Cetmodules 2: overview, highlights, and implications for UPS to Spack migration

Chris Green, FNAL

LArSoft Coordination Meeting, 2021-11-02

Recap

Quick recap:

- Developing a long term replacement for our current UPS-based ecosystem with wide applicability across HEP.
- Spack / cetmodules / SpackDev / BuildCache vs UPS & ssibuildshims / cetbuildtools / MRB / SciSoft.
- LArSoft updated to use cetbuildtools 8 (see presentations from 2021-09-21).

Cetmodules 2 *vs* Cetbuildtools 8

- *"What's the difference between building with Cetbuildtool 8 and building for Cetmodules 2?"*
  - Short answer: **"Not very much at all, really."**
  - Long answer: **"Pretty much everything, really."**

Let me explain. . .

“Not very much at all, really”

- From *Using cetbuildtools 8 and mrb 5*:
  “cetbuildtools 8 is a wrapper around cetmodules 2 with increased compatibility and fewer deprecation warnings.”
- If all one did was change “cetbuildtools” to “cetmodules” in “`CMakeLists.txt`” and “`ups/product_deps`”—with corresponding version changes where appropriate—your package should still build, install, and be usable by dependents using either Cetbuildtools 8 or Cetmodules 2 without alteration . . .
- . . . albeit with (probably many) more deprecation warnings than before.

“Pretty much everything, really”

You now have the power and flexibility to:

- improve build times for your own and dependent packages, and avoid unnecessary transitive dependencies
- reduce library size and memory footprint
- improve code correctness and safety by eliminating ODR violations caused by use of plugin services and tools
- eliminate the need to use `include_directories()` or `target_include_directories()` for direct or transitive dependencies

“Pretty much everything, really”

- eliminate the need to use `find_package()`, or `find_ups_product()` for transitive-only dependencies
- eliminate the need to use `find_library()`, `find_file()`, or `find_program()` for non-CMake external dependencies.
- deal trivially with header-only dependencies
- eliminate the need to use Cetbuildtools, or Cetmodules in dependent packages built with CMake
- eliminate the need to generate and maintain `product_deps` files

“But couldn’t I do all this without using Cetbuildtools *or* Cetmodules?”

Sure, but that would involve:

- ***either***:
  - writing a whole lot of CMake infrastructure ***or***
  - hand-rolling CMake config files for every package, and keeping track of transitive dependencies manually
- writing a *lot* more CMake code in each package
- doing a lot of work that’s already been done for you

“OK, *fine*, so what’s the value added by Cetmodules?”

- Auto-generation of CMake config files, including:
  - `find_dependency()` calls for transitive dependencies
  - useful variable definitions, including Cetbuildtools compatibility (`<project>_OLD_STYLE_CONFIG_VARS`)
  - `CMAKE_MODULE_PATH` additions to find provided CMake modules
  - Scoped target definitions (*e.g.* `dk2nu::Tree`, `art_plugin_support::ToolMaker`)
- All the accounting required to collect the information needed to enable auto-generation.

“OK, *fine*, so what’s the value added by Cetmodules?”

- Mechanisms to enable the developer to provide the information:
  - overrides for CMake functions: `include()`, `find_package()`
  - functions combining multiple CMake features: `cet_make_library()`, `cet_test()`, `basic_plugin()`...
- A path from UPS-dependent building and packaging to an agnostic system suitable for (*e.g.*) Spack.

New concepts: dependency types

Direct dependencies shared library **X1** from package **X** needs shared library **Y1** from package **Y**.

Indirect transitive dependencies library **W1** from package **W** needs library **X1**, but needs to know where **Y1** is so **X1**'s load-time dependencies can be satisfied.

Direct transitive dependencies library **W1** from package **W** uses `FancyHeader.h` from **X**, which defines an inlined function requiring symbols defined in library **Z1**.

New concepts: targets *vs* variables

- Variables: one name -> one value

  `ART_FRAMEWORK_SERVICES_OPTIONAL_RANDOMNUMBERGENERATOR_SERVICE ->`
  `.../art_Framework_Services_Optional_RandomNumberGenerator_service.so`

- Targets: one name -> many properties:

  `art::Framework_Services_Optional_RandomNumberGeneratorService ->`

  - `SHARED`
  - `IMPORTED`
  - `INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"`
  - `INTERFACE_LINK_LIBRARIES`
    `"art::Framework_Services_Registry;art::Utilities..."`
  - `IMPORTED_LOCATION_RELWITHDEBINFO`
    `".../libart_Framework_Services_Optional_RandomNumberGenerator.so"`
  - `...`

New concepts: library types (familiar)

- `STATIC` -> `libXXX.a`
  - concrete file
  - code defining required symbols extracted from `.a` and included in dependent library or executable
- `SHARED` -> `libXXX.so`
  - concrete file
  - code defining required symbols loaded from `.so` at runtime for use by dependent library or executable

New concepts: library types (new)

- `INTERFACE` libraries
  - Convey header-propagated transitive dependencies
  - No library!

```
add_library(art_plugin_support::toolMaker INTERFACE IMPORTED)

set_target_properties(art_plugin_support::toolMaker PROPERTIES
  INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
  INTERFACE_LINK_LIBRARIES "art::Utilities;canvas::canvas..."
  INTERFACE_SOURCES "${_IMPORT_PREFIX}/include/art/Utilities/make_tool.h"
  )
```

New concepts: library types (new)

- `MODULE` -> `libXXX.so`, `XXX.so`
  - concrete file
  - **cannot be linked to** – usable only via `dlopen()`
  - ideal for plugin registration code
- `OBJECT` -> `mypkg::XXX_objects` -> `ethel.o`, `bill.o`, `charlie.o`, `anthea.o`
  - virtual, CMake-only concept
  - list of object files: compile once, use in both `SHARED` and `STATIC` libraries.

New concepts: plugin implementation and registration libraries

- Currently, registration code and implementation code are always in the same library.
- Problematic for plugins with user-visible interface: services, tools.
  - C-linkage -> ODR violations
  - memory bloated with code that may never be needed
- Solution: put implementation, registration code in separate libraries

New concepts: plugin implementation and registration libraries

- `libart_Framework_Services_Optional_RandomNumberGenerator.so`
  - `SHARED`, linkable
  - `art::Framework_Services_Optional_RandomNumberGenerator_service`
  - `RandomNumberGenerator.cc`
- `libart_Framework_Services_Optional_RandomNumberGenerator_service.so`
  - `MODULE`, not linkable—no exported target for dependencies
  - `RandomNumberGenerator_service.cc`
- Linking, naming, export, import all handled by Cetmodules.

“So how do I use all this stuff, then?”

- Understand the code you’re writing
  - What is it providing?
  - What is required to use it?
  - What types of dependency does it have and/or confer on users?
- New keywords to `find_package()`:

  `PUBLIC` dependents will also need to know where this package is.

  `PRIVATE` Needed only by us: macros, build-only, test dependencies, data, . . .

“So how do I use all this stuff, then?”

- Use targets in library link lists, not library file names or variables
  - header locations, transitive dependencies, . . .
- New keywords for library link lists:

`INTERFACE` you don’t need this library, but users will.

`PUBLIC` needed to link this library, and also needed *directly* by dependents.

`PRIVATE` needed only to link this library

“So how do I use all this stuff, then?”

- New keywords for `cet_make_library()`
- Move away from `GLOB`bed lists
  - Prevents rebuilds, hysteresis