Overview

dfttools is a modular Python framework for managing and automating hardware operations such as force application, measurement (voltage, current, resistance, frequency), and I2C communication. All instructions and configurations are accessible via:

```
from dfttools import *
```

The library is designed for flexibility: users can customize callback functions to interface with their specific hardware. These callbacks receive hardware meta data, and must return both the existence (availability) of the hardware and the requested value.

Key Features

- **Unified Import**: Access all instructions and configurations with `from dfttools import *`.
- **Global Context Management**: Centralized `GlobalContext` object (`g`) tracks hardware state, output, and callback functions.
- **Meta-driven Callbacks**: Callback functions receive meta data and must be tailored to the user's hardware. They return both availability and measured value.
- **Extensible Instructions**: Predefined instructions for force, measurement, sweeps, and I2C, easily extended for new hardware.
- **Traceable Output**: All measurement and operation results are logged in the global context for later analysis.

Installation

Clone the repository and install dependencies:

```
git clone &lt;repository_url&gt;
cd dfttools
pip install -r requirements.txt
```

Library Structure & Code Analysis

1. Root `__init__.py`

This file imports all instructions and callback modules, making them available for direct use:

```
from dfttools.instructions.force import *
from dfttools.instructions.meas import *
from dfttools.instructions.force_sweep import *
from dfttools.instructions.I2C import *
```

```
from dfttools.glob import g
from dfttools.callbacks.force_callbacks import *
from dfttools.callbacks.measure_callbacks import *
from dfttools.callbacks.force_sweep_callbacks import *
from dfttools.callbacks.i2c_callback import *
```

Analysis: This design enables users to access all high-level functions and configurations with a single import statement, simplifying usage and reducing boilerplate.

---

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

Defines the `GlobalContext` class, which holds:

- Output logs
- Registered instructions
- Device-under-test (DUT) description
- Hardware availability flags
- Callback function registry

```
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 hardware 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 context object is the backbone of the library. It stores all relevant state and provides a single point for callback registration and output collection, ensuring modularity and maintainability.

---

3. Measurement Instructions (`instructions/meas.py`)

Defines meta-driven measurement instructions:

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

def VMEASURE(signal: str = 'VCC', reference: str = 'GND', expected_value:
(int|float) = 0.0):
    """
    Measure voltage between a signal and a reference. Return expected value if
hardware is unavailable.
    """
    hardware_available, measured_value = apply_force_and_measure(g, signal,
reference, 'voltage_measure')
    if not hardware_available:
        return expected_value
    g.output.append({'type': 'MEASURE', 'signal': signal, 'reference': reference,
'measured_value': measured_value})
    return measured_value

# Similar functions: AMEASURE, RESMEASURE, FREQMEASURE
```

Analysis: Each instruction calls the relevant callback through `apply_force_and_measure`, passing meta data (signal, reference, etc.). If the hardware is unavailable, a default value is returned. All results are logged in the global context.

---

4. Hardware Meta Function (`hardware/measure.py`)

The meta function that routes measurement requests to the appropriate callback:

```
def apply_force_and_measure(g, signal, reference, force_type, *args, **kwargs):
    # Check hardware availability using the callback if defined
    if g.hardware_callbacks[force_type]:
        hardware_available, measured_value = g.hardware_callbacks[force_type](g,
signal, reference)
        if hardware_available:
            return hardware_available, measured_value
```

```
            return False, 0
    return False, 0
```

Analysis: This function is the abstraction layer between instructions and hardware. It ensures that every callback receives meta data and must return a tuple: (hardware_available, measured_value).

---

5. Callback Implementations (`callbacks/measure_callbacks.py`)

Example callback functions for measurement:

```
def voltage_measure_callback(g, signal, reference):
    measure_hardware_available = True
    measured_value = 3.3  # Simulated voltage measurement
    return measure_hardware_available, measured_value

def current_measure_callback(g, signal, reference):
    measure_hardware_available = True
    measured_value = 1.2  # Simulated current measurement
    return measure_hardware_available, measured_value

def resistance_measure_callback(g, signal, reference):
    measure_force_hardware_available = True
    measured_value = 1000  # Simulated resistance measurement
    return measure_force_hardware_available, measured_value

def frequency_measure_callback(g, signal, reference):
    measure_hardware_available = True
    measured_value = 50  # Simulated frequency measurement
    return measure_hardware_available, measured_value
```

Analysis: Callbacks are the user extension point. They must accept meta data and return (availability, value). Users should customize these functions to interface with their actual hardware.

---

6. Registering Callbacks and Using Instructions (`main.py`)

```
# Register callbacks in global context
g.hardware_callbacks['voltage_measure'] = voltage_measure_callback
g.hardware_callbacks['current_measure'] = current_measure_callback
g.hardware_callbacks['resistance_measure'] = resistance_measure_callback
g.hardware_callbacks['frequency_measure'] = frequency_measure_callback

print("Voltage Measurement:", VMEASURE(signal='VCC', reference='GND',
expected_value=3.5))
print("Current Measurement:", AMEASURE(signal='VCC', reference='GND',
expected_value=1.5))
print("Resistance Measurement:", RESMEASURE(signal='R1', reference='GND',
expected_value=1200))
```

```
print("Frequency Measurement:", FREQMEASURE(signal='CLK', reference='GND',
expected_value=60))
```

Analysis: This demonstrates how to register callbacks and perform measurements. The results show how the library can be used for automated hardware validation or simulation.

Customizing for Your Hardware

- **Callback Functions:** Write your own callback functions to interface with your hardware. Each must accept meta data (typically `g`, `signal`, `reference`, and others as needed) and return `(hardware_exists, value)`.
- **Registration:** Register your callbacks in the `g.hardware_callbacks` dictionary using the appropriate key.

Example: Customizing a Callback

```
def my_voltage_measure_callback(g, signal, reference):
    # Replace this logic with actual hardware communication
    hardware_exists = check_my_hardware(signal, reference)
    value = read_voltage_from_my_hardware(signal, reference) if hardware_exists
else 0
    return hardware_exists, value

g.hardware_callbacks['voltage_measure'] = my_voltage_measure_callback
```

Extending the Library

- Add new instructions to the `instructions` folder.
- Define new callbacks in the `callbacks` folder.
- Update the `hardware_callbacks` dictionary in `GlobalContext`.

Summary

- All dfttools instructions and configurations can be accessed with `from dfttools import *`.
- Callback functions receive hardware meta data and must be customized for your hardware.
- Each callback must return a tuple: (hardware_exists, value).
- The system is modular, extensible, and suitable for both simulation and real hardware integration.
