SISTEMAS ELECTRÓNICOS DIGITALES

Trabajo

Micros: Monitorización de la temperatura y humedad relativa ambiental, y simulación de su control empleando un servomotor

---

INTEGRANTES DEL GRUPO	
Apellidos, Nombre	**Matrícula**
Kamenov Iliev, Simeon	55309
López Montero, Jorge	55322
Navarro Bados, Cristina	55380

Grupo Teoría	A404 y EE403
Grupo Trabajo	17
Profesor	Rubén Núñez

FECHA DE ENTREGA: 28/06/2023

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE IMÁGENES

1. Introducción

En este trabajo, se presenta el desarrollo de un sistema basado en el microcontrolador STM32F411 para la programación de un sensor externo de temperatura. El objetivo principal es diseñar un sistema que sea capaz de medir la temperatura ambiente y controlar una serie de dispositivos en función de los valores obtenidos.

En primer lugar, se utiliza el sensor DTH22 para adquirir datos de temperatura con precisión, para la visualización de la temperatura se ha incluido un LCD con un adaptador I2C. Además, se ha incorporado una barra de LED que muestra visualmente la temperatura ambiente de manera intuitiva. Cada LED se enciende progresivamente a medida que la temperatura aumenta en intervalos de 5 grados. Esta funcionalidad ofrece una representación visual y sencilla de la temperatura actual.

Además de la propia medición de la temperatura, se pretende poder simular su control. El "control" estará representado por la actuación de un servomotor, que varía la velocidad de giro en función de la señal de error, siendo esta la diferencia entre la temperatura de establecimiento y la lectura del sensor.

La temperatura de establecimiento se establece inicialmente a $25[^{\circ}C]$, pero se ha habilitado la interrupción del "user button" del microcontrolador para poder acceder a un estado de cambio de consigna. La temperatura de referencia deseada se ajustará mediante el accionamiento de un potenciómetro.

En resumen, este trabajo presenta la implementación de un sistema de medición de temperatura basado en el microcontrolador STM32F411. La combinación del sensor DHT22, la barra de LED, el LCD y el potenciómetro ha permitido desarrollar un sistema versátil y fácil de usar.

2. Componentes electrónicos[1]

Sensor DHT-22

La apariencia de este sensor se puede apreciar en la Imagen 1. Este sensor tiene 4 patillas cuyas funciones, de izquierda a derecha, son: pin de alimentación $(3.3 – 5.5)[V]$; pin de datos; y los 2 últimos son ambos pines de tierra.

Debido a lo último, este sensor también se puede vender con un acoplamiento que cortocircuita las dos puestas a tierras, por lo que el sensor quedaría con 3 pines. Además de cortocircuitar estos pines, el acoplamiento incluye una

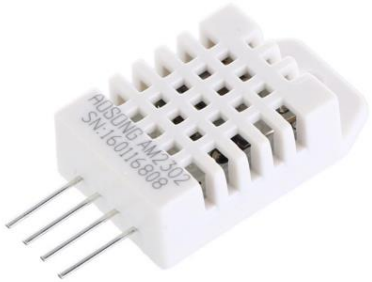


Imagen 1: Sensor DHT-22

[1] Se dejan los datasheet de los productos en el apartado "Anexo II – Enlaces de interés".

resistencia de pull-up entre el pin de datos y la alimentación. En nuestro caso hemos tenido que incluir una externa de valor 10$[k\Omega]$, aunque valía cualquiera entre (1-10)$[k\Omega]$. El datasheet recomendaba un valor de 1$[k\Omega]$, tal y como se puede ver en la Figura 1.

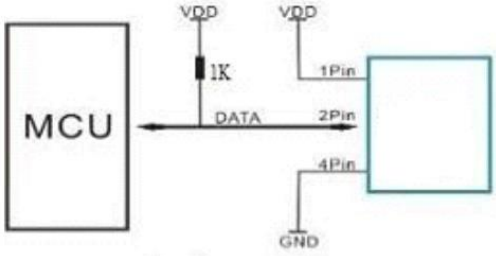


Figura *1*: Conexiones del sensor DHT-22

El sensor se utiliza un protocolo de comunicación serie, que es descrito en el datasheet y que se analizará cuando describamos el código de la librería "DHT.h".

La información se envía en packs de 40 bits, siendo los 16 primeros los referentes a la humedad relativa, los 16 siguientes los de la temperatura y los últimos 8 se utilizan a modo de contrastación para confirmar el correcto envío de la información.

Por último, mencionar que el periodo de muestreo interno del sensor es de 2 segundos, aunque nosotros hemos decidido actualizar el valor de las lecturas cada 5 segundos con la intención de ampliar el contenido del trabajo e incluir una interrupción usando un temporizador interno del microcontrolador.

Pantalla LCD con módulo i2c

El display LCD (Liquid Crystal Display) permite leer información a través de su pantalla, el formato del display utilizado en el proyecto es de 16X2, es decir, muestra 2 filas de 16 caracteres cada una.

Como se ve en la Imagen *2*, el modelo que estamos usando tiene incluido un módulo de comunicación serie i2c para facilitar el control del mismo. El display funcionará en el modo "4 bits" debido a que solamente están unidos 4 pines de datos del LCD al chip integrado del módulo i2c. Esta conexión se aprecia en la Figura 2.

Imagen 2: Pantalla LCD con modulo i2c

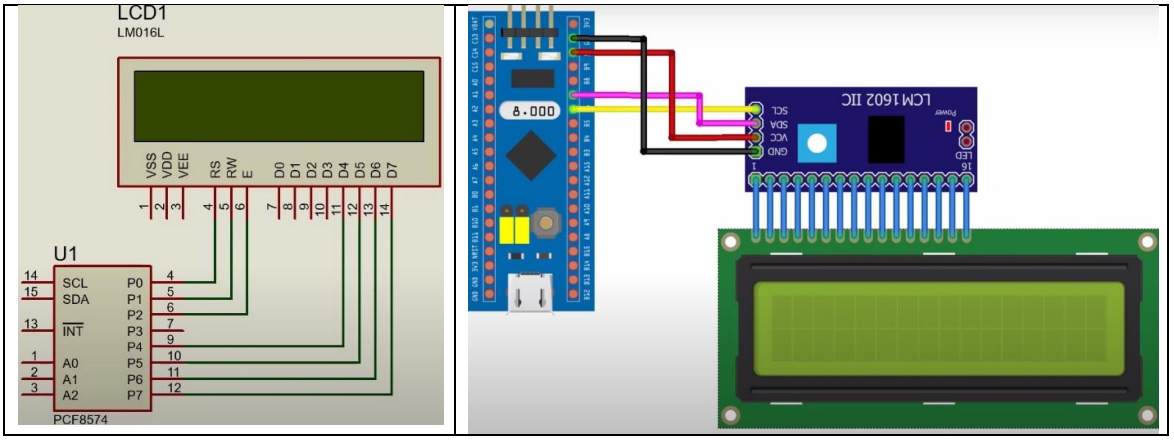


Figura *2*: Conexión interna del LCD con el encapsulado PCF8574 (izquierda) y conexión externa entre el módulo i2c y el LCD

Una cuestión importante a tener en cuenta es que los pines de direccionamiento del chip PCF8574 (A0, A1, A2) se encuentran en nivel alto, por lo que se deberá consultar al datasheet para la dirección de memoria de escritura del esclavo en la comunicación i2c.

Potenciómetro

Es un dispositivo electromecánico utilizado para controlar la resistencia eléctrica de un circuito. El potenciómetro escogido es de tipo B100K, donde la B representa el tipo de curva resistiva y 100K indica una resistencia de $100[k\Omega]$.

Imagen 3: Potenciómetro B100K

El potenciómetro (Imagen *3*) es de tipo rotativo y consta de 3 terminales: el terminal central se contacta al contacto deslizante y los terminales externos se conectan a los extremos de la pista resistiva. Al aplicar una tensión entre los terminales externos, la resistencia entre el terminal central y uno de los extremos varía según la posición del contacto.

A diferencia de los demás componentes electrónicos, este está alimentado a $3[V]$ debido a las características del ADC del microcontrolador, pues con $5[V]$ se producía desbordamiento y las lecturas eran erráticas.

Barra de LED

Se ha escogido un display LED de 10 segmentos multicolor, del fabricante "Seeed Studio". No disponemos del datasheet del producto, aunque sí contamos con un esquema de conexión (Figura 3), que no presenta gran complejidad dado que son 10 diodos individuales.

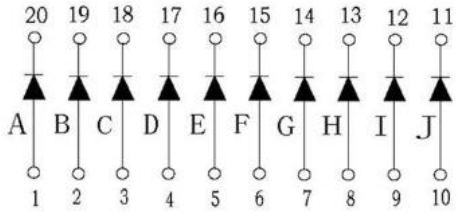


Figura *3*: Configuración interna de la barra de LED

Hemos decidido conectarlos en cátodo común e incluir resistencias de $200[\Omega]$ para limitar la corriente que circulan por los diodos. Hemos estimado que la caída de los diodos es estándar y hemos tomado $1.8[V]$

Teniendo en cuenta que la intensidad de funcionamiento normal de los diodos es $20[mA]$ y están alimentados a $5[V]$, se puede hacer el cálculo de la resistencia auxiliar como:

$$R = \frac{V_{in} - V_{diodo}}{I} \qquad (1)$$

$$\boxed{R = \frac{5 - 1.8}{0.02} = 160[\Omega]}$$

Esta resistencia es algo menor a la conectada por lo que la intensidad que circulen los diodos será menor modificando levemente los niveles de luminosidad de los LED.

Servomotor

Hemos escogido un servomotor MG996R simulando la actuación de un ventilador de manera que cuando la diferencia entre la temperatura de establecimiento y la detectada por el propio sensor sea mayor de 10ºC, éste se mueva con una mayor velocidad.

Las principales ventajas por las que escoger este servomotor MG996R son:

Alta potencia: El MG996R ofrece una alta potencia de torque por lo que puede proporcionar una fuerza considerable.

Durabilidad: Esta diseñado para ser duradero y resistente. Puede soportar cargas pesadas y condiciones de trabajo exigentes.

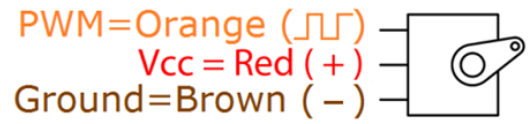
Fácil integración: Es compatible con varios sistemas de control, como Arduino y gran cantidad de microcontroladores.

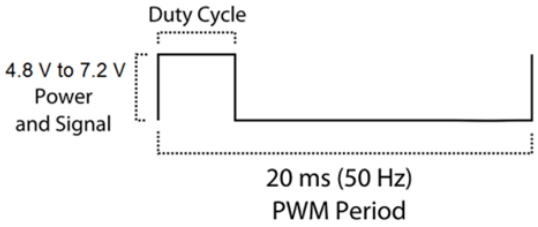
Todas estas ventajas hacen que sea especialmente útil para el caso del ventilador que estamos simulando.

Por último, su costo es asequible en comparación con otros servomotores de alta calidad.

Specifications

- Weight: 55 g
- Dimension: 40.7 x 19.7 x 42.9 mm approx.
- Stall torque: 9.4 kgf·cm (4.8 V), 11 kgf·cm (6 V)
- Operating speed: 0.17 s/60º (4.8 V), 0.14 s/60º (6 V)
- Operating voltage: 4.8 V a 7.2 V
- Running Current 500 mA – 900 mA (6V)
- Stall Current 2.5 A (6V)
- Dead band width: 5 µs
- Stable and shock proof double ball bearing design
- Temperature range: 0 °C – 55 °C





3. Implementación del código

En primer lugar, queremos mencionar que para ayudarnos a entender el funcionamiento del sensor de temperatura y de la pantalla LCD, hemos usado los tutoriales y librerías del creador de contenido “ControllersTech”[2], del cual se recomendaban sus videos en el Moodle de la

[2] Los enlaces a los tutoriales y librerías sin modificar están en el apartado ”Anexo II – Enlaces de interés”

asignatura. Los siguientes apartados contendrán la implementación del código, y los comentarios y modificaciones que hemos hecho a las librerías mencionadas anteriormente.

Muestreo de la temperatura y la humedad relativa: librería "DTH.h"

La librería "DHT.h" es compatible tanto con el sensor DHT-11 como con el DHT-22, pero presenta ciertas diferencias entre un modelo y otro, principalmente en los tiempos que se deben aplicar en cuestiones relacionadas a la inicialización y muestreo periódico de la señal.

Para facilitar el manejo de la información se crea una estructura de datos que reúne la temperatura y humedad. Éstas, son variables "float" aunque se podrían definir como "

uint16_t" debido a que la información se envía en 2 paquetes de 8 bits, como ya comentamos anteriormente.

```
typedef struct
{
        float Temperature;
        float Humidity;
}DHT_DataTypedef;


void DHT_GetData (DHT_DataTypedef *DHT_Data);
```

Esta estructura se pasa como argumento a una función **DHT_GetData()** que se encarga de crear los pulsos de inicialización del sensor, así como de verificar que el sensor ha detectado estos pulsos. Una vez hechas las comprobaciones se procede a realizar la lectura y se asignan los valores a la estructura definida. Esta asignación solo se realizará si el byte de verificación "SUM" es correcto. Por otro lado, si el sensor no ha detectado la petición del microcontrolador, se asignará el máximo valor posible a las variables de la estructura.

```
void DHT_GetData (DHT_DataTypedef *DHT_Data)
{
        DHT_Start ();
        Presence = DHT_Check_Response ();
        if (Presence == 1){
        Rh_byte1 = DHT_Read ();
        Rh_byte2 = DHT_Read ();
        Temp_byte1 = DHT_Read ();
        Temp_byte2 = DHT_Read ();
        SUM = DHT_Read();

        if (SUM == (Rh_byte1+Rh_byte2+Temp_byte1+Temp_byte2)) //Comprobación de
envío correcto
        {
                #if defined(TYPE_DHT11)
                DHT_Data->Temperature = Temp_byte1;
                DHT_Data->Humidity = Rh_byte1;
                #endif

                #if defined(TYPE_DHT22)
                DHT_Data->Temperature = ((Temp_byte1<<8)|Temp_byte2);
                DHT_Data->Humidity = ((Rh_byte1<<8)|Rh_byte2);
                #endif
        }
        }
        else if (Presence == -1){ //Si se produce un error en la detección las señales se ponen a
nivel alto
```

```
                #if defined(TYPE_DHT11)
                        DHT_Data->Temperature = 0xFF;
                        DHT_Data->Humidity = 0xFF;
                #endif

                #if defined(TYPE_DHT22)
                        DHT_Data->Temperature = 0xFFFF;
                        DHT_Data->Humidity = 0xFFFF;
                #endif
        }

}
```

A continuación, pasaremos a describir las funciones que son llamadas dentro de **DHT_GetData()**. Empezaremos por **DHT_Start()**, que se encarga de generar el primer la petición del microcontrolador al sensor:

```
void DHT_Start (void){ //Petición de la placa para recibir datos.
                       //Se crea un pulso a nivel bajo de 1-10ms, seguido de uno a nivel alto de 20-40us. (DTH22)
                       //Tras mandar el pulso se pone el pin como entrada para recibir datos del sensor

        DWT_Delay_Init();
        Set_Pin_Output (DHT_PORT, DHT_PIN);  // set the pin as output
        HAL_GPIO_WritePin (DHT_PORT, DHT_PIN, 0);    // pull the pin low

#if defined(TYPE_DHT11)
        delay (18000);    // wait for 18ms
#endif

#if defined(TYPE_DHT22)
        delay (10000);  // >1ms delay
#endif

    HAL_GPIO_WritePin (DHT_PORT, DHT_PIN, 1);    // pull the pin high
    delay (30);    // wait for 30us
    Set_Pin_Input(DHT_PORT, DHT_PIN);     // set as input
}

uint32_t DWT_Delay_Init(void)
{
  /* Disable TRC */
  CoreDebug->DEMCR &= ~CoreDebug_DEMCR_TRCENA_Msk; // ~0x01000000;
  /* Enable TRC */
  CoreDebug->DEMCR |=  CoreDebug_DEMCR_TRCENA_Msk; // 0x01000000;

  /* Disable clock cycle counter */
  DWT->CTRL &= ~DWT_CTRL_CYCCNTENA_Msk; //~0x00000001;
  /* Enable  clock cycle counter */
  DWT->CTRL |=  DWT_CTRL_CYCCNTENA_Msk; //0x00000001;

  /* Reset the clock cycle counter value */
  DWT->CYCCNT = 0;

     /* 3 NO OPERATION instructions */
     __ASM volatile ("NOP");
     __ASM volatile ("NOP");
     __ASM volatile ("NOP");

  /* Check if clock cycle counter has started */
     if(DWT->CYCCNT)
     {
       return 0; /*clock cycle counter started*/
     }
     else
```

```
  {
    return 1; /*clock cycle counter not started*/
  }
}

__STATIC_INLINE void delay(volatile uint32_t microseconds) //Se crea un delay usando la frecuencia del reloj del sistema
{
  uint32_t clk_cycle_start = DWT->CYCCNT;

  /* Go to number of cycles for system */
  microseconds *= (HAL_RCC_GetHCLKFreq() / 1000000);

  /* Delay till end */
  while ((DWT->CYCCNT - clk_cycle_start) < microseconds);
}

void Set_Pin_Output (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin) //Un pin dado se define como salida
{
        GPIO_InitTypeDef GPIO_InitStruct = {0};
        GPIO_InitStruct.Pin = GPIO_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
}

void Set_Pin_Input (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin) //Un pin dado se define como entrada
{
        GPIO_InitTypeDef GPIO_InitStruct = {0};
        GPIO_InitStruct.Pin = GPIO_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
}
```

Primero de todo, vamos a comentar las funciones **DWT_Delay_Init()** y **delay()**. Estas funciones se usan para establecer la duración de los pulsos que enviamos al sensor, así como comprobar la correcta duración de los pulsos recibidos del sensor. Estas funciones trabajan con los ciclos de reloj de la CPU del microcontrolador, es decir toman la frecuencia del RCC (Reset and Clock Controller). Nosotros hemos configurado el reloj para que el sistema opere a $50[MHz]$, . Para otras frecuencias de reloj habría que modificar el número de ciclos que se han de esperar.

El delay se podría haber implementado también con la función **HAL_Delay()** o con estructuras de control usando **HAL_GetTick()**. Si hemos elegido esta opción es porque así se había implementado en la librería originalmente.

Dicho esto, pasemos a analizar el resto del código. Como la comunicación usa un protocolo serie, se ha de poder modificar el modo de funcionamiento del GPIO del microcontrolador, es decir, cuando se quiera enviar la petición de lectura al sensor el GPIO estará en modo “output” y cuando se quiera recibir estará en modo “input”. Es para esto que se han desarrollado las funciones **Set_Pin_Output()** y **Set_Pin_Input()**.

Con todo esto, ya podemos crear el tren de pulsos que satisfaga las condiciones de detección del sensor. Los intervalos de mantenimiento para el sensor DHT-22 se ven reflejados en la Figura 4. El intervalo coloreado en gris hace referencia a la respuesta del sensor, segmento que se analiza con la función **DHT_Check_Respone()**. Esta función nos devuelve un 0 si la señal no se ha modificado desde que el pin se ha modificado como entrada (lo que indica que el sensor no ha detectado la petición); un -1 si se ha detectado la petición, pero no se han configurado

bien los tiempos de espera (también podría ser un defecto del sensor); y un 1 si se ha detectado la petición de lectura correctamente.

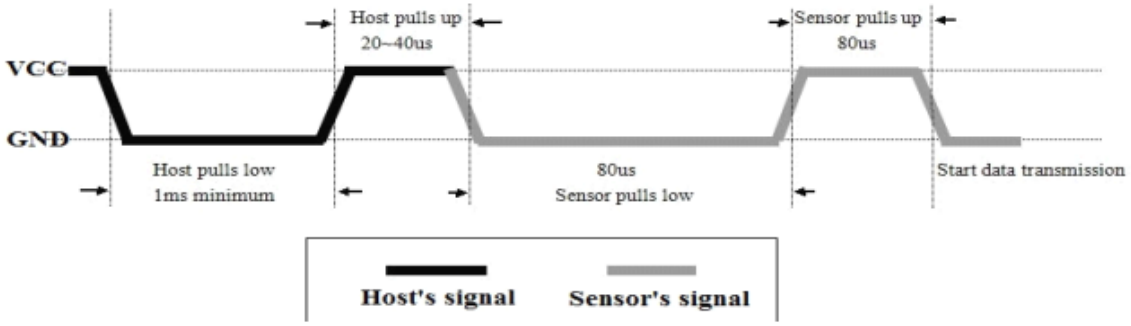


Figura *4*: Tren de pulsos de petición de lectura y verificación de petición recibida

```
uint8_t DHT_Check_Response (void){ //Comprobación de que el sensor ha detectado la petición de la placa
                                                                                    //Devuelve 0 si no lo ha detectado, -1 si el pulso de confirmación es errónea y 1 si es correcto

        uint8_t Response = 0;
        delay (40);
        if (!(HAL_GPIO_ReadPin (DHT_PORT, DHT_PIN)))
        {
                delay (80);
                if ((HAL_GPIO_ReadPin (DHT_PORT, DHT_PIN))) Response = 1;
                else Response = -1;
        }
        while (HAL_GPIO_ReadPin (DHT_PORT, DHT_PIN));    // wait for the pin to go low

        return Response;
}
```

Por último, queda por analizar la función **DHT_Read()**. Para entender dicha función es necesario observar en el datasheet el tren de pulsos resultante al enviar 1 bit de información. La Figura 5 muestra las duraciones de la señal a nivel alto si el bit enviado es un “1” o un “0”. Esto se plasma en el código con la función delay(35), pues transcurrido ese tiempo si la señal sigue estando a nivel alto el bit enviado el un “0” y al contrario si está a nivel bajo.

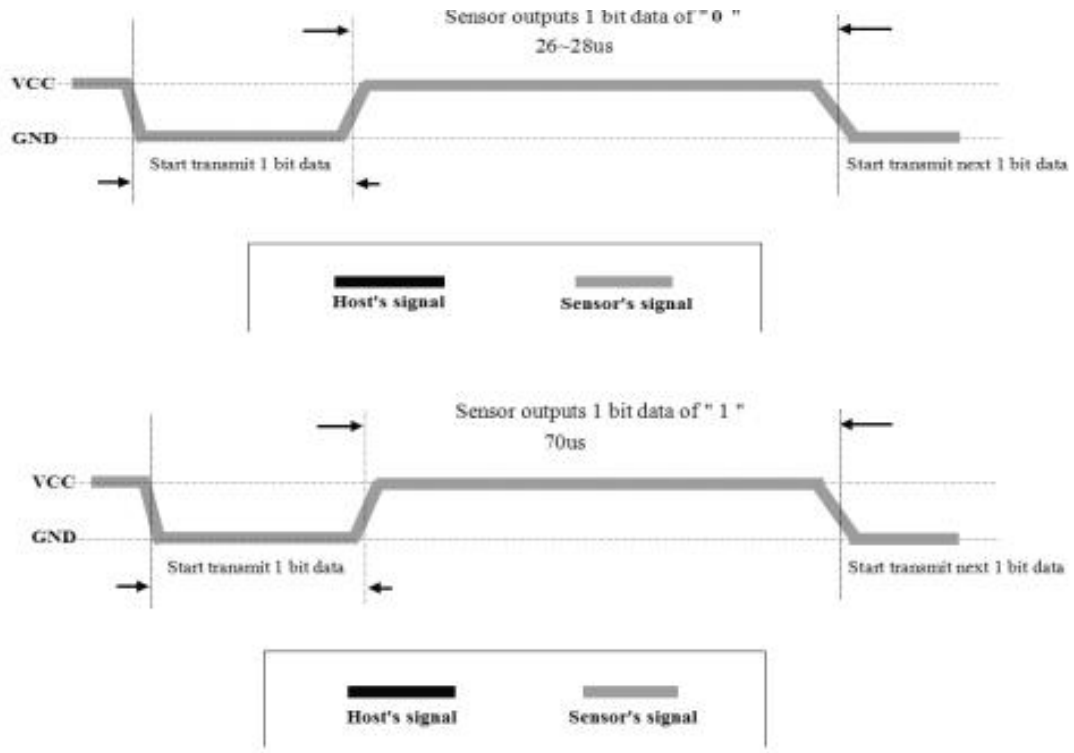


Figura *5*: Tren de pulsos para el envío de un bit a nivel bajo y nivel alto

```
uint8_t DHT_Read (void){ //Lee un byte de datos (1 bit cada 35us). Desplaza y concatena cada bit en una sola variable

        uint8_t i,j;
        for (j=0;j<8;j++)
        {
                while (!(HAL_GPIO_ReadPin (DHT_PORT, DHT_PIN)));    // wait for the pin to go high
                delay (35);    // wait for 35 us
                if (!(HAL_GPIO_ReadPin (DHT_PORT, DHT_PIN)))    // if the pin is low
                {
                        i&= ~(1<<(7-j));    // write 0
                }
                else i|= (1<<(7-j));  // if the pin is high, write 1
                while ((HAL_GPIO_ReadPin (DHT_PORT, DHT_PIN)));  // wait for the pin to go low
        }
        return i;
}
```

Por último, mencionar que este proceso de lectura se repite 8 veces por cada variable para formar el byte de información correspondiente. Una característica negativa de este protocolo es la duración variable a la hora de transmitir la información.

Pantalla LCD 16x2: Librería “i2c-lcd.h”

La pantalla se comunica con el microcontrolador mediante protocolo i2c, donde la pantalla es el esclavo y la placa el master. Como se dijo antes, primero debemos de seleccionar la dirección de escritura del esclavo:

```
#define SLAVE_ADDRESS_LCD 0x4E
```

Dicha dirección corresponde a $[A_0, A_1, A_2] = [1,1,1]$, como se puede comprobar en el datasheet. Ahora se debe de especificar el modo de funcionamiento del display y a continuación inicializarlo. El código que se muestra cumple esa función:

```
void lcd_init (void) //La sucesión de inicialización se encuentra en el datasheet y en la memoria. Se usa el modo de 4 bits
{
        // 4 bit initialisation
        HAL_Delay(50);  // wait for >40ms
        lcd_send_cmd (0x30);
        HAL_Delay(5);  // wait for >4.1ms
        lcd_send_cmd (0x30);
        HAL_Delay(1);  // wait for >100us
        lcd_send_cmd (0x30);
        HAL_Delay(10);
        lcd_send_cmd (0x20);  // 4bit mode
        HAL_Delay(10);

  // dislay initialisation
        lcd_send_cmd (0x28); // Function set --> DL=0 (4 bit mode), N = 1 (2 line display) F = 0 (5x8 characters)
        HAL_Delay(1);
        lcd_send_cmd (0x08); //Display on/off control --> D=0,C=0, B=0  ---> display off
        HAL_Delay(1);
        lcd_send_cmd (0x01);  // clear display
        HAL_Delay(1);
        HAL_Delay(1);
        lcd_send_cmd (0x06); //Entry mode set --> I/D = 1 (increment cursor) & S = 0 (no shift)
        HAL_Delay(1);
        lcd_send_cmd (0x0C); //Display on/off control --> D = 1, C and B = 0. (Cursor and blink, last two bits)
}

void lcd_send_cmd (char cmd)
{
  char data_u, data_l;
        uint8_t data_t[4];
        data_u = (cmd&0xf0);
        data_l = ((cmd<<4)&0xf0);
        data_t[0] = data_u|0x0C;  //en=1, rs=0
        data_t[1] = data_u|0x08;  //en=0, rs=0
        data_t[2] = data_l|0x0C;  //en=1, rs=0
        data_t[3] = data_l|0x08;  //en=0, rs=0
        HAL_I2C_Master_Transmit (&hi2c1, SLAVE_ADDRESS_LCD,(uint8_t *) data_t, 4, 100);
}
```

La función **lcd_init()** consiste en una recopilación de comandos que se tienen que enviar cuando se inicializa el display. Dentro de esta, se está empleando la función **lcd_send_cmd()** que recibe como argumentos variables de tamaño byte, no obstante, dentro se debe de hacer la división en 2 conjuntos de 4 bits debido a que estamos en el modo "4 bits".

El protcolo i2c trabaja con bloques de bytes por lo que debemos de realizar unas modificaciones a los conjuntos de 4 bits. Para poder enviar comandos al LCD falta por especificar la configuración de los pines de habilitación y escritura de este. Es por eso que se realiza la unión de los subconjuntos de 4 bits con la configuración seleccionada.

Como resultado, para enviar 1 byte de información se necesitan enviar 4 bytes, donde 2 de ellos contienen la información de 4 bits y las instrucciones de habilitación, y los otros 2 la información de los 4 restantes.

Dicho esto, en la Figura 6 se puede apreciar la secuencia de comandos que inicializan el LCD y se verifica que son los mismos que se envían con el código mostrado.

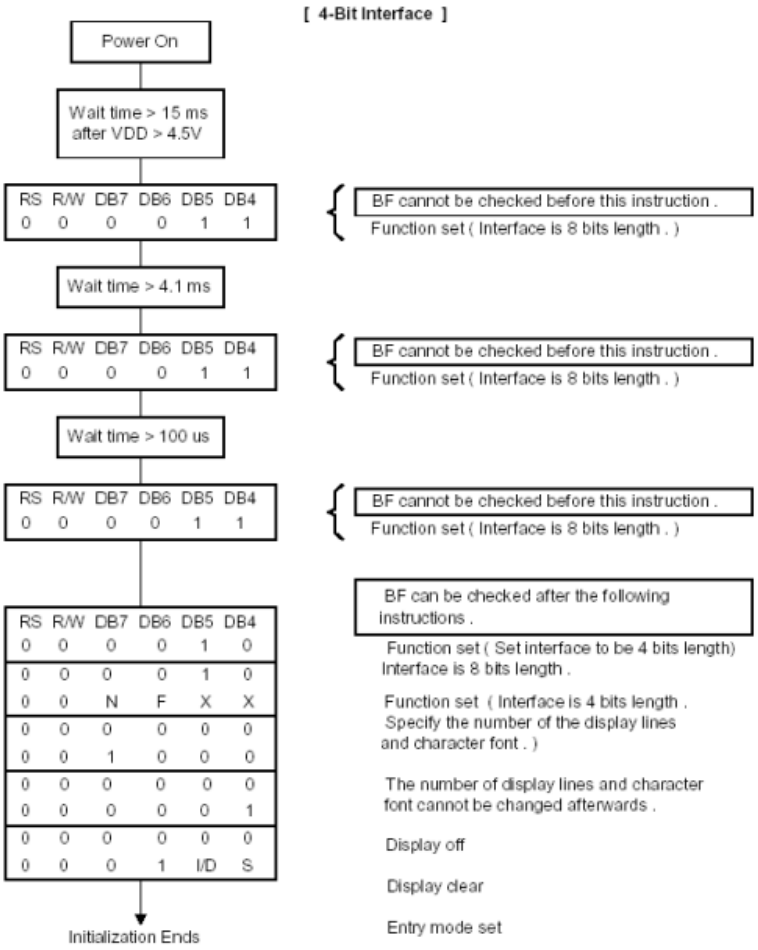


Figura *6*: Secuencia de inicialización del LCD

Si lo que queremos es enviar datos en vez de comandos deberemos de modificar la configuración de los pines de habilitación del LCD. Esto se hace con la función **lcd_send_data()** y la única diferencia con **lcd_send_cmd()** es el bit rs = 1.

```
void lcd_send_data (char data)
{
        char data_u, data_l;
        uint8_t data_t[4];
        data_u = (data&0xf0);
        data_l = ((data<<4)&0xf0);
        data_t[0] = data_u|0x0D;  //en=1, rs=0
        data_t[1] = data_u|0x09;  //en=0, rs=0
        data_t[2] = data_l|0x0D;  //en=1, rs=0
        data_t[3] = data_l|0x09;  //en=0, rs=0
        HAL_I2C_Master_Transmit (&hi2c1, SLAVE_ADDRESS_LCD,(uint8_t *) data_t,
4, 100);
}
```

Ahora pasaremos a tratar la función **lcd_put_cur()**,la cual te permite poner el cursor en una posición que especifiquemos. Tiene como entradas las filas y columnas. En la Figura 7 se ve la dirección de cada posición.

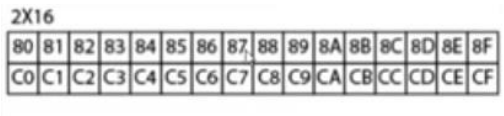


Figura *7*: Direcciones de cada posición del cursor

```
void lcd_put_cur(int row, int col) //row [0,1], col[0,15]
{
    switch (row)
    {
        case 0:
            col |= 0x80;
            break;
        case 1:
            col |= 0xC0;
            break;
    }

    lcd_send_cmd (col);
}
```

Para limpiar el contenido de la pantalla usamos las funciones **lcd_clear_row()** y **lcd_clear()**. La primera puede borrar el contenido de una fila que especifiquemos y la segunda borra el contenido completo.

```
void lcd_clear_row(int row)
{
        lcd_put_cur(row , 0);
        for (int i=0; i<35; i++)
        {
                lcd_send_data (' ');
        }
}

void lcd_clear (void)
{
        lcd_clear_row(1);
        lcd_clear_row(2);
}
```

Por último, vamos a mostrar las funciones **Display_Rh()** y **Dispaly_Temp()**. Estas usan las anteriores citadas para mostrar por pantalla la temperatura y la humedad relativa. Como primera acción, se realiza la limpieza del display lo cual ocasiona algunos problemas con la frecuencia a la se refresca la pantalla, pues se origina un efecto visual indeseado. En la implementación del "main.c" se han incluido bucles con la intención de disminuir esta frecuencia y obtener una imagen constante. Quizás hubiese sido mejor no incluir la limpieza cada vez que se muestrea.

```
void Display_Rh(float Rh){   // Antes de mostrar el dato se limpia la fila

        char str[20] = {0};
        lcd_clear_row(1);
        lcd_put_cur(1,0);

        sprintf(str,"RH:%.2f",Rh/10);
        lcd_send_string(str);
        lcd_send_data('%');
}

void Display_Temp(float Temp, int row){   // Antes de mostrar el dato se limpia la fila

        char str[20] = {0};
        lcd_clear_row(row);
        lcd_put_cur(row,0);

        sprintf(str,"TEMP:%.2f",Temp/10);
        lcd_send_string(str);
        lcd_send_data('C');
}

void lcd_send_string (char *str)
```

```
{
        while (*str) lcd_send_data (*str++);
}
```

Mencionar que en la función **Display_Temp()** podemos seleccionar la fila del cursor porque nos interesaba poder cambairlo. En este extracto de código también se ha incluido la función **lcd_send_string()** que recorre una cadena de caracteres y los va enviando uno a uno (el tipo char es de tamaño byte).

Interrupciones: externas e internas

Las interrupciones se han usado para poder establecer una nueva temperatura de establecimiento y para realizar un muestreo periódico con el sensor. La primera corresponde a una interrupción externa que habilita el usuario pulsando el "user button" de la placa STM32F411 (PA0) y la segunda corresponde a una interrupción interna habilitada con un temporizador (TIM2).

Interrupción externa: Se ha definido un "marcador" de tipo volátil int que modifica su valor cuando el usuario presiona el "user button". Es importante que se defina como volátil para que el compilador no realice simplificaciones en el código que puedan alterar su funcionalidad.

Para detectar la pulsación correcta del botón, se ha incluido un "debouncer" para eliminar rebotes en la señal. La función **debouncer()** se ha cogido del repositorio de Alberto Brunete.

```
volatile int Seleccion_estado = 0; //Se emplea para cambiar el estado y seleccionar la temperatura que queremos

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) // Genera la interrupción cuando se pulsa el "user boton" de la placa (PA0)
{
        //Interrupcion debida al PIN_0
        if(GPIO_Pin==GPIO_PIN_0){
        Seleccion_estado = 1;
        }
}

int debouncer(volatile int* button_int, GPIO_TypeDef* GPIO_port, uint16_t GPIO_number){
        static uint8_t button_count=0;
        static int counter=0;

        if (*button_int==1){
                if (button_count==0) {
                        counter=HAL_GetTick();
                        button_count++;
                }
                if (HAL_GetTick()-counter>=20){
                        counter=HAL_GetTick();
                        if (HAL_GPIO_ReadPin(GPIO_port, GPIO_number)!=1){
                                button_count=1;
                        }
                        else{
                                button_count++;
                        }
                        if (button_count==4){ //Periodo antirebotes
                                button_count=0;
                                *button_int=0;
                                return 1;
```

```
                                }
                        }
                }
                return 0;
}

main(){
                while(1){
                        if(debouncer(&Seleccion_estado, GPIOA, GPIO_PIN_0)){ //Estado
para la eleccion de la temperatura de control
                                while(!debouncer(&Seleccion_estado, GPIOA,
GPIO_PIN_0)){
                                }
                        {
                        else{}
                }
}
```

Se ha incluido también la estructura funcional del main. Cuando se activa el "marcador" se pasa a un estado de selección de la temperatura de control y mientras no se vuelva a pulsar el botón se permanece en este estado. Se ha podido implementar así debido a que el marcador se limpia en la función **debouncer()** con lo que solo se detecta el flanco en el pulsador.

También recalcar que no se pasa automáticamente a este estado de selección, sino que se debe esperar a que termine la ejecución de la rama secundaria del "if". Se ha dejado de esta manera puesto que hemos considerado que el cambio de temperatura de control no es una tarea de prioritaria. Si hubiésemos querido hacerla prioritaria deberíamos activar las interrupciones del ADC.

Interrupción interna: A diferencia de en el caso anterior, este tipo de interrupción la hemos considerado prioritaria pues la acción de control depende directamente de la lectura de las variables de salida por lo que se deben de actualizar con la mayor rapidez posible. Se ha decidido tomar muestras de la temperatura y de la humedad relativa cada 5 segundos. Para calcular el periodo de un temporizador se usa la siguiente expresión:

$$T = \left(\frac{f_{CLK}}{(Prescaler + 1) \cdot (Period + 1)}\right)^{-1} \qquad (2)$$

Simplificando la expresión e introduciendo datos tenemos:

$$\boxed{T = \left(\frac{50[MHz]}{50000 \cdot 5000}\right)^{-1} \approx 5[s]}$$

El código empleado es:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) // Genera una
interrupción cada 5 segundos
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(htim);
  /* NOTE : This function Should not be modified, when the callback is needed,
        the __HAL_TIM_PeriodElapsedCallback could be implemented in the user file */

  /* Obtencion de datos del sensor */
  /* Ver libreria "DHT.h" */
  DHT_GetData(&DHT22_Data);

  /* Asignación de variables globales de Temperatura y Humedad*/
```

```
  Temperature = DHT22_Data.Temperature;
  Humidity = DHT22_Data.Humidity;

}
```

Manipulación de la temperatura de establecimiento

Como ya se ha ido mencionando en otros apartados, la temperatura de control se modifica con un potenciómetro por lo que tenemos que habilitar el ADC y realizar una calibración inicial para ajustar a cada posición del potenciómetro una temperatura determinada. Esto último se puede ver como una interpolación lineal, es decir la ecuación de una recta. El código mostrado es el que va dentro de la rama principal del "if".

```
lcd_clear();
lcd_put_cur(0,0);
lcd_send_string("Choose new temp");
int i = 1;

while(!debouncer(&Seleccion_estado, GPIOA, GPIO_PIN_0)){

        //Habilitacion del conversor
        HAL_ADC_Start(&hadc1);

        //Lectura del conversor
        if(HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY) == HAL_OK)
                Lec_Temp_Control = HAL_ADC_GetValue(&hadc1);

                //Asignación de la temperatura de control
                Temp_control = (float)Lec_Temp_Control / 255 * T_diff + 15;
//La pendiente de la interpolación es 25/255

                //Mostramos por pantalla la temperatura que escogemos
                //Se ha añadido el bucle para reducir la frecuencia de la pantalla y evitar el parpadeo. Se
ha tomado un valor empírico
                if (!(i++ % 20000)){
                        Display_Temp(Temp_control*10 , 1);
                        i = 1;
                }
        }
        //Deshabilitacion del conversor
        HAL_ADC_Stop(&hadc1);
```

T_diff es la diferencia entre la temperatura máxima que se prevé alcanzar y la mínima. En nuestro caso hemo escogido 40ºC y 15ºC respectivamente.

```
/*VARIABLES GLOBALES*/
static uint8_t T_min = 15 , T_max = 40 ; //Se definen unos márgenes de funcionamiento que se
pueden cambiar
uint8_t T_diff = 0; ; //Diferencia entre los márgenes de temperatura. Se emplea para la interpolación
del potenciómetro

/* DENTRO DEL MAIN*/
T_diff = T_max - T_min ;
```

Indicación luminosa de la temperatura mediante una barra de LED

La implementación de la barra de LEDs nos sirve como representación visual del estado del sistema, mostrando así la información de la temperatura de la sala de una manera más intuitiva.

A continuación, se presentan las definiciones correspondientes a la barra de LEDs:

```
#define NUMERO_LED 10

typedef struct PINES{
        GPIO_TypeDef *GPIOx;
        uint16_t GPIO_Pin;
}PINES;
```

```
PINES PINES_LED[] = {{GPIOB,GPIO_PIN_14},{GPIOB,GPIO_PIN_15},{GPIOD,GPIO_PIN_8},
                     {GPIOD,GPIO_PIN_9},{GPIOD,GPIO_PIN_10},{GPIOD,GPIO_PIN_11},
                     {GPIOD,GPIO_PIN_12},{GPIOD,GPIO_PIN_13},{GPIOD,GPIO_PIN_14},
                     {GPIOD,GPIO_PIN_15}};
```

Estas definiciones asignan los pines de salida del microcontrolador STM32 a los pines correspondientes de la barra de LEDs. En este caso, se ha definido una estructura PINES, la cual cuarda el tipo de Pin y el numero de este para poder pasarle estos datos a la función **'encenderled()'**

A continuación, se muestra el código correspondiente a la implementación de la barra, donde se ha definido a función **'encenderled'** , esta se encarga de enceder los LEDs de la barra. Recibe como parámetro **'i'** y '**PINES_LED[]'** tipo PINES.

```
void encenderled(int i, PINES PINES_LED[]);
```

```
void encenderled(int i, PINES PINES_LED[]){

        int j = 0;

        for(int n = 0 ; n < i ; n++){
                HAL_GPIO_WritePin(PINES_LED[n].GPIOx, PINES_LED[n].GPIO_Pin, GPIO_PIN_SET);
                j++;
        }//End_FOR

        while(j < NUMERO_LED){
                HAL_GPIO_WritePin(PINES_LED[j].GPIOx, PINES_LED[j].GPIO_Pin, GPIO_PIN_RESET);
                j++;
        }//End_WHILE
};
```

El bloque principal, se ha establecido un salto térmico dependiendo de los límites de temperatura escogidos, en nuestro caso 10 como temperatura mínima y 40 como temperatura máxima, dependiendo de esta temperatura se encenderán más o menos LEDs

Por ejemplo, si la temperatura es menor que 35 grados, se encenderán todas las barras de LEDs excepto las 2 últimas.

Simulación del control de la temperatura: comportamiento del servomotor

Para regular la temperatura, utilizamos el servomotor a modo de ventilador. En primer lugar, hemos definido las variables de control del servo, definiendo una velocidad mínima, una velocidad máxima y un rango de temperatura:

```
// Variables de control del servo
float velocidadMinima = 1.0;  // Valor mínimo de velocidad
float velocidadMaxima = 10.0; // Valor máximo de velocidad
float rangoTemperatura = 10.0; // Rango de temperatura para
```

Posteriormente, desarrollamos la función para calcular la velocidad en función de la diferencia de temperatura. Con esto, conseguimos que, si la diferencia de temperatura es superior a 10 grados, la velocidad del servo sea máxima, para que el ventilador gire a mayor velocidad regulando así la temperatura.

```
// Función para calcular la velocidad en función de la dif
float calcularVelocidad(float diferenciaTemperatura) {
    if (diferenciaTemperatura <= 0) {
        return velocidadMinima; // Sin diferencia de tempe
    } else if (diferenciaTemperatura >= rangoTemperatura)
        return velocidadMaxima; // Diferencia máxima de te
    } else {
        // Interpolación lineal para calcular la velocidac
        float pendiente = (velocidadMaxima - velocidadMini
        return velocidadMinima + (pendiente * diferenciaTe
    }
```

Ajustamos la posición del servo según la velocidad:

```
// Función para ajustar la posición del servo según la veloc
void ajustarServo(float velocidad) {
```

Por último, dentro de la función principal, calculamos la diferencia de temperatura, la velocidad en función de la diferencia de temperatura y ajustamos la posición del servo según la velocidad calculada.

```
float diferenciaTemperatura = temperaturaEstablecimiento - t

// Calcular la velocidad en función de la diferencia de temp
float velocidad = calcularVelocidad(diferenciaTemperatura);

// Ajustar la posición del servo según la velocidad calculad
ajustarServo(velocidad);
```

Para la conexión del servomotor nos hemos guiado por el repositorio bitbucket de moodle en la parte de Documentación STM32F4-Discovery / control de motores.

4. Resumen de conexiones

En la siguiente tabla se aglutina la información de las conexiones entre el microcontrolador y los componentes que se mencionan en el apartado "**2. Componentes electrónicos**". Se especifica el pin/pines mediante el cual están unidos a la placa, el modo de funcionamiento de dicho pin y una descripción para detallar la configuración escogida.

Además, se incluye una columna para la tensión de alimentación pues todos los componentes estas alimentados por el microcontrolador ya que no demandan una gran potencia y no hay peligro de sobrecarga.

Componente	Tensión (V)	Pin	Modo de funcionamiento	Descripción
Sensor DHT-22	5	PA2	ADC1_IN2	
Pantalla LCD con módulo i2c	5	PB9 PB8	I2C1_SDA I2C1_SCL	GPIO mode: Alternate Funcion No pull-up no pull-down Maximun output speed: Very High
Potenciómetro	3	PA0 PA1	GPIO_EXTI0 GPIO_Output	
Barra de LED	5	PD8..-..PD15	GPIO_Output	
Servomotor	5	PC7,PC9,PE6	INI1,INI2 Enable	

Tabla 1: Resumen de conexiones con el microcontrolador

5. Anexo I - Presupuesto

La construcción del diseño final conllevaría los siguientes gastos:

Concepto	Coste unitario	Cantidad	Coste total
Microcontrolador STM32F411	15,00 €	1	15,00 €
Sensor DHT-22	8,14€	1	8,14€
Pantalla LCD con módulo i2c	6,51 €	1	6,51 €
Potenciómetro	1,31 €	1	1,31 €
Barra de LED	1,71 €	1	1,71 €
Resistencia 200$[\Omega]$	0,018€	10	0,18€
Resistencia 30$[k\Omega]$	0,044€	3	0,14€
Cables	3,26 €	1	3,26 €
Servomotor	3,1	1	3,10
		TOTAL:	**39,35€**

Tabla 2: Presupuesto

Se ha considerado como si hubiésemos usado un pack de 40 cables para realizar alguna aproximación del coste.

6. Anexo II - Enlaces de interés

Repositorio de GitHub

Sensor-temperatura [Online]: Cnbados55380/Sensor-temperatura (github.com) [Accedido en junio de 2023].

Datasheet de componentes

Microcontrolador STM32F411 [Online]: STM32F411 - PDF Documentation [Accedido en junio de 2023].

Sensor DHT-22 [Online]: Digital+humidity+and+temperature+sensor+AM2302.pdf (adafruit.com) [Accedido en junio de 2023].

Pantalla LCD [Online]: 没有幻灯片标题 (waveshare.com) [Accedido en junio de 2023].

Encapsulado PCF8574 [Online]: ti.com/lit/ds/symlink/pcf8574.pdf [Accedido en junio de 2023].

Servomotor… [Online]: MG996R Tower-Pro (components101.com) [Accedido en junio de 2023].

Tutoriales y material consultado

Videos de ControllersTech:

- (15) How Did I write the I2C-LCD Library || Explained - YouTube
- (15) LCD via I2C in STM32 || CUBEIDE || PCF8574 - YouTube
- (15) DHT11 || DHT22 || DS18B20 with STM32 using TIMER Delay - YouTube
- https://bitbucket.org/abrunete/sistemas-electronicos-digitales/

Librerías:

- How to use DHT22 with STM32 » ControllersTech
- LCD 16x2 via I2C with STM32 (controllerstech.com)