Titulo:	Estación Meteorológica

Módulo de Hardware o Software de investigación - “desafío”
Comunicación I2C.

Ciclo Lectivo:	2016	Curso:	R2054

Integrantes

Apellido Nombres	Legajo	Calificación individual	Fecha
Gonzalez Itzik, Federico.	1570687		
Sivilotti, Pablo.	1477377		

Calificación grupal:		Fecha:

Profesor:	
Auxiliar/es Docente/s:	

Canje por 2° parcial	**SI** **NO**

Observaciones primera entrega	
Observaciones segunda entrega	

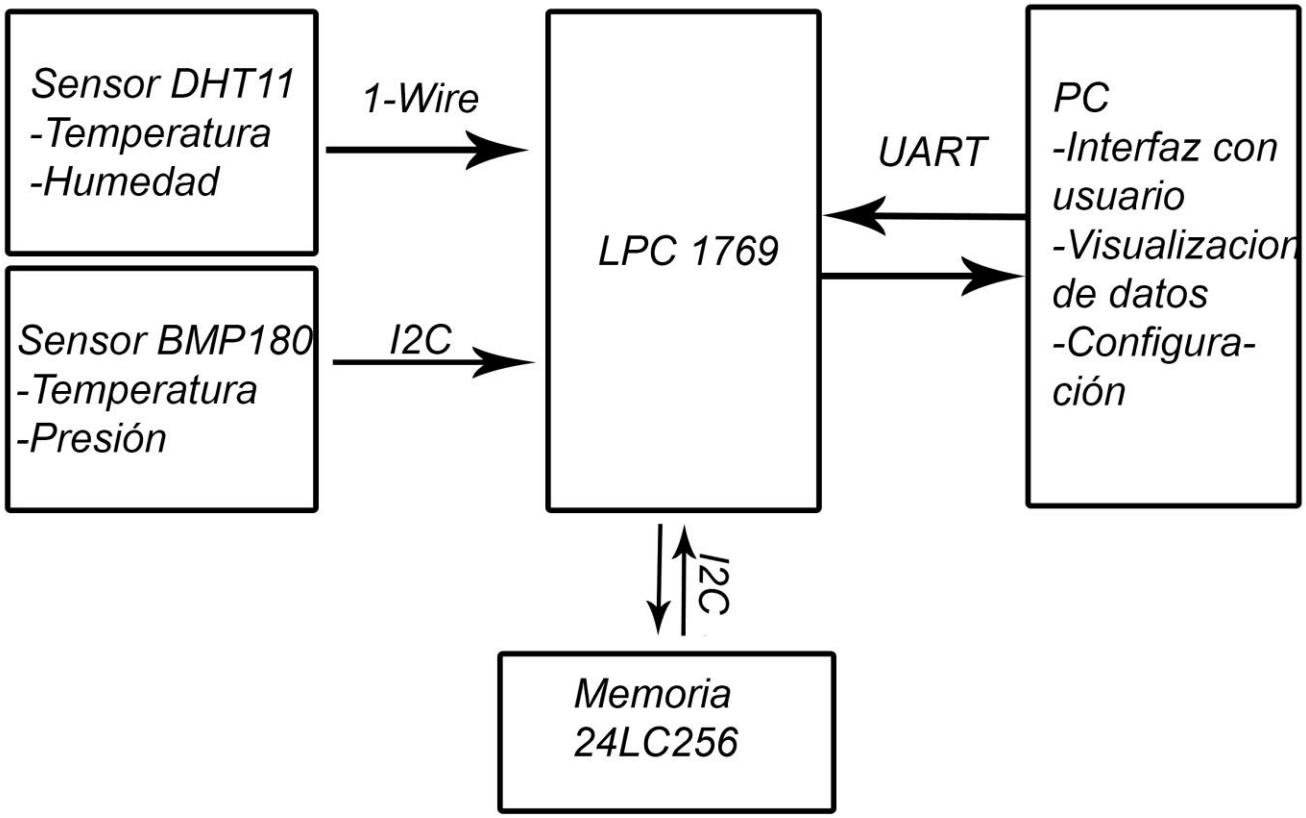
INDICE

Introducción:

La Estación Meteorológica es una instalación de dispositivos que captan los distintos cambios del medio ambiente, está destinada a medir, registrar y enviar con regularidad los datos censados. El usuario podrá seleccionar la frecuencia con la que desea registrar los datos.

Los objetivos de la práctica son los siguientes:
-Desarrollar un sistema de medición de variables físicas.
-Desarrollar un software de interfaz que permita la interacción entre la estación y el usuario.

Diagrama en bloques



Descripción detallada de bloques y HW:

-DHT11:

Este bloque es el encargado de registrar valores de Temperatura y de Humedad, el hardware utilizado es el sensor digital DHT11.

Características generales del sensor:

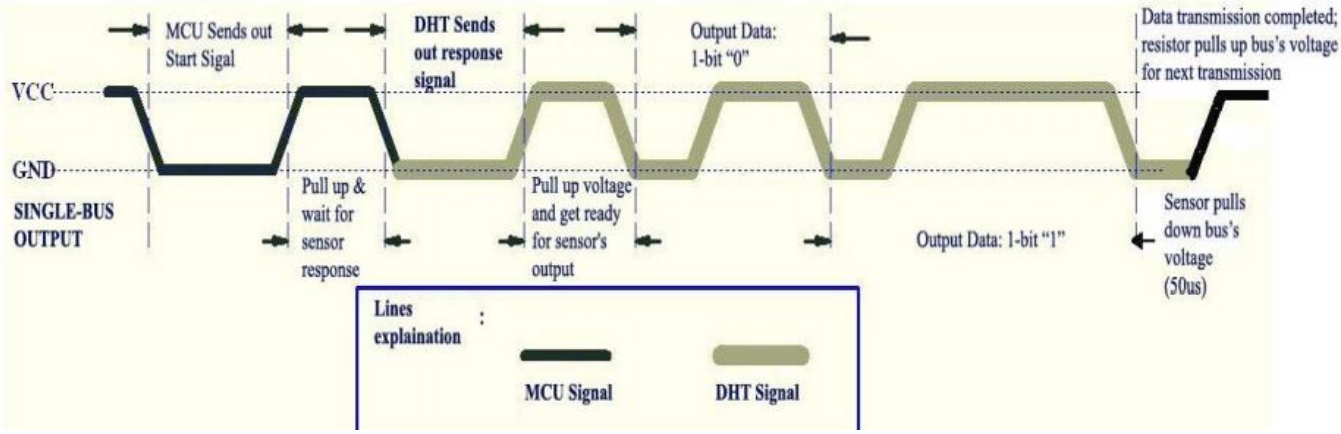
Humedad relativa:
Resolución: 16bits.
Rango: de 20% a 95%.
Exactitud: +-5%RH.

Temperatura:
Resolución: 16bits.
Rango: de 0°C a 50°C.
Exactitud: +-2°C.

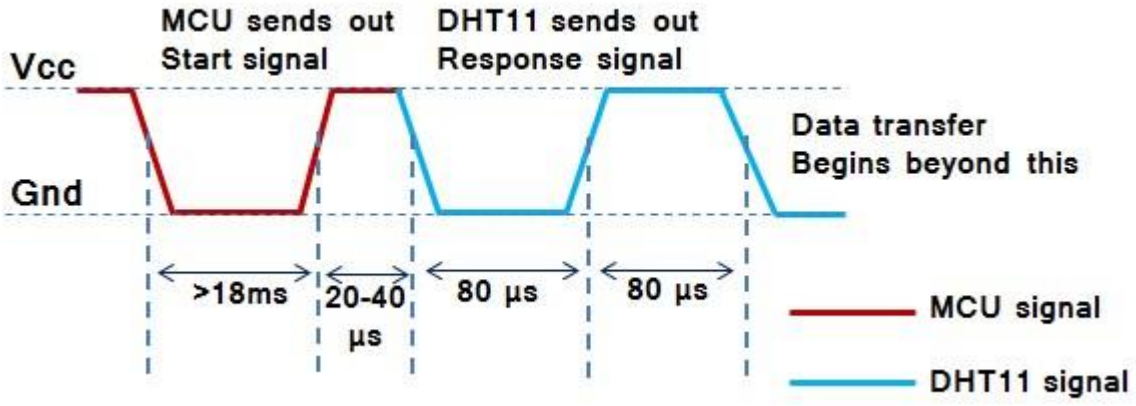
Funcionamiento:

Este dispositivo es el encargado de transformar una variable física, como puede ser la temperatura o la humedad, en una variable eléctrica, capaz de ser interpretada por nuestro microcontrolador. Para ello es necesario que de alguna forma se comuniquen el dispositivo y el microcontrolador. El tipo de comunicación utilizado por el sensor DHT11 responde al protocolo 1-wire que será explicado a continuación.

Protocolo 1-wire:

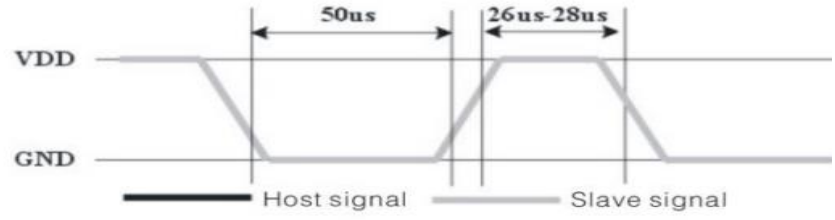


Para que el sensor tome los datos físicos primero debemos pedirle, para ello debemos, de alguna forma, enviarle una señal que pueda ser interpretada por el sensor. A esta señal llamaremos "Señal MCU" o "MCU Signal". Esta señal consiste en mantener la línea en estado bajo 18ms y levantarla, entre 20uS y 40uS el DHT responderá estableciendo un nivel bajo de 80uS y un nivel alto de 80uS, a continuación comienza el envío de datos (cabe recalcar que la línea reposa en estado alto).

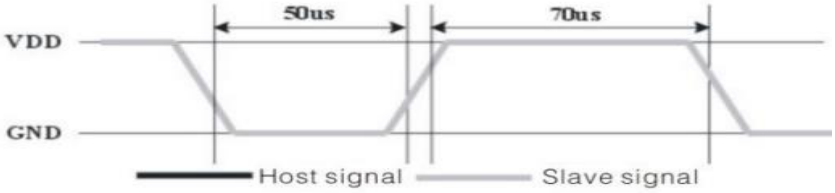


A continuación recibimos 40bytes de datos.
Byte1: Parte entera de humedad relativa.
Byte2: Parte decimal de humedad relativa.
Byte3: Parte entera de temperatura.
Byte4: Parte decimal de temperatura.
Byte5: Checksum.
Puesto que el modelo DHT11 no utiliza parte decimal, los Bytes 2 y 4 pueden ser ignorados.

El envío de datos se realiza de la siguiente forma:
Primero el DHT11 envía un estado bajo de 50uS, luego envía un estado alto de 70uS para comunicar un 1 o un estado alto de entre 26uS y 28uS para comunicar un 0.



Bit data "0" bit format



Bit data "1" bit format

Luego de ser recibidos los 32 bits con los parámetros físicos, se espera recibir el Checksum. La función del Checksum es verificar si los datos son correctos y se utiliza de la siguiente manera:
La suma del Byte1 + Byte2 + Byte3 + Byte4, debe ser igual al Byte de Checksum, de ser así los datos son correctos.

Software:

Software para saber la duración del pulso:

```
void EINT3_IRQHandler(void)
{

    // Interrumple si cambia de estado el pin conectado al DHT y tomo
    if (IOIntStatus & 0x04) // La interrup vino por algún bit de P2?
       {
          //Interrumpe por flanco Ascendente
          if((IO2IntStatR >> 8) & 0x01)// Fue el P2[8] Expansion 16?
          {
              frecuenciaDHT = T0TC;
              IO2IntClr |= 0x00000001<<8; // apago el flag del P2[8]
          }


          //Interrumpe por flanco descendente
          if((IO2IntStatF >> 8) & 0x01)// Fue el P2[8] Expansion 16?
          {
              frecuenciaDHT =  T0TC - frecuenciaDHT; //diferencia entre el contador en cada pulso
              LeerBit (frecuenciaDHT); //Envio la frecuencia en us
              IO2IntClr |= 0x00000001<<8; // apago el flag del P2[8]
          }



    }

}
```

1) El PIN del DHT11 encargado de enviar y recibir pulsos se conecta al pin 2,8 (expansión 16 del kit) y se configura como GPIO.
2) Se configuran las interrupciones por GPIO para que ese pin interrumpa por flanco ascendente y por flanco descendente.
3) Cuando interrumpe por flanco ascendente (se levanta la línea) se toma el valor de T0TC (registro de timer0 en que se encuentra el valor de la cuenta, cuyo valor se incrementa en 1 para un micro segundo).
4) Cuando interrumpe por flanco descendente (se baja la línea) al actual de T0TC se le resta el valor que tenía al momento de interrumpir por flanco ascendente. Así de esta manera se logra obtener, en micro segundos, el tiempo que la línea estuvo en estado alto.
5) Se envía ese valor a una función cuyo objetivo es validar los datos y tiene el siguiente código:

```
if ((frec >= 78) && (frec <= 90)) //Respues
{
    bit = 0;
}

if (bit < 32){
    if ((frec <= 30) && (frec >= 20)){
        (bufferDHT) &= ~(0x01<<(31-i));
        i++;
        bit++;
    }
    if ((frec >= 65) && (frec <= 75)){
        (bufferDHT) |= (0x01<<(31-i));
        i++;
        bit++;
    }
}

if (bit >= 32 && bit < 40){
    if ((frec >= 65) && (frec <= 75)){
        (CheckSum) |= (0x01<<(8-j));
        j++;
        bit++;
    }
    if ((frec <= 30) && (frec >= 20)){
        (CheckSum) &= ~(0x01<<(8-j));
        j++;
        bit++;
    }
```

6) Si la línea estuvo en estado alto entre 78uS y 90uS significa que el DHT va a comenzar a enviar los datos, se le da el valor 0 a la variable bit.
7) Los primeros 32 bits son los datos de los parámetros, si la línea estuvo en estado alto entre 20uS y 30uS se guarda un 0 en el buffer y se incrementa la variable bit, si estuvo entre 65uS y 75uS se guarda un 1 en el buffer y se incrementa la variable bit
8) Después de recibidos los primeros 32 bits se espera recibir los 8 bits del Checksum, se opera de la misma manera pero esta vez los datos se guardan en una variable Checksum. Luego hay que verificar si los datos recibidos son correctos de la siguiente manera:

```
if (bit >= 40){


    if (((((bufferDHT>>8 & 0xFF) +  (bufferDHT>>24 & 0xFF)) & 0xFF) == CheckSum)&& (CheckSum != 0))
        H_byte = (bufferDHT>>24 & 0xFF);
        H_byte2 = (bufferDHT>>16 & 0xFF);
        Temp_byte = (bufferDHT>>8 & 0xFF);
        Temp_byte2 = (bufferDHT & 0xFF);

    }

    i = 0;
    j = 0;
    bufferDHT = 0;
    CheckSum = 0;

}
```

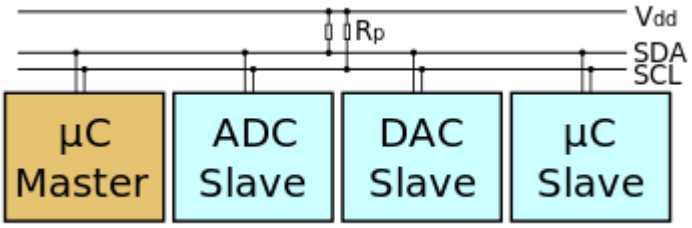
9) Si los primeros 8 bits de la suma entre los valores de Humedad y Temperatura es igual al CheckSum se guardan los valores de Humedad y Temperatura en variables globales.

Bus I2C:

Introducción:
Un circuito interintegrado (I2C, del inglés Inter-Intergrated Circuit) es un bus serie de datos desarrollado en 1982 por Philips Semiconductors (hoy NXP Semiconductors). Se utiliza principalmente internamente para la comunicación entre diferentes partes de un circuito, por ejemplo, entre un controlador y circuitos periféricos integrados. Es una comunicación sincrónica, digital y de dos bandas entre dispositivos.
El microcontrolador LPC1769 cuenta con 3 unidades I2C.

Características:
El I2C precisa de dos líneas de señal: una será la señal de reloj (SDL, Serial Clock) y la otra la línea de datos (SDA, Serial data).
Funciona con un sistema Master-Slave.

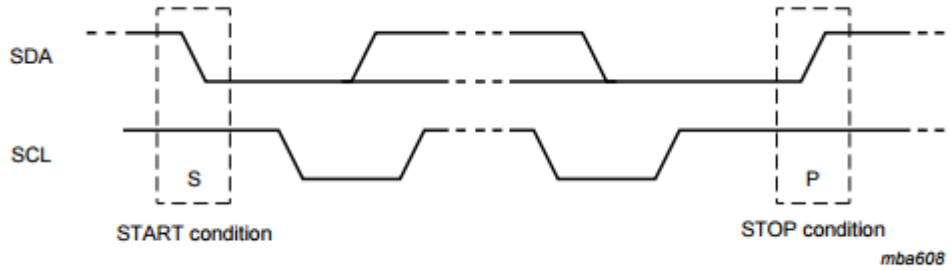
Pueden ser seleccionadas varias velocidades diferentes, algunas son: Standard Mode (0,1 Mbit/s), Fast Mode(0,4 Mbit/s), Fast Mode Plus (1,0Mbit/s).
Los datos y direcciones se transmiten con palabras de 8bits.



Funcionamiento:

Condiciones de START y STOP:

El Master es el dispositivo que inicia la transferencia en el bus y genera la señal de clock. El Master debe informar el comienzo de la comunicación a partir de una condición de START, la línea SDA cae a cero mientras SCL permanece en nivel alto. A partir de este momento comienza la transferencia de datos. Una vez finalizada la comunicación se debe informar de esta situación, estableciendo una condición de STOP, que consta en que la línea SDA pasa a nivel alto mientras SDL permanece en estado alto.

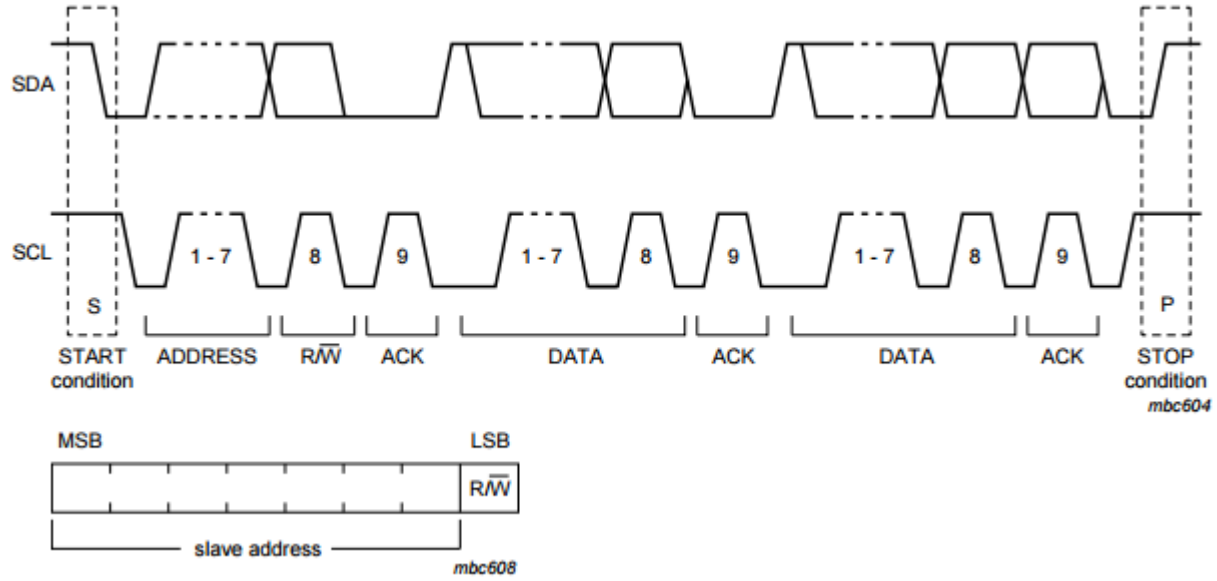


START and STOP conditions

Transferencia de datos:

Una vez que el master genero la condición de START comienza la transmisión de datos. Cada palabra puesta en el bus de SDA debe contener 1 byte. El primer byte enviado debe contener la dirección del esclavo al que quiero acceder (ya que puedo tener conectados varios esclavos en un mismo maestro), este byte consta de 7 bits con la dirección y el ultimo bit indica si quiero escribir o si quiero leer del esclavo (0 indica escritura, 1 indica lectura).

Una vez que se envía la dirección del esclavo al que quiero acceder este responde con un bit de ACK (Acknowledge, o reconocimiento en castellano) o NACK (Not Acknowledge, o no reconocimiento en castellano). Con un 0 indica un ACK y la comunicación continua. Con un 1 se indica un NACK y el Master puede optar por frenar la comunicación estableciendo un bit de STOP o volver a mandar un bit de START.



En el código:

Nuestra forma de acceder a una comunicación I2C es la siguiente:

```
363
364 uint8_t Motor_I2C ( )
365 {
366     //Envío bit de start
367     I2C1CONSET =  CONSET_STA;
368
369     FlagMaster = OCUPADO;
370
371     while ((FlagMaster == OCUPADO) || (FlagMaster == I2C_STARTED))
372     {
373         if (I2C_Timeout >= TIMEOUT){
374             FlagMaster = ESTADO_TIMEOUT;
375             break;
376         }
377         I2C_Timeout++;
378     }
379
380     I2C1CONCLR = I2CONCLR_STAC;
381
382     return (FlagMaster);
383
384 }
```

El funcionamiento es simple:
En la línea 367 lo primero que se hace es enviar el bit de start y entra en un ciclo while hasta que la comunicación finalice y hasta que pase determinado tiempo.

Una vez enviado el bit de start la forma que tenemos de enviar o recibir archivos es desde el handler de interrupciones.
Pueden haber 11 razones diferentes por la cual se entra al handler (para el uso que le damos, para otros usos puede haber más). Una vez enviado el bit de start en el handler pasara lo siguiente:

```
else if ((I2C1STAT & 0xFF) == 0x08){

    /*
    A START condition has been transmitted. Se envía la direccion del esclavo + R/W.*/
    IndexEscritura = 0;
    IndexLectura = 0;
    /*1. Se escribe la direccion del esclavo+R/W en I2C0DAT*/
    I2C1DAT = MasterBuffer[IndexEscritura];
    /*2. Write 0x04 to I2CONSET to set the AA bit.*/
    I2C1CONSET = 0x04;
    /*3. Write 0x08 to I2CONCLR to clear the SI flag.*/
    I2C1CONCLR = 0x28;

    FlagMaster = I2C_STARTED;

    IndexEscritura++;

}
```

Si todo funciona bien, entrará en el siguiente if, y nosotros debemos enviarle la dirección a la cual queremos acceder, la cual esta previamente guardada en MasterBuffer[0], y se la enviamos por medio del registro I2C1DAT.

El próximo estado será el 0x18 (en caso de escritura), en este se transmite el primer byte de datos o se finaliza la comunicación en caso de que no haya bytes de datos.

```
else if ((I2C1STAT & 0xFF) == 0x18){
    /*
    Estado previo: 0x08 o 0x10, Se transmitió la dirección de esclavo
    y se recibió el ACK. Se transmite el primer byte de datos.*/

    if (LargoEscritura == 1)
    {
        I2C1CONSET = I2CONSET_STO;
        FlagMaster = I2C_NO_DATA;
    }

    else{
        /*1. Se carga el I2C0DAT con el primer byte de datos del buffer.*/
        I2C1DAT = MasterBuffer[IndexEscritura];
        /*4. Increment Master Transmit buffer pointer.*/
        IndexEscritura++;
    }
    /*2. Write 0x04 to I2CONSET to set the AA bit.*/
    I2C1CONSET = 0x04;
    /*3. Write 0x08 to I2CONCLR to clear the SI flag.*/
    I2C1CONCLR  = 0x28;
    //printf ("0x18\n");
}
```

El próximo estado será el 0x28 donde se imprimen los datos en I2C1DAT hasta que sea el último, en caso de ser el último dato se enviara un bit de STOP

```
else if ((I2C1STAT & 0xFF) == 0x28){
    /*
    Los datos se transmitieron y se recibio el ACK. Si el dato transmitido es el ultimo
    se envía bit de STOP, sino se envía el siguiente byte de datos.*/

    if (IndexEscritura < LargoEscritura)
    {
        /*5. Load I2DAT with next data byte from Master Transmit buffer.*/
        I2C1DAT = MasterBuffer [IndexEscritura];
        /*8. Increment Master Transmit buffer pointer*/
        IndexEscritura++;
        I2C1CONCLR = 0x08;
    }
```

Si nunca se recibe el ACK también se envía un bit de STOP y se finaliza la comunicación

En caso de lectura el estado siguiente al 0x08 es el 0x40, en este estado se recibió un dato, debemos retornar un ACK o un NACK en caso de que sea el último dato esperado.

```
  else if ((I2C1STAT & 0xFF) == 0x40){
      /*
      Estado anterior: 0x08 o 0x10. Se transmitio dirreccion de esclavo,
      Se recibio ACK. Data will be received and ACK returned.*/

      if ((IndexEscritura)< LargoLectura)
      {
          /*1. Write 0x04 to I2CONSET to set the AA bit.*/
          I2C1CONSET = 0x04;
      }
      else
      {
          I2C1CONCLR = I2CONCLR_AAC;
      }
      /*2. Write 0x08 to I2CONCLR to clear the SI flag.*/
      I2C1CONCLR = 0x08;
      //printf ("0x40\n");

  }
```

En caso de haber retornado un ACK el siguiente estado es el 0x50 donde guardamos los datos en un buffer y se retorna un ACK en caso de ser el último dato esperado, se retorna NACK

```
else if ((I2C1STAT & 0xFF) == 0x50)
{
    /*
    Se recibio dato, Se envió ACK. Se leen los datos de I2DAT. Additional
    data will be received. If this is the last data byte then NOT ACK will be returned,
    otherwise ACK will be returned.*/

    /*1. Read data byte from I2DAT into Master Receive buffer.*/
    BufferEsclavo[IndexLectura++] = I2C1DAT;

    /*2. Decrement the Master data counter, skip to step 5 if not the last data byte.*/
    if (IndexLectura+1 < LargoLectura){
        /* 3. Write 0x0C to I2CONCLR to clear the SI flag and the AA bit.*/
    //  I2C1CONCLR = 0x0C;
        I2C1CONSET = 0x04;
    }else
    {
    /*5. Write 0x04 to I2CONSET to set the AA bit.*/
    //I2C1CONSET = 0x04;
    /*6. Write 0x08 to I2CONCLR to clear the SI flag.*/
    I2C1CONCLR = 0x04;
    }
```

En caso de no haberse retornado un ACK en el estado 0x40 el siguiente estado es el 0x58 en donde guardamos los datos en el buffer y damos por finalizada la comunicación.

-BMP180:

Este bloque es el encargado de medir Temperatura y Presión, el hardware utilizado será el sensor digital BMP180, para nuestro proyecto el encargado de realizar mediciones de temperatura será este sensor ya que es más eficaz que el DHT11.

Características generales del sensor:

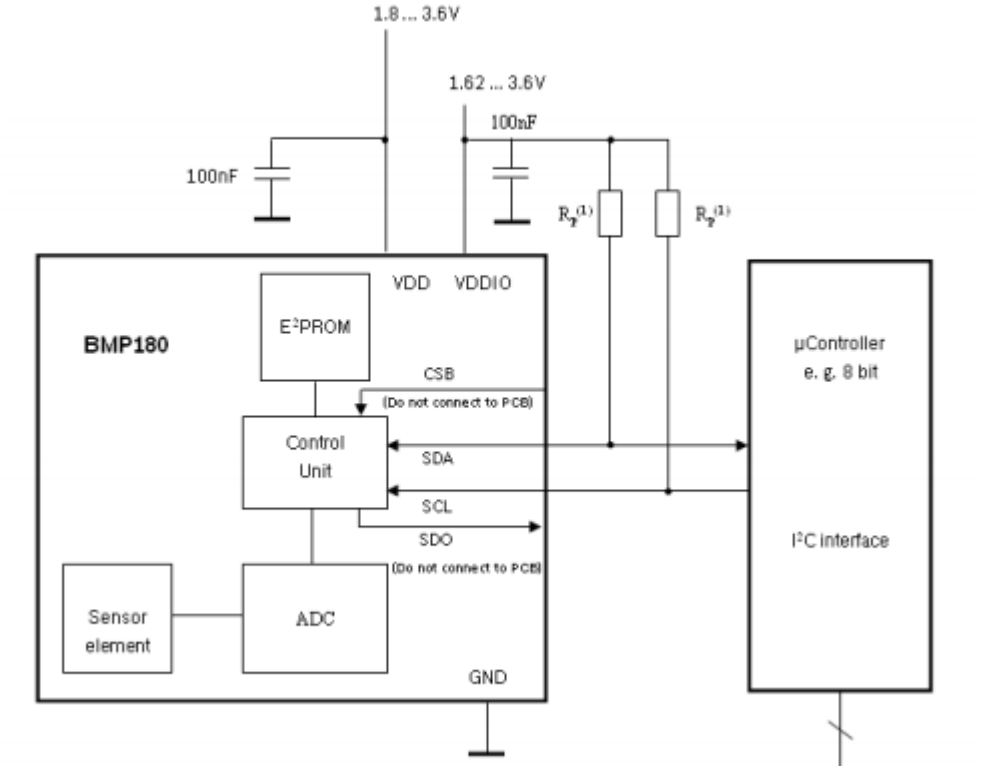
Presión relativa:
Resolución: 16bits.
Rango: de 950hPa a 1050hPa.
Exactitud: +-0.12hPa.

Temperatura:
Resolución: 16bits.
Rango: de -40°C a 85°C.
Exactitud: +-2°C.

Funcionamiento:

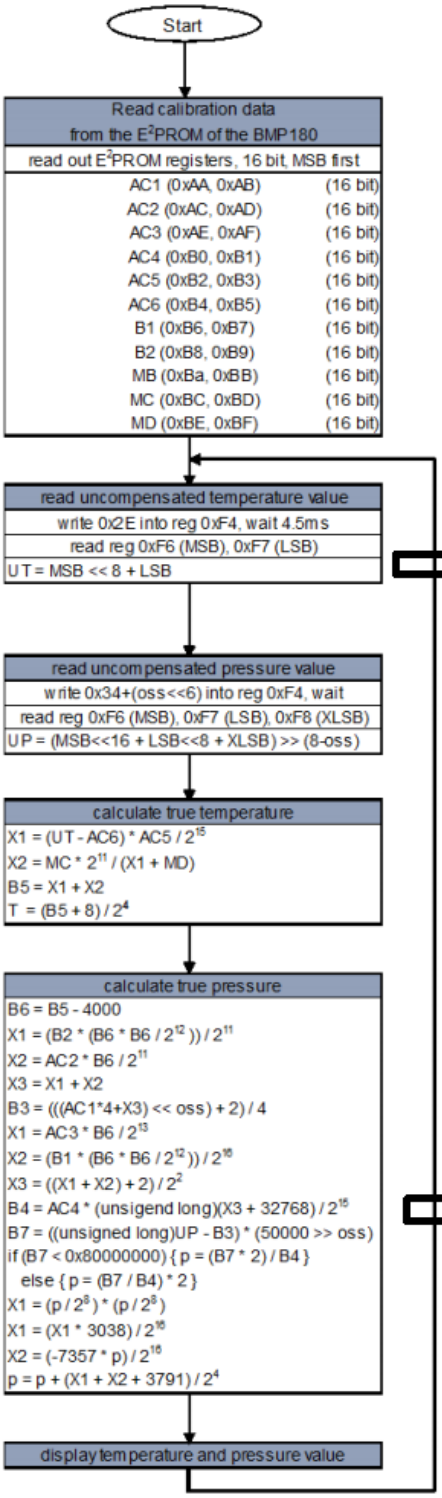
El funcionamiento es similar al ya mencionado DHT11, la diferencia radica en el tipo de comunicación con el microcontrolador. El protocolo de comunicación es el I2C y ese será nuestro desafío con este proyecto.

El desarrollo del bloque es el siguiente:



El funcionamiento es el siguiente: Primero el BMP180 toma los datos físicos, luego los digitaliza en el módulo ADC. El microcontrolador utiliza el I2C para leer los datos de calibración en la memoria E2PROM, una vez que se tengan los datos de calibración se toman los parámetros de temperatura y presión y se realizan una serie de cálculos para determinar la temperatura y la presión.

El funcionamiento del software es el siguiente:




```
void BMP180_init (void)
{

    //Read Calibration Data from E2PROM of the BMP
    AC1 = Leer_BMP180_I2C_short (0xAA, 0xAB);
    AC2 = Leer_BMP180_I2C_short (0xAC, 0xAD);
    AC3 = Leer_BMP180_I2C_short (0xAE, 0xAF);
    AC4 = Leer_BMP180_I2C_Ushort (0xB0, 0xB1);
    AC5 = Leer_BMP180_I2C_Ushort (0xB2, 0xB3);
    AC6 = Leer_BMP180_I2C_Ushort (0xB4, 0xB5);
    B1 = Leer_BMP180_I2C_short (0xB6, 0xB7);
    B2 = Leer_BMP180_I2C_short (0xB8, 0xB9);
    MB = Leer_BMP180_I2C_short (0xBA, 0xBB);
    MC = Leer_BMP180_I2C_short (0xBC, 0xBD);
    MD = Leer_BMP180_I2C_short (0xBE, 0xBF);


}
```

```
uint8_t Start_Temperatura (void)
uint16_t  BMP180_Temperatura (void){

        //Read uncompensated temperatu
        UT = Get_Temperatura ( );
        //Calculate true temperatura
        X1_1 = (UT - AC6);
        X1_2 = X1_1*AC5;
        X1 = X1_2/32768;
        X2 = MC * 2048 / (X1+MD);
        B5 = X1 + X2;
        T = (B5 + 8)/16;

        return T;
}
```

```
uint8_t Start_pressure (void)

UP = Get_pressure ( );
//Calculate true pressure
B6 = B5 - 4000;
X1 = (B2 * (B6 * B6/4096))/2048;
X2 = AC2 * B6 / 2048;
X3 = X1 + X3;
B3 = (((AC1*4+X3)<<oss)+2)/4;
X1 = AC3 * B6 / 8192;
X2 = (B1 * (B6 * B6/4096))/65536;
X3 = ((X1 + X2)+2)/4;
B4 = AC4 * (unsigned long)(X3 + 32768 )/32768;
B7 = ((unsigned long)UP -B3)*(50000>>oss);
if (B7<0x80000000){p = (B7*2)/B4;}
else{
    p = (B7/B4) * 2;
}
X1 = (p/256) * (p/256);
X1 = (X1*3038)/65536;
X2 = (-7357 * p)/65536;
p = p+ (X1 + X2 + 3791)/16;

return p;
```

Memoria 24LC256:

Introducción:

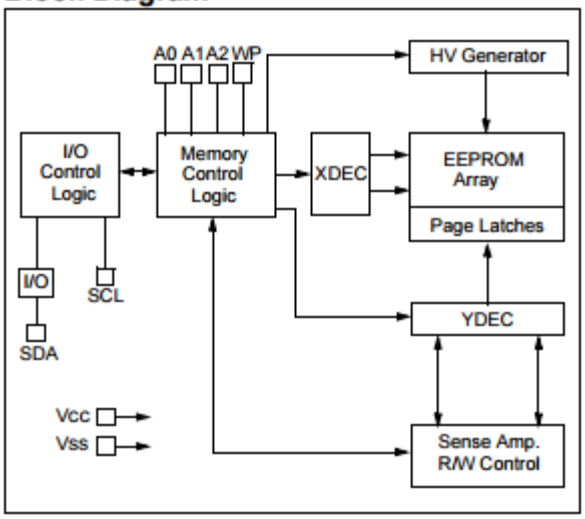
Memorias E2PROM

EEPROM son las siglas de *Erasable Programmable Read-Only Memory* (ROM programable borrable). Es un tipo de chip de memoria ROM no volátil inventado por el ingeniero Dov Frohman de Intel[1] . Está formada por celdas de FAMOS (Floating Gate Avalanche-Injection Metal-Oxide Semiconductor) o "transistores de puerta flotante", cada uno de los cuales viene de fábrica sin carga, por lo que son leídos como 1 (por eso, una EPROM sin grabar se lee como FF en todas sus celdas).

Características:

La memoria 24LC256 es una memoria EEPROM de 256Kbits con páginas de 64Bytes que se comunica mediante I2C con nuestro controlador. El objetivo es poder guardar los parámetros obtenidos por medio de los sensores independientemente de que nuestro dispositivo esté conectado a un PC.

Block Diagram



Funcionamiento:

La forma de escribir y leer en esta memoria es similar a la forma utilizada en el BMP180 ya que este sensor también posee una memoria EEPROM y también se comunica mediante I2C.

Para poder escribir en nuestra memoria debemos seguir el siguiente protocolo:

FIGURE 6-1: BYTE WRITE

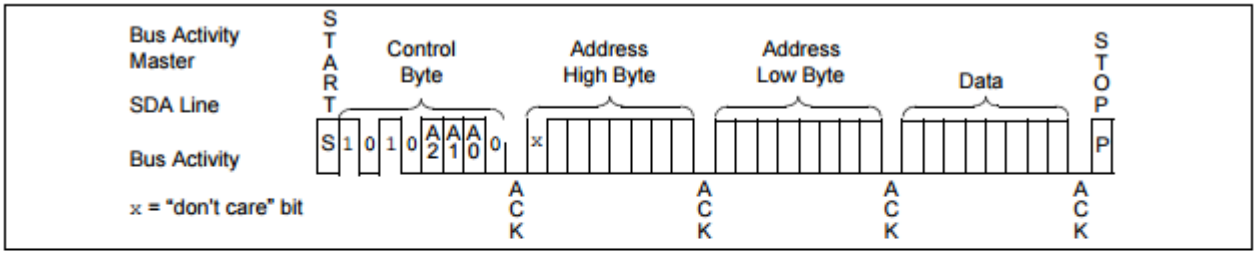
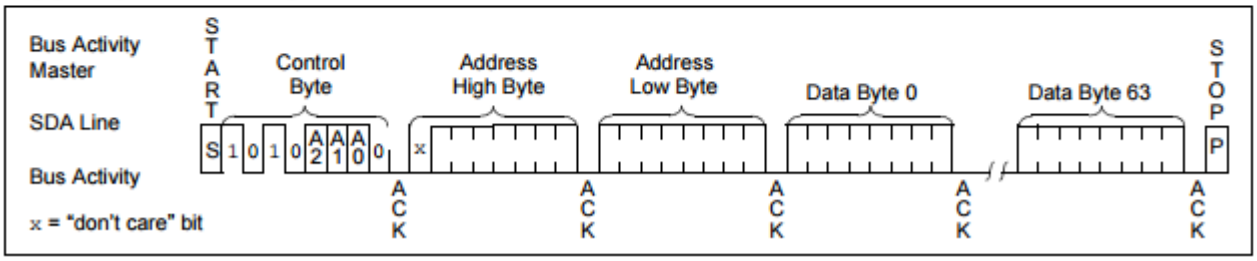


FIGURE 6-2: PAGE WRITE



Para escribir 1 byte primero debemos enviar por medio del I2C la dirección de la memoria que siempre es $1010A_2A_1A_00$ (modo escritura), donde el valor de A se elige conectando la pata de la memoria a estado alto o estado bajo, en nuestro caso las tres patas están conectadas a tierra, por ende la dirección que debemos enviar es 10100000 (0xA0). Una vez enviada esa dirección si recibimos el bit de confirmación significa que ya hemos establecido una comunicación, el siguiente paso es comunicarle a la memoria la dirección donde queremos escribirle (desde 0x0000 hasta 0x7FFF) y por ultimo le enviamos los bytes que deseamos escribir.

```
void EscribirMemoria(uint8_t Address_Comienzo_High,uint8_t Address_Comienzo_Low, char *cadena)
{
    uint16_t i;
    MasterBuffer[0] = LC_Address;
    MasterBuffer[1] = Address_Comienzo_High;
    MasterBuffer[2] = Address_Comienzo_Low;

    for (i=0; cadena[i]!='\0'; i++)
    {
        MasterBuffer[i+3] = cadena[i];
    }

    LargoEscritura = i+3;
    Motor_I2C ( );
}
```

Para poder leer de nuestra memoria debemos seguir el siguiente protocolo:

FIGURE 8-2: RANDOM READ

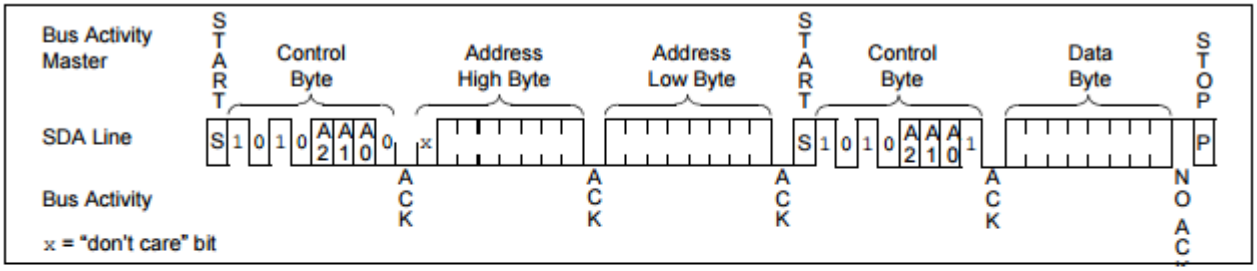
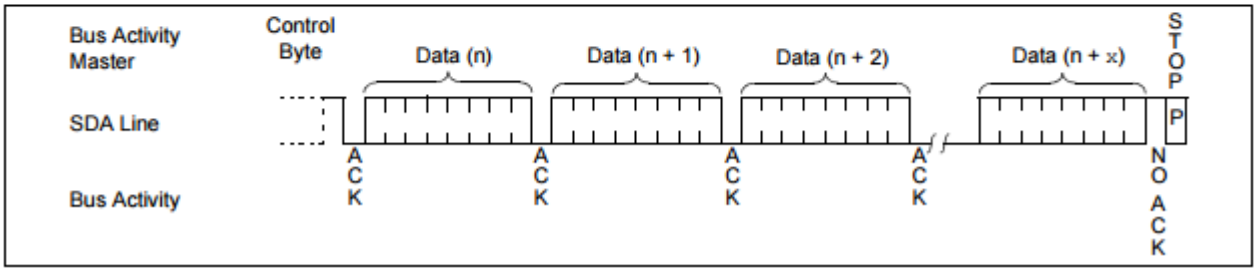


FIGURE 8-3: SEQUENTIAL READ



Primero debemos enviarle la dirección del dispositivo y la dirección de memoria donde queremos acceder de la misma manera que hicimos para escribir. Una vez enviados esos 3 bytes debemos volver a enviar un bit de START y volver a enviar la dirección del dispositivo solo que esta vez en modo lectura (0xA1), luego recibimos los datos de la memoria.

```
uint8_t LeerMemoria (uint8_t Address_High, uint8_t Address_Low)
{
    int i;
    MasterBuffer[0] = (LC_Address);
    MasterBuffer[1] = Address_High;
    MasterBuffer[2] = Address_Low;
    LargoEscritura = 3;
    LargoLectura = 64;
    START_REPEAT=1;

    Motor_I2C ();

    for (i=0; i<64; i++)
    {
        BufferMemoria[i] = BufferEsclavo[i];

    }
```

Nuestra aplicación:
Nuestra aplicación no usa directamente las funciones LeerMemoria y EscribirMemoria, sino que intentamos hacerlo de manera que fuera como leer un archivo, por eso desarrollamos las siguientes funciones:

```
uint16_t BuscarFinal (void)[
```

Esta función lee las direcciones 0x0000 y 0x0001 donde se encuentra la posición del primer byte libre y la devuelve.

```
uint16_t Insertar_Memoria (char *cadena, uint16_t posicion)
```

Esta función inserta una cadena en la memoria y en la posición que nosotros quisiéramos (le enviamos la posición final), y devuelve la nueva posición final.

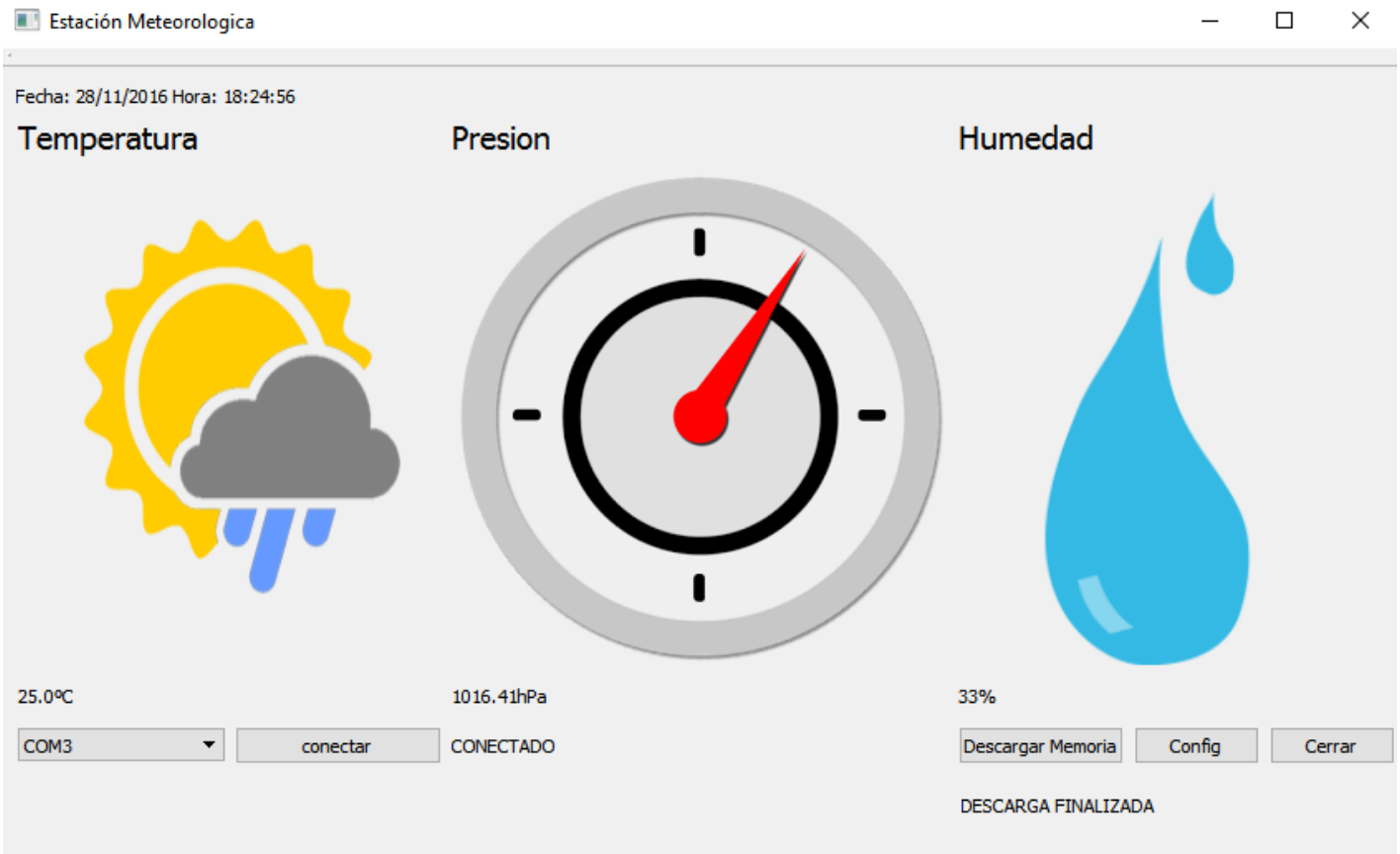
```
uint8_t LeerPagina (uint16_t Pagina)
```

Esta función lee una página, en vez de enviarle la dirección de memoria que deseamos leer, le enviamos la página que deseamos leer.

-Bloques LPC1769:

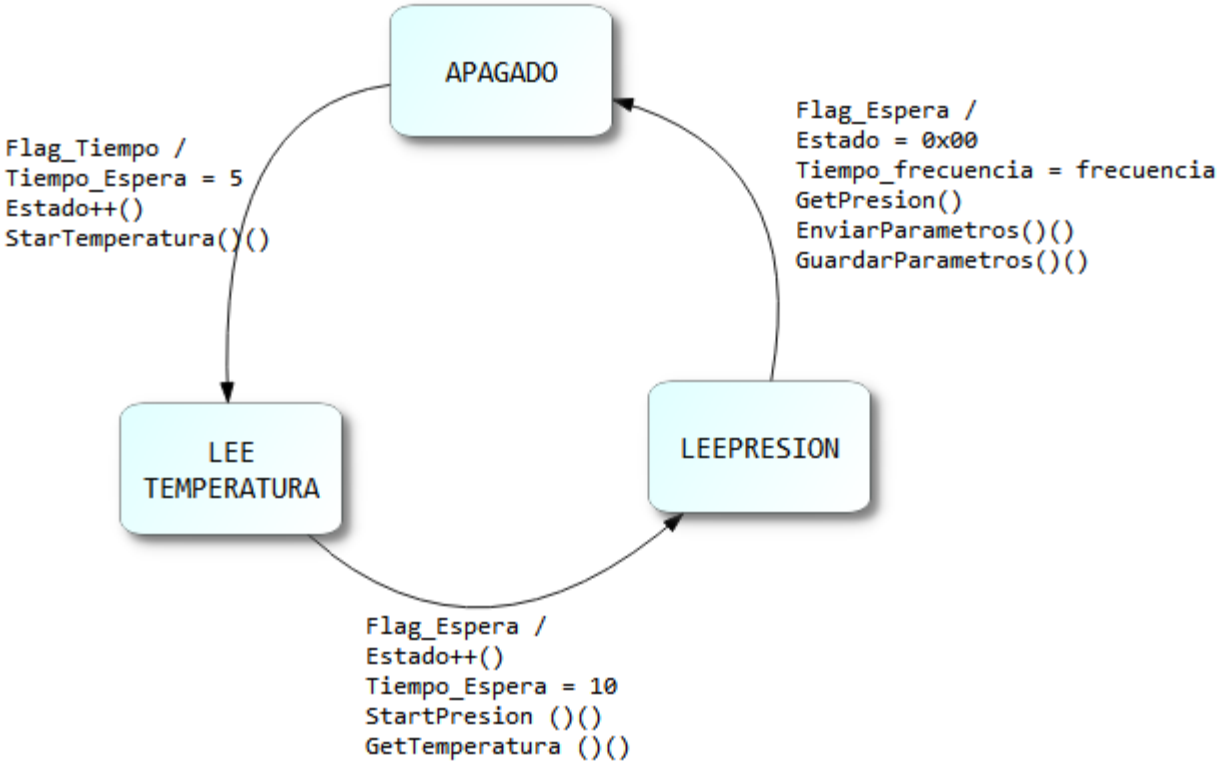
LPC es una familia de microcontroladores integrados de 32 bits. Los chips de LPC se agrupan en distintas series en relación con el núcleo del procesador ARM de 32 bits, como el Cortex-M4F, CortexM3, Cortex-M0+, o Cortex-M0. En nuestro caso, el LPC 1769 están basados en el ARM Cortex-M3 core.
Este es el bloque fundamental, es el encargado de procesar toda la información y distribuirla entre los distintos dispositivos.

-Bloque de interfaz:



Este es el bloque que permite visualizar la información e interactuar con ella. El usuario podrá visualizar las diferentes mediciones en un software de computadora y podrá decidir la frecuencia en la que quiere que se tomen los parámetros físicos. También podrá recibir la información almacenada en la memoria 24LC256.

Máquina de estado:



Funcionamiento:

El Flag_Tiempo amanece activado, por ende al inicio del programa pasa de estado apagado al estado Lee temperatura, en ese momento Inicia una comunicación con el BMP180 e inicia la lectura de la temperatura. Después de esperados 5ms el software pasa de estado leer temperatura al estado leer presión, en este momento Toma los datos de Temperatura e inicia otra comunicación con el BMP180 pero en este caso para leer presión y espera 10ms, luego de esperados 10 ms vuelve al estado apagado, en este momento lee la humedad, lee la presión envía los parámetros al software en la PC y los guarda en la memoria, luego pasa a estado apagado hasta que se vuelva a levantar el Flag_tiempo, este flag se levanta cuando Tiempo_frecuencia llega a 0 y el valor de esta variable se puede configurar desde la PC (por defecto 5 segundos).

Problemas encontrados a lo largo del desarrollo:

- En cuanto al código, hubo bastantes pequeños errores fáciles de solucionar posteriormente pero que nos fueron atrasando.
- Estuvimos mucho tiempo pensando que no funcionaba el código de la UART0 hasta que nos dimos cuenta que el problema estaba con un corto en el kit.

Beneficios encontrados a lo largo del desarrollo:

- A lo largo de la realización del TPO, aprendimos a solucionar problemas, a interpretar hojas de datos, a entender verdaderamente el objetivo de la materia, a utilizar las herramientas que nos ofrecen los IDE utilizados, como el debugger.
- Gracias al DHT aprendimos básicamente a usar un osciloscopio.

Conclusiones:

Se logró el objetivo inicial consiguiendo un dispositivo capaz de sensar variables físicas y ser visualizadas desde una PC.
Con el desarrollo del proyecto pudimos aplicar y afianzar el conocimiento adquirido en la cursada

Links:

https://akizukidenshi.com/download/ds/aosong/DHT11.pdf
https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf
http://www.nxp.com/documents/user_manual/UM10204.pdf
https://es.wikipedia.org/wiki/I%C2%B2C
https://en.wikipedia.org/wiki/NXP_LPC
https://es.wikipedia.org/wiki/Memoria_EPROM