Using Ceedling

Created by Bryce Deary, last modified just a moment ago

Test embedded firmware often involves extended hardware support to test code correctness. Just getting the code to compile is just a first step. An independent unit test framework provides one method of verifying the correctness of code in the context of the programming language being used. In this case C. Constructing the code in such a way as to reduce dependencies on actual hardware further increases the ability to unit test the core logic.

This page is an extension and expansion on my notes: Ceedling On Windows

Modules

The idea of a module is very important in unit testing. A module is simply a collection of at least three files with the same root name and correspond to the traditional C module files of header, translation unit, and test file. If a module becomes too large over time, it is highly encouraged to break it up into related sub-pieces, specifically breaking out stand alone objects or structures that may share some higher order configuration and may ultimately be grouped together into the original Large module. These sub_modules can then be unit tested in isolation from the larger module and any environmental setup needed to test and be limited in scope. Since all the sub parts are tested separately, the unit test of the root module need only test the thin layer of logic within the root module and the sub_modules can be mocked out to simplify testing and remove lower level dependencies.

For Example, in our project we have a root object called sierra and a header file called sierra.h . In the sierra.h header there is a structure declaration for an object of type Sierra that is used in most sierra related function APIs. The actual Sierra object is quite large and the functions that manipulate it continue to grow. To keep the file size reasonable and to make testing easier, related functions that are still methods of the sierra object have been broken out to sub modules with any sub module declaration being declared in separate sub_headers that are themselves included in the original root header.

sub headers in main header file sierra.h

```
// Any definitions or configuration needed by any sub_module can go here (or in the project.h file)
#define SOME_SIERRA_SPECIAL (42)

// Some additional parts from broken out translation units
#include "sierra_cmu_central.h"
#include "sierra_adc.h"
#include "sierra_dac.h"
#include "sierra_lsadc.h"
#include "sierra_clock_sync.h"

typedef struct Sierra {  // this type def is probable a forward ref in other headers
    Converter base; // Extend Converter (also ABC)
// Area of virtual jump table of Sierra related functions
    Status (*inlCalibrationStart)(struct Sierra*, int isBackground);
    Status (*inlCalibrationStop)(struct Sierra*, int isBackground);
//      :
//      :
// sierra specific vars
    MacroBlkState rx_state;         // setChanState ADC
    MacroBlkState tx_state;         // setChanState DAC
//      :
//      :
// The root object may contain types declared in the sub_headers above
    ClockSyncConfig adc_sync;  // where ClockSyncConfig id declared in sierra_clock_sync.h
    ClockSyncConfig dac_sync;

} Sierra;
```

Named Projects

The unittest folder is a collection of unittest projects arranged by named_devices, where each named device corresponds to a project folder in source/project. Each named device is uniquely configured by a project.h file (see below). The unittest folder also contains folders for sub_modules and common_tests which are tests that exercise code that is used across more than one named_device but that may behave differently based on the project.h file. To this end, when testing in a Named Project area, running a test:all will run tests across all three areas specifically configured for the named project.
For example, the Sierra object and its related sub_objects are part of both the Darwin family of projects (EMA, Baldwin, ...) and the Eiger family of projects (Bishorn, Matterhorn, ...) When you run unit tests in the Matterhorn named project area, all the sierra related translation units are compiled with the Matterhorn project.h file and so any special Eiger configurations will be active. This is also true of the common tests but generally the common modules should be decoupled from projects as much as possible.

The Project file.

Our firmware builds rely on a project configuration file (configuration header) called project.h . In our design, the project.h file contains any special configuration macros and then includes the project specific platform header which in turn includes all the headers need by the platform including the headers derived from the various csr (control status registers) files that contain the actual hardware mapping of register and field names. Because the project.h file is super important in doing any special configuration tweaks, it is important for any modules that are mocked to have access to the configurations. To do this, the project.yml file in the test folder has a special line under the :cmock: key called :include_c_pre_header: which we set to <project.h>. The project.yml file contains the correct path to the specific project.h file.

Test Files are Weird

On first inspection the test files look like normal C translation units but the ceedling framework uses them to direct the construction of the actual test runner. Ceedling is written in ruby and it uses a program called rake instead of make. The Ceedling program reads the explicitly included header names in the test file and uses them to create a rake file. Any module header found will cause the corresponding module translation unit to be compiled and linked into the test runner. If the module header is included with the substring mock_ pre_pended to the real name, then ceedling will ignore the module translation unit and instead create mock functions from any function descriptions (prototypes) found in the original header file.

This automatic test runner generation feature means that the include files listed in the test file must include modules or mocks that satisfy dependencies of of other included modules. This is where mocks come in very handy since mocks don't have additional dependencies themselves.

Split modules or modules where the header and translation unit have different names need special notation. If you need to link in a file that does not have a matching header you use the

TEST_FILE("file_name.c")

notation in the test file to have that file compiled and linked into your test.

If you have an alternate translation unit that implements an interface in a header with different name you may need to directly specify files by name in the project.yml, First by removing their paths from the paths area adding them back in a separate files area for example emaB0_jesd.c impliments the interface in ema_jesd.c so we hant to test the emaB0_jesd.c code not the original.

project.yml changes

```
:paths:
  :source:
#-> - +:../../../source/link/jesd/electrama  # <-- remove this line to prevent finding code file
  :include:
    - +:../../../source/link/jesd/electrama  # add this line to only find the header

:files: # add a files area
  :source:
    - -:../../../source/link/jesd/electrama/ema_jesd.c      # exclude this file
    - +:../../../source/link/jesd/electrama/emaB0_jesd.c    # add alternate translation
```

Use Ceedling to create new Modules

If you need to create a new module, it is best to allow ceedling to create it since it will automatically create the three files and populate them with a basic Jariet template. The create command requires you to specify where you want the files to be located using the module:create command.

ceedling module create command

```
ceedling module:create[<module name>]
```

Our configuration sets module name relative to the source folder, so to place a module foo in the common folder the name would be common/foo and if you wanted to place the file bar in the electramaapi folder the name would be api/sierraapi/darwinapi/electramaapi/bar.

Example of Creation

```
...\dev\fw\trunk\unittest\named_devices\electrama>ceedling module:create[common/foo]
```

```
...\dev\fw\trunk\unittest\named_devices\electrama>ruby ..\..\vendor\ceedling\bin\ceedling module:create[common/foo]
File ../../../source/common/foo.c created
File ../../../source/common/foo.h created
File electrama_tests/common/test_foo.c created
Generate Complete

...\dev\fw\trunk\unittest\named_devices\electrama>ceedling module:create[api/sierraapi/darwinapi/electramaapi/bar]

...\dev\fw\trunk\unittest\named_devices\electrama>ruby ..\..\vendor\ceedling\bin\ceedling module:create[api/sierraapi/darwinapi/electramaapi/bar]
File ../../../source/api/sierraapi/darwinapi/electramaapi/bar.c created
File ../../../source/api/sierraapi/darwinapi/electramaapi/bar.h created
File electrama_tests/api/sierraapi/darwinapi/electramaapi/test_bar.c created
Generate Complete
```

With the module header and translation unit files created at the path relative to the source folder, the test file will be created in the named_device folder in the unittest tree and should be moved to the appropriate test folder if they are not actually specific to that named device.
You can wait to move the file until after you get it running. This allows you to do other work on other named projects that don't yet include the new module. Remember unit tests are always done in the context of some named project.

If you make a mistake you can use the ceedling module:destroy[<module name>] to remove the created files, using the same path name used to create it.

It is best to use a unique name for a module, So avoid using a unique folder name and then some generic family name for the translation unit. It will make later editing much less confusing. (Note that the project module is the exception to this rule but with the project.h file required to be project.h in the specific project folder in the source/project tree.)

Use Test Driven Development (TDD)

Once a module is created using the module:create command it is ready to run. (Even though it won't do anything yet). The basic idea of TDD is write the test, then write the code. So if you are designing new interfaces or functions write the prototypes in the header, write a simple calling test and run the test. It will fail of course because there is not function but you now have a caller for your function. Now write the function. You don't need to make it work yet, just write a shell in the translation unit such that your test will compile and run, with a reported failure. Next add some behavior to your function to make your first test pass.
You are now at a stable point in your process. Continue iterating by adding new tests or refine existing tests, prove to yourself that the tests fail, then add logic to your code to make them pass.
Don't check in code if the tests don't compile, and if you have failing tests, add notes to the check-in message to that effect. Don't write one complex test, write many simple well named tests. There is no penalty for test size or number and it is always Ok to copy tests to make new tests. At come point you can go back and refactor the copied code if you want to make things more general. This is especially true of code needed to setup your environment or to set to a needed state. Be clear, be verbose, use long descriptive names, you should be able to tell by reading the tests what the code is suppose to do.

Of course it is much more complicated than that.

Obviously in embedded code we are primarily dealing with user commands to hardware, reading status of hardware, and doing some intermediate processing. In our system the firmware does not do much un-commanded regulation or control. Also C is a compiled language and is not reflective like python or Java so tests can't look at the state of the system unless we specifically capture it for view. One of the most important symbols in our design is the pointer to the Platform object. Through this pointer we can construct our test environment.

STATIC functions

Scoping rules in C allow us to write functions and define variables that are file scope static which prevents those symbols from being accessed outside the translation unit. For the purpose of unit testing, we have a macro of all caps STATIC that should be used instead of the keyword static. This allows us to export our normally hidden symbols only during tests. These hidden symbols must be declared external in the actual test files to access them.

More to Come.....

Step-by-step guides

Running the tests

1. As stated above, Ceedling tests are run from a named_device project folder in the unittest folder tree.
   do a cd to unittest\named_devices\<project>
2. From the command line type:
   ceedling
3. the bare ceedling command defaults to ceedling test:all and causes the test runner to search all the folders declared as test folders in the project.yml file for translation units whose name starts with the substring test_ .
4. To run just a single test use the module name, so to test the module sierra_adc you would type ceedling test:sierra_adc . This will cause ceedling to find the file test_sierra_adc.c then compile, link and run the resulting test.

Create New Module

1. From the command line with directory set to one of the named_devices\<project> folders
2. ceedling module:create[<source relative name>] where
   source relative name is a path below the source folder. In our tree, to create a new module in the common folder called foo:
   ceedling module:create[common/foo]
   See (Use Ceedling to create new Modules) above.

Related articles

Using Ceedling

pytest usage

c unittest kb-how-to-article