DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
NORTH SOUTH UNIVERSITY

SENIOR DESIGN PROJECT

ENVIRONMETRIC

FACULTY ADVISOR

DR. SHAHNEWAZ SIQQIQUE

ASSISTANT PROFESSOR

NUREN SHAMS 1530784043
F M A FAISAL SAKIB 1510709042
ABRAR FAHIM 1610522043
MD. JABER HOSSAIN 1610718043

DECLARATION

This is to declare that no part of this report or the project has been previously submitted elsewhere for the fulfillment of any other degree or program. Proper acknowledgement has been provided for any material that has been taken from previously published sources in the bibliography section of this report.

_____
Nuren Shams
1530784043
ECE Department
North South University, Bangladesh

_____
F M A Faisal Sakib
1510709042
ECE Department
North South University, Bangladesh

_____
Abrar Fahim
1610522043
ECE Department
North South University, Bangladesh

_____
Md. Jaber Hossain
1610718043
ECE Department
North South University, Bangladesh

APPROVAL

The capstone project entitled "EnvironMetric: A Standalone Pipeline to Collect, Process, Archive, Analyze & Visualize Atmospheric Pollutants" by Nuren Shams (ID# 153 0784 043), F M A Faisal Sakib (ID# 151 0709 042), Abrar Fahim (ID# 161 0522 043) and Md. Jaber Hossain (ID# 161 0718 043) has been accepted as satisfactory and is approved to partially meet the requirements for obtaining their respective Bachelor of Science Degrees from the Department of Electrical and Computer Engineering on January 2019.

Signature of the Project Supervisor

_____

Dr. Shahnewaz Siddique
Assistant Professor
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh

Signature of the Department Chairman

_____

Dr. K. M. A. Salam
Professor
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh

ACKNOWLEDGEMENT

EnvironMetric is the culmination of what we have learned in our years at North South University. It represents our hard work and effort as much as the faculties' who went above and beyond to prepare us not only to conceive & undertake such a project but see it through to success.

We express our sincerest gratitude to Dr. Shahnewaz Siddique for guiding us through the ups and downs of our project and helping us navigate the difficult aspects, both technical and logistical, of our project. And our heartfelt gratefulness for believing in us and allowing us the creative freedom that we needed to tackle the project.

ABSTRACT

The EnvironMetric system is a standalone pipeline designed to collect, process, archive, analyze and visualize atmospheric pollution data. The system consists of four principal components - The Remote Sensor Array to collect and process the data; The Dashboard to archive and perform micro-level visualizations on the data; DataStudio to analyze and perform macrolevel visualizations on the data; UAV Carrier Platform to allow the Remote Sensor Array to collect spatial data.

TABLE OF CONTENTS

Introduction

Our physical environment is an intricately balanced dynamical system. This system has remained in a state of equilibrium for centuries that is until we came along with our industrial revolution and changed everything. As much as our industries have done to improve our way of life, it also had an adverse effect on the natural environment. We are burning fossil fuels at an alarmingly increasing rate [1] to drive our industries and this is producing huge amounts of carbon-di-oxide [2]. In addition to carbon-di-oxide, various industries also release other hazardous gasses like methane into the atmosphere. Methane is a more potent greenhouse gas than carbon-di-oxide and records show a shocking increase in its atmospheric concentration in recent decades [3]. The rate of deforestation has also been at an all-time high in the last few years and is showing a steady increasing trend [4]. With an increasing population that is projected to continue increasing over the next century [5], climate change is as real as it gets with sharp rises in global CO_2 and CH_4 levels and a noticeable increase in global temperature [6]

In addition to the greenhouse gas emissions, a type of pollution known as the Particulate Matter (PM) pollution also exists in the atmosphere. Components of PM pollution include finely divided solids or liquids such as dust, fly ash, soot, smoke, aerosols, fumes, mists and condensing vapors that can remain suspended in the air for extended periods of time. PM originate from a variety of stationary and mobile sources and may be directly emitted (primary emissions) or formed in the atmosphere (secondary emissions) by transformation of gaseous emissions. Primary PM sources are derived from both human and natural activities. A significant portion of PM sources is generated from a variety of human (anthropogenic) activity. These types of activities include agricultural operations, industrial processes, combustion of wood and fossil fuels, construction and demolition activities, and entrainment of road dust into the air. Natural (nonanthropogenic or biogenic) sources also contribute to the overall PM problem. These include windblown dust and wildfires. Secondary PM sources directly emit air contaminants into the atmosphere that form or help form PM. Hence, these pollutants are considered precursors to PM formation. These secondary pollutants include SOx, NOx, VOCs, and ammonia.

Motivation

The adverse effects of climate change and pollution are most apparent in the developing countries. These countries must often adopt technologies and practices in order to keep up with the developed world but don't always have the necessary infrastructure to support and properly sustain them; a prime example of which is Bangladesh. In recent time Bangladesh has been making strides in economic and industrial development but has made headlines with Dhaka being one of the most polluted cities in the world. With an AQI score of 163, it was ranked as the 2nd/3rd most polluted city according to many sources. According to official reports at least 123,000 people died in Bangladesh in 2017 due to indoor and outdoor air pollution. The pollution reduced the average life expectancy of adults by as much as 1.3 years. Children growing up in the polluted environment have it much worse with their life-expectancy being shortened by up to 2.5 years. The main culprit responsible for this is the PM2.5. PM2.5 are Particulate Matters that are about 2.5 micrometers in diameter and are small enough to be breathed deep into the lungs. This has devastating short- and long-term health effects.

It is crucial to monitor the changes in the environment and pollution levels and enable environmental protection agencies to regulate and suppress the sources of pollution. Existing weather and pollution monitoring systems are mostly stationary and rely on multiple sensor nodes located at different sites to collect spatial data. Because of their high expense, the sensor nodes are few in numbers and provide (spatially) sparse data which may lead to less accurate data interpolations. Additionally, areas of heavy pollution and contamination may be inaccessible or even hazardous for humans to physically visit.

Problem Statement

Many countries have undergone the rapid industrial development as Bangladesh did. But unlike Bangladesh, they had monitoring and regulatory systems in place which allowed such rapid growth without drastically increasing their air pollution levels. Looking into the matter deeply reveals that although the Bangladeshi government does set a regulatory standard for industries to follow, they don't have the necessary equipment to collect real-world and real-time data to enforce their standards upon the industries.

Project EnvironMetric aims to deliver a complete solution that will be capable of collecting and processing the data to present them in a format that can be read and understood by domain experts, thus giving them actionable intelligence to enforce the standards that are set by the government.

Solution Approach

This project aims to create a comprehensive system, one that is modularized but also unified in its functionality; as such, it requires elements from a multitude of disciplines ranging from low-level hardware interfacing through mid-level algorithm design and going all the way up to the cutting edge of computation and artificial intelligence. The EnvironMetric System consists of 9 core components as described in Figure 1

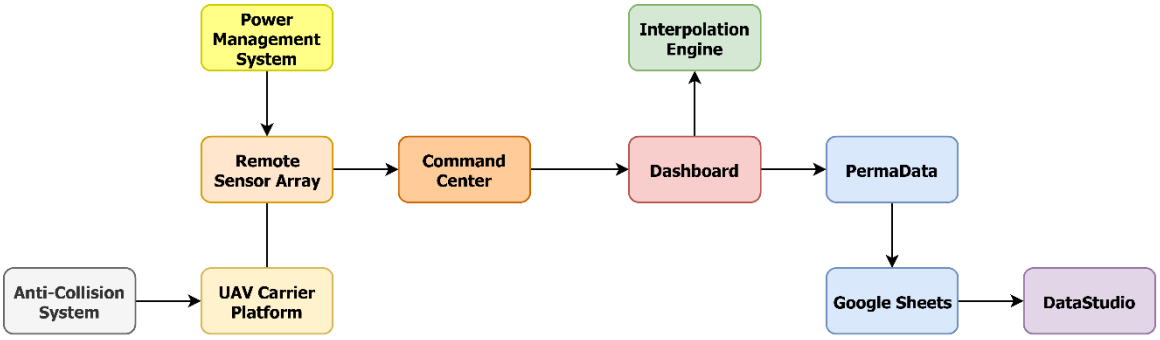


Figure 1: System level block diagram of the EnvironMetric core architecture

1: Anti-Collision System (ACS)

The Anti-Collision System is a closed-loop feedback controller that aids autonomous UAV navigation through preemptive collision detection. It uses the Intel RealSense D435i stereoscopic camera to provide depth & color video streams. The two frames are merged to segment the image & identify distinct objects within each compound frame using a CUDA accelerated version of the Super-pixel Linear Iterative Clustering algorithm running on a Nvidia Jetson Nano. The segmented image is superimposed on top of the raw depth frame & used as a mask to isolate objects of interest & calculate their distance. Afterwards, Perspective geometry is used to determine the UAV's flight tunnel within its current heading & a simple threshold-based controller is used to predict possible collisions & the appropriate control signals are transmitted to the UAV's flight controller through its telemetry channel.

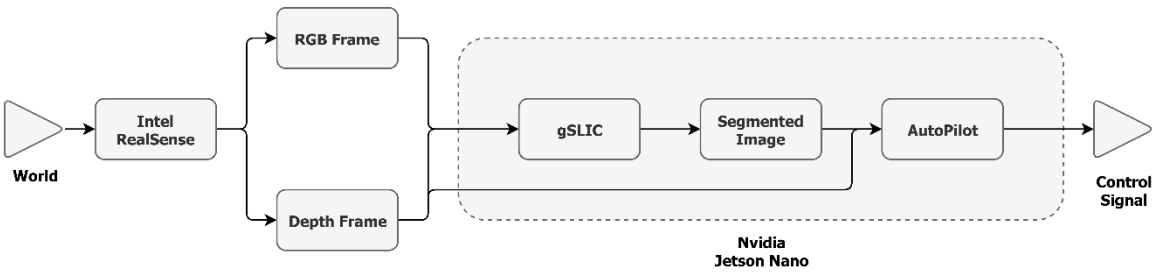


Figure 2: System level block diagram of the Anti-Collision System (ACS) architecture

1.1: Intel RealSense D435i

A stereo camera is a type of camera with two or more lenses with a separate image sensor for each lens. This allows the camera to simulate human binocular vision, giving it the ability to capture three-dimensional images, a process known as stereo photography. The distance between the lenses in a typical stereo camera (intra-axial distance) is about the distance between one's eyes (known as the intra-ocular distance) and is about 6.35 cm. [7]

For the purposes of the UAV's ACS, the Intel RealSense D435i stereo vision depth camera system was utilized. The subsystem assembly contains a stereo depth module and a vision processor with USB 2.0/USB 3.1 Gen 1 or MIPI1 connection to host processor. This camera sub system was specifically chosen due to its small size and ease of integration with the UAV's frame by using the 1/4-20 UNC screw mount on the camera chassis. The stereo depth module has two camera sensors referred to as imagers, they are identical parts and are configured with identical settings. The imagers are labeled "left" and "right" from the perspective of the camera module looking outward. The module also features an infrared projector which improves the ability of the stereo camera system to determine depth by projecting a static infrared pattern on the scene to increase texture on low texture scenes. The infrared projector meets class 1 laser safety under normal operation. The power delivery and laser safety circuits are on the stereo depth module. Additionally, the module also houses a color sensor to provide RGB texture information. [8]

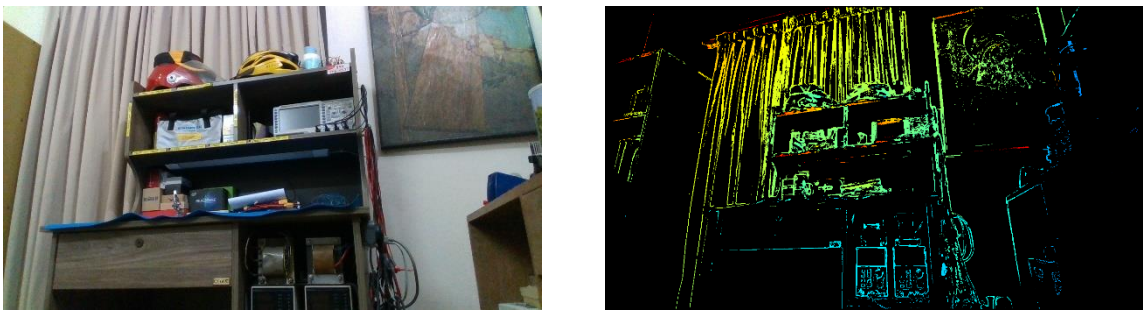


Figure 3: RGB image (left) and corresponding Depth point cloud (right) from the D435i video stream

Figure 4: Intel RealSense D435i (front); depicting (from left-to-right) left-imager, IR-projector, right-imager & color sensor

Figure 5: Intel RealSense D435i; rear view (left) and top view (right)

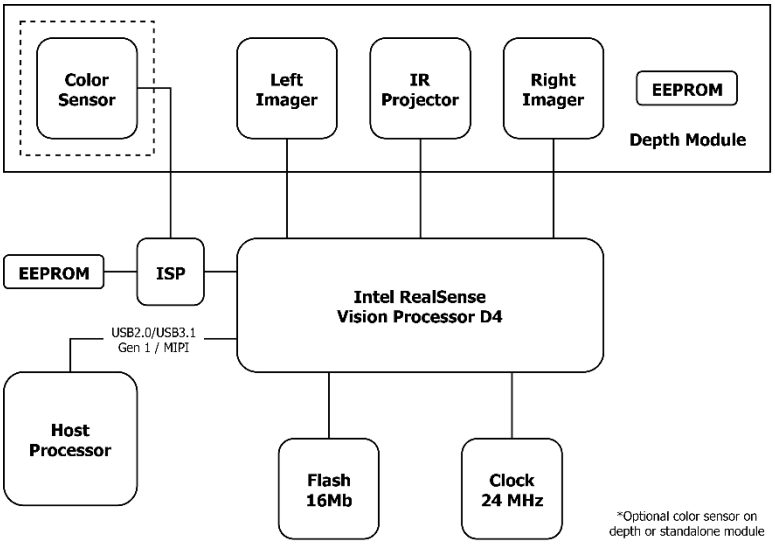


Figure 6: System level block diagram of the D435i

Image Sensor	OmniVision OV2740
Color Image Signal Processor	Discrete
Active Pixels	1920 X 1080
Sensor Aspect Ratio	16:9
Format	10-bit RAW RGB
F Number	f/2.0
Focal Length	1.88mm
Filter Type	IR Cut Filter
Focus	Fixed
Shutter Type	Rolling Shutter
Signal Interface	MIPI CSI-2, 1 Lane
Horizontal Field of View	69.4°
Vertical Field of View	42.5°
Diagonal Field of View	77°
Distortion	≤1.5%

Table 1: Color Sensor Properties

Image Sensor	OmniVision OV9282
Active Pixels	1280 X 800
Sensor Aspect Ratio	8:5
Format	10-bit RAW
F Number	f/2.0
Focal Length	1.93mm
Filter Type	None
Focus	Fixed
Shutter Type	Global Shutter
Signal Interface	MIPI CSI-2, 2X Lanes
Horizontal Field of View	91.2°
Vertical Field of View	65.5°
Diagonal Field of View	100.6°
Distortion	≤1.5%

Table 2: Left and Right Imager Properties

Projector	Infrared
Pattern Type	Static
Illuminating Component	Vertical-cavity surface-emitting laser (VCSEL) + optics
Laser Controller	PWM
Optical Power	360mW average, 4.25W peak
Laser Wavelength	850nm ± 10 nm nominal @ 20°C
Laser Compliance	Class 1, IEC 60825-1:2007 Edition 2, IEC 60825-1:2014 Edition 3
Horizontal Field of Projection	86°±3°
Vertical Field of Projection	57°±3°
Diagonal Field of Projection	94°±3°

Table 3: Infrared Projector Properties

Use Environment	Indoor/Outdoor
Image Sensor Technology	Global Shutter, 3µm x 3µm pixel size
Camera Module	Intel RealSense Module D430 + RGB Camera
Vision Processor Board	Intel RealSense Vision Processor D4
Depth Technology	Active IR Stereo
Minimum Depth Distance (Min-Z)	0.105 m
Maximum Range	Approx. 10 meters. Accuracy varies depending on calibration, scene, and lighting condition
Depth Field of View (FOV)	87°±3° x 58°±1° x 95°±3°
Depth Output Resolution & Frame Rate	Up to 1280 x 720 active stereo depth resolution. Up to 90 fps
RGB Sensor FOV (H x V x D)	69.4° x 42.5° x 77° (+/- 3°)
RGB Sensor Resolution	1920 x 1080
RGB Frame Rate	30 fps
Form Factor	Camera Peripheral
Connectors	USB-C 3.1 Gen 1
Length x Depth x Height	90 mm x 25 mm x 25 mm
Mounting Mechanism	One 1/4-20 UNC thread mounting point. Two M3 thread mounting points

Table 4: General Overview

1.2: Super-pixel Linear Iterative Clustering (SLIC)

For a stereoscopic imaging based Anti-Collision System implementation, the distances to each individual pixel in each depth frame from the D435i video stream must be calculated in real-time (~30 fps at minimum) on an edge device with limited computational capabilities; the use of cloud-based remote computation was intentionally avoided due to high bandwidth requirements and potentially high latency. The solution was to use unsupervised learning algorithms to perform image segmentation of pixels with similar characteristics, in terms of color and depth/distance values, so that the ACS would have to keep track of a much smaller number of (super-)pixels.

The very first approach used K-Means with a pixel's RGBD values to segment an image into a predefined number of clusters. While the approach did cluster the image, it essentially produced a lower color depth version of the original image. This meant that even non-contiguous pixels were classified into the same cluster if they all have similar RGBD values. Although this is not an issue in the case of an Anti-Collision System that merely detects collision and restricts the UAV's movement in the obstacle's general direction, it will be an issue when the ACS operation concepts are translated to the Collision-Avoidance System (CAS); CAS is intended to be a future extension of the ACS whereby the UAV is autonomously piloted away from obstacles while still maintaining the general flight heading. For the CAS, it is necessary for the pixel clusters to only consist of spatially contiguous pixels with similar RGBD values.

The second approach used X and Y coordinates of the pixel in addition to the RGBD values to cluster the pixels with K-Means. While this performed better in terms of spatially localized clustering, it was quite slow to calculate each frame; managing only ~8 fps with the GPU accelerated Python TensorFlow 1.0 running on a Nvidia GTX 1660 and an Intel Core i7 4770. Although the TensorFlow implementation is designed to leverage the GPU compute power, the bottleneck may have been a result of the Intel RealSense using the C++ drivers through the python bindings or it could have been due to the CPU and GPU having a non-unified memory architecture.

To circumvent the bottle necks, the entire algorithm was re-implemented directly on the Nvidia Jetson Nano using C++. The only difference being in that the vanilla K-Means algorithm was swapped out for a CUDA accelerated Super-pixel Linear Iterative Clustering algorithm implementation; as a side note, the SLIC algorithm still makes use of a K-Means based algorithm with the input image represented in the CIELAB color space. Referred to as gSLICr by its authors at the Oxford University [9], it performed much better at ~25 fps for a resolution of 640 x 480 pixels on the Nvidia Jetson Nano.

Algorithm

1: Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S
2: Perturb cluster centers in an $n \times n$ neighborhood, to the lowest gradient position
3: repeat
4: for each cluster center C_k do
5: Assign the best matching pixels from a $2S \times 2S$ square neighborhood around the cluster center according to the distance measure
6: end for
7: Compute new cluster centers and residual error E {$L1$ distance between previous centers and recomputed centers}
8: until $E \leq$ threshold
9: Enforce connectivity

1.3: Nvidia Jetson Nano

For the compute platform, the ACS sub system required a small form factor, low power device specialized for performing computer vision tasks. The Nvidia Jetson Nano was chosen for this purpose not only because it satisfied all the requirements, but it also belongs to a family of SBCs that included much more powerful offerings such as the Xavier and TX series of platforms. They all share an inter-compatible architecture of CUDA core driven GPU and an ARM based CPU which would allow for an efficient algorithm to be developed on the Nano platform and be ported over to a much more powerful platform like the Xavier with minimal effort.

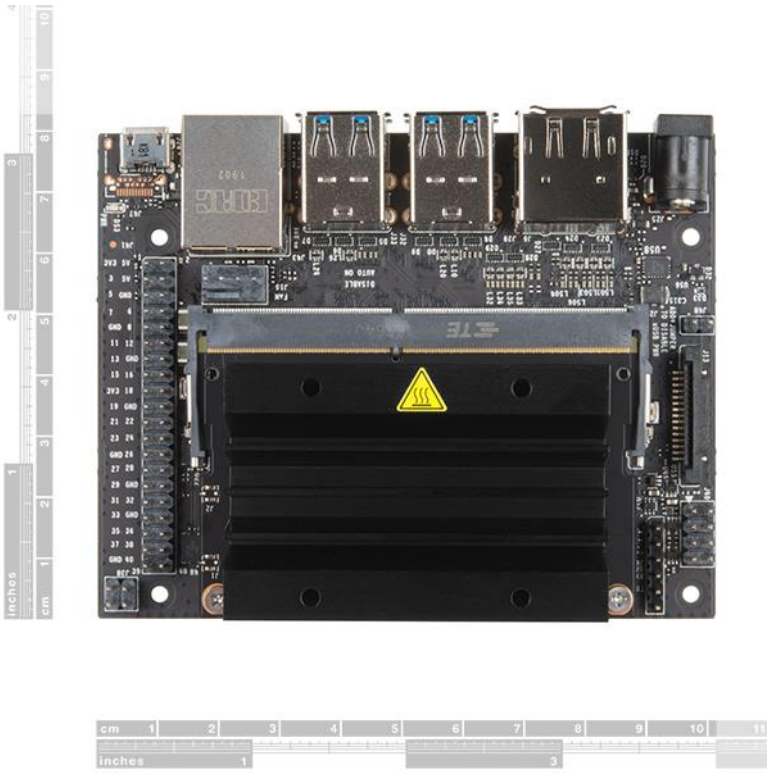


Figure 7: Nvidia Jetson Nano (top-down view) with scale

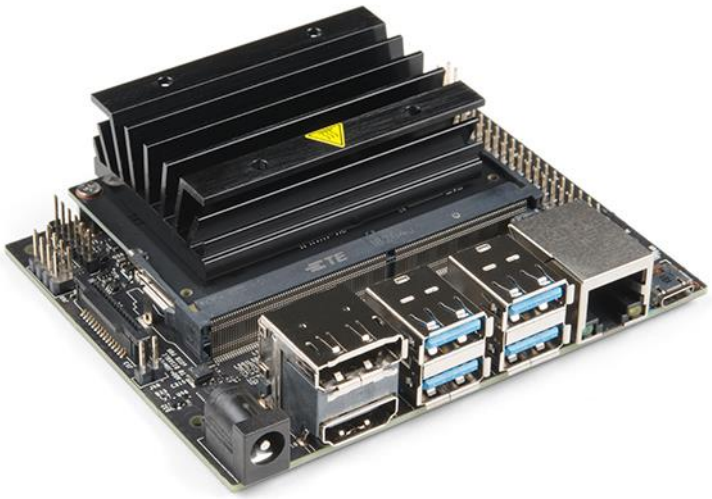


Figure 8: Nvidia Jetson Nano (rear view) depicting its various I/O ports

The Nvidia Jetson Nano Developer Kit is a small, powerful computer that can run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts. At just 70 x 45 mm, the Jetson Nano module is the smallest Jetson device. It is a production-ready System on Module (SOM) which delivers big when it comes to deploying AI to devices at the edge across multiple industries from smart cities to robotics. The Jetson Nano delivers 472 GFLOPs for running modern AI algorithms fast. It runs multiple neural networks in parallel and processes several high-resolution sensors simultaneously, making it ideal for applications like entry-level Network Video Recorders (NVRs), home robots, and intelligent gateways with full analytics capabilities.

Jetson Nano is also supported by Nvidia JetPack, which includes a board support package (BSP), Linux OS, Nvidia CUDA, cuDNN, and TensorRT software libraries for deep learning, computer vision, GPU computing and multimedia processing. The software is distributed as an easy-to-flash SD card image, making it fast and easy to get started. The same JetPack SDK is used across the entire Nvidia Jetson family of products and is fully compatible with Nvidia 's AI platform for training and deploying AI software. This proven software stack reduces complexity and overall effort for developers.

Figure 9: Nvidia Jetson Nano module

Another key advantage of using the Jetson Nano or any other Jetson platform is their modularity. They all make use of separate boards for their System on Module (SOM) which allows them to be plugged into other (production-optimized) carrier boards.

GPU	NVIDIA Maxwell™ architecture with 128 NVIDIA CUDA® cores 0.5 TFLOPs (FP16)
CPU	Quad-core ARM® Cortex®-A57 MPCore processor
Memory	4 GB 64-bit LPDDR4 1600MHz - 25.6 GB/s
Storage	16 GB eMMC 5.1 Flash
Video Encode	250 MP/sec 1x 4K @ 30 (HEVC) 2x 1080p @ 60 (HEVC) 4x 1080p @ 30 (HEVC)
Video Decode	500 MP/sec 1x 4K @ 60 (HEVC) 2x 4K @ 30 (HEVC) 4x 1080p @ 60 (HEVC) 8x 1080p @ 30 (HEVC)
Camera	12 lanes (3x4 or 4x2) MIPI CSI-2 D-PHY 1.1 (18 Gbps)
Connectivity	Wi-Fi requires external chip 10/100/1000 BASE-T Ethernet
Display	HDMI 2.0 or DP1.2 | eDP 1.4 | DSI (1 x2) 2 simultaneous
UPHY	1 x1/2/4 PCIE, 1x USB 3.0, 3x USB 2.0
I/O	3x UART, 2x SPI, 2x I2S, 4x I2C, GPIOs
Size	69.6 mm x 45 mm
Mechanical	260-pin edge connector

Table 5: Nvidia Jetson Nano technical specifications

1.4: Flight Tunnel

Because the ACS specification includes different levels of warning -

- **Obstruction detected**: When an obstruction is inside the UAV's flight path but is more than 20m away
- **Possible Collision**: When an obstruction is inside the UAV's flight path and is less than 20m away
- **Imminent Collision**: When an obstruction is inside the UAV's flight path and is less than 10m away
- **Manual Override**: When an obstruction is within a 5m radius of the UAV

- the first three of which concern the UAV's flight path, it needs to understand the "tunnel" through which the UVA will fly through in three-dimensional space if it were to maintain its current heading. Assuming the stereo camera is mounted at the center of the UAV's chassis pointing in the same direction as the UAV's heading, the flight tunnel will be a three dimensional cuboid with its height and width matching that of the UAV's chassis and while its depth will be equal to the maximum distance sensing capability of the stereo camera.

To reduce the computational needs of such a three-dimensional navigation model, it may be flattened to two dimensions. But doing so will require the 3D flight tunnel (with its height, width and depth) to be represented in a two-dimensional space. This is where visual perspective and projective geometry come into play.

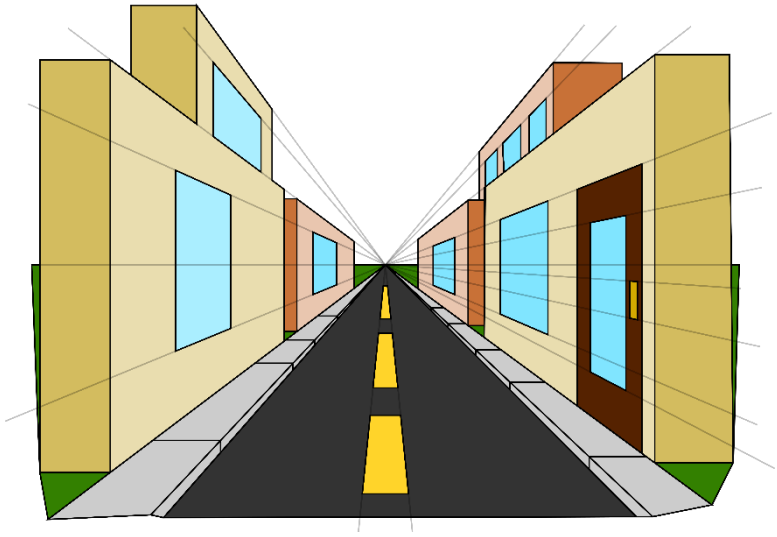


Figure 10: An illustration of visual perspective

Perspective is the art and mathematics of realistically depicting three-dimensional objects in a two-dimensional plane, sometimes called centric or natural perspective to distinguish it from bicentric perspective. The study of the projection of objects in a plane is called projective geometry. The principles of perspective drawing were elucidated by the Florentine architect F. Brunelleschi (1377-1446). These rules are summarized by Dixon (1991) [10]:

1. The horizon appears as a line.
2. Straight lines in space appear as straight lines in the image.
3. Sets of parallel lines meet at a vanishing point.
4. Lines parallel to the picture plane appear parallel and therefore have no vanishing point.

There is a graphical method for selecting vanishing points (The point or points to which the extensions of parallel lines appear to converge in a perspective drawing) so that a cube or box appears to have the correct dimensions (Dixon 1991). Using these principles, the 3D flight tunnel might be represented in a 2D space as depicted in Figure 11.

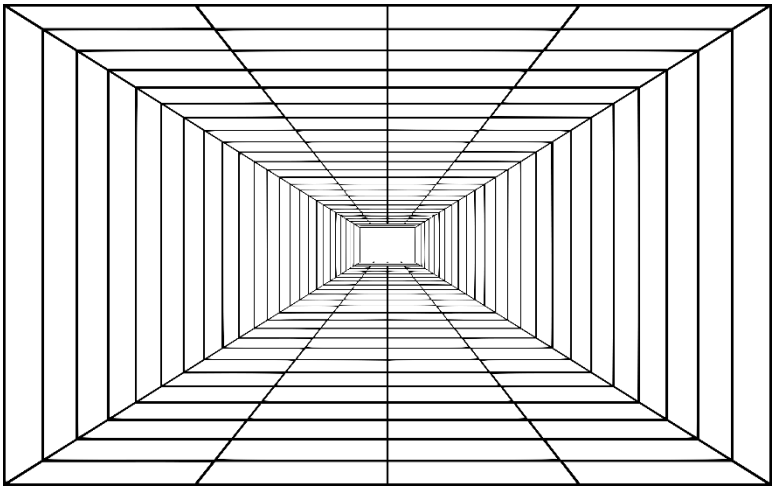


Figure 11: 2D representation of the 3D flight tunnel using projective geometry and the principles of centric perspective

Figure 11 illustrates a flight tunnel with 25 frames (25 individually nested rectangles). The purpose of this flight tunnel is to implement a threshold-based controller that only identifies objects as potential obstructions when they are in the correct position within the video frame. For example, if the depth sensor detects an object that is 25 meters away and is not in the center of the image i.e. outside of the UAV's flight path with its current heading, it would be meaningless for the ACS to raise a warning. The idea is to use four or more frames - at least one for each of the four different warning levels - to generate a flight tunnel through which the UAV is expected to fly if it maintains its current heading. If an object is detected within

one of the frames and it's closer than the threshold distance for that frame, the ACS will raise a warning and the UAV's Flight Controller will be signaled to take an appropriate action to change its heading or stop moving altogether.

1.5: ACS sub-controller

A fully functional secondary controller to detect and process the ACS warnings to aid the UAV's primary Flight Controller (Raspberry Pi 3B+ w/ the Navio2 HAT running the ArduPilot stack) was not implemented within the capstone project timeline; it remains to be implemented in future extensions of the project.

The specifications for this secondary controller call for a polling architecture based finite state model that would take the processed video frames from the gSLIC algorithm and detect if any of the superpixels are triggering any of the flight tunnel frames. If a trigger is detected, the secondary controller would send instructions to the UAV's primary flight controller to change its current heading - if the warning is "Obstruction Detected" or "Possible Collision" or "Imminent Collision" - or stop autonomous flight altogether and wait (in loiter mode) for the human pilot to take manual flight control - if the warning is "Manual Override".

1.6: MAVLink Protocol

Because the secondary controller to process the ACS signals was not built, a functional communication channel was not required and thus not implemented within the capstone project timeline; it remains to be implemented in future extensions of the project. The original specifications for this communication channel intended on using the Micro Air Vehicle Link (MAVLink) protocol to allow the secondary controller to communicate with the UAV's primary Flight Controller (Raspberry Pi 3B+ w/ the Navio2 HAT running the ArduPilot stack).

MAVLink or Micro Air Vehicle Link is a protocol for communicating with small unmanned vehicle. It is designed as a header-only message marshaling library. MAVLink was first released in early 2009 by Lorenz Meier under LGPL license. It is used mostly for communication between a Ground Control Station (GCS) and Unmanned vehicles, and in the inter-communication of the subsystem of the vehicle. It can be used to transmit the orientation of the vehicle, its GNSS location and speed. It is a serial communication protocol with the following data packet structure [11]

Field name	Index (Bytes)	Purpose
Start-of-frame	0	Denotes the start of frame transmission (v2: 0xFD)
Payload-length	1	length of payload (n)
incompatibility flags	2	Flags that must be understood for MAVLink compatibility
compatibility flags	3	Flags that can be ignored if not understood
Packet sequence	4	Each component counts their send sequence. Allows for detection of packet loss.
System ID	5	Identification of the SENDING system. Allows to differentiate different systems on the same network.
Component ID	6	Identification of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
Message ID	7 to 9	Identification of the message - the id defines what the payload "means" and how it should be correctly decoded.
Payload	10 to (n+10)	The data into the message, depends on the message id.
CRC	(n+11) to (n+12)	Checksum of the entire packet, excluding the packet start sign (LSB to MSB)
Signature	(n+12) to (n+26)	Signature to verify that messages originate from a trusted source. (optional)

Table 6: MAVLink post v2 data packet structure

To ensure message integrity, a cyclic redundancy check (CRC) is calculated and appended to every message as two bytes. Another function of the CRC field is to ensure the sender and receiver both agree in the message that is being transferred. It is computed using an ITU X.25/SAE AS-4 hash of the bytes in the packet, excluding the Start-of-Frame indicator (so 6+n+1 bytes are evaluated, the extra +1 is the seed value). Additionally, a seed value is appended to the end of the data when computing the CRC. The seed is generated with every new message set of the protocol, and it is hashed in a similar way as the packets from each message specifications. Systems using the MAVLink protocol can use a precomputed array to this purpose.

2: Unmanned Aerial Vehicle - Carrier Platform (UAV-CP)

In order to collect data in the spatial paradigm, the RSA must be carried by a UAV. The UAV uses a 1000 class hexacopter frame as its chassis. It has six 5008 340KV brushless motors to provide thrust & two 3S1P 16Ah LiPo batteries connected in series to power them through six 40A ESC. The UAV CP uses a Raspberry Pi 3B+ with a Navio2 HAT running the ArduPilot stack as its flight controller. Autonomous & Telemetry communication uses the MAVLink protocol while a 2.4GHz 16-channel radio system is used for manual control.

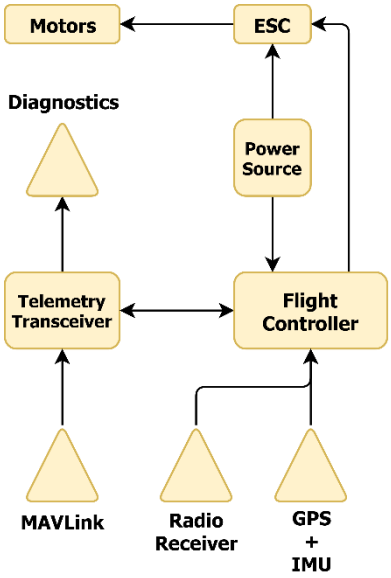


Figure 12: System level block diagram of the Unmanned Aerial Vehicle - Carrier Platform (UAV-CP) architecture

2.1: Chassis

For our purpose, we used Tarot-Frame-TL6X001 which has a diameter of 960mm (motor to motor). The Tarot X Series Airframe is designed to carry heavy gimbal and cameras (3-Axis Gimbal and DSLR Camera) for aerial cinematography. There are many features such as integrated PCB board for elegant cable routing, foldable arms for easy transportation (no tools required) and motorized landing gear. The frame is designed to have a low center of gravity. Battery is mounted on the center for a more stable flight. Tarot 3-Axis Brushless Motor Gimbal and special shape arms can hold the ESC inside for cleaner layout, even the Motor Mount has dampers to minimize vibration generated from the propeller.

Figure 13: Tarot X6 Frame (unfolded)

Figure 14: Tarot X6 Frame (folded)

Motor to Motor spacing	960mm
Propeller Standard	18"
Arms' Length	392mm
Arms' Diameter	25mm
Arms' Weight	113g
Main Frame Diameter	328mm
Ground Clearance	320mm (Rail to ground)
Battery Standard	22.2V(6S), 10000-20000mAh

Table 7: Tarot X6 specifications

2.2: Motor

A brushless DC electric motor (BLDC motor or BL motor), also known as electronically commutated motor (ECM or EC motor) and synchronous DC motors, are synchronous motors powered by direct current (DC) electricity via an inverter or switching power supply which produces electricity in the form of alternating current (AC) to drive each phase of the motor via a closed loop controller. The controller provides pulses of current to the motor windings that control the speed and torque of the motor. The construction of a brushless motor system is typically similar to a permanent magnet synchronous motor (PMSM), but can also be a switched reluctance motor, or an induction (asynchronous) motor. The advantages of a brushless motor over brushed motors are high power-to-weight ratio, high speed, electronic control, and low maintenance. Brushless motors find applications in such places as computer peripherals (disk drives, printers), hand-held power tools, and vehicles ranging from model aircraft to automobiles. For our purposes, Multistar Elite 5008 330KV was used.

Key advantages of the Multistar Elite 5008 330KV

- Motor Spindle: Multistar Elite motor adopts authentic Japan Kawasaki 20JNE1200 material. With this, the motor becomes more efficient and gets lower temperature during flight.
- Magnet: NdFeB N45UH high heat resistance magnet, with nickel plating on the surface.
- Varnished Wire: Multistar Elite Motor adopts 155-degree high heat resistance vanished wires.
- Bearings: Multistar Elite 5008 adopts authentic EZO bearing.
- Dynamic Balance: We ensure every Multistar Elite Motor has been processed by dynamic balance. And the balance quality can reach in 0.01 left/right. (This is the extremity of the balancing machine at present)

Figure 15: Multistar Elite 5008 330KV Motor

Motor Diameter	57.7mm
Motor Length	27.5mm
Stator Diameter	50mm
Stator Length	8mm
Shaft Diameter	4mm
Weight	185.5g
Batteries	6 cells
Top Mounting Holes	12mm*12mm
Bottom Mounting Holes	25mm*25mm

Table 8: Multistar Elite 5008 330KV Motor Specifications

Volts (V)	Prop	Amps (A)	Watts (W)	Thrust (G)	Thrust (oz)	Efficiency (G/W)	Efficiency (oz/W)
24V	CF1755	2.0	48.0	565	19.93	11.77	0.42
		4.0	96.0	960	33.86	10.00	0.35
		6.0	144.0	1260	44.44	8.75	0.31
		8.0	192.0	1540	54.32	8.02	0.28
		12.0	288.0	2035	71.78	7.07	0.25
		14.0	336.0	2280	80.42	6.79	0.24
		16.0	384.0	2530	89.24	6.59	0.23
		17.8	427.2	2730	96.30	6.39	0.23
	CF1865	2.0	48.0	590	20.81	12.29	0.43
		6.0	144.0	1320	46.56	9.17	0.32
		12.0	288.0	2070	73.02	7.19	0.25
		14.0	336.0	2300	81.13	6.85	0.24
		16.0	384.0	2500	88.18	6.51	0.23
		20.0	480.0	2820	99.47	5.88	0.21
		24.7	592.8	3300	116.40	5.57	0.20

Table 9: Multistar Elite 5008 330KV Motor Thrust Characteristics

2.3: Battery

A lithium polymer battery, or more correctly lithium-ion polymer battery (abbreviated as LiPo, LIP, Li-poly, lithium-poly and others), is a rechargeable battery of lithium-ion technology using a polymer electrolyte instead of a liquid electrolyte. High conductivity semisolid (gel) polymers form this electrolyte. These batteries provide higher specific energy than other lithium battery types and are used in applications where weight is a critical feature, like mobile devices and radio-controlled aircraft.

Venom Professional DRONE series LiPo batteries are engineered for the demanding requirements of multirotor, fixed wing and UAV applications, produced using the best LiPo materials and technology available. This Venom DRONE Professional series 16000mAh 3S 11.1V 15C LiPo battery with XT90-S plug was designed with power hungry multirotor, long range fixed wing, and sensor rich unmanned system applications in mind, providing unprecedented flight times and unmatched overall performance. It includes XT90-S Anti-Spark plug connector for easy connectivity. All DRONE high capacity batteries feature a removable balance wire, reducing the potential for damage to the balance lead, and eliminating unnecessary wire clutter. This Professional DRONE series batteries came packaged in a heavy duty; lockable, travel-ready composite case complete with a pressure equalization valve. Also includes three high quality battery straps and a handy LiPo monitor allowing for up to the minute cell voltage readings.

Figure 16: Venom 3S 16000mAh Battery

Battery Type	Lithium Polymer (LiPo Batteries)
C Rate	15C
Volts	11.1
Capacity	16000mAh
Cell Count	3S
Cell Configuration	3S1P
Continuous Discharge	15C(240A)
Maximum Burst Rate	30C(480A)
Maximum Volts per pack	12.6V
Minimum Volts per pack	9V
Charge Rate	1C(16A)
Wire Gauge	10 AWG Soft and Flexible Low Resistance Silicone Wire
Plug Type	XT90-S Anti Spark
Dimension	200 x 75.5 x 30 mm / 7.9 x 3 x 1.2 in
Watt Hours	177.6

Table 10: Venom 3S 16000mAh Battery Specifications

2.4: Electronic Speed Controller (ESC)

An electronic speed control follows a speed reference signal (derived from a throttle lever, joystick, or other manual input) and varies the switching rate of a network of field effect transistors (FETs) .[1] By adjusting the duty cycle or switching frequency of the transistors, the speed of the motor is changed. The rapid switching of the transistors is what causes the motor itself to emit its characteristic high-pitched whine, especially noticeable at lower speeds. The speed of the motor is varied by adjusting the timing of pulses of current delivered to the windings of the motor.

Brushless ESC systems basically create three-phase AC power, NOT like a VFD variable frequency drive, to run brushless motors. Brushless motors are ideal for radio-controlled aircrafts because of their efficiency, power, longevity and light weight in comparison to traditional brushed motors. On the downside, brushless DC motor controllers are much more complicated than brushed motor controllers.

The correct phase varies with the motor rotation, which is to be considered by the ESC: usually, back EMF from the motor is used to detect this rotation, but variations exist that use magnetic (Hall effect) or optical detectors. Computer-programmable speed controls generally have user-specified options which allow setting low voltage cut-off limits, timing, acceleration, braking and direction of rotation. Reversing the motor's direction may also be accomplished by switching any two of the three leads from the ESC to the motor.

For our project we used HobbyWing XRotor Pro 40A 3D ESC. The XRotor Pro series of ESCs from HobbyWing have been optimized for multi-rotor use. With their special core program for multi-rotor greatly improved throttle response and software specially engineered for compatibility with disc type out runner motors they offer unsurpassed smoothness and control. All settings except the timing are pre-set, making using the XRotor Pro series very simple and versatile.

Crosstalk from the extra-long signal wires is effectively reduced by twisting the wires instead of laying them parallel which again, helps to simplify installation and improve the flight characteristics of your multi-rotor. Most flight controllers are compatible with the XRotor Pro series which supports signal frequencies of up to 621Hz (anything over 500Hz is non-standard.). The high performance of the drive chip is enhanced further using extra low-resistance MOSFETs that also allow high currents to be held stable for longer.

A unique feature of the series is the Driving Efficiency Optimization (DEO) function. This significantly improves throttle linearity and driving efficiency by automatically braking and quickly reducing the motor speed when decreasing the throttle amount. This can improve the maneuverability and stability of your multi-rotor and improves the efficiency of the ESC as well as reducing its operating temperature.

Key features of the HobbyWing XRotor Pro 40A 3D ESC

- Super-efficient ESC
- Slim design for fitting inside tubular arms
- Driving Efficiency Optimization function
- Long leads for simple installation
- Programmable DEO function

Figure 17: HobbyWing XRotor Pro 40A 3D

Constant Current	40A
Burst Current	60A(10seconds)
BEC Mode	No BEC(opto-coupled)
LiPo Cells	3~6S(11.3~22.6V)
Power Lead Length	280mm
Signal Lead Length	370mm
Motor lead length	75mm
Motor Connectors	3.5mm bullet
Size	66*22*11mm
Weight	50g

Table 11: HobbyWing XRotor Pro 40A 3D Specifications

2.5: Flight Controller

A flight controller (FC) is a small circuit board of varying complexity. Its function is to direct the RPM of each motor in response to input. A command from the pilot for the multi-rotor to move forward is fed into the flight controller, which determines how to manipulate the motors accordingly. Most flight controllers also employ sensors to supplement their calculations. These range from simple gyroscopes for orientation to barometers for automatically holding altitudes. GNSS can also be used for auto-pilot or fail-safe purposes. More on that shortly. With a proper flight controller setup, a pilot's control inputs should correspond exactly to the behavior of the craft. Flight controllers are configurable and programmable, allowing for adjustments based on varying multi-rotor configurations. Gains or PIDs are used to tune the controller, yielding snappy, locked-in response. Depending on your choice of flight controller, various software is available to write your own settings.

2.5.1: Raspberry Pi 3B+

The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV and uses a standard keyboard and mouse. It's capable of doing everything a desktop computer can do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.

We used Raspberry Pi 3B+ in our project. Raspberry Pi 3 B+ is provides us with gigabit and PoE capable Ethernet, as well as better overheating protection for the 64-bit processor. Our particular model consists of Broadcom BCM2837, an ARM Cortex-A53 64-bit Quad Core Processor System-on-Chip operating at 1.4GHz. The GPU provides OpenGL ES 2.0, hardware accelerated OpenVG and 1080p30 H.264 high-profile decode. It is capable of 1Gpixel/s, 1.5Gtexel/s or 24 GFLOPs of general-purpose compute.

Key features of the Raspberry Pi 3B+

- Broadcom BCM2837B0 64-bit ARM Cortex-A53 Quad Core Processor SoC running @ 1.4GHz
- 1GB RAM LPDDR2 SDRAM
- 4x USB2.0 Ports with up to 1.2A output
- Extended 40-pin GPIO Header
- Video/Audio Out via 4-pole 3.5mm connector, HDMI, CSI camera, or Raw LCD (DSI)
- Storage: microSD
- Gigabit Ethernet over USB 2.0 (maximum throughput 300Mbps)
- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE
- H.264, MPEG-4 decode (1080p30); H.264 encode (1080p30); OpenGL ES 1.1, 2.0 graphics
- Low-Level Peripherals:
    - 27x GPIO
    - UART
    - I2C bus
    - SPI bus with two chip selects
    - Regulated 5V rail
    - Regulated 3.3V rail
- Power Requirements: 5V @ 2.5A via microUSB power source
- Supports Raspbian, Windows 10 IoT Core, OpenELEC, OSMC, Pidora, Arch Linux, RISC OS etc.
- 85mm x 56mm x 17mm

Figure 18: Raspberry Pi 3B+

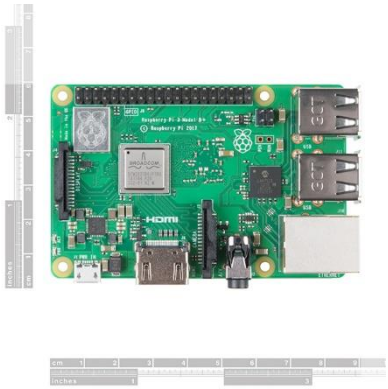


Figure 19: Raspberry Pi 3B+ with scale

2.5.2: Navio2

For our application, a Navio2 HAT is used to convert a Raspberry Pi 3B+ into the hexacopter flight controller. Using the Navio2 HAT in conjunction with the Raspberry Pi 3B+ allows writing powerful programs within the flight controller by utilizing the Linux environment of the Raspbian OS. The Raspbian OS is built around the Linux kernel and is capable of hosting C/C++ and python programs among many others, this feature alone opens many possibilities in terms of autonomous flight capabilities for future extensions of the project.

Figure 20: Navio2 Raspberry Pi HAT

Key features of the Navio2 Raspberry Pi HAT

- MPU9250 9DOF IMU
- LSM9DS1 9DOF IMU
- MS5611 Barometer
- U-blox M8N GNSS Module
- Cortex-M3 RC I/O co-processor
- RGB LED

Navio2 I/O Ports

- Power module
- UART
- I²C
- ADC
- 12 PWM servo outputs
- PPM/S.Bus input

Dimensions	55*65mm
Weight	23g
Operating Temperature	-40°C to +85°C

Table 12: Mechanical Specification

Supply Voltage	4.75-5.25V
Power Supply	Power Module, Servo rail, USB
Average Current Consumption	<150mA

Table 13: Electrical Specification

2.5.2.1: MPU9250 9DOF IMU

The MPU-9250, delivered in a 3x3x1mm QFN package, is the world's smallest and second generation 9-axis Motion Tracking device. The MPU-9250 is a System in Package (SiP) that combines two chips: the MPU-6500, which contains a 3-axis gyroscope, a 3-axis accelerometer, and an onboard Digital Motion Processor™ (DMP™) capable of processing complex Motion Fusion algorithms; and the AK8963, a 3-axis digital compass.

MPU-9250 features three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs, three 16-bit ADCs for digitizing the accelerometer outputs, and three 16-bit ADCs for digitizing the magnetometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±2000°/sec (dps), a user-programmable accelerometer full-scale range of ±2g, ±4g, ±8g, and ±16g, and a magnetometer full-scale range of ±4800μT.

The device features I2C and SPI serial interfaces, a VDD operating range of 2.4V to 3.6V, and a separate digital IO supply, VDDIO from 1.71V to VDD. Communication with all registers of the device is performed using either I2C at 400kHz or SPI at 1MHz. For applications requiring faster communications, the sensor and interrupt registers may be read using SPI at 20MHz.

The triple-axis MEMS gyroscope in the MPU-9250 includes a wide range of features:

- Digital-output X-, Y-, and Z-Axis angular rate sensors (gyroscopes) with a user-programmable full-scale range of ±250, ±500, ±1000, and ±2000°/sec and integrated 16-bit ADCs
- Digitally programmable low-pass filter
- Gyroscope operating current: 3.2mA
- Sleep mode current: 8μA
- Factory calibrated sensitivity scale factor
- Self-test

The triple-axis MEMS accelerometer in MPU-9250 includes a wide range of features:

- Digital-output triple-axis accelerometer with a programmable full-scale range of ±2g, ±4g, ±8g and
- ±16g and integrated 16-bit ADCs
- Accelerometer normal operating current: 450μA
- Low power accelerometer mode current: 8.4μA at 0.98Hz, 19.8μA at 31.25Hz
- Sleep mode current: 8μA
- User-programmable interrupts
- Wake-on-motion interrupt for low power operation of applications processor
- Self-test

The triple-axis MEMS magnetometer in MPU-9250 includes a wide range of features:

- 3-axis silicon monolithic Hall-effect magnetic sensor with magnetic concentrator
- Wide dynamic measurement range and high resolution with lower current consumption.
- Output data resolution of 14 bit (0.6μT/LSB)
- Full scale measurement range is ±4800μT
- Magnetometer normal operating current: 280μA at 8Hz repetition rate
- Self-test function with internal magnetic source to confirm magnetic sensor operation on end products

The MPU-9250 includes the following additional features:

- Auxiliary master I2C bus for reading data from external sensors (e.g. pressure sensor)
- 3.5mA operating current when all 9 motion sensing axes and the DMP are enabled
- VDD supply voltage range of 2.4 – 3.6V
- VDDIO reference voltage for auxiliary I2C devices
- Smallest and thinnest QFN package for portable devices: 3x3x1mm
- Minimal cross-axis sensitivity between the accelerometer, gyroscope and magnetometer axes
- 512-byte FIFO buffer enables the applications processor to read the data in bursts
- Digital-output temperature sensor
- User-programmable digital filters for gyroscope, accelerometer, and temp sensor
- 10,000 g shock tolerant
- 400kHz Fast Mode I2C for communicating with all registers
- 1MHz SPI serial interface for communicating with all registers
- 20MHz SPI serial interface for reading sensor and interrupt registers
- MEMS structure hermetically sealed and bonded at wafer level
- RoHS and Green compliant

Parameter	Conditions	Min	Typ	Max	Units
Full-Scale Range	FS_SEL=0		±250		°/s
	FS_SEL=1		±500		°/s
	FS_SEL=2		±1000		°/s
	FS_SEL=3		±2000		°/s
Gyroscope ADC Word Length			16		bits
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)
	FS_SEL=1		65.5		LSB/(°/s)
	FS_SEL=2		32.8		LSB/(°/s)
	FS_SEL=3		16.4		LSB/(°/s)
Sensitivity Scale Factor Tolerance	25°C		±3		%
Sensitivity Scale Factor Variation Over Temperature	-40°C to +85°C		±4		%
Nonlinearity	Best fit straight line; 25°C		±0.1		%
Cross-Axis Sensitivity			±2		%
Initial ZRO Tolerance	25°C		±5		°/s
ZRO Variation Over Temperature	-40°C to +85°C		±30		°/s
Total RMS Noise	DLPFCFG=2 (92 Hz)		0.1		°/s-rms
Rate Noise Spectral Density			0.01		°/s/√Hz
Gyroscope Mechanical Frequencies		25	27	29	kHz
Low Pass Filter Response	Programmable Range	5		250	Hz
Gyroscope Startup Time	From Sleep mode		35		ms
Output Data Rate	Programmable, Normal mode	4		8000	Hz

Table 14: Gyroscope Specifications

Parameter	Conditions	Min	Typ	Max	Units
Full Scale Range	AFS_SEL = 0		±2		g
	AFS_SEL = 1		±4		g
	AFS_SEL = 2		±8		g
	AFS_SEL = 3		±16		G
ADC Word length	Output in two's complement format		16		bits
Sensitivity Scale Factor	AFS_SEL = 0		16,384		LSB/g
	AFS_SEL = 1		8,192		LSB/g
	AFS_SEL = 2		4,096		LSB/g
	AFS_SEL = 3		2,048		LSB/g
Initial tolerance	Component level		±3		%
Sensitivity change vs. temperature	-40°C to +85°C AFS_SEL=0 Component-level		±0.026		%/°C
Nonlinearity	Best fit straight line		±0.5		%
Cross axis sensitivity			±2		%
Zero-G initial calibration Tolerance	Component level X, Y		±60		mg
	Component level Z		±80		mg
Zero-G level change vs Temperature	-40°C to +85°C		±1.5		mg/°C
Noise power spectral density	Low noise mode		300		µg/√Hz
Total RMS Noise	DLPFCFG=2 (94Hz)			8	mg-rms
Low Pass Filter Response	Programmable Range	5		260	Hz
Intelligence Function Increment			4		mg/LSB
Output Data Rate	Low power (duty-cycled)	0.24		500	Hz
	Duty-cycled, over temp		±15		%
	Low noise (active)	4		4000	Hz

Table 15: Accelerometer Specifications

Parameter	Conditions	Min	Typ	Max	Units
Magnetometer Sensitivity					
Full-Scale Range			±4800		µT
ADC Word Length			14		bits
Sensitivity Scale Factor			0.6		µT / LSB
Zero-Field Output					
Initial Calibration Tolerance			±500		LSB

Table 16: Magnetometer Specifications

2.5.2.2: LSM9DS1 9DOF IMU

The LSM9DS1 is a system-in-package featuring a 3D digital linear acceleration sensor, a 3D digital angular rate sensor, and a 3D digital magnetic sensor. The LSM9DS1 has a linear acceleration full scale of ±2g/±4g/±8/±16 g, a magnetic field full scale of ±4/±8/±12/±16 gauss and an angular rate of ±245/±500/±2000 dps. The LSM9DS1 includes an I2C serial bus interface supporting standard and fast mode (100 kHz and 400 kHz) and an SPI serial standard interface.

Key features of the LSM9DS1 9DOF IMU:

- 3 acceleration channels, 3 angular rate channels, 3 magnetic field channels
- • ±2/±4/±8/±16 g linear acceleration full scale
- • ±4/±8/±12/±16 gauss magnetic full scale
- • ±245/±500/±2000 dps angular rate full scale
- • 16-bit data output
- • SPI / I2C serial interfaces
- • Analog supply voltage 1.9 V to 3.6 V
- • “Always-on” eco power mode down to 1.9 mA
- • Programmable interrupt generators
- • Embedded temperature sensor

Symbol	Parameter	Test conditions	Min.	Typ.	Max.	Unit
LA_FS	Linear acceleration measurement range			±2		g
				±4		
				±8		
				±16		
M_FS	Magnetic measurement range			±4		gauss
				±8		
				±12		
				±16		
G_FS	Angular rate measurement range			±245		dps
				±500		
				±2000		
LA_So	Linear acceleration sensitivity	Linear acceleration FS = ±2 g		0.061		mg/LSB
		Linear acceleration FS = ±4 g		0.122		
		Linear acceleration FS = ±8 g		0.244		
		Linear acceleration FS = ±16 g		0.732		
M_GN	Magnetic sensitivity	Magnetic FS = ±4 gauss		0.14		mgauss/ LSB
		Magnetic FS = ±8 gauss		0.29		
		Magnetic FS = ±12 gauss		0.43		
		Magnetic FS = ±16 gauss		0.58		
G_So	Angular rate sensitivity	Angular rate FS = ±245 dps		8.75		mdps/ LSB
		Angular rate FS = ±500 dps		17.50		
		Angular rate FS = ±2000 dps		70		
LA_TyOff	Linear acceleration typical zero-g level offset accuracy	FS = ±8 g		±90		mg
M_TyOff	Zero-gauss level	FS = ±4 gauss		±1		gauss
G_TyOff	Angular rate typical zero-rate level	FS = ±2000 dps		±30		dps
M_DF	Magnetic disturbance field	Zero-gauss offset starts to degrade			50	gauss
T_{OP}	Operating temperature range		-40		+85	°C

Table 17: Sensor Characteristics

Symbol	Parameter	Test conditions	Min.	Typ.	Max.	Unit
V_{DD}	Supply voltage		1.9		3.6	V
V_{DD_IO}	Module power supply for I/O		1.71		V_{DD}+0.1	
I_{DD_XM}	Current consumption of the accelerometer and magnetic sensor in normal mode			600		µA
I_{DD_G}	Gyroscope current consumption in normal mode			4.0		mA
T_{OP}	Operating temperature range		-40		+85	°C
T_{RISE}	Time for power supply rising		0.01		100	ms
T_{WAIT}	Time delay between V_{DD_IO} and V_{DD}		0		10	ms

Table 18: Electrical Characteristics

Symbol	Parameter	Test condition	Min.	Typ.	Max.	Unit
T_{ODR}	Temperature refresh rate	Gyro OFF		50		Hz
		Gyro ON		59.5		
T_{SEN}	Temperature sensitivity			16		LSB/°C
T_{OP}	Operating temperature range		-40		+85	°C

Table 19: Temperature Sensor Characteristics

Symbol	Parameter	I²C Standard mode		I²C Fast mode		Unit
		Min	Max	Min	Max	
f(SCL)	SCL clock frequency	0	100	0	400	kHz
tw(SCLL)	SCL clock low time	4.7		1.3		µs
tw(SCLH)	SCL clock high time	4.0		0.6		
tsu(SDA)	SDA setup time	250		100		ns
th(SDA)	SDA data hold time	0	3.45	0	0.9	µs
th(ST)	START condition hold time	4		0.6		µs
tsu(SR)	Repeated START condition setup time	4.7		0.6		
tsu(SP)	STOP condition setup time	4		0.6		
tw(SP:SR)	Bus free time between STOP and START condition	4.7		1.3		

Table 20: I²C slave timing values

2.5.2.3: MS5611 Barometer

The MS5611 is a new generation of high-resolution altimeter sensors from MEAS Switzerland with SPI and I2C bus interface. This barometric pressure sensor is optimized for altimeters and barometers with an altitude resolution of 10 cm. The sensor module includes a high linearity pressure sensor and an ultra-low power 24-bit ΔΣ ADC with internal factory calibrated coefficients. It provides a precise digital 24 Bit pressure and temperature value and different operation modes that allow the user to optimize for conversion speed and current consumption.

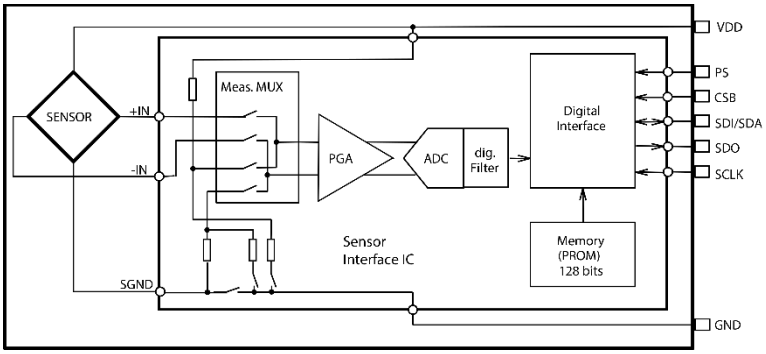


Figure 21: Internal block diagram

Parameter	Symbol	Conditions	Min.	Typ.	Max	Unit
Operating Supply voltage	V_{DD}		1.8	3.0	3.6	V
Operating Temperature	T		-40	+25	+85	°C
Supply current (1 sample per sec.)	I_{DD}	OSR 4096 2048 1024 512 256		12.5 6.3 3.2 1.7 0.9		µA
Peak supply current		during conversion		1.4		mA
Standby supply current		at 25°C		0.02	0.14	µA
VDD Capacitor		From VDD to GND	100			nF

Table 21: Electrical Characteristics

Parameter	Conditions		Min.	Typ.	Max	Unit
Operating Pressure Range	P_{RANGE}	Full Accuracy	450		1100	mbar
Extended Pressure Range	P_{EXT}	Linear Range of ADC	10		1200	mbar
Total Error Band, no autozero	at 25°C, 700.1100 mbar		-1.5		+1.5	mbar
	at 0.50°C, 450.1100 mbar		-2.0		+2.0	
	at -20.85°C, 450.1100 mbar		-3.5		+3.5	
	at -40.85°C, 450.1100 mbar		-6.0		+6.0	
Total Error Band, autozero at one pressure point	at 25°C, 700.1100 mbar		-0.5		+0.5	mbar
	at 10.50°C, 450.1100 mbar		-1.0		+1.0	
	at -20.85°C, 450.1100 mbar		-2.5		+2.5	
	at -40.85°C, 450.1100 mbar		-5.0		+5.0	
Maximum error with supply voltage	V_{DD}=1.8V…3.6V			±2.5		mbar
Long-term stability				±1		mbar/yr
Recovering time after reflowing (1)				7		days
Resolution RMS	OSR	4096 2048 1024 512 256		0.012 0.018 0.027 0.042 0.065		mbar

Table 22: Pressure Output Characteristics (VDD = 3 V, T = 25°C unless otherwise noted)

2.5.2.4: U-blox M8N GNSS Module

The NEO-M8 series of concurrent GNSS modules is built on the high performing u-blox M8 GNSS engine in the industry proven NEO form factor. The NEO-M8 series utilizes concurrent reception of up to three GNSS systems (GPS/Galileo together with BeiDou or GLONASS), recognizes multiple constellations simultaneously and provides outstanding positioning accuracy in scenarios where urban canyon or weak signals are involved. For even better and faster positioning improvement, the NEO-M8 series supports augmentation of QZSS, GAGAN and IMES together with WAAS, EGNOS, and MSAS. The NEO-M8 series also supports message integrity protection, geofencing, and spoofing detection with configurable interface settings to easily fit to customer applications. The NEO form factor allows easy migration from previous NEO generations. The NEO-M8M is optimized for cost sensitive applications, while NEO-M8N/M8Q provides best performance and easier RF integration. The NEO-M8N offers high performance also at low power consumption levels.

Parameter	**Specification**					
Receiver type	72-channel u-blox M8 engine GPS L1C/A, SBAS L1C/A, QZSS L1C/A, QZSS L1 SAIF, GLONASS L1OF, BeiDou B1I, Galileo E1B/C					
Accuracy of time pulse signal	RMS 99%	30 ns 60 ns				
Frequency of time pulse signal		0.25 Hz…10 MHz (configurable)				
Operational limits	Dynamics	≤ 4 g				
	Altitude	50,000 m				
	Velocity	500 m/s				
Velocity accuracy		0.05 m/s				
Heading accuracy		0.3 degrees				
GNSS		**GPS & GLONASS**	**GPS**	**GLONASS**	**BeiDou**	**Galileo**
Horizontal position accuracy		2.5 m	2.5 m	4 m	3 m	TBC
Max navigation update rate	NEO-M8N	5 Hz	10 Hz	10 Hz	10 Hz	10 Hz
	NEO-M8Q	10 Hz	18 Hz	18 Hz	18 Hz	18 Hz
Time-To-First Fix	Cold start	26 s	29 s	30 s	34 s	45 s
	Hot start	1 s	1 s	1 s	1 s	1 s
	Aided starts	2 s	2 s	2 s	3 s	7 s
Sensitivity	Tracking & Navigation	–167 dBm	–166 dBm	-166 dBm	-160 dBm	-159 dBm
	Reacquisition	–160 dBm	–160 dBm	-156 dBm	-157 dBm	-153 dBm
	Cold start	–148 dBm	–148 dBm	-145 dBm	-143 dBm	-138 dBm
	Hot start	–157 dBm	–157 dBm	-156 dBm	-155 dBm	-151 dBm

Table 23: Specification & Performance

2.6: Ardupilot

ArduPilot is an open source, unmanned vehicle Autopilot Software Suite, capable of controlling autonomous –

- Multirotor drones
- Fixed-wing and VTOL aircraft
- Helicopters
- Ground rovers
- Boats
- Submarines
- Antenna trackers

ArduPilot was developed by hobbyists to control model aircraft and rovers and has evolved into a full-featured and reliable autopilot used by industry, research organizations. ArduCopter is an advanced open-source autopilot system for multi-copters, helicopters, and other rotor vehicles. It offers a wide variety of flight modes from fully manual to fully autonomous.

Key features of ArduCopter:

- High precision acrobatic mode: perform aggressive maneuvers including flips!
- Auto-level and Altitude Hold modes: Fly level and straight with ease or add simple mode which removes the need for the pilot to keep track of the vehicle's heading. We must push the stick the way we want the vehicle to go, and the autopilot figures out what that means for whatever orientation the copter is in.
- Loiter and Position Hold modes: the vehicle will hold its position using its GNSS, accelerometer and barometer.
- Return to launch: Flip a switch to have Copter fly back to the launch location and land automatically.
- Ad-hoc commands in Flight: With a two-way telemetry radio installed, just click on the map and the vehicle will fly to the desired location.
- Autonomous missions: Use the ground station to define complex missions with up to hundreds of GPS waypoints. Then switch the vehicle to "AUTO" and watch it take-off, execute the mission, then return home, land and disarm all without any human intervention.
- Failsafe: The software monitors the state of the system and triggers an autonomous return-to-home in case of loss of contact with the pilot, low battery or the vehicle strays outside a defined geofence.
- Flexible and customizable: Copter can fly all shapes and sizes of vehicles just how we want it to because the user has access to hundreds of parameters that control its behavior.
- No vendor lock-in: ArduPilot is fully open source with a diverse community of developers behind it.

2.7: MAVLink Protocol

Refer to section 1.6 for details

2.9: Radio Control System

Radio control (often abbreviated to R/C or simply RC) is the use of control signals transmitted by radio to remotely control a device. Examples of simple radio control systems are garage door openers and keyless entry systems for vehicles, in which a small handheld radio transmitter unlocks or opens doors. A rapidly growing application is control of unmanned aerial vehicles (UAVs or drones) for both civilian and military uses, although these have more sophisticated control systems than traditional applications. In our application, the UAV is manually controlled by the FrSky 2.4GHz ACCST Taranis X9D Plus.

Figure 22: Taranis X9D Plus

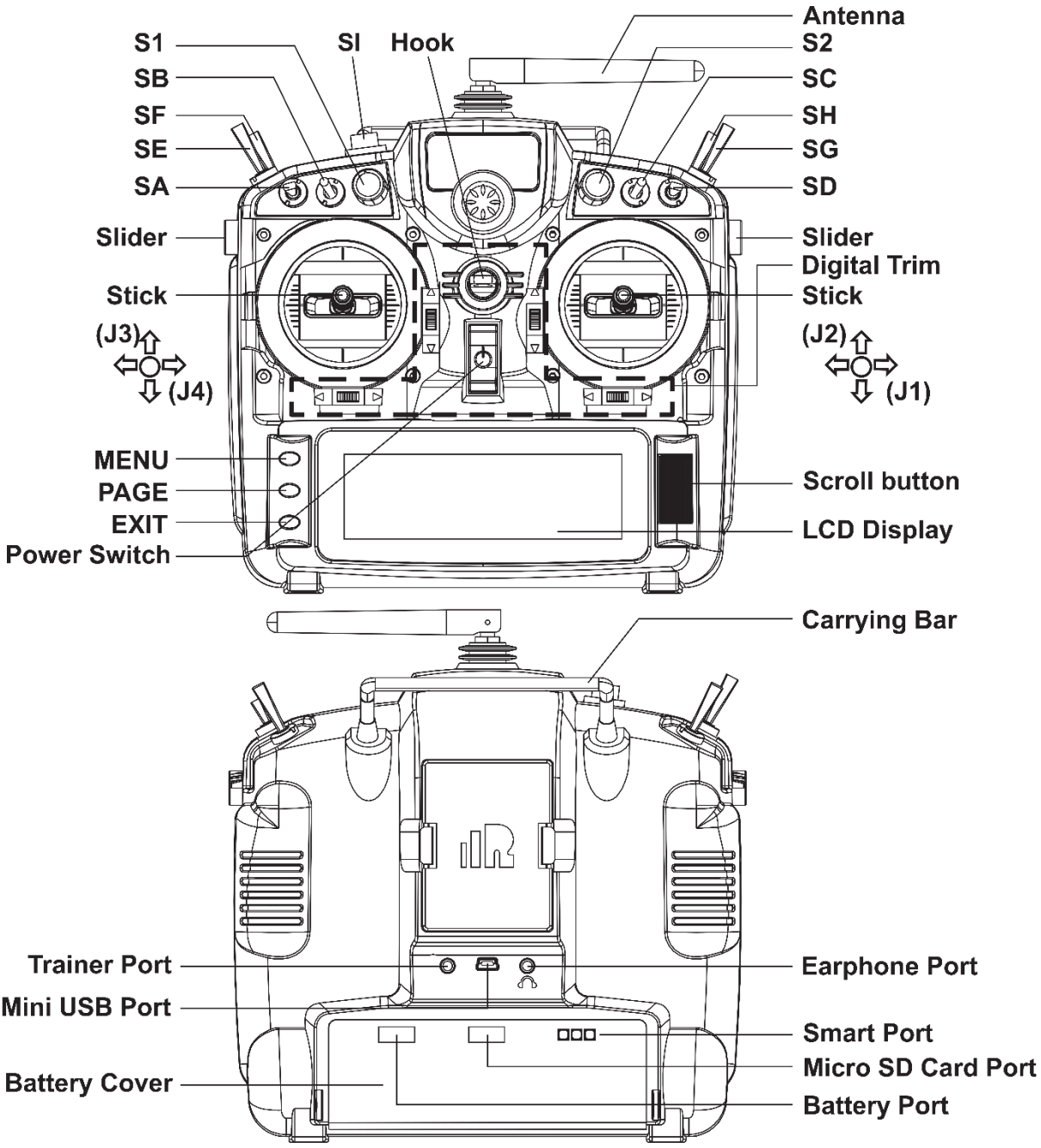


Figure 23: Taranis X9D Plus (labeled)

Key features of the Taranis X9D Plus:

- Classic Taranis form factor design
- High-speed module digital interface
- Easy Launch Momentary Button
- Installed with ACCESS protocol
- Supports spectrum analyzer function
- SWR indicator
- Haptic vibration alerts and voice speech outputs

Model Name	Taranis X9D plus 2019
Dimension	200*194*110
Weight	670g (without battery)
Operating current	130mA@8.2V(typ)
Backlight LCD Resolution	212*64
Model Memories	60models (extendable by Micros SD Cards)
Number of Channels	24 channels
Operating Voltage Range	DC 6.5~8.4V
Operating Temperature	-10℃~60℃ (14°F~140°F)

Table 24: Specifications

3: Power Management System (PMS)

The PMS is a polling architecture based closed-loop feedback controller to monitor the voltage level of the RSA's power source. It probes the INA219 at a frequency of ~1kHz to determine the voltage & communicates the state of the RSA & instructions through a pair of digital Rx/Tx lines. But the observations are contaminated with noise which interferes with the comparison operations in the ATMega4809 micro-controller. To attenuate the noise while still maintaining a reasonably fast response on the output, a combination of Linear Kalman Filter & Complementary Filter is used. The PMS also models a finite-state machine with 4 distinct states - Normal mode, Warning mode, Critical mode & Hibernation mode. Depending on a number of triggers like Battery-Voltage, Power-Button-Press & RSA-Power-State, the PMS shifts from one state to another.

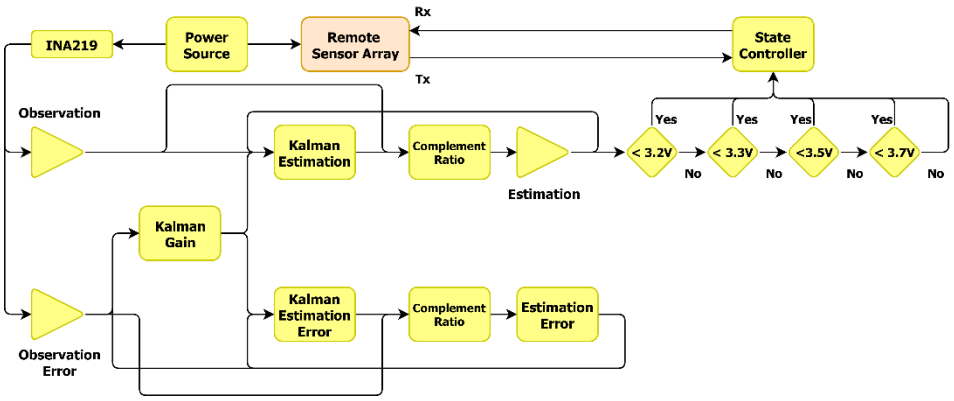


Figure 24: System level block diagram of the Power Management System (PMS) architecture

3.1: Polling Architecture

Polling is the process where the computer or controlling device waits for an external device to check for its readiness or state, often with low-level hardware. For example, when a INA219 Bi-directional Current sensing IC is connected via I2C protocol, the computer waits until the sensor has received the data packet. This is sometimes used synonymously with busy-wait polling. In this situation, when an I/O operation is required, the computer does nothing other than check the status of the I/O device until it is ready, at which point the device is accessed. In other words, the computer waits until the device is ready. Polling also refers to the situation where a device is repeatedly checked for readiness, and if it is not, the computer returns to a different task. Although not as wasteful of CPU cycles as busy waiting, this is generally not as efficient as the alternative to polling, interrupt driven I/O.

In a simple single-purpose system, even busy-wait is perfectly appropriate if no action is possible until the I/O access, but more often than not this was traditionally a consequence of simple hardware or non-multitasking operating systems.

Polling is often intimately involved with very low-level hardware. For example, polling a INA219 sensor port to check whether it is ready for another packet involves examining as little as one bit of a byte. That bit represents, at the time of reading, whether a single wire in the sensor wire is at low or high voltage. Polling has the disadvantage that if there are too many devices to check, the time required to poll them can exceed the time available to service the I/O device.

3.2 INA219

The INA219 is a current shunt and power monitor with an I2C- or SMBUS-compatible interface. The device monitors both shunt voltage drop and bus supply voltage, with programmable conversion times and filtering. A programmable calibration value, combined with an internal multiplier, enables direct readouts of current in amperes. An additional multiplying register calculates power in watts. The I2C- or SMBUS-compatible interface features 16 programmable addresses.

The INA219 is available in two grades: A and B. The B grade version has higher accuracy and higher precision specifications. The INA219 senses across shunts on buses that can vary from 0 to 26 V. The device uses a single 3- to 5.5-V supply, drawing a maximum of 1 mA of supply current. The INA219 operates from –40°C to 125°C.

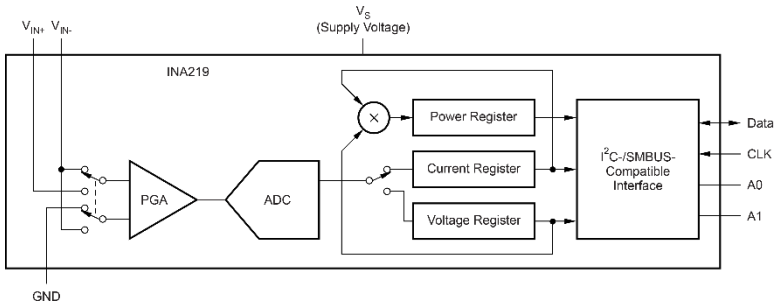


Figure 25: Simplified Schematic Diagram of INA219

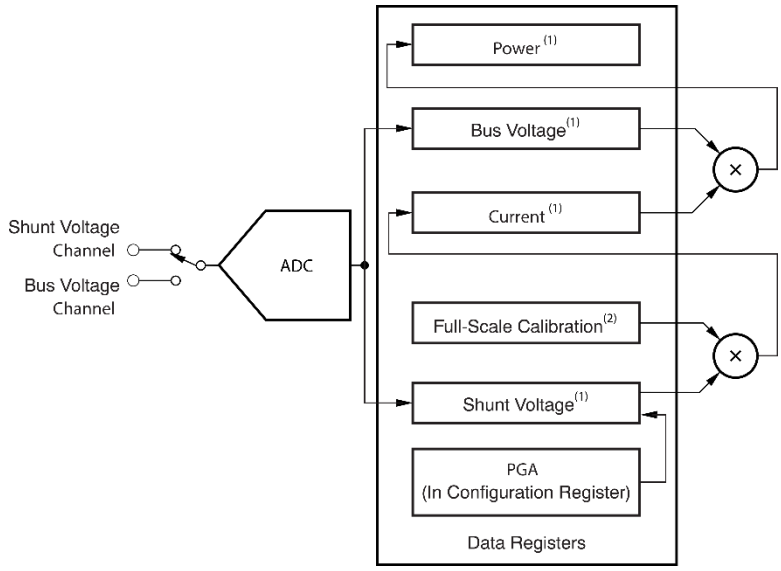


Figure 26: Functional Block Diagram of INA219

Parameter		Min	Max	Unit
V_S Supply voltage			6	V
Analog Inputs	Differential ($V_{IN+} - V_{IN-}$)	–26	26	V
IN_+, IN_-	Common-mode ($V_{IN+} + V_{IN-}$) / 2	-0.3	26	V
SDA		GND – 0.3	6	V
SCL		GND – 0.3	V_S + 0.3	V
Input current into any pin			5	mA
Open-drain digital output current			10	mA
Operating temperature		–40	125	°C
T_J Junction temperature			150	°C
T_{STG} Storage temperature		–65	150	°C

Table 25: Absolute Maximum Rating

Parameter	Min	Nom	Max	Unit
V_{CM}		12		V
V_S		3.3		V
T_A	–25		85	ºC

Table 26: Recommended Operating Conditions

Thermal Metric	D (SOIC) 8 PINS	DCN (SOT) 8 PINS	Unit
$R_{\theta JA}$ Junction-to-ambient thermal resistance	111.3	135.4	°C/W
$R_{\theta JC(TOP)}$ Junction-to-case (top) thermal resistance	55.9	68.1	°C/W
$R_{\theta JB}$ Junction-to-board thermal resistance	52	48.9	°C/W
ψ_{JT} Junction-to-top characterization parameter	10.7	9.9	°C/W
ψ_{JB} Junction-to-board characterization parameter	51.5	48.4	°C/W
$R_{\theta JC(BOT)}$ Junction-to-case (bottom) thermal resistance	N/A	N/A	°C/W

Table 27: Thermal Information

Parameter		Test Conditions	INA219A			INA219B			Unit
			Min	Typ	Max	Min	Typ	Max	
Input									
V_{SHUNT} Full-scale current sense (input) voltage range		PGA = /1	0		±40	0		±40	mV
		PGA = /2	0		±80	0		±80	mV
		PGA = /4	0		±160	0		±160	mV
		PGA = /8	0		±320	0		±320	mV
Bus voltage (input voltage) range		BRNG = 1	0		32	0		32	mV
		BRNG = 0	0		16	0		16	mV
CMRR Common-mode rejection		V_{IN+} = 0 to 26 V	100	120		100	120		dB
V_{OS}	Offset voltage, RTI vs Temperature	PGA = /1		±10	±100		±10		µV
		PGA = /2		±20	±125		±20		µV
		PGA = /4		±30	±150		±30		µV
		PGA = /8		±40	±200		±40		µV
		T_A = –25°C to 85°C		0.1			0.1		µV/°C
PSRR vs Power Supply		V_S = 3 to 5.5 V		10			10		µV/V
Current sense gain error vs Temperature				±40			±40		m%
		T_A = –25°C to 85°C		1			1		m%/°C
IN_+ pin input bias current		Active mode		20			20		µA
IN_- pin input bias current || V_{IN}– pin input impedance		Active mode		20 || 320			20 || 320		µA || impedance kΩ
IN_+ pin input leakage		Power-down mode		0.1	±0.5		0.1	±0.5	µA
IN_- pin input leakage		Power-down mode		0.1	±0.5		0.1	±0.5	µA
DC Accuracy									
ADC basic resolution				12			12		bits
Shunt voltage, 1 LSB step size				10			10		µV
Bus voltage, 1 LSB step size				4			4		mV
Current measurement error over Temperature				±0.2%	±0.5%		±0.2%	±0.3%	
		T_A = –25°C to 85°C			±1%			±0.5%	
Bus voltage measurement error over Temperature				±0.2%	±0.5%		±0.2%	±0.5%	
		T_A = –25°C to 85°C			±1%			±1%	
Differential nonlinearity				±0.1			±0.1		LSB
ADC Timing									
ADC conversion time		12-bit		532	586		532	586	µs
		11-bit		276	304		276	304	µs
		10-bit		148	163		148	163	µs
		9-bit		84	93		84	93	µs
Minimum convert input low time			4			4			µs
SMBus									
SMBus timeout				28	35		28	35	ms

Digital Inputs (SDA as Input, SCL, A0, A1)								
Input capacitance			3			3		pF
Leakage input current	$0 \leq V_{IN} \leq V_S$		0.1	1		0.1	1	µA
V_{IH} input logic level		0.7 (V_S)		6	0.7 (V_S)		6	V
V_{IL} input logic level		−0.3		0.3 (VS)	−0.3		0.3 (VS)	V
Hysteresis			500			500		mV
Open-Drain Digital Outputs (SDA)								
Logic 0 output level	$I_{SINK} = 3$ mA		0.15	0.4		0.15	0.4	V
High-level output leakage current	$V_{OUT} = V_S$		0.1	1		0.1	1	µA
Power Supply								
Operating supply range		3		5.5	3		5.5	V
Quiescent current			0.7	1		0.7	1	mA
Quiescent current, power-down mode			6	15		6	15	µA
Power-on reset threshold			2			2		V

Table 28: Electrical Characteristics

3.3: Arduino Nano Every

The Nano Every is Arduino's 5V compatible board in the smallest available form factor: 45x18mm! The Arduino Nano is the preferred board for many projects requiring a small and easy to use microcontroller board. The small footprint and low price make the Nano Every particularly suited for wearable inventions, low cost robotics, electronic musical instruments, and general use to control smaller parts of a larger projects. The Arduino Nano Every is an evolution of the traditional Arduino Nano board, but features a lot more powerful processor, the ATMega4089. This will allow you to make larger programs than with the Arduino Uno (it has 50% more program memory), and with a lot more variables (the RAM is 200% bigger).

Microcontroller	ATMega4809
Operating Voltage	5V
Input Voltage (limit)	21V
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Clock Speed	20MHz
CPU Flash Memory	48KB (ATMega4809)
SRAM	6KB (ATMega4809)
EEPROM	256byte (ATMega4809)
PWM Pins	5 (D3, D5, D6, D9, D10)
UART	1
SPI	1
I2C	1
Analog Input Pins	8 (ADC 10 bit)
Analog Output Pins	Only through PWM (no DAC)
External Interrupts	all digital pins
LED_BUILTIN	13
USB	Uses the ATSAMD11D14A
Length	45 mm
Width	18 mm
Weight	5 grams (with headers)

Table 29: ATMega4809 microcontroller Specifications

3.4: Finite state Machine

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs and/or a condition is satisfied; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition. Finite state machines are of two types – deterministic finite state machines and non-deterministic finite state machines. A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

The behavior of state machines can be observed in many devices in modern society that perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines, which dispense products when the proper combination of coins is deposited, elevators, whose sequence of stops is determined by the floors requested by riders, traffic lights, which change sequence when cars are waiting, and combination locks, which require the input of a sequence of numbers in the proper order.

The PMS models a finite-state machine with 4 distinct states - Normal mode, Caution Mode, Warning mode, Critical mode & Hibernation mode. Depending on a number of triggers like Battery-Voltage, Power-Button-Press & RSA-Power-State, the PMS shifts from one state to another. The four modes are also divided in active and passive modes.

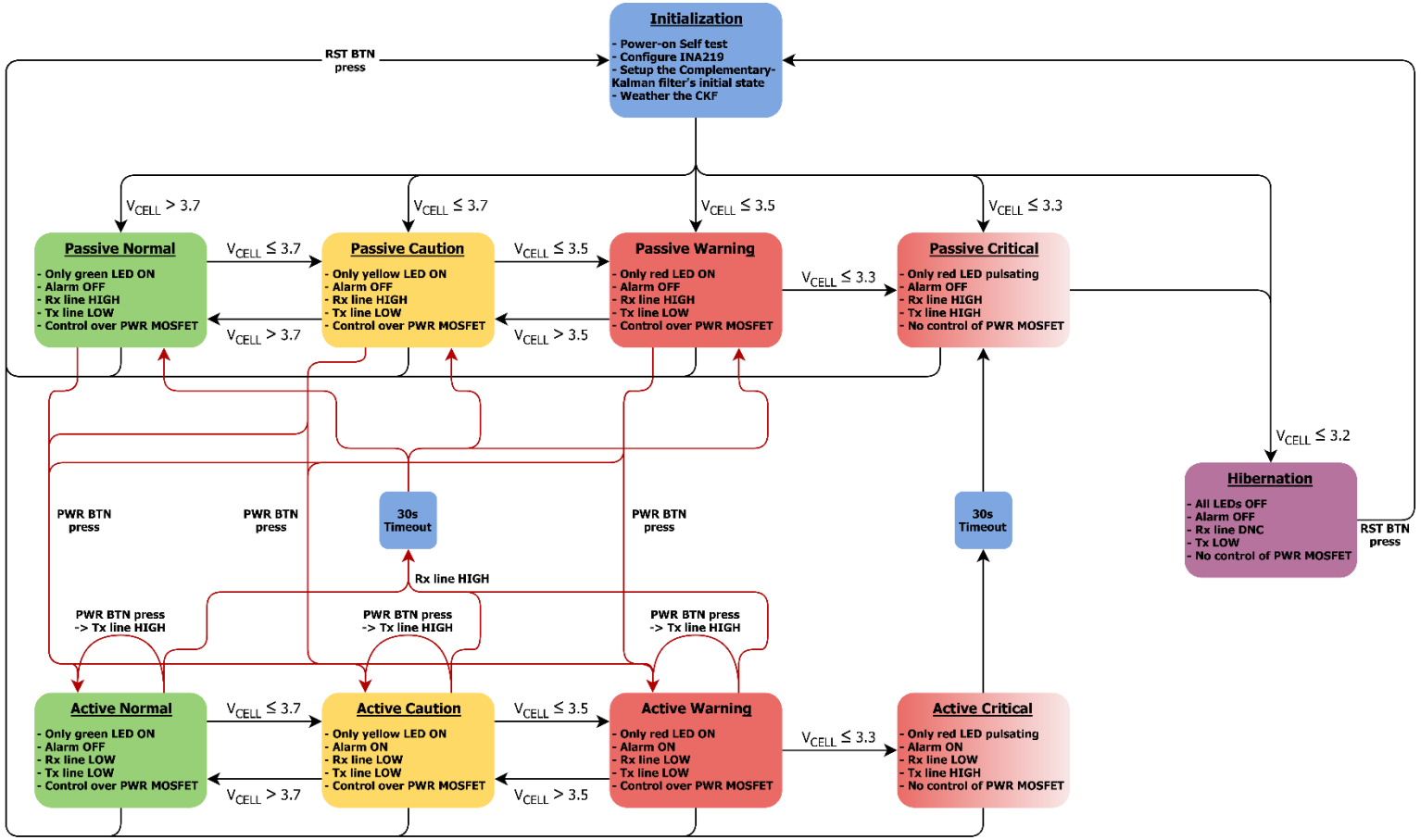


Figure 27: PMS state transition diagram

At power-on-reset the PMS Initialization process takes place. At the Initialization stage The PMS initializes POST (Power On Self-Test) test which is a diagnostic test to determine if the hardware is working properly or not. At this stage the INA219 is configured and the Complementary Kalman Filter's (CKF) initial state is setup and weather. After that the Initial stage transitions to any of the other four passive stages or Hibernation stage depending upon the per cell voltage of the LiPo battery of the RSA.

If the per cell voltage is more than 3.7V the PMS goes to Passive Normal Mode. At this stage the RSA is off. As the per cell voltage is more than the threshold voltage of 3.2V per cell & the RSA is also not powered on the alarm does not go on .At this stage only the Green LED is on, the RX line is high & TX line is low and the user has full control over the power MOSFET. Now if the voltage falls below 3.7V the Passive Caution Mode gets initiated.

The Passive Caution Mode can also be initiated after the initialization stage if the voltage per cell falls below 3.5V. At this stage only the Yellow LED is turned on, but the alarm is off because at the passive condition the RSA is not powered. Here the RX is high, and Tx is low & the user has full control over the power MOSFET. Like before if the voltage drops from that stage's required voltage to voltage below 3.5V the PMS goes to Passive Warning Mode.

After the initialization state the PMS can move to Passive Warning Mode if the per cell voltage is below 3.5V Similar to the previous stages here the RED LED is turned on but no alarm goes on due to the fact that the RSA is already off. Here the RX is high, and Tx is low & the user has full control over the power MOSFET. Now if the per cell stage falls below 3.3V per cell then the PMS moves to Passive Critical stage.

Similarly, just after initialization the PMS can move to Passive Critical Mode if the per cell voltage falls below 3.3V. At this mode the RED LED starts to blink but does turns the alarm due to unpowered RSA. But at this stage the user loses the control over the power MOSFET and both the RX & Tx line become high goes to Hibernation Mode

At the Hibernation mode the all the LEDs are turned off, no alarm goes on, RX becomes do not care & TX line is high, and the user loses all the control over the power MOSFET. Now if the Reset button of the Arduino Nano Every is pressed then the PMS again goes to Initialization stage.

When the power button is pressed & the PMS is at either Passive Normal or Passive Caution or Passive Warning mode PMS moves to an appropriate active state as determined by the cell voltage.

During the Active Normal Mode only the green LED is on as a result there is no warning alarm at this stage both RX & TX is low, and the user has full control over the power MOSFET. Now if the cell voltage falls below 3.7V in the active mode the PMS moves to Active Caution Mode where the Yellow LED turns on. With the initiation of Yellow LED, the caution warning sets in to warn the user at this mode the both RX & TX line are low, and the user has full control over the power MOSFET. If the cell voltage even falls below 3.5V the PMS goes to active Warning mode where the RED LED is turned on and the Alarm becomes more frequent. Like the previous states the RX and TX line are low, and the user has full control over the power MOSFET. After that when the cell voltage becomes below 3.3V the PMS goes to Active Critical Mode where the RED starts pulsating, and the alarm becomes even more frequent than that of any other previous modes. At this stage the RX line becomes low and the TX line becomes high and the user loses all the control over the power MOSFET. From this mode the PMS goes to Passive Critical Mode with a 30 second timeout interval After that the PMS goes to a Hibernation Mode from the Passive Critical Mode

Just like the Hibernation Mode any of the Active or Passive Modes of the PMS can be taken to initialization stage just by triggering the Reset switch of the Arduino Nano Every.

3.5: Complementary Kalman Filter

The PMS uses an Arduino Nano Every as its operational core. The Arduino Nano Every itself is powered by an ATMega4809 microcontroller. The comparators inside the ATMega4809 are very prone to noise as a result they produce jitter. Thus, the PMS requires a robust filtering system. Here a 1-D Kalman filter is used to solve the problem.

In statistics and control theory, Kalman filter a.k.a. Linear Quadratic Estimation (LQU) is an algorithm which uses several measurements over a period that contains statistical noise and inaccuracies to produce estimated measurements of that variable which tend to be more accurate than that of a single measurement alone. This algorithm was first developed by Hungarian born American electrical engineer, mathematician & inventor Rudolf Emil Kalman. According to the type of data the filter is designed to work with, the Kalman Filter can be divided into two categories: Linear Kalman Filter &. Non-Linear Kalman Filter.

The Linear Kalman Filter is an algorithm that is designed to filter noise from linear data sources. Linear Kalman Filters can be designed to work with a single dimension of data or 1-D Kalman Filter. The 1-D Kalman filter schematic is given below

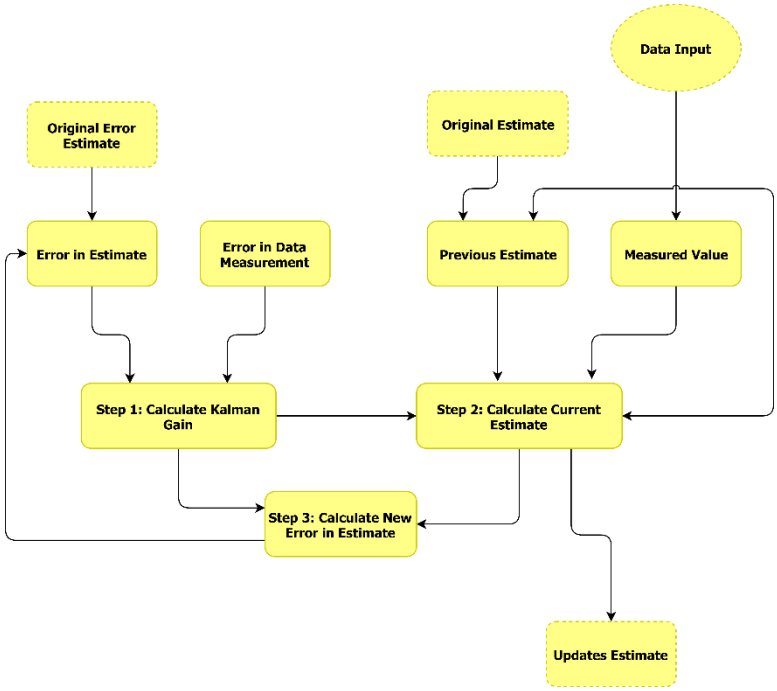


Figure 28: 1-D Kalman Filter Schematics

But by using the 1-D Kalman filter another problem arises. Due to the use of 1-D Kalman filter the slew rate of the entire system decreases. So, to remove the issue a Complementary filter is used. A complementary filter provides a way to combine two data sources by using their weighted average.

4: Remote Sensor Array (RSA)

The Remote Sensor Array is a suite of sensors that provide data collection capabilities to the EnvironMetric system. It is based on the single-producer-multi-consumer architecture & implements the Asynchronous Remote Acquisition protocol which allows it to serve multiple clients concurrently. The RSA has a dedicated radio communication system established by a mesh network composed of XBee modems. The RSA can operate in both a temporal and spatial paradigm, the latter with the aid of a (UAV) carrier platform. The RSA utilizes an AHRS algorithm to accurately keep track of its position. The RSA uses a Raspberry Pi 4B 4GB model as its compute platform. The various sensors connect with the Raspberry Pi through the PolyBus interface which allows a single I2C master controller to emulate multiple instances of itself. This allows a single master controller to interface with multiple instances of the same sensor. This helps to increase data refresh rates & reduce measurement noise.

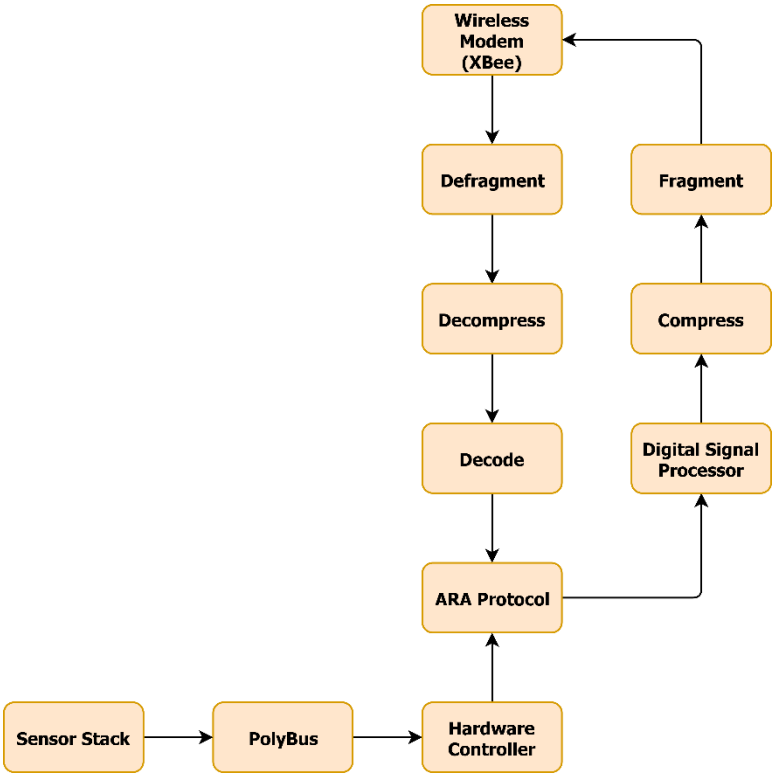


Figure 29: System level block diagram of the Remote Sensor Array (RSA) architecture

4.1: Sensor Stack

The sensor stack, in its current state, operates with low cost consumer-grade sensors made for educational purposes. While these sensors are of high quality and offer reasonable accuracy and precision, EnvironMetric's specifications for the stack is designed to accommodate professional-grade sensors. This will allow incorporating them into the model with minimal modifications to the source code and almost no modifications to the overall model design.

4.1.1: BME280

Figure 30: Bosch Sensortec BME280 SparkFun breakout board

The BME280 is as combined digital humidity, pressure and temperature sensor based on proven sensing principles. The sensor module is housed in an extremely compact metal-lid LGA package with a footprint of only 2.5 × 2.5 mm^2 with a height of 0.93 mm. Its small dimensions and its low power consumption allow the implementation in battery driven devices such as handsets, GPS modules or watches. The BME280 is register and performance compatible to the Bosch Sensortec BMP280 digital pressure sensor (see chapter 5.2 for details).

The BME280 achieves high performance in all applications requiring humidity and pressure measurement. These emerging applications of home automation control, in-door navigation, health care as well as GPS refinement require a high accuracy and a low TCO at the same time.

The humidity sensor provides an extremely fast response time for fast context awareness applications and high overall accuracy over a wide temperature range. The pressure sensor is an absolute barometric pressure sensor with extremely high accuracy and resolution and drastically lower noise than the Bosch Sensortec BMP180. The integrated temperature sensor has been optimized for lowest noise and highest resolution. Its output is used for temperature compensation of the pressure and humidity sensors and can also be used for estimation of the ambient temperature. In order to tailor data rate, noise, response time and current consumption to the needs of the application, a variety of oversampling modes, filter modes and data rates can be selected.

The sensor provides both SPI and I^2C interfaces and can be supplied using 1.71 to 3.6 V for the sensor supply VDD and 1.2 to 3.6 V for the interface supply VDDIO. Measurements can be triggered by the host or performed in regular intervals. When the sensor is disabled, current consumption drops to 0.1 µA.

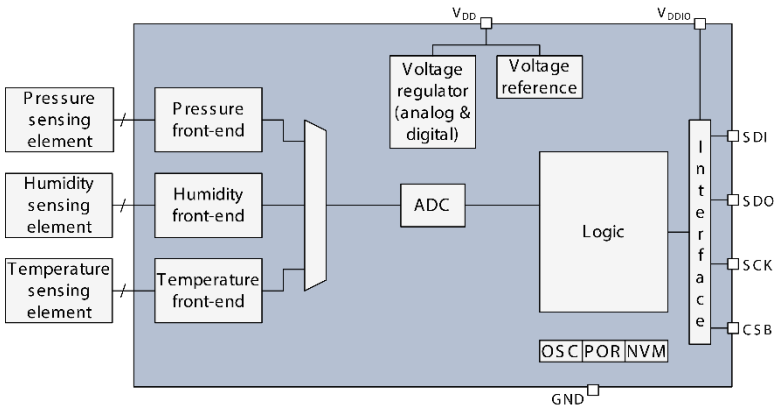


Figure 31: Functional block diagram of BME280

BME280 can be operated in three power modes

- **Sleep mode**: Default mode after power on reset. No measurements are performed and power consumption (IDDSM) is at a minimum. All registers are accessible.
- **Normal mode**: Comprised of an automated perpetual cycle between an (active) measurement period and an (inactive) standby period. The measurements are performed in accordance to the selected measurement and filter options. The total cycle time depends on the sum of the active time and standby time.
- **Forced mode**: A single measurement is performed in accordance to the selected measurement and filter options. Afterwards, the sensor returns to sleep mode and the measurement results can be obtained from the data registers.

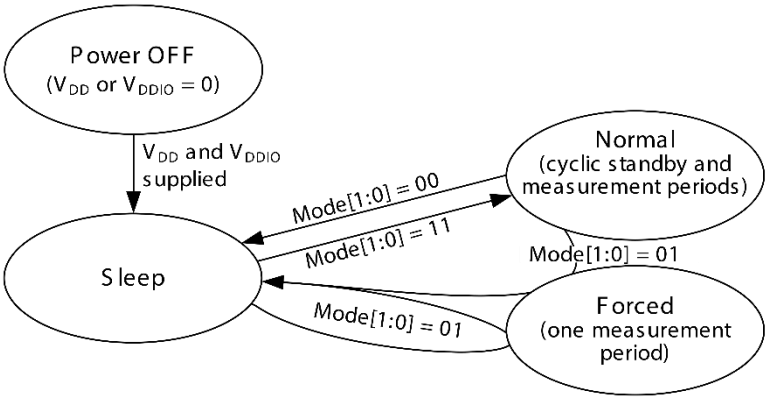


Figure 32: Sensor mode state transition diagram

The BME280 measurement period consists of a temperature, pressure and humidity measurement with selectable oversampling. After the measurement period, the pressure and temperature data can be passed through an optional IIR filter, which removes short-term fluctuations in pressure (e.g. caused by slamming a door). For humidity, such a filter is not needed and has not been implemented. The flow is depicted in the diagram below.

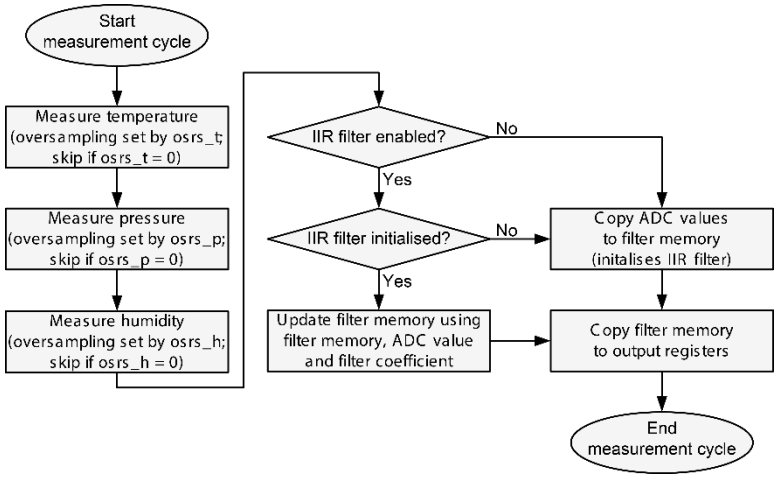


Figure 33: BME280 measurement cycle

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Supply Voltage (Internal Domains)	V_{DD}	ripple max. 50 mV_{PP}	1.71	1.8	3.6	V
Supply Voltage (I/O Domain)	V_{DDIO}		1.2	1.8	3.6	V
Sleep current	I_{DDSL}			0.1	0.3	µA
Standby current (inactive period of normal mode)	I_{DDSB}			0.2	0.5	µA
Current during humidity measurement	I_{DDH}	Max value at 85 °C		340		µA
Current during pressure measurement	I_{DDP}	Max value at -40 °C		714		µA
Current during temperature measurement	I_{DDT}	Max value at 85 °C		350		µA
Start-up time	$t_{STARTUP}$	Time to first communication after both VDD > 1.58 V and VDDIO > 0.65 V			2	ms
Power supply rejection ratio (DC)	PSSR	full VDD range			±0.01 ±5	%RH/V Pa/V
Standby time accuracy	$\Delta t_{STANDBY}$			±5	±25	%

Table 30: General electrical specifications

Parameter	Symbol	Condition	Min	Typ	Max	Parameter
Operating range[2]	R_H	For temperatures	-40	25	85	°C
		< 0 °C and > 60 °C	0		100	%RH
Supply current	$I_{DD, H}$	1 Hz forced mode, humidity and temperature		1.8	2.8	µA
Absolute accuracy tolerance	A_H	20…80 %RH, 25 °C, including hysteresis		±3		%RH
Hysteresis[3]	H_H	10→90→10 %RH, 25 °C		±1		%RH
Nonlinearity[4]	NL_H	10→90 %RH, 25 °C		1		%RH
Response time to complete 63% of step[5]	$\tau_{63\%}$	90→0 or 0→90 %RH, 25°C		1		s
Resolution	R_H			0.008		%RH
Noise in humidity (RMS)	N_H	Highest oversampling		0.02		%RH
Long term stability	ΔH_{STAB}	10…90 %RH, 25 °C		0.5		%RH/year

Table 31: Humidity sensor specifications[1]

[1]Target values

[2]When exceeding the operating range (e.g. for soldering), humidity sensing performance is temporarily degraded and reconditioning is recommended. Operating range only for non-condensing environment

[3]For hysteresis measurement the sequence 10→30→50→70→90→70→50→30→10 %RH is used. The hysteresis is defined as the difference between measurements of the humidity up/down branch and the averaged curve of both branches

[4]Non-linear contributions to the sensor data are corrected during the calculation of the relative humidity

[5]The airflow in direction to the vent-hole of the device must be dimensioned in a way that sufficient air exchange inside to outside will be possible. To observe effects on the response timescale of the device an air-flow velocity of approx. 1 m/s is needed

Parameter	Symbol	Condition	Min	Typ	Max	Parameter
Operating temperature range	T_A	operational	-40	25	+85	°C
		full accuracy	0		+65	
Operating pressure range	P	full accuracy	300		1100	hPa
Supply current	$I_{DD, LP}$	1 Hz forced mode, pressure and temperature, lowest power		2.8	4.2	µA
Temperature coefficient of offset[1]	TCO_P	25…65 °C, 900 hPa		±1.5		Pa/K
				±12.6		cm/K
Absolute accuracy pressure	$A_{P, full}$	300 . . . 1100 hPa 0 . . . 65 °C		±1.0		hPa
Relative accuracy pressure V_{DD} = 3.3V	A_{REL}	700 … 900hPa 25 . . . 40 °C		±0.12		hPa
Resolution of pressure output data	R_P	Highest oversampling		0.18		Pa
Noise in pressure	$N_{P, FULL-BW}$	Full bandwidth, highest oversampling		1.3		Pa
				11		cm
	$N_{P, FILTERED}$	Reduced bandwidth, highest oversampling		0.2		Pa
				1.7		cm
Solder drift		Minimum solder height 50µm	-0.5		+2.0	hPa
Long term stability[2]	ΔP_{STAB}	per year		±1.0		hPa
Possible sampling rate	f_{SAMPLE_P}	Lowest oversampling	157	182		Hz

Table 32: Pressure sensor specifications

[1]When changing temperature by e.g. 10 °C at constant pressure/altitude, the measured pressure/altitude will change by (10 × TCO_P)

[2]Long term stability is specified in the full accuracy operating pressure range 0…65 °C

Parameter	Symbol	Condition	Min	Typ	Max	Parameter
Operating range	T	Operational	-40	25	85	°C
		Full accuracy	0		65	°C
Supply current	$I_{DD, T}$	1 Hz forced mode, temperature measurement only		1.0		µA
Absolute accuracy temperature[1]	$A_{T, 25}$	25 °C		±0.5		°C
	$A_{T, FULL}$	0…65 °C		±1.0		°C
Output resolution	R_T	API output resolution		0.01		°C
RMS noise	N_T	Lowest oversampling		0.005		°C

Table 33: Temperature sensor specifications

[1]Temperature measured by the internal temperature sensor. This temperature value depends on the PCB temperature, sensor element self-heating and ambient temperature and is typically above ambient temperature

Parameter	Condition	Min	Max	Unit
Voltage at any supply pin	V_{DD} and V_{DDIO} pin	-0.3	4.25	V
Voltage at any interface pin		-0.3	VDDIO + 0.3	V
Storage temperature	≤ 65% RH	-45	+85	°C
Pressure		0	20 000	hPa
ESD	HBM, at any pin		±2	kV
	CDM		±500	V
	Machine model		±200	V
Condensation	No power supplied	Allowed		

Table 34: Absolute maximum ratings

4.1.2: CCS811

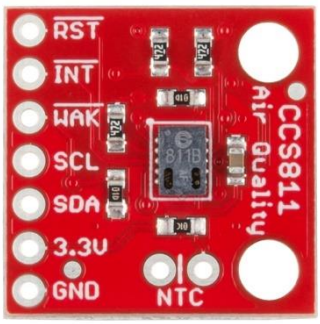


Figure 34: ams AG CCS811 SparkFun breakout board

The CCS811 is an ultra-low power digital gas sensor solution which integrates a metal oxide (MOX) gas sensor to detect a wide range of Volatile Organic Compounds (VOCs) for indoor air quality monitoring with a microcontroller unit (MCU), which includes an Analog-to-Digital converter (ADC), and an I^2C interface.

CCS811 is based on ams unique micro-hotplate technology which enables a highly reliable solution for gas sensors, very fast cycle times and a significant reduction in average power consumption. The integrated MCU manages the sensor drive modes and raw sensor data measured while detecting VOCs. The I^2C digital interface significantly simplifies the hardware and software design, enabling a faster time to market.

CCS811 supports intelligent algorithms to process raw sensor measurements to output a TVOC value or equivalent CO_2 (eCO_2) levels, where the main cause of VOCs is from humans. It supports multiple measurement modes that have been optimized for low-power consumption during an active sensor measurement and idle mode extending battery life in portable applications.

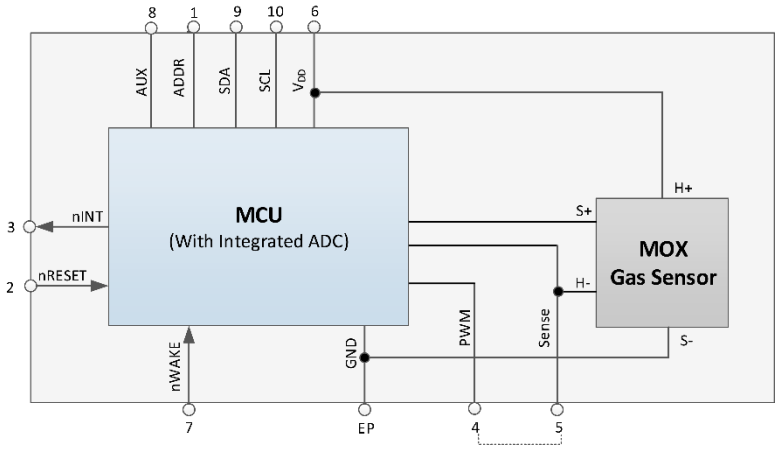


Figure 35: CCS811 functional block diagram

The CCS811 has 5 modes of operation as follows

- **Mode 0**: Idle, low current mode
- **Mode 1**: Constant power mode, IAQ measurement every second
- **Mode 2**: Pulse heating mode IAQ measurement every 10 seconds
- **Mode 3**: Low power pulse heating mode IAQ measurement every 60 seconds
- **Mode 4**: Constant power mode, sensor measurement every 250ms

In Modes 1, 2, 3, the equivalent CO_2 concentration (ppm) and TVOC concentration (ppb) are calculated for every sample. Mode 1 reacts fastest to gas presence but has a higher operating current. Mode 3 reacts more slowly to gas presence but has the lowest average operating current.

When a sensor operating mode is changed to a new mode with a lower sample rate (e.g. from Mode 1 to Mode 3), it should be placed in Mode 0 (Idle) for at least 10 minutes before enabling the new mode. When a sensor operating mode is changed to a new mode with a higher sample rate (e.g. from Mode 3 to Mode

1), there is no requirement to wait before enabling the new mode. Mode 4 is intended for systems where an external host system wants to run an algorithm with raw data and this mode provides new sample data every 250ms.

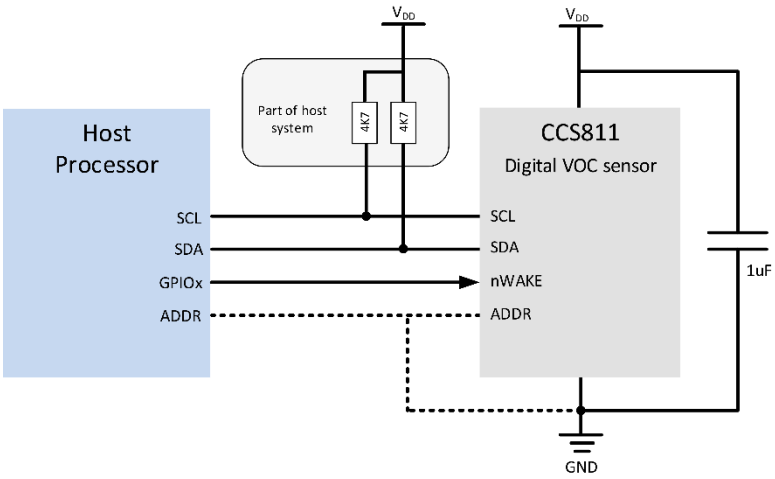


Figure 36: CCS811 typical application circuit

CCS811 performance in terms of resistance levels and sensitivities will change during early life use. The change in resistance is greatest over the first 48 hours of operation. ams advises customers to run CCS811 for 48 hours in the selected mode of operation to ensure sensor performance is stable. After early-life (Burn-In) period is complete the conditioning period is the time required to achieve good sensor stability before measuring VOCs after long idle period. After writing to MEAS_MODE to configure the sensor in mode 1-4, run CCS811 for 20 minutes, before accurate readings are generated. The conditioning period must also be observed before writing to the BASELINE register.

The equivalent CO_2 (eCO_2) output range for CCS811 is from 400ppm to 8192ppm. Values outside this range are clipped.

The Total Volatile Organic Compound (TVOC) output range for CCS811 is from 0ppb to 1187ppb. Values outside this range are clipped. This is calibrated to a typical TVOC mixture in an indoor environment. If the ratio of compounds in the environment is significantly different the TVOC output will be affected as some VOC compounds will have greater or lesser effect on the sensor.

Symbol	Parameter	Min	Max	Units	Comments
Electrical Parameters					
V_{DD}[1]	Supply Voltage	1.8[2]	3.6	V	
I_{DD}	Supply Current		30	mA	In mode 1
P	Power Consumption		60	mW	In mode 1
Electrostatic Discharge					
ESD_{HBM}	Human Body Model	±2000		V	
Environmental Conditions					
T_{AMB}	Ambient Temperature for Operation	-5	50	°C	
T_{STRG}	Storage Temperature	-40	125	°C	
RH_{NC}	Relative Humidity (non-condensing)	10	95	%	
MSL	Moisture Sensitivity Level	1			Unlimited max. floor lifetime

Table 35: Absolute Maximum Ratings

[1]The supply voltage VDD is sampled during boot and should not vary during operation.

[2]The minimum supply voltage VDD is 1.8V and should not drop below this value for reliable device operation.

Stresses beyond those listed under Absolute Maximum Ratings may cause permanent damage to the device. These are stress ratings only. Functional operation of the device at these or any other conditions beyond those indicated under Electrical Characteristics is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Parameters	Conditions	Min	Typ	Max	Units
Supply Voltage (V_{DD})		1.8		3.3	V
Supply Current (I_{DD})	During measuring at 1.8V		26		mA
	Average over pulse cycle at 1.8V		0.7		mA
	Sleep Mode at 1.8V		19		µA
Power Consumption	Idle Mode 0 at V_{DD}= 1.8V		0.034		mW
	Mode 1 & 4 at V_{DD}= 1.8V		46		mW
	Mode 2 at V_{DD}= 1.8V		7		mW
	Mode 3 at V_{DD}= 1.8V		1.2		mW
Logic High Input	nRESET, nWAKE, ADDR	V_{DD} - 0.6		V_{DD}	V
Logic Low Input	nRESET, nWAKE, ADDR	0		0.6	V
Logic High Output	nINT	V_{DD} - 0.7			V
Logic Low Output	nINT			0.6	V
Analogue Input	AUX	0		V_{DD}	V

Table 36: Electrical Characteristics

Parameters	Conditions	Min	Typ	Max	Units
t_{AWAKE}[1]	Time until active after nWAKE asserted	50			µs
t_{DWAKE}	Minimum time nWAKE should be de-asserted	20			µs
t_{RESET}	Minimum nRESET low pulse	20			Ms
t_{START}[2]	Time until active after Power on		18	20	ms
	Time until active after nRESET		1	2	ms
f_{I2C}	Frequency of I²C Bus Supported	10	100	400	kHz

Table 37: Timing Characteristics

[1]nWAKE should be asserted prior to and during any I²C transaction

[2]Up to 70ms on the first Reset after new application download

4.1.3: SPS30

Figure 37: Sensirion SPS30

Figure 38: Sensirion SPS30 with scale

The SPS30 Particulate Matter (PM) sensor is an optical Particulate Matter sensor. Its measurement principle is based on laser scattering and makes use of Sensirion's innovative contamination resistance technology. This technology, together with high-quality and long-lasting components, enables accurate measurements from its first operation and throughout its lifetime of more than eight years. In addition, Sensirion's advanced algorithms provide superior accuracy for different PM types and higher-resolution

particle size binning, allowing the detection of different sorts of environmental dust and other particles. With dimensions of only 41 x 41 x 12 mm3, it is the perfect solution for a portable application such as the EnvironMetric's sensor stack.

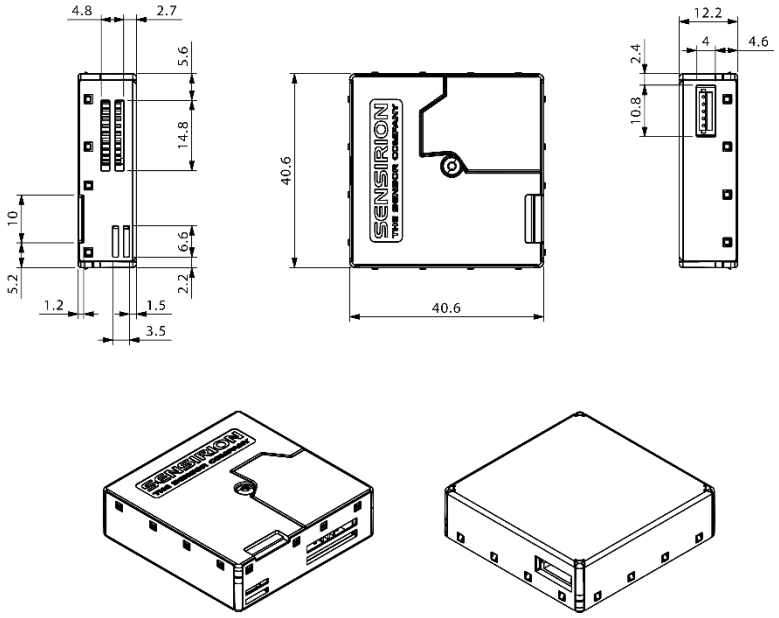


Figure 39: Package outline dimensions (in mm) of the SPS30

Parameter	Conditions	Value	Units
Mass concentration accuracy[1]	0 to 100 µg/m³	±10	µg/m³
	100 to 1'000 µg/m³	±10	%
Mass concentration range	-	0 to 1'000	µg/m³
Mass concentration resolution	-	1	µg/m³
Mass concentration size range[2]	PM1.0	0.3 to 1.0	µm
	PM2.5	0.3 to 2.5	µm
	PM4	0.3 to 4.0	µm
	PM10	0.3 to 10.0	µm
Number concentration range	-	0 to 3'000	1/cm³
Number concentration size range[2]	PM0.5	0.3 to 0.5	µm
	PM1.0	0.3 to 1.0	µm
	PM2.5	0.3 to 2.5	µm
	PM4	0.3 to 4.0	µm
	PM10	0.3 to 10.0	µm
Sampling interval	-	1	S
Start-up time	-	< 8	S
Lifetime[3]	24 h/day operation	> 8	Years
Acoustic emission level	0.2 m	25	dB(A)
Weight	-	26	g

Table 38: Particulate Matter Sensor Specifications

[1]Deviation to TSI DustTrak™ DRX Aerosol Monitor 8533 reference. PM2.5 accuracy is verified for every sensor after calibration using a defined potassium chloride particle distribution. Ask Sensirion for further details on accuracy characterization procedures

[2]PMx defines particles with a size smaller than "x" micrometers (e.g. PM2.5 = particles smaller than 2.5 µm)

[3]Validated with accelerated aging tests. Ask Sensirion for further details on accelerated aging validation procedures. Lifetime might vary depending on different operating conditions

Parameter	Conditions	Value	Units
Supply voltage	-	4.5 to 5.5	V
Idle current	Idle-Mode	< 8	mA
Average supply current	Measurement-Mode	60	mA
Max. supply current	First ~200 ms after start of Measurement-Mode	80	mA
Input high level voltage (V_{IH})	-	> 2.31	V
Input low level voltage (V_{IL})	-	< 0.99	V
Output high level voltage (V_{OH})	-	> 2.9	V
Output low level voltage (V_{OL})	-	< 0.4	V

Table 39: Electrical Characteristics

Parameter	Rating
Supply voltage V_{DD}	-0.3 to 5.5 V
Interface Select SEL	-0.3 to 4.0 V
I/O pins (RX/SDA, TX/SCL)	-0.3 to 5.5 V
Max. current on any I/O pin	±16 mA
Operating temperature range	-10 to +60 °C
Storage temperature range	-40 to +70 °C
Operating humidity range	0 to 95 %RH (non-condensing)
ESD CDM (charge device model)[1]	±4 kV contact, ±8 kV air
Electromagnetic field immunity to high frequencies[2]	3 V/m (80 MHz to 1000 MHz)
High frequency electromagnetic emission[3]	30 dB 30 MHz to 230 MHz; 37 dB 230 MHz to 1000 MHz
Low frequency electromagnetic emission[4]	30-40 dB 0.15 MHz to 30 MHz

Table 40: Absolute Minimum and Maximum Ratings

[1]According to IEC 61000-4-2.
[2]According to IEC 61000-4-3.
[3]According to CISPR 14.
[4]According to CISPR 22.

4.1.4: SGP30

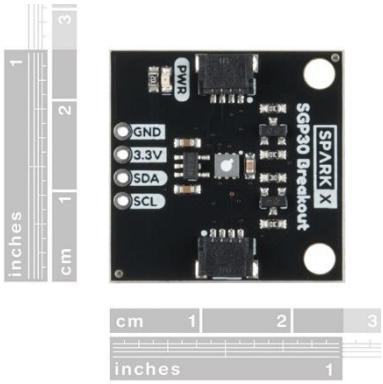


Figure 40: Sensirion SGP30 on a SparkFun breakout board with scale

The SGP30 is a digital multi-pixel gas sensor designed for easy integration into IoT applications. Sensirion's CMOSens technology offers a complete sensor system on a single chip featuring a digital I2C interface, a temperature controlled micro hotplate, and two preprocessed indoor air quality signals. As the first metal-oxide gas sensor featuring multiple sensing elements on one chip, the SGP30 provides more detailed information about the air quality. The sensing element features an unmatched robustness against contaminating gases present in real-world applications enabling a unique long-term stability and low drift. The very small 2.45 x 2.45 x 0.9 mm3 DFN package enables applications in limited spaces. Sensirion's state-of-the-art production process guarantees high reproducibility and reliability. Tape and reel packaging,

together with suitability for standard SMD assembly processes make the SGP30 predestined for high-volume applications.

Parameter	Signal	Values	Comments	
Measurement range[1]	Ethanol signal	0 ppm to 1000 ppm		
	H_2 signal	0 ppm to 1000 ppm		
Specified measurement range	Ethanol signal	0.3 ppm to 30 ppm	The specifications below are defined for this measurement range[2]. The specified measurement range covers the gas concentrations expected in indoor air quality applications.	
	H_2 signal	0.5 ppm to 10 ppm		
Accuracy[3]	Ethanol signal	typical: 15% of meas. value	Accuracy of the concentration c determined by $\ln(c/c_{REF}) = \frac{(s_{REF} - s_{OUT})}{a}$ $a = 512$ $s_{OUT} = {C_2H_5OH}/{H_2}$ signal output at concentration c $s_{REF} = {C_2H_5OH}/{H_2}$ signal output at 0.5 ppm H_2	$c_{REF} = 0.4$ ppm
	H_2 signal	typical: 10% of meas. value		$c_{REF} = 0.5$ ppm
Long-term drift[3, 4]	Ethanol signal	typical: 1.3% of meas. value	Change of accuracy over time: Siloxane accelerated lifetime test[5]	
	H_2 signal	typical: 1.3% of meas. value		
Resolution	Ethanol signal	0.2 % of meas. value	Resolution of Ethanol and H_2 signal outputs in relative change of the measured concentration	
	H_2 signal			
Sampling frequency	Ethanol signal	Max. 40 Hz	Compare with minimum measurement duration	
	H_2 signal			

Table 41: Gas Sensing Performance

[1]Exposure to ethanol and H2 concentrations up to 1000 ppm have been tested. For applications requiring the measurement of higher gas concentrations please contact Sensirion

[2]ppm: parts per million. 1 ppm = 1000 ppb (parts per billion)

[3]90% of the sensors will be within the typical accuracy tolerance, >99% are within the maximum tolerance

[4]The long-term drift is stated as change of accuracy per year of operation

[5]Test conditions: operation in 250 ppm deca-methyl-cyclo-penta-siloxane (D5) for 200h simulating 10 years of operation in an indoor environment

Parameter	Signal	Values		Comments
Output range	TVOC signal	0 ppb to 60000 ppb		Maximum possible output range
	CO_2eq signal	400 ppm to 60000 ppm		
Resolution	TVOC signal	0 ppb - 2008 ppb	1 ppb	
		2008 ppb – 11110 ppb	6 ppb	
		11110 ppb – 60000 ppb	32 ppb	
	CO_2eq signal	400 ppm – 1479 ppm	1 ppm	
		1479 ppm – 5144 ppm	3 ppm	
		5144 ppm – 17597 ppm	9 ppm	
		17597 ppm – 60000 ppm	31 ppm	
Sampling rate	TVOC signal	1 Hz		The on-chip baseline compensation algorithm has been optimized for this sampling rate. The sensor shows best performance when used with this sampling rate.
	CO_2eq signal	1 Hz		

Table 42: Air Quality Signals

The sensor shows best performance when operated within recommended normal temperature and humidity range of 5 – 55 °C and 4 –20 g/m3, respectively. Long-term exposure (operated and not operated) to conditions outside the recommended range, especially at high humidity, may affect the sensor performance. Prolonged exposure to extreme conditions may accelerate aging

Parameter	Min.	Typ.	Max.	Unit	Comments
Supply voltage V_{DD}	1.62	1.8	1.98	V	Minimal voltage must be guaranteed also for the specified maximum supply current.
Hotplate supply voltage V_{DDH}	1.62	1.8	1.98	V	
Supply current in measurement mode		48.2		mA	The measurement mode is activated by sending an “Init_air_quality” or “Measure_raw_signal” command. Specified at 25°C and typical V_{DD}
Sleep current		2	10	µA	The sleep mode is activated after power-up or after a soft reset. Specified at 25°C and typical V_{DD}
LOW-level input voltage	-0.5		$0.3*V_{DD}$	V	
HIGH-level input voltage	$0.7*V_{DD}$		$V_{DD}+0.5$	V	
V_{HYS} hysteresis of Schmitt trigger inputs			$0.05*V_{DD}$	V	
LOW-level output voltage			$0.2*V_{DD}$	V	(open-drain) at 2mA sink current
Communication	Digital 2-wire interface, I^2C fast mode.				

Table 43: Electrical Specifications

The SGP30 supports I²C fast mode. For detailed information on the I²C protocol, refer to NXP I²C-bus specification8. All SGP30 commands and data are mapped to a 16-bit address space. Additionally, data and commands are protected with a CRC checksum to increase the communication reliability. The 16-bit commands that are sent to the sensor already include a 3-bit CRC checksum. Data sent from and received by the sensor is always succeeded by an 8-bit CRC. In write direction it is mandatory to transmit the checksum, since the SGP30 only accepts data if it is followed by the correct checksum. In read direction it is up to the master to decide if it wants to read and process the checksum.

The typical communication sequence between the I²C master (e.g., a microcontroller in a host device) and the sensor is described as follows:

1. The sensor is powered up, communication is initialized
2. The I²C master periodically requests measurement and reads data, in the following sequence:
    a. I²C master sends a measurement command
    b. I²C master waits until the measurement is finished, either by waiting for the maximum execution time or by waiting for the expected duration and then poll data until the read header is acknowledged by the sensor
    c. I²C master reads out the measurement result

The sensor starts powering-up after reaching the power-up threshold voltage VPOR. After reaching this threshold voltage, the sensor needs the time t_{PU} to enter the idle state. Once the idle state is entered it is ready to receive commands from the master. Each transmission sequence begins with a START condition (S) and ends with a STOP condition (P) as described in the I²Cbus specification.

A measurement communication sequence consists of a START condition, the I²C write header (7-bit I²C device address plus 0 as the write bit) and a 16-bit measurement command. The proper reception of each byte is indicated by the sensor. It pulls the SDA pin low (ACK bit) after the falling edge of the 8th SCL clock to indicate the reception. With the acknowledgement of the measurement command, the SGP30 starts measuring. When the measurement is in progress, no communication with the sensor is possible and the sensor aborts the communication with a XCK condition. After the sensor has completed the measurement, the master can read the measurement results by sending a START condition followed by an I²C read header. The sensor will acknowledge the reception of the read header and responds with data. The response is structured in data words, where one word consists of two bytes of data followed by one byte CRC checksum. Each byte must be acknowledged by the microcontroller with an ACK condition for the sensor to continue sending data. If the sensor does not receive an ACK from the master after any byte of data, it will not continue sending data. After receiving the checksum for the last word of data, an XCK and STOP condition must be sent. The I²C master can abort the read transfer with a XCK followed by a STOP condition after any data byte if it is not interested in subsequent data, e.g. the CRC byte or following data bytes, in order to save time. Note that the data cannot be read more than once, and access to data beyond the specified amount will return a pattern of 1s.

The SGP30 uses a dynamic baseline compensation algorithm and on-chip calibration parameters to provide two complementary air quality signals. Based on the sensor signals a total VOC signal (TVOC) and a CO2 equivalent signal (CO2eq) are calculated. Sending an “Init_air_quality” command starts the air quality measurement. After the “Init_air_quality” command, a “Measure_air_quality” command must be sent in regular intervals of 1s to ensure proper operation of the dynamic baseline compensation algorithm. The sensor responds with 2 data bytes (MSB first) and 1 CRC byte for each of the two preprocessed air quality signals in the order CO2eq (ppm) and TVOC (ppb). For the first 15s after the “Init_air_quality” command the sensor is in an initialization phase during which a “Measure_air_quality” command returns fixed values of 400 ppm CO2eq and 0 ppb TVOC. The SGP30 also provides the possibility to read and write the baseline values of the baseline correction algorithm. This feature is used to save the baseline in regular intervals on an external non-volatile memory and restore it after a new power-up or soft reset of the sensor. The command “Get_baseline” returns the baseline values for the two air quality signals. The sensor responds with 2 data bytes (MSB first) and 1 CRC byte for each of the two values in the order CO2eq and TVOC. These two values should be stored on an external memory. After a power-up or soft reset, the baseline of the baseline correction algorithm can be restored by sending first an “Init_air_quality” command followed by a “Set_baseline” command with the two baseline values as parameters in the order as (TVOC, CO2eq). An example implementation of a generic driver for the baseline algorithm can be found in the document SGP30_driver_integration_guide. A new “Init_air_quality” command has to be sent after every power-up or soft reset.

The command “Measure_raw_signals” is intended for part verification and testing purposes. It returns the sensor raw signals which are used as inputs for the on-chip calibration and baseline compensation algorithms. The command performs a measurement to which the sensor responds with 2 data bytes (MSB first) and 1 CRC byte for 2 sensor raw signals in the order H2_signal (s_{OUT_H2}) and Ethanol_signal ($s_{OUT\_ETHANOL}$). Both signals can be used to calculate gas concentrations c relative to a reference concentration c_{REF} by

$$\ln\left(c/c_{REF}\right) = \frac{(s_{REF} - s_{OUT})}{a}$$

with $a = 512$, s_{REF} the H2_signal or Ethanol_signal output at the reference concentration, and $s_{OUT} = s_{OUT\_H2}$ or $s_{OUT} = s_{OUT\_ETHANOL}$

The SGP30 features an on-chip humidity compensation for the air quality signals (CO_2eq and TVOC) and sensor raw signals (H2-signal and Ethanol_signal). To use the on-chip humidity compensation an absolute humidity value from an external humidity sensor like the SHTxx is required. Using the “Set_humidity” command, a new humidity value can be written to the SGP30 by sending 2 data bytes (MSB first) and 1 CRC byte. The 2 data bytes represent humidity values as a fixed-point 8-bit number with a minimum value of 0x0001 (=1/256 g/m^3) and a maximum value of 0xFFFF (255 g/m^3 + 255/256 g/m^3). For instance, sending a value of 0x0F80 corresponds to a humidity value of 15.50 g/m^3 (15 g/m^3 + 128/256 g/m^3). After setting a new humidity value, this value will be used by the on-chip humidity compensation algorithm until a new humidity value is set using the “Set_humidity” command. Restarting the sensor (power-on or soft reset) or sending a value of 0x0000 (= 0 g/m^3) sets the humidity value used for compensation to its default value (0x0B92 = 11.57 g/m^3) until a new humidity value is sent. Sending a humidity value of 0x0000 can therefore be used to turn off the humidity compensation.

4.1.5: BNO055

The BNO055 is a System in Package (SiP), integrating a triaxial 14-bit accelerometer, a triaxial 16-bit gyroscope with a range of ±2000 degrees per second, a triaxial geomagnetic sensor and a 32-bit cortex M0+ microcontroller running Bosch Sensortec sensor fusion software, in a single package. The corresponding chipsets are integrated into one single 28-pin LGA 3.8mm x 5.2mm x 1.1 mm housing. For optimum system integration the BNO055 is equipped with digital bidirectional I2C and UART interfaces. The I2C interface can be programmed to run with the HID-I2C protocol turning the BNO055 into a plug-and-play sensor hub solution for devices running the Windows 8.0 or 8.1 operating system.

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Start	T_{SUP}	From Off to configuration mode		400		ms
POR Time	T_{POR}	From Reset to Normal mode		650		ms
Data Rate	DR	s. Par Fusion Data Rates				
Data Rate Tolerance 9DOF @100Hz output data rate (if internal oscillator is used)	DR_{TOL}			±1		%

Table 44: Operating Conditions BNO055

Parameter	Symbol	Condition	Min	Max	Unit
Voltage at Supply Pin	V_{DD} Pin		-0.3	4.2	V
	V_{DDIO} Pin		-0.3	3.6	V
Voltage at any logic pin	$V_{non\ supply}$ Pin		-0.3	V_{DDIO}+0.3	V
Passive storage Temp. Range	T_{rps}	≤ 65% rel. H.	-50	+150	°C
Mechanical Shock	$MechShock_{200µs}$	Duration ≤ 200µs		10,000	g
	$MechShock_{1ms}$	Duration ≤ 200µs		2,000	g
	$MechShock_{freesfall}$	Free fall onto hard surfaces		1.8	M
ESD	ESD_{HBM}	HBM, at any Pin		2	kV
	ESD_{CDM}	CDM		400	V
	ESD_{MM}	MM		200	V

Table 45: Absolute Maximum Rating (preliminary target values)

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Acceleration Range	g_{FS2g}	Selectable via serial digital interface		±2		g
	g_{FS4g}			±4		g
	g_{FS8g}			±8		g
	g_{FS16g}			±16		g

Table 46: Operating Conditions Accelerometer

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Sensitivity	S	All g_{FSXg} Values, T_A=25°C		1		LSB/mg
Sensitivity Tolerance	S_{TOL}	T_a=25°C, g_{FS2g}		±1	±4	%
Sensitivity Temperature Drift	TCS	g_{FS2g}, Nominal V_{DD} supplies, Temp operating conditions		±0.03		%/K
Sensitivity Supply Volt. Drift	S_{VDD}	g_{FS2g}, T_A=25°C, $V_{DD\_min} \leq V_{DD} \leq V_{DD\_max}$		0.065	0.2	%/V
Zero-g Offset (x,y.z)	Off_{xyz}	g_{FS2g}, T_A=25°C, nominal V_{DD} supplies, over lifetime	-150	±80	±150	mg
Zero-g Offset Temperature Drift	TCO	g_{FS2g}, Nominal VDD supplies		±1	+/- 3.5	mg/K
Zero-g Offset Supply Volt. Drift	Off_{VDD}	g_{FS2g}, T_A=25°C, $V_{DD\_min} \leq V_{DD} \leq V_{DD\_max}$		1.5	2.5	mg/V
Bandwidth	bw_8	2nd order filter, bandwidth programmable		8		Hz
	bw_{16}			16		Hz
	bw_{31}			31		Hz
	bw_{63}			63		Hz
	bw_{125}			125		Hz
	bw_{250}			250		Hz
	bw_{500}			500		Hz
	bw_{1000}			1000		Hz
Nonlinearity	NL	best fit st raight line, g_{FS2g}		0.5	2	%FS
Output Noise Density	n_{rms}	g_{FS2g}, T_A=25°C Nominal V_{DD} supplies Normal mode		150	190	µg/√Hz

Table 47: Output Signal Accelerometer (Accelerometer Only Mode)

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Rate Range	R_{FS125}	Selectable via serial digital interface		125	°/s	%
	R_{FS250}			250	°/s	
	R_{FS500}			500	°/s	
	R_{FS1000}			1000	°/s	
	R_{FS2000}			2000	°/s	

Table 48: Operating Conditions Gyroscope

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Sensitivity via register	S	$T_A = 25°C$		16.0 900		LSB/°/s Rad/s
Sensitivity Tolerance	S_{TOL}	T_a=25°C, R_{FS2000}		±1	±3	%
Sensitivity change over Temperature	TCS	Nominal VDD supplies -40°C ≤T_A≤ + 85°C, R_{FS2000}		±0.03	±0.07	%/K
Sensitivity Supply Volt. Drift	S_{VDD}	T_A=25°C, $V_{DD\_min} \leq V_{DD} \leq V_{DD\_max}$		<0.4		%/V
Nonlinearity	NL	best fit st raight line, R_{FS2000}, R_{FS2000}		±0.05	±0.2	%FS
Zero rate Offset	Off Ω_x Ω_y and Ω_z	T_A=25°C, nominal V_{DD} supplies, slow and fast offset cancellation off	-3	±1	+3	°/s
Zero-Ω Offset Temperature Drift	TCO	Nominal VDD supplies -40°C ≤T_A≤ + 85°C, R_{FS2000}		±0.015	±0.03	°/s/K
Zero-Ω Offset Supply Volt. Drift	Off Ω_{VDD}	T_A=25°C, $V_{DD\_min} \leq V_{DD} \leq V_{DD\_max}$		0.1		°/s/V
Output Noise	n_{rms}	rms, BW=47Hz (@ 0.014°/s/√Hz)		0.1	0.3	°/s
Bandwidth BW	f_{-3dB}			523 230 116 64 47 32 23 12		Hz

Table 49: Output Signal Gyroscope (Gyro Only Mode)

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Magnetic Field Range	BRG, xy	TA = 25°C	±1200	±1300		µT
	BRG, Z		±2000	±2500		µT
Magnetometer Heading accuracy	As Heading	30µT horizontal geomagnetic field component, TA=25°C			±2.5	deg

Table 50: Operating Conditions Magnetometer (Magnetometer only mode)

Parameter	Symbol	Condition	Min	Typ	Max	Unit
Device Resolution	$D_{res,m}$	T_A = 25°C		0.3		µT
Gain Error	$G_{err,m}$	After API compensat ion T_A=25°C Nominal VDD supplies		±5	±8	%
Sensitivity Temperature Drift	TCS_m	After API compensat ion -40°C ≤ T_A ≤ + 85°C Nominal VDD supplies		±0.01	±0.03	%/K
Zero-B offset	OFF_m	T_A = 25°C		±40		µT
Zero-B offset	$OFF_{m.cal}$	After calibration in fusion mode -40°C ≤ T_A ≤ + 85°C		±2		µT
Zero-B offset temperature drift	TCO_m	-40°C ≤ T_A ≤ +85°C		±0.23	±0.37	µT/K
Full scale nonlinearity	$NL_{m, FS}$	best fit straight line			1	%FS
Output Noise	$n_{rms,lp,m,xy}$	Low power preset x, y-axis, T_A=25°C Nominal V_{DD} supplies		1.0		µT
	$n_{rms,lp,m,z}$	Low power preset z-axis, T_A=25°C Nominal V_{DD}		1.4		µT
	$n_{rms,rg,m}$	Regular Preset. T_A=25°C Nominal V_{DD}		0.6		µT
	$n_{rms,eh,m}$	Enhanced Regular Preset. T_A=25°C Nominal V_{DD}		0.5		µT
	$n_{rms,ha,m}$	High accuracy preset, T_A=25°C Nominal V_{DD}		0.3		µT
Power Supply Rejection Rate	$PSRR_m$	T_A=25°C Nominal V_{DD} supplies		±0.5		µT/V

Table 51: Magnetometer Output Signal

The BNO055 has two distinct power supply pins:

- V_{DD} is the main power supply for the internal sensors
- V_{DDIO} is a separate power supply pin used for the supply of the µC and the digital interfaces

For the switching sequence of power supply VDD and VDDIO it is mandatory that VDD is powered on and driven to the specified level before or at the same time as VDDIO is powered ON. Otherwise there are no limitations on the voltage levels of both pins relative to each other, as long as they are used within the specified operating range. The sensor features a power-on reset (POR), initializing the register map with the default values and starting in CONFIG mode. The POR is executed at every power on and can also be triggered either by applying a low signal to the nRESET pin for at least 20ns or by setting the RST_SYS bit in the SYS_TRIGGER register. The BNO055 can be configured to run in one of the following power modes: normal mode, low power mode, and suspend mode.

The BNO055 support three different power modes:

- **Normal Mode**: In normal mode all sensors required for the selected operating mode are always switched ON. The register map and the internal peripherals of the MCU are always operative in this mode.
- **Low Power Mode**: If no activity (i.e. no motion) is detected for a configurable duration (default 5 seconds), the BNO055 enters the low power mode. In this mode only the accelerometer is active. Once motion is detected (i.e. the accelerometer signals an any-motion interrupt), the system is woken up and normal mode is entered. Additionally, the interrupt pins can also be configured to provide HW interrupt to the host. The BNO055 is by default configured to have optimum values for entering sleep and waking up. To restore these values, trigger system reset by setting RST_SYS bit in SYS_TRIGGER register.
- **Suspend Mode**: In suspend mode the system is paused, and all the sensors and the microcontroller are put into sleep mode. No values in the register map will be updated in this mode. To exit from suspend mode the mode should be changed by writing to the PWR_MODE register.

The BNO055 provides a variety of output signals, which can be chosen by selecting the appropriate operation mode. The table below lists the different modes and the available sensor signals.

Operating Mode		Available sensor signals			Fusion Data	
		Accel	Mag	Gyro	Relative orientation	Absolute orientation
	CONFIGMODE	-	-	-	-	-
Non-Fusion modes	ACCONLY	X	-	-	-	-
	MAGONLY	-	X	-	-	-
	GYROONLY	-	-	X	-	-
	ACCMAG	X	X	-	-	-
	ACCGYRO	X	-	X	-	-
	MAGGYRO	-	X	X	-	-
	AMG	X	X	X	-	-
Fusion modes	IMU	X	-	X	X	-
	COMPASS	X	X	-	-	X
	M4G	X	X	-	X	-
	NDOF_FMC_OFF	X	X	X	-	X
	NDOF	X	X	X	-	X

Table 52: Operating modes overview

The default operation mode after power-on is CONFIGMODE. When the user changes to another operation mode, the sensors which are required in that particular sensor mode are powered, while the sensors whose signals are not required are set to suspend mode.

- **Config Mode**: This mode is used to configure BNO, wherein all output data is reset to zero and sensor fusion is halted. This is the only mode in which all the writable register map entries can be changed. (Exceptions from this rule are the interrupt registers (INT and INT_MSK) and the operation mode register (OPR_MODE), which can be modified in any operation mode.)
- **Non-Fusion Modes**: These modes are designed to output un-calibrated data from the sensors. They offer higher data output rates and are intended to be used in conjunction with external digital signal processing and filtering algorithms.
    - **ACCONLY**: If the application requires only raw accelerometer data, this mode can be chosen. In this mode the other sensors (magnetometer, gyro) are suspended to lower the power consumption. In this mode, the BNO055 behaves like a stand-alone acceleration sensor.
    - **MAGONLY**: In MAGONLY mode, the BNO055 behaves like a stand-alone magnetometer, with acceleration sensor and gyroscope being suspended.
    - **GYROONLY**: In GYROONLY mode, the BNO055 behaves like a stand-alone gyroscope, with acceleration sensor and magnetometer being suspended.
    - **ACCMAG**: Both accelerometer and magnetometer are switched on, the user can read the data from these two sensors.

- **ACCGYRO**: Both accelerometer and gyroscope are switched on; the user can read the data from these two sensors.
- **MAGGYRO**: Both magnetometer and gyroscope are switched on, the user can read the data from these two sensors.
- **AMG**: All three sensors accelerometer, magnetometer and gyroscope are switched on.

❖ **Fusion modes**: Sensor fusion modes are meant to calculate measures describing the orientation of the device in space. It can be distinguished between non-absolute or relative orientation and absolute orientation. Absolute orientation means orientation of the sensor with respect to the earth and its magnetic field. In other words, absolute orientation sensor fusion modes calculate the direction of the magnetic north pole. In non-absolute or relative orientation modes, the heading of the sensor can vary depending on how the sensor is placed initially. All fusion modes provide the heading of the sensor as quaternion data or in Euler angles (roll, pitch and yaw angle). The acceleration sensor is both exposed to the gravity force and to accelerations applied to the sensor due to movement. In fusion modes it is possible to separate the two acceleration sources, and thus the sensor fusion data provides separately linear acceleration (i.e. acceleration that is applied due to movement) and the gravity vector.

- **IMU (Inertial Measurement Unit)**: In the IMU mode the relative orientation of the BNO055 in space is calculated from the accelerometer and gyroscope data. The calculation is fast (i.e. high output data rate).
- **COMPASS**: The COMPASS mode is intended to measure the magnetic earth field and calculate the geographic direction. The earth magnetic field is a vector with the horizontal components x, y and the vertical z component. It depends on the position on the globe and natural iron occurrence. For heading calculation (direction of compass pointer) only the horizontal components x and y are used. Therefore, the vector components of the earth magnetic field must be transformed in the horizontal plane, which requires the knowledge of the direction of the gravity vector. To summarize, the heading can only be calculated when considering gravity and magnetic field at the same time. However, the measurement accuracy depends on the stability of the surrounding magnetic field. Furthermore, since the earth magnetic field is usually much smaller than the magnetic fields that occur around and inside electronic devices, the compass mode requires calibration.
- **M4G (Magnet for Gyroscope)**: The M4G mode is similar to the IMU mode, but instead of using the gyroscope signal to detect rotation, the changing orientation of the magnetometer in the magnetic field is used. Since the magnetometer has much lower power consumption than the gyroscope, this mode is less power consuming in comparison to the IMU mode. There are no drift effects in this mode which are inherent to the gyroscope. However, as for compass mode, the measurement accuracy depends on the stability of the surrounding magnetic field. For this mode no magnetometer calibration is required and also not available.
- **NDOF_FMC_OFF**: This fusion mode is same as NDOF mode, but with the Fast Magnetometer Calibration turned 'OFF'.
- **NDOF**: This is a fusion mode with 9 degrees of freedom where the fused absolute orientation data is calculated from accelerometer, gyroscope and the magnetometer. The advantages of combining all three sensors are a fast calculation, resulting in high output data rate, and high robustness from magnetic field distortions. In this mode the Fast Magnetometer calibration is turned ON and thereby resulting in quick calibration of the magnetometer and higher output data accuracy. The current consumption is slightly higher in comparison to the NDOF_FMC_OFF fusion mode.

4.2: Asynchronous Remote Acquisition Protocol

The Asynchronous Remote Acquisition protocol is a single-producer-multi-consumer system that manages all the incoming and outgoing requests to and from the Remote Sensor Array (RSA). It is an intelligent system that performs task scheduling to effectively manage CPU time; this allows the RSA to serve multiple clients' requests concurrently. The ARA Protocol makes use of a variant of the Earliest Deadline First (EDF) scheduling algorithm to ensure the commands get executed at the intended time.

Earliest deadline first (EDF) or Least Time to Go is a dynamic priority scheduling algorithm used in real-time operating systems to place processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline; this process is the next to be scheduled for execution. EDF is an optimal scheduling algorithm on preemptive uniprocessors, in the following sense: if a collection of independent jobs, each characterized by an arrival time, an execution requirement and a deadline, can be scheduled (by any algorithm) in a way that ensures all the jobs complete by their deadline, the EDF will schedule this collection of jobs so they all complete by their deadline. The custom implementation of the EDF scheduling algorithm that's used in the ARA Protocol prioritizes the tasks by their next execution time and executes them as soon as they reach their deadline.

Each time the ARA Protocol is booted, it executes a sequence of actions that involve running POST operations to ensure all components are in working order, initializing the hardware to ensure all the sensors are configured properly, setting up internal data structures to maintain the program's state in real-time and activating several independent threads to concurrently perform different tasks throughout the session's runtime. The ARA Protocol is a low-level program that is required to frequently interact with different types of hardware. Since interacting with hardware is predominantly composed of I/O driven operations that are order of magnitude slower than typical compute operations, the ARA Protocol implements four independent threads that each handle a specific type of hardware interaction and their associated computations, they are:

- **I/O Thread**: This thread manages the modem object and is primarily responsible for performing input/output operations with the EnvironMetric Command Center. Because processing incoming commands to construct command objects sometimes require intercepting additional data packet transmissions beyond the first command encoding, this thread is also tasked with generating command objects. The execution complexity of this thread is dominated by its interactions with the modem interface that's responsible for receiving and transmitting data packets in and out of the RSA.
- **Hardware Controller Thread**: This thread is almost entirely dedicated to managing the various sensors that are connected to the RSA at any given time. The thread requests the sensors to send their up-to-date readouts and uses those to update the program's internal cache. Additionally, this thread is also responsible for performing different types of digital signal processing to filter out the noise from the sensor measurements. Since the hardware interactions are mainly composed of I/O driven operations whereas the data filtering is composed of compute operations, this thread's execution complexity is dominated by the interactions with the sensors.
- **Task Execution Thread**: This thread is primarily tasked with executing the operator requested commands; the process of executing commands may include but isn't limited to querying various programmatic aspects of the ARA Protocol to determine its current state and fetching sensor measurements from the internal readout cache. At the end of each command execution, this thread has to compute the state of the command object in order to determine if it requires another scheduled execution; if it is a persistent command, that has surplus executions remaining, the thread enqueues it onto the Waiting Task Queue or else the task is enqueued onto the Completed Task Queue. This thread doesn't necessarily perform any heavy computations or hardware interactions, so this thread's execution complexity is primarily dominated by the various semaphores it comes across when it is reading from or writing to the different queues.
- **Printer Thread**: This thread is dedicated to displaying (warning/error) messages on a display. These messages may be generated by various components and constructs of the RSA BIOS, RSA ARA Protocol etc. Depending on the display technology used at runtime, this thread might be the most complex to execute being driven heavily by I/O operations.

The Task Schedular also makes use of a multitude of queues to help it maintain orderly operation, they are:

- **Inbox Queue**: A regular queue that stores all the incoming commands that arrive at the RSA BIOS from the client Command Centers. The commands are enqueued into Inbox through a programmatic event that's raised by the modem interface whenever it receives a message. The enqueued commands are processed immediately after they are received because many of them require subsequent transmissions to decode and process into a command object.
- **Outbox Queue**: A regular queue that stores all the outgoing payloads that the RSA BIOS has generated, ready to be transmitted to the appropriate client Command Center. Payloads are generated by certain command objects at certain times when they are executed by the Task Execution Thread. The I/O Thread dequeues payloads from this queue and transmits them through the modem interface.
- **Printer Queue**: A regular queue that stores the warning and error message strings to be printed by the Printer Thread. Any component or construct within the RSA program can enqueue objects onto this queue. Only the Printer Thread can dequeue from this queue.
- **Task Wait Queue**: A Priority Queue that holds the tasks that are waiting for their deadline to expire to be executed. The queue uses a Unix timestamp as the metric to sort the queue. The I/O Thread enqueues Command Objects with their associated priority value when the objects are constructed right after they are received on the RSA side. Normally, the Task Execution Thread dequeues command objects from this queue to execute them but it may also enqueue the same object if they require further execution.
- **Task Execution Queue**: A regular queue that holds the tasks to be executed by the Task Execution Thread. Command objects are enqueued onto this queue by the I/O Thread from the Task Waiting Queue when their deadline expires. Tasks are dequeued by the Task Execution Thread when they are about to be executed.
- **Task Abort Queue**: A regular queue that holds all the tasks that were aborted for logging purposes.
- **Task Completed Queue**: A regular queue that holds all the tasks that were completed for logging purposes.

4.3: PolyBus

The PolyBus is a custom I2C interface that sits in between an I2C slave device and master controller. A single I2C controller can only address 128 distinct addresses and some of them are even reserved for special purposes. Should the need arise where multiple instance of the same address i.e. multiple instances of the same address (with the same I2C address), multiple I2C master controllers would be necessary. The PolyBus works on the principle of high-speed multiplexing which allows the master controller to emulate multiple instances of itself, where each instance can separately and independently address 128 distinct dev ices minus the reserved address. This allows a single microprocessor or micro controller with a single I2C master controller to be able to address more than 128 addresses and more importantly multiple instances of the same address/device.

In the scope of this application, the PolyBus allows connecting multiple temperature sensors - which have a very slow data acquisition rate because they easily overheat during fast and frequent data acquisitions thus giving false readouts - which allows for increased data updates, better noise rejection and redundancy against sensor malfunction and failure.

4A: Digital Signal Processing - Absolute Heading Reference System (DSP-AHRS)

In order to collect spatial data for the Remote Sensor Array (RSA) the EnvironMetric uses an Unmanned Aerial Vehicle (UAV) platform to traverse a large area with the RSA to collect spatial data. But to understand the pollutant data concentration of a particular position that the UAV had already traversed an accurate positioning system is required. Therefore, Attitude and Heading Reference System is implemented to find the Roll, Pitch and Yaw of the UAV from which along with GPS data of the flight controller Navio2 the accurate Position, Velocity and Heading can be found.

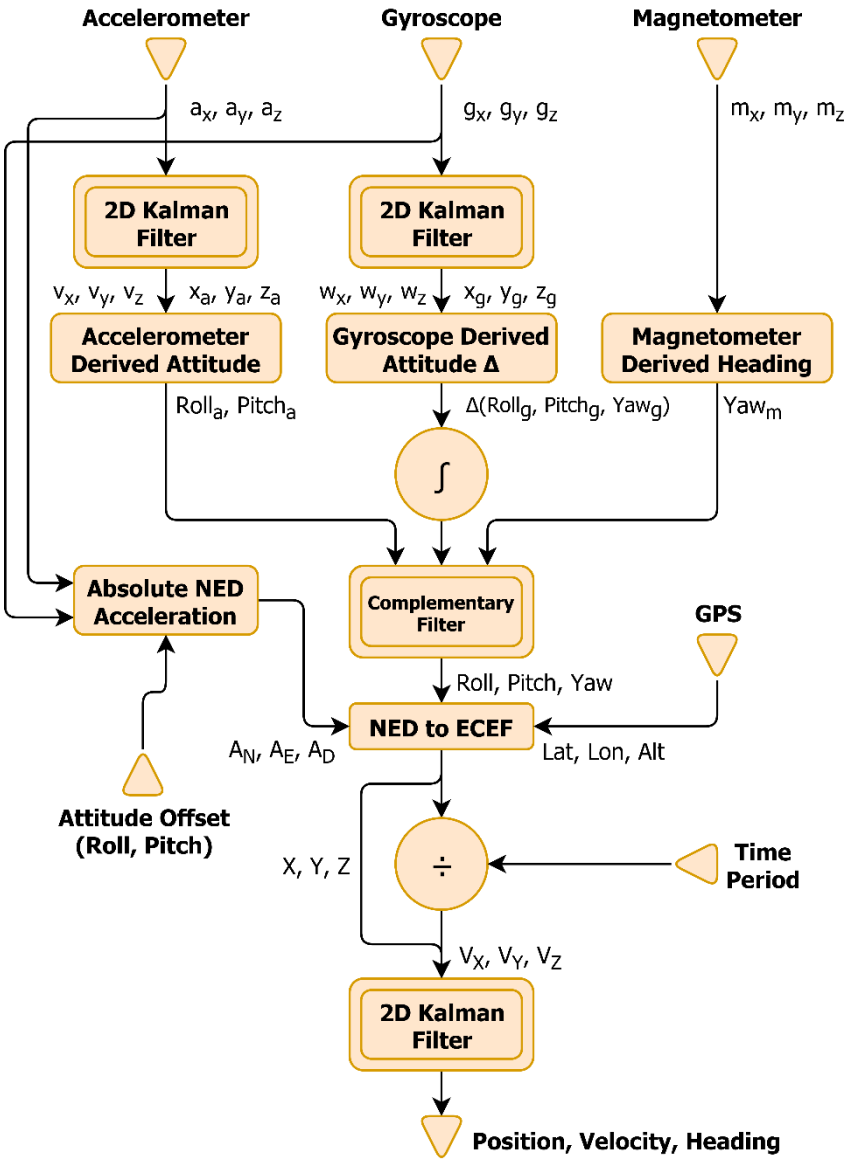


Figure 41: System level block diagram depicting the architecture of the Absolute Heading Reference System inside the Digital Signal Processor (DSP-AHRS)

The position & attitude of the UAV is found through dead reckoning the accelerometer, gyroscope & magnetometer data streams & combining them with a GPS data stream. To attenuate the noise in the accelerometer & gyroscope data streams, they are passed through two independent 2D linear Kalman filters. The attitude, Δ attitude & heading are then derived from the accelerometer, gyroscope & magnetometer respectively. A time-integral is performed on the Δ attitude from the gyroscope to get the attitude which is passed through a complementary filter in conjunction with the attitude & heading from the accelerometer & magnetometer respectively to obtain a more robust attitude estimation of the UAV. The absolute NED (North East Down) acceleration is calculated by using any attitude offsets (due to the physical sensor mount) along with the raw accelerometer & gyroscope data.

From the absolute NED acceleration, the robust attitude estimation & GPS coordinates (Latitude, Longitude, Altitude), the ECEF (Earth-Centered Earth-Fixed) coordinates (UAV position) are generated. The ECEF is differentiated with respect to time to obtain the UAV's velocity. The ECEF coordinates & velocity are passed through a 2D linear Kalman Filter to get a better position & velocity estimation of the UAV. The Attitude Heading Reference System (AHRS) algorithm output along with GPS data stream the UAV's position, heading & velocity can be achieved.

4A.1: Inertial Measurement Unit

The Inertial Measurement Unit or IMU that is used in the system is a collection of Accelerometer, Gyroscope & Magnetometer that measures a body's (UAV) specific force, angular rate and orientation.

For the project, a BNO-055 9-DOF (Degrees Of Freedom) IMU is used which uses a 3-axis Accelerometer, a 3-axis Gyroscope & a 3-axis Magnetometer to measure the linear acceleration, angular velocity and magnetic field respectively. Refer to Section 4.1.5 for further details.

The individual Accelerometer, Gyroscope and Magnetometer data collected from BNO-055 may not make sense individually but combining them using the Attitude and Heading Reference System (AHRS) can provide the orientation of the UAV in terms of Roll, Pitch and Yaw information.

4A.2: Attitude & Heading Reference System

The Attitude and Heading Reference System (AHRS) algorithm can be implemented by using the IMU. Inside the IMU both gyroscope & accelerometers provide data in high frequency but prone to noise as a result the data drift off from the actual point. But (From Figure X) the integrated output of the gyroscope combined with accelerometer output can pin down the drift of the gyroscope and can provide roll and pitch information. But the accelerometer cannot provide any yaw information. Although the integrated output of the gyroscope provides yaw data as well, but the integrated yaw output provided by the gyroscope alone cannot be trusted as the low-cost gyroscope drifts quickly. Here the magnetometer can aid to solve the problem to get the better yaw or heading information with its reading in relation to Earth's magnetic field. Thus, the algorithm determines the roll, pitch, and yaw information of the UAV.

4A.2.1: Accelerometer Derived Attitude (Roll (ϕ_a) and Pitch (θ_a))

[12] The roll and pitch of the UAV can be found from the UAV's gravity vector sensed from the Accelerometer using the following equations:

$$Absolute\ NED\ (North, East\ \&\ Down) acceleration = \begin{pmatrix} A_N \\ A_E \\ A_D \end{pmatrix}$$
$$= \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + \begin{pmatrix} 0 & v_z & v_y \\ -v_z & 0 & v_x \\ v_y & -v_x & 0 \end{pmatrix} \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} + \begin{pmatrix} -w_z^2 - w_y^2 & w_x w_y - w_z^2 & w_x w_z + g_y \\ w_x w_y + g_z & -w_x^2 - w_z^2 & w_z w_y - g_x \\ w_x w_z - g_y & w_z w_y g_x & -w_y^2 - w_x^2 \end{pmatrix} \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix}$$
$$+ g \begin{pmatrix} sin(\theta_\Delta) \\ -cos(\theta_\Delta)\, sin(\emptyset_\Delta) \\ -cos(\theta_\Delta)\, cos(\emptyset_\Delta) \end{pmatrix}$$

Equation 1: Absolute NED acceleration

Roll (ϕa) and Pitch (θa) from absolute NED acceleration:

$$\emptyset_a = tan^{-1}\left(\frac{A_E}{A_D}\right)$$

Equation 2: Roll from Absolute NED Acceleration

$$\theta_a = tan^{-1}\left(\frac{A_E}{\sqrt{A_E^2 + A_D^2}}\right)$$

Equation 3: Pitch from Absolute NED Acceleration

Here A_N, A_E, and A_D are forces from the three-axis accelerometers; v_x , v_y , and v_z are the linear velocity components along the three axes in the body frame with the origin at the center of gravity; s_x , s_y and s_z are the accelerometers coordinates or linear distance along each axis in the same body frame with the origin at the center of gravity; w_x , w_y and w_z are the three-axis gyroscope angular velocity; g_x , g_y and g_z angular accelerations along the three axes in the body frame with the origin at the center of gravity. $\Delta\phi$ and $\Delta\theta$ are the roll and pitch offset found from the BNO-055's positioning on the UAV.

4A.2.2: Gyroscope Derived Attitude (Roll (ϕ_g), Pitch (θ_g) & Yaw (Ψ_g))

The roll(ϕ_g), Pitch (θ_g) & Yaw (Ψ_g) of the UAV can be derived from the three axis Gyroscope. The equation for determining the roll, pitch, and yaw from the Gyroscope through the Euler Kinematics is given below:

$$\begin{pmatrix} \dot{\emptyset}_g \\ \dot{\theta}_g \\ \dot{\Psi}_g \end{pmatrix} = \begin{pmatrix} 1 & sin(\emptyset_\Delta)tan(\theta_\Delta) & cos(\emptyset_\Delta)tan(\theta_\Delta) \\ 0 & cos(\emptyset_\Delta) & -sin(\emptyset_\Delta) \\ 0 & sin(\emptyset_\Delta)sec(\theta_\Delta) & cos(\emptyset_\Delta)sec(\theta_\Delta) \end{pmatrix} \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix}$$

Equation 4: Gyroscope Derived Attitude

Here $\dot{\emptyset}_g$, $\dot{\theta}_g$ & $\dot{\Psi}_g$ represents rate of change of roll, pitch and yaw derived from gyroscope measured value.

4A.2.3: Magnetometer Derived Heading (Yaw (Ψ_m))

The Magnetometer can be assumed as the digital version of a compass. Thus, it can be used to find the heading of the UAV. The yaw angle determined using the magnetic field strength measured by the Magnetometer is given below:

$$\Psi_m = tan\left(\frac{m_z sin(\emptyset_\Delta) - m_x cos(\emptyset_\Delta)}{m_x cos(\theta_\Delta) + sin(\theta_\Delta)(m_y sin(\emptyset_\Delta) + m_z cos(\emptyset_\Delta))}\right)$$

Equation 5: Magnetometer Derived Heading

Here m_x, m_y and m_z are magnetic field strength along the three axes in the body frame and Ψm is the yaw or heading derived from the Magnetometer [13].

4A.3: Filter Selection

The data received from the IMU sensors are prone to Additive White Gaussian Noise (AWGN) as a result Kalman filter and Complementary filter are used.

4A.3.1: Kalman Filter

In statistics and control theory, Kalman filter a.k.a. Linear Quadratic Estimation (LQU) is an algorithm which uses several measurements over a period that contains statistical noise and inaccuracies to produce estimated measurements of that variable which tend to be more accurate than that of a single measurement alone. This algorithm was first developed by Hungarian born American electrical engineer, mathematician & inventor Rudolf Emil Kálmán. According to the type of data the filter is designed to work with, the Kalman Filter Can be divided into two categories:

Linear Kalman Filter is an algorithm that is designed to filter noise from linear data sources. Linear Kalman Filters can be designed to work with a single dimension of data (1-D Kalman Filter) or multiple dimensions of data (multi- dimensional Kalman Filter) where each dimension can be mathematically modeled/programmed to have a dependence on any of the other dimensions.

Non-Linear Kalman Filter is designed to filter out noises from non-linear data streams. Two of the most used non-linear Kalman filter algorithms are the Extended Kalman Filter and Unscented Kalman Filter.

4A.3.1.1: 2-D Kalman Filter

[14] [15] For estimating the position of the UAV 1-D linear Kalman filter is not that much of a use. Because the linear position of a drone depends on both linear velocity and acceleration. As a result, the 2-D linear Kalman filter is used to determine the position of drone and clean the IMU data.

At the initial state of the 2-D Kalman filter the observer takes into account the initial position and velocity of the UAV. Usually the initial position and velocity of the UAV is taken as zero if it is considered that the UAV just stats from the resting position.

$$Initial\ State\ Matrix = \begin{pmatrix} x_0 \\ v_0 \end{pmatrix}$$

Equation 6: Initial State Matrix

The predicted state matrix is calculated using the initial state matrix, adaptive matrices, control matrix along with observation error.

$$X_{kp} = AX_{k-1} + BU_k + w_k$$

Equation 7: Predicted State Matrix

The initial process covariance matrix is calculated using variance and covariance of the velocity and position which are in other words considered as Process errors.

$$P_{k-1} = \begin{pmatrix} \sigma_x^2 & \sigma_x\sigma_v \\ \sigma_x\sigma_v & \sigma_V^2 \end{pmatrix}$$

Equation 8: Initial Process Covariance Matrix

From there using the adaptive matrix, initial process covariance matrix, transpose of that adaptive matrix & along with process noise covariance matrix the predicted process covariance matrix can be found.

$$P_{kp} = AP_{k-1} + Q_k$$

Equation 9: Predicted Process Covariance Matrix

The Kalman Gain is calculated using the predicted process covariance matrix, adaptive matrix, transpose of that adaptive matrix & sensor noise covariance matrix.

$$KG = \frac{P_{kp}H^T}{HP_{kp}H^T + R}$$

Equation 10: Kalman Gain

From the new sensor measurement along with adaptive matrix and considering measurement noise a new observation matrix is formed.

$$Y_k = CY_{km} + Z_m$$

Equation 11: New Observation

Using the Kalman gain and new observation matrix along with adaptive matrix and identity matrix a new predicted state matrix and a predicted process covariance matrix are formed. Then the new or current state matrix and current predicted process covariance matrix are used as the previous state matrix and previous predicted process covariance matrix in lieu of initial state matrix & initial process covariance matrix respectively.

$$X_k = X_{kp} + KG\left[Y_k - HX_{kp}\right]$$

Equation 12: Current State Matrix

$$P_k = [I - (KG \times H)]P_{kp}$$

Equation 13: Current Process Covariance Matrix

$$Current\ State\ Matrix: x_k \implies Previous\ State\ Matrix: x_{k-1}$$

$$Current\ Process\ Covariance\ Matrix: P_k \implies Previous\ Process\ Covariance\ Matrix: P_{k-1}$$

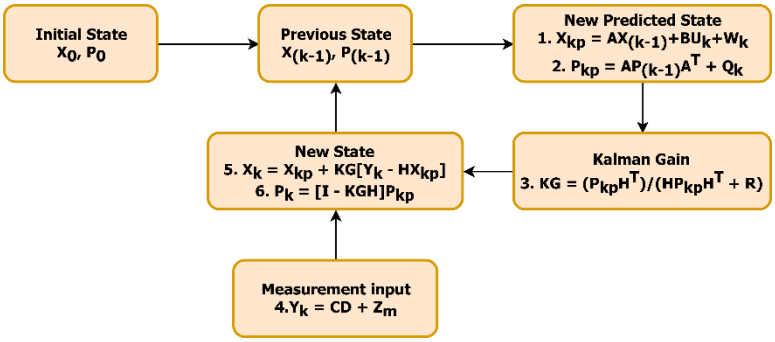


Figure 42: 2-D kalman Filter algorithm

$\boldsymbol{X_{kp}}$: *Predicted State Matrix*
$\boldsymbol{X_k}$: *Current State Matrix*
$\boldsymbol{X_{(k-1)}}$: *Previous State Matrix*
$\boldsymbol{U_k}$: *Control Variable Matrix*
$\boldsymbol{W_k}$: *Noise in the Process (Observation Error)*
$\boldsymbol{P_{kp}}$: *Predicted State Covariance Matrix*
$\boldsymbol{P_k}$: *Current State Covariance Matrix*
$\boldsymbol{P_{(k-1)}}$: *Previous State Covariance Matrix*
$\boldsymbol{KG}$: *Kalman Gain*
$\boldsymbol{Y_k}$: *New Observation*
$\boldsymbol{D}$: *Data Received*
$\boldsymbol{Z_m}$: *Measurement Noise*
$\boldsymbol{Q}$: *Process Noise Covariance Matrix*
$\boldsymbol{R}$: *Sensor Noise Covariance Matrix*
$\boldsymbol{I}$: *Identity Matrix*
$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{H}$: *Adaptive Matrices*
$\boldsymbol{H^T}$: *Transpose of Adaptive Matrix*

4A.3.2: Complementary Filter

[16] The AHRS uses the complementary filter. Where Gyroscope derived roll is passed through a high pass filter and the Accelerometer derived roll is passed through a low pass filter. Similarly, another complementary filter is used for fusing Accelerometer & Gyroscope derived pitch angles. For fusing the yaw angles of the Magnetometer and gyroscope another individual Kalman filter is used.

$$y[n] = (1 - \alpha)x[n] + \alpha y[n - 1]$$

Equation 14: Low-pass filter

Angles obtained from accelerometers are used here

$x[n]$: *represents the pitch/roll that you get from the accelerometer*

$y[n]$: *represents the filtered final pitch/roll which is fed to the complementary filter equation*

$$y[n] = (1-\alpha)y[n-1] + (1-\alpha)(x[n] - x[n-1])$$

Equation 15: High-pass filter

Angles obtained from gyroscopes are used here

$x[n]: represents\ the\ pitch/roll/yaw\ from\ the\ gyroscope$

$y[n]: represents\ the\ filtered\ pitch/roll/yaw\ which\ is\ fed\ to\ the\ complementary\ filter\ equation$

$\boldsymbol{n} = Current\ Sample\ Indicator$

$\boldsymbol{\alpha}\ represents\ the\ end\ of\ accelerometer\ reading\ and\ beginning\ of\ the\ gyroscope\ reading.$
$It\ also\ represents\ how\ much\ dependency\ has\ been\ put\ on\ to\ the\ present\ and\ the\ previous\ data$

$$\alpha = \frac{\tau}{\tau + dt}$$

$\tau = represents\ the\ time\ constant\ which\ tells\ how\ fast\ the\ reading\ will\ respond$

$fs = 1/dt = Sampling\ frequency.$

$$angle = (1-\alpha)\left(angle + \int (gyroanglefromHPF)dt\right) + \alpha(accelerometeranglefromLPF)$$

Equation 16: Complementary Filter

4A.4: AHRS & GPS Data Fusion

In the modern days the Global Positioning System (GPS) and Inertial Navigation System (INS) are one of the most commonly used navigation devices. The GPS provides excellent navigational performance in a low update rate with a good precision where there is a good satellite signal available. However, the GPS's performance deteriorates drastically and even completely losing its capability where there is no satellite connectivity. On the other hand, the INS provides continuous navigation information in a high frequency which is not influenced by any other external sources. But the INS is formed by using IMU which itself is very much prone to Additive White Gaussian Noise (AWGN) as a result the IMU data drifts with time. So, to solve the problem the high accuracy but slow update rate GPS data is fused with high update rate but low accuracy IMU data to provide the actual correct data.

For the IMU where the AHRS algorithm is implemented, the roll(ϕ), pitch(θ) & yaw(Ψ) , the absolute NED acceleration along with the GPS data(Latitude, Longitude, Altitude) are combine to gather to form the rotation matrix. Which is then converted to Earth Centered Earth Fixed (ECEF) coordinate system (by using python NavPy library's ned2ecef function). Thus, providing the position values. From which the velocity can be found via time period of the IMU. Then again, the positions and velocities in the ECEF are fed to a 2-D Kalman filter to remove any further noise in them.

5: Command Center (CC)

The Command Center is a bi-directional interpreted command-based API-like communications interface that connects the EnvironMetric's front end (Dashboard) with its backend (RSA). It is purpose designed to provide an abstraction for the underlying hardware initialization/operation, low-level serial communication, multi-stage digital signal processing, encoding/decoding, compression/decompression & low-level wireless communication. The command language that's implemented by the Command Center is designed to have a close resemblance with Unix bash. It is a standalone API-like module, complete with its own interpreter that implements standard command language constructs like Named-Parameters, Piping & Aliasing. In addition to supporting key features like acquiring data & executing/querying sensor calibrations, the Command Center is also intended to be leveraged by technical operators to configure, debug, diagnose & benchmark the RSA.

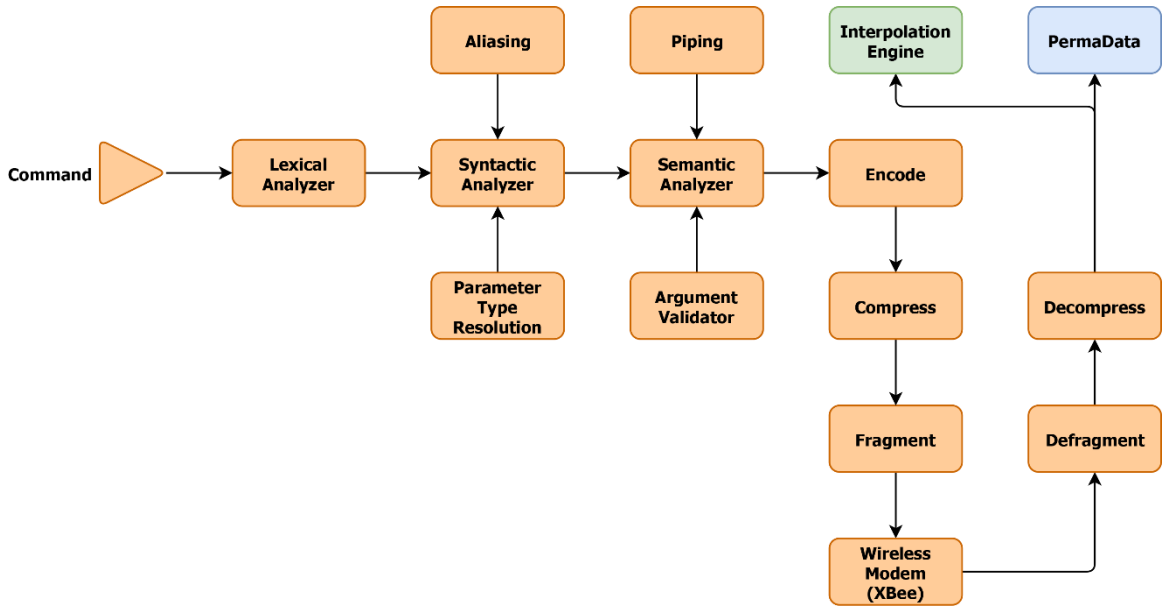


Figure 43: System-level block diagram of the Command Center (CC) architecture

5.1: Command Center Interpreter

In computer science, an interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:

1. Parse the source code and perform its behavior directly.
2. Translate source code into some efficient intermediate representation and immediately execute this.
3. Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

The interpreter that's implemented for the EnvironMetric's Command Center is designed to interpret a bash-like command language. A command language is a language used for job control in computing. It is a domain-specific and interpreted language; common examples of a command language are shell or batch programming languages. These languages can be used directly at the command line but can also automate tasks that would normally be performed manually at the command line. They share this domain - lightweight automation - with scripting languages, though a command language usually has stronger coupling to the underlying operating system. Command languages often have either very simple grammars or syntaxes very close to natural language, to shallow the learning curve, as with many other domain-specific languages. The Command Center API is composed of two types of commands -

Abstract Commands (ABS) are ambiguous in their functionality and are merely precursors to the concrete commands that are derived through annexing multivariate parameters. Abstract commands are not stand-alone commands and therefore cannot be transmitted to the RSA's Basic Input/Output System (RSA BIOS)

to perform precise operations. **Concrete Commands** are derived from abstract commands and are functionally unambiguous in their operation. They are designed to carry-out a precise and well-defined operation relating to the EnvironMetric System that may involve the frontend (DBD Command) or the backend (RSA Command). RSA Commands require a backend be connected to the frontend as they need to intercommunicate to execute the requested operation. Additionally, each RSA command has a unique Task Identifier (TID) assigned to them by the RSA that may be used to track and refer the command instance within the same session. DBD Commands, on the other hand, execute entirely on the Dashboard and thus don't require the frontend be connected to a backend.

The functional behavior of select commands can be modified to fit the operator's operational requirements by assigning values to its parameters through keyword (named) arguments. The command, whose functional behavior is being modified through manipulating its parameters, is referred to as the Carrier Command. The parameters can be one of four types.

Multivariate Parameters, when appended to abstract commands, manipulate their trailing parametric syntax to derive concrete commands; this changes the carrier command's functional paradigm. **Univariate Parameters** are essentially a vector of position independent flag parameters each of which can assume one of a set of values to enable/disable key components of the carrier command's functional behavior. Univariate parameters are combinatoric by nature, offering an N Choose R scenario; here, N is the number of options to choose from and R is the length of the argument vector. **Flag Parameters** are position dependent binary switches that can either enable or disable a key component of the carrier command's functional behavior. The presence of a specific flag enables the corresponding comportment while its absence disables it. **Assignment Parameters** always appear in key-value pairs. They are used to assign a literal (value) to a variable (key) through the carrier command.

The Principal Commands, usually a verb, describes an action that is to be performed. Some of these commands are vague in their operation (abstract commands) and require annexing additional components, usually one or more nouns and adverbs, called (Multivariate) parameters to derive concrete commands that are well-defined in terms of the operation they perform. Additionally, the operational behavior of certain Concrete Commands may be modified by (Univariate or Flag or Assignment) parameters. These parameters are said to be modifying the operational behavior of their respective Carrier Commands. To describe these functional requirements, the Command Center makes use of a standard notation for expressing the syntax for its commands. Using this notation, a single parameter may be described as,

-ParameterType::ParameterName

This representation describes a parameter that's mandatory for the operator to clearly define through a single argument when the command is invoked. ParameterType refers to the type of parameter - one of multivariate, univariate, flag or assignment. ParameterName is the operator-friendly identifier unique to each parameter of a carrier command for use when it is invoked with keyword (named) arguments.

The Command Center's architecture requires some parameters to be optional i.e. they may or may not be provided by the operator when the command is invoked. Implying that the carrier command is expected to modify the precise action it performs based purely on the parameter's absence/presence. These parameters are described as,

[-ParameterType::ParameterName]

The syntax notation also includes a standard for expressing parameter vectors. That is, parameters which may be provided as a list containing one or more components. Such parameters are described as,

-[ParameterType::ParameterName]

Additionally, these two notations can be combined to produce,

[-[ParameterType::ParameterName]]

which represents an optional vector of parameters.

Having multiple parameters for the same carrier command introduces the concept of parameter coupling whereby providing the argument to one parameter mandates the argument of a second parameter be provided. This is implicitly achieved in the case where both parameters are mandatory. In the case where the concerned parameters are optional, they can be expressed as,

[-ParameterType_A::ParameterName_A -ParameterType_B::ParameterName_B]

This denotes that the two parameters are optional but if either is provided with an argument, so must the other.

The Command Center interpreter interprets each command as it receives it on a character-by-character, word-by-word and line-byline basis. It consists of three distinct sub systems - the **Lexical Analyzer** to validate the individual characters, the **Syntactic Analyzer** to validate the words that are formed from the characters and the **Semantic Analyzer** to validate the sentence (command) formed from the words. If the command passes the validation checks in all three of the sub systems, it is further processed and encoded to be executed locally or remotely.

5.1.1: Lexical Analyzer

Figure 44: Lexical Analyzer block diagram

The Lexical Analyzer is the simplest of the three sub systems of the Command Center interpreter; with the principle function of validating the character tokens that make up the unprocessed command input. It is also tasked with unifying all the characters to adhere to a single case (uppercase) to make the command language case insensitive. This was a design decision made by the system architect to favor ease-of-use over expressive-power. With its current specification, the language has no support for user defined variables. With plans to revise the specification to allow for custom user defined variables to store tensors, the case-insensitive aspect of the language may be deprecated to gain more expressive-power.

The Current specification of the EnvironMetric command language considers the following ASCII characters as valid.

Bin	Oct	Dec	Hex	Glyph		
				‘63	‘65	‘67
010 0000	040	32	20	space		
010 0010	042	34	22	"		
010 1101	055	45	2D	-		
010 1110	056	46	2E	.		
011 0000	060	48	30	0		
011 0001	061	49	31	1		
011 0010	062	50	32	2		
011 0011	063	51	33	3		
011 0100	064	52	34	4		
011 0101	065	53	35	5		
011 0110	066	54	36	6		
011 0111	067	55	37	7		
011 1000	070	56	38	8		
011 1001	071	57	39	9		
100 0001	101	65	41	A		
100 0010	102	66	42	B		
100 0011	103	67	43	C		
100 0100	104	68	44	D		
100 0101	105	69	45	E		
100 0110	106	70	46	F		
100 0111	107	71	47	G		
100 1000	110	72	48	H		
100 1001	111	73	49	I		
100 1010	112	74	4A	J		
100 1011	113	75	4B	K		
100 1100	114	76	4C	L		
100 1101	115	77	4D	M		
100 1110	116	78	4E	N		
100 1111	117	79	4F	O		
101 0000	120	80	50	P		
101 0001	121	81	51	Q		
101 0010	122	82	52	R		
101 0011	123	83	53	S		
101 0100	124	84	54	T		
101 0101	125	85	55	U		
101 0110	126	86	56	V		
101 0111	127	87	57	W		
101 1000	130	88	58	X		
101 1001	131	89	59	Y		
101 1010	132	90	5A	Z		

Bin	Oct	Dec	Hex	Glyph		
				‘63	‘65	‘67
110 0001	141	97	61		a	
110 0010	142	98	62		b	
110 0011	143	99	63		c	
110 0100	144	100	64		d	
110 0101	145	101	65		e	
110 0110	146	102	66		f	
110 0111	147	103	67		g	
110 1000	150	104	68		h	
110 1001	151	105	69		i	
110 1010	152	106	6A		j	
110 1011	153	107	6B		k	
110 1100	154	108	6C		l	
110 1101	155	109	6D		m	
110 1110	156	110	6E		n	
110 1111	157	111	6F		o	
111 0000	160	112	70		p	
111 0001	161	113	71		q	
111 0010	162	114	72		r	
111 0011	163	115	73		s	
111 0100	164	116	74		t	
111 0101	165	117	75		u	
111 0110	166	118	76		v	
111 0111	167	119	77		w	
111 1000	170	120	78		x	
111 1001	171	121	79		y	
111 1010	172	122	7A		z	
111 1100	174	124	7C	ACK	¬	|

Table 53: List of standard ASCII characters allowed in the EnvironMetric command language

5.1.2: Syntactic Analyzer

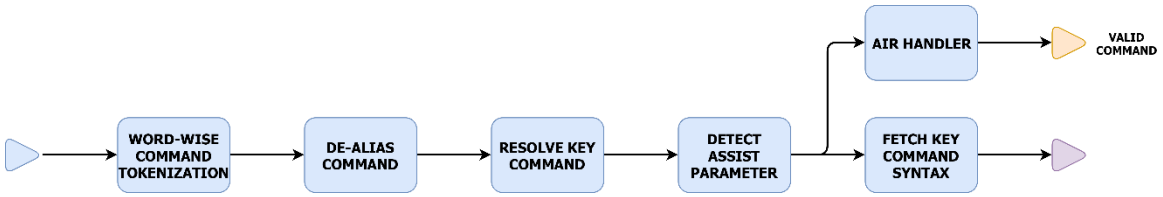


Figure 45: Syntactic Analyzer block diagram

The principle function of the Syntactic Analyzer is to validate the raw command input on a word-by-word basis. To do this, the sub system first fragments the command based on a whitespace separator. Then each word token is de-aliased by using a hashmap where the key is an alias and the value is its corresponding keyword. The key command is finally resolved by using a looping construct to repeatedly annex the i^{th} token (array index) starting from $i = 0$ and shrinking the token array from the head. The key command is typically a verb or verb-noun tuple that appears at the beginning of the command input and is used to determine the syntax signature for the command against which the user provided named parameter-argument tuples for the command will be checked. The syntactic analyzer then checks for the -Assist flag to determine if the user requested to query the Assist Information Repository (AIR) system; if so, the key command along with the assist flag is returned by the command validator and then the command handler transfers control over to the AIR system to handle the requested query. On the other hand, if no -Assist flag is detected, the syntax signature for the resolved key command is retrieved from another hashmap to be passed on to the Semantic Analyzer for further validation.

Command	Type	Summary
Abort	Abstract Command	An abstract command to irrevocably terminate hosted operations
Abort-Task	Remote Sensor Array Command	Irrevocably terminates the execution of a task from the execution queue
Acquire	Abstract Command	An abstract command to procure the various tensors that are generated by the RSA BIOS
Acquire-Data	Remote Sensor Array Command	Procures the data tensors that are generated by the RSA BIOS
Assist	Dashboard Command	Queries the AIR system to retrieve information about operating the EnvironMetric Command Center
Back	Dashboard Command	Switches operation from the console environment back to the GUI environment
Clear	Dashboard Command	Clears the Command Center console buffer
Connect	Dashboard Command	Establishes a connection between the Frontend and the Backend
Disconnect	Dashboard Command	Terminates the operational connection between the Frontend and the Backend
Exit	Dashboard Command	Terminates the Command Center and Dashboard applications
Halt	Abstract Command	An abstract command to indefinitely suspend hosted operations
Halt-Task	Remote Sensor Array Command	Indefinitely suspends the execution of a task from the execution queue
Ping	Abstract Command	An abstract command to query the various backend components for status and information
Ping-Hardware	Remote Sensor Array Command	Queries the various Backend hardware components for status and information
Reboot	Remote Sensor Array Command	Invokes a full system reboot on the RSA
Resume	Abstract Command	An abstract command to immediately reinstate previously suspended operations
Resume-Task	Remote Sensor Array Command	Immediately reinstates a previously suspended task onto the execution queue
Shutdown	Remote Sensor Array Command	Invokes a hard system shutdown on the RSA

Table 54: List of key commands supported by the current version of the EnvironMetric command language

5.1.3: Semantic Analyzer

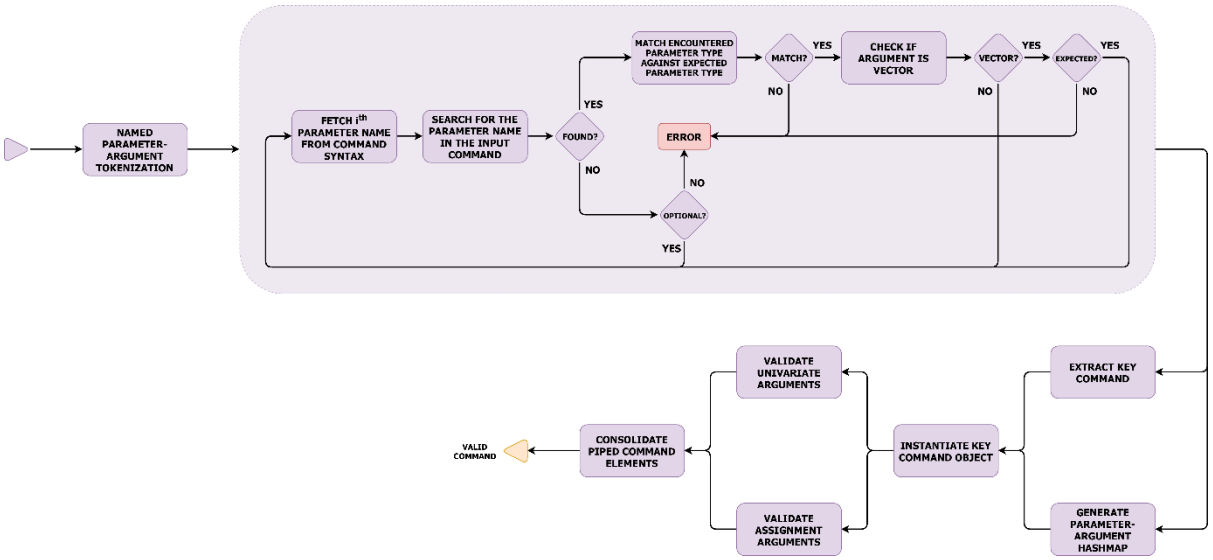


Figure 46: Semantic Analyzer block diagram

Because the semantics can vary from command to command, the semantic analyzer is split into two parts. The first part is generic and applies to any and all input commands; the second part is specialized to meet the semantic requirement of each command and is implemented within the encoder routine of each command class. Most of the semantic checking i.e. parameter validation is performed in the first stage, with the second stage almost exclusively being used for argument validation purposes.

The remaining tokens, that are not part of the key command, are joined using whitespace to get the original form of the user provided named arguments and passed on to the first stage of the semantic analyzer. This form is then split using the '-' character and each fragment is further split using the whitespace character; this sequence of splits, referred to as the “named parameter-argument tokenization” in the figure, results in a list of tuples being generated where the parameter name along with each of its arguments are grouped together. Then each of the provided parameter is checked against the expected parameter for type errors. This is also when the command is checked for missing mandatory parameters and argument vectors.

If a command meets all the checks in the generic stage of the semantic analyzer, it is passed on to the specialized semantic analyzer that’s specific to each command. The generic semantic analyzer outputs two principle components to represent the command that was originally input by the operator; the key command that was resolved in the syntactic analyzer by annexing the leading word tokens of the command and a parameter-argument hashmap that’s formed within the generic sematic analyzer by the named parameter-argument tokenization process. The specialized semantic analyzer resides in the encoder of each command object. So, a command object that corresponds to the operator requested key command is first instantiated using a switch construct with the parameter-argument hashmap passed on as arguments to the object’s constructor. The semantic analyzer within the command object handles the special aspects of semantic analysis that’s specific to the key commands; this includes but is not limited to value checking for univariate parameters and assignment parameters. This is also the stage where any piped constructs of the command are consolidated.

Command	Syntax
Abort	-Multivariate::OperationType [-Flag::Assist]
Abort-Task	-Assignment::PID [-Flag::Assist]
Acquire	-Multivariate::TensorType [-Flag::Assist]
Acquire-Data	[-[Univariate::DataComponents]] [-Assignment::Frequency -Assignment::Time] [-Flag::Assist]
Assist	
Back	[-Flag::Assist]
Clear	[-Flag::Flush] [-Flag::Assist]
Connect	-Univariate::ModemType [-Flag::Assist]
Disconnect	[-Flag::Assist]
Exit	[-Flag::Assist]
Halt	-Multivariate::OperationType [-Flag::Assist]
Halt-Task	-Assignment::PID [-Flag::Assist]
Ping	-Multivariate::Component [-Flag::Assist]
Ping-Hardware	-Univariate::Filter [-Flag::Assist]
Reboot	[-Flag::Override] [-Flag::Assist]
Resume	-Multivariate::OperationType [-Flag::Assist]
Resume-Task	-Assignment::PID [-Flag::Assist]
Shutdown	[-Flag::Override] [-Flag::Assist]

Table 55: Syntax for the key commands supported by the current version of the EnvironMetric command language

5.2: Encoder

Each of the key commands is programmatically represented as a concrete class of its own. Each command class is derived from a common base class that defines the standard interface for the command classes. One such interface is the Encode method which is used to obtain an intermediate binary digit representation of the key command along with all its parameters and arguments that will be transmitted to and executed by the RSA. This intermediate representation resembles machine code and is easy for the RSA to process and execute. Additionally, this representation is much more compressed than the original form and makes the transmission much faster.

The key command is encoded with a simple 8-bit binary number, below is a table listing all the key command encodings with the absent binary numbers not being mapped to any key command yet.

Binary Encoding	Key Command
00000000	Assist
00000001	Back
00000010	Exit
00000011	Clear
00000100	Connect
00000101	Disconnect
00000110	Acquire
00000111	Ping
00001000	Abort
00001001	Halt
00001010	Resume

Binary Encoding	Key Command
Shutdown	10000000
Reboot	10000001
Ping-Hardware	10000010
Ping-Software*	10000011
Acquire-Data	10000100
Abort-Task	10000101
Halt-Task	10000110
Resume-Task	10000111

Table 56: List of binary encodings for all the key commands implemented in the current version of the EnvironMetric command language

*The Ping-Software command was not ultimately implemented in the first public demonstration of the language because of time constraints. It was originally intended to be used to ping the various software components of the Asynchronous Acquisition Protocol running on the Remote Sensor Array's Basic Input/Output System.

The parameters and arguments for the key commands are encoded in a variety of ways depending on their characteristic properties. For example, a single Flag parameter is represented by a single bit that can be either 1 or 0 to represent the engaged or disengaged states respectively. Extending this idea, an array of

Flags that can be engaged and disengage independently of each other is encoded using an array of bits - one for each flag; a similar technique is used for Univariate vectors because they exhibit the same behavior as Flags in terms of combinatorics. On the other hand, a Univariate parameter is encoded using a binary number similar to the key command encoding scheme; because it is only allowed to take on one of n values, a binary number encoding results in a much more compressed representation than would result from a bit array encoding which makes it more efficient for transmission. Assignment parameters are encoded with a single bit to indicate if their value was provided in the operator's command input or not while the assignment value itself is compressed and transmitted in a separate data packet.

5.3: Compression

The process that's referred to as compression is this section is, in reality, one that involves encoding more than compression as the internal programmatic representations of integers, floats and strings don't have any superfluous bits and are therefore as compressed as possible. This implies that it is not possible to losslessly compress them without applying probabilistic models and using probabilistic models to compress random numbers is not only outside the scope of this project due to time constraints but is likely impossible! With that being said, data compressions primarily take place during the transmission of values that are passed on as the arguments to assignment parameters. The problem with transmitting integers and floats is that their conventional (human-readable) string representations can sometimes reach sizes that are much greater in bit length than their internal programmatic representations. Take the number 18,446,744,073,709,551,615 for example, it is programmatically represented by a 64-bit unsigned integer in memory while its conventional string representation takes 160 bits of information - 8 bits for each of the 20 numeric characters. The XBee API that is used as the command center's physical modem is only capable of transmitting bit arrays as a string of extended ASCII characters; so, an encoding scheme had to be devised that would allow the compression of such data packets to minimize transmission overheads while still preserving all the information in the process.

The initial plan for compressing such data for transmission was to convert the internal 64 bits of data, representing the number, into a string of 8 extended ASCII characters - each character representing 8 bits of the 64 bit array in sequence. Implementing this compression scheme would have essentially zeroed the data transmission overhead but this scheme can produce any of the 256 distinct extended ASCII characters, even the ASCII control characters, during the encoding process depending on the integer value being encoded. To further complicate matters, the ZigBee protocol that's utilized by the XBee modems use ASCII control characters during data transmission for internal system signaling purposes. This meant that those control characters could not be transmitted by the ZigBee protocol as plain data as the system would interpret them as being special control signals.

The encoding scheme was updated to compress the data using hexadecimal numbers. The 64 bits were represented using 16 hexadecimal digits which was then converted to a human-readable string. This scheme uses 128 bits to encode the 160 bit long numbers. This scheme guaranteed that none of the ASCII control characters would appear in the encoded form thus maintaining compatibility with the ZigBee protocol. This data packet would then be transmitted over the XBee modem network using the little-endian format and be decompressed on the receiving side by converting the hexadecimal digits into their binary form and performing bitwise operations.

5.4: Fragmentation

The ZigBee protocol has an inherent limitation that only allows 256-byte sized data packets to be transmitted at a time, but the Command Center application requires transmitting large payloads containing complex data structures as response to certain commands. So, the payloads that exceed that limit are split into 256-byte chunks and transmitted as separate data packets to be concatenated on the receiving side; this process is referred to as fragmentation. But during the defragmentation phase, the receiving side of the communication channel also needs to be able to distinguish the data packets that belong to the same payload from the ones that belong to other payloads as a single payload might potentially end up being transmitted over one or more data packet transactions in this scheme. This is achieved by introducing another layer of handshakes on top of the ZigBee protocol that mark the start and end of each payload transmission that may constitute multiple data packet transactions.

5.5: Modem

The Command Center wirelessly connects the backend and the frontend of the EnvironMetric system; for this purpose, the XBee 3 Pro modules from Digi International was utilized. These modules use the IEEE 802.15.4 networking protocol for fast point-to-multipoint or peer-to-peer networking. They are designed for high-throughput applications requiring low latency and predictable communication timing. The XBee's can operate either in a transparent data mode or in a packet-based application programming interface (API) mode. In the transparent mode, data coming into the Data IN (DIN) pin is directly transmitted over-the-air to the intended receiving radios without any modification. Incoming packets can either be directly addressed to one target (point-to-point) or broadcast to multiple targets (star). This mode is primarily used in instances where an existing protocol cannot tolerate changes to the data format. AT commands are used to control the radio's settings. In API mode the data are wrapped in a packet structure that allows for addressing, parameter setting and packet delivery feedback, including remote sensing and control of digital I/O and analog input pins.

Figure 47: Digi XBee 3 PRO Zigbee 3.0 (bottom-up view)

Figure 48: Digi XBee 3 PRO Zigbee 3.0 (top-down view) with scale

PERFORMANCE	
Transceiver Chipset	Silicon Labs EFR32MG SoC
Data Rate	RF 250 Kbps, serial up to 1 Mbps
Indoor/Urban Range*	Up to 90 m (300 ft)
Outdoor/RF Line-of-Sight Range*	Up to 3200 m (2 miles)
Transmit Power	+19 dBm
Receiver Sensitivity (1% Per)	-103 dBm Normal Mode
FEATURES	
Serial Data Interface	UART, SPI, I²C
Configuration Method	API or AT commands, local or over-the-air (OTA)
Frequency Band	ISM 2.4 GHz
Form Factor	Micro, through-hole, surface mount
Interference Immunity	DSSS (Direct Sequence Spread Spectrum)
ADC Inputs	(4) 10-bit ADC inputs
Digital I/O	15
Antenna Options	Through-hole: PCB Antenna, U.FL Connector, RPSMA Connector SMT: RF Pad, PCB Antenna, or U.FL Connector Micro: U.FL Antenna, RF Pad, Chip Antenna
Operating Temperature	-40º C to 85º C (-40º F to 185º F)
Dimensions (L x W x H)	Through-hole: 2.438 x 2.761 cm (0.960 x 1.087 in) SMT: 2.199 x 3.4 x 0.305 cm (0.866 x 1.33 x 0.120 in) Micro: 13 x 19 x 2 mm (0.533 x 0.76 x 0.087 in)
PROGRAMMABILITY	
Memory	1 MB / 128 KB RAM (32KB are available for MicroPython)
NETWORKING AND SECURITY	
Protocol	Zigbee® 3.0
Encryption	128/256 bit AES
Reliable Packet Delivery	Retries/acknowledgements
IDs	PAN ID and addresses, cluster IDs and endpoints (optional)
Channels	16 channels
POWER REQUIREMENTS	
Supply Voltage	2.1 to 3.6 V
Transmit Current	135 mA @ 19 dBm
Receive Current	17 mA
Power-Down Current	2 micro Amp @ 25º C (77º F)
REGULATORY APPROVALS	
FCC, IC (North America)	Yes
ETSI (Europe)	No
RCM (Australia)	Yes
ANATEL (Brazil)	Yes
TELECK MIC (Japan)	No
KCC (South Korea)	No

Table 57: Digi XBee 3 PRO Zigbee 3.0 Specifications

*Range figure estimates are based on free-air terrain with limited sources of interference. Actual range will vary based on transmitting power, orientation of transmitter and receiver, height of transmitting antenna, height of receiving antenna, weather conditions, interference sources in the area, and terrain between receiver and transmitter, including indoor and outdoor structures such as walls, trees, buildings, hills, and mountains.

6: Dashboard (DBD)

Dashboard is the nerve center of the EnvironMetric system which makes use of a straightforward GUI to acquire, visualize & archive the data from the RSA. The Dashboard is essentially another layer of abstraction on top of the Command Center. It has three key components for the purposes of data acquisition, visualization & archival - Vernier, Explorer & Diagnostics respectively. The Vernier uses the underlying Command Center API to provide an easy-to-use GUI for collecting environmental data & uploading them to PermaData for archiving. The Explorer downloads data from PermaData to visualize them through scatterplots, heatmaps & line graphs. The Diagnostics is used to receive status updates from the RSA in order to gauge the state of the various software & hardware components comprising the RSA.

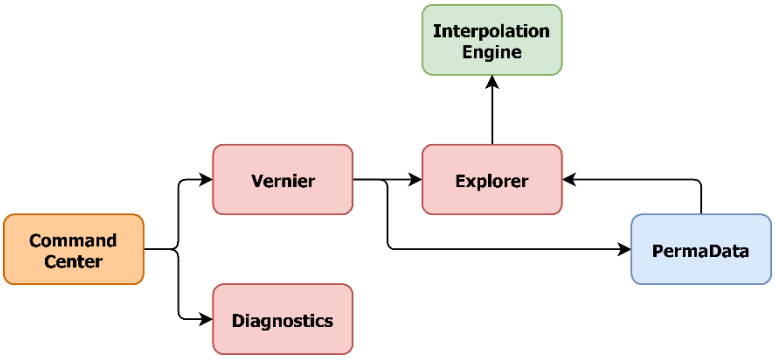


Figure 49: System-level block diagram of the Dashboard (DBD) architecture

There are two types of interfaces the Command Line Interface (CLI) and the Graphical User Interface (GUI). The Command Center uses the CLI that provides interface to collect data and store it. User need to type the commands and set the parameters. The Dashboard provides the Graphical Interface to the users. The Dashboard has 3 features. Using these features a user can acquire data, visualize it and archive the data to the database storage. This has been implemented for the non-technical people so that they can use our system easily. They need not type any commands to acquire data or to visualize it or even archive the data to the database. This dashboard has been implemented using C# programming language. As this has been implemented using C# so this dashboard can only be run in the windows platform. We have used Visual Studio as the framework to implement this dashboard. We can move to the Command Console from the Dashboard by pressing on the Command Console button.

6.1 Vernier

This Vernier functionality has been used for the data acquisition. This Vernier uses the underlying command center API to collect the environmental data that's provides an easy-to-use GUI. Vernier provides 2 types of functionalities. A user can collect the data on spatial paradigm and also on the temporal paradigm.

In the spatial paradigm, the user will receive data from RSA with respect to space. That means, the RSA will be moving from one place to another and collect the data. In that case, the RSA will be attached to the Hexa-copter Drone and the drone will fly from one place to another and send the data to the user. In this paradigm a user can set a certain parameter. The user can select the dimensions. That is a user can select two-dimensional paradigm or can select the three-dimensional paradigm. Two-dimensional paradigm provides the data of a certain latitude and longitude. Three-dimensional paradigm provides the data of a certain latitude, longitude and altitude. So, the user can set the basis which may be 2 dimensional or 3 dimensional. A user can also select the criteria. The criteria can be pressure, temperature, humidity, CO2, MOX, TVOC, and Total Volatile etc. A user can select all of them or can select some of them based on his/her willingness. The user can set the frequency or sampling resolution. Another functionality provided by the Vernier is the X-Ceiling and the Y-Ceiling. This let the user to limit the 2D space. For example, if a user selects 200 and 200 in meter unit then the Vernier create a 200 by 200 two-dimensional space. This feature would be used if the user knows the region from where the data is being collected. The user can see the properties of the received data in this section.

In the Temporal paradigm, the user will receive data from the RSA with respect to time. In this case the RSA will not be moving from one place to another. The RSA will be kept in one place for a certain period of time and the RSA will send the data to the user. A user can set the parameters to collect the data from the temporal paradigm. The users can set the metrics. User can select temperature, humidity, pressure, CO2, MOX, TVOC etc. the user can also specify the sampling resolution. In this case the sampling resolution is the

time. User can set time from 1second to maximum 1week. For example, if a user selects 1hour then he/she will be receiving data from the RSA for 1 hour. The user can also select the sample limit. How many data does he/she need? The user can also set the time limit in this temporal paradigm. Like the spatial paradigm a user can also see the properties of the data received from the RSA.

After collecting the data, the user can store the data to the permanent database. User needs to provide the description, the base name, base latitude and base longitude of the acquired data. After that the data will be saved to BigQuery permanent data storage.

6.2 Explorer

Explorer provides the data visualization functionality. To analyze any scientific experiment even in the environmental science data visualization plays a very vital role. We can show data in many different ways like heatmap, line graph, pie chart, bar chart etc. As our system is a complete system so there must be a data visualization section. This explorer provides that functionality. In this section the data will be downloaded from the permanent data storage. The user can select the date which dynamically filter the received data. The user can also set the range of date. If the user set the range of date then he/she will see the filtered data that is within this date range. The user can search for any particular data and if the user do so the list will be dynamically filtered and show the searched data. The user can select the particular data and can see different types of data visualization graph. A user can also see the real time data plotting visualization.

6.3 Diagnostics

In our system there are two types of sensors. One type of sensor that collects the environmental data and another type of sensors those collect the inertial measurement data. In our system we are using BNO055 to collect the inertial data. This BNO055 provides us the acceleration data, magnetometer data, Angular Velocity. Rest of the sensors provide the environmental data. For example, BME280 provide the pressure, temperature and humidity. CCS811 provides the eCO_2, MOX, and TVOC. In this section, a user can see the raw data. This section shows the data in every stage of the data processing. In this section, a user can see the raw data, the filtered data or the preprocessed data. There is a calibrate button that calibrates the sensors. The user can calibrate the inertial measurement sensors and can also calibrate the environmental monitoring sensors.

7. PermaData (PD)

PermaData, short for Permanent Data Storage, is a tensor-based data archive interface currently implemented by using Google's BigQuery Service. A key-value tensor-based data storage schema was chosen because of the flexibility it offers in-terms adding/removing measurement metrics during runtime, albeit at the cost of query efficiency.

The main purposes of our project are not only collecting the environmental data but also archiving those collected data. To archive the data, we need a permanent data storage. We have chosen the Google's BigQuery as this can perform SQL query over a huge amount of data. The query speed is faster than any other SQL databases. A user can store a huge amount of data and can also retrieve it from the storage very efficiently. A user can collect temperature, pressure, humidity, eCO_2, TVOC, MOX etc. over a certain period with a given frequency. In each session we can have thousands of data of different types. The normal MySQL database will take more time than the BigQuery to push data or retrieve to/from the storage. As a result, BigQuery is the best option to use as the permanent data storage for our project. This BigQuery is also free service provided by Google.

After collecting the data from the RSA, the user will be asked if s/he wants to store the collected data to PermaData. In the command, there will appear an option after collecting data. If the user wants to archive the data, then s/he type "yes/y" and if he/she does not want to store the data then "no/n" may be typed.

The tensor has a structure that allows it to be extremely flexible. We have made the following structure

$$[label_1{:}value_1\ lable_2{:}value_2\ label_3{:}value_3\ \ldots\ label_n{:}value_n]$$

for example, [temperature:25 humidity:60] is one single tensor that contains 2 types of values temperature and humidity. To separate the components, we first split in the "space" which provides us 2 components again split in the "colon" to separate the temperature label and temperature value. In this process we split the tensor and can produce rows or columns as per the application demands.

7.1: Google BigQuery

BigQuery, Google's serverless, highly scalable enterprise data warehouse, is designed to make data analysts more productive with unmatched price-performance. Because there is no infrastructure to manage, you can focus on uncovering meaningful insights using familiar SQL without the need for a database administrator.

BigQuery separates storage and compute to enable elastic scaling that streamlines capacity planning for data warehouses. BigQuery meets the challenges of real-time analytics by leveraging Google's serverless infrastructure that uses automatic scaling and high-performance streaming ingestion to load data. BigQuery's managed columnar storage, massively parallel execution, and automatic performance optimizations empower users to quickly and simultaneously analyze data from your cloud data lake regardless of the number of users or data size.

BigQuery makes it easy to maintain a strong security and governance foundation. Eliminate data operation burdens with automatic data replication for disaster recovery and high availability of processing for no additional charge. BigQuery offers a 99.9% SLA subject to terms here and adheres to the Privacy Shield Principles.

Key features of Google BigQuery:

- **Serverless**: With serverless data warehousing, Google does all resource provisioning behind the scenes, so you can focus on data and analysis rather than worrying about upgrading, securing, or managing the infrastructure.
- **Real-time analytics**: BigQuery's high-speed streaming insertion API provides a powerful foundation for real-time analytics, making your latest business data immediately available for analysis.
- **Automatic high availability**: BigQuery transparently and automatically provides highly durable, replicated storage in multiple locations and high availability with no extra charge and no additional setup.
- **Standard SQL**: BigQuery supports a standard SQL dialect which is ANSI:2011 compliant, which thereby reduces the need for code rewrites. BigQuery also provides ODBC and JDBC drivers at no cost to ensure your current applications can interact with its powerful engine.

- **Federated query and logical data warehousing**: Through powerful federated query, BigQuery can process external data sources in object storage (Cloud Storage), transactional databases (Cloud Bigtable), or spreadsheets in Drive — all without duplicating data.
- **Storage and compute separation**: With BigQuery's separated storage and compute, you have the option to choose the storage and processing solutions that make sense for your business and control access and costs for each.
- **Automatic backup and easy restore**: BigQuery automatically replicates data and keeps a seven-day history of changes, allowing you to easily restore and compare data from different times.
- **Geospatial data types and functions**: BigQuery GIS brings SQL support for arbitrary points, lines, polygons, and multi-polygons in WKT and GeoJSON format. You can simplify your geospatial analyses, see your location-based data in new ways, or unlock entirely new lines of business.
- **Data transfer service**: The BigQuery Data Transfer Service automatically transfers data from external data sources, like Google Marketing Platform, Google Ads, YouTube, and partner SaaS applications to BigQuery on a scheduled and fully managed basis. Users can also easily transfer data from Teradata and Amazon S3 to BigQuery.
- **Big data ecosystem integration**: With Cloud Dataproc and Cloud Dataflow, BigQuery provides integration with the Apache Big Data ecosystem, allowing existing Hadoop/Spark and Beam workloads to read or write data directly from BigQuery.
- **Petabyte scale**: Get great performance on your data, while knowing you can scale seamlessly to store and analyze petabytes more without having to buy more capacity.
- **Flexible pricing models**: On-demand pricing lets you pay only for the storage and compute that you use. Flat-rate pricing enables high-volume users or enterprises to choose a stable monthly cost. For more information, see BigQuery pricing or cost controls.
- **Data governance and security**: BigQuery makes it easy to maintain strong security with fine-grained identity and access management with Cloud Identity and Access Management, and your data is always encrypted at rest and in transit.
- **Geo expansion**: BigQuery gives you the option of geographic data control (in US, Asia, and European locations), without the headaches of setting up and managing clusters and other computing resources in region.
- **Foundation for AI**: Besides bringing ML to your data with BigQuery ML, integrations with AI Platform and TensorFlow enable you to train powerful models on structured data in minutes with just SQL.
- **Foundation for BI**: BigQuery forms the data warehousing backbone for modern BI solutions and enables seamless data integration, transformation, analysis, visualization, and reporting with tools from Google and our technology partners.
- **Flexible data ingestion**: Load your data from Cloud Storage or stream it into BigQuery at thousands of rows per second to enable real-time analysis of your data. Use familiar data integration tools like Informatica, Talend, and others out of the box.
- **Programmatic interaction**: BigQuery provides a REST API for easy programmatic access and application integration. Client libraries are available in Java, Python, Node.js, C#, Go, Ruby, and PHP. Business users can use Google Apps Script to access BigQuery from Sheets.
- **Rich monitoring and logging with Stackdriver**: BigQuery provides rich monitoring, logging, and alerting through Stackdriver Audit Logs and it can serve as a repository for logs from any application or service using Stackdriver Logging.

7.2: Database schema

In the database we have 3 tables UserMetadata, SessionMetadata, SessionData.

UserMetadata table contains the user basic information. It will contain the UID, FName, LName, Email, PashHash, Clearance. Here UID is the primary key which is unique. UID is the user id of the user. FName is the first name of the user. LName is the last name of the user. Email will contain the email of the user. PashHash will contain the password hash code of the user. Clearance will represent the type of the user. This Clearance controls the access right to the database. For example, the admin may observe the data and can delete also but the normal user may only see the data. They can't delete the data from the database. This type of access will be controlled through the Clearance.

SessionMetadata contains the session information provided by the user. After collecting the data from the sensor stack the user will be asked if he wants to store the data. If the user wants to store then he needs to provide some additional information. This additional information will be stored to the SessionMetadata table. This table contains SID, UID, Name, Description, Date, BaseName, BaseLatitude, BaseLongitude. The SID is the session id which is unique in this table. The UID is user id that is used to join the UserMetadata with the SessionMetadata. This UID is the foreign key of this table. Name contains the session name provided by the user. Description contains the short description of the session like the purpose of taking the data. Date contains the date of taking data. BaseName contains the Base name from where the data is archived and analyzed. BaseLatitude and BaseLongitude contains the coordinates of the Base station. A user may collect data as many as he/she wants. So, one user may have one or many session data.

SessionData is another table that contains the tensors. Many session data comprise the SessionMetadata table. SessionData contains the actual data collected from the sensor stack. SessionData contains SID, TID, Timestamp, Tensor. SID is the Session ID of the SessionMetadata table which is the foreign key of this table. TID is the Tensor ID and this is the primary key of this table. Timestamp contains the Unix time. Tensor contains the data tensor of a certain moment. That means, in a certain Timestamp the Tensor will contain the requested sensor data with the label and value.

7.3: Flexibility vs. Efficiency

We have implemented a special type of database's table schema to push data or retrieve data. We have chosen a tensor-based data storage schema. This tensor contains a bunch of different types of data collected at a certain moment of a session. This tensor-based implementation is more flexible than any other schema. If a user requests only for 2 types of data like temperature and pressure in a session, then this tensor will contain only those 2 types of data. Again, if a user requests for all types of data at a same time then this tensor can hold all of them. Another advantage of the tensor-based implementation is that we can convert this tensor to the row, or we can also convert the tensor to the column as well. We can split the tensor to the row or to the column by performing some string operations. If the query must perform the split operation over a large tensor then it will take more time than the normal query which sometimes makes it inefficient. Despite having this disadvantage, we have used this tensor-based implementation as it provides the most runtime flexibility than any other option.

8: Interpolation Engine (IE)

We have implemented a complete system that not only collects the environmental data but also process the data, archive it then analyzes the data and finally visualizes it. The Interpolation Engine is the part of data post processing and data visualization. We collect the raw environmental data using sensor stack then preprocess it and archive the data to the permanent database. Our Graphical User Interface (GUI), Dashboard has a component named Explorer. A user provides the session ID using the explorer which would download data from the permanent database and feed the data to the Interpolation Engine. The Interpolation Engine takes in the downloaded discrete data (Temperature, humidity, pressure etc.) of a region then runs the interpolation model that eventually provides us the continuous data distribution of that region.

We are using different types of sensors to collect the environmental data. These sensors have hardware level limitations to collect the data. For example, SPS30 can collect at most 1 data per second that is 1Hz. This will not cause any problem when we are collecting the data in temporal paradigm. In the temporal paradigm, the sensor will be kept in one place for a certain period. In the spatial paradigm, the sensor will be moving from one place to another. Due to its acquisition rate limitation, it cannot collect the data of every points on its path. For example, if the hexa-copter flies at 40 km/h and the SPS30 collect 1 data per second then the SPS30 can collect at most 1 data every 11m. As a result, we will get about 100 datapoints if the drone travels 1 km straight line path. To get the continuous datapoints we need this interpolation engine. This interpolation engine uses a universal function approximator to interpolate the missing datapoints by considering the ground truth datapoints.

To convert the discrete data into continuous, we have implemented several mathematical models. Although it seems a simple regression problem to solve and interpolate the acquired data, we were trying to make it more realistic by solving 2D heat equation that can eventually model the actual data distribution in the surrounding area from where the data was taken. This pure mathematical eventually provides us the general solution of the 2D heat equation which is a Fourier series. As initially we will not be given any boundary value so we cannot find the exact value of each point and thus cannot make it continuous.

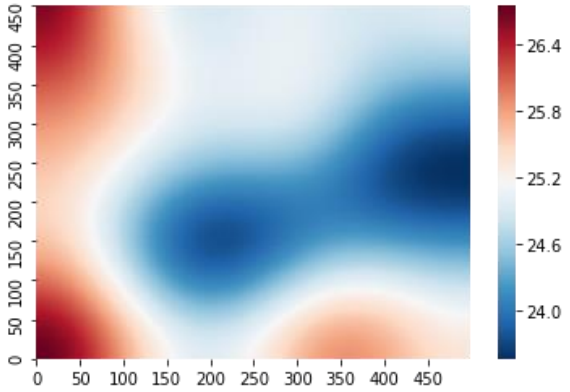


Figure 50: Original Data

Figure 50 depicts the Original picture that is essentially generated by a small function. Our intention is to produce the same output using output using some machine learning algorithms. We have considered some discrete points from this generated output.

We have tried **Multivariate Linear Regression**, **Multivariate Polynomial Regression**, **Dynamic Polynomial Regression**, **Dynamic Combinatoric Polynomial Regression**. Each of these models had some limitations as a result we were not getting the exact approximated output. Among these models the Dynamic Polynomial Regression seems to be performing very well. It was able to interpolate some of the discrete inputs accurately. But this model suffers from the cost function being non-convex that means the gradient descent mostly depends on the random initialization or the starting position. As a result, the optimizer may get stuck in a local optima instead of global optima. The local optima's cost might be very much higher than the cost of the global optima. We calculate the average cost of the given set of (X, Y) values of random sets of weights to find the global optima of this non-convex cost function. After this we pick the set that provides the lowest cost and then perform the Gradient Descent. Although this model was good, but we were looking for some better mathematical model that can interpolate more accurately than any other models.

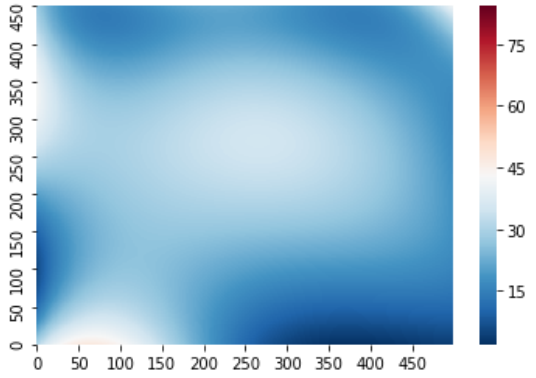


Figure 51: Dynamic Polynomial Regression Reconstruction

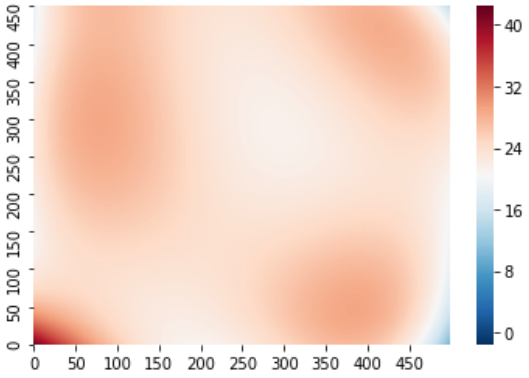


Figure 52: Dynamic Combinatoric Polynomial Regression Reconstruction

Figure 51 is generated by the dynamic polynomial regression and Figure 52 is generated by the Dynamic Combinatoric polynomial regression. If we compare these 2 figures with Figure 50 then we can see they are not similar.

As these models were not good enough to interpolate accurately so we moved to the other mathematical models using Tensor flow regression. We have tried Pure Fourier Series, Fourier series with Exponential Co-efficient and Fourier series with phase shift and Exponential Co-efficient, Dynamic Fourier Series. The main problem of these Fourier series models was the periodicity. The models those use the Fourier series produce the peak and trough in the output heat map. If we want to mitigate the amplitude of the peak and trough with respect to time in our output, we must consider so many Terms in the Fourier series which cannot be the appropriate solution to interpolate over the discrete data points. Because there will be sudden changes due to the peaks and troughs in the output heat map. To reduce these sudden changes, we must increase the variable frequency in the Fourier series equation.

Here we can identify and separate the peaks and troughs. We can see that, there are some layers which can be separated due to the peaks and troughs. If we want to reduce this, we must consider so many terms in the Fourier series. So, these models cannot be the appropriate models for the Interpolation.

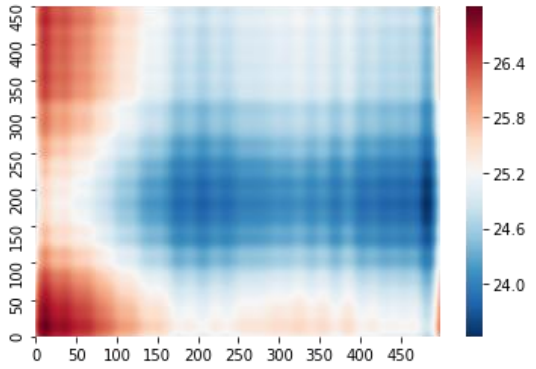


Figure 53: Dynamic Fourier series (variable coefficient & phase)

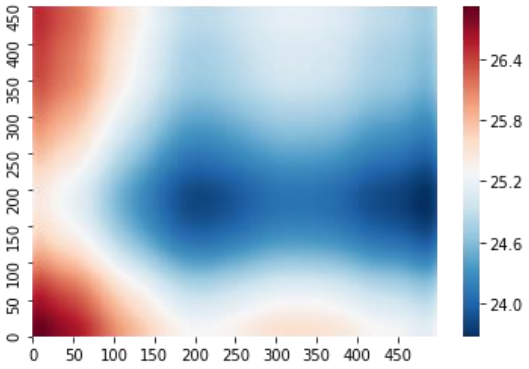


Figure 54: Dynamic Fourier Series (exponential co efficient; variable frequency & phase)

Here, in this case we have got better output than the previously produced output. Figure 53 is like the Figure 50 and Figure 54 is more like the Figure 50. Figure 53 has the sudden changes. This problem has been reduced by using exponential co-efficient in conjunction with variable frequency and phase algorithm. We can see there is still visible differences between Figure 50 and Figure 53, Figure 54. The main problem of the Fourier series models is that they require many terms in the series to reach the desired accuracy level.

Finally, a model has been used in our project to interpolate over the discrete data points and produces the continuous output using a universal function approximator that is using a Fully Connected Feed-Forward Artificial Neural Network (FC-FF-ANN). The Heatmap is a powerful data visualization technique that's used

to understand how a variable varies over a two-dimensional space. Heatmaps require continuous data, but due to the limitations in the sensors' data acquisition capabilities, they are only able to provide discrete datapoints. The interpolation engine takes these discrete data points, splits them into an 80/20 ratio & uses the bigger split to train a FC-FF-ANN model. Once the model is trained it behaves like a de facto universal function approximator that's capable of interpolating the datapoints in-between any spatially adjacent data points. The smaller portion of the split is used to determine the correct model complexity to achieve a proper balance between bias & variance.

8.1: Universal Function Approximator

The Universal function Approximation theorem [17] states that a Feed-Forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subset or on the discrete data points under mild assumptions on the activation function. That means a simple feed-forward neural network containing a specific number of neurons in the hidden layer can approximate almost any known function. Although feed-forward networks with a single hidden layer are universal approximator, the width of such networks must be exponentially large. In 2017 Lu et al [18] proved universal approximation theorem for width-bounded deep neural networks. In this paper, they have shown that the width (n+4) networks with Non-Linear activation function (ReLU, SoftPlus, TanH, SoftMax etc.) can approximate any integral function on n-dimensional input space.

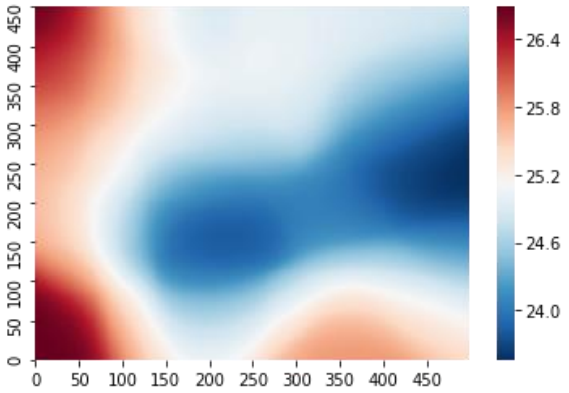


Figure 55: Fully Connected Feed-Forward Artificial Neural Network Reconstruction

This is the output of fully connected feed forward artificial neural network. If we carefully observe the Figure 50 and Figure 55 then we will see that they are almost similar. This Figure 55 is produced considering some discrete points from the Figure 50 matrix. Both images are almost the same. The universal approximator function provides the best output and for this reason we have used the FC-FF-ANN in the final version of the EnvironMetric suite.

8.2: Artificial Neural Network

Artificial Neural Network is specially used for the classification and regression problem. The Artificial Neural Networks ability to learn so quickly is what makes them so powerful and useful for a variety of tasks. For an artificial neural network to learn, it has to learn what it has done wrong and is doing right, this is called feedback. Feedback is how we learn what is right and wrong and this is also what an artificial neural network needs for it to learn. This is where we start to see similarities to the human brain. Neural networks learn things in the same way as the brain, typically by a feedback process called back-propagation. This is where you compare the output of the network with the output it was meant to produce, and using the difference between the outputs to modify the weights of the connections between the neurons in the network, working from the output units through the hidden neurons to the input neurons going backward. Over time, back-propagation causes the network to learn by making the gap between the output and the intended output smaller to the point where the two exactly match, so the neural network learns the correct output [19].

One of the most striking facts about neural networks is that they can compute any function at all. Neural networks have a kind of universality. No matter what function we want to compute, we know that there is

a neural network which can do the job. This universality theorem holds even if we restrict our networks to have just a single layer intermediate between the input and the output neurons a so called single hidden layer. So even very simple network architectures can be extremely powerful.

9: DataStudio (DS)

DataStudio is a macro-level data visualization tool currently implemented by using Google's DataStudio Service. It is purpose designed to perform inter-session data aggregations for most of its visualizations which are intended to give a high-level overview of the environmental pollution data. It offers a wide range of filters including but not limited to City, Area, Date Range & Metric.

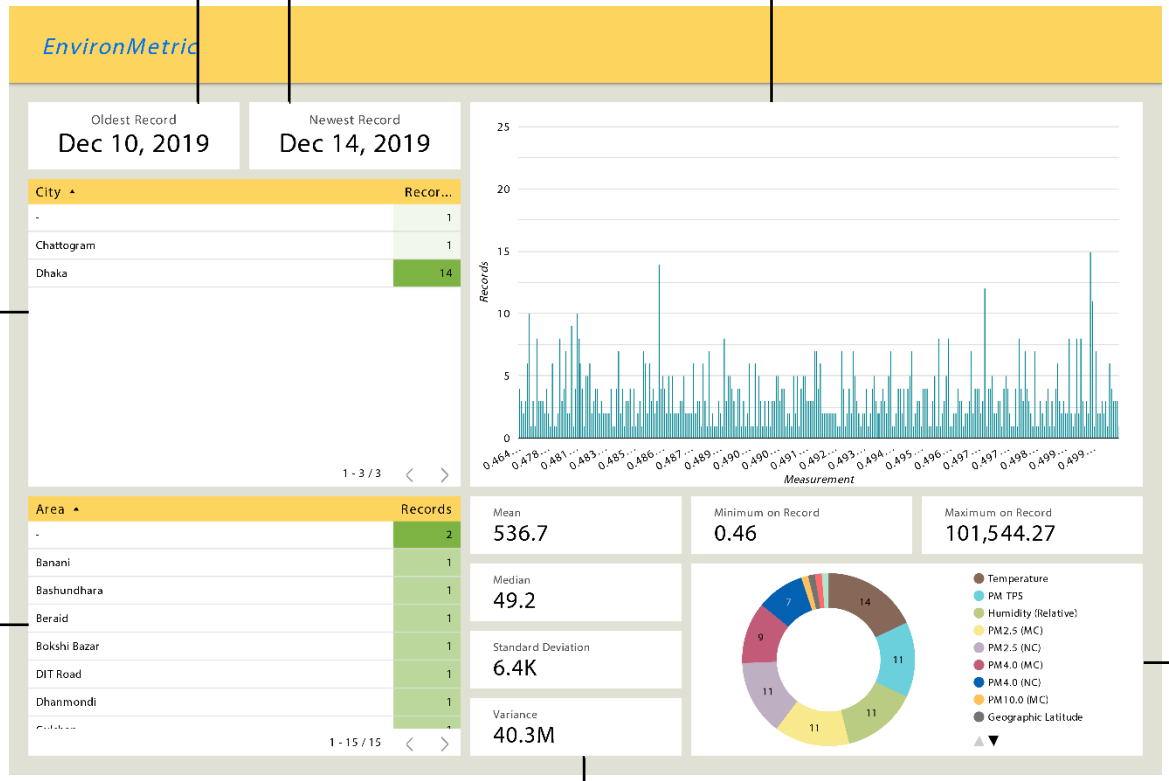
Despite having various data reporting tools, we chose Google's DataStudio as our data visualization tool because of several reasons:

- **Customizable & Efficient**: Prior to the availability and widespread adoption of Data Studio, many people relied on creating reports through Google Analytics' standard dashboard. This platform required users to click between many screens and select between many filters to create reports. A tie consuming process. However, in order to allocate budgets efficiently, decision makers require a fast turnaround on large amounts of accurate data. Reports must be produced quickly. They also must be easy to read and easy to understand. Data Studio is built around these basic concepts, making it our preferred choice for data reporting.
- **Multiple Data Sources**: One of the most significant and useful features of Data Studio is its ability to pull information from a variety of sources. Some of the most popular data sources include AdWords, Analytics, YouTube, Search Console, BigQuery, and MySQL. This function allows users to construct reports that can then be shared with colleagues and clients.
- **Sharing & Collaboration**: Data Studio operates like any other document on G-Drive. The author may grant access to peers with the option to allow or restrict editing. There is also an option to save and share content as a PDF file.
- **Filtering & Real-Time Updates & in Data Studio**: The Data Studio dashboard updates in real time. Users can choose to change the date range at the click of a button. Further to this, users may select from advanced filtering options that allow an at-a-glance view of relevant data sets.
- **No Charge**: Google Data Studio is completely free. Google is doing a fantastic job of building an abundance of intuitive and time saving tools. When Data Studio was first released, there was a 5-report limit for users. This cap has long since been removed. Now, anyone can make use of these proficient dashboards without any limitations. Free of charge.

We can select cities that we have taken environmental measurements, and this also shows how many records are there

This indicates the oldest data that we have gathered and the latest ones

This is a bar diagram showing time vs measurement graphs



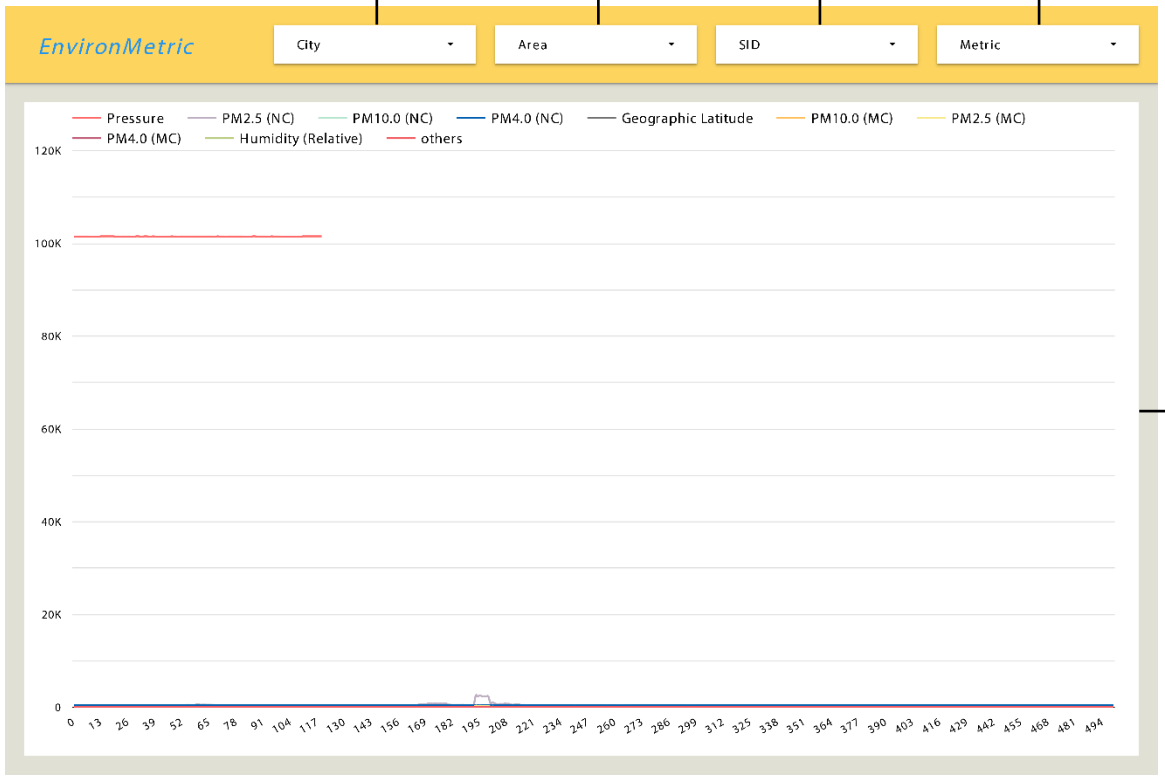
This pie-chart shows variations of readings based on a specific metric

Under city tab, we can choose certain areas within that city and the records of data.

These are statistical data based on which metric is chosen to be shown


These are statistical data showing central tendency of a specific metric within a set date range
EnvironMetric
Select date range
Metric
This section shows period of date and records of data that we want to see such as temperature, pressure etc.
Minimum
0.5
Maximum
101.5K
Average
536.7
PM TPS
Temperature
Humidity (Relative)
PM4.0 (MC)
PM2.5 (MC)
PM10.0 (MC)
Geographic Latitude
PM4.0 (NC)
PM10.0 (NC)
PM2.5 (NC)
500
400
300
200
100
0
-100
Dec 10
Dec 11
Dec 12
Dec 13
Dec 14
This is nothing but a time series graph showing measured values against desired day period

By selecting these blocks, user can see graphical representation of metric measurements as he requires. For example, he can do filtering based on his city selection or metric selection. Moreover, we can also review our records of previous session id by accessing SID.



This is a time series graph showing number of incidents against values of the specified metric/metrics

Outcome

The EnvironMetric's Remote Sensor Array (RSA) is capable of collecting data in both time and spatial paradigm. For collecting data in the temporal paradigm, the RSA does not require a mobile platform to carry it around. The RSA is placed at a designated place and the data of that place is probed and accumulated by the RSA. The EnvironMetric team has managed to conduct several test sessions with the RSA operating in the temporal paradigm and collect atmospheric pollution data for various locations throughout Dhaka and Chittagong. All the data that was collected was also successfully archived in the PermaData database, analyzed & visualized in DataStudio and processed through the Interpolation Engine; thus, showcasing the effectiveness of the EnvironMetric pipeline.

But for spatial paradigm the RSA requires a platform that can carry it and traverse with it over an area for a certain amount of time. That is why the UAV Carrier Platform (UAV-CP) is used in the EnvironMetric system. Keeping design headroom for future sensor addition and upgrades to the RSA in mind the platform was designed to be big and robust. As a result, the UAV that needs to carry the RSA was designed and built to carry massive payloads. Thus, a 1000 class hexacopter frame was used along with six MultiStar Elite 5008 330KV motors was used; further details are given in section 2 of this report. With its current configuration the UAV-CP can carry a theoretical maximum of 18kg of All-Up-Weight.

In Bangladesh, flying a large drone for any purpose requires the permission of the Ministry of Defense (MoD) and Civil Aviation Authority of Bangladesh (CAAB). Considering the size, capability and specifications of EnvironMetric's UAV-CP, it is classified as a military-purpose drone; as a result, getting the permission for flying the UAV for our testing and research purpose became mandatory. Thus, the EnvironMetric team along with the supervisor visited the MoD multiple times for meeting with the members of the MoD, CAAB, various branches of the Law Enforcement & Intelligence Agencies of the Bangladeshi government to apply for a flying permit. It's still an ongoing process as of the December of 2019 but their response to our application are attached herewith.

বিমান বাহিনী সদর দপ্তর

বিমান গোয়েন্দা পরিদপ্তর

ঢাকা সেনানিবাস

দূরাঃ ৫৫০৬০০০০-১০ সঞ্জ ৩০৪৩

০৭.০৩.২৬০০.০০৫,৮৭,২৪৫.১৯.১৯০৩/ ১১ক ০৬ ডিসেম্বর ২০১৯

নর্থ সাউথ বিশ্ববিদ্যালয়ের চারজন ছাত্রের গবেষণার কাজে তৈরি হেক্সা-কপ্টার পরিচালন প্রসঙ্গে

সূত্র

ক। প্রতিরক্ষা মন্ত্রণালয় পত্র নংঃ ২৩,০০,০০০০.০৫০.৬৪.০০২.১৯-২০০ তাং ২৮ নভেম্বর ২০১৪।

১. সূত্র ‘ক’ পত্রের পরিপ্রেক্ষিতে জানানে যাচ্ছে যে, নর্থ সাউথ বিশ্ববিদ্যালয়ের চারজন ছাত্রের গবেষণার কাজে তৈরি হেক্সা-কপ্টার পরিচালনার জন্য প্রয়োজনীয় সকল তথ্য সংযোজন করা হয়নি বিধায় হেক্সা-কপ্টার পরিচালনায় অত্র পরিদপ্তর কর্তৃক মতামতের নিমিত্তে নিম্নলিখিত তথ্যাদি প্রয়োজনেঃ

ক। হেক্সা-কল্টার পরিচালনার নির্দিষ্ট তারিখ ও সময়।

খ। গবেষণার কাজ পরিচালনার জন্য নির্ধারিত এলাকার সীমানা চিত্রসহ নির্ধারণ(অক্ষাংশ ও দ্রাঘিমাংশ সহ)।

গ। নির্ধারিত এলাকাটির হেক্সা-কপ্টার পরিচালনার জন্য নির্দিষ্ট উচ্চতা।

ঘ। নিকটস্থ হাসপাতাল ও ফায়ার সার্ভিস ষ্টেশনের দূরত্বসহ ঠিকানা।

ঙ। অঙ্গীকারনামা।

২। ইহা আপনাদের অবগতি ও পরবর্তী কার্যক্রমের জন্য প্রেরণ করা হল।

মোহাম্মাদ আসাদুজ্জামান

উইং কমান্ডার

পরিচালকের পক্ষে

নর্থ সাউথ বিশ্ববিদ্যালয়ের চারজন ছাত্রের গবেষণার কাজে তৈরি হেক্সা-কপ্টার পরীক্ষা মুলকভাবে উড্ডয়নের বিষয়ে মতামত/সুপারিশ প্রদানের জন্য ০৯ ডিসেম্বর ২০১৯ তারিখে প্রতিরক্ষা মন্ত্রণালয়ে অনুষ্ঠিত এতদসংক্রান্ত আন্তঃমন্ত্রণালয় কমিটির সভার সুপারিশকৃত

শর্তাবলী নিম্নরূপ:

'শর্তাবলী'

১. আবেদনকারী কর্তৃপক্ষ নর্থ সাউথ বিশ্ববিদ্যালয় কর্তৃক আগামী ---------- থেকে -------- তারিখ পর্যন্ত নর্থ সাউথ বিশ্ববিদ্যালয়ের চারজন ছাত্রের গবেষণার কাজে ..... এলাকায় নির্ধারিত দিনে সকাল ---------টা থেকে বিকাল --------- টা মধ্যে সর্বোচ্চ ----- মিটার উচ্চতায় হেক্সা-কপ্টার উড্ডয়ন কার্যক্রম সংশ্লিষ্ট প্রতিষ্ঠানের প্রতিনিধির সঙ্গে সমন্বয়পূর্বক সম্পন্ন করতে হবে। এয়ারপোর্ট/এয়ার ট্রেনিং এলাকার ক্ষেত্রে টাওয়ার কর্তৃপক্ষের সঙ্গে সমন্বয়পূর্বক হেক্সা-কপ্টার উড্ডয়নের উচ্চতার বিষয়টি নিশ্চিত করে কার্যক্রম গ্রহণ করতে হবে;

২. আবেদনে উল্লিখিত এলাকা ব্যতীত অন্য কোন এলাকায় হেক্সা-কপ্টার উড্ডয়ন করা যাবে না এবং অন্য কোন এলাকার ভিডিওচিত্র/আকাশ আলোকচিত্র ধারণ করা যাবেনা-এ ব্যাপারে সর্বোচ্চ সতর্কতা অবলম্বন করতে হবে;

৩. আবেদনকারী কর্তৃপক্ষ কর্তৃক হেক্সা-কপ্টার উড্ডয়ন করার কমপক্ষে ৭(সাত) কার্যদিবসের পূর্বে ফ্লাইট সিডিউলসহ হেক্সা-কপ্টার উড্ডয়নে ব্যবহার্য যন্ত্রপাতির বিবরণ এবং স্যুটিং টিমের অন্তর্ভুক্ত প্রত্যেক সদস্যের বায়াডাটা, আলোকচিত্র ধারণে তাঁদের সংশ্লিষ্টতা ও ঠিকানা ইত্যাদি প্রতিরক্ষা মন্ত্রণালয়কে অবহিতপূর্বক বেসামরিক বিমান চলাচল কর্তৃপক্ষ (বেবিচক), বাংলাদেশ জরিপ অধিদপ্তর, ডিজিএফআই, এনএসআই, বিমান গোয়েন্দা পরিদপ্তর এবং বিমান বাহিনী সদর দপ্তর-এর নিকট দাখিল করতে হবে;

৪. হেক্সা-কপ্টার উড্ডয়ন, আকাশ আলোকচিত্র ধারণ ও সেন্সরিং কার্যক্রমে কোন বিদেশী প্রতিনিধি অন্তর্ভুক্ত থাকলে প্রতিরক্ষা মন্ত্রালয়কে অবহিত করে যথানিয়মে ডিজিএফআই-এর ছাড়পত্র গ্রহণ করতে হবে ।

৫. আবেদনকারী কর্তৃপক্ষের দায়িত্বে হেক্সা-কপ্টার উড্ডয়ন ও আকাশ আলোকচিত্র ধারণকালে বেবিচক, ডিজিএফআই, এনএসআই, বাংলাদেশ জরিপ অধিদপ্তর, বিমান বাহিনী ও স্বরাষ্ট্র মন্ত্রণালয়ের সাথে সমন্বয়পূর্বক উক্ত সংস্থাসমূহের প্রতিনিধিগণের উপস্থিতি নিশ্চিত করতে হবে;

৬. আবেদনকারীর প্রস্তাবিত ফ্লাইট সিডিউল নিশ্চিত করা সাপেক্ষে বেবিচক কর্তৃপক্ষ হেক্সা-কপ্টার উড্ডয়ন কার্যক্রম পরিচালনার বিষয়ে যাবতীয় তথ্য-উপাত্ত বিমানসদরসহ সংশ্লিষ্ট সকলকে অবহিত করবেন।

৭. হেক্সা-কপ্টার উড্ডয়নের মাধ্যমে ধারণকৃত ভিডিওচিত্রে/আকাশ আলোকচিত্র KPI/Vulnerable Points-এর দৃশ্য অন্তর্ভুক্ত করা যাবে না এবং এরূপ কোন অন্তর্ভুক্তি থাকলে জরিপ অধিদপ্তর উক্ত অংশ/অংশসমূহ অবলোপন করতে পারবে;

৮. আবেদনকারী কর্তৃপক্ষের ধারণকৃত ভিডিওচিত্র/আকাশ আলোকচিত্র জরিপ অধিদপ্তরে সংরক্ষণের জন্য জমা দিতে হবে। জরিপ অধিদপ্তর কর্তৃক ডিজিএফআই, এনএসআই-এর প্রতিনিধির উপস্থিতিতে সেন্সরিং ও ভেটিং-এর পর ছাড়পত্র প্রদান সাপেক্ষে ভিডিও চিত্রটি সংশ্লিষ্ট সংস্থার নিকট হস্তান্তর করা হবে। সংশ্লিষ্ট কর্তৃপক্ষের ছাড়পত্র প্রদানের পূর্বে এর কোন কপি করা যাবেনা;

৯. ধারণকৃত ভিডিওচিত্রে/আকাশ আলোকচিত্রে কোন ধরনের সুপার-ইমপোজ বা Extra-Imaging করা যাবে না এবং এমন কোন ভাবে ব্যবহার করা যাবে না যাতে বাংলাদেশের ভাবমূর্তি, স্বার্থ ক্ষুন্ন হয় এবং তথ্যের বিকৃতি ঘটে;

১০. আবেদনে উল্লিখিত ক্ষেত্র ব্যতীত অন্য কোন ক্ষেত্রে ধারণকৃত ভিডিওচিত্র/আকাশ আলোকচিত্র ব্যবহার করা যাবেনা এবং কোন প্রকার বাণিজ্যিক বিপণন করা যাবেনা;

১১. ধারণকৃত ভিডিওচিত্র/আকাশ আলোকচিত্র শ্রেণীভুক্ত দলিল হিসেবে গণ্য হবে এবং এ দলিল সংরক্ষণ ও ব্যবহারের ক্ষেত্রে সংশ্লিষ্ট সকল বিধি-বিধান/নিয়ম-কানুন যথাযথভাবে প্রতিপালন করতে হবে ।

১২. ধারণকৃত আকাশ আলোকচিত্র /ভিডিওচিত্রের কপি জরিপ অধিদপ্তরে সংরক্ষণ করা হবে, যা সরকার কর্তৃক রাষ্ট্রীয় প্রয়োজনে ব্যবহার করা যাবে;

১৩. 'নর্থ সাউথ বিশ্ববিদ্যালয় কর্তৃপক্ষ ড্রোন উড্ডয়ন ও ভিডিওচিত্র/আকাশ আলোকচিত্র ধারণের প্রচলিত সরকারি নীতিমালা ও যাবতীয় বিধি-বিধান যথাযথভাবে মেনে চলতে বাধ্য থাকবেন ।

১৪. হেক্স-কপ্টার উড্ডয়ন এবং ভিডিওচিত্র/আকাশ আলোকচিত্র /সার্ভে ডাটা ধারণকালে কোন দুর্ঘটনা/অনাকাঙ্খিত ঘটনার কারণে জীবন ও সম্পত্তির হানি ঘটলে তার ক্ষতিপূরণ সরকার কর্তৃক নির্ধারিত হারে নর্থ সাউথ বিশ্ববিদ্যালয় কর্তৃপক্ষকে বহন করতে হবে;

১৫. উক্ত শর্তাবলীর ব্যত্যয় ঘটলে সংশ্লিষ্ট প্রতিষ্ঠানকে কালো তালিকাভুক্ত করে আইনানুগ ব্যবস্থা গ্রহণ করা হবে; এবং

১৬. প্রতিরক্ষা মন্ত্রণালয় কেবলমাত্র প্রস্তাবিত এলাকায় হেক্সা-কপ্টার উড্ডয়ন ও ভিডিওচিত্র/ আকাশ আলোকচিত্র ধারণের অনুমতির সুপারিশ প্রদান করছে। ড্রোন ক্রয়, আমদানি, ব্যবহারকারী প্রতিষ্ঠানের লাইসেন্সিং ইত্যাদি যাবতীয় বিষয়ে আবেদনকারী কর্তৃপক্ষ সরকারের যথাযথ নিয়ম প্রতিপালনে বাধ্য থাকবেন।

References

[1] "Fossil fuels still dominate U.S. energy consumption despite recent market share decline," U.S. Energy Information Administration (EIA), 2016.

[2] "Climate Change: How Do We Know?," NASA Climate.

[3] "Methane Levels," [Online]. Available: https://www.methanelevels.org/.

[4] Phys.org, "Effects of past tropical deforestation will be felt for years to come," 2016. [Online]. Available: https://phys.org/news/2016-07-effects-tropical-deforestation-felt-years.html.

[5] "World Population Growth," Our World in Data, 2017.

[6] "World of Change: Global Temperatures," NASA Earth Observatory.

[7] "Stereo camera," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Stereo_camera.

[8] "Intel RealSense D435i Datasheet," Intel Corporation, [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/Intel-RealSense-D400-Series-Datasheet.pdf.

[9] V. A. P. a. I. D. R. Carl Yuheng Ren, "gSLICr: SLIC superpixels at over 250Hz," University of Oxford, 2015. [Online]. Available: http://www.robots.ox.ac.uk/~victor/gslicr/.

[10] "Perspective," Wolfram Math World, [Online]. Available: http://mathworld.wolfram.com/Perspective.html. [Accessed 2019].

[11] "MAVLink," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/MAVLink. [Accessed 2019].

[12] "Coordinate Transformations," [Online]. Available: https://navpy.readthedocs.io/en/latest/code_docs/coordinate_transformations.html. [Accessed 2019].

[13] M. B. I. R. S. S. N. M. A. S. B. a. M. A. M. A. Md. Syedul Amin, "A Novel Vehicle Stationary Detection Utilizing Map Matching and IMU Sensors".

[14] M. V. Biezen, "Special Topics: The Kalman Filter," September 2015. [Online]. Available: www.ilectureonline.com.

[15] R. A. S. a. K. H. Matthew B. Rhudy, in *A Kalman Filter Tutorial for Undergraduate Students*.

[16] "Complementary Filter," [Online]. Available: https://sites.google.com/site/myimuestimationexperience/filters/complementary-filter. [Accessed 2019].

[17] A. Kratsios, "Universal Approximation Theorems," 2019.

[18] H. P. F. W. Z. H. a. L. W. Zhou Lu, "The Expressive Power of Neural Networks: A View from the Width," *Advances in Neural Information Processing Systems,* 2017.

[19] J. Dacombe, "An introduction to Artificial Neural Networks (with example)," 2017. [Online]. Available: https://medium.com/@jamesdacombe/an-introduction-to-artificial-neural-networks-with-example-ad459bb6941b.

A: RSA BIOS source code

BNO055.py

```
# Dependencies
from time import sleep as Sleep

import Backend.PolyBus as PolyBus

class BNO055:

    # Device Characteristics
    NAME = "BNO055"
    MEASUREMENTS = ["ACCELERATION|RAW", "ACCELERATION|LINEAR", "ACCELERATION|GRAVITY", "ANGULARVELOCITY|RAW", "MAGNETICFIELD|RAW", "ATTITUDE|EULER", "ATTITUDE|QUATERNION"]

    # Registers (Page 0)
    CHIP_ID          = 0x00 # BNO055 CHIP ID
    ACC_ID           = 0x01 # ACC chip ID
    MAG_ID           = 0x02 # MAG chip ID
    GYR_ID           = 0x03 # GYRO chip ID
    SW_REV_ID_LSB    = 0x04 # SW Revision ID <7:0>
    SW_REV_ID_MSB    = 0x05 # SW Revision ID <15:8>
    BL_REV_ID        = 0x06 # Boot loader Version
    PAGE_ID          = 0x07 # Page ID
    ACC_DATA_X_LSB   = 0x08 # Acceleration Data X <7:0>
    ACC_DATA_X_MSB   = 0x09 # Acceleration Data X <15:8>
    ACC_DATA_Y_LSB   = 0x0A # Acceleration Data Y <7:0>
    ACC_DATA_Y_MSB   = 0x0B # Acceleration Data Y <15:8>
```

```
ACC_DATA_Z_LSB   = 0x0C # Acceleration Data Z <7:0>
ACC_DATA_Z_MSB   = 0x0D # Acceleration Data Z <15:8>
MAG_DATA_X_LSB   = 0x0E # Magnetometer Data X <7:0>
MAG_DATA_X_MSB   = 0x0F # Magnetometer Data X <15:8>
MAG_DATA_Y_LSB   = 0x10 # Magnetometer Data Y <7:0>
MAG_DATA_Y_MSB   = 0x11 # Magnetometer Data Y <15:8>
MAG_DATA_Z_LSB   = 0x12 # Magnetometer Data Z <7:0>
MAG_DATA_Z_MSB   = 0x13 # Magnetometer Data Z <15:8>
GYR_DATA_X_LSB   = 0x14 # Gyroscope Data X <7:0>
GYR_DATA_X_MSB   = 0x15 # Gyroscope Data X <15:8>
GYR_DATA_Y_LSB   = 0x16 # Gyroscope Data Y <7:0>
GYR_DATA_Y_MSB   = 0x17 # Gyroscope Data Y <15:8>
GYR_DATA_Z_LSB   = 0x18 # Gyroscope Data Z <7:0>
GYR_DATA_Z_MSB   = 0x19 # Gyroscope Data Z <15:8>
EUL_DATA_X_LSB   = 0x1A # Heading Data <7:0>
EUL_DATA_X_MSB   = 0x1B # Heading Data <15:8>
EUL_DATA_Y_LSB   = 0x1C # Roll Data <7:0>
EUL_DATA_Y_MSB   = 0x1D # Roll Data <15:8>
EUL_DATA_Z_LSB   = 0x1E # Pitch Data <7:0>
EUL_DATA_Z_MSB   = 0x1F # Pitch Data <15:8>
QUA_DATA_W_LSB   = 0x20 # Quaternion w Data <7:0>
QUA_DATA_W_MSB   = 0x21 # Quaternion w Data <15:8>
QUA_DATA_X_LSB   = 0x22 # Quaternion x Data <7:0>
QUA_DATA_X_MSB   = 0x23 # Quaternion x Data <15:8>
QUA_DATA_Y_LSB   = 0x24 # Quaternion y Data <7:0>
QUA_DATA_Y_MSB   = 0x25 # Quaternion y Data <15:8>
QUA_DATA_Z_LSB   = 0x26 # Quaternion z Data <7:0>
QUA_DATA_Z_MSB   = 0x27 # Quaternion z Data <15:8>
LIA_DATA_X_LSB   = 0x28 # Linear Acceleration Data X <7:0>
LIA_DATA_X_MSB   = 0x29 # Linear Acceleration Data X <15:8>
LIA_DATA_Y_LSB   = 0x2A # Linear Acceleration Data Y <7:0>
```

```
LIA_DATA_Y_MSB   = 0x2B # Linear Acceleration Data Y <15:8>
LIA_DATA_Z_LSB   = 0x2C # Linear Acceleration Data Z <7:0>
LIA_DATA_Z_MSB   = 0x2D # Linear Acceleration Data Z <15:8>
GRV_DATA_X_LSB   = 0x2E # Gravity Vector Data X <7:0>
GRV_DATA_X_MSB   = 0x2F # Gravity Vector Data X <15:8>
GRV_DATA_Y_LSB   = 0x30 # Gravity Vector Data Y <7:0>
GRV_DATA_Y_MSB   = 0x31 # Gravity Vector Data Y <15:8>
GRV_DATA_Z_LSB   = 0x32 # Gravity Vector Data Z <7:0>
GRV_DATA_Z_MSB   = 0x33 # Gravity Vector Data Z <15:8>
TEMP             = 0x34 # Temperature
CALIB_STAT       = 0x35
ST_RESULT        = 0x36
INT_STA          = 0x37
SYS_CLK_STATUS   = 0x38
SYS_STATUS       = 0x39 # System Status Code
SYS_ERR          = 0x3A # System Error Code
UNIT_SEL         = 0x3B
OPR_MODE         = 0x3D
PWR_MODE         = 0x3E
SYS_TRIGGER      = 0x3F
TEMP_SOURCE      = 0x40
AXIS_MAP_CONFIG  = 0x41
AXIS_MAP_SIGN    = 0x42
ACC_OFFSET_X_LSB = 0x55 # Accelerometer Offset X <7:0>
ACC_OFFSET_X_MSB = 0x56 # Accelerometer Offset X <15:8>
ACC_OFFSET_Y_LSB = 0x57 # Accelerometer Offset Y <7:0>
ACC_OFFSET_Y_MSB = 0x58 # Accelerometer Offset Y <15:8>
ACC_OFFSET_Z_LSB = 0x59 # Accelerometer Offset Z <7:0>
ACC_OFFSET_Z_MSB = 0x5A # Accelerometer Offset Z <15:8>
MAG_OFFSET_X_LSB = 0x5B # Magnetometer Offset X <7:0>
MAG_OFFSET_X_MSB = 0x5C # Magnetometer Offset X <15:8>
```

```
MAG_OFFSET_Y_LSB = 0x5D # Magnetometer Offset Y <7:0>
MAG_OFFSET_Y_MSB = 0x5E # Magnetometer Offset Y <15:8>
MAG_OFFSET_Z_LSB = 0x5F # Magnetometer Offset Z <7:0>
MAG_OFFSET_Z_MSB = 0x60 # Magnetometer Offset Z <15:8>
GYR_OFFSET_X_LSB = 0x61 # Gyroscope Offset X <7:0>
GYR_OFFSET_X_MSB = 0x62 # Gyroscope Offset X <15:8>
GYR_OFFSET_Y_LSB = 0x63 # Gyroscope Offset Y <7:0>
GYR_OFFSET_Y_MSB = 0x64 # Gyroscope Offset Y <15:8>
GYR_OFFSET_Z_LSB = 0x65 # Gyroscope Offset Z <7:0>
GYR_OFFSET_Z_MSB = 0x66 # Gyroscope Offset Z <15:8>
ACC_RADIUS_LSB   = 0x67 # Accelerometer Radius
ACC_RADIUS_MSB   = 0x68 # Accelerometer Radius
MAG_RADIUS_LSB   = 0x69 # Magnetometer Radius
MAG_RADIUS_MSB   = 0x6A # Magnetometer Radius

# Registers (Page 1)
ACC_Config       = 0x08
MAG_Config       = 0x09
GYR_Config_0     = 0x0A
GYR_Config_1     = 0x0B
ACC_Sleep_Config = 0x0C
GYR_Sleep_Config = 0x0D
INT_MSK          = 0x0F
INT_EN           = 0x10
ACC_AM_THRES     = 0x11
ACC_INT_Settings = 0x12
ACC_HG_DURATION  = 0x13
ACC_HG_THRES     = 0x14
ACC_NM_THRES     = 0x15
ACC_NM_SET       = 0x16
GYR_INT_SETTING  = 0x17
```

```
GYR_HR_X_SET     = 0x18
GYR_DUR_X        = 0x19
GYR_HR_Y_SET     = 0x1A
GYR_DUR_Y        = 0x1B
GYR_HR_Z_SET     = 0x1C
GYR_DUR_Z        = 0x1D
GYR_AM_THRES     = 0x1E
GYR_AM_SET       = 0x1F

# Hardware & Software Initializer routine
def __init__(Self, ChannelID, DeviceAddress, Verbose=True):
    if Verbose: print("    > Initializing the BNO055 at 0x"+format(DeviceAddress, "02X")+" on channel "+str(ChannelID)); pass
    Self.ChannelID = ChannelID
    Self.DeviceAddress = DeviceAddress

    if Verbose: print("      > Performing a Power-on-Reset operation. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.SYS_TRIGGER, 0x20); Sleep(0.650)
    if Verbose: print(" Success!"); pass

    if Verbose: print("      > Configuring oscillator, resetting interrupts and performing POST. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.SYS_TRIGGER, 0B01000001); Sleep(1.0)
    #                                                                                  ▲▲▲▲▲▲▲▲
    #                                                                                  ||||||||
    #                                                                                  |||||||└──> SELF_TEST: 1 = Perform POST, 0 = Ignore
    #                                                                                  |||||||
    #                                                                                  ||||||└───> Reserved
    #                                                                                  |||||└────> Reserved
    #                                                                                  ||||└─────> Reserved
    #                                                                                  |||└──────> Reserved
    #                                                                                  |||
    #                                                                                  ||└───────> RST_SYS: 1 = Perform Power-on-Reset (PoR), 0 = Ignore
```

```
#                                                                                  |└──────────> RST_INT: 1 = Reset all interrupt status bits & INT output, 0 = Ignore
#                                                                                  └──────────> CLK_SEL: 0 = Use internal oscillator, 1 = Use external oscillator

POSTResults = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.ST_RESULT, False)
POSTValues = [False, False, False, False, False, False, False, False]
POSTValues[0] = not bool(POSTResults & 0B1)
POSTValues[1] = not bool((POSTResults & 0B10) >> 1)
POSTValues[2] = not bool((POSTResults & 0B100) >> 2)
POSTValues[3] = not bool((POSTResults & 0B1000) >> 3)
POSTLables = ["ST_ACC", "ST_MAG", "ST_GYR", "ST_MCU", "Reserved", "Reserved", "Reserved", "Reserved"]
FailedPOST = False
ExceptionMessage = "ERROR: The device failed to POST on"
for Iterator in range(len(POSTValues)):
    if POSTValues[Iterator]:
        ExceptionMessage += " "+POSTLables[Iterator]
        FailedPOST = True
        pass
    pass
ExceptionMessage += "\n-----\n"
if FailedPOST:
    if Verbose: print(" Fail!"); pass
    raise Exception(ExceptionMessage)
else:
    if Verbose: print(" Pass!"); pass
    pass

if Verbose: print("      > Configuring the thermometer data source. . .", end=""); pass
# The temperature can be read from one of two sources: Accelerometer or Gyroscope.
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.TEMP_SOURCE, 0B00000000); Sleep(0.10)
#                                                                                  ▲▲▲▲▲▲▲▲
#                                                                                  ||||||||
```

```
#                                                                                                  |||||||└─> ┐               ┌ 00 = Accelerometer, 01 = Gyroscope
#                                                                                                  ||||||└─> ┴ TEMP_SOURCE ─┴ Others = Undefined
#                                                                                                  ||||||
#                                                                                                  |||||└─────> Reserved
#                                                                                                  ||||└──────> Reserved
#                                                                                                  |||└───────> Reserved
#                                                                                                  ||└────────> Reserved
#                                                                                                  |└─────────> Reserved
#                                                                                                  └──────────> Reserved
if Verbose: print(" Success!"); pass

if Verbose: print("      > Configuring data acquisition units. . .", end=""); pass
# The measurement units for the various data outputs (regardless of operation mode) is user configurable.
#
#     +----------------------+------------------+
#     | DATA                 | UNITS            |
#     +----------------------+------------------+
#     | Acceleration,        |                  |
#     | Linear acceleration, | m/s^2, mg        |
#     | Gravity vector       |                  |
#     +----------------------+------------------+
#     | Magnetic-            |                  |
#     | Field-               | Micro Tesla      |
#     | Strength             |                  |
#     +----------------------+------------------+
#     | Angular Velocity     | °/s, ᶜ/s         |
#     +----------------------+------------------+
#     | Euler Angles         | °, ᶜ             |
#     +----------------------+------------------+
#     | Quaternion           | Quaternion units |
#     +----------------------+------------------+
```

```
#     | Temperature          | °C, °F           |
#     +----------------------+------------------+
#
# The data output format can be selected by the user to switch between the distinct
# orientation definitions described by the Windows and the Android operating systems.
#
#     +----------------+-------------------------------------+-------------------------------------+
#     | ROTATION ANGLE | RANGE (ANDROID FORMAT)              | RANGE (WINDOWS FORMAT)              |
#     +----------------+-------------------------------------+-------------------------------------+
#     | Pitch          |            +180° to -180°           |            -180° to +180°           |
#     |                | (turning clockwise decreases values) | (turning clockwise increases values) |
#     +----------------+-------------------------------------+-------------------------------------+
#     | Roll           | -90° to +90° (increasing with increasing inclination)                      |
#     +----------------+-------------------------------------------------------------------------------+
#     | Heading / Yaw  | 0° to 360° (turning clockwise increases values)                            |
#     +----------------+-------------------------------------------------------------------------------+
#
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.UNIT_SEL, 0B00000000); Sleep(0.10)
#                                                                                 ▲▲▲▲▲▲▲▲
#                                                                                 ||||||||
#                                                                                 |||||||└──> ACC_Unit: 0 = meters-per-second-squared, 1 = milli-g
#                                                                                 ||||||└───> GYR_Unit: 0 = Degrees per second, 1 = Radians per second
#                                                                                 |||||└────> EUL_Unit: 0 = Degrees, 1 = Radians
#                                                                                 |||||
#                                                                                 ||||└─────> Reserved
#                                                                                 ||||
#                                                                                 |||└──────> TEMP_Unit: 0 = Celsius, 1 = Fahrenheit
#                                                                                 |||
#                                                                                 ||└───────> Reserved
#                                                                                 |└────────> Reserved
#                                                                                 |
```

```
#                                                                                 └──────────> ORI_ANDROID_WINDOWS: 0 = Windows orientation, 1 = Android orientation
if Verbose: print(" Success!"); pass

if Verbose: print("      > Configuring the power mode. . .", end=""); pass
# The BNO055 supports three different power modes: Normal mode, Low Power Mode and Suspend mode.
#
# Normal Mode
# -----------
# In normal mode all sensors required for the selected operating mode are always switched ON.
# The register map and the internal peripherals of the MCU are always operative in this mode.
#
# Low Power Mode
# --------------
# If no motion is detected for a configurable duration (default 5 seconds), the BNO055 enters
# the low power mode. In this mode only the accelerometer is active. Once motion is detected,
# the system is woken up and normal mode is entered.
#
# Suspend Mode
# ------------
# In suspend mode the system is paused and all the sensors and the microcontroller are put
# into sleep mode. No values in the register map will be updated in this mode. Exit from
# suspend mode by writing to the PWR_MODE register.
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.PWR_MODE, 0B00000000); Sleep(0.20)
#                                                                               ▲▲▲▲▲▲▲▲
#                                                                               ||||||||
#                                                                               |||||||└──> ┐            ┌ 00 = Normal Mode, 01 = Low Power Mode
#                                                                               ||||||└──> ┴ PWR_MODE ─┴ 10 = Suspend Mode, 11 = Undefined
#                                                                               ||||||
#                                                                               |||||└────> Reserved
#                                                                               ||||└─────> Reserved
#                                                                               |||└──────> Reserved
```

```
#                                                                                ||└──────> Reserved
#                                                                                |└──────> Reserved
#                                                                                └───────> Reserved
if Verbose: print(" Success!"); pass

if Verbose: print("      > Configuring the operation mode. . .", end=""); pass
# The BNO055 provides a variety of output signals, which can be chosen by selecting the
# appropriate operation mode. The default operation mode after power-on is CONFIGMODE.
#
# Default sensor settings
#
#     +----------------+-----------+-----------+
#     | SENSOR         | RANGE     | BANDWIDTH |
#     +----------------+-----------+-----------+
#     | Accelerometer  | 4G        | 62.5 Hz   |
#     +----------------+-----------+-----------+
#     | Magnetometer   | N/A       | 10 Hz     |
#     +----------------+-----------+-----------+
#     | Gyroscope      | 2000 dps  | 32 Hz     |
#     +----------------+-----------+-----------+
#
# All but the accelerometer range settings are overridden in the Fusion modes.
#
# Configuration mode
# ------------------
#
#     CONFIGMODE
#     ----------
#     This mode is used to configure the BNO055, wherein all output data is reset to zero and sensor
#     fusion is halted. This is the only mode in which all the writable register map entries can be
#     changed.
```

```
#
# Non-Fusion modes
# ----------------
#
#     ACCONLY
#     -------
#     In this mode, only the accelerometer is switched on; the other sensors (magnetometer, gyro) are
#     suspended to lower the power consumption.
#
#     MAGONLY
#     -------
#     In this mode, the BNO055 behaves like a stand-alone magnetometer with the accelerometer and
#     gyroscope being suspended.
#
#     GYROONLY
#     --------
#     In this mode, the BNO055 behaves like a stand-alone gyroscope, with the accelerometer and
#     magnetometer being suspended.
#
#     ACCMAG
#     ------
#     Both the accelerometer and the magnetometer are switched on; with the gyroscope being suspended.
#
#     ACCGYRO
#     -------
#     Both the accelerometer and the gyroscope are switched on; with the magnetometer being suspended.
#
#     MAGGYRO
#     -------
#     Both the magnetometer and the gyroscope are switched on; with the accelerometer being suspended.
#
```

```
#     AMG
#     ---
#     All three sensors accelerometer, magnetometer and gyroscope are switched on.
#
#
# Fusion modes
# ------------
#
#     IMU
#     ---
#     In the IMU mode the relative orientation of the BNO055 in space is calculated from the
#     accelerometer and gyroscope data. This mode has a high data throughput.
#     Maximum Data Acquisition Frequency is 100Hz
#
#     COMPASS
#     -------
#     This mode is intended to measure the magnetic earth field and calculate the geographic
#     direction. The earth magnetic field is a vector with the horizontal components x, y and the
#     vertical z component. For heading calculation, only the horizontal components x and y are used.
#     The measurement accuracy depends on the stability of the surrounding magnetic field.
#     Furthermore, since the earth magnetic field is usually much smaller than the magnetic fields
#     that occur around and inside electronic devices, the compass mode requires calibration.
#     Maximum Data Acquisition Frequency is 20Hz
#
#     M4G
#     ---
#     The M4G mode is similar to the IMU mode, but instead of using the gyroscope signal to detect
#     rotation, the changing orientation of the magnetometer in the magnetic field is used. This mode
#     is less power consuming in comparison to the IMU mode. There are no drift effects in this mode
#     which are inherent to the gyroscope. The measurement accuracy depends on the stability of the
#     surrounding magnetic field. For this mode, no magnetometer calibration is required and also not
```

```
        #     available. Maximum Data Acquisition Frequency is 50Hz
        #
        #     NDOF_FMC_OFF
        #     ------------
        #     This fusion mode is same as NDOF mode, but with the Fast Magnetometer Calibration turned 'OFF'.
        #     Maximum Data Acquisition Frequency is 100Hz
        #
        #     NDOF
        #     ----
        #     This is a fusion mode with 9 degrees of freedom where the fused absolute orientation data is
        #     calculated from accelerometer, gyroscope and magnetometer. This mode has high data throughput
        #     and high resilience from magnetic field distortions. In this mode, the Fast Magnetometer
        #     calibration is turned ON thereby resulting in quick calibration of the magnetometer and higher
        #     output data accuracy. The current consumption is slightly higher in comparison to the
        #     NDOF_FMC_OFF fusion mode. Maximum Data Acquisition Frequency is 100Hz
        PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.OPR_MODE, 0B00001100); Sleep(0.019)
        #                                                                                 ▲▲▲▲▲▲▲▲
        #                                                                                 ||||||||
        #                                                                                 |||||||└──> ┐            ┌ 0000 = CONFIGMODE, 0001 = ACCONLY, 0010 = MAGONLY, 0011 =
GYROONLY,
        #                                                                                 ||||||└──> |            | 0100 = ACCMAG, 0101 = ACCGYRO, 0110 = MAGGYRO, 0111 = AMG,
        #                                                                                 |||||└──> ├ OPR_MODE ─┤ 1000 = IMU, 1001 = COMPASS, 1010 = M4G, 1011 = NDOF_FMC_OF
F,
        #                                                                                 ||||└──> ┘            └ 1100 = NDOF, Others = Undefined
        #                                                                                 ||||
        #                                                                                 |||└──────> Reserved
        #                                                                                 ||└──────> Reserved
        #                                                                                 |└──────> Reserved
        #                                                                                 └──────> Reserved
        if Verbose: print(" Success!"); pass
        if Verbose: print("    > Initialized the BNO055 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID)+"\n"); pass
```

```
    # Todo: Add the calibration routine
    pass

def __del__(Self):
    print("    > Powering down the BNO055 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BNO055.SYS_TRIGGER, 0x20)
    pass

# Updates the object cache with the current
# measurement values from the sensor
def UpdateInternalCache(Self):
    ADCDataBank = PolyBus.RxRegisterBlock(Self.ChannelID, Self.DeviceAddress, BNO055.ACC_DATA_X_LSB, 32, False)

    Self.ADC_AccelerometerRawX = (ADCDataBank[1]<<8 | ADCDataBank[0]) & 0xFFFF
    Self.ADC_AccelerometerRawY = (ADCDataBank[3]<<8 | ADCDataBank[2]) & 0xFFFF
    Self.ADC_AccelerometerRawZ = (ADCDataBank[5]<<8 | ADCDataBank[4]) & 0xFFFF

    Self.ADC_MagnetometerRawX = (ADCDataBank[7]<<8 | ADCDataBank[6]) & 0xFFFF
    Self.ADC_MagnetometerRawY = (ADCDataBank[9]<<8 | ADCDataBank[8]) & 0xFFFF
    Self.ADC_MagnetometerRawZ = (ADCDataBank[11]<<8 | ADCDataBank[10]) & 0xFFFF

    Self.ADC_GyroscopeRawX = (ADCDataBank[13]<<8 | ADCDataBank[12]) & 0xFFFF
    Self.ADC_GyroscopeRawY = (ADCDataBank[15]<<8 | ADCDataBank[14]) & 0xFFFF
    Self.ADC_GyroscopeRawZ = (ADCDataBank[17]<<8 | ADCDataBank[16]) & 0xFFFF

    Self.ADC_AttitudeRoll = (ADCDataBank[21]<<8 | ADCDataBank[20]) & 0xFFFF
    Self.ADC_AttitudePitch = (ADCDataBank[23]<<8 | ADCDataBank[22]) & 0xFFFF
    Self.ADC_AttitudeYaw = (ADCDataBank[19]<<8 | ADCDataBank[18]) & 0xFFFF

    Self.ADC_QuaternionW = (ADCDataBank[25]<<8 | ADCDataBank[24]) & 0xFFFF
```

```
    Self.ADC_QuaternionX = (ADCDataBank[27]<<8 | ADCDataBank[26]) & 0xFFFF
    Self.ADC_QuaternionY = (ADCDataBank[29]<<8 | ADCDataBank[28]) & 0xFFFF
    Self.ADC_QuaternionZ = (ADCDataBank[31]<<8 | ADCDataBank[30]) & 0xFFFF

    ADCDataBank = PolyBus.RxRegisterBlock(Self.ChannelID, Self.DeviceAddress, BNO055.LIA_DATA_X_LSB, 13, False)

    Self.LinearAccelerationX = (ADCDataBank[1]<<8 | ADCDataBank[0]) & 0xFFFF
    Self.LinearAccelerationY = (ADCDataBank[3]<<8 | ADCDataBank[2]) & 0xFFFF
    Self.LinearAccelerationZ = (ADCDataBank[5]<<8 | ADCDataBank[4]) & 0xFFFF

    Self.GravityVectorX = (ADCDataBank[7]<<8 | ADCDataBank[6]) & 0xFFFF
    Self.GravityVectorY = (ADCDataBank[9]<<8 | ADCDataBank[8]) & 0xFFFF
    Self.GravityVectorZ = (ADCDataBank[11]<<8 | ADCDataBank[10]) & 0xFFFF

    Self.ADC_Thermometer = ADCDataBank[12] & 0xFF
    return True

def GetMeasurementCache(Self, Type_A, Type_B):
    if Type_A.upper()=="ACCELERATION":
        return Self.GetAccelerationCache(Type_B)

    elif Type_A.upper()=="ANGULARVELOCITY":
        return Self.GetAngularVelocityCache(Type_B)

    elif Type_A.upper()=="MAGNETICFIELD":
        return Self.GetMagneticFieldCache(Type_B)

    elif Type_A.upper()=="ATTITUDE":
        return Self.GetAttitudeCache(Type_B)

    elif Type_A.upper()=="TEMPERATURE":
```

```
        return Self.GetTemperatureCache(Type_B)

    else: raise ValueError("ERROR: "+str(Type_A)+" is not recognized as a valid measurement type\n-----\n")

# Read and return the linear acceleration components from
# one of the accelerometer caches as a tuple of x, y, z
def GetAccelerationCache(Self, Type="Linear"):
    Type = Type.upper()

    if Type=="RAW":
        if Self.ADC_AccelerometerRawX>32767:
            Self.ADC_AccelerometerRawX -= 65536
            pass

        if Self.ADC_AccelerometerRawY>32767:
            Self.ADC_AccelerometerRawY -= 65536
            pass

        if Self.ADC_AccelerometerRawZ>32767:
            Self.ADC_AccelerometerRawZ -= 65536
            pass

        return Self.ADC_AccelerometerRawX/100.0, Self.ADC_AccelerometerRawY/100.0, Self.ADC_AccelerometerRawZ/100.0

    elif Type=="LINEAR":
        if Self.LinearAccelerationX>32767:
            Self.LinearAccelerationX -= 65536
            pass

        if Self.LinearAccelerationY>32767:
            Self.LinearAccelerationY -= 65536
```

```
            pass

        if Self.LinearAccelerationZ>32767:
            Self.LinearAccelerationZ -= 65536
            pass

        return Self.LinearAccelerationX/100.0, Self.LinearAccelerationY/100.0, Self.LinearAccelerationZ/100.0

    elif Type=="GRAVITY":
        if Self.GravityVectorX>32767:
            Self.GravityVectorX -= 65536
            pass

        if Self.GravityVectorY>32767:
            Self.GravityVectorY -= 65536
            pass

        if Self.GravityVectorZ>32767:
            Self.GravityVectorZ -= 65536
            pass

        return Self.GravityVectorX/100.0, Self.GravityVectorY/100.0, Self.GravityVectorZ/100.0

    else: raise ValueError("ERROR: "+str(Type)+" is not recognized as a valid accelerometer data type\n-----\n")

# Read and return the angular velocity components from
# the gyroscope cache as a tuple of x, y, z
def GetAngularVelocityCache(Self, Type="Raw"):
    Type = Type.upper()

    if Type=="RAW":
```

```
        if Self.ADC_GyroscopeRawX>32767:
            Self.ADC_GyroscopeRawX -= 65536
            pass

        if Self.ADC_GyroscopeRawY>32767:
            Self.ADC_GyroscopeRawY -= 65536
            pass

        if Self.ADC_GyroscopeRawZ>32767:
            Self.ADC_GyroscopeRawZ -= 65536
            pass

        return Self.ADC_GyroscopeRawX/16.0, Self.ADC_GyroscopeRawY/16.0, Self.ADC_GyroscopeRawZ/16.0

    else: raise ValueError("ERROR: "+str(Type)+" is not recognized as a valid gyroscope data type\n-----\n")

# Read and return the magnetic field components from
# the magnetometer cache as a tuple of x, y, z
def GetMagneticFieldCache(Self, Type="Raw"):
    Type = Type.upper()

    if Type=="RAW":
        if Self.ADC_MagnetometerRawX>32767:
            Self.ADC_MagnetometerRawX -= 65536
            pass

        if Self.ADC_MagnetometerRawY>32767:
            Self.ADC_MagnetometerRawY -= 65536
            pass

        if Self.ADC_MagnetometerRawZ>32767:
```

```
            Self.ADC_MagnetometerRawZ -= 65536
            pass

        return Self.ADC_MagnetometerRawX/16.0, Self.ADC_MagnetometerRawY/16.0, Self.ADC_MagnetometerRawZ/16.0

    else: raise ValueError("ERROR: "+str(Type)+" is not recognized as a valid magnetometer data type\n-----\n")

# Read and return the sensor attitude from the internal cache as either
# Euler angles (tuple of roll, pitch, yaw) (default)
# or quaternions (tuple of w, x, y, z)
def GetAttitudeCache(Self, Format="Euler"):
    Format = Format.upper()

    if Format=="EULER":
        if Self.ADC_AttitudeRoll>32767:
            Self.ADC_AttitudeRoll -= 65536
            pass

        if Self.ADC_AttitudePitch>32767:
            Self.ADC_AttitudePitch -= 65536
            pass

        if Self.ADC_AttitudeYaw>32767:
            Self.ADC_AttitudeYaw -= 65536
            pass

        return Self.ADC_AttitudeRoll/16.0, Self.ADC_AttitudePitch/16.0, Self.ADC_AttitudeYaw/16.0

    elif Format=="QUATERNION":
        Scaler = 1.0/(1<<14)
```

```
        if Self.ADC_QuaternionW>32767:
            Self.ADC_QuaternionW -= 65536
            pass

        if Self.ADC_QuaternionX>32767:
            Self.ADC_QuaternionX -= 65536
            pass

        if Self.ADC_QuaternionY>32767:
            Self.ADC_QuaternionY -= 65536
            pass

        if Self.ADC_QuaternionZ>32767:
            Self.ADC_QuaternionZ -= 65536
            pass

        return Self.ADC_QuaternionW*Scaler, Self.ADC_QuaternionX*Scaler, Self.ADC_QuaternionY*Scaler, Self.ADC_QuaternionZ*Scaler

    else: raise ValueError("ERROR: "+str(Format)+" is not recognized as a valid attitude angle data format\n-----\n")

# Read and return the temperature value in either kelvin or
# celsius (default) or fahrenheit from the thermometer cache
def GetTemperatureCache(Self, Unit="Celsius"):
    Unit = Unit.upper()
    if Self.TEMP>127:
        Self.TEMP -= 256
        pass

    if Unit=="CELSIUS": return Self.TEMP
    elif Unit=="FAHRENHEIT": return (Self.TEMP * 9.0/5.0) + 32
    else: raise ValueError("ERROR: "+str(Unit)+" is not recognized as a valid thermometer data unit\n-----\n")
```

BME280.py

```
# Dependencies
from math import log
from time import sleep as Sleep

import Backend.PolyBus as PolyBus

class BME280:

    # Device Characteristics
    NAME = "BME280"
    MEASUREMENTS = ["HUMIDITY|RELATIVE", "PRESSURE", "TEMPERATURE"]

    # Registers
    HUM_LSB    = 0xFE
    HUM_MSB    = 0xFD
    TEMP_XLSB  = 0xFC
    TEMP_LSB   = 0xFB
    TEMP_MSB   = 0xFA
    PRESS_XLSB = 0xF9
    PRESS_LSB  = 0xF8
    PRESS_MSB  = 0xF7
    CONFIG     = 0xF5
    CTRL_MEAS  = 0xF4
    STATUS     = 0xF3
    CTRL_HUM   = 0xF2
    CALIB_00   = 0x88
    CALIB_01   = 0x89
    CALIB_02   = 0x8A
    CALIB_03   = 0x8B
```

```
CALIB_04   = 0x8C
CALIB_05   = 0x8D
CALIB_06   = 0x8E
CALIB_07   = 0x8F
CALIB_08   = 0x90
CALIB_09   = 0x91
CALIB_10   = 0x92
CALIB_11   = 0x93
CALIB_12   = 0x94
CALIB_13   = 0x95
CALIB_14   = 0x96
CALIB_15   = 0x97
CALIB_16   = 0x98
CALIB_17   = 0x99
CALIB_18   = 0x9A
CALIB_19   = 0x9B
CALIB_20   = 0x9C
CALIB_21   = 0x9D
CALIB_22   = 0x9E
CALIB_23   = 0x9F
CALIB_24   = 0xA0
CALIB_25   = 0xA1
CALIB_26   = 0xE1
CALIB_27   = 0xE2
CALIB_28   = 0xE3
CALIB_29   = 0xE4
CALIB_30   = 0xE5
CALIB_31   = 0xE6
CALIB_32   = 0xE7
CALIB_33   = 0xE8
CALIB_34   = 0xE9
```

```
    CALIB_35   = 0xEA
    CALIB_36   = 0xEB
    CALIB_37   = 0xEC
    CALIB_38   = 0xED
    CALIB_39   = 0xEE
    CALIB_40   = 0xEF
    CALIB_41   = 0xF0
    RESET      = 0xE0
    ID         = 0xD0

    # Hardware & Software Initializer routine
    def __init__(Self, ChannelID, DeviceAddress, Verbose=True):
        if Verbose: print("    > Initializing the BME280 at 0x"+format(DeviceAddress, "02X")+" on channel "+str(ChannelID)); pass
        Self.ChannelID = ChannelID
        Self.DeviceAddress = DeviceAddress

        # Reset the device
        if Verbose: print("      > Resetting the device. . .", end=""); pass
        PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.RESET, 0xB6); Sleep(1.0)
        if Verbose: print(" Success!"); pass

        # Load the calibration data for the device
        if Verbose: print("      > Loading the calibration cache from the device. . .", end=""); pass
        Self.DIG_T1 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_00, False)
        Self.DIG_T2 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_02, True)
        Self.DIG_T3 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_04, True)

        Self.DIG_P1 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_06, False)
        Self.DIG_P2 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_08, True)
        Self.DIG_P3 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_10, True)
        Self.DIG_P4 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_12, True)
```

```
Self.DIG_P5 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_14, True)
Self.DIG_P6 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_16, True)
Self.DIG_P7 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_18, True)
Self.DIG_P8 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_20, True)
Self.DIG_P9 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_22, True)

Self.DIG_H1 = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_25, False)
Self.DIG_H2 = PolyBus.RxRegisterWord(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_26, True)
Self.DIG_H3 = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_28, False)
Self.DIG_H4 = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_29, True)
Self.DIG_H4 = Self.DIG_H4 << 4
Self.DIG_H4 = Self.DIG_H4 | (PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_30, False)&0x0F)
Self.DIG_H5 = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_31, True)
Self.DIG_H5 = Self.DIG_H5 << 4
Self.DIG_H5 = Self.DIG_H5 | ((PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_30, False)>>4)&0x0F)
Self.DIG_H6 = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CALIB_32, True)
if Verbose: print(" Success!"); pass

# Put the device to sleep mode to allow for its configuration
if Verbose: print("      > Preparing the device for configuration. . .", end=""); pass
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
if Verbose: print(" Success!"); pass

if Verbose: print("        > Configuring the Rate, Filter and Interface parameters. . .", end=""); pass
# Set the rate, filter and interface options
#   Data acquisition rate:
#     Higher acquisition rate -> More power consumption, more up-to-date data
#     Lower acquisition rate --> Less power consumption, less up-to-date data
#
#   IIR Filter coefficient:
#     Higher filter coefficient -> Less noise, slower sensor input response
```

```
#     Lower filter coefficient --> High noise, faster sensor input response
#
#   Interface options:
#     Enable/Disable the SPI interface
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CONFIG, 0B00001000)
#                                                                          ▲▲▲▲▲▲▲▲
#                                                                          ||||||||
#                                                                          |||||||└──> 0 = SPI disabled, 1 = SPI enabled
#                                                                          ||||||
#                                                                          ||||||└──> Don't care
#                                                                          |||||
#                                                                          |||||└────> ┐ ┌ 000 = No filter, 001 = 2, ┐
#                                                                          ||||└─────> ├─┤ 010 = 4,         011 = 8, ├──> IIR Filter coefficient
#                                                                          |||└──────> ┘ └ 100, others = 16         ┘
#                                                                          |||
#                                                                          ||└──────> ┐ ┌ 000 = 0.5ms, 001 = 62.5ms, 010 = 125ms,  ┐
#                                                                          |└──────> ├─┤ 011 = 250ms, 100 = 500ms,  101 = 1000ms, ├──> Inactive/Standby duration
#                                                                          └───────> ┘ └ 110 = 10ms,  111 = 20ms                  ┘
Self.CONFIG_Cache = 0B00001000
Sleep(0.05)
if Verbose: print(" Success!"); pass

if Verbose: print("        > Configuring the Hygrometer Oversampling parameter. . .", end=""); pass
# Set the Hygrometer oversampling rate
#   Higher oversampling -> More power consumption, less noise in measurement, more restricted measurement rate
#   Lower oversampling --> Less power consumption, more noise in measurement, less restricted measurement rate
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_HUM, 0B00000101)
#                                                                           ▲▲▲▲▲▲▲▲
#                                                                           ||||||||
#                                                                           |||||||└──> ┐ ┌ 000 = No Oversampling, 001 = 1x Oversampling,
#                                                                           ||||||└──> ├─┤ 010 = 2x Oversampling, 011 = 4x Oversampling
```

```
        #                                                                                                  |||||└─────> ┘ └ 100 = 8x Oversampling, 101, others = 16x Oversampling
        #                                                                                                  ||||└──────> Don't care
        #                                                                                                  |||└──────> Don't care
        #                                                                                                  ||└──────> Don't care
        #                                                                                                  |└──────> Don't care
        #                                                                                                  └──────> Don't care
        Self.CTRL_HUM_Cache = 0B00000101
        Sleep(0.05)
        if Verbose: print(" Success!"); pass

        if Verbose: print("        > Configuring the Thermometer and Barometer Oversampling parameters. . .", end=""); pass
        # Set the sensor mode
        #   Modes:
        #     Sleep mode:: No operation, all registers accessible, lowest power, selected after startup
        #     Forced mode: Perform one measurement, store results and return to sleep mode
        #     Normal mode: Perpetual cycling of measurements and inactive periods
        #
        # Set the Thermometer and Barometer oversampling rates
        #   Higher oversampling -> More power consumption, less noise in measurement, more restricted measurement rate
        #   Lower oversampling --> Less power consumption, more noise in measurement, less restricted measurement rate
        PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B10110111)
        #                                                                                                  ▲▲▲▲▲▲▲▲
        #                                                                                                  ||||||||
        #                                                                                                  |||||||└──> ┐ 00 = Sleep mode, 01 & 10 = Forced mode
        #                                                                                                  ||||||└──> ┘ 11 = Normal mode
        #                                                                                                  ||||||
        #                                                                                                  |||||└──> ┐ ┌ 000 = No Oversampling, 001 = 1x Oversampling,          ┐
        #                                                                                                  ||||└──> ├─┤ 010 = 2x Oversampling, 011 = 4x Oversampling           ├──> Barometer
oversampling
        #                                                                                                  |||└──────> ┘ └ 100 = 8x Oversampling, 101, others = 16x Oversampling ┘
        #                                                                                                  |||
```

```
        #                                                                                ||└──────> ┐ ┌ 000 = No Oversampling, 001 = 1x Oversampling,         ┐
        #                                                                                |└──────> ├─┤ 010 = 2x Oversampling, 011 = 4x Oversampling          ├──> Thermomet
er oversampling
        #                                                                                └───────> ┘ └ 100 = 8x Oversampling, 101, others = 16x Oversampling ┘
        Self.CTRL_MEAS_Cache = 0B10110111
        Sleep(0.05)
        if Verbose:
            print(" Success!")
            print("      > Resumed systematic operation")
            print("    > Initialized the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID)+"\n")
            pass
        pass

    def __del__(Self):
        print("    > Powering down the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
        PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.RESET, 0xB6)
        pass

    def SetMeasurementCycleFrequency(Self, Frequency, Verbose=True):
        if Verbose:
            print("Configuring the Measurement Cycle Frequency of the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
            print("  > Processing the configuration request. . .", end="")
            pass

        SupportedOversampling = [0, 1, 2, 4, 8, 16]
        SupportedStandbyTime = [0.5, 62.5, 125, 250, 500, 1000, 10, 20]
        HygrometerOversampling = Self.CTRL_HUM_Cache & 0B00000111
        HygrometerOversampling = SupportedOversampling[HygrometerOversampling]
        BarometerOversampling = (Self.CTRL_MEAS_Cache & 0B00011100) >> 2
        BarometerOversampling = SupportedOversampling[BarometerOversampling]
        ThermometerOversampling = (Self.CTRL_MEAS_Cache & 0B11100000) >> 5
```

```
    ThermometerOversampling = SupportedOversampling[ThermometerOversampling]
    MeasurementTime = 1 + 2*ThermometerOversampling + 2*BarometerOversampling+0.5 + 2*HygrometerOversampling+0.5
    RequiredStandbyTime = 1000/Frequency - MeasurementTime
    StandbyTimeDeltas = [abs(RequiredStandbyTime-StandbyTime) for StandbyTime in SupportedStandbyTime]
    RequiredStandbyTime = SupportedStandbyTime[StandbyTimeDeltas.index(min(StandbyTimeDeltas))]
    ControlByteFragment = SupportedStandbyTime.index(RequiredStandbyTime)
    if Verbose:
        print(" Success!")
        print("  > Preparing the device for configuration. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
    if Verbose: print(" Success!"); pass
    Self.CONFIG_Cache = (Self.CONFIG_Cache & 0B00011111) | (ControlByteFragment << 5)
    if Verbose: print("    > Configuring the Standby Duration parameter. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CONFIG, Self.CONFIG_Cache)
    if Verbose:
        print(" Success!")
        print("  > Resuming systematic operation. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, Self.CTRL_MEAS_Cache)
    if Verbose:
        print(" Success!")
        print("Configured the BME280 with a Measurement Cycle Frequency of "+format(1000.0/float(MeasurementTime+RequiredStandbyTime),".2f")+"Hz")
        pass

    return True

# Updates the object cache with the current measurement values from the sensor
def UpdateInternalCache(Self, Block=True):
```

```
    if Block:
        while PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.STATUS, False) & 0x08: pass
        ADCDataBank = PolyBus.RxRegisterBlock(Self.ChannelID, Self.DeviceAddress, BME280.PRESS_MSB, 8, False)
        Self.ADC_Hygrometer = (ADCDataBank[6]<<8) | ADCDataBank[7]
        Self.ADC_Barometer = ((ADCDataBank[0]<<16) | (ADCDataBank[1]<<8) | ADCDataBank[2]) >> 4
        Self.ADC_Thermometer = ((ADCDataBank[3]<<16) | (ADCDataBank[4]<<8) | ADCDataBank[5]) >> 4
        return True

    elif not PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.STATUS, False) & 0x08:
        ADCDataBank = PolyBus.RxRegisterBlock(Self.ChannelID, Self.DeviceAddress, BME280.PRESS_MSB, 8, False)
        Self.ADC_Hygrometer = (ADCDataBank[6]<<8) | ADCDataBank[7]
        Self.ADC_Barometer = ((ADCDataBank[0]<<16) | (ADCDataBank[1]<<8) | ADCDataBank[2]) >> 4
        Self.ADC_Thermometer = ((ADCDataBank[3]<<16) | (ADCDataBank[4]<<8) | ADCDataBank[5]) >> 4
        return True

    else:
        return False

def GetMeasurementCache(Self, Type_A, Type_B=None):
    if Type_A.upper()=="HUMIDITY": return Self.GetHumidityCache()
    elif Type_A.upper()=="PRESSURE": return Self.GetPressureCache()
    elif Type_A.upper()=="TEMPERATURE": return Self.GetTemperatureCache()
    else: raise ValueError("ERROR: "+str(Type_A)+" is not recognized as a valid measurement type\n-----\n")

# Set the hygrometer oversampling multiplier
def SetHygrometerOversampling(Self, RequiredOversampling, Verbose=True):
    if Verbose:
        print("Configuring the Hygrometer Oversampling multiplier of the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
        print("  > Processing the configuration request. . .", end="")
        pass
```

```
    SupportedOversampling = [0, 1, 2, 4, 8, 16]
    OversamplingDeltas = [abs(RequiredOversampling-Oversampling) for Oversampling in SupportedOversampling]
    RequiredOversampling = SupportedOversampling[OversamplingDeltas.index(min(OversamplingDeltas))]
    if RequiredOversampling==0: ControlByteFragment=0; pass
    else: ControlByteFragment = int(log(RequiredOversampling, 2) + 1); pass
    if Verbose:
        print(" Success!")
        print("  > Preparing the device for configuration. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
    if Verbose: print(" Success!"); pass
    Self.CTRL_HUM_Cache = (Self.CTRL_HUM_Cache & 0B11111000) | ControlByteFragment
    if Verbose: print("    > Configuring the Hygrometer Oversampling parameter. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_HUM, Self.CTRL_HUM_Cache)
    if Verbose:
        print(" Success!")
        print("  > Resuming systematic operation. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, Self.CTRL_MEAS_Cache)
    if Verbose:
        print(" Success!")
        print("Configured the BME280 with a Hygrometer Oversampling multiplier of x"+str(RequiredOversampling))
        pass

    return True

# Set the IIR filter co-efficient for the hygrometer
def SetHygrometerFilterCoefficient(Self, RequiredFilterCoefficient, Verbose=True):
    if Verbose: print("WARNING: The BME280 does not support configuring its Hygrometer Filter Coefficient\n-------\n"); pass
```

```
    return False

# Read and return the compensated relative humidity (%) from the hygrometer cache
def GetHumidityCache(Self):
    Temperature = float(Self.ADC_Thermometer)
    Stage_1 = (Temperature/16384.0 - float(Self.DIG_T1)/1024.0) * float(Self.DIG_T2)
    Stage_2 = ((Temperature/131072.0 - float(Self.DIG_T1)/8192.0) * (Temperature/131072.0 - float(Self.DIG_T1)/8192.0)) * float(Self.DIG_T3)
    Temperature = int(Stage_1 + Stage_2)

    Humidity = float(Temperature) - 76800.0
    Humidity = (float(Self.ADC_Hygrometer) - (float(Self.DIG_H4)*64.0 + float(Self.DIG_H5)/16384.0*Humidity)) * (
    float(Self.DIG_H2)/65536.0 * (1.0 + float(Self.DIG_H6)/67108864.0*Humidity * (
    1.0 + float(Self.DIG_H3)/67108864.0*Humidity)))
    Humidity = Humidity * (1.0 - float(Self.DIG_H1)*Humidity/524288.0)
    if Humidity>100: Humidity = 100; pass
    elif Humidity<0: Humidity = 0; pass
    return Humidity

# Set the barometer oversampling multiplier
def SetBarometerOversampling(Self, RequiredOversampling, Verbose=True):
    if Verbose:
        print("Configuring the Barometer Oversampling multiplier of the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
        print("  > Processing the configuration request. . .", end="")
        pass

    SupportedOversampling = [0, 1, 2, 4, 8, 16]
    OversamplingDeltas = [abs(RequiredOversampling-Oversampling) for Oversampling in SupportedOversampling]
    RequiredOversampling = SupportedOversampling[OversamplingDeltas.index(min(OversamplingDeltas))]
    if RequiredOversampling==0: ControlByteFragment=0; pass
    else: ControlByteFragment = int(log(RequiredOversampling, 2) + 1); pass
    if Verbose:
```

```
        print(" Success!")
        print("  > Preparing the device for configuration. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
    if Verbose: print(" Success!"); pass
    Self.CTRL_MEAS_Cache = (Self.CTRL_MEAS_Cache & 0B11100011) | (ControlByteFragment<<2)
    if Verbose: print("    > Configuring the Barometer Oversampling parameter. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, Self.CTRL_MEAS_Cache)
    if Verbose:
        print(" Success!")
        print("  > Resumed systematic operation")
        print("Configured the BME280 with a Barometer Oversampling multiplier of x"+str(RequiredOversampling))
        pass

    return True

# Set the IIR filter co-efficient for the barometer
def SetBarometerFilterCoefficient(Self, RequiredFilterCoefficient, Verbose=True):
    if Verbose:
        print("Configuring the Barometer Filter Coefficient of the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
        print("  > Processing the configuration request. . .", end="")
        pass

    SupportedFilterCoefficient = [1, 2, 4, 8, 16]
    FilterCoefficientDeltas = [abs(RequiredFilterCoefficient-FilterCoefficient) for FilterCoefficient in SupportedFilterCoefficient]
    RequiredFilterCoefficient = SupportedFilterCoefficient[FilterCoefficientDeltas.index(min(FilterCoefficientDeltas))]
    ControlByteFragment = int(log(RequiredFilterCoefficient, 2))
    if Verbose:
        print(" Success!")
        print("  > Preparing the device for configuration. . .", end="")
```

```
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
    if Verbose: print(" Success!"); pass
    Self.CONFIG_Cache = (Self.CONFIG_Cache & 0B11100011) | (ControlByteFragment<<2)
    if Verbose: print("    > Configuring the Barometer Filter Coefficient parameter. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CONFIG, Self.CONFIG_Cache)
    if Verbose:
        print(" Success!")
        print("  > Resuming systematic operation. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, Self.CTRL_MEAS_Cache)
    if Verbose:
        print(" Success!")
        print("Configured the BME280 with a Barometer Filter Coefficient of "+str(RequiredFilterCoefficient))
        pass

    return True

# Read and return the compensated pressure (Pa) from the barometer cache
def GetPressureCache(Self):
    Temperature = float(Self.ADC_Thermometer)
    Stage_1 = (Temperature/16384.0 - float(Self.DIG_T1)/1024.0) * float(Self.DIG_T2)
    Stage_2 = ((Temperature/131072.0 - float(Self.DIG_T1)/8192.0) * (Temperature/131072.0 - float(Self.DIG_T1)/8192.0)) * float(Self.DIG_T3)
    Temperature = int(Stage_1 + Stage_2)

    Pressure = float(Self.ADC_Barometer)
    Stage_1 = float(Temperature)/2.0 - 64000.0
    Stage_2 = Stage_1*Stage_1 * float(Self.DIG_P6)/32768.0
    Stage_2 = Stage_2 + Stage_1*float(Self.DIG_P5)*2.0
```

```
    Stage_2 = Stage_2/4.0 + float(Self.DIG_P4)*65536.0
    Stage_1 = (float(Self.DIG_P3)*Stage_1*Stage_1/524288.0 + float(Self.DIG_P2)*Stage_1) / 524288.0
    Stage_1 = (1.0 + Stage_1/32768.0) * float(Self.DIG_P1)
    if Stage_1==0: return 0
    Pressure = 1048576.0 - Pressure
    Pressure = ((Pressure - Stage_2/4096.0) * 6250.0) / Stage_1
    Stage_1 = float(Self.DIG_P9)*Pressure*Pressure / 2147483648.0
    Stage_2 = Pressure*float(Self.DIG_P8) / 32768.0
    Pressure = Pressure + (Stage_1+Stage_2+float(Self.DIG_P7)) / 16.0
    return Pressure

# Set the thermometer oversampling multiplier
def SetThermometerOversampling(Self, RequiredOversampling, Verbose=True):
    if Verbose:
        print("Configuring the Thermometer Oversampling multiplier of the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
        print("  > Processing the configuration request. . .", end="")
        pass

    SupportedOversampling = [0, 1, 2, 4, 8, 16]
    OversamplingDeltas = [abs(RequiredOversampling-Oversampling) for Oversampling in SupportedOversampling]
    RequiredOversampling = SupportedOversampling[OversamplingDeltas.index(min(OversamplingDeltas))]
    if RequiredOversampling==0: ControlByteFragment=0; pass
    else: ControlByteFragment = int(log(RequiredOversampling, 2) + 1); pass
    if Verbose:
        print(" Success!")
        print("  > Preparing the device for configuration. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
    if Verbose: print(" Success!"); pass
    Self.CTRL_MEAS_Cache = (Self.CTRL_MEAS_Cache & 0B00011111) | (ControlByteFragment<<5)
```

```
    if Verbose: print("    > Configuring the Thermometer Oversampling parameter. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, Self.CTRL_MEAS_Cache)
    if Verbose:
        print(" Success!")
        print("  > Resumed systematic operation")
        print("Configured the BME280 with a Thermometer Oversampling multiplier of x"+str(RequiredOversampling))
        pass

    return True

# Set the IIR filter co-efficient for the thermometer
def SetThermometerFilterCoefficient(Self, RequiredFilterCoefficient, Verbose=True):
    if Verbose:
        print("Configuring the Thermometer Filter Coefficient of the BME280 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
        print("  > Processing the configuration request. . .", end="")
        pass

    SupportedFilterCoefficient = [1, 2, 4, 8, 16]
    FilterCoefficientDeltas = [abs(RequiredFilterCoefficient-FilterCoefficient) for FilterCoefficient in SupportedFilterCoefficient]
    RequiredFilterCoefficient = SupportedFilterCoefficient[FilterCoefficientDeltas.index(min(FilterCoefficientDeltas))]
    ControlByteFragment = int(log(RequiredFilterCoefficient, 2))
    if Verbose:
        print(" Success!")
        print("  > Preparing the device for configuration. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, 0B00100100); Sleep(0.50)
    if Verbose: print(" Success!"); pass
    Self.CONFIG_Cache = (Self.CONFIG_Cache & 0B11100011) | (ControlByteFragment<<2)
    if Verbose: print("    > Configuring the Thermometer Filter Coefficient parameter. . .", end=""); pass
    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CONFIG, Self.CONFIG_Cache)
```

```
    if Verbose:
        print(" Success!")
        print("  > Resuming systematic operation. . .", end="")
        pass

    PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, BME280.CTRL_MEAS, Self.CTRL_MEAS_Cache)
    if Verbose:
        print(" Success!")
        print("Configured the BME280 with a Thermometer Filter Coefficient of "+str(RequiredFilterCoefficient))
        pass

    return True

# Read and return the compensated temperature (°C) from the thermometer cache
def GetTemperatureCache(Self):
    Temperature = float(Self.ADC_Thermometer)
    Stage_1 = (Temperature/16384.0 - float(Self.DIG_T1)/1024.0) * float(Self.DIG_T2)
    Stage_2 = ((Temperature/131072.0 - float(Self.DIG_T1)/8192.0) * (Temperature/131072.0 - float(Self.DIG_T1)/8192.0)) * float(Self.DIG_T3)
    Temperature = (Stage_1+Stage_2) / 5120.0
    return Temperature
```

```
# Dependencies
from time import sleep as Sleep

import Backend.PolyBus as PolyBus

class CCS811:

    # Device Characteristics
    NAME = "CCS811"
    MEASUREMENTS = ["ECO2", "TVOC"]

    # Registers
    STATUS          = 0x00
    MEAS_MODE       = 0x01
    ALG_RESULT_DATA = 0x02
    RAW_DATA        = 0x03
    ENV_DATA        = 0x05
    NTC             = 0x06
    THRESHOLDS      = 0x10
    BASELINE        = 0x11
    HW_ID           = 0x20
    HW_VERSION      = 0x21
    FW_BOOT_VERSION = 0x23
    FW_APP_VERSION  = 0x24
    ERROR_ID        = 0xE0
    APP_ERASE       = 0xF1
    APP_DATA        = 0xF2
    APP_VERIFY      = 0xF3
    APP_START       = 0xF4
```

```
SW_RESET        = 0xFF

# Hardware & Software Initializer routine
def __init__(Self, ChannelID, DeviceAddress, Verbose=True):
    if Verbose: print("    > Initializing the CCS811 at 0x"+format(DeviceAddress, "02X")+" on channel "+str(ChannelID)); pass
    Self.ChannelID = ChannelID
    Self.DeviceAddress = DeviceAddress

    # Reset the device
    if Verbose: print("      > Resetting the device. . .", end=""); pass
    PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.SW_RESET, [0x11, 0xE5, 0x72, 0x8A]); Sleep(0.50)

    StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False)
    if (StatusByte&0B10000000)==0B10000000:
        if Verbose: print(" Fail!"); pass
        raise Exception("ERROR: The device did not respond to the Reset command\n-----\n")
    elif Verbose: print(" Success!"); pass

    # Check the application firmware on the device
    if Verbose: print("      > Validating the application firmware on the device. . .", end=""); pass
    if (StatusByte&0B00010000)==0B00000000:
        if Verbose: print(" Fail!"); pass
        raise Exception("ERROR: The device does not have a valid application firmware on ROM\n-----\n")
    elif Verbose: print(" Pass!"); pass

    # Check device for errors
    if Verbose: print("      > Scanning the device for errors. . .", end=""); pass
    if (StatusByte&0B00000001)==0B00000001:
        if Verbose: print(" Fail!"); pass
        ErrorMessage = "ERROR: The device reported one or more error(s)\n-----\n"
        ErrorByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.ERROR_ID, False)
```

```
    if (ErrorByte&0B00000001)==0B00000001: ErrorMessage+="       WRITE_REG_INVALID\n"; pass
    if (ErrorByte&0B00000010)==0B00000010: ErrorMessage+="       READ_REG_INVALID\n"; pass
    if (ErrorByte&0B00000100)==0B00000100: ErrorMessage+="       MEASMODE_INVALID\n"; pass
    if (ErrorByte&0B00001000)==0B00001000: ErrorMessage+="       MAX_RESISTANCE\n"; pass
    if (ErrorByte&0B00010000)==0B00010000: ErrorMessage+="       HEATER_FAULT\n"; pass
    if (ErrorByte&0B00100000)==0B00100000: ErrorMessage+="       HEATER_SUPPLY\n"; pass
    raise Exception(ErrorMessage)
elif Verbose: print(" Pass!"); pass

# Launch the application on the device
if Verbose: print("      > Launching the on-device application. . .", end=""); pass
PolyBus.TxByte(Self.ChannelID, Self.DeviceAddress, CCS811.APP_START); Sleep(1.0)
StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False)
if (StatusByte&0B10000000)==0B00000000:
    if Verbose: print(" Fail!"); pass
    raise Exception("ERROR: The device did not respond to the Application Launch command\n-----\n")
elif Verbose: print(" Success!"); pass

'''
# Check device for errors
if Verbose: print("      > Scanning the device for errors. . .", end=""); pass
if (StatusByte&0B00000001)==0B00000001:
    if Verbose: print(" Fail!"); pass
    ErrorMessage = "ERROR: The device reported one or more error(s)\n-----\n"
    ErrorByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.ERROR_ID, False)
    if (ErrorByte&0B00000001)==0B00000001: ErrorMessage+="       WRITE_REG_INVALID\n"; pass
    if (ErrorByte&0B00000010)==0B00000010: ErrorMessage+="       READ_REG_INVALID\n"; pass
    if (ErrorByte&0B00000100)==0B00000100: ErrorMessage+="       MEASMODE_INVALID\n"; pass
    if (ErrorByte&0B00001000)==0B00001000: ErrorMessage+="       MAX_RESISTANCE\n"; pass
    if (ErrorByte&0B00010000)==0B00010000: ErrorMessage+="       HEATER_FAULT\n"; pass
    if (ErrorByte&0B00100000)==0B00100000: ErrorMessage+="       HEATER_SUPPLY\n"; pass
```

```
    raise Exception(ErrorMessage)
elif Verbose: print(" Pass!"); pass
'''

if Verbose: print("      > Configuring the operation mode of the device. . .", end=""); pass
# Set the sensor mode
#   DRIVE_MODE:
#     Mode 0: Idle (Measurements are disabled in this mode)
#     Mode 1: Constant power mode, IAQ measurement every second
#     Mode 2: Pulse heating mode IAQ measurement every 10 seconds
#     Mode 3: Low power pulse heating mode IAQ measurement every 60 seconds
#     Mode 4: Constant power mode, sensor measurement every 250ms
#
# Enable/Disable interrupt generation
#   INT_DATARDY:
#     0: Interrupt generation is disabled
#     1: The nINT signal is asserted (driven low) when a new sample is ready in
#        ALG_RESULT_DATA. The nINT signal will stop being driven low when
#        ALG_RESULT_DATA is read on the I²C interface
#
# Set interrupt generation behaviour
#   INT_THRESH:
#     0: Interrupt mode (if enabled) operates normally
#     1: Interrupt mode (if enabled) only asserts the nINT signal (driven low) if the new
#        ALG_RESULT_DATA crosses one of the thresholds set in the THRESHOLDS register
#        by more than the hysteresis value (also in the THRESHOLDS register)
PolyBus.TxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.MEAS_MODE, 0B00010000)
#                                                                             ▲▲▲▲▲▲▲▲
#                                                                             ||||||||
#                                                                             |||||||└──> Reserved
#                                                                             ||||||└───> Reserved
```

```
#                                                                              ||||||
#                                                                              |||||└────> INT_THRESH
#                                                                              |||||
#                                                                              ||||└─────> INT_DATARDY
#                                                                              ||||
#                                                                              |||└──────> ┐                ┌ 000 = Mode 0, 001 = Mode 1
#                                                                              ||└───────> ├ DRIVE_MODE ─┤ 010 = Mode 2, 011 = Mode 3
#                                                                              |└────────> ┘                └ 100 = Mode 4, 101, others = Reserved
#                                                                              |
#                                                                              └─────────> Reserved - Assert "0"
Self.MEAS_MODE_Cache = 0B00010000
Sleep(0.1)
if Verbose: print(" Success!"); pass

# Check device for errors
if Verbose: print("      > Scanning the device for errors. . .", end=""); pass
StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False)
if (StatusByte&0B00000001)==0B00000001:
    if Verbose: print(" Fail!"); pass
    ErrorMessage = "ERROR: The device reported one or more error(s)\n-----\n"
    ErrorByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.ERROR_ID, False)
    if (ErrorByte&0B00000001)==0B00000001: ErrorMessage+="       WRITE_REG_INVALID\n"; pass
    if (ErrorByte&0B00000010)==0B00000010: ErrorMessage+="       READ_REG_INVALID\n"; pass
    if (ErrorByte&0B00000100)==0B00000100: ErrorMessage+="       MEASMODE_INVALID\n"; pass
    if (ErrorByte&0B00001000)==0B00001000: ErrorMessage+="       MAX_RESISTANCE\n"; pass
    if (ErrorByte&0B00010000)==0B00010000: ErrorMessage+="       HEATER_FAULT\n"; pass
    if (ErrorByte&0B00100000)==0B00100000: ErrorMessage+="       HEATER_SUPPLY\n"; pass
    raise Exception(ErrorMessage)
elif Verbose: print(" Pass!"); pass

if Verbose: print("    > Initialized the CCS811 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID)+"\n"); pass
```

```
    pass

def __del__(Self):
    print("    > Powering down the CCS811 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
    PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.SW_RESET, [0x11, 0xE5, 0x72, 0x8A])
    pass

def SetCompensationData(Self, Humidity, Temperature):
    BitValue = 128
    HumidityBytes = ""
    for Iterator in range(16):
        if (Humidity-BitValue/2.0)>=0:
            Humidity -= BitValue/2.0
            HumidityBytes += "1"
            pass

        else:
            HumidityBytes += "0"
            pass

        BitValue /= 2.0
        pass

    HumidityBytes = list(map("".join, zip(*[iter(HumidityBytes)]*8)))
    for Iterator in range(len(HumidityBytes)):
        HumidityBytes[Iterator] = int(HumidityBytes[Iterator], 2)
        pass

    BitValue = 128
    Temperature += 25
    TemperatureBytes = ""
```

```
    for Iterator in range(16):
        if (Temperature-BitValue/2.0)>=0:
            Temperature -= BitValue/2.0
            TemperatureBytes += "1"
            pass

        else:
            TemperatureBytes += "0"
            pass

        BitValue /= 2.0
        pass

    TemperatureBytes = list(map("".join, zip(*[iter(TemperatureBytes)]*8)))
    for Iterator in range(len(TemperatureBytes)):
        TemperatureBytes[Iterator] = int(TemperatureBytes[Iterator], 2)
        pass

    CompensationDataBlock = HumidityBytes + TemperatureBytes
    PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.ENV_DATA, CompensationDataBlock)
    return

def UpdateInternalCache(Self, Block=False):
    if Block:
        while not (PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False) & 0x08): pass
        ADCDataBank = PolyBus.RxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.ALG_RESULT_DATA, 4, False)
        Self.ADC_eCO2 = (ADCDataBank[0] << 8) | ADCDataBank[1]
        Self.ADC_TVOC = (ADCDataBank[2] << 8) | ADCDataBank[3]
        return True

    elif PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False) & 0x08:
```

```
        ADCDataBank = PolyBus.RxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.ALG_RESULT_DATA, 4, False)
        Self.ADC_eCO2 = (ADCDataBank[0] << 8) | ADCDataBank[1]
        Self.ADC_TVOC = (ADCDataBank[2] << 8) | ADCDataBank[3]
        return True

    else:
        return False

def GetMeasurementCache(Self, Type_A):
    if Type_A.upper()=="ECO2": return Self.GeteCO2Cache()
    elif Type_A.upper()=="TVOC": return Self.GetTVOCCache()
    else: raise ValueError("ERROR: "+str(Type_A)+" is not recognized as a valid measurement type\n-----\n")

def GeteCO2Cache(Self):
    return Self.ADC_eCO2

def GetTVOCCache(Self):
    return Self.ADC_TVOC

def _FlashFirmware_(Self, FilePath):
    print("CCS811 APPLICATION FIRMWARE FLASHING ROUTINE\n-----------------------------------------\n")
    FirmwareBlocks = []
    Firmware = []
    try:
        with open(FilePath, "rb") as FirmwareFile:
            print("> Reading the Application Firmware from the *.bin file. . .", end="")
            Byte = FirmwareFile.read(1)
            while Byte:
                Firmware.append(int(Byte.hex(), base=16))
                Byte = FirmwareFile.read(1)
                pass
```

```
    print(" Success!")
    print("  > The Application Firmware is "+str(len(Firmware))+" bytes in size")
    if len(Firmware)%8: raise ValueError("\nERROR: The Application Firmware is corrupt\n-----\n")
    pass

FirmwareBlock = []
print("  > Application Firmware raw dump", end="")
for Iterator in range(len(Firmware)):
    if not Iterator%8:
        print("\n   ", end="")
        if FirmwareBlock:
            FirmwareBlocks.append(FirmwareBlock)
            FirmwareBlock = []
            pass
        pass

    print(" 0x"+format(Firmware[Iterator], "02X"), end="")
    FirmwareBlock.append(Firmware[Iterator])
    pass

FirmwareBlocks.append(FirmwareBlock)
Choice = input("\n\nDo you want to proceed with Flashing the Application Firmware? [yes/NO] > ")
Choice = Choice.upper()
if Choice=="" or Choice=="N" or Choice=="NO":
    print("\n> The Application Firmware Flash operation has been aborted")
    return False
elif Choice=="Y" or Choice=="YES":
    print("\n> Preparing to Flash the Application Firmware")
    print("> Resetting the CCS811 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID)+". . .", end="")
    PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.SW_RESET, [0x11, 0xE5, 0x72, 0x8A]); Sleep(1.0)
    StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False)
```

```
if (StatusByte&0B10000000)==0B10000000:
    print(" Fail!")
    return False
else: print(" Success!"); pass
if (StatusByte&0B00010000)==0B00010000: print("  > The device has a pre-validated Application Firmware on it"); pass
else: print("  > The device doesn't have a pre-validated Application Firmware on it"); pass
print("\n> Erasing the Application Firmware on the device. . .", end="")
PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.APP_ERASE, [0xE7, 0xA7, 0xE6, 0x09]); Sleep(1.0)
print(" Success!"); print("  > Resetting the device. . .", end="")
PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.SW_RESET, [0x11, 0xE5, 0x72, 0x8A]); Sleep(1.0)
StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False)
if (StatusByte&0B10000000)==0B10000000: print(" Fail!"); pass
else: print(" Success!"); pass
if (StatusByte&0B00010000)==0B00000000: print("  > Application Firmware erased successfully!"); pass
FirmwareBlockCount = len(FirmwareBlocks)
print("\n> Flashing the new Application Firmware, "+str(FirmwareBlockCount)+" blocks")
Counter = 1
for FirmwareBlock in FirmwareBlocks:
    print("  > Transmitting Application Firmware Block "+str(Counter)+"/"+str(FirmwareBlockCount))
    PolyBus.TxRegisterBlock(Self.ChannelID, Self.DeviceAddress, CCS811.APP_DATA, FirmwareBlock)
    Sleep(0.5)
    Counter += 1
    pass

print("\n> Validating the new Application Firmware. . .", end="")
PolyBus.TxByte(Self.ChannelID, Self.DeviceAddress, CCS811.APP_VERIFY); Sleep(1.0)
StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False)
Counter = 0
while not (StatusByte&0B00010000)==0B00010000:
    StatusByte = PolyBus.RxRegisterByte(Self.ChannelID, Self.DeviceAddress, CCS811.STATUS, False); Sleep(1.0); Counter += 1
    if Counter==60:
```

```
                print(" Fail!")
                return False
            pass

        print(" Success!")
        return True
    else: return False
except FileNotFoundError:
    print("ERROR: '"+FilePath+"' does not exist\n-----\n")
    return False

except ValueError as Error:
    print(Error)
    return False
```

SGP30.py

```
# Dependencies
from time import sleep as Sleep

import Backend.PolyBus as PolyBus

class SGP30:

    # Device Characteristics
    NAME = "SGP30"
    MEASUREMENTS = []

    # Hardware & Software Initializer routine
    def __init__(Self, ChannelID, DeviceAddress, Verbose=True):
        if Verbose: print("Initializing the SPS30 at 0x"+format(DeviceAddress, "02X")+" on channel "+str(ChannelID)); pass
        Self.ChannelID = ChannelID
        Self.DeviceAddress = DeviceAddress
        # Start code
        # End code
        pass

    def CMD_InitAirQuality(Self):
        PolyBus.TxBlock(
            ChannelID=Self.ChannelID,
            DeviceAddress=Self.DeviceAddress,
            Data=[0x20, 0x03]
        )
        return

    def CMD_MeasureAirQuality(Self):
```

```
    return

def CMD_GetBaseline(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x20, 0x15],
        Length=6,
        Signed=False
    )

    if not Self._ComputeCRC8_([Data[0], Data[1]])==Data[2]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")
    if not Self._ComputeCRC8_([Data[3], Data[4]])==Data[5]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")

    return Data

def CMD_SetBaseline(Self, Value):
    PolyBus.TxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x20, 0x1E]+Value
    )
    return

def CMD_SetHumidity(Self, Value):
    Value = Self._EncodeDataPacket_(Value)
    Value = [(Value >> 8), (Value & 0xFF)]
    PolyBus.TxBlock(
        ChannelID=Self.ChannelID,
```

```
        DeviceAddress=Self.DeviceAddress,
        Data=[
            0x20,
            0x61,
            Value[0],
            Value[1],
            Self._ComputeCRC8_([
                Value[0],
                Value[1]
            ])
        ]
    )
    return

def CMD_GetFeatureSetVersion(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x20, 0x2F],
        Length=3,
        Signed=False
    )

    if not Self._ComputeCRC8_([Data[0], Data[1]])==Data[2]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")

    ProductType = (Data[0] & 0xF0) >> 4
    ProductVersion = (Data[1] & 0xE0) >> 5
    ProductRevision = Data[1] & 0x1F
    return ProductType, ProductVersion, ProductRevision
```

```
def CMD_MeasureRawSignals(Self):
    return

def CMD_GetSerialID(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x36, 0x82],
        Length=9,
        Signed=False
    )

    if not Self._ComputeCRC8_([Data[0], Data[1]])==Data[2]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")
    if not Self._ComputeCRC8_([Data[3], Data[4]])==Data[5]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")
    if not Self._ComputeCRC8_([Data[6], Data[7]])==Data[8]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")

    Data = (Data<<40) | (Data<<32) | (Data<<24) | (Data<<16) | (Data<<8) | (Data)

    return Data

def _ComputeCRC8_(Self, Data):
    CRC = 0xFF

    for Iterator in range(len(Data)):
        CRC ^= (Data[Iterator]&0xFF)
        for SubIterator in range(8):
            if CRC&0x80: CRC = (((CRC<<1)^0x31)&0xFF); pass
            else: CRC = (CRC<<1)&0xFF; pass
```

```
            pass
        pass

    return CRC

def _EncodeDataPacket_(Self, DecodedValue):
    BitMask = 128
    EncodedValue = ""
    for Iterator in range(16):
        if DecodedValue>=BitMask:
            DecodedValue -= BitMask
            EncodedValue += '1'
            pass
        else: EncodedValue += '0'; pass
        BitMask /= 2.0
        pass
    return int(EncodedValue, 2)

def _DecodeDataPacket_(Self, EncodedValue):
    BitMask = 0x8000
    DecodedValue = 0
    EncodedValue &= 0xFFFF
    for Iterator in range(16):
        if EncodedValue>=BitMask:
            EncodedValue -= BitMask
            DecodedValue += 2**(7-Iterator)
            pass
        BitMask = BitMask>>1
        pass
    return DecodedValue
```

SPS30.py

```
# Dependencies
from time import sleep as Sleep
import struct

import Backend.PolyBus as PolyBus

class SPS30:

    # Device Characteristics
    NAME = "SPS30"
    MEASUREMENTS = ["PM|TPS", "PM1.0|MC", "PM2.5|MC", "PM4.0|MC", "PM10.0|MC", "PM0.5|NC", "PM1.0|NC", "PM2.5|NC", "PM4.0|NC", "PM10.0|NC"]

    # Hardware & Software Initializer routine
    def __init__(Self, ChannelID, DeviceAddress, Verbose=True):
        if Verbose: print("    > Initializing the SPS30 at 0x"+format(DeviceAddress, "02X")+" on channel "+str(ChannelID)); pass
        Self.ChannelID = ChannelID
        Self.DeviceAddress = DeviceAddress

        if Verbose: print("      > Resetting the device. . .", end=""); pass
        Self.CMD_Reset(); Sleep(1.0)
        if Verbose: print(" Success!"); pass

        if Verbose: print("      > Caching the device's information. . .", end=""); pass
        Self.ArticleCode = Self.CMD_ReadArticleCode()
        Self.SerialNumber = Self.CMD_ReadSerialNumber()
        if Verbose: print(" Success!"); pass

        if Verbose: print("      > Commencing the measurement cycle. . .", end=""); pass
        Self.CMD_StartMeasurement(); Sleep(0.50)
```

```
    if Verbose: print(" Success!"); pass

    if Verbose: print("      > Executing the fan cleaning routine. . .", end=""); pass
    Self.CMD_StartFanCleaning(); Sleep(10.0)
    if Verbose: print(" Success!"); pass

    if Verbose: print("    > Initialized the SPS30 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID)+"\n"); pass
    pass

def __del__(Self):
    print("    > Powering down the SPS30 at 0x"+format(Self.DeviceAddress, "02X")+" on channel "+str(Self.ChannelID))
    Self.CMD_Reset()
    pass

def UpdateInternalCache(Self, Block=False):
    if Block:
        Self._DataReadySemaphore_()
        [
            Self.MCPM1,
            Self.MCPM2_5,
            Self.MCPM4,
            Self.MCPM10,
            Self.NCPM0_5,
            Self.NCPM1,
            Self.NCPM2_5,
            Self.NCPM4,
            Self.NCPM10,
            Self.TypicalParticleSize
        ] = Self.CMD_ReadMeasuredValues()
        return True
```

```
    else:
        Data = Self.CMD_ReadDataReadyFlag()
        if Data[0]==Data[1]:
            return False

        else:
            [
                Self.MCPM1,
                Self.MCPM2_5,
                Self.MCPM4,
                Self.MCPM10,
                Self.NCPM0_5,
                Self.NCPM1,
                Self.NCPM2_5,
                Self.NCPM4,
                Self.NCPM10,
                Self.TypicalParticleSize
            ] = Self.CMD_ReadMeasuredValues()
            return True

def GetMeasurementCache(Self, Type_A, Type_B):
    if Type_A.upper()=="PM0.5":
        if Type_B.upper()=="NC": return Self.NCPM0_5
        pass

    elif Type_A.upper()=="PM1.0":
        if Type_B.upper()=="NC": return Self.NCPM1
        elif Type_B.upper()=="MC": return Self.MCPM1
        pass

    elif Type_A.upper()=="PM2.5":
```

```
        if Type_B.upper()=="NC": return Self.NCPM2_5
        elif Type_B.upper()=="MC": return Self.MCPM2_5
        pass

    elif Type_A.upper()=="PM4.0":
        if Type_B.upper()=="NC": return Self.NCPM4
        elif Type_B.upper()=="MC": return Self.MCPM4
        pass

    elif Type_A.upper()=="PM10.0":
        if Type_B.upper()=="NC": return Self.NCPM10
        elif Type_B.upper()=="MC": return Self.MCPM10
        pass

    elif Type_A.upper()=="PM":
        if Type_B.upper()=="TPS": return Self.TypicalParticleSize[0]
        pass

    else: raise ValueError("ERROR: "+str(Type_A)+"|"+str(Type_B)+" is not recognized as a valid measurement type\n-----\n")

def CMD_StartMeasurement(Self):
    PolyBus.TxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[
            0x00,
            0x10,
            0x03,
            0x00,
            Self._ComputeCRC8_([
                0x03,
```

```
                0x00
            ])
        ]
    )
    return

def CMD_StopMeasurement(Self):
    PolyBus.TxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x01, 0x04]
    )
    return

def CMD_ReadDataReadyFlag(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x02, 0x02],
        Length=3,
        Signed=False
    )

    if not Self._ComputeCRC8_([Data[0], Data[1]])==Data[2]:
        raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")

    return Data

def CMD_ReadMeasuredValues(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
```

```
        DeviceAddress=Self.DeviceAddress,
        Data=[0x03, 0x00],
        Length=60,
        Signed=False
    )
    Readouts = [
        # Mass Concentration PM1.0 [µg/m³]
        # Mass Concentration PM2.5 [µg/m³]
        # Mass Concentration PM4.0 [µg/m³]
        # Mass Concentration PM10 [µg/m³]
        # Number Concentration PM0.5 [#/cm³]
        # Number Concentration PM1.0 [#/cm³]
        # Number Concentration PM2.5 [#/cm³]
        # Number Concentration PM4.0 [#/cm³]
        # Number Concentration PM10 [#/cm³]
        # Typical Particle Size [µm]
    ]
    for Iterator in range(5, 60, 6):
        if (Self._ComputeCRC8_([Data[Iterator-5], Data[Iterator-4]])==Data[Iterator-3]) and (Self._ComputeCRC8_([Data[Iterator-2], Data[Iterator-1]])==Data[Iterator]):
            Readouts.append(struct.unpack(">f", bytes([Data[Iterator-5], Data[Iterator-4], Data[Iterator-2], Data[Iterator-1]])))
            pass
        else:
            raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")
        pass
    return Readouts

def CMD_ReadAutoCleaningInterval(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0x80, 0x04],
```

```
            Length=6,
            Signed=False
        )
        for Iterator in range(2, 6, 3):
            if not Self._ComputeCRC8_([Data[Iterator-2], Data[Iterator-1]])==Data[Iterator]:
                raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")
                pass
            pass
        Value = (Data[0]<<24) | (Data[1]<<16) | (Data[3]<<8) | (Data[4])
        return Value

    def CMD_WriteAutoCleaningInterval(Self, Value):
        if Value>4294967295: raise ValueError("ERROR: "+str(Value)+" is larger than what can be represented by a 32-bit unsigned integer (4,294,967,295)\n-----\n")
        Value &= 0xFFFFFFFF
        DataPackets = [0x80, 0x04]
        DataPackets.append((Value&0xFF000000)>>24)
        DataPackets.append((Value&0x00FF0000)>>16)
        DataPackets.append(Self._ComputeCRC8_([DataPackets[2], DataPackets[3]]))
        DataPackets.append((Value&0x0000FF00)>>8)
        DataPackets.append(Value&0x000000FF)
        DataPackets.append(Self._ComputeCRC8_([DataPackets[5], DataPackets[6]]))
        PolyBus.TxBlock(
            ChannelID=Self.ChannelID,
            DeviceAddress=Self.DeviceAddress,
            Data=DataPackets
        )
        return

    def CMD_StartFanCleaning(Self):
        PolyBus.TxBlock(
            ChannelID=Self.ChannelID,
```

```
        DeviceAddress=Self.DeviceAddress,
        Data=[0x56, 0x07]
    )
    return

def CMD_ReadArticleCode(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
        Data=[0xD0, 0x25],
        Length=48,
        Signed=False
    )

    DataString = ""
    for Iterator in range(2, 48, 3):
        if Self._ComputeCRC8_([Data[Iterator-2], Data[Iterator-1]])==Data[Iterator]:
            DataString += chr(Data[Iterator-2]) + chr(Data[Iterator-1])
            pass

        else:
            raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")

        pass

    return DataString.replace('\0', '')

def CMD_ReadSerialNumber(Self):
    Data = PolyBus.TxRxBlock(
        ChannelID=Self.ChannelID,
        DeviceAddress=Self.DeviceAddress,
```

```
            Data=[0xD0, 0x33],
            Length=48,
            Signed=False
        )

        DataString = ""
        for Iterator in range(2, 48, 3):
            if Self._ComputeCRC8_([Data[Iterator-2], Data[Iterator-1]])==Data[Iterator]:
                DataString += chr(Data[Iterator-2]) + chr(Data[Iterator-1])
                pass

            else:
                raise Exception("ERROR: A cyclic redundancy check failed\n-----\n")

            pass

        return DataString.replace('\0', '')

    def CMD_Reset(Self):
        PolyBus.TxBlock(
            ChannelID=Self.ChannelID,
            DeviceAddress=Self.DeviceAddress,
            Data=[0xD3, 0x04]
        )
        return

    def _DataReadySemaphore_(Self):
        Data = Self.CMD_ReadDataReadyFlag()
        while Data[0]==Data[1]:
            Data = PolyBus.RxBlock(
                ChannelID=Self.ChannelID,
```

```
            DeviceAddress=Self.DeviceAddress,
            Length=3,
            Signed=False
        )
        pass
    return

def _ComputeCRC8_(Self, Data):
    CRC = 0xFF

    for Iterator in range(len(Data)):
        CRC ^= (Data[Iterator]&0xFF)
        for SubIterator in range(8):
            if CRC&0x80: CRC = (((CRC<<1)^0x31)&0xFF); pass
            else: CRC = (CRC<<1)&0xFF; pass
            pass
        pass

    return CRC
```

```
# Low-level dependencies
from digi.xbee.devices import XBeeDevice, RemoteXBeeDevice
from digi.xbee.models.address import XBee64BitAddress
from digi.xbee.exception import TimeoutException
from digi.xbee.exception import XBeeException
from queue import Queue, Empty, Full

# XBee Modem class definition
class XBeeModem:

    StartMarker = str(chr(0x06))
    EndMarker = str(chr(0x15))
    MAX_RETRY_COUNT = 5

    # Soft-Hard initialization routine
    def __init__(Self):
        Self.IncomingTransmissionPayload = ""
        DEFAULT_COORDINATOR_NI = "EM_RSA-"
        SERIAL_PORT_NAME = "/dev/serial0"
        BAUD_RATE = 921600

        print("  > Locating the local XBee coordinator modem on the host serial ports. . .", end="")
        Self.Modem = XBeeDevice(SERIAL_PORT_NAME, BAUD_RATE)
        Self.Modem.open()
        print(" Success!")

        print("  > Validating the configuration settings of the XBee modem. . .", end="")
        if DEFAULT_COORDINATOR_NI not in Self.Modem.get_parameter("NI").decode("UTF-8"): Self.Modem = None; pass
        if not 1==Self.Modem.get_parameter("CE")[0]: Self.Modem = None; pass
```

```
    if Self.Modem is None:
        print(" Fail!")
        raise Exception("ERROR: The system was unable to locate a properly configured local XBee coordinator modem\n-----\n")
    Self.MaxPayloadSize = Self.Modem.get_parameter("NP")[1]
    print(" Success!")

    print("  > Setting up the modem's inbox. . .", end="")
    Self.Inbox = Queue(0)
    print(" Success!")

    print("  > Setting up the modem's Rx Sentinel. . .", end="")
    Self.Modem.add_data_received_callback(Self.DataReceived)
    print(" Success!")
    pass

# Soft de-initialization routine
def __del__(Self):
    try:
        Self.Modem.close()
        Self.Modem = None
        pass
    except Exception:
        pass
    pass

# Data receive function
def RxData(Self):
    try:
        return Self.Inbox.get(False, None)

    except Empty:
```

```
        return None

# Data transmit function
def TxData(Self, DataPacket):
    RemoteAddress, Data = DataPacket
    RemoteDevice = RemoteXBeeDevice(Self.Modem, XBee64BitAddress.from_hex_string(RemoteAddress))
    Counter = 0
    while (XBeeModem.MAX_RETRY_COUNT-Counter)>0:
        try:
            Self.Modem.send_data(RemoteDevice, XBeeModem.StartMarker)
            break
        except TimeoutException:
            Counter += 1
            pass
        pass
    if Counter==XBeeModem.MAX_RETRY_COUNT: return False
    Data = Self._FragmentString_(Data, Self.MaxPayloadSize)
    for Datum in Data:
        Counter = 0
        while (XBeeModem.MAX_RETRY_COUNT-Counter)>0:
            try:
                Self.Modem.send_data(RemoteDevice, Datum)
                break
            except TimeoutException:
                Counter += 1
                pass
            pass
        if Counter==XBeeModem.MAX_RETRY_COUNT: return False
        pass
    Counter = 0
    while (XBeeModem.MAX_RETRY_COUNT-Counter)>0:
```

```
        try:
            Self.Modem.send_data(RemoteDevice, XBeeModem.EndMarker)
            break
        except TimeoutException:
            Counter += 1
            pass
        pass
    if Counter==XBeeModem.MAX_RETRY_COUNT: return False
    return True

# Data broadcast transmit function
def TxBroadcastData(Self, Data):
    RemoteDevice = RemoteXBeeDevice(Self.Modem, XBee64BitAddress.from_hex_string("0x000000000000FFFF"))
    Counter = 0
    while (XBeeModem.MAX_RETRY_COUNT-Counter)>0:
        try:
            Self.Modem.send_data_async(RemoteDevice, XBeeModem.StartMarker)
            break
        except XBeeException:
            Counter += 1
            pass
        pass
    if Counter==XBeeModem.MAX_RETRY_COUNT: return False
    Data = Self._FragmentString_(Data, Self.MaxPayloadSize)
    for Datum in Data:
        Counter = 0
        while (XBeeModem.MAX_RETRY_COUNT-Counter)>0:
            try:
                Self.Modem.send_data_async(RemoteDevice, Datum)
                break
            except XBeeException:
```

```
                Counter += 1
                pass
            pass
        if Counter==XBeeModem.MAX_RETRY_COUNT: return False
        pass
    Counter = 0
    while (XBeeModem.MAX_RETRY_COUNT-Counter)>0:
        try:
            Self.Modem.send_data_async(RemoteDevice, XBeeModem.EndMarker)
            break
        except XBeeException:
            Counter += 1
            pass
        pass
    if Counter==XBeeModem.MAX_RETRY_COUNT: return False
    return True

# Data received trigger function
def DataReceived(Self, DataPacket):
    if DataPacket.data.decode("UTF-8")==XBeeModem.StartMarker:
        Self.IncomingTransmissionPayload = ""
        pass
    elif DataPacket.data.decode("UTF-8")==XBeeModem.EndMarker:
        Self.Inbox.put((str(DataPacket.remote_device.get_64bit_addr()), Self.IncomingTransmissionPayload), False, None)
        pass
    else:
        Self.IncomingTransmissionPayload += DataPacket.data.decode("UTF-8")
        pass
    return

def Dispose(Self):
```

```
    Self.Modem.close()
    Self.Modem = None
    return

def _FragmentString_(Self, String, Length):
    StartIndex = 0
    EndIndex = Length
    Fragments = []
    Fragment = String[StartIndex:EndIndex]
    while not Fragment=="":
        Fragments.append(Fragment)
        StartIndex += Length
        EndIndex += Length
        Fragment = String[StartIndex:EndIndex]
        pass
    return Fragments
```

EPD.py

```
from Backend.Display.Waveshare.EPD4IN2.epd4in2 import epd4in2
from PIL import ImageFont
from PIL import ImageDraw
from PIL import Image
from time import sleep

class EPD:

    def __init__(Self):
        Self.EPDDevice = epd4in2()
        Self.EPDDevice.init()
        Self.Buffer = Image.new(
            mode="1",
            size=(Self.EPDDevice.width, Self.EPDDevice.height),
            color=1
        )
        Self.EPDDevice.display_frame_full(Self.EPDDevice.get_frame_buffer(Self.Buffer))
        Self.DisplayBaseState = Self.ClearDisplay
        Self.Font = ImageFont.truetype(
            font="Backend/Fonts/CourierNewBold.ttf",
            size=11
        )
        Self.CharacterWidth = 8
        Self.CharacterHeight = 11
        Self.CursorX = 0
        Self.CursorY = 0
        pass

    def PrintImage(Self, ImagePath):
```

```
    Self.Buffer = Image.open(fp=ImagePath, mode="r").convert(mode="1")
    Self.EPDDevice.display_frame_full(Self.EPDDevice.get_frame_buffer(Self.Buffer))
    return

def PrintText(Self, Text):
    Text = Text.replace("\t", "    ")
    Builder = ImageDraw.Draw(Self.Buffer)
    for Index, Character in enumerate(Text):
        if Self.CursorX>(Self.EPDDevice.width-Self.CharacterWidth):
            Self._ResetCursorX_()
            Self.CursorY += Self.CharacterHeight
            pass
        if Self.CursorX>(Self.EPDDevice.width-Self.CharacterWidth*6) and Self.CursorY>(Self.EPDDevice.height-Self.CharacterHeight*2) and (len(Text)-Index)>5:
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=".", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=" ", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=".", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=" ", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=".", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Self.EPDDevice.display_frame_full(Self.EPDDevice.get_frame_buffer(Self.Buffer))
            sleep(5.0)
            Self._ResetDisplayBaseState_()
            Builder = ImageDraw.Draw(Self.Buffer)
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=".", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=" ", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=".", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=" ", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            Builder.text(xy=(Self.CursorX, Self.CursorY), text=".", fill=0, font=Self.Font); Self.CursorX += Self.CharacterWidth
            pass
        elif Self.CursorY>(Self.EPDDevice.height-Self.CharacterHeight):
            Self.EPDDevice.display_frame_full(Self.EPDDevice.get_frame_buffer(Self.Buffer))
            sleep(5.0)
```

```
            Self._ResetDisplayBaseState_()
            Builder = ImageDraw.Draw(Self.Buffer)
            pass

        if Character=='\n':
            Self.CursorY += Self.CharacterHeight
            Self._ResetCursorX_()
            pass
        else:
            Builder.text(
                xy=(Self.CursorX, Self.CursorY),
                text=Character,
                fill=0,
                font=Self.Font
            )
            Self.CursorX += Self.CharacterWidth
            pass
        pass
    Self.EPDDevice.display_frame_full(Self.EPDDevice.get_frame_buffer(Self.Buffer))
    return

def ClearDisplay(Self):
    Self.DisplayBaseState = Self.ClearDisplay
    Self.Buffer = Image.new(
        mode="1",
        size=(Self.EPDDevice.width, Self.EPDDevice.height),
        color=1
    )
    Self.EPDDevice.display_frame_full(Self.EPDDevice.get_frame_buffer(Self.Buffer))
    Self._ResetCursor_()
    return
```

```
def PrintInformationFrame(Self):
    Self.DisplayBaseState = Self.PrintInformationFrame
    Self.PrintImage("Backend/Graphics/InformationFrame.bmp")
    Self._ResetCursor_()
    return

def PrintWarningFrame(Self):
    Self.DisplayBaseState = Self.PrintWarningFrame
    Self.PrintImage("Backend/Graphics/WarningFrame.bmp")
    Self._ResetCursor_()
    return

def PrintErrorFrame(Self):
    Self.DisplayBaseState = Self.PrintErrorFrame
    Self.PrintImage("Backend/Graphics/ErrorFrame.bmp")
    Self._ResetCursor_()
    return

def _ResetDisplayBaseState_(Self):
    Self.DisplayBaseState.__call__()
    return

def _ResetCursor_(Self):
    Self._ResetCursorX_()
    Self._ResetCursorY_()
    return

def _ResetCursorX_(Self):
    if Self.DisplayBaseState==Self.ClearDisplay:
        Self.CursorX = 0
```

```
        pass
    else:
        Self.CursorX = 4
        pass
    return

def _ResetCursorY_(Self):
    if Self.DisplayBaseState==Self.ClearDisplay:
        Self.CursorY = 0
        pass
    else:
        Self.CursorY = 41
        pass
    return
```

PolyBus.py

```
# Dependencies
from smbus2 import SMBus
from mraa import I2c as I2C
from smbus2 import i2c_msg as I2CMessage

# Parameters
PORT_BLOCK_ADDRESSES = {0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76} # Set of I2C addresses to reserve for the TCA9548A multiplexers

# Runtime placeholders
SMBusEngine = None # I2C System Management Bus object that's used to communicate with I2C enabled devices
MRAAEngine = None # I2C MRAA object that's used to communicate with I2C enabled devices
BusInitialized = False # Indicates whether the PolyBus device has been initialized
PortBlockOnline = {} # Hashmap to catalogue if multiplexers are online/offline
ChannelIDXlat = {} # Lookup table to translate ChannelID to PortBlockAddress
ActiveChannelID = 0 # One-indexed integer to indicate the active channel
ChannelMaps = None # 2D matrix to map I2C devices across all channels

# PolyBus engine initialization routine
# Arguments
#     Verbose (Boolean): Controls the console print statements
#         True : Print information and warning messages to the console
#         False: Suppress all console printers
#
# Return
#     N/A
#
# NOTE: This function raises exceptions if any fatal errors are encountered
def Initialize(Verbose=False):
    global BusInitialized
```

```
if not BusInitialized:
    if Verbose:
        print("  > Initializing the PolyBus engine. . .", end="")
        pass
    # Start initialization routine
    global SMBusEngine; SMBusEngine = SMBus(1) # Initialize the I2C System Management Bus object
    global MRAAEngine; MRAAEngine = I2C(1, True) # Initialize the I2C MRAA object
    global PortBlockOnline; global ChannelIDXlat; global ActiveChannelID # Declare the global placeholders in the local scope
    PortBlockAddresses = list(PORT_BLOCK_ADDRESSES); PortBlockAddresses.sort() # Fetch and sort the TCA9548A multiplexer addresses
    # Iterate through each port block address and. . .
    for Index, PortBlockAddress in enumerate(PortBlockAddresses):
        # . . .Setup its channel ID translater
        for Iterator in range((Index*8), (Index*8+8)):
            # ChannelIDXlat is 1-indexed but Iterator is 0-indexed
            ChannelIDXlat[Iterator+1] = PortBlockAddress
            pass

        try:
            # . . .Check if the port block is online and responsive. . .
            SMBusEngine.write_byte(PortBlockAddress, 0B00000000) # . . .Disengage the port block
            PortBlockOnline[PortBlockAddress] = True # . . .Mark port block as online
            if ActiveChannelID==0:
                PrimeChannelID = Index*8 + 1 # . . .Track the channel with the lowest numeric header
                SMBusEngine.write_byte(PortBlockAddress, 0B00000001) # . . .Activate this channel
                ActiveChannelID = PrimeChannelID # . . .Mark channel as active
                pass
            pass

        except OSError:
            PortBlockOnline[PortBlockAddress] = False # . . .Mark port block as offline
            pass
```

```
    pass

# Check if at least one PolyBus port block device is online and responsive
if not any(PortBlockOnline.values()):
    raise Exception("ERROR: The system could not detect a PolyBus device.\n-----\n")

BusInitialized = True # Mark the PolyBus device as being initialized

# Generate the instantaneous channel map
global ChannelMaps; ChannelMaps = [] # Initialize as an empty list
for ChannelIDIterator in range(max(list(ChannelIDXlat.keys()))):
    ChannelMaps.append(None) # Increase the list's length by one
    # Check if the channel's port block is online
    if PortBlockOnline[ChannelIDXlat[ChannelIDIterator+1]]:
        SwitchChannel(ChannelIDIterator+1) # Switch to this channel
        ChannelMaps[ChannelIDIterator] = [] # Initialize as a "nested" list
        # Iterate through the entire I2C address space
        for I2CAddressIterator in range(128):
            ChannelMaps[ChannelIDIterator].append(_PingI2CAddress_(I2CAddressIterator))
            # False: I2C device is either unresponsive or the address is vacant
            # True: I2C device is online and responsive at the address
            pass
        pass
    pass

SwitchChannel(PrimeChannelID) # Switch to the prime channel
# End initialization routine
if Verbose:
    print(" Success!\n")
    pass
pass
```

```
    else:
        if Verbose:
            print("WARNING: The PolyBus engine is already initialized\n-------\n")
            pass
        pass

    return

# PolyBus engine termination routine
# Arguments
#     Verbose (Boolean): Controls the console print statements
#         True : Print information and warning messages to the console
#         False: Suppress all console printers
#
# Return
#     N/A
def Terminate(Verbose=False):
    global BusInitialized
    if BusInitialized:
        if Verbose:
            print("  > Terminating the PolyBus engine. . .", end="")
            pass
        # Start termination routine
        # Declare the global placeholders in the local scope
        global SMBusEngine; global PortBlockOnline; global ChannelIDXlat
        global ActiveChannelID; global ChannelMaps

        SMBusEngine.close() # Close the I2C System Management Bus object

        # Set the global placeholders to their power-on-reset values
```

```
        SMBusEngine = None; BusInitialized = False; PortBlockOnline = {}
        ChannelIDXlat = {}; ActiveChannelID = 0; ChannelMaps = None
        # End termination routine
        if Verbose:
            print(" Success!")
            pass
        pass

    else:
        if Verbose:
            print("WARNING: The PolyBus engine is not initialized\n-------\n")
            pass
        pass

    return

# A function that allows switching between PolyBus channels
# Arguments
#     TargetChannelID (Integer): Numeric identifier of the channel to switch to
#
# Return
#     Boolean
#         True on success
#
# NOTE: This function raises exceptions if any fatal errors are encountered
def SwitchChannel(TargetChannelID):
    # Declare the global placeholders in the local scope
    global BusInitialized; global ActiveChannelID
    global ChannelIDXlat; global PortBlockOnline
    if not BusInitialized: raise Exception("ERROR: The PolyBus engine is not initialized\n-----\n") # Check if the PolyBus is initialized
```

```
    elif TargetChannelID not in list(ChannelIDXlat.keys()): raise Exception("ERROR: "+str(TargetChannelID)+" is not a valid channel identifier\n-----
\n") # Check if the TargetChannelID is a valid identifier
    PortBlockAddress = ChannelIDXlat[TargetChannelID] # Fetch the port block address of the TargetChannelID
    if not PortBlockOnline[PortBlockAddress]: raise Exception("ERROR: The Port Block at 0x"+format(PortBlockAddress, "02X")+" for Channel "+str(TargetChannelID)+" is offli
ne\n-----\n") # Check if the port block for the TargetChannelID is online
    elif TargetChannelID==ActiveChannelID: return True # Don't perform expensive I2C bus operations if the TargetChannelID is already active
    else:
        _DisableChannel_(ChannelID=ActiveChannelID, PerformanceMode=True) # Disable the PolyBus channel that's currently active
        _EnableChannel_(ChannelID=TargetChannelID, PerformanceMode=True) # Enable the requested PolyBus channel
        ActiveChannelID = TargetChannelID # Mark the requested PolyBus channel as active
        return True # Return success

# A function that returns a tabular map of the specified PolyBus Channel
# Arguments
#     ChannelID (Integer): Numeric identifier of the channel to print
#
# Return
#     If channel is online: String containing the table/map for the requested PolyBus channel
#     If channel is offline: String containing "N/A"
#
# NOTE: This function raises exceptions if any fatal errors are encountered
def GenerateChannelMap(ChannelID):
    global BusInitialized; global ChannelIDXlat; global PortBlockOnline; global ChannelMaps # Declare the global placeholders in the local scope
    if not BusInitialized: raise Exception("ERROR: The PolyBus engine is not initialized\n-----\n") # Check if the PolyBus is initialized
    elif ChannelID not in list(ChannelIDXlat.keys()): raise Exception("ERROR: "+str(ChannelID)+" is not a valid channel identifier\n-----
\n") # Check if the ChannelID is a valid identifier
    elif ChannelMaps[ChannelID-1] is None: return "N/A" # Channel map is not available
    else:
        # Generate the map
        Table = "     0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F" # Units a.k.a. column headers
        # Iterate through each I2C address and construct the map using the "Table" string builder
```

```
        for I2CAddressIterator in range(128):
            if (I2CAddressIterator%16)==0: Table += "\n0x"+format(I2CAddressIterator, "02X"); pass # Hexadecades a.k.a. row headers
            if ChannelMaps[ChannelID-1][I2CAddressIterator]: Table += " 0x"+format(I2CAddressIterator, "02X"); pass
            else: Table += " ----"; pass
            pass
        return Table # Return the string builder

# A function that returns a series of tabular maps, one for each channel of the PolyBus
# Arguments
#     N/A
#
# Return
#     String containing the sequence of tables/maps for each and every one of the PolyBus channels
def GenerateChannelMaps():
    Tables = "" # Declare and initialize the string builder
    global ChannelIDXlat # Declare the global placeholder in the local scope
    Channels = list(ChannelIDXlat.keys()); Channels.sort() # Fetch and sort the channel identifiers
    # Generate the series of tables
    for Channel in Channels:
        Tables += "Channel {0:2}".format(Channel)+"\n----------\n\n" # Channel header
        Tables += GenerateChannelMap(Channel) + "\n\n" # Channel map
        pass
    return Tables # Return the string builder

# Receives an 8-bit data packet from the device with I2C address DeviceAddress
# residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     Signed (Boolean): Identifies the data being read as un/signed
#         True: Interpret the 8-bit data packet as a signed integer
```

```
#         False: Interpret the 8-bit data packet as an unsigned integer
#
# Return
#     Integer
#         8-bit un/signed integer
def RxByte(ChannelID, DeviceAddress, Signed):
    if SwitchChannel(TargetChannelID=ChannelID):
        global SMBusEngine; Data = SMBusEngine.read_byte(i2c_addr=DeviceAddress) & 0xFF
        if Signed:
            if Data >= 128:
                Data -= 256
                pass
            pass
        pass
    return Data

# Transmits an 8-bit data packet with value Data to the device with I2C address
# DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     Data (Integer): Value of the data packet to transmit as an 8-bit integer
#
# Return
#     Boolean
#         True on success
def TxByte(ChannelID, DeviceAddress, Data):
    if SwitchChannel(TargetChannelID=ChannelID):
        Data = Data & 0xFF
        global SMBusEngine; SMBusEngine.write_byte(i2c_addr=DeviceAddress, value=Data)
        pass
```

```
    return True

# Receives a Length sized block of 8-bit data packets from the device with
# I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     Length (Integer): The number of 8-bit data packets to read
#     Signed (Boolean): Identifies the data being read as un/signed
#         True: Interpret each of the 8-bit data packets as a signed integer
#         False: Interpret each of the 8-bit data packets as an unsigned integer
#
# Return
#     Integer Array
#         8-bit un/signed integers
def RxBlock(ChannelID, DeviceAddress, Length, Signed):
    if SwitchChannel(TargetChannelID=ChannelID):
        global MRAAEngine; MRAAEngine.address(DeviceAddress)
        Data = list(MRAAEngine.read(Length))
        for Iterator in range(len(Data)):
            Data[Iterator] = Data[Iterator] & 0xFF
            if Signed:
                if Data[Iterator] >= 128:
                    Data[Iterator] -= 256
                    pass
                pass
            pass
        pass
    return Data

# Transmits a block of 8-bit data packets, each with a value from the ordered array Data, to the
```

```
# device with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     Data (Integer Array): Value of the data packets to sequentially transmit as 8-bit integers
#
# Return
#     Boolean
#         True on success
def TxBlock(ChannelID, DeviceAddress, Data):
    if SwitchChannel(TargetChannelID=ChannelID):
        global MRAAEngine; MRAAEngine.address(DeviceAddress)
        for Iterator in range(len(Data)): Data[Iterator] = Data[Iterator] & 0xFF; pass
        MRAAEngine.write(bytearray(Data))
        pass
    return True

# Transmits a block of 8-bit data packets, each with a value from the ordered array Data. Receives a Length sized block of 8-bit data packets.
# Both the Tx & Rx operations are performed with the device with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# in a single I2C transaction.
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     Data (Integer Array): Value of the data packets to sequentially transmit as 8-bit integers
#     Length (Integer): The number of 8-bit data packets to read
#     Signed (Boolean): Identifies the data being read as un/signed
#         True: Interpret each of the 8-bit data packets as a signed integer
#         False: Interpret each of the 8-bit data packets as an unsigned integer
#
# Return
#     Integer Array
```

```
#         8-bit un/signed integers
def TxRxBlock(ChannelID, DeviceAddress, Data, Length, Signed):
    if SwitchChannel(TargetChannelID=ChannelID):
        global MRAAEngine; MRAAEngine.address(DeviceAddress)
        for Iterator in range(len(Data)): Data[Iterator] = Data[Iterator] & 0xFF; pass
        MRAAEngine.write(bytearray(Data))
        Data = list(MRAAEngine.read(Length))
        for Iterator in range(len(Data)):
            Data[Iterator] = Data[Iterator] & 0xFF
            if Signed:
                if Data[Iterator] >= 128:
                    Data[Iterator] -= 256
                    pass
                pass
            pass
        pass
    return Data

# Receives an 8-bit data packet from the memory register at address RegisterAddress on the device
# with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     RegisterAddress (Integer): Numeric identifier of the memory register unique within the I2C device
#     Signed (Boolean): Identifies the data being read as un/signed
#         True: Interpret the 8-bit data packet as a signed integer
#         False: Interpret the 8-bit data packet as an unsigned integer
#
# Return
#     Integer
#         8-bit un/signed integer
```

```
def RxRegisterByte(ChannelID, DeviceAddress, RegisterAddress, Signed):
    if SwitchChannel(TargetChannelID=ChannelID):
        global SMBusEngine; Data = SMBusEngine.read_byte_data(i2c_addr=DeviceAddress, register=RegisterAddress) & 0xFF
        if Signed:
            if Data >= 128:
                Data -= 256
                pass
            pass
        pass
    return Data

# Transmits an 8-bit data packet with value Data to the memory register at address RegisterAddress on
# the device with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     RegisterAddress (Integer): Numeric identifier of the memory register unique within the I2C device
#     Data (Integer): Value of the data packet to transmit as an 8-bit integer
#
# Return
#     Boolean
#         True on success
def TxRegisterByte(ChannelID, DeviceAddress, RegisterAddress, Data):
    if SwitchChannel(TargetChannelID=ChannelID):
        Data = Data & 0xFF
        global SMBusEngine; SMBusEngine.write_byte_data(i2c_addr=DeviceAddress, register=RegisterAddress, value=Data)
        pass
    return True

# Receives a 16-bit data packet from the memory register at address RegisterAddress on the device
# with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
```

```
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     RegisterAddress (Integer): Numeric identifier of the memory register unique within the I2C device
#     Signed (Boolean): Identifies the data being read as un/signed
#         True: Interpret the 16-bit data packet as a signed integer
#         False: Interpret the 16-bit data packet as an unsigned integer
#
# Return
#     Integer
#         16-bit un/signed integer
def RxRegisterWord(ChannelID, DeviceAddress, RegisterAddress, Signed):
    if SwitchChannel(TargetChannelID=ChannelID):
        global SMBusEngine; Data = SMBusEngine.read_word_data(i2c_addr=DeviceAddress, register=RegisterAddress) & 0xFFFF
        if Signed:
            if Data >= 32768:
                Data -= 65536
                pass
            pass
        pass
    return Data

# Transmits a 16-bit data packet with value Data to the memory register at address RegisterAddress on
# the device with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     RegisterAddress (Integer): Numeric identifier of the memory register unique within the I2C device
#     Data (Integer): Value of the data packet to transmit as a 16-bit integer
#
# Return
```

```
#     Boolean
#         True on success
def TxRegisterWord(ChannelID, DeviceAddress, RegisterAddress, Data):
    if SwitchChannel(TargetChannelID=ChannelID):
        Data = Data & 0xFFFF
        global SMBusEngine; SMBusEngine.write_word_data(i2c_addr=DeviceAddress, register=RegisterAddress, value=Data)
        pass
    return True

# Receives a Length sized block of 8-bit data packets from an array of memory registers starting and auto incrementing from
# address RegisterAddress on the device with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     RegisterAddress (Integer): Numeric identifier of the base memory register unique within the I2C device
#                                Auto increments with each data packet received
#     Length (Integer): The number of 8-bit data packets to read
#     Signed (Boolean): Identifies the data being read as un/signed
#         True: Interpret each of the 8-bit data packets as a signed integer
#         False: Interpret each of the 8-bit data packets as an unsigned integer
#
# Return
#     Integer Array
#         8-bit un/signed integers
def RxRegisterBlock(ChannelID, DeviceAddress, RegisterAddress, Length, Signed):
    if SwitchChannel(TargetChannelID=ChannelID):
        global SMBusEngine; Data = SMBusEngine.read_i2c_block_data(i2c_addr=DeviceAddress, register=RegisterAddress, length=Length)
        for Iterator in range(len(Data)):
            Data[Iterator] = Data[Iterator] & 0xFF
            if Signed:
                if Data[Iterator] >= 128:
```

```
                    Data[Iterator] -= 256
                    pass
                pass
            pass
        pass
    return Data

# Transmits a block of 8-bit data packets, each with a value from the ordered array Data, to an array of memory registers starting and auto
# incrementing from address RegisterAddress on the device with I2C address DeviceAddress residing on the PolyBus channel with identifier ChannelID
# Arguments
#     ChannelID (Integer): Numeric identifier of the PolyBus channel the I2C device resides on
#     DeviceAddress (Integer): Numeric identifier of the I2C device unique within the PolyBus channel
#     RegisterAddress (Integer): Numeric identifier of the base memory register unique within the I2C device
#                                Auto increments with each data packet transmitted
#     Data (Integer Array): Value of the data packets to sequentially transmit as 8-bit integers
#
# Return
#     Boolean
#         True on success
def TxRegisterBlock(ChannelID, DeviceAddress, RegisterAddress, Data):
    if SwitchChannel(TargetChannelID=ChannelID):
        for Iterator in range(len(Data)): Data[Iterator] = Data[Iterator] & 0xFF; pass
        global SMBusEngine; SMBusEngine.write_i2c_block_data(i2c_addr=DeviceAddress, register=RegisterAddress, data=Data)
        pass
    return True

# A function to enable a channel on the PolyBus
# Arguments
#     ChannelID (Integer): Numeric identifier of the channel to enable
#     PerformanceMode (Boolean): Dis/engages high performance mode of the algorithm
#         True: Enables the requested port while disabling all other ports on the ChannelID's PortBlock. Up to 2x speed over PerformanceMode=False
```

```
#         False: Enables the requested port without affecting the state of the other ports on the ChannelID's PortBlock
#
# Return
#     Boolean
#         True on success
#         False on failure
#
# WARNING: This function doesn't perform any error checking
def _EnableChannel_(ChannelID, PerformanceMode=False):
    global ChannelIDXlat; PortBlockAddress = ChannelIDXlat[ChannelID] # Fetch the port block address of the ChannelID
    ChannelID -= 1; PortNumber = ChannelID % 8 # Calculate the port number of the ChannelID
    global SMBusEngine # Declare the global placeholder for the bus object in the local scope

    if PerformanceMode:
        ControlByte = (0B00000001) << PortNumber # Construct the new control byte
        SMBusEngine.write_byte(PortBlockAddress, ControlByte) # Transmit the control byte
        return True # Indicate success

    else:
        ControlByte = SMBusEngine.read_byte(PortBlockAddress) # Fetch the current control byte
        ControlByte = ControlByte | ((0B00000001)<<PortNumber) # Construct the new control byte
        SMBusEngine.write_byte(PortBlockAddress, ControlByte) # Transmit the control byte
        return True # Indicate success

    return False # Indicate failure

# A function to disable a channel on the PolyBus
# Arguments
#     ChannelID (Integer): Numeric identifier of the channel to disable
#     PerformanceMode (Boolean): Dis/engages high performance mode of the algorithm
#         True: Disables the entire PortBlock of the requested ChannelID. Up to 2x speed over PerformanceMode=False
```

```
#         False: Disables the requested port without affecting the state of the other ports on the ChannelID's PortBlock
#
# Return
#     Boolean
#         True on success
#         False on failure
#
# WARNING: This function doesn't perform any error checking
def _DisableChannel_(ChannelID, PerformanceMode=False):
    global ChannelIDXlat; PortBlockAddress = ChannelIDXlat[ChannelID] # Fetch the port block address of the ChannelID
    ChannelID -= 1; PortNumber = ChannelID % 8 # Calculate the port number of the ChannelID
    global SMBusEngine # Declare the global placeholder for the bus object in the local scope

    if PerformanceMode:
        SMBusEngine.write_byte(PortBlockAddress, 0B00000000) # Transmit the control byte to disengage the entire PortBlock
        return True # Indicate success

    else:
        ControlByte = SMBusEngine.read_byte(PortBlockAddress) # Fetch the current control byte
        ControlByte = ControlByte & (~((0B00000001)<<PortNumber)) # Construct the new control byte
        SMBusEngine.write_byte(PortBlockAddress, ControlByte) # Transmit the control byte
        return True # Indicate success

    return False # Indicate failure

# A function to ping for a device on the specified I2C address
# Arguments
#     I2CAddress (Integer): 7-bit I2C numeric identifier/address to ping
#
# Return
#     Boolean
```

```
#         False: If the device is offline or non-responsive
#         True: If the device is online and responsive
#
# WARNING: This function doesn't perform any error checking
def _PingI2CAddress_(I2CAddress):
    global SMBusEngine
    try: SMBusEngine.read_byte(I2CAddress); return True
    except OSError: return False
```

```
# Dependencies
import Backend.PolyBus as PolyBus

# Drivers
from Backend.Drivers.BNO055 import BNO055
from Backend.Drivers.BME280 import BME280
from Backend.Drivers.SPS30 import SPS30

def Address_0x00(ChannelID):
    return None

def Address_0x01(ChannelID):
    return None

def Address_0x02(ChannelID):
    return None

def Address_0x03(ChannelID):
    return None

def Address_0x04(ChannelID):
    return None

def Address_0x05(ChannelID):
    return None

def Address_0x06(ChannelID):
    return None
```

```
def Address_0x07(ChannelID):
    return None

def Address_0x08(ChannelID):
    return None

def Address_0x09(ChannelID):
    return None

def Address_0x0A(ChannelID):
    return None

def Address_0x0B(ChannelID):
    return None

def Address_0x0C(ChannelID):
    return None

def Address_0x0D(ChannelID):
    return None

def Address_0x0E(ChannelID):
    return None

def Address_0x0F(ChannelID):
    return None

def Address_0x10(ChannelID):
    return None

def Address_0x11(ChannelID):
```

```
    return None

def Address_0x12(ChannelID):
    return None

def Address_0x13(ChannelID):
    return None

def Address_0x14(ChannelID):
    return None

def Address_0x15(ChannelID):
    return None

def Address_0x16(ChannelID):
    return None

def Address_0x17(ChannelID):
    return None

def Address_0x18(ChannelID):
    return None

def Address_0x19(ChannelID):
    return None

def Address_0x1A(ChannelID):
    return None

def Address_0x1B(ChannelID):
    return None
```

```
def Address_0x1C(ChannelID):
    return None

def Address_0x1D(ChannelID):
    return None

def Address_0x1E(ChannelID):
    return None

def Address_0x1F(ChannelID):
    return None

def Address_0x20(ChannelID):
    return None

def Address_0x21(ChannelID):
    return None

def Address_0x22(ChannelID):
    return None

def Address_0x23(ChannelID):
    return None

def Address_0x24(ChannelID):
    return None

def Address_0x25(ChannelID):
    return None
```

```
def Address_0x26(ChannelID):
    return None

def Address_0x27(ChannelID):
    return None

def Address_0x28(ChannelID):
    Device = BNO055(ChannelID, 0x28)
    return Device

def Address_0x29(ChannelID):
    return None

def Address_0x2A(ChannelID):
    return None

def Address_0x2B(ChannelID):
    return None

def Address_0x2C(ChannelID):
    return None

def Address_0x2D(ChannelID):
    return None

def Address_0x2E(ChannelID):
    return None

def Address_0x2F(ChannelID):
    return None
```

```
def Address_0x30(ChannelID):
    return None

def Address_0x31(ChannelID):
    return None

def Address_0x32(ChannelID):
    return None

def Address_0x33(ChannelID):
    return None

def Address_0x34(ChannelID):
    return None

def Address_0x35(ChannelID):
    return None

def Address_0x36(ChannelID):
    return None

def Address_0x37(ChannelID):
    return None

def Address_0x38(ChannelID):
    return None

def Address_0x39(ChannelID):
    return None

def Address_0x3A(ChannelID):
```

```
    return None

def Address_0x3B(ChannelID):
    return None

def Address_0x3C(ChannelID):
    return None

def Address_0x3D(ChannelID):
    return None

def Address_0x3E(ChannelID):
    return None

def Address_0x3F(ChannelID):
    return None

def Address_0x40(ChannelID):
    return None

def Address_0x41(ChannelID):
    return None

def Address_0x42(ChannelID):
    return None

def Address_0x43(ChannelID):
    return None

def Address_0x44(ChannelID):
    return None
```

```
def Address_0x45(ChannelID):
    return None

def Address_0x46(ChannelID):
    return None

def Address_0x47(ChannelID):
    return None

def Address_0x48(ChannelID):
    return None

def Address_0x49(ChannelID):
    return None

def Address_0x4A(ChannelID):
    return None

def Address_0x4B(ChannelID):
    return None

def Address_0x4C(ChannelID):
    return None

def Address_0x4D(ChannelID):
    return None

def Address_0x4E(ChannelID):
    return None
```

```
def Address_0x4F(ChannelID):
    return None

def Address_0x50(ChannelID):
    return None

def Address_0x51(ChannelID):
    return None

def Address_0x52(ChannelID):
    return None

def Address_0x53(ChannelID):
    return None

def Address_0x54(ChannelID):
    return None

def Address_0x55(ChannelID):
    return None

def Address_0x56(ChannelID):
    return None

def Address_0x57(ChannelID):
    return None

def Address_0x58(ChannelID):
    return None

def Address_0x59(ChannelID):
```

```
    return None

def Address_0x5A(ChannelID):
    return None

def Address_0x5B(ChannelID):
    return None

def Address_0x5C(ChannelID):
    return None

def Address_0x5D(ChannelID):
    return None

def Address_0x5E(ChannelID):
    return None

def Address_0x5F(ChannelID):
    return None

def Address_0x60(ChannelID):
    return None

def Address_0x61(ChannelID):
    return None

def Address_0x62(ChannelID):
    return None

def Address_0x63(ChannelID):
    return None
```

```
def Address_0x64(ChannelID):
    return None

def Address_0x65(ChannelID):
    return None

def Address_0x66(ChannelID):
    return None

def Address_0x67(ChannelID):
    return None

def Address_0x68(ChannelID):
    return None

def Address_0x69(ChannelID):
    Device = SPS30(ChannelID, 0x69)
    return Device

def Address_0x6A(ChannelID):
    return None

def Address_0x6B(ChannelID):
    return None

def Address_0x6C(ChannelID):
    return None

def Address_0x6D(ChannelID):
    return None
```

```
def Address_0x6E(ChannelID):
    return None

def Address_0x6F(ChannelID):
    return None

def Address_0x70(ChannelID):
    return None

def Address_0x71(ChannelID):
    return None

def Address_0x72(ChannelID):
    return None

def Address_0x73(ChannelID):
    return None

def Address_0x74(ChannelID):
    return None

def Address_0x75(ChannelID):
    return None

def Address_0x76(ChannelID):
    return None

def Address_0x77(ChannelID):
    Device = BME280(ChannelID, 0x77)
    return Device
```

```
def Address_0x78(ChannelID):
    return None

def Address_0x79(ChannelID):
    return None

def Address_0x7A(ChannelID):
    return None

def Address_0x7B(ChannelID):
    return None

def Address_0x7C(ChannelID):
    return None

def Address_0x7D(ChannelID):
    return None

def Address_0x7E(ChannelID):
    return None

def Address_0x7F(ChannelID):
    return None

DeviceResolutionMap = {
    0 : Address_0x00,
    1 : Address_0x01,
    2 : Address_0x02,
    3 : Address_0x03,
    4 : Address_0x04,
```

```
5 : Address_0x05,
6 : Address_0x06,
7 : Address_0x07,
8 : Address_0x08,
9 : Address_0x09,
10 : Address_0x0A,
11 : Address_0x0B,
12 : Address_0x0C,
13 : Address_0x0D,
14 : Address_0x0E,
15 : Address_0x0F,
16 : Address_0x10,
17 : Address_0x11,
18 : Address_0x12,
19 : Address_0x13,
20 : Address_0x14,
21 : Address_0x15,
22 : Address_0x16,
23 : Address_0x17,
24 : Address_0x18,
25 : Address_0x19,
26 : Address_0x1A,
27 : Address_0x1B,
28 : Address_0x1C,
29 : Address_0x1D,
30 : Address_0x1E,
31 : Address_0x1F,
32 : Address_0x20,
33 : Address_0x21,
34 : Address_0x22,
35 : Address_0x23,
```

```
36 : Address_0x24,
37 : Address_0x25,
38 : Address_0x26,
39 : Address_0x27,
40 : Address_0x28,
41 : Address_0x29,
42 : Address_0x2A,
43 : Address_0x2B,
44 : Address_0x2C,
45 : Address_0x2D,
46 : Address_0x2E,
47 : Address_0x2F,
48 : Address_0x30,
49 : Address_0x31,
50 : Address_0x32,
51 : Address_0x33,
52 : Address_0x34,
53 : Address_0x35,
54 : Address_0x36,
55 : Address_0x37,
56 : Address_0x38,
57 : Address_0x39,
58 : Address_0x3A,
59 : Address_0x3B,
60 : Address_0x3C,
61 : Address_0x3D,
62 : Address_0x3E,
63 : Address_0x3F,
64 : Address_0x40,
65 : Address_0x41,
66 : Address_0x42,
```

```
67 : Address_0x43,
68 : Address_0x44,
69 : Address_0x45,
70 : Address_0x46,
71 : Address_0x47,
72 : Address_0x48,
73 : Address_0x49,
74 : Address_0x4A,
75 : Address_0x4B,
76 : Address_0x4C,
77 : Address_0x4D,
78 : Address_0x4E,
79 : Address_0x4F,
80 : Address_0x50,
81 : Address_0x51,
82 : Address_0x52,
83 : Address_0x53,
84 : Address_0x54,
85 : Address_0x55,
86 : Address_0x56,
87 : Address_0x57,
88 : Address_0x58,
89 : Address_0x59,
90 : Address_0x5A,
91 : Address_0x5B,
92 : Address_0x5C,
93 : Address_0x5D,
94 : Address_0x5E,
95 : Address_0x5F,
96 : Address_0x60,
97 : Address_0x61,
```

```
    98 : Address_0x62,
    99 : Address_0x63,
    100 : Address_0x64,
    101 : Address_0x65,
    102 : Address_0x66,
    103 : Address_0x67,
    104 : Address_0x68,
    105 : Address_0x69,
    106 : Address_0x6A,
    107 : Address_0x6B,
    108 : Address_0x6C,
    109 : Address_0x6D,
    110 : Address_0x6E,
    111 : Address_0x6F,
    112 : Address_0x70,
    113 : Address_0x71,
    114 : Address_0x72,
    115 : Address_0x73,
    116 : Address_0x74,
    117 : Address_0x75,
    118 : Address_0x76,
    119 : Address_0x77,
    120 : Address_0x78,
    121 : Address_0x79,
    122 : Address_0x7A,
    123 : Address_0x7B,
    124 : Address_0x7C,
    125 : Address_0x7D,
    126 : Address_0x7E,
    127 : Address_0x7F
}
```

```
Initialized = False
DeviceMaps = None
RankedDeviceList = None
Sensors = None
Readouts = None

def Initialize(Verbose=False):
    global Initialized
    if Initialized:
        if Verbose:
            print("WARNING: The Hardware Controller engine is already initialized\n-------\n")
            pass
        return

    if Verbose: print("> Initializing the Hardware Controller engine"); pass
    PolyBus.Initialize(Verbose=Verbose)
    global DeviceMaps; DeviceMaps = []
    global RankedDeviceList; RankedDeviceList = []
    global Sensors; Sensors = {}
    global Readouts; Readouts = {}
    for ChannelID, ChannelMap in enumerate(PolyBus.ChannelMaps):
        DeviceMaps.append(None)
        if ChannelMap is not None:
            DeviceMaps[ChannelID] = []
            for DeviceAddress, Online in enumerate(ChannelMap):
                DeviceMaps[ChannelID].append(None)
                if Online:
                    if DeviceAddress not in PolyBus.PORT_BLOCK_ADDRESSES:
                        DeviceMaps[ChannelID][DeviceAddress] = _ResolveDevice_(ChannelID=(ChannelID+1), DeviceAddress=DeviceAddress)
                        if DeviceMaps[ChannelID][DeviceAddress] is None: continue
```

```
                        RankedDeviceList.append(DeviceMaps[ChannelID][DeviceAddress])
                        for Measurement in DeviceMaps[ChannelID][DeviceAddress].MEASUREMENTS:
                            if Measurement not in Sensors: Sensors[Measurement] = []; pass
                            Sensors[Measurement].append(DeviceMaps[ChannelID][DeviceAddress])
                            Readouts[Measurement] = 0
                            Measurement = Measurement.replace('|', '_').replace('.', '_')
                            exec(compile("globals()['"+Measurement+"'] = 0", "<string>", "exec"))
                            pass
                        pass
                    pass
                pass
            pass
        pass

    if Verbose: print("> Hardware Controller engine successfully initialized"); pass
    Initialized = True
    return

def Terminate(Verbose=False):
    global Initialized
    if not Initialized:
        if Verbose:
            print("WARNING: The Hardware Controller engine is not initialized\n-------\n")
            pass
        return

    global DeviceMaps; DeviceMaps = None
    global RankedDeviceList; RankedDeviceList = None
    global Sensors; Sensors = None
    global Readouts; Readouts = None
    PolyBus.Terminate(Verbose=Verbose)
```

```
    if Verbose: print("> Terminating the Hardware Controller engine. . .", end=""); pass
    if Verbose: print(" Success!"); pass
    Initialized = False
    return

def UpdateReadoutCache():
    global Initialized
    if not Initialized: raise Exception("ERROR: The Hardware Controller engine is not initialized\n-----\n")

    global RankedDeviceList
    global Readouts
    for Device in RankedDeviceList:
        if Device.UpdateInternalCache():
            for Measurement in Device.MEASUREMENTS:
                # Readouts[Measurement] = Device.GetMeasurementCache(*Measurement.split('|'))
                exec(compile("globals()['"+Measurement.replace('|', '_').replace('.', '_')+"'] = "+str(Device.GetMeasurementCache(*Measurement.split('|'))), "<string>", "e
xec"))
                pass
            pass
        pass
    return

def _ResolveDevice_(ChannelID, DeviceAddress):
    return DeviceResolutionMap[DeviceAddress].__call__(ChannelID)
```

```
from copy import deepcopy as DeepCopy

class ComplementaryFilter:

    def __init__(Self, Initializer, ComplementaryRatio):
        if ComplementaryRatio>1.0: raise ValueError("ERROR: Complementary Filter Ratio cannot have a value greater than 1.0\n-----
\n")
        if type(Initializer)==tuple:
            Self.FilterFunction = Self._FilterTuple_
            Initializer = list(Initializer)
            pass

        else:
            Self.FilterFunction = Self._FilterValue_
            pass

        Self.PreviousEstimate = DeepCopy(Initializer)
        Self.CurrentEstimate = DeepCopy(Initializer)
        Self.ComplementaryRatio = DeepCopy(ComplementaryRatio)
        pass

    def Filter(Self, Measurement):
        return Self.FilterFunction.__call__(Measurement)

    def _FilterValue_(Self, Measurement):
        Self.CurrentEstimate = Self.PreviousEstimate*Self.ComplementaryRatio + Measurement*(1-Self.ComplementaryRatio)
        Self.PreviousEstimate = Self.CurrentEstimate
        return Self.CurrentEstimate
```

```
    def _FilterTuple_(Self, Measurement):
        Measurement = list(Measurement)
        for Iterator in range(len(Measurement)):
            Self.CurrentEstimate[Iterator] = Self.PreviousEstimate[Iterator]*Self.ComplementaryRatio + Measurement[Iterator]*(1-
Self.ComplementaryRatio)
            Self.PreviousEstimate[Iterator] = Self.CurrentEstimate[Iterator]
            pass
        return tuple(Self.CurrentEstimate)

class KalmanFilter:

    def __init__(Self, Initializer, MeasurementError):
        if type(Initializer)==tuple:
            Self.FilterFunction = Self._FilterTuple_
            Initializer = list(Initializer)
            Self.PreviousEstimateError = [DeepCopy(MeasurementError)] * len(Initializer)
            Self.CurrentEstimateError = [DeepCopy(MeasurementError)*1000] * len(Initializer)
            pass

        else:
            Self.FilterFunction = Self._FilterValue_
            Self.PreviousEstimateError = DeepCopy(MeasurementError)
            Self.CurrentEstimateError = DeepCopy(MeasurementError)*1000
            pass

        Self.MeasurementError = DeepCopy(MeasurementError)
        Self.PreviousEstimate = DeepCopy(Initializer)
        Self.CurrentEstimate = DeepCopy(Initializer)
        Self.KalmanGain = DeepCopy(Initializer)
        pass
```

```
    def Filter(Self, Measurement):
        return Self.FilterFunction.__call__(Measurement)

    def _FilterValue_(Self, Measurement):
        Self.KalmanGain = Self.CurrentEstimateError / (Self.CurrentEstimateError+Self.MeasurementError)
        Self.CurrentEstimate = (1-Self.KalmanGain)*Self.PreviousEstimate + Self.KalmanGain*Measurement
        Self.CurrentEstimateError = (1-Self.KalmanGain) * Self.PreviousEstimateError
        Self.PreviousEstimate = Self.CurrentEstimate
        Self.PreviousEstimateError = Self.CurrentEstimateError
        return Self.CurrentEstimate

    def _FilterTuple_(Self, Measurement):
        Measurement = list(Measurement)
        for Iterator in range(len(Measurement)):
            Self.KalmanGain[Iterator] = Self.CurrentEstimateError[Iterator] / (Self.CurrentEstimateError[Iterator]+Self.Measuremen
tError)
            Self.CurrentEstimate[Iterator] = (1-
Self.KalmanGain[Iterator])*Self.PreviousEstimate[Iterator] + Self.KalmanGain[Iterator]*Measurement[Iterator]
            Self.CurrentEstimateError[Iterator] = (1-Self.KalmanGain[Iterator]) * Self.PreviousEstimateError[Iterator]
            Self.PreviousEstimate[Iterator] = Self.CurrentEstimate[Iterator]
            Self.PreviousEstimateError[Iterator] = Self.CurrentEstimateError[Iterator]
            pass
        return tuple(Self.CurrentEstimate)

class ComplementaryKalmanFilter:

    def __init__(Self, Initializer, MeasurementError, ComplementaryRatio):
        if ComplementaryRatio>1.0: raise ValueError("ERROR: Complementary Kalman Filter Ratio cannot have a value greater than 1.0
\n-----\n")
        if type(Initializer)==tuple:
            Self.FilterFunction = Self._FilterTuple_
```

```
            Initializer = list(Initializer)
            Self.PreviousEstimateError = [DeepCopy(MeasurementError)] * len(Initializer)
            Self.CurrentEstimateError = [DeepCopy(MeasurementError)*1000] * len(Initializer)
            pass

        else:
            Self.FilterFunction = Self._FilterValue_
            Self.PreviousEstimateError = DeepCopy(MeasurementError)
            Self.CurrentEstimateError = DeepCopy(MeasurementError)*1000
            pass

        Self.MeasurementError = DeepCopy(MeasurementError)
        Self.PreviousEstimate = DeepCopy(Initializer)
        Self.CurrentEstimate = DeepCopy(Initializer)
        Self.KalmanGain = DeepCopy(Initializer)
        Self.ComplementaryRatio = DeepCopy(ComplementaryRatio)
        pass

    def Filter(Self, Measurement):
        return Self.FilterFunction.__call__(Measurement)

    def _FilterValue_(Self, Measurement):
        Self.KalmanGain = Self.CurrentEstimateError / (Self.CurrentEstimateError+Self.MeasurementError)
        Self.CurrentEstimate = (1-Self.KalmanGain)*Self.PreviousEstimate + Self.KalmanGain*Measurement
        Self.CurrentEstimateError = (1-Self.KalmanGain) * Self.PreviousEstimateError
        Self.PreviousEstimate = Self.CurrentEstimate*Self.ComplementaryRatio + Measurement*(1-Self.ComplementaryRatio)
        Self.PreviousEstimateError = Self.CurrentEstimateError*Self.ComplementaryRatio + Self.MeasurementError*(1-
Self.ComplementaryRatio)
        return Self.CurrentEstimate

    def _FilterTuple_(Self, Measurement):
```

```
        Measurement = list(Measurement)
        for Iterator in range(len(Measurement)):
            Self.KalmanGain[Iterator] = Self.CurrentEstimateError[Iterator] / (Self.CurrentEstimateError[Iterator]+Self.Measuremen
tError)
            Self.CurrentEstimate[Iterator] = (1-
Self.KalmanGain[Iterator])*Self.PreviousEstimate[Iterator] + Self.KalmanGain[Iterator]*Measurement[Iterator]
            Self.CurrentEstimateError[Iterator] = (1-Self.KalmanGain[Iterator]) * Self.PreviousEstimateError[Iterator]
            Self.PreviousEstimate[Iterator] = Self.CurrentEstimate[Iterator]*Self.ComplementaryRatio + Measurement[Iterator]*(1-
Self.ComplementaryRatio)
            Self.PreviousEstimateError[Iterator] = Self.CurrentEstimateError[Iterator]*Self.ComplementaryRatio + Self.MeasurementE
rror*(1-Self.ComplementaryRatio)
            pass
        return tuple(Self.CurrentEstimate)
```

```
# Low-level dependencies
import struct

def CompressUnsignedInteger32(Input):
    return struct.pack(">L", Input).hex().upper()

def DecompressUnsignedInteger32(Input):
    return struct.unpack(">L", bytes.fromhex(Input))[0]

def CompressUnsignedInteger64(Input):
    return struct.pack(">Q", Input).hex().upper()

def DecompressUnsignedInteger64(Input):
    return struct.unpack(">Q", bytes.fromhex(Input))[0]

def CompressInteger32(Input):
    return struct.pack(">l", Input).hex().upper()

def DecompressInteger32(Input):
    return struct.unpack(">l", bytes.fromhex(Input))[0]

def CompressInteger64(Input):
    return struct.pack(">q", Input).hex().upper()

def DecompressInteger64(Input):
    return struct.unpack(">q", bytes.fromhex(Input))[0]

def CompressFloat32(Input):
    return struct.pack(">f", Input).hex().upper()
```

```
def DecompressFloat32(Input):
    return struct.unpack(">f", bytes.fromhex(Input))[0]

def CompressFloat64(Input):
    return struct.pack(">d", Input).hex().upper()

def DecompressFloat64(Input):
    return struct.unpack(">d", bytes.fromhex(Input))[0]
```

```
def GenerateCommandMap():
    CommandMap = {
        "00000000" : "",
        "00000001" : "",
        "00000010" : "",
        "00000011" : "",
        "00000100" : "",
        "00000101" : "",
        "00000110" : "",
        "00000111" : "",
        "00001000" : "",
        "00001001" : "",
        "00001010" : "",
        "00001011" : "",
        "00001100" : "",
        "00001101" : "",
        "00001110" : "",
        "00001111" : "",
        "00010000" : "",
        "00010001" : "",
        "00010010" : "",
        "00010011" : "",
        "00010100" : "",
        "00010101" : "",
        "00010110" : "",
        "00010111" : "",
        "00011000" : "",
        "00011001" : "",
        "00011010" : "",
```

```
"00011011" : "",
"00011100" : "",
"00011101" : "",
"00011110" : "",
"00011111" : "",
"00100000" : "",
"00100001" : "",
"00100010" : "",
"00100011" : "",
"00100100" : "",
"00100101" : "",
"00100110" : "",
"00100111" : "",
"00101000" : "",
"00101001" : "",
"00101010" : "",
"00101011" : "",
"00101100" : "",
"00101101" : "",
"00101110" : "",
"00101111" : "",
"00110000" : "",
"00110001" : "",
"00110010" : "",
"00110011" : "",
"00110100" : "",
"00110101" : "",
"00110110" : "",
"00110111" : "",
"00111000" : "",
"00111001" : "",
```

```
"00111010" : "",
"00111011" : "",
"00111100" : "",
"00111101" : "",
"00111110" : "",
"00111111" : "",
"01000000" : "",
"01000001" : "",
"01000010" : "",
"01000011" : "",
"01000100" : "",
"01000101" : "",
"01000110" : "",
"01000111" : "",
"01001000" : "",
"01001001" : "",
"01001010" : "",
"01001011" : "",
"01001100" : "",
"01001101" : "",
"01001110" : "",
"01001111" : "",
"01010000" : "",
"01010001" : "",
"01010010" : "",
"01010011" : "",
"01010100" : "",
"01010101" : "",
"01010110" : "",
"01010111" : "",
"01011000" : "",
```

```
"01011001" : "",
"01011010" : "",
"01011011" : "",
"01011100" : "",
"01011101" : "",
"01011110" : "",
"01011111" : "",
"01100000" : "",
"01100001" : "",
"01100010" : "",
"01100011" : "",
"01100100" : "",
"01100101" : "",
"01100110" : "",
"01100111" : "",
"01101000" : "",
"01101001" : "",
"01101010" : "",
"01101011" : "",
"01101100" : "",
"01101101" : "",
"01101110" : "",
"01101111" : "",
"01110000" : "",
"01110001" : "",
"01110010" : "",
"01110011" : "",
"01110100" : "",
"01110101" : "",
"01110110" : "",
"01110111" : "",
```

```
"01111000" : "",
"01111001" : "",
"01111010" : "",
"01111011" : "",
"01111100" : "",
"01111101" : "",
"01111110" : "",
"01111111" : "",
"10000000" : "Shutdown",
"10000001" : "Reboot",
"10000010" : "Ping_Hardware",
"10000011" : "Ping_Software",
"10000100" : "Acquire_Data",
"10000101" : "Abort_Task",
"10000110" : "Halt_Task",
"10000111" : "Resume_Task",
"10001000" : "",
"10001001" : "",
"10001010" : "",
"10001011" : "",
"10001100" : "",
"10001101" : "",
"10001110" : "",
"10001111" : "",
"10010000" : "",
"10010001" : "",
"10010010" : "",
"10010011" : "",
"10010100" : "",
"10010101" : "",
"10010110" : "",
```

```
"10010111" : "",
"10011000" : "",
"10011001" : "",
"10011010" : "",
"10011011" : "",
"10011100" : "",
"10011101" : "",
"10011110" : "",
"10011111" : "",
"10100000" : "",
"10100001" : "",
"10100010" : "",
"10100011" : "",
"10100100" : "",
"10100101" : "",
"10100110" : "",
"10100111" : "",
"10101000" : "",
"10101001" : "",
"10101010" : "",
"10101011" : "",
"10101100" : "",
"10101101" : "",
"10101110" : "",
"10101111" : "",
"10110000" : "",
"10110001" : "",
"10110010" : "",
"10110011" : "",
"10110100" : "",
"10110101" : "",
```

```
"10110110" : "",
"10110111" : "",
"10111000" : "",
"10111001" : "",
"10111010" : "",
"10111011" : "",
"10111100" : "",
"10111101" : "",
"10111110" : "",
"10111111" : "",
"11000000" : "",
"11000001" : "",
"11000010" : "",
"11000011" : "",
"11000100" : "",
"11000101" : "",
"11000110" : "",
"11000111" : "",
"11001000" : "",
"11001001" : "",
"11001010" : "",
"11001011" : "",
"11001100" : "",
"11001101" : "",
"11001110" : "",
"11001111" : "",
"11010000" : "",
"11010001" : "",
"11010010" : "",
"11010011" : "",
"11010100" : "",
```

```
"11010101" : "",
"11010110" : "",
"11010111" : "",
"11011000" : "",
"11011001" : "",
"11011010" : "",
"11011011" : "",
"11011100" : "",
"11011101" : "",
"11011110" : "",
"11011111" : "",
"11100000" : "",
"11100001" : "",
"11100010" : "",
"11100011" : "",
"11100100" : "",
"11100101" : "",
"11100110" : "",
"11100111" : "",
"11101000" : "",
"11101001" : "",
"11101010" : "",
"11101011" : "",
"11101100" : "",
"11101101" : "",
"11101110" : "",
"11101111" : "",
"11110000" : "",
"11110001" : "",
"11110010" : "",
"11110011" : "",
```

```
    "11110100" : "",
    "11110101" : "",
    "11110110" : "",
    "11110111" : "",
    "11111000" : "",
    "11111001" : "",
    "11111010" : "",
    "11111011" : "",
    "11111100" : "",
    "11111101" : "",
    "11111110" : "",
    "11111111" : ""
}

return CommandMap
```

```
# Low-level dependencies
from time import time_ns as EpochNS
from time import sleep as Sleep

# High-level dependencies
from Support.NumericDe_Compression import *
import Global

class Halt_Task:

    def __init__(Self, Address, ArgumentVector, PID):
        Self.ArgumentVector = ArgumentVector
        Self.Address = Address
        Self.PID = PID

        Self.PreviousExecutionTimestamp = EpochNS()
        Self.NextExecutionTimestamp = 0
        Self.ExecutionCounter = 1
        Self.Error = False
        Self.ErrorMessage = ""
        Global.Modem.TxData((Self.Address, "NAK"))
        Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
        Data = Global.Modem.RxData()
        while Data is None:
            Sleep(0.05)
            Data = Global.Modem.RxData()
            pass
        Self.PIDArgument = DecompressUnsignedInteger32(Data[1])
        pass
```

```
def GetPriorityObject(Self):
    if Self.ExecutionCounter==0: return None
    else: return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter==0: return None
    else:
        if not Self.Error:
            if Self.PIDArgument>Global.CurrentPID:
                Self.Error = True
                Self.ErrorMessage = "A task with PID=" + str(Self.PIDArgument) + " does not exist in the current RSA session"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDAbort:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has been aborted and cannot be halted"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDComplete:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has already completed its operation"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDHalt:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has already been halted"
                return (Self.Address, "NAK")
            else:
                Global.PIDHalt.add(Self.PIDArgument)
                Self.ExecutionCounter -= 1
                return (Self.Address, "ACK")

        else:
```

```
Self.ExecutionCounter -= 1
return (Self.Address, Self.ErrorMessage)
```

```
# Low-level dependencies
from time import time_ns as EpochNS
from time import sleep as Sleep

# High-level dependencies
from Support.NumericDe_Compression import *
import Global

class Resume_Task:

    def __init__(Self, Address, ArgumentVector, PID):
        Self.ArgumentVector = ArgumentVector
        Self.Address = Address
        Self.PID = PID

        Self.PreviousExecutionTimestamp = EpochNS()
        Self.NextExecutionTimestamp = 0
        Self.ExecutionCounter = 1
        Self.Error = False
        Self.ErrorMessage = ""
        Global.Modem.TxData((Self.Address, "NAK"))
        Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
        Data = Global.Modem.RxData()
        while Data is None:
            Sleep(0.05)
            Data = Global.Modem.RxData()
            pass
        Self.PIDArgument = DecompressUnsignedInteger32(Data[1])
        pass
```

```
def GetPriorityObject(Self):
    if Self.ExecutionCounter==0: return None
    else: return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter==0: return None
    else:
        if not Self.Error:
            if Self.PIDArgument>Global.CurrentPID:
                Self.Error = True
                Self.ErrorMessage = "A task with PID=" + str(Self.PIDArgument) + " does not exist in the current RSA session"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDAbort:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has been aborted and cannot be resumed"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDComplete:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has already completed its operation"
                return (Self.Address, "NAK")
            elif Self.PIDArgument not in Global.PIDHalt:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " is already running"
                return (Self.Address, "NAK")
            else:
                for Task in Global.TaskHalt:
                    if Task.PID==Self.PIDArgument:
                        Global.TaskHalt.remove(Task)
                        Global.PIDHalt.remove(Self.PIDArgument)
                        PriorityObject = Task.GetPriorityObject()
```

```
                if PriorityObject is not None:
                    Global.WaitQueue.put(PriorityObject, False, None)
                    pass
                break
            pass
        Self.ExecutionCounter -= 1
        return (Self.Address, "ACK")

else:
    Self.ExecutionCounter -= 1
    return (Self.Address, Self.ErrorMessage)
```

```
# Low-level dependencies
from time import time_ns as EpochNS
from time import sleep as Sleep

# High-level dependencies
from Support.NumericDe_Compression import *
import Global

class Abort_Task:

    def __init__(Self, Address, ArgumentVector, PID):
        Self.ArgumentVector = ArgumentVector
        Self.Address = Address
        Self.PID = PID

        Self.PreviousExecutionTimestamp = EpochNS()
        Self.NextExecutionTimestamp = 0
        Self.ExecutionCounter = 1
        Self.Error = False
        Self.ErrorMessage = ""
        Global.Modem.TxData((Self.Address, "NAK"))
        Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
        Data = Global.Modem.RxData()
        while Data is None:
            Sleep(0.05)
            Data = Global.Modem.RxData()
            pass
        Self.PIDArgument = DecompressUnsignedInteger32(Data[1])
        pass
```

```
def GetPriorityObject(Self):
    if Self.ExecutionCounter==0: return None
    else: return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter==0: return None
    else:
        if not Self.Error:
            if Self.PIDArgument>Global.CurrentPID:
                Self.Error = True
                Self.ErrorMessage = "A task with PID=" + str(Self.PIDArgument) + " does not exist in the current RSA session"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDAbort:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has already been aborted"
                return (Self.Address, "NAK")
            elif Self.PIDArgument in Global.PIDComplete:
                Self.Error = True
                Self.ErrorMessage = "The task with PID=" + str(Self.PIDArgument) + " has already completed its operation"
                return (Self.Address, "NAK")
            else:
                Global.PIDAbort.add(Self.PIDArgument)
                for Task in Global.TaskHalt:
                    if Task.PID==Self.PIDArgument:
                        Global.TaskHalt.remove(Task)
                        Global.PIDHalt.remove(Self.PIDArgument)
                        break
                    pass
                Self.ExecutionCounter -= 1
                return (Self.Address, "ACK")
```

```
else:
    Self.ExecutionCounter -= 1
    return (Self.Address, Self.ErrorMessage)
```

```
# Low-level dependencies
from copy import deepcopy as DeepCopy
from time import time_ns as EpochNS
from time import sleep as Sleep

# High-level dependencies
import Backend.HardwareController as HardwareController
from Support.NumericDe_Compression import *
import Global

class Acquire_Data:

    DataComponentDecoder = {
        "Geographic|Latitude" :  0B00000000000000000000000001,
        "Geographic|Longitude" : 0B00000000000000000000000010,
        "Cartesian|X" :          0B00000000000000000000000100,
        "Cartesian|Y" :          0B00000000000000000000001000,
        "ACCELERATION|RAW" :     0B00000000000000000000010000,
        "ACCELERATION|LINEAR" :  0B00000000000000000000100000,
        "ACCELERATION|GRAVITY" : 0B00000000000000000001000000,
        "ANGULARVELOCITY|RAW" :  0B00000000000000000010000000,
        "MAGNETICFIELD|RAW" :    0B00000000000000000100000000,
        "ATTITUDE|EULER" :       0B00000000000000001000000000,
        "ATTITUDE|QUATERNION" :  0B00000000000000010000000000,
        "ECO2" :                 0B00000000000000100000000000,
        "TVOC" :                 0B00000000000001000000000000,
        "HUMIDITY|RELATIVE" :    0B00000000000010000000000000,
        "PRESSURE" :             0B00000000000100000000000000,
        "TEMPERATURE" :          0B00000000001000000000000000,
```

```
    "PM|TPS" :               0B00000000001000000000000000,
    "PM0.5|NC" :             0B00000000010000000000000000,
    "PM0.5|MC" :             0B00000000100000000000000000,
    "PM1.0|NC" :             0B00000001000000000000000000,
    "PM1.0|MC" :             0B00000010000000000000000000,
    "PM2.5|NC" :             0B00000100000000000000000000,
    "PM2.5|MC" :             0B00001000000000000000000000,
    "PM4.0|NC" :             0B00010000000000000000000000,
    "PM4.0|MC" :             0B00100000000000000000000000,
    "PM10.0|NC" :            0B01000000000000000000000000,
    "PM10.0|MC" :            0B10000000000000000000000000
}

DataComponentEncoder = {
    "GEOGRAPHIC|LATITUDE" :  "A",
    "GEOGRAPHIC|LONGITUDE" : "B",
    "CARTESIAN|X" :          "C",
    "CARTESIAN|Y" :          "D",
    "ACCELERATION|RAW" :     "E",
    "ACCELERATION|LINEAR" :  "F",
    "ACCELERATION|GRAVITY" : "G",
    "ANGULARVELOCITY|RAW" :  "H",
    "MAGNETICFIELD|RAW" :    "I",
    "ATTITUDE|EULER" :       "J",
    "ATTITUDE|QUATERNION" :  "K",
    "ECO2" :                 "L",
    "TVOC" :                 "M",
    "HUMIDITY|RELATIVE" :    "N",
    "PRESSURE" :             "O",
    "TEMPERATURE" :          "P",
    "PM|TPS" :               "Q",
```

```
    "PM0.5|NC" :             "R",
    "PM0.5|MC" :             "S",
    "PM1.0|NC" :             "T",
    "PM1.0|MC" :             "U",
    "PM2.5|NC" :             "V",
    "PM2.5|MC" :             "W",
    "PM4.0|NC" :             "X",
    "PM4.0|MC" :             "Y",
    "PM10.0|NC" :            "Z",
    "PM10.0|MC" :            "AA"
}

def __init__(Self, Address, ArgumentVector, PID):
    Self.ArgumentVector = ArgumentVector
    Self.Address = Address
    Self.PID = PID

    Self.TxBuffer = []
    Self.TensorIDCounter = 0
    Global.Modem.TxData((Self.Address, "ACK"))
    Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
    Data = Global.Modem.RxData()
    while Data is None:
        Sleep(0.05)
        Data = Global.Modem.RxData()
        pass
    Data = DecompressUnsignedInteger64(Data[1])
    Self.ReadoutKeys = []
    for ReadoutKey in HardwareController.Readouts.keys():
        if bool(Acquire_Data.DataComponentDecoder[ReadoutKey] & Data):
            Self.ReadoutKeys.append(ReadoutKey)
```

```
        pass
    pass
if Self.ArgumentVector[1]=='1':
    Data = Global.Modem.RxData()
    while Data is None:
        Sleep(0.05)
        Data = Global.Modem.RxData()
        pass
    Self.FrequencyArgument = DecompressFloat32(Data[1])
    pass
else:
    Self.FrequencyArgument = 0.0
    pass
if Self.ArgumentVector[2]=='1':
    Data = Global.Modem.RxData()
    while Data is None:
        Sleep(0.05)
        Data = Global.Modem.RxData()
        pass
    Self.TimeArgument = DecompressFloat32(Data[1])
    pass
else:
    Self.TimeArgument = 0.0
    pass
Self.PreviousExecutionTimestamp = EpochNS()
if (not Self.FrequencyArgument==0.0) and (not Self.TimeArgument==0.0):
    Self.ExecutionCounter = int(Self.FrequencyArgument*Self.TimeArgument)
    Self.TimePeriod = int(1000000000/Self.FrequencyArgument)
    Self.NextExecutionTimestamp = Self.PreviousExecutionTimestamp - Self.TimePeriod
    Self.PreviousTxTimestamp = Self.PreviousExecutionTimestamp - Self.TimePeriod
    pass
```

```
    else:
        Self.ExecutionCounter = 1
        Self.TimePeriod = 0
        Self.NextExecutionTimestamp = Self.PreviousExecutionTimestamp - Self.TimePeriod
        Self.PreviousTxTimestamp = Self.PreviousExecutionTimestamp - Self.TimePeriod
        pass
    pass

def GetPriorityObject(Self):
    if Self.ExecutionCounter<0: return None
    else: return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter<0: return None
    elif Self.ExecutionCounter==0:
        Self.ExecutionCounter -= 1
        return (Self.Address, "ACK")
    else:
        Self.TxBuffer.append({})
        Self.TxBuffer[len(Self.TxBuffer)-1]["@"] = [CompressUnsignedInteger64(EpochNS())]
        Self.TxBuffer[len(Self.TxBuffer)-1]["#"] = [CompressUnsignedInteger32(Self.TensorIDCounter)]
        for ReadoutKey in Self.ReadoutKeys:
            # Readout = DeepCopy(HardwareController.Readouts[ReadoutKey])
            Readout = eval(compile("HardwareController."+ReadoutKey.replace('|', '_').replace('.', '_'), "<string>", "eval"))
            if type(Readout)==tuple:
                Readout = list(Readout)
                for Iterator in range(len(Readout)):
                    Readout[Iterator] = str(CompressFloat64(Readout[Iterator]))
                    pass
                pass
            else:
```

```
        Readout = str(CompressFloat64(Readout))
        Readout = [Readout]
        pass
    Self.TxBuffer[len(Self.TxBuffer)-1][Acquire_Data.DataComponentEncoder[ReadoutKey]] = Readout
    pass
Self.TensorIDCounter += 1
Self.ExecutionCounter -= 1
Self.PreviousExecutionTimestamp = EpochNS()
Self.NextExecutionTimestamp = Self.PreviousExecutionTimestamp + Self.TimePeriod
Payload = str(Self.TxBuffer).replace(", ", ",").replace(": ", ":").replace("'", '"')
if Self.ExecutionCounter==0 or (EpochNS()-Self.PreviousTxTimestamp)>=1000000000 or len(Payload)>=4096:
    Self.TxBuffer = []
    Self.PreviousTxTimestamp = EpochNS()
    return (Self.Address, Payload)
else: return None
```

```
# Low-level dependencies
from time import time_ns as EpochNS
from time import time as EpochS

# High-level dependencies
import Backend.HardwareController as HardwareController
from Support.NumericDe_Compression import *
import Global

class Ping_Hardware:

    HardwareFilterDecoder = {
        "000000" : "RSA",
        "000001" : "SENSORS",
        "000010" : "ACCELERATION",
        "000011" : "ACCELERATION|RAW",
        "000100" : "ACCELERATION|LINEAR",
        "000101" : "ACCELERATION|GRAVITY",
        "000110" : "ANGULARVELOCITY",
        "000111" : "ANGULARVELOCITY|RAW",
        "001000" : "MAGNETICFIELD",
        "001001" : "MAGNETICFIELD|RAW",
        "001010" : "ATTITUDE",
        "001011" : "ATTITUDE|EULER",
        "001100" : "ATTITUDE|QUATERNION",
        "001101" : "ECO2",
        "001110" : "TVOC",
        "001111" : "HUMIDITY",
        "010000" : "HUMIDITY|RELATIVE",
```

```
    "010001" : "PRESSURE",
    "010010" : "TEMPERATURE",
    "010011" : "PM",
    "010100" : "PM|TPS",
    "010101" : "PM0.5",
    "010110" : "PM0.5|NC",
    "010111" : "PM0.5|MC",
    "011000" : "PM1.0",
    "011001" : "PM1.0|NC",
    "011010" : "PM1.0|MC",
    "011011" : "PM2.5",
    "011100" : "PM2.5|NC",
    "011101" : "PM2.5|MC",
    "011110" : "PM4.0",
    "011111" : "PM4.0|NC",
    "100000" : "PM4.0|MC",
    "100001" : "PM10.0",
    "100010" : "PM10.0|NC",
    "100011" : "PM10.0|MC"
}

def __init__(Self, Address, ArgumentVector, PID):
    Self.ArgumentVector = ArgumentVector
    Self.Address = Address
    Self.PID = PID

    Self.PreviousExecutionTimestamp = EpochNS()
    Self.NextExecutionTimestamp = Self.PreviousExecutionTimestamp
    Self.ExecutionCounter = 1
    Global.Modem.TxData((Self.Address, "ACK"))
    Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
```

```
    pass

def GetPriorityObject(Self):
    if Self.ExecutionCounter==0: return None
    else: return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter==0: return None
    else:
        Payload = Self.ArgumentVector[1:7]
        Payload = Ping_Hardware.HardwareFilterDecoder[Payload]
        if Payload=="RSA":
            Payload = {}
            if Global.IOThreadOnline: Payload["IOT"] = "1"; pass
            else: Payload["IOT"] = "0"; pass
            if Global.PrinterThreadOnline: Payload["PT"] = "1"; pass
            else: Payload["PT"] = "0"; pass
            if Global.TaskExecutionThreadOnline: Payload["TET"] = "1"; pass
            else: Payload["TET"] = "0"; pass
            if Global.HardwareControllerThreadOnline: Payload["HWCT"] = "1"; pass
            else: Payload["HWCT"] = "0"; pass
            if HardwareController.Initialized: Payload["HWCI"] = "1"; pass
            else: Payload["HWCI"] = "0"; pass
            Payload["TIDC"] = CompressUnsignedInteger32(Global.CurrentPID+1)
            Payload["ULTS"] = CompressUnsignedInteger64(int(EpochS()-Global.UpLinkTimeStamp))
            pass

        elif Payload=="SENSORS":
            Payload = {}
            if len(HardwareController.Sensors.keys())==0:
                Payload["N/A"] = ["N/A", "N/A"]
```

```
            pass
        else:
            SensorSet = set()
            for Sensors in list(HardwareController.Sensors.values()):
                for Sensor in Sensors:
                    SensorSet.add(Sensor)
                    pass
                pass
            SensorSet = list(SensorSet)
            SensorSet.sort(key = lambda Sensor : Sensor.NAME)
            SensorSet.sort(key = lambda Sensor : Sensor.ChannelID)
            Counter = 1
            for Sensor in SensorSet:
                Payload[str(Counter)+":"+Sensor.NAME] = [str(Sensor.ChannelID), str("0x{:X}".format(Sensor.DeviceAddress))]
                Counter += 1
                pass
            pass
        pass

    else:
        Keys = set()
        for Key in list(HardwareController.Sensors.keys()):
            if Payload in Key:
                Keys.add(Key)
                pass
            pass
        SensorSet = set()
        for Key in Keys:
            for Sensor in HardwareController.Sensors[Key]:
                SensorSet.add(Sensor)
                pass
```

```
            pass
        Payload = {}
        if SensorSet:
            SensorSet = list(SensorSet)
            SensorSet.sort(key = lambda Sensor : Sensor.NAME)
            SensorSet.sort(key = lambda Sensor : Sensor.ChannelID)
            Counter = 1
            for Sensor in SensorSet:
                Payload[str(Counter)+":"+Sensor.NAME] = [str(Sensor.ChannelID), str("0x{:X}".format(Sensor.DeviceAddress))]
                Counter += 1
                pass
            pass
        else:
            Payload["N/A"] = ["N/A", "N/A"]
            pass
        pass

    Payload = str(Payload).replace(", ", ",").replace(": ", ":").replace("'", '"')
    Self.ExecutionCounter -= 1
    Self.PreviousExecutionTimestamp = EpochNS()
    return (Self.Address, Payload)
```

```
# Low-level dependencies
from sys import maxsize as MaxSignedLongSize
from time import time_ns as EpochNS
from time import sleep as Sleep
import subprocess
import os

# High-level dependencies
from Support.NumericDe_Compression import *
import Global

class Reboot:

    def __init__(Self, Address, ArgumentVector, PID):
        Global.InboxSemaphore = True
        Self.ArgumentVector = ArgumentVector
        Self.Address = Address
        Self.PID = PID

        Self.PreviousExecutionTimestamp = EpochNS()
        Self.NextExecutionTimestamp = 0
        Self.ExecutionCounter = 1
        Global.Modem.TxData((Self.Address, "ACK"))
        Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
        pass

    def GetPriorityObject(Self):
        if Self.ExecutionCounter==0: return None
        else:
```

```
        if Self.ArgumentVector[1]=='1': Self.NextExecutionTimestamp = 0; pass
        elif Self.ArgumentVector[1]=='0': Self.NextExecutionTimestamp = MaxSignedLongSize*2+1; pass
        return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter==0: return None
    else:
        subprocess.Popen(["/bin/bash", (str(os.path.dirname(os.path.realpath("__file__")))+"/CommandRepository/_Reboot_.sh")])
        print("  ---------------------- END TRANSCRIPT ----------------------\n")
        Global.PrimeSentinel = False; Sleep(2.0)
        Self.ExecutionCounter -= 1
        Self.PreviousExecutionTimestamp = EpochNS()
        return None
```

```
# Low-level dependencies
from sys import maxsize as MaxSignedLongSize
from time import time_ns as EpochNS
from time import sleep as Sleep
import subprocess
import os

# High-level dependencies
from Support.NumericDe_Compression import *
import Global

class Shutdown:

    def __init__(Self, Address, ArgumentVector, PID):
        Global.InboxSemaphore = True
        Self.ArgumentVector = ArgumentVector
        Self.Address = Address
        Self.PID = PID

        Self.PreviousExecutionTimestamp = EpochNS()
        Self.NextExecutionTimestamp = 0
        Self.ExecutionCounter = 1
        Global.Modem.TxData((Self.Address, "ACK"))
        Global.Modem.TxData((Self.Address, CompressUnsignedInteger32(Self.PID)))
        pass

    def GetPriorityObject(Self):
        if Self.ExecutionCounter==0: return None
        else:
```

```
        if Self.ArgumentVector[1]=='1': Self.NextExecutionTimestamp = 0; pass
        elif Self.ArgumentVector[1]=='0': Self.NextExecutionTimestamp = MaxSignedLongSize*2+1; pass
        return (Self.NextExecutionTimestamp, Self)

def Execute(Self):
    if Self.ExecutionCounter==0: return None
    else:
        subprocess.Popen(["/bin/bash", (str(os.path.dirname(os.path.realpath("__file__")))+"/CommandRepository/_Shutdown_.sh")])
        print("  ---------------------- END TRANSCRIPT ----------------------\n")
        Global.PrimeSentinel = False; Sleep(2.0)
        Self.ExecutionCounter -= 1
        Self.PreviousExecutionTimestamp = EpochNS()
        return None
```

```
# Low-level dependencies
from queue import Queue, PriorityQueue

# Pin Identifiers
RX_PIN = 20
TX_PIN = 26

#####################################
# ARA Task Scheduler State Variables #
#####################################

# Core Components
Display = None # Holds the display object
Modem = None # Holds the modem object

# Status Indicators
CurrentPID = -1 # Holds the Process Identifier number of the current task (command)
UpLinkTimeStamp = None # Unix timestamp in seconds of when the RSA was booted up
IOThreadOnline = False # Boolean variable to indicate if the IO Thread is online
PrinterThreadOnline = False # Boolean variable to indicate if the Printer Thread is online
TaskExecutionThreadOnline = False # Boolean variable to indicate if the Task Execution Thread is online
HardwareControllerThreadOnline = False # Boolean variable to indicate if the Hardware Controller Thread is online

# State Controllers
PMSShutdown = False # Boolean variable to indicate if a shutdown was requested by the RSA's Power Management System
PrimeSentinel = True # Boolean variable that controls all the master loops accross all the threads
InboxSemaphore = False # Boolean variable that controls whether the IO Thread listens to the incoming messages from the client(s)

# Queues
```

```
PrintQueue = Queue(0) # FIFO queue that holds the strings that need to be printed
WaitQueue = PriorityQueue(0) # Priority Queue that organizes the tasks (commands) to be executed according to their next-requested-execution-timestamp
ExecutionQueue = Queue(0) # FIFO queue that holds the tasks (commands) that require immediate execution
Outbox = Queue(0) # Datapackets that need to be transmitted to the client(s)

# Task State records
PIDAbort = set() # A set that stores the PID of all the aborted tasks
PIDHalt = set() # A set that stores the PID of all the halted tasks
PIDComplete = set() # A set that stores the PID of all the completed tasks
TaskAbort = set() # A set that stores all the aborted tasks
TaskHalt = set() # A set that stores all the halted tasks
TaskComplete = set() # A set that stores all the completed tasks
```

```
# Low-level dependencies
from sys import maxsize as MaxSignedLongSize
from queue import Queue, PriorityQueue
from time import time_ns as EpochNS
from time import time as EpochS
from time import sleep as Sleep
from threading import Thread
from queue import Empty
import RPi.GPIO as GPIO
import subprocess
import os

# High-level dependencies
from Backend.Display.Waveshare.EPD import EPD
import Backend.HardwareController as HardwareController
from Backend.Modem.XBeeModem import XBeeModem
import Global

# Command Classes
from CommandRepository.COMMAND_MAP import GenerateCommandMap
from CommandRepository.Shutdown import Shutdown
from CommandRepository.Reboot import Reboot
from CommandRepository.Ping_Hardware import Ping_Hardware
from CommandRepository.Acquire_Data import Acquire_Data

from CommandRepository.Abort_Task import Abort_Task
from CommandRepository.Halt_Task import Halt_Task
from CommandRepository.Resume_Task import Resume_Task
```

```
def Daemon():
    print("Asynchronous Remote Acquisition Protocol")
    print("---------------------------------------")
    print("")
    print("Designed and Written by ____________")
    print("                        NUREN SHAMS/")
    print("\n")
    print("> Configuring the Power Management System Sentinel. . .", end="")
    GPIO.setwarnings(False)
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(Global.RX_PIN, GPIO.IN)
    GPIO.setup(Global.TX_PIN, GPIO.OUT)
    GPIO.output(Global.TX_PIN, False)
    print(" Success!")
    print("> Launching IO Thread. . .", end="")
    LaunchIOThread()
    Global.Modem.TxBroadcastData("END"); del Global.Modem
    GPIO.output(Global.TX_PIN, True)
    if Global.PMSShutdown:
        subprocess.Popen(["/bin/bash", (str(os.path.dirname(os.path.realpath("__file__")))+"/CommandRepository/_Shutdown_.sh")])
        pass
    return

def LaunchIOThread():
    print(" Success!")

    print("> Launching Printer Thread. . .", end="")
    PrinterThread = Thread(target=LaunchPrinterThread, args=())
    PrinterThread.start()
    Sleep(1.0)
```

```
print("> Launching Task Execution Thread. . .", end="")
TaskExecutionThread = Thread(target=LaunchTaskExecutionThread, args=())
TaskExecutionThread.start()
Sleep(1.0)

print("> Launching Hardware Controller Thread. . .", end="")
HardwareControllerThread = Thread(target=LaunchHardwareControllerThread, args=())
HardwareControllerThread.start()
while not HardwareController.Initialized: Sleep(0.1); pass
Sleep(1.0)

print("> Initializing Modem")
Global.Modem = XBeeModem()

print("\n> Configuring the Asynchronous Remote Acquisition protocol execution environment. . .", end="")
CommandMap = GenerateCommandMap()
print(" Success!")

print("> Commencing the Asynchronous Remote Acquisition Protocol Daemon")
print("\n  ---------------------- START TRANSCRIPT ----------------------")
Global.Modem.TxBroadcastData("START")
Global.IOThreadOnline = True
while Global.PrimeSentinel:

    # Check the PMS Sentinel
    if GPIO.input(Global.RX_PIN):
        print("  ---------------------- END TRANSCRIPT ----------------------\n")
        Global.InboxSemaphore = True
        Global.PrimeSentinel = False
        Global.PMSShutdown = True
        Sleep(2.0)
```

```
            break

    # Check Inbox
    if not Global.InboxSemaphore:
        DataPacket = Global.Modem.RxData()
        if DataPacket is not None:
            Global.PrintQueue.put(("> "+str(Global.CurrentPID+1)+"::"+DataPacket[1]+"\n"), False, None)
            print("  > "+str(Global.CurrentPID+1)+"::"+DataPacket[1])
            Address, Command = DataPacket
            Command = Command.split("-")
            OpCode = CommandMap[Command[0]]
            ArgumentVector = Command[1]
            Global.CurrentPID += 1
            CommandObject = eval(OpCode+"(\""+Address+"\", \""+ArgumentVector+"\", "+str(Global.CurrentPID)+")")
            PriorityObject = CommandObject.GetPriorityObject()
            Global.WaitQueue.put(PriorityObject, False, None)
            pass
        pass

    # Check Wait Queue
    try:
        Priority, CommandObject = Global.WaitQueue.get(False, None)
        if Priority<=EpochNS():
            Global.ExecutionQueue.put(CommandObject, False, None)
            pass
        elif Priority==(MaxSignedLongSize*2+1):
            for PID in range(Global.CurrentPID, -1, -1):
                if PID not in Global.PIDAbort or PID not in Global.PIDComplete:
                    break
                pass
            if PID==-1:
```

```
                    Global.ExecutionQueue.put(CommandObject, False, None)
                    pass
                else:
                    Global.WaitQueue.put((Priority, CommandObject), False, None)
                    pass
                pass
            else:
                Global.WaitQueue.put((Priority, CommandObject), False, None)
                pass
            pass
        except Empty:
            pass

        # Check Outbox
        try:
            DataPacket = Global.Outbox.get(False, None)
            Global.Modem.TxData(DataPacket)
            pass
        except Empty:
            pass

        pass

    Global.IOThreadOnline = False
    PrinterThread.join()
    TaskExecutionThread.join()
    HardwareControllerThread.join()
    return

def LaunchPrinterThread():
    print(" Success!")
```

```
    Global.Display.PrintInformationFrame()
    Global.PrinterThreadOnline = True
    while Global.PrimeSentinel:
        try:
            String = Global.PrintQueue.get(False, None)
            Global.Display.PrintText(String)
            pass
        except Empty:
            pass
        Sleep(0.1)
        pass
    return

def LaunchTaskExecutionThread():
    print(" Success!")

    # Check if the hardware controller has been initialized
    while not HardwareController.Initialized:
        Sleep(0.1)
        pass

    Global.TaskExecutionThreadOnline = True
    while Global.PrimeSentinel:

        # Check Execution Queue
        try:
            CommandObject = Global.ExecutionQueue.get(False, None)
            if CommandObject.PID in Global.PIDAbort:
                Global.TaskAbort.add(CommandObject)
                continue
            elif CommandObject.PID in Global.PIDHalt:
```

```
                Global.TaskHalt.add(CommandObject)
                continue
            DataPacket = CommandObject.Execute()
            if DataPacket is not None:
                Global.Outbox.put(DataPacket, False, None)
                pass

            PriorityObject = CommandObject.GetPriorityObject()
            if PriorityObject is not None:
                Global.WaitQueue.put(PriorityObject, False, None)
                pass
            else:
                Global.PIDComplete.add(CommandObject.PID)
                Global.TaskComplete.add(CommandObject)
                pass
            pass

        except Empty:
            pass

        pass

    Global.TaskExecutionThreadOnline = False
    return

def LaunchHardwareControllerThread():
    print(" Success!")

    HardwareController.Initialize(Verbose=True)
    Global.HardwareControllerThreadOnline = True
    while Global.PrimeSentinel:
```

```
        HardwareController.UpdateReadoutCache()
        pass
    Global.HardwareControllerThreadOnline = False
    HardwareController.Terminate(Verbose=True)
    return

if __name__ == "__main__":
    Global.UpLinkTimeStamp = EpochS()
    Global.Display = EPD()
    while True:
        try:
            Daemon()
            break
        except Exception as ExceptionInstance:
            Global.PrimeSentinel = False
            StartTimestamp = EpochS()
            while (EpochS()-StartTimestamp)<=10:
                GPIO.output(Global.TX_PIN, True)
                Sleep(0.25)
                GPIO.output(Global.TX_PIN, False)
                Sleep(0.25)
                pass
            Global.Display.PrintErrorFrame()
            Global.Display.PrintText(str(ExceptionInstance))
            StartTimestamp = EpochS()
            while (EpochS()-StartTimestamp)<=20:
                GPIO.output(Global.TX_PIN, True)
                Sleep(0.50)
                GPIO.output(Global.TX_PIN, False)
                Sleep(0.50)
                pass
```

```
# Core Components
Global.Modem.Dispose()
del Global.Modem
Global.Modem = None

# Status Indicators
Global.CurrentPID = -1
Global.IOThreadOnline = False
Global.PrinterThreadOnline = False
Global.TaskExecutionThreadOnline = False
Global.HardwareControllerThreadOnline = False

# State Controllers
Global.PMSShutdown = False
Global.PrimeSentinel = True
Global.InboxSemaphore = False

# Queues
Global.PrintQueue = Queue(0)
Global.WaitQueue = PriorityQueue(0)
Global.ExecutionQueue = Queue(0)
Global.Outbox = Queue(0)

# Task State records
Global.PIDAbort = set()
Global.PIDHalt = set()
Global.PIDComplete = set()
Global.TaskAbort = set()
Global.TaskHalt = set()
Global.TaskComplete = set()
```

```
        continue
    pass
Global.Display.ClearDisplay()
Sleep(10.0)
pass
```