをクリックしてパーティションを並び替えてください。

これにより、正しいメモリマッピングが確保され、ビルド時のアドレス競合を防ぐことができます。

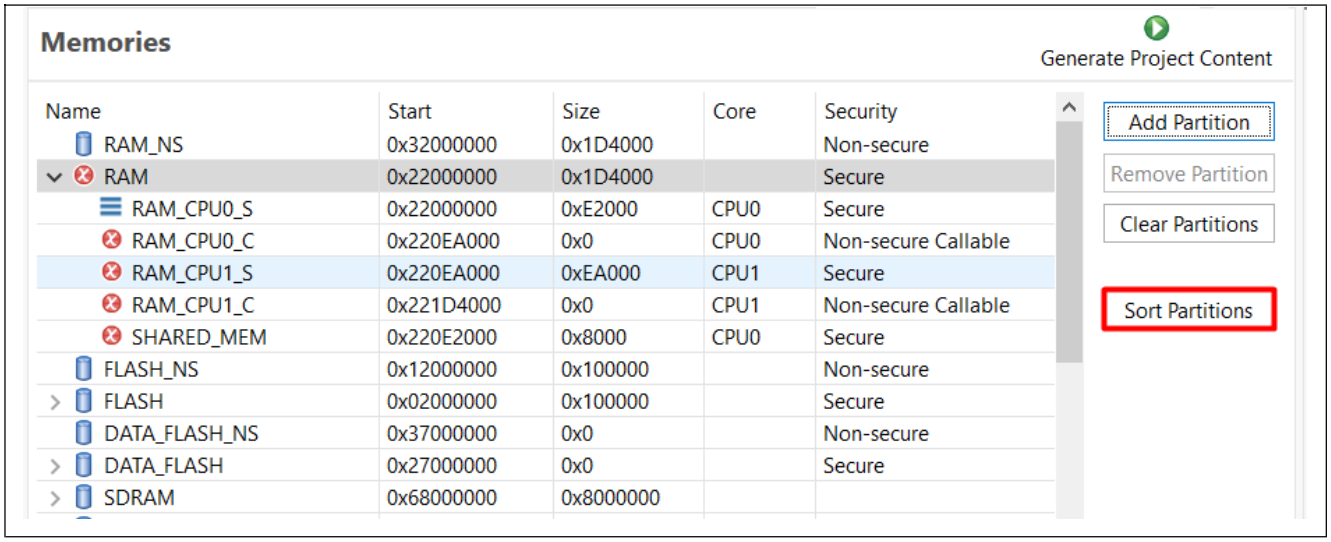


図47. RAMにおけるパーティションの並び替え

正しく設定されたパーティションは、図48に示すとおりです。

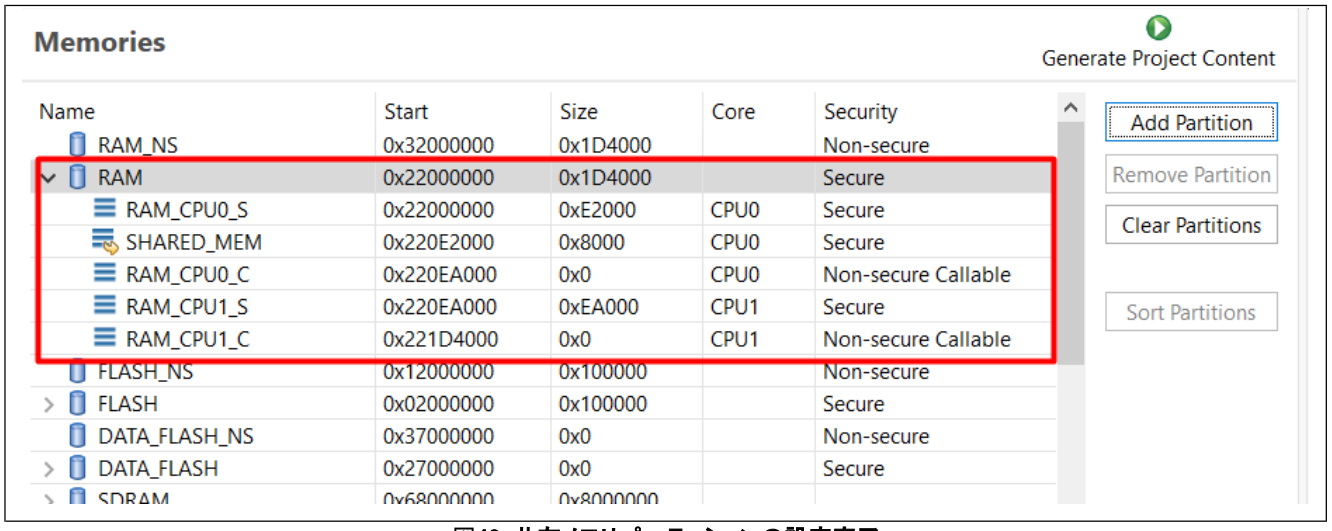


図48. 共有メモリパーティションの設定完了

設定完了後、ソリューションプロジェクトを右クリックし、**Build Project（プロジェクトのビルド）**を選択して変更内容をビルドします。ビルドが正常に完了したら、<solution_project>/build/*.sbdに生成される.sbdファイルをダブルクリックし、共有メモリパーティションが正しく作成されていることを確認します。
このファイルにより、共有メモリ領域が正しく分割されていることを確認できます。

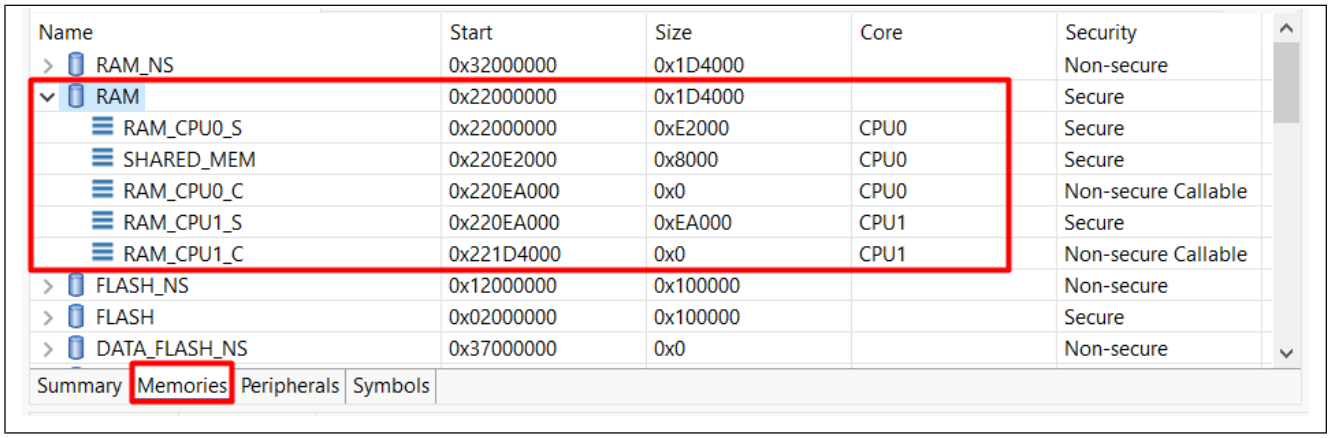


図49. 共有メモリパーティションの作成成功

以降、アプリケーションは共有メモリ領域内にバッファを配置できるようになります。その実装例を図50に示します。この設定により、バッファが共有メモリ空間に配置され、2つのコア間でスムーズなデータ交換が可能となります。

```
/***********************************************************************
 * Global Variables
 ***********************************************************************/
uint8_t g_periodic_irq_flag = RESET_FLAG;
uint8_t g_alarm_irq_flag = RESET_FLAG;
share_mem_t share_memory BSP_PLACE_IN_SECTION(".shared_mem") = {.buf_out = {RESET_VALUE},
                                                               .length = RESET_VALUE,
                                                               .time_buf = {RESET_VALUE}};
```

図50. 共有メモリへのバッファの配置

共有メモリ管理の基本原則は、任意の時点で1つのCPUのみが排他的にアクセスできることを保証する点にあります。本アプリケーションでは、ハードウェアセマフォとプロセッサ間マスカブル割り込み（MI）を組み合わせて使用し、2つのコア間でのアクセス制御および調停を行っています。この仕組みは、図20に示す制御フローに従って動作します。

5.2 RTOS/IPC/共有メモリ/TCMプロジェクト

これらのサンプルプロジェクトでは、RTOS、IPCモジュール、I/Oインタフェース、およびデータキャッシュが有効化されたTCMメモリを活用し、2つのCPUコア間の通信を実現しています。

本アプリケーションは、Arm公式のCMSIS-DSP FFTサンプルをベースにしており、CPU0（CM85コア）上で動作するように移植されています。演算処理は、性能を最大限に引き出すため、TCM上で実行されます。FFTの演算結果は、オンチップIPCモジュールをベースとしたRTOSキューを介してセカンドコアへ送信され、セカンドコアはその結果をUART経由でターミナルに出力します。

RA FSPを用いてArmのCMSIS-DSPサンプルを移植する際は、統合手順の詳細についてアプリケーションノートR01AN5865「Arm® DSP Examples」を参照してください。

本実装では、FreeRTOSベースのキューをハードウェアIPCメッセージFIFOと連携させることで、2つのコア間のメッセージトランザクションを管理しています。その構成を図51に示します。

用途やデータフローの要件に応じて、FreeRTOSのメッセージバッファやストリームバッファをプロセッサ間通信に使用することも可能です。

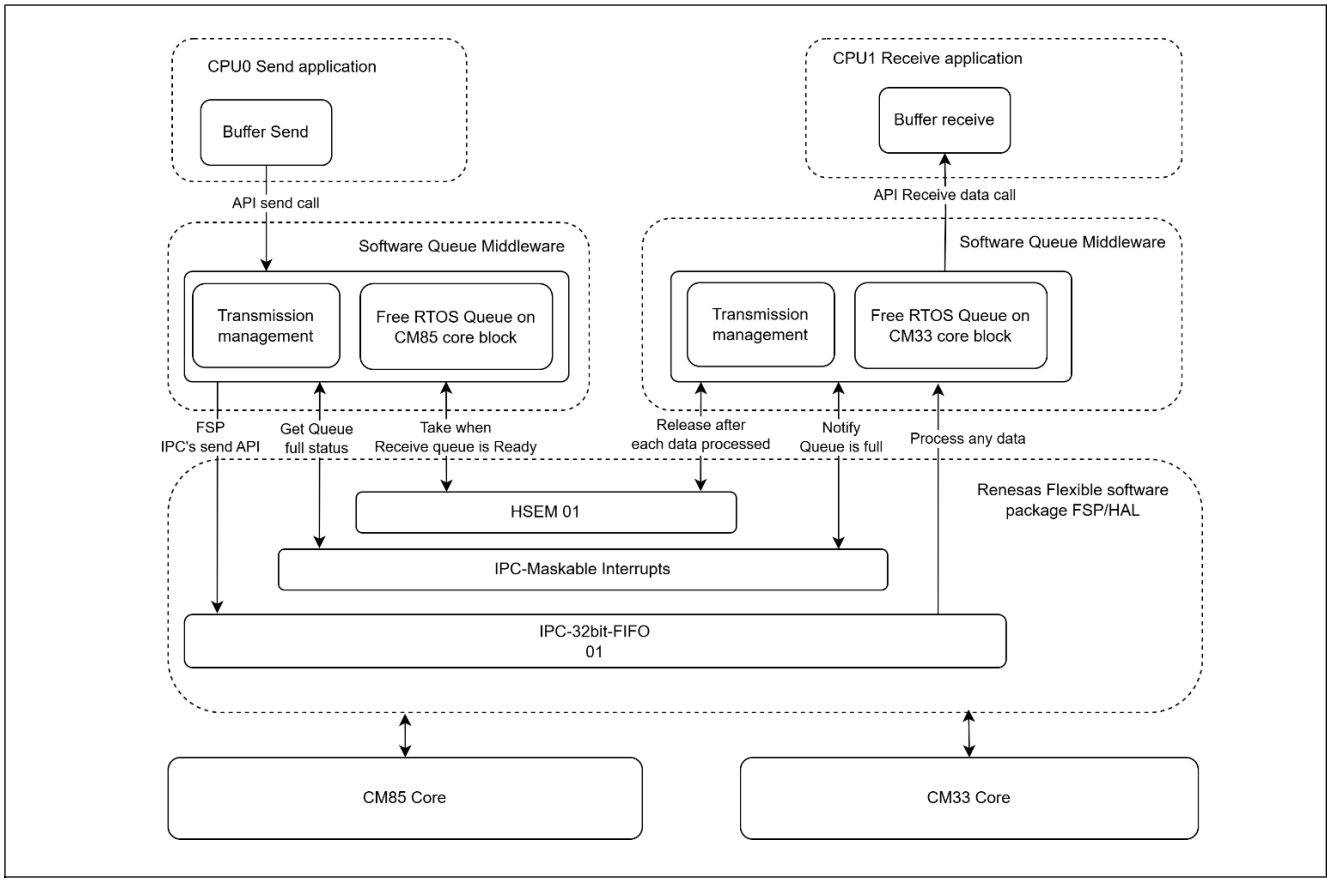


図51. IPCハードウェアを用いたFreeRTOSベースのプロセッサ間メッセージング

6. マルチコアFlat Bare-Metalプロジェクトの検証

e²studioプロジェクトは、デュアルコアアプリケーション開発を支援するために、分割プロジェクトによる開発モデルを採用しています。各コアのプロジェクトはそれぞれ独立したプロジェクトとして作成されており、プロセッサ間通信（IPC）を用いることで、2つのコア間の効率的な通信および共有メモリ管理を実現しています。

e²studio上でサンプルアプリケーションプロジェクトek_ra8p1_dualcoreをビルドおよび実行するには、以下の手順に従ってください。

6.1 プロジェクトのインポート

1. e² studio IDEを起動します。
2. Workspace launcherで任意のワークスペースを選択します。
3. 「ようこそ」のウィンドウを閉じます。
4. **ファイル > インポート**を選択します。
5. インポートダイアログボックスから**既存プロジェクトをワークスペースへ**を選択します。
6. ファイル名でアーカイブファイル "**ek_ra8p1_dualcore.zip**" を選択します
   「**Developing_with_RA8_Dual_Core_MCU.zip**」に含まれるアーカイブファイル「**ek_ra8p1_dualcore.zip**」を選択します。
7. 以下に示すようにソリューションプロジェクトと各コアで開発されたプロジェクトサンプルを選択し、**Finish（終了）**をクリックします

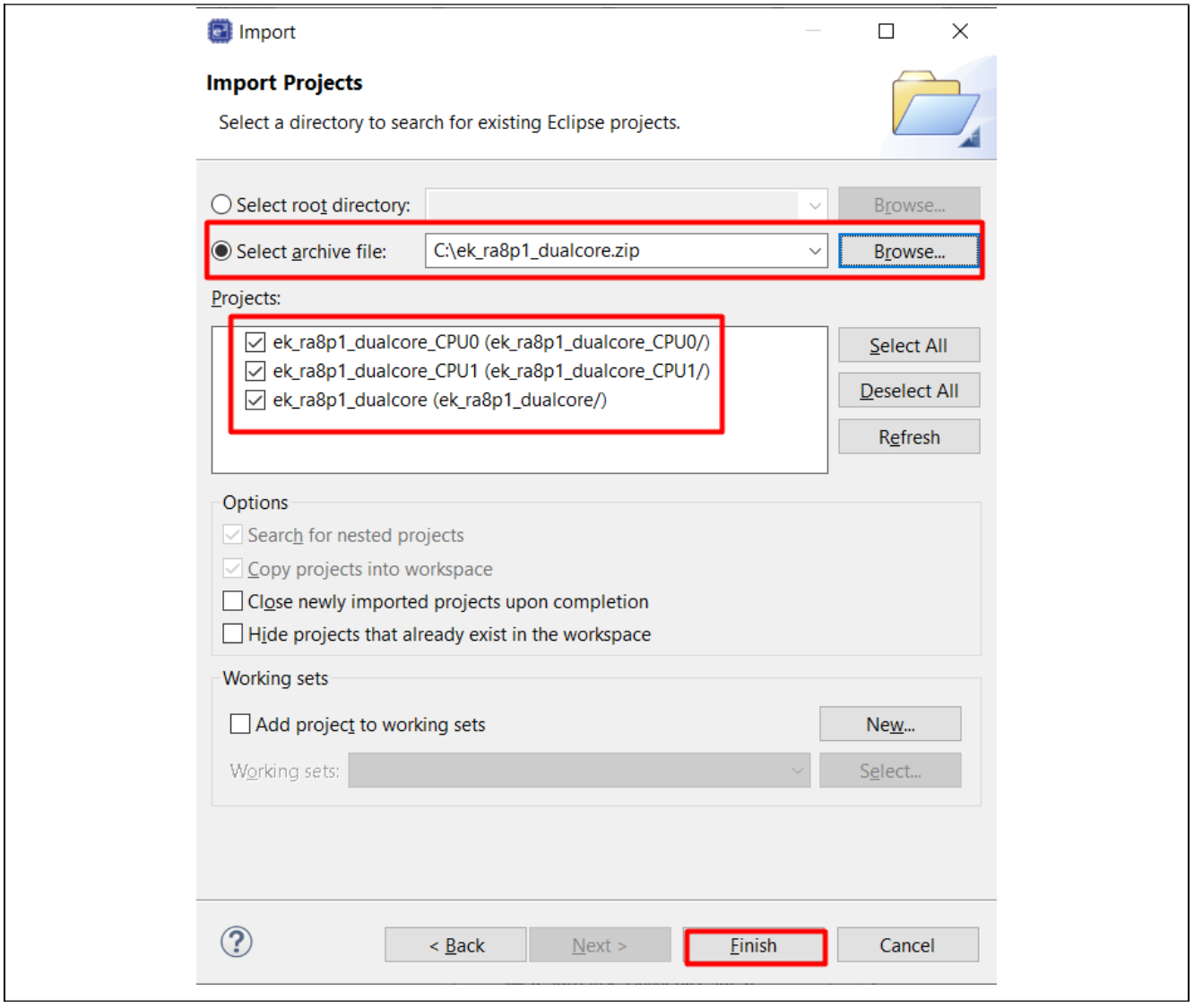


図52. ワークスペースへのプロジェクトのインポート

6.2 プロジェクトのビルド

最も簡単な方法は、ソリューションプロジェクトek_ra8p1_dualcoreを右クリックして**Build Project（プロジェクトのビルド）**を選択することです。ソリューションプロジェクトをビルドすると、両方のCPUプロジェクトが順番に自動的にビルドされます:最初にCPU0がビルドされ、次にCPU1がビルドされます。

プロジェクトを個別にビルドする場合は、CPU0プロジェクトを最初にビルドしてから、CPU1プロジェクトをビルドしてください。

6.2.1 CM85コア向けに開発されたプロジェクトのコンパイル

ek_ra8p1_dualcore_CPU0プロジェクトにある**configuration.xml**をダブルクリックし、「**Generate Project Content**」をクリックします。

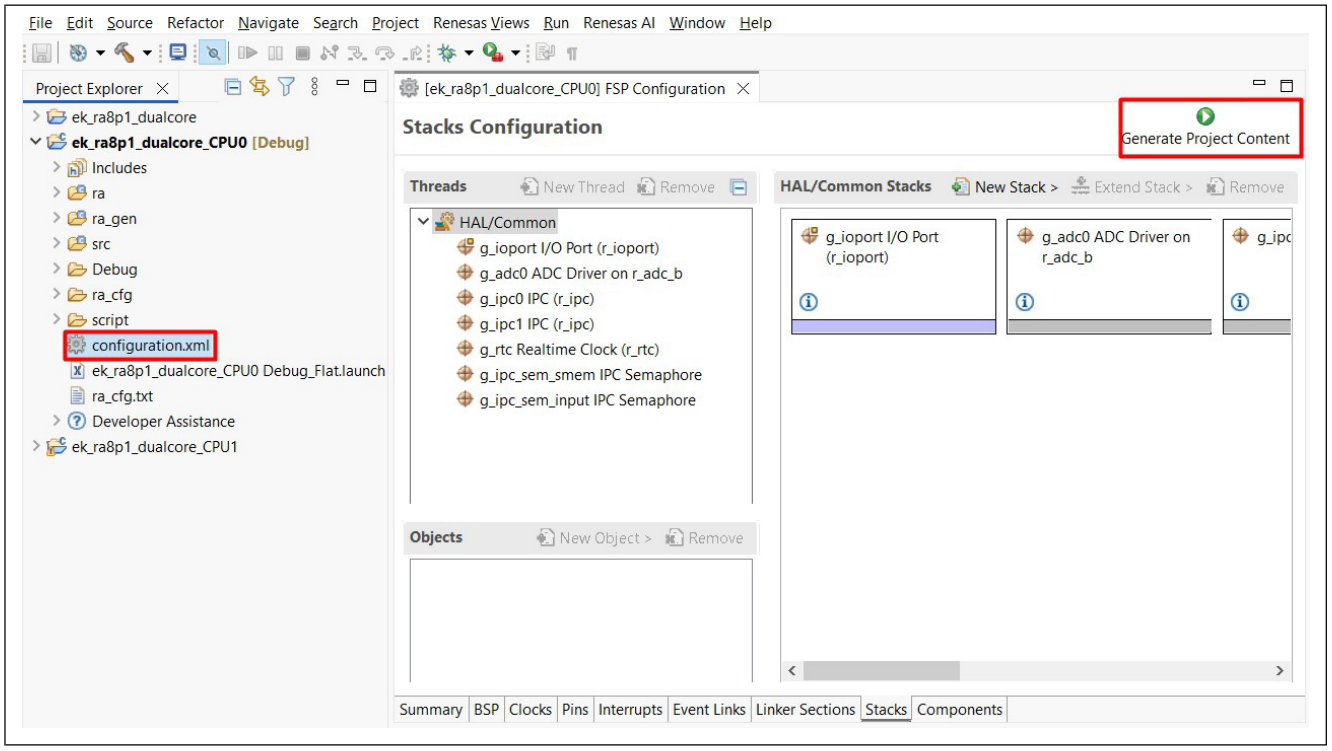


図53. CPU0プロジェクトにおけるプロジェクトコンテンツの生成

プロジェクトコンテンツを生成した後、**ek_ra8p1_dualcore_CPU0**を右クリックし、**Build Project（プロジェクトのビルド）**を選択してCPU0コアアプリケーションをコンパイルします。

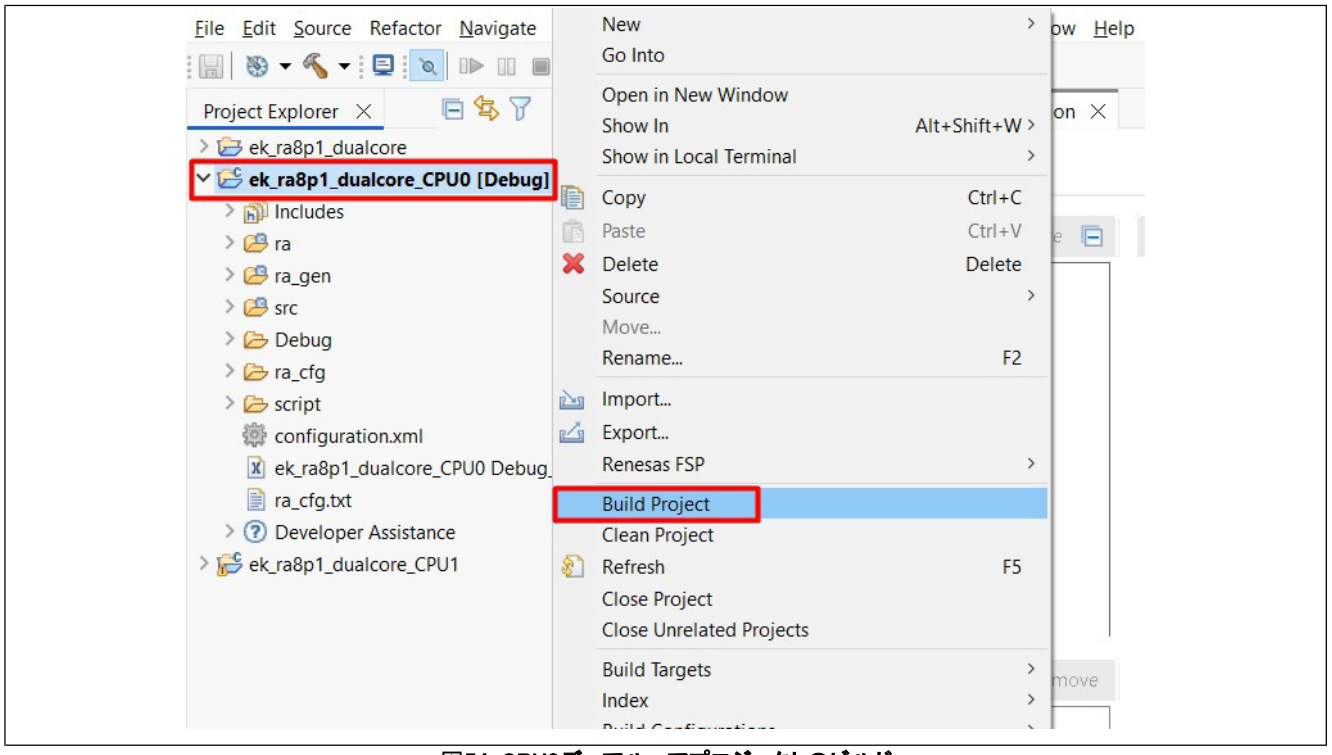


図54. CPU0デュアルコアプロジェクトのビルド

Build Logコンソールの出力を確認して、ビルドが正常に完了したことを確認します。

6.2.2 CM33コア向けに開発されたプロジェクトのコンパイル

ek_ra8p1_dualcore_CPU1プロジェクトで、**configuration.xml**をダブルクリックし、「**Generate Project Content**」をクリックして、CPU1コアの設定を適用します。

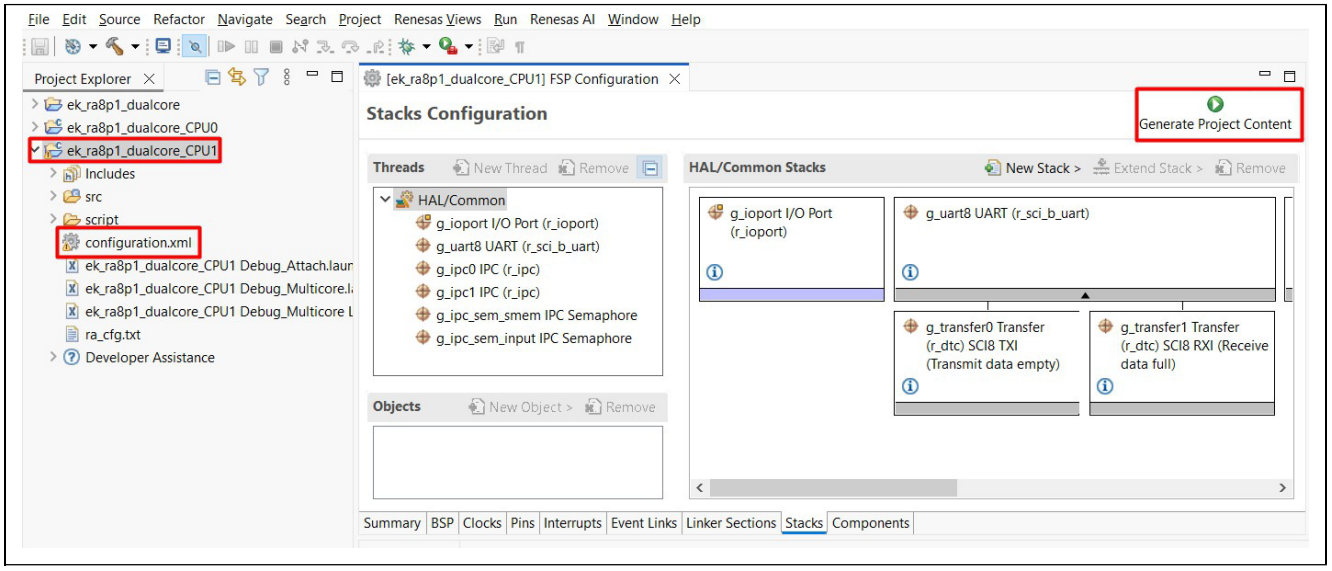


図55. CPU1プロジェクトでのプロジェクトコンテンツの生成

コンテンツが生成後、**ek_ra8p1_dualcore_CPU1**を右クリックし、**Build Project**(**プロジェクトのビルド**)を選択してCPU1コアアプリケーションをコンパイルします。

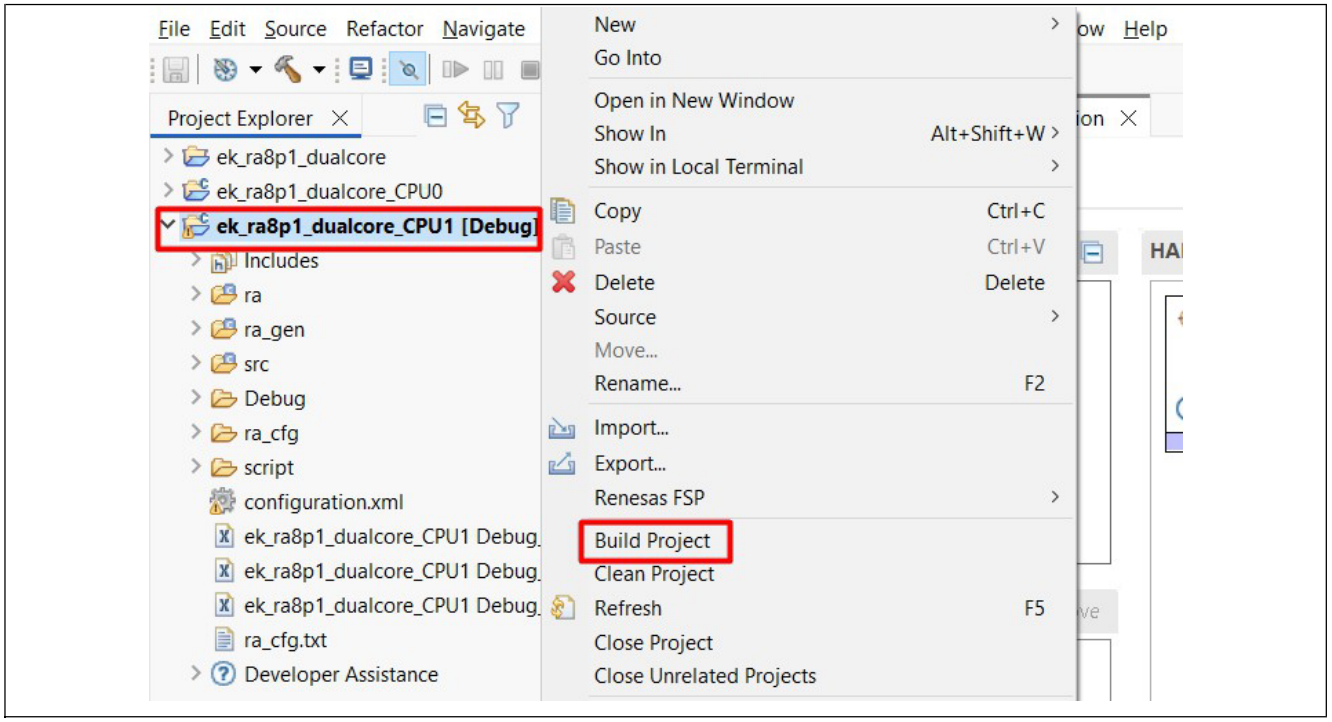


図56. CPU1デュアルコアプロジェクトのビルド

Build Logコンソールの出力を確認して、ビルドが正常に完了したことを確認します。

また、ソリューションプロジェクトを使用して、2つのコアをまとめてビルドすることも可能です。**ek_ra8p1_dualcore**ソリューションプロジェクトを右クリックし、図57に示すように**Build Project**(**プロジェクトのビルド**)を選択してください。
この操作により、CPU0→CPU1の順で、ソリューション内のすべてのプロジェクトが順次ビルドされます。

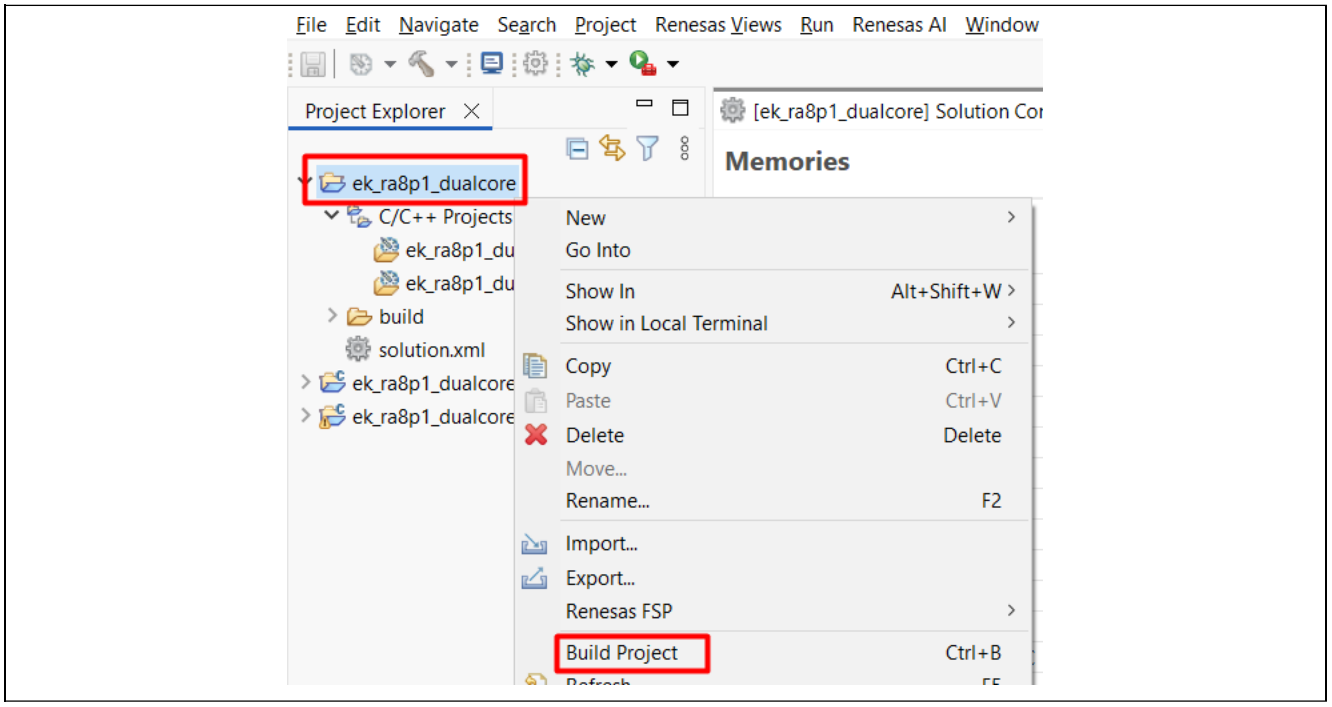


図57. RAソリューションプロジェクトデュアルコアのビルド

6.3　プロジェクトのダウンロードおよび実行

3.2章のデバッグ設定で説明したとおり、本構成ではデバイスをOEM_PL2状態で初期化する必要があり、TrustZoneの境界設定は不要です。

デュアルコアのデバッグセッションを開始するには、図58に示すようにDebug Configurationsダイアログを開きます。
次に、「ek_ra8p1_dualcore_CPU1 Debug_Multicore Launch Group」を選択し、図59に示すように**デバッグ**をクリックすると、2つ

のコアのデバッグセッションが同時に起動されます。

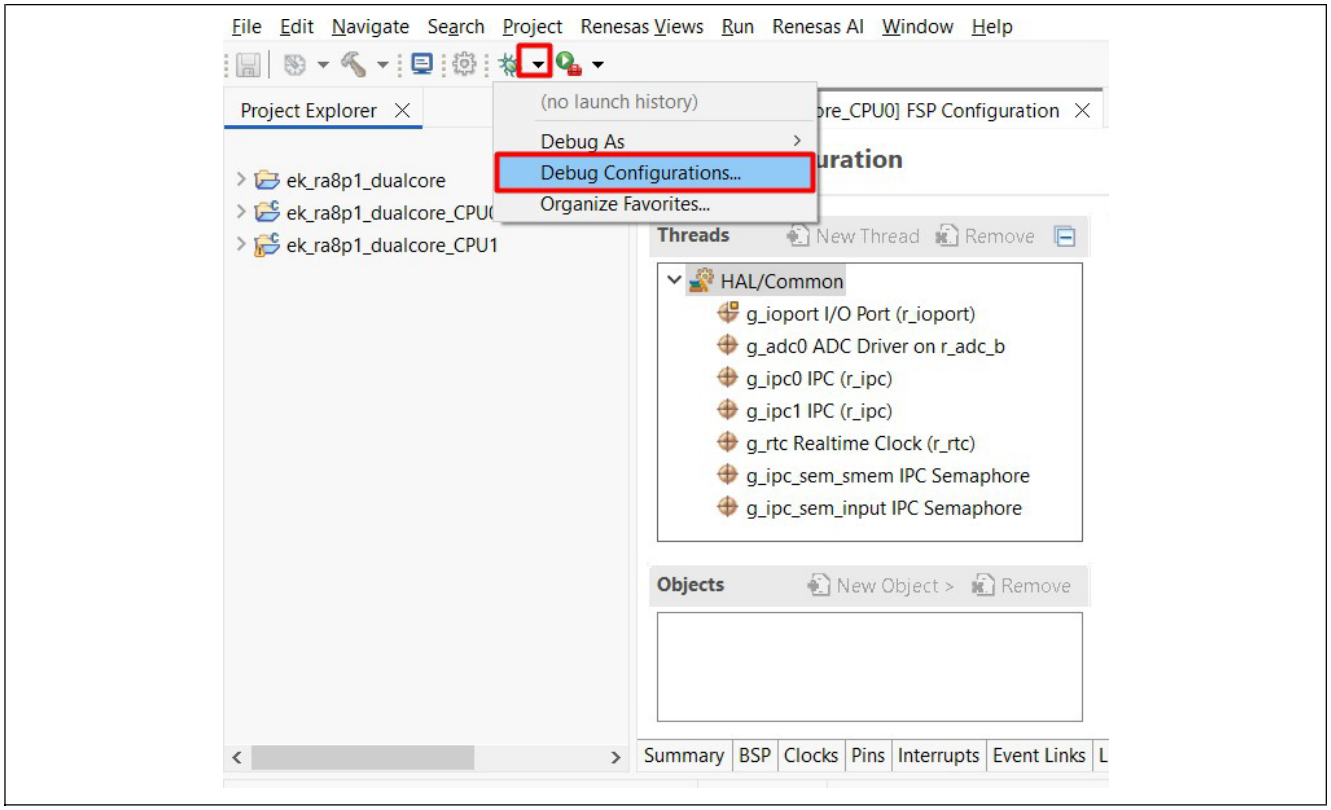


図58. ツールバーからのデバッグ設定選択

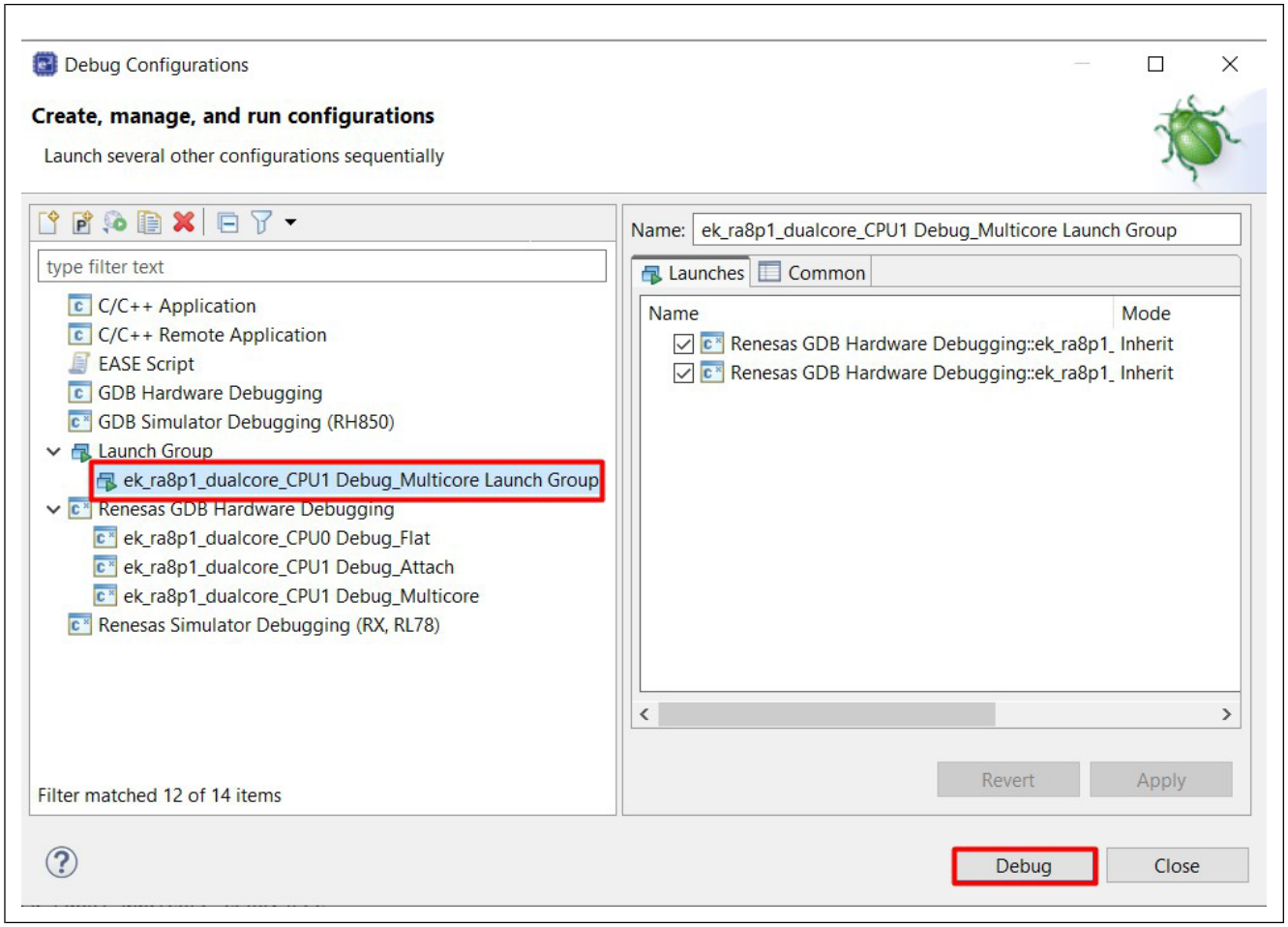


図59. Multicore Launch Groupによるデバッグ

デバッグ情報がデバッグコンソールに表示されます。再開をクリックしてアプリケーションの実行を開始する前に、アプリケーション動作を確認するためのTeraTermターミナルを設定する必要があります。

ターミナルの設定手順を以下に示します。

1. Tera Termを起動します（図60）。
2. シリアル接続オプションから、**J-Link CDC UARTポート**を選択し、ターゲットデバイスとの通信を確立します（図60）。
3. **Setup＞Terminal**を選択し、**Local Echo**にチェックを入れてローカルエコーを有効にします（図61）。
4. **Setup＞Serial Port**を選択し、ボーレートおよびその他のシリアル通信パラメータを、プロジェクトの要件に合わせて設定します（図62）。

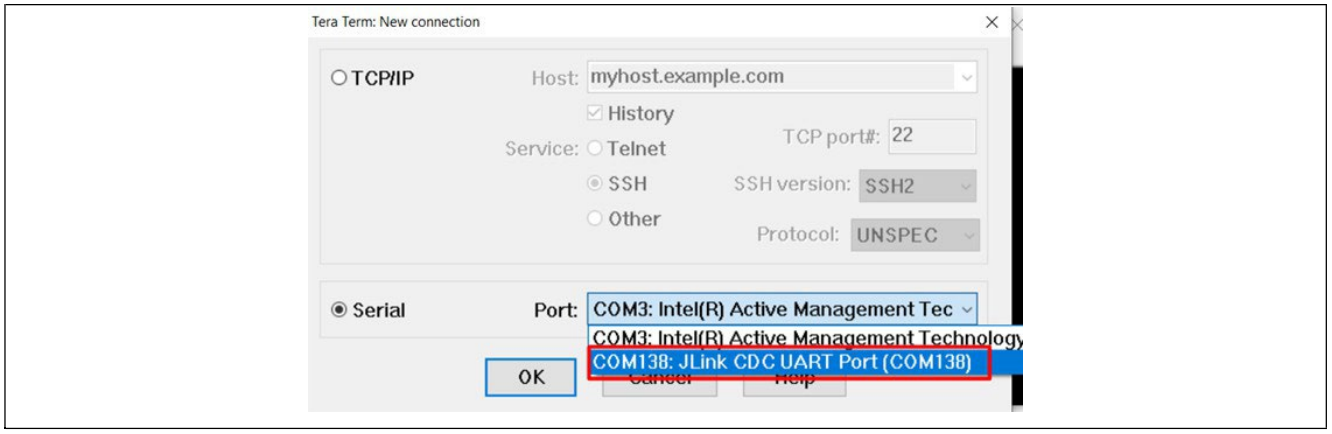


図60. Tera Termターミナルへの接続

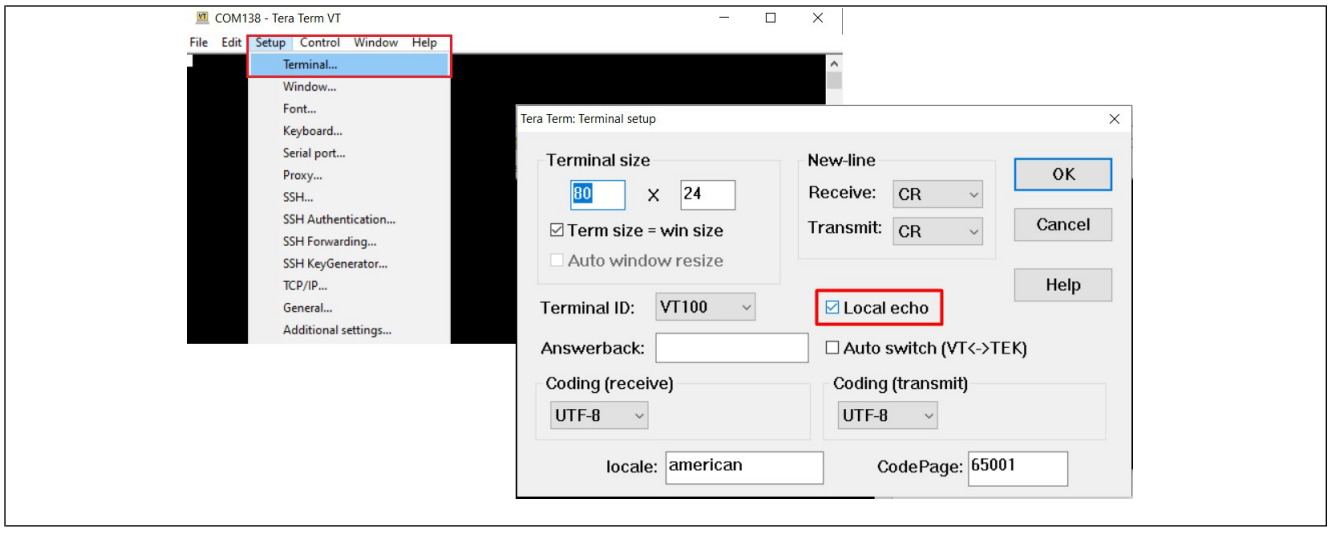


図61. Tera Termターミナルにおけるローカルエコーの有効化

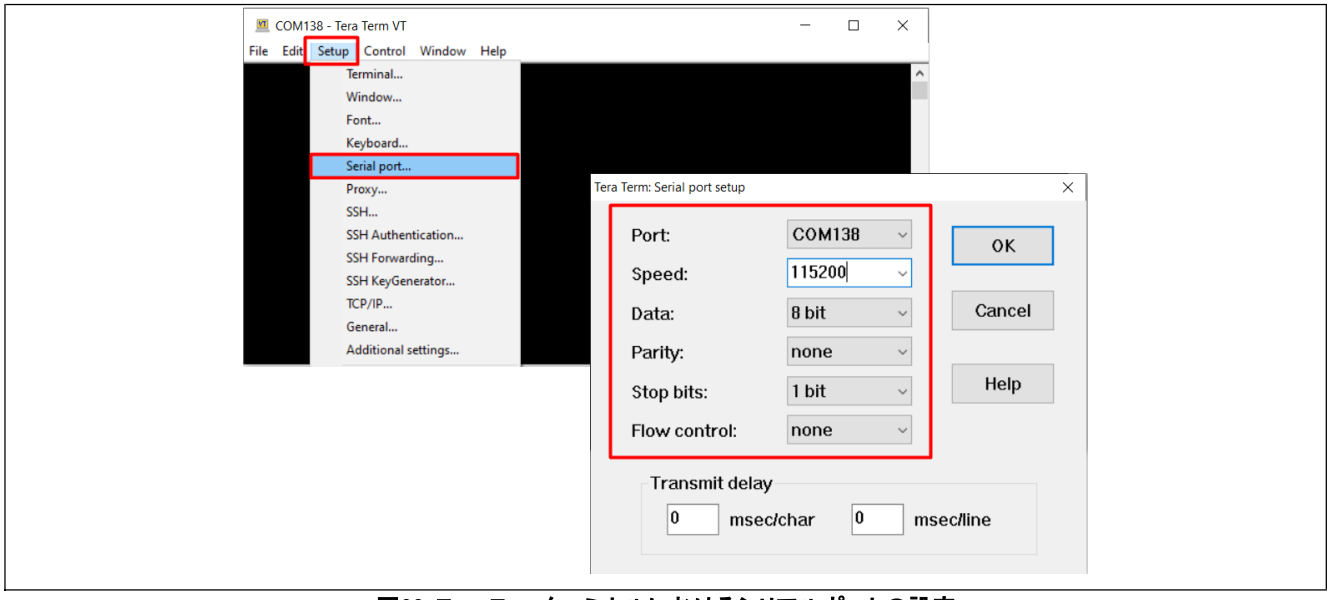


図62. Tera Termターミナルにおけるシリアルポートの設定

TeraTermターミナルの設定を完了した後、e²studioに戻り、再開を3回クリックしてアプリケーションの実行を開始します。

初期ターミナル画面を図63に示します。

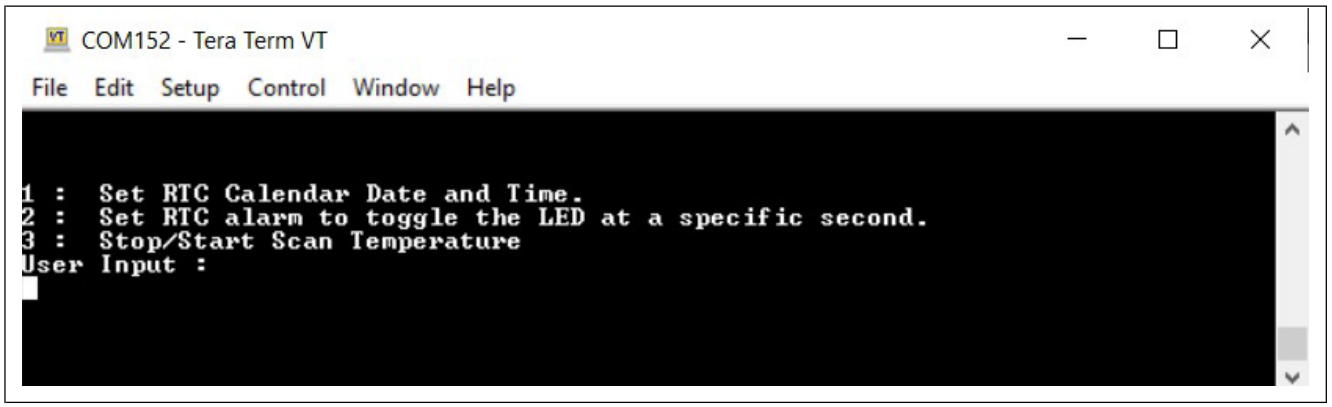


図63. アプリケーションの初期ターミナル画面

入力に応じて、ターミナルの表示内容は動的に変化します。図64に、ターミナル画面の例を示します。

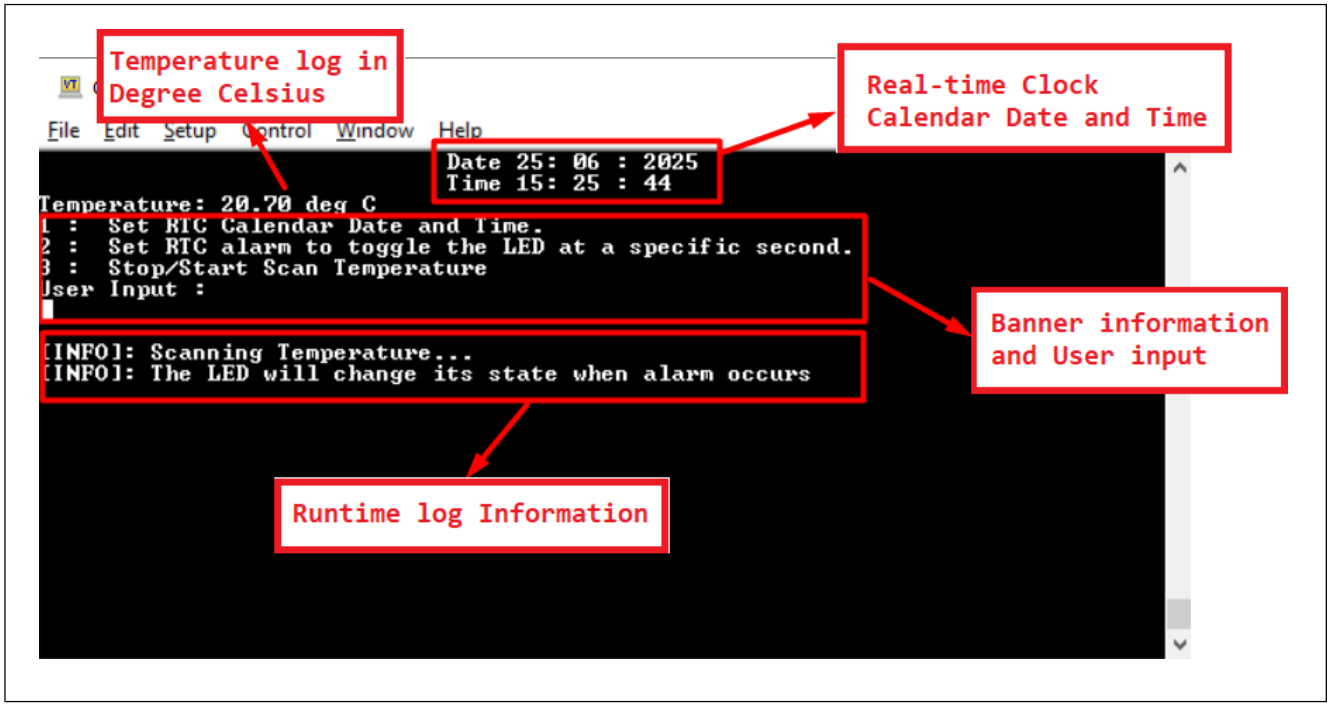


図64. アプリケーションのターミナル表示内容

注:

RTCの秒カウンタとアラーム設定の秒値が一致すると、RTCアラームが動作します。

本アプリケーションプロジェクトを操作する際は、オプション1 → オプション2 → オプション3の順で実行してください。

これは、温度表示およびRTCアラームの両方がRTCタイマによってトリガされるためです。

7. FreeRTOSベースプロジェクトの検証

7.1 プロジェクトのインポート

1. e² studio IDEを起動します。
2. Workspace launcherで任意のワークスペースを選択します。
3. Welcomeウィンドウを閉じます。
4. **メニュー**から**ファイル>インポート**を選択します。
5. インポートダイアログボックスから**既存プロジェクトをワークスペースへ**を選択します。
6. **「Developing_with_RA8_Dual_Core_MCU.zip」**に含まれるアーカイブファイル**「dsp_example_dual_core.zip」**を選択します。
7. 下図に示すように、ソリューションプロジェクトおよび各コア用に開発されたサンプルプロジェクトを選択し、**Finish（終了）**をクリックします。

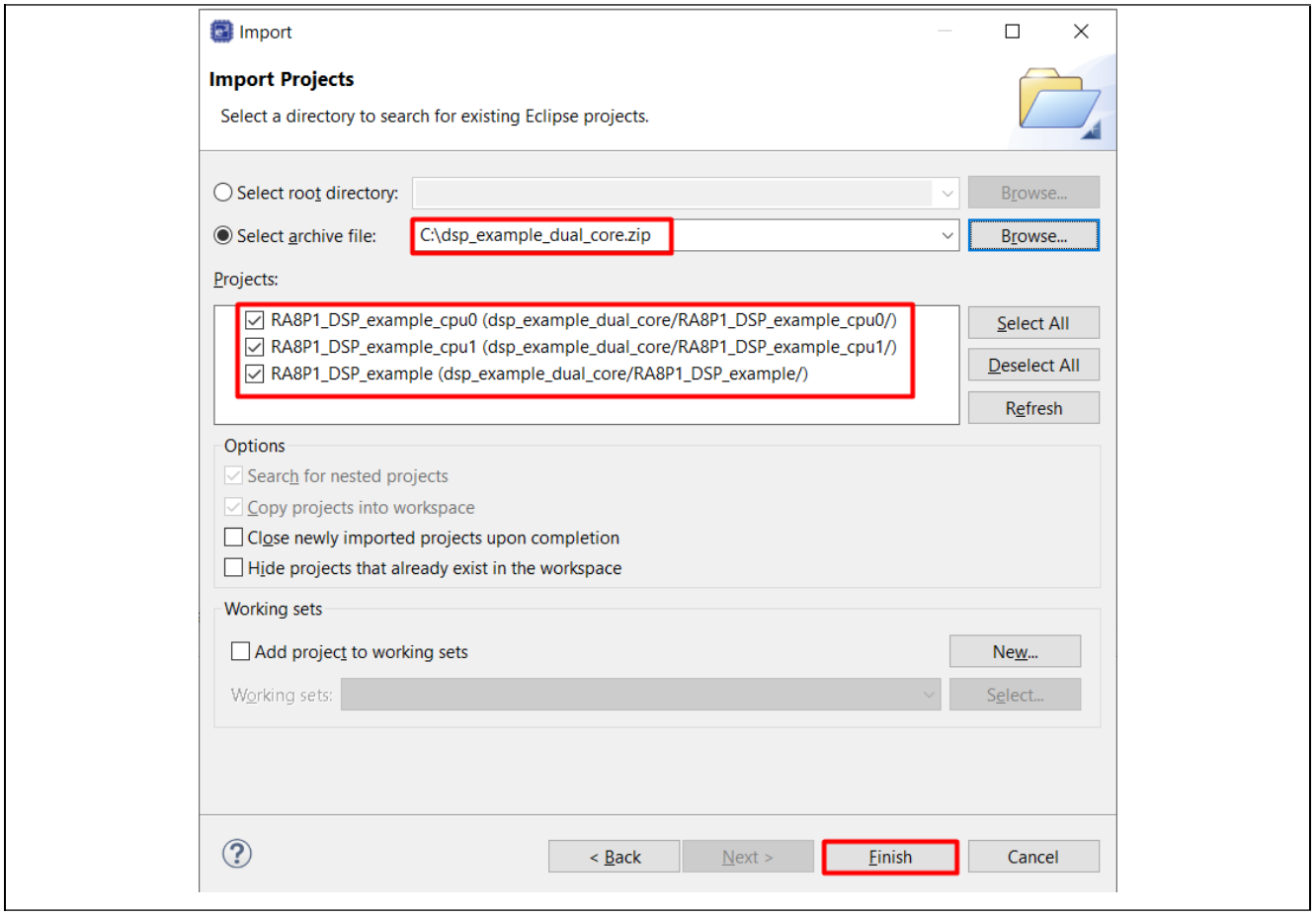


図65. DSPサンプルプロジェクトのインポート

7.2 プロジェクトのビルド

6.2章で説明した手順に従い、CPU0→CPU1の順でプロジェクトを順次ビルドします。または、図66に示すようにソリューションプロジェクトを右クリックし、Build Project（プロジェクトのビルド）を選択することで、含まれるすべてのプロジェクトを一括でビルドすることも可能です。

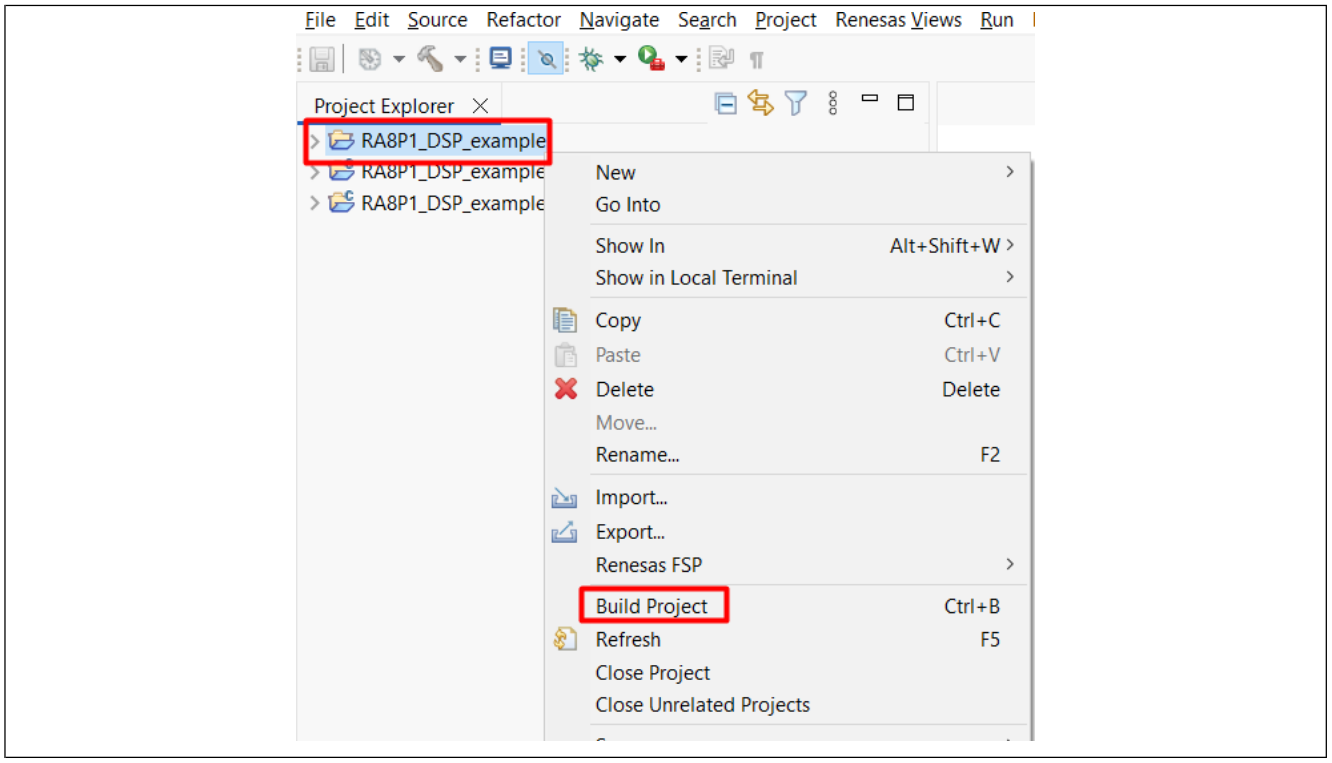


図66. DSPサンプルプロジェクトのビルド

Build Logコンソールでビルド結果を確認し、CPU0およびCPU1の両プロジェクトが正常にビルドされていることを確認してください。

7.3　プロジェクトのダウンロードおよび実行

3.2章のデバッグ設定で説明したとおり、本構成ではデバイスをOEM_PL2状態で初期化する必要があり、TrustZoneの境界設定は不要です。

2つのコアを同時にデバッグするには、以下の手順を実行します。

1. 図67に示すように**Debug Configurations**を開きます。
2. 図68に示すように**「RA8P1_DSP_example_cpu1 Debug_Multicore Launch Group.」**を選択します。
3. **デバッグ**をクリックしてデュアルコアデバッグセッションを起動します。

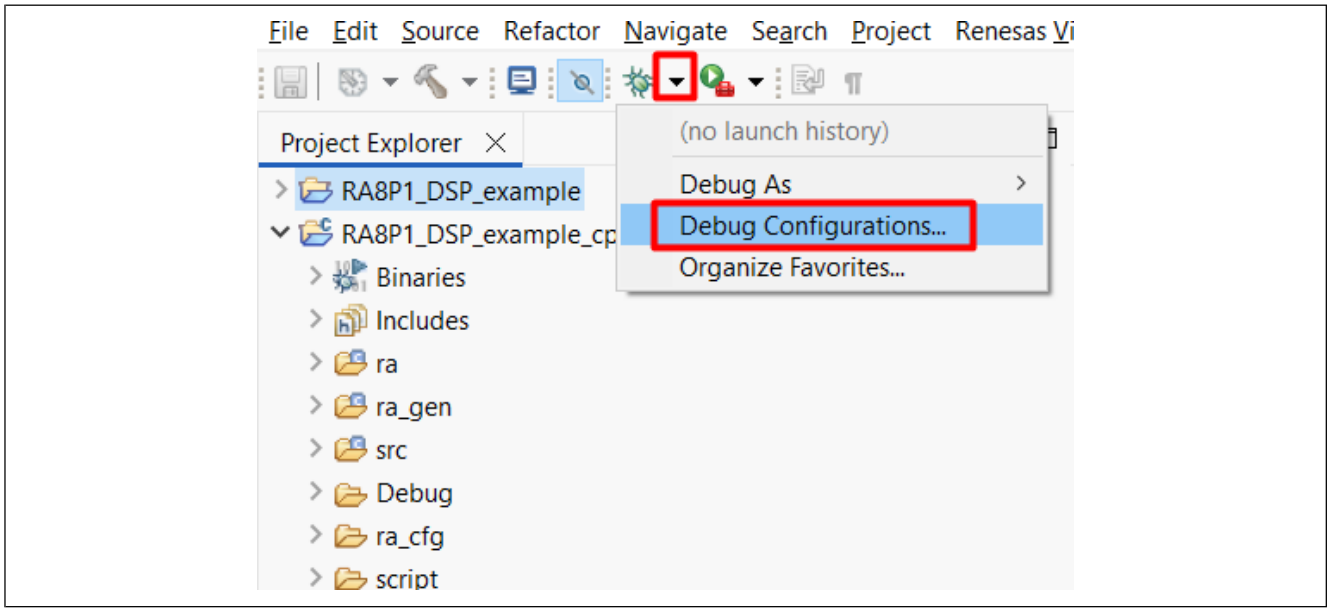


図67. DSPサンプルでのDebug Configuration起動

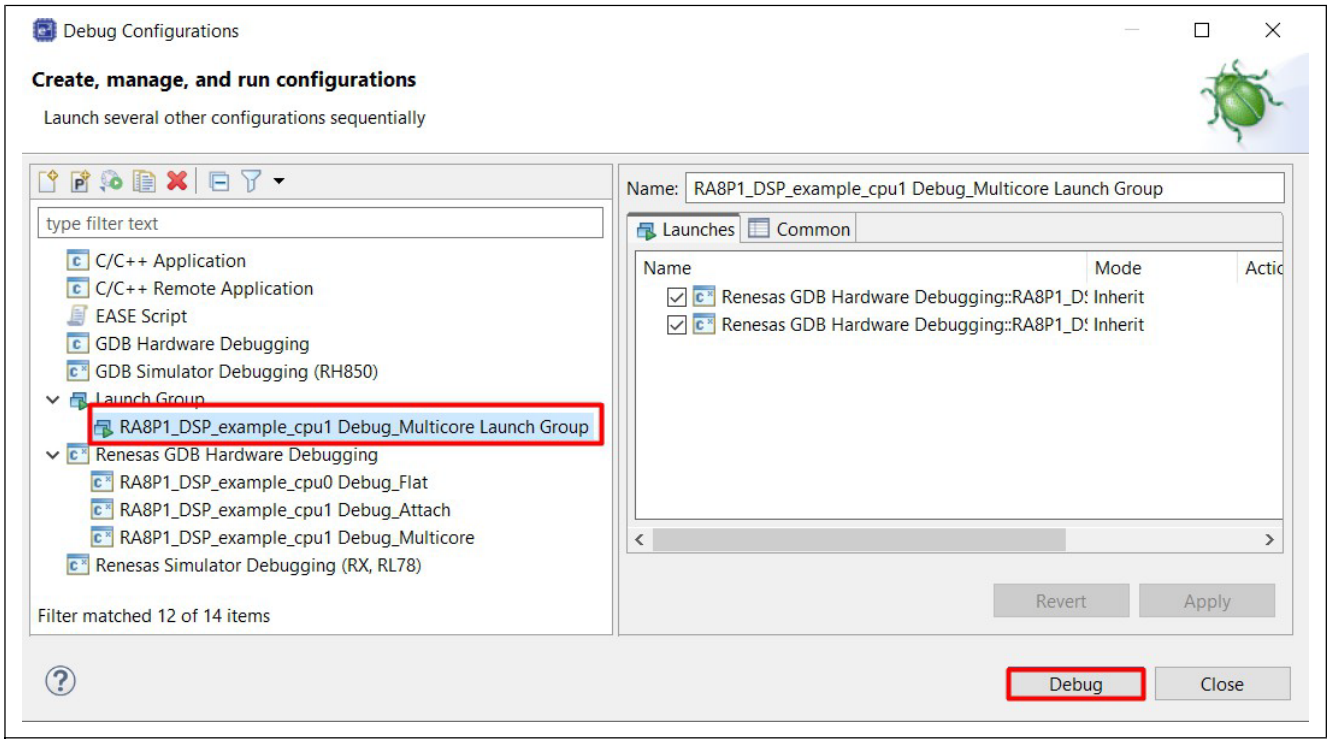


図68. DSPデュアルコアDSPサンプルのデバッグ

シリアルターミナルを起動し、以下の設定でJ-Link CDC UARTポートに接続します：
ボーレート：115200bps、データ長：8ビット、パリティ：なし、ストップビット：1、フロー制御：なし
図69および図70に、Tera Termを使用した接続および設定手順の例を示します。

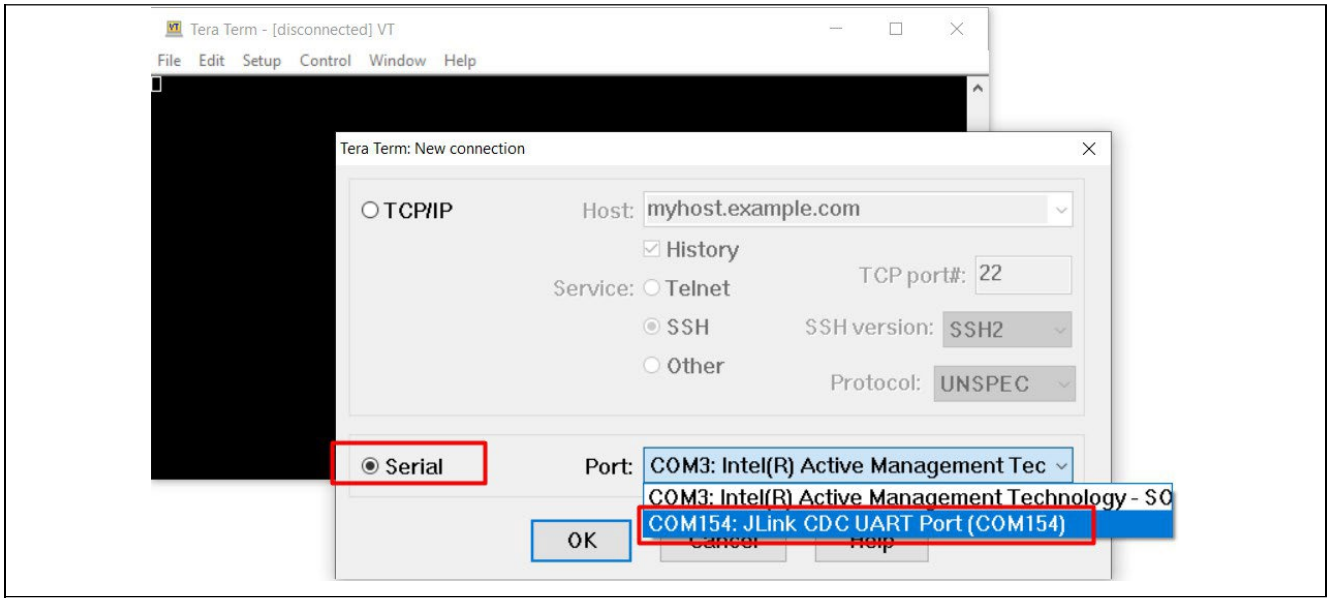


図69. JLink CDC UARTポートへの接続

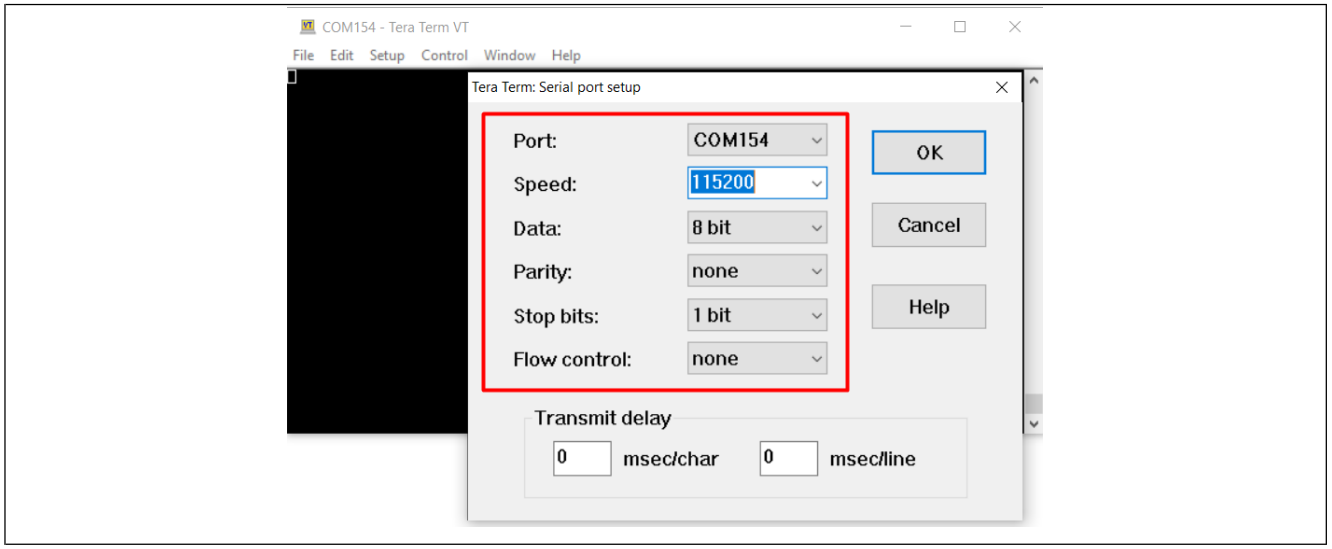


図70. Tera Termシリアルターミナルのセットアップ

e²studioに戻り、再開 を3回クリックして、両コア上でアプリケーションを実行します。実行が完了すると、図71に示すように、CMSIS-DSP FFTサンプルの実行結果がターミナルに出力されます。

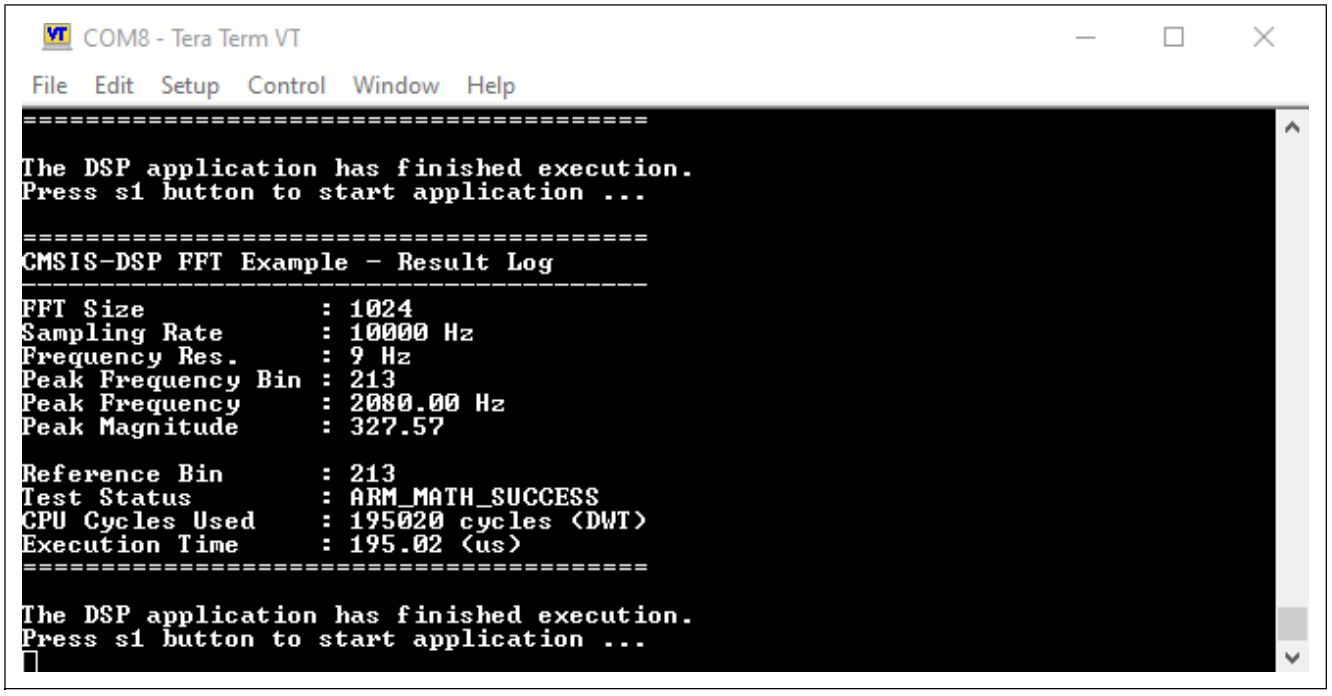


図71. CMSIS-DSP FFTサンプルの実行成功

8.　デュアルコアサンプルプロジェクトの新しいデュアルコアMCUへの移行手順

本章では、RA8P1向けに提供されているdsp_example_dual_coreアプリケーションを、RA8D2、RA8M2、RA8T2などの他のRA8デュアルコアMCUへ移行する手順について説明します。

本章では、RA8D2デバイスを例として説明しますが、記載されている内容は、後述する差分を除き、他のデュアルコアMCUにも同様に適用できます。また、本アプリケーションノートに含まれるek_ra8p1_dualcoreプロジェクトについても、同じ移行手順を適用できます。

移行プロセス全体は、以下の3つの手順に分けることができます。

・ソリューションプロジェクトの移行
対象ボードを選択して新しいソリューションプロジェクトを作成し、solution.xml内のメモリパーティション、ピン設定、クロック設定を移行先MCUに合わせて更新します。

・CPU0プロジェクトの移行
BSP設定およびFSPモジュールを調整し、CPU0上で動作する移行先プロジェクトの要件に合わせて必要なソースコード修正を行います。

・CPU1プロジェクトの移行
BSP設定およびFSPモジュールを調整し、CPU1上で動作する移行先プロジェクトの要件に合わせて必要なソースコード修正を行います。

手順1：新しいデュアルコアソリューションプロジェクトの作成

3.1章で説明した手順に従い、**Multicore Flat FreeRTOS**テンプレートを使用して、新しいソリューションプロジェクトRA8D2_DSP_exampleを作成します。

Device Selectionダイアログでは、ターゲットボードとして**EK-RA8D2**を選択し、ツールチェーンには**LLVM**を選択します（図72）。

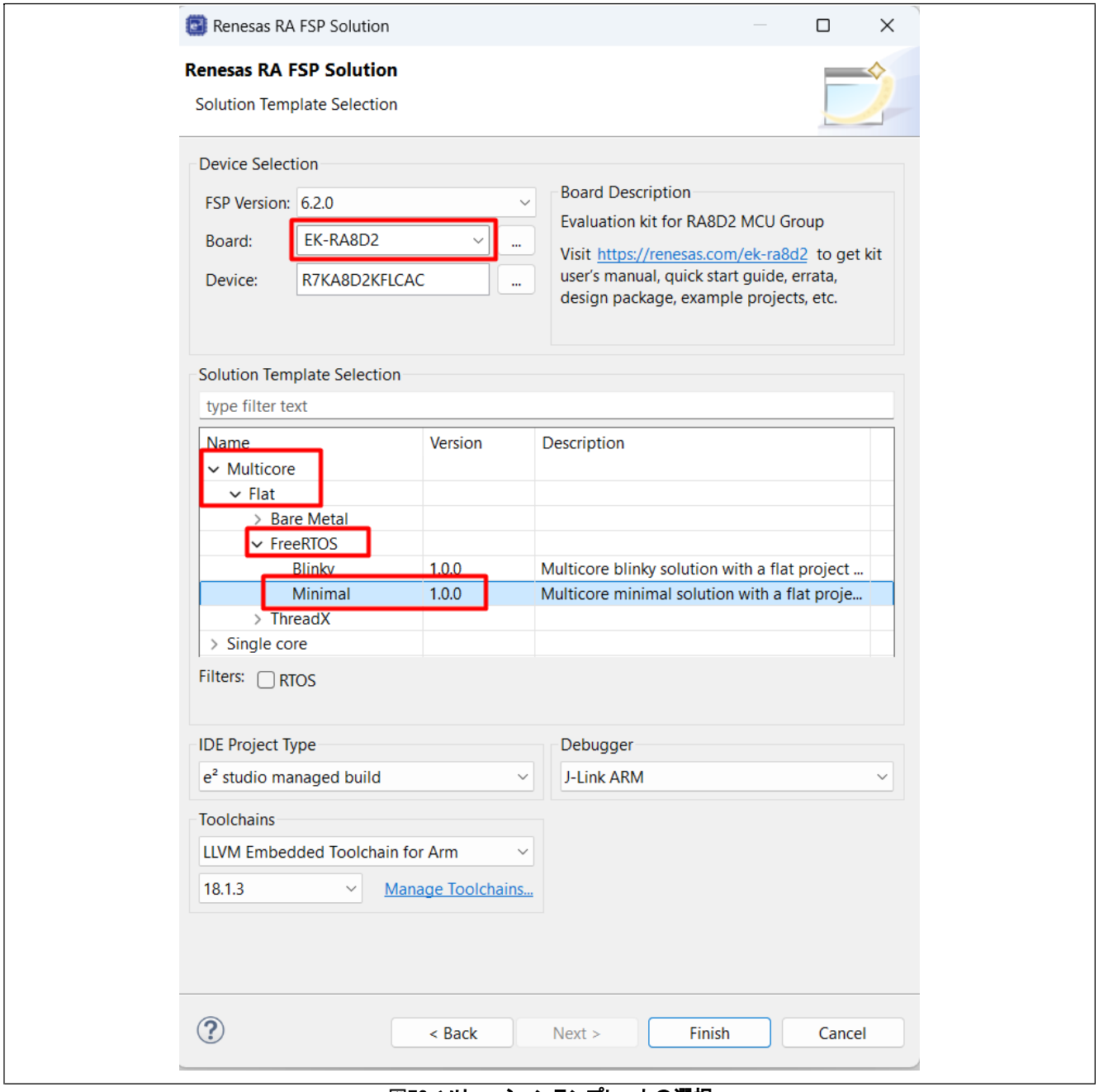


図72. ソリューションテンプレートの選択

手順2:ソリューションプロジェクトでのメモリパーティション定義、ピンおよびクロック設定

- SHARED_MEMなどのパーティションを追加します。
- SHARED_MEMパーティションに対して、Start、Size、Core、Securityを設定します。

ek_ra8p1_dualcoreをベースとした移行の場合、これら2つの設定手順は5.1.2章を参照してください。
RA8P1_DSP_exampleプロジェクトでは、メモリマップはデフォルト構成のままとなっているため、移行後のRA8D2_DSP_exampleソリューションプロジェクトでは、メモリマップ設定の変更は不要です。

- **マルチコアプロジェクト全体のクロック設定**
  RA8D2_DSP_exampleでは、CPU1上でVCOM UARTを用いたログ出力を行うため、SCICLKを120MHzに設定します。
  RA8D2_DSP_exampleプロジェクトのsolution.xmlにある**Clocksタブ**から設定を行います（図73）。使用しないクロックは無効化し、システム性能の最適化および消費電力の低減を行ってください。

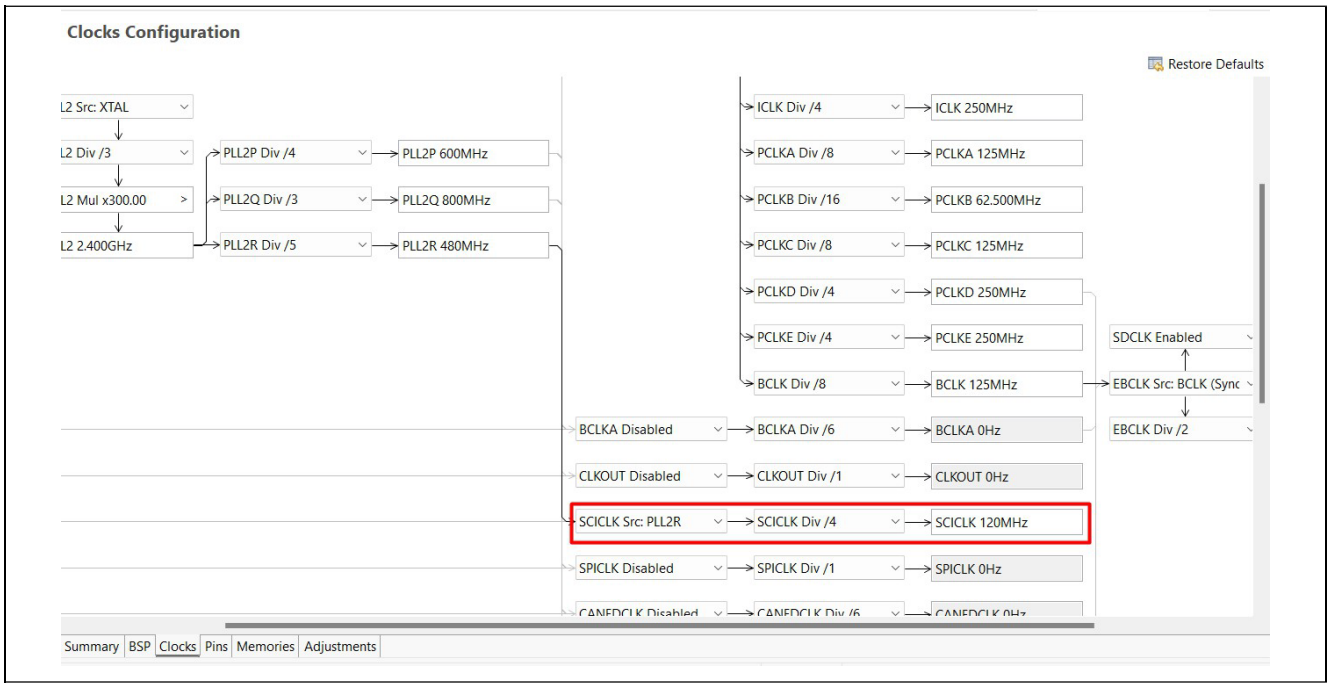


図73. ソリューションプロジェクトでのクロック設定

- **ピン設定**:
  SCI UART用のPD02（RXD8）、PD03（TXD8）および外部割り込みピンP009（IRQ13-DS）は、CPU1プロジェクトでVCOM通信およびボタン割り込み入力に使用されます。
  そのため、これらのピンはCPU1プロジェクト側で設定し、ソリューションプロジェクトおよびCPU0プロジェクトでは関連するピン設定をすべて無効化します。

  Pinsタブに移動し、**Connectivity：SCI**を展開して**SCI8**を選択し、**Operation Mode**を**Disable**に設定します。

  次に**IRQ**設定に移動し、**IRQ13-DS**を**None**に設定します。

注：ソリューションプロジェクトで行った設定変更は、RA8D2_DSP_exampleソリューションプロジェクトを右クリックして**Build Project（プロジェクトのビルド）**を実行することで、すべての関連プロジェクトに反映されることを確認してください。

手順3：CPU0プロジェクトへのFSPスタック追加および設定

RA8D2_DSP_example_CPU0では、IPCの設定およびIPCマスクカブル割り込みの設定を行います。
プロセッサ間通信（IPC）は、2つのコア間での迅速なメッセージ交換を実現するため、最優先度で動作する専用IPCスレッド内で処理されます。また、IPC処理中にはミューテックスも使用されます。

RA8D2_DSP_example_CPU0/configuration.xmlの**Stacksタブ**で**New Thread**をクリックし、**Ipc Thread**という新しいスレッドを作成します。スレッドの設定例を図74に示します。

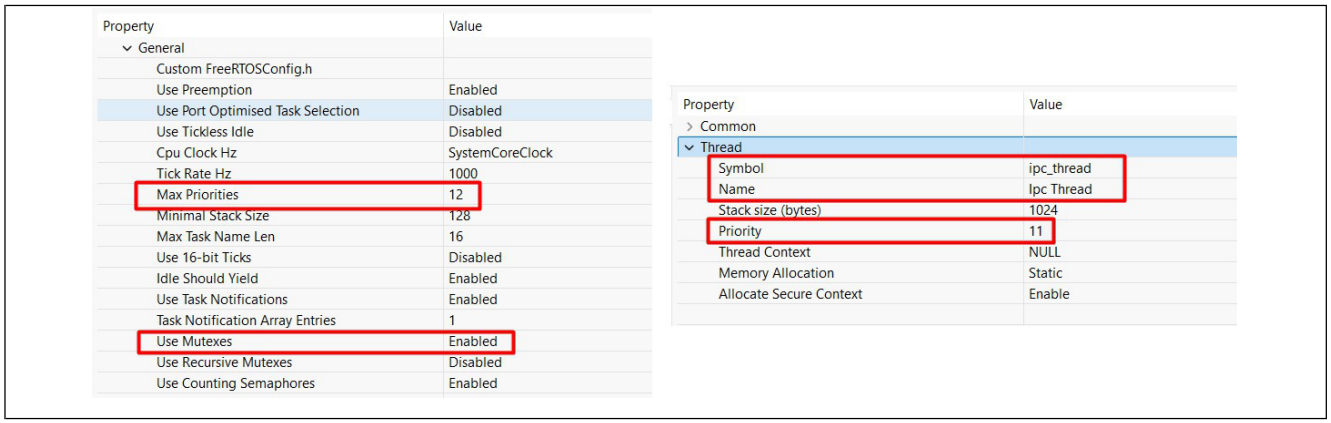


図74. RA8D2_DSP_example_CPU0におけるIpc Threadの設定

New Stack > System > IPC (r_ipc)を選択して、Ipc Threadの下に2つのIPCスタックを追加します。IPCの設定例を図75に示します。

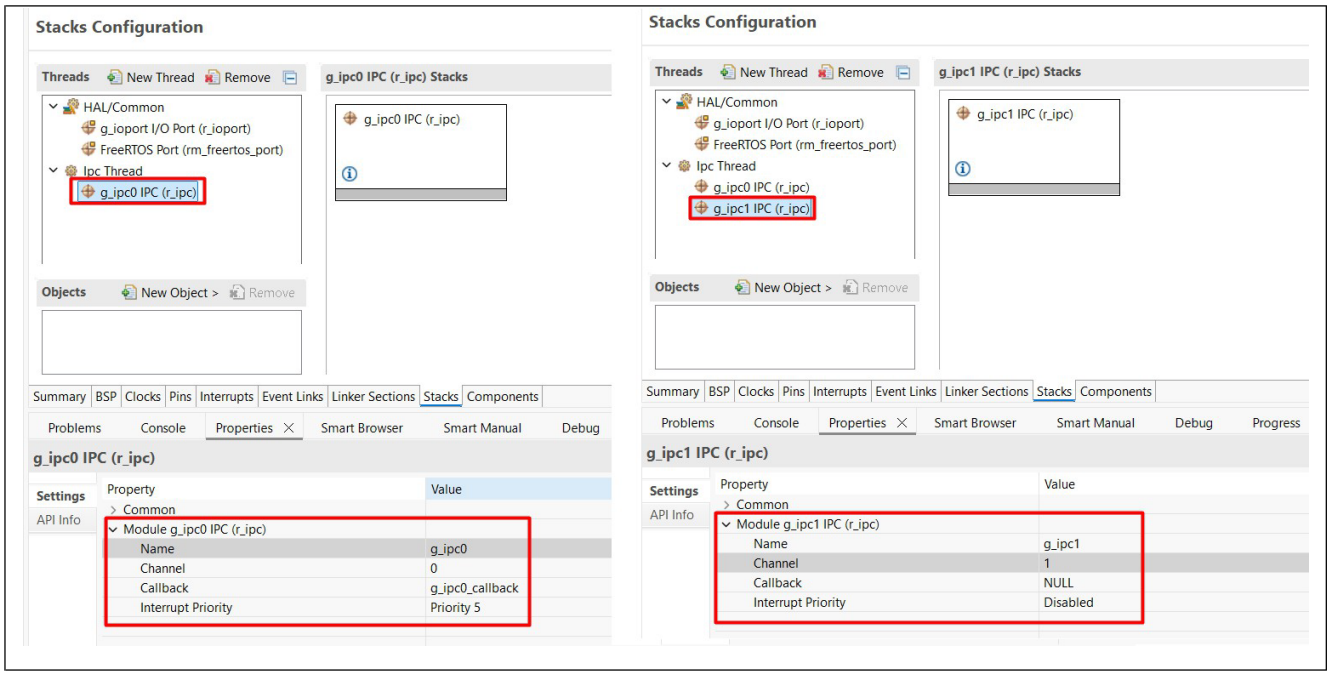


図75. RA8D2_DSP_example_CPU0におけるIPCの設定

New Stack > System > IPC Semaphoreをクリックして、CPU0プロジェクトのIpc Threadの下にハードウェアセマフォ0と1を追加します。IPCセマフォの設定を図76に示します。

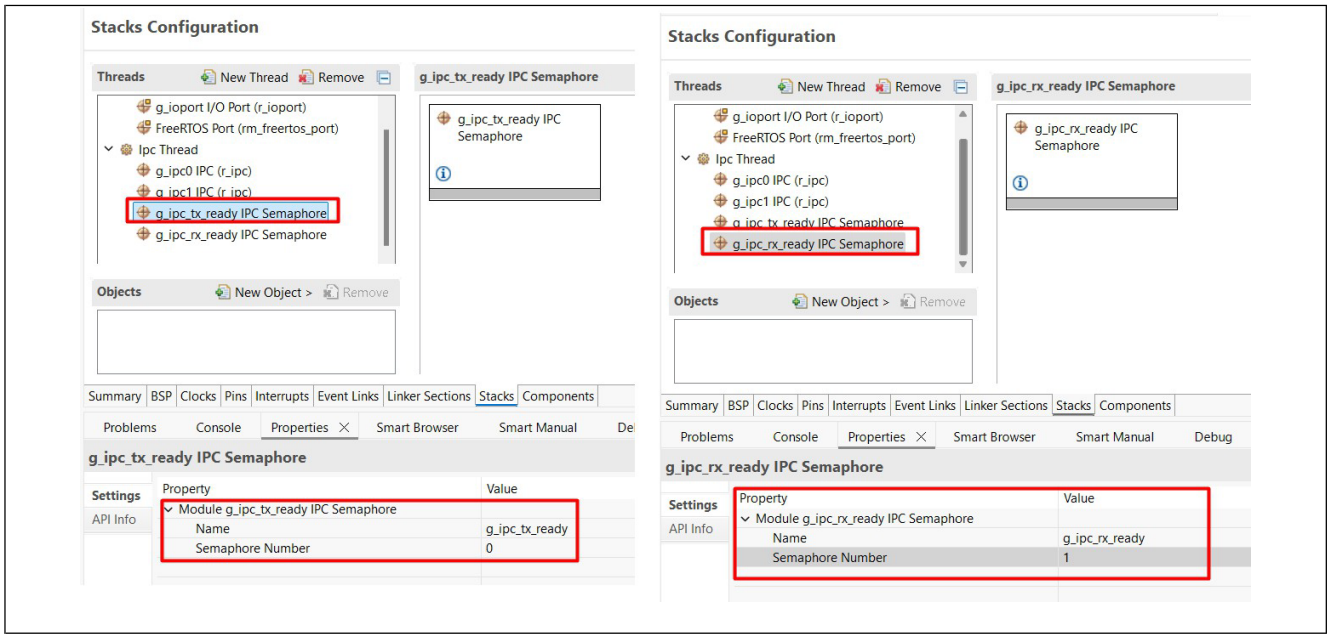


図76. RA8D2_DSP_example_CPU0プロジェクトにおけるハードウェアセマフォの設定

RA8D2_DSP_example_CPU0では、Arm DSPサンプルは別のスレッドで実行されます。
RA8D2_DSP_example_CPU0/configuration.xmlを開き、**New Thread**をクリックして、**DSP Thread**という名前のスレッドを作成します。
このスレッドの設定を図77に示します。

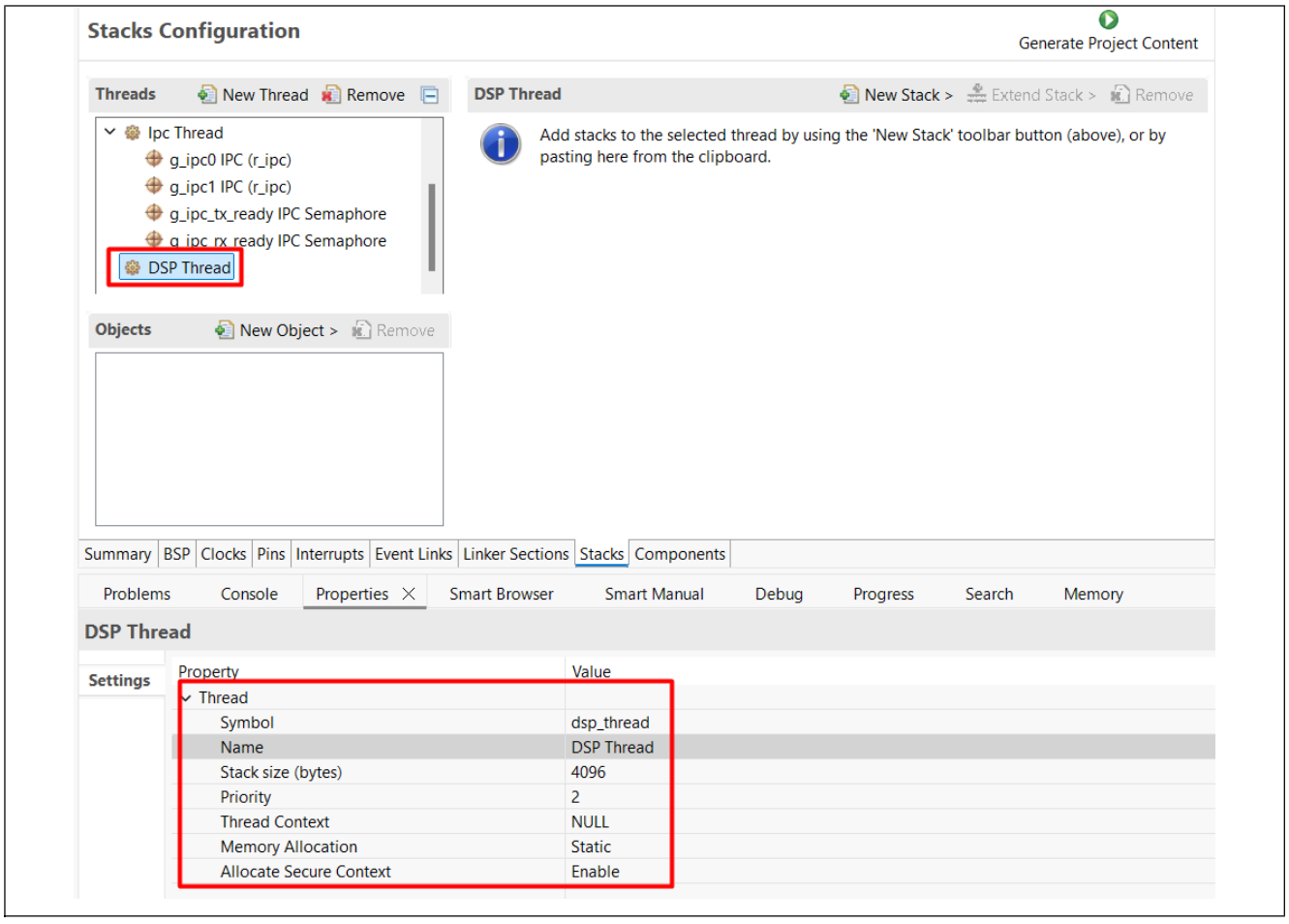


図77. RA8D2_DSP_example_CPU0プロジェクトにおけるDSP Threadの追加

次に、**DSP Thread**配下にRA8D2_DSP_example_CPU0プロジェクト用のArm DSPライブラリを追加します。
Stacksタブで**New Stack＞DSP＞Arm CMSIS DSP Library Source**を選択してください。
RA8D2_DSP_example_CPU0におけるFSPスタックの最終構成を図78に示します。

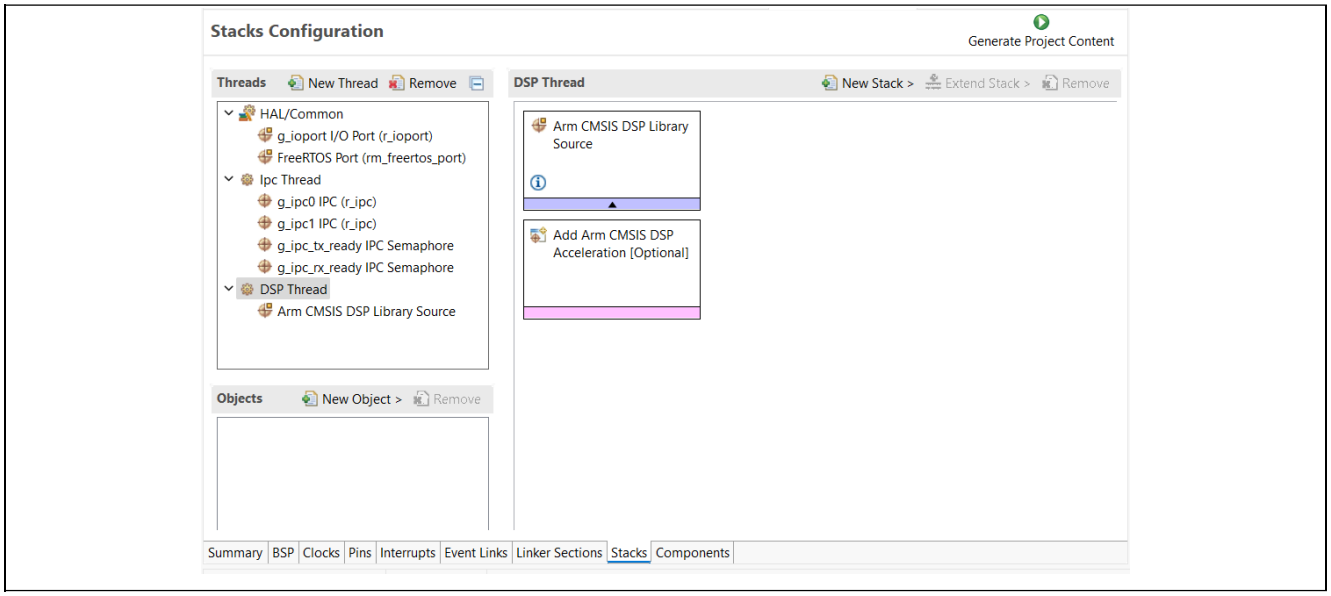


図78. RA8D2_DSP_example_CPU0プロジェクトにおけるFSPのスタック構成

手順4：RA8D2_DSP_example_CPU0プロジェクトでのピン設定

手順1で述べたように、SCI UARTチャネル8ピン（P02、P03）と外部割込みピン（P009）も、以下のようにCPU0プロジェクトで無効にする必要があります。

1. **Pinsタブ**で**Connectivity：SCI**を展開し、**SCI8**を選択して**Operation Mode**を**Disable**に設定します。
2. **IRQ**設定で**IRQ13-DS**を**None**に設定します。

手順5：arm_cmplx_mag_f32関数をITCMに、testInput_f32_10khzバッファをDTCMに配置

4.4.1章を参照してarm_cmplx_mag_f32関数をITCMに配置し、4.4.2章を参照してtestInput_f32_10khzバッファをDTCMに配置します。
設定完了例を図79に示します。

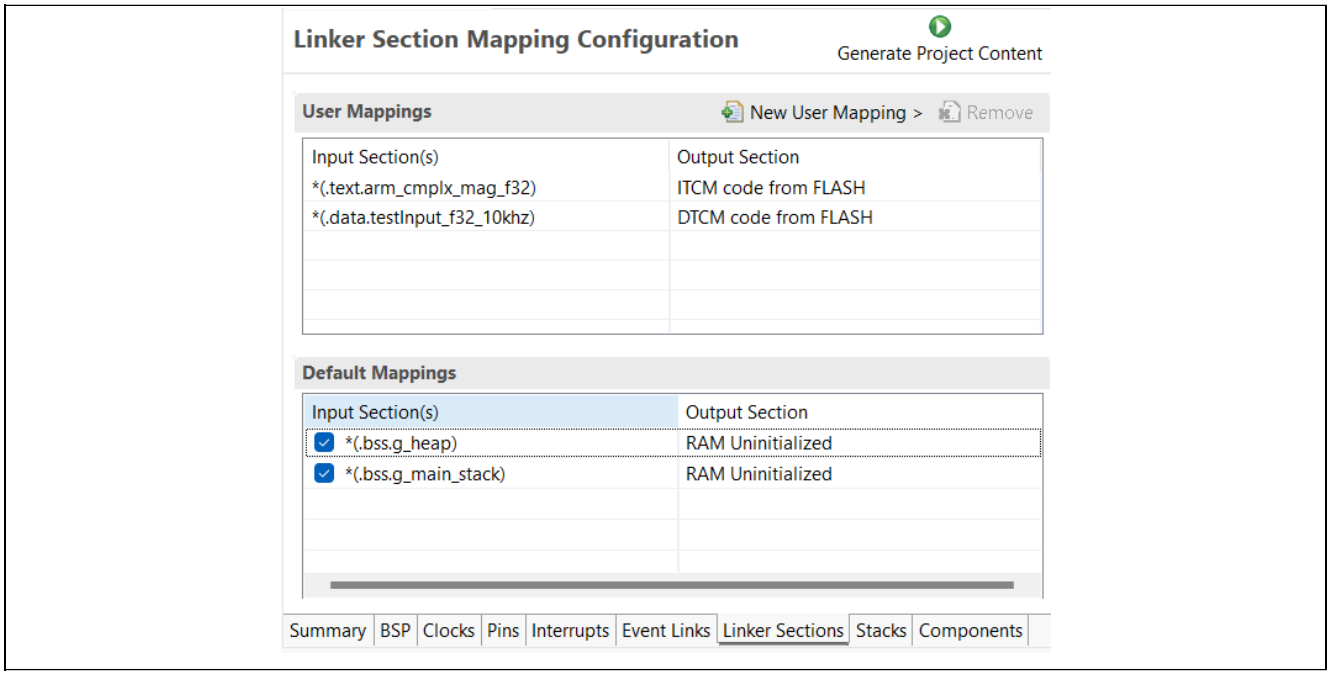


図79. DTCMおよびITCMへの関数・データ配置の完了

手順6：RA8D2_DSP_example_CPU0プロジェクトでのGenerate Project Content実行

RA8D2_DSP_example_CPU0/configuration.xmlで**Generate Project Content**をクリックし、CPU0プロジェクトのFSP設定を反映します。

手順7：CPU0固有のタスクを実装

RA8P1_DSP_example_cpu0プロジェクトの/srcフォルダ内に実装されているアプリケーションソースは、RA8D2_DSP_example_CPU0プロジェクトと互換性があります。
そのため、RA8D1_DSP_example_cpu0/srcからRA8D2_DSP_example_CPU0/srcへ、すべての内容をコピーするだけで対応できます。

手順8：CPU0プロジェクトのビルド

RA8D2_DSP_example_CPU0プロジェクトを右クリックし、**Build Project（プロジェクトのビルド）**を選択します。

手順9： CPU1プロジェクトへのFSPスタック追加

RA8P1_DSP_example_cpu1プロジェクトのFSP設定をRA8D2_DSP_example_CPU1プロジェクトへ適用します。
手順3で行った手動設定とは異なり、ここではe²studioが提供する**Import／Export FSP Stacks**機能を使用します。
この機能を利用することで、既存プロジェクトを別のターゲットMCUへ移行する際の作業を簡素化できます。
RA8P1_DSP_example_cpu1プロジェクトから設定をエクスポートします。

1. RA8P1_DSP_example_cpu1プロジェクトの**configuration.xml**を開きます。
2. **Ipc Thread**配下の各モジュールを右クリックし、図80に示すように**Export**を選択します。
3. エクスポートした設定を新しい.xmlファイル（例：ipc_thread_config.xml）として保存します。この際、**Include Common Properties**にチェックを入れ、図81に示すように**Finish（終了）**をクリックします。

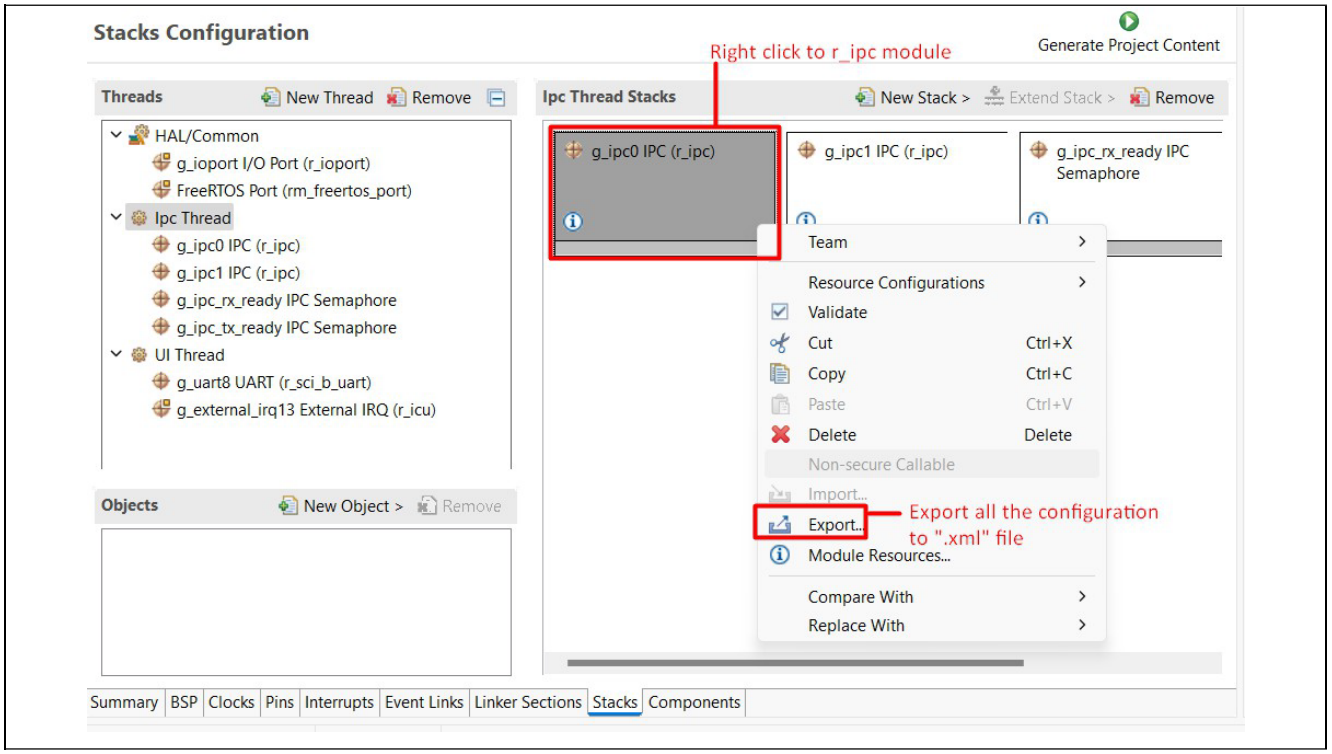


図80. RA8P1_DSP_example_cpu1でのFSPスタックのエクスポート

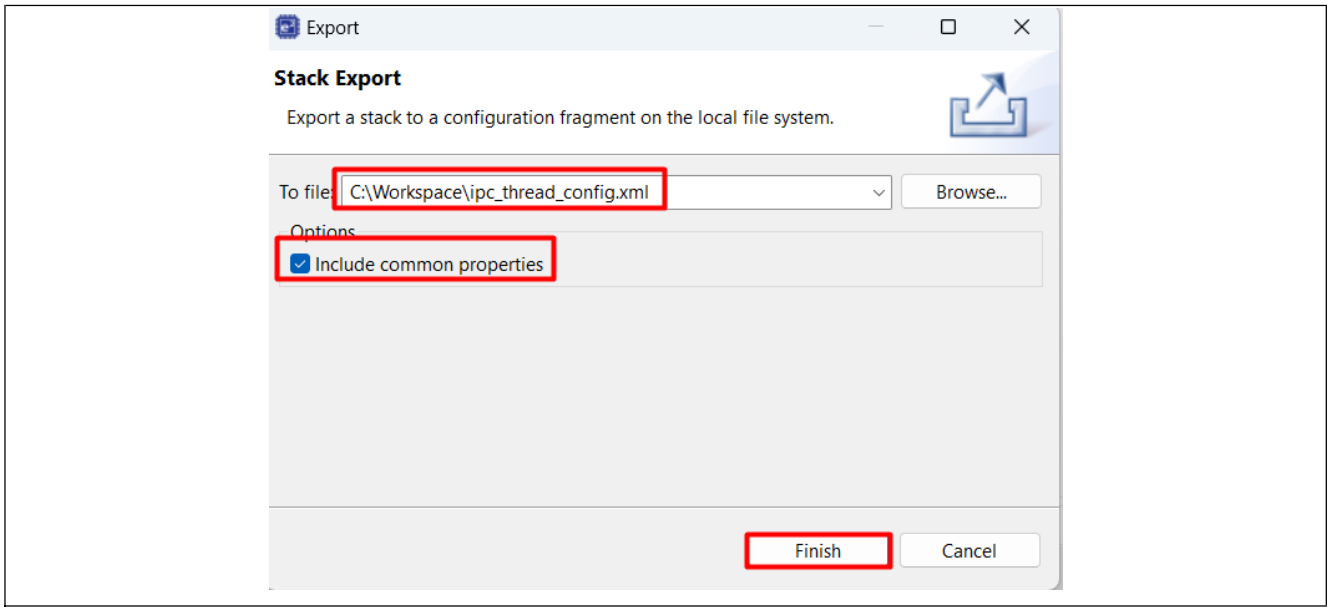


図81. Stack Exportダイアログにおける設定

RA8P1_DSP_example_cpu1プロジェクトで使用されているすべての**FSPスタック**について、同様のエクスポート作業を繰り返します。
プロジェクトの移植や他のMCUへのマイグレーション時に設定管理や再インポートを容易にするため、各スレッドに**FSPスタック**を個別にエクスポートすることを推奨します。
本例では、**Ipc Thread**配下のスタックをipc_thread_config.xmlに、**UI Thread**配下のスタックをui_thread_config.xmlにそれぞれ保存します。
RA8P1_DSP_example_cpu1プロジェクトからすべてのFSPスタックを正常にエクスポートしたら、次の手順に進み、RA8D2_DSP_example_CPU1プロジェクトへFSPスタックをインポートします。

手順10：RA8D2_DSP_example_CPU1プロジェクトへのFSPスタックのインポート

RA8D2_DSP_example_CPU1プロジェクトのconfiguration.xmlをダブルクリックし、**Stacksタブ**に移動します。次に、図82に示すGeneral設定において、**Ipc Thread**および**UI Thread**の2つのスレッドを手動で作成します。各スレッドの詳細設定例を図83に示します。

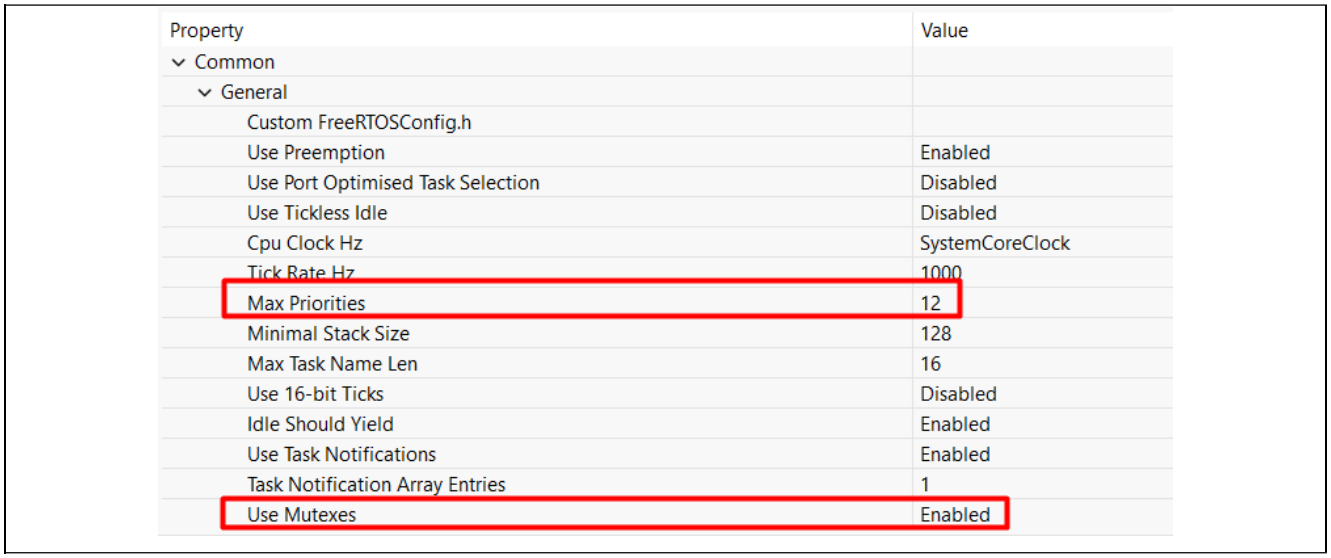


図82. RA8D2_DSP_example_CPU1プロジェクトにおけるFreeRTOS Threadの設定

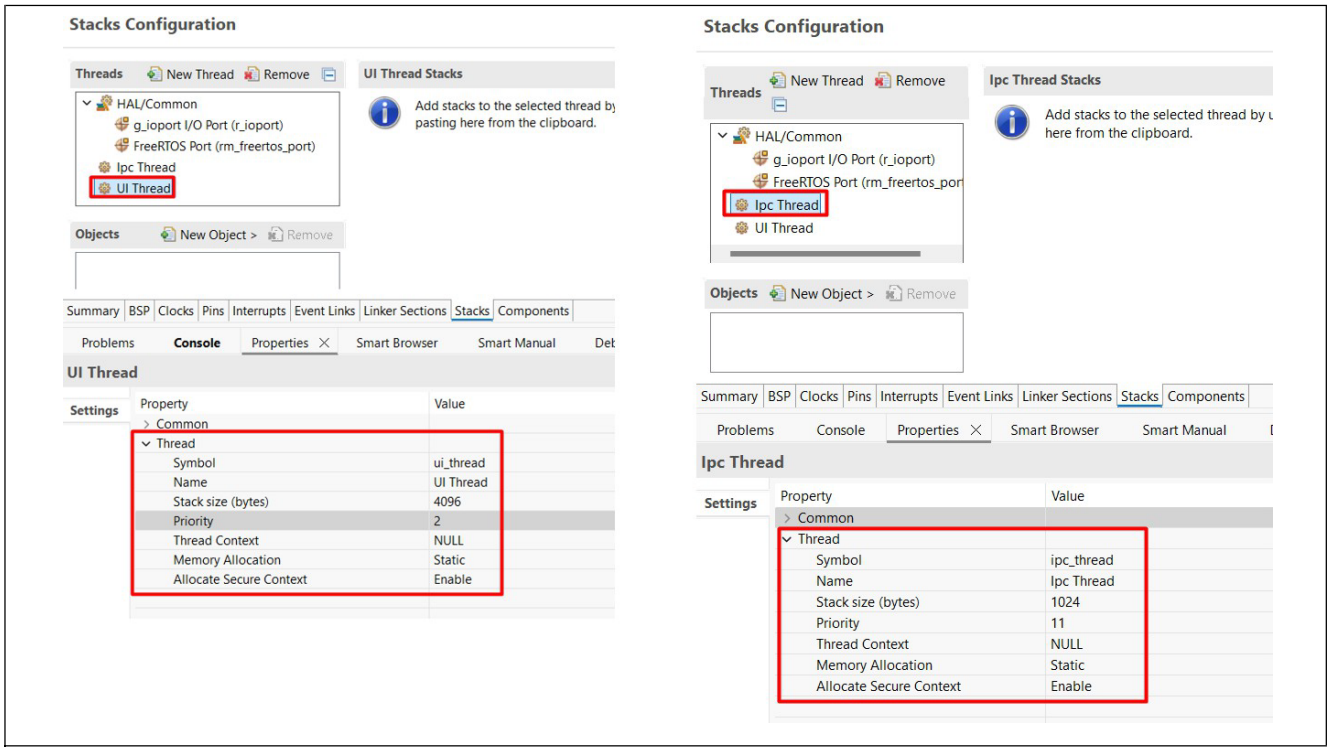


図83. RA8D2_DSP_example_CPU1プロジェクトにおけるThreadの設定

RA8P1_DSP_example_cpu1プロジェクトからエクスポートしたFSPスタックをインポートします。

1. **Ipc Thread Stacks**を右クリックし、**Import**を選択します。
2. **Stack Import**ダイアログで、**手順9**でエクスポートしたipc_thread_config.xmlを選択し、**Select All**をクリックします。
   Include Common Propertiesにチェックを入れ、**Finish（終了）**をクリックしてインポートを完了します。

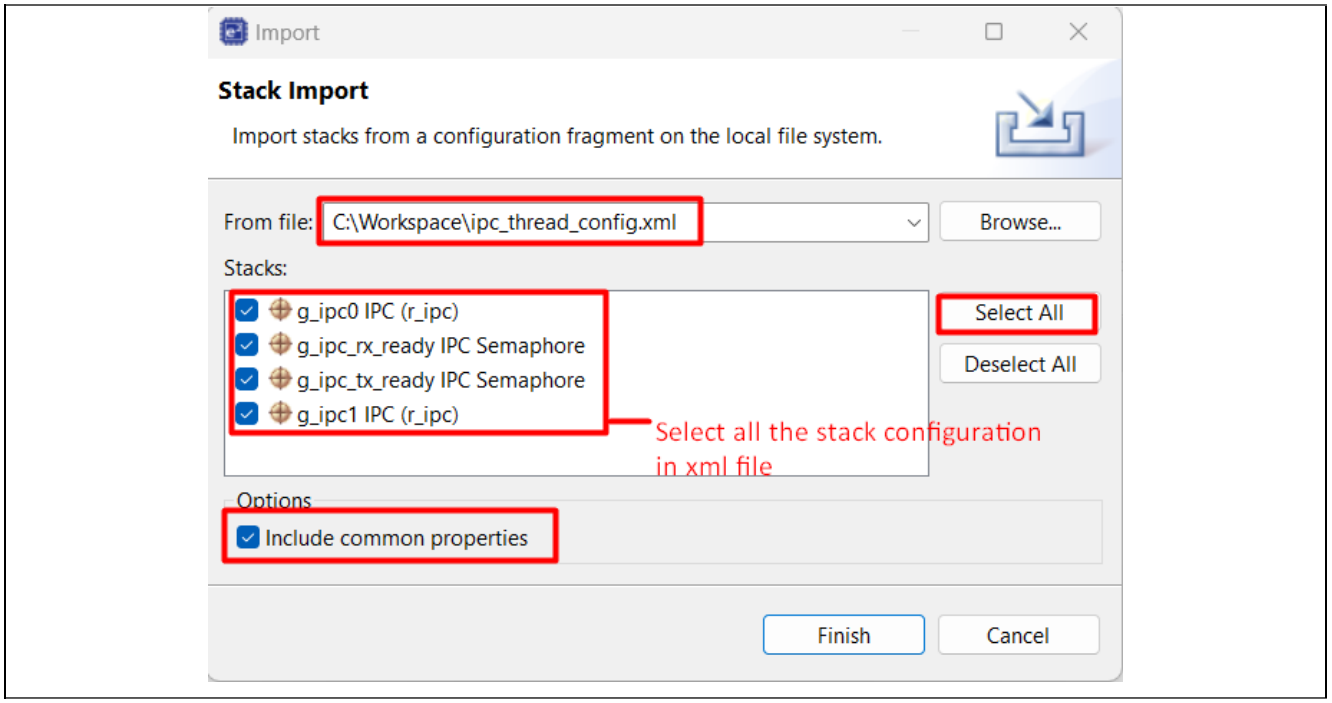


図84. RA8D2_DSP_example_CPU1プロジェクトにおけるIpc Thread用スタックのインポート

同様に、**UI Thread**配下のFSP設定についてもインポートを行います。

1. **UI Thread Stacks**を右クリックし、**Import**を選択します。
2. **Stack Import**ダイアログにおいて、手順9でエクスポートしたui_thread_config.xmlを選択し、**Select All**をクリックします。
   Include Common Propertiesにチェックを入れ、**Finish（終了）**をクリックしてインポートを完了します。

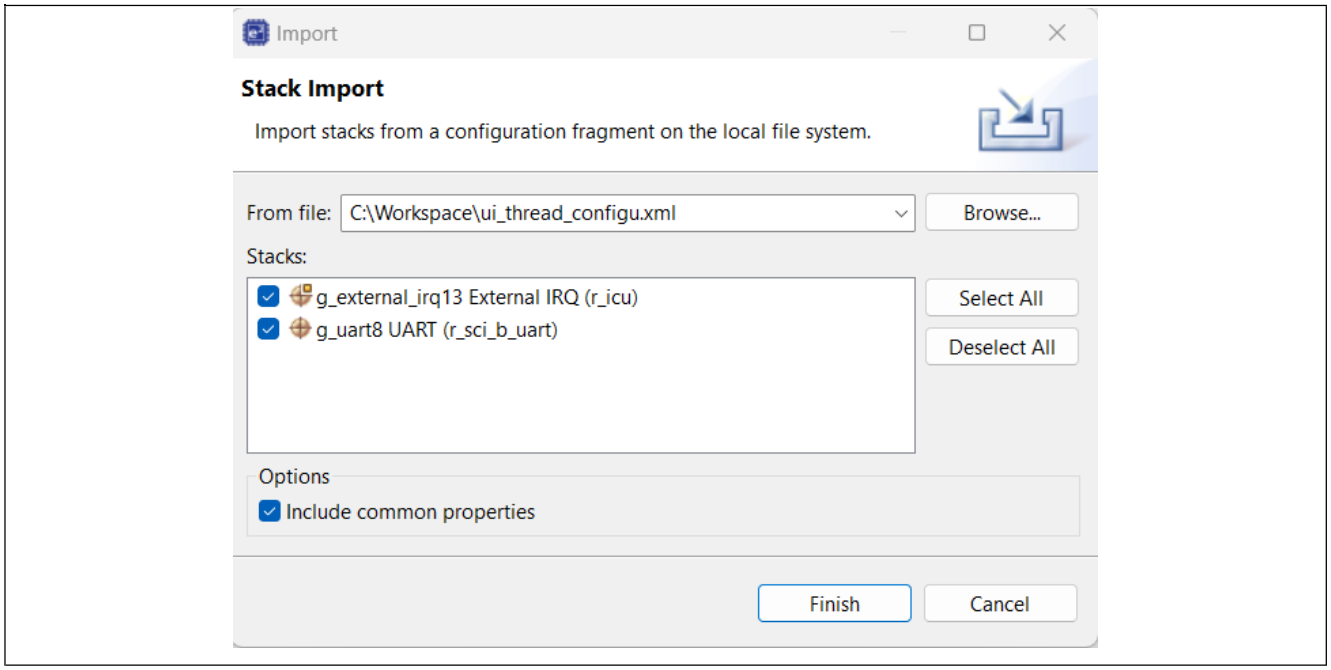


図85.RA8D2_DSP_example_CPU1プロジェクトにおけるUI Thread用スタックのインポートダイアログ

手順11：RA8D2_DSP_example_CPU1プロジェクトにおけるピン設定

RA8D2_DSP_example_CPU1プロジェクトにおけるFSP Configuratorの**Pinsタブ**に移動し、SCI UARTチャネル8のピン（PD02、PD03）および外部割り込みピン（P009）が有効になっていることを確認します。設定例を図86および図87に示します。

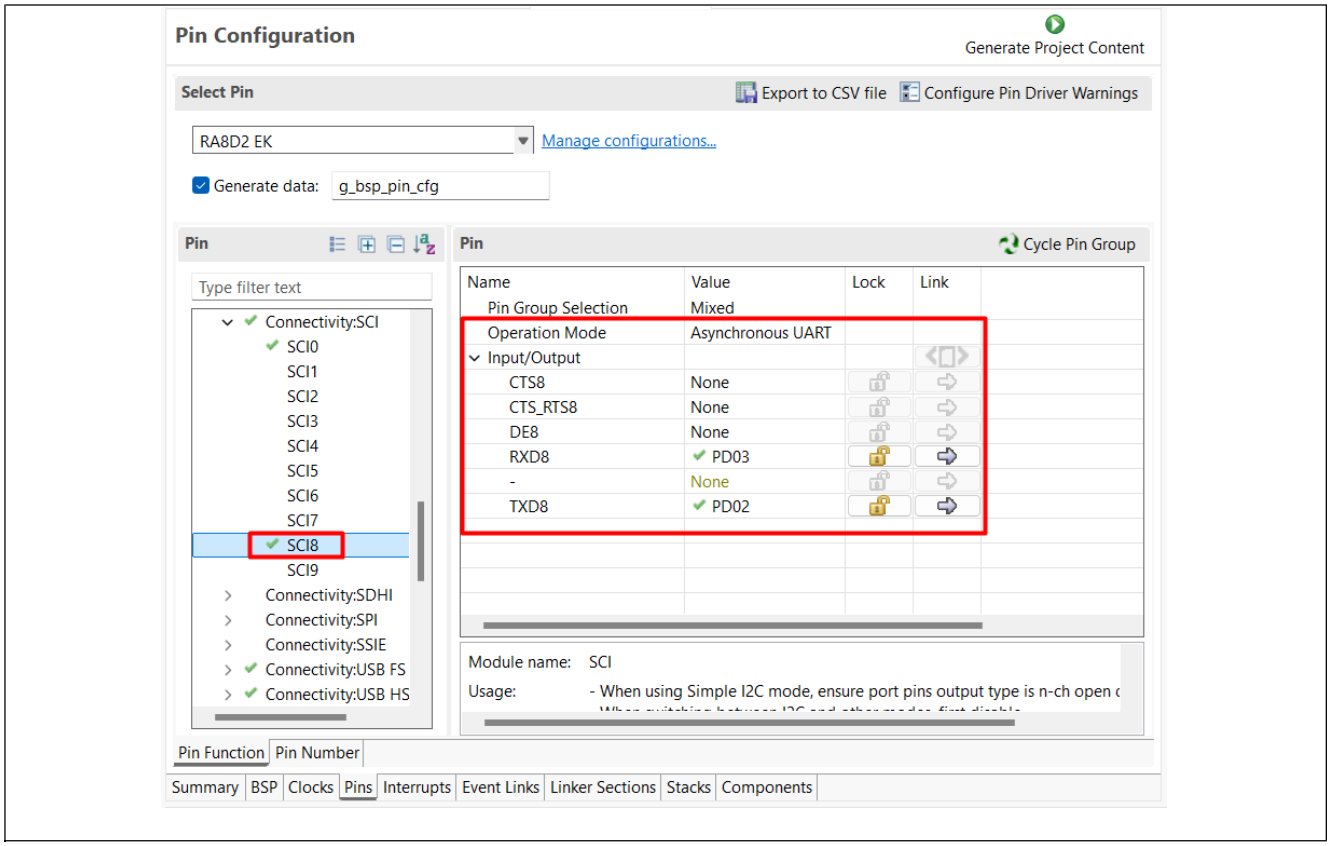


図86.RA8D2_DSP_example_CPU1プロジェクトにおけるSCI UARTピンの設定

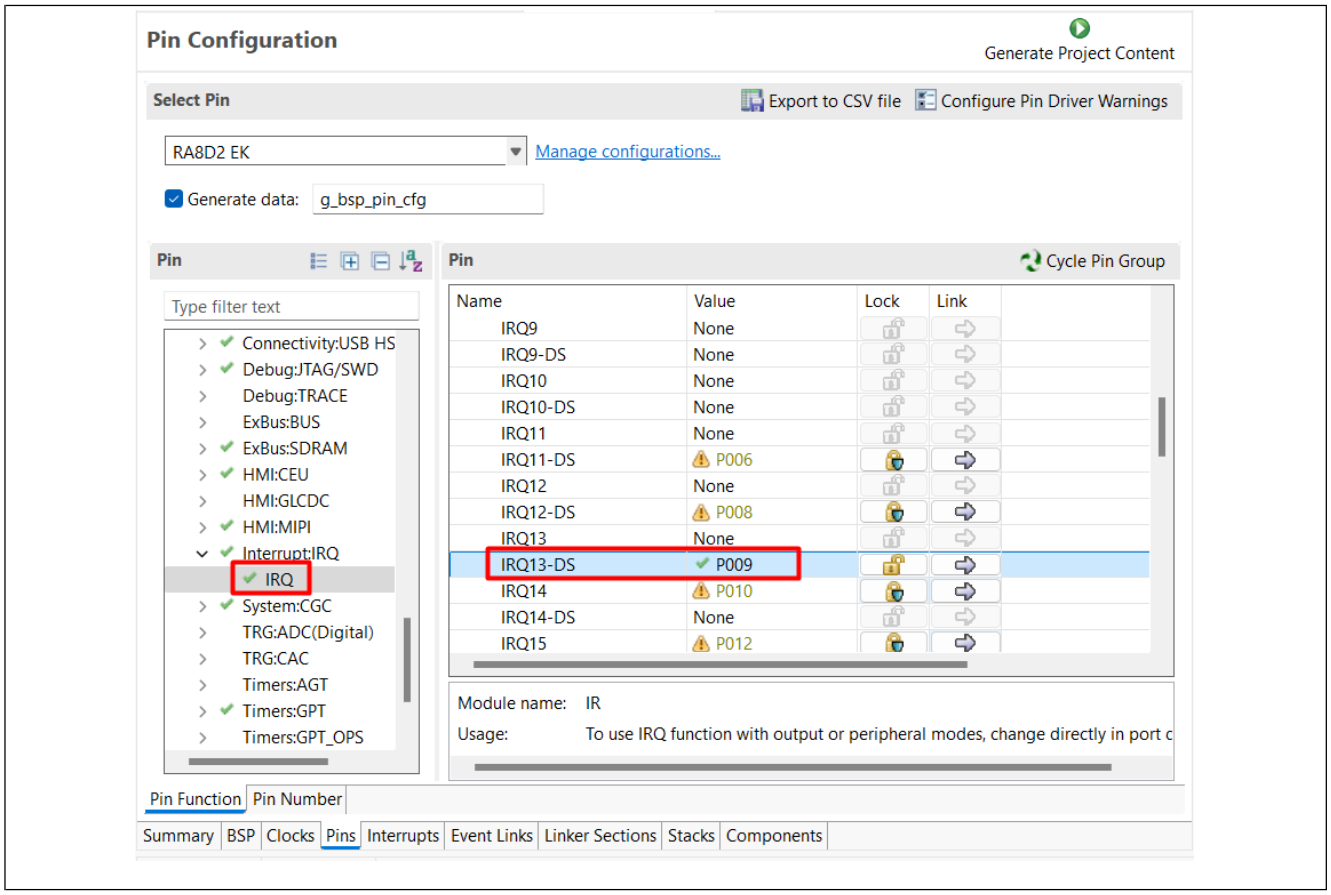


図87.RA8D2_DSP_example_CPU1プロジェクトにおける外部割込みピンの設定

手順12：ipc_callbackをCTCMに配置

ipc_callback関数をCTCMに配置する方法については、4.4.3章を参照してください。

手順13：プロジェクトコンテンツの生成

Generate Project Contentをクリックし、すべての設定を反映します。

手順14：CPU1固有のタスクを実装

RA8P1_DSP_example_cpu1/src配下のファイルを、RA8D2_DSP_example_CPU1/srcへコピーします。

手順15：ソリューションプロジェクトのビルド

RA8D2_DSP_exampleソリューションプロジェクトを右クリックし、**Build Project（プロジェクトのビルド）**を選択してビルドチェーンを適用します。

手順16：デバッグおよび動作確認

7.3章で説明した手順に従い、デバッグおよび動作確認を行います。

9. リファレンス

本クイックデザインガイドの作成にあたっては、以下の関連ドキュメントを参照しています。

- Renesas RA8P1 Group User's Manual Hardware、ドキュメント番号R01UH1064
- Renesas RA8D2 Group User's Manual Hardware、ドキュメント番号R01UH1065
- Renesas RA8M2 Group User's Manual Hardware、ドキュメント番号R01UH1066
- Renesas RA8T2 Group User's Manual Hardware、ドキュメント番号R01UH1067
- RA Flexible Software Package Documentation Release v6.2.0
- RA8P1 Memory Architecture App Note、ドキュメント番号R01AN7880
- Reference System Design for Vision AI Design using Ethos-U NPU、ドキュメント番号R11AN0995
- Using the Ethos-U NPU with RA8 MCUs、ドキュメント番号R01AN7712
- Arm Cortex®-M85 Processor Technical Reference Manual、ドキュメント番号101924、Armより入手可能

ウェブサイトおよびサポート

以下のURLから、RAファミリに関する情報およびドキュメント、サポートについて確認できます。

RAファミリの製品情報	www.renesas.com/ra
RAファミリ製品のサポートフォーラム	www.renesas.com/ra/forum
RA Flexible Software Package	www.renesas.com/FSP
ルネサスのサポート	www.renesas.com/support

改版記録

Rev.	発行日	改定内容	
		ページ	ポイント
1.10	Apr.20.26	–	初版（R01AN7881EU0110の日本語版）

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。

https://www.renesas.com/contact/