What the What?

Assembling build environments for C projects – especially with automated unit tests – is a pain. Whether it's Make or Rake or Premake or what-have-you, set up with an all-purpose build environment tool is tedious and requires considerable glue code to pull together the necessary tools and libraries. Ceedling allows you to generate an entire test and build environment for a C project from a single YAML configuration file. Ceedling is written in Ruby and works with the Rake build tool plus other goodness like Unity and CMock the unit testing and mocking frameworks for C. Ceedling and its complementary tools can support the tiniest of embedded processors, the beefiest 64 bit power houses available, and everything in between.

For a build project including unit tests and using the default toolchain gcc, the configuration file could be as simple as this:

```
:project:
  :build_root: project/build/
  :release_build: TRUE


:paths:
  :test:
    - tests/**
  :source:
    - source/**
```

From the command line, to build the release version of your project, you would simply run `rake release`. To run all your unit tests, you would run `rake test:all`. That's it!

Of course, many more advanced options allow you to configure your project with a variety of features to meet a variety of needs. Ceedling can work with practically any command line toolchain and directory structure – all by way of the configuration file. Further, because Ceedling piggy backs on Rake, you can add your own Rake tasks to accomplish project tasks outside of testing and release builds. A facility for plugins also allows you to extend Ceedling's capabilities for needs such as custom code metrics reporting and coverage testing.

What's with this Name?

Glad you asked. Ceedling is tailored for unit tested C projects and is built upon / around Rake (Rake is a Make replacement implemented in the Ruby scripting language). So, we've got C, our Rake, and the fertile soil of a build environment in which to grow and tend your project and its unit tests. Ta da – *Ceedling*.

What Do You Mean “tailored for unit tested C projects”?

Well, we like to write unit tests for our C code to make it lean and mean (that whole Test-Driven Development thing). Along the way, this style of writing C code spawned two tools to make the job easier: a unit test framework for C called *Unity* and a mocking library called *CMock*. And, though it's not directly related to testing, a C framework for exception handling called *CException* also came along.

These tools and frameworks are great, but they require quite a bit of environment support to pull them all together in a convenient, usable fashion. We started off with Rakefiles to assemble everything. These ended up being quite complicated and had to be hand-edited or created anew for each new project. Ceedling replaces all that tedium and rework with a configuration file that ties everything together.

Though Ceedling is tailored for unit testing, it can also go right ahead and build your final binary release artifact for you as well. Or, Ceedling and your tests can live alongside your existing release build setup.

Hold on. Back up. Ruby? Rake? YAML? Unity? CMock? CException?

Seem overwhelming? It's not bad at all, and for the benefits tests bring us, it's all worth it.

Ruby is a handy scripting language like Perl or Python. It's a modern, full featured language that happens to be quite handy for accomplishing tasks like code generation or automating one's workflow while developing in a compiled language such as C.

Rake is a utility written in Ruby for accomplishing dependency tracking and task automation common to building software. It's a modern, more flexible replacement for Make. Rakefiles are Ruby files, but they contain build targets similar in nature to that of Makefiles (but you can also run Ruby code in your Rakefile).

YAML is a “human friendly data serialization standard for all programming languages.” It's kinda like a markup language, but don't call it that. With a YAML library, you can serialize data structures to and from the file system in a textual, human readable form. Ceedling uses a serialized data structure as its configuration input.

Unity is a unit test framework for C. It provides facilities for test assertions, executing tests, and collecting / reporting test results. It consists of a single C source file and two C header files. Unity derives its name from its implementation in a single source file and from the nature of its implementation – Unity will build in any C toolchain and is configurable for even the very minimalist of processors.

CMock is a tool written in Ruby able to generate entire mock functions in C code from a given C header file. Mock functions are invaluable in interaction-based unit testing. CMock's generated C code uses Unity.

CException is a C source and header file that provide a simple exception mechanism for C by way of wrapping up the setjmp / longjmp standard library calls. Exceptions are a much cleaner and preferable alternative to managing and passing error codes up your return call trace.

Notes:

- YAML support is included with Ruby – requires no special installation or configuration.
- Unity, CMock, and CException are bundled with Ceedling, and Ceedling is designed to glue them all together for your project as seamlessly as possible.

Installation & Setup: What Exactly Do I Need to Get Started?

Installation and setup requires a handful of steps. Eventually, Ceedling will be packaged up in a better, more Ruby-appropriate fashion. For now – from scratch:

1. Download and install Ruby
2. Use Ruby's command line gem package manager to install Rake: `gem install rake`
3. Grab the Ceedling package and place it in your file system
   (it already contains Unity, CMock, and CException)
4. Create an empty build directory for your project
   (Ceedling will fill out the directory structure below the build root when executed)
5. Create a simple Rakefile (`rakefile.rb`) that contains a load call to Ceedling on your file system and a default task (tasks available to tie to `default` are listed in the next section):

```
load '<path>/ceedling/lib/rakefile.rb'

task :default => ['test:all', :release] # ex. run all tests & build release artifact
     # namespaced tasks must be quoted
     # top-level tasks are Ruby symbols denoted by a leading ':'
```

6. Create your project YAML file (more on this later in this document). `project.yml` is the default file name Ceedling recognizes in the working directory from which Rake is run (Rake is the tool we actually use to take advantage of what Ceedling provides).

Notes:

- Steps 1-3 are a one time affair for your local environment. When steps 1-3 are completed once, only steps 4-6 are needed for each new project.
- See the sample starter project for a working setup. When steps 1-3 are complete and assuming you have gcc in your path (Ceedling's default toolchain), you will only need to edit the path within the sample Rakefile (see step 5 above) to yield a working, albeit simple, project. The default task need not be defined, but it's not a bad idea to do so.
- Certain advanced features of Ceedling rely on gcc and cpp as preprocessing tools. In most *nix systems, these tools are already available. For Windows environments, we recommend the mingw project (Minimalist GNU for Windows). This represents an optional, additional setup / installation step to complement the list above.
- To use a project file name other than the default `project.yml` or place the project file in a directory other than the one in which you'll run Rake, create an environment variable `CEEDLING_MAIN_PROJECT_FILE` with your desired project file path.
- To better understand Rake conventions, Rake execution, and Rakefiles, consult the Rake tutorial, examples, and user guide.

Now What? How Do I Make It GO?

We're getting a little ahead of ourselves here, but it's good context on how to drive this bus. Everything is done via the command line. We'll cover conventions and how to actually configure your project in later sections.

To run tests, build your release artifact, etc., you will be interacting with Rake on the command line. Ceedling works with Rake to present you with named tasks that coordinate the file generation and build steps needed to accomplish something useful. You can also add your own independent Rake tasks or create plugins to extend Ceedling (more on this later).

Rake Command	**What It Does**
`rake` *[no arguments]*	Run the default Rake task (conveniently recognized by the name `default` by Rake). Neither Rake nor Ceedling provide a default task. Rake will abort if run without arguments when no default task is defined. You can conveniently define a default task in the Rakefile discussed in the preceding setup & installation section of this document.
`rake -T`	List all available Rake tasks with descriptions (Rake tasks without descriptions are not listed). `-T` is a command line switch for Rake and not the same as tasks that follow.
`rake <tasks...> --trace`	For advanced users troubleshooting a confusing build error, debug Ceedling or a plugin, `--trace` provides a stack trace of dependencies walked during task execution and any Ruby failures along the way. Note that `--trace` is a command line switch for Rake and is not the same as tasks that follow.
`rake paths:*`	List all paths collected from [:paths] entries in your YAML config file where * is the name of any section contained in [:paths]. This task is helpful in verifying the expansion of path wildcards / globs specified in the [:paths] section of your config file.

`rake files:assembly`
`rake files:header`
`rake files:source`
`rake files:test`
	List all files and file counts collected from the relevant search paths specified by the [:paths] entries of your YAML config file. The `files:assembly` task will only be available if assembly support is enabled in the [:release_build] section of your configuration file.

`rake options:*`	Load and merge configuration settings into the main project configuration. Each task is named after a *.yml file found in the configured options directory. See documentation for the configuration setting [:project][:options_path] and for options files in advanced topics.
`rake test:all`	Run all unit tests (rebuilding anything that's changed along the way).
`rake test:delta`	Run only those unit tests for which the source or test files have changed (i.e. incremental build). Note: with the [:project][:use_test_preprocessor] configuration file option set, runner files are always regenerated limiting the total efficiency this text execution option can afford.
`rake test:*`	Execute the named test file or the named source file that has an accompanying test. No path. Examples: `rake test:foo.c` or `rake test:test_foo.c`
`rake test:pattern[*]`	Execute any tests whose name and/or path match the regular expression pattern (case sensitive). Example: `rake "test:pattern[(I|i)nit]"` will execute all tests named for initialization testing. Note: quotes may be necessary around the rake parameter to distinguish regex characters from command line operators.
`rake test:path[*]`	Execute any tests whose path contains the given string (case sensitive). Example: `rake test:path[foo/bar]` will execute all tests whose path contains `foo/bar`. Note: both directory separator characters `/` and `\` are valid.
`rake release`	Build all source into a release artifact (if the release build option is configured).
`rake release:compile:*`	Sometimes you just need to compile a single file dagnabit. Example: `rake release:compile:foo.c`
`rake release:assemble:*`	Sometimes you just need to assemble a single file doggonit. Example: `rake release:assemble:foo.s`
`rake logging <tasks...>`	Enable logging to `<build path>/logs`. Must come before test and release tasks to log their steps and output. Log names are a concatenation of project, user, and option files loaded. User and option files are documented in the advanced topics section of this document.
`rake verbosity[x] <tasks...>`	Change the default verbosity level. [x] ranges from 0 (quiet) to 4 (obnoxious). Level [3] is the default. The verbosity task must precede all tasks in the command line list for which output is desired to be seen. Verbosity settings are generally most meaningful in conjunction with test and release tasks.
`rake clean`	Deletes all toolchain binary artifacts (object files, executables), test results, and any temporary files. Clean produces no output at the command line unless verbosity has been set to an appreciable level.
`rake clobber`	Extends clean task's behavior to also remove generated files: test runners, mocks, preprocessor output. Clobber produces no output at the command line unless verbosity has been set to an appreciable level.

To better understand Rake conventions, Rake execution, and Rakefiles, consult the Rake tutorial, examples, and user guide.

Individual test and release file tasks are not listed in `-T` output. Because so many files may be present it's unwieldy to list them all.

Multiple rake tasks can be executed at the command line (order is executed as provided). For example, `rake clobber test:all release` will removed all generated files; build and run all tests; and then build all source – in that order. If any Rake task fails along the way, execution halts before the next task.

The `clobber` task removes certain build directories in the course of deleting generated files. In general, it's best not to add

to source control any Ceedling generated directories below the root of your top-level build directory. That is, leave anything Ceedling & its accompanying tools generate out of source control (but go ahead and add the top-level build directory that holds all that stuff). Also, since Ceedling is pretty smart about what it rebuilds and regenerates, you needn't clobber often.

Important Conventions

Directory Structure, Filenames & Extensions

Much of Ceedling's functionality is driven by collecting files matching certain patterns inside the paths it's configured to search. See the documentation for the [:extensions] section of your configuration file (found later in this document) to configure the file extensions Ceedling uses to match and collect files. Test file naming is covered later in this section.

Test files and source files must be segregated by directories. Any directory structure will do. Tests can be held in subdirectories within source directories, or tests and source directories can be wholly separated at the top of your project's directory tree.

Search Path Order

When Ceedling searches for files (e.g. looking for header files to mock) or when it provides search paths to any of the default gcc toolchain executables, it organizes / prioritizes its search paths. The order is always: test paths, support paths, source paths, and then include paths. This can be useful, for instance, in certain testing scenarios where we desire Ceedling or a compiler to find a stand-in header file in our support directory before the actual source header file of the same name.

This convention only holds when Ceedling is using its default tool configurations and / or when tests are involved. If you define your own tools in the configuration file (see the [:tools] section documented later in this here document), you have complete control over what directories are searched and in what order. Further, test and support directories are only searched when appropriate. That is, when running a release build, test and support directories are not used at all.

Source Files & Binary Release Artifacts

Your binary release artifact results from the compilation and linking of all source files Ceedling finds in the specified source directories. At present only source files with a single (configurable) extension are recognized. That is, *.c and *.cc files will not both be recognized – only one or the other. See the configuration options and defaults in the documentation for the [:extensions] sections of your configuration file (found later in this document).

Test Files & Executable Test Fixtures

Ceedling builds each individual test file with its accompanying source file(s) into a single, monolithic test fixture executable. Test files are recognized by a naming convention: a (configurable) prefix such as "`test_`" in the file name with the same file extension as used by your C source files. See the configuration options and defaults in the documentation for the [:project] and [:extensions] sections of your configuration file (found later in this document). Depending on your configuration options, Ceedling can recognize a variety of test file naming patterns in your test search paths. For example: `test_some_super_functionality.c`, `TestYourSourceFile.cc`, or `testing_MyAwesomeCode.C` could each be valid test file names. Note, however, that Ceedling can recognize only one test file naming convention per project.

Ceedling knows what files to compile and link into each individual test executable by way of the #include list contained in each test file. Any C source files in the configured search directories that correspond to the header files included in a test file will be compiled and linked into the resulting test fixture executable. From this same #include list, Ceedling knows which files to mock and compile and link into the test executable (if you use mocks in your tests). That was a lot of clauses and information in a very few sentences; the example that follows in a bit will make it clearer.

By naming your test functions according to convention, Ceedling will extract and collect into a runner C file calls to all your test case functions. This runner file handles all the execution minutiae so that your test file can be quite simple and so that you never forget to wire up a test function to be executed. In this generated runner lives the `main()` entry point for the resulting test executable. There are no configuration options for the naming convention of your test case functions. A test case function signature must have these three elements: void return, void parameter list, and the function name prepended with lowercase "`test`". In other words, a test function signature should look like this: `void test`*`[any name you like]`*`(void)`.

A commented sample test file follows on the next page. Also, see the sample project contained in the Ceedling documentation bundle.

```
// test_foo.c ---------------------------------------------
#include "unity.h"     // compile/link in Unity test framework
#include "types.h"     // header file with no *.c file -- no compilation/linking
#include "foo.h"       // source file foo.c under test
#include "mock_bar.h"  // bar.h will be found and mocked as mock_bar.c + compiled/linked in;
                       // foo.c includes bar.h and uses functions declared in it
#include "mock_baz.h"  // baz.h will be found and mocked as mock_baz.c + compiled/linked in
                       // foo.c includes baz.h and uses functions declared in it


void setUp(void) {}    // every test file requires this function;
                       // setUp() is called by the generated runner before each test case function

void tearDown(void) {} // every test file requires this function;
                       // tearDown() is called by the generated runner before each test case function

// a test case function
void test_Foo_Function1_should_Call_Bar_AndGrill(void)
{
    Bar_AndGrill_Expect();                   // setup function from mock_bar.c that instructs our
                                             // framework to expect Bar_AndGrill() to be called once
    TEST_ASSERT_EQUAL(0xFF, Foo_Function1()); // assertion provided by Unity
                                             // Foo_Function1() calls Bar_AndGrill() & returns a byte
}

// another test case function
void test_Foo_Function2_should_Call_Baz_Tec(void)
{
    Baz_Tec_ExpectAnd_Return(1);       // setup function provided by mock_baz.c that instructs our
                                       // framework to expect Baz_Tec() to be called once and return 1
    TEST_ASSERT_TRUE(Foo_Function2()); // assertion provided by Unity
}

// end of test_foo.c ---------------------------------------
```

From the test file specified above Ceedling will generate `test_foo_runner.c`; this runner file will contain `main()` and call both of the example test case functions.

The final test executable will be `test_foo.exe` (for Windows machines or `test_foo.out` for *nix systems – depending on default or configured file extensions). Based on the #include list above, the test executable will be the output of the linker having processed `unity.o`, `foo.o`, `mock_bar.o`, `mock_baz.o`, `test_foo.o`, and `test_foo_runner.o`. Ceedling finds the files, generates mocks, generates a runner, compiles all the files, and links everything into the test executable. Ceedling will then run the test executable and collect test results from it to be reported to the developer at the command line.

For more on the assertions and mocks shown, consult the documentation for Unity and CMock.

The Magic of Dependency Tracking

Ceedling is pretty smart in using Rake to build up your project's dependencies. This means that Ceedling automagically rebuilds all the appropriate files in your project when necessary: when your configuration changes, Ceedling or any of the other tools are updated, or your source or test files change. For instance, if you modify a header file that is mocked, Ceedling will ensure that the mock is regenerated and all tests that use that mock are rebuilt and re-run when you initiate a relevant testing task. When you see things rebuilding, it's for a good reason. Ceedling attempts to regenerate and rebuild only what's needed for a given execution of a task. In the case of large projects, assembling dependencies and acting upon them can cause some delay in executing tasks.

With one exception, the trigger to rebuild or regenerate a file is always a disparity in timestamps between a target file and its source – if an input file is newer than its target dependency, the target is rebuilt or regenerated. For example, if the C source file from which an object file is compiled is newer than that object file on disk, recompilation will occur (of course, if no object file exists on disk, compilation will always occur). The one exception to this dependency behavior is specific to your input configuration. Only if your logical configuration changes will a system-wide rebuild occur. Reorganizing your input configuration or otherwise updating its file timestamp without modifying the values within the file will not trigger a rebuild. This behavior handles the various ways in which your input configuration can change (discussed later in this document) without having changed your actual project YAML file.

Ceedling needs a bit of help to accomplish its magic with deep dependencies. Shallow dependencies are straightforward: a mock is dependent on the header file from which it's generated, a test file is dependent upon the source files it includes (see the preceding conventions section), etc. Ceedling handles these “out of the box.” Deep dependencies are specifically a C-related phenomenon and occur as a consequence of include statements within C source files. Say a source file includes a header file and that header file in turn includes another header file which includes still another header file. A change to the deepest header file should trigger a recompilation of the source file, a relinking of all the object files comprising a test fixture, and a new execution of that test fixture.

Ceedling can handle deep dependencies but only with the help of a C preprocessor. Ceedling is quite capable, but a full C preprocessor it ain't. Your project can be configured to use a C preprocessor or not. Simple projects or large projects constructed so as to be quite flat in their include structure generally don't need auxiliary dependency preprocessing – and can enjoy the benefits of faster execution. Legacy code, on the other hand, will almost always want to be tested with auxiliary preprocessing enabled. Set up of the C preprocessor is covered in the documentation for the [:project] and [:tools] section of the configuration file (later in this document). Ceedling contains all the configuration necessary to use the gcc preprocessor by default. That is, as long as gcc is in your system search path, auxiliary preprocessing of deep dependencies is available to you by simply enabling it in your project configuration file.

Ceedling's Build Output

Ceedling requires a top-level build directory for all the stuff that it, the accompanying test tools, and your toolchain generate. That build directory's location is configured in the [:project] section of your configuration file (discussed later). There can be a ton of generated files. By and large, you can live a full and meaningful life knowing absolutely nothing at all about the files and directories generated below the root build directory.

As noted already, it's good practice to add your top-level build directory to source control but nothing generated beneath it. You'll spare yourself headache if you let Ceedling delete and regenerate files and directories in a non-versioned corner of your project's filesystem beneath the top-level build directory.

The `artifacts` directory is the one and only directory you may want to know about beneath the top-level build directory. The subdirectories beneath `artifacts` will hold your binary release target output (if your project is configured for release builds) and will serve as the conventional location for plugin output. This directory structure was chosen specifically because it tends to work nicely with Continuous Integration setups that recognize and list build artifacts for retrieval / download.

The Almighty Project Configuration File (in Glorious YAML)

Please consult YAML documentation for the finer points of format and to understand details of our YAML-based configuration file. We recommend Wikipedia's entry on YAML for this. A few highlights from that reference page:

- YAML streams are encoded using the set of printable Unicode characters, either in UTF-8 or UTF-16
- Whitespace indentation is used to denote structure; however tab characters are never allowed as indentation
- Comments begin with the number sign (#), can start anywhere on a line, and continue until the end of the line unless enclosed by quotes
- List members are denoted by a leading hyphen (-) with one member per line, or enclosed in square brackets ([ ]) and separated by comma space (,)
- Hashes are represented using the colon space (:) in the form key: value, either one per line or enclosed in curly braces ({ }) and separated by comma space (,)
- Strings (scalars) are ordinarily unquoted, but may be enclosed in double-quotes ("), or single-quotes (')
- YAML requires that colons and commas used as list separators be followed by a space so that scalar values containing embedded punctuation can generally be represented without needing to be enclosed in quotes
- Repeated nodes are initially denoted by an ampersand (&) and thereafter referenced with an asterisk (*)

Notes on what follows:

- Each of the following sections represent top-level entries in the YAML configuration file.
- Unless explicitly specified in the configuration file, default values are used by Ceedling.
- These three settings, at minimum, must be specified:
  - [:project][:build_root]
  - [:paths][:source]
  - [:paths][:test]
- As much as is possible, Ceedling validates your settings in properly formed YAML.
- Improperly formed YAML will cause a Ruby error when the YAML is parsed. This is usually accompanied by a complaint with line and column number pointing into the project file.
- Certain advanced features rely on gcc and cpp as preprocessing tools. In most *nix systems, these tools are already available. For Windows environments, we recommend the mingw project (Minimalist GNU for Windows).
- Ceedling is primarily meant as a build tool to support automated unit testing. All the heavy lifting is involved there. Creating a binary release build artifact is quite trivial in comparison. Consequently, most default options and the construction of Ceedling itself is skewed towards supporting testing though Ceedling can, of course, build your binary release artifact as well.

Let's Be Careful Out There: Ceedling performs validation on the values you set in your configuration file (this assumes your YAML is correct and will not fail format parsing, of course). That said, validation is limited to only those settings Ceedling uses and those that can be reasonably validated. Ceedling does not limit what can exist within your configuration file. In this way, you can take full advantage of YAML as well as add sections and values for use in your own custom plugins (documented later). The consequence of this is simple but important. A misspelled configuration section name or value name is unlikely to cause Ceedling any trouble. Ceedling will happily process that section or value and simply use the properly spelled default maintained internally – thus leading to unexpected behavior without warning.

project: *global project settings*

Setting	Description	Default
build_root	Top level directory into which generated path structure and files are placed. *Note: this is one of the handful of configuration values that must be set.* The specified path can be absolute or relative to your working directory.	<none>
use_exceptions	Configures the build environment to make use of CException. Note that if you do not use exceptions, there's no harm in leaving this as its default value.	TRUE
use_mocks	Configures the build environment to make use of CMock. Note that if you do not use mocks, there's no harm in leaving this setting as its default value.	TRUE
use_test_preprocessor	This option allows Ceedling to work with test files that contain conditional compilation statements (e.g. #ifdef) and header files you wish to mock that contain conditional preprocessor statements and/or macros. Ceedling and CMock are advanced tools with sophisticated parsers. However, they do not include entire C language preprocessors. Consequently, with this option enabled, Ceedling will use gcc's preprocessing mode and the cpp preprocessor tool to strip down / expand test files and headers to their applicable content which can then be processed by Ceedling and CMock. With this option enabled, the gcc & cpp tools must exist in an accessible system search path and test runner files are *always* regenerated.	FALSE
use_auxiliary_dependencies	The base rules and tasks that Ceedling creates using Rake capture most of the dependencies within a standard project (e.g. when the source file accompanying a test file changes, the corresponding test fixture executable will be rebuilt when tests are re-run). However, deep dependencies cannot be captured this way. If a typedef or macro changes in a header file three levels of #include statements deep, this option allows the appropriate incremental build actions to occur for both test execution and release builds. This is accomplished by using the dependencies discovery mode of gcc. With this option enabled, gcc must exist in an accessible system search path.	FALSE
test_file_prefix	Ceedling collects test files by convention from within the test file search paths. The convention includes a unique name prefix and a file extension matching that of source files. Why not simply recognize all files in test directories as test files? By using the given convention, we have greater flexibility in what we do with C files in the test directories.	“test_”
options_path	Just as you may have various build configurations for your source codebase, you may need variations of your project configuration. By specifying an options path, Ceedling will search for other project YAML files, make command line tasks available (`rake options:variation` for a `variation.yml` file), and merge the project configuration of these option files in with the main project file at runtime. See advanced topics. Note these Rake tasks, like verbosity or logging control, at the command line must come before the test or release task they are meant to modify.	<none>
release_build	When enabled, a `release` Rake task is exposed. This configuration option requires a corresponding release compiler and linker to be defined (gcc is used as the default). More release configuration options are available in the `release_build` section.	FALSE

Example [:project] YAML blurb

```
:project:
  :build_root: project_awesome/build
  :use_exceptions: FALSE
  :use_test_preprocessor: TRUE
  :use_auxiliary_dependencies: TRUE
  :release_build: TRUE
```

release_build: *configuration of optional release build binary artifact generation*

Ceedling is primarily concerned with facilitating the somewhat complicated mechanics of automating unit tests. The same mechanisms are easily capable of building a final release binary artifact (i.e. non test code; the thing that is your final working software that you execute on target hardware).

Setting	**Description**	**Default**
output	The name of your release build binary artifact to be found in `<build path>/artifacts/release`. Ceedling sets the default artifact file extension to that as is explicitly specified in the [:extensions] section or as is system specific otherwise.	project.exe or project.out
use_assembly	If assembly code is present in the source tree, this option causes Ceedling to create appropriate build directories and use an assembler tool (default is the GNU tool as – override available in the [:tools] section.	FALSE

Example [:release_build] YAML blurb

```
:release_build:
  :output: top_secret.exe
  :use_assembly: TRUE
```

paths: *options controlling search paths for source and header (and assembly) files*

Setting	**Description**	**Default**
test	All C files containing unit test code. *Note: this is one of the handful of configuration values that must be set.*	[] (empty)
source	All C files containing release code (code to be tested). *Note: this is one of the handful of configuration values that must be set.*	[] (empty)
support	Any C files you might need to aid your unit testing. For example, on occasion, you may need to create a header file containing a subset of function signatures matching those elsewhere in your code (e.g. a subset of your OS functions, a portion of a library API, etc.). Why? To provide finer grained control over mock function substitution or limiting the size of the generated mocks.	[] (empty)
include	Any header files not already in the source search path. Note there's no practical distinction between this search path and the source search path; it's merely to provide options or to support any peculiar source tree organization.	[] (empty)
test_toolchain_include	System header files needed by the test toolchain – should your compiler be unable to find them, finds the wrong system include search path, or you need a creative solution to a tricky technical problem. Note that if you configure your own toolchain in the [:tools] section, this search path is largely meaningless to you. However, this is a convenient way to control the system include path should you rely on the default gcc tools.	[] (empty)
release_toolchain_include	Same as preceding albeit related to the release toolchain.	[] (empty)

Notes on path grammar within the [:paths] section:

- The order of the search paths listed in the [:paths] section is preserved when used by an entry in the [:tools] section
- Wherever multiple path lists are combined for use Ceedling prioritizes path groups as follows: test paths, support paths, source paths, include paths. This can be useful, for instance, in certain testing scenarios where we desire Ceedling or the compiler to find a stand-in header file before the actual source header file of the same name.
- Paths:
  1. can be absolute or relative
  2. can be singly explicit – a single fully specified path
  3. can include a glob operator (more on this below)
  4. default as an addition to a specific search list (more on this in the examples)
  5. can act to subtract from a glob included in the path list (more on this in the examples)

Globs as used by Ceedling are wildcards for specifying directories without the need to list each and every required search path. Ceedling globs operate just as Ruby globs except that they are limited to matching directories and not files. Glob operators include the following * ** ? [-] {,} (note: this list is space separated and not comma separated as commas are used within the bracket operators).

`*` All subdirectories of depth 1 below the parent path and including the parent path

`**` All subdirectories recursively discovered below the parent path and including the parent path

`?` Single alphanumeric character wildcard

`[x-y]` Single alphanumeric character as found in the specified range

`{x,y}` Single alphanumeric character from the specified list

Example [:paths] YAML blurbs

```
:paths:
  :source:              #together the following comprise all source search paths
    - project/source/*  #expansion yields all subdirectories of depth 1 plus parent directory
    - project/lib       #single path
  :test:                #all test search paths
    - project/**/test?  #expansion yields any subdirectory found anywhere in the project that
                        #begins with “test” and contains 5 characters


:paths:
  :source:                            #all source search paths
    - +:project/source/**             #all subdirectories recursively discovered plus parent directory
    - -:project/source/os/generated   #subtract os/generated directory from expansion of above glob
                                      #note that '+:' notation is merely aesthetic; default is to add
  :test:                              #all test search paths
    - project/test/bootloader         #explicit, single search paths (searched in the order specified)
    - project/test/application
    - project/test/utilities
```

Globs can require trial and error to arrive at your intended results. Use the `rake paths:*` command line options (documented in preceding section) to verify your settings.

environment: *inserts environment variables into the shell instance executing configured tools*

Ceedling creates environment variables from any key / value pairs in the environment section. Keys become an environment variable name in uppercase. The values are strings assigned to those environment variables.

Ceedling is able to execute inline Ruby string substitution code to set environment variables. This evaluation occurs when the project file is first processed for any environment pair's value string including the Ruby string substitution pattern `#{…}`. Note that environment value strings that *begin* with this pattern should always be enclosed in quotes. YAML defaults to processing unquoted text as a string; quoting text is optional. If an environment pair's value string begins with the Ruby string substitution pattern, YAML will interpret the string as a Ruby comment (because of the '#'). Enclosing each environment value string in quotes is a safe practice.

[:environment] entries are processed in the configured order (later entries can reference earlier entries).

Example [:environment] YAML blurb

```
:environment:
  - :license_server: gizmo.intranet         #LICENSE_SERVER set with value "gizmo.intranet"
  - :license: "#{`license.exe`}"            #LICENSE set to string generated from shelling out to
                                            #execute license.exe; note use of enclosing quotes
  - :path: Tools/gizmo/bin;#{ENV['PATH']}   #ruby code will prepend PATH with gizmo tool path
                                            #pattern #{…} triggers ruby evaluation string substitution
                                            #note value string did not require enclosing quotes
  - :logfile: system/logs/thingamabob.log   #LOGFILE set with path for a log file
```

defines: *command line defines used in test and release compilation by configured tools*

Setting	Description	Default
test	Defines needed for testing. Useful for: 1. test files containing conditional compilation statements (i.e. tests active in only certain contexts) 2. testing legacy source wherein the isolation of source under test afforded by Ceedling and its complementary tools leaves certain symbols unset when source files are compiled in isolation	[] (empty)
test_preprocess	If [:project][:use_test_preprocessor] or [:project][:use_auxiliary_dependencies] is set and code is structured in a certain way, the gcc preprocessor may need symbol definitions to properly preprocess files to extract function signatures for mocking and extract deep dependencies for incremental builds.	[] (empty)
release	Defines needed for the release build binary artifact.	[] (empty)
release_preprocess	If [:project][:use_auxiliary_dependencies] is set and code is structured in a certain way, the gcc preprocessor may need symbol definitions to properly preprocess files for incremental release builds due to deep dependencies.	[] (empty)

Example [:defines] YAML blurb

```
:defines:
  :test:
    - UNIT_TESTING  #for select cases in source to allow testing with a changed behavior or interface
    - OFF=0
    - ON=1
    - FEATURE_X=ON
  :source:
    - FEATURE_X=ON
```

extension: *configure file name extensions used to collect lists of files searched in [:paths]*

Setting	Description	Default
header	C header files	.h
source	C code files (whether source or test files)	.c
assembly	Assembly files (contents wholly assembly instructions)	.s
object	Resulting binary output of C code compiler (and assembler)	.o
executable	Binary executable to be loaded and executed upon target hardware	.exe or .out (Win or *nix)
testpass	Test results file (not likely to ever need a new value)	.pass
testfail	Test results file (not likely to ever need a new value)	.fail
dependencies	File containing make-style dependency rules created by gcc preprocessor	.d

Example [:extension] YAML blurb

```
:extension:
  :source: .cc
  :executable: .bin
```

cmock: *configure CMock's code generation options and set symbols used to modify CMock's compiled features*

Ceedling sets values for a subset of CMock settings. All CMock options are available to be set, but only those options set by Ceedling in an automated fashion are documented below. See CMock documentation.

Setting	Description	Default
enforce_strict_ordering	Tests fail if expected call order is not same as source order	TRUE
mock_path	Path for generated mocks	`<build path>/tests/mocks`
defines	List of conditional compilation symbols used to configure CMock's compiled features. See CMock documentation to understand available options. No symbols must be set unless defaults are inappropriate for your specific environment. All symbols are used only by Ceedling to compile CMock C code; contents of [:defines] are ignored by CMock's Ruby code when instantiated.	[] (empty)
verbosity	If not set, defaults to Ceedling's verbosity level	
plugins	If [:project][:use_exceptions] is enabled, the internal plugins list is pre-populated with 'cexception'. Whether or not you have included [:cmock][:plugins] in your configuration file, Ceedling automatically adds 'cexception' to the plugin list if exceptions are enabled. To add to the list Ceedling provides CMock, simply add [:cmock][:plugins] to your configuration and specify your desired additional plugins.	
includes	If [:cmock][:unity_helper] set, pre-populated with unity_helper file name (no path). The [:cmock][:includes] list works identically to the plugins list above with regard to adding additional files to be inserted within mocks as #include statements.	

The last four settings above are directly tied to other Ceedling settings; hence, why they are listed and explained here. The first setting above, [:enforce_strict_ordering], defaults to FALSE within CMock. It is set to TRUE by default in Ceedling as our way of encouraging you to use strict ordering. It's a teeny bit more expensive in terms of code generated, test execution time, and complication in deciphering test failures.

However, it's good practice. And, of course, you can always disable it by overriding the value in the Ceedling YAML configuration file.

cexception: *configure symbols used to modify CException's compiled features*

Setting	Description	Default
defines	List of conditional compilation symbols used to configure CException's features in its source and header files. See CException documentation to understand available options. No symbols must be set unless the defaults are inappropriate for your specific environment.	[] (empty)

unity: *configure symbols used to modify Unity's compiled features*

Setting	Description	Default
defines	List of conditional compilation symbols used to configure Unity's features in its source and header files. See Unity documentation to understand available options. No symbols must be set unless the defaults are inappropriate for your specific environment.	[] (empty)

Notes on Unity configuration:

- **Verification** – Ceedling does no verification of your configuration values. In a properly configured setup, your Unity configuration values are processed, collected together with any test define symbols you specify elsewhere, and then passed to your toolchain during test compilation. Unity's conditional compilation statements, your toolchain's preprocessor, and/or your toolchain's compiler will complain appropriately if your specified configuration values are incorrect, incomplete, or incompatible.
- **Routing $stdout** – Unity defaults to using `putchar()` in C's standard library to display test results. For more exotic environments than a desktop with a terminal (e.g. running tests directly on a non-PC target), you have options. For example, you could create a routine that transmits a character via RS232 or USB. Once you have that routine, you can replace `putchar()` calls in Unity by overriding the function-like macro `UNITY_OUTPUT_CHAR`. Consult your toolchain and shell documentation.

Example [:unity] YAML blurbs

```
:unity: #itty bitty processor & toolchain with limited test execution options
  :defines:
    - UNITY_INT_WIDTH=16           #16 bit processor without support for 32 bit instructions
    - UNITY_EXCLUDE_FLOAT          #no floating point unit
        #let's say environment & tools provide no way to run tests on desktop so we gotta go on target
        #replace putchar() with write_usart() via command line specified macro (gcc style)
        #note escaped quotes for our hypothetical shell that doesn't like parens in arguments
        #transformed into -D"UNITY_OUTPUT_CHAR(a)=write_usart(a)" at command line by [:tools] entry
    - "\"UNITY_OUTPUT_CHAR(a)=write_usart(a)\""


:unity: #great big gorilla processor that grunts and scratches
  :defines:
    - UNITY_SUPPORT_64                   #big memory, big counters, big registers
    - UNITY_LINE_TYPE=\"unsigned int\"    #apparently we're using really long test files,
    - UNITY_COUNTER_TYPE=\"unsigned int\" #and we've got a ton of test cases in those test files
    - UNITY_FLOAT_TYPE=\"double\"         #you betcha
```

tools: *a means for representing command line tools for use under Ceedling's automation framework*

Ceedling requires a variety of tools to work its magic. By default, the GNU toolchain (gcc, cpp, as) are configured and ready for use with no additions to the project configuration YAML file. However, as most work will require a project-specific toolchain, Ceedling provides a generic means for specifying / overriding tools.

Setting	Description	Default
test_compiler	Compiler for test & source-under-test code	gcc
test_linker	Linker to generate test fixture executables	gcc
test_fixture	Executable test fixture	${1}
test_includes_preprocessor	Extractor of #include statements	cpp
test_file_preprocessor	Preprocessor of test files (expanding macros, handling conditional compilation statements)	gcc
test_dependencies_generator	Discovers deep dependencies of test and source-under-test files (for incremental builds)	gcc
release_compiler	Compiler for release source code	gcc
release_assembler	Assembler for release assembly code	as
release_linker	Linker for release source code	gcc
release_dependencies_generator	Discovers deep dependencies of source files (for incremental builds)	gcc

A Ceedling tool has a handful of essential elements:

1. [:executable] – command line executable having the form of:
    1. a fully specified absolute file path
    2. a relative file path
    3. executable file name with no path (but available in a system search path)
2. [:name] – simple name (e.g. “nickname”) of tool beyond its executable name (if not explicitly set then Ceedling will form a name from the tool's YAML entry name)
3. [:stderr_redirect] – optional control of capturing $stderr messages
(defaults to :none if unspecified; currently only meaningfully used in test fixture tools)
4. [:arguments] – list of command line arguments and substitutions necessary to cause tool to accomplish useful work

Tool Element Runtime Substitution

To accomplish useful work on multiple files, a configured tool will most often require that some number of its arguments or even the executable itself change for each run. Consequently, every tool's argument list and executable field possess two means for substitution at runtime. Ceedling provides two kinds of inline Ruby execution and a notation for populating elements with dynamically gathered values within the build environment.

Tool Element Runtime Substitution: Inline Ruby Execution

In-line Ruby execution works similarly to that demonstrated for the [:environment] section except that substitution occurs as the tool is executed and not at the time the configuration file is first scanned.

`#{…}` Ruby string substitution pattern wherein the containing string is expanded to include the string generated by Ruby code between the braces. Multiple instances of this expansion can occur within a single tool element entry string. Note that if this string substitution pattern occurs at the very beginning of a string in the YAML configuration the entire string should be enclosed in quotes (see the [:environment] section for further explanation on this point).

`{…}` If a tool element string begins and ends with braces, it signifies that Ceedling should execute the Ruby

code contained within those braces. Say you have a collection of paths on disk and some of those paths include spaces. Further suppose that a single tool that must use those paths requires those spaces to be escaped, but all other uses of those paths requires the paths to remain unchanged. You could use this Ceedling feature to insert Ruby code that iterates those paths and escapes those spaces in the array as used by the tool of this example.

Tool Element Runtime Substitution: Notational Substitution

A Ceedling tool's other form of dynamic substitution relies on a '$' notation. These '$' operators can exist anywhere in a string and can be decorated in any way needed. To use a literal '$', escape it as '\\$'.

`$` Simple substitution for value(s) globally available within the runtime (most often a string or an array).

`${#}` When a Ceedling tool's command line is expanded from its configured representation and used within Ceedling Ruby code, certain calls to that tool will be made with a parameter list of substitution values. Each numbered substitution corresponds to a position in a parameter list. Ceedling Ruby code expects that configured compiler and linker tools will contain ${1} and ${2} replacement arguments. In the case of a compiler ${1} will be a C code file path, and ${2} will be the file path of the resulting object file. For a linker ${1} will be an array of object files to link, and ${2} will be the resulting binary executable. For an executable test fixture ${1} is either the binary executable itself (when using a local toolchain such as gcc) or a binary input file given to a simulator in its arguments.

Example [:tools] YAML blurbs

```
:tools:
  :test_compiler:
     :executable: compiler              #exists in system search path
     :name: 'acme test compiler'
     :arguments:
        - -I"$": COLLECTION_PATHS_TEST_TOOLCHAIN_INCLUDE              #expands to -I search paths
        - -I"$": COLLECTION_PATHS_TEST_SUPPORT_SOURCE_INCLUDE_VENDOR  #expands to -I search paths
        - -D$: COLLECTION_TEST_DEFINES  #expands to all -D defined symbols
        - --network-license             #simple command line argument
        - -optimize-level 4             #simple command line argument
        - "#{`args.exe -m acme.prj`}"   #in-line ruby sub to shell out & build string of arguments
        - -c ${1}                       #source code input file (Ruby method call param list sub)
        - -o ${2}                       #object file output (Ruby method call param list sub)
  :test_linker:
     :executable: /programs/acme/bin/linker.exe    #absolute file path
     :name: 'acme test linker'
     :arguments:
        - ${1}               #list of object files to link (Ruby method call param list sub)
        - -l$-lib:           #inline yaml array substitution to link in foo-lib and bar-lib
           - foo
           - bar
        - -o ${2}            #executable file output (Ruby method call param list sub)
  :test_fixture:
     :executable: tools/bin/acme_simulator.exe  #relative file path to command line simulator
     :name: 'acme test fixture'
     :stderr_redirect: :win                     #inform Ceedling what model of $stderr capture to use
     :arguments:
        - -mem large   #simple command line argument
        - -f "${1}"    #binary executable input file to simulator (Ruby method call param list sub)
```

Resulting command line constructions from preceding example [:tools] YAML blurbs

```
> compiler -I"/usr/include" -I"project/tests" -I"project/tests/support" -I"project/source"
  -I"project/include" -DTEST -DLONG_NAMES –network-license -optimize-level 4 arg-foo arg-bar arg-baz -c
  project/source/source.c -o build/tests/out/source.o
```

*[notes: (1.) "arg-foo arg-bar arg-baz" is a fabricated example string collected from $stdout as a result of shell execution of args.exe
(2.) the -c and -o arguments are fabricated examples simulating a single compilation step for a test; ${1} & ${2} are single files]*

```
> \programs\acme\bin\linker.exe thing.o unity.o test_thing_runner.o test_thing.o mock_foo.o mock_bar.o
  -lfoo-lib -lbar-lib -o build\tests\out\test_thing.exe
```

[note: in this scenario ${1} is an array of all the object files needed to link a test fixture executable]

```
> tools\bin\acme_simulator.exe -mem large -f "build\tests\out\test_thing.bin 2>&1"
```

*[notes: (1.) :executable could have simply been ${1} – if we were compiling and running native executables instead of cross compiling
(2.) we're using $stderr redirection to allow us to capture simulator error messages to $stdout for display at the run's conclusion]*

Notes:

- The upper case names are Ruby global constants that Ceedling builds
- "COLLECTION_" indicates that Ceedling did some work to assemble the list. For instance, expanding path globs, combining multiple path globs into a convenient summation, etc.
- At present, $stderr redirection is primarily used to capture errors from test fixtures so that they can be displayed at the conclusion of a test run. For instance, if a simulator detects a memory access violation or a divide by zero error, this notice might go unseen in all the output scrolling past in a terminal. $stderr redirection can be any of the following: :none, :auto, :win, :unix, :tcsh.
- The preprocessing tools can each be overridden with non-gcc equivalents. However, this is an advanced feature not yet documented and requires that the replacement toolchain conform to the same conventions used by gcc.

Ceedling Collection Used in Compilation	**Description**
COLLECTION_PATHS_TEST	All test paths
COLLECTION_PATHS_SOURCE	All source paths
COLLECTION_PATHS_INCLUDE	All include paths
COLLECTION_PATHS_SUPPORT	All test support paths
COLLECTION_PATHS_SOURCE_AND_INCLUDE	All source and include paths
COLLECTION_PATHS_SOURCE_INCLUDE_VENDOR	All source and include paths + applicable vendor paths (e.g. CException's source path if exceptions enabled)
COLLECTION_PATHS_TEST_TOOLCHAIN_INCLUDE	All test toolchain include paths
COLLECTION_PATHS_TEST_SUPPORT_SOURCE_INCLUDE	All test, source, and include paths
COLLECTION_PATHS_TEST_SUPPORT_SOURCE_INCLUDE_VENDOR	All test, source, include, and applicable vendor paths (e.g. Unity's source path plus CMock and CException's source paths if mocks and exceptions are enabled)
COLLECTION_PATHS_RELEASE_TOOLCHAIN_INCLUDE	All release toolchain include paths
COLLECTION_DEFINES_TEST_AND_VENDOR	All symbols specified in [:defines][:test] + symbols defined for enabled vendor tools – e.g. [:unity][:defines], [:cmock][:defines], and [:cexception][:defines]
COLLECTION_DEFINES_RELEASE_AND_VENDOR	All symbols specified in [:defines][:release] plus symbols defined by [:cexception][:defines] if exceptions are enabled

Notes:

- Other collections exist within Ceedling. However, they are only useful for advanced features not yet documented.
- Wherever multiple path lists are combined for use Ceedling prioritizes path groups as follows:
  test paths, support paths, source paths, include paths. This can be useful, for instance, in certain testing scenarios where we desire Ceedling or the compiler to find a stand-in header file before the actual source header file of the same name.

plugins: *Ceedling extensions*

Setting	Description	Default
base_path	Base path to search for plugin subdirectories	<none>
enabled	List of plugins to be used – a plugin's name is identical to the subdirectory that contains it (and the name of certain files within that subdirectory)	[] (empty)

Plugins can provide a variety of added functionality to Ceedling. In general use, it's assumed that at least one reporting plugin will be used to format test results. However, if no reporting plugins are specified, Ceedling will print to `$stdout` the (quite readable) raw test results from all test fixtures executed.

Example [:plugins] YAML blurb

```
:plugins:
  :base_path: project/tools/ceedling/plugins
  :enabled:
    - stdout_pretty_tests_report      #nice test results at your command line
    - our_custom_code_metrics_report  #maybe you needed line count and complexity metrics, so you
                                      #created a plugin to scan all your code and collect that info
```

Provided Plugins	Description
stdout_pretty_tests_report	Prints to `$stdout` a well-formatted list of ignored and failed tests, final test counts, and any extraneous output (e.g. printf statements or simulator memory errors) collected from executing the test fixtures.
stdout_ide_tests_report	Prints to `$stdout` simple test results formatted such that an IDE executing testing-related Rake tasks can recognize file paths and line numbers in your test messages. Thus, you can click a test result and jump to the failure (or ignored test) in your test file.
file_xml_tests_report	Creates an XML file of test results in the xUnit format (handy for Continuous Integration build servers or as input to other reporting tools). See `readme.txt` inside the `file_xml_tests_report` directory for configuration instructions.
bullseye	Adds additional Rake tasks to execute tests with the commercial code coverage tool provided by Bullseye. See `readme.txt` inside the `bullseye` directory for configuration instructions.

Advanced Topics (Coming)

Modifying Your Configuration without Modifying Your Project File: Option Files & User Files

Modifying your project file without modifying your project file

Debugging and/or printf()

When you gotta get your hands dirty...

Ceedling Plays Nice with Others – Using Ceedling for Tests Alongside Another Release Build Setup

You've got options.

Adding Handy Rake Tasks for Your Project (without Fancy Pants Custom Plugins)

Simple as snot.

Working with Non-Desktop Testing Environments

For those crazy platforms lacking command line simulators and for which cross-compiling on the desktop just ain't gonna get it done.

Creating Custom Plugins

Oh boy. This is going to take some explaining.