White-Box Testing for Solar Tracking Device Documentation
Created by Sorin Liviu Jurj and Raul Rotar

Software testing is one of the most important steps in development of any modern product. Software testing provides us with information about the quality of the software product and also points out bugs that might be present in the software. Inadequate testing or failure to fix problems noticed in test results has been the reason for a lot of accidents and wasted resources. There are different forms of testing, some of the most widely used include:

- White Box testing.
- Black Box testing.
- Visual Testing.

White-box testing verifies the internal structures of a program. This means that in white box testing, we are more interested in the workings of the source code than the functionality provided to the end user. White-box testing is very efficient in finding hidden errors and optimizing code base, but one disadvantage is that it does not help us fine unimplemented or missing issues. White-box testing can be used in Unit testing, Integration testing and Regression testing. Some important types of white-box testing include:

- Control Flow Testing
- Branch Testing
- Basis Path Testing
- Data Flow Testing
- Loop Testing

In our project, We will be applying White-box testing to analyze the program structure and attempt to find any bugs or programming errors.

Some of the common Errors encountered during software programming include:

- Functionality Errors
- Communication Errors.
- Syntax Errors.
- Error handling errors.
- Control Flow and Calculation Errors.

We will not be exploring syntax errors because we are working with a compiled language (C++) and the programmer is informed of all the syntax errors at compile time. We will pay more attention to Communication, Control Flow and Error handling errors.

Communication errors are errors in communication from software to end-user or between within the software. In our project we will be pay particular attention to communication errors within the software because we have 2 communication links, the first is between the server and secondary microcontroller (MQTT over TCP/IP), and the second is between the primary (Arduino Uno) and Secondary microcontroller (ESP8266).
Error handling errors arise when there is no proper structure in place to handle unexpected values or actions in the program. In our project Such unexpected actions could be hardware faults, circuit noise e.t.c.

Control flow of a software decides what it will do next or what action it will take under certain conditions. Errors in control flow can lead to buffer overflow, unpredictable system states and a host of other problems.

In our Firmware we have identified the possible error points in the system using our internal knowledge of the firmware structure and tested it using a unit testing library for the Arduino Platform, AIUnit.
AIUnit is a unit testing framework that draws inspiration from the Google Test and Arduino Unit APIs. You can read more about AIUnit here https://github.com/bxparks/AUnit

Implementation

Here is a brief description of the implementation of the testing techniques on the ESP8266 firmware, which acts like a middle man between the arduino uno and the mqtt server.

The first tests (displayed in fig. 1), test to make sure that the ssid, password and server addresses provided are valid. This test checks if the values are int the alphanumeric range.
A wrong ssid, password, or mqtt server address by the user will lead to connection failure as the device will not be able to connect to the wifi or the mqtt broker address.

```
/*
 * This test is configured to run only once and
 * ensures that our ssid string contains only valid characters.
 */
test(ssidTest)
{
  for(uint8_t i=0; i<sizeof(ssid); i++)
  {
    assertMoreOrEqual(ssid[i], ' ');
  }
}


/*
 * This test is configured to run only once and
 * ensures that our password string contains only valid characters.
 */
test(passwordTest)
{
  for(uint8_t i=0; i<sizeof(password); i++)
  {
    assertMoreOrEqual(password[i], ' ');
  }
}


/*
 * This test is configured to run only once and
 * ensures that our mqtt address string is in the right format,
 */
test(mqttAddrTest)
{
  for(uint8_t i=0; i<sizeof(mqtt_server); i++)
  {
    assertMoreOrEqual(mqtt_server[i], '.');
    assertLessOrEqual(mqtt_server[i], '9');
  }
}
```

Fig.1

Another possible error point which we have tested is the mqtt Topic array. This is the topic to which the mqtt client subscribes in order to receive messages from the mqtt

broker. This topic is unique for each instance of the device, therefore we have to generate them dynamically using the Unique Identification number of the microcontroller. We read this unique identification number and use it to generate a string which is then our Incoming messages topic, is there is an error in this topic we will not be able to receive any incoming messages from the broker and our device will fail. Therefore knowing the expected length of the Unique identification number, we run a test which which will ensure that the Incoming messages topic is not longer than this length, and that all the characters are alphanumeric characters.

```
/*
 * This test a test to check the inTopic array
 * which is dynamically generated using strings
 * which are notorious for memory problems. In our
 * test we check that all the characters are valid
 * characters and also that the length of the array is
 * less than 10
 */
test(InTopicTest)
{
  uint8_t b = sizeof(InTopic);
  assertLess(b, 10);    // assert that the size of the array is less than 10

  for(uint8_t i=0; i<sizeof(InTopic); i++)
  {
    assertMoreOrEqual(InTopic[i], ' ');
  }
}
```

Fig.2

The most error prone part of this firmware is the Incoming data values. These are the values that are received from the mqtt broker and forwarded to the arduino uno. The values could be corrupted as a result of communication errors or unexpected user input. If we successfully forward a wrong value from to the arduino uno, this can lead to unexpected or random behaviour of the stepper motors. Therefore we test to make sure that the values received are within a specified range.

```
/-
 * This test checks if the value of the sensor data received is in the
 * reasonable range. If this value is outside the range 0 - 1024, we can
 * concluse that the data was corrupted during transmission or a memmory
 * problem has occured.
 * This is a continous test and will be run over and over again.
 */
testing(sensorDataTest)
{
  assertEqual(isDigit(ltsensor), 1);      // makes sure that its a digit and not NAN
  assertEqual(isDigit(rtsensor), 1);
  assertEqual(isDigit(rdsensor), 1);
  assertEqual(isDigit(ldsensor), 1);

  assertMoreOrEqual(ltsensor, 0);         // makes sure it is an unsigned integer, i.e greater than zero
  assertMoreOrEqual(rtsensor, 0);
  assertMoreOrEqual(rdsensor, 0);
  assertMoreOrEqual(ldsensor, 0);

  assertLessOrEqual(ltsensor, 1023);      // makes sure that it is less than 1023
  assertLessOrEqual(rtsensor, 1023);
  assertLessOrEqual(rdsensor, 1023);
  assertLessOrEqual(ldsensor, 1023);
}
```

Fig.3

It is not uncommon for internet communication to drop unexpectedly, or for MQTT clients to disconnect from the broker. In a situation like this the device needs to be able to detect that the connection has been lost and attempt to reconnect. In this case we implement a control flow test, shown in Fig.4, to check the mqtt connection and attempt to reconnect if connection has been lost.

```
/*
 * Sometimes connection to the MQTT Server can break for many reasons. conditional
 * statement will attempt to reconnect to the server if the connectin is broker.
 * But if the reconnection fails, we then restart the system.
 */
if (!client.connected())
{
   MQTT_Connect();

   // we add a 1 second delay to the reconnect loop so that we do not bang the server
   // with too many requests.
   delay(1000);
}
```

Fig.4

Different routers allow connections at different speeds, some take a longer time to authenticate before internet access is allowed, therefore it is necessary to test that the device has been successfully connected to the router before we proceed with execution of the program. Failure to do this will lead to data loss as the device may attempt to send data when a connection has not been established.

```
WiFi.begin(ssid, password);

while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
  ESP.wdtFeed();
}
```

Fig. 5

Uno Code

The main test on the firmware for the arduino uno code is to ensure that the data received via serial connection is valid. Serial communication, is prone to different types of errors especially at high speeds (e.g 115200 baud rate). If we fail to verify the integrity of the received data, we may feed wrong values to the stepper motor controls which can lead to unpredictable or unintended movements. Since these are unsigned analog values, our main test is to ensure that they are not above 1024. For a more safety critical system we can apply checksums to further validate the integrity of the firmware.

You can read more about check sums here; https://en.wikipedia.org/wiki/Checksum

```
//-----------------------------------------
/*
 * We will run a continious test on the LTSensor,
 * LDSensor, RTSensor, and RDSensor data values to
 * make that they are within the specified range.
 * This can be a source of error if the values are
 * outside the specified range, as this can lead to
 * errora and unpredictable movement of the stepper
 * motors.
 */
testing(sensorData)
{
  assertLessOrEqual(ltsensor, 1024);
  assertLessOrEqual(rtsensor, 1024);
  assertLessOrEqual(rdsensor, 1024);
  assertLessOrEqual(ldsensor, 1024);
}
```

In conclusion, we have implemented a lot of good programming practices using our result form testing and based on our development experience to reduce error points and optimise the firmware. For example, we have minimized the use of buffers and arrays, instead we opted to use individual objects to minimize memory fragmentation and avoid buffer overflow errors. We have also minimized the use ot strings which are notorious for bad memory management on the arduino platform.
The arduino bootloader, checks the firmware while it is being uploaded using checksums for each data segment uploaded, thus eliminating or minimizing the possibility of flash errors. Flash errors are detected while uploading and the user is prompted to attempt the upload process again.

We have also implemented a functionality that enables the arduino microcontroller to report live analog values from the solar sensors to the mqtt broker. This data gives the user real time feedback on the results of the movement of the stepper motors and can be gathered by the end user for storage visualization and further analysis,