Renesas Synergy™ プラットフォーム

SSP モジュール開発ガイド

R11AN0132JU0102
Rev.1.02
2018.05.09

本資料は英語版を翻訳した参考資料です。内容に相違がある場合には英語版を優先します。資料によっては英語版のバージョンが更新され、内容が変わっている場合があります。日本語版は、参考用としてご使用のうえ、最新および正式な内容については英語版のドキュメントを参照ください。

要旨

Renesas Synergy™ プラットフォームを導入すると、Synergy ソフトウェアパッケージ （SSP） を構成する動作確認済みのソフトウェアモジュールやフレームワークを使用でき、ただちにソフトウェア開発が可能になります。またソフトウェア開発者は、独自に開発するモジュール、ドライバ、フレームワークをパッケージ化し、それらを他の開発者に配布することができます。このガイドでは、ソフトウェア開発者が独自にモジュール、ドライバ、フレームワークを開発するための情報を記載しています。またそれらをパッケージ化して配布するのに必要なプロセスを説明します。

対象デバイス

Synergy MCU ファミリ

本書の理解の前提となる他のドキュメント

SSP ユーザマニュアル「要旨」に相当する複数の章
SSP データシート v1.2.0 またはそれ以降
SSP Development Best Practices Guide (英語)

注記：本ドキュメントでは、上記のドキュメントを理解していることを前提に説明しています。

目的

本資料では、ソフトウェア開発者が、独自のモジュール、ドライバ、フレームワークを開発するために必要とする情報に加え、Synergy Configurator を作成してそれらをパッケージ化して、配布するために必要なプロセスを説明します。

対象とするユーザ

Synergy プラットフォームの基礎を理解し、MCU を使用して独自のドライバ、モジュール、フレームワークの作成を検討しているユーザを対象としています。

目次

1. ドライバ開発の概要（Driver Development Overview）

Renesas Synergy プラットフォームを導入することにより、モジュールやフレームワークがすぐに使用でき、Synergy Configurator と呼ばれるグラフィカルコンフィギュレータによりソフトウェアの構築が可能になります。またソフトウェア開発者は、独自のカスタムモジュール、コンフィグレーター（configurator）、ドライバ、フレームワークを、Synergy Configurator を使用して配布可能なパッケージとして作成することができます。このプロセスを実施するには、次の図に示す手順が必要です。

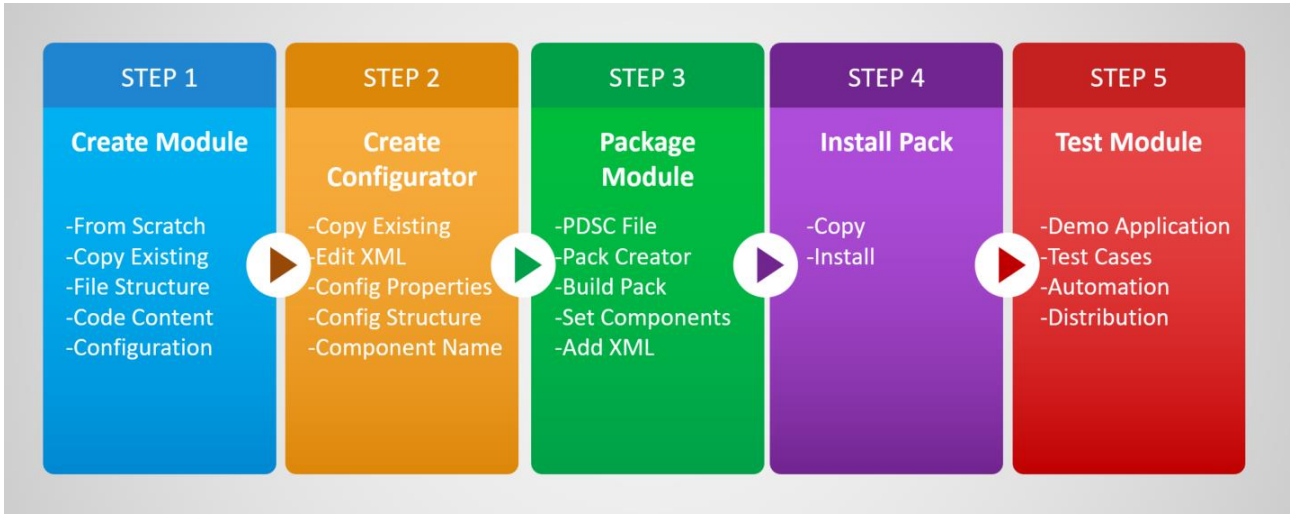


図 1 カスタムモジュールの作成手順

新規にモジュールを開発する場合と、既存のモジュールを改訂する場合で、スタート位置が多少異なる可能性があります。一般的には、SSP 内に既に存在しているモジュールを参照し、それらをコピー、ペースト、変更する形で目的を達成することを推奨しています。この手法は、プロセスの大幅な簡略化と時間短縮が図れます。

Synergy モジュールは 3 種類の情報で構成されており、それらはモジュールのパッケージ化と提供の際に必要です。

1. パック（Pack）とともに配布されるコンテンツ (コードやドキュメント)。
2. PDSC ファイル (パックデスクリプタと最終的に `.pack` となるファイル)。
3. XML コンフィギュレータ。`.module_descriptions` 内に配置されている XML ファイルです。

以下の章では、独自の SSP モジュールの開発とパッケージ化を行う際に必要な手順を説明します。このドキュメントに記載されたモジュールの作成に関する説明は、ドライバやフレームワーク作成の場合にもそのコンセプトを適用することができます。

2. ソフトウェアコンテンツの作成（Creating the Software Content）

カスタムモジュールを作成するための最初のステップは、ソフトウェアコンテンツ（software content）の作成です。ソフトウェアコンテンツには、最終的なファイルフォルダ構造（end file directory structure）、コンフィギュレーションファイル（configuration file）、ヘッダのソースファイルを含むソフトウェアモジュールで、ユーザが希望するドキュメントやプリコンパイル済みライブラリなども含まれます。この章では、ソフトウェアコンテンツを作成するための推奨事項や方法について説明します。

2.1 モジュールのファイル構造の編成（Module file structure organization）

ソフトウェアのコーディングやコピーを開始する前に、新しいモジュールフォルダ構造（module's directory structure）を作成する必要があります。ここで作成するフォルダ構造は、最終的にパッケージ化されたモジュールが、Synergy プロジェクトのどこにコピーされるかを決定します。e²studio 内の Synergy プロジェクトを表すローカルファイル構造を作成することを推奨します。例えば、`sf_example` という名前の新しい Synergy フレームワークモジュールを作成する場合、次の図に示すように、`synergy/ssp/src/framework/sf_example` の下に、そのフレームワークに対応するフォルダを作成します。これは必須ではありませんが、この方法に従うとパックの作成プロセスが容易になります。

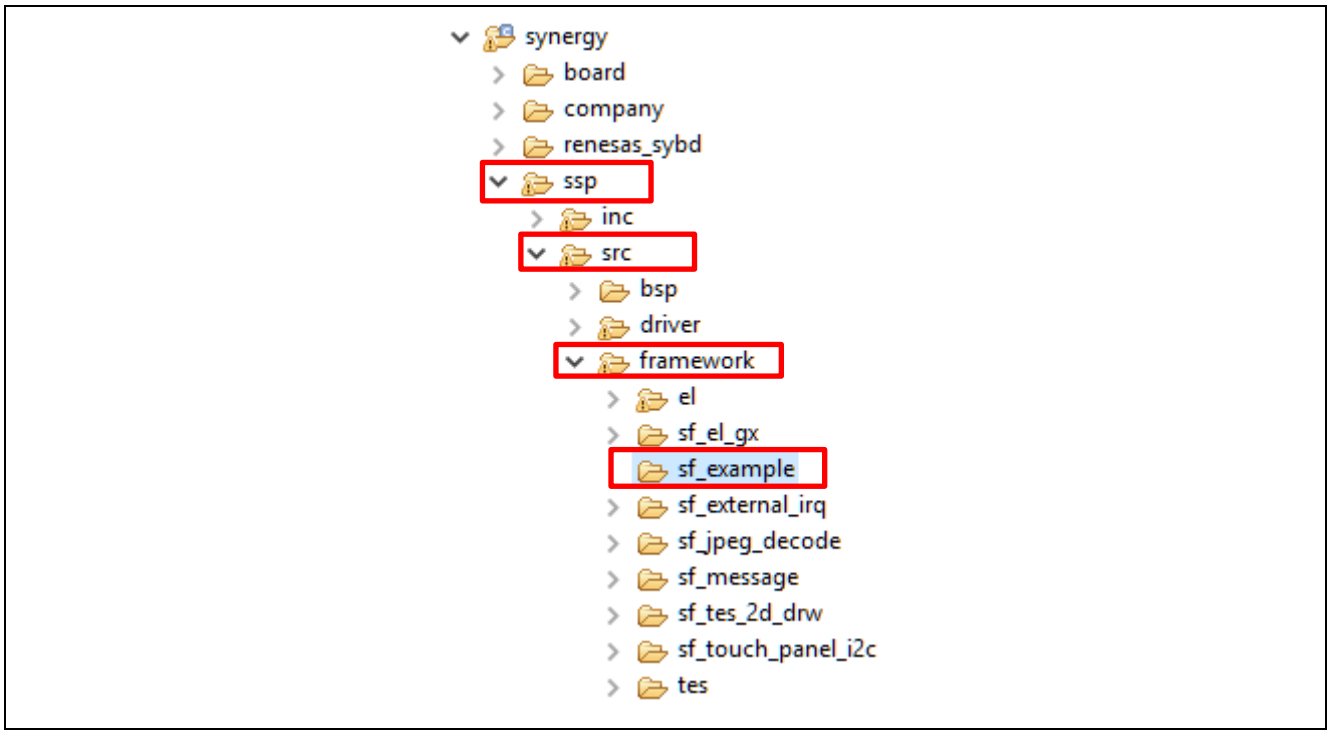


図 2 ソフトウェアフレームワークのフォルダ構造に関する例

サードパーティのソフトウェア開発者の場合、SSP モジュールを作成せず、独自のカスタムモジュールとカスタムフレームワークを作成することを推奨します。SSP コンポーネントとサードパーティコンポーネントを明確に区別するために、`synergy/ssp` とは異なる別のフォルダにモジュールを配置すること、すなわち `synergy/<company>` のような使用を推奨します。この場合、前述の `sf_example` コードは、次の図に示すように、`synergy/<company>/sf_example` の下に配置されます。`sf_` というフォルダ名の先頭テキストの使用は任意ですが、テキスト sf_はソフトウェアフレームワークレイヤモジュール（software framework layer module）であることを表す、SSP の標準手法となっています。

カスタムモジュールが最終的にパッケージ化されたとき、そのパッケージの一部として提供されるいずれのコンテンツも、プロジェクト内では読み取り専用になります。これは標準 SSP モジュールと同様です。このカスタムモジュールがプロテクト (暗号化)されて いない場合は、プロジェクト内でそのカスタムモジュールは変更が可能です。ただし変更後にこのモジュールを元のモジュールと同じフォルダに配置した場合は、ビルドまたは **[Generate Project Content]** ボタンをクリックしたときに、元のモジュールがパッケージから再度解凍（re-extracted）され、モジュールに対して加えた変更は全て上書きされ（overwritten）、失われます。上書き変更は、パッケージから生成したコードのみが対象になります。カスタムモジュールを開発する場合は、Synergy フォルダ内に別の独自のファイル構造やコードを作成することができます。このようにすれば、開発者が行う変更はパッケージが作成されるまでは上書きされません。

注記：パッケージの作成、インストール、プロジェクトへの追加を行った時点で、開発者のフォルダは上書きされます。作業データの消失防止のため、開発者はこまめにバックアップを行うよう注意してください。

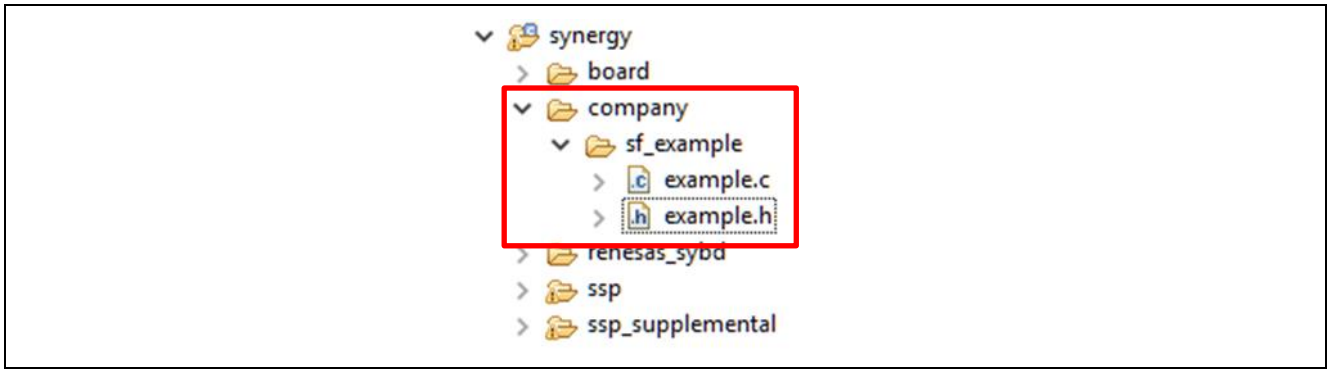


図 3 SSP 以外のモジュールに対応するカスタムモジュールの場所

synergy_cfg フォルダ内にあるすべてのコンテンツは通常、パッケージの一部として提供されません。この理由は、Synergy Configurator が作成する値を基に e² studio が synergy_cfg フォルダのコンテンツを作成するためです。パッケージから解凍されるコンテンツは読み取り専用であり、synergy_cfg で設定ファイルを作成すると、プロジェクトに問題が発生する可能性があります。デフォルトのコンフィギュレーションを使用し、ツールチェーンが生成する設定を使用するのが最善です。後の章で、このプロセスについて詳細に説明します。

2.2 新規に詳細の設定をする方法（Filling in the details from scratch）

開発者は既存のコードをできるだけ活用し、新しいモジュールの開発に費やす時間、コスト、労力を減らす必要があります。場合によっては、既存のドライバをコピーし、コピー後のドライバに機能追加やカスタマイズを行うこともできます。一方、既存のコードを活用せずに、新規にモジュール開発する場合もあります。後者の場合は、以下の手順で開発を進めてください。

1. synergy_module_template.zip (または任意の Application Project zip フォルダ) を解凍します。
2. サンプルのインタフェースファイル（Interface file）の名前を変更し、必要な変更を加え、移動先フォルダに移します。例：synergy/ssp/inc/framework/api/sf_example_api.h
3. サンプルのインスタンスファイル（Instance File）の名前を変更し、必要な変更を加え、移動先フォルダに移します。例：synergy/ssp/inc/framework/instances/sf_example.h
4. サンプルソースフォルダ内にあるソースファイル（source file）の名前を変更し、必要な変更を加え、移動先フォルダに移します。例：synergy/ssp/src/framework/sf_example。

2.3 既存のモジュールを活用し詳細の設定をする方法（Filling in the details from an existing module）

CRC HAL Driver モジュールなど、既存のモジュールを利用する場合、以下の手順を進めてください。

1. スタートしたいモジュールで、Synergy プロジェクトを作成します。
2. 既存モジュールのソースフォルダのコピーを作成し、コピー先フォルダの名前を変更します。
   — 例えば、synergy/ssp/src/driver/r_crc フォルダを、synergy/ssp/src/driver/r_mydriver というフォルダにコピーします。
3. 名前を変更したフォルダの中のソースファイルの名前も変更します。
   — 例えば、synergy/ssp/src/driver/r_crc/r_crc.c を synergy/src/driver/r_mydriver /r_mydriver.c に変更します。
4. コピー先フォルダのソースに変更を加えます。
5. 既存モジュールのインスタンスヘッダファイル（Instance header file）をコピーします。
   — 例えば、synergy/ssp/inc/driver/instances/r_crc.h を synergy/ssp/inc/driver/instances/r_mydriver.h にコピーします。
6. コピー先の新しいインスタンスヘッダファイル（Instance header file）に変更を加えます。
7. 必要に応じて、新しいフレームワークモジュールの中でこの手順を繰り返します。

この時点で、パッケージ化の作業用ソフトウェアコンテンツが作成できたことになります。モジュールのパッケージ化を行う前に、Synergy Configurator を使用したモジュールのプロパティ（property）の設定を容

易にするため、XML コンフィギュレータファイル（XML configurator file）を作成する必要があります。次の章では、XML ファイルの作成方法を説明します。

3. ISDE に対応するモジュールコンフィギュレータ XML ファイルの作成（Creating a Module Configurator XML File for the ISDE）

3.1 モジュールコンフィギュレータの概要（Module configurator overview）

XML コンフィギュレータファイルを使用すると、Synergy Configurator の **[Threads]** タブに新しいモジュールが表示されます。このタブから、モジュールをアプリケーションスレッド（application thread）に追加することができ、**[Properties]** ビューを使用した設定が容易になります。

次の図は、XML コンフィギュレータファイルの構成と、ISDE（統合ソリューション開発環境）や生成コードとの関係を示しています。モジュールコンフィギュレータ（module configurator）の XML ファイルは、`.module_descriptions` フォルダ内にあります。このフォルダは ISDE で表示されませんが、プロジェクトフォルダ内に存在します。XML データには、`config` 要素（element）と `module` 要素（element）の 2 つの主な要素があります。`config` 要素は、アプリケーションレベルでのモジュールの動作に影響する設定を持つのに対して、`module` 要素は具体的なインスタンスの動作を規定します。

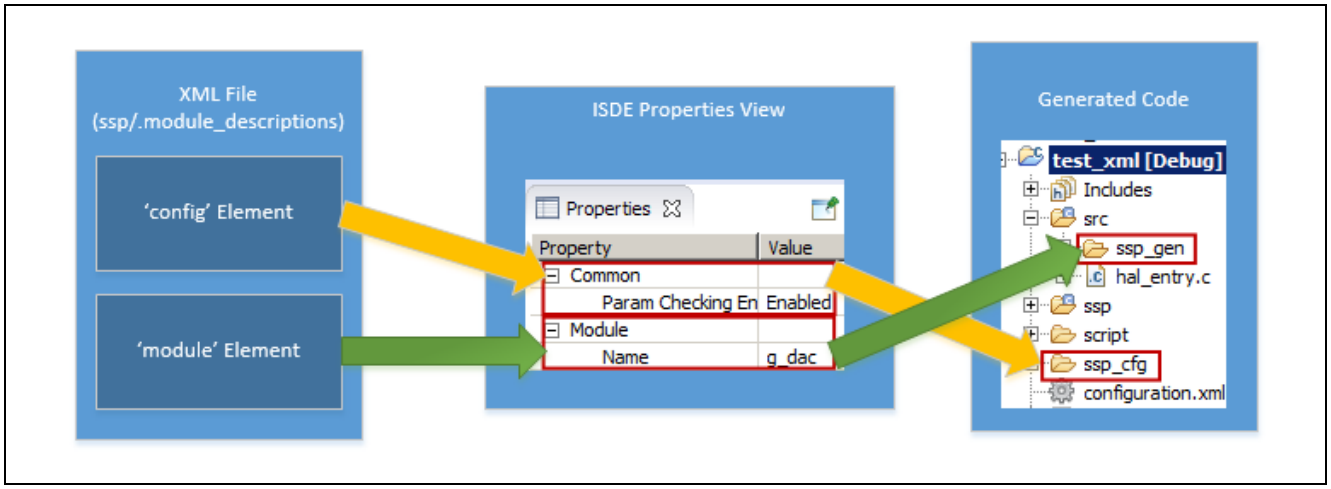


図 4 XML ファイルからの設定情報の生成

[Threads] タブ内でモジュールをプロジェクトに追加した後に、開発者がそのモジュールをクリックすると、ISDE 内で XML データは 1 つの GUI として表示されます。設定の値は、**[Properties]** ビューで利用できます。XML データは、2 つのプロジェクト領域内に複数のファイルを生成します。第 1 に、module 要素のデータから `src/synergy_gen` フォルダ内のコードが生成されます。これは、インスタンス固有の設定であり、上の図では緑の矢印によって表示されています。第 2 に、`config` 要素のデータから、`synergy/ssp_cfg` フォルダ内にコードが生成されます。これはビルド時の共通の設定であり、上の図では黄色の矢印によって表示されています。

ISDE は、データドリブンで柔軟性の高いモジュールコンフィギュレーションツール（Module configuration tool）を持っています。開発者は、モジュールのコンフィギュレータ XML ファイルをカスタマイズ可能です。その後、このファイルは Eclipse の標準的な **[Properties]** ビューを使用してプレーンテキストで表示できます。このコンフィギュレータを解釈し、コンフィギュレーションフィールドを設定することは容易です。次の図に、開発中のモジュール向けに作成できるサンプルのコンフィギュレータと、この設定情報を使用してコードを生成する方法を示します。

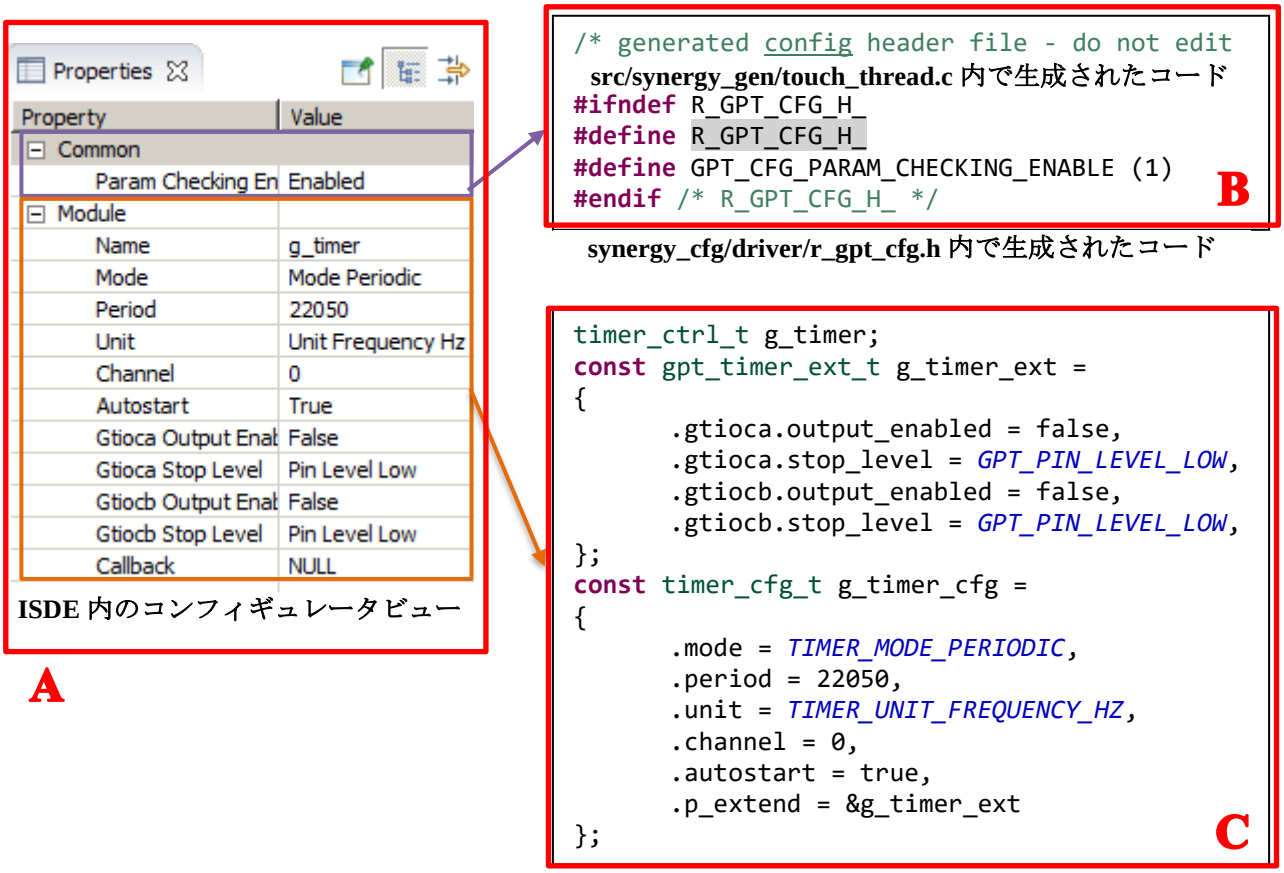


図 5 設定ヘッダとソースコードを生成する ISDE 内のコンフィギュレータビュー

設定が完了した後、**[Generate Project Content]** ボタンをクリックするとモジュールが必要とするパラメータが生成されます。さらに、プロジェクトの src/synergy_gen フォルダ内にモジュールのインスタンスが作成され、また synergy_cfg フォルダ内に共通コードが作成されます。上図 A では 例として、touch_thread スレッド内にある audio フレームワークの一部として使用されるタイマモジュールを示しています。プロジェクトのコンテンツを生成した後、図 B に示したように設定ヘッダファイル内に共通の設定プロパティが生成されます。また、上図 C では、コードモジュール内でモジュールの設定プロパティが生成されています。

注記：ソフトウェアのプログラムを作成する場合、モジュールをテストする目的でこれらの設定値に対応する独自の XML ファイルを記述することもできます。その後、そのコードを使用してコンフィギュレータ XML ファイルを開発し、さらに他のユーザ向けのコンフィギュレータを作成することもできます。

3.2 XML の作成（Creating the XML）

XML コンフィギュレータを作成するもっとも簡単な方法は、既に存在しているコンフィギュレータを使用することです。この場合のプロセスは以下の通りです。

- 類似の設定要素（configuration elements）を持つ XML ファイルを見つけます。
- その XML ファイルのコピーを作成します。
- 次の章「XML ファイルの命名規則」の規則に従ってコピーした XML ファイルの名前を変更します。
- 新しいモジュールに合わせて、XML ファイルを編集、更新します。
- ピアレビュー (peer review：他の開発者による確認) やテストを実施し、ファイルを正しく更新できたことを確認します。

例えば、r_crc モジュールをベースとして独自モジュールを作成する場合、既存モジュールのインスタンスに対応する XML コンフィギュレータファイル（configuration file）をコピーし、ファイル名を変更した後、上記プロセスに従って編集することができます。

この場合、以下の作業が必要になります。

- .module_descriptions/Renesas##HAL Drivers##all##r_crc####1.2.0.xml をクリップボードにコピーする。
- ペーストを行い、.module_descriptions/Renesas##HAL Drivers##all##r_mydriver####1.2.0.xml という名前に変更する。

次の章で、これらの手順と XML ファイルの規則について説明します。

3.3 XML ファイルの命名規則（XML file naming convention）

ソフトウェアスタック（software stack）をビルドして、Synergy モジュール向け実行コードを生成する機能は、XML コンフィギュレータにより実現されます。これらは、プロジェクト内の `.module_descriptions` フォルダに存在します。これら XML ファイルは、プロジェクトを作成する際に、特定のバージョンで利用できるパッケージすべてから解凍されます。また、これらの XML ファイルは **[Threads]** タブ内にある [new stack] ボタンの下にどのようなオプションが表示されるか決める役割も果たします。**XML コンフィギュレータのファイル名は、PDSC ファイル内にある <component> 要素（component）の設定と一致している必要があります**。次の図に、PDSC ファイル内にある component 要素の例を示します。

```
<component Cclass="Company"
           Cgroup="all"
           Csub="sf_example"
           Cvendor="Renesas"
           Cversion="1.2.0"
condition="">
```

図 6 パック内の PDSC ファイルの中にある component 要素

この XML ファイルは、Renesas##Company##all##sf_example####1.2.0.xml と命名する必要があります。ここで、色の付いた文字（text：テキスト）は、PDSC の component 要素によって提供された情報に対応しています。これらオプションのいずれかが一致していない場合は、component は **[Components]** タブ内に表示されますが、Synergy Configuration の **[Threads]** タブ内にある新しいスタックツリー（stack tree）の下には表示されません。

各モジュールの XML 設定ファイル（XML Configuration File）に割り当てられる名前によって、**[ISDE components]** タブ内でのモジュールの表示場所が決まります。例えば、コンフィギュレータ名を次のようにすると、Synergy Configurator の **[Components]** タブ内で、コンポーネントは次の図のように表示されます。

`Renesas##Flashloaderv1.0.0##sf_bootloader####1.2.0-b.1.xml`

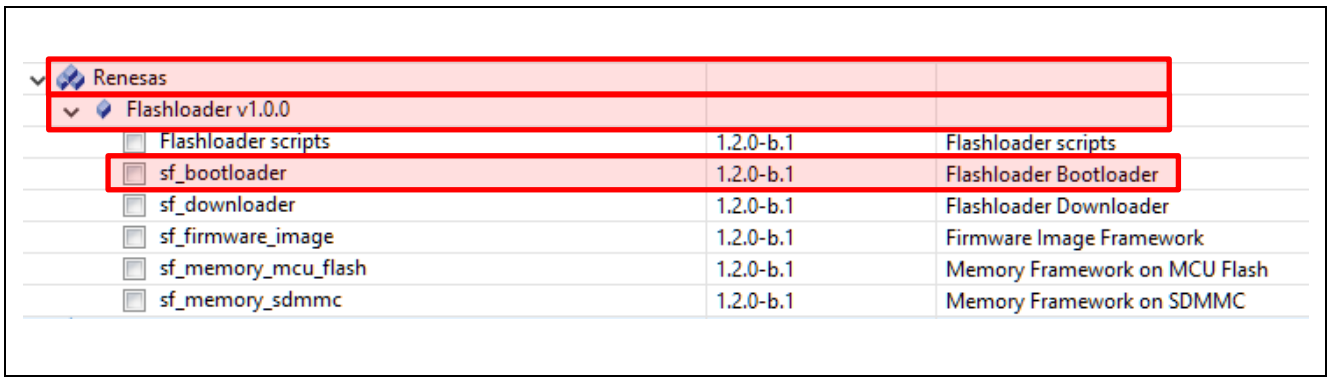


図 7 XML ファイルの命名方法により表示位置が決まる例

サードパーティがモジュールを開発している場合は、次のテンプレートを使用してコンポーネントに名前を付けることができます。

`Renesas##Company##FrameworkName##all##Module####Target_SSP_Version.xml`

名前の「####」より後に表記される最後の部分は、このコンポーネントに指定ささ れ、コンポーネントと組み合わせて動作することがテストされる SSP のバージョンを使用します。

コンフィギュレータ XML ファイル（configurator XML file）に命名する際に従うべき、 2 つのルールを以下に示します。

1. <components> の Cvendor 属性（attribute）を、「Renesas」に設定する必要があります。ルネサスエレクトロニクスやそのグループ会社に属されない方がモジュールをビルドする場合も、必ずこのように設定してください。この属性を適切に設定しない場合、<component> が e² studio の **[Components]** タブ内に正しく表示されません。
2. 作成する XML コンフィギュレータ（XML configurator）の名前は、「Renesas」で始まり、<component> の Cvendor 属性に一致している必要があります。

3.4 モジュールのコンフィギュレータ XML ファイルのセクションとタグ（Module configurator XML file section and tags）

コンフィギュレータ XML ファイルを作成した後は、コンフィギュレータ XML ファイルを構成しているさまざまなセクションについて理解しておくと便利です。モジュールのコンフィギュレータ XML ファイルは、config および module という 2 つの主要要素に区分できます。主な違いは、config 要素のサブ要素が (synergy_cfg フォルダ内にある) どのモジュールのインスタンスでも反復されているのに対し、module 要素のサブ要素はモジュールの単一のインスタンスに属していることです。例えば、あるアプリケーションが 2 個の GPT タイマチャネル（timer channels）を使用している場合、チャネルごとの設定 (timer_cfg_t 構造体を定義するために使用される)が、XML ファイル内にある module 要素のサブ要素であるのに対し、共有設定 (synergy_cfg/driver/r_gpt_cfg.h 内のマクロを定義するために使用される) は、XML ファイル内にある config 要素のサブ要素になります。

各セクションは、モジュールの動作を記述するさまざまな XML タグ（XML tag）を有しています。どの設定情報を表示するか、どのモジュールと依存関係があるか、どのインタフェースが提供されるか、などを記述します。次の表に、利用可能な XML タグ（XML tag）を示し、各タグの使用方法に関する説明を掲載します。

これらのタスクを完了させるために、サンプルの XML タグをコピーして、自らのコンフィギュレータに貼り付けることができます。これらのタグの使用方法の詳細とサンプルについては、7 章　付録 - モジュールコンフィギュレータ XML ファイルに関するルール（Rules for the Module Configurator XML File）　を参照してください。

表 1　　XML タグ

XML タグ	目的
<requires>	モジュールが動作するうえで必須のコンポーネントを識別します。これらはコンポーネントの依存関係を表します。
<provides>	このコンポーネントに依存しているモジュールに対して、コンポーネントが何を提供するかを表します。
<constraints>	このモジュールが機能するうえで満たさなければならない制約を定義します。例えば「インスタンスは固有の名前を持っていなければならない」などです。
"Name"	重複エラーを避けるために、データ構造体に対して固有（Unique）な名前を割り当てます。
<override>	依存関係オプションをハードコーディングする場合に、<requires> 内で使用することができます。
<property>	開発者が設定することのできる設定オプションを作成します。
<option>	<property> タブ内で表示される、さまざまな設定ドロップダウンオプションを提供します。
<config>	すべてのインスタンスにまたがって適用されるモジュールに存在する、高水準の設定オプションを定義します。
<module>	モジュールの単一のインスタンス（single module instance）に対して適用される要素を定義します。

<header>	インスタンス構造体など、開発者が自らのコード内で使用する extern (外部) グローバル変数を定義します。
<includes>	必須のインクルードパス（include paths）を記述するもので、src/synergy_gen 内で生成されたヘッダファイル（header file）に対して直接コピーされます。
<declarations>	open call の際に必須となるデータの宣言を記述します。データはプライベート C ファイル（private C file）へとコピーされ、開発者からアクセスできるヘッダファイルへと展開されます。
<init>	open 関数（open function）を呼び出すためのコードを記述します。これは、開発者のスレッドに合わせて生成されたコードであり、開発者が作成したスレッドのコード (entry 関数) より先に実行されます。

3.5 XML ファイル内で 1 個のモジュールに対応する複数のコンフィギュレータ（Multiple configuration per Module in XML files）

モジュールの作成を開始した段階で、単一のモジュールに対して複数のコンフィギュレータを作成する方法が必要な場合があります。NetX™ モジュールはその良い例で、単一のモジュールに対して以下の機能の 3 つの基本的なコンフィギュレータ（configurator）が存在します。

- 共通のコアコード（The common core code）
- IP インスタンスの作成（Creating IP instance）
- パケットプールインスタンスの作成（Creating Packet Pool Instance）

これら 3 つのコンフィギュレータはいずれも、SSP 内の nx フォルダから取得した同じコードを使用します。単一のモジュールで複数のコンフィギュレータを有効にするには、同じ XML ファイルの中で、追加の <config> 要素と <module> 要素を作成する必要があります。この事例として、次のファイル内を参照してください。

```
Renesas##Framework Services##all##sf_i2c####x.xx.xx.xml.
```

この XML ファイルには、2 つの <module> 要素と、2 つの <config> 要素があります。一方は（Shared Bus） 向け、もう一方は デバイス（Device） 向けです。

3.6 モジュールコンフィギュレータのチェックリスト（Module configurator checlist）

コンフィギュレータ XML ファイル（configurator XML file）はシンプルなものですが、複雑になりがちです。次のチェックリストを使用して、必要な項目すべてが記述されていることを確認してください。

項目	はい/いいえ
Config プロパティ（Property） - パラメータのチェックが必須です。チェックしましたか？	
Module の説明 - ユーザが特定のプロパティをクリックしたときに、ユーザに対して提示される説明がありますか？	
Module が interface 要素（element）を示していますか。	
Module で、オーバーライド（override）が有効な interface 要素が必須になっていますか？(該当する場合)	
Module が以下の状況で要素（element）をオーバーライド（override）しますか？ 開発したモジュールが<requires> 要素で指定した下位レベルモジュール（lower level module）を必要とし、なおかつ開発したモジュールが特定の設定値（setting）を必要とする場合に、下位レイヤ（lower layer）に対してこの設定値を強制的に適用するために、 <requires> 要素の <override> サブ要素を使用しますか？ <override property="module.driver|framework.<lowerlevelapi>.<lowerlevelid>" value="module.driver|framework.<lowerlevelapi>.<lowerlevelid>.<lowerlevelvalue> "/> 次に、sf_audio_playback_hw_dac フレームワークに対応する例を下記に示します。ここでは、DAC を対象とする上位レベル（upper level）の Audio Playback が、下位レベルの DAC モジュールに対して、flush_right (右寄せ) というデータ形式の使用を強制的に適用しています。 <requires id="module.framework.sf_audio_playback_hw_dac.requires.dac" interface="interface.driver.dac" display="Add DAC Driver" > <override property="module.driver.dac.data_format" value="module.driver.dac.data_format.data_format_flush_right"/> </requires>	
Property 要素	
モジュールの Property 要素（element）	
モジュールの Header 要素（element）	
モジュールの Includes 要素（element）	
モジュールの Declarations 要素（element）	

下位レベルドライバ（lower level driver）を使用する各モジュールでは、以下の項目が必須です。

項目	はい/いいえ

Module
「1」のみが許容される ThreadX Source の例
`<module config="config.el.tx_src"`
`id="module.framework.tx_src"`
`display="Framework|RTOS|ThreadX Source"`
`common="1"`
`version="0">`
共有されるモジュールンスタンスの数として、無制限を意味する値 ("100" は実質的に無制限) が許容される、I2C フレームワーク共有バスの例
`<module config="config.framework.sf_i2c_bus"`
`display="Framework|Connectivity|${module.framework.sf_i2c_bus.name } I2C Framework`
`Shared Bus on sf_i2c"`
`id="module.framework.sf_i2c_bus_on_sf_i2c"`
`common="100"`
`version="1">`
Constraint 要素（element）	

モジュールの Requires interface 要素（element）	
モジュールの Init 要素 12X	

3.7 モジュールコンフィギュレータのディクショナリ（Module configurator dictionary）

コンフィギュレータ XML （configurator XML）のプロセス（process）の一環として、XML が正しく表示されるように、特定の文字を特定の形式で書き込む必要があります。例えば、">" という記号を使用することはできず、代わりに ">" を使用する必要があります。次の表に、XML ファイル（XML file）内で使用する代替文字のリストを示します。

名前	カテゴリ	説明
===	Javascript	C 言語の "==" と同様に、制約が等しいかどうかを判定する目的で使用します
||	Javascript	C 言語の '||' と同様に、制約の一部として使用します
&	Javascript	XML が生成したコードの中でアンパサンド (&) 文字（ampersand character）を形成するには、`&amp;` を使用します
& &	Javascript	C 言語の '`&&`' と同様に、制約の一部として使用します
>	Javascript	XML が生成したコードの中で大なり不等号 (>) 文字を形成するには、`&gt;` を使用します
<	Javascript	XML が生成したコードの中で小なり不等号 (<) 文字を形成するには、`&lt;` を使用します
"	Javascript	XML が生成したコードの中で二重引用符 (") 文字を形成するには、`&quot;` を使用します
config	要素（element）	ビルド時に使用した設定のプロパティ（property）を記述しています。これらのプロパティは、`ssp_cfg/<driver|framework>/r_<module>_cfg.h` に反映されます。属性の id、パス、バージョンを記述する必要があります。
config	属性（attribute）	モジュールの属性（attribute of module ）であり、そのモジュールの `config` 要素に対応する id 属性と同じ値であることが必要です
constraint	要素（element）	無効な設定（configuration）に制約を加えるためのフレームワークです
display	属性	ユーザに対して表示されるテキスト
declarations	要素（element）	`ctrl` および `cfg` というデータ構造体（data structure）を割り当てられたテキストフィールド（text field）
header	要素（element）	extern 化 (外部化) したグローバル変数（global variables ）を持つテキストフィールド (例：`extern <api>_ctrl_t <module_user_name>;`)
id	属性（attribute）	XML 内で使用する変数です。`${<id>}` と表記すると、使用時点における value パラメータと同じ値に解決されます。例：`${module.driver.timer.unit}` という表記は、(`${module.driver.timer.unit.unit_frequency_khz}` に解決されます。その後、この表記は) `TIMER_UNIT_FREQUENCY_KHZ` に解決されます。このような解決が実施されるのは、GPT タイマ設定の **Unit** ドロップダウンで **Unit Frequency Khz** (単位周波数 kHz) オプションが選択されている場合です。また、この場合、どのテキストフィールドでもユーザに対してこの解決後の値が表示されます。id 属性は、親要素の id テキストで始める必要があります。例：`${module.driver.timer.unit}` は、親モジュールである `${module.driver.timer_on_gpt}` の unit プロパティに対応する id です。
includes	要素（element）	必須のインクルードパス（include paths）を持つテキストフィールド (例：`#include &quot; r_<instance>.h&quot;`)
init	要素（element）	open 関数を呼び出すためのコードを記述するテキストフィールドです。現在はフレームワークレイヤでのみ使用されています。`<user_thread_name>.c` 内で呼び出された後、`<user_thread_name>_entry` が呼び出されます
interface	属性（attribute）	provides と requires の各要素内で、複数のモジュールを互いに結び付ける目的で使用します
macros	要素（element）	マクロを定義するためのテキストフィールド。現在は使用されていません。
module	要素（element）	open 関数に渡す `<user_specified_name>_cfg` 構造体の一部として作成した実行時設定のプロパティを記述します。`config`、`display`、`id`、`version` の各属性を記述する必要があります。
option	要素（element）	特定のプロパティに対応するドロップダウンオプション (常に、property のサブ要素)。各 option で、`display`、`id`、`value` の各属性を記述する必要があります。
property	要素（element）	ユーザが指定する必要のある設定。property は、`config` と `module` の各要素内に記述されています。各 property で、`default`、`display`、`id` の各属性を記述する必要があります。option が何も指定されていない場合は property はテキストフィールドであり、option が指定されている場合は property はドロップダウンです。

provides	要素（element）	ドライバに結び付けるためのインタフェースを上位レイヤ（upper layer）に提供します。また、各チャネルのインスタンス（instance）が 1 個のみ使用されていることを保証するために、constraint が provides を使用します。
requires	要素（element）	下位レベル（lower level）のドライバに対応するインタフェースを必要とするモジュール (通常はフレームワークモジュール) が使用します。
override	要素（element）	`requires` 要素内にある下位レベル（lower level）の設定値をロックします。
value	属性（attribute）	`${<id>}` を解決した（resolved）ときに得られる値を指定します。
common	属性（attribute）	`<module>` 要素内でのみ有効です。この属性（attribute）は、複数の SSP スタック間で単一の module インスタンス（instance）を共有できるかどうかを決定します。

4. 新しいソフトウェアモジュールのパッケージ化（packaging the New Software Module）

4.1 PDSC (パックデスクリプタ)（pack descriptor）

e² studio と IAR Embedded Workbench for Synergy の中でユーザにコンテンツを提供する目的で、CMSIS-Pack を使用します。CMSIS-Pack に関する情報は次の場所に掲載されています。

http://www.keil.com/pack/doc/CMSIS/Pack/html/index.html

CMSIS-Pack は、次の 2 つのパートで構成されています。

- 提供されるコンテンツ (例：コード、ドキュメント)
- コンテンツについて記述する PDSC (パックデスクリプタ) ファイル

次の図は、PDSC ファイルに含まれるべきさまざまな情報を示します。

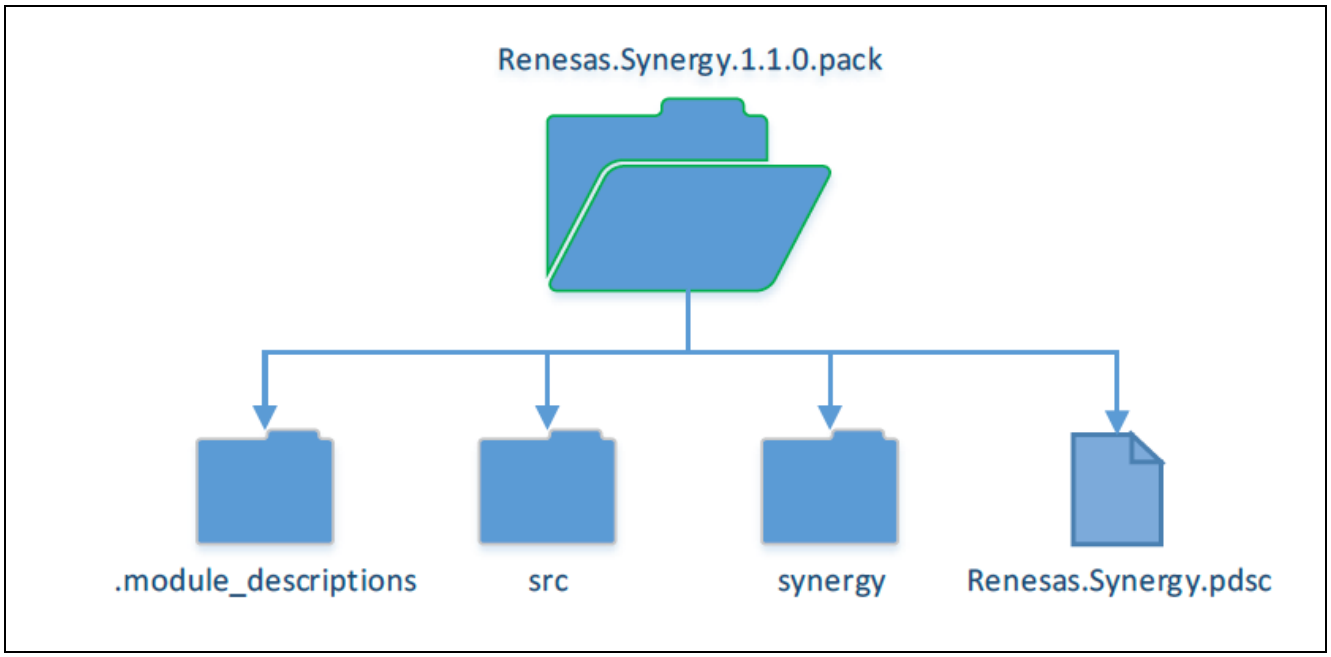


図 8 パックファイルの例

CMSIS-Pack は、`.pack` 拡張子を付けた zip ファイルです。アーカイブ済みファイルのルート（root）に位置するのは、`.pdsc` 拡張子が付いたファイルです。PDSC ファイルの名前は、パックファイル名（Pack Filename）からバージョン番号を取り除いたものと同じ名前にする必要があります。パックに名前を付ける書式は、次の通りです。 `<vendor>.<name>.<version>.pack`。これにより、SSP パックのファイル名は `Renesas.Synergy.1.2.0.pack` のようになります。

PDSC ファイル内で、関連するスキーム（scheme）を指定する XML の詳細は、次の http で説明されています。http://www.keil.com/pack/doc/CMSIS/Pack/html/packformat.html

e² studio 内の、特に カスタムパック作成ツール（Custom Pack Creator Tool） のエクスポート機能を使用して、PDSC ファイルを自動的に生成することもできます。そのため、PDSC ファイルの詳細な説明は省略します。

4.2 カスタムパック作成ツール - e² studio（Custom Pack Creator Tool - e² studio）

SSP （Synergy Software Package）と、Synergy プラットフォームに関連するアドオンソフトウェア（add-on software）は、パック形式で配布されます。パックは、ソフトウェアをとりまとめて開発者へ配布するための便利な方法です。パックは、インストールフォルダ（installation directory）の、`\e2_studio\internal\projectgen\arm\Packs` の下に配置されます。カスタムモジュール（custom module）とフレームワーク（framework）も、同じ方法で配布される必要があります。e² studio 5.2.1 以降では、カスタムパック（custom pack）のエクスポートは比較的簡単なプロセスです。

注記：カスタムパック作成ツール（Custom Pack Creator Tool） が利用できるのは、e² studio バージョン 5.2.1 以降です。IAR EW for Synergy のバージョン 7.71.1 では、この機能は利用できません。

パックファイル（pack file）を作成するには、カスタムコード（custom code）を含むプロジェクトを開き、**[File]** メニューから **[Export]** を選択します。表示されるダイアログボックスを、次の図に示します。

[General] の下にある **[Renesas Synergy User Pack]** オプションを選択し、**[Next]** をクリックします。

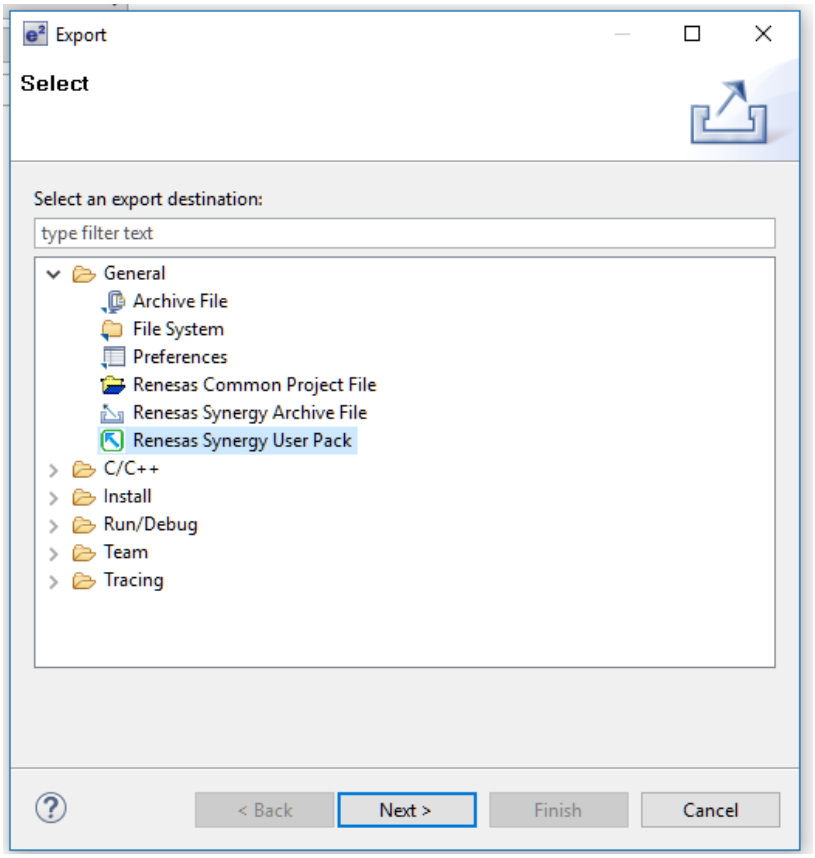


図 9 パックへのエクスポート

次に表示されるダイアログボックスでは、パック名に加えて、バージョン、説明、連絡先情報など、パックに関する他の様々なパラメータも指定できます。この段階で、次の図に示す様々な詳細を入力することができます。この最初の画面には、パックに名前を付ける機能があることに注意してください。既に説明したように、`<Company>.<Component>.<SSP Version>` の形式でパックに名前を付ける必要があります。次の図では、生成されたパックの名前（file name）を中央部の赤枠で強調して表示しました。

デフォルトでは、パック情報は保存されません。パックをテストし、変更を加えた後に、別のパックを作成する場合は、Pack Creator の Save 機能と Open 機能を使用する必要があります。これらの機能は次の図の右上隅にあり、赤枠で強調して表示しました。

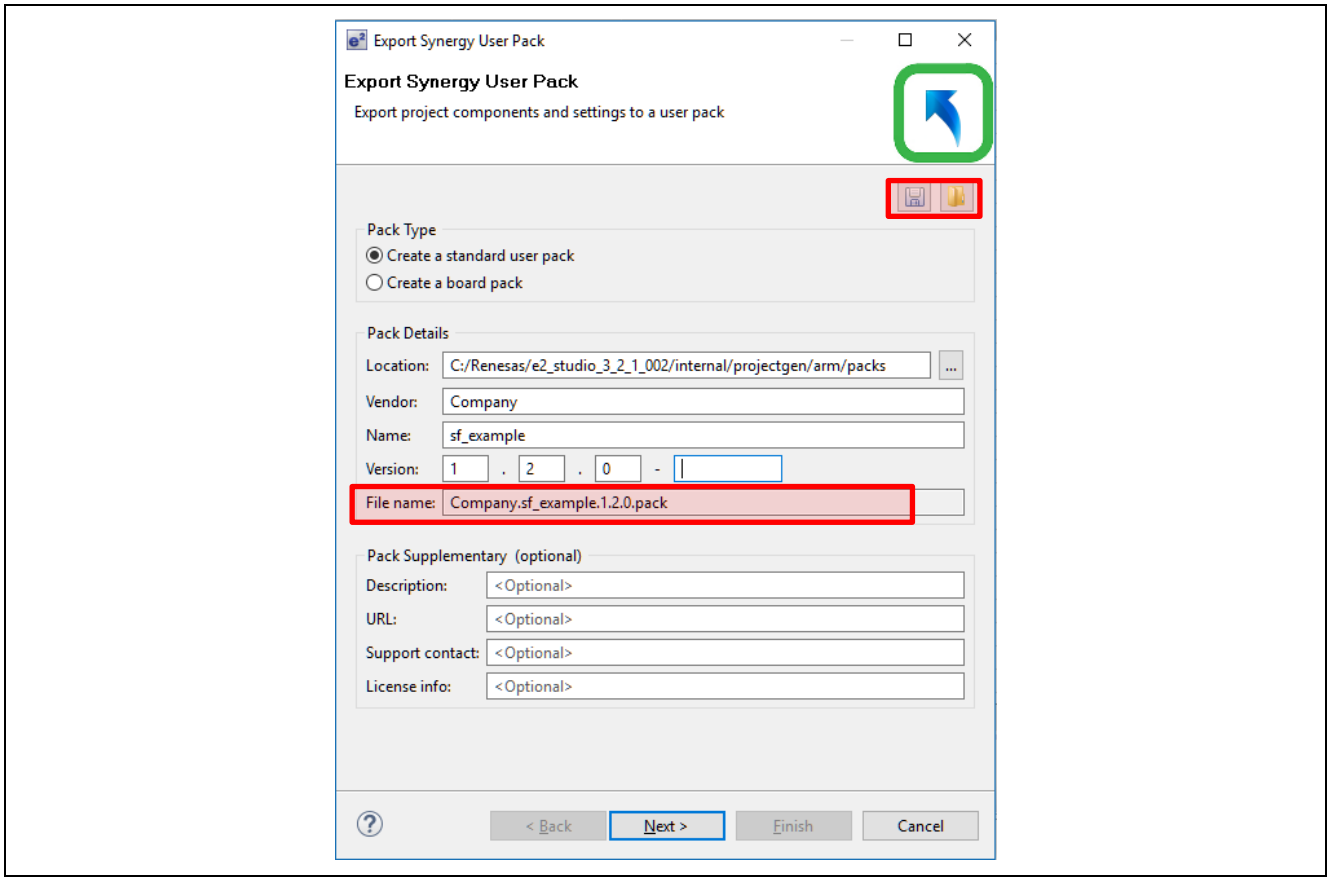
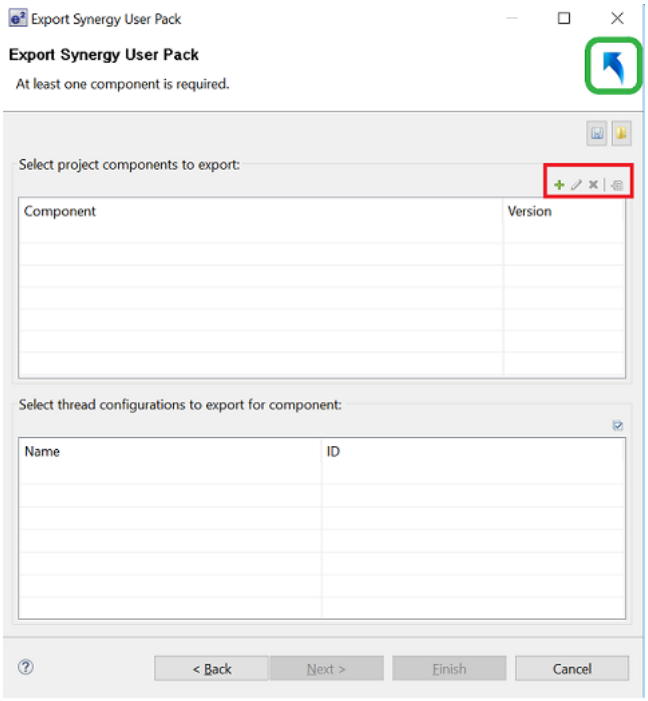


図 10　パック情報の設定

パック名や他の情報を入力した後、 図１３で示されるように、パックに含めるコンポーネント、スレッド、メッセージングコンポーネントを選択することができます。コンポーネント選択ボックス内にある緑色のプラスアイコン (次の図では赤枠の中に表示) を使用して、パックファイルに含める新しいコンポーネントを作成することができます。このアイコンをクリックすると、図１１に示すダイアログボックスが表示されます。



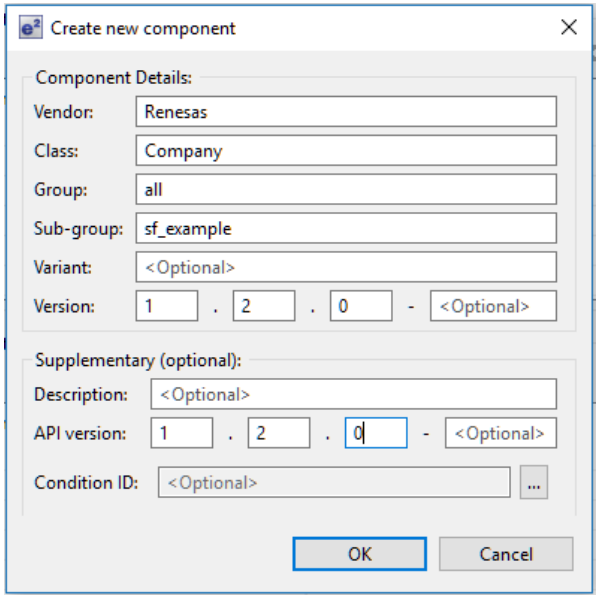


図 11　新しいコンポーネントの作成

3.3 章で説明したように、コンポーネント名を正確につけることが重要です。コンポーネント名は、XML ファイルの名前に一致させる必要があります。命名規則（naming conversion）を次図に再度示します。これらのフィールドは、コンポーネント名（component name）に対して直接マッピングされます。これらの名前が正確に一致していない場合、**[Threads]** タブ内のスタックメニュー（stack menus）でコンポーネントは表示されません。

```
<component Cclass="Company"
           Cgroup="all"
           Csub="sf_example"
           Cvendor="Renesas"
           Cversion="1.2.0"
           condition="">
```

図 12 XML ファイルの命名規則

パック（pack）を命名するときに従うべきいくつかのルールを以下に示します。

1. XML ファイル名の <version> 部分は、ピリオドで区切られた複数の数値を記述する必要があります。3 番目のフィールドにテキスト（text）が記述されている場合は、4 番目のバージョンフィールド（version field）が使用可能です。有効例は、以下の通りです。
   A. 1.0.0
   B. 1.1.0
   C. 1.1.0-beta.1
2. SSP の一連のバージョンに対してパックのテストが実施される必要があります。これは、モジュールを SSP と組み合わせて使用する場合、モジュールに、SSP に一致するバージョンを割り当てる必要があることを意味します。開発したモジュールが複数のバージョンの SSP と組み合わせて使用する場合は、バージョンが異なる複数のパックを個別に作成する必要があります。各モジュールは、該当するバージョンの SSP との互換性を確保することになります。Renesas.Synergy.1.1.0.pack (Renesas が供給している SSP パックの 1 つ) を使用した場合の例は、以下の通りです。
   A. Renesas.SynergyExample.1.1.0.pack が表示されます。
   B. Renesas.SynergyExample.1.0.0.pack は表示されません。
   C. Renesas.SynergyExample.1.2.0.pack は表示されません。
3. e² studio は、2 番目のフィールドより後にあるすべての Version フィールドを無視します。例えば、Renesas.Synergy.1.1.0.pack を使用している場合、以下に示すすべてのパックがサポート対象となります。
   A. Renesas.SynergyExample.1.1.0.pack
   B. Renesas.SynergyExample.1.1.1.pack
   C. Renesas.SynergyExample.1.1.2.pack
   D. Renesas.SynergyExample.1.1.0-alpha.1.pack

コンポーネントを作成した後、開発者はパッケージ化しようとするスレッド（thread）とファイル（file）を選択する必要があります。この作業を実施するには、最初に新しいコンポーネント（component）にチェックマークを付け、その後、コンポーネント名をクリックします。この作業が終わった時点で、次の図に示すようなダイアログボックスが表示されます。

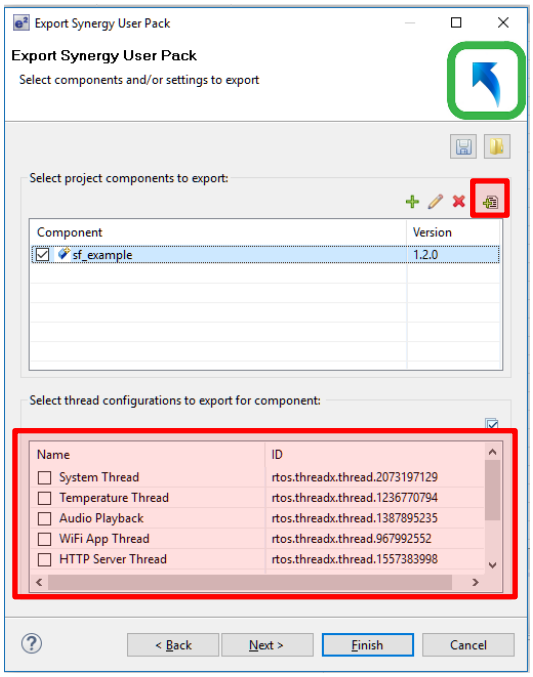


図 13　エクスポートするコンポーネントの追加

ウィンドウ下部の赤枠で強調表示したスレッド（thread）にチェックマークを付けることで、パックに含めるスレッドを選択することができます。右上に赤枠で強調表示した[add file] ボタンを使用して、ファイルを追加することもできます。次の図に、作成した新しいファイルをコンポーネントに追加する方法例を示します。左側に、エクスポート（export）できる複数のファイルが表示されます。追加する各ファイルにチェックマークを付けた後、画面の中央にある **[Add]** ボタンをクリックして、それらのファイルはコンポーネントファイルに追加します。最後に、下部で、ファイルに対応するインクルードパス（include path）を追加することができます。コンポーネント（component）を選択したときに、該当のファイル（file）がプロジェクト（project）に対して自動的に追加されるようになります。

以下の項目はエクスポート（export）できないことに注意してください。

1. SSP コンポーネントと SSP ソースファイル (例　`<Project>\synergy\ssp`)
2. SSP が生成したファイル (例　`<Project>\synergy_cfg\ssp_cfg`)

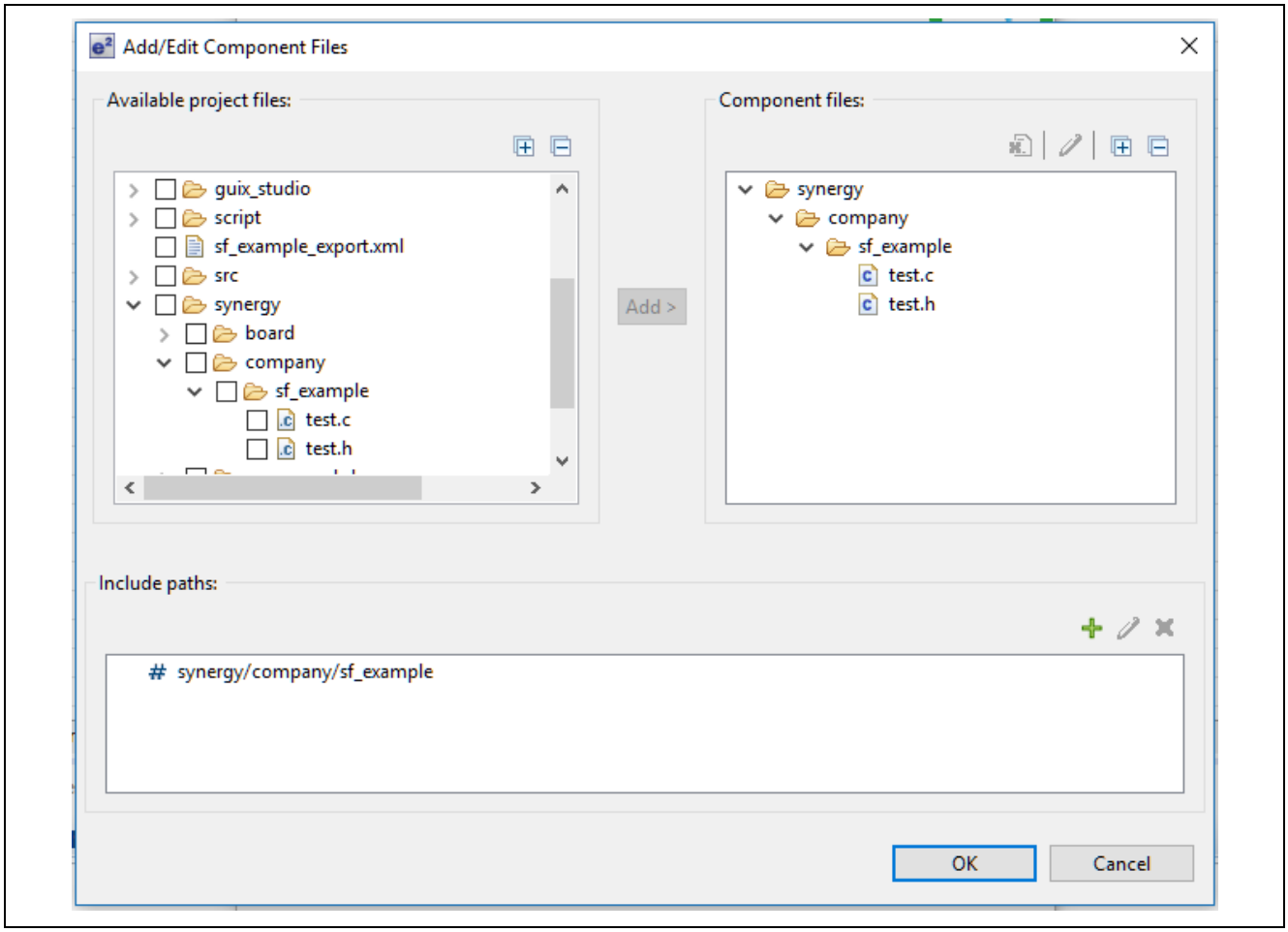


図 14　ファイルの選択とインクルードパスの設定

ファイルを追加した後、**[Finish]** をクリックしてパックを生成することができます。代わりに、**[Next]** をクリックし、エクスポートの対象となるパックの条件を選択することもできます。これらの条件を使用して、特定のコンパイラやターゲットプロセッサなどを指定します。設定を将来再使用するなら、[Finish] をクリックしてパックを生成する前に、設定を確実に保存してください。

4.3　上記のカスタムパックに変更を加えて XML ファイルを含める（Modifying the custom pack from above to include the XML file）

e^2 studio v5.4.0.023 の時点で、モジュールコンフィギュレータ XML ファイル（Module Configurator XML files）を `.module_descriptions` フォルダからパックファイル（pack file）に直接追加することはできません。これらのファイルは、マニュアル（手作業）でパックに追加する必要があります。

1. 次の図に示すように、パックファイル（pack file）をフォルダ（folder）に解凍します。そして、.pack ファイルを削除します。

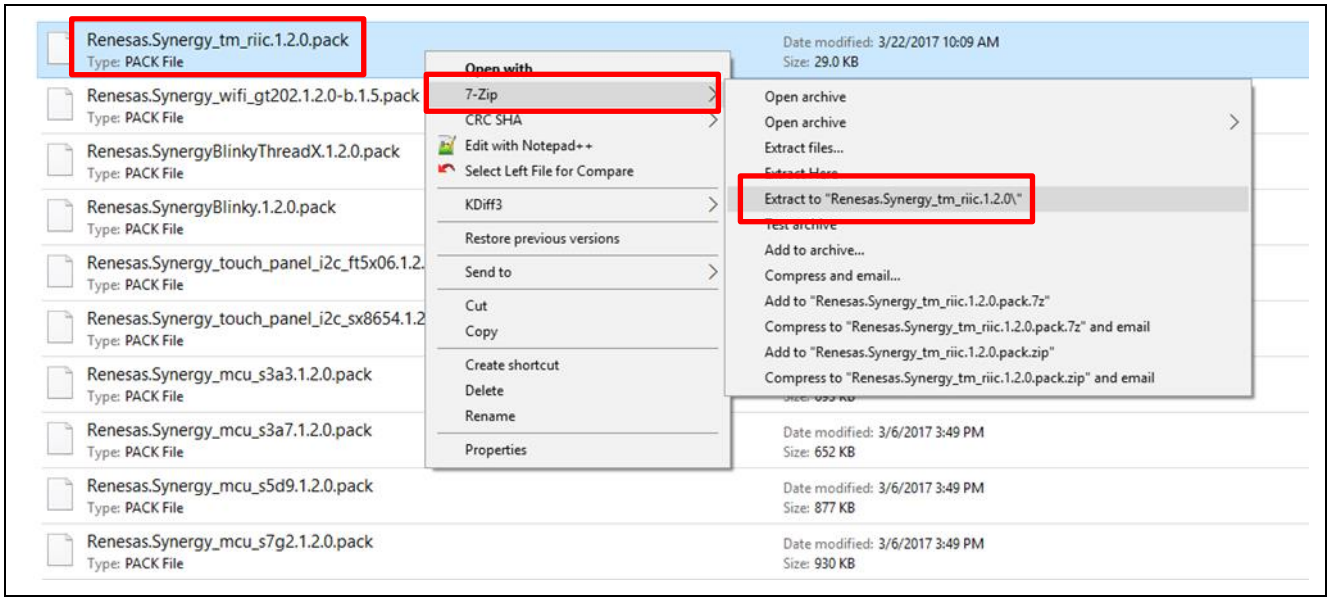


図 15　パックファイルのコンテンツの解凍

2. モジュールコンフィギュレータ XML ファイル（Module Configurator XML files）を .module_descriptions フォルダにコピーします。そのフォルダの中に .module_descriptions というフォルダが存在していない場合、この名前でフォルダを作成します。パックに含めようとするコンテンツは、次の図のように表示されます。

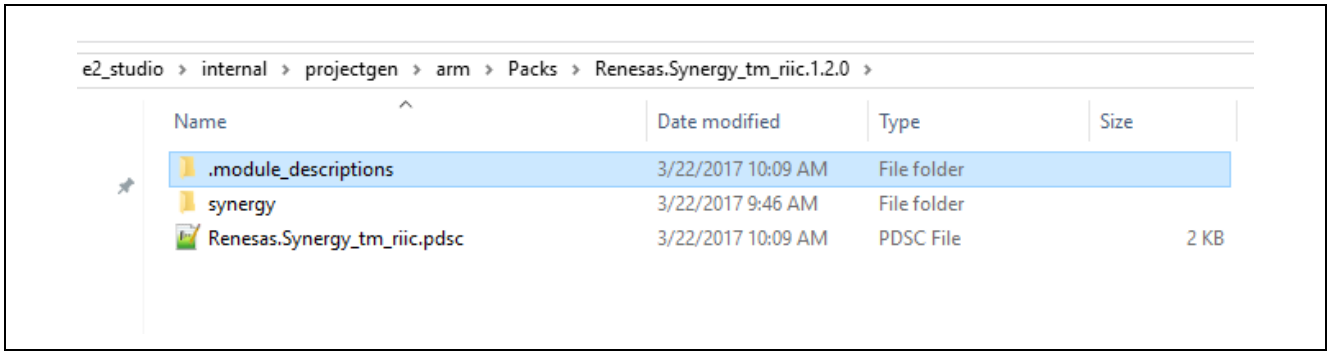


図 16　パックに含めるフォルダのコンテンツ

3. 次の図に示すように、フォルダ（folder）のコンテンツ（contents）を圧縮して .zip ファイルを作成します。この .zip ファイルを `e2_studio/internal/projectgen/arm/Packs` フォルダにコピーします。圧縮されていない元のフォルダを削除します。

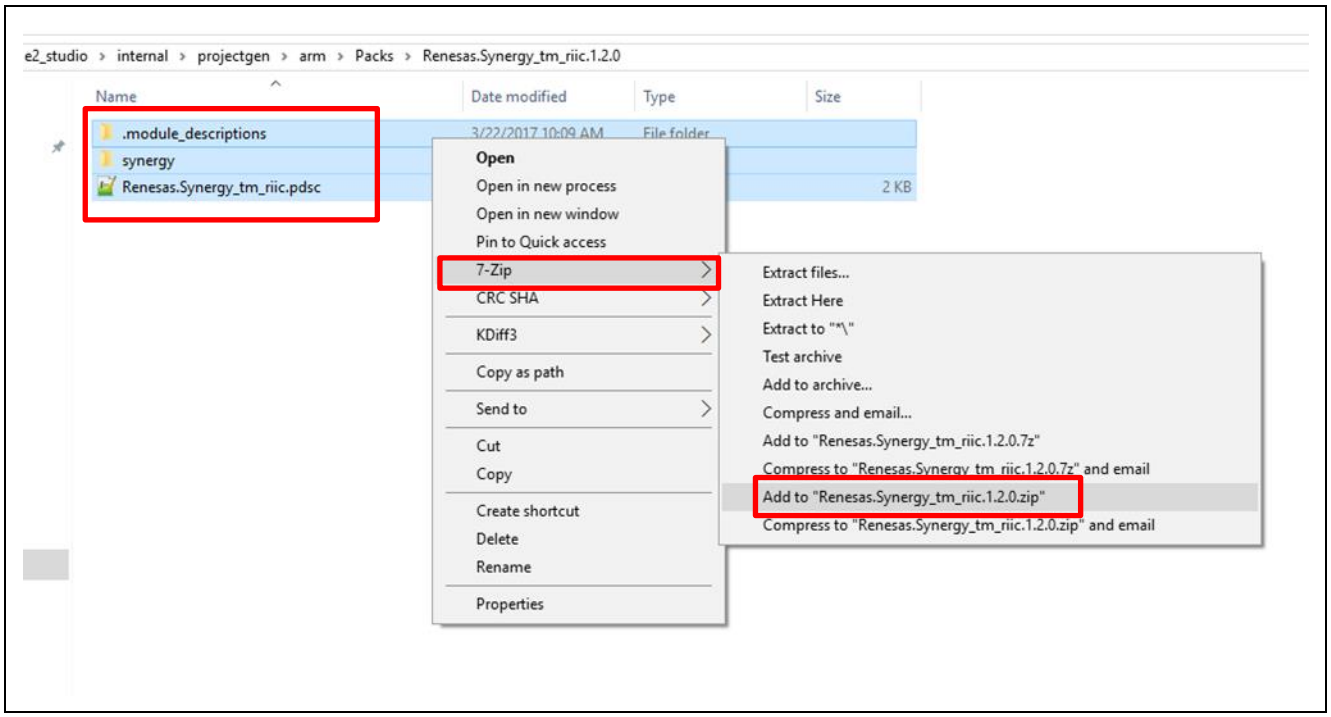


図 17　フォルダのコンテンツを圧縮して .zip ファイルを作成

4. .zip ファイルの名前を変更し、.pack 拡張子を持つファイルにします。変更後のファイルは、**Renesas.Synergy_tm_riic.1.2.0.pack** のようになります。最終的なフォルダは、次の図のように表示されます。

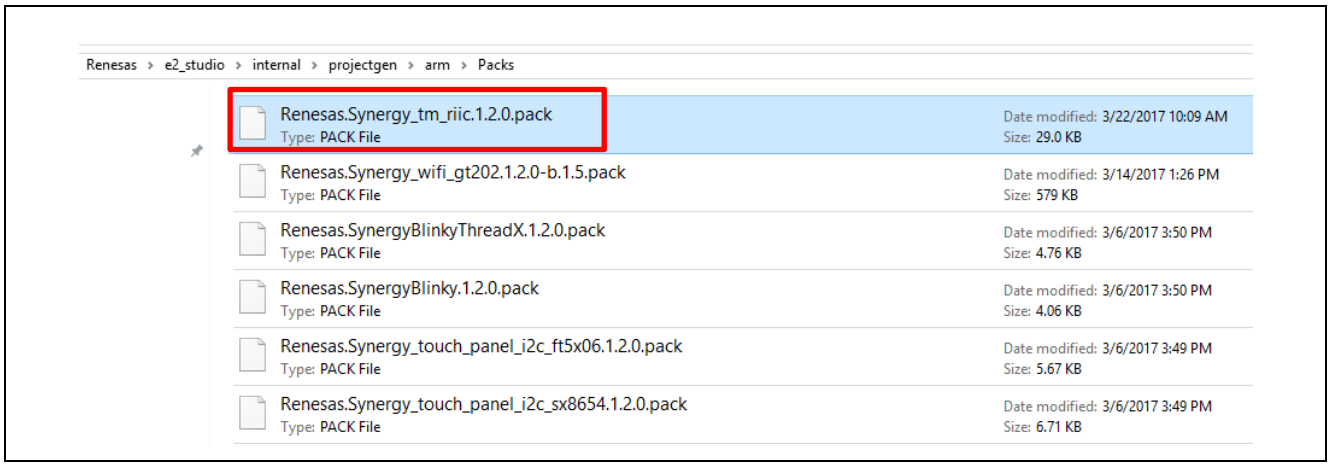


図 18　Packs フォルダ

4.4 IAR Embedded Workbench 向けのパック作成（Pack Creation for IAR Embedded Workbench ）

IAR Embedded Workbench では カスタムパック作成ツール（Custom Pack Creator Tool ）)が利用できないため、いずれのパックファイルも以下のようにマニュアル（手作業）で作成する必要があります。

1. コンテンツフォルダを作成します。
2. PDSC ファイルをマニュアル（手作業）で作成します。
3. モジュールコンフィギュレータ XML ファイル（Module Configurator XML files）を作成し、`.module_descriptions` フォルダに配置します。
4. 上記のコンテンツを .zip ファイルに追加し、.pack ファイルを作成します。

このケースでの追加ステップは、PDSC ファイルのマニュアル（手作業）による作成のみです。

また、パックファイル、XML ファイル、PDSC ファイルの命名規則に注意する必要もあります。

5. カスタム Synergy モジュールの使用（Using the Custom Synergy Module）

5.1 カスタムパックのインストール（Install the Custom Pack）

新しいパックを `<e2_studio_install_folder>/internal/projectgen/arm/Packs` フォルダ（directry）にコピーします。[Synergy Configuration] ウィンドウが既に開いている場合、e² studio は次の図のように、“表示を更新するかどうか？”を問い合わせるウィンドウをポップアップ表示します。Synergy Configuration が開いていない場合、このウィンドウを次に開くときにパックリスト（pack list）が更新されます。

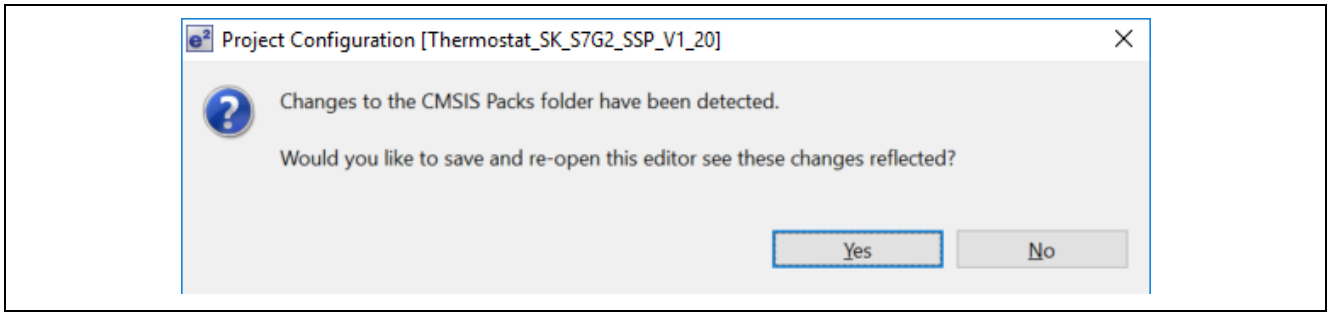


図 19 パック更新ダイアログ

作成したパックが e² studio で正しく認識されるように、いくつかの非常に重要なルールに従う必要があります。

1. 各パックは、特定の e² studio バージョン（version）に固有のものです。e² studio のバージョンが複数存在している場合、バージョンごとにパックをインストールすると、パックが e² studio の各バージョンで認識され、表示されるようになります。
2. SSP の各バージョンは、e² studio の対応バージョンごとにテストが実施済みであり、対応バージョンに関連付ける形でリリースされています。対応バージョンは、SSP のリリースノートに記載されています。独自のモジュールと XML を開発する場合、そのモジュール（module）が対応している SSP バージョンで推奨されているバージョンの e² studio を使用する必要があります。

PDSC のコンポーネントセクション（component section）は、Synergy Configurator ツール内に新しいソフトウェアコンポーネント（software component ）を表示するのに使われています。モジュール（module）に対して、特定の XML コンフィギュレータ（XML configurator）が関連付けられている場合、次の図に示すように、モジュールオプション（module option）の中に新しいモジュールが表示されます。

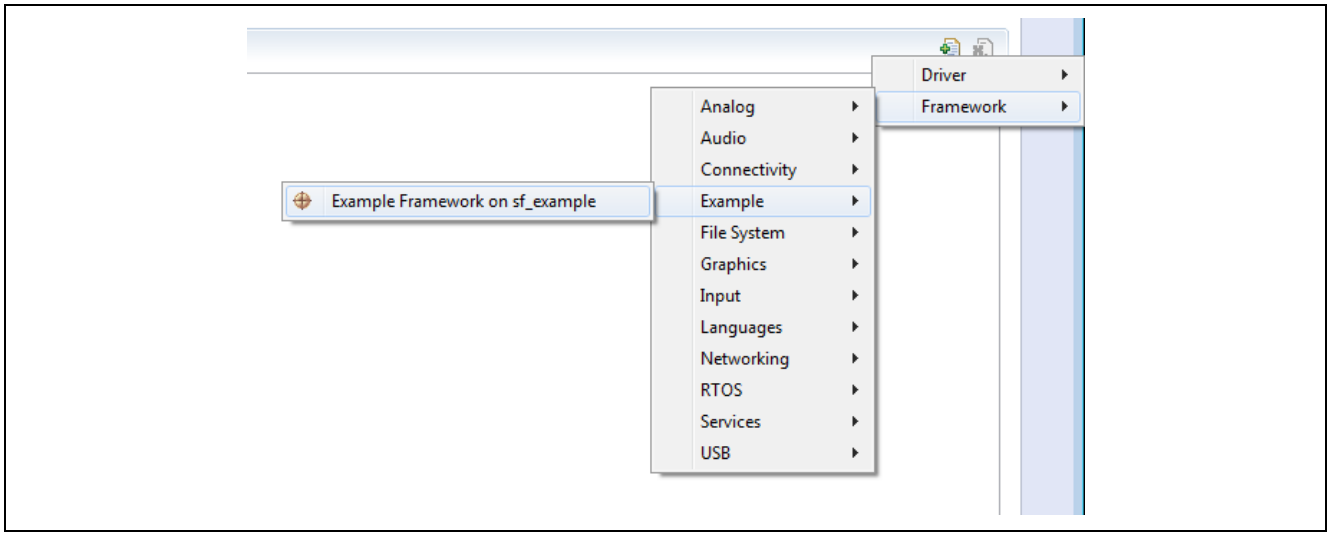


図 20　sf_example にコンフィギュレータが関連付けられ、[Configurator] メニュー内に表示される例

[Threads] タブでいずれかのモジュールを選択すると、e² studio は **[Components]** タブ内でそのモジュールに関連付けられているコンポーネントに対して自動的にチェックマークを付けます。モジュールが XML コンフィギュレータ（XML configurator ）を有していない場合は、**[Components]** タブでコンポーネントに対してマニュアル（手作業）でチェックマークを付ける必要があります。

いずれかのコンポーネントを選択した場合、`<component>` 内で記述されているファイルのみが解凍されます。ここに、サンプルの PDSC ファイルの `<components>` 要素を示します。

注記：カスタムパック（custom pack）のインストールプロセス（installation process）は、既に説明した IAR Embedded Workbench の場合に似ていますが、パックファイルを `<SSC_install_folder>/internal/projectgen/arm/Packs` に配置する必要がある点が異なります。

6. 新しいモジュールの作成例 - 外部 Wi-Fi モジュールに対応する Wi-Fi ドライバを Synergy Wi-Fi フレームワークに追加（Example of Creating a New Module - Adding Wi-Fi Drivers for an external Wi-Fi Module to Synergy Wi-Fi Framework）

デフォルトで、Synergy プラットフォームはさまざまなドライバ（driver）とフレームワーク（framework）をサポートしています。現在、多くの種類のモジュールが利用可能であり、新しいモジュールも継続的にリリースされています。一方で、選択した Wi-Fi モジュールに対応するドライバが、Wi-Fi フレームワークでまだサポートされていないこともあります。このような場合、独自のドライバを作成し Wi-Fi フレームワークに追加することができます。この章では、新しい Wi-Fi モジュールに対応するドライバを Synergy Wi-Fi フレームワークに追加する方法を説明します。

一般的に、出発点として最も簡単な方法は、GT202 のような既存のドライバをコピーし、そのドライバをベースラインとして新しいドライバを作成することです。ドライバを完全に新規で作成しなおすと、新しい Wi-Fi モジュールの作成に要する時間が長くなり、複雑度も増します。

この章では、新しい Wi-Fi ドライバを追加するコピーアンドペーストの方法について、また上位レベルのアプリケーションで使用する Wi-Fi デバイスドライバを取り巻く標準的な構造体や API について説明します。

この章では、追加する新しい Wi-Fi モジュールを **myWiFiDriver** と呼ぶことにします。説明に従って作業を進める場合は、**myWifiDriver** という表記をすべて BCM43362、RS9113、GT202 などの、実際のモジュール名に置き換えてください。

6.1 Wi-Fi フレームワークのソースファイルとヘッダファイル（Wi-Fi Framework source and header files）

最初のステップは、新しいドライバに対応する、Wi-Fi フレームワーク（Wi-Fi Framework）のソースファイル（source）とヘッダファイル（header file）を作成することです。Wi-Fi フレームワークのソースファイルとヘッダファイルは、次のフォルダ（directory）の下に配置します。

```
/synergy/ssp_supplemental/add-on/
```

gt202 のような既存ドライバをコピー&ペーストした後に、選択したモジュールに基づいてコピーしたモジュール名を変更することで、このフォルダ構造（directory structure）を作成できます。次の図で、緑色で強調しているファイルとフォルダが、コピー&ペースト、および名前変更の対象です。赤で強調しているのは、開発者が作成する新しいフォルダ（directory）とファイル構造（file structure）です。

ファイル構造（file structure）を作成した後、エクスプローラ（Explore）を使用して開発中のプロジェクトフォルダ（project directory）に移動し、`.module_descriptions` フォルダ（folder）を参照してください。このフォルダ（folder）でフィルタ機能を使用して、次のコンフィギュレータ XML ファイルを参照します。

```
Renesas##Framework Services##wifi-addon 1.0.0-b.2##sf_wifi_gt202####1.2.0-
b.1.3.xml
```

Wi-Fi フレームワークのリリース番号やバージョン番号に応じて、SSP が異なる可能性があります。コンフィギュレータ（configurator）をコピーし、新しいモジュールに合わせてコピー先の名前を変更します。次に、**myWiFiModule** での例を示します。

```
Renesas##Framework Services##wifi-addon 1.0.0-
b.2##sf_wifi_myWifiModule####1.2.0-b.1.3.xml.
```

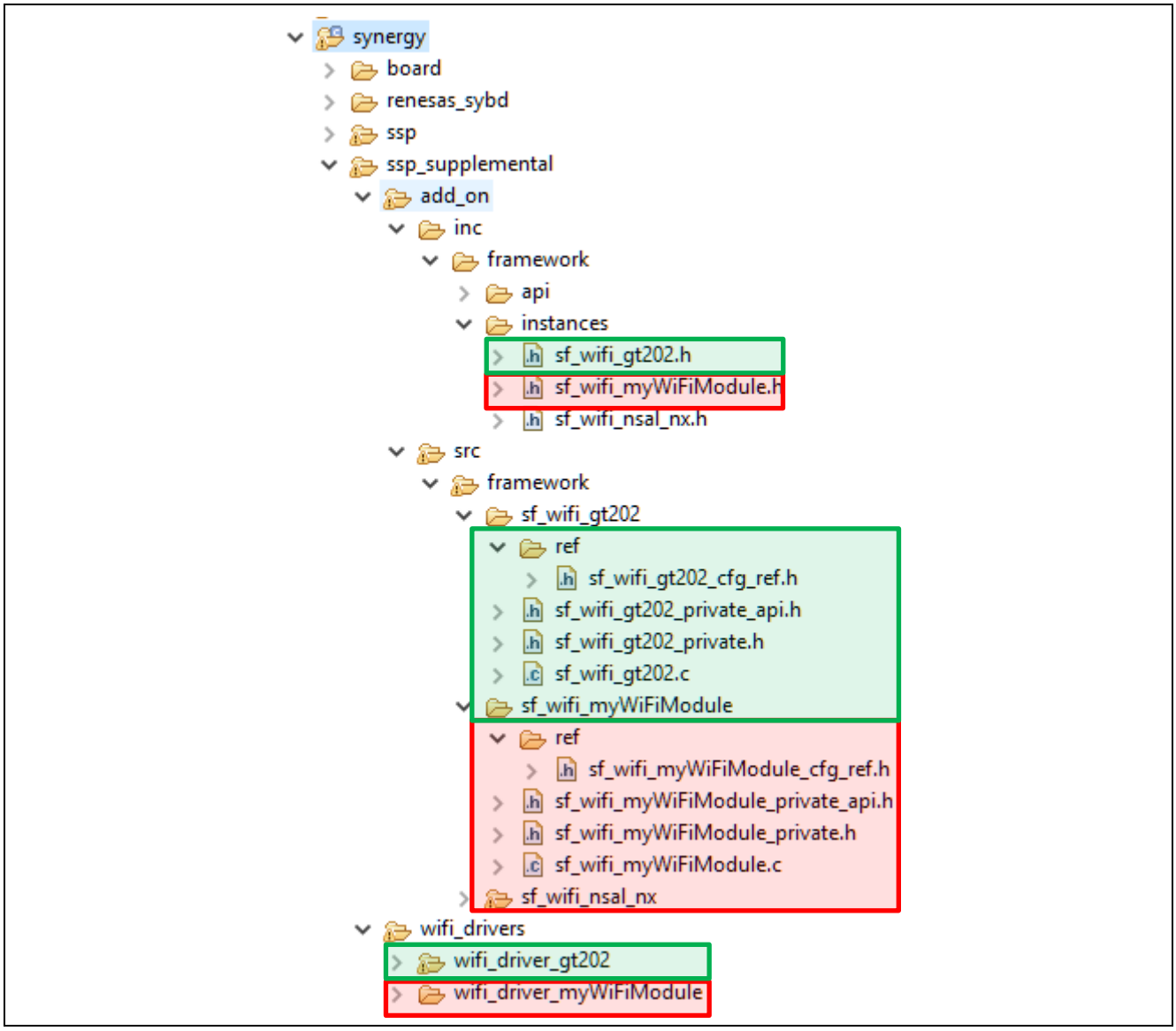


図 21　Wi-Fi モジュールのフォルダとファイル構造の設定

次の章では、ここで作成したファイルの考察を行い、新しい **myWiFiModule** と組み合わせたときに動作するように、それらのファイルに加える必要のある変更について説明します。

6.1.1　インスタンスヘッダファイルの更新（Updating the instance header name）

インスタンスヘッダファイル（instance header file） `sf_wifi_myWiFiModule.h` は、拡張された設定に対応する構造体（structure）を定義しています。拡張された設定の中には、 `r_spi`, `r_sdmmc` など下位レベル（lower level）の通信インタフェースインスタンスへのポインタ（pointer）や、Reset ピンとスレーブ選択ピンなど Wi-Fi モジュールが使用する IO PORT ピン、またドライバタスクスレッド（driver task thread）の優先度 (Wi-Fi モジュールのデバイスドライバのコードが内部で作成するスレッドの優先度) が記述されています。この構造体の中に、Wi-Fi モジュールに固有の設定可能な付加的フィールドを含めることもできます。

新しいインスタンスヘッダファイルを新しいドライバに対応させるために、以下の手順を実施してファイルを更新する必要があります。

1. GT202 に関するすべての参照を **MYWIFIMODULE** に変更します。
2. gt202 に関するすべての参照を **myWiFiModule** に変更します。

3. インクルードするファイルをレビューします。
   ほぼすべての Wi-Fi モジュールに、Wi-Fi フレームワークによる監視または制御を必要とする割り込みピン（interrupt pin）と IO があります。これらのインクルードファイル（included file）は、通常そのまま使用できます。通信（communication）が持つファイルはレビューする必要があります。SPI 通信（SPI communication）を使用する Wi-Fi モジュールは、`r_spi_api.h` を含む必要があります。IIC を使用する場合は、`r_iic_api.h` が必要です。正しい通信ドライバ API（communication API） が含まれていることを確認してください。
4. 以下の形式で定義されているインスタンス（instance）のバージョン番号を更新します。
   `SF_WIFI_MYWIFIMODULE_CODE_VERSION_MAJOR`
   `SF_WIFI_MYWIFIMODULE_CODE_VERSION_MINOR`

6.1.2 参照設定ファイルの更新（Updating the reference configuration file）

参照設定ファイル（reference configuration file）は、設定可能な共通の定義に対応するマクロ（macro）を保持しています。その中には、`SF_WIFI_MYWIFIMODULE_CFG_PARAM_CHECKING_ENABLE` のような定義（definition）と、それぞれのデフォルト値（default value）が含まれています。

通常、必要とされているのは、`sf_wifi_myWiFiModule_cfg_ref.h` 内で宣言されているデフォルトのマクロのみです。開発者は単に、GT202 に対するすべての参照（reference）を **myWiFiModule** に変更するだけで十分です。更新後の参照ファイルは、次の図のようになります。

```
#ifndef SF_WIFI_MYWIFIMODULE_CFG_H
#define SF_WIFI_MYWIFIMODULE_CFG_H

/** Specify whether to include code for API parameter checking

#define SF_WIFI_MYWIFIMODULE_CFG_PARAM_CHECKING_ENABLE    (BSP_CFG_PARAM_CHECKING_ENABLE)

#define SF_WIFI_MYWIFIMODULE_CFG_ONCHIP_STACK_SUPPORT     (0)

#endif /* SF_WIFI_MYWIFIMODULE_CFG_H */
```

図 22 更新後の参照設定ファイル

6.1.3 フレームワーク API の更新（Updating the Framework APIs）

`sf_wifi_myWiFiModule_private_api.h` ファイルには、**myWiFiModule** がサポートしているすべての API のプロトタイプを記述しています。**myWifiModule** に対応する API を更新するには、検索/置換操作（find/replace operation）を再度実行し、GT202 というテキストすべてを **myWiFiModule** に置き換えます。結果は次の図のように表示されます。

```
#ifndef SF_WIFI_myWiFiModule_PRIVATE_API_H
#define SF_WIFI_myWiFiModule_PRIVATE_API_H

 * Private Instance API Functions. DO NOT USE! Use functions through Interface API structure instead.
ssp_err_t SF_WIFI_myWiFiModule_Open(sf_wifi_ctrl_t * p_ctrl, sf_wifi_cfg_t const * const p_cfg);
ssp_err_t SF_WIFI_myWiFiModule_Close(sf_wifi_ctrl_t * const p_ctrl);
ssp_err_t SF_WIFI_myWiFiModule_MulticastListAdd(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac_addr);
ssp_err_t SF_WIFI_myWiFiModule_MulticastListDelete(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac_addr);
ssp_err_t SF_WIFI_myWiFiModule_StatisticsGet(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_stats_t  * const p_wifi_device_stats);
ssp_err_t SF_WIFI_myWiFiModule_Transmit(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_buf, uint32_t length);
ssp_err_t SF_WIFI_myWiFiModule_ProvisioningSet(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_provisioning_t const * const p_wifi_provisioning);
ssp_err_t SF_WIFI_myWiFiModule_ProvisioningGet(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_provisioning_t * const p_wifi_provisioning);
ssp_err_t SF_WIFI_myWiFiModule_InfoGet(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_info_t * const p_wifi_info);
ssp_err_t SF_WIFI_myWiFiModule_Scan(sf_wifi_ctrl_t * const p_ctrl, sf_wifi_scan_t * const p_scan, uint8_t * const p_cnt);
ssp_err_t SF_WIFI_myWiFiModule_AccessControlListAdd(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);
ssp_err_t SF_WIFI_myWiFiModule_AccessControlListDelete(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);
ssp_err_t SF_WIFI_myWiFiModule_MACAddressGet(sf_wifi_ctrl_t * const p_ctrl, uint8_t * const p_mac);
ssp_err_t SF_WIFI_myWiFiModule_MACAddressSet(sf_wifi_ctrl_t * const p_ctrl, uint8_t const * const p_mac);
ssp_err_t SF_WIFI_myWiFiModule_VersionGet(ssp_version_t * const p_version);

#endif /** SF_WIFI_myWiFiModule_PRIVATE_API_H */
```

図 23 更新後の API 呼び出し

6.1.4 プライベート構造体/マクロの定義の更新（Updating the private structure/macro definitions）

sf_wifi_myWiFiModule_private.h ファイルは、myWiFiModule に対して Wi-Fi フレームワークが使用するプライベート構造体（private structure）とマクロ（macro）を記述しています。このファイルでは、デバイスドライバ（device driver）のプライベート構造体/データ型/マクロの定義を記述しないでください。このファイルは、Wi-Fi フレームワークの制御構造体（control structure）の内部で使用される構造体定義（structure definition）を記述しているほか、Wi-Fi デバイスドライバとの通信を取り扱います。

ここでも、myWiFiModule に合わせてこのファイルを更新する作業は単純で、GT202 と gt202 に対するすべての参照を検索して、それぞれ **MYWIFIMODULE** と **myWiFiModule** で置き換えるだけです。

6.1.5 フレームワーク API の実装の更新（Updating the Framework API implementation）

sf_wifi_myWiFiModule.c ファイルは、Synergy Wi-Fi フレームワークの API 実装を有しています。これらの API に関する説明は、Wi-Fi フレームワークのアプリケーションノートとプロジェクト（project）に掲載されています。このファイル内の実装は汎用的ですが、myWiFiModule ドライバと制御構造体（control structure）の呼び出しを行います。このため、ここでも検索/置換機能を使用して、GT202 と gt202 に対するすべての参照を、それぞれ **MYWIFIMODULE** と **myWiFiModule** で置き換える必要があります。

6.1.6 Wi-Fi ドライバのソースコードの更新（Updating the Wi-Fi driver source code）

/wifi_drivers/wifi_driver_myWiFiModule/ フォルダには、Wi-Fi モジュールのデバイスドライバが格納されています。新しいモジュールを動作させるための変更の大部分は、このフォルダで実施する必要があります。モジュールごとに違いはありますが、ドライバの更新はシンプルなプロセスです。プロセスの例を以下に示しますが、開発者のモジュールによってはこの手順が当てはまらないこともあります。

1. Wi-Fi モジュールベンダのドライバコード（driver code）をダウンロードします。SSP 内で、このドライバを Wi-Fi フレームワークに統合する必要があります。
   /wifi_drivers/wifi_driver_myWiFiModule/ フォルダ内で、**gt202_ctrl** フォルダ（folder）の名前を **myWiFiModule_ctrl** に変更した後、各ファイルの名前も変更して新しい名前を **myWiFiModule_filename** にします。wifi_driver_myWiFiModule フォルダは、次の図に示すようになります。

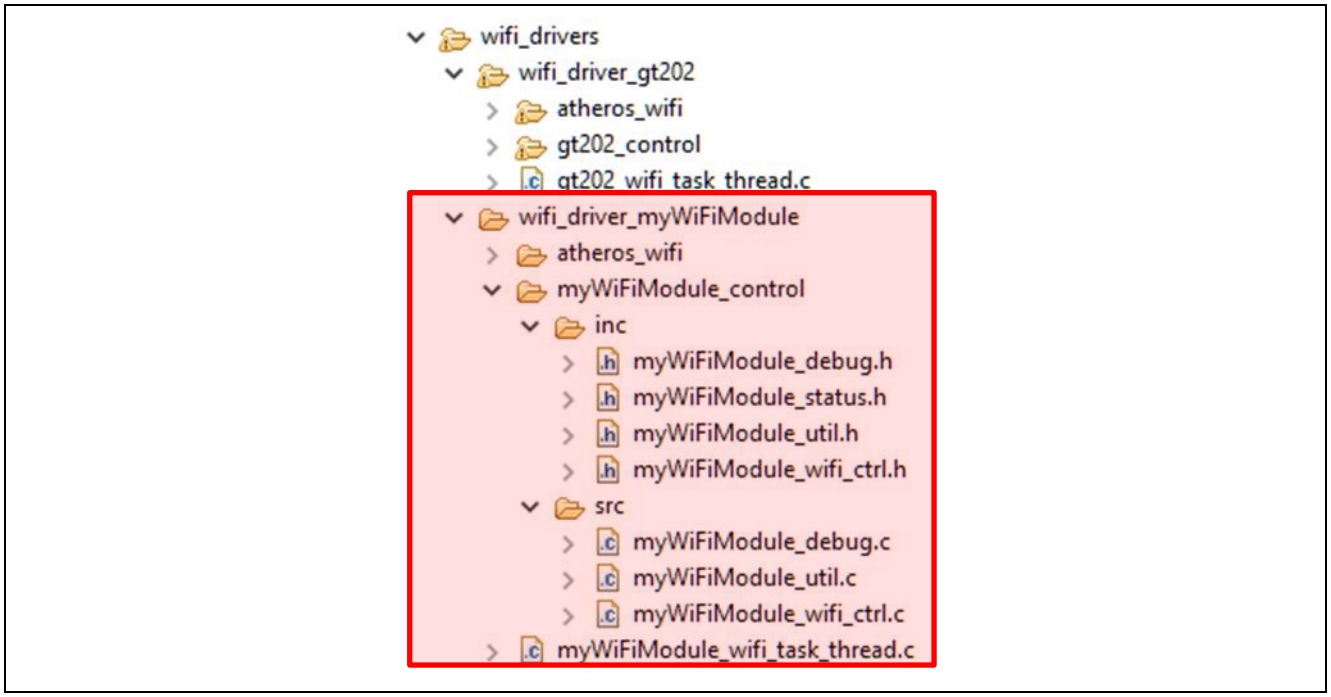


図 24 wifi_driver_myWiFiModule フォルダの構造

2. myWiFiModule_control フォルダ内の各ファイルをレビューし、全ての「GT202」というテキストを **MYWIFIMODULE** に変換します。また全ての「gt202」というテキストも **myWiFiModule** に変換します。
3. myWiFiModule_wifi_ctrl.c ソースファイルをレビューします。実装済みの各関数が、Wi-Fi モジュールの動作に必要とされる正しいコードが記述されていることを確認します。必要に応じて、新しいモジュールと互換性が確保できるようにコードを変更します。
4. プロジェクトのプロパティ（project property）を参照し、コンパイラがファイルを見つけられるように、ヘッダファイルのパスが追加されていることを確認します。

6.1.7 Wi-Fi ドライバの設定ヘッダファイルの更新（Updating the Wi-Fi driver configuration header file）

他のすべてのモジュールと同様に、新しい myWiFiModule でも設定ファイル（configuration file）が必要になります。sf_wifi_gt202_cfg.h ファイルは、sf_wifi_myWiFiModule_cfg.h 作成用テンプレートの役割を果たします。以下の手順に従って、設定ヘッダファイルを作成します。

1. sf_wifi_gt202_cfg.h を、synergy_cfg/ssp_cfg/framework/ フォルダ（directory）からクリップボードにコピーします。そのファイルを貼り付け、次の図に示すように sf_wifi_myWiFiModule_cfg.h という名前に変更します。ここでも、緑色で強調したファイルは元（original）のファイル、赤で強調したファイルは新規作成したコピー先ファイルです。

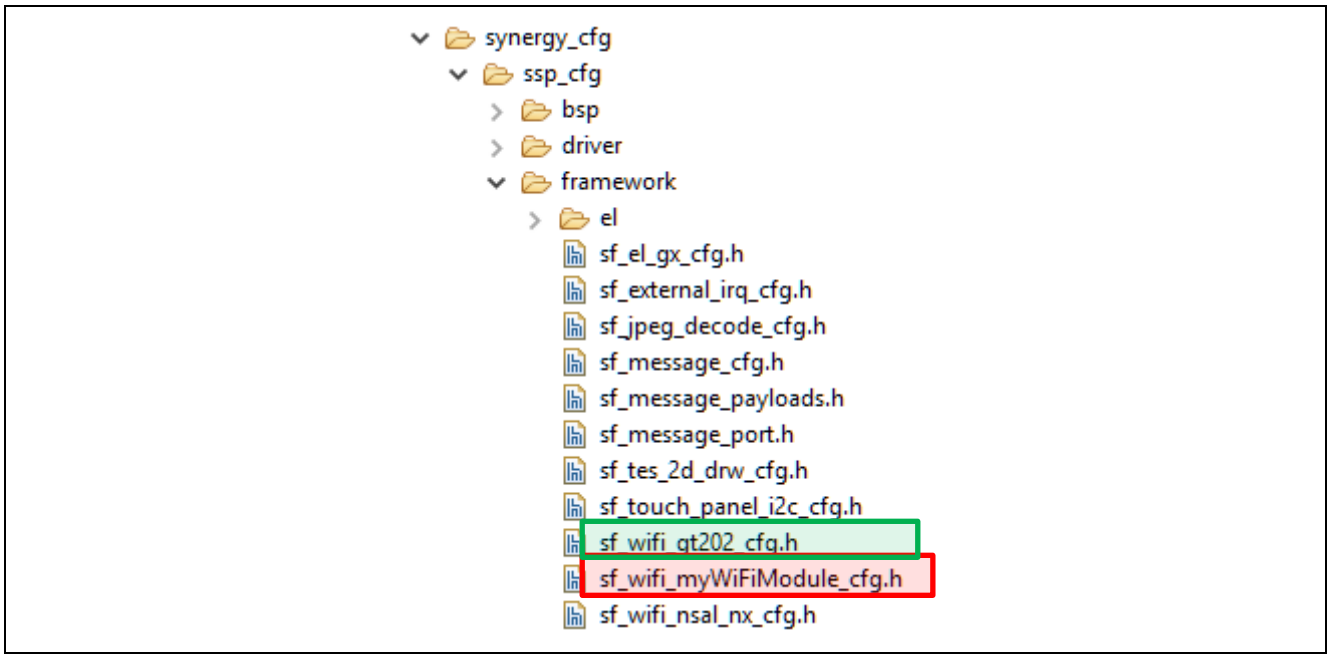


図 25 設定ヘッダの作成

2. sf_wifi_myWiFiModule_cfg.h という新しいファイルを再度開き、検索/置換機能を使用して、GT202 と gt202 に対するすべての参照を、それぞれ **MYWIFIMODULE** と **myWiFiModule** に置き換えます。
3. ドライバをビルドします。ビルドが正常に完了しない場合、問題の解決が必要です。

6.1.8 設定 XML の更新（Updating the configuration XML）

作成済みの設定 xml ファイル（configuration XML file）は、Renesas##Framework Services##wifi-addon##sf_wifi_myWiFiModule####x.xx.xx.xml という名前になっています。このファイルには ISDE コンフィギュレータ（configurator）で設定されるモジュール設定オプション（module configuration option）が含まれます。このファイルを開き、gt202 に関するすべての参照を **myWiFiModule** に変更します。

このファイルはバックアップを作成してください。

6.1.9 配布用のパックの作成（Create the Pack for distribution）

この時点で、カスタムパック作成ツール（Custom Pack Creator ）を使用して、新しい Wi-Fi ドライバ（driver）の作成と配布をすることができます。

7. 付録 - モジュールコンフィギュレータ XML ファイルに関するルール（Appendix – Rules for the Module Configurator XML File）

この章では、モジュールコンフィギュレータ XML ファイル内で利用できるさまざまなタグ（tag）の詳細について説明します。以下の事例とルールを参照することで、XML ファイルをカスタマイズする際に必要な詳細な知識を習得できます。

7.1 ユーザに対して表示されるテキストに関するベストプラクティス（Best practices for user-visible text）

ユーザに対して表示されるすべてのテキストフィールド (「display=」で始まる属性) は、人間が読み取れるものにする必要があります。

略称や頭字語が正しく表記されるるように、すべての display id をマニュアル（手作業）で編集します。例えば、次に示す図で太字になっているテキストは、元は「Khz」という表記でしたが、読みやすくなるように「kHz」に変更しました。

```
<option display="Unit Frequency kHz"
        id="module.driver.timer.unit.unit_frequency_khz"
        value="TIMER_UNIT_FREQUENCY_KHZ"/>
```

7.2 ユーザに対して表示されるテキストのコンテンツ（Content of text visible to users）

```
<option display="Unit Frequency Khz"
        id="module.driver.timer.unit.unit_frequency_khz"
        value="TIMER_UNIT_FREQUENCY_KHZ"/>
```

7.3 要素を変数として使用（Using elements as variables）

注記：要素 id は、display 属性（attribute）や Declarations 要素（element）などのテキストフィールド（text field）で使用される変数に似ています。順序は重要ではありません。テキストフィールドは、XML ファイル中の後方で定義される要素を解決することができます。変数を使用する場合、${<element_id>} という表記は、ユーザが選択した値にリゾルブ（resolve）されます。

ユーザが Name フィールドを編集していない場合、次の例では、${module.driver.timer.name} は g_timer (デフォルト属性) にリゾルブ（resolve）されます。

```
<module config="config.driver.gpt"
        display="${module.driver.timer.name} TIMER Driver on GPT${module.driver.timer.channel}"
        id="module.driver.timer_on_gpt" version="1">
    ...
    <property default="g_timer" display="Name" id="module.driver.timer.name">
```

次の例では、ISDE 内で kHz 周波数単位（Unit Frequency kHz）が選択されていることを想定しています。

1. module.driver.timer.unit の値は、module.driver.timer.unit.unit_frequency_khz という value 属性（attribute）に設定されています。
2. いずれのテキストフィールド（text field）でも、${module.driver.timer.unit} を参照すると、TIMER_UNIT_FREQUENCY_KHZ にリゾルブ（resolve）されます。

```
<option display="Unit Frequency kHz"
        id="module.driver.timer.unit.unit_frequency_khz"
        value="TIMER_UNIT_FREQUENCY_KHZ"/>
```

7.4 Config 要素（Config element）

XML ファイル内の最初の要素は、config 要素です。config 要素は、ビルド時の設定（build time configuration）を記述しています。この設定は、ssp_config/<driver|framework>/<namespace>_<instance>_cfg.h ファイルに反映されます。

ヘッダファイル（header file）<namespace>_<instance>_cfg.h のような config 要素は、モジュールのどのインスタンス（instance）においても共通です。例えば、GPT2 に対応する TIMER ドライバで、あるパラメータのチェックをオンからオフに変更すると、このパラメータのチェックマークは、GPT5 に対応する TIMER ドライバでもオンからオフに自動的に変更されます。パラメータのチェックマークオプションは、チャネルごとに画面で設定できますが、チャネルごとに異なるオプションになるように定義することはできません。

7.5 属性の id、パス、バージョン（Attributes id, path, and version）

config 要素を定義する最初の行で、以下の属性（attribute）を定義する必要があります。

- id= config.<driver|framework>.<instance>
  後で <module> 要素がこの定義を使用して、その <module> 要素を特定の <config> 要素に結び付けます。
- path= ssp_cfg/<driver|framework>/<namespace>_<instance>_cfg.h
  これは、出力ファイル(<namespace>_<instance>_cfg.h) が生成される synergy_cfg フォルダに関連するパス（path）です。
- version= 1.0

以下に、GPT から取得した 例を示します。

```
<config id="config.driver.gpt" path="ssp_cfg/driver/r_gpt_cfg.h" version="0">
```

7.6 Property 要素（Property elements）

各 config 要素は、少なくとも 3 つのオプションで構成された property ドロップダウン（drop down）を記述する必要があります。各モジュールが必要とするパラメータチェックマクロ（parameter check macro ）が、これらのオプションを使用します。

1. マニュアル（手作業）で、BSP の option 要素を編集します (次の例を参照)。
2. マニュアル（手作業）で、デフォルト値を編集して、config.driver.<module>.param_checking_enable.bsp に変更します。
3. マニュアル（手作業）で、display の値を編集して、**Parameter Checking** に変更します。
4. 必要に応じて、ビルド時の他の設定の値をマニュアル（手作業）で編集します。

以下に、GPT プロパティの 例を示します。

```
<property default="config.driver.gpt.param_checking_enable.bsp"
          display="Parameter Checking"
          id="config.driver.gpt.param_checking_enable">

    <option display="Default (BSP)"
            id="config.driver.gpt.param_checking_enable.bsp"
            value="(BSP_CFG_PARAM_CHECKING_ENABLED)"/>

    <option display="Enabled"
            id="config.driver.gpt.param_checking_enable.enabled"
            value="(1)"/>

    <option display="Disabled"
            id="config.driver.gpt.param_checking_enable.disabled"
            value="(0)"/>

</property>
```

7.7 Module 要素（Module element）

モジュール内にある各要素は、モジュールの単一のインスタンス（instance）に対して適用されます。例えば、ひとつのアプリケーションが複数の GPT チャネル（channel）を使用する可能性があります。GPT チャネル 2 と GPT チャネル 5 を使用している場合、各チャネルはそれぞれ独自の Module 要素を保存することになります (この点は、共有される config 要素とは異なります)。

7.8 "common" 属性とその考え方（Attributes and the idea of “common”）

module 要素を定義する最初の行で、以下の属性（attribute）を定義する必要があります。

```
config=” config.<driver|framework>.<instance>”
Must match the id of the config element created above.
display="${module.<driver|framework>.<api>.name} <API> Driver on <module_name>
id=” module.<driver|framework>.<api>_on_<instance>”
version=” 1”
```

(Optional, see below) (オプションであり、詳細は以下を参照) `common= n.`

ソフトウェアスタック（software stack）内でモジュール（module）を使用する方法は 3 種類あります。適切なオプションは、`<module>` 要素内で common 属性（attribute）を使用しているかどうか、およびその値（value）に依存します。

1. Common 属性（attribute）が見つからない場合。
   — これはデフォルトのケースです。このモジュールインスタンス（ Module Instance ISDE 内のひとつのブロック) が、現在の ソフトウェアスタック（Software Stack）内でのみ使用できることを意味します。異なるソフトウェアスタックを持つ 2 つのスレッド（thread）を使用している場合、この設定を定義している <module> はどちらか一方のスレッドで使用できますが、もう一方のスレッドでは使用できません。もう一方のスタックでは、新しいモジュールインスタンスを作成する必要があります。同じスレッド内に存在する複数のスタックについても同様です。
   — ほとんどの HAL ドライバは現在、このアプローチ（approach）を使用しています。ドライバはスレッドセーフ（thread safe）ではなく、共有することは意図していません。アプリケーションレベルでドライバを共有することは可能です。
2. Common 属性（attribute）が **1** に設定されている場合。
   — **1** という値は、このモジュールに対して存在できる モジュールインスタンス（ Module Instance）がただ 1 つであることを意味します。この <module> に対応する モジュール インスタンスが既に存在している状況で、同じ <module> に対応する新しいモジュール インスタンスを作成しようとしても、利用できるのは、既存の Module インスタンスを使用するオプションのみです。
   — このアプローチを使用するモジュール例は、X-Ware ライブラリです。そして、存在可能な NetX ライブラリは 1 個のみです。IP とパケットプール（packet pool）を複数回使用することは可能ですが、共通ライブラリ（common library）のインスタンスは 1 つのみ作成可能です。
3. Common 属性（attribute）が **n** (任意の整数) に設定されている場合。
   — **1** より大きい整数値を指定すると、この <module> が **n** 個の モジュールインスタンスを持てることを意味します。例えば、この属性の値が **4** の場合、モジュールの個別インスタンスを 4 個作成できます。その後、既存の Module インスタンスのいずれかを選択する必要があります。通常、**100** という値を使用します。この値は、**無制限**を意味します。一般的な使用状況で、モジュールに対して **100** 個のインスタンスを作成する自体はほぼ発生しないからです。
   — このアプローチ例として、SPI と I²C フレームワークのバス（bus）、および NetX パケットプールを使用する状況が考えられます。1 個の NetX IP インスタンスと 1 つの NetX HTTP Client の間で、共通する 1 個のパケットプールを共有したいと考えることがあります。

次に、共通属性（common attribute）を設定していない GPT の例を示します。

```
<module config="config.driver.gpt"
        display="${module.driver.timer.name} Timer Driver on r_gpt"
        id="module.driver.timer_on_gpt"
        version="1">
```

次に、1 個のインスタンスのみを許可する ThreadX® ソース（source）の例を示します。

```
<module config="config.el.tx_src"
        id="module.framework.tx_src"
        display="Framework|RTOS|ThreadX Source"
        common="1"
        version="0">
```

次に、I²C フレームワーク共有バス（I²C Framework Shared Bus）の例を示します。ここでは、モジュールの共有インスタンス数として、無制限 **(100)** を許可しています。

```
<module config="config.framework.sf_i2c_bus"
        display="Framework|Connectivity|${module.framework.sf_i2c_bus.name} I2C Framework
                 Shared Bus on sf_i2c"
        id="module.framework.sf_i2c_bus_on_sf_i2c"
        common="100"
        version="1">
```

7.9 Constraint 要素（Constraint element）

Constraint は、開発プロセスの早い段階で開発者にエラーを警告する目的で使用します。Constraint は、ビルド段階やその後のデバッグ段階ではなく、ISDE を使用している開発段階でエラーを検出します。

Constraint は、JavaScript の条件文（conditional statement）に挿入しなくてはいけない制約があります。

```
if (!<constraint>) { /* Error */ }
```

注記：constraint 中でプロパティ（property）を参照する場合、現在のモジュールから取得したモジュールプロパティのみを使用できます。下位レベル（lower level）のモジュールに対して制約を適用する方法については、Override 要素の説明を参照してください。

複数のインスタンスを持つことのできるどのモジュールでも、次の constraint の記述が必要です。

display=”Module instances must have unique names” (Module のインスタンスは固有の名前を持たなければならない)

次に、audio フレームワーク（framework）に対応する例を示します。

```
<constraint display="Module instances must have unique names">
"${interface.framework.sf_audio_playback.${module.framework.sf_audio_playback.name}}" === "1"
```

7.10 Provides interface 要素（Provide interface element）

provides 要素によって、上位レイヤー（upper layer）のモジュール間を結び付けることができます。指定したインタフェースは他のモジュールからアクセスすることできます。また、そのインタフェースが提供される回数を指定することもできます。

以下の要素（element）を追加します。

- <provides interface="interface.driver|framework.<api>”/>
- <provides interface="interface.driver|framework.<api>_on_<instance>”/>

作成するドライバが複数のインスタンスをサポートするる場合は、以下の要素（elements）を追加します。

- <provides interface="interface.driver|framework.<api>.${module.driver|framework.<api>.name}” />
- <provides interface="interface.driver|framework.<api>_on_<instance>.${module.driver|framework.<api>.name}” />

次に、GPT に対応する例を示します。

```
<provides interface="interface.driver.timer" />

<provides interface="interface.driver.timer_on_gpt" />

<provides interface="interface.driver.timer.${module.driver.timer.name}" />

<provides interface="interface.driver.timer_on_gpt.${module.driver.timer.name}" />
```

7.11 Requires interface 要素

(BSP 以外の) 下位レベル（lower level）のモジュールが必要な場合、次の行を追加します (大部分の HAL モジュールで、次の行は必須ではありません)。

注記：<requires> 要素内にある interface 属性（attribute）は、下位レベルのモジュール内にある <provides> 要素の interface 要素と一致している必要があります。この方法で、両者を結び付けることができます。

```
<requires id="module.driver|framework.<api>.requires.<lowerlevelapi>"
interface="interface.driver|framework.<lowerlevelapi>" display="Add <Lower Level
Module Name>" />

If your (framework) module requires ThreadX, use:
     <requires interface="_rtos" />
```

- このオプションは、モジュールコンフィギュレータ（module configuration）の **[Threads]** タブでこのモジュールが利用できるようになることを意味します。このオプションを記述しない場合、モジュールコンフィギュレータの **[HAL]** タブでこのモジュールが利用できるようになります。

次に、sf_audio_playback フレームワーク（framework）に対応する例を示します。

```
<requires interface="_rtos" />

<requires id="module.framework.sf_audio_playback_common.requires.sf_message"
          interface="interface.framework.sf_message"
          display="Add Messaging Framework" />

<requires id="module.framework.sf_audio_playback_common.requires.sf_audio_playback_hw"
          interface="interface.framework.sf_audio_playback_hw"
          display="Add Audio Playback Hardware" />
```

7.12 Override 要素（Override element）

開発したモジュールが、<requires> 要素で指定した下位レベルモジュール（lower level module）を必要とし、なおかつ開発したモジュールが特定の設定値を必要とする場合、<requires> 要素の <override> サブ要素が、下位レベルに対してこの設定値を強制的に適用します。

```
<override property="module.driver|framework.<lowerlevelapi>.<lowerlevelid>"
value="module.driver|framework.<lowerlevelapi>.<lowerlevelid>.<lowerlevelvalue
> "/>
```

以下に、 sf_audio_playback_hw_dac フレームワーク（framework）に対応する例を示します。この例では、DAC を対象とする上位レベルの Audio Playback が、下位レベルの DAC モジュールに対して、flush_right (右寄せ) というデータ形式の使用を強制的に適用しています。

```
<requires id="module.framework.sf_audio_playback_hw_dac.requires.dac"
          interface="interface.driver.dac"
          display="Add DAC Driver" >
    <override property="module.driver.dac.data_format"
              value="module.driver.dac.data_format.data_format_flush_right"/>
</requires>
```

7.12.1 Property 要素（Property elements）

Name 要素（Name element）

各 module 要素で少なくとも、name プロパティのテキストフィールドを記述し、開発者がシンボル名（the name of the Symbol ）を入力する必要があります。このプロパティは、各 XML にとって最初のプロパティであることが必要です。

name フィールド（field）は、モジュールのこのインスタンスに関連付けられている <api>_instance_t 構造体（structure）の名前です。name フィールドを使用して、interface 属性（attribute）で module を識別します。チャネル番号を指定するためにこの name を選択します。その結果、チャネル番号が変化する場合でも、アプリケーションコードに変更を加える必要がなくなります。

以下の属性を使用して、name プロパティ要素をマニュアル（手作業）で追加します。

- default= g_<api>
- display=Name
- id= module.driver|framework.<api>.name
- constraint 要素
- testSymbol= ${module.driver|framework.<api>.name}

この constraint 要素については、これより後の章「**プロパティ要素の Constraint**」で説明します。

次に、GPT に対応する例を示します。

```
<property default="g_timer" display="Name" id="module.driver.timer.name">
    <constraint display="Name must be a valid C symbol">
        testSymbol("${module.driver.timer.name}")
    </constraint>
</property>
```

*_cfg_t の以下の各フィールドの設定に対応するプロパティ要素

- 列挙型（Enumerations）
- stdint/stdbool 型 (bool、uint8_t、int16_t など)
- 構造体/共用体（Structure/Unions）
  *_cfg_t 構造体の中に他の構造体が存在している場合、各構造体/共用体の各要素を設定するためのプロパティ要素を作成します。
  id= module.<driver|framework>.<api>.structure_union_name_element_name

以下は、GPT から取得した 例です。この行は timer 拡張（extension）に属しており、.gtiocb は stop_level という要素を持つ構造体です。結果として得られる id は、module.driver.timer.gtiocb_stop_level です。

```
<property default="module.driver.timer.gtiocb_stop_level.pin_level_low"
          display="Stop Level"
          id="module.driver.timer.gtiocb_stop_level">
```

- ポインタ（Pointers）
  *_cfg_t 構造体の中に任意のポインタが存在している場合、ポインタを設定するためのプロパティ要素を作成します。
  コールバック/コンテキストポインタ（callback/context ponters）の場合、次の章「**コールバックとコンテキストのプロパティ要素**」を参照してください。
- 1 行で記述する Typedef/Forward 宣言（Declaration）

7.12.2 コールバックとコンテキストのプロパティ要素（Call back and context property elements）

以下で、p_callback と p_context の各引数（argument）を指定するプロセスを説明します。

以下の属性（attribute）を使用して、callback プロパティ要素をマニュアル（手作業）で追加します。

- default= NULL
- display= Callback
- id= module.driver|framework.<api>.p_callback
- constraint 要素
- testSymbol= ${module.driver|framework.<api>.p_callback}

以下は、GPT に対応する 例です。

```
<property default="NULL"
          display="Callback"
          id="module.driver.timer.p_callback">
    <constraint display="Name must be a valid C symbol">
        testSymbol("${module.driver.timer.p_callback}")
    </constraint>
```

7.13 プロパティ要素の constraint（Property element constrains）

注記：現在、1 つの <property> 要素に対し、 1 つの <constraint> 要素が許されています。作成するプロパティで複数の constraint (制約) を使用するには、**&&** と **||** の各演算子を使用して、それらの制約を組み合わせてください。

テキストエントリ（text entry）の各種プロパティ (option を指定していないプロパティ) で constraint を追加して、テキストのタイプを指定できます。一般的な constraint の例を示します。

- `testInteger(${<id>})` - 入力が、 JavaScript 式であるかどうかを判定します。
  `("${module.driver.block_media_on_spi_flash.block_size}" > 0)`

```
<constraint display="Value must be an integer greater than 0">
    testInteger("${module.driver.timer.period}") &amp;&amp;
    ("${module.driver.timer.period}" &gt; 0)
```

- `testSymbol(${<id>})` - 入力が、C の有効なシンボルかどうかを判定します。
  この記述は、シンボルが C の有効なシンボルかどうかだけを判定します。C のそのシンボルがプロジェクト内で定義されているかどうかを判定しません。例えば、`bad_ch@racters` は C の有効なシンボルではありませんが、`my_variable` は C の有効なシンボルです。

```
<constraint display="Name must be a valid C symbol">
    testSymbol("${module.driver.timer.name}")
</constraint>
```

7.14 Header 要素（Header element）

`header` 要素は、開発者が自らのアプリケーションコードで使用するインスタンス構造体（instance structure） (`<api>_instance_t`) などを、extern (外部) グローバル変数（global variables）として定義します。この要素から取得したテキストは、開発者がアクセスできる (`src/ssp_gen` フォルダ内にある) ヘッダファイル（header file）に直接コピーされます。

`header` 要素（element）を追加します。インスタンス構造体 (つまり、`<api>_instance_t`) を extern (外部) 形式で宣言します。

コールバック関数（callback function）を指定する場合、header セクションの先頭でプロトタイプを記述する必要があります。このような記述を行うには、コールバックが NULL かどうかを最初に判定する必要があります。NULL は定義済みのマクロ（macro）なので、コールバックが定義済みであるかどうか判定できます。コールバックが NULL ではない場合、プロトタイプは宣言済み（declared）です。

以下のように、GPT (timer インタフェース) から取得した例に基づいてコールバック関数のプロトタイプを追加します。

以下は、GPT から取得した 例です。

```
<header>
    /** Timer on GPT Instance. */
    extern const timer_instance_t ${module.driver.timer.name};
    #ifdef ${module.driver.timer.p_callback}
    #define TIMER_ON_GPT_CALLBACK_USED_${module.driver.timer.name} (0)
    #else
    #define TIMER_ON_GPT_CALLBACK_USED_${module.driver.timer.name} (1)
    #endif
    #if TIMER_ON_GPT_CALLBACK_USED_${module.driver.timer.name}
    void ${module.driver.timer.p_callback}(timer_callback_args_t * p_args);
    #endif
</header>
```

7.14.1 Includes 要素（Include element）

includes 要素で、必須のインクルードパス（include path）を記述します。この要素は、開発者がアクセスできる (src/ssp_gen フォルダ内にある) ヘッダファイルに直接コピーされます。

必須のインクルードパスをすべて指定して includes 要素を追加します。通常、このパス指定は r_<instance>.h です。

以下は、GPT から取得した 例です。

```
<includes>
    #include &amp;quot;r_gpt.h&amp;quot;
</includes>
```

7.14.2 Declarations 要素（Declarations element）

declarations 要素で、open 呼び出し（open call）の際に必須となるデータの宣言（declaration）を記述します。このデータは、C のプライベートファイル（private file）にコピーされ、開発者がアクセスできるヘッダファイルへ（header file）と拡張されます。開発者は通常、インスタンス構造体（instance structure）を通じてすべてのデータにアクセスできます。この構造体は、開発者が指定した **name** に従って作成および命名されたものです。

- インスタンス構造体が指す制御構造体を作成するには、declarations 要素を編集します。

```
<declarations>
static timer_ctrl_t ${module.driver.timer.name}_ctrl;
</declarations>
```

- *_cfg_t 構造体の中にあるすべての要素を適切に編成するには、declarations 要素を編集します。property 要素セクションで property 要素を追加した場合、各要素に対応するコードも追加します。
  — 以下は、GPT timer の出力比較拡張（output compare extension） (構造体の中に他の構造体がある) から取得したです。gpt_timer_ext_t 構造体の中に gtiocb 要素があります。この要素は、output_enabled という要素を持つ他の構造体です。この要素に対応する property は、module.driver.timer.gtiocb_output_enabled です。
  — <api>_cfg_t 構造体を初期化するために、コールバック（callback）とコンテキスト（context）も追加する必要があります。コンテキストは、インスタンス構造体へのポインタ（pointer）にする必要があります。

```
<property default="module.driver.timer.gtiocb_output_enabled.false"
          display="Gtiocb Output Enabled"
          id="module.driver.timer.gtiocb_output_enabled">

    <option display="True"
            id="module.driver.timer.gtiocb_output_enabled.true"
            value="true"/>

    <option display="False"
            id="module.driver.timer.gtiocb_output_enabled.false"
            value="false"/>

</property>

...
<declarations>

static const timer_on_gpt_cfg_t ${module.driver.timer.name}_extend =

{

    .gtioca = { .output_enabled = ${module.driver.timer.gtioca_output_enabled},

                .stop_level     = ${module.driver.timer.gtioca_stop_level}

              },

    .gtiocb = { .output_enabled = ${module.driver.timer.gtiocb_output_enabled},

                .stop_level     = ${module.driver.timer.gtiocb_stop_level}

              }

};

static const timer_cfg_t ${module.driver.timer.name}_cfg =

{

    .mode                = ${module.driver.timer.mode},

    .period              = ${module.driver.timer.period},

    .unit                = ${module.driver.timer.unit},

    .duty_cycle          = ${module.driver.timer.duty_cycle},

    .duty_cycle_unit     = ${module.driver.timer.duty_cycle_unit},

    .channel             = ${module.driver.timer.channel},

    .autostart           = ${module.driver.timer.autostart},

    .p_callback          = ${module.driver.timer.p_callback},

    .p_context           = &amp;${module.driver.timer.name},

    .p_extend            = &amp;${module.driver.timer.name}_extend

};
</declarations>
```

— 以下は、GPT から取得した 例です (構造体へのポインタ)。 transfer_info_t 構造体は、transfer_cfg_t 構造体より先に作成済みです。また、p_info 要素は前者の構造体を指しています。

```
transfer_info_t ${module.driver.transfer_on_dtc.name}_info =
{
    .dest_addr_mode      = ${module.driver.transfer.dest_addr_mode},
    .repeat_area         = ${module.driver.transfer.repeat_area},
    ...
};
Static const transfer_cfg_t ${module.driver.transfer.activation_source}_cfg =
{
    .p_info              = &amp;${module.driver.transfer.name}_info,
    ...
};
```

— 制御構造体（control structure）と設定構造体（configuration structure）が作成された状態で、両者を指すインスタンス構造体（instance structure）を作成することができます。また、このインスタンス（instance）に対応する API 構造体へのポインタも追加します。このモジュールを使用するあらゆる状況で、このインスタンス構造体を共通して使用することになります。通常、名前（name）として g_<api>_on_<instance> を指定します。

```
<declarations>
/* Instance structure to use this module. */
const timer_instance_t ${module.driver.timer.name} =
{
    .p_ctrl        = &amp;${module.driver.timer.name}_ctrl,
    .p_cfg         = &amp;${module.driver.timer.name}_cfg,
    .p_api         = &amp;g_timer_on_gpt
};
</declarations>
```

7.15 Init 要素（Init element）

init 要素 (この時点では単純なフレームワークレイヤ) で、open 関数（function）を呼び出すためのコードを記述します。このコードは、src/ssp_gen/<user_thread_name>.c の中にある開発者のスレッド（thread）に合わせて生成されたコード中で呼び出されます。このコードは、(src/<user_thread_name>_entry.c 内にある) 開発者のスレッドコード（thread code）より先に実行されます。

フレームワークレイヤ open 関数（framework layer open function）を呼び出す init 要素を追加します。

以下に、audio フレームワーク（framework）から取得した例を示します。

```
<init>

    ssp_err_t ssp_err_${module.framework.sf_audio_playback.name};

    ssp_err_${module.framework.sf_audio_playback.name} =
        ${module.framework.sf_audio_playback.name}.p_api->
        open(${module.framework.sf_audio_playback.name}.p_ctrl,
        ${module.framework.sf_audio_playback.name}.p_cfg);

    if (SSP_SUCCESS != ssp_err_${module.framework.sf_audio_playback.name})

    {

        while (1);

    }
</init>
```

ホームページとサポート窓口

サポート: https://synergygallery.renesas.com/support

テクニカルサポート：

- アメリカ： https://www.renesas.com/en-us/support/contact.html
- ヨーロッパ： https://www.renesas.com/en-eu/support/contact.html
- 日本: https://www.renesas.com/ja-jp/support/contact.html

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2018.01.23	-	初期リリース
1.01	2018.02.02	-	構文と使用法に関する小規模な編集
1.02	2018.05.09		図 10, 11 を修正
		22	“module_descriptions” を “.module_descriptions”に訂正
		25	e² studio インスタレーションを e² studio バージョンに訂正

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品、本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準： コンピュータ、OA機器、通信機器、計測機器、AV機器、
   家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、
   金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
6. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
10. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
12. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

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

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

(Rev.4.0-1 2017.11)

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口 http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/