Custom Firmware Implementation for LSM6DSV16X IMU with Moteus C1

-Neel Adke

Goal

The goal of this project was to create a new firmware mode for the Moteus C1 motor controller that reads raw accelerometer and gyroscope data from the LSM6DSV16X IMU at the same time. The existing firmware only supported quaternion output or accelerometer-only mode, but our application needed direct access to all six axes of raw motion data for custom sensor fusion and motion analysis. The new mode needed to provide continuous 6-axis data at high enough rates for real-time robotics applications without causing performance issues or data loss. Reading the onboard encoder and MA600 encoder values connected on AUX2.

Summary

This report documents the development of a custom firmware mode for the Moteus C1 motor controller to simultaneously read raw accelerometer and gyroscope data from an LSM6DSV16X IMU. The implementation builds upon existing work which has added quaternion support to the moteus firmware. Our modifications create a new operating mode that provides direct access to all six axes of motion data (three accelerometer axes and three gyroscope axes). The resulting system provides continuous 6-axis motion data at 240 Hz with appropriate scaling for robotics applications.

Important links

- Moteus existing firmware: https://github.com/mjbots/moteus/blob/main/docs/reference.md
- Custom firmware with IMU support: https://github.com/otaviogood/moteus/tree/main/lib
- Sparkfun LSM6DSV16X: https://www.sparkfun.com/sparkfun-micro-6dof-imu-breakout-lsm6dsv16x-qwiic.html
- Moteus C1 : https://mjbots.com/products/moteus-c1
- MA600: https://mjbots.com/products/ma600-breakout

Existing Firmware Capabilities

The original moteus firmware by mjbots supports various position encoders via I2C and SPI interfaces. Otavio Good's fork extended this support to include the LSM6DSV16X IMU with two modes:

Type 3 (lsm6dsv16x): Quaternion mode that uses the IMU's internal Sensor Fusion Low Power (SFLP) algorithm to calculate orientation quaternions from gyroscope data. This mode outputs three quaternion components (X, Y, Z) in float16 format, with the W component calculated from the constraint that quaternion magnitude equals 1.

Type 4 (lsm6dsv16xAccel): Accelerometer-only mode that disables the gyroscope and outputs raw three-axis accelerometer data.

Our application requires simultaneous access to both raw accelerometer and raw gyroscope data for custom sensor fusion algorithms and motion analysis. The goal was to create a third mode that reads all six raw sensor values (three accelerometer, three gyroscope) reliably and continuously.

Architecture Overview

The implementation follows a layered architecture consistent with the existing moteus firmware design. Data flows from the IMU hardware through several processing stages before reaching the user's Python application.

At the hardware layer, the LSM6DSV16X IMU connects to the STM32G4 microcontroller via I2C on the AUX1 port. The firmware polls the IMU every 2 milliseconds through interrupt-driven I2C transactions. Raw sensor bytes are stored in local buffers, then parsed into 16-bit integer values and made available through the register interface. Python applications query these registers via CAN bus commands, receiving the processed data for analysis or control purposes.

Key Design Decisions

Direct register reading instead of FIFO: Rather than using the IMU's internal FIFO (First in First Out) buffer like the existing modes, we read directly from the sensor output registers. This eliminates FIFO overflow issues and simplifies the interrupt handler, reducing execution time from potentially 500+ microseconds to approximately 3 microseconds per poll cycle.

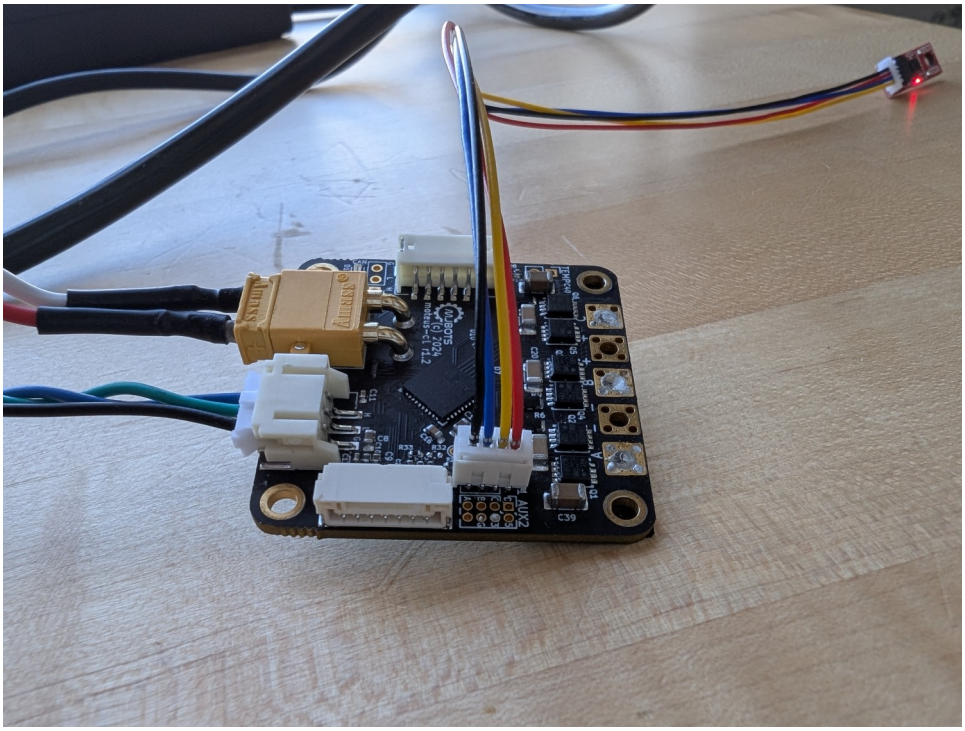
Alternating sensor reads: To maintain real-time performance, each interrupt cycle reads either gyroscope or accelerometer data, not both. At a 2 millisecond poll rate (500 Hz), this provides effective update rates of 250 Hz per sensor, which exceeds the IMU's configured 240 Hz output rate and ensures no samples are missed. If both the accelerometer and gyro values are queried together, the total interrupt time would be longer causing motor control issues.

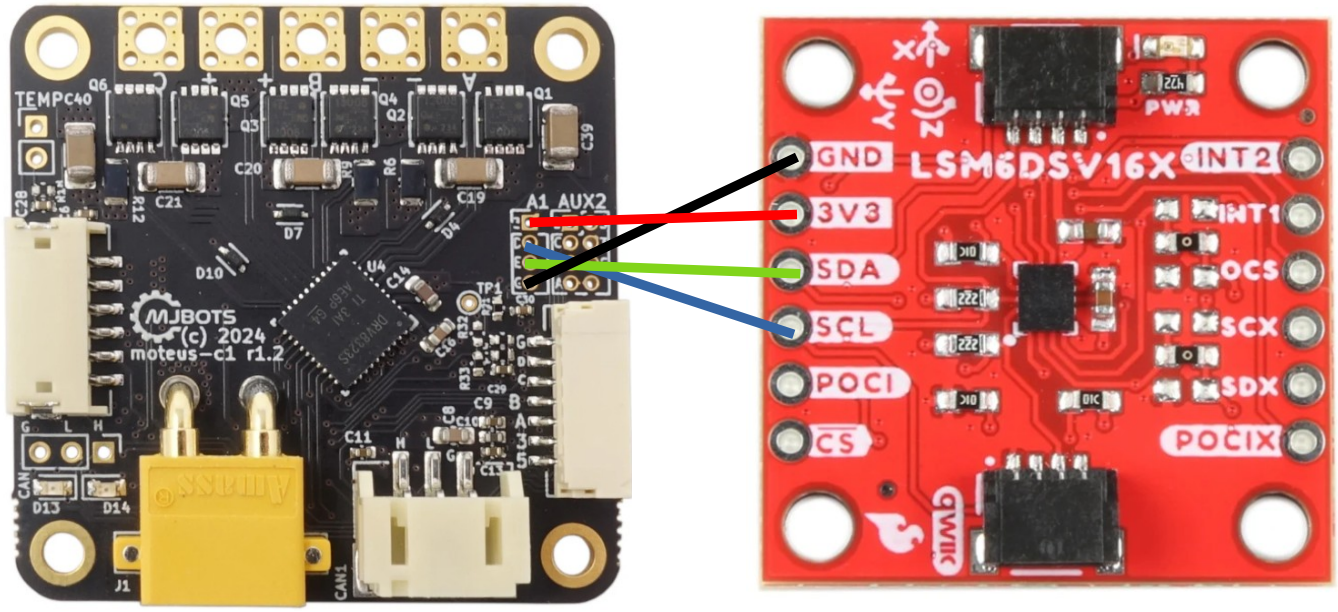
Register reuse: The existing quaternion registers (0x072, 0x073, 0x074) are repurposed for accelerometer data in the new mode. Three new registers (0x080, 0x081, 0x082) are allocated for gyroscope data.

Implementation Details

Hardware

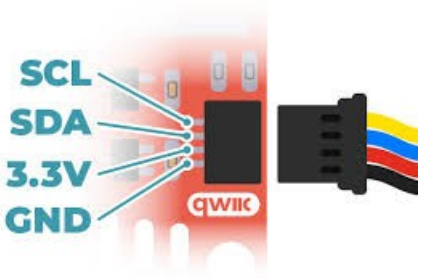
moteus c1	Con	AUX	SPI	ADC/Sin/Cos	I2C	HW Quad/PWM	UART	5VT
D		3			SCL	2.1	RX	X
E		4			SDA	2.2	TX	X





MA600 encoder

Qwiic connector:



Process

Step 1: Data Structure Extensions

File: fw/aux_common.h

```
  // Added new, gyro data structures
  int16_t gyro_x = 0;
  int16_t gyro_y = 0;
  int16_t gyro_z = 0;
```

```
    //Added new, gyro serialization
    //Exposes in telemetry, enables saving in config files and enables CAN reading
    a->Visit(MJ_NVP(gyro_x));
    a->Visit(MJ_NVP(gyro_y));
    a->Visit(MJ_NVP(gyro_z));
  }
};
```

Three new fields were added to the I2C device status structure to store gyroscope data. These use signed 16-bit integers (int16_t) to properly represent both positive and negative rotation rates. The fields were also added to the serialization function, making them visible to the telemetry system and accessible via CAN bus queries.

The accel_x, accel_y, and accel_z fields are reused for accelerometer data in the new mode. This dual-purpose design allows the same memory locations to serve different functions depending on the configured device type.

Step 2: Device Type Definition

File: fw/aux_common.h

```
struct I2C {
  struct DeviceConfig {
    enum Type {
      kNone,
      kAs5048,
      kAs5600,
      kLsm6dsv16x,
      kLsm6dsv16xAccel,
      kLsm6dsv16xRaw,     // Added new
      kNumTypes,
```

```
  static std::array<std::pair<T, const char*>, T::kNumTypes> map() {
    return {{
        { T::kNone, "none" },
        { T::kAs5048, "as5048" },
        { T::kAs5600, "as5600" },
        { T::kLsm6dsv16x, "lsm6dsv16x" },
        { T::kLsm6dsv16xAccel, "lsm6dsv16xAccel" },
        { T::kLsm6dsv16xRaw, "lsm6dsv16xRaw" }, //Added new, for tview
      }};
  }
};
```

A new enumeration value (kLsm6dsv16xRaw) was added to the device type list, assigned the value 5. This creates a unique identifier for the new mode that the firmware can recognize and route appropriately. A human-readable string mapping was also added, allowing configuration files and user interfaces to display "lsm6dsv16xRaw" instead of the numeric value 5.

Step 3: Register Definitions

File: fw/moteus_controller.cc

```
//Added new, gyro registers
kAux2GyroX = 0x080,
kAux2GyroY = 0x081,
kAux2GyroZ = 0x082,
```

```
case Register::kAux2GyroX:      // Added new
case Register::kAux2GyroY:      // Added new
case Register::kAux2GyroZ:      // Added new, wrtie only registers
```

Three new register addresses (0x080, 0x081, 0x082) were defined for gyroscope data. These registers are marked as read-only in the write function's switch statement, preventing invalid attempts to write to sensor data.

Step 4: IMU Initialization

File: fw/aux_port.h

```
    case DC::kLsm6dsv16xRaw: {          // ADD THIS CASE ISR_ParseI2c() called, Checks config.type Sees type = kLsm6dsv16xRaw Calls ISR_ParseLsm6dsv16xRaw(&status) Parses bytes into quat_x/y/z and gyro_x/y/z
      ISR_ParseLsm6dsv16xRaw(&status);
      break;
    }
```

```
void ISR_ParseLsm6dsv16xRaw(aux::I2C::DeviceStatus* status) {
  status->active = i2c_startup_complete_;

  status->nonce += 1;

  // Raw accelerometer data (int16, NOT float16)
  // Data comes in as X_low, X_high, Y_low, Y_high, Z_low, Z_high
  status->accel_x = (accel_raw_data_[1] << 8) | accel_raw_data_[0];  // Accel X
  status->accel_y = (accel_raw_data_[3] << 8) | accel_raw_data_[2];  // Accel Y
  status->accel_z = (accel_raw_data_[5] << 8) | accel_raw_data_[4];  // Accel Z

  // Raw gyroscope data (int16)
  // Data comes in as X_low, X_high, Y_low, Y_high, Z_low, Z_high
  status->gyro_x = (gyro_raw_data_[1] << 8) | gyro_raw_data_[0];  // Gyro X
  status->gyro_y = (gyro_raw_data_[3] << 8) | gyro_raw_data_[2];  // Gyro Y
  status->gyro_z = (gyro_raw_data_[5] << 8) | gyro_raw_data_[4];  // Gyro Z
}
```

```
      case DC::kLsm6dsv16xRaw: {
        if (!state.initialized) {
          if (InitLsm6dsv16xRaw(config)) state.initialized = true;
          return;
        }
        break;
      }
      case DC::kNone:
      case DC::kNumTypes: {
        MJ_ASSERT(false);
        break;
      }
    }
    // We can initialize at most one device per poll cycle.
    state.initialized = true;
    return;
  }
```

```
bool InitLsm6dsv16xRaw(const auto config) {
  // Make sure I2C is in a clean state
  while (i2c_->busy()) {
    i2c_->Poll();
  }
  // Check poll rate - must poll faster than IMU data rate
  // IMU at 240 Hz = 4.17ms period, we need < 2ms poll rate
  const int hz = 240;
  if (config.poll_rate_us >= 1000000 / 120 / 2) {
    DigitalOut db1_led(g_hw_pins.debug_led1, 0);
    return false;
  }

  // Select data rate based on hz setting
  uint8_t ctrl1 = 0x06;  // Default: 120 Hz
  switch (hz) {
    case 240: ctrl1 = 0x07; break;
    case 480: ctrl1 = 0x08; break;
    default: break;  // remain at 0x06 for 120Hz
  }

  // Configure accelerometer: 240Hz output rate
  i2c_->StartWriteMemory(config.address, 0x10, std::string_view(
      reinterpret_cast<const char*>(&ctrl1), 1));
  wait_i2c();

  // Set accelerometer full scale to ±16g
  uint8_t ctrl8 = 0x03;  // FS_XL = 11b for ±16g
  i2c_->StartWriteMemory(config.address, 0x17, std::string_view(
      reinterpret_cast<const char*>(&ctrl8), 1));
  wait_i2c();

  // Configure gyroscope: 240Hz output rate
  uint8_t ctrl2 = 0x07;  // Same rate as accel
  i2c_->StartWriteMemory(config.address, 0x11, std::string_view(
      reinterpret_cast<const char*>(&ctrl2), 1));
  wait_i2c();

  // Set gyroscope full scale to ±2000 dps
  uint8_t ctrl6 = 0x04;  // FS_G = 100b for ±2000 dps
  i2c_->StartWriteMemory(config.address, 0x15, std::string_view(
      reinterpret_cast<const char*>(&ctrl6), 1));
  wait_i2c();
  return true;
```

A new initialization function (InitLsm6dsv16xRaw) configures the LSM6DSV16X for dual-sensor raw data output. The function performs four I2C write operations to set up both sensors with appropriate parameters.

First, the polling rate is validated to ensure it exceeds twice the IMU's minimum output rate, satisfying the Nyquist criterion for proper sampling. The firmware fails initialization if the poll rate is too slow, preventing data loss.

The accelerometer is configured by writing 0x07 (00000111 in binary) to register 0x10 (CTRL1_XL) for 240 Hz output rate, and 0x03 to register 0x17 (CTRL8_XL) for plus/minus 16g full scale range. Similarly, the gyroscope receives 0x07 to register 0x11 (CTRL2_G) for 240 Hz output and 0x04 to register 0x15 (CTRL6) for plus/minus 2000 degrees per second range. This is based on the data sheet of the IMU which has pre-defined 8-bit binary values for the registers corresponding to various modes.

*Notably absent from this initialization are the embedded functions bank switching, SFLP sensor fusion enabling, and FIFO configuration steps present in Otavio's implementation. These features are unnecessary for raw data output and their omission reduces initialization time from approximately 5 milliseconds to 2 milliseconds.

Step 5: Data Reading Function

File: fw/aux_port.h

```
// Reads raw accelerometer and gyroscope data from LSM6DSV16X
// This function runs in interrupt context, so it must be FAST!
// We alternate between reading gyro and accel to keep each interrupt short.
void ReadRawIMUData(uint8_t address) {
  // Alternate between reading gyro and accel each cycle
  // This keeps interrupt time short (~3 microseconds per call)
  static bool read_gyro = true;

  if (read_gyro) {
    // Read raw gyroscope data from registers 0x22-0x27 (6 bytes)
    // These are OUTX_L_G, OUTX_H_G, OUTY_L_G, OUTY_H_G, OUTZ_L_G, OUTZ_H_G
    StartI2cRead(address, 0x22, gyro_raw_data_, 6);
  } else {
    // Read raw accelerometer data from registers 0x28-0x2D (6 bytes)
    // These are OUTX_L_A, OUTX_H_A, OUTY_L_A, OUTY_H_A, OUTZ_L_A, OUTZ_H_A
    StartI2cRead(address, 0x28, accel_raw_data_, 6);
  }

  // Toggle for next cycle
  read_gyro = !read_gyro;

  // Return immediately - don't wait for I2C to complete
  // The data will be ready on the next interrupt cycle
}
```

The ReadRawIMUData function implements non-blocking I2C reads from the IMU output registers. It uses a static boolean variable to alternate between reading gyroscope data (registers 0x22-0x27) and accelerometer data (registers 0x28-0x2D) on successive interrupt calls.

This alternating pattern is critical for maintaining real-time performance. Each I2C read transfers six bytes and completes in approximately 500 microseconds. By initiating only one read per interrupt and returning immediately without waiting, the interrupt handler executes in roughly 3 microseconds. The I2C transaction continues in the background, with results available by the next interrupt cycle.

At a 2 millisecond interrupt period, gyroscope data updates every 4 milliseconds (250 Hz effective rate) and accelerometer data updates on the alternate cycles (also 250 Hz). Both rates exceed the IMU's 240 Hz output rate, ensuring no samples are lost.

Step 6: Data Parsing Function

File: fw/aux_port.h

```
      case DC::kLsm6dsv16xRaw: {          // ADD THIS CASE ISR_ParseI2c() called, Checks config.type Sees type = kLsm6dsv16xRaw Calls
ISR_ParseLsm6dsv16xRaw(&status) Parses bytes into quat_x/y/z and gyro_x/y/z
        ISR_ParseLsm6dsv16xRaw(&status);
        break;
      }

      case DC::kNone:
      case DC::kNumTypes: {
        // Ignore.
        break;
      }
    }
  }
```

The ISR_ParseLsm6dsv16xRaw function assembles the six-byte buffers into usable 16-bit integer values. The LSM6DSV16X outputs data in little-endian (Little-endian is a computer architecture where the least significant byte of a multi-byte data word is stored at the lowest memory address**)** format, with the low byte followed by the high byte for each axis.

The parsing operation uses bit shifting and bitwise OR to combine bytes. For example, to assemble the gyroscope X value, the high byte is shifted left by 8 bit positions (equivalent to multiplying by 256), then combined with the low byte using bitwise OR. This produces the correct 16-bit signed integer representation of the sensor reading.

Both accelerometer and gyroscope data are parsed in the same function call, storing accelerometer values in accel_x/y/z and gyroscope values in the newly added gyro_x/y/z fields. The active flag and nonce counter are updated to indicate valid data and track update cycles.

Step 7: Integration and Routing

File: fw/aux_port.h

```
case DC::kLsm6dsv16xRaw: {
  ReadRawIMUData(config.address);
  break;
}
```

```
void ISR_ParseLsm6dsv16xRaw(aux::I2C::DeviceStatus* status) {
  status->active = i2c_startup_complete_;

  status->nonce += 1;

  // Raw accelerometer data (int16, NOT float16)
  // Data comes in as X_low, X_high, Y_low, Y_high, Z_low, Z_high
  status->accel_x = (accel_raw_data_[1] << 8) | accel_raw_data_[0];  // Accel X
  status->accel_y = (accel_raw_data_[3] << 8) | accel_raw_data_[2];  // Accel Y
  status->accel_z = (accel_raw_data_[5] << 8) | accel_raw_data_[4];  // Accel Z

  // Raw gyroscope data (int16)
  // Data comes in as X_low, X_high, Y_low, Y_high, Z_low, Z_high
  status->gyro_x = (gyro_raw_data_[1] << 8) | gyro_raw_data_[0];  // Gyro X
  status->gyro_y = (gyro_raw_data_[3] << 8) | gyro_raw_data_[2];  // Gyro Y
  status->gyro_z = (gyro_raw_data_[5] << 8) | gyro_raw_data_[4];  // Gyro Z
}
```

Three switch statements in the interrupt polling code were extended to handle the new device type. These route operations to the appropriate functions based on the configured device type.

The parsing switch directs incoming I2C data to ISR_ParseLsm6dsv16xRaw when type 5 is active. The initialization switch calls InitLsm6dsv16xRaw once at startup to configure the IMU. The reading switch invokes ReadRawIMUData on each interrupt cycle to continuously fetch fresh sensor data.

Two six-byte buffers (accel_raw_data_ and gyro_raw_data_) were added as class member variables to store the raw I2C data before parsing.

Step 8: Python Library Updates

File: lib/python/moteus/moteus.py

```
AUX2_GYROX = 0x080
AUX2_GYROY = 0x081
AUX2_GYROZ = 0x082
```

```
elif register == Register.AUX2_GYROX:
    return parser.read_int(resolution)
elif register == Register.AUX2_GYROY:
    return parser.read_int(resolution)
elif register == Register.AUX2_GYROZ:
    return parser.read_int(resolution)
```

Three register constant definitions were added for the gyroscope registers, following the existing naming convention. The parse_register function was extended with three additional cases to handle the new gyroscope registers, specifying that they should be decoded as signed integers.

These changes enable Python applications to query the gyroscope registers and receive properly formatted data. The parser cases are essential because without them, the Python library returns zeros for unknown registers.

Step 9: Port Selection

File: fw/moteus_controller.cc

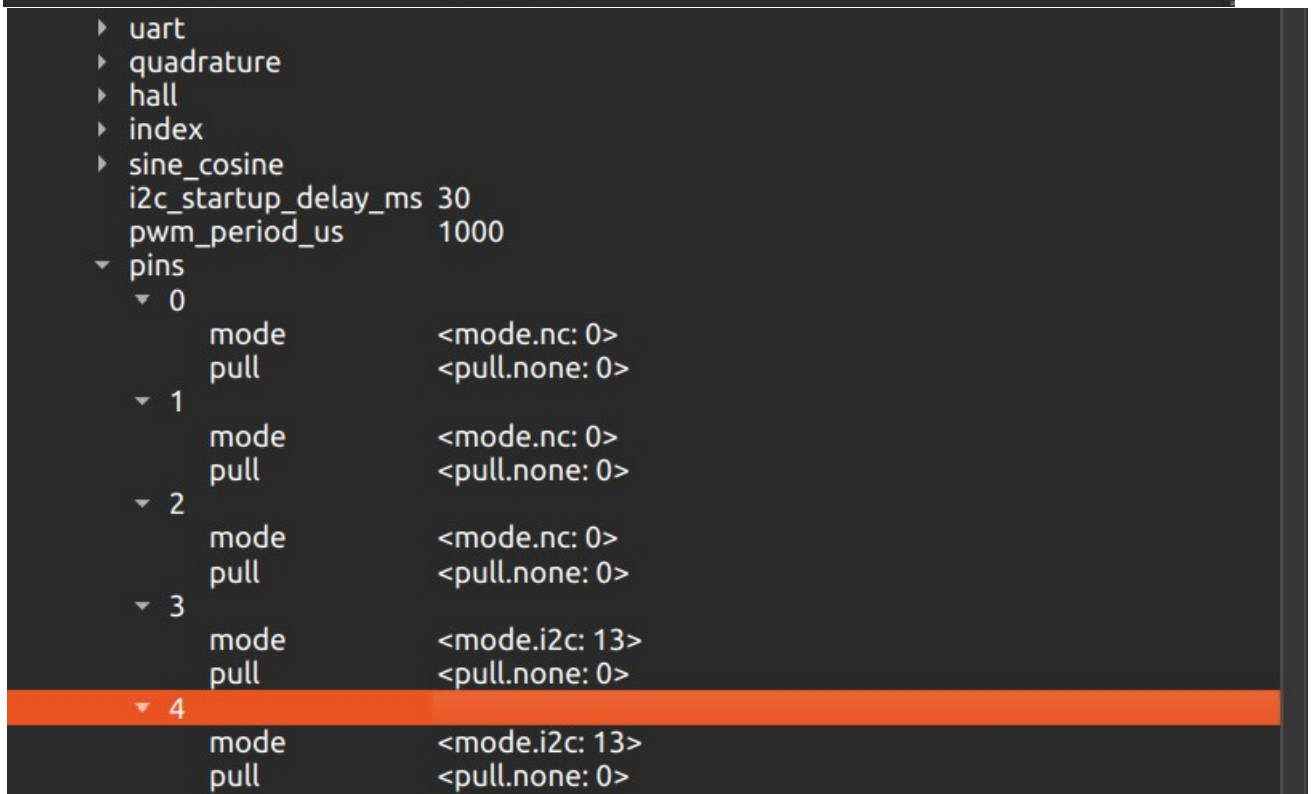
```
QuaternionValues ReadQuaternionAtomic() const {
  QuaternionValues result;

  __disable_irq();  // Disable interrupts to ensure atomic reads
  auto* status = const_cast<AuxPort&>(aux1_port_).status();     //change2: (aux2_port_) to (aux1_port_)
```

An important issue was discovered during testing. The ReadQuaternionAtomic and ReadGyroValue functions were hardcoded to read from aux2_port_, but the IMU was physically connected to AUX1. This caused Python queries to return zeros despite tview showing valid data, because tview accesses the raw status structure directly while Python goes through the register system.

The fix changed both functions to read from aux1_port_ instead, matching the actual hardware configuration. After this correction, Python scripts successfully received sensor data.

For flashing the firmware, please follow the steps; https://mjbots.github.io/moteus/reference/firmware/?h=flashing.

Tview settings



```
aux2
  i2c
  spi
    mode                    <mode.ma600: 5>
    rate_hz                 6000000
    filter_us               64
    bct                     0
    trim                    <trim.none: 0>
  uart
  quadrature
  hall
  index
  sine_cosine
  i2c_startup_delay_ms      30
  pwm_period_us             1000
  pins
    0
      mode                  <mode.spi: 1>
      pull                  <pull.none: 0>
    1
      mode                  <mode.spi: 1>
      pull                  <pull.none: 0>
    2
      mode                  <mode.spi: 1>
      pull                  <pull.none: 0>
    3
      mode                  <mode.spi_cs: 2>
      pull                  <pull.none: 0>
    4
      mode                  <mode.nc: 0>
      pull                  <pull.none: 0>
```

```
motor_position
  sources
    0
      aux_number            2
      type                  <type.spi: 1>
      i2c_device            0
      incremental_index     -1
      cpr                   65536
      offset                0.0
      sign                  1
      debug_override        -1
      timeout_s             0.20000000298023224
      reference             <reference.rotor: 0>
      pll_filter_hz         100.0
      compensation_table
      compensation_scale    0.0
    1
      aux_number            1
      type                  <type.spi: 1>
      i2c_device            0
      incremental_index     -1
      cpr                   16384
      offset                0.0
      sign                  1
      debug_override        -1
      timeout_s             0.20000000298023224
      reference             <reference.rotor: 0>
      pll_filter_hz         400.0
      compensation_table
      compensation_scale    0.0
```

Python script

readIMUma6.py

This script communicates with a moteus C1 motor controller over the CAN bus to read sensor and encoder data in real time. It uses the moteus Python API to send periodic query messages that request specific internal registers from the controller. A QueryResolution object is configured with custom register addresses corresponding to the IMU and encoder data. These include raw accelerometer registers at addresses 0x072 to 0x074, raw gyroscope registers at 0x080 to 0x082, the motor position register 0x001 for the external MA600 encoder, and the absolute position register 0x006 for the onboard AS5047P encoder. When controller.query() is called, the moteus C1 responds with the current values stored in these registers, which are returned as a dictionary indexed by register address.

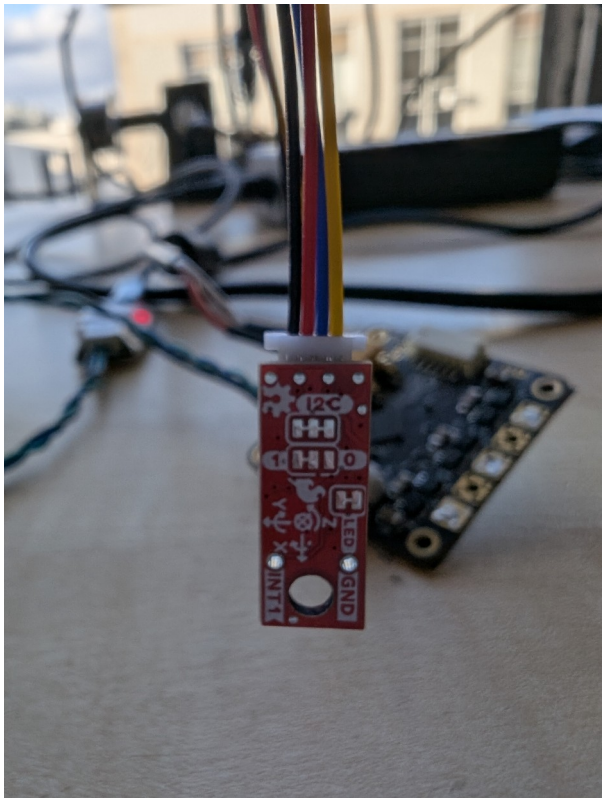
The IMU registers provide raw sixteen bit values from the LSM6DSV16X sensor configured in Type 5 mode on the AUX1 I2C interface. Since CAN queries return values as unsigned integers, the script converts them back into signed sixteen bit values so negative accelerations and angular rates are interpreted correctly. Scale factors based on the IMU configuration are then applied to convert raw counts into physical units, with accelerometer data expressed in meters per second squared and gyroscope data in degrees per second. The script also computes the magnitude of the acceleration vector from the three axes, which should be close to gravity when the system is not moving, helping verify that the IMU data is reasonable.

For position sensing, the script reads floating point values from two encoder related registers. The external MA600 encoder is connected over AUX2 SPI and configured as the primary motor position source, which is read from register 0x001. The onboard AS5047P encoder is connected over AUX1 SPI and provides an absolute shaft position through register 0x006. Both values are reported in revolutions by the controller firmware, so the script converts them into degrees before displaying them. All processed IMU and encoder values are printed at a fixed rate defined by the user, providing a clear and continuous view of motion and position data coming directly from the moteus C1.

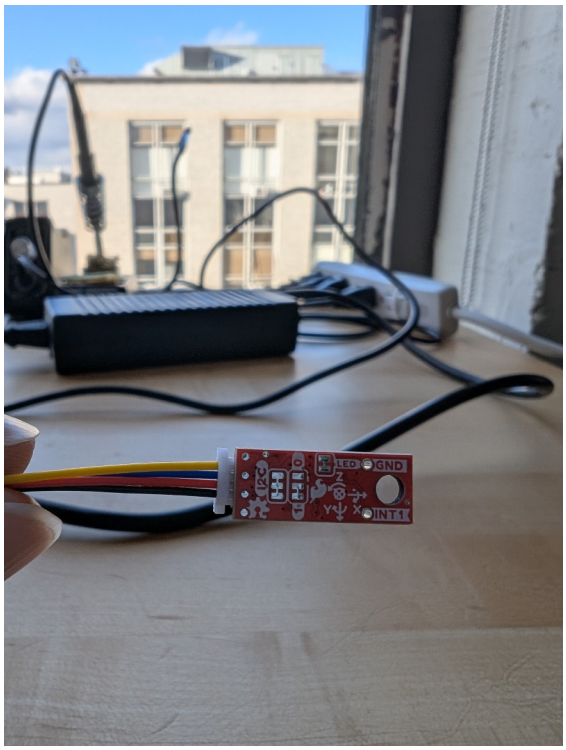
Results and Validation

The new firmware mode successfully streams all six raw IMU values at a stable rate, confirmed through tview where accelerometer and gyroscope readings updated continuously with a steadily increasing nonce counter. Movement produced the expected variations: accelerometer readings spiked with linear motion, and gyroscope values responded cleanly to rotation. When the device remained still, the accelerometer magnitude stayed close to one g or 9.8 m/s^2 and the gyroscope noise stayed within the normal small range, showing that the scaling and parsing were correct. Python scripts were able to read the new registers reliably and convert the raw values into physical units, with no freezing or overflow issues. To validate the results, the IMU was tested by orienting it facing the three axes and checking the terminal for the acceleration due to gravity.

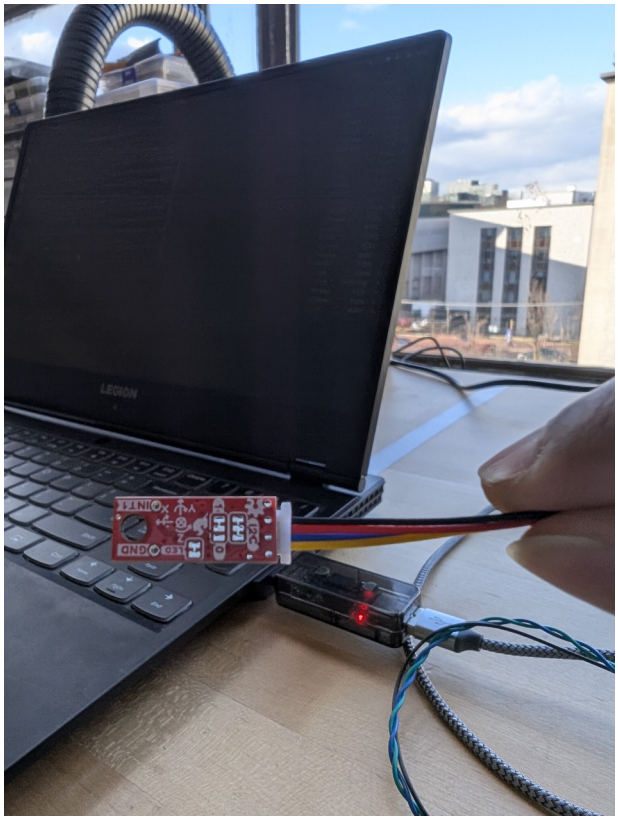
```
--------------------------------------------------------------------------
Accelerometer (m/s²): X:   9.835  Y:   0.689  Z:  -0.234  |Accel|: 9.861
Gyroscope (deg/s):    X:   -4.55  Y:    1.33  Z:   -2.03
--------------------------------------------------------------------------
```



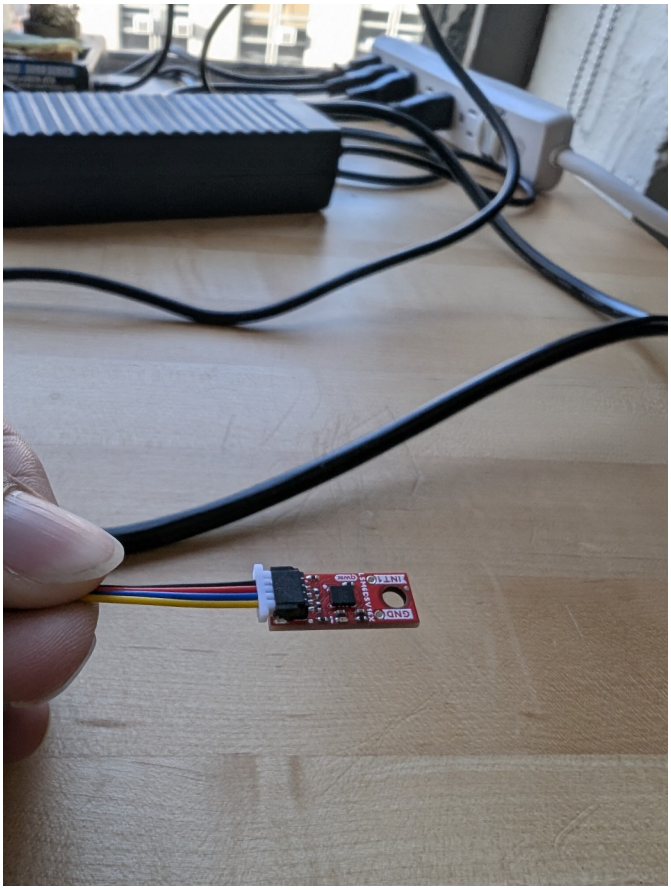
```
--------------------------------------------------------------------------
Accelerometer (m/s²): X:  -9.753  Y:  -0.416  Z:  -1.038  |Accel|: 9.817
Gyroscope (deg/s):    X:   -0.77  Y:   -3.08  Z:    0.42
--------------------------------------------------------------------------
```



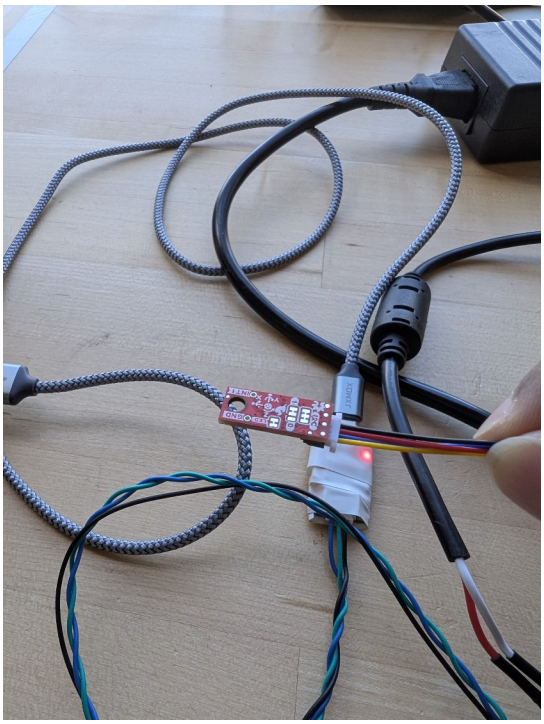

```
--------------------------------------------------------------------------
Accelerometer (m/s²): X:  -0.837  Y:  -9.533  Z:  -1.450  |Accel|: 9.679
Gyroscope (deg/s):    X:   -3.43  Y:   -2.03  Z:    1.89
--------------------------------------------------------------------------
```




```
--------------------------------------------------------------------------
Accelerometer (m/s²): X:   1.378  Y:   9.858  Z:   0.608  |Accel|: 9.973
Gyroscope (deg/s):    X:   -2.59  Y:   -0.49  Z:    0.28
--------------------------------------------------------------------------
```



```
------------------------------------------------------------------------
Accelerometer (m/s²): X:   -1.254  Y:   -0.775  Z:   -9.820  |Accel|: 9.930
Gyroscope (deg/s):    X:    -0.07  Y:    0.63  Z:    1.26
------------------------------------------------------------------------
```



```
------------------------------------------------------------------------
Accelerometer (m/s²): X:    1.029  Y:    0.067  Z:    9.729  |Accel|: 9.784
Gyroscope (deg/s):    X:    -1.33  Y:    -1.19  Z:    0.49
------------------------------------------------------------------------
```

Similarly, the gyroscope readings were validated by rotating the IMU around the three axes and checking the terminal output for changes in the respective values.

```
--------------------------------------------------------------------
Accelerometer (m/s²): X:   9.514  Y:   0.617  Z:   5.058  |Accel|: 10.793
Gyroscope (deg/s):    X:  135.73  Y:   32.90  Z:    3.85
--------------------------------------------------------------------
```

```
--------------------------------------------------------------------
Accelerometer (m/s²): X:   9.849  Y:  -4.278  Z:   3.025  |Accel|: 11.156
Gyroscope (deg/s):    X:    5.88  Y: -111.44  Z:  -51.59
--------------------------------------------------------------------
```

```
--------------------------------------------------------------------
Accelerometer (m/s²): X:   8.777  Y:  -2.857  Z:   1.345  |Accel|: 9.328
Gyroscope (deg/s):    X:   43.75  Y:  -10.29  Z: -104.16
--------------------------------------------------------------------
```

The values of both the encoders were tested with a magnet with output in degrees.

```
MA600 Encoder (deg):      Position:   55.59°
Onboard Encoder (deg):    Position:  348.92°
```