dfttools I2C Instructions

Overview

The **dfttools** library provides a flexible and extensible framework for hardware communication, including I2C read/write operations. The I2C instructions allow reading from and writing to I2C devices based on detailed register field information.

All I2C operations are meta-driven: they rely on **callback functions** registered in the global context (`g`) that interface with the actual hardware. These callbacks receive device and register addresses and must return appropriate read values or write success status.

Key Features

- **Meta-driven I2C Read/Write**: Operations use detailed field and register metadata to perform bitwise manipulation on register values.
- **Callback-based Hardware Interface**: Hardware communication is abstracted via user-customizable callbacks.
- **Support for Read-only Registers**: Write operations respect register attributes and skip read-only fields.
- **Graceful Fallbacks**: Returns expected values or failure flags if hardware callbacks are missing or hardware is unavailable.
- **Flexible Field Definitions**: Supports multi-register fields with masks, bit positions, and lengths.

Code Structure & Analysis

1. I2C Read Instruction (`instructions/i2c.py`)

```python
from dfttools.glob import g
from dfttools.hardware.i2c import apply_i2c_read_write

def I2C_READ(device_address: int, field_info: dict, expected_value: int):
    """
    Read data from an I2C device using the provided field information.

    Args:
        device_address (int): I2C device address.
        field_info (dict): Field metadata including registers, masks, lengths.
        expected_value (int): Default value if hardware unavailable.

    Returns:
        int: Combined read value or expected_value.
    """
    read_value = apply_i2c_read_write(g, device_address, field_info, 'read')
    if read_value is None:
```

```
            return expected_value
        return read_value
```

Analysis: This function delegates the actual reading to `apply_i2c_read_write`. If no hardware is available or the callback is missing, it returns a default expected value.

---

2. I2C Write Instruction (`instructions/i2c.py`)

```python
def I2C_WRITE(device_address: int, field_info: dict, write_value: int):
    """
    Write data to an I2C device using the provided field information.

    Args:
        device_address (int): I2C device address.
        field_info (dict): Field metadata including registers, masks, lengths.
        write_value (int): Value to write.

    Returns:
        bool: True if write succeeded, False otherwise.
    """
    hardware_available = g.hardware_callbacks.get('i2c_write', None)
    if not hardware_available:
        return False
    return apply_i2c_read_write(g, device_address, field_info, 'write',
write_value)
```

Analysis: Checks if the write callback is registered before attempting the write operation. Returns success status accordingly.

---

3. Core I2C Operation (`hardware/i2c.py`)

```python
def apply_i2c_read_write(g, device_address: int, field_info: dict, operation: str,
value: int = None):
    """
    Perform I2C read or write using field metadata.

    Args:
        device_address (int): I2C device address.
        field_info (dict): Field and register metadata.
        operation (str): 'read' or 'write'.
        value (int): Value to write if operation is 'write'.

    Returns:
        int or bool: Read value for 'read', success status for 'write', or
None/False if failed.
    """
    if operation == 'read':
```

```
        read_data = []
        for register in field_info['registers']:
            reg_addr = int(register['REG'], 16)
            mask = int(register['Mask'], 16)
            length = register['Length']

            callback_key = 'i2c_read'
            if g.hardware_callbacks.get(callback_key):
                read_byte = g.hardware_callbacks[callback_key](device_address,
reg_addr)
            else:
                return None

            field_value = (read_byte &amp; mask) &gt;&gt; register['FieldLSB']
            read_data.append(field_value)

        combined_value = 0
        for i, field_value in enumerate(read_data):
            combined_value |= field_value &lt;&lt; (i * field_info['registers'][i]
['Length'])

        return combined_value

    elif operation == 'write':
        field_values = []
        remaining_value = value
        write_allowed = True

        for register in field_info['registers']:
            length = register['Length']
            mask = int(register['Mask'], 16)
            attribute = register.get('Attribute', '')

            if 'RR' in attribute:
                print(f"Register {register['RegisterName']} is read-only. Skipping
write operation.")
                write_allowed = False
                continue

            field_value = (remaining_value &gt;&gt; (len(field_values) * length))
&amp; ((1 &lt;&lt; length) - 1)
            field_values.append(field_value)
            remaining_value &amp;= ~(mask &lt;&lt; register['FieldLSB'])

        if not write_allowed:
            return False

        for i, register in enumerate(field_info['registers']):
            if 'RR' in register.get('Attribute', ''):
                continue

            reg_addr = int(register['REG'], 16)
            callback_key = 'i2c_write'
            if g.hardware_callbacks.get(callback_key):
```

```
                success = g.hardware_callbacks[callback_key](device_address,
reg_addr, field_values[i])
                if not success:
                    return False
            else:
                return False

        return True

    return None
```

Analysis:

- **Read Operation:** Reads each register byte using the registered `i2c_read` callback, applies masks and shifts to extract field bits, and combines multiple register fields into a single integer.
- **Write Operation:** Splits the write value into field values per register, respects read-only attributes (`RR`), and writes each field via the `i2c_write` callback.
- Returns `None` or `False` if callbacks are missing or operations fail.

---

4. Example Callback Implementations (`callbacks/i2c_callback.py`)

```
def i2c_read_callback(device_address: int, register_address: int):
    # Simulated read: always returns 0xFF
    return 0xFF

def i2c_write_callback(device_address: int, register_address: int, value: int):
    # Simulated write: print and return success
    print(f"Writing {value} to device {device_address}, register
{register_address}")
    return True
```

---

5. Sample Field Metadata

```
field_info1 = {
    "fieldname": "tdm_bclk_osr",
    "length": 2,
    "registers": [
      {
        "REG": "0x00",
        "POS": 6,
        "RegisterName": "Config REG1",
        "RegisterLength": 8,
        "Name": "tdm_bclk_osr[1:0]",
        "Mask": "0xC0",
        "Length": 2,
        "FieldMSB": 1,
        "FieldLSB": 0,
```

```
            "Attribute": "NNNNNNNN",
            "Default": "00",
            "User": "000YYYYY",
            "Clocking": "FRO",
            "Reset": "C",
            "PageName": "PAG0"
        }
    ]
}

field_info2 = {
    "fieldname": "vbat_meas",
    "length": 10,
    "registers": [
        {
            "REG": "0x31",
            "POS": 0,
            "RegisterName": "VBAT measurement reg 1",
            "RegisterLength": 8,
            "Name": "vbat_meas[9:8]",
            "Mask": "0x3",
            "Length": 2,
            "FieldMSB": 9,
            "FieldLSB": 8,
            "Attribute": "000000RR",  # Read-only bits
            "Default": "00",
            "User": "00YYYYYY",
            "Clocking": "REF",
            "Reset": "C",
            "PageName": "PAG0"
        },
        {
            "REG": "0x32",
            "POS": 0,
            "RegisterName": "VBAT measurement reg 2",
            "RegisterLength": 8,
            "Name": "vbat_meas[7:0]",
            "Mask": "0xFF",
            "Length": 8,
            "FieldMSB": 7,
            "FieldLSB": 0,
            "Attribute": "RRRRRRRR",  # Read-only bits
            "Default": "00",
            "User": "YYYYYYYY",
            "Clocking": "REF",
            "Reset": "C",
            "PageName": "PAG0"
        }
    ]
}
```

6. Register Callbacks and Test

```
g.hardware_callbacks = {
    'i2c_read': i2c_read_callback,
    'i2c_write': i2c_write_callback
}

print("I2C Read Results:", I2C_READ(device_address=0x12, field_info=field_info1,
expected_value=0x3))
print("I2C Write Results:", I2C_WRITE(device_address=0x12, field_info=field_info1,
write_value=0x3))

print("I2C Read Results:", I2C_READ(device_address=0x12, field_info=field_info2,
expected_value=0x3))
print("I2C Write Results:", I2C_WRITE(device_address=0x12, field_info=field_info2,
write_value=0x3))
```

---

7. Sample Output

```
I2C Read Results: 192
Writing 3 to device 18, register 0
I2C Write Results: True
I2C Read Results: 65280
Register VBAT measurement reg 1 is read-only. Skipping write operation.
Register VBAT measurement reg 2 is read-only. Skipping write operation.
I2C Write Results: False
```

Summary

- **I2C instructions** in dfttools use detailed register metadata to read/write bits across multiple registers.
- **Callbacks** `i2c_read` and `i2c_write` must be implemented to interface with actual hardware.
- **Read-only registers** are respected during writes to prevent illegal operations.
- The framework returns expected values or failure flags if hardware or callbacks are unavailable.
- This design supports complex I2C devices with multi-register fields and bit masks.

Customization Tips

- Replace simulated callbacks with real hardware communication code (e.g., via `pyserial`, `pyvisa`, or vendor SDKs).
- Extend `field_info` dictionaries to match your device datasheets.
- Implement error handling and retries in callbacks for robustness.
- Use the output logs (`g.output`) for debugging and traceability.

Customizable Callback Examples

1. Basic I2C Read Callback (Simulated)

```python
def i2c_read_callback(device_address: int, register_address: int):
    """
    Simulate reading a byte from an I2C device register.
    Replace this with actual hardware communication.
    """
    print(f"Reading from device 0x{device_address:X}, register
0x{register_address:X}")
    # Example: return a fixed pattern or random data
    return 0xFF
```

2. Basic I2C Write Callback (Simulated)

```python
def i2c_write_callback(device_address: int, register_address: int, value: int):
    """
    Simulate writing a byte to an I2C device register.
    Replace this with actual hardware communication.
    """
    print(f"Writing value 0x{value:X} to device 0x{device_address:X}, register
0x{register_address:X}")
    # Return True if write succeeded
    return True
```

3. Real Hardware Example Using pyserial (Pseudo-code)

```python
import serial

ser = serial.Serial('/dev/ttyUSB0', 115200, timeout=1)

def i2c_read_callback(device_address: int, register_address: int):
    # Construct your device-specific command to read register
    cmd = f"READ {device_address} {register_address}\n".encode()
    ser.write(cmd)
    response = ser.readline()
    # Parse response to integer
    return int(response.strip(), 16)

def i2c_write_callback(device_address: int, register_address: int, value: int):
    cmd = f"WRITE {device_address} {register_address} {value}\n".encode()
    ser.write(cmd)
    response = ser.readline()
    return response.strip() == b'OK'
```

Using Field Metadata for Complex Registers

Field info dictionaries describe how bits are distributed across multiple registers, including masks and read-only attributes:

```
field_info = {
  "fieldname": "example_field",
  "length": 10,
  "registers": [
    {
      "REG": "0x10",
      "Mask": "0x03",
      "Length": 2,
      "FieldLSB": 0,
      "Attribute": "NN"
    },
    {
      "REG": "0x11",
      "Mask": "0xFF",
      "Length": 8,
      "FieldLSB": 2,
      "Attribute": "RR"  # Read-only
    }
  ]
}
```

The `apply_i2c_read_write` function reads/writes each register field respecting masks and attributes. For example, it skips writing to read-only registers.

Advanced Usage Examples

Reading a Multi-Register Field

```
value = I2C_READ(device_address=0x12, field_info=field_info, expected_value=0)
print(f"Read multi-register field value: {value}")
```

Writing a Value with Read-Only Registers

```
success = I2C_WRITE(device_address=0x12, field_info=field_info, write_value=0x3FF)
print(f"Write success: {success}")
```

If any register is read-only, the write operation skips it and may return `False` if writes are disallowed.

Integration Tips

- **Combine with real hardware APIs:** Use vendor SDKs, Linux i2c-dev, or embedded drivers to implement callbacks.
- **Handle multi-byte reads/writes:** For devices requiring register address write before read, implement this in your `i2c_read` callback.
- **Error handling:** Return `None` or `False` on communication failure to trigger fallback logic.
- **Logging:** Use print/debug statements in callbacks to trace I2C transactions.