CMock Configuration Guide

C Standards, Compilers and Microcontrollers

The embedded software world contains its challenges. Compilers support different revisions of the C Standard. They ignore requirements in places, sometimes to make the language more usable in some special regard. Sometimes it's to simplify their support. Sometimes it's due to specific quirks of the microcontroller they are targeting. Simulators add another dimension to this menagerie.

CMock is designed to build mocks for almost anything that is targeted by a C compiler. It would be awesome if this could be done with zero configuration. While there are some targets that come close to this dream, it is sadly not universal. It is likely that you are going to need at least a couple of the configuration options described in this document.

CMock is configured in two ways. First, when mock files are created, there are configuration options that can be passed in during the creation process to help CMock interpret your needs and the needs of your target and compiler. Second, there are configuration options as `#defines` when the mocks and cmock.c are being compiled. Let's look at each.

How to Specify Code Generation Options

CMock is a Ruby script which contains a class called CMock. Therefore, when you want to specify configuration options while generating mocks, there are three ways to do it:

1. You can put your configuration options into a yaml file and then specify that this yaml file is to be used when you call the script. This is the most common method and is the method used by Ceedling.
2. You can directly add the options to the command line call when you ask CMock to generate the mock. This could make for some really long command line calls, but it works.
3. You can pull CMock into your own custom Ruby scripts (including rakefiles!) and pass all the arguments in as a hash. If you know Ruby already, this is a good route.

How to Specify Compile-Time Options

It doesn't matter if you're using a target-specific compiler and a simulator or a native compiler. In either case, you've got a couple choices for configuring the compile-time options:

1. Because these options are specified via C defines, you can pass most of these options to your compiler through command line compiler flags. Even if you're using an embedded target that forces you to use their overbearing IDE for all configuration, there will be a place somewhere in your project to configure defines for your compiler.
2. You can create a custom `cmock_config.h` configuration file (present in your toolchain's search paths). In this file, you will list definitions and macros specific to your

target. All you must do is tell CMock to always include that file by adding it to your list of includes.

Code Generation Options

This is a comprehensive list of code generation options. We do our best to autodetect as much as we can, and when left unspecified, the defaults represent the most common options, in the hope that they will work for you as well.

You'll find that many of these options are related to parsing your header files. CMock is ***not*** a full C parser. Even if it were, it would have to deal with the quirks of ***every*** compiler it encounters code for, which isn't realistic. You'll find that many of these options exist to help work around compiler-specific issues.

- `:attributes:` These are attributes that CMock should ignore for you for testing purposes. Custom compiler extensions and externs are handy things to put here. If your compiler is choking on some extended syntax, this is often a good place to look.
    - defaults: `['__ramfunc', '__irq', '__fiq', 'register', 'extern']`
    - note: this option will reinsert these attributes onto the mock's calls. If that isn't what you are looking for, check out `:strippables`.
- `:c_calling_conventions:` Similarly, CMock may need to understand which C calling conventions might show up in your codebase. If it encounters something it doesn't recognize, it's not going to mock it. We have the most common covered, but there are many compilers out there, and therefore many other options.
    - defaults: `['__stdcall', '__cdecl', '__fastcall']`
    - note: this option will reinsert these attributes onto the mock's calls. If that isn't what you are looking for, check out `:strippables`.
- `:callback_after_arg_check:` Tells the `:callback` plugin to do the normal argument checking before it calls the callback function by setting this to `true`. When `false`, the callback function is called instead of the argument verification.
    - default: `false`
- `:callback_include_count:` Tells the `:callback` plugin to include an extra parameter to specify the number of times the callback has been called. If set to false, the callback has the same interface as the mocked function. This can be handy when you're wanting to use callback as a stub.
    - default: `true`
- `:cexception_include:` Tell `:cexception` plugin where to find `CException.h`. You only need to define this if it's not in your build path already, which it usually will be for the purpose of your builds.
    - default: *`nil`*
- `:enforce_strict_ordering:` CMock always enforces the order that you call a particular function, so if you expect `GrabNabber(int size)` to be called three times,

it will verify that the sizes are in the order you specified. You might *also* want to make sure that all different functions are called in a particular order. If so, set this to true.
    - default: `false`
- `:framework:` Currently the only option is `:unity`. Eventually if we support other unit test frameworks (or if you write one for us), they'll get added here.
    - : default: `:unity`
- `:includes:` An array of additional include files which should be added to the mocks. This is useful for global types and definitions used in your project. There are more specific versions if you care WHERE in the mock files the includes get placed. You can define any or all of these options.
    - `:includes`
    - `:includes_h_pre_orig_header`
    - `:includes_h_post_orig_header`
    - `:includes_c_pre_header`
    - `:includes_c_post_header`
    - default: `nil` #for all 5 options
- `:memcmp_if_unknown`: C developers create a lot of types, either through typedef or preprocessor macros. CMock isn't going to automatically know what you were thinking all the time (though it tries its best). If it comes across a type it doesn't recognize, you have a choice on how you want it to handle it. It can either perform a raw memory comparison and report any differences, or it can fail with a meaningful message. Either way, this feature will only happen after all other mechanisms have failed (The thing encountered isn't a standard type. It isn't in the `:treat_as` list. It isn't in a custom `unity_helper`).
    - default: `true`
- `:mock_path`: The directory where you would like the mock files generated to be placed.
    - default: `mocks`
- `:mock_prefix`: The prefix to prepend to your mock files. For example, if it's "`Mock`", a file "`USART.h`" will get a mock called "`MockUSART.c`". This CAN be used with a suffix at the same time.
    - default: `Mock`
- `:mock_suffix`: The suffix to append to your mock files. For example, it it's "`_Mock`", a file "`USART.h`" will get a mock called "`USART_Mock.h`". This CAN be used with a prefix at the same time.
    - default: `""`
- `:plugins`: An array of which plugins to enable. ':`expect'` is always active. Also available currently:
    - `:ignore`
    - `:ignore_arg`
    - `:expect_any_args`
    - `:array`
    - `:cexception`

    - `:callback`
    - `:return_thru_ptr`
- `:strippables`: An array containing a list of items to remove from the header before deciding what should be mocked. This can be something simple like a compiler extension CMock wouldn't recognize, or could be a regex to reject certain function name patterns. This is a great way to get rid of compiler extensions when your test compiler doesn't support them. For example, use `:strippables: ['(?:functionName\s*\(+.*?\)+)']` to prevent a function `functionName` from being mocked. By default, it is ignoring all gcc attribute extensions.
    - default: `['(?:attribute\s*(+.*?)+)']`
- `:subdir`: This is a relative subdirectory for your mocks. Set this to e.g. "`sys`" in order to create a mock for `sys/types.h` in `(:mock_path)/sys/`.
    - default: `""`
- `:treat_as`: The `:treat_as` list is a shortcut for when you have created typedefs of standard types. Why create a custom unity helper for `UINT16` when the unity function `TEST_ASSERT_EQUAL_HEX16` will work just perfectly? Just add `'UINT16' => 'HEX16'` to your list (actually, don't. We already did that one for you). Maybe you have a type that is a pointer to an array of unsigned characters? No problem, just add `'UINT8_T*' => 'HEX8*'`
    - NOTE: unlike the other options, your specifications MERGE with the default list. Therefore, if you want to override something, you must reassign it to something else (or to *nil* if you don't want it)
    - default:
        - `'int': 'INT'`
        - `'char': 'INT8'`
        - `'short': 'INT16'`
        - `'long': 'INT'`
        - `'int8': 'INT8'`
        - `'int16': 'INT16'`
        - `'int32': 'INT'`
        - `'int8_t': 'INT8'`
        - `'int16_t': 'INT16'`
        - `'int32_t': 'INT'`
        - `'INT8_T': 'INT8'`
        - `'INT16_T': 'INT16'`
        - `'INT32_T': 'INT'`
        - `'bool': 'INT'`
        - `'bool_t': 'INT'`
        - `'BOOL': 'INT'`
        - `'BOOL_T': 'INT'`
        - `'unsigned int': 'HEX32'`
        - `'unsigned long': 'HEX32'`

        - `'uint32': 'HEX32'`
        - `'uint32_t': 'HEX32'`
        - `'UINT32': 'HEX32'`
        - `'UINT32_T': 'HEX32'`
        - `'void*': 'HEX8_ARRAY'`
        - `'unsigned short': 'HEX16'`
        - `'uint16': 'HEX16'`
        - `'uint16_t': 'HEX16'`
        - `'UINT16': 'HEX16'`
        - `'UINT16_T': 'HEX16'`
        - `'unsigned char': 'HEX8'`
        - `'uint8': 'HEX8'`
        - `'uint8_t': 'HEX8'`
        - `'UINT8': 'HEX8'`
        - `'UINT8_T': 'HEX8'`
        - `'char*': 'STRING'`
        - `'pCHAR': 'STRING'`
        - `'cstring': 'STRING'`
        - `'CSTRING': 'STRING'`
        - `'float': 'FLOAT'`
        - `'double': 'FLOAT'`
- `:treat_as_void`: We've seen "*fun*" legacy systems typedef `'void'` with a custom type, like `MY_VOID`. Add any instances of those to this list to help CMock understand how to deal with your code.
    - default: `[]`
- `:treat_externs`: This specifies how you want CMock to handle functions that have been marked as extern in the header file. Should it mock them?
    - `:include` will mock externed functions
    - `:exclude` will ignore externed functions (default).
- `:unity_helper_path`: If you have created a header with your own extensions to unity to handle your own types, you can set this argument to that path. CMock will then automagically pull in your helpers and use them. The only trick is that you make sure you follow the naming convention: `UNITY_TEST_ASSERT_EQUAL_YourType`. If it finds macros of the right shape that match that pattern, it'll use them.
    - default: `[]`
- `:verbosity`: How loud should CMock be?
    - 0 for errors only
    - 1 for errors and warnings
    - 2 for normal (default)
    - 3 for verbose
- `:weak`: When set this to some value, the generated mocks are defined as weak symbols using the configured format. This allows them to be overridden in particular tests.

    - Set to '`__attribute ((weak))`' for weak mocks when using GCC.
    - Set to any non-empty string for weak mocks when using IAR.
    - default: `""`
- `:when_no_prototypes`: When you give CMock a header file and ask it to create a mock out of it, it usually contains function prototypes (otherwise what was the point?). You can control what happens when this isn't true. You can set this to `:warn`, `:ignore`, or `:error`
    - default: `:warn`
- `:when_ptr`: You can customize how CMock deals with pointers (c strings result in string comparisons... we're talking about other pointers here). Your options are `:compare_ptr` to just verify the pointers are the same, `:compare_data` or `:smart` to verify that the data is the same. The :smart option tries to give you the best of both worlds. It will perform pointer comparisons if you specify a null pointer, but will compare data otherwise. Both `:compare_data` and `:smart` behaviors will change slightly based on if you have the array plugin enabled. By default, they compare a single element of what is being pointed to. So if you have a pointer to a struct called `ORGAN_T`, it will compare one `ORGAN_T` (whatever that is). For both, you can specify a number of elements to compare (how convenient!). You can even specify 0 elements, which tells it to go back to comparing pointers again!
    - default: `:smart`
- `:fail_on_unexpected_calls`: By default, CMock will fail a test if a mock is called without _Expect and _Ignore called first. While this forces test writers to be more explicit in their expectations, it can clutter tests with _Expect or _Ignore calls for functions which are not the focus of the test. While this is a good indicator that this module should be refactored, some users are not fans of the additional noise. Therefore, `:fail_on_unexpected_calls` can be set to false to force all mocks to start with the assumption that they are operating as _Ignore unless otherwise specified.
    - default: `true`
    - note: If this option is disabled, the mocked functions will return a default value (0) when called (and only if they have to return something of course).

Compiled Options

A number of `#defines` also exist for customizing the CMock experience. Feel free to pass these into your compiler or whatever is most convenient. CMock will otherwise do its best to guess what you want based on other settings, particularly Unity's settings.

- `CMOCK_MEM_STATIC` or `CMOCK_MEM_DYNAMIC` - Define one of these to determine if you want to dynamically add memory during tests as required from the heap. If static, you can control the total footprint of CMock. If dynamic, you will need to make sure you make some heap space available for CMock.

- `CMOCK_MEM_SIZE` - In static mode this is the total amount of memory you are allocating to CMock. In Dynamic mode this is the size of each *chunk* allocated at once (larger numbers grab more memory but require less mallocs).
- `CMOCK_MEM_ALIGN` - The way to align your data. Not everything is as flexible as x86 architecture, as most embedded designers know. This defaults to 2, meaning align to the closest 2^2 -> 4 bytes (32 bits). You can turn off alignment by setting this to 0, force alignment to the closest uint16 with 1 or even to the closest uint64 with 3.
- `CMOCK_MEM_PTR_AS_INT` - This is used internally to hold pointers... it needs to be big enough. On most processors a pointer is the same as an `unsigned long`... but maybe that's not true for yours?
- `CMOCK_MEM_INDEX_TYPE` - This needs to be something big enough to point anywhere in CMock's memory space... usually it's an `unsigned int`.

Happy Porting

The options, defines and macros in this guide should help you use CMock to target just about any C target we can imagine. If you run into a snag or two, don't be afraid of asking for help on the forums. We love a good challenge!