Arm Keil Studio Visual Studio Code Extensions User Guide

◆著作権

序章

この度は、弊社より Arm 製ソフトウェアツールをご購入いただきありがとうございます。
このガイドブックは Arm 開発ソフトウェア Arm Keil Studio Visual Studio Code Extensions の導入に際し、速やかな立ち上げを支援します。
本ドキュメントは Arm の提供する" ***Arm Keil Studio Visual Studio Code Extensions User Guide Issue 16 (Document ID: 108029_0000_16_en)***" の内容に基づき翻訳、作成されたものです。内容につきましては全て上記ドキュメントをマスターといたしておりますので、ご使用の際には必ず上記ドキュメントを参照の上、本ドキュメントは参考資料として用いる形をお取りくださいますようお願い申し上げます。

目次

1. Extension pack と拡張機能

Arm Keil Studio Visual Studio Code extension pack、Arm Keil Studio Pack は、Arm ベースのマイクロコントローラ(MCU) デバイス上の組み込みシステムおよび IoT ソフトウェア開発のための包括的なソフトウェア開発環境を提供します。Pack に含まれる Keil Studio extension を使用して、CMSIS solution(csolution project) を管理し、選択したハードウェア上で組み込みアプリケーションの作成、ビルド、テスト、デバッグを行えます。

Keil Studio extension は、Arm Keil Microcontroller Development Kit(MDK) の一部です。MDK は、Arm Cortex-M および Ethos-U プロセッサをベースにした組み込みアプリケーション開発用ソフトウェアツールのコレクションです。
MDK では、コマンドラインインターフェイス(CLI) または統合開発環境(IDE) を使用する、あるいはツールを継続的インテグレーション(CI) ワークフローに展開するなど、柔軟に作業を行えます。

1.1 Arm Keil Studio Pack

Arm Keil Studio Pack は、Visual Studio Code extension のコレクションです。この Pack は、Arm ベースのマイクロコントローラ (MCU) デバイス上の組み込みシステムおよび IoT ソフトウェア開発用のソフトウェア開発環境を提供します。

Keil Studio Pack には次の拡張機能が含まれています:

- Arm CMSIS Solution (Identifier: `arm.cmsis-csolution`): この extension は、CMSIS solutions(csolution projects) の使用をサポートします。
- Arm Device Manager (Identifier: arm.device-manager): この extension を使用することで、Arm Cortex-M ベースのマイクロコントローラ、開発ボードおよびデバッグプローブに対するハードウェア接続管理を行えます。
- Arm Debugger (Identifier: `arm.arm-debugger`): この extension は Microsoft Debug Adapter Protocol(DAP) を実装することにより、Visual Studio Code 用 Arm Debugger エンジンへのアクセスを提供します。Arm Debugger は Arm の ULINK ファミリのデバッグ プローブなどの外部デバッグプローブ、または ST-Link や CMSIS-DAP のようなオンボードの低コストデバッグを介して物理ターゲットへの接続をサポートします。
- Arm Environment Manager (Identifier: `arm.environment-manager`): この extension は minifest ファイルで指定したツールを環境にインストールします。たとえば、Arm Compiler for Embedded、CMSIS-Toolbox、CMake や Ninja をインストールして CMSIS solutions を操作できます。
- Arm Virtual Hardware (Identifier: `arm.virtual-hardware`): この拡張機能を使用すると、Arm Virtual Hardware を管理し、仮想ターゲット上で組み込みアプリケーションを実行できます。サービスにアクセスするには authentication トークンが必要です。詳細については、 AVH solutions overview を参照してください。

Arm Debugger は、次の拡張機能を含む Extension Pack でもあります:

- Arm Environment Manager (Identifier: `arm.environment-manager`): Keil Studio Pack を使用せずに Arm Debugger およびその他の関連する extensions をインストールする場合は、Arm Debugger extension pack の Environment Manager を使用できます。
- Memory Inspector (Identifier: `eclipse-cdt.memory-inspector`): この extension を使用することで組み込みシステムのメモリの内容の分析および監視を行えます。プロジェクトの開発フェーズでメモリ関連の問題を特定してデバッグするのに役立ちます。
- Peripheral Inspector (Identifier: `eclipse-cdt.peripheral-inspector`): この extension は System View Description (SVD) ファイルを使用して、ペリフェラルの詳細を表示します。SVD ファイルでは、マイクロコントローラまたは System-on-Chip(SoC) 内のメモリマップされたレジスタとペリフェラルを記述するための標準化された方法を提供します。

- Arm Virtual Hardware extension は現在開発中であり、このガイドには記載がありません。
- Memory Inspector および Peripheral Inspector はサードパーティのオープンソース extensions であり、このガイドには記載がありません。

Pack に含まれる拡張機能を個別にインストールして使用することもできます。ただし、Arm では、環境をすばやく設定し、サンプルを使って開発を開始できるように、Keil Studio Pack を Visual Studio Code Desktop にインストールすることを推奨しています。
詳細については Pack の README のファイルを参照してください。

2. 想定される Extension の用途

Extension の用途は次のとおりです:

- **CMSIS-Packs と solution を使用した組み込みおよび IoT ソフトウェア開発(csolution projects)**:
  Common Microcontroller Software Interface Standard(CMSIS) は数千の MCU と数百の開発ボードに対するドライバ、ペリフェラルおよびミドルウェアのサポートを提供します。csolution project フォーマットを使用することで、CMSIS-Pack ベースのデバイス、ボード、ソフトウェアコンポーネントをアプリケーションに組み込むことができます。CMSIS project でサポートされているハードウェアの詳細については、keil.arm.com の Boards および Devices のページを参照してください。CMSIS-Packs の詳細については、open-cmsis-pack.org を参照してください。

- **既存の Visual Studio Code 組み込みソフトウェア開発ワークフローの強化**:
  追加のオーバーヘッドなしで、USB device 管理と組み込みデバッグを、追加のオーバーヘッドなしで他のプロジェクト形式(CMake など) や、ツールチェインに適応させることができます。このユースケースでは、task をコンフィギュレーションするために Visual Studio Code に精通している必要があります。詳細については、個々の extensions を参照してください。

3. サンプル project を使用した Getting Started

環境をセットアップし、サンプルを使って作業を開始します:

このセクションでは、keil.arm.com から入手できるサンプルの solution または uVision project の操作について説明します。uVision project を開くと、Visual Studio Code によって自動的に csolution 形式に変換され、不足している Pack がインストールされます。また、Visual Studio Code によって Git が初期化され、現在のワークスペースの vcpkg インスタンスが構成されます。AC5 compatibility のラベルを保持する project は、自動変換をサポートしない Arm Compiler 5 のみを使用します。回避策として、Keil uVision で Arm Compiler 5 project を ArmCompiler 6 にアップデートしてから、Visual Studio Code でプロジェクトを csolution に変換できます。詳細については、Keil uVision 用サンプルのダウンロードを参照してください。

また、solution を最初から作成する、あるいは既存の uVision project を solution に変換することもできます。詳細については、Solution の作成または Keil uVision project を solution に変換するを参照してください。

README のファイルの説明に従って、Visual Studio Code Desktop に Keil Studio Pack をインストールすることをお勧めします。この Pack は、すべての Keil Studio extesions と、Red Hat YAML および clangd extensions をインストールします。

clangd を使用しない場合、Microsoft C/C++ と Microsoft C/C++ Themes extensions をインストールして、代わりに IntelliSense を有効にすることができます。

その後:

- keil.arm.com のサンプル solution project を使用してセットアッププロセスを実行します。(推奨)
- keil.arm.com から Keil uVision `*.uvprojx` project をダウンロードし、solution に変換します。(代替)

keil.arm.com にあるサンプルには、Microsoft vcpkg manifest ファイル(`vcpkg-configuration.json`)が付属しています。Environment Manager extension は、manifest ファイルによって Microsoft vcpkg を使用する solution の操作に必要なツールを取得およびアクティブ化します。

各サンプルは project のビルド、実行およびデバッグを行うための `tasks.json` のファイルと `launch.json` のファイルも付属しています。

デフォルトでインストールされるツールは次の通りです:

- Arm Compiler for Embedded
- CMSIS-Toolbox
- CMake and Ninja

開発環境のセットアップを完了します。

- HTTP プロキシのある環境で作業をしている場合、HTTP プロキシの設定の項目を参照してください。
- clangd extension と、この extension がどのようにエディタにスマートな機能を追加するかの詳細については clangd の項目を参照してください。

準備が出来たら以下の手順に移行します:

- サンプル project のビルド
- CMSIS Solution extension で何ができるかを確認します。
  - solution の context を選択する
  - solution outline の確認
  - CMSIS-Packs のインストールと pack からの Software component の選択
- ボードの接続とボード上での solution の実行
- デバッグセッションの開始
- Serial 出力の確認

3.1 サンプル solution のインポート

サンプル solution をインポートします。

手順

1. keil.arm.com にアクセスします。
2. **Hardware** メニューをクリックして、**Board** を選択します。
3. ボードを検索し、**Suggested Boards** のリスト内から選択します。
4. Projects タブから project を見つけます。
   `Keil Studio` compatibility ラベルはこのサンプルが Keil Studio Visual Studio Code extensions と互換性があることを示します。
5. カーソルを **Get Project** に移し、**Open in Keil Studio for VS Code** をクリックしてサンプル solution をインポートします。
   または、**Download .zip** オプションを利用して、solution を含む zip ファイルをダウンロードすることもできます。
6. ブラウザウィンドウの上部に"Open Visual Studio Code?" というダイアログボックスが開くので、Open **Visual Studio Code** をクリックします。
7. Visual Studio Code に表示される"Allow 'Arm Keil Studio Pack' extension to open this URI?" ダイアログボックスで **Open** をクリックします。
8. project をインポートするフォルダを選択し、**Select as Unzip Destination** をクリックします。
9. "Would you like to open the unzipped folder, or add it to the current workspace?" ダイアログボックスで **Open** をクリックします。
10. Environment Manager extension がワークスペースのアクティブ化と、`vcpkg-configuration.json` ファイル内で指定されているツールのダウンロードを自動的に行ったのを確認します。
    もし CMSIS-Pack が足りない場合は、"Solution [solution-name] requires some packs that are not installed" というメッセージのポップアップメッセージが右下隅に表示されます。
11. **Show Missing Packs** をクリックして **Problems** ビューを開きます。

12. **Problems** ビューでエラーを右クリックし、**Install missing pack** を選択します。不足している Pack が複数ある場合は、**Install all missing packs** を使用します。
    ツールチェインで Arm Compiler、Arm Debugger または Fixed Virtual Platforms などのツールを使用するには、ライセンスをアクティブ化する必要があります。Pack のインストール後にライセンスのアクティベーションが行われていない場合、右下隅にポップアップメッセージが表示されます。ライセンシングの詳細については Arm ツールのライセンスをアクティベートする の章を参照してください。
13. **Explorer** をクリックします。
    `vcpkg-configuration.json` が利用できます。このファイルには、project で操作するために必要なコンパイラツールチェインのバージョンなどの vcpkg artifacts が記録されています。ツールをインストールするために何もする必要はありません。Microsoft vcpkg と Environment Manager extensions がセットアップを処理します。Microsoft vcpkg によるツールのインストール を参照してください。

    `tasks.json` のファイルと `launch.json` のファイルも**.vscode** フォルダにあります。Visual Studio Code は `tasks.json` を使用してプロジェクトのビルドおよび実行を行い、`launch.json` をデバッグのために使用します。

3.2 Keil uVision 用サンプルのダウンロード

Keil uVision *.uvprojx project をダウンロードして開くと、Visual Studio Code によって自動的に solution に変換され、不足している Pack がインストールされます。変換は Arm Compiler 5 project に対しては行われないことに注意してください。Arm Compiler 5 project は Web サイトからダウンロードできますが、extensions とともに使用することはできません。変換できるのは Arm Compiler 6 project のみです。
回避策として、Keil uVision で Arm Compiler 5 project を Arm Compiler 6 にアップデートし、Visual Studio Code で project を solution に変換することができます。Arm Compiler 6 への変換に関する詳細なヘルプおよび情報については、Migrate Arm Compiler 5 to Arm Compiler 6 application note および Arm Compiler for Embedded Migration and Compatibility Guide を参照してください。

手順

1. keil.arm.com にアクセスします。
2. ボードを USB 経由で接続し、右下隅にある **Detect Connected Hardware** をクリックします。
3. ウィンドウ上部に表示されるダイアログボックスで、ボードのデバイスファームウェアを選択し **Connect** をクリックします。
4. 右下隅に表示されるポップアップメッセージの **Board** のリンクをクリックします。
   ボードのページが開きます。サンプル project は **Projects** タブにあります。
5. カーソルを使用したい project の **Get Project** ボタンに移し、**Download .zip** をクリックして、Keil uVision *.uvprojx サンプルをダウンロードします。
6. サンプルを解凍し、Visual Studio Code でフォルダを開きます。
   変換はすぐに開始され、不足している必要な Pack があれば自動的にインストールされます。

   ダイアログボックスが表示されます。次のタスクを実行できます:
   - solution を新しいワークスペースで開く(**Open** オプション)
   - solution を新しいウィンドウと新しいワークスペースで開く(**Open project in new window** オプション)

   変換エラーがある場合は、利用可能な `uv2csolution.log` ファイルを確認してください。

7. Environment Manager extension がワークスペースのアクティブ化と、`vcpkg-configuration.json` ファイル内で指定されているツールのダウンロードを自動的に行ったのを確認します。

ツールチェインで Arm Compiler、Arm Debugger または Fixed Virtual Platforms などのツールを使用するには、ライセンスをアクティブ化する必要があります。Pack のインストール後にライセンスのアクティベーションが行われていない場合、右下隅にポップアップメッセージが表示されます。ライセンシングの詳細については Arm ツールのライセンスをアクティベートする の章を参照してください。

`*.cproject.yml` ファイルおよび`*.csolution.yml` ファイルは、**Explorer**  内の`*.uvprojx` の隣にあります。

`vcpkg-configuration.json` が利用できます。このファイルには、project で操作するために必要なコンパイラツールチェインのバージョンなどの vcpkg artifacts が記録されています。ツールをインストールするために何もする必要はありません。Microsoft vcpkg と Environment Manager extensions がセットアップを処理します。Microsoft vcpkg によるツールのインストール を参照してください。

`tasks.json` のファイルと `launch.json` のファイルも**.vscode** フォルダにあります。Visual Studio Code は `tasks.json` を使用してプロジェクトのビルドおよび実行を行い、`launch.json` をデバッグのために使用します。

3.3 開発環境のセットアップを完了する

開発環境のセットアップを完了するには:

- HTTP プロキシの設定を行います。この手順は HTTP プロキシのある環境で作業をしている場合のみ必要です。
- この Pack は、すべての Keil Studio extesions と、Red Hat YAML および clangd extensions をインストールします。この extensions の詳細については clangd の項目を参照してください。

3.3.1 HTTP プロキシの設定 (オプション)

この手順は HTTP プロキシのある環境で作業をしている場合のみ必要です。次の標準環境変数によって、HTTP プロキシを使用するようにツールをコンフィギュレーションできます。

- `HTTP_PROXY`: HTTP リクエストに使用するプロキシを設定します。
- `HTTPS_PROXY`: HTTPS リクエストに使用するプロキシを設定します。
- `NO_PROXY`:拡張機能が正しく動作させるために内部トラフィックのプロキシを無効にするには、少なくとも `localhost`、`127.0.0.1` を含めるように設定します。

3.3.2 clangd

clangd 拡張機能は、コード補完、コンパイルエラー、定義位置への移動などのスマートな機能をエディタに追加します。

clangd extension には clangd language server が必要です。PATH にサーバが見つからない場合は、Command Palette から **clangd: Download language server** コマンドを使用して追加します。詳細については、clangd extension の `README` のファイルを参照してください。

clangd をインストールした後は、追加の設定は必要ありません。CMSIS Solution extension は csolution ファイルが変更されるたびに、または solution の context (**Target Type** と **Build Type** のタイプ) を変更するたびに、solution 内の各 project に対して `compile_commands.json` ファイルを生成します。solution 内の各プロジェクトに対して.clangd のファイルが最新の状態に保たれます。clangd extension は.clangd のファイルを使用して `compile_commands.json` のファイルを見つけ、追加のコンパイルフラグを提供し、IntelliSense を有効にします。詳細については clangd のドキュメントを参照してください。

clangd で IntelliSense を改善するために、追加のスコープ付きコンパイラフラグ(特定のマクロを定義する) がプロジェクトのコンフィギュレーションファイル(`.clangd`) とグローバルユーザーコンフィギュレーションファイル(`config.yaml`) の両方に追加されます。詳細については clangd のドキュメントを参照してください。

`.clangd` のファイルと `compile_commands.json` のファイルの自動生成をオフにするには:

1. settings を開きます:
   - Windows または Linux の場合、**File** > **Preferences** > **Settings** を選択します。
   - macOS の場合、**Code** > **Settings** > **Settings** を選択します。
2. **Cmsis-csolution: Auto Generate Clangd File** と **Cmsis-csolution: Auto Generate Compile Commands** の設定を探し、これらのチェックボックスをクリアします。

マクロ定義のコンパイラフラグの自動追加をオフにするには:

1. settings を開きます:
   - Windows または Linux の場合、**File** > **Preferences** > **Settings** を選択します。
   - macOS の場合、**Code** > **Settings** > **Settings** を選択します。
2. **Cmsis-csolution: Clangd Armclang Macro Query** の設定を探し、チェックボックスをクリアします。

3.4 サンプル project のビルド

サンプルプロジェクトがビルドされることを確認します。プロジェクトは、**Explorer** の **Build** を使用して、**Solution outline**、または Command Palette からビルドできます。

手順

1. プロジェクトのビルド:
   - **Explorer** から:
     a. **Explorer** ビューを開きます。
     b. `*.csolution.yml` ファイルを右クリックし、**Build** を選択します。
     以下のオプションも右クリックメニューにあります:
     ◦ **Clean Output Directories**: アクティブな solution の出力ディレクトリをクリーンします。
     ◦ **Rebuild**: cproject をビルドする前に出力ディレクトリをクリーンします。

   - **Solution outline** から:
     a. Activity Bar の **CMSIS** をクリックします。
     Solution outline が開きます。

     Note: solution の context 内のすべての project をクリーン済みの場合は、Open Build Context Editor をクリックして Build Context ビューを開いて project を選択できます。Solution の context を設定する の章を参照してください。

     b. **Solution outline** 上にカーソルを移動します。
     Build アイコンは solution または project レベルにあります。
     c. **Build** をクリックします。
     Clean Output Directories および **Rebuild** オプションは、**Views and More Actions** からも使用できます。

     Build task を `tasks.json` ファイルで設定することで、build ボタンの動作をカスタマイズできます。`tasks.json` ファイルは、keil.arm.com から入手可能なすべてのサンプルに対して提供されています。詳細については、Build task の設定 の章を参照してください。

   - **Command Palette** から:
     Build, **Clean Output Directories**, および **Rebuild** も Command Palette から、**CMSIS: Build**, **CMSIS: Clean Output Directories**, および **CMSIS: Rebuild** コマンドとして起動できます。

     Note: ENOENT エラーでビルドが失敗した場合は、右下隅に表示されるポップアップメッセージの指示に従って CMSIS-Toolbox をインストールしてください。詳細については、CMSIS-Toolbox が見つからないため ENOENT エラーでビルドが失敗する の章を参照してください。

2. **Terminal** タブを確認して、ELF ファイル(`.axf`) が生成された場所を見つけます。

3.5 Solution の context を選択する

Context とは、solution 内の特定のプロジェクトのターゲットタイプ(build target) とビルドタイプ(build configuration) を組み合わせたものです。

詳細については、Solution の context を設定する の章を参照してください。

3.6 Solution outline の確認

Solution outline では、solution のコンテンツがツリービューで表示されます。

詳細については、Solution outline の使用 の章を参照してください。

3.7 CMSIS-Packs のインストールと pack からの Software component の選択

CMSIS-Packs には、project を迅速に構築できる再利用可能な Software component が含まれています。CMSIS-Packs は、solution の `csolution.yml` ファイル内に一覧があります。CMSIS Solution extension は、Pack の pack キャッシュへのインストールをシームレスに処理します。

詳細については、CMSIS-Packs および CMSIS-Packs のインストール の章を参照してください。

Software Components には、solution のアクティブ project で選択されているすべての Software Component が表示されます。

詳細については、 Software Components の管理 の章を参照してください。

3.8 ボードの接続

ボードを接続します。サポート済み開発ボード、MCU、デバッグプローブの詳細については、サポートされるハードウェア の章を参照してください。

手順

1. Activity Bar の Device Manager  をクリックして、Device Manager extension を開きます。
2. ボードを USB 経由コンピュータに接続します。
   ボードが検出され、ポップアップメッセージが表示されます。
3. ポップアップメッセージ上で **OK** をクリックしてハードウェアを使用します。
   これで、ボードを使用して project を実行およびデバッグする準備が整いました。

3.9 ボード上で solution を実行する

ボード上で solution project を実行します。

手順

1. Activity Bar の **CMSIS**  をクリックします。
   Solution outline が開きます。
2. **Solution outline** 上にカーソルを移動します。
   Run アイコンは、run configuration の **Build Context** ビューで選択した内容に応じて、solution または project レベルで使用できます。詳細については、Solution の context を設定する の章を参照してください。
3. **Run**  をクリックします。
4. マルチコアのデバイスを使用しており、`launch.json` ファイルで"`processorName`" を指定しておらず、CMSIS Solution extension がインストールされていない場合は、ウィンドウの上部に表示される **Select a processor** ドロップダウンリストで project に対して適切なプロセッサを選択する必要があります。
   project がボード上で動作します。
5. **Terminal** タブを確認します。

3.10 デバッグセッションの開始

デバッグセッションを開始します。

手順

1. Activity Bar の **CMSIS**  をクリックします。
   Solution outline が開きます。
2. **Solution outline** 上にカーソルを移動します。
   Debug アイコンは、run configuration の **Build Context** ビューで選択した内容に応じて、solution または project レベルで使用できます。詳細については、Solution の context を設定する の章を参照してください。
3. **Debug**  をクリックします。
4. マルチコアのデバイスを使用しており、`launch.json` ファイルで"`processorName`" を指定しておらず、CMSIS Solution extension がインストールされていない場合は、ウィンドウの上部に表示される **Select a processor** ドロップダウンリストで project に対して適切なプロセッサを選択する必要があります。
   Run and Debug ビューが表示され、デバッグセッションが開始されます。デバッガはプログラムの `main()` 関数で停止します。
5. デバッグ出力を **Debug Console** タブで確認します。

次のステップ

Visual Studio Code で使用できるデバッグ機能の詳細については、Visual Studio Code のドキュメントを参照してください。

4. Arm Environment Manager extension

The Arm Environment Manager extension を使用すると、Microsoft vcpkg によってコンパイラツールチェイン などの environment artifacts を管理できます。extensions は、vcpkg manifest ファイルを使用して、開発環境のセットアップに必要な artifacts を取得およびアクティブ化します。

project の artifacts は project ソースコードの vcpkg-configuration.json ファイルに保存されます。そのため、project を使用するすべてのユーザが同じツールを利用できます。

vcpkg に関する情報は vcpkg.io および Microsoft Learn にあります。

Environment Manager extension と vcpkg を使用しない場合は、CMSIS-Toolbox と CMake および Ninja など、その他必要なツールを手動でダウンロードおよびインストールすることで、プロジェクトの artifacts をインストールできます。詳細については、CMSIS-Toolbox を参照してください。Open-CMSIS-Pack のドキュメント内インストール手順を参照してください。CMSIS-Toolbox のパスを指定する方法については、CMSIS-Toolbox をシステムの PATH に追加する の章を参照してください。その他の具体的なケースについては、特定のインストールに関する使用例 の章を参照してください。

Environment Manager extension には、ツールのライセンスに役立つ機能も含まれています。詳細については、Arm ツールのライセンスをアクティベートする の章を参照してください。

Arm Environment Manager extension のコマンドと設定の完全なリストを利用できます。リストを表示するには、Visual Studio Code Activity Bar で **Extensions**  をクリックします。extensions のリストで **Arm Environment Manager** をクリックし、**Features**(Windows) または **Feature Contributions**(macOS) をクリックします。

4.1 Microsoft vcpkg によるツールのインストール

Microsoft vcpkg は、環境のセットアップのために Pack とともにインストールされた Environment Manager extension と組み合わせて動作します。

個々の公式 Arm サンプル project には、manifest ファイル(vcpkg-configuration.json) が付属しています。manifest ファイルには、project で作業するために必要な vcpkg artifacts が記録されます。artifact とは、作業用開発環境に必要なパッケージのセットです。関連するパッケージの例には、コンパイラ、リンカ、デバッガ、ビルドシステム、プラットフォーム SDK などがあります。

vcpkg の詳細については、公式の Microsoft vcpkg のドキュメントを参照してください。また、artifact の詳細については Microsoft vcpkg-tool repository も参照してください。

Windows 環境の場合、Keil Studio および Environment Manager extension を使用する際に long path のサポートを有効にする必要があります。long path が有効になっていないと、vcpkg artifacts のダウンロードとインストールが失敗する可能性があります。
Environment Manager は、Windows レジストリで long path が有効になっているかどうかを検出し、有効になっていない場合は警告を表示します。Windows の設定で long path を有効にするには、次の手順に従ってください:
Enable long paths in Windows 10, version 1607, and later

4.2 自動アクティベーションの確認

新しいワークスペースを開いたり、既存のワークスペースを複製したり、keil.arm.com からサンプルプロジェクトを開いたりすると、 Environment Manager extension は、ワークスペースを自動的にアクティブ化し、`vcpkg-configuration.json` ファイルで指定されたツールをダウンロードします。ダイアログボックスが開き、アクティブ化について確認できます。`vcpkg-configuration.json` ファイルを開いて、何がインストールされるかを確認できます。

自動アクティベーションはいつでも設定変更できます。

4.3 Microsoft vcpkg でインストールされたツールの確認

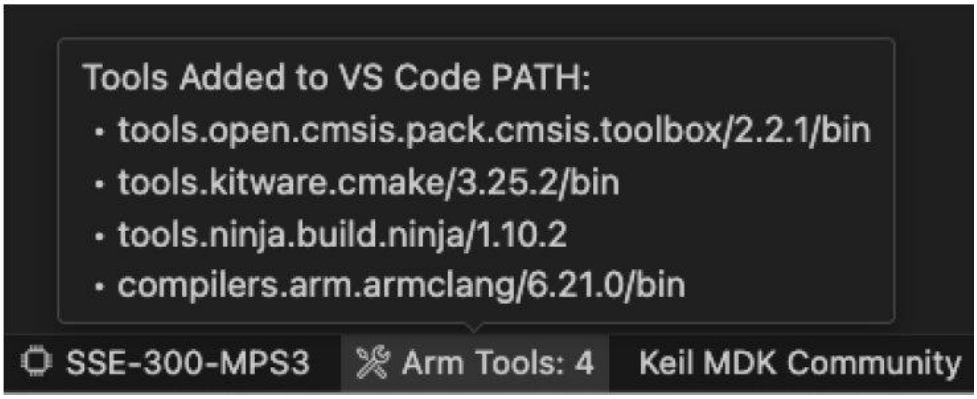
`vcpkg-configuration.json` manifest ファイルは、Microsoft vcpkg にインストールする artifacts を示します。例:

```
"requires": {
        "arm:tools/open-cmsis-pack/cmsis-toolbox": "2.2.1",
    "arm:compilers/arm/armclang": "6.21.0",
        "microsoft:tools/kitware/cmake": "3.25.2",
        "microsoft:tools/ninja-build/ninja": "1.10.2"
}
```

この例の manifest ファイルでインストールされる artefacts は、cmsis-toolbox、armclang(Arm Compiler for Embedded)、cmake、および ninja です。

ステータスバーの **Arm Tools** の上にマウスを移動すると、インストールされているツールが表示されます。

図 4-1: Arm Tools



ステータスバーの **Arm Tools** をクリックし、**View Log** オプションを選択することもできます。これにより、**Output** タブ(**Arm Tools** カテゴリ) が開きます。

デフォルトでは、Microsoft vcpkg は `user` フォルダにツールをインストールします。

- Windows: `c:¥Users¥<user>¥.vcpkg¥artifacts`
- Linux: `/home/<user>/.vcpkg/artifacts`
- macOS: `/Users/<user>/.vcpkg/artifacts`

Microsoft vcpkg が project で有効化されると、 Visual Studio Code で開いたすべての **Terminal** には、デフォルトで PATH に追加されたすべてのツール（Arm Compiler for Embedded、CMSIS-Toolbox、CMake、Ninja）が設定されます。このプロセスにより、`cpackget`、`cbuildgen`、`cbuild` および `csolution` などその他さまざまな CMSIS-Toolbox のツールを実行できます。

4.4 manifest ファイルを手作業で修正する

環境内のツールを追加または変更するには、project の manifest ファイルに含まれる artifacts を変更します。
Arm が提供する artifacts は、keil.arm.com の Arm tools available in vcpkg にリストがあります。
このページでは、 Artifactory repository manager で利用可能なバージョンのみ記載されています。

インストールする artifacts のコードスニペットをコピーし、project の `vcpkg-configuration.json` manifest ファイルの`"requires":` セクションにペーストして、ファイルを保存します。
新しく追加または更新された artifacts は自動的にダウンロードされ、アクティブ化されます。
manifest ファイルを手作業で編集する代替えの方法は、Arm Tools visual editor の使用 の章を参照してください。

4.5 Arm Tools visual editor の使用

`vcpkg-configuration.json` manifest ファイルを直接編集する代わりに、 **Arm Tools visual editor** を使用して、環境内のツールを追加または変更することもできます。

手順

1. **Explorer** ビューのいずれかの場所で右クリックします。
2. 開いたメニューから、**Configure Arm Tools Environment** を選択します。
   Arm Tools editor が開きます。
   ステータスバーの **Arm Tools** をクリックし、ウィンドウ上部に表示されるドロップダウンリストで **Configure Arm Tools Environment** オプションを選択して editor を開くこともできます。
3. ドロップダウンリストから、環境で使用するツールをインストールまたは更新します。
   たとえば、 **Arm Compiler for Embedded** のドロップダウンリストでバージョン 6.22.0 を選択します。
   Artifactory に存在するバージョンのみリポジトリマネージャはユーザインターフェイス上に表示します。
4. **Auto Save** が有効になっていない場合(**File** > **Auto Save**) は設定を保存します
   新しく追加または更新されたツールは自動的にダウンロードされ、アクティブ化されます。
   何がインストールされたかの詳細は、**Output** タブ(**View** > **Output**) 内で確認できます。

4.6 vcpkg アクティベーションオプション

vcpkg-configuration.json ファイルをワークスペースに追加したり、Microsoft vcpkg を使用して環境をアクティブ化、非アクティブ化、または再アクティブ化したり、vcpkg レジストリを更新したりするには、いくつかのオプションがあります。keil.arm.com のサンプルを使用している場合、または Create New Solution ビューで最初からソリューションを作成した場合、その環境ではデフォルトでアクティブ化されています。

手順

1. **Explorer** からワークスペースを開きます。
2. `vcpkg-configuration.json` ファイルを右クリックします。
3. 環境のアクティベーションステータスと **Environment Manager** で選択した設定に応じて、次のオプションがあります:
   - **Configure Arm Tools Environment**: visual editor が開きます。このオプションは、ビジュアルエディタを開き、インストールするツールを選択します。Arm Tools visual editor の使用 の章を参照してください。
   - **Activate Environment**: 環境をアクティブ化します。このオプションでは、以前に環境を非アクティブ化した場合、または **Environment Manager** で **Activate On Config Creation** または **Activate On Workspace Open** の設定を変更した場合にのみ使用できます。ツールは PATH に登録されます。
   - **Deactivate Environment**: アクティブな環境を非アクティブ化します。ツールは PATH からも削除されます。
   - **Reactivate Environment**: 環境の非アクティブ化後にアクティブ化します。(たとえば、vcpkg のコンフィギュレーションを変更した場合)
   - **Update Tool Registry**: レジストリに登録された最新の artifacts をチェックします。

ステータスバーの **Arm Tools** をクリックすると、同様のオプションが利用できます。

ドロップダウンリストの **View Log** オプションを選択すると **Output** タブが開き、インストールされているツールを確認できます。Microsoft vcpkg でインストールされたツールの確認 の章を参照してください。

project に `vcpkg-configuration.json` ファイルが含まれていない場合、またはアクティブな環境を非アクティブ化している場合は、ステータスバーの **Arm Tools** をクリックし、 **Add Arm tools Configuration To Workspace** を選択して visual editor を開き、ツールを選択します。

4.7 コマンドラインから vcpkg を使用する

コマンドラインから vcpkg を使用して、再現できるツールのインストレーションを作成することもできます。
Arm Developer Learning Paths には、vcpkg のインストールと初期化の方法や、コンフィギュレーションファイルを作成して使用する方法を示すサンプルシナリオがあります。Install tools on the command line using vcpkg を参照してください。

4.8 特定のインストールに関する使用例

このセクションでは、次のユースケースについて説明します:

- 特定のバージョンの Arm Compiler for Embedded のインストール
- Arm Compiler for Embedded FuSa の使用
- vcpkg の代わりにインストール済みのツールチェインを使用する
- インターネットにアクセスできないマシンでの作業

4.8.1 Arm Compiler for Embedded を特定のバージョンに切り替える

特定の Arm Compiler for Embedded のバージョンに切り替えるには、Arm Tools visual editor を使用します。

Artifactory リポジトリマネージャ上に存在するバージョンのみユーザインターフェイスに表示されます。
バージョン 6.18.0 以上を選択できます。

Arm Compiler for Embedded の 6.18.0 以前のバージョンは、User-Based Licensing(UBL)をサポートしていません。そのため、これらのバージョンは MDK v6 では動作しません。
User-Based Licensing のサポートと下位互換性の詳細については、User-based licensing User Guide を参照してください。

特定の Arm Compiler for Embedded のバージョンに切り替えるには:

1. Arm Tools visual editor で、必要とする Arm Compiler for Embedded のバージョンを選択します。
   たとえば、**Arm Compiler for Embedded** ドロップダウンリストからバージョン 6.18.0 を選択します。

2. project の`*.csolution.yml` または`*.cproject.yml` ファイルにインストールしたバージョンを定義します。
   たとえば、バージョン 6.18.0 の場合、Open-CMSIS-Pack ドキュメントの説明に従って、`*.csolution.yml` ファイルに `compiler: AC6@6.18.0` を追加します。
3. Arm Compiler for Embedded ツールチェインを使用している OS のグローバルな環境変数に追加します。
   たとえば、バージョン 6.18.0 の場合 AC6_TOOLCHAIN_6_18_0 の環境変数が、ツールチェインのバイナリを指すようにします。Open-CMSIS-Pack のドキュメントを参照してください。
4. Visual Studio Code を再起動します。
5. `*.csolution.yml` ファイルを右クリックし、**Rebuild** を選択して project をリビルドします。

4.8.2 Arm Compiler for Embedded FuSa を使用する

プロジェクトに機能安全(FuSa:Functional Safety) の要件がある場合は、Arm Compiler for Embedded FuSa を使用できます。

Arm Compiler for Embedded FuSa は Product Download Hub(PDH) からのみ入手できます。使用には MDK-Professional のライセンスが必要です。

Arm Compiler for Embedded FuSa の 6.16.2 以前のバージョンは、User-Based Licensing(UBL) をサポートしていません。そのため、これらのバージョンは MDK v6 では動作しません。
User-Based Licensing のサポートと下位互換性の詳細については、User-based licensing User Guide を参照してください。

Arm Compiler for Embedded FuSa version 6.16.2 をインストールするには:

1. Product Download Hub (PDH) から Arm Compiler for Embedded FuSa のバージョン 6.16.2 をダウンロードし、手作業でマシンにインストールします。

   PDH にアクセスするには、Arm アカウントが必要です。Arm Compiler for Embedded FuSa をダウンロードするには、アカウントが有効な MDK-Professional ライセンスに関連付けられている必要があります。

2. project の`*.csolution.yml` または`*.cproject.yml` ファイルにインストールしたバージョンを定義します。
   たとえば、バージョン 6.18.0 の場合、Open-CMSIS-Pack ドキュメントの説明に従って、`*.csolution.yml` ファイルに `compiler: AC6@6.16.2` を追加します。
3. Arm Compiler for Embedded FuSa ツールチェインを使用している OS のグローバルな環境変数に追加します。
   AC6_TOOLCHAIN_6_16_2 の環境変数が、ツールチェインのバイナリを指すようにします。Open-CMSIS-Pack のドキュメントを参照してください。

4. Visual Studio Code を再起動します。
5. プロジェクトをビルドします。サンプル project のビルド の章を参照してください。

4.8.3 インストール済みのツールチェインを使用する

Keil Studio Pack をインストールする前にインストール済みのツールチェインを使用するには、個人設定との競合を避けるために vcpkg を非アクティブ化する必要があります。project に vcpkg-configuration.json ファイルが含まれていない場合は、何もする必要はありません。

手順

1. vcpkg を非アクティブ化します:
   a. settings を開きます:
      - Windows または Linux の場合、**File** > **Preferences** > **Settings** を選択します。
      - macOS の場合、**Code** > **Settings** > **Settings** を選択します。
   b. **Activate on Workspace Open** の設定を探し、チェックボックスをクリアします。
2. ツールチェインが正しくインストールされ、使用している OS のグローバルな環境変数にツールチェインが追加されていることを確認します。
   たとえば、Arm Compiler for Embedded version 6.18.0 の場合、AC6_TOOLCHAIN_6_18_0 の環境変数が、ツールチェインのバイナリを指すようにします。Open-CMSIS-Pack のドキュメントを参照してください。
3. Visual Studio Code を再起動します。
4. 環境変数のパスをチェックするには `cbuild list toolchains -v` を使用します。

4.8.4 ネットワークから遮断されている PC で Keil Studio extensions を使用する

Activate Environment オプション(vcpkg アクティベーションオプション の章を参照) を使用するか、接続されたマシンのターミナルから `vcpkg activate` コマンドを実行して、vcpkg ルートディレクトリをネットワークから遮断されている PC に転送することができます。これにより、必要なすべてのツールがネットワーク接続のできないマシンに転送されます。
その後、インターネット接続なしでネットワークから遮断されている PC を使用できるようになります。

5. Arm CMSIS Solution extension

Arm CMSIS Solution extension は、CMSIS solutions(csolution projects)の操作をサポートします。この拡張機能は、solution の作成に必要な情報を管理します。

CMSIS Solution extension を使用すると、次のタスクを実行できます:

- Solution の context を設定する
- Solution outline の使用
- CMSIS-Packs のインストール
- Software Components の管理
- Configuration Wizard を使用してスタートアップコードやその他のコンフィギュレーションファイルをカスタマイズする

以下も行えます:

- Solution を最初から作成する
- Keil uVision project を solution に変換する
- Build task の設定
- Solution の初期化
- CMSIS csolution API の使用

新しい project を最初から作成するのではなく、keil.arm.com の既存のサンプルプロジェクトを使用する方法については、サンプル project を使用した Getting Started の章を参照してください。

Arm CMSIS Solution extension のコマンドとコンフィギュレーションの完全なリストがあります。リストを表示するには、Visual Studio Code Activity Bar の **Extensions**  をクリックします。extensions のリストで **Arm CMSIS Solution** をクリックしてから、**Features**(Windows) または **Feature Contributions**(macOS) をクリックします。

5.1 CMSIS solutions

solution は、大規模なアプリケーションの一部であり、個別にビルドできる関連するプロジェクトを整理するために使用されるコンテナです。関連プロジェクトのプロジェクト設定を参照してください。solution の例については、Project Setup for Related Projects を参照してください。

solution は、`*.csolution.yml` ファイルを使用して YAML フォーマットで定義されます。`*.csolution.yml` ファイルは、アプリケーションの完全なスコープと、アプリケーションに含まれる project のビルド順序を定義します。個々の project は、`*.cproject.yml` ファイルを使用して定義されます。`*.cproject.yml` ファイルは、独立したビルドの内容を定義します。各 project は、1 つのバイナリファイル(build artifact) に対応します。

ソリューションの`*.csolution.yml` および`*.cproject.yml` ファイルは手作業で編集できます。

Keil Studio Pack には Red Hat YAML extension が含まれており、CMSIS Solution extension では YAML スキーマを使用してこれらのファイルの編集が簡単にできるようになっています。extension の詳細については vscode-yaml repository を参照してください。

ソリューションとプロジェクトの構造を理解するには、CMSIS-Toolbox のドキュメント内 Build Overview および Project Examples を参照してください。csolution project ファイルの詳細については、CMSIS Solution Project File Format を参照してください。

5.2 Solution の context を設定する

solution contexts を確認します。context とは、solution 内の特定の project の Target Type と Build Type の組み合わせです。

手順:

1. Activity Bar の **CMSIS**  をクリックして **CMSIS** ビューを開きます。
2. 以下のオプションのいずれかを選択します:
   - **Solution outline** ヘッダの  をクリックします。
   - Command Palette で **CMSIS: Manage Solution Settings** を選択します。
   - status bar で  をクリックします。

   Build Context ビューが開きます。
3. solution に対して有効な context を確認します。Target Type と build に含まれる project および Build Type を変更できます。
   run configuration や debug configuration を変更したり、新しい configurations を追加することもできます。
   - **Active Target**: Target Type を選択して、solution のビルドに使用するハードウェアを指定します。一部のサンプルは Arm Virtual Hardware(AVH) ターゲットとも互換性があり、その場合にはさらに多くのオプションが利用できます。詳細については、AVH solutions overview を参照してください。

     Edit targets in the csolution.yml をクリックして、YAML ファイルを直接編集して target types を指定します。

   - **Active Projects**:
     - **Project Name**: ビルドに含まれる project です。solution に複数の project がある場合は、含める project をここで選択できます。
     - **Build Type**: build configuration です。build configuration を使用すると、特定のテストに対して個々の target type を設定できます。solution 内の個々のプロジェクトに対して異なる build type を設定できます。アプリケーションの要求に応じて、独自の build type を作成できます。よく使われる 2 つの例としては、インタラクティブデバッグ用のソフトウェアのフルデバッグビルドである **Debug** と、システムの最終のコードデプロイとなる **Release** です。

   project の横にある **Edit cproject.yml** をクリックして、`<project-name>.cproject.yml` ファイルを開きます。YAML 構文のサポートにより容易に編集できます。

選択できる projects と build type は特定の target の context によって定義されます。一部のオプションは、選択したターゲットでは除外されていることがあり、その場合は利用できません。context の詳細と変更方法については CMSIS-Toolbox のドキュメント内、Context および Conditional build を参照してください。たとえば、**`for-context`** と **`not-for-context`** を使用して、**`*.csolution.yml`** ファイルの **`project:`** レベルで target type の追加や除外ができます。

- **Run and Debug**:
  Run Configuration および **Debug Configuration**: solution で使用する run configuration と debug configuration を選択します。solution に含まれる project ごとに異なる run configuration と debug configuration を選択することもできます。

  以下も行えます:
  - リスト内のエントリ上にマウスを移し、ペンアイコンをクリックして、visual editor で既存の configuration を編集します。
  - **+ Add new** をクリックして、新しい configuration を追加した後:
    - run configuration は、Arm Debugger extension を使用している場合、ウィンドウの上部に表示されるドロップダウンリスト内で `arm-debugger.flash: Flash Device` タスクを選択します。
    - debug configuration は、Arm Debugger extension を使用している場合、`launch.json` ファイルに表示されるドロップダウンリスト内で `Arm Debugger: Attach`、`Arm Debugger: Launch` または `Arm Debugger: Launch FVP` を選択します。

  詳細については Run and Debug Configuration visual editor を使った run configuration および Run and Debug Configuration visual editor を使った debug configuration の章を参照してください。

4. ウィンドウの下部にある Problems タブに移動し、エラーがないか確認します。Problems タブが表示されていない場合は、View メニューに移動して Problems をクリックするか、Ctrl+Shift+M (Windows) または Cmd+Shift+M (macOS) を押します。
5. `main.c` のファイルを開き、有効な IntelliSense の機能を確認します。その他の機能については IntelliSense の Visual Studio Code のドキュメントを参照してください。

5.3 Solution outline の使用

Solution outline では、solution のコンテンツがツリービューで表示されます。

Activity Bar の **CMSIS** 

をクリックして、**CMSIS** ビューを開きます。**Solution outline** が左側に表示されます。

以前に solution の context 内のすべての project をクリアした場合、**Open Build Context Editor** をクリックして **Build Context** ビューを開き、project を選択できます。Solution の context を設定する の章を参照してください。

Solution outline には、**Build Context** ビューで選択された solution に含まれる project(cproject) が表示されます。各 cproject ファイルには、configuration の設定、ソースコードファイル、build settings、およびその他 project 固有の情報が含まれています。extension は、これらの設定とファイルを使用して、ボードまたはデバイスのソフトウェア project を管理およびビルドします。

以下の project に関する情報があります:

- `Groups`: Groups は、コードファイルを論理ブロックに構造化する方法です。groups にファイルを追加できます。
  グループの上にカーソルを移動して をクリックし、以下のうちいずれかのオプションを選択します:
  - **Add New File**: ファイルを作成して groups に追加します。
  - **Add Existing File**: 既存のファイルを選択して groups に追加します。
  - **Add From Component Code Template**: ドロップダウンリストから component code template を選択し、groups に追加します。code template は、project の software components に含まれる事前定義済みファイルで、project の開発を始める手助けになります。

  追加したファイルは、solution の`*.cproject.yml` ファイルの `groups` キーの下にリストされます。
- `Components`: project に対して選択されたすべての software components です。Components は component class (Cclass) 別に分類されます。選択された Components のコードファイル、user code templates、および API は、親コンポーネントの下に表示されます。files、templates、または API をクリックすると、エディタ上で開かれます。
- `Layers`: clayer ファイルである`*.clayer.yml` は、project のソフトウェアレイヤを定義します。ソフトウェアレイヤは、ソースファイル、事前コンフィギュレーション済み software component、および configuration file のセットです。clayer ファイルは複数の project で使用できます。project 内の各レイヤで使用される software component は、ツリービューに表示されます。

Solution outline ラベルには、アクティブな solution の名前が表示されます。ラベルの上にカーソルを移動すると、次のいずれかのアクションを選択できます:

- **Build**: をクリックすると、アクティブな solution に含まれるすべての project をビルドします。また、各 project を個別にビルドすることもできます。
- **Run**: をクリックすると、ハードウェア上で solution を実行します。debug configuration に対する Build Context ビューで選択した内容に応じて、**Run** アイコンは project レベルでも利用可能です。
- **Debug**: をクリックすると、solution をデバッグします。debug configuration に対する Build Context ビューで選択した内容に応じて、**Debug** アイコンは project レベルでも利用可能です。
- **Open Csolution File**: メインの `csolution.yml` ファイルを開きます。project または layer 上にカーソルを移動させた場合は、Open File オプションも利用できます。
- **Manage Solution Settings**: をクリックすると、Solution の context を設定できます。
- **Collapse All**: をクリックすると、outline 内のエントリをすべて閉じます。

- **Views and More Actions**: ⋯
  - **Clean Output Directories**: アクティブな solution のアウトプットディレクトリをクリーンします。
  - **Rebuild**: project のビルド前にアウトプットディレクトリをクリーンします。
  - **Refresh**: Explorer ビューをリフレッシュします。
  - **Activate Solution**: アクティブな solution を選択します。ワークスペースに複数の solution がある場合、ワークスペースでこのオプションを使用すると、特定の solution から別の solution に切り替えることができます。csolution project ファイルを含むフォルダを右クリックするか、`csolution.yml` ファイル自体を右クリックする操作を **Explorer** で行うと同じオプションを利用できます。アクティブな solution はハイライトされます。
  - **Convert uvproj to csolution**: Keil uVision project を solution に変換します。
  - **New Solution**: はじめから Solution を作成します。

Solution outline では、各 project の横に選択した build type と target type が表示されます。各 project で選択されている software components を確認できます。 をクリックすることで **Software Components** ビューが開きます。詳細については、Software Components の管理 の章を参照してください。

Ctrl+F(Windows) または Cmd+F(macOS) を押すと、Solution outline 内の要素を検索できます。

`*.csolution.yml`、`*.cproject.yml` および`*.clayer.yml` ファイルフォーマットについては、Open-CMSIS-Pack のドキュメントで説明されています。

5.4 CMSIS-Packs

CMSIS-Packs を使用することで、Cortex-M デバイス用の組み込みソフトウェアアプリケーションを迅速かつ簡単に作成、ビルド、デバッグできます。
CMSIS-Packs は、Software Components、デバイスパラメータ、ボードサポートの配信メカニズムです。
CMSIS-Packs は、以下のようなファイルのコレクションです:

- ソースコード、ヘッダファイル、ソフトウェアライブラリ ― たとえば、RTOS、DSP、汎用ミドルウェア
- メモリレイアウトやデバッグ設定などのデバイスパラメータ、スタートアップコード、フラッシュプログラミングアルゴリズム
- ボードサポート(ドライバ、ボードパラメータ、デバッグ接続の説明など)
- ドキュメントとソースコードテンプレート
- コンポーネントを組み立てて完全な動作システムを構築する方法を示すサンプルプロジェクト

CMSIS-Packs は、さまざまなシリコンおよびソフトウェアベンダによって開発されており、数千種類のボードやデバイスをカバーしています。また、社内ソフトウェアコンポーネントのライフサイクル管理にも使用できます。

詳細については Open-CMSIS-Pack のドキュメントを参照してください。

新しい CMSIS-Packs については、keil.arm.com/packs で確認してください。 csolution.yml ファイルに Pack を追加したり、cpackget add を使用して Pack をインストールしたりするためにコピーできるスニペットが各 Pack で用意されています。

5.5 CMSIS-Packs のインストール

keil.arm.com から入手できるサンプルからスタートした場合、サンプルに必要な CMSIS-Packs は、csolution.yml ファイルの packs キーの下に既に登録されています。CMSIS Solution extension は Pack キャッシュをスキャンし、不足している Pack をインストールするよう提案します。詳細については、不足している CMSIS-Packs のインストール の章を参照してください。

サンプル solution に CMSIS-Packs を追加する必要がある場合、または solution を最初から作成する場合は、keil.arm.com に存在する CMSIS-Packs を調べることができます。詳細については、利用可能な CMSIS-Packs を調べる の章を参照してください。

パブリックな Pack とプライベートな Pack の違い、およびコマンドラインからパックを管理する方法については、Pack のサポート の章を参照してください。

5.5.1 不足している CMSIS-Packs のインストール

solution で不足している CMSIS-Packs をインストールします。

手順

1. **Explorer** ビュー  から solution の`*.csolution.yml` ファイルを開きます。必要な Pack は、`csolution.yml` ファイルの `packs` キーの下に登録されています。1 つ以上の CMSIS-Packs が不足している場合は、**Problems** ビューにエラーが表示され、右下隅に"Solution [solution-name] requires some packs that are not installed" というポップアップ メッセージが表示されます。
2. **Install** をクリックします。
   または、**Problems** ビューでエラーを右クリックし **Install missing pack** を選択します。不足している Pack が複数ある場合は、**Install all missing packs** を使用します。

   Command Palette から **CMSIS: Install required packs for active solution** コマンドを使用して、不足している Pack をインストールすることもできます。

5.5.2 利用可能な CMSIS-Packs を調べる

keil.arm.com から入手可能な CMSIS‑Packs を調べてインストールします。

手順

1. keil.arm.com の CMSIS‑Packs のページに移動します。
2. Pack を検索し、**Result** の一覧から選択します。たとえば、`wolfSSL` と入力します。
3. `packs` スニペットをコピーし、 Visual Studio で `csolution.yml` ファイルの `packs` キーを更新します。

図 5-1: wolfSSL サンプル

4. 右下隅に表示されるポップアップメッセージ内の **Install** をクリックして Pack をインストールします。

5.6 Software Components の管理

Software Components ビューには、solution のアクティブな project 内で選択されているすべての software components が表示されます。

このビューでは、Open-CMSIS-Pack のドキュメント内で attributes と呼ばれるすべての component の詳細を確認できます。

以下のタスクも実行できます:

- project に含める software components を変更し、solution で定義されている各 target type の components 間、またはすべての target type 間で依存関係を管理する
- Pack と component のバージョンの異なる組み合わせ、および異なるツールチェインのバージョンを使用した solution のビルド

5.6.1 Software Components ビューを開く

Software Components ビューを開く方法について説明します:

手順

1. **Activity Bar** の **CMSIS** をクリックして、**CMSIS** ビューを開きます。

> Note
> 以前に solution の context 内のすべての project をクリアした場合、 Open Build Context Editor をクリックして Build Context ビューを開き、project を選択できます。 Solution の context を設定する の章を参照してください

2. **Solution outline** 上にカーソルを移動し、project レベルで **Manage software components** をクリックします。

結果
Software Components ビューが開きます。

デフォルトビューには、solution でリストされている Pack から使用可能な component が表示されます。(**Software packs: Solution-name** ドロップダウンリストと **All** トグルボタン)

ビューに component が表示されない場合は、**Install Missing Packs** をクリックして問題を解決してください。

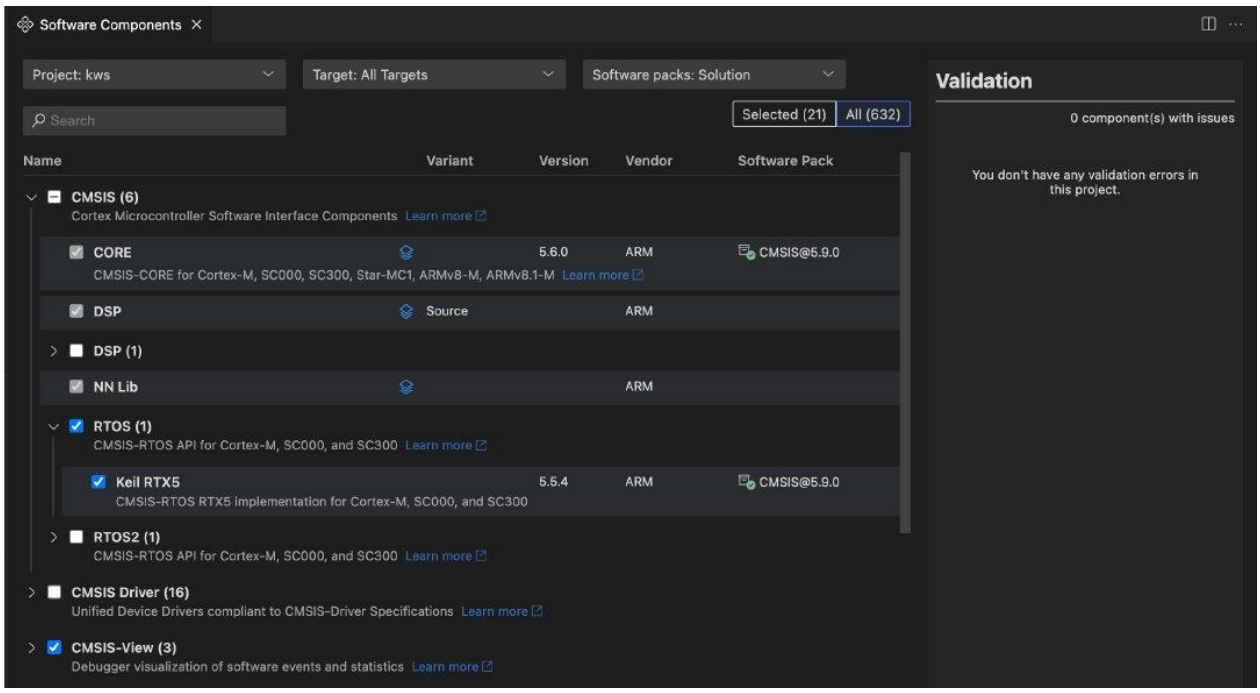
Search フィールドを使用して component のリストを検索できます。

Project ドロップダウンリストで、Software Component を変更する project を選択します。

Target ドロップダウン リストで、**All Targets** または特定の target type を選択して、solution 内のすべての target type の Software components を一度に変更するか、特定のターゲットのみを変更します。

Software Packs のドロップダウンリストを使用すると、Solution にリストされている Pack から使用可能なコンポーネントをフィルタリングしたり、CMSIS エコシステムで使用可能なすべての Pack のコンポーネントを表示したりできます。

図 5-1: Solution にリストされている Pack から利用可能なすべてのコンポーネントを表示する'Software Components' ビュー



CMSIS-Pack の仕様では、各 software components には次の属性が必要であると規定されています。

- Component class (Cclass) ： トップレベルの cmponent 名(例 **CMSIS**)
- Component group (Cgroup) ： component グループ名 (例 **CMSIS** component クラスの **CORE**)
- Component version (Cversion) ： software component のバージョン番号

オプションで、software component には次の追加属性が含まれる場合があります:

- Component subgroup (Csub) :複数の互換性のある component が混在する場合に使用されるされる component のサブグループ(例 **CMSIS** > **RTOS2** 下の **Keil RTX5**)
- Component variant (Cvariant) :一般的にはたとえば **Keil RTX5** の **Library** のように、同一の実装に複数のトップレベルのコンフィギュレーションがある場合に使用される software component のバリアント
- Component vendor (Cvendor) ： software component のサプライヤ(例 ARM)
- Bundle (Cbundle) :複数の software components を 1 つの software bundle にまとめることができます。バンドルには、使用可能な component の異なるセットがあります。バンドル内のすべての component は互いに互換性がありますが、別のバンドルの component とは互換性がありません。たとえば、 **Compiler** component クラスにおける **ARM Compiler** が該当します。

Layer  アイコンは layer で使用されている component を表示します。現在のバージョンでは、layer が read-only であるため、**Software Components** ビューで選択やクリアを行うことができません。component の Layer アイコンをクリックすると`*.clayer.yml` やその他の関連するファイルが開かれます。
一部の component では、class、group、または subgroup レベルでドキュメントへのリンクがあります。component の **Learn more** のリンクをクリックすると、関連するドキュメントが開きます。

5.6.2 プロジェクトの Software Components を変更する

project に対し、すでに選択されている Pack だけでなく、利用可能なすべての Pack から component を追加できます。

手順

1. **Project** ドロップダウンリストで、software component を変更したい project を選択します。
2. **Target** ドロップダウンリストで、特定の target type を選択するか、または一度にすべての target type を変更したい場合は **All Targets** を選択します。(一部の例では target は 1 つだけになっているのに注意してください)
3. **Software packs** のドロップダウンリストでは、利用可能なコンポーネントをフィルタリングできます。solution にリストされている Pack を表示するか(**Solution: Solution-name** オプション)、CMSIS エコシステムで利用可能なすべての Pack の component を表示します。(**All packs** オプション)
4. 利用可能なすべての component を表示する場合は **All** トグルボタンが選択されていることを確認しますが、Selected に切り替えるとすでに選択されている component のみが表示されるようになります。
5. 必要に応じて、チェックボックスで component を選択またはクリアします。一部の compoment では、vendor、variant、または version を選択することもできます。`cproject.yml` ファイルは自動的に更新されます。
6. component 間の依存関係を管理し、**Validation** パネルで検証上の問題を解決します。問題の個所は赤で強調表示され、エクスクラメーションのアイコン  が隣に表示されます。サジェストリストを参照し、競合する component を削除するか、不足している component を追加します。

7. 検証上の問題がある場合は、**Validation** パネル上にカーソルを移動することで詳細が表示されます。提案された修正をクリックすると、リスト内の component が見つかります。場合によっては、異なる修正の方法を選択する必要があります。ドロップダウンリストから修正の方法を選択し、必要な component を選択してから、**Apply** をクリックします。
   solution に Pack がない場合は、"Component's pack is not included in your solution" というメッセージが **Validation** パネルに表示されます。また、**Problems** ビューにもエラーが表示されます。
   CMSIS-Packs のインストール方法については、CMSIS-Packs のインストール の章を参照してください。

   次のような問題が発生することもあります:
   - 選択した component が、選択したハードウェアおよびツールチェインと互換性がない
   - 選択した component で、選択したハードウェアおよびツールチェインと互換性のない依存関係がある
   - 選択した component に解決できない依存関係がある。このような場合は、component を削除し、**Validation** パネルから **Apply** をクリックします。

5.6.3 変更を元に戻す

現在のバージョンでは、Source Control ビューから、または cproject.yml ファイルを直接編集して変更を元に戻すことができます。

5.7 Configuration Wizard の使用

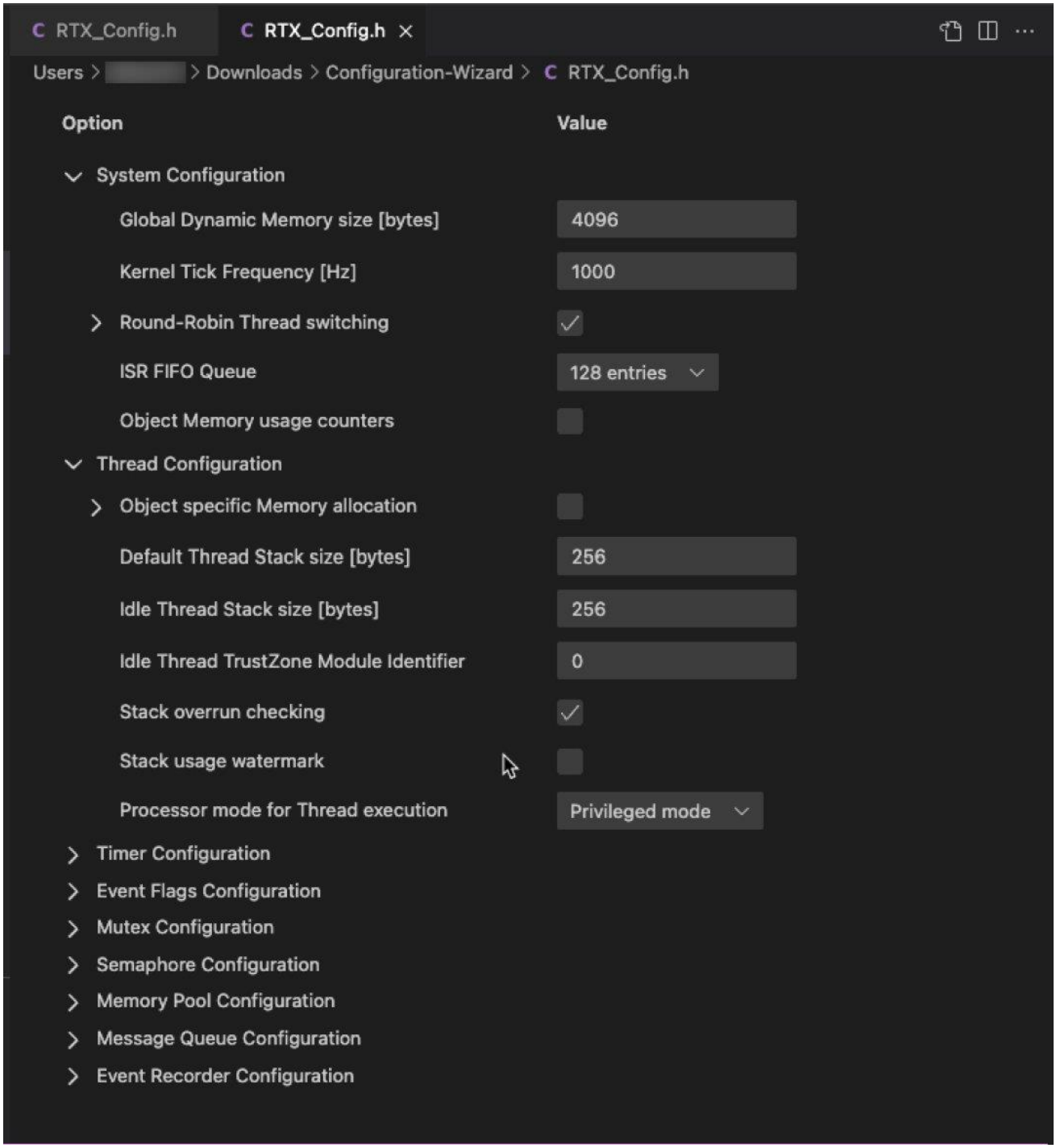
Configuration Wizard は、直感的なダイアログスタイルのインターフェイスを提供することで、スタートアップコードと configuration ファイルのカスタマイズを簡略化します。これにより、手動での大規模な編集を必要とせずに、configuration の設定を迅速に変更できます。

Configuration Wizard インターフェイスは、configuration ファイル自体に含まれるアノテーションから生成されます。

これらのアノテーションは、埋め込まれたマークアップタグのようなもので、project で使用される software component の configuration ファイルで既に有効になっている場合もあれば、自分で追加することもできます。これらのアノテーションを作成するための一連のルールについては、Open-CMSIS-Pack ドキュメンテーション内 Configuration Wizard Annotations を参照してください。

Configuration Wizard を開くには、アノテーションを含む configuration ファイルを開き、**Open Preview**  をクリックします。これにより Configuration Wizard が表示されます。

図 5-3：Configuration Wizard



Configuration Wizard で行った変更は、ソースファイルに即座に反映されます。ソースファイルを直接編集することもできます。

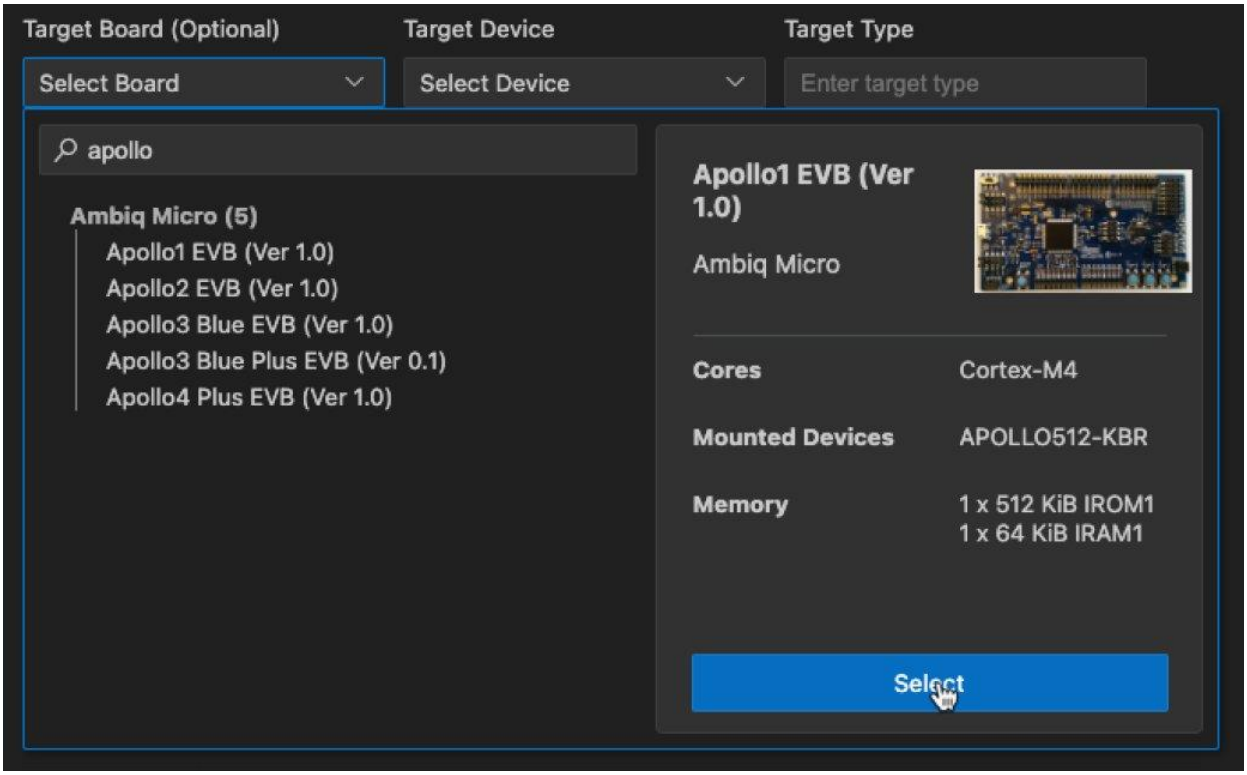
5.8 Solution の作成

solution project を最初から作成します。

手順

1. solution を作成するには、次のいずれか実行します:
   - Activity Bar の **CMSIS** をクリックして、**CMSIS** ビューを開きます。続いて:
     - 空のワークスペースから開始する場合は、**Create a New Solution** をクリックします。
     - ワークスペースで既に solution を開いており、同じワークスペースに新しい solution を作成する場合は、**Solution outline** にカーソルを移動し、**Views and More Actions** ··· > **New Solution** をクリックします。
   - **File** メニューから **New File…** を選択し、ドロップダウンリストから **New Solution** を選択します。

   Create New Solution ビューが開きます。
2. **Target Board** ドロップダウンリストをクリックし、検索語を入力してボードを選択します。
   ピッカーには、選択したボードの詳細が表示されます。



3. **Select** をクリックします。
   Target Device ドロップダウンリストと **Target Type** フィールドには、選択したボードに実装されているデバイスの名前がデフォルトで入力されます。

   あるいは、Target Device ドロップダウンリストでデバイスを直接選択することもできます。

4. **Target Type** フィールドでは、solution にデプロイするために使用するターゲットハードウェアの名前をカスタマイズできます。**Target Type** は **Build Context** ビューに表示されるとともに、`<solution_name>.csolution.yml` ファイルにセットされます。(`target-types: - type:`)
5. **Templates, Reference Applications, and Examples** ドロップダウンリストから以下のいずれかを選択します:
   - Create a blank solution
   - Create a TrustZone solution　TrustZone は、Arm ベースのプロセッサ上でセキュアな実行環境を提供するハードウェアベースのセキュリティ機能です。セキュアなゾーンと非セキュアなゾーンを分離し、機密データやアプリケーションのセキュアな処理を可能にします。選択したボードまたはデバイスで互換性がある場合は、solution が TrustZone を使用するかどうかや、solution 内のどの project でセキュアまたは非セキュアのゾーンを使用するかを定義できます。
   - Use a CMSIS solution example as a starting point

   利用可能なオプションは、選択したボードまたはデバイスによって異なります。利用可能なサンプルが複数ある場合は、検索語を入力してからサンプルを選択してください。
6. blank および Trustzone の solution の場合のみ、solution 内の project の configuration が必要です。
   - `Blank solution` を選択した場合: ターゲットハードウェアのプロセッサごとに 1 つの project が追加されます。project 名は変更できます。ボードまたはデバイスで互換性がある場合は、**TrustZone** ドロップダウンリストを使用して `secure` または `non-secure` のゾーンの追加を決定できます。デフォルトでは、TrustZone はオフになっています。
   - `TrustZone solution` を選択した場合: TrustZone solution を選択した場合: TrustZone をサポートするターゲットハードウェアのプロセッサごとに、2 つの project (secure project と non-secure project) が追加されます。project 名は変更できます。また、TrustZone ドロップダウンリストでゾーン (`secure` または `non-secure`) を変更したり、オフを選択して TrustZone を解除したりすることもできます。
7. Add Project をクリックして、solution に project を追加し、configuration を行います。TrustZone の場合、特定のプロセッサに必要な数だけ `secure` または `non-secure` project を追加できます。
8. blank および TrustZone solution の場合のみ、コンパイラを選択します: **Arm Compiler 6**、**GCC**、または **LLVM**
9. **Solution Name** フィールドで solution の名前を変更できます。
10. **Solution Location** フィールドの横にある **Browse** をクリックし、表示されるシステムダイアログボックスで solution ファイルを保存する場所を選択します。
11. **Initialize Git repository** チェックボックスをオンにすると、solution を Git リポジトリとして初期化できます。
    solution を Git リポジトリに変換しない場合は、チェックボックスをオフにします。
12. **Create** をクリックします。
    extension により solution が作成されます。`*.uvprojx` 形式でのみ提供されているサンプルは自動的に変換されます。変換エラーがある場合は、利用可能な `uv2csolution.log` ファイルを確認してください。

    ダイアログボックスが表示されます。次のタスクを実行できます:
    - solution を新しい workspace で開く(**Open** オプション)

- solution を新しいウィンドウと新しい workspace で開く(**Open project in new window** オプション)
- 現在のワークスペースの solution を追加する(**Add project to vscode workspace** オプション)

13. いずれかのオプションを選択します。
    この extension は、開発環境をセットアップするために必要なツールの情報ををを含む `vcpkg-configuration.json` ファイルも生成します。
    Arm Environment Activation ダイアログボックスが開きます。
14. **Environment Manager** extension がワークスペースを自動的にアクティブ化し、`vcpkg-configuration.json` ファイルで指定されたツールがダウンロードされることを確認します。
    不足している CMSIS-Packs は自動的にインストールされます。インストールを確認するポップアップメッセージが右下隅に表示されます。
15. solution のファイルが作成されたことを確認します:
    - `vcpkg-configuration.json` ファイル
    - `<solution_name>.csolution.yml` ファイル
    - 別々のフォルダ内に存在する、1 つまたはそれ以上の`<project_name>.cproject.yml` ファイル
    - 各 project に対し `main.c` のテンプレート

次のステップ

`csolution.yml` および `cproject.yml` ファイルの編集に使用できるオートコンプリート機能を確認します。project サンプルの CMSIS-Toolbox > Build Overview のドキュメントを参照してください。**Software Components** ビューを使用して CMSIS component を追加します。component を追加すると、`cproject.yml` ファイルが更新されます。

5.9 Keil uVision project を solution に変換する

Compiler 5 project は Web サイトからダウンロードできますが、extension で使用することはできません。変換できるのは Arm Compiler 6 project のみです。回避策として、Keil uVision で Arm Compiler 5 project を Arm Compiler 6 にアップデートしてから、Visual Studio Code で project を solution に変換できます。Arm Compiler 6 への変換に関する詳細なヘルプと情報については、Migrate Arm Compiler 5 to Arm Compiler 6 application note および Arm Compiler for Embedded Migration and Compatibility Guide を参照してください。

手順

1. 変換したい`*.uvprojx` を含むフォルダを Visual Studio Code ワークスペースにドラッグアンドドロップして開いてください。または、keil.arm.com から uVision project をインポートするか、GitHub から project をクローンすることもできます。
2. *.uvprojx を右クリックし、**Explorer** で **Convert uvproj to csolution** を選択します。
   変換はすぐに開始されます。

   あるいは、空のワークスペースから開始した場合は、**Activity Bar** の **CMSIS**  をクリックして **CMSIS** ビューを開きます。次に、以下の 2 つのオプションのいずれかを選択します:
   - **Convert a uVision Project** をクリックし、`*.uvprojx` ファイルを開いて変換します。
   - カーソルを **Solution outline** に移動し、**Views and More Actions** をクリックして **Convert uvproj to csolution** を選択し、`*.uvprojx` ファイルを開いて変換します。

ダイアログボックスが表示されます。次のタスクを実行できます:

- solution を新しい workspace で開く(**Open** オプション)
- solution を新しいウィンドウと新しい workspace で開く(**Open project in new window** オプション)

Command Palette から **CMSIS: Convert uvproj to csolution** コマンドを実行することもできます。その場合は、マシン上で変換する`*.uvprojx` を選択し、**Select** をクリックします。

変換エラーがある場合は、利用可能な `uv2csolution.log` ファイルを確認してください。

3. **Environment Manager** extension がワークスペースを自動的にアクティブ化し、`vcpkg-configuration.json` ファイルで指定されたツールがダウンロードされることを確認します。
4. **Output** タブ(**View** > **Output**) を確認します。**Output** タブの右側にあるドロップダウンリストで、**uVision to Csolution Conversion** を選択します。
   `*.cproject.yml` および`*.csolution.yml` ファイルは、`*.uvprojx` が保存されているフォルダにあります。

5.10 Build task の設定

Visual Studio Code では、`tasks.json` というファイルを設定することで、特定のタスクを自動化できます。詳細については Integrate with External Tools via Tasks を参照してください。

CMSIS Solution extension を使用すると、`tasks.json` のファイルを使用してビルドタスクを設定し、project をビルドできます。ビルドタスクを実行すると、extension は定義されたオプションを使用して `cbuild` を実行します。

サンプル project を使用した Getting Started で説明したように、keil.arm.com で提供されているサンプルには、プロジェクトをビルドするための設定がすでに含まれた `tasks.json` ファイルが付属しています。必要に応じて、デフォルトの configuration を変更できます。

ビルドタスクがまだ configuration されていないサンプルを使用している場合は、次の手順に従ってください:

1. **Terminal** > **Configure Tasks…** に移動します。
2. ウィンドウの上部に表示されているドロップダウンリストで、CMSIS Build task を選択します。
   `tasks.json` がデフォルトの configuration で開きます。
3. Configuration を修正します。

   IntelliSense を使用すると、`tasks.json` ファイルで使用可能なタスクプロパティと値の完全なセットを表示できます。Command Palette から **Trigger Suggest** を使用してサジェスチョンを表示できます。
   Terminal で `cbuild --help` と入力して、cbuild 固有のタスク プロパティを表示することもできます。
4. `tasks.json` のファイルを保存します。

または **Terminal** > **Configure Default Build Task...** を使用してデフォルトのビルドタスクを定義することもできます。
Terminal > **Run Build Task...** オプションは、デフォルトのビルドタスクの実行をトリガします。

5.11 Solution の初期化

`vcpkg-configuration.json` ファイル、`tasks.json` ファイル、および `launch.json` ファイルがまだ含まれていない solution の場合は、**Initialize CMSIS project** オプションを使用してこれらのファイルを生成し、project の作業を開始できます。keil.arm.com のサンプルや、**Create New Solution** ビューで最初から作成された solution には、必要な JSON ファイルが既に含まれています。

手順

1. Explorer からワークスペースを開きます。
2. ワークスペース内の任意の場所で右クリックし、**Initialize CMSIS project** を選択します。
   extension は、configuration 済みの `vcpkg-configuration.json` ファイル、`tasks.json` ファイル、および `launch.json` ファイルを生成します。

5.12 CMSIS csolution API の使用

独自の Visual Studio Code csolution extension を作成する場合、CMSIS Solution extension 他の extension が使用できる API を公開しています。

API の仕様については、CMSIS csolution extension API のページを参照してください。

extension の作成については、Visual Studio Code ドキュメントの Extension API の章を参照してください。
solution のサンプルについては、keil.arm.com を参照してください。

6. Arm Device Manager extension

Keil Studio extensions でサポートされているハードウェアを確認します。

次に、Device Manager を使用してハードウェアを管理します:

- ハードウェアへの接続
- ハードウェア接続の編集
- Serial モニタを開く

Arm Device Manager extension のコマンドと設定の完全なリストがあります。リストを確認するには、Visual Studio Code Activity Bar の **Extensions**  をクリックします。extensions のリストで **Arm Device Manager** をクリックしてから **Features** をクリックします。

6.1 サポートされるハードウェア

Device Manager extension およびその他の Keil Studio extensions がサポートするハードウェアについて説明します。

6.1.1 サポートされる開発ボードとデバイス

extensions は keil.arm.com にある開発ボードおよび MCU をサポートしています。

6.1.2 サポートされるデバッグプローブ

以下のデバッグプローブをサポートしています。

6.1.2.1 WebUSB 対応 CMSIS-DAP デバッグプローブ

extension は、次のような CMSIS-DAP protocol を実装するデバッグプローブをサポートします:

- DAPLink 実装: ARMmbed/DAPLink リポジトリを参照
- LPC-Link2 実装: LPC-Link2 のドキュメントを参照
- Nu-Link2 実装: Nuvoton リポジトリを参照
- ULINKplus(firmware version 2) 実装: Keil MDK のドキュメントを参照

一般的な情報については CMSIS-DAP のドキュメントを参照してください。

6.1.2.2 ST-LINK デバッグプローブ

extension は、ST-LINK/V2 以降のプローブと、これらのプローブで使用可能な ST-LINK ファームウェアをサポートします。

ST-LINK ファームウェアのデバッグ実装の推奨バージョンは次のとおりです:

- ST-LINK/V2 および ST-LINK/V2-1 プローブ用: J36 およびそれ以降
- STLINK-V3 プローブ用: J6 およびそれ以降

ネーミングルールの詳細については Overview of ST-LINK derivatives の"Firmware naming rules" を参照してください。

6.2 ハードウェアへの接続

ハードウェアを初めて接続する方法について説明します。

手順

1. Activity Bar で **Device Manager**  をクリックします。
2. USB 経由でコンピュータをハードウェアに接続します。
   ハードウェアが検出され、右下隅にポップアップメッセージが表示されます。
3. ハードウェアを使用するには、**OK** をクリックします。
   または、**Add Device** 

   をクリックしてウィンドウ上部に表示されるドロップダウンリスト内でハードウェアを選択します。
   これで、ハードウェアを使用してプロジェクトを実行およびデバッグする準備が整いました。

次のステップ

ハードウェアをさらに追加する必要がある場合は、右上隅にある **Add Device**  をクリックします。

6.3 ハードウェア接続の編集

ボードが検出されない、または外部デバッグプローブを使用している場合は、Device Manager からハードウェアエントリを編集し、Device Family Pack(DFP) と Pack から取得したデバイス名を指定して、ハードウェアで動作するようにできます。DFP はデバイスサポートを処理します。

手順

1. 編集するハードウェアの上にカーソルを移動し、**Edit Device**  をクリックします。
2. 必要に応じて、ウィンドウの上部に表示されるフィールドでハードウェア名を編集し、**Enter** キーを押します。この名前が Device Manager に表示されます。
3. ドロップダウンリストから、ハードウェアの Device Family Pack(DFP) CMSIS-Pack を選択します。
4. フィールドの CMSIS-Pack から使用するデバイス名を選択し、**Enter** キーを押します。

6.4 Serial モニタを開く

Serial モニタを開きます。Serial output にはボードの出力が表示されます。Serial output はデバッグツールとして、あるいはボードと直接通信するために使用できます。

手順

1. Serial monitor を開きたいハードウェアの上にカーソルを移動し、**Open Serial**  をクリックします。
   ウィンドウの上部にドロップダウンリストが表示され、ボーレート(コンピュータとハードウェア間のビット／秒単位のデータレート) を選択できます。ハードウェア出力を正しく表示するには、適切なボーレートを選択しなければなりません。選択するボーレートは、アクティブな project のボーレートと同じである必要があります。
2. ボーレートを選択します。
   ボーレートが選択された状態で **Terminal** タブが開きます。

7. Arm Debugger extension

Arm Debugger を使用してハードウェア上で project を実行し、Arm Debugger extension を使用して Arm Debugger debug session を開始します。

Arm Debugger エンジンのコマンドの完全なリストと使用方法およびサンプルについては、Arm Debugger Command Reference guide を参照してください。extension の **Features** タブ(Windows) または **Feature Contributions** タブ(macOS)でコマンドと設定のリストを表示することもできます。

Visual Studio Code Activity Bar の **Extensions**  で、extensions の一覧から **Arm Debugger** をクリックし、**Features** または **Feature Contributions** をクリックします。

keil.arm.com で提供されているほとんどのサンプルでは、実行および debug configuration を含む、**tasks.json** ファイルと **launch.json** ファイルが付属しています。必要に応じて、デフォルトのコンフィギュレーションを変更できます。

7.1 Arm Debugger を使用してハードウェア上で project を実行する

ハードウェア上で project を実行するための task をコンフィギュレーション する方法と、configuration オプションについて説明します。

7.1.1 task をコンフィギュレーションする

ハードウェア上で project を実行するには、まず task をコンフィギュレーションする必要があります。task は、バイナリをハードウェアのフラッシュメモリ上の適切なメモリ位置に転送します。

`arm-debugger.flash: Flash Device` の task を使用します。project で使用される CMSIS-Packs がフラッシュダウンロードを制御します。

keil.arm.com で提供されているほとんどのサンプルには、run configuration を含む **tasks.json** のファイルが付属しています。必要に応じて、デフォルトの configuration を変更できます。

手順

1. Command Palette を開きます。`Tasks: Configure Task` を探して選択します。
   または、**Terminal** メニューに移動して、**Configure Tasks...** を選択します。
2. `arm-debugger.flash: Flash Device task`(または **Flash Device**)を選択します。
   この task は、project の `.vscode` フォルダ内の `tasks.json` のファイルにデフォルトの run configuration オプションを追加します。
3. `tasks.json` ファイルを保存します。

7.1.2 Arm Debugger のデフォルトの run configuration オプションを上書きまたは拡張する

デフォルトの configuration オプションを上書きまたは拡張できます。Arm Debugger run configuration オプションの章を参照してください。

ハードウェアデバイスをフラッシュするには、task configuration で、どの CMSIS-Pack から情報を読み取るかと、CMSIS-Pack で使用するデバイス名を認識する必要があります。これらの設定は `cmsisPack` および `deviceName` という名称を持ち、複数の方法で指定できます。

ターゲットハードウェアが自動的に検出されるか、またはハードウェアの Pack とデバイス名が設定されている場合は、次のコードを使用して task configuration でこれらを自動的に取得できます:

```
{
    [...]
    "serialNumber": "${command:device-manager.getSerialNumber}",
    "cmsisPack": "${command:cmsis-csolution.getTargetPack}",
    "deviceName": "${command:device-manager.getDeviceName}",
    [...]
}
```

あるいは、これらの設定を CMSIS-Pack のファイルまたはマシン上のフォルダへのフルパスとして直接指定することもできます:

```
{
    [...]
    "serialNumber": "${command:device-manager.getSerialNumber}",
    "cmsisPack": "/Users/me/mypack.pack",
    "deviceName": "STM32H745XIHx",
    [...]
}
```

CMSIS-Pack の短縮コードを<vendor>::<pack>@<version> の形式で使用することもできます:

```
{
    [...]
    "serialNumber": "${command:device-manager.getSerialNumber}",
    "cmsisPack": "Keil::STM32H7xx_DFP@3.1.0",
    "deviceName": "STM32H745XIHx",
    [...]
}
```

このコードは CMSIS-Pack の自動ダウンロードを引き起こすことに注意してください。

もしインストール済みの CMSIS Solution extension がない場合、以下を利用できます:

- **`"cmsisPack": "${command:cmsis-csolution.getTargetPack}"`**
  の代わりに
  `"cmsisPack": "${command:device-manager.getDevicePack}"`
- **`"deviceName": "${command:cmsis-csolution.getDeviceName}"`**
  の代わりに
  `"deviceName": "${command:device-manager.getDeviceName}"`

7.1.3 Arm Debugger run configuration オプション

extension では以下の run configuration オプションを提供します。

configuration オプション	説明
"**`cmsisPack`**"	ハードウェアの DFP(Device Family Pack) CMSIS-Pack へのパス(ファイルまたは URL) 以下が使用できます: • **`cmsis-csolution.getTargetPack`**:**CMSIS Solution Build Context** ビューで選択されたターゲットタイプの DFP CMSIS-Pack を取得します。**`cmsis-csolution.getTargetPack`** は solution 固有です。 • **`device-manager.getDevicePack`**:選択したデバイスの DFP CMSIS-Pack を取得します。このコマンドは pack index 内で利用可能な最新の pack を使用します。
"**`connectMode`**"	接続モードです。 選択可能な値: • **`auto`**: デバッガが決定します。 • **`haltOnConnect`**: 実行前にリセットのため停止します。 • **`underReset`**: 外部 NRST ラインをアサートしたままにします。 • **`preReset`**: NRST を使用してプリリセットを行います。 • **`running`**: 状態を変更せずに実行中のターゲットに接続します。 デフォルト: **`auto`**
"**`dbgconf`**"	CMSIS-Pack デバッグシーケンスの実行をコンフィギュレーションする **`.dbgconf`** ファイルへのパス。Arm Debugger v6.1.0 以降が必要です。
"**`debugClockSpeed`**"	デバッグ接続の最大周波数。実際に使用される周波数はデバッグプローブによって異なります。auto はターゲット固有のデフォルトを使用します。Arm Debugger v6.0.2 以降が必要です。 選択可能な値:**`auto`**, **`50MHz`**, **`33MHz`**, **`25MHz`**, **`20MHz`**, **`10MHz`**, **`5MHz`**, **`2MHz`**, **`1MHz`**, **`500kHz`**, **`200kHz`**, **`100kHz`**, **`50kHz`**, **`20kHz`**, **`10kHz`**, **`5kHz`** デフォルト: **`auto`**

Configuration オプション	説明
"**debugPortMode**"	デバッグ接続に使用するデバッグポートモード。Arm Debugger v6.0.2 以降が必要です。 選択可能な値:**auto**, **JTAG**, **SWD** デフォルト: **auto**
"**deviceName**"	CMSIS-Pack デバイス名 以下とともに使用できます: • **cmsis-csolution.getDeviceName**: ソリューションの ***.csolution.yml** ファイル内のプローブまたはボードで利用可能な情報からデバイス名を取得します。 • **device-manager.getDeviceName**: 選択したデバイスの DFP からデバイス名を取得します。
"**eraseMode**"	使用するフラッシュ消去のタイプ。Arm Debugger v6.1.0 以降が必要です。 選択可能な値: **sectors**: プログラムするセクタのみを消去します。 **none**: フラッシュ消去をスキップします。 デフォルト: **sectors**
"**flms**"	フラッシュアルゴリズムのコンフィギュレーションです。各エントリは、対応する DFP(Device Family Pack) で定義されているデフォルトのアルゴリズムを変更するか、さらにアルゴリズムを追加します。DFP からデフォルトのアルゴリズムを非アクティブ化するには、'**ignore**' のフィールドを使用します。Arm Debugger v6.1.0 以降が必要です。 要求される値: • "**path**": DFP 内のフラッシュアルゴリズムファイルへの相対パス、またはマシンのファイルシステム内にあるフラッシュアルゴリズムファイルへの絶対パス オプションの値: • "**regionStart**": フラッシュアルゴリズムの対象となるメモリ領域の 10 進数または 16 進数フォーマットでの開始アドレス。設定されていない場合は、DFP CMSIS-Pack のアルゴリズムのデフォルトの開始アドレスを使用します。 • "**regionSize**": フラッシュアルゴリズムの対象となるメモリ領域の 10 進数または 16 進数フォーマットでのサイズ。設定されていない場合は、DFP CMSIS-Pack のアルゴリズムのデフォルトのサイズを使用します。 • "**ramStart**": フラッシュアルゴリズムの実行に使用されるターゲットシステムの RAM の開始アドレス。設定されていない場合は、DFP CMSIS-Pack のアルゴリズムのデフォルトを使用します。 • "**ramSize**": フラッシュアルゴリズムの実行に使用されるターゲットシステムの RAM のサイズ。設定されていない場合は、DFP CMSIS-Pack のアルゴリズムのデフォル

	トを使用します。 このオプションを使用すると次のことができます: ➢ DFP のデフォルトアルゴリズムを無効にします。たとえば、デフォルトアルゴリズムをローカルのバージョンで上書きします。 ➢ デフォルト以外のアルゴリズムを有効にします。たとえば、ターゲットボードの設計に依存する外部フラッシュメモリの場合などです。 アルゴリズムは、大多数のユースケースに適用できると予想される場合、通常は DFP でデフォルトとしてマークされます。
"**openSerial**"	フラッシュ後にデバイスのシリアル出力を開くためのボーレート(Arm Device Manager が必要です)。 選択可能な値: **115200, 57600, 38400, 19200, 9600, 4800, 2400, 1800, 1200, 600**
"**pdsc**"	PDSC ファイルへのパス(ファイルまたは URL)
"**probe**"	デバッグ接続に使用するプローブの名前 選択可能な値: **ULINKpro, ULINKpro D, ULINK2, CMSIS-DAP, ULINKplus, ST-Link** デフォルト: **CMSIS-DAP**
"**processorName**"	マルチコアデバイスの CMSIS-Pack プロセッサ名
"**program**" または "**programs**"	ロード順で使用する 1 つ以上の project へのパス(ファイルまたは URL)。Arm Debugger v6.0.2 以降が必要です。 以下とともに使用できます: • **arm-debugger.getApplicationFile**: CMSIS の実行とデバッグに使用される AXF または ELF ファイルを返します。
"**serialNumber**"	使用する接続された USB ハードウェアのシリアル番号 以下とともに使用できます: • **device-manager.getSerialNumber**: 選択したデバイスのシリアル番号を取得します。
"**targetAddress**"	**serialNumber** と同義です。
"**targetInitScript**"	接続後、他の操作の前に実行されるターゲット初期化スクリプト(**.ds/.py**) へのパス。Arm Debugger v6.1.1 以降が必要です
"**vendorName**"	CMSIS-Pack ベンダ名
"**verifyFlash**"	フラッシュにダウンロードされた内容をベリファイします。Arm Debugger v6.1.0 以降が必要です。
"**workspaceFolder**"	現在の Arm Debugger のワークスペースフォルダ。 デフォルト:"**${workspaceFolder}**"

Visual Studio Code のその他のオプションも利用できます。**Trigger Suggestions** コマンド(**Ctrl+Space**)を使用して、利用可能なオプションを確認し、タスクに関する Visual Studio Code のドキュメントおよび tasks.json の Schema のページをご覧ください。

7.1.4 Run and Debug Configuration visual editor を使った run configuration

solution の tasks.json ファイルを編集して run configuration オプションを変更する代わりに、Run and Debug Configuration visual editor を使用することもできます。

手順

1. editor を開くには以下のいずれかを実行します。
   - **Explorer** 上で solution の **`.vscode`** フォルダに保存されている **`tasks.json`** ファイルを右クリックし、**Open Run and Debug Configuration** を選択します。
   - **Explorer** 上で、**`tasks.json`** ファイルを右クリックして **Open With...** を選択し、ウィンドウの上部に表示されるドロップダウンリストから **Run and Debug Configuration** を選択します。
   - editor で **`tasks.json`** のファイルをすでに開いている場合は、右上隅にある **Open Run and Debug Configuration**  をクリックします。
2. **`tasks.json`** ファイルでは、複数の run configurations を定義できます。**Selected Configuration** ドロップダウンリストで、**New Configuration** を選択して JSON ファイルに新しい configuration block を追加します。また、**Duplicate** をクリックして、現在選択されている configuration を複製し、変更することもできます。
3. run configuration を修正します:
   - **Configuration Name** フィールドで configuration の名前を変更できます。
   - **Probe Type**: ドロップダウンリストで使用しているデバッグプローブか、ボード上のデバッグユニットのタイプを選択します。
     - デフォルト値: **`CMSIS-DAP`**。Arm Debugger extension が自動的に probe type を設定できない場合、デフォルト値は **`CMSIS-DAP`** となります。
     - プローブまたはボードを USB 経由でコンピュータに接続した場合、Arm Debugger extension は検出されたハードウェアのシリアル番号に基づいて probe type を設定します。
   - **Serial Number**: ドロップダウンリストで使用しているデバッグプローブか、ボード上のデバッグユニットのシリアル番号を選択します。
     - デフォルト値: **`auto`**。**`auto`** の場合、Arm Device Manager extension 内でアクティブなデバイスのシリアル番号がデフォルトで使用されます。JSON ファイルに **`serialNumber`** のための "**`${command:device-manager.getSerialNumber}`**" コマンドが追加されます。
     - ドロップダウンリストから選択するか、あるいは直接シリアル番号を入力することでアクティブなデバイスのシリアル番号を選択することもできます。

     アクティブなデバイスを確認するには **Open Arm Device Manager** をクリックします。
   - **CMSIS-Pack**: ターゲットデバッグプローブまたはボードの Device Family Pack(DFP) を選択します。
     - デフォルト値: **`auto (CMSIS Solution)`**。solution の**`*.csolution.yml`** ファイルで定義されているアクティブなデバイスの DFP がデフォルトで使用されます。**`CMSIS-Pack`** のための JSON ファイルに"**`${command:cmsis-csolution.getTargetPack}`**"コマンドが追加され

ます。

- ー **auto** (**Device Manager**): Arm Device Manager extension 内のアクティブなデバイスの DFP がデフォルトで使用されます。**CMSIS-Pack** のための JSON ファイルに "**${command:device-manager.getDevicePack}**"コマンドが追加されます。
- ー ドロップダウンリストから選択するか、あるいは直接 DFP の名前を **<vendor>::<pack>@<version>** のフォーマットで入力することでアクティブなデバイスの DFP を選択することもできます。
  例:**ARM::V2M_MPS3_SSE_300_BSP@1.4.0**

・ **CMSIS-Pack Device Name**: ターゲットデバイスの名前(ボード上のターゲットチップ)を選択します。
- ー デフォルト値: **auto (CMSIS Solution)**。デバイス名は Arm Device Manager extension 内のプローブまたはボードとして利用可能な情報から推測されます。JSON ファイルに **deviceName** のための"**${command:device-manager.getDeviceName}**" コマンドが追加されます。
- ー **auto** (**Device Manager**): デバイス名は Arm Device Manager extension 内のプローブまたはボードとして利用可能な情報から推測されます。JSON ファイルに **deviceName** のための "**${command:device-manager.getDeviceName}**" コマンドが追加されます。
- ー ドロップダウンリストから選択するか、あるいは直接デバイス名を入力することもできます。
  例: **PS3_SSE_300**
  ドロップダウンリストで選択できるデバイス名は、solution の***.csolution.yml** ファイルで定義されているものです。

・ **Processor Name**: マルチコアデバイスを使用している場合、使用するプロセッサを選択します。
- ー デフォルト値: **auto**。**auto** の場合、デフォルトでは solution の***.csolution.yml** ファイル内に定義されたプロセッサ名が使用されます。JSON ファイルに"**processorName**" のための "**${command:cmsiscsolution.getProcessorName}**" コマンドが追加されます
- ー 直接プロセッサ名を入力することもできます。例:**cm4**

・ **Program Files**: ハードウェア上で実行する 1 つまたはそれ以上のプログラム
- ー デフォルト値: 新しい configration block を追加すると、 JSON ファイルに"**program**"のための "**${command:arm-debugger.getApplicationFile}**" コマンドが追加されます。このコマンドは、生成された最新の AXF または ELF ファイルを検出します。
- ー ファイルを直接指定するには、**Add File** をクリックします。必要な数のファイルを追加できます。Arm Debugger は、追加した順序でファイルを使用します。AXF ファイルと ELF ファイルはデフォルトでサポートされています。他のファイルタイプを追加することもできます。
- ー**${command:arm-debugger.getApplicationFile}** コマンドが利用できない場合は、**Detect File** をクリックして追加します。
- ーコマンドまたはファイル名の上にカーソルを移動し、delete アイコン  をクリックして選択を削除します。

・ **Connection Mode**: 接続モードを選択します。接続モードは、デバッガがターゲットデバッグプローブまたはボードに接続するときに行われる操作を制御します。
- ー デフォルト値: **auto**。デバッガは、接続されたターゲットデバイスに基づいて、使用する接続モードを決定します。ST のボードの場合、**auto** を選択すると、**underReset** が使用されます。その他のボードの場合は、**haltOnConnect** が使用されます。
- ー **haltOnConnect**: フラッシュダウンロードの前に、ターゲットデバッグプローブまたはボードの CPU を停止してリセットします。

  - **underReset**: 接続中にハードウェアリセットをアサートします。
  - **preReset**: 接続前にハードウェアリセットパルスをトリガします。
  - **running**: 接続中にプログラムの実行を停止せずに CPU に接続します。
- **Port Mode**: 使用するデバッグポートモードを選択します。デバッグポートを使用すると、マイクロコントローラやその他の組み込みシステムと通信してデバッグできます。
  - デフォルト値: **auto**。**auto** の場合、デバッガは接続されたターゲットデバイスに基づいて、使用するデバッグポートモードを決定します。
  - **JTAG**: JTAG デバッグポートモードを使用します。
  - **SWD**: SWD デバッグポートモードを決定します。
- **Clock Speed**: デバッグ通信の最大クロック周波数。クロック周波数はデバッグ操作中にデバッガとターゲットデバイス間でデータが転送される速度です。実際に使用される周波数はデバッグプローブの機能によって異なり、サポートされる周波数まで低下する場合があります。
  - デフォルト値: **auto**。**auto** の場合、デバッガは接続されたターゲットデバイスに基づいて、使用するクロック周波数を決定します。
  - その他選択可能な値: **50MHz, 33MHz, 25MHz, 20MHz, 10MHz, 5MHz, 2MHz, 1MHz, 500kHz, 200kHz, 100kHz, 50kHz, 20kHz, 10kHz, 5kHz**
- **Erase Mode**: フラッシュ消去に使用するタイプ
  - デフォルト値: **sectors**。**sectors** の場合プログラムするフラッシュメモリのセクタのみ消去されます。これら特定のセクタ内のすべてのデータが消去されます。
  - **none**: フラッシュ消去をスキップします。フラッシュメモリの内容はプログラミング前に消去されません。
- **Verify Flash**: フラッシュダウンロード中にフラッシュメモリにダウンロードされた内容を確認するには、このチェックボックスを選択します。
- **Flash Algorithms**: デフォルトのフラッシュアルゴリズムは、solution の Device Family Pack (DFP) に含まれます。独自のアルゴリズムを作成して、configuration で使用することもできます。詳細については、Open-CMSIS-Pack のドキュメントを参照してください。使用するフラッシュアルゴリズムを選択するか、チェックボックスをクリックして選択を解除します。DFP に存在し、**default** としてマークされているアルゴリズムがデフォルトで選択されます。

  以下のフィールドがあります:
  - **Path**: DFP 内のデフォルトのフラッシュアルゴリズムファイルへの相対パス、またはマシンのファイルシステム内のフラッシュアルゴリズムファイルへの絶対パス
  - **Region Start**: 選択したフラッシュアルゴリズムの対象となるメモリ領域の開始アドレス
  - **Region Size**: 選択したフラッシュアルゴリズムの対象となるメモリ領域のサイズ
  - **Ram Start**: フラッシュアルゴリズムの実行に使用されるターゲットシステムの RAM の開始アドレス
  - **Ram Size**: フラッシュアルゴリズムの実行に使用されるターゲットシステムの RAM のサイズ

  Region Start, **Region Size**, **Ram Start**, および **Ram Size** は 10 進数または 16 進数フォーマットで表すことができます。設定されていない場合は、DFP のデフォルト値が使用されます。

自身のフラッシュアルゴリズムを追加するには **task.json** のファイルを手作業で編集します。

a. JSON ファイルの"**flms**" キーの下に"**path**" を追加します。例:

```
"flms": [
{
 "path": "<Path to FLM file>"
}
]
```

b. visual editor、または直接 JSON ファイルを編集して、**Region Start**, **Region Size**, **Ram Start**, および **Ram Size** のフィールドを追加します。

```
"flms": [
{
 "path": "<Path to FLM file>",
 "regionStart": "<Start address of memory region>",
 "regionSize": "<Size of memory region>",
 "ramStart": "<Start address of target system's RAM>",
 "ramSize": "<Size of target system's RAM>"
}
]
```

- **Baud Rate**: ターゲットデバッグプローブまたはボードのシリアル出力を正しく表示するには、ボーレートを選択します。
  選択可能な値: **115200, 57600, 38400, 19200, 9600, 4800, 2400, 1800, 1200, 600**

4. もし **Auto Save** が有効になっていない場合、(File > Auto Save) で変更を保存します。
   tasks.json のファイルが更新されます。

7.1.5 Project の実行

ハードウェア上で project を実行します。

始める前に
マシン上の 1 つのフォルダ内に複数の solution がグループ化されている場合、Visual Studio Code は、各 solution 用に作成された可能性がある **tasks.json** のファイルと launch.json のファイルを考慮しません。代わりに Visual Studio Code は、workspace の root にある **.vscode** フォルダに新しい JSON のファイルを生成し、他の JSON ファイルを無視します。

その結果、project の実行またはデバッグで問題が発生する可能性があります。

回避策として、最初に 1 つの solution を開き、次に **File > Add Folder to Workspace** オプションを使用して、他の solution を workspace に追加します。

手順

1. ハードウェアがコンピュータに接続されているか確認します。
2. Command Palette を開きます。**`Tasks: Run Task`** を検索して選択します。
3. ドロップダウンリストから **`arm-debugger.flash: Flash Device`** を選択します。
   または Keil Studio Pack をインストール済みの場合、**CMSIS** ビューに移動し、Build Context ビュー を開き、選択されている run configuration を確認します。その後、**Solution outline** ヘッダ内の **Run** をクリックします。

   Run アイコンは、run configuration の **Build Context** ビューで選択した内容に応じて、solution レベルまたは project レベルで使用できます。詳細については、Solution の context を設定する を参照してください。
4. マルチコアのデバイスを使用しており、**`tasks.json`** ファイルで"**`processorName`**" を指定しておらず、CMSIS Solution extension がインストールされていない場合は、ウィンドウの上部に表示される **Select a processor** ドロップダウンリストで project にあったプロセッサを選択する必要があります。
5. **Terminal** タブをチェックして、project が正しく実行されたことを確認します。
   Arm Debugger エンジンがマシン上に見つからない場合は、**Arm Debugger not found** のダイアログボックスが表示されます。

   以下から 1 つを選択します:
   - **Install Arm Debugger** をクリックして、環境に追加します。**`vcpkg-configuration.json`** のファイルが更新されます。ステータスバー Arm Tools: 8 で Arm Tool がインストールされていることを確認します。
   - **Configure Path** をクリックして、settings で Arm Debugger エンジンへのパスを指定します。

7.2 Arm Debugger を使用して project をデバッグする

project をデバッグします。

7.2.1 configuration の追加

project を実行する場合と同様に、project をデバッグするには、まず launch configuration を追加する必要があります。launch configuration ファイルを生成すると、デバッグ設定の詳細をコンフィギュレーションして保存できます。Visual Studio Code は、デバッグコンフィギュレーション情報を **`launch.json`** ファイルに保存します。configuration が検出されない場合は、エラーが発生します。**`launch.json`** ファイルを開いて、Arm Debugger の launch configuration を追加するように求められます。

keil.arm.com で提供されているほとんどのサンプルには、debug configuration を含む **`launch.json`** のファイルが付属しています。必要に応じて、デフォルトの configuration を変更できます。

手順

1. Command Palette を開きます。**Debug: Add Configuration** を検索して選択します。

   launch.json が開きます。

   または、**Run** メニューに移動し、**Add Configuration…** を選択します。
2. **Arm Debugger: Attach**, **Arm Debugger: Launch**, または **Arm Debugger: Launch FVP** のタスクを選択します。

   これにより、project の **.vscode** フォルダ内の **launch.json** ファイルにデフォルトの debug configuration オプションが追加されます。
3. **launch.json** のファイルを保存します。

もしインストール済みの CMSIS Solution extension がない場合、以下を利用できます:

- **"cmsisPack": "${command:cmsis-csolution.getTargetPack}"** の代わりに **"cmsisPack": "${command:device-manager.getDevicePack}"**
- **"deviceName": "${command:cmsis-csolution.getDeviceName}"** の代わりに **"deviceName": "${command:device-manager.getDeviceName}"**

7.2.2 Arm Debugger のデフォルトの debug configuration オプションを上書きまたは拡張する

必要に応じて、デフォルトの configuration option を上書きまたは拡張できます。詳細については、Arm Debugger configuration オプション を参照してください。
また、tasks.json ファイル の **cmsisPack** および **deviceName** の詳細 も参照してください。ハードウェアデバイスをデバッグするには、launch configuration でどの CMSIS-Pack から情報を読み取るかと、使用する CMSIS-Pack 内のデバイス名を認識している必要があります。

7.2.3 Arm Debugger configuration オプション

extension では以下の debug configuration オプションを提供します。

物理ターゲットでの debug configuration オプション

configuration オプション	説明
"**cdbentry**"	選択する Arm Debugger Configuration Database Entry
"**cdbEntryParams**"	選択した **cdbEntry** に固有の 1 つまたはそれ以上のキー/値設定 例: **model_params: -f ${workspaceFolder}/model_config.txt**
"**cmsisDevice**"	CMSIS-Pack 名、デバイスベンダ、デバイス名、およびプロセッサ名(マルチコアの場合)の連なり 非推奨。代わりに **cmsisPack**, **pdsc**, **vendorName**, **deviceName**, および **processorName** を使用してください。
"**cmsisPack**"	ハードウェアの DFP (Device Family Pack) CMSIS-Pack へのパス(ファイルまたは URL) 以下とともに使用できます: • **cmsis-csolution.getTargetPack**: **CMSIS Solution Build Context** ビューで選択したターゲットタイプの DFP CMSIS-Pack を取得します。**cmsis-csolution.getTargetPack** は solution に固有です。 • **device-manager.getDevicePack**: 選択したデバイスの DFP CMSIS-Pack を取得します。このコマンドは、pack index で使用可能な最新の pack を使用します。
"**conncetMode**"	接続モード 選択可能な値: • **auto**: デバッガが決定します。 • **haltOnConnect**: 実行前にリセットのため停止します。 • **underReset**: 外部 NRST ラインをアサートしたままにします。 • **preReset**: NRST を使用してプリリセットを行います。 • **running**: 状態を変更せずに実行中のターゲットに接続します。 デフォルト: **auto**
"**dbgconf**"	CMSIS-Pack デバッグシーケンスの実行をコンフィギュレーションする **.dbgconf** ファイルへのパス。Arm Debugger v6.1.0 以降が必要です。

configuration オプション	説明
"**debugClockSpeed**"	デバッグ接続の最大周波数。実際に使用される周波数はデバッグプローブによって異なります。auto はターゲット固有のデフォルトを使用します。Arm Debugger v6.0.2 以降が必要です。 選択可能な値:**auto**, **50MHz**, **33MHz**, **25MHz**, **20MHz**, **10MHz**, **5MHz**, **2MHz**, **1MHz**, **500kHz**, **200kHz**, **100kHz**, **50kHz**, **20kHz**, **10kHz**, **5kHz** デフォルト: **auto**
"**debugFrom**"	デバッグ前にデバッガがそこまで実行するシンボル デフォルト: "**main**"
"**debugInitScript**"	接続後に実行され、**debugFrom** まで実行されるデバッグ初期化スクリプト(.ds/.py) へのパス。 Arm Debugger v6.1.1 以降が必要です。
"**debugPortMode**"	デバッグ接続に使用するデバッグポートモード。Arm Debugger v6.0.2 以降が必要です。 選択可能な値:**auto**, **JTAG**, **SWD** デフォルト: **auto**
"**deviceName**"	CMSIS-Pack デバイス名 以下とともに使用できます: • **cmsis-csolution.getDeviceName**: ソリューションの ***.csolution.yml** ファイル内のプローブまたはボードで利用可能な情報からデバイス名を取得します。 • **device-manager.getDeviceName**: 選択したデバイスの DFP からデバイス名を取得します。
"**PathMapping**"	ソース ファイルを解決するためのリモートパスとローカルパスのマッピング
"**pdsc**"	PDSC ファイルへのパス(ファイルまたは URL)
"**probe**"	デバッグ接続に使用するプローブの名前 選択可能な値: **ULINKpro**, **ULINKpro D**, **ULINK2**, **CMSIS-DAP**, **ULINKplus**, **ST-Link** デフォルト: **CMSIS-DAP**
"**ProcessorName**"	マルチコアデバイスの CMSIS-Pack プロセッサ名
"**program**" または "**programs**"	ロード順で使用する 1 つ以上の project へのパス(ファイルまたは URL)。Arm Debugger v6.0.2 以降が必要です。 以下とともに使用できます: • **arm-debugger.getApplicationFile**: CMSIS の実行とデバッグに使用される AXF または ELF ファイルを返します。

configuration オプション	説明
"**programMode**"	アプリケーションをターゲットにプログラムするモード 選択可能な値: **auto, flash, ram, mixed** デフォルト: **auto**
"**resetAfterConnect**"	プロセッサの制御を得た後、デバイスをリセットします。
"**resetMode**"	使用するリセットのタイプ 選択可能な値: • **auto**: デバッガが決定 • **system**: : ResetSystem シーケンスを使用 • **hardware**: ResetHardware シーケンスを使用 • **process**: ResetProcessor シーケンスを使用 デフォルト: **auto**
"**searchPaths**"	ソースの場所に対するパスの配列
"**serialNumber**"	使用する接続された USB ハードウェアのシリアル番号 以下とともに使用できます: • **device-manager.getSerialNumber**: 選択したデバイスのシリアル番号を取得します。
"**svd**"	SVD ファイルへのパス(ファイルまたは URL)
"**targetAddress**"	**serialNumber** と同義です。
"**targetInitScript**"	接続後、他の操作の前に実行されるターゲット初期化スクリプト(**.ds/.py**) へのパス。Arm Debugger v6.1.1 以降が必要です。
"**vendorName**"	CMSIS-Pack ベンダ名
"**worspaceFolder**"	現在の Arm Debugger のワークスペースフォルダ。 デフォルト:"**${workspaceFolder}**"

仮想ターゲット(Fixed Virtual Platforms) での debug configuration オプション

FVP は、Windows と Linux でのみネイティブに使用できます。Mac を使用している場合は、Docker を使用して Linux コンテナで FVP を実行できます。Arm Developer Learning Path に従って Docker をインストールし、https://github.com/Arm-Examples/FVPs-on-Mac リポジトリをクローンしてください。

configuration オプション	説明
"**cdbentry**"	• 選択する Arm Debugger Configuration Database Entry
"**cdbEntryParams**"	選択した **cdbEntry** に固有の 1 つまたはそれ以上のキー/値設定 例: **model_params: -f ${workspaceFolder}/model_config.txt**
"**debugFrom**"	デバッグ前にデバッガがそこまで実行するシンボル デフォルト: "**main**"
"**debugInitScript**"	接続後に実行され、**debugFrom** まで実行されるデバッグ初期化スクリプト(.ds/.py) へのパス。 Arm Debugger v6.1.1 以降が必要です。
"**fvpParameters**"	FVP parameter configuration ファイルへのパス
"**PathMapping**"	ソース ファイルを解決するためのリモートパスとローカルパスのマッピング
"**program**" または "**programs**"	ロード順で使用する 1 つ以上の project へのパス(ファイルまたは URL)。Arm Debugger v6.0.2 以降が必要です。 以下とともに使用できます: • **arm-debugger.getApplicationFile**: CMSIS の実行とデバッグに使用される AXF または ELF ファイルを返します。
"**programMode**"	アプリケーションをターゲットにプログラムするモード デフォルト: **ram**
"**searchPaths**"	ソースの場所に対するパスの配列
"**svd**"	SVD ファイルへのパス(ファイルまたは URL)
"**targetInitScript**"	接続後、他の操作の前に実行されるターゲット初期化スクリプト(**.ds/.py**) へのパス。Arm Debugger v6.1.1 以降が必要です。
"**worspaceFolder**"	現在の Arm Debugger のワークスペースフォルダ。 デフォルト:"**${workspaceFolder}**"

7.2.4 Run and Debug Configuration visual editor を使った debug configuration

solution の **`launch.json`** ファイルを編集して debug configuration オプションを変更する代わりに、**Run and Debug Configuration** visual editor を使用することもできます。

7.2.4.1 物理ターゲットの debug configuration

このセクションでは、Run and Debug Configuration visual editor を使用して configuration を **`Attach`** または **`Launch`** する方法について説明します。

手順

1. editor を開くには以下のいずれかを行います:
   - **Explorer** で、solution の **`.vscode`** フォルダに保存されている **`launch.json`** ファイルを右クリックし、**Open Run and Debug Configuration** を選択します。
   - **Explorer** で、**`launch.json`** ファイルを右クリックし、**Open With…** を選択し、ウィンドウの上部に表示されるドロップダウンリストで **Run and Debug Configuration** を選択します。
   - **`launch.json`** ファイルがすでに editor で開かれている場合は、右上隅にある **Open Run and Debug Configuration** をクリックします。
2. **`launch.json`** ファイルでは、物理ターゲットに対する複数の debug configuration を定義できます。**Selected Configuration** ドロップダウンリストで、**New Configuration** を選択し、次のオプションのいずれかを選択します:
   - **Attach**: すでに実行中のプログラムをデバッグする場合は、**`Attach`** configuration を使用します。
   - **Launch**: 物理ターゲットを使用してプログラムをデバッグモードで起動するには、**`Launch`** configuration を選択します。

     Visual Studio Code の **`Launch`** および `Attach` コアデバッグモードの説明については、Launch versus attach configurations を参照してください。

   configuration を選択すると、JSON ファイルに新しい configuration block が追加されます。

   Duplicate をクリックして、現在選択されている configuration を複製し、変更することもできます。
3. Attach configuration を定義している場合は、debug configuration を次のように変更します。
   - Arm Debugger engine が離れたサーバで実行されている場合は、サーバのアドレスを **`ws://<host>:<port>`** (websocket) のフォーマットで指定します。
   - Arm Debugger engine がマシン上で実行されている場合は、**`<host>:<port>`** (socket) を使用します。
4. **`Launch`** configuration を定義する場合は、次のように debug configuration を変更します:
   - **Probe Type**: ドロップダウンリストで、使用しているデバッグプローブまたはボード上のデバッグユニットのタイプを選択します。
     - デフォルト値: **`CMSIS-DAP`**。Arm Debugger extension が probe type を自動的に設定できない場合、デフォルト値は **`CMSIS-DAP`** です。

  - プローブまたはボードを USB 経由でコンピュータに接続した場合、Arm Debugger extension は、検出されたハードウェアのシリアル番号に基づいて probe type を設定します。
- **Serial Number**: ドロップダウンリストで使用しているデバッグプローブか、ボード上のデバッグユニットのシリアル番号を選択します。
  - デフォルト値: **auto**。**auto** の場合、Arm Device Manager extension 内でアクティブなデバイスのシリアル番号がデフォルトで使用されます。JSON ファイルに **serialNumber** のための"**${command:device-manager.getSerialNumber}**" コマンドが追加されます。
  - ドロップダウンリストから選択するか、あるいは直接シリアル番号を入力することでアクティブなデバイスのシリアル番号を選択することもできます。

  アクティブなデバイスを確認するには **Open Arm Device Manager** をクリックします。
- **CMSIS-Pack**: ターゲットデバッグプローブまたはボードの Device Family Pack(DFP) を選択します。
  - デフォルト値: **auto (CMSIS Solution)**。solution の***.csolution.yml** ファイルで定義されているアクティブなデバイスの DFP がデフォルトで使用されます。**CMSIS-Pack** のための JSON ファイルに"**${command:cmsis-csolution.getTargetPack}**"コマンドが追加されます。
  - **auto** (**Device Manager**): Arm Device Manager extension 内のアクティブなデバイスの DFP がデフォルトで使用されます。**CMSIS-Pack** のための JSON ファイルに ″**${command:device-manager.getDevicePack}**″コマンドが追加されます。
  - ドロップダウンリストから選択するか、あるいは直接 DFP の名前を **<vendor>::<pack>@<version>** のフォーマットで入力することでアクティブなデバイスの DFP を選択することもできます。
    例:**ARM::V2M_MPS3_SSE_300_BSP@1.4.0**
- **CMSIS-Pack Device Name**: ターゲットデバイスの名前(ボード上のターゲットチップ)を選択します。
  - デフォルト値: **auto (CMSIS Solution)**。デバイス名は Arm Device Manager extension 内のプローブまたはボードとして利用可能な情報から推測されます。JSON ファイルに **deviceName** のための"**${command:device-manager.getDeviceName}**" コマンドが追加されます。
  - **auto** (**Device Manager**): デバイス名は Arm Device Manager extension 内のプローブまたはボードとして利用可能な情報から推測されます。JSON ファイルに **deviceName** のための"**${command:device-manager.getDeviceName}**" コマンドが追加されます。
  - ドロップダウンリストから選択するか、あるいは直接デバイス名を入力することもできます。
    例: **PS3_SSE_300**
    ドロップダウンリストで選択できるデバイス名は、solution の***.csolution.yml** ファイルで定義されているものです。
- **Processor Name**: マルチコアデバイスを使用している場合、使用するプロセッサを選択します。ます。
  - デフォルト値: **auto**。**auto** の場合、デフォルトでは solution の***.csolution.yml** ファイル内に定義されたプロセッサ名が使用されます。JSON ファイルに"**processorName**" のための "**${command:cmsiscsolution.getProcessorName}**" コマンドが追加

されます。
  - 直接プロセッサ名を入力することもできます。例：**cm4**
- **Connection Mode**：接続モードを選択します。接続モードは、デバッガがターゲットデバッグプローブまたはボードに接続するときに行われる操作を制御します。
  - デフォルト値： **auto**。デバッガは、接続されたターゲットデバイスに基づいて、使用する接続モードを決定します。ST のボードの場合、**auto** を選択すると、**underReset** が使用されます。その他のボードの場合は、**haltOnConnect** が使用されます。
  - **haltOnConnect**： フラッシュダウンロードの前に、ターゲットデバッグプローブまたはボードの CPU を停止してリセットします。
  - **underReset**： 接続中にハードウェアリセットをアサートします。
  - **preReset**： 接続前にハードウェアリセットパルスをトリガします。
  - **running**： 接続中にプログラムの実行を停止せずに CPU に接続します。
- **Reset after Connect**：プロセッサの制御を取得した後にデバイスをリセットするには、このオプションを選択します。
- **Reset Mode**：リセットモードを選択します。リセットモードは、デバッガによって実行されるリセット操作を制御します。
  - **auto** (デフォルト)：デバッガは、CMSIS-Pack からの情報に基づいて、使用するリセットを決定します。
  - **system**：CMSIS-Pack の **ResetSystem** シーケンスを使用します。
  - **hardware**：CMSIS-Pack の **ResetHardware** シーケンスを使用します。
  - **processor**：CMSIS-Pack の **ResetProcessor** シーケンスを使用します。
- **Debug From**：デバッガが動作開始する関数を選択します。デフォルト値：**main**。デバッグセッションが開始されると、デバッガはプログラムの **main()** 関数で停止します。
- **Program Mode**：プログラムモードを選択します。プログラムモードはデバッグの種類を定義します：フラッシュデバッグ **flash**、RAM デバッグ **RAM**、またはその両方の **mixed**。
  デフォルト値：**auto**。**auto** モードではデバッガが決定します。
  フラッシュデバッグと RAM デバッグの主な違いは、デバッグセッション中にコードを保存および実行するために使用されるメモリの種類にあります。
  - フラッシュデバッグ：コードはフラッシュメモリに保存され、そこから実行されます。デバッガは内部的にデバッグ情報を読み込みますが、ターゲットには何もロードしません。
  - RAM デバッグ：デバッガはターゲットシステムに接続した後、コードを RAM に読み込みます。コードは実行前にまず保存場所(フラッシュメモリなど)から RAM にコピーされます。
- **Port Mode**：使用するデバッグポートモードを選択します。デバッグポートを使用すると、マイクロコントローラやその他の組み込みシステムと通信してデバッグできます。
  - デフォルト値： **auto**。**auto** の場合、デバッガは接続されたターゲットデバイスに基づいて、使用するデバッグポートモードを決定します。
  - **JTAG**： JTAG デバッグポートモードを使用します。
  - **SWD**： SWD デバッグポートモードを決定します。
- **Clock Speed**：デバッグ通信の最大クロック周波数。クロック周波数はデバッグ操作中にデバッガとターゲットデバイス間でデータが転送される速度です。実際に使用される周波数はデバッグプローブの機能によって異なり、サポートされる周波数まで低下する場合があります。
  - デフォルト値： **auto**。**auto** の場合、デバッガは接続されたターゲットデバイスに基づいて、使用するクロック周波数を決定します。

  - その他選択可能な値: `50MHz`, `33MHz`, `25MHz`, `20MHz`, `10MHz`, `5MHz`, `2MHz`, `1MHz`, `500kHz`, `200kHz`, `100kHz`, `50kHz`, `20kHz`, `10kHz`, `5kHz`
- **Program Files**: ハードウェア上で実行する 1 つまたはそれ以上のプログラム
  - デフォルト値: 新しい configration block を追加すると、JSON ファイルに"`program`"のための"`${command:arm-debugger.getApplicationFile}`" コマンドが追加されます。このコマンドは、生成された最新の AXF または ELF ファイルを検出します。`${command:arm-debugger.getApplicationFile}` コマンドが利用できない場合は、**Detect File** をクリックして追加します。
  - ファイルを直接指定するには、**Add File** をクリックします。必要な数のファイルを追加できます。Arm Debugger は、追加した順序でファイルを使用します。AXF ファイルと ELF ファイルはデフォルトでサポートされています。他のファイルタイプを追加することもできます。

5. もし **Auto Save** が有効になっていない場合、(**File** > **Auto Save**) で変更を保存します。`tasks.json` のファイルが更新されます。

7.2.4.2 Virtual target (Fixed Virtual Platforms) の debug configuration

このセクションでは、Run and Debug Configuration visual editor を使用して configuration を Attach または Launch する方法について説明します。

始める前に

Fixed Virtual Platforms(FVP) モデルを使用して仮想ターゲットをデバッグするには、モデルをマシンにインストールする必要があります。

project の `vcpkg-configuration.json` ファイルの"`requires`": セクションに"`arm:models/arm/avhfvp`" が含まれていることを確認します。

Arm Tools visual editor(**Arm Virtual Hardware for Cortex®-M based on Fast Models** オプション) を使用するか、`vcpkg-configuration.json` ファイルを直接編集して、FVP を追加できます。

FVP は、Windows と Linux でのみネイティブに使用できます。Mac を使用している場合は、Docker を使用して Linux コンテナで FVP を実行できます。Arm Developer Learning Path に従って Docker をインストールし、https://github.com/Arm-Examples/FVPs-on-Mac リポジトリをクローンしてください。

手順

1. editor を開くには以下のいずれかを行います:
   - **Explorer** で、solution の `.vscode` フォルダに保存されている `launch.json` ファイルを右クリックし、**Open Run and Debug Configuration** を選択します。
   - **Explorer** で、`launch.json` ファイルを右クリックし、**Open With…** を選択し、ウィンドウの上部に表示されるドロップダウンリストで **Run and Debug Configuration** を選択します。
   - `launch.json` ファイルがすでに editor で開かれている場合は、右上隅にある **Open Run and Debug Configuration** をクリックします。

2. **Selected Configuration** ドロップダウンリストで **New Configuration** を選び、**Launch FVP** を選択します。

   Duplicate をクリックして、現在選択されている configuration を複製し、変更することもできます。
3. 以下の手順で debug configuration を修正します:
   - **Configuration Database Entry**: configuration database は、Arm Debugger が接続できるプロセッサ、デバイス、ボードに関する情報を保存する場所です。使用する FVP (例:**MPS2_Cortex_M4**) を選択してから、プロセッサ(例:**Cortex-M4**) を選択します。使用可能な FVP のリストは、project の **`vcpkg-configuration.json`** ファイルで指定されている **`avh-fvp`** のバージョンによって異なります。
   - **FVP Paramaters**: より詳細な configuration を行うには、FVP parameters のリストを生成し、ファイルにリストされている引数を変更します。**Generate File** をクリックして、**`fvp_config.txt`** ファイルを生成します。ペンアイコン  をクリックして FVP Parameters editor を開き、ファイルを変更します。既に使用可能なファイルがある場合は、**Select File** をクリックして選択します。
   - **Debug From**: デバッガが動作開始する関数を選択します。デフォルト値: **`main`**。デバッグセッションが開始されると、デバッガはプログラムの **`main()`** 関数で停止します。
   - **Program Files**: ハードウェア上で実行する 1 つまたはそれ以上のプログラム
     - デフォルト値: 新しい configration block を追加すると、 JSON ファイルに"**`program`**"のための"**`${command:arm-debugger.getApplicationFile}`**" コマンドが追加されます。このコマンドは、生成された最新の AXF または ELF ファイルを検出します。**`${command:arm-debugger.getApplicationFile}`** コマンドが利用できない場合は、**Detect File** をクリックして追加します。
     - ファイルを直接指定するには、**Add File** をクリックします。必要な数のファイルを追加できます。Arm Debugger は、追加した順序でファイルを使用します。AXF ファイルと ELF ファイルはデフォルトでサポートされています。他のファイルタイプを追加することもできます。
     - コマンドまたはファイル名の上にカーソルを移動し、delete アイコン  をクリックして選択を削除します。
4. もし **Auto Save** が有効になっていない場合、(**File** > **Auto Save**) で変更を保存します。**`tasks.json`** のファイルが更新されます。

7.2.5 Arm Debugger session の開始

Debug session を開始します。

始める前に

マシン上の 1 つのフォルダ内に複数の solution がグループ化されている場合、Visual Studio Code は、各 solution 用に作成された可能性がある **`tasks.json`** のファイルと launch.json のファイルを考慮しません。代わりに Visual Studio Code は、workspace の root にある **`.vscode`** フォルダに新しい JSON のファイルを生成し、他の JSON ファイルを無視します。

その結果、project の実行またはデバッグで問題が発生する可能性があります。

回避策として、最初に 1 つの solution を開き、次に **File > Add Folder to Workspace** オプションを使用して、他の solution を workspace に追加します。

7.2.5.1 物理ターゲットの debug session を開始する

物理ターゲットの debug session を開始するには以下の手順を行います:

手順

1. デバイスがコンピュータに接続されていることを確認します。
2. Debug session を開始するには **Run and Debug** ビュー に移動し、リスト 

内の **`Arm Debugger`** configuration を選択します。**Start Debugging** をクリックします。
あるいは、Keil Studio Pack をインストールしている場合は、**CMSIS** ビューに移動し、**Build Context** ビュー を開いて、どの debug configuration が選択されているかを確認します。次に、**Solution outline** ヘッダで **Debug** をクリックします。

   Debug アイコンは、**Build Context** ビューで debug configuration で選択した内容に応じて、solution レベルまたは project レベルで使用できます。詳細については、Solution の context を設定する を参照してください。
3. マルチコアのデバイスを使用しており、`launch.json` ファイルで"`processorName`" を指定しておらず、CMSIS Solution extension がインストールされていない場合は、ウィンドウの上部に表示される **Select a processor** ドロップダウンリストで project に対して適切なプロセッサを選択する必要があります。
   Run and Debug ビューが表示され、デバッグセッションが開始されます。デバッガはプログラムの `main()` 関数で停止します。
4. デバッグ出力を **Debug Console** タブで確認します。
   Arm Debugger エンジンがマシン上に見つからない場合は、**Arm Debugger not found** のダイアログボックスが表示されます。

   以下から 1 つを選択します:
   - **Install Arm Debugger** をクリックして、環境に追加します。**`vcpkg-configuration.json`** のファイルが更新されます。ステータスバー 

で Arm Tool がインストールされていることを確認します。
   - **Configure Path** をクリックして、settings で Arm Debugger エンジンへのパスを指定します。

7.2.5.2 Virtual target の debug session を開始する

Virtual target で debug session を開始します。

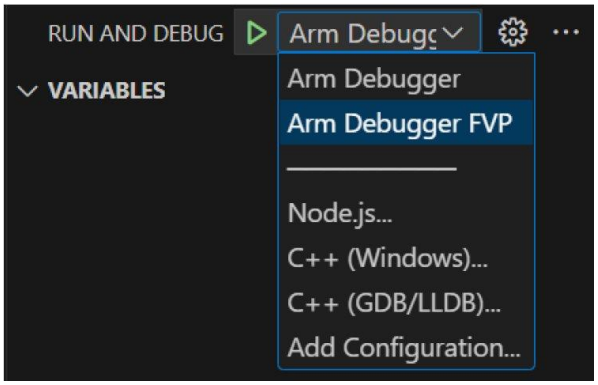
始める前に

FVP は、Windows と Linux でのみネイティブに使用できます。Mac を使用している場合は、Docker を使用して Linux コンテナで FVP を実行できます。Arm Developer Learning Path に従って Docker をインストールし、https://github.com/Arm-Examples/FVPs-on-Mac リポジトリをクローンしてください。

手順

1. **Device Manager** に移動し、使用したい FVP を選択します。例：**MPS2 Cortex M4**
2. Debug session を開始するには、**Run and Debug** ビュー に移動し、リスト内の **`Arm Debugger FVP`** configuration を選択します。**Start Debugging** をクリックします。

   図 7-1：FVP Configuration



   あるいは、Keil Studio Pack をインストールしている場合は、**CMSIS** ビューに移動し、**Build Context** ビュー を開いて、どの debug configuration が選択されているかを確認します。次に、**Solution outline** ヘッダで **Debug** をクリックします。

   Debug アイコンは、**Build Context** ビューで debug configuration で選択した内容に応じて、solution レベルまたは project レベルで使用できます。詳細については、Solution の context を設定する を参照してください。

   Run and Debug ビューが表示され debug session が開始されます。デバッガはプログラムの `main()` 関数で停止します。
3. デバッグ出力を **Debug Console** タブで確認します。
   Arm Debugger エンジンがマシン上に見つからない場合は、**Arm Debugger not found** のダイアログボックスが表示されます。

   以下から 1 つを選択します：
   - **Install Arm Debugger** をクリックして、環境に追加します。**`vcpkg-configuration.json`** のファイルが更新されます。ステータスバー Arm Tools: 8 Arm Tool がインストールされていることを確認します。
   - **Configure Path** をクリックして、settings で Arm Debugger エンジンへのパスを指定します。

7.2.6 ブレークポイントを設定する

ブレークポイントは、コードのどの部分を調べたいかがわかっている場合に便利です。変数の値を確認したり、コードブロックが実行されているかどうかを確認したりするには、1 つ以上のブレークポイントを設定して実行中のコードを一時停止することができます。

より詳細については Visual Studio Code のドキュメントを参照してください。

Note

Arm Debugger extension の現在のバージョンでは、デフォルトではアセンブリファイルにブレークポイントを設定できません。アセンブリファイルにブレークポイントを設定できるようにするには、settings に移動して **Allow Breakpoints Everywhere** を選択します。

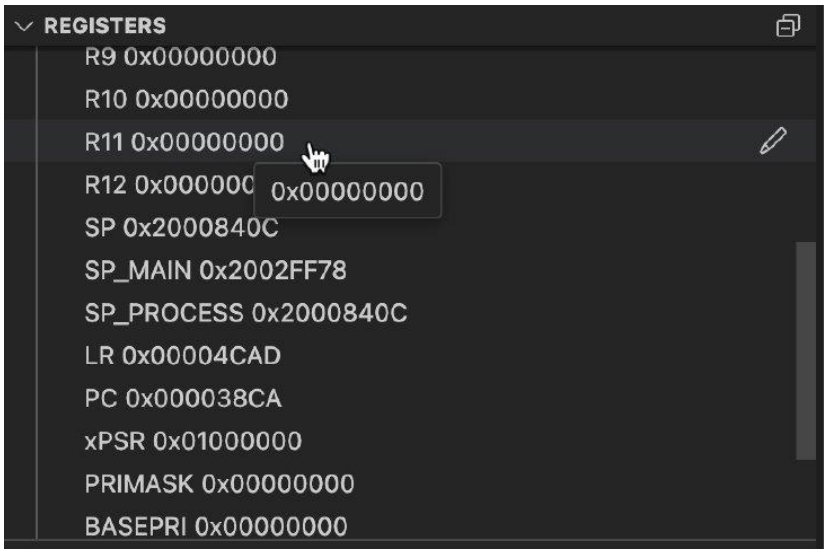
7.2.7 レジスタを参照する

Registers ビューには、検出されたプロセッサのレジスタの内容が表示されます。Arm Debugger session の開始 の説明に従って debug session を開始すると、**Run and Debug** ビューに **Registers** ビューが表示されます。

Registers ビューでは、レジスタがグループにまとめられています。これらのグループは、使用しているプロセッサタイプとデバッグしているシステムによって異なります。デバッグ中は、コードの実行に応じてレジスタの値が変わります。

以下は、Cortex-M4 プロセッサに対して **Registers** ビューに表示される内容の例です。

図 7-2：Cortex-M4 プロセッサの Registers ビュー



Registers ビューは以下を含みます：

- Processor core registers：Arm プロセッサでは、各プロセッサコアに、プログラム実行中に一時的なデータの保存と操作に使用される汎用レジスタのセットがあります。これらのレジスタは、算術、論理、データ移動命令などのさまざまな操作のためプロセッサによって使用されます。さらに Arm プロセッサには、プログラムフローの管理とスタックの維持に不可欠な Program Counter(PC) や Stack Pointer(SP) などの他の特定のレジスタがあります。これらのレジスタは、プロセッサコアのレジスタファイルをまとめて形成し、処理中にプロセッサがデータを保存および取得するための高速で効率的な手段を提供します。
- System registers：Arm プロセッサでは、システムレジスタは、プロセッサの動作のさまざまな側面を制御およびコンフィギュレーションする特殊用途のレジスタです。これらのレジスタは Arm アーキテクチャの一部であり、システムレベルの機能の管理に重要な役割を果たします。システムレジスタは、プロセッサの動作モード、割り込み処理、およびその他のシステム関連機能の制御に役立ちます。

- Floating-Point Unit (FPU) registers: Arm プロセッサでは、FPU が浮動小数点演算の処理を担当します。FPU には、汎用レジスタとは異なる独自のレジスタセットがあります。これらのレジスタは、浮動小数点数を格納し、それらに対して加算、減算、乗算、除算などの演算を実行するために使用されます。

7.2.7.1 レジスタを編集する

Debug session 中にレジスタを編集します。

手順

1. Arm Debugger session の開始 の説明の通り debug session を開始します。
2. **Pause** をクリックして debug session を中断します。
   Registers ビューには、編集可能なレジスタ値が表示されます。
3. レジスタ値の上にカーソルを移動し、更新する値の pen アイコン をクリックします。
4. ウィンドウの上部に開いたフィールドに値または式を入力し、**Enter** キーを押します。
   式を入力すると、式の結果がレジスタに書き込まれます。
   例: `$SP+0x20` を使用して、`SP` レジスタの内容に `0x20` を追加
   式の詳細については、Arm Debugger Command Reference guide を参照してください。

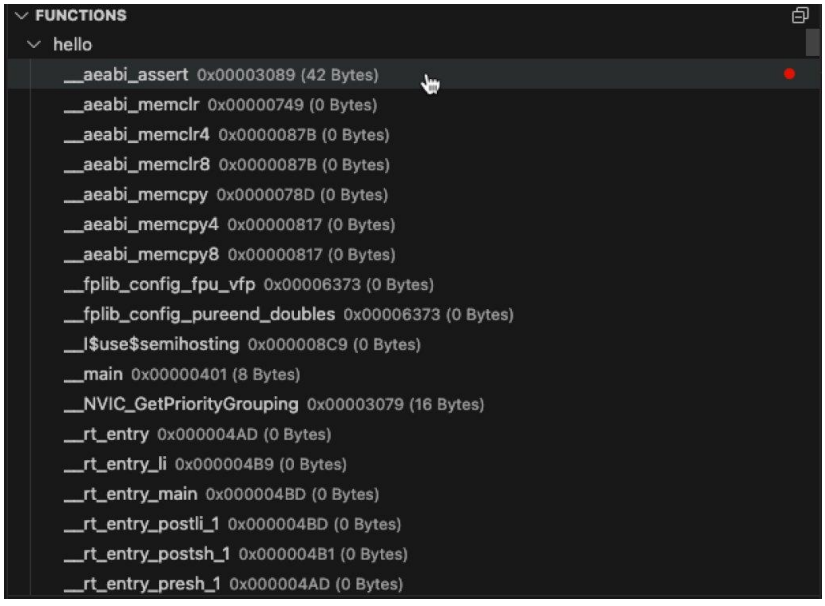
   修正された値は **Registers** ビューでハイライトされます。

7.2.8 関数を参照する

Functions ビューには、コード内の主な関数、ライブラリ関数、およびユーザ定義関数が表示されます。Arm Debugger session の開始 の説明に従って debug session を開始すると、Run and Debug ビューに Functions ビューが表示されます。
Functions ビューには、関数ごとに以下の詳細が表示されます:

- 関数の名前
- 関数が格納されているアドレス
- 関数のサイズ(バイト単位)

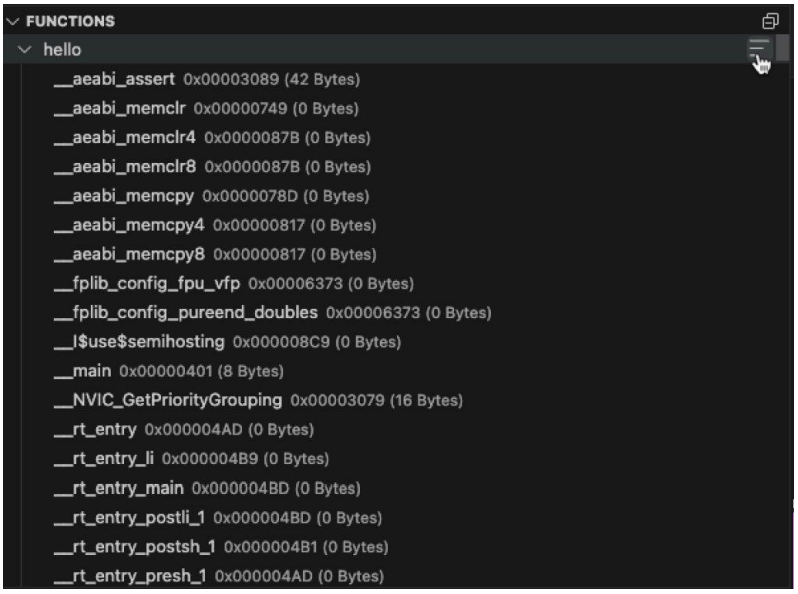
図 7-3: Functions ビュー



関数は名前、アドレス、サイズで並べ替えることができます。

1. **Functions** ビューで solution の名前の上にカーソルを移動し、**Sort** をクリックします。

図 7-4：Functions ビューSort ボタン



2. ウィンドウの上部に表示されるドロップダウンリストからオプションを選択します。
   - **Sort By Label**
   - **Sort By Address (Desc)**
   - **Sort By Function Size (Desc)**

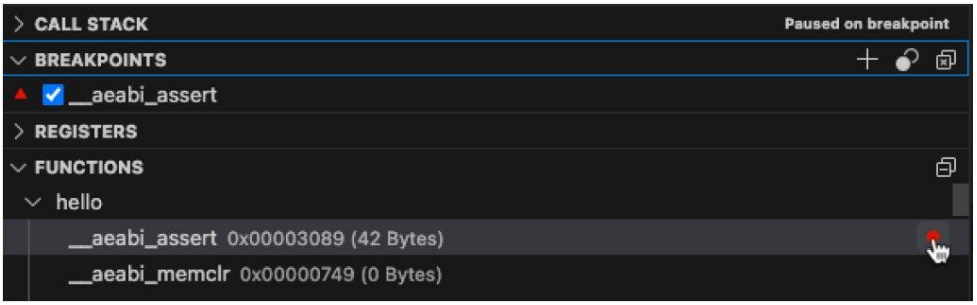
関数が呼び出されたときに実行を停止させるために、関数ブレークポイントを追加できます。ブレークポイントを利用した実行制御は、関数の名前はわかっているものの実際の場所がわからない場合に便利です。

関数ブレークポイントを設定するには以下の手順を行います：

Functions ビューでブレークポイントを追加する関数の上にカーソルを移動し、次のいずれかを実行します：

- 関数名の右側に表示される赤い点をクリックします
- 関数を右クリックして、**Set function breakpoint** を選択します。

図 7-5：関数ブレークポイントの追加



関数ブレークポイントは **Breakpoints** ビューに表示されます。

Breakpoints ビューから、以下を行えます:

-  ですべてのブレークポイントを削除するか、特定のブレークポイントを削除します。

図 7-6: 特定のブレークポイントの削除

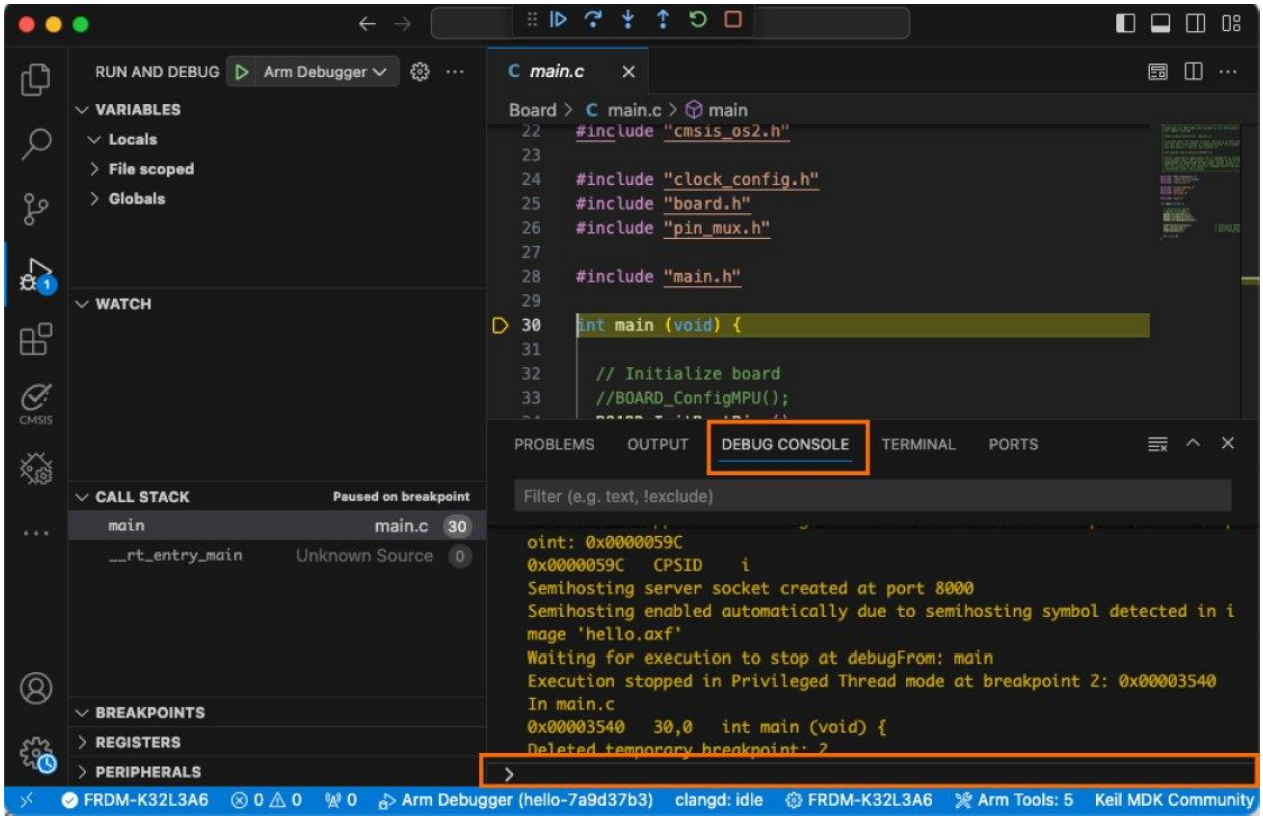
-  で関数ブレークポイントを追加します。
-  ですべてのブレークポイントを有効化または無効化します。

7.2.9 Debug Console を使用する

Debug Console には、project のデバッグ出力が表示されます。debug session 中に生成されたメッセージ、エラー、警告、その他の出力が表示されます。

debug session を開始すると、**Debug Console** が自動的に表示されます。**View** > **Debug Console** を選択して表示することもできます。

図 7-7: Debug Console



7.2.9.1 Arm Debugger コマンドの実行

Arm Debugger コマンドは、**Debug Console** から直接実行できます。**Debug Console** プロンプトで **help** に続けてコマンド名を入力すると、コマンドに関する情報とその使用方法が表示されます。

たとえば、**help step** では次のような内容が表示されます。

```
step
step
    Steps through an application at the source level stopping on the first
    instruction of each source line including stepping into all function calls.You
    must compile your code with debug information to use this command successfully.

    You can modify the behavior of this command with the set step-mode command.

Syntax
    step [<count>]
    Where:
    <count>
        Specifies the number of source lines to execute.
        Note:
            Execution stops immediately if a breakpoint is reached, even if fewer
            than <count> source lines are executed.
Examples
    step                                # Execute one source line.
    step 5                              # Execute five source lines.
```

例:

1. **Debug Console** プロンプトに **break main.c:10** と入力して **Enter** キーを押すと、**main.c** ファイルの 10 行目にブレークポイントが追加されます。
2. プロンプトに **continue** と入力して **Enter** キーを押すと、debug session が続行されます。デバッガは最初に遭遇したブレークポイントまで実行され、停止します。
3. 次の行を実行するには **step** と入力します。

すべての Arm Debugger コマンドは、Arm Debugger Command Reference 内の Arm Debugger commands listed in alphabetical order に記載されています。

プロンプトで **help** に続けてグループ名を入力すると、そのグループに属するすべてのコマンドが表示されます。

たとえば、**help group_log** は次のような表示が行われます:

```
group_log
log
    List of all the Arm Debugger commands that enable you to control runtime
    messages from the debugger.
    log config
        Specifies the type of logging configuration to output runtime messages from
        the debugger.
    log file
         Specifies an output file to receive runtime messages from the debugger.
         Enter help followed by a command name for more information on a specific
         command.
```

グループは、Arm Debugger Command Reference 内の Arm Debugger commands listed in groups に記載されています。

7.2.9.2 Expression を使用する

Debug Console を使用して式を評価および解決できます。

Debug Console プロンプトで **`$expr:<expression>`** を使用します。**`<expression>`** では、プログラムシンボル、レジスタシンボルおよび算術演算または論理演算を記述できます。

スタックポインタレジスタシンボル**`$SP`** と算術演算の例:

1. **Debug Console** プロンプトに**`$expr:$SP`** と入力し、**Enter** キーを押します。
   この式は、スタックポインタの現在の値(例: **`537133040`**) を返します。
2. **`$expr:$SP+0x20`** と入力し、**Enter** キーを押します。
   この式は、スタックポインタの現在の値+32 (この例では **`537133072`**) を返します。

グローバル変数 SystemCoreClock と論理演算の例:

1. **`$expr:SystemCoreClock`** と入力します。
   この式は、グローバル変数の現在の値 **`25000000`** を返します。
2. **`$expr:SystemCoreClock == 25000000`** と入力し、**Enter** キーを押します。
   この式は、結果が true であるため **`1`** を返します。
3. **`$expr:SystemCoreClock != 25000000`** と入力し、**Enter** キーを押します。
   これは、結果が false であるため **`0`** を返します。

7.2.10 Scope resolution 演算子

Scope resolution 演算子(**`::`**) を使用して、イメージ、ファイル、namespace、またはクラス内の変数や関数にアクセスできます。

Scope resolution 演算子は、複数の AXF または ELF ファイル(たとえば、少なくとも 2 つの ELF ファイルで構成される TrustZone のサンプル) を含むプロジェクトをデバッグする必要がある場合に役立ちます。

シンボル式を使用して、特定のファイル(たとえば、**main** 関数) 内のシンボルを明示的に指定できます。**Scope resolution** の詳細については、Arm Debugger Command Reference を参照してください。たとえば、**Run and Debug Configuration visual editor** を使用してデバッグを開始する関数を選択するには、**Debug From** フィールドに次の式を指定します:

```
"hello.axf"::main
```

式は引用符で囲む必要があります。

launch.json ファイルに次の行が追加されます。

```
"debugFrom": "¥"hello.axf¥"::main"
```

引用符をエスケープするには、バックスラッシュを使用します。

式では、絶対または相対ファイルパスを使用することもできます。

Scope resolution 演算子は、ウォッチ式、**Debug Console**、または関数ブレークポイントでも役立ちます。

7.2.11 次のステップ

Visual Studio Code で利用できるデバッグ機能の詳細については、Visual Studio Code のドキュメントを参照してください。

7.3 スクリプトの操作

スクリプトを使用して、デバッグワークフローをカスタマイズし、タスクを自動化します。

Arm Debugger extension は、一般的な Python(CPython) や Jython などの言語を使用したスクリプト作成をサポートしています。Jython は Python の Java 実装であり、より大規模で複雑なスクリプトに最適です。

高度なスクリプトを作成、Jython テンプレートを使用するなどした後、**Debug Console** からスクリプトを実行する、あるいは"targetInitScript" および"debugInitScript" configuration option を使用して、project 内の tasks.json または launch.json ファイルからスクリプトを呼び出せます。

7.3.1 前提条件

Python および Jython スクリプトを使用するには、サポートされているバージョンの Python をマシンにインストールする必要があります。Arm Debugger extension は Python 2.7.2 以上をサポートしていますが、version 3 はサポートしていません。Python インタープリタをインストールするには、Visual Studio Code ドキュメントの説明を参照してください。

macOS の場合: Python のシステムインストールはサポートされていません。Homebrew などのパッケージ管理システムを使用することをお勧めします。

また、Visual Studio Code に Microsoft Python extension をインストールして、Python 言語の豊富なサポートを活用することもお勧めします。Microsoft Python extension は、IntelliSense と Microsoft Python Debugger のサポートを提供する Microsoft Pylance とともにインストールされます。

いずれかのバージョンの Python と Python extension をインストールしたら、Visual Studio Code のドキュメントの説明に従って、Command Palette から **Python: Select Interpreter** コマンドを使用して Python のバージョンを選択します。または、手動でインタープリタを指定します。

7.3.2 高度なスクリプトまたはデフォルトの Jython テンプレートを使用する

独自の Jython スクリプトを作成するか、使用開始時に役立つデフォルトのテンプレートを使用して、スクリプトを実行します。

手順

以下のいずれかを行います:

- .py 拡張子を持つ Jython ファイルを作成します。詳細については、Arm Development Studio User Guide の About Jython scripts および Jython script concepts and interfaces を参照してください。
- 提供されているテンプレートの 1 つを使用します。

a. **File** メニューから **New File…** を選択し、ウィンドウの上部に表示されるドロップダウン リストから **Jython Template** を選択します。
b. ドロップダウン リストから **Basic template** または **Advanced template** オプションを選択します。

   Basic template には、通常 Jython スクリプトの先頭に配置されるモジュールのインポートが含まれています。

   Advanced template は、ターゲットへの接続、アプリケーションのロードとアプリケーションの起動までを実行し、一部のレジスタ値を読み取ります。
c. ファイルを編集して、必要なスクリプトコマンドを追加します。
d. テンプレートをプロジェクトと同じフォルダに保存します。

次のステップ

Debug Console から **`source`** コマンドを使用して Jython スクリプトを実行できます。

例:

```
source myScripts¥myFile.py   # Run a Jython script from file myFile.py.
```

相対および絶対パスの両方をサポートします。
詳細については、Arm Debugger Command Reference を参照してください。

また、"**targetInitScript**" および"**debugInitScript**" configuration option を使用して、project 内の **tasks.json** または **launch.json** ファイルからスクリプトを呼び出すこともできます。

例:

```
"targetInitScript": "myScripts¥myFile.py"
```

相対および絶対パスの両方をサポートします。
Arm Debugger run configuration オプション および Arm Debugger configuration オプション を参照して下さい。

7.4 Arm Debugger extension の設定

Arm Debugger extension には次の設定があります。

名前	説明
Application Pattern	**arm-debugger .getApplicationFile** コマンドによってアプリケーションファイルとして認識されるファイルを指定する正規表現。デフォルトは ****/*.{axf,elf}**
Debugger Path	Arm Environment Manager extension 経由で Arm Debugger extension をインストールしなかった場合に使用する Arm Debugger 実行可能ファイルへのパス
Experimental Features	まだ製品クォリティに達していないプレビュー機能を有効にするオプション
Logging Verbosity	デバッガログの詳細レベル。これは、**Arm Debugger** output channel の **Output** タブ(**View** > **Output**) に表示される出力に影響します。また、Arm Debugger は **Logging Verbosity** の設定が **debug** に設定されている場合、デバッグセッションごとに追加のログファイルを生成します。ログファイルの名前は、**armdbg-<date>_<time>.log** になります。
Pack Asset Url	CMSIS DFP(Device Family Packs) がマシンにまだインストールされていない場合、ダウンロードするための URL

7.4.1 設定へのアクセス

Arm Debugger 設定にアクセスするには

手順

1. Settings を開きます。
   - Windows または Linux では、**File** > **Preferences** > **Settings** を選択します。
   - macOS では、**Code** > **Settings** > **Settings** を選択します。
2. **Extensions** カテゴリに移動し、**Arm Debugger** をクリックします。

8. Arm ツールのライセンスをアクティベートする

ツールチェインに含まれる Arm Compiler, Arm Debugger または Fixed Virtual Platforms (FVPs) などのツールを使用するには、ライセンスをアクティベートする必要があります。

ライセンスなしでライセンスツールを使用しようとすると、ステータスバーに **No Arm License** というステータスが表示され、ポップアップメッセージが表示されます。

エラーは、`vcpkg-configuration.json` ファイルと **Problems** タブ(**View** > **Problems**) にも表示されます。

1. 右下隅に表示されるポップアップで Manage Arm license をクリックします。
2. ウィンドウ上部のドロップダウンリストで、次のいずれかのオプションを選択します:
   - **Activate Arm Keil MDK Community Edition**:このオプションを選択すると、Keil MDK Community Edition のライセンスに切り替えられます。このライセンスは、非商用 project にのみ使用できます。
   - **Activate or manage Arm licenses**: このオプションを選択すると、Keil MDK Professional Edition や Keil MDK Essential Edition などの商用ライセンスに切り替えられます。このオプションを選択すると、**Arm License Management Utility** ウィンドウが開き、製品の actibation code を入力するか、ライセンスサーバを使用してライセンスをアクティベートできます。

Arm License Management Utility ウィンドウにアクセスしてライセンスを管理するには、**Command Palette** から `Environment: Manage tool licenses` コマンドを使用することもできます。

ライセンスのアクティベーションの詳細については、User-based Licensing User Guide 内の Activate your product using an activation code および Activate your product using a license server を参照してください。

Backwards compatibility のトピックでは、Keil MDK Professional の製品ライセンスを使用して、MDK の古いバージョン(MDK 5.36 以前)、PK51、PK166、DK251 のライセンスを取得する方法についても説明があります。

8.1 期限切れまたはキャッシュ期限切れライセンスに対するトラブルシューティング

有効期限が切れたライセンスまたはキャッシュの有効期限が切れたライセンスツールを使用しようとすると、ステータスバーに警告が表示され、右下隅にポップアップメッセージが表示されます。

ライセンスキャッシュの有効期限切れは、ネットワークの問題、デバイスの空き容量不足、または権限の問題により、ローカルライセンスを更新できなかった場合に発生します。

1. ポップアップで Manage Arm license をクリックします。
2. ライセンスに応じて、ウィンドウ上部のドロップダウンリストに次のいずれかのオプションが表示されます:
   - ライセンスの有効期限が切れている場合は、**Get help for expired license** オプションが表示されます。このオプションを選択すると、実行すべき手順に関する情報が表示されます。
   - ライセンスのキャッシュが期限切れの場合は、**Get help for cache-expired license** オプションが表示されます。このオプションを選択すると、実行すべき手順に関する情報が表示されます。

9. コマンドから CMSIS-Toolbox を使用する

CMSIS-Toolbox は、Keil Studio extensions に統合されているコマンドラインツールのセットです。
コマンドラインからスタンドアロンなツールとして使用することもできます。

keil.arm.com の公式サンプルを使用し、推奨どおりに Keil Studio Pack をインストールした場合、サンプル project を使用した Getting Started で説明されているように、CMSIS-Toolbox はすでにマシン上で使用可能になっています。

CMSIS-Toolbox が提供する主なツールのうち、コマンドラインで使用できるものは次のとおりです。

- **`cpackget`**: 開発環境で CMSIS-Packs をインストールおよび管理するために使用します。
- **`cbuild`**: ビルド呼び出し。プロジェクトを実行可能なバイナリイメージに変換するビルドプロセスを調整するために使用されます。cbuild はさまざまなツール(csolution、cpackget、および cbuildgen) を呼び出し、CMake コンパイルプロセスを開始します。
- **`csolution`**: project マネージャ。1 つ以上の関連する project で構成される組み込みアプリケーションのビルド情報を作成するために使用されます。

Build Tools のページでは、コマンドラインでこれらのツールを使用する方法について説明があります。

9.1 CMSIS-Toolbox をシステムの PATH に追加する

Environment Manager extension は、CMSIS-Toolbox をインストールし、ツールを Visual Studio Code システム PATH に追加します。
Environment Manager extension と vcpkg を使用せずに CMSIS-Toolbox をインストールする場合は、インストールパスをシステム PATH に追加するか、**Cmsis-csolution: Cmsis Toolbox Path** 設定を使用してパスを追加します。

9.2 Pack のサポート

CMSIS-Packs(software packs とも呼ばれます) には、特定のマイクロコントローラファミリまたは開発ボードで作業するために必要なものがすべて含まれています。
使用できる pack の種類は次のとおりです。

- Public packs。これらは、Arm またはシリコンベンダおよびソフトウェアベンダが作成し、公開されている pack です。Public Pack は、keil.arm.com の CMSIS-Packs のページから入手できます。
- Private packs。これらは、自身で作成したがまだ共有していない pack 、または他のユーザが個人的に共有した pack です。これらは、システムで使用できる local packs、または web 上にある remote packs です。

このセクションでは、さまざまな種類の pack の管理方法の概要を説明します。

Open-CMSIS-Pack のドキュメントでは、コマンドラインから pack を追加または削除するさまざまな方法について詳しく説明しています。Adding packs および Removing packs を参照してください。

9.2.1 Public Pack の追加

CMSIS Solution extension で利用可能な機能を使用して、public packs をインストールできます。詳細については、CMSIS-Packs のインストール を参照してください。

または、ターミナルから cpackget add コマンドを使用して、ベンダの package index にリストされている public pack の最新の公開バージョンをインストールすることもできます。package index ファイルには、ベンダによってホストおよび管理されているすべての CMSIS-Packs がリストされています。package index ファイルの詳細については、Open-CMSIS-Pack のドキュメント を参照してください。

keil.arm.com/packs で利用可能な CMSIS-Packs を調べ、利用可能なスニペットを使用して **`csolution.yml`** ファイルを更新し、**`cpackget add`** で pack をインストールします。

たとえば、次のコマンドは、public pack の最新のパブリックバージョンをインストールします:
`cpackget pack add Vendor::PackName`

ここでは以下の通りです:
- **`Vendor`**: CMSIS-Pack を作成したベンダ名
- **`PackName`**: CMSIS-Pack の名前

`cpackget add` を実行した後、Visual Studio Code をリロードして、ユーザインターフェイスに表示されるデータを更新します。

9.2.2 Local Pack の追加

ローカルで作成した CMSIS-Pack を操作するには、ターミナルから **`cpackget add`** コマンドを使用して Visual Studio Code をリロードし、CMSIS csolution extension が登録された pack を認識するようにします。pack のコンポーネントが **Software Components** ビューに表示され、ファイルの検証で新しいパックが認識されます。

たとえば、次のコマンドは PDSC(pack description) ファイルを使用して local pack を登録します:
`cpackget add /path/to/Vendor.PackName.pdsc`

ここでは以下の通りです:
- **`Vendor`**: CMSIS-Pack を作成したベンダ名
- **`PackName`**: CMSIS-Pack の名前

PDSC ファイルには、pack の内容に関する情報が含まれています。

cpackget add を実行して pack を pack ルートフォルダに追加した後、Visual Studio Code をリロードして、ユーザインターフェイスに表示されるデータを更新します。

Software Components ビューで、追加した pack のコンポーネントが表示されない場合は、**Cmsis-csolution: Pack Cache Path** の設定と **CMSIS_PACK_ROOT** 環境変数を確認してください。

9.2.3 Privete remote Pack の追加

Web 上にある remote pack をインストールするには、**cpackget add** コマンドと pack の URL を使用します。

たとえば、次のコマンドは、web からダウンロードできる pack のバージョンをインストールします:
```
cpackget add https://vendor.com/example/Vendor.PackName.x.y.z.pack
```

ここでは以下の通りです:
- **Vendor**: CMSIS-Pack を作成したベンダ名
- **PackName**: CMSIS-Pack の名前
- **x.y.z**: インストールする pack の特定のバージョン

cpackget add を実行した後、Visual Studio Code をリロードして、ユーザインターフェイスに表示されるデータを更新します。

9.2.4 Pack の削除

システムから pack を削除するには、**cpackget rm** を使用します。

たとえば、次のコマンドは、特定の pack のバージョンを削除します:
```
cpackget rm Vendor.PackName.x.y.z
```

ここでは以下の通りです:
- **Vendor**: CMSIS-Pack を作成したベンダ名
- **PackName**: CMSIS-Pack の名前
- **x.y.z**: インストールする pack の特定のバージョン

cpackget rm を実行した後、Visual Studio Code をリロードして、ユーザインターフェイスに表示されるデータを更新します。

10. 既知の問題とトラブルシューティング

Keil Studio extensions に関する既知の問題と、一般的な問題のトラブルシューティングの方法について説明します。

10.1 既知の問題

既知の問題は次のとおりです。

Arm CMSIS Solution extension

CMSIS Solution extension には、次の既知の問題があります:

- cdefaults.yml はサポートされていません。**Software Components** ビューと検証では、cdefaults ファイルで設定されたコンパイラは使用されません。

10.2 トラブルシューティング

extension の使用時に発生する可能性のある一般的な問題に対する解決策を提供します。

10.2.1 CMSIS-Toolbox が見つからないため ENOENT エラーでビルドが失敗する

extension が CMSISToolbox を見つけられないため、solution のビルドは ENOENT エラーで失敗します。

解決策

ポップアップメッセージの指示に従ってください。

Environment Manager はインストールされているが、環境に CMSIS-Toolbox が含まれていない場合:

- CMSIS-Toolbox を `vcpkg-configuration.json` ファイルに追加(**Add to Vcpkg** オプション) します。これにより、Environment Manager とともに CMSIS-Toolbox がインストールされます。
- または、CMSIS-Toolbox を手動でインストールして PATH に追加するか、設定でパスをコンフィギュレーション(**Open Installation Documentation** オプション) します。

Environment Manager がインストールされていない場合:

- **Extensions** ビューから Environment Manager をインストール(**Install Environment Manager** オプション) し、`vcpkg-configuration.json` ファイルを作成します。ステータスバーで **Arm Tools** をクリックし、**Add Arm tools Configuration To Workspace** を選択して visual editor を開き、ツールを選択します。
  これにより、project について保存できる `vcpkg-configuration.json` ファイルが作成されます。
- または、CMSIS-Toolbox を手動でインストールして PATH に追加するか、設定でパスをコンフィギュレーション(**Open Installation Documentation** オプション) します。

CMSIS-Toolbox のドキュメントに CMSIS-Toolbox を手動でインストールする方法について説明があります。

10.2.2 Windows で vcpkg artifacts のダウンロードとインストールに失敗する

Arm Environment Manager extension を使用すると、デフォルトのインストールフォルダのパス名の長さが原因で、Windows で vcpkg artifacts のダウンロードとインストールが失敗します。

解決策
こちらで説明されているように、Windows の設定で long path サポートを有効にします: Enable Long Paths in Windows 10, Version 1607, and Later

10.2.3 ビルドでツールチェインが見つからない

CMSIS Solution extension を使用すると、build output に `ld: unknown option: --cpu=Cortex-M4` などのエラーが表示されます。この例では、CMSIS-Toolbox は Arm Compiler の armlink ではなくシステムリンカを使用しようとしています。

解決策
1. Keil Studio Pack を使用せずに CMSIS Solution extension を個別にインストールした場合は、installing and setting up CMSISToolbox の手順に従ってください。特に、`CMSIS_COMPILER_ROOT` の環境変数が正しく設定されていることを確認してください。または、Keil Studio Pack をインストールして、Microsoft vcpkg による自動セットアップを利用することもできます。
2. solution をクリーンアップします。特に、`out` ディレクトリと `tmp` ディレクトリは必ず削除します。
3. 再度ビルドを行います。

10.2.4 接続している開発ボードまたはデバッグプローブが見つからない

開発ボードまたはデバッグプローブを接続しましたが、Device Manager extension でハードウェアを検出できません。

解決策
- **Device Manager**(Windows)、**System Information**(Mac)、または **hardinfo** などの Linux システムユーティリティツール(Linux) を実行し、ハードウェアの横に警告が表示されていないか確認します。警告はハードウェアのドライバがインストールされていないことを示している可能性があります。必要に応じて、ハードウェアに適したドライバを入手してインストールします。
- Windows の場合: ST の開発ボードとプローブには追加のドライバが必要です。これらは ST のサイトからダウンロードできます。
- Windows の場合: マシンに Mbed serial port driver がインストールされているかどうかを確認します。Mbed serial port driver は、Windows 7 でのみ必要です。シリアルポートは、Windows 8.1 以降ではすぐに使用できます。Mbed serial port driver は、デフォルトですべてのボードに DAPLink ファームウェアを要求するため、ドライバを更新するために Windows のネイティブ機能を上書きします。Arm では、ドライバが不要な場合はアンインストールすることを推奨します。または、無効にすることもできます。
  次のいずれかが可能です:
  ー Mbed serial port driver をアンインストールします(推奨): 管理者としてコマンドプロンプトを開き

ます。**mbedserial_x64.inf** および **mbedcomposite_x64.inf** ドライバを探して削除します。

```
pnputil /enum-drivers
pnputil /delete-driver {oemnumber.inf} /force
```

次に、USB ケーブルを使用してハードウェアを接続し、Windows Device Manager を開きます。**Ports (COM & LPT)** および **Universal Serial Bus controllers** で、**mbed** のエントリを探して、右クリックして両方をアンインストールします。最後にハードウェアを切断して再接続します。

- Mbed serial port driver を無効化します: Windows Device Manager を開きます。**Ports (COM & LPT)** で、**mbed** のエントリを探します。右クリックして **Properties** を選択します。Driver タブを選択して **Update Driver** をクリックします。**Browse my computer for drivers** をクリックしたあと、**Let me pick from a list of available drivers on my computer** をクリックします。**mbed Serial Port** の代わりに **USB Serial Device** を選択します。

- Linux の場合: udev rules は、USB のボードおよびデバイスにアクセスする権限を付与します。project をビルドしてハードウェア上で実行したり、デバッグしたりするには、udev rules をインストールする必要があります。

  pyOCD repository をクローンし、README.md ファイルの説明に従って、udev フォルダにある rules のファイルを**/etc/udev/rules.d/** にコピーします。**README** 内の指示に従ってください。

  udev ルールをインストールすると、接続されたハードウェアが Device Manager extension で検出されます。serial output にアクセスするときに、権限の問題が発生する場合があります。その場合は、**sudo adduser "$USER" dialout** を実行してからマシンを再起動してください。

- ボードまたはデバッグプローブのファームウェアバージョンがサポートされていることを確認し、ファームウェアを最新バージョンに更新します。詳細については、ファームウェアが Out-of-date と言われる を参照してください。
- ボードまたはデバイスが他のプロセスまたはツールによって要求されている可能性があります。(たとえば、Visual Studio Code の複数のインスタンスでボードまたはデバイスにアクセスしようとしている場合や、Visual Studio Code と別の IDE でボードまたはデバイスにアクセスしようとしている場合など)
- **Manage All Devices** 設定を有効にします。この設定により、コンピュータに接続されている任意の USB ハードウェアを選択できます。デフォルトでは、Device Manager extension では、既知のベンダのハードウェアにのみアクセスできます。
  1. settings を開きます:
     - Windows または Linux の場合: **File** > **Preferences** > **Settings** に移動
     - MacOS の場合: **Code** > **Settings** > **Settings** に移動
  2. settings を開きます: **Device-manager: Manage All Devices** の設定を探してチェックボックスをオンにします。

10.2.5 ファームウェアが Out-of-date と言われる

開発ボードまたはデバッグプローブを接続すると、ファームウェアが古いことを示すポップアップ メッセージが表示されます。

解決策

ボードまたはデバッグプローブのファームウェアを最新バージョンに更新します。

- DAPLink の場合。**daplink.io** でボードまたはプローブが見つからない場合は、ハードウェアの製造元の Web サイトを確認してください。
- ST-Link の場合。Windows 上の ST 開発ボードおよびプローブには追加のドライバが必要であることに注意してください。これらは ST のサイト からダウンロードできます。
- その他の WebUSB-enabled CMSIS-DAP ファームウェアの更新については、ボードまたはデバッグプローブのベンダにお問い合わせください。

FRDM-KL25Z ボードを使用していて、標準の DAPLink ファームウェア更新手順が機能しない場合は、こちらの手順に従ってください。(Windows 7 または Windows XP の場合)

ファームウェア更新の詳細については、Debug Probe Firmware Update Information Application Note も参照してください。

11. フィードバックを送る

Keil Studio extensions に関してご提案がある場合、または問題を発見した場合は Arm までご報告ください。https://www.keil.arm.com/support/ にアクセスし、**Keil Studio for VS Code** のカテゴリで提供されているリンクを使用してください。

所有権通知

所有権通知については、オリジナルの Arm Keil Studio Visual Studio Code Extensions User Guide 内の **Proprietary Notice** のセクションを参照してください。

製品およびドキュメント情報

製品およびドキュメント情報については、オリジナルの Arm Keil Studio Visual Studio Code Extensions User Guide 内の **Product and document information** のセクションを参照してください。

凡例

以下のサブセクションでは、Arm のドキュメントで使用される表記規則について説明します。

Glossary

"Arm Glossary"は、Arm のドキュメントで使用されている用語と、それらの用語の定義のリストです。Arm Glossary には、Arm において用いる意味が一般に受け入れられている意味と異なる場合を除き、業界標準の用語は含まれていません。

詳細については、Arm® Glossary を参照してください: developer.arm.com/glossary

表記規則

Arm のドキュメントでは、特定の意味を伝えるために、表記規約を使用します。

表記	用法
斜体 italic	引用
強調文字 bold	メニュー名などのインターフェイス要素 信号名 記述リストの用語(適切な場合)
等幅表記 (monospace)	コマンド、ファイル名、プログラム名、ソースコードなど、キーボードから入力できるテキスト
下線付き等幅表記 (monospace underline)	コマンドまたはオプションの許可される省略形。完全なコマンドまたはオプション名の代わりに下線付きのテキストの内容を入力できます。
<and>	コードまたはコードフラグメント内に現れるアセンブラ構文内で置き換え可能な用語を囲っています。 例: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
小型英大文字 (SMALL CAPITALS)	Arm Glossary で定義されている特定の技術的意味を持つ用語。 例:**IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, および **UNPREDICTABLE** など

	推奨事項。これらの推奨事項に従わないと、システムの故障や損傷につながるおそれがあります。
	システム要件。これらの要件を守らないと、システムの故障や破損につながる可能性があります。
	システムの要件。これらの要件に従わないと、システムが故障したり破損したりすることが予測されます。
	注意が必要な重要な情報
	作業をより簡単に、より良好に、またはより迅速に行うことができる有用な tip
	お読みいただいている情報に関係する重要なポイントのリマインダ

タイミング図

役立つリソース

このドキュメントには、この製品に固有の情報が含まれています。その他の情報の詳細については、次のリソースを参照してください。

Arm のドキュメントへのアクセスは、その Confidentiality (機密性) に応じて異なります:

- Non-Confidential (非機密) ドキュメントは developer.arm.com/documentation から入手できます。以下の表に各ドキュメントに対するオンラインバージョンのリンクがあります。
- Confidential (機密) ドキュメントは製品パッケージを通じてライセンシにのみ提供されます。

Arm product resources	Document ID	Confidentiality
Arm Keil Microcontroller Development Kit (MDK) Getting Started Guide	109350	Non-Confidential
Arm Keil Studio Cloud User Guide	102497	Non-Confidential
uVision User Guide	101407	Non-Confidential