// mock_daq.py

Why did I use **“defaultdict”**

```
import time
import threading
from collections import defaultdict
```

It provides a default value for the key that does not exist. This means that if a key is not found in the dictionary, instead of a KeyError being thrown, a new key is created with the default value.

New concept that I learnt here:

1) Threading:

A technique used to run multiple threads (sub-tasks) concurrently within a program, allowing for parallel execution and efficient use of CPU resources. They require careful synchronization to avoid conflicts.

In the given MockDAQDevice class, threading is used to handle the toggling of digital pins independently of the main program flow. Here's a detailed explanation of how threading is applied and what happens in the code:

self._stop_toggle = threading.Event(): This event is used to signal the thread to stop digital pin toggling execution.
self._toggle_thread = None: This attribute holds the reference to the toggle thread argument.

start_toggle:
Purpose: Starts a thread to toggle the values of the specified input and output channels at regular intervals.
Validation: Ensures the input channel is configured as 'input' and the output channel as 'output'.
Event Clearing: Clears the stop event (self._stop_toggle.clear()), ensuring the thread will run.
Thread Creation: Creates a new thread (self._toggle_thread) targeting the _toggle_pin method with arguments for channels, interval, and tolerance.
Thread Start: Starts the thread (self._toggle_thread.start()), initiating the toggling process in parallel with the main program.

stop_toggle:
Purpose: Signals the toggle thread to stop and waits for it to terminate.
Set Stop Event: Sets the stop event (self._stop_toggle.set()) to signal the running thread to halt.
Join Thread: Waits for the thread to finish execution (self._toggle_thread.join()), ensuring a clean termination.

_toggle_pin:
Purpose: The function run by the toggle thread to toggle the input and output channel values at specified intervals.
Initial Setup: Calculates the next toggle time (next_toggle_time = time.time() + interval).
Loop: Continuously checks if the stop event is set.
Time Check: If the current time is within the tolerance range of the next toggle time, it toggles the channel values.
Value Toggle: Changes the input channel value to its opposite and sets the output channel to the inverse of the new input value.
Update Toggle Time: Advances the next toggle time by the specified interval.
Sleep: Sleeps for a short duration (time.sleep(tolerance)) to avoid busy-waiting and ensure the next toggle check occurs after a small delay.

Explanation of What Exactly is Happening in MockDAQDevice

Initialization:
When an instance of MockDAQDevice is created, it initializes the digital pins and prepares the threading event and thread reference.
Configuration:

The configure_digital_channel method sets up channels as either input or output.
Toggling Process:
When start_toggle is called, it starts a new thread that runs _toggle_pin:
This thread runs concurrently with the main program, repeatedly toggling the specified channels at the given interval.
The while loop within _toggle_pin keeps the thread active, checking for the stop event to determine when to cease operation.
The time.sleep(tolerance) call ensures that the thread periodically wakes up to check and toggle the pin values, achieving near real-time toggling with a tolerance for timing accuracy.

Stopping the Toggling:
When stop_toggle is called, it sets the stop event, causing the loop in _toggle_pin to exit.
The join method ensures that the main program waits for the thread to fully terminate before continuing, ensuring no stray threads are left running.

Summary
Threading in this code allows the MockDAQDevice class to toggle digital pin values independently of the main execution flow. This enables real-time updates and responsive interactions without blocking or delaying other operations in the program. By using threading events and careful timing, the code achieves efficient and precise toggling behavior.

Toggle Pin Method:

The _toggle_pin method is the function executed by the thread to perform the actual toggling:

```python
def _toggle_pin(self, input_channel, output_channel, interval, tolerance):
    next_toggle_time = time.time() + interval
    while not self._stop_toggle.is_set():
        current_time = time.time()
        if abs(current_time - next_toggle_time) <= tolerance:
            current_value = self.digital_pins[input_channel]['value']
            new_value = not current_value
            self.digital_pins[input_channel]['value'] = new_value
            self.digital_pins[output_channel]['value'] = not new_value
            next_toggle_time += interval
        time.sleep(tolerance)  # Check in small intervals for more accurate
toggling
```

This method continuously checks whether it is time to toggle the pins based on the given interval and tolerance. If it is, it toggles the input and output pins and updates next_toggle_time for the next iteration. The loop will keep running until a signal **self._stop_toggle** tells it to stop.

Self._stop_toggle is called in the test_mock_daq.py script.

Some Lessons Learnt:

1) Usage of **raise ValueError** instead of Print statements**.**
2) Writing unit tests for asynchronous behavior. using **time.sleep()** to wait for sufficient time to observe the toggling behavior and ensuring cleanup with addCleanup or equivalent methods helps manage the asynchronous aspects
3) In the provided script, **digital_pins** is a shared resource that represents the state of digital pins on a hardware device. Multiple threads can modify this resource, such as reading pin values, writing new values, and toggling pins. There can be a better way to handle this.
4) Need to look into **Thread Locks** to have this integration and mitigate the risks associated with race conditions.

What are the different Test scenarios?

1) Test #1: Test Initial State

It reads the initial state of pin1 using self.daq_device.read_digital('pin1') and the initial state of pin2 directly from self.daq_device.digital_pins['pin2']['value'].
Next, it prints the initial state of both pins using print(f"Initial state - Pin1: {pin1_initial}, Pin2: {pin2_initial}").
It asserts that both pin1_initial and pin2_initial are False, indicating that pin1 is configured as an input and pin2 as an output, and they both start in an inactive state.
If the assertion passes, it prints "Initial state test passed." to indicate that the test was successful.
Finally, it prints the latest pin statuses using self.print_latest_pin_status().

This test ensures that the MockDAQDevice instance is correctly initialized with the specified pin configurations, and the pins are in the expected initial state before any operations are performed on them.
It serves as a baseline check to verify that subsequent operations in other tests do not inadvertently change the initial state of the pins.

In summary, this test provides assurance that the initial setup of the pins in the MockDAQDevice object is correct and consistent with the expectations defined in the test setup.

2) Test #2: Test Write Digital

The test writes a digital True value to pin2 using self.daq_device.write_digital('pin2', True). This action simulates turning on an output pin. It then reads the current value of pin2 from the MockDAQDevice instance using pin2_value = self.daq_device.digital_pins['pin2']['value'].
The test asserts that the value of pin2 after the write operation is True by using self.assertTrue(pin2_value). This ensures that the write operation was successful and that the output pin reflects the expected value. If the assertion passes, it prints "Write operation test passed." to indicate that the test was successful.
Finally, it prints the latest pin statuses using self.print_latest_pin_status().
This test verifies that the write_digital method of the MockDAQDevice class correctly updates the value of an output pin to the desired state. It ensures that the device behaves as expected when instructed to set an output pin to a specific digital value.
In summary, this test ensures that writing to the output pin pin2 results in the expected value change and verifies the correctness of the write operation functionality in the MockDAQDevice class. We do the same thing for Failure pin as well.

3) Test #3: Test Toggle Pins

The test invokes the start_toggle method of the MockDAQDevice instance to initiate toggling between pin1 and pin2 with a 1-second interval. The method call also specifies a tolerance of 0.01 seconds.
To observe two toggles, it waits for a little more than 2 seconds using time.sleep(2.1).
After the first toggle interval, it reads the values of pin1 and pin2 using self.daq_device.read_digital('pin1') and self.daq_device.digital_pins['pin2']['value'], respectively, and prints the state of both pins.
It asserts that pin1 and pin2 have different values after the first toggle.
Then, it waits for a little more than 1 second to observe another toggle (time.sleep(1.1)).
After the second toggle interval, it again reads the values of pin1 and pin2, prints their states, and verifies that pin1 and pin2 have toggled as expected.
If the assertions pass, it prints "Toggle test passed. Stopping the toggle process." to indicate that the test was successful.
Finally, it stops the toggling process using self.daq_device.stop_toggle() and prints the latest pin statuses using self.print_latest_pin_status().

This test ensures that the start_toggle method correctly toggles between pin1 and pin2 with the specified interval and that pin2 mirrors the state of pin1.
It validates that the device behaves as expected when toggling pins and maintains the desired relationship between the two pins.

In summary, this test verifies the toggling functionality between pin1 and pin2, ensuring that pin1 toggles every second and pin2 mirrors the state of pin1.

4) Test #4: Test Timing Mechanisms

The test invokes the start_toggle method of the MockDAQDevice instance to initiate toggling between pin1 and pin2 with a 1-second interval and a tolerance of 0.01 seconds. This can be changed in the Test input arguments for whatever mechanism you want to design your test fixture to be.
It records the start time using start_time = time.time() just before the toggle operation begins.
The test then waits for a little more than 2 seconds (specifically, 2.1 seconds) using time.sleep(2.1) to observe two toggles. After waiting, it records the end time using end_time = time.time(). It calculates the elapsed time by subtracting the start time from the end time: elapsed_time = end_time - start_time. The test then prints the actual elapsed time and compares it with the expected duration (approximately 2 seconds). It uses the assertAlmostEqual method to verify that the actual elapsed time is close to the expected time (2.1 seconds) within a specified tolerance of 0.05 seconds. If the assertion passes, it prints "Timing mechanism test passed." to indicate the success of the test. Finally, it stops the toggling process using self.daq_device.stop_toggle() and prints the latest pin statuses using self.print_latest_pin_status().

This test ensures that the timing mechanism of the toggle operation functions correctly, producing toggles at the expected intervals. It validates that the actual duration of toggles closely matches the expected duration, considering the specified tolerance. By verifying the timing mechanism, it ensures the reliability and accuracy of time-based operations in the MockDAQDevice. In summary, the test_timing_mechanism method verifies that the toggle operation's timing mechanism operates within the expected duration with a specified tolerance, ensuring accurate timing behavior**.**

5) Test #5: Test Invalid Operations
The test_invalid_operations method tests for invalid write and read operations on the MockDAQDevice object. Specifically, it verifies that attempting to write to an input pin (pin1) and attempting to read from an output pin (pin2) raises a ValueError, as these operations are not allowed according to the configuration of the MockDAQDevice.

Here's how the test works:

Invalid Write Operation Test:
The test attempts to write a value of True to the input pin - pin1 using self.daq_device.write_digital('pin1', True).
It expects a **ValueError** to be raised since writing to an input pin is not allowed.
The assertRaises context manager is used to assert that the specified exception is raised.
After the test, the latest pin statuses are printed using self.print_latest_pin_status().

Invalid Read Operation Test:
The test attempts to read from the output pin pin2 using self.daq_device.read_digital('pin2').
It expects a **ValueError** to be raised since reading from an output pin is not allowed..
After the test, the latest pin statuses are printed.

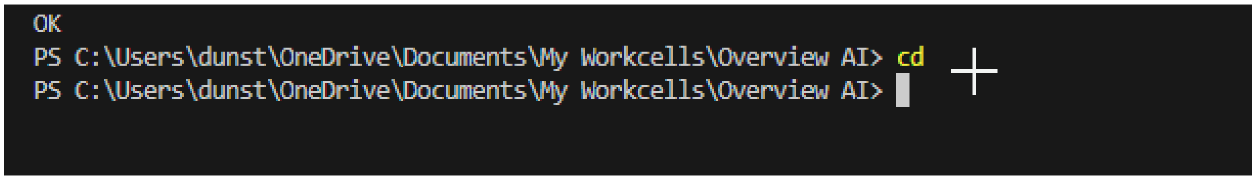
The pin statuses are not explicitly written before the test or while testing in this method.

6) Test #6 and #Test 7: Test External Signal Generator for Pin1 and Pin2

This test emulates the behavior of an external signal generator. This function toggles the value of pin1 and pin2 at regular intervals for a specified duration. The helper function called mock_external_signal_generator, is defined within the test method to emulate the behavior of an external signal generator by continuously toggling the value of a specified channel (pin1and pin2 in this case) at regular intervals (interval) for a specified duration (duration). A new thread (external_signal_thread) is created to execute the mock_external_signal_generator function.
The target function for this thread is set to mock_external_signal_generator, and the arguments passed are the mock DAQ device instance (self.daq_device) and the channel (pin1 and pin2) to simulate external signals. The test waits for a specified duration (6 seconds) to observe multiple toggles of pin1 by the mock external signal generator. After the observation period, the test reads the final state of pin1 using the read_digital method of the mock DAQ device.
It prints the final state of pin1 to the console to indicate whether the simulated external signal generation process affected the state of pin1 as expected.

To Run this HARDWARE PROJECT:

1) Make sure all the project files are in the same directory.
2) Make sure you have CD'd into the OVERVIEW AI directory, just like this.

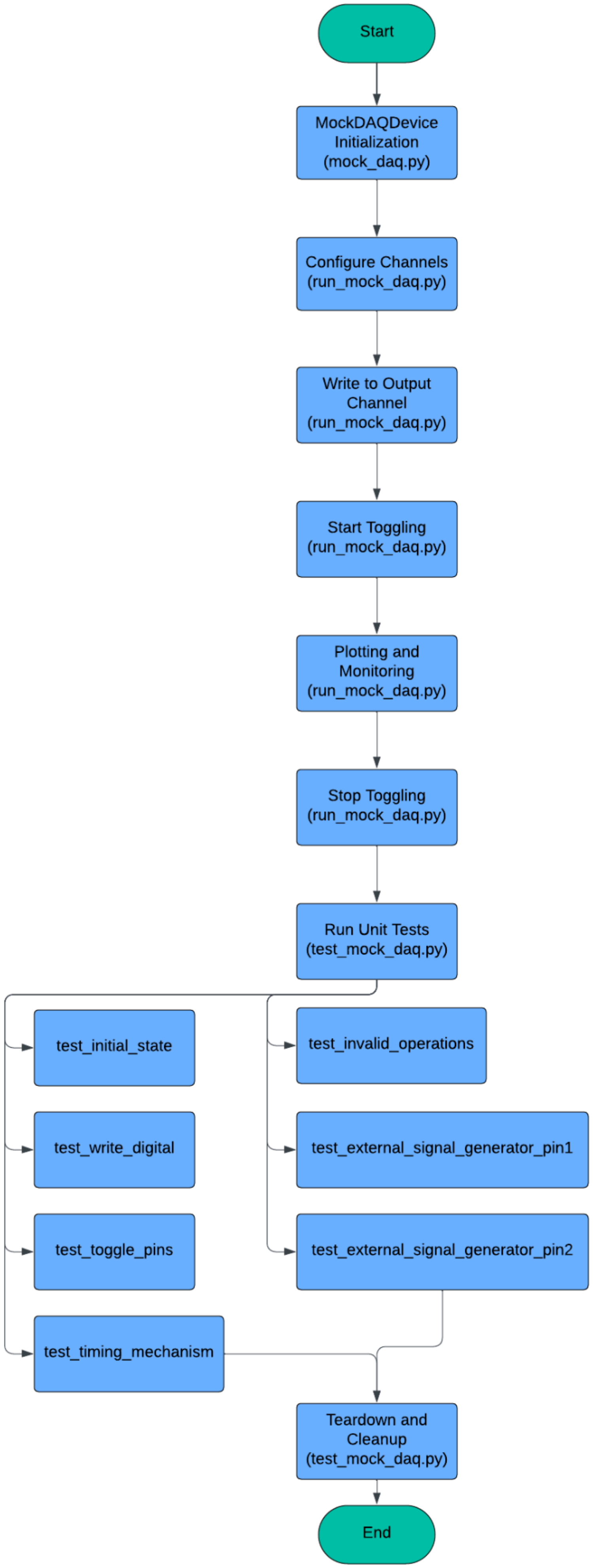


```
OK
PS C:\Users\dunst\OneDrive\Documents\My Workcells\Overview AI> cd
PS C:\Users\dunst\OneDrive\Documents\My Workcells\Overview AI>
```

3) Make sure you have all the dependencies installed. If not PIP them manually through the terminal and install them.
4) Once you see no errors. Open the “test_mock_daq.py” and run the script to begin the testing.
5) To get a FAIL scenario, you can modify LINE 141, elapsed_time to 2.0 seconds with delta = 0.05. For a PASS scenario set elapsed_time to 2.1 seconds with delta = 0.05
6) The run_mock_daq, just gives you a visual representation of how the 2 pins toggle in matplotlib
7) Each time you make a modification in the script, ensure to SAVE the file before running the script. Else, your changes most likely wont take place and the script running wouldn't be the latest and the greatest.

   Below is the flowchart to give you a better picture of the skeleton diagram of the HARDWARE Project.

   If you plan on adding an external hardware, we would have to comment out, or remove the External Signal Generator part. Also while testing the test_toggle_pins, we might have to modify the script to configure the digital channels or read or write pins based on the attributes available from the NIDAQ library. We have to adjust the interval and tolerances, as there can be some lag when conducting the HiL(Hardware In Loop) test.


Start
MockDAQDevice Initialization (mock_daq.py)
Configure Channels (run_mock_daq.py)
Write to Output Channel (run_mock_daq.py)
Start Toggling (run_mock_daq.py)
Plotting and Monitoring (run_mock_daq.py)
Stop Toggling (run_mock_daq.py)
Run Unit Tests (test_mock_daq.py)
test_initial_state
test_write_digital
test_toggle_pins
test_timing_mechanism
test_invalid_operations
test_external_signal_generator_pin1
test_external_signal_generator_pin2
Teardown and Cleanup (test_mock_daq.py)
End

What are my tests doing?

1) Test #1: Test Initial State

It reads the initial state of pin1 using self.daq_device.read_digital('pin1') and the initial state of pin2 directly from self.daq_device.digital_pins['pin2']['value'].
Next, it prints the initial state of both pins using print(f"Initial state - Pin1: {pin1_initial}, Pin2: {pin2_initial}").
It asserts that both pin1_initial and pin2_initial are False, indicating that pin1 is configured as an input and pin2 as an output, and they both start in an inactive state.
If the assertion passes, it prints "Initial state test passed." to indicate that the test was successful.
Finally, it prints the latest pin statuses using self.print_latest_pin_status().

This test ensures that the MockDAQDevice instance is correctly initialized with the specified pin configurations, and the pins are in the expected initial state before any operations are performed on them.
It serves as a baseline check to verify that subsequent operations in other tests do not inadvertently change the initial state of the pins.
In summary, this test provides assurance that the initial setup of the pins in the MockDAQDevice object is correct and consistent with the expectations defined in the test setup.

2) Test #2: Test Write Digital

The test writes a digital True value to pin2 using self.daq_device.write_digital('pin2', True). This action simulates turning on an output pin. It then reads the current value of pin2 from the MockDAQDevice instance using pin2_value = self.daq_device.digital_pins['pin2']['value'].
The test asserts that the value of pin2 after the write operation is True by using self.assertTrue(pin2_value). This ensures that the write operation was successful and that the output pin reflects the expected value. If the assertion passes, it prints "Write operation test passed." to indicate that the test was successful.
Finally, it prints the latest pin statuses using self.print_latest_pin_status().
This test verifies that the write_digital method of the MockDAQDevice class correctly updates the value of an output pin to the desired state. It ensures that the device behaves as expected when instructed to set an output pin to a specific digital value.
In summary, this test ensures that writing to the output pin pin2 results in the expected value change and verifies the correctness of the write operation functionality in the MockDAQDevice class. We do the same thing for Failure pin as well.

3) Test #3: Test Toggle Pins

The test invokes the start_toggle method of the MockDAQDevice instance to initiate toggling between pin1 and pin2 with a 1-second interval. The method call also specifies a tolerance of 0.01 seconds.

To observe two toggles, it waits for a little more than 2 seconds using time.sleep(2.1). After the first toggle interval, it reads the values of pin1 and pin2 using self.daq_device.read_digital('pin1') and self.daq_device.digital_pins['pin2']['value'], respectively, and prints the state of both pins.
It asserts that pin1 and pin2 have different values after the first toggle.
Then, it waits for a little more than 1 second to observe another toggle (time.sleep(1.1)).
After the second toggle interval, it again reads the values of pin1 and pin2, prints their states, and verifies that pin1 and pin2 have toggled as expected.
If the assertions pass, it prints "Toggle test passed. Stopping the toggle process." to indicate that the test was successful.
Finally, it stops the toggling process using self.daq_device.stop_toggle() and prints the latest pin statuses using self.print_latest_pin_status().

This test ensures that the start_toggle method correctly toggles between pin1 and pin2 with the specified interval and that pin2 mirrors the state of pin1.
It validates that the device behaves as expected when toggling pins and maintains the desired relationship between the two pins.
In summary, this test verifies the toggling functionality between pin1 and pin2, ensuring that pin1 toggles every second and pin2 mirrors the state of pin1.

4) Test #4: Test Timing Mechanisms

The test invokes the start_toggle method of the MockDAQDevice instance to initiate toggling between pin1 and pin2 with a 1-second interval and a tolerance of 0.01 seconds. This can be changed in the Test input arguments for whatever mechanism you want to design your test fixture to be.
It records the start time using start_time = time.time() just before the toggle operation begins.
The test then waits for a little more than 2 seconds (specifically, 2.1 seconds) using time.sleep(2.1) to observe two toggles. After waiting, it records the end time using end_time = time.time(). It calculates the elapsed time by subtracting the start time from the end time: elapsed_time = end_time - start_time. The test then prints the actual elapsed time and compares it with the expected duration (approximately 2 seconds). It uses the assertAlmostEqual method to verify that the actual elapsed time is close to the expected time (2.1 seconds) within a specified tolerance of 0.05 seconds. If the assertion passes, it prints "Timing mechanism test passed." to indicate the success of the test.
Finally, it stops the toggling process using self.daq_device.stop_toggle() and prints the latest pin statuses using self.print_latest_pin_status().

This test ensures that the timing mechanism of the toggle operation functions correctly, producing toggles at the expected intervals. It validates that the actual duration of toggles closely matches the expected duration, considering the specified tolerance. By verifying the timing mechanism, it ensures the reliability and accuracy of time-based operations in the MockDAQDevice. In summary, the test_timing_mechanism method verifies that the toggle operation's timing mechanism operates within the expected duration with a specified tolerance, ensuring accurate timing behavior**.**

5) Test #5: Test Invalid Operations

The test_invalid_operations method tests for invalid write and read operations on the MockDAQDevice object. Specifically, it verifies that attempting to write to an input pin (pin1) and attempting to read from an output pin (pin2) raises a ValueError, as these operations are not allowed according to the configuration of the MockDAQDevice.

Here's how the test works:

Invalid Write Operation Test:
The test attempts to write a value of True to the input pin - pin1 using self.daq_device.write_digital('pin1', True).
It expects a **ValueError** to be raised since writing to an input pin is not allowed.
The assertRaises context manager is used to assert that the specified exception is raised.
After the test, the latest pin statuses are printed using self.print_latest_pin_status().

Invalid Read Operation Test:
The test attempts to read from the output pin pin2 using self.daq_device.read_digital('pin2').
It expects a **ValueError** to be raised since reading from an output pin is not allowed..
After the test, the latest pin statuses are printed.

The pin statuses are not explicitly written before the test or while testing in this method.

6) **Test #6 and #Test 7: Test External Signal Generator for Pin1 and Pin2**

This test emulates the behavior of an external signal generator. This function toggles the value of pin1 and pin2 at regular intervals for a specified duration. The helper function called mock_external_signal_generator, is defined within the test method to emulate the behavior of an external signal generator by continuously toggling the value of a specified channel (pin1and pin2 in this case) at regular intervals (interval) for a specified duration (duration). A new thread (external_signal_thread) is created to execute the mock_external_signal_generator function.
The target function for this thread is set to mock_external_signal_generator, and the arguments passed are the mock DAQ device instance (self.daq_device) and the channel (pin1 and pin2) to simulate external signals. The test waits for a specified duration (6 seconds) to observe multiple toggles of pin1 by the mock external signal generator. After the observation period, the test reads the final state of pin1 using the read_digital method of the mock DAQ device.
It prints the final state of pin1 to the console to indicate whether the simulated external signal generation process affected the state of pin1 as expected.