DS18B20 Waterproof Digital Temperature Sensor

1. About the Sensor Module

The DS18B20 is a digital temperature sensor chip. It is produced by Maxim Integrated Circuits and the sensor comes in different variants, for example as SMD package (Surface Mount Device) or with three pins as TO-92 package; that is a compact type of casing often used for transistors and small ICs (Integrated Circuits). There are also water-proof metal casings with a longer wire attached to it, which makes sensing in harsher environments, for example under water, possible. As the sensor will be used to measure the water temperature close to the surface and at the bottom of the pond, the waterproof version was used in this case (Figure 1).

The sensor is not able to sense the temperature directly through the package, instead, it primarily uses the GND pin for measuring the temperature. Therefore, it is necessary that the GND pin has a sufficient thermal connection to the heat source to be measured. The better the connection, the more accurate the results and the faster can temperature changes be measured. In the case of the waterproof version, the GND pin is thermally connected to the metal casing of the probe.

Like the DHT-22, the DS18B20 features only 3 (connected) pins as it works using 1-Wire communication which will be explained in more detail in section 2. Unlike the DHT-22, the resolution of the DS18B20 can be configured to 9, 10, 11 or 12 bits, where 12 bits is the default resolution. The increments of temperature of the resolutions is 0.5°C, 0.25°C, 0.125°C and 0.0625°C, respectively. Even though a decrease in resolution impairs the accuracy of the measurement, it has the advantage of shortening the time necessary for taking a measurement and reducing power consumption.

It also features a variety of other possibilities such a powering the device through a pullup resistor connected to the Data line (parasite mode) or setting alarms for a high and a low temperature.

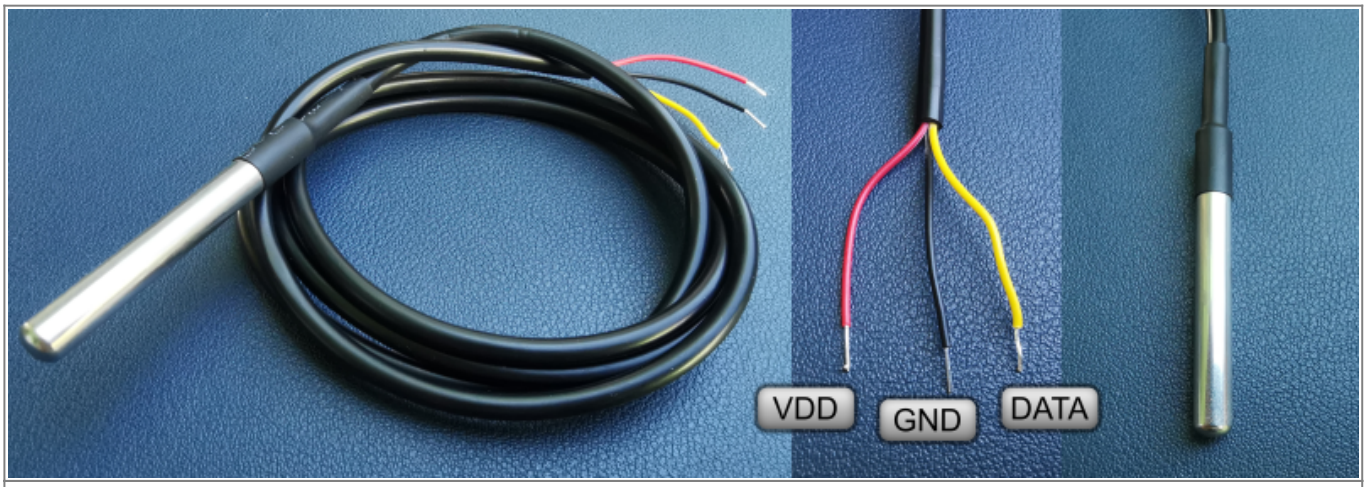


Figure 1 DS18B20 waterproof digital temperature sensor and pins.

2. Data Transmission and Working Principle of the Module

2.1 Serial Code

One of the biggest advantages of the DS18B20 in contrast to the DHT-22 is, that each sensor has a unique 64 serial code, which works as an address during communication. Due to the unique address many sensors can be connected to the same 1-Wire bus and can then be addressed by the microcontroller individually. Because of this, only a single GPIO pin of the MCU is needed to read data from all the DS18B20s connected to it.

The serial code contains different information:

- The 8 LSB contain the DS18B20's 1-Wire family code: 0x28.
- The following 48 bits contain a unique serial number.
- The 8 MSB contain the 8-bit CRC byte (Cyclic Redundancy Check), which is calculated from the 56 bits before.

The 64-bit address is stored in a 64-bit ROM (Read Only Memory), that means, the address is assigned during manufacture and cannot be changed afterwards; this prevents conflicts of two devices having the same address later on.

There are two different CRCs, one for the serial code, which does not change and one for the data contained in the memory of the module (scratchpad memory), which changes depending on the temperature readings. Both CRCs are a method for the microcontroller for validating that the transmission of the requested information, either from the ROM or the scratchpad, was done correctly (for more information check the datasheet). The DHT-22 used a similar approach for data validation with the check-sum byte.

2.2 Data Storage

The 64-bit ROM and the 1-Wire port are connected through the memory control logic to the scratchpad. The scratchpad is a high-speed internal memory that is used for storing small, often temporary, pieces of data which can be rapidly retrieved if needed. The scratchpad memory, which gets erased after powering down, is connected to the temperature sensor itself and stores its data in a 2-byte register. Furthermore, the module has a small EEPROM (Electronically Erasable Programmable Read Only Memory), that means a nonvolatile data storage, which retains the data even when the device is powered down. The Arduino UNO and the ESP32 also have such storages, but of larger size. The module can transfer the aforementioned alarm temperatures TH and TL, as well as the configuration register from the scratchpad into the EEPROM and reload them back to the scratchpad memory after powering up again (figure 2).

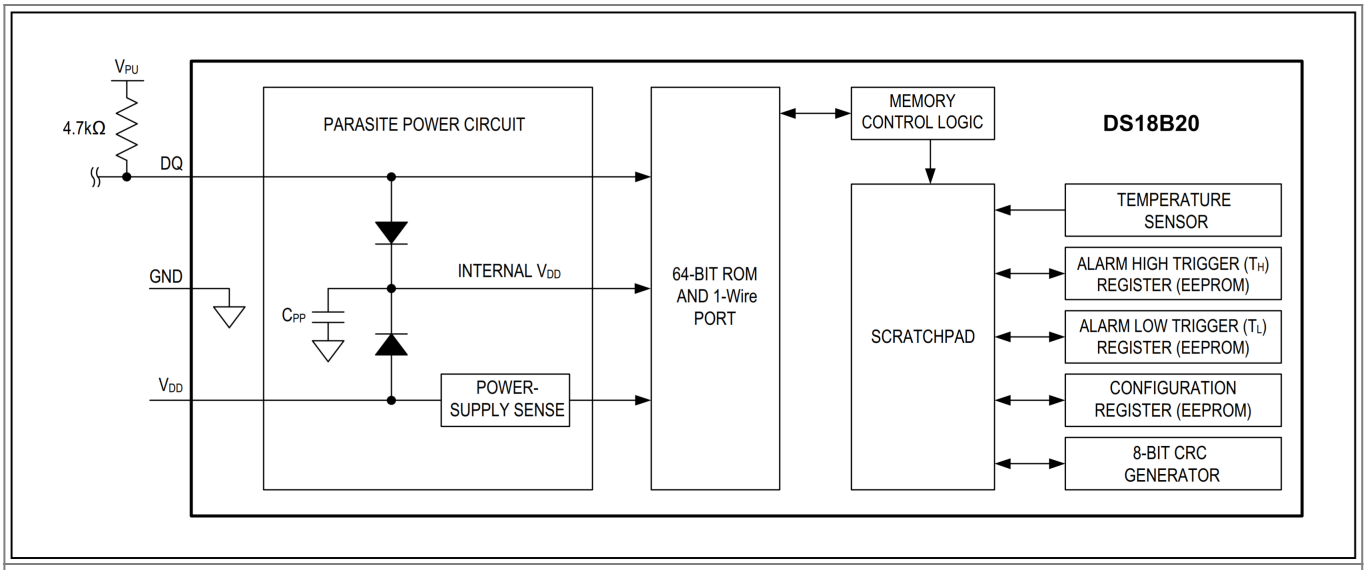


Figure 2 DS18B20 block diagram (Source: DS18B20 datasheet p. 5)

2.3 Temperature Register

Normally the sensor is in an idle state after powering up. When the MCU issues a Convert T command, the sensor measures the temperature and does an analog to digital conversion and stores the result in the scratchpad 2-byte temperature register (figure 3) and the sensor goes back into idle state.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LS Byte	2^3 8	2^2 4	2^1 2	2^0 1	2^{-1} 0.5	2^{-2} 0.25	2^{-3} 0.125	2^{-4} 0.0625
	Bit 15	**Bit 14**	**Bit 13**	**Bit 12**	**Bit 11**	**Bit 10**	**Bit 9**	**Bit 8**
MS Byte	S	S	S	S	S	2^6 64	2^5 32	2^4 16

Figure 3 Structure of the temperature register, LS = Least Significant, MS = Most Significant, S = Sign.

In the register, the 5 leading bits are representing the sign of the temperature. If the temperature is below 0°C, S = 1 and if it is above 0°C, S = 0. The other 11 bits represent the temperature indicated in figure 3, where bit 0, 1 and 2 can be undefined depending on the resolution stored in the configuration register:

- 12 bit: all bits contain valid data. (0.0625 increments)
- 11 bit: bit 0 is undefined. (0.125 increments)
- 10 bit: bit 0 and bit 1 are undefined. (0.25 increments)
- 9 bit: bit 0, bit 1, and bit 2 are undefined. (0.5 increments)

The maximum range of values that can be displayed is thus -127.9375 – 127.9375, which exceeds the range of the temperature sensor. Some examples of different temperatures displayed in 12-bit resolution can be seen in table 1.

Table 1 **Different temperature values and how they are stored in the temperature register.**

MS Byte	LS Byte	Temperature
0000 0111	1101 0000	+ 125 °C

Table 1 **Different temperature values and how they are stored in the temperature register.**

MS Byte	LS Byte	Temperature
0000 0001	0111 0001	+ 23.0625 °C
0000 0000	1000 1100	+ 8.75 °C
0000 0000	0000 0000	0 °C
1111 1111	0011 1101	- 12.125 °C
1111 1100	1001 0000	- 55°C

2.4 Configuration Register

The configuration register is stored in the scratchpad memory (and EEPROM) and contains 1 byte of data. The MSB (bit 7) and the 5 LSB (bit 0 – 4) are reserved for internal use and cannot be changed. Bit 6 (R1) and bit 5 (R0) can be changed to adjust the resolution (figure 4). As can be seen, the conversion time doubles each time the resolution is increased by 1. A 12-bit resolution measurement takes 8 times as long as a 9-bit conversion.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	R1	R0	1	1	1	1	1

Thermometer Resolution Configuration:

R1	R0	Resolution (Bits)	Maximum Conversion Time
0	0	9	93.75 ms
0	1	10	187.5 ms
1	0	11	375 ms
1	1	12	750 ms

Figure 4 Schematic of the configuration register and the different thermometer resolution configurations.

2.5 1-Wire Bus

In the 1-Wire bus, the MCU, in this case the ESP32, acts as master and the DS1820 sensors act as slaves. It should be noted that all commands and data are transmitted with their least significant bit first in the bus.

Sending signals on the 1-Wire bus works in a similar way to the transmissions with the DHT-22. Normally, that is when no communication is happening and the data-bus is available, the potential of the data bus is high; with the ESP32 that means 3.3V. In this case, it is necessary to add an external pullup resistor of approximately 5kΩ. The external resistor and the resistor in the input of the ESP32 act as a voltage divider and because the input resistance is much higher, the potential in the data-bus is almost 3.3V. The DS18B20 (as well as the ESP32) communicate with each other by shorting the data-bus to GND. The DS18B20 is an open drain device. That means that its signal output is done using a transistor, which shorts the collector/drain (the data bus) to the emitter/source (GND).

Each communication between MCU and DS18B20 needs to follow a transaction sequence that consists of 3 steps:

1. Initialization

2. ROM command (followed by any required data exchange)
3. DS18B20 Function command (followed by any required data exchange)

The initialization consists of a reset pulse issued by the master (ESP32) followed by a presence pulse transmitted by the slaves which informs the master that there are 1-Wire devices on the bus which are ready to operate.

After having detected a presence pulse, the master can transmit a ROM command, which operates on the 64-bit ROM codes that each slave device has. There are different ROM commands, which allow the master to search and identify all the slaves on the bus and address a single device (or send a command for temperature measurement to all of them at once).

In the third step, a DS18B20 function command is issued by the master which allows to initiate a temperature measurement, read certain information like the power supply mode or temperature, or configure the temperature alarms or sensor resolution.

Similar to the DHT-22 protocol, the 1-Wire data bus uses a protocol with exactly defined time periods for the low and high voltage level pulses for sending or receiving a logical 1 or 0 and for pulses used for initialization and ROM commands. The exact data can be found in the datasheet of the DS18B20.

3. Technical Specifications and Setup of the Sensor

When connecting the pins (figure 1), pin 1 (red) is VDD, pin 2 (black) is GND and pin 3 (yellow) is the data pin. In this project two sensors, one measuring the temperature at the bottom of the pond and one at the surface, are connected to the ESP32. As each sensor can be addressed individually by its serial code, the data pins can be connected to the same GPIO pin of the ESP32 (here pin 14 was used). As mentioned before, the sensors need a pullup resistor connecting the data pin with VDD. As both sensors are connected in the same data bus, only a single 5kΩ resistor between their data pins and VDD needs to be added.

Like the DHT-22, VDD can be anywhere between 3.0 V to 5.5 V and can thus be powered by the ESP32 without a problem. Further information on the sensor's specifications can be found in table 2 and in the datasheet.

Table 2 Specifications of the DS18B20 digital temperature sensor (local power supply with ESP32).

Sensor	**DS18B20**	
Supply Voltage V_{DD}	3.3 V	
Pullup Supply Voltage V_{PU}	3.3 V	
Input Logic-Low V_{IL}	-0.3 V – 0.8 V	
Input Logic-High V_{IH}	2.2 – 3.6 V	
Active Current I_{DD}	1 – 1.5 mA	
Standby Current I_{DDS}	0.75 – 1.0 uA	
Thermometer Range	-55 °C – 125°C	
Thermometer Error	-10 °C – 85 °C	±0.5 °C
	-30 °C – 100 °C	±1 °C
	-55 °C – 125 °C	±2 °C

Resolution, Temperature Increments, Conversion Time	9 bit	0.5 °C	750 ms
	10 bit	0.25 °C	375 ms
	11 bit	0.125 °C	187.5 ms
	12 bit	0.0625 °C	93.75 ms

4. Programming the DS18B20

4.1 Obtain the Sensor Addresses

For the programming of the DS18B20, the OneWire.h library and the DallasTemperature.h library were used which can be found on github or in the Arduino library manager.
To distinguish the two sensors, it is advisable to first determine their individual serial codes which can be done using the OneWire.h library. The simple sketch searches for devices on the 1-Wire bus and prints their address in the serial monitor. To distinguish the two sensors, the one for the bottom of the pond was marked with tape (sensor 1), while the one for the surface was left as it was (sensor 2). The sketch was uploaded to the ESP32, sensor 1 was connected to the data bus and the ESP32 was restarted (EN – Enable on the ESP32 module) while connected to the computer. The address was obtained, and the process was repeated for the second sensor.

4.1.1 Code

DS18B20_Device_Address.ino

```
//1-Wire Device Address determination

#include <OneWire.h>                                    //1
#include <DallasTemperature.h>
const uint8_t ONE_WIRE_BUS = 14;                        //2

OneWire oneWire(ONE_WIRE_BUS);                          //3

DeviceAddress SensorAddress;                            //4

void setup() {
  Serial.begin(115200);                                 //5
  Serial.println("Searching for devices on the bus...");
  oneWire.search(SensorAddress);                        //6
  Serial.print("Device Address: ");
  printAddress(SensorAddress);                          //7
  Serial.println("============================");
  Serial.println();
}

void loop() {
}

void printAddress(DeviceAddress SensorAddress){         //8
```

```
  for(uint8_t i=0; i<8;i++){                                    //9
    Serial.print("0x");                                        //10
    if(SensorAddress[i]<16) Serial.print("0");                  //11
    Serial.print(SensorAddress[i],HEX);                        //12
    if(i<7) Serial.print(",");                                 //13
  }
  Serial.println();
}
```

4.1.2 The Code Explained

1. Both libraries need to be included for the sketch to work.
2. The GPIO pin of the ESP32 connected to the 1-Wire bus is defined.
3. The object oneWire, representing the data bus, is created as an instance of the OneWire class. As an argument it needs the GPIO pin defined before.
4. The object Sensor Address of the class DeviceAddress is created. It stores the device address in an array.
5. The serial connection is started to display the address on the serial monitor.
6. The method search() of oneWire is executed, which searches for devices on the bus and stores their addresses in the SensorAddress object.
7. The function printAddress, defined below, is executed to print the address obtained in (6).
8. The function printAddress does not have a return value (void) and needs an object of the class DeviceAddress as an argument.
9. Each byte from the address array is extracted individually. As one address contains 8 bytes, the loop is executed 8 times.
10. The printing of the values is done in hexadecimal because it is more convenient than binary. The 0x indicates a hexadecimal number. In the sketch for reading the temperature, the device addresses will be added manually for both sensor in hexadecimal format. Using the serial print commands, the address is printed in such a way, that it can just be copied from the serial monitor into the DeviceAddress array in the real sketch later on.
11. An 8 bit number (1 byte) contains 2 digits when written as hexadecimal. If the byte has value below 16 (> 0001 00002 / 0x10), the first digit will be 0 in hexadecimal. When printing numbers on the serial monitor, leading 0s are omitted. Therefore, if the byte contains a leading 0, it has to be printed manually to the serial monitor.
12. The byte with index i is printed. The for loop goes through the first 8 bytes (the complete address) with the incrementing variable i.
13. For all bytes, except the last one, a comma is inserted, so that it can be copied into the array.

4.1.3 Results

After the procedure described in section 4.1, the serial monitor will displays something like this:

```
Searching for devices on the bus...
Device Address: 0x28,0xC4,0xA0,0x51,0x38,0x19,0x01,0xC2 //1
==========================

Searching for devices on the bus...
```

```
Device Address: 0x28,0x0B,0xDB,0x60,0x38,0x19,0x01,0xA3 //2
============================

Searching for devices on the bus...
Device Address: 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 //3
============================
```

1. This is the address of sensor 1 for the bottom of the pond.
2. When disconnecting sensor 1 and connecting sensor 2 and pressing the EN (enable) button on the ESP32, the address of the second sensor is printed.
3. If the ESP32 just prints 0 as address, this can have different reasons. It could be that there is no sensor connected, it should be checked if the data pin is connected with the GPIO pin and if VDD and GND are connected properly. Another reason could be a missing pullup resistor.

4.2 Function Sketch

The following sketch activates the sensors, measures the temperatures and calculates the mean value for more accurate results which are then printed to the serial monitor. The function defined in the sketch will be later implemented into the complete code for the monitoring station. To use the sketch, both sensors need to be connected correctly to the ESP32 like explained in section 3.

4.2.1 Code

DS18B20_Temperature_function.ino

```
//DS18B20 ESP32 Temperature Function

//Definitions and Pins
const int ONE_WIRE_BUS = 14;

//Libraries and Objects
#include <OneWire.h>
#include <DallasTemperature.h>

DeviceAddress bottomSensorAddress =
{0x28,0xC4,0xA0,0x51,0x38,0x19,0x01,0xC2};  //1
DeviceAddress surfaceSensorAddress =
{0x28,0x0B,0xDB,0x60,0x38,0x19,0x01,0xA3};    //2
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature DS18B20(&oneWire);                    //3

//Variables
float bottomTem = 0.00;                              //4
String bottomTemperature = "";                                //5

float surfaceTem = 0.00;
String surfaceTemperature = "";
```

```
const uint8_t AveragingNumberDS18B20 = 5;              //6

const uint8_t RESOLUTION = 12;                        //7
const uint8_t TCONV = 750;                            //8
uint8_t delayTime = (TCONV/pow(2, (12-RESOLUTION))) + 10;   //9

void setup() {
  Serial.begin(115200);
  Serial.println("Measurement is starting ...");
}

void loop() {
  measureDS18B20Tem(AveragingNumberDS18B20);          //10
  Serial.println("Sensor 1 (Bottom) Measurement: " + bottomTemperature
+ " °C");//11
  Serial.println("Sensor 2 (Surface) Measurement: " +
surfaceTemperature + " °C");
  Serial.println("=============================================");

  delay(2000);                              //12
}

//DS18B20 Measurement function
void measureDS18B20Tem (const uint8_t AveragingNumber){          //13
  DS18B20.begin();                          //14
  bottomTem = 0;                          //15
  surfaceTem = 0;
  bottomTemperature = "";
  surfaceTemperature = "";

  for(byte i = 0; i < AveragingNumber; i++)          //16
  {
    DS18B20.requestTemperatures();                //17
    delay(delayTime);                           //18
    bottomTem += DS18B20.getTempC(bottomSensorAddress);             //19
    surfaceTem += DS18B20.getTempC(surfaceSensorAddress);
  }
  bottomTem /= AveragingNumber;                             //20
  surfaceTem /= AveragingNumber;

  if(bottomTem<10)                          //21
    bottomTemperature = "0";
  bottomTemperature += bottomTem;                //22

  if(surfaceTem<10)
    surfaceTemperature = "0";
  surfaceTemperature += surfaceTem;
}
```

4.2.2 The Code Explained

1. This is the address of the first sensor for the pond bottom. It is copied from the first sketch for determining the address.
2. This is the address for the second sensor at the pond surface.
3. The object DS18B20 from the DallasTemperature class represents all the DS18B20 sensors connected in this specific 1-Wire bus. As an argument it needs a pointer to the OneWire object created before.
4. The float variable (bottomTem & surfaceTem) are used for taking up the sensor readings, summing them and averaging them over the number of readings.
5. The String object (bottomTemperature & surfaceTemperature) are used for printing the readings and in the final project for sending them to an MQTT Broker.
6. This variable defines how many temperature readings should be taken and averaged.
7. The variable RESOLUTION stores the resolution of the DS18B20 sensors. As the resolution is not changed during the sketch, it is by default 12 bit.
8. The variable TCONV stores the time necessary for the sensor to convert the temperature reading to a digital signal and store it in the scratchpad memory with a resolution of 12 bit.
9. After issuing a temperature conversion command, the sensor needs a certain time (table 2) depending on the resolution for the data to be actualized. If the temperature data is read before, the old data is retrieved. When powering up the sensor first, the default value for T equals +85 °C. If the delay after the convert command is too short, the readings can be highly erroneous. The necessary time for the conversion is $t_{delay}=\frac{t_{CONV}}{2^{(12-RESOLUTION)}}$. As it is an integer variable, the conversion time for 10 bit and 9 bit resolution is rounded down. Therefore, a safety margin of 10ms was added here. The function pow() calculates the value of a number that is raised to a power. The first argument is the base, the second is the exponent. It would also be possible to instead chose a fixed value for delayTime, but when the resolution would be changed, the fixed value would lead to unnecessary time losses unless it is adjusted manually. The expression used here makes the sketch more versatile when the resolution would be changed later on by adjusting the delay automatically.
10. The function measureDS18B20Tem() is defined below and performs the temperature measurements and actualizes the variables. It is to be implemented later on in the complete project sketch.
11. The actualized string objects containing the temperatures is printed to the serial monitor.
12. The delay is just added to make looking at new data a bit easier.
13. The function measureDS18B20Tem() does not have a return value and needs to the number of readings to be done and averaged as an argument (AveragingNumber).
14. The sensors on the bus are started using the begin() method on the DS18B20 object.
15. The past readings are erased, and the strings are emptied.
16. The for-loop repeats as many times as the AveragingNumber states.
17. The method requestTemperatures() issues a global request, that means a command to all devices on the bus, to measure the temperature and refresh the data in the scratchpad memory.
18. The delay is necessary for the data to be refreshed as explained in (9).
19. The getTempC() method reads the scratchpad memory data for the temperature in °C of the sensor whose address is given as function argument. Here the readings are summed up in the float variables.
20. When all the measurements are summed, the sum is divided by the number of measurements (like in the DHT-22 function) to obtain their mean value.
21. Here the string objects are actualized. To make the string always have the same length, a 0 is added manually if the temperature drops below 10 °C. This only works if the temperatures are

not below 0 °C, but so far, the system is not planned to be used during the winter months.
22. The averaged sensor readings are added to the strings.

4.2.3 Results

When the sketch is uploaded and the sensors are correctly connected to the ESP32, the serial monitor reads something similar to this:

```
14:34:50.335 -> Measurement is starting ...
14:34:53.197 -> Sensor 1 (Bottom) Measurement:  25.56 °C //1
14:34:53.197 -> Sensor 2 (Surface) Measurement: 32.54 °C //2
14:34:53.197 -> ============================================
14:35:00.187 -> Sensor 1 (Bottom) Measurement:  25.56 °C
14:35:00.187 -> Sensor 2 (Surface) Measurement: 32.66 °C
14:35:00.187 -> ============================================
14:35:07.209 -> Sensor 1 (Bottom) Measurement:  25.56 °C
14:35:07.209 -> Sensor 2 (Surface) Measurement: 32.78 °C
14:35:07.209 -> ============================================
```

The first sensor (1) was measuring the room temperature on a rather warm day. The second sensor (2) measured the temperature of my hand. Both measurements seem reasonable and when both sensors measure the room temperature, their results are almost equal. However, the temperature probes need some time to get to thermal equilibrium with their environment. When submerged in the water of the pond continuously, this should not be a problem whatsoever.

From:
https://wiki.eolab.de/ - **HSRW EOLab Wiki**

Permanent link:
https://wiki.eolab.de/doku.php?id=amc2020:group_n:ds18b20&rev=1595094586

Last update: **2021/08/24 17:34**