



Notas del código

```
#include <Wire.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define ONE_WIRE_BUS
```

#include <Wire.h>

- **Significado:** Esta línea incluye la biblioteca Wire, que proporciona comunicación I2C (Inter-Integrated Circuit) para el Arduino. I2C es un protocolo que permite la comunicación entre un microcontrolador (en este caso, Arduino) y periféricos como sensores y expansores de E/S.
- **Uso:** Se utiliza para gestionar la comunicación I2C con el módulo HW-171, que controla el display de siete segmentos en este proyecto.

#include <OneWire.h>

- **Significado:** Esta línea incluye la biblioteca OneWire, que permite la comunicación con dispositivos que usan el protocolo de comunicación 1-Wire, desarrollado por Dallas Semiconductor.
- **Uso:** Específicamente, se utiliza para interactuar con el sensor de temperatura DS18B20, que está conectado al Arduino a través de un solo pin de datos.

#include <DallasTemperature.h>

- **Significado:** Esta línea incluye la biblioteca DallasTemperature, que está construida sobre la biblioteca OneWire y proporciona funciones adicionales para interactuar con los sensores de temperatura DS18B20 de manera más fácil.

- **Uso:** Facilita la lectura de temperaturas de los sensores DS18B20.

```
#define ONE_WIRE_BUS 2
```

- **Significado:** Esta línea define una constante llamada `ONE_WIRE_BUS` y le asigna el valor 2.
- **Uso:** Especifica que el sensor DS18B20 está conectado al pin digital 2 del Arduino. Este valor se usa al inicializar la comunicación OneWire.

Información adicional relevante:

- **Wire Library:** La biblioteca Wire maneja la transmisión y recepción de datos en el bus I2C, utilizando las funciones `Wire.begin()`, `Wire.beginTransmission()`, `Wire.write()`, y `Wire.endTransmission()`.
- **OneWire Library:** Permite la comunicación con dispositivos 1-Wire a través de un solo pin digital. Dispositivos como el DS18B20 usan este protocolo.
- **DallasTemperature Library:** Proporciona métodos específicos para sensores de temperatura DS18B20, como `sensors.begin()`, `sensors.requestTemperatures()`, y `sensors.getTempCByIndex(0)`.

```
const float BETA = 3950;

int analogValue;
float average_temp;

const int analogPinPotenciometer = A1;

int valPotenciometer;
int interval;
```

```
const float BETA = 3950;
```

- **Significado:** Declara una constante de tipo `float` llamada `BETA` y le asigna el valor 3950.

- **Uso:** Esta constante se utiliza en el cálculo de la temperatura a partir del sensor NTC (Negative Temperature Coefficient) utilizando la ecuación de Steinhart-Hart. El valor BETA es una característica del sensor NTC que describe su comportamiento térmico.

int analogValue;

- **Significado:** Declara una variable de tipo `int` llamada `analogValue`.
- **Uso:** Almacena el valor leído del pin analógico conectado al sensor NTC. Este valor representa la resistencia del sensor que varía con la temperatura.

float average_temp;

- **Significado:** Declara una variable de tipo `float` llamada `average_temp`.
- **Uso:** Almacena la temperatura promedio calculada a partir de los valores de los dos sensores de temperatura (NTC y DS18B20).

const int analogPinPotenciometer = A1;

- **Significado:** Declara una constante de tipo `int` llamada `analogPinPotenciometer` y le asigna el valor `A1`.
- **Uso:** Especifica que el potenciómetro está conectado al pin analógico A1 del Arduino. Este valor se utiliza para leer la posición del potenciómetro, que ajusta el intervalo de tiempo entre mediciones.

int valPotenciometer;

- **Significado:** Declara una variable de tipo `int` llamada `valPotenciometer`.
- **Uso:** Almacena el valor leído del potenciómetro. Este valor se usa luego para calcular el intervalo de tiempo entre mediciones.

int interval;

- **Significado:** Declara una variable de tipo `int` llamada `interval`.
- **Uso:** Almacena el intervalo de tiempo calculado (en milisegundos) entre mediciones de temperatura. Este intervalo se determina a partir del valor del potenciómetro y se utiliza para pausar el bucle `loop()`.

Información adicional relevante:

- **BETA Value:** En la ecuación de Steinhart-Hart, el valor BETA es una constante que depende del material del sensor NTC y es esencial para convertir la resistencia medida a una temperatura en grados Celsius.
- **Analog Input:** Los pines analógicos del Arduino (`A0` , `A1` , etc.) permiten leer voltajes entre 0 y 5V, que se convierten en valores digitales entre 0 y 1023.
- **Potentiometer:** Un potenciómetro es un divisor de voltaje variable. Al girar el potenciómetro, se cambia el voltaje en el pin analógico, permitiendo al usuario ajustar dinámicamente el intervalo de tiempo entre mediciones.

```
OneWire oneWire(ONE_WIRE_BUS);  
DallasTemperature sensors(&oneWire);
```

`OneWire oneWire(ONE_WIRE_BUS);`

- **Significado:** Crea una instancia del objeto `OneWire` llamada `oneWire` y la inicializa con el pin especificado por `ONE_WIRE_BUS` , que en este caso es el pin 2.
- **Uso:** La biblioteca OneWire permite la comunicación con dispositivos que usan el protocolo OneWire, como el sensor de temperatura DS18B20. Esta línea configura el pin 2 del Arduino para la comunicación OneWire.

`DallasTemperature sensors(&oneWire);`

- **Significado:** Crea una instancia del objeto `DallasTemperature` llamada `sensors` y la inicializa utilizando la instancia `oneWire` .
- **Uso:** La biblioteca DallasTemperature facilita la interacción con los sensores DS18B20, que utilizan el protocolo OneWire. Esta línea establece que la comunicación con los sensores de temperatura se realizará a través del objeto `oneWire` .

Información adicional relevante:

- **OneWire Library:** La biblioteca OneWire permite la comunicación con dispositivos OneWire mediante un único pin de datos. Este protocolo es útil para conectar múltiples dispositivos a un solo pin de microcontrolador.

- **DallasTemperature Library:** Esta biblioteca simplifica la interacción con los sensores de temperatura DS18B20, proporcionando funciones para leer la temperatura y manejar múltiples sensores en el mismo bus OneWire.
- **Instanciación de objetos:** En estas líneas de código, se crean e inicializan objetos necesarios para la comunicación con los sensores. La instancia `oneWire` es pasada como un argumento al constructor de `DallasTemperature`, lo que permite que la biblioteca DallasTemperature utilice la comunicación configurada por la biblioteca OneWire.

Uso en el programa:

- `oneWire` : Es el objeto que maneja la comunicación de bajo nivel con el bus OneWire.
- `sensors` : Es el objeto que proporciona métodos de alto nivel para interactuar con los sensores de temperatura DS18B20, como solicitar mediciones y leer valores de temperatura.

```
const int segmentUnitPins[7] = {3, 4, 5, 6, 7, 8, 9};

const byte unitsNumbers[10] = {
    B11111100, // 0
    B01100000, // 1
    B11011010, // 2
    B11110010, // 3
    B01100110, // 4
    B10110110, // 5
    B10111110, // 6
    B11100000, // 7
    B11111110, // 8
    B11110110  // 9
};
```

```
const int segmentUnitPins[7] = {3, 4, 5, 6, 7, 8, 9};
```

- **Significado:** Declara un arreglo constante de enteros llamado `segmentUnitPins` que contiene los números de los pines del Arduino que están conectados a los segmentos del display de siete segmentos.
- **Pines:** Los valores en el arreglo (3, 4, 5, 6, 7, 8, 9) corresponden a los pines del Arduino que controlan cada segmento del display.
 - **Segmentos:** Cada número en el arreglo representa un segmento del display (por lo general, los segmentos se denominan a, b, c, d, e, f, g en un display de siete segmentos).

```
const byte unitsNumbers[10] = { ... };
```

- **Significado:** Declara un arreglo constante de bytes llamado `unitsNumbers` que contiene los patrones binarios necesarios para representar los números del 0 al 9 en un display de siete segmentos.
- **Patrones binarios:** Cada valor en el arreglo es un byte que utiliza la notación binaria para representar los segmentos que deben estar encendidos para mostrar un dígito específico.
 - **Formato:** Los valores están en formato binario (`B11111100`), donde cada bit representa el estado (encendido o apagado) de un segmento del display.
 - **Bits:** Los bits en el byte (de izquierda a derecha) corresponden a los segmentos del display (a, b, c, d, e, f, g, dp), donde `1` significa encendido y `0` significa apagado. En este caso, el bit menos significativo (el último a la derecha) se ignora ya que no se utiliza el punto decimal (dp).

Desglose de los patrones binarios:

- **0 (B11111100):** Enciende todos los segmentos excepto el segmento 'g'.
- **1 (B01100000):** Enciende solo los segmentos 'b' y 'c'.
- **2 (B11011010):** Enciende todos los segmentos excepto 'b' y 'f'.
- **3 (B11110010):** Enciende todos los segmentos excepto 'e' y 'f'.
- **4 (B01100110):** Enciende los segmentos 'b', 'c', 'f', y 'g'.
- **5 (B10110110):** Enciende todos los segmentos excepto 'b' y 'e'.
- **6 (B10111110):** Enciende todos los segmentos excepto 'b'.

- **7 (B11100000):** Enciende solo los segmentos 'a', 'b', y 'c'.
- **8 (B11111110):** Enciende todos los segmentos.
- **9 (B11110110):** Enciende todos los segmentos excepto 'e'.

Información adicional relevante:

- **Display de siete segmentos:** Un componente electrónico utilizado para mostrar dígitos del 0 al 9. Tiene 7 LEDs (segmentos) dispuestos en una figura similar a un "8", más un punto decimal opcional.
- **Uso de los pines:** Los pines del Arduino especificados en `segmentUnitPins` se conectan a los segmentos del display. El programa enciende o apaga estos pines para mostrar los números correspondientes.

Uso en el programa:

- `segmentUnitPins`: Este arreglo se utiliza para establecer los pines del Arduino que controlarán cada segmento del display.
- `unitsNumbers`: Este arreglo se utiliza para definir los patrones de encendido y apagado de los segmentos del display, permitiendo mostrar los números del 0 al 9.

```
const int hw171Address = 0x20;
```

```
byte tensNumbers[10] = {
    0b00111111, // 0
    0b00000110, // 1
    0b01011011, // 2
    0b01001111, // 3
    0b01100110, // 4
    0b01101101, // 5
    0b01111101, // 6
    0b00000111, // 7
    0b01111111, // 8
}
```

```
    0b01101111  // 9
};
```

```
const int hw171Address = 0x20;
```

- **Significado:** Declara una constante entera llamada `hw171Address` que almacena la dirección I2C del módulo HW-171.
- **I2C:** I2C (Inter-Integrated Circuit) es un protocolo de comunicación que permite la conexión de múltiples dispositivos con solo dos cables (SDA y SCL). Cada dispositivo en el bus I2C tiene una dirección única.
- **Dirección:** `0x20` es la dirección en hexadecimal del módulo HW-171 en el bus I2C. Esta dirección se utiliza para comunicarse con el módulo desde el Arduino.

```
byte tensNumbers[10] = { ... };
```

- **Significado:** Declara un arreglo de bytes llamado `tensNumbers` que contiene los patrones binarios necesarios para representar los números del 0 al 9 en un display de siete segmentos.
- **Patrones binarios:** Cada valor en el arreglo es un byte que utiliza la notación binaria para representar los segmentos que deben estar encendidos para mostrar un dígito específico.
 - **Formato:** Los valores están en formato binario (`0b00111111`), donde cada bit representa el estado (encendido o apagado) de un segmento del display.
 - **Bits:** Los bits en el byte (de derecha a izquierda) corresponden a los segmentos del display (a, b, c, d, e, f, g), donde `1` significa encendido y `0` significa apagado.

Desglose de los patrones binarios:

- **0 (0b00111111):** Enciende todos los segmentos excepto el segmento 'g'.
- **1 (0b00000110):** Enciende solo los segmentos 'b' y 'c'.
- **2 (0b01011011):** Enciende todos los segmentos excepto 'b' y 'f'.
- **3 (0b01001111):** Enciende todos los segmentos excepto 'e' y 'f'.

- **4 (0b01100110):** Enciende los segmentos 'b', 'c', 'f', y 'g'.
- **5 (0b01101101):** Enciende todos los segmentos excepto 'b' y 'e'.
- **6 (0b01111101):** Enciende todos los segmentos excepto 'b'.
- **7 (0b00000111):** Enciende solo los segmentos 'a', 'b', y 'c'.
- **8 (0b01111111):** Enciende todos los segmentos.
- **9 (0b01101111):** Enciende todos los segmentos excepto 'e'.

Información adicional relevante:

- **Display de siete segmentos:** Un componente electrónico utilizado para mostrar dígitos del 0 al 9. Tiene 7 LEDs (segmentos) dispuestos en una figura similar a un "8".
- **HW-171:** Un módulo I2C que puede ser utilizado para controlar displays de siete segmentos o realizar otras tareas que requieran comunicación I2C.
- **Uso de los pines:** Los segmentos del display están controlados por el patrón binario correspondiente al número que se desea mostrar. Cada bit en el byte indica si un segmento específico debe estar encendido o apagado.

Uso en el programa:

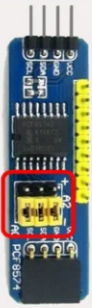
- `hw171Address` : Esta constante se utiliza en las funciones que manejan la comunicación I2C para especificar la dirección del módulo HW-171.
- `tensNumbers` : Este arreglo se utiliza para definir los patrones de encendido y apagado de los segmentos del display, permitiendo mostrar los números del 0 al 9.

[PCF8574 GPIO Extender - With Arduino and NodeMCU : 15 Steps - Instructables](#)

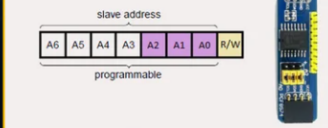
Step 6: Addressing

PCF8574 Addressing

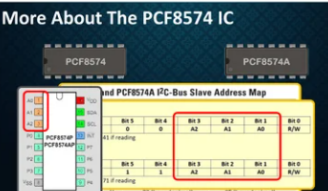
A0	A1	A2	Address Pins
0	0	0	= 0x20
0	0	1	= 0x21
0	1	0	= 0x22
0	1	1	= 0x23
1	0	0	= 0x24
1	0	1	= 0x25
1	1	0	= 0x26
1	1	1	= 0x27



PCF8574 Addressing



More About The PCF8574 IC



By connecting the three address bits A0, A1 and A2 to VIN or HIGH you can get different combination of the addresses.

This is how an address byte of the PCF8574 looks like. First 7-bits combine to form the slave address. The last bit of the slave address defines the operation (read or write) to be performed. When it is high (1), a read is selected, while a low (0) selects a write operation.

```
void setup() {
  Serial.begin(9600);
  sensors.begin();

  for (int i = 0; i < 7; i++) {
    pinMode(segmentUnitPins[i], OUTPUT);
  }

  Wire.begin();
}
```

La función `setup()` en un programa de Arduino se ejecuta una vez cuando se inicia el microcontrolador. Esta función se utiliza para inicializar variables, pines y bibliotecas. En este código, `setup()` realiza las siguientes acciones:

1. `Serial.begin(9600);`

- **Función:** Inicializa la comunicación serie a una velocidad de 9600 baudios.
 - **Propósito:** Permite la comunicación entre el Arduino y el ordenador para enviar y recibir datos a través del puerto serie. Esto es útil para depuración y monitoreo de datos.
2. `sensors.begin();`
- **Función:** Inicializa la biblioteca `DallasTemperature`, que se utiliza para comunicarse con sensores de temperatura DS18B20.
 - **Propósito:** Configura el sensor DS18B20 para que esté listo para medir la temperatura.
3. `for (int i = 0; i < 7; i++) {}`
- **Función:** Un bucle `for` que itera desde 0 hasta 6.
 - **Propósito:** Este bucle se utiliza para establecer los pines del 3 al 9 como salidas.
4. `pinMode(segmentUnitPins[i], OUTPUT);`
- **Función:** Configura cada pin en `segmentUnitPins` como una salida.
 - **Propósito:** Permite que el Arduino controle los segmentos del display de siete segmentos. Los pines especificados en `segmentUnitPins` (3, 4, 5, 6, 7, 8 y 9) serán utilizados para enviar señales a los segmentos del display.
5. `Wire.begin();`
- **Función:** Inicia la biblioteca Wire, que se utiliza para la comunicación I2C.
 - **Propósito:** Prepara al Arduino para comunicarse con dispositivos I2C, como el módulo HW-171 que se menciona en el código.

Propósito general del `setup()` :

- **Inicialización de la comunicación serie:** Para monitorear datos en el puerto serie.
- **Configuración del sensor de temperatura DS18B20:** Preparar el sensor para medir la temperatura.

- **Configuración de los pines de salida:** Preparar los pines para controlar un display de siete segmentos.
- **Inicialización de la comunicación I2C:** Preparar la comunicación con dispositivos I2C.

Relevancia:

La función `setup()` es esencial para preparar el entorno del Arduino antes de que comience a ejecutar el código en el `loop()`. Al establecer correctamente las configuraciones necesarias (comunicaciones, modos de pines, etc.), aseguramos que el dispositivo funcione según lo esperado durante la ejecución continua del `loop()`.

```
void loop() {  
  
    interval = potentiometer_interval();  
    Serial.print("Intervalo de tiempo entre mediciones: ");  
    Serial.print(interval);  
    Serial.println(" ms");  
  
    float ntc_temp = ntc_sensor();  
    float ds18b20_temp = ds18b20_sensor();  
  
    average_temp = average_temperature(ntc_temp, ds18b20_temp);  
  
    Serial.print("Temperatura: ");  
    Serial.print(average_temp);  
    Serial.println(" °C");  
  
    unitDisplayNumber(getUnits(average_temp));  
    tensDisplayNumber(getTens(average_temp));  
  
    delay(interval);  
}
```

La función `loop()` en un programa de Arduino se ejecuta repetidamente después de que se ha ejecutado `setup()`. En este código, `loop()` realiza varias tareas en cada ciclo:

1. `interval = potentiometer_interval();`

- **Función:** Llama a la función `potentiometer_interval()` y almacena el valor de retorno en `interval`.
- **Propósito:** Determina el intervalo de tiempo (en milisegundos) entre cada medición de temperatura basado en la posición del potenciómetro.

2. `Serial.print("Intervalo de tiempo entre mediciones: ");`

- **Función:** Envía una cadena de texto al monitor serial.
- **Propósito:** Imprime un mensaje descriptivo en el monitor serial.

3. `Serial.print(interval);`

- **Función:** Envía el valor de `interval` al monitor serial.
- **Propósito:** Imprime el intervalo de tiempo actual en milisegundos.

4. `Serial.println(" ms");`

- **Función:** Envía la cadena "ms" al monitor serial y añade un salto de línea.
- **Propósito:** Indica que el valor de `interval` está en milisegundos.

5. `float ntc_temp = ntc_sensor();`

- **Función:** Llama a la función `ntc_sensor()` y almacena el valor de retorno en `ntc_temp`.
- **Propósito:** Mide la temperatura utilizando el sensor NTC y guarda el resultado.

6. `float ds18b20_temp = ds18b20_sensor();`

- **Función:** Llama a la función `ds18b20_sensor()` y almacena el valor de retorno en `ds18b20_temp`.
- **Propósito:** Mide la temperatura utilizando el sensor DS18B20 y guarda el resultado.

7. `average_temp = average_temperature(ntc_temp, ds18b20_temp);`
 - **Función:** Llama a la función `average_temperature()` con `ntc_temp` y `ds18b20_temp` como argumentos, y almacena el valor de retorno en `average_temp`.
 - **Propósito:** Calcula la temperatura promedio de los dos sensores y guarda el resultado.
8. `Serial.print("Temperatura: ");`
 - **Función:** Envía la cadena "Temperatura: " al monitor serial.
 - **Propósito:** Imprime un mensaje descriptivo en el monitor serial.
9. `Serial.print(average_temp);`
 - **Función:** Envía el valor de `average_temp` al monitor serial.
 - **Propósito:** Imprime la temperatura promedio actual.
10. `Serial.println(" °C");`
 - **Función:** Envía la cadena " °C" al monitor serial y añade un salto de línea.
 - **Propósito:** Indica que el valor de `average_temp` está en grados Celsius.
11. `unitDisplayNumber(getUnits(average_temp));`
 - **Función:** Llama a la función `getUnits()` con `average_temp` como argumento y pasa el valor de retorno a `unitDisplayNumber()`.
 - **Propósito:** Extrae las unidades de la temperatura promedio y las muestra en un display de siete segmentos.
12. `tensDisplayNumber(getTens(average_temp));`
 - **Función:** Llama a la función `getTens()` con `average_temp` como argumento y pasa el valor de retorno a `tensDisplayNumber()`.
 - **Propósito:** Extrae las decenas de la temperatura promedio y las muestra en un display de siete segmentos utilizando el módulo I2C HW-171.
13. `delay(interval);`
 - **Función:** Pausa la ejecución del programa durante el tiempo especificado por `interval` en milisegundos.

- **Propósito:** Controla el tiempo entre cada ciclo de medición y actualización de la temperatura.

Propósito general del `loop()`:

- **Lectura del potenciómetro:** Ajusta el intervalo de tiempo entre mediciones basado en la posición del potenciómetro.
- **Medición de temperatura:** Obtiene las temperaturas de dos sensores diferentes y calcula su promedio.
- **Visualización:** Muestra la temperatura promedio en el monitor serial y en un display de siete segmentos.
- **Control de tiempo:** Pausa el programa para que las mediciones y actualizaciones se realicen a intervalos regulares definidos por el potenciómetro.

Relevancia:

El `loop()` es el núcleo del programa de Arduino, ejecutando continuamente las tareas necesarias para medir, calcular y mostrar la temperatura promedio, proporcionando una actualización regular y controlada por el usuario.

```
float ntc_sensor() {  
  
    float celsius_of_ntc;  
    analogValue = analogRead(A0);  
    celsius_of_ntc = 1 / (log(1 / (1023. / analogValue - 1)) / I  
  
    return celsius_of_ntc;  
}
```

La función `ntc_sensor()` mide la temperatura usando un sensor NTC (Negative Temperature Coefficient) conectado a la entrada analógica A0 del Arduino. Luego, convierte esta lectura a grados Celsius utilizando una fórmula específica basada en el comportamiento del sensor NTC.

1. Declaración de la función `float ntc_sensor() {`

- Define una función que retorna un valor `float` que representa la temperatura medida por el sensor NTC.

2. Declaración de la variable `float celsius_of_ntc :`

- Declara una variable `float` para almacenar la temperatura calculada en grados Celsius.

3. Lectura del valor analógico del sensor NTC `analogValue = analogRead(A0);`

- Lee el valor analógico del pin A0 donde está conectado el sensor NTC. El valor leído está en el rango de 0 a 1023, que corresponde a una señal de voltaje entre 0 y 5V.

4. Conversión de la lectura analógica a grados Celsius:

```
celsius_of_ntc = 1 / (log(1 / (1023. / analogValue - 1)) / BETA + 1.0 / 298.15) - 273.15;
```

- **Propósito:** Convierte la lectura analógica a una temperatura en grados Celsius utilizando la fórmula derivada de la ecuación de Steinhart-Hart para termistores NTC.
- **Detalles de la fórmula:**
 - `1023. / analogValue - 1` : Convierte la lectura analógica a una resistencia proporcional.
 - `log(1 / (1023. / analogValue - 1))` : Aplica el logaritmo natural a la inversa de la resistencia proporcional.
 - `/ BETA` : Divide el logaritmo por la constante BETA del termistor, que es una propiedad específica del material del termistor (3950 en este caso).
 - `+ 1.0 / 298.15` : Ajusta el resultado basado en la temperatura en Kelvin (298.15 K es 25 °C).
 - `1 / ...` : Toma el inverso del resultado para completar la fórmula de conversión a Kelvin.
 - `- 273.15` : Convierte de Kelvin a grados Celsius.

5. Retorno de la temperatura calculada `return celsius_of_ntc;`

- Devuelve la temperatura calculada en grados Celsius.

Relevancia:

La función `ntc_sensor()` es fundamental para el proyecto, ya que proporciona la medida de temperatura desde el sensor NTC. Esta medida es una de las dos entradas utilizadas para calcular la temperatura promedio que se muestra y monitorea en el sistema.

Resumen de la fórmula de conversión:

La ecuación utilizada se basa en la ecuación de Steinhart-Hart simplificada para termistores NTC, que permite convertir una resistencia proporcional (derivada del valor analógico leído) a una temperatura en grados Celsius. La constante BETA (3950) y el ajuste a Kelvin aseguran que la conversión sea precisa para el sensor específico utilizado.

```
float ds18b20_sensor() {  
  
    sensors.requestTemperatures();  
  
    float celsius_of_ds18b20 = sensors.getTempCByIndex(0);  
  
    return celsius_of_ds18b20;  
}
```

La función `ds18b20_sensor()` mide la temperatura utilizando un sensor digital DS18B20 conectado a través de un bus OneWire. El DS18B20 es un sensor de temperatura digital que comunica los datos de temperatura directamente sin necesidad de conversión analógica-digital.

1. Declaración de la función `float ds18b20_sensor() {`

- Define una función que retorna un valor `float` representando la temperatura medida por el sensor DS18B20.

2. Solicitud de temperatura al sensor `sensors.requestTemperatures();`

- **Propósito:** Envía una solicitud al sensor DS18B20 para que mida la temperatura. La biblioteca `DallasTemperature` maneja la comunicación con el sensor a través del bus OneWire.
- **Detalles:** Esta función inicia la lectura de temperatura de todos los sensores DS18B20 conectados al bus. Se debe esperar un breve periodo para que los sensores completen la medición antes de leer los valores, pero la biblioteca maneja esto internamente.

3. Lectura de la temperatura en grados Celsius:

```
float celsius_of_ds18b20 = sensors.getTempCByIndex(0);
```

- **Propósito:** Lee la temperatura medida por el primer sensor DS18B20 conectado al bus (índice 0) y la almacena en la variable `celsius_of_ds18b20`.
- **Detalles:** La función `getTempCByIndex(0)` retorna la temperatura en grados Celsius del sensor especificado por el índice (en este caso, 0). Esta función asume que hay al menos un sensor conectado al bus.

4. Retorno de la temperatura calculada `return celsius_of_ds18b20;`

- Devuelve la temperatura medida en grados Celsius.

Relevancia:

La función `ds18b20_sensor()` es crucial para el proyecto, ya que proporciona una lectura de temperatura digital precisa desde el sensor DS18B20. Esta medida es una de las dos entradas utilizadas para calcular la temperatura promedio que se muestra y monitorea en el sistema.

Resumen del funcionamiento:

- **Solicitud de temperatura:** La función `sensors.requestTemperatures()` inicia el proceso de medición de temperatura del sensor DS18B20.
 - **Lectura de temperatura:** La función `getTempCByIndex(0)` recupera la temperatura medida por el primer sensor DS18B20 conectado al bus OneWire y la convierte a grados Celsius.
 - **Retorno:** La función devuelve este valor de temperatura para su uso en el cálculo de la temperatura promedio en el sistema.
-

```

float average_temperature(float celsius_ntc, float celsius_ds18B20)
{
    // Convertir los valores float a enteros escalados
    int16_t celsius_ntc_int = celsius_ntc * 100;
    int16_t celsius_ds18B20_int = celsius_ds18B20 * 100;
    int16_t average_temp_int;

    asm volatile(
        "movw r24, %1\n\t"          // Cargar celsius_ntc_int en r24
        "movw r26, %2\n\t"          // Cargar celsius_ds18B20_int en r26
        "add r24, r26\n\t"          // Sumar los bytes menos signifi
        "adc r25, r27\n\t"          // Sumar los bytes más signifi
        "lsr r25\n\t"               // Desplazar un bit a la derech
        "ror r24\n\t"               // Rotar un bit a la derecha
        "movw %0, r24\n\t"          // Guardar el resultado en average
        : "=r" (average_temp_int)
        : "r" (celsius_ntc_int), "r" (celsius_ds18B20_int)
        : "r24", "r25", "r26", "r27"
    );

    // Convertir el resultado de vuelta a float
    return average_temp_int / 100.0;
}

```

La función `average_temperature(float celsius_ntc, float celsius_ds18B20)` calcula la temperatura promedio utilizando valores de temperatura en formato float, pero realiza la operación de promediado en ensamblador para mejorar la eficiencia.

1. Declaración de la función:

```
float average_temperature(float celsius_ntc, float celsius_ds18B20) {
```

- Define una función que retorna un valor `float` representando la temperatura promedio entre dos sensores, un NTC y un DS18B20.

2. Conversión de float a enteros escalados:

```
int16_t celsius_ds18b20_int = celsius_ds18b20 * 100;
```

```
int16_t celsius_ntc_int = celsius_ntc * 100;
```

- **Propósito:** Convierte las temperaturas en formato float a enteros escalados para facilitar las operaciones aritméticas en ensamblador.
- **Detalles:** Multiplicar por 100 convierte un float a un entero representando centésimas de grado (por ejemplo, 25.34 °C se convierte en 2534).

3. Declaración de la variable para el resultado `int16_t average_temp_int;`

- Declara una variable para almacenar el resultado intermedio de la operación en ensamblador.

▼ What is `int16_t`

`int16_t` es un tipo de datos que representa un número entero con signo de 16 bits o 2 bytes.

4. Código en ensamblador:

- **Propósito:** Realiza la suma de las temperaturas y el promedio usando instrucciones de ensamblador para mayor eficiencia.
- **Detalles:**
 - `movw r24, %1\n\t`: Carga `celsius_ntc_int` en los registros r24:r25.
 - `movw r26, %2\n\t`: Carga `celsius_ds18b20_int` en los registros r26:r27.
 - `add r24, r26\n\t`: Suma los bytes menos significativos.
 - `adc r25, r27\n\t`: Suma los bytes más significativos con acarreo.
 - `lsl r25\n\t`: Desplaza un bit a la derecha en r25 (divide por 2).
 - `ror r24\n\t`: Rota un bit a la derecha en r24 (divide por 2).
 - `movw %0, r24\n\t`: Guarda el resultado en `average_temp_int`.

5. Conversión del resultado de vuelta a float:

- `return average_temp_int / 100.0;`
- **Propósito:** Convierte el resultado promedio de centésimas de grado a grados Celsius en formato float.

- **Detalles:** Dividir por 100 convierte el entero a un float representando la temperatura en grados Celsius (por ejemplo, 2534 se convierte en 25.34).

Relevancia:

La función `average_temperature()` es esencial para calcular la temperatura promedio de dos sensores diferentes. Utilizar ensamblador para realizar la suma y el promedio mejora la eficiencia, especialmente en microcontroladores como el ATmega328p, donde los recursos son limitados.

Resumen del funcionamiento:

- **Conversión:** Las temperaturas en formato float se convierten a enteros escalados.
- **Operación en ensamblador:** Se realiza la suma y el promedio de los valores utilizando instrucciones de ensamblador.
- **Conversión final:** El resultado se convierte de nuevo a formato float para ser utilizado en otras partes del programa.

▼ Más información sobre el código en ensamblador

1. `"movw r24, %1\\n\\t"` : Esta línea carga el primer argumento, `celsius_ntc_int` , en los registros de propósito general r24 y r25. Se utiliza `movw` porque los registros r24 y r25 se combinan para manejar valores de 16 bits.
2. `"movw r26, %2\\n\\t"` : Similar al anterior, esta línea carga el segundo argumento, `celsius_ds18b20_int` , en los registros de propósito general r26 y r27.
3. `"add r24, r26\\n\\t"` y `"adc r25, r27\\n\\t"` : Estas dos líneas suman los valores de las temperaturas que están en los registros r24:r25 y r26:r27. La instrucción `add` realiza la suma de los bytes menos significativos, y `adc` suma los bytes más significativos, teniendo en cuenta el acarreo.
4. `"lsr r25\\n\\t"` : Esta línea desplaza un bit a la derecha el contenido del registro r25. Esto se hace para dividir por 2 el valor de r25, ya que es el byte más significativo de la suma de las temperaturas.

5. `"ror r24\\n\\t"` : Aquí se rota un bit a la derecha el contenido del registro r24. Esto se hace para completar la división por 2, ya que el bit que se desplazó de r25 ahora se coloca como el bit más significativo de r24.
6. `"movw %0, r24\\n\\t"` : Esta línea guarda el resultado de la suma en los registros r24:r25 en la variable `average_temp_int`.

Los registros r24, r25, r26 y r27 son usados como registros de propósito general en lenguaje ensamblador. El código `: "r24", "r25", "r26", "r27"` en las líneas de ensamblador indica que estos registros serán utilizados y modificados durante la ejecución de la instrucción ASM. La notación `"%0"`, `"%1"`, y `"%2"` se refiere a los valores de las variables en los argumentos de entrada y salida, con `%0` representando `average_temp_int`, `%1` representando `celsius_ntc_int`, y `%2` representando `celsius_ds18b20_int`.

```
int potentiometer_interval() {  
  
    int interval_of_potentiometer;  
    valPotenciometer = analogRead(analogPinPotenciometer);  
    interval_of_potentiometer = map(valPotenciometer, 0, 1023, 500, 5000);  
  
    return interval_of_potentiometer;  
}
```

Este fragmento de código implementa una función llamada `potentiometer_interval` que devuelve un valor entero representando un intervalo basado en la lectura de un potenciómetro analógico. Aquí está la explicación línea por línea:

1. `int interval_of_potentiometer;` : Declara una variable entera llamada `interval_of_potentiometer` para almacenar el intervalo calculado.
2. `valPotenciometer = analogRead(analogPinPotenciometer);` : Lee el valor analógico del pin conectado al potenciómetro y lo almacena en una variable llamada `valPotenciometer`. La función `analogRead` devuelve un valor entre 0 y 1023, que corresponde al rango de voltajes del pin analógico.
3. `interval_of_potentiometer = map(valPotenciometer, 0, 1023, 500, 5000);` : Utiliza la función `map` para convertir el valor leído del potenciómetro (que puede estar

en el rango de 0 a 1023) a un nuevo rango de valores. En este caso, mapea el valor leído desde el rango de 0 a 1023 al rango de 500 a 5000. Por lo tanto, si el valor leído del potenciómetro es 0, el intervalo será 500, y si es 1023, el intervalo será 5000. Esto permite ajustar el intervalo según la posición del potenciómetro.

4. `return interval_of_potentiometer` : Devuelve el intervalo calculado.

En resumen, esta función lee un valor analógico de un potenciómetro, lo mapea a un rango de intervalos específico y devuelve este valor mapeado. Esto puede ser útil para ajustar parámetros en un sistema en función de la posición del potenciómetro.

```
void unitDisplayNumber(int num) {  
  
    for (int i = 0; i < 8; i++) {  
        digitalWrite(segmentUnitPins[i], HIGH);  
    }  
  
    for (int i = 0; i < 8; i++) {  
        if (bitRead(unitsNumbers[num], i) == LOW) {  
            digitalWrite(segmentUnitPins[7 - i], LOW);  
        }  
    }  
}
```

Esta función `unitDisplayNumber` se encarga de mostrar un número en un display de siete segmentos. Aquí está cómo funciona línea por línea:

1. **Configuración inicial de pines:** El primer bucle `for` itera sobre los pines del display de siete segmentos. Para cada iteración, establece el pin correspondiente en alto (HIGH) utilizando la función `digitalWrite()`. Esto se hace para asegurarse de que todos los segmentos estén apagados inicialmente.
2. **Mostrar el número:** El segundo bucle `for` itera sobre cada bit del patrón de segmentos asociado al número que se quiere mostrar (`num`). Utiliza la función `bitRead()` para leer cada bit del patrón de segmentos correspondiente al

número `num`. Si el bit es bajo (LOW), significa que el segmento debe encenderse. En ese caso, se establece el pin del display de siete segmentos correspondiente a ese bit en bajo (LOW) utilizando `digitalWrite()`.

- `unitsNumbers[num]` : Accede al patrón de segmentos asociado al número `num` en el array `unitsNumbers`.
- `bitRead(unitsNumbers[num], i)` : Lee el bit en la posición `i` del patrón de segmentos asociado al número `num`.
- `segmentUnitPins[7 - i]` : Selecciona el pin del display de siete segmentos correspondiente al bit actual. La resta `7 - i` se utiliza para invertir el orden de los bits, ya que los segmentos suelen estar conectados de manera inversa en hardware.

En resumen, esta función configura inicialmente todos los segmentos del display de siete segmentos en alto (apagados) y luego enciende selectivamente los segmentos necesarios para mostrar el número proporcionado (`num`).

En la función `unitDisplayNumber`, el término `7 - i` se utiliza para invertir el orden de los índices del array `segmentUnitPins` al acceder a los pines del display de siete segmentos. Esto es necesario porque los segmentos del display están conectados de manera inversa en el hardware. Al invertir los índices, se asegura que los segmentos se enciendan en el orden correcto para mostrar el número deseado.

```
void tensDisplayNumber(int num) {  
  
    Wire.beginTransaction(hw171Address);  
    Wire.write(tensNumbers[num]);  
    Wire.endTransmission();  
}
```


1. Declaración de la función `void tensDisplayNumber(int num) {}`

- Define una función que toma un número entero `num` como argumento y no retorna ningún valor.

2. Iniciar la transmisión I2C `Wire.beginTransmission(hw171Address);`

- **Propósito:** Inicia una transmisión I2C al dispositivo con la dirección `hw171Address`.
- **Detalles:** `hw171Address` es la dirección I2C del módulo HW-171, que se define anteriormente como `0x20`.

3. Enviar el valor al display `Wire.write(tensNumbers[num]);`

- **Propósito:** Escribe un byte al bus I2C, que representa el patrón binario para mostrar el número de las decenas en el display de 7 segmentos.
- **Detalles:** `tensNumbers` es un array que contiene los patrones binarios para los números 0 a 9. El índice `num` corresponde al número que queremos mostrar.

4. Finalizar la transmisión I2C `Wire.endTransmission();`

- **Propósito:** Finaliza la transmisión I2C, enviando el byte al módulo HW-171.
- **Detalles:** Esto asegura que el patrón binario enviado se procese correctamente y se muestre en el display.

Relevancia:

Esta función es crucial para la visualización de la parte de las decenas de la temperatura en un display de 7 segmentos controlado por el módulo HW-171. Utilizando el protocolo I2C, la función garantiza una comunicación eficiente y precisa con el módulo de display.

Contexto adicional:

- **Protocolo I2C:** Es un protocolo de comunicación serial que permite que múltiples dispositivos se comuniquen con un solo bus de datos. En este caso, el microcontrolador Arduino se comunica con el módulo HW-171.

- **HW-171:** Es un módulo comúnmente utilizado para controlar displays de 7 segmentos a través de I2C, facilitando la visualización de números sin necesidad de conectar cada segmento individualmente al microcontrolador.

```
int getUnits(float numero) {

    int entero = (int)numero;
    int unidades;

    asm volatile(
        "ldi r18, 10\n\t"      // Load immediate value 10 into r18
        "movw r24, %1\n\t"    // Move the value of entero into r24
        "1: subi r24, 10\n\t"  // Subtract 10 from r24
        "brcs 2f\n\t"         // If carry set, branch to label 2
        "rjmp 1b\n\t"         // Jump back to label 1
        "2: subi r24, -10\n\t" // Add 10 to r24 (undo last subtraction)
        "mov %0, r24\n\t"     // Move the final result (units) to %0
        : "=r"(unidades)      // Output operands
        : "r"(entero)         // Input operands
        : "r18", "r24", "r25" // Clobbers
    );

    return unidades;
}
```

1. Declaración de la función `int getUnits(float numero) {}`

- Define una función que toma un número de punto flotante `numero` y devuelve las unidades de la parte entera de este número.

2. Conversión de float a int `int entero = (int)numero;`

- Convierte el número de punto flotante `numero` a un entero `entero`.

3. Declaración de la variable de unidades `int unidades;`

- Declara una variable `unidades` que almacenará el valor de las unidades del número entero.

4. **Inicio del ensamblador en línea** `asm volatile()`
 - Comienza un bloque de código en ensamblador en línea. La palabra clave `volatile` indica al compilador que no optimice este bloque de código.
5. **Cargar el valor 10 en r18** `"ldi r18, 10\\n\\t"`
 - Carga el valor inmediato 10 en el registro `r18`.
6. **Mover el valor de entero a r24:r25** `"movw r24, %1\\n\\t"`
 - Mueve el valor de `entero` (almacenado en un registro de entrada) a los registros `r24` y `r25`.
7. **Etiqueta 1: Restar 10 de r24** `"1: subi r24, 10\\n\\t"`
 - Resta 10 del registro `r24`.
8. **Si hay carry, saltar a la etiqueta 2** `"brcs 2f\\n\\t"`
 - Si el resultado de la resta anterior causa un acarreo (carry), salta a la etiqueta 2 (final).
9. **Saltar de nuevo a la etiqueta 1** `"rjmp 1b\\n\\t"`
 - Salta de nuevo a la etiqueta 1 para continuar restando 10 hasta que el resultado sea menor que 10.
10. **Etiqueta 2: Añadir 10 a r24 (deshacer la última resta)** `"2: subi r24, -10\\n\\t"`
 - Añade 10 a `r24` para deshacer la última resta que causó el acarreo.
11. **Mover el resultado a la variable de salida** `"mov %0, r24\\n\\t"`
 - Mueve el valor final en `r24` (las unidades) a la variable de salida `unidades`.
12. **Definición de operandos de salida** `:"=r"(unidades)`
 - Especifica que `unidades` es un operando de salida, que se almacenará en un registro.
13. **Definición de operandos de entrada** `:"r"(entero)`
 - Especifica que `entero` es un operando de entrada, que se cargará desde un registro.
14. **Clobbers** `:"r18", "r24", "r25"`

- Informa al compilador que estos registros serán modificados por el ensamblador en línea.

15. Retorno del resultado `return unidades;`

- Devuelve el valor de las unidades calculadas.

```
int getTens(float numero) {

    int entero = (int)numero;
    int decenas;

    asm volatile(
        // Dividir por 10
        "ldi r18, 10\n\t" // Load immediate value 10 into register r18
        "clr r25\n\t"     // Clear register r25 (high byte of r24)
        "movw r24, %1\n\t" // Move the value of entero into r24
        "divloop:\n\t"
        "cp r24, r18\n\t" // Compare r24 with 10
        "brcs enddiv\n\t" // If carry set, branch to enddiv
        "subi r24, 10\n\t" // Subtract 10 from r24
        "inc r25\n\t"     // Increment the quotient
        "rjmp divloop\n\t" // Repeat the loop
        "enddiv:\n\t"
        // Obtener el dígito de las decenas
        "mov r24, r25\n\t" // Move the quotient to r24
        "andi r24, 0x0F\n\t" // Mask the lower nibble to get the tens digit
        "mov %0, r24\n\t" // Move the final result (tens) to decenas
        : "=r"(decenas) // Output operands
        : "r"(entero) // Input operands
        : "r18", "r24", "r25" // Clobbers
    );

    return decenas;
}
```

1. **Declaración de la función** `int getTens(float numero) {}` :
 - Define una función que toma un número de punto flotante `numero` y devuelve las decenas de la parte entera de este número.
2. **Conversión de float a int** `int entero = (int)numero;` :
 - Convierte el número de punto flotante `numero` a un entero `entero`.
3. **Declaración de la variable de decenas** `int decenas;` :
 - Declara una variable `decenas` que almacenará el valor de las decenas del número entero.
4. **Inicio del ensamblador en línea** `asm volatile(:`
 - Comienza un bloque de código en ensamblador en línea. La palabra clave `volatile` indica al compilador que no optimice este bloque de código.
5. **División por 10:**
 - Se realiza una división entera del valor entero por 10 utilizando instrucciones en ensamblador. El resultado de la división se almacena en `r24` y el cociente en `r25`.
6. **Etiqueta de bucle de división** `"divloop:\\n\\t"` :
 - Etiqueta para el bucle de división.
7. **Comparación con 10** `"cp r24, r18\\n\\t"` :
 - Compara `r24` con 10 para determinar si es mayor o igual.
8. **Si hay carry, saltar a enddiv** `"brcs enddiv\\n\\t"` :
 - Si hay un carry (indicando que `r24` es menor que 10), salta a la etiqueta `enddiv`.
9. **Resta 10 de r24** `"subi r24, 10\\n\\t"` :
 - Resta 10 de `r24` para continuar dividiendo por 10.
10. **Incrementa el cociente** `"inc r25\\n\\t"` :
 - Incrementa el cociente para contar las veces que se ha restado 10.
11. **Salto de vuelta al bucle de división** `"rjmp divloop\\n\\t"` :

- Salta de vuelta al inicio del bucle de división.

12. Etiqueta de finalización de división `"enddiv:\\n\\t"`:

- Etiqueta para el final de la división cuando `r24` es menor que 10.

13. Obtener el dígito de las decenas:

- `"mov r24, r25\\n\\t"`
- `"andi r24, 0x0F\\n\\t"`
- Extrae el dígito de las decenas del cociente (almacenado en `r25`) y lo almacena en `r24`.

14. Mover el resultado a la variable de salida `"mov %0, r24\\n\\t"`:

- Mueve el valor final en `r24` (las decenas) a la variable de salida `decenas`.

▼ Instrucción `andi`

Realiza una operación AND bit a bit entre un registro y un valor inmediato (constante) y almacena el resultado en el registro de destino. La sintaxis general es:

En el código que proporcionaste, la instrucción `andi` se utiliza para realizar una operación de máscara en el registro `r24`. La operación de máscara se realiza para extraer solo los bits de interés y descartar los demás.

En resumen, `andi r24, 0x0F` está limpiando (borrando) los bits superiores del registro `r24`, dejando solo los 4 bits inferiores, que probablemente se utilizan para almacenar un valor dentro del rango de 0 a 9. Esto podría ser parte del proceso para extraer un dígito decimal específico de un número mayor.