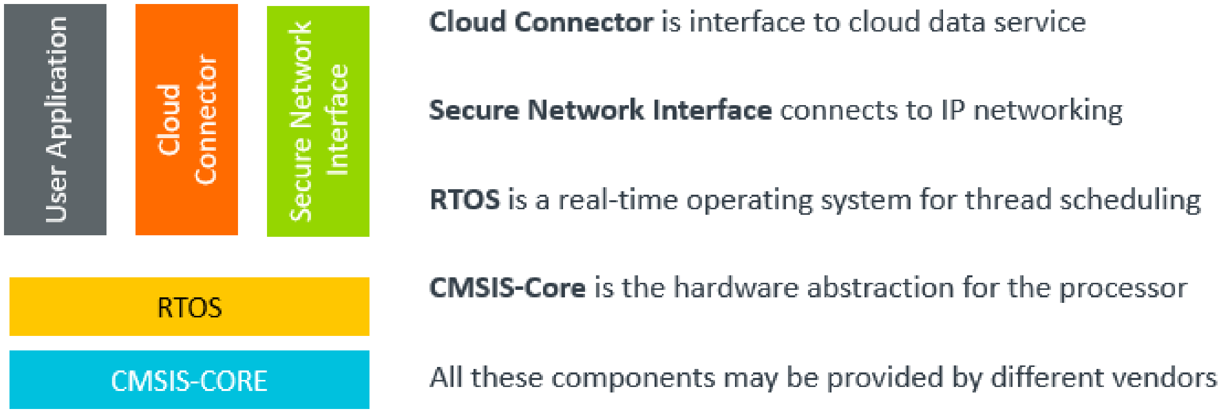
Scalable Software Stack with CMSIS-Pack Technology

Today’s IoT applications are based on software stacks (or software components) that are frequently provided by a Cloud vendor. As IoT endnotes have diverse requirements, these software stacks should be scalable to a wide range of devices. A complete IoT application may look like shown in the diagram below.



Cloud Connector is interface to cloud data service

Secure Network Interface connects to IP networking

RTOS is a real-time operating system for thread scheduling

CMSIS-Core is the hardware abstraction for the processor

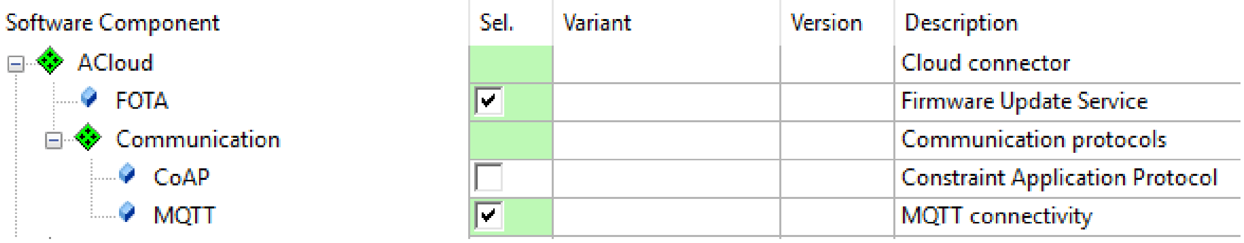
All these components may be provided by different vendors

Arm has developed the CMSIS-Pack system which makes it easy to combine software components that are developed independently and even from different vendors.

The CMSIS-Pack system supports today more than 6,000 different microcontrollers and provides ways to manage software components from different sources. A CMSIS-Pack can include a collection of software components provided as source code or library with header files and related documentation.

Software Component Lists

The CMSIS-Pack system gives you an intuitive user view to software components with the notion of **Cclass**, **Cgroup**, and **Csub** attributes. Below is the user view in the RTE window of a software component where the user can choose any combination of components for his project.



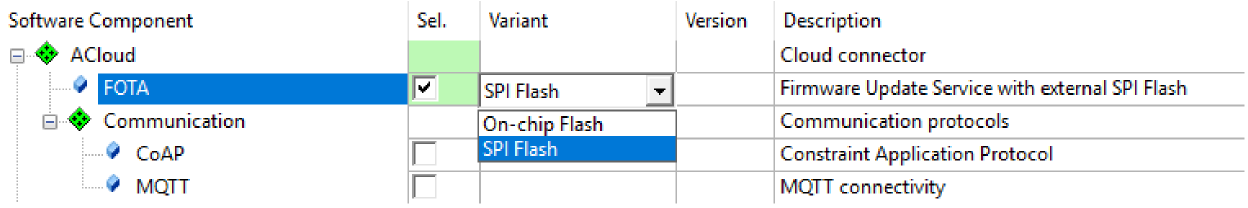
Software Component	Sel.	Variant	Version	Description
⊟ ACloud				Cloud connector
FOTA	☑			Firmware Update Service
⊟ Communication				Communication protocols
CoAP	☐			Constraint Application Protocol
MQTT	☑			MQTT connectivity

The relevant *.PDSC file of this software component list is shown here.

```
  <taxonomy>
    <description Cclass="ACloud">Cloud connector</description>
    <description Cclass="ACloud" Cgroup="Communication">Communication protocols</description>
  </taxonomy>

  <components>
    <component Cclass="ACloud" Cgroup="Communication" Csub="MQTT">
      <description>MQTT connectivity</description>
      <files>
        …
      </files>
    </component>
    <component Cclass="ACloud" Cgroup="Communication" Csub="CoAP">
      <description>Constraint Application Protocol</description>
      <files>
        …
      </files>
    </component>
    <component Cclass="ACloud" Cgroup="FOTA">
      <description>Firmware Update Service</description>
      <files>
        …
      </files>
    </component>
  </components>
```

Software Component Variants

Sometimes, it is required to choose between a Debug or Release variant, or for example for FOTA select between SPI or On-Chip Flash. The **Cvariant** attribute allows to choose between different variants of a component.

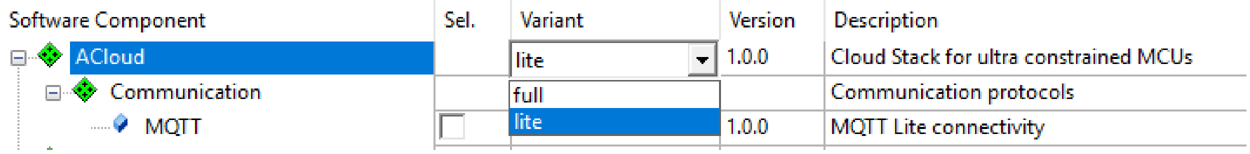


Software Component	Sel.	Variant	Version	Description
ACloud				Cloud connector
FOTA	☑	SPI Flash On-chip Flash SPI Flash		Firmware Update Service with external SPI Flash
Communication				Communication protocols
CoAP	☐			Constraint Application Protocol
MQTT	☐			MQTT connectivity

This is the related portion of the *.PDSC file:

```
    <component Cclass="ACloud" Cgroup="FOTA" Cvariant="SPI Flash">
      <description>Firmware Update Service with external SPI Flash</description>
      <files>
        …
      </files>
    </component>
    <component Cclass="ACloud" Cgroup="FOTA" Cvariant="On-chip Flash">
      <description>Firmware Update Service with on-chip device Flash</description>
      <files>
        …
      </files>
    </component>
```

If there are different tiers of the complete **ACloud** stack, for example a “Lite” and a “Full” version, the **Cbundle** attribute can be used. Cbundle is like a variant but affects the complete Cclass and allows to switch the whole component set. A caveat of Cbundle is that it limits how the Cclass component can be extended by other packs. Effectively all components of a Cbundle should be part of the same pack. The components of a Cclass outside of a Cbundle can be extended by other packs.



Software Component	Sel.	Variant	Version	Description
ACloud		lite full lite	1.0.0	Cloud Stack for ultra constrained MCUs
Communication				Communication protocols
MQTT	☐		1.0.0	MQTT Lite connectivity

This is the related portion of the *.PDSC file:

```
    <bundle Cbundle ="lite" Cclass="ACloud" Cversion="1.0.0">
      <description>Cloud Stack for ultra-constrained MCUs</description>
      <component Cclass="ACloud" Cgroup="Communication" Csub="MQTT">
```

```
        <description>MQTT Lite connectivity</description>
        <files>
          …
        </files>
      </component>
    </bundle>
    <bundle Cbundle ="full" Cclass="ACloud" Cversion="1.0.0">
      <description>Cloud Stack for featured MCUs</description>
      <component Cclass="ACloud" Cgroup="Communication" Csub="MQTT">
        <description>MQTT connectivity</description>
         …
      </component>
      <component Cclass="ACloud" Cgroup="Communication" Csub="CoAP">
         …
      </component>
      <component Cclass="ACloud" Cgroup="FOTA" Cvariant="SPI Flash">
         …
      </component>
      <component Cclass="ACloud" Cgroup="FOTA" Cvariant="On-chip Flash">
         …
      </component>
    </bundle>
  </components>
```

Component Selection exported to header file

Frequently, the source code of components has conditional compile sections that depend on the selection of other components. The CMSIS-Pack system offers multiple ways to generate #define symbols that depend on component selections:

- **Pre_Include_Global_h** is a pre-include file that is available to all source files in a project.
- **Pre_Include_Local_Component_h** is available to a specific software component.
- **RTE_Components.h** is a central header file that can be used as #include file in the whole project.

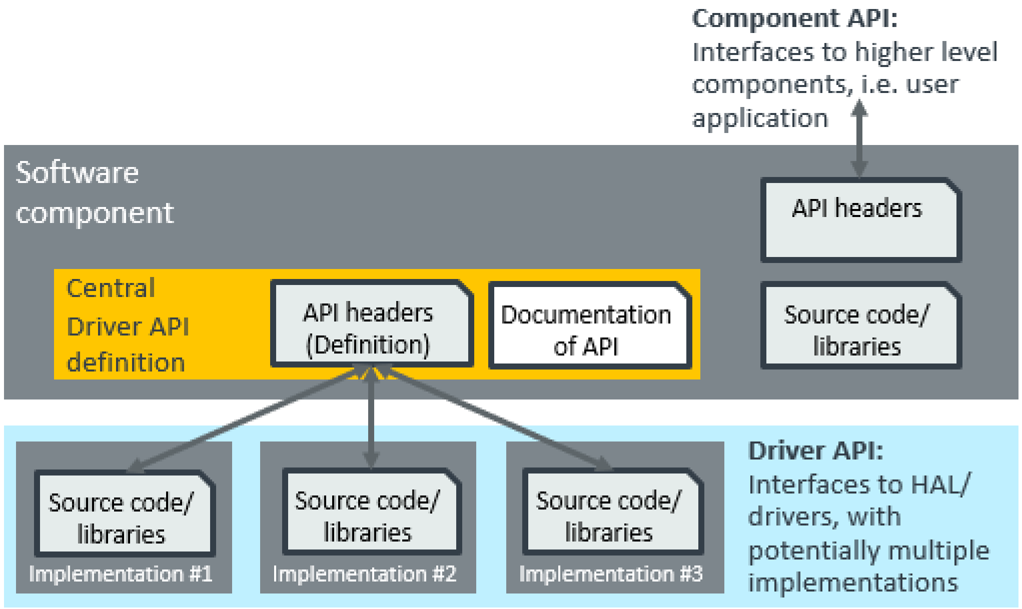
The pre-include variants have a similar effect as compiler define statements at the command-line. For example:

```
     <component Cclass="ACloud" Cgroup="Communication" Csub="MQTT">
        <description>MQTT connectivity</description>
        <Pre_Include_Global_h>
        #define ACloud_MQTT 1
        </Pre_Include_Global_h>
        <files>
          …
        </files>
      </component>
      <component Cclass="ACloud" Cgroup="Communication" Csub="CoAP">
        <description>Constraint Application Protocol</description>
        <Pre_Include_Global_h>
        #define ACloud_CoAP 1
        </Pre_Include_Global_h>
        <files>
          …
        </files>
      </component>
```

Interfaces

Software components typically provide interfaces to other parts of the software. There are two different types of interfaces:

- **Component API** that allows to use the functionality of the software component itself.
- **Driver API** to interface with hardware or other software (called by the software component

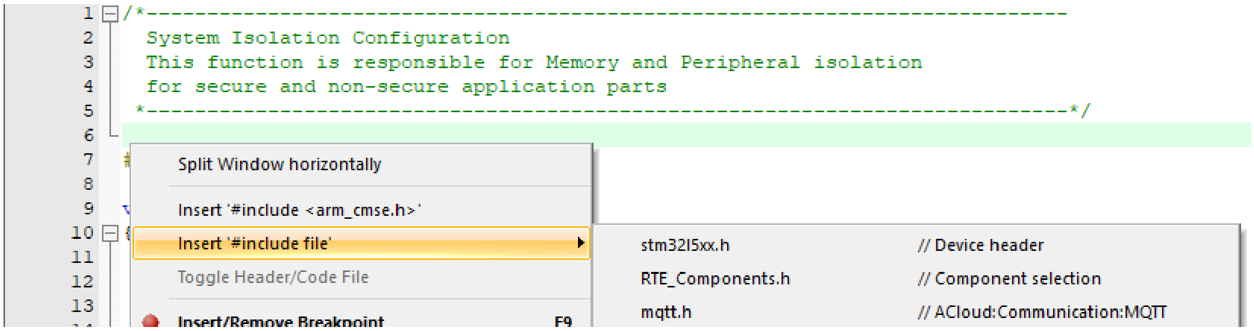


Component API

The API interface of a component can be easily exposed by including the related header file into the files list of that component as show in the *.PDSC snipped below.

```
     <component Cclass="ACloud" Cgroup="Communication" Csub="MQTT">
        <description>MQTT Lite connectivity</description>
        <files>
          <file category="sourceC" name="mqtt/source/mqtt.c"/>
          <file category="header"  name="mqtt/include/mqtt.h"/>
        </files>
      </component>
```

Some IDEs give customers easy access to these header files, for example via the context menu in the editor:



```
/*-----------------------------------------------------------------------
  System Isolation Configuration
  This function is responsible for Memory and Peripheral isolation
  for secure and non-secure application parts
 *-----------------------------------------------------------------------*/

#

v
{
```

Split Window horizontally
Insert '#include <arm_cmse.h>'
Insert '#include file'
Toggle Header/Code File
Insert/Remove Breakpoint F9
stm32l5xx.h // Device header
RTE_Components.h // Component selection
mqtt.h // ACloud:Communication:MQTT

Driver API

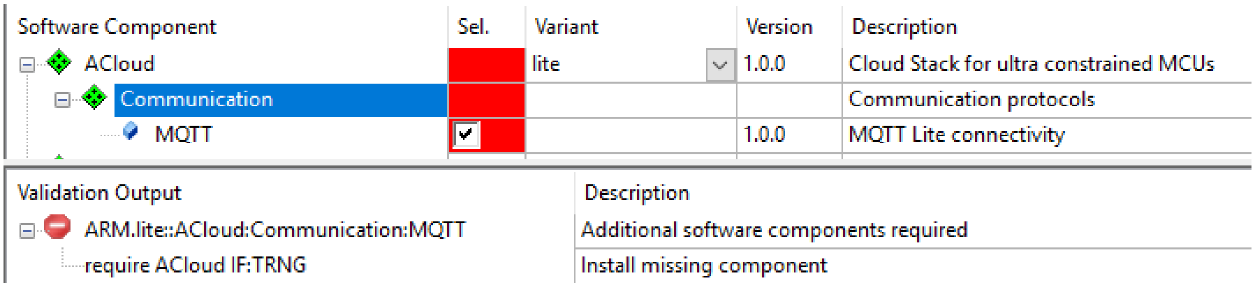
A common problem when providing a software component is that driver API headers evolve over time.

The **apis** element allows a software component to define an API to lower level software or hardware drivers. It shares header file and documentation of an API interface across multiple other software components to ensure consistency.

The example below defines a TRNG API for a true random number generator. In this example, a TRNG is a requirement for the MQTT software component. The related *.PDSC section in the ACloud pack is shown below:

```
<taxonomy>
  …
  <description Cclass="ACloud IF"> Low-Level Interfaces of ACloud component<</description>
</taxonomy>

<apis>
  <api Cclass="ACloud IF" Cgroup="TRNG" Capiversion="1.0.0">
    <description> True Random Number Generator</description>
    <files>
      <file category="header" name="include/trng.h"/>
    </files>
  </api>
</apis>

<conditions>
  <condition id="Requires_RNG">
    <require Cclass="ACloud IF" Cgroup="TRNG"/>
  </condition>
</conditions>



    <component Cclass="ACloud" Cgroup="Communication" Csub="MQTT" condition="Requires_RNG">
      <description>MQTT Lite connectivity</description>
      <files>
        <file category="sourceC" name="mqtt/source/mqtt.c"/>
        <file category="header"  name="mqtt/include/mqtt.h"/>
      </files>
    </component>
```

If the component MQTT is now selected, the RTE Management highlights that there are additional components required:



Software Component	Sel.	Variant	Version	Description
ACloud	🟥	lite	1.0.0	Cloud Stack for ultra constrained MCUs
Communication	🟥			Communication protocols
MQTT	☑		1.0.0	MQTT Lite connectivity

Validation Output	Description
ARM.lite::ACloud:Communication:MQTT	Additional software components required
require ACloud IF:TRNG	Install missing component

The TRNG might have several implementations available in different software packs. For example:

- TRNG – Emulation: software simulation provided in together with ACloud (in the same pack).
- TRNG – MCU: software driver for TRNG integrated in the microcontroller (in a device pack).
- TRNG – TFM: interface to a service of the Trusted Firmware for Cortex-M (in the TFM pack).

This is an example of the related definitions in the various pack *.PDSC files.

ACloud pack:

```
  <component Cclass="ACloud IF" Cgroup="TRNG" Csub="Emulation">
    <description>Software simulation of random number generator</description>
    <files>
        <file category="sourceC" name="source/trng_emulation.c"/>
    </files>
  </component>
```

Device support pack (here for the STM32F7 device family):

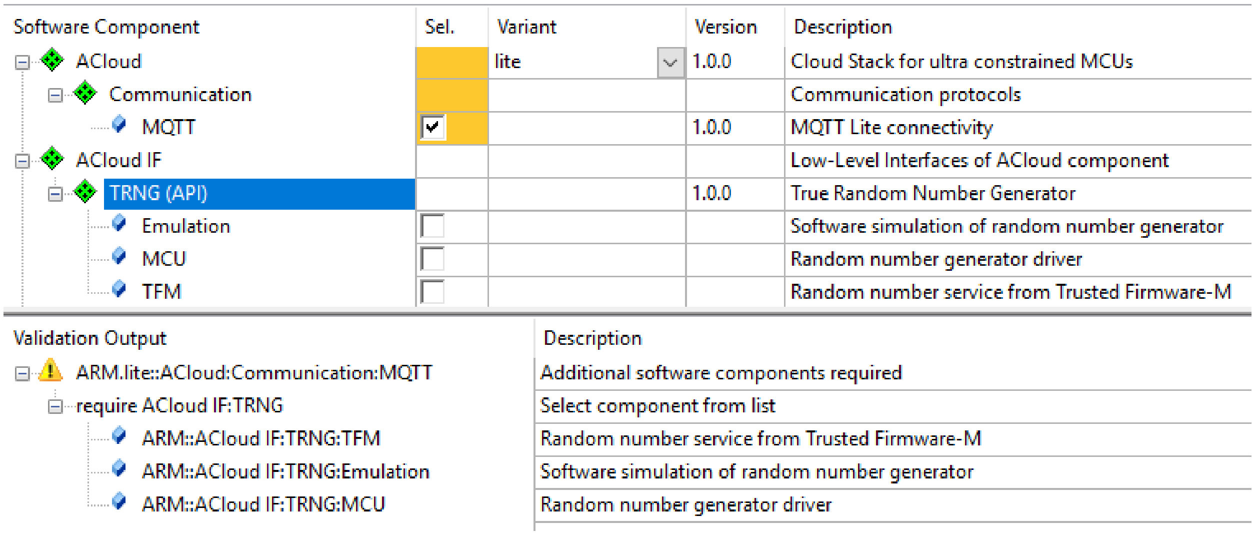
```
  <conditions>
    <condition id="STM32F7">
      <require Dvendor="STMicroelectronics:13" Dname="STM32F7*"/>
    </condition>
  </conditions>
…
  <component Cclass="ACloud IF" Cgroup="TRNG" Csub="MCU" condition="STM32F7" Capiversion="1.0.0">
    <description>Random number generator driver</description>
    <files>
      <file category="sourceC" name="TRGN_Driver.c"/>
    </files>
  </component>
```

TFM pack:

```
   <component Cclass="ACloud IF" Cgroup="TRNG" Csub="TFM" Capiversion="1.0.0" Cversion="1.2.0">
      <description>Random number service from Trusted Firmware-M</description>
      <files>
        <file category="sourceC" name="TRGN_TFM_Interface.c"/>
      </files>
    </component>
```

All these three packs refer to the same API header file `include/trng.h` published in the ACloud pack.

When all software packs are installed and an STM32F7 microcontroller is used, the RTE Management gives the user the choice of three different implementations for the TRNG. It shows the following:



Software Component	Sel.	Variant	Version	Description
ACloud		lite	1.0.0	Cloud Stack for ultra constrained MCUs
Communication				Communication protocols
MQTT	☑		1.0.0	MQTT Lite connectivity
ACloud IF				Low-Level Interfaces of ACloud component
TRNG (API)			1.0.0	True Random Number Generator
Emulation	□			Software simulation of random number generator
MCU	□			Random number generator driver
TFM	□			Random number service from Trusted Firmware-M

Validation Output	Description
ARM.lite::ACloud:Communication:MQTT	Additional software components required
require ACloud IF:TRNG	Select component from list
ARM::ACloud IF:TRNG:TFM	Random number service from Trusted Firmware-M
ARM::ACloud IF:TRNG:Emulation	Software simulation of random number generator
ARM::ACloud IF:TRNG:MCU	Random number generator driver

You may have noticed that the component **ACloud:IF:TRNG:MCU** has a **condition** that makes it dependent to a specific microcontroller series. This component is only available when the microcontroller used for the project matches.

You can have multiple device support packs that define a component **ACloud:IF:TRNG:MCU**. For the user this has the benefit that changing the microcontroller with a selected component **ACloud:IF:TRNG:MCU** automatically selects the matching driver for the random number generator.

Using API Versions

The Capiversion attribute defines the compatibility of the API header (in the element <apis>) with the implementation of the API driver (in the element <component>).

This component requires the API version 1.1.0 or higher:

```
<component Cclass="ACloud IF" Cgroup="TRNG" Csub="MCU" condition="STM32F7" Capiversion="1.1.0">
  <description>Random number generator driver</description>
  <files>
    <file category="sourceC" name="TRGN_Driver.c"/>
  </files>
</component>
```

This <api> definition is compatible with the component as it defines version 1.2.0:

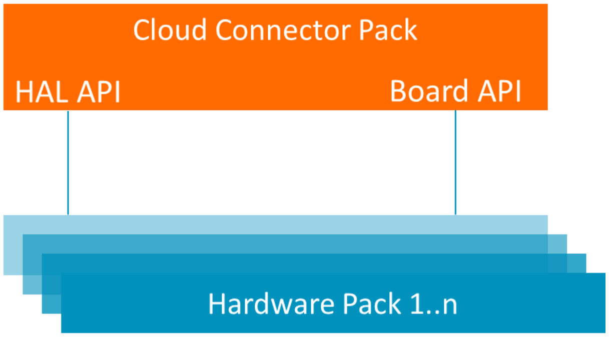
```
<apis>
  <api Cclass="ACloud IF" Cgroup="TRNG" Capiversion="1.2.0">
```

This <api> definition is not compatible, as it is version 1.0.0 and therefore to old for the component.

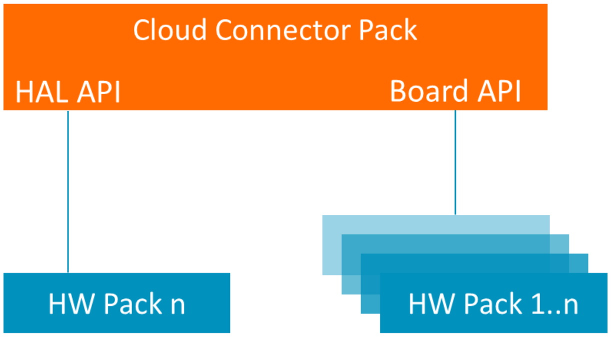
```
<apis>
  <api Cclass="ACloud IF" Cgroup="TRNG" Capiversion="1.0.0">
```

Deploy a Software Stack to Various Hardware Platforms

A software pack can define one or more APIs. It might also contain a couple of implementations of this API, but this is not mandatory. Usually, the actual implementations reside outside in other packs that relate to a certain device or board:

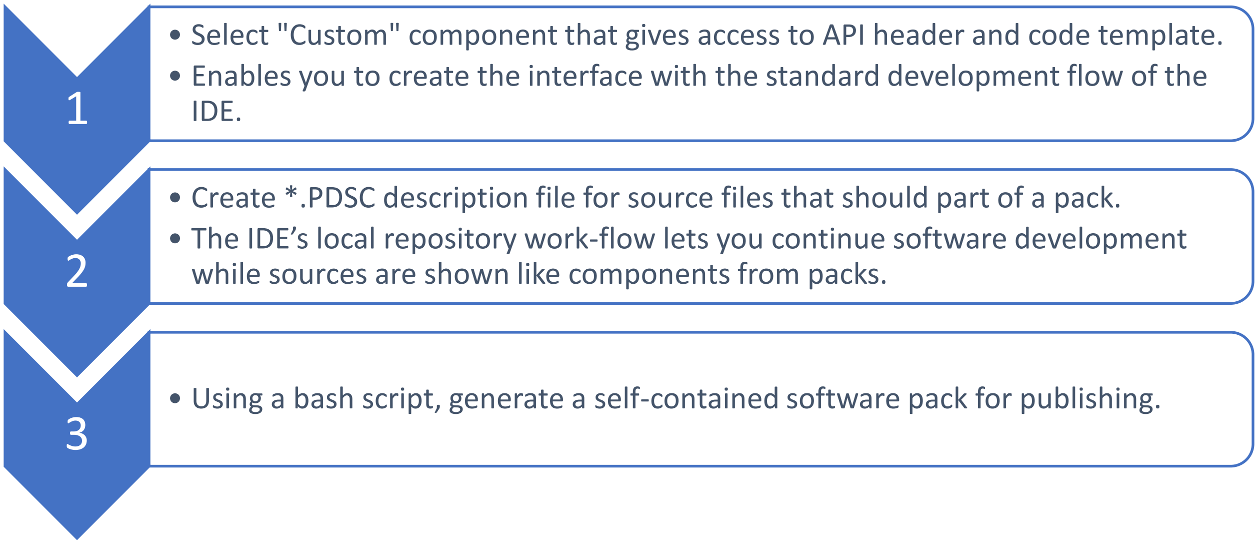


Depending on the relationship between API components, the actual implementations also might be delivered in different packs and/or in a different granularity:


Cloud Connector Pack
HAL API
Board API
HW Pack n
HW Pack 1..n

Implement a custom interface component

Developers implementing the API need a reference that helps to reduce the overall time for creating the component. Thus, the original API pack should contain a “custom component” that can be selected in the IDE and gives access to user code templates that provide a quick start to the developer. The overall flow to create the custom interface component is the following:



1

- Select "Custom" component that gives access to API header and code template.
- Enables you to create the interface with the standard development flow of the IDE.

2

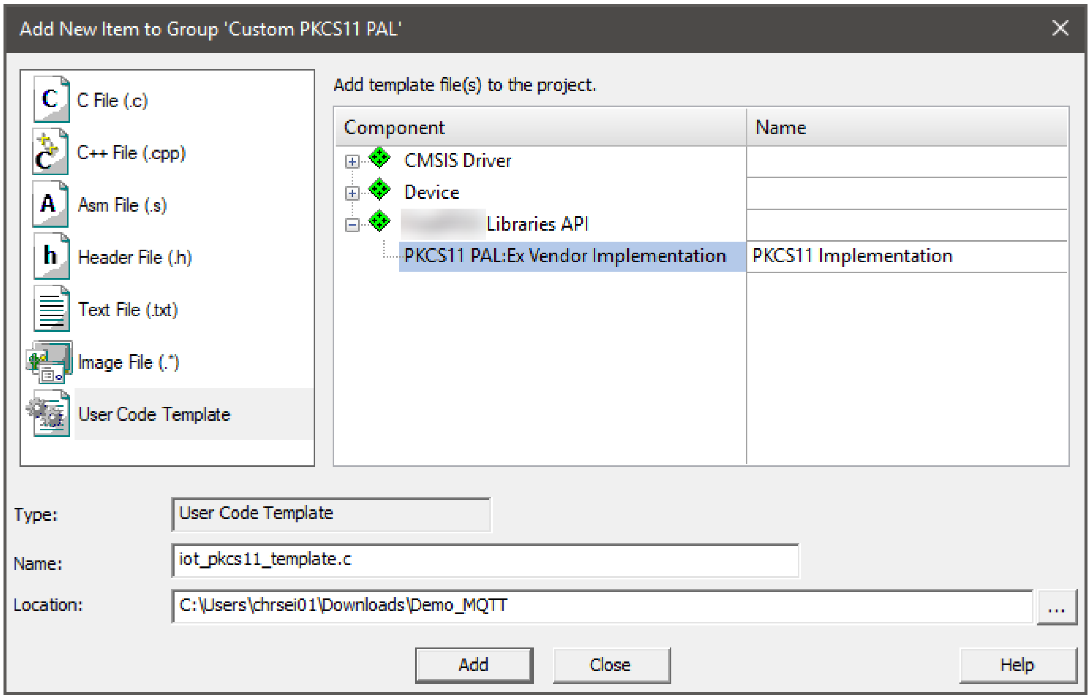
- Create *.PDSC description file for source files that should part of a pack.
- The IDE’s local repository work-flow lets you continue software development while sources are shown like components from packs.

3

- Using a bash script, generate a self-contained software pack for publishing.

Selecting the custom component

For the API, select the custom component (here: ”Ex Vendor Iplementation”) and add it to the project. In the IDE, add the related user code template to your project that is used as a starting point for the actual implementation:



Add New Item to Group 'Custom PKCS11 PAL'

Add template file(s) to the project.

Component	Name
CMSIS Driver	
Device	
[redacted] Libraries API	
PKCS11 PAL:Ex Vendor Implementation	PKCS11 Implementation

Type: User Code Template

Name: iot_pkcs11_template.c

Location: C:\Users\chrsei01\Downloads\Demo_MQTT

The template file is added to the project and you can start working on the implementation, using the standard development flow of the IDE. If the implementation is only used once in this project, you are done once you have finished the component development.

Creating a custom pack

Usually, such components are shared amongst developers – either within one company or with a wider audience. To create a pack for easy delivery of this component, use the local repositories flow of the IDE that enables you to continue developing the sources while they are shown as software components in the IDE.

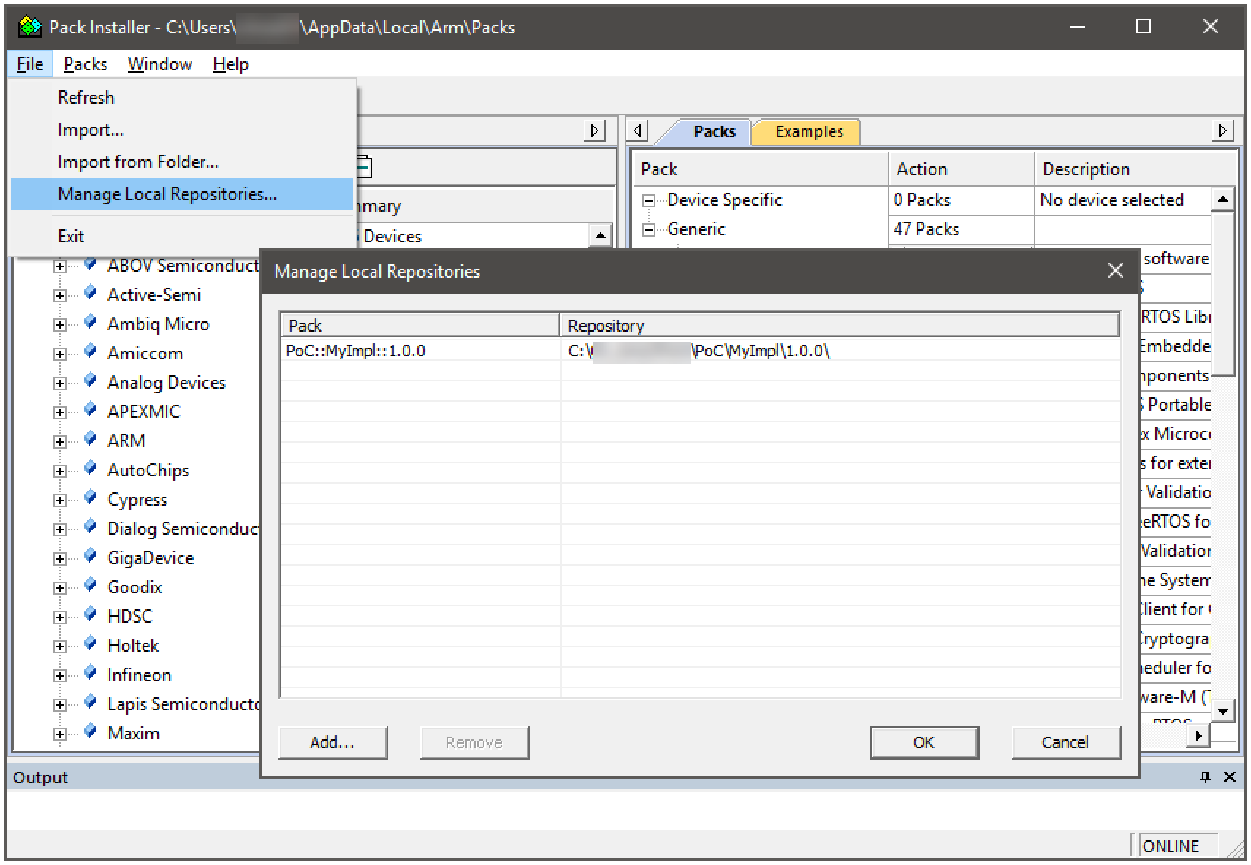
Start by creating a pack-like structure on your hard disk following the convention <vendor>/<pack_name>/<version>. A PDSC template file is available that should be copied here. Apart from mandatory sections like pack name, releases, and keywords, it contains an example component for custom API implementations:

```
  <components>
    <component Cvendor="Add vendor name here if not same as pack vendor" Cclass="Enter Cclass name of API" Cgroup="Add if available" Csub="Add if available" Capiversion="0.0.0" Cversion="0.0.0" condition="Add if required">
      <!-- Enter a short description below -->
      <description>Describe implementation details here</description>
      <files>
        <!-- Add source file(s) below -->
        <file category="sourceC" name="source/custom_implementation.c"/>
        <!-- Add configuration file (usually header files) below -->
        <file category="header"  name="config/custom_impl.h" attr="config"/>
        <!-- Add link to further information/documentation below -->
        <file category="doc"  name="Documentation/documentation.html"/>
      </files>
    </component>
  </components>
```

Copy the source/config/header files that you have created in your project to the folder structure and make the required changes to the PDSC file. Don’t forget to rename the PDSC file to <vendor>.<pack_name>.pdsc. Now, your directory is ready to be picked up by the IDE as a local repository.

Adding the pack as a local repository

The pack management utility of your IDE should have the ability to recognize an unzipped pack in a folder structure correctly. This helps with developing packs using source control flows like GitHub:



Pack	Action	Description
Device Specific	0 Packs	No device selected
Generic	47 Packs	

Pack	Repository
PoC::MyImpl::1.0.0	C:\[redacted]\PoC\MyImpl\1.0.0\

In this example, the <vendor> is “PoC”, the <pack_name> is “MyImpl” and the <version> number is 1.0.0.

Testing the component in the IDE

Next, go back to your IDE and select the newly added component instead of “Ex Vendor Implementation”. The IDE will now use this component in the project, and you can continue to develop it further.

Publishing the custom pack

Once finished, use the https://github.com/ARM-software/CMSIS_5/tree/develop/CMSIS/Pack/Bash/gen_pack.sh Shell script to create the zipped version of the custom pack. This can now be shared, for example by adding it to Arm’s indexing server available at https://developer.arm.com/embedded/cmsis/cmsis-packs.