hitex

Insiders' Guide to CMSIS Packs

Part 3: Considerations for CMSIS workflow

Practical Considerations When Making 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.

Jumping straight into CMSIS-Toolbox for the tools is just part of an approach. This guide aims to get some of the basics and avoid a few pitfalls trying to manage your own CMSIS pack workflow.

What is needed to create a CMSIS pack?

In short, the tools are provided from the CMSIS-Toolbox repository for multiple platforms. If your machine is capable of running the scripts, along with a carefully crafted description file, it can produce the pack for you.

It isn't going to be *that* simple but I recommend reading *Creating your own CMSIS pack* TechTip for the basic details.

Managing the creation

There are lots of aspects which can form a CMSIS pack. Other than core code, two of the most useful are documentation and examples. They go hand-in-hand with each other and CMSIS packs have XML tags to express these in the IDE. Excessive documentation is hard to follow where a working example shows the system in action. Conversely, examples only get you so far without good documentation describing the overall system and fundamental

concepts. They should be early targets to include in your pack instead of providing a set of isolated source files on their own. (See file category="doc" and <examples> in the .pdsc description.)

A useful approach to managing the creation of your own packs is to treat it as a mini-project with its own process and development flow. Just by writing things down at key stages helps focus what your pack will be. A full-on engineering process may not be suitable at times but the ethos can still be useful. Some useful stages may be:

Wishlist

As a developer you already have a view on what you want your pack to become – what it represents but also what it could be in the future. As soon as others learn of something that could help them out, they'll have a view too. Placing all the ideas into a pot and seeing how things cluster together will give an insight into the shape of things.

Roadmap

Not all wishes come true, or at least can't all be done at once. The wish list can be whittled down into stages of development where each smaller stage can be done, and done well, before moving onto the next stage. A roadmap helps provide direction, which others can see. A different direction could be taken but with a visible plan in advance it should be understandable by all that development can't drop what it's doing and do something completely different.

CMSIS packs have a semantic 3-digit version number scheme. Once you have created a few packs, creating a new version isn't that difficult. Version numbers are there to be used and really helps with incremental step development.

Avoid taking on too much in one go. Splitting a development into smaller roadmap stages can help tackle one problem area at a time. Each stage should be to produce a good pack, even if it doesn't yet include everything wanted. Tackling too many ideas simultaneously spreads your work too thinly.

Development

The surrounding pieces of management make the actual development part much easier. By keeping matters focussed with narrow and clear goals, it becomes a turnkey set of coding. As with any development, sprawling requirements are your enemy here.

As a project which needs to be supported and developed over, hopefully, a long life-time then now is an ideal opportunity to instil good practices such as Test Driven Development (See Useful links for details) or any other systematic approaches. It is a rare opportunity to start something completely afresh, so it shouldn't be squandered.

By keeping to development/release cycle approach, all the supporting work can keep in-step too. Updating examples, documentation, templates are part of the same cycle instead of being pushed aside in the pursuit of getting the most functional code as fast as possible.

Checklists

As a mini-project with limited visibility, it is all too tempting to push a pack out the door so you can crack-on and make use of it in a project. Being hasty will always come back to haunt you with mistake fixing being put into the next immediate roadmap section – more to do before you start the new features proper.

As mundane as some checklist items can be, it is always good practice to confirm the obvious.

- Update any internal version identifiers.
- Ensure Test Driven Development tests pass.
- New features are in documentation.
- Documentation is reviewed/released.
- Clean the code.
- Find and destroy any //TODO markers. (Finish off / Feedback into roadmap / Delete)

Changelogs

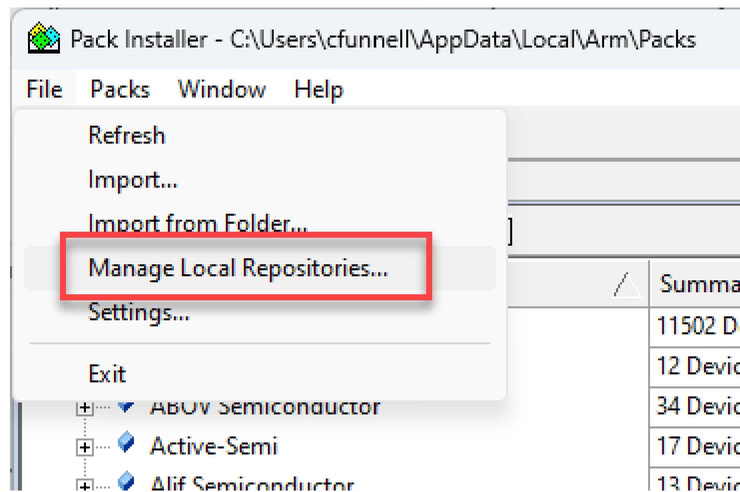
Changelogs are helpful to show what has really happened during the development. It should be a handy form of self-checking against the roadmap for this version. It can be a little too easy to wander off down a path of improving things but still miss the initial aims. All the improvements should be included in a changelog. It's an advertisement of your efforts.

All of the above can be informal for your own use, which may seem a little unnecessary, but it helps focus the development work and provides people with an instant snapshot of where things are and where they are going.

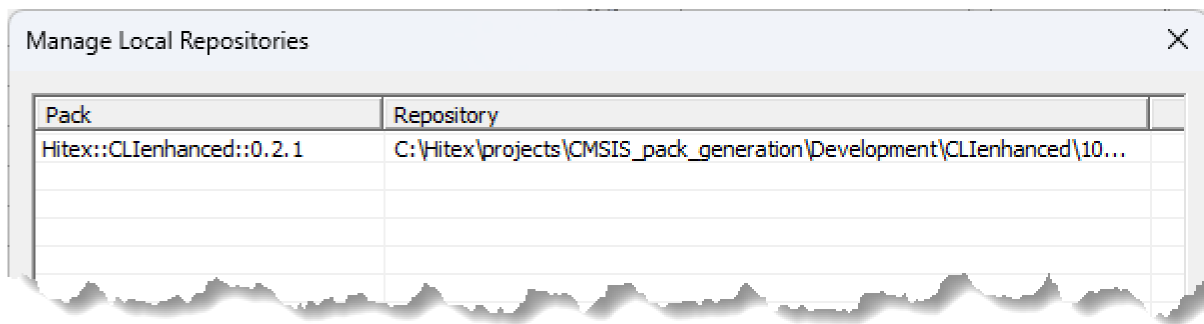
Using temporary packs when making an example project within a pack

For a code example to be representative in the pack system it must make use of the CMSIS pack in question and at the same time be inside the same pack. This chicken-and-egg situation can be handled by using the contents of a pack without formally creating and installing it.

To use this, you do have to go through the process of creating a Pack Description file (.pdsc) and verifying it is valid. Once it is, you can point µVision at it by opening the Pack Installer and Manage Local Repositories…



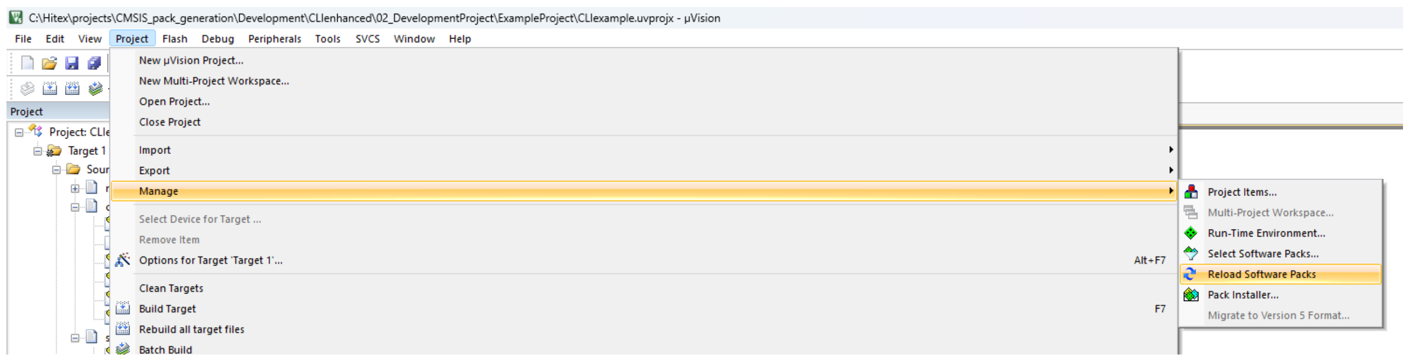
By using the file browser, select the .pdsc file and through it µVision understands where the source files are and which version of the pack it is.



Once selected the system should recognise a new software pack and refresh accordingly.

You will now be able to see your pack appear in the Run-Time Environment selection like any other pack – and before the final pack_gen is run. This saves fiddling around with pack generation before it is really ready and provides a means to "dummy-run" the pack use. By creating your example project that uses your pack as intended, it helps iron-out user issues where your design doesn't quite meet in the middle between the two worlds.

Note that if the code is a work in progress you will need to retrigger the adoption of any new code by using asking µVision to Reload Software Packs.



When your pack is complete and generated you can install that pack simply by double-clicking on it – like any other external pack. Don't forget to remove your local repository version of the same pack to avoid confusion!

Side benefits

One of my favourite outcomes of creating a CMSIS pack is that it can inadvertently cause better code to be written. Let me explain.

Whenever code is expected to be used or seen by other people, developers are automatically on their best behaviour. All the corners that are cut because they weren't applicable to their needs now gets attention before someone else sees it. As soon as developers and management realise there will be ownership of the pack then an appreciable amount of time can be allocated and spent by properly defining and implementing the work. Re-using code goes from copy-paste from another project to a more polished piece of modular work.

Managing the pack production into a published item helps Engineering in general. Checks, reviews and documentation become part of the production process instead of being whatever state developer X left their code in when they stopped working on it.

For anything that needs to be maintained over a long lifetime, Test Driven Development (TDD) can become part of the design flow.

This helps the roadmap approach where there could be long gaps between developments. The pack can be proven to be working before any new changes are made and shown the changes don't break anything.

Focussed efforts from a roadmap can sit nicely with a V-model or Agile based workflow. Even with a team of one, a good production process helps keep people on the "straight and narrow".

Having a roadmap view of the world stops the casual approach of throwing extra pieces of code in to help solve immediate problems to get people out of a hole.

In summary

There are few negatives to creating CMSIS packs for reusable code within your organisation.

There are overheads, yes, but these are required for any good quality piece of reusable and robust code. If you're using CMSIS packs already in your development system you can now piggy-back on the technology to make your work available in the same way. Large organisations benefit from the consistency. Small organisations benefit from the improved quality that can be obtained.

Useful links

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

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

TechTips:

Hitex Knowledge Base

Unity Software Testing Framework

Switching Mindset to Test Driven Development

From the Insiders' Guide to CMSIS Packs series:

Creating your own CMSIS pack

Background to CMSIS

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