IMU Exercise Appendix I: Code Snippets

This appendix contains some helper functions to help build the IMU component as part of our lab exercise. Each is expected to be defined as a member function within the IMU component.

`setupRegisterRead` : Sets up the IMU to know where to read the next value from

This function performs the write to the IMU to tell it where the next read should pull from.

Key Features:

- Wrap buffer around single byte
- Write to I2C bus

```
/**
 * \brief sets up the IMU to know what register the next read should be from
 *
 * The MPU-6050 requires a write call with a register's address before a read \
 * sets up that read address by writing it to the IMU via the I2C write port.
 *
 * \param reg: IMU internal address to the first register to be read
 * \return: I2C from the write call
 */
Drv::I2cStatus Imu ::setupReadRegister(U8 reg) {
    Fw::Buffer buffer(&reg, sizeof reg);
    return this->write_out(0, m_i2cDevAddress, buffer);
}
```

`readRegisterBlock` : Reads a register block from IMU

This function sets-up and performs a read of a block of registers from the IMU.

Key Features:

- Check IMU status (from `setupReadRegister` helper)
- Read data from I2C bus

```
/**
 * \brief reads a block of registers from the IMU
 *
 * This function starts by writing the startRegister to the IMU by passing it
 * the read port of the I2C bus to read data from the IMU. It will read `buffe
 * and as such the caller must set this up.
 *
 * \param startRegister: register address to start reading from
 * \param buffer: buffer to read into. Determines size of read.
 * \return: I2C status of transactions
 */
Drv::I2cStatus Imu ::readRegisterBlock(U8 startRegister, Fw::Buffer& buffer) {
    Drv::I2cStatus status;
    status = this->setupReadRegister(startRegister);
    if (status == Drv::I2cStatus::I2C_OK) {
        status = this->read_out(0, m_i2cDevAddress, buffer);
    }
    return status;
}
```

`deserializeVector` : Unpacks the data read from the IMU

This function takes the data from the IMU read and unpacks it into a 3-element buffer.

Key Features:

- Work with a modeled array type
- Use deserialization interface of `Fw::Buffer`

```
/**
 * \brief unpacks a buffer into a vector with scaled elements
 *
 * This will unpack data from buffer into a Gnc::Vector type by unpacking 3x I
 * scales each by dividing by the scaleFactor provided.
 *
 * \param buffer: buffer wrapping data, must contain at least 6 byes (3x I16)
 * \param scaleFactor: scale factor to divide each element by
 * \return initalized vector
 */
Gnc::Vector Imu ::deserializeVector(Fw::Buffer& buffer, F32 scaleFactor) {
    Gnc::Vector vector;
    I16 value;
    FW_ASSERT(buffer.getSize() >= 6, buffer.getSize());
```

```
    FW_ASSERT(buffer.getData() != nullptr);
    // Data is big-endian as is fprime internal storage so we can use the buil
    Fw::SerializeBufferBase& deserializeHelper = buffer.getSerializeRepr();
    deserializeHelper.setBuffLen(buffer.getSize()); // Inform the helper what
    FW_ASSERT(deserializeHelper.deserialize(value) == Fw::FW_SERIALIZE_OK);
    vector[0] = static_cast<F32>(value) / scaleFactor;

    FW_ASSERT(deserializeHelper.deserialize(value) == Fw::FW_SERIALIZE_OK);
    vector[1] = static_cast<F32>(value) / scaleFactor;

    FW_ASSERT(deserializeHelper.deserialize(value) == Fw::FW_SERIALIZE_OK);
    vector[2] = static_cast<F32>(value) / scaleFactor;
    return vector;
}
```

`config` : Configures the IMU

This function calls the IMU twice with write commands. These commands configure the accelerometer and gyroscope.

Key Features:

- Wrap `Fw::Buffer` around a C++ array
- Send events

```
//! Configure the accelerometer and gyroscope
//!
void Imu ::config() {
    U8 data[IMU_REG_SIZE_BYTES * 2];
    Fw::Buffer buffer(data, sizeof data);

    // Set gyro range to +-250 deg/s
    data[0] = GYRO_CONFIG_ADDR;
    data[1] = 0;

    Drv::I2cStatus status = write_out(0, m_i2cDevAddress, buffer);
    if (status != Drv::I2cStatus::I2C_OK) {
        this->log_WARNING_HI_SetUpConfigError(status);
    }

    // Set accel range to +- 2g
    data[0] = ACCEL_CONFIG_ADDR;
    data[1] = 0;
    status =this-> write_out(0, m_i2cDevAddress, buffer);
```

```
    if (status != Drv::I2cStatus::I2C_OK) {
        this->log_WARNING_HI_SetUpConfigError(status);
    }
}
```

`power` : Switches IMU power state

This function turns on or off the IMU power state (sleep mode vs powered mode).

Key Features:

- Wrap `Fw::Buffer` around a C++ array
- Send events
- Work with modeled enum types

```
//! Turn power on/off of device
//!
void Imu ::power(PowerState powerState) {
    U8 data[IMU_REG_SIZE_BYTES * 2];
    Fw::Buffer buffer(data, sizeof data);

    // Check if already on/off
    if (powerState.e == m_power) {
        return;
    }

    data[0] = POWER_MGMT_ADDR;
    data[1] = (powerState.e == PowerState::ON) ? POWER_ON_VALUE : POWER_OFF_VA

    Drv::I2cStatus status = this->write_out(0, m_i2cDevAddress, buffer);
    if (status != Drv::I2cStatus::I2C_OK) {
        this->log_WARNING_HI_PowerModeError(status);
    } else {
        m_power = powerState.e;
        // Must configure at power on
        if (m_power == PowerState::ON) {
            config();
        }
    }
}
```

`updateAccel` : reads accelerometer data

This function will read accelerometer data from the IMU, sends the data as telemetry, and updates the local copy. **Note:** the `updateGyro` function is nearly identical and is this not provided.

Key Features:

- Wrap `Fw::Buffer` around a C++ array
- Send telemetry
- Work with modeled serializable types
- Send events

```
//! Read, telemeter, and update local compy of accelerometer data
//!
void Imu ::updateAccel() {
    U8 data[IMU_MAX_DATA_SIZE_BYTES];
    Fw::Buffer buffer(data, sizeof data);

    // Read a block of registers from the IMU at the accelerometer's address
    Drv::I2cStatus status = this->readRegisterBlock(IMU_RAW_ACCEL_ADDR, buffer

    // Check a successful read of 6 bytes before processing data
    if ((status == Drv::I2cStatus::I2C_OK) && (buffer.getSize() == 6) && (buff
        Gnc::Vector vector = deserializeVector(buffer, accelScaleFactor);

        m_accel.setvector(vector);
        m_accel.settime(this->getTime());
        m_accel.setstatus(Svc::MeasurementStatus::OK);

        this->tlmWrite_accelerometer(m_accel.getvector(), m_accel.gettime());
    } else {
        this->log_WARNING_HI_TelemetryError(status);
        m_accel.setstatus(Svc::MeasurementStatus::FAILURE);
    }
}
```