Universidad del País Vasco Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE SISTEMAS ELECTRÓNICOS PARA MODELOS DE COHETE USADOS EN COHETERÍA RECREATIVA

Alumno/Alumna: FEIJOO ALONSO, ANDER
Director/Directora (1): SAINZ DE MURIETA MANGADO, JOSEBA ANDONI

Curso: 2018-2019

Fecha: 19/07/2019

ÍNDICE

Índice de figuras

Índice de tablas

1. OBJETIVO DEL TRABAJO

El presente Trabajo de Fin de Grado tiene como objetivo el desarrollo de un sistema de control para aeromodelos usados en cohetería recreativa. En dicho sistema de control se incluye, control de estabilización mediante modificación de vector de empuje, control de apogeo y adquisición de datos de vuelo.

1.1. OBJETIVOS

Los objetivos principales que se pretenden alcanzar con este trabajo son de carácter académico y técnico, entre los cuales podemos distinguir los siguientes:

Respecto a los objetivos académicos se destaca:

- Adquirir las competencias relacionadas de la realización del trabajo de Fin de Grado recogidas en la titulación de Grado en Ingeniera Electrónica Industrial y Automática

- Realizar una aplicación práctica de las competencias específicas de varias de las asignaturas cursadas en el grado (Electrónica Industrial, Regulación Automática, etc.)

Respecto a los objetivos técnicos se pretende:

- Analizar los dispositivos de posicionamiento en el espacio, para su uso con plataformas de desarrollo de microcontroladores de arquitectura ARM

- Analizar un control de estabilización mediante un doble controlador PID cerrado.

- Diseñar e implementar:
  - Una estación de pruebas estáticas para simular la respuesta del vehículo.
  - Un dispositivo de control para el modelo de cohete
  - Un sistema físico para la estabilización del motor.
  - Un pequeño modelo de cohete que albergará los dispositivos de control y estabilización.
  - Realizar un sistema adaptable para los modelos de fabricación propia de las comunidades de cohetería recreativa.

2. DATOS DE PARTIDA

2.1. COHETERÍA RECREATIVA

En términos generales y comunes, el aeromodelismo es un hobby en el que se construye y vuelan aeronaves. Generalmente, se suele hacer referencia, casi en la mayoría de las ocasiones, a aviones y helicópteros, aunque últimamente también a los multirotores, como drones, cuadricópteros o hexacópteros. En este hobby existe una vertiente, que no es muy común ni está muy extendida, que está destinada a los modelos de cohetes tanto comerciales, como de construcción propia.

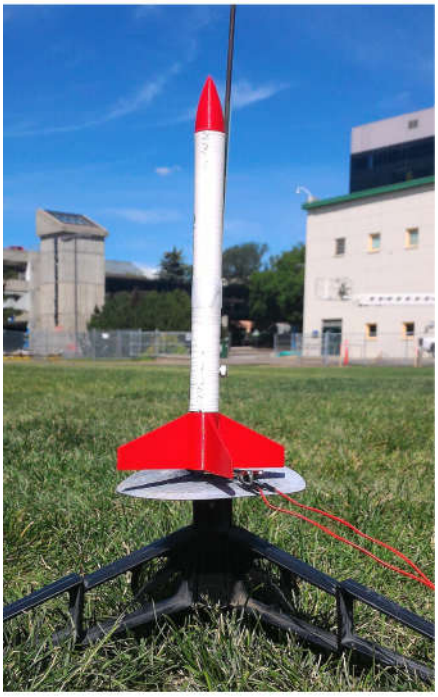


Figura 1: Modelo de cohete

Pese a haber nacido en los años 50, este hobby ha experimentado un reciente auge debido a la conquista espacial por parte de las agencias espaciales del mundo, y unido también a la fascinación del ser humano por el espacio. EEUU es el país que mayor aficionados tiene la cohetería amateur, donde se realizan conferencias, reuniones y campeonatos anualmente.

La cohetería amateur busca la experimentación a través de lo recreativo, del mundo de los cohetes, donde se construyen modelos básicos de pequeño alcance (alrededor de 1 o 2 kilómetros). Pese a ser una construcción básica, es una actividad en la cual se puede experimentar con muchos parámetros para poder mejorar el modelo y alcanzar cotas mayores.

Hasta hace pocos años, estos modelos iban desprovistos de cualquier tipo de electrónica, sin embargo, la actual incorporación de ésta a los modelos, amplía enormemente las posibilidades que permite el registro para su posterior análisis de parámetros como por ejemplo velocidad, aceleración, altitud, presión, etc.

Además de la obtención de datos, la incorporación de sistemas electrónicos, permite dotar a los modelos de cohete de sistemas de control de apogeo, de control de estabilidad y permite ensayar distintos algoritmos de control para la optimización de resultados.

2.1.1 EMPUJE VECTORIAL

El empuje es la fuerza que experimenta una aeronave mediante la cual asciende, al vencer la fuerza de la gravedad sumada a la de su propio peso.

La habilidad de una aeronave para modificar el empuje respecto del eje longitudinal del vehículo se conoce como empuje vectorial. Es un método recurrente a la hora de realizar trayectorias en aeronaves que tienen como único elemento de empuje un motor estático. Esta variación del vector de empuje, también conocida como TVC[1], se puede conseguir de varias maneras: desde elementos mecánicos, como aspas en la salida de gases del elemento propulsor, a sistemas dinámicos para orientar la cámara de combustión de manera mecánica o toberas de láminas orientables para orientar la salida de gases.

1 Del inglés Thrust Vector Control. (Control del Vector de Empuje) Mecanismo de control para modificar el vector de empuje. Logra que el empuje se incline en ambos ejes del plano XY (plano axial del modelo de cohete)

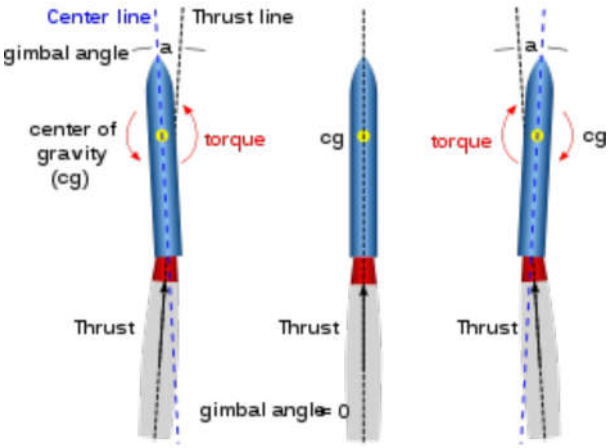


Figura 2: Ejemplo de funcionamiento del TVC

Este mecanismo de control del vector de empuje, es un método de control activo sobre la estabilidad del vuelo de un modelo de cohete. Si bien los elementos aerodinámicos también aportan estabilidad al mismo, el TVC permite la posibilidad de regular pequeñas variaciones en el primer tramo de un lanzamiento.

A día de hoy el uso de este habilidad en modelos de cohete recreativos, es prácticamente escaso debido a las condiciones de espacio que permiten esos modelos. Generalmente se usan modelos con motores fijos, donde la dirección del empuje es estática y no permite ser usada para dotar al modelo de cohete de una mayor estabilización.

Debido a esto último es común ver en los modelos de cohete, elementos aerodinámicos y guías de lanzamiento.

2.1.2 APOGEO

Segundos después de un lanzamiento, un modelo de cohete habrá gastado todo su propelente, perdiendo así su fuente de empuje. Desde ese momento, la aeronave continuará su trayectoria ascendente mientras pierde velocidad debido al rozamiento del aire y a su propio peso.

El punto máximo que alcanzará el modelo de cohete se llama apogeo. Tras ese pequeño instante de velocidad nula, la aeronave comenzará el descenso de caída libre de manera incontrolada.

Una vez iniciada la caída el modelo de cohete deberá desplegar los sistemas de recuperación que lleve integrados, tanto para recuperar el modelo con el menor daños posible, como por razones de seguridad del entorno.

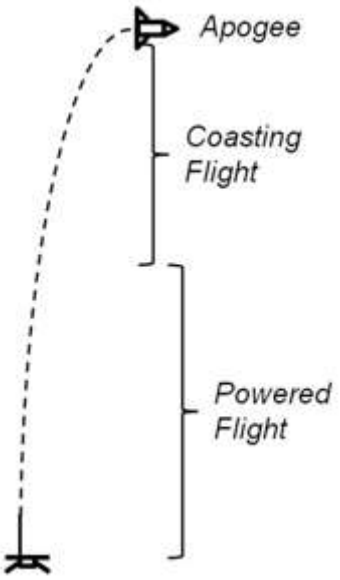


Figura 3: Fases de vuelo antes del apogeo

Los modelos de cohete comerciales, no tienen detección alguna sobre el punto de apogeo. El uso de la electrónica, además de las ventajas mencionadas en el apartado anterior, permitiría un control fiable de la altura para activar con seguridad los sistemas de recuperación, eliminando así posibles problemas y aumentando la seguridad del modelo.

2.2. ESTADO DEL ARTE

En el mundo de la cohetería recreativa existen distintos tipos de modelos. Desde los más sencillos hasta los que suponen un presupuesto elevado y que alcanzan cotas muy altas.

En cuanto a los elementos de empuje, existen de manera general dos tipos diferenciados de motores. Los de propelente sólido que son fáciles de adquirir o de fabricar, y que se usan en modelos de pequeña escala, hasta los modelos híbridos de propelente líquido que requieren una mayor inversión y que por ese motivo se usan en modelos mucho más grandes.

En cuanto a la filosofía de las comunidades de cohetería, suelen ser comunidades de aprendizaje colaborativo. Uno de los mayores pilares de desarrollo electrónico es la cultura maker. La llamada "tercera revolución industrial", representa la cultura de la la creación propia basada en la tecnología. En lo referido a la electrónica, gracias a su bajo coste y apoyado por el conocimiento abierto y las comunidades de desarrollo libre, han potenciado enormemente el uso de la electrónica como solución a multitud de problemas y/o proyectos.

En la comunidad de cohetería existen multitud de personas usuarias que no han aplicado las tecnologías usadas en la cultura maker. Generalmente en las comunidades de esta cultura hay proyectos de todo tipo y de todos los ámbitos, como por ejemplo sonido, radio-control, etc. En cambio en el ámbito de cohetería, queda relegada a cotas bastante pequeñas debido también a que no es un hobby muy extendido.

Gracias a estas comunidades podemos tener acceso a multitud de librerías de libre uso para todo tipo de dispositivos electrónicos. Uno de esos dispositivos son los sensores de movimiento inercial que es el núcleo de este proyecto. Pese a que estos dispositivos electrónicos se pueden encontrar en multitud de dispositivos, como móviles, mandos, gafas de Realidad Virtual, tienen multitud de aplicaciones finales que no han sido muy explotadas.

En lo que refiere a sistemas de recuperación en modelos comerciales, se podría decir que la totalidad de ellos usan cargas pirotécnicas para su activación. Mediante el uso de cargas retardantes de pólvora en los motores de propelente sólido, se consigue de una manera algo ineficaz desplegar los sistemas de recuperación provistos en un modelo de cohete.

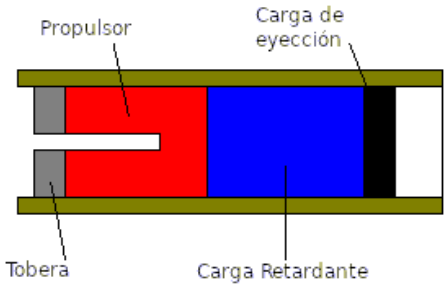


Figura 4: Diagrama de motor con carga retardante

2.3. JUSTIFICACIÓN

A la vista del estado del arte comentado en el apartado anterior, el presente proyecto pretende recoger alguna de las opciones que se han considerado más interesantes desde el punto de vista de los sistemas electrónicos, como es por ejemplo el control activo.

Por ello, dotar a un modelo de cohetería de todo el control y sensorización que nos permite la electrónica, es una idea realizable que permite profundizar más allá en el mundo de la cohetería recreativa. Su mayor potencial reside en la posibilidad de un control de estabilización para limitar los desplazamientos laterales, realizar un control preciso del apogeo para la activación de los sistemas de recuperación y a su vez la obtención de parámetros de vuelo que otorgan información relevante al usuario para mejorar su aeronave.

Este proyecto pretende ser una mejora sustancial y activa de las soluciones que se han venido usando en la cohetería recreativa a nivel bajo. Todo ello, siguiendo la filosofía de modificación libre, permite que estas soluciones basadas en sistemas electrónicos puedan ser adaptadas a las dimensiones y motores que cualquier persona usuaria quiera realizar.

Aunque este proyecto no pretende desarrollar un modelo de cohete comercial, si va a desarrollar un modelo de cohete propio así como una estructura para realizar pruebas estáticas de estabilización.

Figura 5: Modelo de cohete de fabricación propia para este proyecto

2.4. IMPACTO

El escenario actual de la cohetería recreativa está prácticamente abarrotado del uso de modelos comerciales desprovistos de estabilización activa, y en la mayoría de los casos sin ningún tipo de feedback para el usuario.

El principal problema recae en las dificultades de construcción y adaptación de un sistema de control como el propuesto, a todos los modelos comerciales, dado que está pensado para el uso en modelos completamente personalizables a gusto del usuario. A su vez, otro problema puede ser el uso de este tipo de aeronaves, puesto que sería necesario un espacio bastante grande, medidas de seguridad específicas y la obtención de permisos para realizar pruebas o lanzamientos.

Dependiendo de las plataformas de desarrollo y hardware usado se puede ajustar el coste del proyecto. Hay que tener presente que al usar plataformas de desarrollo ampliamente usadas en la cultura maker, es un proyecto que es fácilmente replicable, dado que no hay intención de crear

un prototipo comercial, si no un proyecto accesible a toda la comunidad para abordar la experimentación de la cohetería amateur.

2.5. HIPÓTESIS

Dado que el proyecto está pensando para mejorar ciertos aspectos de los modelos básicos usados en la comunidad de la cohetería amateur, el desarrollo de dispositivos de coste reducido y ampliamente usados por las comunidades colaborativas de la cultura maker, es una interesante apuesta para desarrollar un proyecto de tal implicación en la comunidad de cohetería.

Dado que es un hobby personal, este tipo de proyecto supone una motivación añadida y una excelente oportunidad para poner en práctica los conocimientos adquiridos a lo largo del grado, aparte de apuntar más allá en la adquisición de conocimientos relacionados y que permiten una experimentación añadida, tanto del campo conocido como el campo no tan conocido.

3. MARCO TEÓRICO

3.1. CONCEPTOS BÁSICOS DE COHETERÍA RECREATIVA

La cohetería recreativa es una disciplina del aeromodelismo considerada como un hobby, aunque tiene bastantes conceptos de física e ingeniería integrados en ella.

Esta disciplina consiste en diseñar, construir, lanzar y recuperar modelos de cohete con fines tanto lúdicos, como científicos. Aunque generalmente estos modelos de cohete pueden estar diseñados y construidos por uno mismo, a día de hoy, existen bastantes opciones comerciales de distintos modelos que pueden adquirirse para el fin deseado.

Un modelo consta de varias partes:

- *Elementos de propulsión* : como son los motores encargados de generar un empuje vertical ascendente.

- *Elementos de recuperación* : como son paracaídas, aparte de sistemas GPS para localizar de manera más exacta.

- *Elementos aerodinámicos* : como son las aletas y el cono, que son los encargados de dotar al modelo de estabilización aerodinámica

En los modelos comerciales, estos elementos son difícilmente intercambiables por otros. Es por ello que es bastante común fabricar modelos propios, incluidos los propios motores de propulsión, para ajustar todos los elementos en pro de superar una marca concreta, que generalmente es el apogeo conseguido.

Figura 6: Distintos tipos de motores comerciales de propelente sólido

Los motores de propulsión mayormente usados en cohetería recreativa, son los llamados motores de propelente sólido. Contienen un propelente, que es la mezcla de un combustible y un oxidante, en estado sólido. Generalmente se usa como oxidante el nitrato potásico, mientras que como combustible se usa sorbitol[2], aunque se puede sustituir por dextrosa o azúcar. [5]

2 El sorbitol es un compuesto químico de tipo polialcohol, con fórmula $C_6H_{14}O_6$, que se obtiene de la reducción del monosacárido glucosa. Se utiliza comúnmente como aditivo alimenticio, aunque su uso en cohetería es muy común.

Empuje en un modelo de cohetería recreativa

Una vez que se inicia el quemado del propelente, los gases que son producidos dentro del motor, ejercen una enorme presión y tienden a buscar una vía de escape. La salida de esos gases por la tobera dispuesta, hace aparecer una reacción de sentido opuesto, por la 3ª ley de newton, resultando un empuje vertical que siendo mayor que el peso del modelo de cohete, hará que este ascienda.

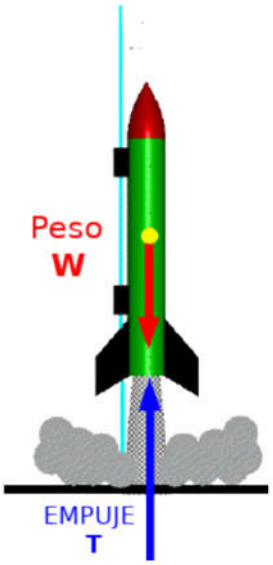


Figura 7: Diagrama empuje

Esta fuerza de empuje, como ya se ha explicado anteriormente, es una fuerza que en los modelos comerciales no se puede dirigir, ni regular. Si bien existen modelos con motores de combustible líquido cuyo empuje puede ser regulado controlando el flujo de combustible u oxidante en la cámara de combustión, en los motores de propelente sólido la combustión es completa, sin pausas, y con una curva de empuje tipo escalón, tal y como se puede ver en la Figura 8.

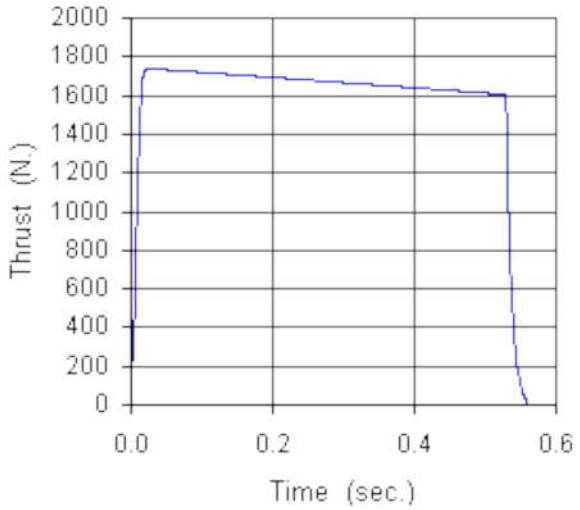


Figura 8: Curva de empuje típica de motores de propelente sólido.

Conceptos de los que depende la estabilización del modelo

El modelo de cohete está sometido a una amplia variedad de efectos durante su ascensión. Sus características dinámicas tienen un grado importante de implicación en cuanto a la estabilización se refiere.

Existen varias características en el modelo del cohete que tienen gran importancia:

- **Centro de Gravedad**[3]

Es el punto donde se concentra todo el peso del modelo. Hay que tener en cuenta que ese centro de gravedad varía a lo largo de la combustión, puesto que el propelente del motor se desprende en forma de gas, y por tanto el peso de conjunto disminuye.

Se usa el símbolo para determinar el CG.

3 Abreviado CG

- **Centro de Presiones**[4]

Es el punto donde se concentran todas las fuerzas aerodinámicas que actúan sobre el modelo. Dicho de otra forma, es el punto donde actúa la fuerza resultante de todas las fuerzas depresión que ejerce el aire en la superficie de todo el modelo durante una ascensión. Este punto depende tanto de la forma, como de los elementos aerodinámicos dispuestos sobre el modelo (cono, aletas, etc.)

Se usa el símbolo ⊙ para determinar el CP.

- **Angulo de ataque**

Es el ángulo que forma el eje longitudinal del modelo con el vector de dirección del mismo. Dependiendo de cuanto mayor o menor sea ese ángulo, mayor o menor serán las fuerzas de arrastre y la fuerza de sustentación.

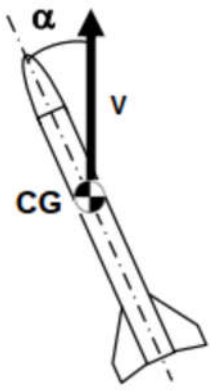


Figura 9: Representación ángulo de ataque

4 Abreviado CP

- **Fuerza de arrastre**

Es la fuerza aerodinámica que actúa sobre el CP en sentido contrario a la dirección del vuelo cuando se mueve a través del aire. Depende de la superficie de la sección transversal del cohete, por tanto, cuanto más inclinado, mayor será dicha fuerza.

La fuerza de arrastre es una fuerza que frena el movimiento del modelo de cohete. Tal y como se puede apreciar en la Figura 10.

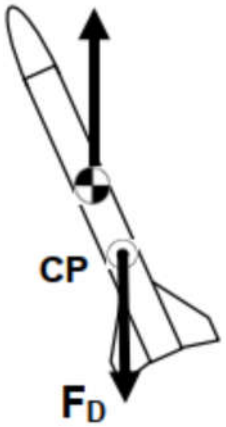


Figura 10: Representación fuerza arrastre

- **Fuerza de sustentación**

Es la fuerza que actúa de forma perpendicular al eje longitudinal en el CP. Es la fuerza resultante de todas las fuerzas aerodinámicas, y mayor responsable de que el modelo de cohete gire, alrededor de su CG. Cuando mayor es el ángulo de ataque, mayor es la fuerza de sustentación y mayor tendencia de desestabilización tendrá el modelo.

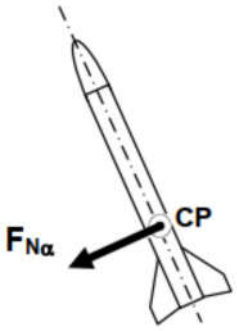


Figura 11: Representación de la fuerza de sustentación

Estabilidad en un modelo de cohete

La estabilidad en un modelo de cohete garantiza la seguridad tanto del entorno, como de las personas, por tanto es importante asegurarse que la trayectoria de vuelo será vertical.

Un modelo de cohete en vuelo libre actúa como un péndulo invertido en movimiento oscilatorio, en el que el punto de giro es alrededor de su CG. La relación entre los dos centros (gravedad y presión) se conoce como margen de estabilidad. Es la distancia entre el CP y CG. A la hora de diseñar un modelo de cohete y conocer su grado de estabilidad, hay que estudiar tanto la posición de estos dos centros como que la distancia entre ellos sea mayor a la del diámetro del modelo de cohete para considerar un vuelo estable.

Para estudiar el grado de estabilidad, habría que aplica la teoría de momentos a dicho modelo de cohete. Tal y como conocemos, el momento es la tendencia de giro alrededor de un punto al estar sometido a una fuerza. Su formula sería,

$$M = F \cdot d \tag{1}$$

Donde *F* es la fuerza de sustentación representada en la Figura 11, *d* sería la distancia entre el CG y CP, y *M* sería el momento que genera dicha fuerza en el CG.

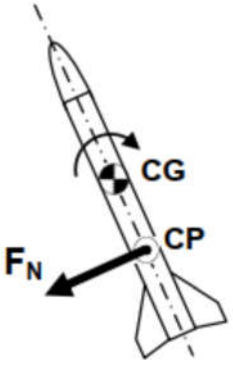


Figura 12: Representación momento de giro

Tal y como podemos ver en la Figura 12, en un modelo de cohete, el punto de giro es el CG, mientras que la fuerza a la que está sometida es la fuerza resultante en su CP. Si esta fuerza fuese prácticamente nula, la tendencia de giro sería prácticamente nula.

Cuando más se incline el modelo de cohete, mayor será su ángulo de ataque, y mayor fuerza resultante estará presente en el centro de presiones, haciendo que el modelo de cohete sea inestable debido a su tendencia a girar. La Figura 13 representa la estrecha relación entre el módulo de la fuerza de sustentación y cómo es mayor cuanto mayor es el ángulo de ataque.

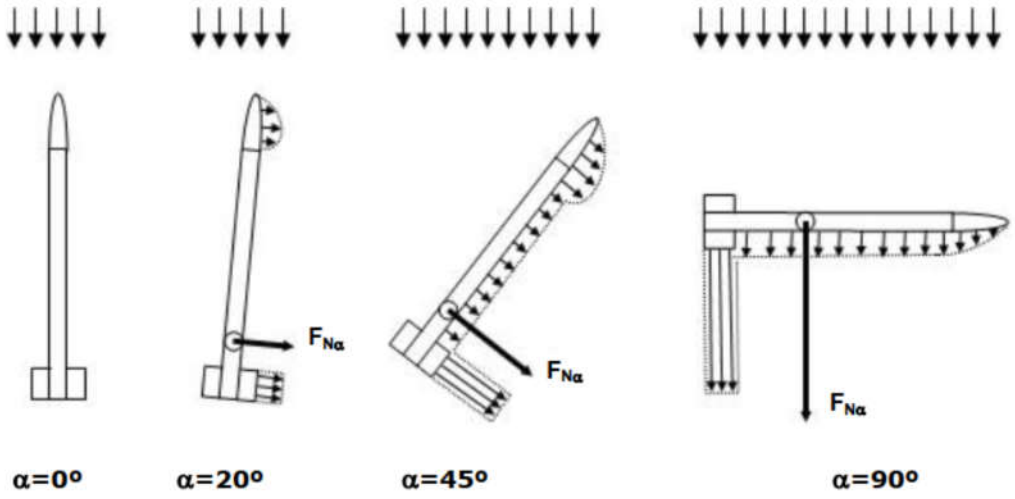


Figura 13: Relación entre ángulo de ataque y fuerza de sustentación

4. FUNDAMENTO TEÓRICO DEL PROYECTO

Mediante este proyecto se pretende innovar en dos aspectos anteriormente descritos de la cohetería recreativa en los que no se ha profundizado en exceso. El control del empuje vectorial para una mejora de la estabilización del sistema y la detección del apogeo para una activación precisa de los sistemas de recuperación.

4.1. EMPUJE VECTORIAL

Tal y como se ha especificado en el apartado anterior, uno de los parámetros más importantes es la estabilización debido a que es una característica vital para aumentar la seguridad de los modelos de cohetería recreativa. La importancia del margen de estabilidad para este cometido es crítica.

Pero existe una forma de control activo para dicha tarea que es ampliamente utilizada en cohetería profesional, como es el control del vector de empuje o TVC (nota al pie 1 en página 4). La modificación de la dirección de este vector, permite una corrección activa del ángulo de ataque y por tanto un sistema eficaz para controlar la estabilidad del modelo.

Existen multitud de sistemas para controlar el vector de empuje:

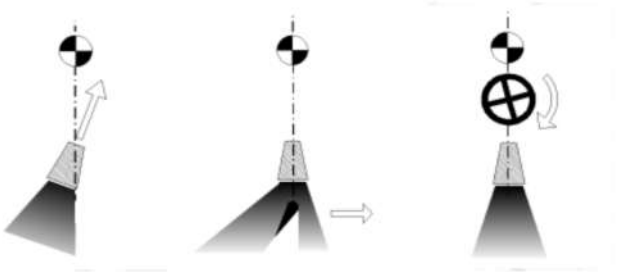


Figura 14: Distintos tipos de sistemas TVC

Según la Figura 14, tenemos de izquierda a derecha:

- *Tobera orientable* : muy común en cohetería profesional. Consta de una tobera de láminas orientables que mediante actuadores mecánicos permiten orientar la salida de gases y por tanto generar un empuje inclinado respecto al eje longitudinal.

- *Aspas* : colocación de aspas de material de gran resistencia térmica en la tobera, que orientan la salida de gases y generan un empuje direccionable.

- Motores de inercia : motores provistos de un volante de masa que generan momentos en ambos ejes en pos de inclinar el empuje.

Dado que este proyecto se ha centrado en el control de modelos de cohete con motores de propulsor sólido, se ha optado por diseñar un sistema para poder inclinar el motor. Una solución similar se puede encontrar en modelos de cohete algo más grandes, con motores de propelente líquido, los cuales tienen la posibilidad de orientar la cámara de combustión.[1]

Una solución basada en esta última, y debido a que las condiciones de espacio de los modelos de cohetería recreativa no lo permiten, ha sido implementar en un sistema gimbal de dos ejes, capaz de inclinar el motor de combustible sólido en su totalidad dentro del cuerpo del modelo de cohete. De esta manera se puede inclinar el vector de empuje, que resulta en una fuerza descompuesta en dos ejes.

La componente de la fuerza vertical continúa haciendo ascender al modelo de cohete, mientras que la componente de la fuerza horizontal, tal y como podemos ver en la Figura 15 genera un momento de giro que ayuda a la estabilización del vuelo del modelo de cohete.

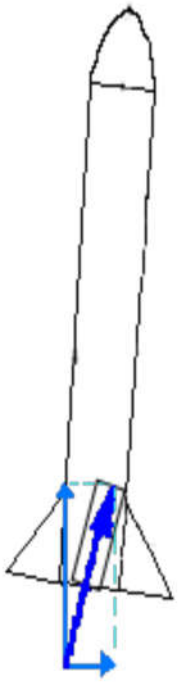


Figura 15: Vector horizontal al inclinar el motor cuando existe ángulo de ataque

4.2. DETECCIÓN APOGEO

Para realizar un control de altitud, es necesario conocer a que altitud se encuentra el aeromodelo. Este punto es importante tanto desde el punto de vista de analítico como desde el punto de vista de seguridad. Tras una motorización correcta de la altitud, se puede determinar con precisión cuando accionar el sistema de recuperación.

Figura 16: Sistema de recuperación basado en paracaídas

Otras maneras de calcular o estimar dicho apogeo, son las citadas igniciones retardadas o estimaciones de tiempo realizadas con software de cohetería, que son las que se usan en modelos de cohetes comerciales. Estos métodos son ineficaces, y pueden generar problemas de seguridad con los aeromodelos.

Para monitorizar la altura, hay multitud de dispositivos electrónicos, como pueden ser los sensores barométricos como posicionamiento GPS:

Altitud por GPS

A la hora de medir la altura con el posicionamiento GPS, el dato se obtiene de un complicado algoritmo que determina la posición tanto en el plano horizontal como en el plano vertical. La precisión de la posición en el plano vertical tiene bastante error por razones geométricas y que depende enormemente de la posición de los satélites GPS. A esto habría que

sumarle, las dificultades añadidas de la orografía del terreno, rebotes de la señal GPS en elementos como edificaciones, vegetaciones, etc.

Altitud por sensor barométrico

Estos sensores tienen un mecanismo mucho más sencillo y fiable que el anterior. Cuentan con un transductor que traduce la presión atmosférica del aire que le rodea, a distancia en metros que se encuentra sobre el nivel del mar.

Sabiendo que la presión atmosférica desciende aproximadamente 1hPa cada 8,2 metros [6], con una calibración adecuada se puede efectuar el cálculo de la altura mediante las ecuaciones pertinentes.

4.3. MICROCONTROLADOR

Todas las mejoras que pretende realizar este proyecto están basadas en el ámbito de la electrónica y en concreto de los microcontroladores.

Todo sistema o proyecto necesita de un elemento electrónico central de procesamiento. Esta unidad es el encargado de gestionar los datos de entrada, ejecutar las tareas en orden, realizar los cálculos necesarios y activar las salidas correspondientes. Es la parte más importante de todo proyecto, puesto que sin él los dispositivos electrónicos no podrían funcionar, dado que la mayoría no son autónomos.

Pese a que en el mercado hay multitud de dispositivos que pueden ser empleados como unidad central de procesamiento para adquisición de señales, control y activación de actuadores, el microcontrolador es

Figura 17: Microcontrolador comúnmente usado

uno de los dispositivos que está más extendido. Para este proyecto es una solución muy óptima, debido a su potencial, y a su tamaño relativamente pequeño.

Un microcontrolador es un circuito integrado programable, que junto con unidades de memoria y puerto de entrada /salida puede ejecutar el algoritmo con el que ha sido programado. En el caso que atañe a este proyecto, el microcontrolador es de vital importancia, pues es el elemento que va a controlar todos y cada uno de los sistemas de control, por tanto es necesario que tenga suficiente capacidad de cómputo para ejecutar las tareas de manera precisa y con suficiente celeridad.

Debido a la condiciones de espacio que hay en un modelo de cohete, el microcontrolador es una opción ideal.

4.3.1 PROTOCOLOS DE COMUNICACIÓN

Los protocolos de comunicación son las diferente formas que tienen los dispositivos de comunicarse. Tienen ciertas series de reglas que permiten la transmisión de información, que se da generalmente entre las unidades centrales de procesamiento y un número indefinido de dispositivos que generan dicha información, como sensores, o dispositivos que reciben información como actuadores.

Los protocolos de comunicación más comunes que usan los microcontroladores, son la comunicación serial, comunicación SPI o comunicación I2C. Estos dos últimos son importantes para el desarrollo de este proyecto.

4.3.1.1 SPI

El SPI, del inglés Serial Peripheral Interface, es un protocolo de comunicación síncrona entre distintos dispositivos electrónico desarrollado por Motolora en 1982. Es un estándar para controlar casi cualquier dispositivo electrónico digital que acepte un flujo de bits sincronizado mediante una señal de reloj. Permite alcanzar velocidades de transmisión alta.

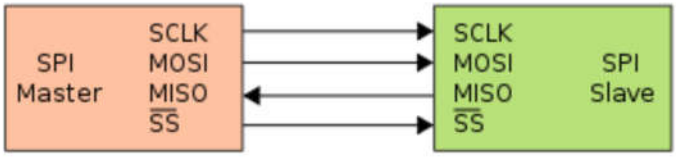


Figura 18: Diagrama de interconexión para el protocolo SPI [7]

Como podemos ver en la Figura 18 es un protocolo tipo maestro-esclavo

Este bus dispone de 4 pines sin contar las señales de alimentación:

- SCLK o SCK : Señal de reloj del bus. Esta señal rige la velocidad a la que se transmite cada bit.
- MISO(Master Input Slave Output): Señal de datos del Esclavo al maestro
- MOSI(Master Output Slave Input): Señal de datos del maestro al esclavo
- SS o CS: Chip Select o Slave Select, habilita el integrado hacia el que se envían los datos.

El funcionamiento del protocolo es la siguiente:

1. Se usa la señal SS o CS para seleccionar el dispositivo al que hay que enviar la información.
2. La señal SCK, genera el clock para sincronizar
3. Se carga en MOSI los 8 bits a enviar que estarán sincronizados con el flanco de subida de SCK
4. Una vez terminado se pone el CS en estado alto

Este sistema de comunicación permite la comunicación full-duplex. En la misma transmisión tanto el maestro como el esclavo pueden estar transmitiendo un byte.

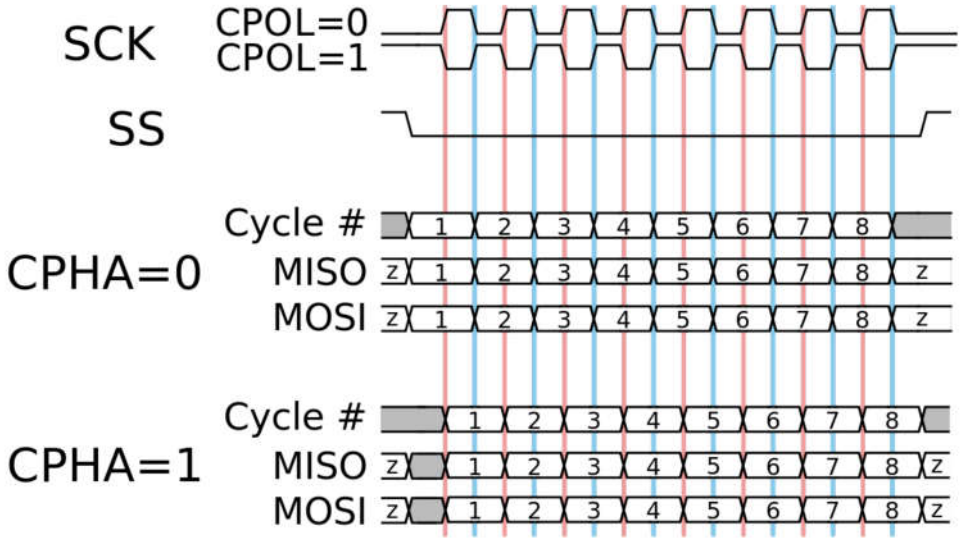


Figura 19: Ejemplo de comunicación SPI

Las principales ventajas del SPI es que es un protocolo full-duplex y tiene velocidades de transmisión altas. Su mayor desventaja es que no tiene confirmación de la recepción, lo cual podría dificultar tareas importantes u ocasionar pérdidas de datos.

4.3.1.2 I2C

El I2C, del inglés Inter-Integrated Circuit, es un bus de datos desarrollado por Philips Semiconductor en 1982. Es un protocolo de comunicación que está orientado a la comunicación entre controlador y circuitos periféricos integrados. La tipología del bus de datos es maestro-esclavo, donde la transferencia de datos es iniciada por el maestro, y el esclavo reacciona a los datos entregados [8] .

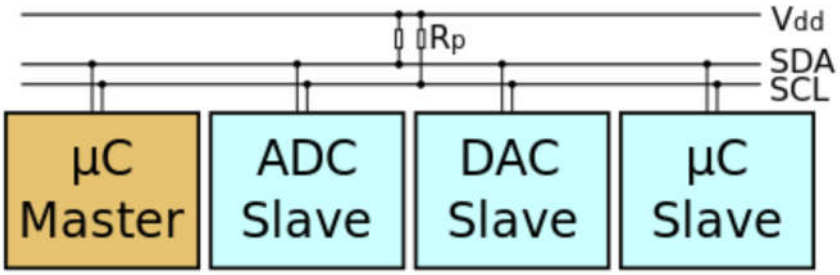


Figura 20: Esquema de conexión de dispositivos I2C

En términos eléctricos, este protocolo precisa de 2 líneas de señal. La línea de reloj SCL, y la línea de datos SDA. Ambos dispositivos deben compartir las líneas de alimentación y masa. El I2C es un protocolo síncrono. Esto significa que la señal que circula por el cable SDA está sincronizada y coordinada por la señal de reloj.

El funcionamiento del protocolo es el siguiente: Los datos que se quieran enviar son divididos en mensajes y a su vez estos en tramas de 8 bits. Cada mensaje lleva un trama con una dirección la cuál transporta la dirección binaria del esclavo al que va dirigido el mensaje, y una o más tramas que llevan la información del mensaje.

Este funcionamiento está regido por una serie de condiciones y flags tal y como podemos ver en la Figura 21.

Figura 21: Composición de mensaje en el protocolo I2C

Los flags requeridos para la comunicaciones se generan de la siguiente manera:

- START: La vía SDA pasa de nivel alto a bajo antes que lo haga el reloj.
- STOP: La vía SDA pasa de nivel bajo a alto después de que lo haga el reloj.
- Dirección: es especifica al dirección a la que se quiere comunicar. El esclavo corrobora que es su dirección para leer el mensaje.
- Bit L/E: Mediante nivel bajo, se indica si el modo es escritura (el maestro le enviará un dato) o si es lectura (el maestro solicita un dato)
- BIT ACK/NACK: Tras cada trama, para preservar la efectividad de la comunicación se usa el Bit ACK para determinar que la trama ha llegado con éxito, y NACK para indicar que esos datos no han llegado correctamente.

El protocolo I2C, tiene unas ventajas y desventajas conocidas. Cabe destacar como ventajas que solo usa dos cable de comunicación, soporta múltiples esclavos, y hay confirmaciones reiteradas del estado de la comunicación mediante los bits ACK/NACK.

Las desventajas respecto al protocolo SPI es el tamaño de paquetes reducido a 8 bits, además de ser algo más lento.

5. ANÁLISIS DE ALTERNATIVAS

5.1. EMPUJE VECTORIAL

De todas las formas de control del empuje vectorial, fueron descartadas todas debido a la imposibilidad de aplicar alguna de las soluciones a un modelo de cohete comercial por problemas de espacio.

Los motores de inercia, que a priori son la solución ideal, necesitan de un volante de masa considerable para poder generar los momentos necesarios para corregir las desviaciones del ángulo de ataque. Ese peso, sumado a las dimensiones de los motores, son un hándicap importante para el peso final de la aeronave y por extensión de su ascensión.

Exactamente lo mismo se decidió para descartar la opción de tobera orientable y aspas. No se encontró una tobera comercial de capas orientables de tamaño reducido que pudiese formar parte del sistema de propulsión de un modelo de cohete pequeño.

Con estas soluciones descartadas se optó por adaptar la solución de tobera orientable, y efectuar la orientación de todo el sistema de propulsión.

Para efectuar el control del vector de empuje, es necesario medir los grados de inclinación en cada eje del sistema.

De este cometido se encargan los sensores inerciales, o comúnmente conocidos como IMU[5].

5 Del Inglés Inertial Measurement Unit: Unidad de Medición Inercial.

Un IMU es un dispositivo que está provisto de un acelerómetro, un giroscopio y un magnetómetro. Los dos primeros dispositivos se encargan de detectar fuerzas de aceleración y velocidades angulares, mientra que el último se encarga de situar el norte magnético.

El acelerómetro tiene un funcionamiento basado en la 3era ley de newton. Cuando es sometido a una fuerza, se detectan fuerzas internas mediante efecto piezoeléctrico, las cuales se descomponen en cada eje.

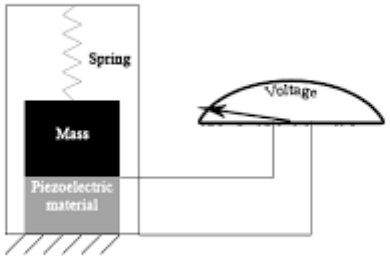


Figura 22: Efecto piezoeléctrico en acelerómetros

Por otro lado, un giroscopio detecta de manera independiente la rotación angular que sufre cada eje mediante el cálculo de la fuerza centrífuga que aparece cuando existe una rotación.

Respecto al magnetómetro, es un componente que se encarga de leer las componentes del campo magnético. De esta forma, conociendo la dirección del campo magnético terrestre podemos calcular la orientación con respecto al norte magnético de la tierra.

Con la combinación de los 9 valores y una serie de cálculos, podemos determinar que posición relativa a la tierra mantiene el sensor, su

inclinación, su movimiento angular, y como está de orientado al norte magnético terrestre.

De entre todas las opciones de IMU existentes, las siguientes son de las más usadas en el mercado:

MPU6250: Módulo bastante usado para movimientos. Esta provisto de acelerómetro y giroscopio. No tiene magnetómetro, por lo tanto sufre de un error de calculo derivado de los errores por ruido.

MPU9250: Módulo igual al anterior, pero con magnetómetro. Gracias a este último las mediciones son más exactas y se eliminan problemas que tenía el antecesor.

ICM20948: Nueva versión del anterior. Es relativamente nuevo, tiene mayor sensibilidad.

Los tres dispositivos son del mismo fabricante, Invesense. Tienen características muy similares, y acordes con las necesidades del proyecto. En la Tabla 1 podemos observar una comparación de sus principales características

Tabla 1: Comparación MPU6250, MPU9250 e ICM20948

Modelo	Rango Gyro (º/s)	Range Accel (g)	Sensibilidad (LSB/g)	Salida	Grados Libertad
 MPU6250	±250	±2 ±4 ±8 ±16	4800	i2C	6
 ICM20948	±250	±2 ±4 ±8 ±16	16384	i2C	9

 MPU9250	±250	±2 ±4 ±8 ±16	4800	i2C	9

En un principio se probó el MPU6250. Tras observar problemas de lecturas por ruido en ciertos movimientos, se optó por usar la versión MPU9250 la cual incluía magnetómetro. La nueva versión de esta familia de IMUs aún no está ampliamente testada. Se eligió el MPU9250 por ser muy utilizado y tener bastante comunidad de desarrollo detrás suyo.

5.2. DETECCIÓN APOGEO

Para detectar el apogeo, es necesario conocer la altura a la que se encuentra el modelo. Por lo tanto es necesario usar un sensor barométrico, los cuales son bastante comunes a la hora de resolver esta cuestión.

Los sensores barométricos más utilizados son el BMP180, el BMP 280 y el LPS331AP.

Las características de los 3 son bastante parecidas. Los BMP, cuyo fabricante es Bosch, tienen un rango de 300 a 1100 hPa, mientras que el rango del LPS331AP es 260 a 1260 hPa. Pese a tener un rango mayor, estaríamos hablando de un rango de -500 a 9000 metros, aproximadamente, respecto del nivel del mar. Esta especificación no fue determinante, pues era un rango que cumplía de largo con las necesidades del proyecto.

Respecto al ruido RMS, ambos tienen 0,02 hPA en el modo de alta resolución.

Ante estas características tan parejas, se decidió escoger el BMP280, dado que era más accesible y económico, además de tener un algoritmo en su firmware específico para linealizar la lectura de presión, dado que la presión respecto de la altura sigue la siguiente gráfica.

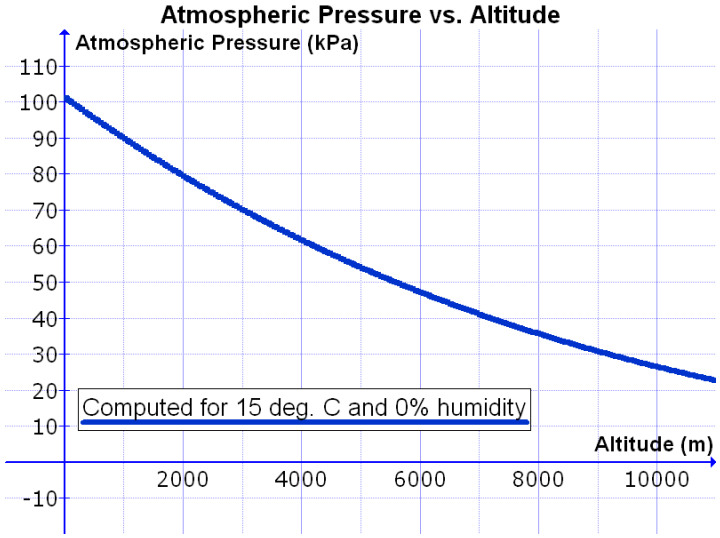


Figura 23: Relación entre altura y presión atmosférica

5.3. MICROCONTROLADOR

A la hora de decidir un microcontrolador para un proyecto, es necesario ser consciente de las necesidades técnicas que el proyecto requiere, para poder seleccionar la alternativa más idónea.

Para este proyecto es fundamental que el microcontrolador tenga una capacidad de cómputo suficiente para realizar todas las tareas, tanto

las críticas como las menos importantes, para que el proyecto sea realizable y fiable.

Entre todos los microcontroladores que se usan en el mercado, tenemos multitud de opciones. En el cultura maker es muy común la plataforma Arduino, que lleva microcontroladores del fabricante ATMEL. Esta plataforma se hizo muy popular por simplificar el acceso a los microprocesadores de bajo presupuesto, generando una comunidad de desarrollo detrás bastante grande a nivel mundial, donde podemos encontrar casi cualquier dispositivo que queramos adaptado a dicha plataforma. En cuanto a sus capacidades técnicas son, por norma general, microcontrolador de 8-bits, que funciona a una frecuencia de 16mhz, 23 pines de E/S, 3 timers entre otras características destacables.

Figura 24: Placa de prototipado Arduino con microcontrolador ATMEGA328p

Otra de esas soluciones podrían ser las FPGA. Son dispositivos hardware que se describen mediante un lenguaje VHDL, y cuyo resultado es un circuito integrado con el circuito hardware digital equivalente. Esta solución es perfecta puesto que tendríamos una solución de circuitos digitales completamente personalizable y ajustada a las necesidades del proyecto, teniendo como ventaja principal la posibilidad de ejecutar tareas de manera paralela, algo que los microprocesadores no pueden realizar, dado que son puramente secuenciales. Una de sus principales desventajas, es que al circuito integrado hay que añadirle diferentes módulos de memoria, debido a que no lo tiene integrado internamente como los microprocesadores.

Por otra parte tenemos los microcontroladores ARM que son más actuales que los microcontroladores que usa la plataforma arduino. Son circuitos integrados que disponen de una arquitectura capaz de ejecutar conjunto de instrucciones de 32 bits, los cuales son de facto mucho más potentes que su principal análogo de la plataforma arduino, recordando como añadido que tienen frecuencias de reloj muchísimo más altas (120 MHz vs 16 MHz de la plataforma arduino).

Esta arquitectura de microcontroladores se ha expandido mucho en la cultura maker, debido a que son plataformas de bajo presupuesto y gran capacidad, lo cual ha generado que la comunidad de desarrollo haya adaptado muchos firmwares de dispositivos electrónicos a dicha plataforma. Siendo ambas plataformas de bajo consumo y bajo presupuesto, pero teniendo ARM más capacidad de cómputo, lo convierte en una opción idónea.

Tabla 2: Comparativa entre ventajas e inconvenientes entre distintas plataformas hardware

	Microcontrolador 8bits	**Microcontrolador 32bits**	**FPGA**
Capacidad cómputo	Media	Alta	Muy alta
Comunidad de desarrollo	Alta	Alta	Media
Firmwares de periféricos	Disponibles	Disponibles o adaptables	Necesario Implementar

Teniendo en cuenta la Tabla 2 comparativa, se ha decidido usar la plataforma ARM, dado que la realización de FPGA se antojaba más completa pero compleja, y la plataforma ARM tiene mayor capacidad de cómputo y mayor posibilidades de conexión y comunicación con diferentes dispositivos electrónicos necesarios.

El firmware de este proyecto está escrito en lenguaje C/C++ bajo la plataforma OpenSource de desarrollo ARMbed, la cual tiene integradas multitud de placas de desarrollo en su sistema, entre las que se encuentra la seleccionada para hacer de ordenador de abordo del modelo de cohete.

La placa de prototipado elegida es una Nucleo F446RE [9]. Es un dispositivo de alta capacidad y bajo presupuesto que lleva un microcontrolador STM32F4 ARM de 32 bits del fabricante STMicroelectronics. A su vez, esta placa tiene las siguientes características importantes:

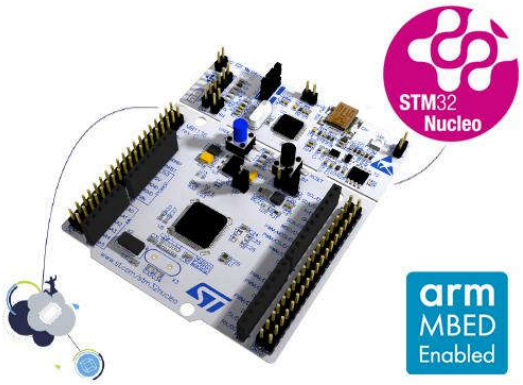


Figura 25: Placa de prototipado Nucleo F446RE

- 180 MHz de frecuencia de CPU
- 4x SPI
- 3x I2C
- 10 Timers

Así como otras características que pueden ser usadas en futuras ampliaciones del proyecto como:

- Bus can
- Interfaz de cámara

- ADC y DAC de 12-bit
- 2x UART

6. DESCRIPCIÓN SOLUCIÓN PROPUESTA

Para lograr los objetivos detallados en el punto 1, se diseña la siguiente solución como se vé en la Figura 26.

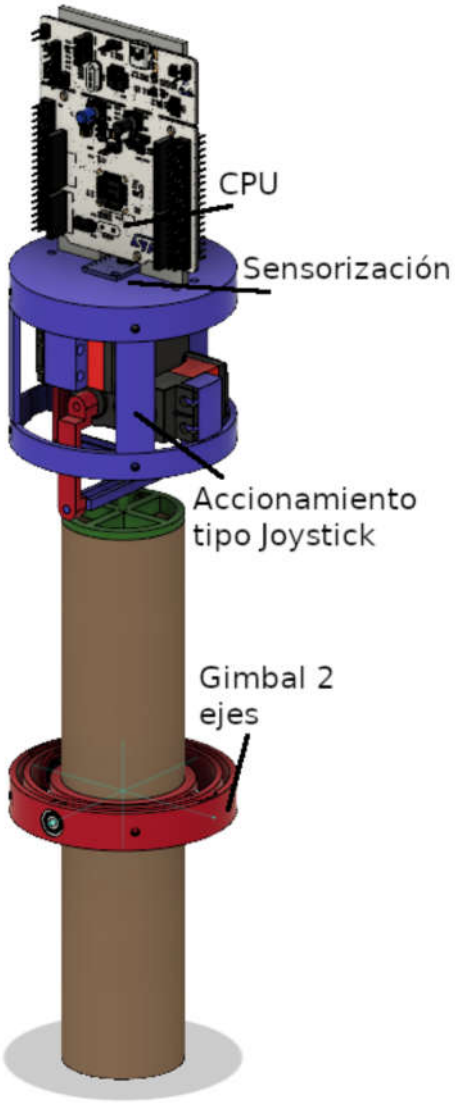


Figura 26: Esquema general del sistema integrado dentro del modelo de cohete

El esquema eléctrico del sistema es el siguiente:

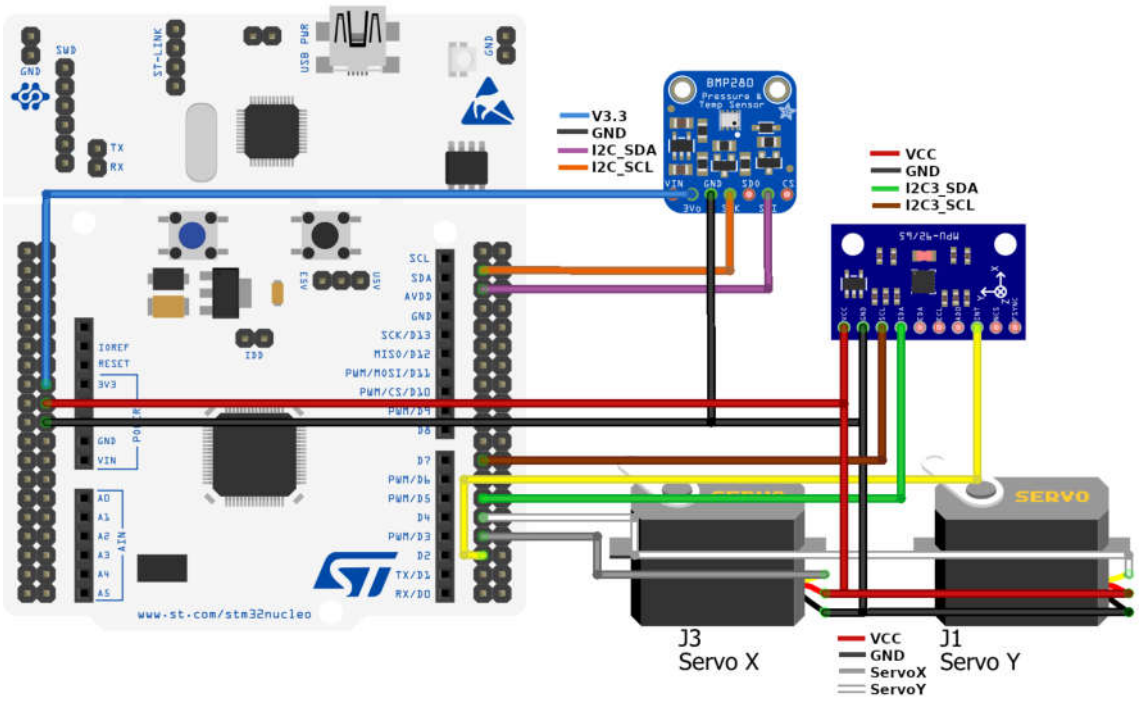


Figura 27: Esquema eléctrico del sistema

A la placa de prototipado Nucleo F446RE se conectan varios dispositivos.

Pese a que el protocolo I2C permite la conexión de varios dispositivos a las mismas líneas de datos y de reloj, esta placa tiene varios buses I2C, por lo tanto usaremos un dispositivo en cada bus. El BMP280 irá conectado a los pines I2C_SDA e I2C_SDL. Su alimentación es de 3.3.

El IMU MPU9250 está conectado a los pines I2C3_SDA e I2C3_SCL. La alimentación que se usa para este módulo es 5V.

Por otra parte, los dos servos comparten la alimentación de 5V. Las señales de control de los mismos, ServoX e ServoY están conectados a los pines PB_3 y PB_5 respectivamente.

El sistema general ejecutará el siguiente flujograma

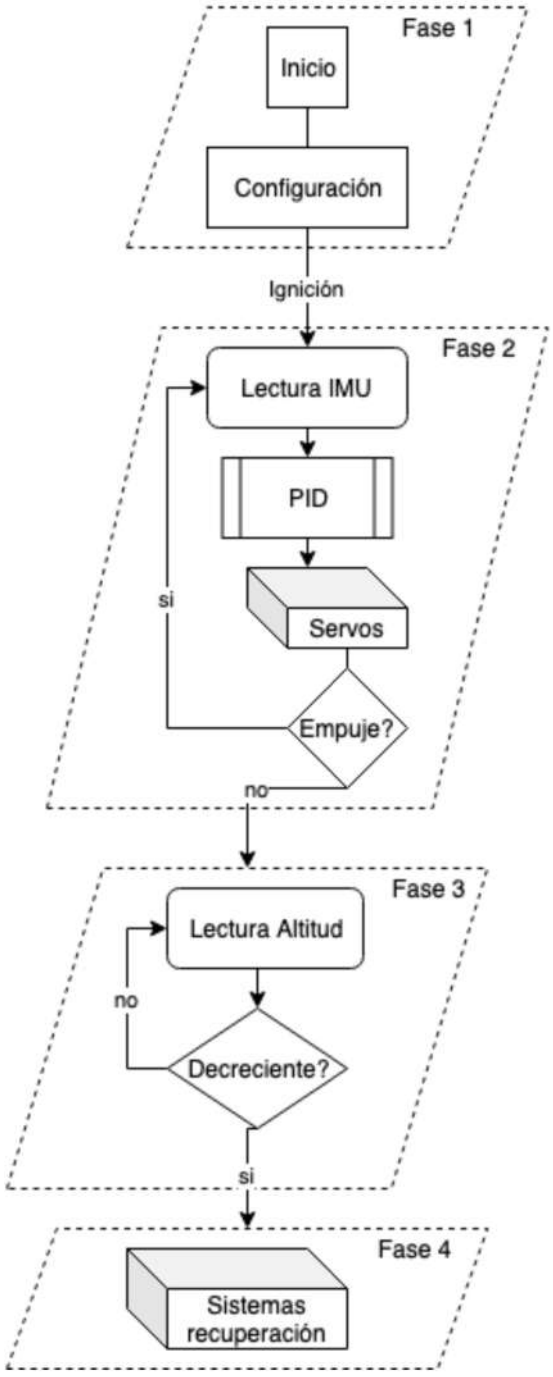


Figura 28: Flujograma del firmware

El flujo de ejecución del firmware programado que podemos ver en la Figura 28 es el siguiente:

1. Fase :Inicio y Configuración

   Se inicializan todos los sistemas y se configuran correctamente. Una vez preparado y cerciorándose que las lecturas de los dispositivos son correctas se procede a la ignición

2. Fase: Control Vector de Empuje

   Tras la ignición del motor, se monitoriza el ángulo de ataque leído con un sensor inercial. El control PID determinará cuanto hay que inclinar el motor en ambos ejes para corregir ese ángulo de ataque.

3. Fase: Altímetro

   Una vez que el propelente se ha agotado, se monitoriza la altura con un sensor barométrico. Cuando la variación sea negativa, significará que el modelo ha llegado al apogeo y está en descenso, momento que se activarán los sistema de recuperación

4. Fase: Recuperación

   El sistema de recuperación consta de un paracaídas que depositará de manera segura el modelo de nuevo en tierra.

Durante todo el funcionamiento, el ordenador de abordo monitoriza y almacena todos los parámetros leídos y los guarda en una hoja de cálculo para su posterior análisis, como son las aceleraciones en cada eje, altura, presión, temperatura, y tiempo.

Todo el diseño físico se ha realizado con el software CAD/CAM Fusion 360, dado que se han tenido que diseñar las piezas necesarias a medida. En el ANEXO 2 se podrán consultar todos los planos de las piezas que componen el sistema.

Como sistema de referencia elegido para el diseño, se establece el eje Z como el eje longitudinal correspondiente al modelo de cohete, siendo por tanto los ejes X e Y los que forman el plano axial perpendicular al mismo, tal y como se puede ver en la Figura 29.

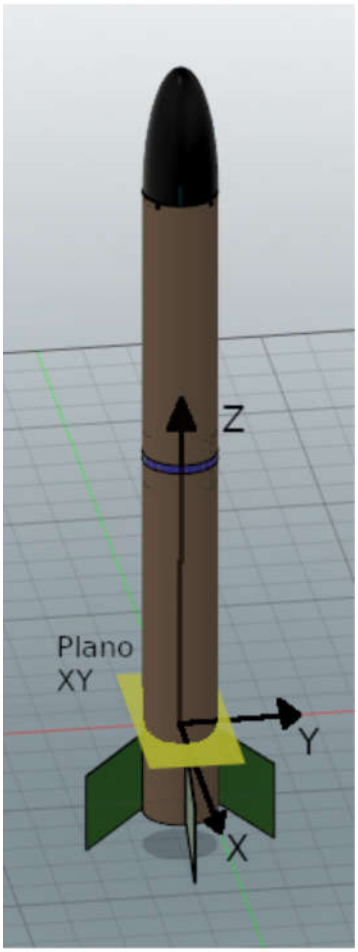


Figura 29: Ejes de referencia y plano XY

6.1. CONTROL EMPUJE VECTORIAL

La variación del empuje vertical se efectúa mediante un control PID que actúa sobre un mecanismo mecánico. Para ello es necesario diseñar un sistema físico que permita la rotación del motor y usar un IMU para conocer

la posición angular del modelo de cohete en el espacio, que será la lectura de retroalimentación del lazo de control.

6.1.1 SISTEMA DE ACCIONAMIENTO MECÁNICO.

El sistema diseñado es un mecanismo conocido como gimbal que permite dos grados de libertad, rotación respecto al plano axial del motor de propelente sólido.

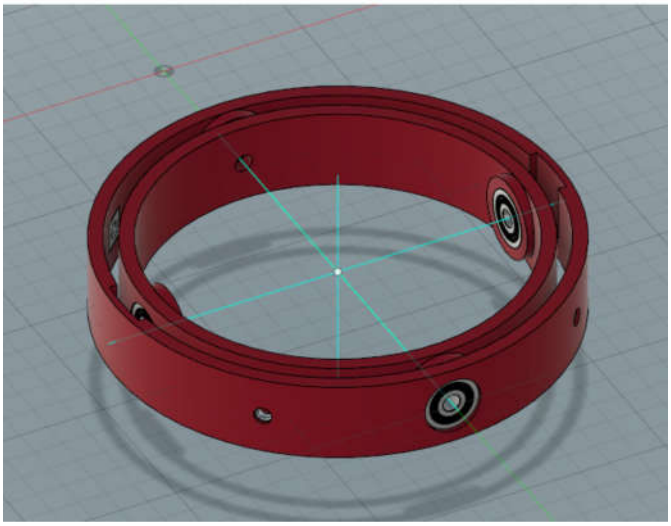


Figura 30: Sistema gimbal

Este sistema físico diseñado en 3D que se puede apreciar en la Figura 30, consta de dos anillos provistos uno dentro de otro y cada cual con dos rodamientos 623zz[6] en cada extremo del eje. El anillo exterior está rígidamente unido al cuerpo del modelo de cohete, mientras que el anillo interior está rígidamente unido a los rodamientos del anillo exterior. Estos dos ejes de rotación forman 90 grados en el plano axial que pasa por el centro geométrico del motor. Se asume que la distribución de masa del propelente sólido es homogénea y que el centro de gravedad coincide con el centro geométrico.

6 Rodamiento de bolas, de medidas: Φ exterior 10, Φ interior 3, espesor 4 (en mm)

El portamotor (encapsulado donde va alojado el motor) está rígidamente unido al anillo interior. De esta forma, este sistema permite que el motor se puede inclinar en los ejes X e Y, tal y como se puede comprobar en la Figura 31.

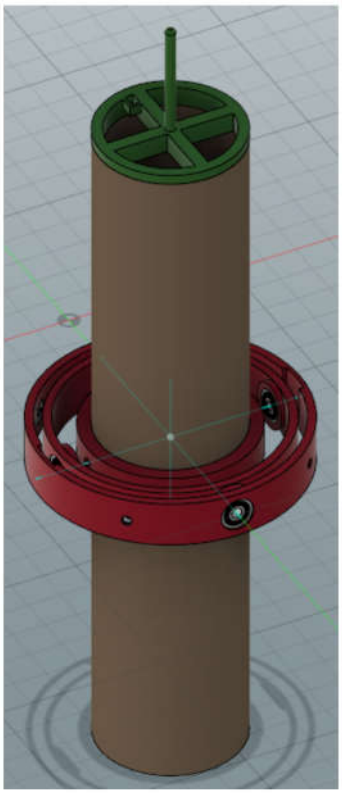


Figura 31: Portamotor rígidamente unido al anillo interior del gimbal

El accionamiento para lograr la referida inclinación se realiza con dos servomotores, uno por cada eje, mediante un sistema físico de tipo joystick. Estos servomotores van anclados a una pieza que los dispone de manera que los ejes de rotación de ambos forman 90 grados en el plano XY y están centrados respecto del eje longitudinal del modelo de cohete.

El extremo superior del portamotor dispone de una pieza con un eje que se encuentra introducido dentro de dos raíles. Estos raíles se moverán en función del movimiento giratorio de los servomotores, forzando al portamotor a girar respecto de los dos únicos ejes en los que tiene permitido girar.

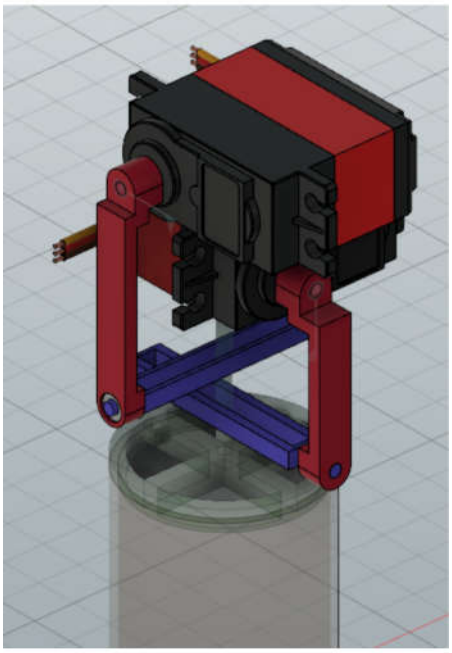


Figura 32: Sistema de accionamiento tipo Joystick

En esta imagen podemos comprobar que cuando el servomotor del eje X, gira de manera positiva, el raíl del joystick del eje X fuerza la inclinación del motor. Ocurre lo mismo cuando el servomotor del eje Y actúa; el raíl del joystick del eje Y fuerza a la inclinación en dicho eje. Este juego de movimientos se realizan de manera completamente simultánea. Es decir, son movimientos independientes que se pueden realizar de manera simultánea sin interferir entre sí.

En cuanto a los actuadores usados son dos servomotores comerciales de la marca JX modelo PDI-6208MG. Funcionan en un rango de voltaje de 4,8V a 6V. Su característica más importante es su rapidez de

funcionamiento, siendo capaces de girar 60º en 0,09 segundos con un torque de 6,8kg. Aunque la característica del torque no sea crítica en este proyecto, debido a que los motores de propelente sólido pesan mucho menos, si que es interesante la rapidez de actuación, dado que permite una acción directa y rápida, siendo esto último un factor importante a la hora de usar un mecanismo para modificar el vector de empuje.

Figura 33: Servomotores PDI6208MG

6.1.2 SENSOR INERCIAL

Mediante el sensor inercial MPU9250, se calculan los ángulos de inclinación del modelo de cohete en el espacio. Este dispositivo, usa el protocolo de comunicación I2C para transmitir los datos al ordenador de abordo, el cual solicitará los valores de aceleración, velocidad angular y campo magnético en cada eje para realizar los cálculos necesarios.

Debido a que la ejecución de este tarea es una parte crítica para el cálculo de la posición relativa en el espacio, es necesario que sea lo más determinista posible. Es de vital importancia que las lecturas se realicen con una frecuencia y que esta permanezca invariable. De esta manera tendremos lecturas fiables para el lazo de control.

Este cometido se realiza con interrupciones. El propio chip MPU9250, tiene la función de generar una interrupción hardware. Genera una señal cuadrada de 200hz en el pin INT, que puede ser usada como interrupción general en el sistema.

El sensor inercial MPU9250, se dispondrá cerca del centro de gravedad del modelo de cohete y está rígidamente al cuerpo del mismo. Asimismo el eje longitudinal Z del circuito pasa por el centro longitudinal del cuerpo del modelo de cohete.

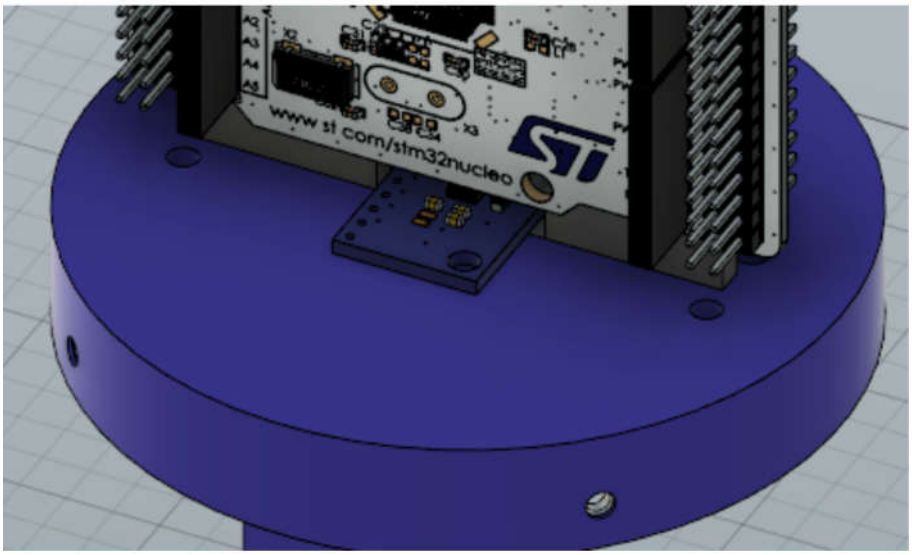


Figura 34: Posición del IMU

Este sensor tiene la posibilidad de configurar las resoluciones con la que queremos leer los datos de aceleración, velocidad angular y campo magnético por cada eje. La configuración que debe tener es la siguiente:

- Resolución Acelerómetro: 2G
- Resolución Giroscopio: 250 DPS
- Resolución Magnetómetro: MFS_16 bits

Esta configuración se usará posteriormente para calcular la resolución del dispositivo.

Una vez el dispositivo está configurado, realiza una serie de medidas en estado estacionario, para calcular los errores, que posteriormente serán sustraídas de las medidas.

6.2. CONTROL ALTURA

El sensor barométrico BMP280 está conectado mediante el protocolo de comunicación I2C con el ordenador de abordo, quién realiza lecturas durante todo el recorrido del lanzamiento del modelo de cohete. Una vez que el ordenador de abordo establezca la fase de recuperación, se realiza un control sobre dicha lectura.

Cuando la lectura tenga una variación negativa, implica que el modelo de cohete ha llegado al apogeo y que éste comienza a descender. En ese momento el ordenador de abordo activará los sistema de recuperación del modelo de cohete.

El control será un lazo cerrado simple, de ejecución continua. Durante ese proceso se continua guardando en la memoria todos los parámetro de vuelo.

Los parámetros con los que ha sido configurado el BMP280 según datasheet son:

- Modo: MODE_NORMAL
- Muestreo de temperatura: SAMPLING_X2
- Muestreo presión: SAMPLING_X16
- Filtrado: FILTER_X16
- Tiempo de standby: STANDBY_MS_500

6.3. ORDENADOR DE ABORDO

El ordenador de abordo se encarga de efectuar el flujograma de la 41. El firmware programado, efectúa las distinta fases del lanzamiento, así como realizar las lecturas del sensor IMU con una frecuencia de 200hz y las lecturas del sensor barométrico para monitorizar la altitud a la que se encuentra el modelo de cohete.

Este dispositivo está anclado mediante una pieza diseñada en 3D para evitar movimientos internos cuando se encuentra el modelo en funcionamiento.

Figura 35: Posición como ordenador de abordo

6.4. ESTACIÓN DE PRUEBAS

La estación de pruebas en un elemento para poder simular y verificar el control del vector de empuje. De esta manera podremos ajustar mejor los valores del lazo de control para realizar una rectificación más ajustada.

Figura 36: Estación de pruebas con el modelo de cohete instalado

La estructura está realizada con tubería de PVC la cual dentro va reforzada con cemento. De esta manera la estructura gana rigidez y peso, dado que es importante que la única parte móvil del sistema sea el modelo de cohete.

El modelo de cohete esta dispuesto dentro de un sistema gimbal de dos ejes como se puede apreciar con más exactitud en la Figura 37.

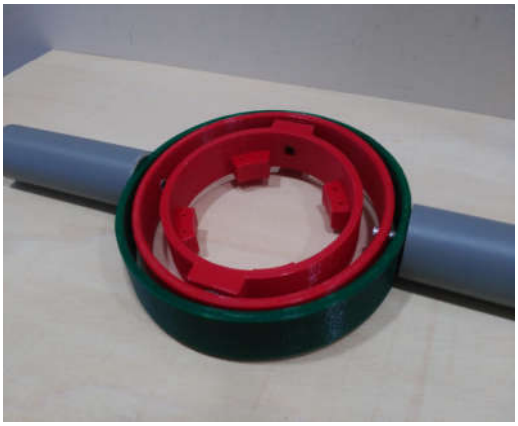


Figura 37: Sistema gimbal de la estación de pruebas.

Este mecanismo es exactamente igual que el sistema gimbal diseñado para el portamotor en el apartado 6.1.1 pero de dimensiones mucho mayores, debido a que tiene que sostener el modelo de cohete en su totalidad.

Los rodamientos usados en este gimbal, son los 608ZZ, que al ser más grandes, permiten soportar más fuerzas. El anillo interior en este caso tendrá unas fijaciones para anclar el modelo de cohete y que permanezca rígido.

De esta forma el movimiento del modelo de cohete dentro de la estación de pruebas está limitado a dos grados de libertad, que son los giros en el eje X e Y respecto al plano terrestre.

La dimensiones de esta estructura se pueden consultar en el ANEXO 2. Hay que tener en cuenta que está pensada para testear motores de propelente sólido los cuales tienen una deflagración considerable, que podemos apreciar en la Figura 38. Por lo tanto dicha salida de escape de gases no debería estar muy próxima a la estructura en cualquier parte de su rango de movimientos.

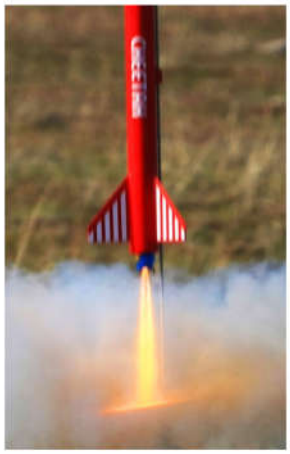


Figura 38: Deflagración de un motor de propelente sólido

6.5. MODELO DE COHETE

El modelo de cohete de pruebas para implementar y testear el sistema explicado, es de diseño y fabricación propia, debido a cuestiones de presupuesto y dificultad de obtención de modelos comerciales.

El proceso de diseño se ha realizado mediante el software OpenRocket, el cual es una alternativa de software libre apropiada para la modelización de cohetes.

Este software está pensando para el diseño y simulación de modelos de cohete. De esta manera se pueden estudiar posibles mejoras a la hora de diseñar un modelo. Se pueden realizar cálculos de trayectoria, altitud, tiempos, etc... con un rango ajustado mediante las distintas fórmulas matemáticas que tiene en su algoritmo de simulación.

Aparte de ello, dispone de una amplia variedad de componentes para su diseño, desde elementos aerodinámicos, como motores comerciales. Su punto más fuerte es el cálculo y visualización gráfica del CG como del CP, permitiendo diseñar el resto de componentes en función de ambos puntos.

Figura 39: Diseño realizado con el software OpenRocket

Este programa no dispone de elemento de control activo como el sistema de control del vector de empuje diseñado. Con motivo de preservar la fiabilidad del punto de CG se han introducido como componentes de masa.

Las partes que se han desarrollado para el modelo del presente trabajo son: el cuerpo, el cono y el motor.

Cuerpo

El modelo consta de un cuerpo tubular de cartón de 750mm de largo, 80mm de diámetro interior y espesor de 1,6mm. Esta construido con cartón prensado. Debido a la relación robustez/peso es una de la soluciones más comunes usadas en cohetería para la fabricación de modelos.

En su interior van alojados el sistema de accionamiento del empuje vectorial, el ordenador de abordo y el sistema de recuperación, los cuales están rígidamente unidos al cuerpo del modelo de cohete mediante tornillos M3 para evitar movimientos internos.

Para facilitar el montaje interno de los diferentes sistemas, se ha seccionado el tubo en 2 partes, que son acoplados posteriormente a una pieza 3D y atornillada convenientemente.

Cono

El cono ha sido diseñado con un perfil aerodinámico en forma de media elipse de 30mm de longitud larga y 83,2mm de longitud corta, con un espesor de 1mm. Su fabricación ha sido mediante impresión 3D.

Motor

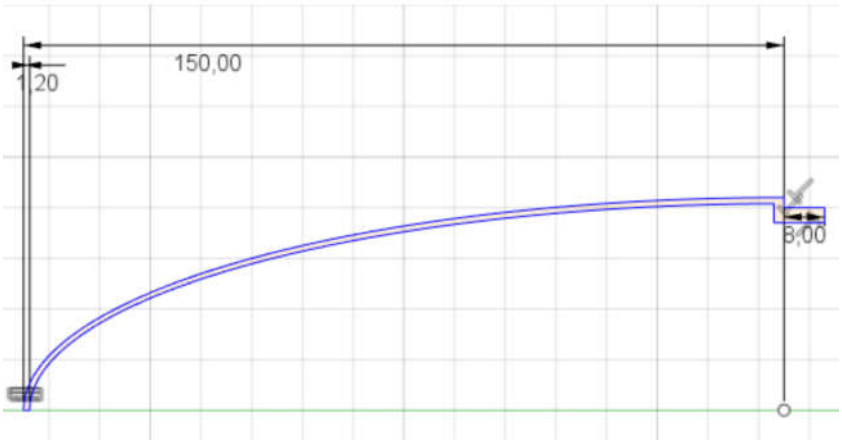


Figura 40: Sección transversal del cono

Respecto al motor, este modelo de cohete está pensando en ser versátil, pudiendo portar motores comerciales tanto motores de fabricación propia. En el caso del software de diseño, se ha escogido un motor comercial que posteriormente sería replicado en fabricación propia.

El motor es de la marca AeroTech. Tiene 29mm de diámetro y una longitud de 172mm, con un impulso total de 200N.

Para la realización de pruebas de estabilidad en la estación de pruebas, en pos de realizar tests de manera más cómoda y evitar el gasto de motores de propelente sólido, se dispone de un sistema de turbina eléctrica para simular el efecto de empuje vertical

La turbina eléctrica tipo EDF[7] de 35mm de diámetro, que se acciona con un ESC de 20A con un empuje total de 200g.

7 Del Inglés: Electric Ducted Fan.

7. CÁLCULOS Y PROGRAMACIÓN

En este apartado se explican los cálculos que se realizan en el proyecto.

7.1. CÁLCULO DE POSICIÓN

Primeramente necesitamos conocer los valores de aceleración, velocidad angular y campo magnético por cada eje, los cuales el MPU9250 los almacena en los registros indicados en su datasheet que podemos encontrar en el ANEXO 1

Estas lecturas se realizan con la función ReadRawAccGyroMag() que se detalla a continuación

```
1 void MPU9250::ReadRawAccGyroMag(){
2    readAccelData(accelCount);  // Read the x/y/z adc values
3    AccelXYZCal();
4
5    readGyroData(gyroCount);  // Read the x/y/z adc values
6    GyroXYZCal();
7
8    readMagData(magCount);  // Read the x/y/z adc values
9    MagXYZCal();
10      }
```

Esos 9 datos tenemos que adaptarlos al factor de resolución configurado en el dispositivo y restarle el error medido en estado estacionario.

Los valores de aceleración, velocidad angular y campo magnético se usan en un algoritmo para sistemas de referencia de orientación y rumbo. Este algoritmo llamado madgwickQuaternionUpdate() emplea una representación de la orientación por medio de cuaterniones. Calcula las

orientaciones a partir de las velocidades angulares, y las fusiona con las estimaciones de la orientación a partir de los vectores medidos del campo gravitacional y del campo magnético. Por último normaliza el cuaternio para tener una salida estable. Este algoritmo esta explicado en el artículo [4].

Posteriormente al cálculo del cuaternio, es requerido realizar una conversión a los ángulos Tait-Bryan, que son los conocidos Yaw, Pitch y Roll que se usan de manera generalizada en el mundo del aeronáutica.

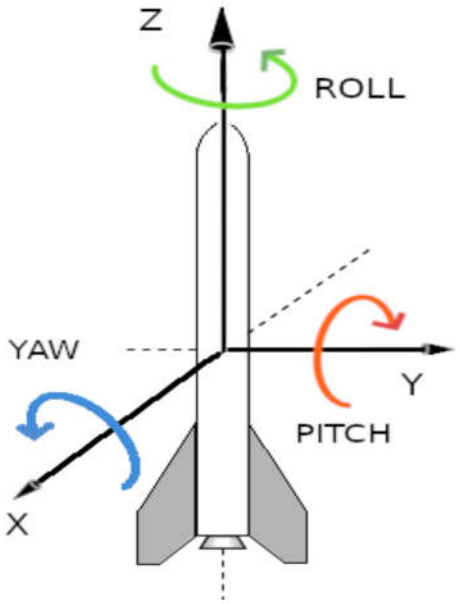


Figura 41: Diagrama de giros Yaw, Pitch, Roll

Teniendo el vector del cuaternio, se puede representar como una matriz de rotación

$$q=\{x,y,z,w\} \quad M_q=\begin{bmatrix} \dots & 2xy-2wz & \dots \\ \dots & 1-2x^2-2z^2 & \dots \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{bmatrix} \tag{2}$$

Asimismo la matriz rotación de un cuaternio para adecuarlo al sistema de ángulos de Tait-Bryan, es

$$M_r=\begin{bmatrix} c\psi\, c\phi - s\psi\, s\theta\, s\phi & -s\psi\, c\theta & c\psi\, s\phi + s\psi\, s\theta\, c\phi \\ c\psi\, s\theta\, s\phi + s\psi\, c\phi & c\psi\, c\theta & s\psi\, s\phi - c\psi\, s\theta\, c\phi \\ -c\theta\, s\phi & s\theta & c\theta\, c\phi \end{bmatrix} \quad (3)$$

$$nota: con\ y\ \sin\ han\ sido\ abreviadas\ por\ c\ y\ s$$

De ambas matrices podemos determinar entonces

$$\begin{aligned} -\tan\psi &= \frac{2xy-2wz}{1-2x^2-2z^2} \rightarrow \psi = \arctan\left(\frac{-2xy+2wz}{1-2x^2-2z^2}\right) \\ \sin\theta &= 2yz+2wx \rightarrow \theta = \arcsin(2yz+2wx) \\ -\tan\phi &= \frac{2xz-2wy}{1-2x^2-2y^2} \rightarrow \phi = \arctan\left(\frac{-2xz+2wy}{1-2x^2-2y^2}\right) \end{aligned} \quad (4)$$

Esos ángulos corresponden al Yaw, Pitch y Roll de cualquier sistema aeronáutico. En nuestro caso el Roll no es necesario, dado que la rotación sobre su eje longitudinal es un elemento que no pretende ser regulado.

Estos ángulos, expresado en grados, tienen unos valores límites de 90 a -90. Estos valores son la entrada de retroalimentación del control PID que se dispone para regular el ángulo de ataque.

Cabe destacar que al valor de inclinación en Pitch hay que ajustarlo según la declinación magnética respecto del polo. En este caso, en Bilbao a fecha 1/2/2019 la declinación magnética era de - 0º 23' W, por tanto ese valor debe ser compensado en el cálculo de dicho ángulo.

7.2. CÁLCULO DE ALTURA

Para calcular la altura a la que se encuentra el sensor barométrico, es necesario la lectura de registros que nos indica el fabricante, para poder aplicar vía firmware la función del cálculo de altura. Dicha lectura se efectúa vía I2C.

Previamente al funcionamiento normal, es necesario efectuar la lectura de los valores de calibración proporcionados por el fabricante. Que constan de 9 registros destinados a valores de presión y 3 destinados a los valores de calibración de la temperatura.

Según datasheet, los registros a leer son REGISTER_PRESSUREDATA y REGISTER_TEMPDATA, que corresponden a los registros 0xF7 y 0xFA respectivamente, tal y como se puede comprobar en la página 24 del datasheet del ANEXO 1.

Mediante los valores de esos registros, aplicamos el algoritmo del ANEXO 1, pagina 22 que corresponde con la fórmula hipsométrica.

$$Altitud = \frac{\left(\left(\frac{P_0}{P}\right)^{\left(\frac{1}{5,257}\right)} - 1\right) \cdot \left(T + 273,15\right)}{0,0065} \tag{5}$$

De donde P_0 es la presión a nivel del mar y T la temperatura.

Esta ecuación, ampliamente usada en meteorología, relaciona el cociente entre presiones atmosféricas con el grosor de las capas atmosféricas en función de la temperatura. Es la ecuación resultante de la derivación de la fórmula hidrostática y la ley de gases ideales.

7.3. CONTROL PID DEL VECTOR DE EMPUJE

El control del vector de empuje se realiza con un control de lazo cerrado, debido a que existe retroalimentación del estado de la salida. Dicha retroalimentación es el ángulo de inclinación del modelo de cohete. Para ser más exactos, dado que tenemos la lectura de inclinación de cada eje por separado, habría que implementar dos lazos de control cerrado completamente independientes.

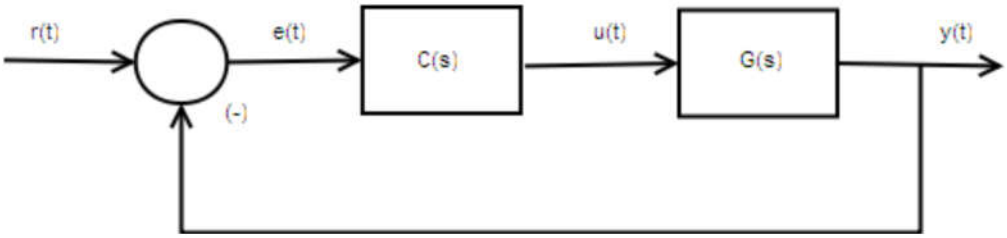


Figura 42: Esquema de control de lazo cerrado

En la Figura 42 podemos ver un lazo de control cerrado, donde G(s) es la planta o sistema a controlar y C(s) el controlador, que estabiliza la salida en función de la señal de consigna r(t)

Para el control del vector de empuje, existen multitud de opciones completamente válidas dentro de la familia de los controladores, ya sea el tipo PD, PI o PID, los cuales son los más comunes.

El controlador PD es un control anticipativo. La parte derivativa calcula la pendiente del error, y la emplea para anticipar la dirección del mismo y corregir la salida. La parte derivativa depende de si el error es estable o varía muy lentamente, en cuyo caso solo tendrá acción la parte proporcional. Es un controlador de acción rápida pero como desventaja, presenta una posible ampliación de las señales de ruido.

Por otra parte, el controlador PI mejora el error en estado estacionario, dado que reduce el error cuando éste es estable, pero al mismo tiempo la parte integral empeora la estabilidad relativa, dado que aumenta el sobreimpulso de la respuesta transitoria y las oscilaciones.

En cambio el controlador PID, reúne las cualidades de los dos tipos de control anteriores. Aumenta la estabilidad gracias a la parte derivativa y proporcional, la rapidez gracias a la parte derivativa, y la exactitud gracias a

la parte integral. Debido a el sistema a implementar en el modelo de cohete, es conveniente usar un controlador PID por estas dos últimas características.

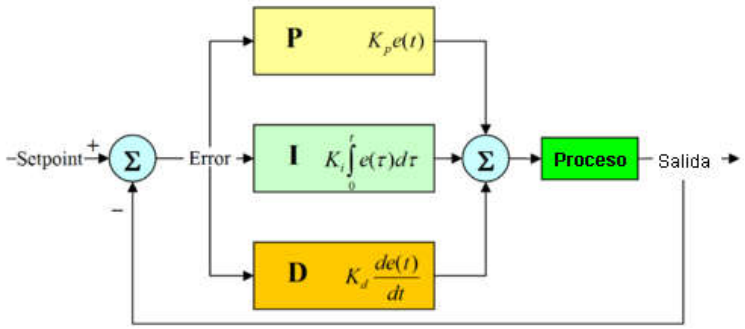


Figura 43: Esquema de control de lazo cerrado con controlador PID desglosado

En la Figura 43 podemos observar el desglose de la parte proporcional, integral y derivativa que componen un PID

A la hora de elegir y configurar un PID, es necesario conocer la función de transferencia de la planta del sistema. Como se ha mencionado anteriormente en el apartado "Estabilidad en un modelo de cohete" el sistema se comporta como un péndulo invertido.

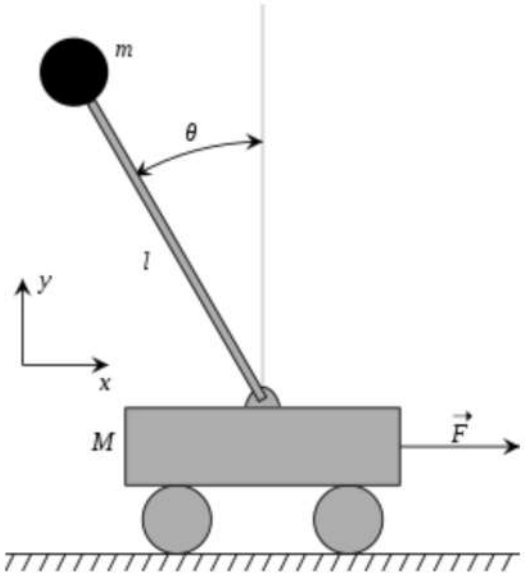


Figura 44: Diagrama péndulo invertido

La diferencia con principal con el sistema de péndulo invertido, es que la fuerza horizontal aplicada al carro, en este proyecto corresponde a la fuerza horizontal que aparece cuando inclinamos el motor en dicho eje, tal y como viene reflejado en la Figura 15.

Al no ser objetivo del proyecto el desarrollo matemático de la modelización del sistema, se parte de un modelo matemático estándar de péndulo invertido, que se puede encontrar detallado en [2],[3]

$$Planta(s)=\frac{1{,}045\,e^{-5}s}{2{,}3\,e^{-6}s^3-0{,}7s^2-0{,}5\,s-1{,}025\,e^{-5}} \tag{6}$$

Mientras que el modelo matemático del servomotor sería

$$Actuador(s)=\frac{0.7}{0.01\,s+1} \tag{7}$$

Con la ecuación (6) y (7) se designa en el software MatLab mediante la herramienta Simulink el lazo de control.

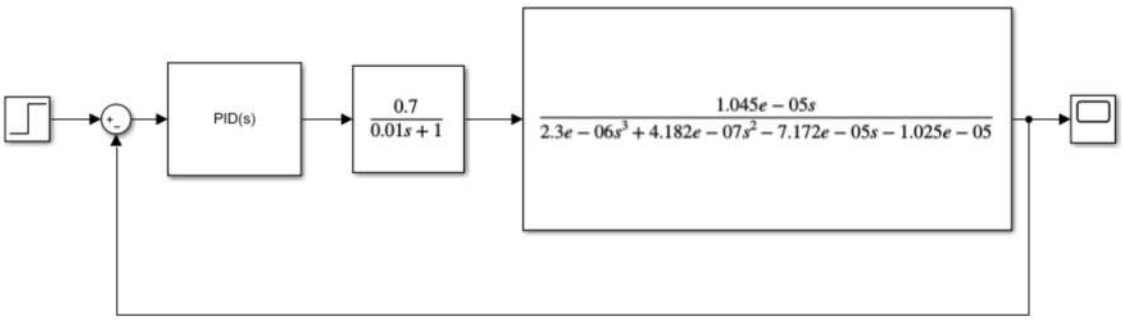


Figura 45: Lazo cerrado de control realizado y simulado con Simulink (MatLab)

Existen varias formas para sintonizar y ajustar los valores Kp, Ki y Kd del PID en pos de una estabilización correcta. Una de ella es a prueba y error, visualizando y midiendo como responde el sistema, y ajustar los valores en una secuencia de pasos.

Otra forma más óptima para realizar la sintonización es usar la herramienta integrada en MatLab llamada "PIDTurner".

Mediante ésta, podemos ajustar dichos valores en función de la curva de salida, para estimar el amortiguamiento, rebasamiento y tiempo de estabilización que otorgaría ese controlador PID al sistema.

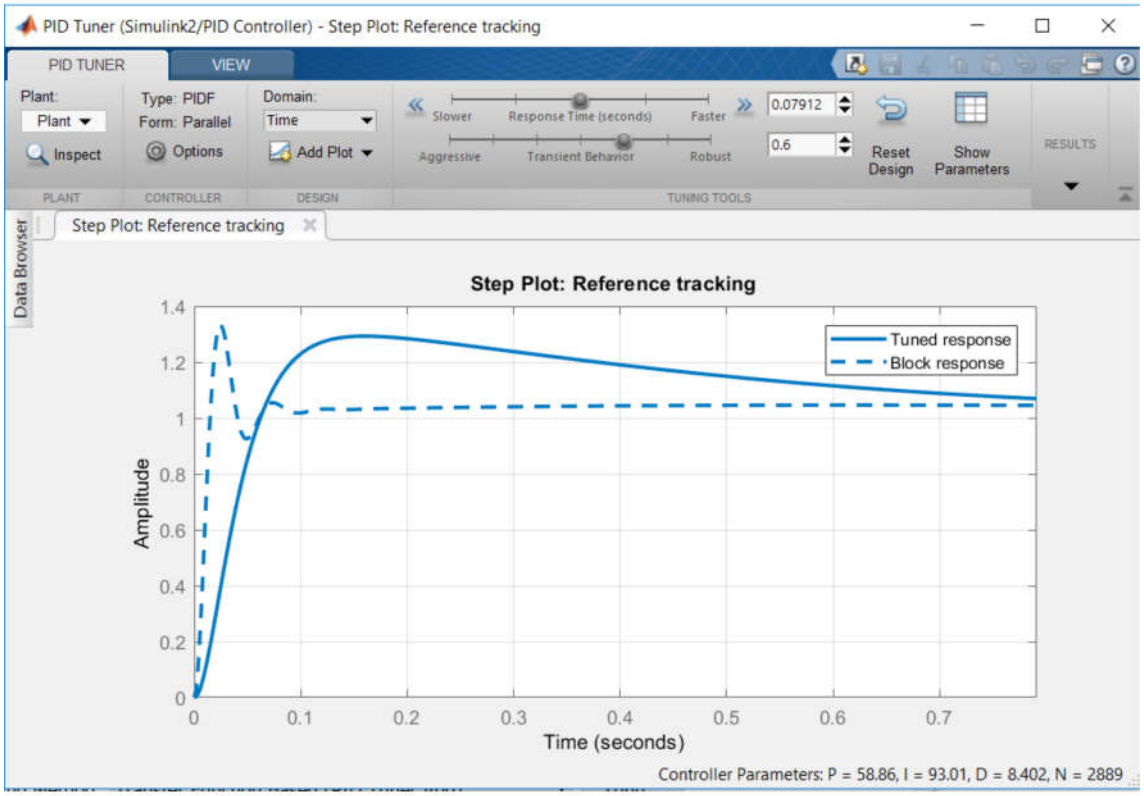


Figura 46: Sintonización de valores K de un PID mediante la herramienta PIDTurner

Una vez hallados los valores Kp, Ki y Kd, y ajustado el PID, podemos observar la salida del sistema ante una entrada escalón.

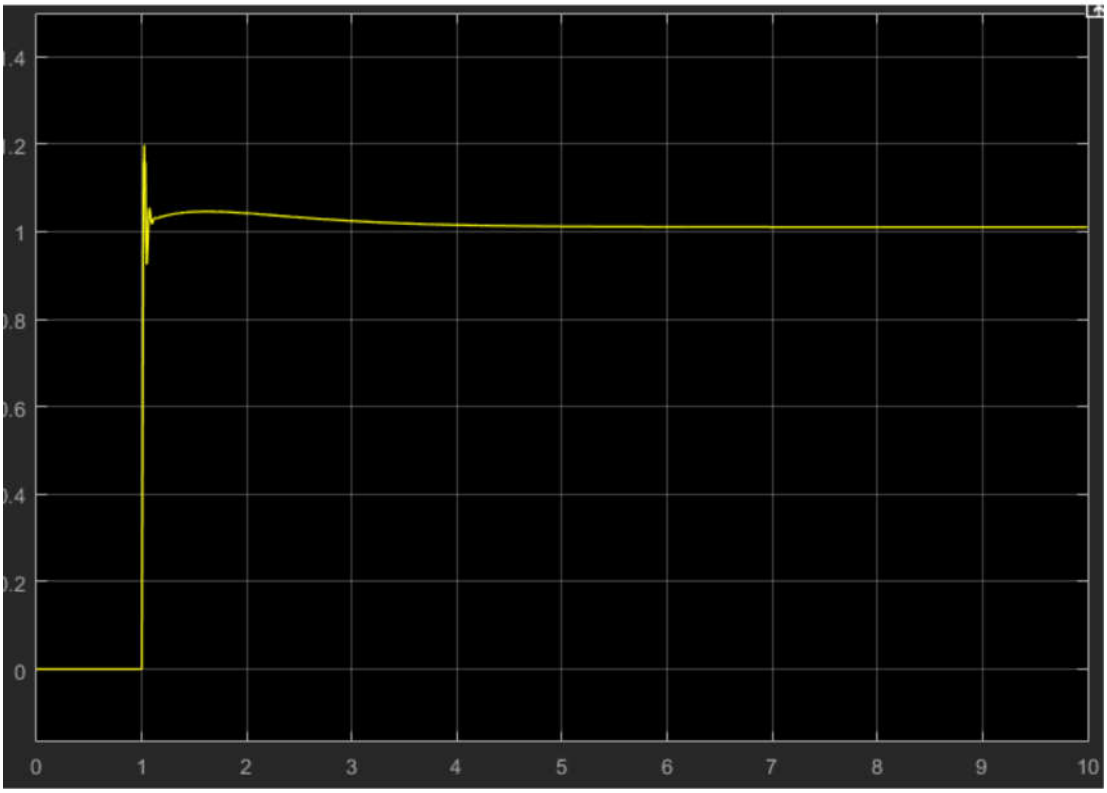


Figura 47: Gráfica de salida del sistema ante una estrada escalón

Tal y como se puede observar en la Figura 47, es un control rápido, dado que se estabiliza en 0,1 segundos, asumiendo un amortiguamiento del 20%, y apenas apreciar rebasamiento en estado estacionario.

Debido a que no existen diferencias entre ambos ejes X e Y, el otro lazo de control, se considera análogo. Por lo que se usarán los valores PID en ambos algoritmos de control.

7.4. PROGRAMACIÓN

El firmware se puede consultar en el ANEXO 3, si bien en este apartado se explica dichas líneas de código con detalle.

7.4.1 MÁQUINA DE ESTADOS GENERAL

El flujograma principal se efectúa mediante una maquina de estados finita. Refleja las 4 fases, explicadas en el apartado 6. Es necesario cargar las librerías MPU9250, BMP280 y servos.

```
1 #include "mbed.h"
2  #include "FSM.h"
3  #include ‹MPU9250›
4  #include ‹BMP280›
5  #include ‹servos›
6
7
8  Serial pc(USBTX,USBRX);
9
10      bool ignition=false;
11      bool nothrust=false;
12      bool descend=false;
13
14      int actualState=0;
15
16      void init_config(){
17       config_servos();
18       config_imu();
19       config_baro();
20      }
21      void tvc(){
22       calc_tvc();
23      }
24      void altitude(){
25       calc_baro();
26      }
27      void recover(){
28       eject();
29      }
```

Para programa una máquina de estados es necesario definir el número de los mismos y su nombre. Las condiciones de transición serán variables flags activadas según varios eventos en este caso los booleanos ignition, nothrust y descend, que reflejas las transiciones de la Figura 28.

Su ejecución se realiza a través de la función loop(), la cual de manera continua evalúa en que estado está, ejecutando las funciones asociadas a cada estado y cambiando de estado cuando se active el flag correspondiente.

```
30    State fase1 = State(init_config);
31    State fase2 = State(tvc);
32    State fase3 = State(altitude);
33    State fase4 = State(recover);
34
35    FSM rocketflow = FSM(init_config);
36
37    void loop(){
38
39      if(rocketflow.isInState(init_config)){
40       actualState=0;
41      }else if(rocketflow.isInState(tvc)){
42       actualState=1;
43      }else if(rocketflow.isInState(altitude)){
44       actualState=2;
45      }else if(rocketflow.isInState(recover)){
46       actualState=3;
47      }
48
49      switch (actualState) {
50        case 0:
51          if(ignition){rocketflow.transitionTo(tvc);}
52        break;
53        case 1:
54          if(nothrust){rocketflow.transitionTo(altitude);}
55        break;
56        case 2:
57          if(descend){rocketflow.transitionTo(recover);}
58        break;
59        case 3:
```

```
60              //Nothing
61            break;
62            }
63          rocketflow.update();
64      }
65
66
67      int main(){
68          pc.printf("INIT\r\n");
69          while(1){loop();}
70      }
```

La función main() suele usarse para inicializar configuraciones debido a que es una función que solo se ejecuta una vez. En este caso no tiene ningún uso, porque se dichas configuración se ejecutan en la fase1 de la máquina de estados finita.

7.4.2 CLASE MPU9250

En el estado de la fase2, entra en funcionamiento el algoritmo del MPU9250. Primero es necesario llamar a las librerías implicadas, que serían la del sensor IMU y la del PID. Hay que recordar que el sistema tiene dos ejes independientes así que crearemos dos funciones PID separadas.

```
1 #include "MPU9250.h"
2 #include "PID.h"
3 #include "servos.h"
4 #include "mbed.h"
5
6 static DigitalOut testLed(LED1);
7
8 volatile bool newData = false;
9 InterruptIn isrPin(PA_10);
10
11      Serial pc2(USBTX, USBRX);
12
13      //PIDX
14      double Input_X, Output_X;
15      double Setpoint_X= 0;
16      double Kp_X=40, Ki_X=20, Kd_X=2;
```

```
17      PID xPID(&Input_X, &Output_X, &Setpoint_X, Kp_X, Ki_X, Kd_X,
  P_ON_E, REVERSE);
18
19      //PIDY
20      double Input_Y, Output_Y;
21      double Setpoint_Y= 0;
22      double Kp_Y=2, Ki_Y=5, Kd_Y=1;
23      PID yPID(&Input_Y, &Output_Y, &Setpoint_Y, Kp_Y, Ki_Y, Kd_Y,
  P_ON_E, REVERSE);
24
25      //MPU9250
26      MPU9250 mpu9250(PB_4,PA_8,USBTX, USBRX);
```

Primero realizamos la lectura de los valores con la función mpu9250.ReadRawAccGyroMag() y posteriormente se realiza el cálculo explicado en el apartado 7.1 con la función mpu9250.YPR();. Acto seguido ejecutamos los PID, cuya salida se pasa como valor en microsegundos a los servos.

```
27      void mpuisr(){
28        newData=true;
29      }
30
31      void calc_tvc(){
32        if(newData){
33          newData = false;
34          mpu9250.ReadRawAccGyroMag();
35
36          mpu9250.YPR();
37          Input_X=(int)mpu9250.pitch;
38          Input_Y=(int)mpu9250.roll;
39          xPID.Compute();
40          yPID.Compute();
41          SX.pulsewidth_us((int)Output_X);
42          SY.pulsewidth_us((int)Output_Y);
43
44          pc2.printf("%d %d \n\r",(int)Input_X,(int)Output_X);
45
46          testLed=!testLed;
47        }
48      }
49
```

```
50    int config_imu(){
51
52      pc2.baud(9600);
53
54      mpu9250.start();
55      xPID.SetMode(AUTOMATIC);
56      xPID.SetOutputLimits(servoXMMM[0], servoXMMM[2]);
57      Output_X=servoXMMM[1];
58      yPID.SetMode(AUTOMATIC);
59      yPID.SetOutputLimits(servoYMMM[0], servoYMMM[2]);
60      Output_Y=servoYMMM[1];
61
62      isrPin.rise(&mpuisr);
63
64      }
65    }
```

Cabe recordar que este algoritmo para efectuar la lectura de la posición relativa en el espacio debe ser lo más determinista posible. Es decir, hay que asegurarse que su ejecución es constante y a la misma frecuencia. Es por ello que se realiza una interrupción que podemos observar en la línea 62.

La señal cuadrada de interrupción que genera el MPU9250 hará que el microprocesador ejecute la función calc_tvc() a cada flanco de subida de dicha señal.

7.4.3 CLASE BMP280

Tal y como vemos en la Figura 27, creamos la comunicación I2C designando los pines I2C_SDA e I2C_SCL.

```
1 #include "mbed.h"
2 #include "BMP280.h"
3
4 Serial pc(USBTX,USBRX);
5 Timer t;
6
7 I2C i2c(I2C_SDA, I2C_SCL);
```

Creamos el objeto BMP280 según la sintaxis de la línea 8. En la función config_baro(); se configura el muestreo que hemos definido en el apartado 7.2 según el datasheet del fabricante.

```
8 BMP280 baro(i2c);
9
10      void calc_altitude(){
11
12          pc.printf("%d:",t.read_ms());
13          pc.printf("%f,%f\r\n",baro.readAltitude(1013.25),
    baro.readPressure());
14          wait_ms(100);
15
16          //pc.putc(27);      // ESC command
17          // pc.printf("[2J");  //Clear Terminal in putty
18
19      }
20
21
22      int config_baro(){
23          pc.baud(9600);
24
25          if (!baro.start()) {
26              pc.printf("Not valid BMP280 sensor, check wiring!\r\n");
27          while (1);
28          }
29
30          baro.setSampling(BMP280::MODE_NORMAL,  /* Operating Mode. */
31                      BMP280::SAMPLING_X2,    /* Temp. oversampling */
32                      BMP280::SAMPLING_X16,   /* Pressure oversampling */
33                      BMP280::FILTER_X16,     /* Filtering. */
34                      BMP280::STANDBY_MS_500); /* Standby time. */
35
36
37          t.start();
38
39          }
40      }
```

7.4.4 CLASE SERVO

Con esta clase definimos los pines a los cuales se conectan las señales de ambos servos, así como los rangos de movimientos mínimo, medio y máximo para nuestro sistema.

```
1 #define ServoX_Output PB_3
2 #define ServoY_Output PB_5
3
4 PwmOut SX(ServoX_Output);
5 PwmOut SY(ServoY_Output);
6
7 int servoXMMM[3]= {1025,1325,1625}; //range 600
8 int servoYMMM[3]= {1175,1425,1675}; //range 500
9
10      int rangeServoX = servoXMMM[2]-servoXMMM[0];
11      int rangeServoY = servoYMMM[2]-servoYMMM[0];
```

De esta manera, 1325 para el servo X y 1425 para el servo Y son los valores de inicio del sistema, donde el portamotor está alineado con el eje longitudinal del modelo de cohete. Los valores máximos y mínimos son los límites físicos donde el portamotor golpea el cuerpo exterior del modelo. Estos valores han sido hallados una vez montado el sistema y probando dichos límites

```
12      void config_servos(){
13        SX.period_ms(20);
14        SY.period_ms(20);
15        SX.pulsewidth_us(servoXMMM[1]);
16        SY.pulsewidth_us(servoYMMM[1]);
17      }
```

8. DIAGRAMA DE GANTT

El siguiente esquema representa las diferentes tareas realizados a los largo de la elaboración del proyecto:

TAREA 1	**Objetivo**: Diseño general del proyecto, bloques funcionales **Resultado**: definición de los requerimientos y objetivos del proyecto	
	Subtarea 1.1	**Objetivo**: Investigación teórica sobre los requerimientos del proyecto **Resultado**: Documentación sobre diferentes sistemas para diseñar
TAREA 2	**Objetivo**: investigación de modelos y motores de cohetería recreativa **Resultado**: designación de modelos y motores aplicables a los requerimientos del proyecto	
	Subtarea 2.1	**Objetivo**: Estudio teórico sobre modelos y motores de cohetería recreativa **Resultado**: Redacción de parte de la memoria técnica.
TAREA 3	**Objetivo**: Diseño de los sistemas electrónicos que cumplan con los requerimientos. **Resultado**: Modelo de los sistemas electrónicos en función de los requerimientos elegidos.	
	Subtarea 3.1	**Objetivo**: Diseño del sistema TVC **Resultado**: Modelo del sistema del TVC

<table>
<tr><td></td><td></td><td><table>
<tr><td>Subtarea 3.1.1</td><td>Objetivo: Diseño funcional del accionamiento mecánico
Resultado: Sistema físico de accionamiento. Gimbal y accionamiento mediante servomotores.</td></tr>
<tr><td>Subtarea 3.1.2</td><td>Objetivo:Investigación, elección y prueba del sistema electrónico del TVC
Resultado: Elección de IMU y servomotores. Código de obtención del ángulo de inclinación.</td></tr>
<tr><td>Subtarea 3.1.3</td><td>Objetivo: Diseño de control del sistema TVC
Resultado: Diseño y sintonización de PID.</td></tr>
</table></td></tr>
<tr><td></td><td>Subtarea 3.2</td><td>Objetivo: Diseño, elección y prueba del sistema de detección de altitud
Resultado: Elección de barómetro. Código de obtención de altitud.</td></tr>
<tr><td>TAREA 4</td><td colspan="2">Objetivo: Programación integral de todos los sistemas
Resultado: Firmware del proyecto</td></tr>
<tr><td>TAREA 5</td><td colspan="2">Objetivo: Diseño de estación de pruebas estáticas.
Resultado: Estructura adecuada para la ejecución de pruebas.</td></tr>
<tr><td>TAREA 6</td><td colspan="2">Objetivo: Realización de documentación técnica.
Resultado: Documento técnico del proyecto.</td></tr>
</table>

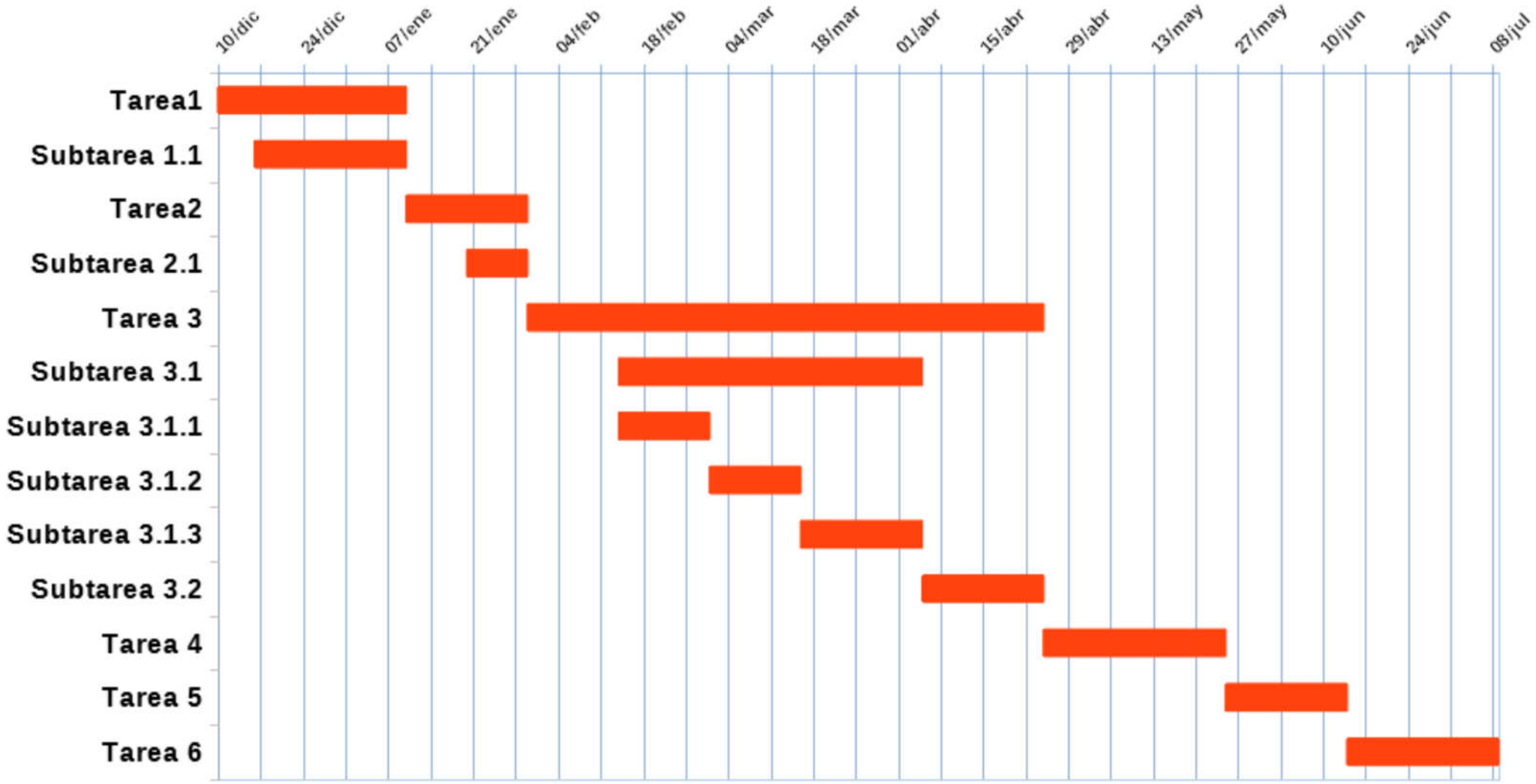


Figura 48: Diagrama de Gantt del desarrollo y elaboración del proyecto

9. PRESUPUESTO

A continuación se muestran los costes de elaboración, diseño e integración de algoritmos del proyecto.

Respecto a la mano de obra, se divide en ambas tareas realizar por un ingeniero junior. Son tareas de aprendizaje, investigación y diseño del sistema, como la fabricación propia del modelo.

Tabla 3: Costes asociados a la mano de obra

Tipo	Concepto	Horas (h)	Coste unitario (€/h)	Coste Total (€)
Ingeniero Junior	Investigación y diseño	300	10	3000
	Realización	30	10	300
Subtotal				3300€

En cuanto a las licencias del software usado, se puede ver uno de los puntos fuertes del proyecto. Se ha usado software open source, y software privativo con licencia de estudiante, por lo que los costes derivados de las licencias de software son nulos.

Tabla 4: Costes derivados de las licencias de software

Software	Tipo	Coste
OpenRocket	OpenSource	0
Fusion 360	Estudiante	0
MatLab2017	Estudiante	0
LibreOffice 6.2.4.2	OpenSource	0
ARMbed	OpenSource	0

Respecto al material, tenemos multitud de conceptos. Se separan en varias las siguientes tablas en función de la parte del proyecto correspondiente.

Tabla 5: Coste del material transversal a todo el proyecto

Material	Concepto	Cantidad	Coste unitario (€/ud)	Coste total (€)
Ordenador portátil	Portátil	1	600	Amortizado
Tektronix TBS1032b	Osciloscopio	1	510	Amortizado
Fluke 115	Multímetro	1	166	Amortizado
Subtotal				Amortizado

Tabla 6: Coste de materiales del Modelo de cohete

Material	Concepto	Cantidad	Coste unitario (€/ud)	Coste total (€)
Tubo cartón ϕ 80mm	Cuerpo modelo	1	6	6
Tubo cartón ϕ 40mm	Portamotor	1	2	2
Nucleo F446RE	Microcontrolador	1	15	15
JX PDI-6208MG	Servomotores	2	9,5	19
MPU9250	IMU	1	2	2
BMP280	Barómetro	1	1,65	1,65
Tuercas cuadradas	Tuercas	50	0,02	1
681ZZ	Rodamiento	10	0,17	1,7
623ZZ	Rodamiento	10	0,13	1,3
PLA	Filamento 3D	0,5	20	10
Subtotal				59,65€

Tabla 7: Coste de los materiales de la estación de pruebas

Material	Concepto	Cantidad	Coste unitario (€/ud)	Coste total (€)
Tubo presión ϕ 40mm 3m	Tubería	5	2,6	13
Codo ϕ40 90º	Unión tubería	10	0,46	4,6
Té ϕ40 45º	Unión tubería	16	0,7	11,2
Té ϕ40 90º	Unión tubería	22	1,19	26,19
Cemento	Saco cemento	1	2	2
PLA	Filamento 3D	0,25	20	5
608ZZ	Rodamientos	10	0,15	1,5
EDF ϕ35	Turbina	1	18	18
ESC 20A	Controlador turbina	1	12	12
Turnigy 2200mAh 11.1v	Batería 3S	1	10	10
Subtotal				103,49

La suma total de las tablas pormenorizadas anteriores da como resultado:

Tabla 8: Coste total del proyecto

Concepto	Coste(€)
Mano de Obra	3300
Licencias de software	0
Material transversal	0
Material Modelo Cohete	59,65
Material Estación de pruebas	103,49
TOTAL	3463,14

10. CONCLUSIONES

De acuerdo a los objetivos iniciales se concluye que se ha conseguido alcanzar de manera satisfactoria el diseño e implementación de un sistema de control de estabilización mediante empuje vectorial, así como detección de apogeo. A su vez, se ha logrado el objetivo de realizar una maqueta de construcción propia, incluida una estación de pruebas estáticas, tanto para motores de propelente sólido como para simulaciones.

A nivel personal, se ha observado la cantidad de problemas que supone la realización de un proyecto integral, pese a los conocimientos adquiridos a lo largo de la carrera, se ha tenido que ampliar, aprender, y controlar virtudes no estudiadas, sin las que este proyecto no hubiese podido llevarse a cabo.

Tal y como se ha descrito en el apartado 2.2 y 2.3, este proyecto supone una ventana abierta a la mejora de los sistemas presentes, así como el diseño e implementación de multitud de sistema electrónicos que podrían tener cabida en la cohetería amateur.

Cabría destacar, por ejemplo, dispositivos de recuperación activa, como la monitorización por GPS para realizar un seguimiento más preciso de la recuperación; así como transmisión de video en tiempo real con los datos de las mediciones realizadas tanto con los sensores de medición inercial, como con los sensores barométricos. Uno de los mayores potenciales que tiene este proyecto basado en la filosofía marker, es que al ser de código y hardware abierto, cualquier integrante de cualquier comunidad de cohetería, podría implementar este sistema, mejorarlo o ampliar el horizonte de la cohetería recreativa con los ejemplos arriba expuestos.

11. BIBLIOGRAFÍA

Árticulos:

[1] B. Yu and W. Shu, "A Novel Control Approach for a Thrust Vector System With an Electromechanical Actuator," in *IEEE Access* ,2007

[2] Aponte Rodríguez, J. y Amaya Hurtado, D. y Rubiano Fonseca, A. y Prada Jiménez, V. (2010). MODELADO, DISEÑO Y CONSTRUCCIÓN DE UN SISTEMA ACTIVO DE CONTROL DE ESTABILIDAD DE BAJO COSTO PARA COHETES EXPERIMENTALES TIPO AFICIONADO. *Ciencia e Ingeniería Neogranadina,* [en linea] 20(1), pp.77-96. Disponible en: http://www.redalyc.org/articulo.oa?id=91114807006

[3] "Inverted Pendulum: System Modeling" (http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling)

[4] Chérigo, C. y Rodríguez, H. (2017) Evaluación de algoritmos de fusión de datos para estimación de la orientación de vehículos aéreos no tripulados, *I+D Tecnológico*, 13(2), pp. 90-99. Disponible en: https://revistas.utp.ac.pa/index.php/id-tecnologico/article/view/1719

Páginas web:

[5] http://www.nakka-rocketry.net/pvcmot4.html
[6] http://www.rinconsolidario.org/meteorologia/webs/atmpre.htm
[7] https://aprendiendoarduino.wordpress.com/2016/11/13/bus-spi/
[8] https://teslabem.com/nivel-intermedio/fundamentos-del-protocolo-i2c-aprende/
[9] https://os.mbed.com/platforms/ST-Nucleo-F446RE/

Información general:

Stine, G. Harry: "Handbook of Model Rocketry". Wiley, 1994

http://www.nakka-rocketry.net/#Quick

https://www.grc.nasa.gov/www/k-12/rocket/rktstab.html

Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE SISTEMAS ELECTRÓNICOS PARA MODELOS DE COHETE USADOS EN COHETERÍA RECREATIVA

ANEXO 1- DATASHEETS DE MPU9250 Y BMP280

Alumno/Alumna: FEIJOO ALONSO, ANDER
Director/Directora (1): SAINZ DE MURIETA MANGADO, JOSEBA ANDONI

Curso: 2018-2019

Fecha: 19/07/2019

InvenSense Inc.
1745 Technology Drive, San Jose, CA 95110 U.S.A.
Tel: +1 (408) 988-7339 Fax: +1 (408) 988-8104
Website: www.invensense.com

Document Number: PS-MPU-9250A-01
Revision: 1.1
Release Date: 06/20/2016

MPU-9250
Product Specification
Revision 1.1

CONTENTS

1 Document Information

1.1 Revision History

Revision Date	Revision	Description
12/18/13	1.0	Initial Release
06/20/16	1.1	Updated Section 4

1.2 Purpose and Scope

This document provides a description, specifications, and design related information on the MPU-9250 MotionTracking device. The device is housed in a small 3x3x1mm QFN package.

Specifications are subject to change without notice. Final specifications will be updated based upon characterization of production silicon. For references to register map and descriptions of individual registers, please refer to the MPU-9250 Register Map and Register Descriptions document.

1.3 Product Overview

MPU-9250 is a multi-chip module (MCM) consisting of two dies integrated into a single QFN package. One die houses the 3-Axis gyroscope and the 3-Axis accelerometer. The other die houses the AK8963 3-Axis magnetometer from Asahi Kasei Microdevices Corporation. Hence, the MPU-9250 is a 9-axis MotionTracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor™ (DMP) all in a small 3x3x1mm package available as a pin-compatible upgrade from the MPU-6515. With its dedicated I²C sensor bus, the MPU-9250 directly provides complete 9-axis MotionFusion™ output. The MPU-9250 MotionTracking device, with its 9-axis integration, on-chip MotionFusion™, and run-time calibration firmware, enables manufacturers to eliminate the costly and complex selection, qualification, and system level integration of discrete devices, guaranteeing optimal motion performance for consumers. MPU-9250 is also designed to interface with multiple non-inertial digital sensors, such as pressure sensors, on its auxiliary I²C port.

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 ±2*g*, ±4*g*, ±8*g*, and ±16*g*, and a magnetometer full-scale range of ±4800µT.

Other industry-leading features include programmable digital filters, a precision clock with 1% drift from -40°C to 85°C, an embedded temperature sensor, and programmable interrupts. 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.

By leveraging its patented and volume-proven CMOS-MEMS fabrication platform, which integrates MEMS wafers with companion CMOS electronics through wafer-level bonding, InvenSense has driven the package size down to a footprint and thickness of 3x3x1mm, to provide a very small yet high performance low cost package. The device provides high robustness by supporting 10,000*g* shock reliability.

1.4 Applications

- Location based services, points of interest, and dead reckoning
- Handset and portable gaming
- Motion-based game controllers
- 3D remote controls for Internet connected DTVs and set top boxes, 3D mice
- Wearable sensors for health, fitness and sports

2 Features

2.1 Gyroscope Features

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

2.2 Accelerometer Features

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 ±2*g*, ±4*g*, ±8*g* and ±16*g* 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

2.3 Magnetometer Features

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

2.4 Additional Features

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

2.5 MotionProcessing

- Internal Digital Motion Processing™ (DMP™) engine supports advanced MotionProcessing and low power functions such as gesture recognition using programmable interrupts
- Low-power pedometer functionality allows the host processor to sleep while the DMP maintains the step count.

3 Electrical Characteristics

3.1 Gyroscope Specifications

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

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 1 Gyroscope Specifications

3.2 Accelerometer Specifications

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

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
Accelerometer Startup Time	From Sleep mode		20		ms
	From Cold Start, 1ms V_{DD} ramp		30		ms
Output Data Rate	Low power (duty-cycled)	0.24		500	Hz
	Duty-cycled, over temp		±15		%
	Low noise (active)	4		4000	Hz

Table 2 Accelerometer Specifications

3.3 Magnetometer Specifications

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

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

3.4 Electrical Specifications

3.4.1 D.C. Electrical Characteristics

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

PARAMETER	CONDITIONS	MIN	TYP	MAX	Units	Notes
SUPPLY VOLTAGES						
VDD		2.4	2.5	3.6	V	
VDDIO		1.71	1.8	VDD	V	
SUPPLY CURRENTS						
Normal Mode	9-axis (no DMP), 1 kHz gyro ODR, 4 kHz accel ODR, 8 Hz mag. repetition rate		3.7		mA	
	6-axis (accel + gyro, no DMP), 1 kHz gyro ODR, 4 kHz accel ODR		3.4		mA	
	3-axis Gyroscope only (no DMP), 1 kHz ODR		3.2		mA	
	6-axis (accel + magnetometer, no DMP), 4 kHz accel ODR, mag. repetition rate = 8 Hz		730		µA	
	3-Axis Accelerometer, 4kHz ODR (no DMP)		450		µA	
	3-axis Magnetometer only (no DMP), 8 Hz repetition rate		280		µA	
Accelerometer Low Power Mode (DMP, Gyroscope, Magnetometer disabled)	0.98 Hz update rate		8.4		µA	1
	31.25 Hz update rate		19.8		µA	1
Full Chip Idle Mode Supply Current			8		µA	
TEMPERATURE RANGE						
Specified Temperature Range	Performance parameters are not applicable beyond Specified Temperature Range	-40		+85	°C	

Table 3 D.C. Electrical Characteristics

Notes:

1. Accelerometer Low Power Mode supports the following output data rates (ODRs): 0.24, 0.49, 0.98, 1.95, 3.91, 7.81, 15.63, 31.25, 62.50, 125, 250, 500Hz. Supply current for any update rate can be calculated as:

   Supply Current in µA = Sleep Current + Update Rate * 0.376

3.4.2 A.C. Electrical Characteristics

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

Parameter	Conditions	MIN	TYP	MAX	Units
Supply Ramp Time	Monotonic ramp. Ramp rate is 10% to 90% of the final value	0.1		100	ms
Operating Range	Ambient	-40		85	°C
Sensitivity	Untrimmed		333.87		LSB/°C
Room Temp Offset	21°C		0		LSB
Supply Ramp Time (T_{RAMP})	Valid power-on RESET	0.01	20	100	ms
Start-up time for register read/write	From power-up		11	100	ms
I²C ADDRESS	AD0 = 0 AD0 = 1		1101000 1101001		
V_{IH}, High Level Input Voltage		0.7*VDDIO			V
V_{IL}, Low Level Input Voltage				0.3*VDDIO	V
C_I, Input Capacitance			< 10		pF
V_{OH}, High Level Output Voltage	R_{LOAD}=1MΩ;	0.9*VDDIO			V
V_{OL1}, LOW-Level Output Voltage	R_{LOAD}=1MΩ;			0.1*VDDIO	V
$V_{OL.INT1}$, INT Low-Level Output Voltage	OPEN=1, 0.3mA sink Current			0.1	V
Output Leakage Current	OPEN=1		100		nA
t_{INT}, INT Pulse Width	LATCH_INT_EN=0		50		µs
V_{IL}, LOW Level Input Voltage		-0.5V		0.3*VDDIO	V
V_{IH}, HIGH-Level Input Voltage		0.7*VDDIO		VDDIO + 0.5V	V
V_{hys}, Hysteresis			0.1*VDDIO		V
V_{OL}, LOW-Level Output Voltage	3mA sink current	0		0.4	V
I_{OL}, LOW-Level Output Current	V_{OL}=0.4V V_{OL}=0.6V		3 6		mA mA
Output Leakage Current			100		nA
t_{of}, Output Fall Time from V_{IHmax} to V_{ILmax}	C_b bus capacitance in pf	20+0.1C_b		250	ns
V_{IL}, LOW-Level Input Voltage		-0.5V		0.3*VDDIO	V
V_{IH}, HIGH-Level Input Voltage		0.7* VDDIO		VDDIO + 0.5V	V
V_{hys}, Hysteresis			0.1* VDDIO		V
V_{OL1}, LOW-Level Output Voltage	VDDIO > 2V; 1mA sink current	0		0.4	V
V_{OL3}, LOW-Level Output Voltage	VDDIO < 2V; 1mA sink current	0		0.2* VDDIO	V
I_{OL}, LOW-Level Output Current	V_{OL} = 0.4V V_{OL} = 0.6V		3 6		mA mA
Output Leakage Current			100		nA
t_{of}, Output Fall Time from V_{IHmax} to V_{ILmax}	C_b bus capacitance in pF	20+0.1C_b		250	ns
Sample Rate	Fchoice=0,1,2 SMPLRT_DIV=0		32		kHz
	Fchoice=3; DLPFCFG=0 or 7 SMPLRT_DIV=0		8		kHz
	Fchoice=3; DLPFCFG=1,2,3,4,5,6; SMPLRT_DIV=0		1		kHz
Clock Frequency Initial Tolerance	CLK_SEL=0, 6; 25°C	-2		+2	%

	CLK_SEL=1,2,3,4,5; 25°C	-1		+1	%
Frequency Variation over Temperature	CLK_SEL=0,6	-10		+10	%
	CLK_SEL=1,2,3,4,5		±1		%

Table 4 A.C. Electrical Characteristics

3.4.3 Other Electrical Specifications

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

PARAMETER	CONDITIONS	MIN	TYP	MAX	Units
SPI Operating Frequency, All Registers Read/Write	Low Speed Characterization		100 ±10%		kHz
	High Speed Characterization		1 ±10%		MHz
SPI Operating Frequency, Sensor and Interrupt Registers Read Only			20 ±10%		MHz
I²C Operating Frequency	All registers, Fast-mode			400	kHz
	All registers, Standard-mode			100	kHz

Table 5 Other Electrical Specifications

3.5 I2C Timing Characterization

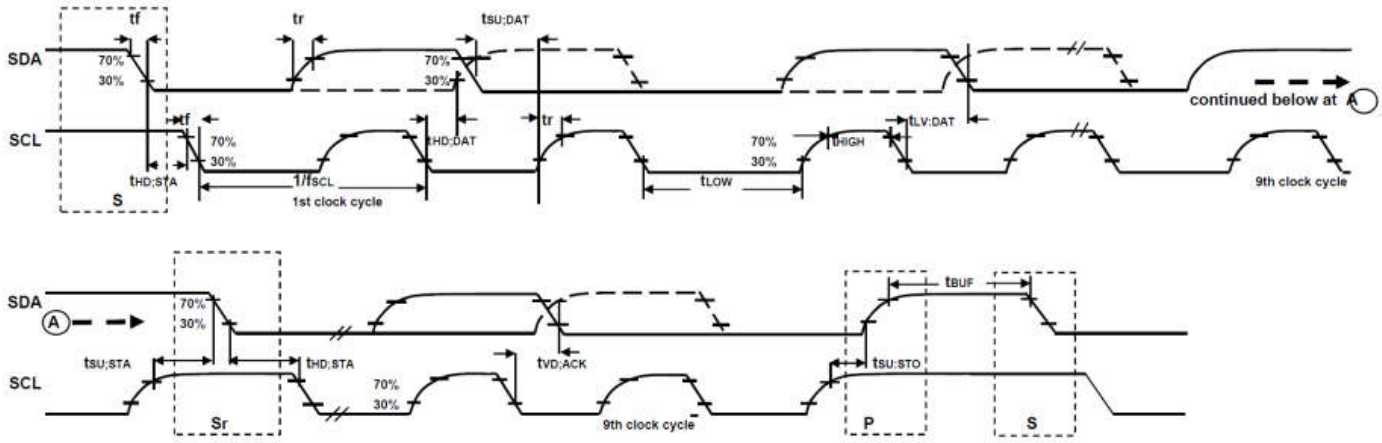
Typical Operating Circuit of section 4.2, VDD = 2.4V to 3.6V, VDDIO = 1.71 to VDD, T_A=25°C, unless otherwise noted.

Parameters	Conditions	Min	Typical	Max	Units	Notes
I²C TIMING	**I²C FAST-MODE**					
f_{SCL}, SCL Clock Frequency				400	kHz	
$t_{HD.STA}$, (Repeated) START Condition Hold Time		0.6			µs	
t_{LOW}, SCL Low Period		1.3			µs	
t_{HIGH}, SCL High Period		0.6			µs	
$t_{SU.STA}$, Repeated START Condition Setup Time		0.6			µs	
$t_{HD.DAT}$, SDA Data Hold Time		0			µs	
$t_{SU.DAT}$, SDA Data Setup Time		100			ns	
t_r, SDA and SCL Rise Time	C_b bus cap. from 10 to 400pF	20+0.1C_b		300	ns	
t_f, SDA and SCL Fall Time	C_b bus cap. from 10 to 400pF	20+0.1C_b		300	ns	
$t_{SU.STO}$, STOP Condition Setup Time		0.6			µs	
t_{BUF}, Bus Free Time Between STOP and START Condition		1.3			µs	
C_b, Capacitive Load for each Bus Line			< 400		pF	
$t_{VD.DAT}$, Data Valid Time				0.9	µs	
$t_{VD.ACK}$, Data Valid Acknowledge Time				0.9	µs	

Table 6 I²C Timing Characteristics

Notes:

- Timing Characteristics apply to both Primary and Auxiliary I2C Bus
- Based on characterization of 5 parts over temperature and voltage as mounted on evaluation board or in sockets



I²C Bus Timing Diagram

3.6 SPI Timing Characterization

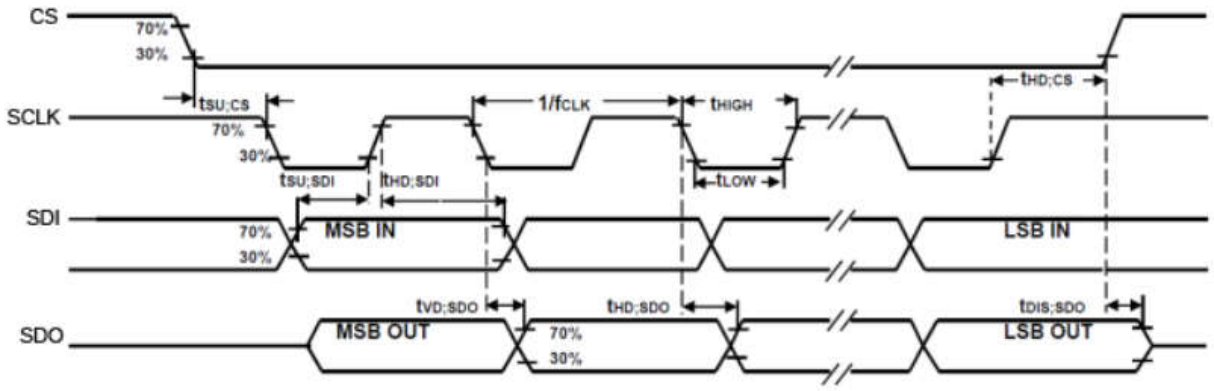
Typical Operating Circuit of section 4.2, VDD = 2.4V to 3.6V, VDDIO = 1.71V to VDD, T_A=25°C, unless otherwise noted.

Parameters	Conditions	Min	Typical	Max	Units	Notes
SPI TIMING						
f_{SCLK}, SCLK Clock Frequency				1	MHz	
t_{LOW}, SCLK Low Period		400			ns	
t_{HIGH}, SCLK High Period		400			ns	
$t_{SU.CS}$, CS Setup Time		8			ns	
$t_{HD.CS}$, CS Hold Time		500			ns	
$t_{SU.SDI}$, SDI Setup Time		11			ns	
$t_{HD.SDI}$, SDI Hold Time		7			ns	
$t_{VD.SDO}$, SDO Valid Time	C_{load} = 20pF			100	ns	
$t_{HD.SDO}$, SDO Hold Time	C_{load} = 20pF	4			ns	
$t_{DIS.SDO}$, SDO Output Disable Time				50	ns	

Table 7 SPI Timing Characteristics

Notes:

1. Based on characterization of 5 parts over temperature and voltage as mounted on evaluation board or in sockets



SPI Bus Timing Diagram

3.6.1 fSCLK = 20MHz

Parameters	Conditions	Min	Typical	Max	Units
SPI TIMING					
f_{SCLK}, SCLK Clock Frequency		0.9		20	MHz
t_{LOW}, SCLK Low Period		-		-	ns
t_{HIGH}, SCLK High Period		-		-	ns
$t_{SU.CS}$, CS Setup Time		1			ns
$t_{HD.CS}$, CS Hold Time		1			ns

$t_{SU.SDI}$, SDI Setup Time		0			ns
$t_{HD.SDI}$, SDI Hold Time		1			ns
$t_{VD.SDO}$, SDO Valid Time	C_{load} = 20pF		25		ns
$t_{DIS.SDO}$, SDO Output Disable Time				25	ns

Table 8 fCLK = 20MHz

Note:

1. Based on characterization of 5 parts over temperature and voltage as mounted on evaluation board or in sockets

3.7 Absolute Maximum Ratings

Stress above those listed as "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these conditions is not implied. Exposure to the absolute maximum ratings conditions for extended periods may affect device reliability.

Specification	Symbol	Conditions	MIN	MAX	Units
Supply Voltage	V_{DD}		-0.5	4.0	V
	V_{DDIO}		-0.5	4.0	V
Acceleration		Any axis, unpowered, 0.2ms duration		10,000	*g*
Temperature		Operating	-40	105	°C
		Storage	-40	125	°C
ESD Tolerance		HBM	2		KV
		MM	250		V

4 Applications Information

4.1 Pin Out and Signal Description

Pin Number	Pin Name	Pin Description
1	RESV	Reserved. Connect to VDDIO.
7	AUX_CL	I²C Master serial clock, for connecting to external sensors
8	VDDIO	Digital I/O supply voltage
9	AD0 / SDO	I²C Slave Address LSB (AD0); SPI serial data output (SDO)
10	REGOUT	Regulator filter capacitor connection
11	FSYNC	Frame synchronization digital input. Connect to GND if unused.
12	INT	Interrupt digital output (totem pole or open-drain)
13	VDD	Power supply voltage and Digital I/O supply voltage
18	GND	Power supply ground
19	RESV	Reserved. Do not connect.
20	RESV	Reserved. Connect to GND.
21	AUX_DA	I²C master serial data, for connecting to external sensors
22	nCS	Chip select (SPI mode only)
23	SCL / SCLK	I²C serial clock (SCL); SPI serial clock (SCLK)
24	SDA / SDI	I²C serial data (SDA); SPI serial data input (SDI)
2 – 6, 14 - 17	NC	Not internally connected. May be used for PCB trace routing.

Table 9 Signal Descriptions

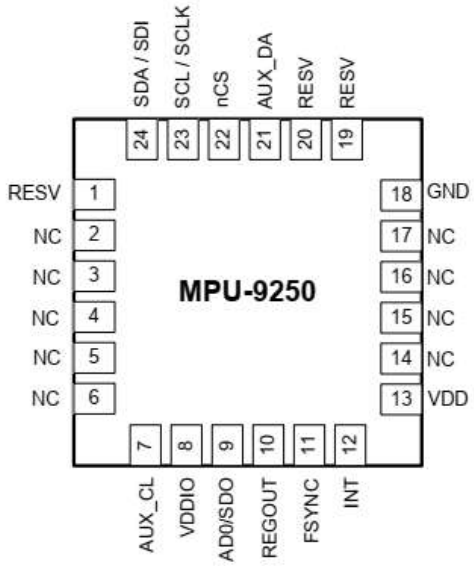


Figure 1 Pin Out Diagram for MPU-9250 3.0x3.0x1.0mm QFN

4.2 Typical Operating Circuit

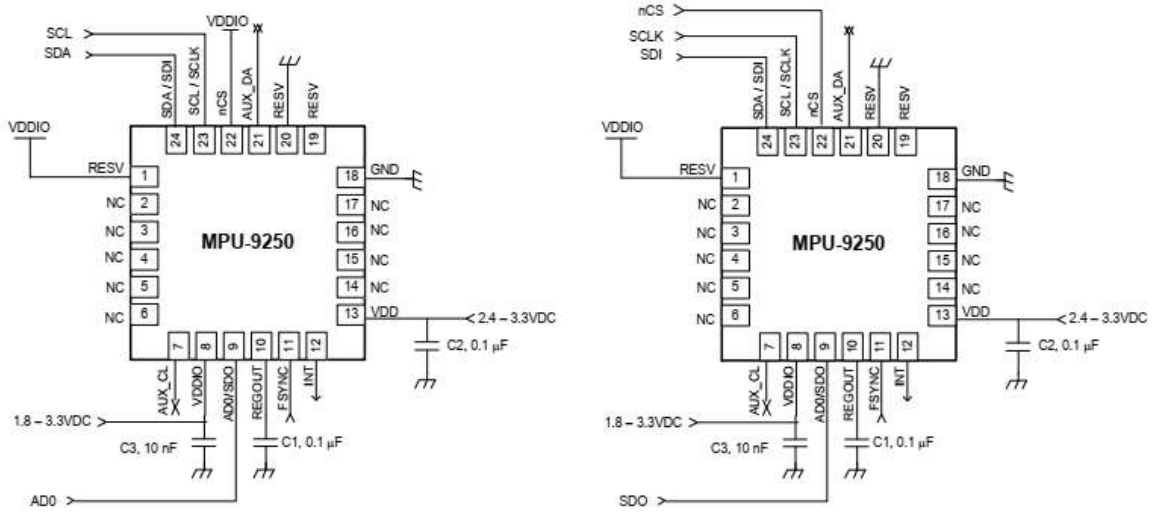


Figure 2 MPU-9250 QFN Application Schematic: (a) I2C operation, (b) SPI operation

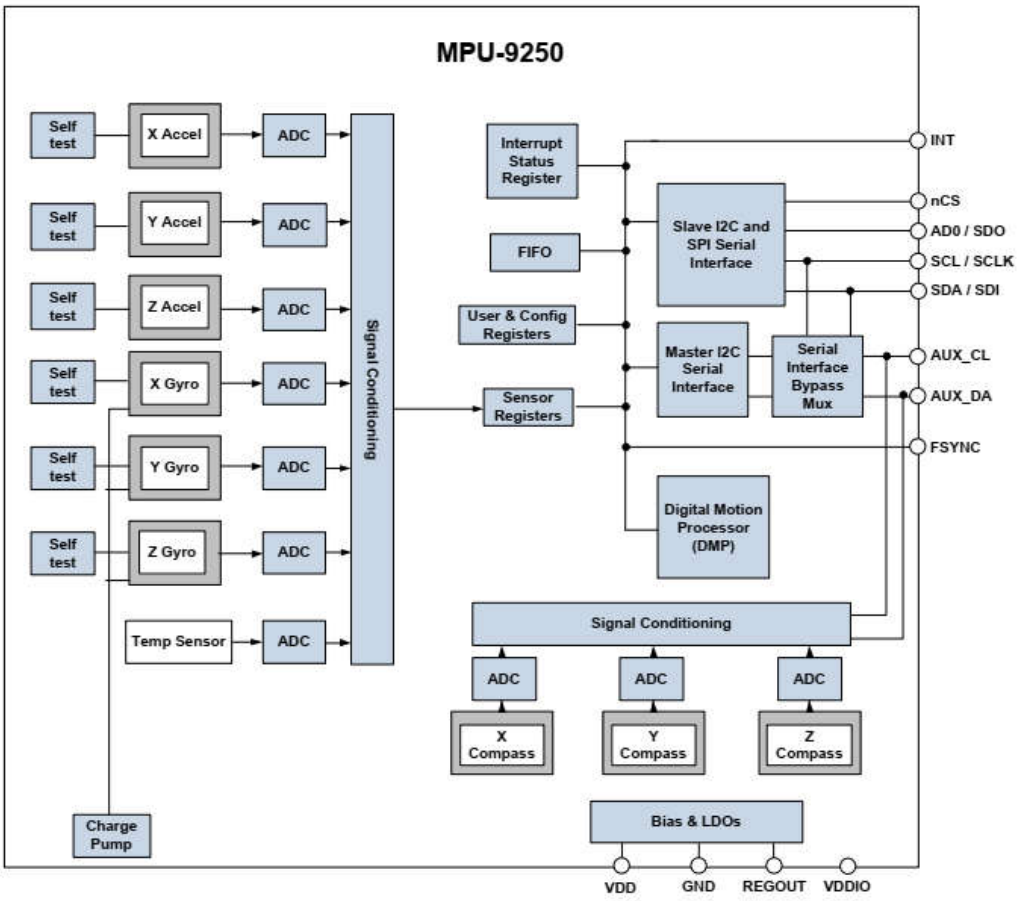
Note that the INT pin should be connected to a GPIO pin on the system processor that is capable of waking the system processor from suspend mode.

4.3 Bill of Materials for External Components

Component	Label	Specification	Quantity
Regulator Filter Capacitor	C1	Ceramic, X7R, 0.1µF ±10%, 2V	1
VDD Bypass Capacitor	C2	Ceramic, X7R, 0.1µF ±10%, 4V	1
VDDIO Bypass Capacitor	C3	Ceramic, X7R, 10nF ±10%, 4V	1

Table 10 Bill of Materials

4.4 Block Diagram



4.5 Overview

The MPU-9250 is comprised of the following key blocks and functions:

- Three-axis MEMS rate gyroscope sensor with 16-bit ADCs and signal conditioning
- Three-axis MEMS accelerometer sensor with 16-bit ADCs and signal conditioning
- Three-axis MEMS magnetometer sensor with 16-bit ADCs and signal conditioning
- Digital Motion Processor (DMP) engine
- Primary I^2C and SPI serial communications interfaces
- Auxiliary I^2C serial interface for 3rd party sensors
- Clocking
- Sensor Data Registers
- FIFO
- Interrupts
- Digital-Output Temperature Sensor
- Gyroscope, Accelerometer and Magnetometer Self-test
- Bias and LDO
- Charge Pump

4.6 Three-Axis MEMS Gyroscope with 16-bit ADCs and Signal Conditioning

The MPU-9250 consists of three independent vibratory MEMS rate gyroscopes, which detect rotation about the X-, Y-, and Z- Axes. When the gyros are rotated about any of the sense axes, the Coriolis Effect causes a vibration that is detected by a capacitive pickoff. The resulting signal is amplified, demodulated, and filtered to produce a voltage that is proportional to the angular rate. This voltage is digitized using individual on-chip 16-bit Analog-to-Digital Converters (ADCs) to sample each axis. The full-scale range of the gyro sensors may be digitally programmed to ±250, ±500, ±1000, or ±2000 degrees per second (dps). The ADC sample rate is programmable from 8,000 samples per second, down to 3.9 samples per second, and user-selectable low-pass filters enable a wide range of cut-off frequencies.

4.7 Three-Axis MEMS Accelerometer with 16-bit ADCs and Signal Conditioning

The MPU-9250's 3-Axis accelerometer uses separate proof masses for each axis. Acceleration along a particular axis induces displacement on the corresponding proof mass, and capacitive sensors detect the displacement differentially. The MPU-9250's architecture reduces the accelerometers' susceptibility to fabrication variations as well as to thermal drift. When the device is placed on a flat surface, it will measure 0*g* on the X- and Y-axes and +1*g* on the Z-axis. The accelerometers' scale factor is calibrated at the factory and is nominally independent of supply voltage. Each sensor has a dedicated sigma-delta ADC for providing digital outputs. The full scale range of the digital output can be adjusted to ±2*g*, ±4*g*, ±8*g*, or ±16*g*.

4.8 Three-Axis MEMS Magnetometer with 16-bit ADCs and Signal Conditioning

The 3-axis magnetometer uses highly sensitive Hall sensor technology. The magnetometer portion of the IC incorporates magnetic sensors for detecting terrestrial magnetism in the X-, Y-, and Z- Axes, a sensor driving circuit, a signal amplifier chain, and an arithmetic circuit for processing the signal from each sensor. Each ADC has a 16-bit resolution and a full scale range of ±4800 µT.

4.9 Digital Motion Processor

The embedded Digital Motion Processor (DMP) is located within the MPU-9250 and offloads computation of motion processing algorithms from the host processor. The DMP acquires data from accelerometers,

gyroscopes, magnetometers and additional 3rd party sensors, and processes the data. The resulting data can be read from the DMP's registers, or can be buffered in a FIFO. The DMP has access to one of the MPU's external pins, which can be used for generating interrupts. This pin (pin 12) should be connected to a pin on the host processor that can wake the host from suspend mode.

The purpose of the DMP is to offload both timing requirements and processing power from the host processor. Typically, motion processing algorithms should be run at a high rate, often around 200Hz, in order to provide accurate results with low latency. This is required even if the application updates at a much lower rate; for example, a low power user interface may update as slowly as 5Hz, but the motion processing should still run at 200Hz. The DMP can be used as a tool in order to minimize power, simplify timing, simplify the software architecture, and save valuable MIPS on the host processor for use in the application.

4.10 Primary I2C and SPI Serial Communications Interfaces

The MPU-9250 communicates to a system processor using either a SPI or an I²C serial interface. The MPU-9250 always acts as a slave when communicating to the system processor. The LSB of the of the I²C slave address is set by pin 9 (AD0).

4.11 Auxiliary I2C Serial Interface

The MPU-9250 has an auxiliary I²C bus for communicating to off-chip sensors. This bus has two operating modes:

- I²C Master Mode: The MPU-9250 acts as a master to any external sensors connected to the auxiliary I²C bus
- Pass-Through Mode: The MPU-9250 directly connects the primary and auxiliary I²C buses together, allowing the system processor to directly communicate with any external sensors.
- Note: AUX_DA and AUX_CL should be left unconnected if the Auxiliary I²C mode is not used.

Auxiliary I²C Bus Modes of Operation:

- I²C Master Mode: Allows the MPU-9250 to directly access the data registers of external digital sensors, such as a magnetometer. In this mode, the MPU-9250 directly obtains data from auxiliary sensors without intervention from the system applications processor.

  For example, In I²C Master mode, the MPU-9250 can be configured to perform burst reads, returning the following data from a magnetometer:

  - X magnetometer data (2 bytes)
  - Y magnetometer data (2 bytes)
  - Z magnetometer data (2 bytes)

  The I²C Master can be configured to read up to 24 bytes from up to 4 auxiliary sensors. A fifth sensor can be configured to work single byte read/write mode.

- Pass-Through Mode: Allows an external system processor to act as master and directly communicate to the external sensors connected to the auxiliary I²C bus pins (AUX_DA and AUX_CL). In this mode, the auxiliary I²C bus control logic (3rd party sensor interface block) of the MPU-9250 is disabled, and the auxiliary I²C pins AUX_DA and AUX_CL are connected to the main I²C bus through analog switches internally.

Pass-Through mode is useful for configuring the external sensors, or for keeping the MPU-9250 in a low-power mode when only the external sensors are used. In this mode, the system processor can still access MPU-9250 data through the I²C interface.
Pass-Through mode is also used to access the AK8963 magnetometer directly from the host. In this configuration the slave address for the AK8963 is 0X0C or 12 decimal.

Auxiliary I²C Bus IO Logic Levels

For MPU-9250, the logic level of the auxiliary I²C bus is VDDIO. For further information regarding the MPU-9250 logic levels, please refer to Section 10.2.

4.12 Self-Test

Please refer to the register map document for more details on self-test.

Self-test allows for the testing of the mechanical and electrical portions of the sensors. The self-test for each measurement axis can be activated by means of the gyroscope and accelerometer self-test registers (registers 13 to 16).

When the self-test is activated, the electronics cause the sensors to be actuated and produce an output signal. The output signal is used to observe the self-test response.

The self-test response is defined as follows:

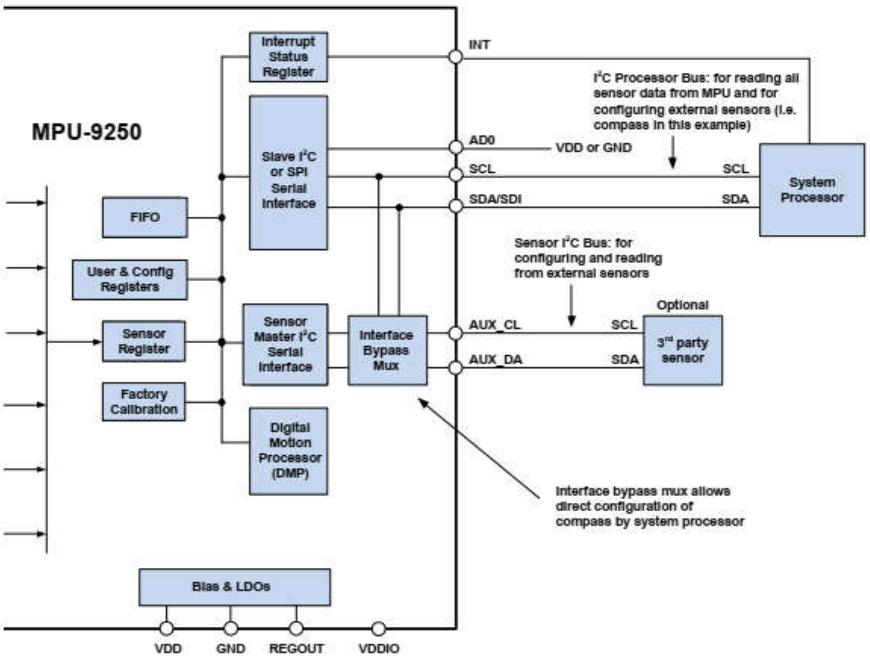
Self-test response = Sensor output with self-test enabled – Sensor output without self-test enabled

When the value of the self-test response is within the appropriate limits, the part has passed self-test. When the self-test response exceeds the appropriate values, the part is deemed to have failed self-test. It is recommended to use InvenSense MotionApps software for executing self-test. Further details, including the self-test limits are included in the MPU-9250 Self-Test applications note available from InvenSense.

4.13 MPU-9250 Solution Using I2C Interface

In the figure below, the system processor is an I²C master to the MPU-9250. In addition, the MPU-9250 is an I²C master to the optional external 3rd party sensor. The MPU-9250 has limited capabilities as an I²C Master, and depends on the system processor to manage the initial configuration of any auxiliary sensors. The MPU-9250 has an interface bypass multiplexer, which connects the system processor I²C bus (SDA and SCL) directly to the auxiliary sensor I²C bus (AUX_DA and AUX_CL).

Once the auxiliary sensors have been configured by the system processor, the interface bypass multiplexer should be disabled so that the MPU-9250 auxiliary I²C master can take control of the sensor I²C bus and gather data from the auxiliary sensors. The INT pin should be connected to a GPIO on the system processor that can wake the system from suspend mode.



4.14 MPU-9250 Solution Using SPI Interface

In the figure below, the system processor is a SPI master to the MPU-9250. The CS, SDO, SCLK, and SDI signals are used for SPI communications. Because these SPI pins are shared with the I²C slave pins, the system processor cannot access the auxiliary I²C bus through the interface bypass multiplexer, which connects the processor I²C interface pins to the sensor I²C interface pins.

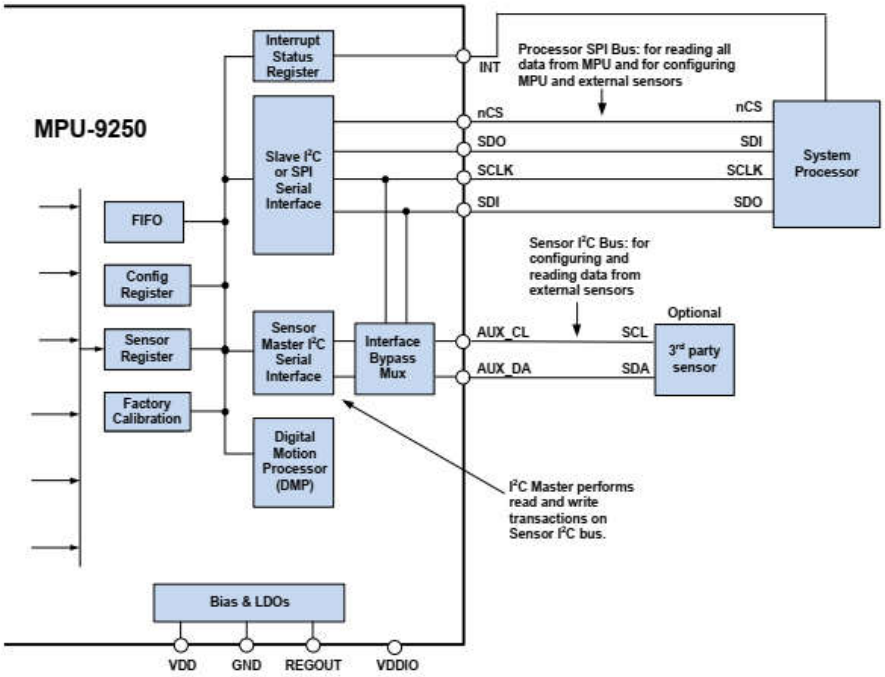
Since the MPU-9250 has limited capabilities as an I²C Master, and depends on the system processor to manage the initial configuration of any auxiliary sensors, another method must be used for programming the sensors on the auxiliary sensor I²C bus (AUX_DA and AUX_CL).

When using SPI communications between the MPU-9250 and the system processor, configuration of devices on the auxiliary I²C sensor bus can be achieved by using I²C Slaves 0-4 to perform read and write transactions on any device and register on the auxiliary I²C bus. The I²C Slave 4 interface can be used to perform only single byte read and write transactions.

Once the external sensors have been configured, the MPU-9250 can perform single or multi-byte reads using the sensor I²C bus. The read results from the Slave 0-3 controllers can be written to the FIFO buffer as well as to the external sensor registers.

The INT pin should be connected to a GPIO on the system processor capable of waking the processor from suspend

For further information regarding the control of the MPU-9250's auxiliary I²C interface, please refer to the MPU-9250 Register Map and Register Descriptions document.



4.15 Clocking

The MPU-9250 has a flexible clocking scheme, allowing a variety of internal clock sources to be used for the internal synchronous circuitry. This synchronous circuitry includes the signal conditioning and ADCs, the DMP, and various control circuits and registers. An on-chip PLL provides flexibility in the allowable inputs for generating this clock.

Allowable internal sources for generating the internal clock are:

- An internal relaxation oscillator
- Any of the X, Y, or Z gyros (MEMS oscillators with a variation of ±1% over temperature)

Selection of the source for generating the internal synchronous clock depends on the requirements for power consumption and clock accuracy. These requirements will most likely vary by mode of operation. For example, in one mode, where the biggest concern is power consumption, the user may wish to operate the Digital Motion Processor of the MPU-9250 to process accelerometer data, while keeping the gyros off. In this case, the internal relaxation oscillator is a good clock choice. However, in another mode, where the gyros are active, selecting the gyros as the clock source provides for a more accurate clock source.

Clock accuracy is important, since timing errors directly affect the distance and angle calculations performed by the Digital Motion Processor (and by extension, by any processor).

There are also start-up conditions to consider. When the MPU-9250 first starts up, the device uses its internal clock until programmed to operate from another source. This allows the user, for example, to wait for the MEMS oscillators to stabilize before they are selected as the clock source.

4.16 Sensor Data Registers

The sensor data registers contain the latest gyroscope, accelerometer, magnetometer, auxiliary sensor, and temperature measurement data. They are read-only registers, and are accessed via the serial interface. Data from these registers may be read anytime.

4.17 FIFO

The MPU-9250 contains a 512-byte FIFO register that is accessible via the Serial Interface. The FIFO configuration register determines which data is written into the FIFO. Possible choices include gyro data, accelerometer data, temperature readings, auxiliary sensor readings, and FSYNC input. A FIFO counter keeps track of how many bytes of valid data are contained in the FIFO. The FIFO register supports burst reads. The interrupt function may be used to determine when new data is available.

For further information regarding the FIFO, please refer to the MPU-9250 Register Map and Register Descriptions document.

4.18 Interrupts

Interrupt functionality is configured via the Interrupt Configuration register. Items that are configurable include the INT pin configuration, the interrupt latching and clearing method, and triggers for the interrupt. Items that can trigger an interrupt are (1) Clock generator locked to new reference oscillator (used when switching clock sources); (2) new data is available to be read (from the FIFO and Data registers); (3) accelerometer event interrupts; and (4) the MPU-9250 did not receive an acknowledge from an auxiliary sensor on the secondary I^2C bus. The interrupt status can be read from the Interrupt Status register.

The INT pin should be connected to a pin on the host processor capable of waking that processor from suspend.

For further information regarding interrupts, please refer to the MPU-9250 Register Map and Register Descriptions document.

4.19 Digital-Output Temperature Sensor

An on-chip temperature sensor and ADC are used to measure the MPU-9250 die temperature. The readings from the ADC can be read from the FIFO or the Sensor Data registers.

4.20 Bias and LDO

The bias and LDO section generates the internal supply and the reference voltages and currents required by the MPU-9250. Its two inputs are an unregulated VDD and a VDDIO logic reference supply voltage. The LDO output is bypassed by a capacitor at REGOUT. For further details on the capacitor, please refer to the Bill of Materials for External Components.

4.21 Charge Pump

An on-chip charge pump generates the high voltage required for the MEMS oscillators.

4.22 Standard Power Mode

The following table lists the user-accessible power modes for MPU-9250.

Mode	Name	Gyro	Accel	Magnetometer	DMP
1	Sleep Mode	Off	Off	Off	Off
2	Standby Mode	Drive On	Off	Off	Off
3	Low-Power Accelerometer Mode	Off	Duty-Cycled	Off	On or Off
4	Low-Noise Accelerometer Mode	Off	On	Off	On or Off
5	Gyroscope Mode	On	Off	Off	On or Off
6	Magnetometer Mode	Off	Off	On	On or Off
7	Accel + Gyro Mode	On	On	Off	On or Off
8	Accel + Magnetometer Mode	Off	On	On	On or Off
9	9-Axis Mode	On	On	On	On or Off

Notes:

1. Power consumption for individual modes can be found in Electrical Characteristics section.

4.23 Power Sequencing Requirements and Power on Reset

During power up and in normal operation, VDDIO must not exceed VDD. During power up, VDD and VDDIO must be monotonic ramps. As stated in Table 4, the minimum VDD rise time is 0.1ms and the maximum rise time is 100 ms. Valid gyroscope data is available 35 ms (typical) after VDD has risen to its final voltage from a cold start and valid accelerometer data is available 30 ms (typical) after VDD has risen to its final voltage assuming a 1ms VDD ramp from cold start. Magnetometer data is valid 7.3ms (typical) after VDD has risen to its final voltage value from a cold start.

5 Advanced Hardware Features

The MPU-9250 includes advanced hardware features that can be enabled and disabled through simple hardware register settings. The advanced hardware features are not initially enabled after device power up. These features must be individually enabled and configured. These advanced hardware features enable the following motion-based functions without using an external microprocessor:

- Low Power Quaternion (3-Axis Gyro & 6-Axis Gyro + Accel)
- Android Orientation (A low-power implementation of Android's screen rotation algorithm)
- Tap (detects the tap gesture)
- Pedometer
- Significant Motion Detection

To ensure significant motion detection can operate properly, the INT pin should be connected to a GPIO pin on the host processor that can wake that processor from suspend mode.

Note: Android Orientation is compliant to the Ice Cream Sandwich definition of the function.

For further details on advanced hardware features please refer to the MPU-9250 Register Map.

6 Programmable Interrupts

The MPU-9250 has a programmable interrupt system which can generate an interrupt signal on the INT pin. Status flags indicate the source of an interrupt. Interrupt sources may be enabled and disabled individually.

Table of Interrupt Sources

Interrupt Name	Module
Motion Detection	Motion
FIFO Overflow	FIFO
Data Ready	Sensor Registers
I²C Master errors: Lost Arbitration, NACKs	I²C Master
I²C Slave 4	I²C Master

For information regarding the interrupt enable/disable registers and flag registers, please refer to the MPU-9250 Register Map and Register Descriptions document. Some interrupt sources are explained below.

6.1 Wake-on-Motion Interrupt

The MPU-9250 provides motion detection capability. A qualifying motion sample is one where the high passed sample from any axis has an absolute value exceeding a user-programmable threshold. The following flowchart explains how to configure the Wake-on-Motion Interrupt. For further details on individual registers, please refer to the MPU-9250 Registers Map and Registers Description document.

In order to properly enable motion interrupts, the INT pin should be connected to a GPIO on the system processor that is capable of waking up the system processor.

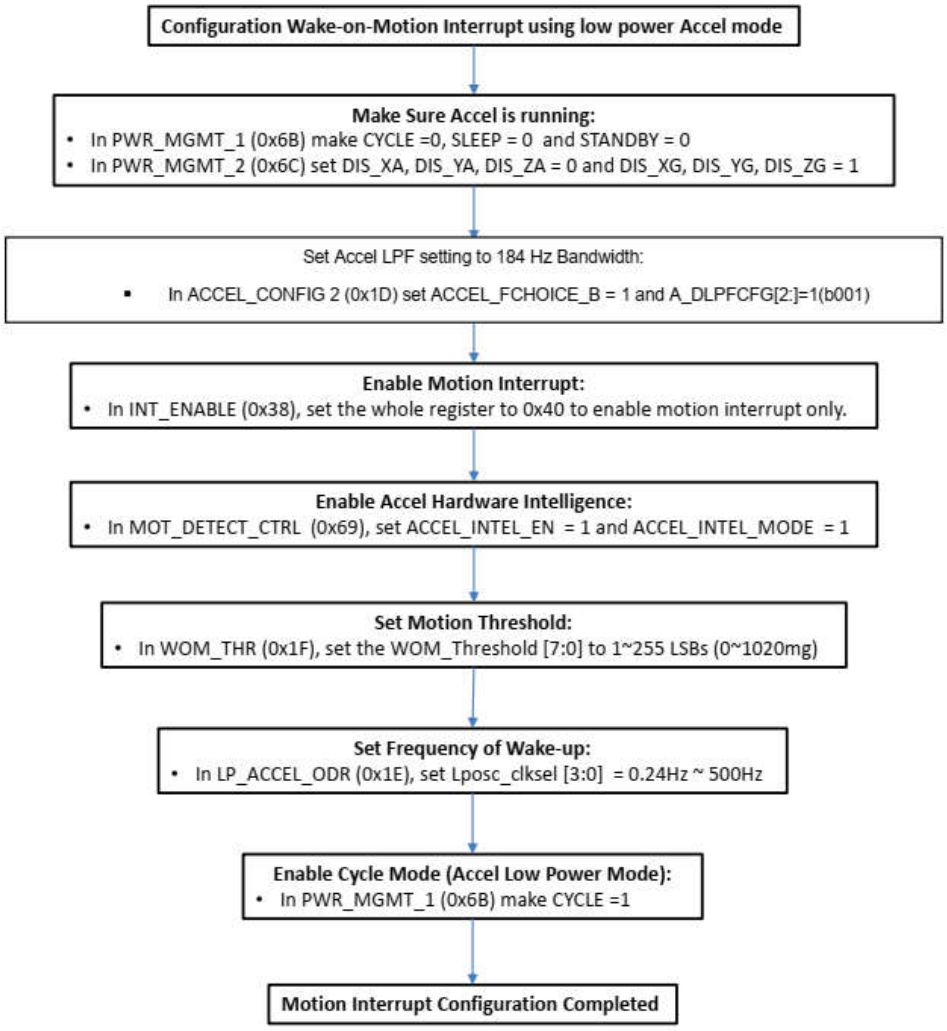


Figure 3. Wake-on-Motion Interrupt Configuration

7 Digital Interface

7.1 I2C and SPI Serial Interfaces

The internal registers and memory of the MPU-9250 can be accessed using either I²C at 400 kHz or SPI at 1MHz. SPI operates in four-wire mode.

Serial Interface

Pin Number	Pin Name	Pin Description
8	VDDIO	Digital I/O supply voltage.
9	AD0 / SDO	I²C Slave Address LSB (AD0); SPI serial data output (SDO)
23	SCL / SCLK	I²C serial clock (SCL); SPI serial clock (SCLK)
24	SDA / SDI	I²C serial data (SDA); SPI serial data input (SDI)

Note:

To prevent switching into I²C mode when using SPI, the I²C interface should be disabled by setting the *I2C_IF_DIS* configuration bit. Setting this bit should be performed immediately after waiting for the time specified by the "Start-Up Time for Register Read/Write" in Section 6.3.

For further information regarding the *I2C_IF_DIS* bit, please refer to the MPU-9250 Register Map and Register Descriptions document.

7.2 I2C Interface

I²C is a two-wire interface comprised of the signals serial data (SDA) and serial clock (SCL). In general, the lines are open-drain and bi-directional. In a generalized I²C interface implementation, attached devices can be a master or a slave. The master device puts the slave address on the bus, and the slave device with the matching address acknowledges the master.

The MPU-9250 always operates as a slave device when communicating to the system processor, which thus acts as the master. SDA and SCL lines typically need pull-up resistors to VDD. The maximum bus speed is 400 kHz.

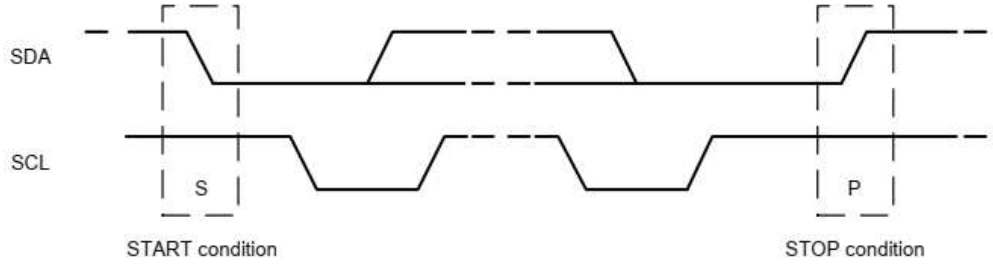
The slave address of the MPU-9250 is b110100X which is 7 bits long. The LSB bit of the 7 bit address is determined by the logic level on pin AD0. This allows two MPU-9250s to be connected to the same I²C bus. When used in this configuration, the address of the one of the devices should be b1101000 (pin AD0 is logic low) and the address of the other should be b1101001 (pin AD0 is logic high).

7.3 I2C Communications Protocol

START (S) and STOP (P) Conditions

Communication on the I²C bus starts when the master puts the START condition (S) on the bus, which is defined as a HIGH-to-LOW transition of the SDA line while SCL line is HIGH (see figure below). The bus is considered to be busy until the master puts a STOP condition (P) on the bus, which is defined as a LOW to HIGH transition on the SDA line while SCL is HIGH (see figure below).

Additionally, the bus remains busy if a repeated START (Sr) is generated instead of a STOP condition.

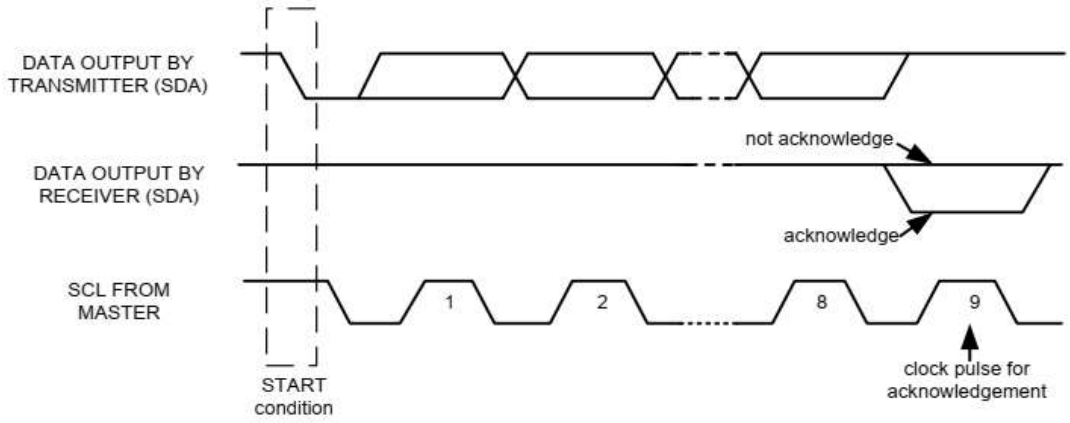


START and STOP Conditions

Data Format / Acknowledge

I²C data bytes are defined to be 8-bits long. There is no restriction to the number of bytes transmitted per data transfer. Each byte transferred must be followed by an acknowledge (ACK) signal. The clock for the acknowledge signal is generated by the master, while the receiver generates the actual acknowledge signal by pulling down SDA and holding it low during the HIGH portion of the acknowledge clock pulse.

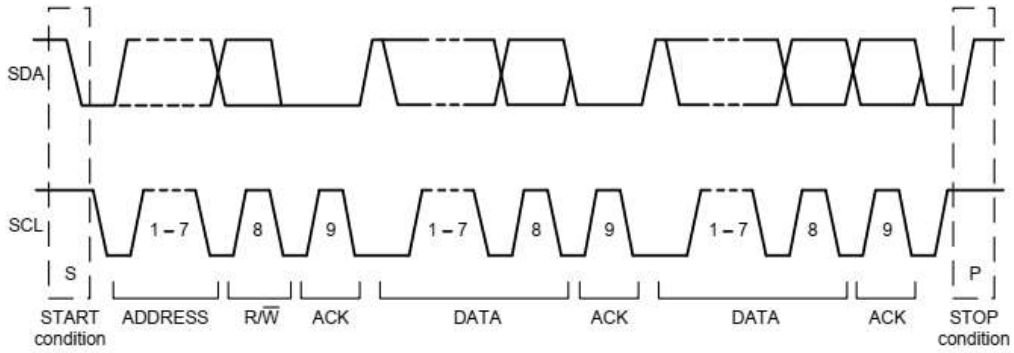
If a slave is busy and cannot transmit or receive another byte of data until some other task has been performed, it can hold SCL LOW, thus forcing the master into a wait state. Normal data transfer resumes when the slave is ready, and releases the clock line (refer to the following figure).



Acknowledge on the I²C Bus

Communications

After beginning communications with the START condition (S), the master sends a 7-bit slave address followed by an 8th bit, the read/write bit. The read/write bit indicates whether the master is receiving data from or is writing to the slave device. Then, the master releases the SDA line and waits for the acknowledge signal (ACK) from the slave device. Each byte transferred must be followed by an acknowledge bit. To acknowledge, the slave device pulls the SDA line LOW and keeps it LOW for the high period of the SCL line. Data transmission is always terminated by the master with a STOP condition (P), thus freeing the communications line. However, the master can generate a repeated START condition (Sr), and address another slave without first generating a STOP condition (P). A LOW to HIGH transition on the SDA line while SCL is HIGH defines the stop condition. All SDA changes should take place when SCL is low, with the exception of start and stop conditions.



Complete I²C Data Transfer

To write the internal MPU-9250 registers, the master transmits the start condition (S), followed by the I²C address and the write bit (0). At the 9th clock cycle (when the clock is high), the MPU-9250 acknowledges the transfer. Then the master puts the register address (RA) on the bus. After the MPU-9250 acknowledges the reception of the register address, the master puts the register data onto the bus. This is followed by the ACK signal, and data transfer may be concluded by the stop condition (P). To write multiple bytes after the last ACK signal, the master can continue outputting data rather than transmitting a stop signal. In this case, the MPU-9250 automatically increments the register address and loads the data to the appropriate register. The following figures show single and two-byte write sequences.

Single-Byte Write Sequence

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

Burst Write Sequence

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

To read the internal MPU-9250 registers, the master sends a start condition, followed by the I²C address and a write bit, and then the register address that is going to be read. Upon receiving the ACK signal from the MPU-9250, the master transmits a start signal followed by the slave address and read bit. As a result, the MPU-9250 sends an ACK signal and the data. The communication ends with a not acknowledge (NACK) signal and a stop bit from master. The NACK condition is defined such that the SDA line remains high at the 9th clock cycle. The following figures show single and two-byte read sequences.

Single-Byte Read Sequence

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

7.4 I2C Terms

Signal	Description
S	Start Condition: SDA goes from high to low while SCL is high
AD	Slave I²C address
W	Write bit (0)
R	Read bit (1)
ACK	Acknowledge: SDA line is low while the SCL line is high at the 9th clock cycle
NACK	Not-Acknowledge: SDA line stays high at the 9th clock cycle
RA	MPU-9250 internal register address
DATA	Transmit or received data
P	Stop condition: SDA going from low to high while SCL is high

7.5 SPI Interface

SPI is a 4-wire synchronous serial interface that uses two control lines and two data lines. The MPU-9250 always operates as a Slave device during standard Master-Slave SPI operation.

With respect to the Master, the Serial Clock output (SCLK), the Serial Data Output (SDO) and the Serial Data Input (SDI) are shared among the Slave devices. Each SPI slave device requires its own Chip Select (CS) line from the master.

CS goes low (active) at the start of transmission and goes back high (inactive) at the end. Only one CS line is active at a time, ensuring that only one slave is selected at any given time. The CS lines of the non-selected slave devices are held high, causing their SDO lines to remain in a high-impedance (high-z) state so that they do not interfere with any active devices.

SPI Operational Features

1. Data is delivered MSB first and LSB last
2. Data is latched on the rising edge of SCLK
3. Data should be transitioned on the falling edge of SCLK
4. The maximum frequency of SCLK is 1MHz
5. SPI read and write operations are completed in 16 or more clock cycles (two or more bytes). The first byte contains the SPI Address, and the following byte(s) contain(s) the SPI data. The first bit of the first byte contains the Read/Write bit and indicates the Read (1) or Write (0) operation. The following 7 bits contain the Register Address. In cases of multiple-byte Read/Writes, data is two or more bytes:

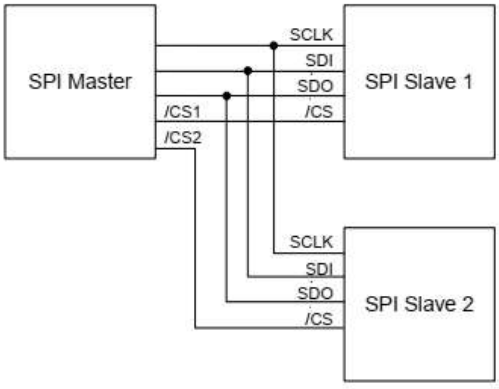
SPI Address format

MSB							**LSB**
R/W	A6	A5	A4	A3	A2	A1	A0

SPI Data format

MSB							**LSB**
D7	D6	D5	D4	D3	D2	D1	D0

6. Supports Single or Burst Read/Writes.



Typical SPI Master / Slave Configuration

8 Serial Interface Considerations

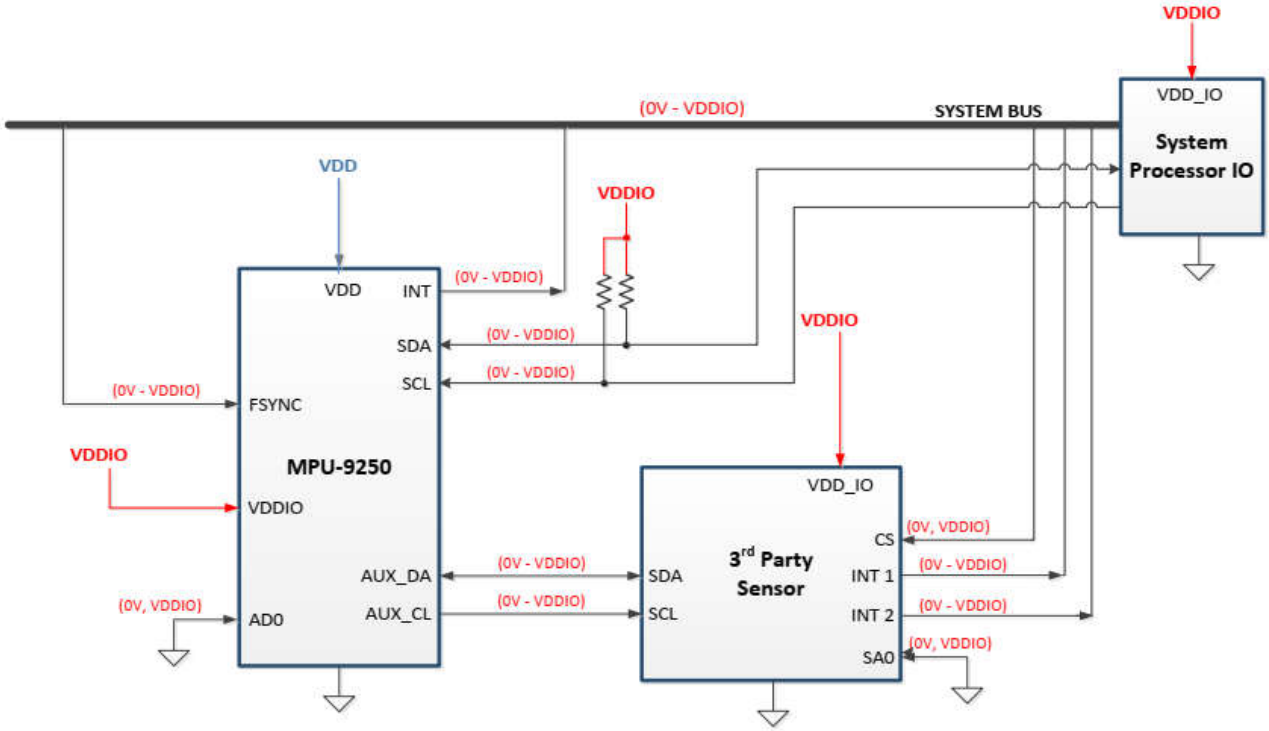
8.1 MPU-9250 Supported Interfaces

The MPU-9250 supports I²C communications on both its primary (microprocessor) serial interface and its auxiliary interface.

The MPU-9250's I/O logic levels are set to be VDDIO.

The figure below depicts a sample circuit of MPU-9250 with a third party sensor attached to the auxiliary I²C bus. It shows the relevant logic levels and voltage connections.

Note: Actual configuration will depend on the auxiliary sensors used.



I/O Levels and Connections

9 Assembly

This section provides general guidelines for assembling InvenSense Micro Electro-Mechanical Systems (MEMS) devices packaged in quad flat no-lead package (QFN) surface mount integrated circuits.

9.1 Orientation of Axes

The diagram below shows the orientation of the axes of sensitivity and the polarity of rotation. Note the pin 1 identifier (•) in the figure.

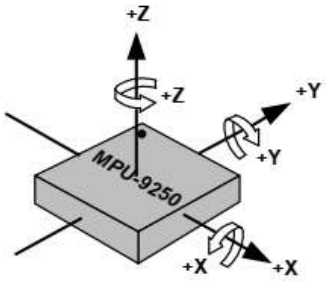


Figure 4. Orientation of Axes of Sensitivity and Polarity of Rotation for Accelerometer and Gyroscope

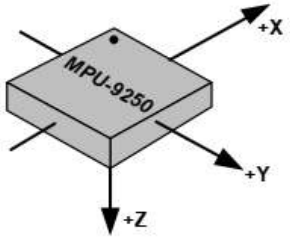
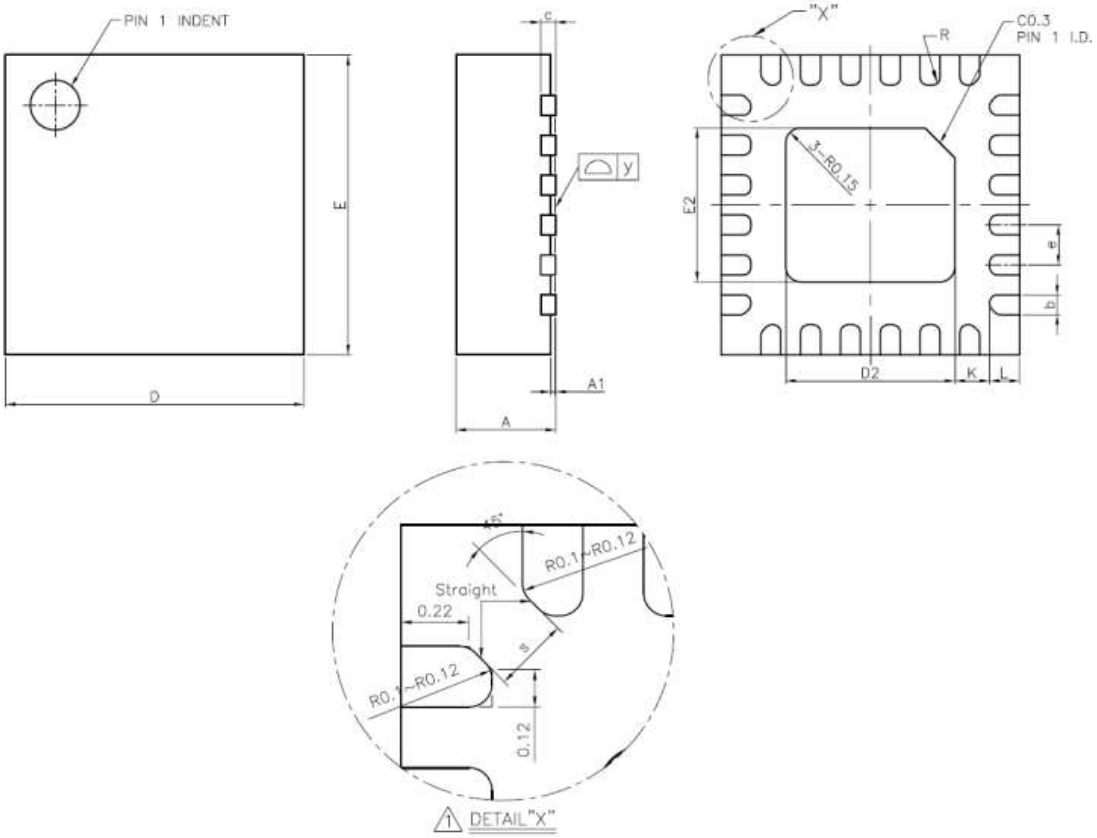


Figure 5. Orientation of Axes of Sensitivity for Compass

9.2 Package Dimensions

24 Lead QFN (3x3x1) mm NiPdAu Lead-frame finish



SYMBOLS	DESCRIPTION	DIMENSIONS IN MILLIMETERS		
		MIN	NOM	MAX
A	Package thickness	0.95	1.00	1.05
A1	Lead finger (pad) seating height	0.00	0.02	0.05
b	Lead finger (pad) width	0.15	0.20	0.25
c	Lead frame (pad) height	---	0.15 REF	---
D	Package width	2.90	3.00	3.10
D2	Exposed pad width	1.65	1.70	1.75
E	Package length	2.90	3.00	3.10
E2	Exposed pad length	1.49	1.54	1.59
e	Lead finger-finger (pad-pad) pitch	---	0.40	---
f (e-b)	Lead-lead (Pad-Pad) space	0.15	0.20	0.25
K	Lead (pad) to Exposed Pad Space	---	0.35 REF	---
L	Lead (pad) length	0.25	0.30	0.35
R	Lead (pad) corner radius	0.075	REF	---
s	Corner lead (pad) outer radius to corner lead outer radius	---	0.25 REF	---
y		0.00	---	0.075

10 Part Number Package Marking

The part number package marking for MPU-9250 devices is summarized below:

Part Number	Part Number Package Marking
MPU-9250	MP92

11 Reliability

11.1 Qualification Test Policy

InvenSense's products complete a Qualification Test Plan before being released to production. The Qualification Test Plan for the MPU-9250 followed the JEDEC JESD 47I Standard, "Stress-Test-Driven Qualification of Integrated Circuits," with the individual tests described below.

11.2 Qualification Test Plan

Accelerated Life Tests

TEST	Method/Condition	Lot Quantity	Sample / Lot	Acc / Reject Criteria
(HTOL/LFR) High Temperature Operating Life	JEDEC JESD22-A108D Dynamic, 3.63V biased, Tj>125°C [read-points: 168, 500, 1000 hours]	3	77	(0/1)
(HAST) Highly Accelerated Stress Test (1)	JEDEC JESD22-A118A Condition A, 130°C, 85%RH, 33.3 psia., unbiased [read-point: 96 hours]	3	77	(0/1)
(HTS) High Temperature Storage Life	JEDEC JESD22-A103D Condition A, 125°C Non-Bias Bake [read-points: 168, 500, 1000 hours]	3	77	(0/1)

Device Component Level Tests

TEST	Method/Condition	Lot Quantity	Sample / Lot	Acc / Reject Criteria
(ESD-HBM) ESD-Human Body Model	JEDEC JS-001-2012 (2KV)	1	3	(0/1)
(ESD-MM) ESD-Machine Model	JEDEC JESD22-A115C (250V)	1	3	(0/1)
(ESD-CDM) ESD-Charged Device Model	JEDEC JESD22-C101E (500V)	1	3	(0/1)
(LU) Latch Up	JEDEC JESD-78D Class II (2), 125°C; ±100mA 1.5X Vdd Over-voltage	1	6	(0/1)
(MS) Mechanical Shock	JEDEC JESD22-B104C, Mil-Std-883, Method 2002.5 Cond. E, 10,000*g's*, 0.2ms, ±X, Y, Z – 6 directions, 5 times/direction	3	5	(0/1)
(VIB) Vibration	JEDEC JESD22-B103B Variable Frequency (random), Cond. B, 5-500Hz, X, Y, Z – 4 times/direction	1	5	(0/1)
(TC) Temperature Cycling (1)	JEDEC JESD22-A104D Condition G [-40°C to +125°C], Soak Mode 2 [5'] [read-Point: 1000 cycles]	3	77	(0/1)

(1) Tests are preceded by MSL3 Preconditioning in accordance with JEDEC JESD22-A113F

12 Reference

Please refer to "InvenSense MEMS Handling Application Note (AN-IVS-0002A-00)" for the following information:

- Manufacturing Recommendations
  - Assembly Guidelines and Recommendations
  - PCB Design Guidelines and Recommendations
  - MEMS Handling Instructions
  - ESD Considerations
  - Reflow Specification
  - Storage Specifications
  - Package Marking Specification
  - Tape & Reel Specification
  - Reel & Pizza Box Label
  - Packaging
  - Representative Shipping Carton Label
- Compliance
  - Environmental Compliance
  - DRC Compliance
  - Compliance Declaration Disclaimer

InvenSense

Data sheet

BMP280
Digital Pressure Sensor

Bosch Sensortec

BOSCH
Invented for life

BMP280: Data sheet

Document revision	1.14
Document release date	May 5th, 2015
Document number	BST-BMP280-DS001-11
Technical reference code(s)	0273 300 416
Notes	Data in this document are subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance.

BMP280

DIGITAL PRESSURE SENSOR

Key parameters

• Pressure range	300 ... 1100 hPa (equiv. to +9000...-500 m above/below sea level)
• Package	8-pin LGA metal-lid Footprint : 2.0 × 2.5 mm², height: 0.95 mm
• Relative accuracy (950 ... 1050hPa @25°C)	±0.12 hPa, equiv. to ±1 m
• Absolute accuracy (950 ...1050 hPa, 0 ...+40 °C)	typ. ±1 hPa
• Temperature coefficient offset (25 ... 40°C @900hPa)	1.5 Pa/K, equiv. to 12.6 cm/K
• Digital interfaces	I²C (up to 3.4 MHz) SPI (3 and 4 wire, up to 10 MHz)
• Current consumption	2.7µA @ 1 Hz sampling rate
• Temperature range	-40 ... +85 °C
• RoHS compliant, halogen-free	
• MSL 1	

Typical applications

- Enhancement of GPS navigation
  (e.g. time-to-first-fix improvement, dead-reckoning, slope detection)
- Indoor navigation (floor detection, elevator detection)
- Outdoor navigation, leisure and sports applications
- Weather forecast
- Health care applications (e.g. spirometry)
- Vertical velocity indication (e.g. rise/sink speed)

Target devices

- Handsets such as mobile phones, tablet PCs, GPS devices
- Navigation systems
- Portable health care devices
- Home weather stations
- Flying toys
- Watches

General Description

Robert Bosch is the world market leader for pressure sensors in automotive and consumer applications. Bosch's proprietary APSM (Advanced Porous Silicon Membrane) MEMS manufacturing process is fully CMOS compatible and allows a hermetic sealing of the cavity in an all silicon process. The BMP280 is based on Bosch's proven Piezo-resistive pressure sensor technology featuring high EMC robustness, high accuracy and linearity and long term stability.

The BMP280 is an absolute barometric pressure sensor especially designed for mobile applications. The sensor module is housed in an extremely compact 8-pin metal-lid LGA package with a footprint of only 2.0 × 2.5 mm^2 and 0.95 mm package height. Its small dimensions and its low power consumption of 2.7 µA @1Hz allow the implementation in battery driven devices such as mobile phones, GPS modules or watches.

As the successor to the widely adopted BMP180, the BMP280 delivers high performance in all applications that require precise pressure measurement. The BMP280 operates at lower noise, supports new filter modes and an SPI interface within a footprint 63% smaller than the BMP180.

The emerging applications of in-door navigation, health care as well as GPS refinement require a high relative accuracy and a low TCO at the same time. BMP180 and BMP280 are perfectly suitable for applications like floor detection since both sensors feature excellent relative accuracy is ±0.12 hPa, which is equivalent to ±1 m difference in altitude. The very low offset temperature coefficient (TCO) of 1.5 Pa/K translates to a temperature drift of only 12.6 cm/K. Please contact your regional Bosch Sensortec partner for more information about software packages enhancing the calculation of the altitude given by the BMP280 pressure reading.

Table 1: Comparison between BMP180 and BMP280

Parameter	BMP180	BMP280
Footprint	3.6 × 3.8 mm	2.0 × 2.5 mm
Minimum V_{DD}	1.80 V	1.71 V
Minimum V_{DDIO}	1.62 V	1.20 V
Current consumption @3 Pa RMS noise	12 µA	2.7 µA
RMS Noise	3 Pa	1.3 Pa
Pressure resolution	1 Pa	0.16 Pa
Temperature resolution	0.1°C	0.01°C
Interfaces	I²C	I²C & SPI (3 and 4 wire, mode '00' and '11')
Measurement modes	Only P or T, forced	P&T, forced or periodic
Measurement rate	up to 120 Hz	up to 157 Hz
Filter options	None	Five bandwidths

Index of Contents

1. Specification

If not stated otherwise,

- All values are valid over the full voltage range
- All minimum/maximum values are given for the full accuracy temperature range
- Minimum/maximum values of drifts, offsets and temperature coefficients are ±3σ values over lifetime
- Typical values of currents and state machine timings are determined at 25 °C
- Minimum/maximum values of currents are determined using corner lots over complete temperature range
- Minimum/maximum values of state machine timings are determined using corner lots over 0...+65 °C temperature range

The specification tables are split into pressure and temperature part of BMP280

Table 2: Parameter specification

Parameter	Symbol	Condition	Min	Typ	Max	Units
Operating temperature range	T_A	operational	-40	25	+85	°C
		full accuracy	0		+65	
Operating pressure range	P	full accuracy	300		1100	hPa
Sensor supply voltage	V_{DD}	ripple max. 50mVpp	1.71	1.8	3.6	V
Interface supply voltage	V_{DDIO}		1.2	1.8	3.6	V
Supply current	$I_{DD,LP}$	1 Hz forced mode, pressure and temperature, lowest power		2.8	4.2	µA
Peak current	I_{peak}	during pressure measurement		720	1120	µA
Current at temperature measurement	I_{DDT}			325		µA
Sleep current[1]	I_{DDSL}	25 °C		0.1	0.3	µA
Standby current (inactive period of normal mode) [2]	I_{DDSB}	25 °C		0.2	0.5	µA
Relative accuracy pressure V_{DD} = 3.3V	A_{rel}	700 ... 900hPa		±0.12		hPa
		25 . . . 40 °C		±1.0		m

[1] Typical value at VDD = VDDIO = 1.8 V, maximal value at VDD = VDDIO = 3.6 V.

[2] Typical value at VDD = VDDIO = 1.8 V, maximal value at VDD = VDDIO = 3.6 V.

Offset temperature coefficient	TCO	900hPa 25 . . . 40 °C		±1.5		Pa/K
				12.6		cm/K
Absolute accuracy pressure	A^P_{ext}	300 . . . 1100 hPa -20 . . . 0 °C		±1.7		hPa
	A^P_{full}	300 . . . 1100 hPa 0 . . . 65 °C		±1.0		hPa
Resolution of output data in ultra high resolution mode	R^P	Pressure		0.0016		hPa
	R^T	Temperature		0.01		°C
Noise in pressure	$V_{p,full}$	Full bandwidth, ultra high resolution See chapter 3.5		1.3		Pa
				11		cm
	$V_{p,filtered}$	Lowest bandwidth, ultra high resolution See chapter 3.5		0.2		Pa
				1.7		cm
Absolute accuracy temperature[3]	A^T	@ 25 °C		±0.5		°C
		0 . . . +65 °C		±1.0		°C
PSRR (DC)	PSRR	full V_{DD} range			±0.005	Pa/mV
Long term stability[4]	ΔP_{stab}	12 months		±1.0		hPa
Solder drifts		Minimum solder height 50 µm	-0.5		+2	hPa
Start-up time	$t_{startup}$	Time to first communication after both V_{DD} > 1.58V and V_{DDIO} > 0.65V			2	ms
Possible sampling rate	f_{sample}	*osrs_t* = *osrs_p* = 1; See chapter 3.8	157	182	tbd[5]	Hz
Standby time accuracy	$\Delta t_{standby}$			±5	±25	%

[3] 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.
[4] Long term stability is specified in the full accuracy operating pressure range 0 ... 65°C
[5] Depends on application case, please contact Application Engineer for further questions

2. Absolute maximum ratings

The absolute maximum ratings are provided in Table 3.

Table 3: Absolute maximum ratings

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	V_{DDIO} + 0.3	V
Storage Temperature	≤ 65% rel. H.	-45	+85	°C
Pressure		0	20 000	hPa
ESD	HBM, at any Pin		±2	kV
	CDM		±500	V
	Machine model		±200	V

3. Functional description

The BMP280 consists of a Piezo-resistive pressure sensing element and a mixed-signal ASIC. The ASIC performs A/D conversions and provides the conversion results and sensor specific compensation data through a digital interface.

BMP280 provides highest flexibility to the designer and can be adapted to the requirements regarding accuracy, measurement time and power consumption by selecting from a high number of possible combinations of the sensor settings.

BMP280 can be operated in three power modes (see chapter 3.6):

- sleep mode
- normal mode
- forced mode

In sleep mode, no measurements are performed. Normal mode comprises an automated perpetual cycling between an active measurement period and an inactive standby period. In forced mode, a single measurement is performed. When the measurement is finished, the sensor returns to sleep mode.

A set of oversampling settings is available ranging from ultra low power to ultra high resolution setting in order to adapt the sensor to the target application. The settings are predefined combinations of pressure measurement oversampling and temperature measurement oversampling. Pressure and temperature measurement oversampling can be selected independently from 0 to 16 times oversampling (see chapter 3.3.1 and 3.3.2):

- Temperature measurement
- Ultra low power
- Low power
- Standard resolution
- High resolution
- Ultra high resolution

BMP280 is equipped with a built-in IIR filter in order to minimize short-term disturbances in the output data caused by the slamming of a door or window. The filter coefficient ranges from 0 (off) to 16.

In order to simplify the device usage and reduce the high number of possible combinations of power modes, oversampling rates and filter settings, Bosch Sensortec provides a proven set of recommendations for common use-cases in smart-phones, mobile weather stations or flying toys (see chapter 3.4):

- Handheld device low-power (e.g. smart phones running Android)
- Handheld device dynamic (e.g. smart phones running Android)
- Weather monitoring (setting with lowest power consumption)
- Elevator / floor change detection
- Drop detection
- Indoor navigation

3.1 Block diagram

Figure 1 shows a simplified block diagram of the BMP280:

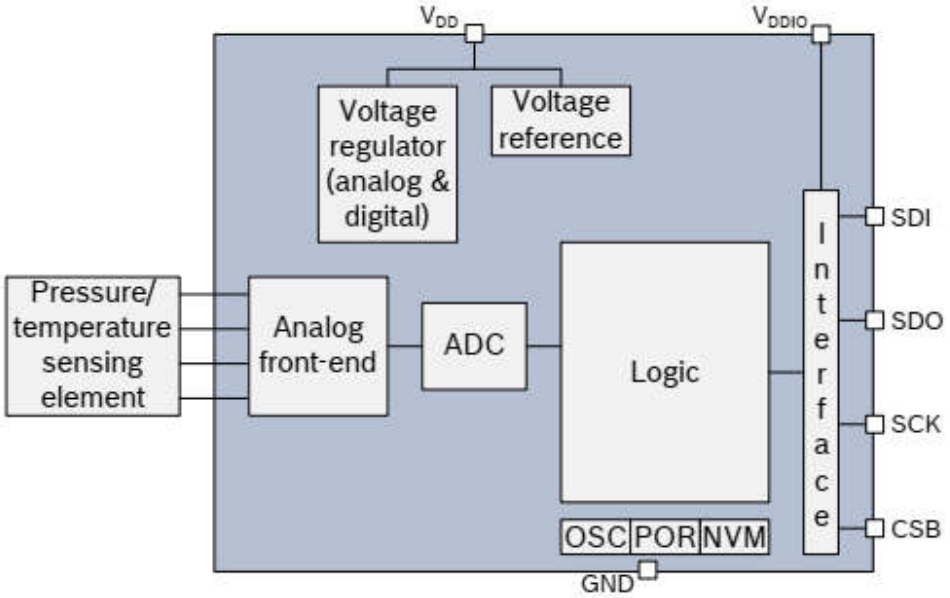


Figure 1: Block diagram of BMP280

3.2 Power management

The BMP280 has two separate power supply pins

- V_{DD} is the main power supply for all internal analog and digital functional blocks
- V_{DDIO} is a separate power supply pin, used for the supply of the digital interface

A power-on reset generator is built in which resets the logic circuitry and the register values after the power-on sequence. There are no limitations on slope and sequence of raising the V_{DD} and V_{DDIO} levels. After powering up, the sensor settles in sleep mode (see 3.6.1).

Warning. Holding any interface pin (SDI, SDO, SCK or CSB) at a logical high level when V_{DDIO} is switched off can permanently damage the device due caused by excessive current flow through the ESD protection diodes.

If V_{DDIO} is supplied, but V_{DD} is not, the interface pins are kept at a high-Z level. The bus can therefore already be used freely before the BMP280 V_{DD} supply is established.

3.3 Measurement flow

The BMP280 measurement period consists of a temperature and pressure measurement with selectable oversampling. After the measurement period, the data are passed through an optional IIR filter, which removes short-term fluctuations in pressure (e.g. caused by slamming a door). The flow is depicted in the diagram below.

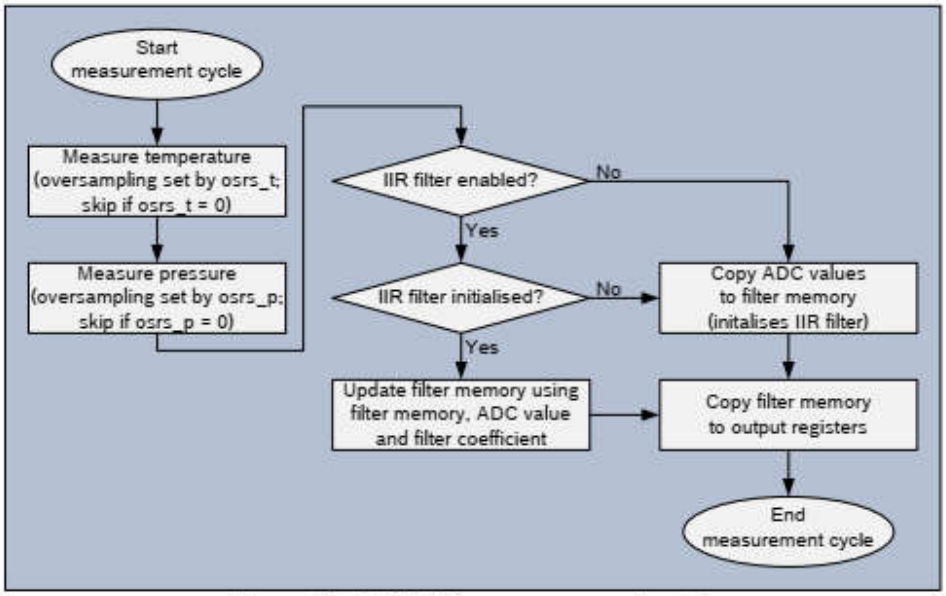


Figure 2: BMP280 measurement cycle

The individual blocks of the diagram above will be detailed in the following subchapters.

3.3.1 Pressure measurement

Pressure measurement can be enabled or skipped. Skipping the measurement could be useful if BMP280 is used as temperature sensor. When enabled, several oversampling options exist. Each oversampling step reduces noise and increases the output resolution by one bit, which is stored in the XLSB data register 0xF9. Enabling/disabling the measurement and oversampling settings are selected through the osrs_p[2:0] bits in control register 0xF4.

Table 4: *osrs_p* settings

Oversampling setting	Pressure oversampling	Typical pressure resolution	Recommended temperature oversampling
Pressure measurement skipped	Skipped (output set to 0x80000)	–	As needed
Ultra low power	×1	16 bit / 2.62 Pa	×1
Low power	×2	17 bit / 1.31 Pa	×1
Standard resolution	×4	18 bit / 0.66 Pa	×1
High resolution	×8	19 bit / 0.33 Pa	×1
Ultra high resolution	×16	20 bit / 0.16 Pa	×2

In order to find a suitable setting for *osrs_p*, please consult chapter 3.4.

3.3.2 Temperature measurement

Temperature measurement can be enabled or skipped. Skipping the measurement could be useful to measure pressure extremely rapidly. When enabled, several oversampling options exist. Each oversampling step reduces noise and increases the output resolution by one bit, which is stored in the XLSB data register 0xFC. Enabling/disabling the temperature measurement and oversampling setting are selected through the osrs_t[2:0] bits in control register 0xF4.

Table 5: *osrs_t* settings

***osrs_t*[2:0]**	**Temperature oversampling**	**Typical temperature resolution**
000	Skipped (output set to 0x80000)	–
001	×1	16 bit / 0.0050 °C
010	×2	17 bit / 0.0025 °C
011	×4	18 bit / 0.0012 °C
100	×8	19 bit / 0.0006 °C
101, 110, 111	×16	20 bit / 0.0003 °C

It is recommended to base the value of *osrs_t* on the selected value of *osrs_p* as per Table 4. Temperature oversampling above ×2 is possible, but will not significantly improve the accuracy of the pressure output any further. The reason for this is that the noise of the compensated pressure value depends more on the raw pressure than on the raw temperature noise. Following the recommended setting will result in an optimal noise-to-power ratio.

3.3.3 IIR filter

The environmental pressure is subject to many short-term changes, caused e.g. by slamming of a door or window, or wind blowing into the sensor. To suppress these disturbances in the output data without causing additional interface traffic and processor work load, the BMP280 features an internal IIR filter. It effectively reduces the bandwidth of the output signals[6]. The output of a next measurement step is filter using the following formula:

$$data\_filtered = \frac{data\_filtered\_old \cdot (filter\_coefficient - 1) + data\_ADC}{filter\_coefficient},$$

where data_filtered_old is the data coming from the previous acquisition, and data_ADC is the data coming from the ADC before IIR filtering.

The IIR filter can be configured using the filter[2:0] bits in control register 0xF5 with the following options:

[6] Since most pressure sensors do not sample continuously, filtering can suffer from signals with a frequency higher than the sampling rate of the sensor. E.g. environmental fluctuations caused by windows being opened and closed might have a frequency <5 Hz. Consequently, a sampling rate of ODR = 10 Hz is sufficient to obey the Nyquist theorem.

Table 6: *filter* settings

Filter coefficient	Samples to reach ≥75 % of step response
Filter off	1
2	2
4	5
8	11
16	22

In order to find a suitable setting for *filter*, please consult chapter 3.4.

When writing to the register *filter*, the filter is reset. The next value will pass through the filter and be the initial memory value for the filter. If temperature or pressure measurement is skipped, the corresponding filter memory will be kept unchanged even though the output registers are set to 0x80000. When the previously skipped measurement is re-enabled, the output will be filtered using the filter memory from the last time when the measurement was not skipped.

3.4 Filter selection

In order to select optimal settings, the following use cases are suggested:

Table 7: Recommended filter settings based on use cases

Use case	Mode	Over-sampling setting	osrs_p	osrs_t	IIR filter coeff. (see 3.3.3)	I_{DD} [µA] (see 3.7)	ODR [Hz] (see 3.8.2)	RMS Noise [cm] (see 3.5)
handheld device low-power (e.g. Android)	Normal	Ultra high resolution	×16	×2	4	247	10.0	4.0
handheld device dynamic (e.g. Android)	Normal	Standard resolution	×4	×1	16	577	83.3	2.4
Weather monitoring (lowest power)	Forced	Ultra low power	×1	×1	Off	0.14	1/60	26.4
Elevator / floor change detection	Normal	Standard resolution	×4	×1	4	50.9	7.3	6.4
Drop detection	Normal	Low power	×2	×1	Off	509	125	20.8
Indoor navigation	Normal	Ultra high resolution	×16	×2	16	650	26.3	1.6

3.5 Noise

Noise depends on the oversampling and filter settings selected. The stated values were determined in a controlled pressure environment and are based on the average standard deviation of 32 consecutive measurement points taken at highest sampling speed. This is needed in order to exclude long term drifts from the noise measurement.

Table 8: Noise in pressure

Typical RMS noise in pressure [Pa]					
Oversampling setting	**IIR filter coefficient**				
	off	**2**	**4**	**8**	**16**
Ultra low power	3.3	1.9	1.2	0.9	0.4
Low power	2.6	1.5	1.0	0.6	0.4
Standard resolution	2.1	1.2	0.8	0.5	0.3
High resolution	1.6	1.0	0.6	0.4	0.2
Ultra high resolution	1.3	0.8	0.5	0.4	0.2

Table 9: Noise in temperature

Typical RMS noise in temperature [°C]	
Temperature oversampling	**IIR filter off**
oversampling ×1	0.005
oversampling ×2	0.004
oversampling ×4	0.003
oversampling ×8	0.003
oversampling ×16	0.002

3.6 Power modes

The BMP280 offers three power modes: sleep mode, forced mode and normal mode. These can be selected using the mode[1:0] bits in control register 0xF4.

Table 10: *mode* settings

***mode*[1:0]**	**Mode**
00	Sleep mode
01 and 10	Forced mode
11	Normal mode

3.6.1 Sleep mode

Sleep mode is set by default after power on reset. In sleep mode, no measurements are performed and power consumption (I_{DDSM}) is at a minimum. All registers are accessible; Chip-ID and compensation coefficients can be read.

3.6.2 Forced mode

In forced mode, a single measurement is performed according to selected measurement and filter options. When the measurement is finished, the sensor returns to sleep mode and the measurement results can be obtained from the data registers. For a next measurement, forced mode needs to be selected again. This is similar to BMP180 operation. Forced mode is recommended for applications which require low sampling rate or host-based synchronization.

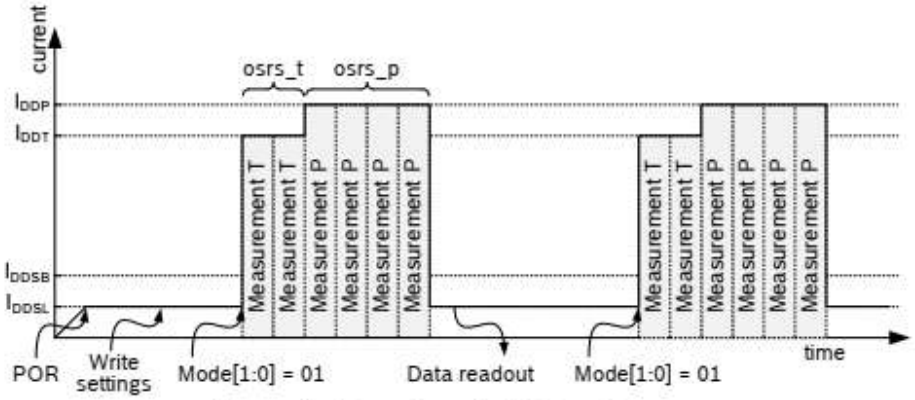


Figure 3: Forced mode timing diagram

3.6.3 Normal mode

Normal mode continuosly cycles between an (active) measurement period and an (inactive) standby period, whose time is defined by $t_{standby}$. The current in the standby period (I_{DDSB}) is slightly higher than in sleep mode. After setting the mode,measurement and filter options, the last measurement results can be obtained from the data registers without the need of further write accesses. Normal mode is recommended when using the IIR filter, and useful for applications in which short-term disturbances (e.g. blowing into the sensor) should be filtered.

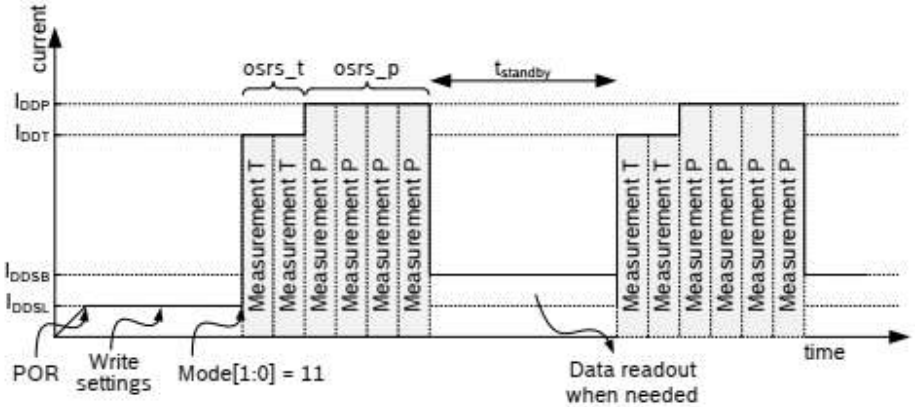


Figure 4: Normal mode timing diagram

The standby time is determined by the contents of the t_sb[2:0] bits in control register 0xF5 according to the table below:

Table 11: *t_sb* settings

***t_sb*[1:0]**	**$t_{standby}$ [ms]**
000	0.5
001	62.5
010	125
011	250
100	500
101	1000
110	2000
111	4000

3.6.4 Mode transition diagram

The supported mode transitions are displayed below. If the device is currently performing a measurement, execution of mode switching commands is delayed until the end of the currently running measurement period. Further mode change commands are ignored until the last mode change command is executed. Mode transitions other than the ones shown below are tested for stability but do not represent recommended use of the device.

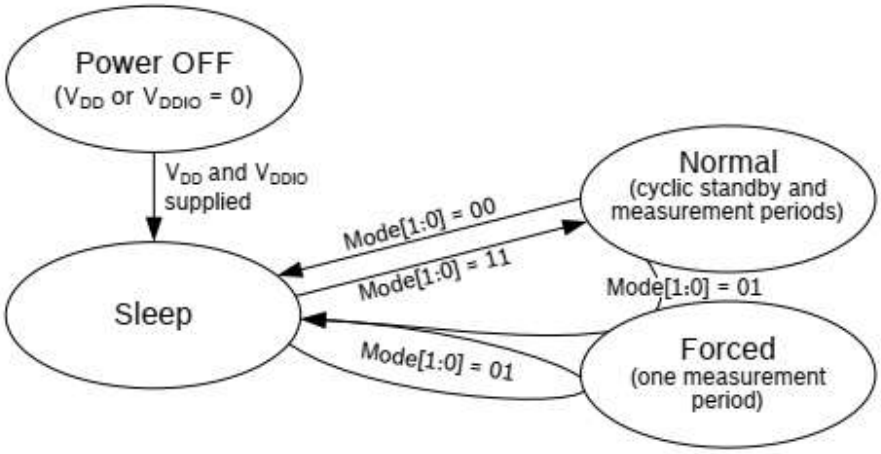


Figure 5: Mode transition diagram

3.7 Current consumption

The current consumption depends on ODR and oversampling setting. The values given below are normalized to an ODR of 1 Hz. The actual consumption at a given ODR can be calculated by multiplying the consumption in Table 12 with the ODR used. The actual ODR is defined either by the frequency at which the user sets forced measurements or by oversampling and $t_{standby}$ settings in normal mode in Table 14.

Table 12: Current consumption

Oversampling setting	Pressure oversampling	Temperature oversampling	I_{DD} [µA] @ 1 Hz forced mode	
			Typ	Max
Ultra low power	×1	×1	2.74	4.16
Low power	×2	×1	4.17	6.27
Standard resolution	×4	×1	7.02	10.50
High resolution	×8	×1	12.7	18.95
Ultra high resolution	×16	×2	24.8	36.85

3.8 Measurement timings

The rate at which measurements can be performed in forced mode depends on the oversampling settings *osrs_t* and *osrs_p*. The rate at which they are performed in normal mode depends on the oversampling setting settings *osrs_t* and *osrs_p* and the standby time $t_{standby}$. In the following table the resulting ODRs are given only for the suggested osrs combinations.

3.8.1 Measurement time

The following table explains the typical and maximum measurement time based on selected oversampling setting. The minimum achievable frequency is determined by the maximum measurement time.

Table 13: measurement time

Oversampling setting	Pressure oversampling	Temperature oversampling	Measurement time [ms]		Measurement rate [Hz]	
			Typ	Max	Typ	Min
Ultra low power	×1	×1	5.5	6.4	181.8	155.6
Low power	×2	×1	7.5	8.7	133.3	114.6
Standard resolution	×4	×1	11.5	13.3	87.0	75.0
High resolution	×8	×1	19.5	22.5	51.3	44.4
Ultra high resolution	×16	×2	37.5	43.2	26.7	23.1

3.8.2 Measurement rate in normal mode

The following table explains which measurement rates can be expected in normal mode based on oversampling setting and $t_{standby}$.

Table 14: typical output data Rate (ODR) in normal mode [Hz]

Oversampling setting	$t_{standby}$ [ms]							
	0.5	62.5	125	250	500	1000	2000	4000
Ultra low power	166.67	14.71	7.66	3.91	1.98	0.99	0.50	0.25
Low power	125.00	14.29	7.55	3.88	1.97	0.99	0.50	0.25
Standard resolution	83.33	13.51	7.33	3.82	1.96	0.99	0.50	0.25
High resolution	50.00	12.20	6.92	3.71	1.92	0.98	0.50	0.25
Ultra high resolution	26.32	10.00	6.15	3.48	1.86	0.96	0.49	0.25

Table 15: Sensor timing according to recommended settings (based on use cases)

Use case	Mode	Over-sampling setting	osrs_p	osrs_t	IIR filter coeff. (see 3.3.3)	Timing	ODR [Hz] (see 3.8.2)	BW [Hz] (see 3.3.3)
handheld device low-power (e.g. Android)	Normal	Ultra high resolution	×16	×2	4	$t_{standby}$ = 62.5 ms	10.0	0.92
handheld device dynamic (e.g. Android)	Normal	Standard resolution	×4	×1	16	$t_{standby}$ = 0.5 ms	83.3	1.75
Weather monitoring (lowest power)	Forced	Ultra low power	×1	×1	Off	1/min	1/60	full
Elevator / floor change detection	Normal	Standard resolution	×4	×1	4	$t_{standby}$ = 125 ms	7.3	0.67
Drop detection	Normal	Low power	×2	×1	Off	$t_{standby}$ = 0.5 ms	125	full
Indoor navigation	Normal	Ultra high resolution	×16	×2	16	$t_{standby}$ = 0.5 ms	26.3	0.55

3.9 Data readout

To read out data after a conversion, it is strongly recommended to use a burst read and not address every register individually. This will prevent a possible mix-up of bytes belonging to different measurements and reduce interface traffic. Data readout is done by starting a burst read from 0xF7 to 0xFC. The data are read out in an unsigned 20-bit format both for pressure and for temperature. It is strongly recommended to use the BMP280 API, available from Bosch Sensortec, for readout and compensation. For details on memory map and interfaces, please consult chapters 3.12 and 5 respectively.

The timing for data readout in forced mode should be done so that the maximum measurement times (see chapter 3.8.1) are respected. In normal mode, readout can be done at a speed similar to the expected data output rate (see chapter 3.8.2). After the values of 'ut' and 'up' have been read, the actual pressure and temperature need to be calculated using the compensation parameters stored in the device. The procedure is elaborated in chapter 3.11.

3.10 Data register shadowing

In normal mode, measurement timing is not necessarily synchronized to readout. This means that new measurement results may become available while the user is reading the results from the previous measurement. In this case, shadowing is performed in order to guarantee data consistency. Shadowing will only work if all data registers are read in a single burst read. Therefore, the user must use burst reads if he does not synchronize data readout with the measurement cycle. Using several independent read commands may result in inconsistent data.

If a new measurement is finished and the data registers are still being read, the new measurement results are transferred into shadow data registers. The content of shadow registers is transferred into data registers as soon as the user ends the burst read, even if not all data registers were read. Reading across several data registers can therefore only be guaranteed to be consistent within one measurement cycle if a single burst read command is used. The end of the burst read is marked by the rising edge of CSB pin in SPI case or by the recognition of a stop condition in I2C case. After the end of the burst read, all user data registers are updated at once.

3.11 Output compensation

The BMP280 output consists of the ADC output values. However, each sensing element behaves differently, and actual pressure and temperature must be calculated using a set of calibration parameters. The recommended calculation in chapter 3.11.3 uses fixed point arithmetic. In high-level languages like Matlab™ or LabVIEW™, fixed-point code may not be well supported. In this case the floating-point code in appendix 8.1 can be used as an alternative. For 8-bit micro controllers, the variable size may be limited. In this case a simplified 32 bit integer code with reduced accuracy is given in appendix 8.2.

3.11.1 Computational requirements

The table below shows the number of clock cycles needed for compensation calculations on a 32 bit Cortex-M3 micro controller with GCC optimization level –O2. This controller does not contain a floating point unit, so all floating-point calculations are emulated. Floating point is only recommended for PC applications where an FPU is present.

Table 16: Computational requirements for compensation formulas

Compensation of	Number of clock cycles (ARM Cortex-M3)		
	32 bit integer	64 bit integer	Double precision
Temperature	~46	–	~2400 [7]
Pressure	~112 [8]	~1400	~5400 [7]

[7] Use only recommended for high-level programming languages like Matlab™ or LabVIEW™
[8] Use only recommended for 8-bit micro controllers

3.11.2 Trimming parameter readout

The trimming parameters are programmed into the devices' non-volatile memory (NVM) during production and cannot be altered by the customer. Each compensation word is a 16-bit signed or unsigned integer value stored in two's complement. As the memory is organized into 8-bit words, two words must always be combined in order to represent the compensation word. The 8-bit registers are named calib00...calib25 and are stored at memory addresses 0x88...0xA1. The corresponding compensation words are named dig_T# for temperature compensation related values and dig_P# for pressure compensation related values. The mapping is shown in Table 17.

Table 17: Compensation parameter storage, naming and data type

Register Address LSB / MSB	Register content	Data type
0x88 / 0x89	dig_T1	unsigned short
0x8A / 0x8B	dig_T2	signed short
0x8C / 0x8D	dig_T3	signed short
0x8E / 0x8F	dig_P1	unsigned short
0x90 / 0x91	dig_P2	signed short
0x92 / 0x93	dig_P3	signed short
0x94 / 0x95	dig_P4	signed short
0x96 / 0x97	dig_P5	signed short
0x98 / 0x99	dig_P6	signed short
0x9A / 0x9B	dig_P7	signed short
0x9C / 0x9D	dig_P8	signed short
0x9E / 0x9F	dig_P9	signed short
0xA0 / 0xA1	reserved	reserved

3.11.3 Compensation formula

Please note that it is strongly advised to use the API available from Bosch Sensortec to perform readout and compensation. If this is not wanted, the code below can be applied at the user's risk. Both pressure and temperature values are expected to be received in 20 bit format, positive, stored in a 32 bit signed integer.

The variable t_fine (signed 32 bit) carries a fine resolution temperature value over to the pressure compensation formula and could be implemented as a global variable.
The data type "BMP280_S32_t" should define a 32 bit signed integer variable type and can usually be defined as "long signed int".

The data type "BMP280_U32_t" should define a 32 bit unsigned integer variable type and can usually be defined as "long unsigned int".

For best possible calculation accuracy, 64 bit integer support is needed. If this is not possible on your platform, please see appendix 8.2 for a 32 bit alternative.

The data type “BMP280_S64_t” should define a 64 bit signed integer variable type, which on most supporting platforms can be defined as “long long signed int”. The revision of the code is rev.1.1.

```
// Returns temperature in DegC, resolution is 0.01 DegC. Output value of "5123" equals 51.23 DegC.
// t_fine carries fine temperature as global value
BMP280_S32_t t_fine;
BMP280_S32_t bmp280_compensate_T_int32(BMP280_S32_t adc_T)
{
    BMP280_S32_t var1, var2, T;
    var1  = ((((adc_T>>3) - ((BMP280_S32_t)dig_T1<<1))) * ((BMP280_S32_t)dig_T2)) >> 11;
    var2  = (((((adc_T>>4) - ((BMP280_S32_t)dig_T1)) * ((adc_T>>4) - ((BMP280_S32_t)dig_T1))) >> 12) *
       ((BMP280_S32_t)dig_T3)) >> 14;
    t_fine = var1 + var2;
    T  = (t_fine * 5 + 128) >> 8;
    return T;
}
"",
// Returns pressure in Pa as unsigned 32 bit integer in Q24.8 format (24 integer bits and 8 fractional bits).
// Output value of "24674867" represents 24674867/256 = 96386.2 Pa = 963.862 hPa
BMP280_U32_t bmp280_compensate_P_int64(BMP280_S32_t adc_P)
{
    BMP280_S64_t var1, var2, p;
    var1 = ((BMP280_S64_t)t_fine) - 128000;
    var2 = var1 * var1 * (BMP280_S64_t)dig_P6;
    var2 = var2 + ((var1*(BMP280_S64_t)dig_P5)<<17);
    var2 = var2 + (((BMP280_S64_t)dig_P4)<<35);
    var1 = ((var1 * var1 * (BMP280_S64_t)dig_P3)>>8) + ((var1 * (BMP280_S64_t)dig_P2)<<12);
    var1 = (((((BMP280_S64_t)1)<<47)+var1))*((BMP280_S64_t)dig_P1)>>33;
    if (var1 == 0)
    {
        return 0; // avoid exception caused by division by zero
    }
    p = 1048576-adc_P;
    p = (((p<<31)-var2)*3125)/var1;
    var1 = (((BMP280_S64_t)dig_P9) * (p>>13) * (p>>13)) >> 25;
    var2 = (((BMP280_S64_t)dig_P8) * p) >> 19;
    p = ((p + var1 + var2) >> 8) + (((BMP280_S64_t)dig_P7)<<4);
    return (BMP280_U32_t)p;
```

3.12 Calculating pressure and temperature

The following figure shows the detailed algorithm for pressure and temperature measurement.

This algorithm is available to customers as reference C source code (“BMP28x_ API”) from Bosch Sensortec and via its sales and distribution partners.

Please contact your Bosch Sensortec representative for details.

Calculation of pressure and temperature for BMP280

Sample trimming values

Register Address (LSB / MSB)	Name	Value	Type
0x88 / 0x89	dig_T1	27504	unsigned short
0x8A / 0x8B	dig_T2	26435	short
0x8C / 0x8D	dig_T3	-1000	short
0x8E / 0x8F	dig_P1	36477	unsigned short
0x90 / 0x91	dig_P2	-10685	short
0x92 / 0x93	dig_P3	3024	short
0x94 / 0x95	dig_P4	2855	short
0x96 / 0x97	dig_P5	140	short
0x98 / 0x99	dig_P6	-7	short
0x9A / 0x9B	dig_P7	15500	short
0x9C / 0x9D	dig_P8	-14600	short
0x9E / 0x9F	dig_P9	6000	short
0xA0 / 0xA1			

Sample measurement values

Register Address (MSB / LSB / XLSB)	Name	Value	Type	
0xF7 / 0xF8 / 0xF9[7:4]	UT [20 bit]	519888	signed long (*)	(*) Value is always positive, even though the compensation functions expect a signed integer as input
0xFA / 0xFB / 0xFC[7:4]	UP [20 bit]	415148	signed long (*)	(*) Value is always positive, even though the compensation functions expect a signed integer as input

var1 =	128793,1787		var1 = (((double)adc_T)/16384.0 - ((double)dig_T1)/1024.0) * ((double)dig_T2);
var2 =	-370,8917052		var2 = (((((double)adc_T)/131072.0 - ((double)dig_T1)/8192.0) * (((double)adc_T)/131072.0 - ((double)dig_T1)/8192.0)) * ((double)dig_T3);
tfine =	128422		t_fine = (BMP280_S32_t)(var1 + var2);
T =	25,08	*Temperature [°C]*	T = (var1 + var2) / 5120.0;
integer result (**):	2508	*Temperature [1/100 °C]*	
var1 =	211,1435029		var1 = (((double)t_fine)/2.0) - 64000.0;
var2 =	-9,523652701		var2 = var1 * var1 * ((double)dig_P6) / 32768.0;
var2 =	59110,65716		var2 = var2 + var1 * ((double)dig_P5) * 2.0;
var2 =	187120057,7		var2 = (var2/4.0)+(((double)dig_P4) * 65536.0);
var1 =	-4,302618389		var1 = (((double)dig_P3) * var1 * var1 / 524288.0 + ((double)dig_P2) * var1) / 524288.0;
var1 =	36472,21037		var1 = (1.0 + var1 / 32768.0) * ((double)dig_P1);
p =	633428		p = 1048576.0 - (double)adc_P;
p =	100717,8456		p = (p - (var2 / 4096.0)) * 6250.0 / var1;
var1 =	28342,24444		var1 = ((double)dig_P9) * p * p / 2147483648.0;
var2 =	-44875,50492		var2 = p * ((double)dig_P8) / 32768.0;
p =	**100653,27**	*Pressure [Pa]*	p = p + (var1 + var2 + ((double)dig_P7)) / 16.0;
int32 result (**):	**100653**	*Pressure [Pa]*	
int64 result (**):	**25767236**	*Pressure [1/256 Pa]*	

(**) The actual result of the integer calculation may deviate slightly from the values shown here due to integer calculation rounding errors

4. Global memory map and register description

4.1 General remarks

All communication with the device is performed by reading from and writing to registers. Registers have a width of 8 bits. There are several registers which are reserved; they should not be written to and no specific value is guaranteed when they are read. For details on the interface, consult chapter 5.

4.2 Memory map

The memory map is given in Table 18 below. Reserved registers are not shown.

Table 18: Memory map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]				filter[2:0]			spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]				osrs_p[2:0]		mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x58
calib25...calib00	0xA1...0x88	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

4.3 Register description

4.3.1 Register 0xD0 *"id"*

The *"id"* register contains the chip identification number chip_id[7:0], which is 0x58. This number can be read as soon as the device finished the power-on-reset.

4.3.2 Register 0xE0 *"reset"*

The *"reset"* register contains the soft reset word reset[7:0]. If the value 0xB6 is written to the register, the device is reset using the complete power-on-reset procedure. Writing other values than 0xB6 has no effect. The readout value is always 0x00.

4.3.3 Register 0xF3 *"status"*

The *"status"* register contains two bits which indicate the status of the device.

Table 19: Register 0xF3 *"status"*

Register 0xF3 *"status"*	Name	Description
Bit 3	measuring[0]	Automatically set to '1' whenever a conversion is running and back to '0' when the results have been transferred to the data registers.
Bit 0	im_update[0]	Automatically set to '1' when the NVM data are being copied to image registers and back to '0' when the copying is done. The data are copied at power-on-reset and before every conversion.

4.3.4 Register 0xF4 *"ctrl_meas"*

The *"ctrl_meas"* register sets the data acquisition options of the device.

Table 20: Register 0xF4 *"ctrl_meas"*

Register 0xF4 *"ctrl_meas"*	Name	Description
Bit 7, 6, 5	osrs_t[2:0]	Controls oversampling of temperature data. See chapter 3.3.2 for details.
Bit 4, 3, 2	osrs_p[2:0]	Controls oversampling of pressure data. See chapter 3.3.1 for details.
Bit 1, 0	mode[1:0]	Controls the power mode of the device. See chapter 3.6 for details.

Table 21: register settings *osrs_p*

osrs_p[2:0]	Pressure oversampling
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, Others	oversampling ×16

Table 22: register settings *osrs_t*

***osrs_t*[2:0]**	**Temperature oversampling**
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, 110, 111	oversampling ×16

4.3.5 Register 0xF5 *"config"*

The *"config"* register sets the rate, filter and interface options of the device. Writes to the "config" register in normal mode may be ignored. In sleep mode writes are not ignored.

Table 23: Register 0xF5 *"config"*

Register 0xF5 *"config"*	**Name**	**Description**
Bit 7, 6, 5	t_sb[2:0]	Controls inactive duration $t_{standby}$ in normal mode. See chapter 3.6.3 for details.
Bit 4, 3, 2	filter[2:0]	Controls the time constant of the IIR filter. See chapter 3.3.3 for details.
Bit 0	spi3w_en[0]	Enables 3-wire SPI interface when set to '1'. See chapter 5.3 for details.

4.3.6 Register 0xF7...0xF9 *"press" (_msb, _lsb, _xlsb)*

The *"press"* register contains the raw pressure measurement output data up[19:0]. For details on how to read out the pressure and temperature information from the device, please consult chapter3.9.

Table 24: Register 0xF7 ... 0xF9 *"press"*

Register 0xF7-0xF9 *"press"*	**Name**	**Description**
0xF7	press_msb[7:0]	Contains the MSB part up[19:12] of the raw pressure measurement output data.
0xF8	press_lsb[7:0]	Contains the LSB part up[11:4] of the raw pressure measurement output data.
0xF9 (bit 7, 6, 5, 4)	press_xlsb[3:0]	Contains the XLSB part up[3:0] of the raw pressure measurement output data. Contents depend on temperature resolution, see table 5.

4.3.7 Register 0xFA...0xFC *"temp" (_msb, _lsb, _xlsb)*

The *"temp"* register contains the raw temperature measurement output data ut[19:0]. For details on how to read out the pressure and temperature information from the device, please consult chapter 3.9.

Table 25: Register 0xFA ... 0xFC *"temp"*

Register 0xF7-0xF9 *"press"*	Name	Description
0xFA	temp_msb[7:0]	Contains the MSB part ut[19:12] of the raw temperature measurement output data.
0xFB	temp_lsb[7:0]	Contains the LSB part ut[11:4] of the raw temperature measurement output data.
0xFC (bit 7, 6, 5, 4)	temp_xlsb[3:0]	Contains the XLSB part ut[3:0] of the raw temperature measurement output data. Contents depend on pressure resolution, see Table 4.

5. Digital interfaces

The BMP280 supports the I²C and SPI digital interfaces; it acts as a slave for both protocols. The I²C interface supports the Standard, Fast and High Speed modes. The SPI interface supports both SPI mode '00' (CPOL = CPHA = '0') and mode '11' (CPOL = CPHA = '1') in 4-wire and 3-wire configuration.

The following transactions are supported:

- Single byte write
- multiple byte write (using pairs of register addresses and register data)
- single byte read
- multiple byte read (using a single register address which is auto-incremented)

5.1 Interface selection

Interface selection is done automatically based on CSB (chip select) status. If CSB is connected to V_{DDIO}, the I²C interface is active. If CSB is pulled down, the SPI interface is activated. After CSB has been pulled down once (regardless of whether any clock cycle occurred), the I²C interface is disabled until the next power-on-reset. This is done in order to avoid inadvertently decoding SPI traffic to another slave as I²C data. Since power-on-reset is only executed when both V_{DD} and V_{DDIO} are established, there is no risk of incorrect protocol detection due to power-up sequence used. However, if I²C is to be used and CSB is not directly connected to V_{DDIO} but rather through a programmable pin, it must be ensured that this pin already outputs the V_{DDIO} level during power-on-reset of the device. If this is not the case, the device will be locked in SPI mode and not respond to I²C commands.

5.2 I²C Interface

The I²C slave interface is compatible with Philips I²C Specification version 2.1. For detailed timings refer to Table 27. All modes (standard, fast, high speed) are supported. SDA and SCL are not pure open-drain. Both pads contain ESD protection diodes to VDDIO and GND. As the devices does not perform clock stretching, the SCL structure is a high-Z input without drain capability.

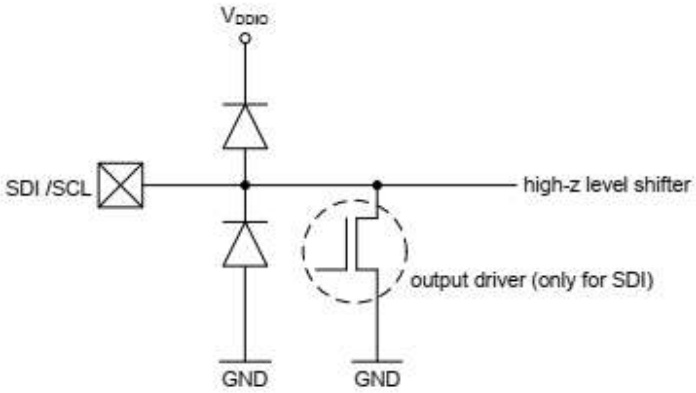


Figure 6: SDI/SCK ESD drawing

The 7-bit device address is 111011x. The 6 MSB bits are fixed. The last bit is changeable by SDO value and can be changed during operation. Connecting SDO to GND results in slave address 1110110 (0x76); connection it to V_{DDIO} results in slave address 1110111 (0x77), which

is the same as BMP180's I²C address. The SDO pin cannot be left floating; if left floating, the I²C address will be undefined.

The I²C interface uses the following pins:

- SCK: serial clock (SCL)
- SDI: data (SDA)
- SDO: Slave address LSB (GND = '0', V_{DDIO} = '1')

CSB must be connected to V_{DDIO} to select I²C interface. SDI is bi-directional with open drain to GND: it must be externally connected to V_{DDIO} via a pull up resistor. Refer to chapter 6 for connection instructions.

The following abbreviations will be used in the I²C protocol figures:

- S Start
- P Stop
- ACKS Acknowledge by slave
- ACKM Acknowledge by master
- NACKM Not acknowledge by master

5.2.1 I²C write

Writing is done by sending the slave address in write mode (RW = '0'), resulting in slave address 111011X0 ('X' is determined by state of SDO pin. Then the master sends pairs of register addresses and register data. The transaction is ended by a stop condition. This is depicted in Figure 7.

Figure 7: I²C multiple byte write (not auto-incremented)

5.2.2 I²C read

To be able to read registers, first the register address must be sent in write mode (slave address 111011X0). Then either a stop or a repeated start condition must be generated. After this the slave is addressed in read mode (RW = '1') at address 111011X1, after which the slave sends out data from auto-incremented register addresses until a NOACKM and stop condition occurs. This is depicted in Figure 8, where two bytes are read from register 0xF6 and 0xF7.

Figure 8: I²C multiple byte read

5.3 SPI interface

The SPI interface is compatible with SPI mode '00' (CPOL = CPHA = '0') and mode '11' (CPOL = CPHA = '1'). The automatic selection between mode '00' and '11' is determined by the value of SCK after the CSB falling edge.

The SPI interface has two modes: 4-wire and 3-wire. The protocol is the same for both. The 3-wire mode is selected by setting '1' to the register spi3w_en. The pad SDI is used as a data pad in 3-wire mode.

The SPI interface uses the following pins:

- CSB: chip select, active low
- SCK: serial clock
- SDI: serial data input; data input/output in 3-wire mode
- SDO: serial data output; hi-Z in 3-wire mode

Refer to chapter 6 for connection instructions.

CSB is active low and has an integrated pull-up resistor. Data on SDI is latched by the device at SCK rising edge and SDO is changed at SCK falling edge. Communication starts when CSB goes to low and stops when CSB goes to high; during these transitions on CSB, SCK must be stable. The SPI protocol is shown in Figure 9. For timing details, please review Table 28.

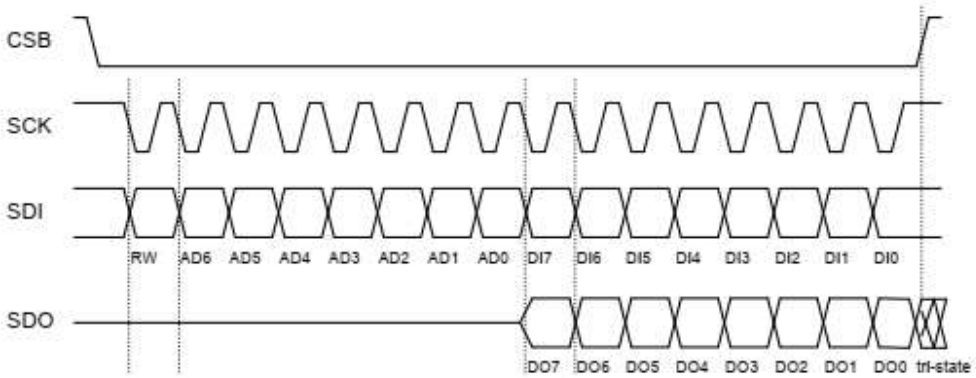


Figure 9: SPI protocol (shown for mode '11' in 4-wire configuration)

In SPI mode, only 7 bits of the register addresses are used; the MSB of register address is not used and replaced by a read/write bit (RW = '0' for write and RW = '1' for read).
Example: address 0xF7 is accessed by using SPI register address 0x77. For write access, the byte 0x77 is transferred, for read access, the byte 0xF7 is transferred.

5.3.1 SPI write

Writing is done by lowering CSB and sending pairs control bytes and register data. The control bytes consist of the SPI register address (= full register address without bit 7) and the write command (bit7 = RW = '0'). Several pairs can be written without raising CSB. The transaction is ended by a raising CSB. The SPI write protocol is depicted in Figure 10.

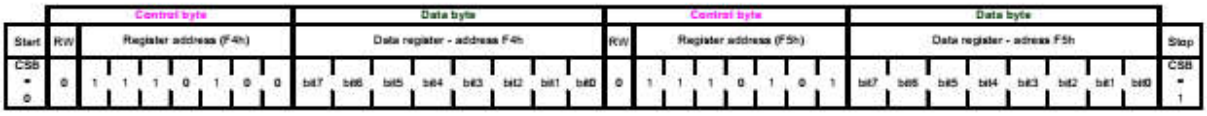


Figure 10: SPI multiple byte write (not auto-incremented)

5.3.2 SPI read

Reading is done by lowering CSB and first sending one control byte. The control bytes consist of the SPI register address (= full register address without bit 7) and the read command (bit 7 = RW = '1'). After writing the control byte, data is sent out of the SDO pin (SDI in 3-wire mode); the register address is automatically incremented. The SPI read protocol is shown in Figure 11.

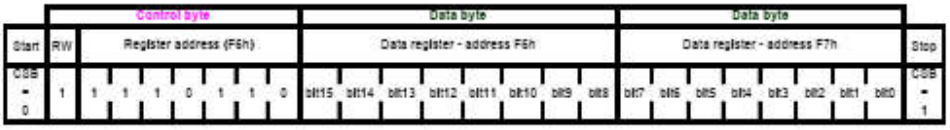


Figure 11: SPI multiple byte read

5.4 Interface parameter specification

5.4.1 General interface parameters

The general interface parameters are given in Table 26 below.

Table 26: interface parameters

Parameter	Symbol	Condition	Min	Typ	Max	Units
Input – low level	Vil_si	V_{DDIO}=1.2V to 3.6V			0.2 * V_{DDIO}	V
Input – high level	Vih_si	V_{DDIO}=1.2V to 3.6V	0.8 * V_{DDIO}			V
Output – low level for I2C	Vol_SDI	V_{DDIO}=1.62V, iol=3 mA			0.2 * V_{DDIO}	V
Output – low level for I2C	Vol_SDI_1.2	V_{DDIO}=1.20V, iol=3 mA			0.23 * V_{DDIO}	V
Output – low level	Vol SDO	V_{DDIO}=1.62V, iol=1 mA			0.2 * V_{DDIO}	V
Output – low level	Vol_SDO_1.2	V_{DDIO}=1.20V, iol=1 mA			0.23 * V_{DDIO}	V
Output – high level	Voh	V_{DDIO}=1.62V, ioh=1 mA (SDO, SDI)	0.8 * V_{DDIO}			V
Output – high level	Voh_1.2	V_{DDIO}=1.2V, ioh=1 mA (SDO, SDI)	0.6 * V_{DDIO}			V
Pull-up resistor	Rpull	Internal pull-up resistance to V_{DDIO}	70	120	190	kΩ
I^2C bus load capacitor	Cb	On SDI and SCK			400	pF

5.4.2 I²C timings

For I²C timings, the following abbreviations are used:

- “S&F mode” = standard and fast mode
- “HS mode” = high speed mode
- Cb = bus capacitance on SDA line

All other naming refers to I²C specification 2.1 (January 2000).

The I²C timing diagram is shown in
Figure 12. The corresponding values are given in Table 27.

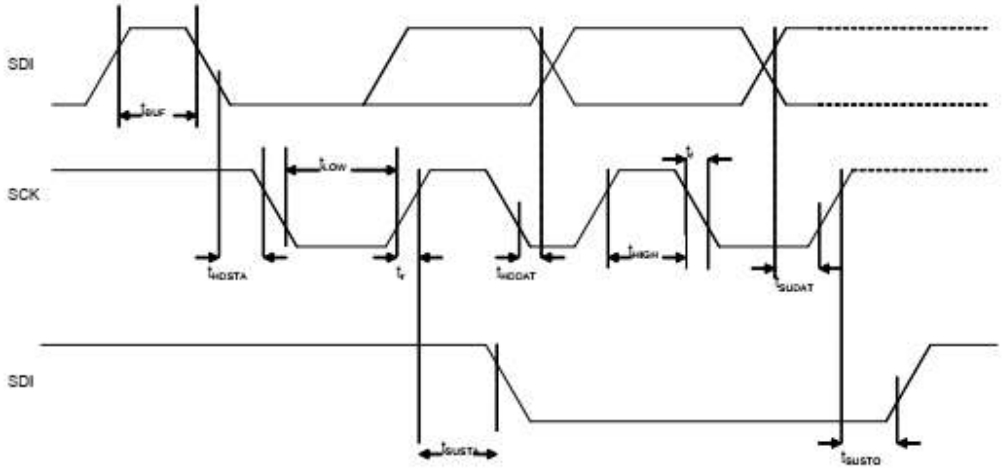


Figure 12: I²C timing diagram

Table 27: I²C timings

Parameter	Symbol	Condition	Min	Typ	Max	Units
SDI setup time	$t_{SU;DAT}$	S&F Mode HS mode	160 30			ns ns
SDI hold time	$t_{HD;DAT}$	S&F Mode, Cb≤100 pF S&F Mode, Cb≤400 pF HS mode, Cb≤100 pF HS mode, Cb≤400 pF	80 90 18 24		 115 150	ns ns ns ns
SCK low pulse	t_{LOW}	HS mode, Cb≤100 pF V_{DDIO} = 1.62 V	160			ns
SCK low pulse	t_{LOW}	HS mode, Cb≤100 pF V_{DDIO} = 1.2 V	210			ns

The above-mentioned I2C specific timings correspond to the following internal added delays:

- Input delay between SDI and SCK inputs: SDI is more delayed than SCK by typically 100 ns in Standard and Fast Modes and by typically 20 ns in High Speed Mode.
- Output delay from SCK falling edge to SDI output propagation is typically 140 ns in Standard and Fast Modes and typically 70 ns in High Speed Mode.

5.4.3 SPI timings

The SPI timing diagram is in Figure 13, while the corresponding values are given in Table 28. All timings apply both to 4- and 3-wire SPI.

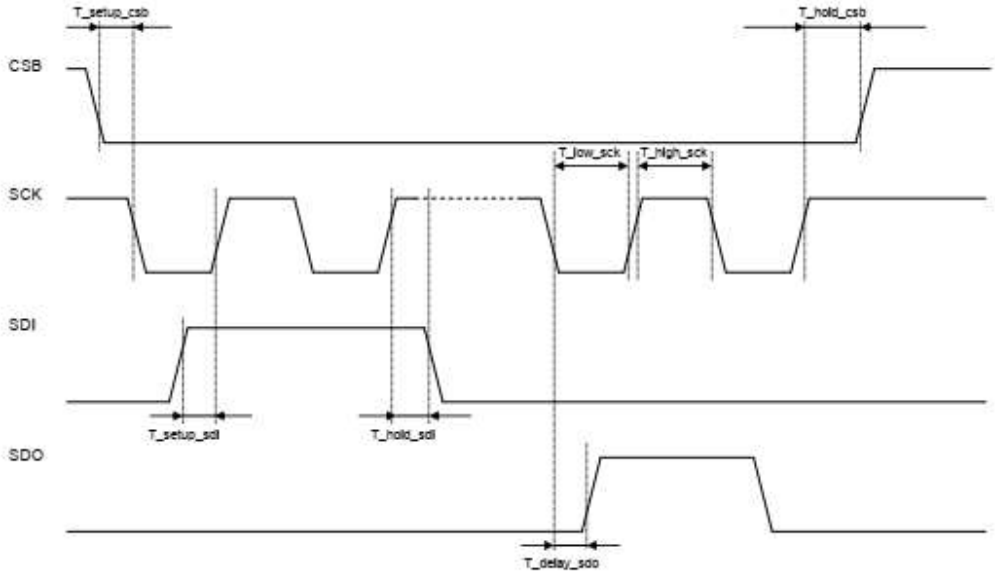


Figure 13: SPI timing diagram

Table 28: SPI timings

Parameter	Symbol	Condition	Min	Typ	Max	Units
SPI clock input frequency	F_spi		0		10	MHz
SCK low pulse	T_low_sck		20			ns
SCK high pulse	T_high_sck		20			ns
SDI setup time	T_setup_sdi		20			ns
SDI hold time	T_hold_sdi		20			ns
SDO output delay	T_delay_sdo	25pF load, V_{DDIO}=1.6V min			30	ns
SDO output delay	T_delay_sdo	25pF load, V_{DDIO}=1.2V min			40	ns
CSB setup time	T_setup_csb		20			ns
CSB hold time	T_hold_csb		20			ns

6. Pin-out and connection diagram

6.1 Pin-out

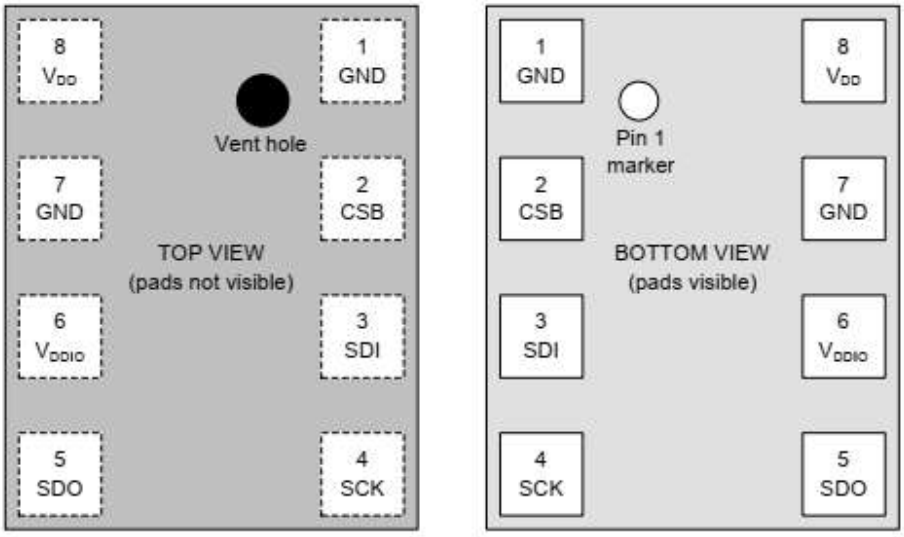


Figure 14: Pin-out top and bottom view

Table 29: Pin description

Pin	Name	I/O Type	Description	Connect to		
				SPI 4W	SPI 3W	I²C
1	GND	Supply	Ground	GND		
2	CSB	In	Chip select	CSB	CSB	V_{DDIO}
3	SDI	In/Out	Serial data input	SDI	SDI/SDO	SDA
4	SCK	In	Serial clock input	SCK	SCK	SCL
5	SDO	In/Out	Serial data output	SDO	DNC	GND for default address
6	V_{DDIO}	Supply	Digital interface supply	V_{DDIO}		
7	GND	Supply	Ground	GND		
8	V_{DD}	Supply	Analog supply	V_{DD}		

6.2 Connection diagram 4-wire SPI

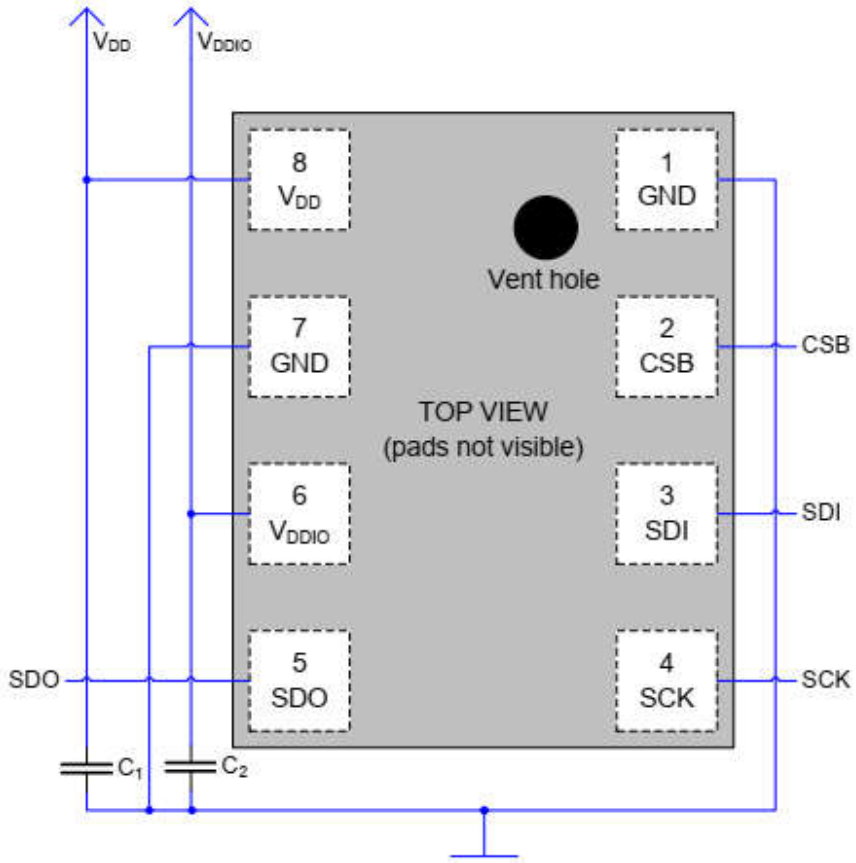


Figure 15: 4-wire SPI connection diagram (Pin1 marking indicated)

Note: the recommended value for C_1, C_2 is 100 nF.

6.3 Connection diagram 3-wire SPI

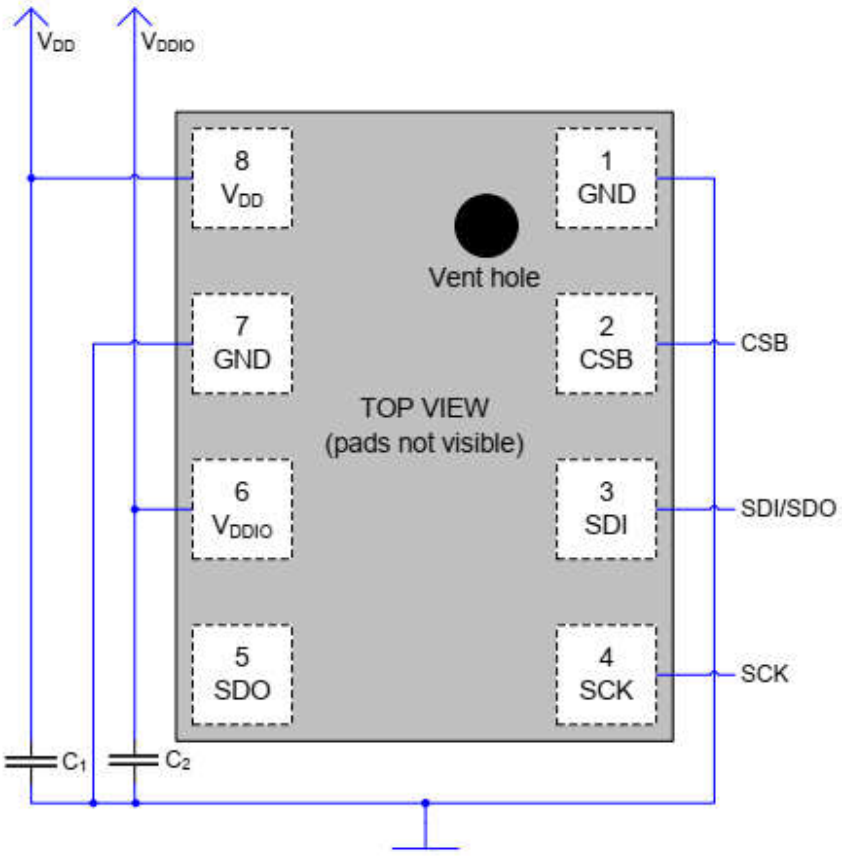


Figure 16: 3-wire SPI connection diagram (Pin1 marking indicated)

Note: the recommended value for C_1, C_2 is 100 nF.

6.4 Connection diagram I²C

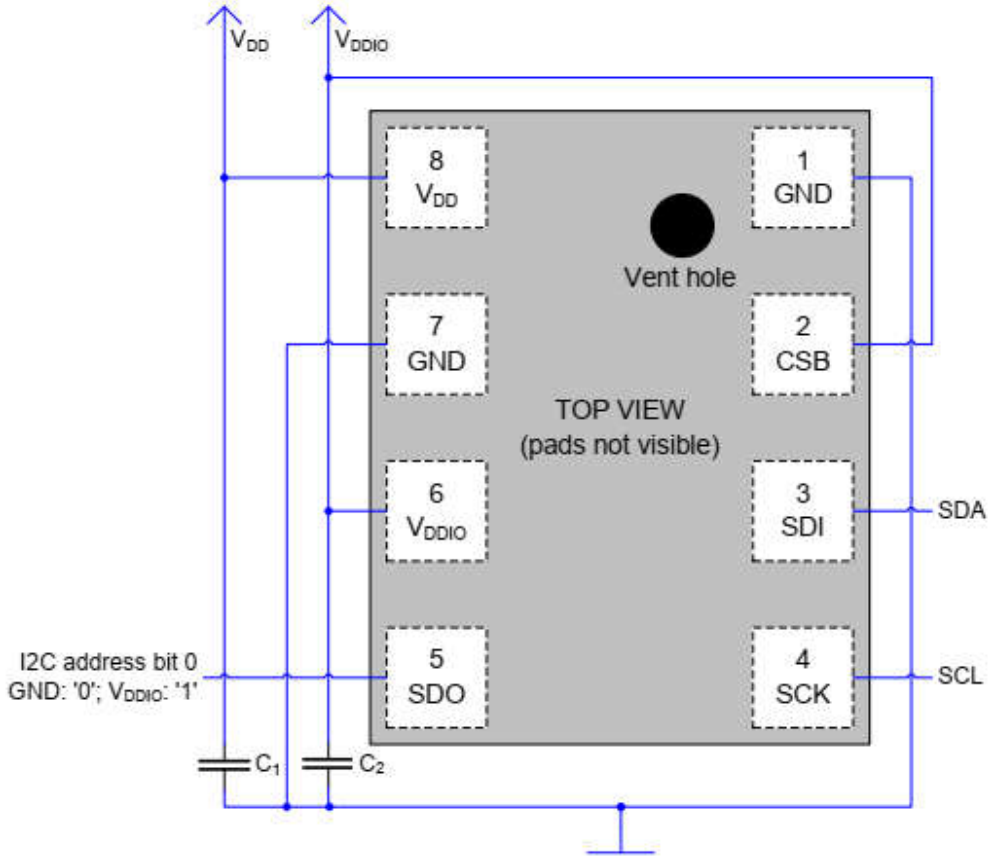


Figure 17: I²C connection diagram (Pin1 marking indicated)

Notes:

- The recommended value for C_1, C_2 is 100 nF.
- A direct connection between CSB and V_{DDIO} is recommended. If CSB is detected as low during startup, the interface will be locked into SPI mode. See chapter 5.1.

7. Package, reel and environment

7.1 Outline dimensions

The sensor housing is an 8-pin metal-lid LGA 2.0 × 2.5× 0.95 mm³ package. Its dimensions are depicted in Figure 18.

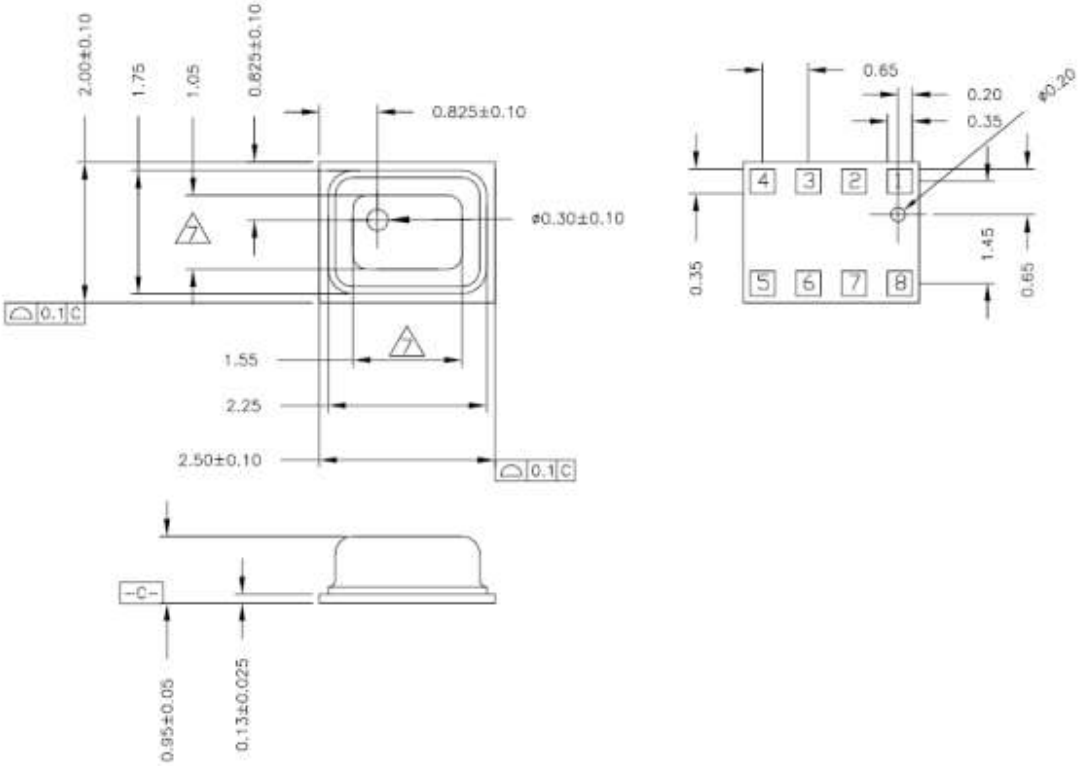


Figure 18: Package outline dimensions for top, bottom and side view

Note: General tolerances are ±50 µm (linear) and ±1° µm (angular)

7.2 Landing pattern recommendation

For the design of the landing pattern, the following dimensioning is recommended:

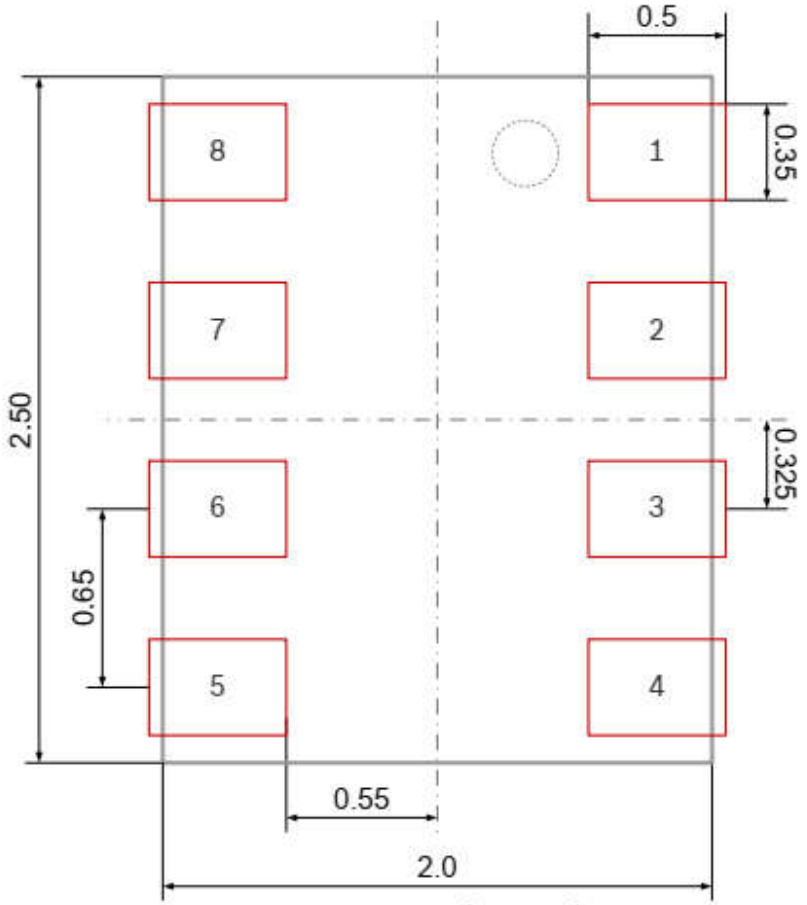


Figure 19: Recommended landing pattern (top view); dimensions are in mm

Note: red areas demark exposed PCB metal pads.

- In case of a solder mask defined (SMD) PCB process, the land dimensions should be defined by solder mask openings. The underlying metal pads are larger than these openings.
- In case of a non solder mask defined (NSMD) PCB process, the land dimensions should be defined in the metal layer. The mask openings are larger than the these metal pads.

7.3 Marking

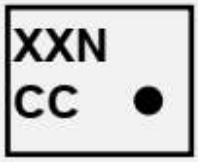
7.3.1 Mass production devices

Table 30: Marking of mass production samples

Labeling	Name	Symbol	Remark
	Lot counter	CCC	3 alphanumeric digits, variable to generate mass production trace-code
	Product number	T	1 alphanumeric digit, fixed to identify product type, T = "K" "K" is associated with the product BMP280 (part number 0 273 300 354)
	Sub-con ID	L	1 alphanumeric digit, variable to entify sub-con (L = "P", L = "U", L = "N"or L = "W")
	Orientation marker	●	Vent hole

7.3.2 Engineering samples

Table 31: Marking of engineering samples

Labeling	Name	Symbol	Remark
	Eng. Sample ID	N	1 alphanumeric digit, fixed to identify engineering sample, N = " * " or "e" or "E"
	Sample ID	XX	2 alphanumeric digits, variable to generate trace-code
	Counter ID	CC	2 alphanumeric digits, variable to generate trace-code
	Orientation marker	●	Vent hole

7.4 Soldering guidelines

The moisture sensitivity level of the BMP280 sensors corresponds to JEDEC Level 1, see also:

- IPC/JEDEC J-STD-020C "Joint Industry Standard: Moisture/Reflow Sensitivity Classification for non-hermetic Solid State Surface Mount Devices"
- IPC/JEDEC J-STD-033A "Joint Industry Standard: Handling, Packing, Shipping and Use of Moisture/Reflow Sensitive Surface Mount Devices".

The sensor fulfils the lead-free soldering requirements of the above-mentioned IPC/JEDEC standard, i.e. reflow soldering with a peak temperature up to 260°C. The minimum height of the solder after reflow shall be at least 50µm. This is required for good mechanical decoupling between the sensor device and the printed circuit board (PCB).

Profile Feature		Pb-Free Assembly
Average Ramp-Up Rate (Ts_{max} to Tp)		3° C/second max.
Preheat – Temperature Min (Ts_{min}) – Temperature Max (Ts_{max}) – Time (ts_{min} to ts_{max})		150 °C 200 °C 60-180 seconds
Time maintained above: – Temperature (T_L) – Time (t_L)		217 °C 60-150 seconds
Peak/Classification Temperature (Tp)		260 °C
Time within 5 °C of actual Peak Temperature (tp)		20-40 seconds
Ramp-Down Rate		6 °C/second max.
Time 25 °C to Peak Temperature		8 minutes max.

Note 1: All temperatures refer to topside of the package, measured on the package body surface.

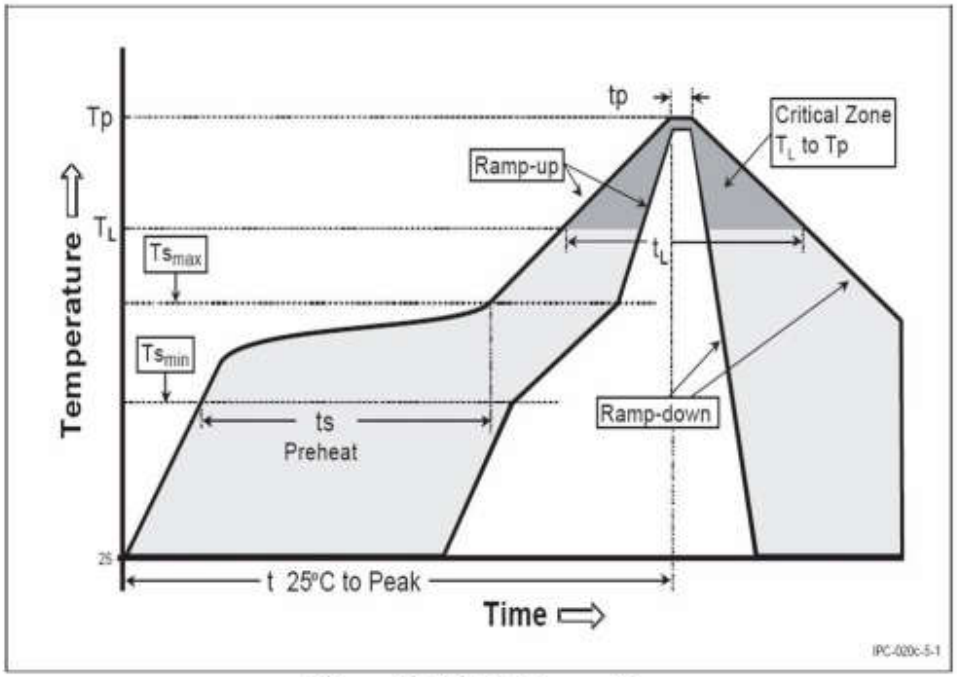


Figure 20: Soldering profile

7.5 Tape and reel specification

7.5.1 Dimensions

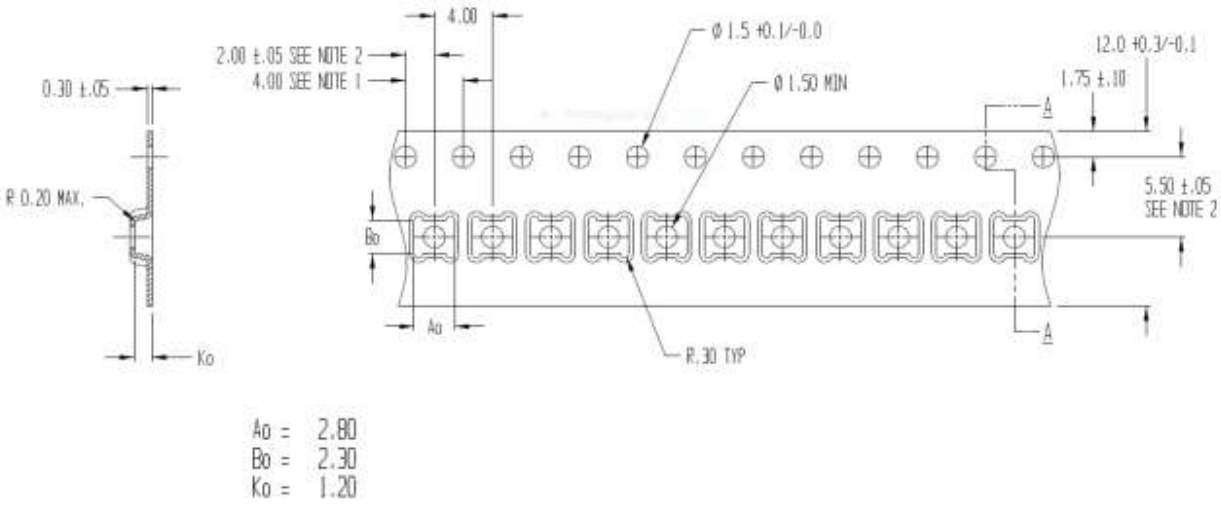


Figure 21: Tape and Reel dimensions

Quantity per reel: 10 kpcs.

7.5.2 Orientation within the reel

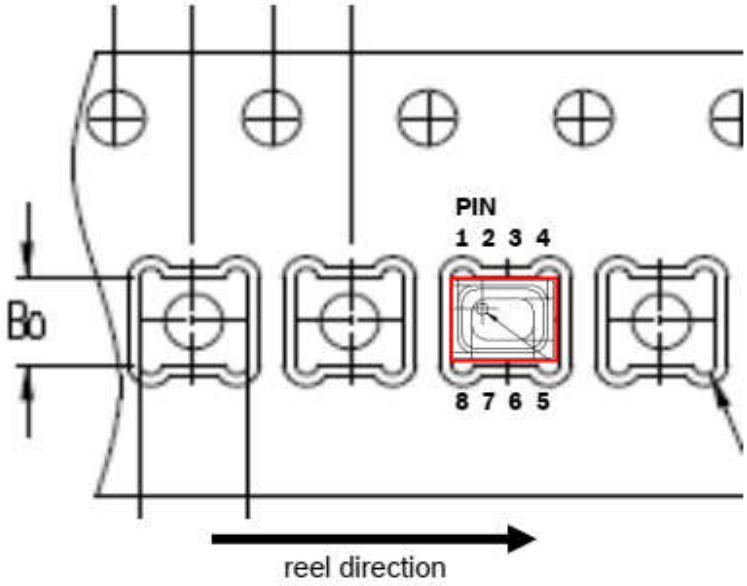


Figure 22: Orientation within tape

7.6 Mounting and assembly recommendations

In addition to "Handling, soldering & mounting instructions BMP280", the following recommendations should be taken into consideration when mounting a pressure sensor on a printed-circuit board (PCB):

- The clearance above the metal lid shall be 0.1mm at minimum.
- For the device housing appropriate venting needs to be provided in case the ambient pressure shall be measured.
- Liquids shall not come into direct contact with the device.
- During operation the sensor chip is sensitive to light, which can influence the accuracy of the measurement (photo-current of silicon). The position of the vent hole minimizes the light exposure of the sensor chip. Nevertheless, BST recommends to avoid the exposure of BMP280 to strong light sources.
- Soldering may not be done using vapor phase processes since the sensor might be damaged.

7.7 Environmental safety

7.7.1 RoHS

The BMP280 sensor meets the requirements of the EC restriction of hazardous substances (RoHS) directive, see also:

Directive 2011/65/EU of the European Parliament and of the Council of 8 June 2011 on the restriction of the use of certain hazardous substances in electrical and electronic equipment.

7.7.2 Halogen content

The BMP280 is halogen-free. For more details on the analysis results please contact your Bosch Sensortec representative.

7.7.3 Internal package structure

Within the scope of Bosch Sensortec's ambition to improve its products and secure the mass product supply, Bosch Sensortec qualifies additional sources (e.g. 2nd source) for the LGA package of the BMP280.

While Bosch Sensortec took care that all of the technical packages parameters are described above are 100% identical for all sources, there can be differences in the chemical content and the internal structural between the different package sources.

However, as secured by the extensive product qualification process of Bosch Sensortec, this has no impact to the usage or to the quality of the BMP280 product.

8. Appendix 1: Computation formulae for 32 bit systems

8.1 Compensation formula in floating point

Please note that it is strongly advised to use the API available from Bosch Sensortec to perform readout and compensation. If this is not wanted, the code below can be applied at the user's risk. Both pressure and temperature values are expected to be received in 20 bit format, positive, stored in a 32 bit signed integer.

The variable t_fine (signed 32 bit) carries a fine resolution temperature value over to the pressure compensation formula and could be implemented as a global variable.

The data type “BMP280_S32_t” should define a 32 bit signed integer variable type and could usually be defined as “long signed int”. The revision of the code is rev.1.1.

```
// Returns temperature in DegC, double precision. Output value of "51.23" equals 51.23 DegC.
// t_fine carries fine temperature as global value
BMP280_S32_t t_fine;
double bmp280_compensate_T_double(BMP280_S32_t adc_T)
{
    double var1, var2, T;
    var1  = (((double)adc_T)/16384.0 - ((double)dig_T1)/1024.0) * ((double)dig_T2);
    var2  = ((((double)adc_T)/131072.0 - ((double)dig_T1)/8192.0) *
        (((double)adc_T)/131072.0 - ((double) dig_T1)/8192.0)) * ((double)dig_T3);
    t_fine = (BMP280_S32_t)(var1 + var2);
    T  = (var1 + var2) / 5120.0;
    return T;
}

// Returns pressure in Pa as double. Output value of "96386.2" equals 96386.2 Pa = 963.862 hPa
double bmp280_compensate_P_double(BMP280_S32_t adc_P)
{
    double var1, var2, p;
    var1 = ((double)t_fine/2.0) - 64000.0;
    var2 = var1 * var1 * ((double)dig_P6) / 32768.0;
    var2 = var2 + var1 * ((double)dig_P5) * 2.0;
    var2 = (var2/4.0)+(((double)dig_P4) * 65536.0);
    var1 = (((double)dig_P3) * var1 * var1 / 524288.0 + ((double)dig_P2) * var1) / 524288.0;
    var1 = (1.0 + var1 / 32768.0)*((double)dig_P1);
    if (var1 == 0.0)
    {
        return 0; // avoid exception caused by division by zero
    }
    p = 1048576.0 - (double)adc_P;
    p = (p - (var2 / 4096.0)) * 6250.0 / var1;
    var1 = ((double)dig_P9) * p * p / 2147483648.0;
    var2 = p * ((double)dig_P8) / 32768.0;
    p = p + (var1 + var2 + ((double)dig_P7)) / 16.0;
    return p;
}
```

8.2 Compensation formula in 32 bit fixed point

Please note that it is strongly advised to use the API available from Bosch Sensortec to perform readout and compensation. If this is not wanted, the code below can be applied at the user’s risk. Both pressure and temperature values are expected to be received in 20 bit format, positive, stored in a 32 bit signed integer.

The variable t_fine (signed 32 bit) carries a fine resolution temperature value over to the pressure compensation formula and could be implemented as a global variable.
The data type “BMP280_S32_t” should define a 32 bit signed integer variable type and can usually be defined as “long signed int”.

The data type “BMP280_U32_t” should define a 32 bit unsigned integer variable type and can usually be defined as “long unsigned int”.
Compensating the pressure value with 32 bit integer has an accuracy of typically 1 Pa (1-sigma). At very high filter levels this adds a noticeable amount of noise to the output values and reduces their resolution.

```
// Returns temperature in DegC, resolution is 0.01 DegC. Output value of "5123" equals 51.23 DegC.
// t_fine carries fine temperature as global value
BMP280_S32_t t_fine;
BMP280_S32_t bmp280_compensate_T_int32(BMP280_S32_t adc_T)
{
    BMP280_S32_t var1, var2, T;
```

```
    var1  = ((((adc_T>>3) - ((BMP280_S32_t)dig_T1<<1))) * ((BMP280_S32_t)dig_T2)) >> 11;
    var2  = (((((adc_T>>4) - ((BMP280_S32_t)dig_T1)) * ((adc_T>>4) - ((BMP280_S32_t)dig_T1))) >> 12) *
        ((BMP280_S32_t)dig_T3)) >> 14;
    t_fine = var1 + var2;
    T  = (t_fine * 5 + 128) >> 8;
    return T;
}

// Returns pressure in Pa as unsigned 32 bit integer. Output value of "96386" equals 96386 Pa = 963.86 hPa
BMP280_U32_t bmp280_compensate_P_int32(BMP280_S32_t adc_P)
{
    BMP280_S32_t var1, var2;
    BMP280_U32_t p;
    var1 = (((BMP280_S32_t)t_fine)>>1) - (BMP280_S32_t)64000;
    var2 = (((var1>>2) * (var1>>2)) >> 11 ) * ((BMP280_S32_t)dig_P6);
    var2 = var2 + ((var1*((BMP280_S32_t)dig_P5))<<1);
    var2 = (var2>>2)+(((BMP280_S32_t)dig_P4)<<16);
    var1 = (((dig_P3 * (((var1>>2) * (var1>>2)) >> 13 )) >> 3) + ((((BMP280_S32_t)dig_P2) * var1)>>1))>>18;
    var1 =((((32768+var1))*((BMP280_S32_t)dig_P1))>>15);
    if (var1 == 0)
    {
        return 0; // avoid exception caused by division by zero
    }
    p = (((BMP280_U32_t)(((BMP280_S32_t)1048576)-adc_P)-(var2>>12)))*3125;
    if (p < 0x80000000)
    {
        p = (p << 1) / ((BMP280_U32_t)var1);
    }
    else
    {
        p = (p / (BMP280_U32_t)var1) * 2;
    }
    var1 = (((BMP280_S32_t)dig_P9) * ((BMP280_S32_t)(((p>>3) * (p>>3))>>13)))>>12;
    var2 = (((BMP280_S32_t)(p>>2)) * ((BMP280_S32_t)dig_P8))>>13;
    p = (BMP280_U32_t)((BMP280_S32_t)p + ((var1 + var2 + dig_P7) >> 4));
    return p;
}
```

9. Legal disclaimer

9.1 Engineering samples

Engineering Samples are marked with an asterisk (*) or (e) or (E). Samples may vary from the valid technical specifications of the product series contained in this data sheet. They are therefore not intended or fit for resale to third parties or for use in end products. Their sole purpose is internal client testing. The testing of an engineering sample may in no way replace the testing of a product series. Bosch Sensortec assumes no liability for the use of engineering samples. The Purchaser shall indemnify Bosch Sensortec from all claims arising from the use of engineering samples.

9.2 Product use

Bosch Sensortec products are developed for the consumer goods industry. They are not designed or approved for use in military applications, life-support appliances, safety-critical automotive applications and devices or systems where malfunctions of these products can reasonably be expected to result in personal injury. They may only be used within the parameters of this product data sheet.

The resale and/or use of products are at the Purchaser's own risk and the Purchaser's own responsibility.

The Purchaser shall indemnify Bosch Sensortec from all third party claims arising from any product use not covered by the parameters of this product data sheet or not approved by Bosch Sensortec and reimburse Bosch Sensortec for all costs in connection with such claims.

The Purchaser accepts the responsibility to monitor the market for the purchased products, particularly with regard to product safety, and inform Bosch Sensortec without delay of any security relevant incidents.

9.3 Application examples and hints

With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Bosch Sensortec hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights or copyrights of any third party. The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. They are provided for illustrative purposes only and no evaluation regarding infringement of intellectual property rights or copyrights or regarding functionality, performance or error has been made.

10. Document history and modification

Rev. No	Chapter	Description of modification/changes	Date
0.1		Document creation	2012-08-06
1.0	9.2	Change of product use	2013-11-26
	Table 2	Update of min/max data (only for restricted version)	
		Added comment on the sampling rate	
1.1	1, 3.3.1	Changed value for resolution, values for *osrs_p* settings changed	2014-02-10
	5.2	Changed sentence and added drawing	2014-02-18
	3.7	Added max values for current consumption	2014-05-08
1.11	4.5.3	Modified write in normal mode	2014-06-25
	5.2	Modified SDI/SCK ESD drawing	
1.12	1	Changed min/max values for standby current, only valid for 25 °C	2014-07-12
	Table 1	Pressure resolution 0.16Pa	2014-07-12
1.13	Page 2	New technical reference codes added	2014-11-12
	7.3	New details about laser marking added	
1.14	Table 6	Changed contents of table	2015-05-04
	Page 1	Removed TRC 0 273 300 354 & 0273 300 391	
	Page 44	Updated RoHS directive to 2011/65/EU effective 8 June 2011	2015-05-07

Bosch Sensortec GmbH
Gerhard-Kindler-Strasse 8
72770 Reutlingen / Germany

contact@bosch-sensortec.com
www.bosch-sensortec.com

Modifications reserved | Printed in Germany
Specifications subject to change without notice
Document number: BST-BMP280-DS001-11
Revision_1.14_052015

Universidad del País Vasco | Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO FIN DE GRADO

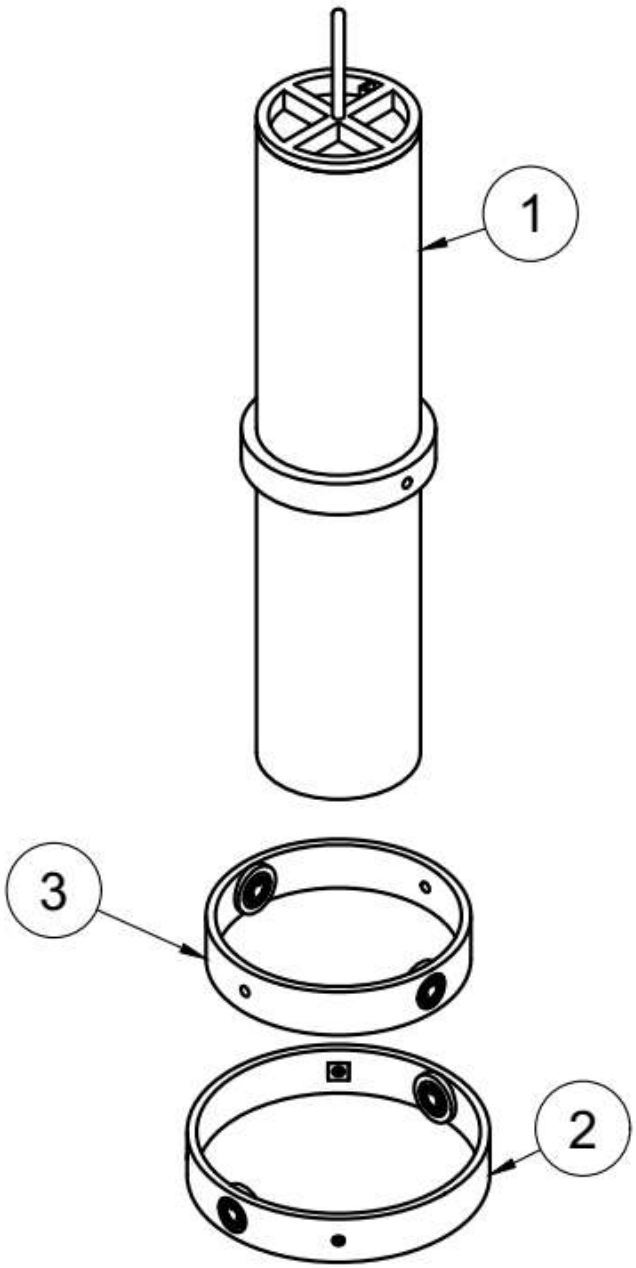
DISEÑO E IMPLEMENTACIÓN DE SISTEMAS ELECTRÓNICOS PARA MODELOS DE COHETE USADOS EN COHETERÍA RECREATIVA

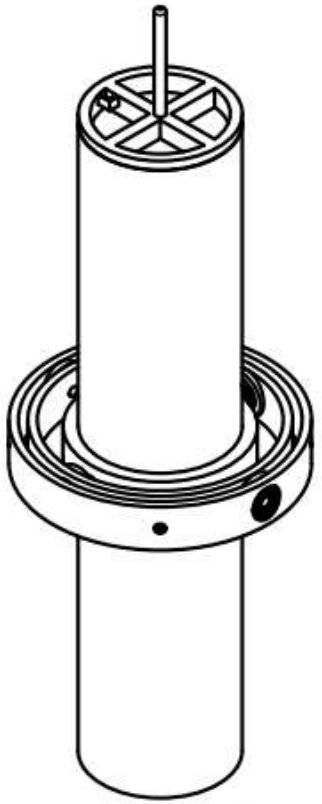
ANEXO 2- PLANOS DE DISEÑO

Alumno/Alumna: FEIJOO ALONSO, ANDER
Director/Directora (1): SAINZ DE MURIETA MANGADO, JOSEBA ANDONI

Curso: 2018-2019

Fecha: 19/07/2019





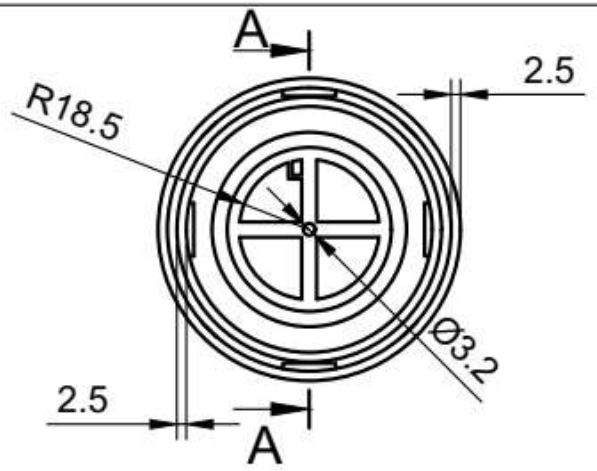
Item	Part Name
3	Interior
2	Exterior
1	Motor

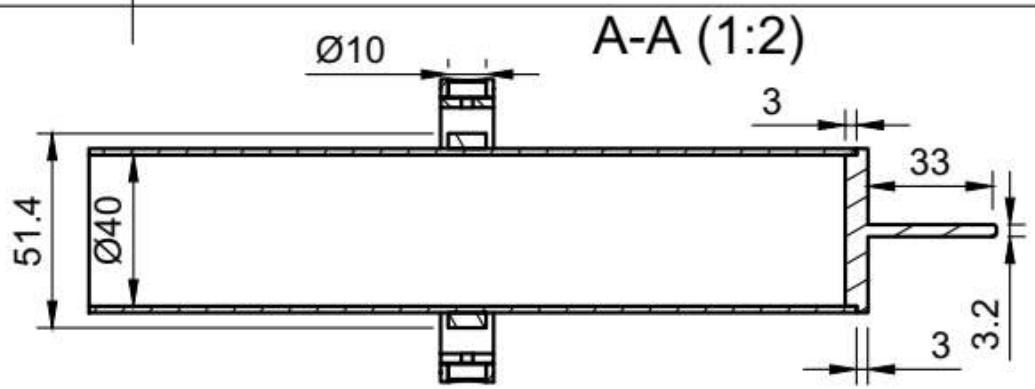
Parts List

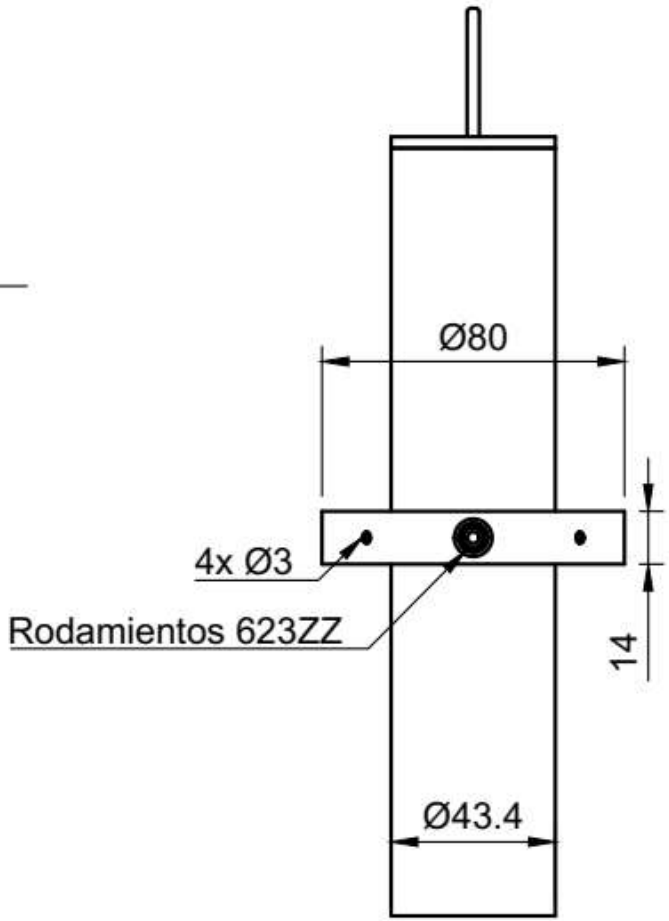
Escala 1:2

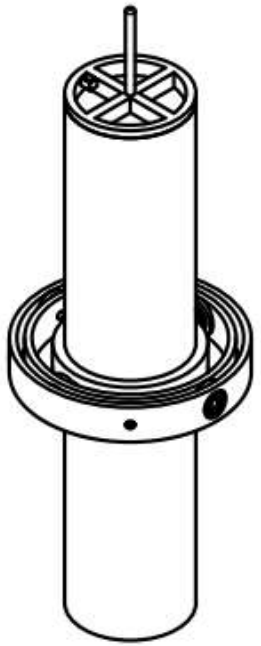
Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Despiece
Title Rocket DEF1 Gimbal	EHU

Sheet 1/2



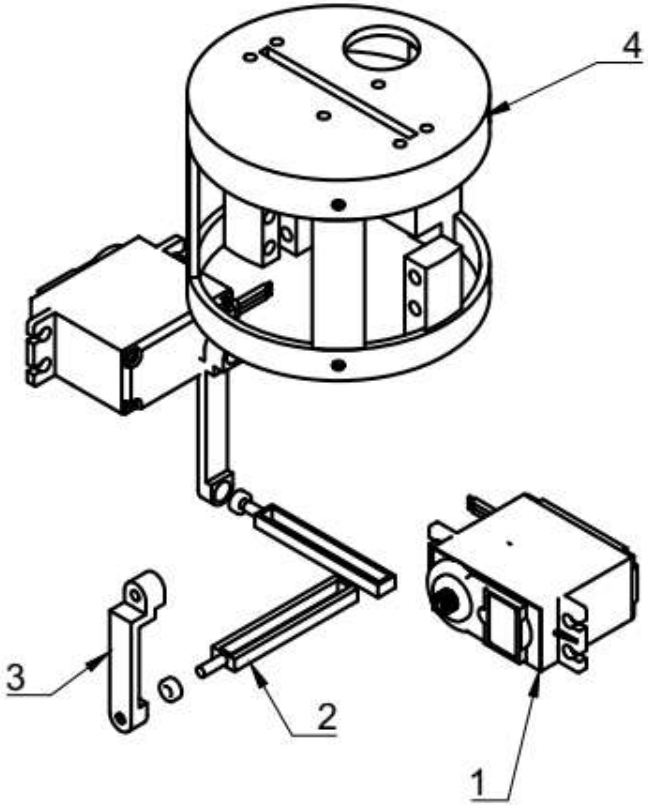


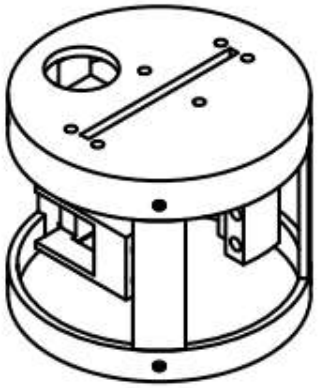




Escala 1:2

Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Diseño
Title Rocket DEF1 Gimbal	EHU Sheet 2/2





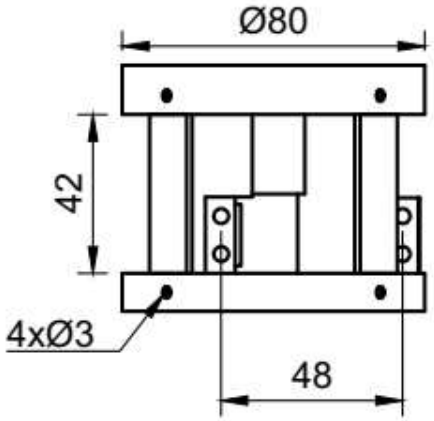
Item	Part Number
4	Base
3	Brazo X e Y
2	Joy X e Y
1	Servo X e Y

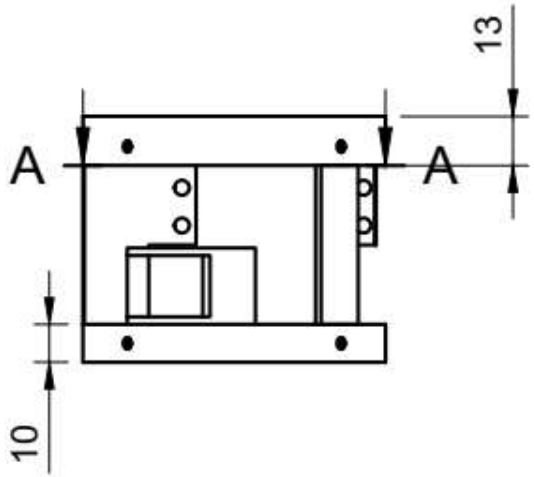
Parts List

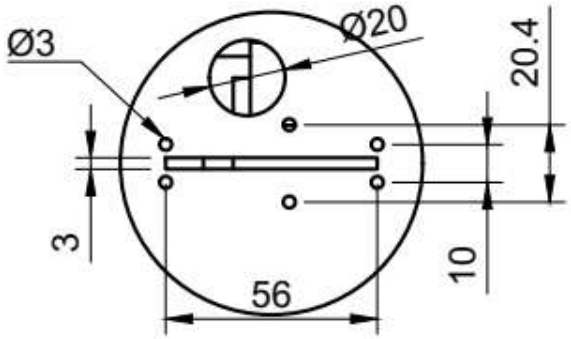
Escala 1:2

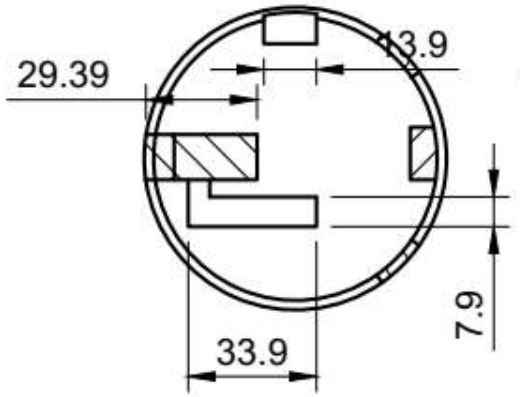
Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Despiece
Title Rocket DEF1 Joystick	EHU

Sheet 1/5





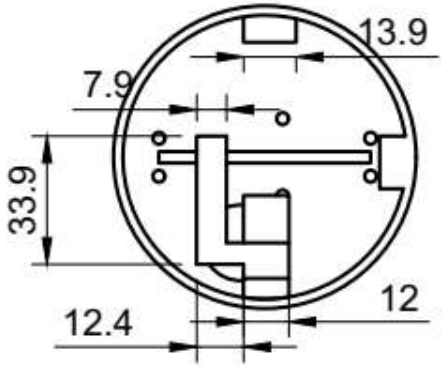


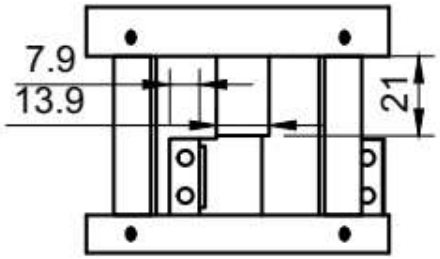


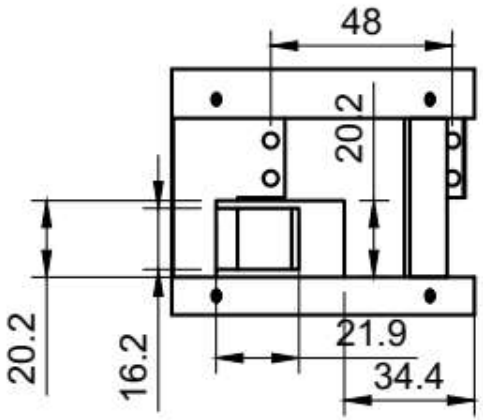
A-A (1:2)

Escala 1:2

Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Diseño
Title Rocket DEF1 Joystick	EHU Sheet 2/5

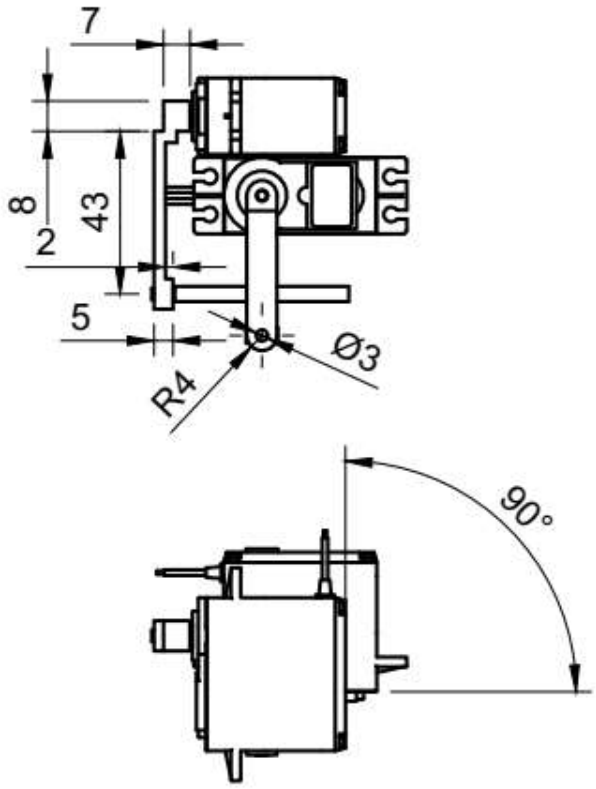


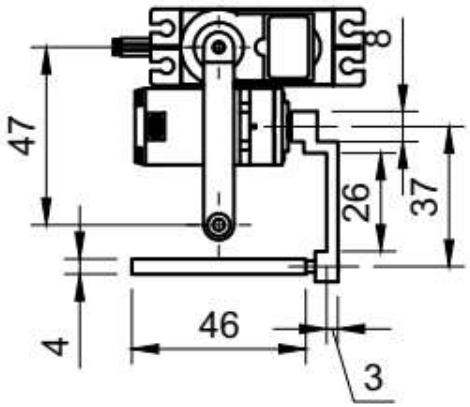




Escala 1:2

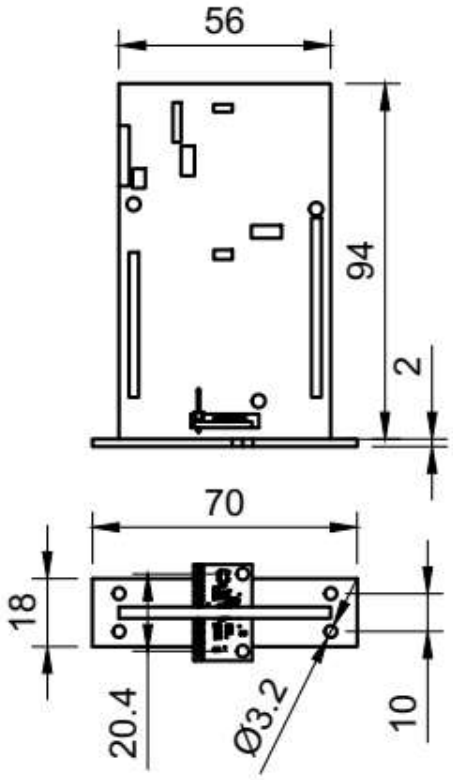
Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Diseño
Title Rocket DEF1 Joystick	EHU Sheet 3/5

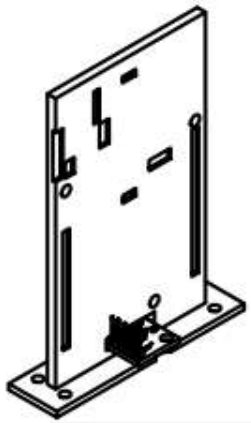




Escala 1:2

Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Diseño
Title Rocket DEF1 Joystick	EHU Sheet 4/5





Escala 1:2

Created by Ander Feijoo	17/07/2019
Document type TFG	Document status Diseño
Title Rocket DEF1 CPU	EHU Sheet 5/5

Universidad del País Vasco · Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE SISTEMAS ELECTRÓNICOS PARA MODELOS DE COHETE USADOS EN COHETERÍA RECREATIVA

ANEXO 3- CÓDIGO DE PROGRAMACIÓN

Alumno/Alumna: FEIJOO ALONSO, ANDER
Director/Directora (1): SAINZ DE MURIETA MANGADO, JOSEBA ANDONI

Curso: 2018-2019

Fecha: 19/07/2019

Índice de ANEXO 3

1. ESTRUCTURA FIRMWARE

El firmware tiene una estructura de archivos y ‹carpetas› de la siguiente manera:

+---‹FirmwareCohete›
| \---‹ClaseMPU9250›
| | MPU9250.cpp
| | MPU9250.h
| | register.h
| | MainMPU9250.cpp
| \---‹ClaseBMP280›
| | BMP.cpp
| | BMP.h
| | MainBMP.cpp
| \---‹ClaseServo›
| | Servos.h
| \---‹ClaseFSM›
| | FSM.cpp
| | FSM.h
| \---‹PID›
| | PID.cpp
| | PID.h
| main.cpp

2. CÓDIGO ‹FirmwareCohete›

2.1. ‹ClaseMPU9250›

2.1.1. MPU9250.cpp

```
1 #include "MPU9250.h"
2
3 MPU9250::MPU9250(PinName sda, PinName scl, PinName tx, PinName rx) : i2c(sda,
   scl), pc(tx,rx){
4
5    i2c.frequency(400000);
6
7    Ascale = AFS_2G;    // AFS_2G, AFS_4G, AFS_8G, AFS_16G
8    Gscale = GFS_250DPS; // GFS_250DPS, GFS_500DPS, GFS_1000DPS,
   GFS_2000DPS
9    Mscale = MFS_16BITS;
10         Mmode = 0x06;      // Either 8 Hz 0x02) or 100 Hz (0x06) magnetometer data
   ODR
11
12         for(int i=0; i<=2; i++){
13           magCalibration[i] = 0;
14           magbias[i] = 0;
15           gyroBias[i] = 0;
16           accelBias[i] = 0;
17           magScale[i]=0;
18         }
19
20
21         //Filtros
22
23         GyroMeasError = PI * (60.0f / 180.0f);    // gyroscope measurement error in
   rads/s (start at 60 deg/s), then reduce after ~10 s to 3
24         beta = sqrt(3.0f / 4.0f) * GyroMeasError;  // compute beta
25         GyroMeasDrift = PI * (1.0f / 180.0f);     // gyroscope measurement drift in
   rad/s/s (start at 0.0 deg/s/s)
26         zeta = sqrt(3.0f / 4.0f) * GyroMeasDrift;  // compute zeta, the other free
   parameter in the Madgwick scheme usually set to a small or zero value
27
28         lastUpdate=0;
29         firstUpdate=0;
30         Now=0;
31
```

```
32          deltat=0.0f;
33
34          q[0]=1.0f;q[1]=0.0f;q[2]=0.0f;q[3]=0.0f;
35
36          eInt[0]=0.0f; eInt[1]=0.0f; eInt[2]=0.0f;
37
38      }
39
40      void MPU9250::start(){
41
42          t.start();
43
44          //Biases
45          gyroBias[0]=0.938931;
46          gyroBias[1]=-0.870229;
47          gyroBias[2]=0.389313;
48          accelBias[0]=-0.138550;
49          accelBias[1]=-0.456299;
50          accelBias[2]=0.004150;
51
52          whoami = readByte(MPU9250_ADDRESS, WHO_AM_I_MPU9250);  // Read
    WHO_AM_I register for MPU-9250
53          pc.printf("I AM 0x%x\n\r", whoami);
54
55          if (whoami == 0x71 || whoami == 0x73 || whoami == 0x68) { // WHO_AM_I
    should always be 0x68
56              pc.printf("MPU9250 WHO_AM_I is 0x%x is online\n\r", whoami);
57              wait(1);
58
59              //resetMPU9250(); // Reset registers to default in preparation for device
    calibration
60              //MPU9250SelfTest(SelfTest); // Start by performing self test and reporting
    values
61              /*
62              pc.printf("x-axis self test: acceleration trim within : %f % of factory value\n\
    r", SelfTest[0]);
63              pc.printf("y-axis self test: acceleration trim within : %f % of factory value\n\
    r", SelfTest[1]);
64              pc.printf("z-axis self test: acceleration trim within : %f % of factory value\n\
    r", SelfTest[2]);
65              pc.printf("x-axis self test: gyration trim within : %f % of factory value\n\r",
    SelfTest[3]);
66              pc.printf("y-axis self test: gyration trim within : %f % of factory value\n\r",
    SelfTest[4]);
67              pc.printf("z-axis self test: gyration trim within : %f % of factory value\n\r",
    SelfTest[5]);
```

```
68          */
69          //calibrateMPU9250(gyroBias, accelBias); // Calibrate gyro and
  accelerometers, load biases in bias registers
70          /*
71          pc.printf("x gyro bias = %f\n\r", gyroBias[0]);
72          pc.printf("y gyro bias = %f\n\r", gyroBias[1]);
73          pc.printf("z gyro bias = %f\n\r", gyroBias[2]);
74          pc.printf("x accel bias = %f\n\r", accelBias[0]);
75          pc.printf("y accel bias = %f\n\r", accelBias[1]);
76          pc.printf("z accel bias = %f\n\r", accelBias[2]);
77          */
78          wait(2);
79          initMPU9250();
80          pc.printf("MPU9250 initialized for active data mode....\n\r"); // Initialize
  device for active mode read of acclerometer, gyroscope, and temperature
81          initAK8963(magCalibration);
82          pc.printf("AK8963 initialized for active data mode....\n\r"); // Initialize
  device for active mode read of magnetometer
83
84          /*
85          pc.printf("Accelerometer full-scale range = %f  g\n\r", 2.0f*(float)
  (1<<Ascale));
86          pc.printf("Gyroscope full-scale range = %f  deg/s\n\r", 250.0f*(float)
  (1<<Gscale));
87          if(Mscale == 0) pc.printf("Magnetometer resolution = 14  bits\n\r");
88          if(Mscale == 1) pc.printf("Magnetometer resolution = 16  bits\n\r");
89          if(Mmode == 2) pc.printf("Magnetometer ODR = 8 Hz\n\r");
90          if(Mmode == 6) pc.printf("Magnetometer ODR = 100 Hz\n\r");
91          */
92          wait(1);
93        } else {
94          pc.printf("Could not connect to MPU9250: \n\r");
95          pc.printf("%#x \n",  whoami);
96
97          while(1) ; // Loop forever if communication doesn't happen
98        }
99
100
101       getAres(); // Get accelerometer sensitivity
102       getGres(); // Get gyro sensitivity
103       getMres(); // Get magnetometer sensitivity
104       /*
105       pc.printf("Accelerometer sensitivity is %f LSB/g \n\r", 1.0f/aRes);
106       pc.printf("Gyroscope sensitivity is %f LSB/deg/s \n\r", 1.0f/gRes);
107       pc.printf("Magnetometer sensitivity is %f LSB/G \n\r", 1.0f/mRes);
108       */
```

```
109
110       //magcalMPU9250(magbias,magScale); //Para calibrar correctamente los
    magbias
111       //magbias[0] = +470.;  // User environmental x-axis correction in milliGauss,
    should be automatically calculated
112       //magbias[1] = +120.;  // User environmental x-axis correction in milliGauss
113       //magbias[2] = +125.;  // User environmental x-axis correction in milliGauss
114       magbias[0] = 214.809540;
115       magbias[1] = 232.447067;
116       magbias[2] = 279.946625;
117       magScale[0] = 1.113367;
118       magScale[1]= 0.937322;
119       magScale[2] = 0.966226;
120     }
121
122     void MPU9250::writeByte(uint8_t address, uint8_t subAddress, uint8_t data){
123
124       char data_write[2];
125       data_write[0] = subAddress;
126       data_write[1] = data;
127       i2c.write(address, data_write, 2, 0);
128     }
129
130     char MPU9250::readByte(uint8_t address, uint8_t subAddress){
131
132       char data[1]; // `data` will store the register data
133       char data_write[1];
134       data_write[0] = subAddress;
135       i2c.write(address, data_write, 1, 1); // no stop
136       i2c.read(address, data, 1, 0);
137       return data[0];
138     }
139
140     void MPU9250::readBytes(uint8_t address, uint8_t subAddress, uint8_t count,
    uint8_t * dest){
141
142       char data[14];
143       char data_write[1];
144       data_write[0] = subAddress;
145       i2c.write(address, data_write, 1, 1); // no stop
146       i2c.read(address, data, count, 0);
147       for(int ii = 0; ii < count; ii++) {
148         dest[ii] = data[ii];
149       }
150     }
151
```

```
152   void MPU9250::getAres(){
153
154     switch (Ascale){
155           // Possible accelerometer scales (and their register bit settings) are:
156           // 2 Gs (00), 4 Gs (01), 8 Gs (10), and 16 Gs  (11).
157           // Here's a bit of an algorith to calculate DPS/(ADC tick) based on that 2-
    bit value:
158         case AFS_2G:
159           aRes = 2.0/32768.0;
160           break;
161         case AFS_4G:
162           aRes = 4.0/32768.0;
163           break;
164         case AFS_8G:
165           aRes = 8.0/32768.0;
166           break;
167         case AFS_16G:
168           aRes = 16.0/32768.0;
169           break;
170     }
171   }
172
173   void MPU9250::getGres(){
174
175     switch (Gscale){
176           // Possible gyro scales (and their register bit settings) are:
177           // 250 DPS (00), 500 DPS (01), 1000 DPS (10), and 2000 DPS  (11).
178           // Here's a bit of an algorith to calculate DPS/(ADC tick) based on that 2-
    bit value:
179         case GFS_250DPS:
180           gRes = 250.0/32768.0;
181           break;
182         case GFS_500DPS:
183           gRes = 500.0/32768.0;
184           break;
185         case GFS_1000DPS:
186           gRes = 1000.0/32768.0;
187           break;
188         case GFS_2000DPS:
189           gRes = 2000.0/32768.0;
190           break;
191     }
192   }
193
194   void MPU9250::getMres(){
195
```

```
196       switch (Mscale){
197           // Possible magnetometer scales (and their register bit settings) are:
198           // 14 bit resolution (0) and 16 bit resolution (1)
199         case MFS_14BITS:
200           mRes = 10.0*4219.0/8190.0; // Proper scale to return milliGauss
201           break;
202         case MFS_16BITS:
203           mRes = 10.0*4219.0/32760.0; // Proper scale to return milliGauss
204           break;
205       }
206     }
207
208     void MPU9250::readAccelData(int16_t * destination){
209
210       uint8_t rawData[6];  // x/y/z accel register data stored here
211       readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);  // Read
    the six raw data registers into data array
212       destination[0] = (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;  // Turn the
    MSB and LSB into a signed 16-bit value
213       destination[1] = (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
214       destination[2] = (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
215
216     }
217
218     void MPU9250::readGyroData(int16_t * destination){
219
220       uint8_t rawData[6];  // x/y/z gyro register data stored here
221       readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);  // Read
    the six raw data registers sequentially into data array
222       destination[0] = (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;  // Turn the
    MSB and LSB into a signed 16-bit value
223       destination[1] = (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
224       destination[2] = (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
225
226     }
227
228     void MPU9250::readMagData(int16_t * destination){
229
230       uint8_t rawData[7];  // x/y/z gyro register data, ST2 register stored here, must
    read ST2 at end of data acquisition
231       if(readByte(AK8963_ADDRESS, AK8963_ST1) & 0x01) { // wait for
    magnetometer data ready bit to be set
232         readBytes(AK8963_ADDRESS, AK8963_XOUT_L, 7, &rawData[0]);  // Read
    the six raw data and ST2 registers sequentially into data array
233         uint8_t c = rawData[6]; // End data read by reading ST2 register
```

```
234         if(!(c & 0x08)) { // Check if magnetic sensor overflow set, if not then report
  data
235           destination[0] = (int16_t)(((int16_t)rawData[1] << 8) | rawData[0]);  // Turn
  the MSB and LSB into a signed 16-bit value
236           destination[1] = (int16_t)(((int16_t)rawData[3] << 8) | rawData[2]) ;  // Data
  stored as little Endian
237           destination[2] = (int16_t)(((int16_t)rawData[5] << 8) | rawData[4]) ;
238         }
239       }
240     }
241
242     void MPU9250::readTempData(){
243       int16_t destination;
244       uint8_t rawData[2];  // x/y/z gyro register data stored here
245       readBytes(MPU9250_ADDRESS, TEMP_OUT_H, 2, &rawData[0]);  // Read the
  two raw data registers sequentially into data array
246       destination = (int16_t)(((int16_t)rawData[0]) << 8 | rawData[1]) ;  // Turn the
  MSB and LSB into a 16-bit value
247       destination = ((float) destination) / 333.87f + 21.0f;
248       temperature = destination;
249     }
250
251     void MPU9250::ReadRawAccGyroMag(){
252       // If intPin goes high, all data registers have new data
253       readAccelData(accelCount);  // Read the x/y/z adc values
254       AccelXYZCal();
255
256       readGyroData(gyroCount);  // Read the x/y/z adc values
257       GyroXYZCal();
258
259       readMagData(magCount);  // Read the x/y/z adc values
260       MagXYZCal();
261     }
262
263     void MPU9250::YPR(){
264
265       Now = t.read_us();
266       deltat = (float)((Now - lastUpdate)/1000000.0f) ; // set integration time by
  time elapsed since last filter update
267       lastUpdate = Now;
268
269       MadgwickQuaternionUpdate(deltat, ax, ay, az, gx*PI/180.0f, gy*PI/180.0f,
  gz*PI/180.0f,  mx, mz, my);
270       //MahonyQuaternionUpdate(ax, ay, az, gx*PI/180.0f, gy*PI/180.0f,
  gz*PI/180.0f, my, mx, mz);
271       //IMUfilter(deltat,ax, ay, az, gx*PI/180.0f, gy*PI/180.0f, gz*PI/180.0f);
```

```
272
273       // yaw   = atan2(2.0f * (q[1] * q[2] + q[0] * q[3]), q[0] * q[0] + q[1] * q[1] - q[2] *
   q[2] - q[3] * q[3]);
274       // pitch = -asin(2.0f * (q[1] * q[3] - q[0] * q[2]));
275       //roll  = atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), q[0] * q[0] - q[1] * q[1] - q[2] * q[2]
   + q[3] * q[3]);
276
277       yaw   = atan2(2.0f * (q[1] * q[2] + q[0] * q[3]), 1.0f - 2.0f*(q[2]*q[2] + q[3]*q[3]));
278       pitch = asin(2.0f * (q[0] * q[2] - q[3] * q[1]));
279       roll  = atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), 1.0f - 2.0f*(q[1]*q[1] + q[2]*q[2]));
280
281
282       pitch *= 180.0f / PI;
283       yaw   *= 180.0f / PI;
284       yaw   += 0.2f; // Declination at Danville, California is 13 degrees 48 minutes
   and 47 seconds on 2014-04-04
285       roll  *= 180.0f / PI;
286
287       pc.printf("#YPR= %f %f %f \n\r", yaw, pitch, roll);
288     }
289
290     void MPU9250::AccelXYZCal(){
291       ax = (float)accelCount[0]*aRes - accelBias[0];  // get actual g value, this
   depends on scale being set
292       ay = (float)accelCount[1]*aRes - accelBias[1];
293       az = (float)accelCount[2]*aRes - accelBias[2];
294     }
295
296     void MPU9250::GyroXYZCal(){
297       gx = (float)gyroCount[0]*gRes - gyroBias[0];  // get actual gyro value, this
   depends on scale being set
298       gy = (float)gyroCount[1]*gRes - gyroBias[1];
299       gz = (float)gyroCount[2]*gRes - gyroBias[2];
300     }
301
302     void MPU9250::MagXYZCal(){
303
304       mx = (float)magCount[0]*mRes*magCalibration[0] - magbias[0];  // get
   actual magnetometer value, this depends on scale being set
305       my = (float)magCount[1]*mRes*magCalibration[1] - magbias[1];
306       mz = (float)magCount[2]*mRes*magCalibration[2] - magbias[2];
307
308       mx *= magScale[0];
309       my *= magScale[1];
310       mz *= magScale[2];
311     }
```

```
312
313    void MPU9250::initMPU9250(){
314
315        // Initialize MPU9250 device
316        // wake up device
317        writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00); // Clear sleep mode bit
   (6), enable all sensors
318        wait(0.1); // Delay 100 ms for PLL to get established on x-axis gyro; should
   check for PLL ready interrupt
319
320        // get stable time source
321        writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);  // Set clock source to
   be PLL with x-axis gyroscope reference, bits 2:0 = 001
322
323        // Configure Gyro and Accelerometer
324        // Disable FSYNC and set accelerometer and gyro bandwidth to 44 and 42 Hz,
   respectively;
325        // DLPF_CFG = bits 2:0 = 010; this sets the sample rate at 1 kHz for both
326        // Maximum delay is 4.9 ms which is just over a 200 Hz maximum rate
327        writeByte(MPU9250_ADDRESS, CONFIG, 0x03);
328
329        // Set sample rate = gyroscope output rate/(1 + SMPLRT_DIV)
330        writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x04);  // Use a 200 Hz rate; the
   same rate set in CONFIG above
331
332        // Set gyroscope full scale range
333        // Range selects FS_SEL and AFS_SEL are 0 - 3, so 2-bit values are left-shifted
   into positions 4:3
334        uint8_t c =  readByte(MPU9250_ADDRESS, GYRO_CONFIG);
335        writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c & ~0xE0); // Clear self-test
   bits [7:5]
336        writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c & ~0x18); // Clear AFS bits
   [4:3]
337        writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c | Gscale << 3); // Set full
   scale range for the gyro
338
339        // Set accelerometer configuration
340        c =  readByte(MPU9250_ADDRESS, ACCEL_CONFIG);
341        writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c & ~0xE0); // Clear self-test
   bits [7:5]
342        writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c & ~0x18); // Clear AFS bits
   [4:3]
343        writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c | Ascale << 3); // Set full
   scale range for the accelerometer
344
345        // Set accelerometer sample rate configuration
```

```
346       // It is possible to get a 4 kHz sample rate from the accelerometer by
    choosing 1 for
347       // accel_fchoice_b bit [3]; in this case the bandwidth is 1.13 kHz
348       c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG2);
349       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, c & ~0x0F); // Clear
    accel_fchoice_b (bit 3) and A_DLPFG (bits [2:0])
350       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, c | 0x03); // Set
    accelerometer rate to 1 kHz and bandwidth to 41 Hz
351
352       // The accelerometer, gyro, and thermometer are set to 1 kHz sample rates,
353       // but all these rates are further reduced by a factor of 5 to 200 Hz because of
    the SMPLRT_DIV setting
354
355       // Configure Interrupts and Bypass Enable
356       // Set interrupt pin active high, push-pull, and clear on read of INT_STATUS,
    enable I2C_BYPASS_EN so additional chips
357       // can join the I2C bus and all can be controlled by the Arduino as master
358       writeByte(MPU9250_ADDRESS, INT_PIN_CFG, 0x12);
359       writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x01);  // Enable data ready (bit
    0) interrupt
360     }
361
362     // Function which accumulates gyro and accelerometer data after device
    initialization. It calculates the average
363     // of the at-rest readings and then loads the resulting offsets into
    accelerometer and gyro bias registers.
364     void MPU9250::calibrateMPU9250(float * dest1, float * dest2){
365
366       uint8_t data[12]; // data array to hold accelerometer and gyro x, y, z, data
367       uint16_t ii, packet_count, fifo_count;
368       int32_t gyro_bias[3] = {0, 0, 0}, accel_bias[3] = {0, 0, 0};
369
370       // reset device, reset all registers, clear gyro and accelerometer bias registers
371       writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x80); // Write a one to bit 7
    reset bit; toggle reset device
372       wait(0.1);
373
374       // get stable time source
375       // Set clock source to be PLL with x-axis gyroscope reference, bits 2:0 = 001
376       writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
377       writeByte(MPU9250_ADDRESS, PWR_MGMT_2, 0x00);
378       wait(0.2);
379
380       // Configure device for bias calculation
381       writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x00);   // Disable all interrupts
382       writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);     // Disable FIFO
```

```
383       writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);  // Turn on internal
   clock source
384       writeByte(MPU9250_ADDRESS, I2C_MST_CTRL, 0x00); // Disable I2C master
385       writeByte(MPU9250_ADDRESS, USER_CTRL, 0x00);   // Disable FIFO and I2C
   master modes
386       writeByte(MPU9250_ADDRESS, USER_CTRL, 0x0C);   // Reset FIFO and DMP
387       wait(0.015);
388
389       // Configure MPU9250 gyro and accelerometer for bias calculation
390       writeByte(MPU9250_ADDRESS, CONFIG, 0x01);     // Set low-pass filter to 188
   Hz
391       writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);  // Set sample rate to 1
   kHz
392       writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0x00);  // Set gyro full-scale
   to 250 degrees per second, maximum sensitivity
393       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00); // Set accelerometer
   full-scale to 2 g, maximum sensitivity
394
395       uint16_t gyrosensitivity  = 131;  // = 131 LSB/degrees/sec
396       uint16_t accelsensitivity = 16384;  // = 16384 LSB/g
397
398       // Configure FIFO to capture accelerometer and gyro data for bias calculation
399       writeByte(MPU9250_ADDRESS, USER_CTRL, 0x40);  // Enable FIFO
400       writeByte(MPU9250_ADDRESS, FIFO_EN, 0x78);    // Enable gyro and
   accelerometer sensors for FIFO (max size 512 bytes in MPU-9250)
401       wait(0.04); // accumulate 40 samples in 80 milliseconds = 480 bytes
402
403       // At end of sample accumulation, turn off FIFO sensor read
404       writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);      // Disable gyro and
   accelerometer sensors for FIFO
405       readBytes(MPU9250_ADDRESS, FIFO_COUNTH, 2, &data[0]); // read FIFO
   sample count
406       fifo_count = ((uint16_t)data[0] << 8) | data[1];
407       packet_count = fifo_count/12;// How many sets of full gyro and
   accelerometer data for averaging
408
409       for (ii = 0; ii < packet_count; ii++) {
410         int16_t accel_temp[3] = {0, 0, 0}, gyro_temp[3] = {0, 0, 0};
411         readBytes(MPU9250_ADDRESS, FIFO_R_W, 12, &data[0]); // read data for
   averaging
412         accel_temp[0] = (int16_t) (((int16_t)data[0] << 8) | data[1] ) ;  // Form signed
   16-bit integer for each sample in FIFO
413         accel_temp[1] = (int16_t) (((int16_t)data[2] << 8) | data[3] ) ;
414         accel_temp[2] = (int16_t) (((int16_t)data[4] << 8) | data[5] ) ;
415         gyro_temp[0]  = (int16_t) (((int16_t)data[6] << 8) | data[7] ) ;
416         gyro_temp[1]  = (int16_t) (((int16_t)data[8] << 8) | data[9] ) ;
```

```
417         gyro_temp[2]  = (int16_t) (((int16_t)data[10] << 8) | data[11]) ;
418
419         accel_bias[0] += (int32_t) accel_temp[0]; // Sum individual signed 16-bit
    biases to get accumulated signed 32-bit biases
420         accel_bias[1] += (int32_t) accel_temp[1];
421         accel_bias[2] += (int32_t) accel_temp[2];
422         gyro_bias[0]  += (int32_t) gyro_temp[0];
423         gyro_bias[1]  += (int32_t) gyro_temp[1];
424         gyro_bias[2]  += (int32_t) gyro_temp[2];
425
426       }
427       accel_bias[0] /= (int32_t) packet_count; // Normalize sums to get average
    count biases
428       accel_bias[1] /= (int32_t) packet_count;
429       accel_bias[2] /= (int32_t) packet_count;
430       gyro_bias[0]  /= (int32_t) packet_count;
431       gyro_bias[1]  /= (int32_t) packet_count;
432       gyro_bias[2]  /= (int32_t) packet_count;
433
434       if(accel_bias[2] > 0L) {
435         accel_bias[2] -= (int32_t) accelsensitivity;  // Remove gravity from the z-
    axis accelerometer bias calculation
436       } else {
437         accel_bias[2] += (int32_t) accelsensitivity;
438       }
439
440       // Construct the gyro biases for push to the hardware gyro bias registers,
    which are reset to zero upon device startup
441       data[0] = (-gyro_bias[0]/4  >> 8) & 0xFF; // Divide by 4 to get 32.9 LSB per deg/s
    to conform to expected bias input format
442       data[1] = (-gyro_bias[0]/4)       & 0xFF; // Biases are additive, so change sign on
    calculated average gyro biases
443       data[2] = (-gyro_bias[1]/4  >> 8) & 0xFF;
444       data[3] = (-gyro_bias[1]/4)       & 0xFF;
445       data[4] = (-gyro_bias[2]/4  >> 8) & 0xFF;
446       data[5] = (-gyro_bias[2]/4)       & 0xFF;
447
448       /// Push gyro biases to hardware registers
449       /*  writeByte(MPU9250_ADDRESS, XG_OFFSET_H, data[0]);
450         writeByte(MPU9250_ADDRESS, XG_OFFSET_L, data[1]);
451         writeByte(MPU9250_ADDRESS, YG_OFFSET_H, data[2]);
452         writeByte(MPU9250_ADDRESS, YG_OFFSET_L, data[3]);
453         writeByte(MPU9250_ADDRESS, ZG_OFFSET_H, data[4]);
454         writeByte(MPU9250_ADDRESS, ZG_OFFSET_L, data[5]);
455       */
```

```
456       dest1[0] = (float) gyro_bias[0]/(float) gyrosensitivity; // construct gyro bias in
  deg/s for later manual subtraction
457       dest1[1] = (float) gyro_bias[1]/(float) gyrosensitivity;
458       dest1[2] = (float) gyro_bias[2]/(float) gyrosensitivity;
459
460       // Construct the accelerometer biases for push to the hardware
  accelerometer bias registers. These registers contain
461       // factory trim values which must be added to the calculated accelerometer
  biases; on boot up these registers will hold
462       // non-zero values. In addition, bit 0 of the lower byte must be preserved
  since it is used for temperature
463       // compensation calculations. Accelerometer bias registers expect bias input
  as 2048 LSB per g, so that
464       // the accelerometer biases calculated above must be divided by 8.
465
466       int32_t accel_bias_reg[3] = {0, 0, 0}; // A place to hold the factory
  accelerometer trim biases
467       readBytes(MPU9250_ADDRESS, XA_OFFSET_H, 2, &data[0]); // Read factory
  accelerometer trim values
468       accel_bias_reg[0] = (int16_t) ((int16_t)data[0] << 8) | data[1];
469       readBytes(MPU9250_ADDRESS, YA_OFFSET_H, 2, &data[0]);
470       accel_bias_reg[1] = (int16_t) ((int16_t)data[0] << 8) | data[1];
471       readBytes(MPU9250_ADDRESS, ZA_OFFSET_H, 2, &data[0]);
472       accel_bias_reg[2] = (int16_t) ((int16_t)data[0] << 8) | data[1];
473
474       uint32_t mask = 1uL; // Define mask for temperature compensation bit 0 of
  lower byte of accelerometer bias registers
475       uint8_t mask_bit[3] = {0, 0, 0}; // Define array to hold mask bit for each
  accelerometer bias axis
476
477       for(ii = 0; ii < 3; ii++) {
478         if(accel_bias_reg[ii] & mask) mask_bit[ii] = 0x01; // If temperature
  compensation bit is set, record that fact in mask_bit
479       }
480
481       // Construct total accelerometer bias, including calculated average
  accelerometer bias from above
482       accel_bias_reg[0] -= (accel_bias[0]/8); // Subtract calculated averaged
  accelerometer bias scaled to 2048 LSB/g (16 g full scale)
483       accel_bias_reg[1] -= (accel_bias[1]/8);
484       accel_bias_reg[2] -= (accel_bias[2]/8);
485
486       data[0] = (accel_bias_reg[0] >> 8) & 0xFF;
487       data[1] = (accel_bias_reg[0])    & 0xFF;
488       data[1] = data[1] | mask_bit[0]; // preserve temperature compensation bit
  when writing back to accelerometer bias registers
```

```
489       data[2] = (accel_bias_reg[1] >> 8) & 0xFF;
490       data[3] = (accel_bias_reg[1])      & 0xFF;
491       data[3] = data[3] | mask_bit[1]; // preserve temperature compensation bit
    when writing back to accelerometer bias registers
492       data[4] = (accel_bias_reg[2] >> 8) & 0xFF;
493       data[5] = (accel_bias_reg[2])      & 0xFF;
494       data[5] = data[5] | mask_bit[2]; // preserve temperature compensation bit
    when writing back to accelerometer bias registers
495
496       // Apparently this is not working for the acceleration biases in the MPU-9250
497       // Are we handling the temperature correction bit properly?
498       // Push accelerometer biases to hardware registers
499       /*  writeByte(MPU9250_ADDRESS, XA_OFFSET_H, data[0]);
500         writeByte(MPU9250_ADDRESS, XA_OFFSET_L, data[1]);
501         writeByte(MPU9250_ADDRESS, YA_OFFSET_H, data[2]);
502         writeByte(MPU9250_ADDRESS, YA_OFFSET_L, data[3]);
503         writeByte(MPU9250_ADDRESS, ZA_OFFSET_H, data[4]);
504         writeByte(MPU9250_ADDRESS, ZA_OFFSET_L, data[5]);
505       */
506       // Output scaled accelerometer biases for manual subtraction in the main
    program
507       dest2[0] = (float)accel_bias[0]/(float)accelsensitivity;
508       dest2[1] = (float)accel_bias[1]/(float)accelsensitivity;
509       dest2[2] = (float)accel_bias[2]/(float)accelsensitivity;
510     }
511
512     void MPU9250::initAK8963(float * destination){
513
514       // First extract the factory calibration for each magnetometer axis
515       uint8_t rawData[3];  // x/y/z gyro calibration data stored here
516       writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down
    magnetometer
517       wait(0.01);
518       writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x0F); // Enter Fuse ROM access
    mode
519       wait(0.01);
520       readBytes(AK8963_ADDRESS, AK8963_ASAX, 3, &rawData[0]);  // Read the
    x-, y-, and z-axis calibration values
521       destination[0] =  (float)(rawData[0] - 128)/256.0f + 1.0f;   // Return x-axis
    sensitivity adjustment values, etc.
522       destination[1] =  (float)(rawData[1] - 128)/256.0f + 1.0f;
523       destination[2] =  (float)(rawData[2] - 128)/256.0f + 1.0f;
524       writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down
    magnetometer
525       wait(0.01);
526       // Configure the magnetometer for continuous read and highest resolution
```

```
527       // set Mscale bit 4 to 1 (0) to enable 16 (14) bit resolution in CNTL register,
528       // and enable continuous mode data acquisition Mmode (bits [3:0]), 0010 for
    8 Hz and 0110 for 100 Hz sample rates
529       writeByte(AK8963_ADDRESS, AK8963_CNTL, Mscale << 4 | Mmode); // Set
    magnetometer data resolution and sample ODR
530       wait(0.01);
531     }
532
533     void MPU9250::magcalMPU9250(float * dest1, float * dest2){ //MagCal() de
    MPU9250_interface
534
535       uint16_t ii = 0, sample_count = 0;
536       int32_t mag_bias[3] = {0, 0, 0}, mag_scale[3] = {0, 0, 0};
537       int16_t mag_max[3] = {-32767, -32767, -32767}, mag_min[3] = {32767, 32767,
    32767}, mag_temp[3] = {0, 0, 0};
538
539       // shoot for ~fifteen seconds of mag data
540       if(Mmode == 0x02) sample_count = 128;  // at 8 Hz ODR, new mag data is
    available every 125 ms
541       if(Mmode == 0x06) sample_count = 1500;  // at 100 Hz ODR, new mag data is
    available every 10 ms
542
543       for(ii = 0; ii < sample_count; ii++) {
544       readMagData(mag_temp);  // Read the mag data
545
546       for (int jj = 0; jj < 3; jj++) {
547         if(mag_temp[jj] > mag_max[jj]) mag_max[jj] = mag_temp[jj];
548         if(mag_temp[jj] < mag_min[jj]) mag_min[jj] = mag_temp[jj];
549       }
550
551       if(Mmode == 0x02) wait(0.135);  // at 8 Hz ODR, new mag data is available
    every 125 ms
552       if(Mmode == 0x06) wait(0.012);  // at 100 Hz ODR, new mag data is available
    every 10 ms
553       }
554
555       // Get hard iron correction
556       mag_bias[0]  = (mag_max[0] + mag_min[0])/2;  // get average x mag bias in
    counts
557       mag_bias[1]  = (mag_max[1] + mag_min[1])/2;  // get average y mag bias in
    counts
558       mag_bias[2]  = (mag_max[2] + mag_min[2])/2;  // get average z mag bias in
    counts
559
560       dest1[0] = (float) mag_bias[0]*mRes*magCalibration[0];  // save mag biases
    in G for main program
```

```
561       dest1[1] = (float) mag_bias[1]*mRes*magCalibration[1];
562       dest1[2] = (float) mag_bias[2]*mRes*magCalibration[2];
563
564       // Get soft iron correction estimate
565       mag_scale[0]  = (mag_max[0] - mag_min[0])/2;  // get average x axis max
    chord length in counts
566       mag_scale[1]  = (mag_max[1] - mag_min[1])/2;  // get average y axis max
    chord length in counts
567       mag_scale[2]  = (mag_max[2] - mag_min[2])/2;  // get average z axis max
    chord length in counts
568
569       float avg_rad = mag_scale[0] + mag_scale[1] + mag_scale[2];
570       avg_rad /= 3.0f;
571
572       dest2[0] = avg_rad/((float)mag_scale[0]);
573       dest2[1] = avg_rad/((float)mag_scale[1]);
574       dest2[2] = avg_rad/((float)mag_scale[2]);
575     }
576
577
578     // Accelerometer and gyroscope self test; check calibration wrt factory
    settings
579     void MPU9250::MPU9250SelfTest(float * destination){   // Should return percent
    deviation from factory trim values, +/- 14 or less deviation is a pass
580
581       uint8_t rawData[6] = {0, 0, 0, 0, 0, 0};
582       uint8_t selfTest[6];
583       int16_t gAvg[3], aAvg[3], aSTAvg[3], gSTAvg[3];
584       float factoryTrim[6];
585       uint8_t FS = 0;
586
587       writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00); // Set gyro sample rate
    to 1 kHz
588       writeByte(MPU9250_ADDRESS, CONFIG, 0x02); // Set gyro sample rate to 1
    kHz and DLPF to 92 Hz
589       writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 1<<FS); // Set full scale range
    for the gyro to 250 dps
590       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, 0x02); // Set accelerometer
    rate to 1 kHz and bandwidth to 92 Hz
591       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 1<<FS); // Set full scale range
    for the accelerometer to 2 g
592
593       for( int ii = 0; ii < 200; ii++) { // get average current values of gyro and
    acclerometer
594
```

```
595         readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]); // Read
  the six raw data registers into data array
596         aAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ; // Turn the
  MSB and LSB into a signed 16-bit value
597         aAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
598         aAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
599
600         readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]); // Read
  the six raw data registers sequentially into data array
601         gAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ; // Turn the
  MSB and LSB into a signed 16-bit value
602         gAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
603         gAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
604       }
605
606       for (int ii =0; ii < 3; ii++) { // Get average of 200 values and store as average
  current readings
607         aAvg[ii] /= 200;
608         gAvg[ii] /= 200;
609       }
610
611     // Configure the accelerometer for self-test
612       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0xE0); // Enable self test on
  all three axes and set accelerometer range to +/- 2 g
613       writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0xE0); // Enable self test on
  all three axes and set gyro range to +/- 250 degrees/s
614       //delay(25); // Delay a while to let the device stabilize
615
616       for( int ii = 0; ii < 200; ii++) { // get average self-test values of gyro and
  acclerometer
617
618         readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]); // Read
  the six raw data registers into data array
619         aSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ; // Turn the
  MSB and LSB into a signed 16-bit value
620         aSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
621         aSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
622
623         readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]); // Read
  the six raw data registers sequentially into data array
624         gSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ; // Turn the
  MSB and LSB into a signed 16-bit value
625         gSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
626         gSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
627       }
628
```

```
629       for (int ii =0; ii < 3; ii++) { // Get average of 200 values and store as average
   self-test readings
630         aSTAvg[ii] /= 200;
631         gSTAvg[ii] /= 200;
632       }
633
634     // Configure the gyro and accelerometer for normal operation
635       writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);
636       writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0x00);
637       wait_ms(25); // Delay a while to let the device stabilize
638
639       // Retrieve accelerometer and gyro factory Self-Test Code from USR_Reg
640       selfTest[0] = readByte(MPU9250_ADDRESS, SELF_TEST_X_ACCEL); // X-axis
   accel self-test results
641       selfTest[1] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_ACCEL); // Y-axis
   accel self-test results
642       selfTest[2] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_ACCEL); // Z-axis
   accel self-test results
643       selfTest[3] = readByte(MPU9250_ADDRESS, SELF_TEST_X_GYRO); // X-axis
   gyro self-test results
644       selfTest[4] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_GYRO); // Y-axis
   gyro self-test results
645       selfTest[5] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_GYRO); // Z-axis
   gyro self-test results
646
647       // Retrieve factory self-test value from self-test code reads
648       factoryTrim[0] = (float)(2620/1<<FS)*(pow( (float)1.01 , ((float)selfTest[0] -
   (float)1.0) )); // FT[Xa] factory trim calculation
649       factoryTrim[1] = (float)(2620/1<<FS)*(pow( (float)1.01 , ((float)selfTest[1] -
   (float)1.0) )); // FT[Ya] factory trim calculation
650       factoryTrim[2] = (float)(2620/1<<FS)*(pow( (float)1.01 , ((float)selfTest[2] -
   (float)1.0) )); // FT[Za] factory trim calculation
651       factoryTrim[3] = (float)(2620/1<<FS)*(pow( (float)1.01 , ((float)selfTest[3] -
   (float)1.0) )); // FT[Xg] factory trim calculation
652       factoryTrim[4] = (float)(2620/1<<FS)*(pow( (float)1.01 , ((float)selfTest[4] -
   (float)1.0) )); // FT[Yg] factory trim calculation
653       factoryTrim[5] = (float)(2620/1<<FS)*(pow( (float)1.01 , ((float)selfTest[5] -
   (float)1.0) )); // FT[Zg] factory trim calculation
654
655     // Report results as a ratio of (STR - FT)/FT; the change from Factory Trim of the
   Self-Test Response
656     // To get percent, must multiply by 100
657       for (int i = 0; i < 3; i++) {
658         destination[i] = (float)100.0*((float)(aSTAvg[i] - aAvg[i]))/factoryTrim[i]; //
   Report percent differences
```

```
659         destination[i+3] = (float)100.0*((float)(gSTAvg[i] - gAvg[i]))/factoryTrim[i+3];
   // Report percent differences
660       }
661
662     }
663
664     void MPU9250::resetMPU9250(){
665       // reset device
666       writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x80); // Write a one to bit 7
   reset bit; toggle reset device
667       wait(0.1);
668     }
669
670     // Implementation of Sebastian Madgwick's "...efficient orientation filter for...
   inertial/magnetic sensor arrays"
671     // (see http://www.x-io.co.uk/category/open-source/ for examples and more
   details)
672     // which fuses acceleration, rotation rate, and magnetic moments to produce a
   quaternion-based estimate of absolute
673     // device orientation -- which can be converted to yaw, pitch, and roll. Useful
   for stabilizing quadcopters, etc.
674     // The performance of the orientation filter is at least as good as conventional
   Kalman-based filtering algorithms
675     // but is much less computationally intensive---it can be performed on a 3.3 V
   Pro Mini operating at 8 MHz!
676     void MPU9250::MadgwickQuaternionUpdate(float deltat, float ax, float ay,
   float az, float gx, float gy, float gz, float mx, float my, float mz){
677
678         float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];   // short name local variable for
   readability
679         float norm;
680         float hx, hy, _2bx, _2bz;
681         float s1, s2, s3, s4;
682         float qDot1, qDot2, qDot3, qDot4;
683
684         // Auxiliary variables to avoid repeated arithmetic
685         float _2q1mx;
686         float _2q1my;
687         float _2q1mz;
688         float _2q2mx;
689         float _4bx;
690         float _4bz;
691         float _2q1 = 2.0f * q1;
692         float _2q2 = 2.0f * q2;
693         float _2q3 = 2.0f * q3;
694         float _2q4 = 2.0f * q4;
```

```
695         float _2q1q3 = 2.0f * q1 * q3;
696         float _2q3q4 = 2.0f * q3 * q4;
697         float q1q1 = q1 * q1;
698         float q1q2 = q1 * q2;
699         float q1q3 = q1 * q3;
700         float q1q4 = q1 * q4;
701         float q2q2 = q2 * q2;
702         float q2q3 = q2 * q3;
703         float q2q4 = q2 * q4;
704         float q3q3 = q3 * q3;
705         float q3q4 = q3 * q4;
706         float q4q4 = q4 * q4;
707 
708         // Normalise accelerometer measurement
709         norm = sqrt(ax * ax + ay * ay + az * az);
710         if (norm == 0.0f) return; // handle NaN
711         norm = 1.0f/norm;
712         ax *= norm;
713         ay *= norm;
714         az *= norm;
715 
716         // Normalise magnetometer measurement
717         norm = sqrt(mx * mx + my * my + mz * mz);
718         if (norm == 0.0f) return; // handle NaN
719         norm = 1.0f/norm;
720         mx *= norm;
721         my *= norm;
722         mz *= norm;
723 
724         // Reference direction of Earth's magnetic field
725         _2q1mx = 2.0f * q1 * mx;
726         _2q1my = 2.0f * q1 * my;
727         _2q1mz = 2.0f * q1 * mz;
728         _2q2mx = 2.0f * q2 * mx;
729         hx = mx * q1q1 - _2q1my * q4 + _2q1mz * q3 + mx * q2q2 + _2q2 * my * q3 +
    _2q2 * mz * q4 - mx * q3q3 - mx * q4q4;
730         hy = _2q1mx * q4 + my * q1q1 - _2q1mz * q2 + _2q2mx * q3 - my * q2q2 + my *
    q3q3 + _2q3 * mz * q4 - my * q4q4;
731         _2bx = sqrt(hx * hx + hy * hy);
732         _2bz = -_2q1mx * q3 + _2q1my * q2 + mz * q1q1 + _2q2mx * q4 - mz * q2q2 +
    _2q3 * my * q4 - mz * q3q3 + mz * q4q4;
733         _4bx = 2.0f * _2bx;
734         _4bz = 2.0f * _2bz;
735 
736         // Gradient decent algorithm corrective step
```

```
737         s1 = -_2q3 * (2.0f * q2q4 - _2q1q3 - ax) + _2q2 * (2.0f * q1q2 + _2q3q4 - ay) -
    _2bz * q3 * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (-_2bx * q4 +
    _2bz * q2) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + _2bx * q3 * (_2bx *
    (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);
738         s2 = _2q4 * (2.0f * q2q4 - _2q1q3 - ax) + _2q1 * (2.0f * q1q2 + _2q3q4 - ay) - 4.0f
    * q2 * (1.0f - 2.0f * q2q2 - 2.0f * q3q3 - az) + _2bz * q4 * (_2bx * (0.5f - q3q3 - q4q4) +
    _2bz * (q2q4 - q1q3) - mx) + (_2bx * q3 + _2bz * q1) * (_2bx * (q2q3 - q1q4) + _2bz *
    (q1q2 + q3q4) - my) + (_2bx * q4 - _4bz * q2) * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f -
    q2q2 - q3q3) - mz);
739         s3 = -_2q1 * (2.0f * q2q4 - _2q1q3 - ax) + _2q4 * (2.0f * q1q2 + _2q3q4 - ay) -
    4.0f * q3 * (1.0f - 2.0f * q2q2 - 2.0f * q3q3 - az) + (-_4bx * q3 - _2bz * q1) * (_2bx * (0.5f -
    q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (_2bx * q2 + _2bz * q4) * (_2bx * (q2q3 -
    q1q4) + _2bz * (q1q2 + q3q4) - my) + (_2bx * q1 - _4bz * q3) * (_2bx * (q1q3 + q2q4) +
    _2bz * (0.5f - q2q2 - q3q3) - mz);
740         s4 = _2q2 * (2.0f * q2q4 - _2q1q3 - ax) + _2q3 * (2.0f * q1q2 + _2q3q4 - ay) + (-
    _4bx * q4 + _2bz * q2) * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (-
    _2bx * q1 + _2bz * q3) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + _2bx * q2
    * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);
741         norm = sqrt(s1 * s1 + s2 * s2 + s3 * s3 + s4 * s4);   // normalise step magnitude
742         norm = 1.0f/norm;
743         s1 *= norm;
744         s2 *= norm;
745         s3 *= norm;
746         s4 *= norm;
747
748         // Compute rate of change of quaternion
749         qDot1 = 0.5f * (-q2 * gx - q3 * gy - q4 * gz) - beta * s1;
750         qDot2 = 0.5f * (q1 * gx + q3 * gz - q4 * gy) - beta * s2;
751         qDot3 = 0.5f * (q1 * gy - q2 * gz + q4 * gx) - beta * s3;
752         qDot4 = 0.5f * (q1 * gz + q2 * gy - q3 * gx) - beta * s4;
753
754         // Integrate to yield quaternion
755         q1 += qDot1 * deltat;
756         q2 += qDot2 * deltat;
757         q3 += qDot3 * deltat;
758         q4 += qDot4 * deltat;
759         norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);   // normalise quaternion
760         norm = 1.0f/norm;
761         q[0] = q1 * norm;
762         q[1] = q2 * norm;
763         q[2] = q3 * norm;
764         q[3] = q4 * norm;
765     }
766
767     // Similar to Madgwick scheme but uses proportional and integral filtering on
    the error between estimated reference vectors and
```

```
768     // measured ones.
769     void MPU9250::MahonyQuaternionUpdate(float ax, float ay, float az, float gx,
    float gy, float gz, float mx, float my, float mz){
770 
771             float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];   // short name local variable
    for readability
772             float norm;
773             float hx, hy, bx, bz;
774             float vx, vy, vz, wx, wy, wz;
775             float ex, ey, ez;
776             float pa, pb, pc;
777 
778             // Auxiliary variables to avoid repeated arithmetic
779             float q1q1 = q1 * q1;
780             float q1q2 = q1 * q2;
781             float q1q3 = q1 * q3;
782             float q1q4 = q1 * q4;
783             float q2q2 = q2 * q2;
784             float q2q3 = q2 * q3;
785             float q2q4 = q2 * q4;
786             float q3q3 = q3 * q3;
787             float q3q4 = q3 * q4;
788             float q4q4 = q4 * q4;
789 
790             // Normalise accelerometer measurement
791             norm = sqrt(ax * ax + ay * ay + az * az);
792             if (norm == 0.0f) return; // handle NaN
793             norm = 1.0f / norm;       // use reciprocal for division
794             ax *= norm;
795             ay *= norm;
796             az *= norm;
797 
798             // Normalise magnetometer measurement
799             norm = sqrt(mx * mx + my * my + mz * mz);
800             if (norm == 0.0f) return; // handle NaN
801             norm = 1.0f / norm;       // use reciprocal for division
802             mx *= norm;
803             my *= norm;
804             mz *= norm;
805 
806             // Reference direction of Earth's magnetic field
807             hx = 2.0f * mx * (0.5f - q3q3 - q4q4) + 2.0f * my * (q2q3 - q1q4) + 2.0f * mz *
    (q2q4 + q1q3);
808             hy = 2.0f * mx * (q2q3 + q1q4) + 2.0f * my * (0.5f - q2q2 - q4q4) + 2.0f * mz *
    (q3q4 - q1q2);
809             bx = sqrt((hx * hx) + (hy * hy));
```

```
810         bz = 2.0f * mx * (q2q4 - q1q3) + 2.0f * my * (q3q4 + q1q2) + 2.0f * mz * (0.5f -
  q2q2 - q3q3);
811
812         // Estimated direction of gravity and magnetic field
813         vx = 2.0f * (q2q4 - q1q3);
814         vy = 2.0f * (q1q2 + q3q4);
815         vz = q1q1 - q2q2 - q3q3 + q4q4;
816         wx = 2.0f * bx * (0.5f - q3q3 - q4q4) + 2.0f * bz * (q2q4 - q1q3);
817         wy = 2.0f * bx * (q2q3 - q1q4) + 2.0f * bz * (q1q2 + q3q4);
818         wz = 2.0f * bx * (q1q3 + q2q4) + 2.0f * bz * (0.5f - q2q2 - q3q3);
819
820         // Error is cross product between estimated direction and measured
  direction of gravity
821         ex = (ay * vz - az * vy) + (my * wz - mz * wy);
822         ey = (az * vx - ax * vz) + (mz * wx - mx * wz);
823         ez = (ax * vy - ay * vx) + (mx * wy - my * wx);
824         if (Ki > 0.0f)
825         {
826           eInt[0] += ex;     // accumulate integral error
827           eInt[1] += ey;
828           eInt[2] += ez;
829         }
830         else
831         {
832           eInt[0] = 0.0f;    // prevent integral wind up
833           eInt[1] = 0.0f;
834           eInt[2] = 0.0f;
835         }
836
837         // Apply feedback terms
838         gx = gx + Kp * ex + Ki * eInt[0];
839         gy = gy + Kp * ey + Ki * eInt[1];
840         gz = gz + Kp * ez + Ki * eInt[2];
841
842         // Integrate rate of change of quaternion
843         pa = q2;
844         pb = q3;
845         pc = q4;
846         q1 = q1 + (-q2 * gx - q3 * gy - q4 * gz) * (0.5f * deltat);
847         q2 = pa + (q1 * gx + pb * gz - pc * gy) * (0.5f * deltat);
848         q3 = pb + (q1 * gy - pa * gz + pc * gx) * (0.5f * deltat);
849         q4 = pc + (q1 * gz + pa * gy - pb * gx) * (0.5f * deltat);
850
851         // Normalise quaternion
852         norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);
853         norm = 1.0f / norm;
```

```
854             q[0] = q1 * norm;
855             q[1] = q2 * norm;
856             q[2] = q3 * norm;
857             q[3] = q4 * norm;
858 
859         }
860 
861     //New filter
862     void MPU9250::IMUfilter(float halfT, float ax, float ay, float az, float gx, float gy,
      float gz) {
863         float q0 = q[0], q1 = q[1], q2 = q[2], q3 = q[3];
864         float norm;
865         float vx, vy, vz;
866         float ex, ey, ez;
867 
868         // normalise the measurements
869         norm = sqrt(ax*ax + ay*ay + az*az);
870         if(norm == 0.0f) return;
871         ax /= norm;
872         ay /= norm;
873         az /= norm;
874 
875         // estimated direction of gravity
876         vx = 2*(q1*q3 - q0*q2);
877         vy = 2*(q0*q1 + q2*q3);
878         vz = q0*q0 - q1*q1 - q2*q2 + q3*q3;
879 
880         // error is sum of cross product between reference direction of field and
      direction measured by sensor
881         ex = (ay*vz - az*vy);
882         ey = (az*vx - ax*vz);
883         ez = (ax*vy - ay*vx);
884 
885         // integral error scaled integral gain
886         eInt[0] += ex*Ki;
887         eInt[1] += ey*Ki;
888         eInt[2] += ez*Ki;
889 
890         // adjusted gyroscope measurements
891         gx += Kp*ex + eInt[0];
892         gy += Kp*ey + eInt[1];
893         gz += Kp*ez + eInt[2];
894 
895         // integrate quaternion rate and normalise
896         float q0o = q0; // he did the MATLAB to C error by not thinking of the
      beginning vector elements already being changed for the calculation of the rest!
```

```
897     float q1o = q1;
898     float q2o = q2;
899     float q3o = q3;
900     q0 += (-q1o*gx - q2o*gy - q3o*gz)*halfT;
901     q1 += (q0o*gx + q2o*gz - q3o*gy)*halfT;
902     q2 += (q0o*gy - q1o*gz + q3o*gx)*halfT;
903     q3 += (q0o*gz + q1o*gy - q2o*gx)*halfT;
904
905     // normalise quaternion
906     norm = sqrt(q0*q0 + q1*q1 + q2*q2 + q3*q3);
907     q[0] = q0 / norm;
908     q[1] = q1 / norm;
909     q[2] = q2 / norm;
910     q[3] = q3 / norm;
911 }
```

2.1.2. MPU9250.h

```
1 #ifndef MPU9250_H
2 #define MPU9250_H
3
4 #include "mbed.h"
5 #include "math.h"
6 #include "register.h"
7
8 #define PI 3.14159265358979323846f
9
10     //Mahony filter and fusion scheme, Kp for proportional feedback, Ki for
   integral
11     //#define Kp 2.0f * 5.0f
12     //#define Ki 0.0f
13
14     //New filter algo Kp, Ki
15     #define Kp 0.5f
16     #define Ki 0.0f
17
18
19     class MPU9250{
20
21     protected:
22
23     public:
24       enum Ascale {
25         AFS_2G = 0,
26         AFS_4G,
27         AFS_8G,
```

```
28          AFS_16G
29        };
30
31        enum Gscale {
32          GFS_250DPS = 0,
33          GFS_500DPS,
34          GFS_1000DPS,
35          GFS_2000DPS
36        };
37
38        enum Mscale {
39          MFS_14BITS = 0, // 0.6 mG per LSB
40          MFS_16BITS     // 0.15 mG per LSB
41        };
42
43
44        MPU9250(PinName sda, PinName scl, PinName tx, PinName rx);
45
46        I2C i2c;
47        Serial pc;
48        Timer t;
49
50        void start();
51
52        void writeByte(uint8_t address, uint8_t subAddress, uint8_t data);
53        char readByte(uint8_t address, uint8_t subAddress);
54        void readBytes(uint8_t address, uint8_t subAddress, uint8_t count, uint8_t *
    dest) ;
55
56        void initMPU9250();
57        void calibrateMPU9250(float * dest1, float * dest2);
58        void initAK8963(float * destination);
59        void magcalMPU9250(float * dest1, float * dest2);
60
61        void getAres();
62        void getGres();
63        void getMres();
64
65        void readAccelData(int16_t * destination);
66        void readGyroData(int16_t * destination);
67        void readMagData(int16_t * destination);
68        void readTempData();
69        void ReadRawAccGyroMag();
70        void YPR();
71
72        void AccelXYZCal();
```

```
73        void GyroXYZCal();
74        void MagXYZCal();
75
76        void MPU9250SelfTest(float * destination);
77        void resetMPU9250();
78
79        float ax, ay, az, gx, gy, gz, mx, my, mz; // variables to hold latest sensor data
  values
80
81        //Filtros
82        void MadgwickQuaternionUpdate(float deltat, float ax, float ay, float az, float
  gx, float gy, float gz, float mx, float my, float mz);
83        void MahonyQuaternionUpdate(float ax, float ay, float az, float gx, float gy,
  float gz, float mx, float my, float mz);
84        void IMUfilter(float dt, float ax, float ay, float az, float gx, float gy, float gz);
85
86        float GyroMeasError,beta;    // gyroscope measurement error in rads/s (start
  at 60 deg/s), then reduce after ~10 s to 3
87        float GyroMeasDrift,zeta;    // gyroscope measurement drift in rad/s/s (start
  at 0.0 deg/s/s)
88
89        float pitch, yaw, roll;
90        float deltat;                // integration interval for both filter schemes
91        int lastUpdate, firstUpdate, Now;
92
93        float q[4];
94        float eInt[3];
95
96      protected:
97        float temperature;
98        uint8_t whoami;
99
100     private:
101       int16_t accelCount[3]; // Stores the 16-bit signed accelerometer sensor
  output
102       int16_t gyroCount[3];  // Stores the 16-bit signed gyro sensor output
103       int16_t magCount[3];   // Stores the 16-bit signed magnetometer sensor
  output
104
105       float magCalibration[3], magbias[3],magScale[3];  // Factory mag calibration
  and mag bias
106       float gyroBias[3], accelBias[3]; // Bias corrections for gyro and accelerometer
107       //int16_t tempCount;   // Stores the real internal chip temperature in degrees
  Celsius
108       float SelfTest[6];
109
```

```
110     uint8_t Ascale; // AFS_2G, AFS_4G, AFS_8G, AFS_16G
111     uint8_t Gscale; // GFS_250DPS, GFS_500DPS, GFS_1000DPS, GFS_2000DPS
112     uint8_t Mscale; // MFS_14BITS or MFS_16BITS, 14-bit or 16-bit
  magnetometer resolution
113     uint8_t Mmode;      // Either 8 Hz 0x02) or 100 Hz (0x06) magnetometer data
  ODR
114     float aRes, gRes, mRes;     // scale resolutions per LSB for the sensors
115   };
116
117   #endif
```

2.1.3. register.h

```
1 //Magnetometer Registers
2 #define AK8963_ADDRESS  0x0C<<1
3 #define WHO_AM_I_AK8963  0x00 // should return 0x48
4 #define INFO         0x01
5 #define AK8963_ST1     0x02  // data ready status bit 0
6 #define AK8963_XOUT_L   0x03  // data
7 #define AK8963_XOUT_H   0x04
8 #define AK8963_YOUT_L   0x05
9 #define AK8963_YOUT_H   0x06
10    #define AK8963_ZOUT_L   0x07
11    #define AK8963_ZOUT_H   0x08
12    #define AK8963_ST2     0x09  // Data overflow bit 3 and data read error status
  bit 2
13    #define AK8963_CNTL     0x0A  // Power down (0000), single-measurement
  (0001), self-test (1000) and Fuse ROM (1111) modes on bits 3:0
14    #define AK8963_ASTC     0x0C  // Self test control
15    #define AK8963_I2CDIS   0x0F  // I2C disable
16    #define AK8963_ASAX     0x10  // Fuse ROM x-axis sensitivity adjustment value
17    #define AK8963_ASAY     0x11  // Fuse ROM y-axis sensitivity adjustment value
18    #define AK8963_ASAZ     0x12  // Fuse ROM z-axis sensitivity adjustment value
19
20    #define SELF_TEST_X_GYRO 0x00
21    #define SELF_TEST_Y_GYRO 0x01
22    #define SELF_TEST_Z_GYRO 0x02
23
24    /*
25    #define X_FINE_GAIN     0x03 // [7:0] fine gain
26    #define Y_FINE_GAIN     0x04
27    #define Z_FINE_GAIN     0x05
28    #define XA_OFFSET_H     0x06 // User-defined trim values for accelerometer
29    #define XA_OFFSET_L_TC  0x07
30    #define YA_OFFSET_H     0x08
31    #define YA_OFFSET_L_TC  0x09
```

```
32    #define ZA_OFFSET_H    0x0A
33    #define ZA_OFFSET_L_TC  0x0B
34    */
35
36    #define SELF_TEST_X_ACCEL 0x0D
37    #define SELF_TEST_Y_ACCEL 0x0E
38    #define SELF_TEST_Z_ACCEL 0x0F
39
40    #define SELF_TEST_A    0x10
41
42    #define XG_OFFSET_H    0x13  // User-defined trim values for gyroscope
43    #define XG_OFFSET_L    0x14
44    #define YG_OFFSET_H    0x15
45    #define YG_OFFSET_L    0x16
46    #define ZG_OFFSET_H    0x17
47    #define ZG_OFFSET_L    0x18
48    #define SMPLRT_DIV     0x19
49    #define CONFIG         0x1A
50    #define GYRO_CONFIG    0x1B
51    #define ACCEL_CONFIG   0x1C
52    #define ACCEL_CONFIG2  0x1D
53    #define LP_ACCEL_ODR   0x1E
54    #define WOM_THR        0x1F
55
56    #define MOT_DUR        0x20  // Duration counter threshold for motion
  interrupt generation, 1 kHz rate, LSB = 1 ms
57    #define ZMOT_THR       0x21  // Zero-motion detection threshold bits [7:0]
58    #define ZRMOT_DUR      0x22  // Duration counter threshold for zero motion
  interrupt generation, 16 Hz rate, LSB = 64 ms
59
60    #define FIFO_EN        0x23
61    #define I2C_MST_CTRL   0x24
62    #define I2C_SLV0_ADDR  0x25
63    #define I2C_SLV0_REG   0x26
64    #define I2C_SLV0_CTRL  0x27
65    #define I2C_SLV1_ADDR  0x28
66    #define I2C_SLV1_REG   0x29
67    #define I2C_SLV1_CTRL  0x2A
68    #define I2C_SLV2_ADDR  0x2B
69    #define I2C_SLV2_REG   0x2C
70    #define I2C_SLV2_CTRL  0x2D
71    #define I2C_SLV3_ADDR  0x2E
72    #define I2C_SLV3_REG   0x2F
73    #define I2C_SLV3_CTRL  0x30
74    #define I2C_SLV4_ADDR  0x31
75    #define I2C_SLV4_REG   0x32
```

```
76    #define I2C_SLV4_DO    0x33
77    #define I2C_SLV4_CTRL  0x34
78    #define I2C_SLV4_DI    0x35
79    #define I2C_MST_STATUS  0x36
80    #define INT_PIN_CFG    0x37
81    #define INT_ENABLE     0x38
82    #define DMP_INT_STATUS  0x39  // Check DMP interrupt
83    #define INT_STATUS     0x3A
84    #define ACCEL_XOUT_H   0x3B
85    #define ACCEL_XOUT_L   0x3C
86    #define ACCEL_YOUT_H   0x3D
87    #define ACCEL_YOUT_L   0x3E
88    #define ACCEL_ZOUT_H   0x3F
89    #define ACCEL_ZOUT_L   0x40
90    #define TEMP_OUT_H     0x41
91    #define TEMP_OUT_L     0x42
92    #define GYRO_XOUT_H    0x43
93    #define GYRO_XOUT_L    0x44
94    #define GYRO_YOUT_H    0x45
95    #define GYRO_YOUT_L    0x46
96    #define GYRO_ZOUT_H    0x47
97    #define GYRO_ZOUT_L    0x48
98    #define EXT_SENS_DATA_00 0x49
99    #define EXT_SENS_DATA_01 0x4A
100   #define EXT_SENS_DATA_02 0x4B
101   #define EXT_SENS_DATA_03 0x4C
102   #define EXT_SENS_DATA_04 0x4D
103   #define EXT_SENS_DATA_05 0x4E
104   #define EXT_SENS_DATA_06 0x4F
105   #define EXT_SENS_DATA_07 0x50
106   #define EXT_SENS_DATA_08 0x51
107   #define EXT_SENS_DATA_09 0x52
108   #define EXT_SENS_DATA_10 0x53
109   #define EXT_SENS_DATA_11 0x54
110   #define EXT_SENS_DATA_12 0x55
111   #define EXT_SENS_DATA_13 0x56
112   #define EXT_SENS_DATA_14 0x57
113   #define EXT_SENS_DATA_15 0x58
114   #define EXT_SENS_DATA_16 0x59
115   #define EXT_SENS_DATA_17 0x5A
116   #define EXT_SENS_DATA_18 0x5B
117   #define EXT_SENS_DATA_19 0x5C
118   #define EXT_SENS_DATA_20 0x5D
119   #define EXT_SENS_DATA_21 0x5E
120   #define EXT_SENS_DATA_22 0x5F
121   #define EXT_SENS_DATA_23 0x60
```

```
122   #define MOT_DETECT_STATUS 0x61
123   #define I2C_SLV0_DO    0x63
124   #define I2C_SLV1_DO    0x64
125   #define I2C_SLV2_DO    0x65
126   #define I2C_SLV3_DO    0x66
127   #define I2C_MST_DELAY_CTRL 0x67
128   #define SIGNAL_PATH_RESET  0x68
129   #define MOT_DETECT_CTRL  0x69
130   #define USER_CTRL      0x6A  // Bit 7 enable DMP, bit 3 reset DMP
131   #define PWR_MGMT_1     0x6B // Device defaults to the SLEEP mode
132   #define PWR_MGMT_2     0x6C
133   #define DMP_BANK       0x6D  // Activates a specific bank in the DMP
134   #define DMP_RW_PNT     0x6E  // Set read/write pointer to a specific start
  address in specified DMP bank
135   #define DMP_REG        0x6F  // Register in DMP from which to read or to which
  to write
136   #define DMP_REG_1      0x70
137   #define DMP_REG_2      0x71
138   #define FIFO_COUNTH    0x72
139   #define FIFO_COUNTL    0x73
140   #define FIFO_R_W       0x74
141   #define WHO_AM_I_MPU9250 0x75 // Should return 0x71
142   #define XA_OFFSET_H    0x77
143   #define XA_OFFSET_L    0x78
144   #define YA_OFFSET_H    0x7A
145   #define YA_OFFSET_L    0x7B
146   #define ZA_OFFSET_H    0x7D
147   #define ZA_OFFSET_L    0x7E
148
149
150   // Using the MSENSR-9250 breakout board, ADO is set to 0
151   // Seven-bit device address is 110100 for ADO = 0 and 110101 for ADO = 1
152   //mbed uses the eight-bit device address, so shift seven-bit addresses left by
  one!
153   #define ADO 0
154   #if ADO
155   #define MPU9250_ADDRESS 0x69<<1  // Device address when ADO = 1
156   #else
157   #define MPU9250_ADDRESS 0x68<<1  // Device address when ADO = 0
158   #endif
```

2.1.4. MainMPU9250.cpp

```
1 #include "MPU9250.h"
2 #include "mbed.h"
3
4 static DigitalOut testLed(LED1);
5
6 I2C i2c(PB_4,PA_8);
7 Serial pc(USBTX,USBRX);
8
9 MPU9250 mpu9250(i2c,pc);
10
11      volatile bool newData = false;
12      InterruptIn isrPin(PA_10);
13
14      void mpuisr(){
15        newData=true;
16      }
17
18      void calc_tvc(){
19        if(newData) {  // On interrupt, check if data ready interrupt
20          newData = false;//if(mpu9250.readByte(MPU9250_ADDRESS, INT_STATUS)
    & 0x01) {  // On interrupt, check if data ready interrupt
21          mpu9250.ReadRawAccGyroMag();
22        }
23
24        mpu9250.YPR();
25
26        testLed=!testLed;
27
28      }
29
30
31      int config_imu(){
32
33        mpu9250.start();
34
35        isrPin.rise(&mpuisr);
36
37        while(1){loop();}
38      }
39
```

2.2. ‹ClaseBMP280›

2.2.1. BMP.cpp

```
1 #include "BMP280.h"
2 #include "mbed.h"
3
4 /*!
5  * @brief  BMP280 constructor using i2c
6  * @param  *theWire
7  *       optional wire
8  */
9 BMP280::BMP280(I2C& i2c): bmp_i2c(i2c){
10
11       bmp_i2c.frequency(400000);
12       _configReg.t_sb=3;
13       _configReg.filter=3;
14       _configReg.spi3w_en=1;
15
16       _measReg.osrs_t=3;
17       _measReg.osrs_p=3;
18       _measReg.mode=2;
19
20
21
22     }
23     /*!
24      * Initialises the sensor.
25      * @param addr
26      *       The I2C address to use (default = 0x77)
27      * @param chipid
28      *       The expected chip ID (used to validate connection).
29      * @return True if the init was successful, otherwise false.
30      */
31     bool BMP280::start(uint8_t addr, uint8_t chipid) {
32       _i2caddr = addr;
33
34       if (read8(BMP280_REGISTER_CHIPID) != chipid){
35         return false;
36       }
37       readCoefficients();
38       // write8(BMP280_REGISTER_CONTROL, 0x3F); /* needed? */
39       setSampling();
40       wait_ms(100);
```

```
41      return true;
42    }
43
44    /*!
45     * Sets the sampling config for the device.
46     * @param mode
47     *      The operating mode of the sensor.
48     * @param tempSampling
49     *      The sampling scheme for temp readings.
50     * @param pressSampling
51     *      The sampling scheme for pressure readings.
52     * @param filter
53     *      The filtering mode to apply (if any).
54     * @param duration
55     *      The sampling duration.
56     */
57    void BMP280::setSampling(sensor_mode mode,
58                             sensor_sampling tempSampling,
59                             sensor_sampling pressSampling,
60                             sensor_filter filter,
61                             standby_duration duration) {
62      _measReg.mode = mode;
63      _measReg.osrs_t = tempSampling;
64      _measReg.osrs_p = pressSampling;
65
66      _configReg.filter = filter;
67      _configReg.t_sb = duration;
68
69      write8(BMP280_REGISTER_CONFIG, _configReg.get());
70      write8(BMP280_REGISTER_CONTROL, _measReg.get());
71    }
72
73    /**************************************************************************/
74    /*!
75        @brief  Writes an 8 bit value over I2C/SPI
76    */
77    /**************************************************************************/
78    void BMP280::write8(uint8_t reg, uint8_t value) {
79        char data_write[2];
80        data_write[0] = reg;
81        data_write[1] = value;
82        bmp_i2c.write(_i2caddr,data_write,2,0);
83
84    }
85
86    /*!
```

```
87      * @brief  Reads an 8 bit value over I2C/SPI
88      * @param  reg
89      *       selected register
90      * @return value from selected register
91      */
92     uint8_t BMP280::read8(uint8_t reg) {
93      uint8_t out;
94
95      char data[1];
96      char data_write[1];
97      data_write[0] = reg;
98      bmp_i2c.write(_i2caddr,data_write,1,1);
99      bmp_i2c.read(_i2caddr,data,1,0);
100     out=data[0];
101
102     return out;
103    }
104
105    /*!
106     * @brief  Reads a 16 bit value over I2C/SPI
107     */
108    uint16_t BMP280::read16(uint8_t reg) {
109     uint16_t out;
110
111     char data[2];
112     char data_write[1];
113     data_write[0] = reg;
114     bmp_i2c.write(_i2caddr,data_write,1,1);
115     bmp_i2c.read(_i2caddr,data,2,0);
116
117     out = (data[0] << 8) | data[1];
118
119     return out;
120    }
121
122    uint16_t BMP280::read16_LE(uint8_t reg) {
123     uint16_t temp = read16(reg);
124     return (temp >> 8) | (temp << 8);
125    }
126
127    /*!
128     *  @brief  Reads a signed 16 bit value over I2C/SPI
129     */
130    int16_t BMP280::readS16(uint8_t reg) {
131     return (int16_t)read16(reg);
132    }
```

```
133
134  int16_t BMP280::readS16_LE(uint8_t reg) {
135   return (int16_t)read16_LE(reg);
136  }
137
138  /*!
139   *  @brief  Reads a 24 bit value over I2C/SPI
140   */
141  uint32_t BMP280::read24(uint8_t reg) {
142   uint32_t out;
143
144   char data[3];
145   char data_write[1];
146   data_write[0] = reg;
147   bmp_i2c.write(_i2caddr,data_write,1,1);
148   bmp_i2c.read(_i2caddr,data,3,0);
149
150   out = data[0];
151   out <<= 8;
152   out |= data[1];
153   out <<=8;
154   out |= data[2];
155
156   return out;
157  }
158
159  /*!
160   *  @brief  Reads the factory-set coefficients
161   */
162  void BMP280::readCoefficients() {
163   _bmp280_calib.dig_T1 = read16_LE(BMP280_REGISTER_DIG_T1);
164   _bmp280_calib.dig_T2 = readS16_LE(BMP280_REGISTER_DIG_T2);
165   _bmp280_calib.dig_T3 = readS16_LE(BMP280_REGISTER_DIG_T3);
166
167   _bmp280_calib.dig_P1 = read16_LE(BMP280_REGISTER_DIG_P1);
168   _bmp280_calib.dig_P2 = readS16_LE(BMP280_REGISTER_DIG_P2);
169   _bmp280_calib.dig_P3 = readS16_LE(BMP280_REGISTER_DIG_P3);
170   _bmp280_calib.dig_P4 = readS16_LE(BMP280_REGISTER_DIG_P4);
171   _bmp280_calib.dig_P5 = readS16_LE(BMP280_REGISTER_DIG_P5);
172   _bmp280_calib.dig_P6 = readS16_LE(BMP280_REGISTER_DIG_P6);
173   _bmp280_calib.dig_P7 = readS16_LE(BMP280_REGISTER_DIG_P7);
174   _bmp280_calib.dig_P8 = readS16_LE(BMP280_REGISTER_DIG_P8);
175   _bmp280_calib.dig_P9 = readS16_LE(BMP280_REGISTER_DIG_P9);
176
177  }
178
```

```
179  /*!
180   * Reads the temperature from the device.
181   * @return The temperature in degress celcius.
182   */
183  float BMP280::readTemperature() {
184   int32_t var1, var2;
185
186   int32_t adc_T = read24(BMP280_REGISTER_TEMPDATA);
187   adc_T >>= 4;
188
189   var1 = ((((adc_T >> 3) - ((int32_t)_bmp280_calib.dig_T1 << 1))) *
190       ((int32_t)_bmp280_calib.dig_T2)) >>
191       11;
192
193   var2 = (((((adc_T >> 4) - ((int32_t)_bmp280_calib.dig_T1)) *
194        ((adc_T >> 4) - ((int32_t)_bmp280_calib.dig_T1))) >>
195        12) *
196       ((int32_t)_bmp280_calib.dig_T3)) >>
197       14;
198
199   t_fine = var1 + var2;
200
201   float T = (t_fine * 5 + 128) >> 8;
202   return T / 100;
203  }
204
205  /*!
206   * Reads the barometric pressure from the device.
207   * @return Barometric pressure in hPa.
208   */
209  float BMP280::readPressure() {
210   int64_t var1, var2, p;
211
212   // Must be done first to get the t_fine variable set up
213   readTemperature();
214
215   int32_t adc_P = read24(BMP280_REGISTER_PRESSUREDATA);
216   adc_P >>= 4;
217
218   var1 = ((int64_t)t_fine) - 128000;
219   var2 = var1 * var1 * (int64_t)_bmp280_calib.dig_P6;
220   var2 = var2 + ((var1 * (int64_t)_bmp280_calib.dig_P5) << 17);
221   var2 = var2 + (((int64_t)_bmp280_calib.dig_P4) << 35);
222   var1 = ((var1 * var1 * (int64_t)_bmp280_calib.dig_P3) >> 8) +
223       ((var1 * (int64_t)_bmp280_calib.dig_P2) << 12);
224   var1 =
```

```
225       (((((int64_t)1) << 47) + var1)) * ((int64_t)_bmp280_calib.dig_P1) >> 33;
226
227     if (var1 == 0) {
228       return 0; // avoid exception caused by division by zero
229     }
230     p = 1048576 - adc_P;
231     p = (((p << 31) - var2) * 3125) / var1;
232     var1 = (((int64_t)_bmp280_calib.dig_P9) * (p >> 13) * (p >> 13)) >> 25;
233     var2 = (((int64_t)_bmp280_calib.dig_P8) * p) >> 19;
234
235     p = ((p + var1 + var2) >> 8) + (((int64_t)_bmp280_calib.dig_P7) << 4);
236     return (float)p / 256;
237   }
238
239   /*!
240    * @brief Calculates the approximate altitude using barometric pressure and
   the
241    * supplied sea level hPa as a reference.
242    * @param seaLevelhPa
243    *      The current hPa at sea level.
244    * @return The approximate altitude above sea level in meters.
245    */
246   float BMP280::readAltitude(float seaLevelhPa) {
247     float altitude;
248
249     float pressure = readPressure(); // in Si units for Pascal
250     pressure /= 100;
251
252     //Formula Adafruit
253     //altitude = 44330 * (1.0f - pow(pressure / seaLevelhPa, 0.1903f));
254     //Formula hyposmetrica
255     altitude=((pow(seaLevelhPa/pressure,1.0f/5.257f)-1.0f)*(readTemperature()
   +273.15f))/(0.0065f);
256
257     return altitude;
258   }
259
260   /*!
261    *  @brief  Take a new measurement (only possible in forced mode)
262    *  !!!todo!!!
263    */
264   /*
265   void Adafruit_BMP280::takeForcedMeasurement()
266   {
267     // If we are in forced mode, the BME sensor goes back to sleep after each
268     // measurement and we need to set it to forced mode once at this point, so
```

```
269     // it will take the next measurement and then return to sleep again.
270     // In normal mode simply does new measurements periodically.
271     if (_measReg.mode == MODE_FORCED) {
272       // set to forced mode, i.e. "take next measurement"
273       write8(BMP280_REGISTER_CONTROL, _measReg.get());
274       // wait until measurement has been completed, otherwise we would read
275       // the values from the last measurement
276       while (read8(BMP280_REGISTER_STATUS) & 0x08)
277           delay(1);
278     }
279   }
280   */
```

2.2.2. BMP.h

```
1 /*!
2  * @file Adafruit_BMP280.h
3  *
4  * This is a library for the Adafruit BMP280 Breakout.
5  *
6  * Designed specifically to work with the Adafruit BMP280 Breakout.
7  *
8  * Pick one up today in the adafruit shop!
9  * ------> https://www.adafruit.com/product/2651
10     *
11     * These sensors use I2C to communicate, 2 pins are required to interface.
12     *
13     * Adafruit invests time and resources providing this open source code,
14     * please support Adafruit andopen-source hardware by purchasing products
15     * from Adafruit!
16     *
17     * K.Townsend (Adafruit Industries)
18     *
19     * BSD license, all text above must be included in any redistribution
20     */
21    #ifndef __BMP280_H__
22    #define __BMP280_H__
23
24    #include "mbed.h"
25
26    extern Serial pc;
27
28
29    /*!
30     * I2C ADDRESS/BITS/SETTINGS
31     */
```

```
32    #define BMP280_ADDRESS (0x76<<1) /**< The default I2C address for the
  sensor. */
33    #define BMP280_CHIPID (0x58) /**< Default chip ID. */
34
35    /*!
36     * Registers available on the sensor.
37     */
38    enum {
39      BMP280_REGISTER_DIG_T1 = 0x88,
40      BMP280_REGISTER_DIG_T2 = 0x8A,
41      BMP280_REGISTER_DIG_T3 = 0x8C,
42      BMP280_REGISTER_DIG_P1 = 0x8E,
43      BMP280_REGISTER_DIG_P2 = 0x90,
44      BMP280_REGISTER_DIG_P3 = 0x92,
45      BMP280_REGISTER_DIG_P4 = 0x94,
46      BMP280_REGISTER_DIG_P5 = 0x96,
47      BMP280_REGISTER_DIG_P6 = 0x98,
48      BMP280_REGISTER_DIG_P7 = 0x9A,
49      BMP280_REGISTER_DIG_P8 = 0x9C,
50      BMP280_REGISTER_DIG_P9 = 0x9E,
51      BMP280_REGISTER_CHIPID = 0xD0,
52      BMP280_REGISTER_VERSION = 0xD1,
53      BMP280_REGISTER_SOFTRESET = 0xE0,
54      BMP280_REGISTER_CAL26 = 0xE1, /**< R calibration = 0xE1-0xF0 */
55      BMP280_REGISTER_CONTROL = 0xF4,
56      BMP280_REGISTER_CONFIG = 0xF5,
57      BMP280_REGISTER_PRESSUREDATA = 0xF7,
58      BMP280_REGISTER_TEMPDATA = 0xFA,
59    };
60
61    /*!
62     *  Struct to hold calibration data.
63     */
64    typedef struct {
65      uint16_t dig_T1; /**< dig_T1 cal register. */
66      int16_t dig_T2;  /**<  dig_T2 cal register. */
67      int16_t dig_T3;  /**< dig_T3 cal register. */
68
69      uint16_t dig_P1; /**< dig_P1 cal register. */
70      int16_t dig_P2;  /**< dig_P2 cal register. */
71      int16_t dig_P3;  /**< dig_P3 cal register. */
72      int16_t dig_P4;  /**< dig_P4 cal register. */
73      int16_t dig_P5;  /**< dig_P5 cal register. */
74      int16_t dig_P6;  /**< dig_P6 cal register. */
75      int16_t dig_P7;  /**< dig_P7 cal register. */
76      int16_t dig_P8;  /**< dig_P8 cal register. */
```

```
77      int16_t dig_P9;  /**< dig_P9 cal register. */
78
79      uint8_t dig_H1; /**< dig_H1 cal register. */
80      int16_t dig_H2; /**< dig_H2 cal register. */
81      uint8_t dig_H3; /**< dig_H3 cal register. */
82      int16_t dig_H4; /**< dig_H4 cal register. */
83      int16_t dig_H5; /**< dig_H5 cal register. */
84      int8_t dig_H6;  /**< dig_H6 cal register. */
85    } bmp280_calib_data;
86
87    /**
88     * Driver for the Adafruit BMP280 barometric pressure sensor.
89     */
90    class BMP280 {
91    public:
92      /** Oversampling rate for the sensor. */
93      enum sensor_sampling {
94        /** No over-sampling. */
95        SAMPLING_NONE = 0x00,
96        /** 1x over-sampling. */
97        SAMPLING_X1 = 0x01,
98        /** 2x over-sampling. */
99        SAMPLING_X2 = 0x02,
100       /** 4x over-sampling. */
101       SAMPLING_X4 = 0x03,
102       /** 8x over-sampling. */
103       SAMPLING_X8 = 0x04,
104       /** 16x over-sampling. */
105       SAMPLING_X16 = 0x05
106     };
107
108     /** Operating mode for the sensor. */
109     enum sensor_mode {
110       /** Sleep mode. */
111       MODE_SLEEP = 0x00,
112       /** Forced mode. */
113       MODE_FORCED = 0x01,
114       /** Normal mode. */
115       MODE_NORMAL = 0x03,
116       /** Software reset. */
117       MODE_SOFT_RESET_CODE = 0xB6
118     };
119
120     /** Filtering level for sensor data. */
121     enum sensor_filter {
122       /** No filtering. */
```

```
123       FILTER_OFF = 0x00,
124       /** 2x filtering. */
125       FILTER_X2 = 0x01,
126       /** 4x filtering. */
127       FILTER_X4 = 0x02,
128       /** 8x filtering. */
129       FILTER_X8 = 0x03,
130       /** 16x filtering. */
131       FILTER_X16 = 0x04
132     };
133
134     /** Standby duration in ms */
135     enum standby_duration {
136       /** 1 ms standby. */
137       STANDBY_MS_1 = 0x00,
138       /** 63 ms standby. */
139       STANDBY_MS_63 = 0x01,
140       /** 125 ms standby. */
141       STANDBY_MS_125 = 0x02,
142       /** 250 ms standby. */
143       STANDBY_MS_250 = 0x03,
144       /** 500 ms standby. */
145       STANDBY_MS_500 = 0x04,
146       /** 1000 ms standby. */
147       STANDBY_MS_1000 = 0x05,
148       /** 2000 ms standby. */
149       STANDBY_MS_2000 = 0x06,
150       /** 4000 ms standby. */
151       STANDBY_MS_4000 = 0x07
152     };
153
154     BMP280(I2C& i2c);
155
156     bool start(uint8_t addr = BMP280_ADDRESS, uint8_t chipid =
  BMP280_CHIPID);
157
158     float readTemperature();
159
160     float readPressure(void);
161
162     float readAltitude(float seaLevelhPa = 1013.25);
163
164     // void takeForcedMeasurement();
165
166     void setSampling(sensor_mode mode = MODE_NORMAL,
167                      sensor_sampling tempSampling = SAMPLING_X16,
```

```
168             sensor_sampling pressSampling = SAMPLING_X16,
169             sensor_filter filter = FILTER_X8,
170             standby_duration duration = STANDBY_MS_1);
171   protected:
172     I2C& bmp_i2c;
173
174   private:
175
176     /** Encapsulates the config register */
177     struct config {
178       /** Inactive duration (standby time) in normal mode */
179       unsigned int t_sb;
180       /** Filter settings */
181       unsigned int filter;
182       /** Enables 3-wire SPI */
183       unsigned int spi3w_en;
184       /** Used to retrieve the assembled config register's byte value. */
185       unsigned int get() { return (t_sb << 5) | (filter << 2) | spi3w_en; }
186     };
187
188     /** Encapsulates trhe ctrl_meas register */
189     struct ctrl_meas {
190       /** Temperature oversampling. */
191       unsigned int osrs_t;
192       /** Pressure oversampling. */
193       unsigned int osrs_p;
194       /** Device mode */
195       unsigned int mode;
196       /** Used to retrieve the assembled ctrl_meas register's byte value. */
197       unsigned int get() { return (osrs_t << 5) | (osrs_p << 2) | mode; }
198     };
199
200     void readCoefficients(void);
201     void write8(uint8_t reg, uint8_t value);
202     uint8_t read8(uint8_t reg);
203     uint16_t read16(uint8_t reg);
204     uint32_t read24(uint8_t reg);
205     int16_t readS16(uint8_t reg);
206     uint16_t read16_LE(uint8_t reg);
207     int16_t readS16_LE(uint8_t reg);
208
209     uint8_t _i2caddr;
210
211     int32_t _sensorID;
212     int32_t t_fine;
213     bmp280_calib_data _bmp280_calib;
```

```
214     config _configReg;
215     ctrl_meas _measReg;
216   };
217
218   #endif
```

2.2.3. MainBMP.cpp

```
1 #include "mbed.h"
2 #include "BMP280.h"
3
4 Serial pc(USBTX,USBRX);
5 Timer t;
6
7 I2C i2c(I2C_SDA, I2C_SCL);
8
9 BMP280 baro(i2c);
10
11      void altitude(){
12
13        pc.printf("%d:",t.read_ms());
14        pc.printf("%f,%f\r\n",baro.readAltitude(1013.25),baro.readPressure());
15        wait_ms(100);
16
17        //pc.putc(27);      // ESC command
18       // pc.printf("[2J");  //Clear Terminal in putty
19
20      }
21
22      int config_baro(){
23        pc.baud(9600);
24
25        if (!baro.start()) {
26          pc.printf("Could not find a valid BMP280 sensor, check wiring!\r\n");
27        while (1);
28        }
29        baro.setSampling(BMP280::MODE_NORMAL,    /* Operating Mode. */
30                         BMP280::SAMPLING_X2,    /* Temp. oversampling */
31                         BMP280::SAMPLING_X16,   /* Pressure oversampling */
32                         BMP280::FILTER_X16,     /* Filtering. */
33                         BMP280::STANDBY_MS_500); /* Standby time. */
34
35        t.start();
36        while(1){loop();}
37      }
```

2.3. ‹ClaseServo›

2.3.1. Servos.h

```
1 #include "mbed.h"
2
3 #define ServoX_Output PB_3
4 #define ServoY_Output PB_5
5
6 PwmOut SX(ServoX_Output);
7 PwmOut SY(ServoY_Output);
8
9 int servoXMMM[3]= {1025,1325,1625}; //range 600
10     int servoYMMM[3]= {1175,1425,1675}; //range 500
11
12     int rangeServoX = servoXMMM[2]-servoXMMM[0];
13     int rangeServoY = servoYMMM[2]-servoYMMM[0];
14
15     void config_servos(){
16       SX.period_ms(20);
17       SY.period_ms(20);
18       SX.pulsewidth_us(servoXMMM[1]);
19       SY.pulsewidth_us(servoYMMM[1]);
20     }
```

2.4. ‹ClaseFSM›

2.4.1. FSM.cpp

```
1 /*
2 ||
3 || @file FSM.cpp
4 || @version 1.7
5 || @author Alexander Brevig
6 || @contact alexanderbrevig@gmail.com
7 ||
8 || @description
9 || | Provide an easy way of making finite state machines
10     || #
11     ||
12     || @license
13     || | This library is free software; you can redistribute it and/or
14     || | modify it under the terms of the GNU Lesser General Public
```

```
15    || | License as published by the Free Software Foundation; version
16    || | 2.1 of the License.
17    || |
18    || | This library is distributed in the hope that it will be useful,
19    || | but WITHOUT ANY WARRANTY; without even the implied warranty of
20    || | MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
21    || | Lesser General Public License for more details.
22    || |
23    || | You should have received a copy of the GNU Lesser General Public
24    || | License along with this library; if not, write to the Free Software
25    || | Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
26    || #
27    ||
28    */
29
30    #include "FSM.h"
31
32    //FINITE STATE
33    State::State( void (*updateFunction)() ){
34      userEnter = 0;
35      userUpdate = updateFunction;
36      userExit = 0;
37    }
38
39    State::State( void (*enterFunction)(), void (*updateFunction)(), void
  (*exitFunction)() ){
40      userEnter = enterFunction;
41      userUpdate = updateFunction;
42      userExit = exitFunction;
43    }
44
45    //what to do when entering this state
46    void State::enter(){
47      if (userEnter){
48        userEnter();
49      }
50    }
51
52    //what to do when this state updates
53    void State::update(){
54      if (userUpdate){
55        userUpdate();
56      }
57    }
58
59    //what to do when exiting this state
```

```
60    void State::exit(){
61      if (userExit){
62        userExit();
63      }
64    }
65    //END FINITE STATE
66
67
68    //FINITE STATE MACHINE
69    FSM::FSM(State& current){
70      needToTriggerEnter = true;
71      currentState = nextState = &current;
72      stateChangeTime = 0;
73      t_FSM.start();
74    }
75
76    FSM& FSM::update() {
77      //simulate a transition to the first state
78      //this only happens the first time update is called
79      if (needToTriggerEnter) {
80        currentState->enter();
81        needToTriggerEnter = false;
82      } else {
83        if (currentState != nextState){
84          immediateTransitionTo(*nextState);
85        }
86        currentState->update();
87      }
88      return *this;
89    }
90
91    FSM& FSM::transitionTo(State& state){
92      nextState = &state;
93      stateChangeTime = t_FSM.read_ms();
94      return *this;
95    }
96
97    FSM& FSM::immediateTransitionTo(State& state){
98      currentState->exit();
99      currentState = nextState = &state;
100     currentState->enter();
101     stateChangeTime = t_FSM.read_ms();
102     return *this;
103   }
104
105   //return the current state
```

```
106   State& FSM::getCurrentState() {
107     return *currentState;
108   }
109
110   //check if state is equal to the currentState
111   bool FSM::isInState( State &state ) const {
112     if (&state == currentState) {
113       return true;
114     } else {
115       return false;
116     }
117   }
118
119   unsigned long FSM::timeInCurrentState() {
120     return t_FSM.read_ms() - stateChangeTime;
121   }
122   //END FINITE STATE MACHINE
```

2.4.2. FSM.h

```
1 /*
2 ||
3 || @file FiniteStateMachine.h
4 || @version 1.7
5 || @author Alexander Brevig
6 || @contact alexanderbrevig@gmail.com
7 ||
8 || @description
9 || | Provide an easy way of making finite state machines
10    || #
11    ||
12    || @license
13    || | This library is free software; you can redistribute it and/or
14    || | modify it under the terms of the GNU Lesser General Public
15    || | License as published by the Free Software Foundation; version
16    || | 2.1 of the License.
17    || |
18    || | This library is distributed in the hope that it will be useful,
19    || | but WITHOUT ANY WARRANTY; without even the implied warranty of
20    || | MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
21    || | Lesser General Public License for more details.
22    || |
23    || | You should have received a copy of the GNU Lesser General Public
24    || | License along with this library; if not, write to the Free Software
25    || | Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
26    || #
```

```
27      ||
28      */
29
30      #ifndef FSM_H
31      #define FSM_H
32
33      #include "mbed.h"
34
35      #define NO_ENTER (0)
36      #define NO_UPDATE (0)
37      #define NO_EXIT (0)
38
39      //#define FSM FiniteStateMachine
40
41      //define the functionality of the states
42      class State {
43        public:
44          State( void (*updateFunction)() );
45          State( void (*enterFunction)(), void (*updateFunction)(), void (*exitFunction)
    () );
46          //State( byte newId, void (*enterFunction)(), void (*updateFunction)(), void
    (*exitFunction)() );
47
48          //void getId();
49          void enter();
50          void update();
51          void exit();
52        private:
53          //byte id;
54          void (*userEnter)();
55          void (*userUpdate)();
56          void (*userExit)();
57      };
58
59      //define the finite state machine functionality
60      class FSM {
61        public:
62          FSM(State& current);
63
64          FSM& update();
65          FSM& transitionTo( State& state );
66          FSM& immediateTransitionTo( State& state );
67
68          State& getCurrentState();
69          bool isInState( State &state ) const;
70
```

```
71          unsigned long timeInCurrentState();
72
73        private:
74          bool   needToTriggerEnter;
75          State* currentState;
76          State* nextState;
77          unsigned long stateChangeTime;
78          Timer t_FSM;
79      };
80
81      #endif
82
83      /*
84      || @changelog
85      || | 1.7 2010-03-08- Alexander Brevig : Fixed a bug, constructor ran update,
    thanks to René Pressé
86      || | 1.6 2010-03-08- Alexander Brevig : Added timeInCurrentState() , requested
    by sendhb
87      || | 1.5 2009-11-29- Alexander Brevig : Fixed a bug, introduced by the below fix,
    thanks to Jon Hylands again...
88      || | 1.4 2009-11-29- Alexander Brevig : Fixed a bug, enter gets triggered on the
    first state. Big thanks to Jon Hylands who pointed this out.
89      || | 1.3 2009-11-01 - Alexander Brevig : Added getCurrentState : &State
90      || | 1.3 2009-11-01 - Alexander Brevig : Added isInState : boolean, requested by
    Henry Herman
91      || | 1.2 2009-05-18 - Alexander Brevig : enter and exit bug fix
92      || | 1.1 2009-05-18 - Alexander Brevig : Added support for cascaded calls
93      || | 1.0 2009-04-13 - Alexander Brevig : Initial Release
94      || #
95      */
```

2.5. ‹PID›

2.5.1. PID.cpp

```
1 /**********************************************************************************************
2  * Arduino PID Library - Version 1.2.1
3  * by Brett Beauregard ‹br3ttb@gmail.com› brettbeauregard.com
4  *
5  * This Library is licensed under the MIT License
6  **********************************************************************************************/
7
8 #include ‹PID.h›
9
```

```
10    /*Constructor (...)*********************************************************
11     *   The parameters specified here are those for for which we can't set up
12     *   reliable defaults, so we need to have the user set them.
13     ************************************************************************/
14    PID::PID(double* Input, double* Output, double* Setpoint,
15          double Kp, double Ki, double Kd, int POn, int ControllerDirection)
16    {
17      t_pid.start();
18      myOutput = Output;
19      myInput = Input;
20      mySetpoint = Setpoint;
21      inAuto = false;
22
23      PID::SetOutputLimits(0, 255);          //default output limit corresponds to
24                                 //the arduino pwm limits
25
26      SampleTime = 100;                 //default Controller Sample Time is 0.1
  seconds
27
28      PID::SetControllerDirection(ControllerDirection);
29      PID::SetTunings(Kp, Ki, Kd, POn);
30
31      lastTime = t_pid.read_ms()-SampleTime;
32    }
33
34    /* Compute() **********************************************************************
35     *   This, as they say, is where the magic happens.  this function should be
  called
36     *  every time "void loop()" executes.  the function will decide for itself whether
  a new
37     *  pid Output needs to be computed.  returns true when the output is
  computed,
38     *  false when nothing has been done.
39     **********************************************************************************/
40    bool PID::Compute()
41    {
42      if(!inAuto) return false;
43      unsigned long now = t_pid.read_ms();
44      unsigned long timeChange = (now - lastTime);
45      if(timeChange>=SampleTime)
46      {
47        /*Compute all the working error variables*/
48        double input = *myInput;
49        double error = *mySetpoint - input;
50        double dInput = (input - lastInput);
51        outputSum+= (ki * error);
```

```
52
53        /*Add Proportional on Measurement, if P_ON_M is specified*/
54        if(!pOnE) outputSum-= kp * dInput;
55
56        if(outputSum > outMax) outputSum= outMax;
57        else if(outputSum < outMin) outputSum= outMin;
58
59        /*Add Proportional on Error, if P_ON_E is specified*/
60         double output;
61        if(pOnE) output = kp * error;
62        else output = 0;
63
64        /*Compute Rest of PID Output*/
65        output += outputSum - kd * dInput;
66
67         if(output > outMax) output = outMax;
68        else if(output < outMin) output = outMin;
69         *myOutput = output;
70
71        /*Remember some variables for next time*/
72        lastInput = input;
73        lastTime = now;
74         return true;
75      }
76      else return false;
77    }
78
79    /* SetTunings(...)*************************************************************
80     * This function allows the controller's dynamic performance to be adjusted.
81     * it's called automatically from the constructor, but tunings can also
82     * be adjusted on the fly during normal operation
83     ******************************************************************************/
84    void PID::SetTunings(double Kp, double Ki, double Kd, int POn)
85    {
86      if (Kp<0 || Ki<0 || Kd<0) return;
87
88      pOn = POn;
89      pOnE = POn == P_ON_E;
90
91      dispKp = Kp; dispKi = Ki; dispKd = Kd;
92
93      double SampleTimeInSec = ((double)SampleTime)/1000;
94      kp = Kp;
95      ki = Ki * SampleTimeInSec;
96      kd = Kd / SampleTimeInSec;
97
```

```
98      if(controllerDirection ==REVERSE)
99      {
100       kp = (0 - kp);
101       ki = (0 - ki);
102       kd = (0 - kd);
103     }
104   }
105
106   /* SetTunings(...)*************************************************************
107    * Set Tunings using the last-rembered POn setting
108    ********************************************************************************/
109   void PID::SetTunings(double Kp, double Ki, double Kd){
110     SetTunings(Kp, Ki, Kd, pOn);
111   }
112
113   /* SetSampleTime(...) *********************************************************
114    * sets the period, in Milliseconds, at which the calculation is performed
115    ********************************************************************************/
116   void PID::SetSampleTime(int NewSampleTime)
117   {
118     if (NewSampleTime > 0)
119     {
120       double ratio  = (double)NewSampleTime
121                       / (double)SampleTime;
122       ki *= ratio;
123       kd /= ratio;
124       SampleTime = (unsigned long)NewSampleTime;
125     }
126   }
127
128   /* SetOutputLimits(...)****************************************************
129    *     This function will be used far more often than SetInputLimits.  while
130    *  the input to the controller will generally be in the 0-1023 range (which is
131    *  the default already,)  the output will be a little different.  maybe they'll
132    *  be doing a time window and will need 0-8000 or something.  or maybe they'll
133    *  want to clamp it from 0-125.  who knows.  at any rate, that can all be done
134    *  here.
135    **************************************************************************/
136   void PID::SetOutputLimits(double Min, double Max)
137   {
138     if(Min >= Max) return;
139     outMin = Min;
140     outMax = Max;
141
142     if(inAuto)
143     {
```

```
144         if(*myOutput > outMax) *myOutput = outMax;
145         else if(*myOutput < outMin) *myOutput = outMin;
146
147         if(outputSum > outMax) outputSum= outMax;
148         else if(outputSum < outMin) outputSum= outMin;
149       }
150     }
151
152     /* SetMode(...)****************************************************************
153      * Allows the controller Mode to be set to manual (0) or Automatic (non-zero)
154      * when the transition from manual to auto occurs, the controller is
155      * automatically initialized
156      ******************************************************************************/
157     void PID::SetMode(int Mode)
158     {
159       bool newAuto = (Mode == AUTOMATIC);
160       if(newAuto && !inAuto)
161       {  /*we just went from manual to auto*/
162         PID::Initialize();
163       }
164       inAuto = newAuto;
165     }
166
167     /* Initialize()****************************************************************
168      *  does all the things that need to happen to ensure a bumpless transfer
169      *  from manual to automatic mode.
170      ******************************************************************************/
171     void PID::Initialize()
172     {
173       outputSum = *myOutput;
174       lastInput = *myInput;
175       if(outputSum > outMax) outputSum = outMax;
176       else if(outputSum < outMin) outputSum = outMin;
177     }
178
179     /* SetControllerDirection(...)*************************************************
180      * The PID will either be connected to a DIRECT acting process (+Output leads
181      * to +Input) or a REVERSE acting process(+Output leads to -Input.)  we need to
182      * know which one, because otherwise we may increase the output when we
       should
183      * be decreasing.  This is called from the constructor.
184      ******************************************************************************/
185     void PID::SetControllerDirection(int Direction)
186     {
187       if(inAuto && Direction !=controllerDirection)
188       {
```

```
189         kp = (0 - kp);
190        ki = (0 - ki);
191        kd = (0 - kd);
192      }
193      controllerDirection = Direction;
194    }
195
196    /* Status Funcions*************************************************************
197     * Just because you set the Kp=-1 doesn't mean it actually happened.  these
198     * functions query the internal state of the PID.  they're here for display
199     * purposes.  this are the functions the PID Front-end uses for example
200     ******************************************************************************/
201    double PID::GetKp(){ return  dispKp; }
202    double PID::GetKi(){ return  dispKi;}
203    double PID::GetKd(){ return  dispKd;}
204    int PID::GetMode(){ return  inAuto ? AUTOMATIC : MANUAL;}
205    int PID::GetDirection(){ return controllerDirection;}
206
```

2.5.2. PID.h

```
1 #ifndef PID_h
2 #define PID_h
3 #define LIBRARY_VERSION 1.2.1
4
5 #include "mbed.h"
6
7 class PID
8 {
9   public:
10
11       //Constants used in some of the functions below
12       #define AUTOMATIC 1
13       #define MANUAL   0
14       #define DIRECT  0
15       #define REVERSE  1
16       #define P_ON_M 0
17       #define P_ON_E 1
18
19       //commonly used functions
    ***************************************************************************
20         PID(double*, double*, double*,      // * constructor.  links the PID to the Input,
    Output, and
21             double, double, double, int, int);//   Setpoint.  Initial tuning parameters are
    also set here.
22                                 //   (overload for specifying proportional mode)
```

```
23
24        void SetMode(int Mode);          // * sets PID to either Manual (0) or Auto (non-
  0)
25
26        bool Compute();                  // * performs the PID calculation.  it should be
27                                         //   called every time loop() cycles. ON/OFF and
28                                         //   calculation frequency can be set using SetMode
29                                         //   SetSampleTime respectively
30
31        void SetOutputLimits(double, double); // * clamps the output to a specific
  range. 0-255 by default, but
32                                               //   it's likely the user will want to change this
  depending on
33                                               //   the application
34
35
36
37       //available but not commonly used functions
  *********************************************************
38        void SetTunings(double, double,     // * While most users will set the tunings
  once in the
39                        double);           //   constructor, this function gives the user the option
40                                         //   of changing tunings during runtime for Adaptive control
41        void SetTunings(double, double,     // * overload for specifying proportional
  mode
42                        double, int);
43
44        void SetControllerDirection(int);    // * Sets the Direction, or "Action" of the
  controller. DIRECT
45                                         //   means the output will increase when error is positive.
  REVERSE
46                                         //   means the opposite.  it's very unlikely that this will be
  needed
47                                         //   once it is set in the constructor.
48        void SetSampleTime(int);         // * sets the frequency, in Milliseconds, with
  which
49                                         //   the PID calculation is performed.  default is 100
50
51
52
53       //Display functions ****************************************************************
54        double GetKp();                  // These functions query the pid for interal values.
55        double GetKi();                  //  they were created mainly for the pid front-end,
56        double GetKd();                  // where it's important to know what is actually
57        int GetMode();                   //  inside the PID.
58        int GetDirection();              //
```

```
59
60      private:
61       void Initialize();
62
63       double dispKp;          // * we'll hold on to the tuning parameters in user-
   entered
64       double dispKi;         //   format for display purposes
65       double dispKd;          //
66
67       double kp;             // * (P)roportional Tuning Parameter
68       double ki;            // * (I)ntegral Tuning Parameter
69       double kd;             // * (D)erivative Tuning Parameter
70
71       int controllerDirection;
72       int pOn;
73
74       double *myInput;          // * Pointers to the Input, Output, and Setpoint
   variables
75       double *myOutput;         //   This creates a hard link between the variables
   and the
76       double *mySetpoint;       //   PID, freeing the user from having to constantly
   tell us
77                          //   what these values are.  with pointers we'll just know.
78
79       unsigned long lastTime;
80       double outputSum, lastInput;
81
82       unsigned long SampleTime;
83       double outMin, outMax;
84       bool inAuto, pOnE;
85       Timer t_pid;
86     };
87     #endif
88
```

2.6. main.cpp

```
1 #include "mbed.h"
2 #include "FSM.h"
3 #include ‹MPU9250›
4 #include ‹BMP280›
5 #include ‹servos›
6
7
8 Serial pc(USBTX,USBRX);
9
```

```
10    bool ignition=false;
11    bool nothrust=false;
12    bool descend=false;
13
14    int actualState=0;
15
16    void init_config(){
17      config_servos();
18      config_imu();
19      config_baro();
20    }
21    void tvc(){
22      calc_tvc();
23    }
24    void altitude(){
25      calc_baro();
26    }
27    void recover(){
28      eject();
29    }
30    State fase1 = State(init_config);
31    State fase2 = State(tvc);
32    State fase3 = State(altitude);
33    State fase4 = State(recover);
34
35    FSM rocketflow = FSM(init_config);
36
37    void loop(){
38
39      if(rocketflow.isInState(init_config)){
40        actualState=0;
41      }else if(rocketflow.isInState(tvc)){
42        actualState=1;
43      }else if(rocketflow.isInState(altitude)){
44        actualState=2;
45      }else if(rocketflow.isInState(recover)){
46        actualState=3;
47      }
48
49      switch (actualState) {
50        case 0:
51          if(ignition){rocketflow.transitionTo(tvc);}
52        break;
53        case 1:
54          if(nothrust){rocketflow.transitionTo(altitude);}
55        break;
```

```
56          case 2:
57            if(descend){rocketflow.transitionTo(recover);}
58          break;
59          case 3:
60            //Nothing
61          break;
62        }
63        rocketflow.update();
64    }
65
66
67    int main(){
68      pc.printf("INIT\r\n");
69      while(1){loop();}
70    }
```

3. LIBRERÍAS EXTERNAS

La librería externa usada es la propia de la plataforma armbed. El contenido de dicha librería sus clases y métodos pueden consultarse en www.armbed.com