# ***Sensor Integration***

* 1. ***Current Sensor***

The ACS712 30A is a Hall-effect-based current sensor capable of measuring AC and DC currents up to 30A. It provides an analog output voltage proportional to the current flowing through the sensor.

* + 1. ***Sensor Purpose***

A current sensor is a device that detects electric current in a wire and generates a signal proportional to that current. The generated signal can be:

* Analog voltage or current for direct measurement.
* Digital output for processing in microcontrollers or data acquisition systems.

The output signal can be used for:

* Displaying the measured current on an ammeter.
* Storing data for further analysis.
* Implementing control mechanisms in systems like the ROV.
  + 1. ***Interfacing the ACS712 Sensor with the Microcontroller***

The ACS712 sensor outputs an analog voltage proportional to the current passing through it. This output is connected to the ADC (Analog-to-Digital Converter) pin of the microcontroller for processing.

* + 1. ***Current Calculation Formula***

The current can be calculated using the following formula:

Where:

* Zero Current ADC Value: The ADC value when no current is flowing (e.g., 512 for 2.5V at 10-bit resolution).
* ADC Resolution: 1023 for a 10-bit ADC.
* VCC: Supply voltage (5V).
* Sensitivity: 66 mV/A.
  + 1. ***Pseudocode***
* Main File

BEGIN

// Define constants and variables

CURRENT\_PIN = A0

INPUT\_VOLTAGE = 5.0

ARDUINO\_ADC = 1023

SENSOR\_SCALE\_FACTOR = 66

current\_u16Interval = 1000

SAFEST\_CURRENT = 8

MIDPOINT\_CYCLES = 100

READING\_CYCLES = 10

// Initialize sensor and serial communication

current\_sensor = ACS712(CURRENT\_PIN, INPUT\_VOLTAGE, ARDUINO\_ADC, SENSOR\_SCALE\_FACTOR)

Serial.begin(9600)

// Calibrate sensor midpoint

current\_sensor.autoMidPoint(MIDPOINT\_CYCLES)

PRINT current\_sensor.getMidPoint()

// Main loop

WHILE TRUE

current\_u32CurrentMillis = millis()

IF current\_u32CurrentMillis - current\_u32PreviousMillis >= current\_u16Interval

current\_u32PreviousMillis = current\_u32CurrentMillis

Local\_u8DCReading = current\_sensor.mA\_DC(READING\_CYCLES) // Take average of 10 cycles

PRINT Local\_u8DCReading

IF Local\_u8DCReading > SAFEST\_CURRENT

// Stop motor or take action

END IF

END IF

END WHILE

END

* 1. ***MPU6050***
     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.

* + 1. ***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.

* + - 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.

* + - 1. ***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:

​

* 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:

* 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
    - 1. ***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.

* + - 1. ***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:

​​​

Calculating Tilt Angle from Gyroscope:

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

Where 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:

* is the filter constant, typically between 0.95 and 0.99, to prioritize gyroscope data.
  + 1. ***Software Interface***
       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.

* + - 1. ***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().
    - 1. ***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

* + 1. ***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 () in degrees per second (or radians per second).
    - Yaw is derived by integrating this angular velocity over time:
  + Any bias or noise in ​ 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 and in the horizontal plane.
    - The yaw angle is calculated as:
    - 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:
    - Here:
      * 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

* 1. ***HMC5883L 3-Axis Magnetometer Sensor***
     1. ***Overview***

The HMC5883L is a 3-axis digital magnetometer designed to measure magnetic fields in three perpendicular axes (X, Y, and Z). It is widely used in applications such as:

* Compassing: Determining heading or direction relative to the Earth's magnetic field.
* Navigation: Assisting in orientation and positioning systems.
* Robotics: Enabling robots to sense and navigate their environment.
* Consumer Electronics: Used in smartphones, tablets, and wearables for orientation sensing.

The sensor communicates via the I2C protocol, making it easy to interface with microcontrollers like Arduino.

* + 1. ***Operating Principle***
       1. ***Magnetic Field Measurement***

The HMC5883L measures the Earth's magnetic field using its magneto resistive sensors. Each axis (X, Y, Z) outputs a voltage proportional to the magnetic field strength. The ADC converts these voltages into digital values, which are then processed to calculate the heading or direction.

* + - 1. ***Heading Calculation***

The heading (direction) is calculated using the arctangent of the Y and X axis data:

* If , the heading is:
* If , the heading is:
* If :
  + If , the heading is .
  + If , the heading is .
    - 1. ***Measurement Modes***
* Single-Measurement Mode:
  + The sensor takes a single reading and then enters idle mode.
  + Suitable for low-power applications.
* Continuous-Measurement Mode:
  + The sensor continuously takes readings at a specified data rate.
  + Ideal for real-time applications.
    1. ***Calibration***

The HMC5883L sensor may have inherent offsets and sensitivity variations due to manufacturing tolerances and external magnetic interference. Calibration ensures accurate and reliable measurements.

* + - 1. ***Calibration Procedure***
* Collect Raw Data:
  + Rotate the sensor slowly in all directions (360° in the X-Y plane).
  + Record the raw X, Y, and Z values for multiple orientations.
* Calculate Offsets:
  + For each axis, calculate the average of the minimum and maximum values:
* Calculate Scale Factors:
  + Normalize the data to account for sensitivity differences between axes:
* Apply Calibration:
  + Subtract the offsets and multiply by the scale factors to correct the raw data:
    1. ***Software Interface***
* Initialization
  + Configure the I2C communication protocol.
  + Set the sensor's configuration registers (e.g., data rate, gain, measurement mode).
* Data Acquisition
  + Read the raw magnetic field data from the X, Y, and Z axis registers.
  + Apply calibration coefficients to compensate for sensor offsets and sensitivity variations.
  + Calculate the heading using the arctangent function.
    - 1. ***Pseudocode***
* Header File

DECLARE FUNCTION initHMC5883L()

DESCRIPTION: Initializes the HMC5883L sensor and configures its settings.

DECLARE FUNCTION setMeasurementMode(mode)

INPUT: mode (Integer)

DESCRIPTION: Sets the measurement mode (single or continuous).

DECLARE FUNCTION setDataRate(rate)

INPUT: rate (Integer)

DESCRIPTION: Sets the data output rate.

DECLARE FUNCTION setGain(gain)

INPUT: gain (Integer)

DESCRIPTION: Sets the sensor gain.

DECLARE FUNCTION readRawData(data)

INPUT: data (Array of 3 integers)

DESCRIPTION: Reads raw magnetic field data from the X, Y, and Z axes.

DECLARE FUNCTION calibrateSensor(offsets, scaleFactors)

INPUT: offsets (Array of 3 floats), scaleFactors (Array of 3 floats)

DESCRIPTION: Applies calibration offsets and scale factors to raw data.

DECLARE FUNCTION calculateHeading(x, y) RETURNS Float

INPUT: x (Float), y (Float)

DESCRIPTION: Calculates the heading from X and Y axis data.

DECLARE FUNCTION getHeading() RETURNS Float

DESCRIPTION: Returns the current heading in degrees.

* Source File

BEGIN PROGRAM

DEFINE I2C\_ADDRESS = 0x1E

DEFINE offsets AS float array of size 3 initialized to 0

DEFINE scaleFactors AS float array of size 3 initialized to 1

FUNCTION initHMC5883L()

INITIALIZE I2C communication

SET configuration registers:

- Measurement mode: Continuous

- Data rate: 15 Hz

- Gain: 1090 LSB/Gauss

END FUNCTION

FUNCTION setMeasurementMode(mode)

WRITE mode to measurement mode register

END FUNCTION

FUNCTION setDataRate(rate)

WRITE rate to data rate register

END FUNCTION

FUNCTION setGain(gain)

WRITE gain to gain register

END FUNCTION

FUNCTION readRawData(data)

READ 6 bytes from data output registers

COMBINE bytes into X, Y, and Z axis data

STORE data in the provided array

END FUNCTION

FUNCTION calibrateSensor(offsets, scaleFactors)

FOR each axis (X, Y, Z) DO

APPLY offset and scale factor to raw data

END FOR

END FUNCTION

FUNCTION calculateHeading(x, y)

RETURN atan2(y, x) \* 180 / PI

END FUNCTION

FUNCTION getHeading()

DECLARE rawData AS integer array of size 3

CALL readRawData(rawData)

CALL calibrateSensor(offsets, scaleFactors)

RETURN calculateHeading(rawData[0], rawData[1])

END FUNCTION

END PROGRAM

* Main File

INCLUDE LIBRARY: HMC5883L.h

DEFINE FUNCTION setup()

BEGIN

START serial communication at 9600 baud rate.

CALL initHMC5883L() to initialize the sensor.

END

DEFINE FUNCTION loop()

BEGIN

PRINT "Heading = " to Serial Monitor.

PRINT RETURN VALUE of getHeading() to Serial Monitor.

WAIT for 1000 milliseconds.

END

* 1. ***Temperature Sensor***
     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. ***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. ***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.

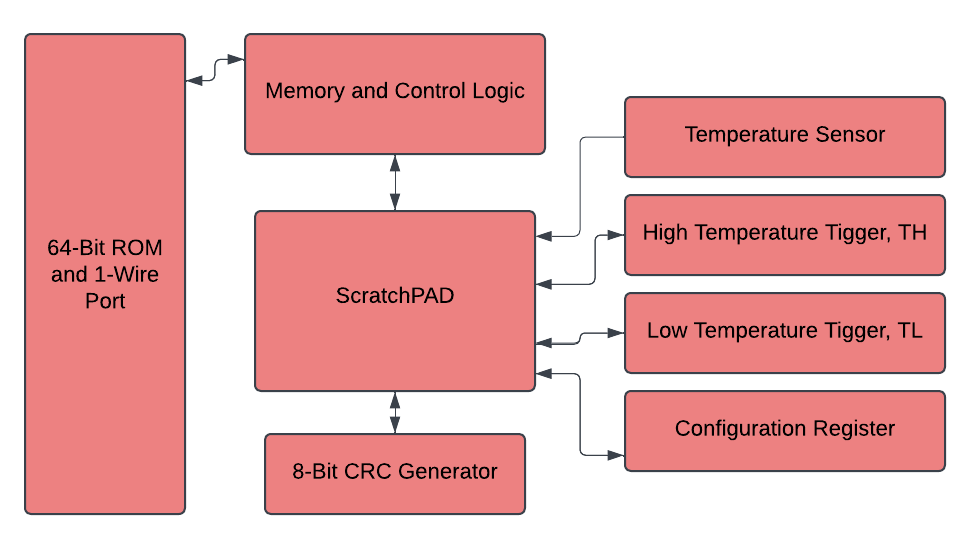


Figure (1)

* + 1. ***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. ***Sensor Interface***
       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. ***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. ***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. ***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. ***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. ***Pseudocode***
* Header file

DECLARE class TemperatureSensor

METHOD: begin()

METHOD: setResolution(resolution)

METHOD: readTemperature()

ENDCLASS

* 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

* 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

* 1. ***Pressure and Depth Sensor***
     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.

* + 1. ***Functional Description***
       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.

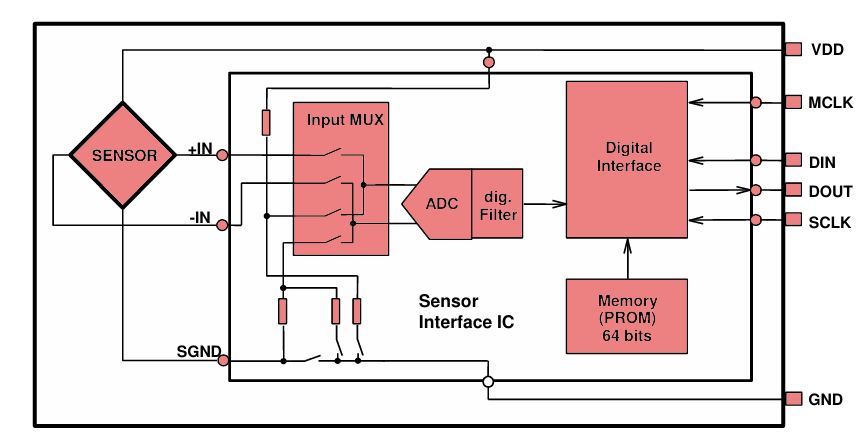


Figure (3)

* + - 1. ***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.

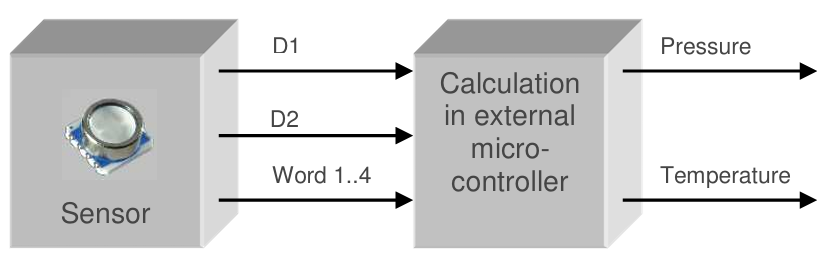


Figure (4)

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).

A graph with lines and numbers

Description automatically generated

Figure 5

A graph of pressure and pressure

Description automatically generated

Figure 6

* + - 1. ***Compensation Algorithm***

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

* *D1*: Raw pressure data.
* *D2*: Raw temperature data.
* *A, B, C, P1, P2, P3*: PROM coefficients.
  + 1. ***Software Interface***

1. 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.
2. 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.
3. Send the reset command to initialize communication and set the default states.
4. Read the PROM (Programmable Read-Only Memory) to fetch the calibration coefficients.
5. Send the Convert D1 command to initiate the pressure data acquisition process.
6. Wait for the specified conversion time, which is approximately 35 ms for high resolution.
7. Send the ADC Read command to retrieve the raw pressure data (D1).
8. Repeat the above steps for obtaining temperature data (D2).
9. Use the calibration coefficients read from PROM.
10. Apply compensation equations to calculate corrected temperature and pressure values.
11. Convert pressure data to depth:
    * 1. ***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

* + 1. ***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.

* + - 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.

1. ***Sensor Fusion***
2. ***Control Systems***
   1. ***PID Control***

A Proportional-Integral-Derivative (PID) controller is a feedback-based control loop mechanism widely used in industrial and automation systems to regulate processes requiring continuous control and automatic adjustments. It operates by comparing a desired target value (setpoint or SP) with the actual system output (process variable or PV). The difference between these values, known as the error (e(t)), is used to apply corrective actions through three control terms: Proportional (P), Integral (I), and Derivative (D).

Key Components of a PID Controller:

* Proportional (P): Responds to the current error magnitude, providing immediate correction proportional to the error.
* Integral (I): Addresses residual steady-state errors by integrating past errors over time.
* Derivative (D): Predicts future error trends based on the rate of change, reducing overshoot and improving stability.

PID controllers enhance automation by minimizing human intervention and errors, ensuring precise control.

* + 1. ***Fundamental Operation***

The PID controller continuously calculates the error e(t) = SP – PV and applies corrections using a weighted sum of the P, I, and D terms. The control variable u(t) (e.g., valve position) is adjusted to minimize the error over time.

* Proportional Term (P): Directly proportional to the current error. A high proportional gain (Kp) results in a stronger response but may cause instability if too high.
* Integral Term (I): Eliminates steady-state errors by integrating past errors. However, excessive integral gain (Ki) can cause overshoot.
* Derivative Term (D): Dampens system response by predicting future errors based on the error rate of change. High derivative gain (Kd) improves stability but can amplify noise.
  + 1. ***Tuning***

PID controllers require tuning to balance the P, I, and D terms for optimal performance. Tuning involves adjusting the gains (Kp, Ki, Kd) based on the system's response characteristics. Common tuning methods include:

* Manual Tuning: Adjust gains iteratively to achieve desired performance.
* Ziegler-Nichols Method: A systematic approach to determine initial gain values.
* Software-Based Tuning: Advanced tools automate tuning by analyzing system responses and suggesting optimal parameters.
  + 1. ***Mathematical Form***

The PID control function is expressed as:

Where:

* *Kp*​: Proportional gain
* *Ki*​: Integral gain
* *Kd*​: Derivative gain
* *e*(*t*): Error (SP – PV)

In the Laplace domain, the transfer function is:

* + 1. ***Selective Use of Control Terms***

Not all applications require all three terms. Common configurations include:

* PI Controller: Used when derivative action is sensitive to noise.
* PD Controller: Applied where integral action is unnecessary.
* P or I Controller: Simplified controllers for specific use cases.
  + 1. ***Controller Theory***

The PID controller output is the sum of the P, I, and D terms. The manipulated variable (MV) is calculated as:

* + 1. ***Detailed Control Terms***
       1. ***Proportional Term***
* Output:
* High *Kp*​ improves responsiveness but risks instability.
* Low *Kp*​ results in sluggish control.
  + - 1. ***Integral Term***
* Output:
* Eliminates steady-state errors but can cause overshoot.
  + - 1. ***Derivative Term***
* Output:
* Improves stability by damping oscillations but is sensitive to noise.
  + 1. ***Loop Tuning***

Tuning ensures stability and optimal performance. Key considerations:

* Stability: Avoid excessive gains to prevent oscillations.
* Manual Tuning: Start with Kp*Kp*​, then adjust Ki*Ki*​ and Kd*Kd*​.
* Software Tools: Automate tuning for complex systems.
  + 1. ***Common Issues and Solutions***
       1. ***Integral Windup***
* Cause: Integral term accumulates excessive error during large setpoint changes.
* Solution: Disable integration or limit integral term bounds.
  + - 1. ***Overshooting***
* Cause: Rapid changes in setpoint or disturbances.
* Solution: Use setpoint ramping or derivative of the process variable.
  + 1. ***PID Controller Simulation Using Simulink and Output Response***

This section documents the simulation of a closed-loop control system using a PID controller and a Brushless DC Motor (BLDC) as the plant. The goal is to achieve a desired output response by tuning the PID parameters.

* + 1. ***Simulation Overview***

The simulation models a closed-loop system where:

**A diagram of a computer

Description automatically generated**

* The setpoint is the desired output value.
* The PID controller adjusts the control input to minimize the error between the setpoint and the process variable (PV).
* The BLDC motor represents the plant, and its dynamics are modeled using a transfer function.
* The output response is observed and tuned using a Scope.
  + 1. ***Components of the Simulation***
* Constant Block: Represents the setpoint (desired value).
* Sum Block: Computes the error e(t)=Setpoint−Process Variable (PV)*e*(*t*)=Setpoint−Process Variable (PV).
* PID Controller: Applies proportional, integral, and derivative actions to correct the error.
* Transfer Function Block: Models the BLDC motor dynamics using the transfer function:
* Scope: Visualizes the system’s output response for analysis and tuning.
  + 1. ***Tuning Process and Output Response***

The PID controller was tuned manually to achieve the desired output response. The steps and observations are documented below:

Step 1: Initial Setup

* Set *Kp*​=0, *Ki*​=0, and *Kd*​=0.
* Observation: The process variable (PV) is far from the setpoint, indicating no control action.

A graph with a line in the middle

Description automatically generated

Step 2: Tuning the Proportional Gain (*Kp*​)

* Gradually increase *Kp*​ to reduce the steady-state error.
* Observation: As *Kp*​ increases, the steady-state error decreases significantly.
* Key Observations:
  + If *Kp*​ is too low, the system response is slow and sluggish.
  + If *Kp*​ is too high, the system becomes unstable and oscillates.

A graph with lines and numbers

Description automatically generated

Step 3: Tuning the Integral Gain (*Ki*​)

* Increase *Ki*​ to eliminate residual steady-state error.
* Observation: The steady-state error is eliminated, but overshooting increases.
* Key Observations:
  + If *Ki*​ is too low, the steady-state error persists.
  + If *Ki*​ is too high, the system becomes slower and less stable.

A graph on a black background

Description automatically generated

Step 4: Tuning the Derivative Gain (*Kd*​)

* Increase *Kd*​ to reduce oscillations and overshooting.
* Observation: The system achieves a stable response with minimal overshooting and no steady-state error.
* Final Tuned Parameters:
  + *Kp*​=10
  + *Ki*​=10
  + *Kd*​=3

**A graph with a line going up

Description automatically generated**

* 1. ***Thrusters and Movement Control***
     1. ***Thruster Configurations***
* Front Thrusters:
  + T1: Front-right thruster (45° from the centerline).
  + T2: Front-left thruster (-45° from the centerline).
* Back Thrusters:
  + T3: Back-right thruster (45° from the centerline).
  + T4: Back-left thruster (-45° from the centerline).
* Vertical Thrusters:
  + T5: Front vertical thruster.
  + T6: Back vertical thruster.

Here’s a diagram of the thruster arrangement in Figure (6)

* + 1. ***Equations for Thrust Calculation***

The equations for this configuration will account for the 45-degree angles of the horizontal thrusters. Each thruster contributes to surge, sway, yaw, and roll based on its orientation.

* + - 1. ***Horizontal Thrusters (T1, T2, T3, T4)***

These thrusters control surge, sway, yaw, and roll:

* Explanation:
  + surge: Controls forward/backward motion.
    - Positive surge = forward motion.
    - Negative surge = backward motion.
  + sway: Controls left/right motion.
    - Positive sway = right motion.
    - Negative sway = left motion.
  + outputYaw: Controls rotation (yaw).
    - Positive outputYaw = rotate counterclockwise.
    - Negative outputYaw = rotate clockwise.
  + outputRoll: Controls tilting (roll).
    - Positive outputRoll = tilt right.
    - Negative outputRoll = tilt left.

Each thruster contributes to these motions based on its orientation:

* T1 (Front-Right, 45°): Combines surge, sway, yaw, and roll.
* T2 (Front-Left, -45°): Combines surge, sway, yaw, and roll.
* T3 (Back-Right, 45°): Combines surge, sway, yaw, and roll.
* T4 (Back-Left, -45°): Combines surge, sway, yaw, and roll.
  + - 1. ***Vertical Thrusters (T5, T6)***

These thrusters control heave and pitch:

* Explanation:
  + outputHeave: Controls up/down motion.
    - Positive outputHeave = upward motion.
    - Negative outputHeave = downward motion.
  + outputPitch: Controls tilting forward/backward.
    - Positive outputPitch = tilt forward.
    - Negative outputPitch = tilt backward.

Each vertical thruster contributes to heave and pitch:

* T5: Combines heave and pitch.
* T6: Combines heave and pitch.
  + 1. ***How Thrust Vectors Work***

The thrust vectors are calculated by combining the contributions of each motion (surge, sway, heave, roll, pitch, yaw) for each thruster:

* + - 1. ***Horizontal Thrusters (T1–T4)***
* Surge and Sway:
  + The horizontal thrusters are at 45-degree angles, so they contribute to both surge and sway.
  + For example:
    - T1 (Front-Right, 45°) contributes equally to surge and sway.
    - T2 (Front-Left, -45°) contributes positively to surge and negatively to sway.
* Yaw and Roll:
  + Yaw is achieved by differential thrust between thrusters on opposite sides.
  + Roll is achieved by differential thrust between thrusters on the same side.
    - 1. ***Vertical Thrusters (T5, T6)***
* Heave:
  + Heave is achieved by equal thrust from both vertical thrusters (T5, T6).
* Pitch:
  + Pitch is achieved by differential thrust between the vertical thrusters.
    1. ***Example Scenario***
* Inputs:
  + surge = 1.0 (forward motion).
  + sway = 0.5 (right motion).
  + outputYaw = 0.2 (rotate counterclockwise).
  + outputRoll = 0.1 (tilt right).
  + outputHeave = 0.8 (upward motion).
  + outputPitch = 0.3 (tilt forward).
* Calculations:
  + Horizontal Thrusters:
  + Vertical Thrusters:
* Interpretation:
  + T1: Strong forward and right thrust.
  + T2: Slight forward thrust.
  + T3: Slight backward thrust.
  + T4: Strong backward and left thrust.
  + T5: Strong upward thrust with forward tilt.
  + T6: Moderate upward thrust with backward tilt.