Ceedling: Powerful Plugins

At its core, Ceedling is a test build manager. It was originally built to orchestrate the symphony of compilers, linkers, and simulators and blend them with Unity and CMock. There are a multitude of other tools that can be useful for the embedded mad scientist. This is where Ceedling's plugins come in. It provides a variety of useful plugins for customizing your experience.

Using Plugins

Like most of the configuration options in Ceedling, the first place to visit is the project.yml file. Specifically, we are looking for the `:plugins:` subsection. Here, we can enable the plugin with the `:enabled:` list.

Most often, you will be using the plugins that are supplied with Ceedling to do the majority of your work. If installed in the default location, then you will have nothing further to do. You have the option to install the plugins to your own custom project directory or to add additional paths to your plugin search. In these cases, you will also want to configure the `:load_paths:` trait.

The general plugin configuration will end up looking something like this:

```
:plugins:
    :enabled:
        - beep
        - gcov
    :load_paths:
        - project/support/
```

Often, specific plugins will have additional configuration. We'll tell you about some of the useful features in this document. For more detail, each plugin contains a readme file which will detail all of its options.

Beep

Running an entire suite of tests can take time. When things start to take too long, we, as responsible developers (or as short-attention-span humans) switch to doing something else… checking our email, surfing the internet, sword-fighting with coworkers. This is just a good use of our time, until the tests have finished and we're still too busy doing the other thing to notice.

That's where the beep plugin comes in. It will happily emit a beep to let us know when our process has completed and it's ready for us to pay attention again. It has a couple of configuration options, allowing ceedling to beep on error and / or completion. For each, it allows us to configure the beep method. At the time of this writing, those options are `:bell` to use the ASCII bell character (and rely on your shell to make the noise) or `:speaker_test` to use the

linux speaker-test command. It wouldn't surprise us if this is expanded to support mp3 files and whatnot.

```
:plugins:
    :enabled:
        - beep
:tools:
    :beep_on_done: :bell
    :beep_on_error: :bell
```

Command Hooks

This plugin adds hooks at various points in the build and test process, enabling us to fully customize our toolchain. Do you need to transform the output of the linker file into a different format before simulating? Do you need to launch a simulator before each test is run? This plugin provides a large list of hooks which you can insert your own calls.

Each hook allows you to specify one or more executables to be called (or anything callable from your shell, really). Each executable can be called with an optional list of arguments. For example:

```
:plugins:
    :enabled:
        - command_hooks
:tools:
    :post_link_execute:
        :executable: objcopy.exe
        :args:
            - ${1}
            - output.srec
            - --strip-all
    :post_test:
        - :executable: echo
          :args: "${1} was glorious!"
        - :executable: echo
          :args:
              - It kinda made me cry a little.
              - You?
```

There are a few things to notice here. First, if you only have a single executable to call, as in our `:post_link_execute:` example, then there is no need to specify it as a list, as we did in

`:post_test:`. Second, notice the `${1}`? This is helpfully replaced by the target name of that step. In this case, it would be replaced by the executable name output by the linker step, or the test name in the second example.

The following hooks can be used:

```
    :pre_mock_generate
    :post_mock_generate
    :pre_runner_generate
    :post_runner_generate
    :pre_compile_execute
    :post_compile_execute
    :pre_link_execute
    :post_link_execute
    :pre_test_fixture_execute
    :pre_test_fixture_execute
    :pre_test
    :post_test
    :pre_release
    :post_release
    :pre_build
    :post_build
```

This plugin will allow you to add the majority of the extra tools you may want to include with Ceedling if you learn to work with it. In fact, most other plugins are really using similar internal hooks with very little additional functionality.

Code Coverage

Once a unit-testing initiative has gained momentum, one of the questions that is often raised is that of coverage. How much of the code has been tested? In some industries, this is an important metric and may even be a requirement. In others, it is merely a curiosity. In either case, Ceedling can support a couple of code coverage engines.

Note that both of these plugins are for interfacing to the respective coverage tool, and are not complete without them. In the case of the GNU Coverage plugin, the python library `gcovr` is used, as well as `gcov` itself. Both will need to be installed before enabling the plugin. In the case of the Bullseye Coverage plugin, the bullseye command line tools will need to be installed before enabling the plugin.

Once installed, the coverage engines work much like normal testing. If you want to run coverage on all your project, you can execute the following:

```
ceedling gcov:all utils:gcov
```

This chains together two commands which are separable. The `gcov:all` portion runs your tests, just like test:all did… but it runs it with all the necessary gcov stubbing in place. The latter, `utils:gcov`, generates html reports from the results. There are many options for the detail level and features enabled when reporting coverage. Check out the documentation with the plugin itself to learn more.

Module Generator

The module generator is a useful tool for creating boilerplate files when we're practicing Test Driven Development. With a single call, Ceedling can generate a source, header, and test file for a new module. If given a pattern, it can even create a series of submodules to support specific design patterns. Finally, it can just as easily remove related modules, avoiding the need to delete each individually.

Let's say, for example, that you want to create a single module named MadScience.

```
    ceedling module:create[MadScience]
```

It says we're speaking to the module plugin, and we want to create a module. The name of the module is between the brackets. It will keep this case, unless you have specified a different default. After you execute this command, it will create three files: `MadScience.c`, `MadScience.h`, and `TestMadScience.c`. Note that it is important that there are no spaces between the brackets. We know, it's annoying… but it's the rules.

You can also create an entire pattern of files, as we discuss in this class. To do that, just add a second argument with the pattern ID. That looks like this:

```
    ceedling module:create[SecretLair,mch]
```

In this example, it would create 9 files: 3 are headers, 3 source files, and 3 test files. These files would be named `SecretLairModel`, `SecretLairHardware`, and `SecretLairConductor`. Isn't that nice?

Like much of Ceedling, the module generator runs with a common set of defaults. By default, the module generator will place new source and header files in the src/ directory. It will place the corresponding test in the test/ directory.

This works well for the default project configuration. However, if we desired a different structure, we can tweak these defaults by updating our yaml. In fact, we can tweak a number of things in the yaml file related to this plugin:

```
:module_generator:
  :project_root: ./
  :source_root: source/
  :inc_root: includes/
  :test_root: tests/
  :includes:
      - defines.h
      - board.h
  :boilerplates: |
      /*****************************************
      * Awesome Project                        *
      * ---------------                        *
      * This is the sort of thing people put at the*
      * top of every file. Notice the '|' above?   *
      * That starts a multi-line text block. It    *
      * continues until we are no longer indenting *
      * as much as we did on the first line after  *
      * that pipe. Neat, huh?                  *
      *****************************************/
  :naming: :camel
```

Do you see the `:includes` section? That allows you to inject standard include files into every generated module. The boilerplates does the same for a boilerplate header (and honestly, it doesn't have to be just a header… it's any chunk of code you might like injected at the very top of your file).

The next option, `:naming`, enforces a file naming convention. This is particularly handy for keeping your file capitalization scheme consistent. It's options are:

- `:bumpy` - BumpyFileIsNamedLikeThis
- `:camel` - camelFilesAreNamedLikeSo
- `:snake` - snake_files_are_lower_cased_and_snake_together
- `:caps` - CAPS_FEELS_LIKE_YOU_ARE_SCREAMING

This plugin also supports `module:destroy[Filename]` to let you remove a series of files.

Subprojects

The subproject plugin is for building subprojects into static libraries, and then using those static libraries as part of a release build (along with more source code). These subprojects can still support their own tests, which can get run as part of the suite, just like everything else. It's an organization method and helps with large coherent libraries like an RTOS, file system, etc.

It has a number of configuration options. For example, you can add the library extension to the extensions list:

```
:extension:
   :subprojects: '.a'
```

You list each of your subprojects as a list under a new `:subprojects::path:` section of your project file. Each one gets a name, it's own defines, and their own source, include, and build paths:

```
:subprojects:
  :paths:
   - :name: libprojectA
     :source:
       - ./subprojectA/first/dir
       - ./subprojectA/second/dir
     :include:
       - ./subprojectA/include/dir
     :build_root: ./subprojectA/build/dir
     :defines:
       - DEFINE_JUST_FOR_THIS_FILE
       - AND_ANOTHER
   - :name: libprojectB
     :source:
       - ./subprojectB/only/dir
     :include:
       - ./subprojectB/first/include/dir
       - ./subprojectB/second/include/dir
     :build_root: ./subprojectB/build/dir
     :defines: [] #none for this one
```

You can also specify the compiler and linker, just as you would a release build… clearly with the purpose of making a static library instead:

```
tools:
  :subprojects_compiler:
    :executable: gcc
    :arguments:
      - -g
      - -I"$": COLLECTION_PATHS_SUBPROJECTS
      - -D$: COLLECTION_DEFINES_SUBPROJECTS
      - -c "${1}"
      - -o "${2}"
  :subprojects_linker:
    :executable: ar
    :arguments:
      - rcs
      - ${2}
      - ${1}
```

Once set up, if there are changes to any of a subproject's source files, the library will be rebuilt automatically before a release build is performed. Awesome!

Dependencies

The dependencies plugin is for when your project makes use of statically built libraries that don't have their own tests that you need to trigger. These libraries may be statically part of your existing project or may be built from source using an external build tool (like the classics, make and cmake). They may be fetched using a variety of methods, including as a zip file or via subversion or git.

Each dependency can be configured to track one or more static or dynamic libraries, as well as their related header files. When these artifacts don't exist already, Ceedling will automatically fetch them for you (if configured to do so), and build the artifacts before proceeding. These libraries can then be linked into your release code or even mocked during tests.

In addition to being able to customize build steps, the developer can specify environment variables which can be specified while executing these builds, providing a straightforward method of wrapping other build dependencies into a single process.

Now there's no need to create your own build scripts to get all those wacky libraries your project depends upon!

```
:dependencies:
  :libraries:
    - :name: SecretLibrary
      :source_path: libraries/secret/source
      :build_path: libraries/secret/build
      :artifact_path: libraries/secret/install
      :fetch:
        :method: :git
        :source: https://github.com/throwtheswitch/secretlib.git
      :environment: [ ]
      :build:
        - autoconf
        - make clean
        - make all
        - make install
      :artifacts:
        :static_libraries:
          - libsecret.a
        :includes:
          - include/**
```

Tests Reports

There are many formats that Ceedling can use to output test results. Many of them are used to fit into particular IDE's or Continuous Integration Systems. At the time of this printing, you can only have one report enabled at a time (though this is rumored to be changing soon).

```
:plugins:
   # enable only one of these:
   -  :json_tests_report
   -  :junit_tests_report
   -  :raw_tests_report
   -  :stdout_gtestlike_tests_report
   -  :stdout_ide_tests_report
   -  :stdout_pretty_tests_report
   -  :teamcity_tests_report
   -  :xml_tests_report
```

Warnings Report

This is useful for searching your output and collecting all the compiler and linker warnings into a single place for you to reference later. It will put all of them in a `warnings.log` file in your artifacts directory.

Compile Commands

Numerous IDE's now support the `compile_commands.json` file for fetching information from the compiler to autocomplete function and methods. This is the format made popular by the clang project. This plugin may be enabled, which will make Ceedling build a compile_commands.json file in the build/test directory whenever tests are built. You can point your IDE at this file and it will then know where all your functions and types are coming from!