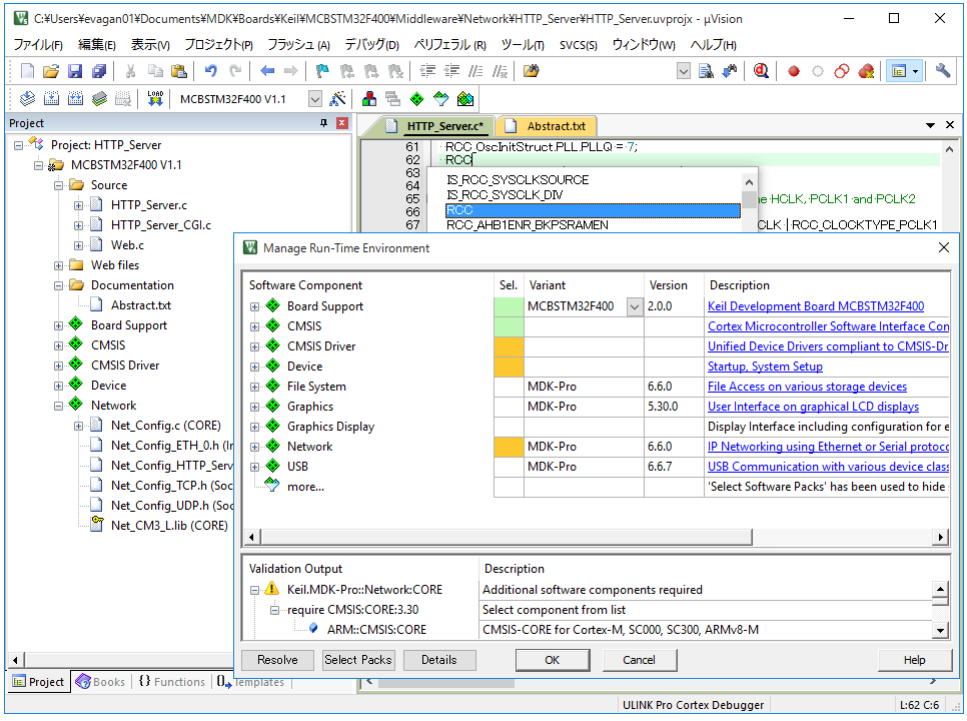
ARM® KEIL®
Microcontroller Tools

# スタートガイド

## ARM® Cortex®-M マイクロコントローラ用の MDK バージョン 5 でアプリケーションを作成する



**注**
本書は、Microsoft Windows、ハードウェア、Cortex®-M プロセッサの命令セットに精通している読者を対象に作成しています。

# 序章

ARM® Keil® からご利用いただける MDK バージョン 5 マイクロコントローラ開発キットをご利用いただき、ありがとうございます。Cortex-M プロセッサベースの組み込みアプリケーションを開発するための最適なソフトウェアツールを提供するために、当社ではソフトウェアエンジニアリングを簡単で生産性の高いものにするためのツールを設計しています。ARM では、ULINK™ デバッグ/トレースアダプタや幅広い評価ボードなどの補完製品もご用意しております。MDK は、さまざまなサードパーティ製ツール、スタータキット、デバッグアダプタを使用して拡張することができます。

## 章の概要

本書では、まず MDK のインストールから説明し、ソフトウェアコンポーネントと、プロジェクトを開始してからハードウェアのデバッグを行うまでの完全なワークフローについて説明していきます。本書は以下の章から構成されています。

**MDK の紹介** - MDK コアやソフトウェアパックの概要について、そしてサンプルプロジェクトを使用した製品のインストールについて説明します。

**CMSIS** - Cortex-M ベースのマイクロコントローラ上で実行される組み込みアプリケーション用のソフトウェアフレームワークです。ソフトウェアの再利用を簡単に行うための一貫したソフトウェアインタフェースとハードウェア抽象化レイヤが揃っています。

**ソフトウェアコンポーネント Compiler（コンパイラ）** - さまざまな標準 I/O チャネルの I/O 関数のターゲット変更について説明します。

**アプリケーションを作成する** - CMSIS とデバイス関連ソフトウェアコンポーネントを使用してプロジェクトを作成および変更する方法をご案内します。実践型チュートリアルに、ツールオプションを設定するためのメインのコンフィギュレーションダイアログが示されます。

**アプリケーションのデバッグ** - 実際のハードウェアでアプリケーションをデバッグするプロセスと接続方法について説明します。

**ミドルウェア** - MDK-Professional エディションのユーザがご利用いただけるミドルウェアについての詳細を説明します。

**Middleware の使用** - MDK-Professional で使用可能なミドルウェアを使用したアプリケーションの作成方法について説明します。すぐに作業を開始するために不可欠なヒントやコツが記載されています。

# 目次

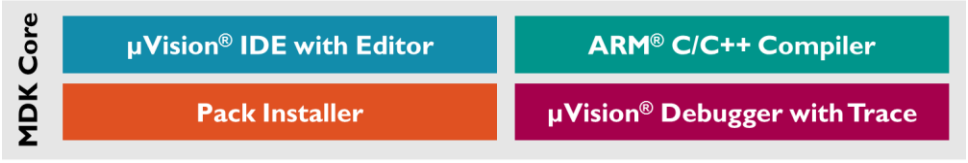
# MDK の紹介

Keil マイクロコントローラ開発キット (MDK) は、ARM Cortex-M プロセッサベースのデバイス向けの組み込みアプリケーションを作成する際に役立ちます。MDK は強力でありながら、習得が簡単で開発システムを使用します。MDK バージョン 5 は、MDK コアとデバイス固有のソフトウェアパックで構成されており、お使いのアプリケーションの要件に基づいてダウンロードし、インストールすることができます。

MDK バージョン 5 では、www.keil.com/mdk5/legacy から従来のサポートをインストールすると、MDK バージョン 4 プロジェクトを使用することができます。こうすることで、ARM7、ARM9、および Cortex-R プロセッサベースのデバイスに対するサポートが追加されます。

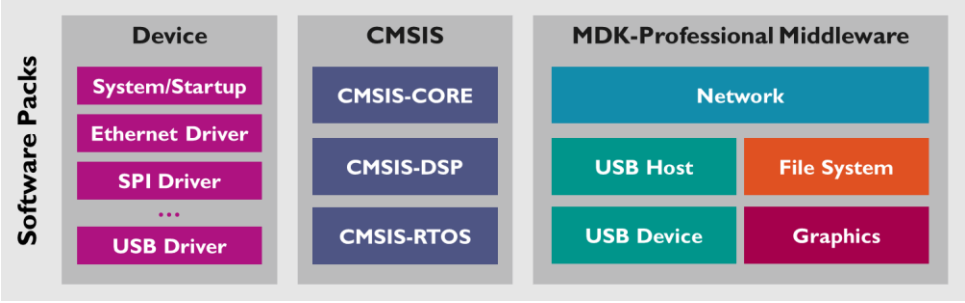
# MDK コア

MDK コアには、Cortex-M プロセッサベースのマイクロコントローラデバイス用に組み込みアプリケーションを作成、ビルド、デバッグするのに必要なすべてのコンポーネントが含まれています。MDK コアにいつでも追加できるソフトウェアパックの管理は、Pack Installer によって行われます。こうすることで、ツールチェーンに関係なく、新しいデバイスのサポートとミドルウェアのアップデートを行うことができます。



# ソフトウェアパック

ソフトウェアパックには、デバイスサポート、CMSIS ライブラリ、ミドルウェア、ボードサポート、コードテンプレート、サンプルプロジェクトが含まれています。




# MDKのエディション

MDK には、C/C++ またはアセンブリ言語を使用してアプリケーションの作成とデバッグを行うツールと環境が用意されており、さまざまなエディションにてご利用いただけます。各エディションには、µVision® IDE、デバッガ、コンパイラ、アセンブラ、リンカ、ミドルウェアライブラリ、および CMSIS-RTOS RTX が含まれています。

- **MDK-Professional** には、洗練された組み込みアプリケーション用の広範なミドルウェアライブラリに加え、**MDK-Standard** のすべての機能が含まれています。
- **MDK-Standard** は、Cortex-M、一部の Cortex-R、ARM7 および ARM9 プロセッサベースのマイクロコントローラをサポートしています。
- **MDK-Cortex-M** は、Cortex-M プロセッサベースのマイクロコントローラをサポートしています。
- **MDK-Lite** は、コードサイズが 32 KB に制限されており、製品の評価、小規模プロジェクト、教育市場を対象としています。

http://www.keil.com/mdk5/selector にある製品選択ガイドを見ると、各エディションで使用可能な機能の概要がわかります。

### ライセンスの種類

**MDK-Lite** を除き、MDK エディションにはライセンスコードを使用したアクティブ化が必要です。以下のライセンスの種類を使用できます。

- **シングルユーザライセンス**（ノードロック）では、1 人の開発者が 2 台のコンピュータで同時に製品を使用できます。
- **フローティングユーザライセンス**または**FlexLM ライセンス**では、複数の開発者が数台のコンピュータで同時に製品を使用できます。
- **7 日間 MDK-Professional 試用版ランセンス**では、コードサイズの制限なしでミドルウェアを包括的にテストできます。

詳細については、『*ライセンスユーザガイド*』（www.keil.com/support/man/docs/license）を参照して下さい。

# インストール

## ソフトウェアとハードウェアの要件

以下は MDK のハードウェアとソフトウェアの最低要件です。

- Microsoft Windows（32 ビットまたは 64 ビット）オペレーティングシステム搭載の PC
- 4 GB の RAM および 8 GB のハードディスク空き容量
- 1280 x 800 以上のディスプレイ解像度、マウスまたはその他のポインティングデバイス

## MDK コアのインストール

**MDK バージョン 5** を www.keil.com/download の［Product Downloads（製品ダウンロード）］からダウンロードし、インストーラを実行します。

指示に従って、MDK コアをローカルコンピュータにインストールします。このインストールにより、ARM **CMSIS** と **MDK-Professional Middleware** のソフトウェアパックも追加されます。

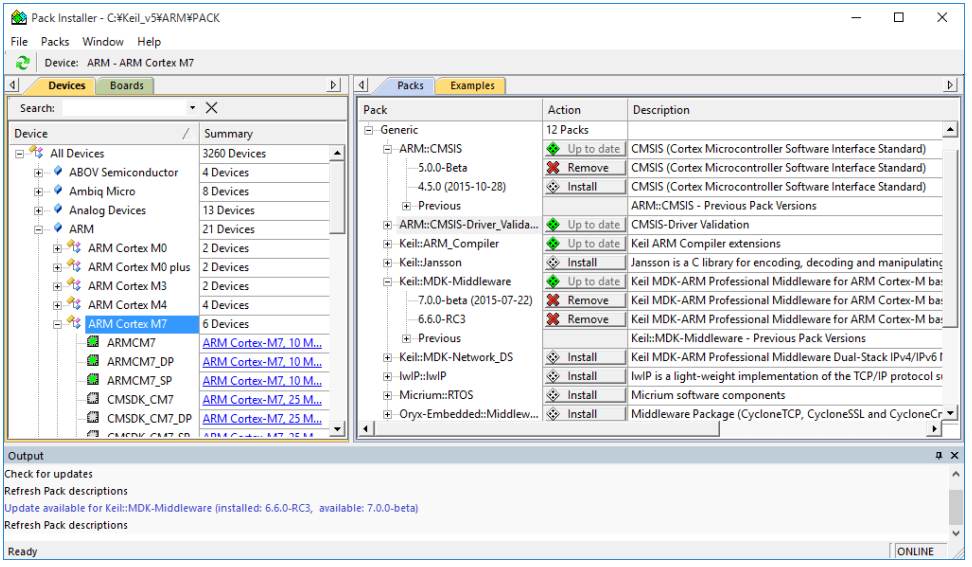
MDK コアのインストールが完了すると、Pack Installer が自動的に起動するため、ここから補足のソフトウェアパックを追加できます。最低でも、お使いのターゲットマイクロコントローラデバイスをサポートするソフトウェアパックをインストールする必要があります。

## ソフトウェアパックのインストール

Pack Installer は、ローカルコンピュータでソフトウェアパックを管理するためのユーティリティです。

Pack Installer はインストール時に自動で実行されますが、µVision で［Project（プロジェクト）］ - ［Manage（管理）］ - ［Pack Installer］の順にメニュー項目を選択して実行することもできます。デバイスやサンプルプロジェクトにアクセスするには、お使いのターゲットデバイス、または評価ボードに関連したソフトウェアパックをインストールして下さい。

*注*
*公開版のソフトウェアパックの詳細を得るために、Pack Installer は www.keil.com/pack に接続します。*



Pack Installer の下にあるステータスバーには、インターネット接続とインストールの進行状況に関する情報が表示されます。

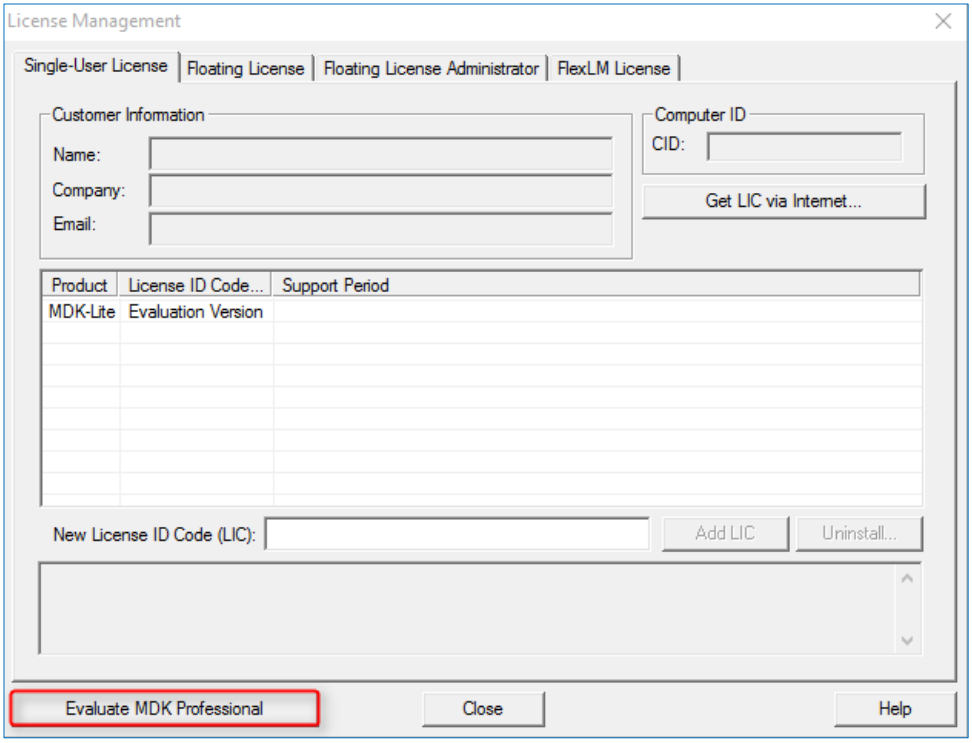
ヒント：デバイスデータベース（www.keil.com/dd2）には、使用可能なすべてのデバイスの一覧と、関連したソフトウェアパックをダウンロードするためのアクセスが記載されています。Pack Installer で www.keil.com/pack にアクセスできない場合は、メニューコマンドの［File（ファイル）］ - ［Import（インポート）］を使用するか、*.PACK ファイルをダブルクリックして、手動でソフトウェアパックをインストールすることができます。

# MDK-Professional 試用版ライセンス

MDK には、MDK-Professional の 7 日間**無料**試用版ライセンスが組み込まれています。このライセンスを使うとコードサイズの制限がなくなるため、ミドルウェアを包括的に探索し、テストすることができます。

管理者権限で µVision を起動します。

1. µVision で、**[File（ファイル）]** - **[License Management（ライセンス管理）...]** に移動し、**[Evaluate MDK Professional（MDK Professional の評価）]** をクリックします。



2. 次の画面で、[Start MDK Professional Evaluation for 7 Days (MDK Professional の 7 日間評価を開始)] をクリックします。インストールが完了すると、有効期限の日時に関する情報が画面に表示されます。

*注*
*このライセンスは FlexLM に基づいているため、7 日間の MDK Professional 試用版をアクティブにすると、[FlexLM License (FlexLM ライセンス)] タブの [Use Flex Server (Flex サーバを使用する)] というオプションが有効になります。*

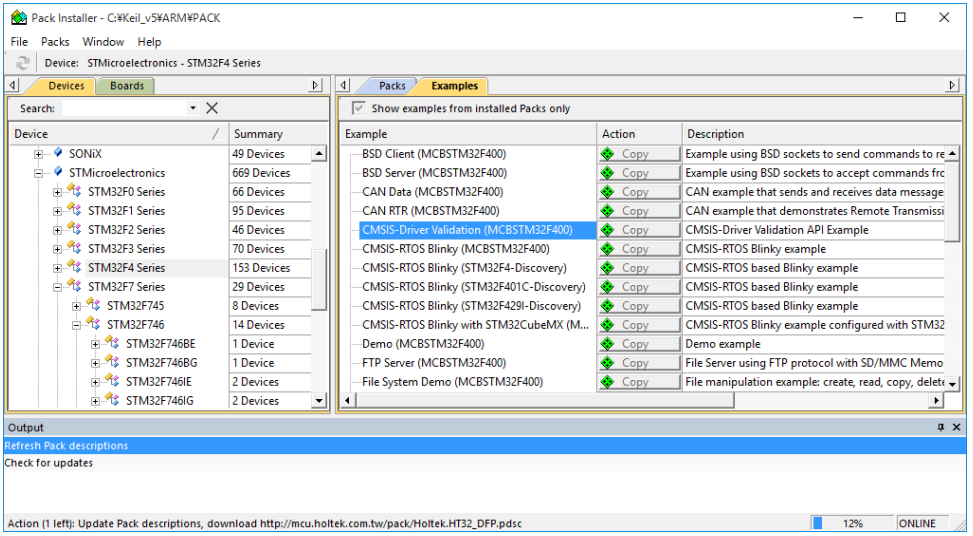
## サンプルプロジェクトを使用したインストールの検証

お使いのデバイスのソフトウェアパックを選択、ダウンロード、インストールしたら、ソフトウェアパックに付属のサンプルの 1 つを使用して、インストールを検証することができます。ソフトウェアパックのインストールを検証する際は、通常、ターゲットボードで LED が点滅する *Blinky* サンプルを使用することをお勧めします。

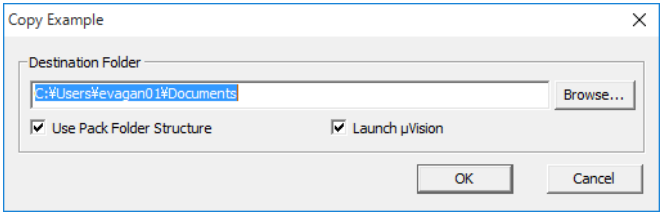
ヒント：http://www.keil.com/mdk5 のスタートガイドビデオで、評価キットの接続方法と作業方法を確認して下さい。

## サンプルプロジェクトのコピー

Pack Installer の［Examples（サンプル）］タブを選択します。ツールバーのフィルタでサンプル一覧を絞り込みます。




［Copy（コピー）］をクリックし、作業ディレクトリの［Destination Folder（デスティネーションフォルダ）］の名前を入力します。




*注*
*選択した作業ディレクトリにサンプルプロジェクトをコピーする必要があります。*

- [Launch µVision (µVision の起動)] を有効にして、サンプルプロジェクトを IDE で直接開きます。
- [Use Pack Folder Structure (パックフォルダ構造を使用)] を有効にして、サンプルプロジェクトを共通フォルダにコピーします。コピーしておくと、ファイルが他のサンプルプロジェクトで上書きされるのを防ぐことができます。[Use Pack Folder Structure (パックフォルダ構造を使用)] を無効にして、サンプルパスの複雑さを軽減させます。
- [OK] をクリックして、コピープロセスを開始します。

## µVision を使用したサンプルアプリケーションの使用

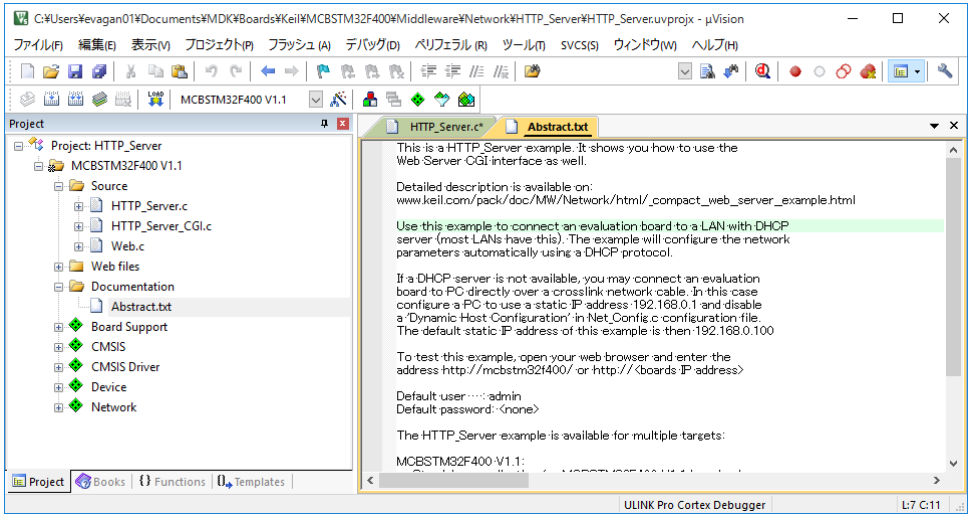
これで、µVision が起動してサンプルプロジェクトがロードされます。
ここでは以下のことができます。


アプリケーションをビルドします。関連するソースファイルのコンパイルとリンクが行われます。


アプリケーションをダウンロードします。通常はデバイスのオンチップの Flash ROM 上にダウンロードされます。


デバッガを使用して、ターゲットハードウェア上でアプリケーションを実行します。

これらのタスクの実行手順は、順を追って説明されています。サンプルのコピーが終わると、µVision が起動し、以下のような画面が表示されます。




ヒント：ほとんどのサンプルプロジェクトには、操作方法とハードウェアコンフィギュレーションに関する重要な情報が記載された「*Abstract.txt*」というファイルが含まれています。

## アプリケーションのビルド

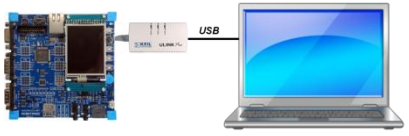
 [Rebuild（再ビルド）] ツールバーボタンを使用して、アプリケーションをビルドします。

[Build Output（ビルド出力）] ウィンドウには、ビルドプロセスに関する情報が表示されます。エラーのないビルドについては、プログラムサイズに関する情報が表示されます。

```
Build Output
*** Using Compiler 'V5.06 update 1 (build 61)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'STM32F407 Flash'
compiling Blinky.c...
compiling Thread_LED.c...
compiling Buttons_F4Discovery.c...
compiling LED_F4Discovery.c...
compiling RTX_Conf_CM.c...
compiling stm32f4xx_hal.c...
compiling stm32f4xx_hal_cortex.c...
compiling stm32f4xx_hal_gpio.c...
compiling stm32f4xx_hal_pwr.c...
compiling stm32f4xx_hal_pwr_ex.c...
compiling stm32f4xx_hal_rcc.c...
compiling stm32f4xx_hal_rcc_ex.c...
assembling startup_stm32f407xx.s...
compiling system_stm32f4xx.c...
linking...
Program Size: Code=11216 RO-data=592 RW-data=84 ZI-data=3348
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:09
```

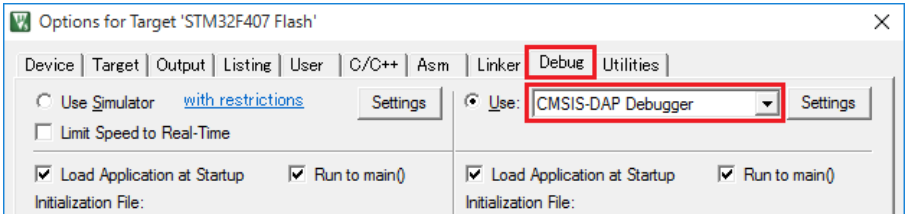
## アプリケーションのダウンロード

一般的に USB で接続するデバッグアダプタを使用して、ターゲットハードウェアをお使いのコンピュータに接続します。複数の評価ボードにオンボード型のデバッグアダプタが装備されています。




次に、デバッグアダプタの設定を確認します。通常、サンプルプロジェクトは評価キットに合わせて事前に設定されているため、これらの設定を変更する必要はありません。

ツールバーで［Options for Target（ターゲットのオプション）］をクリックし、［Debug（デバッグ）］タブを選択します。使用している評価ボードに適したデバッグアダプタが選択され、有効になっていることを確認します。例えば、［CMSIS-DAP Debugger（CMSIS-DAP デバッガ）］は、複数のスタータキットの一部となっているデバッグアダプタです。

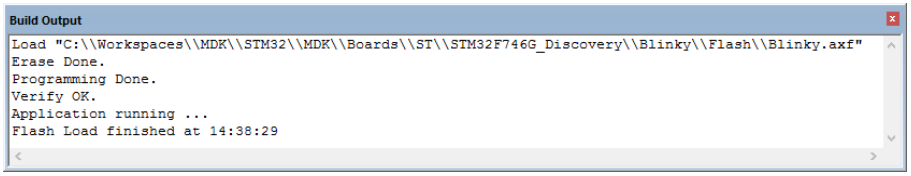



［Load Application at Startup（起動時にアプリケーションをロード）］を有効にして、デバッグセッションを開始するたびにアプリケーションを µVision デバッガにロードします。

［Run to main()（main() まで実行）］を有効にすると、main() 関数の
最初の実行可能ステートメントまで命令が実行されます。この命令は、リセットのたびに実行されます。

ヒント：［Settings（設定）］ボタンをクリックして通信設定を確認し、ターゲットハードウェアの問題を診断します。詳細を確認するには、ダイアログの［Help（ヘルプ）］ボタンをクリックします。問題がある場合は、スタータキットのユーザガイドを参照して下さい。

ツールバーの [Download（ダウンロード）] をクリックすると、アプリケーションがターゲットハードウェアにロードされます。




[Build Output（ビルド出力）] ウィンドウには、ダウンロードの進行状況に関する情報が表示されます。

### アプリケーションの実行

ツールバーの [Start/Stop Debug Session（デバッグセッションの開始/停止）] をクリックすると、ハードウェア上のアプリケーションのデバッグが開始されます。

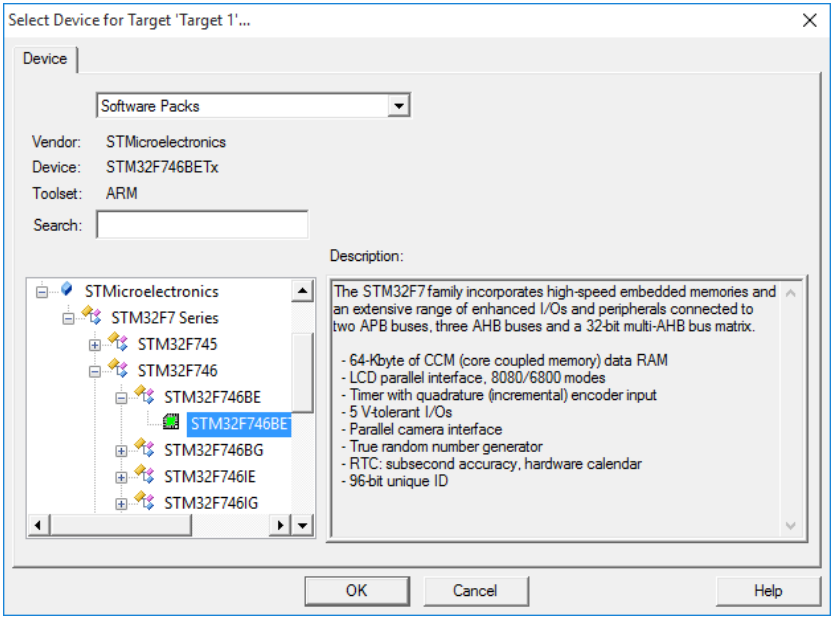
デバッグツールバーの [Run（実行）] をクリックすると、アプリケーションの実行が開始されます。ターゲットハードウェアの LED が点滅します。

## ソフトウェアパックの使用

ソフトウェアパックには、アプリケーションの構成要素として使用可能なマイクロコントローラデバイスとソフトウェアコンポーネントに関する情報が含まれています。

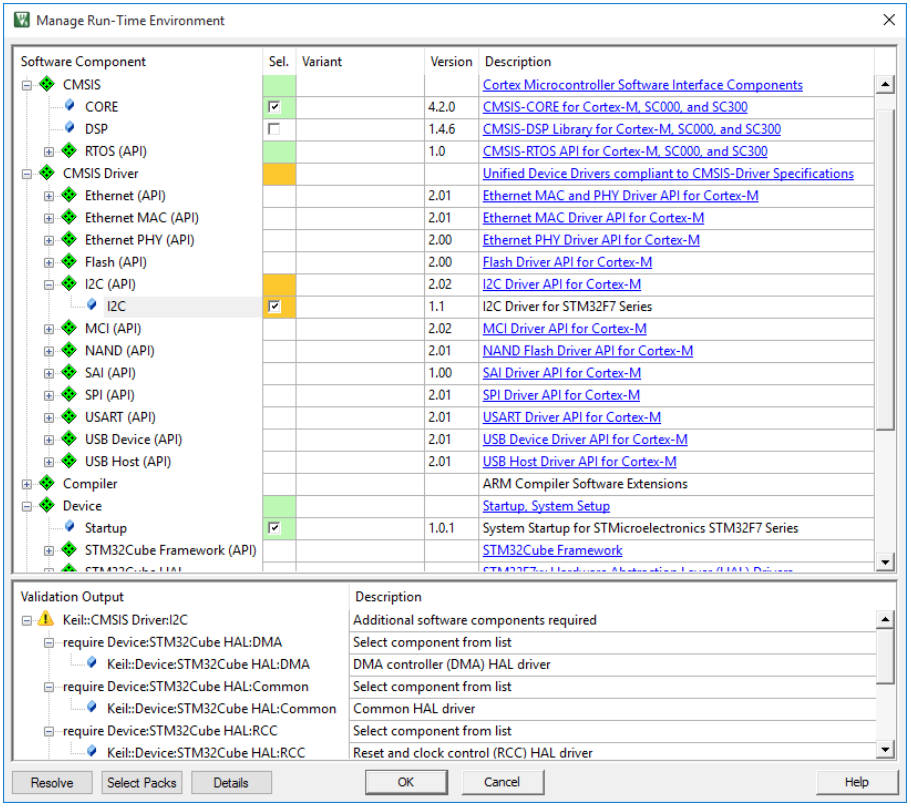
デバイス情報によって開発ツールが事前設定され、選択したデバイスに関連したオプションのみが表示されます。

µVision を起動し、［Project（プロジェクト）］ - ［New µVision Project（新しい µVision プロジェクト）］メニューを使用します。プロジェクトディレクトリを選択してプロジェクト名を指定したら、ターゲットデバイスを選択します。




ヒント：インストールされているソフトウェアパックの一部になっているデバイスのみが表示されます。デバイスが見つからない場合は、Pack Installer を使用して関連したソフトウェアパックを追加します。検索ボックスは、デバイス一覧を絞り込む際に役立ちます。

- デバイスの選択が完了すると、**[Manage Run-Time Environment（実行時環境の管理）]** ウィンドウに、このデバイスに関連したソフトウェアコンポーネントが表示されます。




ヒント： *[Description（説明）]* 列のリンクをクリックすると、各ソフトウェアコンポーネントのマニュアルにアクセスできます。

---

*注*

*コンポーネントの参照には、「**::<コンポーネントクラス>:<グループ>:<名前>**」という表記が用いられます。例えば、**::CMSIS:CORE** は、上のダイアログで CMSIS-CORE コンポーネントが選択されていることを示します。*

---

## ソフトウェアコンポーネントの概要

以下の表は、典型的なインストールに含まれているソフトウェアコンポーネントを示しています。選択したデバイスによっては、これらのソフトウェアコンポーネントの一部が［Manage Run-Time Environment（実行時環境の管理）］ウィンドウに表示されないことがあります。追加のソフトウェアパックをインストールした場合は、さらに多くのソフトウェアコンポーネントを使用できます。

| ソフトウェアコンポーネント | 説明 | ページ |
|---|---|---|
| **Board Support（ボードサポート）** | 評価ボードのペリフェラルへのインタフェース。 | 59 |
| **CMSIS** | CORE、DSP、CMSIS-RTOS などの CMSIS インタフェースコンポーネント。 | 29 |
| **CMSIS Driver（CMSIS ドライバ）** | ミドルウェアとユーザアプリケーション用の統一されたデバイスドライバ。 | 52 |
| **Compiler（コンパイラ）** | I/O 操作のターゲットを変更する際（printf スタイルのデバッグなど）の ARM コンパイラ固有のソフトウェアコンポーネント。 | 52 |
| **Device（デバイス）** | システムスタートアップ、および低レベルのデバイスドライバ。 | 62 |
| **File System（ファイルシステム）** | さまざまなストレージデバイスの種類でファイルにアクセスするためのミドルウェアコンポーネント。 | 109 |
| **Graphics（グラフィックス）** | グラフィカルユーザインタフェースを作成するためのミドルウェアコンポーネント。 | 112 |
| **Network（ネットワーク）** | イーサネット、またはシリアルプロトコルを使用した TCP/IP ネットワークを設定するためのミドルウェアコンポーネント。 | 107 |
| **USB** | 標準の USB デバイスクラスをサポートする USB ホストと USB デバイスのためのミドルウェアコンポーネント。 | 110 |

## ソフトウェアパックを使用した製品ライフサイクル管理

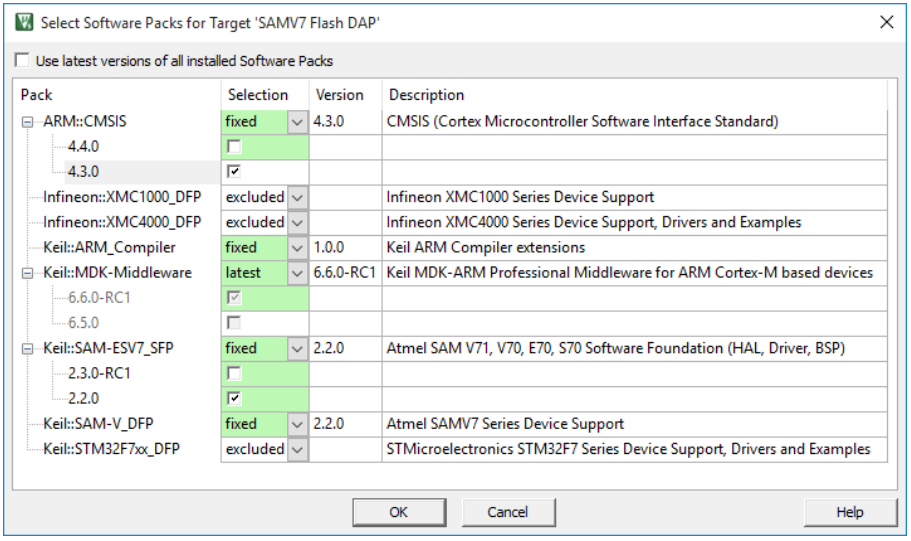
MDK では、複数バージョンのソフトウェアパックをインストールできます。これによって、多くのプロジェクトでよく使用される製品ライフサイクル管理（PLM）が可能になります。

PLM には以下の 4 つのフェーズがあります。

- **概念**：主なプロジェクト要件を定義し、関数プロトタイプを使用して調査すること。
- **設計**：最終的な技術的特徴と要件に基づいて、プロトタイプのテストと製品の実装を行うこと。
- **リリース**：製品を製造して市場に投入すること。
- **サービス**：顧客のサポートも含めた製品のメンテナンスを行うこと。最終的にフェーズアウトするか、耐用年数終了を迎えます。

概念フェーズと設計フェーズでは、新機能を盛り込んでバグをすばやく修正できるように、通常は最新版のソフトウェアパックを使用します。製品リリースの前には、ソフトウェアコンポーネントをテスト済みの既知の状態にフリーズさせます。製品のサービスフェーズでは、固定バージョンのソフトウェアコンポーネントを使用して、現場の顧客をサポートします。

**［Select Software Packs（ソフトウェアパックの選択）］**ダイアログを使用すると、プロジェクト内の各ソフトウェアパックのバージョンを管理する際に役立ちます。



プロジェクトが完了したら、[Use latest version of all installed Software Packs（インストールされているすべてのソフトウェアパックの最新版を使用する）] オプションを無効にして、[Selection（選択範囲）] の設定でソフトウェアパックを指定します。

- latest（最新）：最新版のソフトウェアパックを使用します。新しいバージョンのソフトウェアパックがインストールされると、ソフトウェアコンポーネントが更新されます。
- fixed（固定）：インストールされているソフトウェアパックのバージョンを指定します。プロジェクトターゲットのソフトウェアコンポーネントでこれらのバージョンが使用されます。
- excluded（除外）：このソフトウェアパックのソフトウェアコンポーネントは使用されません。

色表示は、現在のプロジェクトターゲットにおけるソフトウェアコンポーネントの使用状況を示しています。

このパックの一部のソフトウェアコンポーネントが使用されています。

このパックの一部のソフトウェアコンポーネントが使用されていますが、パックは除外されます。

このパックのソフトウェアコンポーネントは使用されていません。

## ソフトウェアバージョンコントロールシステム（SVCS）

µVision では、ソフトウェアバージョンコントロールシステム（SVCS）をサポートするために、GIT、SVN、CVS 用のテンプレートファイルを用意しています。

アプリケーションノート 279「Using Git for Project Management with µVision（µVision でのプロジェクト管理における Git の使用）」（www.keil.com/appnotes/docs/apnt_279.asp）には、ソフトウェアパックを使用してプロジェクトをバージョン管理する際の堅牢なワークフローの確立方法が記載されています。

# マニュアルへのアクセス

MDK には、オンラインマニュアルとコンテキストヘルプが用意されています。µVision の［Help（ヘルプ）］メニューを使用すると、『*µVision ユーザガイド*』、スタートガイドのマニュアル、コンパイラ、リンカ、アセンブラリファレンスガイドなどが収録されたメインのヘルプシステムが表示されます。

**多くのダイアログ**には、マニュアルにアクセスできて、ダイアログのオプションや設定に関する説明を確認できる、コンテキスト依存の［Help（ヘルプ）］ボタンがあります。

エディタで F1 キーを押すと、RTOS 関数、コンパイラディレクティブ、ライブラリルーチンなどの言語要素のヘルプにアクセスすることができます。［Output（出力）］ウィンドウのコマンドラインで F1 キーを押すと、デバッグコマンドや、一部のエラーメッセージと警告メッセージに関するヘルプが表示されます。

［Books（ブック）］ウィンドウには、デバイスのリファレンスガイド、データシート、ボードのマニュアルなどが収められています。独自のマニュアルを追加して、［Project（プロジェクト）］ - ［Manage（管理）］ - ［Components, Environment, Books（コンポーネント、環境、ブック）］ - ［Books（ブック）］の順にメニューを選択して［Books（ブック）］ウィンドウでそのマニュアルを有効にすることもできます。

［Manage Run-Time Environment（実行時環境の管理）］ダイアログでは、*［Description（説明）］*列のリンクからマニュアルにアクセスできます。

［Project（プロジェクト）］ウィンドウで、ソフトウェアコンポーネントグループを右クリックし、対応する要素のマニュアルを開くことができます。

『µVision ユーザガイド』には、www.keil.com/support/man/docs/uv4 からオンラインでアクセスできます。

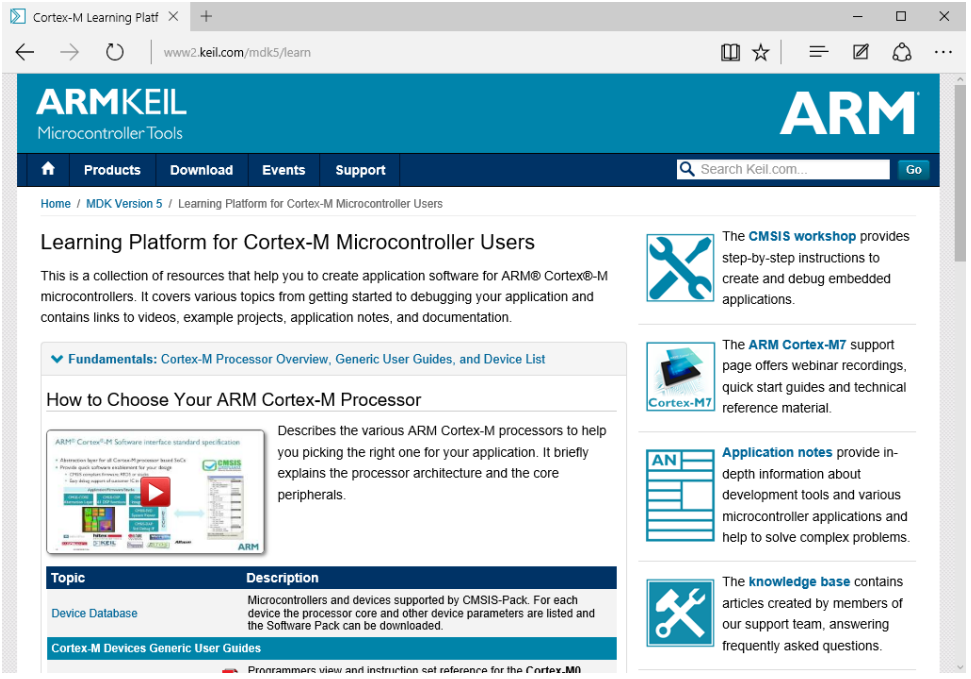
# サポートのリクエスト

当社のソフトウェアに関してご提案がある場合や問題が生じた場合は、当社までご報告下さい。サポートおよび情報チャネルには、www.keil.com/support からアクセスできます。

問題を報告する際は、ライセンスコードがある場合はライセンスコードと製品のバージョンをお知らせ下さい。この情報は、µVision メニューで［Help（ヘルプ）］ - ［About（バージョン情報）］の順に選択して確認できます。

# 学習用プラットフォーム

当社では、ARM Cortex ベースのマイクロコントローラのプログラミングについて詳しく知ることができる Web サイトをご用意しています。チュートリアル、ビデオ、その他のマニュアル、さらに他の Web サイトへの役立つリンクが掲載されており、www.keil.com/learn からご利用いただけます。



# クイックスタートガイド

クイックスタートガイドを参照すれば、お使いのターゲットハードウェアをすばやく使用できます。MDK を使用して開発ボードを立ち上げるのに必要な手順について説明されているほか、必要なソフトウェアパックの一覧や、デバッグアダプタを統合するためのドライバ要件も記載されています。

*注*
*使用可能なクイックスタートガイドはすべて www.keil.com/mdk5/qsg から参照できます。*

# CMSIS

**Cortex マイクロコントローラソフトウェアインタフェース規格**（CMSIS）は、Cortex-M ベースのマイクロコントローラ上で実行される組み込みアプリケーションの基礎となるソフトウェアフレームワークを提供します。CMSIS により、プロセッサとペリフェラルとの間に一貫したシンプルなソフトウェアインタフェースを実現でき、ソフトウェアの再利用が簡素化されるため、マイクロコントローラ開発者にとって習熟に要する期間が短縮されます。

*注*
*本章は参照セクションです。「アプリケーションの作成」の章（60 ページ）に、CMSIS を使用したアプリケーションコードの作成方法が記載されています。*

さまざまなシリコンベンダとソフトウェアベンダが緊密に連携して定義された CMSIS は、インタフェースペリフェラル、リアルタイムオペレーティングシステム、およびミドルウェアコンポーネントに対して共通のアプローチを提供します。

CMSIS アプリケーションソフトウェアコンポーネントは以下のとおりです。

- **CMSIS-CORE**：Cortex-M プロセッサコアとペリフェラルの API を定義します。整合性のあるシステム起動コードが付属しています。例外、割り込み、デバイスペリフェラルを使用するネイティブプロセッサでアプリケーションを作成して実行するために必要なソフトウェアコンポーネントは、**::CMSIS:CORE** と **::Device:Startup** だけです。
- **CMSIS-RTOS** RTX：標準化されたリアルタイムオペレーティングシステム API を提供しているほか、サポート対象の RTOS システム全体で機能するソフトウェアテンプレート、ミドルウェア、ライブラリ、その他のコンポーネントを使用できるようにします。本書では、CMSIS-RTOS RTX 実装の使用方法について説明しています。
- **CMSIS-DSP**：固定小数点（小数 q7、q15、q31）や単精度浮動小数点（32 ビット）などのさまざまなデータ型に対応し、60 を超える関数を持つデジタル信号処理（DSP）用のライブラリコレクション。
- **CMSIS** ドライバ：ミドルウェアスタックとユーザアプリケーションのペリフェラルドライバインタフェースについて記述しているソフトウェア API。CMSIS ドライバ API は、汎用で特定の RTOS に依存しない設計になっているため、サポート対象の幅広いマイクロコントローラデバイス間で再利用することができます。

# CMSIS-CORE

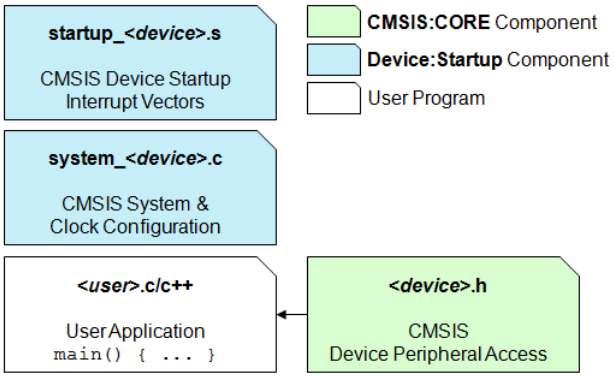
このセクションでは、Cortex-M プロセッサでネイティブに実行されるアプリケーションにおける CMSIS-CORE の使用方法について説明します。このような種類の操作では、リアルタイムオペレーティングシステムを使用しないため、「*ベアメタル*」とも呼ばれています。

# CMSIS-CORE の使用

CMSIS を使用するネイティブ Cortex-M アプリケーションでは、**::CMSIS:CORE** というソフトウェアコンポーネントを使用します。このコンポーネントは、**::Device:Startup** というソフトウェアコンポーネントと併用する必要があります。これらのコンポーネントには、以下の集中管理ファイルがあります。

*注*
*実際のファイル名では、〈デバイス〉はマイクロコントローラデバイスの名前になります。*

- リセットハンドラと例外ベクタを持つ *startup_〈デバイス〉.s* ファイル
- 基本のデバイス設定（クロックおよびメモリ BUS）の *system_〈デバイス〉.c* コンフィギュレーションファイル。




- *〈デバイス〉.h* には、マイクロコントローラデバイスへのユーザコードアクセス用のファイルが含まれています。

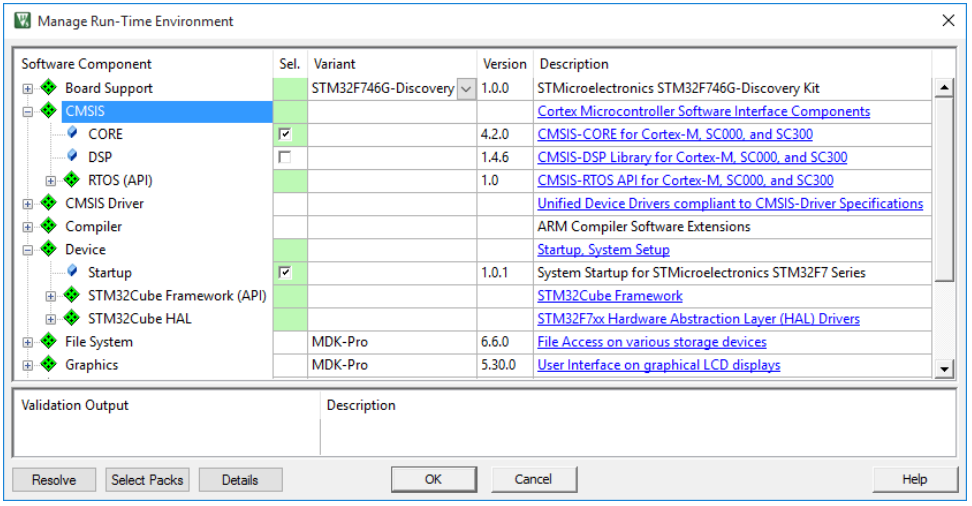
*〈デバイス〉.h* ヘッダファイルが C ソースファイルに含まれており、以下を定義しています。

- 標準化されたレジスタレイアウトを使用したペリフェラルアクセス。
- 割り込みと例外、およびネスト型ベクタ割り込みコントローラ（NVIC）へのアクセス。
- スリープモードをアクティブにするなどの特殊な命令を生成する組み込み関数。

- 定期的なタイマの割り込みを構成および開始するための Systick タイマ（SYSTICK）関数。
- オンチップの CoreSight™ を使用した printf 形式の I/O および ITM 通信に対するデバッグアクセス。

## プロジェクトへのソフトウェアコンポーネントの追加

::CMSIS:CORE コンポーネントと ::Device:Startup コンポーネントのファイルは、µVision の [Manage Run-Time Environment（実行時環境の管理）] ダイアログを使用してプロジェクトに追加されます。以下の図と同じようにソフトウェアコンポーネントを選択して下さい。




µVision 環境によって関連ファイルが追加されます。

## ソースコードの例

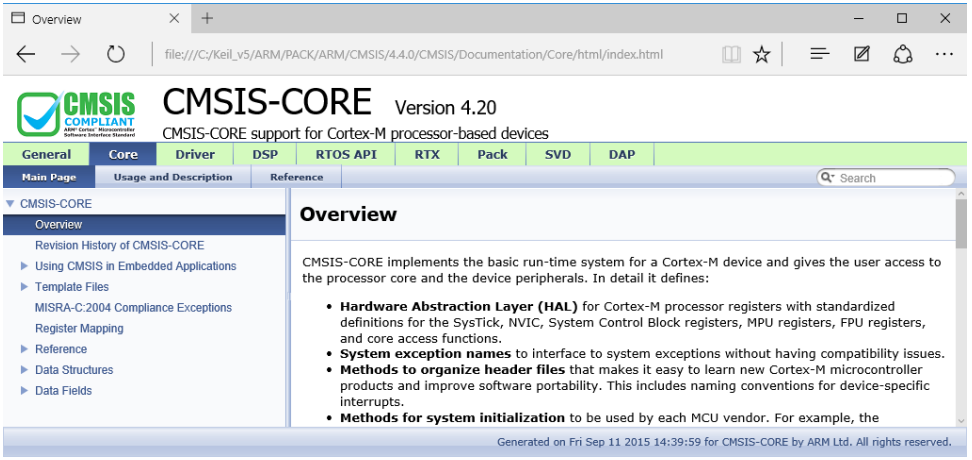
以下のソースコード行は、CMSIS-CORE レイヤの使用状況を示しています。

CMSIS-CORE レイヤの使用例

```
#include "stm32f4xx.h"                // File name depends on device used

uint32_t volatile msTicks;            // Counter for millisecond Interval
uint32_t volatile frequency;          // Frequency for timer

void SysTick_Handler (void) {         // SysTick Interrupt Handler
  msTicks++;                          // Increment Counter
}

void WaitForTick (void) {
  uint32_t curTicks;
  curTicks = msTicks;                 // Save Current SysTick Value
  while (msTicks == curTicks) {       // Wait for next SysTick Interrupt
    __WFE ();                         // Power-Down until next Event
  }
}

void TIM1_UP_IRQHandler (void) {      // Timer Interrupt Handler
  ; // Add user code here
}

void timer1_init(int frequency) {     // Set up Timer (device specific)
  NVIC_SetPriority (TIM1_UP_IRQn, 1); // Set Timer priority
  NVIC_EnableIRQ (TIM1_UP_IRQn);      // Enable Timer Interrupt
}

// Configure & Initialize the MCU
void Device_Initialization (void) {
  if (SysTick_Config (SystemCoreClock / 1000)) {     // SysTick 1ms
    : // Handle Error
  }
  timer1_init (frequency);        // Setup device-specific timer
}

// The processor clock is initialized by CMSIS startup + system file
int main (void) {                 // User application starts here
  Device_Initialization ();       // Configure & Initialize MCU

  while (1) {                     // Endless Loop (the Super-Loop)
    __disable_irq ();             // Disable all interrupts
    // Get_InputValues ();
    __enable_irq ();              // Enable all interrupts
    // Process_Values ();
    WaitForTick ();               // Synchronize to SysTick Timer
```

```
  }
}
```

詳細については、［Project（プロジェクト）］ウィンドウの［CMSIS］グループを右クリックして、［Open Documentation（マニュアルを開く）］を選択するか、CMSIS-CORE マニュアル（http://www.keil.com/cmsis/core）を参照して下さい。




# CMSIS-RTOS RTX

このセクションでは、CMSIS-RTOS RTX リアルタイムオペレーティングシステムを紹介し、その利点について説明した後、この RTOS のコンフィギュレーション設定と機能について解説します。

*注*
*MDK は多くのサードパーティ製 RTOS ソリューションと互換性があります。ただし、CMSIS-RTOS RTX は MDK に完全に統合されており、多機能で、組み込みシステムの要件に合わせて十分にカスタマイズされています。*

# ソフトウェアの概念

組み込みアプリケーションには、以下の 2 つの基本設計概念があります。

- **無限ループ設計：**無限ループとしてプログラムが実行されます。割り込みサービスルーチン（ISR）によりデータ処理を伴うタイムクリティカルなジョブが実行されている間、プログラム関数（スレッド）がループ内から呼び出されます。
- **RTOS 設計：リアルタイムオペレーティングシステム（RTOS）**を使用したいくつかのスレッドが実行されます。RTOS は、スレッド間通信と時間管理関数を提供します。プリエンプティブ RTOS により、タイムクリティカルなデータ処理が最優先スレッドで実行されるので、割り込み関数の複雑さが緩和されます。

## 無限ループ設計

無限ループでの組み込みプログラムの実行は、単純な組み込みアプリケーションに適したソリューションです。通常はハードウェア割り込みによってトリガされるタイムクリティカルな機能は、必須のデータ処理が実行される ISR 内で実行されます。メインループには、タイムクリティカルではなく、バックグラウンドで実行される基本演算のみが含まれています。

## RTOS カーネルのメリット

RTOS カーネルは、CMSIS-RTOS RTX と同様に、並列実行スレッド（タスク）の概念に基づいています。実際には、アプリケーションで複数の異なるタスクに応じることが必要になります。RTOS ベースのアプリケーションは、お使いのソフトウェアでこのモデルを再現します。これにはさまざまな利点があります。

- スレッドの優先順位と実行時のスケジューリングは、実証済みのコードベースを使用して RTOS カーネルによって処理されます。
- RTOS は、スレッド間通信のための明確に定義されたインタフェースです。

- プリエンプティブなマルチタスクの概念により、大規模な開発チームでもアプリケーションの進行的な拡張が簡素化されます。重要性の高いスレッドの応答時間を気にせずに、新機能を追加することができます。
- 無限ループソフトウェアの概念では、発生した割り込みに対してポーリングが行われることがよくあります。これに対し、RTOS カーネル自体は割り込み駆動型であるため、ポーリングの大部分を排除することができます。このため、CPU をスリープ状態にしたり、スレッド処理の頻度を増やしたりすることができます。

最近の RTOS カーネルは割り込みシステムに対して透過的です。これは、厳しいリアルタイム要件を持つシステムでは必須条件です。IRQ からタスクへの通信に通信機能を使用し、割り込みを上位半分/下位半分で処理することができます。

## CMSIS-RTOS RTX の使用

CMSIS-RTOS RTX はライブラリとして実装されており、ヘッダファイル*cmsis_os.h* からその機能を公開します。

CMSIS-RTOS RTX の実行は、最初のスレッドとして *main()* 関数から始まります。このため、開発者はスレッドを内部作成する他のミドルウェアライブラリを初期化しながらも、ユーザアプリケーションの残りの部分では **main** スレッドのみを使用できるというメリットがあります。この結果、RTOS の使用状況はアプリケーションプログラマには表示されませんが、ライブラリで CMSIS-RTOS RTX 機能を使用することができます。

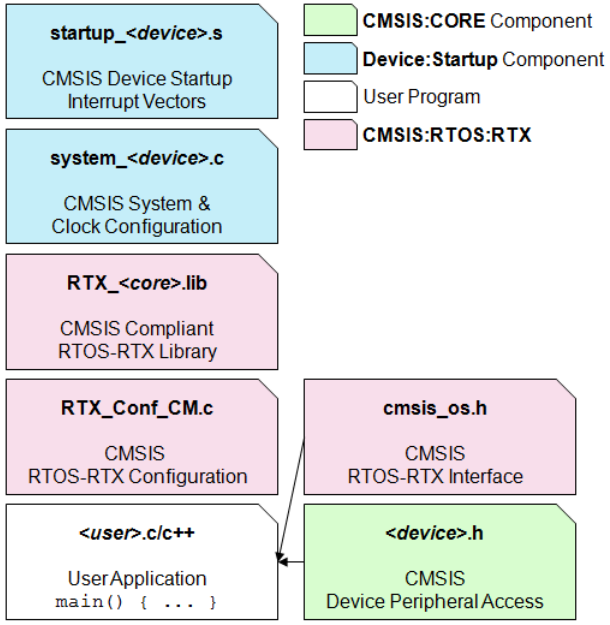
ソフトウェアコンポーネント **::CMSIS:RTOS:Keil RTX** は、**::CMSIS:CORE** コンポーネントおよび **::Device:Startup** コンポーネントと併用する必要があります。これらのコンポーネントを選択すると、以下の集中管理 CMSIS-RTOS RTX ファイルが得られます。

---

*注*
*実際のファイル名では、〈デバイス〉がマイクロコントローラデバイスの名前になります。*
*〈デバイスコア〉はデバイスプロセッサファミリを表します。*

---

- *RTX_<コア>.lib* ファイルは、RTOS 関数を持つライブラリです。
- スレッドオプション、タイマコンフィギュレーション、RTX カーネル設定を定義する *RTX_Conf_CM.c* コンフィギュレーションファイル。
- ヘッダファイル *cmsis_os.h* は、RTX 機能をユーザアプリケーションに公開します。
- *main()* 関数はスレッドとして実行されます。




これらのファイルがプロジェクトの一部になったら、開発者は CMSIS-RTOS RTX 関数の使用を開始することができます。以下のコードサンプルは、CMSIS-RTOS RTX 関数の使用状況を示しています。

**CMSIS-RTOS RTX 関数の使用例**

```
#include "cmsis_os.h"                       // CMSIS RTOS header file

void job1 (void const *argument) {          // Function 'job1'
  // execute some code
  osDelay (10);                             // Delay execution for 10ms
}

osThreadDef(job1, osPriorityLow, 1, 0);     // Define job1 as thread

int main (void) {
  osKernelInitialize ();                    // Initialize RTOS kernel
  // setup and initialize peripherals
  osThreadCreate (osThread(job1), NULL);    // Create the thread

  osKernelStart ();                         // Start kernel & job1 thread
}
```

## ヘッダファイル cmsis_os.h

*cmsis_os.h* ファイルは CMSIS-RTOS RTX のテンプレートヘッダファイルで、以下が含まれています。

- CMSIS-RTOS API 関数定義。
- パラメータと戻り値型の定義。
- CMSIS-RTOS API 関数で使用されているステータスと優先順位の値。
- スレッドや、ミューテックス、セマフォ、メモリプールなどの他のカーネルオブジェクトを定義するマクロ。

CMSIS-RTOS 関数に対して一意の名前空間を指定するために、すべての定義には os という接頭辞が付いています。os_ という接頭辞が付いている定義はアプリケーションコードで使用されず、このヘッダファイルに対してローカルになります。モジュールに属するすべての定義と関数はグループ化され、スレッドの場合は osThread と付くなど、共通の接頭辞が付けられます。

## オブジェクト定義の定義と参照

#define osObjectsExternal を使用すると、オブジェクトは外部シンボルとして定義されます。これによって、以下に示すように、プロジェクト全体で一貫したヘッダファイルを作成できます。

**ヘッダファイルの例：osObjects.h**

```
#include "cmsis_os.h"                                 // CMSIS RTOS header

extern void thread_1 (void const *argument);          // Function prototype
osThreadDef (thread_1, osPriorityLow, 1, 100);        // Thread definition

osPoolDef (MyPool, 10, long);                         // Pool definition
```

このヘッダファイルは「*osObjects.h*」と呼ばれ、C/C++ ソースファイルにインクルードされた場合にすべてのオブジェクトを定義します。ヘッダファイルのインクルードの前に #define osObjectsExternal が存在する場合、オブジェクトは外部シンボルとして定義されます。このため、1 つの一貫したヘッダファイルをプロジェクト全体で使用することができます。

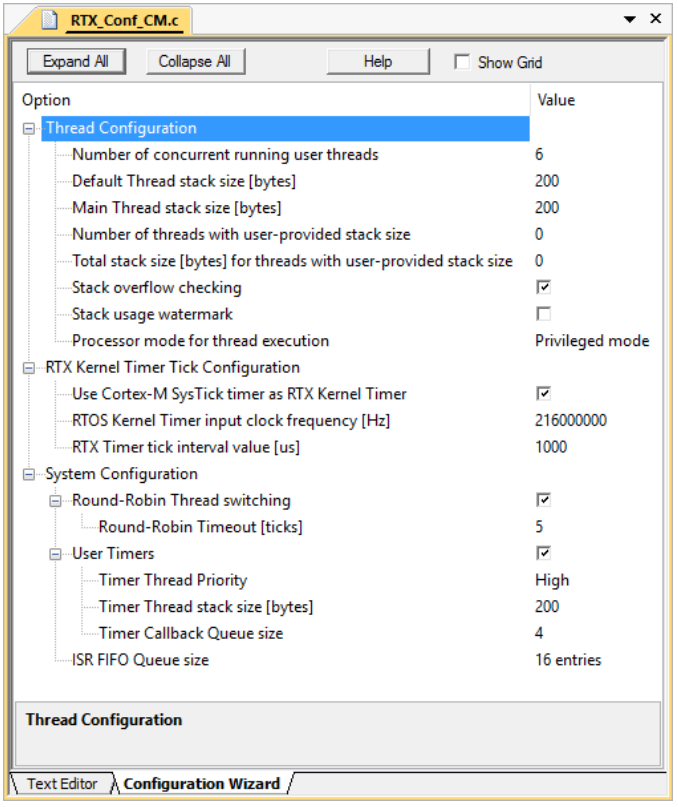
C ファイルでの一貫したヘッダファイルの使用

```
#define osObjectExternal      // Objects defined as external symbols
#include "osObjects.h"        // Reference to the CMSIS-RTOS objects
```

詳細については、オンラインマニュアル（www.keil.com/cmsis/rtos）の「Header File Template: cmsis_os.h（ヘッダファイルテンプレート：cmsis_os.h）」セクションを参照して下さい。

## CMSIS-RTOS RTX コンフィギュレーション

*RTX_Conf_CM.c* ファイルには、CMSIS-RTOS RTX のコンフィギュレーションパラメータが含まれています。このファイルのコピーは、RTX コンポーネントを使用したあらゆるプロジェクトの一部になります。



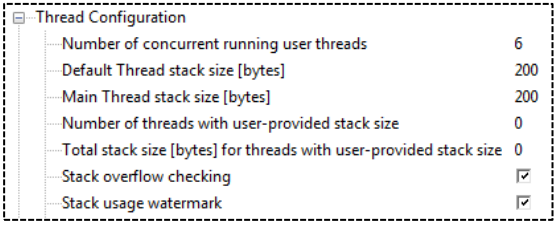
スレッドスタックのパラメータの設定、Tick タイマの設定、ラウンドロビンタイムスライスの設定、スレッドのユーザタイマ動作の定義などを行うことができます。

コンフィギュレーションオプションの詳細については、**[Manage Run-Time Environment（実行時環境の管理）]** ウィンドウから RTX のマニュアルをご覧下さい。「**Configuration of CMSIS-RTOS RTX（CMSIS-RTOS RTX のコンフィギュレーション）**」セクションに、使用可能なすべての設定の説明が記載されています。お使いのアプリケーションでの適応を必要とする、最も重要な設定を以下に示します。

## スレッドスタックコンフィギュレーション

スレッドは *osThreadDef()* 関数を使用してコード内で定義されます。*stacksz* パラメータはスレッドのスタック要件を指定するもので、スタックの割り当て方法に影響を及ぼします。CMSIS-RTOS RTX では、次の 2 つの方法で *RTX_Conf_CM.c* ファイルにスタック要件を割り当てることができます。

- 固定メモリプールを使用する：*stacksz* パラメータが 0 の場合は、**[Default Thread stack size [bytes]（デフォルトのスレッドのスタックサイズ［バイト］）]** に指定された値で、スレッド関数のスタックサイズが設定されます。
- ユーザ空間を使用する：*stacksz* が 0 でない場合は、ユーザ空間からスレッドスタックが割り当てられます。このユーザ空間の合計サイズは、**[Total stack size [bytes] for threads with user-provided stack size（ユーザが指定したスタックサイズのスレッドの合計スタックサイズ［バイト］）]** で指定されます。




**[Number of concurrent running threads（同時実行中のスレッド数）]** では、固定サイズのメモリプールからスタックを割り当てるスレッドの最大数が指定されます。

**[Default Thread stack size [bytes]（デフォルトのスレッドのスタックサイズ [バイト]）]** では、ユーザ指定のスタックなしで定義されたスレッドのスタックサイズ（ワード単位）が指定されます。

**[Main Thread stack size [bytes]（メインスレッドのスタックサイズ [バイト]）]** は、*main()* 関数のスタック要件です。

**[Number of threads with user-provided stack size（ユーザが指定したスタックサイズのスレッド数）]** では、特定のスタックサイズで定義されたスレッド数が指定されます。

**[Total stack size [bytes] for threads with user-provided stack size（ユーザが指定したスタックサイズでのスレッドの合計スタックサイズ [バイト]）]** は、特定のスタックサイズで定義されたすべてのスレッドの要件（ワード単位）を合計したものです。

**[Stack overflow checking（スタックオーバーフローの確認）]** では、スレッドの切り替え時にスタックオーバーフローを確認できます。このオプションを有効にすると、スレッド切り替えにかかる時間が少し長くなります。

**[Stack usage watermark（スタック消費量の透かし）]** では、スレッド作成時に、スレッドスタックが透かしパターンで初期化されます。これにより、各スレッドのスタック消費量を監視して（スレッド切り替え時だけではなく）、スレッド内でスタックオーバーフローの問題を見つけるのに役立てることができます。このオプションを有効にすると、*osThreadCreate()* の実行にかかる時間が非常に長くなります。

---

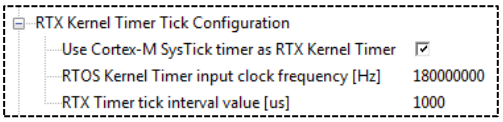
*注*

*これらの設定は慎重に検討して下さい。十分なメモリを割り当てなかったり、十分なスレッドを指定しなかったりすると、アプリケーションは機能しません。*

---

## RTX カーネルのタイマ Tick コンフィギュレーション

CMSIS-RTOS RTX 関数は、**タイマ tick の値**から派生した遅延をミリ秒単位で指定します。1 ミリ秒の間隔が生成されるようにタイマ Tick を設定することをお勧めします。これよりも長い間隔を設定するとエネルギー消費量は減りますが、タイムアウトの粒度に影響します。



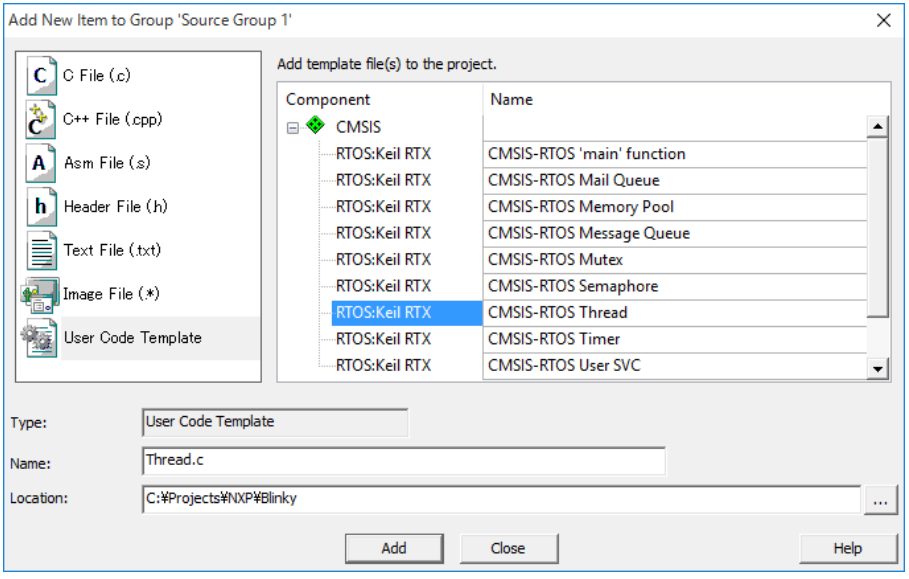
［Use Cortex-M Systick timer as RTX Kernel Timer（Cortex-M Systick タイマを RTX カーネルタイマに使用する）］を有効にすることをお勧めします。こうすることで、クロックソースとしてプロセッサクロックを使用する組み込み SysTick タイマが選択されます。この場合、［RTOS Kernel Timer input clock frequency（RTOS カーネルのタイマ入力クロック周波数）］を起動ファイル *system_<デバイス>.c* の CMSIS 変数 *SystemCoreClock* と同じにする必要があります。

詳細については、オンラインドキュメントの「Configuration of CMSIS-RTOS RTX - Tick Timer Configuration（CMSIS-RTOS RTX のコンフィギュレーション - Tick タイマコンフィギュレーション）」セクションを参照して下さい。

## CMSIS-RTOS ユーザコードテンプレート

MDK には、アプリケーションの C ソースコードを作成するために使用できるユーザコードテンプレートが用意されています。

☞ ［Project（プロジェクト）］ウィンドウで、グループを右クリックし、［Add New Item to Group（新規項目をグループに追加）］を選択します。［User Code Template（ユーザコードテンプレート）］を選択し、任意のテンプレートを選択して、［Add（追加）］をクリックします。



## CMSIS-RTOS RTX API 関数

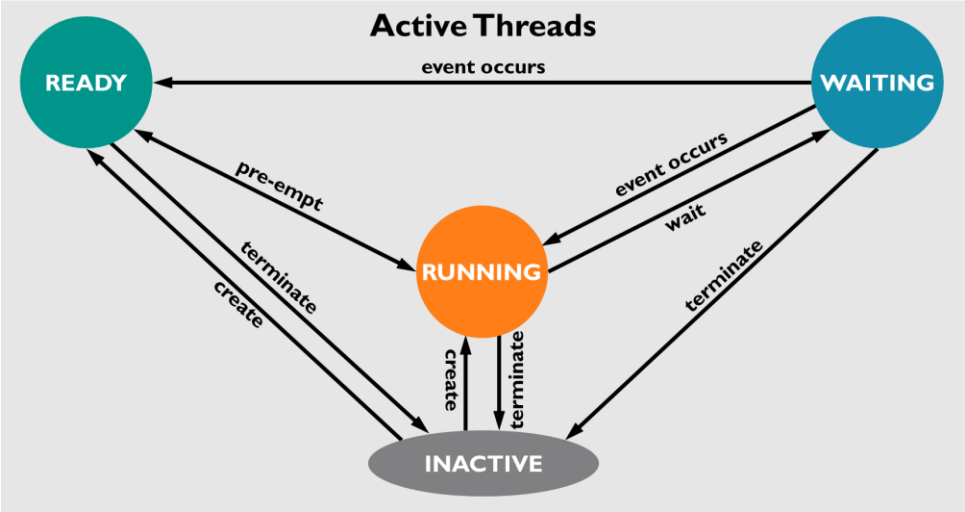
以下の表には、CMSIS-RTOS RTX で使用できるさまざまな API 関数のカテゴリが示されています。

| API カテゴリ | 説明 |
|---|---|
| **スレッド管理** | スレッド関数を定義、作成、および制御します。 |
| **タイマ管理** | タイマとコールバック関数を作成および制御します。 |
| **信号管理** | シグナルフラグを制御または待機します。 |
| **ミューテックス管理** | スレッドの実行とミューテックスを同期させます。 |
| **セマフォ管理** | 共有リソースへのアクセスを制御します。 |
| **メモリプール管理** | 固定サイズのメモリプールを定義および管理します。 |
| **メッセージキュー管理** | メッセージを制御、送受信、または待機します。 |
| **メールキュー管理** | メールを制御、送受信、または待機します。 |

ヒント： www.keil.com/pack/doc/CMSIS/RTX/html/index.html で見ることができる CMSIS-RTOS RTX チュートリアルでは、API 関数の使用方法について説明します。

## スレッド管理

スレッド管理関数を使用すると、システムで独自のスレッド関数を定義、作成、および制御できます。関数 *main()* はシステムの初期化時に開始される特殊なスレッド関数で、初期優先順位は *osPriorityNormal* です。




上記の図からわかるように、CMSIS-RTOS RTX は、スレッドがスケジュールされていると想定しています。スレッドの状態は、以下のように変化します。

- スレッドは *osThreadCreate()* 関数を使用して作成されます。これにより、スレッドは「READY（準備完了）」または「RUNNING（実行中）」状態になります（スレッドの優先順位に基づく）。
- CMSIS-RTOS はプリエンプティブです。最も高い優先順位が設定されたアクティブなスレッドは、イベントを待機していない場合、「RUNNING（実行中）」スレッドになります。スレッドの初期優先順位は *osThreadDef()* を使用して定義されますが、実行中に *osThreadSetPriority()* 関数を使用して変更できます。
- 「*RUNNING（実行中）*」スレッドは、イベントの待機中は「*WAITING（待機中）*」状態に移行します。

- アクティブなスレッドは、*osThreadTerminate()* 関数を使用していつでも終了できます。また、通常の*永久ループ* から抜けたり、スレッド関数から*戻る* だけでもスレッドを終了することができます。終了したスレッドは［INACTIVE（非アクティブ）］状態になり、通常は動的メモリリソースを消費しません。

## シングルスレッドプログラム

標準 C プログラムは、*main()* 関数を使用して実行を開始します。組み込みアプリケーションの場合、通常、この関数は無限ループとなり、連続して実行される 1 つのスレッドと見なされます。

## プリエンプティブなスレッド切り替え

優先順位が同じスレッドで他のタスクを実行するには、ラウンドロビンタイムアウトまたは *osDelay()* 関数を明示的に呼び出す必要があります。以下の例では、*job2* の優先順位が *job1* よりも高い場合、*job2* の実行が直ちに開始されます。*job2* は *job1* の実行に対してプリエンプティブになります（このタスク切り替えは高速で行われ数ミリ秒しかかかりません）。

**ラウンドロビンタスク切り替えを使用した簡単な RTX プログラム**

```
#include "cmsis_os.h"

int counter1;
int counter2;

void job1 (void const *arg) {
  while (1) {                                // Loop forever
    counter1++;                              // Increment counter1
  }
}

void job2 (void const *arg) {
  while (1) {                                // Loop forever
    counter2++;                              // Increment counter2
  }
}

osThreadDef (job1, osPriorityNormal, 1, 0);  // Define thread for job1
osThreadDef (job2, osPriorityNormal, 1, 0);  // Define thread for job2

int main (void) {                            // main() runs as thread
  osKernelInitialize ();                     // Initialize RTX
```

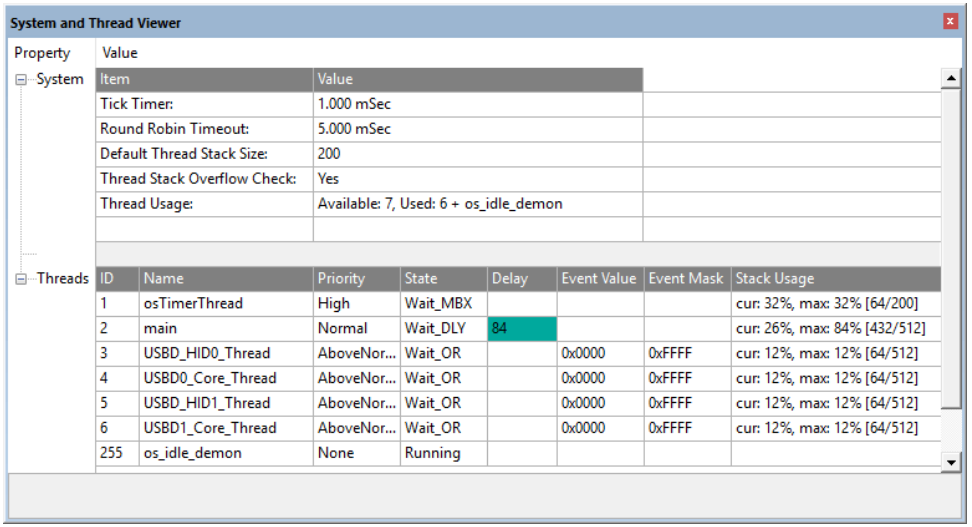
```
  osThreadCreate (osThread (job1), NULL);    // Create and start job1
  osThreadCreate (osThread (job2), NULL);    // Create and start job2

  osKernelStart ();                          // Start RTX kernel

  while (1) {
    osThreadYield ();                        // Next thread
  }
}
```

スレッドの優先順位が高い *job2* の開始

```
:
osThreadDef (osThread (job2), osPriorityAboveNormal, 1, 0);
:
```

# CMSIS-RTOS システムとスレッドビューア

CMSIS-RTOS RTX カーネルには、RTOS 認識デバッグのサポートが組み込みまれています。デバッグ中に、**[Debug（デバック）]** ・ **[OS Support（OS サポート）]** の順に開き、**[System and Thread Viewer（システムとスレッドビューア）]** を選択します。このウィンドウには、システム状態情報および実行中のスレッドが表示されます。




**[System（システム）]** プロパティには、アプリケーションの RTOS コンフィギュレーションに関する一般的な情報が表示されます。**[Thread Usage（スレッド使用状況）]** には、使用可能なスレッドと現在アクティブになっている使用済みスレッドの数が表示されます。

**[Threads（スレッド）]** プロパティには、アプリケーションのスレッド実行に関する詳細が表示されます。スレッドごとに、優先順位、実行状態、およびスタック消費量に関する情報が表示されます。

*RTX_Conf_CM.c* ファイルの［Thread Configuration（スレッドコンフィギュレーション）］に対して［Stack usage watermark（スタック消費量の透かし）］オプションを有効にすると、［Stack Usage（スタック消費量）］フィールドに［cur:］および［max:］のスタックロードが表示されます。［cur:］の値は、実際のプログラム位置における現在のスタック消費量です。［max:］の値は、スタック消費量の透かしパターンの上書きに基づく、スレッド実行中に発生した最大スタックロードです。これを使用すると、以下のことができるようになります。

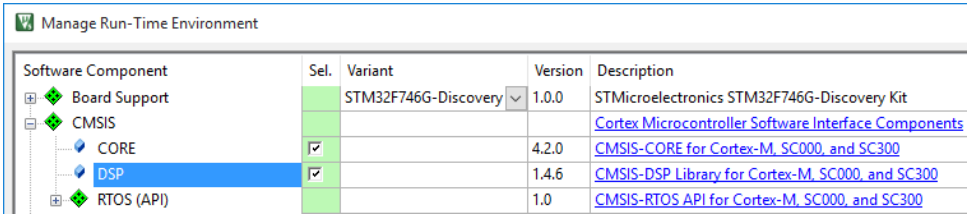
- スレッド実行中のスタックオーバーフローを識別する。
- スレッドに使用するスタック空間を最適化して縮小する。

*注*
*トレースを使用する場合、デバッガには詳しいタイミング情報を表示するビューも用意されています。詳細については、「［Event Viewer（イベントビューア）］」セクション（98ページ）を参照して下さい。*

# CMSIS-DSP

CMSIS-DSP ライブラリは、一般的なデジタル信号処理（DSP）関数のセットです。ライブラリは、さまざまな Cortex-M プロセッサ用に最適化されたいくつかのバリアントで使用できます。

ソフトウェアコンポーネント ::CMSIS:DSP を［Manage Run-Time Environment（実行時環境の管理）］ダイアログで有効にすると、選択したデバイスの最適なライブラリが自動的にプロジェクトに含められます。

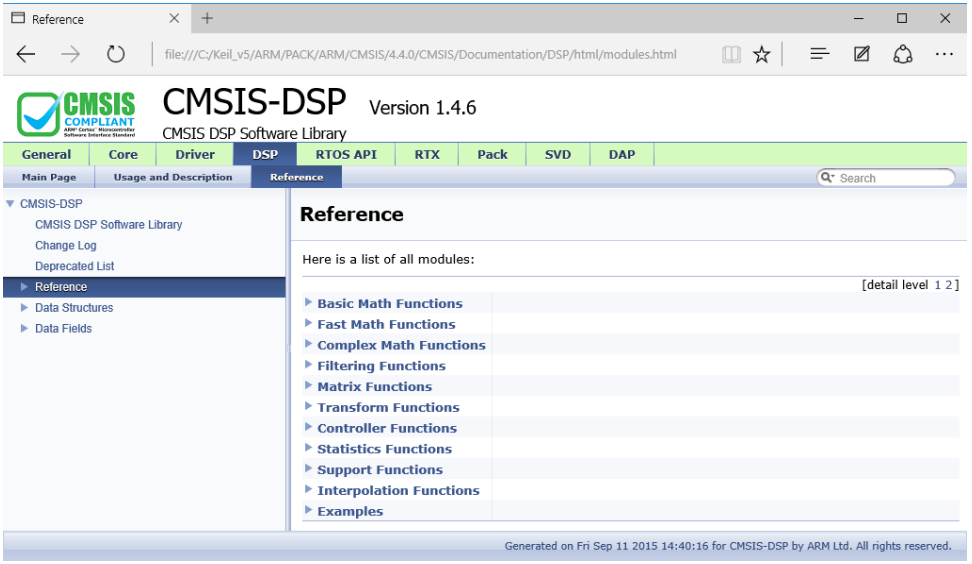


以下のコードは、CMSIS-DSP ライブラリ関数の使用例です。

**DSP 関数を使用する 2 つのマトリックスの乗算**

```
#include "arm_math.h"                 // ARM::CMSIS:DSP

const float32_t buf_A[9] = {          // Matrix A buffer and values
 1.0,  32.0,  4.0,
 1.0,  32.0,  64.0,
 1.0,  16.0,  4.0,
};

float32_t buf_AT[9];                  // Buffer for A Transpose (AT)
float32_t buf_ATmA[9] ;               // Buffer for (AT * A)

arm_matrix_instance_f32 A;            // Matrix A
arm_matrix_instance_f32 AT;           // Matrix AT(A transpose)
arm_matrix_instance_f32 ATmA;         // Matrix ATmA( AT multiplied by A)

uint32_t rows = 3;                    // Matrix rows
uint32_t cols = 3;                    // Matrix columns

int main(void) {
  // Initialize all matrixes with rows, columns, and data array
  arm_mat_init_f32 (&A, rows, cols, (float32_t *)buf_A);  // Matrix A
  arm_mat_init_f32 (&AT, rows, cols, buf_AT);             // Matrix AT
  arm_mat_init_f32 (&ATmA, rows, cols, buf_ATmA);         // Matrix ATmA

  arm_mat_trans_f32 (&A, &AT);        // Calculate A Transpose (AT)
  arm_mat_mult_f32 (&AT, &A, &ATmA); // Multiply AT with A

  while (1);
}
```

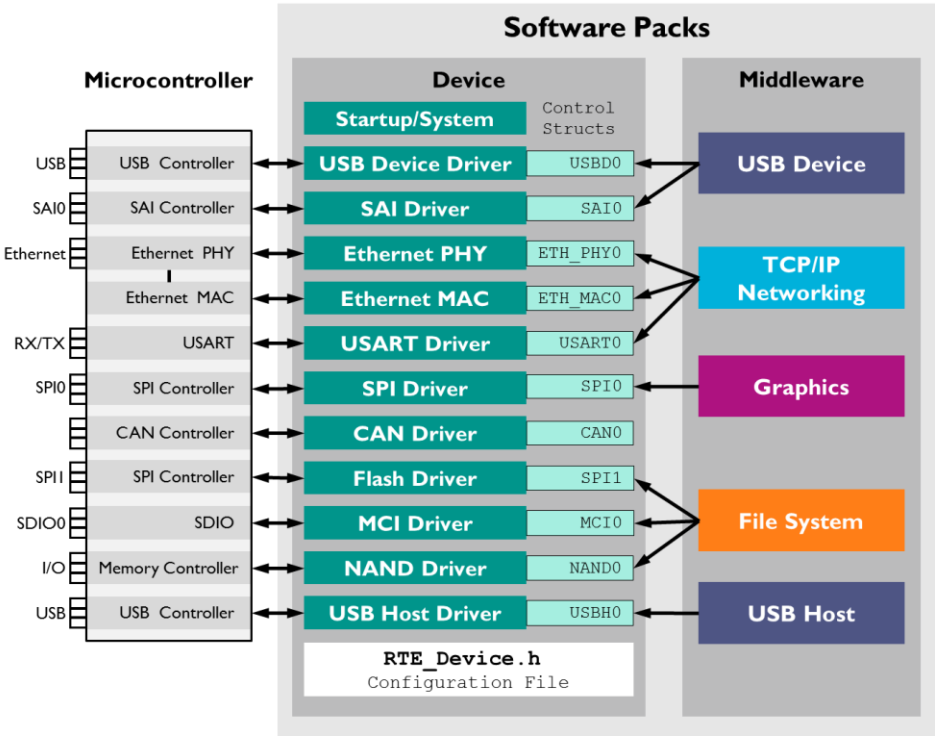
詳細については、www.keil.com/cmsis/dsp にある CMSIS-DSP のマニュアルを参照して下さい。



# CMSIS ドライバ

デバイス固有の **CMSIS ドライバ**には、ミドルウェアとマイクロコントローラペリフェラル間のインタフェースが用意されています。これらのドライバは、MDK-Professional Middleware に限らず、他のさまざまなミドルウェアスタックがペリフェラルを使用するときにも便利です。

通常、デバイス固有のドライバはソフトウェアパックの一部になっており、マイクロコントローラデバイスをサポートし、CMSIS ドライバ標準に準拠しています。www.keil.com/dd2 のデバイスデータベースには、デバイスのソフトウェアパックに含まれているドライバが一覧表示されています。




通常、ミドルウェアコンポーネントにはさまざまなコンフィギュレーションファイルが含まれており、これらのドライバに接続します。ほとんどのデバイスでは、*RTE_Device.h* ファイルによって、マイクロコントローラデバイスの実際のピン接続にドライバが設定されます。

ミドルウェア/アプリケーションコードは、*control struct* を介してドライバインスタンスに接続します。この *control struct* の名前にはデバイスのペリフェラルインタフェースが反映されます。多くの通信ペリフェラルのドライバは、デバイスサポートを提供するソフトウェアパックの一部です。

指定していないドライバを実装するには、CMSIS ドライバ標準に従って従来の C ソースコードを使用します。

CMSIS ドライバの API インタフェースの詳細については、www.keil.com/cmsis/driver を参照して下さい。

*注*

***アプリケーションノート 250「Creating a Software Pack with a New Peripheral Driver（新しいペリフェラルドライバを使用したソフトウェアパックの作成）」** では、ソフトウェアパックに存在しない新しいペリフェラルドライバを作成する方法について説明しています。*

*www.keil.com/appnotes を参照して下さい。*

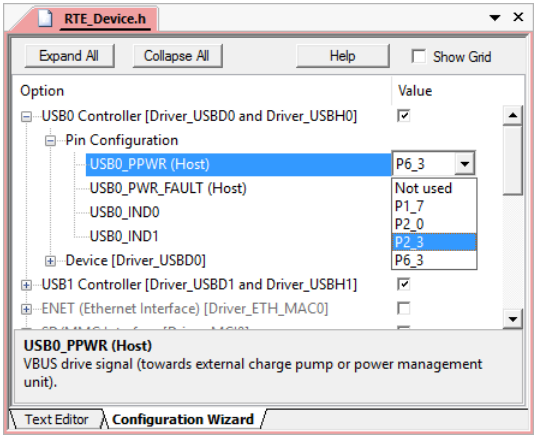
## コンフィギュレーション

CMSIS ドライバを設定する方法は、いくつかあります。標準的な方法は、デバイスサポートに付属している *RTE_Device.h* ファイルを使用することです。

その他のデバイスは、ユーザがデバイスのピン位置および対応するドライバを設定できる、サードパーティ製のグラフィカルコンフィギュレーションツールを使用して設定できます。通常、これらのコンフィギュレーションツールは、µVision プロジェクトにインポートするために必要な C コードを自動的に作成します。

## RTE_Device.h の使用

ほとんどのデバイスでは、*RTE_Device.h* ファイルによって、マイクロコントローラデバイスの実際のピン接続にドライバが設定されます。




［Configuration Wizard（コンフィギュレーションウィザード）］ビューを使用すると、ドライバインタフェースのヘッダファイルの #defines を手動で編集することなく、そのドライバインタフェースをグラフィカルモードで設定することができます。

## STM32CubeMX の使用

MDK は、STM32CubeMX を使用して CMSIS ドライバのコンフィギュレーションをサポートしています。このグラフィカルソフトウェアコンフィギュレーションツールでは、STMicroelectronics デバイス用のグラフィカルウィザードを使用して、C 初期化コードを生成できます。

必要な操作は、［Manage Run-Time Environment（実行時環境の管理）］ウィンドウで必要な CMSIS ドライバを選択し、**Device:STM32Cube Framework (API):STM32CubeMX** を選択するだけです。これにより、デバイスとドライバコンフィギュレーションに必要な STM32CubeMX が開きます。終了すると、コンフィギュレーションコードを生成して、µVision にインポートします。

詳細については、www.keil.com/pack/doc/STM32Cube/General/html/index.html のオンラインマニュアルを参照して下さい。

# 検証

CMSIS ドライバ検証テスト用のソフトウェアパックは、www.keil.com/pack から入手できます。このソフトウェアパックには、CMSIS ドライバ検証スイートのソースコードとマニュアルの他に、必要なコンフィギュレーションファイル、さまざまなターゲットプラットフォームでの使用方法を示す例が含まれています。

CMSIS ドライバ検証スイートは、以下のテストを実行します。

- API 関数呼び出しの一般的な検証
- コンフィギュレーションパラメータの検証
- ループバックとの通信テストの検証
- 通信速度などの通信パラメータの検証
- イベント関数の検証

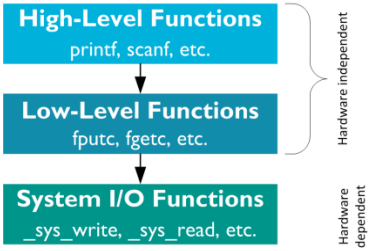
テスト結果は、コンソールに印刷したり、ITM printf を介して出力したり、メモリバッファに出力したりできます。www.keil.com/cmsis/driver で見ることができる CMSIS ドライバマニュアルの**「Driver Validation（ドライバの検証）」**セクションを参照して下さい。

# ソフトウェアコンポーネント

## Compiler（コンパイラ）

ソフトウェアコンポーネント [Compiler（コンパイラ）] を使用すると、標準 C ランタイムライブラリの I/O 関数のターゲットを変更できます。入出力操作を実行するために、アプリケーションコードでは、*printf()*、*scanf()*、または *fgetc()* などの標準的な I/O ライブラリ関数がよく使用されます。

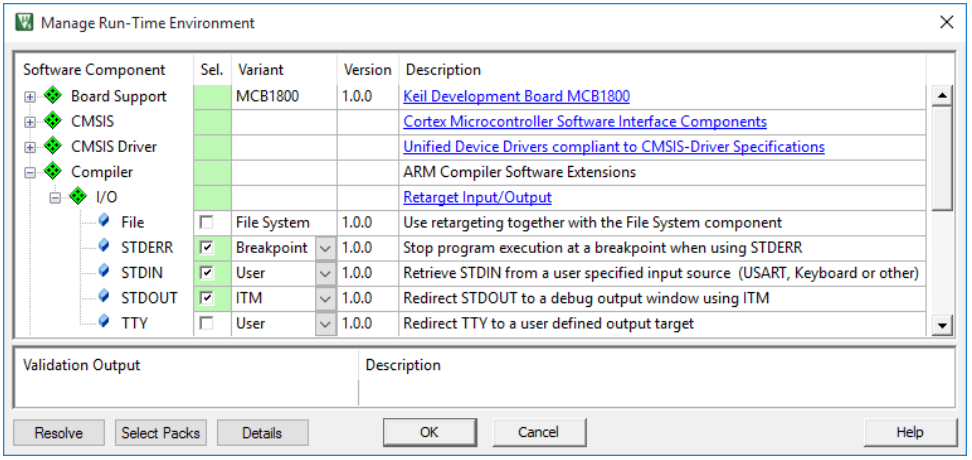
標準的な ARM コンパイラ C ランタイムライブラリ内のこれらの関数の構造は、以下のようになります。




高レベル関数および低レベル関数はターゲットに依存せず、ハードウェアとのインタフェースにシステム I/O 関数を使用します。

ARM コンパイラ C ランタイムライブラリの MicroLib は、低レベル関数を介してハードウェアとインタフェースをとります。MicroLib が実装する一連の高レベル関数は制限されているため、システム I/O 関数は実装されません。

ソフトウェアコンポーネント [Compiler（コンパイラ）] は、さまざまな標準 I/O チャネル（File、STDERR、STDIN、STDOUT、および TTY）に合わせて I/O 関数のターゲットを変更します。



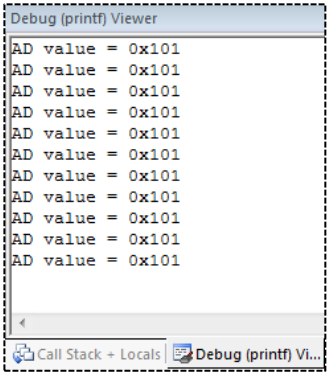

| I/O チャネル | 説明 |
|---|---|
| **File（ファイル）** | すべてのファイル関連操作チャネル（fscanf、fprintf、fopen、fclose など）。 |
| **STDERR** | 診断メッセージを出力するアプリケーションの標準エラーストリーム。 |
| **STDIN** | アプリケーションの一部になる標準入力ストリーム（scanf など）。 |
| **STDOUT** | アプリケーションの標準出力ストリーム（printf など）。 |
| **TTY** | エラー出力の最後の手段となるテレタイプ端末。 |

バリアントを選択すると、I/O チャネルのハードウェアインタフェースを変更できます。

| バリアント | 説明 |
|---|---|
| **File System（ファイルシステム）** | ファイルシステムコンポーネントは、ファイル関連操作のインタフェースとして使用します。 |
| **Breakpoint（ブレークポイント）** | I/O チャネルを使用すると、アプリケーションは BKPT 命令で停止します。 |
| **ITM** | デバッガを介した I/O 通信用のインストルメンテーショントレースマクロセル（ITM）を使用します。 |
| **User（ユーザ）** | I/O 関数をユーザ定義ルーチン（USART、キーボードなど）にターゲット変更します。 |

ソフトウェアコンポーネント **[Compiler（コンパイラ）]** は、バリアントの設定に応じて設定される *retarget_io.c* ファイルを追加します。 **[User（ユーザ）]** バリアントの場合、ユーザコードテンプレートを使用すると、独自の機能を実装する際に役立ちます。詳細については、マニュアルを参照して下さい。

Cortex-M3/M4/M7 の ITM は *printf* 形式のデバッグをサポートしています。バリアント **[ITM]** を選択すると、I/O ライブラリ関数は、 **[Debug (printf) Viewer（デバッグ（printf）ビューア）]** ウィンドウで I/O 操作を実行します。

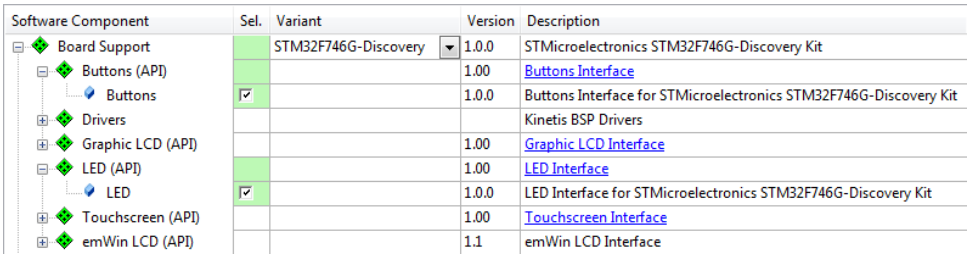



# Board Support（ボードサポート）

LED、プッシュボタン、ジョイスティック、A/D および D/A コンバータ、LCD、タッチスクリーンなどの開発ボードや、温度計、加速度計、磁力計、ジャイロスコープなどの外部センサでよく使用されるインタフェースがいくつかあります。

**ボードサポートインタフェース API** では、これらのインタフェースに対し、標準化されたアクセスを行います。これにより、ソフトウェア開発者は、特定の GPIO に切り替えるためにレジスタ設定についてデバイスのマニュアルを確認するのではなく、アプリケーションコードに集中することができます。

多くのデバイスファミリパック（DFP）にはボードサポートが含まれています。ボードサポートは［Manage Run-Time Environment（実行時環境の管理）］ウィンドウで選択できます。




特定の開発ボードに対して正しいピン設定を行うことができるように、適切な［**Variant（バリアント）**］を選択して下さい。

ボードのソフトウェアパックに必要なサポートファイルを作成すると、カスタムボードにボードサポートを追加できます。http://www.keil.com/pack/doc/mw/Board/html/index.html で見ることができる API マニュアルを参照して下さい。

# アプリケーションの作成

本章では、前の章で説明した CMSIS を使用して、プロジェクトを作成および変更するために必要な手順について説明します。

---

*注*

*このセクションのサンプルコードは、MCB1800 評価ボード（LPC1857 を搭載）に対応しています。別のスタータキットまたはボードを使用する場合は、コードとポートピンの設定を適合させて下さい。*

---

チュートリアルでは、*Blinky* プロジェクトを次の 2 つの基本設計概念で作成します。

- CMSIS-RTOS RTX を使用した RTOS 設計。
- RTOS カーネルのないベアメタルシステム用の無限ループ設計。

# CMSIS-RTOS RTX を使用した Blinky

このセクションでは、以下の手順を使用してプロジェクトの作成方法を説明します。

- **プロジェクトのセットアップ：**プロジェクトファイルを作成し、マイクロコントローラデバイスおよび関連する CMSIS コンポーネントを選択します。
- **デバイスクロック周波数の設定：**デバイスのシステムクロック周波数と CMSIS-RTOS RTX カーネルを設定します。
- **ソースコードファイルの作成：**アプリケーションファイルを追加および作成します。
- **アプリケーションイメージのビルド：**アプリケーションをコンパイルおよびリンクして、マイクロコントローラデバイスのオンチップフラッシュメモリにダウンロードします。

「**デバッガの使用**」セクション（84 ページ）では、評価ボードを PC に接続し、アプリケーションをターゲットにダウンロードする手順について説明しています。

*Blinky* プロジェクトでは、以下のアプリケーションファイルを作成します。

*main.c* このファイルには *main()* 関数が含まれており、RTOS カーネル、ペリフェラルを初期化し、スレッドの実行を開始します。

*LED.c* このファイルには、GPIO ポートと *blink_LED()* スレッド関数を初期化して制御する関数が含まれています。*LED_Initialize()* 関数は、GPIO ポートピンを初期化します。*LED_On()* 関数と *LED_Off()* 関数は、LED とのインタフェースをとるポートピンを制御します。

*LED.h* ヘッダファイルには関数の関数プロトタイプが *LED.c* に含まれており、ヘッダファイルは *main.c* ファイルに含まれています。

また、システムクロックと CMSIS-RTOS RTX を設定します。

## プロジェクトのセットアップ

µVision メニューバーで、**[Project（プロジェクト）]** - **[New µVision Project（新しい µVision プロジェクト）]** を選択します。

☞ 空のフォルダを選択し、「*Blinky*」などのプロジェクト名を入力します。**[Save（保存）]** をクリックします。指定された名前で空のプロジェクトファイル（*Blinky.uvproj*）が作成されます。

次に、**[Select Device for Target（ターゲットのデバイスを選択）]** ダイアログが開きます。

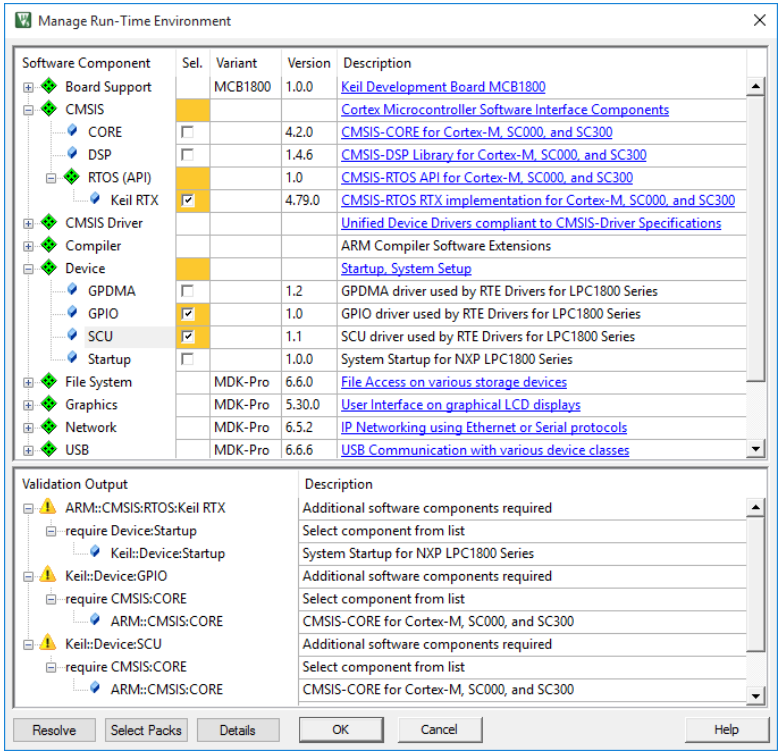
☞ LPC1857 を選択し、**[OK]** をクリックします。

デバイス選択によって、コンパイラ制御、リンカのメモリレイアウト、フラッシュプログラミングアルゴリズムなどの基本的なツール設定が定義されます。

**[Manage Run-Time Environment（実行時環境の管理）]** ダイアログが開き、インストール済みで、選択したデバイスで使用できるソフトウェアコンポーネントが表示されます。

☞ **::CMSIS:RTOS(API)** を拡張して **:Keil RTX** を有効にします。

**::Device** を拡張して **:GPIO** と **:SCU** を有効にします。

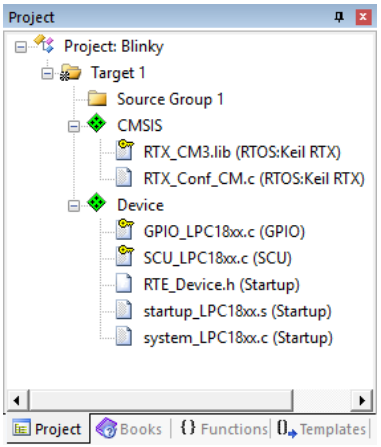



［Validation Output（検証出力）］フィールドには、他のソフトウェアコンポーネントとの依存関係が表示されます。この場合、::Device:Startup コンポーネントは必須です。

ヒント：メッセージをクリックすると、関連するソフトウェアコンポーネントがハイライトされます。

☞ [Resolve（解決）] をクリックします。

すべての依存関係が解決され、他の必須ソフトウェアコンポーネント（ここでは、::CMSIS:Core と ::Device:Startup）が有効になります。




☞ [OK] をクリックします。

選択したソフトウェアコンポーネントは、スタートアップファイル、RTX コンフィギュレーションファイル、および CMSIS システムファイルと共にプロジェクトに含められます。 [Project（プロジェクト）] ウィンドウには、選択したソフトウェアコンポーネントおよび関連するファイルが表示されます。エディタで開くには、ファイルをダブルクリックします。

## デバイスクロック周波数の設定

システムまたはコアクロックは *system_<デバイス>.c* ファイルに定義されます。コアクロックは RTOS カーネルタイマの入力クロックでもあるため、RTX コンフィギュレーションファイルはこの設定に一致する必要があります。

*注*

*一部のデバイスは、main 関数の一部としてシステムセットアップを実行したり、外部のユーティリティを使用して設定されるソフトウェアフレームワークを使用したりします。*

*詳細については、「**デバイススタートアップの例**」セクション（74 ページ）を参照して下さい。*

アプリケーションのクロックコンフィギュレーションは、クロックソース（XTAL またはオンチップオシレータ）や、メモリおよびペリフェラルの要件などのさまざまな要因に応じて異なります。シリコンベンダはデバイス固有のファイル *system_<デバイス>.c* を用意しているため、関連するドキュメントを読む必要があります。

ヒント：マイクロコントローラクロックシステムの詳細については、 [Books（ブック）] ウィンドウのリファレンスマニュアルを開いて下さい。

MCB1800 開発キットは、外部の 12 MHz XTAL を使用して実行されます。PLL は 180 MHz のコアクロック周波数を生成します。これはデフォルトであるため、変更する*必要はありません*。ただし、*system_LPC18xx.c* ファイルのカスタム開発ボードの設定は変更できます。

☞ *system_LPC18xx.c* ファイルを編集するには、 [Device（デバイス）] グループを [Project（プロジェクト）] ウィンドウで拡張します。ファイル名をダブルクリックして、以下のコードを変更します。

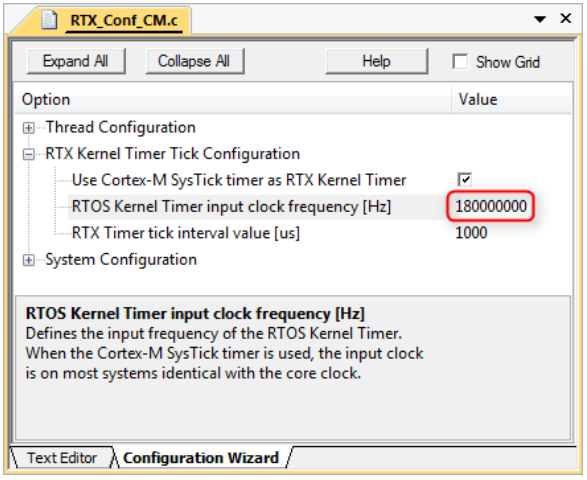
*system_LPC18xx.c* の PLL パラメータの設定

```
:
/* PLL1 output clock: 180MHz, Fcco: 180MHz, N = 1, M = 15, P = x        */
#define PLL1_NSEL   0           /* Range [0 -   3]: Pre-divider  ratio N */
#define PLL1_MSEL  14           /* Range [0 - 255]: Feedback-div ratio M */
#define PLL1_PSEL   0           /* Range [0 -   3]: Post-divider ratio P */

#define PLL1_BYPASS 0           /* 0: Use PLL, 1: PLL is bypassed        */
#define PLL1_DIRECT 1           /* 0: Use PSEL, 1: Don't use PSEL        */
#define PLL1_FBSEL  0           /* 0: FCCO is used as PLL feedback       */
                                /* 1: FCLKOUT is used as PLL feedback    */
:
```

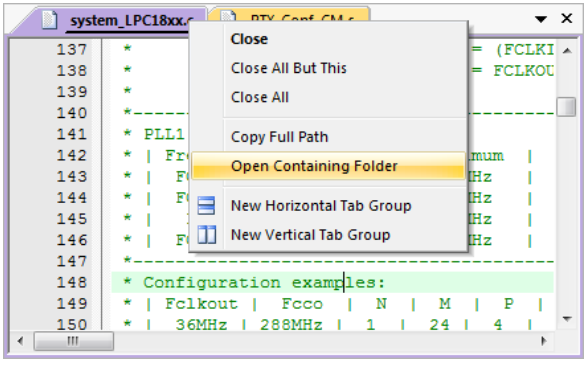
## CMSIS-RTOS RTX カーネルのカスタマイズ

☞ [Project（プロジェクト）] ウィンドウで、[CMSIS] グループを拡張し、*RTX_Conf_CM.c* ファイルを開きます。次に、エディタの下部にある [Configuration Wizard（コンフィギュレーションウィザード）] タブをクリックします。

[RTX Kernel Timer Tick Configuration（RTX カーネルタイマ Tick コンフィギュレーション）] を拡張し、[Timer clock value（タイマクロック値）] をコアクロックに一致させます。



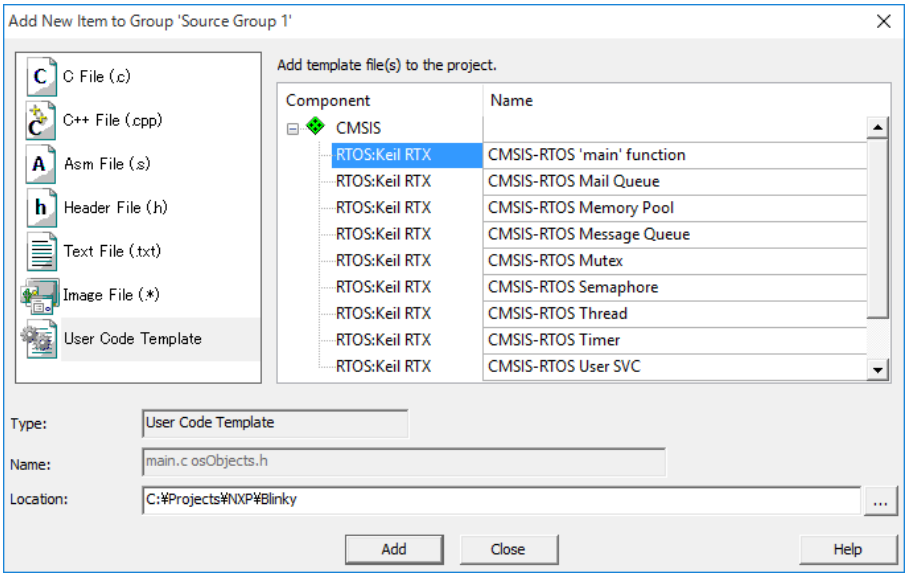

ヒント：コンパイラ定義の設定と *system_<デバイス>.c* をサンプルプロジェクトからコピーできます。エディタでファイル名を右クリックし、[Open Containing Folder（含んでいるフォルダを開く）] を選択してファイルを検索します。



## ソースコードファイルの作成

ソフトウェアコンポーネントの機能に似たルーチンを含んでいる、事前設定された**ユーザコードテンプレート**を使用して、アプリケーションコードを追加します。

☞ **[Project（プロジェクト）]** ウィンドウで、**[Source Group 1（ソースグループ 1）]** を右クリックして、**[Add New Item to Group（新規項目をグループに追加）]** ダイアログを開きます。




☞ **[User Code Template（ユーザコードテンプレート）]** をクリックして、プロジェクトに含まれているソフトウェアコンポーネントに使用できるコードテンプレートを表示します。**[CMSIS-RTOS 'main' function（CMSIS-RTOS 'main' 関数）]** を選択して **[Add（追加）]** をクリックします。

*main.c* ファイルがプロジェクトグループ **[Source Group 1（ソースグループ 1）]** に追加されます。これで、アプリケーション固有のコードをこのファイルに追加できます。

☞ *main.c* ファイル内の空白行を右クリックして、[Insert '#include files' ('#include files' を挿入)] を選択します。選択したデバイスのヘッダファイル *LPC18xx.h* を含めます。

次に、下のコードを追加して *blink_LED()* 関数を作成します。評価キットの LED が点滅します。*osThreadDef()* を使用して *blink_LED()* を RTOS スレッドとして定義し、*osThreadCreate()* で開始します。

***main.c* のコード**

```
/*----------------------------------------------------------------------
 * CMSIS-RTOS 'main' function template
 *---------------------------------------------------------------------*/

#define osObjectsPublic           // Define objects in main module
#include "osObjects.h"            // RTOS object definitions
#include "LPC18xx.h"              // Device header
#include "LED.h"                  // Initialize and set GPIO Port

/*
 * main: initialize and start the system
 */
int main (void) {
  osKernelInitialize ();          // Initialize CMSIS-RTOS
  // initialize peripherals here
  LED_Initialize ();              // Initialize LEDs

  // create 'thread' functions that start executing,
  // example: tid_name = osThreadCreate (osThread(name), NULL);
  Init_BlinkyThread ();           // Start Blinky thread
  osKernelStart ();               // Start thread execution

  while (1);
}
```

☞ [Add New Item to Group（新規項目をグループに追加）] ダイアログを使用して *LED.c* という名前の空の C ファイルを作成します。コードを追加して初期化し、LED を制御する GPIO ポートピンにアクセスします。

*LED.c* のコード

```
/*----------------------------------------------------------------------
 * File LED.c
 *----------------------------------------------------------------------*/
#include "SCU_LPC18xx.h"
#include "GPIO_LPC18xx.h"
#include "cmsis_os.h"                   // ARM::CMSIS:RTOS:Keil RTX

void blink_LED (void const *argument);  // Prototype function

osThreadDef (blink_LED, osPriorityNormal, 1, 0); // Define blinky thread

void LED_Initialize (void) {
  GPIO_PortClock     (1);               // Enable GPIO clock

  /* Configure pin: Output Mode with Pull-down resistors */
  SCU_PinConfigure (13, 10, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 24, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 24, 0);
}

void LED_On (void) {
  GPIO_PinWrite     (6, 24, 1);         // LED on: set port
}

void LED_Off (void) {
  GPIO_PinWrite     (6, 24, 0);         // LED off: clear port
}

// Blink LED function
void blink_LED(void const *argument) {
  for (;;) {
    LED_On ();                          // Switch LED on
    osDelay (500);                      // Delay 500 ms
    LED_Off ();                         // Switch off
    osDelay (500);                      // Delay 500 ms
  }
}

void Init_BlinkyThread (void) {
  osThreadCreate (osThread(blink_LED), NULL);    // Create thread
}
```

*注*
*Board Support（ボードサポート） コンポーネント（59 ページに説明）が指定する関数を使用することもできます。*

☞ [Add New Item to Group（新規項目をグループに追加）] ダイアログを使用して *LED.h* という名前の空のヘッダファイルを作成し、*LED.c* の関数プロトタイプを定義します。

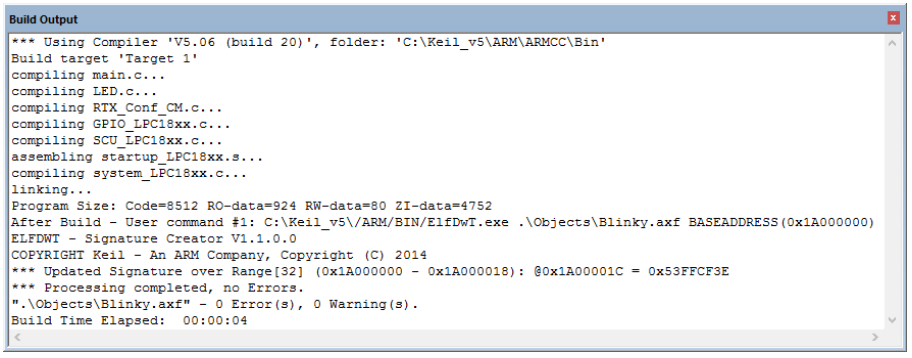
*LED.h* のコード

```
/*----------------------------------------------------------------------
 * File LED.h
 *----------------------------------------------------------------------*/
void LED_Initialize ( void );                 // Initialize GPIO
void LED_On ( void );                         // Switch Pin on
void LED_Off ( void );                        // Switch Pin off

void blink_LED ( void const *argument );      // Blink LEDs in a thread
void Init_BlinkyThread ( void );              // Initialize thread
```

## アプリケーションイメージのビルド

アプリケーションをビルドします。すべての関連するソースファイルのコンパイルおよびリンクが行われます。

[Build Output（ビルド出力）] に、ビルドプロセスに関する情報が表示されます。エラーのないビルドについては、プログラムサイズ情報、エラーゼロ、および警告ゼロが表示されます。




「**デバッガの使用**」セクション（84 ページ）では、評価ボードをワークス

テーションに接続し、アプリケーションをターゲットハードウェアにダウンロードする方法について説明しています。

ヒント：LED を 1 秒間隔でチェックすることで、ターゲットハードウェアの正しいクロックと RTOS コンフィギュレーションを検証できます。

# 無限ループ設計を使用した Blinky

前の例に基づき、無限ループ設計を使用し、CMSIS-RTOS RTX 関数を使用せずに、Blinky アプリケーションを作成します。このプロジェクトには、以下のユーザコードファイルが含まれています。

*main.c* このファイルには、*main()* 関数、システム Tick タイマを初期化する *Systick_Init()* 関数、およびそのハンドラ関数 *SysTick_Handler()* が含まれています。*Delay()* 関数は一定時間待機します。

*LED.c* ファイルには、GPIO ポートピンを初期化し、ポートピンをオンまたはオフに設定する関数が含まれています。*LED_Initialize()* 関数は GPIO ポートピンを初期化します。*LED_On()* 関数と *LED_Off()* 関数は、ポートピンを有効または無効にします。

*LED.h* ヘッダファイルには *LED.c* で作成された関数プロトタイプが含まれており、*main.c* ファイルに含める必要があります。

[Manage Run-Time Environment (実行時環境の管理) ] を開き、ソフトウェアコンポーネント ::CMSIS:RTOS (API):Keil RTX の選択を解除します。

☞ *main.c* ファイルを開きます。コードを追加してシステム Tick タイマを初期化し、システム Tick タイマ割り込みハンドラおよび delay 関数を書き込みます。

```
/*----------------------------------------------------------------------
 * file main.c
 *----------------------------------------------------------------------*/

#include "LPC18xx.h"                    // Device header
#include "LED.h"                        // Initialize and set GPIO Port

int32_t volatile msTicks = 0;           // Interval counter in ms

// Set the SysTick interrupt interval to 1ms
void SysTick_Init (void) {
  if (SysTick_Config (SystemCoreClock / 1000)) {
    // handle error
  }
}

// SysTick Interrupt Handler function called automatically
void SysTick_Handler (void) {
  msTicks++;                            // Increment counter
```

```
}

// Wait until msTick reaches 0
void Delay (void) {
  while (msTicks < 499);             // Wait 500ms
  msTicks = 0;                       // Reset counter
}

int main (void) {
  // initialize peripherals here
  LED_Initialize ();                 // Initialize LEDs
  SystemCoreClockUpdate();           // Update SystemCoreClock to 180 MHz
  SysTick_Init ();                   // Initialize SysTick Timer

  while (1) {
    LED_On ();                       // Switch on
    Delay ();                        // Delay
    LED_Off ();                      // Switch off
    Delay ();                        // Delay
  }
}
```

☞ *LED.c* ファイルを開き、不要な関数を削除します。コードは以下のようになります。

```
/*----------------------------------------------------------------------
 * File LED.c
 *----------------------------------------------------------------------*/
#include "SCU_LPC18xx.h"
#include "GPIO_LPC18xx.h"

void LED_Initialize (void) {

GPIO_PortClock     (1);                    // Enable GPIO clock

/* Configure pin: Output Mode with Pull-down resistors */
SCU_PinConfigure (13, 10, (SCU_CFG_MODE_FUNC4 | SCU_PIN_CFG_PULLDOWN_EN));
GPIO_SetDir      (6, 24, GPIO_DIR_OUTPUT);
GPIO_PinWrite    (6, 24, 0);
}

void LED_On (void) {
GPIO_PinWrite     (6, 24, 1);              // LED on: set port
}

void LED_Off (void) {
GPIO_PinWrite     (6, 24, 0);              // LED off: clear port
}
```

*LED.h* ファイルを開き、コードを修正します。

```
/*-----------------------------------------------------------------------
 *  file: LED.h
 *-----------------------------------------------------------------------*/
void LED_Initialize (void);                 // Initialize LED Port Pins
void LED_On (void);                         // Set LED on
void LED_Off (void);                        // Set LED off
```

### アプリケーションイメージのビルド

アプリケーションをビルドします。すべての関連するソースファイルのコンパイルおよびリンクが行われます。

「**デバッガの使用**」セクション（84 ページ）では、評価ボードを PC に接続し、アプリケーションをターゲット ハードウェアにダウンロードする方法について説明しています。

ヒント：LED を 1 秒間隔でチェックすることで、ターゲットハードウェアの正しいクロックコンフィギュレーションを検証できます。

# デバイススタートアップの例

一部のデバイスは、デバイスハードウェア抽象化レイヤ（HAL）の一部としてシステムセットアップにおいて重要な役割を果たしているため、デバイスの初期化は main 関数内から実行されます。このようなデバイスでは、外部ユーティリティで構成されたソフトウェアフレームワークがよく使用されます。

そのため、**::Device** ソフトウェアコンポーネントには、デバイスを起動するために必要な追加のコンポーネントが含まれる可能性があります。詳細については、オンラインヘルプシステムを参照して下さい。以下のセクションでは、デバイス起動の例をご紹介します。

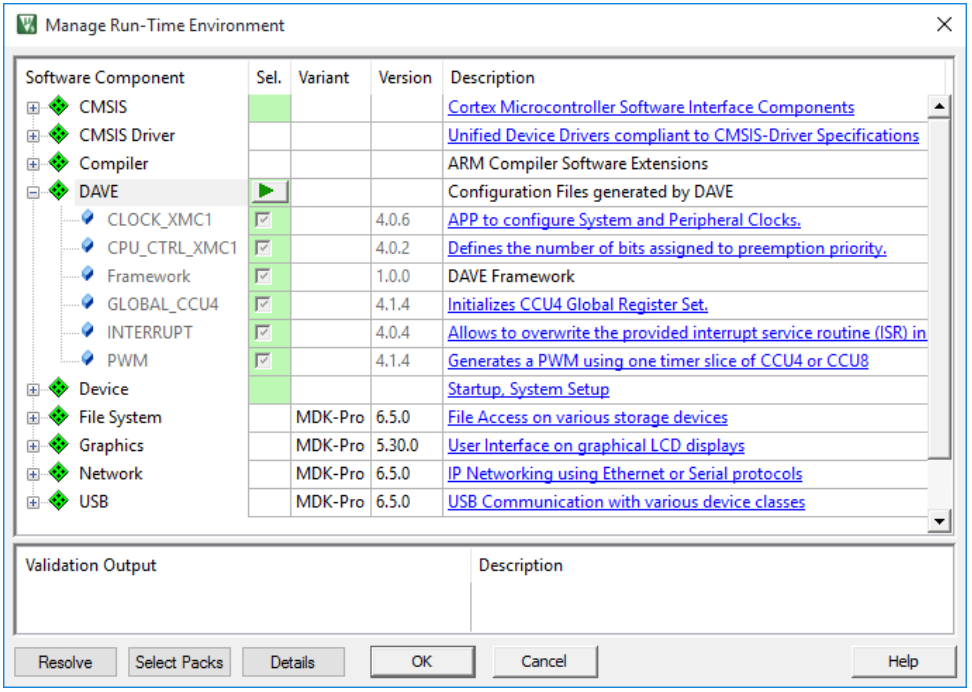
## 例：DAVE を使用する Infineon XMC1000

Infineon の **DAVE™** を使用すると、いわゆる DAVE アプリケーションに基づくコードを自動的に生成できます。Eclipse ベースの IDE 内で、使用アプリケーションに合うように、このアプリケーションを追加、設定、および接続することができます。このプロセス中に、**CLOCK_XMC_1_0** アプリケーショ

ン（XMC1000 ファミリの場合）を使用してクロック設定を指定します。このアプリケーションは、コア内で正しいレジスタを設定して、目的の周波数にします。コード生成の終わりに、CMSIS 関数 *SystemCoreClockUpdate()* を呼び出します。

DAVE プロジェクトを µVision にインポートするためのすべての手順については、http://www.keil.com/appnotes/docs/apnt_258.asp にある「アプリケーションノート 258」を参照して下さい。

µVision がプロジェクトのインポートを完了すると、**[Manage Run-Time Environment（実行時環境の管理）]** ウィンドウに、**::DAVE3** グループとコンポーネントとして生成されたアプリケーションが表示されます。




µVision 内部で、コンポーネント **::DAVE** はロックされます。アプリケーションのコンフィギュレーションを変更するには、**開始ボタン**  を使用して DAVE を開きます。

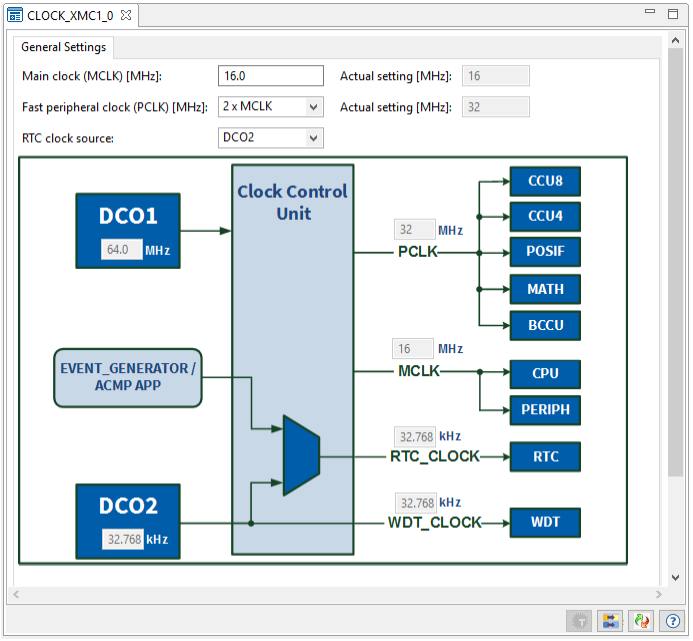
*clock_xmc1_conf.c* ファイルには、クロックレジスタを設定するためのデータ構造が含まれています。以下の例は、DAVE がツール内からコンフィギュレーションに従って値を設定する方法を示しています。

*clock_xmc1_conf.c* のコード

```
:
/******************************************************************************* DATA STRUCTU
RES
*******************************************************************************/
const XMC_SCU_CLOCK_CONFIG_t CLOCK_XMC1_0_CONFIG =
{
.pclk_src = XMC_SCU_CLOCK_PCLKSRC_DOUBLE_MCLK,
.rtc_src = XMC_SCU_CLOCK_RTCCLKSRC_DCO2,
.fdiv = 0U,  /**< Fractional divider */
.idiv = 1U,  /**< 8Bit integer divider */
};
```

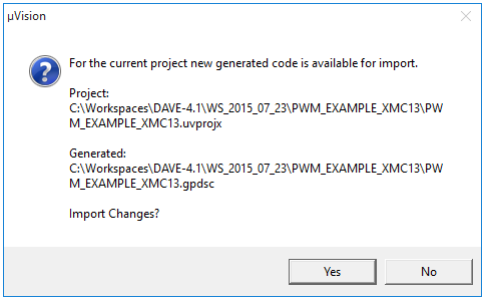
## DAVE を使用したクロックセットアップの変更

クロック値を変更する必要がある場合は、[Manage Run-Time Environment（実行時環境の管理）] ウィンドウを開き、**開始ボタン**  を押して DAVE を開きます。[Configure APP Instance（アプリケーションインスタンスの設定）…] を使用して、クロック設定を変更します。




DAVE でコード生成  を再実行します。

生成されたファイルが変更され、µVision により自動的に認識されます。




[Yes（はい）] をクリックして、変更されたファイルをリロードします。

# サンプル：STM32Cube

多くの STM32 デバイスは **STM32Cube Framework** を使用しており、*RTE_Device.h* コンフィギュレーションファイルを使用する従来の方法で設定したり、**STM32CubeMX** を使用して設定したりできます。

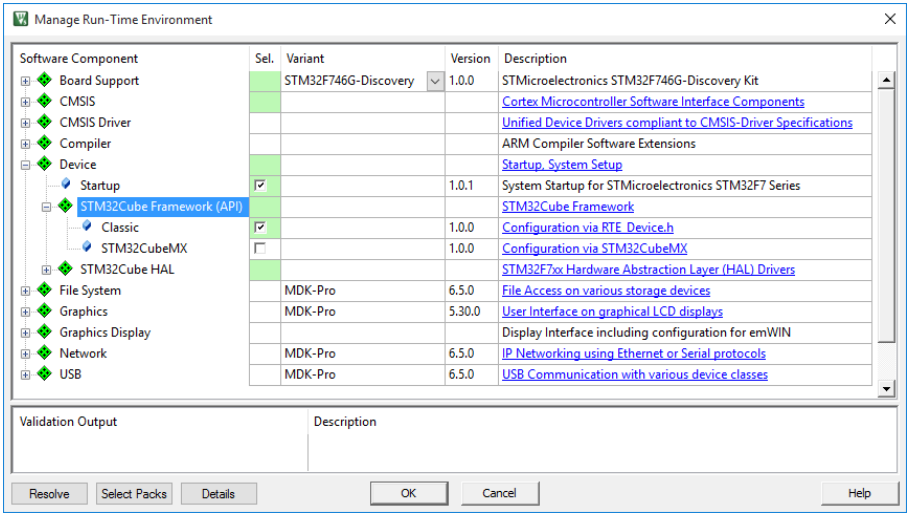
従来の **STM32Cube Framework** コンポーネントには固有のユーザコードテンプレートがあり、これを使用してシステムセットアップを実装します。**STM32CubeMX** を使用すると、起動に必要な *main.c* ファイルおよび他のソースファイルが、**STM32CubeMX:Common Sources** グループの下にあるプロジェクトにコピーされます。

## 従来のフレームワークを使用したプロジェクトのセットアップ

この例では、従来の方法を使用して STM32F746G-Discovery キットのプロジェクトを作成します。**[Manage Run-Time Environment（実行時環境の管理）]** ウィンドウで、以下のように選択します。

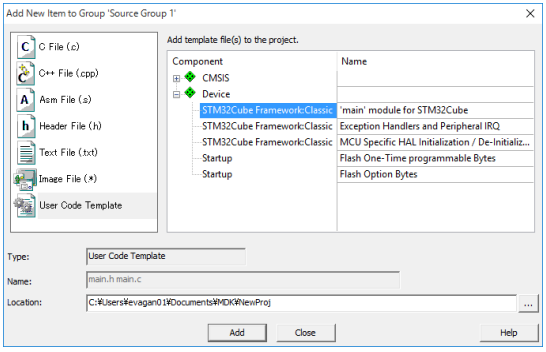
☞ **::Device:STM32Cube Framework (API)** を拡張し、**:Classic** を有効にします。

**::Device** を拡張し、**:Startup** を有効にします。




☞ **[Resolve（解決）]** をクリックして他の必要なソフトウェアコンポーネントを有効にしてから、**[OK]** をクリックします。

☞ [Project（プロジェクト）] ウィンドウで、[Source Group 1（ソースグループ 1）] を右クリックして、[Add New Item to Group（新規項目をグループに追加）] ダイアログを開きます。




☞ [User Code Template（ユーザコードテンプレート）] をクリックして、プロジェクトに含まれているソフトウェアコンポーネントに使用できるコードテンプレートを表示します。[‘main’ module for STM32Cube（STM32Cube の ‘main’ モジュール）] を選択して [Add（追加）] をクリックします。

*main.c* ファイルには *SystemClock_Config()* 関数が含まれています。ここでは、クロックセットアップを以下のように設定する必要があります。

***main.c* のコード**

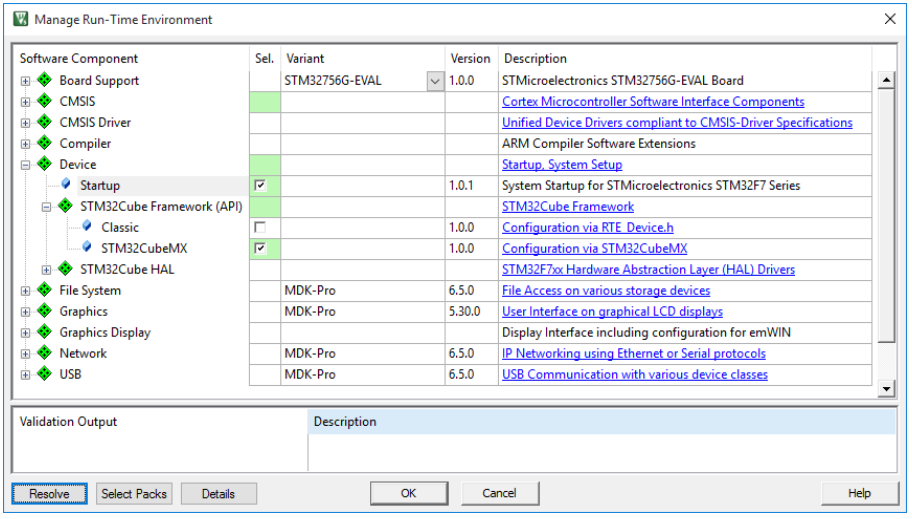
```
:
static void SystemClock_Config (void) {
RCC_ClkInitTypeDef RCC_ClkInitStruct;
RCC_OscInitTypeDef RCC_OscInitStruct;

/* Enable HSE Oscillator and activate PLL with HSE as source */
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
RCC_OscInitStruct.HSEState = RCC_HSE_ON;
RCC_OscInitStruct.HSIState = RCC_HSI_OFF;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
RCC_OscInitStruct.PLL.PLLM = 25;
RCC_OscInitStruct.PLL.PLLN = 432;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 9;
HAL_RCC_OscConfig(&RCC_OscInitStruct);

/* Activate the OverDrive to reach the 216 MHz Frequency */
HAL_PWREx_EnableOverDrive();
```

```
/* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2 clocks divid
ers */
RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_
PCLK1 | RCC_CLOCKTYPE_PCLK2);
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7);
}
:
```

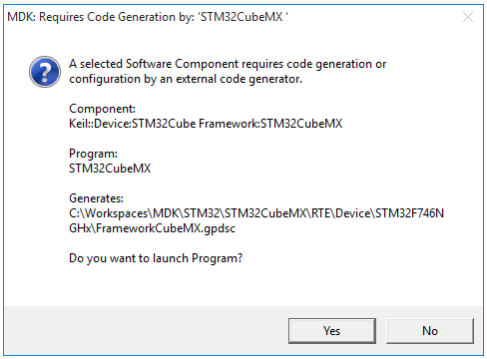
## STM32CubeMX を使用したプロジェクトのセットアップ

この例では、STM32CubeMX を使用して前回と同じプロジェクトを作成します。［Manage Run-Time Environment（実行時環境の管理）］ウィンドウで、以下のように選択します。

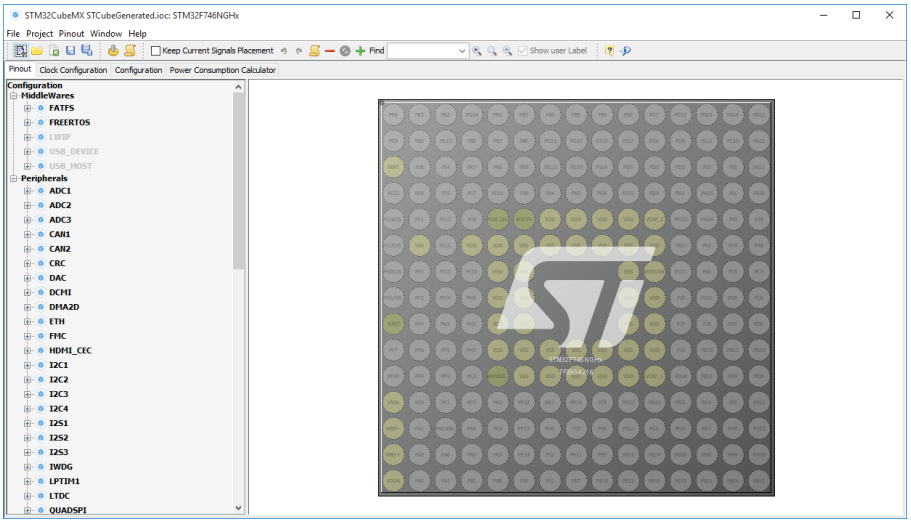
☞ ::Device:STM32Cube Framework (API) を拡張し、:STM32CubeMX を有効にします。::Device を拡張し、:Startup を有効にします。



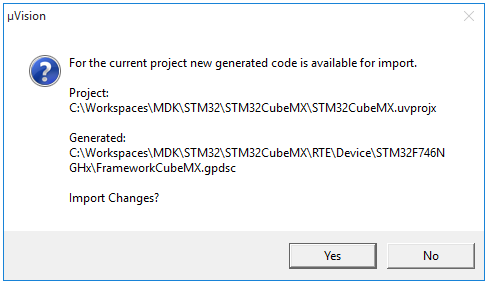
［Resolve（解決）］をクリックして他の必要なソフトウェアコンポーネントを有効にしてから、［OK］をクリックします。STM32CubeMX を起動するよう求める新しいウィンドウが表示されます。




STM32CubeMX が起動し、適切なデバイスが選択されました。



☞ 必要に応じてデバイスを設定します。設定が終了したら、**[Project（プロジェクト）]** ・ **[Generate Code（コードの生成）]** の順に移動して、GPDSC ファイルを作成します。µVision では以下のウィンドウが表示されます。




☞ ［Yes（はい）］をクリックして、プロジェクトをインポートします。**main.c** ファイルおよび他の生成ファイルが **STM32CubeMX:Common Sources** という名前のフォルダに追加されます。

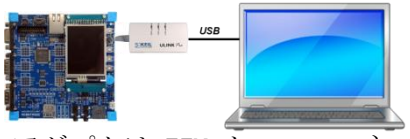
# デバッグアプリケーション

ARM CoreSight テクノロジは、ARM Cortex-M プロセッサベースのデバイスに統合され、強力なデバッグおよびトレース機能を提供します。これによって実行制御が可能になり、プログラム、ブレークポイント、メモリアクセス、およびフラッシュプログラミングを開始したり停止したりできます。サンプリング、データトレース、プログラムカウンタ（PC）割り込みを含む例外、インストルメンテーショントレースなどの機能をほとんどのデバイスで使用できます。デバイスは、ETM、ETB、または MTB を使用して命令トレースを統合し、プログラム実行の解析を行います。デバッグおよびトレース機能の詳しい概要については、www.keil.com/coresight を参照して下さい。

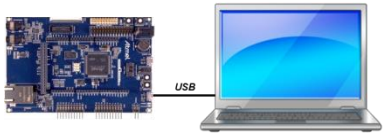
# デバッガ接続

MDK には、さまざまなデバッグ/トレースアダプタに接続する µVision デバッガが含まれており、フラッシュメモリをプログラミングすることができます。単純ブレークポイント、複合ブレークポイント、ウォッチポイント、実行制御などの従来の機能をサポートしています。トレースを使用すると、イベント/例外ビューア、ロジックアナライザ、実行プロファイラ、コードカバレッジなどの追加の機能がサポートされます。

- ULINK2 および ULINK-ME デバッグアダプタは、JTAG/SWD デバッグコネクタとインタフェースをとり、シリアルワイヤ出力（SWO）でトレースをサポートします。また、ULINK*pro* デバッグ/トレースアダプタは ETM トレースコネクタともインタフェースをとり、ストリーミングトレーステクノロジを使用して、コードカバレッジと実行プロファイリングに必要なすべての命令トレースをキャプチャします。詳細については、www.keil.com/ulink を参照して下さい。




- CMSIS-DAP ベースの USB JTAG/SWD デバッグインタフェースは、一般に評価ボードまたはスタータキットの一部であり、統合されたデバッグ機能を備えています。MDK では、類似したテクノロジを備えた独自のインタフェースを他にもいくつかサポートしています。



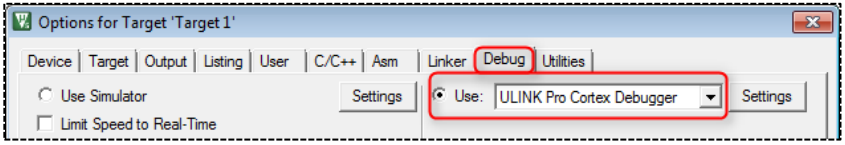
- MDK は、Segger J-Link や J-Trace などのサードパーティ製デバッグソリューションに接続します。一部のスタータキットボードには、オンボードソリューションと同様の J-Link Lite テクノロジがあります。

# デバッガの使用

次に、ハードウェアに関する前の章で作成した *Blinky* アプリケーションをデバッグします。デバッグ接続とフラッシュプログラミングユーティリティを設定する必要があります。

デバッグアダプタを選択し、デバッグオプションを設定します。

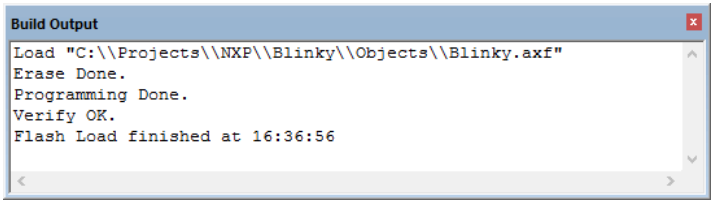
ツールバーで［Options for Target（ターゲットのオプション）］を選択し、［Debug（デバッグ）］タブをクリックします。［Use（使用）］を有効にして適切なデバッグドライバを選択します。



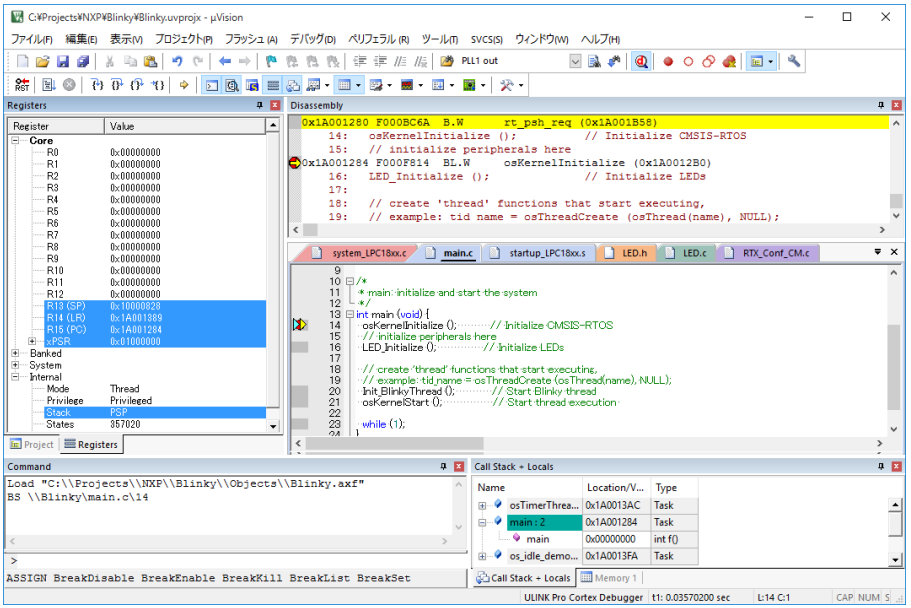

デバイス選択では、オンチップメモリのフラッシュプログラミングアルゴリズムが既に設定されています。［Settings（設定）］ボタンを使用してコンフィギュレーションを確認します。

アプリケーションをフラッシュメモリにプログラミングします。

ツールバーで［Download（ダウンロード）］を選択します。［Build Output（ビルド出力）］ウィンドウに、ダウンロードの進行状況に関するメッセージが表示されます。



ハードウェアでデバッグを開始します。ツールバーで［Start/Stop Debug Session（デバッグセッションの開始/停止）］を選択します。




デバッグセッションの開始中に、µVision によりアプリケーションがロードされ、起動コードが実行されて、main C 関数で停止します。

ツールバーの［Run（実行）］をクリックします。LED が 1 秒間隔で点滅します。

## デバッグツールバー

デバッグツールバーでは、以下のデバッグコマンドにすぐにアクセスできます。

［Step（ステップ）］は、プログラムをステップスルーして、関数呼び出しにステップインします。

［Step Over（ステップオーバー）］は、プログラムをステップスルーして、関数呼び出しをステップオーバーします。

[Step Out（ステップアウト）] は、現在の関数からステップアウトします。

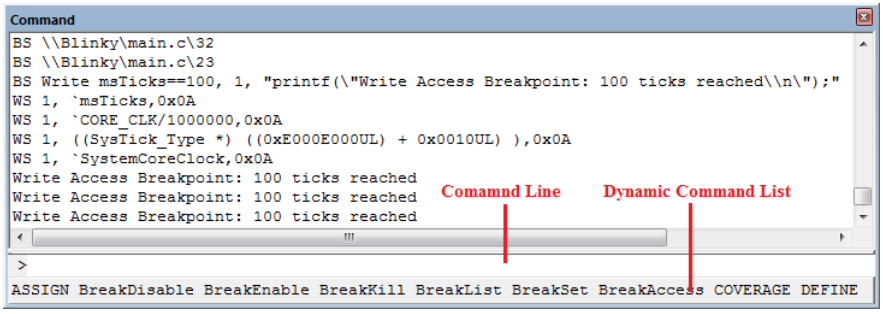
[Stop（停止）] は、プログラムの実行を停止します。

[Reset（リセット）] は、CPU リセットを実行します。

[Show（表示）] は、次（現在の PC の場所）のステートメントを実行するステートメントを示します。

## [Command（コマンド）] ウィンドウ

[Command（コマンド）] ウィンドウにデバッグコマンドを入力することもできます。

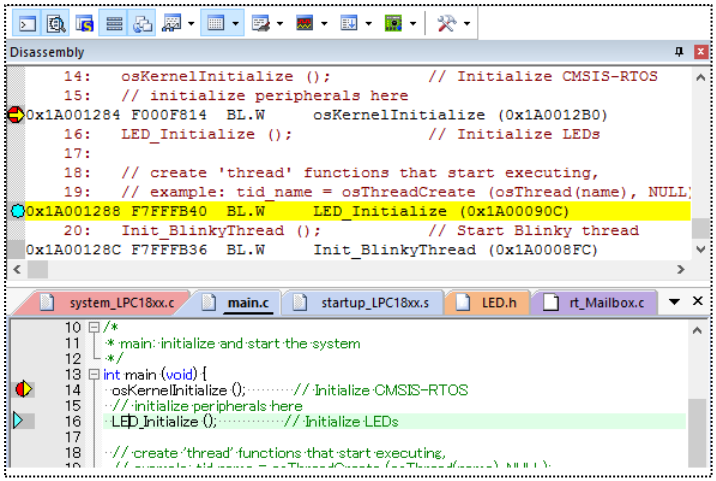



[Command Line（コマンドライン）] にデバッグコマンドを入力するか、F1 キーを押すと、詳細なヘルプ情報にアクセスできます。

# [Disassembly（逆アセンブリ）] ウィンドウ

[Disassembly（逆アセンブリ）] ウィンドウには、プログラムの実行がアセンブリコードとソースコード（使用可能な場合）が混在して表示されます。このウィンドウがアクティブになっている場合、すべてのデバッグステップ実行コマンドはアセンブリレベルで動作します。

ウィンドウの余白には、ブレークポイントやブックマーク用のマーカ、および次の実行ステートメント用のマーカが表示されます。



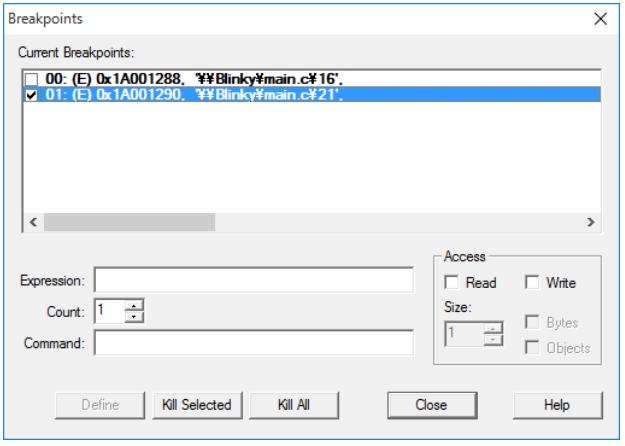
# ブレークポイント

以下のように、ブレークポイントを設定できます。

- プログラムソースコードを作成または編集しているとき。エディタのグレーの余白または [Disassembly（逆アセンブリ）] ウィンドウをクリックして、ブレークポイントを設定します。
- ツールバーのブレークポイントボタンを使用する。
- [Debug（デバッグ）] - [Breakpoints（ブレークポイント）] メニューを使用する。
- [Command（コマンド）] ウィンドウにコマンドを入力する。
- [Disassembly（逆アセンブリ）] ウィンドウのコンテキストメニューまたはエディタを使用する。

## [Breakpoints（ブレークポイント）] ウィンドウ

[Breakpoints（ブレークポイント）] ウィンドウを使用して精密なブレークポイントを定義できます。




[Debug（デバッグ）] メニューから [Breakpoints（ブレークポイント）] ウィンドウを開きます。

[Current Breakpoints（現在のブレークポイント）] フィールドのチェックボックスを使用して、ブレークポイントを有効または無効にします。定義を変更するには、既存のブレークポイントをダブルクリックします。

新しいブレークポイントを追加するには、[Expression（式）] を入力します。式に応じて、以下のブレークポイントタイプのいずれかが定義されます。

- **実行ブレークポイント（E）**：式でコードアドレスが指定されたときに作成され、このコードアドレスに到達したときに呼び出されます。

- **アクセスブレークポイント（A）**：式でメモリアクセス（読み出し、書き込み、またはその両方）が指定されたときに作成され、このメモリアドレスにアクセスしたときに呼び出されます。比較（==）演算子を使用して、指定した値を比較できます。

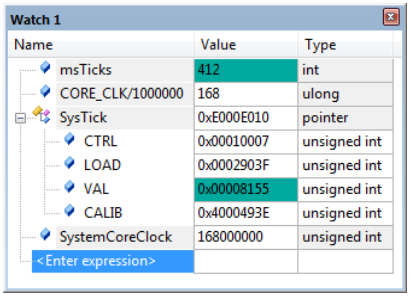
ブレークポイントに［Command（コマンド）］を指定すると、µVision によってそのコマンドが実行され、ターゲットプログラムの実行が再開します。

［Count（カウント）］の値は、ブレークポイントがプログラムの実行を停止する前に、ブレークポイント式が true となる回数を指定します。

## ［Watch（ウォッチ）］ウィンドウ

［Watch（ウォッチ）］ウィンドウでは、プログラムシンボル、レジスタ、メモリ領域、および式を確認できます。

ツールバーまたはメニューで［View（表示）］ - ［Watch Windows（ウォッチウィンドウ）］の順に選択して、［Watch（ウォッチ）］ウィンドウを開きます。




［Watch（ウォッチ）］ウィンドウに変数を追加します。

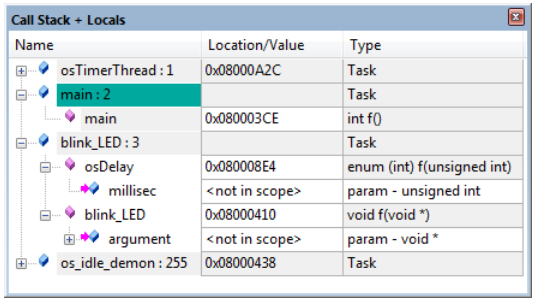
- ［<Enter expression>（<式の入力>）］フィールドをクリックして式をダブルクリックするか、F2 キーを押します。
- エディタで、カーソルが変数の上に置かれている場合は、コンテキストメニューを使用して［Add <*item name*> to…（<*項目名*> を追加...）］を選択します。
- 変数を［Watch（ウォッチ）］ウィンドウにドラッグアンドドロップします。
- ［Command（コマンド）］ウィンドウで、WATCHSET コマンドを使用します。

プログラムの実行が停止した場合や、プログラムの実行中に［View（表示）］ - ［Periodic Window Update（ウィンドウを定期的に更新）］を有効にした場合は、このウィンドウの内容が更新されます。

## ［Call Stack and Locals（コールスタックと

# ローカル）］ウィンドウ

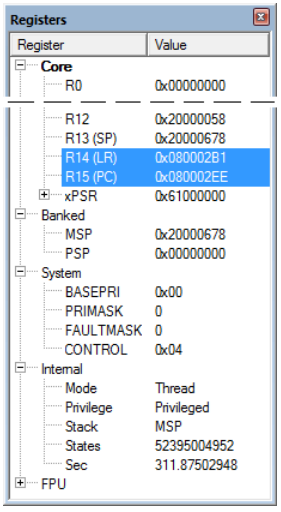
［Call Stack + Locals（コールスタックとローカル）］ウィンドウには、現在のプログラム位置の関数ネストと変数が表示されます。




ツールバーまたはメニューで［表示（View）］ - ［Call Stack Window（コールスタックウィンドウ）］の順に選択して、［Call Stack + Locals（コールスタックとローカル）］ウィンドウを開きます。

プログラムの実行が停止すると、［Call Stack + Locals（コールスタックとローカル）］ウィンドウに現在の関数ネストおよびローカル変数が自動的に表示されます。CMSIS-RTOS RTX を使用するアプリケーションのスレッドが表示されます。

# [Register（レジスタ）] ウィンドウ

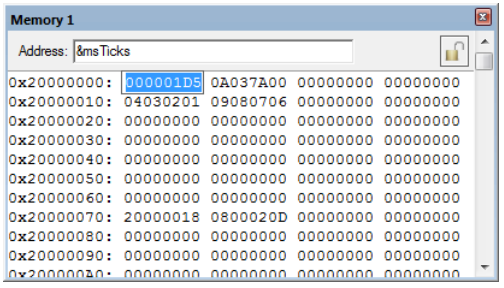



[Register（レジスタ）] ウィンドウには、マイクロコントローラのレジスタの内容が表示されます。

ツールバーまたは [View（表示）] - [Registers Window（レジスタウィンドウ）] メニューから、[Registers（レジスタ）] ウィンドウを開きます。

レジスタの値をダブルクリックするか、F2 キーを押して選択した値を編集することで、レジスタの内容を変更できます。現在変更されているレジスタは、青色でハイライトされます。プログラムの実行が停止すると、ウィンドウの値が
更新されます。

# [Memory（メモリ）] ウィンドウ




[Memory Windows（メモリウィンドウ）] を使用してメモリ領域を監視します。

ツールバーまたはメニューで [View（表示）] - [Memory Windows（メモリウィンドウ）] の順に選択して、[Memory（メモリ）] ウィンドウを開きます。

- ［Address（アドレス）］フィールドに式を入力して、メモリ領域を監視します。
- メモリの内容を監視するには、［Memory（メモリ）］ウィンドウのコンテキストメニューの［Modify Memory at（メモリの変更） …］コマンドを使用して、値をダブルクリックします。
- **コンテキストメニュー**を使用して、出力形式を選択できます。
- ［Memory（メモリ）］**ウィンドウ**を定期的に更新するには、［View（表示）］ - ［Periodic Window Update（ウィンドウを定期的に更新）］を有効にします。ウィンドウを手動で更新するには、［Toolbox（ツールボックス）］の［Update Windows（ウィンドウを更新）］を使用します。

［Memory（メモリ）］ウィンドウの更新を停止するには、［Lock（ロック）］ボタンをクリックします。ロック機能を使用して同じアドレス空間の値を比較するには、2 番目の［Memory（メモリ）］ウィンドウで同じセクションを表示します。

## ペリフェラルレジスタ

ペリフェラルレジスタは、プロセッサがペリフェラルを制御するために書き込みおよび読み出しを行うことができる、メモリマップされたレジスタです。［Peripherals（ペリフェラル）］メニューを使用すると、ネスト型ベクタ割り込みコントローラやシステム Tick タイマなどの**コアペリフェラル**にアクセスできます。［System Viewer（システムビューア）］を使用すると、デバイスのペリフェラルレジスタにアクセスできます。

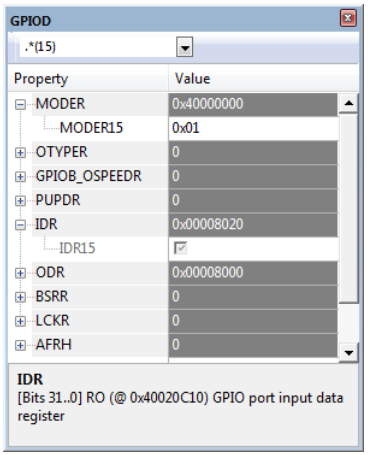
---

*注*
*［Peripherals（ペリフェラル）］メニューの内容は、選択されているマイクロコントローラによって変わります。*

---

### System Viewer（システムビューア）

［System Viewer（システムビューア）］ウィンドウには、デバイスのペリフェラルレジスタに関する情報が表示されます。




ツールバー、またはメニューの［Peripherals（ペリフェラル）］ - ［System Viewer（システムビューア）］から、ペリフェラルレジスタを開きます。

［System Viewer（システムビューア）］を使用すると、以下の操作を実行できます。

- ペリフェラルレジスタのプロパティと値を表示します。［View（表示）］ - ［Periodic Window Update（ウィンドウを定期的に更新）］が有効にすると、値は定期的に更新されます。
- デバッグ時にプロパティ値を変更する。
- 検索フィールドの［TR1 Regular Expressions（TR1 正規表現）］を使用して、特定のプロパティを検索します。正規表現の構文については、『*µVision ユーザガイド*』の付録で説明しています。

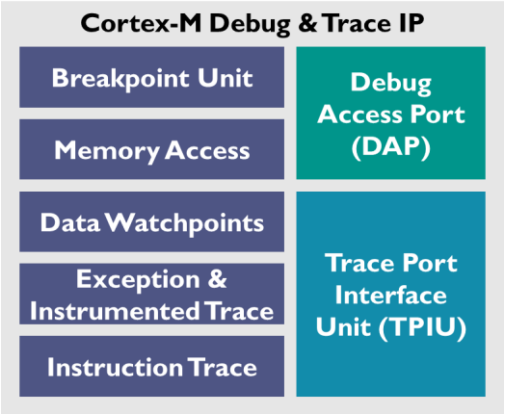
ペリフェラルレジスタへのアクセス方法とその使用方法の詳細については、オンラインマニュアルを参照して下さい。

# トレース

前述のとおり、デバッグの実行 - 停止には、モータ制御や通信アプリケーションなど、処理時間が重視されるプログラムをテストするときに明らかになる制限がいくつかあります。例えば、ブレークポイントやシングルステップコマンドによってシステムの動的動作が変わります。デバッグの代替として、このセクションで説明するトレース機能を使用して実行中のシステムを解析します。

Cortex-M プロセッサは、次のトレース情報を生成できる CoreSight ロジックを統合します。

- **データウォッチポイント**はデータ値とプログラムアドレスと共にメモリアクセスを記録し、必要に応じてプログラムの実行を停止します。
- **例外トレース**は、割り込みと例外の詳細を出力します。
- **インストルメント化トレース**はプログラムのイベントを通信し、printf 形式のデバッグメッセージと RTOS イベントビューアを有効にします。
- **命令トレース**は完全なプログラム実行をストリーミングし、記録と解析を行います。




**トレースポートインタフェースユニット**（TPIU）はほとんどの Cortex-M3、Cortex-M4、および Cortex-M7 プロセッサベースのマイクロコントローラで利用でき、以下を介して上記のトレース情報を出力します。

- **シリアルワイヤトレース出力**（SWO）は、JTAG ではなくシリアルワイヤデバッグモードと組み合わせた場合にのみ動作し、命令トレースをサポートしません。
- **4 ピントレース出力**は、ハイエンドのマイクロコントローラで利用でき、命令トレースに必要な高帯域幅を使用できます。

一部のマイクロコントローラでは、標準デバッグインタフェースを使用して読み取ることができるオンチップ**トレースバッファ**にトレース情報を格納できます。

- Cortex-M3、Cortex-M4、および Cortex-M7 には、前述のすべてのトレースデータを格納する、オプションの **Embedded Trace Buffer**（ETB）が用意されています。
- Cortex-M0+ には、命令トレースのみをサポートする、オプションの**マイクロトレースバッファ**（MTB）が用意されています。

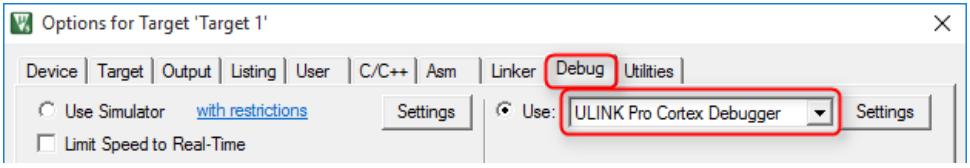
必須のトレースインタフェースは、マイクロコントローラとデバッグアダプタの両方でサポートされている必要があります。以下の表に、さまざまなデバッグアダプタでサポートされているトレース方法を示します。

| 機能 | ULINKpro | ULINKpro-D | ULINK2 | ST-Link v2 |
|---|---|---|---|---|
| シリアルワイヤ出力（SWO） | • | • | • | • |
| 最大 SWO クロック周波数 | 200 MHz | 200 MHz | 3.75 MHz | 2 MHz |
| ストリーミング用 4 ピントレース出力 | • | • | • | • |
| Embedded Trace Buffer（ETB） | • | • | • | • |
| マイクロトレースバッファ（MTB） | • | • | • | • |

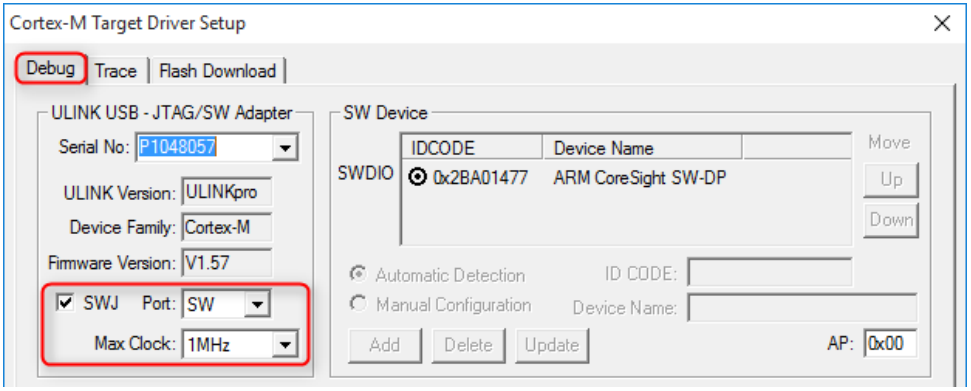
## シリアルワイヤ出力を使用したトレース

シリアルワイヤトレース出力（SWO）を使用するには、次の手順に従います。

ツールバーで［Options for Target（ターゲットのオプション）］をクリックし、［Debug（デバッグ）］タブを選択します。適切なデバッグアダプタが選択され有効になっていることを確認します。



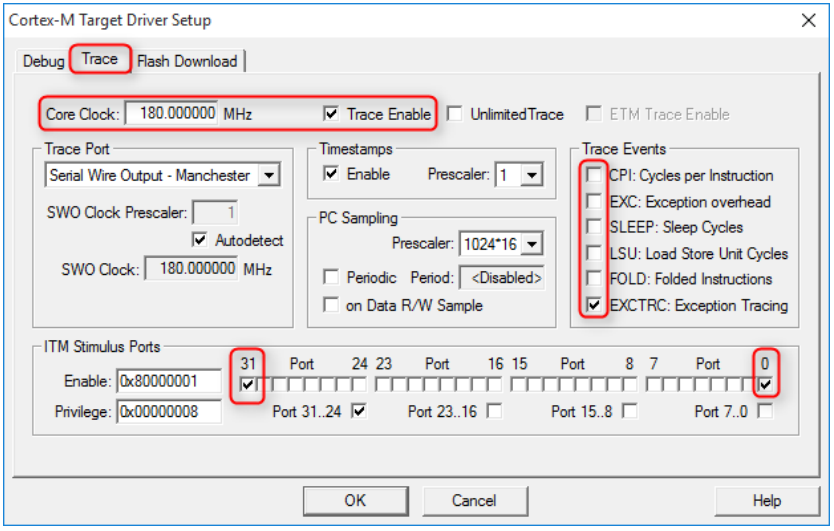

［Settings（設定）］ボタンをクリックします。［Debug（デバッグ）］ダイアログで、デバッグの［Port:（ポート：）］で［SW］を選択し、デバイスのデバッグユニットと通信するための［Max Clock（最大クロック）］周波数を設定します。




☞ [Trace（トレース）] タブをクリックします。 [Core Clock（コアクロック）] の設定が正しいことを確認します。 [Trace Enable（トレースを有効化）] をオンにして、監視する [Trace Events（トレースイベント）] を選択します。

printf 形式のデバッグを行うには、 [ITM Stimulus Port（ITM スティムラスポート）] の [0] をオンにします。

RTOS イベントを表示するには、 [ITM Stimulus Port（ITM スティムラスポート）] の [31] をオンにします。



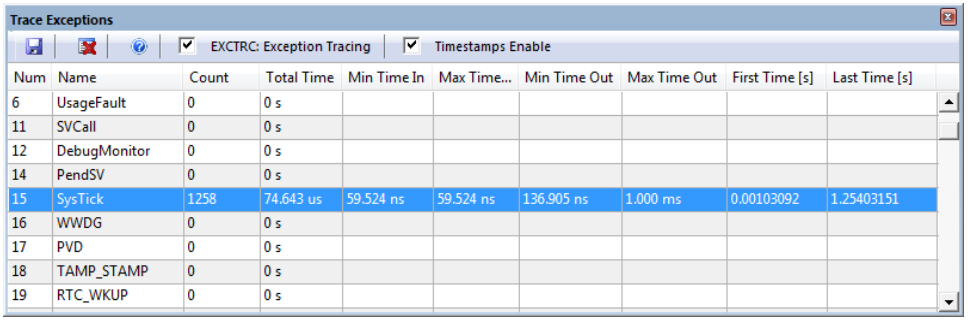
*注*
*多くのトレース機能を有効にすると、シリアルワイヤ出力の通信がオーバーフローする場合があります。このような接続エラーは µVision ステータスバーに表示されます。*

*ULINKpro デバッグ/トレースアダプタには高帯域幅トレースが用意されているため、このような通信のオーバーフローが起こることはまれです。トレース通信でオーバーフローを避けるために現在必要とされているトレース機能のみを有効にして下さい。*

# トレースの例外

［Exception Trace（例外トレース）］ウィンドウには、例外と割り込みに関する統計データが表示されます。

［Trace Windows（トレースウィンドウ）］をクリックし、ツールバーから［Trace Exceptions（トレースの例外）］を選択するか、メニューで［View（表示）］ - ［Trace（トレース）］ - ［Trace Exceptions（トレースの例外）］の順に選択してウィンドウを開きます。




［Trace Exceptions（トレースの例外）］ウィンドウでデータを取得するには、次の手順に従います。

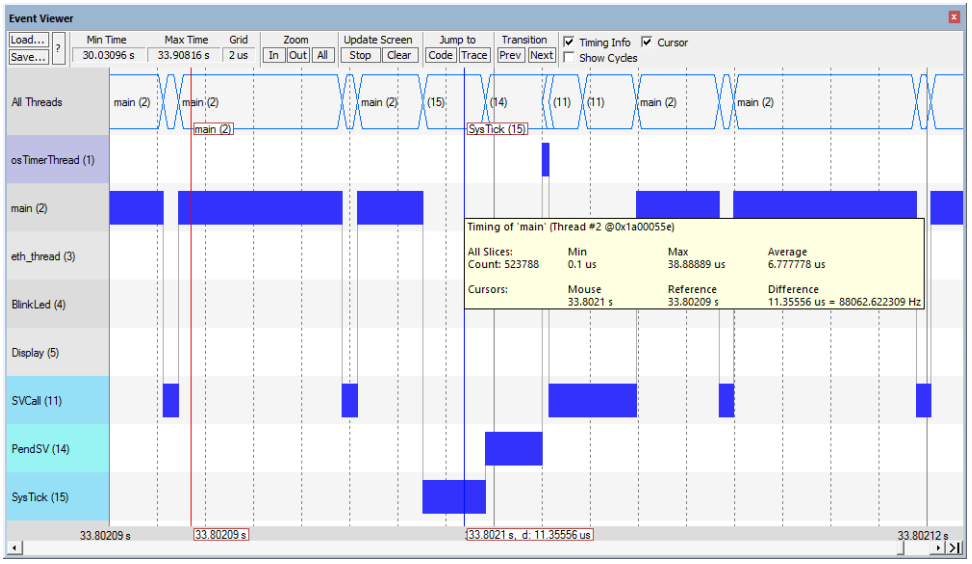
- 上記で説明したように、［Debug Settings Trace（デバッグ設定のトレース）］ダイアログの［Trace Enable（トレースを有効化）］を設定します。
- ［EXCTRC: Exception Tracing（EXCTRC：例外のトレース）］をオンにします。

- [Timestamps Enable（タイムスタンプを有効化）] を設定します。

*注*
*ロジックアナライザで設定された変数アクセスも［Trace Data（トレースデータ）］ウィンドウに表示されます。*

## [Event Viewer（イベントビューア）]

[Event Viewer（イベントビューア）] には、RTOS スレッドだけでなく、割り込みおよび例外のタイミング情報も表示されます。このウィンドウを開くには、メニューで [Debug（デバッグ）] - [OS Support（OS サポート）] - [Event Viewer（イベントビューア）] の順に選択します。




[Event Viewer（イベントビューア）] ウィンドウでデータを取得するには、次の手順に従います。

- 上記で説明したように、［Debug Settings Trace（デバッグ設定のトレース）］ダイアログの [Trace Enable（トレースを有効化）] を設定します。
- CMSIS-RTOS スレッドのタイミング情報に対する [ITM Stimulus Port（ITM スティムラスポート）] の [31] をオンにします。

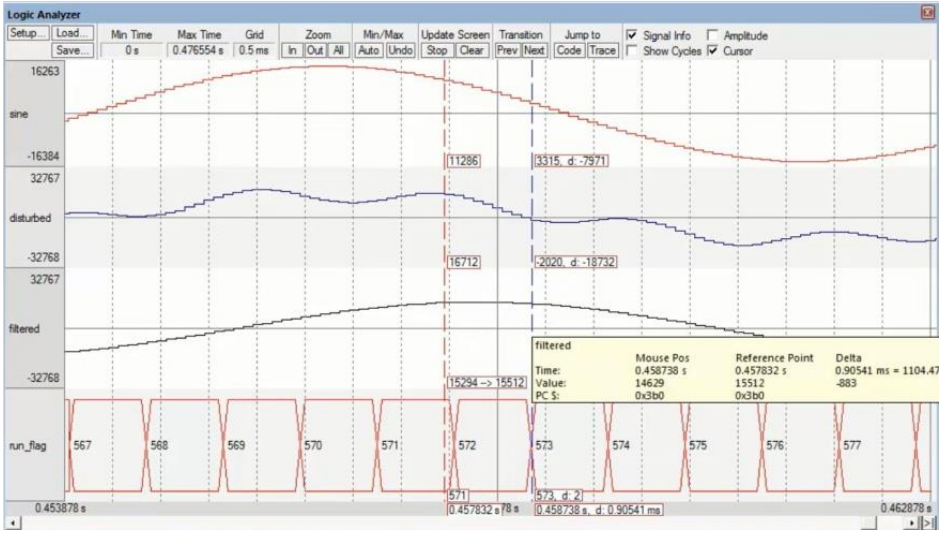
- 割り込みおよび例外のタイミング情報の［EXCTRC: Exception Tracing（EXCTRC：例外トレース）］をオンにします。
- ［Timestamps Enable（タイムスタンプを有効化）］を設定します。

*注*

*デバッガからは、トレースを行わなくても入手できる詳細な RTOS およびスレッドのステータス情報も得られます。詳細については、「**CMSIS-RTOS システムとスレッドビューア**」セクション（48 ページ）を参照して下さい。*

## ［Logic Analyzer（ロジックアナライザ）］

［Logic Analyzer（ロジックアナライザ）］ウィンドウには、変数値の変化が経時的に表示されます。監視できる変数は最大 4 つです。［Logic Analyzer（ロジックアナライザ）］に変数を追加するには、デバッグモードの while 文で変数を右クリックし、［Add <variable> to…（<変数> を … に追加）］- ［Logic Analyzer（ロジックアナライザ）］の順に選択します。［Logic Analyzer（ロジックアナライザ）］ウィンドウを開くには、［View（表示）］ - ［Analysis Windows（解析ウィンドウ）］ - ［Logic Analyzer（ロジックアナライザ）］の順に選択します。




［Logic Analyzer（ロジックアナライザ）］ウィンドウでデータを取得するには、次の手順に従います。

- 上記で説明したように、［Debug Settings Trace（デバッグ設定のトレース）］ダイアログの［Trace Enable（トレースを有効化）］を設定します。
- ［Timestamps Enable（タイムスタンプを有効化）］を設定します。

*注*
*［Logic Analyzer（ロジックアナライザ）］で監視する変数アクセスは、［Trace Data（トレースデータ）］ウィンドウにも表示されます。詳細については、『µVision ユーザガイド』の「Debugging（デバッグ）」を参照して下さい。*

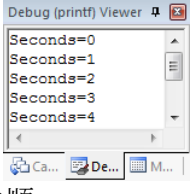
# デバッグ（printf）ビューア

［Debug（printf）Viewer（デバッグ（printf）ビューア）］ウィンドウには、［ITM Stimulus Port（ITM スティムラスポート）］の［0］を通じて順に送信されるデータストリームが表示されます。*printf()* デバッグを有効にするには、56 ページで説明されているように、Compiler（コンパイラ）ソフトウェアコンポーネントを使用します。

この *fputc()* 関数は、*printf()* メッセージを［Debug（printf）Viewer（デバッグ（printf）ビューア）］にリダイレクトします（下図を参照）。

```
int seconds;                          // Second counter
  :
while (1) {
  LED_On ();                          // Switch on
  delay ();                           // Delay
  LED_Off ();                         // Switch off
  delay ();                           // Delay
  printf ("Seconds=%d¥n", seconds++); // Debug output
}
```

ウィンドウを開くには、［Serial Windows（シリアルウィンドウ）］をクリックして、ツールバーから［Debug（printf）Viewer（デバッグ（printf）ビューア）］を選択するか、メニューで［View（表示）］ - ［Serial Windows（シリアルウィンドウ）］ - ［Debug（printf）Viewer（デバッグ（printf）ビューア）］の順に選択します。

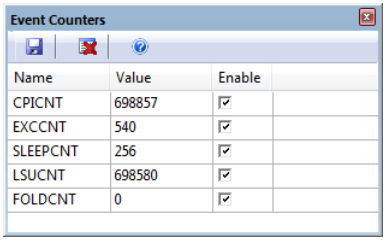


［Debug（printf）Viewer（デバッグ（printf）ビューア）］ウィンドウでデータを取得するには、次の手順に従います。

- 上記で説明したように、［Debug Settings Trace（デバッグ設定のトレース）］ダイアログの［Trace Enable（トレースを有効化）］を設定します。
- ［Timestamps Enable（タイムスタンプを有効化）］を設定します。
- ［ITM Stimulus Port（ITM スティムラスポート）］の［0］をオンにします。

## ［Event Counters（イベントカウンタ）］

［Event Counters（イベントカウンタ）］には、イベントをトリガした回数を示す累積数が表示されます。




ツールバーで［Trace Windows（トレースウィンドウ）］ - ［Event Counters（イベントカウンタ）］の順に選択します。

メニューで［View（表示）］ - ［Trace（トレース）］ - ［Event Counters（イベントカウンタ）］の順に選択します。

このウィンドウでデータを取得するには、次の手順に従います。

- 上記で説明したように、［Debug Settings Trace（デバッグ設定のトレース）］ダイアログの［Trace Enable（トレースを有効化）］を設定します。
- 必要に応じて、ダイアログで［Event Counters（イベントカウンタ）］を有効にします。

イベントカウンタは、次の項目を示すパフォーマンスインジケータです。

- CPICNT：例外オーバーヘッドサイクル：フラッシュのウェイト状態を示します。
- EXCCNT：命令あたりのエクストラサイクル：例外の頻度を示します。
- SLEEPCNT：スリープサイクル：スリープモードに費した時間を示します。

- LSUCNT：ロードストアユニットサイクル：マルチサイクルロードストア命令を実行するために必要な追加サイクルを示します。
- FOLDCNT：Folded 命令：ゼロサイクルで実行する命令を示します。

# 4 ピン出力を使用したトレース

4 ピントレース出力を使用すると、「**シリアルワイヤ出力を使用したトレース**」セクションに記載されているすべての機能を使用できますが、トレース通信帯域幅は高くなります。命令トレースも可能です。

**ULINKpro デバッグ/トレースアダプタ**は、プログラムの実行を詳細に把握できるこのパラレル 4 ピントレース出力（ETM トレースとも呼ばれる）をサポートしています。

---

*注*
*以下に記載されている機能の詳細については、『µVision ユーザガイド』の「Debugging（デバッグ）」を参照して下さい。*

---

ULINKpro と組み合わせて使用すると、MDK は以下の高度な解析機能の命令トレースデータをストリーミングできます。

- **［Code Coverage（コードカバレッジ）］**は、実行済みのコードをマークし、コード実行に関する統計を提供します。この機能は突発的な実行エラーの特定に役立つため、ソフトウェア認証の要件となっていることがよくあります。
- **［Performance Analyzer（パフォーマンスアナライザ）］**では、関数およびプログラムブロックの実行時間を記録し、それを表示します。ここにはプロセッサのサイクル使用状況が表示され、アルゴリズムのホットスポットを特定して最適化できます。
- **［Trace Data（トレースデータ）］ウィンドウ**には、Cortex-M デバイスで実行された命令の履歴が表示されます。

# オンチップトレースバッファを使用したトレース

- 場合によっては、マイクロコントローラまたはターゲットハードウェアでトレース出力ピンを使用できないことがあります。代替方法として、**［Trace Data（トレースデータ）］ウィンドウ**をサポートするオンチップ**トレースバッファ**を使用できます。

# ミドルウェア

今日のマイクロコントローラデバイスには、多くのエンベデッド設計要件を満たすさまざまな通信ペリフェラルが用意されています。このような複雑なオンチップペリフェラルを効率的に使用するには、ミドルウェアが不可欠です。

---

*注*

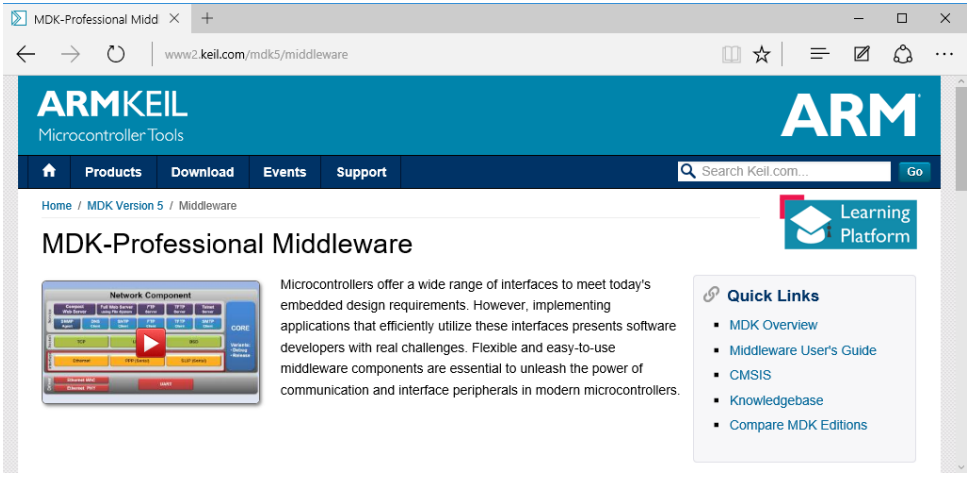
*本章では、MDK-Professional に含まれているミドルウェアについて説明します。MDK は、他のさまざまなベンダのミドルウェアでも使用できます。*

*公開されているソフトウェアパックの一覧については、http://www.keil.com/pack を参照して下さい。*

---

**MDK-Professional** ソフトウェアパックには、**TCP/IP ネットワーク**、**USB ホスト**と **USB デバイス**の通信、データ保存用の**ファイルシステム**、および**グラフィカルユーザインタフェース**のコンポーネントを備えた、ロイヤルティフリーのミドルウェアが含まれています。

MDK-Professional の一部として提供されているミドルウェアの詳細については、www.keil.com/mdk5/middleware を参照して下さい。



この Web ページには、以下のミドルウェアの概要とリンクが記載されています。

- MDK-Professional Middleware の**ユーザガイド**
- デバイス固有のドライバに関する情報を記載した**デバイスリスト**
- **サンプルプロジェクト**の情報と使用方法

ミドルウェアは、デバイス固有の CMSIS ドライバを使用してデバイスペリフェラルとのインタフェースをとります。詳細については、「**CMSIS** ドライバ」セクション（52 ページ）を参照して下さい。

マイクロコントローラアプリケーションでは、複数のコンポーネントを組み合わせることがよくあります。**[Manage Run-Time Environment（実行時環境の管理）]** ダイアログを使用すると、MDK-Professional Middleware を簡単に選択して組み合わせることができます。ソフトウェアパックとして提供されているサードパーティ製コンポーネントをミドルウェアコンポーネントのリストに追加することもできます。

MDK-Professional Middleware の一般的な使用例は次のとおりです。

- ストレージ機能を備えた Web サーバ：ネットワークおよびファイルシステムコンポーネント
- USB メモリスティック：USB デバイスおよびファイルシステムコンポーネント
- 表示およびログ機能を備えた産業用制御ユニット：グラフィックス、USB ホスト、およびファイルシステムコンポーネント
- 複数のミドルウェアコンポーネントの組み合わせ例を示した「FTP サーバサンプル」セクション（115 ページ）を参照して下さい。

これ以降のセクションでは、MDK-Professional Middleware の各ソフトウェアコンポーネントの概要について説明します。

---

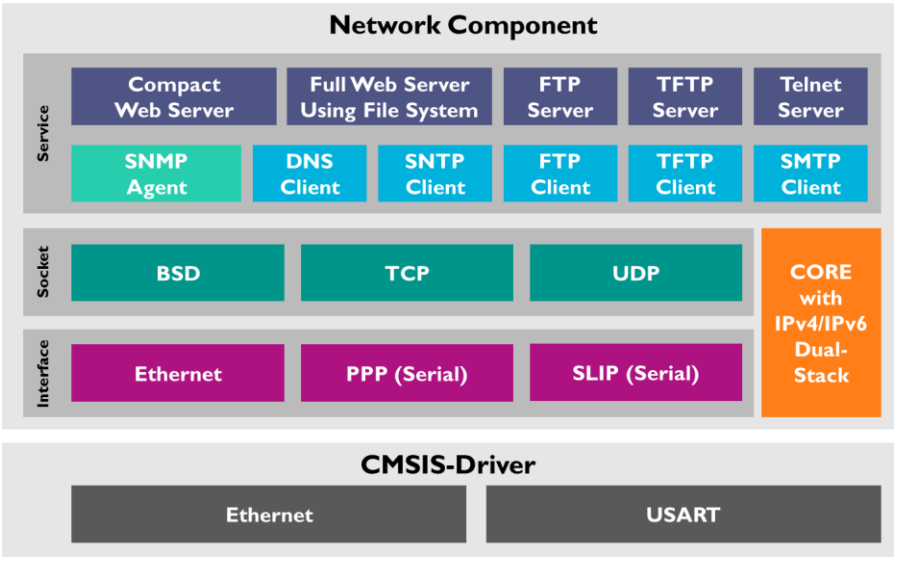
*注*

*インストールすると、MDK-Professional の 7 日間評価ライセンスが付与されます。詳細については、「**インストール**」の章（10 ページ）を参照して下さい。*

---

# ネットワークコンポーネント

**ネットワークコンポーネント**は TCP/IP 通信プロトコルを使用し、サービス、プロトコルソケット、および物理通信インタフェースをサポートしています。IPv4 および IPv6 接続がサポートされています。




さまざまな**サービス**から、共通ネットワークタスク用のプログラムテンプレートが提供されています。

- **コンパクト Web サーバ**は Web ページを ROM に保存し、**フル Web サーバ**は **[File System（ファイルシステム）]** コンポーネントを使用してページデータを保存します。どちらのサーバも、CGI スクリプト、AJAX、および SOAP テクノロジを使用して、動的なページコンテンツをサポートします。
- **FTP** または **TFTP** はファイル転送をサポートします。FTP はすべてのファイル操作コマンドを備え、TFTP はリモートデバイスをブートロードできます。どちらもクライアントとサーバで使用できます。
- **Telnet サーバ**は、IP ネットワークのコマンドラインインタフェースを提供します。
- **SNMP エージェント**は、シンプルネットワークマネージメントプロトコルを使用して、ネットワークマネージャにデバイス情報を通知します。

- **DNS クライアント**は、それぞれの IP アドレスに対してドメイン名を解決します。自由に設定可能なネームサーバを利用します。
- **SNTP クライアント**はクロックを同期し、デバイスがデータネットワーク上で正確な時間信号を取得できるようになります。
- **SMTP** クライアントは、簡易メール転送プロトコルを使用してステータス電子メールを送信します。

すべての**サービス**は通信ソケットに依存しており、これは、**TCP**（接続指向、信頼性の高いフルデュプレックスプロトコル）、**UDP**（データストリーミングを行うトランザクション指向プロトコル）、または **BSD**（バークレーソケットインタフェース）のいずれかである可能性があります。

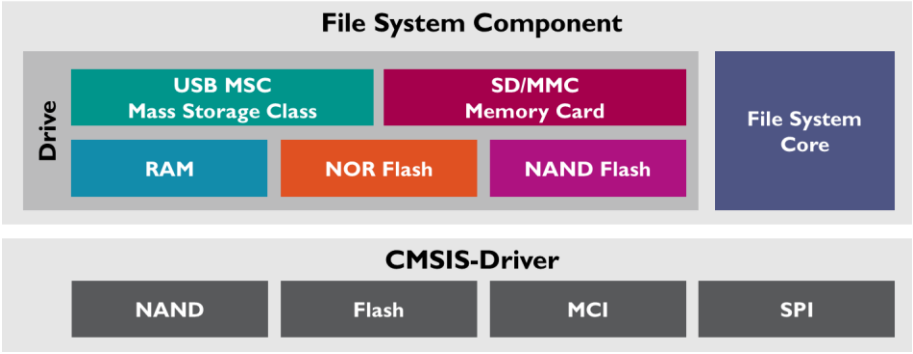
物理**インタフェース**は、**イーサネット**（LAN 接続向け）か、**PPP**（2 つのデバイス間の直接接続向け）または **SLIP**（シリアル接続を介したインターネットプロトコル）などのシリアル接続のいずれかである可能性があります。

インタフェースによっては、ネットワークコンポーネントがデバイス固有のハードウェアインタフェースを提供するために **CMSIS ドライバ**に依存するものがあります。イーサネットでは**イーサネット MAC** および **PHY** ドライバが必要で、シリアル接続（PPP/SLIP）では **UART** または**モデム**ドライバが必要になります。

**ネットワークコア**は、さまざまな診断メッセージを備えた*デバッグ* バリアント、およびこれらの診断を省略する*リリース* バリアントで使用できます。IPv4 および IPv6 を使用した IP 通信をサポートします。

# ファイルシステム コンポーネント

**ファイルシステムコンポーネント**を使用すると、組み込みアプリケーションを使用して、RAM、NAND や NOR 型フラッシュ、メモリカード、または USB メモリスティックなどのストレージデバイスでファイルの作成、保存、読み出し、変更を行うことができます。



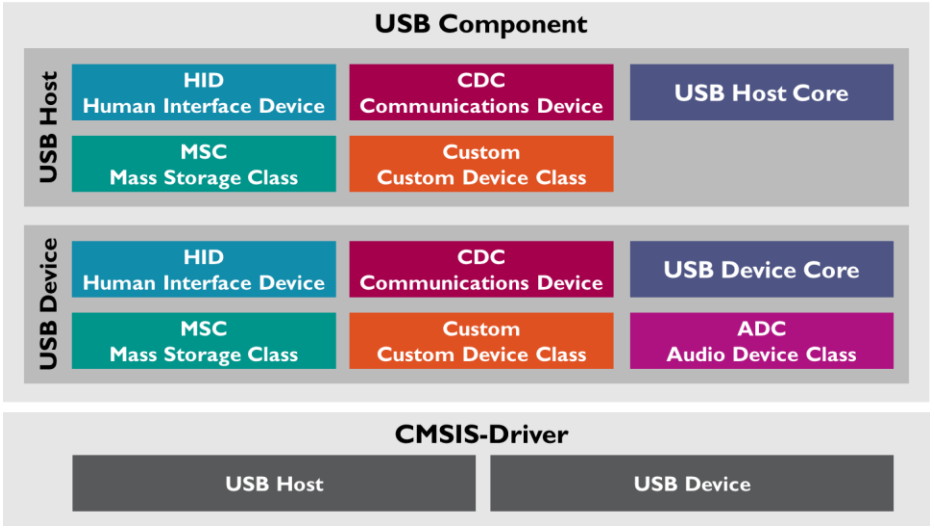

各ストレージデバイスは、**ドライブ**としてアクセスおよび参照できます。ファイルシステムコンポーネントは同じ種類の複数のドライブをサポートします。例えば、システムで複数のメモリカードを使用することができます。

**ファイルシステムコア**はスレッドセーフで、複数のドライブに同時にアクセスでき、8.3 形式の短いファイル名と最大 255 文字の長いファイル名の 2 つのファイル名バリアントで使用できる FAT システムを使用します。

NAND および NOR 型フラッシュチップや、MCI または SPI を使用したメモリカードなどの物理メディアにアクセスするには、**CMSIS ドライバ**が必要です。

# USB コンポーネント

**USB デバイスコンポーネント**は USB ホストおよびデバイスの機能を実装し、ほとんどのコンピュータシステムで使用できる標準デバイスドライバクラスを使用します。ホストドライバを開発する必要はありません。



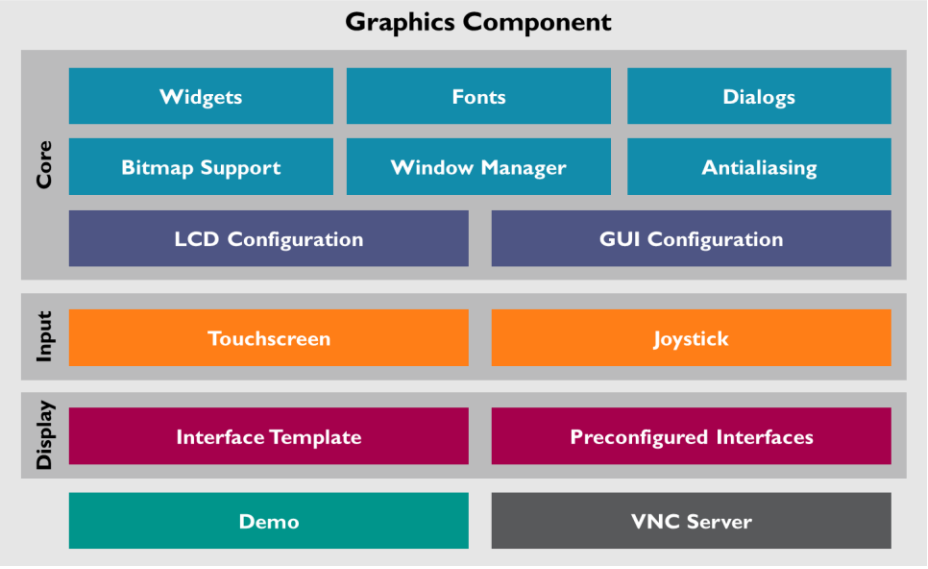

- **ヒューマンインタフェースデバイスクラス**（HID）は、キーボード、ジョイスティック、またはマウスを実装します。ただし、HID は単純なデータ交換にも使用できます。
- **マスストレージクラス**（MSC）は、ファイル交換に使用します（USB メモリスティックなど）。
- **通信デバイスクラス**（CDC）は、仮想シリアルポート（サブクラス ACM を使用）またはネットワーク接続（サブクラス NCM を使用）を実装します。
- **オーディオデバイスクラス**（ADC）は、オーディオストリーミングを行います。
- **カスタムクラス**は、新しい USB クラスかサポートされていない USB クラスに使用します。

USB コンポーネントは、複数のデバイスクラスを実装する**複合 USB デバイス**をサポートします。

このコンポーネントには、USB CMSIS ドライバが必要です。アプリケーションによっては、USB 1.1 (Full-Speed USB) や USB 2.0 (High-Speed USB) 仕様に準拠する必要があります。

# グラフィックスコンポーネント

**グラフィックスコンポーネント**は、グラフィカルユーザインタフェースの構築に必要な内容がすべて含まれている、包括的なライブラリです。




中心的な機能は次のとおりです。

- 任意の数のウィンドウまたはダイアログを操作する**ウィンドウマネージャ**。
- そのままの状態で使用できる**フォント**、および**ウィジェット**や**ダイアログ**と呼ばれるウィンドウ要素。
- JPEG やその他の一般的な形式を含む、**ビットマップサポート**。
- 表示を滑らかにする**アンチエイリアシング**。
- 柔軟性が高く設定可能な**ディスプレイ**および**ユーザインタフェース**パラメータ。
- ユーザインタフェースは、**タッチスクリーン**や**ジョイスティック**などの入力デバイスを使用して制御できます。

グラフィックスコンポーネントは、一般的なディスプレイ向けに**事前設定されたインタフェース**を使用して、さまざまなディスプレイコントローラとのインタフェースをとります。新しいディスプレイ向けのサポートを追加する**インタフェーステンプレート**に対応します。

**VNC サーバ**を使用すると、**ネットワークコンポーネント**を使用して TCP/IP を介してグラフィカルユーザインタフェースをリモート制御できます。

**デモ**はすべての主要機能を示し、GUI の豊富なコードスニペットの基になります。

# ミドルウェアバージョン 7 への移行

MDK には、µVision プロジェクトを新しいミドルウェアバージョン 7 に移行するときに役立つ内蔵機能が揃っています。大部分のコンポーネントでは、コンフィギュレーションファイルの更新のみが必要になります（以下を参照）。ただし、ネットワークコンポーネントは、IPv4 のみのサポートから IPv4/IPv6 のデュアルスタックサポートに変更されるため、それ以外の移行作業も必要となります。

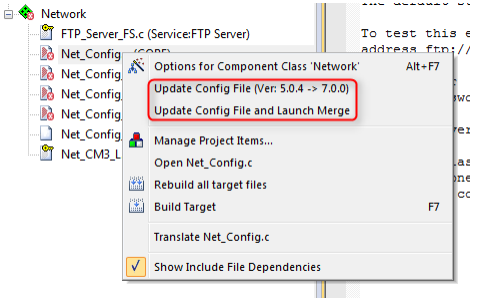
## ネットワークコンポーネントの変更点

### コアの変更

これまで、ネットワークコンポーネントのコアは **Release** または **Debug** バリアントで利用できました。ミドルウェア v7 では、これが **IPv4/IPv6 Release** または **IPv4/IPv6 Debug** に変更されます。旧コンポーネントでプロジェクトを開くと、**[Build Output（ビルド出力）]** ウィンドウにエラーが表示されます。対応する新しいバリアントに変更して下さい。

## コンフィギュレーションファイルの更新

µVision の［Project（プロジェクト）］ウィンドウに特別なアイコンが表示され、更新が必要なコンフィギュレーションファイルがハイライトされます。これまでのコンフィギュレーションファイルを上書きするか、更新して内容をマージするかを選択できます。




［Tools（ツール）］ ・ ［Configure Merge Tool（マージツールの設定）］に移動し、選択するマージツールを指定します。

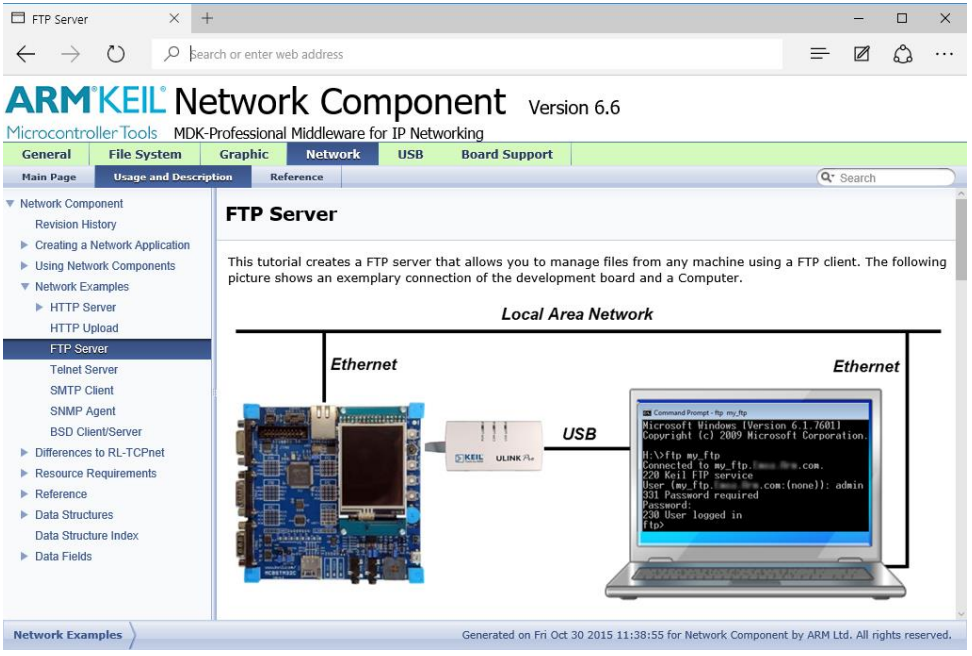
## API の変更点

ネットワークコンポーネントのマニュアルには、旧 API から新 API へのプロジェクトの移行方法に関するセクションが用意されています。このセクションには、サービス、ソケット、およびインタフェースの移行に関する一般的な推奨事項や、ミドルウェア v5/v6 から移行しているのか RL-TCPnet から移行しているのるかに応じた API の対照比較が記載されています。

# FTP サーバサンプル

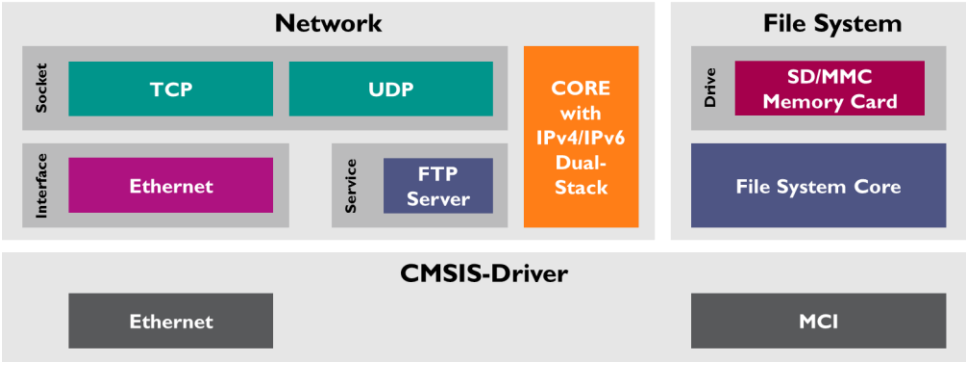
FTP サーバサンプルは、複数のミドルウェアコンポーネントの組み合わせを示す参照アプリケーションです。使用できるさまざまなサンプルプロジェクトの詳細については、「**サンプルプロジェクトを使用したインストールの検証**」セクション（14 ページ）を参照して下さい。

FTP サーバを使用すると、TCP/IP ネットワーク上でファイルを交換および操作できます。ミドルウェアのマニュアルには、FTP サーバおよび参照アプリケーションの詳細が記載されています。



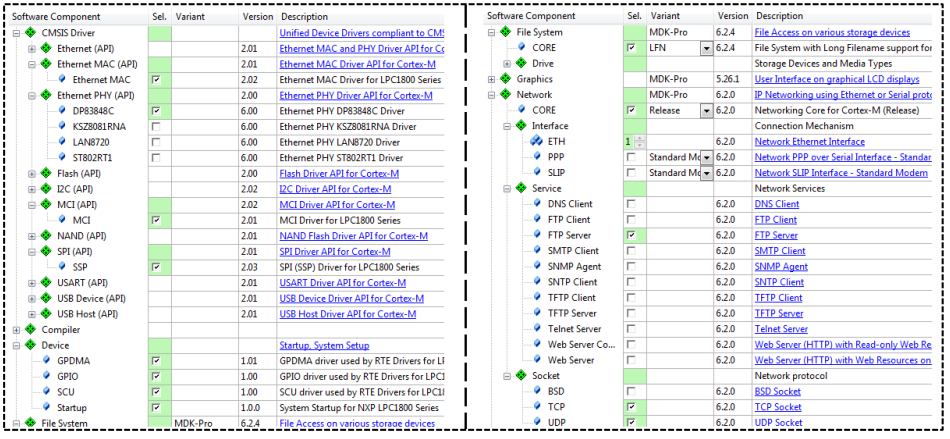
複数のミドルウェアコンポーネントがこの FTP サーバのビルディングブロックになっています。**ファイルシステム**はファイル操作を行うときに必要です。**ネットワーク**コンポーネントのさまざまなパーツがネットワーキングインタフェースを構築します。

FTP サーバサンプルを作成するには、MDK-Professional Middleware の以下のソフトウェアコンポーネントが必要です。




前述のとおり、CMSIS ドライバには、マイクロコントローラペリフェラルと MDK-Professional Middleware 間のインタフェースが装備されています。

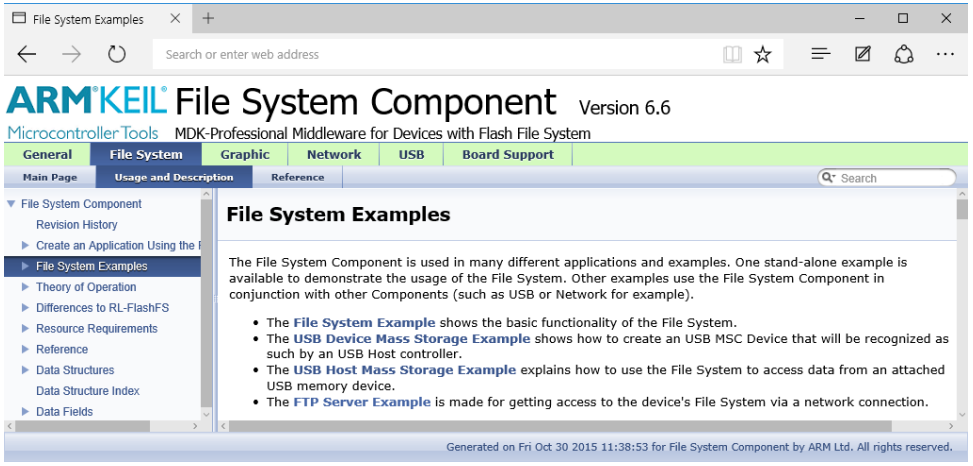
**[Manage Run-Time Environment（実行時環境の管理）]** ダイアログには、FTP サーバサンプル用に選択されたソフトウェアコンポーネントが示されています。



# ミドルウェアの使用

MDK-Professional Middleware コンポーネントを使用して、独自のアプリケーションを作成します。詳細については、『MDK-Professional Middleware ユーザガイド』を参照して下さい。このガイドには、以下の内容についてコンポーネントごとに説明しているセクションがあります。

- 「**サンプルプロジェクト**」では、ソフトウェアコンポーネントの主要な製品機能について概説します。サンプルは、複数の評価ボードでテスト、実装、および実証されています。参照アプリケーションまたは開発の基礎として使用して下さい。
- 「**リソース要件**」では、CMSIS-RTOS およびメモリフットプリントのスレッドおよびスタックリソースについて説明します。
- 「**アプリケーションの作成**」には、組み込みアプリケーションのコンポーネントを使用するときに必要な手順が記載されています。
- 「**参考文献**」には、API およびファイルドキュメントが含まれています。




学習プラットフォーム（www.keil.com/learn）には、ミドルウェアの一般的な使用事例を示すチュートリアルやビデオが用意されています。また、以下のアプリケーションノートも参照して下さい。

- www.keil.com/appnotes/docs/apnt_268.asp - ファイルシステムおよびグラフィカルユーザインタフェースを使用した USB ホストアプリケーション。

- www.keil.com/appnotes/docs/apnt_271.asp - Web 対応 MEMS センサプラットフォーム。
- www.keil.com/appnotes/docs/apnt_272.asp - Web 対応ボイスレコーダ。
- www.keil.com/appnotes/docs/apnt_273.asp - USB デバイスインタフェースを備えたアナログ/デジタルデータロガー。

さまざまなミドルウェアコンポーネントを使用する一般的な手順は以下のとおりです。

- **ソフトウェアコンポーネントの追加**（120 ページ）：**[Manage Run-Time Environment（実行時環境の管理）]** ダイアログで、アプリケーションに必要なソフトウェアコンポーネントを選択します。
- **ミドルウェアの設定**（122 ページ）：関連するコンフィギュレーションファイルで、ソフトウェアコンポーネントのパラメータを調整します。
- **ドライバの設定**（124 ページ）：ミドルウェアコンポーネントをマイクロコントローラの物理 I/O ピンに接続するペリフェラルインタフェースを識別して設定します。
- **システムリソースの調整**（125 ページ）：ミドルウェアコンポーネントは RTOS、メモリ、およびスタックリソースを使用し、これにより、CMSIS-RTOS RTX などへの設定が暗黙的に行われる場合があります。
- **アプリケーション機能の実装**（128 ページ）：選択したコンポーネントの API 関数を使用して、アプリケーション固有の動作を実装します。コードテンプレートを使用すると、関連するソースコードを簡単に作成できます。
- **ビルドとダウンロード**（131 ページ）：アプリケーションのコンパイルとリンク付けを行った後、「**デバッガの使用**」の章（84 ページ）で説明する手順に従って、イメージをターゲットハードウェアにダウンロードします。
- **検証とデバッグ**（132 ページ）：テストユーティリティとデバッグおよびトレース機能については、「アプリケーションの作成」の章（60 ページ）で説明しています。

# USB デバイス HID サンプル

- 前述の手順は一般的なものであり、MDK-Professional Middleware のすべてのコンポーネントに適用されますが、以下の USB デバイス HID サンプルではこれらの手順を実践しています。このサンプルでは、USB を介してマイクロコントローラをホストコンピュータに接続する USB HID デバイスアプリケーションを作成します。PC でユーティリティプログラム *HIDClient.exe* を使用して、開発ボードの LED を制御します。
- この USB デバイス HID サンプルでは、LPC1857 マイクロコントローラが搭載された MCB1800 開発ボードを使用します。このサンプルは、「**CMSIS-RTOS RTX を使用した Blinky**」プロジェクト（61 ページ）、ソースファイル *main.c*、*LED.c*、*LED.h*、およびコンフィギュレーションファイルを基にしています。

*注*
*他のスタータキットまたは評価ボードでこのサンプルを使用する場合は、コードおよびピン設定を調整する必要があります。このサンプルは、CMSIS ドライバをサポートしている多くのマイクロコントローラデバイスファミリの Pack Installer でビルド済みプロジェクトとして使用できます。*

# ソフトウェアコンポーネントの追加

USB デバイス HID サンプルを作成するには、「**CMSIS-RTOS RTX を使用した Blinky**」プロジェクト（61 ページ）から始めます。

［**Manage Run-Time Environment（実行時環境の管理）**］ダイアログを使用して、特定のソフトウェアコンポーネントを追加します。

**USB コンポーネント（110 ページで説明）から、以下の操作を行います。**

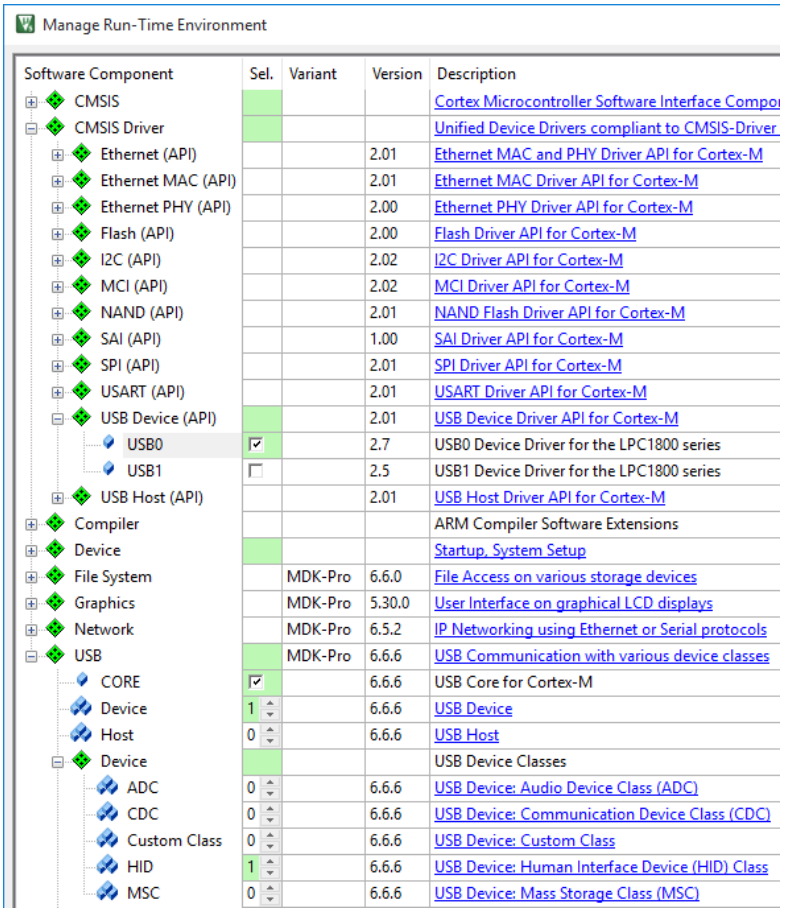
- **::USB:CORE** を選択して、USB 通信に必要な基本機能を含めます。
- **::USB:Device** を［1］に設定して、1 つの USB デバイスインスタンスを作成します。
- **::USB:Device:HID** を［1］に設定して、HID デバイスクラスインスタンスを作成します。同じクラスの複数のインスタンスを選択したり、他のデバイスクラスを含めたりした場合は、複合 USB デバイスが作成されます。

USB コンポーネント（110 **ページで説明）から、以下の操作を行います。**

- **::Drivers:USB Device (API)** から、アプリケーションに適したドライバを選択します。デバイスによっては USB Full-Speed および High-Speed 向けの固有のドライバが用意されている場合もあるほか、複合版ドライバを備えたマイクロコントローラもあります。ここでは、［**USB0**］を選択します。

ヒント：［**Description（説明）**］列のハイパーリンクをクリックすると、各ソフトウェアコンポーネントの詳細なマニュアルが表示されます。

以下の図は、これらのコンポーネントを追加した後の［Manage Run-Time Environment（実行時環境の管理）］ダイアログです。

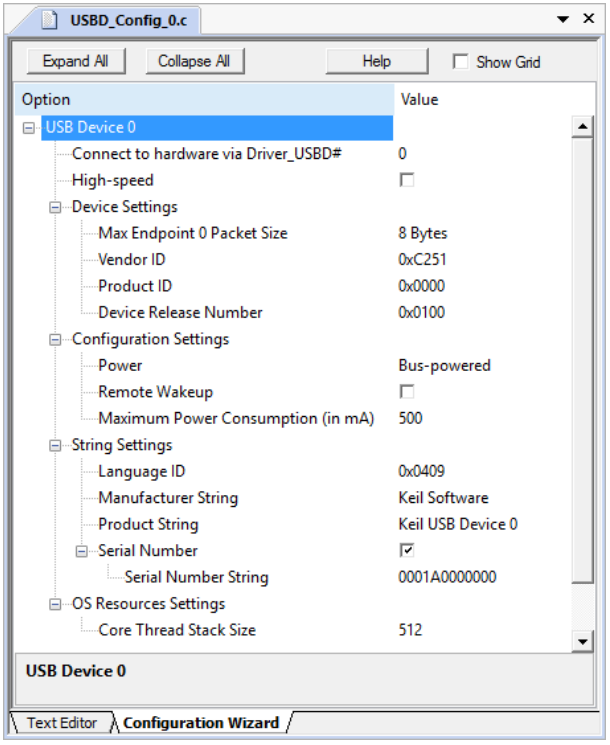


# ミドルウェアの設定

すべての MDK-Professional Middleware コンポーネントには、アプリケーション固有のパラメータを調整してドライバインタフェースを決定するコンフィギュレーションファイルのセットがあります。これらのコンフィギュレーションファイルには、コンポーネントクラスグループの **[Project (プロジェクト)]** ウィンドウからアクセスします。通常、ファイル名は *<コンポーネント名>_Config_0.c* または *<コンポーネント名>_Config_0.h* のようになります。

これらのファイルの設定の中には、ドライバおよびデバイスのコンフィギュレーションファイル (*RTE_Device.h*) に対応する設定が必要なものがあります。これについては次のセクションで説明します。

USB HID デバイスサンプルには、*USBD_Config_0.c* と *USBD_Config_HID_0.h* の 2 つのコンフィギュレーションファイルがあります。



*USBD_Config_0.c* ファイルには、特定の USB デバイス用の重要な設定が数多く含まれています。

- **[Connect to Hardware via Driver_USBD# (Driver_USBD# を介してハードウェアに接続)]** 設定では、ペリフェラルインタフェースを反映する *control struct* を指定します。この場合は、デバイスインタフェースとして使用する USB コントローラです。USB コントローラが 1 つしかないマイクロコントローラの場合、数字は「0」です。詳細については、「CMSIS ドライバ」セクション (52 ページ) を参照して下さい。
- USB コントローラでサポートされている場合は、**[High-Speed (高速)]** を選択します。この設定を使用するには、USB High-Speed 通信をサポートするドライバが必要です。
- **[Vendor ID (ベンダ ID)]** **(VID)** をプライベート VID に設定します。USB Implementer's Forum (http://www.usb.org/developers/vendor) には、有効なベンダ ID の適用方法に関する詳細な情報が記載されています。
- 各デバイスには固有の**製品 ID** が必要です。ホストコンピュータのオペレーティングシステムは、製品 ID を VID と共に使用して、デバイスに適したドライバを検出します。
- PC のオペレーティングシステムで USB デバイスを識別するために、**[Manufacturer (製造元)]** および **[Product String (製品文字列)]** を設定します。

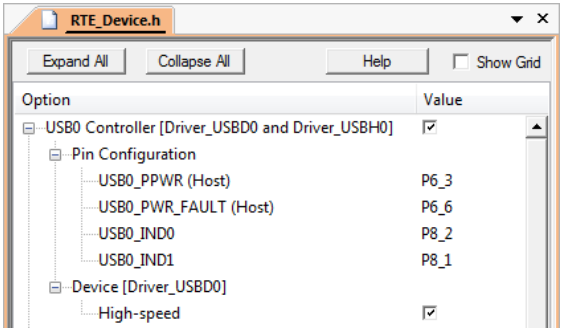
*USBD_Config_HID_0.h* ファイルには、デバイスクラス固有のエンドポイント設定が含まれています。このサンプルでは、変更する必要はありません。

# ドライバの設定

ドライバには、I/O ピン割り当て、クロック設定、DMA チャネルの使用法などの属性を定義する所定のプロパティがあります。多くのデバイスでは、このようなドライバプロパティは *RTE_Device.h* コンフィギュレーションファイルに含まれています。通常は、アプリケーションが使用する実際のペリフェラルインタフェースの設定が必要です。マイクロコントローラデバイスによっては、別のハードウェアペリフェラルを有効にしたり、ピン設定を指定したり、実装するクロック設定を変更したりできる場合があります。

USB HID デバイスサンプルは、以下のように設定する必要があります。

- **[USB0 Controller（USB0 コントローラ）]** をオンにして、このセクションを拡張します。
- 以下に示すように、**[Pin Configuration（ピン設定）]** を変更します。
- **Device:High-speed** をオンにします。



# システムリソースの調整

各ミドルウェアコンポーネントには所定のメモリおよび RTOS リソースの要件があります。『MDK-Professional Middleware ユーザガイド』の「**リソース要件**」セクションには、各コンポーネントの要件が記載されています。


ほとんどのミドルウェアコンポーネントでは CMSIS-RTOS を使用します。要件をサポートするように RTOS が設定されていることが重要です。

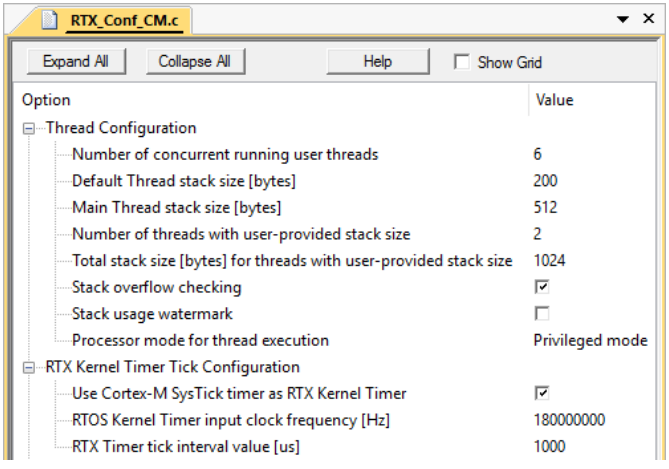
CMSIS-RTOS RTX では、*RTX_Conf_CM.c* ファイルによってスレッドおよびスタックが設定されます。詳細については、「**CMSIS-RTOS RTX コンフィギュレーション**」セクション（40 ページ）を参照して下さい。

USB HID デバイスサンプルでは、以下の設定が適用されます。

- **::USB:Device** コンポーネントには、USBDn_CoreThread という 1 つのスレッドとユーザが指定した 512 バイトのスタックが必要です。
- また、**::USB:Device:HID** コンポーネントには、USBD_HIDn_Thread という 1 つのスレッドとユーザが指定した 512 バイトのスタックが必要です。

これらの要件を *RTX_Conf_CM.c* ファイルの設定に反映させます。

- **[Number of concurrent running threads（同時実行中のスレッド数）]**：USB デバイスコンポーネントの 2 つのスレッドを同時に実行するには 6（デフォルト）で十分です。ユーザアプリケーションでさらに多くのスレッドを実行する場合は、この設定を調整します。
- **[Default Thread stack size [bytes]（デフォルトのスレッドのスタックサイズ［バイト］）]**：USB コンポーネントはユーザが指定したスタックで実行されるため、この設定は重要ではありません。
- **[Main Thread stack size [bytes]（メインスレッドスタックサイズ［バイト］）]**：値は 512。USB デバイスコンポーネントを初期化する API 呼び出しにはスタックが必要です。
- **[Number of threads with user-provided stack size（ユーザが指定したスタックサイズのスレッド数）]**：値は 2。ユーザが指定したスタックを持つ
2 つのスレッド（**::USB:Device** および **::USB:Device:HID** 用）を指定します。
- **[Total stack size [bytes] for threads with user-provided stack size（ユーザが指定したスタックサイズのスレッドの合計スタックサイズ［バイト］）]**：値は 1024。2 つのスレッドの合計スタックサイズを指定します。
- **[Timer Clock value [Hz]（タイマクロック値 [Hz]）]** はシステムクロック（180000000）と一致する必要があります。

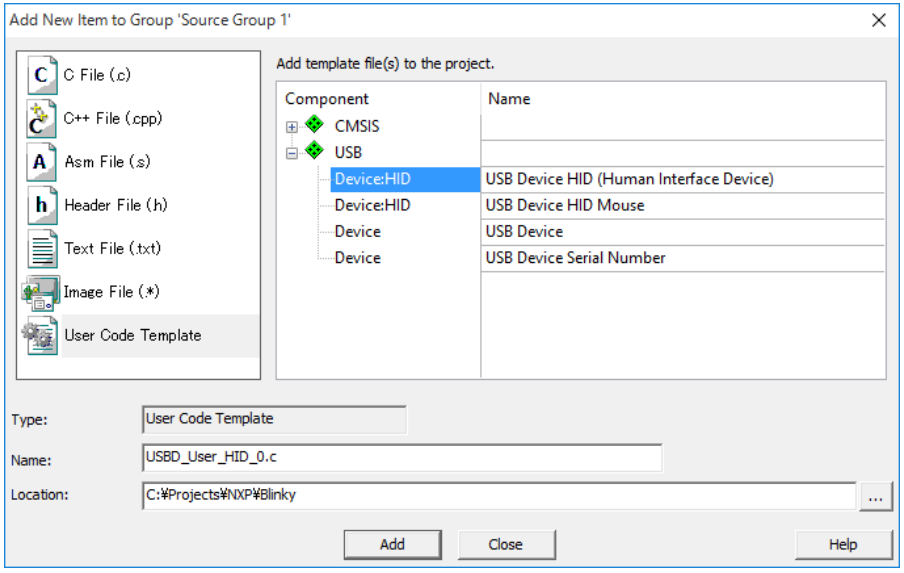


# アプリケーション機能の実装

アプリケーション固有の機能を実装するコードを作成します。ここには、**CMSIS-RTOS RTX を使用した Blinky** プロジェクト（61 ページ）で最初に作成された *main.c*、*LED.c*、および *LED.h* ファイルに対する変更が含まれています。

このミドルウェアは、アプリケーションソフトウェアの基礎として、**ユーザコードテンプレート**を提供します。

☞ **[Project（プロジェクト）]** ウィンドウで、**[Source Group 1（ソースグループ 1）]** を右クリックし、**[Add New Item to Group（新規項目をグループに追加）]** ダイアログを開きます。**::USB:Device:HID - [USB Device HID (Human Interface Device)（USB デバイス HID（ヒューマンインタフェースデバイス））]** でユーザコードテンプレートを選択し、**[Add（追加）]** をクリックします。




PC USB アプリケーションをマイクロコントローラデバイスに接続するには、*USBD_HID0_SetReport()* 関数を変更します。この関数は USB ホストのデータを処理します。例えば、データは **HIDClient.exe** ユーティリティを使用して作成されます。

エディタで *USBD_User_HID_0.c* ファイルを開き、以下のようにコードを変更します。こうすることによって、評価ボードの LED を制御します。

```
#include "LED.h"                        // access functions to LEDs
 :
bool USBD_HID0_SetReport (uint8_t rtype, uint8_t req, uint8_t rid,
                          const uint8_t *buf, int32_t len) {
  uint8_t i;

  switch (rtype) {
    case HID_REPORT_OUTPUT:
      for (i = 0; i < 4; i++) {
        if (*buf & (1 << i)) LED_On (i);
        else                 LED_Off (i);
      }
      break;

    case HID_REPORT_FEATURE:
      break;
  }
  return true;
}
```

*LED.c* ファイルで関数を拡張してボード上の複数の LED を制御し、LED を点滅させるスレッドはもう必要ないため削除します。

☞ エディタで *LED.c* ファイルを開き、以下のようにコードを変更します。

```
/*----------------------------------------------------------------------
 * File LED.c
 *----------------------------------------------------------------------*/
#include "SCU_LPC18xx.h"
#include "GPIO_LPC18xx.h"
#include "cmsis_os.h"                   // ARM::CMSIS:RTOS:Keil RTX

const GPIO_ID LED_GPIO[] = {            // LED GPIO definitions
  { 6, 24 },
  { 6, 25 },
  { 6, 26 },
  { 6, 27 }
};

void LED_Initialize (void) {
  GPIO_PortClock     (1);               // Enable GPIO clock

  /* Configure pin: Output Mode with Pull-down resistors */
  SCU_PinConfigure (13, 10, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 24, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 24, 0);
  SCU_PinConfigure (13, 11, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 25, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 25, 0);
  SCU_PinConfigure (13, 12, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 26, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 26, 0);
  SCU_PinConfigure (13, 13, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 27, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 27, 0);
}

void LED_On (uint32_t num) {
  GPIO_PinWrite     (LED_GPIO[num].port, LED_GPIO[num].num, 1);
}

void LED_Off (uint32_t num) {
  GPIO_PinWrite     (LED_GPIO[num].port, LED_GPIO [num].num, 0);
}
```

☞ エディタで *LED.h* ファイルを開き、*LED.c* に対して行った変更と一致するように変更します。*LED_On()* および *LED_Off()* 関数にはパラメータがあります。

```
/*----------------------------------------------------------------------
 * File LED.h
 *----------------------------------------------------------------------*/
void LED_Initialize ( void );
void LED_On ( uint32_t num );
void LED_Off ( uint32_t num );
```

☞ *main.c* ファイルを以下のように変更します。LED を点滅させるスレッドを開始させずに、USB デバイスコンポーネントを初期化して開始するコードを追加します。詳細については、『ミドルウェアユーザガイド』を参照して下さい。

```
/*----------------------------------------------------------------------
 * File main.c
 *----------------------------------------------------------------------*/
#define osObjectsPublic                 // define objects in main module
#include "osObjects.h"                  // RTOS object definitions
#include "LPC18xx.h"                    // Device header
#include "LED.h"                        // Initialize and set GPIO Port
#include "rl_usb.h"                     // Keil.MDK-Pro::USB:CORE

/*
 * main: initialize and start the system
 */
int main (void) {
  osKernelInitialize ();         // Initialize CMSIS-RTOS

  // initialize peripherals here
  LED_Initialize ();             // Initialize LEDs

  USBD_Initialize (0);           // USB Device 0 Initialization
  USBD_Connect (0);              // USB Device 0 Connect

  osKernelStart ();              // Start thread execution
  while (1);
}
```

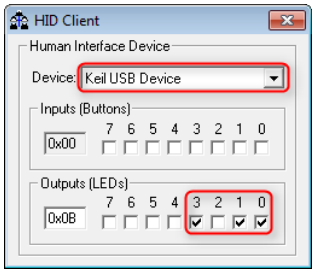
# ビルドとダウンロード

「**アプリケーションの作成**」の章（60 ページ）および「**デバッガの使用**」の章（84 ページ）の説明に従って、プロジェクトをビルドしてターゲットにダウンロードします。

# 検証とデバッグ

別の USB ケーブルを使用して、PC に開発ボードを接続します。これで、マイクロコントローラの USB デバイスペリフェラルに接続されます。ボードが接続されると、USB HID デバイスのデバイスドライバがインストールされたことを示す通知が表示されます。

MDK に含まれている *HIDClient.exe* ユーティリティプログラムにより、PC と開発ボード間の接続をテストできます。このユーティリティは、MDK インストールフォルダ *.¥Keil¥ARM¥Utilities¥HID_Client¥Release* にあります。

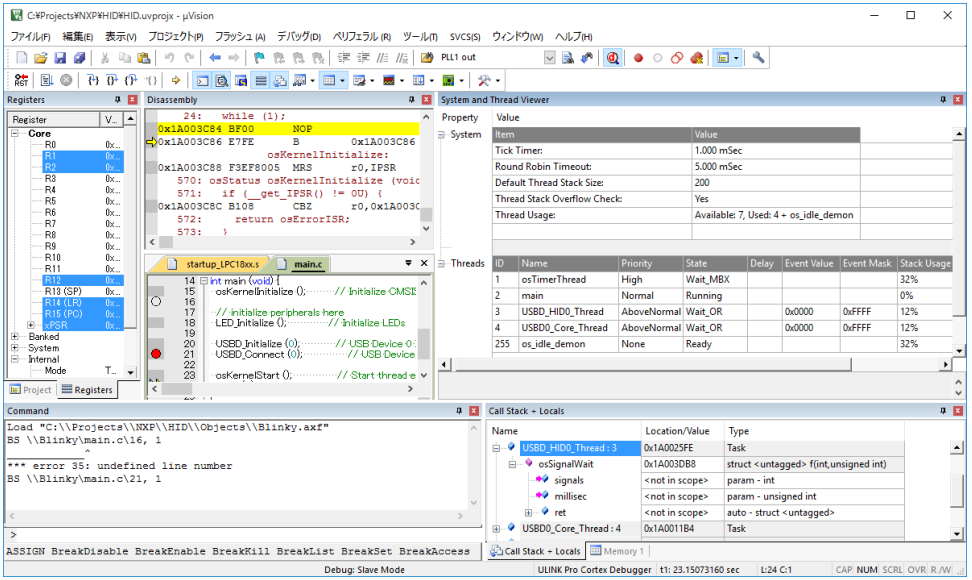



USB HID デバイスの機能をテストするには、*HIDClient.exe* ユーティリティを実行して、次の手順に従います。

- 通信チャネルを確立するデバイスを選択します。この例では、［*Keil USB Device (Keil USB デバイス)*］です。
- **［Outputs (LEDs) (出力 (LED) )］**チェックボックスを変更して、アプリケーションをテストします。開発ボードで、それぞれの LED が切り替わります。

開発ボードとの接続に問題がある場合は、デバッガを使用すると根本原因を明らかにできます。

ツールバーで**［Start/Stop Debug Session (デバッグセッションの開始/停止)］**を選択します。




デバッグウィンドウを使用して問題を絞り込みます。例えば、［Call Stack + Locals（コールスタックとローカル）］ウィンドウを使用して、*USBD_User_HID_0.c* ファイルのローカル変数の値を調べます。ブレークポイントは、コードの所定の行で停止して変数の内容を調べるときに役立ちます。

---

*注*

*通信プロトコルのデバッグは難しい場合があります。デバッガを開始したりブレークポイントを使用したりすると、通信プロトコルのタイムアウト時間を超える場合があるため、アプリケーションのデバッグが難しくなります。ブレークポイントは慎重に使用して下さい。*

*USB接続に失敗した場合は、USB 接続を切断して、ターゲットハードウェアをリセットし、アプリケーションを実行してから、もう一度 PC に接続して下さい。*

---

# 索引

## [



## B



## C



## D



## H



## I



## M



## P



## R

## U



## ア



## オ



## ク



## グ



## サ

## ソ



## タ



## デ

## ト



## ネ



## バ



## ビ

## フ



## ブ



## マ



## ミ



## メ



## ユ



## 学



## 従

## 新