CAL POLY
Mechanical Engineering
CAL POLY

MEMORANDUM

12/14/2025

TO: Charlie Refvem, Mechanical Engineering Department, Cal Poly SLO
crefvem@calpoly.edu

FROM: Arturo Ramirez
arami339@calpoly.edu
Tarsem Pal
shpal@calpoly.edu
Billy Hawkins
whawkins@calpoly.edu

GROUP: Mecha 01
SUBJECT: Final Report

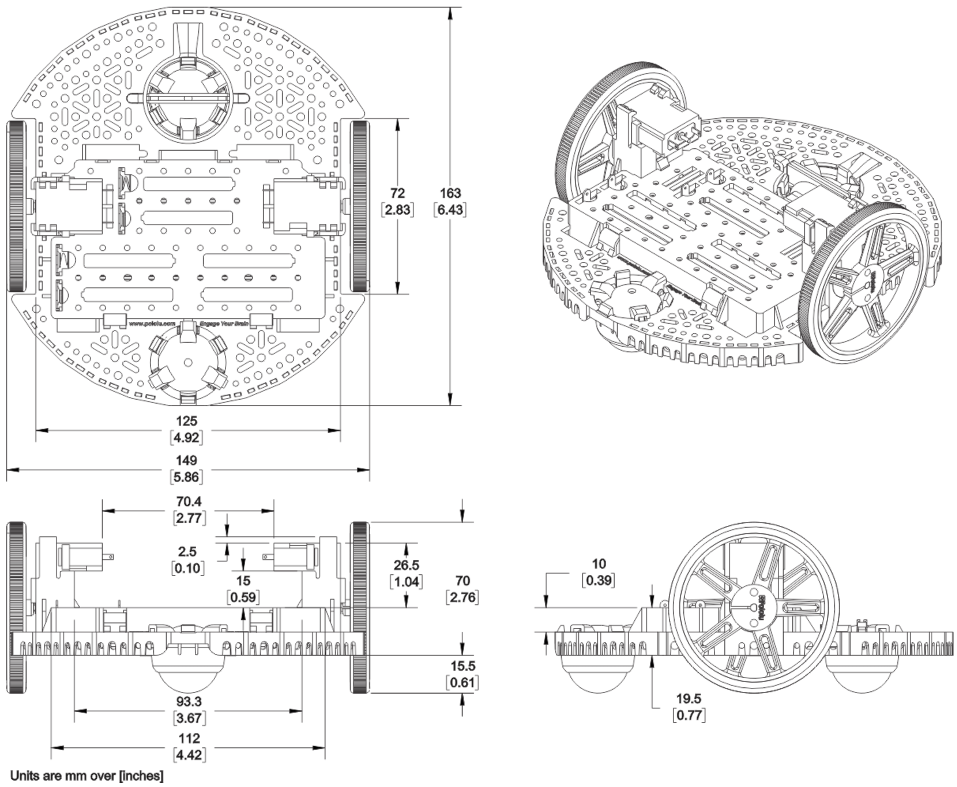
Overview

Hardware

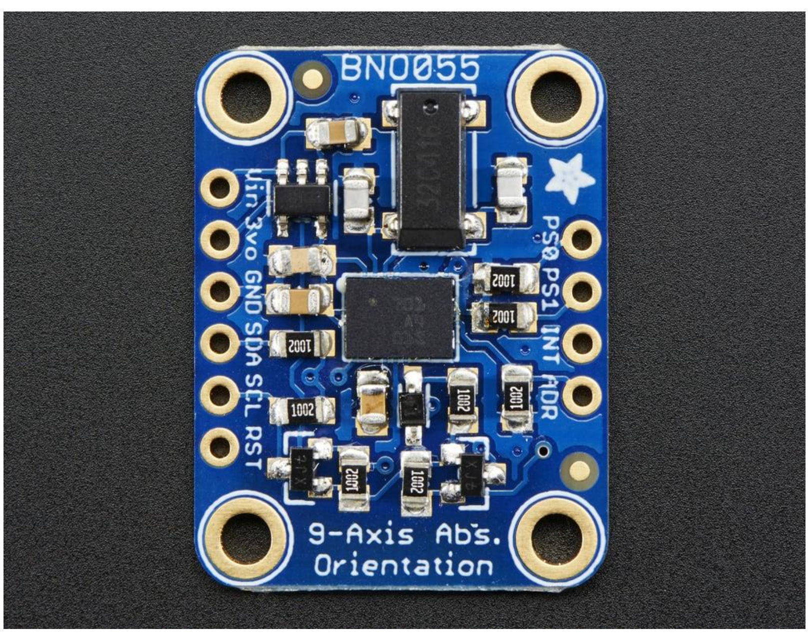
Chassis

Includes power distribution board and is from polulu. Also has motors, encoders, wheels

CAL POLY

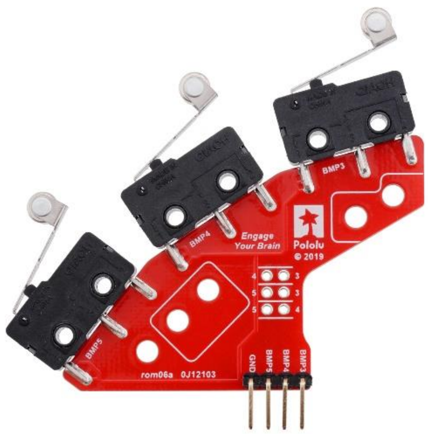

72
[2.83]
163
[6.43]
125
[4.92]
149
[5.86]
70.4
[2.77]
2.5
[0.10]
15
[0.59]
26.5
[1.04]
70
[2.76]
15.5
[0.61]
93.3
[3.67]
112
[4.42]
10
[0.39]
19.5
[0.77]
Units are mm over [inches]

IMU

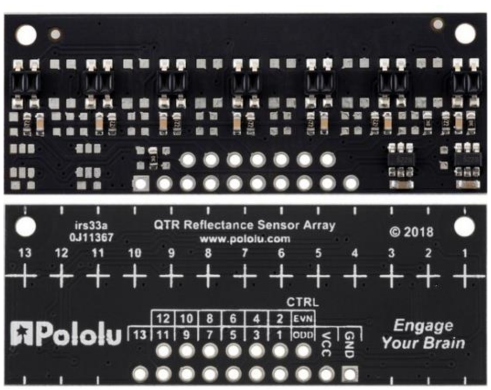

BN0055
Vin 3vo GND SDA SCL RST
PS0 PS1 INT ADR
9-Axis Abs.
Orientation

Sensors

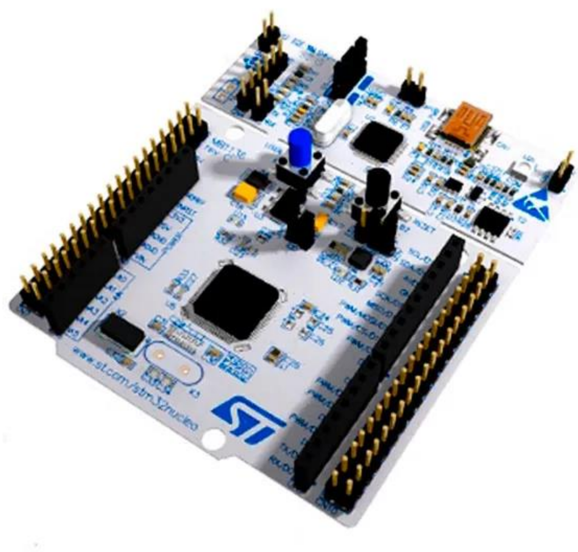
Bump sensor, it was shorted to have 1 output or whatever the documentation says to have less complexity and simply detrect any sort of bump



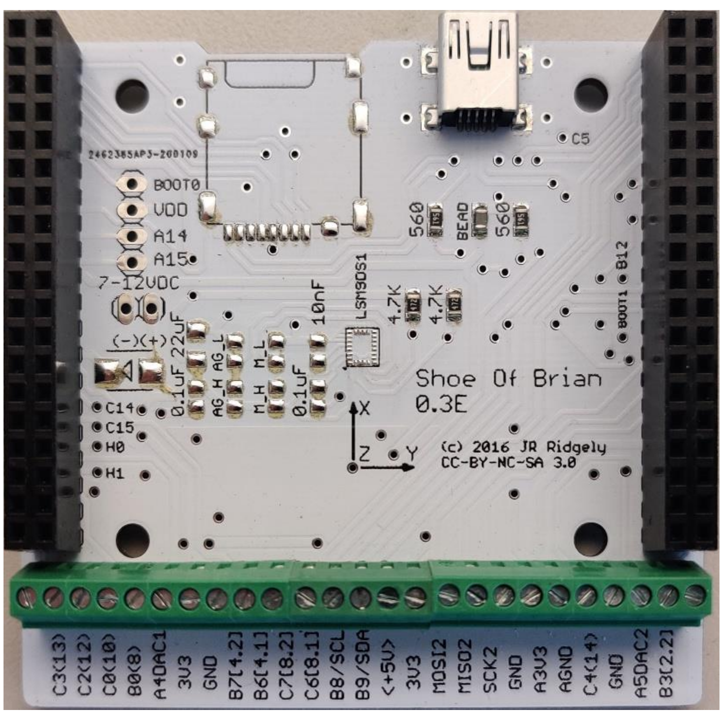
Ir Sensor



Nucleo Board, STM32L



Shoe of brian



Assembly

Wiring

Nucleo Board

Shoe of Brian

Task and State Diagrams

As part of our system design process, our team developed a comprehensive task diagram and a set of finite state machine (FSM) diagrams corresponding to each task. The task diagram outlines how data and control signals are shared among the serial communication, motor control, and data processing tasks, while the FSM diagrams capture the logical flow and state transitions within each task. These diagrams have been updated to match the most recent implementation and will continue to evolve as we refine the code and add new functionality in future labs.

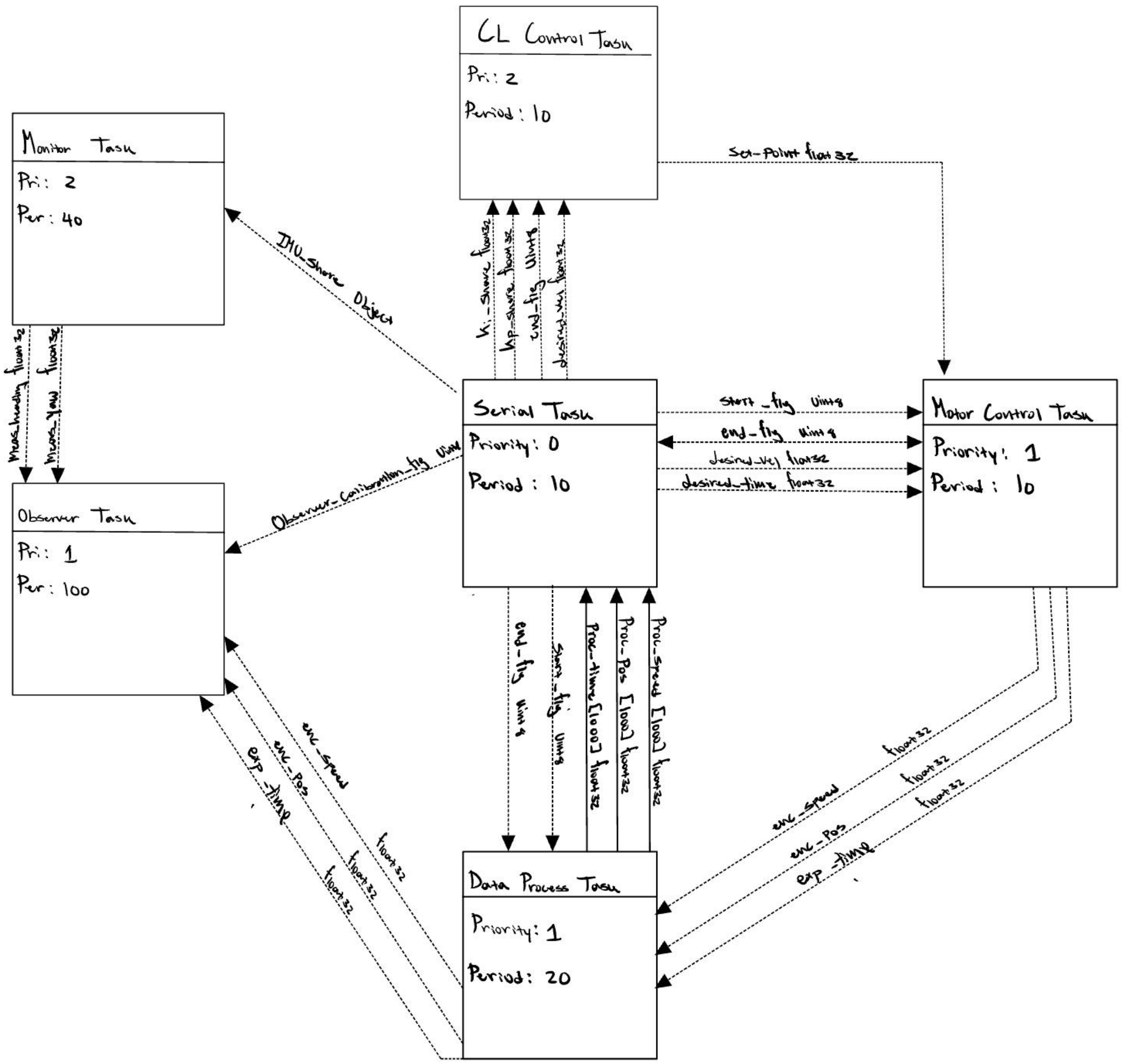


Figure 1: Task Diagram

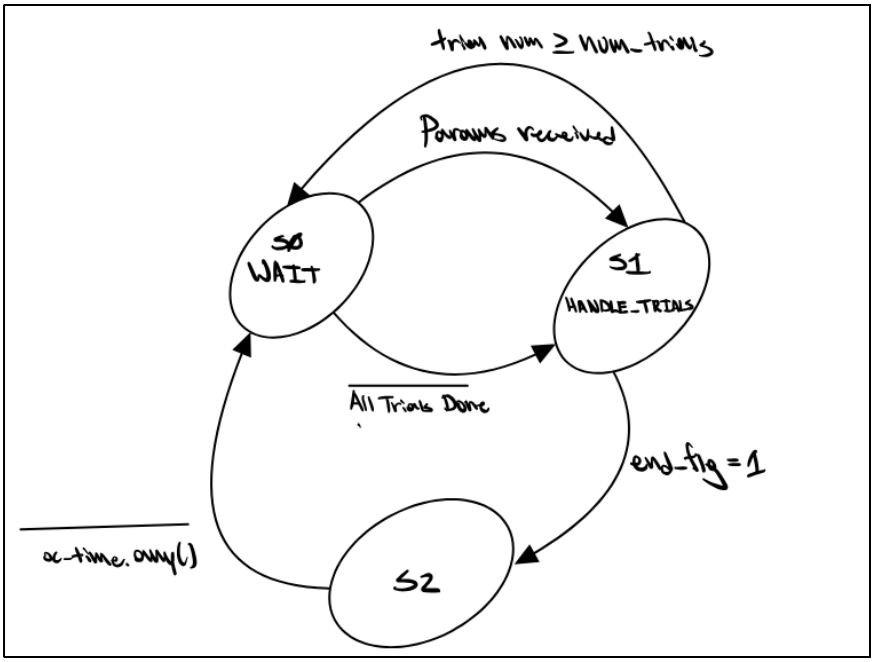


Figure 2: Serial Task FSM

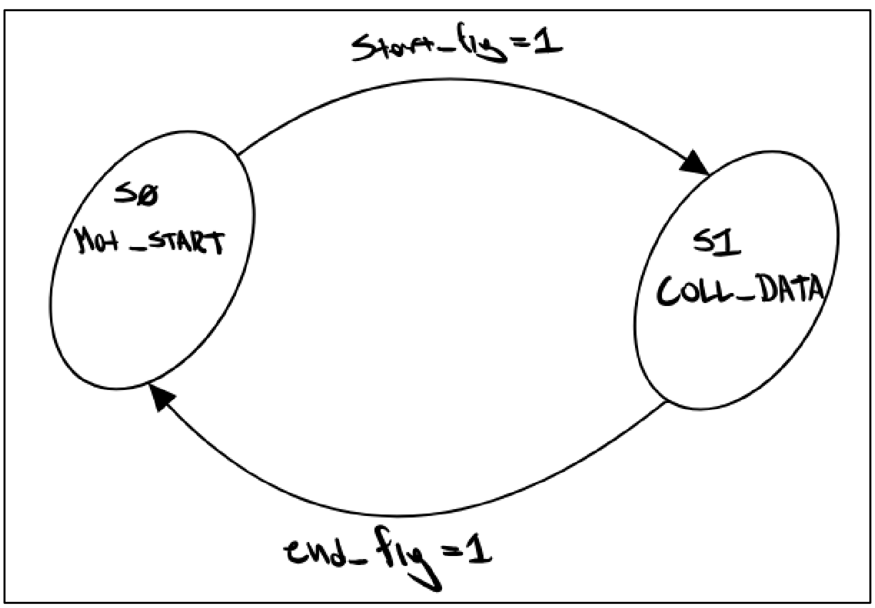


Figure 3: Motor Control FSM

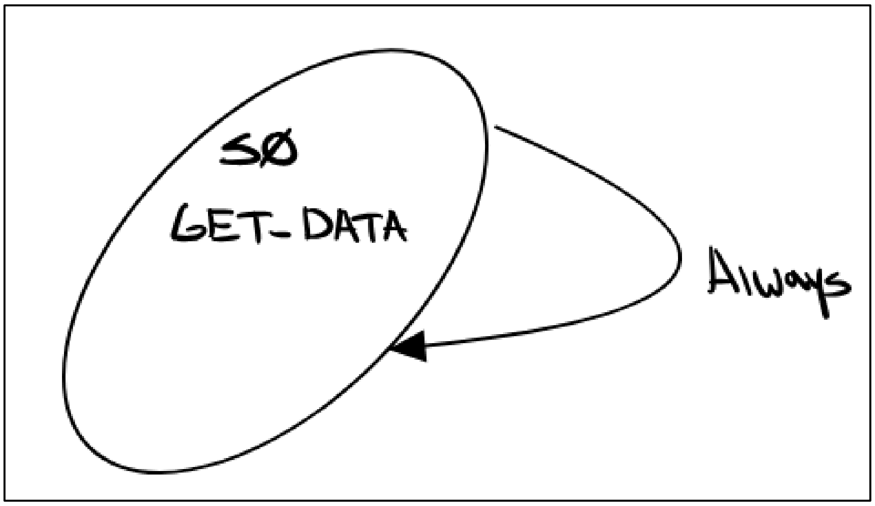


Figure 4: Data Processing FSM

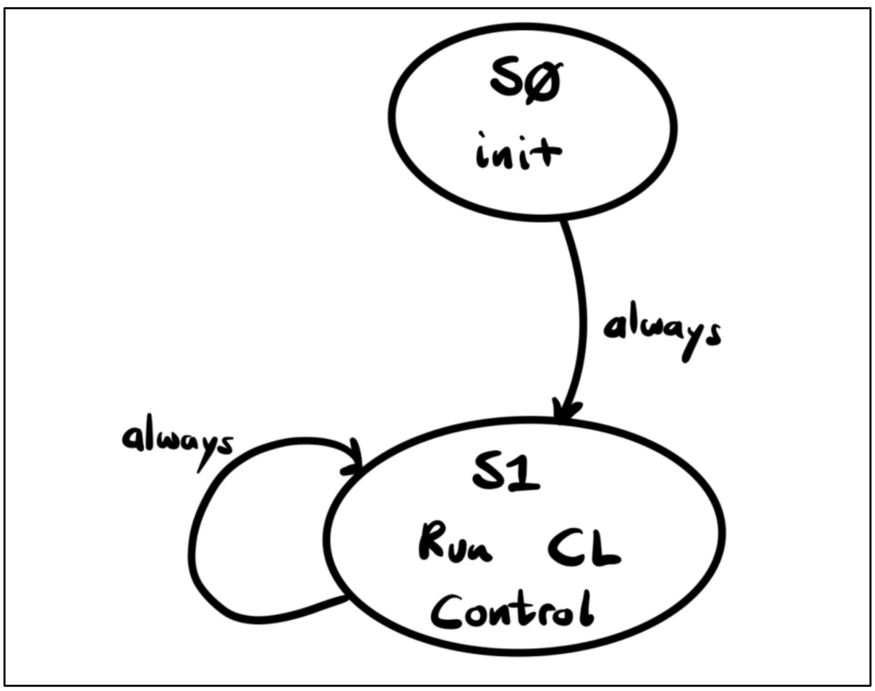


Figure 5: Closed Loop Control FSM

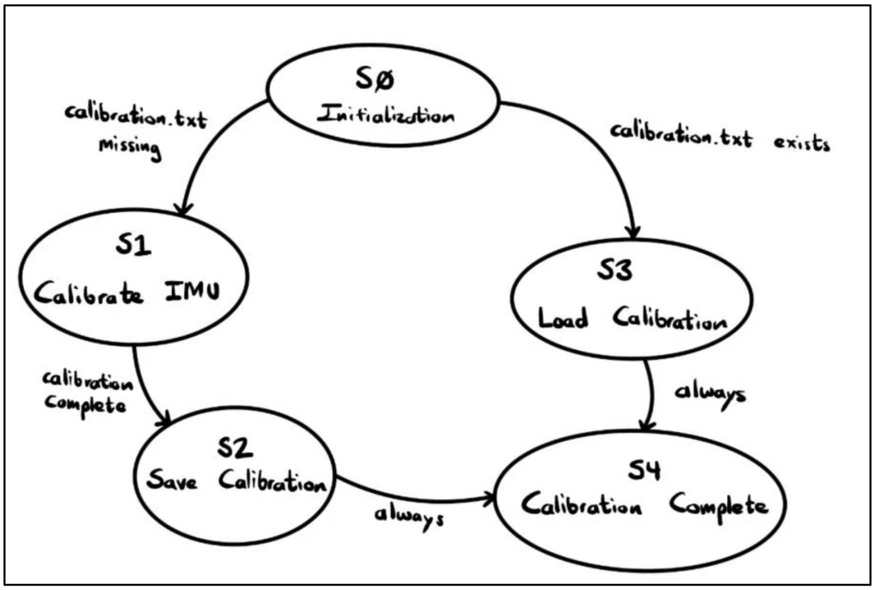


Figure 6: IMU Handler FSM

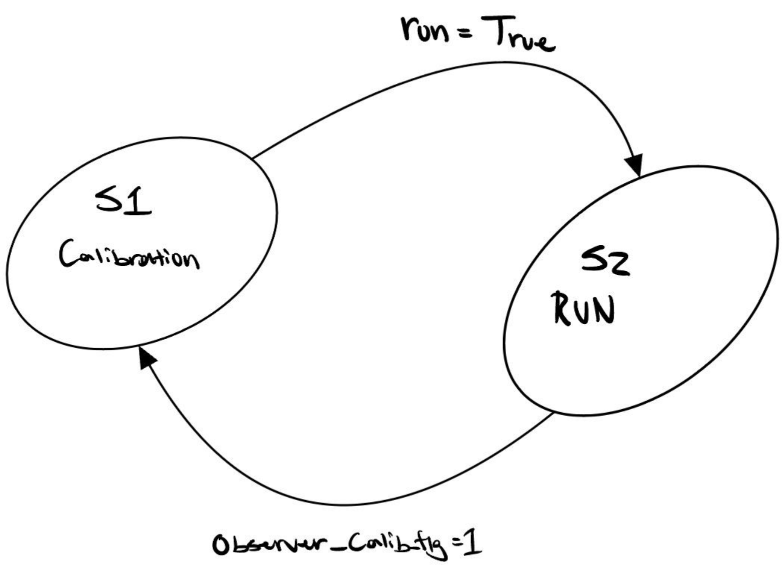


Figure 7: Observer FSM

Alternative Approaches

RK4

State space

References

[1] C. Refvem, *ME 405 Lab Notes and Instructions*, California Polytechnic State University, San Luis Obispo, 2025.

Attachments (Canvas):

1. IMU_handler.py
2. IMU_driver.py
3. observer_fcn.py

**The transcripts for all attachments are included in the pages below

Attachment A: IMU_handler.py

```
'''
Handler for BNO055 IMU with calibration persistence
Implements flowchart: check for calibration.txt -> load or manual calibrate -> run
'''

import gc
import pyb
import cotask
import task_share
from pyb import Pin
from IMU_driver import BNO055
from CalibrationManager import CalibrationManager

def calibration_task_fun(shares):
'''Task to handle calibration startup routine'''
S_INIT = 0
S_CHECK_FILE = 1
S_LOAD_CALIB = 2
S_MANUAL_CALIB = 3
S_SAVE_CALIB = 4
S_DONE = 5
STATE = S_INIT
imu = None
calib_mgr = None
calib_status = None
done_printed = False
while True:
if STATE == S_INIT:
print("=" * 50)
print("=== BNO055 Initialization ===")
print("=" * 50)
rst = Pin('B15', Pin.OUT_PP)
rst.low()
pyb.delay(10)
rst.high()
pyb.delay(700)
```

```
try:
imu = BNO055(scl_pin='B13', sda_pin='B14', addr=0x28)
print("[OK] BNO055 hardware initialized")
calib_mgr = CalibrationManager(imu)
STATE = S_CHECK_FILE
except Exception as e:
print("[FAIL] BNO055 initialization failed: " + str(e))
print("Check: VIN connected? COM3/ADR pin state?")
pyb.delay(1000)
elif STATE == S_CHECK_FILE:
print("\n=== Checking for Calibration File ===")
if calib_mgr.calibration_exists():
print("[OK] calibration.txt found")
STATE = S_LOAD_CALIB
else:
print("[INFO] calibration.txt not found")
print("Manual calibration required (rotate IMU in all directions)")
STATE = S_MANUAL_CALIB
elif STATE == S_LOAD_CALIB:
print("\n=== Loading Calibration from File ===")
if calib_mgr.load_calibration():
pyb.delay(100)
STATE = S_DONE
else:
print("Falling back to manual calibration...")
STATE = S_MANUAL_CALIB
elif STATE == S_MANUAL_CALIB:
print("\n=== Manual Calibration Mode ===")
print("Waiting for full calibration (Sys: 3/3)...")
print("Rotate the IMU slowly in all directions...")
try:
calib_status = imu.get_calibration_status()
sys_cal = calib_status['sys']
if sys_cal == 3:
print("[OK] Fully calibrated! (Sys: " + str(sys_cal) + "/3)")
STATE = S_SAVE_CALIB
#else:
# msg = ("Sys: " + str(calib_status['sys']) + "/3, Gyro: " +
```

CAL POLY

```
# str(calib_status['gyro']) + "/3, Accel: " +
# str(calib_status['accel']) + "/3, Mag: " +
# str(calib_status['mag']) + "/3")
# print(" " + msg)
# pyb.delay(500)
except Exception as e:
print("[FAIL] Calibration read error: " + str(e))
pyb.delay(500)
elif STATE == S_SAVE_CALIB:
print("\n=== Saving Calibration ===")
if calib_mgr.save_calibration():
STATE = S_DONE
else:
print("Warning: Could not save calibration")
STATE = S_DONE
elif STATE == S_DONE:
if not done_printed:
print("\n" + "=" * 50)
print("=== Calibration Complete - Starting Main Loop ===")
print("=" * 50 + "\n")
done_printed = True
STATE = S_DONE
yield STATE

def imu_monitor_task_fun(shares):
'''Task to read and display IMU data after calibration'''
last = None
while True:
imu = shares[0].get()
heading_share = shares[1]
yaw_rate_share = shares[2]

if last != "HEADING":
heading = imu.get_heading()
heading_share.put(heading)
last = "HEADING"
else:
yaw_rate = imu.get_yaw_rate()
```

```
yaw_rate_share.put(yaw_rate)
last = "YAW_RATE"
Yield
```

Attachment B: IMU_driver.py

```
"""
IMU_V1.py – BNO055 IMU driver (MicroPython, software I2C on B13/B14)
IMPROVED: Better timeout handling and recovery
"""

import pyb
import struct
from micropython import const
from pyb import Pin
from time import sleep_us, sleep_ms

# ── I2C Address
────────────────────────────────────────────────────────────
BNO055_ADDRESS_A = const(0x28)
BNO055_ADDRESS_B = const(0x29)

# ── Registers
─────────────────────────────────────────────────────────────
─
REG_CHIP_ID = const(0x00)
REG_PAGE_ID = const(0x07)
REG_UNIT_SEL = const(0x3B)
REG_OPR_MODE = const(0x3D)
REG_PWR_MODE = const(0x3E)
REG_SYS_TRIGGER = const(0x3F)
REG_TEMP_SOURCE = const(0x40)
REG_AXIS_MAP_CONFIG= const(0x41)
REG_AXIS_MAP_SIGN = const(0x42)
REG_CALIB_STAT = const(0x35)

# Euler (heading, roll, pitch) – little endian, 1 LSB = 1/16 degree
REG_EUL_HEADING_LSB= const(0x1A)
REG_EUL_ROLL_LSB = const(0x1C)
REG_EUL_PITCH_LSB = const(0x1E)

# Angular velocity (gyro) – little endian, 1 LSB = 1/16 deg/s
```

```
REG_GYR_DATA_X_LSB = const(0x14)
REG_GYR_DATA_Y_LSB = const(0x16)
REG_GYR_DATA_Z_LSB = const(0x18)

# Calibration profile
REG_CALIB_START = const(0x55)
CALIB_BLOB_LEN = const(22)

# ── Modes ──────────────────────────────────────────────────────────

MODE_CONFIG = const(0x00)
MODE_IMU = const(0x08)
MODE_COMPASS = const(0x09)
MODE_M4G = const(0x0A)
MODE_NDOF_FMC_OFF = const(0x0B)
MODE_NDOF = const(0x0C)

PWR_NORMAL = const(0x00)


class SoftI2C:
"""Software I2C implementation with improved timeout handling"""
def __init__(self, scl_pin, sda_pin, delay_us=50):
"""Initialize with pin names and timing delay"""
self.scl = Pin(scl_pin, Pin.OUT_OD)
self.sda = Pin(sda_pin, Pin.OUT_OD)
self.scl.high()
self.sda.high()
self.delay = delay_us # Increased default for reliability
def _start(self):
"""I2C START condition"""
self.sda.high()
self.scl.high()
sleep_us(self.delay)
self.sda.low()
sleep_us(self.delay)
self.scl.low()
```

```
sleep_us(self.delay)
def _stop(self):
"""I2C STOP condition"""
self.scl.low()
sleep_us(self.delay)
self.sda.low()
sleep_us(self.delay)
self.scl.high()
sleep_us(self.delay)
self.sda.high()
sleep_us(self.delay)
def _repeated_start(self):
"""I2C REPEATED START condition"""
self.scl.low()
sleep_us(self.delay)
self.sda.high()
sleep_us(self.delay)
self.scl.high()
sleep_us(self.delay)
self.sda.low()
sleep_us(self.delay)
self.scl.low()
sleep_us(self.delay)
def _write_bit(self, bit):
"""Write one bit"""
if bit:
self.sda.high()
else:
self.sda.low()
sleep_us(self.delay)
self.scl.high()
sleep_us(self.delay * 2)
self.scl.low()
sleep_us(self.delay)
def _read_bit(self):
"""Read one bit"""
self.sda.high()
sleep_us(self.delay)
```

```
self.scl.high()
sleep_us(self.delay)
bit = self.sda.value()
sleep_us(self.delay)
self.scl.low()
sleep_us(self.delay)
return bit
def _write_byte(self, byte):
"""Write one byte, return ACK bit"""
for i in range(8):
self._write_bit((byte >> (7 - i)) & 1)
ack = not self._read_bit()
return ack
def _read_byte(self, ack):
"""Read one byte, send ACK/NACK"""
byte = 0
for i in range(8):
byte = (byte << 1) | self._read_bit()
self._write_bit(not ack)
return byte
def _bus_reset(self):
"""Reset the I2C bus - clock out up to 9 bits"""
self.sda.high()
for _ in range(9):
self.scl.high()
sleep_us(self.delay)
self.scl.low()
sleep_us(self.delay)
self._stop()
def mem_read(self, nbytes, addr, reg):
"""Read nbytes from register with bus recovery"""
try:
self._start()
if not self._write_byte((addr << 1) | 0):
self._stop()
raise OSError("No ACK on address write")
if not self._write_byte(reg):
self._stop()
```

```
raise OSError("No ACK on register write")
self._repeated_start()
if not self._write_byte((addr << 1) | 1):
self._stop()
raise OSError("No ACK on address read")
data = []
for i in range(nbytes):
data.append(self._read_byte(i < nbytes - 1))
self._stop()
return bytes(data)
except Exception as e:
self._bus_reset()
raise e
def mem_write(self, data, addr, reg):
"""Write data to register with bus recovery"""
try:
self._start()
if not self._write_byte((addr << 1) | 0):
self._stop()
raise OSError("No ACK on address write")
if not self._write_byte(reg):
self._stop()
raise OSError("No ACK on register write")
for byte in data:
if not self._write_byte(byte):
self._stop()
raise OSError("No ACK on data write")
self._stop()
except Exception as e:
self._bus_reset()
raise e


def _le_i16(b):
"""bytes->int16 little-endian"""
return struct.unpack('<h', b)[0]
```

```
class BNO055:
"""Driver for BNO055 using software I2C on arbitrary pins"""

def __init__(self, scl_pin, sda_pin, addr=BNO055_ADDRESS_A,
autodetect=True, retries=10, retry_delay_ms=20):
"""
Initialize with SCL and SDA pin names (strings like 'B13', 'B14')
Args:
scl_pin: SCL pin name (e.g., 'B13')
sda_pin: SDA pin name (e.g., 'B14')
addr: I2C address (0x28 or 0x29)
autodetect: Try both addresses if chip ID doesn't match
retries: Number of retry attempts for I2C operations
retry_delay_ms: Delay between retries (increased for stability)
"""
self.i2c = SoftI2C(scl_pin, sda_pin, delay_us=150) # Increased from 100
self.addr = addr
self._retries = retries
self._retry_delay_ms = retry_delay_ms
self._consecutive_errors = 0

# Confirm device presence
try:
chip = self._mem_read(1, REG_CHIP_ID)[0]
except OSError:
chip = 0
if chip != 0xA0 and autodetect:
for candidate in (BNO055_ADDRESS_A, BNO055_ADDRESS_B):
if candidate != self.addr:
self.addr = candidate
try:
chip = self._mem_read(1, REG_CHIP_ID)[0]
if chip == 0xA0:
break
except OSError:
pass
if chip != 0xA0:
pyb.delay(700)
```

```python
chip = self._mem_read(1, REG_CHIP_ID)[0]
if chip != 0xA0:
raise RuntimeError("BNO055 not detected or wrong chip ID: 0x%02X" % chip)

# Ensure PAGE 0
self._mem_write(bytes([0x00]), REG_PAGE_ID)

# CONFIG mode
self._set_mode_internal(MODE_CONFIG)

# Normal power mode
self._mem_write(bytes([PWR_NORMAL]), REG_PWR_MODE)
pyb.delay(10)

# Exit to NDOF mode
self.set_mode(MODE_NDOF)

# ── Public API
────────────────────────────────────────────────────────────


def set_mode(self, fusion_mode):
"""Change the operating mode"""
self._set_mode_internal(MODE_CONFIG)
self._set_mode_internal(fusion_mode)

def get_calibration_status(self):
"""Return dict with calibration status"""
val = self._mem_read(1, REG_CALIB_STAT)[0]
sys = (val >> 6) & 0x03
gyro = (val >> 4) & 0x03
accel= (val >> 2) & 0x03
mag = (val >> 0) & 0x03
return {"sys": sys, "gyro": gyro, "accel": accel, "mag": mag}

def read_calibration_blob(self):
"""Read calibration coefficients (22 bytes) as binary"""
cur_mode = self._get_mode()
try:
```

```
self._set_mode_internal(MODE_CONFIG)
self._mem_write(bytes([0x00]), REG_PAGE_ID)
blob = self._mem_read(CALIB_BLOB_LEN, REG_CALIB_START)
return bytes(blob)
finally:
self._set_mode_internal(cur_mode)

def write_calibration_blob(self, blob):
"""Write calibration coefficients (exactly 22 bytes)"""
if not isinstance(blob, (bytes, bytearray)) or len(blob) != CALIB_BLOB_LEN:
raise ValueError("Calibration blob must be 22 bytes long")

cur_mode = self._get_mode()
try:
self._set_mode_internal(MODE_CONFIG)
self._mem_write(bytes([0x00]), REG_PAGE_ID)
for i in range(CALIB_BLOB_LEN):
self._mem_write(bytes([blob[i]]), REG_CALIB_START + i)
pyb.delay(2)
finally:
self._set_mode_internal(cur_mode)
pyb.delay(25)

def read_euler(self):
"""Read Euler angles (heading, roll, pitch) in degrees"""
raw = self._mem_read(6, REG_EUL_HEADING_LSB)
h = _le_i16(raw[0:2]) / 16.0
r = _le_i16(raw[2:4]) / 16.0
p = _le_i16(raw[4:6]) / 16.0
return (h, r, p)

def get_heading(self):
"""Convenience: return heading (yaw) in degrees"""
return self.read_euler()[0]

def read_angular_velocity(self):
"""Read angular velocity (gx, gy, gz) in deg/s"""
raw = self._mem_read(6, REG_GYR_DATA_X_LSB)
```

```
gx = _le_i16(raw[0:2]) / 16.0
gy = _le_i16(raw[2:4]) / 16.0
gz = _le_i16(raw[4:6]) / 16.0
return (gx, gy, gz)

def get_yaw_rate(self):
"""Convenience: return gz (yaw rate) in deg/s"""
return self.read_angular_velocity()[2]

# ── Low-Level Helpers
────────────────────────────────────────────────────────────


def _get_mode(self):
return self._mem_read(1, REG_OPR_MODE)[0]

def _set_mode_internal(self, mode):
"""Internal: write OPR_MODE with delays"""
self._mem_write(bytes([mode]), REG_OPR_MODE)
if mode == MODE_CONFIG:
pyb.delay(25)
else:
pyb.delay(10)

def _mem_write(self, data, reg):
"""Write bytes to register with improved retry logic"""
if not isinstance(data, (bytes, bytearray)):
data = bytes([data])
last_err = None
for attempt in range(self._retries):
try:
self.i2c.mem_write(data, self.addr, reg)
self._consecutive_errors = 0 # Reset error counter on success
return
except OSError as e:
last_err = e
self._consecutive_errors += 1
# Longer delay on repeated errors
delay = self._retry_delay_ms * (attempt + 1)
```

```
pyb.delay(min(delay, 100)) # Cap at 100ms
raise last_err

def _mem_read(self, nbytes, reg):
"""Read bytes from register with improved retry logic"""
last_err = None
for attempt in range(self._retries):
try:
data = self.i2c.mem_read(nbytes, self.addr, reg)
self._consecutive_errors = 0 # Reset error counter on success
return data
except OSError as e:
last_err = e
self._consecutive_errors += 1
# Longer delay on repeated errors
delay = self._retry_delay_ms * (attempt + 1)
pyb.delay(min(delay, 100)) # Cap at 100ms
raise last_err
```

Attachment C: Observer_fcn.py

```
import ulab
from ulab import numpy as np
from math import pi, cos, sin
import battery_adc

def observer_fun(t, x, shares):
'''!@brief Implements the observer task function to compute the state
estimates for the system using the outputs from the IMU
'''
(meas_heading_share, meas_yaw_rate_share,
left_enc_pos, right_enc_pos,
left_enc_speed, right_enc_speed,
obs_heading_share, obs_yaw_rate_share,
left_set_point, right_set_point,
observer_calibration_flg, big_X_share, big_Y_share, initial_heading_share) = shares
# Electromechanical properties
K = 250*2*pi/60/4.5 # Motor Gain [rad/(V*s)]
tau = 0.1 # Motor Time Constant [s]
# Romi geometric properties
r = 0.035 # Wheel radius [m]
w = 0.141 # Track width [m]
TICKS_PER_REV = 1437.1 # Encoder ticks per revolution
WHEEL_CIRC_MM = 2 * 3.14159 * 35 # Wheel circumference in mm

batt_v = battery_adc.battery_voltage() # Get battery voltage from adc
# Get the latest heading and yaw rate from the shares
heading = meas_heading_share.get()
yaw_rate = meas_yaw_rate_share.get()
l_voltage = batt_v * left_set_point.get() / 100.0 # Scale set point to voltage
r_voltage = batt_v * right_set_point.get() / 100.0 # Scale set point to voltage
l_pos = left_enc_pos.get() * (WHEEL_CIRC_MM / TICKS_PER_REV) # Convert ticks to mm
r_pos = right_enc_pos.get() * (WHEEL_CIRC_MM / TICKS_PER_REV)

# print(f"Observer inputs: Heading = {heading:.2f} deg, Yaw Rate = {yaw_rate:.2f} deg/s, Left
Pos = {l_pos:.2f} m, Right Pos = {r_pos:.2f} m, Left Voltage = {l_voltage:.2f} V, Right Voltage =
{r_voltage:.2f} V")
```

```
# Observer Implementation
"""
X_hat_dot = (A - LC) * X_hat + (B - LD) * u + L*y

AD = A - LC discretized for 100 Hz
BD = B - LDL discretized for 100 Hz
L = Observer gain matrix
"""

AD = np.array([
[ 3.8702, -2.9670, -9.7279, 0.0000 ],
[ 2.9088, -2.0056, -9.7276, 0.0000 ],
[ -0.0176, 0.0179, 1.0179, 0.0000 ],
[ 0.0000, 0.0000, 0.0000, 0.9704 ]
])

BD = np.array([
[ 1.4277, -0.8744, 4.8640, 4.8640, 0.0000, 12.5549 ],
[ 0.8573, -0.3040, 4.8638, 4.8638, 0.0000, 12.3155 ],
[ -0.0052, 0.0053, -0.0089, -0.0089, 0.0000, -0.0752 ],
[ 0.0000, 0.0000, -0.0021, 0.0021, 0.0293, -0.0099 ]
])

L = np.array([
[ 506.4, 506.4, 0.0, 1242.9 ],
[ 506.4, 506.4, 0.0, 1218.5 ],
[ -1.0, -1.0, 0.0, -7.4 ],
[ -0.2, 0.2, 3.0, -1.0 ]
])

u_star = np.array([
[l_voltage],
[r_voltage],
[l_pos],
[r_pos],
[heading],
[yaw_rate]
```

```
])

# State variables
Omega_L = x[0,0]
Omega_R = x[1,0]
S = x[2,0]
Psi = x[3,0]

#print("Omega_L: {:.2f}, Omega_R: {:.2f}, S: {:.2f}, Psi: {:.2f}".format(Omega_L, Omega_R, S,
Psi))
# State equations
xd = np.array([
[-1/tau * Omega_L + K/tau * u_star[0,0]], # dOmega_L/dt
[-1/tau * Omega_R + K/tau * u_star[1,0]], # dOmega_R/dt
[r/2 * (Omega_R + Omega_L)], # dS/dt
[r/w * (Omega_R - Omega_L)] # dPsi/dt
])
# Output Equations
y = np.array( [
[ S - w/2 * Psi ], # S_L
[ S + w/2 * Psi ], # S_R
[ heading ], # Psi
[ yaw_rate] # Psi_dot
])
x_hat = x
x_hat_dot = np.dot(AD, x_hat) + np.dot(BD, u_star) + np.dot(L, y)
return x_hat_dot, y

def RK4_solver(fcn, x_0, tspan, tstep, shares):
'''!@brief Implements a Runge-Kutta 4th order solver
@param fcn A function handle to the function to solve
@param x_0 The initial value of the state vector
@param tspan A span of time over which to solve the system specified
as a list with two elements representing initial and
final time values
@param tstep The step size to use for the integration algorithm
@param shares Tuple of shares to pass to fcn
@return A tuple containing both an array of time values and an
```

```
array of output values
'''

# Define a column of time values without relying on arange or zeros
num_steps = int(round((tspan[1] - tspan[0]) / tstep)) + 1
tout = np.array([tspan[0] + idx * tstep for idx in range(num_steps)])

# Get dimensions
n_states = x_0.shape[0]
# Determine the dimension of the output vector
_, y_test = fcn(0, x_0, shares)
n_outputs = y_test.shape[0]
# Preallocate arrays
xout = np.zeros((len(tout)+1, n_states))
yout = np.zeros((len(tout), n_outputs))

# Initialize output array with initial state vector (flatten to 1D row)
for i in range(n_states):
xout[0, i] = x_0[i, 0]

# Initialize K1, K2, K3, K4
k1, k2, k3, k4 = [], [], [], []

# Iterate through the algorithm but stop one cycle early because
# the algorithm predicts one cycle into the future
for n in range(len(tout)):
# Get current state as a column vector
x = np.array([[xout[n, i]] for i in range(n_states)])
# Get current time
t = tout[n]
# Calculate k1
k1, y = fcn(t, x, shares)
# Calculate k2
k2, _ = fcn(t + 0.5*tstep, x + 0.5*k1*tstep, shares)
# Calculate k3
k3, _ = fcn(t + 0.5*tstep, x + 0.5*k2*tstep, shares)
# Calculate k4
k4, _ = fcn(t + tstep, x + k3*tstep, shares)
```

```
# Weighted average of derivatives - convert back to row for storage
dx = (k1 + 2*k2 + 2*k3 + k4) * tstep / 6.0
for i in range(n_states):
xout[n+1, i] = xout[n, i] + dx[i, 0]
# Store the output (using y from k1 evaluation)
for i in range(n_outputs):
yout[n, i] = y[i, 0]
return tout, yout


def observer_task_fcn(shares):
'''!@brief Runs the observer calculations
'''
(meas_heading_share, meas_yaw_rate_share,
left_enc_pos, right_enc_pos,
left_enc_speed, right_enc_speed,
obs_heading_share, obs_yaw_rate_share,
left_set_point, right_set_point,
observer_calibration_flg,
big_X_share, big_Y_share, initial_heading_share) = shares
# Electromechanical properties
K = 250*2*pi/60/4.5 # Motor Gain [rad/(V*s)]
tau = 0.1 # Motor Time Constant [s]
# Romi geometric properties
r = 0.035 # Wheel radius [m]
w = 0.141 # Track width [m]

FIRST_RUN = True
run = False

while True:
if observer_calibration_flg.get() == 1:
# Create initial state by setting X, Y and heading to zero, and setting the initial heading to the current measured heading
initial_heading = meas_heading_share.get()
initial_heading_share.put(initial_heading)
print(f"Initial heading set to {initial_heading:.2f} degrees")
initial_X = 100
```

```
initial_Y = 800
big_X_share.put(initial_X)
big_Y_share.put(initial_Y)
observer_calibration_flg.put(0) # Set flag to indicate calibration is done
run = True

if run:
if FIRST_RUN:
x_0 = np.array([
[0.0], # Omega_L
[0.0], # Omega_R
[0.0], # S
[0.1/w * (right_enc_pos.get() - left_enc_pos.get())] # Psi, set based on initial encoder
positions
])
x = x_0
FIRST_RUN = False
else:
x = np.array([
[left_enc_speed.get()], # Omega_L
[right_enc_speed.get()], # Omega_R
[r/2 * (left_enc_pos.get() + right_enc_pos.get())], # S (not used in observer)
[meas_heading_share.get()] # Psi (not used in observer)
])
_, y_Obs = RK4_solver(observer_fun, x, [0, 0.100], 1e-1, shares)

# Modify heading estimate to be relative to initial heading
# also return compounded heading so heading doesnt drop from 0 to 360 after a full rotation
obs_heading = y_Obs[-1,2] - initial_heading_share.get()
if obs_heading < -180:
obs_heading += 360
elif obs_heading > 180:
obs_heading -= 360



# Push outputs into shares
obs_heading_share.put(obs_heading) # Psi
obs_yaw_rate_share.put(y_Obs[-1,3]) # Psi_dot
```

```
big_X = 100 * r/2 * (y_Obs[-1,0] + y_Obs[-1,1]) * cos(obs_heading_share.get()*pi/180) +
initial_X # S_L and S_R
big_Y = -100 * r/2 * (y_Obs[-1,0] + y_Obs[-1,1]) * sin(obs_heading_share.get()*pi/180) +
initial_Y # S_L and S_R

big_X_share.put(big_X)
big_Y_share.put(big_Y)
yield
```