1. Temperature Sensor

1.1. Sensor Purpose

The DS18B20 temperature sensor is integrated into the ROV to monitor environmental conditions critical to the system's operation. Its primary purpose is to ensure thermal conditions for marine creatures, especially underwater coral reefs. Temperature data is also used to issue alerts if thresholds are exceeded and contribute to long-term performance logging.

1.2. Software Requirements

To interface with the DS18B20 temperature sensor, the following software tools and configurations are required:

- Libraries:
  - OneWire: Manages the 1-Wire communication protocol for data transfer.
  - Dallas Temperature: Provides high-level functions for working with the DS18B20.
- Microcontroller Resources:
  - One GPIO pin for communication.
  - Digital pin configured for bidirectional communication on the 1-Wire bus.

1.3. Overview

The DS18B20 is a 1-Wire digital temperature sensor designed to provide precise temperature readings with minimal wiring. It features four primary data components:

- 64-bit Lasered ROM:
  - Uniquely identifies the sensor on a shared 1-Wire bus, enabling multiple sensors to operate simultaneously on a single communication line. This ROM allows addressing specific sensors for commands.
- Temperature Sensor:
  - Captures temperature measurements and converts them to digital format with a configurable resolution (9 to 12 bits).
- Nonvolatile Temperature Alarm Triggers (TH and TL):
  - Stores threshold values for temperature alarms. These registers are nonvolatile and persist even after power loss. They can be repurposed as general-purpose memory.
- Configuration Register:
  - Contains resolution settings (affecting conversion time) and other control parameters.

The DS18B20 supports two power modes:

- Parasite Power Mode:
  - Utilizes energy stored in an internal capacitor charged during high states of the 1-Wire signal.
- External Power Mode:
  - Operates with an external 3V–5.5V supply.

Figure (1) explains the sensor's internal architecture and we will discuss later how to use the sensor simply.

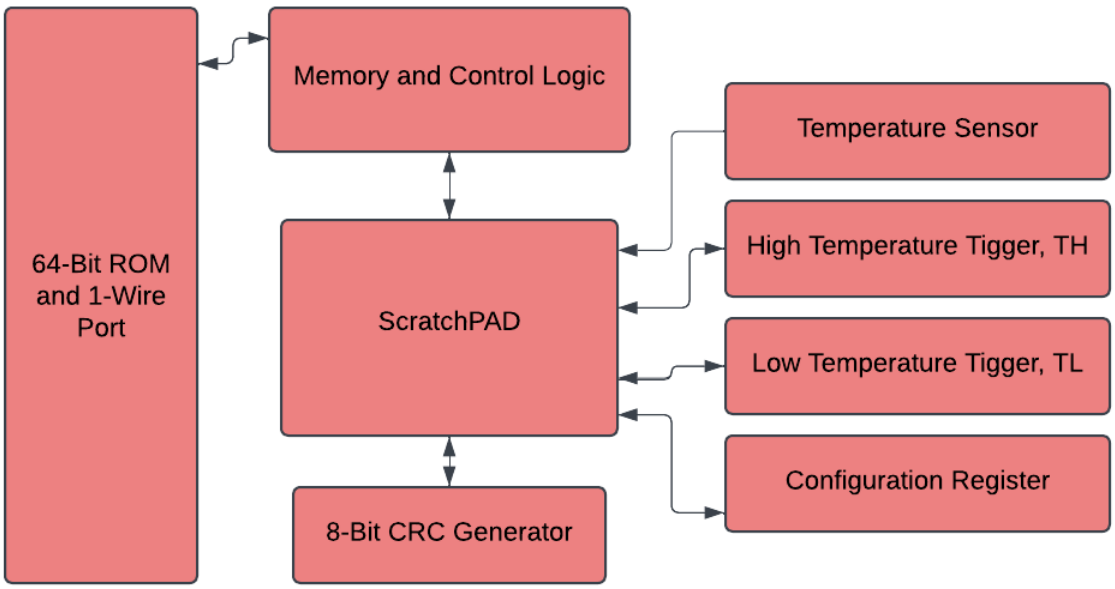

Figure (1)

1.4. Software Architecture

The temperature sensor's role within the software architecture can be divided into the following layers:

- Data Acquisition Layer:
  - Handles the communication with the DS18B20 sensor using the OneWire protocol.
  - Ensures periodic data requests based on system requirements.
- Processing Layer:
  - Converts raw temperature readings to human-readable formats (Celsius and Fahrenheit).
  - Applies filtering or averaging for noise reduction.
- Integration Layer:
  - Sends processed data to subsystems such as the control system (for decision-making) and UI (for display and logging).
- Error Handling Layer:
  - Detects and addresses disconnection or invalid readings.

1.5. Sensor Interface

1.5.1. Initialization and Configuration

The DS18B20 communicates through a 1-Wire protocol, and the DallasTemperature library simplifies this communication by abstracting the low-level operations and not getting involved in the complicated process mentioned before. We first import the OneWire and DallasTemperature libraries so we can use the functions provided by these libraries and then configure the OneWire GPIO pin connected to the sensor (ex, pin 2) and create an object from both libraries.

Then we initialized the sensor and set the resolution to (9 to 12 bits) higher resolution increases accuracy but requires a longer conversion time. (9 bits: ~93 ms and 12 bits: ~750 ms)

1.5.2. Data Retrieval

Acquiring temperature data involves requesting readings and handling delays for conversion we first start by requesting the temperature data from the sensor as it operates asynchronously, requiring a request before retrieving the data and after that, we fetch the temperature data from the sensor in Celsius or Fahrenheit thanks to DallasTemperature library the provide the functions and attributes to do so

1.5.3. Data Processing and Units Conversion

Raw data from the DS18B20 is converted into human-readable units directly by the DallasTemperature library. Temperature is available in both Celsius and Fahrenheit formats.

Scaling and Calibration are available due to systematic error by adding or subtracting offsets.

1.5.4. Integration

We can use the temperature data in logs for performance analysis and maintenance and send warnings or notifications via the UI or telemetry system in case of overheating

1.5.5. Error Handling

The sensor may be disconnected so we check it and print if it's connected or make It restart if it's disconnected. Another error may happen if the temperature data exceeds plausible temperature ranges, flag it for review.

1.6. Pseudocode

1.6.1. Header file

```
DECLARE class TemperatureSensor
    METHOD: begin()
    METHOD: setResolution(resolution)
    METHOD: readTemperature()
ENDCLASS
```

1.6.2. Source file

```
DEFINE TemperatureSensor
    METHOD: begin()
        START OneWire and DallasTemperature objects

    METHOD: setResolution(resolution)
        SET desired resolution for the sensor

    METHOD: readTemperature()
        REQUEST temperature measurement
        RETURN temperature value
ENDDEFINE
```

1.6.3. Main file

```
IMPORT TemperatureSensor library

DECLARE TemperatureSensor object sensor
```

```
SETUP:
    CALL sensor.begin()
    CALL sensor.setResolution(12 bits)

LOOP:
    CALL sensor.readTemperature()
    PRINT temperature value
ENDLOOP
```

2. OneWire Communication Protocol

2.1. Overview

The OneWire library in Arduino is a communication protocol developed by Dallas Semiconductor that allows multiple devices to communicate over a single data wire (bus). This library is commonly used to interface with OneWire devices like temperature sensors (e.g., DS18B20), EEPROMs, or ID chips. These devices all share a single data pin for communication, allowing multiple devices to connect to the same pin on an Arduino.

OneWire uses a master-slave protocol where the Arduino acts as the master and each connected sensor/device is a slave. Each device has a unique 64-bit address, allowing the master to identify and communicate with each sensor individually, even though they all share the same physical bus (data line).

2.2. Architecture

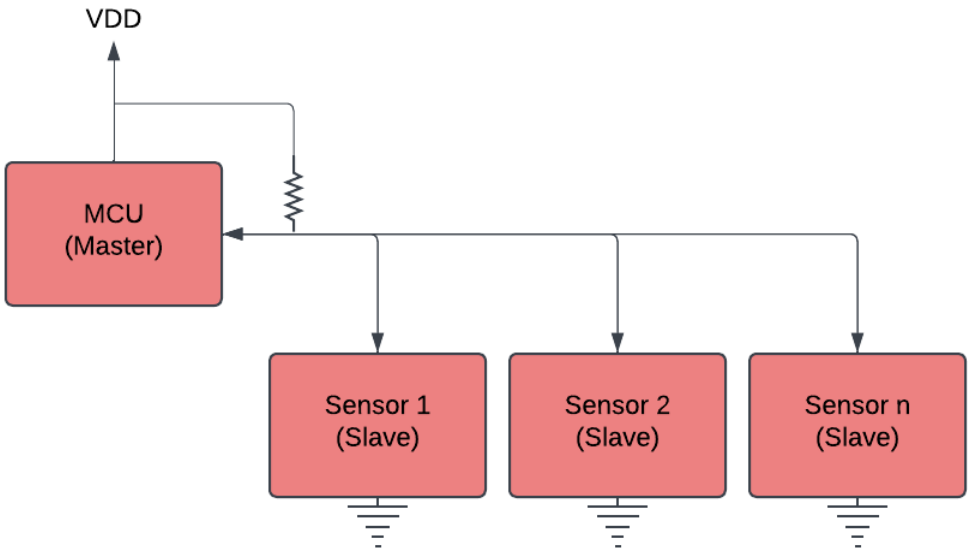

Figure (2)

Figure (2) depicts the OneWire connections, which utilize a single data line for both communication and power. In parasite mode, each device is assigned a unique 64-bit ROM code, facilitating communication among multiple devices. These devices can be powered through the data line (parasite power mode) or via an external power source.

Communication between the bus master and devices occurs using pull-up resistors and time-sensitive signaling. Like a UART/USART controller, the system natively manages clocked operations through a buffer, effectively relieving the processing burden from the host processor, such as a sensor gateway or microcontroller, thereby enhancing accuracy. Often, external pull-up resistors are unnecessary.

The configuration allows for the connection of up to 255 devices on the same wire, supporting half-duplex bidirectional communication.

2.3. Library Installation

- Open the Arduino IDE.
- Go to Sketch > Include Library > Manage Libraries.
- Search for "OneWire" in the library manager.
- Install OneWire by Jim Studt or similar versions.

2.4. Key Functionalities

- Instantiate the OneWire object with the pin connected to the OneWire bus

```
IMPORT OneWire library
DECLARE object oneWire
INITIALIZE oneWire with pinNumber
```

- Detect devices connected to the bus.

```
DECLARE boolean isPresent
DECLARE array address[8]

CALL oneWire.search(address)
    IF a device is found THEN
        SET isPresent = true
        STORE device's ROM code in address array
    ELSE
        SET isPresent = false
```

- Resets the OneWire bus, preparing it for communication.

```
CALL oneWire.reset()
```

- Sends data to a device.

```
DECLARE variable data
CALL oneWire.write(data)
```

- Reads data from a device.

```
DECLARE variable receivedData
SET receivedData = CALL oneWire.read()
```

2.4.1. ROM Commands

- Commands for selecting specific devices on the bus:
    - `0x33` - Read ROM: Retrieves the unique ROM code of the device.

- `0x55` - Match ROM: Communicates with a specific device using its ROM code.
- `0xCC` - Skip ROM: Communicates with all devices on the bus simultaneously.

2.5. Conclusion

The OneWire library simplifies communication with OneWire devices by handling protocol intricacies. Combined with libraries like DallasTemperature, it allows for robust implementation in projects requiring sensor networks or single-wire data communication.

3. Pressure and Depth Sensor

3.1. Overview

The MS5540C is a digital pressure sensor module used for barometric and underwater pressure measurements. It provides pressure and temperature data through a fully calibrated 16-bit ADC converted into depth readings. This sensor is crucial for underwater applications like ROVs. Communication is achieved via a simple SPI-like interface, making it suitable for embedded systems.

3.2. Functional Description

3.2.1. Internal Component

The MS5540C consists of a piezo-resistive sensor and a sensor interface IC. The main function of the MS5540C is to convert the uncompensated analogue output voltage from the piezo-resistive pressure sensor to a 16-bit digital value, as well as providing a 16-bit digital value for the temperature of the sensor. The interface IC consists of PROM that stores the sensor's calibration coefficients used for temperature and pressure compensation. Also have a control logic to manage data acquisition and communication with the microcontroller. Figure (3) give a clear visualization for internal components of the IC.

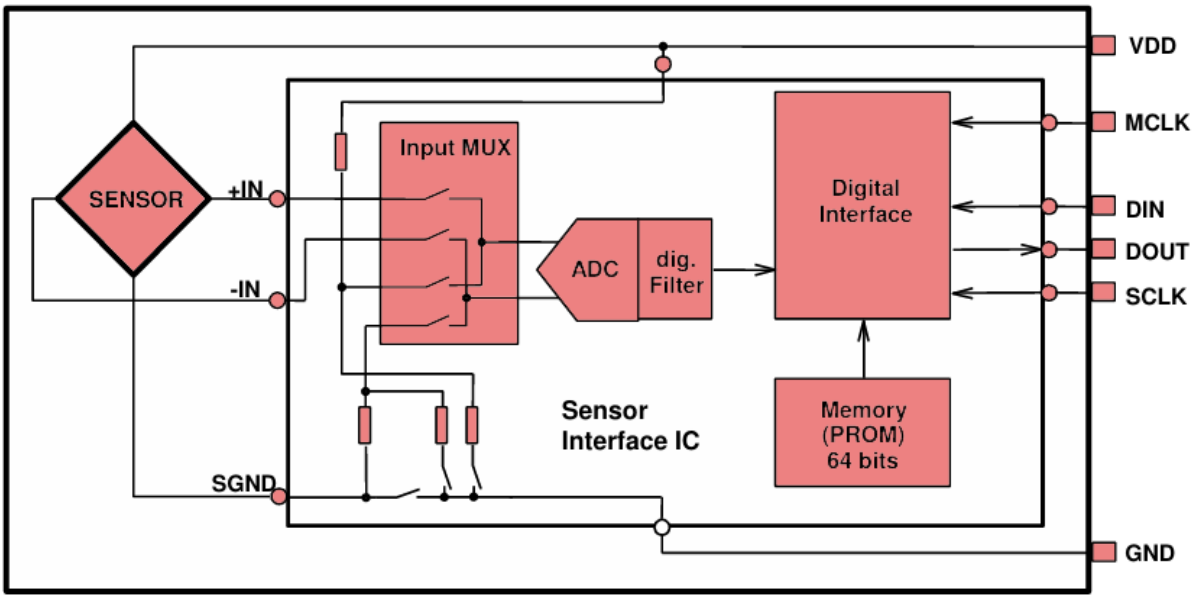

Figure (3)

3.2.2. Operating Principle

The sensor detects pressure using its piezoresistive element as well as temperature. Since the output voltage of a pressure sensor is heavily influenced by temperature and process tolerances, it is essential to compensate for these

effects during pressure measurement. The differential output voltage from the pressure sensor is converted accordingly. For temperature measurement, the resistance of the sensor bridge is monitored and converted using an internal thermistor. Analog signals are digitized by the ADC (sigma-delta converter).

Each module is individually factory-calibrated at two different temperatures and two pressure levels, leading to the calculation and storage of six coefficients necessary for compensating temperature and process variations in the 64-bit PROM of each module as in (Figure 4). This 64-bit data, divided into four 16-bit words, must be accessed by the microcontroller's software and utilized in the program to convert the variables D1 and D2 into compensated pressure and temperature values. The raw data is corrected for both pressure and temperature readings, and data is transferred to the microcontroller through a 3-wire SPI interface.

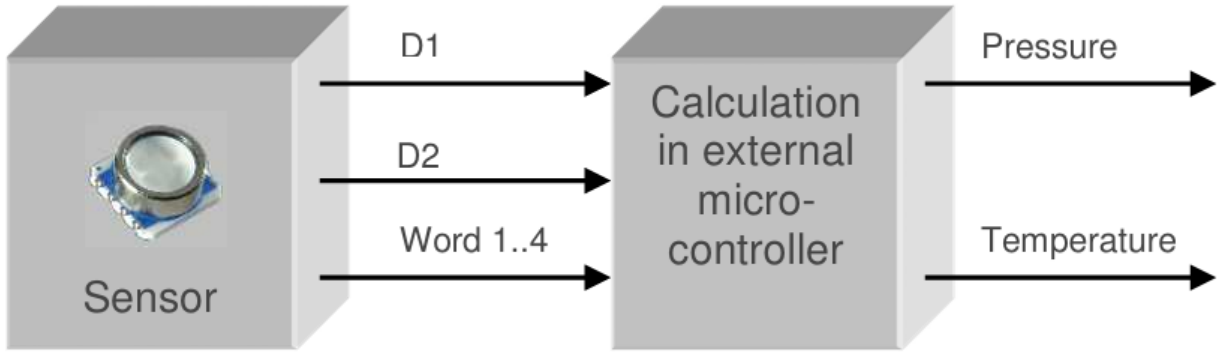

Figure (4)

This behavior is clearly evident in the performance curves below for the raw pressure, the actual pressure (Figure 5), and the Absolute Pressure Accuracy after calibration with 2nd order compensation (Figure 6).

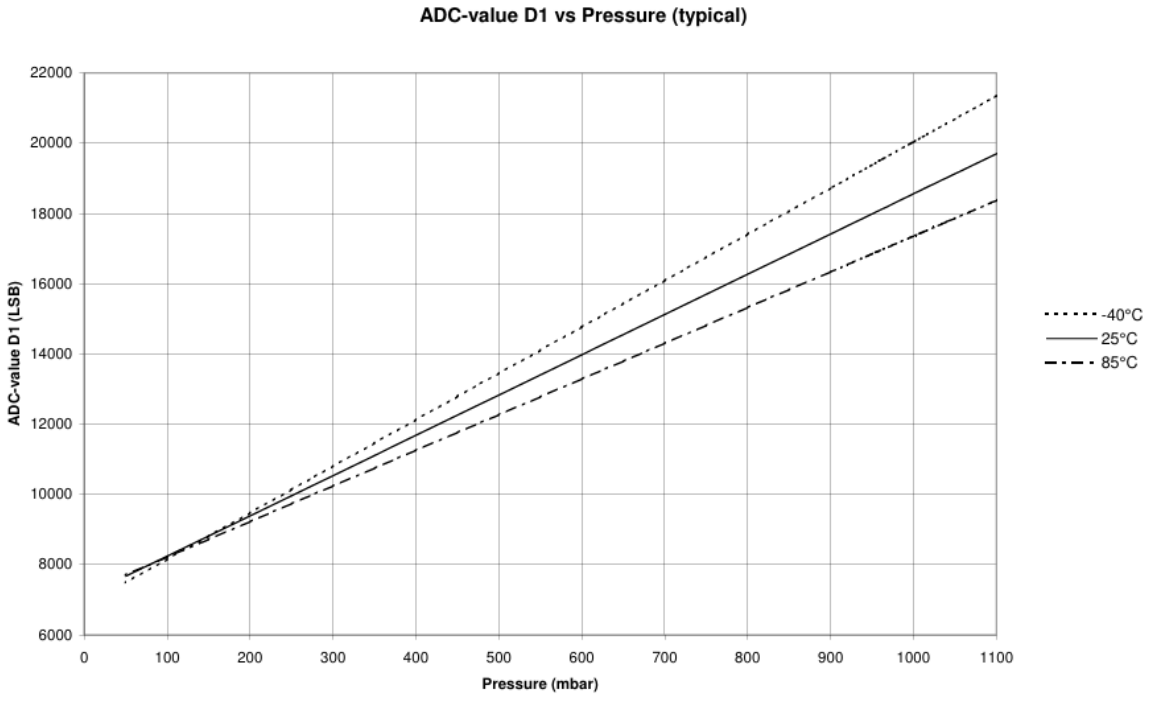

Figure 5

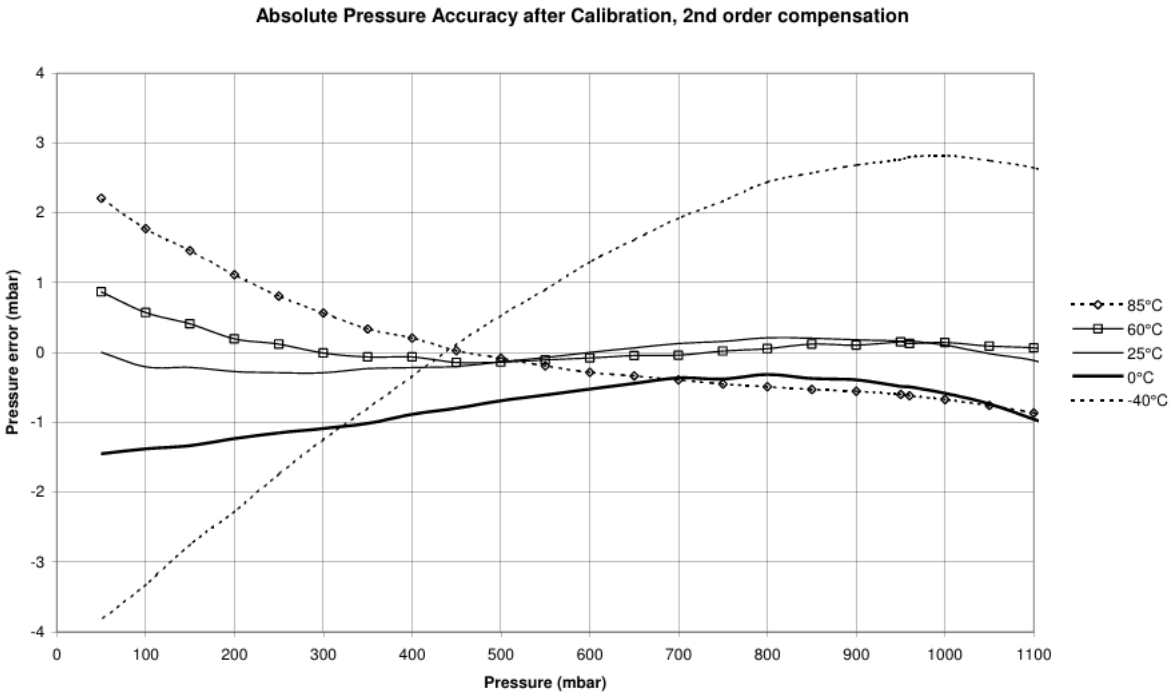

Figure 6

3.2.3. Compensation Algorithm

The raw pressure and temperature readings are processed using polynomial equations based on the PROM coefficients:

$$Temperature\ (°C) = A + B \times D2 + C \times D2^2$$
$$Pressure\ (mbar) = P1 + P2 \times D1 + P3 \times D1^2$$

- *D1*: Raw pressure data.
- *D2*: Raw temperature data.
- *A, B, C, P1, P2, P3*: PROM coefficients.

3.3. Software Interface

I. To begin, we initialize the SPI communication protocol by configuring the settings in the code. We set the bit order to MSB (Most Significant Bit) first and the clock divider to 32.
II. We also created a function to reset the SPI communication. This is crucial to avoid any data corruption that may arise from sensor readings. It is essential to use this function before retrieving any data, such as before each calibration word retrieval, as well as for the raw pressure and raw temperature readings.
III. Send the reset command to initialize communication and set the default states.
IV. Read the PROM (Programmable Read-Only Memory) to fetch the calibration coefficients.
V. Send the Convert D1 command to initiate the pressure data acquisition process.
VI. Wait for the specified conversion time, which is approximately 35 ms for high resolution.
VII. Send the ADC Read command to retrieve the raw pressure data (D1).
VIII. Repeat the above steps for obtaining temperature data (D2).
IX. Use the calibration coefficients read from PROM.
X. Apply compensation equations to calculate corrected temperature and pressure values.
XI. Convert pressure data to depth:

$$Depth\ (meters) = \frac{Pressure\ (mbar) - Atmospheric\ Pressure\ (mbar)}{Fluid\ Density \times g}$$

3.4. Pseudocode

- Header File

```
DECLARE FUNCTION resetsensor()
    DESCRIPTION: Resets the communication state of the sensor.

DECLARE FUNCTION SPISetup(clockPin)
    INPUT: clockPin (Integer)
    DESCRIPTION: Initializes SPI communication with the given clock pin.

DECLARE FUNCTION clockSetup(clockPin)
    INPUT: clockPin (Integer)
    DESCRIPTION: Starts communication by generating a clock signal.

DECLARE FUNCTION getCallibrationWords(words)
    INPUT: words (Array of 4 unsigned integers)
    DESCRIPTION: Retrieves calibration words from the sensor.

DECLARE FUNCTION getCoefficients(coefficients)
    INPUT: coefficients (Array of 6 long integers)
    DESCRIPTION: Calculates coefficients from calibration words.

DECLARE FUNCTION calculateRawPressure(D1)
    INPUT: D1 (Array of 1 unsigned integer)
    DESCRIPTION: Reads raw pressure data from the sensor.

DECLARE FUNCTION calculateRawTemperature(D2)
    INPUT: D2 (Array of 1 unsigned integer)
    DESCRIPTION: Reads raw temperature data from the sensor.

DECLARE FUNCTION getCompensatedPressureTemp(values)
    INPUT: values (Array of 2 floating-point numbers)
    DESCRIPTION: Computes temperature and pressure from raw sensor data.

DECLARE FUNCTION getTemperatureinC() RETURNS Float
    DESCRIPTION: Gets temperature in Celsius.

DECLARE FUNCTION getPressureinMBAR() RETURNS Float
    DESCRIPTION: Gets pressure in millibars.

DECLARE FUNCTION getPressureinMMHG() RETURNS Float
    DESCRIPTION: Gets pressure in millimeters of mercury.

DECLARE FUNCTION getDepthinMeter(pressureInMBar) RETURNS Float
    INPUT: pressureInMBar (Float)
    DESCRIPTION: Calculates depth from pressure in meters.
```

- Source Code

```
BEGIN PROGRAM

DEFINE valuesArray AS float array of size 3 initialized to 0
```

```
FUNCTION resetsensor()
    SET SPI data mode to SPI_MODE0
    SEND SPI commands 0x15, 0x55, 0x40
END FUNCTION

FUNCTION SPISetup(clockPin)
    INITIALIZE SPI
    SET SPI bit order to MSBFIRST
    SET SPI clock divider to SPI_CLOCK_DIV32
    CONFIGURE clockPin as OUTPUT
    WAIT for 100 milliseconds
END FUNCTION

FUNCTION clockSetup(clockPin)
    CONFIGURE Timer1 settings to generate MCKL signal
    OUTPUT analog signal 128 on clockPin
END FUNCTION

FUNCTION getCallibrationWords(words)
    DECLARE inByteWords AS unsigned int array of size 4 initialized to 0
    CALL resetsensor()

    FOR each calibration word from 1 to 4 DO
        SEND specific SPI command to request calibration word
        SET SPI data mode to SPI_MODE1
        READ first byte of word and shift it
        READ second byte of word
        COMBINE first and second byte into words array
        PRINT calibration word
        CALL resetsensor()
    END FOR
END FUNCTION

FUNCTION getCoefficients(coefficients)
    DECLARE wordArray AS unsigned int array of size 4
    CALL getCallibrationWords(wordArray)

    EXTRACT coefficients using bit manipulation on wordArray:
        c1 = (wordArray[0] >> 1) & 0x7FFF
        c2 = ((wordArray[2] & 0x003F) << 6) | (wordArray[3] & 0x003F)
        c3 = (wordArray[3] >> 6) & 0x03FF
        c4 = (wordArray[2] >> 6) & 0x03FF
        c5 = ((wordArray[0] & 0x0001) << 10) | ((wordArray[1] >> 6) & 0x03FF)
        c6 = wordArray[1] & 0x003F

    PRINT all coefficients
END FUNCTION

FUNCTION calculateRawPressure(D1)
    CALL resetsensor()
    SEND SPI command to start pressure conversion
    WAIT for conversion to complete
```

```
    SET SPI data mode to SPI_MODE1
    READ and combine two bytes of pressure data into D1
    PRINT raw pressure value
END FUNCTION

FUNCTION calculateRawTemperature(D2)
    CALL resetsensor()
    SEND SPI command to start temperature conversion
    WAIT for conversion to complete
    SET SPI data mode to SPI_MODE1
    READ and combine two bytes of temperature data into D2
    PRINT raw temperature value
END FUNCTION

FUNCTION getCompensatedPressureTemp(values)
    DECLARE coefficientsArray AS long array of size 6
    DECLARE rawPressure, rawTemperature AS unsigned int arrays of size 1

    CALL getCoefficients(coefficientsArray)
    CALL calculateRawPressure(rawPressure)
    CALL calculateRawTemperature(rawTemperature)

    COMPUTE compensation values:
        UT1 = (coefficientsArray[4] << 3) + 20224
        dT = rawTemperature[0] - UT1
        TEMP = 200 + ((dT * (coefficientsArray[5] + 50)) >> 10)
        OFF = (coefficientsArray[1] * 4) + (((coefficientsArray[3] - 512) *
dT) >> 12)
        SENS = coefficientsArray[0] + ((coefficientsArray[2] * dT) >> 10) +
24576
        X = (SENS * (rawPressure[0] - 7168) >> 14) - OFF
        PCOMP = ((X * 10) >> 5) + 2500

    ASSIGN TEMP and PCOMP to values array
END FUNCTION

FUNCTION getTemperatureinC()
    CALL getCompensatedPressureTemp(valuesArray)
    RETURN valuesArray[0] / 10
END FUNCTION

FUNCTION getPressureinMBAR()
    CALL getCompensatedPressureTemp(valuesArray)
    RETURN valuesArray[1]
END FUNCTION

FUNCTION getPressureinMMHG()
    CALL getCompensatedPressureTemp(valuesArray)
    RETURN valuesArray[1] * 750.06 / 10000
END FUNCTION

FUNCTION getDepthinMeter(pressureInMBar)
    DECLARE rho AS 1025.0
```

```
    DECLARE g AS 9.81
    RETURN pressureInMBar / (rho * g)
END FUNCTION

END PROGRAM
```

- Main File

```
INCLUDE LIBRARY: PressureAndDepthSensor.h

DECLARE CONSTANT clock = 9
    DESCRIPTION: Pin for generating MCKL signal.

DEFINE FUNCTION setup()
    BEGIN
        START serial communication at 9600 baud rate.
        CALL SPISetup(clock) to initialize SPI communication with the clock
pin.
    END

DEFINE FUNCTION loop()
    BEGIN
        CALL clockSetup(clock) to generate MCKL signal.

        PRINT "Pressure in mbar = " to Serial Monitor.
        PRINT RETURN VALUE of getPressureinMBAR() to Serial Monitor.

        PRINT "Pressure in mmHg = " to Serial Monitor.
        PRINT RETURN VALUE of getPressureinMMHG() to Serial Monitor.

        PRINT "Depth in m = " to Serial Monitor.
        CALL getPressureinMBAR() and PASS its RETURN VALUE to
getDepthinMeter().
        PRINT RESULT to Serial Monitor.

        PRINT "Temperature in C = " to Serial Monitor.
        PRINT RETURN VALUE of getTemperatureinC() to Serial Monitor.

        WAIT for 5000 milliseconds.
    END
```

3.5. Problems and Solutions

The sensor can withstand up to 100 meters under water, however the linear range for factory calibrated coefficients stored on the sensor ROM adjusted for linear range of about 7m, so if we want a more linear range we may calibrate our own factors and use each group of factors for each range.

3.5.1. Steps to Calibrate the MS5540C Sensor

A calibrated reference pressure source that can generate pressure within our desired range of 0 to 100 meters is required. Additionally, a temperature-controlled environment is necessary to perform calibrations across the

expected operating temperatures. We need a data acquisition system to read and log the sensor's ADC values (D1, D2), along with the reference pressure and temperature.

Software or scripts will be needed to compute the best-fit calibration coefficients. For each pressure step within the desired range (in increments of 5 meters up to 100 meters), we will record the following:

- Raw pressure ADC (D1) and temperature ADC (D2) from the sensor.
- Reference pressure and temperature using a high-accuracy device.

For each pressure step, we will vary the temperature to capture data at multiple points. For example, we will record data at 0°C, 25°C, and 50°C. Each data point must include D1, D2, the reference pressure, and the reference temperature.

The collected data will be used to compute the calibration coefficients (C1–C6) by following these steps:

- Fit the Data: Use curve-fitting or regression analysis to match the recorded D1 and D2 values against the reference pressure and temperature. Software tools such as MATLAB, Python (NumPy, SciPy), or Excel can assist with this process.
- Calculate Coefficients: The coefficients C1 to C6 will represent offsets, scaling factors, and temperature dependencies. We need to fit the following equations:
    - Pressure Offset: This accounts for any zero-pressure offset.
    - Pressure Sensitivity: This relates the raw data to the reference pressure.
    - Temperature Compensation Terms: These correct for the effects of temperature on the pressure readings.

3.6. Conclusion

The MS5540C is an advanced digital pressure sensor that delivers accurate pressure and temperature measurements. With its user-friendly communication protocol and factory-calibrated coefficients, it is well-suited for embedded systems. Following the recommended software implementation procedures ensures reliable data acquisition, enhancing performance in real-world applications.

4. MPU6050

4.1. Introduction

The MPU6050 is a 6-axis motion tracking device that integrates a 3-axis gyroscope and a 3-axis accelerometer on a single chip. It is commonly used in applications like robotics, drones, game controllers, and ROVs due to its compact size, low power consumption, and accuracy.

4.2. Key Features

- Three-Axis Gyroscope: Measures rotational velocity around three axes (X, Y, and Z). It provides readings in degrees per second (°/s), allowing the measurement of how quickly an object is turning.
- Three-Axis Accelerometer: Measures acceleration forces acting on the sensor along the three axes. These readings help detect the sensor’s orientation relative to the ground and measure acceleration.
- Digital Motion Processor (DMP): The DMP can handle complex motion processing algorithms internally, offloading processing from the main microcontroller. This feature is optional and not used in simpler setups.
- I2C Communication: The MPU6050 uses I2C protocol, making it simple to communicate with microcontrollers like the Arduino. Its default I2C address is 0x68.

The following Figure (7) illustrates the internal components of the MPU60X0

The MPU-60X0 series integrates a 3-axis MEMS gyroscope, a 3-axis MEMS accelerometer, and an embedded Digital Motion Processor (DMP) for efficient motion tracking. The gyroscope detects rotation using the Coriolis Effect, producing a voltage signal proportional to angular rate, which is converted by on-chip 16-bit ADCs. The accelerometer measures linear acceleration with calibrated proof masses, offering precise readings independent of supply voltage. Both sensors feature programmable ranges for versatile applications.

The DMP processes motion algorithms independently, reducing load on the host processor and enabling high-frequency operations for accurate motion tracking. Communication is facilitated via I2C or SPI (MPU-6000 only), while an auxiliary I2C interface supports external sensors, such as magnetometers, in master or pass-through modes for seamless data integration. These features make the MPU-60X0 ideal for applications requiring robust motion sensing and efficient processing.

4.3. Operation

To understand the operation of the MPU-6050, let us break it down step-by-step, beginning with raw data acquisition from the gyroscope and accelerometer, their conversion into usable units (degrees and g), and finally the application of a complementary filter for accurate orientation estimation.

4.3.1. Raw Data Acquisition

The MPU-6050 measures angular velocity using its gyroscope and linear acceleration using its accelerometer. The raw data is accessed via I2C or SPI communication. The device registers hold the 16-bit signed integers for each axis of the gyroscope and accelerometer.

- Gyroscope Data:
  The gyroscope provides raw angular velocity in counts per second for each axis (X, Y, Z). These values represent the rate of rotation about each axis.
- Accelerometer Data:
  The accelerometer provides raw acceleration in counts for each axis (X, Y, Z). These values represent linear acceleration, including the influence of gravity.

Register Mapping:

- Gyroscope raw data: Registers `GYRO_XOUT_H`, `GYRO_XOUT_L`, etc.
- Accelerometer raw data: Registers `ACCEL_XOUT_H`, `ACCEL_XOUT_L`, etc.

Raw data values are signed integers ranging from -32768 to 32767.

4.3.2. Conversion of Raw Data

The raw values must be converted into meaningful units like degrees per second (for gyroscope) and g (for accelerometer).

Gyroscope Conversion:

The gyroscope's raw data is converted into degrees per second using the selected full-scale range (±250, ±500, ±1000, or ±2000 dps). The conversion formula is:

$$Angular\ Velocity\ (dps) = \frac{Raw\ Gyro\ Value}{Gyro\ Sensitivity}$$

- Gyro Sensitivity values depend on the full-scale range:
  - ±250 dps → 131 LSB/dps
  - ±500 dps → 65.5 LSB/dps
  - ±1000 dps → 32.8 LSB/dps
  - ±2000 dps → 16.4 LSB/dps

Accelerometer Conversion:

The accelerometer's raw data is converted into g (gravitational force units) using the selected full-scale range (±2g, ±4g, ±8g, or ±16g). The conversion formula is:

$$Acceleration\ (g) = \frac{Raw\ Accel\ Value}{Accel\ Sensitivity}$$

- Accel Sensitivity values depend on the full-scale range:
  - ±2g → 16384 LSB/g
  - ±4g → 8192 LSB/g
  - ±8g → 4096 LSB/g
  - ±16g → 2048 LSB/g

4.3.3. Combining Gyroscope and Accelerometer Data

The gyroscope and accelerometer are complementary sensors:

- Gyroscope measures angular velocity and calculates the angular position by integrating over time.
- Accelerometer measures the direction of gravity to estimate tilt.

Each sensor has limitations:

- The gyroscope drifts over time.
- The accelerometer is noisy and affected by external accelerations.

To overcome these limitations, a Complementary Filter is applied.

4.3.4. Applying the Complementary Filter

The complementary filter combines the short-term stability of the gyroscope and the long-term stability of the accelerometer to calculate a stable tilt angle.

Calculating Tilt Angle from Accelerometer:

Using trigonometric functions, the tilt angles in the pitch (x-axis) and roll (y-axis) directions are calculated as:

$$Pitch\ (Accel) = arctan\left(\frac{AccelY}{\sqrt{AccelX^2 + AccelZ^2}}\right)$$

$$Roll\ (Accel) = arctan\left(\frac{-AccelX}{\sqrt{AccelY^2 + AccelZ^2}}\right)$$

Calculating Tilt Angle from Gyroscope:

The gyroscope's angular velocity is integrated over time to estimate the angular position:

$$Angle\ (Gyro) = Previous\ Angle + (Angular\ Velocity \times \Delta t)$$

Where Δt is the time elapsed between measurements.

Combining with Complementary Filter:

The complementary filter blends the accelerometer and gyroscope data to get a stable angle estimate:

$$Filtered\ Angle = \alpha(Angle\ (Gyro)) + (1 - \alpha)(Angle\ (Accel))$$

- α is the filter constant, typically between 0.95 and 0.99, to prioritize gyroscope data.

4.4. Software Interface

4.4.1. Dependencies

To interact effectively with the MPU6050 and enhance its functionality, several libraries are used in the software architecture. Each library serves a specific purpose:

- Core MPU6050 Libraries
  - `Adafruit_Sensor`: Provides a unified sensor interface that standardizes the way sensors like the MPU6050 are managed in software.
  - `Adafruit_BusIO`: Facilitates low-level communication via I2C or SPI protocols, ensuring efficient and error-free data transfer between the microcontroller and the sensor.
- Auxiliary Libraries for Data Visualization
  - `Adafruit_GFX_Library`: A graphics library for rendering text, shapes, and bitmaps on displays. It is used to visually represent data from the MPU6050, such as accelerometer readings or angular tilt.
  - `Adafruit_SSD1306`: Specifically designed for OLED displays, this library renders MPU6050 output on small OLED screens, providing real-time monitoring of motion data.

You can download it by opening the Arduino libraries in the sidebar and searching for these libraries to download them.

4.4.2. Interface

- Install Libraries:
  - Open Arduino IDE and install the `MPU6050_tockn` library using the Library Manager.
  - Verify that the `Wire` library is already included with yo.
- Setup Code:
  - Include the `MPU6050_tockn.h` and `Wire.h` headers in your project.
  - Create an instance of the `MPU6050` class, passing the `Wire` object for I2C communication.
- Initialize the Sensor:
  - Start the Serial communication for debugging.
  - Use `Wire.begin()` to start the I2C interface.
  - Initialize the MPU6050 sensor with `mpu6050.begin()`.
- Calibrate Gyroscope Offsets:
  - Use the `calcGyroOffsets(true)` function if needed to measure offsets while the sensor is stationary.
  - Alternatively, set the offsets manually using `setGyroOffsets(x, y, z)`.
- Read and Print Sensor Data:

- Call `mpu6050.update()` in the `loop` function to fetch the latest sensor data.
- Use the library's built-in functions to get filtered angles (pitch, roll, yaw) via `getAngleX()`, `getAngleY()`, and `getAngleZ()`.

4.4.3. Pseudocode

Main File

```
BEGIN InterfaceWithMPU6050

    // Step 1: Install Required Libraries
    // a. Open Arduino IDE.
    // b. Go to Sketch > Include Library > Manage Libraries.
    // c. Search for "MPU6050_tockn" and install it.
    // d. Ensure the "Wire" library is pre-installed (default with Arduino
IDE).

    // Step 2: Include Libraries
    INCLUDE MPU6050_tockn.h
    INCLUDE Wire.h

    // Step 3: Initialize MPU6050 Object
    CREATE mpu6050 OBJECT USING Wire

    // Step 4: Define Global Timer
    INITIALIZE MPU_longTimer TO 0

    // Step 5: Setup Function
    FUNCTION setup()
        BEGIN
            // Initialize Serial Communication
            CALL Serial.begin WITH PARAMETER (9600)

            // Initialize I2C Communication
            CALL Wire.begin

            // Initialize MPU6050
            CALL mpu6050.begin

            // Optional: Measure Gyroscope Offsets
            // a. Place MPU6050 in a stationary, level position.
            // b. Uncomment the following line to calculate offsets:
            //    CALL mpu6050.calcGyroOffsets WITH PARAMETER (true)
            // c. After obtaining offsets, use the setGyroOffsets function
below.

            // Set Manual Gyroscope Offsets (Obtained from Measurements)
            CALL mpu6050.setGyroOffsets WITH PARAMETERS (-5.49, 0.64, -0.75)
        END FUNCTION

    // Step 6: Loop Function
    FUNCTION loop()
```

```
        BEGIN
            // Update MPU6050 Sensor Data
            CALL mpu6050.update

            // Check Timer for Output Interval
            IF (CURRENT_TIME - MPU_longTimer > 1000) THEN
                BEGIN
                    // Fetch Filtered Angles from MPU6050
                    SET angleX TO CALL mpu6050.getAngleX
                    SET angleY TO CALL mpu6050.getAngleY
                    SET angleZ TO CALL mpu6050.getAngleZ

                    // Print Filtered Angles (Pitch, Roll, Yaw)
                    CALL Serial.print WITH PARAMETER ("angleX: ")
                    CALL Serial.print WITH PARAMETER (angleX)
                    CALL Serial.print WITH PARAMETER ("\tangleY: ")
                    CALL Serial.print WITH PARAMETER (angleY)
                    CALL Serial.print WITH PARAMETER ("\tangleZ: ")
                    CALL Serial.println WITH PARAMETER (angleZ)

                    // Update Timer
                    SET MPU_longTimer TO CURRENT_TIME
                END
            END IF
        END FUNCTION

END InterfaceWithMPU6050
```

4.5. Problems and Solutions

Yaw angle, representing rotation around the vertical axis, is crucial in applications like navigation and robotics. Calculating yaw using gyroscope data alone involves integrating angular velocity over time. However, due to small errors in gyroscope readings (bias and noise), the integration accumulates these errors, leading to cumulative drift. Over time, the yaw angle becomes increasingly unreliable, especially for long-duration operations.

- Detailed Analysis

- Gyroscope Contribution:
  - The gyroscope measures angular velocity (ω) in degrees per second (or radians per second).
  - Yaw is derived by integrating this angular velocity over time:

$$Yaw\ (Gyro) = \int \omega z\, dt$$

  - Any bias or noise in ωz gets integrated, causing the yaw value to drift.

- Lack of Absolute Reference:
  - Unlike pitch and roll, which can be stabilized using accelerometer data due to gravity, yaw lacks a natural stabilizing reference. This absence makes it more prone to drift.
- Impacts of Drift:
  - Drift leads to incorrect orientation, causing errors in navigation and control systems.
  - For instance, a robot using yaw for direction will deviate from its intended path.

- Solution

To mitigate cumulative drift, a magnetometer can be combined with the gyroscope data. The magnetometer measures the Earth's magnetic field and provides an absolute reference for the yaw angle.

- Yaw Calculation Using a Magnetometer:
  - The magnetometer outputs the magnetic field components Mx and My in the horizontal plane.
  - The yaw angle is calculated as:

$$Yaw\ (Magnetometer) = \arctan 2(My, Mx)$$

  - This provides an absolute yaw measurement, free from drift.
- Complementary Filter Integration:
  - To combine the short-term accuracy of the gyroscope with the long-term stability of the magnetometer, a complementary filter is used:

$$Filtered\ Yaw = \alpha(Yaw\ (Gyro)) + (1 - \alpha)(Yaw\ (Magnetometer))$$

  - Here:
    - α is a tuning parameter (0.98) that balances responsiveness and stability.
    - The gyroscope contributes rapid changes, while the magnetometer corrects drift over time.

By integrating a magnetometer and applying a complementary filter, the cumulative drift problem in yaw calculations is effectively resolved, enhancing the overall reliability of orientation systems.