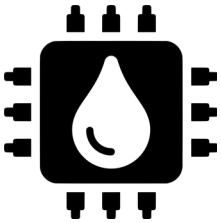

Web-server Embedded-system Tester

Test Language User Guide

The WET System makes use of the *pytest* Python library in order to coordinate and run user tests. This ensures a readable way to create simple and complex tests that can be quickly read and written. Additionally a `dut` (device-under-test) library has been created for quick and easy calls to make assertions about the device. For example, consider these three simple tests:

```
import dut

def test_error():
    print("@@ 10")
    print("** 1")

def test_1():
    dut.init()
    dut.delay(5)
    dut.port_is(PORTD, "0100XXXX")
    print("** 4")

def test_2():
    dut.init()
    dut.waitfor(PORTB, "XXXXXXX1")
    dut.port_is({PORTB: "XXXXXX01",
                 PORTD: "XXXXXX10"})
    print("** 5")
```

Tests are written each inside of a python `def` declaration, and can each be assigned points so that they have different weights in an assignment. Generally, testing makes use of `assert` methods provided by Python – however this is built in to the library. Each test is ran independently, meaning that each test will start from the same point and run linearly on its own.

Test File Requirements

The `dut` library provides functionality for desired testing methods in order to go through recorded I/O and make testing assertions. First, the library must be referenced via the `import dut` directive at the top of the test file.

Afterward, test methods may be created using the `def` directive. Tests should be named with the "test_" prefix, such as `def test_LED():`

At this point, the tests can be written in regular Python programming. Generally, if variables are needed, they should be kept within the scope of the test they are being used in. For example, if a counter is used inside a test, it should be defined within the test function itself. For testing, there are some pre-defined methods to make device checking easy.

To give credit for tests, some extra functionality is required in order to assign points to each test directive. First, a "blank" test describing the total amount of points in the project is required, using `@@`:

```
def test_error():
    print("@@ 10")
    print("** 0")
```

The total points may be changed freely. Note the formatting of either message, as it is required by the system. At the end of each test, a print message is required telling how many points to give if the test is passed (even if the points for that test is zero, in the case of `test_error`, e.g.

```
print("** 5")
```

Test Method Documentation

`dut.init()`

This method should be called at the start of every test. It has the purpose of resetting all the simulation values, such as the current time and state of the chip. It also loads the testing results file into an easily-indexable form. If an argument is passed, it loads the results json file with the argument value over the default `results.json`. The testing follows a linear fashion in which the device will take on a particular set of port values at a given time value. Advancing in time will advance the port values if the real-life values changed at that time.

`dut.port_is(port, value)` (or)

`dut.port_is({port1: value1, port2: value2, ...})`

This function is the equivalent of an assertion, except using port values on the device under test at the current testing time. It can be used in two ways, to check either one port value or multiple simultaneously. In the first case, `port` is any of `PORTB, PORTC, PORTD, PORTE`. Value is a string of the desired port value, for example "`00010110`", "`XXXX0101`" where X denotes a "don't care" value, e.g.,

`dut.port_is(PORTC, "XXXX101")` (PORTC is 7 bits wide)

In the case of multiple desired port values, they can be entered as dictionary elements, such as:

`dut.port_is({PORTC:"XXXX101", PORTD:"0110XXXX"})`

In the case that a port does not match the wanted value, there will be an error thrown and the test will not pass. The error will provide some details regarding the current and wanted state of the device ports, but these can be disabled:

`dut.port_is(PORTC, "XXXX101", verbose = False)`

`dut.waitfor(port, value)` (or)

`dut.waitfor({port1: value1, port2: value2, ...})`

This method will advance the current time to the first occurrence of the port arguments. It is similar to **`port_is`** in that the arguments take the same form. If the desired port values never occur, this method will throw an error. Otherwise, the first timestep where *value* occurs will be the current time of the testing, for example:

`dut.waitfor(PORTC, "XXXXXX1")`

This is an effective way to wait for a clock sync pulse. It will look for the first time PORTC has a 1 on bit 0. From there, the **`port_is`**, **`delay`**, or **`waitfor`** functions may be used at the new time step, to check a different port value, wait longer (now that it's synchronized), or wait for a clock to go low, for example.

`dut.delay(time)`

This function will advance the current time by the supplied value and update the current value of the device accordingly. The supplied argument can be negative as well, if the operator wishes to return to a previous step. Note that the time value is in seconds (and not clock cycles) – so the amount of device cycles will be advanced by 16E6 multiplied by the supplied argument. This may be used to simply wait in between events, for example in UART:

`dut.waitfor(PORTD, "XXXXXX0X")` to synchronize to the start bit, and then **`dut.delay(1/baudrate)`** to get the first data bit. The rest of the library supports plain Python implementation, so all functional programming (looping, conditionals, etc) is all available via the easily-writably Python language.

`dut.segment_is(value)`

This method is used to read the value of a quad 7-segment display attached via the shield attachment. The value supplied will be a 4-digit hex number, e.g.

`dut.segment_is(0x1234)`

will assert that the segments (left-to-right) will read 1, 2, 3, and 4. The reading is done left-to-right as well, i.e. PB3 should be zero (enabling the first digit), then PB2, PB1, and PB0 in that order. The method will also form assertions that the anodes are sent low, and will error otherwise. As the digits are flickered, some time generally passes to be able to test the current value of the digits, however at the end of the method the time step will return to the original pre-checked value. This is due to the fact that some student implementations will write right-to-left flickering methods, and will take 4 flickering cycles to check rather than just one. In any case, this would only cause error if the digits change at a pace faster than readable.