Ceedling: Managing Release Code

Ceedling knows about all of your source code. It knows about your development toolchain. Often, it's tied into your IDE and your Continuous Integration Server. After Ceedling has unit tested all your modules, what's more natural than having it build the official release?

Let's talk about some of the things that make Ceedling well suited for this task.

Here, being driven primarily from a command prompt is an advantage. When attached to a CI server, it's easy to have Ceedling make the "official" release as soon as all of the tests pass. Developers can work on their own machines, knowing that they're building in the same manner as the master server. All the headaches of differences between an IDE build and a CI build go away.

Ceedling handles release builds in a similar manner to the way it handles test builds… but of course there's a single build instead of a build per source module. It provides the same dependency tracking, file discovery, and command hooks as a test does.

The rest of this document will explore the configuration options that allow you to customize your release build experience.

Getting Started:

Configuring your project for a release build starts in the project file. In fact, it starts in the project section of the project file. Specifically, we need to enable the release build option, which is off by default.

```
:project:
  :release_build: TRUE
```

Once it's enabled, you're going to have to tell Ceedling a little bit about what you're looking for. This starts in the release build section. You tell it things like "what is the name of the output file I'm building" and "does it include assembly files" and even "are there other build artifacts I wish to save away when we're done?"

```
:release_build:
  :output: SuperSecretExecutable.hex
  :use_assembly: TRUE
  :artifacts:
   -  SuperSecretExecutable.map
```

Paths and Files

Next we're going to want to tell Ceedling where to find the files. Very likely, this work has been done for us already, if we're already using Ceedling to test. Both the :paths and :files sections of the project yaml file have :source and :include paths. These serve double-duty. They're where our release code was stored when we were making tests… and they are still our release code now that we want to build an actual release!

If you refer to the way paths and files are handled in the test documentation, you'll find that all the same tricks work here. Do you want to add a file to your release? Simple add it to a directory included in your release OR add the new directory to the search path. Do you want to remove a certain file from your release, but not the rest of that directory? You can use the -: operator to remove a specific file.

As an extra bonus, we recognize that there are standard libraries and paths that aren't needed during tests, but are critical to the success of your release application. For these, you can add to the :release_toolchain_include path listing.

```
:paths:
  :source:
    - src/**
  :include:
    - inc/*
  :release_toolchain_include:
    - /somewhere/else/includes
:files:
  :source:
    - -:src/not_ready_yet.c
  :include:
    - +:../another_include.h
```

Tools

Much like for tests, Ceedling allows you to configure your toolchain. You can add specifications for your compiler, assembler (if required), and linker. Ceedling will work to make sure that all the dependency tracking happens, and will inject the arguments you need in order to keep these builds running. Because the tool handling works the same as it does for testing, we're only going to mention that it's an option here… and you can refer to the test documentation or the full docs for more details.

```
:tools:
  :release_compiler:
  :release_assembler:
```

```
  :release_linker:
  :release_dependencies_generator:
```

Defines

Also similar to the test tool configuration, Ceedling allows you to inject specific defines. It supports defines for both the release and the preprocessor.

```
:defines:
  :release:
  :release_preprocess:
```

Libraries

Often, release builds depend on libraries. These may be provided by your toolchain vendor, your rtos, other tools, or even built internally for your own use. In any case, we can't have a complete release system without supporting them.

```
:libraries:
  :placement: :end
  :source:
     - m
  :system:
  :flag: "${1}" # or "-L ${1}" for example
```

The placement field allows you to configure where in the linking step the libraries are appended, before or after the object. You may add as many libraries as you require to the source section. Each sits on its own line and starts with an indented dash. The system section accepts paths to system libraries, in case the linker needs to know about those. Finally, the flag field is used to specify how the libraries are actually passed to the linker.

Running

Asking Ceedling to build a release is straightforward. The simplest approach is to just ask it to build a full release:

```
ceedling release
```

Like all ceedling commands, we can chain in a clobber in order to ensure that we're building everything from scratch:

```
ceedling clobber release
```

When needed, we can even ask particular files to be compiled directly on their own. This is done by inserting the filename for the * below:

```
ceedling release:compile:*
ceedling release:assemble:*
```

Final Words

We hope you've found this porting guide helpful. Keep in mind that there are more details in the CeedlingPacket.md that comes with the project, as well as additional documents packaged with this class. If none of these are sufficient, you are always welcome to reach out on the throwtheswitch.org/forums.

Happy Testing!