



eetac

Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Sistema sensor autónomo inalámbrico de bajo consumo para el control de la iluminación y riego de cultivos en invernaderos

TITULACIÓN: Grau en Ingeniería de Sistemas de Telecomunicaciones

AUTOR: Alejandro García Moreno

DIRECTOR: Francesc Josep Robert Sanxis

FECHA: 1 de septiembre del 2016

Título: Sistema sensor autónomo inalámbrico de bajo consumo para el control de la iluminación y riego de cultivos en invernaderos

Autor: Alejandro García Moreno

Director: Francesc Josep Robert Sanxis

Fecha: 1 de septiembre del 2016

Resumen

El proyecto trata de implementar un sistema de control de riego y lumínico para un invernadero. Para realizar dicho control, el sistema deberá de medir con sensores diferentes parámetros como la temperatura ambiente, la humedad relativa del aire, la humedad de la tierra y la intensidad lumínica. Una vez medidos estos valores, el sistema sensor deberá de enviar las ordenes correspondientes a los diferentes actuadores y además enviar, vía Wireless, esos datos a nuestra aplicación Labview para ser mostrados por pantalla y poder realizar un estudio de estos parámetros en base al tiempo.

El sistema sensor se tendrá que realizar con un microcontrolador PIC18 de bajo consumo para que el sistema pueda ser autónomo con poca energía de alimentación. El PIC será programado en C usando el estilo de programación FSM. Los primeros programas se simularán en el simulador virtual Proteus para poder detectar bugs de una manera más eficaz. Una vez realizadas las simulaciones, el programa será cargado en nuestro PIC físico en nuestra placa de pruebas donde comprobaremos que todo lo simulado funciona.

En la elección de sensores, se intentará trabajar con sensores analógicos y digitales indistintamente. Respecto los sensores con interface digital, será importante realizar un estudio de sus protocolos de comunicación.

Finalmente, después de realizar el prototipo final en nuestra placa de pruebas, se diseñará con Eagle la correspondiente placa de circuito impreso(PCB) de nuestro sistema sensor y ,a poder ser, también se hará el mecanizado de la misma.

Title: Sensor system of irrigation and lighting control for a greenhouse

Author: Alejandro Garcia Moreno

Director: Francesc Josep Robert Sanxis

Date: September 1 st 2016

Overview

The project aims to implement a system of irrigation and lighting control for a greenhouse. To perform such control, the system must sensors to measure various parameters such as ambient temperature, relative humidity, ground humidity and light intensity. Once measured these values, the sensor system must send orders for the different actuators and also send ,via Wireless, these data to our Labview application to be displayed on the screen and be able to study these parameters based on time.

The sensor system must be performed with a low power microcontroller PIC18 for the system to be autonomous with little energy supply. The PIC will be programmed in C using the FSM style programming. The first programs will be simulated in the virtual simulator Proteus to detect bugs more effectively. After performing the simulations, the program will be loaded into our physical PIC on our testboard where we find that everything works as we simulated.

In choosing sensors, it will try to work with either analog and digital sensors. Regarding the sensors with digital interface, it is important to conduct a study of its communication protocols.

Finally, after making the final prototype in our testboard, our printed circuit board (PCB) of our sensor system will be designed with Eagle and, if possible, the machining of the PCB will be done as well.

ÍNDICE

Introducción.....	1
Capítulo 1. Especificaciones del proyecto	2
1.1 Lista de especificaciones	2
1.2 Teoría general.....	2
1.2.1 Parámetros de entorno a medir	3
1.2.2 Comunicación inalámbrica.....	5
1.2.3 Energy Harvesting	6
1.2.4 Actuadores a utilizar	6
1.3 Diagrama general del sistema.....	7
Capítulo 2. Planificación	9
2.1 Fase 1 del proyecto.....	10
2.1.1 Programa en C	11
2.2 Fase 2 del proyecto.....	14
2.2.1 Comunicación con el sensor DHT22	15
2.2.2 Comunicación I ² C SHT25.....	15
2.2.3 Modificación de nuestro programa en C fase 2	16
2.3 Fase 3 del proyecto.....	18
2.3.1 Modificación de nuestro programa en C fase 3	18
Capítulo 3. Desarrollo	19
3.1 Desarrollo de la Fase 1	19
3.1.1 Interrupción INT0IF	19
3.1.2 Muestreo del ADC con el Timer3.....	20
3.2 Desarrollo de la Fase 2	20
3.2.1 Comunicación con el sensor DHT22	20
3.2.2 Comunicación I ² C	21
3.3 Desarrollo de la Fase 3	23
3.3.1 PWM.....	23
3.3.2 Comunicación serial	24
Capítulo 4. Implementación del sistema	25
4.1 Elección de los sensores.....	25
4.1.1 Temperatura Analog KY-013	25
4.1.2 Humedad Tierra Analog.....	26
4.1.3 Iluminancia Analog	27
4.1.4 Temperatura Digital I ² C MCP9808	29
4.1.5 Temperatura y Humedad Relativa Digital OneWire Protocol DHT22	31

4.2	Implementación de nuestra aplicación Labview	32
4.2.1	Funcionalidades.....	32
4.2.2	Diseño de la aplicación	33
4.3	Implementación de sistema sensor en placa de pruebas.....	34
4.3.1	Esquema de conexión de los sensores con el MCU.....	36
4.3.2	Esquema de conexión PICKIT y UART-BLUETOOTH	37
4.3.3	Esquema de alimentación de la Development Board	38
4.4	Implementación del sistema sensor en PCB con Eagle	40
4.4.1	Schematic.....	40
4.4.2	Board	41
4.4.3	Mecanizado	44
Capítulo 5.	Conclusiones	45
5.1	Mejoras en la alimentación de los sensores.....	45
5.2	Problemas de la comunicación Bluetooth	46
5.3	Implementación de un sistema de Energy Harvesting	47
5.3.1	Usar Zigbee	47
5.3.2	Implementar un sistema de almacenamiento de energía	48
Capítulo 6.	Referencias	49
Capítulo 7.	ANEXOS	51
7.1	Módulos del PIC18F26K20.....	51
7.1.1	TIMER3.....	51
7.1.2	PWM con TIMER2.....	53
7.1.3	Comunicación I ² C	56
7.1.4	EUSART	59
7.1.5	Sleep Mode.....	61
7.1.6	ADC-10bits	63
7.2	Sensores	65
7.2.1	DHT22	65
7.3	Proceso de mecanizado de la PCB.....	69
7.3.1	CAM.....	69
7.3.2	Drills.....	69
7.3.3	Metalizado	70
7.3.4	Pistas TOP y BOTTOM.....	72
7.4	Datos capturados durante 2h30min	73
7.4.1	Iluminancia.....	74
7.4.2	Temperaturas	74
7.4.3	Humedad Relativa	75

7.4.4	Humedad de la tierra	76
7.5	Código integro en C del programa principal	77

Introducción

Actualmente se están diseñando sistemas autónomos de bajo consumo de todo tipo ya sea para medir parámetros medioambientales, controlar el entorno de una ciudad, controlar ritmo cardiovascular, etc. Nosotros hemos querido implementar un sistema autónomo de bajo consumos e inalámbrico para realizar el control de riego y lumínico de un invernadero.

El proyecto se dividirá en 4 capítulos: Especificaciones, Planificación, Desarrollo e Implementación.

En la parte de *Especificaciones* hablaremos de los objetivos que debemos de cumplir y de la teoría general que hay detrás del sistema que queremos implementar.

En el proceso de *Planificación y Desarrollo* dividiremos en 3 fases el desarrollo del proyecto realizando un primer proyecto sencillo el cual iremos mejorando en las siguientes fases de desarrollo. Cuando acabemos la planificación de una fase del proyecto realizaremos simulaciones con *Proteus* en el capítulo de Desarrollo.

En el capítulo de *Implementación* pondremos en funcionamiento todo lo que hemos desarrollado para que funcione correctamente en sistema electrónico que implementaremos. También implementaremos nuestra aplicación de supervisión con labview utilizando Bluetooth como vía de comunicación.

Finalmente en las conclusiones hablaremos de cómo funciona el sistema y qué se puede mejorar.

Capítulo 1. Especificaciones del proyecto

1.1 *Lista de especificaciones*

El proyecto tiene que cumplir los siguientes objetivos:

- Sistema sensor capaz de medir magnitudes relacionadas con los cultivos en invernaderos

El sistema sensor tiene que ser capaz de medir, tanto con sensores analógicos o digitales, las magnitudes físicas necesarias para, posteriormente, controlar el riego y realizar el control de la radiación solar que reciben las plantas.

- Implementación del sistema sensor y de control de actuadores con un MCU PIC18F26K20 extreme low power

La obtención de los parámetros medidos se realizarán con un MCU y los actuadores serán controlados por el MCU.

- Comunicación inalámbrica con nuestro sistema sensor

Nos podremos comunicar con el sistema sensor de forma inalámbrica para realizar la monitorización de los parámetros medidos.

- Sistema sensor autónomo y de bajo consumo

Se procurará que el sistema funcione de manera autónoma y que consuma la menor energía posible.

- Monitorización de los parámetros del invernadero con una aplicación de Labview

La aplicación de Labview deberá de mostrar los valores medidos. A poder ser, que los parámetros se puedan representar en una gráfica para poder realizar estudio en base al tiempo.

1.2 *Teoría general*

Actualmente la mayoría de los sistemas de riego actuales realizan riegos programados, 1 o 2 veces al día según la estación del año. Nuestro sistema sensor pretende controlar de forma autónoma el riego y la radiación solar recibida según las condiciones de entorno en tiempo real y no de forma programada.

Detrás de todo lo que implementaremos existe una base científico-técnica que explicaremos a continuación.

1.2.1 Parámetros de entorno a medir

Nuestro sistema sensor pretende controlar el riego y el control lumínico de un invernadero de forma autónoma. Para conseguirlo es necesario poder medir las magnitudes físicas necesarias para nuestro cometido.

Control de Riego:

Para realizar el control de riego tendremos en cuenta los parámetros de temperatura ambiente, humedad relativa ambiental y humedad de la tierra. Y según la relación de estos 3 parámetros decidiremos cuándo regar o no.[2]

I. Temperatura ambiente

Existen muchos tipos de sensores de temperatura, a continuación mostraremos explicaremos el funcionamiento de los más utilizados para medir temperatura.

1. Termopares

Un termopar es un sensor para medir la temperatura. Se compone de dos metales diferentes, unidos en un extremo. Cuando la unión de los dos metales se calienta o enfriá, se produce una tensión que es proporcional a la temperatura. Las aleaciones de termopar están comúnmente disponibles como alambre.[3]

2. RTD

Un RTD es un detector de temperatura resistivo, es decir, un sensor de temperatura basado en la variación de la resistencia de un conductor con la temperatura.

3. Termistor NTC y PTC

Un termistor es un sensor de temperatura por resistencia. Su funcionamiento se basa en la variación de la resistividad que presenta un semiconductor con la temperatura. Existen dos tipos de termistor: los NTC de coeficiente negativo y los PTC de coeficiente positivo.

II. Humedad relativa

La humedad relativa depende de la temperatura ambiente y la cantidad moléculas de H_2O que hay en el aire. El sensor de escojamos, además de medir cuantas moléculas de H_2O contiene el aire, tendrá que medir la temperatura ambiente para que haga la conversión a humedad relativa.

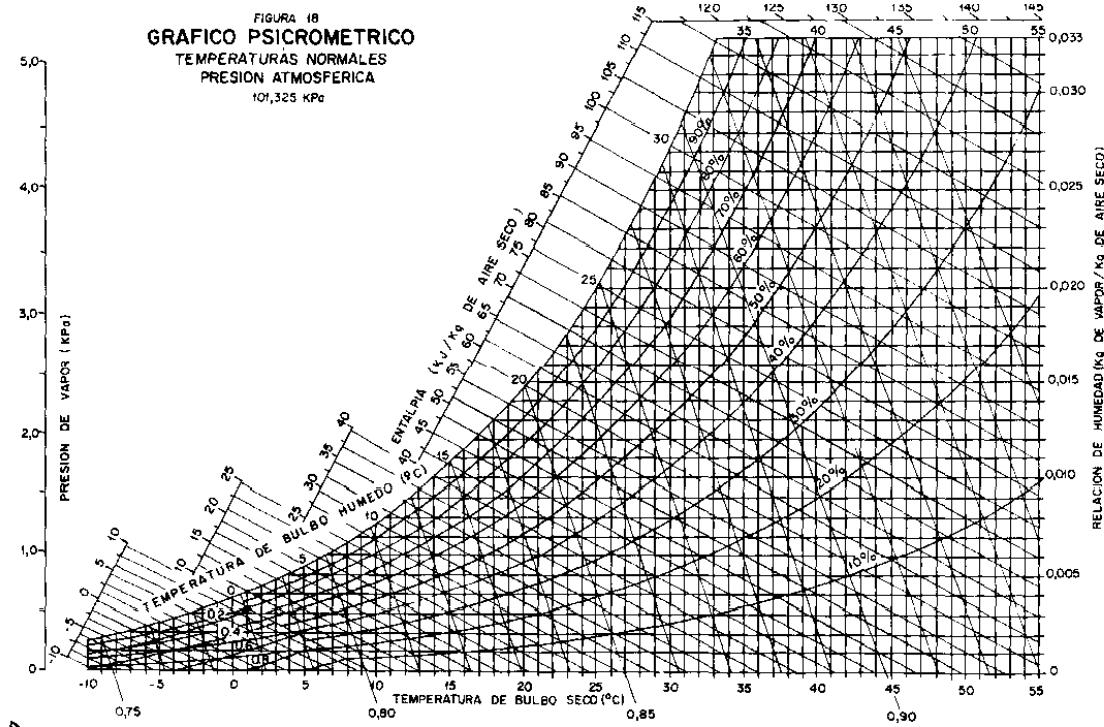


Fig. 1.1 Gráfica de la relación humedad relativa-temperatura[1]

La **Fig. 1.1** representa que cuando la temperatura ambiente es mayor, mayor es la cantidad de moléculas que puede contener el aire.

III. Humedad de la tierra

La humedad de la tierra es la cantidad de moléculas de agua que hay por m^2 de tierra pero con un sensor eso no lo podemos medir. Lo que sí que podemos hacer es medir la conductividad de la tierra.

Control lumínico:

I. Intensidad lumínica

Para realizar el control lumínico tendremos en cuenta la Intensidad lumínica que recibimos del sol.

La unidad del sistema internacional de la intensidad lumínica es la candela (I_v , símbolo cd). Esta unidad de medida es proporcional a lumens por estereoradianes ($I_v = lm/sr$). Pero la unidad que nos será más cómoda para trabajar será la iluminancia (E_v , símbolo lux), en la cual E_v es proporcional a lumens por metro cuadrado ($E_v = lm/m^2$).

Un LDR es una resistencia que varía su resistividad según la luz que capte. Un fabricante de LDRs nos suele proporcionar la relación resistividad-iluminancia(Ω/lux), con lo cual obtener la iluminancia es relativamente sencillo midiendo la resistividad.

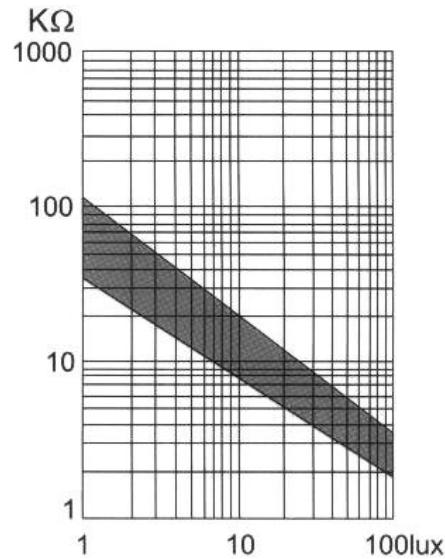


Fig. 1.2 Gráfica Lux-Resistencia de un LDR comercial(Modelo GL5528)[14]

1.2.2 Comunicación inalámbrica

Los MCUs suelen tener salidas de SPI, I2C, UART, algunos USB y algún otro protocolo. En el mercado existen módulos convertidores SPI-bluetooth/Zigbee, UART-bluetooth/zigbee, etc. Con lo cual, la comunicación de manera inalámbrica se puede realizar de una manera sencilla.

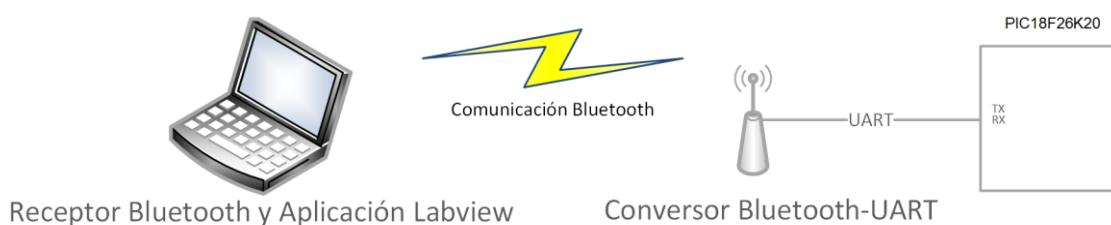


Fig. 1.3 Esquema de la comunicación MCU-Labview

En el ordenador, donde tenemos la aplicación de control, tendríamos el receptor correspondiente al protocolo de comunicaciones inalámbrico utilizado.

1.2.3 Energy Harvesting

Aprovechando que el sistema que se realizará será de bajo consumo, sería totalmente posible realizar un sistema de *Energy Harvesting* para alimentar de manera autosuficiente el sistema sensor.

En un invernadero se podría utilizar varios métodos de recuperación de energía pero la más óptima sería realizala con placas fotovoltaicas y almacenar la energía en una batería pequeña. Ver ejemplo de un sistema de *Energy Harvesting* en (Fig. 1.4).

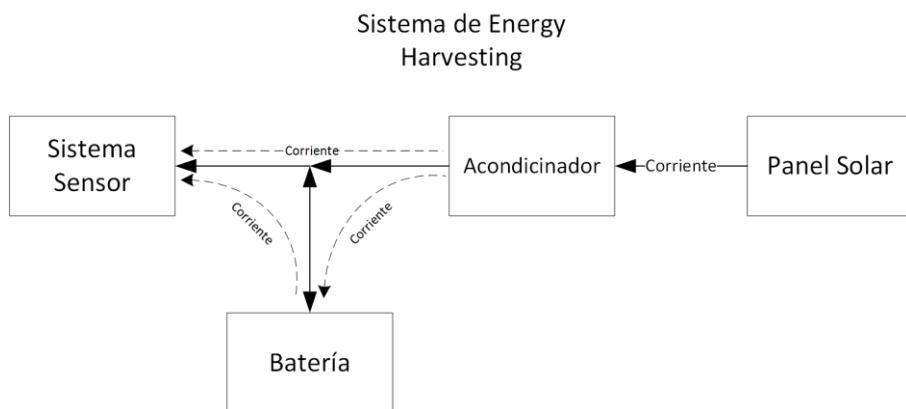


Fig. 1.4 Diagrama de bloques de un sistema de Energy Harvesting utilizando un panel solar

1.2.4 Actuadores a utilizar

Válvulas:

El control de riego se realizaría con una válvula controlada por el MCU utilizando el módulo PWM para poder regular el flujo de agua del circuito de riego una manera gradual.

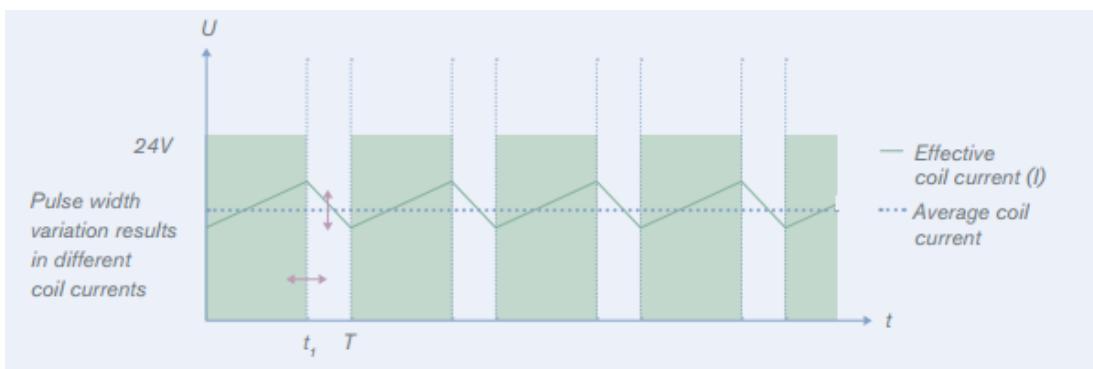


Fig. 1.5 Bürkert Solenoid Control Valves[15]



Fig. 1.6 Diferentes modelos de Burkert Solenoid Control Valves[15]

Motores:

El control lumínico consistiría en mover de posición algún objeto que tape el paso de los rayos de sol. Este movimiento tiene que ser controlador por algún motor. Los motores paso a paso son sencillos de configurar y son óptimos para realizar movimientos controlados.



Fig. 1.7 Motor paso a paso

1.3 Diagrama general del sistema

En la figura (**Fig. 1.8**) se muestra un esbozo de nuestro sistema instalado en un invernadero. El sistema estaría encargado de realizar el control de riego y lumínico de los cultivos. Además, el microcontrolador(MCU) enviará todos los datos del entorno previamente medidos por los sensores para hacer la monitorización con la aplicación de labview. Ver diagrama general del sistema(**Fig. 1.9**).

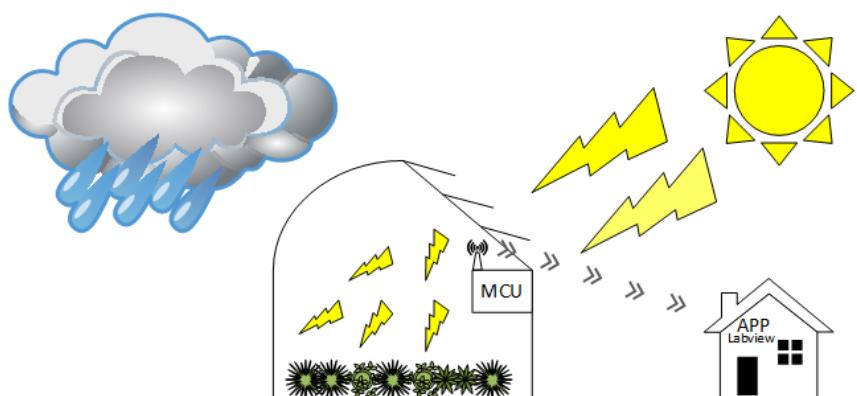


Fig. 1.8 Dibujo de nuestro sistema de control(Visio)

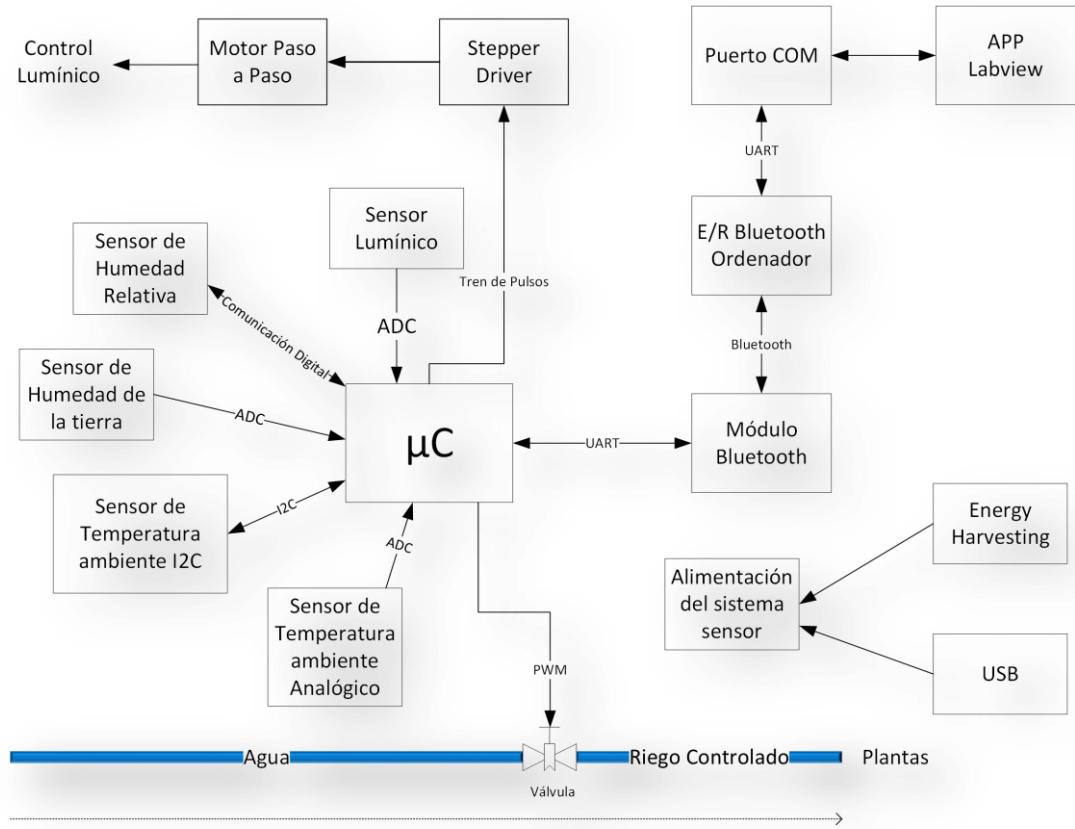


Fig. 1.9 Diagrama general del sistema

Capítulo 2. Planificación

El sistema sensor se implementará con el microcontrolador XLP PIC18F26K20.

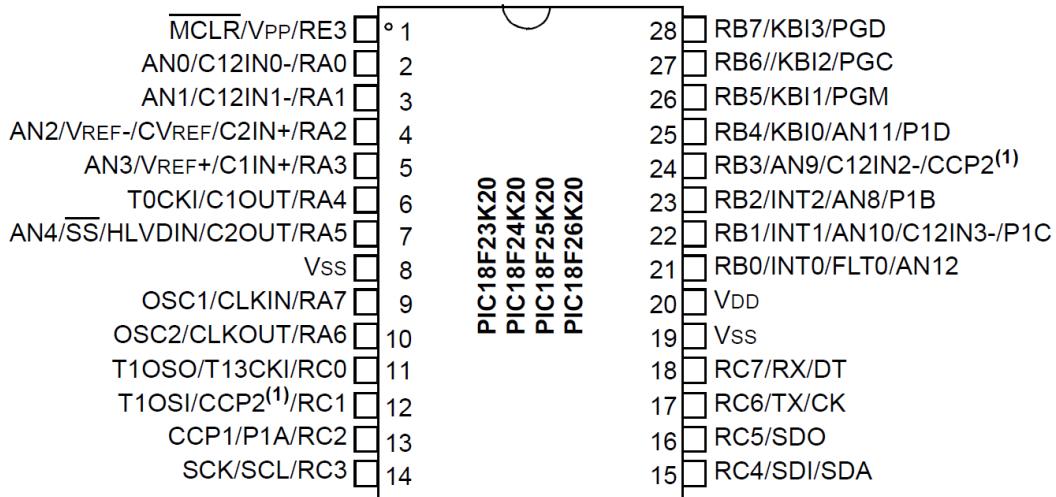


Fig. 2.1 Diagrama de Pines de nuestro PIC18F26K20 [6]

Características principales del microcontrolador son:

- Es un microcontrolador de 28 pines.
- Reloj interno de 16MHz/31KHz.
- Comunicación USART, SPI, I²C y PWM.
- 12 entradas al ADC-10bits.
- Alimentación 1.8V-3.6V.
- Sleep Mode: < 100 nA @ 1.8V y Watchdog Timer: < 800 nA @ 1.8V

Condiciones de diseño del sistema sensor:

- El sistema debe de estar en standby 1 minuto después de realizar la adquisición y transmisión de los datos obtenidos.
- Cuando se adquieran datos con el ADC se tendrá que tomar 4 muestras en un segundo y la media de las muestras se enviarán a la aplicación de control.
- La aplicación tiene que ser capaz de utilizar el protocolo I²C u otros protocolos digitales.
- El baudrate de la comunicación serie tiene que ser 9600.
- El programa en C debe seguir el método de programación FSM.

El programa principal tendrá una estructura idéntica a la de un programa Arduino: Setup() y Loop(). Usando el MPLAB-IDE, el programa principal se ejecuta en un *int main(void)* de modo que tendremos que emplear una estructura de programa como la siguiente (**Fig. 2.2**).

```

int main(void) {
    init_system();
    while (1) {

    }
}

```

Fig. 2.2 Programa principal

De esta manera, `init_system` sería `Setup()` y `while(1){}` el `Loop()`.

El programa en C debe seguir el método de programación FSM. Para realizarlo, dentro del loop tendremos 2 funciones ejecutándose: una se encargue de controlar el cambio de estado(`output_logic()`) y otra para ejecutar las tareas del estado(`state_logic()`).

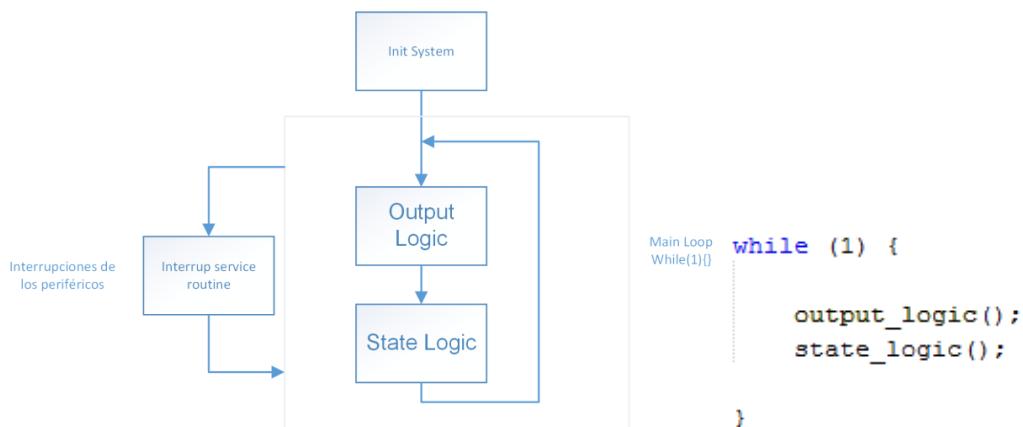


Fig. 2.3 Estructura del programa principal en C

2.1 Fase 1 del proyecto

En la primera fase del proyecto realizaremos un sistema que sea capaz de estar en un estado de reposo y que cuando le llegue una orden externa(en este caso el pulso de un botón) comience a capturar datos con el ADC. Una vez capturados los datos, éste los transmitirá y volverá al estado de reposo(Idle).

Nuestra Máquina de estado tendrá los siguientes estados(**Fig. 2.2**):

A. Idle

En el estado de Idle el programa esperará hasta que se pulse el botón.

B. Setup ADC/Timers/etc

En el *Setup* se pondrá en marcha con la configuración deseada el ADC y los Timers a utilizar.

C. Acquiring & Transmiting Data

En el estado de *A&T Data* se tomarán las muestras, se realizará un procesado a las muestras y finalmente se enviarán a nuestra aplicación de Labview.

D. Stop ADC/Timers/etc

En el estado de *Stop* pararemos el ADC y los Timers.

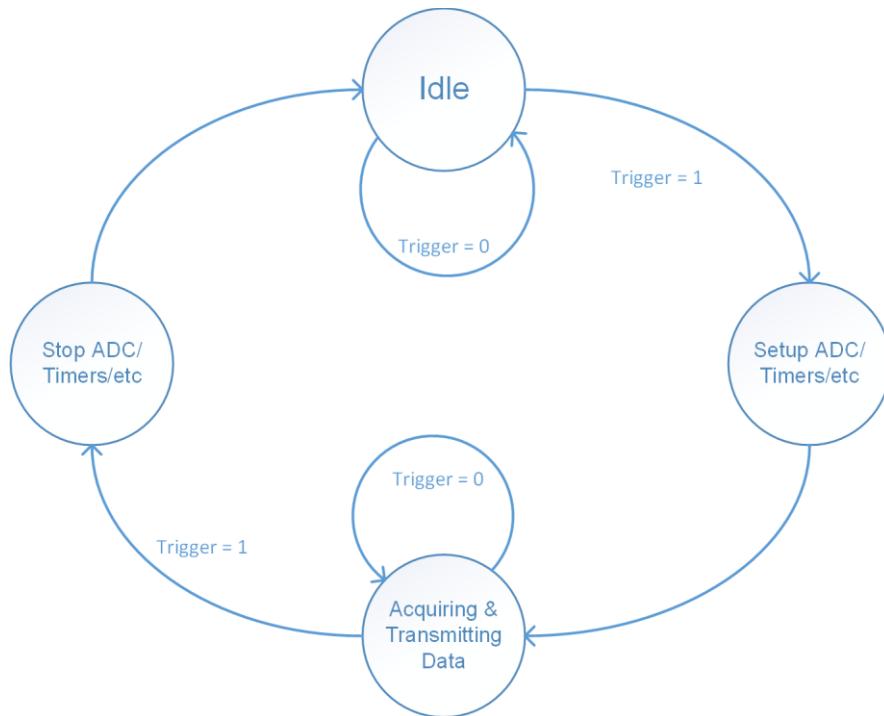


Fig. 2.4 Diagrama de estados de la fase 1

2.1.1 Programa en C

El programa en C que implementaremos seguirá la estructura de los siguientes diagramas de flujo **Fig. 2.5** y **Fig. 2.6**:

La función `output_logic()`, que se está ejecutando en loop, se encarga de realizar el cambio de estado. Utilizaremos la variable `Present_state` para guardar en qué estado estamos y con un `switch case` ejecutaremos la parte del código correspondiente al estado en el que estemos.

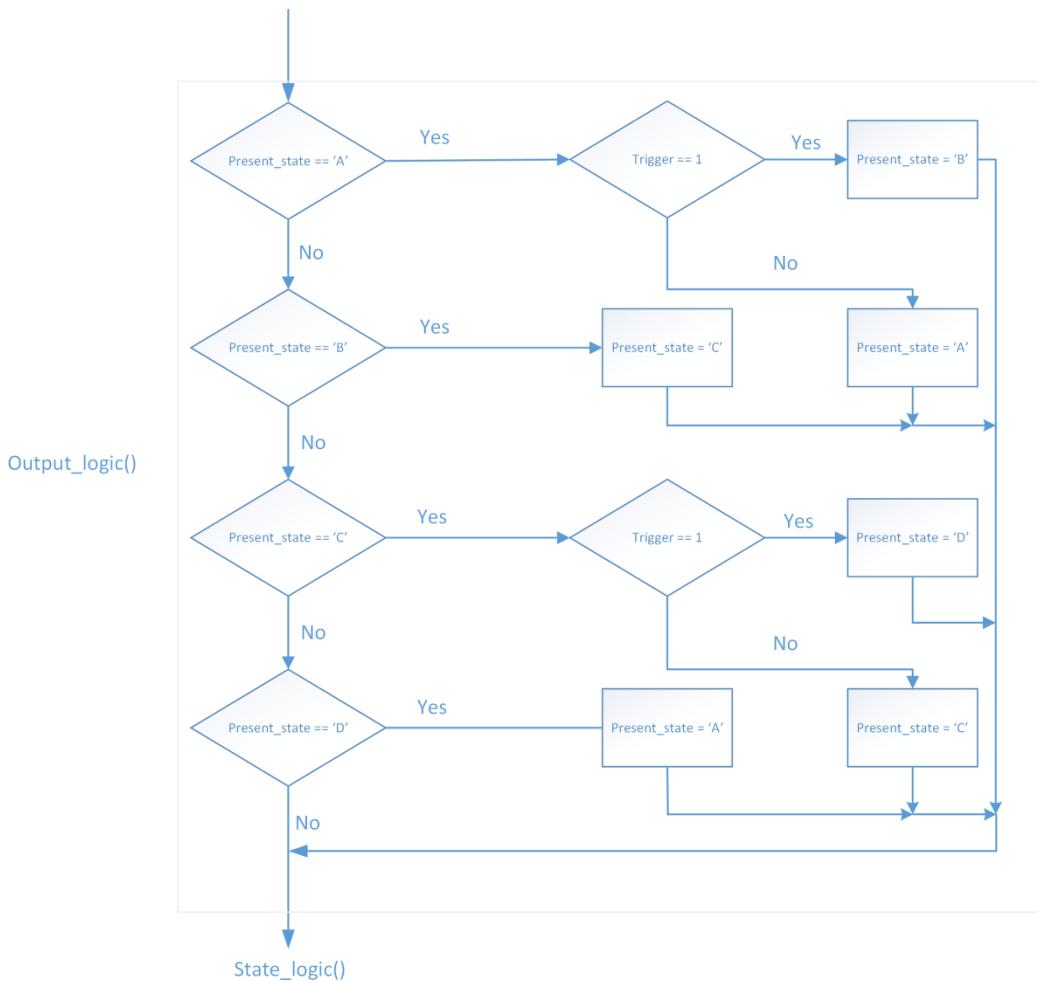


Fig. 2.5 Implementación en C de la función output_logic

En los estados *Idle*(A) y *A&T Data*(C) solamente habrá un cambio de estado si la variable *Trigger* es '1'. El *Trigger* se pondrá en '1' cuando pulsemos el botón en el estado *Idle* y también se pondrá en '1' cuando haya acabado de tomar 4 muestras en el estado *A&T Data*. En los estados de *Setup*(B) y *Stop*(D) solamente se ejecutará el código una vez y se pasará al siguiente estado.

En la función *State_logic()* ejecutamos las tareas correspondientes al estado actual en el cual nos encontramos utilizando un *switch case* también.

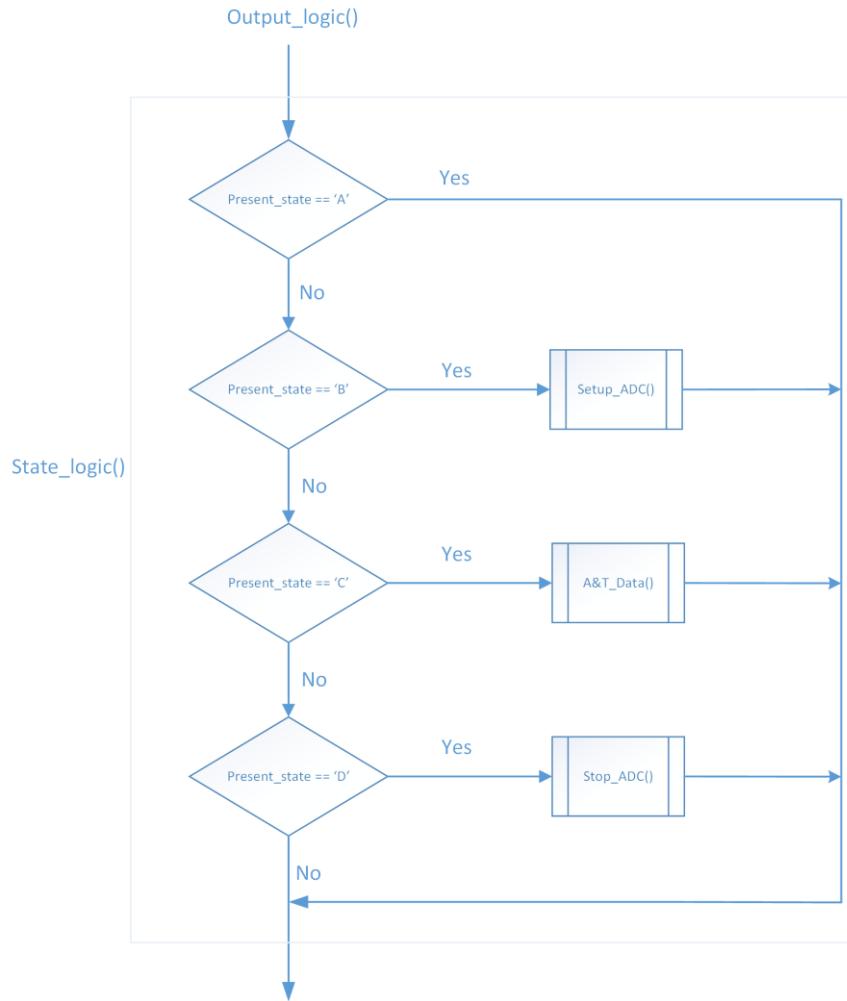


Fig. 2.6 Implementación en C de la función state_logic

En la **Fig. 2.6** podemos ver que tenemos los siguientes subprocessos:

- Setup_ADC

En el *Setup_ADC* tenemos el código en C correspondiente a la puesta en marcha el ADC y el TIMER3.

- A&T_Data

En el *A&T_Data* realizamos todas las tareas necesarias para hacer la captura de 4 muestras con el ADC y luego realizar la media de las mismas.

- Stop_ADC

En el *Stop_ADC* tenemos el código en C encargado de apagar los módulos ADC y el TIMER3.

En la función *IntServe()* se ejecutan las órdenes correspondientes cuando ocurre una interrupción. Para la fase 1 sólo utilizaremos la interrupciones INT0IF y TMR3IF. La interrupción INT0IF salta cuando por el puerto INT0

recibimos un flanco de bajada o de subida(depende de la configuración que hayamos escogido en el Init_System()), dicha interrupción la provocará el botón. Por último, la interrupción TMR3IF del TIMER3 se producirá cada 1ms(configurado en el estado de *Setup*) y cuando hayamos contado 249 interrupciones pondremos el flag *postscalersampleflag* en '1' para que se tome una muestra del ADC. Explicación en detalle del funcionamiento del TIMER3 y el ADC en los anexos **7.1.1 y 7.1.6**.

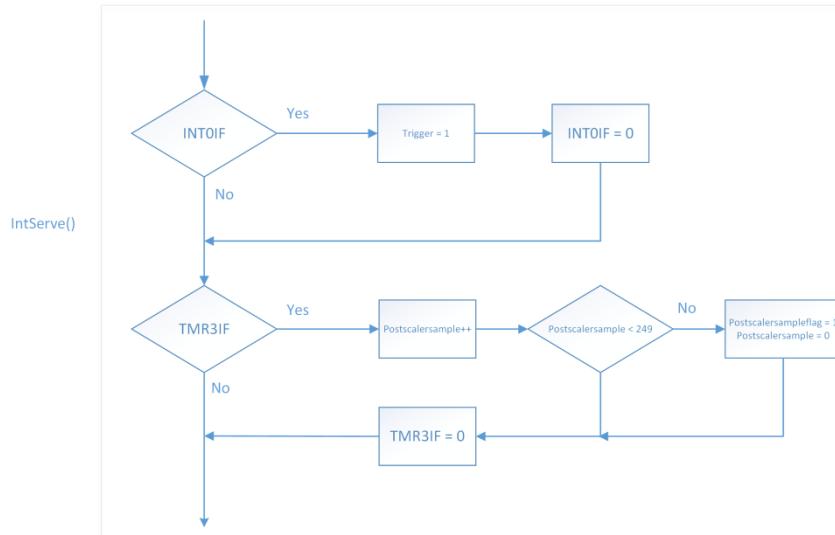


Fig. 2.7 Implementación en C de la función IntServe()

```

void __interrupt IntServe(void) {
    GIE = 0; //Disable interrupts while attending one of them ...
    if (INTOIF == 1) {
        Trigger = 1;
        INTOIF = 0;
    }
    if (TMR3IF == 1) {
        TMR3H = 0xF8;           // TMR3H must be written first
        TMR3L = 0x46;           // count up from F830 but there are a small overhead and T
        if ((postscalersample < postscalersamplemax) && (present_state!=STATE_Idle)) {
            postscalersample++;
        } else if (present_state!=STATE_Idle){
            postscalersample = 0;
            postscalersampleflag = 1;
        }
        TMR3IF = 0;
    }
}

```

Fig. 2.8 Código de la función IntServe()

2.2 Fase 2 del proyecto

En la Fase 2 del proyecto nos comunicaremos con 2 sensores digitales: el sensor DHT22 que tiene un protocolo particular que deberemos implementar en nuestro programa para poder leer los datos y un sensor que utiliza el protocolo I²C el SHT25. [13]

2.2.1 Comunicación con el sensor DHT22

El DHT22 es un sensor de temperatura y humedad relativa, que podemos simular en Proteus, con interface digital. La comunicación de realizará utilizando sólo un pin del MCU y se realizará de forma asíncrona porque no tenemos un clock como en la comunicación I²C.

El sensor utiliza un protocolo propio en el cual se nos explica en el datasheet [5]. Es un protocolo sencillo en el cual tendremos que utilizar el Timer2 para contar la duración de los pulsos porque según la duración del ellos, un pulso significará que se ha emitido un '0' o '1'. Explicación en detalle del funcionamiento del DHT22 en el anexo 7.2.1.

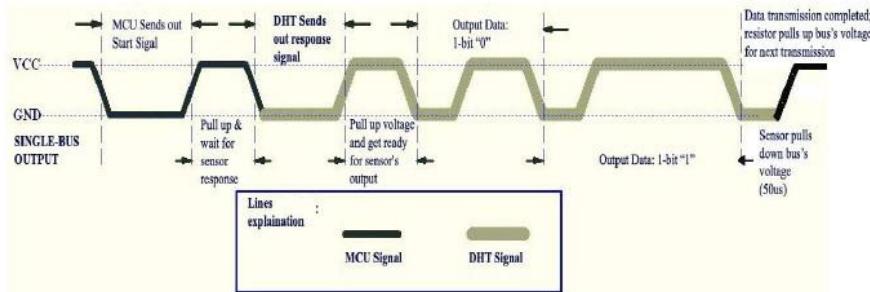


Fig. 2.9 Esquema de la comunicación con el sensor DHT22

En el transcurso de la transmisión, el sensor enviará un total de 5 Bytes: 2Bytes de Temperatura, 2Bytes de Humedad Relativa y 1Byte para el Checksum. Una vez recibamos los datos, no tendremos que utilizar una fórmula para descifrar que valor hemos recibido porque en el MSB tenemos su valores enteros y en su LSB sus decimales.

2.2.2 Comunicación I²C SHT25

La puesta en marcha de la comunicación I²C queda explicada en el anexo 7.1.3.

El sensor SHT25 es un sensor de temperatura y de humedad relativa, que podemos simular en Proteus.

El sensor tiene la siguiente dirección I²C: '1000000'. Se le puede solicitar que nos dé los datos de temperatura o humedad relativa. Nosotros solicitaremos los datos de humedad relativa con el comando '11110101' proporcionado en el datasheet del fabricante. [13]

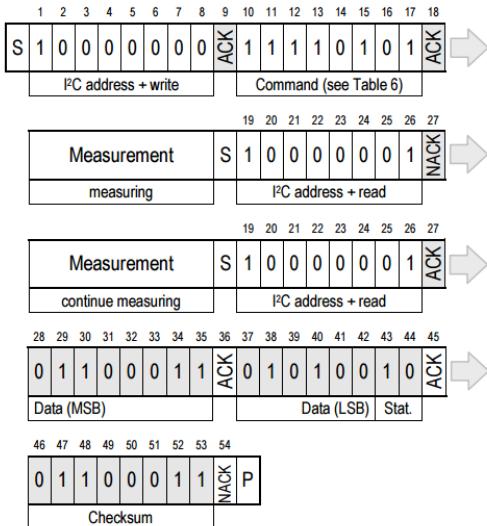


Fig. 2.10 Ejemplo de Comunicación I²C proporcionada por el datasheet

El datasheet del SHT25 nos proporciona la fórmula (2.1) que nos convierte el código recibido a humedad relativa.

$$RH = -6 + 125 \times \frac{S_{rh}}{2^{16}} \quad (2.1)$$

2.2.3 Modificación de nuestro programa en C fase 2

Como en la fase 1, empezaremos el proceso de adquisición de datos pulsando un botón. En este caso tendremos los siguientes estados:

A. Idle

En el estado de Idle el programa esperará hasta que se pulse el botón.

B. Setup DHT22

El MCU pondrá el puerto RB4 como puerto de salida y en '1' para preparar la secuencia de START. Además, se encenderá TIMER2 para que lea los bits que nos envíe el DHT22.

C. A&T Data DHT22

El MCU se comunicará con el sensor DHT22 utilizando el protocolo proporcionado por el fabricante(2.2.1). Una vez reciba los 5 bytes de datos, el MCU enviará los datos de temperatura ambiente y humedad relativa a la aplicación Labview. Finalmente, el MCU pondrá el flag trigger a '1' para pasar al estado siguiente.

D. Stop DHT22

El MCU apagará el TIMER2 porque no se utilizará en ningún estado de los siguientes

E. Setup I2C

El MCU encenderá la comunicación I²C configurando el clock a 100KHz, poniendo la MCU como Master y la comunicación I²C en Idle mode.

F. A&T Data I2C

El MCU se comunicará con el SHT25 utilizando el protocolo I²C[2.2.2]. Una vez recibidos los datos, el MCU utilizará la fórmula(apartado 2.2.2) para convertir los 2Bytes de datos en celcius. Por último, enviará por puerto serie los datos a la aplicación Labview y pondrá el flag trigger a '1' para pasar al estado siguiente.

G. Stop I2C

El MCU apagará la comunicación I²C.

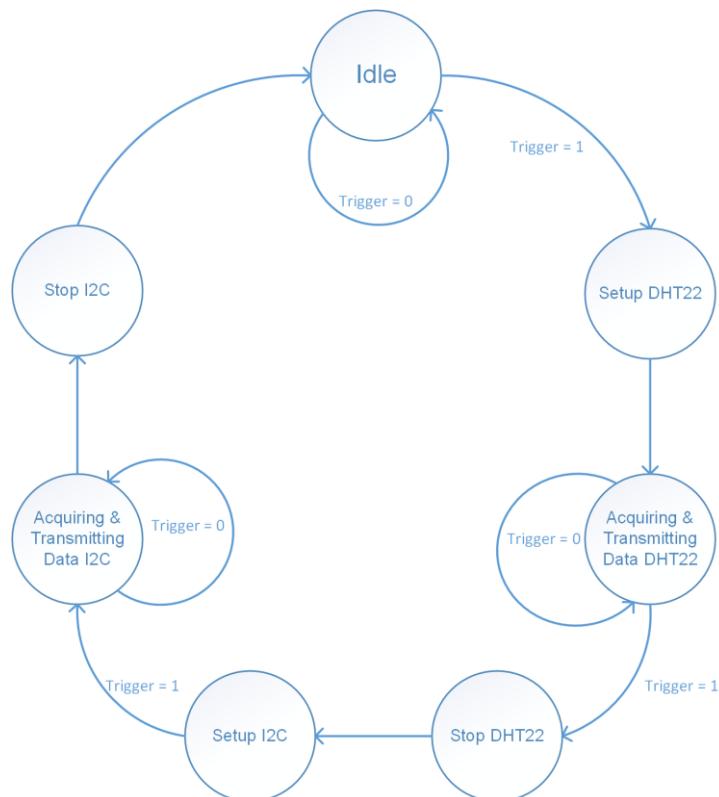


Fig. 2.11 Diagrama de estados de la fase 2

La estructura del programa en C en la fase 2 es la misma que hemos utilizado en la fase1 solamente con 3 estados más.

2.3 Fase 3 del proyecto

En la fase 3 añadiremos la opción de habilitar la salida PWM según el resultado de los datos obtenidos por los sensores. La salida PWM nos puede permitir controlar válvulas que controlen el flujo de agua(1.2.4). Explicación de la puesta en marcha del módulo PWM en el anexo **7.1.2**.

Por último, pondremos en marcha la comunicación UART para establecer comunicación con la aplicación de monitorización de Labview. Cabe añadir que en la fase 1 y 2 se explica que en los estados de *A&T Data* se envían los datos a la aplicación. Sin embargo la comunicación UART aún no estaba operativa en las dos primeras fases pero el código sí que estaba preparado para ello porque se sabía que en la fase 3 se iba a unir fase 1, 2 y lo nuevo de la fase 3. Explicación de la puesta en marcha de la comunicación UART en el anexo **7.1.4**.

2.3.1 Modificación de nuestro programa en C fase 3

El código necesario para poner en marcha la comunicación UART se realizará en el `Init_system()` utilizando las librerías proporcionadas por microchip.

Para poner en marcha el módulo la señal PWM crearemos un estado nuevo en nuestra máquina de estados. En él evaluaremos qué *duty cycle* aplicaremos o si lo ponemos en OFF según los datos obtenidos en los sensores.

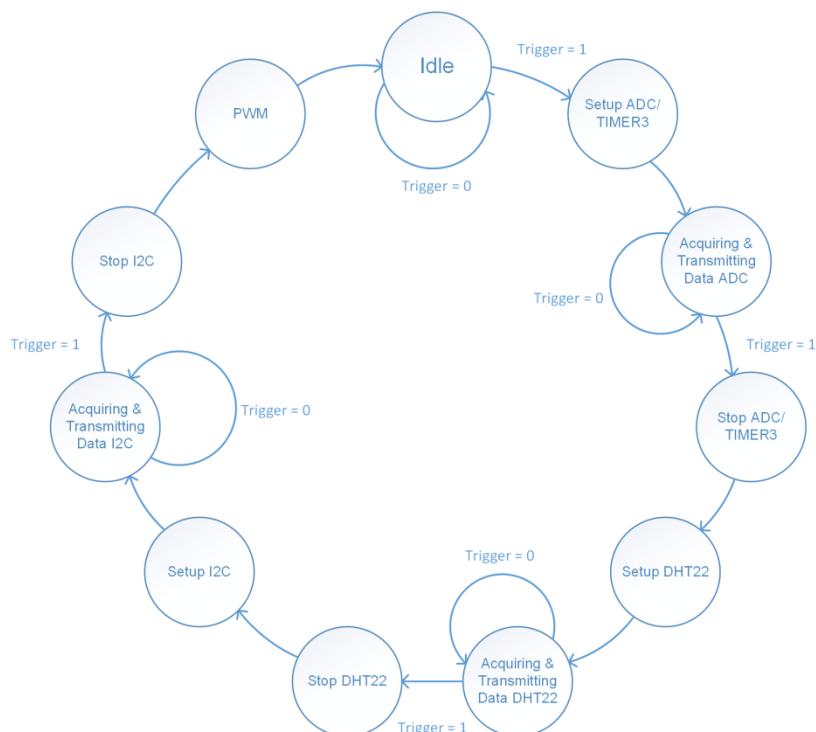


Fig. 2.12 Diagrama de estados de la fase 3

Capítulo 3. Desarrollo

Para realizar el desarrollo de las diferentes fases de planificación hemos utilizado el simulador Proteus. El simulador Proteus nos permite simular nuestro PIC18F26K20, algunos sensores digitales, la comunicación UART y otros elementos que nos han ayudado a probar el ADC y la comunicación I²C. Además, Proteus nos permite realizar debugging y de esta manera poder encontrar rápidamente soluciones a un bugs en el código.

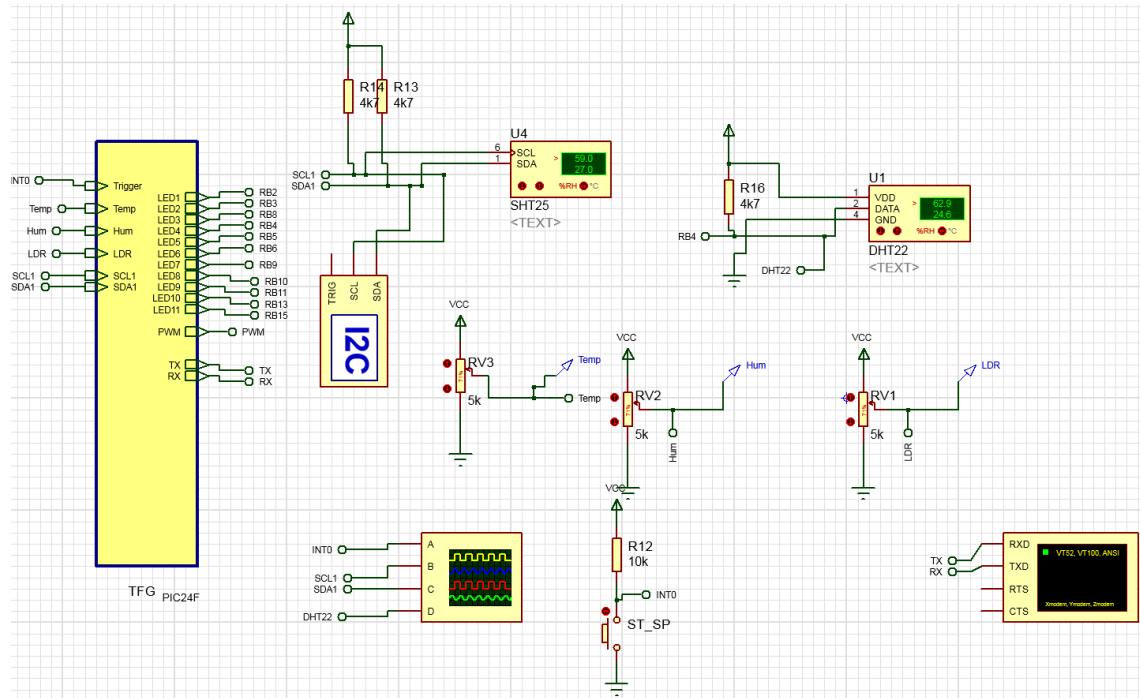


Fig. 3.1 Esquemático de nuestro proyecto Proteus

A continuación veremos las simulaciones correspondientes a las novedades añadidas en cada fase de desarrollo.

3.1 Desarrollo de la Fase 1

3.1.1 Interrupción INT0IF

La entrada INT0 de nuestro PIC18 la tenemos configurada para que la interrupción INT0IF se active cuando detecte un flanco de bajada. Para comprobar que funciona de la manera esperada también podemos utilizar el método el breakpoint pero esta vez hemos utilizado el osciloscopio y el Virtual Terminal.

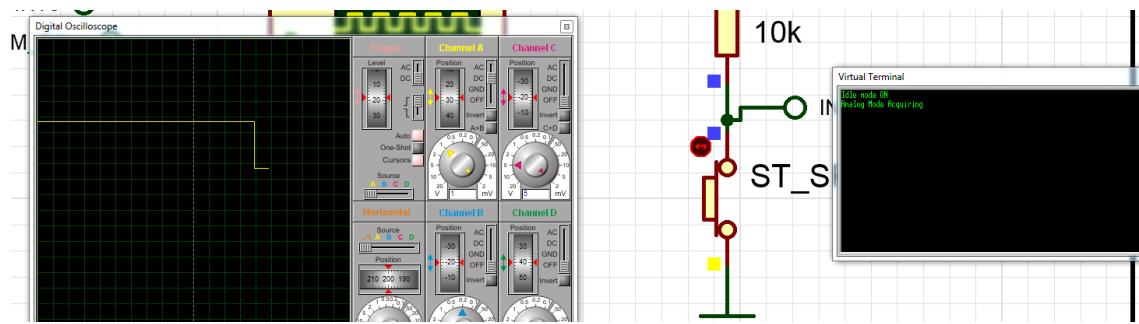


Fig. 3.2 Simulación de la interrupción INT0

Cómo podemos ver en la **Fig. 3.2**, cuando hay un flanco de bajada en INT0, el PIC envía un mensaje por comunicación UART dejando de estar en el estado Idle.

3.1.2 Muestreo del ADC con el Timer3

Hemos diseñado el sistema sensor para que tome 4 muestras en 1 segundo, por lo tanto cada vez el entre muestra y muestra deben de pasar 250ms. Utilizando el debugger de Proteus, hemos puesto un breakpoint en la función *GetSampleADC()* sabiendo que en el estado *Acquiring & Transmission ADC Analog Temp* la llamará 4 veces. Proteus nos proporciona el tiempo entre breakpoint, con lo cual tenemos que comprobar si ese intervalo es equivalente a lo previamente definido en nuestro diseño.

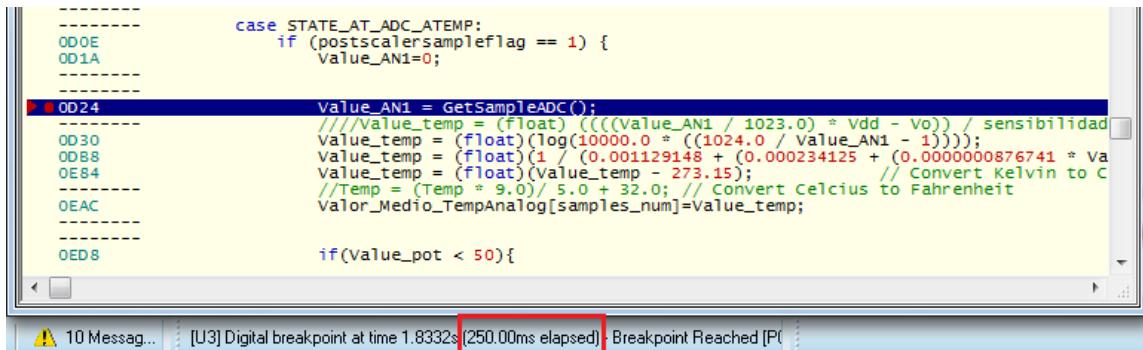


Fig. 3.3 Simulación de la interrupción TMR3IF para realizar el periodo de muestreo

3.2 Desarrollo de la Fase 2

3.2.1 Comunicación con el sensor DHT22

Con el osciloscopio hemos podido comprobar que el formato de trama en la transmisión de información es igual que en el datasheet del sensor.

En primer lugar tenemos una captura de la señal de inicio de comunicación en el cual el sensor envía una señal de pull down de 18ms y una de pull up hasta que responde el sensor.(Ver **Fig. 3.4**)

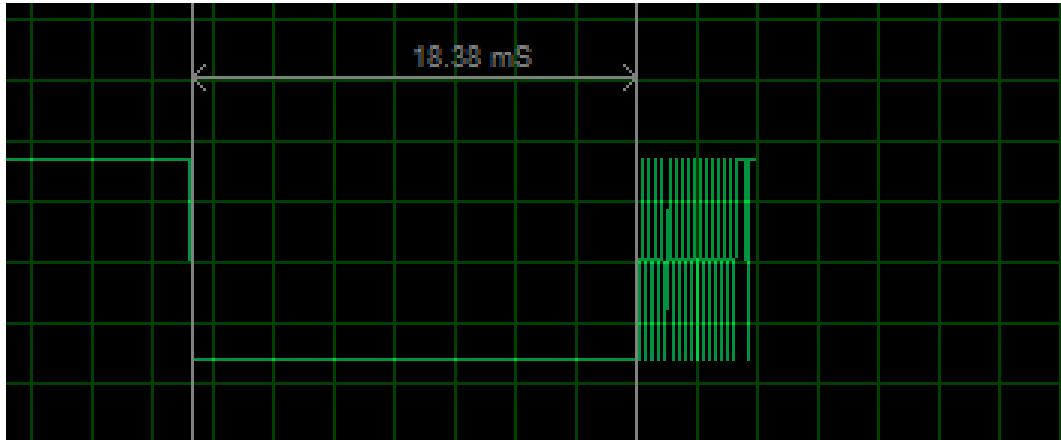


Fig. 3.4 Señal de Start del MCU

En segundo lugar comprobamos que los bits de datos se envían en pulsos cortos '0' y pulsos largos '1'.(Ver **Fig. 3.5**)

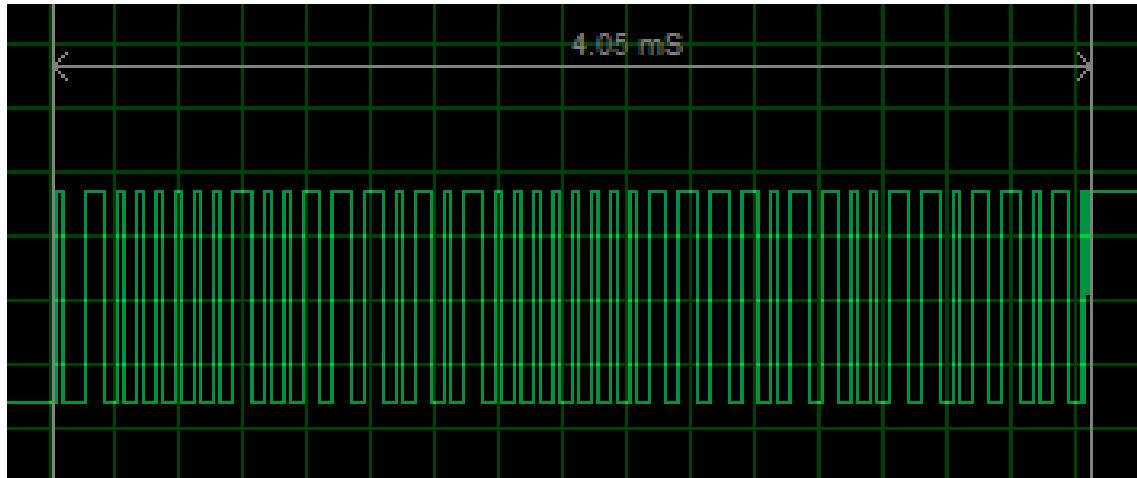


Fig. 3.5 Transmisión de datos del DHT22

3.2.2 Comunicación I²C

Proteus nos permite hacer debug de la comunicación I²C y esta manera podemos ver que el formato de trama es como explicamos[Apartado I2C, Anexo].

Configuramos el SHT25 con una humedad relativa del 59% y el debugger nos reporta lo siguiente:(ver **Fig. 3.6**)

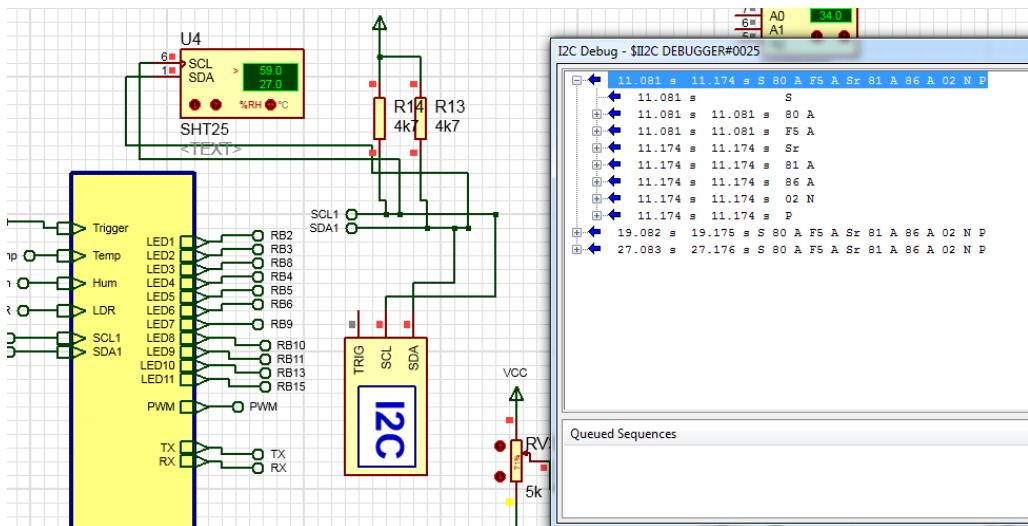


Fig. 3.6 Debug de la comunicación I²C

Report en Hexadecimal: [S 80 A F5 A Sr 81 A 86 A 02 N P]. En Negrita lo enviado por el MCU y en normal lo enviado por el sensor. El formato de trama del protocolo I²C nos indica que el MSB de la información es '86' y el LSB es '02'.

Utilizando la fórmula (2.1) S_{rh} es 8602 en hexadecimal y en decimal es 34306.

$$RH = -6 + 125 \times \frac{34306}{2^{16}} = 59.43 \quad (3.1)$$

Por lo tanto comprobamos que funciona bien la comunicación.

Otro aspecto a analizar de la comunicación I²C es el análisis de la trama en el osciloscopio. Primero mostraremos una captura de la primera parte de la comunicación y luego la segunda.

En la primera parte de la comunicación avisamos al sensor de que datos queremos que nos envíe y esperamos un tiempo para que el sensor tenga tiempo de procesar los datos(en nuestro diseño esperamos 90ms). [Anexo 7.1.3]

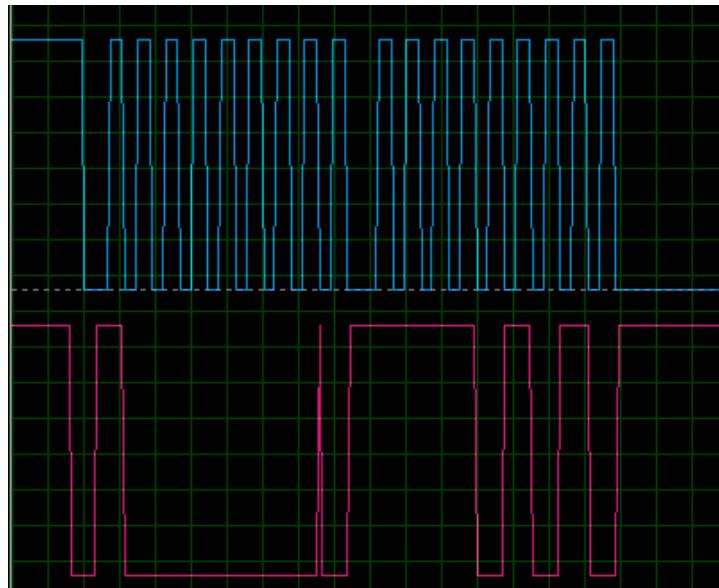


Fig. 3.7 Inicio de la trasmisión I²C: **S 80 A F5 A**; azul clock, rojo SDA

En la segunda parte de la comunicación avisamos al sensor de que estamos preparados para recibir los datos.

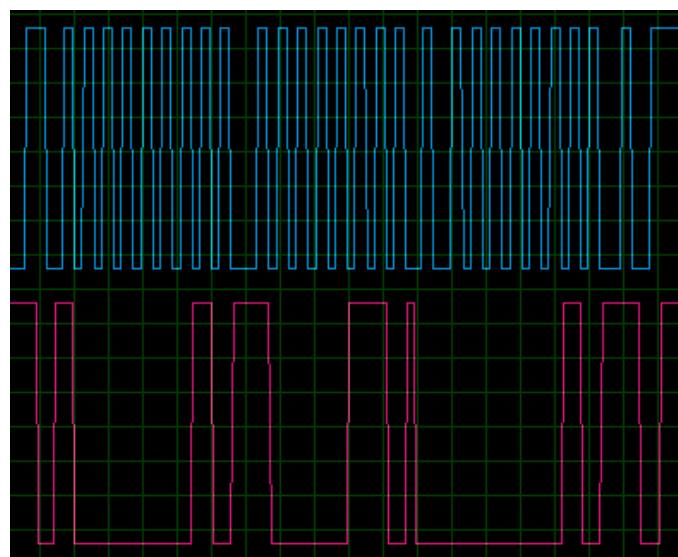


Fig. 3.8 Recepción de los datos: **Sr 81 A 86 A 02 N P**; azul clock , rojo SDA

3.3 Desarrollo de la Fase 3

3.3.1 PWM

Cuando llegamos al estado de *PWM* el MCU nos crea una señal PWM con un duty cycle del 20% tal y como lo hemos configurado en nuestro código en C.

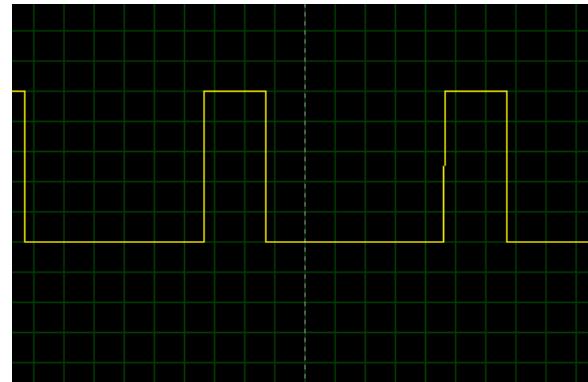


Fig. 3.9 PWM configurado al 20%

3.3.2 Comunicación serial

El Virtual Terminal de Proteus nos permite visualizar los mensajes que se transmiten y reciben por el puerto serie. (Ver **Fig. 3.10**)

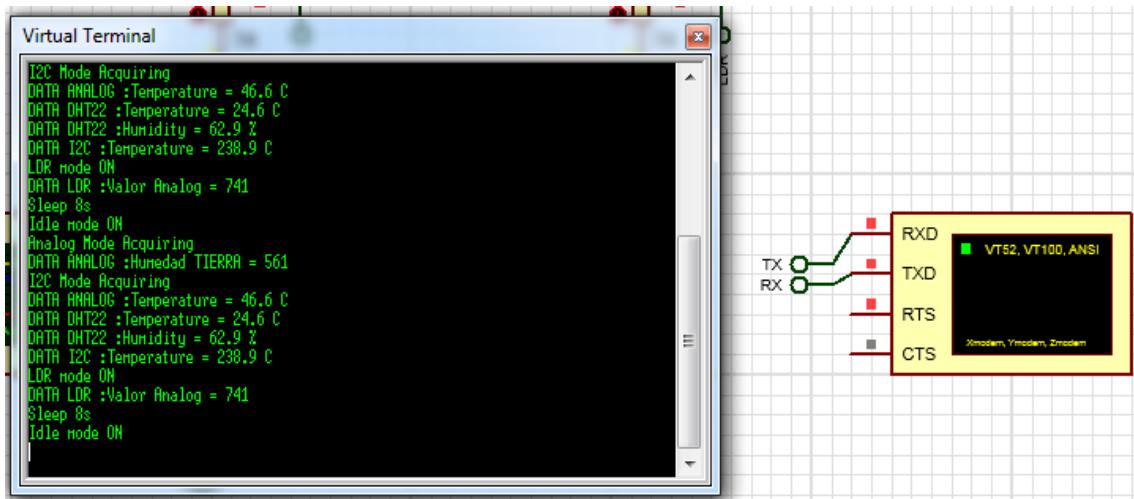


Fig. 3.10 Virtual terminal

Cuando la aplicación de Labview reciba los mensajes, la misma realizará un *parsing* de los mensajes que transmite el MCU para obtener los datos de los sensores.

Capítulo 4. Implementación del sistema

4.1 Elección de los sensores

Para tomar muestras de las diferentes señales que tenemos que medir utilizaremos tanto sensores con interface de salida analógica como digital, con el fin de estudiar diferentes tipos de sensores y protocolos de comunicación.

4.1.1 Temperatura Analog KY-013

El KY-013 es un termistor NTC, con lo cual su resistencia varía respecto la temperatura siguiendo un comportamiento exponencial. [foto sensor]

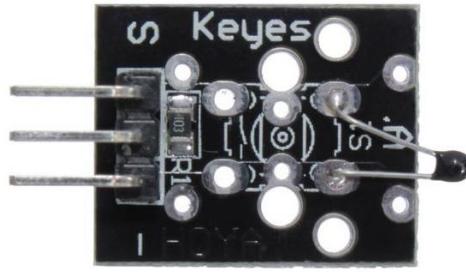


Fig. 4.1 Módulo de Temperatura ambiental KY-013

El fabricante nos proporciona la fórmula(4.2) que nos convierte el código leído por el ADC y Temperatura.[Anexo]

$$NTC = \frac{1024 \times 10K\Omega}{AN1-1} \quad (4.1)$$

$$T = \frac{1}{0.001129148 + (0.000234125 + (0.0000000876741 \times \ln NTC \times \ln NTC)) \times \ln NTC} - 273.15 \quad (4.2)$$

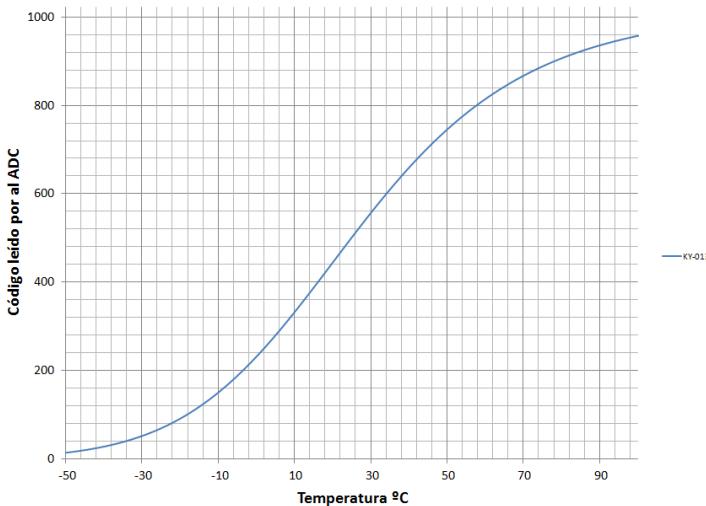


Fig. 4.2 Gráfica código ADC-temperatura KY-013

Además el fabricante nos explica cómo conectar el sensor a nuestro MCU. El KY-013 tiene 3 pines: 1 para ser conectado a VCC, 1 para conectar a GND y 1 para conectar la salida analógica(S) a una entrada del ADC(AN1).

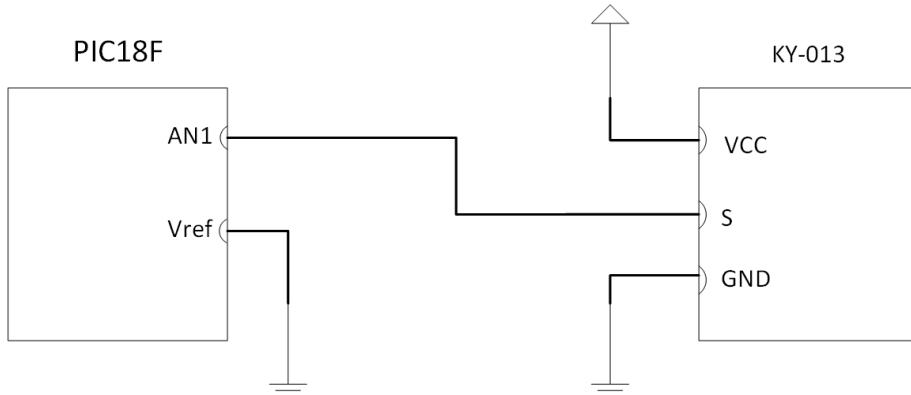


Fig. 4.3 Esquema de conexión del KY-013

4.1.2 Humedad Tierra Analog

El HTA es un sensor que detecta la conductancia de la tierra. El HTA tiene integrado un acondicionador de señal para que podamos medir la humedad de la tierra desde Vref hasta VCC con el ADC. Además el sensor tiene una salida digital, que no utilizaremos, que se pone en on/off cuando la tensión de salida del A0 es VCC/2.

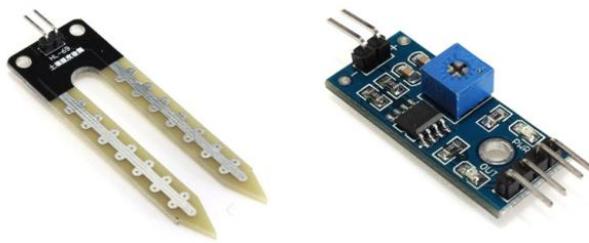


Fig. 4.4 A la izquierda el sensor de humedad y a la derecha el acondicionador de señal

En este caso, conectaremos la salida A0 a la entrada del ADC AN0 y el resto de conexiones tal y como se muestra en la figura 1.

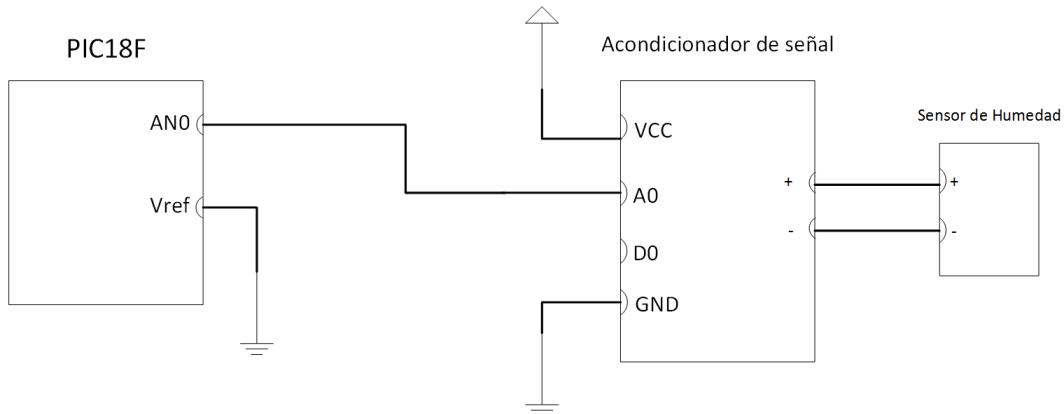


Fig. 4.5 Esquema que de conexión del módulo sensor de humedad de la tierra

4.1.3 Iluminancia Analog

El LDR es un sensor que detecta la Iluminancia que capta en su superficie fotosensible y hace variar su resistencia. El sistema sensor del LDR tiene integrado, igual que el HTA, un acondicionador de señal para que podamos medir la humedad de la tierra desde Vref hasta VCC con el ADC. Además el sensor tiene una salida digital, que no utilizaremos, que se pone en on/off cuando la tensión de salida del A0 es VCC/2.



Fig. 4.6 Módulo sensor de Luz

En este caso, conectaremos la salida A0 a la entrada del ADC AN11 y el resto de conexiones tal y como se muestra en la figura (Fig. 4.7).

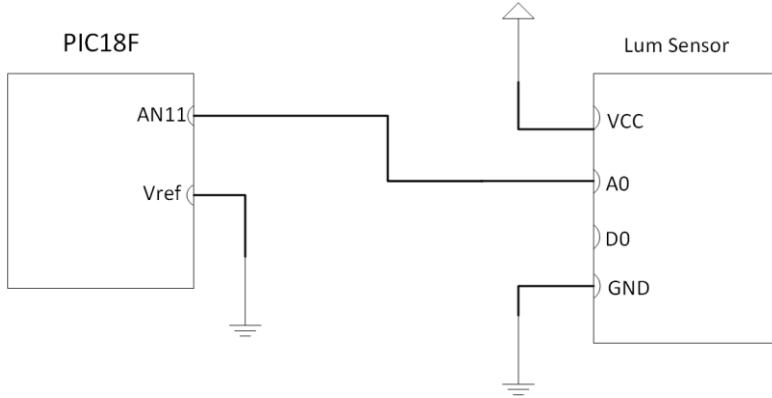


Fig. 4.7 Esquema que de conexión del sensor de luz

Para conocer la resistencia del LDR debemos tener en cuenta el esquema interno del módulo sensor(**Fig. 4.8**).

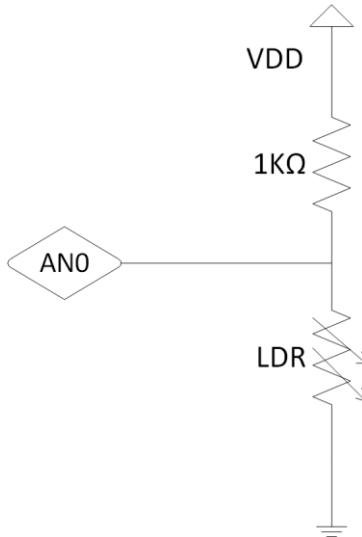


Fig. 4.8 Estructura del módulo sensor

Siguiendo la estructura del módulo sensor, la resistencia fija de $1\text{K}\Omega$ y el LDR forman un divisor de tensión(Fórmula 4.4). Podemos extraer el valor resistivo del LDR utilizando la siguiente expresión(Fórmula 4.5):

$$AN0 = \frac{LDR}{1K\Omega + LDR} \times VDD \quad (4.4)$$

$$LDR = \frac{AN0 \times 1K\Omega}{VDD - AN0} \quad (4.5)$$

El fabricante de nuestro módulo sensor no nos proporciona el modelo de LDR que tenemos. Consecuentemente hemos tenido que caracterizar nuestro LDR basándonos en la forma de caracterización de otros fabricantes.

Dos parámetros son necesarios para tener caracterizado nuestro LDR: La resistencia en la oscuridad y el parámetro α . Nuestro LDR en la oscuridad tiene una resistividad de $21\text{K}\Omega$ y el parámetro α debería de proporcionarlo el fabricante pero sabemos los valores típicos de este parámetro que oscilan entre 0.5 y 0.8. Para nuestro diseño hemos escogido 0.6 porque nos daba en la práctica valores de E_v más coherentes(**Fig. 4.9**).

La fórmula para convertir el valor resistivo de nuestro LDR a lux es la siguiente(Fórmula 3.6):

$$E_v = 10^{\frac{\log(LDR) - \log(21\text{K}\Omega)}{-\alpha}} \quad (3.6)$$

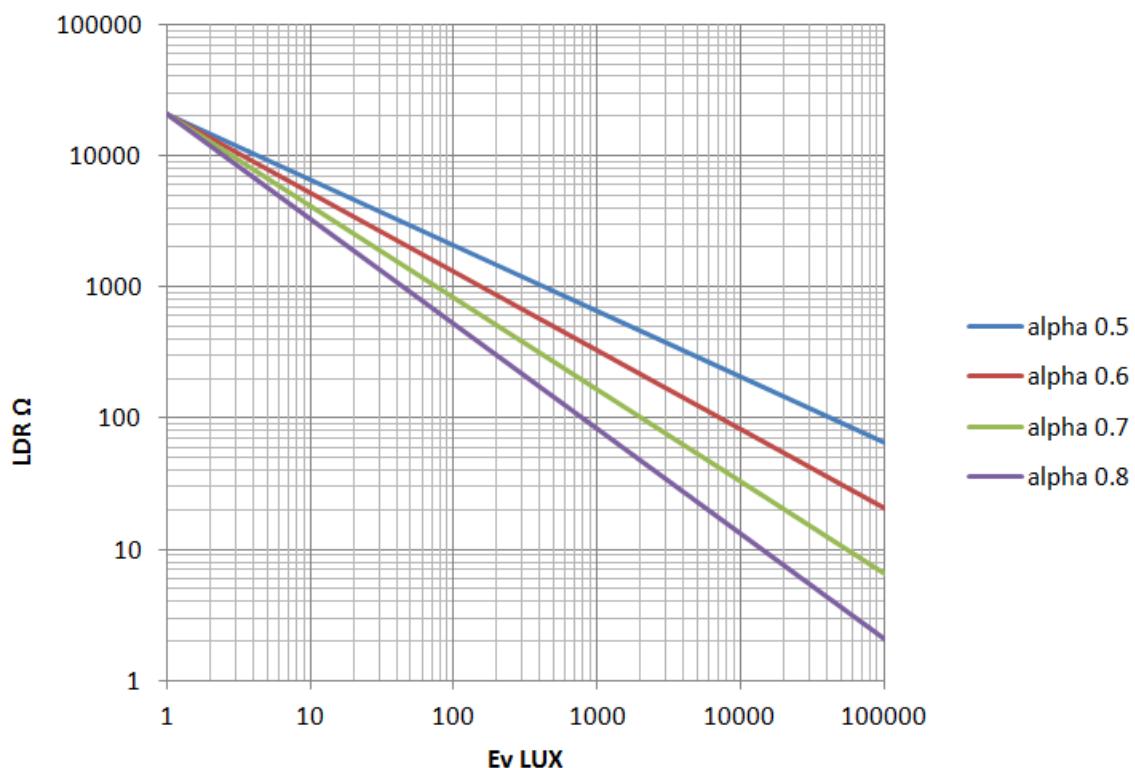


Fig. 4.9 Valores de lluminancia para diferentes valores α

4.1.4 Temperatura Digital I²C MCP9808

El MCP9808 es un sensor de temperatura con interface digital que utiliza el protocolo de comunicaciones I²C por lo tanto tendremos que configurar la MCU como Master porque el sensor será Slave.



Fig. 4.10 Sensor MCP9808

El MCP9808 tiene esta dirección I²C 0011XXX donde los últimos 3 bits de la dirección son configurables por vía externa. En el esquema de conexión, los puertos de A0, A1 y A2 están conectados a masa para que sean '0' y por lo tanto la dirección I²C sea 0011000.

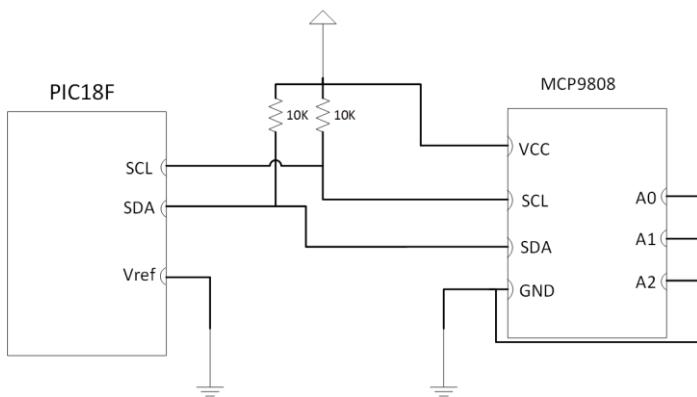


Fig. 4.11 Esquema que de conexión del sensor MCP9808

Durante la comunicación I²C[Anexo 7.1.3], deberemos de enviar el comando de 8bits '00000101' al sensor para que nos responda proporcionándonos la temperatura.

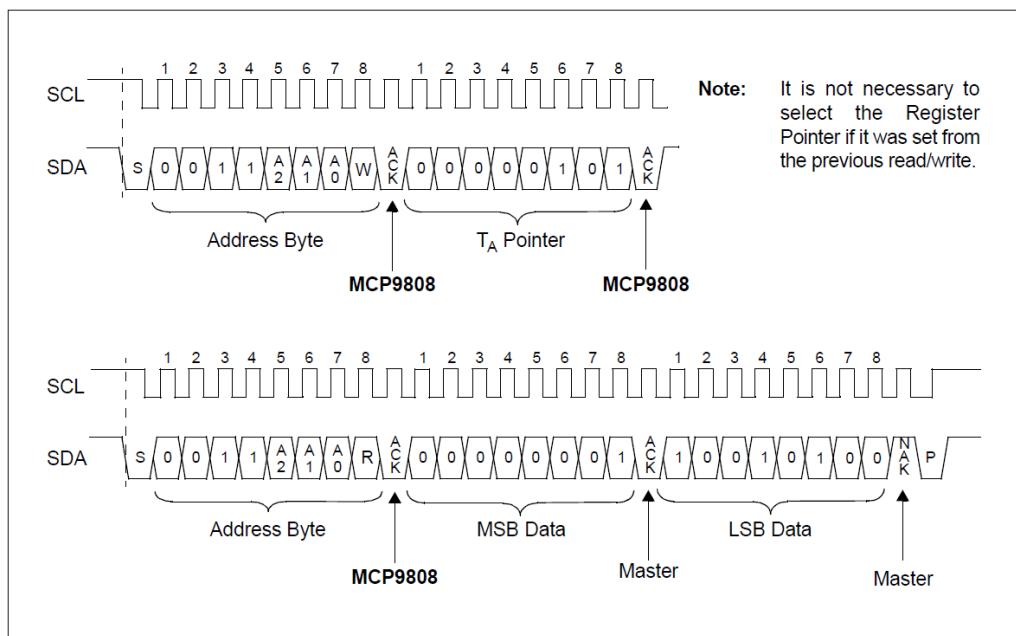


Fig. 4.12 Esquema de la comunicación I²C con MCP9808 [4]

Finalmente, el MCP9808 nos responderá con 2 bytes: un MSB y LSB. Entonces, gracias a la fórmula que nos proporciona el fabricante podremos traducir el código a ' $^{\circ}\text{C}$ '. Sin embargo en el datasheet del fabricante nos proporciona el código C que tenemos que utilizar para hacer la conversión(**Fig. 4.13**).

```
//Convert the temperature data
//First Check flag bits
if ((UpperByte & 0x80) == 0x80){      //TA > TCRIT
}
if ((UpperByte & 0x40) == 0x40){      //TA > TUPPER
}
if ((UpperByte & 0x20) == 0x20){      //TA < TLOWER
}
UpperByte = UpperByte & 0x1F;          //Clear flag bits
if ((UpperByte & 0x10) == 0x10){      //TA < 0°C
    UpperByte = UpperByte & 0x0F;    //Clear SIGN
    Temperature = 256 - (UpperByte * 16 + LowerByte / 16);
} else                                //TA > 0°C
    Temperature = (UpperByte * 16 + LowerByte / 16);
//Temperature = Ambient Temperature (°C)|
```

Fig. 4.13 Código C para convertirnos los bytes recibidos en Temperatura

4.1.5 Temperatura y Humedad Relativa Digital OneWire Protocol DHT22

En la fase 2 de planificación hemos estudiado el funcionamiento del DHT22, el cual utilizaremos para realizar nuestro sistema sensor definitivo.

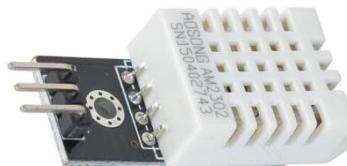


Fig. 4.14 Sensor DHT22

El datasheet nos muestra cómo debemos conectar el DHT22 con el MCU(Ver **Fig. 4.15**).

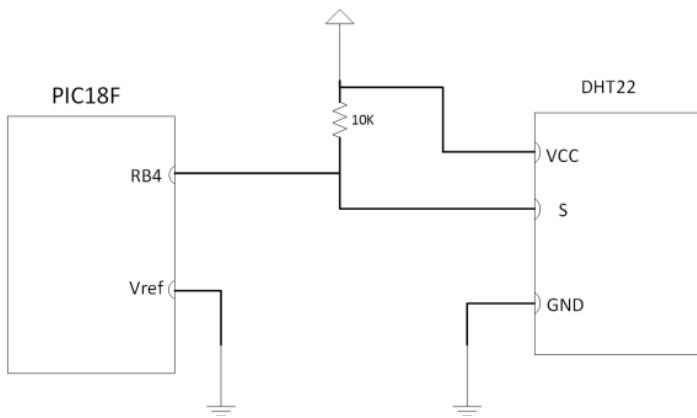


Fig. 4.15 Esquema que de conexión del sensor DHT22

4.2 Implementación de nuestra aplicación Labview

Desde nuestra aplicación de Labview podremos recibir los datos y transmitir información por un puerto COM.

4.2.1 Funcionalidades

La comunicación aplicación-MCU es bidireccional, desde el Panel de control podemos sacar del modo Sleep a nuestra MCU y que nos envíe los datos de forma instantánea.

Nuestra aplicación realiza un parsing de los mensajes que nos envía el sistema sensor para extraer los datos. Los datos de temperatura los mostramos en los termómetros y los demás datos en medidores en forma de porcentaje.



Fig. 4.16 Interfaz gráfica de nuestra aplicación Labview

Además, cada dato de los sensores pueden ser dibujados en las gráficas y también podemos exportar las gráficas a Excel.

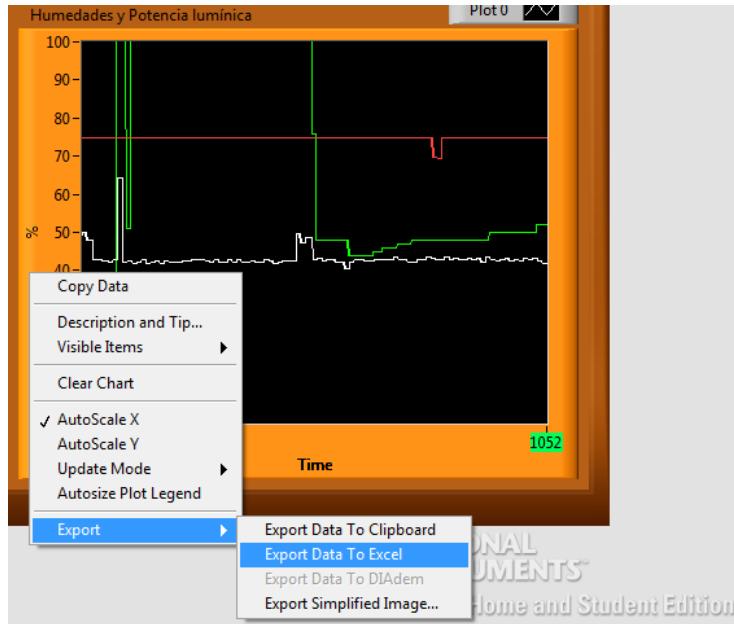


Fig. 4.17 Exportación de los datos en excel

4.2.2 Diseño de la aplicación

Labview es una plataforma de diseño de aplicaciones de ingeniería y ciencia pensada para acelerar la productividad de los ingenieros. Con una sintaxis de programación gráfica que facilita visualizar, crear y codificar sistemas de ingeniería.[12]

La creación de la aplicación consiste en ir uniendo módulos y al unirlos formar un diagrama de bloques.(Ver esquemático en **Fig. 4.18**)

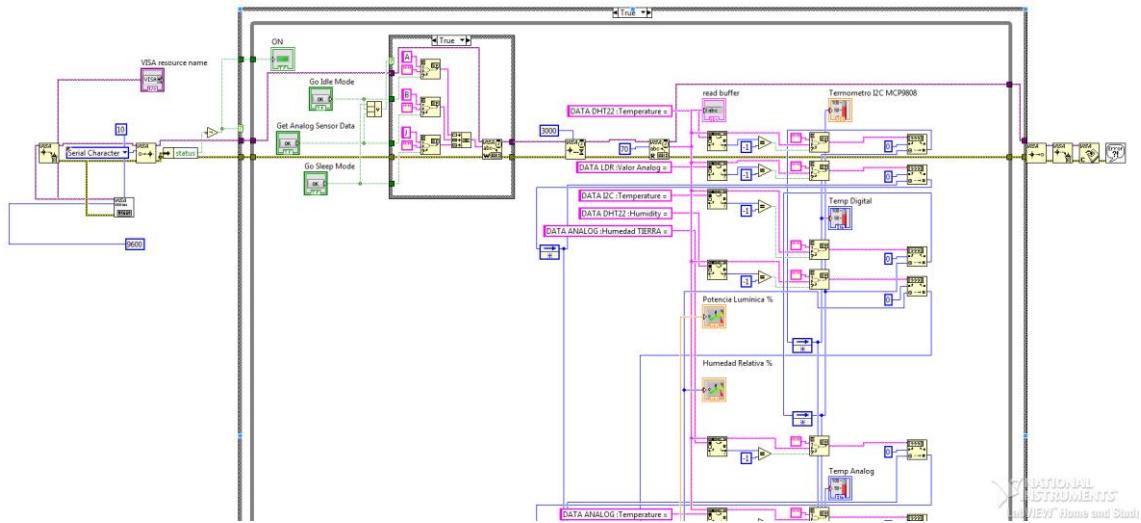


Fig. 4.18 Esquemático de nuestra aplicación en desarrollo

4.3 Implementación de sistema sensor en placa de pruebas

En los inicios del desarrollo del sistema sensor, utilizamos la placa de entrenamiento XLP 16-bits Development Board de Microchip. El mayor inconveniente de trabajar con esta placa era conectar nuestro PIC18 porque la placa está diseñada para un PIC24F16KA102.[9]



Fig. 4.19 XLP 16-bits Development Board de Microchip

La solución a este problema fue trabajar con una protoboard para poder conectar las I/O del PIC18 con la placa de manera correcta.

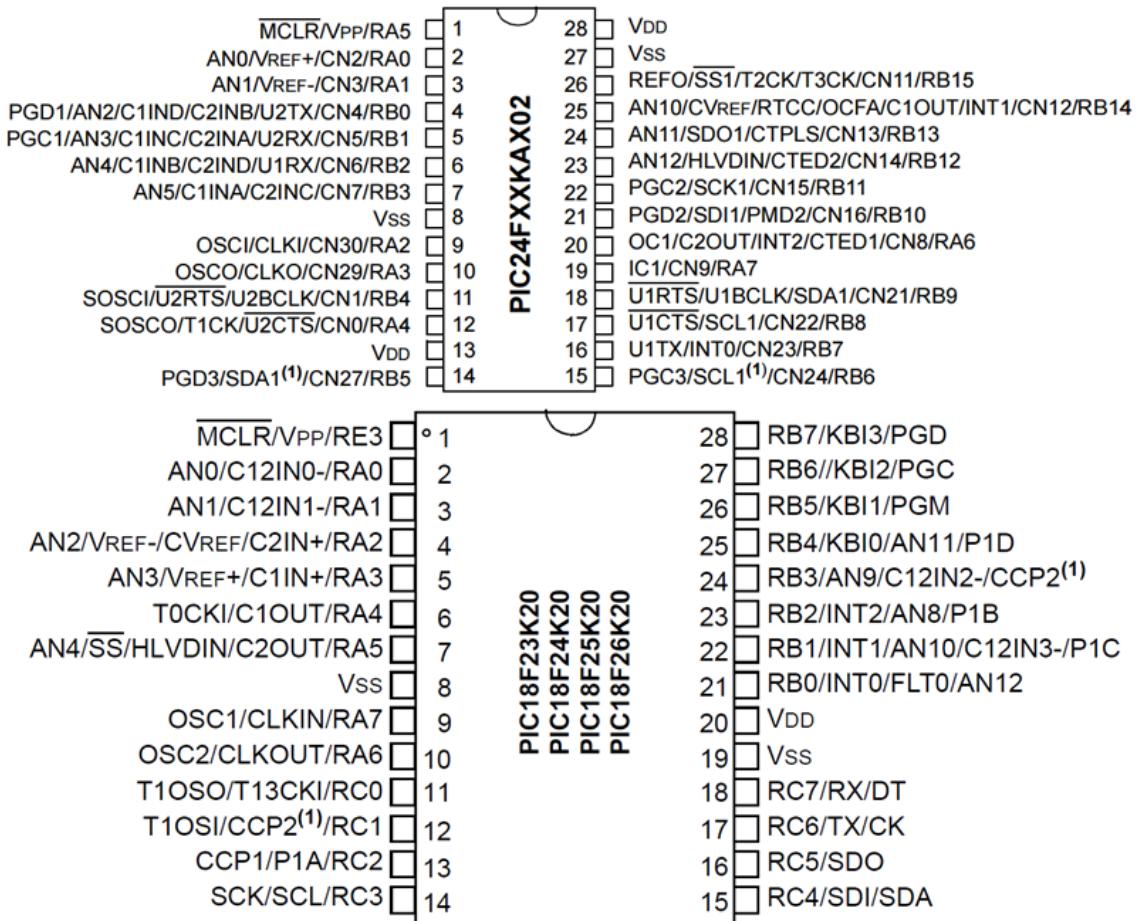


Fig. 4.20 Diagrama de pines: arriba PIC24F16KA102 y abajo nuestro PIC18F26K20

De este modo fue posible utilizar los dispositivos de la placa de entrenamiento, los más importantes: botones, conector pickit, potenciómetro y circuito de alimentación.

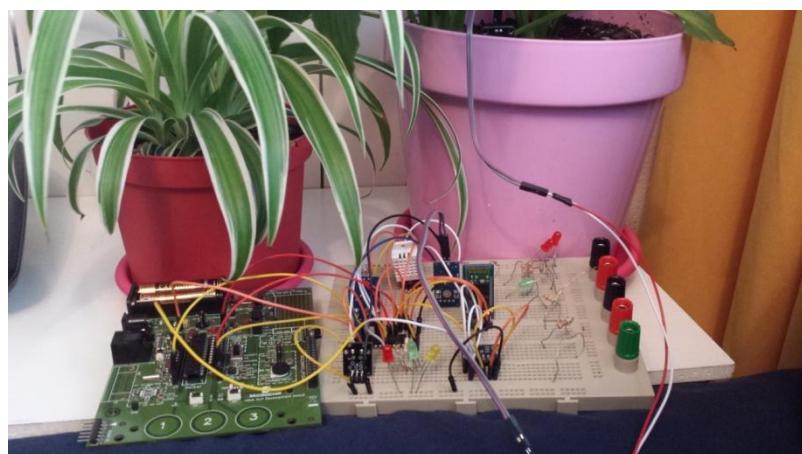


Fig. 4.21 Implementación del sistema sensor en placa de pruebas

4.3.1 Esquema de conexión de los sensores con el MCU

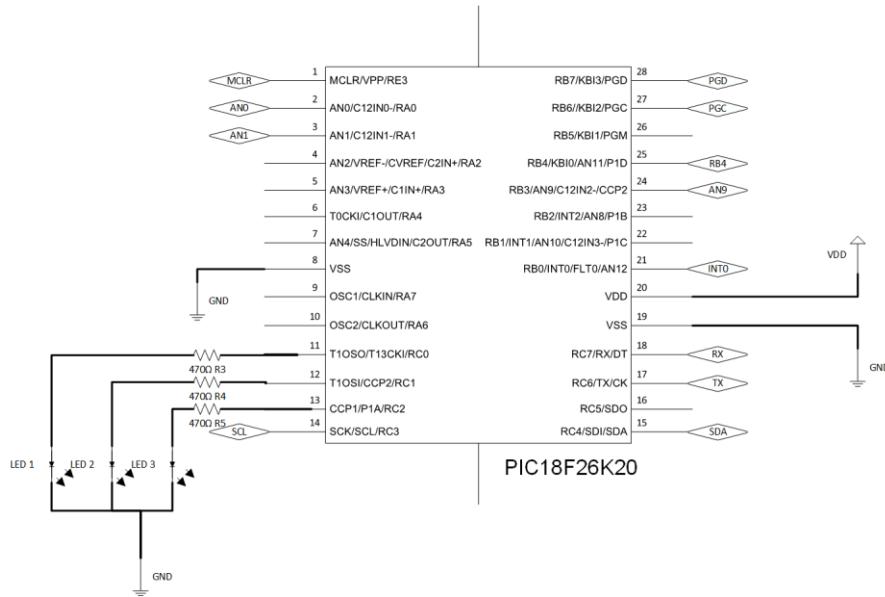


Fig. 4.22 Conexiones con nuestro MCU

Además de conectar los sensores al MCU, hemos conectado unos LEDs en los puertos de salida RC0, RC1, RC2 para tener una referencia del estado en el que se encontraba nuestra máquina de estados.

Las conexiones de los sensores se han realizado como explicamos en el apartado 4.1. Sin embargo, no hemos puesto las resistencias de pull up porque ya vienen integradas en la PCBs que integran los sensores.

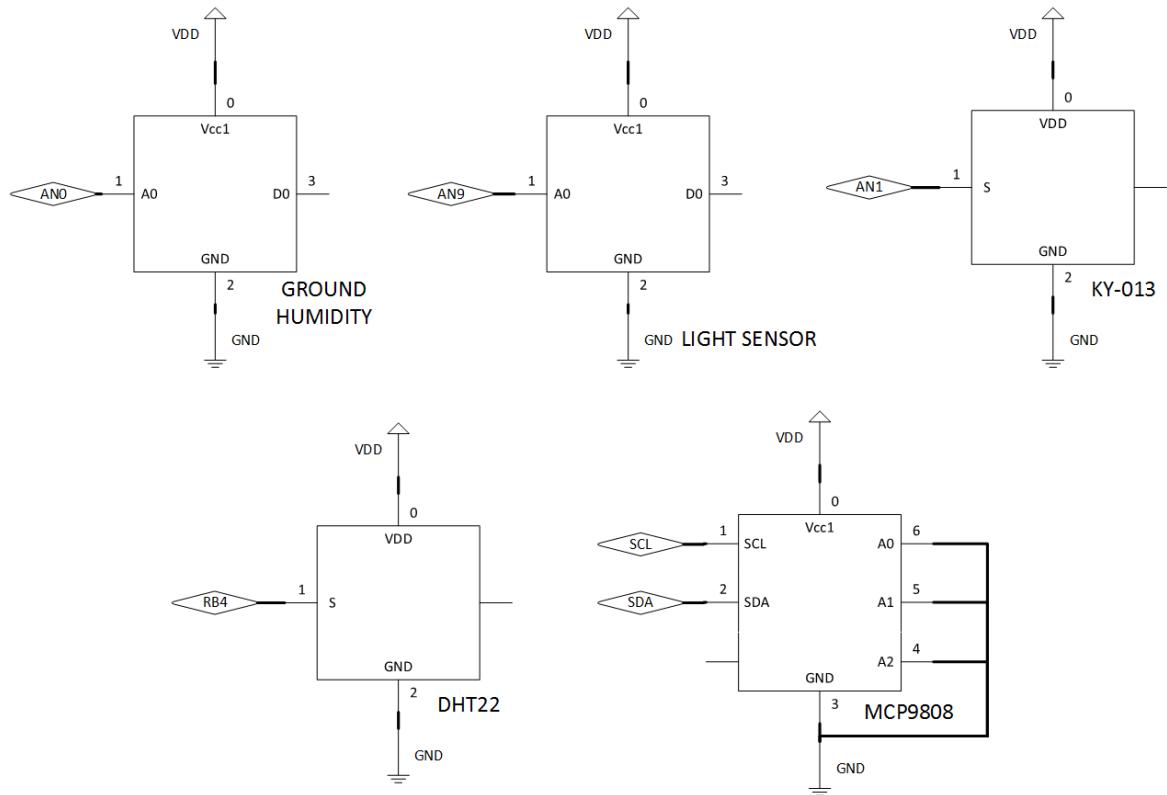


Fig. 4.23 Conexiones de nuestros sensores con nuestro MCU

4.3.2 Esquema de conexión PICKIT y UART-BLUETOOTH

Para programar nuestro PIC es necesario tener acondicionado los pines del PICKIT siguiendo el esquema que nos indica Microchip. Utilizando la placa de entrenamiento no tendremos problema porque ya viene integrada la conexión con PICKIT pero si tuviéramos que conectarlo sin la placa de entrenamiento deberíamos de realizar las conexiones siguiendo el siguiente esquema.[11]

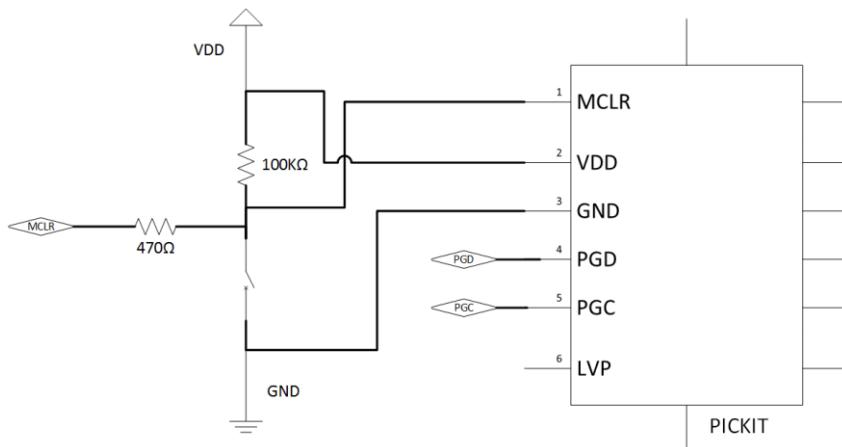


Fig. 4.24 Conexión con Pickit

Cuando conectemos nuestro transmisor-receptor bluetooth a nuestro MCU, deberemos conectar el puerto RX_{mcu} al TX_{bluetooth} y el TX_{mcu} al RX_{bluetooth}[Ver].

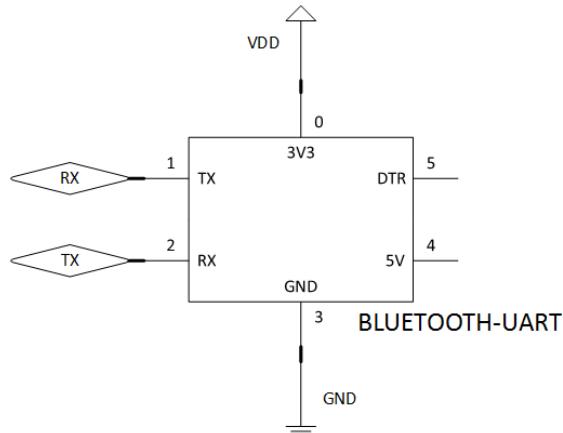


Fig. 4.25 Conexiones de nuestro módulo Bluetooth con nuestro MCU

4.3.3 Esquema de alimentación de la Development Board

Hay 4 maneras de alimentar nuestro sistema sensor con nuestra placa de entrenamiento, una utilizar el esquema de conexión del PICKIT y hacer que alimente el PICKIT todo nuestro sistema.

La segunda opción es utilizar la entrada USBmini y con el siguiente sistema regulador convertir los 5V del USB a 3.3V(VDD de nuestro prototipo).[7]

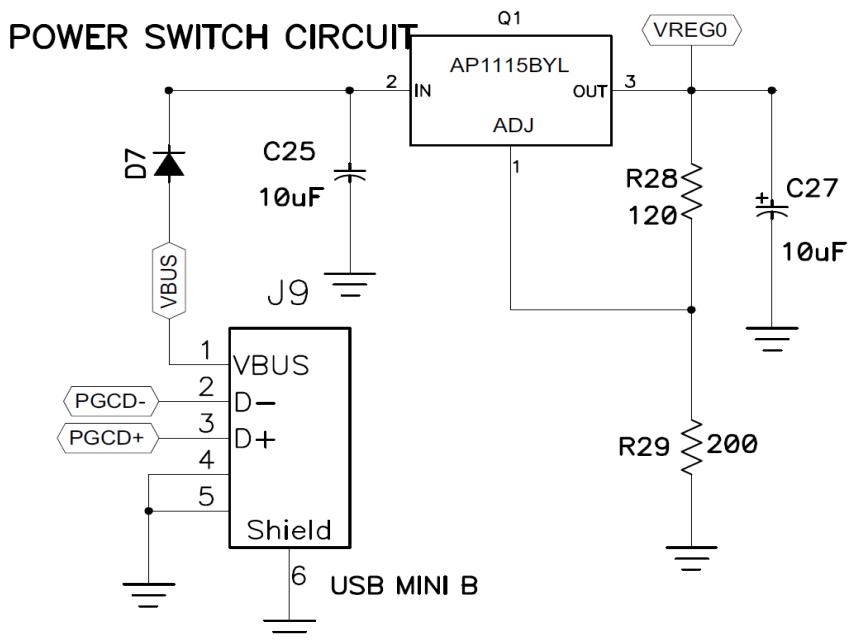


Fig. 4.26 Circuito de alimentación por mini USB

La tercera opción es utilizar las pilas de mercurio para realizar una alimentación de 2,9-3V. Este modo de conexión no es el más óptimo porque el voltaje de alimentación es inferior al voltaje de funcionamiento de alguno de nuestros sensores. Aún teniendo este inconveniente el sistema sensor excepto el sensor DHT22 que no funciona.

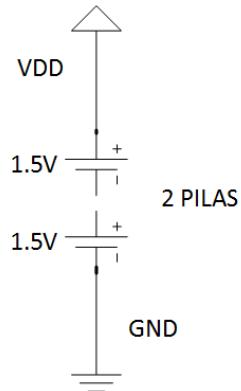


Fig. 4.27 Circuito de alimentación con 2 pilas de 1.5V

La cuarta opción consiste en utilizar el Cymbet's EVAL-08 Solar Energy Harvester que incluye nuestra development board. El mismo está diseñado para ir cargando poco a poco una batería y que el sistema consuma nA, sin embargo nuestro sistema necesita 3.3V de alimentación constante por lo tanto no se puede utilizar para alimentar el sistema sensor.[8]

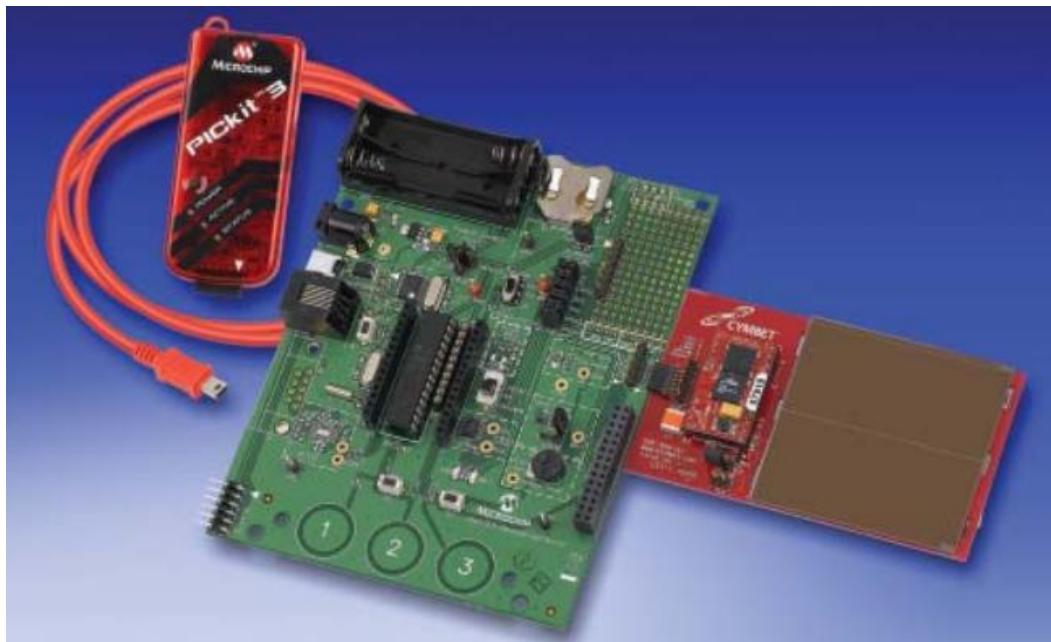


Fig. 4.28 Alimentación con Cymbet's EVAL-08 Solar Energy Harvester

4.4 Implementación del sistema sensor en PCB con Eagle

EAGLE, (siglas de Easily Applicable Graphical Layout Editor) es un programa de diseño de diagramas y PCBs con autoenrutador.

1. Lo primero que deberemos realizar para el diseño de nuestra propia PCB, es dibujar el esquema eléctrico de lo que va a ser nuestra placa para luego generar nuestra Board.
2. Una vez generada la board, deberemos de colocar los elementos del circuito en nuestra placa e ir conectando los elementos.
3. Una vez acabado del diseño de la placa, generaremos los archivos gerber para realizar el mecanizado de nuestra placa.

4.4.1 Schematic

Lo primero que hay que hacer antes de empezar, es saber qué package tienen nuestros elementos del circuito porque el package delimita qué agujeros y/o forma de la pista para luego el dispositivo encaje al soldarlo. Dicho esto, en las librerías de Eagle no tienen todos los dispositivos pero casi todos los tipos de package sí suelen estar. Entonces, cuando no encorramos el dispositivo podemos hacer 2 cosas: crear nuestro propio dispositivo utilizando un package de la librería o coger un dispositivo existente con el mismo package.

En nuestro diseño hemos cogido elementos existentes con el mismo package para hacer el esquemático porque lo importante es el diseño de la board.

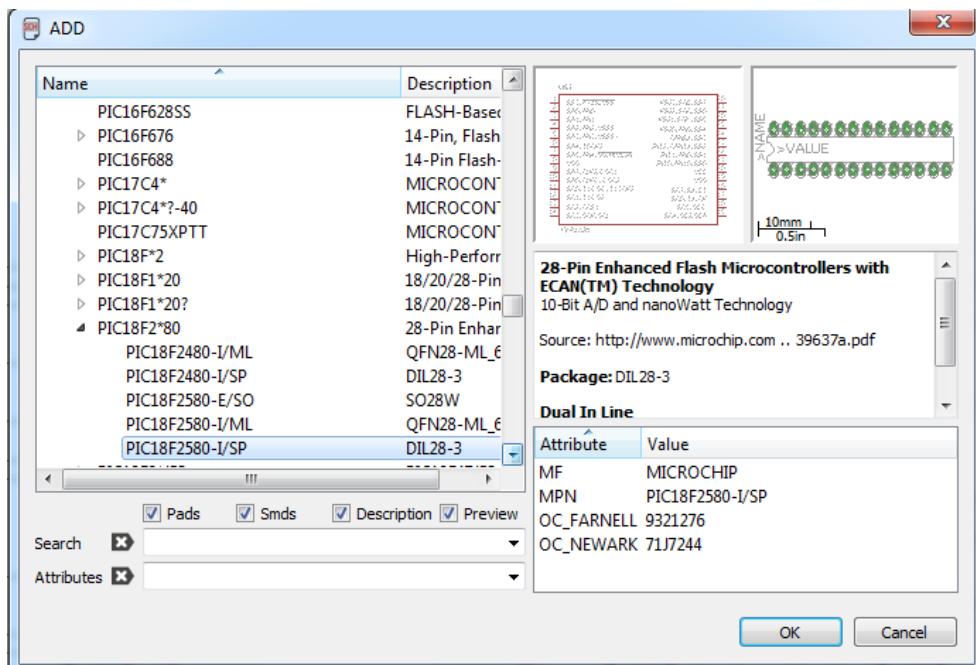


Fig. 4.29 Elección de componentes en Eagle

El esquema es casi el mismo que en la placa de pruebas, la única diferencia la tenemos en la alimentación[Anexos], y, pensando en la board, hemos cambiado la mayoría de los package a SMD categoría 1206.[Detalles en anexos] Las conexiones con los sensores y bluetooth se han hecho con puertos jumpers hembra.

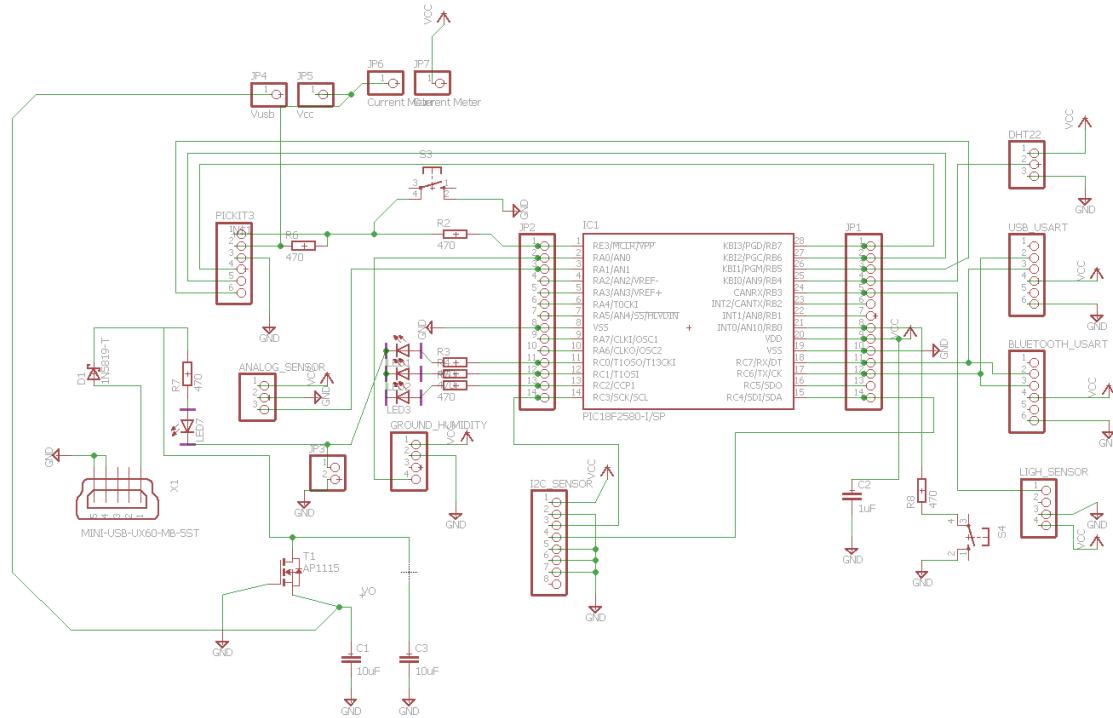


Fig. 4.30 Esquemático de nuestro proyecto en Eagle

4.4.2 Board

Una vez realizado el esquematizado, tenemos que generar nuestra board. Con la board generada, tenemos que ir colocando los elementos pensado cómo las pistas tendrán que conectar entre pin y pin (Ver **Fig. 4.31**).

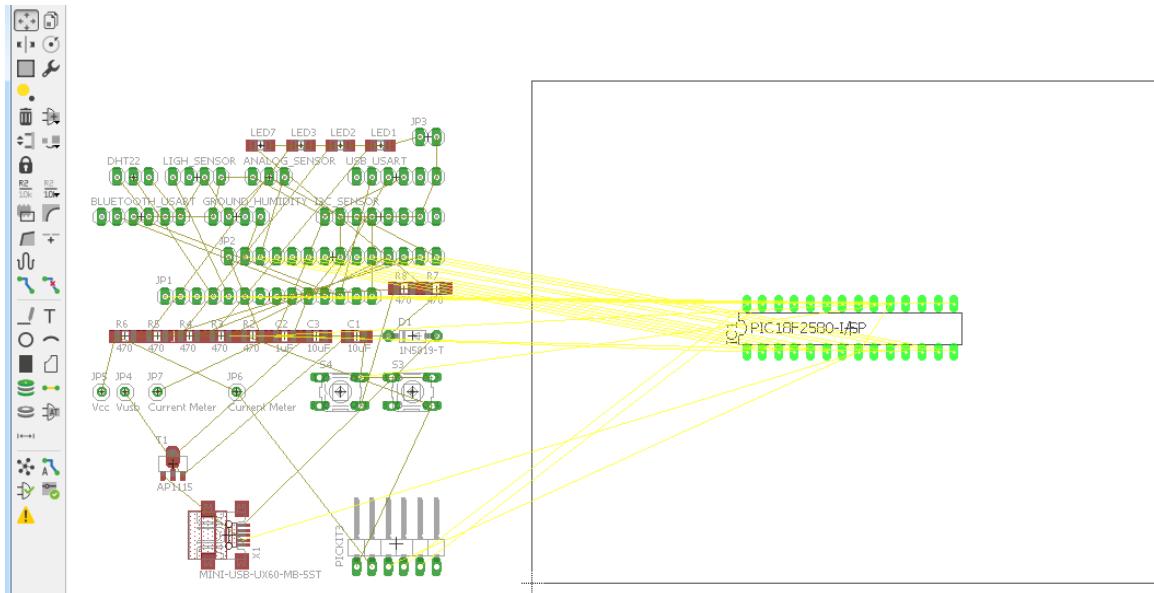


Fig. 4.31 Colocación de los elementos del circuito en la placa

Con todos los elementos colocados, tenemos que dibujar las pistas sin que se solapen entre ellas. Para dibujar las pistas, podemos usar el *autoroute* pero la mejor opción es dibujarlas de forma manual porque normalmente el *autoroute* realiza rutas de una manera incoherente y dificulta el cumplimiento *design rules*. Nuestra placa es de doble capa(TOP y BOTTOM), entonces cuando dibujamos las pistas podemos usar sin ningún problema vías para pasar una pista por debajo de otra para evitar que se solapen (Ver Fig. 4.32).

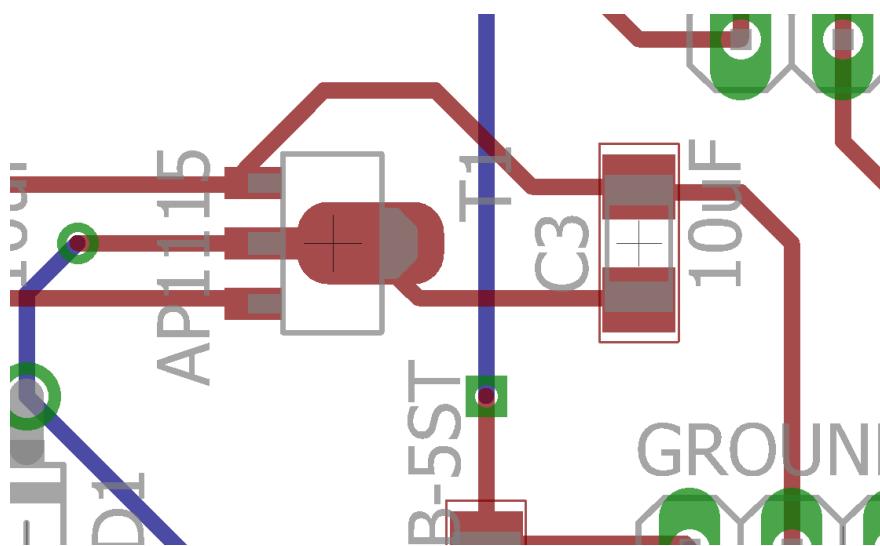


Fig. 4.32 Línea roja y azul pista de la capa top y bottom respectivamente

Finalmente, tenemos que dibujar un polígono que ocupe todo el área de la placa para que Eagle nos dibuje el cobre que no se utiliza que está al lado de las pistas (Ver Fig. 4.33).

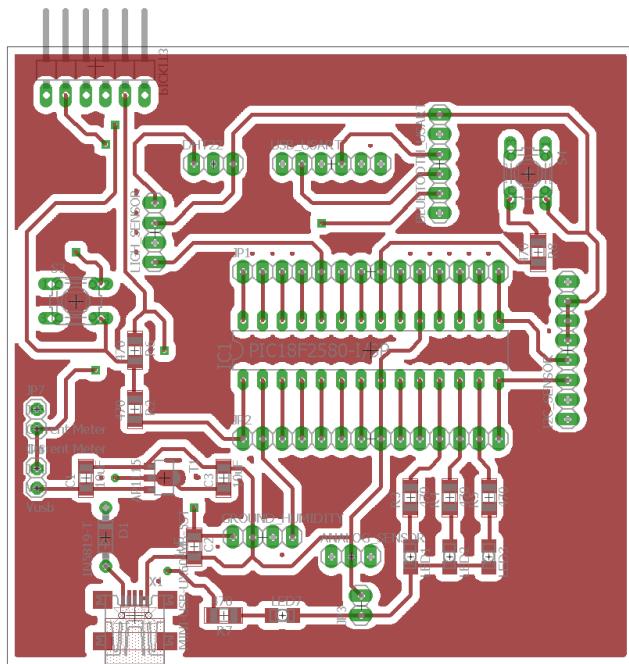


Fig. 4.33 Cara top de la placa

Además, en la cara bottom conectaremos el cobre sobrante a masa (Ver cara bottom en **Fig. 4.34**).

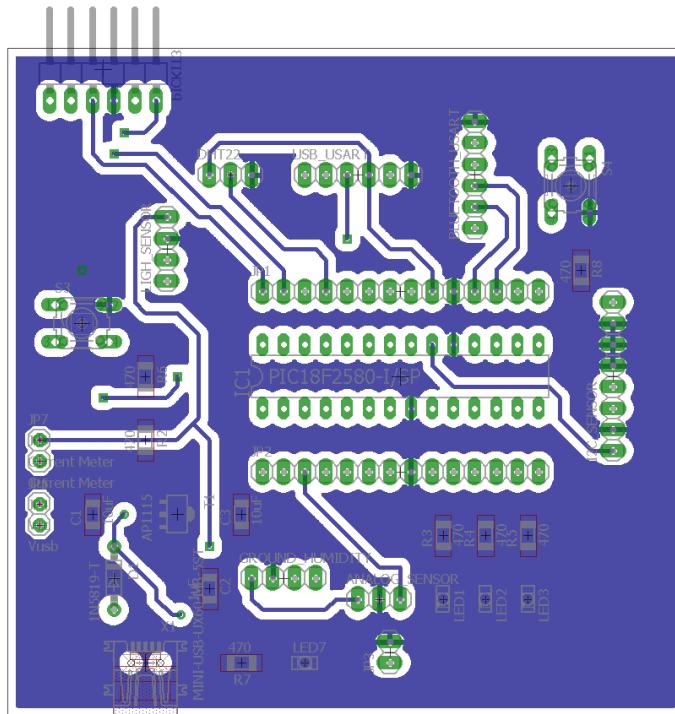


Fig. 4.34 Cara top de la placa

4.4.3 Mecanizado

Con la placa acabada, sólo falta generar los archivos *gerber* para realizar el mecanizado de nuestra placa. Deberemos de generar 4 archivos: TOP, BOTTOM, DRILLS y BOARD.

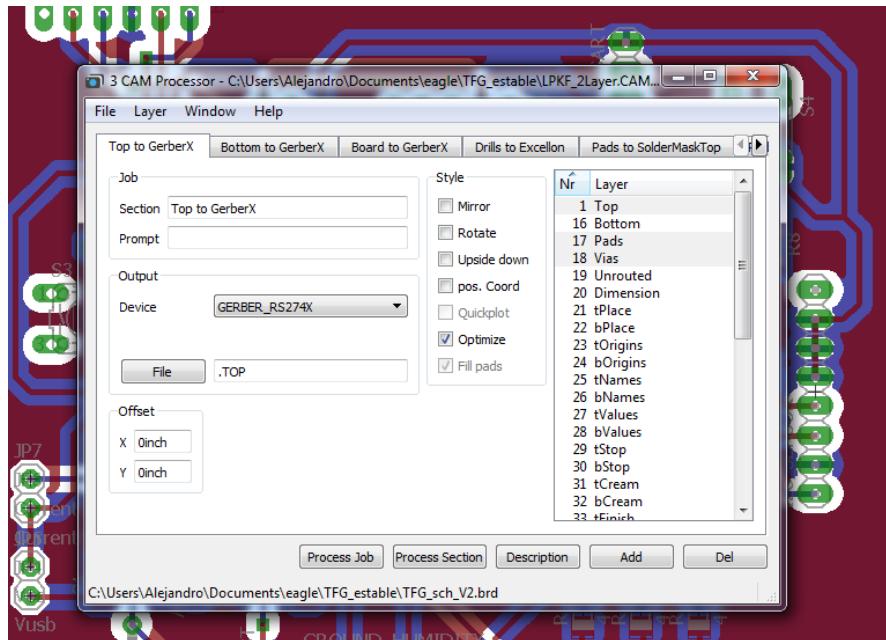


Fig. 4.35 Generación de los archivos *gerber*

Una vez generados los archivos gerber, ya podemos empezar el proceso de mecanizado de nuestra placa. El proceso está explicado en el anexo 7.3.

Una vez finalizado el proceso de mecanizado, sólo queda soldar los elementos y poner en marcha la placa (ver **Fig. 4.36**).

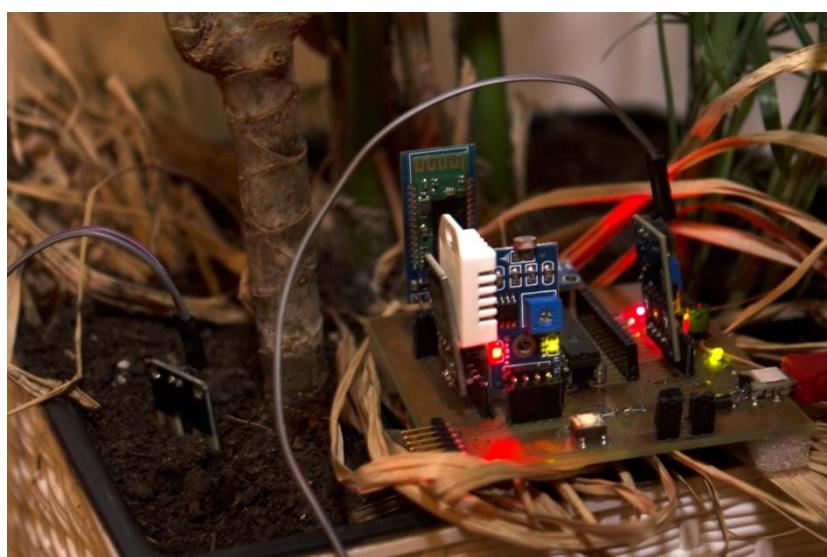


Fig. 4.36 Placa en funcionamiento

Capítulo 5. Conclusiones

Nuestro sistema sensor funciona correctamente pero, probando el sistema, hay campos donde se puede mejorar.

5.1 Mejoras en la alimentación de los sensores

Ahora mismo nuestro sistema sensor siempre mantiene alimentados a los sensores y modulo bluetooth. Tener siempre conectado a la alimentación los sensores implica que cuando entras en modo sleep(puesta en marcha del modo sleep explicada en el anexo 7.1.5), si no desconectas los periféricos, el sistema no está optimizado para el bajo consumo.

En nuestro diseño del sistema sensor en PCB pusimos puentes jumpers para poder realizar mediciones de corriente para analizar los consumos. Tabla de consumos (**Tabla 2.1.**) teniendo en cuenta que la alimentación de la placa PCB es de 3.5V(la placa de pruebas a 3.3V):

Tabla 2.1. Tabla de consumos

Dispositivo	A&T Mode (μ A)	Sleep Mode(actual) (μ A)	Sleep Mode Ideal (μ A)
PIC18F26K20	4530	1.5	1.5
HC-06(bluetooth)	21000-32000*	3000-7000*	~0
KY-013	182	182	~0
MCP9808	180	180	~0
GH Sensor	2400	2400	~0
Light Sensor	2270	2270	~0
DHT22	11	11	~0
Total	28mA-39mA*	13mA-17mA*	~1.5 μ A

(*)El consumo del módulo blueetooth se dispara hasta los 43mA cuando no está vinculado a ningún dispositivo.

La mejora en el circuito de alimentación consistiría en controlar la alimentación de los sensores y el módulo de comunicaciones con un transistor Mosfet NPN. El transistor estaría en On si uno de los puertos del MCU conectado al Gate está en '1' y si el mismo está en '0' el transistor estaría en Off sin alimentar a los sensores y Bluetooth. Con esta solución, nos podríamos acercar al Sleep Mode ideal de la tabla.

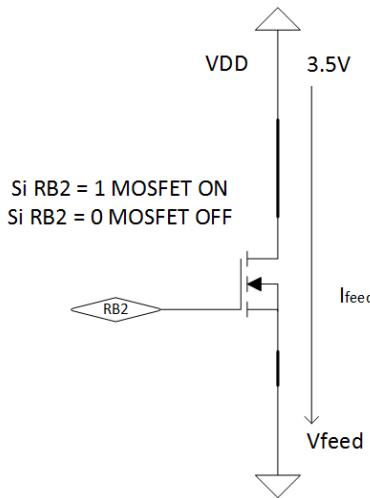


Fig. 5.1 Podríamos utilizar el puerto RB2 del MCU para controlar el MOSFET

Finalmente el consumo, en comparación con el consumo de corriente que nos indica el fabricante en Sleep mode con el WDT activo, es de $1.5\mu\text{A}$ alimentado a 3.5V respecto a 800nA alimentado 1.8V. Podemos decir que es un dato muy positivo.

5.2 Problemas de la comunicación Bluetooth

Un problema derivado de usar Bluetooth se produce cuando se pierde la vinculación entre dispositivos porque no se produce de una manera automática.

Un segundo problema que no afecta al funcionamiento del sistema es el poco alcance, aproximadamente 10m-25m.

Ambos problemas implican realizar el estudio de otras tecnologías de transmisión Wireless como: Zigbee, Wi-Fi, GSM, 3G, LTE, etc.

Zigbee es una tecnología pensada en realizar transmisiones de baja velocidad y es bastante más eficiente energéticamente que bluetooth. Cuando el módulo está en espera consume $1\mu\text{A}-10\mu\text{A}$ respectivamente frente a los $3\text{mA}-7\text{mA}$ del Bluetooth. Ver en (**Fig. 5.2**).

Specification	XBee	XBee-PRO
Power Requirements		
Supply Voltage	2.1 - 3.6 V	3.0 - 3.4 V
Operating Current (Transmit, max output power)	40mA (@ 3.3 V, boost mode enabled) 35mA (@ 3.3 V, boost mode disabled)	295mA (@3.3 V), 170mA (@3.3 V) international variant
Operating Current (Receive))	40mA (@ 3.3 V, boost mode enabled) 38mA (@ 3.3 V, boost mode disabled)	45 mA (@3.3 V)
Idle Current (Receiver off)	15mA	15mA
Power-down Current	< 1 uA @ 25°C	< 10 uA @ 25°C

Fig. 5.2 Tabla de Power Requirements de ZigBee RF Modules by Digi International[10]

5.3 *Implementación de un sistema de Energy Harvesting*

Para implementar un sistema de Energy Harvesting, antes se tendrían que solucionar los problemas anteriores porque no sería posible realizarlo por exceso de consumo.

Nuestra fuente de energía podría provenir de la energía solar, energía geotérmica, energía eólica, los gradientes de salinidad, la energía cinética u otras fuentes de energía.

Una vez escogido el modo en el que realizaremos Energy Harvesting, deberemos realizar los siguientes cambios o novedades en nuestro sistema.

5.3.1 Usar Zigbee

Antes de implementar el sistema de Energy Harvesting cambiaríamos de tecnología de inalámbrica para poder aprovecharnos del Sleep del Zigbee y ser más eficientes cuando no transmitimos datos.

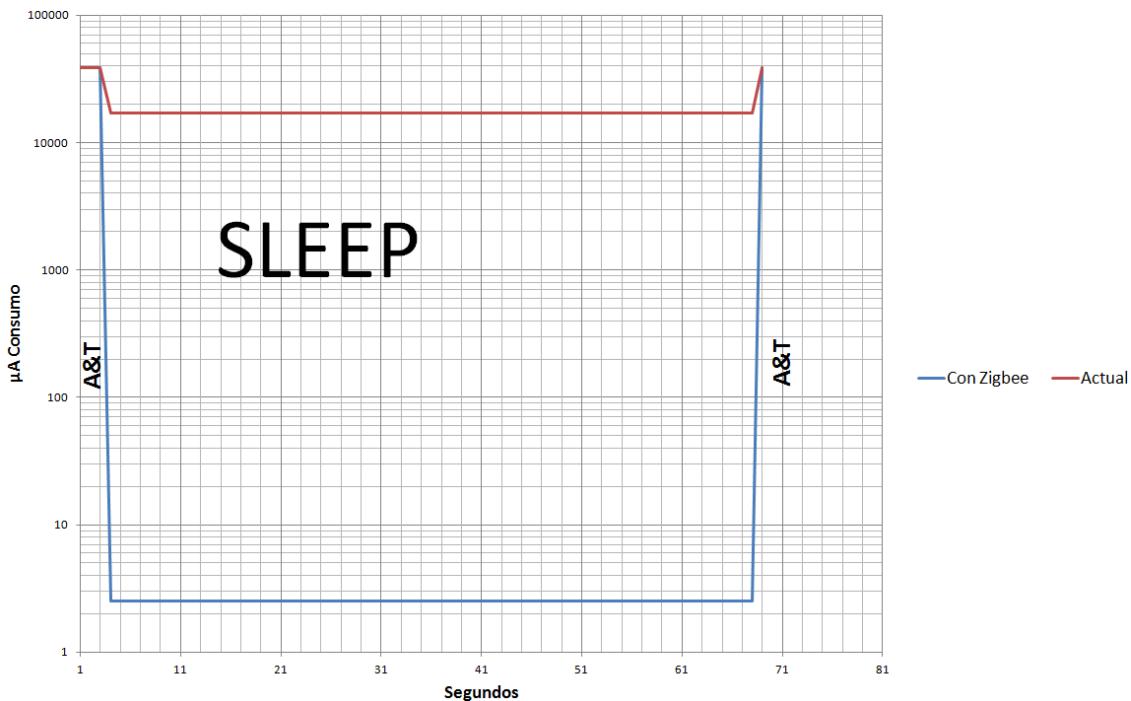


Fig. 5.3 Predicción de mejora usando Zigbee y desconectando los sensores con un Mosfet

5.3.2 Implementar un sistema de almacenamiento de energía

La idea consiste en ir almacenando energía mientras nuestro MCU se encuentra en Sleep para tener la suficiente energía cuando esté adquiriendo y transmitiendo datos. (Ver en (Fig. 1.4).

Capítulo 6. Referencias

- [1] Julio 2016, "Secado y humedecimiento adiabático del aire", http://www.fao.org/docrep/x5027s/x5027S0o.htm#Secado_y_humedecimiento_adiabatico_del_aire
- [2] Junio 2016, "Riego Automático", <http://www.novedades-agricolas.com/es/riego/sistemas-de-riego/riego-automatico>
- [3] Abril 2016, "¿Qué es un sensor termopar?", <http://es.omega.com/prodinfo/termopares.html>
- [4] Agosto 2016, MCP9808 datasheet, <http://www.microchip.com/wwwproducts/en/en556182>
- [5] Agosto 2016, DHT22 datasheet, <https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf>
- [6] Agosto 2016, PIC18F26K20 datasheet, <http://www.microchip.com/wwwproducts/en/PIC18F26K20>
- [7] Agosto 2016, regulador AP1115 datasheet, <http://www.diodes.com/downloads/8034>
- [8] Agosto 2016, EnerChip™ EH Solar Energy Harvester Evaluation Kit , <http://www.cymbet.com/pdfs/DS-72-08.pdf>
- [9] Agosto 2016, XLP 16-bit Development Board, <http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=DM240311>
- [10] Agosto 2016, "XBEE/RF SOLUTIONS" , <http://www.digi.com/products/xbee-rf-solutions>
- [11] Julio 2016, PICKIT3, <http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=PG164130>
- [12] Julio 2016, Labview, <http://www.ni.com/labview/esa/>
- [13] Agosto 2016, datasheet SHT25, <https://www.soselectronic.com/productdata/10/17/69/101769/SHT25.pdf>
- [14] Julio 2016, datasheet LDR GL5528, <https://pi.gate.ac.uk/pages/airpi-files/PD0001.pdf>
- [15] Agosto 2016, válvulas Bürket, <http://www.burkert.com/en/products/solenoid-valves>



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ANEXOS

TÍTOL DEL TFG

TITULACIÓ:

AUTOR: Manuel Pérez Pérez

DIRECTOR: José González González

DATA: 23 de setembre del 2014

Capítulo 7. ANEXOS

7.1 Módulos del PIC18F26K20

7.1.1 TIMER3

Deseamos crear una base de tiempo de 1ms y para ello utilizaremos un el TIMER3 de PIC18F.

FIGURE 15-2: TIMER3 BLOCK DIAGRAM (16-BIT READ/WRITE MODE)

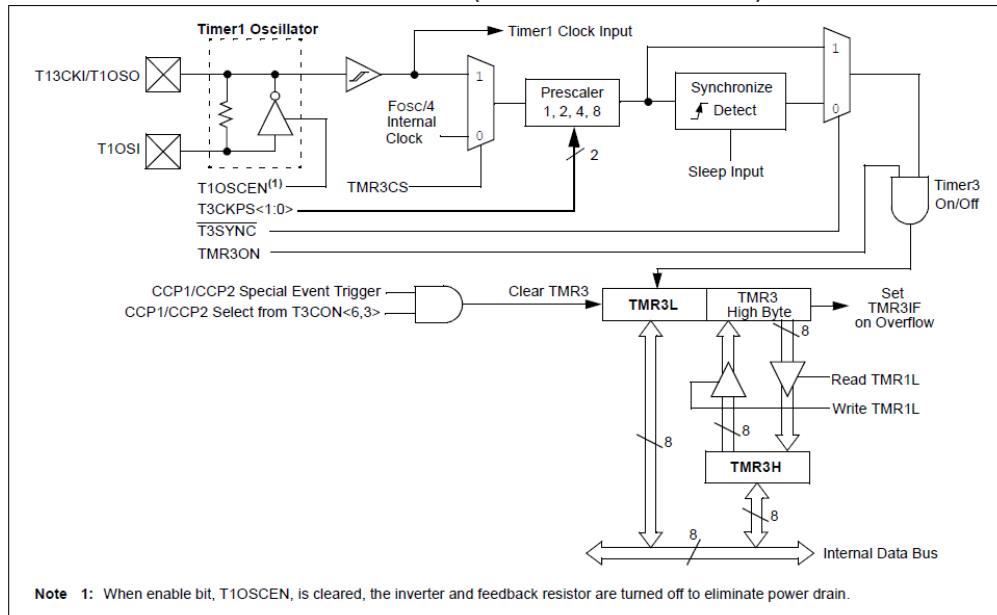
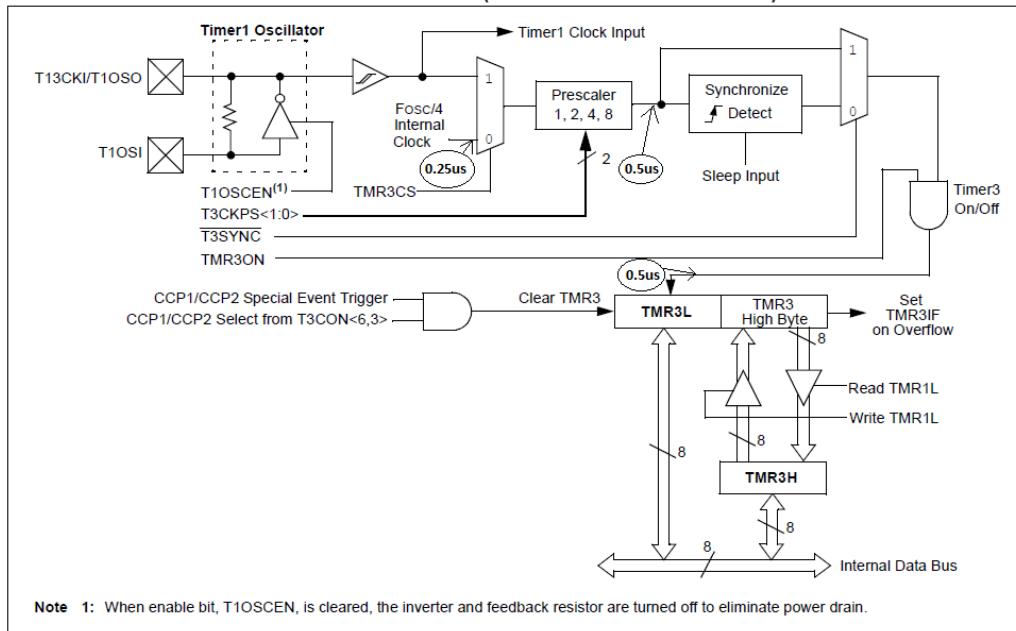


Fig. 7.1 Diagrama de bloques del TIMER3

Primero de todo utilizaremos el oscilador interno del PIC de 16Mhz, a continuación veremos el tiempo de ciclo paso a paso:

FIGURE 15-2: TIMER3 BLOCK DIAGRAM (16-BIT READ/WRITE MODE)**Fig. 7.2** Tiempo de ciclo del pulso en las diferentes etapas del Timer3

En el módulo TIMER3 se produce la interrupción TMR3IF cada vez que TMR3 llega su máximo (FFFFx0). Cada 0.5us se suma uno al contador TMR3 por lo tanto necesitamos que se produzca la interrupción cada 2000 pulsos. Para ello, antes deberemos de precargar un valor a TMR3 para que solamente se produzcan 2000 pulsos. Siendo un contador de 16bits habrán 65536 pulsos para que se produzca la interrupción TMR3IF, para conseguir que solamente cuente 2000 pulso debemos pre cargar en el registro TMR3 el valor de F830x0 igual a 63536.

```

9   #include "TIMER3.h"
10
11 void InitializeTimer3(void) {
12
13     TMR3IE = 1;
14     T3CON = 0;
15     T3CONbits.RD16 = 1;      // 0 = R/W as 8 bits, 1 = R/W as 16 bits
16     T3CONbits.T3CCP2 = 0;    // used with both CCP modules
17     T3CONbits.T3CKPS1 = 0;  // 2 bit Prescale value: 00: PS = 1,
18     T3CONbits.T3CKPS0 = 1;  // 01: PS = 2, 10: PS = 4, 11: PS = 8
19     T3CONbits.T3CCP1 = 0;    // used with CCP2 module
20     T3CONbits.T3SYNC = 0;    // Timer 1 Osc Sync: 0 = on, 1 = off
21     T3CONbits.TMR3CS = 0;    // Clock Source: 0 = Fcy, 1 = Ext Clk
22     T3CONbits.TMR3ON = 1;    // 0 = turn timer off, 1 = turn timer on
23     TMR3H = 0xF8;           // TMR3H must be written first
24     TMR3L = 0x46;           // count up from F830 but there is a small overhead and TMR3L must be 0x46
25     TMR3IF = 0;              // reset the interrupt flag
26
27
28 }
```

Fig. 7.3 Código utilizado para configurar el TIMER3

Con la base tiempo de 1ms lograda solamente nos falta añadir un postscaler para lograr la frecuencia de muestreo que se desee.

7.1.2 PWM con TIMER2

La generación del PWM se genera con la utilización del TIMER2 para generar el periodo PWM y la anchura del pulso PWM.

FIGURE 11-3: SIMPLIFIED PWM BLOCK DIAGRAM

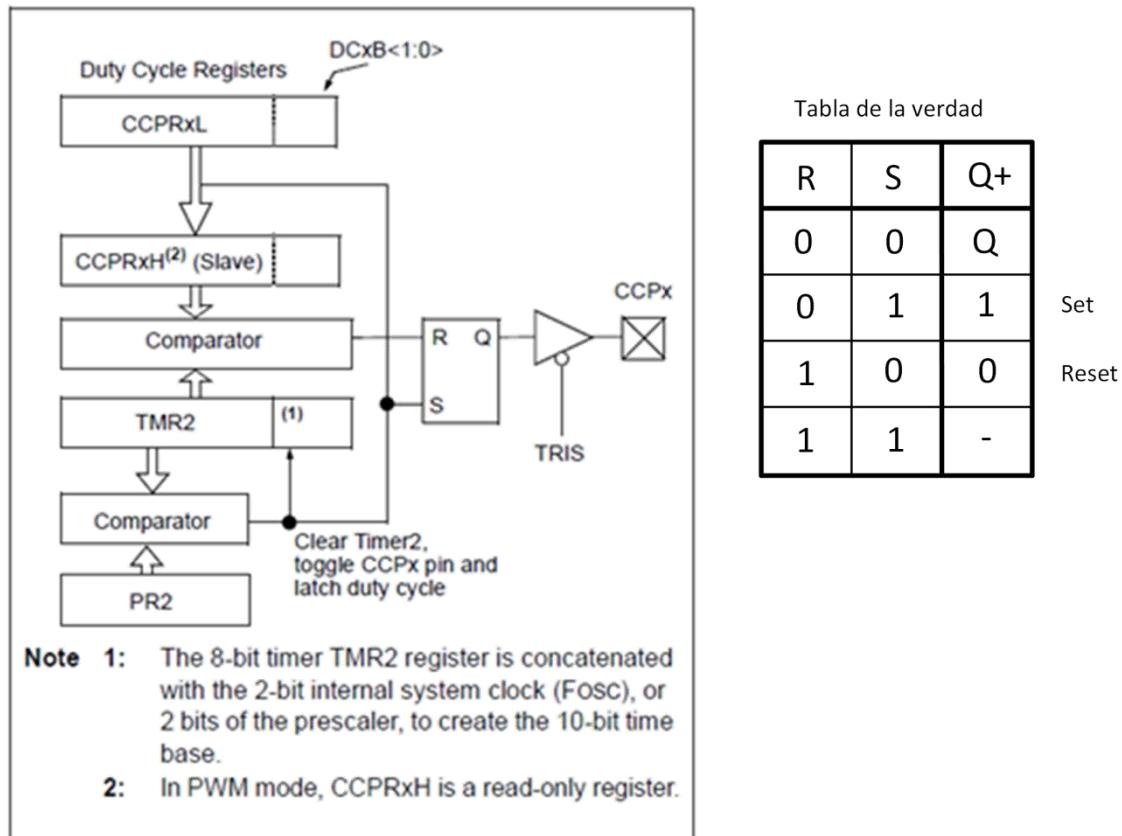


Fig. 7.4 Diagrama de bloques simplificado de PWM

Primero de todo tendremos que configurar el periodo PWM con el TIMER2. Tendremos que seleccionar los valores de **Prescaler** y **PR2** para obtener el periodo deseado (ver **Fig. 7.5**).

FIGURE 14-1: TIMER2 BLOCK DIAGRAM

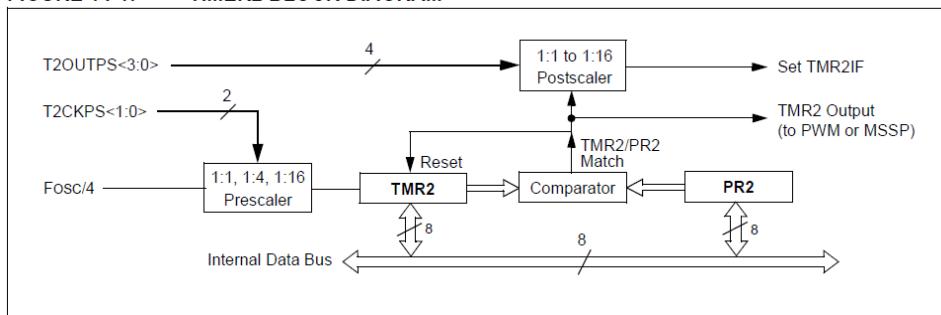


Fig. 7.5 Diagrama de bloques del TIMER2

El datasheet nos proporciona la fórmula para calcular el periodo PWM(Eq 11-1) y la anchura del pulso(11-2).

EQUATION 11-1: PWM PERIOD

$$\text{PWM Period} = [(PR2) + 1] \cdot 4 \cdot TOSC \cdot \\ (TMR2 Prescale Value)$$

Note: $TOSC = 1/Fosc$.

Fig. 7.6 Ecuación para calcular el periodo del PWM

Configuraremos un PR2 de 199 y TMR2 Prescale de 16. La Fosc es de 16MHz

$$PWM_{period} = [199 + 1] \times 4 \times 0.0625\mu s \times 16 = 800\mu s$$

EQUATION 11-2: PULSE WIDTH

$$\text{Pulse Width} = (CCPRxL:DCxB<1:0>) \cdot \\ TOSC \cdot (TMR2 Prescale Value)$$

EQUATION 11-3: DUTY CYCLE RATIO

$$\text{Duty Cycle Ratio} = \frac{(CCPRxL:DCxB<1:0>)}{4(PR2 + 1)}$$

Fig. 7.7 Ecuaciones para calcular la anchura del pulso PWM y el Duty Cycle

La duración del pulso PWM nunca será mayor al periodo PWM si configuramos bien los valores de **PR2** y **CCPRxL:DCxB<1:0>**.

A continuación mostraremos un gráfico (**Fig. 7.8**) donde se muestra el funcionamiento del módulo PWM con el TIMER2 siguiendo la tabla de la verdad de la figura (**Fig. 7.4**).

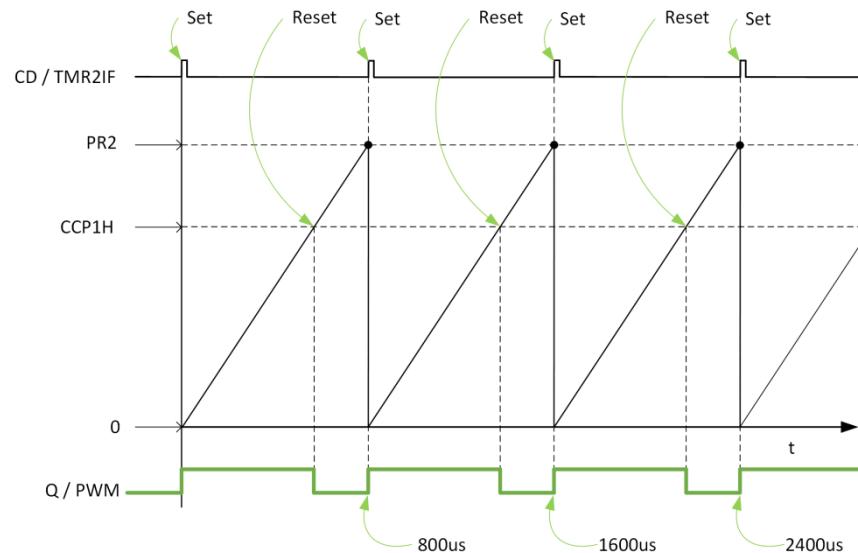


Fig. 7.8 Generación del la señal PWM

7.1.3 Comunicación I²C

En el proyecto utilizaremos el protocolo I²C para comunicarnos con sensores que utilicen este protocolo.

Con la ayuda de las librerías de periféricos que nos proporciona microchip nos hace más fácil la implementación del protocolo.

Primero de todo tenemos que seleccionar con la función `Open_I2C` la frecuencia de clock(100KHz) que usaremos y el modo Master mode.(código Fig. 7.10)

TABLE 17-3: I²C™ CLOCK RATE W/BRG

Fosc	Fcy	BRG Value	F _{SCL} (2 Rollovers of BRG)
64 MHz	16 MHz	27h	400 kHz ⁽¹⁾
64 MHz	16 MHz	32h	313.7 kHz
64 MHz	16 MHz	3Fh	250 kHz
40 MHz	10 MHz	18h	400 kHz ⁽¹⁾
40 MHz	10 MHz	1Fh	312.5 kHz
40 MHz	10 MHz	63h	100 kHz
16 MHz	4 MHz	09h	400 kHz ⁽¹⁾
16 MHz	4 MHz	0Ch	308 kHz
16 MHz	4 MHz	27h	100 kHz
4 MHz	1 MHz	09h	100 kHz

Fig. 7.9 Tablas para seleccionar el clock

```
void Open_I2C1(void)
{
    SSPADD= 0x27; //100Khz clock
    OpenI2C(MASTER, SLEW_OFF);

    IdleI2C();
}
```

Fig. 7.10 Código de configuración

A continuación mostraremos en las 2 siguientes imágenes cómo nos debemos comunicar con el sensor según el datasheet de PIC que PIC28F26K20.

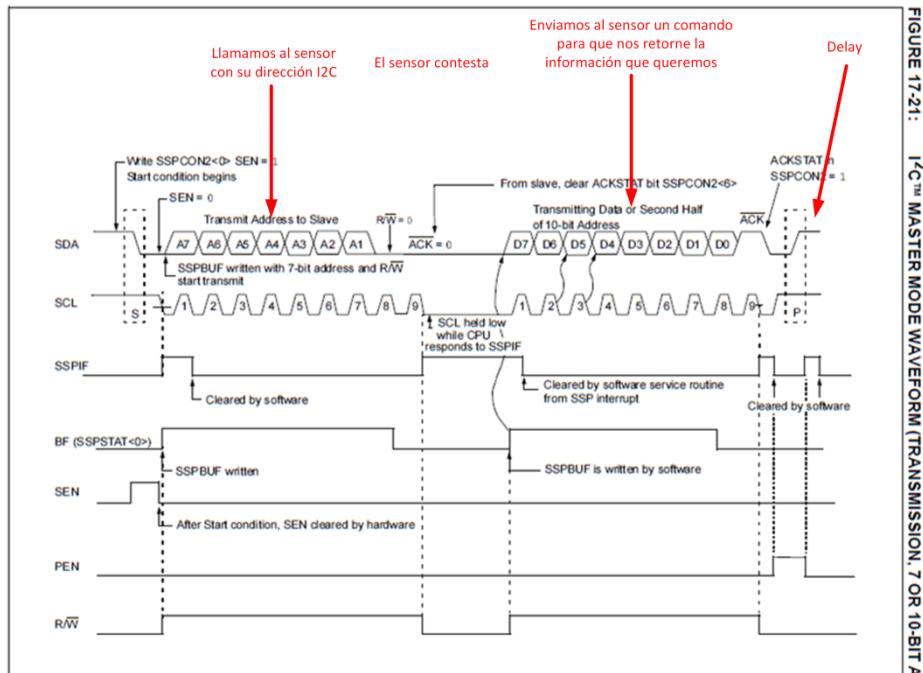


Fig. 7.11 Primera fase de la comunicación

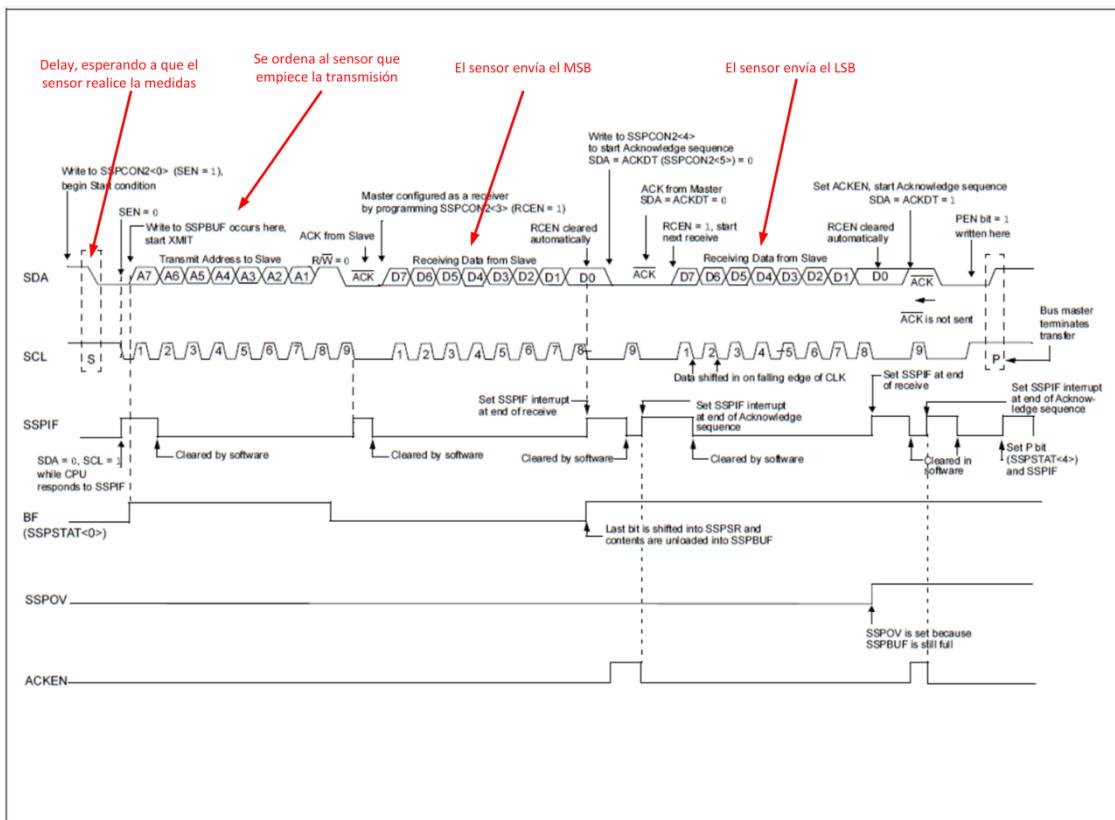


Fig. 7.12 Segunda fase de la comunicación

Código utilizado:

```

unsigned char UpperByte = 0;
unsigned char LowerByte = 0;
unsigned int tmp = 0;
float Temperature = 0;
ResetVariables_I2C();
StartI2C();

WaitFlag();

//Write Slave address and set master for transmission
WriteI2C(ReadAddressIC);
WaitFlag();
|
WaitForACK();

WriteI2C(RegisterAddress);
WaitFlag();
WaitForACK();
__delay_ms(45);

```

Fig. 7.13 Código de la primera fase de la comunicación

```

__delay_ms(45);
ResetVariables_I2C();
RestartI2C();
WaitFlag();

WriteI2C(ReadAddressIC + 1);
WaitFlag();
WaitForACK();
ResetVariables_I2C();

UpperByte = ReadI2C();
WaitFlag();
AckI2C();
ResetVariables_I2C();

LowerByte = ReadI2C();
WaitFlag();
ResetVariables_I2C();

NotAckI2C();
WaitFlag();
ResetVariables_I2C();
StopI2C();
WaitFlag();
IdleI2C();

```

Fig. 7.14 Código de la Segunda fase de la comunicación

7.1.4 EUSART

El módulo Enhanced Universal Synchronous Asynchronous Receiver Transmitter (EUSART) lo utilizaremos en modo asíncrono siendo la conexión que creemos UART.

A continuación podemos ver los diagramas de bloques del módulo EUSART del transmisor y receptor.

FIGURE 18-1: EUSART TRANSMIT BLOCK DIAGRAM

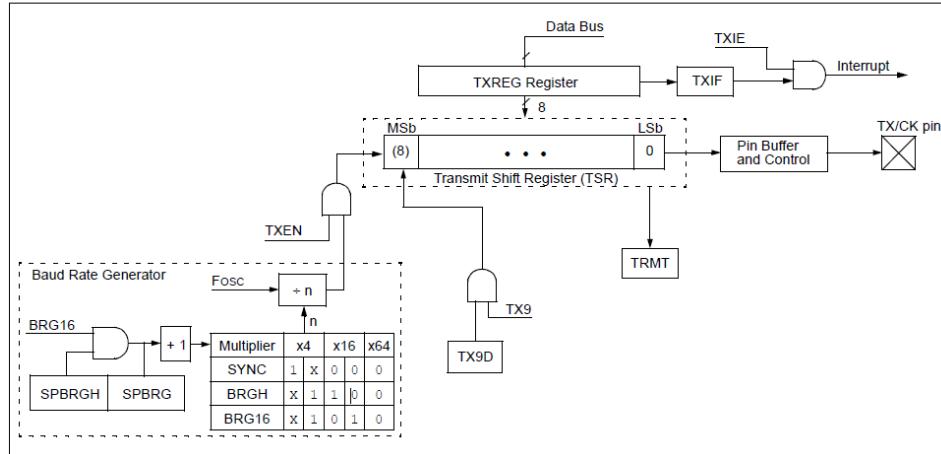


Fig. 7.15 Diagrama de bloques del transmisor EUSART

FIGURE 18-2: EUSART RECEIVE BLOCK DIAGRAM

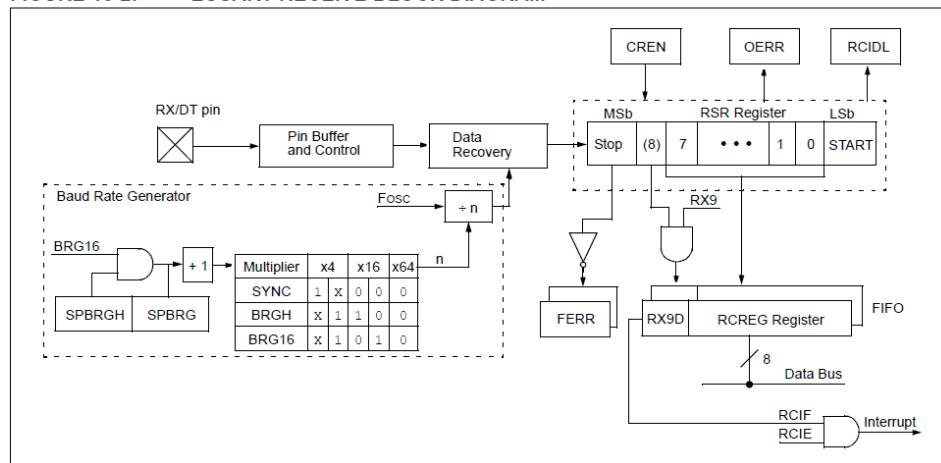


Fig. 7.16 Diagrama de bloques del receptor EUSART

Primero de todo tenemos que configurar nuestra conexión UART. Hay varios parámetros que tenemos que tener en cuenta:

- Baudrate
- Trama de 8-16bits
- Comunicación síncrona o asíncrona
- **BRGH:** High Baud Rate Select bit

Para configurar el baudrate, debemos saber qué configuración utilizaremos en nuestra comunicación y la Fosc(16MHz del oscilador interno). Nuestra configuración será la siguiente: Trama de 8bits(BRG16 = 0), comunicación asíncrona (SYNC = 0) y BRGH = 1.

BAUD RATE	SYNC = 0, BRGH = 1, BRG16 = 0											
	FOSC = 64.000 MHz			FOSC = 18.432 MHz			FOSC = 16.000 MHz			FOSC = 11.0592 MHz		
	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)
300	--	--	--	--	--	--	--	--	--	--	--	--
1200	--	--	--	--	--	--	--	--	--	--	--	--
2400	--	--	--	--	--	--	--	--	--	--	--	--
9600	--	--	--	9600	0.00	119	9615	0.16	103	9600	0.00	71
10417	--	--	--	10378	-0.37	110	10417	0.00	95	10473	0.53	65
19.2k	19.23k	0.16	207	19.20k	0.00	59	19.23k	0.16	51	19.20k	0.00	35
57.6k	57.97k	0.64	68	57.60k	0.00	19	58.82k	2.12	16	57.60k	0.00	11
115.2k	114.29k	-0.79	34	115.2k	0.00	9	111.1k	-3.55	8	115.2k	0.00	5

Fig. 7.17 Tablas para configurar el baudrate de la comunicación UART

Por lo tanto, el valor de SPBRG que debemos de configurar para tener un baudrate de 9600 será 103.

A continuación está el código C utilizado para poner en marcha la comunicación UART utilizando las librerías USART de microchip.

```
OpenUSART (USART_TX_INT_OFF &
           USART_RX_INT_ON &
           USART_SYNCH_MODE &
           USART_EIGHT_BIT &
           USART_CONT_RX &
           USART_BRGH_HIGH, 103); //9600baudios
```

Fig. 7.18 Código utilizado para configurar la comunicación UART

7.1.5 Sleep Mode

El modo sleep hace que el MCU deje de ejecutar código del programa principal con lo cual se reduce el consumo drásticamente y utilice el oscilador de baja frecuencia de 31KHz.

Según las condiciones de diseño queremos que el sistema sensor entre en reposo cuando al acabe el ciclo de medición/trasmisión de datos. El sistema debía de estar en reposo durante 1min. El watchdog timer(WDT) se puede configurar para que despierte el sistema después de un periodo de tiempo, a continuación veremos cómo.

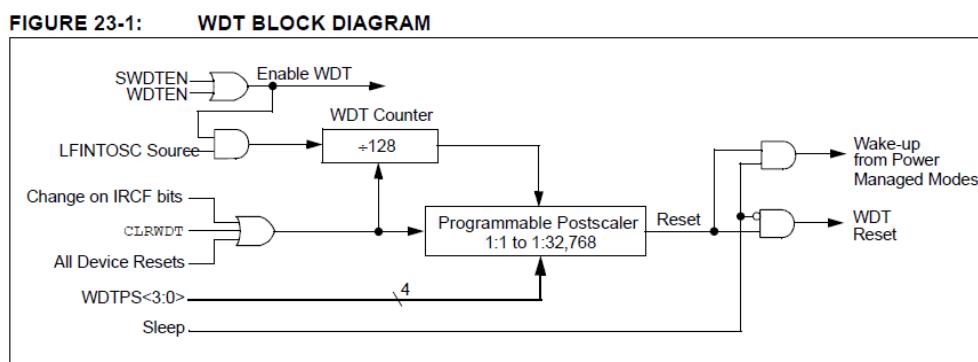


Fig. 7.19 Diagrama de bloques del WDT

El único parámetro configurable es el Postscaler de 1 a 32768 equivalente a 4ms a 131.072 segundos(2.18 min). Nosotros programaremos el postscaler a 16384 para que despierte el sistema cada 65.536 segundo(aproximadamente un minuto).

```
#pragma config WDTEN = ON      // Watchdog Timer Enable bit (WDT is always enabled. SWDTEN bit has no effect)
#pragma config WDTPS = 16384    // Watchdog Timer Postscale Select bits (1:16384) 65s
```

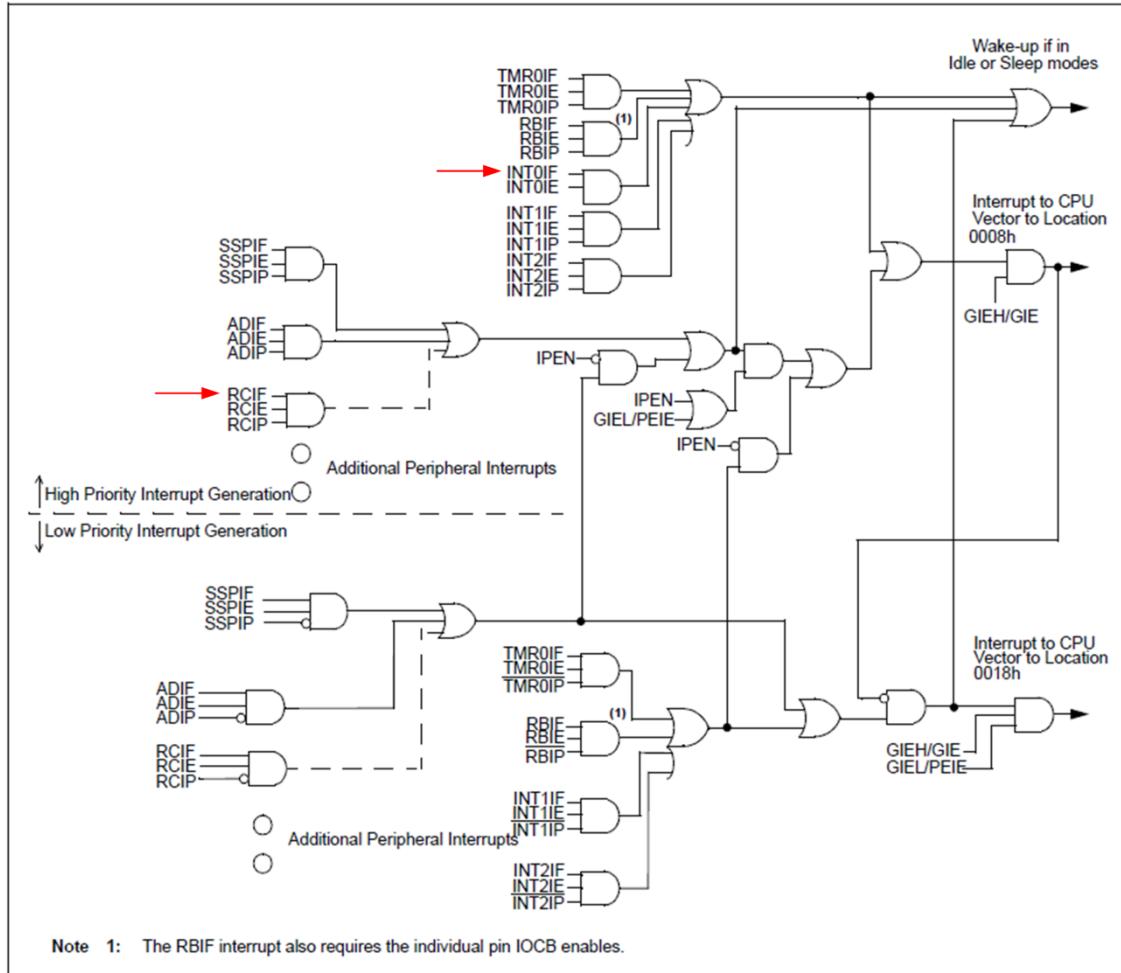
Fig. 7.20 Código en C utilizado para configurar el WDT

El programa entra en Sleep cuando ejecutamos el comando `Sleep()` y el programa deja de correr. Cuando el programa sale del *sleep mode* realizamos un *reset* de los registros de WDT con el comando `CLRWDT()`.

```
Sleep();
CLRWDT();
```

Fig. 7.21 Código en C utilizado para entrar en *Sleep mode*

También podemos despertar el MCU con otra interrupción que no sea la del WDT. A continuación tenemos la lógica de las interrupciones:

FIGURE 9-1:**PIC18 INTERRUPT LOGIC****Fig. 7.22 Lógica de la interrupciones del PIC18**

Éste es esquema general de la lógica de las interrupciones en el cual utilizaremos sólo las interrupciones externas del USART (RCIF) y la del botón (INT0IF) para despertar el MCU.

La interrupción INT0IF despierta automáticamente el MCU pero la interrupción RCIF no sin antes habilitar la opción de que lo pueda hacer vía USART. Código en C:

```
#ifdef USARTWAKEUP
    WUE = 1; // wake up usart enable
#endif
```

Fig. 7.23 Código en C para habilitar que se despierte el MCU por vía USART

7.1.6 ADC-10bits

Nuestro PIC tiene 12 entradas en el ADC. Ver diagrama de bloques de ADC:

FIGURE 19-1: ADC BLOCK DIAGRAM

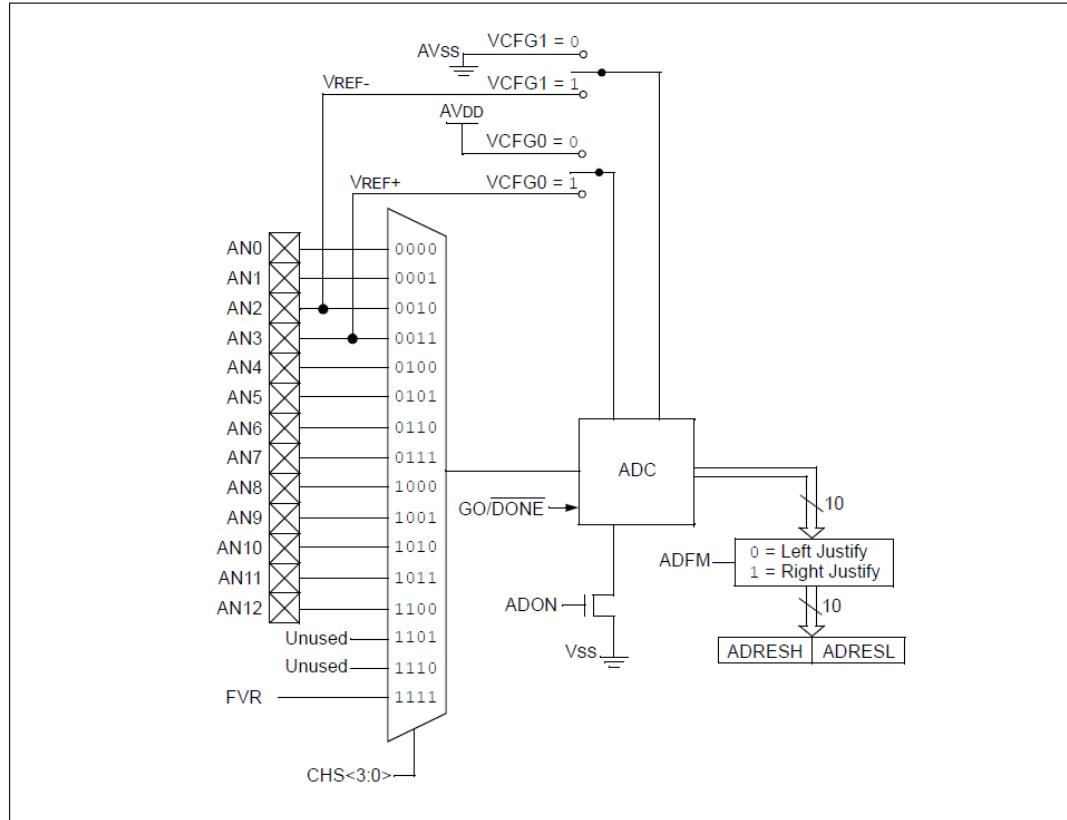


Fig. 7.24 Diagrama de bloques del ADC

Por lo que respecta a la configuración del ADC, para realizar una medida deberemos de encender el módulo ADC y seleccionar qué canal queremos AN queremos capturar.

FIGURE 19-2: 10-BIT A/D CONVERSION RESULT FORMAT

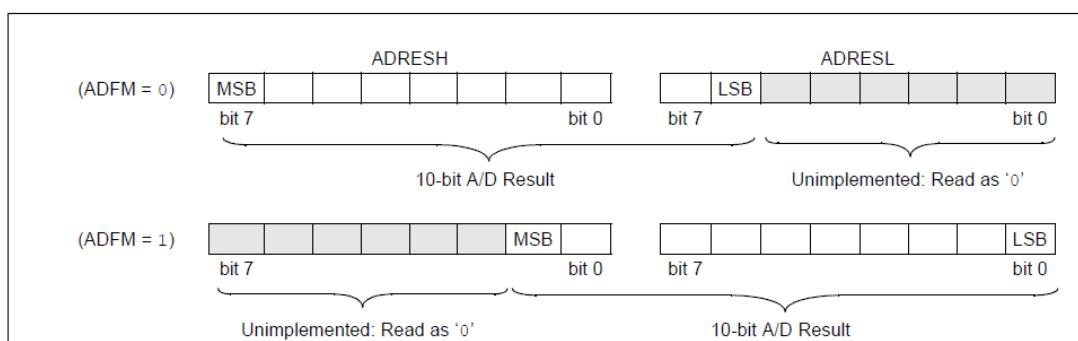


Fig. 7.25 Formato de datos del ADC

Además debemos seleccionar qué formado de datos queremos utilizar(Fig 19-2). Utilizaremos el segundo porque es más sencillo trabajar con él a posteriori. Código C de los métodos de configuración del ADC y captura de una muestra:

```
9  #include "ADC.h"
10
11 void InitADC(){
12
13     ADON = 1;
14     ADFM = 1;
15
16 }
17 unsigned int GetSampleADC() // sensor LDRInitADC(){
18
19     GO = 1;
20     while (DONE) {
21         ;
22     } //aqui pongo el while para que capture
23     Value_AD = ADRES;
24
25     return Value_AD;
26 }
27 void SelectChannelADC(int channel){
28     ADCON0bits.CHS = channel;
29 }
30
31 void StopADC(){
32
33     ADON = 0;
34 }
```

Fig. 7.26 Métodos utilizados para usar el ADC

7.2 Sensores

7.2.1 DHT22

El sensor DHT11 originalmente tiene 4 puertos de los cuales solo usaremos 3 para conectarlo a nuestra MCU tal y como nos indica el fabricante.

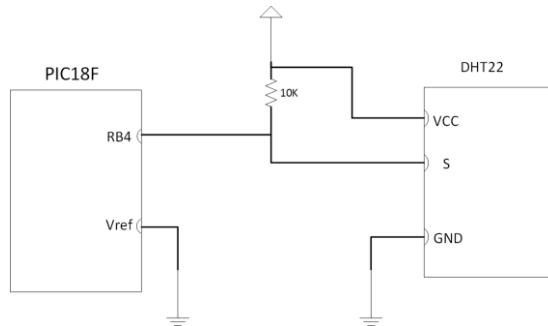


Fig. 7.27 Esquema de conexión del DHT22 con un MCU

Para comunicarnos con el sensor debemos de implementar en nuestro MCU el protocolo que el fabricante nos proporciona el datasheet.

La MCU se comunicará con el DHT y éste le enviará 5 bytes de datos donde los 2 primeros corresponderán a la Humedad Relativa, los 2 siguientes a la Temperatura y el último byte para el Checksum.

Protocolo de comunicación

Primero de todo, el MCU mandará una secuencia de START al DHT y después el mismo esperará a que el DHT responda. Después el DHT mandará una señal al MCU para indicar de que está preparado para inicial la trasmisión bit a bit.

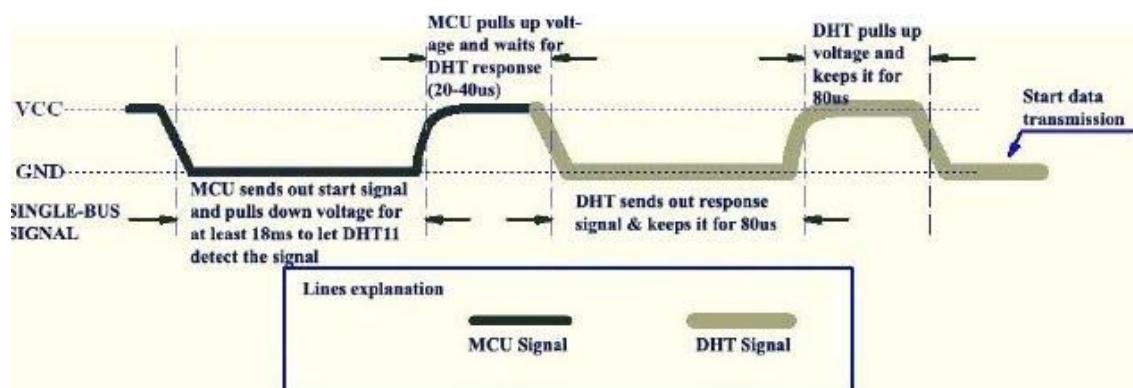


Fig. 7.28 Secuencia de START para comunicarnos con el DHT22

Siempre que inicia la trasmisión de un bit el sensor, mandará un pulso de bajo voltaje de 50us y después enviará un pulso de alto voltaje que según la duración del mismo el pulso significará que es un "0" (26-28us **Fig. 7.28**) o un "1" (70us **Fig. 7.28**).

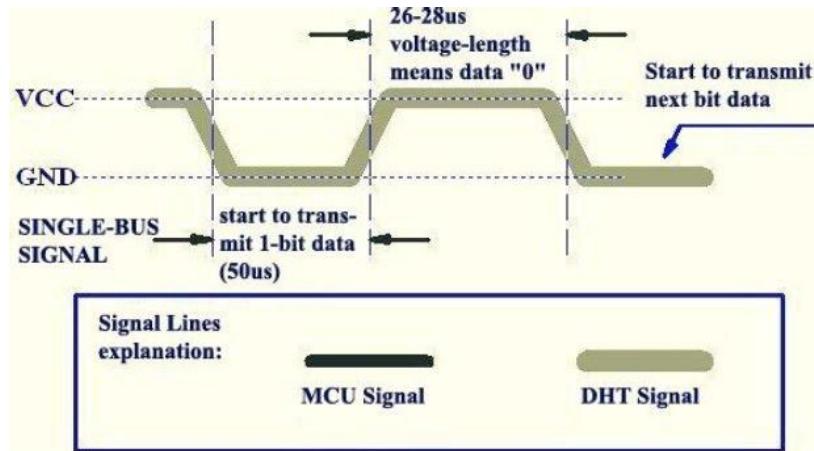


Fig. 7.28 Secuencia del DHT22 para enviar un '0'

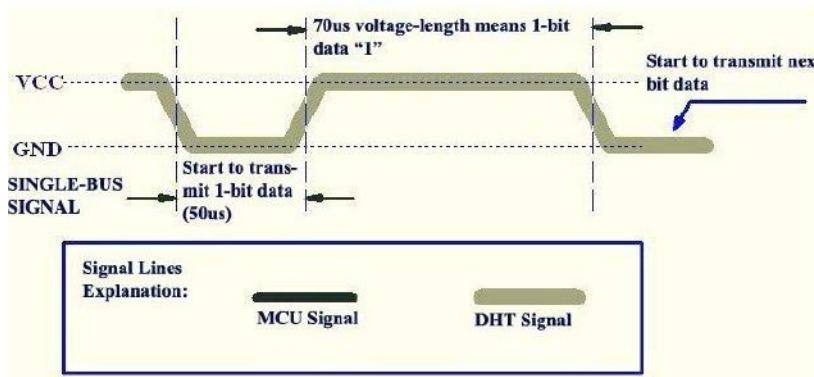


Fig. 7.29 Secuencia del DHT22 para enviar un '1'

```

3 void start_signal()
4 { DataDir = 0;      // Data port is output
5   Data    = 0;
6   __delay_ms(18);
7   DataDir = 1;      // Data port is input
8   __delay_us(30);
9 }
```

Fig. 7.30 Código para realizar la secuencia de START

```
10     unsigned short check_response()
11     { TOUT = 0;
12     TMR2 = 0;
13     TMR2ON = 1; // start timer
14     while (!Data && !TOUT);
15     if (TOUT)
16     return 0;
17     else
18     { TMR2 = 0;
19     while (Data && !TOUT);
20     if (TOUT)
21     return 0;
22     else
23     { TMR2ON = 0;
24     return 1;
25     }
26   }
27 }
```

Fig. 7.32 Código para verificar que recibimos respuesta por parte del sensor para medir el tiempo utilizamos el TIMER2

```
29
30     unsigned short read_byte()
31     { unsigned short num = 0;
32     DataDir = 1;
33     for (i=0; i<8; i++)
34     { while (!Data && !TOUT);
35     TMR2 = 0;
36     TMR2ON = 1;
37     while (Data && !TOUT);
38     TMR2ON = 0;
39     if (TMR2 > 49)
40     num |= 1<<(7-i); // si el tiempo > 49us, Data es 1
41   }
42   return num;
43 }
```

Fig. 7.32 Código para guardar los bits en bytes utilizando el TIMER2

```
44  void GetDataSensorDHT22(void){  
45      char CharTX[];  
46      start_signal();  
47          check = check_response();  
48  
49          if (!check)  
50          {  
51  
52              SendStringUSART("No response from the sensor DHT22");  
53  
54          }  
55      else  
56      {  
57          RH_Byte1 = read_byte();  
58          RH_Byte2 = read_byte();  
59          T_Byte1 = read_byte();  
60          T_Byte2 = read_byte();  
61          CheckSum = read_byte();  
62  
63  
64          rh = RH_Byte2 | (RH_Byte1<<8);  
65          temp = T_Byte2 | (T_Byte1<<8);  
66          sign = 0;  
67          if (temp>0x8000)  
68          { temp &= 0xffff;  
69              sign = 1;  
70          }  
71          Value_rhdht22 = (float) (rh/10.0);  
72          Value_tempdht22 = (float) (temp/10.0);  
73  
74      }  
75  
76  
77  }
```

Fig. 7.33 Código para realizar el protocolo de comunicación del DHT22 utilizando las funciones anteriores

7.3 Proceso de mecanizado de la PCB

7.3.1 CAM

Una vez exportados los ficheros *gerber* del *Eagle* se importan en el programa Circuit CAM para generar los archivos CAM que se utilizarán en la máquina que nos haga la placa.

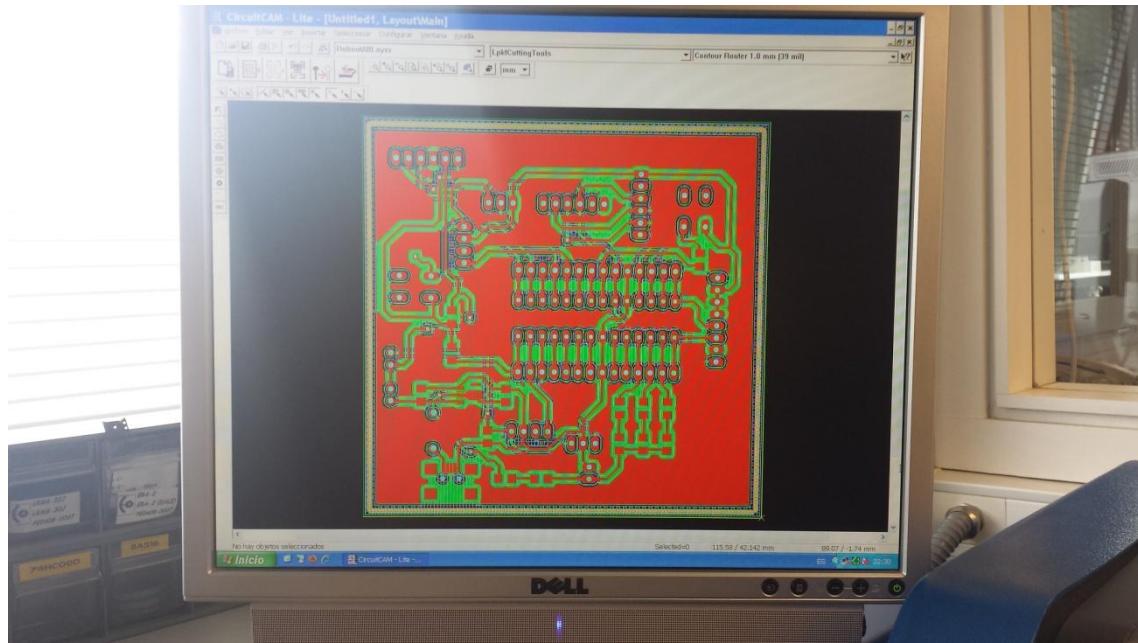


Fig. 7.34 Gerbers importados en el programa Circuit CAM

7.3.2 Drills

El proceso de mecanizado empezará con la realización de los drills. Durante el proceso la máquina nos irá pidiendo que cambiemos las brocas por otras de distintos diámetros.



Fig. 7.35 Brocas de diferentes diámetros

Una vez finalizado el proceso, tendremos que realizar el metalizado de la placa para unir la capa TOP con la BOTTOM por los agujeros.

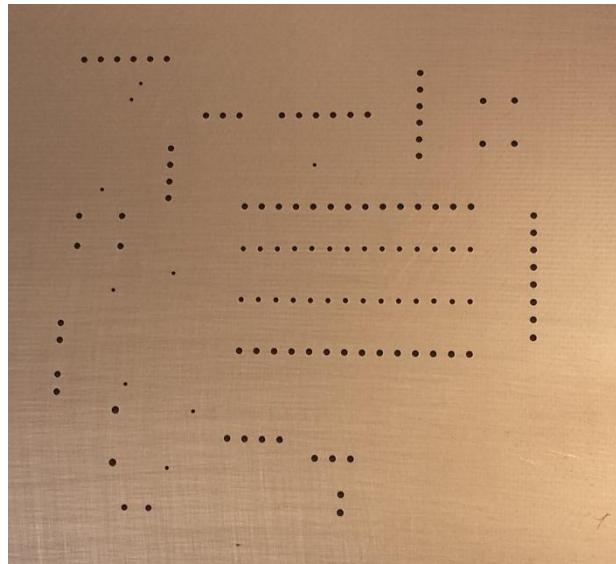


Fig. 7.36 Todos los taladros realizados

7.3.3 Metalizado

El proceso de metalizado tiene 4 fases:

1. Lavado de la placa

Primero se realiza un lavado con agua caliente y luego un lavado con jabón. Ambos procesos duran 2-5 minutos.



Fig. 7.37 Lavado con agua caliente



Fig. 7.38 Lavado con jabón

2. Baño en carbono activo

Seguidamente se realiza un baño en carbono activo, el proceso dura 15 minutos.

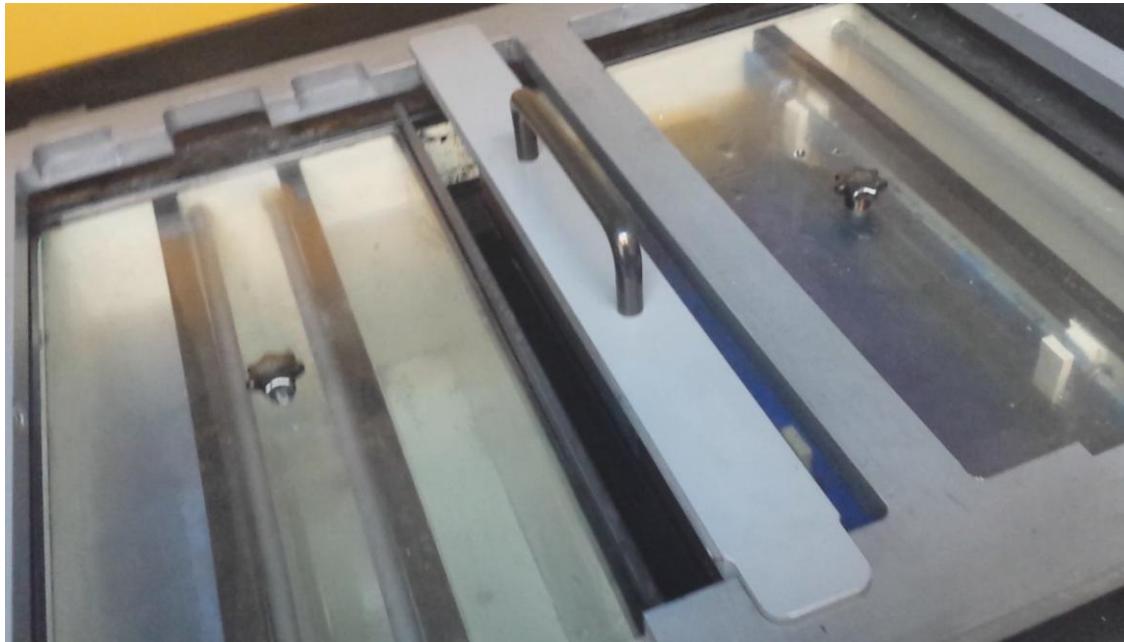


Fig. 7.39 Baño en carbono activo

3. Electrolisis en ácido sulfúrico

Finalmente se realiza una electrolisis en ácido sulfúrico 40% para que las partículas se adhieran en los agujeros y conecten la parte TOP y BOTTOM.

4. Último lavado

Finalmente se realiza un lavado en agua.

7.3.4 Pistas TOP y BOTTOM

Con el proceso de metalizado realizado, seguimos con la máquina para hacer las capas TOP y BOTTOM. Finalizando este proceso, ya tendríamos nuestra placa.

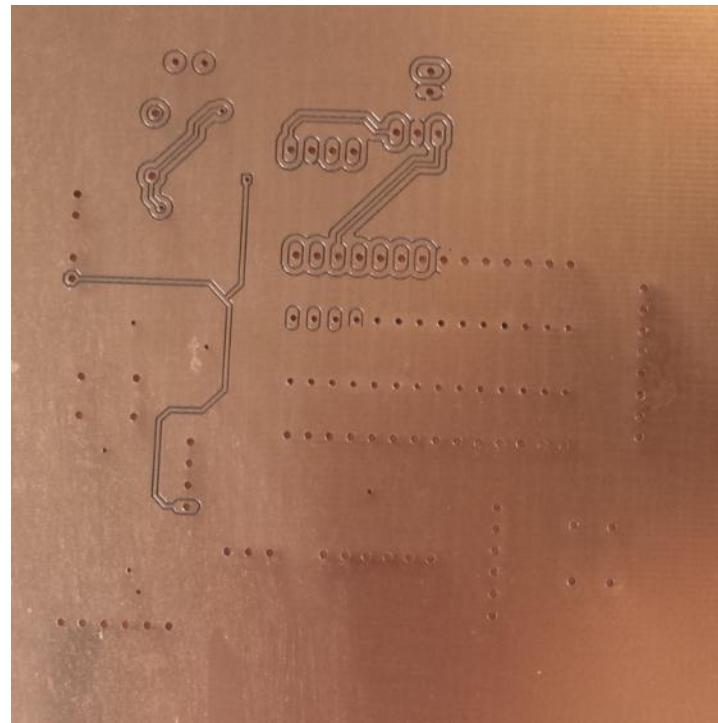


Fig. 7.40 Capa BOTTOM, dibujando las pistas

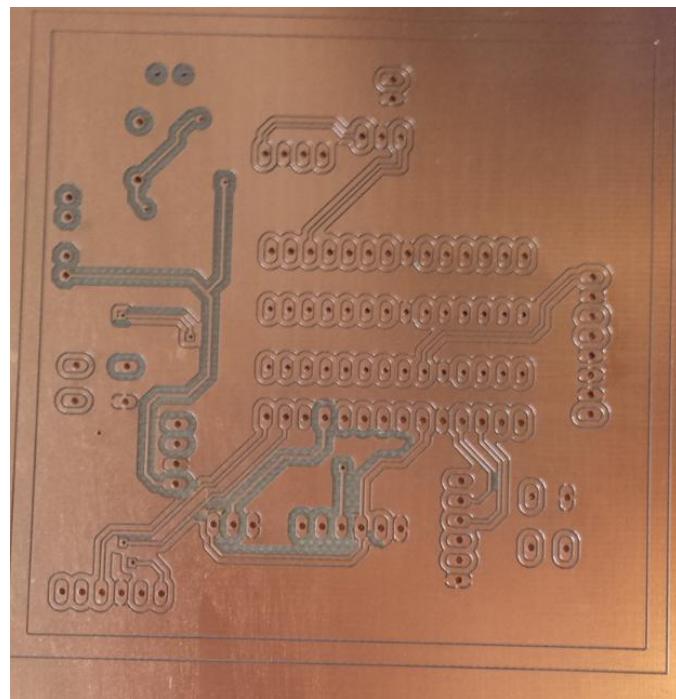


Fig. 7.41 Capa BOTTOM, realizando el *Clearance* de las pistas

7.4 Datos capturados durante 2h30min

La funcionalidad de Labview de poder de exportar los datos a excel nos permite trabajar cómodamente con grandes cantidades de datos y, a su vez, nos puede ayudar a realizar estudios de comportamientos y patrones del medio ambiente.

Hemos realizado una captura de datos de 17:00 a 19:30 en una zona donde al principio le daba la sombra y luego daba el sol a nuestro sistema sensor.

7.4.1 Iluminancia

En la **Fig. 6.4** podemos observar que en el trascurso de la primera hora nuestro sensor de luz le daba la sombra. Después empezó a recibir el sol de ocaso y su iluminancia recibida se iba reduciendo de forma suave.

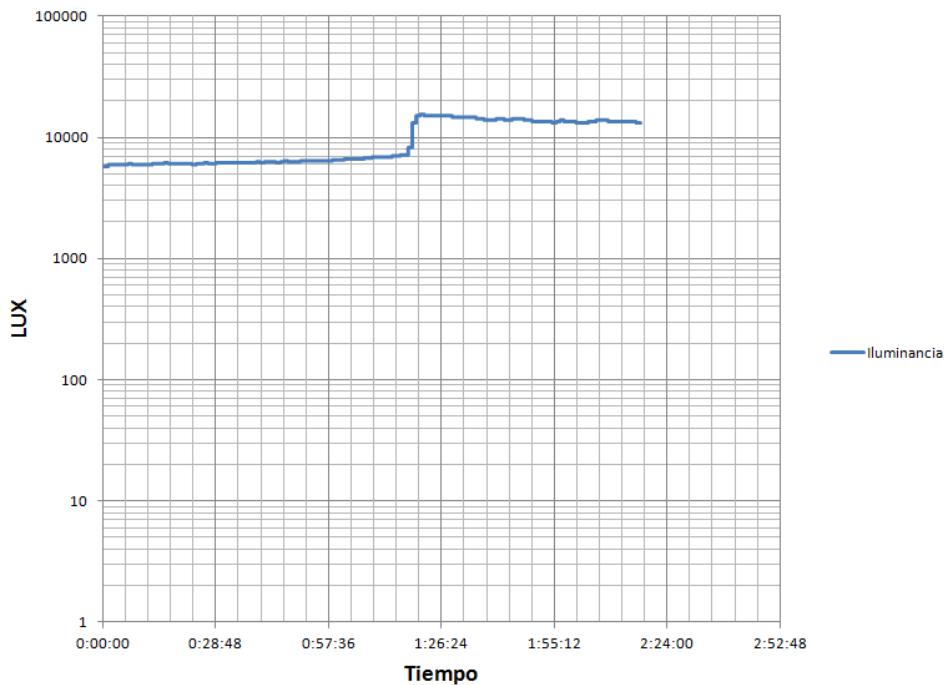


Fig. 6.4 Captura de la Iluminancia

7.4.2 Temperaturas

En la **Fig. 6.4** podemos ver claramente que el sol empezó a darle primero al KY-013 porque fue el primer sensor en capturar el aumento de temperatura.

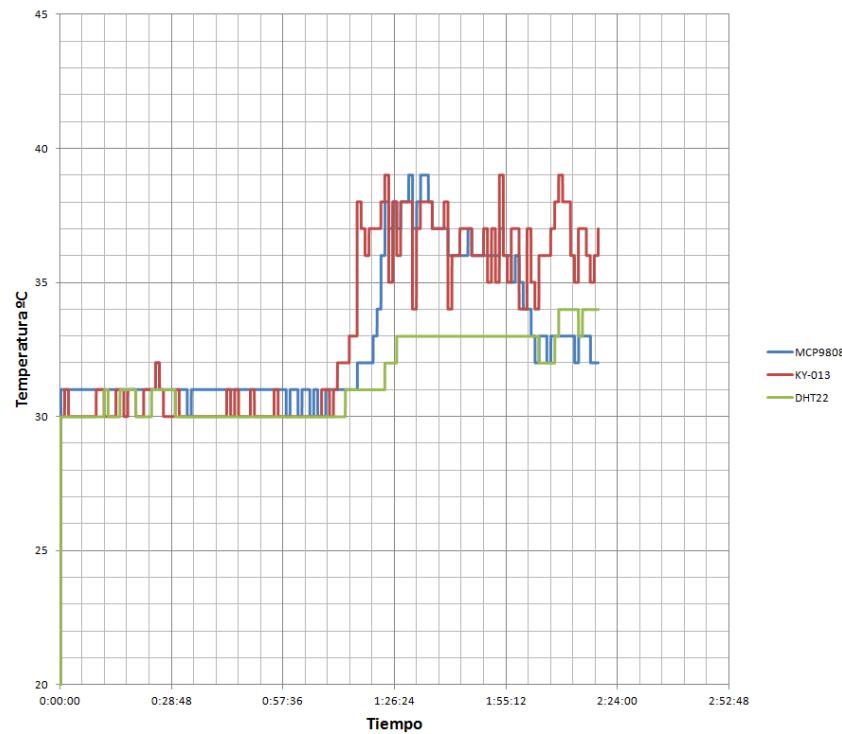


Fig. 6.4 Captura de la temperaturas

7.4.3 Humedad Relativa

La humedad relativa se mantiene un poco oscilante al principio y cuando empieza a dar el sol decrece levemente la humedad relativa como se puede ver en **Fig. 3.2**.

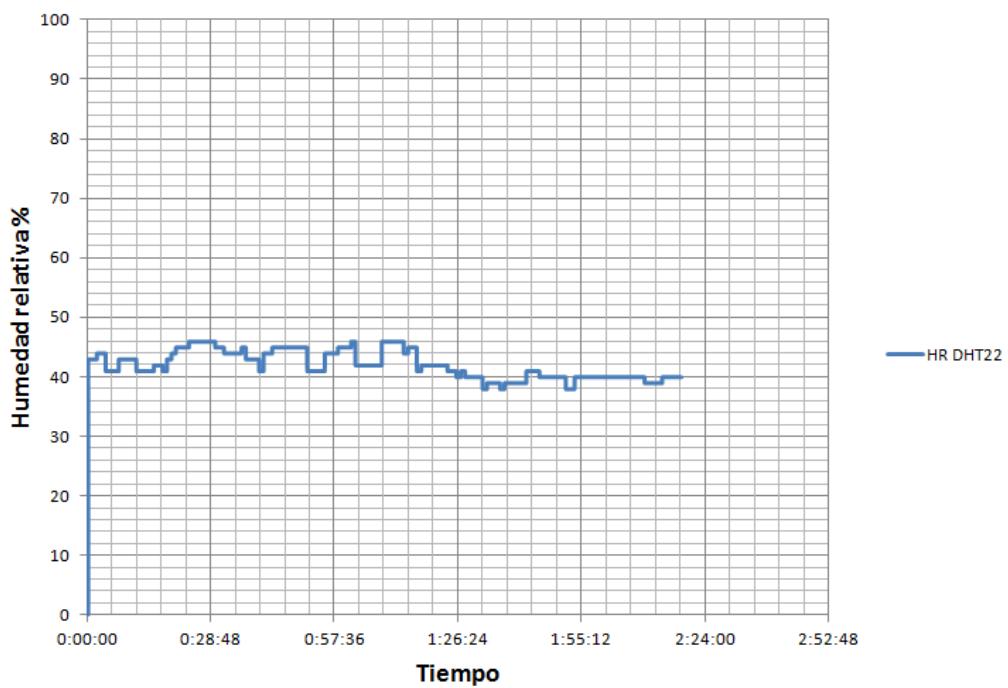
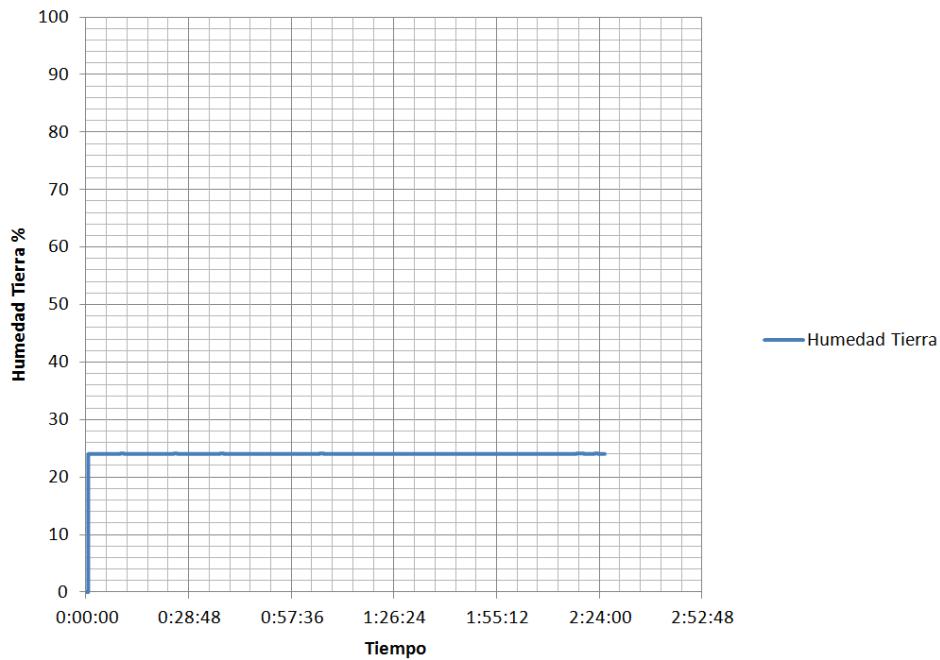


Fig. 6.5 Captura de la temperaturas

7.4.4 Humedad de la tierra

La humedad de la tierra se mantuvo constante durante toda la captura de datos. Se hubiera notado algún cambio si hubiéramos regado.



7.5 Código íntegro en C del programa principal

```
#include <p18F26K20.h>
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "I2C1.h"
#include <plib/usart.h>
#include <stdint.h>
#include "DHT22.h"
#include "ADC.h"
#include "init_system.h"
#include "USART.h"
#include "TIMER3.h"

//dfgdfgdfg
#define MCP9808
///#define MASTER 0x20                                // defines the slave
address of this controller
#define SLAVE 0x10                                    // defines the slave
address to be controlled by this controller
#define TIMEOUT 0xffff                                // defines the timeout
value, must not be too small
#define slave_address 0b10000000
#define Command_Tem 0b11110011
#define Command_Hum 0b11110101

#endif MCP9808
#define Command_TemMCP 0x05
#define AddressByteMCP 0b00110000

#else
#define AddressByteMCP 0b10000000
#define Command_TemMCP 0b11110101
#endif

#define STATE_Idle      'A'
#define STATE_SETUP_ADC_GH    'B'
#define STATE_AT_ADC_GH     'C'
#define STATE_STOP_ADC_GH   'D'
#define STATE_SETUP_ADC_ATEMP 'E'
#define STATE_AT_ADC_ATEMP  'F'
#define STATE_STOP_ADC_ATEMP 'G'
#define STATE_SETUP_ONEWIRE  'H'
#define STATE_AT_ONEWIRE    'I'
#define STATE_STOP_ONEWIRE   'J'
#define STATE_SETUP_I2C      'L'
#define STATE_AT_I2C         'M'
#define STATE_STOP_I2C       'N'
```

```
#define STATE_SETUP_ADC_LDR      'P'  
#define STATE_AT_ADC_LDR        'Q'  
#define STATE_STOP_ADC_LDR 'O'  
  
int SerialInterrupt_flag=0;  
int Trigger = 0;  
unsigned int postscalermax = 1000; ////////////// el led se enciende y se apaga  
cada un segundo  
unsigned int postscaler30smax=30000;  
unsigned int postscalerflag30s=0;  
unsigned int postscaler30s = 0;  
unsigned int postscaler = 0;  
unsigned int postscalerflag = 0;  
unsigned int postscalersampleflag=0;  
unsigned int postscalersample=0;  
unsigned int postscalersamplemax=249;  
  
static unsigned int Value_AN0 = 0; //potenciómetro  
static unsigned int Value_AN1 = 0; // sensor de temperatura  
static unsigned int Value_AN9 = 0; // sensor LDR  
  
unsigned int Valor_Medio_HumedadTierra[4];  
float Valor_Medio_TempAnalog[4];  
unsigned int Valor_Medio_HT=0;  
float Valor_Medio_TA=0;  
unsigned int samples_num = 0;  
  
char dataI2C_Hum;  
char dataI2C_Tem;  
int32_t I2C_Tem = 0;  
int32_t I2C_Hum = 0;  
float I2C_TemFormula = 0.0;  
float I2C_HumFormula = 0.0;  
unsigned int I2C_dev=0;  
unsigned int I2C_manu = 0;  
  
static float Vdd = 3.335; //3.5V  
static float Vo = 0.505; //0°C 0.5V  
static float sensibilidad = 0.01; //10mv/°C  
float Value_temp = 0.0;  
float Value_pot = 0.0;  
  
char CharRX;  
char CharTX[] = "Hello World";
```

```
char CharTX2[] = "Hello World";  
  
void __interrupt IntServe();  
char output_logic(void);  
char state_logic(void);  
static char present_state = STATE_Idle; // state variable  
  
void delay_ms(unsigned long ms);  
  
int main(void) {  
    init_system();  
    __delay_ms(49);  
    while (1) {  
        //SendStringUSART("HOLA");  
        output_logic();  
        state_logic();  
        /*  
         * if (SerialInterrupt_flag == 1) {  
         *     //CharRX = ReadUSART();  
  
         *     if (CharRX == 'A') {  
         *         SendStringUSART("GO IDLE");  
         *         present_state = STATE_Idle;  
         *         T3CON = 0;  
         *         Trigger = 0;  
         *     }  
         *     SerialInterrupt_flag = 0;  
         */  
        }  
    }  
  
    char state_logic(void) {  
        char error = 0;  
  
        switch (present_state) {  
            case STATE_Idle:  
                //IFS0bits.INT0IF = 1;  
                if (Trigger == 1) {  
                    // if a button pressed is detected,  
                    present_state = STATE_SETUP_ADC_GH;  
                    Trigger = 0;  
                    // IFS0bits.INT0IF = 0;  
                }  
        }  
    }  
}
```

```
    } else {
        present_state = STATE_Idle;
    }
    //present_state = Setup_Pot;
    break;
case STATE_SETUP_ADC_GH:
    present_state = STATE_AT_ADC_GH;

    break;
case STATE_AT_ADC_GH:
    if (Trigger == 1) {
        // if a button pressed is detected,
        present_state = STATE_STOP_ADC_GH;
        Trigger = 0;
    } else {
        present_state = STATE_AT_ADC_GH;
    }
    break;

case STATE_STOP_ADC_GH:
    present_state = STATE_SETUP_ADC_ATEMP;

    break;

case STATE_SETUP_ADC_ATEMP:
    present_state = STATE_AT_ADC_ATEMP;
    break;

case STATE_AT_ADC_ATEMP:
    if (Trigger == 1) {
        // if a button pressed is detected,
        present_state = STATE_STOP_ADC_ATEMP;

        Trigger = 0;
        // IFS0bits.INT0IF = 0;
    } else {
        present_state = STATE_AT_ADC_ATEMP;
    }

    //present_state = STATE_STOP_ADC_ATEMP;
    break;

case STATE_STOP_ADC_ATEMP:
    present_state = STATE_SETUP_ONEWIRE;
    break;
```

```
case STATE_SETUP_ONEWIRE:  
    present_state = STATE_AT_ONEWIRE;  
    break;  
  
case STATE_AT_ONEWIRE:  
    if (Trigger == 1) {  
  
        // if a button pressed is detected,  
        present_state = STATE_STOP_ONEWIRE;  
        Trigger = 0;  
        // IFS0bits.INT0IF = 0;  
    } else {  
        present_state = STATE_AT_ONEWIRE;  
    }  
  
    break;  
  
case STATE_STOP_ONEWIRE:  
    present_state = STATE_SETUP_I2C;  
  
    break;  
case STATE_SETUP_I2C:  
    present_state = STATE_AT_I2C;  
    break;  
  
case STATE_AT_I2C:  
    if (Trigger == 1) {  
        // if a button pressed is detected,  
        present_state = STATE_STOP_I2C;  
        Trigger = 0;  
        // IFS0bits.INT0IF = 0;  
    } else {  
        present_state = STATE_AT_I2C;  
    }  
    break;  
  
case STATE_STOP_I2C:  
    present_state = STATE_SETUP_ADC_LDR;  
    break;  
  
case STATE_SETUP_ADC_LDR:  
    present_state = STATE_AT_ADC_LDR;  
    break;  
  
case STATE_AT_ADC_LDR:  
    if (Trigger == 1) {  
        // if a button pressed is detected,  
        present_state = STATE_STOP_ADC_LDR;  
        Trigger = 0;
```

```
// IFS0bits.INT0IF = 0;
} else {
    present_state = STATE_AT_ADC_LDR;
}
break;

case STATE_STOP_ADC_LDR:
    present_state = STATE_Idle;
    break;

default:
    error = 1;
}
return (error);
}

char output_logic(void) {
    char error = 0;
    switch (present_state) {
        case STATE_Idle:
            PORTC = 0b00000111; // Reset all Flip-Flops at PORTB

            Sleep();
            CLRWDT();
            Trigger = 1;

            break;

        case STATE_SETUP_ADC_GH:
            T3CON = 0;
            ADON = 0;
            CloseI2C();

            InitializeTimer3(); //interrupt every second
            InitADC();

            postscaler = 0;
            postscalerflag = 0;

            SendStringUSART("Analog Mode Acquiring");

            PORTC = 0b00000101; // Reset all Flip-Flops at PORTB
            SelectChannelADC(0); //select AN0
            samples_num = 0;
            Valor_Medio_HT=0;
            break;
        case STATE_AT_ADC_GH:

            if (postscalersampleflag == 1) {
```

```
Value_AN0=0;
Value_AN0 = GetSampleADC();
Valor_Medio_HumedadTierra[samples_num]= Value_AN0;

postscalersampleflag = 0;
samples_num++;

}
if(samples_num == 3){
    samples_num=0;
    while(samples_num < 4){

        Valor_Medio_HT      =      Valor_Medio_HT      +
Valor_Medio_HumedadTierra[samples_num];

        samples_num++;
    }
    Valor_Medio_HT = Valor_Medio_HT/4;
    Valor_Medio_HT = 1013 - Valor_Medio_HT;
    sprintf(CharTX, "DATA ANALOG :Humedad TIERRA = %d",Valor_Medio_HT);
    SendStringUSART(CharTX);
    Trigger =1;

}

break;
case STATE_STOP_ADC_GH:
    postscalersampleflag = 0;
    break;

case STATE_SETUP_ADC_ATEMP:
    SendStringUSART("I2C Mode Acquiring");
    PORTC = 0b00000100; // Reset all Flip-Flops at PORTB

    SelectChannelADC(1);//select AN1

    samples_num=0;
    Valor_Medio_TA =0;
    break;

case STATE_AT_ADC_ATEMP:
    if (postscalersampleflag == 1) {
        Value_AN1=0;
```

```
Value_AN1 = GetSampleADC();
///Value_temp = (float) (((Value_AN1 / 1023.0) * Vdd - Vo)) /
sensibilidad; // Sensor Placa XLP
Value_temp = (float)(log(10000.0 * ((1024.0 / Value_AN1 - 1))));
Value_temp = (float)(1 / (0.001129148 + (0.000234125 +
(0.0000000876741 * Value_temp * Value_temp)) * Value_temp));
Value_temp = (float)(Value_temp - 273.15); // Convert Kelvin
to Celcius
//Temp = (Temp * 9.0)/ 5.0 + 32.0; // Convert Celcius to Fahrenheit
Valor_Medio_TempAnalog[samples_num]=Value_temp;

if(Value_pot < 50){
    PORTBbits.RB5 =1;
}
else{
    PORTBbits.RB5 =0;
}

samples_num++;
postscalersampleflag = 0;

}

if(samples_num == 4){
    samples_num=0;
    while(samples_num < 4){

        Valor_Medio_TA = Valor_Medio_TA +
Valor_Medio_TempAnalog[samples_num];

        samples_num++;
    }
    Valor_Medio_TA = Valor_Medio_TA/4.0;
    sprintf(CharTX, "DATA ANALOG :Temperature = %.1f
C",Valor_Medio_TA);
    SendStringUSART(CharTX);
    Trigger =1;

}
```

```
break;

case STATE_STOP_ADC_ATEMP:
    postscalersampleflag = 0;
    break;
case STATE_SETUP_ONEWIRE:
    PORTC = 0b00000111; // Reset all Flip-Flops at PORTB
    TRISBbits.TRISB4 = 0;
    PORTBbits.RB4 = 1;
    TMR2IE = 1;      // Enable Timer2 interrupt
    T2CON = 0;       // Prescaler 1:1, and Timer2 is off initially
    TMR2IF = 0;      // Clear TMR INT Flag bit
    TMR2 = 0;
    T2CONbits.T2CKPS = 0b01;
    break;

case STATE_AT_ONEWIRE:
    GetDataSensorDHT22();

    if (CheckSum == ((RH_Byte1 + RH_Byte2 + T_Byte1 + T_Byte2) &
0xFF)) {
        if (sign) {
            sprintf(CharTX, "DATA DHT22 :Temperature = -%.1f C",
Value_tempdht22);

            SendStringUSART(CharTX);
        } else {
            sprintf(CharTX, "DATA DHT22 :Temperature = %.1f C",
Value_tempdht22);
            SendStringUSART(CharTX);
        }

        sprintf(CharTX, "DATA DHT22 :Humidity = %.1f %%",
Value_rhdht22);
        SendStringUSART(CharTX);

    } else {
        SendStringUSART("Checksum Error! Trying Again ... ");
    }
    Trigger = 1;

    break;

case STATE_STOP_ONEWIRE:
    T2CON = 0;
    break;

case STATE_SETUP_I2C:
```

```
Open_I2C1();
break;

case STATE_AT_I2C:

if (postscalersampleflag == 1) {
    // I2C_dev = WriteDEVICE_I2C(AddressByteMCP, 0x07);
    // I2C_manu = WriteMANU_I2C(AddressByteMCP, 0x06);

    I2C_TemFormula      =      ReadByte_I2C(AddressByteMCP,
Command_TemMCP); //temperature

    sprintf(CharTX2, "DATA I2C :Temperature = %.1f C",
I2C_TemFormula);

    SendStringUSART(CharTX2);

    postscalersampleflag = 0;
    Trigger = 1;
}

break;

case STATE_STOP_I2C:

SendStringUSART("LDR mode ON");
CloseI2C();
break;

case STATE_SETUP_ADC_LDR:
//InitializeTimer3(); //interrupt every second
SelectChannelADC(9);//select AN9

break;

case STATE_AT_ADC_LDR:

if (postscalersampleflag == 1) {
    Value_AN9 = 0;

    Value_AN9 = GetSampleADC();
    //Value_AN9 = 1013 - Value_AN9;
    float alpha = -0.6;
    float resistance = 0.0;
    resistance = (float)Value_AN9*1000.0/1023.0;
    resistance = resistance/(1.0-(float)Value_AN9/1023.0);
}
```

```
float lux = 0.0;
sprintf(CharTX, "DATA LDR :RESISTENCIA = %ld", (long)resistance);
SendStringUSART(CharTX);
lux = (log10(resistance)-log10(21000.0))/alpha;
sprintf(CharTX, "DATA LDR :LOG = %ld", (long)lux);
SendStringUSART(CharTX);
lux = pow(10,lux);
sprintf(CharTX, "DATA LDR :Valor Analog = %ld", (long)lux);

SendStringUSART(CharTX);

postscalersampleflag = 0;
Trigger = 1;
}

break;

case STATE_STOP_ADC_LDR:
T3CON = 0;
StopADC();
SendStringUSART("Sleep 30s");
SendStringUSART("Idle state");
__delay_ms(49);
break;

default:
error = 1;
}

return (error);
}

void __interrupt IntServe(void) {

GIE = 0; //Disable interrupts while attending one of them ...

if (RCIF == 1) {
CharRX = ReadUSART();
while (BusyUSART());
//Trigger == 1;

#ifndef USARTWAKEUP
WUE = 1;
#endif
RCIF = 0;
}

if (INT0IF == 1) {
```

```
Trigger = 1;
INT0IF = 0;
}
if (TMR3IF == 1) {
    TMR3H = 0xF8;          // TMR3H must be written first
    TMR3L = 0x46;          // count up from F830 but there are a small
overhead and TMR3L must be 0x46

    if ((postscalersample      <      postscalersamplemax)      &&
(present_state!=STATE_Idle)) {
        postscalersample++;
    } else if((present_state!=STATE_Idle)){
        postscalersample = 0;
        postscalersampleflag = 1;
    }
/*
if (postscaler30s < postscaler30smax) {
    postscaler30s++;
} else {

    postscaler30s=0;
    Trigger = 1;
}*/
TMR3IF = 0;

}

if (TMR2IF == 1) {
    //Trigger = 1;
    if (present_state == STATE_AT_ONEWIRE) {
        TOUT = 1;
        TMR2ON = 0; // stop timer
        TMR2IF = 0; // Clear TMR0 interrupt flag
    }

    TMR2IF = 0;
}
GIE = 1; //Disable interrupts while attending one of them ...
}
```