AVRTest

A C++ Unit Test Framework for the 8-bit AVR Microcontroller Family

Developed by Charles Jessup Franklin

Introduction 3

Getting Started 3

Installing the Framework 3

Creating a Test Suite 3

Include a Configuration 4

Building Your Suite 4

Running Your Suite 5

Using the Arduino IDE With AVRTest 5

Understanding the Example Sketch 5

Writing and Running Your Tests 6

Configuring Your Tests 6

Test Cases 7

Making a Test Case 7

Naming Cases 8

Defining A Test 8

Test Fixtures 9

Defining a Test Fixture 9

Using a Test Fixture 10

Assertions 11

Configuration 12

General Settings 13

Using Serial Communication 13

Printing Formatting 14

Floating Point Comparisons 15

Printing User Defined Classes 16

Customizing the Makefile 16

General Settings 16

Programmer Settings 17

Compiler Flags 17

Other Settings 17

XML Like Format Definition 18

# Introduction

AVRTest is a lightweight testing framework written in C++ for the 8-bit AVR microcontroller family. Its purpose is to ease the task of writing unit tests and to automate running them, using features such as automatic test registration. It is modeled after other popular C++ unit testing frameworks, and may seem familiar when using it. This document describes how to use AVRTest and details all of its features.

# Getting Started

Writing your first test suite is easy. This section describes how to set up a simple suite from scratch. If you wish to program your device with the Arduino IDE, please read the *Using the Arduino IDE with AVRTest* section instead.

## Installing the Framework

The AVRTest framework is simply a directory can be installed anywhere. It is suggested that it be installed in a directory along a path with no spaces, as this makes linking your code to it easier. Simply copy the AVRTest root directory to your chosen install location.

## Creating a Test Suite

After installing the framework, you can create a test suite. Create a new directory and in that directory make a new C++ file. Give it any name. This is the file where all of your tests will be defined. Open this file and place the following line of code at the top:

#include <AVRTestSuite.h>

This line includes all of the framework’s headers. You will still need to link to the sources during your build. **NOTE** that due to the internal implementation of automatic test registration, only one file per build can include this header. Your tests should be defined after this line. For now use the following predefined test.

AVRTEST\_CASE(example\_case)

AVRTEST(example\_test) {

ASSERT\_TRUE(1 + 1 == 2);

}

ENDCASE

The above is an example of the simplest test. The first line is a macro that declares a test case with the name “example\_case”, which is used to contain a collection of related tests. A test suite can have many test cases, all of which must have unique names. See the *Making a Test* Case section for more naming rules. When the test case is run, all of its tests will also run. If one test in a test case fails, the entire case fails. The following line declares the actual test with the name “example\_test”. The name of a test only needs to be unique within the scope of the case. The test is defined between curly braces as if it were a function. See the *Defining a Test* section to see rules about naming and scope. Inside the test is an assertion macro. The assertion runs a check that must resolve to being true, or the test fails. A test can have many assertions, which execute in order. If one assertion fails, the entire test fails and no further assertions from that test will be executed. There are many predefined assertions, which are described in the *Assertions* section. The assertion in the example states that the boolean expression inside must be true. The final line is another macro that ends the case.

This example test can be found in the directory /AVRTest/Examples/Simple.

## Include a Configuration

To use a configuration other than the default, create a configuration file and place in in your target directory. A sample configuration file, *AVRTest\_Configuration.h*, can be found in the directory /AVRTest/Configuration. If Copy this file to your target directory and define. DO NOT edit the original copy of this file.

The default configuration uses the Arduino framework to set up serial communication. If you do not want to use this, comment out the line that defines AVR\_PRINT\_ARDUINO and uncomment AVR\_PRINT\_USER\_DEFINED. If you’ve done this, please see the *Using Serial Communication* section to see how to properly setup serial communication.

To understand all of the settings in the configuration file, please see the *Configuration* section.

## Building Your Suite

There are many ways to compile and build your test suite in preparation for programming your AVR device. The recommended way is to use one of the predefined makefiles, located in /AVRTest/Makefiles. If you left AVR\_PRINT\_ARDUINO defined in the configuration, then use the file “Makefile (Arduino)”, otherwise use the makefile “Makefile”. The example test “Simple” contains a makefile that is already setup. To learn how to customize the provided makefiles, see the *Customizing the Makefile* section.

If you are using a configuration file, in the makefile, edit the variable CONFIGURATION to be the name of the configuration file. If you are not using the provided makefile, make sure you define AVRTEST\_CONFIGURATION\_FILE as the name of your file at compilation time.

Using the terminal, change to the directory you made a couple of steps ago, and type “make”. If you don’t have GNUMake installed, use Google and install GNUMake. This command compiles and links all of your code, and produces an elf file and an Intel hex file.

## Running Your Suite

The way you run your test suite depends on your device. If you are using an Arduino or an Arduino compatible device, follow these directions. Attach your device using the USB port. Open the makefile and set the variable “PORT” equal to the serial port to which your device is attached. Go back to the terminal and type “make program”. This command uploads the Intel hex file using the Arduino boot loader.

If you are not using an Arduino compatible device, or are using some other method of programing, edit the makefile to use the proper AVRDUDE settings for your programmer.

Open a serial terminal and open the port that is connected to your device. If necessary, reset the device. The results will be displayed in the serial terminal.

# Using the Arduino IDE With AVRTest

Using the Arduino IDE with AVRTest is easy. First locate your copy of the AVRTest framework. Inside, there is a directory called *ArduinoSketch*, which contains a sketch directory called *AVRTestProgram*. Copy this entire directory; DO NOT edit the original copy of the sketch.

The sketch contains every file needed for AVRTest to compile. There are only two files in the sketch that need to be opened and edited. They are *AVRTestProgram.ino* and *AVRTest\_Configuration.h.* All others should remain unedited.

## Understanding the Example Sketch

Open the .ino file. Inside you will see an example of a simple test suite.

#include "AVRTestSuite.h"

AVRTEST\_CASE(SampleCase)

AVRTEST\_TEST(SampleTest) {

ASSERT\_TRUE(1 + 1 == 2)

}

ENDCASE

The first line includes all of the framework’s headers. These headers can only be included in one file. The remaining lines are an example of the simplest test. The first of the remaining lines is a macro that declares a test case with the name “SampleCase”, which is used to contain a collection of related tests. A test suite can have many test cases, all of which must have unique names. See the *Making a Test* Case section for more naming rules. When the test case is run, all of its tests will also run. If one test in a test case fails, the entire case fails. The following line declares the actual test with the name “SampleTest”. The name of a test only needs to be unique within the scope of the case. The test is defined between curly braces as if it were a function. See the *Defining a Test* section to see rules about naming and scope. Inside the test is an assertion macro. The assertion runs a check that must resolve to being true, or the test fails. A test can have many assertions, which execute in order. If one assertion fails, the entire test fails and no further assertions from that test will be executed. There are many predefined assertions, which are described in the *Assertions* section. The assertion in the example states that the boolean expression inside must be true. The final line is another macro that ends the case.

## Writing and Running Your Tests

All of your tests should be defined in the .ino file. Other code can be distributed elsewhere, but do not include the framework headers anywhere else other than the .ino file. Once your tests are written, compile and upload your Arduino as usual. The results of your tests will be printed to the serial terminal by default.

## Configuring Your Tests

One of the two files that should be edited is *AVRTest\_Configuration.h*. Do not confuse this with *AVRTest\_Default\_Configuration.h*. Both are necessary. Do not delete either. Use *AVRTest\_Configuration.h* to change how your tests will run. Inside is a copy of the default configuration and an explanation of other configurations. Additional information on custom configurations can be found in the *Configuration* section of this document.

# Test Cases

A test case is a collection or list of related tests. These tests usually are related by the code that they test. The tests in a case can be completely unrelated, but that goes against convention.

## Making a Test Case

To define a case, use the AVRTEST\_CASE( ) macro. This macro takes one parameter, and that is the name of the case, which will be displayed when the tests are run. To end the scope of a case, use the ENDCASE macro. Any tests defined between these two macros will be part of the case. It is valid to define a case with no tests, although this is pointless. A case with no tests will pass. The following code is an example of a case with two tests.

AVRTEST\_CASE(case\_1)

AVRTEST\_TEST(test\_1) {

ASSERT\_EQUALS(4, 2 + 2);

}

AVRTEST\_TEST(test\_2) {

ASSERT\_TRUE(5 > 4);

}

ENDCASE

Test cases should be defined at the global level and cannot be defined within the scope of another case, function, or class. The following code contains examples of invalid case declarations.

A case within a case:

AVRTEST\_CASE(case\_1)

AVRTEST\_CASE(case\_2)

ENDCASE // case\_2

ENDCASE // case\_1

Missing ENDCASE statement:

AVRTEST\_CASE(case\_1)

AVRTEST\_CASE(case\_2)

ENDCASE // case\_2

Not defined in the global scope:

void my\_function(void) {

AVRTEST\_CASE(case\_1)

ENDCASE

}

No name:

AVRTEST\_CASE()

ENDCASE

An invalid case declaration can result in a cryptic compiler error, so avoid them.

## Naming Cases

Naming a case is just like naming a function or variable. A case name cannot contain spaces or special characters. It’s safest to stick with alphanumeric characters and underscores. Additionally, cases must have unique names. Once again, an invalid case name will result in a cryptic compiler error.

# Defining A Test

A test is essentially a function that is calls the code in question and passes if the code works correctly and fails otherwise. Tests are declared using the AVRTEST\_TEST( ) macro. This macro takes one parameter, the name of the test. The test naming rules are the same as those regarding cases except for one difference. The name of a test only needs to be unique within the scope of its case. In other words, two different cases can have tests with the same name, but one case cannot have two tests with the same name.

A test must be defined within a case. Do not define a test anywhere else.

After the declaration, the actual test is defined between curly braces, just like a function. Different tests do not share the same scope. Anything declared in that test will only exist in the scope of that test. In the declaration of the test, call any code that you would like to test. Correct functionality of any given code is determined using assertions. An assertion takes an expression or other parameters and makes sure that the assertion is true. If the assertion is not true, the test fails immediately and the next test starts executing. Below is an example of a test written for the abs( ) function that passes.

AVRTEST\_TEST(negative\_input) {

int i = abs(-9);

ASSERT\_TRUE(i > 0);

ASSERT\_ZERO(i – 9);

}

There are two assertions. The first one asserts that the expression is true. The second asserts that the expression is equal to zero. Note that semicolons, just like normal lines of code, follow the assertions, although this is not necessary. The following is an example of a test that fails.

AVRTEST\_TEST(iTest) {

int i = 8;

int j = 3;

ASSERT\_FALSE(i == 8);

ASSERT\_EQUAL(max(i, j), j);

ASSERT\_TRUE(i > j);

ASSERT\_ZERO(i);

}

The first assertion passes, but the second one causes the entire test to fail. The third assertion would have passed, but is never executed. Had the second assertion passed, the test still would have failed because of the forth assertion. Note that the second assertion took two parameters. The first parameter is the value that is being tested, the actual value, and the second is the expected value. To see a full list of predefined assertions, please see the assertions section.

# Test Fixtures

A test fixture can be used to help reduce repeated code. A test fixture uses a class to define variables and functions to be used in a test. Test fixtures can optionally be used to define a function *setup( )* that is called automatically before a test that is part of the fixture. Test fixtures can also optionally define a function *taredown( )* that is called after the test is run, regardless of whether the test passes or not.

## Defining a Test Fixture

Defining a test fixture is exactly the same as making a subclass of another class. Start by declaring a new class that is a public subclass of the class *AVRTestFixture*. Below is an example of the simplest test fixture.

class MySimpleFixture : public AVRTestFixture {

protected:

// Define fixture members here

private:

// Define fixture methods here

}

The simplest test fixture is not very useful. Notice that neither *setup( )* nor *taredown( )* are defined, which is fine. Their default behavior is to do nothing. A more useful test fixture might look like this.

class MyUsefulFixture : public AVRTestFixture {

protected:

// Define fixture members here

char \*seasons[4];

private:

// Define fixture methods here

void setup() {

strdup(seasons[0], “Winter”);

strdup(seasons[1], “Spring”);

strdup(seasons[2], “Summer”);

strdup(seasons[3], “Autumn”);

};

void taredown() {

for (int i; i < 4; i++) {

free(seasons[i]);

}

};

char\* whichSeason(char\* month, int date) {

char \*season;

/\* Code that produces the correct season \*/

return season;

}

}

*MyUsefulFixture* has one member, *seasons*, and one method, *whichSeason*, in addition to the *setup* and *taredown* methods. Setup populates *seasons* with the correct names and *taredown* frees up the memory. Any test defined using this fixture would be able to access the variable *seasons* or the method whichSeason from within its scope.

## Using a Test Fixture

To use a test fixture, you define a test with the AVRTEST\_FIX(name, fixture) macro, followed by the definition in between curly braces just as before. The first parameter is the name of the test, and the second is the name of the fixture you would like to use with the test. The same naming rules apply. And as before, a test using a fixture must be defined inside of a test case. Note that a test that uses a fixture and a test that doesn't still cannot have the same name if they are in the same case. The following is an example of a test that uses the fixture defined in the previous section.

#include <Calendar.h> // a fictitious calendar library

using Calendar::myCalendar; // object that accesses my calendar

using Calendar::Date; // class that represents a date

AVRTEST\_CASE(calendar)

AVRTEST\_FIX(isCorrectSeason) {

Date birth = myCalendar.getMyBirthday();

ASSERT\_STR\_EQUAL(whichSeason(birth.month, birth.day),

seasons[2]);

}

ENDCASE

The test verifies that the fictitious calendar library at least has my birthday recorded in the correct season. Notice that the test accesses both the member *seasons* and the method *whichSeason*. The member *seasons* was populated before the test ran when *setup* was called automatically, and the memory was freed afterwards by *taredown*. Even if the test failed, taredown still would have been called.

Any number of tests across any number of different cases can use the same fixture. Additionally, it is valid to mix fixture tests and non-fixture tests in the same case.

Any test that uses a fixture will not see any changes made by any other tests that use the fixture, as fixture members will be reinitialized before every test.

# Assertions

Assertions are used to check the code in question and either allow a test to pass, or cause it to fail. This section lists and describes how to use all of the predefined assertions. Assertions must be used in the scope of a test. Do not use them anywhere else.

#### ASSERT\_TRUE(expression)

Passes if the expression resolves to a true boolean expression.

#### ASSERT\_FALSE(expression)

Passes if the expression resolves to a false boolean expression.

#### ASSERT\_ZERO(expression)

Passes if the expression is equal to zero.

#### ASSERT\_NZERO(expression)

Passes if the expression is not equal to zero.

#### ASSERT\_EQUAL(actual, expected)

Passes if the actual value is equal to the expected value. Do not use for c-style strings or floating-point numbers. Use ASSERT\_STREQUAL or ASSERT\_FLOAT\_EQUAL respectively instead.

#### ASSERT\_NEQUAL(actual, unexpected)

Passes if the actual value is not equal to the unexpected value. Do not use for c-style strings for floating-point numbers. Use ASSERT\_STRNEQUAL or ASSERT\_FLOAT\_NEQUAL respectively instead.

#### ASSERT\_STREQUAL(actual, expected)

Use with c-style strings only. Passes if the actual string is equal to the expected string (i.e. the *strcmp* function would return 0).

#### ASSERT\_STRNEQUAL(actual, unexpected)

Use with c-style strings only. Passes if the actual string is not equal to the unexpected string (i.e. the *strcmp* function would return a non-zero value).

#### ASSERT\_FLOAT\_EQUAL(actual, expected)

Use to compare floating-point numbers. Passes if the actual value is approximately equal to the expected value. The approximation metric can be changed in the configuration.

#### ASSERT\_FLOAT\_NEQUAL(actual, unexpected)

Use to compare floating-point numbers. Passes if the actual value is not approximately equal to the expected value. The approximation metric can be changed in the configuration.

#### ASSERT\_FAIL

This assertion causes the test to fail unconditionally. It takes no parameters. It is suggested that this is only used with a conditional control flow statement.

#### ASSERT\_PASS

This assertion passes unconditionally. It takes no parameters. It differs from all other assertions in the sense that it stops the execution of the test even though it passes.

# Configuration

To use a configuration other than the default configuration, include a configuration file in your target directory. If you are using one of the provided make files, make sure to set the CONFIGURATION variable as the name of your file. If you are not using a provided make file, define AVRTEST\_CONFIGURATION\_FILE as the name of your configuration file at compile time. A sample file can be found in the directory /AVRTest/Configuration. An explanation for every configuration parameter can be found in this section as well as the sample configuration file.

## General Settings

#### Maximum number of tests

In general, there should be no need to set a maximum number of tests, as dynamic memory allocation is used when registering the tests. If for some reason this is causing problems, a maximum number of tests and cases can be defined to turn off dynamic memory allocation. To do this, define AVR\_MAX\_TESTS as the desired number of maximum cases and tests per case.

Default: No Max

#### Arduino IDE

If you want to use the Arduino IDE to program your board, define ARDUINO\_IDE. If you use the provided sketch, this will be defined automatically.

#### Arduino Libraries

If for any reason at all, you link to some of the Arduino libraries, define ARDUINO\_LIBS. It makes sure that the main function calls the correct code to set up the Arduino properly. Do not define if you are using the Arduino IDE.

Default: Not defined

## Using Serial Communication

AVRTest uses serial communication to report the results of the tests back to the user. The following section explains all of the settings for serial communication and how to set up custom serial functions.

NOTE: Of the following definitions, do not define more than one at the same time:

* AVR\_PRINT\_ARDUINO
* AVR\_PRINT\_ADDR
* AVR\_PRINT\_USER\_DEFINED

Default: AVR\_PRINT\_ARDUINO defined as Serial

#### Baud Rate

If you are not using a user-defined function for printing, define AVR\_PRINT\_BAUD as the desired baud rate.

Default: 9600

#### Arduino Serial

If you are linking to the Arduino libraries and would like to use them for serial communication, define AVR\_PRINT\_ARDUINO as the serial port you want used (e.g. Serial, Serial2 …).

Default: Serial

#### Memory Mapped Device

To use either a memory-mapped device or just have the results written to a specific address, define AVR\_PRINT\_ADDR as the address where you want the results printed. Print messages will be written one character at a time to the address as fast as possible.

Default: Not defined

#### User Defined Printing

If you would like to define your own methods for printing, define AVR\_PRINT\_USER\_DEFINED. Defining this statement causes the following functions to be declared but not defined:

* void avr\_print\_user\_setup(void), called before any tests are run
* void avr\_print\_user\_sendchar(char c), called to send a character

The user must define these functions, either in the file that contains the tests, or in another file that is linked in the final build. The function *avr\_print\_user\_sendchar* will be called repeatedly for each character of a string to be sent. If for some reason you want to handle strings differently, also uncomment the line that defines AVR\_PRINT\_USER\_DEFINED\_STRING. This causes the following function to be declared:

* void avr\_print\_user\_sendstring(const char\* c)

The muse must also define this function.

Default: Not defined

## Printing Formatting

The formatting of the results can be customized. This section describes the formatting options available.

#### Human Readable

If you want the results to printed with the intention of being read by a human, define #AVR\_LOG\_HUMAN\_READABLE

#### XML

To print the results in an XML like style, define AVR\_LOG\_XML. To see details of the format, please see the *XML Like Format* section.

Default: Not defined

If using this format, define of AVR\_LOG\_TAB\_CHAR as the desired tab character, or characters.

Default: four spaces

#### Number Base

Numbers can be printed in different bases. It is suggested that you define one of the predefined bases:

* AVR\_PRINT\_DEC, decimal
* AVR\_PRINT\_BIN, binary
* AVR\_PRINT\_HEX, hexadecimal

If you would like to use an arbitrary base, define AVR\_PRINT\_BASE as an integer value representing the base you want.

Note that floating point numbers will always be printed in base 10.

Default: base 10

#### Signed Numbers

Signed numbers can always be printed in decimal. If this is desired, define AVR\_PRINT\_SIGNED\_DEC

#### Floating Point Numbers

Floating point numbers will always be printed in base 10. To change the number of places to the right of the decimal point that will be printed, define AVR\_FLOAT\_POINTS as the desired number.

Default: 4

#### New Line Character

Define AVR\_PRINT\_NL as the desired new line character.

Default: ‘\n’

## Floating Point Comparisons

Doing direct comparisons floating-point numbers is not done, because floating-point calculations are inaccurate and not every decimal can be represented as a float. Instead two different floating-point numbers are considered to be equal if they are close to each other within some threshold. The following section describes how to change how floating-point comparisons are done.

Default: Consecutive Float Comparison

#### Relative Comparison

One method of comparing floating-point numbers is to look at the error between the two, essentially looking at the percent difference. This method also considers two floats to be equal if the fixed difference between the two numbers is within a threshold value. To use this method, define AVR\_FLOAT\_REL\_COMPARE.

Default: Not defined

Set the relative and absolute thresholds by defining AVR\_FLOAT\_MAX\_REL and AVR\_FLOAT\_MAX\_ABS respectively.

Default: .00001 and .000001 respectively

#### Consecutive Float Comparison

Another common method of comparing floating-point numbers is to look at how many representable floats exist between the two numbers. If the difference is below a certain threshold, the two floats are considered to be equal. To use this method, define AVR\_FLOAT\_MAXULPS as an integer value representing the threshold difference.

Default: 100

## Printing User Defined Classes

In order for AVRTest to print your classes and objects, you must define a method that prints the object.

**This feature is undocumented because it is incomplete and does not work well. You will see it commented in the default configuration file. Do not use it.**

# Customizing the Makefile

One common way of building a program is to use GNUMake and a makefile. A basic makefile is provided in the directory /AVRTest/Makefiles. This section describes how to tailor this file to your needs.

## General Settings

At the top, there are a few variables that need to be set.

TARGET sets the name that the final .elf and .hex files will have.

MCU sets the microcontroller ID that you are using. It should be the same as the mmcu option to avr-gcc.

F\_CPU sets the clock frequency of your system in Hz.

AVRTEST sets the directory path to the AVRTest framework installation.

BASE sets the directory path to the Arduino framework. The default is the one included with AVRTest.

LIBS should list any additional Arduino libraries that you are using (e.g. SPI).

TARG\_SRC sets the path to the directory that contains your tests

USER\_DIRS should list any other directories that need to be included for your tests to compile.

CONFIGURATION sets the configuration file to be used. This should either be set to the name of the configuration file in your target directory, or left blank to use the default configuration.

SOURCES should list any files that contain your tests that need to be compiled and linked.

PORT sets the path to the serial port being used to program your device

CC sets the C compiler that is used. The default is avr-gcc.

CXX sets the C++ compiler that is used. The default is avr-g++.

## Programmer Settings

The default program used to program the user device is avrdude. The following variables control how your program is uploaded:

* AVRDUDE
* AVRDUDE\_PROGRAMMER
* AVRDUDE\_PORT
* AVRDUDE\_WRITE\_FLASH
* AVRDUDE\_FLAGS

## Compiler Flags

The following variables are used to set the compiler flags:

* CTUNING
* WARNINGS

## Other Settings

Unless you know what you are doing, do not change anything else.

# XML Like Format Definition

The results of a test run can be printed out in an XML like format. This section defines all of the tags used.

#### AVRTestResults

The opening tag will always be AVRTestResults. This tag has no attributes. This tag can directly enclose the following tags:

* comment (optional)
* AVRTestCase (optional)
* result
* failed

#### AVRTestCase

This tag contains the results of a test case. It has one attribute, title, the title of the case. Its parent must be AVRTestResults. This tag can directly enclose the following tags:

* comment (optional)
* AVRTest (optional)
* result

#### AVRTest

This tag contains the results of a test. Its parent must be AVRTestCase. It has one attribute, title, the title of the test. This tag can directly enclose the following tags:

* serialLog (optional)
* result
* message (only if test failed)
* reason (only if test failed)

#### result

This tag says whether the enclosing object passed or failed. It represents this with a true for pass, or false for fail. There may only be one direct instance of this tag enclosed by a parent tag.

#### message

This tag encloses a string that explains why a test has failed. There may only be one direct instance of this tag enclosed by a parent tag.

#### failed

This tag encloses either the total number of failed tests or cases. The type attribute determines which one. The parent tag must be AVRTestResults.

#### reason

This tag contains the reason a test failed. Its parent must be AVRTest. This tag can directly enclose the following tags:

* variable (the variable being tested)
* expected (the expected value)
* actual (the actual value)

#### variable

This tag encloses the name of a variable.

#### expected

This tag encloses the expected value of a variable.

#### actual

This tag encloses the actual value of a variable.

#### comment

Can enclose any information that may or may not be relevant to the results. The information enclosed by a comment tag can be tossed as it is not important.