dfttools Force Instruction

Overview

The **dfttools** library provides a modular and extensible framework to control hardware forces such as voltage, current, resistance, and frequency. Similar to measurement instructions, **force instructions** apply specified values to hardware signals and rely on customizable callback functions to interface with the actual hardware.

All force instructions and configurations can be accessed via:

```
from dfttools import *
```

Callback functions receive hardware meta data (signal, reference, value) and **must return a tuple** indicating hardware availability and the actual value applied or measured. This design allows users to tailor the library to their specific hardware setup.

Key Features

- **Unified Global Context (`g`)** manages hardware state, output logs, and callback registrations.
- **Force Instructions (`VFORCE`, `AFORCE`, `RESFORCE`, `FREQFORCE`)** to apply voltage, current, resistance, and frequency forces.
- **Meta-driven Callback System**: Callbacks receive meta data and return `(hardware_available, value)`.
- **Output Logging**: All force operations are logged for traceability.
- **Extensible and Customizable**: Users implement hardware-specific callbacks.

Code Structure and Analysis

1. Global Context (`glob/__init__.py`)

```
class GlobalContext:
    """Global context for managing hardware availability and callback
functions."""
    def __init__(self):
        self.output = []
        self.instructions = {}
        self.dut_description = None

        # Hardware availability flags
        self.voltage_force_hardware_available = False
        self.current_force_hardware_available = False
        self.resistance_force_hardware_available = False
        self.frequency_force_hardware_available = False
```

```
        # Callback functions for force and measurement
        self.hardware_callbacks = {
            'voltage_force': None,
            'current_force': None,
            'resistance_force': None,
            'frequency_force': None,
            'voltage_measure': None,
            'current_measure': None,
            'resistance_measure': None,
            'frequency_measure': None,
            'voltage_force_sweep': None,
            'current_force_sweep': None,
            'resistance_force_sweep': None,
            'frequency_force_sweep': None,
            'i2c_read': None,
            'i2c_write': None,
        }

    @property
    def callback_keys(self):
        return self.hardware_callbacks.keys()

g = GlobalContext()
```

Analysis: This global context object (`g`) holds the callback registry and tracks hardware availability flags for force and measurement operations. It also stores output logs of all instructions executed.

---

2. Callback Functions for Force (`callbacks/force_callbacks.py`)

Example callback implementations simulating hardware interaction:

```
def voltage_force_callback(g, signal, reference, value):
    force_hardware_available = True  # Hardware presence flag
    measured_value = 3.295  # Example applied voltage value
    return force_hardware_available, measured_value

def current_force_callback(g, signal, reference, value):
    force_hardware_available = True
    measured_value = 3.295  # Example applied current value
    return force_hardware_available, measured_value

def resistance_force_callback(g, signal, reference, value):
    force_hardware_available = True
    measured_value = 1000  # Example applied resistance value
    return force_hardware_available, measured_value

def frequency_force_callback(g, signal, reference, value):
    force_hardware_available = True
```

```
    measured_value = 100  # Example applied frequency value
    return force_hardware_available, measured_value
```

Analysis: These callbacks must be customized to communicate with your hardware. They receive the global context, signal, reference, and the force value to apply. They return whether the hardware is available and the actual value applied.

---

3. Hardware Meta Function (`hardware/force.py`)

```
def apply_force_and_measure(g, signal, reference, value, force_type):
    # Check if callback is registered for the force_type
    if g.hardware_callbacks[force_type]:
        hardware_available, measured_value = g.hardware_callbacks[force_type](g,
signal, reference, value)
        if hardware_available:
            return hardware_available, measured_value
        return False, 0
    return False, 0
```

Analysis: This function abstracts the interaction between instructions and hardware callbacks. It dispatches the force command to the appropriate callback and returns the hardware availability and measured/applied value.

---

4. Force Instructions (`instructions/force.py`)

```
from dfttools.glob import g
from dfttools.hardware.force import apply_force_and_measure

def VFORCE(signal: str = 'VCC', reference: str = 'GND', value: float = 0.0):
    hardware_available, measured_value = apply_force_and_measure(g, signal,
reference, value, 'voltage_force')
    if not hardware_available:
        return {'signal': signal, 'reference': reference, 'value': value}
    g.output.append({'type': 'FORCE', 'signal': signal, 'reference': reference,
'value': value})
    return measured_value

def AFORCE(signal: str = 'VCC', reference: str = 'GND', value: float = 0.0):
    hardware_available, measured_value = apply_force_and_measure(g, signal,
reference, value, 'current_force')
    if not hardware_available:
        return {'signal': signal, 'reference': reference, 'value': value}
    g.output.append({'type': 'FORCE', 'signal': signal, 'reference': reference,
'value': value})
    return measured_value

def RESFORCE(signal: str = 'VCC', reference: str = 'GND', value: float = 0.0):
```

```
    hardware_available, measured_value = apply_force_and_measure(g, signal,
reference, value, 'resistance_force')
    if not hardware_available:
        return {'signal': signal, 'reference': reference, 'value': value}
    g.output.append({'type': 'FORCE', 'signal': signal, 'reference': reference,
'value': value})
    return measured_value

def FREQFORCE(signal: str = 'VCC', reference: str = 'GND', value: float = 0.0):
    hardware_available, measured_value = apply_force_and_measure(g, signal,
reference, value, 'frequency_force')
    if not hardware_available:
        return {'signal': signal, 'reference': reference, 'value': value}
    g.output.append({'type': 'FORCE', 'signal': signal, 'reference': reference,
'value': value})
    return measured_value
```

Analysis: Each force instruction applies a force of a given type (voltage/current/resistance/frequency) to a hardware signal and reference. If hardware is unavailable, it returns the requested value as-is. All successful operations are logged.

---

5. Example Usage

```
from dfttools import *

# Register force callbacks
g.hardware_callbacks = {
    'voltage_force': voltage_force_callback,
    'current_force': current_force_callback,
    'resistance_force': resistance_force_callback,
    'frequency_force': frequency_force_callback,
}

print("Testing Voltage Force:")
result_voltage = VFORCE(signal='VCC', value=1.1)
print(f"Voltage Force Result: {result_voltage}")

print("\nTesting Current Force:")
result_current = AFORCE(signal='VCC', reference='GND', value=1.0)
print(f"Current Force Result: {result_current}")
```

Analysis: This snippet shows how to register your hardware callbacks and apply voltage and current forces. The printed results demonstrate the applied values or fallback behavior if hardware is unavailable.

Customizable Power Supply Force Callbacks and Usage Examples for dfttools

Overview

This section provides **elaborated, real-world examples** of how to implement and use customizable callback functions for power supply operations (voltage force, current force, electronic loads, etc.) in the `dfttools` framework. These callbacks should interface with your actual hardware APIs or communication protocols (e.g., SCPI, serial, socket, REST API) and **must return a tuple**: `(hardware_exists, force_value)` where `hardware_exists` is a boolean indicating whether the hardware is accessible, and `force_value` is the value applied or measured.

Example: Power Supply Control Callbacks

Below are example callback implementations for voltage force, current force, and electronic load. Replace the simulated logic with your actual hardware communication code.

1. Voltage Force Callback

```
def voltage_force_callback(g, signal, reference, value):
    """
    Set the power supply voltage for a given channel (signal).
    Returns (hardware_exists, actual_voltage_set).
    """
    try:
        # Example: Replace with your hardware API call
        # Example SCPI: psu.write(f"VOLT {value}, (@{signal})")
        print(f"[PowerSupply] Setting voltage {value}V on channel {signal}")
        actual_voltage = value  # In real code, query the hardware for the set
value
        hardware_exists = True  # Set to False if hardware is not found or
communication fails
    except Exception as e:
        print(f"Error setting voltage: {e}")
        hardware_exists = False
        actual_voltage = 0.0
    return hardware_exists, actual_voltage
```

2. Current Force Callback

```
def current_force_callback(g, signal, reference, value):
    """
    Set the power supply current limit for a given channel (signal).
    Returns (hardware_exists, actual_current_set).
    """
    try:
        # Example: Replace with your hardware API call
        # Example SCPI: psu.write(f"CURR {value}, (@{signal})")
        print(f"[PowerSupply] Setting current {value}A on channel {signal}")
        actual_current = value  # In real code, query the hardware for the set
```

```
value
        hardware_exists = True
    except Exception as e:
        print(f"Error setting current: {e}")
        hardware_exists = False
        actual_current = 0.0
    return hardware_exists, actual_current
```

3. Electronic Load (Current Sink) Callback

```
def load_force_callback(g, signal, reference, value):
    """
    Set the electronic load to sink a specific current on a given channel.
    Returns (hardware_exists, actual_load_set).
    """
    try:
        # Example: Replace with your hardware API call
        # Example SCPI: eload.write(f"CURR {value}, (@{signal})")
        print(f"[E-Load] Setting load current {value}A on channel {signal}")
        actual_load = value  # In real code, query the hardware for the set value
        hardware_exists = True
    except Exception as e:
        print(f"Error setting load: {e}")
        hardware_exists = False
        actual_load = 0.0
    return hardware_exists, actual_load
```

Registering Callbacks

Register your callbacks for use in force instructions:

```
from dfttools.glob import g

g.hardware_callbacks['voltage_force'] = voltage_force_callback
g.hardware_callbacks['current_force'] = current_force_callback
g.hardware_callbacks['load_force'] = load_force_callback
```

Using Force Instructions

You can now use the force instructions as follows:

```
from dfttools import *

# Voltage force example
result = VFORCE(signal='CH1', value=5.0)
```

```
print(f"Voltage Force Result: {result}")

# Current force example
result = AFORCE(signal='CH1', reference='GND', value=1.0)
print(f"Current Force Result: {result}")

# Load force example (assuming you have added a LOADFORCE instruction)
result = LOADFORCE(signal='CH1', reference='GND', value=2.0)
print(f"Load Force Result: {result}")
```

Example: Adding a Custom Force Instruction

If you want to add a new force instruction (e.g., for a programmable load):

```
def LOADFORCE(signal: str = 'CH1', reference: str = 'GND', value: float = 0.0):
    hardware_available, measured_value = apply_force_and_measure(g, signal,
reference, value, 'load_force')
    if not hardware_available:
        return {'signal': signal, 'reference': reference, 'value': value}
    g.output.append({'type': 'FORCE', 'signal': signal, 'reference': reference,
'value': value})
    return measured_value
```

Notes on Customization

- **Meta Data**: You can expand the callback signature to include more meta data (e.g., ranges, modes, protection settings) as needed.
- **Error Handling**: Always catch exceptions and return `hardware_exists = False` if the hardware is not accessible.
- **Hardware Abstraction**: For real hardware, use the vendor's Python API, SCPI commands via VISA/pyvisa, serial communication, or network sockets as appropriate.
- **Testing**: For simulation or testing, callbacks can simply print actions and return the requested value.

---

customize the call back functions with powersupply force sweep and read the values back and return.

Certainly! Below is a detailed example showing how to customize **callback functions for power supply force sweeps** in the `dfttools` framework. The callbacks will:

- Apply each sweep step value (voltage or current) to the power supply.
- Read back the actual value from the hardware.
- Return the hardware availability and the measured value.

Custom Power Supply Force Sweep Callbacks with Readback

1. Simulated Hardware Interface Functions

Replace these with your actual hardware API or communication code.

```
import time

def set_power_supply_voltage(channel, voltage):
    print(f"[Hardware] Setting voltage {voltage:.3f} V on channel {channel}")
    time.sleep(0.05)  # Simulate hardware delay
    return True  # Return True if successful

def read_power_supply_voltage(channel):
    # Simulate reading voltage back from hardware
    measured_voltage = 3.3  # Replace with actual read command
    print(f"[Hardware] Reading voltage {measured_voltage:.3f} V from channel
{channel}")
    return measured_voltage

def set_power_supply_current(channel, current):
    print(f"[Hardware] Setting current {current:.3f} A on channel {channel}")
    time.sleep(0.05)
    return True

def read_power_supply_current(channel):
    measured_current = 1.2  # Replace with actual read command
    print(f"[Hardware] Reading current {measured_current:.3f} A from channel
{channel}")
    return measured_current
```

---

2. Customized Sweep Callback Functions

```
def voltage_force_sweep_callback(g, signal, reference, value):
    """
    Callback for voltage force sweep step.
    Sets voltage and reads back actual voltage.
    """
    hardware_available = set_power_supply_voltage(signal, value)
    if not hardware_available:
        return False, 0.0
    measured_value = read_power_supply_voltage(signal)
    return True, measured_value

def current_force_sweep_callback(g, signal, reference, value):
    """
```

```
    Callback for current force sweep step.
    Sets current and reads back actual current.
    """
    hardware_available = set_power_supply_current(signal, value)
    if not hardware_available:
        return False, 0.0
    measured_value = read_power_supply_current(signal)
    return True, measured_value
```

---

3. Register Callbacks in Global Context

```
from dfttools.glob import g

g.hardware_callbacks['voltage_force_sweep'] = voltage_force_sweep_callback
g.hardware_callbacks['current_force_sweep'] = current_force_sweep_callback
```

---

4. Example Usage of Sweep Functions

```
from dfttools import *

print("Starting Voltage Sweep:")
voltage_sweep_results = VFORCESWEEP(signal='CH1', reference='GND',
initial_value=0.0, end_value=5.0, step=1.0, step_time=0.1)
print("Voltage Sweep Results:", voltage_sweep_results)

print("\nStarting Current Sweep:")
current_sweep_results = AFORCESWEEP(signal='CH1', reference='GND',
initial_value=0.0, end_value=2.0, step=0.5, step_time=0.1)
print("Current Sweep Results:", current_sweep_results)
```

---

5. Output Example (Simulated)

```
Starting Voltage Sweep:
[Hardware] Setting voltage 0.000 V on channel CH1
[Hardware] Reading voltage 3.300 V from channel CH1
[Hardware] Setting voltage 1.000 V on channel CH1
[Hardware] Reading voltage 3.300 V from channel CH1
[Hardware] Setting voltage 2.000 V on channel CH1
[Hardware] Reading voltage 3.300 V from channel CH1
[Hardware] Setting voltage 3.000 V on channel CH1
[Hardware] Reading voltage 3.300 V from channel CH1
[Hardware] Setting voltage 4.000 V on channel CH1
[Hardware] Reading voltage 3.300 V from channel CH1
```

```
[Hardware] Setting voltage 5.000 V on channel CH1
[Hardware] Reading voltage 3.300 V from channel CH1
Voltage Sweep Results: [3.3, 3.3, 3.3, 3.3, 3.3, 3.3]

Starting Current Sweep:
[Hardware] Setting current 0.000 A on channel CH1
[Hardware] Reading current 1.200 A from channel CH1
[Hardware] Setting current 0.500 A on channel CH1
[Hardware] Reading current 1.200 A from channel CH1
[Hardware] Setting current 1.000 A on channel CH1
[Hardware] Reading current 1.200 A from channel CH1
[Hardware] Setting current 1.500 A on channel CH1
[Hardware] Reading current 1.200 A from channel CH1
[Hardware] Setting current 2.000 A on channel CH1
[Hardware] Reading current 1.200 A from channel CH1
Current Sweep Results: [1.2, 1.2, 1.2, 1.2, 1.2]
```

---

6. Customization Tips

- Replace the simulated `set_power_supply_*` and `read_power_supply_*` functions with your actual hardware API calls.
- Use your device's SDK, SCPI commands, or communication protocol (e.g., VISA, pyserial).
- Ensure proper error handling and return `(False, 0)` if hardware is not reachable or command fails.
- Adjust `step_time` to allow hardware to stabilize after each step.
- Log or store additional metadata as needed in `g.output`.

Summary Table

Callback Name	Purpose	Must Return
voltage_force_callback	Set power supply voltage	(hardware_exists, voltage)
current_force_callback	Set power supply current	(hardware_exists, current)
load_force_callback	Set electronic load current	(hardware_exists, load)

Example Output

```
[PowerSupply] Setting voltage 5.0V on channel CH1
Voltage Force Result: 5.0

[PowerSupply] Setting current 1.0A on channel CH1
Current Force Result: 1.0

[E-Load] Setting load current 2.0A on channel CH1
Load Force Result: 2.0
```

Summary

- **Force instructions** in dfttools allow applying voltage, current, resistance, and frequency to hardware signals.
- **Callbacks** receive meta data (`g`, `signal`, `reference`, `value`) and must return `(hardware_available, applied_value)`.
- The **global context** `g` manages callbacks, hardware availability flags, and logs all operations.
- Users must **customize callbacks** to interface with their specific hardware.
- The system is **modular, extensible, and traceable**, suitable for automated hardware testing and control.