Microcontroladores y Electrónica de Potencia

Renzo Guarise

Trabajo integrador: Robot balancín

Mendoza - Argentina
2021

Índice

1. Introducción

En el presente proyecto se buscó realizar un robot balancín o también llamado péndulo invertido. La idea inicial fue realizar un proyecto mediante el cual se pueda mostrar los conocimientos adquiridos a lo largo de la materia y además poder investigar sobre un problema que se presenta en varios sistemas mecatrónicos. Para llevar acabo este proyecto se utilizó como sensor un acelerómetro y giróscopo para poder calcular la posición angular del sistema y así con esta realizar un control PID buscando que nuestro robot se pueda equilibrar. Para realizar el balanceo se usaron dos motorreductores, los cuales se controlan a través de un doble puente H y todo el sistema es comandado por un Arduino uno.

Del micro, se utilizó periféricos como: la **UART**, utilizada para comunicarse con el micro a través de bluetooth; **TIMERS**, utilizados para el control de los motorreductores; **I2C** o **TWI** para micros ATmegas, utilizado para la lectura y configuración del sensor; y la **GPIO** utilizada para el control de la dirección de los motores y para interrupciones.

Por otro lado, es importante nombrar las aplicaciones en las que se pueden encontrar el péndulo invertido para poder observar porque se estudia, y es que, a pesar de no estar en la primera plana de los grandes inventos, son vitales para la consecución de muchos otros. Este problema se puede encontrar:

- En los robots bípedos caminantes (Fig. 1), aunque a los humanos el hecho de andar de forma bípeda no aporta gran complejidad, en la robótica sí causa problemas que necesitan solución. Uno de ellos es la inestabilidad del robot cuando levanta uno de sus dos apoyos, por lo que la pierna apoyada se modela como un péndulo invertido permitiendo su estabilidad;
- En el transporte, el clásico segway, el cual es un péndulo invertido, cuyo control está basado en entradas sensoriales de giroscopios montados en su base y un sistema de control por computadora que mantiene el balance mientras las personas se pasean sobre el vehículo;
- En la industria Aeroespacial, donde se requiere el control activo de un cohete para mantenerlo en la posición vertical invertida durante su despegue, aquí, el ángulo de inclinación del cohete es controlado por medio de la variación del ángulo de la aplicación de la fuerza de empuje, colocada en la base de dicho cohete; también es muy utilizado para el modelado de brazo robóticos; entre otros.
- También es muy utilizado para el modelado de brazo robóticos; entre otros.

2. Esquema tecnológico

En la Fig. 2 se aprecia un esquema en bloque que muestra los distintos módulos que interactúan en el proyecto y como se comunican estos con el microprocesador. Se puede ver que el sensor MPU-5060 se comunica mediante la interfaz I2C con el micro, el módulo bluetooth mediante el protocolo UART, y el control de los motores mediante

Figura 1. Robot bipedo Asimo.

señales PWM. Mediante bluetooth es posible darle consignas de posición al robot, es decir, pedirle al robot que se desplace hacia adelante, por ejemplo.

En la Fig. 3 se presenta un esquema eléctrico que muestra el cableado del balancín, este esquema se realizó con el programa Proteus y es importante destacar que los elementos que se muestran en este no son las placas de desarrollo utilizados sino sus componentes principales, por ejemplo, en el esquema el Arduino Uno está representado por el micro Atmega 328p, y si bien, este es el micro de dicha placa, esta se encuentra compuesta por más componentes.

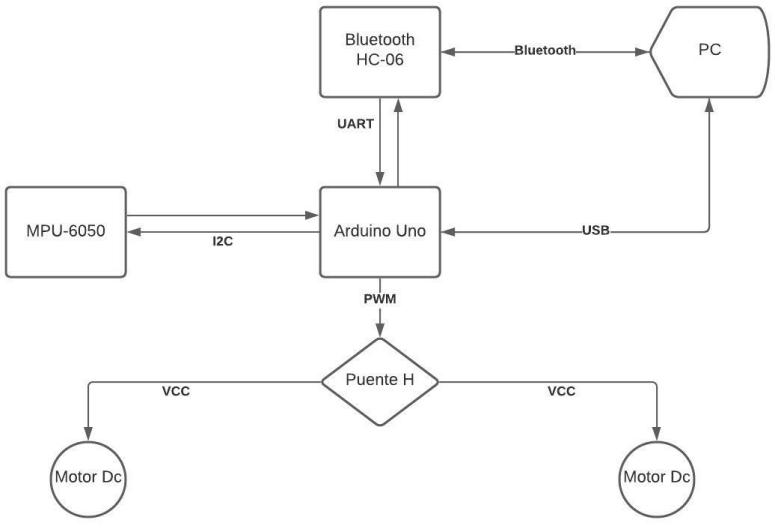

Figura 2. Esquema en bloque.

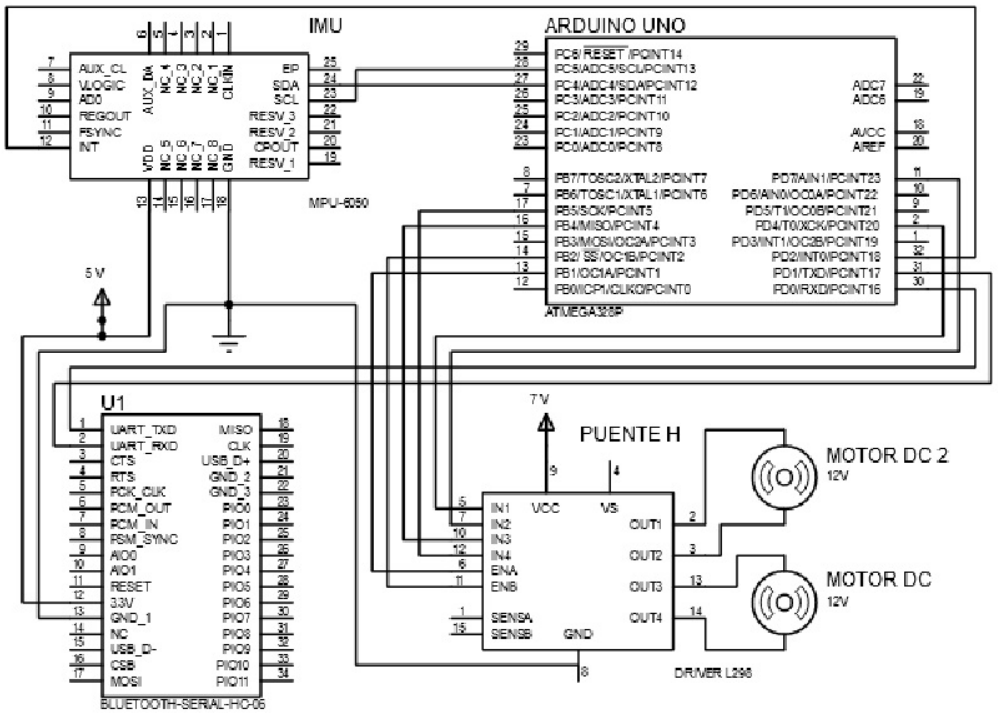

Figura 3. Esquema en bloque.

3. Módulos

3.1. Controlador de motores de corriente continua

Se usó el módulo controlador de motores L298N **??** que nos permitió controlar la velocidad y la dirección de dos motores de corriente continua de una forma muy sencilla, gracias a los dos Puente H que monta.

El rango de tensiones en el que trabaja este módulo va desde 3V hasta 35V, y una intensidad de hasta 2A. A la hora de alimentarlo hay que tener en cuenta que la electrónica del módulo consume unos 3V, así que los motores reciben 3V menos que la tensión con la que alimentemos el módulo.

En la Fig. 4 se muestra el modulo usado y un esquemático del mismo. Se puede ver que en la representación del puente H se modela a los transistores como interruptores y haciendo combinación de estos es posible elegir el sentido de giro de los mismos.

3.2. Módulo acelerómetro y giróscopo

Se utilizó el módulo MPU6050 **??**. Este es un avanzado procesador de movimiento basado en tecnología MEMS (Sistemas-Micro-Electro-Mecánicos) que incluye un acelerómetro de 3 ejes y un giroscopio de 3 ejes en un solo circuito integrado. Además, incluye la tecnología Digital Motion Processor (DMP), un “coprocesador” de movimiento que permite ejecutar algunos algoritmos esenciales y reducir la carga en el procesador principal.

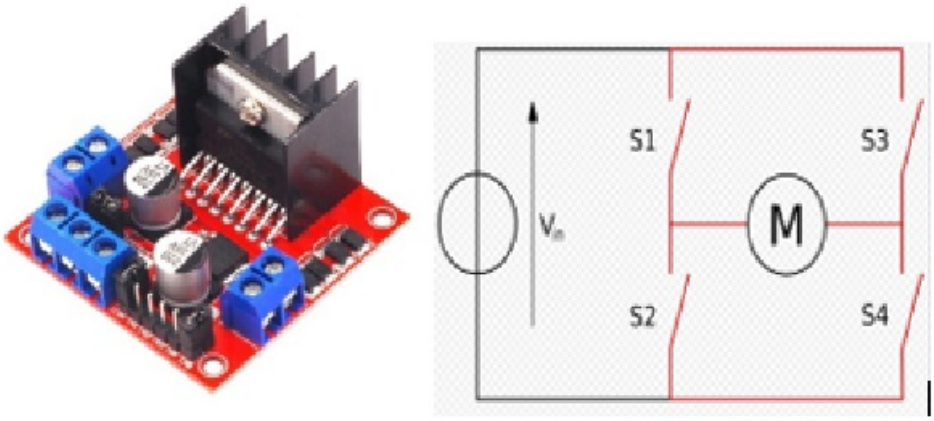

Figura 4. Driver L298N y esquemático.

Este se alimenta con 3V a 5V y para su comunicación con el microprocesador utiliza el protocolo I2C. Además, trae incorporado un conversor AD de 16bit.

3.3. Módulo bluetooth

Para la comunicación serie inalámbrica se utilizó el módulo bluetooth Hc-05 (Fig. 5), el cual ofrece el servicio de puerto serie, creando un enlace de datos transparente entre una PC, celular, tablet o cualquier dispositivo con Bluetooth maestro y el microcontrolador. Posee las siguientes características:

- Funciona como dispositivo maestro y esclavo bluetooth
- Configurable mediante comandos AT
- Bluetooth V2.0+EDR
- Frecuencia de operación: 2.4 GHz Banda ISM
- Seguridad: Autentificación y Encriptación

Figura 5. Módulo bluetooth Hc-05.

3.4. Motor reductor

Se utilizó dos motores (Fig. 6) con las siguientes características:

- TENSIÓN DE TRABAJO: 3V -12 VDC
- TORQUE MÁXIMO: 800 gF.cm
- VELOCIDAD SIN CARGA: 27 a 150 rpm
- CORRIENTE DE CARGA: 70mA a 3V (250mA MAX)
- RECDUTION RATIO: 1:48
- TAMAÑO: 7X2.2X1.8cm
- PESO: 35g

Figura 6. Motor reductor.

3.5. Microcontrolador

Se utilizó un Arduino Uno (Fig. 7) que es una placa microcontrolador basado en el microchip ATmega328P. Este es un micro de 8-bit que tiene una arquitectura RISC, una memoria flash de 32 KB, 1 KB de memoria EEPROM, 2 KB de SRAM, 23 GPIO, 32 registros de proceso general, tres TIMERS con modo de comparación, interrupciones internas y externas, programador de modo USART, SPI, 6-canales 10-bit Conversor A/D , temporizador ”watchdog”programable con oscilador interno, y cinco modos de ahorro de energía seleccionables por software. El dispositivo opera entre 1.8 y 5.5 voltios.

4. Funcionamiento general

En la Fig. 8 se muestra un diagrama de estado del robot:

- Parado: Este sería el estado de comienzo, en el que el sistema se encuentra en reposo.
- Balanceo: Comienza la etapa de equilibrio hasta que se pare el sistema o haya algun error o perturbación excesiva que haga caer la estructura.

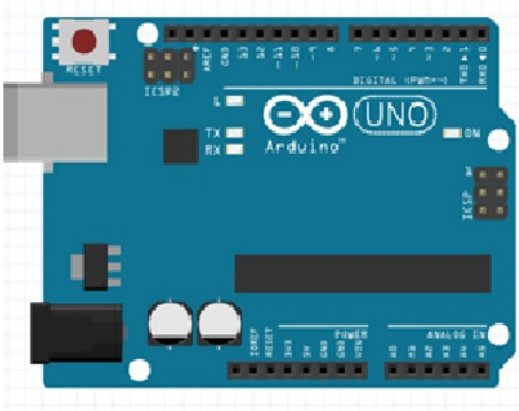

Figura 7. Arduino Uno.

- Movimiento: Cuando se le da una consigna de movimiento al robot.

Es importante notar que el sistema no pasa por un estado de calibracion del sensor, esto es asi porque en nuestro caso elegimos saltar esta parte y mediante la observacion elegir el angulo que el sensor nos da cuando el sistema se encuentra en reposo, puede ser conveniente configurar los offset de los sensores para que dicho angulo nos de cero en reposo pero nos parecio un desperdicio de recurso del micro realizar esta tarea ya que esto se puede salvar facilmente.

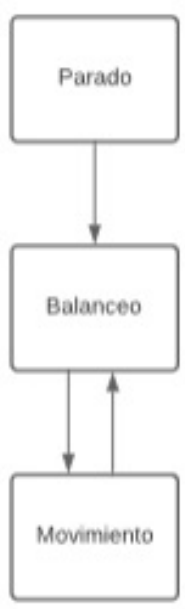

Figura 8. Diagram de estado.

Se ve en el grafico que el estado de BALANCEO es el principal y que siempre se vuelve a este ya que de otra manera el robot se caería.

En la Fig. 9 se presenta un diagrama de flujo, en donde se esquematiza un poco como es el proceso del programa. En primer lugar, se produce el INICIO, en donde

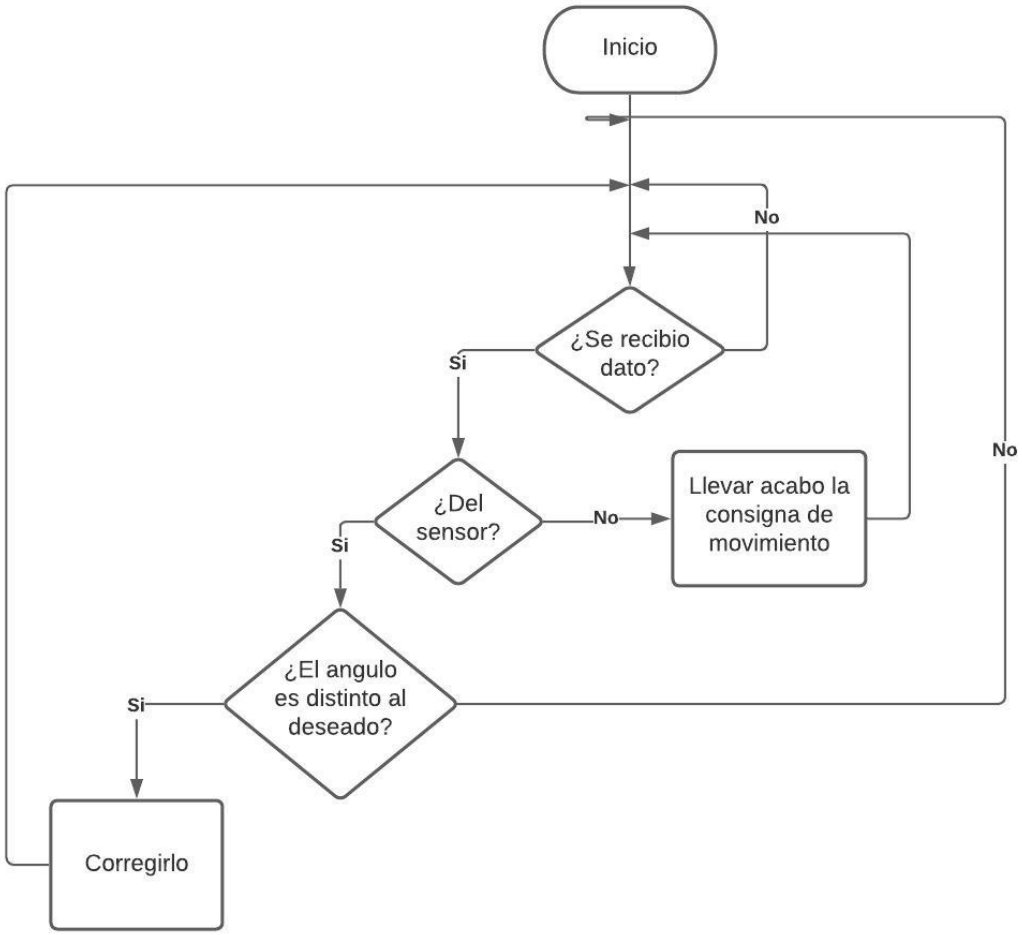

Figura 9. Diagram de flujo.

se configuran los periféricos (UART, I2C, TIMER), el sensor y las interrupciones. Luego en el programa se responde a la pregunta, ¿Se recibió algún dato? mediante interrupciones, para esto se configuró a la IMU para que lance una interrupción cada vez que en sus registros se tuvieran datos para ser leídos, así mediante una rutina de servicio se atiende a esta. Es aquí donde se produce la tarea o estado principal del programa, el BALANCEO, a partir de los datos leídos de la IMU se calcula el ángulo de inclinación del robot, para esto se empleó un filtro complementario, mediante el ángulo obtenido se responde a la pregunta ¿Este es el ángulo deseado? (Angulo en el cual la posición del robot es vertical, que en nuestro caso no es cero porque no se calibró el sensor mediante los offset), de no ser asi, mediante un algoritmo de control se intenta corregir este, el cual actúa sobre los motores del robot. Por otro lado, el dato recibido puede ser una consigna de movimiento, para esto se habilitó la interrupción de la UART que se produce cuando hay un dato en el buffer de esta para ser leído, así mediante otra rutina de servicio se atiende esta interrupción y se lleva a cabo este movimiento. Este ciclo se repite indefinidamente hasta que nuestro robot entre en el estado PARADO.

5. Programación

5.1. I2C

Para la comunicación con la IMU fue necesario configurar el protocolo de comunicación I2C del micro, para realizar esto se procedió a leer el manual del micro **??**.

Algo ventajoso es que para implementarlo el único hardware externo necesario son dos resistencias pull-up como se puede ver en la Fig. 10. Este permite la conexión de varios dispositivos a la vez, siendo la única limitación la dirección de 7 bits y la capacitancia límite de 400 pF del bus. Es importante destacar también que la interfaz esta compuesta por transistores a colector abierto, por lo que el cero es el estado dominante, es decir que para poner la línea en alto todos los dispositivos tienen que estar en un estado indiferente o tercer estado, de esto se concluye que para que la comunicación se de todos los dispositivos de la línea tienen que estar encendidos.

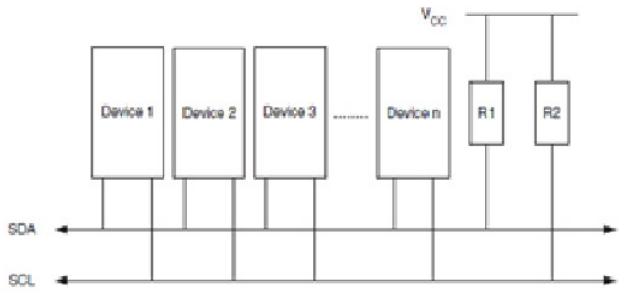

Figura 10. Esquema conexión protocolo I2C.

5.1.1. Transferencia de datos

A continuación se especificará como es la transferencia de datos en este protocolo.

Todo bit transferido (Fig. 11) es acompañado por un pulso de la línea de reloj en alto, la única excepción a esta regla es cuando se genera un bit de START o STOP (Fig. 12).

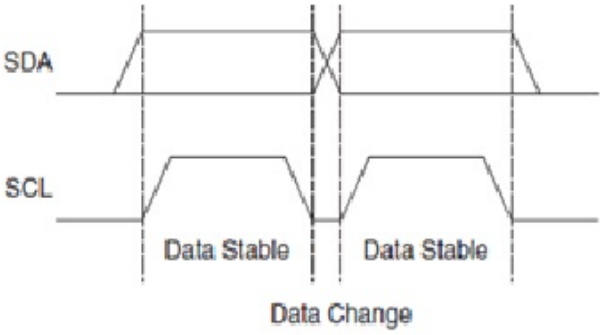

Figura 11. Transmisión de bits.

El maestro inicia y termina la transmisión de datos. La transmisión inicia cuando el maestro lanza la condición de inicio en el bus y termina cuando este lanza la condición

de stop. Entre un START y un STOP el bus es considerado ocupado, y ningún otro maestro debería tratar de controlar el bus. Puede ocurrir también el caso especial que se lance un nuevo STAR antes que se de el STOP, esto se conoce como REPEATED START y se da cuando el maestro quiere iniciar otra transferencia sin perder el control del bus.

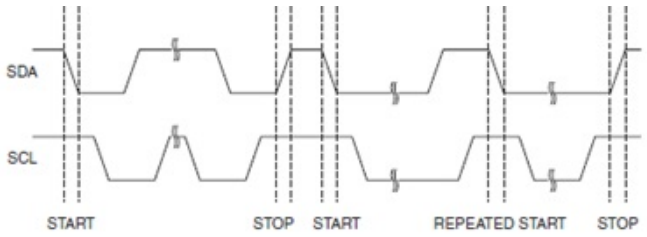

Figura 12. Condición de START y STOP .

5.1.2. Forma del paquete de datos

Primero analizaremos la forma del paquete de direcciones transmitido (SLA+R/W), este tiene una longitud de 9 bit, 7 son bit de la dirección del dispositivo, uno el READ/WRITE bit y el ultimo el bit de acknowledge que indica al maestro que el esclavo recibió correctamente la información. El acknowledge se da poniendo la SDA en bajo en el noveno ciclo del SCL (Fig. 13).

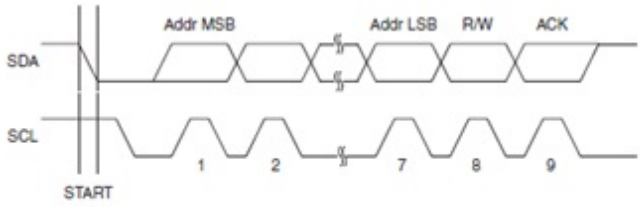

Figura 13. Transmisión paquete SLA+R/W .

Todo paquete de datos transmitido en I2C tiene 9 bits de longitud, consistiendo en 8 bits de datos y un bit de acknowledge. En la Fig. 14 se puede ver como es manejo de las líneas de SDA y SCL por parte del maestro y el esclavo, es importante notar que la línea SCL es únicamente controlada por el maestro y la línea SDA es controla principalmente por el maestro y el esclavo únicamente toma control de esta cuando manda el bit de acknowlegde.

Así la transmisión de datos consiste básicamente en una condición de START, un paquete SLA+R/W, uno o más paquetes de datos y una condición de STOP.

Otros temas a tener en cuenta en este protocolo son la Sincronización y la Arbitración. Estos son dos problemas que se dan en este protocolo ya que es multimaestro y se resuelven fácilmente gracias a la AND lógica que se produce en las líneas al tener una configuración de colector abierto.

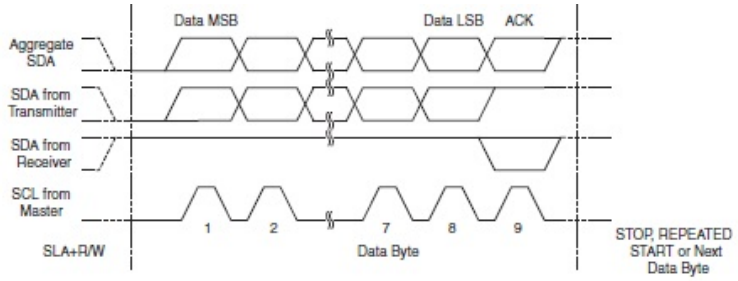

Figura 14. Transmisión paquete de datos.

5.1.3. I2C programación – Escritura

Primero mediante la función **Set_baudrate()** escribimos el registro **TWBR** que juntos con los bits correspondientes al prescaler del registro **TWSR** nos darán el baudrate.

```
void Set_baudrate(i2c *ptr_i2c){

        TWBR= (uint8_t)((((F_CPU / ptr_i2c->baudrate) / ptr_i2c->
           prescaler) - 16 ) / 2);
        TWSR= (ptr_i2c->prescaler-1)<<TWPS0;

}
```

Luego para escribir un byte mediante I2C se creó la función **write_byte()**:

```
void write_byte(uint8_t data, uint8_t disp_dir, uint8_t adress){
        while(!Start_i2c (W(disp_dir)));
        Write_i2c(adress);
        Write_i2c(data);
        Stop_i2c();
}
```

La cual hace uso de:

-
```
uint8_t Start_i2c(uint8_t disp_dir){

TWCR = 0;
TWCR =(1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
while (!(TWCR &(1<<TWINT)));
if ((TWSR & 0xF8) !=TW_START) return 0;
TWDR = disp_dir;
TWCR = (1<<TWINT) | (1<<TWEN);
while (!(TWCR &(1<<TWINT)));
uint8_t twst = TW_STATUS & 0xF8;
if ((twst!=TW_MT_SLA_ACK)&&(twst!=TW_MR_SLA_ACK))
   return 0;

return 1;

}
```

  La cual se encarga de iniciar la comunicación por I2C, está en primera instancia pone el registro TWCR en cero, luego siguiendo las instrucciones del manual

limpia el bit correspondiente al flag de las interrupciones (**TWINT**) poniéndolo en 1, pone en uno el bit **TWSTA** para dar la condición de START y por último habilita las operaciones de la TWI poniendo en uno el bit **TWEN**. Luego espera a que el flag se ponga en 1 y lee el registro de estado **TWSR**, para verificar que el flag se levantó por la condición de START, por último se procese a mandar la trama SDA+W, para esto se pone en el registro de datos **TWDR** la dirección del dispositivo más el bit de escritura y se setea los bits**TWINT** y **TWEN** en el registro **TWCR**, se verifica nuevamente que la información haya sido enviado correctamente mediante el flag **TWINT** y el registro **TWSR**.

```
uint8_t Write_i2c(uint8_t data){

TWDR = data;
TWCR = (1<<TWINT)|(1<<TWEN);
while (!(TWCR &(1<<TWINT)));
if ((TWSR & 0xF8) !=TW_MT_DATA_ACK) return 0;
return 1;

}
```

Esta función nos permite mandar los datos por I2C siguiendo los pasos indicados en el manual. Como ya se vio en la función anterior para mandar cualquier dato es necesario limpiar el flag **TWINT** y habilitar **TWI** con **TWEN** en el registro **TWCR**. Luego se hace la verificación **TWSR** para garantizar que el flag se levantó por un Acknowlegde

```
void Stop_i2c(void){

TWCR =(1<<TWINT)|(1<<TWEN)|(1<<TWSTO);

}
```

Para último, esta función nos permite mandar la condición de STOP para esto setamos el bit **TWSTO**, **TWINT** y **TWEN**.

En la Fig. 15 se muestra un diagrama de estado por los cuales pasa muestro algoritmo para transmitir un dato por I2C:

- START: se da la condición de inicio para empezar la comunicación.
- VERIFICACION: se comprueba que todas los datos y condiciones (START, REPEAT STAR, etc.) hayan sido correctamente recibidos.
- ESCRITURA: transmisión de datos al esclavo.
- STOP: se da fin a la comunicación.

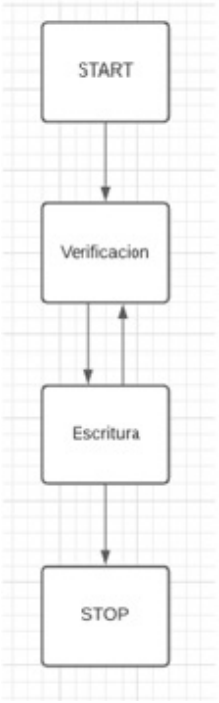

Figura 15. Diagrama de estado. Escritura I2C.

En la Fig. 16 se muestra un diagrama de flujo que sintetiza el proceso para transmitir un dato por I2C.

5.1.4. I2C programación – Lectura

Mediante la función **read_byte()** podemos leer un byte desde el maestro.

```
uint8_t read_byte(uint8_t disp_dir, uint8_t adress){

uint8_t data;
while(!Start_i2c(W(disp_dir)));
Write_i2c(adress);
Start_i2c(R(disp_dir));
data=Read_i2c(0);
Stop_i2c();

return data;

}
```

Para entender esta función comentaremos como es el proceso de lectura que difiere un poco con el proceso de escritura. Para comenzar es necesario establecer la condición de START y como primera instancia mandar la trama SDA+W, ya que para la lectura de datos primero es necesario realizar la escritura de la dirección de memoria a donde se van a leer los datos. Esto se puede ver en el código anterior, primero se hace uso de la función **Start_i2c(W(disp_dir))** y se manda la trama SDA+W y luego se escribe la dirección **Write_i2c(adress)**. Luego se empieza el proceso de lectura para esto establecemos un REPEAT START y mandamos la trama SDA+R y leemos los datos mediante

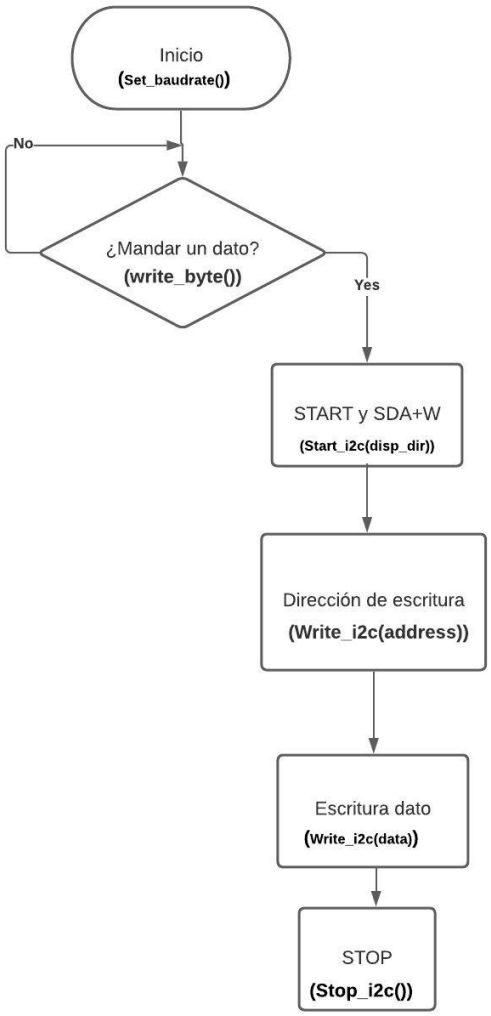

Figura 16. Diagrama de flujo. Escritura I2C.

la función **Read_i2c(0)**. Por último, establecemos la condición STOP. A continuación explicaremos las funciones que intervienen en **read_byte()**:

- Start_i2c() : ya descripta anteriormente.
- Write_i2c : ya descripta anteriormente.
- Stop_i2c(): ya descripta anteriormente.
-
```
uint8_t Read_i2c(uint8_t ack){

TWCR = (1<<TWINT) | (1<<TWEN) | (ack<<TWEA);

while( !(TWCR & (1<<TWINT)) );

return TWDR;

}
```

Esta función realiza la lectura del registro **TWDR**, para habilitar la lectura se debe limpiar el flag **TWINT**, habilitar la TWI **TWEN** y por último dar un acknowlegde o no, si se va a leer un solo bit debo poner en cero el bit **TWEA** para dar un not acknowlegde indicando que ese es el único bit que se va a leer.

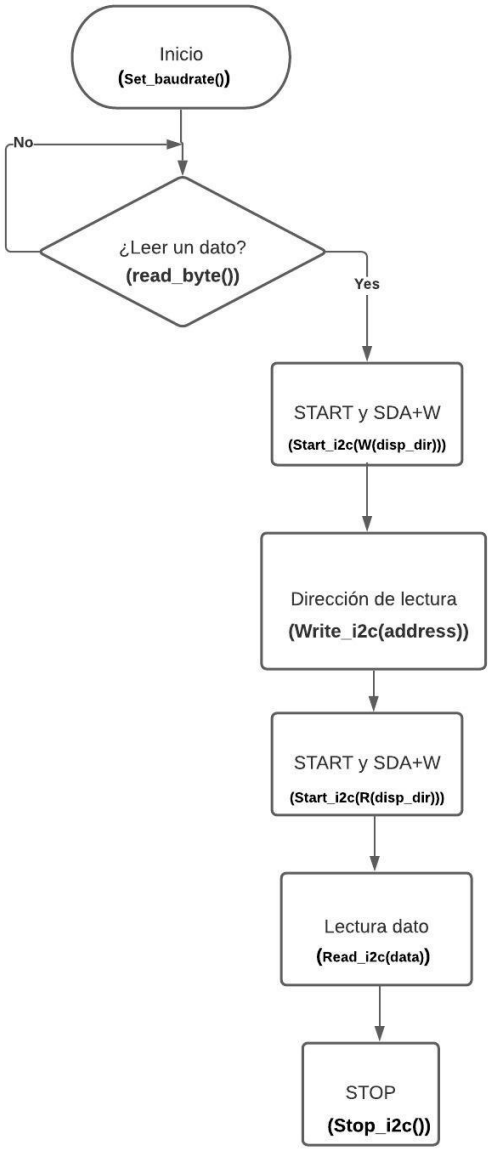

Figura 17. Diagrama de flujo. Lectura I2C.

En la Fig. 17 se muestra un diagrama de flujo que muestra el proceso para leer un dato por I2C.

5.2. USART

En este proyecto se utiliza para comunicarse mediante un módulo bluetooth con otro dispositivo externo. Para implementar este protocolo de comunicación no es ne-

cesario contar con ningún hardware externo, pero dependiendo la aplicación puede ser necesaria una interfaz eléctrica.

5.2.1. Formato de la trama

Se forma por un carácter de bits de datos con bits de sincronización (bit start y stop), y un bit opcional de paridad para verificación de errores. La USART acepta las siguientes combinaciones.

- 1 bit start
- 5, 6, 7, 8 o 9 bit de datos
- 1 bit de paridad o no
- 1 o 2 bits de stop

La trama empieza obviamente con el bit de start, seguida por el bit menos significativo de dato. Luego los siguientes bits de dato son enviados, terminando por el mas significativo. Si está habilitado se envía el bit de paridad y después el bit de stop. Cuando se envió la trama completa esta puede ser seguida por una trama nueva, o si no se pone la línea de comunicación en un estado desocupado (alto). En la Fig. 18 se muestra la trama de comunicación típica:

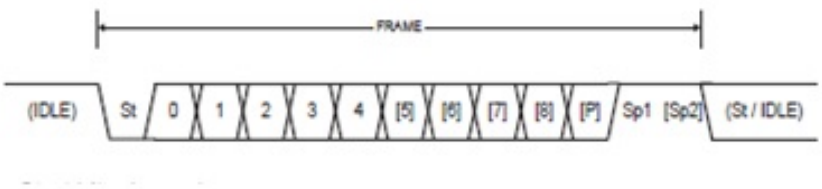

Figura 18. Trama UART.

5.2.2. USART programación

Lo primero que se realiza es inicializar la USART para esto se utilizó la función:

```
void configuracion_uart(unsigned int Int_UartRx, unsigned int
    Int_UartTx, double brate)
{
        UBRR0=F_CPU/16/(brate-1);
        UCSR0B|=((1<<RXEN0)|(1<<TXEN0));
        UCSR0C=(3<<UCSZ00);
        stdout=stdin=&uart_io;
        if (Int_UartRx)
        {
                UCSR0A |=(1<<RXC0);
                UCSR0B|=(1<<RXCIE0);
        }
        if (Int_UartTx)
        {
                UCSR0A |=(1<<TXC0);
```

```
                UCSR0B|=(1<<TXCIE0);
        }
}

}
```

En esta función lo primero que se realizar es establecer el baudrate escribiendo el registro **UBRR0**, luego se habilita la transmisión y recepción de datos mediante los bits **RXEN0** y **TXEN0** del registro **UCSR0B**. Luego, la función nos permite habilitar la interrupción que nos indica cuando se recibió un dato **RXCIE0** y cuando se completó la transmisión de un dato **TXCIE0**.

Por último, de la escritura del registro **UCSR0C** se concluye que la comunicación tendrá las siguientes características:

- Asíncrona
- Bit de paridad deshabilitado
- Un bit de stop
- 8-bit de datos

5.2.3. Recepción de datos

Para la lectura de datos recibidos por la USART se procedió a habilitar la interrupción que nos indica cuando hay un dato para ser leído (**RXCIE0**).

```
ISR(USART_RX_vect)
{
        char dato;
        dato=fgetc();
        switch(dato)
        {
                case ':':
                cmd=1;
                contcomando=0;
                break;
                case '/':
                cmd=1;
                contcomando=0;
                break;
                case 13:
                if (cmd)
                {
                        comando[contcomando]=0;
                        interpretecomando();
                        cmd=0;
                }
                break;
                case 8:
                if (contcomando>0)
                {
                        contcomando--;
                }
```

```
                break;
                default:
                if(contcomando<30)
                {
                        if (cmd)
                        {
                                comando[contcomando++]=dato;
                        }
                }
                break;
        }
        UCSR0A |= (1<<RXC0);
}
```

Para leer el dato recibido se creó la función **mi_getc(FILE *stream)** la cual mediante una etiqueta la redefinimos:

```
#define fgetc()  mi_getc(&uart_io)
int mi_getc( FILE *stream)
{
        while(!(UCSR0A & (1<<RXC0)));
        return UDR0;
}
```

Así redefinimos la función **fgetc()**, función nativa, por una nuestra. La variable **uart_io** se define como:

```
FILE uart_io = FDEV_SETUP_STREAM(mi_putc,mi_getc,_FDEV_SETUP_RW);
```

Donde la función **FDEV_SETUP_STREAM** nos permite establecer un canal que es válido para operaciones de stdio (operación de entrada-salida estándar) y nos devuelve este canal en forma de **FILE**.

Vemos que la función **mi_getc(FILE *stream)**, espera que el flag de la interrupcion se encuentre en alto y devuelve el dato recibido.

Luego se procese a analizar el dato recibido, donde se espera una trama de la forma : $dato$ o $/dato$, este análisis se realiza mediante la sentencia **switch()**, así el ultimo **case** al que entrara nuestra función será **case** 13, lo que indica que se ha recibido toda la información y se llama a la función **interpretecomando()** que básicamente a partir de la información recibida realiza las tareas indicadas.

En la Fig. 19 se muestra un diagrama de flujo que nos ilustra en forma sintética como es el proceso de recibir un dato por USART.

No se ha detallado el proceso de transmisión de datos por USART ya que este se realiza mediante la función nativa **printf()**.

5.3. Timer

El timer es un periférico incluido en todos los micros que nos permite realizar un control del tiempo con un determinado nivel de precisión. El objetivo puede ser generar intervalos de tiempo o medir el tiempo transcurrido entre eventos.

Es básicamente un contador que se puede leer o escribir, cuya entrada de reloj puede conectarse a distintas fuentes de pulsos. Cuando los pulsos son eventos en un pin de entrada, funciona como un simple contador (ej. para contar pulsos de un encoder incre-

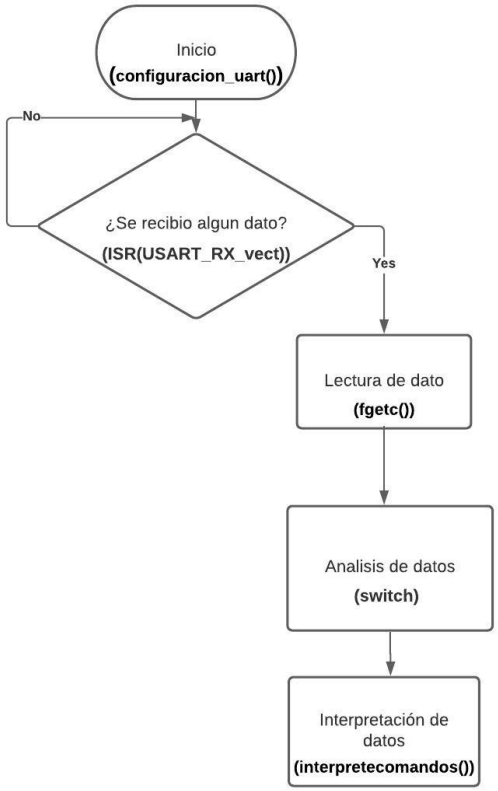

Figura 19. Diagrama de flujo transmisión de datos USART.

mental). Cuando los pulsos provienen de un reloj de frecuencia conocida, normalmente el “clock” interno, funciona como timer, porque provee referencia temporal.

5.3.1. Programación

Para la generación de la señal PWM necesaria para el control por potencia de los motorreductores elegimos el TIMER0. Este es un timer de 8 bits.

Para la inicialización del TIMER0 creamos la variable Timer0 del tipo **T0** que es un tipo de dato que nosotros definimos como:

```
typedef struct{
        T_OUT OCA0;
        T_OUT OCB0;
        modeT0y2 modo0;
        CL_pres0y1 Pres;
}T0;
```

Así **T0** contiene toda la información relevante de la configuración del TIMER0.

```
T0 Timer0;
Timer0.modo0=FastPWM01;
Timer0.OCA0=outClear;
Timer0.OCB0=outClear;
Timer0.Pres=CL_NOpres;
configTimer0(&Timer0);
```

Elegimos el modo FASTPWM para la generación de pulso, este se muestra en la Fig. 20.

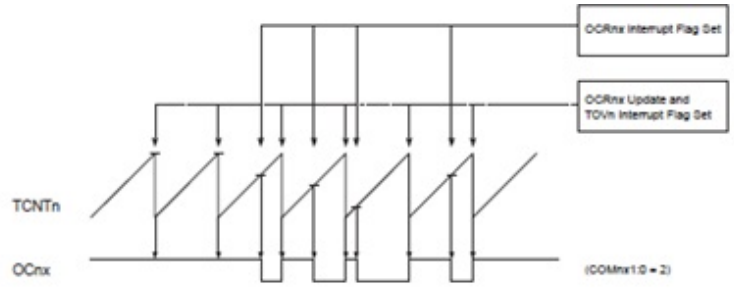

Figura 20. Modo FASTPWM. Formato salida OC0x.

El modo de la salida por comparación es no invertido (Clear), por lo que la salida **OCA0** y **OCB0** es puesta en bajo cuando el registro **TCNT0** iguala a **OCR0** y puesta el alto cuando se reinicia el contador **TCNT0**. Por último el prescaler es puesto en 1.

Para el manejo de los registros de control se ultima la función **configTimer0()**:

```
void configTimer0(T0 *ptr_T0){

        if (ptr_T0->OCA0) PD6_salida;
        if (ptr_T0->OCB0) PD5_salida;
        TCCR0A = (ptr_T0->OCA0<<COM0A0)|(ptr_T0->OCB0<<COM0B0)|(
            ptr_T0->modo0&0x03);
        TCCR0B = (((ptr_T0->modo0)<<1)&0x08)|ptr_T0->Pres;

}
```

En primera instancia vemos que la función configura como salida los pines correspondientes a **OCA0** y **OCB0** si estos se habilitaron. Luego se escriben los registros **TCCR0A** y **TCCR0B**, siguiendo la configuración elegida.

5.4. MPU6050

Se explicará como se programó la IMU.

5.4.1. Programación

Para esto se tomo como referencia la librería desarrollada por Jeff Rowberg.**??**

En la librería creada por nosotros (mpu_6050) se ven algunas de las funciones mas importante. Hay que destacar que la librería creada por Jeff Rowberg carga un firmware a la DPM de la IMU lo que permite que este se encargue del acondicionamiento de las señales muestreadas del giróscopo y acelerómetro. Así leyendo la fifo de la IMU se puede obtener los quaterniones calculados en la DPM y mediante estos obtener las orientaciones angulares. Para la inicializamos de la IMU creamos la función **initialize_mpu(MPU6050 *MPU,INT_MPU *interrup)**, esta configura las distintas características del módulo y sus interrupciones.

```
void initialize_mpu(MPU6050 *MPU,INT_MPU *interrup){

        MPU->smplrt_div=4;
        MPU->Filtro=DLPF_BW_42;
        MPU->EXT_SYN=EXT_SYNC_DISABLED;
        MPU->GYRO_SCALE=GYRO_FS_250;
        MPU->ACCEL_SCALE=ACCEL_FS_2;
        MPU->CLSOURCE=CLOCK_PLL_XGYRO;

        interrup->I2C_MST_INT_EN=false;
        interrup->FIFO_OFLOW_EN=false;
        interrup->DATA_RDY_EN=true;
        interrup->CTR.INT_LEVEL=true;
        interrup->CTR.INT_OPEN=false;
        interrup->CTR.LATCH_INT_EN=true;
        interrup->CTR.INT_RD_CLEAR=true;
        interrup->CTR.FSYNC_INT_EN=false;
        interrup->CTR.FSYNC_INT_LEVEL=false;
        interrup->CTR.I2C_BYPASS_EN=false;


        set_DLPF(MPU);
        setClockSource1(MPU);
        set_GYROSCALE(MPU);
        set_ACCELSCALE(MPU);
        set_samplerate(MPU);
        setTempSensorEnabled(false);
        setSleepEnabled(false);

        int_ctrl(interrup);
        int_enable(interrup);

}
```

Como vemos se crearon las variables **MPU6050** y **INT_MPU** que son del tipo struct las cuales contienen toda la información de las configuraciones en el MPU6050.

Las configuraciones realizadas son:

- Frecuencia de muestra: 200 Hz.
- Filtro pasa bajos BW 42 Hz.
- Escala acelerómetro 2g.
- Escala giróscopo 250 °/s.
- Referencia del reloj, se elije al eje x del giróscopo.
- Se habilitan las interrupciones para informar que hay un dato listo.
- Se setea nivel bajo de manera activa de la interrupción.
- El pin de interrupción es configurado como push-pull.
- La interrupción se limpia cuando se lee algún registro.

Se eligió un filtro pasa bajo con un bando de ancho de 42 Hz porque con los de menor BW se concluyó que se perdería mucha información. La frecuencia de muestreo se eligió de manera que sea aproximadamente 2,5 veces la frecuencia de Naysquit (fn=42Hz), así no caemos en errores por alaising.

Se referencia el reloj del sistema a un eje del giróscopo porque el manual dice que es recomendado para incrementar la estabilidad de la IMU.

Una vez que se ha elegido las configuraciones de la IMU se llama a las distintas funciones nombradas en el código, que mediante las funciones para escritura por I2C, ya descriptas, escriben los registros de control del MPU-6050.

5.5. Main

Para terminar, describiremos la función principal del programa. Esto lo haremos por partes.

- Primero configuramos la GPIO:

```
PD2_Entrada;
PD2_Encender;
```

Estas etiquetas están definidas en main.h:

```
PD2_Entrada (DDRD|=(0<<DDD2))
PD2_Encender (PORTD|=(1<<PORTD2))
```

Esta es la entrada de interrupción para la IMU. El registro **DDRx** nos permite elegir la dirección de los pines de la GPIO y el registro **PORTx** nos permite establecer su estado. Al establecer el PD2 como entrada, escribiendo en 1 el bit **PORTD2** del registro **PORTD** nos permite habilitar una resistencia pull-up para evitar que entre ruido por este pin.

- Luego configuramos la interrupción de la IMU:

```
EIMSK |=(1<<INT0);
EICRA|=((1<<ISC10)|(1<<ISC11));
EIFR |=((1<<INTF0));
```

El registro **EIMSK** nos permite habilitar la interrupción externa (**INT0**), para esta también es necesario habilitar las interrupciones a nivel global mediante la escritura del bit correspondiente en el registro **SREG** o la función **sei()**. Luego mediante el registro **EICRA** seteamos para que evento va a saltar la interrupción en nuestro caso elegimos el flanco de subido. Por último, limpiamos el flag correspondiente en el registro **EIFR**.

- Configuramos el TIMER0:

```
T0 Timer0;
Timer0.modo0=FastPWM01;
Timer0.OCA0=outClear;
```

```
Timer0.OCB0=outClear;
Timer0.Pres=CL_NOpres;
configTimer0(&Timer0);
```

- Configuramos la USART:

```
configuracion_uart(1,0,9600);
```

- Configuramos el TWI:

```
i2c ptr_i2c;
ptr_i2c.baudrate=100000;
ptr_i2c.prescaler=Pres1;
disp_dir=ADDR(0x68);
Set_baudrate(&ptr_i2c);
```

- Configuramos la MPU:

```
MPU6050 MPU;
INT_MPU interrup;
initialize_mpu(&MPU,&interrup);
```

- Configuramos el control PID :

```
pid_init(uint8_t P_ON, uint8_t D_ON, uint8_t I_ON,double Kp,
    double Kd, double Ki, double paso, double *input1, double *
    setpoint1, double *output1) ;
limit(double max, double min) ;

setpoint=parado;
```

La función **pid_init()** nos permite configurar el PID, eligiendo que tipo de control llevaremos a cabo, las constantes del PID y el setpoint. La función **limit()** nos permite establecer el valor máximo y mínimo que puede alcanzar nuestro control. Así en nuestro caso que el TIMER es de 8 bits pondremos max=255 y min=-255, si queremos que los motores se alimenten con la máxima tensión cuando sea necesario. Esto nos permitirá controlar la máxima velocidad de los motores.

- Habilitamos las interrupciones globales:

```
sei();
```

- Por último, analizamos el loop:

```
while (1)
   {

       if (Flag_IMU)
       {
```

```
                    get_gyro_gy(&gy);
                    get_ax_az(&ax,&az);

                    ax_f=ax*(9.81/16384.0);
                    az_f=az*(9.81/16384.0);
                    gy_f=gy*(250.0/32768.0);

                    /* FILTRO COMPLEMENTARIO */
                    input = 0.98*(ang_y_prev-gy_f*0.005) + 0.02*(
                        atan2(ax_f,az_f)*(180.0/3.14));
                    ang_y_prev=input;

                    control_pid();

                    cambio_dir();


                    #ifdef Angulo

                            printf("%.2f,%d\n\r",input,setpoint);

                    #endif // Angulo

                    Flag_IMU=0;
            }
}
```

La variable **Flag_IMU** se setea en 1 cuando se produce un evento en la interrupción de la IMU. Mediante las funciones **get_gyro()** y **get_accel()** obtenemos las valores del giróscopo y acelerómetro, estas funciones simplemente hacen uso de las funciones de lectura de la librería i2c.h para leer los registros correspondientes en la IMU. Luego de los datos obtenidos se procede a escalarlos, teniendo en cuenta el escalado configurado en el acelerómetro y giroscopio se obtiene:

- $EscalaGYRO/resolucionDAC = (2 * 9{,}81/32768)(m/(s^2 * bit))$
- $EscalaACCEL/resolucionDAC = (250{,}0/32768)(/(s * bit))$

Luego se procede a realizar un filtro complementario. La necesidad de este filtro nace de que no se puede utilizar únicamente la medida obtenida del acelerómetro o del giroscopio, ya que el acelerómetro es susceptible a las aceleraciones producto de otros movimientos o fuerzas externas, pero en tiempos largos o reposo este no acumula errores y si trabajamos con el giroscopio solo, si bien este no es susceptible a fuerzas externas, tiene un error llamado DRIFT,el cual se debe a que al integrar la velocidad angular y sumar el ángulo inicial hay un error producto de la mala medición del tiempo o del ruido en la lectura del MPU, el error por más pequeño que sea, se va acumulando en cada iteración y creciendo. Con el tiempo el drift es muy grande y nos sirve solo para mediciones de tiempos cortos.

La ecuación para calcular el ángulo usando el filtro de complemento es:

$$ángulo = 0.98(ángulo + \omega_{giroscopio}\, dt) + 0.02(ang_{acelerómetro})$$

Figura 21. Filtro complementario.

De esta forma el ángulo del acelerómetro está pasando por un filtro pasa bajos, amortiguando las variaciones bruscas de aceleración; y el ángulo calculado por el giroscopio tiene un filtro pasa altos teniendo gran influencia cuando hay rotaciones rápidas.

Luego se procede a hacer el control PID mediante la función **control_pid()** función definida en PID.h.

Por ultimo se realiza la acción necesaria en los motores para mantener el equilibrio mediante la función **cambio_dir()**:

```
void cambio_dir(void){

        if (*myoutput<0){
                PD4_Apagar;
                PD7_Encender;
                PB5_Encender;
                        PB4_Apagar;

                OCR0A=-*myoutput;
                        OCR0B=-*myoutput;
                TCNT0=0;

        }
        if (*myoutput>0){

                PD7_Apagar;
                PD4_Encender;
                PB5_Apagar;
                        PB4_Encender;

                OCR0A=*myoutput;
                OCR0B=*myoutput;
                TCNT0=0;

        }
}
```

Vemos que esta función actúa sobre la dirección de los motores mediante los pines correspondientes de la GPIO y sobre la velocidad de los motores escribiendo los registros **OCR0A** y **OCR0B**.

En la Fig. 22 se muestra un diagrama de flujo de la función main().

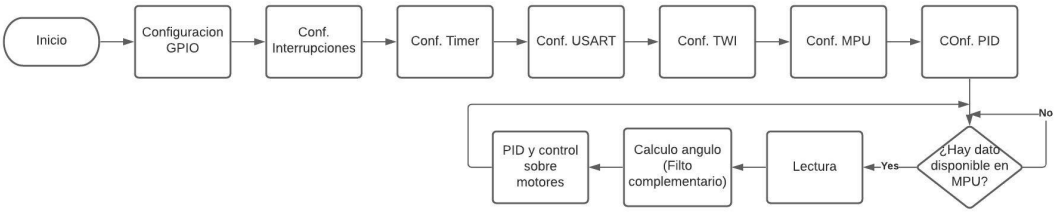

Figura 22. Diagrama de flujo. Función main().

6. Etapas de montaje y ensayos realizados

6.1. Montaje

En la Fig. 23 vemos el robot ya montado. Como estructura se utilizó policarbonato ya que este posee cierta rigidez y es muy liviano, vemos que esta es una estructura baja, ya que se quiso que el centro de masa se encuentre lo mas cercano posible al eje de accion de los motores para facilitar el equilibrio, de esta manera el momento realizado por el peso es menor porque el brazo de palanca es menor.

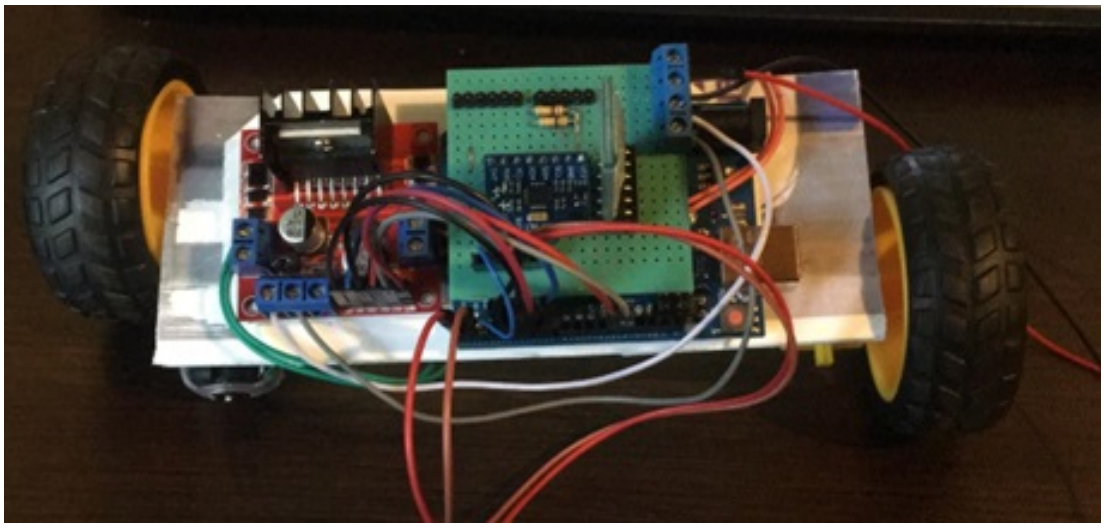

Figura 23. Prototipo robot balancin

Para montar los componentes sobre la estructura se utilizaron unos tacos hechos de policarbonato y se los adherio mediante el pegamente “La gotita”. Esto se aprecia mejor en la Fig. 23 .

Por ultimo para las conexiones electricas y el montaje de los modulos se realizo una especie de SHIELD que se muestra en la figura 24. Las dos resistencia que se ven son las necesarias para usar el protocolo I2C.

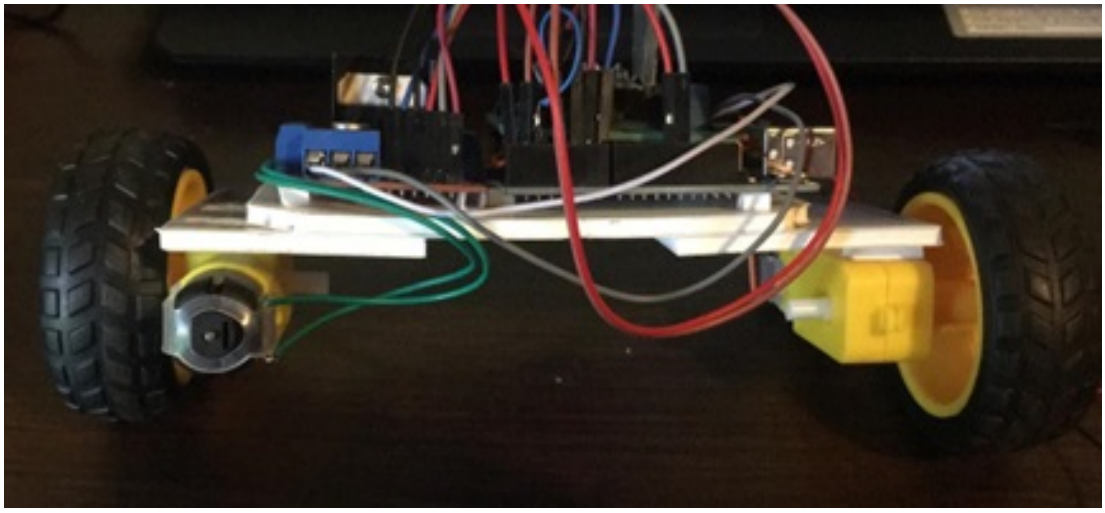

Figura 24. Prototipo robot balancin

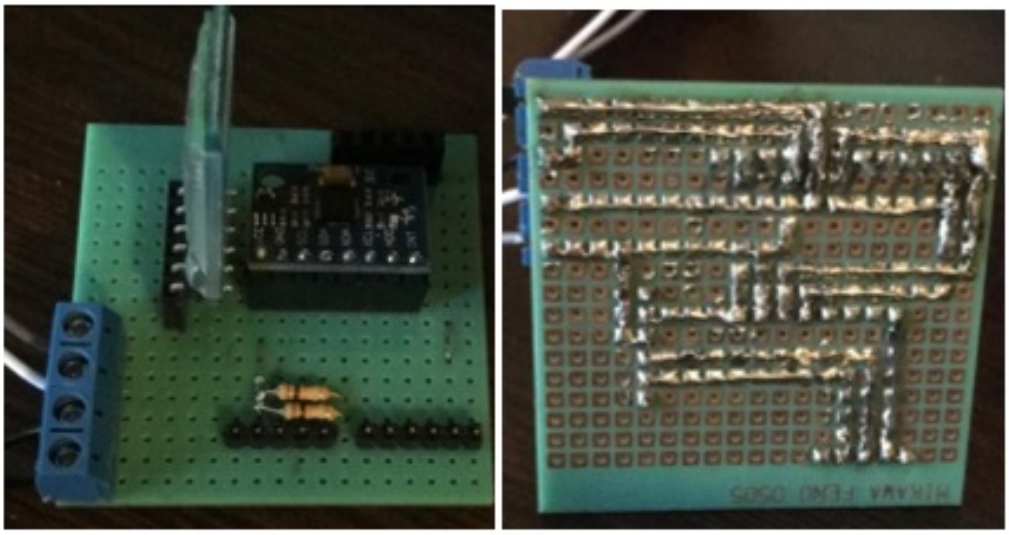

Figura 25. SHIELD desarrollado

6.2. Pruebas

Para llegar a la obtención de la medida correcta de los ángulos se realizaron diferentes pruebas.

Como vemos la curva roja parece tener un retraso importante. Por lo que decidimos jugar con los coeficientes del filtro. Pero obteníamos curvas con mucho ruido cuando aumentábamos el valor del filtro pasa bajos. En la Fig. 27 los coeficientes tienen valores de 0.7 y 0.3, y como vemos la señal es muy mala. Entonces se pensó que era un problema del filtro complementario y tal vez era necesario buscar un filtro mas sofisticado como el filtro de Kalman o intentar reproducir la librería de Jeff Rowberg. También se intento configurar el filtro pasa bajo de la IMU con un menor BW pero se noto que se perdía mucha información y los resultados no eran los correctos.

Lo que no se tuvo en cuenta fue el signo de la velocidad angular dada por el giros-

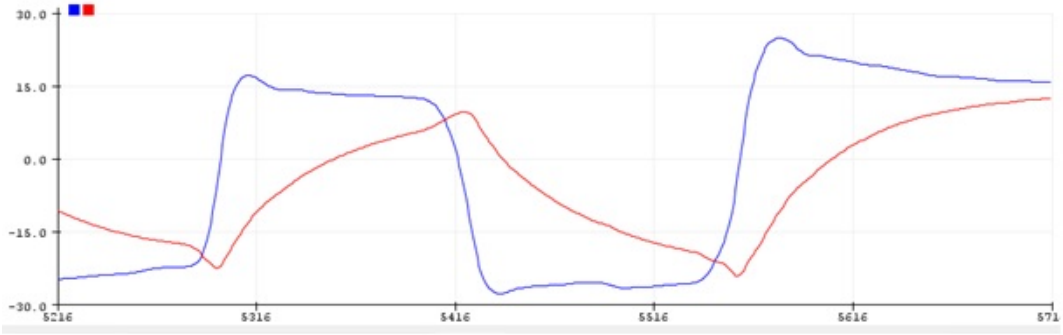

Figura 26. Curva azul (Ángulo obtenido mediante libreria de JR) vs Curva roja (Ángulo obtenido mediante filtro complementario)

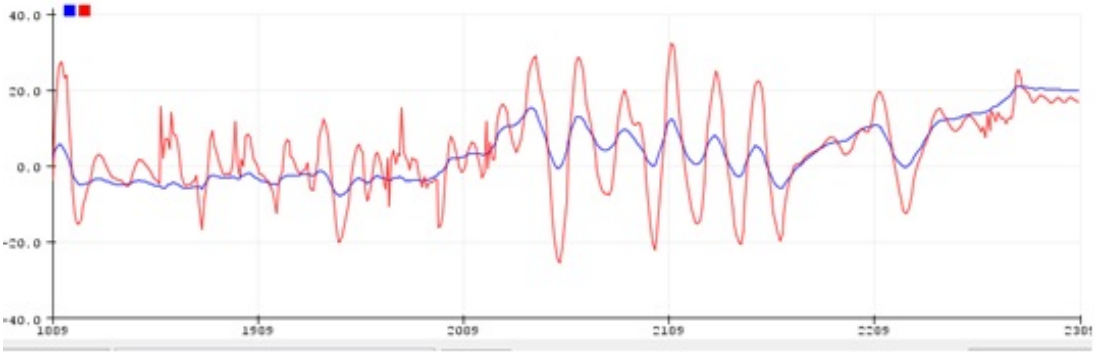

Figura 27. Curva azul (Ángulo obtenido mediante libreria de JR) vs Curva roja (Ángulo obtenido mediante filtro complementario)

copio, claro nada parecía indicar un problema de signos, pero así lo era. En la Fig. 28 se muestra el resultado final.

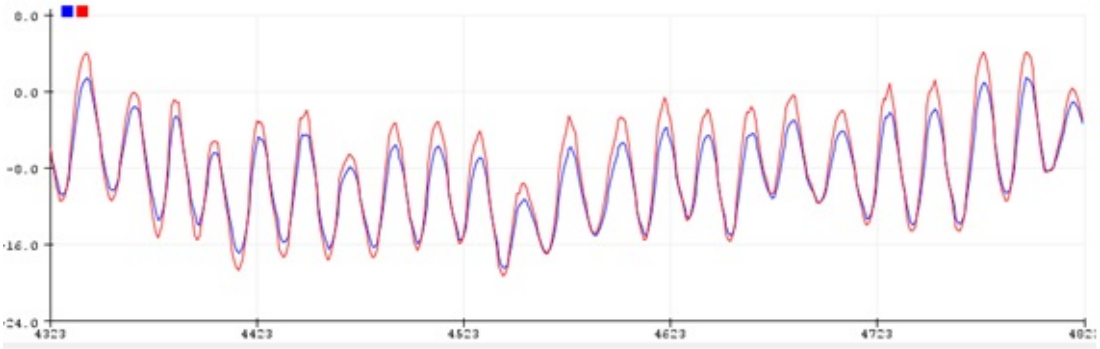

Figura 28. Curva azul (Ángulo obtenido mediante libreria de JR) vs Curva roja (Ángulo obtenido mediante filtro complementario)

La utilización del IDE de Arduino fue únicamente con el fin de realizar los ensayos necesarios para obtener la señal correcta, ya que presentaba gran facilidad para utilizar la librería de Jeff Rowberg que fue utilizada como referencia.

7. Aplicación industrial

Por último, fue propuesto por la catedra que se pensara como se implementaría el prototipo realizado a nivel industrial. Para esto, en primer lugar lo que se mejoraría serian el diseño mecánico, para lo cual se podría utilizar un programa de diseño como SolidWorks u otro semejante. Se buscaría así un diseño mecánico que presente un centro de gravedad cercano al piso, momento de inercia y peso bajo para mejorar su estabilidad. También haciendo un análisis costo beneficio se buscaría un material adecuado que sea resistente y liviano para su construcción.

También se diseñaría un PCB con algún programa específico, como Proteus, así se tendría una placa compacta con todos los módulos que utiliza nuestro robot.

Se realizaría el modelo dinámico del mismo, de esta manera se podría aplicar el método de variables de estado y realizar un mejor algoritmo de control.

Además dependiendo el tamaño del robot y los torques requeridos se elegiría un motor DC adecuado con su correspondiente caja reductora.

Para el control de la posición angular se incluirían encoders para cada motor teniendo así más información y pudiendo mejorar la performance de nuestro robot.

Por último se cambiaría el motor de computo de nuestro robot utilizando una placa con mayores prestaciones que nos permita realizar algoritmos más complejo como podría ser la implementación de un filtro de Kalman para un mejor acondicionamiento de las señales leídas de los sensores. Una buena alternativa seria una Beaglebone black ya que por sus características podría implementarse estos algoritmos sin problemas y gracias a sus PRUs se podría atender por separado la adquisición de datos y la implementación de los algoritmos de control.

8. Resultados, especificaciones finales

Los resultados obtenidos fueron satisfactorios ya que se logro que el robot mantenga su equilibrio, aunque se desearía que este fuese mas estable y resistente ante estímulos externos.

El siguiente es un video que muestra el funcionamiento del balancín: https://youtu.be/VBSCI6RdObE

En cuanto a las especificaciones, el robot es alimentado con 7 V. El puente H se alimenta con 6.5 y el máximo voltaje que actúa sobre los motores es de 3.5 V, como vemos muy por debajo del máximo soportado por los motores de 6 V, pero esto se decidió asi, porque si se los alimenta con voltajes mayores el movimiento de los motores es muy rápido provocando un aumento de la inestabilidad de este.

Por ultimo, se simulo el circuito mostrado en la Fig. 29 con el fin de evaluar el tiempo que le lleva al micro realizar la rutina principal. Se obtuvo que esta le tomaba al procesador un tiempo de 0.386 ms, teniendo en cuenta que en el circuito simulado no se lleva acabo la lectura del sensor por I2C sino que esta lectura se reemplaza por la lectura de un ADC y siendo la frecuencia de muestreo igual a 5ms es posible que esta frecuencia sea muy alta para las caracteristicas de nuestro micro.

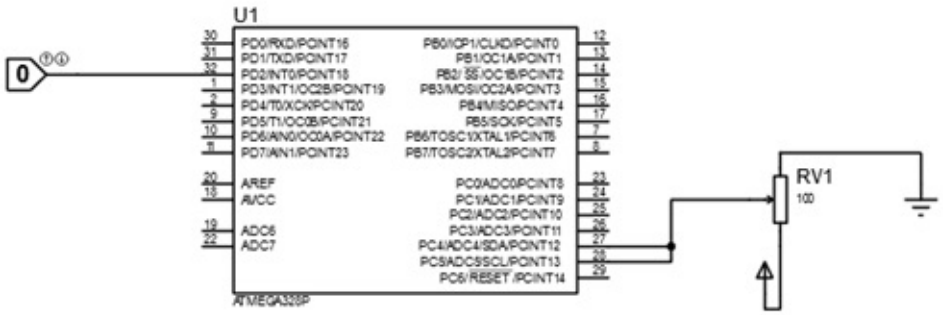

Figura 29. Circuito para simulación

9. Conclusiones

Creo que fue un proyecto muy interesante que me gusto mucho realizar y que me costo mas de lo que pensaba. Es un proyecto que abarca varios temas de la materia, en especial la parte de microcontroladores, pero también se toca temas de electrónica de potencia como es el uso del puente H para el control de los motores. Por otro lado, se entra temas muy interesantes e importantes para la mecatrónica como son la teoría de control y la adquisición de datos de sensores.

Me hubiese gustado no haber perdido tanto tiempo con el tema del filtro complementario ya que era un problema simple.

En cuanto a las mejor a futuros:

- Mejorar el algoritmo de la comunicación I2C, reemplazar los polling al flag de interrupciones por interrupciones con sus rutinas de servicios. Hacer el algoritmo más robusto, capaz de atender los diferentes estados que se pueden dar en la comunicación I2C.
- Disminuir los efectos inerciales productos de los cambios bruscos de velocidades en los motores, creando perfiles de velocidades trapezoidales.
- Mejorar el algoritmo de control

10. Referencias

- L298 H Bridge
- MPU 6050 datasheet
- MPU 6050 register map
- Módulo bluetooth Hc 05 datasheet
- Atmega 328p datasheet
- Libreria MPU 6050 Jeff Rowberg
- Configuración módulo bluetooth
- Tutorial acelerómetro y giroscópio
- Apuntes de catedra