Creating Your Own CMSIS Pack

By Colin Funnell

Introduction

CMSIS packs are at the heart of MDK and are a great way to include managed, standardised chunks of code and other resources into your project. The uses range from low-level silicon start-up code in Device Family Packs, through Board Support Packages to high-level protocols and services. There is no reason why you can't benefit from adopting similar structures for your own needs. In fact, Open-CMSIS-Pack encourages this to promote software reuse.

This article is intended as a very brief guide to the basic steps to create your own CMSIS pack. There is much more than what is shown here but it should be enough to get over the hurdle creating your first one. Everything after that is refinement.

It is strongly recommended to become familiar with CMSIS packs as a user before following the steps to making your own. See also TechTip: Background Primer to CMSIS packs and Practical Considerations when making your own CMSIS pack.

The CMSIS landscape is made more confusing by its rapid spread, development detail and automation within toolsets.

As we saw in the Background introduction the CMSIS-Toolbox is a build system with utilities that support downloading and management of project packs.

This guide should help cut-through the unwanted detail at this stage.

What is needed to create a CMSIS pack?

A set of tools are made available in the gen-pack repository of the Open-CMSIS-Pack project. Like any other github repository, projects can be cloned using command-line tools or the 'Git GUI' tool. See: https://github.com/Open-CMSIS-Pack/gen-pack. Since version strings become important later on, downloading a tagged-release is preferred.

Along with the source itself, the readme.md file is essential to find out which prerequisites are needed to run gen-pack for various platforms. The gen-pack scripts are basically Linux command shell scripts. Windows users may have a little trouble adjusting to this. Sometimes it is the setup of the platform with environmental parameters and file permissions which poses more of a hurdle to the first-time user.

Forming the pack description file (.pdsc)

The pack description file is the key to the whole CMSIS pack system. It is used by pack generation tools to verify that the description makes sense as well as end IDEs to correctly make use of the provided code in an appropriate way.

It's almost impossible to cover all the possible pack options in such a short space, but a good place to start is to examine the .pdsc file from within packs you are already familiar with. You'll know how they interacted with you when bringing them into a project. Browsing in your file system under the location stored in CMSIS_PACK_ROOT will show you the uncompressed packs – exposing their .pdsc files.

An alternative is to look at an example from Open-CMSIS-Pack:

https://github.com/Open-CMSIS-Pack/SW-Pack-HandsOn/blob/main/ACME.ACME_Middleware.pdsc

It can be broken down into a few main areas:

```
<package schemaVersion="1.2" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" xs:noNamespaceSchemaLocation="PACK.xsd">
  <name>CMSIS</name>
  <description>Cortex Microcontroller Software Interface Standard (CMSIS) CORE, DSP, RTOS, Driver</description>
  <vendor>ARM</vendor>
  <!-- <license>CMSIS\CMSIS_END_USER_LICENCE_AGREEMENT.rtf</license> -->
  <url>http://www.keil.com/pack/</url>

  <releases>
    <release version="4.0.0">
      - CMSIS-Driver   2.00  Preliminary (incompatible update)
      - CMSIS-Pack     1.10  Preliminary
      - CMSIS-DSP      1.4.2 (see revision history for details)
      - CMSIS-Core     3.30  (see revision history for details)
      - CMSIS-RTOS RTX 4.74  (see revision history for details)
      - CMSIS-RTOS API 1.02  (unchanged)
      - CMSIS-SVD      1.10  (unchanged)
    </release>
  </releases>
```

Supplier and release information

```
  <conditions>
    <condition id="CMSIS Core">
      <description>CMSIS CORE processor and device specific Startup files</description>
      <require condition="Cortex-M Device"/>
      <require Cclass="Device" Cgroup="Startup"/>
    </condition>

    <condition id="CM0_LE_GCC">
      <description>Cortex-M0 or Cortex-M0+ or SC000 processor based device in little endian mode for the GCC Compiler</description>
      <accept Dcore="Cortex-M0"/>
      <accept Dcore="Cortex-M0+"/>
      <accept Dcore="SC000"/>
      <require Dendian="Little-endian"/>
      <require Tcompiler="GCC"/>
    </condition>
  </conditions>
```

Dependency on other component

Dependency on core, endianness and toolchain

```
  <components>
    <!-- CMSIS-DSP component -->
    <component Cclass="CMSIS" Cgroup="DSP" Cversion="1.4.2" condition="CMSIS DSP">
      <description>CMSIS-DSP Library for Cortex-M, SC000, and SC300</description>
      <files>
        <!-- CPU independent -->
        <file category="doc" name="CMSIS\Documentation\DSP\html\index.html"/>
        <!-- <file category="header" name="CMSIS\Include\arm_common_tables.h"/> -->
        <file category="header" name="CMSIS\Include\arm_math.h"/>
        <!-- CPU and Compiler dependent -->
        <!-- ARMCC -->
        <file category="library" condition="CM0_LE_ARMCC"   name="CMSIS\Lib\ARM\arm_cortexM0l_math.lib" src="CMSIS\DSP\Source\ARM"/>
        <file category="library" condition="CM0_BE_ARMCC"   name="CMSIS\Lib\ARM\arm_c
        <file category="library" condition="CM3_LE_ARMCC"   name="CMSIS\Lib\ARM\arm_c
        <file category="library" condition="CM3_BE_ARMCC"   name="CMSIS\Lib\ARM\arm_cortexM3b_math.lib" src="CMSIS\DSP\Source\ARM"/>
        <file category="library" condition="CM4_LE_ARMCC"   name="CMSIS\Lib\ARM\arm_cortexM4l_math.lib" src="CMSIS\DSP\Source\ARM"/>
        <file category="library" condition="CM4_BE_ARMCC"   name="CMSIS\Lib\ARM\arm_cortexM4b_math.lib" src="CMSIS\DSP\Source\ARM"/>
        <file category="library" condition="CM4F_LE_ARMCC" name="CMSIS\Lib\ARM\arm_cortexM4lf_math.lib" src="CMSIS\DSP\Source\ARM"/>
        <file category="library" condition="CM4F_BE_ARMCC" name="CMSIS\Lib\ARM\arm_cortexM4bf_math.lib" src="CMSIS\DSP\Source\ARM"/>
        <!-- GCC -->
        <file category="library" condition="CM0_LE_GCC"    name="CMSIS\Lib\GCC\libarm_cortexM0l_math.a" src="CMSIS\DSP\Source\GCC"/>
        <file category="library" condition="CM3_LE_GCC"    name="CMSIS\Lib\GCC\libar
        <file category="library" condition="CM4_LE_GCC"    name="CMSIS\Lib\GCC\libar
        <file category="library" condition="CM4F_LE_GCC"   name="CMSIS\Lib\GCC\libarm_cortexM4lf_math.a" src="CMSIS\DSP\Source\GCC"/>
        <!-- G++ -->
        <file category="library" condition="CM0_LE_G++"    name="CMSIS\Lib\G++\libarm_cortexM0l_math.a" src="CMSIS\DSP\Source\G++"/>
        <file category="library" condition="CM3_LE_G++"    name="CMSIS\Lib\G++\libar
        <file category="library" condition="CM4_LE_G++"    name="CMSIS\Lib\G++\libar
        <file category="library" condition="CM4F_LE_G++"   name="CMSIS\Lib\G++\libarm_cortexM4lf_math.a" src="CMSIS\DSP\Source\G++"/>
      </files>
    </component>
  </components>
</package>
```

Common files

Files for ARM compiler

Files for GCC compiler

Files for G++ compiler

Even this can be whittled down further to just the supplier, release information and the bare minimum <component> details within a <components> block.

More information can be found in the Open-CMSIS-Pack description files: https://open-cmsis-pack.github.io/Open-CMSIS-Pack-Spec/main/html/packFormat.html. Here the XML elements are described in further detail. That guide is going to be essential for any complex pack production.

A simple <components> section could comprise of:

```
<component   Cclass="Utility" Cgroup="ThingReuse"
                    Csub="MyUsefulCode"
                    Cversion="0.1.0">
  <description>Brief descr. shown in RTE manager</description>
  <files>
       <file category="doc"    name="./Docs/MyHelpDoc.pdf"/>

       <file category="header" name="./Source/myProj_config.h" attr="config" version="0.1.0"/>

       <file category="source" name="./Source/myProj.c"/>
       <file category="header" name="./Source/myProj.h"/>
  </files>
</component>
```

Here the component attributes help describe where this new piece of code will sit in the tree of all known CMSIS packs and sub-categories. Cclass groups are defined for everyone to follow. This avoids people inventing new places and things getting lost in the CMSIS Pack ecosystem. Note that the component can have its own semantic version number separate from the pack version. More complex packs can have conditions defined which can check for the presence of certain components in the system as well as checking against version.

Finally, we get to refer to the source code files. The most basic form just lists each file and puts them into a source or header category. You'll see one file above with a "config" attribute. The IDE will see this and recognise that it is a user configurable file, and as such creates a Read/Write copy to be used in the project. Typically, pack files are Read-Only so they are kept at a set version and state. This file itself has a version number which allows the IDE to highlight the file if a configuration file needs developer attention due to a pack update during a project's lifetime.

One of the most useful things you can include with your code is some appropriate documentation that is easy to reach. By using the "doc" category on a file, an IDE can associate it with help for this component and will normally provide a link so the developer can jump straight to it.

There are plenty of more things you can embellish your pack with. Project Examples and code templates are all useful things which aid people in picking up and using your pack with minimal effort. The .pdsc XML schema has elements for this and many more things besides. It all helps towards code reuse by improving the developer experience beyond shifting big chunks of code.

Checking the pack description

Creating the .pdsc file the trickiest part of the work – correctly defining it to convey your intention of how the code should be handled. It is all too easy to get into a muddle when so many options are available, and not making an XML typo. A Pack checking tool exists to help check the validity of the description before it is used.

The "packchk" tool used to be distributed along with CMSIS Core (<Ver 6.0.0) but now has a home with CMSIS Toolbox. As a quick and dirty check, the old Core version of packchk can still be used but the Toolbox version will be the way forward.

For those using MDK-6, this is straight-forward since MDK revolves around the use of sub-tools like CMSIS Toolbox. These are brought into the toolset automatically. By opening an MDK6 project, a terminal window can be used to run it. Either way, the command is:

```
packchk  <filename.pdsc>
```

-where the formatting of the filename is important. It must be <Vendor>.<Package name>.pdsc. Vendor and Package name must match the fields within the .pdsc file.

Packchk is very useful to weed out any fundamental errors before proceeding too far. Once the .pdsc file is usable, a Pack Manager (as in Keil µVision) or command cpackget (CMSIS Toolbox command) can be used to bring your pack into your IDE before the final pack is created. This is a useful way to check that the pack is 'seen' as it is meant to be before being deployed into the wild.

Modifying the gen_pack script

Inside the gen-pack source repository are two different, but similarly named, gen-pack and gen_pack files. Gen_pack is inside the template folder and is meant to be modified for your needs and put with your .pdsc file. This can be copied out to your project work area to then customise.

The main part needing changing is:

```
REQUIRED_GEN_PACK_LIB="<pin lib version here>"
```

-where the lib version has to match that of GEN_PACK_LIB_VERSION in gen<-hyphen->pack.

There are many more things which can be customised, but the above is the minimum.

Final output

Once the pack source (code and description) is in a useable state, we can proceed to creating the pack itself. This is done by invoking the gen_pack.sh script with an environmental variable showing where to find the other parts of gen-pack you downloaded. A simple launcher script can set this and anything else you need before invocation.

```
export GEN_PACK_LIB_PATH="../lib0_11_1"
export LANG=$(locale -s -U)
source ./gen_pack.sh
```

gen_pack will show you any warnings and errors along the way, in a similar way to packchk. If it can, it will produce the .pack file in the \output sub-folder. Pack files will have the familiar <Vendor><Component Name><Pack version>.pack filename format.

You are now the proud owner of your very own CMSIS pack!

In summary

The tools to create your own CMSIS packs are out there – and have been for a while. Where they reside and what banner they come under can lead to confusion. There are so many similar related tool names and projects that it is easy to wander down the wrong path.

Ultimately, they are now made more available than ever before, and in a consistent manner. As with many hosted software projects, it can require some work by the end developer to get started and obtain what they need out of it. Once you have created your first pack, everything is in place to start creating bigger and better ones.

Hopefully this article points out some of the common pitfalls to get you going more quickly to producing your own set of reusable and plug-able code.

Useful links

https://www.open-cmsis-pack.org/

https://github.com/Open-CMSIS-Pack/gen-pack

https://open-cmsis-pack.github.io/Open-CMSIS-Pack-Spec/main/html/index.html

https://github.com/Open-CMSIS-Pack/cmsis-toolbox/blob/main/docs/README.md

https://github.com/Open-CMSIS-Pack/SW-Pack-HandsOn

Hitex Knowledge Base

From the Insiders' Guide to CMSIS Packs series:

Background to CMSIS

Practical Considerations when making your own CMSIS pack

Further Information

For more information visit our website: www.hitex.co.uk or get in touch: info@hitex.co.uk. You can also connect with us: LinkedIn