STM32CubeIDE ユーザ·ガイド

概要

STM32CubeIDE は、オールインワンのマルチ OS 開発ツールであり STM32Cube ソフトウェア·エコシステムの一部を構成します。STM32 ベースの製品のソフトウェア開発を支援するための高度な C/C++ 開発プラットフォームが含まれます。

本書は、STM32CubeIDE の機能と使用法を解説するもので、起動、プロジェクトの作成とビルド、標準的な手法と高度な手法によるデバッグの方法、その他多くのソフトウェア解析ソリューションについて紹介します。STM32CubeIDE は、Eclipse C/C++ 開発ツール™（CDT™）および GCC ツールチェーンに基づいています。これらについては、本ユーザ·マニュアルで、すべてを説明することはできません。Eclipse® に関する詳細情報は、STM32CubeIDE の内蔵ヘルプ システムから入手できます。ツールチェーンと GDB サーバに関する詳細が記載された特別なドキュメントが製品に同梱されています。

1 はじめに

STM32CubeIDE は、Arm® Cortex® プロセッサに基づく STM32 製品に対応しています。詳細については、セクション 11 参考文献に示した ST マイクロエレクトロニクスのドキュメントを参照してください。

注 Arm は、米国内およびその他の地域にある Arm Limited 社（またはその子会社）の登録商標です。

1.1 製品情報

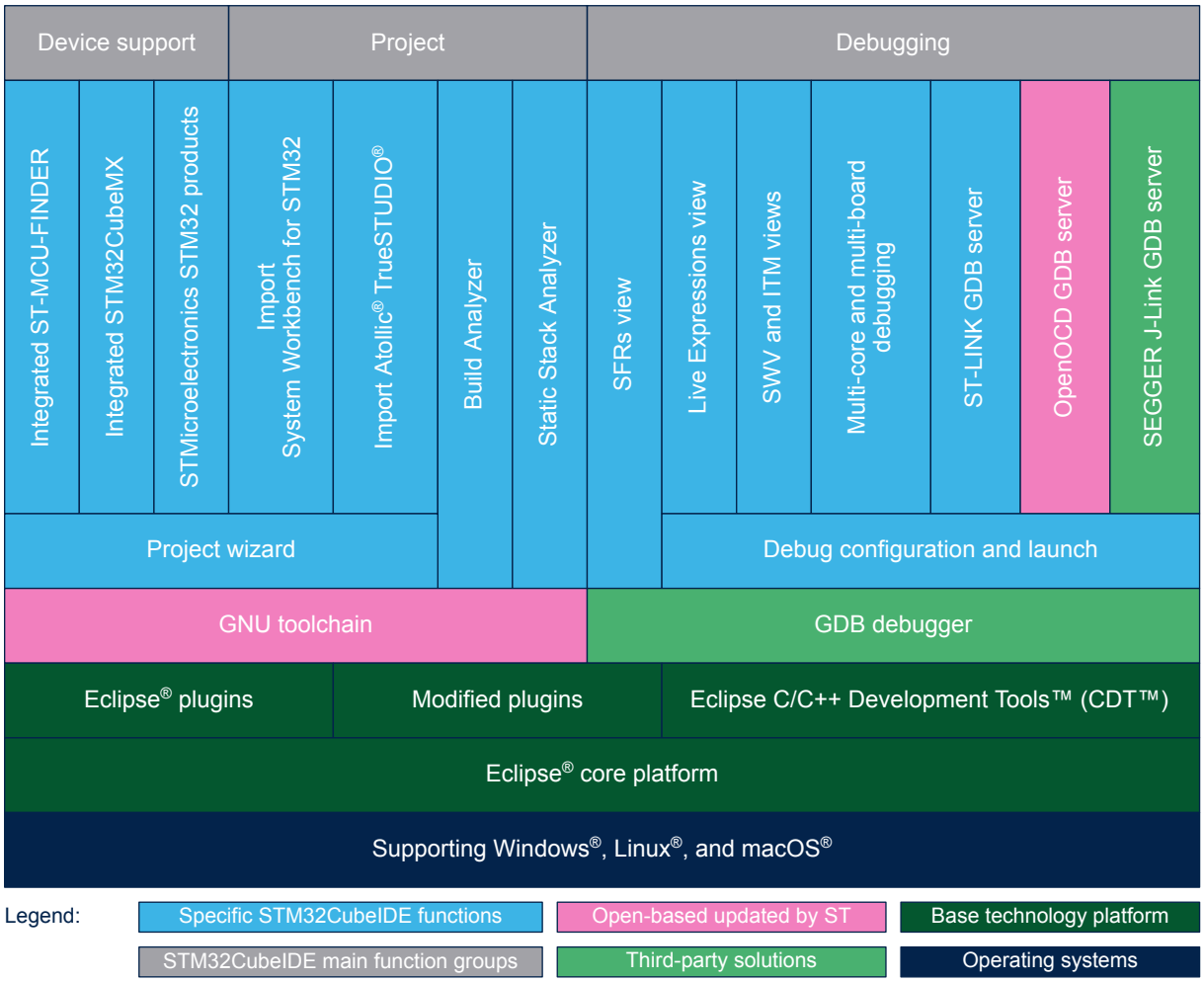
STM32CubeIDE は、ペリフェラル設定、コード生成、コード・コンパイル、リンク、デバッグの機能を備えた高度な C/C++ 開発プラットフォームです。Eclipse®/CDT™ フレームワークと開発用の GCC ツールチェーン、デバッグ用の GDB に基づいています。Eclipse® IDE の機能を補完する、数百に及ぶ既存のプラグインを組み込むことができます。

STM32CubeIDE は、ST MCUFinder（ST-MCU-FINDER-PC）と STM32CubeMX の機能を統合して、オールインワンのツール・エクスペリエンスを提供します。付属の GCC ツールチェーンを使用すれば、簡単に STM32 マイクロコントローラまたはボードのプロジェクトを新規作成し、ビルドすることができます。

STM32CubeIDE には、プロジェクトの状態やメモリ要件に関する有用な情報を提供する、Build Analyzer と Static Stack Analyzer が付属しています。

STM32CubeIDE は、標準および高度なデバッグ機能も内蔵しており、CPU コア・レジスタ、メモリ、ペリフェラル・レジスタの内容表示、ライブ変数ウォッチ、シリアル・ワイヤ・ビューア・インタフェースなどを備えています。Fault Analyzer は、デバッグ・セッション中に STM32 プロセッサがエラーをトリガすると、そのエラーに関する情報を表示します。

図 1. STM32CubeIDE の特徴



1.1.1 システム要件

STM32CubeIDE は、Microsoft 社®、Windows®、Linux®、macOS® の各オペレーティング·システムによって動作をテスト、検証済みです。

重要 STM32CubeIDE は、64 bit 版の OS にのみ対応しています。オペレーティング·システムのサポート対象バージョンの詳細については、[ST-02] を参照してください。

注 macOS® は、米国内およびその他の国々で登録された、Apple Inc. 社の商標です。

Linux® は Linus Torvalds 氏の登録商標です。

1.1.2 STM32CubeIDE 最新バージョンのダウンロード

STM32CubeIDE の最新バージョンは、Web サイト www.st.com/stm32softwaretools から無償でダウンロードできます。

1.1.3 STM32CubeIDE のインストール

STM32CubeIDE のインストール·ガイド [ST-04] には、Windows®、Linux®、macOS® のサポート対象バージョンへのインストール手順が記載されています。STM32CubeIDE は、複数のバージョンを同時にインストールできます。まだ STM32CubeIDE をインストールしていない場合、または新しいバージョンをインストールする必要がある場合は、インストール·ガイドをお読みください。更新や Eclipse 追加プラグインのインストール 更新のインストール方法については、このマニュアルの「更新およびその他の Eclipse プラグインのインストール」でも説明しています。

1.1.4 ライセンス

STM32CubeIDE は、Mix Ultimate Liberty+OSS+3rd パーティ V1 ソフトウェア使用許諾契約（SLA0048）の下で提供されます。

各コンポーネントの使用許諾契約の詳細については、[ST-02] を参照してください。

1.1.5 サポート

ST マイクロエレクトロニクスが提供するサポートには、さまざまな種類の選択肢があります。例えば、ST Community は、同じような興味を抱く世界中のユーザと、いつでもコミュニケーションを取ることができる場を提供しています。サポートの選択肢は、my.st.com/content/my_st_com/en/support/support-home.html にアクセスすることで、お選びいただけます。

1.2 STM32CubeIDE の使用

1.2.1 基本概念と用語

このセクションでは、STM32CubeIDE 使用の基本概念と Eclipse® 関連の用語について概説します。

ワークスペース

STM32CubeIDE を起動すると、ワークスペースが選択されます。ワークスペースには使用する開発環境が含まれます。技術的には、ワークスペースとはプロジェクトを保持するディレクトリです。ユーザはアクティブなワークスペース内の任意のプロジェクトにアクセスできます。

プロジェクトにはファイルが含まれ、サブディレクトリを使用して体系化できます。コンピュータ内のどこか別の場所にあるファイルについても、プロジェクトへのリンクを設定できます。

1 台のコンピュータで、ファイル·システム内のさまざまな場所に複数のワークスペースを保持できます。ワークスペース間の切り換えは可能ですが、一度にアクティブ化できるワークスペースは 1 つだけです。ワークスペースの切り換えは、異なるプロジェクト間をすばやく行き来する方法です。

実際には、ワークスペースとプロジェクトを使用する方式により、適切な階層構造、つまりワークスペースにプロジェクトが含まれ、さらにそのプロジェクトにファイルが含まれるという構造を簡単に構成できます。

Information Center

STM32CubeIDE をはじめて起動し、ワークスペースを選択すると、Information Center が表示されます。Information Center は、新しいプロジェクトを開始したり、STM32CubeIDE のドキュメントを読んだり、ST のサポートやコミュニティにアクセスしたりするためのクイック·アクセスを提供しています。Information Center には、[Information Center]ツールバー·ボタンまたは[Help]メニューからいつでも簡単にアクセスできます。

パースペクティブ、メニュー・バー、ツールバー

Information Center を閉じると、STM32CubeIDE は、メニュー・バー、ツールバー、ビュー、エディタからなるパースペクティブを表示します。各パースペクティブは、特定の作業の種類ごとに最適化されています。例えば、C/C++ パースペクティブは、プロジェクトの作成、編集、ビルドを目的としたものです。デバッグ・パースペクティブは、ハードウェア上のコードをデバッグするときに使用します。

各パースペクティブは、ユーザのニーズに応じてカスタマイズが可能です。また、開いたビューが多すぎる場合や、ビューの並び順が変更された場合などには、いつでもパースペクティブをリセットできます。パースペクティブを新たに作成することも可能です。

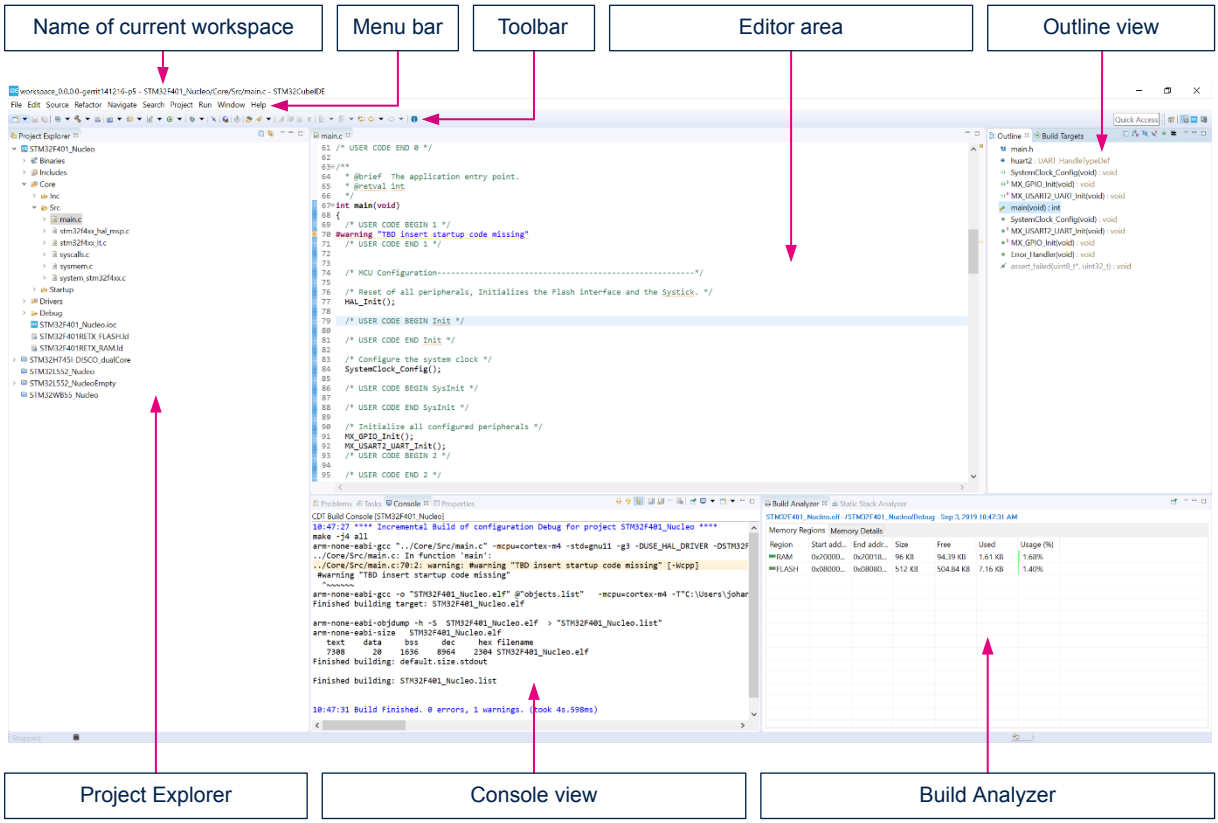
ビューとエディタ

通常、パースペクティブには多数のビューが表示されます。各ビューは、特定の情報を提供するために作成されます。プロジェクトやデバッグ対象の組込みシステムから収集した情報などです。

パースペクティブにはエディタ領域が 1 つ表示されます。このエディタは、プロジェクト・ファイルの編集に使用できます。エディタ内の異なるタブで複数のファイルの編集が可能です。

STM32CubeIDE のウィンドウ

図 2. STM32CubeIDE のウィンドウ



1.2.2 STM32CubeIDE の起動

使用するオペレーティング・システムに応じて、以下の手順を実行することで STM32CubeIDE を起動します。

Windows®

製品のインストール時にデスクトップ・ショートカットを作成した場合、これを使用して STM32CubeIDE を起動できます。Windows® のスタート・メニューにある ST マイクロエレクトロニクスのプログラムから起動することも可能です。

それ以外の場合は、

1. STM32CubeIDE がインストールされている場所を見つけます（例：`C:\ST\STM32CubeIDE_1.0.2`）。

2. STM32CubeIDE フォルダを開きます。
3. `stm32cubeide.exe` プログラムを起動します。

Linux® または macOS®

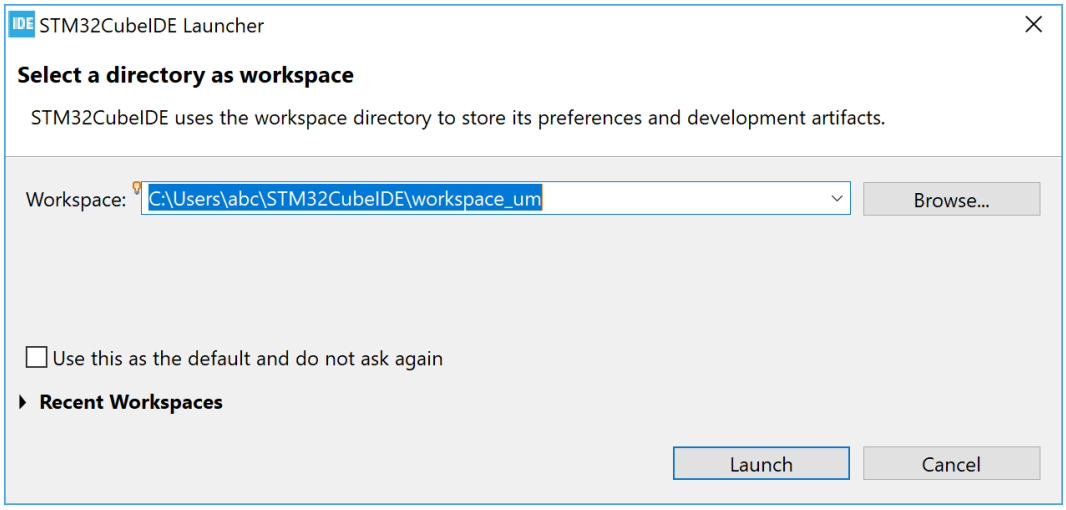
Linux® または macOS® を使用する場合、製品がインストールされている場所の STM32CubeIDE フォルダを開き、同様の方法でプログラムを起動できます。

STM32CubeIDE Launcher

製品を起動すると、ワークスペースを選択する [STM32CubeIDE Launcher]ダイアログが表示されます。製品の初回起動時には、デフォルトの保存場所とワークスペース名が表示されます。このダイアログでは、現在ユーザがアクセスできるすべてのプロジェクトを保持する、アクティブなワークスペースの名前と場所を選択できます。新規作成するプロジェクトは、すべてこのワークスペース内に保存されます。ワークスペースがまだ存在しない場合は、作成されます。

注 Windows® を使用する場合、Windows® のパスの文字数制限を超えないように、ワークスペースのフォルダをファイル・システムのルートから何階層も下がった深い場所に置くことは避けてください。Windows® が処理できるファイル・パスの文字数を超えるとビルド・エラーが発生します。

図 3. STM32CubeIDE Launcher - ワークスペースの選択

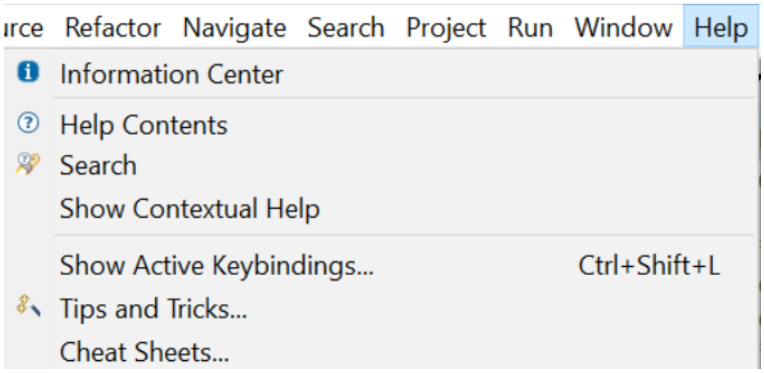


Launch ボタンをクリックして STM32CubeIDE を起動します。初回起動時には Information Center が表示されます。この画面の説明は、セクション 1.3 Information Center を参照してください。

1.2.3 ヘルプ・システム

[Help]メニューからは、図 4 に示すように、各種ヘルプ・システムを選択できます。[Information Center]には、入手可能な STM32CubeIDE ドキュメントのすべてにアクセスできるリンクが表示されます。この画面は、Eclipse® の基本を押さえたい初心者が Eclipse® の各種内蔵ヘルプ・システムを試すのに適した場所です。

図 4. [Help]メニュー



1.3 Information Center

Information Center は、以下の機能を実行するためのクイック·アクセスを提供しています。

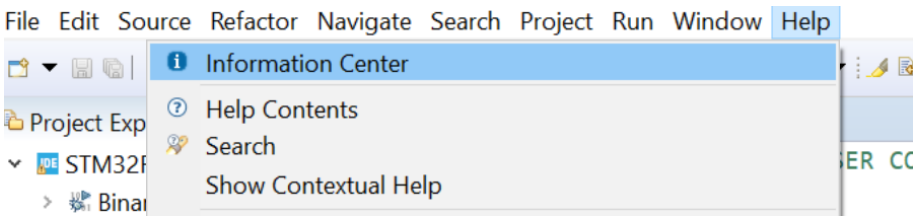
1. 新しいプロジェクトを開始する
2. 既存のプロジェクトをインポートする
3. STM32CubeIDE のドキュメントを読む
4. Getting Started with STM32CubeIDE（STM32CubeIDE クイック·スタート·ガイド [ST-03]）にアクセスする
5. STM32 MPU およびマイクロコントローラの wiki を参照する
6. STM32CubeIDE の新機能紹介（STM32CubeIDE のリリース·ノート [ST-02]）を参照する
7. Twitter™、Facebook™、YouTube™ または ST Community（community.st.com）にアクセスして ST マイクロエレクトロニクスのサポートやコミュニティを利用する
8. ST マイクロエレクトロニクスのアプリケーション·ツールについて調べる

製品を使い始める前に、すべての資料を読む必要はありません。むしろ、Information Center をリファレンス情報集と考え、必要になったときに読み返すことをお勧めします。

1.3.1 Information Center へのアクセス

Information Center には、どのパースペクティブからも、Information Center ツールバー·ボタン  を使用することで、いつでも簡単にアクセスできます。このアイコンは、ツールバーの右側にあります。Information Center は、メニュー·コマンド HelpInformation Center から開くことも可能です。

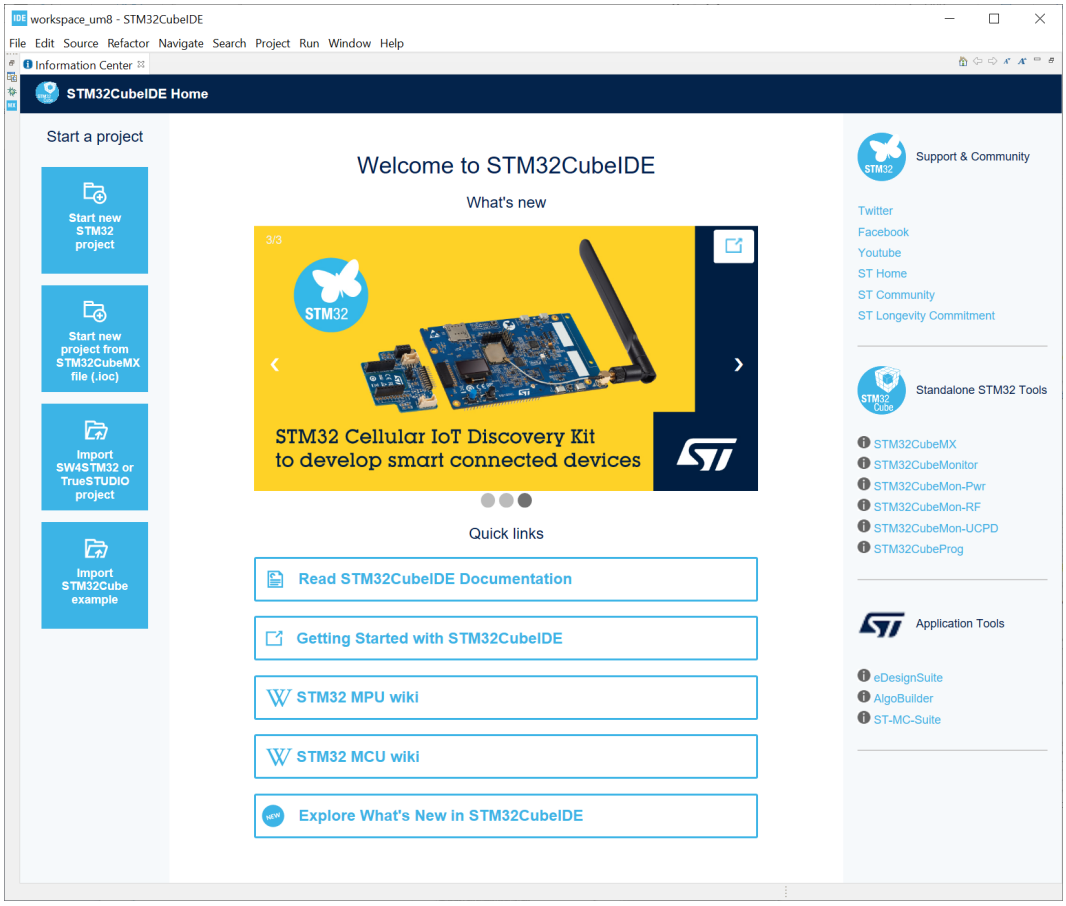
図 5. [Help] - [Information Center]メニュー



1.3.2 [Home]ページ

Information Center を開くと、[Home]ページが表示されます。ここには、新規プロジェクトの開始、プロジェクトのインポート、ドキュメントの参照、ST マイクロエレクトロニクスのサポートやコミュニティへのアクセスに使用するリンクがあります。

図 6. Information Center - [Home]ページ

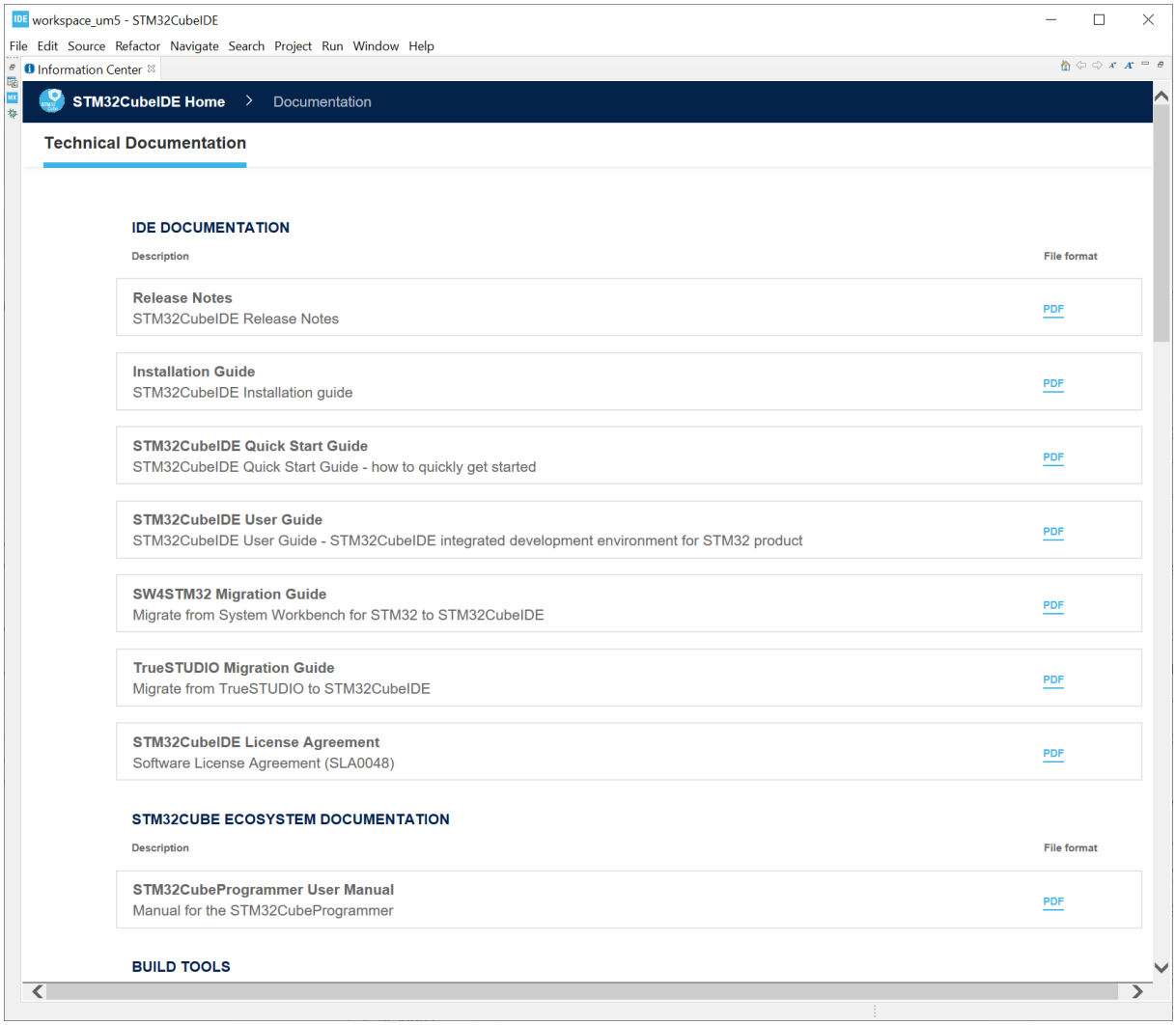


古いワークスペースを使用すると、Information Center に「This page can't be displayed（このページは表示できません）」と表示されたり、ドキュメントにアクセスしたときに古いマニュアルが開いたりして、有効な情報が得られないことがあります。そのような場合は、[Information Center]ウィンドウ右上隅にある Home ボタン をクリックしてページをリロードしてください。

1.3.3 Technical Documentation

Information Center には、[Technical Documentation]ページもあります。このページは、[Home]ページのリンク[Read STM32CubeIDE Documentation]をクリックすると開きます。

図 7. Information Center - [Technical Documentation] ページ



[Technical Documentation]ページをスクロールして、リスト内のドキュメントをクリックして開きます。ドキュメントのリストは、次のようにグループ化されています。

- IDE に関するドキュメント(IDE DOCUMENTATION)
- STM32Cube エコシステムに関するドキュメント(STM32CUBE ECOSYSTEM DOCUMENTATION)
- ビルド·ツール(BUILD TOOLS)
- デバッガに関するドキュメント(DEBUGGER DOCUMENTATION)
- ツールチェーンのマニュアル(GNU Tools for STM32)(TOOLCHAIN MANUALS)

注 STM32CubeIDE は、GNU Tools for STM32 ツールチェーンとともに配布されます。このツールチェーンは、GNU ARM Embedded ツールチェーンにパッチを適用して強化したものです。STM32CubeIDE の最初のバージョンには、GNU ARM Embedded ツールチェーンも含まれていました。これは STM32CubeIDE の更新サイトからインストールできます。その他のツールチェーンをインストールした場合、それらに対応するドキュメントも Information Center の[Technical Documentation]ページのリストに表示されます。GNU Tools for STM32 に適用したパッチに関する情報は、を参照してください。ドキュメントは Information Center の[Technical Documentation]ページから開くことができます。

[Home]ページに戻るには、Information Center の左上にある[STM32CubeIDE Home]をクリックします。

1.3.4 Information Center を閉じる

Information Center を閉じるには、[Information Center]タブを閉じます。

1.4 パースペクティブ、エディタ、ビュー

STM32CubeIDE は、さまざまな機能を搭載したビューを多数備えた強力な製品です。すべてのビューを同時に表示すると、現在手がけている作業とは必ずしも関係ない情報によって、ユーザに過大な負担がかかる場合があります。

そのような状況を回避するために、ビューをパースペクティブ内で整理できます。パースペクティブは、デフォルトで表示される数々の定義済みビューやエディタ領域で構成されます。通常、1 つのパースペクティブでは 1 つの開発作業、例えば C/C++ コードの編集やデバッグを処理します。

1.4.1 パースペクティブ

パースペクティブは、ビューの配置やサイズの変更、新規ビューを開くなど、ユーザのニーズに応じてカスタマイズできます。また、開いたビューが多すぎる場合や、ビューの並び順が変更された場合などには、いつでもパースペクティブをリセットできます。パースペクティブをリセットするには、ツールバーのパースペクティブ·アイコンを右クリックして、リストから Reset を選択します。これによって、ビューがリセットされ、パースペクティブ内に追加したビューが閉じ、デフォルト·ビューは元の位置に戻ります。

図 8. パースペクティブのリセット

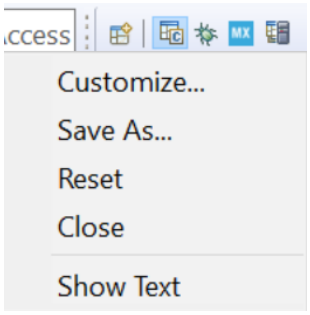
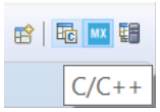


図 8 に示したように、パースペクティブはカスタマイズして、新しい名前で保存することも可能です。

パースペクティブ間の切り換えは、いくつかのビューを非表示にして、他のビューを表示させる簡単な方法です。パースペクティブを切り換えるには、ツールバーの右側にある Open Perspective ボタンを選択します。

図 9. パースペクティブを切り換えるツールバー·ボタン

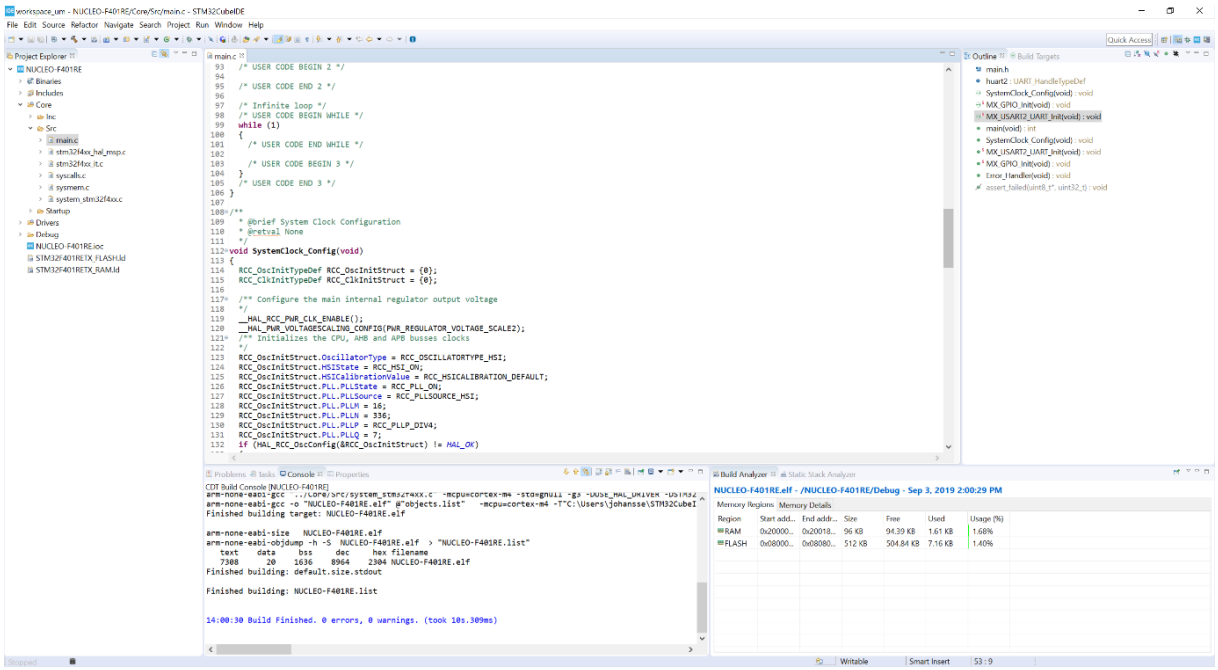


パースペクティブを切り換える、もう一つの方法はメニュー·コマンドの WindowPerspectiveOpen PerspectiveOther… から、使用するパースペクティブを選択します。

1.4.1.1 C/C++ パースペクティブ

C/C++ パースペクティブは、プロジェクトの新規作成、ファイルの編集、プロジェクトのビルドを目的としたものです。パースペクティブの左側には、[Project Explorer]ビューが表示されます。エディタは中央にあります。右側にはプロジェクト関連のいくつかのビュー([Outline]と[Build Targets]ビュー)が表示されます。図 10 に示した画面例の下部には、[Problems]、[Tasks]、[Console]、[Properties]の各ビューがあります。画面右の一番下には、[Build Analyzer]と[Static stack analyzer]ビューが表示されます。

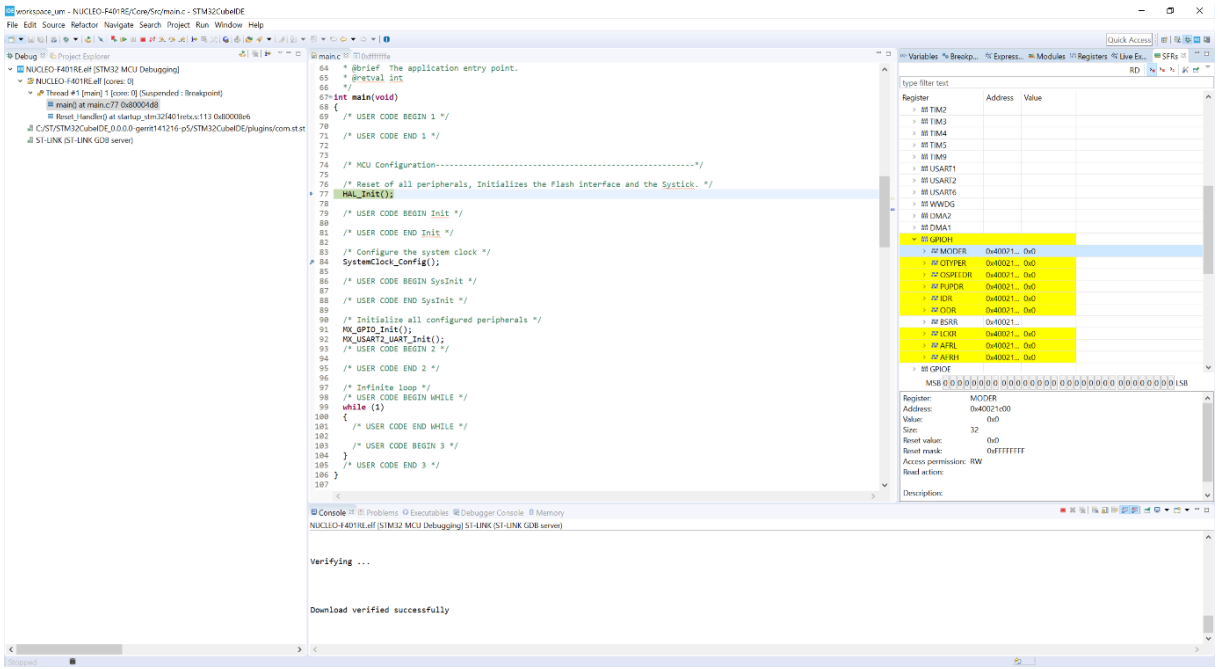
図 10. C/C++ パースペクティブ



1.4.1.2 *デバッグ・パースペクティブ*

デバッグ・パースペクティブは、コードのデバッグを目的としたものです。デバッグ・パースペクティブは通常、新しいデバッグ・セッションを開始すると自動的に開きます。その後、デバッグ・セッションを閉じると、このパースペクティブは元の C/C++ パースペクティブに切り換わります。

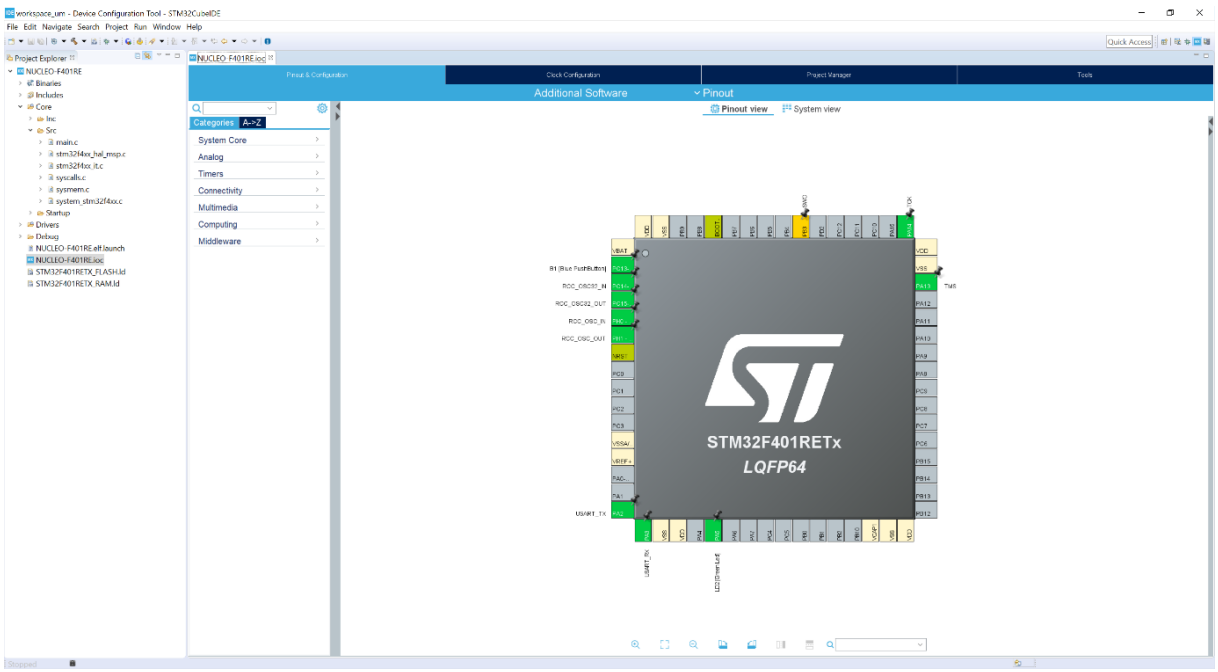
図 11. デバッグ・パースペクティブ



1.4.1.3 *デバイス設定ツール·パースペクティブ*

デバイス設定ツール·パースペクティブには、STM32CubeIDE に統合された STM32CubeMX デバイス設定ツールが含まれます。このパースペクティブは、デバイスの設定に使用されます。エディタで `*.ioc` ファイルを開き、デバイス設定ツール·パースペクティブを使用することで、このパースペクティブ内からデバイスを設定できます。デバイスの設定方法は、[ST-14] で説明しています。

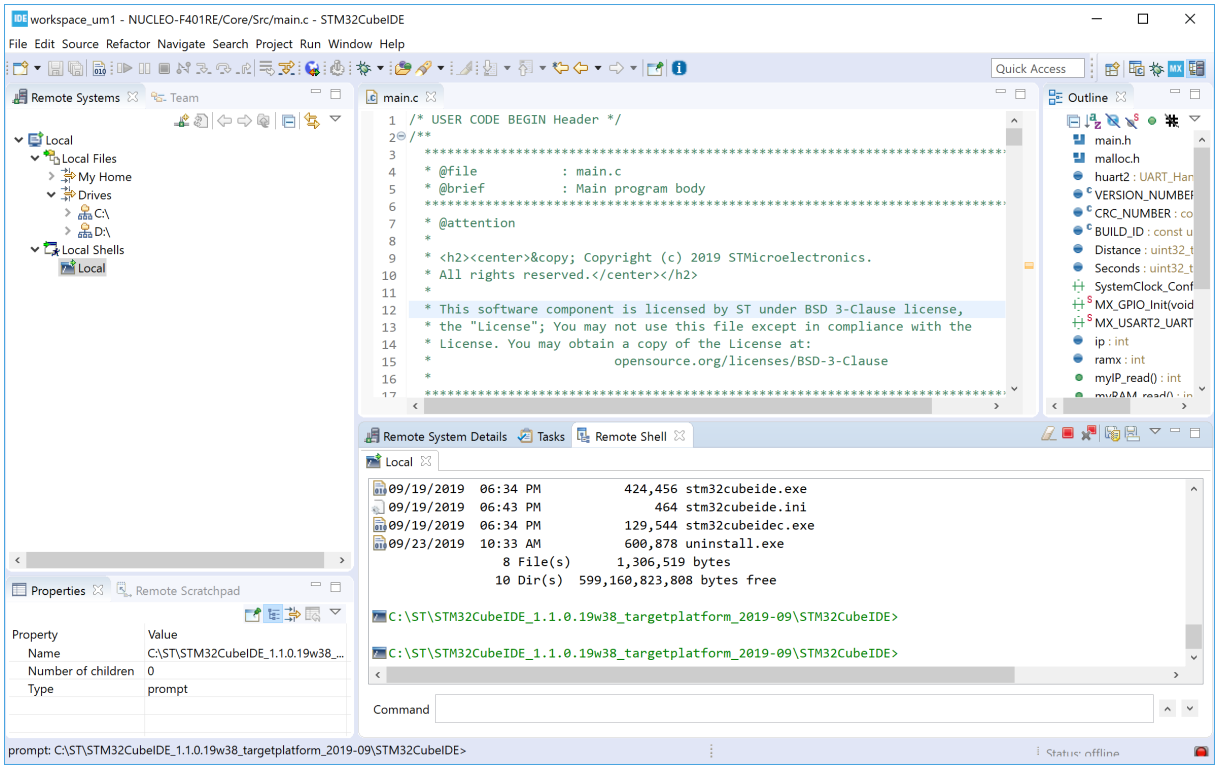
図 12. デバイス設定ツール·パースペクティブ



1.4.1.4 *リモート·システム·エクスプローラ·パースペクティブ*

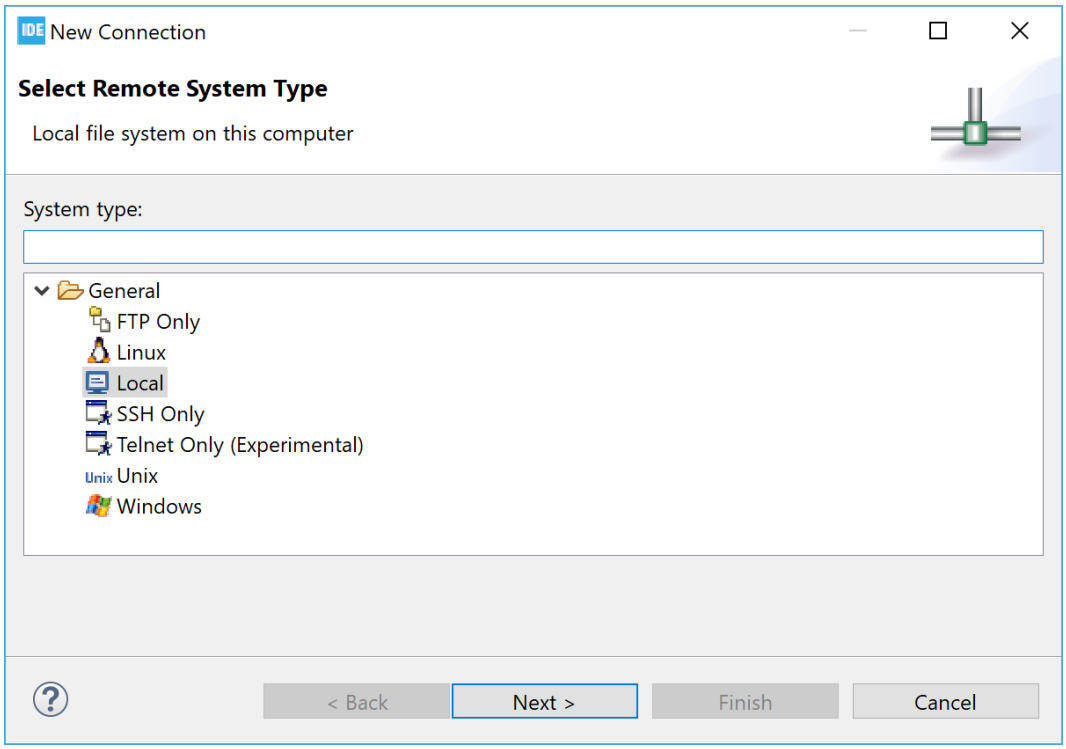
リモート·システム·エクスプローラ·パースペクティブは、基本的に STM32 Arm® Cortex® MPU ベースのシステムを開発するときに使用します。[Remote Systems]ビューはファイル表示に、[Remote Shell]ビューはコマンド実行に使用します。

図 13. リモート・システム・エクスプローラ・パースペクティブ



[Remote Systems]ビューには、FTP、Linux®、ローカル、SSH、Telnet、その他を介した新しい接続を開くボタンがあります。

図 14. 新規接続



1.4.2 エディタ

パースペクティブのエディタ領域は編集に使用します。エディタは同時にいくつでも開くことができますが、アクティブ化できるのは一度に 1 つだけです。各種ファイル拡張子に、異なるエディタを関連付けられます。エディタの例として、C エディタ、リンカ·スクリプト·エディタ、STM32CubeMX デバイスを設定するための .ioc ファイル·エディタなどがあります。

エディタでファイルを開くには、[Project Explorer]ビューでファイルをダブルクリックするか、File メニューを使用します。

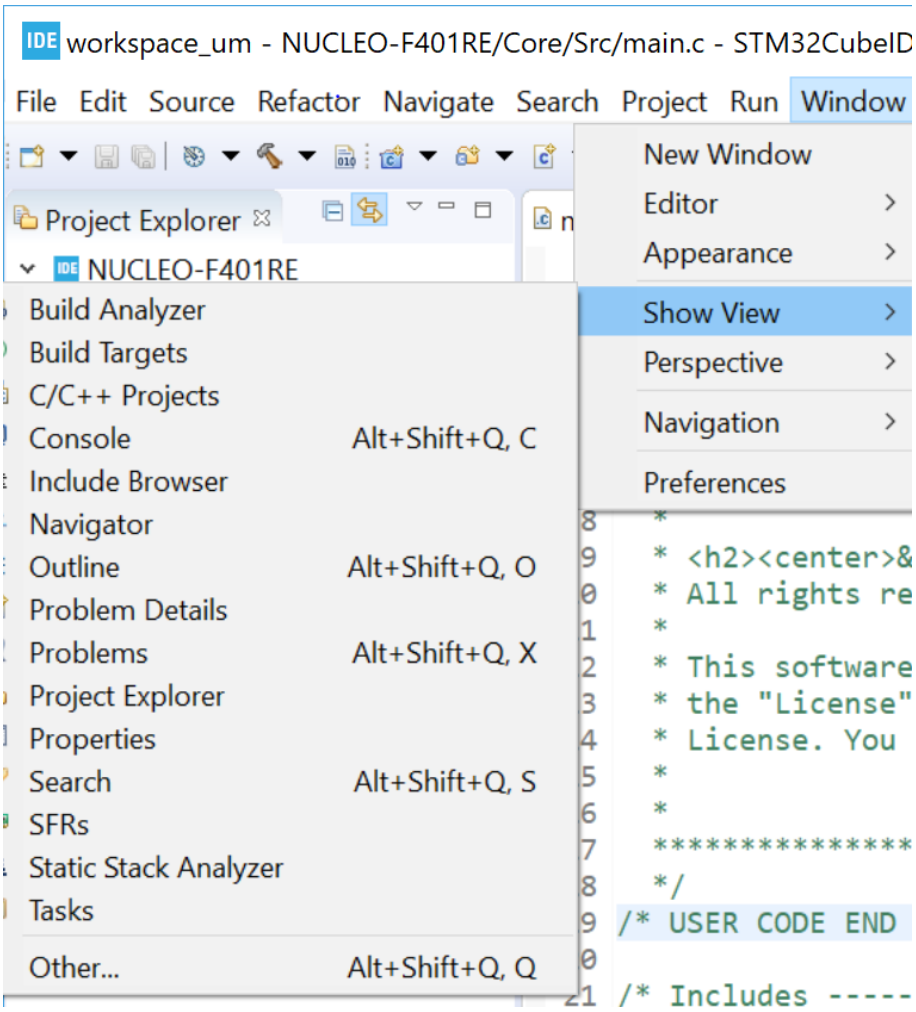
エディタでファイルを変更すると、ファイルに未保存の変更個所があることを示すアスタリスク(*)が表示されます。

1.4.3 ビュー

デフォルトでは、パースペクティブに関連する最も一般的なビューだけが表示されます。この製品には、さまざまな機能に対応する、より多くのビューがあります。それらのビューの中には、デバッグ·セッションの実行中にのみ有効なデータを表示するものもあれば、常時データを表示するものもあります。

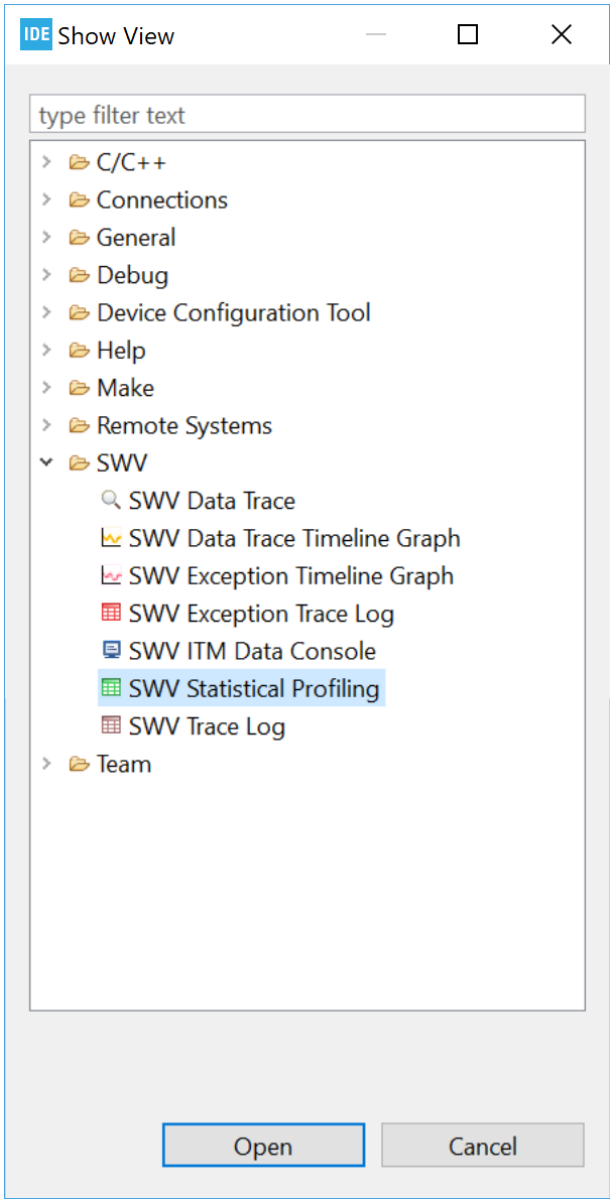
ビューは、WindowShow View メニューのリストから、いずれか 1 つを選択することで開くことができます。

図 15. Show View メニュー



ビューは、上記 図 15 に示したリストの他にもあります。このリストには、現在選択中のパースペクティブに関連する作業で使用する最も一般的なビューしか表示されません。ここに含まれないビューにアクセスするには、リストの Other... を選択します。これによって、[Show View]ダイアログ·ボックスが表示されます。いずれかのビューをダブルクリックして開き、そのビューに含まれるその他の機能にアクセスしてください。

図 16. [Show View]ダイアログ



ビューのサイズと位置は、次のようにして変更できます。単にビューを STM32CubeIDE 内の新しい位置までドラッグします。ビューを画面上の STM32CubeIDE ウィンドウの外にまでドラッグすることも可能です。このように切り離されたビューは、独立したウィンドウに表示されます。切り離されたビューは、他のビューと同じように機能しますが、常にワークベンチの前面に表示されます。切り離されたビューを再び結合するには、そのタブを STM32CubeIDE ウィンドウ内にドラッグします。

パースペクティブを元の状態に戻すには、ツールバーのパースペクティブ·アイコンを右クリックして、リストから Reset を選択します。パースペクティブをリセットするもう一つの方法は、メニュー WindowPerspectiveReset Perspective を使用します。

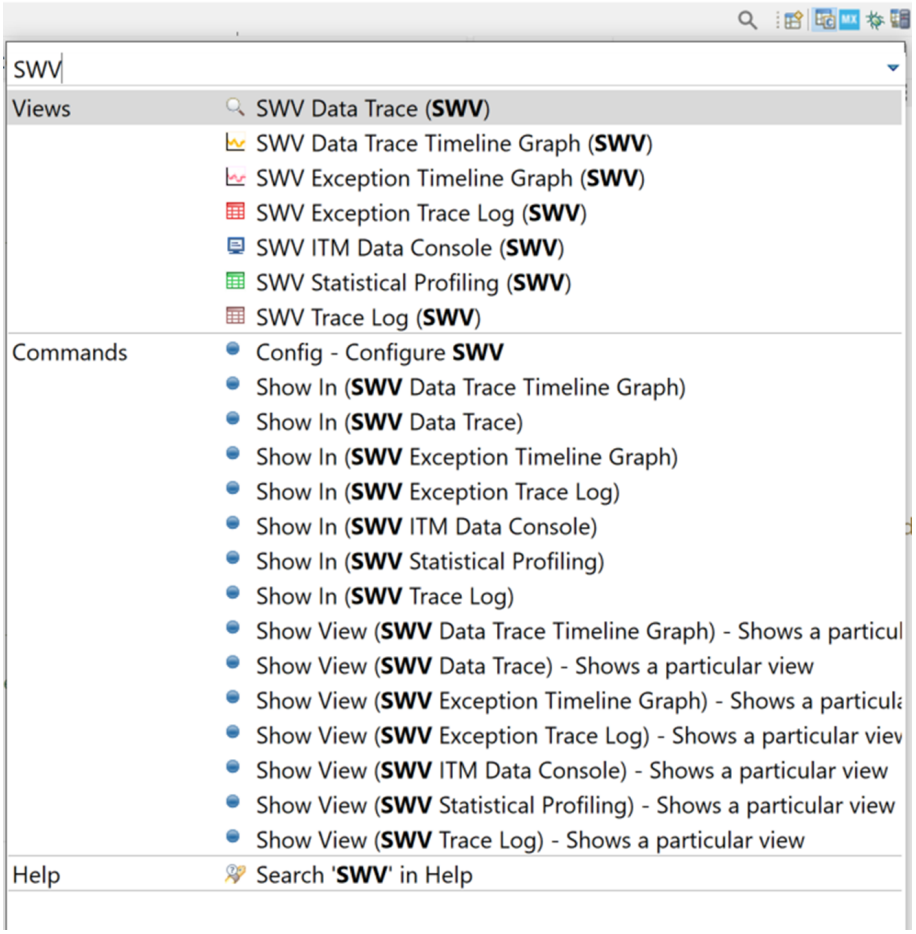
1.4.4 クイック·アクセス編集フィールド

ツールバーの虫眼鏡をクリックすると、クイック·アクセス·テキスト·ボックスが表示されます。このボックスには、検索する任意のフレーズやキーワードを入力できます。メニュー·コマンド、ツールバー·ボタン、プレファレンスの設定、ビューなどの GUI オブジェクトも、このテキスト·ボックスを使用して見つけることができます。何らかの検索文字列を入力すると、検索条件に合致するすべての GUI オブジェクトがクイック·アクセスによってリアルタイムで表示されます。いくつかの文字を入力するにつれて、検索結果が即時に絞り込まれます。

設定ダイアログの深い階層に埋もれているプレファレンス設定など、特定の GUI オブジェクトがなかなか見つからない場合は、クイック·アクセスを使用すると時間を節約できます。また、現在アクティブなパースペクティブでは非表示になっているメニュー·コマンドやツールバー·ボタンを取得する場合にも便利です。

例えば 図 17 のように、クイック·アクセスに検索文字列 SWV を入力すると、一致するビュー、GUI、コマンド、プレファレンス設定のリストが即座に表示されます。検索されたビューやプレファレンス設定を開くには、検索結果のリストから目的の GUI オブジェクトをクリックします。

図 17. クイック·アクセス

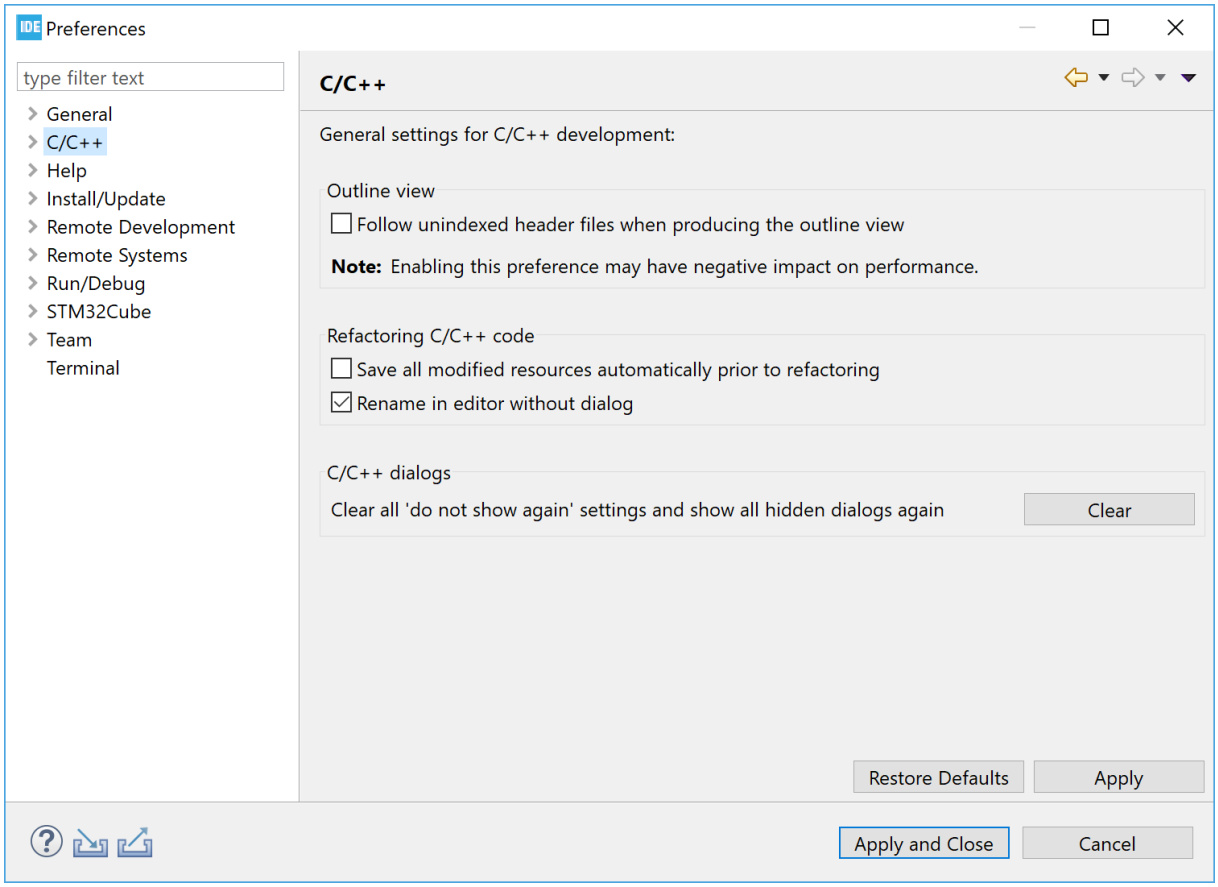


1.5 設定 - プレファレンス

STM32CubeIDE は、さまざまな方法でカスタマイズできます。メニュー WindowPreferences を使用して、[Preferences] ダイアログを開きます。このダイアログの左側ペインは、プレファレンスの特定のページに移動するために使用します。ここにはフィルタ·フィールドもあり、表示される内容を絞り込むことができます。ダイアログ右上の矢印を使用すると、ページ間を前後に移動できます。右側ペインには、表示されたプレファレンスの設定が表示されます。必要な何らかの変更を加えて、Apply をクリックすると設定が更新されます。

Restore Defaults は、すべての変更をリセットします。プレファレンス設定は、アプリケーションのワークスペース内のメタデータ·フォルダに保存されます。セクション 1.7 既存ワークスペースの管理 このユーザ·マニュアルの「セクション 1.7 既存ワークスペースの管理」では、プレファレンスのバックアップや、ワークスペース間でのコピーの方法を解説しています。

図 18. プレファレンス



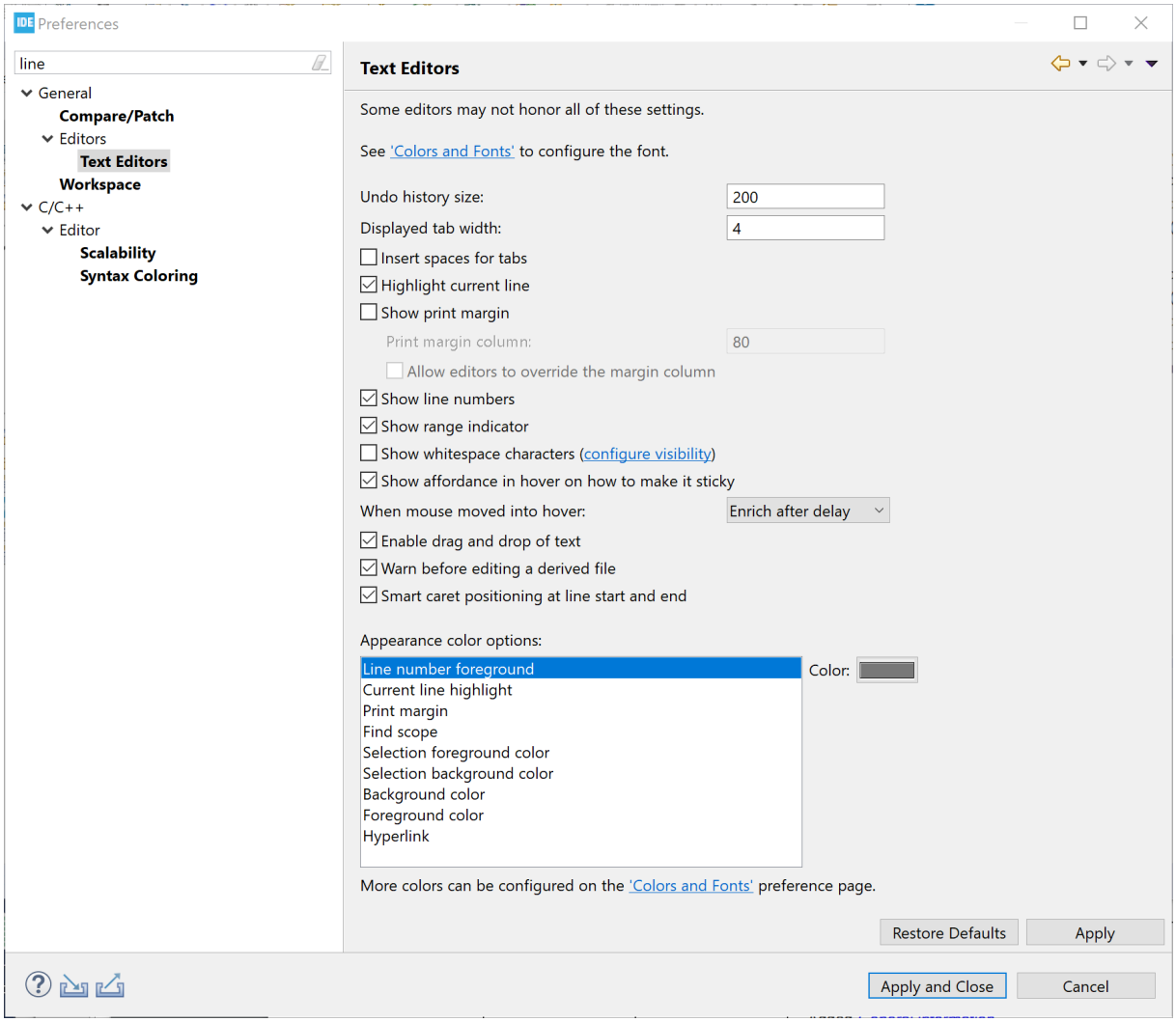
プレファレンスのページを順に一通り参照して、設定可能なオプションを把握することを推奨します。この後のセクションでは、その一部を紹介します。

1.5.1 プレファレンス - エディタ

エディタは、さまざまな設定が可能です。例えば、メニュー GeneralEditorsText Editors を選択すると、エディタの次のような一般的設定を含むプレファレンス·ペインが表示されます。

- 表示するタブの幅(Displayed tab width)
- タブに空白文字を挿入(Insert spaces for tabs)
- 現在の行をハイライト(Highlight current line)
- 行番号を表示(Show line numbers)
- その他

図 19. プレファレンス - テキスト･エディタ

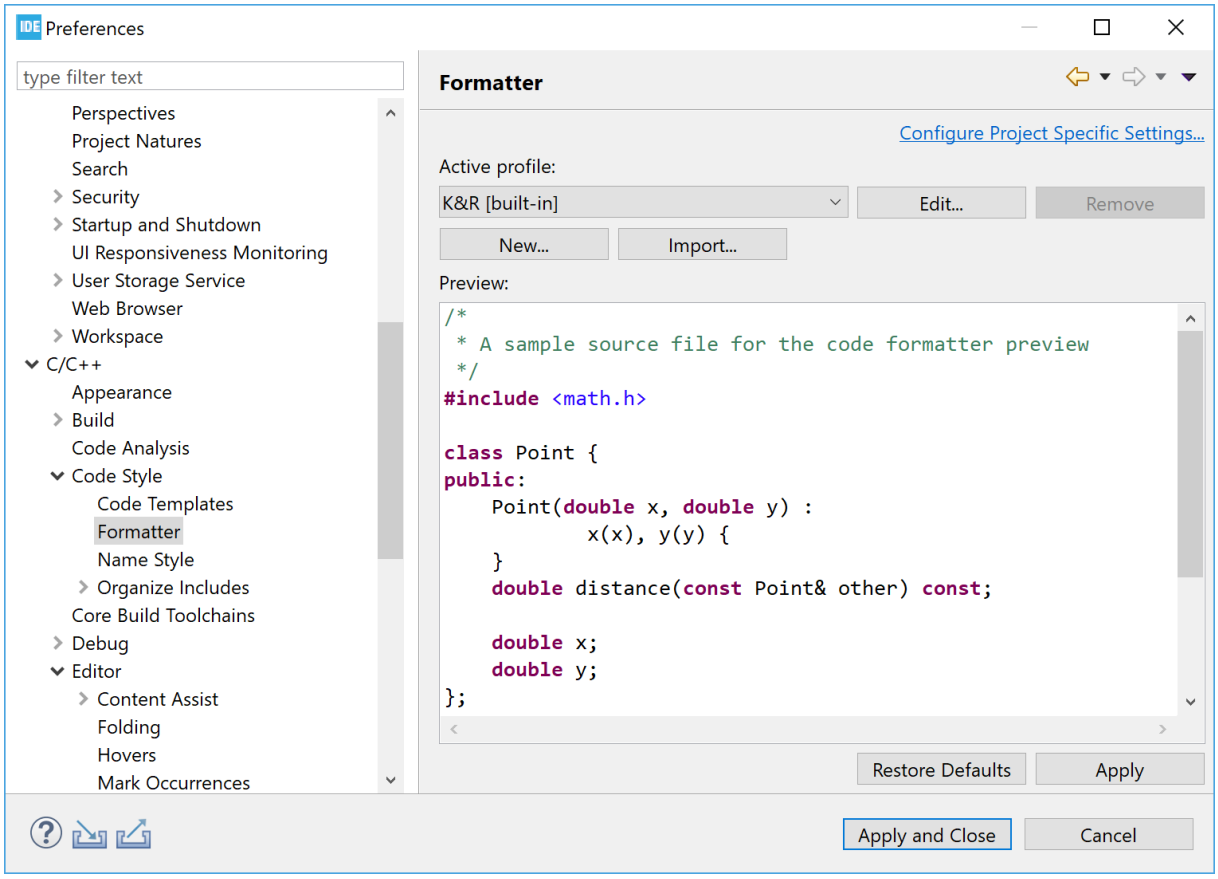


1.5.2 プレファレンス - コード･スタイルの書式設定

特別な書式を使用するようにエディタを設定できます。

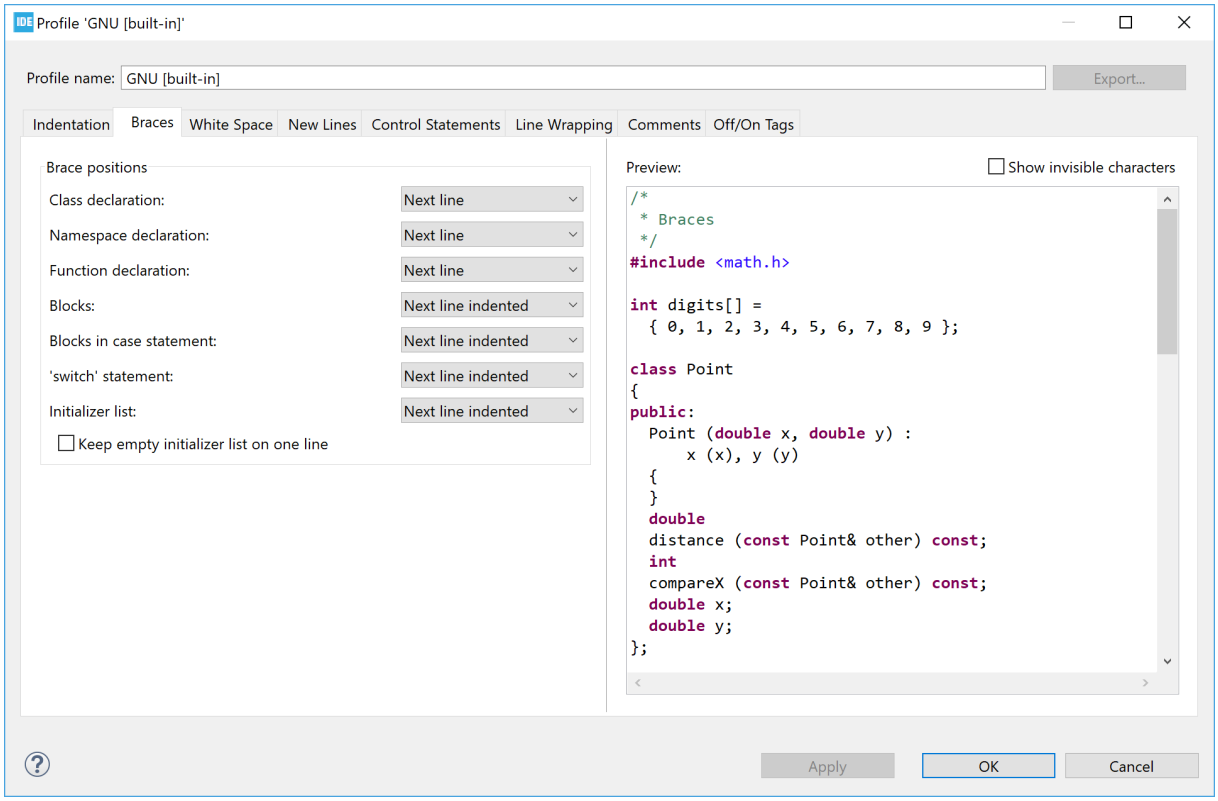
メニュー C/C++Code StyleFormatter を選択すると、アクティブ･プロファイルを設定するプレファレンス･ペインが表示されます。

図 20. プレファレンス - 書式設定



この時点で Edit... をクリックすると新しいダイアログが開き、選択したプロファイルを特定のコーディング·ルールに従って更新できます。このダイアログを 図 21 に示します。

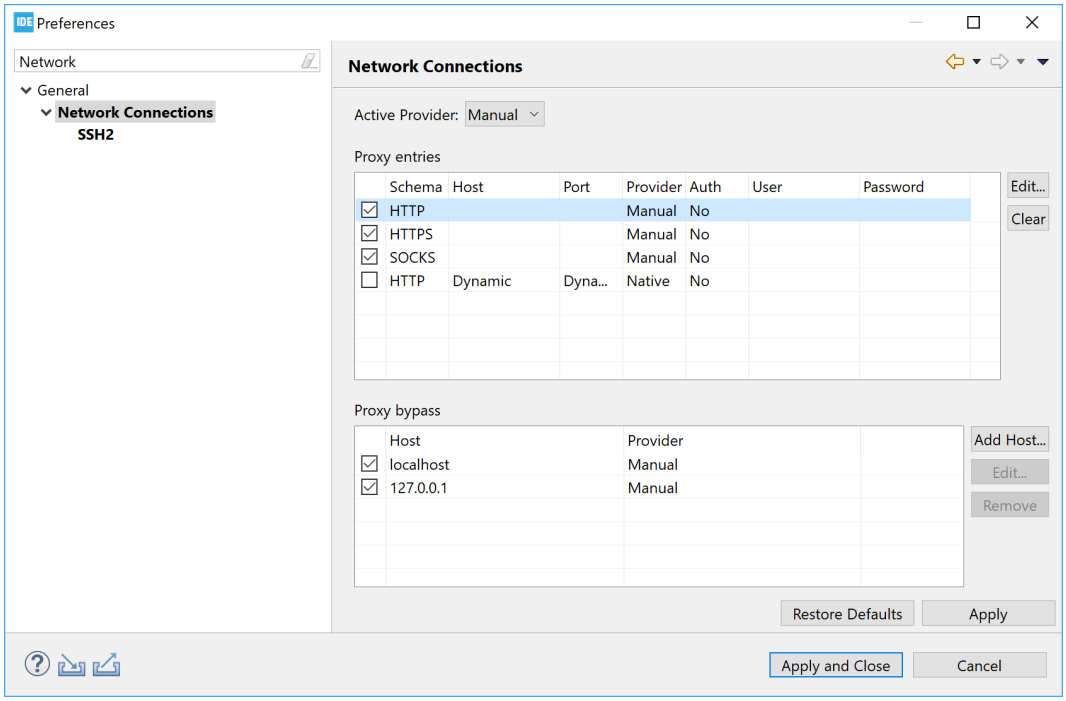
図 21. プレファレンス - コード･スタイルの編集



1.5.3 プレファレンス - ネットワーク･プロキシの設定

STM32CubeIDE のインスタンスは、STM32 デバイス情報にアクセスするためにインターネットを使用します。インターネットへのアクセスにプロキシ･サーバを使用している場合、STM32CubeIDE にいくつかの設定が必要になります。プロキシの設定は、メニュー GeneralNetwork Connections を選択することで表示されるプレファレンス･ペインにあります。設定を変更するには、Active Provider を[Manual]に設定し、[Proxy entries]の中の[HTTP]と[HTTPS]について、Edit... ボタンをクリックしてホスト(Host)、ポート(Port)、ユーザ(User)、パスワード(Password)を特定の値に更新します。

図 22. プレファレンス - ネットワーク接続



注 プロキシ設定の保存に問題が生じる場合は、secure_storage ファイルが壊れている可能性があります。問題を解決するには、次の手順を実行してください。

1. 実行中の STM32CubeIDE アプリケーションをすべて終了します。
2. ファイル C:\Users\user_name\.eclipse\org.eclipse.equinox.security\secure_storage の名前を新しいものに変更します。
3. STM32CubeIDE を再起動します。
4. ユーザやパスワードなどの情報を含むプロキシのネットワーク設定を更新し、新しい secure_storage ファイルとして保存します。

1.5.4 プレファレンス - ビルド変数

STM32CubeIDE のプレファレンスは、IDE 内のみで参照できるビルド変数の機能を備えています。

メニュー C/C++BuildBuild Variables を選択すると、[Build Variables]のプレファレンス･ペインが表示されます。ビルド変数は STM32CubeIDE 内で ${VAR} と表記することで使用できます。Show system variables を有効にすると、使用可能なすべての変数が表示されます。

図 23. プレファレンス - ビルド変数

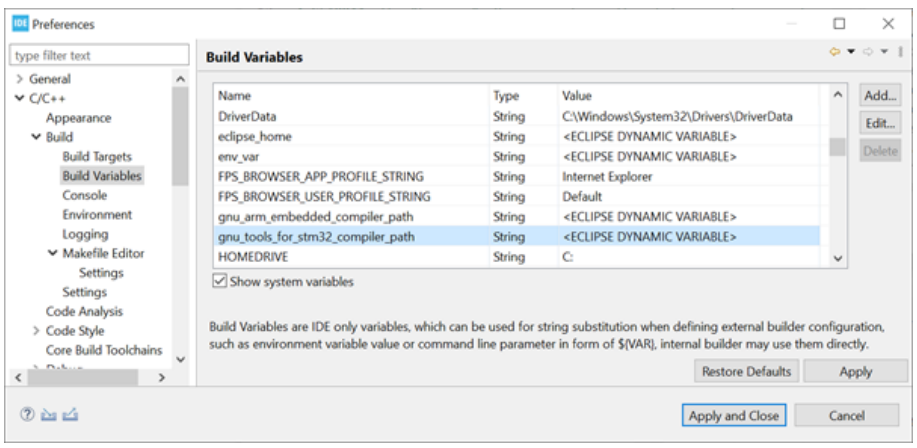
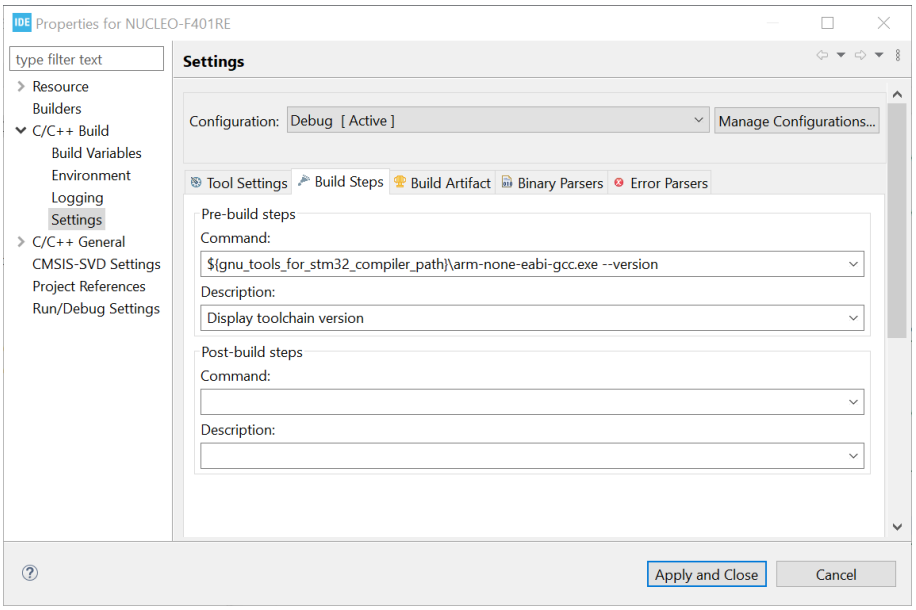


表 1. ツールチェーンのビルド変数の例

変数	説明
gnu_tools_for_stm32_compiler_path	GNU Tools for STM32 ツールチェーンへのパス
gnu_arm_embedded_compiler_path	GNU Arm Embedded ツールチェーンへのパス
stm32cubeide_make_path	make と BusyBox へのパス

図 24 に、ビルド変数を使用してツールチェーンのバージョンを表示する、ビルド前ステップの例を示します。

図 24. ビルド変数によるビルド前ステップ



1.6 ワークスペースとプロジェクト

ワークスペースとプロジェクトの基本概念を比較すると、次のようになります。

- ワークスペースにはプロジェクトが含まれます。ワークスペースは技術的には、プロジェクト·ディレクトリまたはそれらへの参照を含むディレクトリです。
- プロジェクトにはファイルが含まれます。プロジェクトは技術的には、ファイルを含むディレクトリであり、サブディレクトリによって体系化できます。
- 1 台のコンピュータで、ファイル·システム内のさまざまな場所に複数のワークスペースを保持できます。各ワークスペースには、複数のプロジェクトを含めることができます。
- ワークスペース間の切り換えは可能ですが、一度にアクティブ化できるワークスペースは 1 つだけです。
- ユーザはアクティブなワークスペース内の任意のプロジェクトにアクセスできます。他のワークスペース内にあるプロジェクトには、そのワークスペースに切り換えない限りアクセスできません。
- プロジェクトに含まれるファイルは、プロジェクト内のフォルダに物理的に存在する必要はなく、別の場所に保存されたファイルをプロジェクトにリンクすることが可能です。
- ワークスペースの切り換えは、異なるプロジェクト間をすばやく行き来する方法です。これにより、素早く再起動されます。

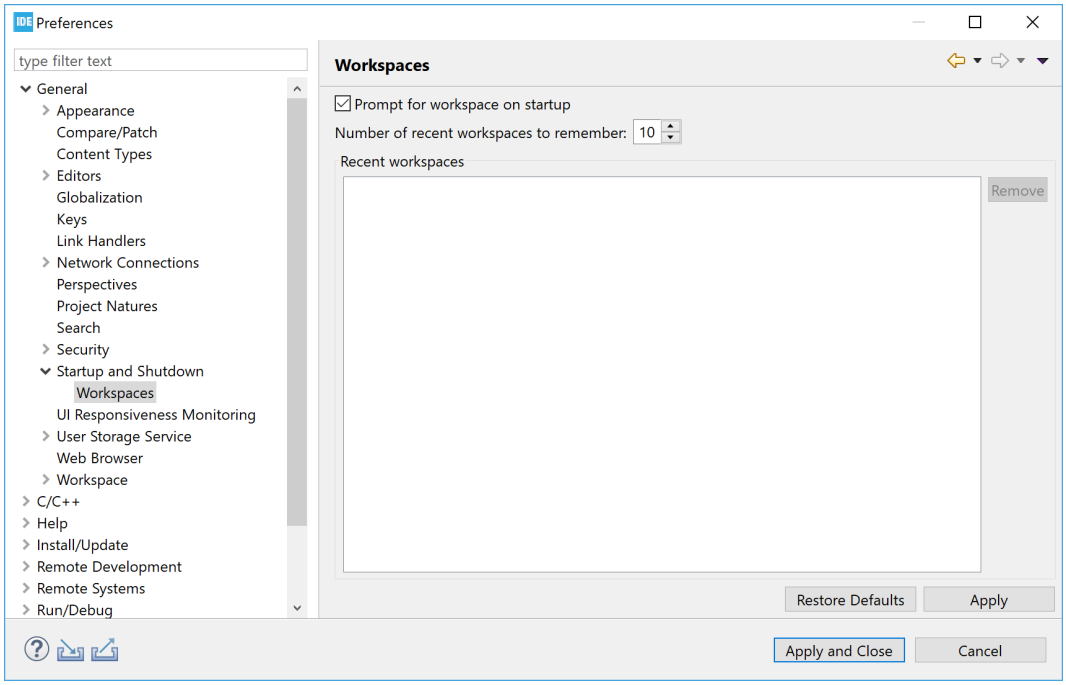
実際には、ワークスペースとプロジェクトを使用する方式により、適切な階層構造、つまりワークスペースにプロジェクトが含まれ、さらにそのプロジェクトにファイルが含まれるという構造を簡単に構成できます。

1.7 既存ワークスペースの管理

ワークスペースは STM32CubeIDE の起動時に選択できます。STM32CubeIDE の使用中も、別のワークスペースへの切り換えが可能です。その場合、新しいワークスペースの選択後、STM32CubeIDE は再起動します。新しいワークスペースで STM32CubeIDE を再起動するには、メニュー FileSwitch Workspace を選択します。

STM32CubeIDE が認識しているワークスペースは、WindowPreferences を選択して、[Preferences]ダイアログの GeneralStartup and ShutdownWorkspaces を選択することで管理できます。右側ペインでは、Prompt for workspace on startup（起動時にワークスペースのプロンプトを表示）を有効にしたり、Number of recent workspaces to remember（記憶する最近のワークスペースの数）に必要な値を設定したりすることが可能です。

図 25. プレファレンス - ワークスペース(Preferences - Workspaces)



最近使用したワークスペースのリストから、不要なものを選択して削除することも可能です。ただし、このリストからワークスペースを削除してもファイルは削除されません。また、ファイル·システムからも削除されません。

1.7.1 ワークスペースのプレファレンスのバックアップ

ワークスペースの既存プレファレンスのコピーを作成しておくことは、一般的に優れた実践行動です。特にクラッシュ発生後にワークスペースを再構築する場合に有用であり、長時間かけて手動で設定をやり直す必要がなくなります。

メニューの FileExport を選択します。次にパネル内の GeneralPreferences を選択します。Next ボタンをクリックして次のページに移り、Export All を有効にするとともに、適切なファイル名を入力します。

1.7.2 ワークスペース間のプレファレンスのコピー

ワークスペース間でプレファレンスをコピーするには、まず ワークスペースのプレファレンスのバックアップ の説明に従って、既存のプレファレンスをエクスポートする必要があります。

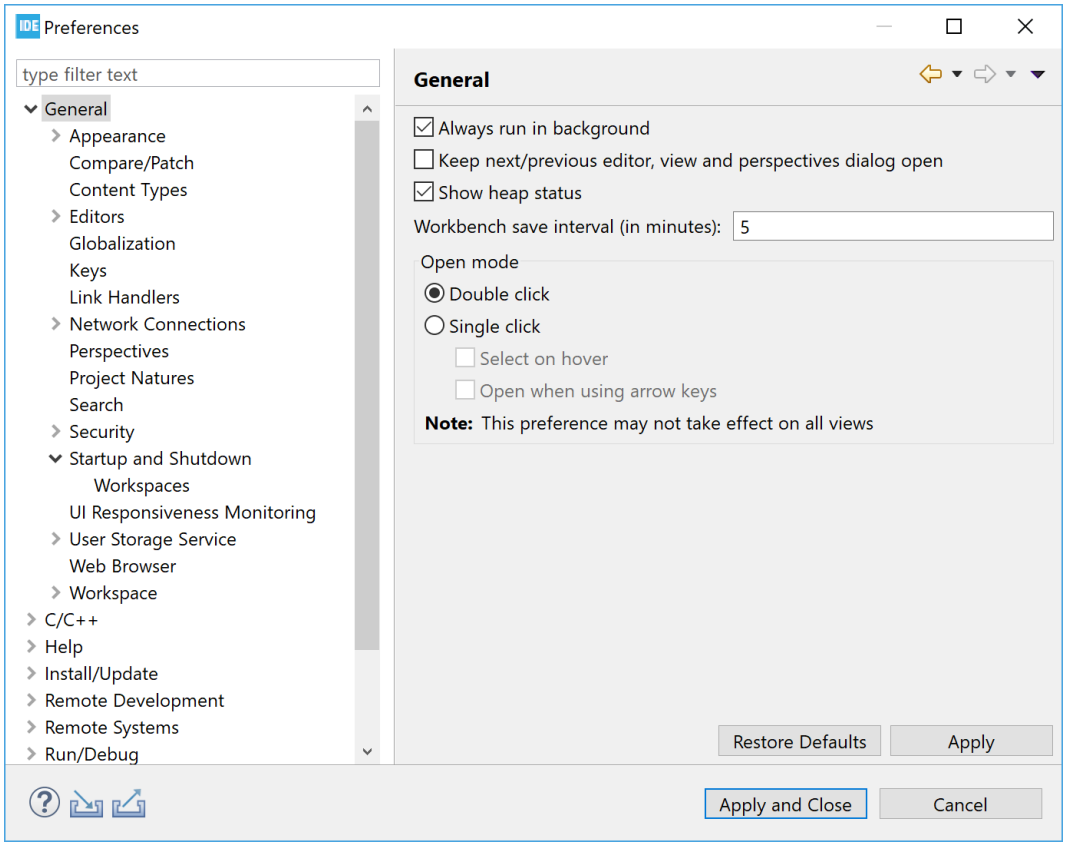
次に FileSwitch Workspace を選択し、新しいワークスペースを指定します。STM32CubeIDE が再起動し、新しいワークスペースが開きます。

メニューの FileImport を選択し、パネル内の GeneralPreferences を選択します。Next ボタンをクリックして次のページに移り、Import All を有効にして、ファイル名を入力します。これで、両方のワークスペースに同じプレファレンスが設定されます。

1.7.3 Java ヒープ領域の監視

Java ヒープ領域の使用量を監視するには、メニュー WindowPreferences を選択します。[Preferences]ページの General ノードを選択し、Show heap status を有効にします。Java ヒープの現在の使用量と空き容量が STM32CubeIDE のステータス·バーに表示されるようになります。ステータス·バーから手動でガベージ·コレクションを開始することも可能です。

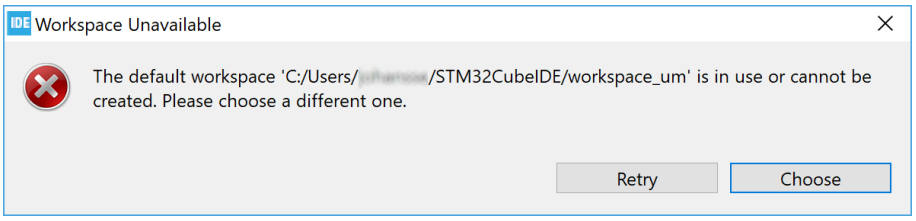
図 26. Java ヒープ領域の状態表示



1.7.4 使用できないワークスペース

ある 1 つのワークスペースには、一度に 1 つの STM32CubeIDE インスタンスしかアクセスできません。これはワークスペース内の変更が競合しないようにするためです。STM32CubeIDE を、同じプログラムの別のインスタンスが既に使用中のワークスペースで起動しようとすると、次のエラー·メッセージが表示されます。

図 27. Workspace unavailable



このメッセージが表示された場合は、別のワークスペースを選択するか、既に実行中の STM32CubeIDE に戻ってください。

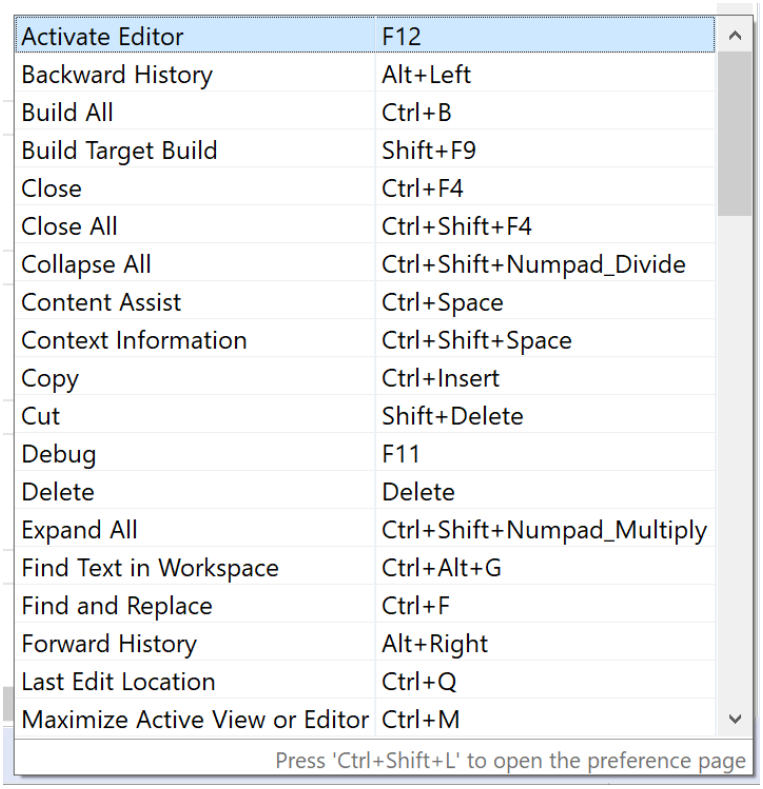
1.8 STM32CubeIDE と Eclipse® の基本

STM32CubeIDE は、極めて多数の機能を備えているため、本当に便利な機能を見逃しがちです。注目すべき機能として、C/C++ コメントのスペルチェック、単語やコードの自動補完、コンテンツ·アシスト、パラメータ·ヒント、コード·テンプレートなどがあります。エディタは、インクルード·ファイルの依存関係ブラウザ、ハイパーテキスト·リンクによるコード·ナビゲーション、ブックマーク、やることリスト、強力な検索機能なども備えています。この後のセクションでは、便利でありながら見逃しがちなツールをいくつか紹介します。

1.8.1 キーボード･ショートカット

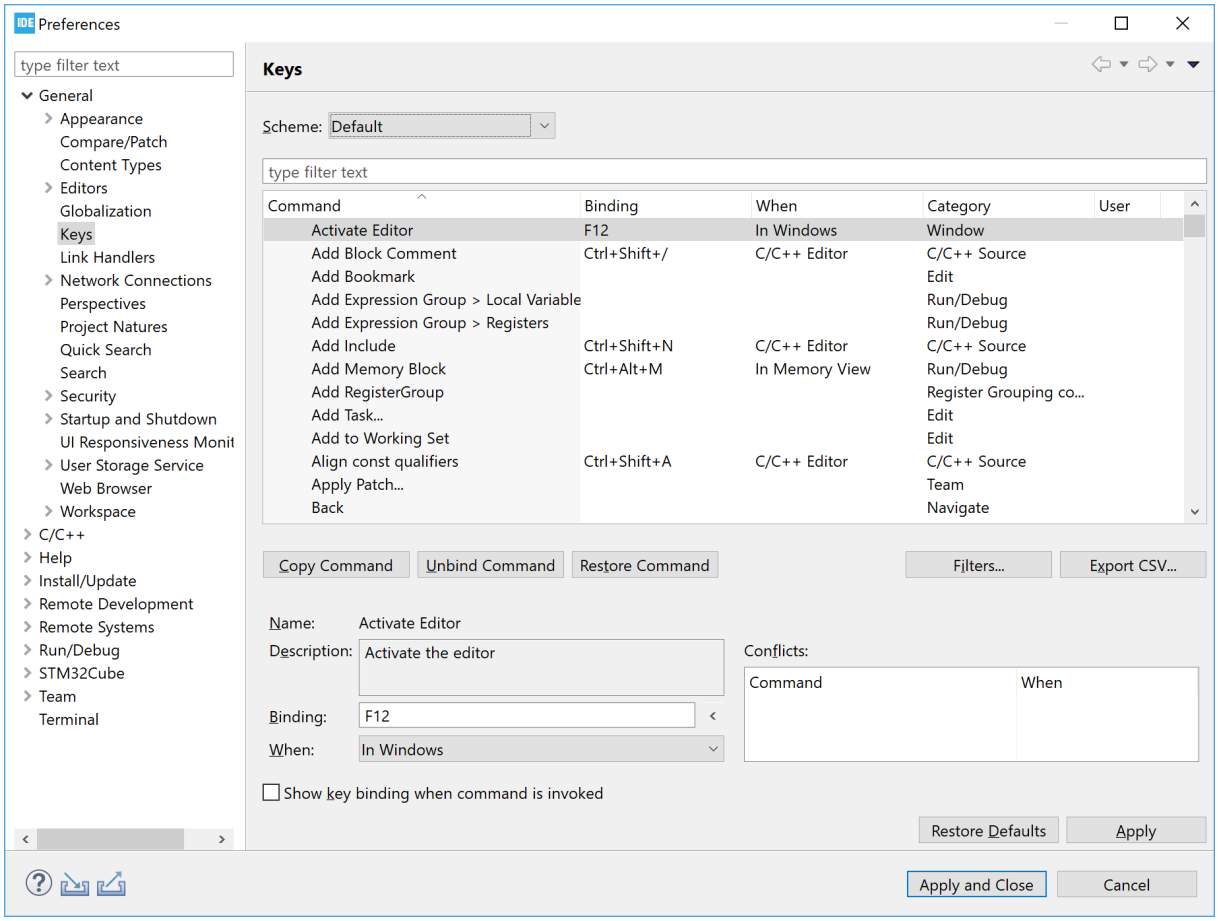
マウスの代わりにキーボード･ショートカットを使用すると便利です。覚えておくべき重要なショートカットの一つが CTRL+Shift+L です。このショートカットは、使用可能なすべてのショートカットの早見表を表示します。

図 28. ショートカット･キー



この早見表の表示中に CTRL+Shift+L を押すと、[Preferences]ダイアログの[Keys]ペインが開きます。

図 29. ショートカットのプレファレンス



[Keys]ペインではショートカットの詳細を確認し、スキームの変更（デフォルト、Emacs、Microsoft 社® Visual Studio）、ショートカット･キーの再設定、その他の操作を行うことができます。

表 2 に、一部のキーの内容とデフォルト割当てを示します。

表 2. キー･ショートカットの例

コマンド	割当て	適用場所
コピー（Copy）	Ctrl+C	ダイアログとウィンドウ
カット（Cut）	Ctrl+X	ダイアログとウィンドウ
貼り付け（Paste）	Ctrl+V	ダイアログとウィンドウ
デバッグ（Debug）	F11	ウィンドウ
宣言を開く（Open declaration）	F3	C/C++ エディタ
リファレンス（References）	Ctrl+Shift+G	C/C++ エディタ / ビュー
ファイルを検索して開く（Find and open files）	Ctrl+Shift+R	C/C++ エディタ / ビュー
通常 / ブロック選択モードの切り換え（Toggle selection mode normal/block）	Alt+Shift+A	C/C++ エディタ / ビュー
ズームイン（Zoom In）	Ctrl++	テキスト編集
ズームアウト（Zoom Out）	Ctrl+-	テキスト編集

1.8.2 エディタのズームインとズームアウト

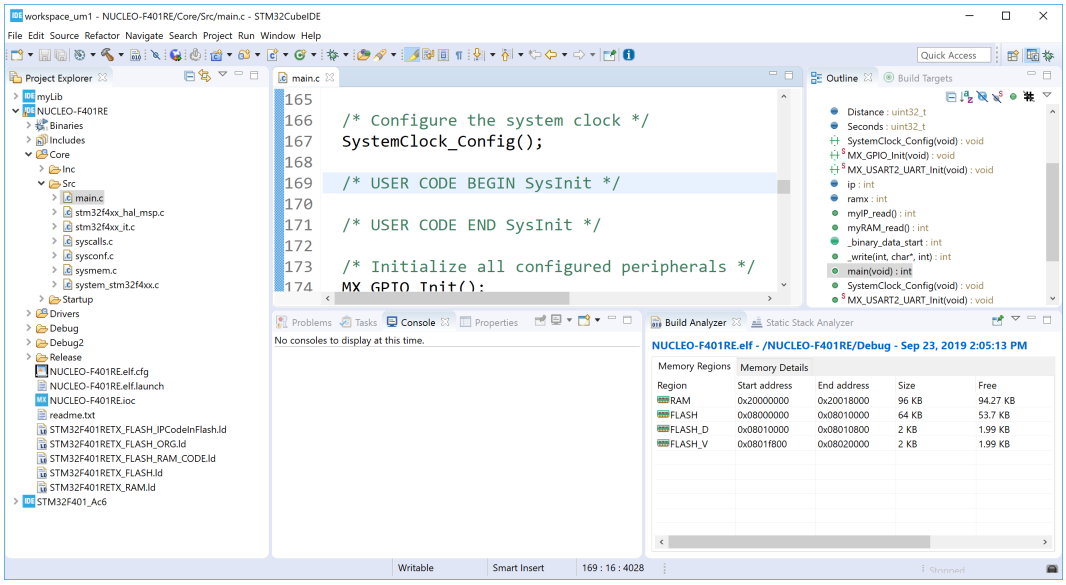
Ctrl++ と Ctrl+- を押すことで、テキスト･エディタのデフォルト･フォント･サイズを増減できます。

- Ctrl++ はテキストにズームインします。

- Ctrl-- はテキストからズームアウトします。

注　数値キーパッド付きキーボードで、数値キーパッドにある + または - キーを使用する場合は、上記に加えて Shift キーも押すことでズームが機能します（Ctrl+Shift+ または Ctrl+Shift-）。

図 30. テキストをズームインしたエディタ



1.8.3　ファイルをすばやく検索して開く

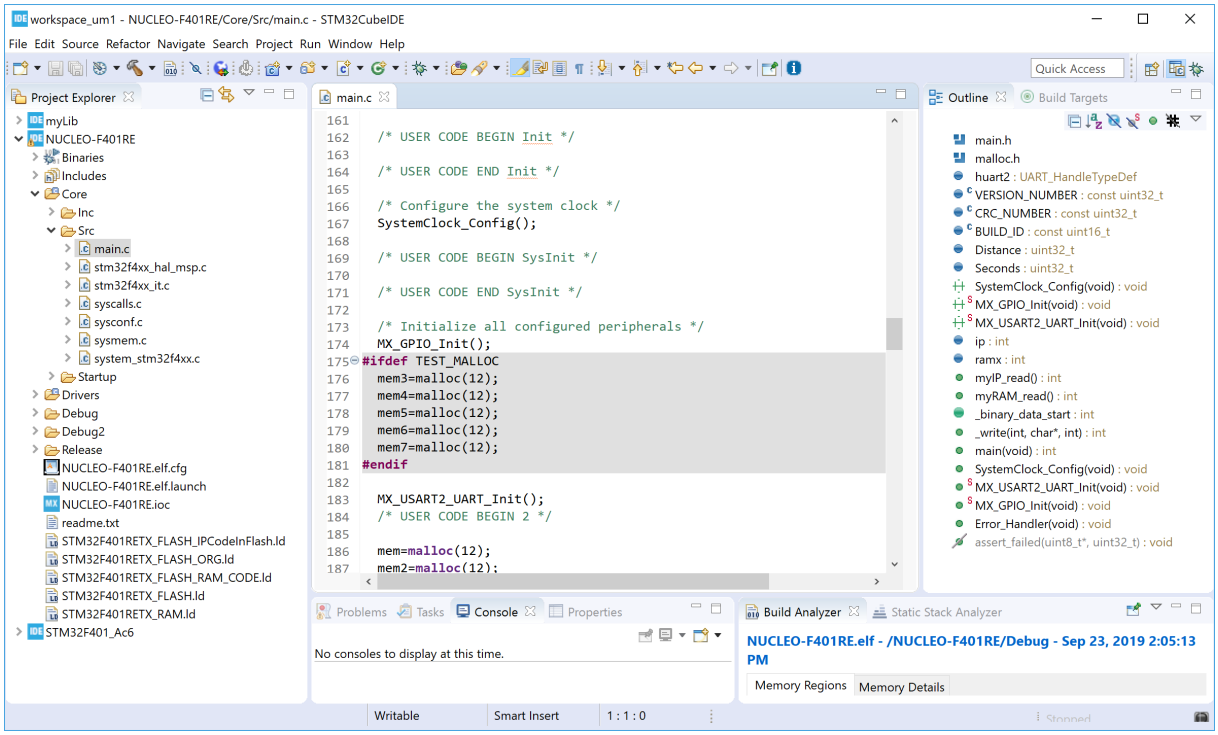
ファイルをすばやく検索して開くキーボード·ショートカット Ctrl+Shift+R は、見逃しがちな機能の一つです。開くファイルの名前の一部を入力します。必要に応じて、* や ? を追加し、ワイルドカード検索を実行することも可能です。エディタが、一致するファイル名のリストを表示します。検索結果のリストから目的のファイルを選択し、次の 3 つの方法のいずれかによりファイルを開きます。

- Show In : ドロップダウン·リストで選択したビューの 1 つにファイルを送ります（#include ファイル依存関係ブラウザ·ビューなど）。
- Open With : ドロップダウン·リストで選択したエディタでファイルを開きます。
- Open : 単に標準 C/C++ エディタでファイルを開きます。おそらく最も一般的に使用されるオプションでしょう。

1.8.4　分岐の折りたたみ

`#if` と `#endif` で囲まれたコード·ブロックは折りたたむことができます。この機能を有効にするには、WindowPreferences から、さらに C/C++EditorFolding に移動して、チェックボックスの Enable folding of preprocessor branches（#if/#endif）にチェックを入れます。チェック後、エディタを再起動する必要があります。ファイルを閉じてから再び開くとエディタの左余白に小さなアイコンが表示されるので、折りたたみ機能が有効になったことがわかります。

図 31. エディタの折りたたみ



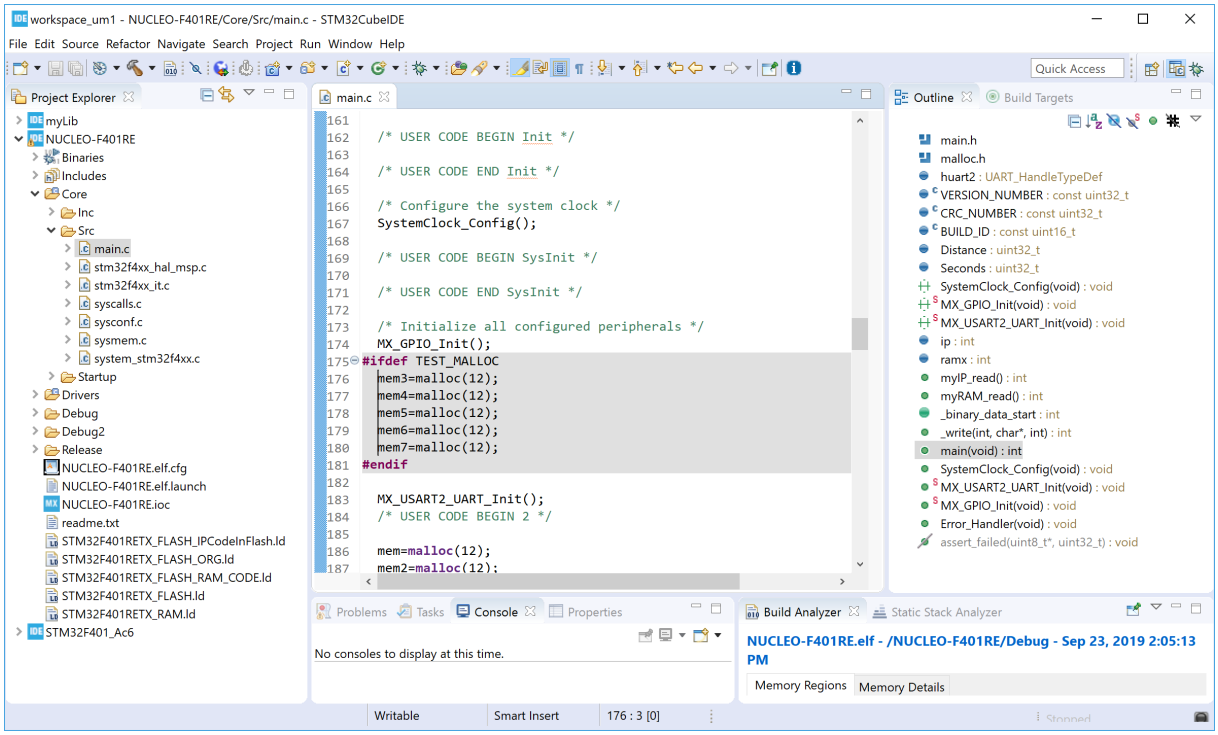
1.8.5 ブロック選択モード

Alt+Shift+A は、標準とブロックの選択モードを切り換えます。ブロック·モードを有効にすると、マウスまたはキーボードの SHIFT+ 矢印キーでテキスト·ブロックを選択できます。

ブロック選択モードの使用方法

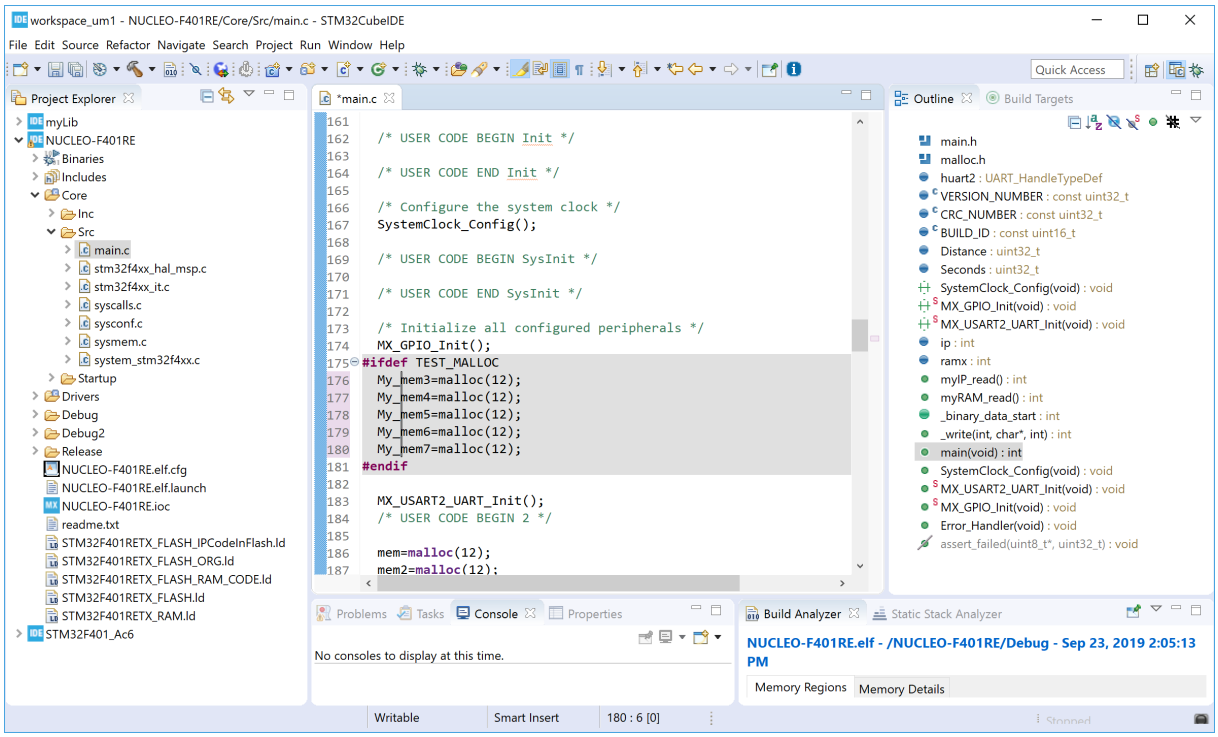
ブロック選択モードの使用を開始するには、Alt+Shift+A を押します。テキスト内の任意の場所をクリックして下にドラッグします。図 32 に示すように、列方向にマークが表示されます。

図 32. エディタのブロック選択



何らかのテキストを追加すると、マークされた行のすべてに、そのテキストが入力されることを確認してください。`My_` というテキストを追加した場合の例を 図 33 に示します。

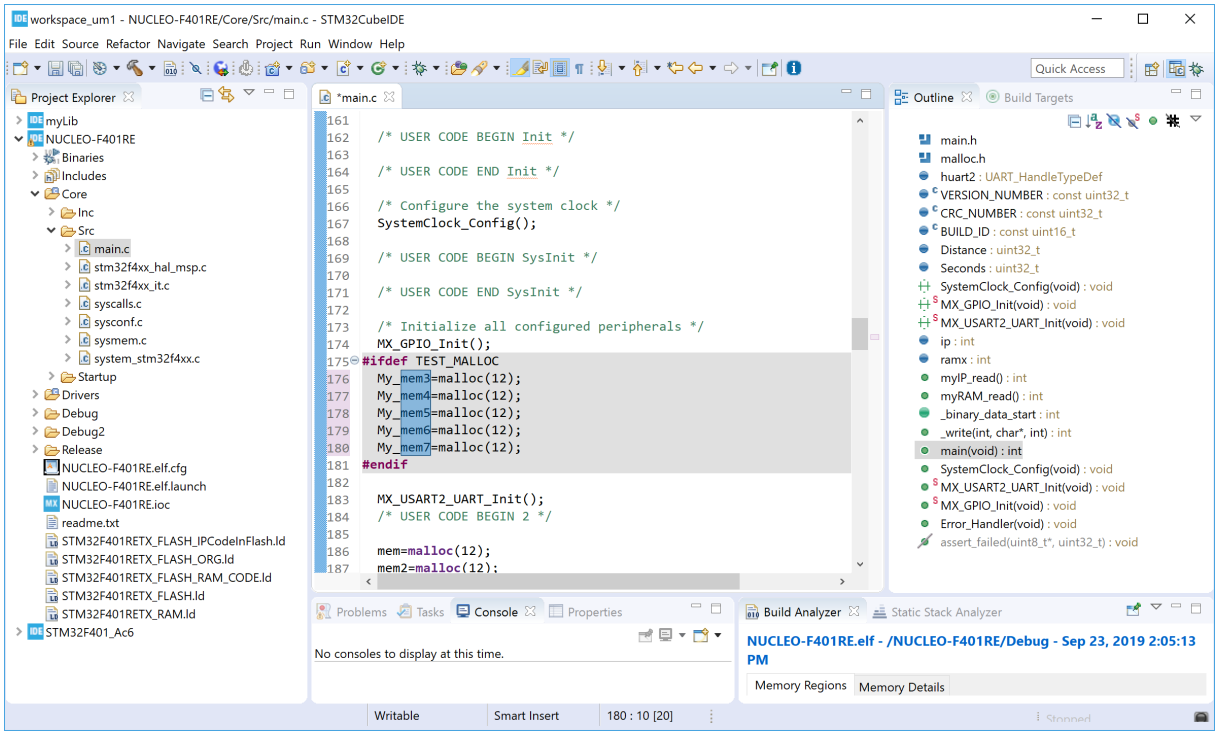
図 33. エディタのテキスト･ブロックへの追加



領域の選択と編集

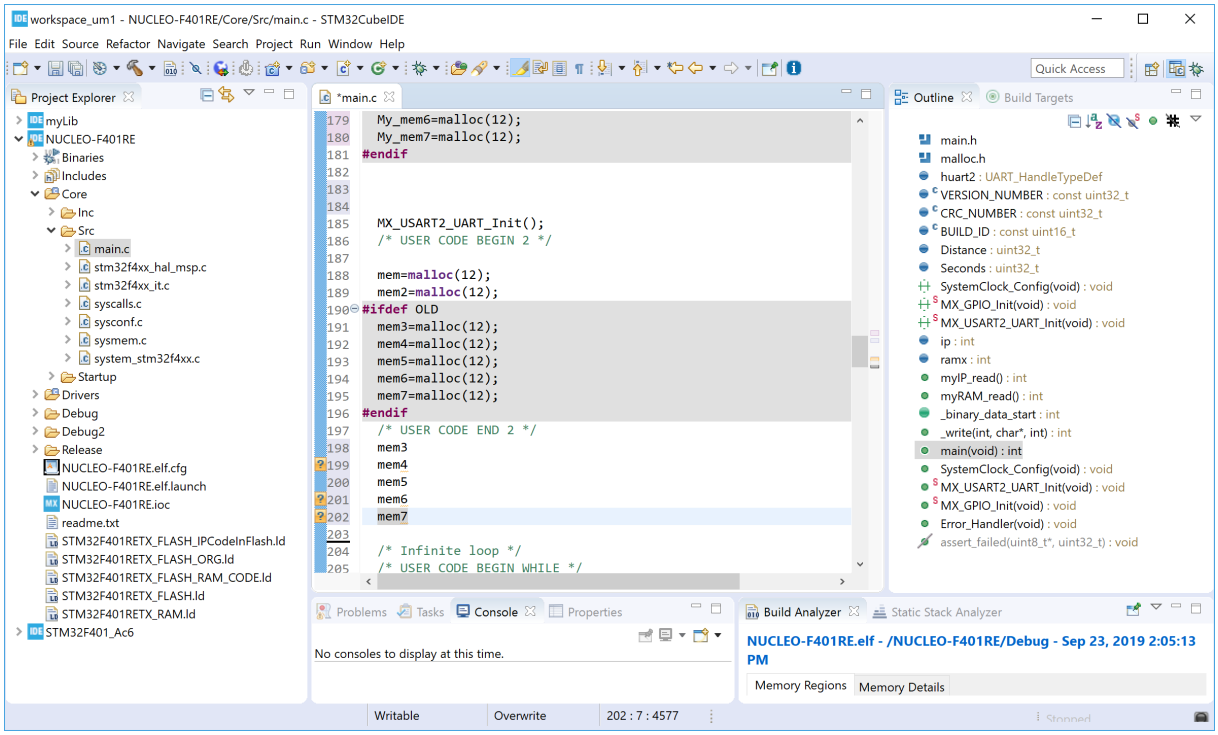
ブロックを選択します。図 34 では、mem3 から mem7 で始まるブロックを選択しています。

図 34. エディタの列ブロック選択



選択したブロックを Ctrl+C によりコピーします。コピーしたテキストは、その後、任意の場所に挿入できます。それには、Alt+Shift+A を押して選択モードを標準モードに戻し、別の行にカーソルを移動してから Ctrl+V を押してコピーした列を新しい行に貼り付けます。

図 35. エディタの列ブロック貼り付け



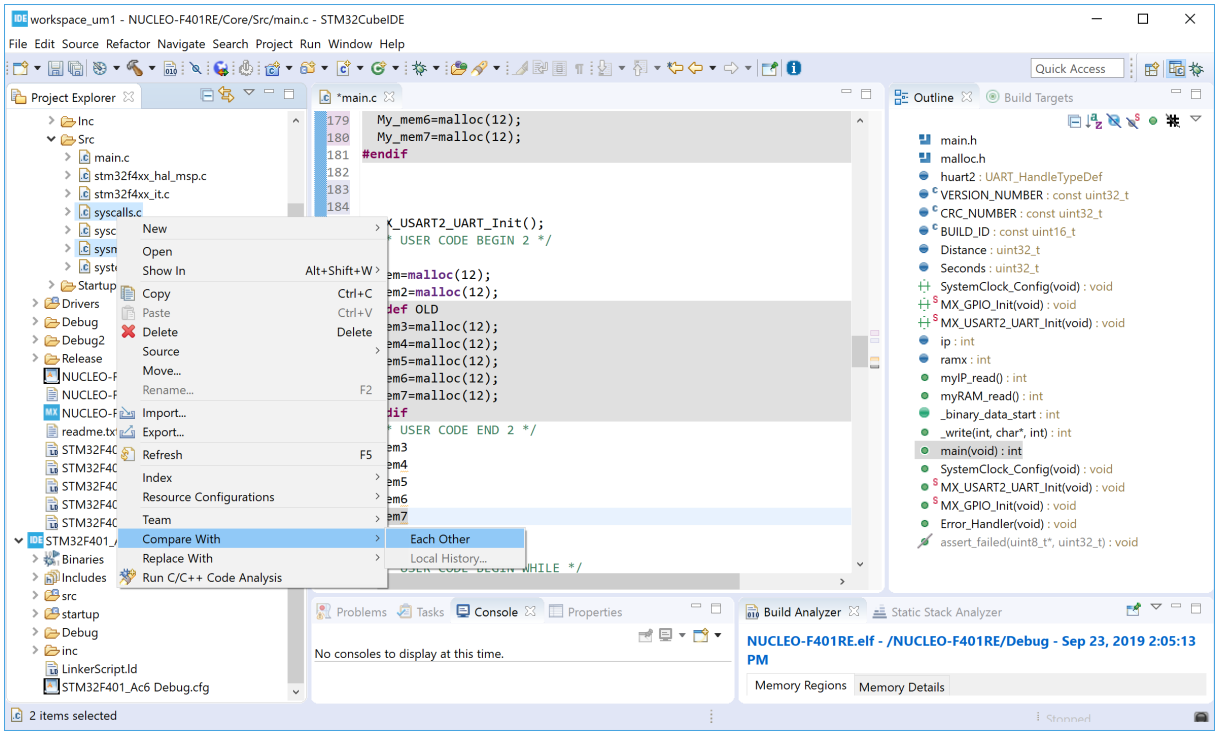
1.8.6 ファイルの比較

STM32CubeIDE では、次の手順により簡単に 2 つのファイルを比較できます。

1. [Project Explorer]ビューで 2 つのファイルを選択します。
2. 一方のファイルをクリックします。
3. CTRL を押します。
4. もう一方のファイルをクリックします。
   これで、[Project Explorer]ビュー内で両方のファイルにマークが付きます。
5. 右クリックして、Compare WithEach Other を選択します。

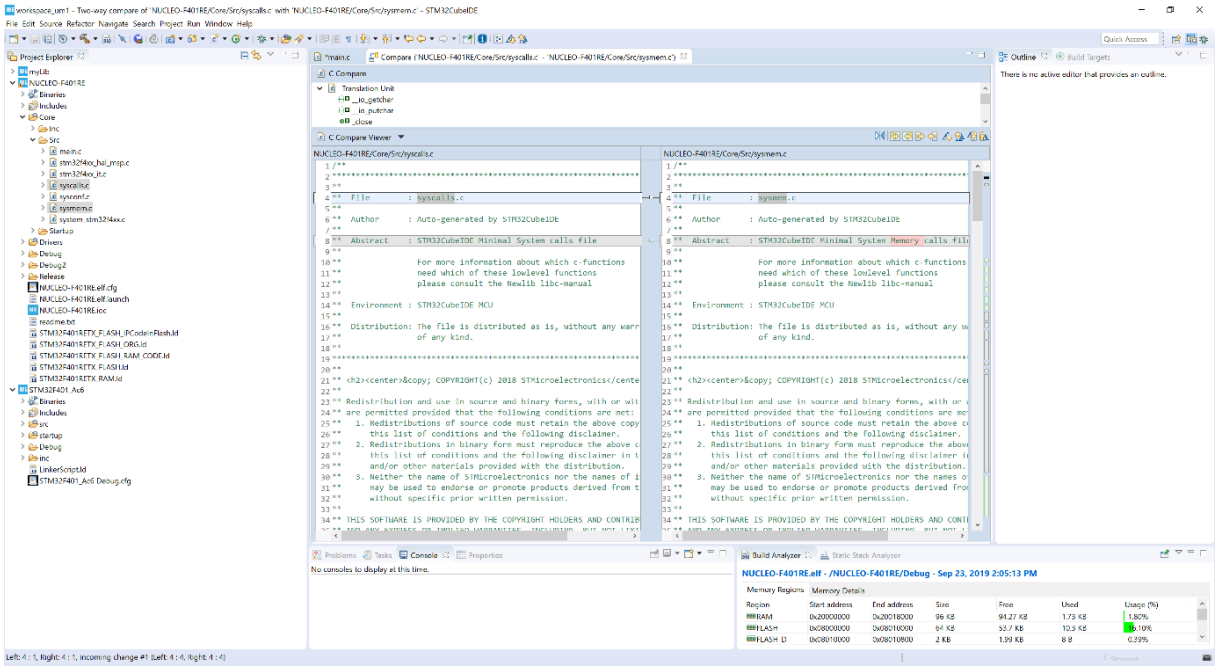
注 ファイル比較の処理方法は設定可能です。例えば、空白を無視する設定をプレファレンスで有効にできます。WindowPreferences により[Preferences]ページを開き、GeneralCompare/Patch を選択して Ignore white space を有効にします。

図 36. エディタ - ファイルの比較



ファイル差分エディタが開き、両ファイルを比較します。

図 37. エディタ - ファイル差分

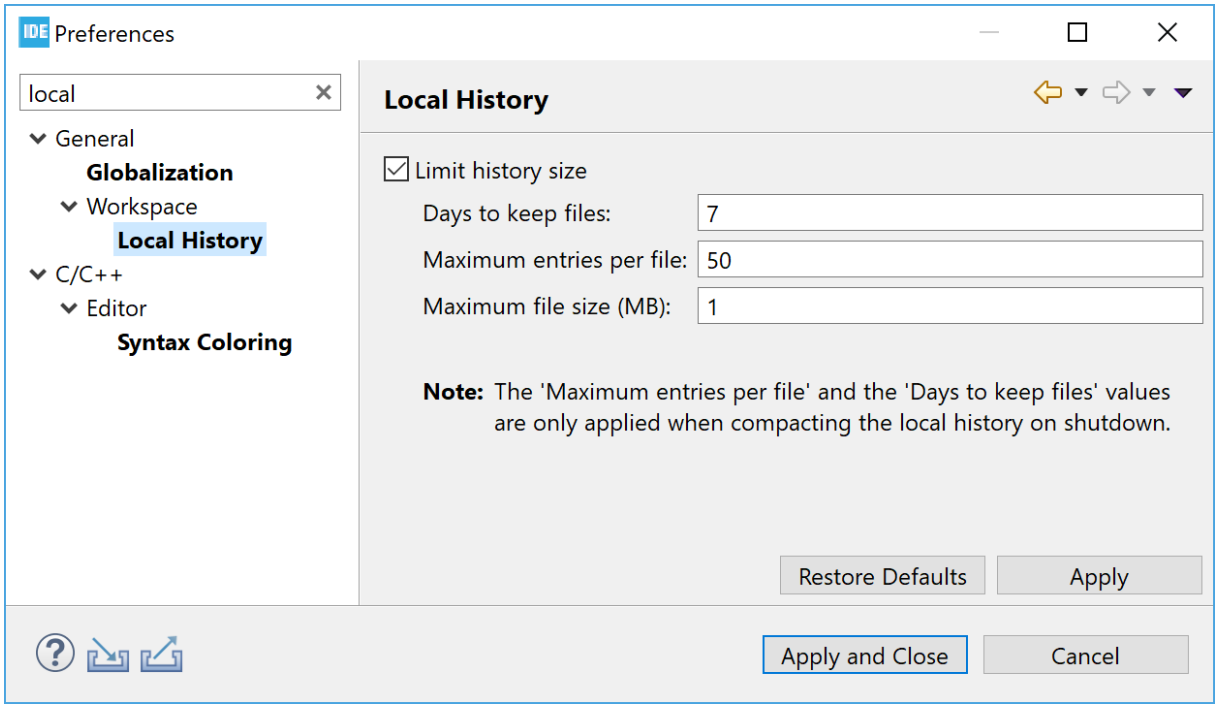


ナビゲーション·ボタンを使用して両ファイルの異なる場所間を移動するか、単にスクロール·バーを使用してビュー内を移動しながらファイル差分を確認していきます。

1.8.7 ローカル·ファイル履歴

プロジェクトは、Apache® Subversion®（SVN）または Git™ などのバージョン管理システムを使用して維持管理することを推奨します。そのようなシステムに加えて、STM32CubeIDE には編集されたファイルの履歴が保存されたローカル·ファイルを保持しています。これは、ファイルが正常に機能しなくなったために何らかの調査が必要になった場合に役立つことがあります。ワークスペースのプレファレンスには、[Local History]ページがあります。

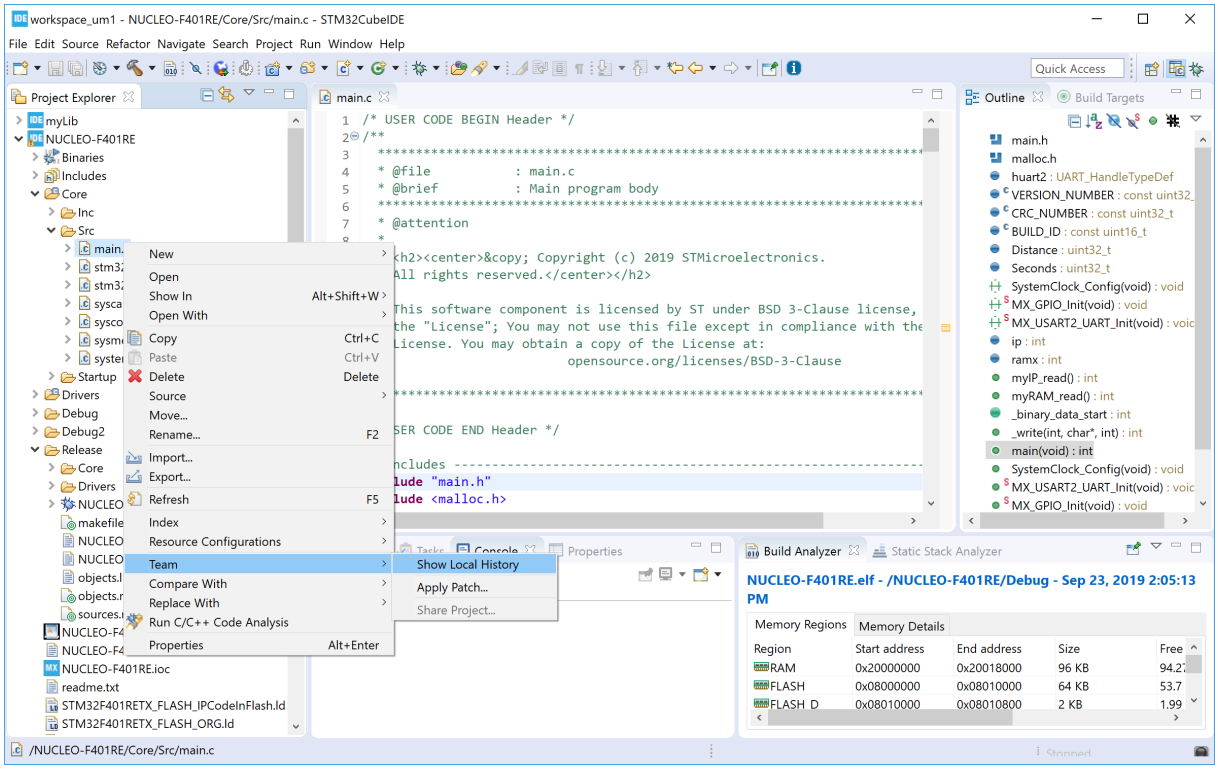
図 38. ローカル履歴



ファイルのローカル履歴を表示するには、次の手順を実行します。

1. [Project Explorer]ビューでファイルを選択します。
2. 右クリックします。
3. TeamShow local History を選択します。

図 39. ローカル履歴の表示



[History]ビューが開き、ファイルの履歴が表示されます。

図 40. ファイル履歴

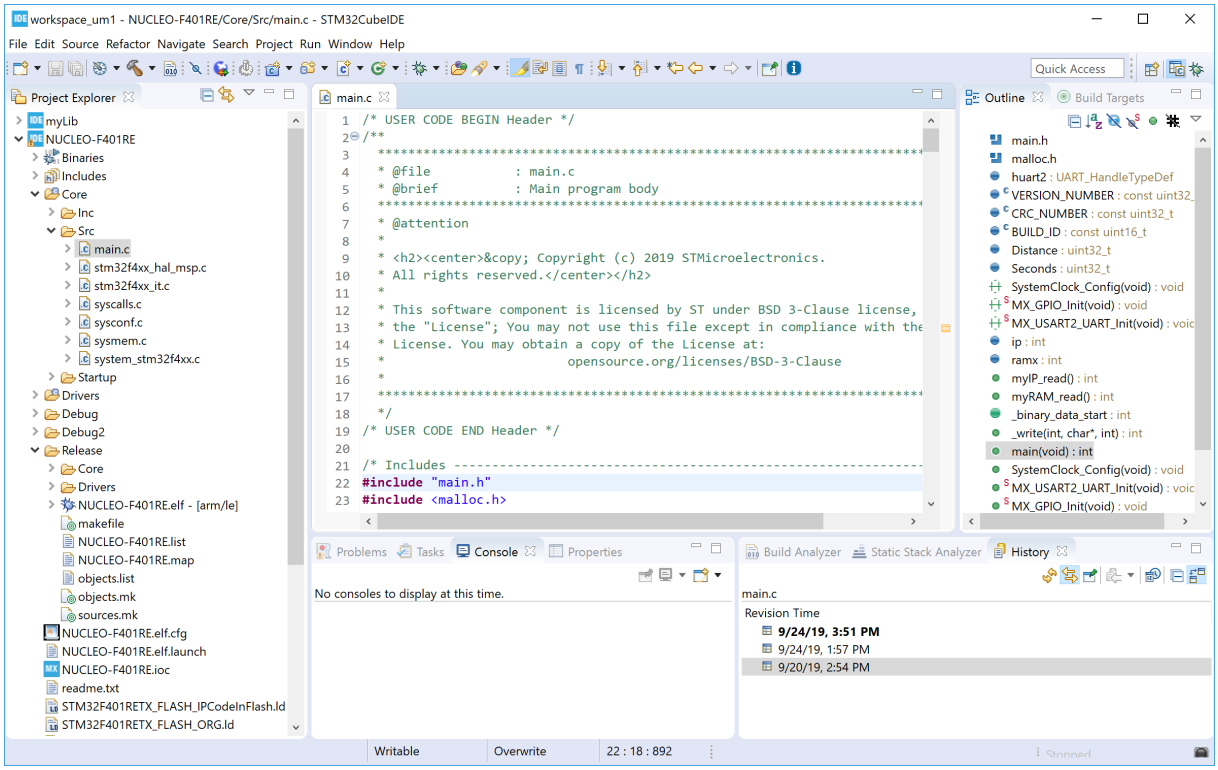
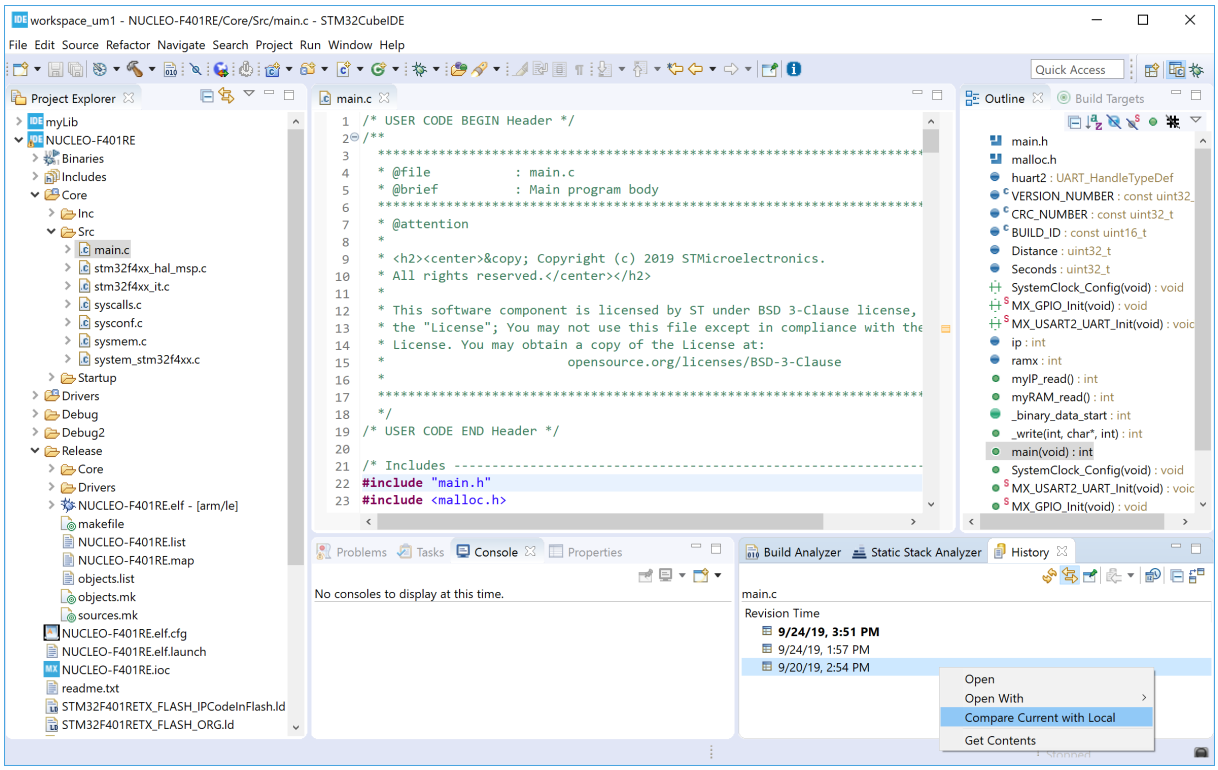


図 40 の例では、main.c に 3 つのリビジョンが存在することがわかります。[History]ビューのファイルをダブルクリックすると、そのファイルがエディタで開きます。

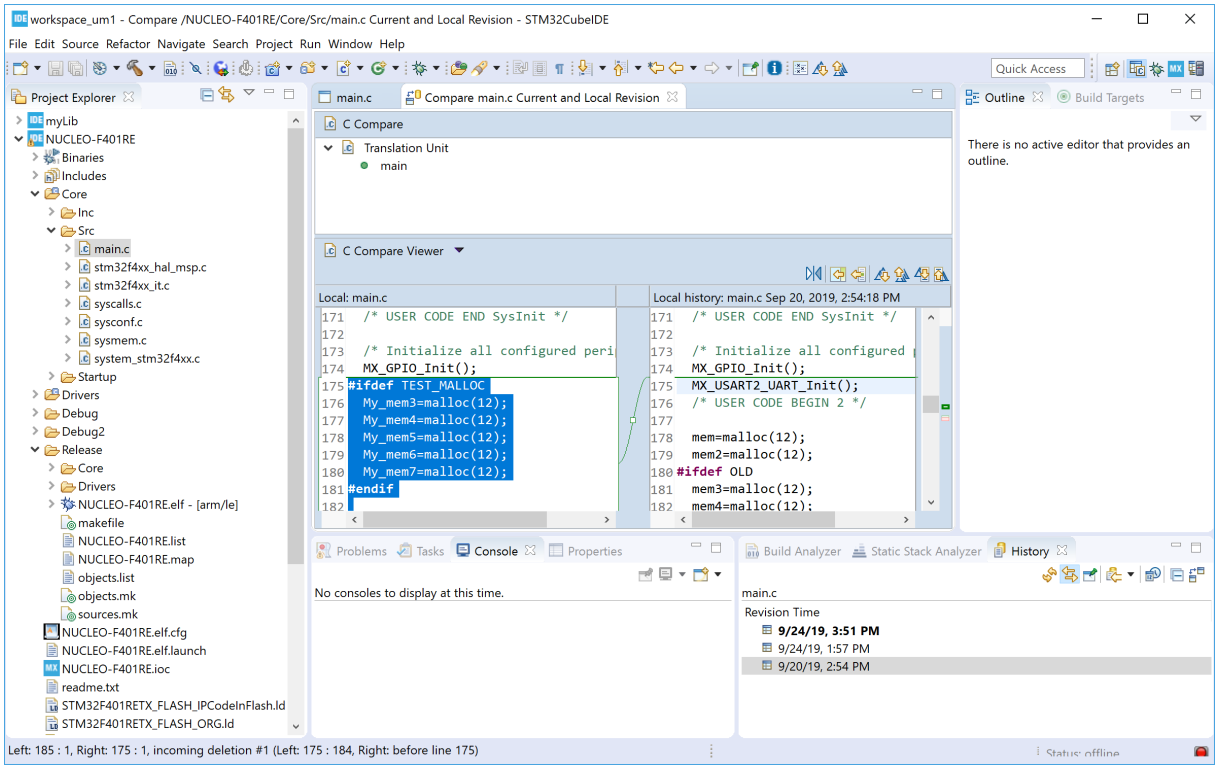
履歴のファイルを右クリックして Compare Current with Local を選択して、そのバージョンと現在のバージョンを比較します。

図 41. 現在の履歴とローカル履歴の比較



この操作によりファイル差分エディタが開き、ファイルの変更個所が表示されます。

図 42. ローカル·ファイル差分の比較



2 C/C++ プロジェクトの作成とビルド

セクション 1.6 ワークスペースとプロジェクト で説明したように、ワークスペースとはプロジェクトを含むディレクトリです。新規作成したばかりのワークスペースは、プロジェクトが一切含まれない空の状態です。ワークスペース内にプロジェクトを作成するかインポートする必要があります。このセクションでは、ワークスペース内のプロジェクトの作成方法とビルド方法に関する情報を提供します。また、プロジェクトのインポートおよびエクスポートの方法も説明します。

2.1 プロジェクトの概要

プロジェクトは、ファイルを含むワークスペース内のディレクトリであり、サブディレクトリによって体系化されています。アクティブなワークスペース内のプロジェクトは、すべてアクセス可能です。プロジェクトに含まれるファイルは、プロジェクト内のフォルダに物理的に存在する必要はなく、別の場所に保存されたファイルをプロジェクトにリンクすることが可能です。他のワークスペース内にあるプロジェクトには、そのワークスペースに切り換えるか、それらのプロジェクトの一部を現在使用中のワークスペースにインポートしない限りアクセスできません。

プロジェクトは名前の変更や削除が可能です。ワークスペースに多数のプロジェクトが含まれる場合、作業をしやすくするために一部を閉じることもできます。閉じたプロジェクトは、いつでも再度開けます。

このセクションでは、STM32CubeIDE がサポートする次の 2 種類の STM32 プロジェクトに注目します。

- 実行可能プログラム
- スタティック・ライブラリ・プロジェクト

ただし、STM32CubeIDE が基盤とする Eclipse® C/C++ Development Toolkit（CDT™）は、基本的なプロジェクト・ウィザードも備えています。これを使用して C マネージド・ビルド、C++ マネージド・ビルド、makefile プロジェクトなどを作成できます。

STM32 のプロジェクトは、次のいずれかによって構成されます。

- C または C++
- 生成された実行可能ファイルまたはライブラリ・ファイル
- STM32Cube に基づくプロジェクト（STM32 ファームウェア・ライブラリ・パッケージを使用）または空のプロジェクト

STM32 のプロジェクトは、高度なアンブレラ・プロジェクト構造にも対応しています。これは、1 つのプロジェクトに多数のプロジェクトが含まれる構造で、マルチコア・デバイスのコアごとに 1 つのプロジェクトが存在する場合などに使用します。

2.2 新しい STM32 プロジェクトの作成

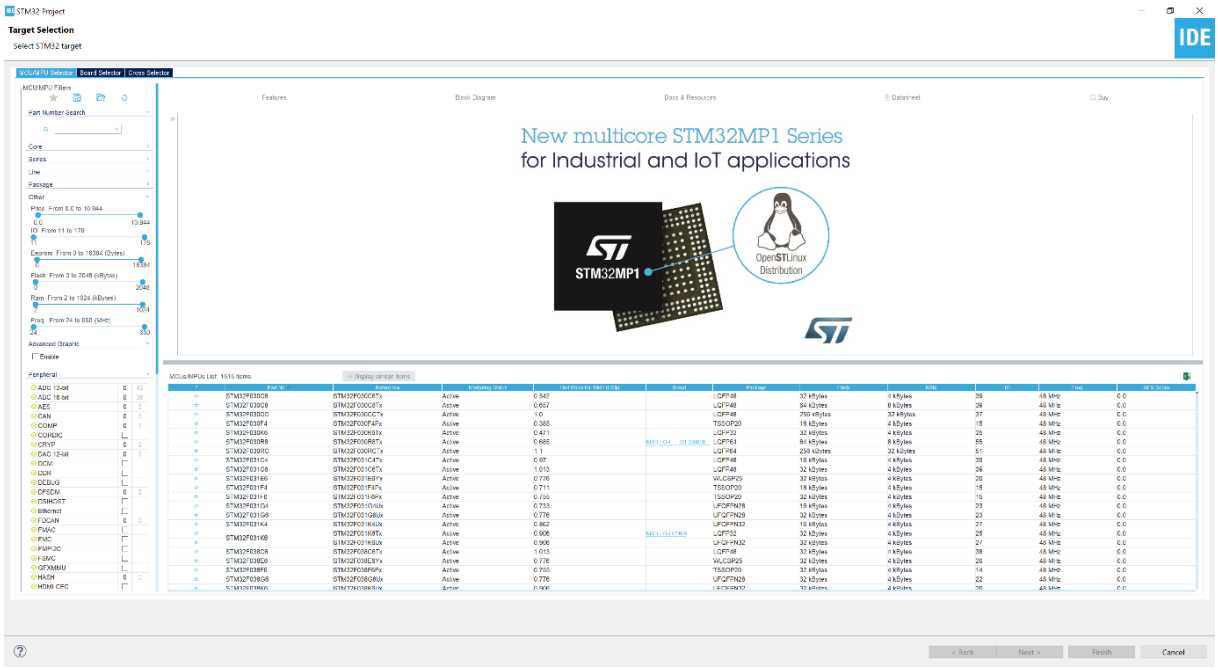
2.2.1 新しい STM32 実行可能プロジェクトの作成

STM32 の C/C++ プロジェクトを新規作成する最も簡単な方法は、STM32 プロジェクト・ウィザードを使用することです。ウィザードはメニュー FileNew STM32 Project で選択します。

C/C++ の新しいプロジェクトを作成するもう一つの方法は、Information Center を開き、Start new STM32 project をクリックします。セクション 1.3 Information Center で説明したとおり、Information Center は、ツールバーの  ボタンをクリックするか、メニュー HelpInformation Center を選択することで開きます。

いずれの方法でも、STM32 プロジェクト・ターゲット選択ツールが初期化され起動します。

図 43. STM32 ターゲット選択

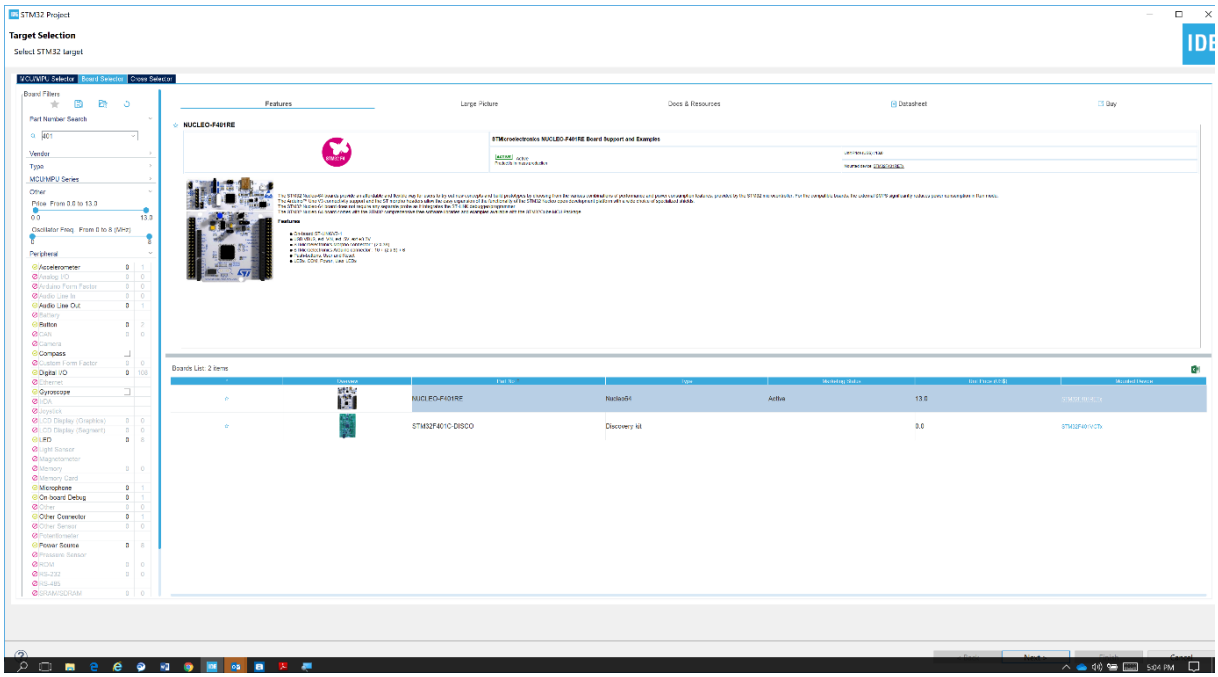


ウィンドウ上部では[MCU/MPU Selector]または[Board Selector]タブを選択できます。特定のデバイス用にプロジェクトを作成する場合は前者を、特定のボード用のプロジェクトが必要な場合は後者を使用します。

このセクションでは、[Board Selector]を使用して、NUCLEO-F401RE ボード用のプロジェクト作成方法を紹介します。

ウィンドウ左側に用意された各種フィルタの中の[Part Number Search]フィールドに「401」と入力することで、名前にこの文字列を含むボードに絞り込みます。図 44 では、2 つのボード Nucleo ボード と ディスカバリ·ボード が表示され、NUCLEO-F401RE ボードが選択されています。

図 44. STM32 ボード選択



[Features]、[Large Pictures]、[Docs & Resources]、[Datasheet]、[Buy]の 5 つのタブで、選択したボードまたはデバイスに関する詳細情報を表示できます。例えば、ボード用に用意されたドキュメントは、[Docs & Resources]タブを選択することで、表示され開くことができます。[Datasheet]を選択すると、ST マイクロエレクトロニクスの Web サイトからボードのデータシートをダウンロードできます。

NUCLEO-F401RE ボードを選択して Next をクリックすると、[Project setup]ページが開きます。

ダイアログ·ボックスでプロジェクト名を入力し、必要なプロジェクト設定を選択します。NUCLEO-F401RE という名前のプロジェクトの場合、図 45 に示した例のように入力します。

図 45. プロジェクト設定

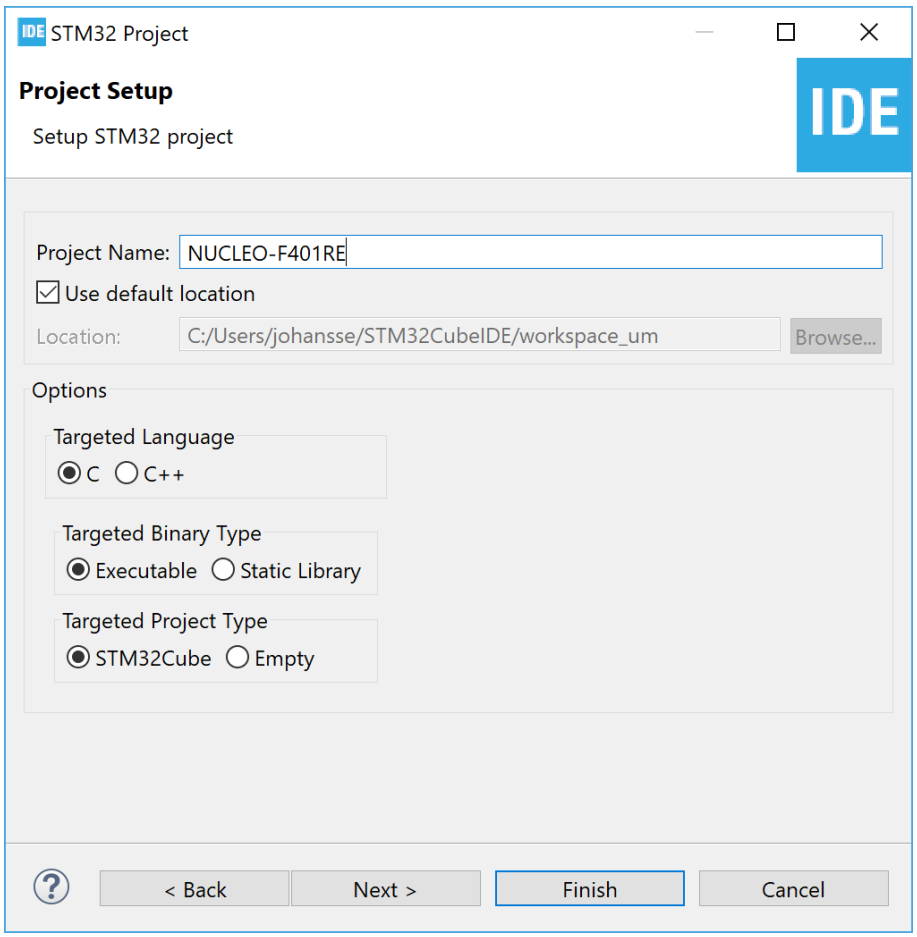
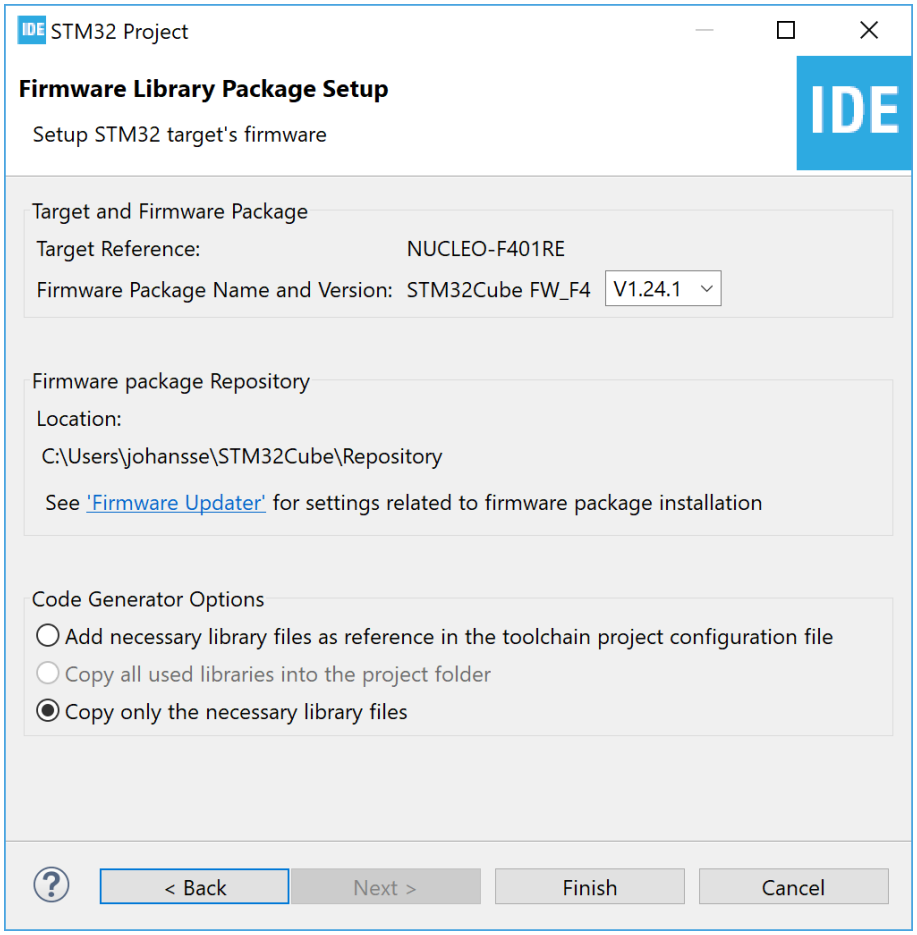


図 45 によると、プロジェクトは次のようなオプションで、デフォルトの場所に保存するように設定されています。

- C 言語のプロジェクト
- 実行可能バイナリ·タイプ
- STM32Cube をターゲットとするプロジェクト·タイプ

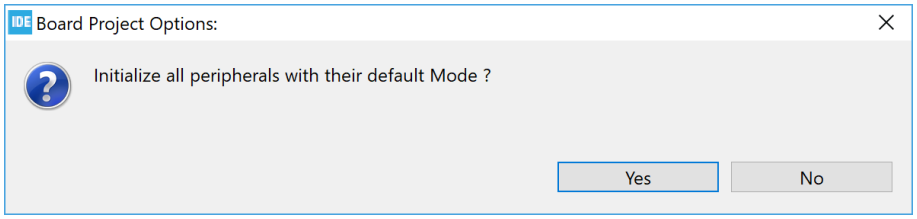
Next をクリックして[Firmware Library Package Setup]ページを開きます。

図 46. ファームウェア·ライブラリ·パッケージの設定



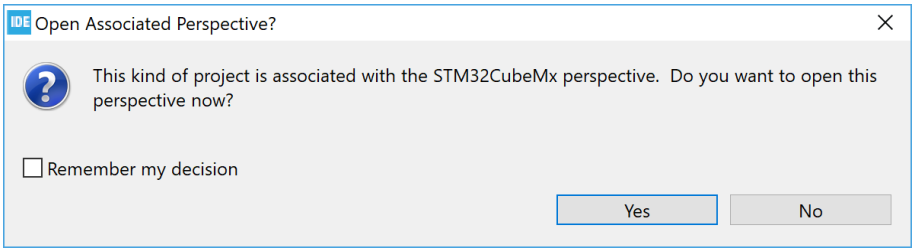
このページでは、プロジェクト作成時に使用する STM32Cube ファームウェア·パッケージを選択できます。この例ではデフォルト設定が使用されています。Finish をクリックしてプロジェクトを作成します。
その結果、次のようなダイアログが表示されます。

図 47. 全ペリフェラルの初期化



ペリフェラルの初期化に必要なソフトウェアを入手することは優れた実践行動であるため Yes をクリックしてください。
これによって、図 48 に示す新しいダイアログが開きます。

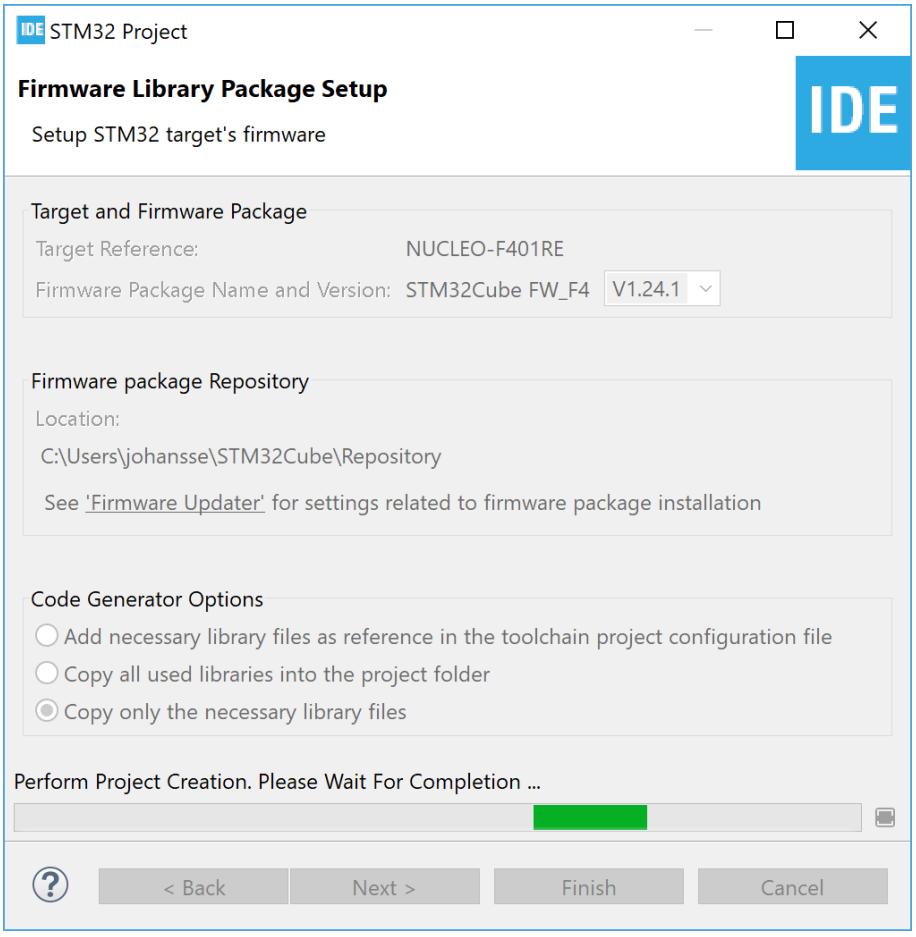
図 48. STM32CubeMX パースペクティブを開く



デバイスを設定する必要性がある場合、STM32CubeMX パースペクティブを開くことは妥当な判断です。次回、新規プロジェクトを作成する際に同じ質問を繰り返されないようにするには Remember my decision を有効にします。Yes をクリックして継続します。

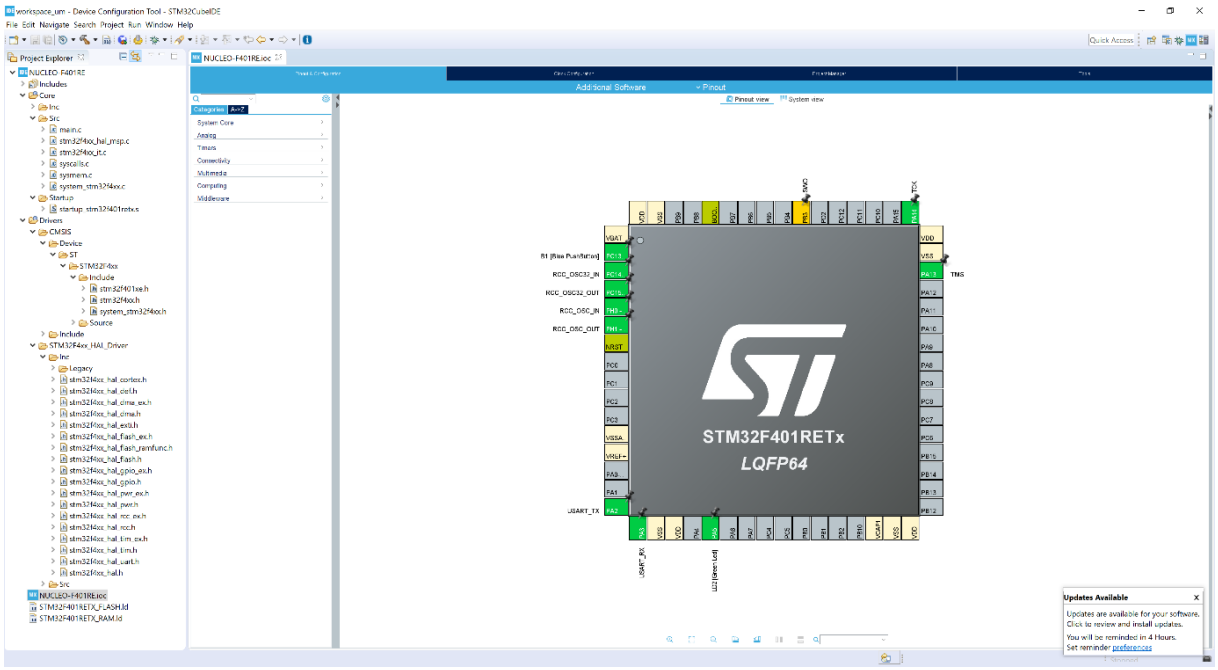
この時点でプロジェクトの作成が開始されます。所要時間は、プロジェクト作成のためにダウンロードする必要があるファイルの数によって決まります。

図 49. プロジェクト作成の開始



プロジェクトが作成されると、ペリフェラル、クロック、ミドルウェア、消費電力を設定するためのウィンドウを含む STM32CubeMX パースペクティブが開きます。

図 50. STM32CubeMX



新規プロジェクトは、含まれるいくつかのフォルダやファイルとともに[Project Explorer]ビューに表示されます。

NUCLEO-F401RE.ioc ファイルは構成設定を含み、STM32CubeMX のエディタで開きます。このエディタには、[Pinout & configuration]、[Clock configuration]、[Project manager]、[Tools]タブがあります STM32CubeMX エディタで変更を加えると、タブに表示されている .ioc ファイルに変更済みのマークが付きます。ファイルを保存すると、「Do you want to generate Code?」(コードを生成しますか)と尋ねるダイアログが表示されるので、新しいデバイス設定をサポートする新規コードをプロジェクト内に簡単に生成できます。STM32CubeMX エディタの使用方法の詳細については、[ST-14] を参照してください。

ターゲットとするプロジェクト·タイプとして STM32Cube ではなく Empty を選択すると(図 45. プロジェクト設定 参照)、含まれるファイルやフォルダの数が少ない STM32 プロジェクトを作成できます。Empty を選択した場合、生成されるプロジェクトに含まれるのは、いくつかのフォルダおよび Reset_Handler のコードとベクタ·テーブルを含むデバイス·スタートアップ·ファイル、main.c ファイル、その他いくつかの C ファイル、リンカ·スクリプト·ファイルのみです。STM32 のヘッダ·ファイル、システム·ファイル、CMSIS ファイルは手動で追加する必要があります。これらのファイルは、例えば STM32Cube をターゲットとする他のプロジェクト、または STM32 のサンプル·プロジェクトからコピーできます。

注 空のプロジェクトの場合、アプリケーションの要件に応じて浮動小数点ユニットの設定、つまりソフトウェア FPU を使用するかハードウェア FPU を使用するかの設定を忘れずに行ってください。ハードウェア FPU を使用するならば、FPU を初期化します。空ではないプロジェクトの場合、FPU の初期化は通常 system_stm32fxxx.c ファイル内の SystemInit 関数で行われます。FPU 設定が必要になる可能性を知らせるために、空のプロジェクトに作成される main.c ファイルには、次のような内容のコンパイラ警告が含まれます。#warning "FPU is not initialized, but the project is compiling for an FPU.Please initialize the FPU before use."(FPU が初期化されていませんがプロジェクトは FPU 向けにコンパイルされます。使用前に FPU を初期化してください。)

2.2.2 新しい STM32 スタティック·ライブラリ·プロジェクトの作成

STM32 スタティック·ライブラリ·プロジェクトの作成手順は、セクション 2.2.1 新しい STM32 実行可能プロジェクトの作成で説明したものと同様です。メニュー FileNew STM32 Project によって、STM32 プロジェクト·ウィザードを開き、ライブラリを作成するデバイスまたはボードを選択します。プロジェクト·ウィザードは、この選択に基づいて、ライブラリに適したコンパイラ·オプションの設定方法を判断します。図 51 に示すダイアログが表示されたら、この例の myLib のようなプロジェクト名を入力し、Static Library を選択してライブラリ·プロジェクトを作成します。

注 STM32 スタティック·ライブラリ·プロジェクトを作成する場合、適切なデバイスを厳密に選択することは重要ではありません。ただし、選択するデバイスは、ライブラリが想定しているデバイスと同じ Cortex® コア(Cortex®-M0+ や Cortex®-M7 など)を備えている必要があります。

図 51. STM32 スタティック・ライブラリ・プロジェクト

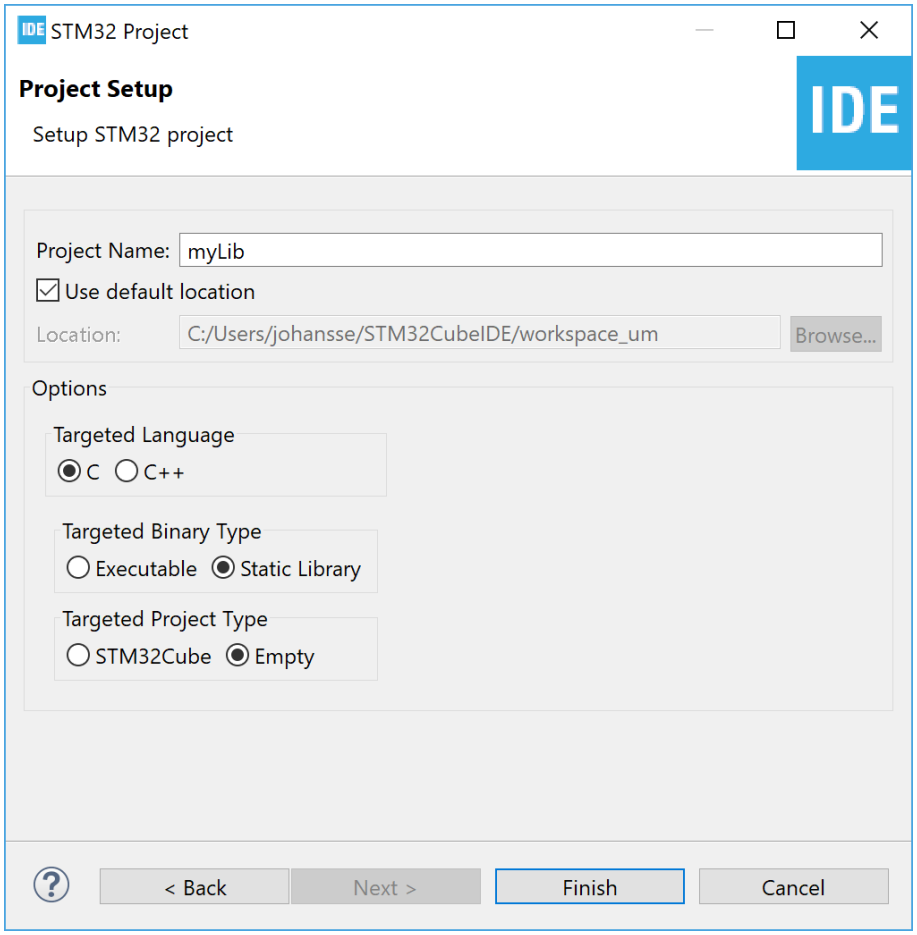


図 51 では、空の[Targeted Project Type]を選択しています。これに対して、プロジェクト・タイプとして STM32Cube を選択した場合、プロジェクトは STM32 のドライバも含めて生成されます。Finish をクリックしてプロジェクトを作成します。

プロジェクトが作成されると、Project Explorer にはこのライブラリ・プロジェクトと先ほど作成した NUCLEO-F401RE ボードのプロジェクトが表示されます。

図 52. STM32 のライブラリとボードのプロジェクト

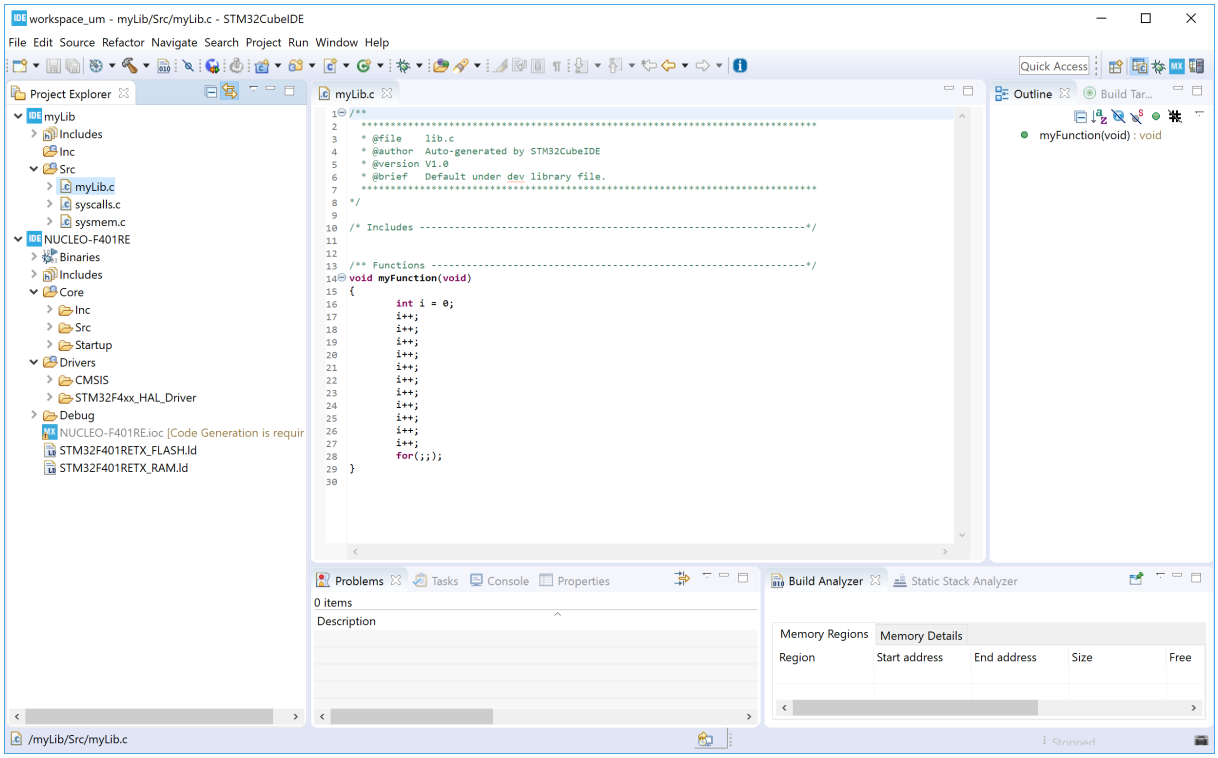


図 52 の[Project Explorer]ビューに示したように、myLib プロジェクトにはフォルダが `Inc` と `Src` しかありません。`Inc` フォルダはデフォルトでは空です。これはヘッダ·ファイルを追加する目的で用意されたフォルダです。`Src` フォルダには 3 つの C ファイルがあります。`<myLib.c>` ファイルは、ライブラリに含める必要があるライブラリ関数を記述するために編集するファイルです。ライブラリ·ファイルの名前は、プロジェクト名と同じです。他の 2 つの C ファイルは、必要な場合に Newlib が使用し、必要に応じて更新が可能です。

`myLib.c` ファイル内の関数名は変更できます。`myLib.c` ファイルには、他にも関数を追加できます。さらに大規模なライブラリを作成する場合は、他の C ファイルの追加も可能です。アプリケーションから呼び出し可能なライブラリ関数のプロトタイプを含むヘッダ·ファイルを作成することをお勧めします。

注 ライブラリ·プロジェクト·フォルダにはリンカ·スクリプトは一切含まれていません。ライブラリ·プロジェクトをビルドする場合、C ファイルについてはコンパイラとして `gcc` が使用され、アーカイブ·ファイルの作成には `ar` が使用されます。このアーカイブ·ファイルは、他の実行可能プロジェクトにリンクして使用できます。メモリ設定については適切なデバイスを選択することは重要ではありません。デバイス名は Cortex® コアを、ライブラリが適用対象と想定しているものに設定するために使用されます。これによって、コンパイラは適切な命令を生成します。

2.3 プロジェクトのビルド設定の設定

作成した STM32 プロジェクトには、デフォルトの C/C++ ビルド設定が含まれます。しかし、GCC には使用できる各種オプションが多数存在し、組込みシステムごとに固有の要件があります。そのため、プロジェクトのビルド設定はデフォルトからさらに詳細な設定が可能です。

また、プロジェクト開発の段階ごと、例えばデバッグとリリースの段階ではビルド設定の要件が異なるのが普通です。このような状況に備え、STM32CubeIDE は、プロジェクトごとに異なるビルド設定を使用できます。このセクションでは、まずビルド設定の概要、続いてその設定方法について紹介します。

2.3.1 プロジェクトのビルド設定

ビルド設定ごとに特定のビルド設定を含めることで、プロジェクトのさまざまなバリアントを使用できます。STM32CubeIDE で STM32 プロジェクトを作成すると、デフォルトで 2 つのビルド設定、Debug と Release が作成されます。Debug 構成でプロジェクトをビルドすると、デバッグ情報が含まれ、最適化は一切実行されません。Release 構成でプロジェクトをビルドすると、コード·サイズを削減する最適化が実行され、デバッグ情報は一切含まれません。プロジェクト作成時のデフォルトでは、アクティブなビルド設定として Debug が設定されます。

プロジェクトのビルド設定はいつでも新規作成できます。そのような新しいビルド設定は、既存のビルド設定に基づいて作成できます。

プロジェクトをビルドする場合、アクティブなビルド設定が適用され、ビルド中に生成されるファイルはアクティブなビルド設定と同じ名前のフォルダに書き込まれます。

注　ビルド設定が対応するのはビルド設定だけです。デバッグ設定の設定方法については後述します。

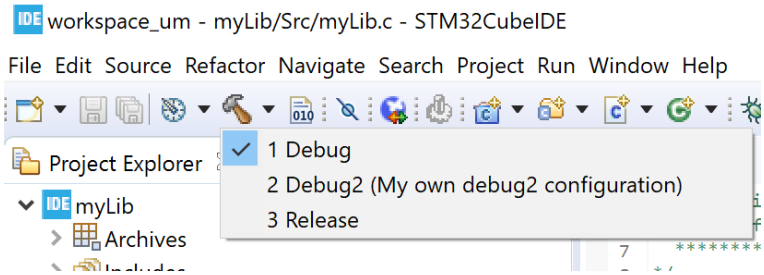
2.3.1.1 *アクティブなビルド設定の変更*

アクティブなビルド設定を変更するには、次の手順を実行します。

1. Project Explorer でプロジェクト名を選択します。
2. C/C++ パースペクティブのツールバーを使用し、Build ツールバー·ボタン  の右にある矢印をクリックします。
3. ビルド設定のリストが表示されます。

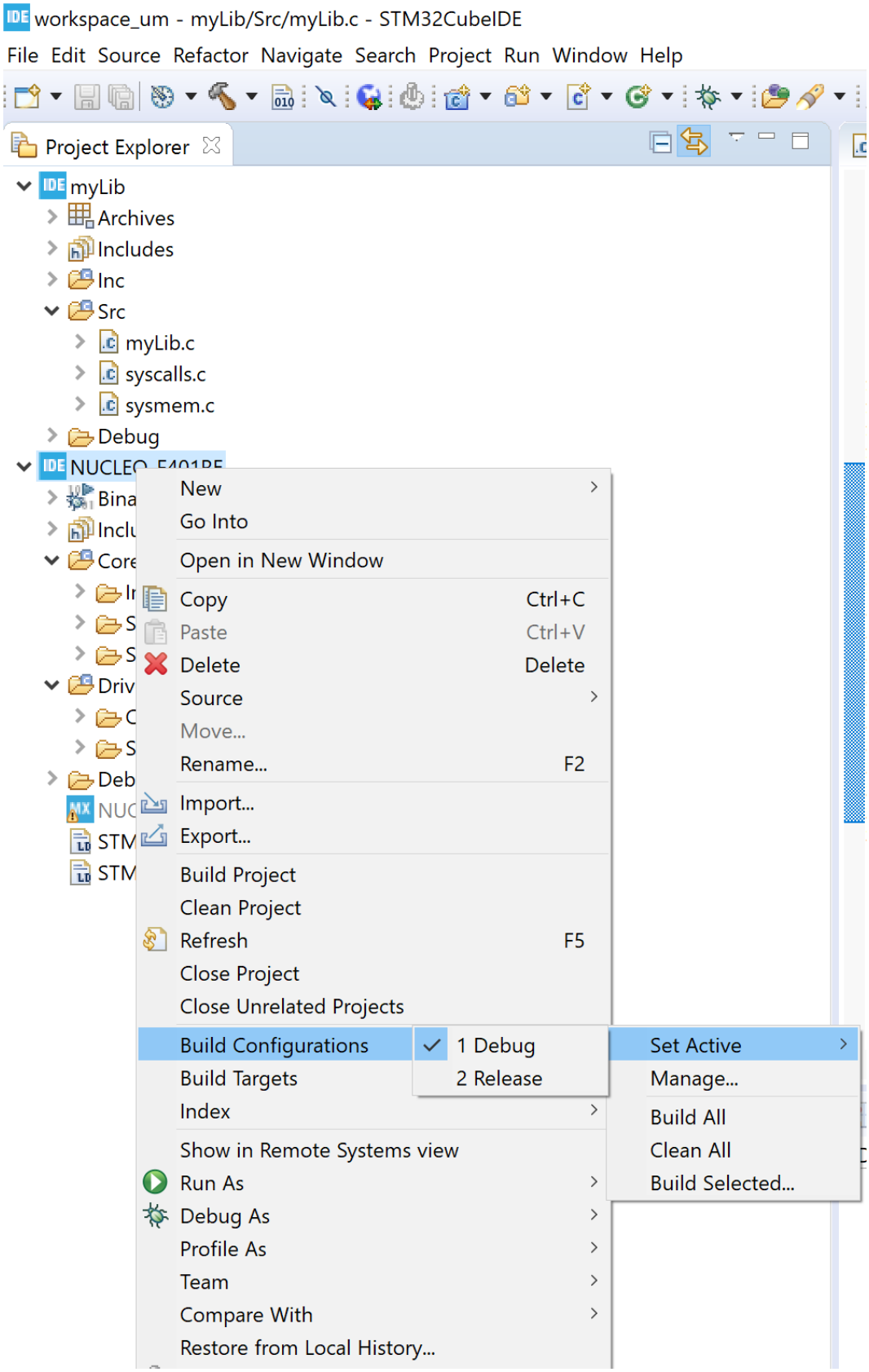
このリストから使用するビルド設定を選択します。

図 53. ツールバーによるアクティブなビルド設定のセット



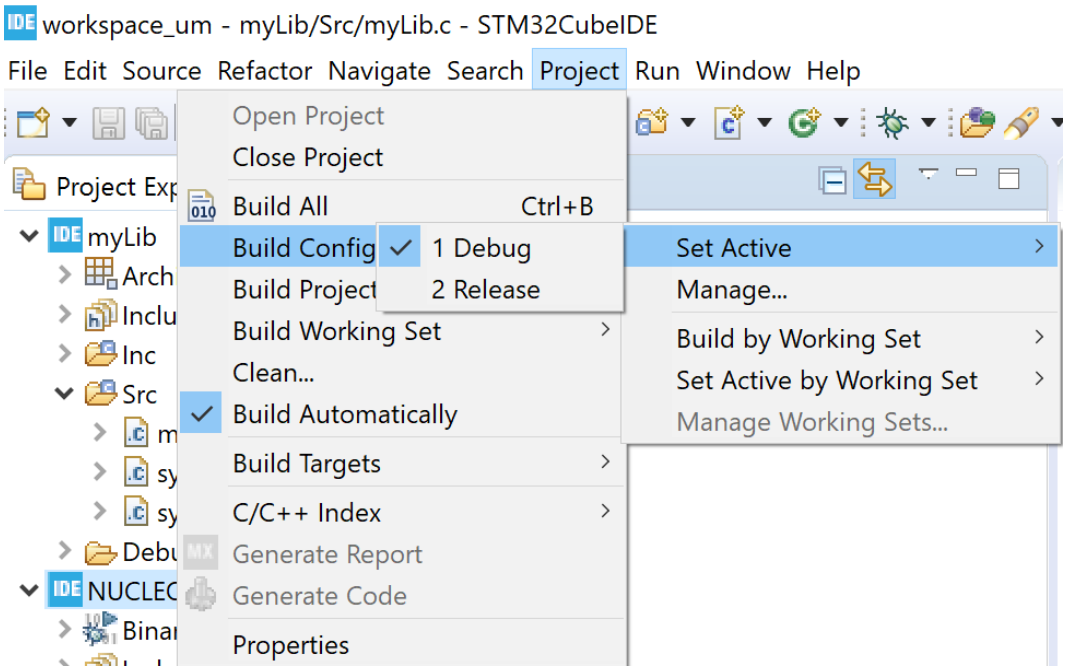
アクティブなビルド設定を変更するもう一つの方法は、[Project Explorer]ビューのプロジェクト名を右クリックし、Build ConfigurationsSet Active を選択してから、目的とするビルド設定を選択します。

図 54. 右クリックによるアクティブなビルド設定のセット



アクティブなビルド設定は、メニュー ProjectBuild ConfigurationsSet Active から選択することも可能です。

図 55. メニューによるアクティブなビルド設定のセット



2.3.1.2 *新しいビルド設定の作成*

ビルド設定を新規作成するには、次の手順を実行します。

1. [Project Explorer]ビューでプロジェクト名を右クリックします。
2. 次のいずれかを実行します。
   - Build ConfigurationsManage… を選択します。
   - メニュー ProjectBuild ConfigurationsManage… を使用します。

いずれの方法でも、[Manage Configurations]ダイアログが表示されます。

図 56. [Manage Configurations]ダイアログ

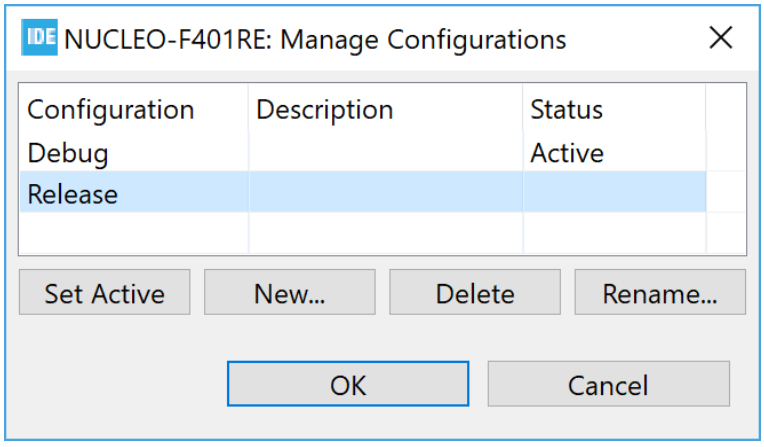


図 56 に示したダイアログのボタンを使用して、ビルド設定を管理します。

- Set Active は、別のビルド設定を選択してそれをアクティブにするときに使用します。
- New... は、ビルド設定の新規作成に使用します。
- Delete は、既存のビルド設定の削除に使用します。
- Rename... は、ビルド設定の名前を変更するときに使用します。

ビルド設定を新規作成するには、New... ボタンをクリックします。これによって[Create New Configuration]ダイアログが表示されます。このダイアログに名前と説明を入力します。名前はディレクトリ名として有効なものである必要があります。新しい構成でプロジェクトをビルドする際に、この名前がディレクトリ名として使用されるからです。

図 57. ビルド設定の新規作成

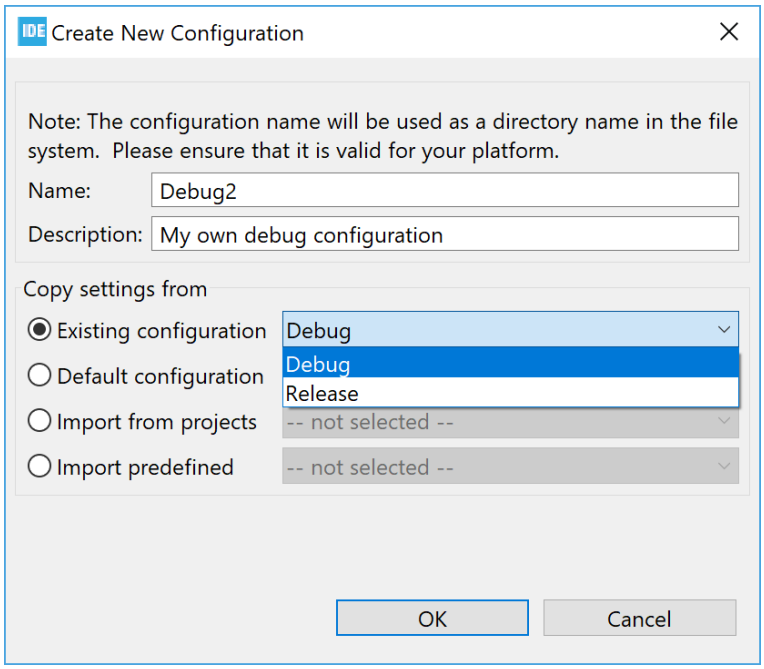
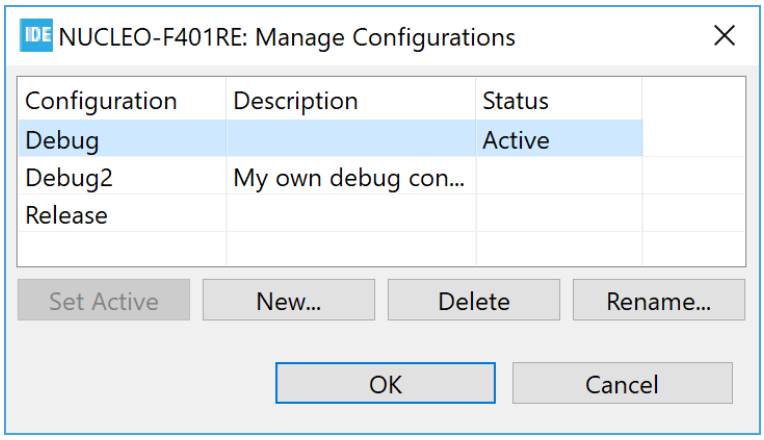


図 57 に示すように、新しいビルド設定は、既存のビルド設定に基づいて作成します。この図に示した新規構成例は、既存の Debug 構成に基づいています。設定が完了したら OK をクリックします。
[Manage Configurations]ダイアログが開き、新しいでデバッグ設定が表示されます。

図 58. 更新された[Manage Configurations]ダイアログ



必要に応じて別の構成をアクティブ化し、構成の管理作業が終了したら OK をクリックして変更を保存し、構成ダイアログを終了します。

2.3.1.3 *ビルド設定の削除*

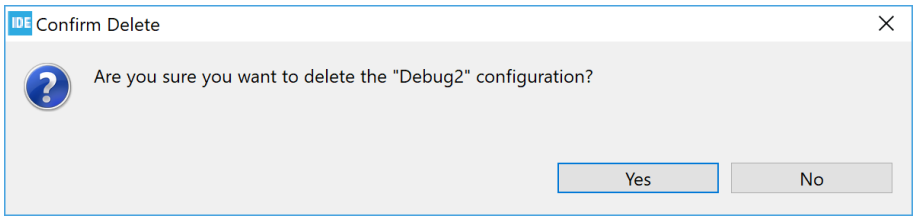
ビルド設定を削除するには、次の手順を実行します。

1. [Manage Configurations]ダイアログを開きます。
2. 削除する構成を選択します。

3. Delete ボタンをクリックします。

例えば、Debug2 構成を選択して Delete ボタンをクリックすると、次のような確認ダイアログが表示されます。

図 59. 構成削除のダイアログ



ここでは No を選択して Debug2 構成は残しておきます。

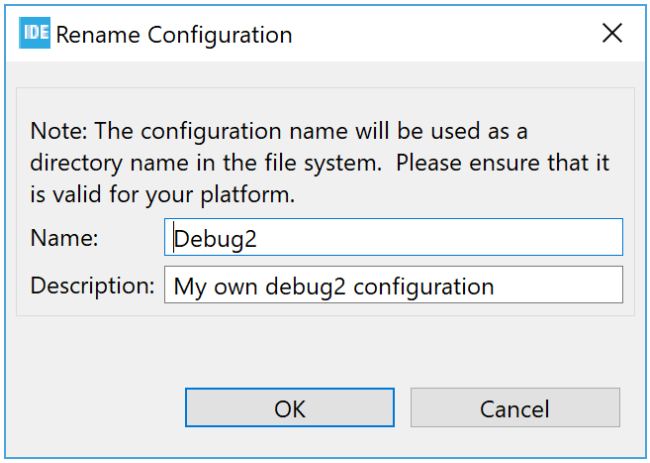
2.3.1.4 *ビルド設定の名前の変更*

ビルド設定の名前を変更するには、次の手順を実行します。

1. [Manage Configurations]ダイアログを開きます。
2. 名前を変更する構成を選択します。
3. Rename... ボタンをクリックします。

例えば、Debug2 構成を選択して Rename... ボタンをクリックすると、次のような確認ダイアログが表示されます。

図 60. 構成の名前変更のダイアログ



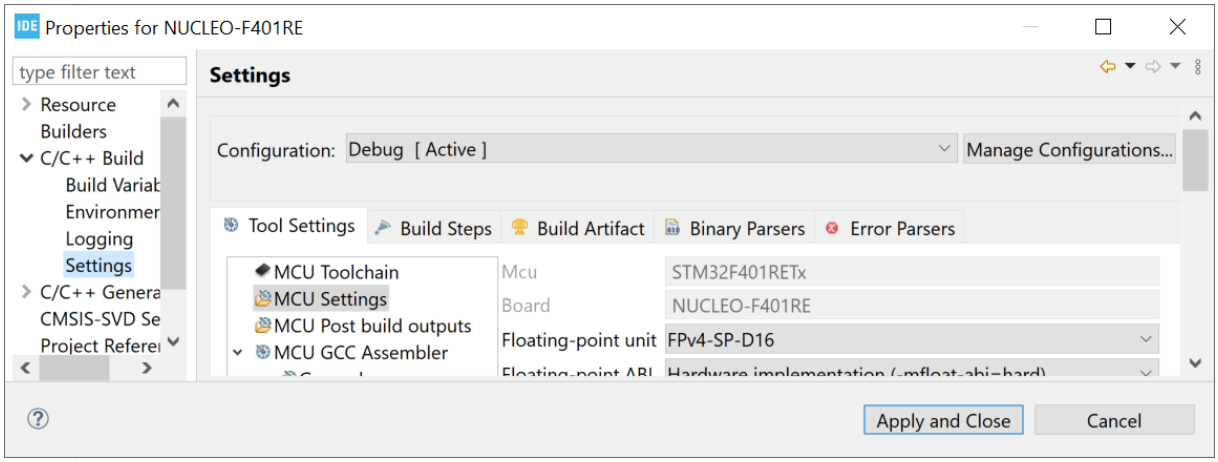
構成の名前、説明、または両方を変更し OK をクリックして Debug2 構成の名前を変更します。ここでは、Cancel を選択して名前は変更しません。

2.3.2 プロジェクト C/C++ ビルド設定

各ビルド設定には、1 つのプロジェクト C/C++ ビルド設定が含まれます。プロジェクト C/C++ ビルド設定は、プロジェクトのプロパティで変更します。ビルド設定を変更するには、[Project Explorer]ビューのプロジェクト名を右クリックして Properties を選択するか、メニュー ProjectProperties を使用します。いずれの方法でも、プロジェクトの[Properties]ウィンドウが開きます。

[Properties]の左ペインで C/C++ BuildSettings を選択します。右側に[Tool Settings]、[Build Steps]、[Build Artifact]、[Binary Parsers]、[Error Parsers]の各タブが表示されます。最初の 2 つのタブが最も役に立ちます。

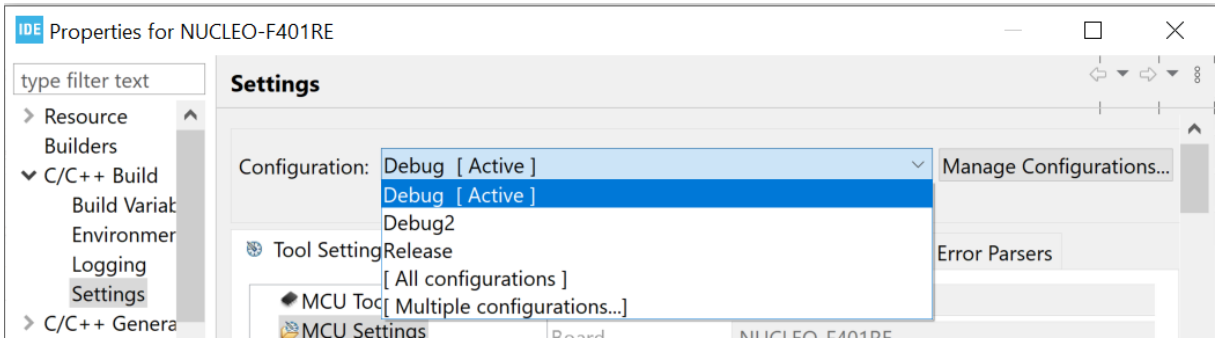
図 61. プロパティのタブ



注 すべてのタブが表示されていない場合は、ダイアログ・ウィンドウのサイズを変更するか右上の矢印ボタンをクリックします。

[Settings]ペインには Configuration 選択フィールドがあり、新たに選択した設定を、アクティブ構成のみ、別の構成、すべての構成、複数の構成のいずれに適用するかを指定できます。Manage Configurations をクリックすると[Manage Configurations]ダイアログが開きます。

図 62. 構成のプロパティ

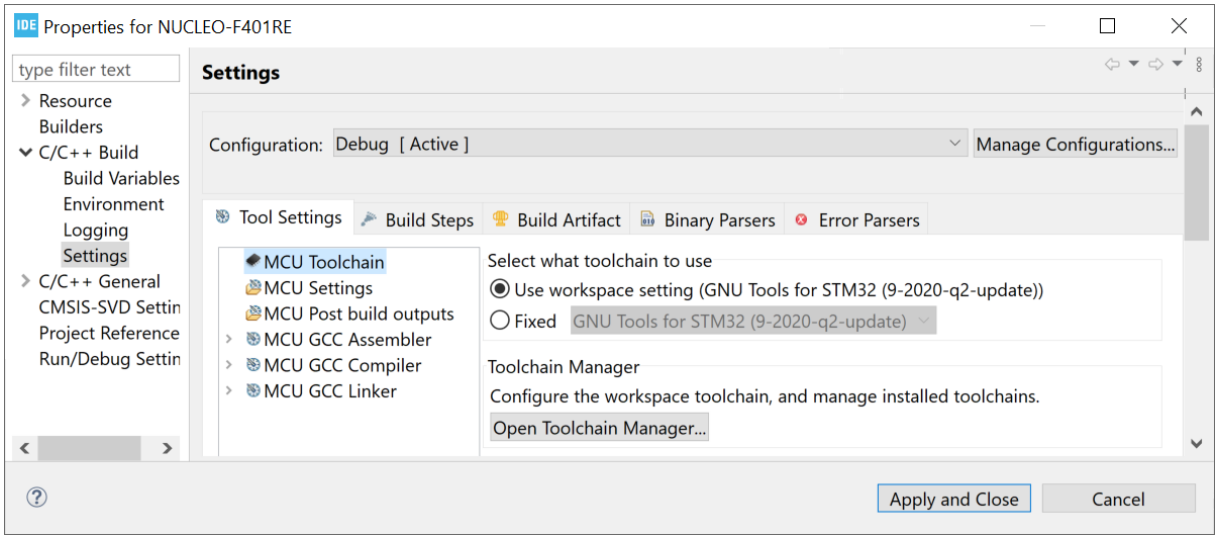


[Tool Settings]タブは、さらに[MCU Toolchain]、[MCU Settings]、[MCU Post build outputs]、[MCU GCC Assembler]、[MCU GCC Compiler]、[MCU GCC Linker]に分かれます。

[MCU Toolchain]は、ツールチェーンを変更するときに使用します。STM32CubeIDE には、いずれかのバージョンの GNU Tools for STM32 が付属します。[Toolchain Manager]は、他の GNU ARM Embedded ツールチェーンをダウンロードし、ローカルの GNU ARM Embedded ツールチェーンを使用する設定を行うために使用します。

GNU Tools for STM32 に適用したパッチに関する情報は、[EXT-12] を参照してください。ドキュメントは Information Center の[Technical Documentation]ページから開くことができます。

図 63. ツールチェーン·バージョンのプロパティ



ツールチェーン選択を有効にするには Fixed を選択します。

図 64. ツールチェーン選択のプロパティ

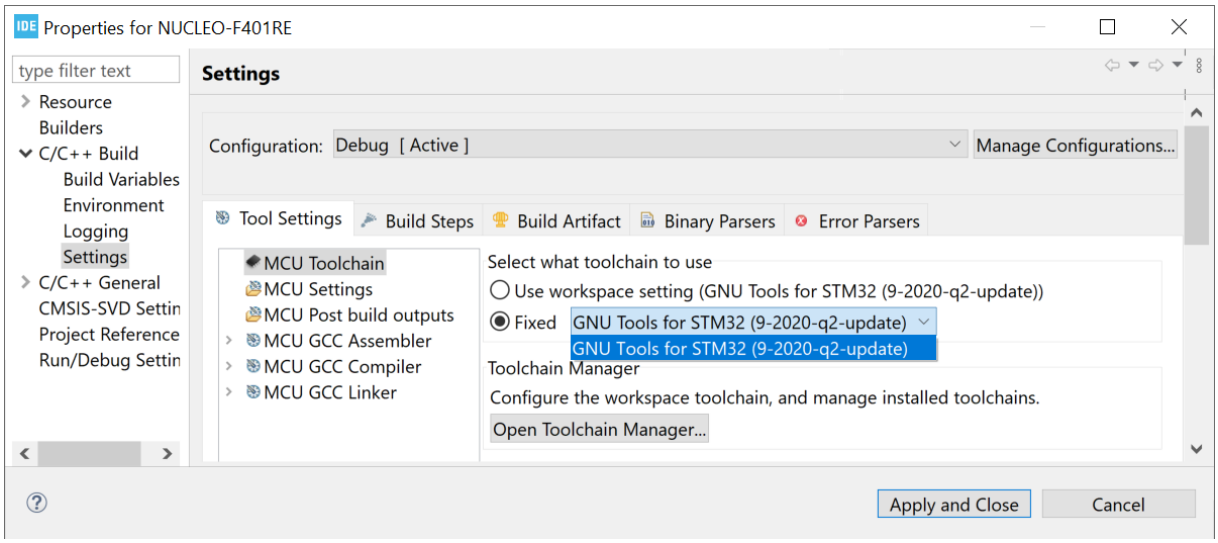
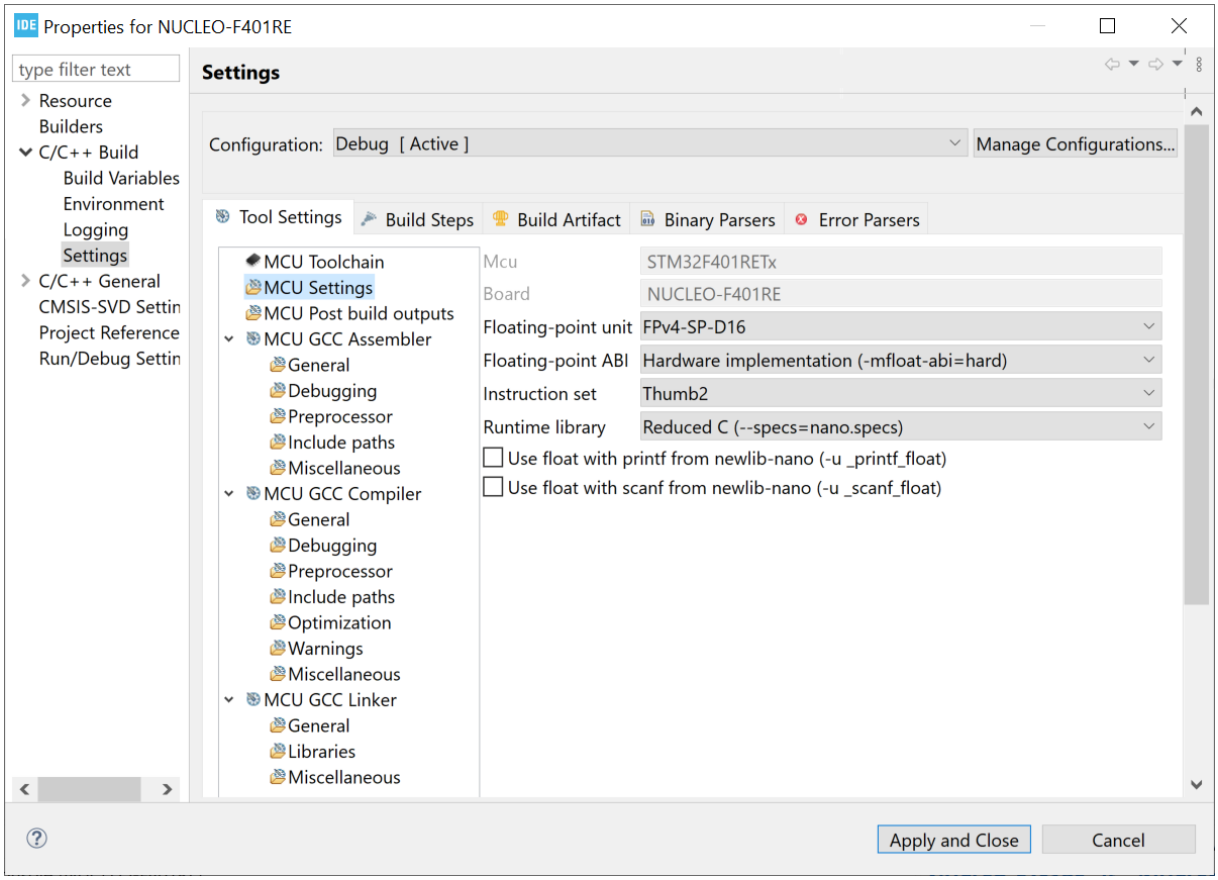


図 64 に示したように、デフォルト設定の場合、使用できるのはデフォルトのツールチェーン GNU Tools for STM32 だけです。その他のツールチェーンをインストールするには、Open Toolchain Manager... ボタンをクリックして Toolchain Manager を開きます。セクション 2.11 Toolchain Manager には、ツールチェーンのインストールとアンインストールの方法、デフォルトのワークスペース·ツールチェーンの選択方法が詳しく記載されています。

[MCU Settings]には、プロジェクトで選択しているマイクロコントローラやボードが表示され、浮動小数点、命令セット、ランタイム·ライブラリの処理方法の選択肢を提示します。

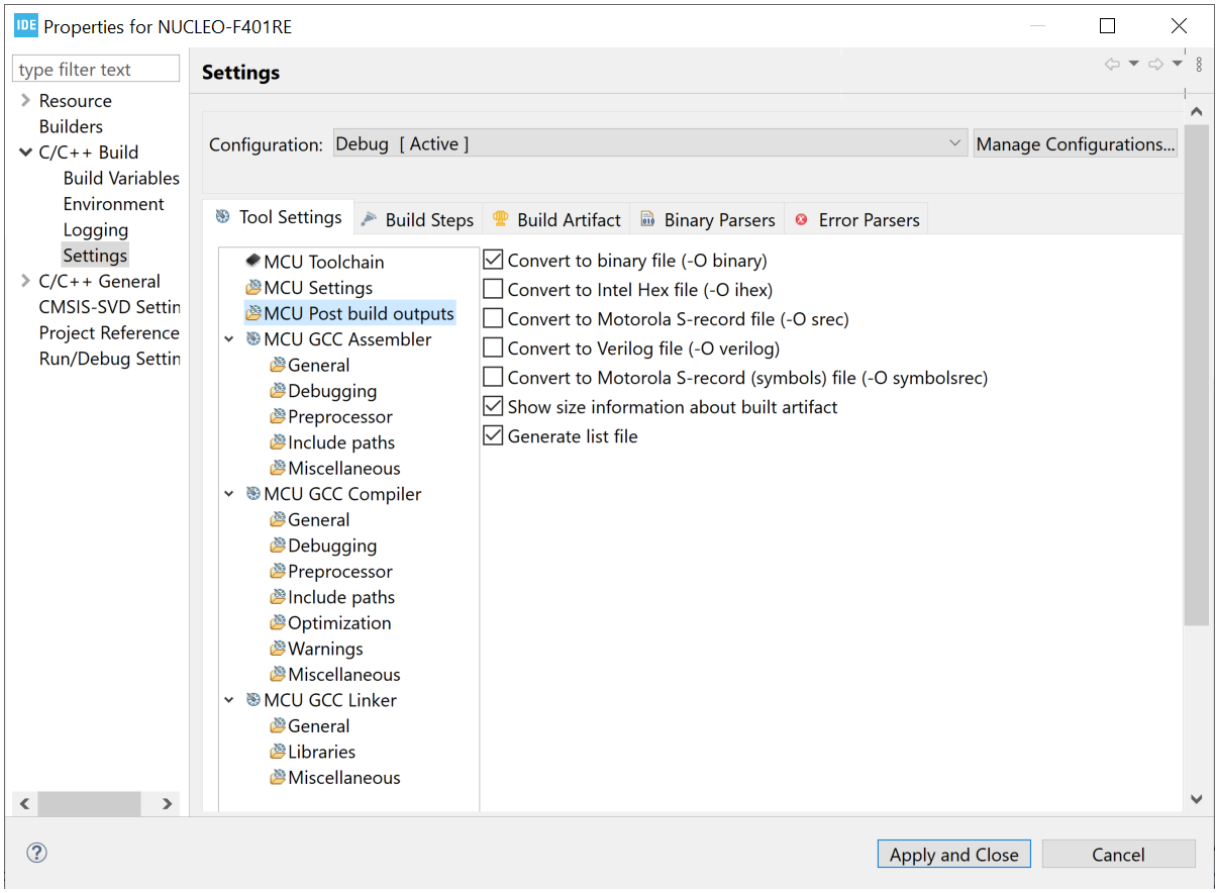
図 65. プロパティ·ツール - マイコンの設定



[MCU Post build outputs]は、`elf` ファイルの別フォーマットへの変換、ビルド·サイズ情報の表示、リスト·ファイルの生成などの選択肢を提示します。出力ファイルは、次のファイルに変換できます。

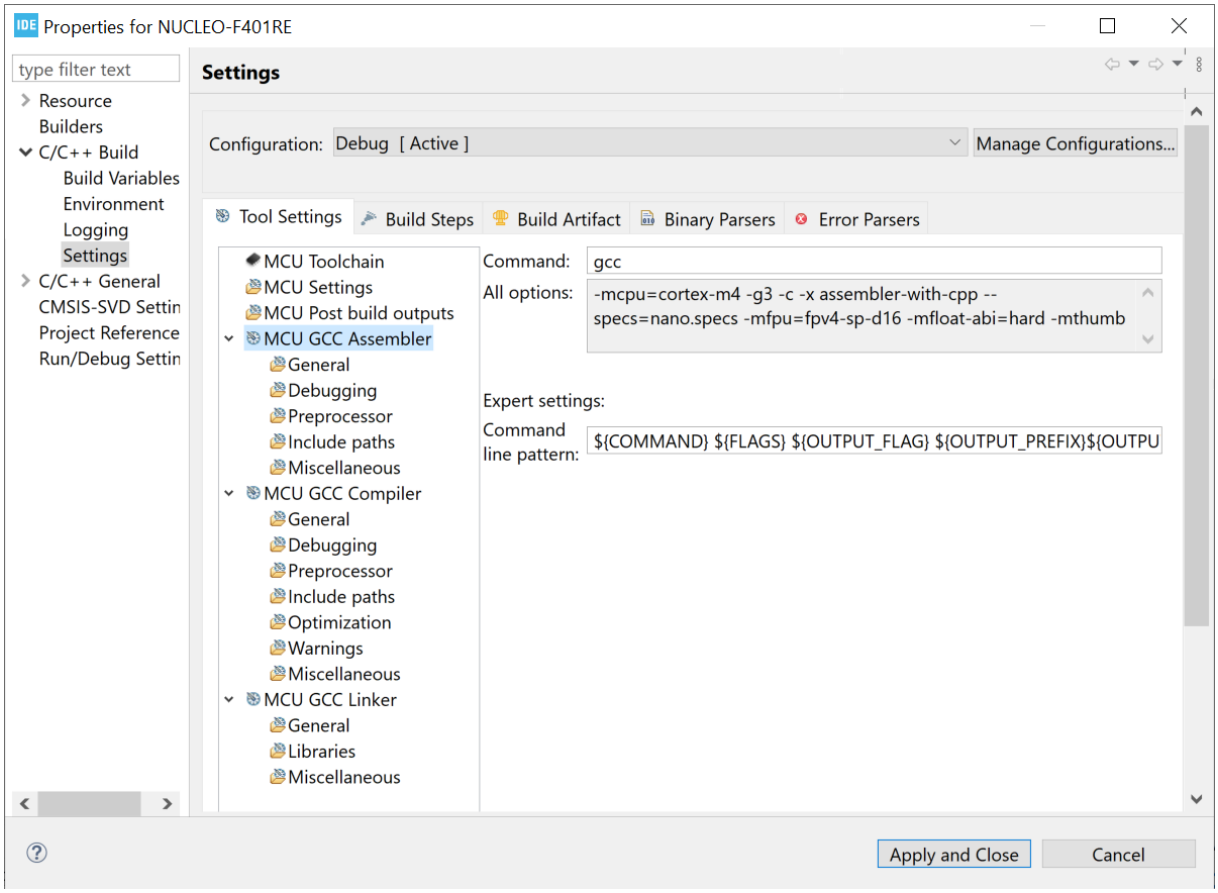
- バイナリ·ファイル
- Intel Hex ファイル
- Motorola S-record ファイル
- Motorola S-record シンボル·ファイル
- Verilog ファイル

図 66. プロパティ·ツール - マイコンのビルド後出力の設定



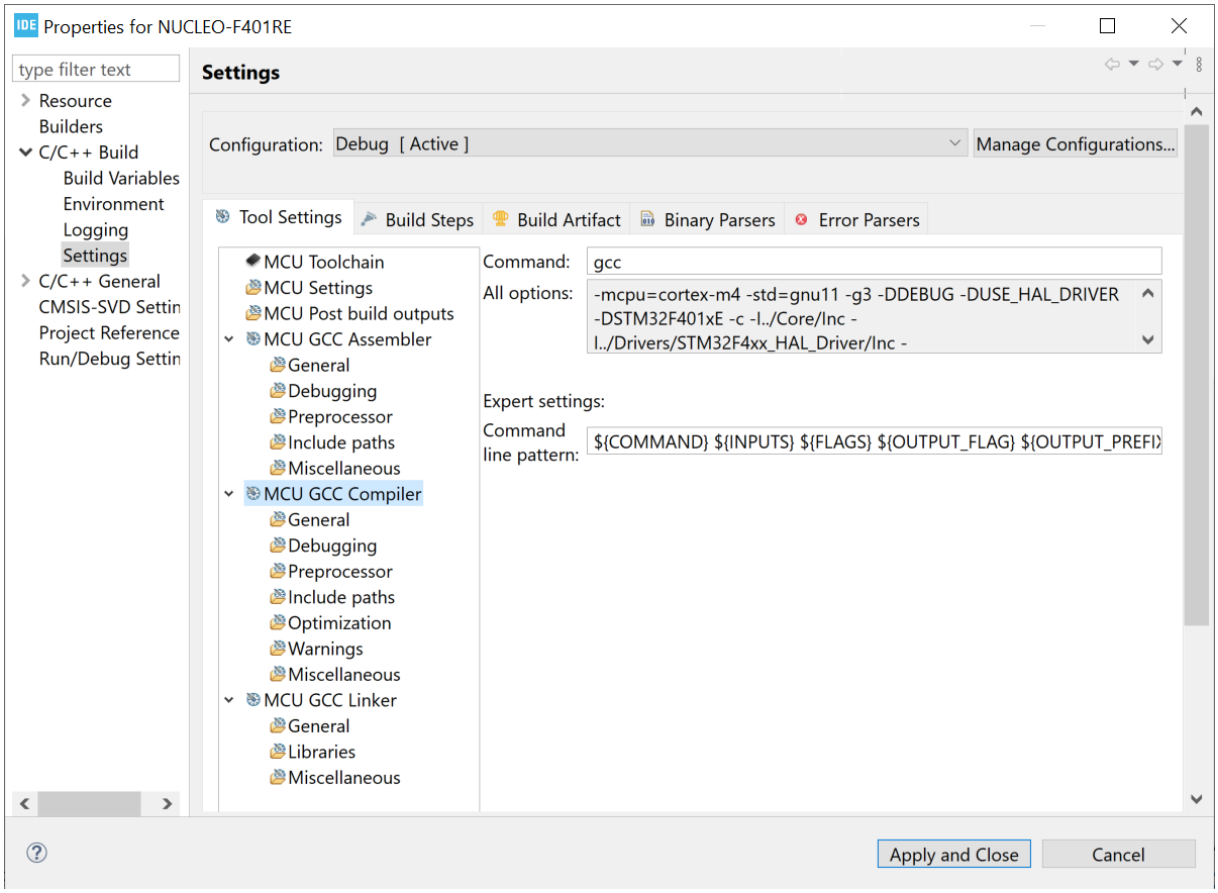
[MCU GCC Assembler]の設定では、アセンブラを選択できます。メイン·ノードを選択すると、サブノード設定で現在有効化されているアセンブラ·コマンドライン·オプションがすべて表示されます。サブノードは、現在の設定を表示するときや、アセンブラのいずれかの設定を変更するときに使用します。

図 67. プロパティ·ツール - GCC アセンブラの設定



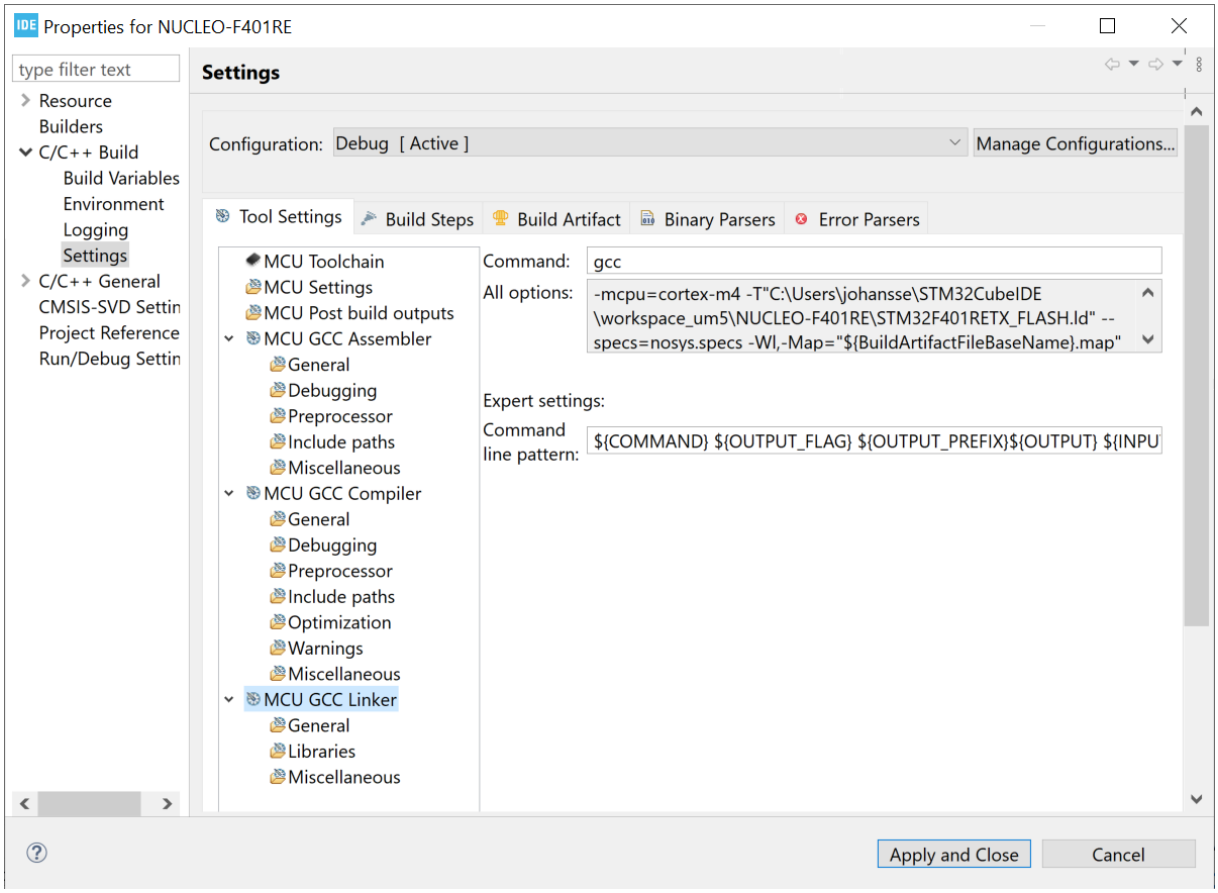
[MCU GCC Compiler]の設定では、コンパイラを選択できます。メイン·ノードを選択すると、サブノード設定で現在有効化されているコンパイラ·コマンドライン·オプションがすべて表示されます。サブノードは、現在の設定を表示するときや、コンパイラのいずれかの設定を変更するときに使用します。

図 68. プロパティ·ツール - GCC コンパイラの設定



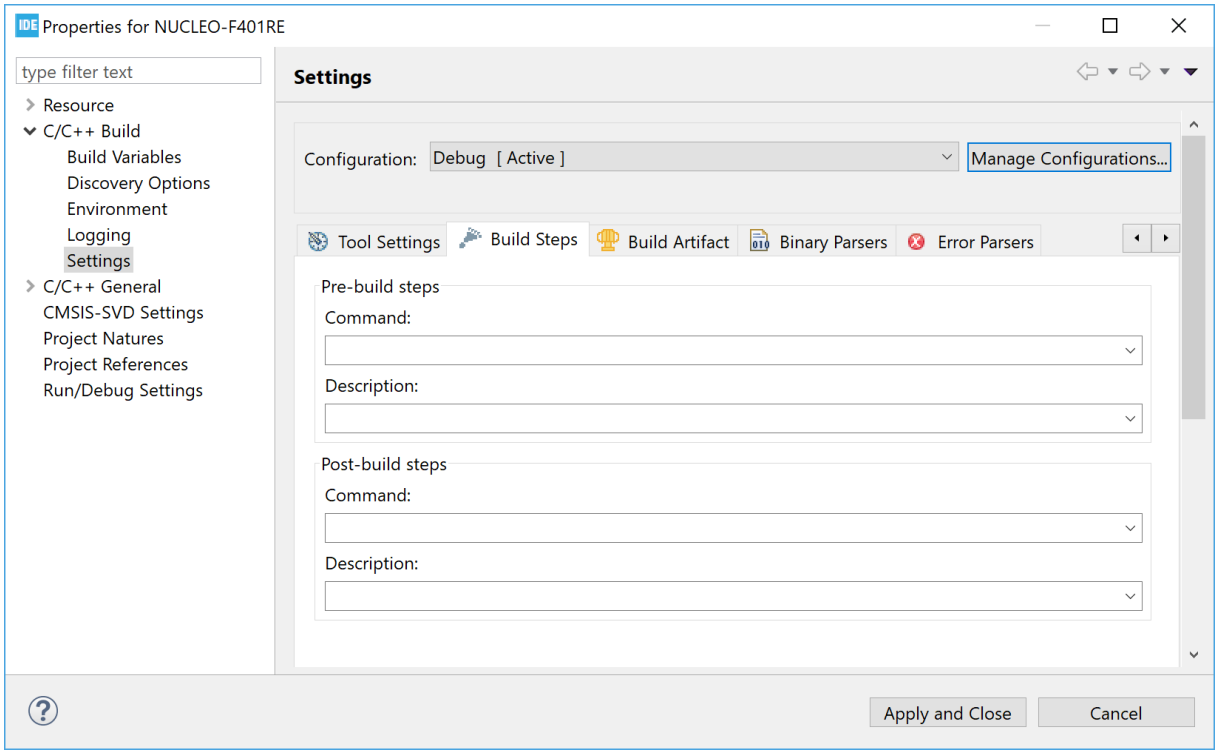
[MCU GCC Linker]の設定では、リンカを選択できます。メイン·ノードを選択すると、サブノード設定で現在有効化されているリンカ·コマンドライン·オプションがすべて表示されます。サブノードは、現在の設定を表示するときや、リンカのいずれかの設定を変更するときに使用します。

図 69. プロパティ·ツール - GCC リンカの設定



[Build Steps]の設定には、ビルド前およびビルド後のステップを指定するために使用するフィールドが含まれます。これらのステップはプロジェクトのビルド前後に実行されます。ビルド前後に実行するステップがある場合は、このフィールドを編集します。

図 70. ビルド·ステップ設定のプロパティ

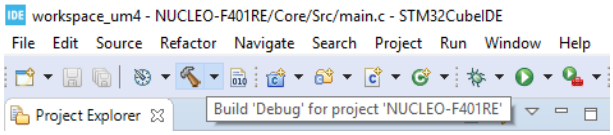


注　セクション 2.4.7　で説明する手順に従って makefile のターゲットを使用すると、より高度なビルド後動作を追加できます。

2.4 プロジェクトのビルド

ビルドを開始するには、[Project Explorer]ビューの対応するプロジェクトを選択し、Build ツールバー·ボタン  をクリックします。

図 71. プロジェクト·ビルド·ツールバー

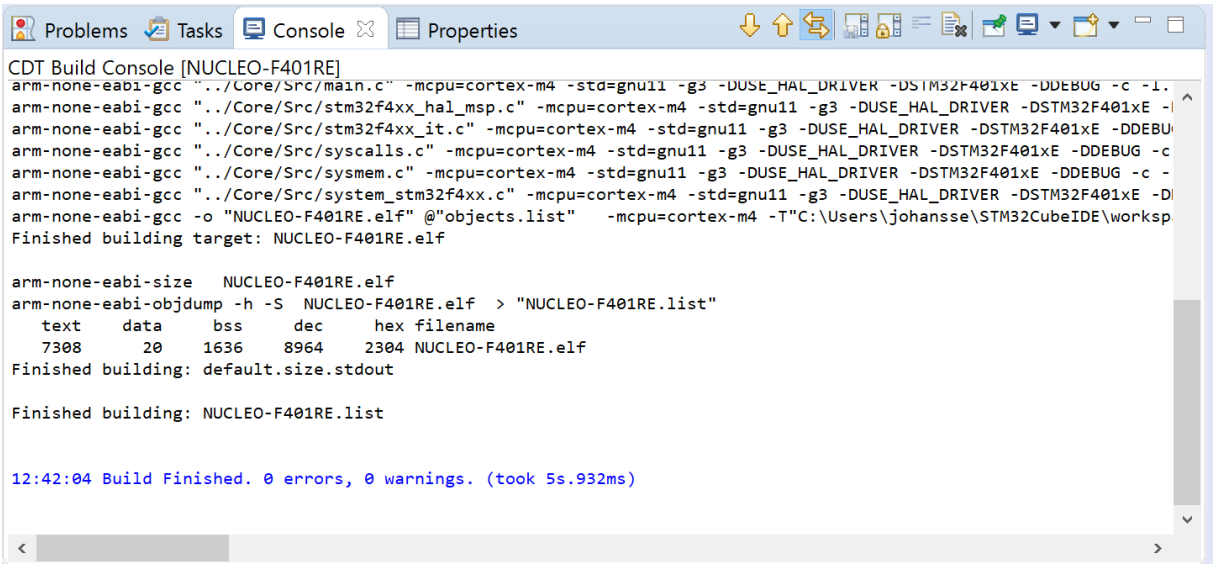


ビルドは、メニュー ProjectBuild Project からも開始できます。Project メニューには、他にも Build All、Build Project、Clean などの便利なビルド·コマンドがあります。

ビルドを開始する、もう一つの方法では[Project Explorer]ビューのプロジェクトを右クリックします。これによって、Build コマンドのほか、いくつかのビルド·オプションを含むコンテキスト·メニューが開きます。

ビルド中は、[Console]ビューにビルド·プロセスが表示されます。最後に、`elf` ファイルが正常に作成されると、サイズ情報が表示されます。

図 72. プロジェクト·ビルド·コンソール

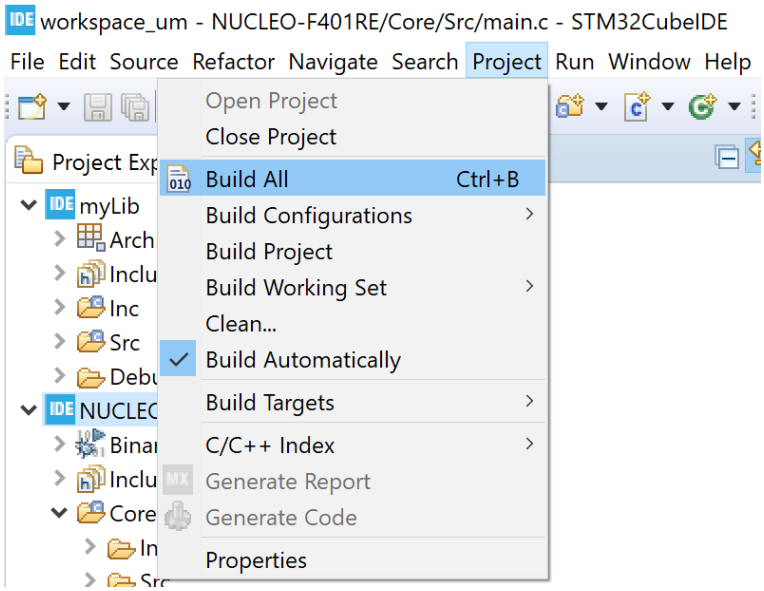


2.4.1 全プロジェクトのビルド

ツールバーには Build all ボタン  があります。これは、ワークスペース内で開かれているすべてのプロジェクトを、アクティブなビルド設定でビルドするときに使用します。

メニュー ProjectBuild All からも全プロジェクトのビルドは開始できます。

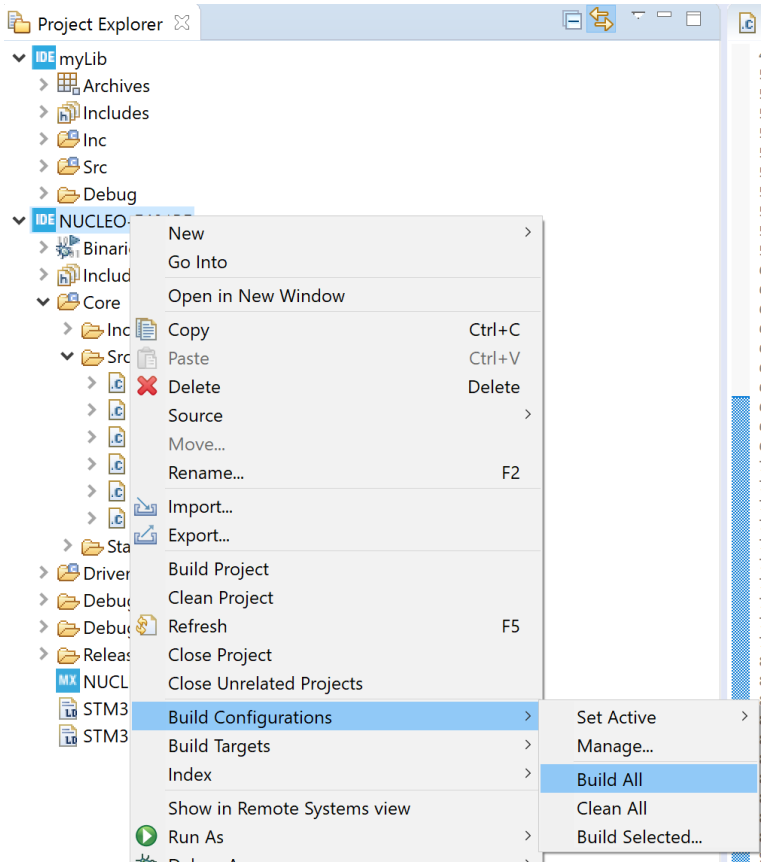
図 73. 全プロジェクトのビルド



2.4.2 全ビルド設定によるビルド

1 つのプロジェクトに対して、すべてのビルド設定でビルドを実行するには、プロジェクトを右クリックして、コンテキスト·メニューの Build ConfigurationsBuild All を選択します。

図 74. 全ビルド設定によるプロジェクトのビルド



2.4.3 ヘッドレス·ビルド

ヘッドレス·ビルドの使用目的は、スクリプト制御のビルドに組み込む必要があるプロジェクトのビルドです。連続統合プロセスや、その他の方法により、ビルド·サーバ上で終夜ビルドを行う場合などです。この場合、STM32CubeIDE の GUI は一切表示されず、ユーザは STM32CubeIDE の手動操作を一切求められません。

STM32CubeIDE には、ヘッドレス·ビルドを実行するための `headless-build` コマンド·ファイルが付属しています。このファイルは、例えば Windows® を使用している場合、STM32CubeIDE のインストール·フォルダ `C:\ST\STM32CubeIDE_1.7.0\STM32CubeIDE` にあります。`headless-build.bat` ファイルは、コマンド·プロンプトからの実行に使用するバッチ·ファイルです。

注 ヘッドレス·ビルドの実行前は、STM32CubeIDE によってワークスペースが開かれていないことを毎回必ず確認してください。ワークスペースを使用して既に STM32CubeIDE が実行されていると、ヘッドレス·ビルド·プロセスがプロジェクトを開いてビルドできないからです。

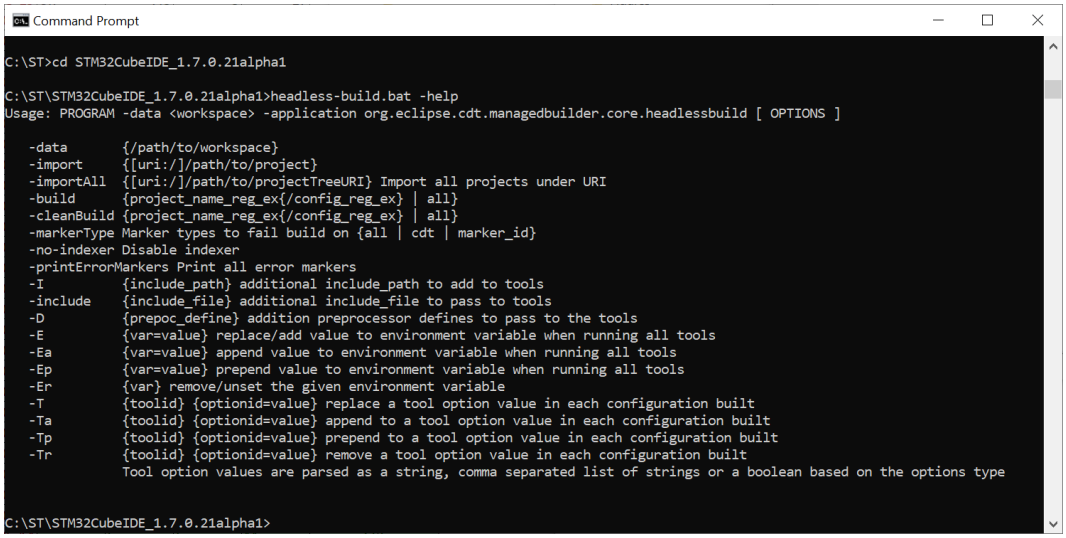
Windows® でヘッドレス·ビルドを実行するには、次の手順を実行します。

1. コマンド·プロンプトを開きます。
2. STM32CubeIDE のインストール·ディレクトリに移動します。IDE が保存されているフォルダを開きます。
   入力例：`cd C:\ST\STM32CubeIDE_1.7.0\STM32CubeIDE`
3. 次のコマンドを入力して、ワークスペース内で NUCLEO-F401RE プロジェクトをビルドします。

```
C:\Users\Name\STM32CubeIDE\workspace_1.7.0:
$ headless-build.bat -data C:\Users\Name\STM32CubeIDE\workspace_1.7.0
-cleanBuild NUCLEO-F401RE
```

ヘッドレス·ビルドのパラメータに関するヘルプを表示するには、`-help` オプションを付けてコマンドを実行します。図 75 に、コマンド `$ headless-build.bat -help` の実行結果を示します。

図 75. ヘッドレス·ビルド



2.4.4 一時アセンブリ·ファイルと前処理済み C コード

一時アセンブリ·ファイルを保存するには、コンパイラに `-save-temps` フラグを付加します。

1. メニューの ProjectProperties を選択します。
2. C/C++ buildSettings を選択します。
3. [Tool Settings]タブを開きます。
4. C CompilerMiscellaneous 設定に `-save-temps` を追加します。
5. プロジェクトを再度ビルドします。

アセンブラのファイルは、`filename.s` という名前でビルド出力ディレクトリに保存されます。

前処理済み C コードを含むファイル `filename.i` も生成されます。このファイルでは、プリプロセッサによる処理が完了したものの、まだコンパイルはされていないコードを確認できます。定義関連の問題が生じた場合、このファイルの内容を調査することをお勧めします。

2.4.5 ビルドのログ

プロジェクトのビルドのログ記録を有効または無効にするには、[Project Explorer]ビューのプロジェクトを右クリックして、Properties を選択します。次に C/C++ BuildLogging を選択します。ログ·ファイルの保存場所と名前も指定します。

ワークスペース内の全プロジェクトについて、グローバルなビルド·ログの記録を有効にするには、Window、Preferences を選択して、C/C++, Build, LoggingEnable global build logging を開きます。

2.4.6 並列ビルドとビルドの動作

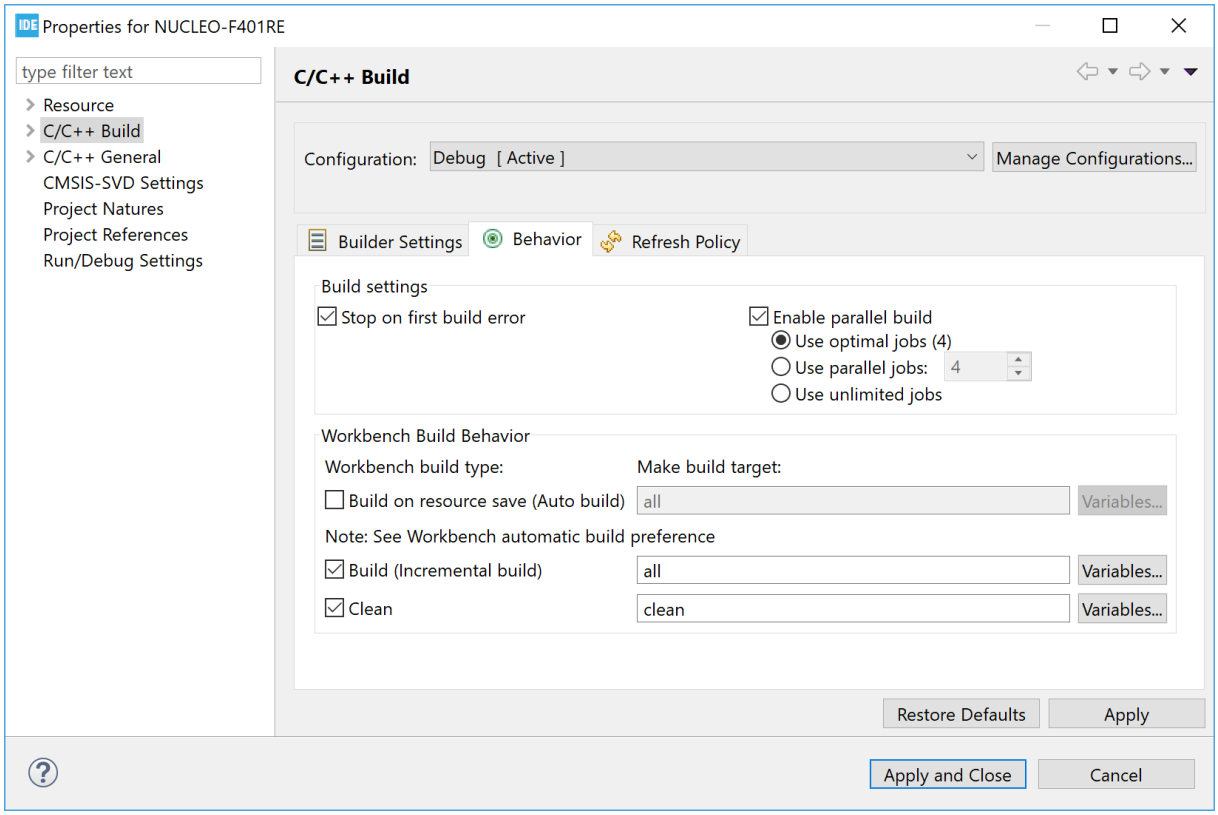
並列ビルドの状態は、コードのコンパイルやビルドに同時に複数のスレッドが使用されると発生します。ほとんどの場合、並列ビルドはビルド時間を大幅に短縮します。使用するスレッド数の最適値は、通常、コンピュータの CPU コア数に一致します。並列ビルドは有効または無効に設定できます。

並列ビルドを設定するには、次の手順を実行します。

1. [Project Explorer]ビューでプロジェクトを右クリックします。
2. メニュー ProjectProperties を選択します。
3. [Properties]パネルの C/C++ Build を選択します。
4. [Behavior]タブを開き、Enable parallel build を設定します。

[Behavior]タブには、エラー時の処理方法や、リソース保存時の自動ビルド、インクリメンタル·ビルド、クリーン·ビルドに関する設定もあります。

図 76. 並列ビルド



2.4.7 makefile ターゲットによるビルド後処理

makefile ターゲットにより高度なビルド後処理スクリプトを追加できます。それには、次の手順を実行します。

1. 新規ファイルを作成します。
2. ファイル名を `makefile.targets` とします。
3. このファイルをプロジェクトのルート·ディレクトリに保存します。

ファイルの内容は、下記の例と同様である必要があります。この例は、単に生成された `elf` ファイルを新しいファイルにコピーし、マクロ `BUILD_ARTIFACT`、`BUILD_ARTIFACT_PREFIX`、`BUILD_ARTIFACT_NAME`、`BUILD_ARTIFACT_EXTENSION` を使用するものです。これらのマクロは、STM32CubeIDE の v1.5.0 以降で、makefile 内に生成されます。

```
secure_target := \
                              $(BUILD_ARTIFACT_PREFIX)$(BUILD_ARTIFACT_NAME)-secure.$
(BUILD_ARTIFACT_EXTENSION)
main-build: $(secure_target)

$(secure_target): $(BUILD_ARTIFACT)
    # Do what you want here... simple copy file for demo
    cp "$<" "$@"
```

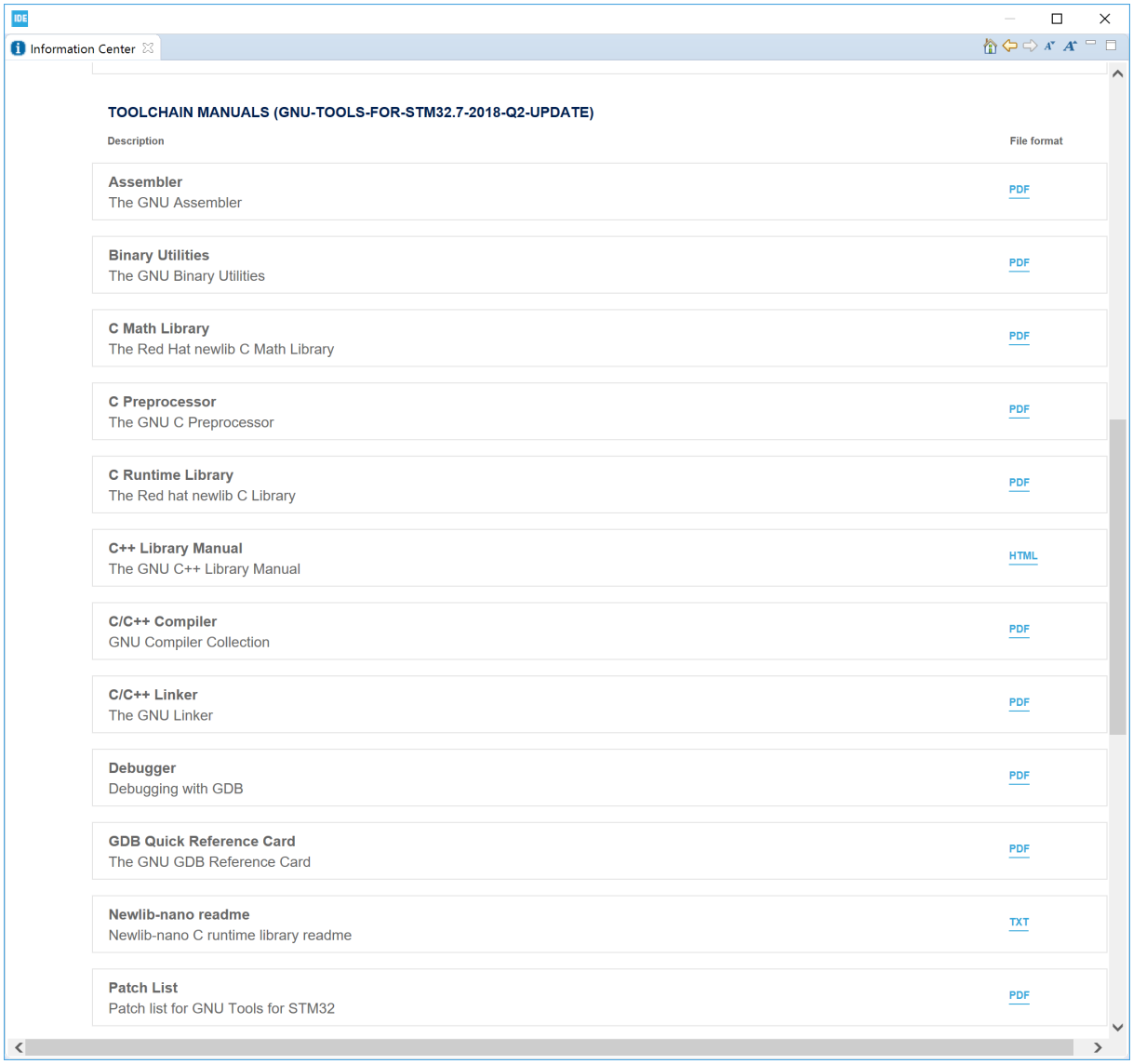
注 `make` では、空白文字ではなくタブ文字を使用する必要があります。

2.5 プロジェクトのリンク

このセクションでは、リンカとリンカのスクリプト·ファイルに関する基本情報を提供します。リンカの詳細は、GNU Linker のマニュアル（[EXT-05]）に記載されています。このドキュメントには、Information Center からアクセスできます。

Information Center ツールバー·ボタン をクリックし、[Information Center]ビューを開きます。C/C++ Linker The GNU Linker PDF のリンクにより、リンカのドキュメントを開きます。

図 77. リンカのドキュメント



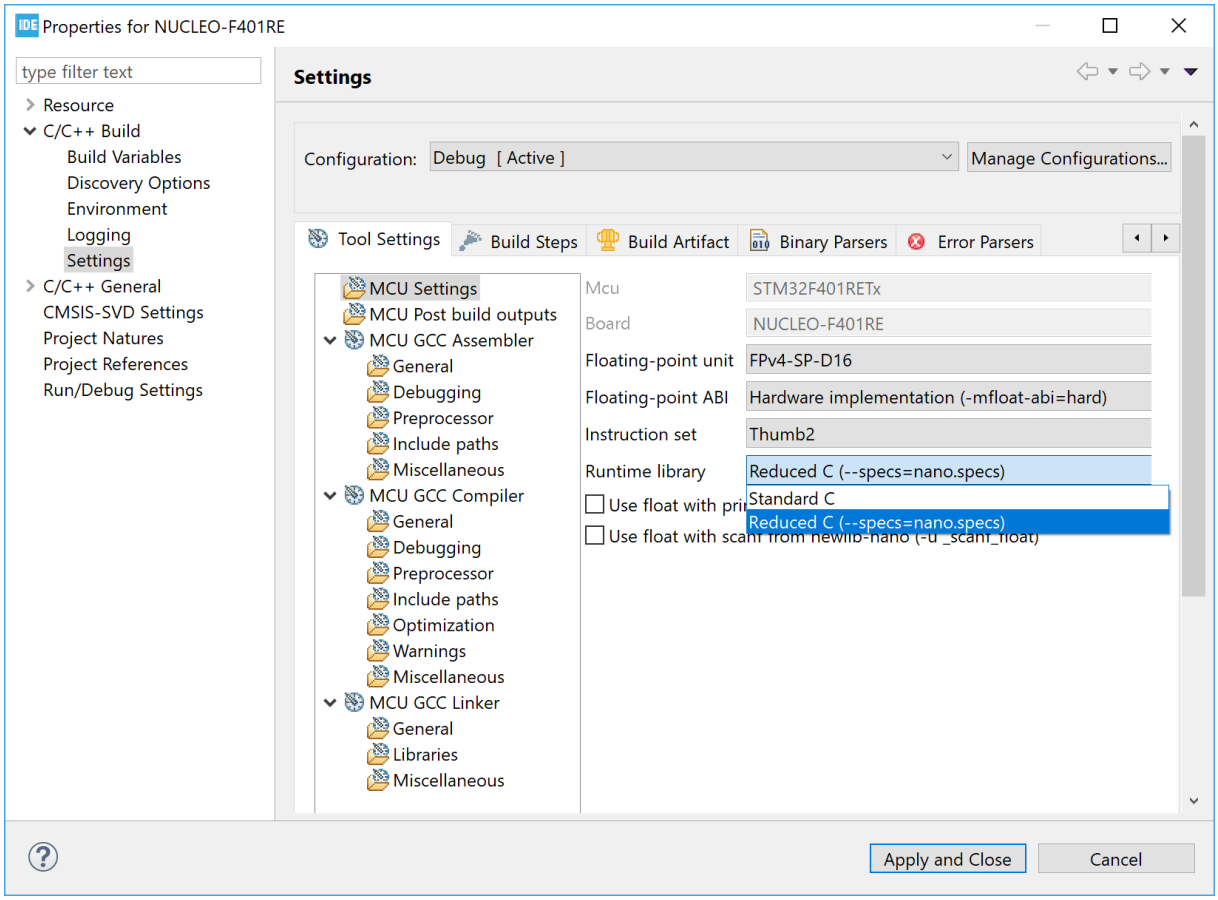
2.5.1 ランタイム ライブラリ

STM32CubeIDE に付属するツールチェーンには、newlib に基づいた 2 つのビルド済み ランタイム C ライブラリが含まれます。一つは標準 C ライブラリの newlib、もう一つは縮小版 C ライブラリの newlib-nano です。コード･サイズを小さくする必要がある場合に、newlib-nano を使用します。newlib-nano と標準の newlib の違いに関する詳細は、newlib-nano の readme ファイル([ST-09])を参照してください。Information Center からアクセスできます。

プロジェクトで使用する ランタイム ライブラリを選択するには、次の手順を実行します。

1. [Project Explorer]ビューでプロジェクトを右クリックします。
2. メニュー ProjectProperties を選択します。
3. [Properties]パネルの C/C++ BuildSettings を選択します。
4. [Tool Settings]タブを開き、MCU Settings を選択して、Runtime library の設定を構成します。

図 78. リンカ ランタイム ライブラリ



`newlib-nano` を使用し、かつ `scanf` や `printf` で浮動小数点数を処理する必要がある場合、追加のオプションが必要になります。`newlib-nano` と `newlib` では浮動小数点数の処理方法が異なるからです。`newlib-nano` では、書式設定された浮動小数点数の入出力は、weak シンボルとして実装されます。このため、`scanf`/`printf` で `%f` を使用する場合、次のように `-u` オプションによって明示的に指定してシンボルを指定する必要があります。

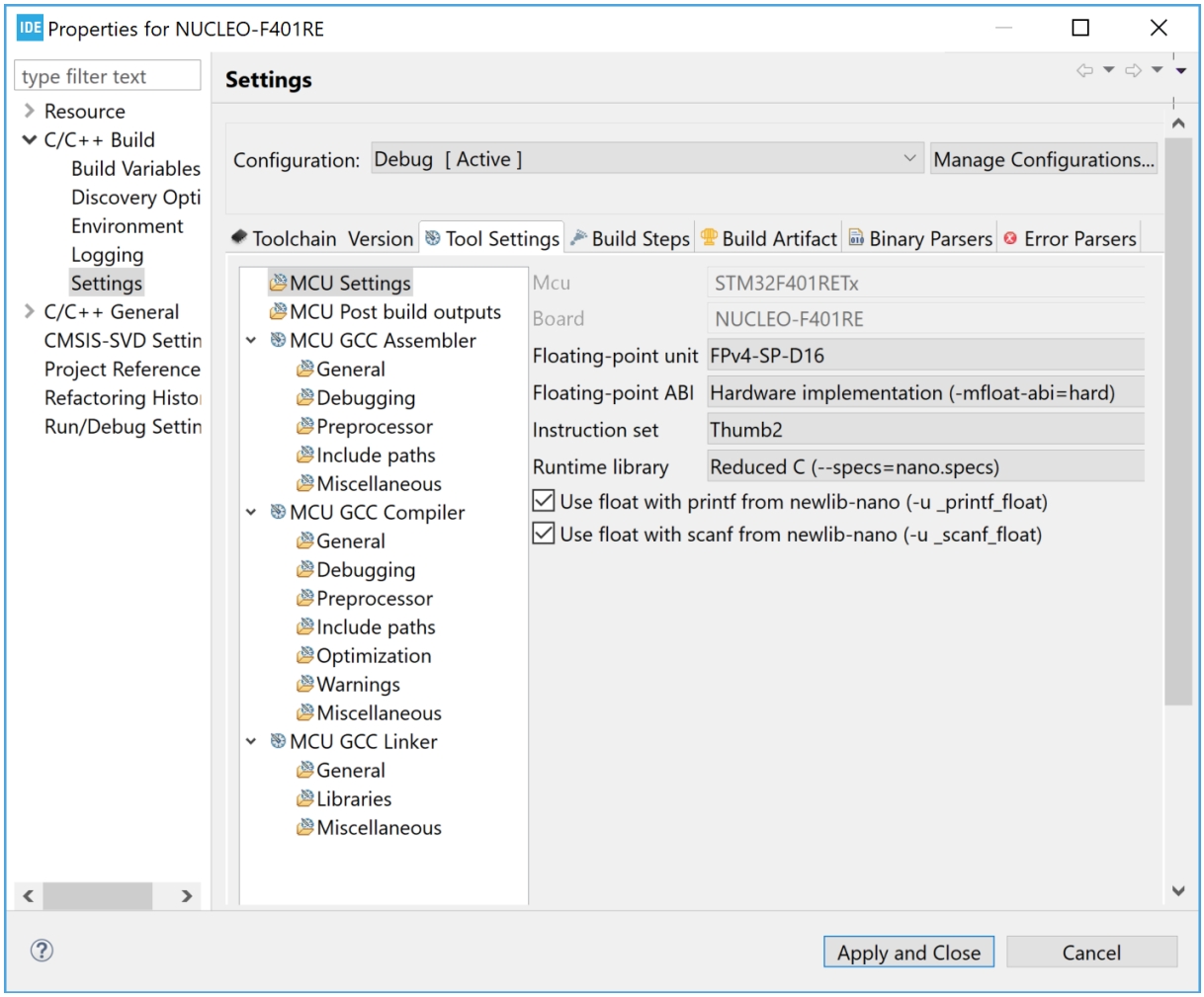
- `-u _scanf_float`
- `-u _printf_float`

例えば、`printf` によって `float` を出力する場合のコマンドラインは、次のようになります。

```
$ arm-none-eabi-gcc --specs=nano.specs -u _printf_float $(OTHER_LINK_OPTIONS)
```

このオプションは、[Tool Settings]タブの MCU Settings にある Use float... チェックボックスを使用することで有効化できます。

図 79. リンカ·ライブラリ newlib-nano と浮動小数点数



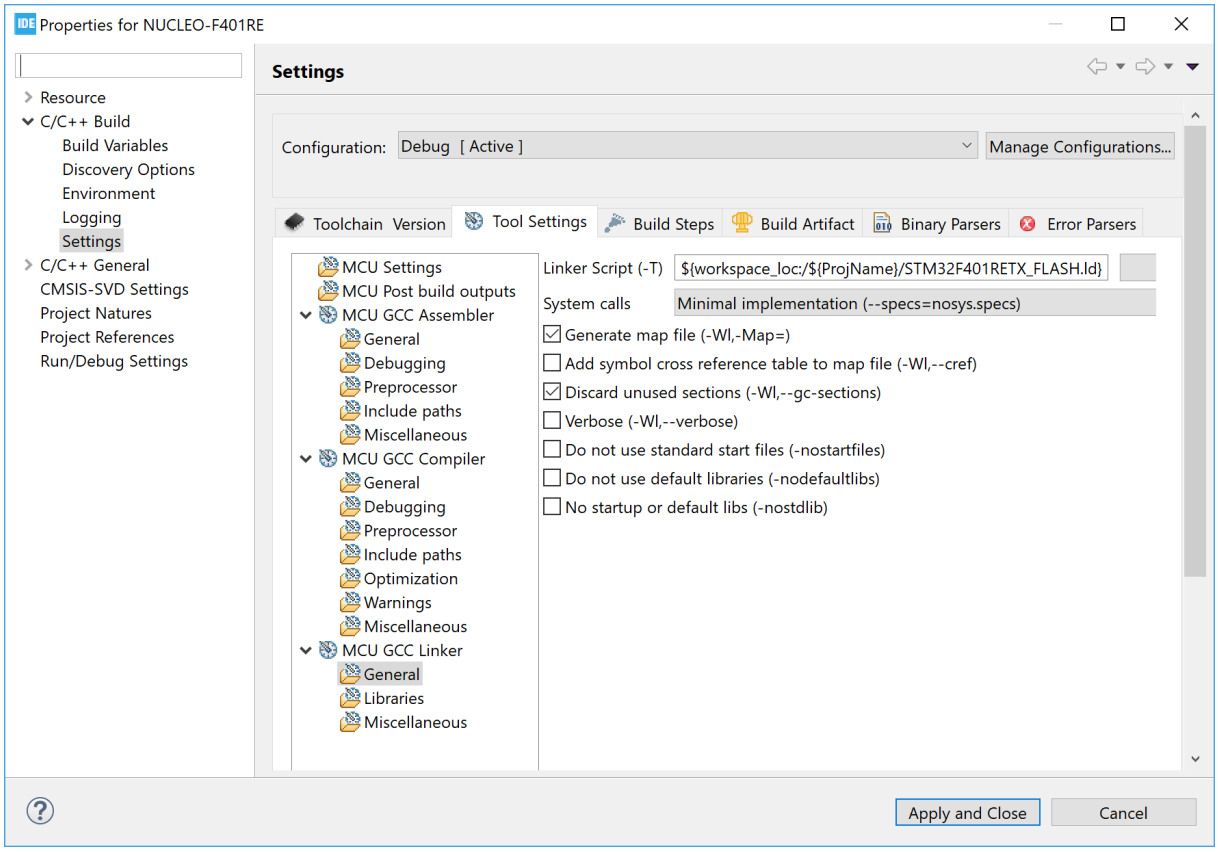
2.5.2 使用していないセクションの破棄

リンカが実行する最適化とは、出力されるバイナリから、使用していないコードやデータのセクション、デッド·コードを削除するプロセスです。ランタイム ·ライブラリとミドルウェア·ライブラリには通常、すべてのアプリケーションで使用されるわけではない多くの関数が含まれているため、出力バイナリから削除しない限り、貴重なメモリを浪費します。

プロジェクトの新規作成時にプロジェクト·ウィザードを使用すると、デフォルト設定として、使用していないセクションがリンカによって破棄されます。不使用のセクションに関する設定の確認または変更が必要になったら、いつでもプロジェクトのビルド設定を開いて、次の手順を実行できます。

1. [Project Explorer]ビューでプロジェクトを右クリックし、Properties を選択します。
2. ダイアログの C/C++ BuildSettings を選択します。
3. パネル内の[Tool Settings]タブを選択します。
4. MCU GCC LinkerGeneral を選択します。
5. プロジェクトの要件に応じて、Discard unused sections (-Wl,--gc-sections) を設定します。
6. プロジェクトを再度ビルドします。

図 80. リンカによる不使用セクションの破棄



2.5.3 malloc のページ･サイズ割当て

GNU Tools for STM32 ツールチェーンを標準 C ライブラリ `newlib` とともに使用する場合、`malloc` のページ･サイズ設定を変更できます。`newlib` のデフォルトのページ･サイズは 4096 バイトです。ユーザ･プロジェクトで `sysconf()` 関数を実装している場合、このユーザ関数が `_malloc_r()` によって呼び出されます。

下記の例は、ページ･サイズ 128 バイトで `sysconf()` 関数を実装する方法を示したものです。アプリケーションで、デフォルトの 4096 バイトよりも小さなページ･サイズを使用する必要が生じた場合、これと同様の関数を追加してください。

```
/**
******************************************************************************
**  File    : sysconf.c
******************************************************************************
**/

/* Includes */
#include <errno.h>
#include <unistd.h>

long sysconf(int name)
{
  if (name==_SC_PAGESIZE)
  {
    return 128;
  }
  else
  {
    errno=EINVAL;
    return -1;
  }
}
```

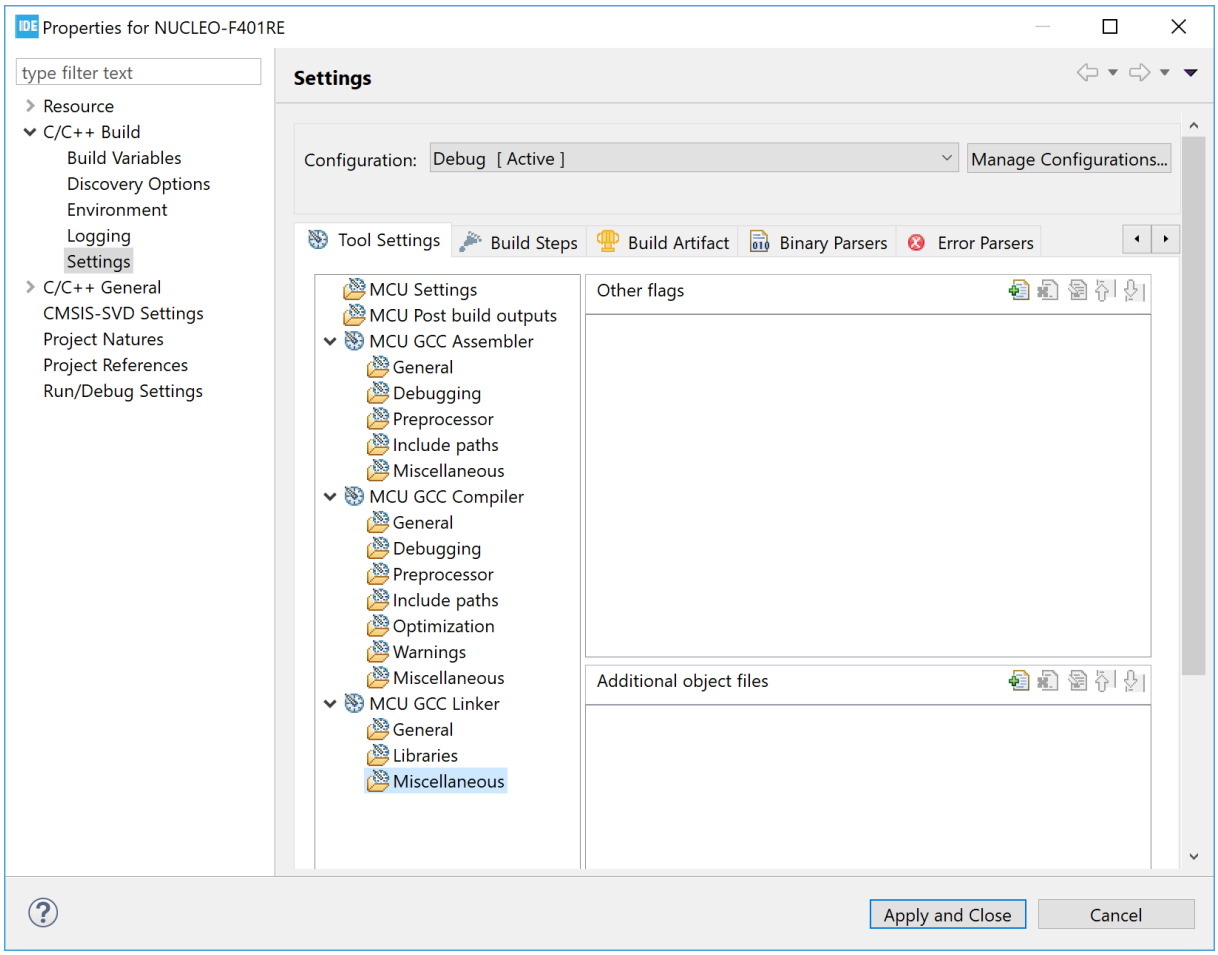
注 GNU ARM Embedded ツールチェーンを使用する場合は、アプリケーションで実装される `sysconf()` 関数は一切呼び出されず、常に `newlib` 内のデフォルトの `sysconf()` 関数が使用されます。また、縮小版 C ライブラリ `newlib-nano` とともに GNU Tools for STM32 ツールチェーンを使用する場合も、`sysconf()` は一切呼び出されません。

2.5.4 追加のオブジェクト·ファイルのインクルード

STM32CubeIDE では、プロジェクトにリンクする必要がある追加のオブジェクト·ファイルを簡単にインクルードできます。他のプロジェクトで作成したファイル、ソース·コードを入手できないコンパイル済みライブラリ、他のコンパイラで作成したオブジェクト·ファイルなどを使用できます。

1. [Project Explorer]ビューでプロジェクトを右クリックし、Properties を選択します。
2. ダイアログの C/C++ BuildSettings を選択します。
3. パネル内の[Tool Settings]タブを選択します。
4. MCU GCC LinkerMiscellaneous を選択します。
5. Add... アイコンをクリックすると、次のような複数の方法でオブジェクト·ファイルを追加できます。
   – [Add file path]ダイアログにファイル名を入力します。
   – Workspace... または File system... ボタンを使用して、ファイルの場所を指定します。

図 81. リンカによる追加のオブジェクト·ファイルのインクルード



2.5.5 リンカによる警告とエラーへの対処

GNU リンカは通常、警告を出力しません。そのような警告の抑制の一例が、通常の `Reset_Handler` 関数を含むスタートアップ·コードがプロジェクトに見つからない場合です。通常のサイレント·モードで動作する GNU リンカは、`elf` ファイルを作成し、`Reset_Handler` の欠落に関する警告出力のみを[Console]ウィンドウに報告します。

以下に警告メッセージの例を示します。

```
arm-none-eabi-gcc -o "NUCLEO-F401RE.elf" @"objects.list" -mcpu=cortex-m4 -
T"C:\Users\username\STM32CubeIDE\workspace_um\NUCLEO-F401RE\STM32F401RETX_FLASH.ld"
--specs=nosys.specs -Wl,-Map="NUCLEO-F401RE.map" -Wl,--gc-sections -static -
mfpu=fpv4-sp-d16 -mfloat-abi=hard -mthumb -Wl,--start-group -lc -lm -Wl,--end-group
c:\st\stm32cubeide_1.1.0.19w37\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.extern
altools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\arm-
none-eabi\bin\ld.exe: warning: cannot find entry symbol Reset_Handler; defaulting
to 0000000008000000
Finished building target: NUCLEO-F401RE.elf
```

この場合、新しい `elf` ファイルが作成されますが、警告が検出されない場合は、プログラムに `Reset_Handler` 関数が含まれていないため、プロジェクトのデバッグは機能しません。`--fatal-warnings` オプションを付加すると、警告をエラーとして処理するようにリンカを設定できます。

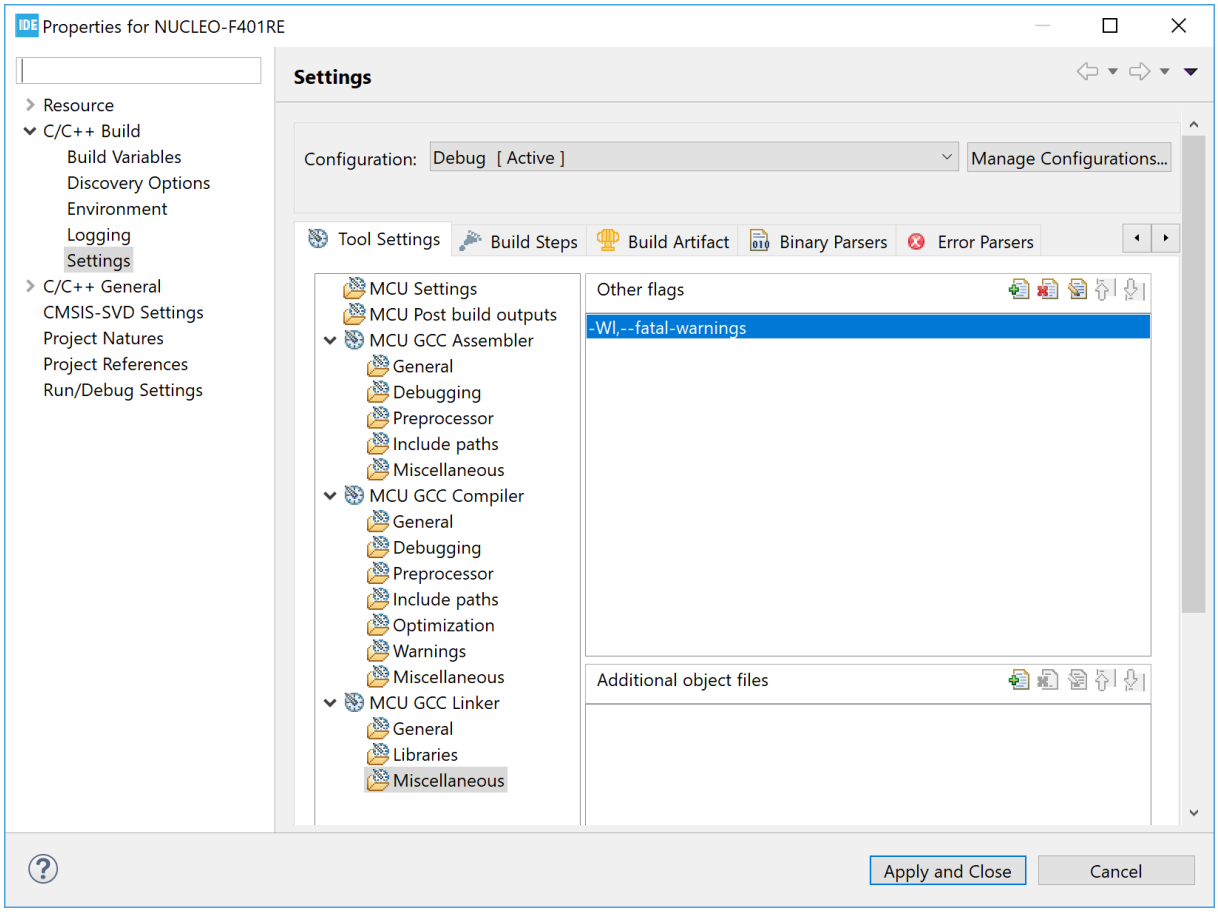
`--fatal-warnings` オプションを使用すると、リンカは `elf` ファイルを生成せず、コンソール·ログに次のようなエラーを表示します。

```
c:\st\stm32cubeide_1.1.0.19w37\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.extern
altools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\arm-
none-eabi\bin\ld.exe: warning: cannot find entry symbol Reset_Handler; defaulting
to 0000000008000000
collect2.exe: error: ld returned 1 exit status
make: *** [makefile:40: NUCLEO-F401RE.elf] Error 1
"make -j4 all" terminated with exit code 2. Build might be incomplete.
11:26:30 Build Failed. 1 errors, 6 warnings. (took 7s.193ms)
```

`-Wl,--fatal-warnings` オプションを使用するには、次の手順を実行します。:

1. [Project Explorer]ビューでプロジェクトを右クリックし、Properties を選択します。
2. ダイアログの C/C++ BuildSettings を選択します。
3. パネル内の[Tool Settings]タブを選択します。
4. MCU GCC LinkerMiscellaneous を選択します。
5. Other flags フィールドに `-Wl,--fatal-warnings` を追加します。

図 82. リンカの致命的警告



2.5.6 リンカ·スクリプト

リンカ·スクリプト·ファイル(`.ld`)は、インクルードするファイルと、それらが最終的にメモリのどこに配置されるかを定義します。この後のセクションでは、リンカ·スクリプト·ファイル内のいくつかの重要な部分について説明します。リンカに関する詳細は、C/C++ のリンカである GNU Linker のマニュアル([ST-05])を参照してください。このマニュアルは、Information Center のドキュメント·セクションから入手できます。特に、セクション 3.6 と 3.7 に注目してください。

リンカ·スクリプトは、メモリ領域と、stack、heap、bss、data、rodata、text およびプログラム·エントリの場所を指定します。スタックとヒープのサイズは、リンカ·スクリプト·ファイルの `_Min_Stack_Size` および `_Min_Heap_Size` の値を編集することで設定できます。ただし、これらの値は、スタックやヒープがメモリに収まるかどうかをリンカが検証する目的にしか使用されません。プログラムの実行時にスタックまたはヒープがより多くのメモリを要求すると、データが上書きされて予期せぬ結果を招く可能性があります。

表 3 に、例として、512K バイトのフラッシュ·メモリと 96K バイトの SRAM を備えた STM32F4 デバイスの典型的なプログラムとメモリのレイアウトを示します。このデバイスは、32 bit アドレス空間(0x0000 0000 ~ 0xFFFF FFFF)を持つ Cortex®-M コアに基づいています。

表 3. メモリ·マップのレイアウト

例 : STM32F4 96-KB SRAM 512-KB Flash メモリ	使用目的	ファイル リンカ·スクリプト .ld または .h および .c ファイル	コメント
0xFFFF FFFF 0xE000 0000	Cortex-M4 内部ペリフェラル	.h および .c ファイル	SysTick、NVIC、ITM、デバッグ、その他
0xDFFF FFFF 0x6000 0000	外部メモリ FMC(フレキシブル·メモリ·コントローラ)	リンカ·スクリプトと .h および .c ファイルへの追加が必要です。(1)	NOR Flash メモリ、NAND Flash メモリ、SPI Flash メモリ、PSRAM、SDRAM、その他
0x5FFF FFFF 0x4000 0000	STM32 ペリフェラル	.h および .c ファイル	GPIO、ADC、タイマ、USB、USART、その他
0x2001 8000 0x2000 0000	96-KB SRAM スタック ヒープ データ	リンカ·スクリプト _estack _Min_Stack_Size _Min_Heap_Size ._user_heap_stack .bss .data	スタックにはローカル·データが格納されます。(2) (3) malloc が使用するヒープ。(4) データ 静的グローバル·データ(.bss および .data) .bss == Uninitialized data スタートアップ·コードによりゼロにクリアされます。 .data == Initialized data スタートアップ·コードにより Flash メモリから SRAM にコピーされます。
0x0808 0000 0x0800 0000	512-KB Flash メモリ データ プログラム 割込み ベクタ·テーブル	リンカ·スクリプト .data .rodata ENTRY Reset_Handler(5) .text .isr_vector(6)	 SRAM にコピーする初期化されたデータ。 Flash メモリに配置される読出し専用のデータ .text == Program、例 : main.c, 内の main()、system_stm32f4xx.c 内の SystemInit()、startup_stm32*.s 内の Reset_Handler、startup_stm32*.s 内の g_pfnVectors、startup_stm32*.s 内のベクタ·テーブル
色の凡例			
Cortex®-M 内部ペリフェラルおよび STM32 ペリフェラル			
外部メモリ。通常、外部メモリを使用するにはリンカ·スクリプト、ヘッダ·ファイル、C ファイルを更新する必要があります。			
プログラム、データ、ヒープ、スタックが配置される Flash メモリと SRAM。通常、STM32CubeIDE でプロジェクトを作成する場合、これらの Flash メモリや RAM の領域は、リンカ·スクリプトやその他のファイルを更新しなくともアクセスし、使用することができます。リンカ·スクリプト·ファイルは、コード、データ、ヒープ、スタックをメモリ内にどのように配置するのかを定義します。			

1. 外部メモリを使用する場合、そのメモリをリンカ·スクリプト·ファイルに追加する必要があります。新しいメモリ領域を追加する方法は、セクション 2.5.7.1 を参照してください。
2. スタックは下方向に拡大していき、ヒープ領域に侵入する可能性があります。
3. プログラムの実行時にスタックまたはヒープがより多くのメモリを要求すると、データが上書きされて予期せぬ結果を招く可能性があります。
4. ヒープは上方向に拡大していき、スタック領域に侵入する可能性があります。
5. リンカ·スクリプト·ファイルにはプログラムのエントリ·ポイント定義が含まれます。通常は ENTRY(Reset_Handler) です。
6. 割込みベクタ·テーブルには、スタック·ポインタのリセット値、プログラム(Reset_Handler)、例外ハンドラ、割込みハンドラの開始アドレスが含まれます。通常、Reset_Handler のコードとベクタ·テーブル(g_pfnVectors)は、ファイル <startup_stm32xxx.s> で確認できます。

下記に、512 KB の Flash メモリと 96 KB の SRAM を備えた STM32F4 デバイス向けに STM32CubeIDE によって生成されたデフォルトのリンカ·スクリプトを示します。

コード抜粋の冒頭には、リンカ·スクリプトのヘッダ、エントリ、スタック、ヒープ、メモリの定義が示されています。

```
/**
 ******************************************************************************
 * @file      LinkerScript.ld
 * @author    Auto-generated by STM32CubeIDE
 *  Abstract     : Linker script for NUCLEO-F401RE Board embedding STM32F401RETx
Device from stm32f4 series
 *                      512Kbytes FLASH
 *                      96Kbytes RAM
 *
 *            Set heap size, stack size and stack location according
 *            to application requirements.
 *
 *            Set memory bank area and size if external memory is used
 ******************************************************************************
 * @attention
 *
 * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the "License"; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *                        opensource.org/licenses/BSD-3-Clause
 *
 ******************************************************************************
 */

/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);    /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x200;    /* required amount of heap  */
_Min_Stack_Size = 0x400;    /* required amount of stack */

/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 96K
  FLASH    (rx)    : ORIGIN = 0x8000000,   LENGTH = 512K
}
```

コード抜粋の後続の部分には、セクションの定義があります。

```
/* Sections */
SECTIONS
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data into "FLASH" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
```

```
  *(.text*)          /* .text* sections (code) */
  *(.glue_7)         /* glue arm to thumb code */
  *(.glue_7t)        /* glue thumb to arm code */
  *(.eh_frame)

  KEEP (*(.init))
  KEEP (*(.fini))

  . = ALIGN(4);
  _etext = .;        /* define a global symbols at end of code */
} >FLASH

/* Constant data into "FLASH" Rom type memory */
.rodata :
{
  . = ALIGN(4);
  *(.rodata)         /* .rodata sections (constants, strings, etc.) */
  *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
  . = ALIGN(4);
} >FLASH

.ARM.extab   : {
  . = ALIGN(4);
  *(.ARM.extab* .gnu.linkonce.armextab.*)
  . = ALIGN(4);
} >FLASH

.ARM : {
  . = ALIGN(4);
  __exidx_start = .;
  *(.ARM.exidx*)
  __exidx_end = .;
  . = ALIGN(4);
} >FLASH

.preinit_array     :
{
  . = ALIGN(4);
  PROVIDE_HIDDEN (__preinit_array_start = .);
  KEEP (*(.preinit_array*))
  PROVIDE_HIDDEN (__preinit_array_end = .);
  . = ALIGN(4);
} >FLASH

.init_array :
{
  . = ALIGN(4);
  PROVIDE_HIDDEN (__init_array_start = .);
  KEEP (*(SORT(.init_array.*)))
  KEEP (*(.init_array*))
  PROVIDE_HIDDEN (__init_array_end = .);
  . = ALIGN(4);
} >FLASH

.fini_array :
{
  . = ALIGN(4);
  PROVIDE_HIDDEN (__fini_array_start = .);
  KEEP (*(SORT(.fini_array.*)))
  KEEP (*(.fini_array*))
  PROVIDE_HIDDEN (__fini_array_end = .);
  . = ALIGN(4);
```

```
  } >FLASH

  /* Used by the startup to initialize data */
  _sidata = LOADADDR(.data);

  /* Initialized data sections into "RAM" Ram type memory */
  .data :
  {
    . = ALIGN(4);
    _sdata = .;        /* create a global symbol at data start */
    *(.data)           /* .data sections */
    *(.data*)          /* .data* sections */
    *(.RamFunc)        /* .RamFunc sections */
    *(.RamFunc*)       /* .RamFunc* sections */

    . = ALIGN(4);
    _edata = .;        /* define a global symbol at data end */

  } >RAM AT> FLASH

  /* Uninitialized data section into "RAM" Ram type memory */
  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order to initialize the .bss section */
    _sbss = .;         /* define a global symbol at bss start */
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;         /* define a global symbol at bss end */
    __bss_end__ = _ebss;
  } >RAM

  /* User_heap_stack section, used to check that there is enough "RAM" Ram  type
memory left */
  ._user_heap_stack :
  {
    . = ALIGN(8);
    PROVIDE ( end = . );
    PROVIDE ( _end = . );
    . = . + _Min_Heap_Size;
    . = . + _Min_Stack_Size;
    . = ALIGN(8);
  } >RAM

  /* Remove information from the compiler libraries */
  /DISCARD/ :
  {
    libc.a ( * )
    libm.a ( * )
    libgcc.a ( * )
  }

  .ARM.attributes 0 : { *(.ARM.attributes) }
}
```

2.5.6.1 *プログラム開始を定義する ENTRY コマンド*

プログラムで最初に実行する命令は、ENTRY コマンドで定義します。

例：

```
/* Entry Point */
ENTRY(Reset_Handler)
```

ENTRY の情報は GDB が使用し、プログラムがロードされたときに、プログラム·カウンタ(PC)に ENTRY のアドレスを設定します。上記の例では、ロード後に GDB に step または continue コマンドが与えられると、プログラムが Reset_Handler から開始されます。

注 GDB スクリプトの load コマンドの後に monitor reset コマンドがあると、プログラムの起動がオーバーライドされる場合があります。その場合は、コードがリセットから開始されます。

2.5.6.2 *スタックの位置*

スタックの位置は通常、_estack シンボルを介してスタートアップ·ファイルで使用されます。スタートアップ·コードは、通常、リンカ·スクリプトで指定されたアドレスによってスタック·ポインタを初期化します。Cortex®-M ベースのデバイスの場合、スタックのアドレスは、割込みベクタ·テーブルの最初のアドレスにも設定されます。

例：

```
/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);    /* end of "RAM" Ram type memory */
```

2.5.6.3 *ヒープとスタックの最小サイズの定義*

リンカ·スクリプトには、システムが使用するヒープとスタックの最小サイズを定義することが一般的です。

例：

```
_Min_Heap_Size = 0x200;    /* required amount of heap  */
_Min_Stack_Size = 0x400;    /* required amount of stack */
```

通常、ここで定義された値は、後ほどリンカ·スクリプトで参照され、ヒープとスタックがメモリに収まるかどうかをリンカが検証するために使用します。使用可能なメモリが不足していると、リンカがエラーを発生する場合があります。

2.5.6.4 *メモリ領域の指定*

メモリ領域は、ORIGIN と LENGTH という名前で指定されます。また、特定のメモリ領域の使用方法を指定する、(rx) などの属性リストを伴うことも一般的です。ここで、r は読出し専用のセクション、x は実行可能なセクションを意味します。属性の指定は必須ではありません。

例：

```
/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)    : ORIGIN = 0x8000000,    LENGTH = 512K
}
```

2.5.6.5 *出力セクション(.text と .rodata)の指定*

出力セクションは、'.text' や '.data'、その他のセクションがメモリ内のどこに配置されるかを定義します。下記の例は、.text や .rodata、その他のセクションをすべて Flash メモリの領域に配置するように、リンカに指示しています。例に記述された glue のセクションは、プログラム内に何らかの混合コードが含まれている場合に、GCC によって使用されます。例えば、Arm® コードが Thumb コードへの呼び出しを行った場合や、その逆の場合に glue コードが使用されます。

例：

```
/* Sections */
SECTIONS
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    .= ALIGN(4);
  } >FLASH

  /* The program code and other data into "FLASH" Rom type memory */
  .text :
  {
    .= ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH
```

2.5.6.6 *初期化データ(.data)の指定*

初期化されるデータ値には特別な処理が必要です。初期化値は Flash メモリに配置し、スタートアップ·コードが、適切な値で RAM 変数を初期化できるようにする必要があるからです。下記の例は、シンボル `_sidata`、`_sdata`、`_edata` を作成しています。スタートアップ·コードは、プログラム起動時に、これらのシンボルを使用して値を Flash メモリから RAM にコピーします。

例：

```
/* Used by the startup to initialize data */
  _sidata = LOADADDR(.data);

  /* Initialized data sections into "RAM" Ram type memory */
  .data :
  {
    .= ALIGN(4);
    _sdata = .;        /* create a global symbol at data start */
    *(.data)           /* .data sections */
    *(.data*)          /* .data* sections */
    *(.RamFunc)        /* .RamFunc sections */
    *(.RamFunc*)       /* .RamFunc* sections */

    . = ALIGN(4);
    _edata = .;        /* define a global symbol at data end */

  } >RAM AT> FLASH
```

2.5.6.7 *非初期化データ(.bss)の指定*

初期化しないデータ値はスタートアップ·コードによって 0 にリセットする必要があります。このため、リンカ·スクリプト·ファイルで、これらの変数の場所を特定する必要があります。下記の例では、シンボル _sbss と _ebss を作成しています。スタートアップ·コードは、これらのシンボルを使用して、初期化されない変数の値を 0 に設定します。

例 :

```
  /* Uninitialized data section into "RAM" Ram type memory */
  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order to initialize the .bss section */
    _sbss = .;          /* define a global symbol at bss start */
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;          /* define a global symbol at bss end */
    __bss_end__ = _ebss;
  } >RAM
```

2.5.6.8 *ユーザのヒープとスタックが RAM に収まることの確認*

通常、スクリプトには、必要なヒープとスタックを他のデータとともにすべて RAM に格納できるかどうかをリンカが確認するための、専用のコード·セクションが 1 つ含まれます。

例 :

```
   /* User_heap_stack section, used to check that there is enough "RAM" Ram type
memory left */
  ._user_heap_stack :
  {
    .= ALIGN(8);
    PROVIDE ( end = . );
    PROVIDE ( _end = . );
    .= . + _Min_Heap_Size;
    . = .+ _Min_Stack_Size;
    . = ALIGN(8);
  } >RAM
```

注 スタックは RAM_ の上部に、ヒープはデータの後に隙間を空けて配置されます。表 3. メモリ·マップのレイアウトを参照してください。

2.5.6.9 *リンカのマップとリスト·ファイル*

STM32CubeIDE で生成したプロジェクトをビルドすると、デバッグまたはリリースのビルド出力フォルダに、マップとリスト·ファイルが作成されます。これらのファイルには、プログラム内のコードとデータの最終的な位置に関して、詳細な情報が含まれています。

[Build Analyzer]ビューを使用すると、プログラムのサイズと位置を詳細に分析できます。詳細は、セクション 8 Build Analyzer を参照してください。

2.5.7 リンカ·スクリプトの変更

このセクションでは、リンカ·スクリプトの編集が必要になる一般的な使用例を紹介します。スクリプトを編集、管理することで、コードやデータをより厳密に配置できるようになります。

2.5.7.1 *新しいメモリ領域へのコード配置*

多くのデバイスが複数のメモリ領域を備えています。リンカ·スクリプトを使用して、コードを異なる特定の領域に配置できます。下記の例は、コードを IP_CODE という名前の新しいメモリ領域に配置するための、リンカ·スクリプトの変更方法を示しています。

例 :

```
Original MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)    : ORIGIN = 0x8000000,    LENGTH = 512K
}

Add IP_CODE into MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)    : ORIGIN = 0x8000000,    LENGTH = 256K
  IP_CODE (rx)    : ORIGIN = 0x8040000,    LENGTH = 256K
}
```

次のコードは、リンカ·スクリプト·ファイルのもう少し下にある、.data { ... } と .bss { ... } の両セクションの間に配置します。

例 :

```
  .ip_code :
  {
    *(.IP_Code*);
  } > IP_CODE
```

これによって、.IP_Code* という名前のセクションを、すべて IP_CODE メモリ領域に配置するようにリンカに指示します。このメモリ領域は、ターゲットのメモリ·アドレス 0x804 0000 から始まるように指定されています。

C コードでコンパイラに対して、このセクションに配置すべき関数を指定するには、関数宣言の前に __attribute__((section(".IP_Code"))) を追加します。

例 :

```
__attribute__((section(".IP_Code"))) int myIP_read()
{
  // Add code here...
  return 1;
}
```

これで、関数 myIP_read() は、リンカによって IP_CODE メモリ領域に配置されるようになります。

2.5.7.2 *RAM へのコードの配置*

RAM にコードを配置するには、リンカ·スクリプトとスタートアップ·コードに、いくつかの変更が必要です。下記の例は、内部 RAM を複数のセクションに分割し、コードをこれら内部 RAM セクションの 1 つに配置して実行する場合に必要となる変更について示したものです。

次のようにして、リンカ·スクリプトで MEMORY {} 領域に、新しいメモリ領域を定義します。

```
Original MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)    : ORIGIN = 0x8000000,    LENGTH = 512K
}

Split RAM into memory areas RAM1, RAM_CODE, RAM

/* Memories definition */
MEMORY
{
  RAM1    (xrw)    : ORIGIN = 0x20000000,    LENGTH = 16K
  RAM_CODE (xrw)    : ORIGIN = 0x20004000,    LENGTH = 16K
  RAM    (xrw)    : ORIGIN = 0x20008000,    LENGTH = 64K
  FLASH    (rx)    : ORIGIN = 0x8000000,    LENGTH = 512K
}
```

リンカ·スクリプトに、コードの出力セクションを定義します。これは、Flash メモリに属するロード·メモリ·アドレス(LMA)と RAM 内の仮想メモリ·アドレス(VMA)によって配置する必要があります。

```
/* load code used by the startup code to initialize the ram code */
_siram_code = LOADADDR(.RAM_CODE);
.RAM_CODE :
{
  . = ALIGN(4);
  _sram_code = .; /* create a global symbol at ram_code start */
  *(.RAM_Code)      /* .RAM_Code sections */
  *(.RAM_Code*)     /* .RAM_Code* sections */
  . = ALIGN(4);
  _eram_code = .; /* define a global symbol at ram_code end */
} >RAM_CODE AT> FLASH
```

RAM のコード領域を初期化し、コードを Flash メモリから RAM コード領域にコピーする必要があります。スタートアップ·コードは、配置情報シンボルの _siram_code、sram_code、_eram_code にアクセスできます。
RAM_CODE のロード·アドレス·シンボルをスタートアップ·ファイルに追加します。

```
/* Load address for RAM_CODE */
.word _siram_code;
.word _sram_code;
.word _eram_code;
```

Flash メモリ(LMA)から RAM(VMA)に RAM コードをコピーするコードを、スタートアップ·コードに追加します。

```
Reset_Handler:
  ldr   sp, =_estack             /* set stack pointer */

/* Copy the ram code from flash to RAM */
  movs  r1, #0
  b  LoopRamCodeInit

RamCodeInit:
  ldr  r3, =_siram_code
  ldr  r3, [r3, r1]
  str  r3, [r0, r1]
  adds  r1, r1, #4

LoopRamCodeInit:
```

```
  ldr  r0, =_sram_code
  ldr  r3, =_eram_code
  adds  r2, r0, r1
  cmp  r2, r3
  bcc  RamCodeInit

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b  LoopCopyDataInit

CopyDataInit:
```

C コードでコンパイラに対して、このセクションに配置すべき関数を指定するには、関数宣言の前に `__attribute__((section(".RAM_Code")))` を追加します。

```
__attribute__((section(".RAM_Code"))) int myRAM_read()
{
  // Add code here...
  return 2;
}
```

STM32CubeIDE で CCM SRAM からアプリケーション·コードを実行する方法の詳細は、[ST-12] を参照してください。関数または割込みハンドラを RAM から実行するために、リンカ·スクリプトとスタートアップ·コードを設定する方法の例を紹介しています。他の RAM 領域を追加し、コード·セクションを RAM に配置する方法のヒントは、[ST-12] の第 4 章に記載された例から得られるでしょう。

2.5.7.3 *特定アドレスへの変数の配置*

変数は、メモリ内の特定アドレスに配置できます。これを実現するには、リンカ·スクリプトを変更する必要があります。このセクションで示す例は、製品の `VERSION_NUMBER`、`CRC_NUMBER`、`BUILD_ID` を処理する一定値の変数を、メモリ内に配置します。

まず、リンカ·スクリプトで新しいメモリ領域を作成します。

```
Original MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)    : ORIGIN = 0x8000000,    LENGTH = 512K
}

Add a new 2K FLASH_V memory region at end of flash
/* Memories definition */
MEMORY
{
  RAM    (xrw)     : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)     : ORIGIN = 0x8000000,    LENGTH = 512K-2K
  FLASH_V (rx)    : ORIGIN = 0x807F800,    LENGTH = 2K
}
```

この時点で、メモリ·セクションを追加する必要があります。

次のコードは、リンカ·スクリプト·ファイルのもう少し下にある、.data { ...} と .bss { ...の両セクションの間に配置します。

```
.flash_v :
{
*(.flash_v*);
} > FLASH_V
```

これによって、flash_v* という名前のセクションを、すべて FLASH_V メモリ領域の flash_v 出力セクションに配置するようにリンカに指示します。このメモリ領域は、ターゲットのメモリ·アドレス 0x807 F800 から始まるように指定されています。
セクションには、data など、いくつかの事前定義されたものを除き、ほぼ任意の名前を付けることができます。

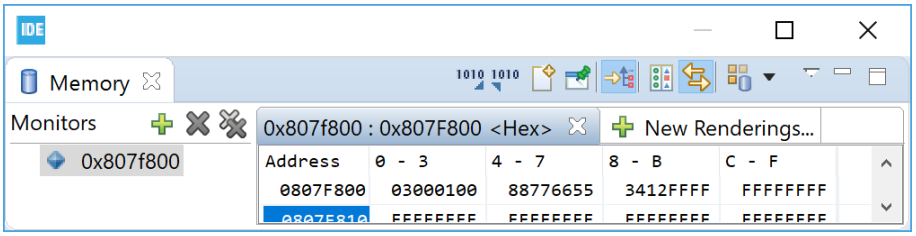
次に、FLASH_V メモリ領域に配置する必要がある変数を、C ファイル内の属性で定義する必要があります。

```
__attribute__((section(".flash_v.VERSION"))) const uint32_t
VERSION_NUMBER=0x00010003;
__attribute__((section(".flash_v.CRC"))) const uint32_t CRC_NUMBER=0x55667788;
__attribute__((section(".flash_v.BUILD_ID"))) const uint16_t BUILD_ID=0x1234;
```

この例をデバッグする際にメモリを調べると、以下を確認できます。

- アドレス 0x807 f800 に VERSION_NUMBER が格納されていること
- アドレス 0x807 f804 に CRC_NUMBER が格納されていること
- アドレス 0x807 f808 に BUILD_ID が格納されていること

図 83. リンカのメモリ出力



Flash メモリ中のデータの順番が重要な場合は、リンカ·スクリプトで変数の順番をマッピングします。こうすることで、変数を任意のファイルで定義できるようになります。リンカが、ファイルのリンク方法に関係なく、定義された順番で変数を出力するからです。その結果、リンカがファイルをビルドした後に何らかの方法で CRC_NUMBER を計算する場合、CRC_NUMBER を別のツールによって Flash メモリ·ファイルに挿入できます。

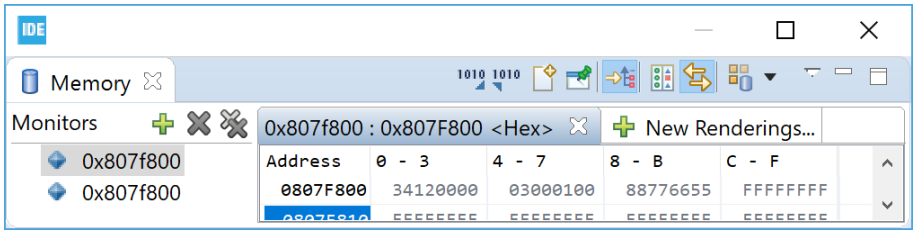
リンカ·スクリプトに、特別な名前を付けるセクションを、BUILD_ID、VERSION_NUMBER、CRC_NUMBER、その他(*)の順に追加することで順番を決定します。

```
.flash_v :
{
  *(.flash_v.BUILD_ID*);
  *(.flash_v.VERSION*);
  *(.flash_v.CRC*);
  *(.flash_v*);
} > FLASH_V
```

この例をデバッグする際にメモリを調べると、以下を確認できます。

- アドレス 0x807 f800 に BUILD_ID が格納されていること
- アドレス 0x807 f804 に VERSION_NUMBER が格納されていること
- アドレス 0x807 f808 に CRC_NUMBER が格納されていること

図 84. リンカによる指定された順序のメモリ出力



2.5.7.4 *バイナリ·データのブロックをリンクする方法*

リンクされるファイルに、バイナリ·データのブロックをリンクできます。下記の例は、`../readme.txt` ファイルを含める方法を示したものです。

例：

```
File: readme.txt
Revision: Version 2
Product news: This release ...
```

プロジェクトにこれを含める一つの方法は、C ファイルにそのための参照を記述します。つまり、セクションに対して、`incbin` ディレクティブとともに、割当て可能のオプション(`"a"`)を指定します。

```
asm(".section .binary_data,\"a\";"
".incbin \"../readme.txt\";"
);
```

これによって、新しいセクション `binary_data` は、Flash メモリに配置することという指示とともに、リンカ·スクリプトに追加されます。キーワード `KEEP()` を使用して入力セクションを囲い込むことで、たとえこのセクションが呼び出されない場合でも、リンカによるガベージ·コレクタの実行時に、このセクションが削除されないようにすることができます。

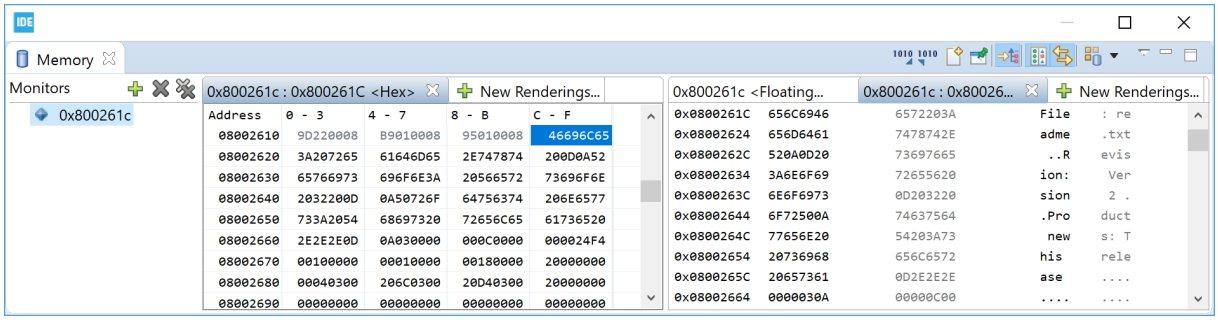
```
.binary_data :
{
  _binary_data_start = .;
  KEEP(*(.binary_data));
  _binary_data_end = .;
} > FLASH
```

この後、このブロックには C コードから次のようにしてアクセスできます。

```
extern int _binary_data_start;
int main(void)
{
  /* USER CODE BEGIN 1 */
  int *bin_area = &_binary_data_start;
```

バイナリ·データ(この場合は `readme` ファイル)は、プロジェクトのデバッグ時に[Memory]ビューで確認できます。

図 85. リンカ - ファイル readme を表示するメモリ



2.5.7.5 *メモリへの非初期化データの配置（NOLOAD）*

スタートアップ時に初期化してはならない変数を Flash または他の不揮発性メモリに配置する必要が生じる場合があります。そのような場合、リンカ･スクリプト内で特定の MEMORY AREA を作成し（FLASH_D）、その領域を使用するセクションに NOLOAD ディレクティブを適用します。

例：

```
The MEMORY AREA can be defined like this

/* Memories definition */
MEMORY
{
  RAM    (xrw)      : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)      : ORIGIN = 0x8000000,    LENGTH = 512K-4K
  FLASH_D (rx)     : ORIGIN = 0x807F000,     LENGTH = 2K
  FLASH_V (rx)     : ORIGIN = 0x807F800,     LENGTH = 2K
}
```

NOLOAD ディレクティブを使用して FLASH_D のセクションを追加します。これは、リンカ･スクリプトの、もう少し下で以下のコードを使用することで実現できます。

```
Place the following a bit further down in the script

 .flash_d (NOLOAD) :
 {
   *(.flash_d*);
 } > FLASH_D
```

最後に、FLASH_D メモリに配置する必要がある変数を宣言するときにセクションの属性を追加することで、プログラム内の任意の場所でデータを使用できます。

```
__attribute__((section(".flash_d"))) uint32_t Distance;
__attribute__((section(".flash_d"))) uint32_t Seconds;
```

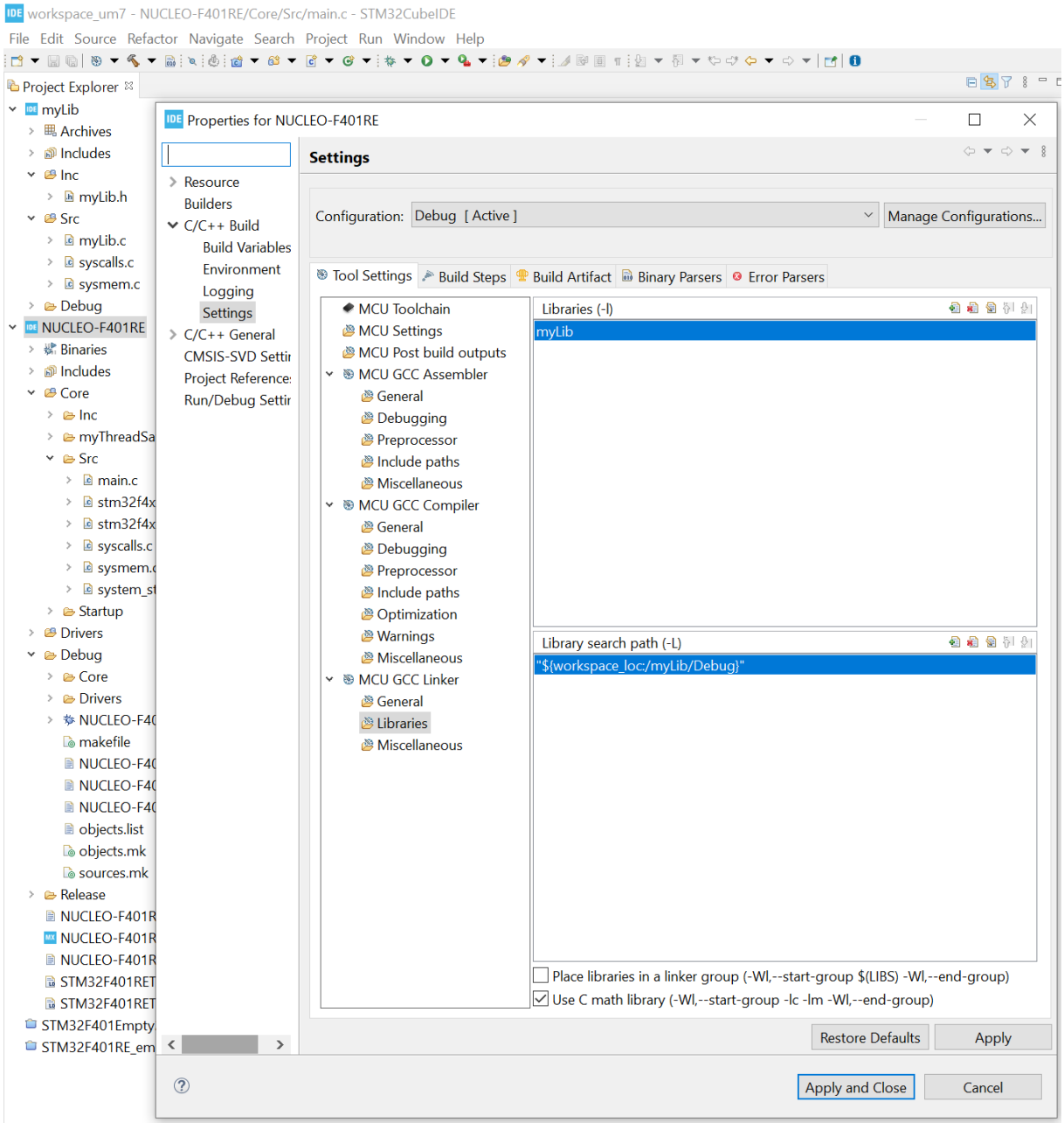
2.5.8 ライブラリのインクルード

ライブラリをプロジェクトにインクルードするには、次の手順を実行します。

1. [Project Explorer]ビューで、ライブラリをインクルードする必要があるプロジェクトを右クリックし、Properties を選択します。
2. ダイアログの C/C++ BuildSettings を選択します。
3. パネル内の[Tool Settings]タブを選択します。
4. C LinkerLibraries を選択します。

5. Libraries フィールドにライブラリ名を追加します。
   必ず、ライブラリのパスではなく名前を追加してください。GCC の表記規則に従い、ライブラリ名は、そのファイル名から接頭辞 `lib` と拡張子 `.a` を除いたものになります。
   例：ファイル名が `libmyLib.a` というライブラリの場合、ライブラリ名は myLib です。
   ライブラリ名が GCC の表記規則に従っていない場合、コロン(:)に続いてライブラリのフルネームを入力できます。
   例：ファイル名が `STemWin524b_CM4_GCC.a` というライブラリの場合、ライブラリ名として :STemWin524b_CM4_GCC.a を追加します。
6. Library Paths のリストには、ライブラリの保存場所へのパスを設定します。パスには、ライブラリ名を含めてはなりません。
   例：`${workspace_loc:/myLib/Debug}` は、アプリケーション・プロジェクトと同じワークスペース内に存在するライブラリ・プロジェクト `myLib` のアーカイブ・ファイルへのパスです。
7. 循環依存性を解決するためにライブラリを複数回リンクする必要がある場合、Place libraries in a linker group (-Wl,--start-group $(LIBS) -Wl,--end-group) を有効にします。

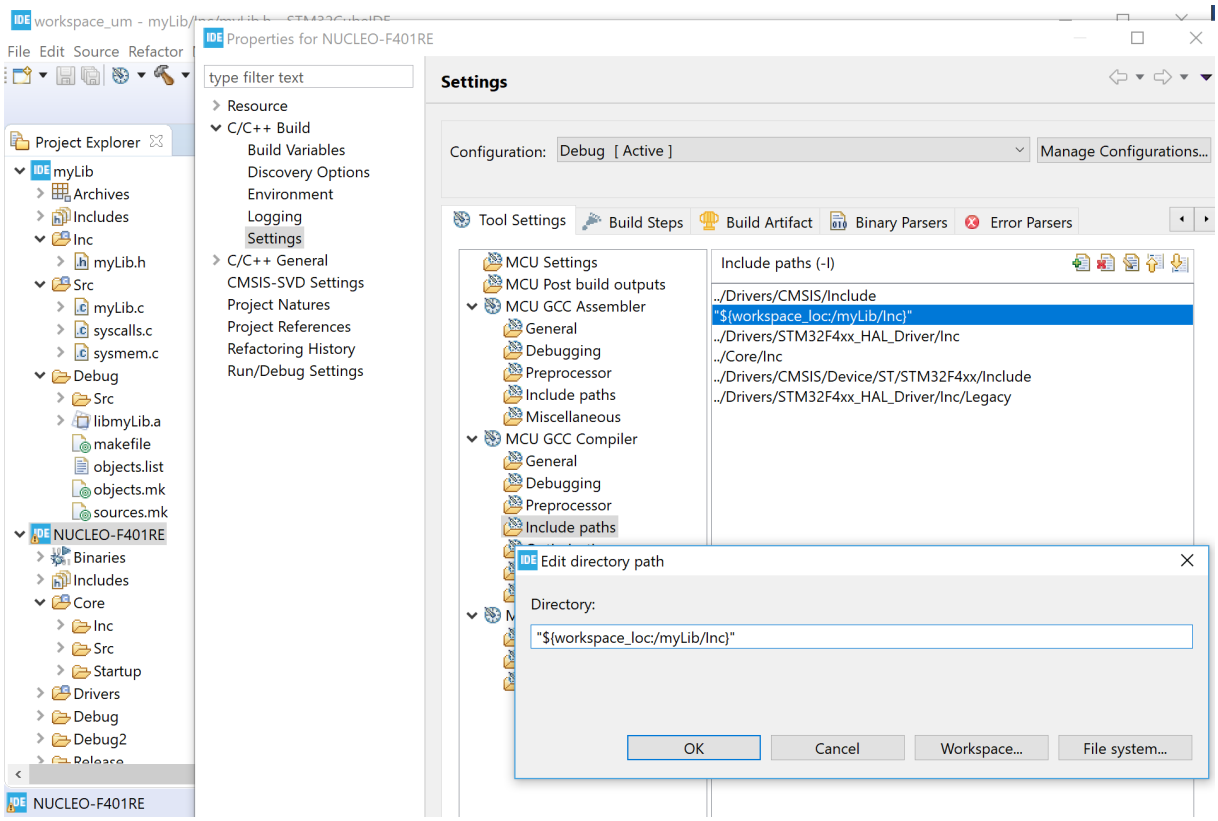
図 86. ライブラリのインクルード



ヘッダ·ファイルのソース·フォルダも Include paths フィールドに追加する必要があります。

1. MCU GCC CompilerInclude paths を選択します。
2. Add... ボタンをクリックし、ライブラリ内のヘッダ·ファイルのソース·フォルダへのパスを追加します。

図 87. インクルード·パスへのライブラリのヘッダ·ファイルの追加



注 インクルード·パスによって追加されたライブラリは、外部パーティから提供されたものであることから、スタティック·ライブラリと見なされます。外部ヘッダ·ファイルの場合、内容が変化することはないため再スキャンは行われません。外部ライブラリを通常のソース·フォルダとして扱う必要がある場合、そのフォルダもソース·フォルダとしてプロジェクトに追加する必要があります。

プロジェクトが別のプロジェクト、ライブラリまたは通常のプロジェクトを参照している場合の詳細は、セクション 2.5.9 プロジェクトの参照 を参照してください。

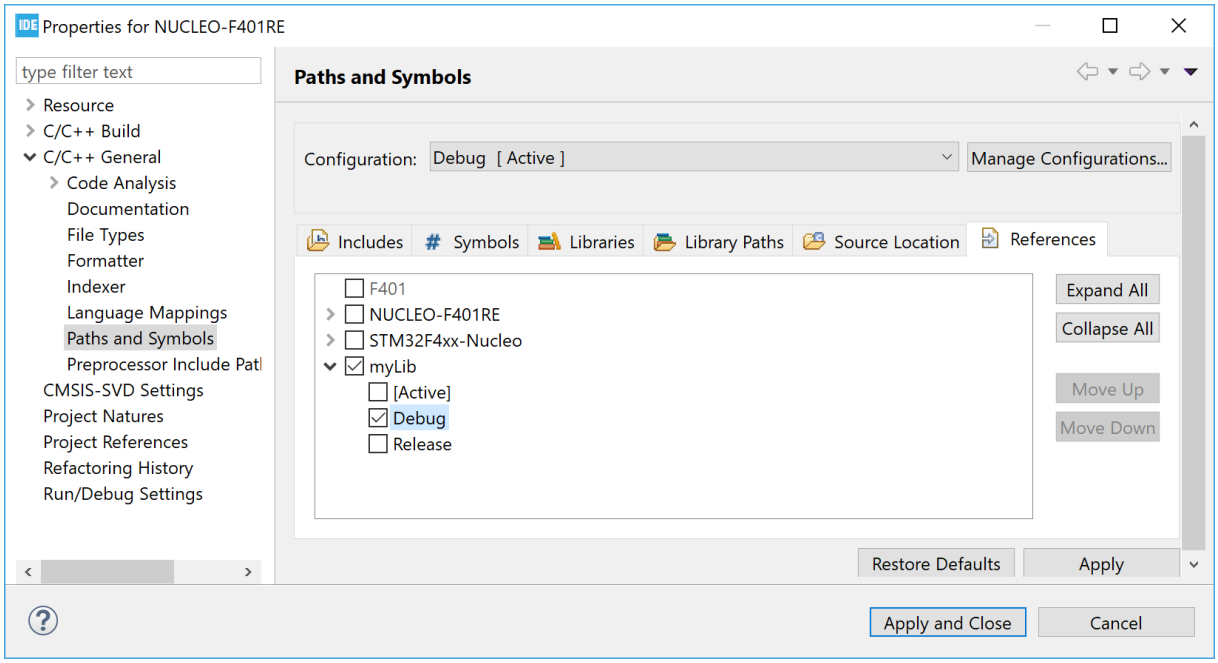
2.5.9 プロジェクトの参照

プロジェクトが他のプロジェクトのコードを使用している場合、両プロジェクトが互いに参照する必要があります。

プロジェクトが他のプロジェクトの特定のビルドを参照するには、次の手順を実行します。

1. 代わりにメニュー ProjectProperties を選択します。
2. C/C++ GeneralPaths and Symbols を選択します。
3. [References]タブを開きます。
4. 現在のプロジェクトが参照している Configuration を選択します。

図 88. プロジェクト参照の設定



注　参照先として複数のプロジェクトを使用する場合、Move Up および Move Down ボタンを使用して優先順位を設定します。

プロジェクトの参照を適切に設定することには、多くのメリットがあります。

- 関連するプロジェクトの再ビルドが必要以上に実行されることがありません。
- インデクサがライブラリから関数を見つけて開くことができます。この機能を使用するには、エディタのライブラリ関数が使われている場所を Ctrl キーを押しながらクリックすると、ライブラリのソース·ファイルがエディタで開きます。
- ライブラリ内の関数に呼び出しの階層構造を作成できます。呼び出し階層を見つけるには、関数名にマークを付け、Ctrl+Alt+H を押すと、[Call Hierarchy]ビューに呼び出しの階層が表示されます。

ライブラリ·プロジェクトを参照として追加している場合、そのライブラリの[Paths and Symbols]プロパティ·ページの設定が、すべて適切に適用されます。このプロパティ·ページに依存するツールの設定も調整されます。

これは、ローカルに開発したライブラリの追加方法として推奨されます。ライブラリの追加に関する詳細は、セクション 2.5.8 ライブラリのインクルードを参照してください。

プロジェクトに互いを参照させるもう一つの方法は、次のとおりです。

1. ProjectProperties を選択します。
2. Project References を選択します。
3. 参照するプロジェクトを選択し、マークを付けます。

ただし、この方法では異なるビルド設定を参照できず、ライブラリの自動設定も行われません。

2.6 入出力のリダイレクト

C ランタイム ライブラリには多くの関数が含まれ、そのいくつかは入出力を処理するものです。`printf()`、`fopen()`、`fclose()`、その他多くの入出力関連 ランタイム 関数があります。これらの関数の入出力を実際の組込みプラットフォームにリダイレクトする手法は広く使われています。例えば、`printf()` の出力を LCD ディスプレイやシリアル·ケーブルにリダイレクトしたり、`fopen()`、`fclose()` などのファイル操作を、Flash メモリのファイル·システム·ミドルウェアにリダイレクトしたりします。

2.6.1 printf() のリダイレクト

`printf()` のリダイレクトには、UART または SWV/ITM を使用するなど、いくつかの方法があります。もう一つの方法は、SEGGER が提供する リアルタイム転送 テクノロジー(RTT)を使用します。

これら 3 つの方法を比較すると、次のような特徴があります。

- UART 出力は、おそらく最も一般的に使用されている方法でしょう。この方法では、組込みシステムからの出力を RS-232 などによってターミナルに送信します。ある程度の CPU オーバーヘッドが生じ、中程度の帯域幅を必要とします。

- 計測トレース･マクロセル（ITM）出力は効率的ですが、デバイスが シリアル･ワイヤ･ビューア（SWV）を伴う Arm® CoreSight™ デバッガ･テクノロジーに対応している必要があります。通常、Cortex®-M3、Cortex®-M4、Cortex®-M7、Cortex®-M33 ベースのデバイスが、これに当てはまります。ただし、SWV 信号を使用でき、ボードに接続する必要もあります。CPU オーバーヘッドは少ないものの帯域幅は限られています。ITM 出力については、セクション 4 シリアル･ワイヤ･ビューア（SWV）トレースを使用したデバッグ で説明します。
- RTT による方法は、SEGGER の Web サイトで解説されています。RTT は高速な手法ですが、SEGGER の J-LINK デバッグ･プローブが必要です。

UART または ITM 出力による入出力のリダイレクトを有効にするには、プロジェクトで `syscalls.c` をインクルードしてビルドする必要があります。`printf()` を使用すると、`_write()` 関数が呼び出され、この関数が `syscalls.c` で実装されているからです。

`syscalls.c` のファイルは通常、STM32CubeIDE のプロジェクトを新規作成するときに、プロジェクト内に作成され、インクルードされます。`printf()` のリダイレクトを有効にするには、このファイル内の `_write()` 関数を変更する必要があります。`__io_putchar()` への呼び出しを変更します。`_write()` 関数の変更の仕方は、ハードウェアやライブラリの実装方法によって異なります。

下記の例は、`printf` の出力を STM32F4 シリーズのデバイスで ITM にリダイレクトするために `syscalls.c` を変更する方法を示したものです。具体的には、`ITM_SendChar()` にアクセスして、`ITM_SendChar()` への呼び出しを実行するために、いくつかのヘッダ･ファイルを追加します。

```
Original _write() function

__attribute__((weak)) int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        __io_putchar(*ptr++);
    }
    return len;
}

Modified with added header files calling ITM_SendChar(*ptr++);

#include "stm32f4xx.h"
#include "core_cm4.h"

__attribute__((weak)) int _write(int file, char *ptr, int len)
 {
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        //__io_putchar(*ptr++);
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

`syscalls.c` の `_write` 関数には weak 属性が含まれていることがわかります。これは、プロジェクトが使用する任意の C ファイルで `_write` 関数を実装できることを意味します。

例えば、新しい `_write()` 関数を `main.c` に直接追加することも可能です。その場合、下記の例のように weak 属性は省略します。

```
int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        //__io_putchar(*ptr++);
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

2.7 空のプロジェクトや CDT™ プロジェクト向けのスレッドセーフ·ウィザード

STM32CubeIDE は、スレッドセーフ·ウィザードを備えています。このウィザードは、アプリケーション·コードや割込みによって、あるいはリアルタイム·オペレーティング·システムを使用する場合に、リソース使用方法を変更できるファイルを生成します。

注 スレッドセーフ·ウィザードは、STM32CubeIDE の空のプロジェクトに対してのみ使用できます。STM32CubeMX で管理しているプロジェクトの場合、スレッドセーフ実装のための設定は、STM32CubeMX のダイアログから行う必要があります。

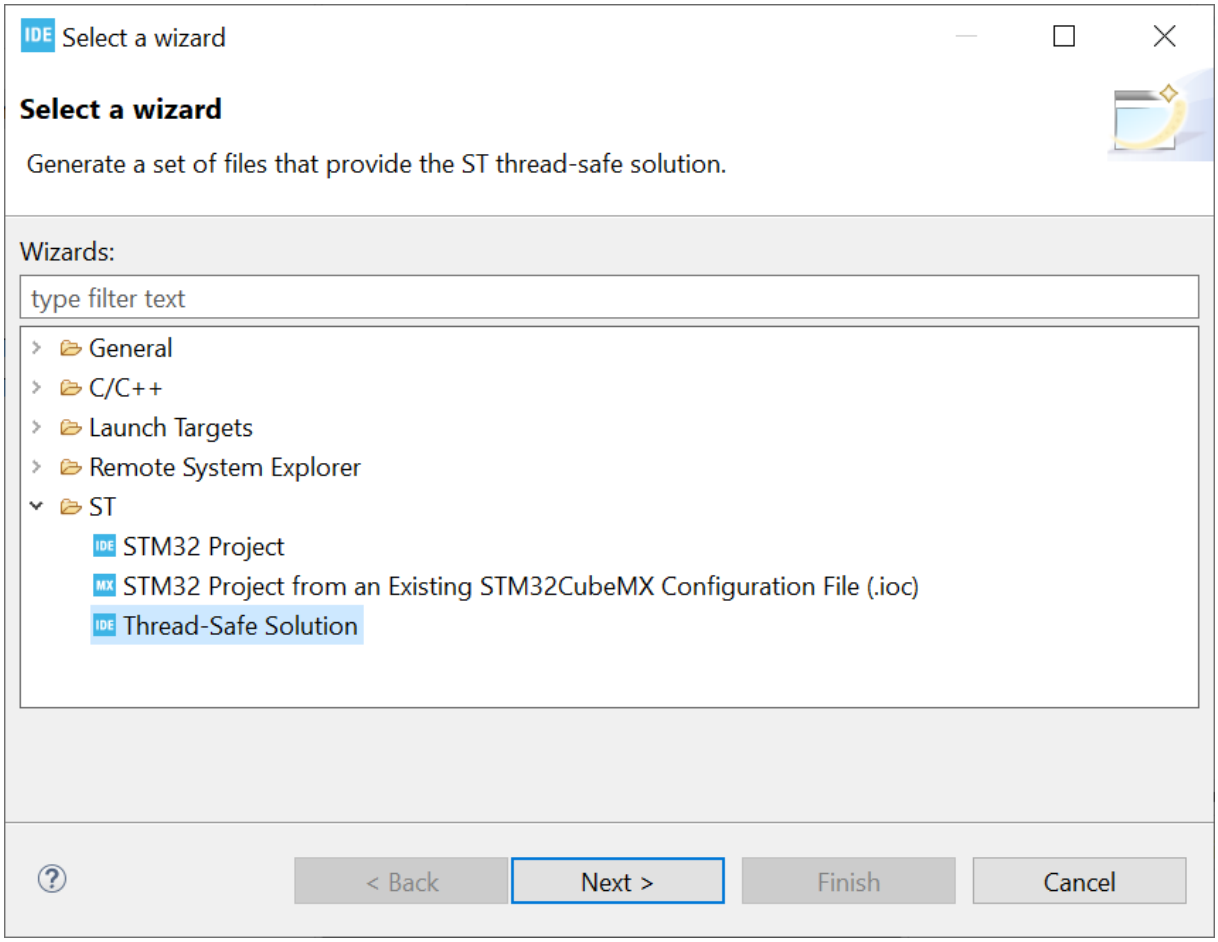
スレッドセーフ·ウィザードは、3 つのファイルを作成し、プロジェクトに `STM32_THREAD_SAFE_STRATEGY` の定義を追加します。次の 3 つのファイルが作成されます。

- `newlib_lock_glue.c`
- `stm32_lock_user.h`
- `stm32_lock.h`

下記の例では、まず、空のプロジェクト内に `myThreadSafe` フォルダが作成されます。[Thread-Safe Solution]ウィザードでは、このフォルダが選択され、この中にファイルが生成されます。

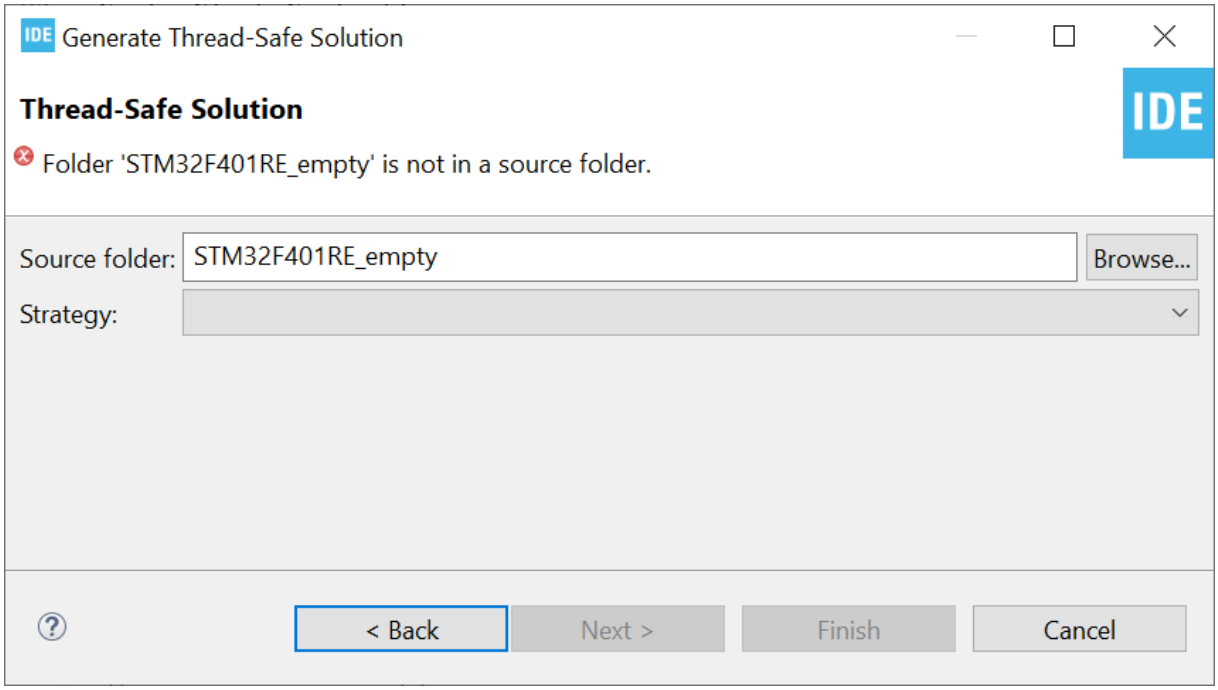
メニュー FileNewOther… を開き、図 89 に示すウィザード選択ウィンドウを表示させます。

図 89. ウィザードの選択



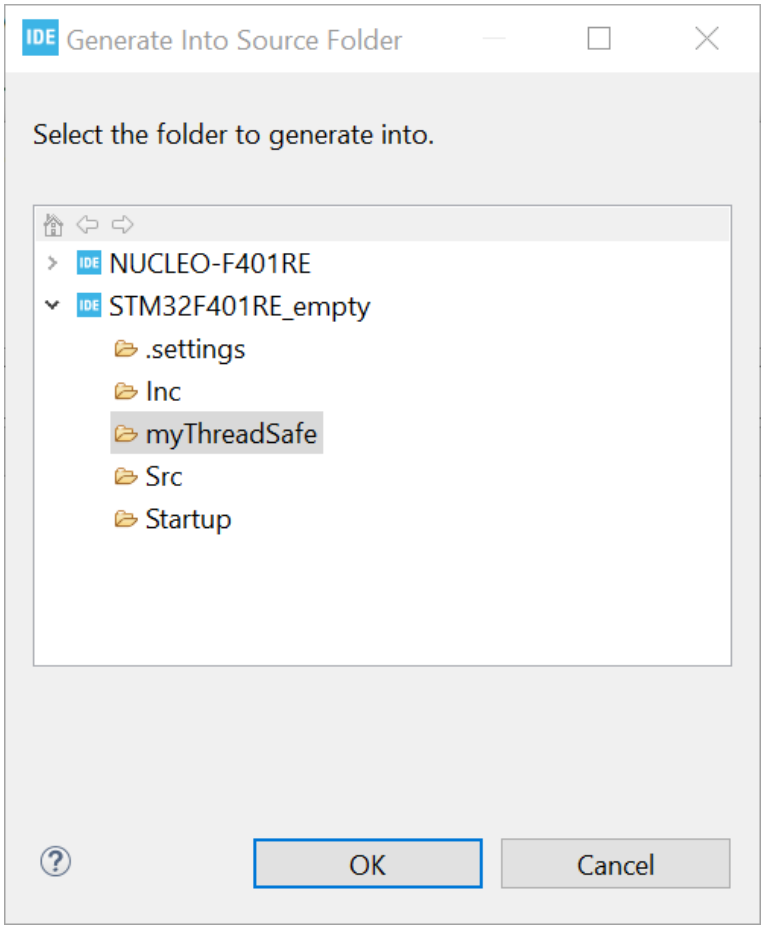
ST ノードの Thread-Safe Solution を選択し、Next > をクリックして［Thread-Safe Solution］ウィザードを開きます。

図 90. [Thread-Safe Solution]ウィザード



Browse をクリックして、[Generate Into Source Folder]ダイアログを開きます。

図 91. スレッドセーフのソース・フォルダの場所



ファイルの生成先となるソース・フォルダを選択して OK をクリックします。
ウィザードは、次の 5 種類のスレッドセーフ・ストラテジーを提示します。

1. ユーザ定義のスレッドセーフ実装
2. 割込みロックの使用を許可
3. 割込みロックの使用を禁止
4. 割込みロックの使用を許可。FreeRTOS™ のロックにより実装
5. 割込みロックの使用を禁止。FreeRTOS™ のロックにより実装

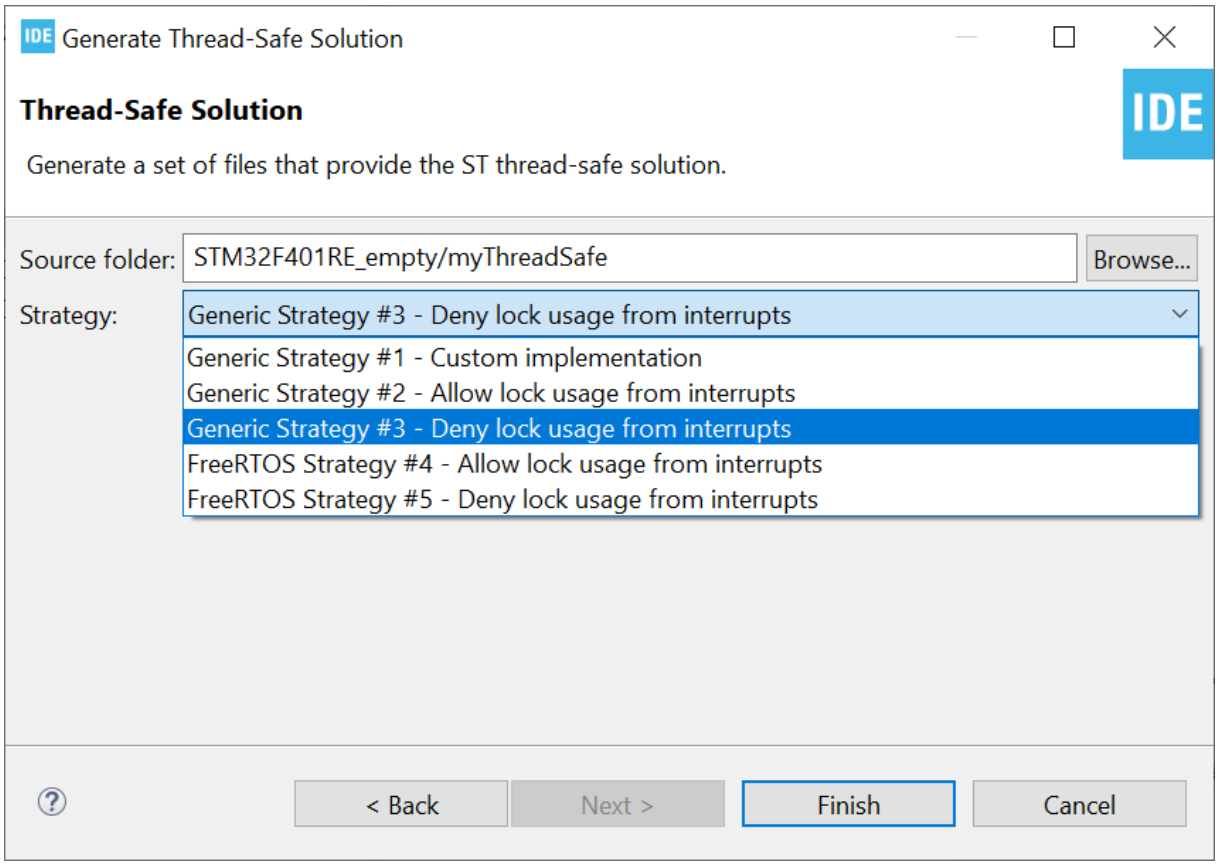
各ストラテジーは、`stm32_lock.h` ファイルの中で説明されています。

```
  * 1. User defined thread-safe implementation.
  *    User defined solution for handling thread-safety.
  *    NOTE: The stubs in stm32_lock_user.h needs to be implemented to gain
  *    thread-safety.
  *
  * 2. Allow lock usage from interrupts.
  *    This implementation will ensure thread-safety by disabling all interrupts
  *    during e.g. calls to malloc.
  *    NOTE: Disabling all interrupts creates interrupt latency which
  *    might not be desired for this application!
  *
  * 3. Deny lock usage from interrupts.
  *    This implementation assumes single thread of execution.
  *    Thread-safety dependent functions will enter an infinity loop
  *    if used in interrupt context.
  *
  * 4. Allow lock usage from interrupts. Implemented using FreeRTOS locks.
```

```
*    This implementation will ensure thread-safety by entering RTOS ISR capable
*    critical sections during e.g. calls to malloc.
*    By default this implementation supports 2 levels of recursive locking.
*    Adding additional levels requires 4 bytes per lock per level of RAM.
*    NOTE: Interrupts with high priority are not disabled. This implies
*    that the lock is not thread-safe from high priority interrupts!
*
* 5. Deny lock usage from interrupts. Implemented using FreeRTOS locks.
*    This implementation will ensure thread-safety by suspending all tasks
*    during e.g. calls to malloc.
*    NOTE: Thread-safety dependent functions will enter an infinity loop
*    if used in interrupt context.
```

図 92 に示すように、ストラテジーを選択します。

図 92. スレッドセーフ・ストラテジーの選択

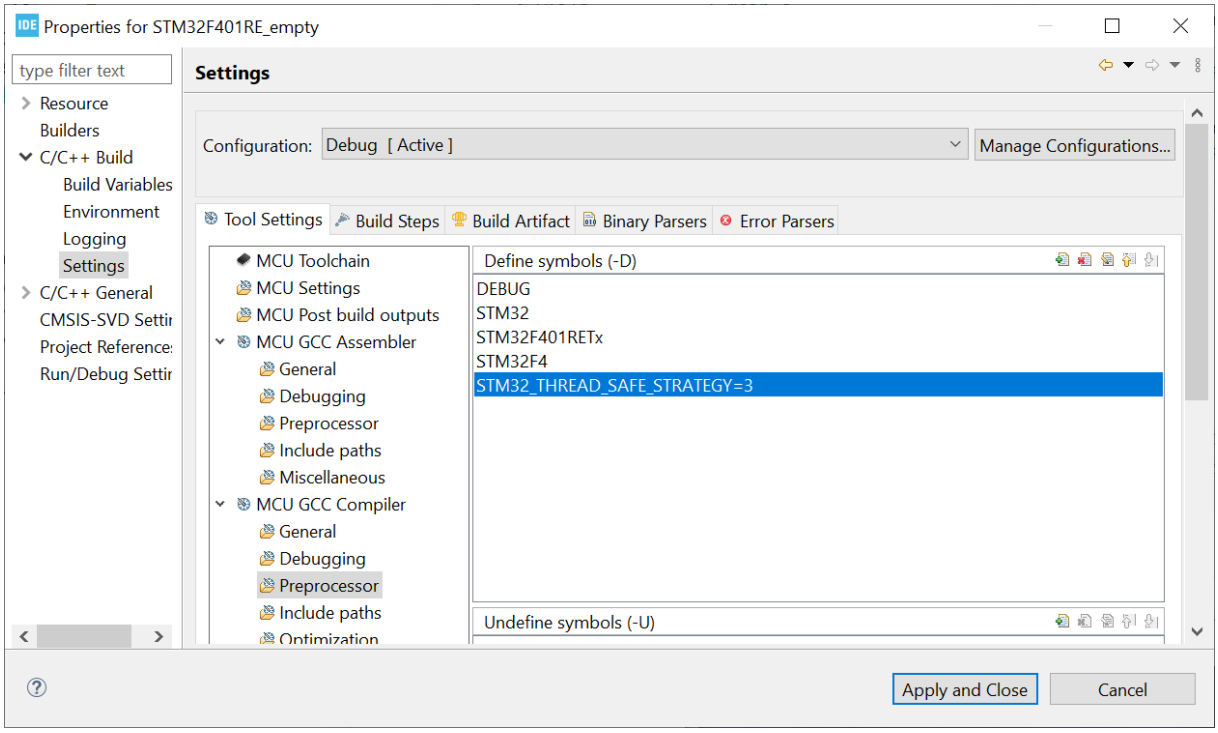


ストラテジーを選択し、Finish をクリックして、選択したソース・フォルダにファイルを生成します。

注　選択したストラテジーにかかわらず、同じファイルが生成され、それらに含まれる情報も同じです。

ウィザードによってプロジェクトに新しい定義 `STM32_THREAD_SAFE_STRATEGY=3` が追加されます。この定義はプロジェクトのビルド時にプリプロセッサが使用します。定義の値は、ウィザードで選択したストラテジーに従って設定されます。定義は、プロジェクト・プロパティを開いて[Tool Settings]タブを参照することで確認できます。

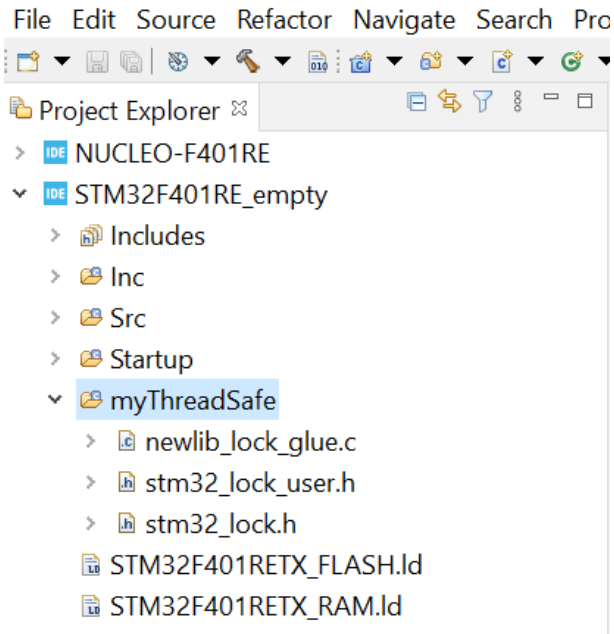
図 93. スレッドセーフのプロパティ



生成されたファイルは、Project Explorer に表示されます。

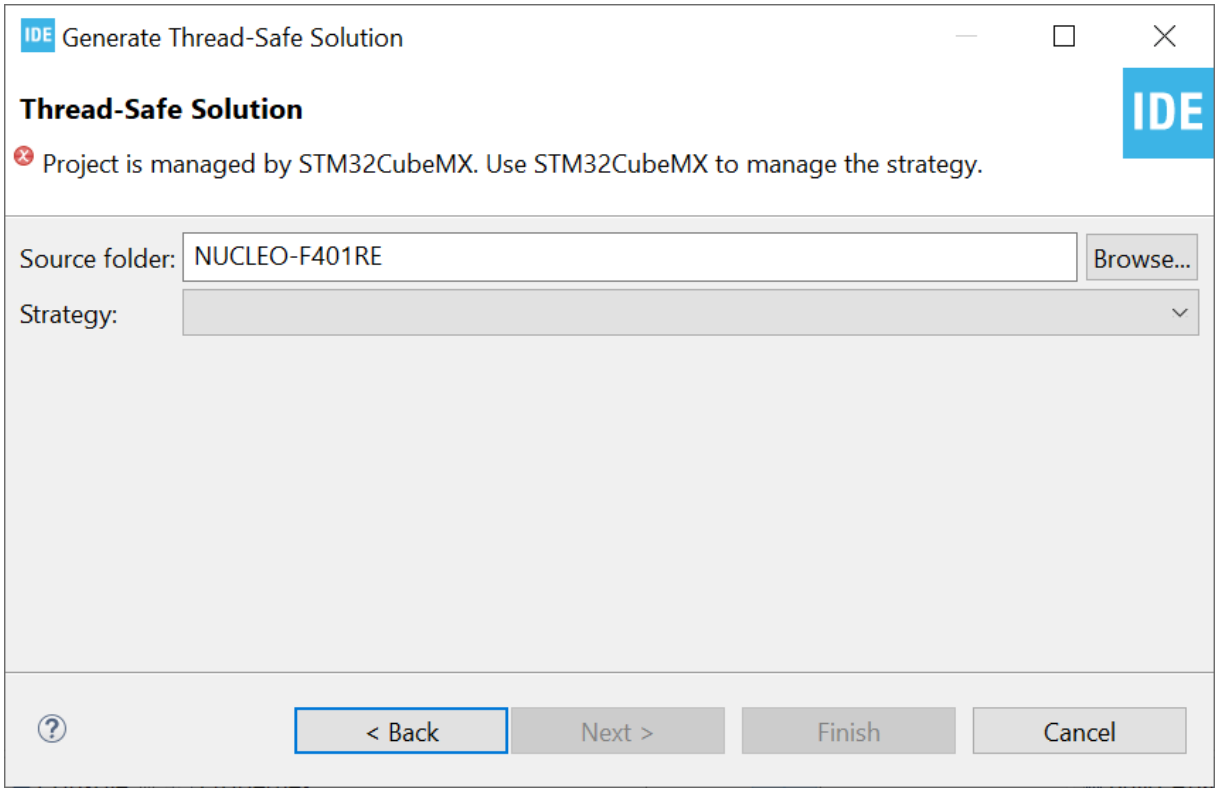
注　生成されたファイルを使用する場合、プロジェクトでいくつかの CMSIS ヘッダ・ファイルのインクルードが必要になります。これらのファイルがない場合は、手動でコピーしてプロジェクトに追加する必要があります。

図 94. スレッドセーフのファイル



STM32CubeMX でプロジェクトを管理している間にウィザードを起動すると、スレッドセーフ・ストラテジーの管理に STM32CubeMX を使用する必要があるという内容のエラーが表示されます。

図 95. スレッドセーフのエラー·ダイアログ



2.8 位置独立コード

これは、システム内で最終的なアドレス位置が定義されないアプリケーションに取り組むユーザ向けのセクションです。このような状況はブートローダを使用する場合などに発生し、システム設計者はアプリケーションの最終的な位置を定義できなければなりません。そのような場合に、位置独立コード(PIC)を使用できます。コンパイラ·オプション `-fPIE` を付加することで、コンパイラ / リンカで位置独立の実行可能ファイルを生成できます。

`-fPIE` オプションによるコンパイルは位置独立実行可能ファイルを生成するため、アプリケーションがアドレス `0x8000000` 向けにリンクされたものの `0x800 1000` に配置されたとしても、正常に動作します。

ただし、このセクションの情報は完全ではありません。ここで説明する方法はゼロに初期化されるグローバル·データ(`.bss`)を使用する場合には機能しますが、初期化されるデータを使用する場合には機能せず、他にもいくつかの制限があります。そうした制限の一つが、STM32 ツールチェーンに含まれる ランタイム ライブラリを使用できないことです。これらのライブラリは、最適化のために `-fPIE` オプションなしでビルドされるからです。システム内で位置独立コードを使用する代わりに、他の解決策を検討する価値があります。

代替案の例 :

システムが ブートローダ と Flash メモリ上の異なるスロットに複数のバージョンのアプリケーションを持つような設計の場合、アプリケーション向けに複数のビルド設定を設定した方が簡単かも知れません。ビルド設定ごとに個別のリンカ·スクリプト·ファイルを使用します。この場合、ランタイム ライブラリを使用できるため、位置独立コードを使用する必要がありません。各ビルド設定は、アプリケーションを Flash メモリ内の一意のスロットにリンクし、スロットごとに単一の `elf` ファイルを生成します。アプリケーションの新しいバージョンをスロットにダウンロードする場合は、正しい `elf` ファイルを使用する必要があります。ブートローダ の設計で、`elf` ファイル内のアドレスを検証し、スロット外のアドレスが含まれていた場合にエラーを生成することも可能です。アプリケーションは、割込みベクタ·テーブルを RAM にコピーし、アプリケーションが保存されたスロットに応じて、ベクタのコピーを更新できます。

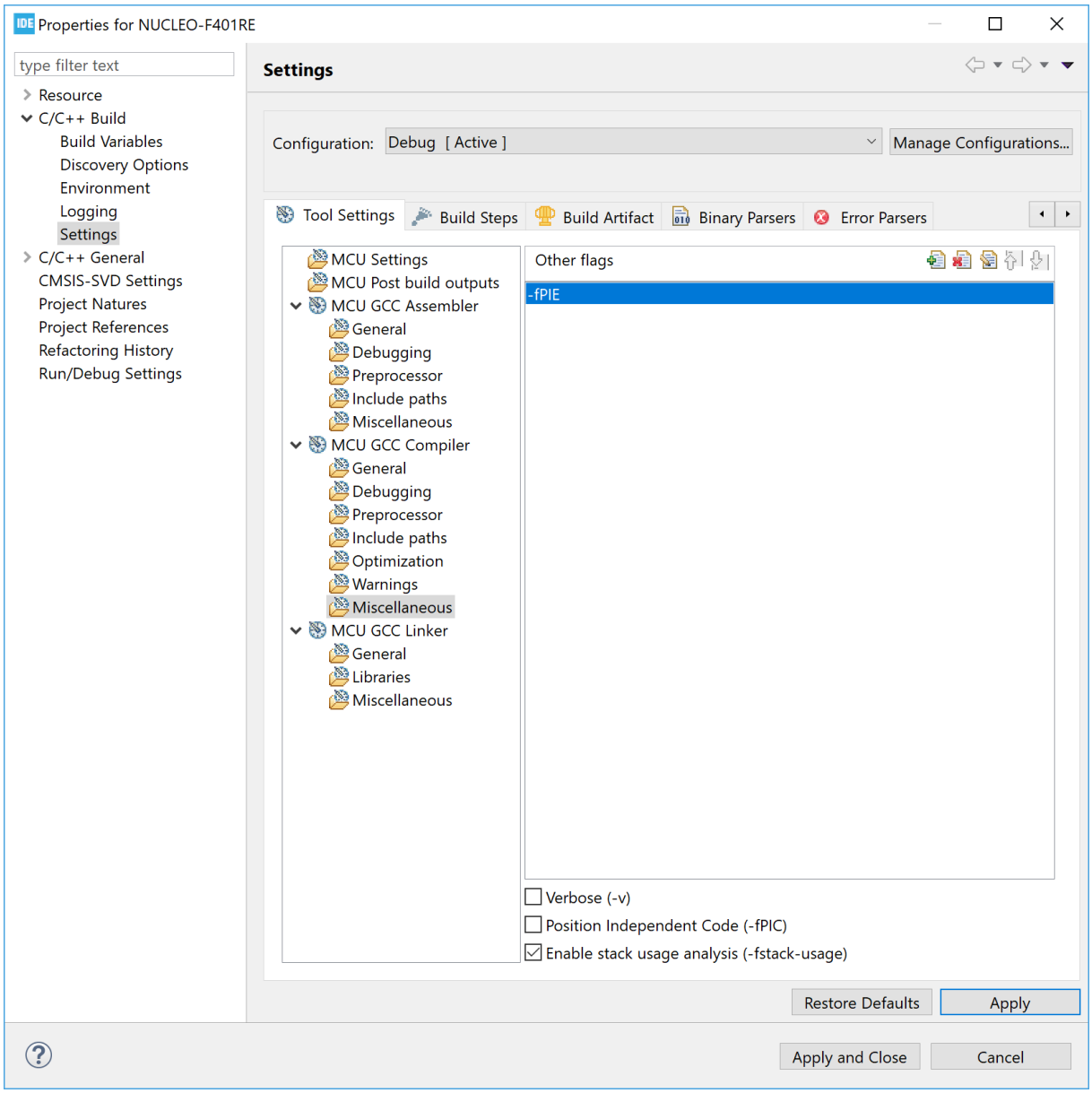
2.8.1 `-fPIE` オプションの追加

ツール設定に `-fPIE` オプションを追加するには、次の手順を実行します。

1. [Project Explorer]ビューでプロジェクトを右クリックし、Properties を選択します。
2. ダイアログの C/C++ BuildSettings を選択します。
3. パネル内の[Tool Settings]タブを選択します。
4. MCU GCC CompilerMiscellaneous を選択します。

5. Other flags フィールドに `-fPIE` を追加します。

図 96. 位置独立コード、`-fPIE`



2.8.2 ランタイム ライブラリ

C の ランタイム ライブラリは、`-fPIE` オプションなしでコンパイルされます。したがって、位置独立の実行ファイルを生成する場合、あらゆるライブラリ呼び出しを回避する必要があります。スタートアップ・コードには、通常 `__libc_init_array` への呼び出しが含まれます。この呼び出しを次の例のように削除する必要があります。

```
/* Call static constructors */
/*    bl __libc_init_array */
```

2.8.3 スタック・ポインタの設定

スタック・ポインタが適切に設定されていることを確認してください。スタック・ポインタは、下記の例のように、スタートアップ・ファイルの `Reset_Handler` で設定します。リセットすれば、スタック・ポインタがベクタ・テーブルから読み出され設定されると思い込まないでください。

```
Reset_Handler:
  ldr   sp, =_estack             /* set stack pointer */
```

2.8.4 割込みベクタ·テーブル

プログラムがオフセットされたアドレスにロードされる場合、ベクタ·テーブル内のベクタを更新する必要があります。プログラムでテーブル内の各ベクタにオフセットを加算する必要がある場合、割込みベクタ·テーブルを RAM にコピーして、そちらのベクタ·テーブルにオフセットを加算します。

ベクタのベース·レジスタについても、新しい場所に配置されたベクタ·テーブルを指すように、次の例のとおり、変更する必要があります。

```
  /* Set Vector Base Address */
  SCB->VTOR=RAM_VectorTable;
```

2.8.5 グローバル·オフセット·テーブル

グローバル·オフセット·テーブル(GOT)とは、ビルド時と `-fPIE` オプションの使用時に、通常は data セクションに保存されるアドレス·テーブルです。実行されたプログラムが、コンパイルの段階では不明だったグローバル変数のアドレスを ランタイム に見つけるために使用します。グローバル変数の位置変更が不要の場合、変数の位置はプログラムのリンク時に配置された場所と同じで構いません。その場合は、GOT テーブルを Flash メモリ領域の `.text` セクションに配置できます。

下記の例は、`.got*` セクションに関するリンカ·スクリプトの更新方法を示したものです。この例では、`GOT_START` と `GOT_END` シンボルも追加して、ツールが GOT の位置とサイズを把握できるようにしています。

```
/* The program code and other data into "ROM" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    GOT_START = .;
    *(.got*)
    GOT_END = .;
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >ROM
```

2.8.6 割込みベクタ·テーブルとシンボル

オフセットのあるコードをデバッグする場合、ロード·アドレスのオフセットと新しいシンボル·アドレスの両方を指定する必要があります。指定するシンボル·アドレスは、`.text` セクションのアドレスです。リンカ·スクリプトを変更して、`.isr_vector` が `.text` 内に配置されるように定義します。これによって、`.text` の場所が見つからないという問題が回避されます。

```
Remove the following

  .isr_vector :
  {
    .= ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

Add KEEP(*(.isr_vector)) instead to first location of .text
  /* The program code and other data into "FLASH" Rom type memory */
  .text :
```

```
{
  . = ALIGN(4);
  KEEP(*(.isr_vector)) /* Startup code */
  *(.text)             /* .text sections (code) */
  *(.text*)            /* .text* sections (code) */
  GOT_START = .;
  *(.got*)
  GOT_END = .;
  *(.glue_7)           /* glue arm to thumb code */
  *(.glue_7t)          /* glue thumb to arm code */
  *(.eh_frame)

  KEEP (*(.init))
  KEEP (*(.fini))

  . = ALIGN(4);
  _etext = .;        /* define a global symbols at end of code */
} >FLASH
```

2.8.7 位置独立コードのデバッグ

オフセット·アドレスに配置された位置独立コードをデバッグする場合、ダウンロード·オフセットと新しいシンボル·アドレスを指定する必要があります。

図 97. 位置独立コードのデバッグ

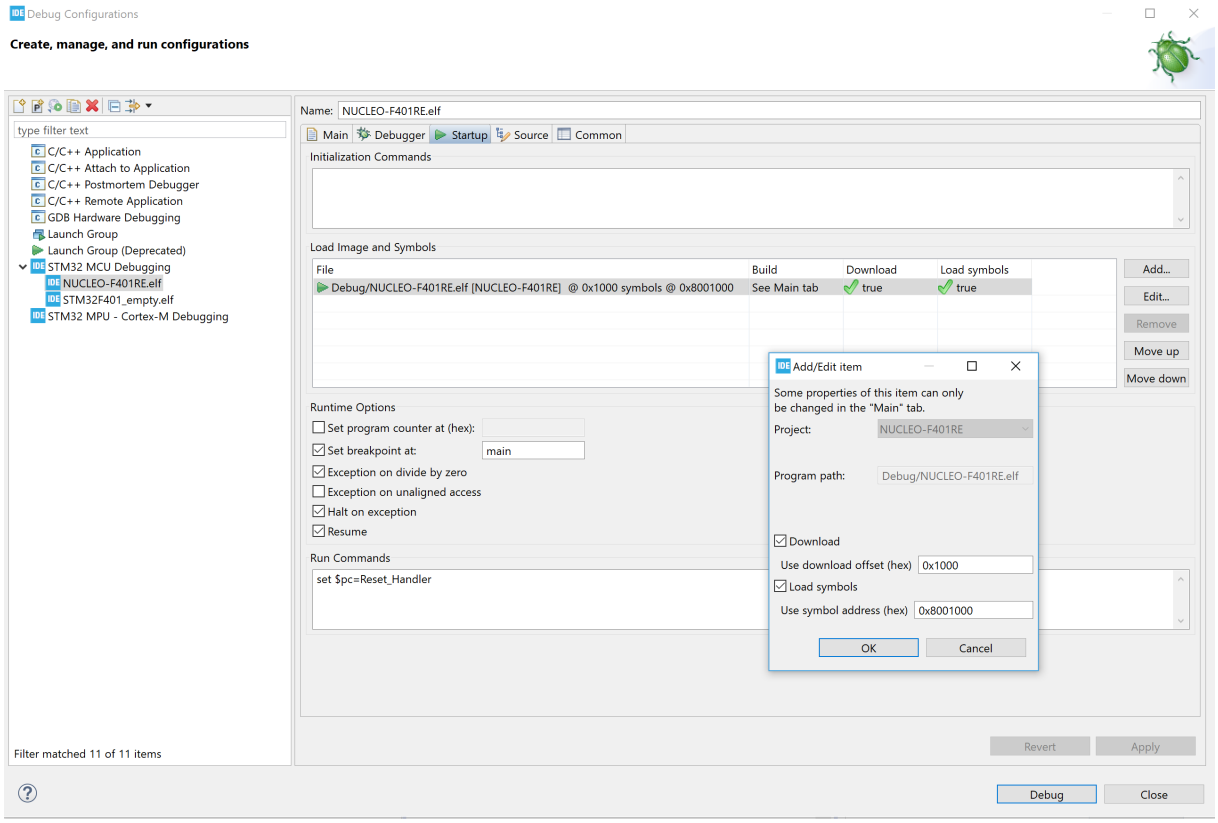


図 97 に、ダウンロード·オフセットが `0x1000`、シンボル·アドレスが `0x800 1000` の場合の例を示します。この場合、セクション 2.8.6 割込みベクタ·テーブルとシンボル で説明したように、`.isr_vector` が `.text` セクションに追加されているので、シンボル·アドレスには `0x800 1000` を設定できます。

これに対して、`.isr_vector` が `.text` の範囲外の別のセクションに配置されている場合、`.text` セクションの開始アドレスはオフセットを加算したうえで使用する必要があります。例えば、マップ·ファイルに `.text` が `0x0000 0000 0800 0194` から始まると表示されている場合、シンボル·アドレスは `0x800 1194` に設定する必要があります。

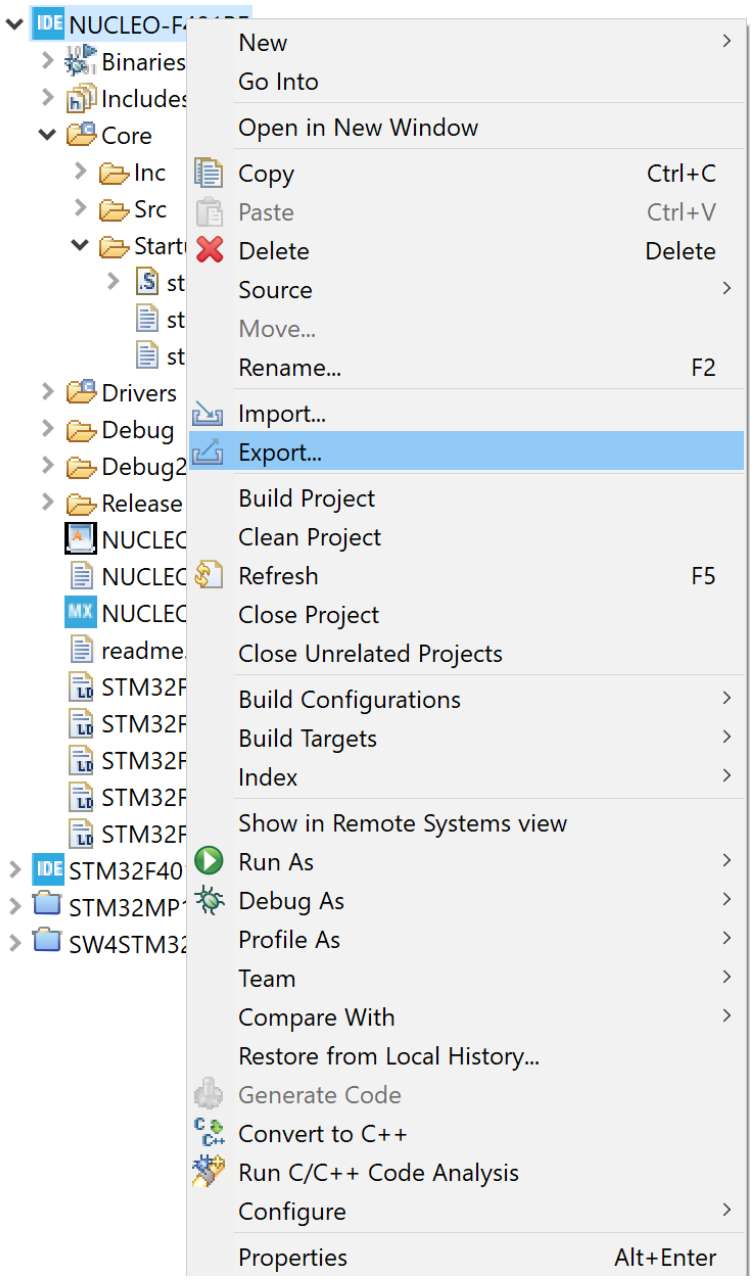
図 97 に示した例では、ブレークポイントを `main` に設定し、Run Commands フィールドで、プログラム·カウンタ(`$pc`)を `Reset_Handler` シンボルに設定しています。このシンボルには、`Reset_Handler` への正しいアドレスが含まれます。gdb はベース·シンボル·アドレスとして `0x800 1000` を使用するからです。このデバッグ設定で `$pc` を設定しない場合、プログラムがロード後に停止するように Resume のチェックボックスを無効化しておく必要があります。その場合、プログラムの開始前に[Registers]ビューでプログラム·カウンタを手動で設定する必要があります。

2.9 プロジェクトのエクスポート

プロジェクトは、さまざまな方法でエクスポートできます。このセクションでは、圧縮された zip ファイルとしてプロジェクトをエクスポートする方法を説明します。

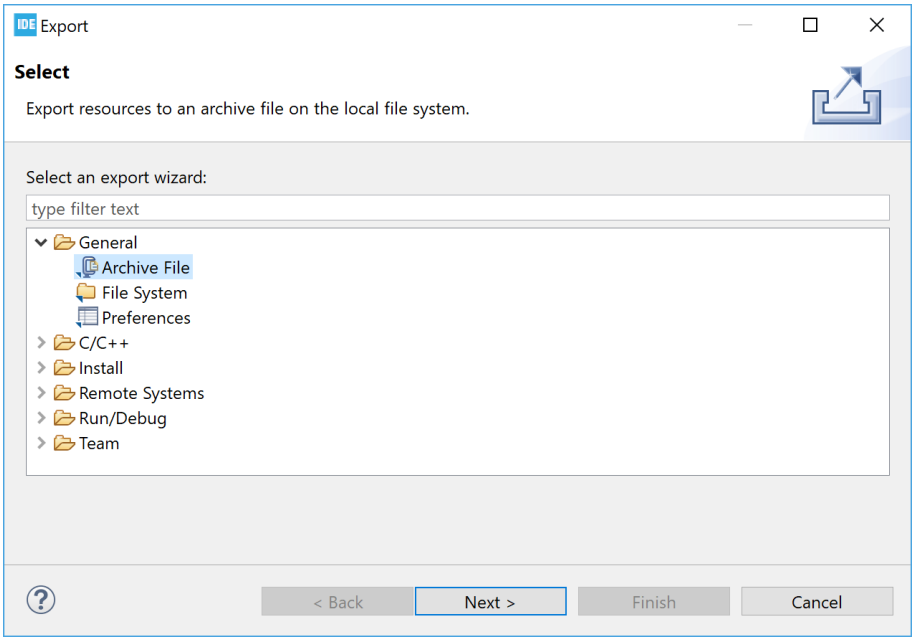
[Project Explorer]ビューでプロジェクトを右クリックし、Export... を選択します。

図 98. プロジェクトのエクスポート



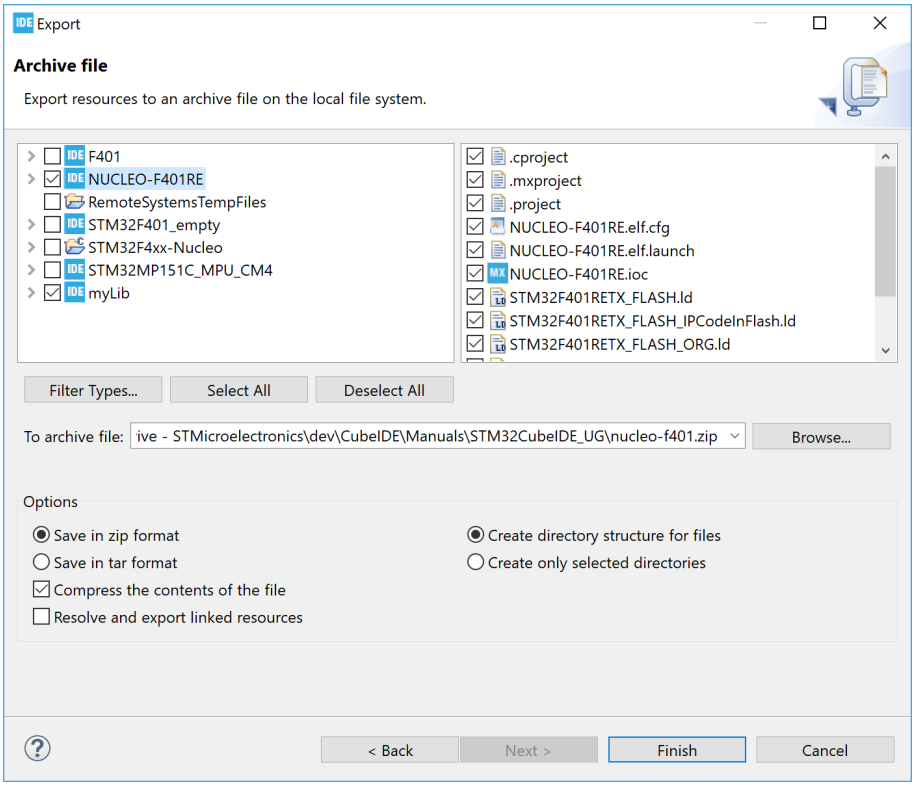
[Export]ダイアログが開きます。GeneralArchive File を選択し、Next > をクリックします。

図 99. [Export]ダイアログ



[Export]ダイアログが更新されます。エクスポートするプロジェクトを選択します。一部のプロジェクト·ファイルを除外してエクスポートすることも可能です。図 100 の例では、すべてのプロジェクトおよびライブラリ·ファイルを含めています。To archive file フィールドにファイル名を入力する必要があります。必要に応じて Browse... ボタンにより、ファイルを保存するフォルダの場所を参照します。この例では、デフォルトのオプションの値を変更せずに、そのまま使用しています。Finish をクリックして、プロジェクトをエクスポートし、zip ファイルを作成します。

図 100. アーカイブのエクスポート



2.10 既存のプロジェクトのインポート

このセクションでは、既存のプロジェクトを STM32CubeIDE のワークスペースにインポートする各種の方法を説明します。Eclipse® のプロジェクトは、Eclipse® の標準インポート機能でインポートできます。STM32CubeIDE で作成したプロジェクトのインポートには、この機能が使用されます。このプロジェクト·インポート機能を、ac6 System Workbench for STM32 プロジェクトと Atollic® TrueSTUDIO® プロジェクトのインポートにも対応するように拡張しています。これらのプロジェクトは、インポートの段階で STM32CubeIDE プロジェクトに変換されます。

他の IDE またはツールチェーンで開発した既存の `elf` ファイルをインポートしてデバッグすることも可能です。その方法の詳細は、セクション 3.8 を参照してください。

2.10.1 STM32CubeIDE プロジェクトのインポート

プロジェクトは、さまざまな方法でインポートできます。このセクションでは、zip 圧縮ファイルとしてエクスポートされたプロジェクトをインポートする方法を説明します。

- [Import]ダイアログを開くには、メニュー FileImport… を使用します。
- もう一つの方法は、[Project Explorer]ビューを右クリックして、Import... を選択します。

図 101. プロジェクトのインポート

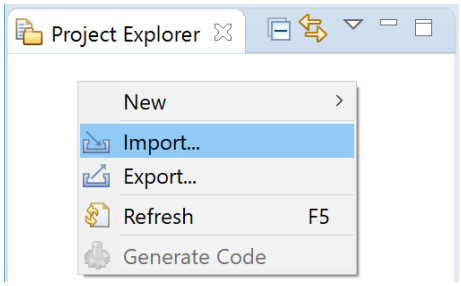


図 102. [Import]ダイアログ

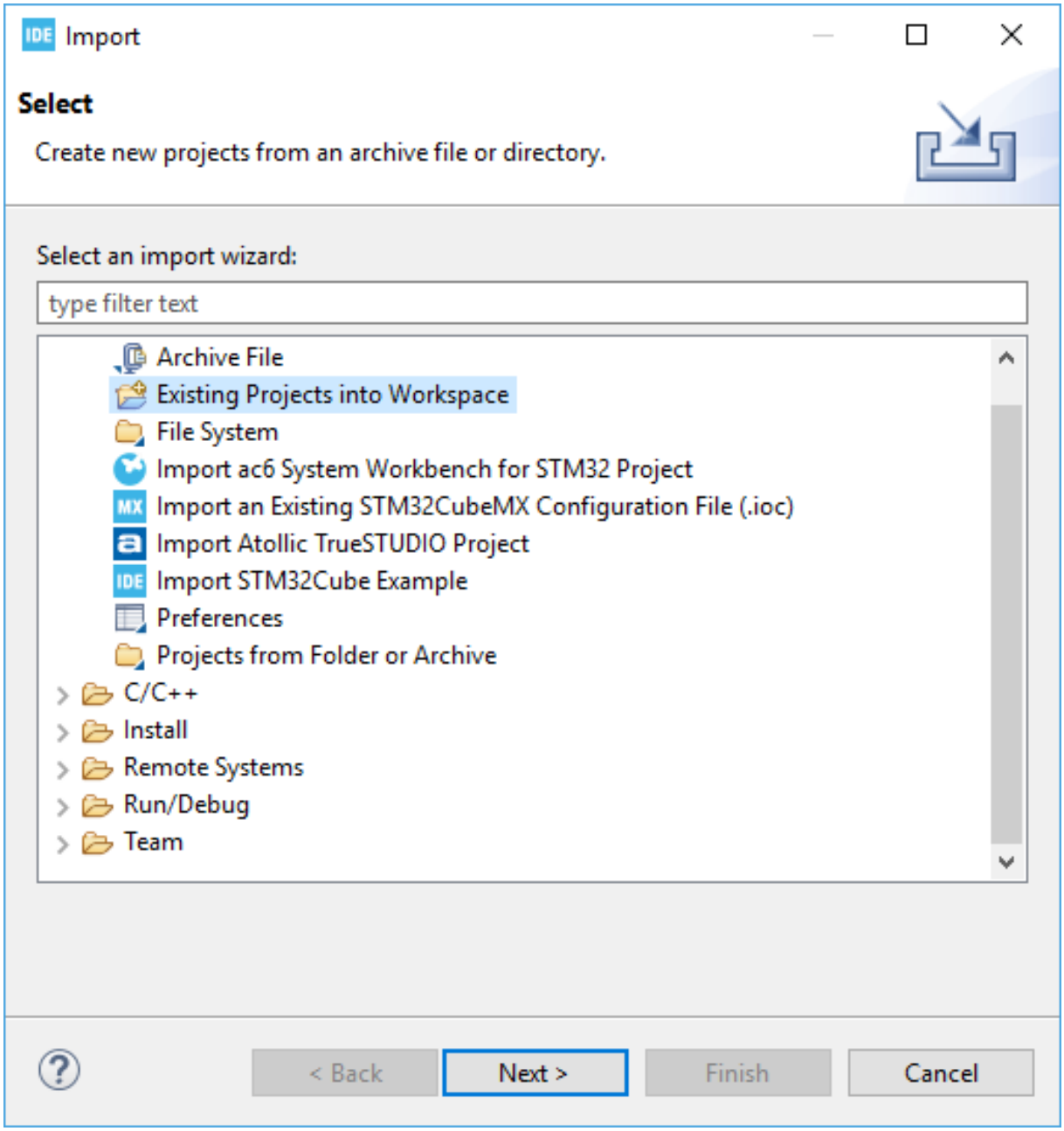
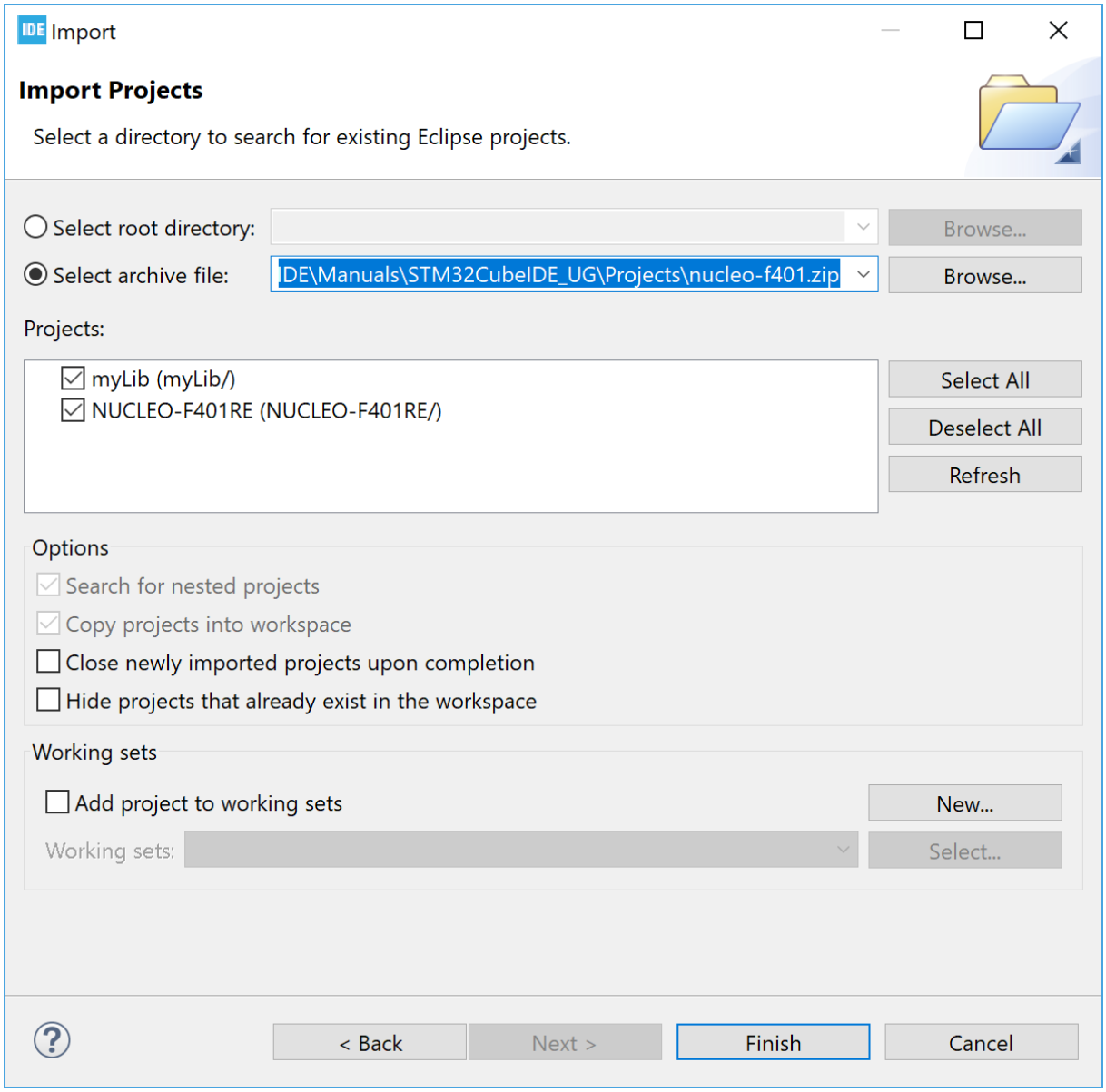


図 103. プロジェクトのインポート



2.10.2 System Workbench および TrueSTUDIO® プロジェクトのインポート

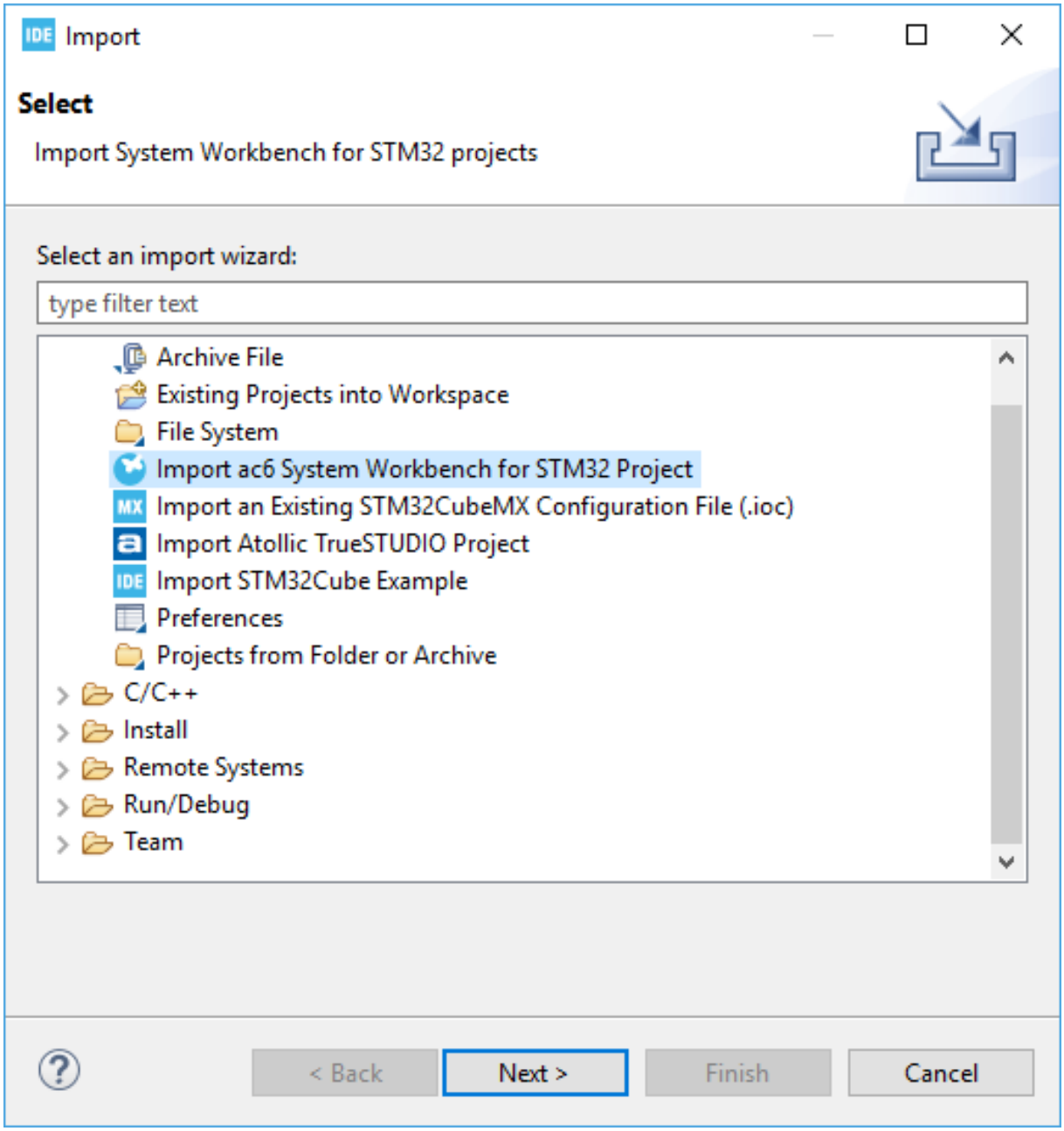
ac6 System Workbench for STM32 プロジェクト、または Atollic® TrueSTUDIO® プロジェクトを STM32CubeIDE にインポートする場合は、プロジェクトのコピーを使用して作業することを推奨します。

1. プロジェクトのコピーは、プロジェクト·フォルダをコピーするか、プロジェクトを zip ファイルにエクスポートして作成します。
2. STM32CubeIDE へのインポートには、コピーしたプロジェクトを使用します。

コピーしたプロジェクトをインポートするには、メニュー FileImport… を使用するか、[Project Explorer]ビューを右クリックして[Import]ダイアログを開きます。

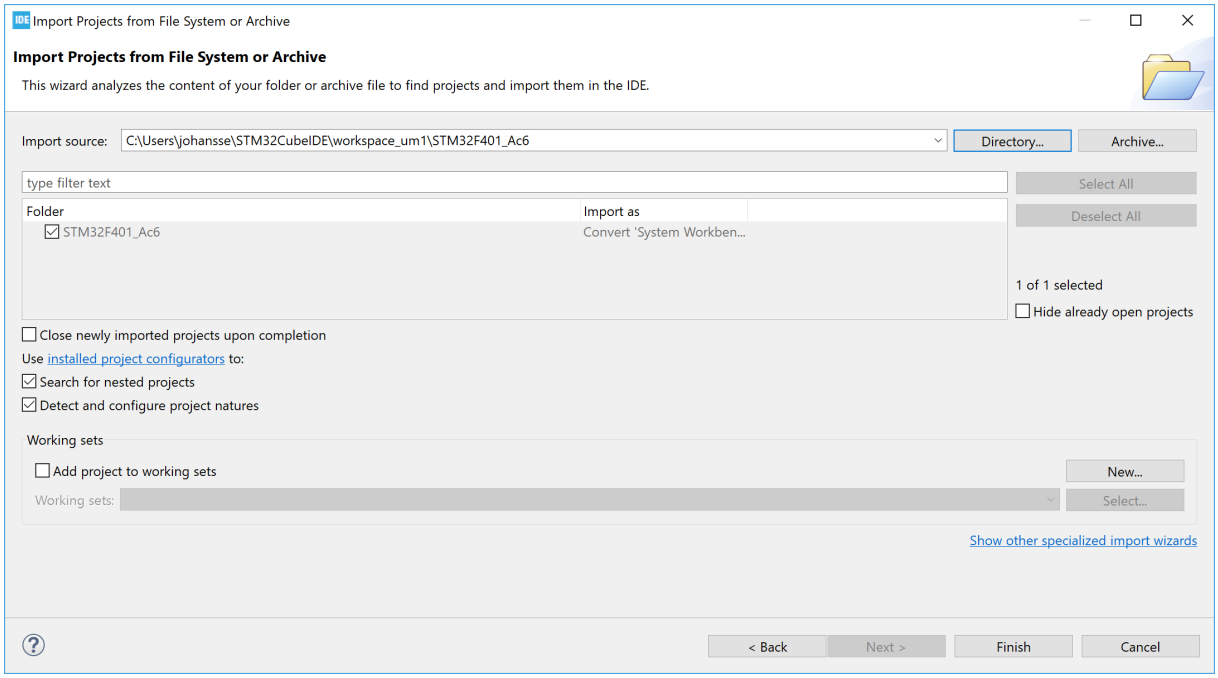
プロジェクトの作成に使用した元のツールに応じて、Import ac6 System Workbench for STM32 project または Import Atollic TrueSTUDIO project を選択し、Next > をクリックします。

図 104. System Workbench プロジェクトのインポート(1/3)



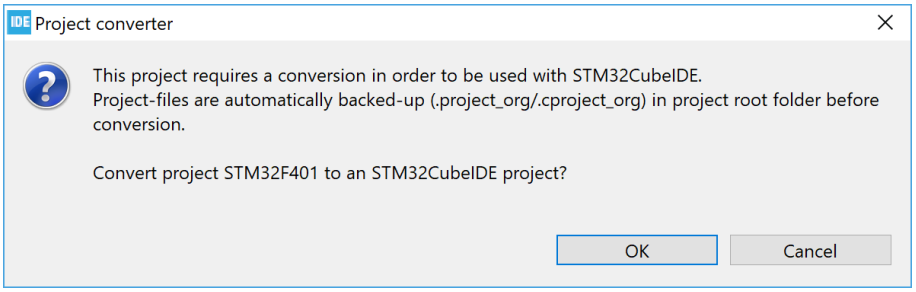
この例では、ac6 プロジェクトを STM32CubeIDE のワークスペースにコピーするため、Directory... ボタンを使用してプロジェクト STM32F401_Ac6 を選択しています。インポート·ウィザードは、これが System Workbench プロジェクトであることを検出します。

図 105. System Workbench プロジェクトのインポート(2/3)



Finish をクリックして、[Project converter]ダイアログを開きます。

図 106. System Workbench プロジェクトのインポート(3/3)



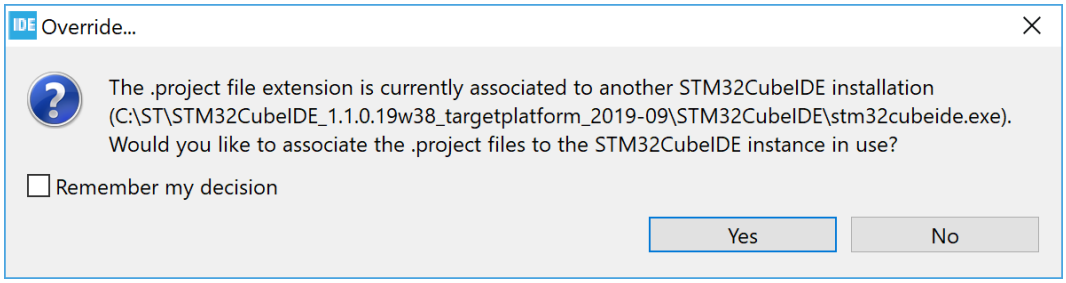
OK をクリックして、プロジェクトを STM32CubeIDE のプロジェクトに変換します。

ac6 System Workbench for STM32([ST-06])および Atollic® TrueSTUDIO®([ST-05])から STM32CubeIDE への移行方法について解説した移行ガイドが 2 つあります。これらのガイドは Information Center の[Technical Documentation]ページから開くことができます。

2.10.3 プロジェクト･ファイルの関連付けによるインポート

STM32CubeIDE の起動時、`.cproject` および `.project` ファイルをプログラムに関連付ける必要があるかどうかを尋ねるポップアップ･ウィンドウが表示されます。

図 107. プロジェクト·ファイルの関連付けによるインポート



関連付けを選択した場合、PC のファイル·ブラウザで `.project` ファイルをダブルクリックすると、STM32CubeIDE により現在のワークスペースへのプロジェクトのインポートが開始されます。プロジェクト·コンバータがプロジェクトを調べ、STM32CubeIDE 向けに作成されたものであった場合、直接インポートされます。プロジェクトが他のツールで作成されたものの場合、プロジェクト·コンバータは、既知のプロジェクト·フォーマットであるかどうかの識別を試みます。既知のフォーマットの場合、セクション 2.10.2 System Workbench および TrueSTUDIO プロジェクトのインポート の説明のように、プロジェクトを STM32CubeIDE プロジェクトに変換します。

2.10.4 GCC not found in path エラーの防止

古いプロジェクトをインポートする場合、[Problems]ビューに Program "gcc" not found in PATH（パスに gcc プログラムが見つかりません）という内容のエラーが表示されることがあります。このエラーは、プロジェクトに廃止予定の探索方法が設定されていることが原因で発生します。このエラーは [Window]>[Preferences]と[Project Properties]の設定を更新することで解消できます。

1. WindowPreferences を開きます。[Preferences]ダイアログで、C/C++Property Pages Settings を選択し、Display “Discovery Options” page チェックボックスを有効にします。
2. Project PropertiesC/C++ BuildDiscovery Options を開き、Automate discovery of paths and symbols チェックボックスを無効にします。

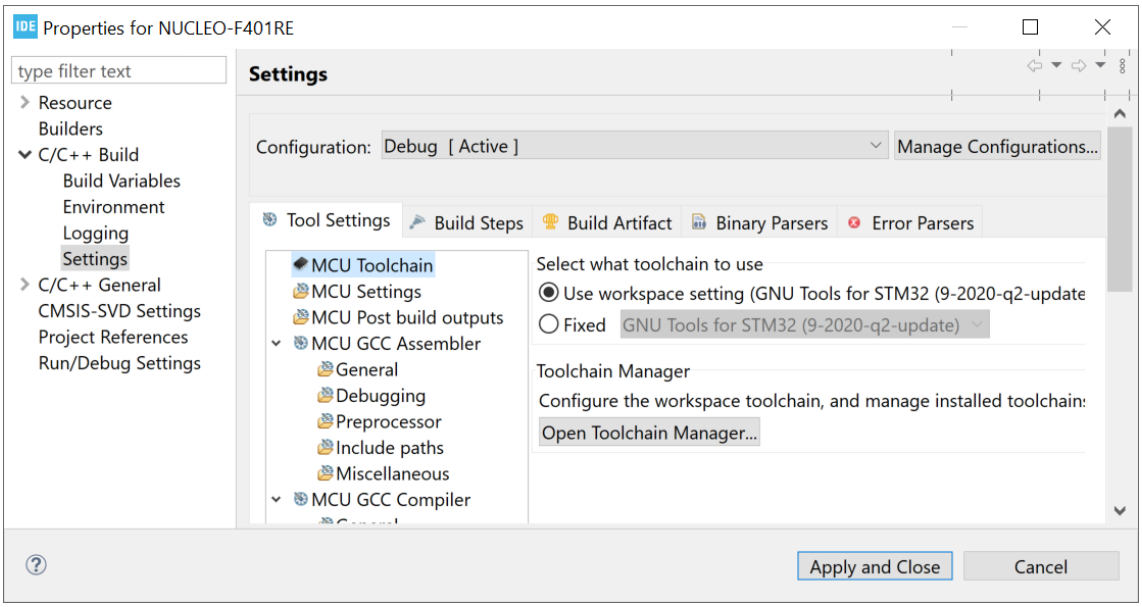
2.11 Toolchain Manager

Toolchain Manager は、ツールチェーンのインストールとアンインストール、プロジェクトのビルド時に使用するデフォルトのワークスペース·ツールチェーンの選択に使用します。

Toolchain Manager をプロジェクト·プロパティの[Tool Settings]タブから開くには、次の手順を実行します。

1. MCU Toolchain ノードを選択します。

図 108. Toolchain Manager を開く

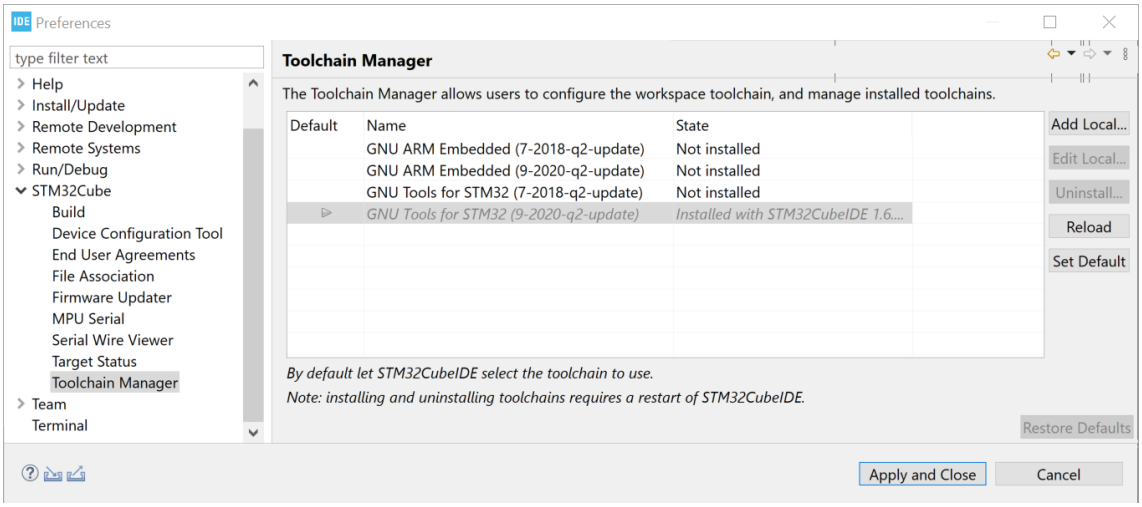


2. Open Toolchain Manager... をクリックします。

Toolchain Manager は、メニュー WindowPreferences から開くこともできます。

1. STM32CubeToolchain Manager を選択します。

図 109. Toolchain Manager



Toolchain Manager の各列の説明を 表 4 に示します。

表 4. Toolchain Manager の列の詳細

列名	説明
Default	緑 / 灰色の矢印記号はデフォルトのワークスペース·ツールチェーンを示します。 矢印の色の意味は次のとおりです。 • 緑色 : ツールチェーンがユーザにより手動でデフォルトに設定された場合 • 灰色 : ツールチェーンが STM32CubeIDE のロジックによってデフォルトに選択された場合
Name	ツールチェーンの名前です。
State	ツールチェーンの状態です。ST マイクロエレクトロニクスのオンライン·リポジトリからダウンロードできるツールチェーンは installed または not installed と表示されます。ユーザによって追加されたローカル·ツールチェーンには local と表示されます。

Toolchain Manager の各ボタンの説明を 表 5 に示します。

表 5. Toolchain Manager のボタン情報

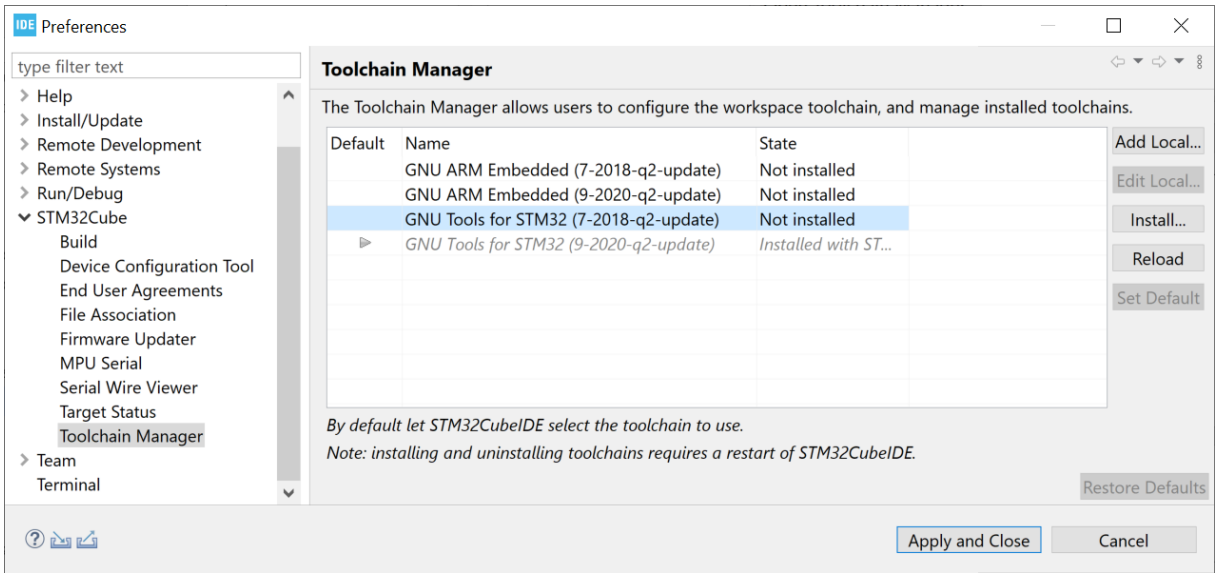
ボタン名	説明
Add Local...	ローカル·ツールチェーンへの参照を追加します。
Edit Local...	ローカル·ツールチェーンへの参照を編集します。
Install... Uninstall... Remove...	ボタンに表示される文字は選択したツールチェーンの種類によって決まります。このボタンは次の目的で使用します。 • リポジトリから提供される選択されたツールチェーンをインストール / アンインストールします。 • 選択したローカル·ツールチェーンを削除します。
Reload	リポジトリからツールチェーンのリストをリロードします。
Set Default	選択したツールチェーンをデフォルトで使用するように設定します。
Restore Defaults	ツールチェーンのデフォルト設定を復元します。

ボタン名	説明
Apply and Close	選択を適用してダイアログを閉じます。
Cancel	ダイアログ操作を取り消します。

2.11.1 新しいツールチェーンのインストール

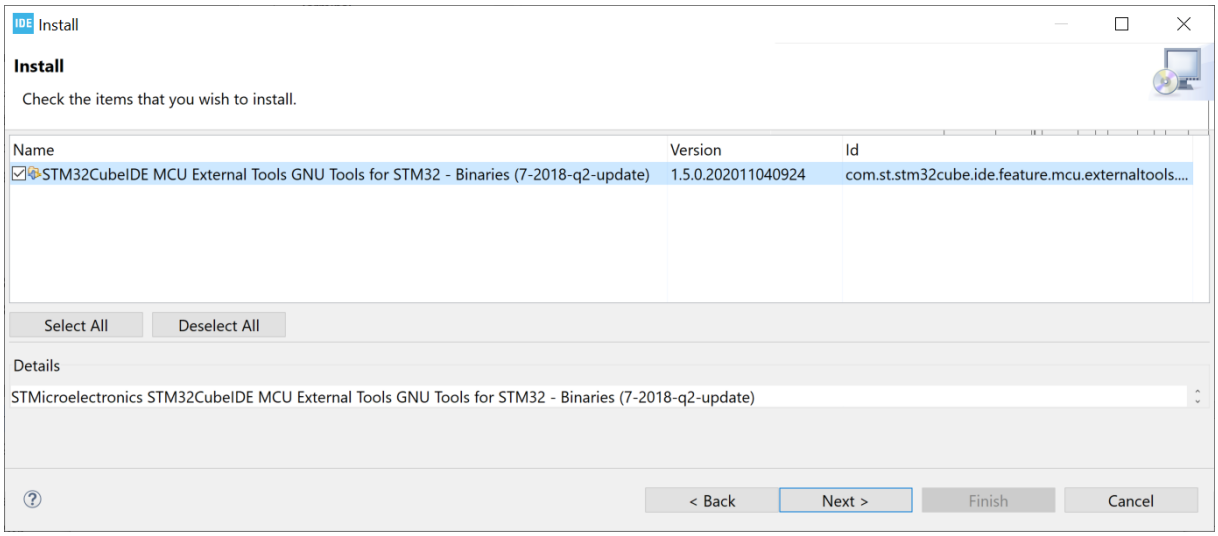
新しいツールチェーンをインストールするには、Toolchain Manager を開きます。

図 110. ツールチェーンのインストール



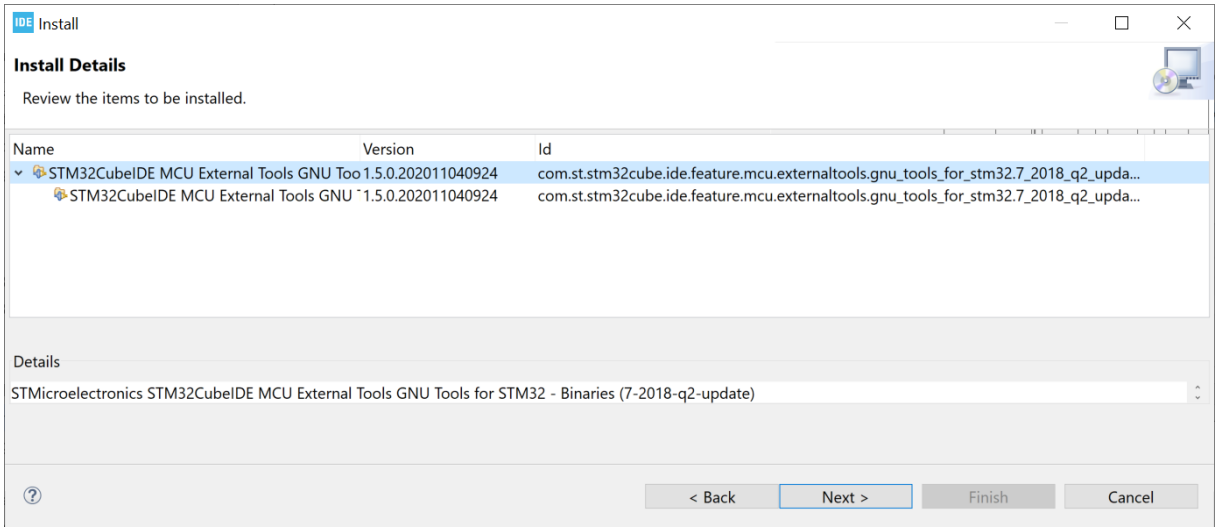
インストールするツールチェーンを選択して Install... をクリックします。[Install]ダイアログが開き、インストールされる項目が表示されます。

図 111. インストール項目の選択



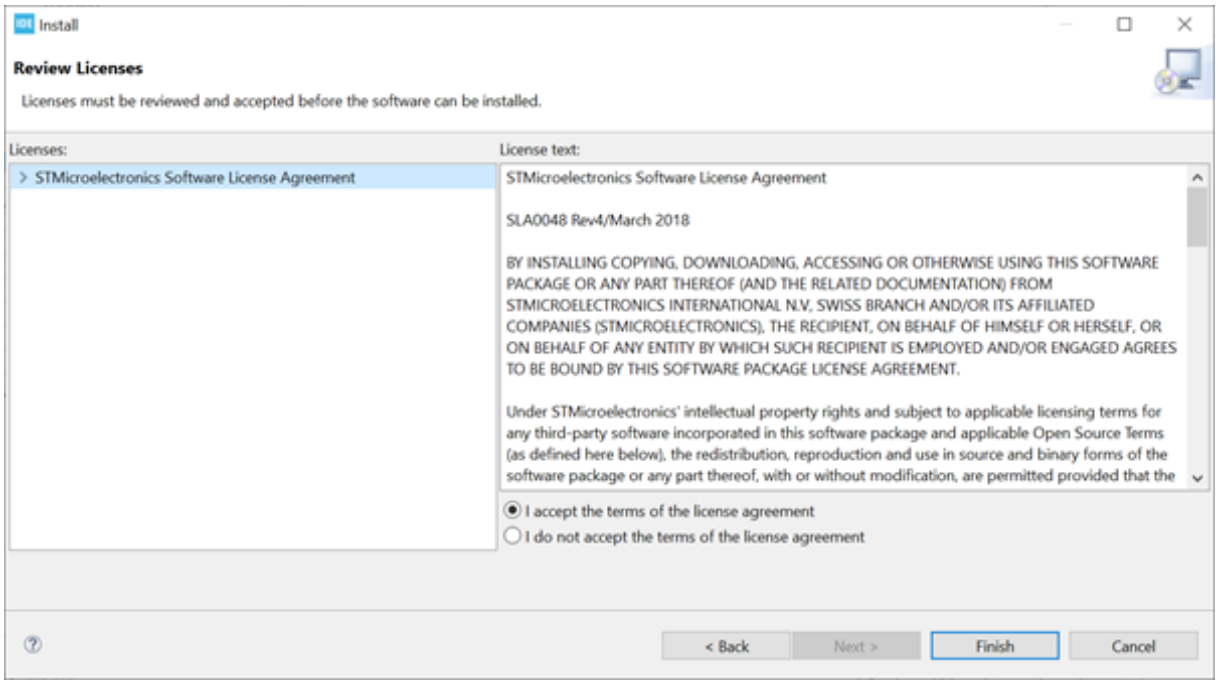
インストールする項目にチェックを入れ Next をクリックします。

図 112. インストール項目の確認



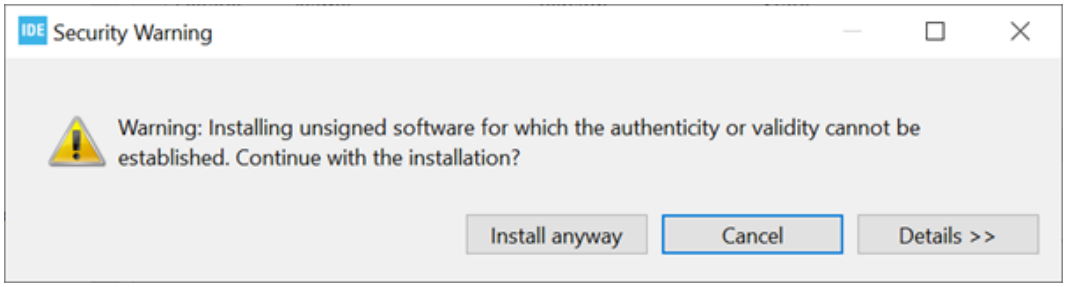
インストールする項目を確認し、Next をクリックします。

図 113. 使用許諾契約の確認と同意



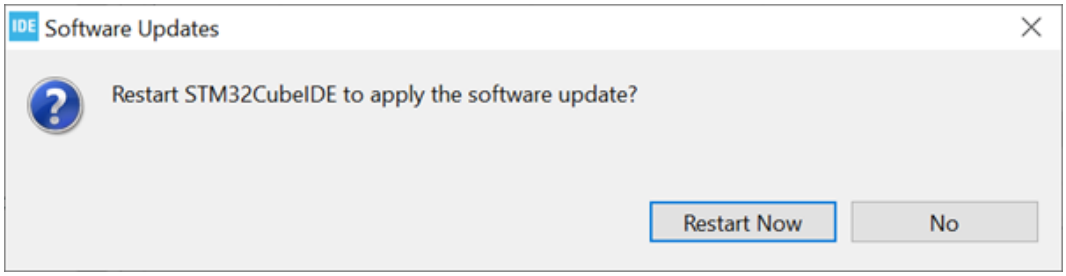
使用許諾契約を確認し、I accept the terms of the license agreements を選択して Finish をクリックします。
この時点でソフトウェアのインストールが開始されます。STM32CubeIDE のウィンドウ下部に進捗バーが表示され、インストールがどの程度完了したかがわかります。インストールが完了するまで、お待ちください。
インストールの完了前に次の警告が表示される場合があります。

図 114. セキュリティ警告



その場合、インストールを完了するには Install anyway をクリックします。しばらくすると、次のようなダイアログが表示されます。

図 115. ソフトウェア更新適用のための再起動



Restart Now をクリックして、インストールしたツールチェーンを STM32CubeIDE で使用できるようにします。STM32CubeIDE が再起動し、新しいツールチェーンを使用できるようになります。
インストールされたことを確認するために Toolchain Manager を開きます。

図 116. インストールされたツールチェーン

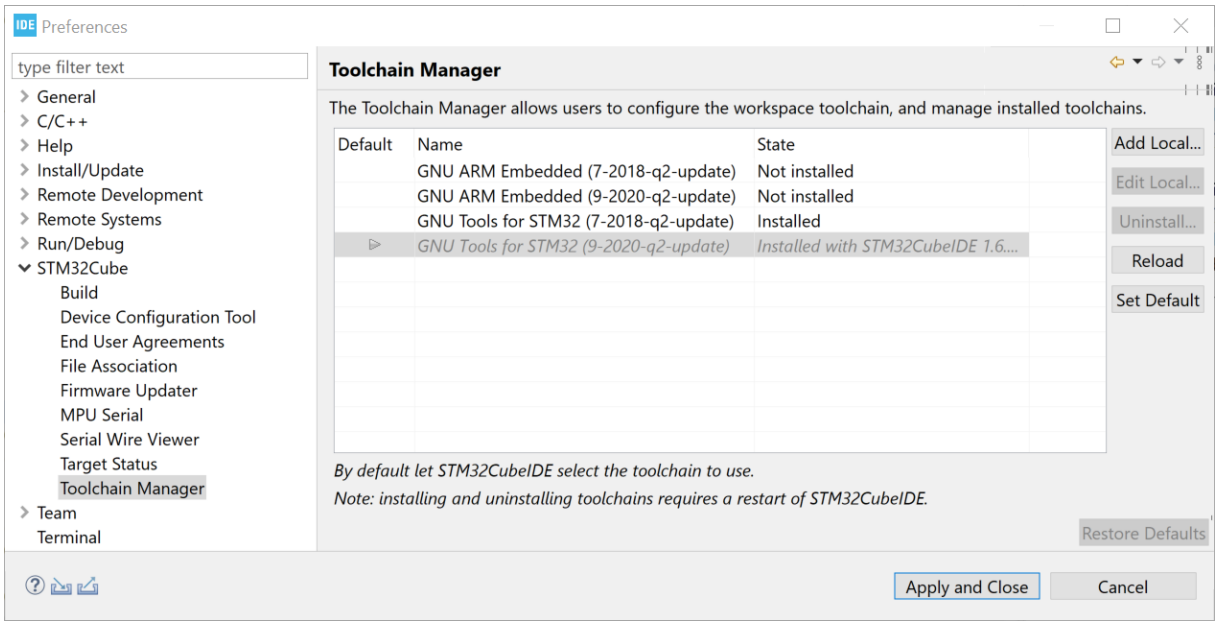


図 116 によると、この例では 2 つのバージョンの GNU Tools for STM32 がインストール済みであることがわかります。

2.11.2 デフォルト·ツールチェーンの管理

Toolchain Manager では、[Default]列の矢印によって、デフォルトのワークスペース·ツールチェーンが示されます。

図 117. デフォルトのツールチェーン

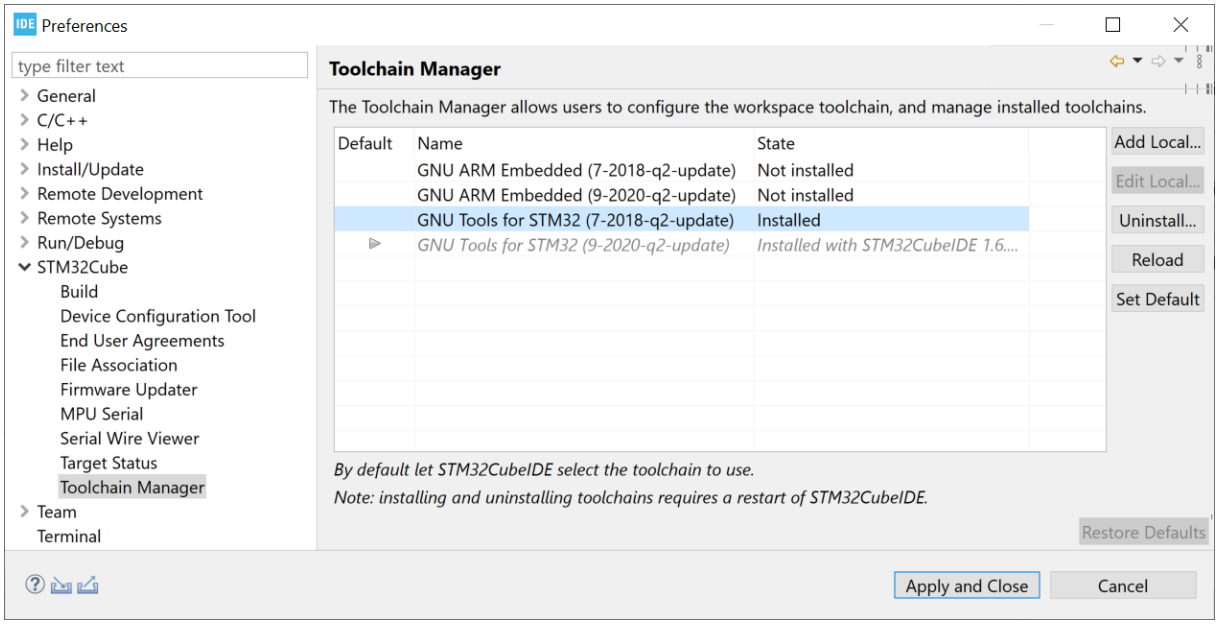
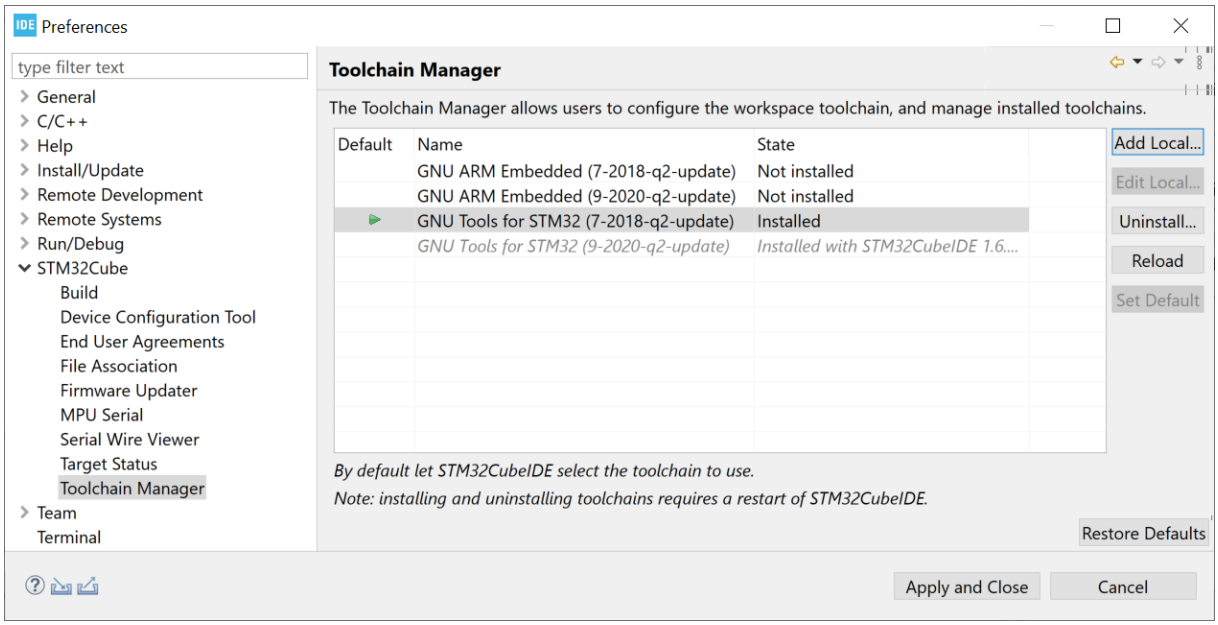


図 117 の場合、GNU Tools for STM32 version 9-2020-q2-update がデフォルトのワークスペース·ツールチェーンです。GNU Tools for STM32 version 7-2018-q2-update の行が青色に強調表示されているのは、このツールチェーンが選択されていることを表します。この表の任意の行をマウスによって選択できます。

Set default をクリックすると、この選択中のツールチェーンがデフォルトのワークスペース·ツールチェーンとして使用されるようになり、Toolchain Manager の[Default]列に矢印が表示されます。

図 118. 更新されたデフォルトのツールチェーン

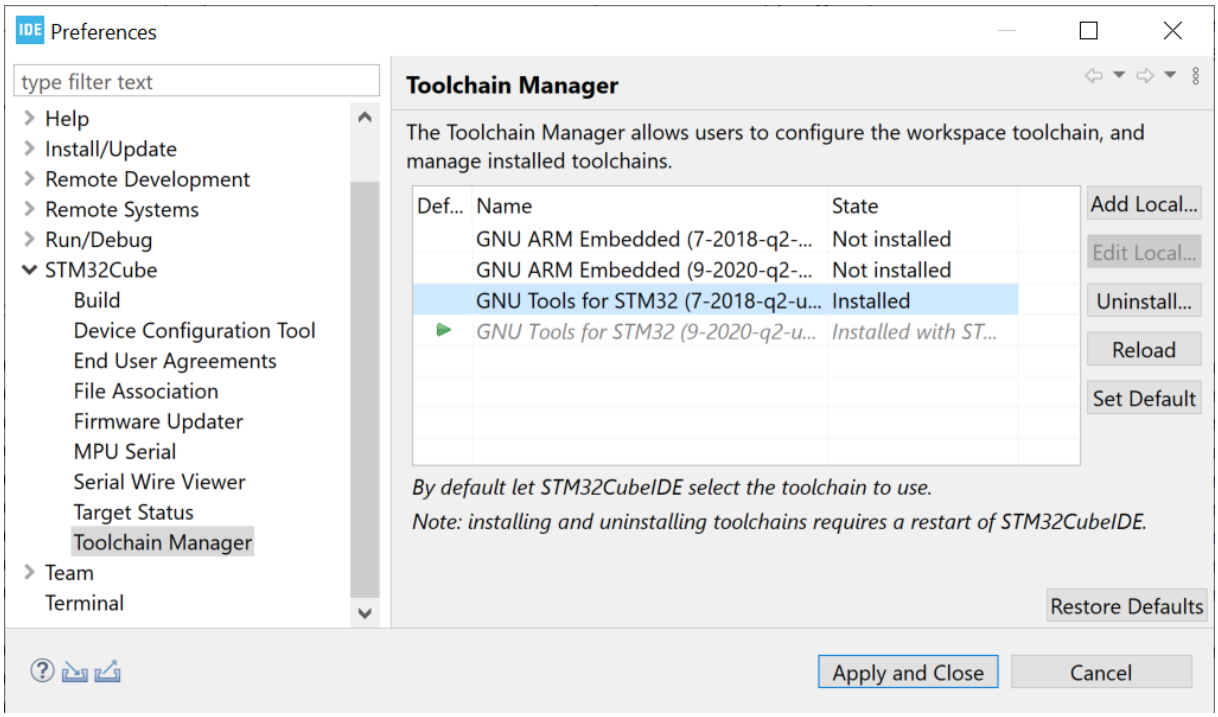


Apply and Close をクリックして設定を適用し、デフォルトのワークスペース·ツールチェーンとして設定されるツールチェーンを更新します。

2.11.3 ツールチェーンのアンインストール

GNU Tools for STM32 ツールチェーンはアンインストールできません。このツールチェーンは、デフォルトで STM32CubeIDE とともにインストールされます。インストール済みの他のツールチェーンは、いずれもアンインストールが可能です。

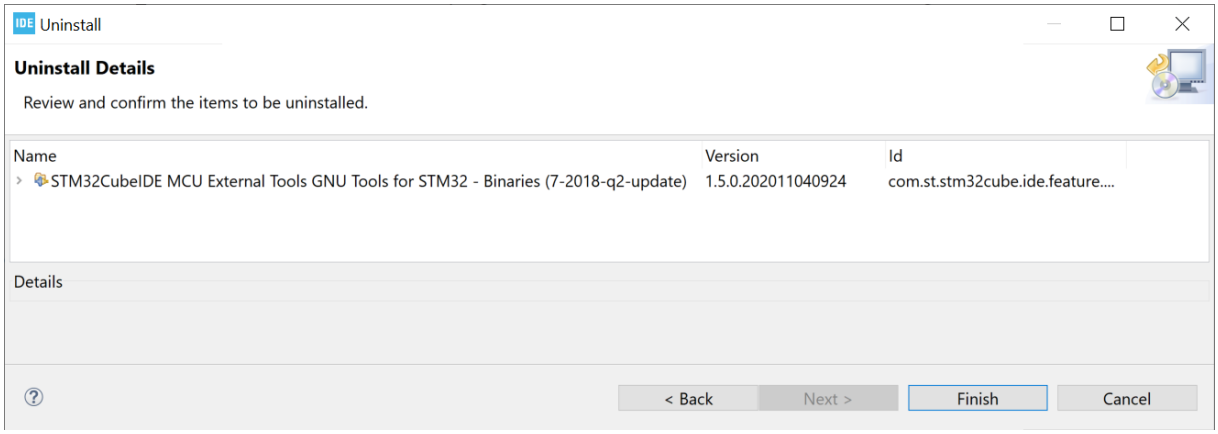
図 119. ツールチェーンのアンインストール



ツールチェーンをアンインストールするには、Toolchain Manager で目的のツールチェーンを選択し、Uninstall... をクリックします。

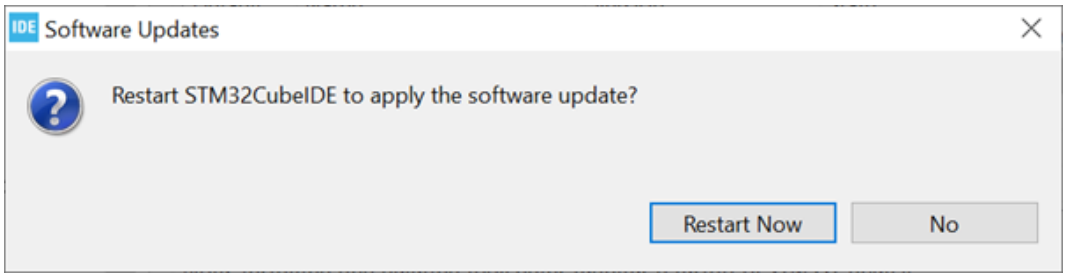
これによって[Uninstall]ダイアログが開きます。

図 120. アンインストールの詳細



Finish をクリックしてソフトウェアのアンインストールを開始します。[Software Updates]ダイアログが表示されます。

図 121. ソフトウェア更新



Restart Now をクリックして、ソフトウェアに加えた変更を適用します。
製品が再起動します。
アンインストールされたことを確認するために Toolchain Manager を開きます。

図 122. アンインストールされたツールチェーン

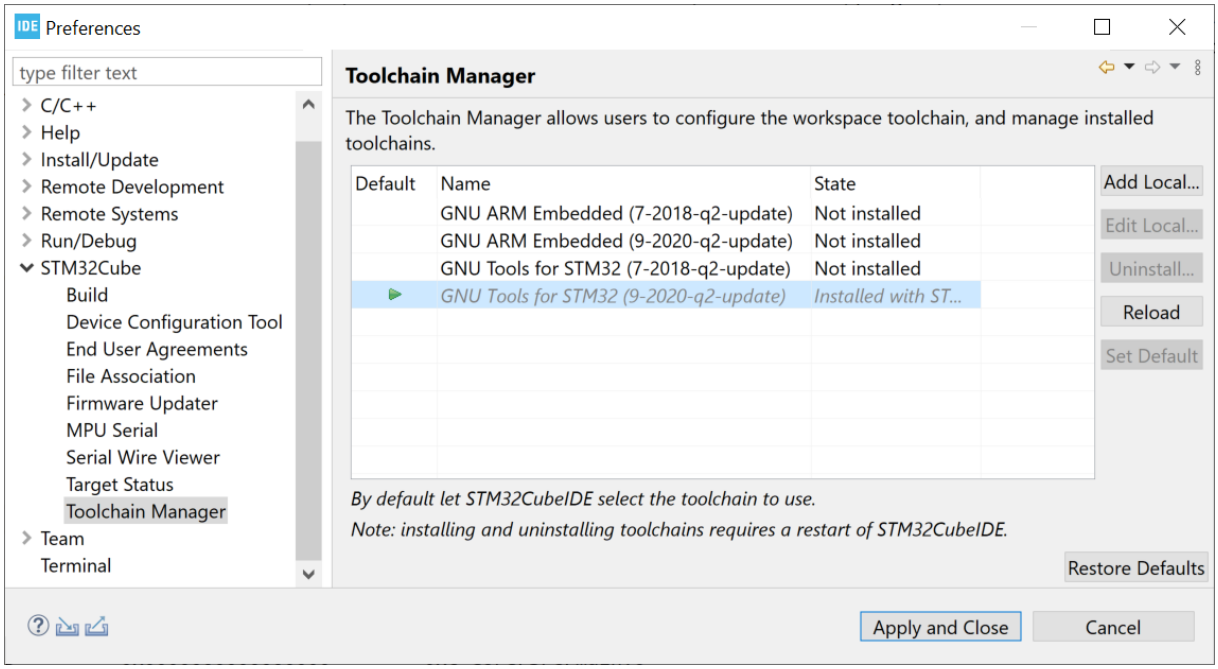


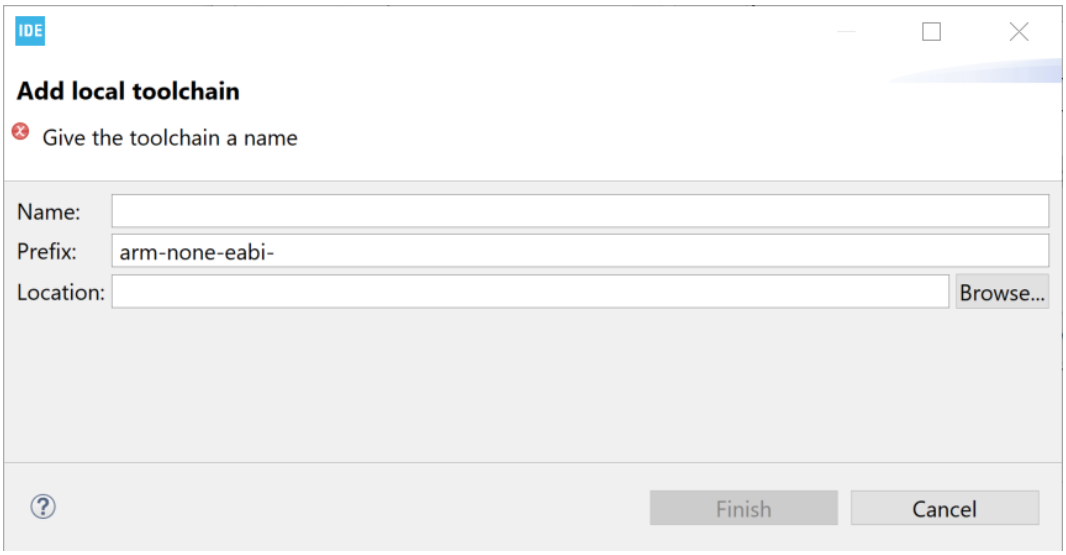
図 122 から、GNU Tools for STM32 は 1 つのバージョンだけインストールされていることがわかります。

2.11.4 ローカル·ツールチェーンの使用

インストール済みのローカルの GNU ARM ツールチェーンを追加して使用できます。ローカル·ツールチェーンを追加するには、次の手順を実行します。

1. Toolchain Manager を開き、Add Local... ボタンをクリックします。

図 123. ローカル・ツールチェーンの追加



2. 名前を追加し、場所を指定します。

図 124. ローカル·ツールチェーンの場所の指定

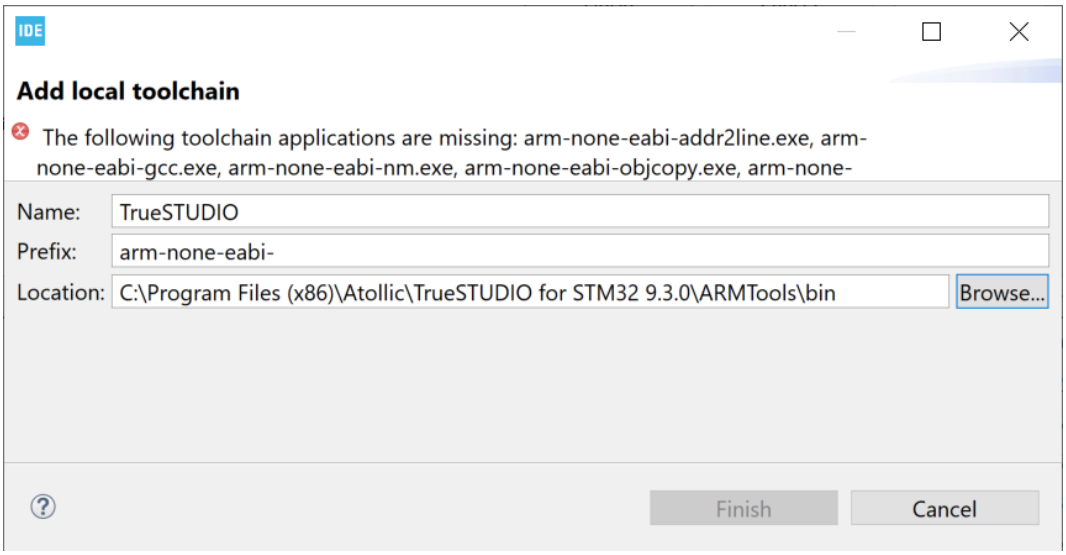
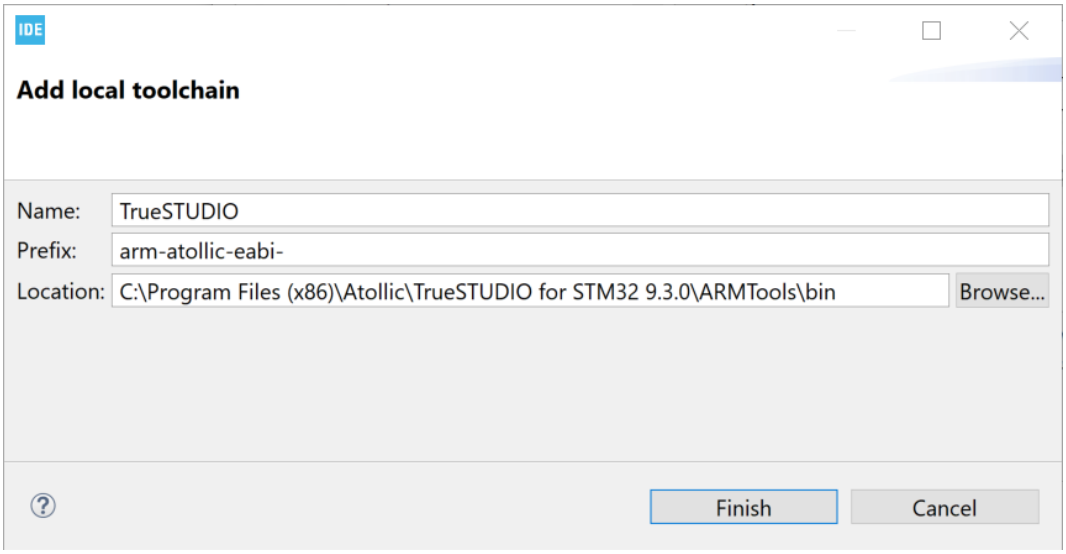


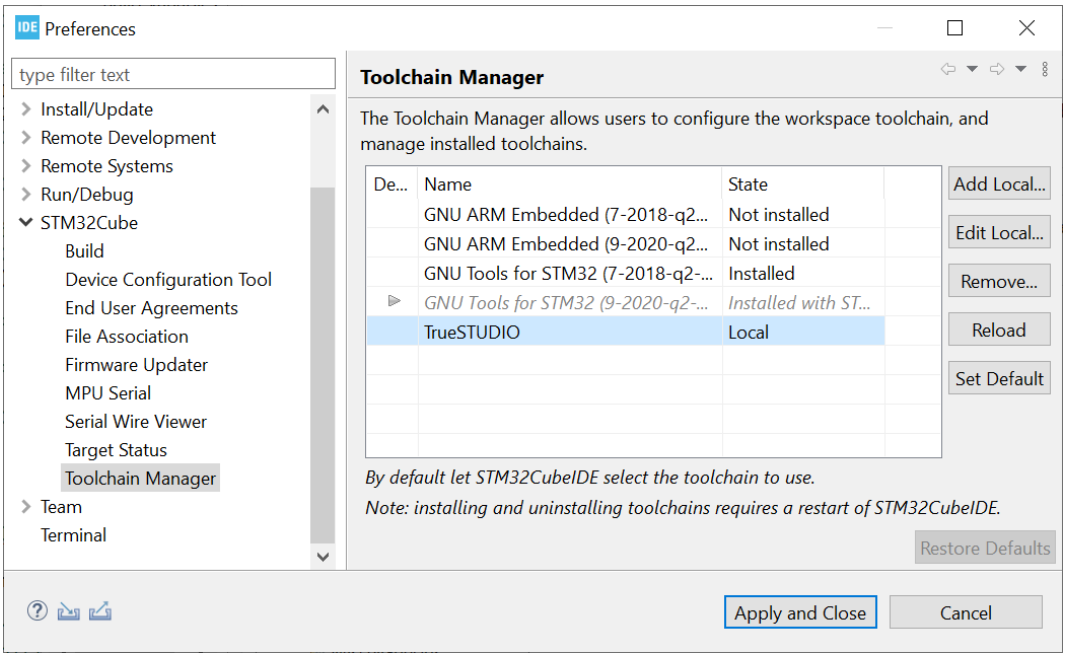
図 124 のように、名前に関する問題が発生することがあります。上記の例では、接頭辞が誤っているためにツールチェーン·アプリケーションの検証が妨げられていることが原因です。
ツールチェーンの接頭辞を変更してください。接頭辞はダッシュ(-)で終わる必要があります。

図 125. ローカル·ツールチェーン接頭辞の指定



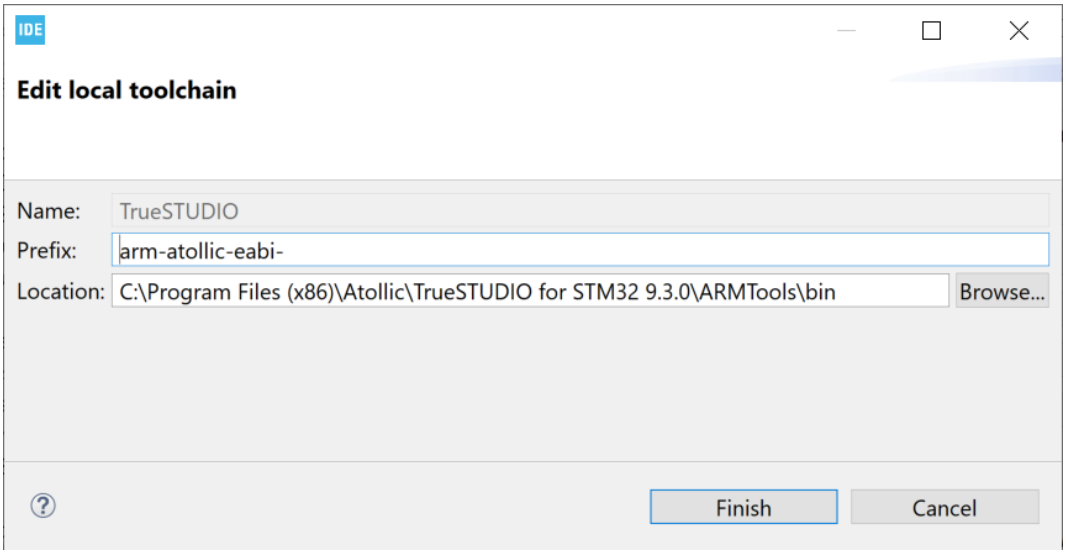
3. Finish をクリックします。

図 126. 追加されたローカル·ツールチェーン



4. ローカル·ツールチェーンを編集するには Edit Local... ボタンを使用します。[Edit local toolchain]ダイアログが開き、[Prefix]と[Location]を変更できます。

図 127. ローカル·ツールチェーンの編集



5. [Prefix]または[Location]を変更し、Finish をクリックして、ローカル·ツールチェーンの設定を更新します。

2.11.5 ネットワーク·エラー

更新サイトへのアクセスに問題が発生すると[Network error...]ダイアログが表示されます。

図 128. ツールチェーンのネットワーク·エラー

ネットワークの設定を確認してください。ネットワークのプロキシの設定方法については、セクション 1.5.3 プレファレンス - ネットワーク·プロキシの設定 で説明しています。

3 デバッグ

3.1 デバッグの概要

STM32CubeIDE は、GDB コマンドライン·デバッガに基づいた、グラフィック表示の強力なデバッガを備えています。ST-LINK および SEGGER J-Link JTAG プローブに対応した GDB サーバ も付属しています。

GDB サーバ とは、PC 上の GDB をターゲット·システムに接続するプログラムです。STM32CubeIDE のデバッグ·セッションによってローカル GDB サーバ を自動起動したり、リモート GDB サーバ に接続したりすることが可能です。

リモート GDB サーバ は同じ PC 上またはネットワークを介してアクセスできる別の PC 上で動作させることができます。リモート PC は、ホスト名または IP アドレスとポート番号で指定します。リモート GDB サーバ に接続する場合、STM32CubeIDE でデバッグ·セッションを開始する前に、この GDB サーバ を起動しておく必要があります。

ローカル·デバッグの自動起動を選択すると、STM32CubeIDE がデバッグ中に必要に応じて GDB サーバ を自動的に起動および停止するので、GDB サーバ がシームレスに統合されます。

注 デバッグする必要があるプロジェクトをビルドする際は、コンパイラの最適化レベルとして `-O0` を使用することを推奨します。最適化レベル `-Og` を使用してもデバッグできる可能性がありますが、最適化レベルが高いと、コンパイラがコードを最適化するためデバッグが困難になります。

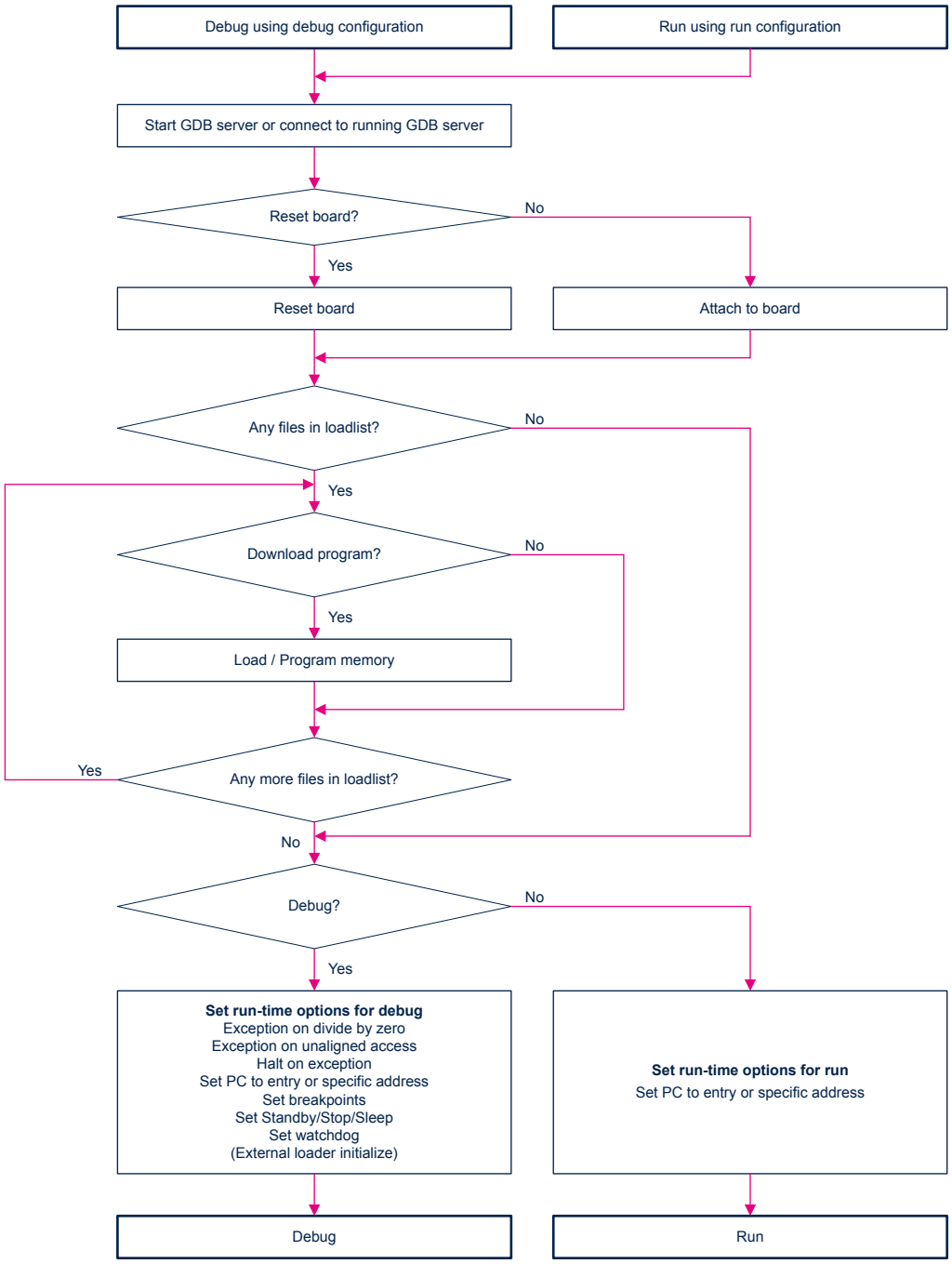
GDB サーバ をアプリケーションのターゲット·システムへのダウンロードだけに使用し、デバッグ·セッションを開始せずにアプリケーションを実行することも可能です。それには、この章で後述する（セクション 3.7 実行用設定 を参照）、実行用の設定を作成します。

STM32CubeIDE を使用して、別の IDE またはツールチェーンで開発した既存の `elf` ファイルをインポートしてデバッグすることも可能です。その場合は、STM32 Cortex®-M の実行可能ファイルのインポートを使用します。これについては、セクション 3.8 STM32 Cortex-M 実行可能ファイルのインポートで説明されています。

3.1.1 一般的なデバッグと実行の起動フロー

STM32 のプログラムのデバッグには、デバッグ設定を使用します。実行用設定は、STM32 に新しいプログラムを書き込んで起動するときに使用します。図 129 のフローチャートに、デバッグ·セッション開始時の GDB サーバ の起動、デバイスのリセット、プログラムのロードから ランタイム オプション、例外、プログラム·カウンタ、ブレークポイント、スタンバイ / 停止 / スリープ、ウォッチドッグの設定、外部ローダ初期化に至る手順を示します。この図から、デバッグと実行の両セッションの違いもわかります。

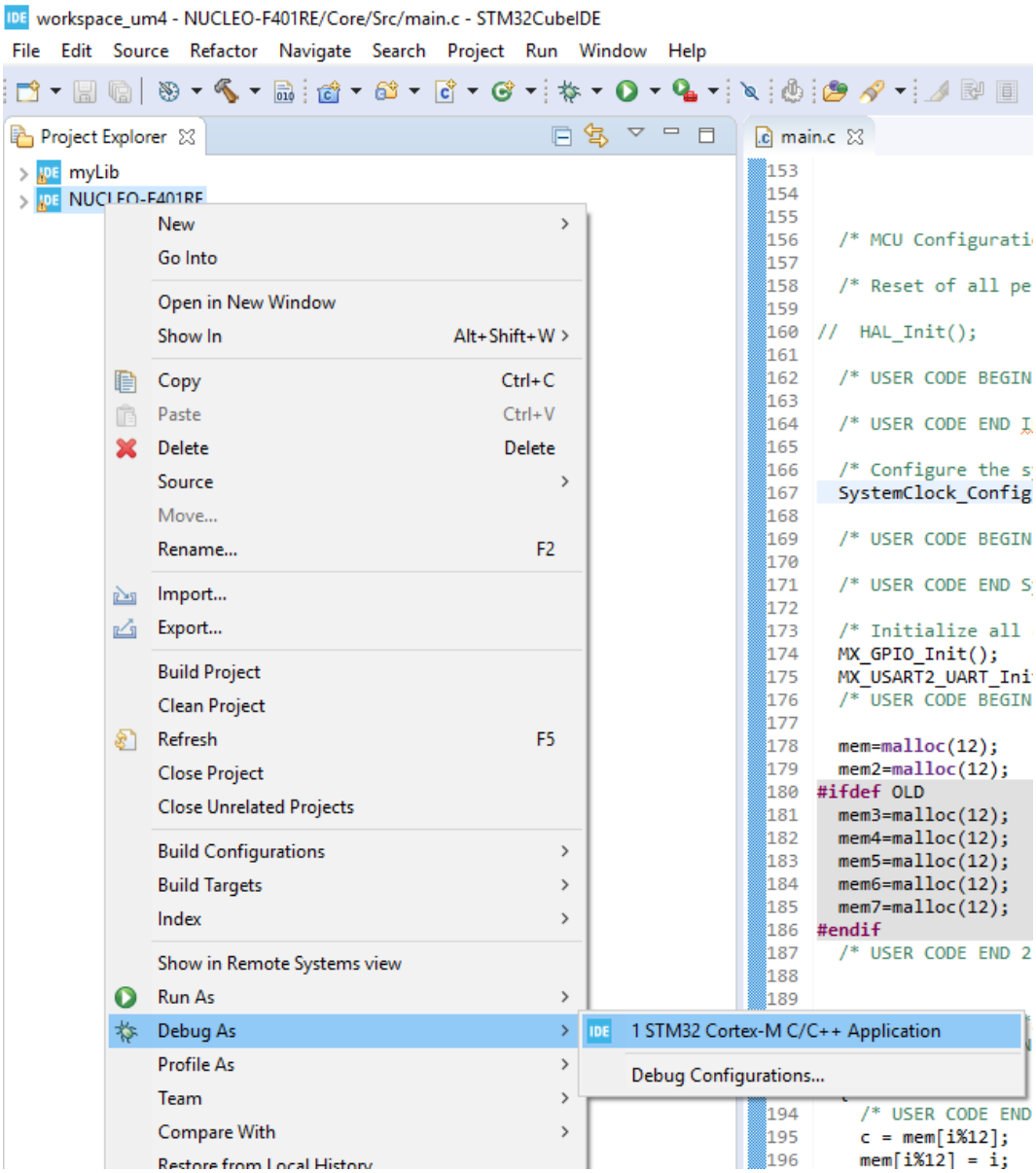
図 129. 一般的なデバッグと実行の起動フロー



3.2 デバッグ設定

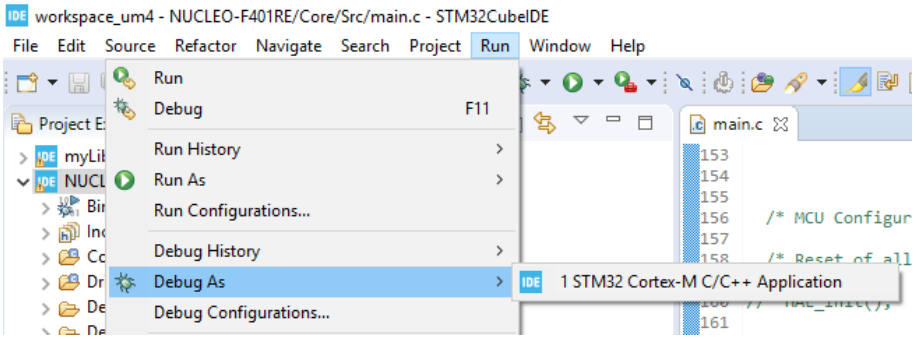
デバッグ·セッションを開始する前に、プロジェクトのデバッグ設定を作成する必要があります。プロジェクトで、はじめてデバッグ設定を作成するには、[Project Explorer]ビューでプロジェクト名を右クリックし、Debug AsSTM32 Cortex-M C/C++ Application を選択します。

図 130. STM32 マイクロコントローラとしてのデバッグ



新しいデバッグ設定を作成する、もう一つの方法は、[Project Explorer]ビューでプロジェクト名を選択し、メニューRunDebug AsSTM32 Cortex-M C/C++ Application を使用します。

図 131. [Debug As]メニューの STM32 マイクロコントローラ



デバッグ設定を新規作成する第三の方法は、[Project Explorer]ビューでプロジェクト名を選択し、F11 キーを押します。
3 種類の方法のいずれを使用しても、[Debug Configuration]ダイアログが開きます。

3.2.1 デバッグ設定

[Debug Configuration]ダイアログには、次のタブがあります。

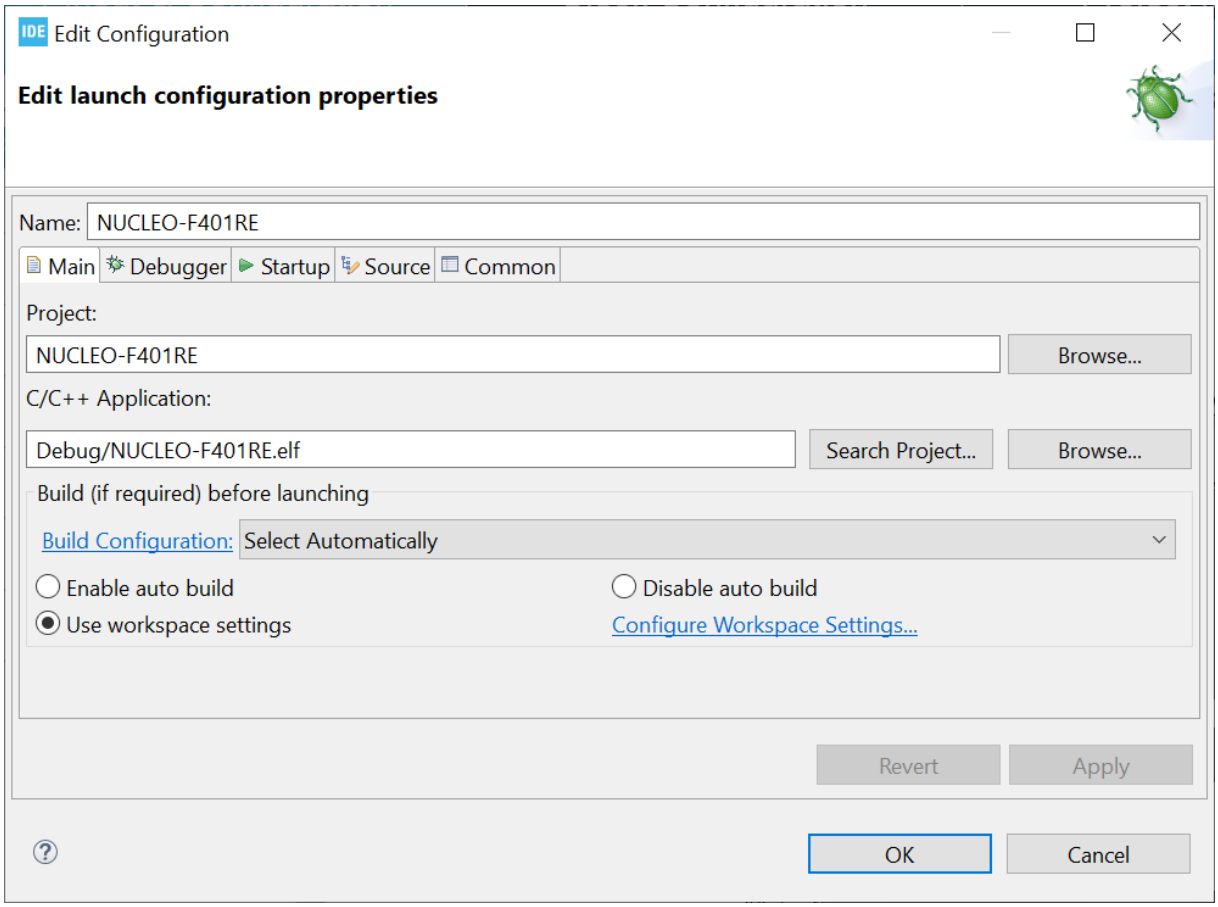
- Main
- Debugger
- Startup
- Source
- Common

[Debugger]と[Startup]の両タブは、デバッグ設定を新規作成したときに更新する必要がありますが、他のタブの更新は不要です。

3.2.2 [Main]タブ

[Main]タブには、デバッグのための C/C++ アプリケーション設定が含まれます。通常、この章の前半で説明した手順でデバッグ設定を作成した場合、[Main]タブの変更は一切必要ありません。正しい `elf` ファイルとプロジェクトが選択されていることを確認してください。

図 132. デバッグ設定 - [Main]タブ

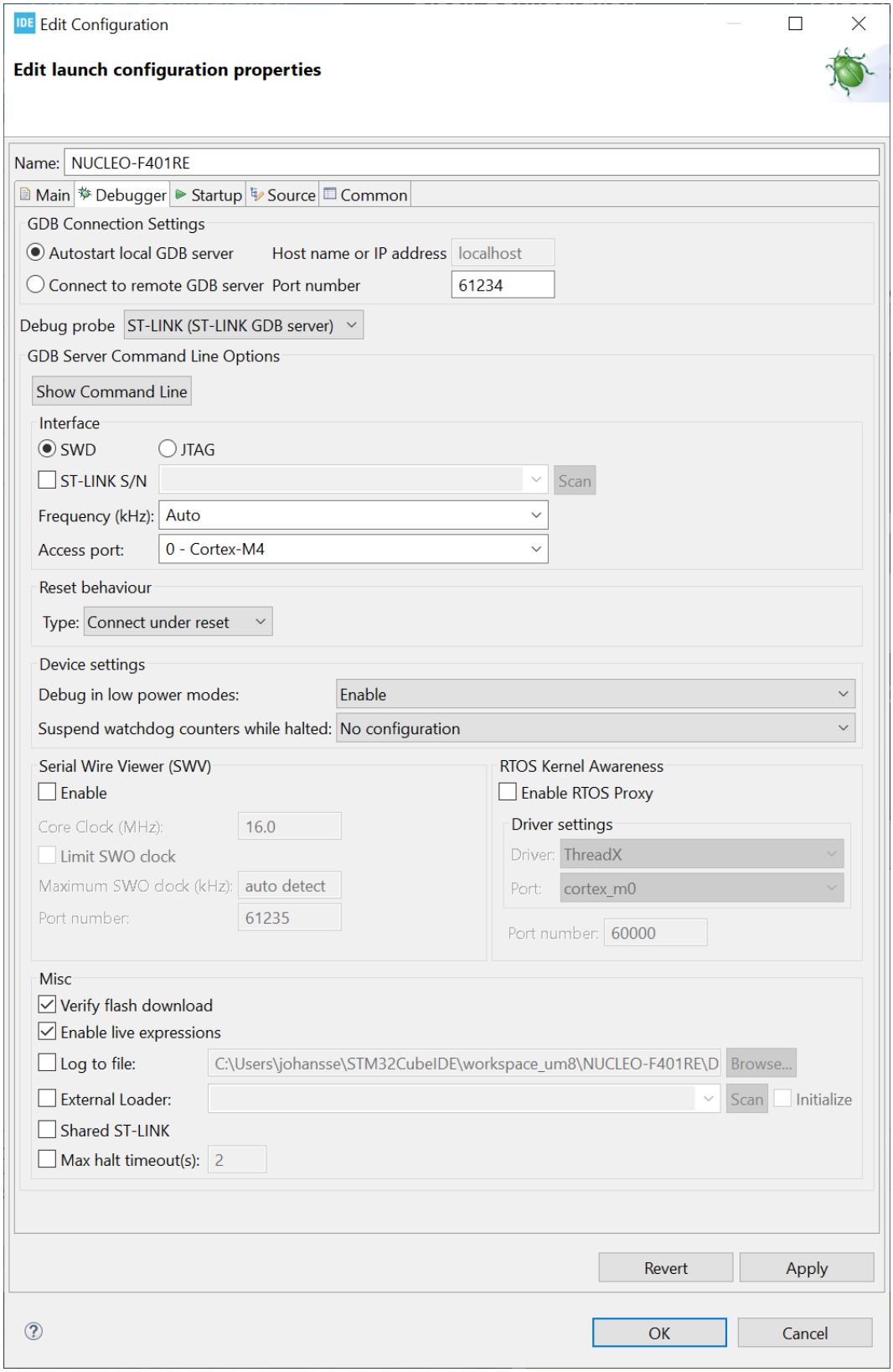


注　デバッグ･セッションの開始前にビルドが必要な場合、[Main]タブでの定義が可能です。

3.2.3 [Debugger]タブ

[Debugger]タブでは、GDB サーバ の起動方法と、サーバへの接続方法を設定します。また、Autostart local GDB server を選択した場合に使用する GDB サーバ も、このタブで定義します。

図 133. デバッグ設定 - [Debugger]タブ



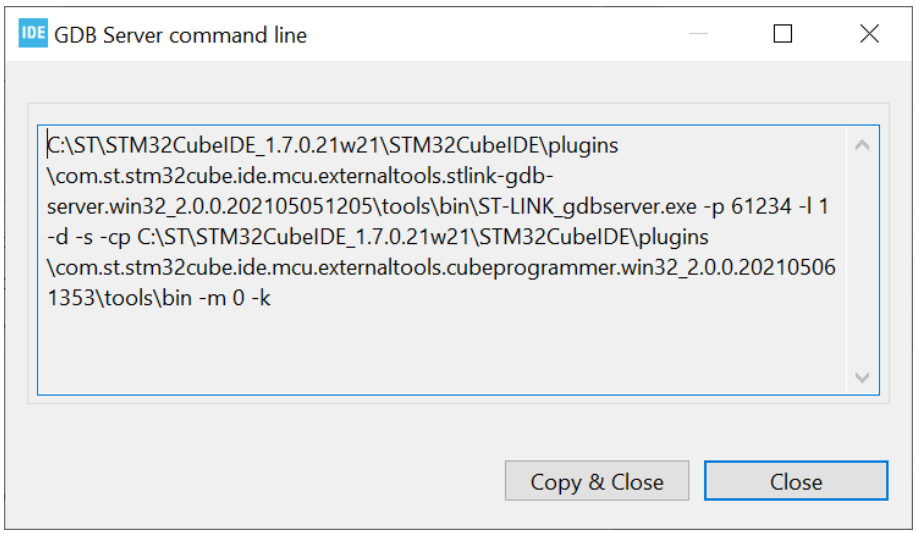
Port number 編集フィールドには、Debug probe フィールドで選択した GDB サーバ が使用するデフォルト値が入力されています。

Connect to remote GDB server を選択した場合、Host name or IP address フィールドを設定する必要があります。

Debug probe フィールドでは、デバッグに使用するプローブと GDB サーバ を選択します。デバッグ・プローブに ST-LINK を使用する場合、ST-LINK GDB サーバ または OpenOCD を使用できます。SEGGER J-LINK プローブを使用する場合は、SEGGER J-LINK GDB サーバ を使用します。

Show Command Line ボタンをクリックすると、[GDB Server command line]ダイアログが開きます。ダイアログには、現在の GDB Server Command Line options の設定に基づいた GDB サーバ の起動方法（コマンド）が表示されます。

図 134. [GDB サーバ コマンドライン]ダイアログ



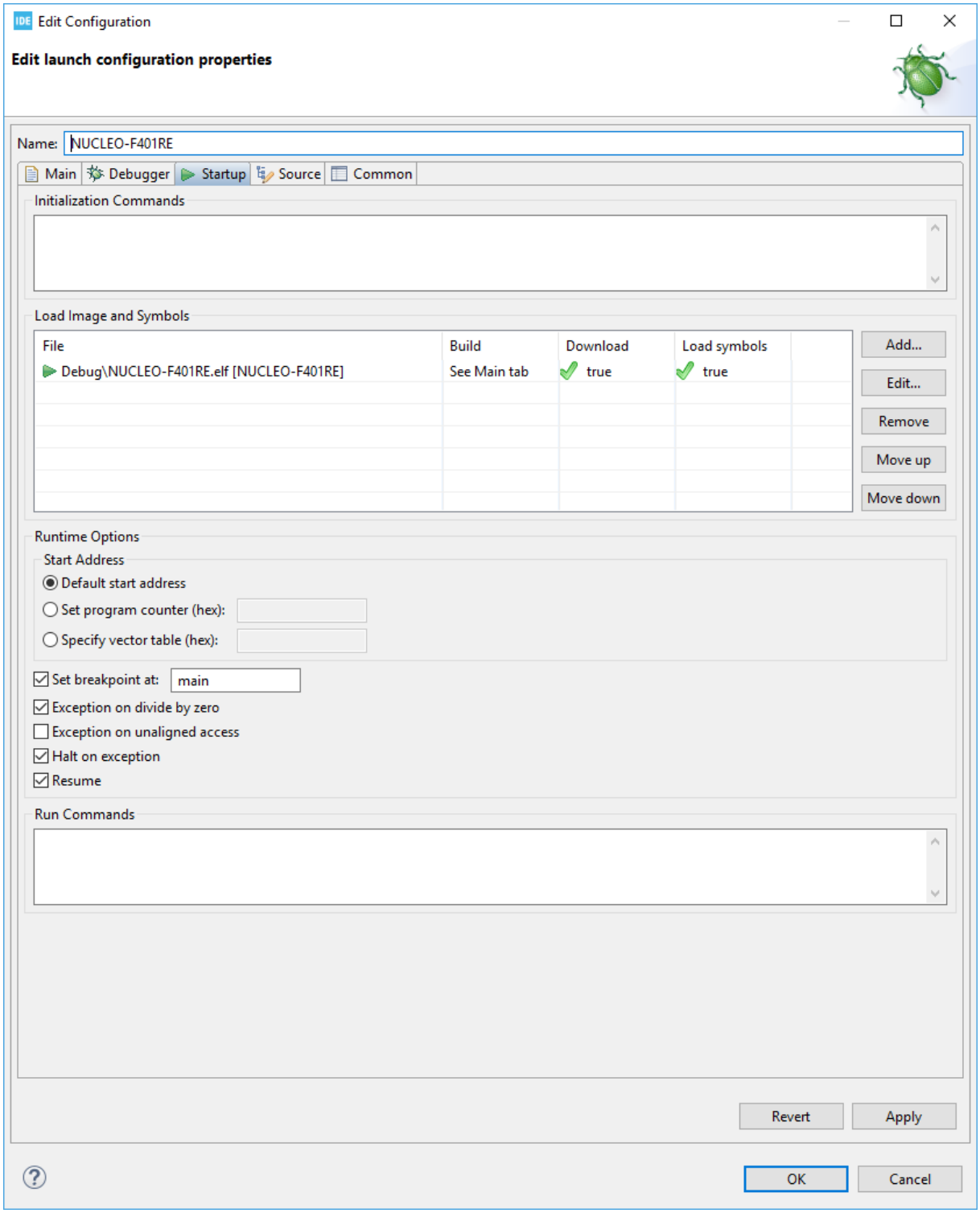
Copy & Close ボタンを使用すると、現在のコマンドライン設定をクリップボードにコピーできます。これをコマンドライン・ウィンドウに貼り付ければ GDB サーバ を手動で起動できます。

GDB Server Command Line options の表示は、Debug probe の選択に応じて更新されます。これらの設定に関する詳細は、セクション 3.4 各種 GDB サーバ によるデバッグ とサブセクションで説明しています。

3.2.4 [Startup]タブ

[Startup]タブでは、デバッグ・セッションの開始方法を設定します。

図 135. デバッグ設定 - [Startup]タブ



Initialization Commands 編集フィールドでは、load コマンドの前に何らかのコマンドを GDB サーバ に送信する特別な必要が生じたときに、GDB または GDB サーバ の monitor コマンドを、種類を問わず指定できます。例えば、ST-LINK GDB サーバ を使用し、ロード前に Flash メモリの消去が必要な場合、ここに `monitor flash mass_erase` コマンドを入力できます。

Load Image and Symbols リスト·ボックスには、デバッグするファイルが登録されている必要があります。このリストに関連して、次のコマンド·ボタンがあります。

- Add…: ダウンロードまたはシンボルをロードするためのファイルを指定する新規行を追加します。

- Edit…: 選択した行を編集します。
- Remove: 選択した行をリストから削除します。
- Move up: 選択した行を上に移動します。
- Move down: 選択した行を下に移動します。

Runtime Options のセクションには、開始アドレスやブレークポイントを設定し、例外処理や再開を有効にするチェックボックスがあります。

開始アドレスについては、次のような選択が可能です。

- Default start address: `$pc` が、最後にロードされた `elf` ファイルで見つかった開始アドレスに設定されます。
- Set program counter (hex): `$pc` が、編集フィールドで指定した 16 進数の値に設定されます。
- Specify vector table (hex): `$pc` が、編集フィールドで指定したアドレスにオフセット 4 を加えた場所に格納されている値に更新されます。この `$pc` の設定方法は、Cortex®-M デバイスでベクタ·テーブルを使用してリセットした場合と同じです。

Set breakpoint at: チェックボックスはデフォルトで有効化され、編集フィールドには `main` と表示されます。これはプログラムのデバッグ時に、デフォルトで `main` にブレークポイントが設定されることを意味します。

例外に関する 3 つのチェックボックス Exception on divide by zero、Exception on unaligned access、Halt on exception を使用すると、アプリケーションのデバッグ時に問題を見つけやすくなります。

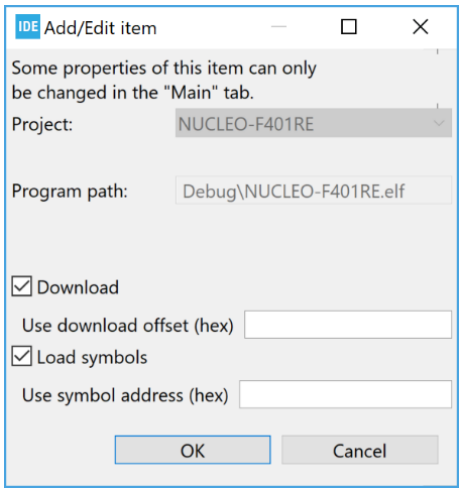
- Exception on divide by zero はデフォルトで有効化され、デバッグ中のゼロによる除算のエラーのトラップを容易にします。
- Exception on unaligned access を有効化すると、アラインされていないアクセスが発生した場合に例外をスローします。
- Halt on exception はデフォルトで有効化され、デバッグ中に例外エラーが発生するとプログラムの実行を停止します。例外が発生した場合、[Fault Analyzer]ビューを使用して問題のある場所を見つけることができます。

注 アプリケーションのデバッグ中だけでなく実行時にもゼロによる除算の例外やアラインされていないアクセスの例外を発行にするには、アプリケーション·ソフトウェアで、これらを有効にする必要があります。CMSIS Cortex®-M のヘッダ·ファイルには、SCB 設定コントロール·レジスタを更新する定義が記述されています。例えば、`core_cm4.h` には SCB->CCR レジスタと、`SCB_CCR_DIV_0_TRP` および `SCB_CCR_UNALIGN_TRP` の定義が含まれます。

Resume を有効に設定すると、プログラム起動のためのロード後に GDB に対して `continue` コマンドが発行されます。通常、その場合、ブレークポイントを main に設定していれば、プログラムは `main` で停止します。それ以外の場合、Resume を無効にすると、プログラムはリンカ·スクリプトで指定した `ENTRY` の位置にとどまります。これは通常 `Reset_Handler` 関数です。その場合、エディタに `Reset_Handler` 関数を表示するステップが必要になるかも知れません。

リストボックスの行を選択して Edit... をクリックすると、ファイルのダウンロードおよびシンボルのロードの要否を選択するための、次のようなダイアログが表示されます。

図 136. 項目の追加 / 編集

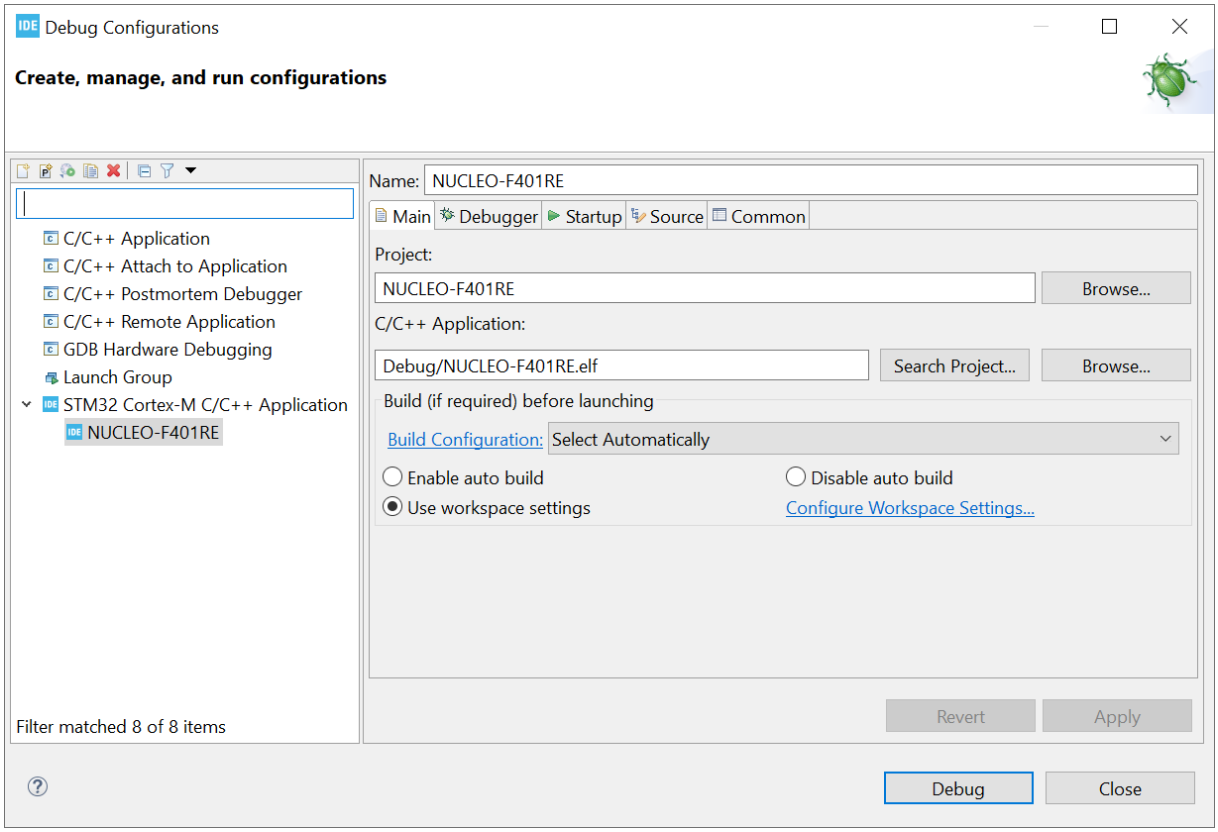


3.3 デバッグ設定の管理

プロジェクトごとに、複数のデバッグ設定を使用できます。既存のデバッグ設定のコピーを作成し、必要な変更を加えるのは簡単です。例えば、ある設定では Flash メモリに新しいプログラムをロードし、別の設定ではプログラムを一切ロードしない場合などがあります。

メニュー RunDebug Configurations… からデバッグ設定を開くと、[Debug Configurations]ダイアログが表示されます。このダイアログの左側にはツールバーを備えたナビゲーション·ウィンドウ、右側には セクション 3.2 デバッグ設定 で説明したタブとフィールドを備えたデバッグ設定があります。

図 137. デバッグ設定の管理



右側ペイン上部の Name フィールドには、設定内容を反映するデバッグ設定の名前を入力できます。Apply をクリックすると、左側のナビゲーション·ウィンドウの STM32 Cortex-M C/C++ Application ノードの下に表示されます。

ナビゲーション·ウィンドウの左側にあるツールバーには、設定を管理するためのアイコンが並び、選択した設定の複製や削除などが可能です。

図 138. デバッグ設定の管理ツールバー

これらのアイコンの目的は、左から右に次のとおりです。

- 起動設定の新規作成
- 設定プロトタイプの新規起動
- 起動設定のエクスポート
- 現在選択中の起動設定の複製
- 選択中の起動設定の削除
- 展開した起動設定すべての折りたたみ
- 起動設定のフィルタ

3.4 各種 GDB サーバ によるデバッグ

STM32CubeIDE には、次の GDB サーバ が付属しています。

- ST-LINK GDB サーバ
- OpenOCD GDB サーバ
- SEGGER J-Link GDB サーバ

いずれの GDB サーバ も、標準的なデバッグ、ライブ式、SWV に対応しています。

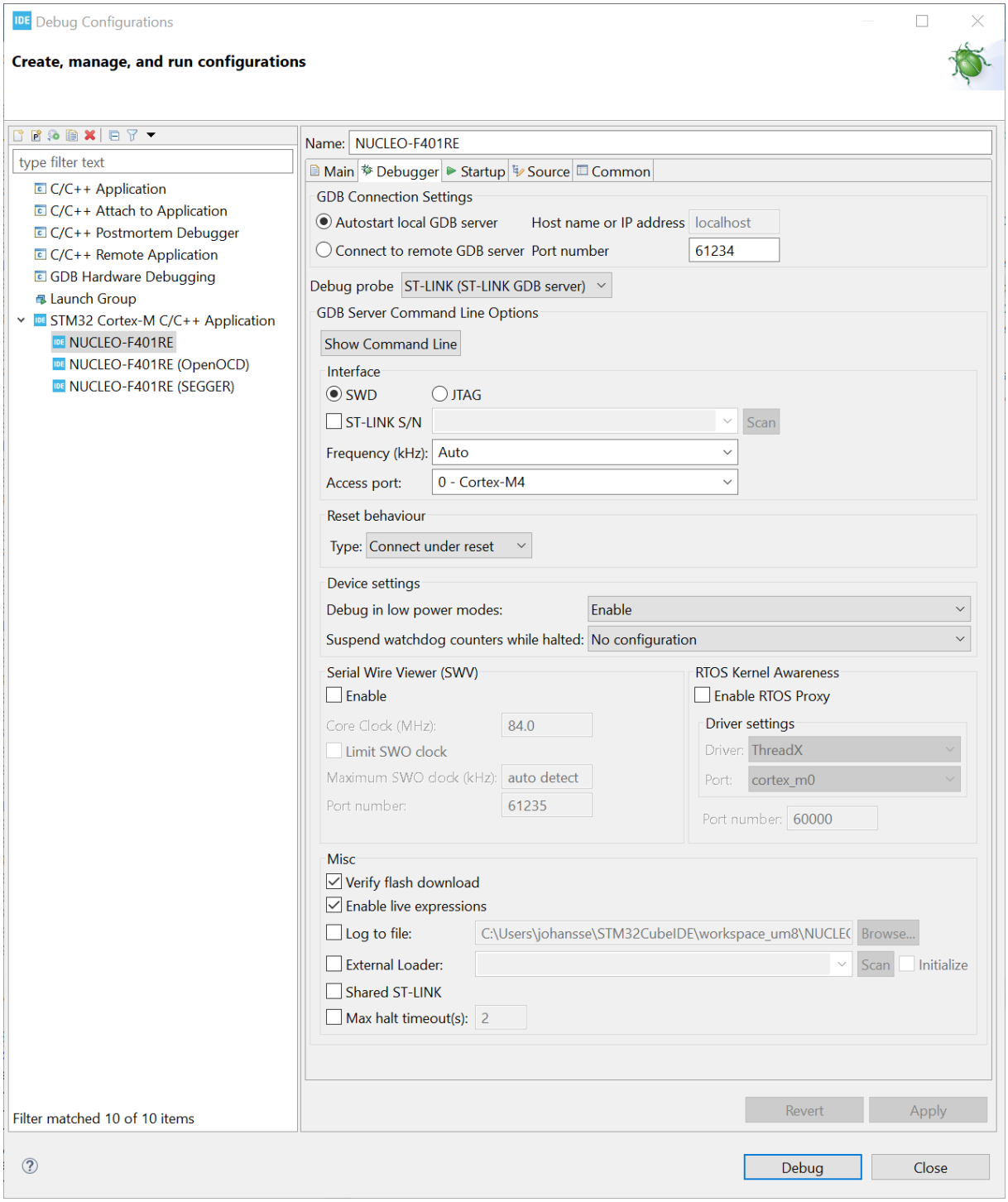
すべての GDB サーバ が、RTOS プロキシを使用する Microsoft 社® Azure® RTOS ThreadX および FreeRTOS™ オペレーティング·システムでの RTOS カーネル認識デバッグにも対応しています。STM32CubeIDE には RTOS プロキシが付属しています。

これらの GDB サーバ を起動する際は、さまざまなコマンドライン·オプションが使用されます。このため、[Debug Configurations]ダイアログの[Debugger]タブには、選択した GDB サーバ サーバに応じて、各種の設定が表示されます。このセクションでは、各サーバ固有の設定について説明します。

3.4.1 ST-LINK GDB サーバ によるデバッグ

通常、デバッグに ST-LINK GDB サーバ を使用する場合、[Debugger]タブの GDB Server Command Line Options を変更する必要はありません。しかし、デフォルト設定の変更が必要になる場合もあります。SWV の使用が予定されている場合や、PC に複数の STM32 ボードが接続される場合などです。

図 139. ST-LINK GDB サーバ の[Debugger]タブ



Interface の SWD または JTAG を選択して、マイクロコントローラに、どのように ST-LINK プローブを接続する必要があるのかを定義します。通常は、SWD インタフェースを選択することを推奨します。SWV を使用する場合は、このオプションを選択する必要があります。

ST-LINK S/N を有効にする場合、編集 / リスト･フィールドに、使用する ST-LINK プローブのシリアル番号を入力する必要があります。Scan ボタンを使用すると、PC に接続されたデバイスをスキャンし、検出された ST-LINK デバイスをすべて一覧表示することができます。スキャン後、これらの ST-LINK デバイスのシリアル番号がリスト･ボックスに表示されるので、そこから必要な ST-LINK を選択できます。Use specific ST-LINK S/N が有効化されている場合、ST-LINK GDB サーバ は起動後、選択されたシリアル番号の ST-LINK にのみ接続します。

Frequency (kHz) のオプションは、ST-LINK と STM32 デバイス間の通信速度を定義します。Auto を選択した場合、ST-LINK が提供する最大速度が適用されます。ハードウェアに制約がある場合は、この周波数を下げてください。

Access port の選択は、マルチコアの STM32 デバイスをデバッグする場合にのみ使用します。この場合、ST-LINK がデバイスに接続され、ST-LINK GDB サーバ にデバッグ対象となるコアを知らせる必要があります。

Reset behaviour には、Type および Halt all cores の選択があります。Halt all cores のオプションが表示されるのは、マルチコア・デバイスの場合のみです。

Type は次のように設定します。

- Connect under reset（デフォルト）: ST-LINK のリセット・ラインがアクティブになり、リセットがアクティブな間は、ST-LINK が SWD または JTAG モードで接続します。その後、リセット・ラインが非アクティブになります。
- Software system reset: レジスタへのソフトウェア書込みによってシステム・リセットがアクティブになります。この方法は、コアとペリフェラルをリセットし、システム全体をリセットできます。ターゲットのリセット・ピンが自動的にアサートされるためです。
- Hardware reset: ST-LINK のリセット・ラインがアクティブ化された後、非アクティブ化されてから（リセット・ラインのパルス駆動）、ST-LINK が SWD または JTAG モードで接続します。
- Core reset: コアのリセットがレジスタへのソフトウェア書込みによってアクティブ化されます（Cortex®-M0、Cortex®-M0+、Cortex®-M33 コアでは使用できません）。この方法では、コアのみがリセットされ、ペリフェラルとリセット・ピンはいずれもリセットされません。
- None: プログラムが既にデバイスにダウンロードされた、動作中のターゲットへのアタッチに使用します。[Startup]タブには、ファイルの書込みコマンドを一切設定できません。

注 デバッグ設定に Flash メモリへの書込みが含まれている場合、ここで選択したリセット動作はオーバーライドされ、ST-LINK GDB サーバ は STM32CubeProgrammer（STM32CubeProg）のコマンドライン・プログラム `STM32_Programmer_CLI` を使用して、Flash メモリへの書込みを実行します。このプログラムは常に `mode=UR reset=hwRst` の設定で ST-LINK GDB サーバ によって開始されるため、新しいプログラムをロードすると None オプションの選択を無視してデバイスはリセットされます。これによって、デバイスへの適切な書込みが保証されます。

Halt all cores は、マルチコア・デバイスのデバッグにのみ使用できるオプションです。Halt all cores のオプションはシングルコア・デバイスでは表示されません。

Device settings では Debug in low power modes（低電力モードによるデバッグ）と Suspend watchdog counters while halted（停止中はウォッチドッグ・カウンタを一時停止）を選択できます。これらは、次のいずれかに定義できます。

- No configuration
- Enable
- Disable

Serial Wire Viewer (SWV) のオプションは、SWD インタフェースを選択した場合にのみ使用できます。SWV を有効にする場合は、Clock Settings の設定が必要になります。Core Clock をデバイスの速度に設定してください。SWV の設定の詳細は、セクション 4.2.1 SWV デバッグ設定 で説明しています。

RTOS Kernel Awareness のオプションは、ThreadX および FreeRTOS™ オペレーティング・システムで RTOS カーネル認識デバッグを有効にする場合に使用します。RTOS カーネル認識デバッグを有効化してデバッグ・セッションを開始すると、[Debug]ビューにすべてのスレッドが一覧表示されます。[Debug]ビューのスレッドを選択することで、そのスレッドで現在実行されている行がエディタに表示されます。RTOS カーネル認識デバッグの詳細は、セクション 6.3 で説明しています。

Misc には、次のオプションがあります。

- Verify flash download（Flash メモリへのダウンロードを検証します）
- Enable live expressions（デバッグ中に[Live Expressions]ビューを使用できるようにするオプションです。スタートアップ時にライブ式の機能を有効にしておく必要があります。このオプションは、デフォルトで有効化されます。）
- Log to file（デバッグの問題が発生した場合に有効化します。ST-LINK GDB サーバ をより高いログ・レベルで起動し、ログをファイルに保存します。）
- External Loader（非内蔵の STM32 Flash メモリへのロードが必要な場合に有効化します）。STM32CubeProgrammer の外部 Flash ローダ・ファイルにアクセスするために Scan ボタンが用意されています。

  External Loader を有効にした場合、Initialize のオプションも表示されます。これを有効にすると、リセット後に STM32CubeProgrammer 内で `Init()` 関数が呼び出されます。外部メモリ・アクセスのためのデバイスの設定に使用できます。通常、初期化はデバッグ済みのアプリケーションで行う必要があります。
- Shared ST-LINK（デバッグ・セッションにおいて他のプログラムも同じ ST-LINK に接続できるようにするには、この共有 ST-LINK のオプションを有効にする必要があります）。詳細については、セクション 3.6.2 ST-LINK 共有を参照してください。

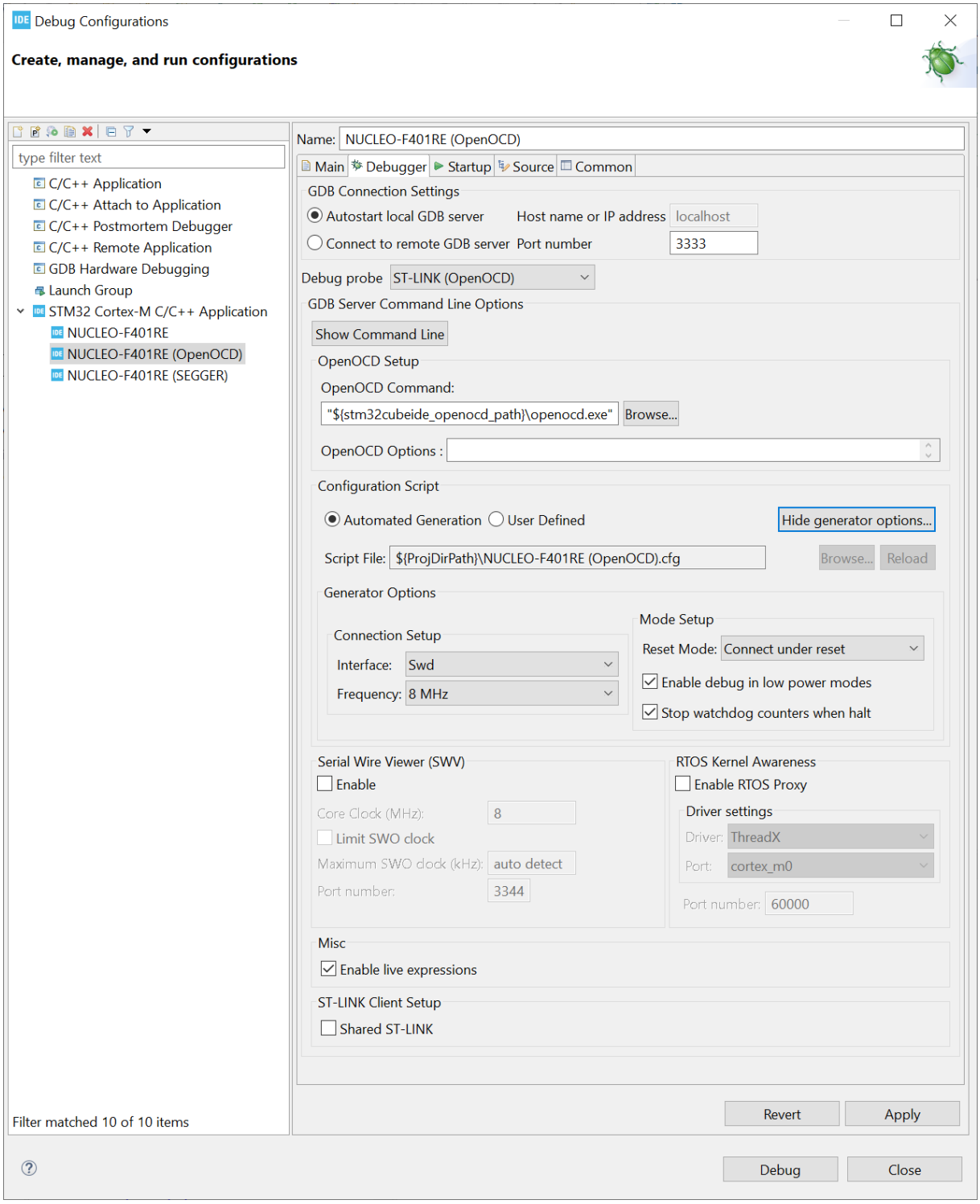
  ST-LINK GDB サーバ の詳細な説明は、ST-LINK GDB サーバ のマニュアル（[ST-07]）に記載されています。このドキュメントは Information Center から入手できます。

注 `STM32_Programmer_CLI` は STM32 または外部の Flash メモリに書き込むために ST-LINK GDB サーバ によって使用されます。そのような外部 Flash メモリへの書込みは、外部ローダを使用して自動的に行われます。

3.4.2 OpenOCD および ST-LINK によるデバッグ

OpenOCD を使用する場合、[Debugger]タブの GDB Server Command Line Options にジェネレータの表示 / 非表示を切り換える Show generator options... フィールドと Hide generator options... が表示されます。このフィールドを Hide generator options... に設定すると、図 140 に示すように追加の GDB Server Command Line Options が表示されます。

図 140. OpenOCD の[Debugger]タブ



OpenOCD Command 編集フィールドには、デバッグ時に使用する `openocd.exe` ファイルが表示されます。Browse ボタンを使用すれば、別のバージョンの OpenOCD を選択できます。

OpenOCD Options 編集フィールドは、OpenOCD の起動時に付加するコマンドライン・パラメータの指定に使用できます。

Configuration Script の設定には、Automated Generation または User Defined のオプションがあります。Automated Generation を選択した場合、[Debugger]タブの選択に基づいて `openocd.cfg` ファイルが自動的に作成されます。User Defined を選択した場合は、Script File 編集フィールドでファイルを指定する必要があります。

Interface のオプション Swd または Jtag によって、マイクロコントローラへの ST-LINK プローブの接続方法を選択します。通常は、Swd を選択することを推奨します。

[Frequency]の選択は、ST-LINK と STM32 デバイス間の通信速度を設定します。

Reset Mode には、次のようなオプションがあります。

- Connect under reset(デフォルト) : ST-LINK のリセット·ラインがアクティブになり、リセットがアクティブな間は、ST-LINK が SWD または JTAG モードで接続します。その後、リセット·ラインが非アクティブになります。
- Hardware reset: ST-LINK のリセット·ラインがアクティブ化された後、非アクティブ化されてから(リセット·ラインのパルス駆動)、ST-LINK が SWD または JTAG モードで接続します。
- Software system reset: レジスタへのソフトウェア書込みによってシステム·リセットがアクティブになります。この方法は、コアとペリフェラルをリセットし、システム全体をリセットできます。ターゲットのリセット·ピンが自動的にアサートされるためです。
- Core reset: コアのリセットがレジスタへのソフトウェア書込みによってアクティブ化されます(Cortex®-M0、Cortex®-M0+、Cortex®-M33 コアでは使用できません)。この方法では、コアのみがリセットされ、ペリフェラルとリセット·ピンはいずれもリセットされません。
- None: プログラムが既にデバイスにダウンロードされた、動作中のターゲットへのアタッチに使用します。[Startup]タブには、ファイルの書込みコマンドを一切設定できません。

Enable debug in low power modes は、STM32 デバイスの低電力モードによるデバッグを有効にします。

Stop watchdog counters when halt を有効にすると、デバッグ·セッションで STM32 デバイスを停止した場合にウォッチドッグが停止します。それ以外の場合、ウォッチドッグ割込みがトリガされる可能性があります。

Serial Wire Viewer (SWV) のオプションは、SWD インタフェースを選択した場合にのみ使用できます。SWV を有効にする場合は、Clock Settings の設定が必要になります。Core Clock をデバイスの速度に設定してください。SWV の設定の詳細は、セクション 4.2.1 SWV デバッグ設定 で説明しています。

RTOS Kernel Awareness のオプションは、ThreadX および FreeRTOS™ オペレーティング·システムで RTOS カーネル認識デバッグを有効にする場合に使用します。RTOS カーネル認識デバッグを有効化してデバッグ·セッションを開始すると、[Debug]ビューにすべてのスレッドが一覧表示されます。[Debug]ビューのスレッドを選択することで、そのスレッドで現在実行されている行がエディタに表示されます。RTOS カーネル認識デバッグの詳細は、セクション 6.3 で説明しています。

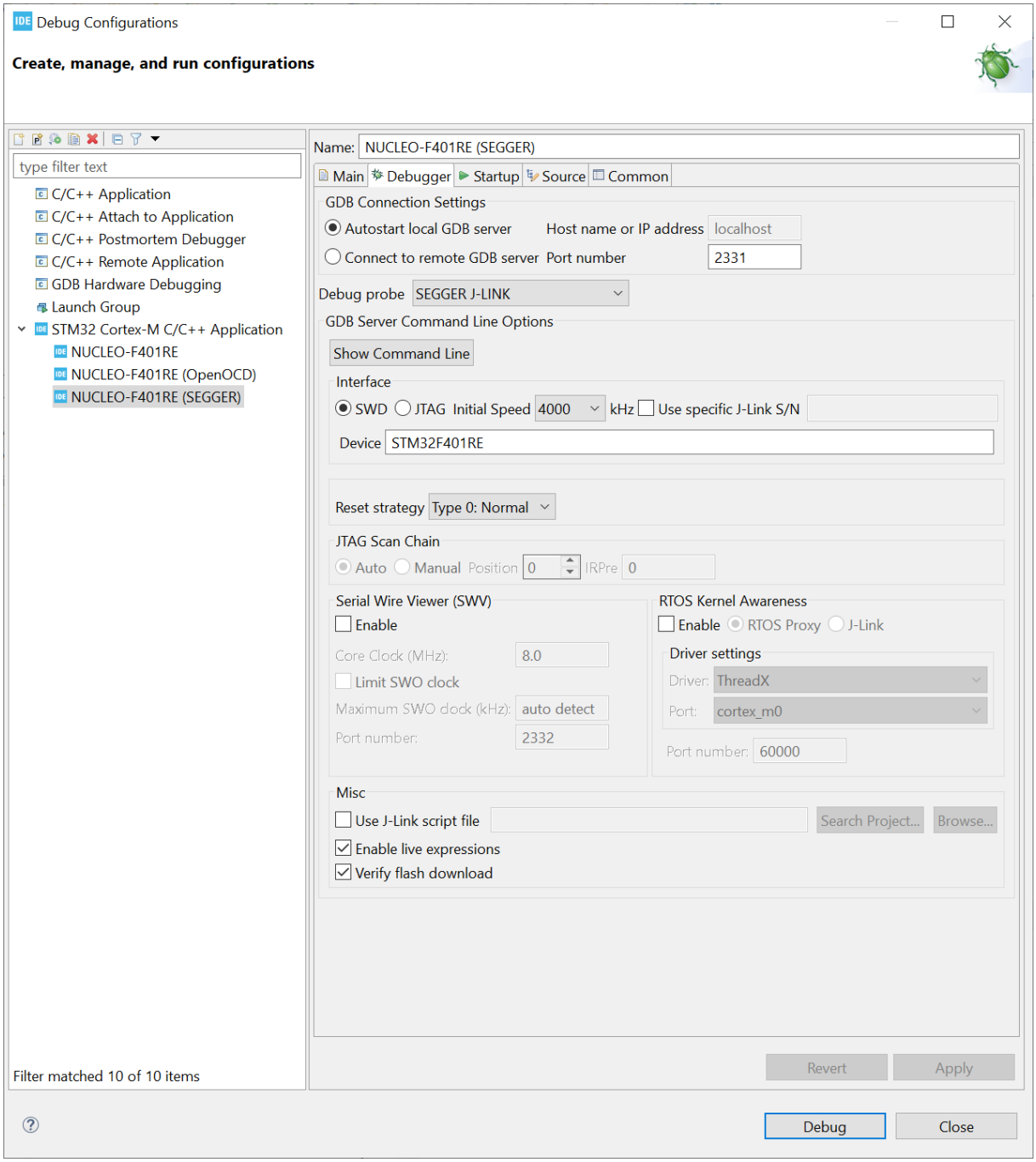
デバッグ中に[Live Expressions]ビューを使用する予定がある場合、Enable live expressions を有効にする必要があります。

デバッグ·セッションにおいて他のプログラムも同じ ST-LINK に接続する必要がある場合、Shared ST-LINK を有効にしてください。詳細については、セクション 3.6.2 ST-LINK 共有を参照してください。

3.4.3 SEGGER J-Link によるデバッグ

[Debugger]タブで SEGGER J-LINK を選択した場合、GDB Server Command Line Options は、SEGGER J-Link GDB server に対応したものになります。

図 141. SEGGER 使用時の[Debugger]タブ



Interface のオプション SWD または JTAG によって、マイクロコントローラへの SEGGER J-Link プローブの接続方法を選択します。通常、SWD インタフェースが推奨されますが、SWV を使用する場合は必須です。

Initial Speed の選択は、SEGGER J-Link と STM32 デバイス間の通信速度を設定します。

Use specific J-Link S/N を有効にする場合、編集 / リスト·フィールドに、デバッグ時に使用する J-Link のシリアル番号を入力します。Use specific ST-LINK S/N が有効化されている場合、SEGGER J-Link GDB サーバ は起動後、選択されたシリアル番号の J-Link にのみ接続します。

[Device]編集フィールドは、値を入力したときだけ使用されます。このフィールドは、STM32CubeIDE で使われているデフォルトのデバイス名では、SEGGER J-Link GDB サーバ の起動に問題が生じる場合に使用できます。そのような場合に、この編集フィールドに SEGGER GDB サーバ が使用するデバイス名を入力します。

Reset strategy には、次のようなオプションがあります。

- Type 0: Normal - デフォルトです。

- None - 実行中のターゲットへのアタッチに使用します。その場合、プログラムが既にデバイスにダウンロードされている必要があります。[Startup]タブには、ファイルの書込みコマンドを一切設定できません。

JTAG Scan Chain のオプションは、JTAG インタフェースを選択した場合にのみ使用できます。

Serial Wire Viewer (SWV) のオプションは、SWD インタフェースを選択した場合にのみ使用できます。SWV を有効にする場合は、Clock Settings の設定が必要になります。Core Clock をデバイスの速度に設定してください。SWV の設定の詳細は、セクション 4.2.1 SWV デバッグ設定 で説明しています。

RTOS Kernel Awareness のオプションは、ThreadX および FreeRTOS™ オペレーティング·システムで RTOS カーネル認識デバッグを有効にする場合に使用します。RTOS カーネル認識デバッグを有効化してデバッグ·セッションを開始すると、[Debug]ビューにすべてのスレッドが一覧表示されます。[Debug]ビューのスレッドを選択することで、そのスレッドで現在実行されている行がエディタに表示されます。RTOS カーネル認識デバッグの詳細は、セクション 6.3 で説明しています。

Misc には、次のオプションがあります。

- Use J-Link script file（J-Link スクリプト·ファイルを使用します）
- Enable live expressions（ライブ式を有効にします）

  デバッグ中に[Live Expressions]ビューを使用できるようにするには、スタートアップ時にライブ式の機能を有効にしておく必要があります。
- Verify flash download（Flash メモリへのダウンロードを検証します）
- FreeRTOS や embOS で Thread-aware RTOS support を使用する場合、Select RTOS variant リスト·ボックスを使用できます。

  Thread-aware RTOS support を使用する場合、[Startup]タブを次のように変更してください。Resume を無効化し、Run Commands に `thread 2` と `continue` コマンドを追加します。これによって、`continue` コマンドが送信される前に、スレッドのコンテキストが強制的に切り換えられます。

注 SEGGER J-Link GDB サーバ の詳細は、SEGGER J-Link のマニュアルに記載されています。このドキュメントには、Information Center からアクセスできます。

3.5 デバッグの開始と停止

お好みの JTAG プローブに合わせてプロジェクトのデバッグ設定を作成したら、デバッグの準備は完了です。この後のセクションでは ST-LINK GDB サーバ を使用します。しかし、STM32 のプロジェクトのデバッグ方法は、ST-LINK GDB サーバ、OpenOCD、SEGGER J-Link のいずれを選択したかにかかわらず同じです。

デバッグに備えて、次の手順を実行してください。

1. ボードが JTAG によるデバッグ、SWD によるデバッグ、またはこれら両方に対応しているかどうかを判断します。

   通常は、SWD モードのデバッグを選択することを推奨します。
2. JTAG プローブとターゲット·ボードを JTAG ケーブルで接続します。

   ST マイクロエレクトロニクスの STM32 Nucleo と ディスカバリ·ボード を使用する場合、ST-LINK は通常ボードに内蔵されています。また、ST マイクロエレクトロニクスのほとんどの STM32 評価ボード にも ST-LINK が組み込まれています。
3. PC と JTAG プローブを USB ケーブルで接続します。
4. ターゲット·ボードが適切な電源に接続されていることを確認します。

上記の手順が完了したら、デバッグ·セッションを開始できます。

3.5.1 デバッグの開始

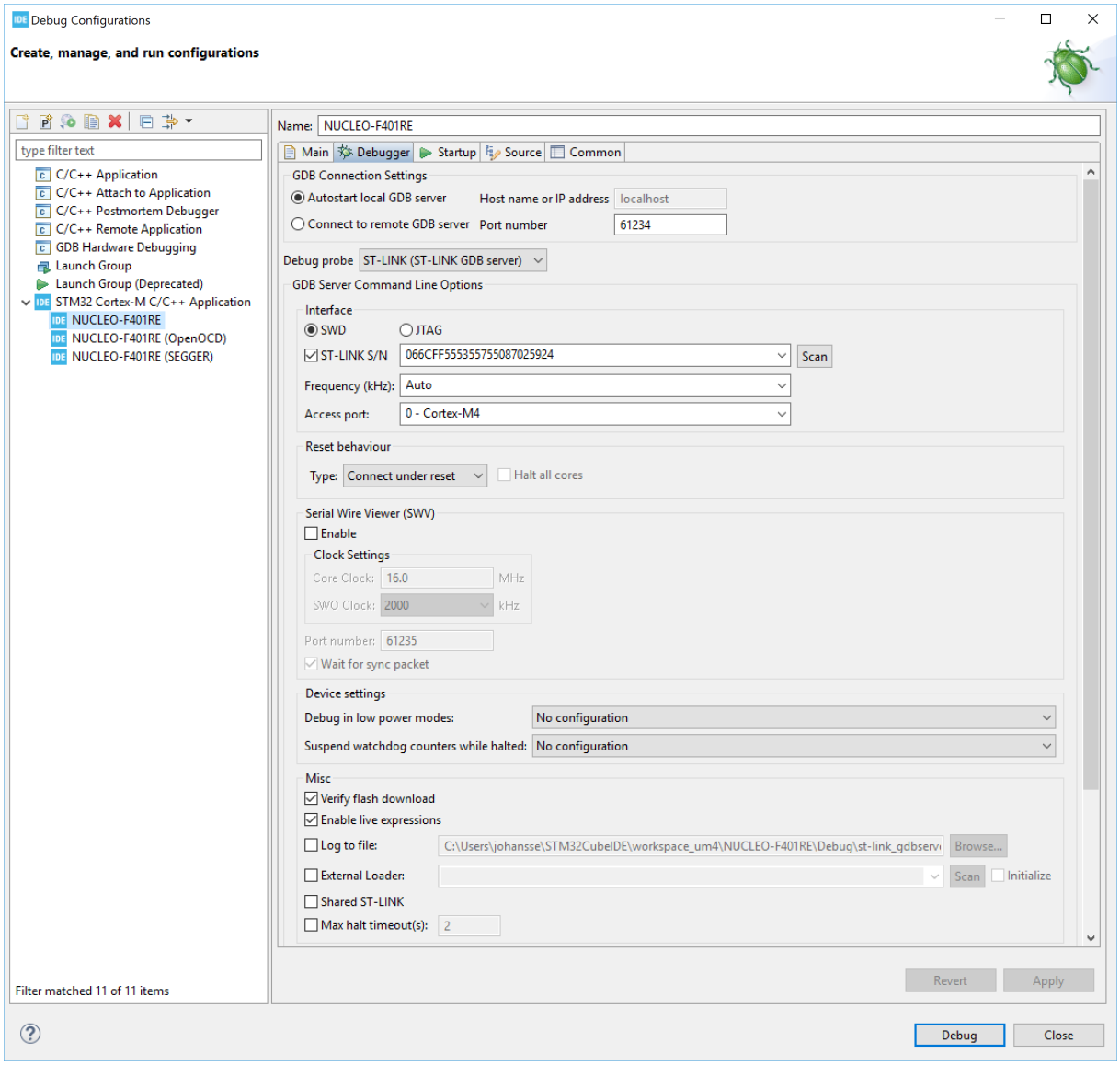
[Project Explorer]ビューのプロジェクト名を右クリックして、Debug As Debug Configurations… を選択することで[Debug Configurations]ダイアログを開きます。

このダイアログは、メニュー Run Debug Configurations... を使用しても開くことができます。

これによって[Debug Configurations]ダイアログが表示されます。

注 [Project Explorer]ビューでプロジェクトを選択し、F11 キーを押すと、閉じた後のデバッグ·セッションを再起動できます。

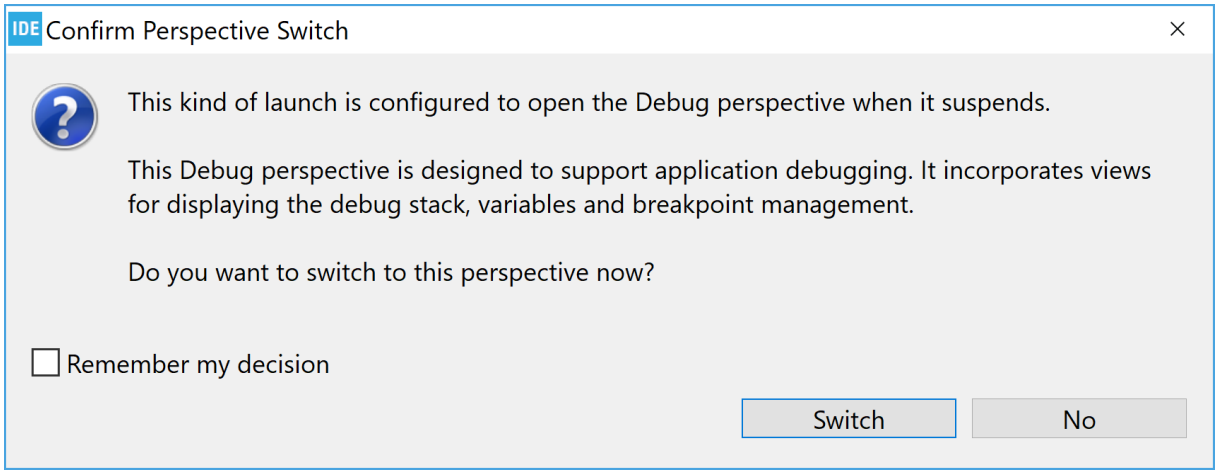
図 142. デバッグ設定



左側ペインで、使用するデバッグ設定を選択しますすべてのデバッグ設定が完了したら、Debug ボタンをクリックすることでデバッグ・セッションを開始します。ファイルを更新していた場合は、プロジェクトがビルドされますが、ビルド動作はデバッグ設定によって異なります。

STM32CubeIDE がデバッガを起動し、次のダイアログが開きます。

図 143. パースペクティブ切り換えの確認

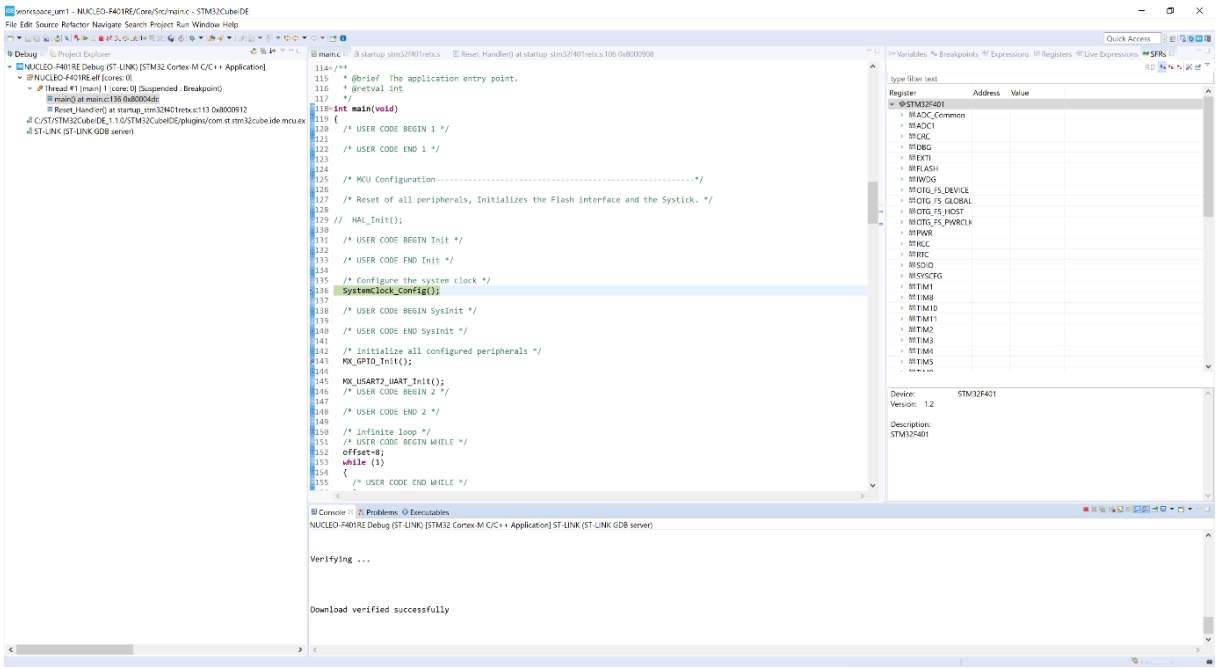


Remember my decision を有効にしたうえで Switch をクリックすることを推奨します。デバッグ・パースペクティブに切り換わります。このパースペクティブにはデバッグに適した多数のビューやウィンドウが表示されます。

3.5.2 デバッグのパースペクティブとビュー

デバッグ・パースペクティブには、デバッグ中に頻繁に使用されるメニュー、ツールバー、ビューが含まれます。

図 144. デバッグ・パースペクティブ



デバッグ・パースペクティブにデフォルトで表示される、最も重要なビューは、次のとおりです。

- [Debug]ビューには、デバッグ対象のプログラムが表示されます。またスレッドのリストも表示され、各行を選択することでスレッド間を移動できます。
- [Editor]ビューには、プログラム・ファイルが表示されます。ここでは、ファイル内にブレークポイントを設定したり、プログラムの実行状況を追ったりすることができます。また、変数の上にマウス・ポインタを移動すると、現在の値が表示されます。関数その他の宣言を開くなど、ファイルの編集中に使用できる機能は、デバッグ中にも使用できます。
- [Variables]ビューには、プログラムが動作していない間、自動的にローカル変数がその現在値とともに表示されます。

- [Breakpoints]ビューには、現在のブレークポイントが表示されます。リスト内のブレークポイントの有効 / 無効を切り換えることができます。[Breakpoints]ビューにはツールバーもあり、ブレークポイントを削除したり、Skip All Breakpoints アイコンを 1 回クリックするだけで、ブレークポイントをスキップしたりすることが可能です。
- [Expressions]ビューは、式の追加や表示に使用します。式としては、単一のグローバル変数、構造体、または何らかの変数の計算式を使用できます。値が更新されるのは、プログラムが停止したときだけです。変数名を入力する代わりに、[Editor]ビューでグローバル変数を選択し、[Expressions]ビューまでドラッグすることも可能です。
- [Registers]ビューには、デバッグ対象のデバイスの現在の値が表示されます。値が更新されるのは、プログラムが停止したときだけです。
- [Live Expressions]ビューには、サンプリングされた式の値が、プログラム実行中に定期的に更新されながら表示されます。`(Index*4+Offset)` のように自動的に評価される数式を作成することも可能です。[Live Expressions]ビューを更新するには、デバッグ設定でライブ式を有効化しておく必要があります。詳細については、セクション 3.6.1 [Live Expressions]ビューを参照してください。
- [SFRs]ビューには、デバッグ対象のデバイスの特殊機能レジスタ(SFR)が表示されます。詳細については、セクション 5 特殊機能レジスタ(SFR)を参照してください。
- [Console]ビューは、各種コンソールの出力です。デフォルトでは、GDB サーバ ログのコンソール出力が表示されます。[Console]ビューの右にある Display Selected Console アイコンをクリックすると、コンソール·ログを変更できます。

デバッグに役立つ、次のようなビューもあります。

- [Debugger Console]ビューは、GDB コマンドを手動で入力する必要が生じたときに使用します。[Debugger Console]ビューを開く最も簡単な方法は、クイック·アクセス·フィールドに `Debugger` と入力します。これによって、[Debugger Console]ビューを含む選択肢が表示されます。これを選択してビューを開きます。[Debugger Console]ビューには、GDB を入力できます。

  例えば、アドレス `0x800 0000` から 16 ワード分のメモリ内容を表示するには、GDB コマンドの `x /16 0x8000000` を入力します。

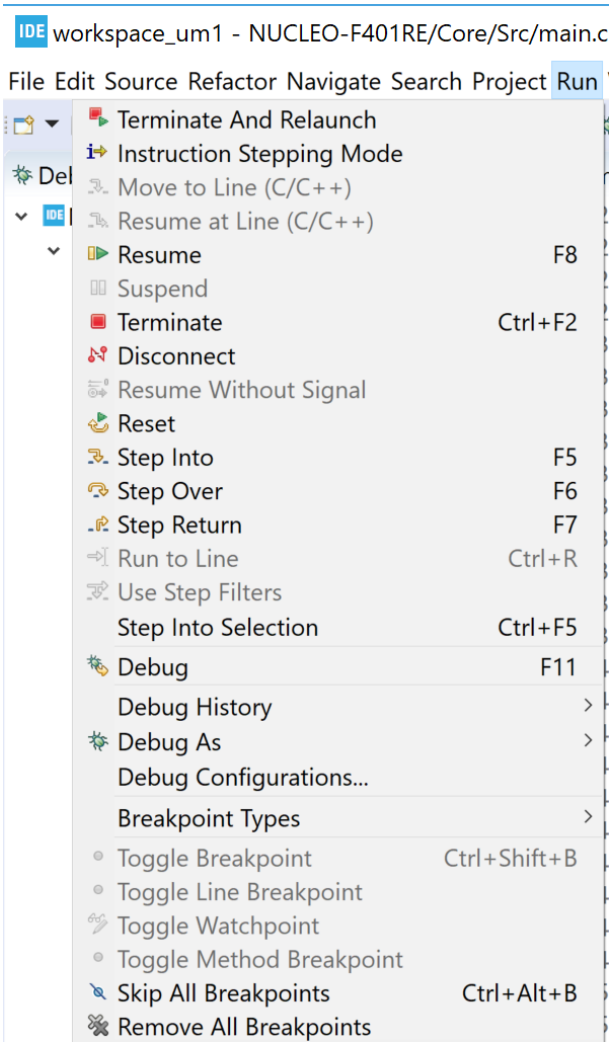
```
x /16 0x8000000
0x8000000:    0x20018000    0x080008b1    0x080007e9    0x080007f7
0x8000010:    0x080007fd    0x08000803    0x08000809    0x00000000
0x8000020:    0x00000000    0x00000000    0x00000000    0x0800080f
0x8000030:    0x0800081d    0x00000000    0x0800082b    0x08000839
```

- [Memory]および[Memory Browser]ビューは、メモリ·データの表示、更新に使用できます。
- [Disassembly]ビューは、アセンブリ·コードの表示やステップ動作に使用します。
- [SWV]ビュー。詳細については、セクション 4 シリアル·ワイヤ·ビューア(SWV)トレースを使用したデバッグを参照してください。
- [Fault Analyzer]ビュー。詳細については、セクション 7 Fault Analyzer を参照してください。

3.5.3 デバッグのメイン制御

デバッグ·パースペクティブの Run メニューには実行を制御する数々の機能があります。

図 145. Run メニュー



その他に、デバッグ・パースペクティブのツールバーには、デバッグのための次のようなメイン制御アイコンがあります。

図 146. デバッグ・ツールバー

これらのアイコンの目的は、左から右に次のとおりです。

- デバイスをリセットしてデバッグ・セッションをリスタート
- すべてのブレークポイントをスキップ（Ctrl+Alt+B）
- 終了して再起動
- 再開（F8）
- サスペンド
- 終了（Ctrl+F2）
- 切断
- ステップ・イン（F5）
- ステップ・オーバー（F6）
- ステップ・リターン（F7）
- 命令ステップ・モード（アセンブラ・ステップ動作）

Terminate and relaunch をクリックすると、現在のデバッグ·セッションを終了し、ソース·コードが変更されている場合は新しいプログラムをビルドしたうえでデバッグ·セッションを再起動します。

Instruction stepping mode をクリックすると、[Disassembly]ビューが開き、以降のステップ動作は、アセンブラ命令のステップ·レベルで進行します。Instruction stepping mode を再度クリックすると、C/C++ レベルのステップ動作に戻ります。

3.5.4 プログラムの実行、開始、停止

プログラムの実行、ステップ動作、停止には、次のツールバー·アイコンを使用します。

- プログラムを実行するには Resume ツールバー·アイコン(F8)を使用します。
- 関数内へとステップ·インするには Step into ツールバー·アイコン(F5)を使用します。
- 関数をステップ·オーバーするには Step over ツールバー·アイコン(F6)を使用します。
- 関数内をリターンまで実行して呼び出し元に戻るには Step return ツールバー·アイコン(F7)を使用します。
- プログラム実行を中止するには Suspend ツールバー·アイコンを使用します。

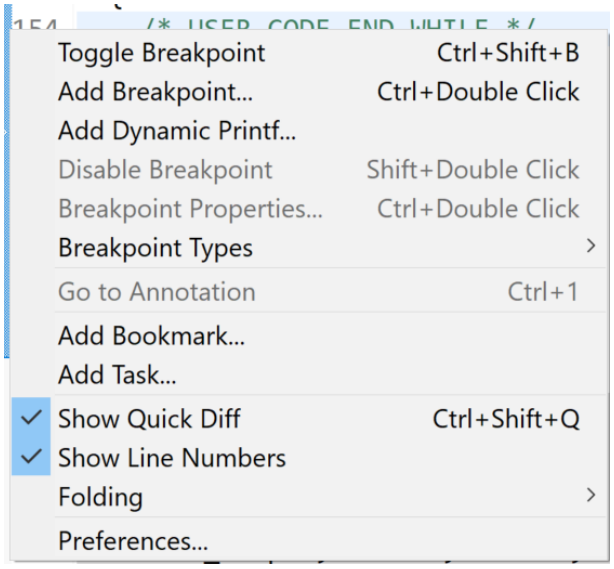
3.5.5 ブレークポイントの設定

コードにブレークポイントを設定し、その行に達するまでコードを実行させることは、デバッグ·セッションでは一般的に行われます。

3.5.5.1 標準ブレークポイント

ソース·コード行に対する標準コード·ブレークポイントは、C/C++ ソース·コード·エディタの左余白をダブルクリックするか、右クリックすることで簡単に設定できます。後者を使用した場合は、コンテキスト·メニューが表示されます。

図 147. デバッグのブレークポイント



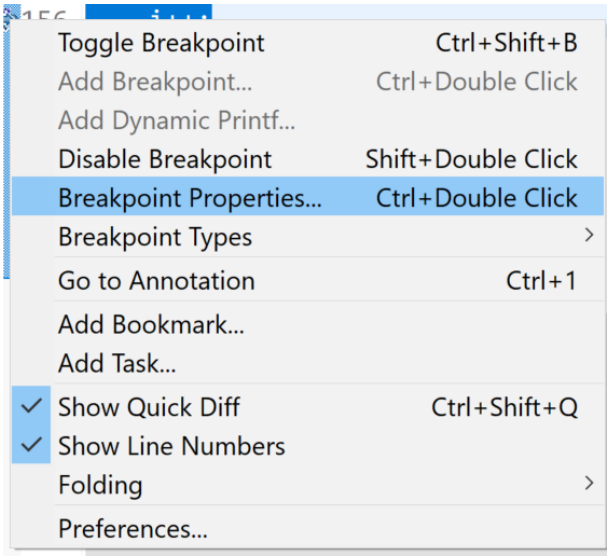
対応するソース·コード行に対してブレークポイントを設定または削除するには、メニュー·コマンド Toggle Breakpoint を使用します。

3.5.5.2 条件付きブレークポイント

ソース·コード行に標準ブレークポイントを設定すると、プログラムはその行に到達するたびに停止します。毎回停止する必要がない場合は、ブレークポイントに条件を設定することで、プログラムを実際にその行で停止するかどうかを制御できます。

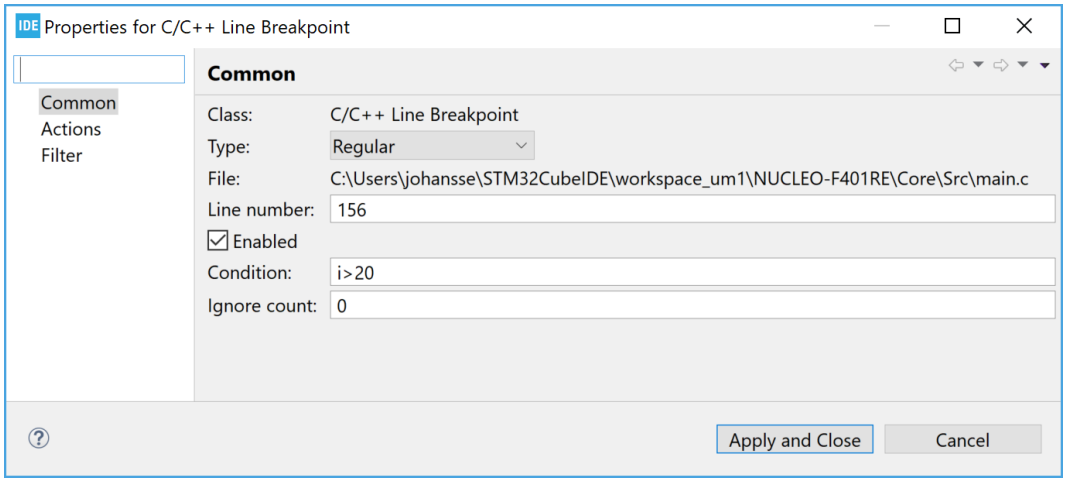
ブレークポイントのプロパティを変更するには、ブレークポイントが設定されている行の左余白にあるブレークポイント·アイコンを右クリックします。Breakpoint Properties は[Breakpoints]ビューからも開くことができます。

図 148. ブレークポイントのプロパティ



Breakpoint Properties... を選択します。次のウィンドウが開きます。下図の例では、条件として `i>20` を入力しています。

図 149. 条件付きブレークポイント



上記の条件を設定した場合、プログラムはこの行が実行されるごとに停止しますが、GDB が条件をテストし、変数 `i` が `20` より大きくない場合は、実行を再開します。GDB による条件評価には、ある程度の時間がかかります。

条件は C 言語の書式で記述します。したがって、`i%2==0` のような式を入力して、より複雑な条件を設定することも可能です。

3.5.6 動作中のターゲットへのアタッチ

STM32CubeIDE とデバッガを JTAG/SWD を介して、リセットを実行することなく組込みターゲットに接続できます。この手法は、発生頻度の低い問題の解決を試みる場合に役に立ちます。CPU クラッシュの場合、問題の根本原因を突き止める作業は、[Fault Analyzer]ビューの使用方法を習得すると、大幅に簡素化されます（セクション 7 Fault Analyzer を参照してください）。

この手法を試すときは、アプリケーションを不適切な状態で停止してハードウェアを損傷する恐れがないかを（モータ制御アプリケーションの場合など）、あらかじめ検討してください。GDB がターゲットに接続するときは CPU が停止するからです。この動作は変更できません。

デバッグ設定を変更して動作中のターゲットにアタッチするには、次の 3 または 4 段階の手順を実行する必要があります。

1. 動作中のターゲットにアタッチするようにデバッグ設定を変更します。

2. デバッグ･プローブを組込みターゲットに接続します。
3. 変更したデバッグ設定を使用してデバッグ･セッションを開始します。
4. 必要に応じて、Fault Analyzer ツールを使用し、CPU のフォルト状態を分析します（Fault Analyzer を参照してください）。

ステップ 1: デバッグ設定の変更

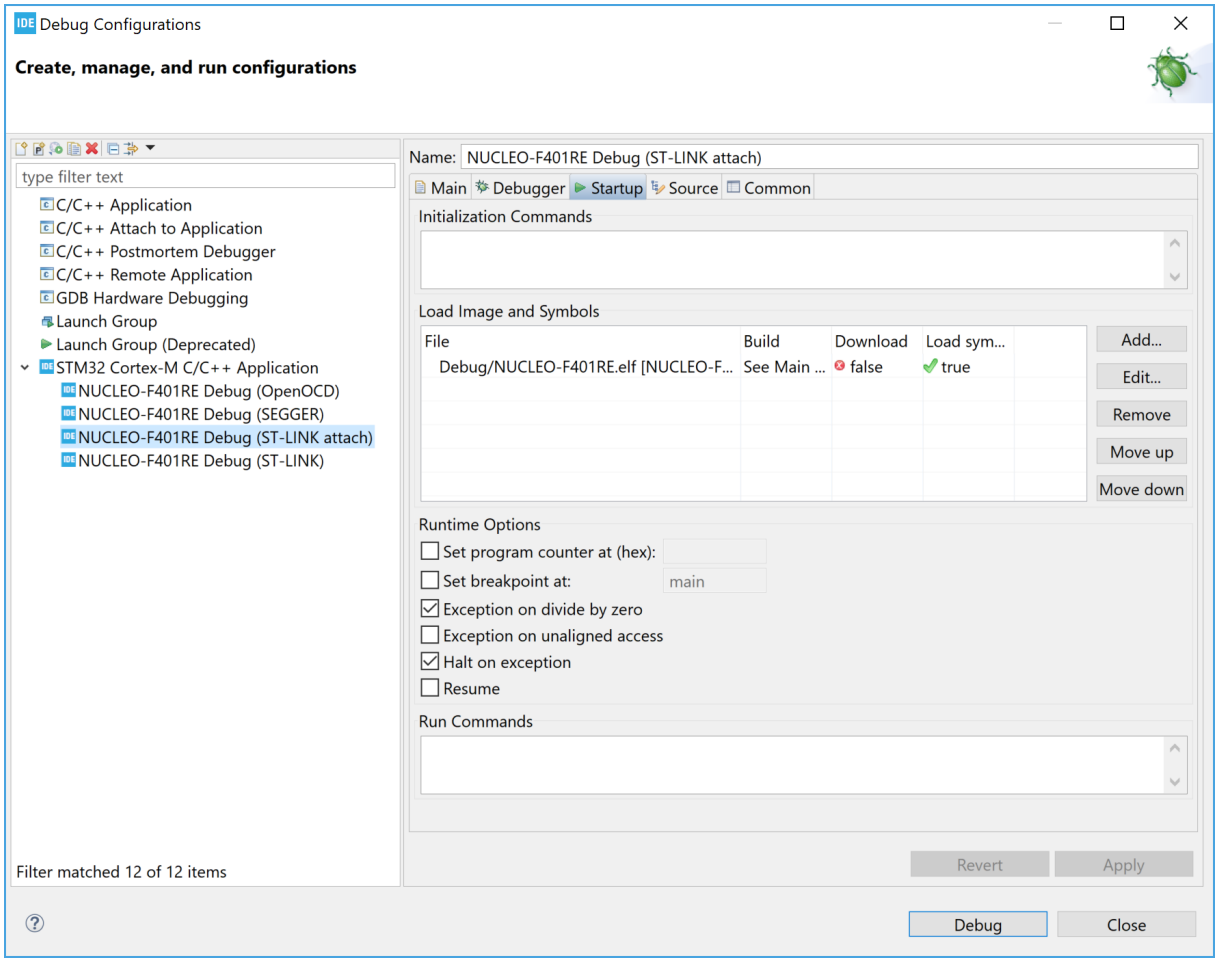
STM32CubeIDE によって生成されるデフォルトのデバッグ設定には、デバイスのリセット、新しいプログラムのダウンロード、`main` へのブレークポイント設定が含まれます。この設定は、動作中のシステムに接続する場合、システムがクラッシュしているかどうかにかかわらず、何の役にも立ちません。

デバッグ設定を変更するには、次の手順を実行します。

1. [Debug Configurations]ダイアログを開きます。
2. [Debug Configurations]ダイアログの左側フレーム内で、デバッグ対象のプロジェクトに対応するデバッグ設定を右クリックし、Duplicate を選択することでコピーを作成します。
3. 複製したデバッグ設定に名前を付けます。
4. [Debug Configurations]の[Debugger]タブの設定を次のように変更します。
   - ST-LINK GDB サーバ および OpenOCD を使用する場合は、Reset behaviour として None を選択します。
   - SEGGER J-Link GDB サーバ を使用する場合は、Reset strategy として None を選択します。
5. ST-LINK GDB サーバ および SEGGER J-Link GDB サーバ のいずれを使用する場合も、[Debug Configurations]ダイアログの[Startup]タブの設定を、次のように変更することが必要あるいは推奨されます。
   - Load Image and Symbols のファイルの Download を無効にします。
   - Set program counter at (hex) を無効にします。
   - Set breakpoint at を無効にします。
   - Exception on divide by zero と Exception on unaligned access は、無効 / 有効のどちらでも構いません。
   - Resume を無効にします。

     Resume を有効にすると、デバッガは接続時にターゲットを停止し、短時間経過してから、`continue` コマンドを送信します。

図 150. 動作中のターゲットにアタッチする場合の[Startup]タブ



ステップ 2: 組込みターゲットへの ST-LINK または SEGGER J-Link の接続

はじめに ST-LINK または SEGGER J-link をコンピュータに接続します。その後、組込みターゲットに接続します。リセットは発行されません。

ステップ 3: 変更したデバッグ設定によるデバッグ·セッションの開始

重要

誤ったデバッグ設定を使用してデバッグ·セッションを起動しないでください。ターゲットの再プログラムやリセットが発生する可能性があります。RunDebug Configurations... を使用して、左側フレーム内の変更済みデバッグ設定を選択し、Debug をクリックします。これは適用するデバッグ設定を完全に制御する、最も安全なデバッグ·セッションの開始方法であり、リセットの可能性を回避できます。

以上の手順によってデバッガが組込みターゲットに接続され、ターゲットは自動的に停止します。この時点で、アプリケーションにおけるさまざまなステータス·レジスタや変数を調べることができます。CPU がクラッシュした場合は、Fault Analyzer を使用すると、根本原因をより的確に把握できます。

3.5.7 デバッグのリスタートまたは終了

このセクションでは、デバッグ·セッションのリスタートと停止のための各種方法を説明します。

3.5.7.1 リスタート

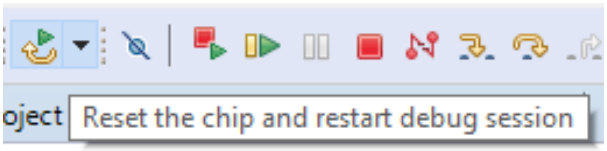
デバッグ中に発見された問題をより詳細に調べるために、プログラムのリスタートが必要になる場合があります。その場合、Reset the chip and restart debug session ツールバー·ボタンまたはメニュー コマンド RunRestart を使用してプログラムをリスタートします。これによって、デバイスがリセットされ、デバッグ設定で Resume が有効化されている場合は、プログラムが開始されます。

注 リスタートを機能させるには、割込みベクタを設定し、ハードウェア·リセットと併せて使用する必要があります。通常、STM32 のプログラムを Flash メモリに配置した場合が、これに当てはまります。しかし、プログラムが RAM など別の場所に配置されている場合は、望み通り `Reset_Handler` からプログラムを起動させるには、何らかの手動操作が必要になる可能性があります。

3.5.7.2 *リスタート設定*

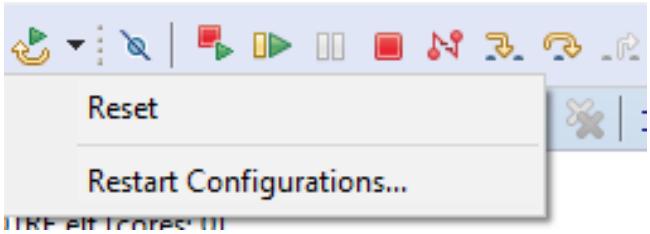
リスタート設定を作成すると、デバッグ·セッションのリセットとリスタートの実行方法を定義できます。Reset the chip and restart debug session ツールバー·アイコンの右にある矢印をクリックします。

図 151. チップ·リセットのツールバー



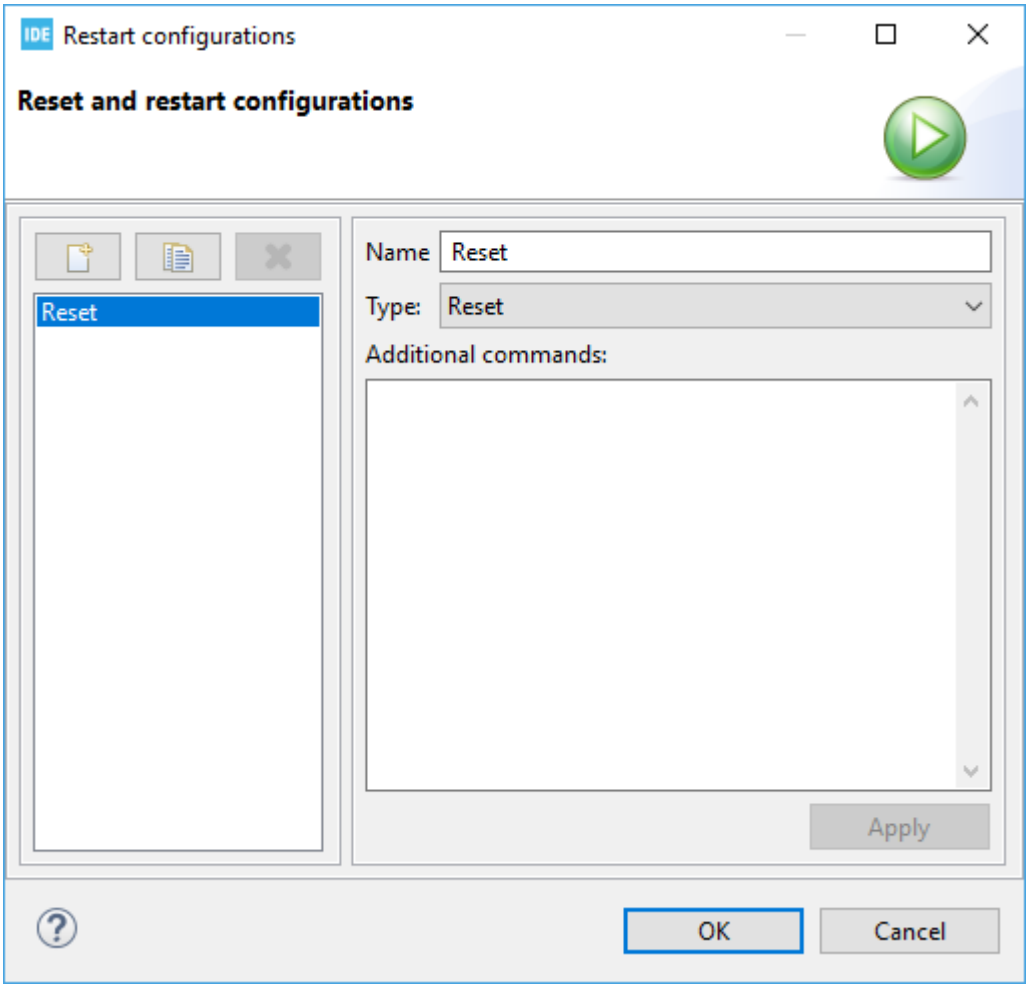
これによって Restart Configurations... オプションを含むメニューが展開されます。

図 152. [Restart Configurations...]オプション



Restart Configurations... を選択すると、リスタート設定のダイアログが開きます。

図 153. [Restart configurations]ダイアログ



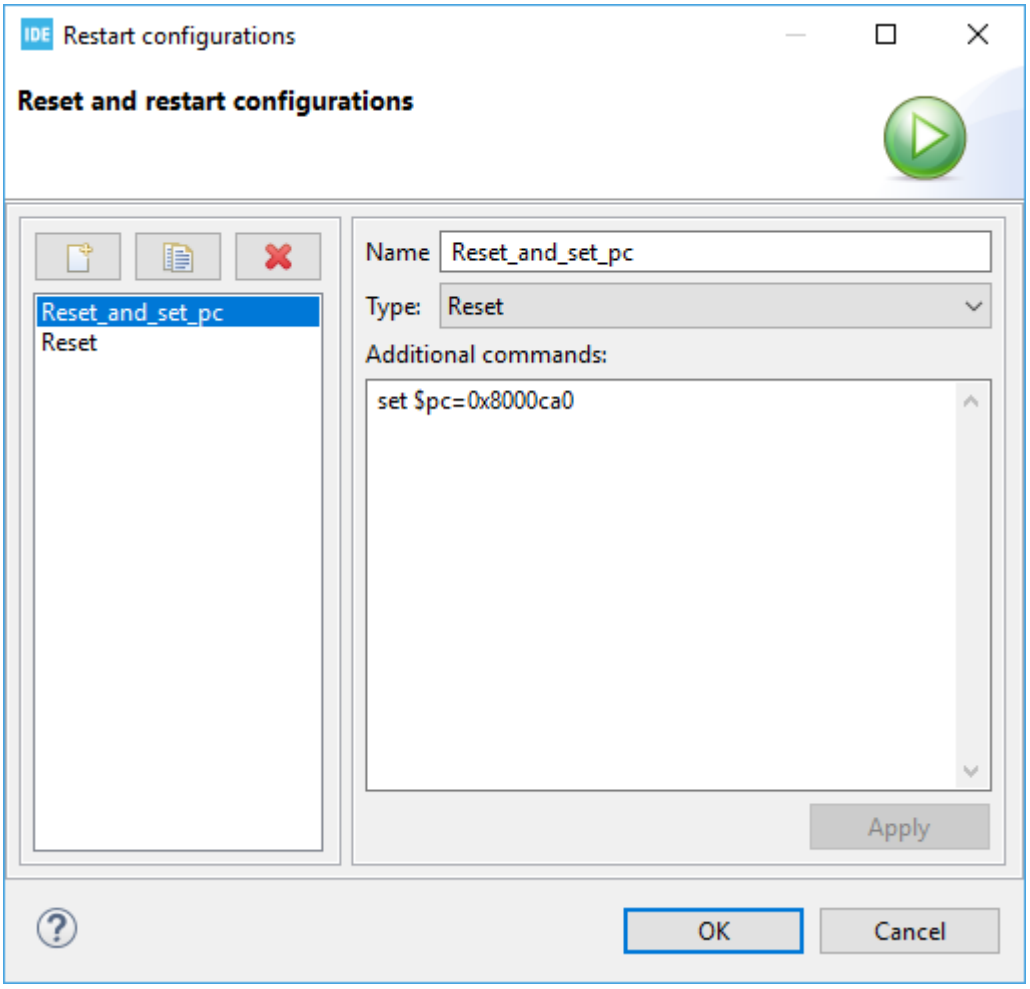
このダイアログには、左右のペインがあります。

- 左のペインはリスタート設定の選択と新規作成、既存のリスタート設定の複製、選択したリスタート設定の削除に使用します。デフォルトのリスタート設定は削除できません。
- 右のペインは、設定名(Name)の指定や、選択した設定で使用するリセットの種類(Type)の選択に使用します。リセットに付加するコマンドを追加することも可能です。

Apply をクリックして設定を保存します。

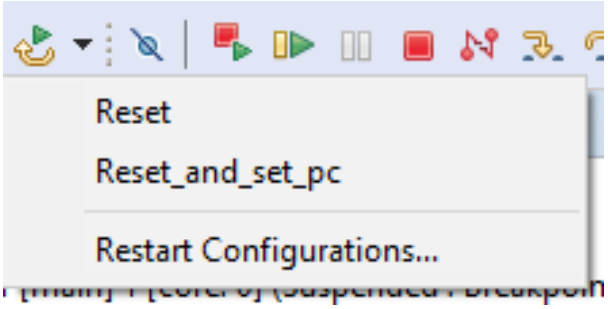
図 154 に、新しいリスタート設定を作成する場合の設定例を示します。ここでは、pc を 0x8000ca0 に設定するコマンドも追加しています。

図 154. 追加コマンドを設定した[Restart configurations]ダイアログ



複数のリセット設定を定義した場合、ツールバー·ドロップダウン·メニューに使用順で表示されます。リセットを実行するために必要な設定を 1 つ選択してください。

図 155. リスタート設定の選択



3.5.7.3 *終了*

デバッグ·セッションを停止する最も一般的な方法は、Terminate ツールバー·ボタンをクリックします。メニュー RunTerminate を使用してもデバッグ·セッションを停止できます。デバッグ·セッションが停止すると、STM32CubeIDE は自動的に C/C++ パースペクティブに切り換わります。

3.5.7.4 *終了して再起動*

デバッグ·セッション中にソース·コードを変更した場合は、Terminate And Relaunch ツールバー·ボタンを使用します。同じ目的に、メニュー·コマンド RunTerminate And Relaunch を使用することも可能です。これによって、デバッグ·セッションが停止し、プログラムが再ビルドされた後、新しいプログラムをロードして、デバッグ·セッションが再起動します。

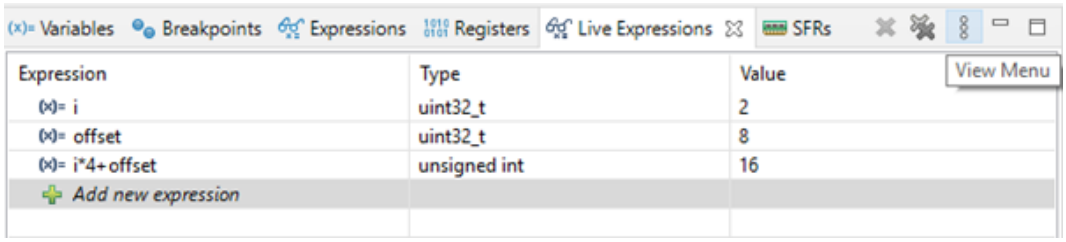
3.6 デバッグ機能

3.6.1 [Live Expressions]ビュー

STM32CubeIDE の[Live Expressions]ビューの動作は、[Expression]ビューと非常に似ていますが、すべての式がデバッグ実行中にライブでサンプリングされる点が異なります。サンプリングの速度は、サンプリングする式の数によって決まります。サンプリング対象の式の数が増えるほど、サンプリング·レートは低下します。

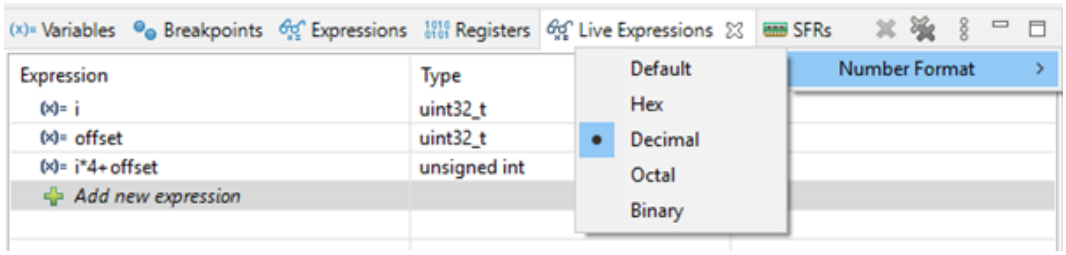
ビューには、さまざまな型のグローバル変数が多数表示されます。`(i * 4 + offset)` のような自動的に評価される数式を作成することも可能です。

図 156. [Live Expressions]



このビューでは、C 言語の構造体のような複雑なデータ型の解析や表示も可能です。同時に使用できる数値のフォーマットは 1 つだけです。このフォーマットの変更には、ドロップダウン·メニューを使用します。

図 157. [Live Expressions]の数値のフォーマット



[Live Expressions]ビューでは、プログラムの実行中にオンザフライで変数の値を変更できます。変数を選択して、値を変更してください。式は、計算を伴わない、変数を 1 つだけ含んだものでなければなりません。

注 デバッグ中に[Live Expressions]ビューを使用できるようにするには、スタートアップ時にライブ式の機能を有効にしておく必要があります。

3.6.2 ST-LINK 共有

ST-LINK GDB サーバ と OpenOCD で使用する[Debug Configurations]ダイアログの[Debugger]タブには、ST-LINK の共有を有効にするオプションがあります。ST-LINK の共有を有効にすると、ST-LINK への通信が ST-LINK サーバを介して送信されます。ST-LINK の共有を有効にすると、ST-LINK サーバにより、複数のプログラムが同じ ST-LINK にアクセスできるようになります。

STM32CubeProgrammer （STM32CubeProg）にも、共有 ST-LINK のための設定があります。これは、STM32CubeIDE のデバッグ設定で ST-LINK 共有を有効にすると、プログラムをデバッグしつつ、同時に STM32CubeProgrammer がデバイスの Flash メモリや RAM にアクセスして読み出せることを意味します。

3.6.3 複数のボードのデバッグ

2 つの ST-LINK または SEGGER J-Link プローブを同時に使用すれば、複数のボードによるデバッグが可能です。2 つの異なるマイクロコントローラに接続された 2 本のプローブを、1 台の PC の異なる USB ポートに接続します。このセクションでは、2 つの異なるボード / マイクロコントローラ HW_A と HW_B を使用する場合を想定します。

HW_A 用と HW_B 用のプロジェクトを 1 つずつ含む STM32CubeIDE の 1 つのインスタンスを実行できます。

使用するデフォルトのポートは、次のとおりです。

- 61234: ST-LINK GDB サーバ
- 3333: OpenOCD
- 2331: SEGGER J-Link

これらの設定は、[Debug Configurations]ダイアログの[Debugger]タブに表示されます。2 つのプロジェクトの一方では使用するポート番号を 61244 などの別のものに変更する必要があります。

デバッグ設定では Autostart local GDB server オプションによる GDB 接続を使用できます。複数のボードをデバッグする場合、複数のデバッグ･プローブが PC に接続されますが、各デバッグ設定に対してプローブの正しいシリアル番号を選択する必要があります。

各ボードが特定のプローブに関連付けられるように両プロジェクトのデバッグ設定を完了したら、まずボードごとの個別のテストとデバッグを開始します。その状態で適切な動作が確認されたら、次の手順に従って両ターゲットの同時デバッグに着手できます。

1. HW_A のデバッグを開始します。
2. HW_A のデバッグ･セッションが開始されると、STM32CubeIDE のパースペクティブが自動的にデバッグに切り換わります。
3. C/C++ パースペクティブに切り換えます。
4. HW_B 用のプロジェクトを選択し、そちらのデバッグを開始します。再度 デバッグ･パースペクティブが開きます。
5. [Debug]ビューには、各プロジェクトに 1 つずつ、2 つのアプリケーション･スタック / ノードが表示されます。[Debug]ビューで選択ノードを変更すると、画面が関連するエディタや変数ビューその他に更新され、選択したプロジェクトに関連する情報が表示されます。

GDB サーバ を手動で起動することも可能です。デバッグ設定の Connect to remote GDB server を選択してください。その場合、GDB サーバ が、使用する個々のポートおよびシリアル番号を定義したパラメータに基づいて起動されていること、各プロジェクトの[Debug Configurations]ダイアログで、対応するポート番号が使用されていることを確認してください。

次の例は、SEGGER J-Link GDB サーバ を、ポート番号 2341 を介して、シリアル番号 123456789 の SEGGER J-Link に接続する場合です。

```
>JLinkGDBServerCL.exe -port 2341 -if SWD -select usb=123456789
```

GDB サーバ を手動で起動する場合に使用するコマンドライン･パラメータの詳細情報は、Information Center より入手可能な GDB サーバ のマニュアルに記載されています。

3.6.4 STM32H7 マルチコアのデバッグ

STM32CubeIDE で STM32H7 マルチコア･デバイスをデバッグする方法の詳細は、[ST-09] に記載されています。

3.6.5 STM32MP1 のデバッグ

STM32CubeIDE で STM32MP1 デバイスをデバッグする方法の詳細は、[ST-08] に記載されています。

STM32MP1 シリーズのドキュメントの改訂については、www.st.com/en/microcontrollers-microprocessors/stm32mp1-series から常に最新情報を入手することをお勧めします。

3.6.6 STM32L5 のデバッグ

STM32CubeIDE で TrustZone® を使用して STM32L5 デバイスをデバッグする方法の詳細は、[ST-10] に記載されています。

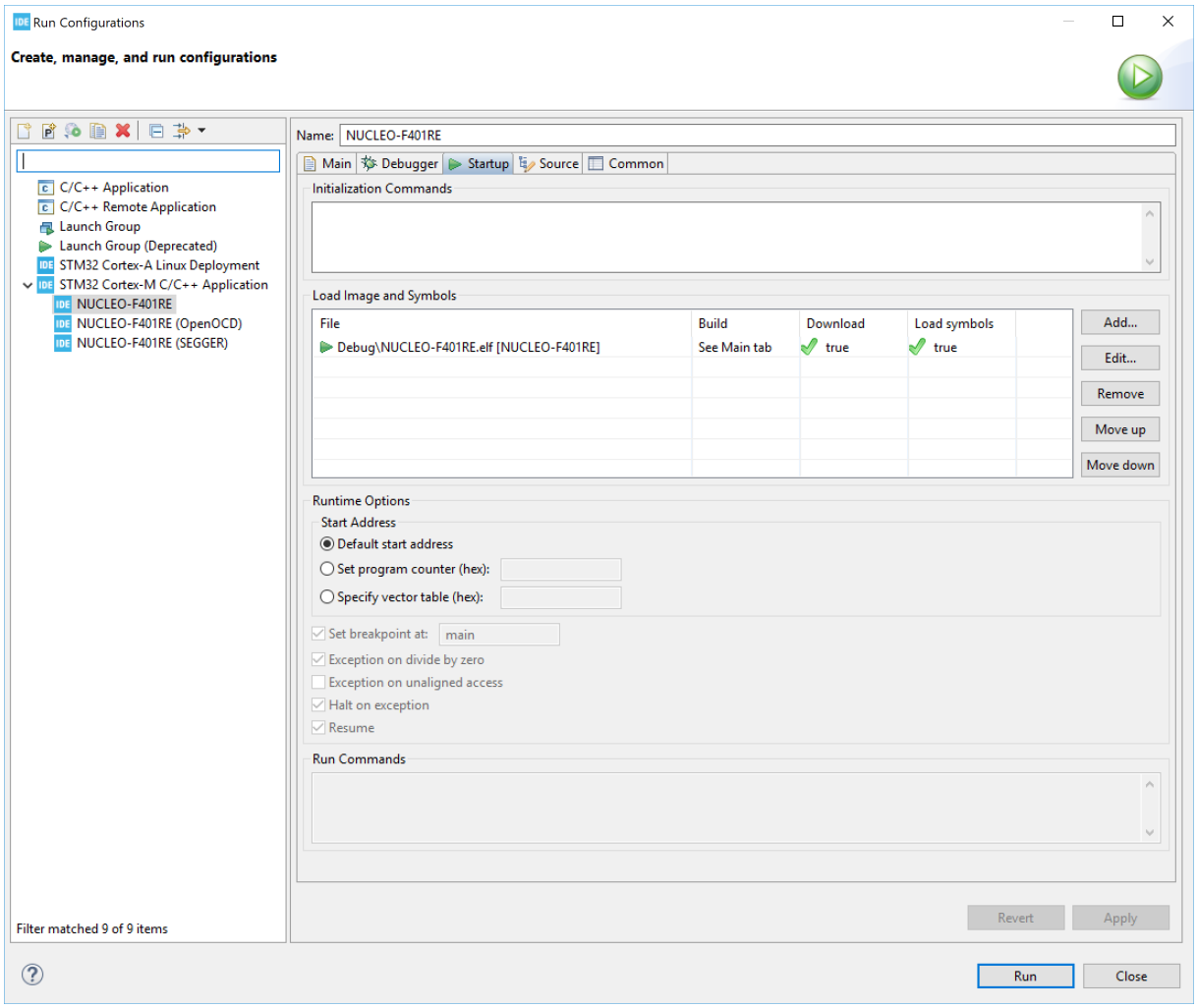
3.7 実行用設定

実行用設定を作成すると、完全なデバッグ･セッションを起動せずに、単にアプリケーションをターゲットにダウンロードしてターゲットをリセットすることが可能になります。[Run Configurations]ダイアログは、[Debug Configurations]ダイアログに似ていますが、[Startup]タブ下部の無効化されたウィジェットが実行されません。実行用設定による実行では、指定されたプログラムが Flash に書き込まれますが、プログラム･カウンタの設定後、ターゲットでプログラム実行が開始され、STM32CubeIDE の「実行」セッションは閉じます。

プロジェクトの実行用設定を作成するには、[Project Explorer]ビューでプロジェクト名を右クリックし、Run AsSTM32 Cortex-M C/C++ Application を選択します。

実行用設定を作成する、もう一つの方法は、[Project Explorer]ビューでプロジェクト名を選択し、メニュー RunRun AsSTM32 Cortex-M C/C++ Application を使用します。

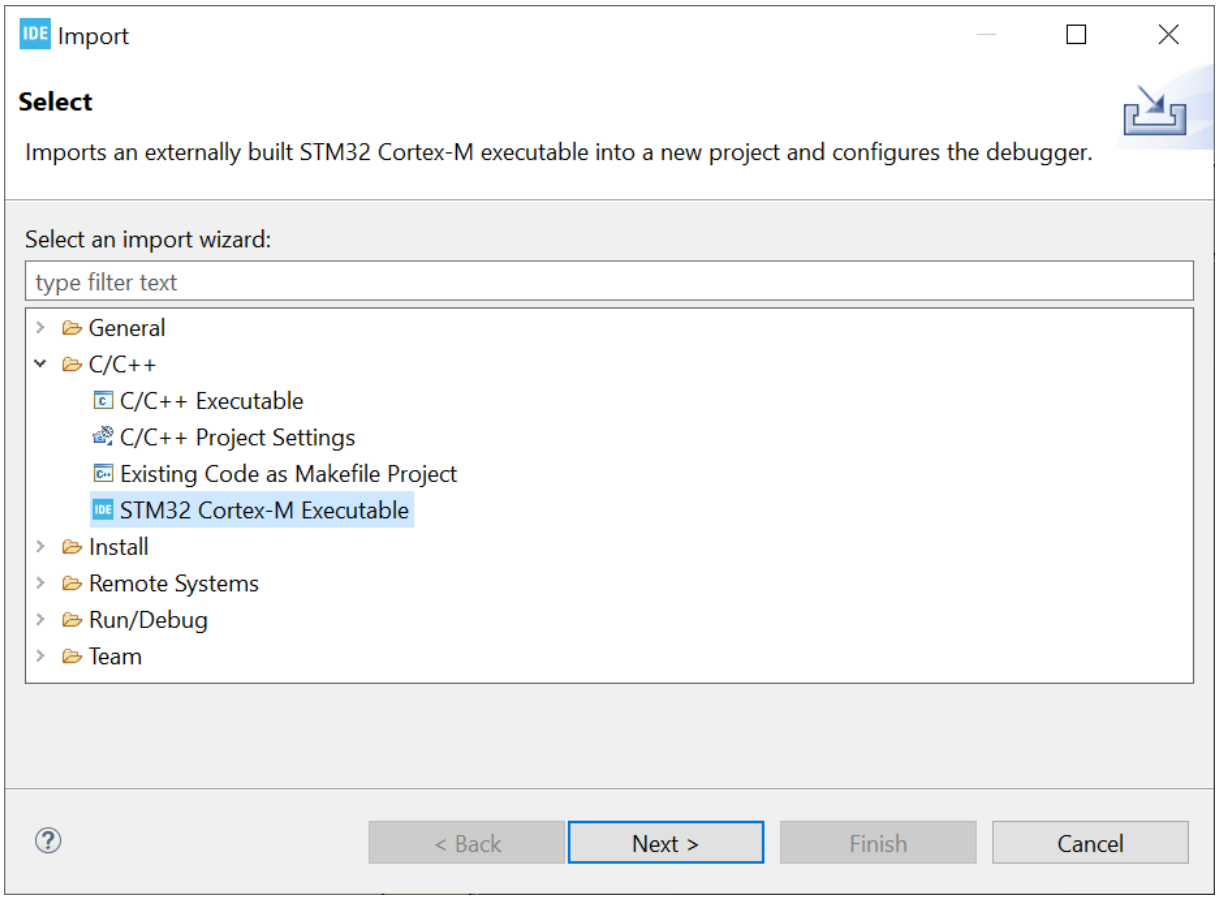
図 158. 実行用設定 - [Startup]タブ



3.8 STM32 Cortex®-M 実行可能ファイルのインポート

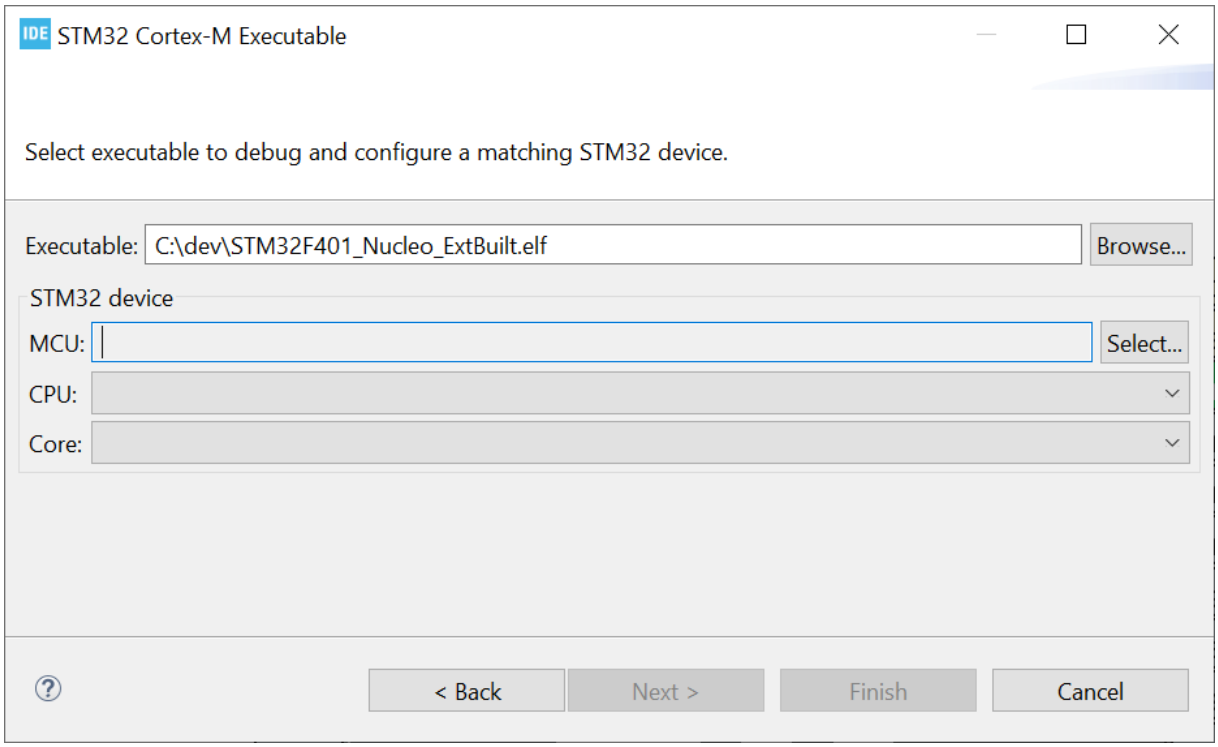
メニュー FileImport... により[Import]ダイアログを開きます。

図 159. Cortex®-M 実行可能ファイルの[Import]ダイアログ



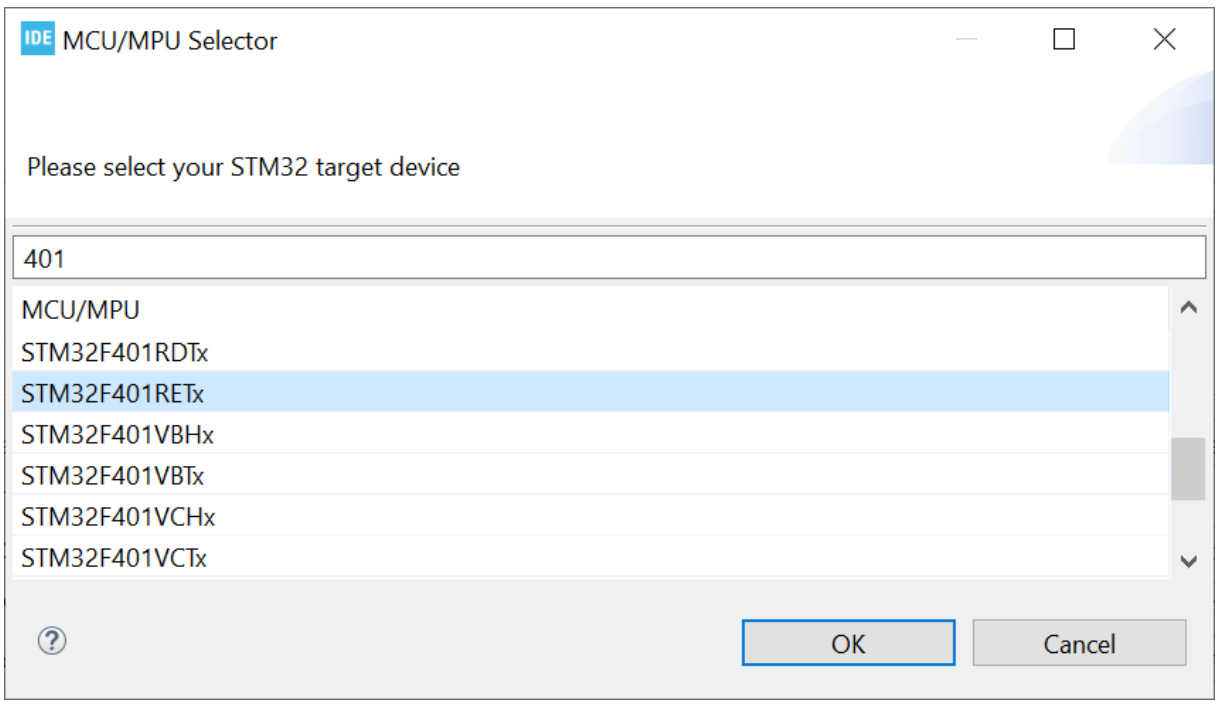
STM32 Cortex-M Executable を選択して Next> をクリックします。

図 160. [STM32 Cortex®-M Executable]ダイアログ



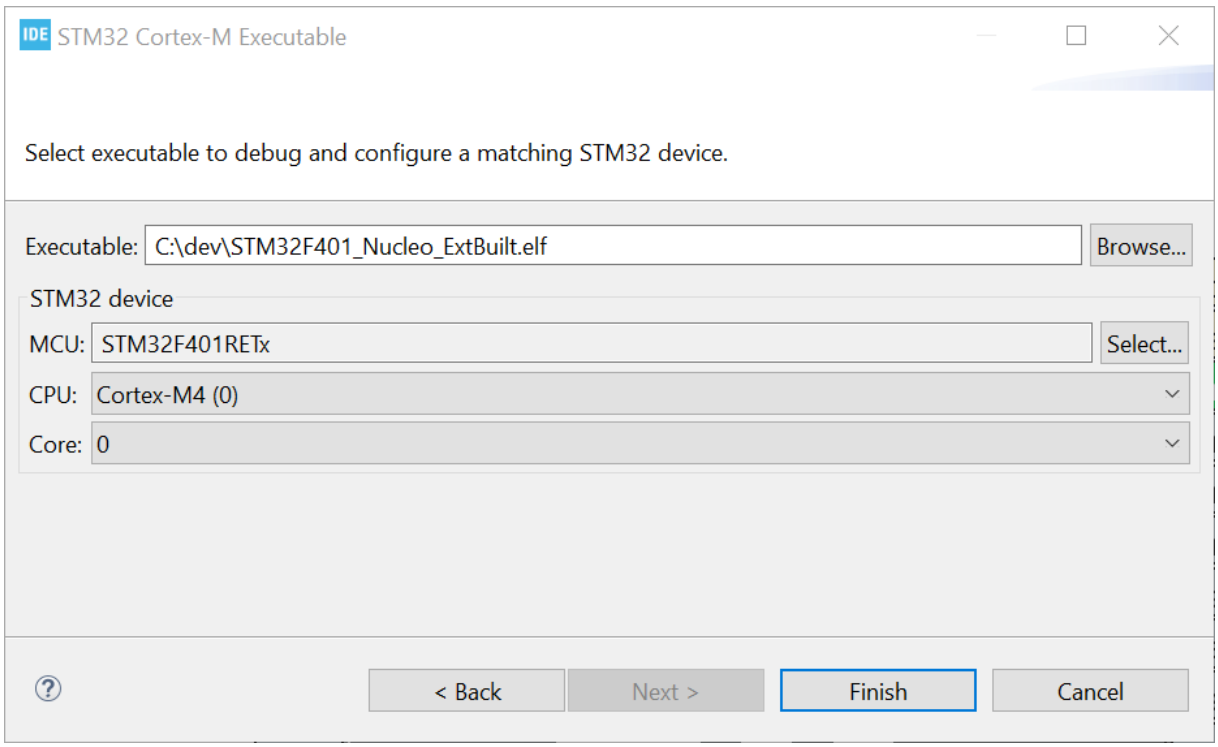
Browse... ボタンを使用して、インポートする `elf` ファイルを選択します。`elf` ファイルを選択する際、デバッグに STM32CubeIDE を使用できるように、手動で STM32 デバイスを選択する必要があります。
Select... をクリックして[MCU/MPU Selector]ダイアログを開きます。

図 161. STM32 Cortex®-M 実行可能ファイルの MCU/MPU 選択



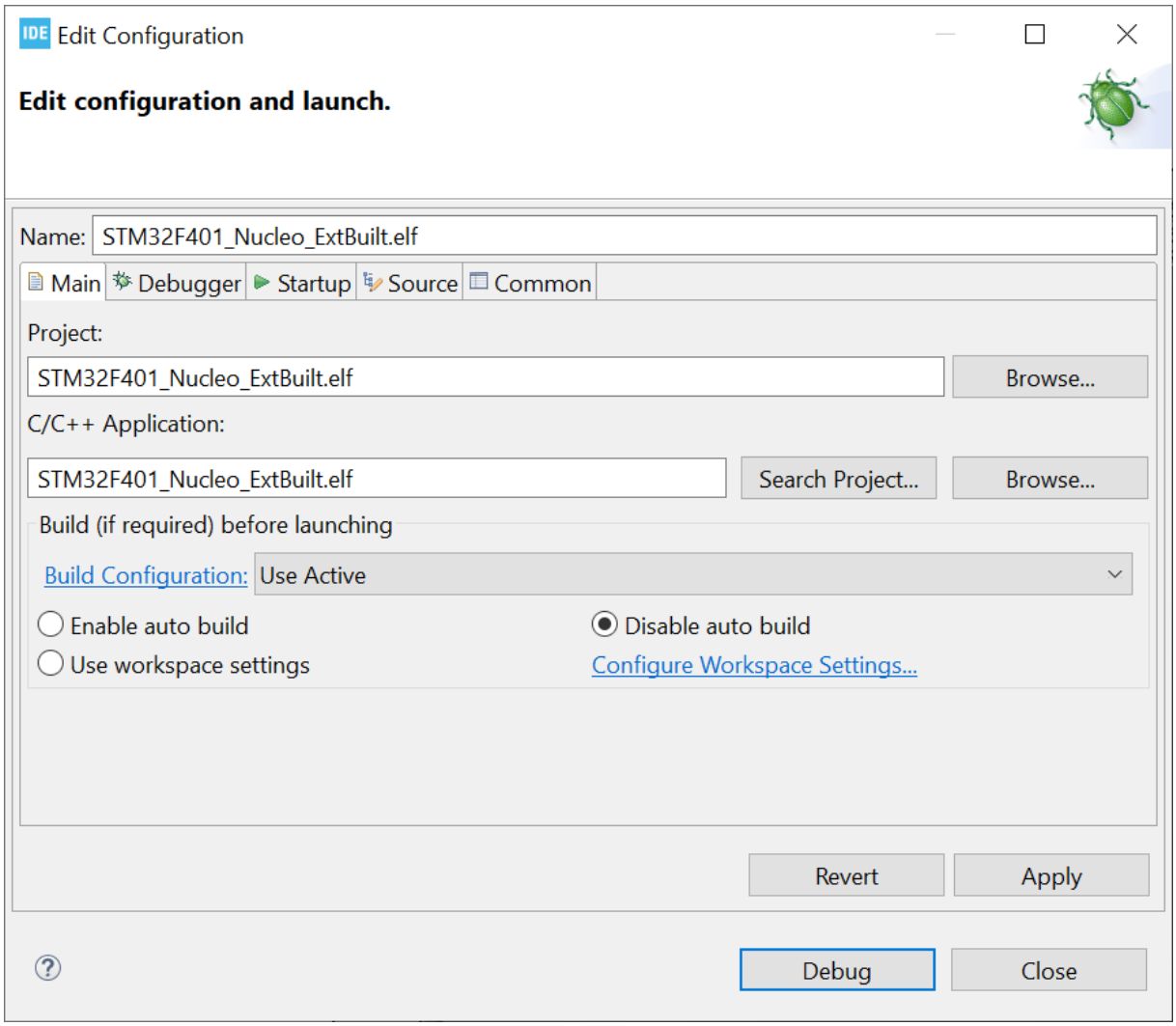
使用するマイクロコントローラまたはマイクロプロセッサを選択します。デバイスを探す際は、検索フィールドを使用できます。デバイスを選択したら OK をクリックします。
CPU とコアがダイアログに表示されます。

図 162. STM32 Cortex®-M CPU とコア



Finish をクリックすると、デバッグ設定のダイアログが自動的に開きます。

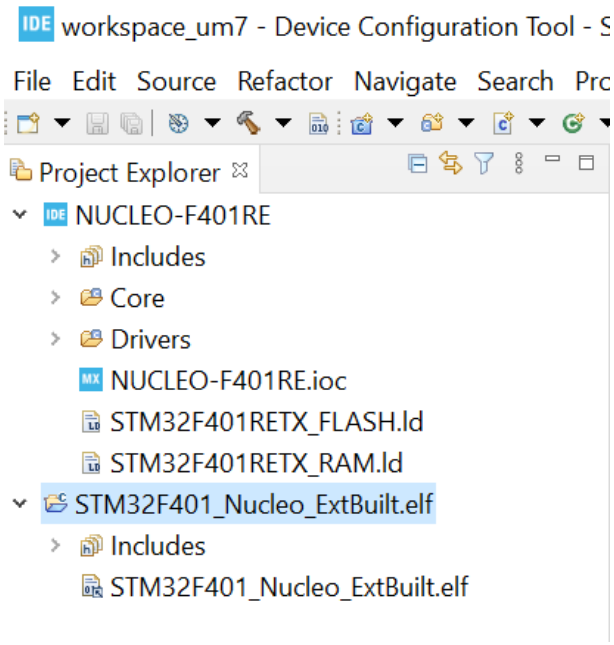
図 163. インポートしたプロジェクトの Cortex®-M デバッグ設定



この後は他の STM32CubeIDE プロジェクトと同様の方法でデバッグ設定を作成できます。設定が完了したら Debug をクリックしてデバッグ·セッションを開始します。

インポートしたプロジェクトは、[Project Explorer]ビューに表示されます。

図 164. インポートしたプロジェクトを含む[Project Explorer]ビュー



4 シリアル·ワイヤ·ビューア(SWV)トレースを使用したデバッグ

4.1 SWV と ITM の概要

このセクションでは、STM32CubeIDE で シリアル·ワイヤ·ビューア(SWV)トレースを使用する方法を説明します。

STM32 のシステム分析やリアルタイム·トレースには、シリアル·ワイヤ·ビューア(SWV)、シリアル·ワイヤ·デバッグ(SWD)、計測トレース·マクロセル(ITM)、シリアル·ワイヤ出力(SWO)など、多数の対話テクノロジーが必要です。これらのテクノロジーは、Arm® CoreSight™ デバッガ·テクノロジーの一部です。以下、それぞれについて説明します。

シリアル·ワイヤ·デバッグ (SWD)は、JTAG に似たデバッグ·ポートです。同じデバッグ機能(実行、ブレークポイントでの停止、シングルステップ)を、より少ないピン数で実現します。JTAG コネクタに置き換わる 2 ピンのインタフェース(クロック·ピンと双方向データ·ピン各 1 本)を使用します。リアルタイム·トレースは、SWD ポートだけでは実行できません。

シリアル·ワイヤ出力(SWO)ピンを SWD と組み合わせて使用します。この第 3 のピンは、プロセッサがリアルタイムのトレース·データを出力するために使用し、SWD の 2 つのピンの機能を拡張します。2 つの SWD ピンと SWO ピンの組合せにより、互換性のある Arm® プロセッサでの シリアル·ワイヤ·ビューア(SWV)リアルタイム·トレースが可能になります。

SWO ピンは 1 本しかありません。このため、SWO の送信容量を超えるデータを生成するような設定を構成してしまう可能性が高く、注意が必要です。

シリアル·ワイヤ·ビューア(SWV)とは、シリアル·ワイヤ·デバッグ(SWD)ポートとシリアル·ワイヤ出力(SWO)ピンを使用するリアルタイム·トレース·テクノロジーです。シリアル·ワイヤ·ビューア は、プロセッサを停止することなくデバッグ情報を抽出する、高度なシステム分析とリアルタイム·トレース機能を提供します。

シリアル·ワイヤ·ビューア (SWD)は、次のような種類のターゲット情報を提供します。

- データ読出しと書込みに関するイベント通知
- 例外処理の開始と終了に関するイベント通知
- イベント·カウンタ
- タイムスタンプと CPU サイクル情報(プログラムの統計的プロファイリングに使用できます)

計測トレース·マクロセル(ITM)により、アプリケーションは任意のデータを SWO ピンに書き出すことができます。このデータをデバッガで解釈、視覚化します。例えば、ITM を使用して `printf()` の出力を、デバッガの SWV コンソール·ビューにリダイレクトできます。この目的には、通常 port 0 を使用します。

ITM ポートには 32 のチャネルがあります。各種データを異なる ITM チャネルに書き出すことで、デバッガは各種チャネルのデータをさまざまな方法で解釈し、視覚化できます。

ITM ポートへの 1 バイト書き出しには 1 回の書込みサイクルしか必要ないため、アプリケーション·ロジックの実行時間が奪われることは、ほぼありません。

SWV と ITM のトレース·データに基づいて、STM32CubeIDE は特別な SWV ビューにより、高度なデバッガ機能を提供できます。

注 Arm® は Cortex®-M0 または Cortex®-M0+ コアに SWV/ITM を組み込んでいません。したがって、これらのコアに基づく STM32L053 マイクロコントローラなどの STM32 デバイスでは、SWV/ITM を使用できません。

4.2 SWV デバッグ

STM32CubeIDE で シリアル·ワイヤ·ビューア(SWV)を使用してデバッグを実行するには、JTAG プローブと GDB サーバが SWV に対応している必要があります。ボードも SWD に対応し、SWO ピンを JTAG プローブに接続して使用できなければなりません。

この後のセクションでは、デバッグ設定の作成手順、SWV 設定の構成方法、デバッグ·セッションにおける SWV トレースの使用方法について説明します。

4.2.1 SWV デバッグ設定

ステップ 1: [Debug Configurations]ダイアログを開く

メニュー RunDebug Configurations... などを使用して、変更する STM32 Cortex®-M デバッグ設定を選択します。

ステップ 2: SWD インタフェースの選択

[Debug Configurations]ダイアログで SWD インタフェースを選択します。

ステップ 3: SWV の有効化

[Debug Configurations]ダイアログで SWV を有効にします。

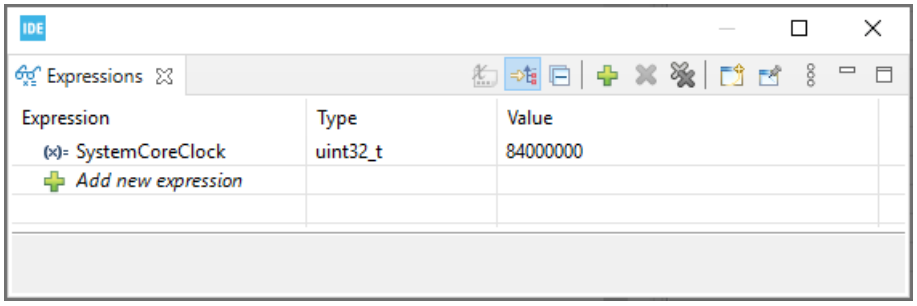
ステップ 4: コア·クロック周波数の入力

[Debug Configurations]ダイアログで Core Clock の周波数を入力します。この値は、実行するアプリケーション·プログラムで設定する値に対応させる必要があります。

STM32 サンプル·ファームウェアからインポートしたプロジェクトや、STM32CubeMX で作成したプロジェクトを使用する場合、通常、コア·クロックの設定は変数 `SystemCoreClock` に格納されます。コア·クロックの値を調べる一つの方法は、デバッグ·セッションを開始し、[Expressions]ビューに変数 `SystemCoreClock` を追加します。値を読み出す前に、アプリケーションによってシステム·コア·クロックが設定済みであることを確認してください。

`SystemCoreClock` が更新されない場合、プログラムを変更して関数 `SystemCoreClockUpdate()` への呼び出しを追加します。プログラムを再ビルドし、デバッグをリスタートして、再度 `SystemCoreClock` の値を調べます。

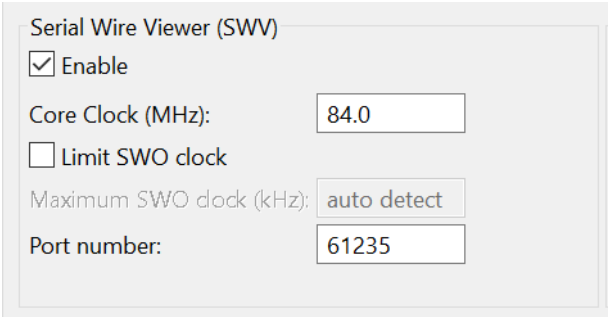
図 165. SWV コア·クロック



ステップ 5: SWO クロック周波数の入力

[Debug Configurations]ダイアログの Serial Wire Viewer (SWV) のオプションは、SWD インタフェースを選択した場合にのみ使用できます。SWV を有効にする場合は、Clock Settings の設定が必要になります。Core Clock をデバイスの速度に設定してください。SWO クロックは、使用するデバッグ·プローブとコア·クロックに応じて、可能な最大速度に自動的に設定されます。ただし、デバッグ対象のハードウェアで高速の SWO クロック速度を使用できない場合は、Limit SWO clock を有効にして、最大 SWO クロック速度を kHz 単位で入力できます。SWV の Port number には、SWV データ通信に使用するポートを設定する必要があります。SWV ポートに GDB 接続の Port number と同じ値を設定することはできません。

図 166. SWV デバッグ設定



ステップ 6: 設定の保存

[Debug Configurations]ダイアログの Apply をクリックして設定を保存します。

ステップ 7: デバッグ·セッションの開始

Debug をクリックしてデバッグ·セッションを開始します。プローブとボードが接続されていることを確認してください。

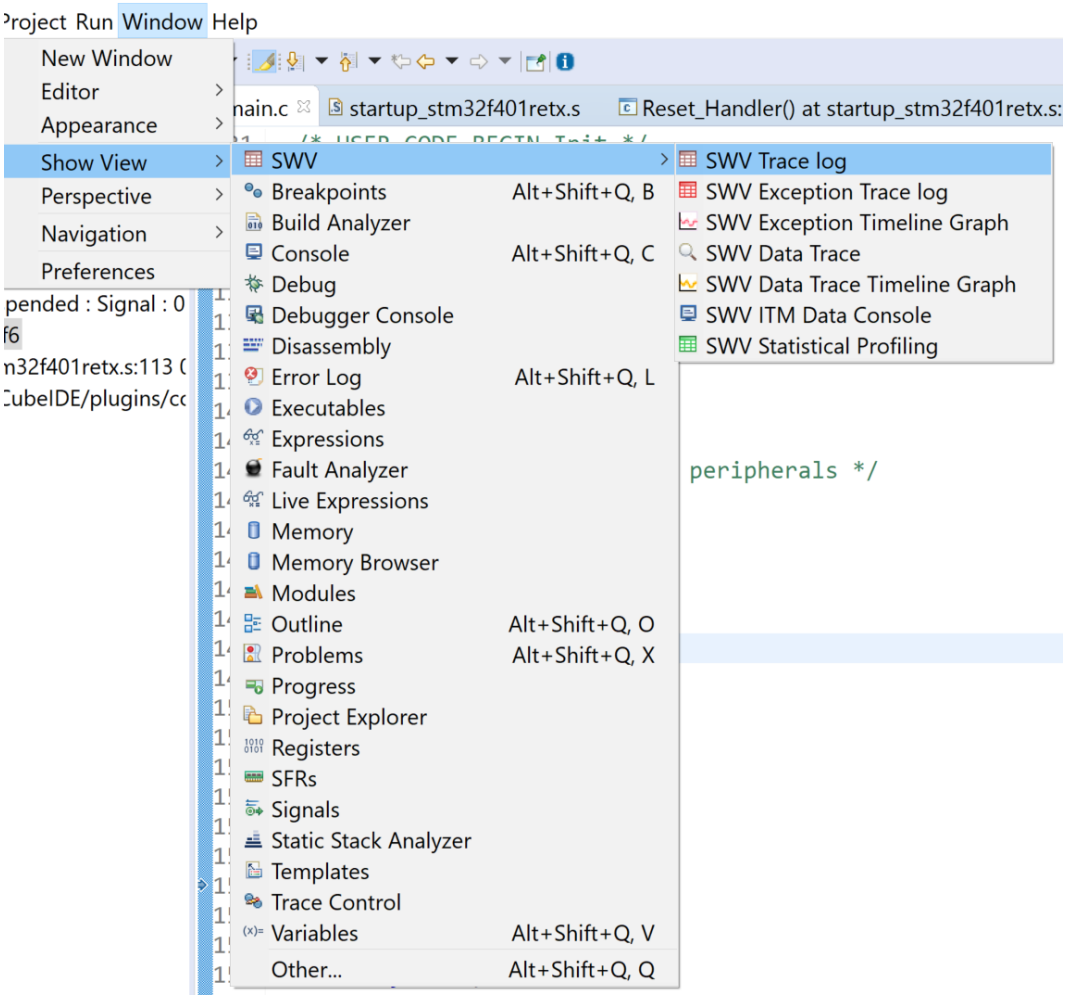
ステップ 8: ターゲットのサスペンド（場合による）

ターゲットがブレークポイントで停止しなかった場合、Suspend を使用します。

ステップ 9: SWV ビューを開く

SWV ビューのいずれか 1 つを開きます。初心者の場合、[SWV Trace log]ビューを開くことを推奨します。このビューでは、受信される SWV パケット の全体像や、トレースが適切に動作しているかどうかを簡単に把握できるからです。
メニュー·コマンド WindowShow ViewSWVSWV Trace log を選択して、[SWV Trace log]ビューを開きます。

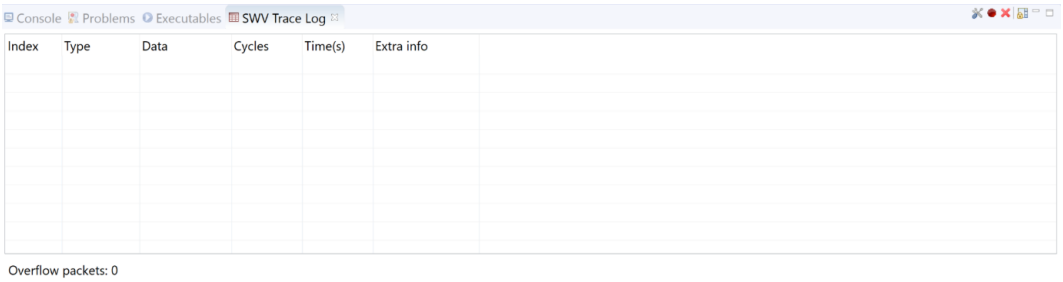
図 167. SWV ビューの表示



ステップ 10: トレース·ログの表示

[SWV Trace log]ビューが表示されました。

図 168. [SWV Trace log]ビュー



4.2.2 SWV 設定の構成

ステップ 1: [Serial Wire Viewer settings]を開く

[SWV Trace Log]ビューの Configure Trace ツールバー･ボタンをクリックして、[Serial Wire Viewer settings]ダイアログを開きます。

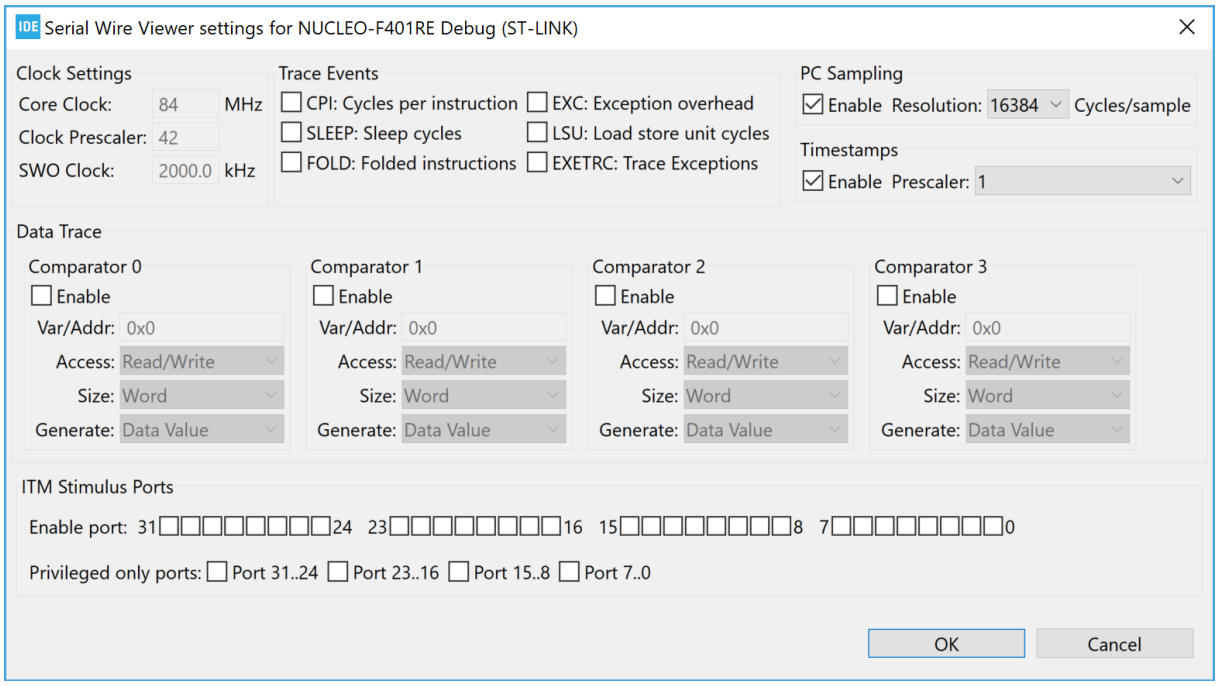
図 169. **SWV** Configure Trace **ツールバー･ボタン**

注 Configure Trace ツールバー･ボタンは、すべての SWV ビューで使用できます。

ステップ 2: トレース･データの設定

[Serial Wire Viewer settings]ダイアログで、トレースするデータを設定します。
この例では、PC Sampling と Timestamps を有効にしています。

図 170. **SWV 設定ダイアログ**



SWV 設定ダイアログには、次の項目があります。

- Clock Settings: これらのフィールドは無効化されており、現在のデバッグ·セッションの[Debug Configurations]で設定し使用している値が表示されるだけです。これらの値を変更する必要がある場合は、デバッグ·セッションを終了し、[Debug Configurations]を開いて設定し直してください。
- Trace Events: 次のイベントをトレースできます。
  - CPI: 命令あたりのサイクル数。命令が 1 サイクル目を使用して以降、サイクルごとに内部カウンタが 1 だけ増加します。このカウンタ(`DWT CPI count`)は、最大 256 までカウントすると 0 にリセットされます。リセットが発生するたびに、この パケット が 1 つ送信されます。これはプロセッサ性能の一側面を示しており、1 秒あたりの命令数の計算に使用されます。値が小さいほど性能が高いことを示しています。
  - SLEEP: スリープ·サイクル数。CPU がスリープ·モードにあるサイクル数を表します。`DWT Sleep count` レジスタでカウントします。CPU がスリープ·モードになる期間が 256 サイクルに達するたびに、この パケット が 1 つ送信されます。消費電力や外部デバイス待機に関するデバッグに使用します。
  - FOLD: フォールドされた命令の数。命令がいくつフォールド(削除)されたかを示すカウンタです。命令が 256 個フォールドされる(使用されるサイクルがゼロ)たびに、これらのイベントが 1 つ受信されます。`DWT Fold count` レジスタでカウントします。

    ブランチ·フォールドとは、ほとんどの分岐に対する予測に基づいて、実行パイプラインに送られる命令ストリームから分岐命令を完全に削除する手法です。これによって、分岐の性能が大幅に向上し、その CPI を 1 未満にまで低減します。
  - EXC: 例外のオーバーヘッド。`DWT Exception count` レジスタは、例外オーバーヘッドで消費される CPU サイクル数を継続的に監視します。カウントにはスタック動作とリターンのサイクルは含まれますが、例外コードの処理に費やされる時間は含まれません。タイマがオーバーフローするごとに、このイベントが 1 つ送信されます。プログラムに対する実際の例外処理コストを計算するために使用します。
  - LSU: ロード·ストア·ユニット(LSU)のサイクル数。`DWT LSU count` レジスタは、プロセッサが処理する LSU 動作の総サイクル数を、最初のサイクル以降、継続してカウントします。タイマがオーバーフローするごとに、このイベントが 1 つ送信されます。

    この測定値から、メモリ動作に費やされる時間を追跡できます。
  - EXETRC: 例外のトレース。例外が発生するたびに、例外処理の開始、終了、例外処理からのリターンのイベントが送信されます。これらのイベントは、[SWV Exception Trace Log]ビューで監視できます。このビューからは、その例外の例外ハンドラ·コードにジャンプできます。
- PC Sampling: このオプションを有効にすると、一定のサイクル間隔によるプログラム·カウンタのサンプリングを開始します。SWO ピンの帯域幅は限られているため、高速すぎるサンプリングは推奨できません。十分な頻度でサンプリングできる Resolution(Cycles/sample 設定)の値を実験によって決定してください。サンプリングから得られる結果はさまざまな場所に使われますが、その一つが[SWV Statistical Profiling]ビューです。
- Timestamps: イベントがいつ発生したのかを知るには、このオプションを有効化する必要があります。Prescaler の変更は、パケット のオーバーフローを削減する最後の手段としてのみ使用してください。
- Data Trace: 最大 4 つの異なる C 変数シンボルまたはメモリの固定数値領域をトレースできます。それには、コンパレータの 1 つを有効にして、トレース対象の変数名またはメモリ·アドレスを入力します。トレースする変数の値は[Data Trace]および[Data Trace Timeline Graph]の両ビューに表示できます。
- ITM Stimulus Ports: 32 個の ITM ポートが用意されており、これらをアプリケーションで使用できます。例えば、CMSIS 関数の `ITM_SendChar` を使用して、port 0 に文字を送信できます(セクション 4.3.5 SWV ITM データ·コンソールと printf のリダイレクト 参照)。ITM ポートからの パケット は、[SWV ITM Data Console]ビューに表示されます。

注 トレースするデータ量は抑えることを推奨します。ほとんどの STM32 マイクロコントローラは、SWO ピンの最大スループットよりも高速にデータを読み書きします。過剰なトレース·データは、データのオーバーフロー、パケット 喪失、場合によってはデータ破壊につながります。最適性能を得るために、現在取り組んでいる作業に必要なデータのみをトレースしてください。

SWV 実行中のオーバーフローは、SWO ピンの処理能力を上回るデータをトレースするように、SWV が設定されていることを示しています。そのような場合は、トレースするデータ量を減らしてください。

STM32CubeIDE のタイムライン·ビューのいずれかを使用する場合は、Timestamps を有効にします。デフォルトの Prescaler は 1 です。SWV の パケット オーバーフロー関連の問題が発生しない限り、この値は変更しないでください。

以下に、SWV トレースの設定例を 3 つ示します。

- 例 1: グローバル変数の値をトレースするには、Comparator を有効にして、トレース対象の変数名またはメモリ·アドレスを入力します。

  トレースする変数の値は[Data Trace]および[Data Trace Timeline Graph]の両ビューに表示されます。
- 例 2: プログラム実行のプロファイリングを行うには、PC sampling を有効にします。最初は、Cycles/sample の値を高く設定することを推奨します。

  PC サンプリングの結果は[SWV Statistical Profiling]ビューに表示されます。

- 例 3: プログラム実行中に発生する例外をトレースするには、Trace Events の EXETRC: Trace Exceptions を有効にします。
  例外に関する情報が、[SWV Exception Trace Log]ビューに表示されます。

ステップ 3: SWV 設定の保存

OK ボタンをクリックして SWV 設定を保存します。設定は、他のデバッグ設定とともに保存され、変更するまで有効です。

4.2.3 SWV トレース

ステップ 1: SWV トレース記録の開始

SWV ビューのいずれかで Start/Stop Trace ツールバー·ボタンをクリックし、SWV 設定をターゲット·ボードに送信し、SWV トレース記録を開始します。このツールバー·ボタンは、すべての SWV ビューで使用できます。ボードは適切な設定が完了するまで SWV パケット を一切送信しません。ターゲット·ボードの設定レジスタがリセットされた場合、SWV 設定を再送信する必要があります。ターゲットの実行が開始されるまで、実際のトレースは始まりません。

図 171. SWV Start/Stop Trace ツールバー·ボタン

注 ターゲットの動作中は、トレースの設定は行えません。新しい設定をボードに送信する場合は、デバッグを一時停止してください。設定は、新規作成または変更するたびにボードに送信しないと有効になりません。設定がボードに送信されるのは、Start/Stop Trace ボタンをクリックした時点です。

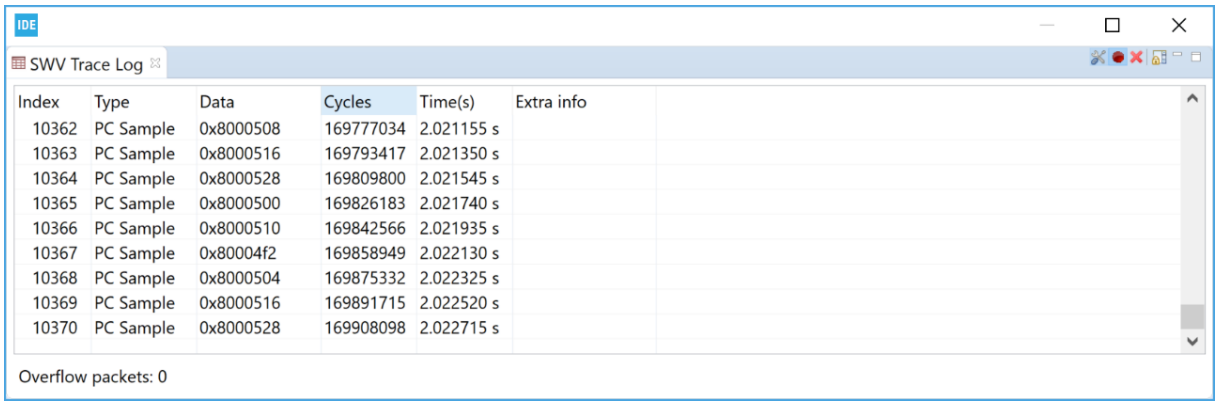
ステップ 2: ターゲットの起動

デバッグ·パースペクティブ上部にある Resume ツールバー·ボタンをクリックして、ターゲットを起動します。

ステップ 3: SWV トレース·ログの表示

SWV パケット は[SWV Trace Log]ビューに表示されます。

図 172. SWV トレース·ログ - PC サンプリング



ステップ 4: 収集された SWV データのクリア

ターゲットが動作中でない場合、収集した SWV データをクリアできます。Remove all collected SWV data ツールバー·ボタンをクリックします。このツールバー·ボタンは、すべての SWV ビューで使用できます。

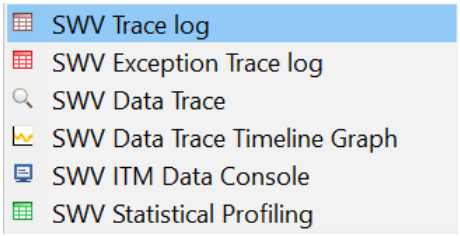
図 173. Remove all collected SWV data ツールバー·ボタン

4.3 SWV ビュー

SWV トレース·データを表示する以下の SWV ビューがあります。

- SWV Trace Log: 受信した、すべての SWV パケット をスプレッドシートに一覧表示します。トレース品質を最初に診断するときに役立ちます。
- SWV Exception Trace Log: このビューには 2 つのタブがあります。一方は[SWV Trace Log]ビューと同様の内容を表示し、もう一方は例外イベントに関する統計情報を表示します。
- SWV Data Trace: 最大 4 つの異なるシンボルまたはメモリ領域を追跡します。
- SWV Data Trace Timeline Graph: 変数の値分布の時間変化をグラフで表示します。
- SWV ITM Data Console: ターゲット·アプリケーションからの可読性のあるテキストを出力します。通常は、`printf()` 出力を ITM チャネル 0 にリダイレクトして使用します。
- SWV Statistical Profiling: プログラム·カウンタ(PC)のサンプリングに基づいた統計情報を表示します。各種関数に費やされた実行時間を表示します。

図 174. メニューから選択可能な SWV ビュー



注 各種イベントを同時に追跡するために、複数の SWV ビューを同時に開くことができます。

SWV ビューのツールバーには、次のような標準的な制御アイコンがあります。

図 175. SWV ビュー共通のツールバー

これらのアイコンの目的は、左から右に次のとおりです。

- トレースの設定
- トレースの開始 / 停止
- 収集したすべての SWV データの削除
- スクロール·ロック
- 最小化
- 最大化

SWV グラフ·ビューのツールバーには、次のような追加の制御アイコンが含まれます。

図 176. SWV グラフ·ビューの追加アイコン

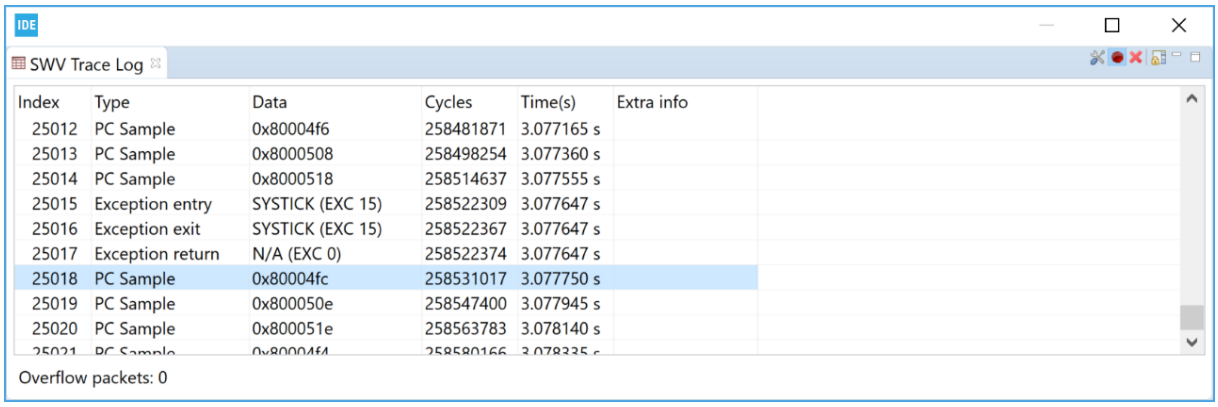
これらのアイコンの目的は、左から右に次のとおりです。

- グラフを画像として保存
- 目盛りを秒とサイクル数で切り換え
- Y 軸設定の最適化
- ズームイン
- ズームアウト

4.3.1 SWV トレース·ログ

[SWV Trace Log]ビューには、受信したすべての SWV パケット がスプレッドシートとして一覧表示されます。このビューのデータは、CSV フォーマットで他のアプリケーションにコピーできます。コピーする行を選択して Ctrl+C を押します。コピーしたデータは、他のアプリケーションに Ctrl+V コマンドで貼り付けることができます。

図 177. SWV トレース·ログ - PC サンプリングと例外



[SWV Trace Log]ビューの列に関する情報を、表 6 に示します。

表 6. SWV トレース·ログ - 列の詳細

列名	説明
Index	パケット の ID。他の SWV パケット と共有されます。
Type	パケット の種類（例 : PC サンプル、データ PC 値（comp 1）、例外、オーバーフロー）
Data	パケット データ情報。
Cycles	サイクル単位で表した パケット のタイムスタンプ。
Time(s)	秒単位で表した パケット のタイムスタンプ。
Extra info	必要に応じて表示される パケット の追加情報。

4.3.2 SWV 例外トレース·ログ

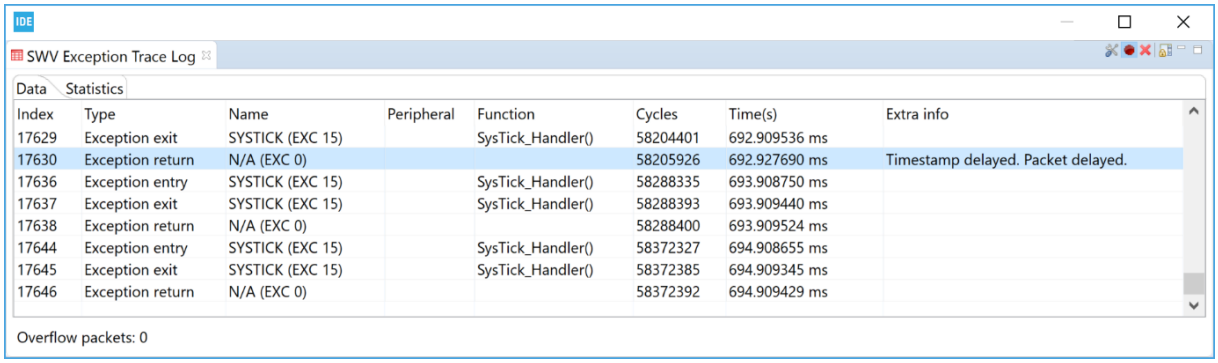
[SWV Exception Trace Log]ビューは、2 つのタブから構成されています。

[Data]タブ

最初のタブは、[SWV Trace Log]ビューに似ていますが、表示は例外イベントに限られます。イベントの種類に関する追加情報も表示されます。データは、コピーして他のアプリケーションに貼り付けることができます。各行は、対応する例外ハンドラのコードにリンクされています。イベントをダブルクリックすると、対応する割込みハンドラのソース·コードが[Editor]ビューに表示されます。

注 プログラム実行中の例外をトレースするには、[Serial Wire Viewer settings]ダイアログで Trace Events の EXETRC: Trace Exceptions を有効にします。各割込み パケット のサイクルと時間をログに記録するには Timestamps を有効にします。

図 178. [SWV Exception Trace Log]ビュー - [Data]タブ



[SWV Exception Trace Log]ビュー - [Data]タブの列に関する情報を、表 7 に示します。

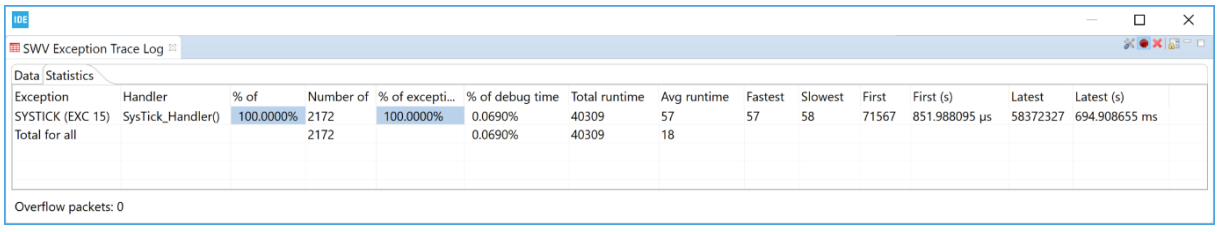
表 7. SWV 例外トレース･ログ - [Data]タブの列の詳細

列名	説明
Index	例外 パケット の ID。他の SWV パケット と共有されます。
Type	例外 1 件あたり、3 つの パケット が生成されます。例外処理の開始（entry）、例外処理の終了（exit）、例外処理からのリターン（return）に対応する 3 つの パケット です。
Name	例外の名前。例外または割込みの番号も表示されます。
Peripheral	例外に対応するペリフェラルです。
Function	この割込みの割込みハンドラ関数の名前。デバッグを一時停止すると更新されます。デバッグ･セッションの全期間、キャッシュに保存されます。関数をダブルクリックすると、その関数のソース･コードがエディタに表示されます。
Cycles	サイクル単位で表した例外のタイムスタンプ。
Time(s)	秒単位で表した例外のタイムスタンプ。
Extra info	必要に応じて表示される パケット の追加情報。

[Statistics]タブ

2 番目のタブには、例外イベントに関する統計情報が表示されます。この情報は、コードの最適化に大きく貢献する場合があります。エディタに例外ハンドラのソース･コードを開く、ハイパーテキスト･リンクも備えています。

図 179. [SWV Exception Trace Log]ビュー - [Statistics]タブ



[SWV Exception Trace Log]ビュー - [Statistics]タブの列に関する情報を、表 8 に示します。

表 8. SWV 例外トレース･ログ - [Statistics]タブの列の詳細

列名	説明
Exception	メーカーが提供する例外の名前です。例外または割込みの番号も表示されます。
Handler	この割込みの割込みハンドラの名前。デバッグを一時停止すると更新されます。デバッグ･セッションの全期間、キャッシュに保存されます。

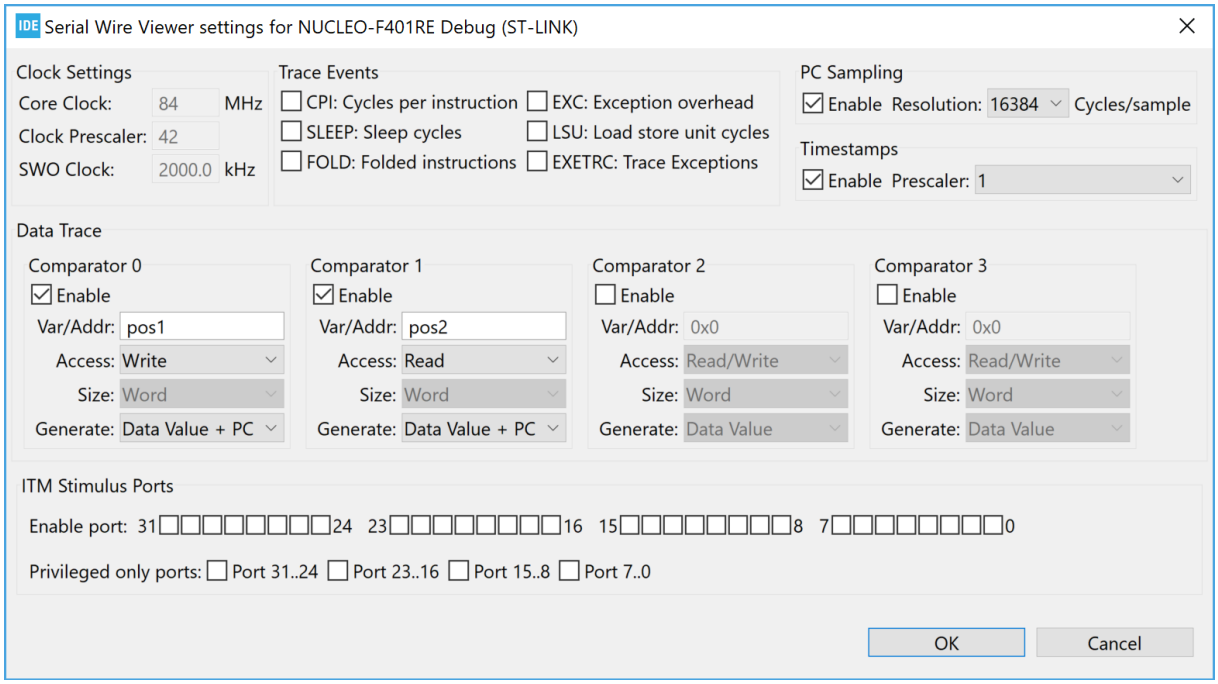
列名	説明
	ハンドラをダブルクリックすると、その関数のソース·コードがエディタに表示されます。
% of	すべての例外件数の中で、この例外の種類が占める割合(%)。
Number of	この例外の種類の SWV によって受信された 開始パケットの総数。
% of exception time	すべての例外の実行時間の中で、この例外の種類が占める割合(%)。
% of debug time	このデバッグ·セッションの総実行時間の中で、この例外の種類が占める割合(%)。すべてのタイマは[Empty SWV Data]ボタンをクリックした時点で再開されます。
Total runtime	サイクル単位で表した、この例外の種類の総実行時間。
Avg runtime	サイクル単位で表した、この例外の種類の平均実行時間。
Fastest	サイクル単位で表した、この例外の種類の最速実行時間。
Slowest	サイクル単位で表した、この例外の種類の最遅実行時間。
First	この例外の種類の開始イベントを最初に受信した時刻(サイクル単位)。
First(s)	この例外の種類の開始イベントを最初に受信した時刻(秒単位)。
Latest	この例外の種類の開始イベントを最後に受信した時刻(サイクル単位)。
Latest(s)	この例外の種類の開始イベントを最後に受信した時刻(秒単位)。

4.3.3 SWV データ·トレース

[SWV Data Trace]ビューでは、最大 4 つの異なるシンボルまたはメモリ領域をトレースできます。例えば、グローバル変数を名前によって参照できます。読出し、書込み、読出し / 書込み時のデータのトレースが可能です。

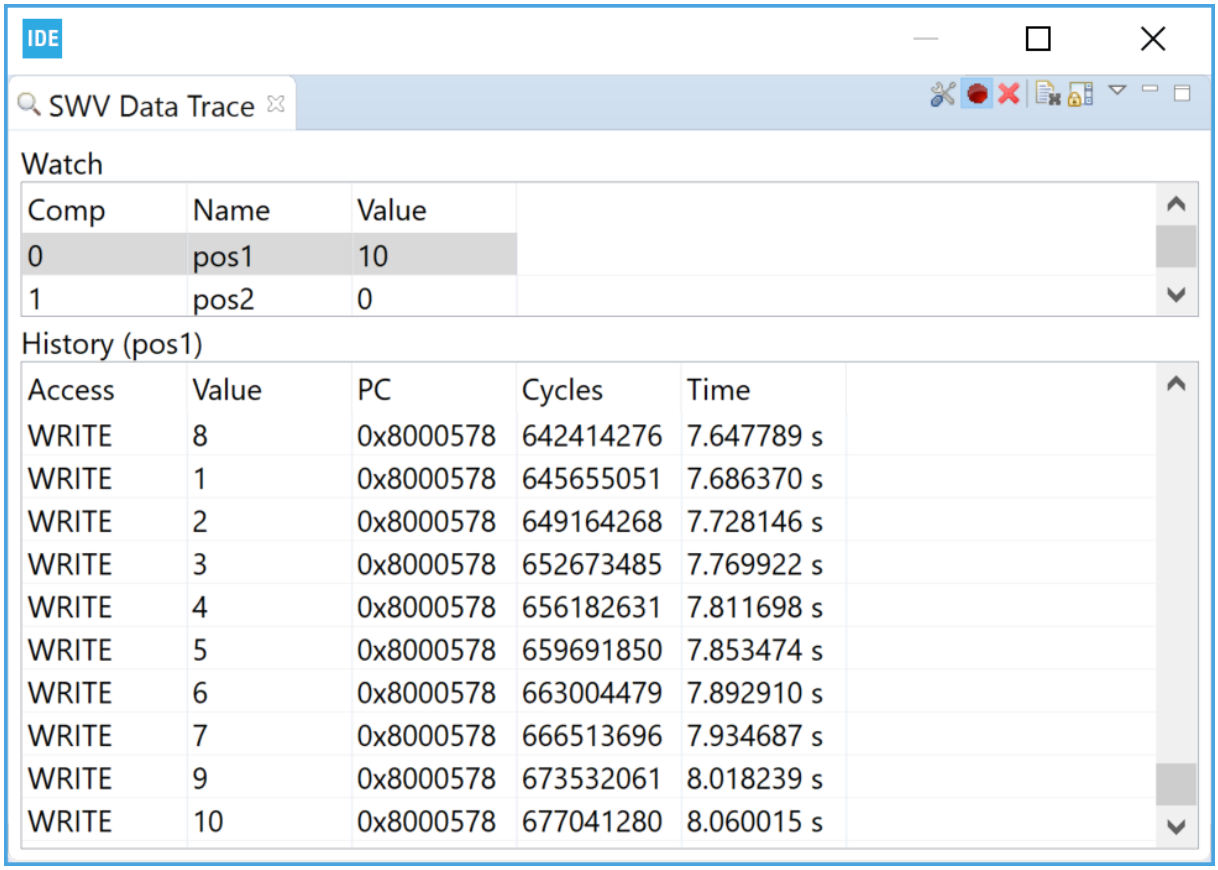
[Serial Wire Viewer settings]ビューの Data Trace を有効にします。図 180 では、プログラム内の 2 つのグローバル変数 `pos1` と `pos2` を Write アクセス時にトレースしています。

図 180. SWV データ·トレースの設定



SWV トレースを有効にしてプログラムをデバッガで実行している間に、`pos1` データを含む Comparator 0 を Watch リストで選択すると、[SWV Data Trace]ビューに、次のような情報が表示されます。

図 181. SWV データ·トレース



[SWV Data Trace]ビューの列に関する情報を、表 9 に示します。

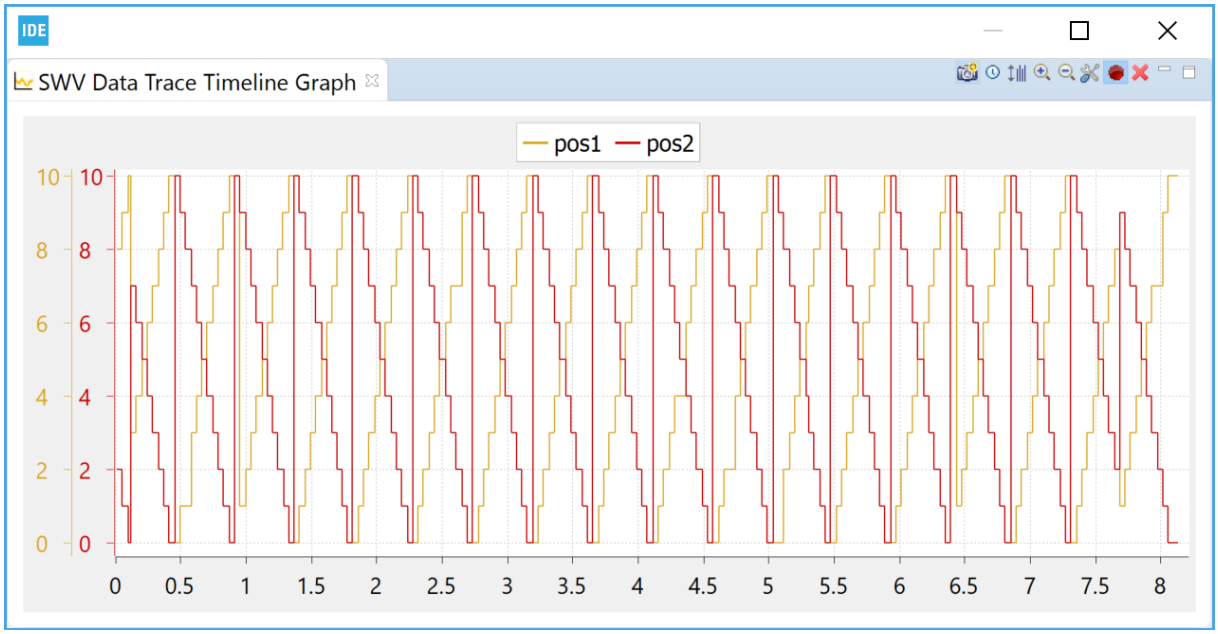
表 9. SWV データ·トレース - 列の詳細

列名	説明
Access	アクセスの種類(読出しまたは書込み)。
Value	読出しまたは書込みデータの値。
PC	読出しまたは書込みアクセスが発生した PC の位置。
Cycles	サイクル単位で表した パケット のタイムスタンプ。
Time(s)	秒単位で表した パケット のタイムスタンプ。

4.3.4 SWV データ·トレース·タイムライン·グラフ

[SWV Data Trace Timeline Graph]ビューには、変数の値分布の時間変化がグラフで表示されます。[SWV Data Trace]ビューの変数またはメモリ領域にも適用されます。カウントアップおよびカウントダウンするグローバル変数 `pos1` と `pos2` を表示するタイムライン·グラフを使用すると、次のような表示が得られます。

図 182. SWV データ·トレース·タイムライン·グラフ



[SWV Data Trace Timeline Graph]ビューには、次のような機能があります。

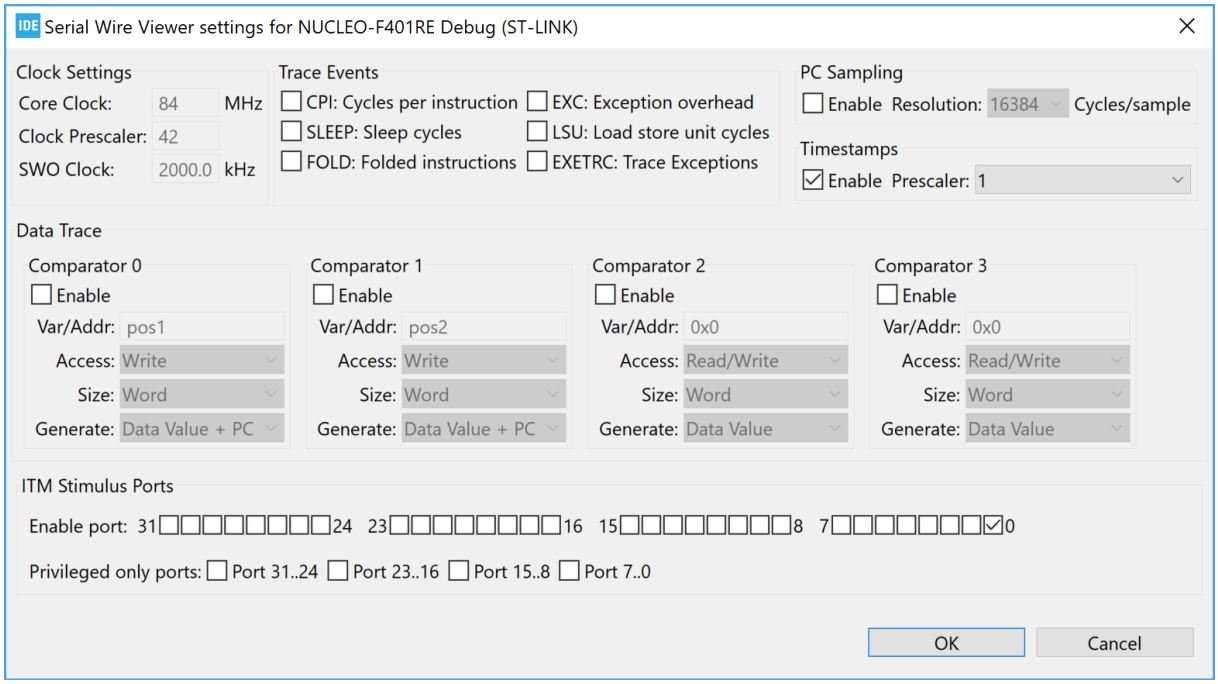
- カメラのツールバー·ボタンをクリックすると、グラフを JPEG 画像ファイルとして保存できます。
- デフォルト設定の場合、グラフには時間が秒単位で表示されますが、時計のツールバー·ボタンをクリックすることでサイクル単位に変更できます。
- Y 軸のツールバー·ボタンをクリックすると、グラフのデータ範囲がちょうど収まるように Y 軸が調整されます。
- + および - のツールバー·ボタンをクリックするとグラフのズームイン / ズームアウトが可能です。
- デバッグの実行中は、ズーム範囲が制限されます。デバッグを一時停止すると、詳細のズームが可能になります。

4.3.5 SWV ITM データ·コンソールと printf のリダイレクト

[SWV ITM Data Console]ビューには、ターゲット·アプリケーションからの可読性のあるテキストが出力されます。通常は、`printf()` 出力を ITM チャネル 0 にリダイレクトして使用します。他の ITM チャネルにも、固有のコンソール·ビューを関連付けることができます。

[SWV ITM Data Console]ビューを使用するには、まず[Serial Wire Viewer settings]ダイアログにある 32 個の ITM ポートの 1 つ以上を有効化します。

図 183. SWV 設定



ITM ポートからの パケット は、[SWV ITM Data Console]ビューに表示されます。アプリケーションで CMSIS 関数の `ITM_SendChar()` を使用すれば、port 0 に文字を送信でき、`printf()` 関数をリダイレクトして `ITM_SendChar()` 関数を使用できます。

以下に、`printf` を ITM を介してリダイレクトするための設定方法を説明します。

1. まず、ファイル `syscalls.c` を設定します。通常、`syscalls.c` ファイルは、`main.c` と同じソース･フォルダに保存されています。

   プロジェクトに `syscalls.c` ファイルが用意されていない場合、別の STM32CubeIDE プロジェクトからコピーできます。ファイルを入手する一つの方法は、まずデバイス用に空の STM32 プロジェクトを新規作成します。このプロジェクトの `Src` フォルダに、`syscalls.c` ファイルが保存されています。このファイルを、必要なプロジェクトのソース･フォルダにコピーします。

2. 次のように、`syscalls.c` ファイル内の `_write()` 関数のコードを `__io_putchar()` 呼び出しの代わりに `ITM_SendChar()` を呼び出すものに置き換えます。

```
int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        //__io_putchar(*ptr++);
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

3. `ITM_SendChar()` 関数を含む `core_cmX.h` ファイルを見つけます。`core_cmX.h` ファイルは、デバイス･ペリフェラル･アクセス･レイヤのヘッダ･ファイルでインクルードされます(`stm32f4xx.h` ファイルなど。このヘッダ･ファイルをさらに `syscalls.c` ファイルでインクルードする必要があります)。

```
#include "stm32f4xx.h"
```

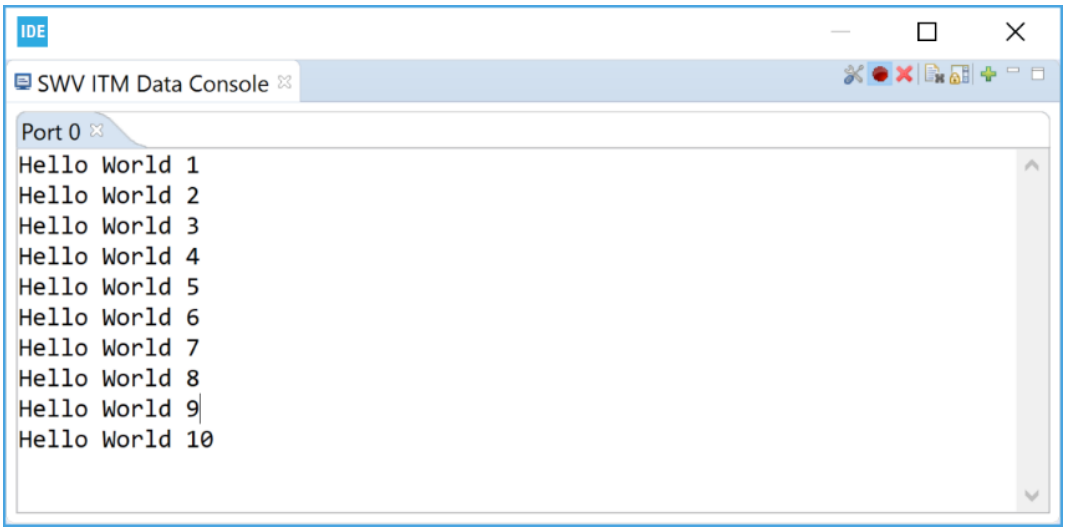
   デバイス･ペリフェラル･アクセス･レイヤのヘッダ･ファイルは[Include Browser]ビューを使用すると見つけることができます。[Include Browser]ビューにコア･ファイルをドロップして、どのファイルが `core_cmX.h` ファイルをインクルードしているのかを確認します。

4. アプリケーションに `stdio.h` のインクルードと `printf()` への呼び出しを追加することでテストします。`printf()` が過度に頻繁に呼び出されないように注意してください。

```
#include <stdio.h>

        printf("Hello World %d\n", pos1);
```

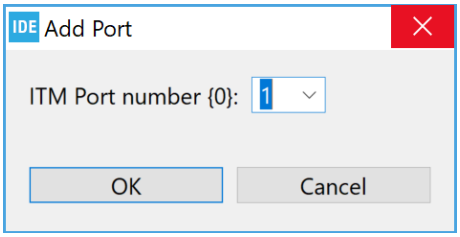
5. デバッグ·セッションを開始し、[SWV ITM Data Console]ビューで ITM port 0 を有効にします。
6. [SWV ITM Data Console]ビューを開き、このビューの赤色の Start/Stop Trace ツールバー·ボタンを使用してトレースを開始します。
7. プログラムを起動します。このビューの[Port 0]タブに、printf コマンドのログが出力されます。

図 184. SWV ITM データ·コンソール



8. ツールバーの緑色の + ボタンをクリックすることで、[Port x]タブ(x は 1 ~ 31 の値)を新規作成できます。これによって[Add Port]ダイアログが表示されます。このダイアログで、[SWV ITM Data Console]ビューにタブとして表示するために開く ITM Port number を選択します。

図 185. SWV ITM ポートの設定



注 他の ITM ポート·チャネルに文字を送信する関数の書き方については、`ITM_SendChar()` 関数の中身を吟味してください。

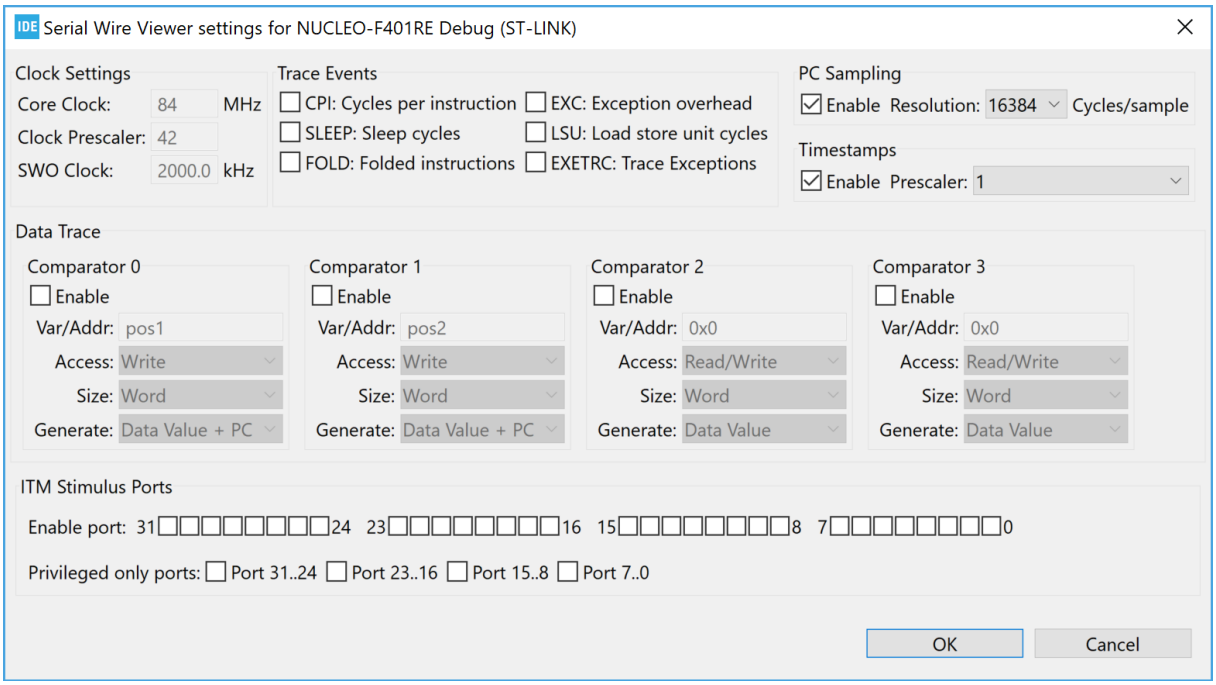
4.3.6 SWV 統計プロファイリング

[SWV Statistical Profiling]ビューには、プログラム·カウンタ(PC)のサンプリングに基づいた統計情報が表示されます。各種関数に費やされた実行時間が表示されます。これは、コードの最適化に役立ちます。データは、コピーして他のアプリケーションに貼り付けることができます。ビューは、デバッグをサスペンドしたときに更新されます。

1. 図 186 のとおり、プログラム·カウンタのサンプルを送信するように SWV を設定します。PC Sampling と Timestamps を有効にします。

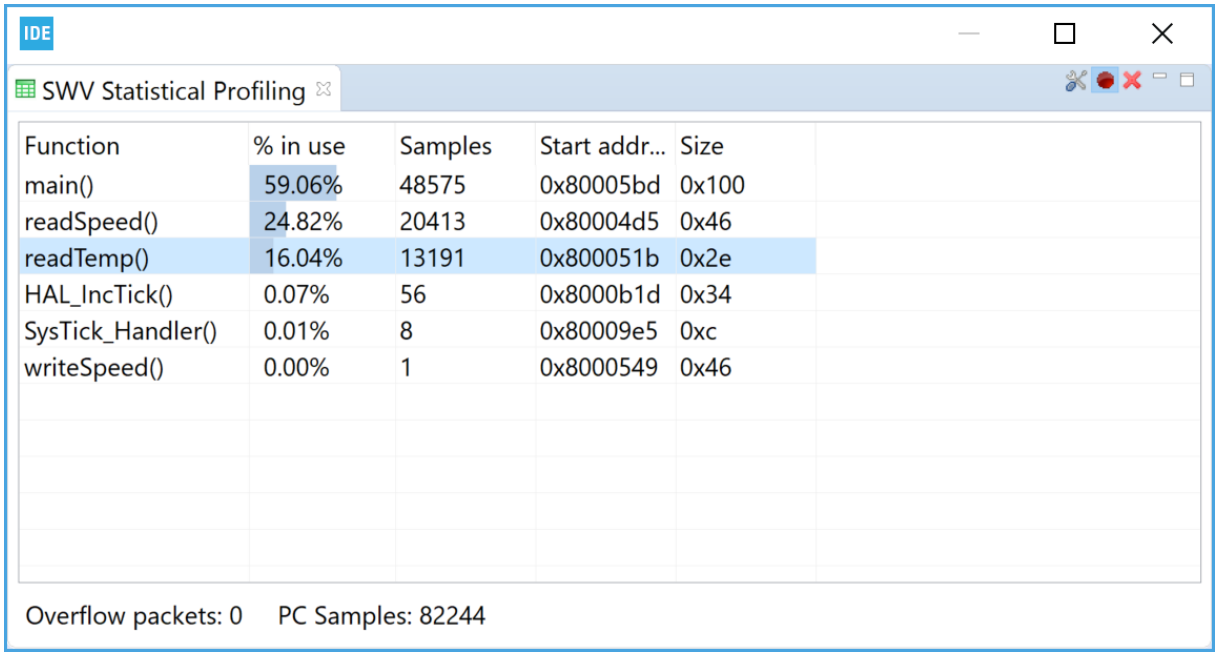
   指定した Core Clock のサイクル間隔で SWV が STM32CubeIDE にプログラム·カウンタの値を送信します。インタフェースがオーバーフローしないように、PC Sampling の Cycles/sample には、大きな値を設定してください。

図 186. SWV PC サンプリングの有効化



2. メニュー WindowShow ViewSWV Statistical Profiling を選択して、[SWV Statistical Profiling]ビューを開きます。まだデータが収集されていないため、空のビューが表示されます。
3. 赤色の Start/Stop Trace ボタンをクリックして設定をボードに送信します。
4. プログラム·デバッグを再開します。ターゲット·システムでコードが実行されると SWV を介して、STM32CubeIDE が関数の使用状況に関する統計情報の収集を開始します。
5. デバッグをサスペンド(一時停止)します。ビューに収集されたデータが表示されます。デバッグ·セッションが長くなるほど、収集される統計情報も増えます。

図 187. SWV 統計プロファイリング



注 [SWV Statistical Profiling]ビューの関数の行をダブルクリックすると、その関数を含むファイルがエディタで開かれます。
[SWV Statistical Profiling]ビューの列に関する情報を、表 10 に示します。

表 10. SWV 統計プロファイリング - 列の詳細

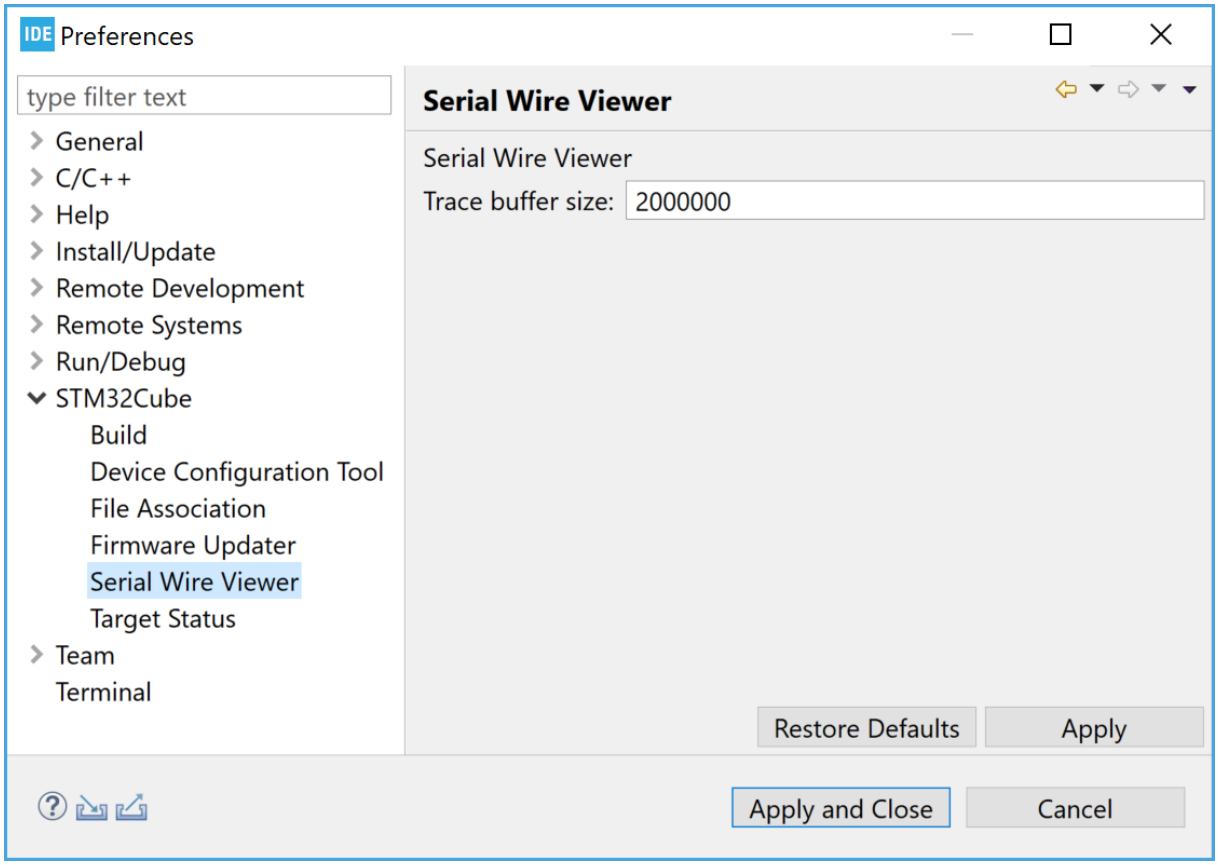
列名	説明
Function	SWV パケット 内のアドレス情報と、プログラムの `elf` ファイルのシンボル情報を比較することで、計算されている関数の名前を表示します。
% in use	関数が使用した時間の割合の計算値(%)。
Samples	関数から受信したサンプルの数。
Start address	関数の開始アドレス。
Size	関数のサイズ。

4.4 SWV トレース·バッファ·サイズの変更

受信される SWV パケット は、シリアル·ワイヤ·ビューア トレース·バッファに保存されます。このバッファのデフォルトの最大サイズは 2 000 000 パケット です。より多くの パケット をトレースするには、この数値を大きくする必要があります。

メニュー WindowsPreferences を選択します。[Preferences]ダイアログの STM32CubeSerial Wire Viewer を選択します。必要に応じて Trace buffer size の設定を変更します。

図 188. SWV のプレファレンス



バッファはヒープに格納されます。割り当てられているヒープを表示するには、まずメニューの WindowsPreferences を選択します。[Preferences]ダイアログの General を選択します。Show heap status を有効にすると、現在のヒープおよび割り当てられているメモリが STM32CubeIDE の右下隅に表示されます。STM32CubeIDE が割り当てることができるメモリ量には上限があります。この上限を大きくして、デバッグ·セッション中に保存する情報量を増やすことができます。

メモリの上限を変更するには、次の手順を実行します。

1. STM32CubeIDE のインストール·ディレクトリに移動します。IDE が保存されているフォルダを開きます。
2. `stm32cubeide.ini` ファイルを編集して `-Xmx1024m` パラメータを必要なサイズ(MB 単位)に変更します。
3. ファイルを保存して STM32CubeIDE を再起動します。

4.5 SWV の一般的な問題

SWV トレースの初心者は、次のような問題で苦労する場合があります。

- 現在使用中のデバッグ設定で SWV が有効化されていない。
- SWV トレースが開始されていない。いずれかの SWV ビューのツールバーにある赤色の Start/Stop Trace ボタンをクリックして、SWV を有効化し、SWV 設定をターゲット·ボードに送信する必要があります。その後、プログラムを起動して SWV データを受信します。SWV ビューの中には、その後プログラムを再度停止して、受信した SWV 情報を視覚化する必要があるものもあります。
- SWO が過剰なデータを受信している。トレースで有効化しているデータ量を減らしてください。
- JTAG プローブ、GDB サーバ、ターゲット·ボード、その他の部品が SWV に対応していない。
- ターゲットの Core Clock が正しく設定されていない。適切な Core Clock を選択することは、非常に重要です。

  ターゲットの Core Clock 周波数が不明の場合は、次のような手順で知ることができる場合があります。まず、プログラム·ループ内にブレークポイントを設定し、そこに達したときに[Expressions]ビューを開きます。

  Add new expression をクリックして、`SystemCoreClock` と入力して Enter キーを押します。CMSIS 規格により、このグローバル変数には、ソフトウェアによって適切な Core Clock 速度が設定されているはずです。

  CMSIS の標準ライブラリに含まれる `SystemCoreClockUpdate()` という名前の関数を `main()` に含めることで、`SystemCoreClock` 変数を設定します。[Variable]ビューにより、この変数を追跡します。

注 実行時にソフトウェアが CPU クロック速度を動的に変化させると、SWV が停止する場合があります。クロック周波数が実行中に突然誤った値になるからです。

すべてのデータを確実に受信できるように、次の手順を実行します。

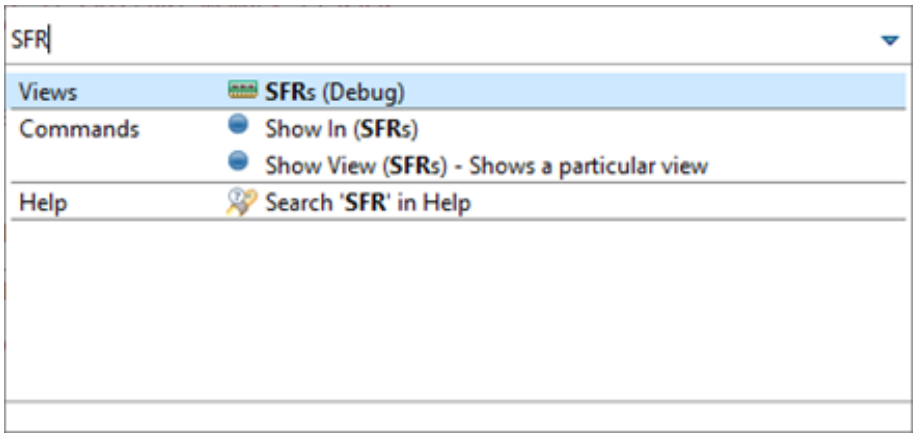
1. SWV の設定を開きます。PC Sampling と Timestamps を除く、すべてのトレースを無効にします。Resolution を可能な最大の値に設定します。
2. 保存して、[SWV Trace Log]ビューを開きます。
3. トレースを開始します。
4. 受信した パケット のすべてが[SWV Trace Log]ビューに表示されることを確認します。

5 特殊機能レジスタ(SFR)

5.1 SFR の概要

特殊機能レジスタ(SFR)は、[SFRs]ビューによって表示、アクセス、編集できます。このビューには、現在のプロジェクトの情報が表示されます。別のプロジェクトを選択すると、ビューの内容は変化します。メニューから、このビューを開くには、メニュー·コマンド WindowShow ViewSFRs を選択するか、クイック·アクセス·フィールドで SFR を検索して、表示される結果からビューを選択します。

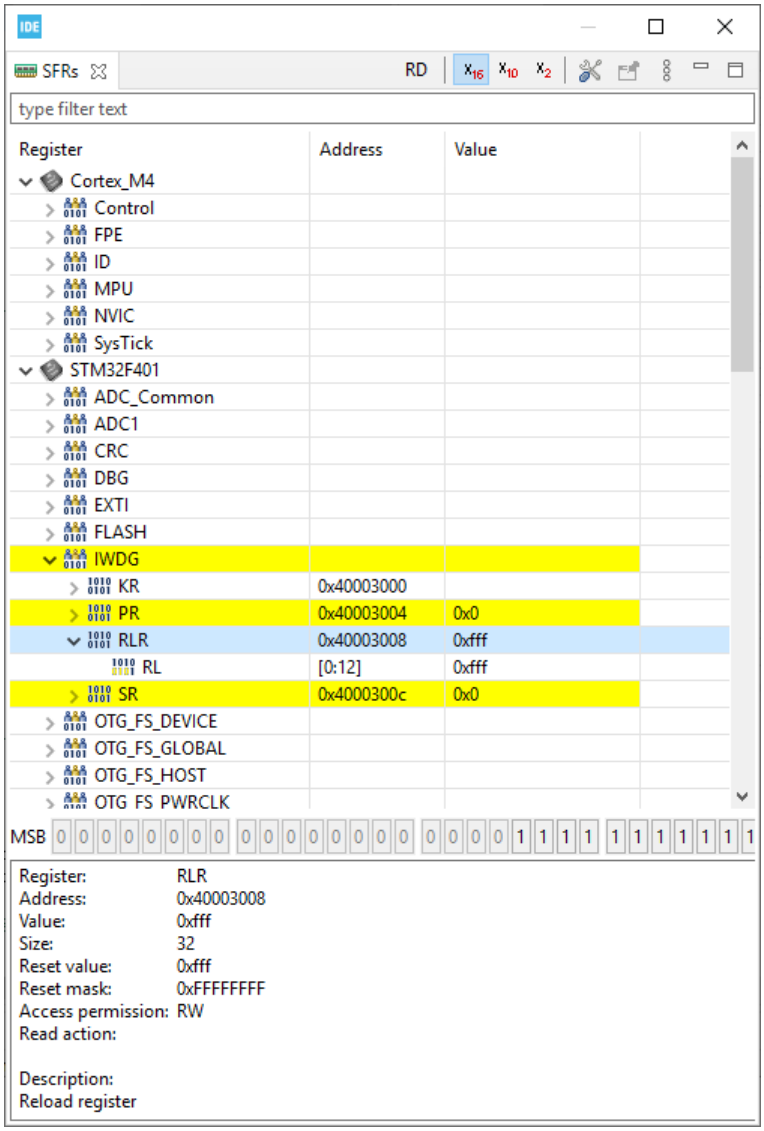
図 189. クイック·アクセス·フィールドを使用して[SFRs]ビューを開く



5.2 [SFRs]ビューの使用方法

[SFRs]ビューには、プロジェクトで使用されている STM32 デバイスのペリフェラル、レジスタ、ビット·フィールドに関する情報が含まれます。プロジェクトをデバッグする場合、レジスタやビット·フィールドにはターゲットから読み出された値が設定されます。このビューには 2 つの主要ノード、Cortex®-M ノードと STM32 ノードがあります。Cortex®-M ノードには、Cortex®-M コアに関する一般的な情報、STM32 ノードには、STM32 デバイス固有のペリフェラルが表示されます。

図 190. [SFRs]ビュー



[SFRs]ビューの上部には、ペリフェラル、レジスタ、ビット･フィールドなど、表示されるノードを絞り込むための検索フィールドがあります。検索フィールドにテキストを入力するのに合わせて、それを含むノードのみに表示が絞り込まれていきます。

[SFRs]ビュー下部には、選択した行に関する詳細情報が表示されます。レジスタやビット･フィールドの場合、Access permission や Read action などの情報です。

Access permission には、次のような詳細情報が含まれます。

- RO(読出し専用)
- WO(書込み専用)
- RW(読出し / 書込み)
- W1(1 回書込み)
- RW1(読出し /1 回書込み)

[Read action] には、レジスタまたはビット･フィールドの読出し時に読出し動作が存在する場合にのみ、次の情報が表示されます。

- clear
- set
- modify
- modifyExternal

ツールバー·ボタンは、[SFRs]ビューの右上隅にあります。

図 191. [SFRs]ビューのツールバー·ボタン

ツールバーの RD ボタンは、選択したレジスタの強制読出しに使用します。この機能は、レジスタまたはレジスタ内の一部ビット·フィールドに SVD ファイルで ReadAction 属性が設定されていても、レジスタを読み出します。

RD ボタンをクリックしてレジスタを読み出すと、ビューに表示されている他のレジスタもすべて再読み出しされ、すべてのレジスタの最新の状態が反映されます。

レジスタを読み出すには、プログラムを停止する必要があります。

基数フォーマット·ボタン(X16、X10、X2)は、レジスタ表示の基数を変更するときに使用します。

Configure SVD settings ボタンは、現在のプロジェクトの[CMSIS-SVD Settings]プロパティ·パネルを開きます。

Pin ボタン(「自動切り換えなし」オプション)を使用すると、[Project Explorer]ビューが他のプロジェクトに切り換わっても、現在表示されている SVD ファイルに対するフォーカスを維持します。

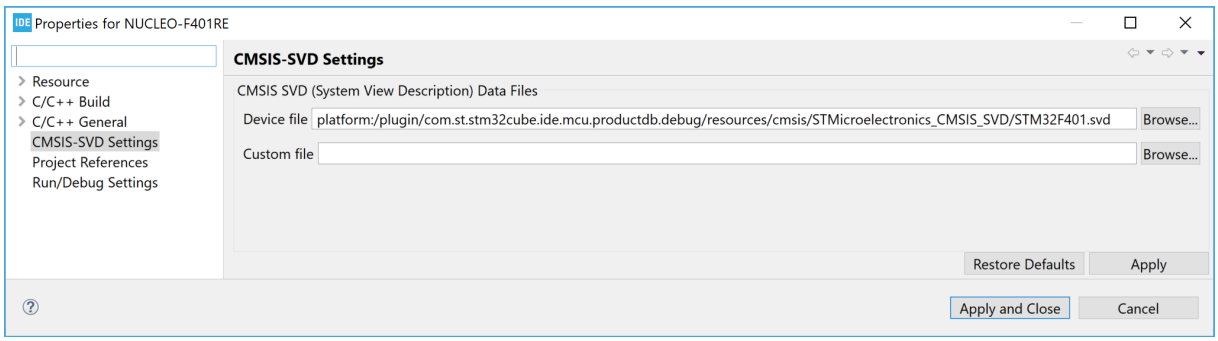
5.3 CMSIS-SVD 設定の変更

プロジェクトの[SFRs]ビューには、そのプロジェクトに対する次の 2 つの CMSIS-SVD(システム·ビューの説明)ファイルを表示できます。

- STM32CubeIDE がデフォルトで選択するファイルは、プロジェクトで選択されているデバイスの SVD ファイルです。
- もう一方は、固有のユーザ·ハードウェア構成を視覚化するために作成したカスタム SVD ファイルです。

設定を変更するには、[SFRs]ビューの Configure SVD settings ツールバー·ボタンを使用して[CMSIS-SVD Settings]プロパティを開きます。

図 192. [SFRs]ビュー - [CMSIS-SVD Settings]プロパティ



すべての SVD ファイルは、CMSIS-SVD 仕様に概説されている構文に準拠している必要があります。この仕様は、Arm® のウェブサイトより入手できます。この要件を満たしていないと、[SFRs]ビューにレジスタ情報が一切表示されない可能性が高まります。

Device file フィールドを システム·ビューの説明(SVD)ファイルの指定に使用します。このファイルで、デバイスのすべてを記述する必要があります。他のビューも、このフィールドで指定した SVD ファイルから情報を取得する可能性があるため、ここでは STM32 デバイスを完全に記述した SVD ファイルにのみこのフィールドを使用することを推奨します。更新された SVD ファイルは、ST マイクロエレクトロニクスより入手できます(ST マイクロエレクトロニクスのウェブサイト(www.st.com)のデバイス説明のセクションで HW Model、CAD Libraries、SVD の列を参照してください)。

Custom file フィールドを使用すると、カスタム·ハードウェアに関連する特殊機能レジスタを定義できるので、各種レジスタの状態表示が簡素化されます。このフィールドには、Device file の内容の一部を抜き出した SFR の「お気に入り」ファイルを作成して指定するという使い方もあります。例えば、頻繁に確認するレジスタのみを含むファイルなどが考えられます。Custom file を指定すると、[SFRs]ビューに新しい最上位ノードが作成されます。ここには、Custom file に関連するレジスタの情報が表示されます。

どちらのフィールドもユーザによる変更が可能です。両フィールドを同時に使用できます。

注

- レジスタとビット·フィールドには、それらに対する書込みアクセス権限がある場合、[Value]列に新しい値を書き込むことができます。
- ST-LINK GDB サーバ を使用していれば、ターゲットの動作中に[SFRs]ビューを使用できます。ただし、その場合はデバッグ設定で Live expression のオプションを有効にしておく必要があります。
- OpenOCD または SEGGER J-Link を使用している場合は、ターゲットの動作中に[SFRs]ビューを使用することができません。
- [SFRs]ビューは、C/C++ 編集パースペクティブでも有用ですが、この場合、レジスタの名前とアドレスのみが表示されます。

6 RTOS 認識デバッグ

リアルタイム·オペレーティング·システム（RTOS）では、スレッド、セマフォ、タイマなど、さまざまな種類のオブジェクトが設計に追加されます。STM32CubeIDE は、Microsoft 社® Azure® RTOS ThreadX および FreeRTOS™ カーネル·オブジェクトを取り扱うための一連の専用ビューを備えています。

これらのビューは、ステップ動作でコード内を移動する場合や、デバッグ·セッションにおいてプログラムがブレークポイントに到達したときに RTOS オブジェクトの状態を視覚化します。

注 FreeRTOS は、米国およびその他の国々にある Amazon 社の商標です。

その他の商標は、それぞれの所有者に帰属します。

6.1 Azure® RTOS ThreadX

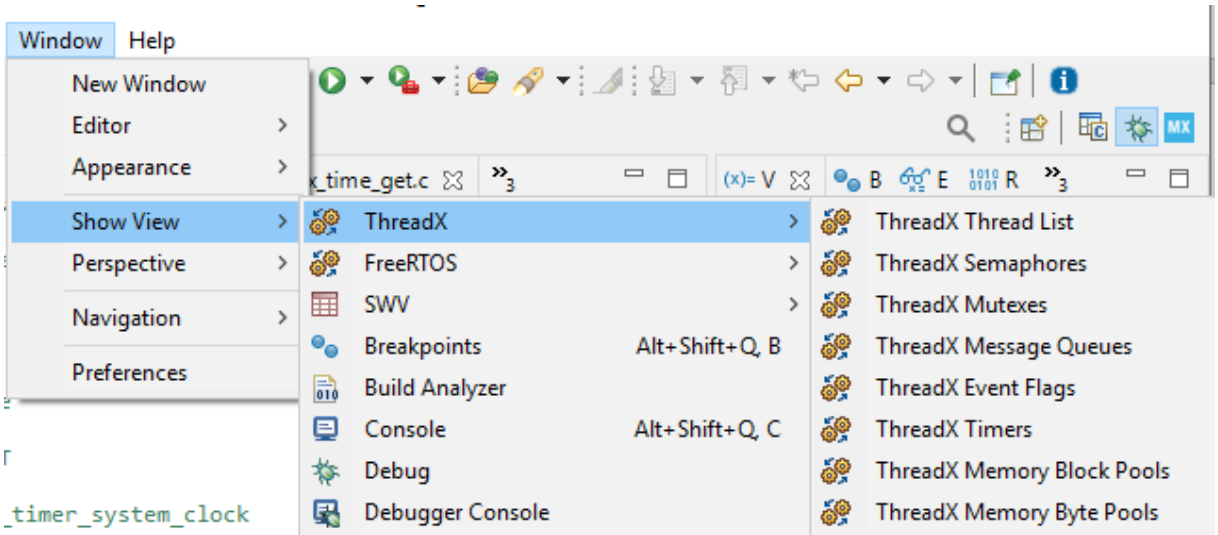
ThreadX については、次のビューが用意されています。

- ThreadX Thread List
- ThreadX Semaphores
- ThreadX Mutexes
- ThreadX Message Queues
- ThreadX Event Flags
- ThreadX Timers
- ThreadX Memory Block Pools
- ThreadX Memory Byte Pools

6.1.1 ビューの検索

デバッガ·パースペクティブでは ThreadX 関連ビューをメニューから開きます。メニュー·コマンド WindowShow ViewThreadX... を選択するか、クイック·アクセス·フィールドで ThreadX を検索して、表示される結果からビューを選択します。

図 193. メニューから選択可能な ThreadX ビュー

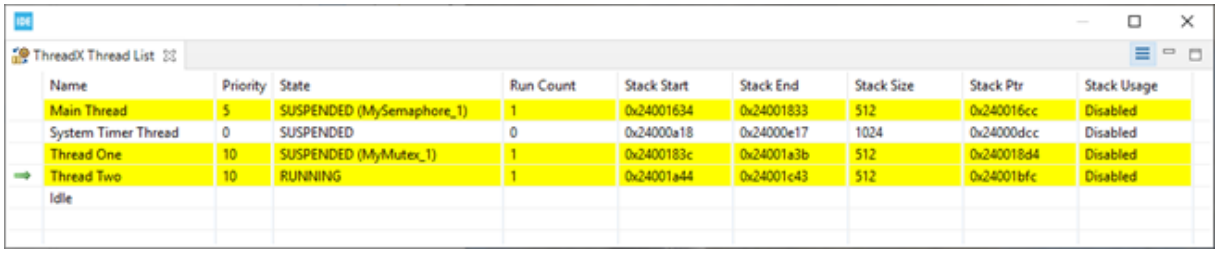


6.1.2 [ThreadX Thread List]ビュー

[ThreadX Thread List]ビューには、ターゲット·システムで利用可能なすべてのスレッドに関する詳細な情報が表示されます。スレッド·リストは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

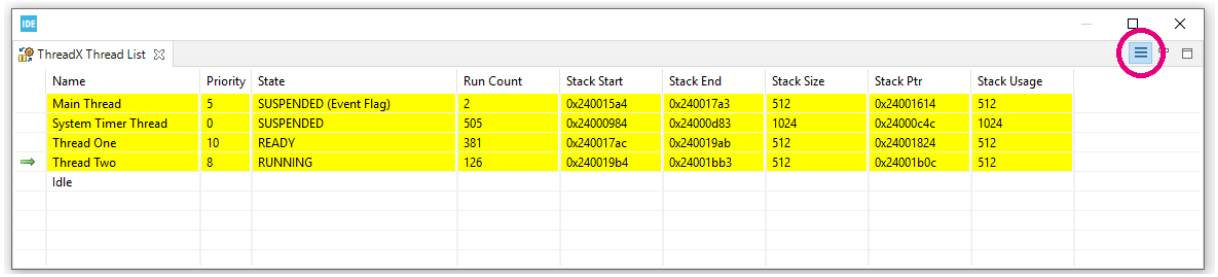
スレッドごとに 1 つの行を割り当て、スレッド·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、スレッドのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 194. [ThreadX Thread List]ビュー(デフォルト)



性能上の理由から、[Stack Usage]列は、デフォルトで無効化されています。スタック解析を有効化するには、[ThreadX Thread List]ビューの Toggle Stack Checking ツールバー·ボタン(図 195 のピンクの円で囲ったボタン)を使用します。

図 195. [ThreadX Thread List]ビュー([Stack Usage]を有効にした場合)



[ThreadX Thread List]ビューの列に関する情報を、表 11 に示します。

表 11. [ThreadX Thread List]ビューの詳細

列名	説明
N/A	緑の矢印記号で現在実行中のスレッドを示します。
Name	スレッドに割り当てられた名前。
Priority	スレッドの優先順位。
State	スレッドの現在の状態。
Run Count	スレッドの実行カウンタ。
Stack Start	スタック領域の開始アドレス。
Stack End	スタック領域の終了アドレス。
Stack Size	スタック領域のサイズ(バイト単位)
Stack Ptr	スタック·ポインタのアドレス。
Stack Usage	スレッド·スタックの最大値(バイト単位)。 ThreadX は、スレッドの作成時にデフォルトでスレッド·スタックのすべてのバイトを `0xEF` のデータ·パターンで埋めます。詳細については、下記の注を参照してください。

注 すべてのスレッドについて、[Stack Usage]列に[Stack Size]列と同じ値が表示される場合、タスク作成時にスレッド·スタックが `0xEF` のデータ·パターンで埋められなかったことが原因である可能性があります。この状況は、スタック·データ·パターンの充填を無効化して ThreadX カーネルをビルドすると発生します。通常、使用する `<tx_user.h>` ファイルには `TX_DISABLE_STACK_FILLING` の定義が含まれます。この定義を下記の例のようにコメントアウトして、プロジェクトを再ビルドすれば問題は解決します。`<tx_user.h>` ファイルには `TX_ENABLE_STACK_CHECKING` 定義も含まれていることを覚えおいてください。この定義を有効化すると、スタック破壊が検出された場合に実行時スタック·チェックが可能になります。その他の情報は、ThreadX のユーザ·ガイドに記載されています。

`TX_ENABLE_STACK_CHECKING` 定義をコメントアウトした ThreadX ヘッダ·ファイル `tx_user.h` の例:

```
/* Determine is stack filling is enabled.By default, ThreadX stack filling is
enabled,
    which places an 0xEF pattern in each byte of each thread's stack.  This is
used by
    debuggers with ThreadX-aw:areness and by the ThreadX run-time stack checking
feature.  */

/* #define TX_DISABLE_STACK_FILLING */

/* Determine whether or not stack checking is enabled. By default, ThreadX stack
checking is
    disabled. When the following is defined, ThreadX thread stack checking is
enabled.  If stack
        checking is enabled (TX_ENABLE_STACK_CHECKING is defined), the
TX_DISABLE_STACK_FILLING
    define is negated, thereby forcing the stack fill which is necessary for the
stack checking
   logic.  */

/*#define TX_ENABLE_STACK_CHECKING*/
```

6.1.3 [ThreadX Semaphores]ビュー

[ThreadX Semaphores]ビューには、ターゲット·システムで利用可能なすべてのリソース·セマフォに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

セマフォごとに 1 つの行を割り当て、セマフォ·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、特定のセマフォのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 196. [ThreadX Semaphores]ビュー

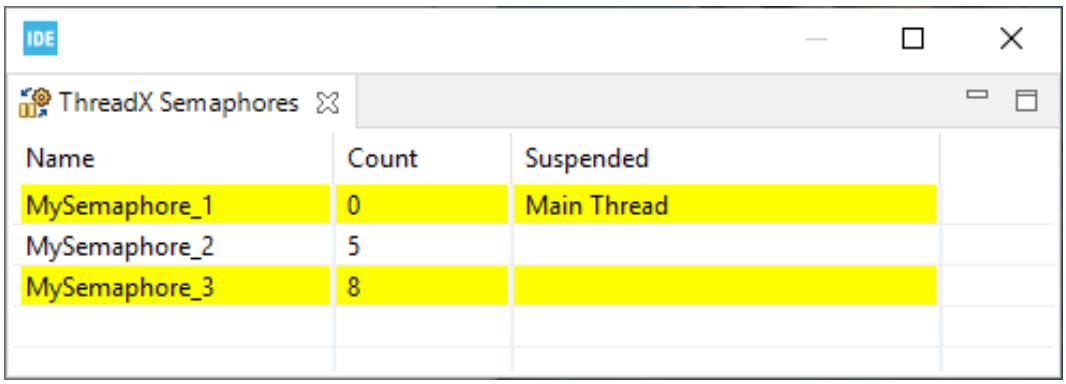


表 12. [ThreadX Semaphores]ビューの詳細

列名	説明
Name	セマフォに割り当てられた名前。
Count	現在のセマフォの数。
Suspended	セマフォの状態が原因で現在サスペンドされているスレッド。

6.1.4 [ThreadX Mutexes]ビュー

[ThreadX Mutexes]ビューには、ターゲット·システムで利用可能なすべてのミューテックスに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

ミューテックスごとに 1 つの行を割り当て、ミューテックス·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、特定のミューテックスのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 197. [ThreadX Mutexes]ビュー

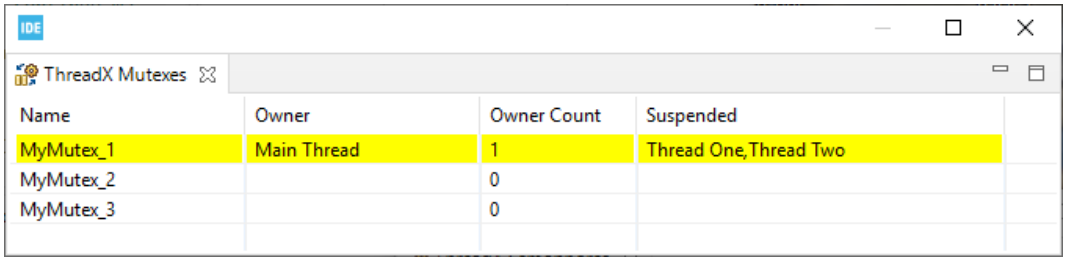


表 13. [ThreadX Mutexes]ビューの詳細

列名	説明
Name	ミューテックスに割り当てられた名前。
Owner	現在ミューテックスを所有しているスレッド。
Owner Count	ミューテックスの所有者の数（所有者スレッドによって実行された get 操作の数）。
Suspended	ミューテックスの状態が原因で現在サスペンドされているスレッド。

6.1.5 [ThreadX Message Queues]ビュー

[ThreadX Message Queues]ビューには、ターゲット・システムで利用可能なすべてのメッセージ・キューに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

メッセージ・キューごとに 1 つの行を割り当て、メッセージ・キュー・パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、特定のメッセージ・キューのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 198. [ThreadX Message Queues]ビュー

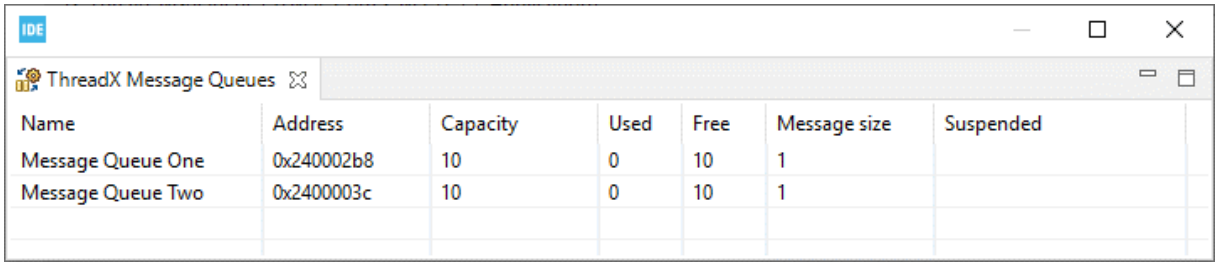


表 14. [ThreadX Message Queues]ビューの詳細

列名	説明
Name	メッセージ・キューに割り当てられた名前。
Address	メッセージ・キューのアドレス。
Capacity	キュー内に保持できるエントリ数の最大値。
Used	現在キュー内で使用されているエントリの数。
Free	現在キュー内で空いているエントリの数。
Message size	メッセージの各エントリのサイズ（32bit ワード単位）。
Suspended	メッセージ・キューの状態が原因で現在サスペンドされているスレッド。

6.1.6 [ThreadX Event Flags]ビュー

[ThreadX Event Flags]ビューには、ターゲット・システムで利用可能なすべてのイベント・フラグ・グループに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

イベント·フラグ·グループごとに 1 つの行を割り当て、パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、特定のイベント·フラグ·グループのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 199. [ThreadX Event Flags]ビュー

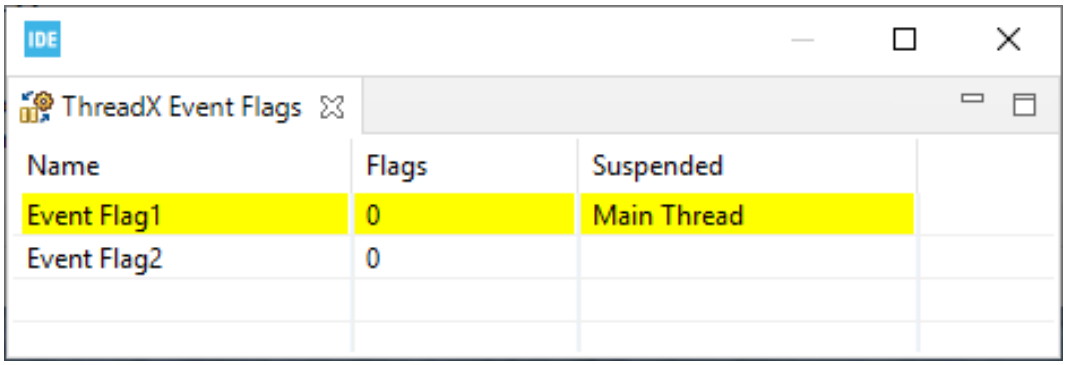


表 15. [ThreadX Event Flags]ビューの詳細

列名	説明
Name	イベント·フラグ·グループに割り当てられた名前。
Flags	イベント·フラグ·グループの現在の値。
Suspended	イベント·フラグ·グループが原因で現在サスペンドされているスレッド。

6.1.7 [ThreadX Timers]ビュー

[ThreadX Timers]ビューには、ターゲット·システムで利用可能なすべてのソフトウェア·タイマに関する詳細な情報が表示されます。タイマのビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

タイマ·パラメータの種類ごとに対応する列があり、タイマごとに 1 つの行を割り当てます。前回デバッガがサスペンドされた後に、特定のタイマのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 200. [ThreadX Timers]ビュー

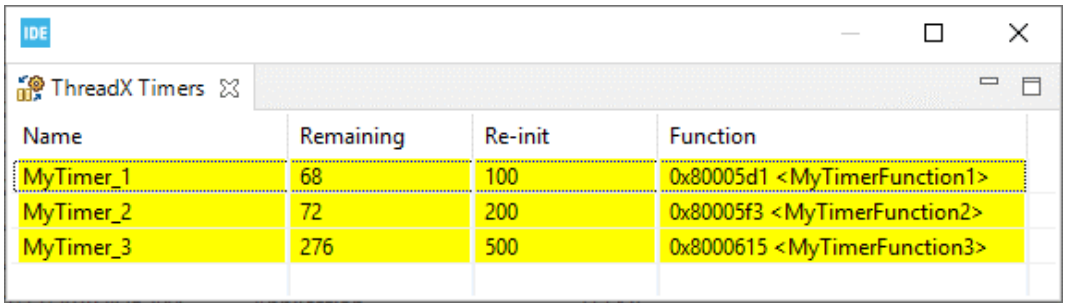


表 16. [ThreadX Timers]ビューの詳細

列名	説明
Name	タイマに割り当てられた名前。
Remaining	タイマが満了するまでの残りティック数。
Re-init	満了後のタイマを再初期化するときの値（ティック単位）。ワンショット·タイマについては、値 0 が表示されます。
Function	タイマの満了時に呼び出される関数のアドレスと名前。

6.1.8 [ThreadX Memory Block Pools]ビュー

[ThreadX Memory Block Pools]ビューには、ターゲット・システムで利用可能なすべてのメモリ・ブロック・プールに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

メモリ・ブロック・プールごとに 1 つの行を割り当て、パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、特定のメモリ・ブロック・プールのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 201. [ThreadX Memory Block Pools]ビュー

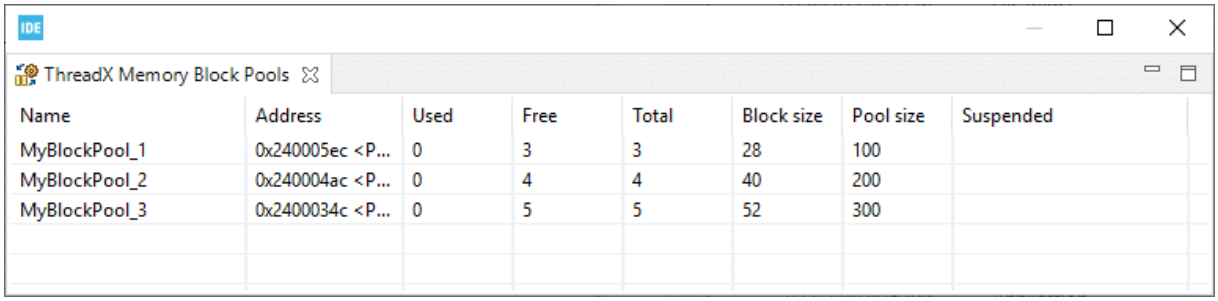


表 17. [ThreadX Memory Block Pools]ビューの詳細

列名	説明
Name	メモリ・ブロック・プールに割り当てられた名前。
Address	メモリ・ブロック・プールの開始アドレス。
Used	現在の割当て済みブロックの数。
Free	現在の空きブロックの数。
Total	利用可能なメモリ・ブロック・プールの合計数。
Block size	各ブロックのサイズ（バイト単位）。
Pool size	プールの合計サイズ（バイト単位）。
Suspended	メモリ・ブロック・プールの状態が原因で現在サスペンドされているスレッド。

6.1.9 [ThreadX Memory Byte Pools]ビュー

[ThreadX Memory Byte Pools]ビューには、ターゲット・システムで利用可能なすべてのメモリ・バイト・プールに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

メモリ・バイト・プールごとに 1 つの行を割り当て、パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、特定のメモリ・バイト・プールのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 202. [ThreadX Memory Byte Pools]ビュー

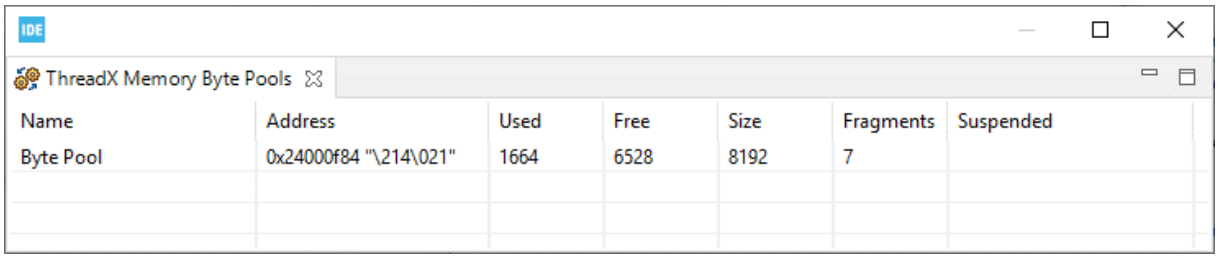


表 18. [ThreadX Memory Byte Pools]ビューの詳細

列名	説明
Name	メモリ・バイト・プールに割り当てられた名前。

列名	説明
Address	メモリ·バイト·プールの開始アドレス。
Used	現在の割当て済みバイトの数。
Free	現在の空きバイトの数。
Size	フラグメントの数。
Fragments	各ブロックのサイズ（バイト単位）。
Suspended	メモリ·バイト·プールの状態が原因で現在サスペンドされているスレッド。

6.2 FreeRTOS™

FreeRTOS™ については、次のビューが用意されています。

- FreeRTOS Task List
- FreeRTOS Timers
- FreeRTOS Semaphores
- FreeRTOS Queues

6.2.1 要件

FreeRTOS™ 関連ビューに RTOS の状態に関する詳細情報を表示するには、FreeRTOS™ カーネルのいくつかのファイルを設定する必要があります。この後のセクションでは、必要な設定の一部を紹介します。詳細は、FreeRTOS リファレンス·マニュアルをご覧ください。

6.2.1.1 トレース情報の有効化

`freeRTOSConfig.h` 内の定義 `configUSE_TRACE_FACILITY` を有効に（'1' に設定）する必要があります。これによって、追加の構造体メンバーおよび関数がビルドに含まれ、[FreeRTOS Task List]ビューにおけるインスタンス·スタック·チェック、[FreeRTOS Semaphores]ビューにおけるセマフォの種類の一覧表示が可能になります。

例：

```
freeRTOSConfig.h
#define configUSE_TRACE_FACILITY                1
```

6.2.1.2 レジストリへの追加

[FreeRTOS Queues]および[FreeRTOS Semaphores]の両ビューにオブジェクトを表示できるようにするには、アプリケーション·ソフトウェアで `vQueueAddToRegistry()` 関数を呼び出す必要があります。この関数は、FreeRTOS™ キュー·レジストリにオブジェクトを追加し、引数として 2 つのパラメータを取ります。一つはキューのハンドル、もう一つは FreeRTOS™ 関連ビューに表示されるキューの説明です。

例：

```
vQueueAddToRegistry(mailId, "osMailQueue");
vQueueAddToRegistry(osQueueHandle, "osQueue");
vQueueAddToRegistry(osSemaphoreHandle, "osSemaphore");
```

6.2.1.3 *RTOS プロファイリング情報*

有効な RTOS ランタイム 統計情報を取得するには、アプリケーションで ランタイム 統計情報のタイムベースを設定する必要があります。タイムベース·クロックは、RTOS のティック割込みの処理に使用するクロック周波数の 10 倍以上高速にすることを推奨します。FreeRTOS™ の ランタイム 統計情報の収集を有効にするには、ファイル `freeRTOSConfig.h` に、以下を含める必要があります。

1. `configGENERATE_RUN_TIME_STATS 1` の定義
2. プロファイリングに使用するタイマを設定する関数を呼び出すための `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` の定義
3. プロファイリング·タイマから現在の値を読み出す関数を呼び出すための `portGET_RUN_TIME_COUNTER_VALUE()` の定義

例 :

```
freeRTOSConfig.h
#define configGENERATE_RUN_TIME_STATS                1
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()  configureRunTime()
#define portGET_RUN_TIME_COUNTER_VALUE()    getRunTimeCounter()
```

または、システムで ランタイム 変数を使用できる場合は、次のようになります。

```
freeRTOSConfig.h
#define configGENERATE_RUN_TIME_STATS                1
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()  (RunTime=0UL)
#define portGET_RUN_TIME_COUNTER_VALUE()    RunTime
```

これら 3 つの定義を設定した後も[FreeRTOS Task List]ビューの[Run Time]列に `N/A` と表示される場合は、プロジェクトが最適化レベル `-O0` でビルドされていないことが問題の原因である可能性があります。原因は、`tasks.c` の `ulTutoralRunTime` の宣言部分で見つかる可能性が大です。

例 :

```
#if ( configGENERATE_RUN_TIME_STATS == 1 )
    PRIVILEGED_DATA static uint32_t ulTaskSwitchedInTime = 0UL;
    /*< Holds the value of a timer/counter the last time a task was switched in.
*/
    PRIVILEGED_DATA static uint32_t ulTotalRunTime = 0UL;
     /*< Holds the total amount of execution time as defined by the run time
counter clock. */
#endif
```

解決策 :

- 変数を volatile として宣言します。

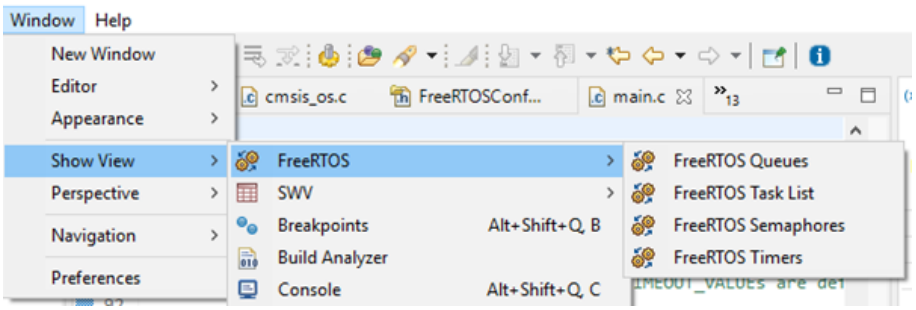
```
PRIVILEGED_DATA volatile static uint32_t ulTotalRunTime = 0UL;
/*< Holds the total amount of execution time as defined by the run time counter
clock. */
```

- または、単に `tasks.c` だけ最適化レベルを変更します。次の手順を実行してください。
  1. [Project Explorer]ビューで、このファイルを右クリックし、[Properties]を選択します。
  2. PropertiesC/C++ BuildSettingsTool SettingsOptimization を選択します。
  3. Optimization Level を `None (-O0)` に設定します。

6.2.2 ビューの検索

デバッガ·パースペクティブでは FreeRTOS™ 関連ビューをメニューから開きます。メニュー·コマンド WindowShow ViewFreeRTOS... を選択するか、クイック·アクセス·フィールドで FreeRTOS を検索して、表示される結果からビューを選択します。

図 203. メニューから選択可能な FreeRTOS™ 関連ビュー

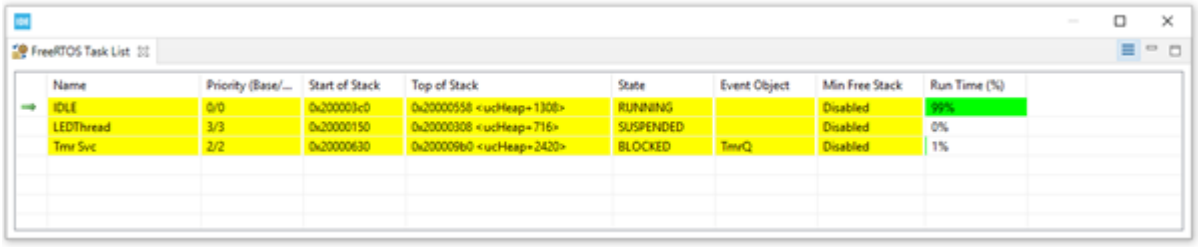


6.2.3 [FreeRTOS Task List]ビュー

[FreeRTOS Task List]ビューには、ターゲット·システムで利用可能なすべてのタスクに関する詳細な情報が表示されます。タスク·リストは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。

タスクごとに 1 つの行を割り当て、タスク·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、タスクのいずれかのパラメータ値が変化した場合、図 204 の例に示したように、対応する行が黄色で強調表示されます。

図 204. [FreeRTOS Task List]ビュー(デフォルト)



性能上の理由から、スタック解析([Min Free Stack]列)は、デフォルトで無効化されています。スタック解析を有効にするには(図 206 参照)、図 205 に示した[FreeRTOS Task List]ビューの[Toggle Stack Checking]ツールバー·ボタンを使用します。

図 205. FreeRTOS™ のスタック·チェックの切り換え

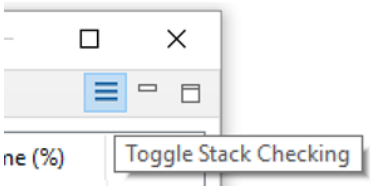


図 206. [FreeRTOS Task List]ビュー([Min Free Stack]を有効にした場合)

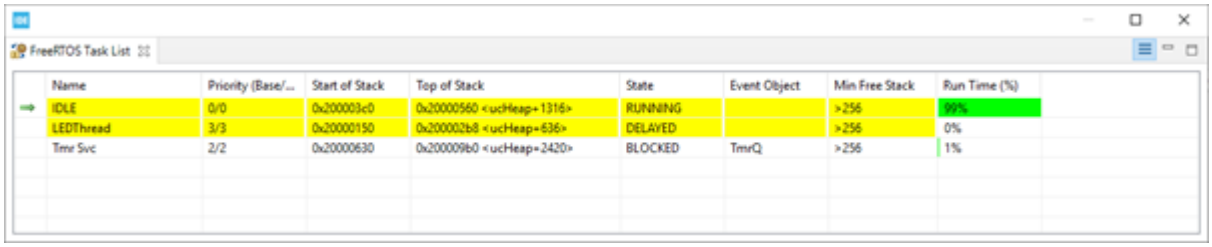
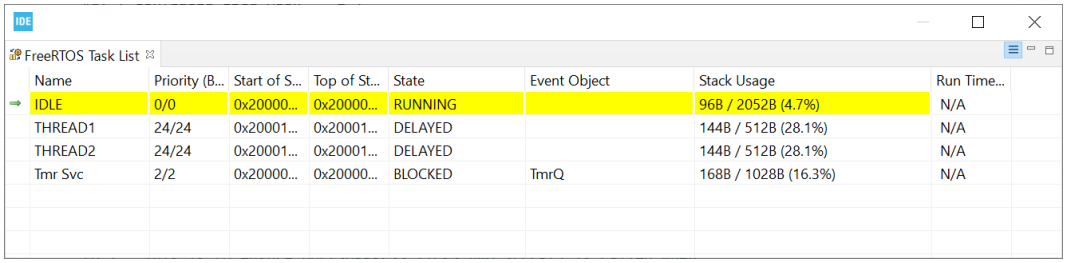


図 206 の[FreeRTOS Task List]ビューでは、Min Free Stack の列に値が表示されています。プロジェクトを、次の定義を設定してビルドすると、この列の情報が[Stack Usage]に切り換わります。

```
#define configRECORD_STACK_HIGH_ADDRESS                1
```

この場合、スタックの使用状況に関する全情報が、図 207 に示すように、使用済み / 総容量（使用割合 %）というフォーマットで表示されます。

図 207. [FreeRTOS Task List]ビュー（ConfigRECORD_STACK_HIGH_ADDRESS を有効にした場合）



[FreeRTOS Task List]ビューの列に関する情報を、表 19 に示します。

表 19. [FreeRTOS Task List]ビューの詳細

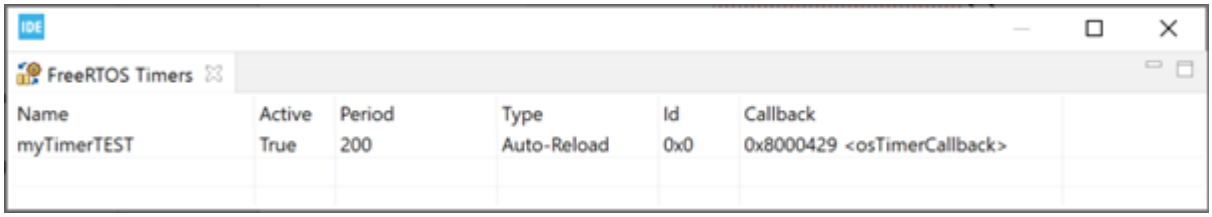
列名	説明
N/A	緑の矢印記号で現在実行中のタスクを示します。
Name	タスクに割り当てられた名前。
Priority (Base/Actual)	タスクのベース優先度と実際の優先度。ベース優先度とは、タスクに割り当てられた優先度です。実際の優先度とは、優先度の継承メカニズムにより一時的にタスクに割り当てられた優先度です。
Start of Stack	タスクに割り当てられたスタック領域のアドレス。
Top of Stack	タスクの保存されているスタック·ポインタのアドレス。
State	タスクの現在の状態。
Event Object	タスクがブロックされる原因となったリソースの名前。
Min Free Stack(1)	スタックの「ハイ·ウォーターマーク」。タスクが使用できる、スタックに残されている最小バイト数が表示されます。値 0 は（高い確率で）、スタック·オーバーフローの発生を示唆しています。 注　この機能は、ビューのツールバーで有効化する必要があります。
Run Time (%)	この ランタイム 統計情報は、タスクがこれまでに費やしてきた時間の割合に関する情報を提供します。これは、開発時のシステム·プロファイリングに使用できます。

1. アプリケーションを `configRECORD_STACK_HIGH_ADDRESS = 1` でビルドした場合、この列名は *"Stack Usage"* に切り換わります。その場合、スタックの使用状況の詳細が使用済み / 総容量（使用割合 %）のフォーマットで表示されます。

6.2.4 [FreeRTOS Timers]ビュー

[FreeRTOS Timers]ビューには、ターゲット·システムで利用可能なすべてのソフトウェア·タイマに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。タイマごとに 1 つの行を割り当て、タイマ·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、タイマのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 208. [FreeRTOS Timers]ビュー



[FreeRTOS Timers]ビューの列に関する情報を、表 20 に示します。

表 20. [FreeRTOS Timers]ビューの詳細

列名	説明
Name	タイマに割り当てられた名前。
Active	アクティブ状態に関する情報。
Period	タイマ起動からコールバック関数の実行までの時間(ティック単位)。
Type	タイマの種類。自動再ロード·タイマは、満了後に自動的に再度アクティブになります。ワンショット·タイマは、1 回だけ満了して終了します。
Id	タイマの識別子。
Callback	タイマの満了時に実行されるコールバック関数のアドレスと名前。

注
1. [Name]フィールドに名前が表示されない場合は、そのタイマが名前付きで作成されているかを確認してください。`xTimerCreate()` を呼び出すときの最初のパラメータにタイマ名の文字列を含める必要があります。
2. ソフトウェア·タイマを使用する場合、`Tmr Svc` タスクと `TmrQ` キューが自動的に作成されます。これらのオブジェクトは、[FreeRTOS Task List]ビューと[FreeRTOS Queues]ビューに表示されます。

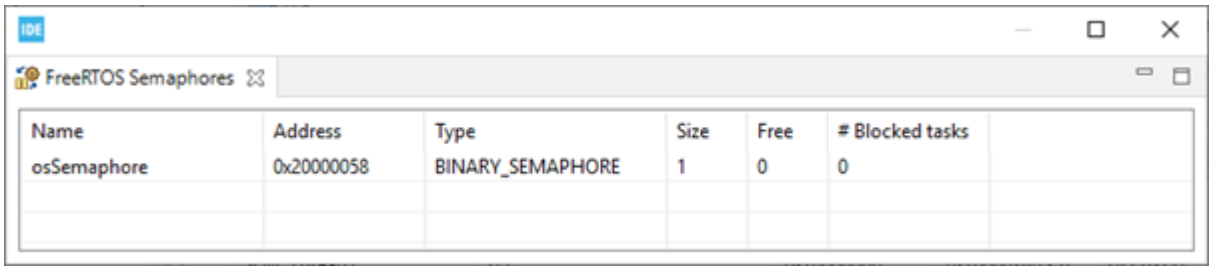
6.2.5 [FreeRTOS Semaphores]ビュー

[FreeRTOS Semaphores]ビューには、ターゲット·システムで利用可能なすべての同期オブジェクトに関する詳細な情報が表示されます。同期オブジェクトには、次のようなものがあります。

- ミューテックス
- 計数セマフォ
- バイナリ·セマフォ
- 再帰セマフォ

ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。セマフォごとに 1 つの行を割り当て、セマフォ·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、セマフォのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 209. [FreeRTOS Semaphores]ビュー



注 [Type]情報に `N/A` と表示される場合、ファイル `FreeRTOSconfig.h` で定義 `configUSE_TRACE_FACILITY` が有効化されていることを確認してください。

[FreeRTOS Semaphores]ビューの列に関する情報を、表 21 に示します。

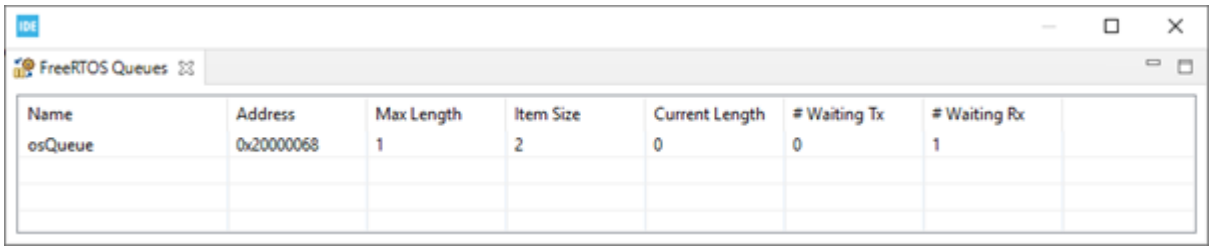
表 21. [FreeRTOS Semaphores]ビューの詳細

列名	説明
Name	セマフォに割り当てられた名前。
Address	オブジェクトのアドレス。
Type	オブジェクトの種類。
Size	所有タスクの最大数。
Free	現在使用可能な空きスロットの数。
#Blocked tasks	オブジェクト待機中で現在ブロックされているタスクの数。

6.2.6 [FreeRTOS Queues]ビュー

[FreeRTOS Queues]ビューには、ターゲット·システムで利用可能なすべてのキューに関する詳細な情報が表示されます。ビューは、ターゲットの実行がサスペンドされるたびに自動的に更新されます。キューごとに 1 つの行を割り当て、キュー·パラメータの種類ごとに対応する列があります。前回デバッガがサスペンドされた後に、キューのいずれかのパラメータ値が変化した場合、対応する行が黄色で強調表示されます。

図 210. [FreeRTOS Queues]ビュー



[FreeRTOS Queues]ビューの列に関する情報を、表 22 に示します。

表 22. [FreeRTOS Queues]ビューの詳細

列名	説明
Name	キュー·レジストリのキューに割り当てられた名前。
Address	キューのアドレス。
Max Length	キューが保持できるアイテムの最大数。
Item Size	各キュー·アイテムのサイズ(バイト単位)。
Current Length	キュー内に現在保持されているアイテムの数。
#Waiting Tx	キューへの送信待機中で現在ブロックされているタスクの数。
#Waiting Rx	キューからの受信待機中で現在ブロックされているタスクの数。

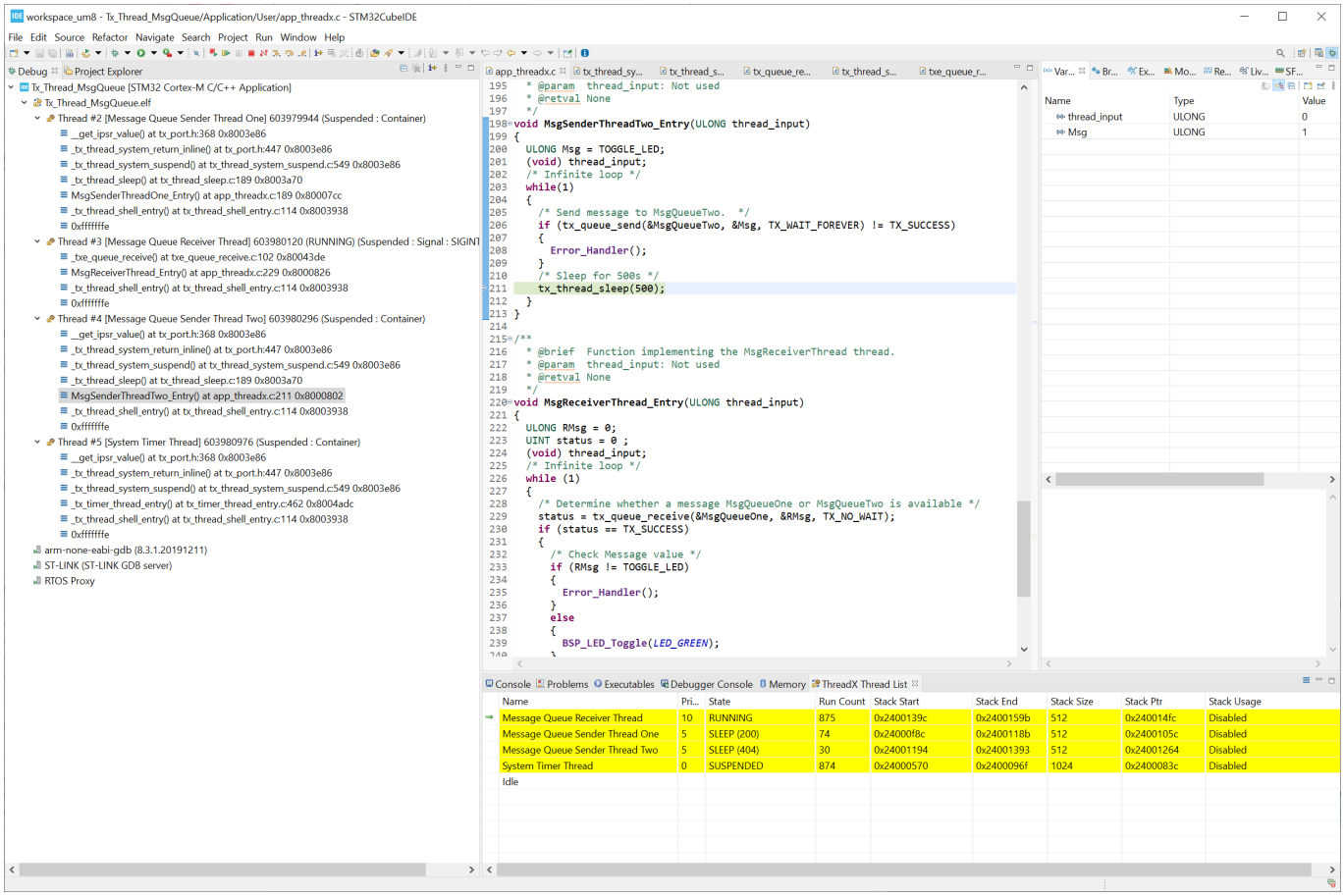
6.3 RTOS カーネル認識デバッグ

STM32CubeIDE の RTOS カーネル認識デバッグは、RTOS プロキシを使用する Microsoft 社® Azure® RTOS ThreadX と FreeRTOS™ オペレーティング·システムに対応しています。RTOS プロキシは STM32CubeIDE に付属しており、ST-LINK GDB サーバ、OpenOCD、SEGGER J-Link GDB サーバ とともに使用できます。

RTOS カーネル認識デバッグを有効化してデバッグ·セッションを開始すると、[Debug]ビューにすべてのスレッドが一覧表示されます。[Debug]ビューのスレッドを選択すると、そのスレッドの現在のコンテキストがさまざまなビューに表示されます。例えば、[Variables]、[Registers]、[Editor]ビューに、アクティブなスタック·フレームの情報が反映されます。

図 211 にデバッグ·セッションを示します。[ThreadX Thread List]ビューを見ると、Message Queue Receiver Thread が実行中(RUNNING 状態)であることがわかります。この状態は、[Debug]ビューでも確認できます。[Debug]ビューでは、`MsgSenderThreadTwo_Entry` 関数が選択され、エディタ領域を見ると、スレッドが 500 ms の期間、スリープ状態で待機していることがわかります。

図 211. RTOS カーネル認識デバッグ



RTOS カーネル認識デバッグを有効にするには、[Debug Configurations]ダイアログの[Debugger]タブで、RTOS のプロキシ、ドライバ(RTOS ThreadX または FreeRTOS™)とポート(Cortex® コア)を有効化し、プロキシで使用するポート番号を設定します。

[RTOS]タブでも、[Driver settings]オプションとして[Driver](ThreadX または FreeRTOS)と使用するポートを選択できます。ドライバの Auto-detect(自動検出)オプションは、まだ実験段階の機能です。

図 212. RTOS カーネル認識デバッグの設定

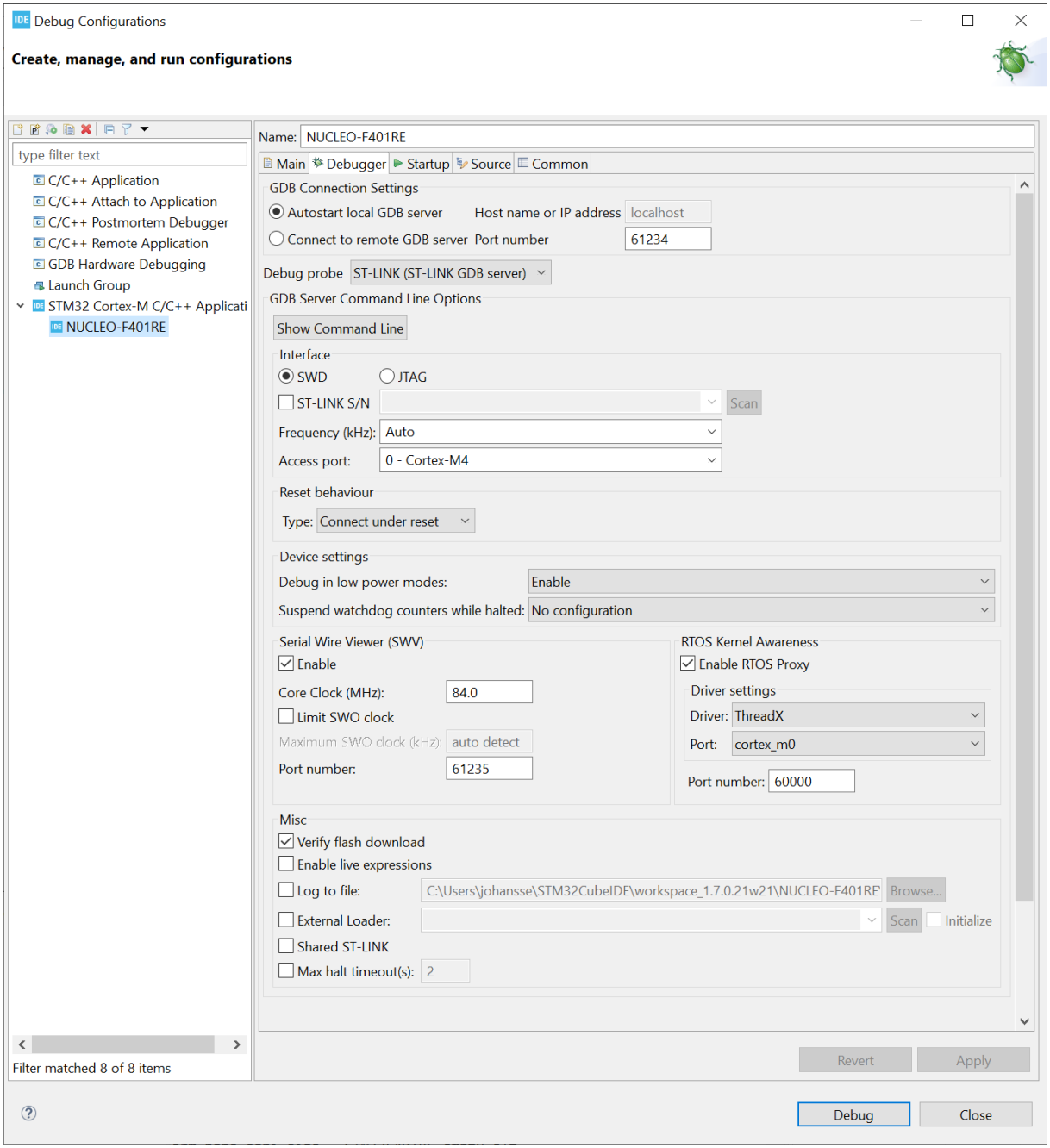
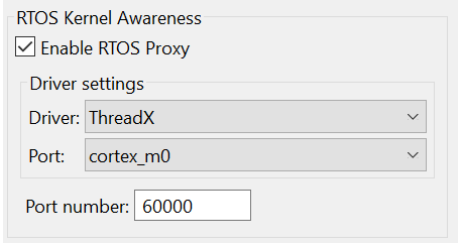


図 213. ThreadX カーネル認識デバッグの設定



ポートのオプションには、サポート対象のコアが一覧表示されます。表示される項目は、図 214 と 図 215 に示すように、選択した RTOS ドライバによって異なります。

図 214. ThreadX ポートの設定

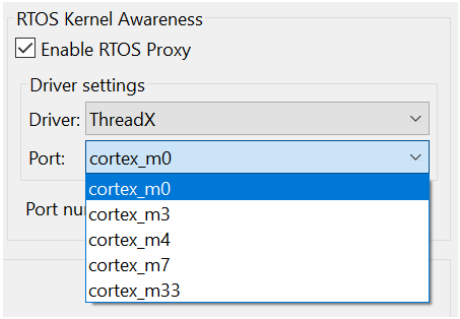
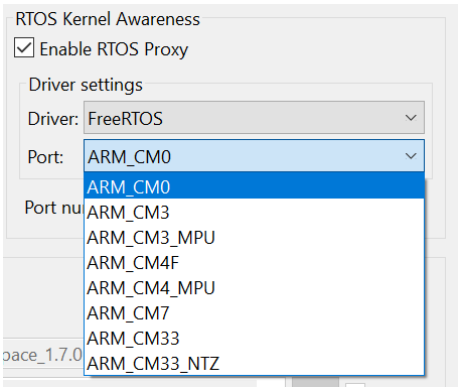


図 215. FreeRTOS™ ポートの設定



既知の制約

- ST-LINK GDB サーバ で使用する場合、ライブ式の機能を無効化する必要があります。
- スワップ·アウトされたスレッドの[Registers]ビューの内容が、一部のレジスタについては、アクティブな CPU コンテキストと混在して表示されます(コンテキスト切り換え時に、すべてのレジスタが保存されるわけではありません)。
- [Registers]ビューでは、浮動小数点レジスタが正しく更新されません。

7 Fault Analyzer

7.1 Fault Analyzer の概要

STM32CubeIDE の Fault Analyzer 機能では、Cortex®-M のネスト化されたベクタ割込みコントローラ（NVIC）から抽出された情報を解釈し、フォルト発生の原因を特定します。この情報は、[Fault Analyzer]ビューで視覚化されます。CPU がアプリケーション・ソフトウェアによってフォルト発生の状態に陥る、発見が困難なシステム・フォルトを特定および解決するのに有効です。

フォルトには、次のような状態があります。

- 無効なメモリ位置へのアクセス
- 境界がアラインされていないメモリ位置へのアクセス
- 未定義命令の実行
- ゼロによる除算

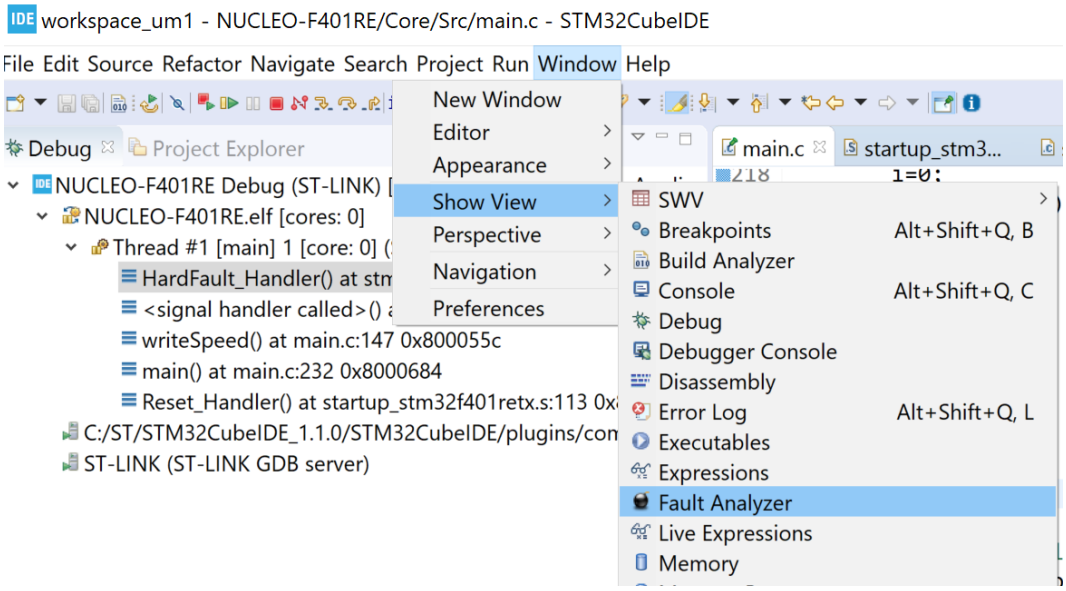
フォルトが発生すると、そのコード行がデバッガに表示されます。ビューには、エラー状態となった理由が表示されます。フォルトは、ハード・フォルト、バス・フォルト、用法フォルト、メモリ・フォルトに大別されます。

- ハード・フォルトとバス・フォルトは、バスを介してペリフェラル・レジスタまたはメモリ位置のいずれかに無効なアクセスを試みると発生します。
- 用法フォルトは、不正な命令、その他のプログラム・エラーの結果、発生します。
- メモリ・フォルトは、不正な位置にアクセスを試みた場合や、メモリ保護ユニット（MPU）が管理するルールに違反した場合などに発生します。

フォルト分析をさらに進めるために、例外スタック・フレームを視覚化するオプションでは、クラッシュ発生時におけるマイクロコントローラのレジスタ値のスナップショットが得られます。フォルトを個々の命令にまで分離することで、フォルトにつながる命令が実行された時点でのマイクロコントローラの状態を再現できます。

デバッガ・パースペクティブで、メニューから[Fault Analyzer]ビューを開きます。メニュー・コマンド WindowShow ViewFault Analyzer を選択するか、クイック・アクセス・フィールドで Fault Analyzer を検索して、表示される結果からビューを選択します。

図 216. [Fault Analyzer]ビューを開く



7.2 [Fault Analyzer]ビューの使用方法

[Fault Analyzer]ビューは、主に 5 つのセクションから構成され、これらは展開と折りたたみが可能です。各セクションには特定のフォルトの発生原因を、より的確に把握するための各種情報が含まれます。5 つのセクションとは、次のとおりです。

- Hard Fault Details（ハード・フォルトの詳細）
- Bus Fault Details（バス・フォルトの詳細）

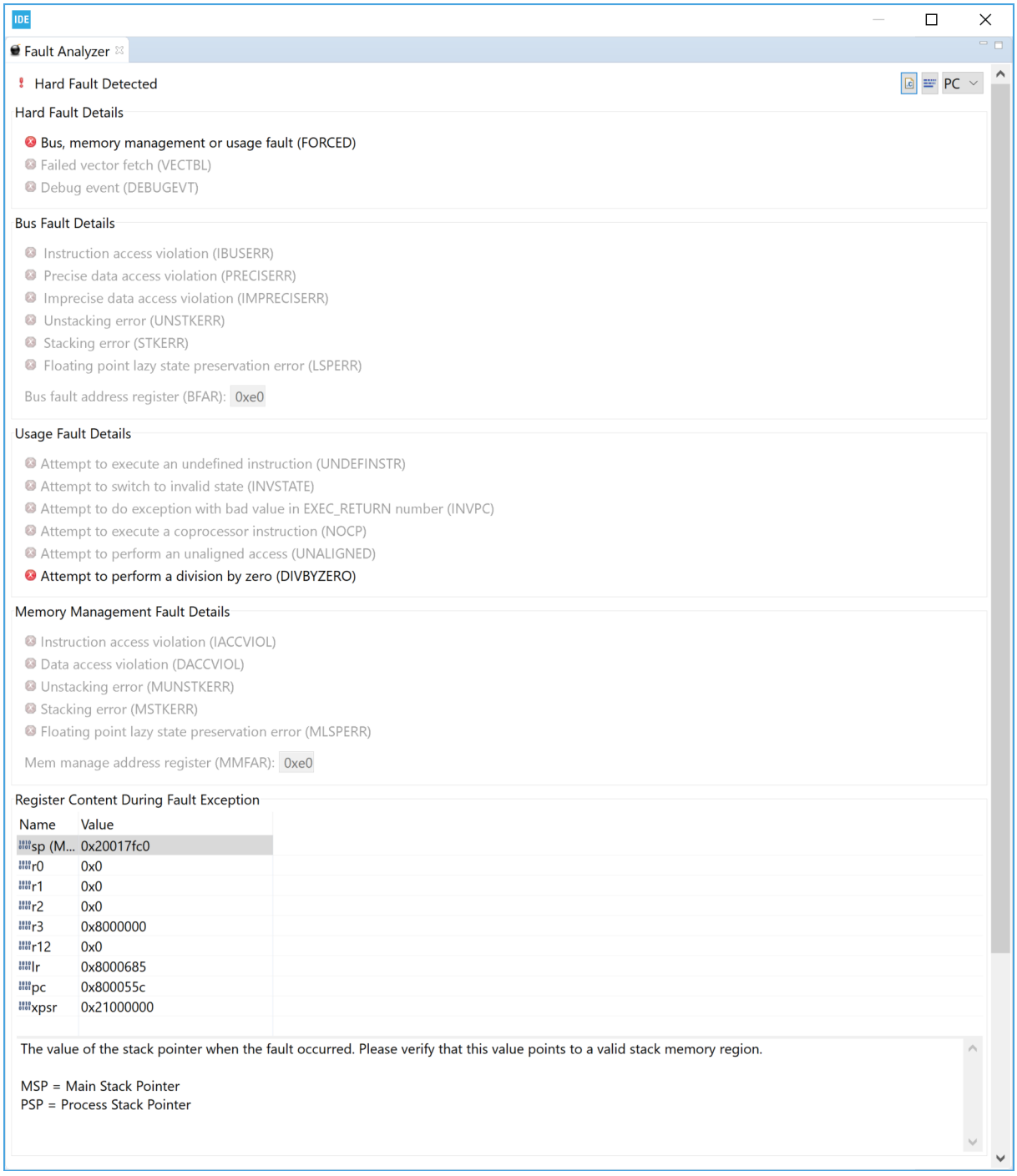
- Usage Fault Details（用法フォルトの詳細）
- Memory Management Fault Details（メモリ管理フォルトの詳細）
- Register Content During Fault Exception（フォルト例外時のレジスタの内容）

フォルトが発生した場合、ビューのボタンをクリックして、Open editor on fault location（フォルト位置をエディタで開く）か、Open disassembly on fault location（フォルト位置をディスアセンブラで開く）ことが可能です。

図 217 に、エラーが検出されたときの[Fault Analyzer]ビューの例を示します。この例では、プロジェクトがゼロによる除算を実行したことでエラーが発生し、デバッガは `HardFault_Handler()` で停止しています。

エラー発生時に[Fault Analyzer]ビューを開くと、エラーの理由が表示されます。この例では、Usage Fault Detected（用法フォルト検出）と Attempt to perform a division by zero (DIVBYZERO)（ゼロによる除算）を表示しています。Register Content During Fault Exception は、問題が発生したときのレジスタ値です。

図 217. [Fault Analyzer]ビュー



[Fault Analyzer]ビューには、次のツールバー･ボタンがあります。

図 218. [Fault Analyzer] のツールバー

- 最初のツールバー･ボタン（左）は、フォルト位置の戻りアドレスを[Editor]ビューで開きます。フォルト位置は、スタック内の PC および LR レジスタの情報と、デバッグ対象の `elf` ファイル内のシンボル情報に基づいて判断します。

- 2 番目のツールバー·ボタン(中央)は、フォルト位置の戻りアドレスを[Disassembly]ビューで開きます。フォルト位置は、スタック内の PC および LR レジスタの情報と、デバッグ対象の `elf` ファイル内のシンボル情報に基づいて判断します。
- 3 番目のツールバー·ボタン(右)は、エラー位置を[Editor]または[Disassembly]ビューで開くときに、PC と LR のどちらのレジスタを使用するかを選択します。

図 219 と 図 220 に、この例のフォルト位置を見つけるために、ツールバー·ボタンを使用して[Editor]および[Disassembly]ビューを開いた画面を示します。

図 219. フォルト発生時に Fault Analyzer から開いた[Editor]ビュー

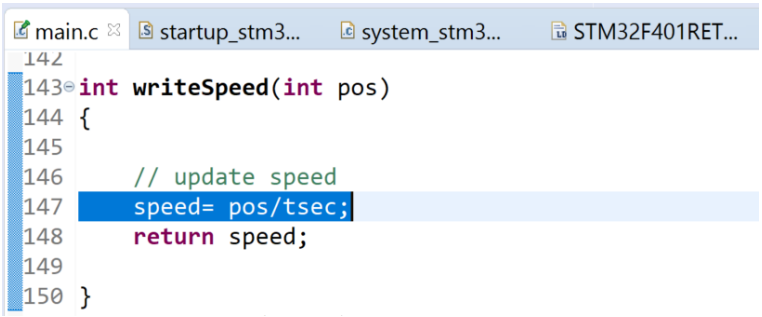
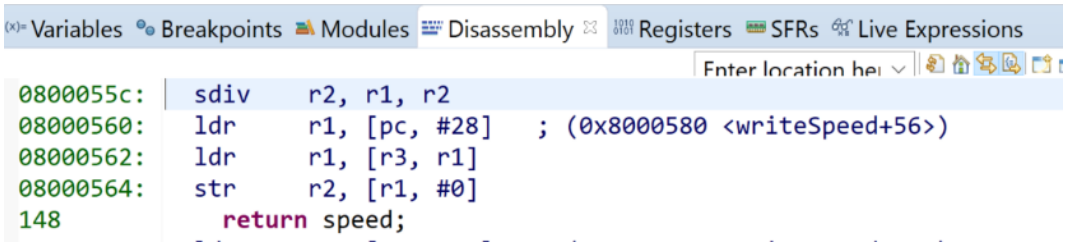


図 220. フォルト発生時に Fault Analyzer から開いた[Disassembly]ビュー



注 Fault Analyzer は、すべての STM32 プロジェクトに使用できます。特別なコードや特別なビルド設定は一切不要です。すべてのデータは、Cortex®-M のレジスタから収集されます。シンボル情報は、デバッグ対象の `elf` ファイルから読み出します。

8 Build Analyzer

8.1 Build Analyzer の概要

STM32CubeIDE の Build Analyzer 機能では、elf ファイルに含まれるプログラム情報を詳細に解釈して、それらをビューに表示します。elf ファイルと同じフォルダに、同様の名前の map ファイルが見つかった場合は、map ファイルの内容も参照して、さらに詳細な情報を提供します。

[Build Analyzer]ビューは、プログラムの最適化または簡素化に役立ちます。ビューには、[Memory Regions]と[Memory Details]の 2 つのタブがあります。

- [Memory Regions]タブには、elf ファイルに、対応する map ファイルが含まれている場合に、データが表示されます。map ファイルを使用できる場合、このタブはメモリ領域に関する簡単なサマリと捉えることができます。領域の名前、開始アドレス、サイズについての情報が表示されます。サイズ情報は、さらに各領域の総容量、空き容量、使用済み容量、使用割合に分かれます。
- [Memory Details]タブには、elf ファイルに基づく詳細なプログラム情報が表示されます。さまざまなセクション名が、アドレスやサイズの情報とともに表示されます。各セクションは、展開と折りたたみが可能です。セクションを展開すると、そのセクションの関数 / データが一覧表示されます。これらの関数 / データは、アドレスとサイズの情報とともに表示されます。

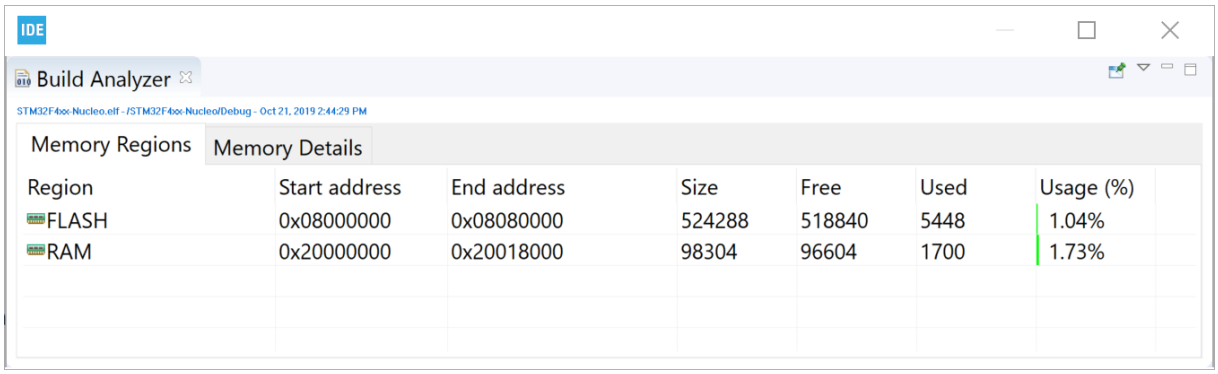
8.2 Build Analyzer の使用方法

[Build Analyzer]ビューは、デフォルト設定の場合 C/C++ パースペクティブで開きます。ビューが閉じている場合は、メニューから開くことができます。メニュー·コマンド WindowShow ViewBuild Analyzer を選択するか、クイック·アクセス·フィールドで Build Analyzer を検索して、表示される結果からビューを選択します。

[Build Analyzer]ビューが開いたら、[Project Explorer]ビューで elf ファイルを選択します。このファイルの情報に基づいて[Build Analyzer]ビューが更新されます。elf ファイルを選択し、同じフォルダ内に同様の名前の map ファイルが見つかった場合、map ファイルからの追加情報もビューの表示に使用されます。

[Build Analyzer]ビューは、[Project Explorer]ビューのプロジェクト·ノードを選択した場合にも更新されます。その場合、Build Analyzer は、そのプロジェクトに対して現在アクティブ化されているビルド設定に対応する elf ファイルを使用します。

図 221. Build Analyzer



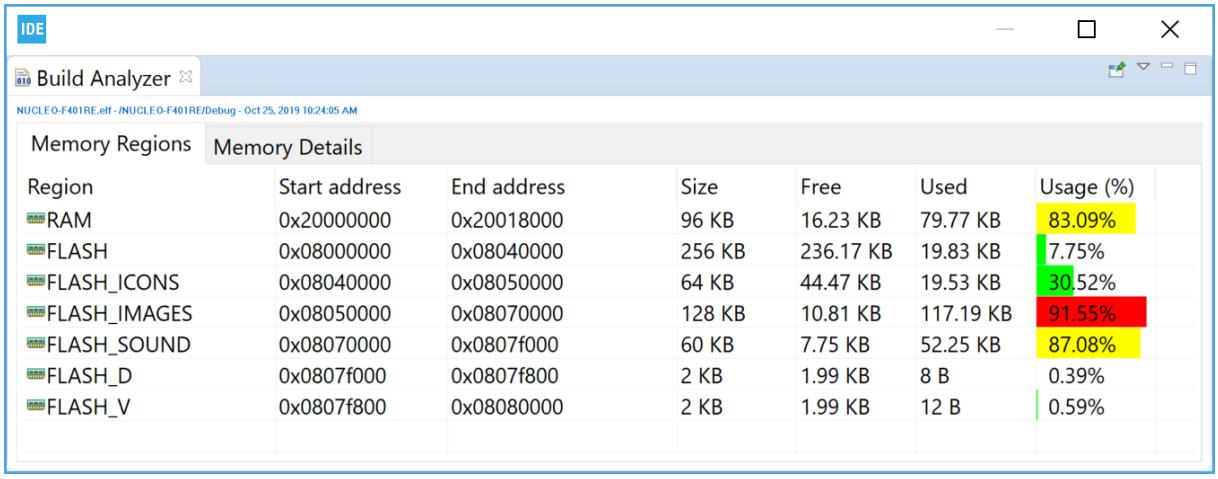
8.2.1 [Memory Regions]タブ

[Build Analyzer]ビューの[Memory Regions]タブには、対応する map ファイルに基づいた情報が表示されます。情報が一切表示されない場合は、対応する map ファイルが見つからなかったことを意味します。map ファイルが見つかった場合は、メモリの領域名、開始アドレス、終了アドレス、領域の合計サイズ、空き容量、使用済み容量、使用状況に関する情報が表示されます。

通常、これらの領域はプログラムのビルド時に使用するリンカ·スクリプト·ファイル(.ld)内で定義します。メモリ領域の位置またはサイズを変更する必要がある場合は、リンカ·スクリプト·ファイルを変更します。

注 [Memory Regions]タブには、elf ファイルに対応する map ファイルが存在しない場合、何も表示されません。

図 222. [Memory Regions]タブ



列に関する情報を 表 23 に示します。

表 23. [Memory Regions]タブの情報

列名	説明
Region	メモリ領域の名前(対応する `map` ファイルが見つかった場合)。
Start address	リンカ·スクリプトで定義された、領域の開始アドレス。
End address	領域の終了アドレス。
Size	メモリ領域の合計サイズ。
Free	メモリ領域の空き容量。
Used	メモリ領域の使用済み容量。
Usage %	メモリ領域の合計サイズに対する使用済み容量の割合(%)。棒グラフの色分けに関する情報は 表 24 を参照してください。

Usage (%) 列には、使用割合の値に対応した棒グラフが表示されます。使用済みメモリの割合に応じて色分けされます。

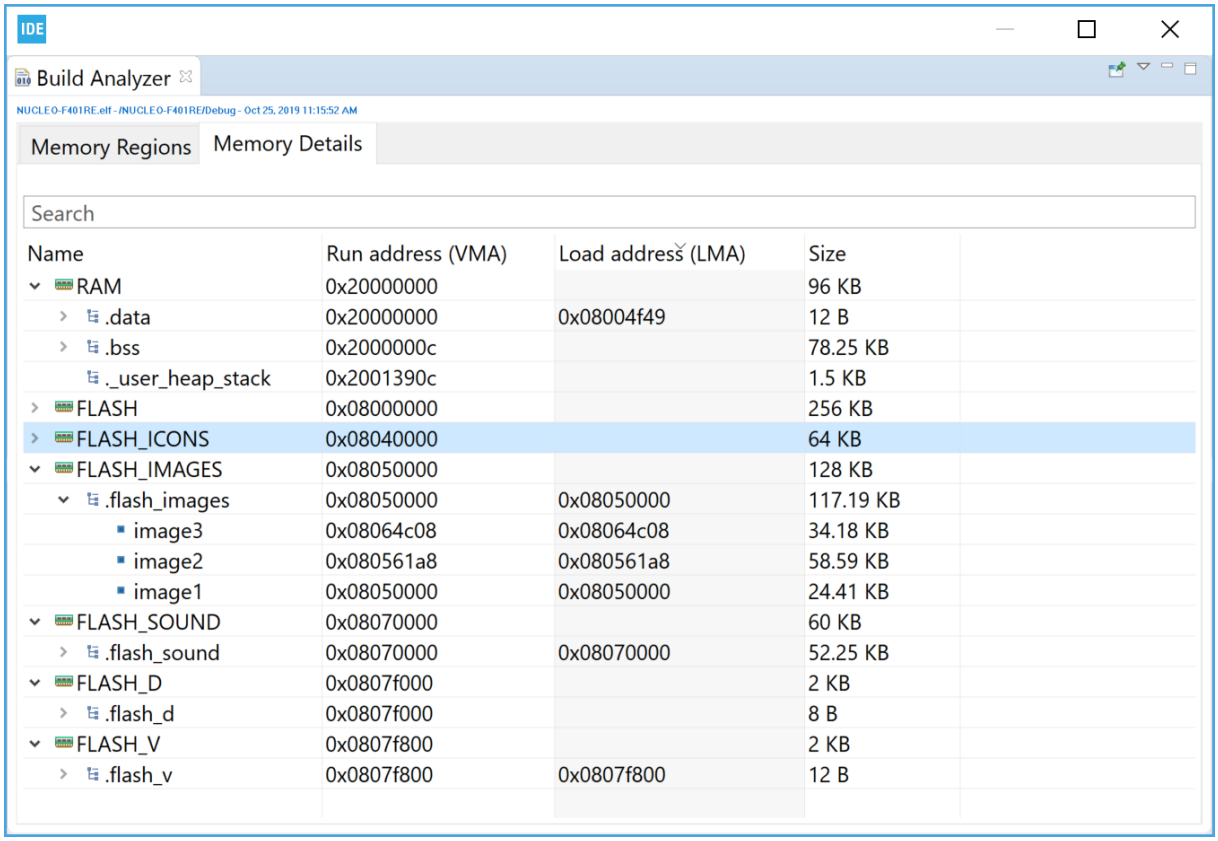
表 24. [Memory Regions]ビュー - 使用割合による色分け

使用量の色	説明
緑	使用済みのメモリが 75% 未満
黄	使用済みのメモリが 75% ~ 90%
赤	使用済みのメモリが 90% 超

8.2.2 [Memory Details]タブ

[Build Analyzer]ビューの[Memory Details]タブには、`elf` ファイルの情報が表示されます。[Memory Details]タブの各セクションは展開して個々の関数やデータを確認できます。タブには、メモリ領域の名前、実行時アドレス、ロード·アドレス、サイズの情報が表示されます。

図 223. [Memory Details]タブ



列に関する情報を 表 25 に示します。

表 25. [Memory Details]タブの情報

列名	説明
Name	メモリ領域、セクション、関数、データの名前。緑のアイコンは関数、青のアイコンはデータ変数を示します。
Run Address (VMA)	仮想メモリ·アドレスにはプログラム実行時に使用されるアドレスが含まれます。
Load Address (LMA)	ロード·メモリ·アドレスは、ロード時に使用するアドレス、例えばグローバル変数の初期値などのアドレスです。
Size	使用サイズ(メモリ領域の合計サイズ)。

注 メモリ領域の名前は、対応する `map` ファイルが見つかった場合にのみ表示されます。

8.2.2.1 *サイズ情報*

[Memory Details]タブのサイズ情報は、`elf` ファイル内のシンボル·サイズから計算されます。対応する `map` ファイルを調べると、異なるサイズ値が含まれている場合があります。通常、C ファイルでは正しい値になりますが、アセンブラ·ファイルについては、その中にどのようにサイズ情報が記述されているかによって異なります。関数が使用する定数は、セクション定義内で定義する必要があります。セクションの末尾で、リンカがサイズ·ディレクティブを使用して、関数のサイズを計算します。

例：`startup.s` ファイル内の Reset_Handler

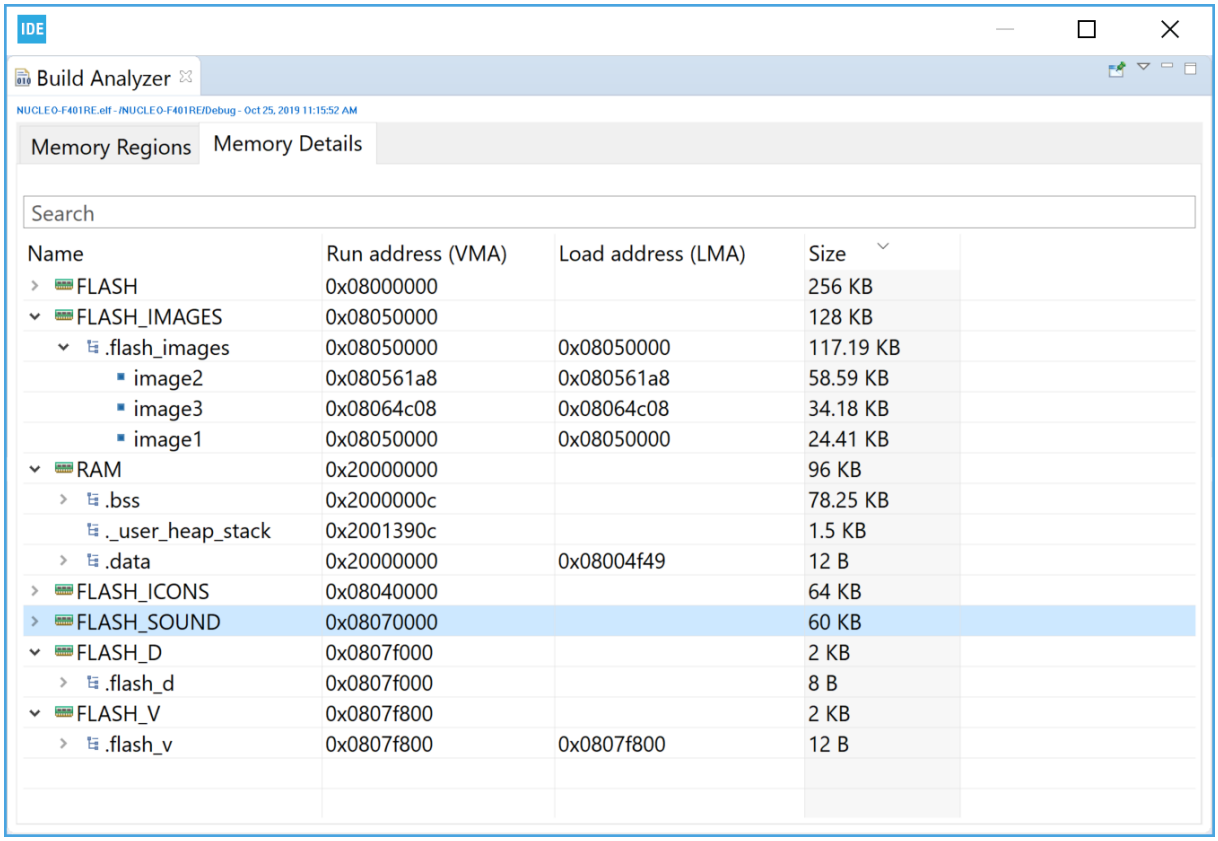
この例は、`Reset_Handler` のサイズ情報の定数、`_sidata`、`_sdata`、`_edata`、`_sbss`、`_ebss` を `elf` ファイルに含めるために、アセンブラのスタートアップ·ファイルで `Reset_Handler` をどのように記述するのかを示したものです。これらの定数が `Reset_Handler` のセクション定義以外の場所で定義されている場合、それらのサイズは `Reset_Handler` のサイズ計算に含まれません。`Reset_Handler` のサイズに含めるには、下記のコード例のように、これらの定義を `Reset_Handler` のセクション内に置く必要があります。

```
.section .text.Reset_Handler
.weak Reset_Handler
.type Reset_Handler, %function

Reset_Handler:
ldr sp, =_estack /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
movs r1, #0
b LoopCopyDataInit

CopyDataInit:
ldr r3, =_sidata

/* initialization code data, bss, ... */
...

/* Call the application's entry point */
bl main
bx lr

/* start address for the initialization values defined in linker script */
.word _sidata
.word _sdata
.word _edata
.word _sbss
.word _ebss

.size Reset_Handler, .-Reset_Handler
```

8.2.2.2 ソート

[Memory Details]タブの列のソート順は、列（カラム）名をクリックすることで変更できます。

図 224. サイズに基づいてソートした[Memory Details]タブ

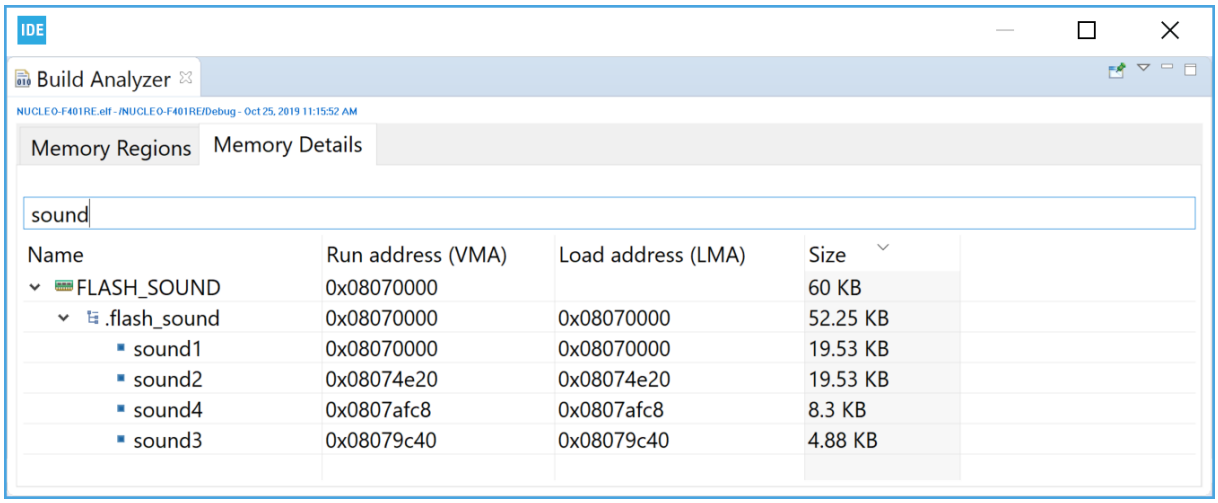


8.2.2.3 *検索とフィルタ*

[Memory Details]タブ内の情報は、検索フィールドに文字列を入力することで絞り込むことができます。

図 225 に、sound という文字列を含む名前を検索した例を示します。

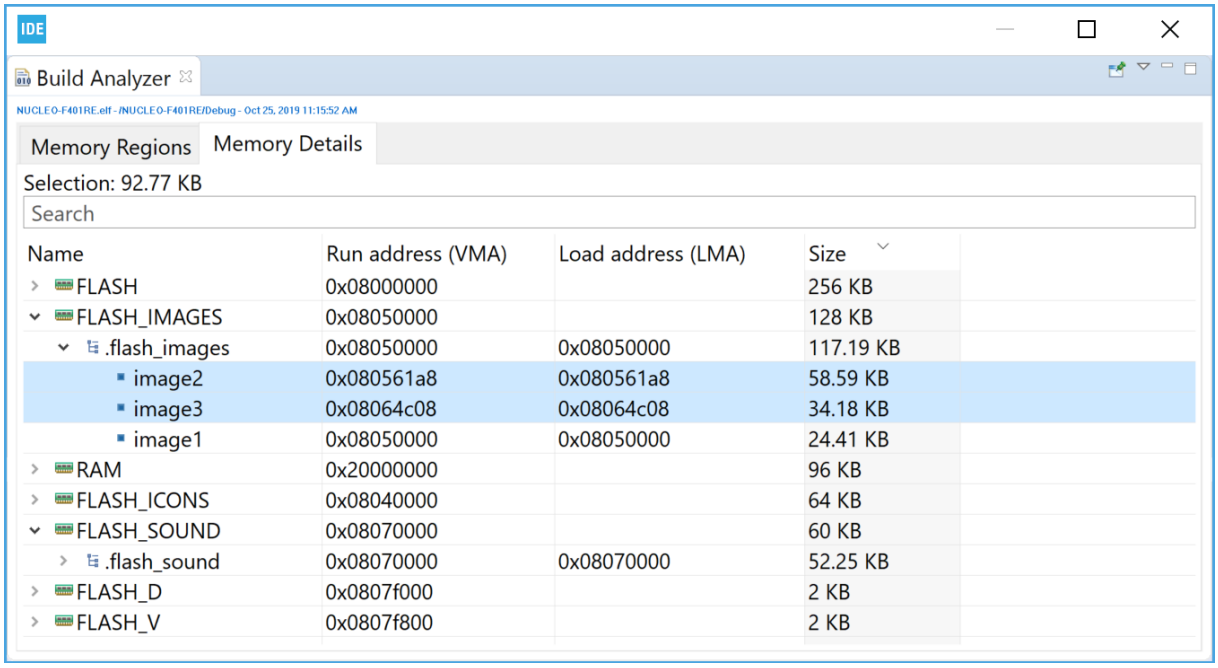
図 225. [Memory Details]タブの検索とフィルタ



8.2.2.4 *サイズ合計の計算*

[Memory Details]タブの複数行にわたるサイズの合計を、ビュー内でこれらの行を選択することで計算できます。選択行の合計は、ビューの Name 列の上に表示されます。

図 226. サイズ合計



8.2.2.5 *バイト·フォーマットによるサイズ情報の表示*

[Build Analyzer]ビューでは、サイズ情報をさまざまなフォーマットで表示できます。フォーマットは Show Byte、Show Hex、Show Human のいずれかを選択します。フォーマットの切り換えには[Build Analyzer]ツールバーのアイコンを使用します。計算などの後処理のために Excel® ドキュメントにデータをコピーして貼り付ける場合は、Show Byte または Show Hex を使用することを推奨します。

図 227. サイズのバイト表示

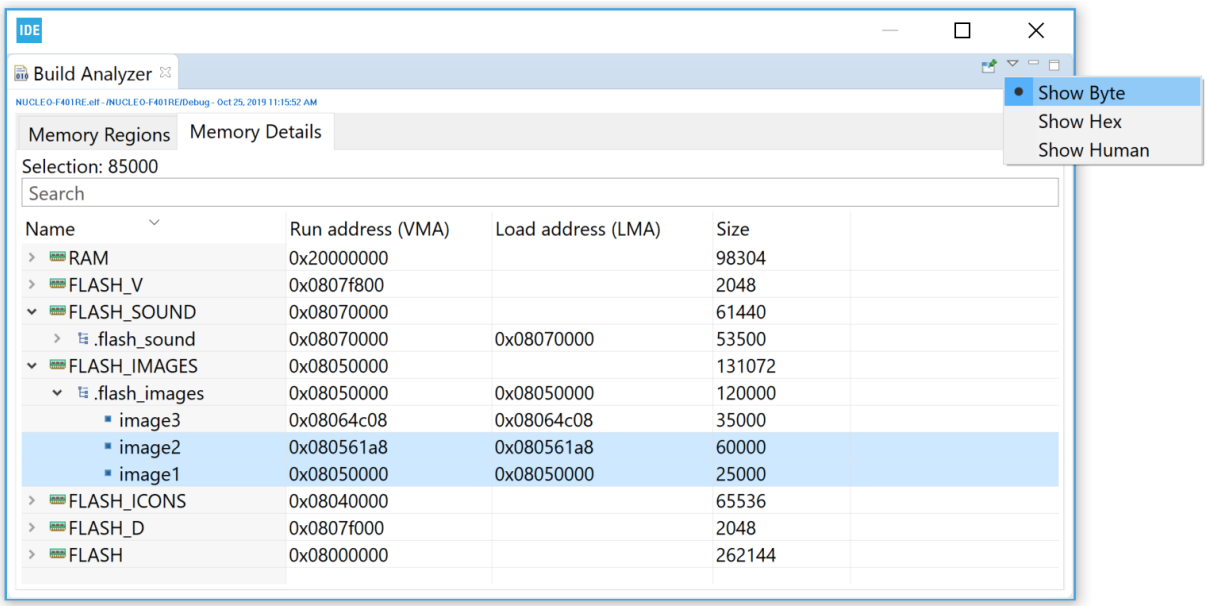
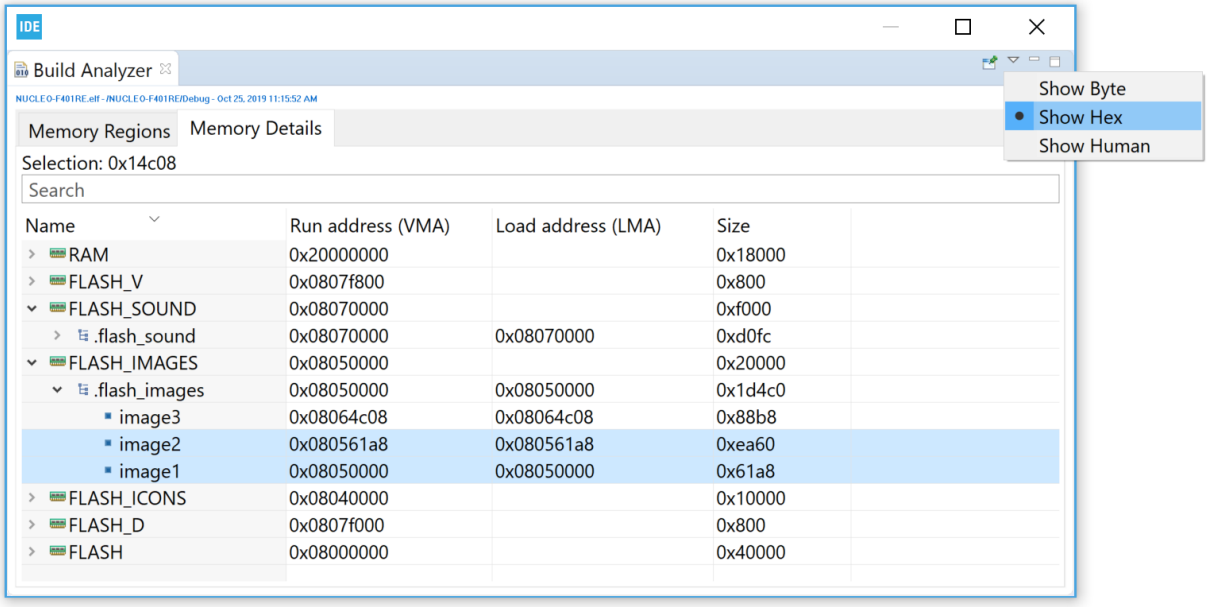


図 228. サイズの 16 進表示



8.2.2.6 *コピーと貼り付け*

[Memory Details]タブのデータは、CSV フォーマットで他のアプリケーションにコピーできます。それには、コピーする行を選択し、Ctrl+C を押します。コピーしたデータは Ctrl+V コマンドによって他のアプリケーションに貼り付けることができます。

図 229. コピーと貼り付け

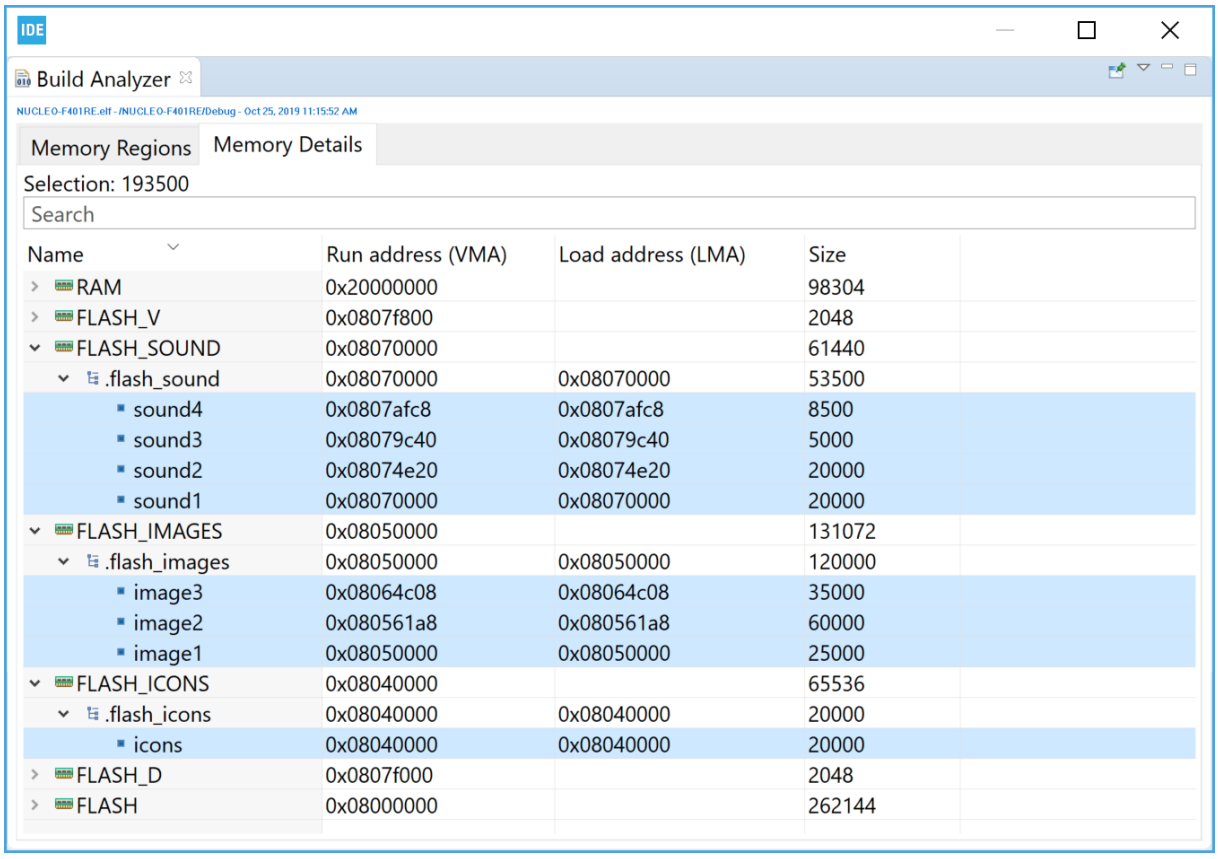


図 229 に示すように選択した行を Ctrl+C によってコピーし、Ctrl+V によって貼り付けると、次のような結果が得られます。

```
"sound4";"0x0807afc8";"0x0807afc8";"8500"
"sound3";"0x08079c40";"0x08079c40";"5000"
"sound2";"0x08074e20";"0x08074e20";"20000"
"sound1";"0x08070000";"0x08070000";"20000"
"image3";"0x08064c08";"0x08064c08";"35000"
"image2";"0x080561a8";"0x080561a8";"60000"
"image1";"0x08050000";"0x08050000";"25000"
"icons";"0x08040000";"0x08040000";"20000"
```

9 Static Stack Analyzer

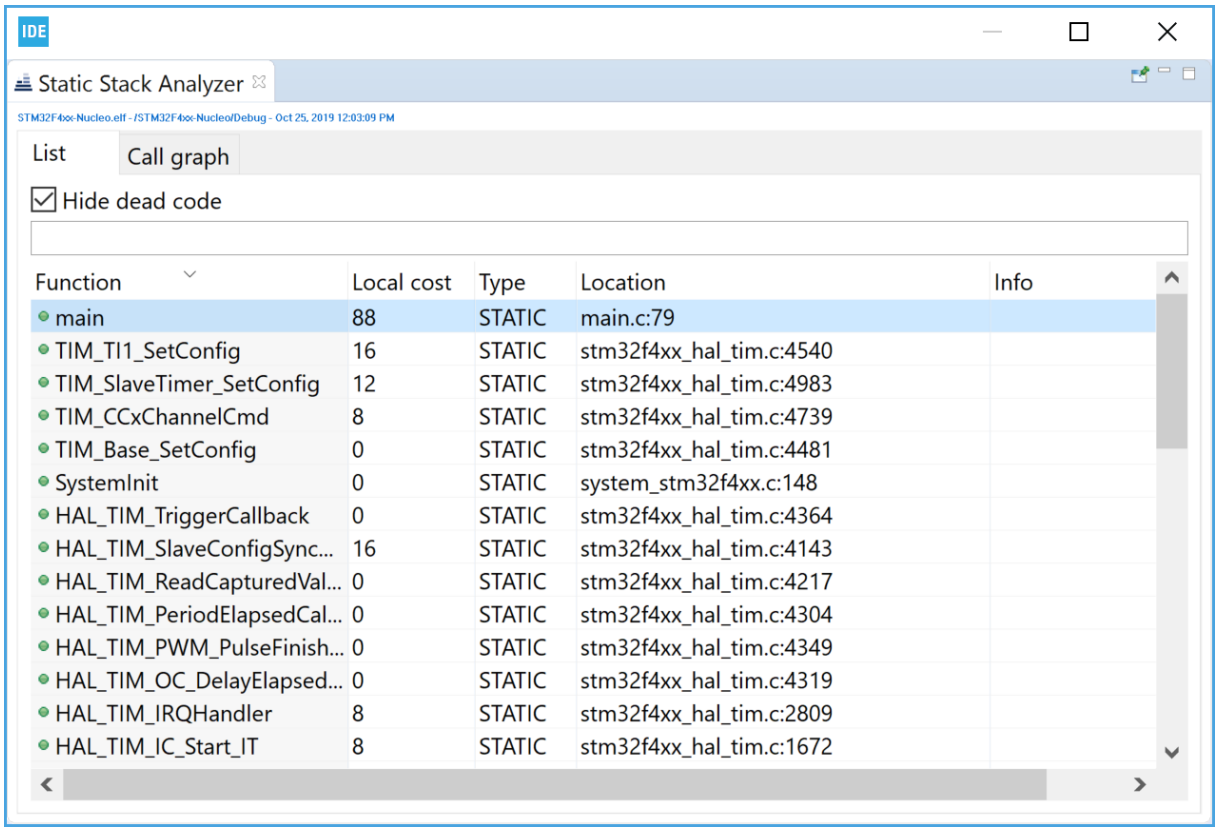
9.1 Static Stack Analyzer の概要

STM32CubeIDE の Static Stack Analyzer は、ビルドされたプログラムに基づいてスタックの使用状況を計算します。gcc によって生成される `.su` ファイルと `elf` ファイルを詳細に分析し、得られた情報をビューに表示します。

ビューには、[List]と[Call Graph]の 2 つのタブがあります。

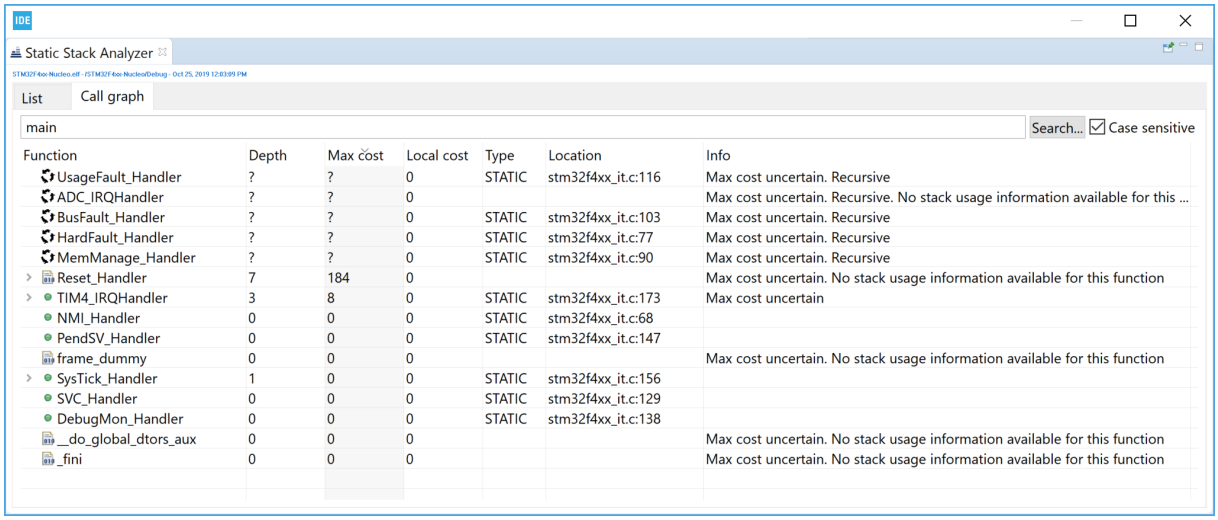
[List]タブには、プログラムに含まれる各関数のスタック使用状況が表示されます。1 つの関数あたり 1 行を使用し、各行は Function、Local cost、Type、Location、Info の列から構成されます。

図 230. Static Stack Analyzer - [List]タブ



[Call Graph]タブには、プログラムに含まれる関数の展開可能なリストが表示されます。他の関数を呼び出す関数の行を展開すると、呼び出しの階層を確認できます。

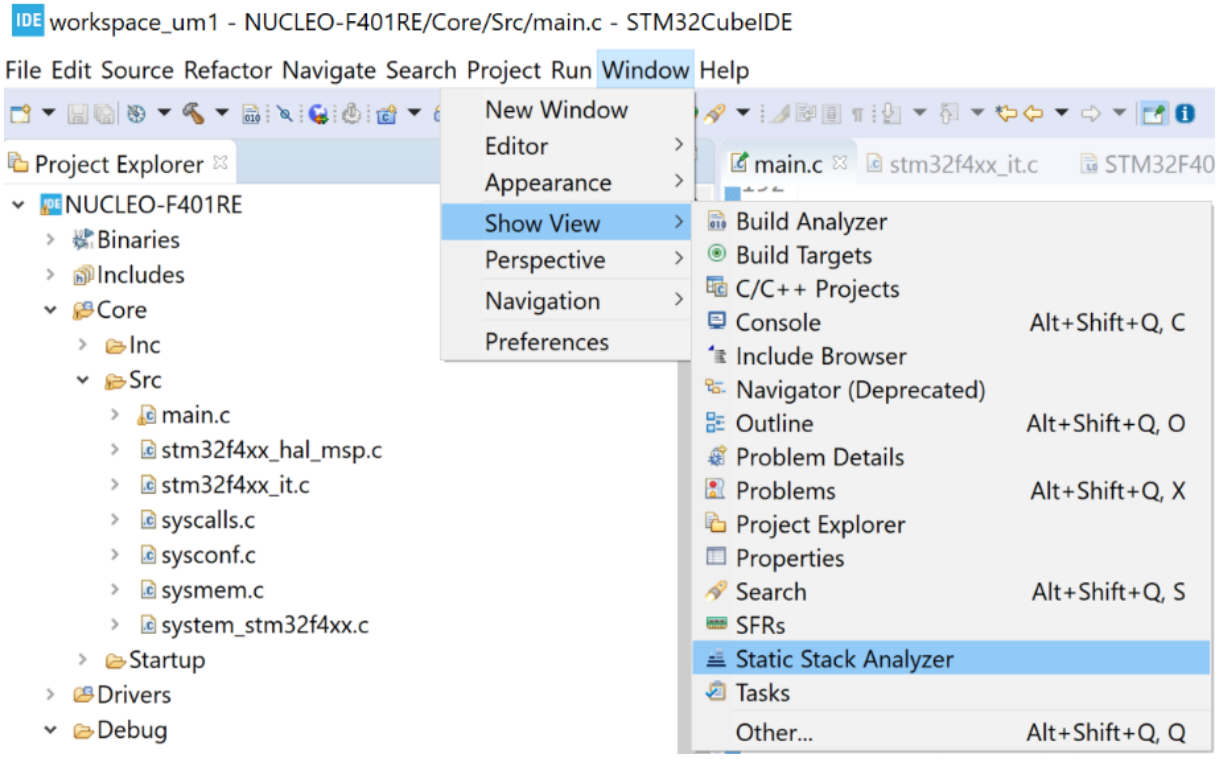
図 231. Static Stack Analyzer - [Call Graph]タブ



9.2 Static Stack Analyzer の使用方法

[Static Stack Analyzer]ビューは、デフォルト設定の場合 C/C++ パースペクティブで開きます。ビューが閉じている場合は、メニューから開くことができます。メニュー·コマンド WindowShow ViewStatic Stack Analyzer を選択します。[Static Stack Analyzer]ビューを開くもう一つの方法は、クイック·アクセス検索バーに Static Stack Analyzer と入力して、表示された検索結果からビューを選択します。

図 232. [Static Stack Analyzer]ビューを開く



[Static Stack Analyzer]ビューのデータは、Project Explorer から、ビルド済みプロジェクトを選択すると表示されます。プロジェクトは Generate per function stack usage information オプションを有効化してビルドしたものでなければなりません。それ以外のプロジェクトではスタック情報が一切表示されません。

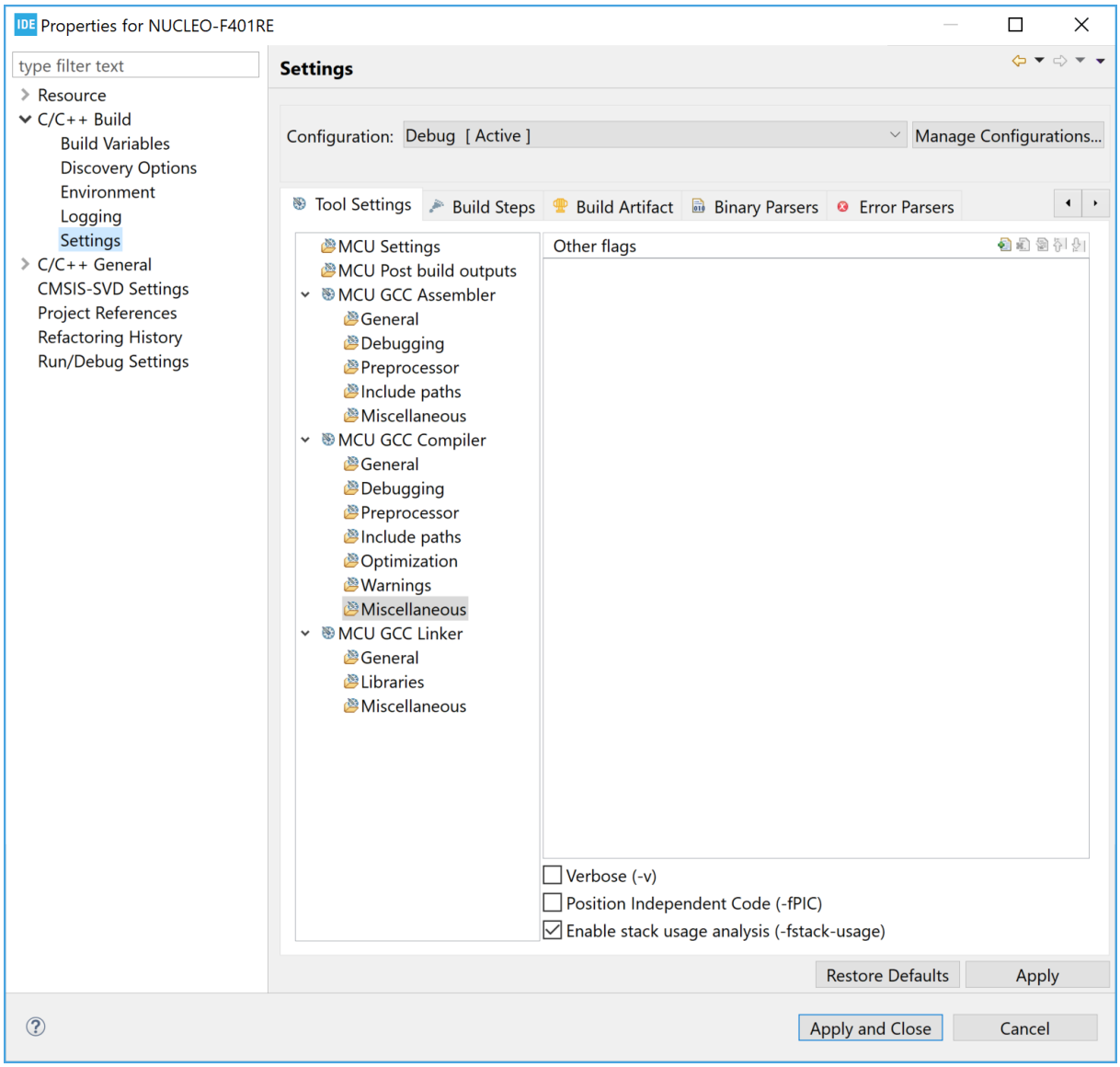
次のセクションでは、スタックの使用状況情報を生成するようにコンパイラを設定する方法を説明します。

9.2.1 スタック使用状況情報の有効化

ビューの上部に `No stack usage information found, please enable in the compiler settings` (スタック使用状況の情報が見つかりません。コンパイラ設定で有効化してください)というメッセージが表示される場合、コンパイラがスタック情報を生成するように、ビルド設定を変更する必要があります。

1. [Project Explorer]ビューでプロジェクトを右クリックするなどの方法でプロジェクト·プロパティを開きます。
2. [Properties]を選択して、ダイアログの C/C++ BuildSettings を選択します。
3. [Tool Settings]タブを選択します。
4. MCU GCC CompilerMiscellaneous を選択します。
5. 図 233 に示したように、Enable stack usage information (-fstack-usage) を選択します。
6. 設定を保存し、プログラムを再ビルドします。

図 233. 関数ごとのスタック使用状況情報の生成の有効化



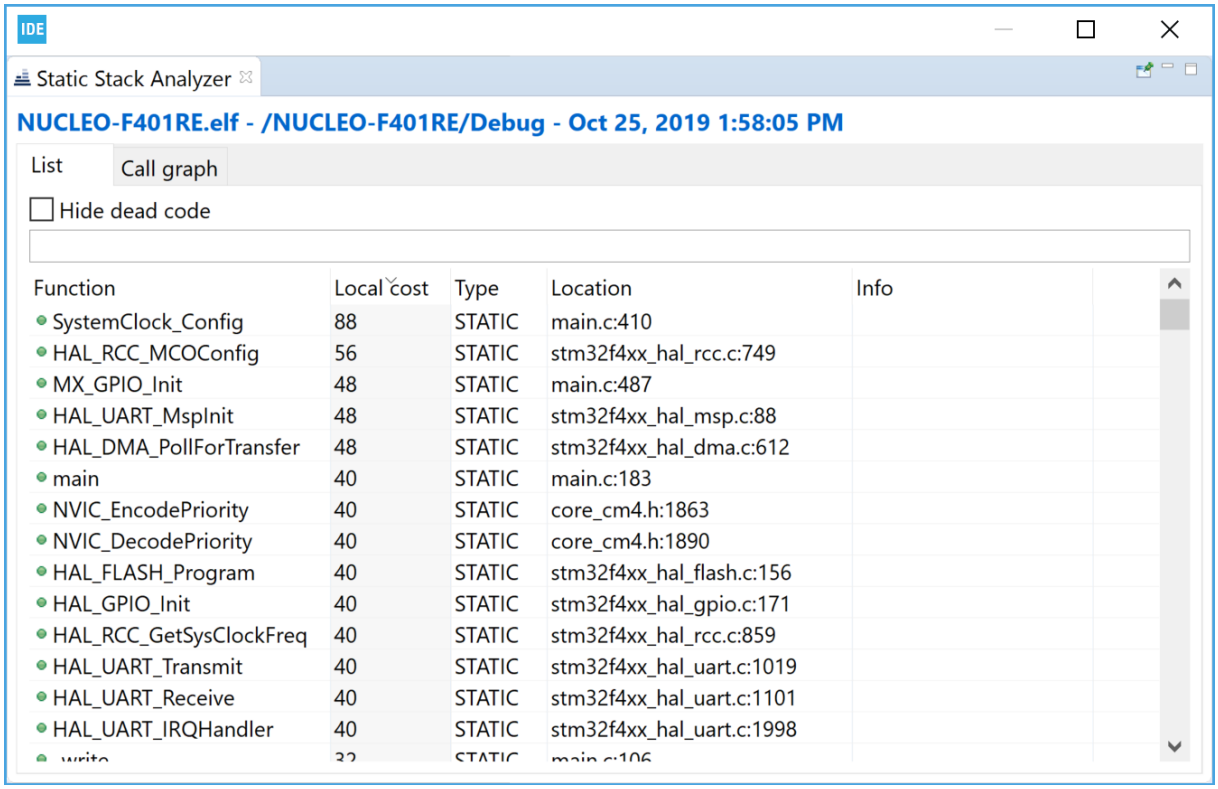
9.2.2 [List]タブ

[List]タブには、選択したプログラムに含まれる、すべての関数が一覧表示されます。[Hide dead code]のオプションと、表示する関数のフィルタ機能を備えています。

デッド·コードとなる関数を非表示にする設定の有効 / 無効は Hide dead code で選択します。

フィルタ·フィールドを使用すると、一覧表示が、フィールドに入力した文字に一致する関数に絞り込まれます。

図 234. Static Stack Analyzer - [List]タブ



[List]タブの列に関する情報を、表 26 に示します。

表 26. Static Stack Analyzer - [List]タブの詳細

列名	説明
Function	関数名。
Local cost	この列の数値は、関数が使用しているスタックのバイト数を表します。
Type	関数が、静的(STATIC)または動的(DYNAMIC)のいずれのスタック割当てを使用しているかの表示です。動的割当てを使用している場合、実際のスタック·サイズは実行時の状況によって決まり、スタックのサイズが動的であることから Local cost の値は不確実です。
Location	関数が宣言されている場所を示します。行をダブルクリックすることで、関数定義を含むファイルをエディタで開くことができます。
Info	計算に関する追加情報。

[List]タブのソート順は、列名をクリックすることで変更できます。

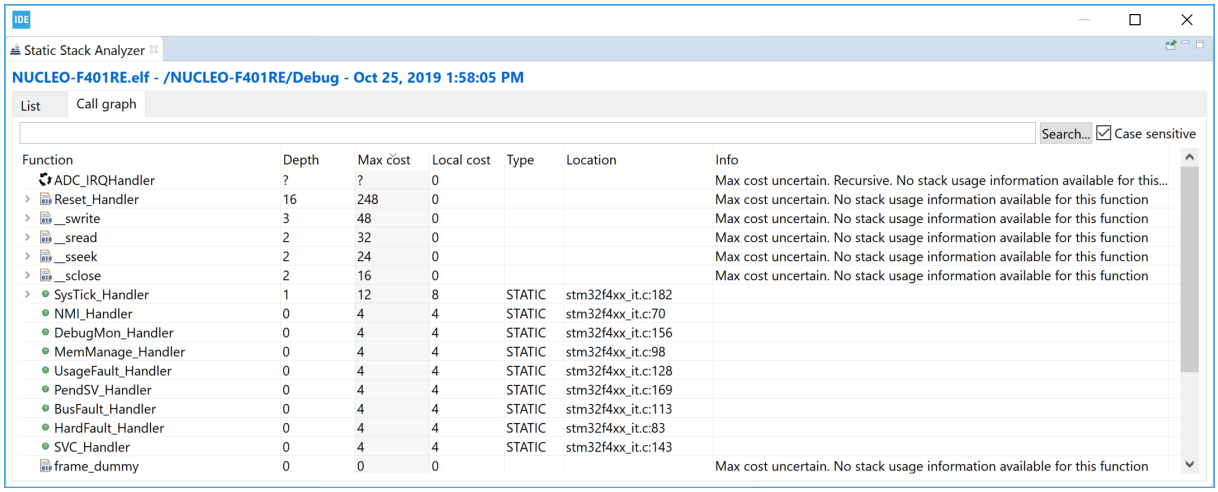
注 [List]タブのファイルの場所と行番号を表示している行をダブルクリックすると、関数が[Editor]ビューで開きます。

9.2.3 [Call Graph]タブ

[Call Graph]タブには、プログラムの詳細情報がツリー表示されます。プログラムに含まれるものの、他のいずれの関数からも呼び出されない関数が最上位に表示されます。ツリーを展開することで、呼び出される関数を確認できます。タブに表示できるのは、`elf` ファイルから把握できる関数のみです。

Search... ボタンを使用すると、検索フィールドの文字に一致する関数のみの表示に絞り込まれます。検索で大文字と小文字を区別するかどうかは、Case sensitive チェックボックスで設定します。

図 235. Static Stack Analyzer - [Call Graph]タブ



[Call Graph]タブの列に関する情報を、表 27 に示します。

表 27. Static Stack Analyzer - [Call Graph]タブの詳細

列名	説明
Function	関数名。
Depth	この関数が使用する呼び出しスタックの深さを表します。 • 0: この関数は、他の関数を一切呼び出していません。 • 1 以上の値 : 関数は他の関数を呼び出しています。 • ?: 関数が再帰呼び出しを行っているか、深さを計算できません。
Max cost	関数が使用するスタックのバイト数を、呼び出される関数側で必要となるものも含めて表示します。
Local cost	関数が使用するスタックのバイト数を表します。この列では、この関数が呼び出す関数が必要とするスタックは考慮されていません。
Type	関数が、静的(STATIC)または動的(DYNAMIC)のいずれのスタック割当てを使用しているかの表示です。 • STATIC: 関数は固定スタックを使用します。 • DYNAMIC: 関数は実行時の状況に応じてスタックを使用します。 • 空欄 : この関数のスタック使用状況に関する情報が得られません。
Location	関数が宣言されている場所を示します。行をダブルクリックすることで、関数定義を含むファイルをエディタで開くことができます。
Info	スタック使用状況の計算に関する固有情報が表示されます。例えば、次のようなメッセージの組合せが考えられます。 • `Max cost uncertain`: 理由として考えられるのは、関数が、スタック情報が不明のサブ関数を呼び出している場合や、関数が再帰呼び出しを行っている場合などです。 • `Recursive`: 関数が再帰呼び出しを行っています。 • `No stack usage information available for this function`: この関数のスタック使用状況の情報を入手できません。 • `Local cost uncertain due to dynamic size, verify at run-time`: 関数が、例えばパラメータなどに従って、スタックを動的に割り当てています。

[Call Graph]タブのソート順は、列名をクリックすることで変更できます。

このタブで、ファイルの場所と行番号が表示されている行をダブルクリックすると、その関数が[Editor]ビューで開きます。

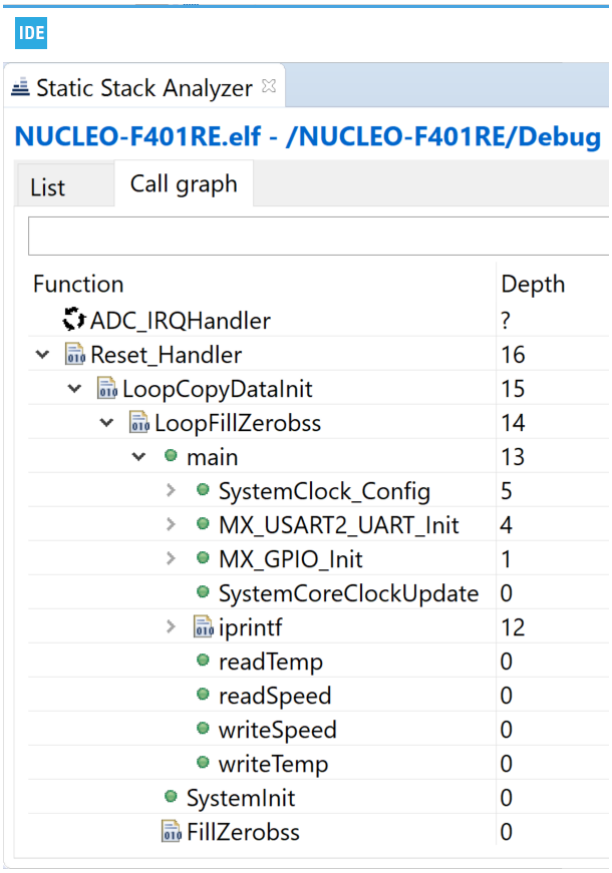
注 `main` 関数は、通常 `Reset_Handler` によって呼び出され、その場合は `Reset_Handler` ノードを展開することで表示されます。

不使用の関数がタブに表示される場合は、プログラムから不使用のコードを削除するリンカのオプション dead code removal が有効化されているかどうかを確認してください。詳細は、セクション 2.5.2 使用していないセクションの破棄を参照してください。

[Function]列の関数名の左にある小さなアイコンは、次のような情報を示しています。

- 緑のドット : 関数が静的スタック割当て(固定スタック)を使用しています。
- 青の正方形 : 関数が動的スタック割当て(実行時の状況に依存)を使用しています。
- 010 アイコン : スタック情報が不明の場合に使用されます。ライブラリ関数またはアセンブラ関数の場合などです。
- 円を描く 3 本の矢 : 関数が再帰呼び出しを行っている場合に、[Call Graph]タブで使用されます。

図 236. Static Stack Analyzer の関数の記号



9.2.4 フィルタと検索フィールドの使用方法

[List]および[Call Graph]タブには、フィルタ / 検索フィールドがあります。文字を入力すると、一致する特定の関数を検索することができます。

図 237 に、フィルタ·フィールドを使用して、名前に read という文字列を含む関数を検索したときの[List]タブを示します。

図 237. **Static Stack Analyzer - [List]タブの検索**

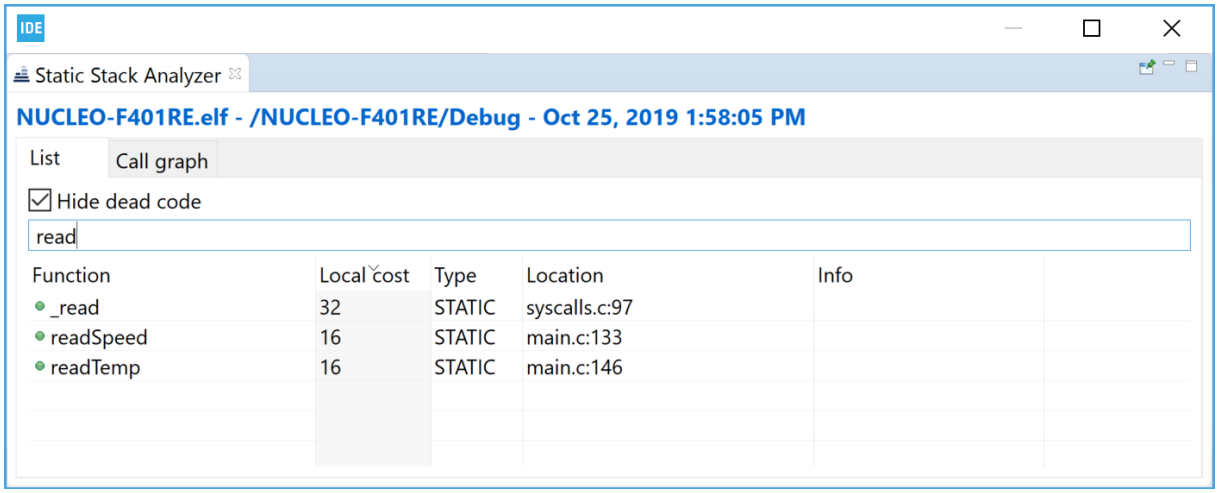
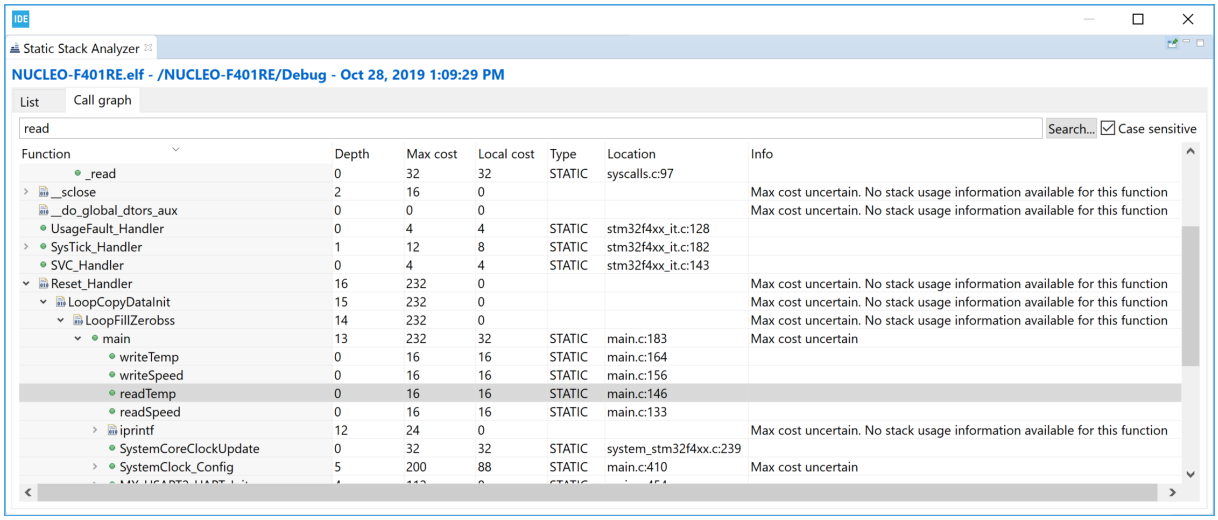


図 238 に、Search... フィールドを使用して、名前に read という文字列を含む関数のみに表示を絞り込んだ[Call Graph]タブを示します。

図 238. **Static Stack Analyzer - [Call Graph]の検索**



9.2.5 コピーと貼り付け

[List]タブのデータは、CSV フォーマットで他のアプリケーションにコピーできます。それには、コピーする行を選択し、Ctrl+C を押します。コピーしたデータは Ctrl+V コマンドによって他のアプリケーションに貼り付けることができます。

図 239. コピーと貼り付け

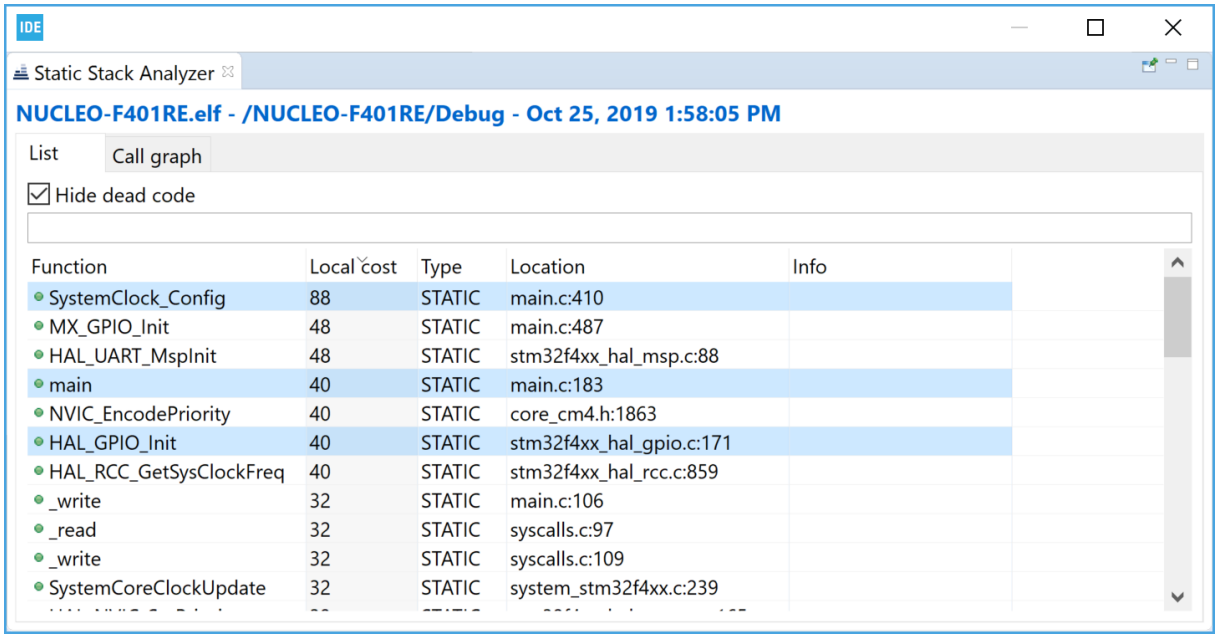


図 239 に示すように選択した行を Ctrl+C によってコピーし、Ctrl+V によって貼り付けると、次のような結果が得られます。

```
"SystemClock_Config";"88";"STATIC";"main.c:410";""
"main";"40";"STATIC";"main.c:183";""
"HAL_GPIO_Init";"40";"STATIC";"stm32f4xx_hal_gpio.c:171";""
```

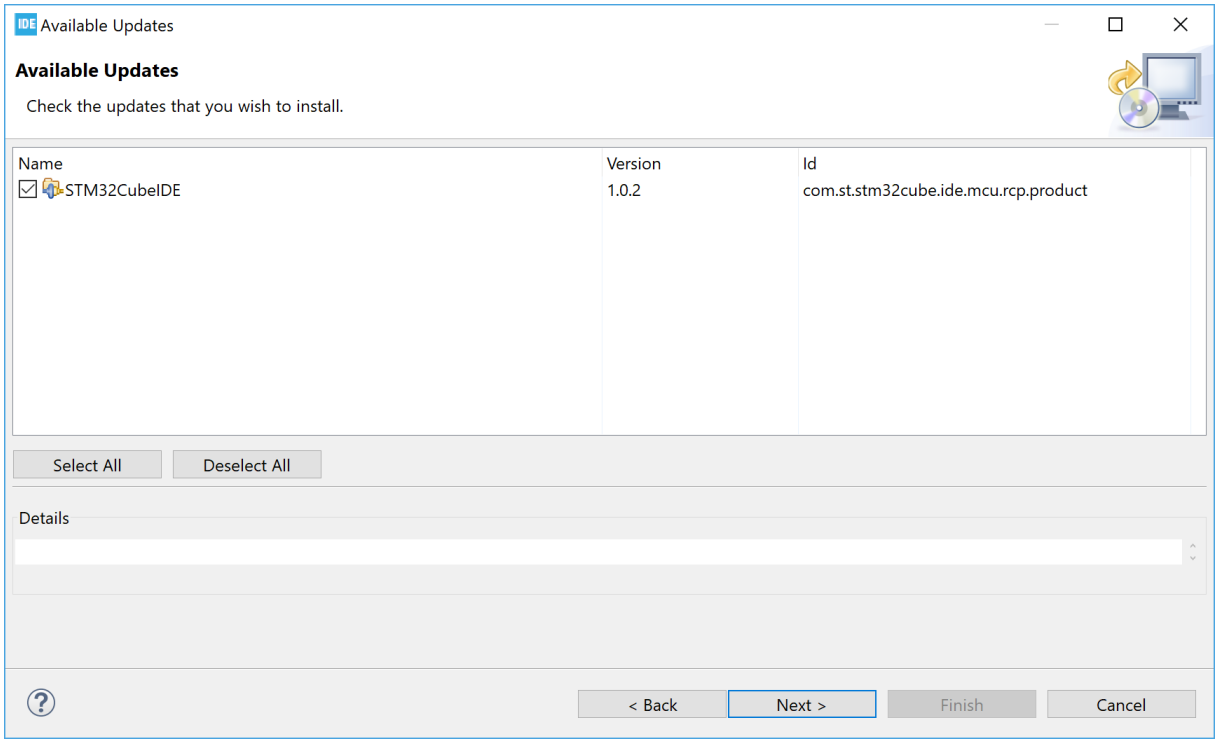
10 更新や Eclipse® 追加プラグインのインストール

10.1 更新の確認

STM32CubeIDE は、利用可能な更新の有無を定期的に確認し、新しい更新が見つかると[Available Updates]ダイアログを開きます。手動で確認することも可能です。使用可能な新しいソフトウェアの有無を確認するには、メニューHelpCheck for Updates を使用します。

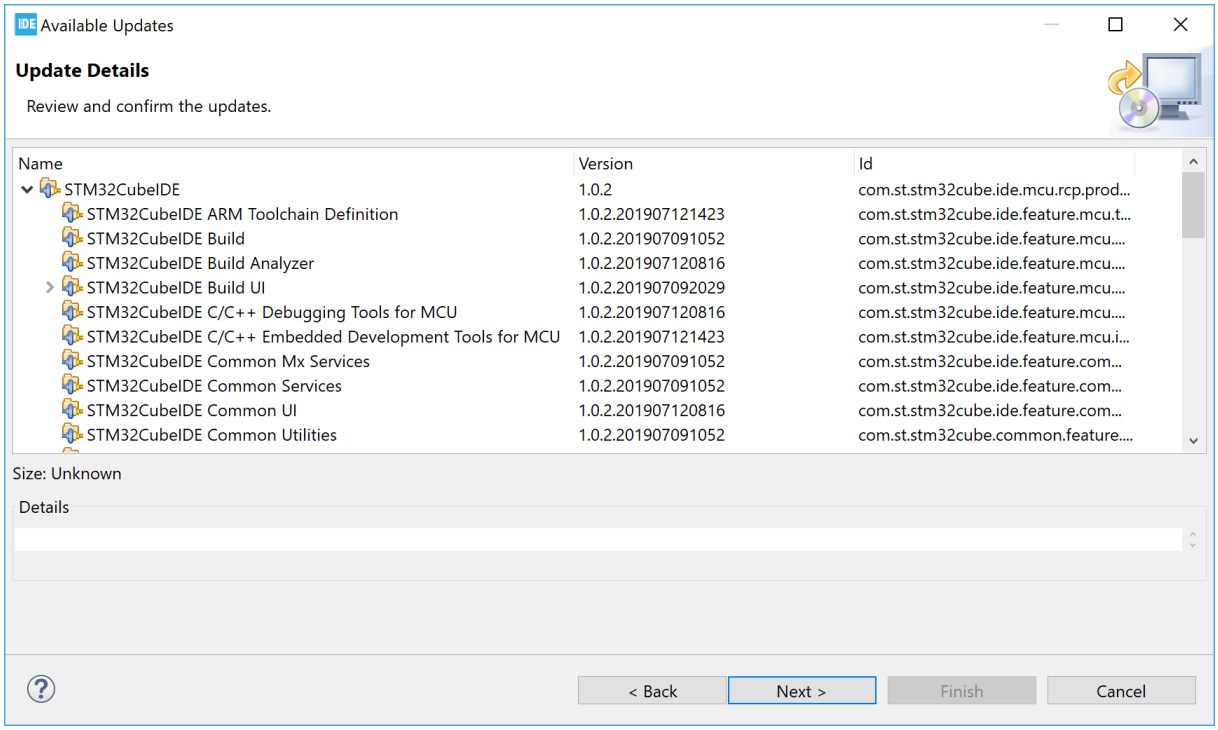
更新が見つかったら、インストールする更新を選択して Next をクリックします。

図 240. STM32CubeIDE の利用可能な更新



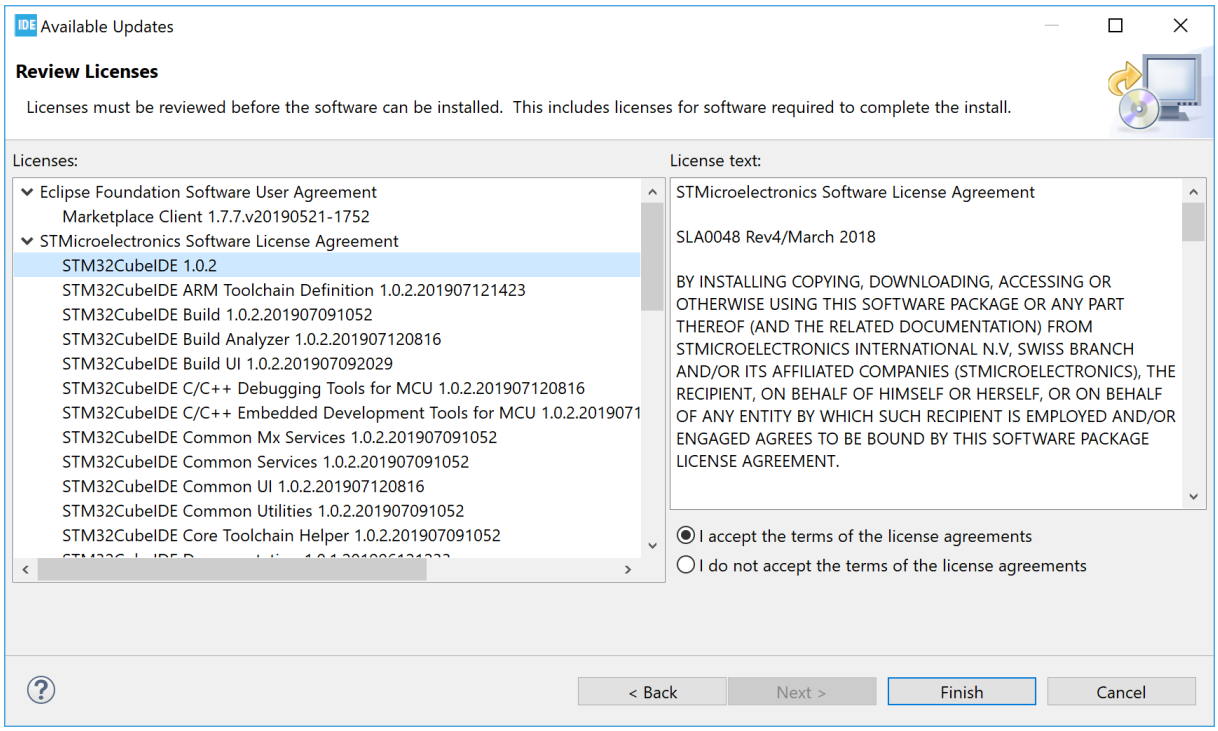
更新の詳細が表示されます。リストの内容を確認し、更新を確定します。[Next]をクリックします。

図 241. STM32CubeIDE の更新の詳細



[Review Licenses]の詳細が表示されます。使用許諾契約を確認し、I accept the terms of the license agreements を選択して Finish をクリックすると、更新がインストールされます。

図 242. STM32CubeIDE の更新のライセンス確認

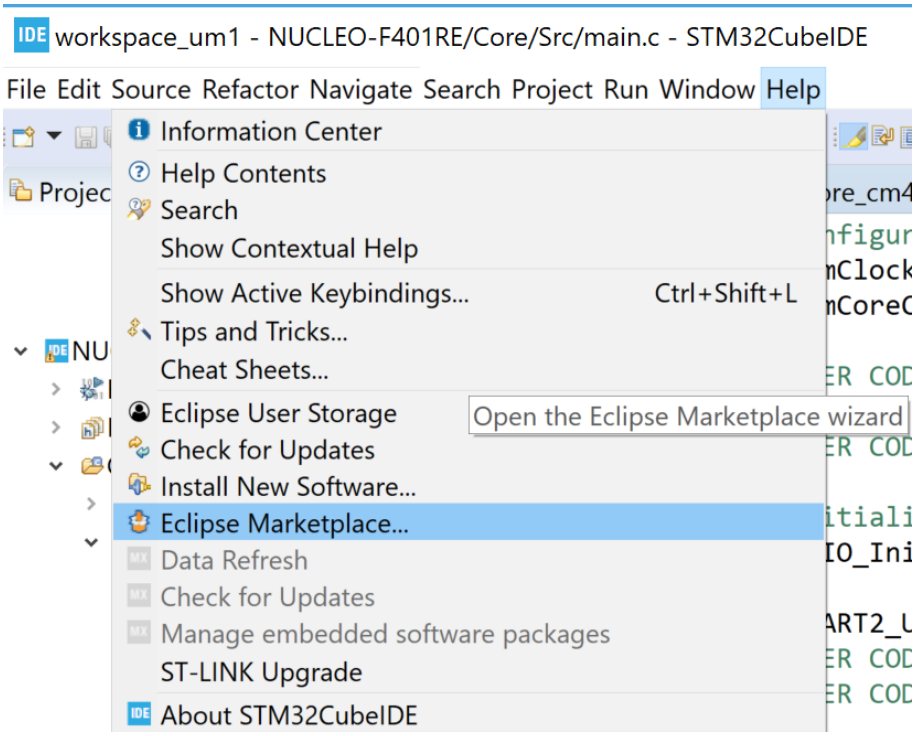


STM32CubeIDE のウィンドウ下部に進捗バーが表示され、インストールがどの程度完了したかがわかります。更新が完了したら、STM32CubeIDE を再起動します。

10.2 Eclipse® マーケットプレイスからのインストール

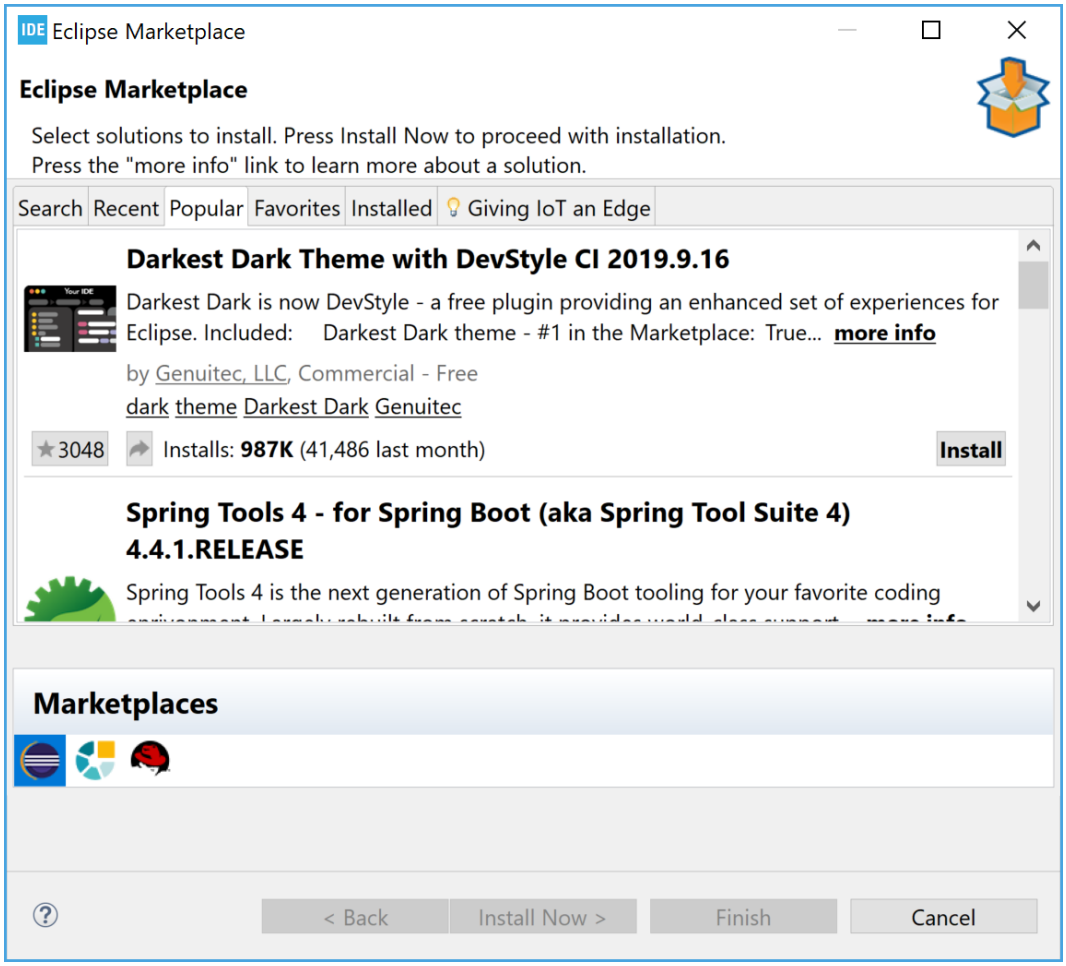
Eclipse マーケットプレイスにアクセスして、3rd パーティによる Eclipse® 追加プラグインを入手し、STM32CubeIDE で使用できます。Eclipse マーケットプレイスからインストールするには、メニュー HelpEclipse Marketplace… を選択します。

図 243. **[Eclipse Marketplace...]メニュー**



[Eclipse Marketplace]ダイアログが開きます。プラグインを検索するか、タブ([Recent]、[Popular]、[Favorites])を使用して、必要なソフトウェアを見つけ、インストールします。

図 244. Eclipse マーケットプレイス



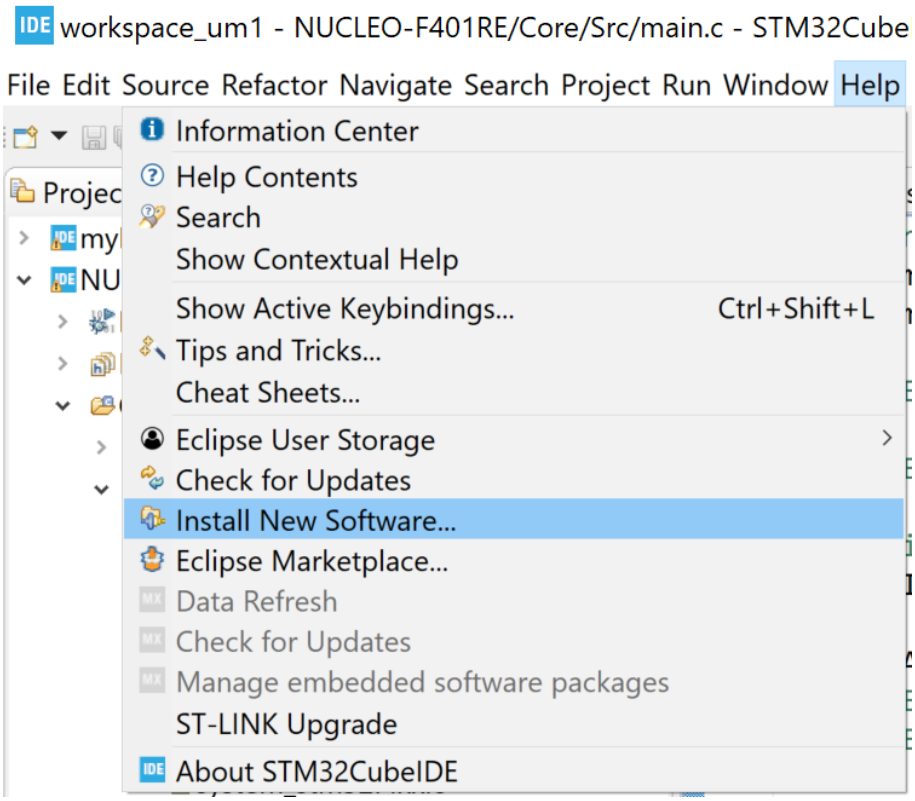
インストールが完了するのを待ち、STM32CubeIDE を再起動します。

10.3 Install new software... によるインストール

新しいソフトウェアをインストールする、もう一つの方法はメニュー HelpInstall New Software... を使用します。

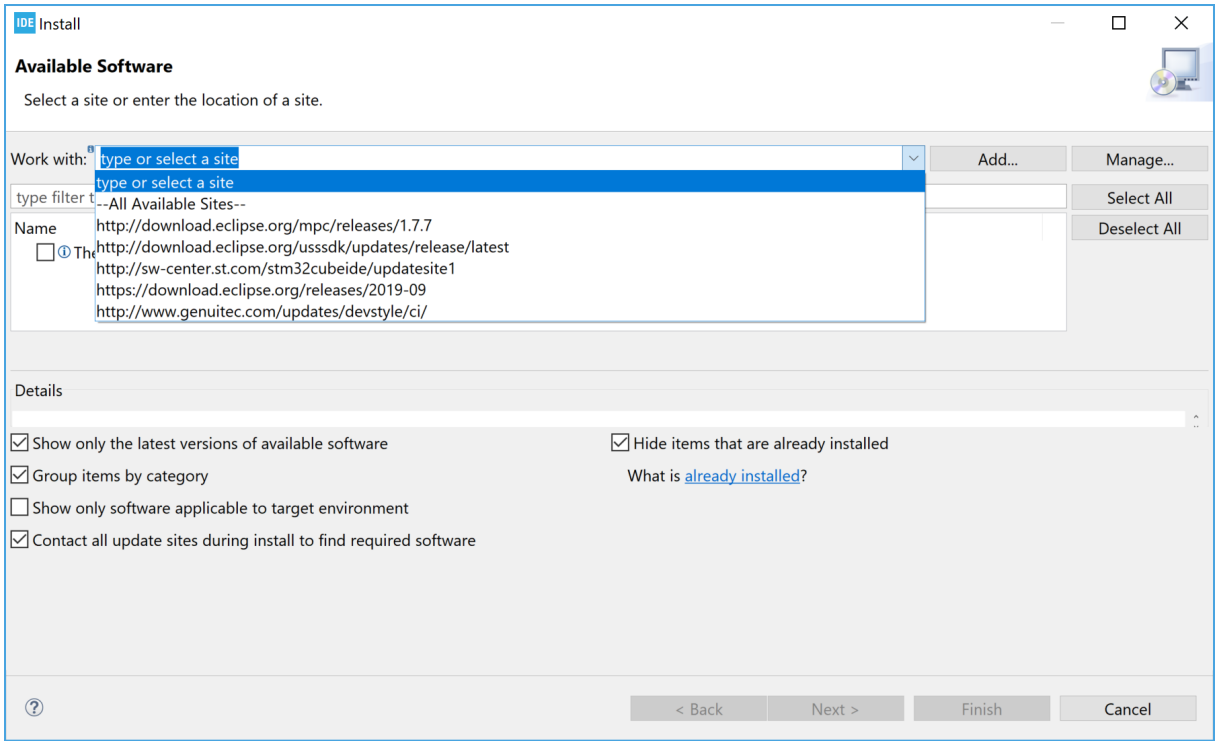
注 新しいツールチェーンをインストールする場合は、セクション 2.11 Toolchain Manager で説明している Toolchain Manager を使用することを推奨します。

図 245. [Install New Software...]メニュー



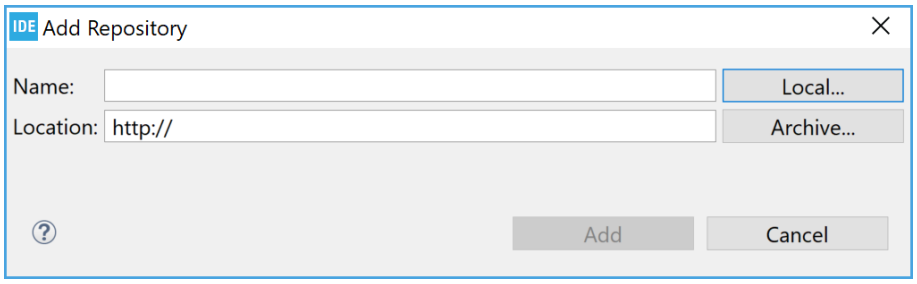
[Install]ダイアログが開きます。プラグイン更新サイトの URL を入力します。URL が不明の場合は、--All Available Sites-- を使用します。

図 246. 新規ソフトウェアのインストール



インターネットへの直接接続を使用できない場合は、インターネットに接続しているコンピュータのアーカイブにプラグインをダウンロードし、その後 STM32CubeIDE がインストールされているコンピュータに手動で移動できます。アーカイブされたファイルを追加するには、Add... ボタンをクリックし、Archive and select the downloaded file を選択します。

図 247. コンピュータからの新規ソフトウェアのインストール



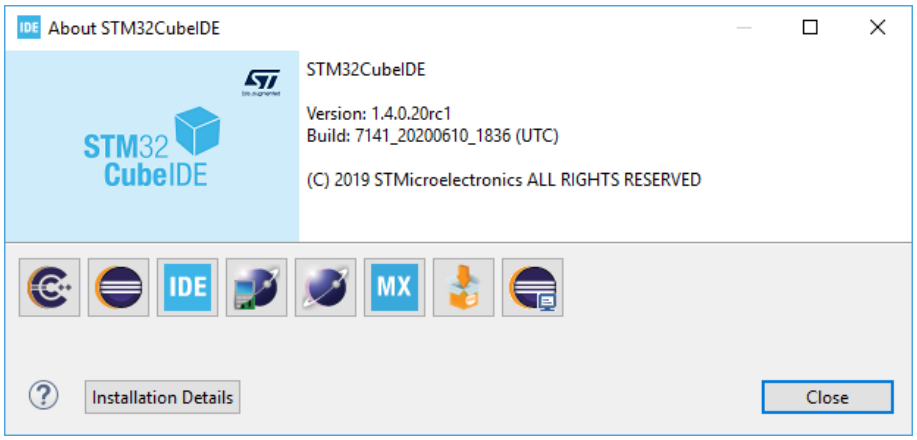
適切なプラグインを選択して、ソフトウェアをインストールします。インストールが完了したら STM32CubeIDE を再起動します。

メモ　すべての Eclipse® プラグインが STM32CubeIDE と互換性があるわけではありません。

10.4 インストール済み Eclipse® 追加プラグインのアンインストール

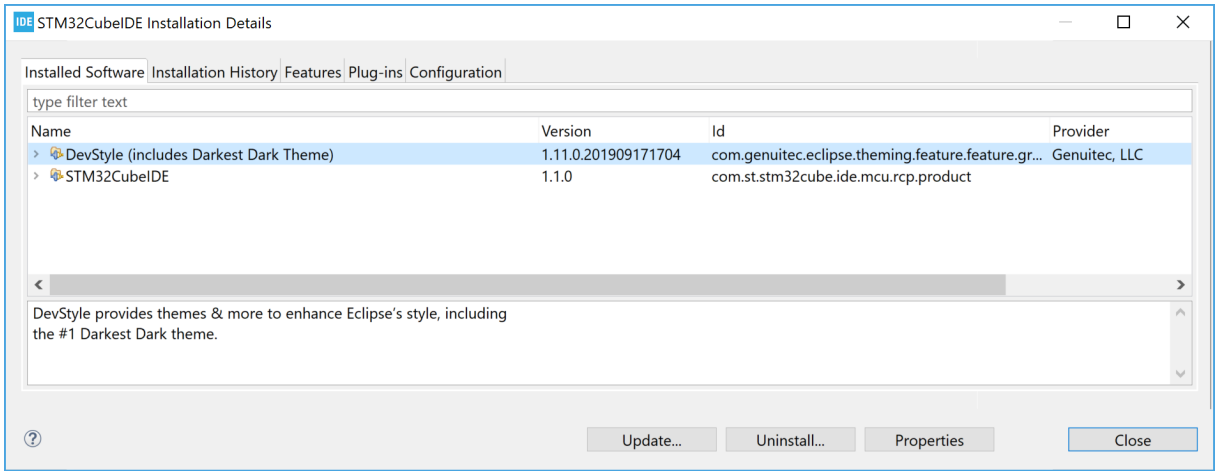
不要になったプラグインをアンインストールするには、メニュー HelpAbout STM32CubeIDE を選択します。

図 248. STM32CubeIDE について



Installation Details ボタンをクリックして、[STM32CubeIDE Installation Details]ダイアログを開きます。

図 249. インストールの詳細

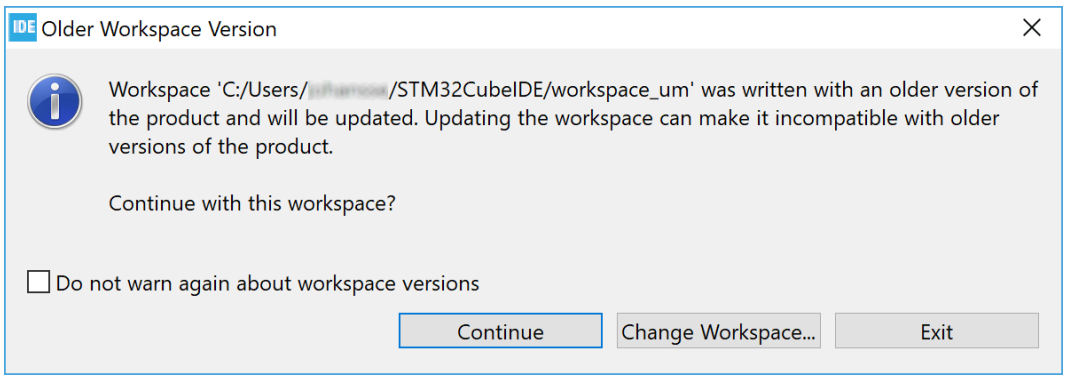


[Installed Software]タブからアンインストールするプラグインを選択し、Uninstall... をクリックします。アンインストールが完了したら、STM32CubeIDE を再起動します。

10.5 新しい CDT™ への更新

新しいバージョンの Eclipse® や CDT™（または両方）に基づく、新しいバージョンの STM32CubeIDE をインストールする場合、以前のワークスペースを使用せずに、新規に作成することを推奨します。新しい STM32CubeIDE で古いワークスペースの使用を試みると、次の警告が表示されます。

図 250. 古いワークスペース・バージョンに関する警告



11 参考文献

表 28. ST マイクロエレクトロニクスの参考ドキュメント

参照番号	ドキュメント略号	内容	ドキュメントの入手先
[ST-01]	DB3871	STM32CubeIDE データ·ブリーフ	www.st.com
[ST-02]	RN0114	STM32CubeIDE リリース·ノート	
[ST-03]	UM2553	STM32CubeIDE クイック·スタート·ガイド	
[ST-04]	UM2563	STM32CubeIDE インストール·ガイド	
[ST-05]	UM2578	TrueSTUDIO® から STM32CubeIDE への移行ガイド	
[ST-06]	UM2579	System Workbench から STM32CubeIDE への移行ガイド	
[ST-07]	UM2576	STM32CubeIDE ST-LINK GDB サーバ	
[ST-08]	STM32MP1 シリーズに基づく STM32CubeIDE プロジェクト入門(1)		wiki.st.com/stm32mpu にアクセスして「Tools」セクションの STM32CubeIDE を参照してください。
[ST-09]	AN5361	デュアルコア STM32H7 マイクロコントローラに基づく STM32CubeIDE プロジェクト入門	www.st.com
[ST-10]	AN5394	STM32L5 シリーズに基づく STM32CubeIDE プロジェクト入門	
[ST-11]	AN5564	デュアルコア STM32WL マイクロコントローラに基づく STM32CubeIDE プロジェクト入門	
[ST-12]	AN4296	IAR Embedded Workbench®、Keil® MDK-ARM、ST マイクロエレクトロニクス STM32CubeIDE、その他の GNU ベースのツールチェーンで STM32F3/STM32G4 CCM SRAM を使用する	
[ST-13]	SLA0048	STM32CubeIDE に適用される使用許諾契約	
[ST-14]	UM1718	STM32CubeMX を STM32 の設定と初期化用 C コード生成に使用する	

1. レガシー·アプリケーション·ノート AN5360 は、現在も www.st.com より入手可能です。

表 29. 外部参考ドキュメント

参照番号	内容	ドキュメントの入手先
[EXT-01]	GNU アセンブラ	GNU ツール·スイート(1)
[EXT-02]	GNU コンパイラ·コレクション	
[EXT-03]	GNU C ライブラリ	
[EXT-04]	GNU C プリプロセッサ	
[EXT-05]	GNU リンカ	
[EXT-06]	GNU バイナリ·ユーティリティ	
[EXT-07]	Red Hat Newlib C ライブラリ	
[EXT-08]	Red Hat Newlib C 数値計算ライブラリ	
[EXT-09]	Newlib nano に関する Readme	
[EXT-10]	GDB によるデバッグ	
[EXT-11]	GDB クイック·リファレンス·カード	
[EXT-12]	GNU Tools for STM32 パッチ·リスト	Information Center

1. GNU のドキュメント指針については www.gnu.org を参照してください。

改版履歴

表 30. 文書改版履歴

日付	版	変更内容
2020 年 7 月 24 日	1	初版発行
2020 年 11 月 2 日	2	STM32CubeIDE v1.5.0 向けにドキュメントを更新。 • デフォルトでは、ツールチェーンが 1 つだけインストールされる。 • [SFRs]ビューに Arm® Cortex® コア·レジスタ·ノードが表示される。 • OpenOCD によるデバッグは、SWV とライブ式をサポートする。 • 「プレファレンス - ビルド変数」を追加。 • 「Toolchain Manager」を追加。 • FreeRTOS™ に関する情報を含む「RTOS 認識デバッグ」を追加。 • 「一般的なデバッグと実行の起動フロー」を追加。 • 「makefile ターゲットによるビルド後処理」を追加。
2021 年 2 月 18 日	3	STM32CubeIDE v1.6.0 向けにドキュメントを更新。 • 「RTOS 認識デバッグ」の章に「Azure RTOS ThreadX」のセクションを追加。「FreeRTOS」のセクションを再構成。 • ローカル·ツールチェーンのサポートに関して「Toolchain Manager」のセクションを更新。 • 「プロジェクト C/C++ ビルド設定」のセクションを更新。MCU ツールチェーンの選択を移動。 • 「Information Center」を更新。 • SWV パケットという用語をドキュメント全体で更新。 • 「参考文献」を更新。 • 「セクション 4.3.3 SWV Exception Timeline Graph」を削除。
2021 年 7 月 5 日	4	STM32CubeIDE v1.7.0 向けにドキュメントを更新。 • 「セクション 2.7 空のプロジェクトや CDT プロジェクト向けのスレッドセーフ·ウィザード」を追加。 • 「セクション 3.8 STM32 Cortex-M 実行可能ファイルのインポート」を追加。 • 「セクション 6.3 RTOS カーネル認識デバッグ」を追加。 • 「Information Center - [Home]ページ」を更新。 • ヘッドレス·ビルド の説明を更新。 • セクション 2.5.6 リンカ·スクリプト に関して、メモリ·マップ·レイアウトの図を更新し、説明を追加。 • 位置独立コード の説明を更新。 • 「各種 GDB サーバ によるデバッグ」の ST-LINK GDB サーバ、OpenOCD、SEGGER に関するデバッグ設定の説明を更新。 • 「[FreeRTOS Task List]ビュー」を更新。

目次

表一覧

図一覧

重要なお知らせ（よくお読み下さい）

STMicroelectronics NV およびその子会社（以下、ST）は、ST 製品および本書の内容をいつでも予告なく変更、修正、改善、改定および改良する権利を留保します。購入される方は、発注前に ST 製品に関する最新の関連情報を必ず入手してください。ST 製品は、注文請書発行時点で有効な ST の販売条件に従って販売されます。

ST 製品の選択並びに使用については購入される方が全ての責任を負うものとします。購入される方の製品上の操作や設計に関して ST は一切の責任を負いません。

明示又は黙示を問わず、ST は本書においていかなる知的財産権の実施権も許諾致しません。

本書で説明されている情報とは異なる条件で ST 製品が再販された場合、その製品について ST が与えたいかなる保証も無効となります。

ST および ST ロゴは ST マイクロエレクトロニクスの商標です。ST の商標の詳細については、www.st.com/trademarks を参照してください。その他の製品またはサービスの名称は、それぞれの所有者に帰属します。

本書の情報は本書の以前のバージョンで提供された全ての情報に優先し、これに代わるものです。この資料は、STMicroelectronics NV 並びにその子会社(以下 ST)が英文で記述した資料（以下、「正規英語版資料」）を、皆様のご理解の一助として頂くために ST マイクロエレクトロニクス㈱が英文から和文へ翻訳して作成したものです。この資料は現行の正規英語版資料の近時の更新に対応していない場合があります。この資料は、あくまでも正規英語版資料をご理解頂くための補助的参考資料のみにご利用下さい。この資料で説明される製品のご検討及びご採用にあたりましては、必ず最新の正規英語版資料を事前にご確認下さい。ST 及び ST マイクロエレクトロニクス㈱は、現行の正規英語版資料の更新により製品に関する最新の情報を提供しているにも関わらず、当該英語版資料に対応した更新がなされていないこの資料の情報に基づいて発生した問題や障害などにつきましては如何なる責任も負いません。

この資料は、STMicroelectronics NV 並びにその子会社(以下 ST)が英文で記述した資料（以下、「正規英語版資料」）を、皆様のご理解の一助として頂くために ST マイクロエレクトロニクス㈱が英文から和文へ翻訳して作成したものです。この資料は現行の正規英語版資料の近時の更新に対応していない場合があります。この資料は、あくまでも正規英語版資料をご理解頂くための補助的参考資料のみにご利用下さい。この資料で説明される製品のご検討及びご採用にあたりましては、必ず最新の正規英語版資料を事前にご確認下さい。ST 及び ST マイクロエレクトロニクス㈱は、現行の正規英語版資料の更新により製品に関する最新の情報を提供しているにも関わらず、当該英語版資料に対応した更新がなされていないこの資料の情報に基づいて発生した問題や障害などにつきましては如何なる責任も負いません。