C4 Planner Unit Tests

Framework

Tests for C4 Planner are written using the ArduinoUnit framework. Follow the instructions under “Getting Started” at https://github.com/mmurdoch/arduinounit/blob/master/readme.md to import this framework.

Writing Unit Tests

If you're already familiar with how unit testing frameworks behave in general, you can probably skip this section. This is a primer on how to write effective unit tests.

Some things to keep in mind:

- Think about your upper and lower bounds and be sure to test outside them
- Don't worry about whether the scenario is realistic
- We're only testing the behavior of a function based on the input provided; we're not testing that it works properly with other systems
- If a test fails, that may be okay; just determine whether the scenario that failed is intended to be caught elsewhere. For example, in an embedded system we might assume correct input because we know another microcontroller is handling the verification and we'd rather have a quick response. Just ensure it's handled somewhere.

How to approach writing unit tests

1. Select a function you would like to test and list out the inputs, output, and dependencies (other functions it calls, objects and variables it uses or updates, etc.)
2. Determine what the expected values are for each of the items listed in the first step and write a list of the values you would like to test. Some examples include:
    - lowest expected value
    - highest expected value
    - zero
    - a value below the lowest expected
    - a value above the highest expected
    - a combination of values that could cause overflow
    - different types of strings (ASCII vs. Unicode)
    - null
    - empty objects
3. Start with tests that ensure it behaves as expected with the input that is expected (the "happy path" scenario)
4. Write one test for each change in expected behavior. For example, if you have a function that takes two variables as input, test the behavior with both variables within

expected ranges, then start testing unexpected values with variable a while variable b remains in expected ranges, then b in unexpected ranges with a in expected ranges, and finally both outside expected ranges.
5. Run the tests as you go to make sure they're working as expected. **Note:** "as expected" means the behavior is what you would expect it to be given the input provided. If the input is bad, you might expect a garbage output, but that output should be predictable and should not cause a crash.
6. If a test fails when you expected it to pass, make sure it was written correctly, but don't assume the code functions properly just because it seems to work without problems; that's what unit tests are designed to catch.

Troubleshooting

Tests Print Out Results Over and Over

This indicates there was something wrong with the tests that caused some kind of error or exception. Usually this is because data that is being accessed by the function isn't initialized.

How to Resolve

1. Comment and uncomment blocks of code to determine which function call is causing the error. If all tests are working correctly, the tests will run once and then stop (even though they're called from the loop).
2. Once you've isolated the function call, follow the logic and determine if any objects or variables are being called that are global or that were passed in. These are the most likely culprits.
3. Check the setup routine for the *.ino file being tested and see if there are any general setup routines being called that may be a dependency for your function.
4. In the unit test, be sure to initialize the values for the variables, structs, and objects being used by the function you're testing.
5. Re-run the test, and if it's still not working repeat steps 3 and 4 until the problem is found.

Example unit tests

This code shows an example of a couple of unit tests on one of the Elcano_C4_Planner functions. More generic examples can be found on the ArduinoUnit page.

Note that these examples don't include tests where the function has a return value (yet). If you need to test a return value, create a variable to store the return value before you call the function. For example:

```
int expected = 0;
int actual = foo(bar);
assertEqual(actual, expected);
```

```
/* ConstructNetwork tests

  Inputs: junction *Map, int MapPoints
  Expected values for *Map: array of junctions
  Expected values for MapPoints: MAP_POINTS

  junction values:
  long east_mm, long north_mm, int destination[4], long Distance[4]

  destination is the index of the graph nodes this junction has an edge to
  Distance is the distance multiplier for each destination node

*/

// Happy path test with small map
// Verifies that the distances are correctly updated in the map passed to
// the function.
test(ConstructNetwork_small)
{
  int m = 2;

  // input junction
  junction actual[2] = {
    0,  0, 1 ,  END,  END,  END,  1, 1, 1, 1,  // 0
    3,  4, 0 ,  END,  END,  END,  1, 1, 1, 1,  // 1

  };

  // expected output
  junction expected[2] = {
    0,  0, 1 ,  END,  END,  END,  5, 1, 1, 1,  // 0
    3,  4, 0 ,  END,  END,  END,  5, 1, 1, 1,  // 1

  };

  // Call the function being tested
  ConstructNetwork(actual, m);

  // Verify the data in the array passed to the function is updated
  // Because we don't have a print overload for junction, we need to
  // make assertations on individual pieces of data within the struct.
  // The same applies to arrays.
  for(int i = 0; i < 2; i++)
  {
    assertEqual(actual[i].east_mm, expected[i].east_mm);
    assertEqual(actual[i].north_mm, expected[i].north_mm);
    for(int j = 0; j < 4; j++){
      assertEqual(actual[i].destination[j], expected[i].destination[j]);
      assertEqual(actual[i].Distance[j], expected[i].Distance[j]);
    }
```

```
  }
}

// Happy path test with large map
// Verifies that the distances are correctly updated in the map passed to
// the function.
test(ConstructNetwork_large)
{
  int m = 100;
  // input junction
  junction actual[100] = {
    0,  0, 1 ,   2 ,   3 ,   4 ,  1, 1, 1, 1,  // 0
    3,  4, 0 ,  END,  END,  END,  1, 1, 1, 1,  // 1
    3,  4, 0 ,  END,  END,  END,  1, 1, 1, 1,  // 2
    0,  0, 0 ,  END,  END,  END,  1, 1, 1, 1,  // 3
    3,  4, 0 ,  END,  END,  END,  1, 1, 1, 1,  // 4
  };

  // expected output
  junction expected[100] = {
    0,  0, 1 ,   2 ,   3 ,   4 ,  5, 5, 0, 5,  // 0
    3,  4, 0 ,  END,  END,  END,  5, 1, 1, 1,  // 1
    3,  4, 0 ,  END,  END,  END,  5, 1, 1, 1,  // 2
    0,  0, 0 ,  END,  END,  END,  0, 1, 1, 1,  // 3
    3,  4, 0 ,  END,  END,  END,  5, 1, 1, 1,  // 4
  };

  // Call the function being tested
  ConstructNetwork(actual, m);

  // Verify the data in the array passed to the function is updated
  // Because we don't have a print overload for junction, we need to
  // make assertations on individual pieces of data within the struct.
  // The same applies to arrays.
  for(int i = 0; i < m; i++)
  {
    assertEqual(actual[i].east_mm, expected[i].east_mm);
    assertEqual(actual[i].north_mm, expected[i].north_mm);
    for(int j = 0; j < 4; j++){
      assertEqual(actual[i].destination[j], expected[i].destination[j]);
      assertEqual(actual[i].Distance[j], expected[i].Distance[j]);
    }
  }
}
```