Renesas Synergy™ Platform

SSP Module Development Guide

R11AN0132EU0103
Rev.1.03
Oct 1, 2018

Introduction

Using the Renesas Synergy™ Platform provides developers with qualified and ready to use software modules and frameworks within the Synergy Software Package (SSP). Software developers may want to create their own modules, drivers, and frameworks to package them and distribute them to other developers. This guide provides developers everything they need to create their own modules, drivers, and frameworks, in addition to the processes necessary to package and distribute them.

Target Device

Synergy MCU Family

Recommended Reading

SSP User's Manual introduction chapters
SSP Datasheet v1.2.0 or later
SSP Development Best Practices Guide

Note: If you are not familiar with the above documents, you should review them before continuing.

Purpose

This document provides you, the developers, with the information you need to develop your own modules, drivers, and frameworks, along with the necessary details for creating Synergy Configurators and packaging them for distribution.

Intended Audience

The intended audience are users that understand the Synergy Platform's fundamentals and want to create their own drivers, modules, and frameworks on MCU's.

Contents

1. Driver Development Overview 4

2. Creating the Software Content 4

2.1 Module file structure organization 4

2.2 Filling in the details from scratch 6

2.3 Filling in the details from an existing module 6

3. Creating a Module Configurator XML File for the ISDE 6

3.1 Module configurator overview 6

3.2 Creating the XML 8

3.3 XML file naming conventions 8

3.4 Module configurator XML file sections and tags 9

3.5 Multiple configurators per Module in XML files 10

3.6 Module configurator checklist 10

3.7 Module configurator dictionary 12

4. Packaging the New Software Module 14

4.1 The PDSC (pack descriptor) 14

4.2 The Custom Pack Creator Tool – e^2 studio 14

4.3 Modifying the custom pack from above to include the XML files 19

4.4 Pack Creation for IAR Embedded Workbench 21

5. Using the Custom Synergy Module 22

5.1 Install the Custom Pack 22

6. Creating Custom packs for Wi-Fi module 23

7. Appendix – Rules for the Module Configurator XML File 23

7.1 Best practices for user-visible text 23

7.2 Content of text visible to user 23

7.3 Using elements as variables 23

7.4 Config element 24

7.5 Attributes id, path, and version 24

7.6 Property elements 24

7.7 Module element 24

7.8 Attributes and the idea of “common” 25

7.9 Constraint element 26

7.10 Provides interface element 26

7.11 Requires interface element 26

7.12 Override element 27

7.12.1 Property elements 27

7.12.2 Call back and context property elements 28

7.13 Property element constraints 29
7.14 Header element 29
7.14.1 Includes element 29
7.14.2 Declarations element 30
7.15 Init element 32

Revision History 34

1. Driver Development Overview

The Synergy Platform provides you with modules and frameworks that are ready to use out of the box and can be configured using graphical configurators known as Synergy Configurators. There is a general process that you can follow to create your own custom modules, configurators, drivers, and frameworks, and then distribute them as a pack that shows up in the Synergy Configurator menus. This process requires the steps shown in Figure 1.

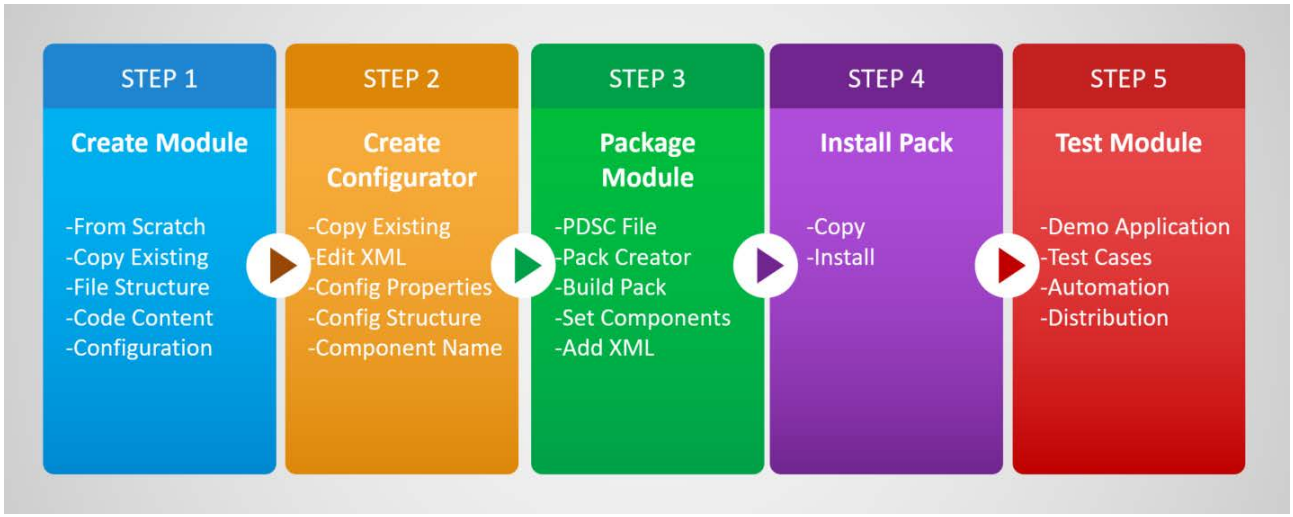

Figure 1 Steps to create a custom module

As a developer, you may have a slightly different starting point, depending on whether you are starting from scratch, or are looking to modify an existing module. In general, the recommendation is that you review the modules that already exist in the SSP and then copy, paste, and modify them to get your desired result. This helps in significantly simplifying and speeding-up the process.

The Synergy module is composed of three different pieces of information that are required to package and deliver the module.

1. The content (such as code and documentation) to be distributed with the pack.
2. The PDSC file (that is, the pack descriptor and final `.pack` file).
3. The XML configurator for the module, that is, a XML file that is located in `.module_descriptions`.

The following sections walk you through the steps that are necessary to create and package your own custom SSP modules. In general, throughout this document, when discussing creating the module, the concept can also be extended to include creating drivers and frameworks.

2. Creating the Software Content

The first step for creating a custom module is to create the software content. The software content includes the end file directory structure, configuration files, and software modules that include the header source files along with any additional files you may require, such as documentation or precompiled libraries. This section describes the recommendations and best practices for creating the software content.

2.1 Module file structure organization

Prior to writing or copying the software program, you should first create your new module's directory structure. The directory structure determines where the final packaged module is copied into a Synergy project, when in use. It is recommended that you create a local file structure that reflects a Synergy project in e^2 studio. For example, when creating a new Synergy Framework module named `sf_example`, you would create a folder for the framework under `synergy/ssp/src/framework/sf_example` as shown in the following figure. This is not a requirement, but it makes the process of creating the pack easier.

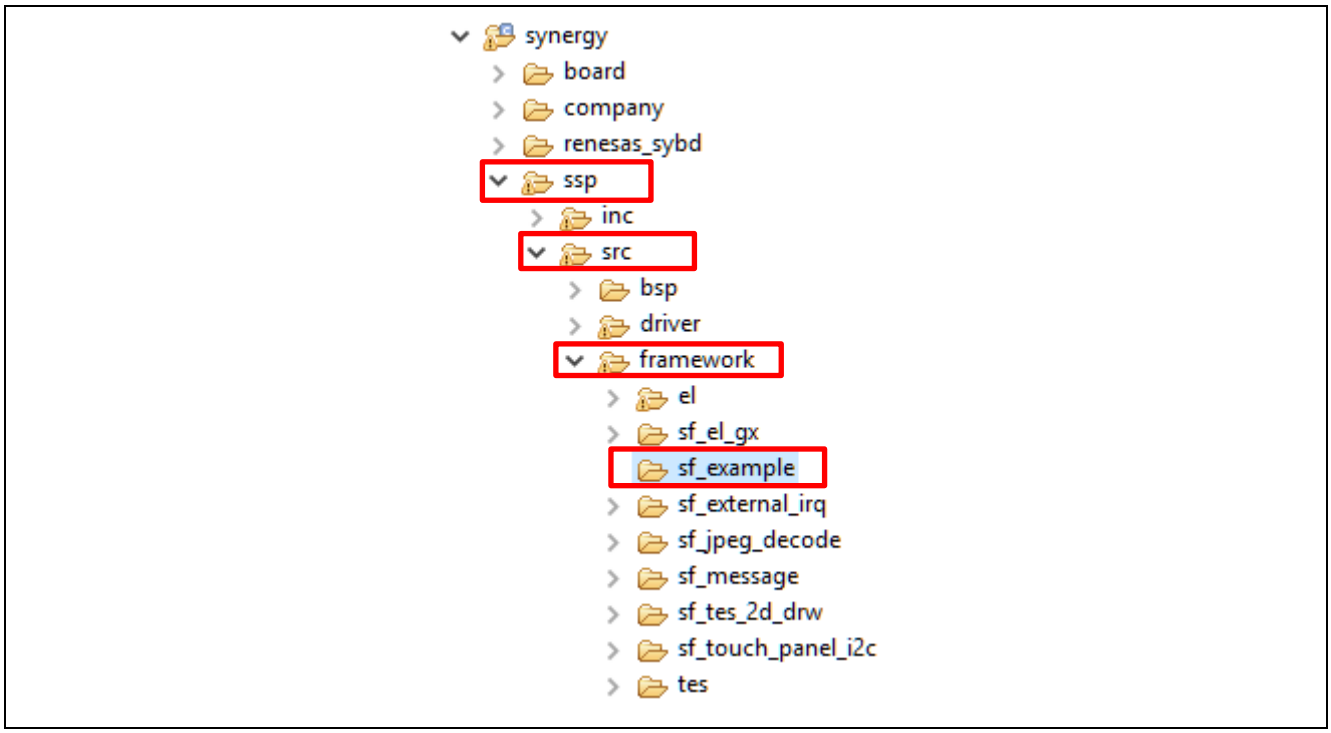

Figure 2 Example software framework folder structure

If you are a third-party developer, it's recommended that you do not create SSP modules, but instead, create your own custom modules and frameworks. In order to keep a clear distinction between the SSP and third-party components, the module should be in a distinct folder that identifies its source rather than `synergy/ssp`. Instead, you can use `synergy/<company>` as the base for the module. In which case, the previous `sf_example` module, `sf_example` code would be located under `synergy/<company>/sf_example` as shown in the following figure. Note that using the prefix `sf_` is optional, but it is a standard used by SSP to denote a software framework layer module.

When the custom module is eventually packaged, any content that is provided in a pack, when used in a project will be read-only, like all native SSP modules. Assuming this custom module is not protected (encrypted), you can modify the module and customize when using it in your project. However, if this module, after modification, is located in the same folder as the original module, then on any subsequent builds, or when the **Generate Project Content** button is pressed, the original module will be re-extracted from the pack and any changes to it will be overwritten and lost in the process. Overwriting changes applies to the pack generated code only. When developing a custom module, you can create your own file structure and code in the Synergy directory without being concerned with your changes being overwritten until you create a pack.

Note: Once a pack is created, installed, and added to a project, your directory is subject to overwriting. Make sure that at this stage, any changes are backed up diligently to prevent losing work.

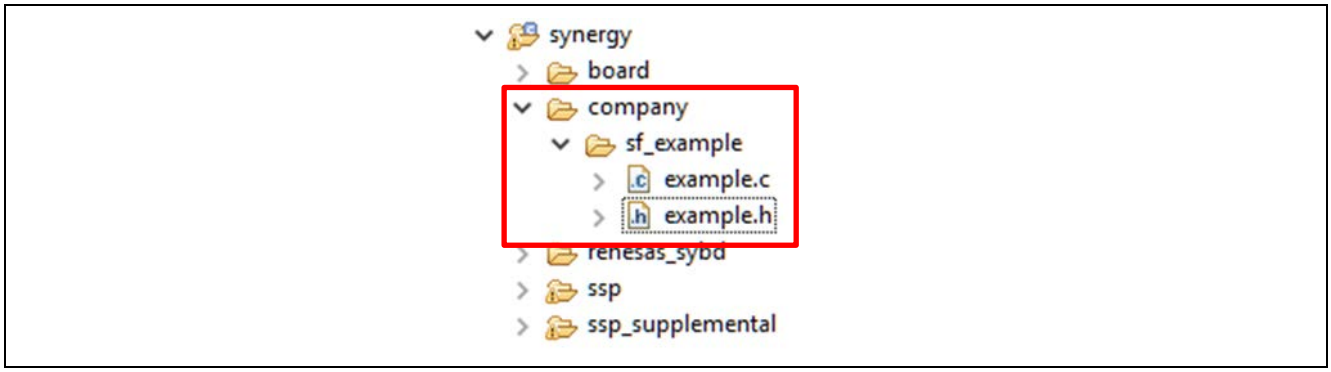

Figure 3 Custom module location for non-SSP modules

All contents in the `synergy_cfg` directory are typically not provided in a pack. The reason for this is that the contents of the `synergy_cfg` directory are expected to be generated by e^2 studio based on the values provided in the

Synergy Configurator entries. Due to the read-only nature of content extracted from a pack, providing configuration files in `synergy_cfg` can create problems in your projects. It is best to provide default values for the configuration and let the toolchain generate the configuration. This process has been further discussed in later sections.

2.2 Filling in the details from scratch

As a developer, you should try to reuse as much of the existing code as possible to decrease time, costs, and efforts in developing a new module. In some cases, you may be able to copy an existing driver and then modify it to include all the changes necessary to support additional functionality or custom behavior of the driver. In other cases, you may not be able to leverage existing code and will need to instead start from scratch. If you are starting from scratch and wish to create a new module, the following steps can be used to help you get started.

1. Extract the contents of synergy_module_template.zip (or any Application Project zip folder).
2. Rename the example Interface file, modify, and move it to destination directory, as necessary. For example, `synergy/ssp/inc/framework/api/sf_example_api.h`
3. Rename the example Instance file, modify, and move it to destination directory, as necessary. For example, `synergy/ssp/inc/framework/instances/sf_example.h`
4. Rename the source files in the example source folder, modify, and move it to destination directory, as necessary, for example, `synergy/ssp/src/framework/sf_example`.

2.3 Filling in the details from an existing module

If you are a developer planning to start from an existing module, such as the CRC HAL Driver module, you can follow the steps that are provided below:

1. Create a Synergy project with the module you want to start from.
2. Make a copy of the existing module's source folder and rename the folder.
   - For example, copy the `synergy/ssp/src/driver/r_crc` folder to `synergy/ssp/src/driver/r_mydriver`.
3. Rename the source files in this renamed folder.
   - For example, change `synergy/ssp/src/driver/r_crc/r_crc.c` to `synergy/src/driver/r_mydriver /r_mydriver.c`.
4. Modify the newly copied source.
5. Copy the existing module's Instance header file.
   - For example, copy `synergy/ssp/inc/driver/instances/r_crc.h` to `synergy/ssp/inc/driver/instances/r_mydriver.h`.
6. Modify the newly copied Instance header file.
7. Repeat as necessary to fill in a new framework module.

At this point, you will have working software content that is ready to be packaged. Before the module can be packaged, you should create an XML configurator file so that you can easily set the module properties through the Synergy Configurator. The next section describes the XML file and how to create it for a custom module.

3. Creating a Module Configurator XML File for the ISDE

3.1 Module configurator overview

The XML configurator file allows the new module to appear in the Synergy Configurator **Threads** tab, from where it can be added to an application thread, and easily configured through the **Properties** view.

The following figure illustrates how the XML configurator file is organized and how it is related to the ISDE (integrated solution development environment) and generated code. The module configurator XML file is located in the `.module_descriptions` folder that is not displayed in the ISDE, but can be found in the project folder. The XML data contains two primary elements, the `config` element, and `module` element. The `config` element contains configuration settings that affect how the module behaves on an application level, while the `module` element determines how the specific instance behaves.

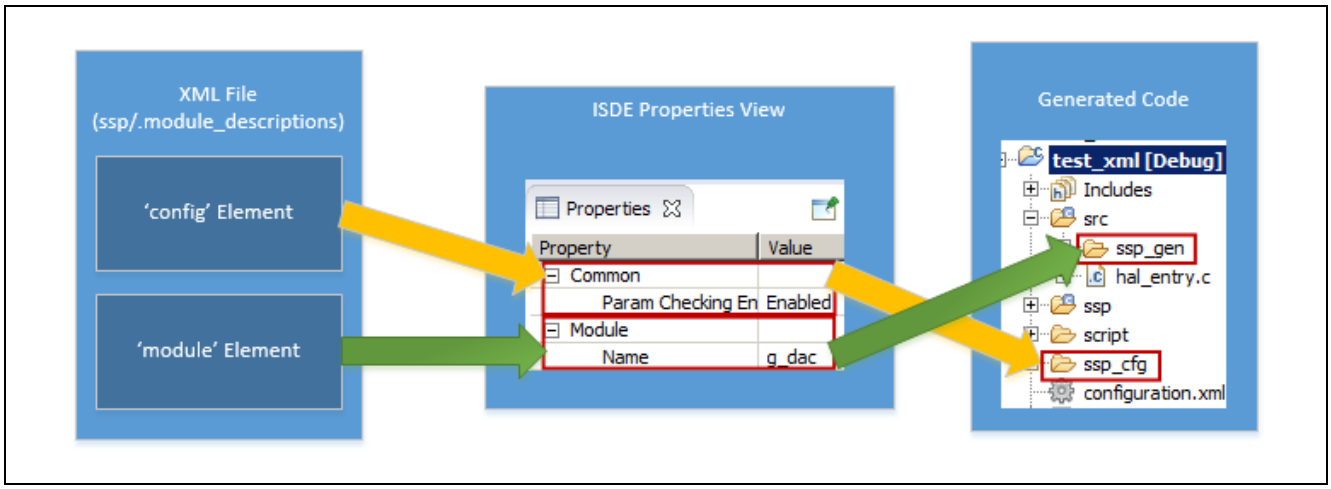

Figure 4 Generating configuration information from the XML file

The XML data is presented as a GUI in the ISDE when the module is added to a project in the **Threads** tab and then the developer clicks on it. The configuration settings are available in the **Properties** view. The XML data generates files in two distinct project areas. First, the module element data generates code in the `src/synergy_gen` folder, that is the instance specific configuration and shown in the figure above by the green arrows. Second, the `config` element data generates code in the `synergy/ssp_cfg` folder, that is the common, build-time configuration represented by the yellow arrows in the above figure.

The ISDE has a flexible module configuration tool that is data driven. You can customize your module's configurator XML file, that then uses the standard eclipse **Properties** view to provide plain text and simple to interpret configurators for you to set the configuration fields. The following figure shows a sample configurator that you might create for your module, and how that configuration information is used to generate code.

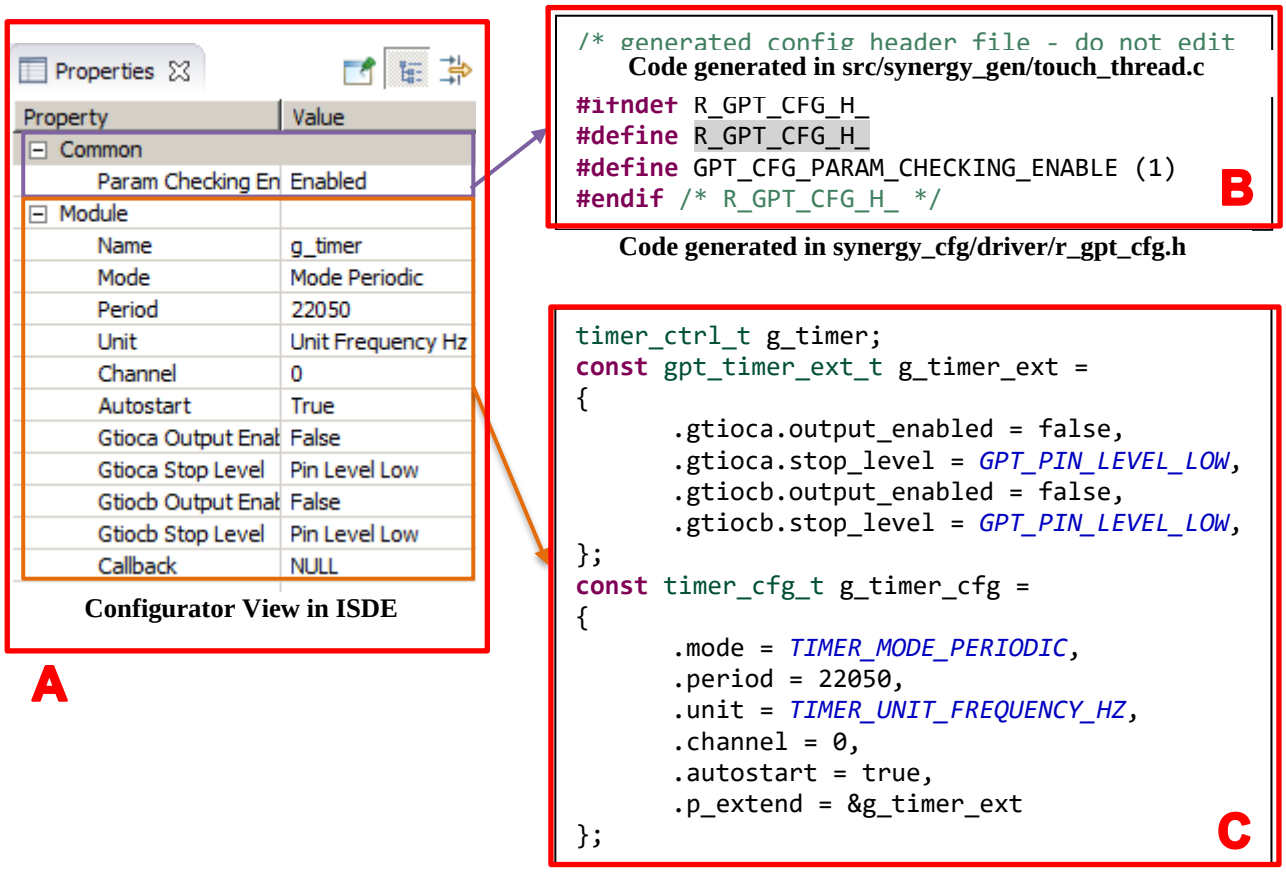

Figure 5 Configurator View in ISDE that Generates Configuration Header and Source Code

After the configuration is complete, you can press the **Generate Project Content** button to generate parameters required for the module. This creates an instance of the module in the `src/synergy_gen` folder of the project and common code in the `synergy_cfg` folder. The example in the above figure-A shows the timer module that is used as part of the audio framework in a thread named `touch_thread`. After generating the project content, the common configuration properties are generated in the configuration header file as shown in the above figure-B, while the module configuration properties are generated in a code module in the above figure-C.

Note: When you create your software program, you also write your own XML file that matches these configuration values for testing the module. This code then can be used to develop the configurator XML file that will then create the configurator for other users.

3.2 Creating the XML

The easiest way to create an XML configurator is to use one that already exists. The process is as follows:

- Identify an XML that has similar configuration elements.
- Create a copy of the XML file.
- Rename the XML file using the XML Naming Conventions section recommendations in the next section.
- Edit and update the XML file for the new module.
- Perform a peer review and/or test to make sure the file has been updated successfully.

For example, if you concluded that you need to create your module based on the r_crc module, you would copy the existing module's Instance XML configurator file, rename it, and then edit it per the above process.

This would require:

- Copying `.module_descriptions/Renesas##HAL Drivers##all##r_crc####1.2.0.xml`
- Pasting and renaming to `.module_descriptions/Renesas##HAL Drivers##all##r_mydriver####1.2.0.xml`

The following sections describe these steps and examine the XML file conventions.

3.3 XML file naming conventions

The ability to build software stacks and generate runtime code for Synergy modules is provided through XML configurators. These are found in the `.module_descriptions` folder in your project. These XMLs are extracted from all the available packs of a certain version when a project is created. These XMLs are also what drive the options under the new stack button in the **Threads** tab. **The file name of the XML configurator must match the settings in the `<component>` element in the PDSC file**. The following figure shows an example for the component element in PDSC file.

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

Figure 6 Component element in the PDSC file in the pack

The XML file must be named Renesas##Company##all##sf_example####1.2.0.xml, where the colored text matches the information provided in the PDSC component element. If any of these options do not match, the component appears in the **Components** tab, but will not be available under the new stack tree in the Synergy Configuration **Threads** tab.

The name that is given to each module's XML configuration file determines where the module is shown in the **ISDE components** tabs. For example, naming a configurator as follows will result in the component being shown in the Synergy Configurator **Components** tab as shown in the following figure.

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

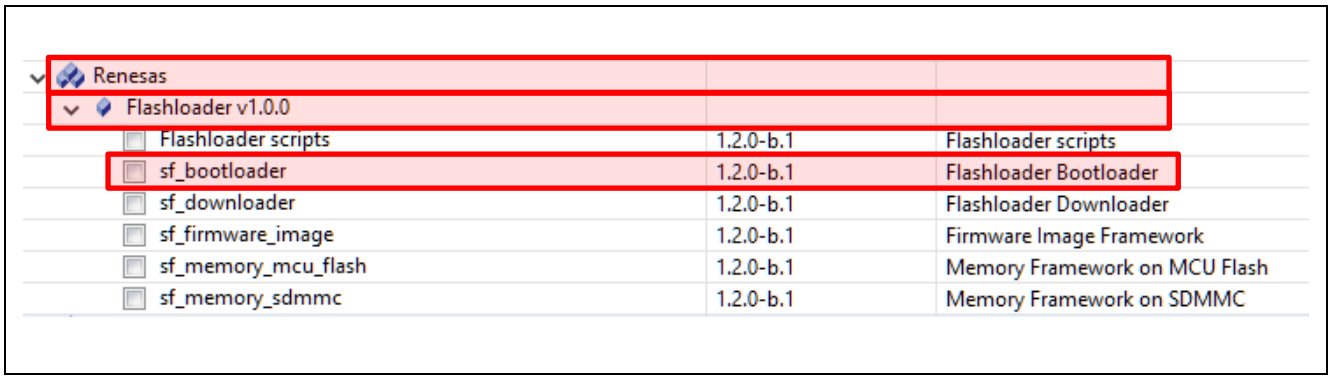

Figure 7 Naming the XML file based on its display location

If the module is being developed by a third party, you can name their component using the following template:

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

The last part of the name that comes after #### is the SSP version that the component is designed and tested to work with.

There are two rules that you should follow when naming the configurator XML files:

1. The Cvendor attribute of your `<components>` must be set to Renesas. This is required even if you are not building a module on behalf of Renesas Electronics or its subsidiaries. If this attribute is not set appropriately, then the `<component>` will not appear correctly in the e^2 studio **Components** tab.
2. The name of your XML configurator must start with Renesas to match the Cvendor attribute of `<component>`.

3.4 Module configurator XML file sections and tags

After the configurator XML has been created, it is useful to understand the different sections that make up the configurator XML file. The module configurator XML file is divided into two main elements – `config` and `module`. The main distinction is that the sub-elements of the `config` element are persistent over all module instances (in the `synergy_cfg` folder), while the sub-elements of the `module` element belong to a single instance of a module. For example, if an application uses two GPT timer channels, the per-channel configurations (used to define the `timer_cfg_t` structures) are sub-elements of the `module` element in the XML file, while the shared configurations (used to define macros in `synergy_cfg/driver/r_gpt_cfg.h`) are sub-elements of the `config` element of the XML file.

Each section can contain various XML tags that dictate how the module behaves, such as what configuration information is displayed to you, which modules are dependencies, which are provided interfaces, and so on. The following table shows the available XML tags and provides a brief description on the usage of each.

You can copy and paste example XML Tags into their configurator in order to complete their task. Developers who would like additional details and examples on how to use these tags can examine section 7 Appendix – Rules for the Module Configurator XML File.

Table 1 XML tags

XML Tag	Purpose
`<requires>`	Identifies the components that are necessary for the module to function. These are component dependencies.
`<provides>`	Conveys what the component provides to the module that is dependent upon this component.
`<constraints>`	Defines a constraint that must be met for the module to function. An example is that the instance must have a unique name.
`“Name”`	Uniquely names data structures to avoid duplication errors.
`<override>`	Can be used within `<requires>` when hardcoding a dependency option.
`<property>`	Creates configuration options that you can configure.
`<option>`	Provides different configuration dropdown options that exist within the `<property>` tab.

<config>	Defines the high-level configuration options that exist for the module that will apply across all instances.
<module>	Defines elements that will apply to a single module instance.
<header>	Defines extern global variables such as instance structures that will be used by a developer in their code.
<includes>	Contains required include paths and is copied directly into the generated header file in src/synergy_gen.
<declarations>	Contains declarations of data required for the open call. Data is copied into a private C file and extended into a header file accessible by the developer.
<init>	Contains code to call the open function. This is the code generated for your thread and is executed before your thread code (entry function).

3.5 Multiple configurators per Module in XML files

Once you have started to create a module, you may want to know how you can create multiple configurators for a single module. The NetX™ module is a good example where three basic configurators exist for a single module. NetX has configurators for:

- The common core code
- Creating IP instances
- Creating Packet Pool Instances

These three configurators all use the same code from the nx folder in SSP. To enable multiple configurators from one module, you need to create additional <config> and <module> elements inside the same XML file. A good place to look at a working example is inside the file:

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

The XML file contains two <module> elements and two <config> elements. One is for the Shared Bus and the other is for a Device.

3.6 Module configurator checklist

The configurator XML files are straight forward, but they can very quickly become complicated. The following checklist can be used to make sure that everything necessary has been included.

Item	Yes/No
Config Property - Parameter checking required	
Module Description – The description that is provided to the user when the user clicks on a certain property	
Module provides interface element	
Module requires interface element with overrides (if applicable)	

Module override element:
If your module requires a lower level module specified by the `<requires>` element, and your module requires certain settings, use the `<override>` subelement of the `<requires>` element to force settings in the lower layer.
`<override property="module.driver|framework.<lowerlevelapi>.<lowerlevelid>" value="module.driver|framework.<lowerlevelapi>.<lowerlevelid>.<lowerlevelvalue> "/>`
An example for the `sf_audio_playback_hw_dac` framework is given below. Here the upper-level Audio Playback on DAC is forcing the lower level DAC module to use a flush-right data format:
`<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"/>`
Property Elements	

Module Property elements	
Module Header element	
Module Includes element	
Module Declarations Element	

Modules with lower level drivers require the following:

Item	Yes/No

Module
An example for ThreadX Source where only 1 is allowed.
`<module config="config.el.tx_src"`
`id="module.framework.tx_src"`
`display="Framework|RTOS|ThreadX Source"`
`common="1"`
`version="0">`
An example for a I2C Framework Shared Bus where an unlimited number (“100” is effectively that) of shared Module Instances is allowed
`<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 element	

Module Requires interface element	
Module Init element12X	

3.7 Module configurator dictionary

The way that the configurator XML is processed requires that certain characters be written in specific fashion to display themselves properly. For example, you cannot use “>” but instead needs to use “>”. A list can be found in the following table for more alternative characters in the XML files.

Name	Category	Description
===	Javascript	Use to test equality in constraints, like '==' in C
||	Javascript	Like '||' in C, used in constraints
&	Javascript	Use `&amp;` to create an ampersand (&) character in XML generated code
& &	Javascript	Like '&&' in C, used in constraints
>	Javascript	Use `&gt;` to create a greater than (>) character in XML generated code
<	Javascript	Use `&lt;` to create a less than (<) character in XML generated code
"	Javascript	Use `&quot;` to create a quotation mark (") character in XML generated code
config	Element	Contains properties of build time configurations that will go in `ssp_cfg/<driver|framework>/r_<module>_cfg.h`. Must have attributes id, path, and version.
config	Attribute	Attribute of module, must be equal to id attribute of `config` element for the module
constraint	Element	Framework to restrict invalid configurations
display	Attribute	Text visible to the user
declarations	Element	Text field with allocated `ctrl` and `cfg` data structures
header	Element	Text field with externed global variables (example: `extern <api>_ctrl_t <module_user_name>;`)
id	Attribute	Variable used in the XML. `${<id>}` resolves to the value parameter when used. Example: `${module.driver.timer.unit}` resolves to (`${module.driver.timer.unit.unit_frequency_khz}`, which resolves to) `TIMER_UNIT_FREQUENCY_KHZ` in all text fields visible to the user when the option **Unit Frequency Khz** is selected in the **Unit** dropdown of a GPT timer configuration. The id attribute should start with the id string of the parent element. Example: `${module.driver.timer.unit}` is the id for the unit property of the `${module.driver.timer_on_gpt}` parent module.
includes	Element	Text field with required include paths (example: `#include &quot; r_<instance>.h&quot;`)
init	Element	Text field with code to call the open function. Currently only used at the framework layer, called in `<user_thread_name>.c` before `<user_thread_name>_entry` is called
interface	Attribute	Used in provides and requires elements to tie modules together
macros	Element	Text field to define macros. Currently unused.
module	Element	Contains properties of run time configurations created as part of the `<user_specified_name>_cfg` structure passed into the open function. Must have attributes `config`, `display`, `id`, and `version`.
option	Element	Drop down option for a specific property (always a subelement of property). Each option must have `display`, `id`, and `value` attributes.
property	Element	Configuration that the user must specify. Properties are found in the `config` and `module` elements. Each property must have `default`, `display`, and `id` attributes. Properties are text fields if no options are specified, or dropdowns if options are specified.
provides	Element	Provides an interface to tie driver to upper layers and for use by constraints to ensure only one instance of each channel is used.
requires	Element	Used by (typically framework) modules that require an interface to a lower level driver.
override	Element	Locks lower level settings in the `requires` element.
value	Attribute	Value that `${<id>}` resolves to.
common	Attribute	Only valid in `<module>` element. This determines whether a Module Instance can be shared between SSP stacks.

4. Packaging the New Software Module

4.1 The PDSC (pack descriptor)

CMSIS-Packs are used to deliver content to users in e^2 studio and IAR Embedded Workbench for Synergy. Information about CMSIS-Packs can be found at:

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

CMSIS-Packs are made up of two parts:

- The content to be delivered (for example, code, documents)
- The PDSC (Pack Descriptor) file that describes the contents

The following figure shows the different information that must be included in the PDSC file.

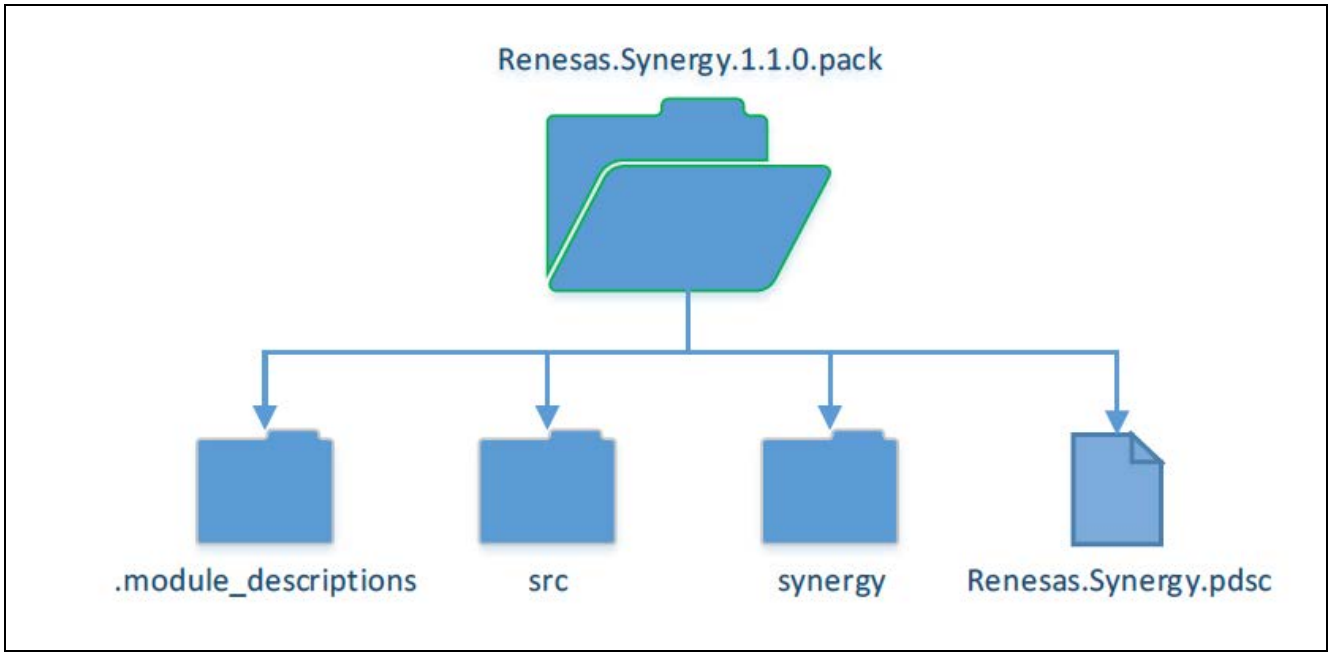

Figure 8 Example Pack file

CMSIS-Packs are zip files with the extension `.pack`. In the root of the archived file is a file with the extension `.pdsc`. The PDSC file name should be the same as the Pack filename without the version number. There is a format for naming packs which are; `<vendor>.<name>.<version>.pack`. The SSP Pack's filename is `Renesas.Synergy.1.2.0.pack`.

The PDSC file contains XML with an associated schema that is discussed here:
http://www.keil.com/pack/doc/CMSIS/Pack/html/packformat.html

The PDSC file can be automatically generated by using the export capabilities within e^2 studio, specifically the Custom Pack Creator Tool. For this reason, the details of the PDSC file won't be covered.

4.2 The Custom Pack Creator Tool – e^2 studio

The SSP and add-on software associated with the Synergy Platform are distributed in a pack format. Packs are a convenient way to collect and distribute software to developers in an organized way. Packs are located within the installation directory located under `\e2_studio\internal\projectgen\arm\Packs`. Custom modules and frameworks must also be distributed in the same manner. Exporting a custom pack is a relatively straight forward process in e^2 studio 5.2.1 or later.

Note: The Custom Pack Creator Tool is only available with e^2 studio versions 5.2.1 and later. This feature is not available with IAR EW for Synergy as of version 7.71.1.

To create a pack file, open the project that contains the custom code. From the **File** menu, select **Export**. The following figure shows the dialog box that appears.

Under **General**, select the **Renesas Synergy User Pack** option and click **Next**.

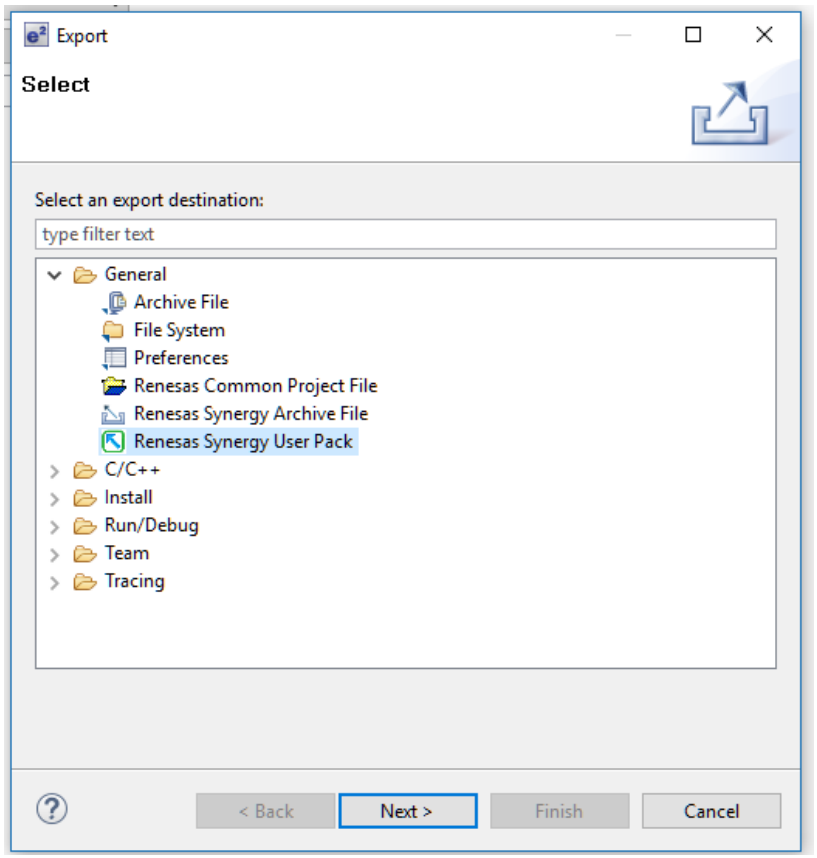

Figure 9 Exporting to a Pack

The next dialog box that is displayed gives you the opportunity to provide not only the pack name but also a variety of other parameters about the pack such as the version, description, and contact information. At this stage, you will want to fill in all the details that are shown in the following figure. Keep in mind that this first screen provides you with the ability to name your pack. Remember that the pack should be named `<Company>.<Component>.<SSP Version>`. The pack name that is generated is highlighted in the following figure.

The pack information is not saved by default. If you want to test the pack, make changes, and then create another pack, you will need to use the save and open features within the pack creator. These are also highlighted in the following figure and can be found in the upper right-hand corner.

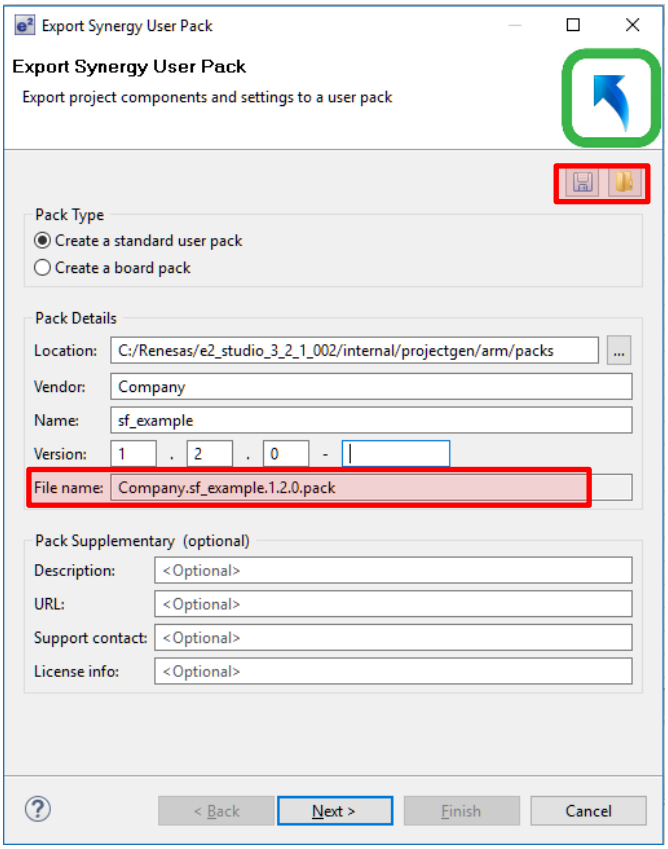

Figure 10 Setting the Pack Information

Once the pack name and information is entered, you can now select the components, threads, and messaging components that will be included in the pack as shown in Figure 13. The green plus icon in the component selection box (shown in the red box in the figure below) can be used to create a new component that will be included in the pack file. Pressing it brings up the dialog box seen in Figure 11.

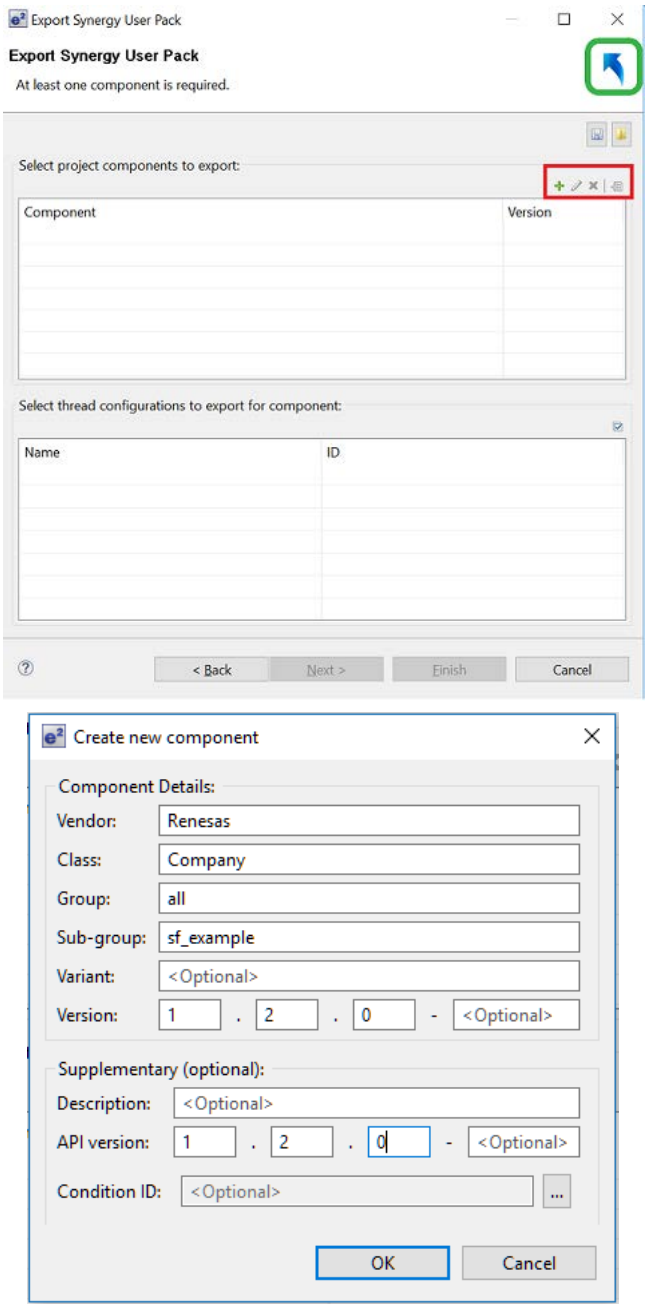

Figure 11 Creating a New Component

Accurate naming of the component is critical as mentioned in section 3.3. The component name needs to match that of the XML file. The naming convention can be seen again in the following figure. These fields directly map to the component name. Not matching them properly results in the component not displaying in the stack menus under the **Threads** tab.

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

Figure 12 XML File Naming Convention

There are several rules that you should follow to name your packs. These include:

1. The `<version>` portion of the XML name must contain numbers separated by periods. Four version fields are accepted if the third has text. Valid examples:
   A. 1.0.0
   B. 1.1.0
   C. 1.1.0-beta.1.
2. Packs are expected to be tested against a set version of SSP. This means that if you want your module to be used with SSP, it must have a matching version number. If your module will work with multiple versions of SSP, then you must create separate packs with different versions that are compatible with the respective SSP version. Examples where Renesas.Synergy.1.1.0.pack (a Renesas supplied SSP pack) is currently being used:
   A. Renesas.SynergyExample.1.1.0.pack will be displayed.
   B. Renesas.SynergyExample.1.0.0.pack will not be displayed.
   C. Renesas.SynergyExample.1.2.0.pack will not be displayed.
3. e^2 studio ignores version all fields after the second field. For example, when Renesas.Synergy.1.1.0.pack is used, all the packs below will be included:
   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.

When the component has been created, you will need to select the threads and files that you want to package. This can be done by first checking the new component and then clicking on the component name. When this is done, the dialog like the one shown in the following figure appears.

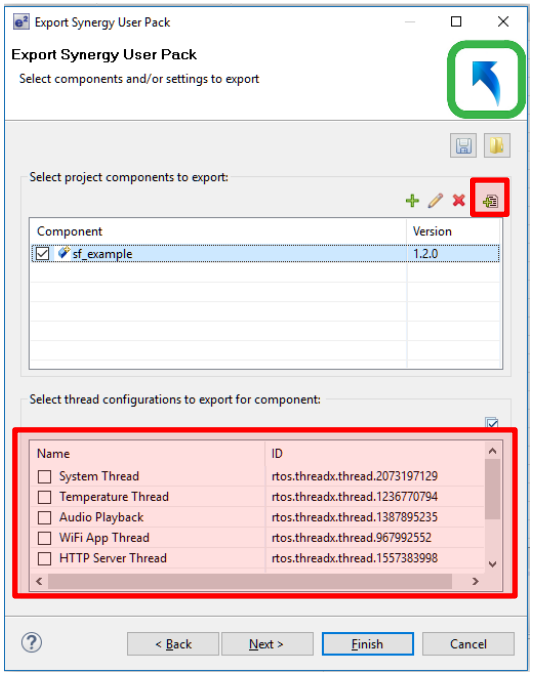

Figure 13 Adding Components to Export

You can select threads to include in the pack by checking the thread at the bottom of the window. The add file button, located in the upper right, can also be used to add files. The following figure is an example of how you would add your new files to the component. On the left-hand side, files that are available for export are displayed. You can check the files that you want to include, and then press the **Add** button in the middle of the screen to add them to the component files. Finally, at the bottom, the include path for the files can be added so that they will be automatically added to the project when you select the component.

Keep in mind that you cannot export:

1. SSP components and SSP source files (for example, `<Project>\synergy\ssp`)
2. SSP generated files (for example, `<Project>\synergy_cfg\ssp_cfg`)

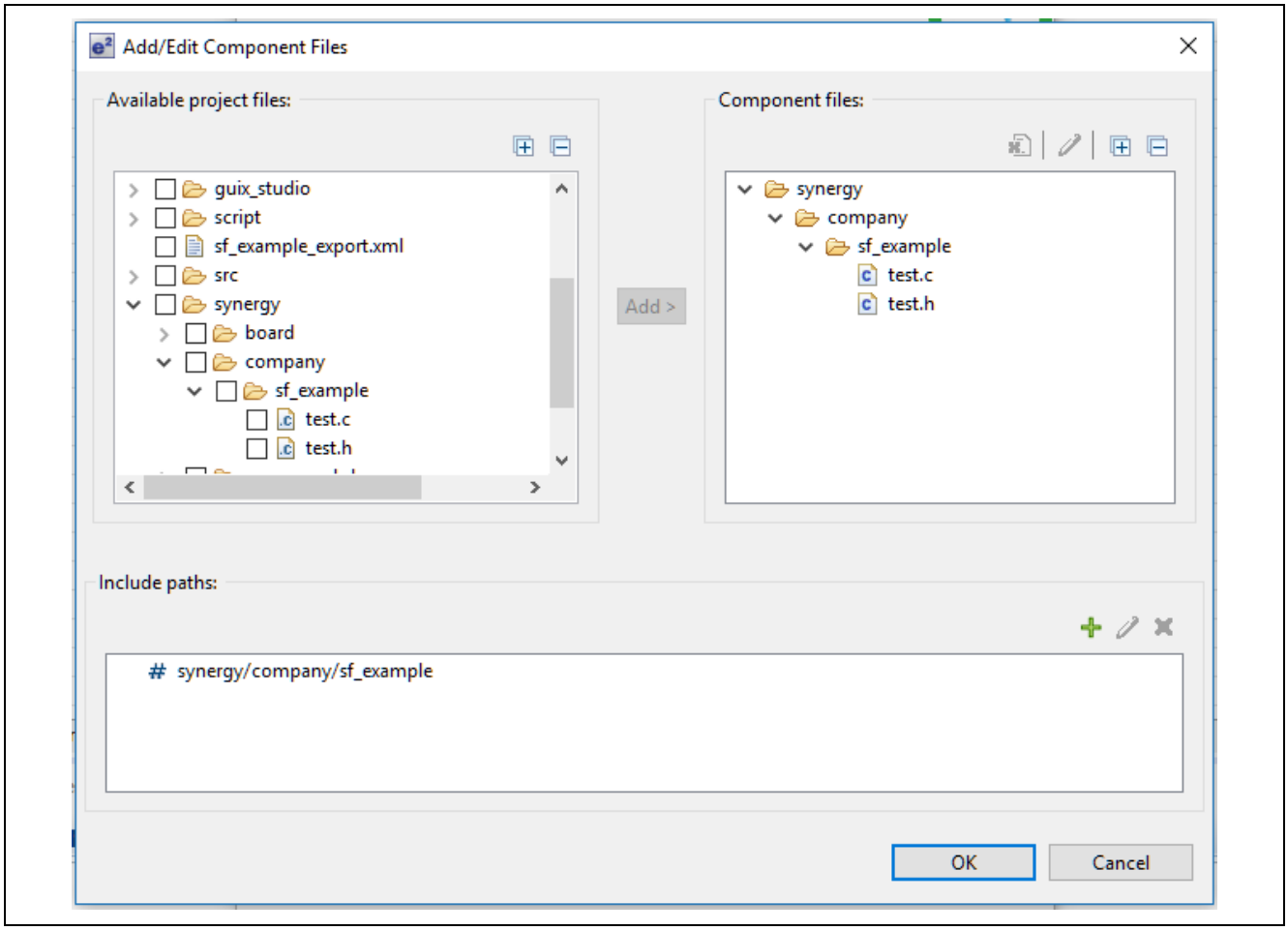

Figure 14 Select Files and Set Include Paths

After adding files, you can click **Finish** to generate the pack. Alternatively, you can click **Next** and select pack conditions to export. These conditions specify the exact compiler, target processor, and so forth. If the settings are to be reused, make sure that the configuration settings are saved before clicking finish and generating the pack.

4.3 Modifying the custom pack from above to include the XML files

As of e^2 studio v5.4.0.023, Module Configurator XML files cannot be directly added to the pack files from the `.module_descriptions` folder. These files should be manually added to the packs.

1. Extract the pack file to a folder as shown in the following figure. Delete the `.pack` file.

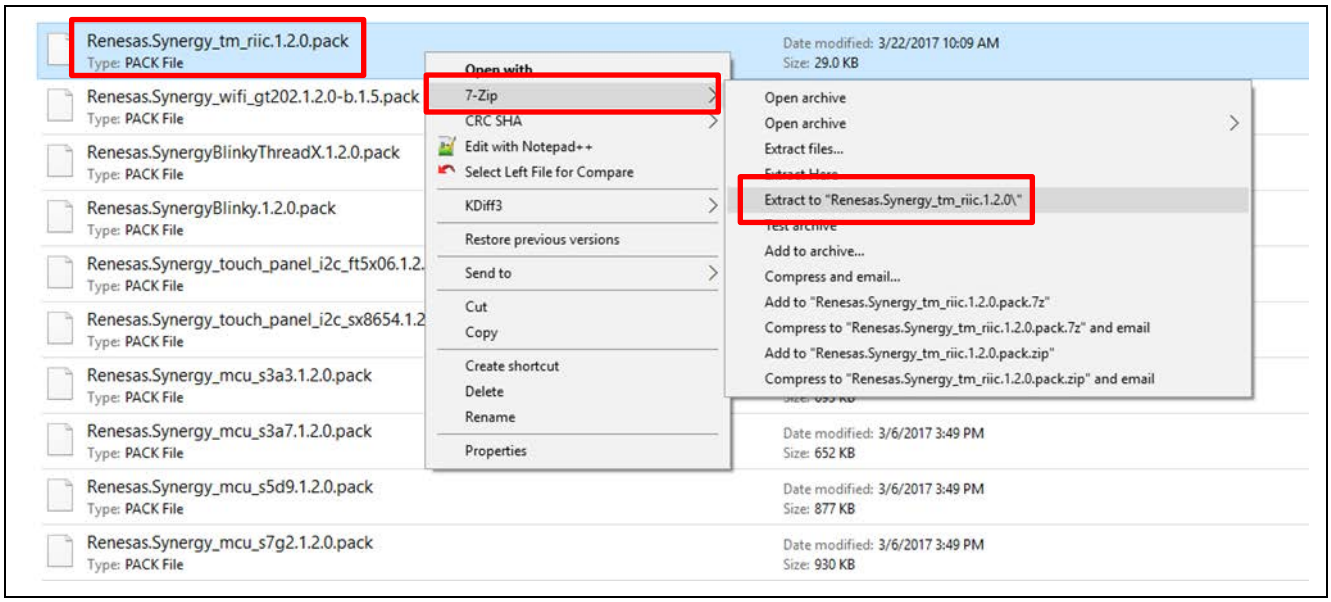

Figure 15 Extracting the contents of pack file

2. Copy the module configurator XML files into the `.module_descriptions` folder. Create a new `.module_descriptions` folder if one is not already there. The contents to be packed should look as shown in the following figure.

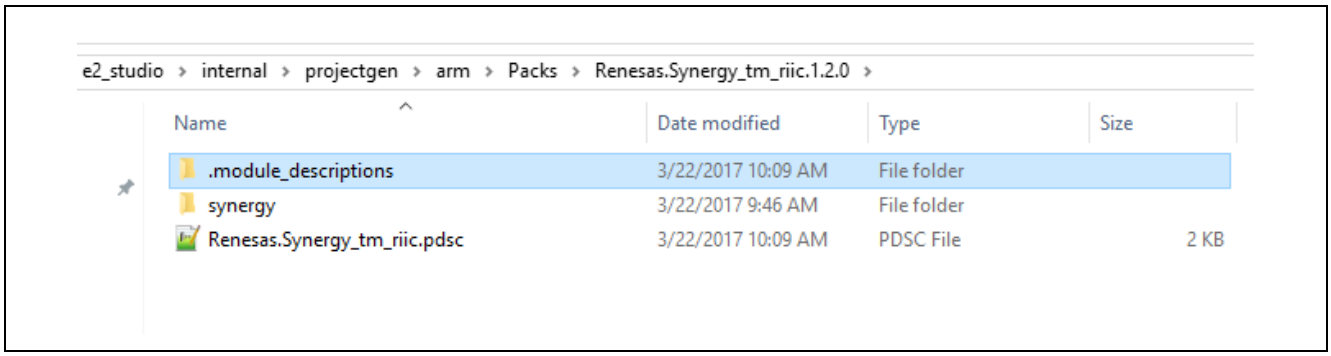

Figure 16 Contents of folder to be packed

3. Compress the contents of the folder into a .zip file as shown in the following figure. Copy the .zip file to `e2_studio/internal/projectgen/arm/Packs` folder. Delete the uncompressed original folder.

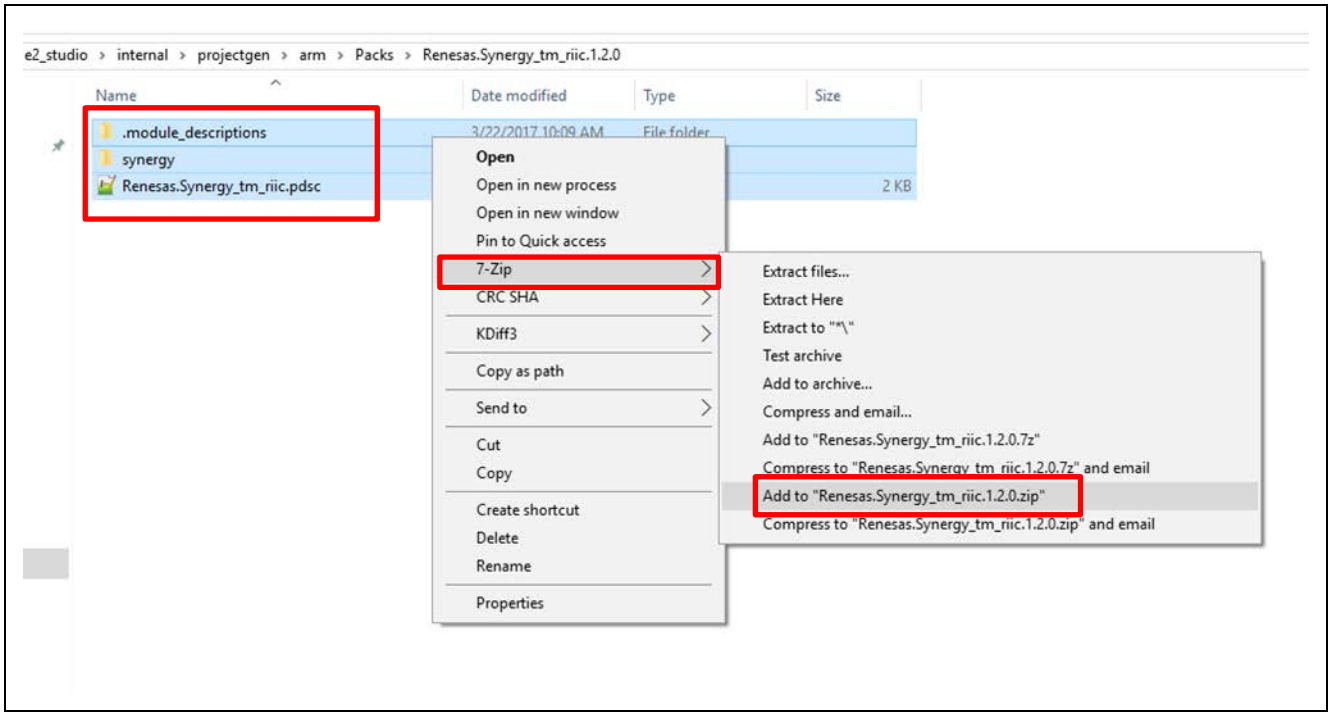

Figure 17 Compress the contents of folder into .zip file

4. Rename the .zip file to .pack file to look like **Renesas.Synergy_tm_riic.1.2.0.pack**. The final folder should look as shown in the following figure.

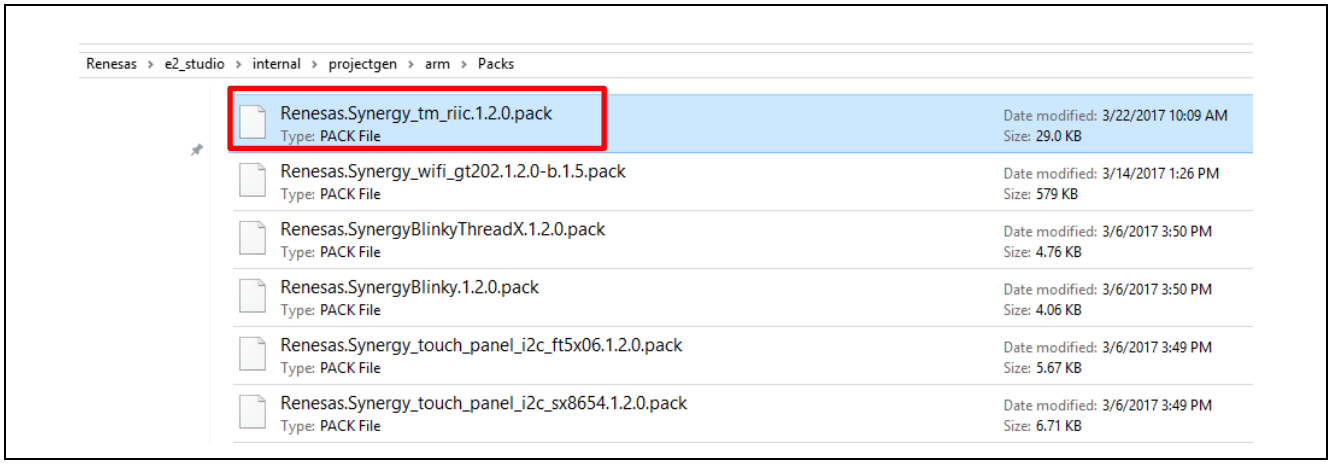

Figure 18 Packs folder

4.4 Pack Creation for IAR Embedded Workbench

As there is no Custom Pack Creator Tool available in IAR Embedded Workbench, all the pack files should be created manually as below:

1. Create the contents folder.
2. Manually create the PDSC file.
3. Create module configuration XML files and place them in `.module_descriptions` folder.
4. Add the above contents to a .zip file and create a .pack file.

The only extra step needed in this case would be manual creation of PDSC file.

Also, care should be taken with the pack file, XML files, and PDSC file naming convention.

5. Using the Custom Synergy Module

5.1 Install the Custom Pack

Copy your new pack to the `<e2_studio_install_folder>/internal/projectgen/arm/Packs` directory. If you have a Synergy Configuration window already open, e² studio has a window pop up asking you to refresh as seen in the following figure. If a Synergy Configuration is not open, the pack list will be refreshed the next time one is opened.

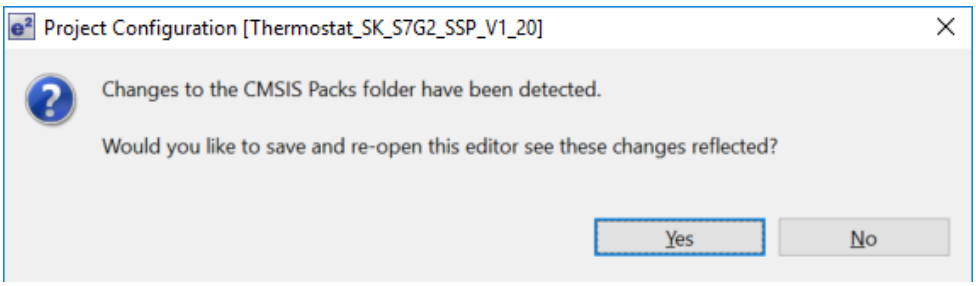

Figure 19 Pack refresh dialog

There are some very important rules that must be followed to make sure that your pack will be properly recognized by e² studio:

1. Packs are specific to a particular e² studio version. If you have multiple versions of e² studio, then you will have to install your pack into each version for it to be recognized and displayed in that version of e² studio.
2. Each SSP version is tested and released with an associated version of e² studio, identified in the Release Notes for SSP. When developing your module and XML, you must use the version of e² studio that is recommended for use with the version of SSP that you are using for the module.

It is useful to note that the PDSC component section is used to list the new software component in the Synergy Configurator tool. If a module has an XML configurator associated with it, the new module appears in the module options as shown in the following figure.

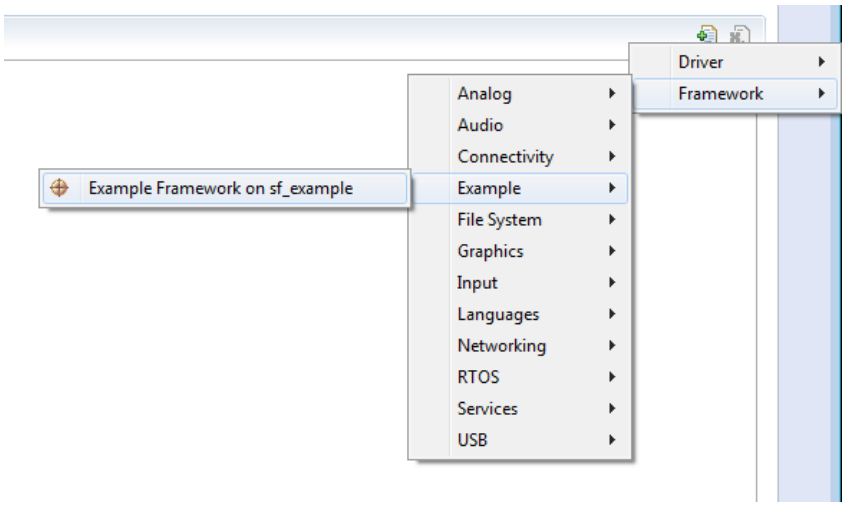

Figure 20 sf_example has a Configurator and appears in the Configurator menu

When you choose a module in the **Threads** tab, e² studio then checks the associated component in the **Components** tab. If a module does not have an XML configurator, you must check the component manually in the **Components** tab.

When a component is chosen, only the files that are described in the `<component>` are extracted. Here is the `<components>` element of the example PDSC file.

Note: Custom Pack Installation process is the same as above for IAR Embedded Workbench as well with the exception that the pack files need to be placed into `<SSC_install_folder>/internal/projectgen/arm/Packs`.

6. Creating Custom packs for Wi-Fi module

For creating custom packs for Wi-Fi modules, use the Wi-Fi module templates that come with the *Wi-Fi Porting Guide Application Project*. Follow the steps defined in the Wi-Fi porting guide for creation and installation of packs.

7. Appendix – Rules for the Module Configurator XML File

This section goes into greater detail on the different tags that are available within the Module Configurator XML file. These additional examples and rules will provide you with the detailed knowledge you need to customize the XML file.

7.1 Best practices for user-visible text

All text fields visible to the user (attributes starting with display=) must be human readable.

Manually, edit all display ids so that abbreviations are spelled out and the acronyms are in the proper case. For example, the bolded text shown below was originally Khz and was edited to kHz to make it easy to read.

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

7.2 Content of text visible to user

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

7.3 Using elements as variables

Note: Element ids are like variables that can be used in any text fields, including display attributes for other elements and in code elements such as the `Declarations` element. Order is not important. A text field can resolve an element that is defined later in the XML file. To use the variables, `${<element_id>}` resolves to the value selected by the user.

In the example below, `${module.driver.timer.name}` will resolve to `g_timer` (default attribute), if the Name field is not edited by the user.

```
<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">
```

In the example below, when Unit Frequency kHz is selected in the ISDE:

1. The value of `module.driver.timer.unit` is set to the value attribute of `module.driver.timer.unit.unit_frequency_khz`
2. `${module.driver.timer.unit}` resolves to `TIMER_UNIT_FREQUENCY_KHZ` in any text fields where it is referenced.

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

7.4 Config element

The first element in the XML files is the `config` element. The `config` element contains build time configurations that will go in the `ssp_config/<driver|framework>/<namespace>_<instance>_cfg.h` file.

Config elements, like the `<namespace>_<instance>_cfg.h` header files they populate, are common to all instances of a module. For example, if parameter checking is changed from enabled to disabled in a TIMER driver on GPT2, it will also be automatically changed from enabled to disabled in a TIMER driver on GPT5. The parameter checking option is configurable on the screen for each channel, but it cannot be defined to different options for different channels.

7.5 Attributes id, path, and version

The first line defining the `config` element should define the following attributes:

- id= `config.<driver|framework>.<instance>`
  This is used later by the <module> element to tie the <module> element to a particular <config> element.
- path= `ssp_cfg/<driver|framework>/<namespace>_<instance>_cfg.h`
  This is the path relative to the `synergy_cfg` folder where the output file (`<namespace>_<instance>_cfg.h`) will be generated.
- version= 1.0

An example from GPT is below:

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

7.6 Property elements

At a minimum, each config element contains a property drop down with three options for the parameter checking macro required by each module:

1. Manually edit the BSP option element (see example below).
2. Manually edit the default value to `config.driver.<module>.param_checking_enable.bsp`.
3. Manually edit the display value to **Parameter Checking.**
4. Manually edit the values of any other build time configurations as appropriate.

An example of the GPT properties is below:

```
<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 element

Elements in the module apply to a particular instance of a module. For example, a single application can use more than one GPT channel. If GPT channel 2 and GPT channel 5 are used, each will have their own elements saved (in contrast to elements of config, which are shared).

7.8 Attributes and the idea of “common”

The first line defining the module element should define the following attributes:

```
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) common= **n**.

There are three ways a module can be used in a software stack. The appropriate option to choose depend on whether the common attribute is found in the `<module>` element, and its value.

1. Common attribute is not found.
   - This is the default case. This means that this Module Instance (that is, a block in the ISDE) can only be used in the current SW stack. If you have two threads with different software stacks, then a `<module>` with this setting can be used in one, but not the other. The other stack would have to create a new Module Instance. The same applies to multiple stacks in the same thread.
   - Most of the HAL drivers currently use approach. They are not thread-safe and are not meant to be shared. You can of course share them at the application level.
2. Common attribute is set to **1**.
   - The value of **1** means only one Module Instance of this module can exist. If you try to create a new Module Instance of this `<module>` when one already exists, then the only option will be to use the existing Module Instance.
   - Example of modules using this are X-Ware libraries. There can only be one NetX library. There can be multiple uses of IPs and packet pools, but the common library can have only one instance.
3. Common attribute is set to **n** (some integer)
   - An integer value greater than **1** means that this `<module>` can have **n** number of Module Instances. If the attribute’s value was **4** then it would allow 4 distinct Module Instances to be created. After that, an existing Module Instance would have to be chosen. We typically use a value of **100** to mean **unlimited** since **100** Module Instances should never occur in typical use cases.
   - Examples of this include SPI and I²C Framework buses and NetX packet pools. You may want to share a common packet pool between a NetX IP instance and a NetX HTTP Client.

The following is an example for GPT where the common attribute is not set.

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

The following is an example for ThreadX® source where only one instance is allowed.

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

The following is an example for an I²C Framework Shared Bus where an unlimited number (**100**) of shared Module Instances is allowed.

```
<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 element

Constraints are used to alert you of errors early in the development process. Constraints catch errors in the ISDE rather than when building or worse yet, when debugging.

Constraints are implemented as a condition that gets inserted into a JavaScript conditional statement
`if (!<constraint>) { /* Error */ }`

Note: When referring to properties in constraints, only module properties from your current module can be used. To constrain lower level modules, see `Override` element.

All modules that can have multiple instances must have the following constraint:

`display=”Module instances must have unique names”`

The following is an example for the audio framework:

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

7.10 Provides interface element

The `provides` element satisfies constraints in upper layer modules used to tie modules together. The interfaces named can be accessed by other modules, and resolve to the number of times they have been provided.

Add the following element(s):

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

If your driver supports more than one instance:

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

An example for GPT is below:

```
<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 element

If you require a lower level module (not including the BSP), add the following line (note that this is not required for most HAL modules):

Note: The interface attribute of the <requires> element must match the interface element of the <provides> element in the lower level module; this is how the connection between the two is made.

```
<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" />
```

- This option means that this module will be available in the **Threads** tab of the module configurator. Without this option, the module would be available in the **HAL** tab of the module configurator.

The following is an example for the sf_audio_playback framework:

```
<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 element

If your module requires a lower level module specified by the <requires> element, and your module requires certain settings, use the <override> sub-element of the <requires> element to force settings in the lower layer.

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

The following is an example for the sf_audio_playback_hw_dac framework. The upper level Audio Playback on DAC is forcing the lower level DAC module to use a flush-right data format:

```
<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 elements

Name element

At a minimum, each module element contains a name property text field for you to enter the name of the symbol. This property must be the first property in each XML.

The field **name** is the name of the <api>_instance_t structure associated with this instance of the module. The name field is used to identify the module in the interface attributes. The name was chosen for this purpose over the channel number so the application code does not have to change if the channel number changes.

Manually add a name property element with the following attributes:

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

This constraint is documented in **Property Element Constraints**.

The following is an example for the GPT:

```
<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>
```

The property elements for configuration of the following fields of *_cfg_t.

- Enumerations
- stdint/stdbool types (bool, uint8_t, int16_t, and so forth)
- Structures/Unions
  If there are structures in your *_cfg_t structures, create property elements for configuration of each element of each structure/union.
  id= module.<driver|framework>.<api>.structure_union_name_element_name

Following is an example from GPT. This is in the timer extension, where .gtiocb is a structure with element stop_level. The resulting id is module.driver.timer.gtiocb_stop_level.

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

- Pointers
  If there are any pointers in your *_cfg_t structures, create property elements for configuration of the pointer. For callback/context pointers, see the following **Callback and Context Property Elements**.
- Single Line Typedefs/Forward Declarations

7.12.2 Call back and context property elements

The process for specifying p_callback and p_context arguments are described below.

Manually, add a callback property element with the following attributes:

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

Following is an example for GPT:

```
<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 Property element constraints

Note: Only one `<constraint>` element is permitted per `<property>` element now. If your property has more than one constraint, combine them into one using **&&** and **|| operators**.

For all text entry properties (properties with no options provided), add constraints to specify what type of text is expected. Common constraints include:

- `testInteger(${<id>})` – to test if input is an integer related, JavaScript logic:
  `("${module.driver.block_media_on_spi_flash.block_size}" &gt; 0)`

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

- `testSymbol(${<id>})` – to test if input is a valid C symbol:
  This tests only that the symbol is a valid C symbol. It does not test that the C symbol is defined in the project. For example, `bad_ch@racters` is not a valid C symbol, but `my_variable` is.

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

7.14 Header element

The `header` element externs global variables such as the instance structure (`<api>_instance_t`) that you will use in your application code. Text from this element is copied directly into header files accessible by you (in the `src/ssp_gen` folder).

Add the `header` element. Extern the instance structure (that is, `<api>_instance_t`).

If you specified a callback function, a prototype must be provided at the top of the header section. To do this, you will first determine if the callback is NULL. NULL is a defined macro, so this is tested by checking if the callback is defined. If the callback is not NULL, a prototype is declared.

Add a callback function prototype based on the example from GPT (timer interface) below.

Following is an example from GPT:

```
<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 element

`The includes` element contains the required include paths and is copied directly into header files accessible by you (in the `src/ssp_gen` folder).

Add the `includes` element with all required include paths, typically `r_<instance>.h`.

Following is an example from GPT:

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

7.14.2 Declarations element

The `declarations` element contains declarations of data required for the `open` call. This data is copied into a private C: file and extended into a header file accessible by you. You will typically access everything through the instance structure that is created and named according to the **name** you have provided.

- Edit the declarations element to create a control structure to be pointed to by the instance structure.

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

- Edit the declarations element to ensure all elements of the `*_cfg_t` structures are populated appropriately. Add code for any property elements added in the property elements section.
  - Following is an example from the GPT timer output compare extension (structure within a structure). The `gpt_timer_ext_t` structure has an element `gtiocb`, that is a structure with an `output_enabled` element. The property for this is `module.driver.timer.gtiocb_output_enabled`.
  - The callback and context also must be added to initialize the `<api>_cfg_t` structure. The context should be a pointer to the instance structure.

```
<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>
```

— An example from DTC is below (pointer to structure). The `transfer_info_t` structure is created before the `transfer_cfg_t` structure, and the `p_info` element points to it.

```
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,
    ...
};
```

— With the control and configuration structures created, we can now create an instance structure that points to them. We will also add a pointer to the API structure for this instance. This will be common across all uses of this module. The name is typically `g_<api>_on_<instance>`.

```
<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 element

The `init` element (framework layers only at this time) contains code to call the open function. This code is called in the generated code for your thread in `src/ssp_gen/<user_thread_name>.c`. This code executes before your thread code (in `src/<user_thread_name>_entry.c`).

Add the `init` element with the call to the framework layer open function.

An example from the audio framework is below:

```
<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>
```

Website and Support

Visit the following vanity URLs to learn about key elements of the Synergy Platform, download components and related documentation, and get support.

Synergy Software	renesassynergy.com/software
Synergy Software Package	renesassynergy.com/ssp
Software add-ons	renesassynergy.com/addons
Software glossary	renesassynergy.com/softwareglossary
Development tools	renesassynergy.com/tools
Synergy Hardware	renesassynergy.com/hardware
Microcontrollers	renesassynergy.com/mcus
MCU glossary	renesassynergy.com/mcuglossary
Parametric search	renesassynergy.com/parametric
Kits	renesassynergy.com/kits
Synergy Solutions Gallery	renesassynergy.com/solutionsgallery
Partner projects	renesassynergy.com/partnerprojects
Application projects	renesassynergy.com/applicationprojects
Self-service support resources:	
Documentation	renesassynergy.com/docs
Knowledgebase	renesassynergy.com/knowledgebase
Forums	renesassynergy.com/forum
Training	renesassynergy.com/training
Videos	renesassynergy.com/videos
Chat and web ticket	renesassynergy.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jan 23, 2018	—	Initial release
1.01	Feb 2, 2018	—	Minor edits for grammar and usage
1.02	May 9, 2018	16, 17	Corrected Figures 10, 11
		20	Corrected “module_descriptions” to “.module_descriptions”
		22	Corrected “Packs are specific to a particular e2 studio installation.” to “e2 studio version”.
1.03	Oct 1, 2018	23	Procedure for creating Custom packs for Wi-Fi module has been moved to a separate document.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

RENESAS

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics Corporation
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 7.2