Unity 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.

Unity is designed to run on 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.

All of Unity's configuration options are `#defines`. Most of these are simple definitions. A couple are macros with arguments. They live inside the unity_internals.h header file. We don't necessarily recommend opening that file unless you really need to. That file is proof that a cross-platform library is challenging to build. From a more positive perspective, it is also proof that a great deal of complexity can be centralized primarily to one place in order to provide a more consistent and simple experience elsewhere.

Using These 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 these 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 `unity_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 define `UNITY_INCLUDE_CONFIG_H` and Unity will rely on `unity_config.h` for any further definitions it may need.

The Options

Integer Types

If you've been a C developer for long, you probably already know that C's concept of an integer varies from target to target. The C Standard has rules about the `int` matching the register size of the target microprocessor. It has rules about the `int` and how its size relates to other integer types. An `int` on one target might be 16 bits while on another target it might be 64. There are more specific types in compilers compliant with C99 or later, but that's certainly not every compiler you are likely to encounter. Therefore, Unity has a number of features for helping to adjust itself to match your required integer sizes. It starts off by trying to do it automatically.

`UNITY_EXCLUDE_STDINT_H`

The first thing that Unity does to guess your types is check `stdint.h`. This file includes defines like `UINT_MAX` that Unity can make use of to learn a lot about your system. It's possible you don't want it to do this (um. why not?) or (more likely) it's possible that your system doesn't support `stdint.h`. If that's the case, you're going to want to define this. That way, Unity will know to skip the inclusion of this file and you won't be left with a compiler error.

Example:

```
#define UNITY_EXCLUDE_STDINT_H
```

`UNITY_EXCLUDE_LIMITS_H`

The second attempt to guess your types is to check `limits.h`. Some compilers that don't support `stdint.h` could include `limits.h` instead. If you don't want Unity to check this file either, define this to make it skip the inclusion.

Example:

```
#define UNITY_EXCLUDE_LIMITS_H
```

`UNITY_EXCLUDE_SIZEOF`

The third and final attempt to guess your types is to use the `sizeof()` operator. Even if the first two options don't work, this one covers most cases. There *is* a rare compiler or two out there that doesn't support sizeof() in the preprocessing stage, though. For these, you have the ability to disable this feature as well.

Example:

```
#define UNITY_EXCLUDE_SIZEOF
```

If you've disabled all of the automatic options above, you're going to have to do the configuration yourself. Don't worry. Even this isn't too bad… there are just a handful of defines that you are going to specify if you don't like the defaults.

`UNITY_INT_WIDTH`

Define this to be the number of bits an `int` takes up on your system. The default, if not autodetected, is 32 bits.

Example:

```
#define UNITY_INT_WIDTH 16
```

`UNITY_LONG_WIDTH`

Define this to be the number of bits a `long` takes up on your system. The default, if not autodetected, is 32 bits. This is used to figure out what kind of 64-bit support your system can handle. Does it need to specify a `long` or a `long long` to get a 64-bit value. On 16-bit systems, this option is going to be ignored.

Example:

```
#define UNITY_LONG_WIDTH 16
```

`UNITY_POINTER_WIDTH`

Define this to be the number of bits a pointer takes up on your system. The default, if not autodetected, is 32-bits. If you're getting ugly compiler warnings about casting from pointers, this is the one to look at.

Example:

```
#define UNITY_POINTER_WIDTH 64
```

`UNITY_INCLUDE_64`

Unity will automatically include 64-bit support if it auto-detects it, or if your `int`, `long`, or pointer widths are greater than 32-bits. Define this to enable 64-bit support if none of the other options already did it for you. There can be a significant size and speed impact to enabling 64-bit support on small targets, so don't define it if you don't need it.

Example:

```
#define UNITY_INCLUDE_64
```

Floating Point Types

In the embedded world, it's not uncommon for targets to have no support for floating point operations at all or to have support that is limited to only single precision. We are able to guess integer sizes on the fly because integers are always available in at least one size. Floating point, on the other hand, is sometimes not available at all. Trying to include `float.h` on these platforms would result in an error. This leaves manual configuration as the only option.

```
UNITY_INCLUDE_FLOAT
UNITY_EXCLUDE_FLOAT
UNITY_INCLUDE_DOUBLE
UNITY_EXCLUDE_DOUBLE
```

By default, Unity guesses that you will want single precision floating point support, but not double precision. It's easy to change either of these using the include and exclude options here. You may include neither, either, or both, as suits your needs. For features that are enabled, the following floating point options also become available.

Example:

```
//what manner of strange processor is this?
#define UNITY_EXCLUDE_FLOAT
#define UNITY_INCLUDE_DOUBLE
```

```
UNITY_FLOAT_VERBOSE
UNITY_DOUBLE_VERBOSE
```

Unity aims for as small of a footprint as possible and avoids most standard library calls (some embedded platforms don't have a standard library!). Because of this, its routines for printing integer values are minimalist and hand-coded. To keep Unity universal, though, we chose to *not* develop our own floating point print routines. Instead, the display of floating point values during a failure are optional. By default, Unity will not print the actual results of floating point assertion failure. So a failed assertion will produce a message like `"Values Not Within Delta"`. If you would like verbose failure messages for floating point assertions, use these options to give more explicit failure messages (e.g. `"Expected 4.56 Was 4.68"`). Note that this feature requires the use of `sprintf` so might not be desirable in all cases.

Example:

```
#define UNITY_DOUBLE_VERBOSE
```

`UNITY_FLOAT_TYPE`

If enabled, Unity assumes you want your `FLOAT` asserts to compare standard C floats. If your compiler supports a specialty floating point type, you can always override this behavior by using this definition.

Example:

```
#define UNITY_FLOAT_TYPE float16_t
```

`UNITY_DOUBLE_TYPE`

If enabled, Unity assumes you want your `DOUBLE` asserts to compare standard C doubles. If you would like to change this, you can specify something else by using this option. For example, defining `UNITY_DOUBLE_TYPE` to `long double` could enable gargantuan floating point types on your 64-bit processor instead of the standard `double`.

Example:

```
#define UNITY_DOUBLE_TYPE long double
```

`UNITY_FLOAT_PRECISION`
`UNITY_DOUBLE_PRECISION`

If you look up `UNITY_ASSERT_EQUAL_FLOAT` and `UNITY_ASSERT_EQUAL_DOUBLE` as documented in the big daddy Unity Assertion Guide, you will learn that they are not really asserting that two values are equal but rather that two values are “close enough” to equal. “Close enough” is controlled by these precision configuration options. If you are working with 32-bit floats and/or 64-bit doubles (the normal on most processors), you should have no need to change these options. They are both set to give you approximately 1 significant bit in either direction. The float precision is 0.00001 while the double is 10^{-12}. For further details on how this works, see the appendix of the Unity Assertion Guide.

Example:

```
#define UNITY_FLOAT_PRECISION 0.001f
```

Toolset Customization

In addition to the options listed above, there are a number of other options which will come in handy to customize Unity's behavior for your specific toolchain. It is possible that you may not need to touch any of these… but certain platforms, particularly those running in simulators, may need to jump through extra hoops to operate properly. These macros will help in those situations.

```
UNITY_OUTPUT_CHAR(a)
UNITY_OUTPUT_FLUSH()
UNITY_OUTPUT_START()
UNITY_OUTPUT_COMPLETE()
```

By default, Unity prints its results to `stdout` as it runs. This works perfectly fine in most situations where you are using a native compiler for testing. It works on some simulators as well so long as they have `stdout` routed back to the command line. There are times, however, where the simulator will lack support for dumping results or you will want to route results elsewhere for other reasons. In these cases, you should define the `UNITY_OUTPUT_CHAR` macro. This macro accepts a single character at a time (as an `int`, since this is the parameter type of the standard C `putchar` function most commonly used). You may replace this with whatever function call you like.

Example:

Say you are forced to run your test suite on an embedded processor with no `stdout` option. You decide to route your test result output to a custom serial `RS232_putc()` function you wrote like thus:

```
#define UNITY_OUTPUT_CHAR(a) RS232_putc(a)
#define UNITY_OUTPUT_START() RS232_config(115200,1,8,0)
#define UNITY_OUTPUT_FLUSH() RS232_flush()
#define UNITY_OUTPUT_COMPLETE() RS232_close()
```

`UNITY_SUPPORT_WEAK`

For some targets, Unity can make the otherwise required `setUp()` and `tearDown()` functions optional. This is a nice convenience for test writers since `setUp` and `tearDown` don't often actually *do* anything. If you're using gcc or clang, this option is automatically defined for you. Other compilers can also support this behavior, if they support a C feature called weak functions. A weak function is a function that is compiled into your executable *unless* a non-weak version of the same function is defined elsewhere. If a non-weak version is found, the weak version is ignored as if it never existed. If your compiler supports this feature, you can let Unity know by defining `UNITY_SUPPORT_WEAK` as the function attributes that would need to be applied to identify a function as weak. If your compiler lacks support for weak functions, you will always need to define `setUp` and `tearDown` functions (though they can be and often will be just empty). The most common options for this feature are:

Example:

```
#define UNITY_SUPPORT_WEAK weak
#define UNITY_SUPPORT_WEAK __attribute__((weak))
```

`UNITY_PTR_ATTRIBUTE`

Some compilers require a custom attribute to be assigned to pointers, like `near` or `far`. In these cases, you can give Unity a safe default for these by defining this option with the attribute you would like.

Example:

```
#define UNITY_PTR_ATTRIBUTE __attribute__((far))
#define UNITY_PTR_ATTRIBUTE near
```

Getting Into The Guts

There will be cases where the options above aren't quite going to get everything perfect. They are likely sufficient for any situation where you are compiling and executing your tests with a native toolchain (e.g. clang on Mac). These options may even get you through the majority of cases encountered in working with a target simulator run from your local command line. But especially if you must run your test suite on your target hardware, your Unity configuration will require special help. This special help will usually reside in one of two places: the `main()` function or the `RUN_TEST` macro. Let's look at how these work.

`main()`

Each test module is compiled and run on its own, separate from the other test files in your project. Each test file, therefore, has a `main` function. This `main` function will need to contain whatever code is necessary to initialize your system to a workable state. This is particularly true for situations where you must set up a memory map or initialize a communication channel for the output of your test results.

A simple main function looks something like this:

```
int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst);
    RUN_TEST(test_TheSecond);
    RUN_TEST(test_TheThird);
    return UNITY_END();
}
```

You can see that our main function doesn't bother taking any arguments. For our most barebones case, we'll never have arguments because we just run all the tests each time. Instead, we start by calling `UNITY_BEGIN`. We run each test (in whatever order we wish). Finally, we call `UNITY_END`, returning its return value (which is the total number of failures).

It should be easy to see that you can add code before any test cases are run or after all the test cases have completed. This allows you to do any needed system-wide setup or teardown that might be required for your special circumstances.

RUN_TEST

The `RUN_TEST` macro is called with each test case function. Its job is to perform whatever setup and teardown is necessary for executing a single test case function. This includes catching failures, calling the test module's `setUp()` and `tearDown()` functions, and calling `UnityConcludeTest()`. If using CMock or test coverage, there will be additional stubs in use here. A simple minimalist RUN_TEST macro looks something like this:

```
#define RUN_TEST(testfunc) \
    UNITY_NEW_TEST(#testfunc) \
    if (TEST_PROTECT()) { \
        setUp(); \
        testfunc(); \
    } \
    if (TEST_PROTECT() && (!TEST_IS_IGNORED)) \
        tearDown(); \
    UnityConcludeTest();
```

So that's quite a macro, huh? It gives you a glimpse of what kind of stuff Unity has to deal with for every single test case. For each test case, we declare that it is a new test. Then we run `setUp` and our test function. These are run within a `TEST_PROTECT` block, the function of which is to handle failures that occur during the test. Then, assuming our test is still running and hasn't been ignored, we run `tearDown`. No matter what, our last step is to conclude this test before moving on to the next.

Let's say you need to add a call to `fsync` to force all of your output data to flush to a file after each test. You could easily insert this after your `UnityConcludeTest` call. Maybe you want to write an xml tag before and after each result set. Again, you could do this by adding lines to this macro. Updates to this macro are for the occasions when you need an action before or after every single test case throughout your entire suite of tests.

Happy Porting

The defines and macros in this guide should help you port Unity to 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!