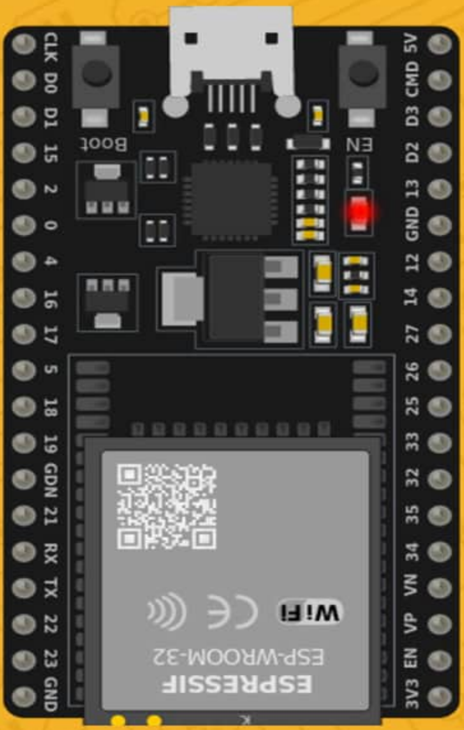
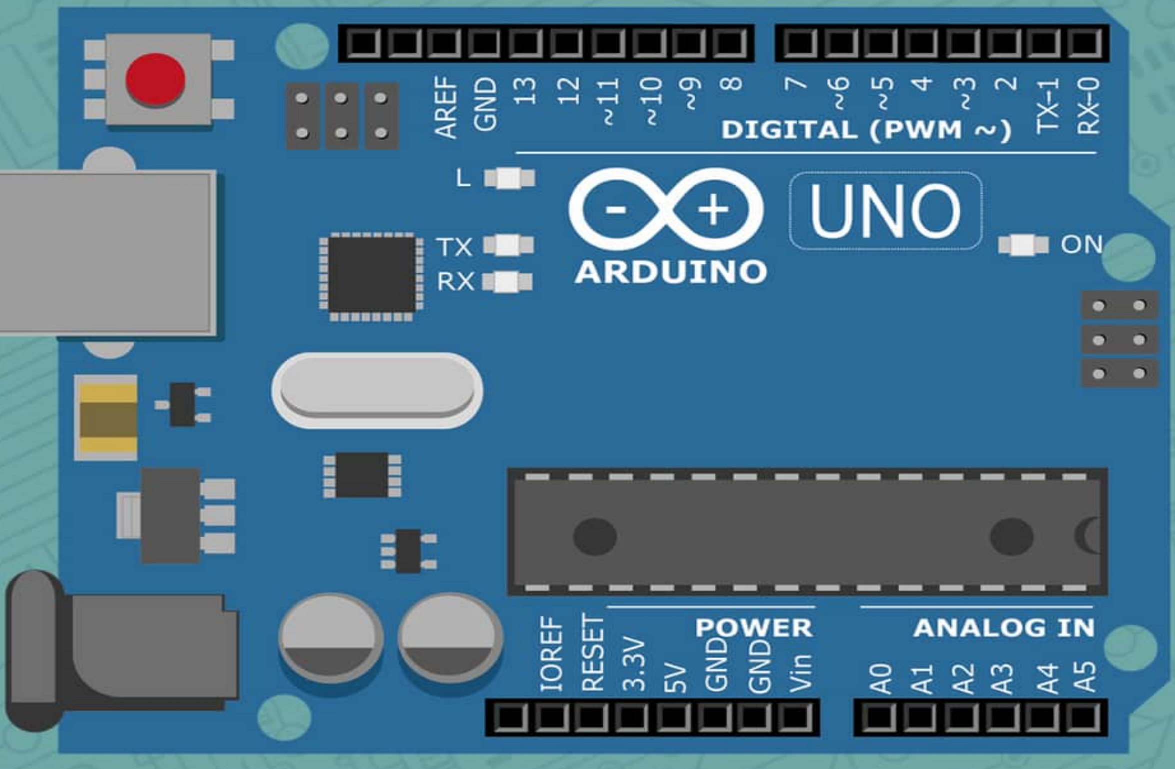
ESP32 AND ARDUINO HARDWARE PROTOCOLS COOKBOOK

Practical Recipes to UART, SPI, I2C, and Non-standard Protocols for Developing Connected Devices





ESP32 AND ARDUINO HARDWARE PROTOCOLS COOKBOOK:

Practical Recipes to UART, SPI, I2C, and Non-standard Protocols for Developing Connected Devices

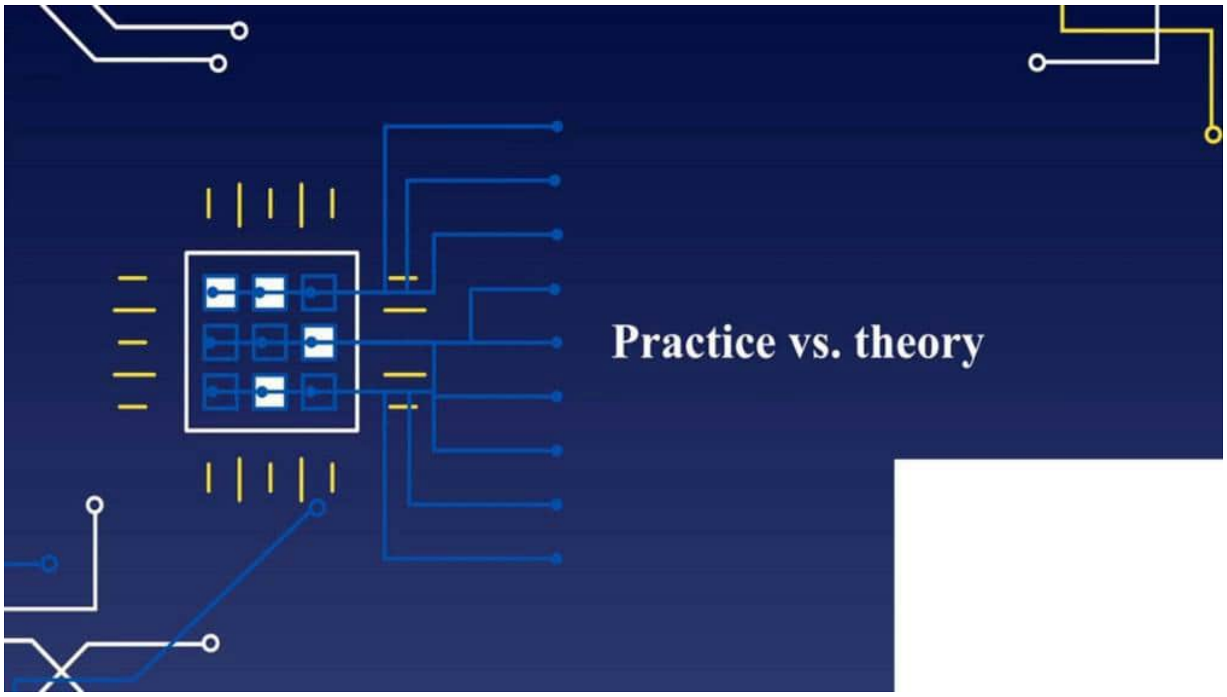
By

Roronoa Hatake

TABLE OF CONTENTS

PRACTICE BEFORE THEORY

Let's talk about my approach according to practice versus theory. So it's good to know in advance how my approach is and how I structured all of the Project content so that you know what you can expect from all of the projects. So my typical approach is that we are talking about the protocols in theory as little as possible. That means that it's not really a university lecturer where I go into details and give you tons of slides with all of the aspects. I'm just focusing on what is really necessary, what we have to do for the protocol and for all of these contexts.



Then we're jumping right away into coding and we are building the sketch step by step. That means I'm not copy pasting anything. I'm developing step by step. And I tell you on each step what I'm doing and why I'm doing this. Then we are uploading the codes and then we're making the analyzes with the logic analyzer to inspect how the whole protocol is set up?

Can we refer the protocol analysis back to the sketch and also from the sketch to the logic analyzer. And then also we investigate and inspect that data sheet so that we can also calculate from the pulse view analysis that, module data or the temperature data, with the calculator. And also, these bits, we want later on to get out from an outer, you know, with our own code so that we really understand what is going on in this protocol, in this module. So combining Sketch logic analyzer, datasheet, and then back to the sketch writing our own libraries to get out the data.

DIFFERENTIATION FROM WIFI, BLUETOOTH ESP NOW

Let us make a delimitation here. We're focusing here only on hardware protocols in the Arduino environment. That means primarily we are talking about UART, we're talking about SPI, OneWire and E2C and those four chapters we are covering. And of course there are many more like USB, Ethernet, JTAG etc. But this will not be covered because the main protocols in the Arduino environment are those four which are covered here. And also the whole Bluetooth topic, so the wireless Wi-Fi, also ESP-Now and ZigBee for example.

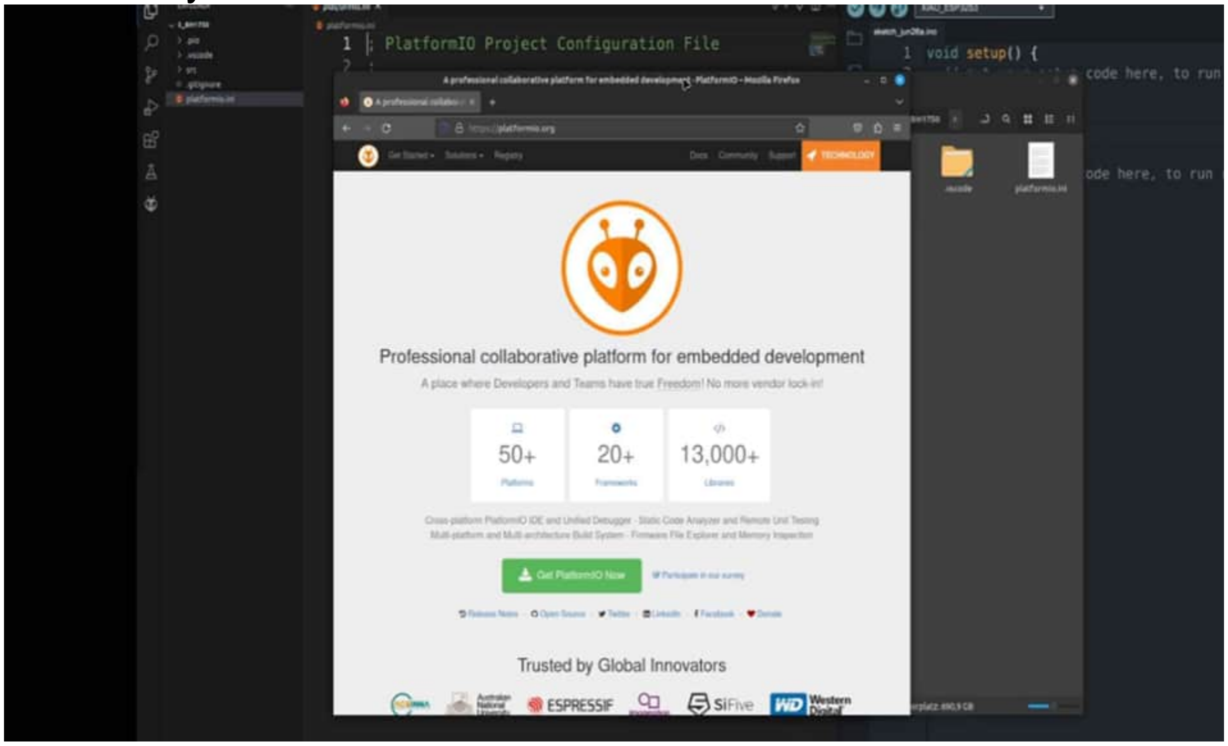
They are necessary but this will be an extra Project because the wireless protocols are much more, I wouldn't say complicated, but a little bit more complex in the way they interact. And therefore this will be a separate chapter and this will be just focusing here on the hardware protocols. That's also the reason why we need a logic analyzer so that we also understand the basics in this Project and can then write our own libraries.

OPEN VSC PLATFORMIO PROGRAM IN ARDUINO IDE

In this project I would like to show you how you can use my code from Visual Studio Code and Platform.io to the Arduino IDE. We are just using here source codes to get data out from the sensors of the Arduino. And this is our main focus. So it doesn't matter which IDE you are using. I am using Visual Studio Code with Platform.io because if you are doing a little bit more with coding and Arduinos, then you will like those components much, much more.

Because the Arduino IDE is OK, but it will lag in certain times. Because one, and although I am really a Linux fan, Microsoft did some really great work with the Visual Studio Code. And this is also open source. And if you have some issues with sending here telemetry data back to Microsoft, you also can google the VS Codium. This is because Visual Studio is an open source program. They are also open source without any Microsoft attachments. But Visual Studio Code, which I am using with Platform.io, is just a great device for coding with the Arduinos.

Give it a try.

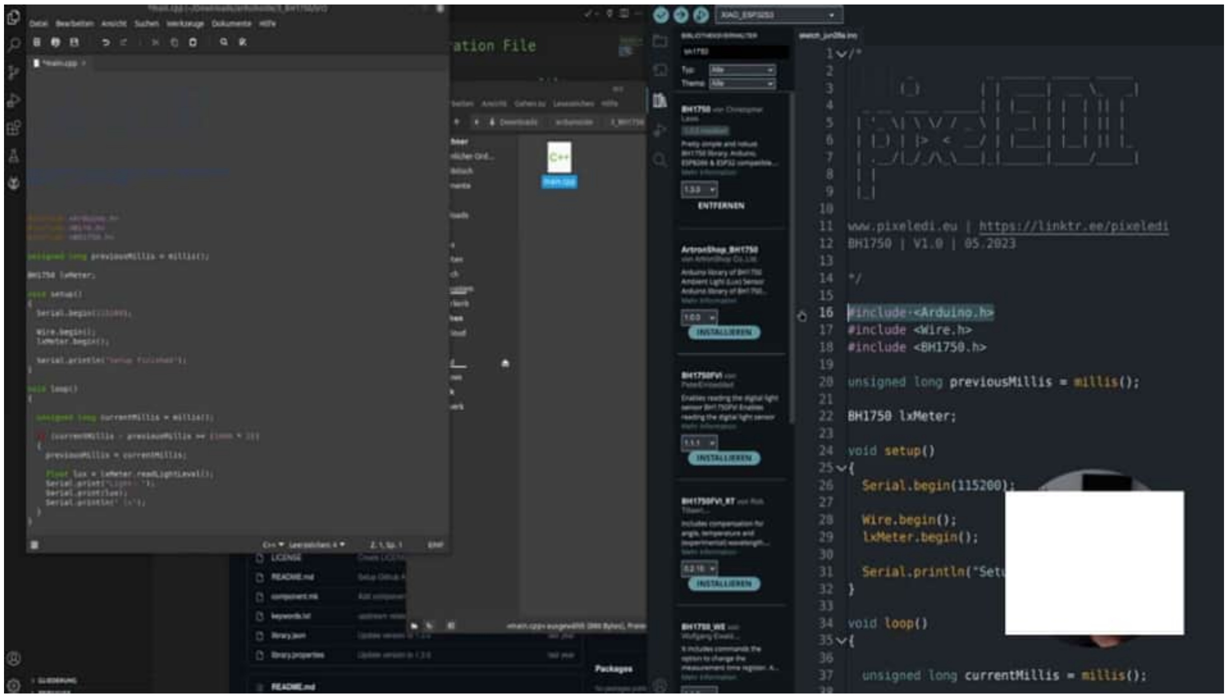

PlatformIO Project Configuration File
void setup() {
Professional collaborative platform for embedded development
A place where Developers and Teams have true Freedom! No more vendor lock-in!
50+
20+
13,000+
Get PlatformIO Now
Trusted by Global Innovators

Because it needs a few minutes of time to set up your environment. But if it is set up, you will have a really great device for coding here. Alone this IntelliSense from Microsoft with this auto-completion of the code, where you have all of these parameters, are a little bit also included in the Arduino IDE. But here you can also use other languages such as PHP, etc. as well. So Platform.io is my recommendation.

But the main project is how you can use my code in the Arduino IDE. Therefore, I have the source code here. For example, if you click here on such a folder, you will have here a source folder and a Platform.io ini. The Platform.io ini is the first thing that you can open. You can see here all of my settings from a hardware perspective. That means I have a USB32 here. This is my baud rate. And here you can find all of my external libraries which are used. And then you can go in the Arduino IDE and can change the baud settings, etc.

And also can find where the libraries are. Here are the bauds. Here are the libraries. And then you can search for example for bh17050. And then you can take a look at who invented it. Clause. This should be this one. Take a look at the information for example. Here you find the clause. That's the same one. And then you can install it and you can have here the same source code as I have. And where is the source code? The source code you can find in the folder Source. And here I have the main CPP. The main CPP can be opened by any text editor.

You can copy and paste all of the content for example. And paste it inside your Arduino IDE. Then you can get rid of the first include statement of the Arduino.h. And that's it.



```
/*
[illegible]

www.pixeledi.eu | https://linktr.ee/pixeledi
BH1750 | V1.0 | 05.2023

*/

#include <Arduino.h>
#include <Wire.h>
#include <BH1750.h>

unsigned long previousMillis = millis();

BH1750 lxMeter;

void setup()
{
  Serial.begin(115200);

  Wire.begin();
  lxMeter.begin();

  Serial.println("Setu
}

void loop()
{

  unsigned long currentMillis = millis();
```

All of this content should work right away. If there are some compiler errors it could be because Visual Platform I.O. and Arduino IDE have some different kinds of C++ compilers. Then there should be some minor changes.

But often the Platform I.O. is more stricter than the Arduino IDE. For example here I have to set up some function prototypes or I have to declare the functions before I use them. In the Arduino IDE that is really a topic and doesn't matter. So for example if I have my function here then I could use this function before it really exists. And this is something that doesn't work in the Platform I.O. for example.

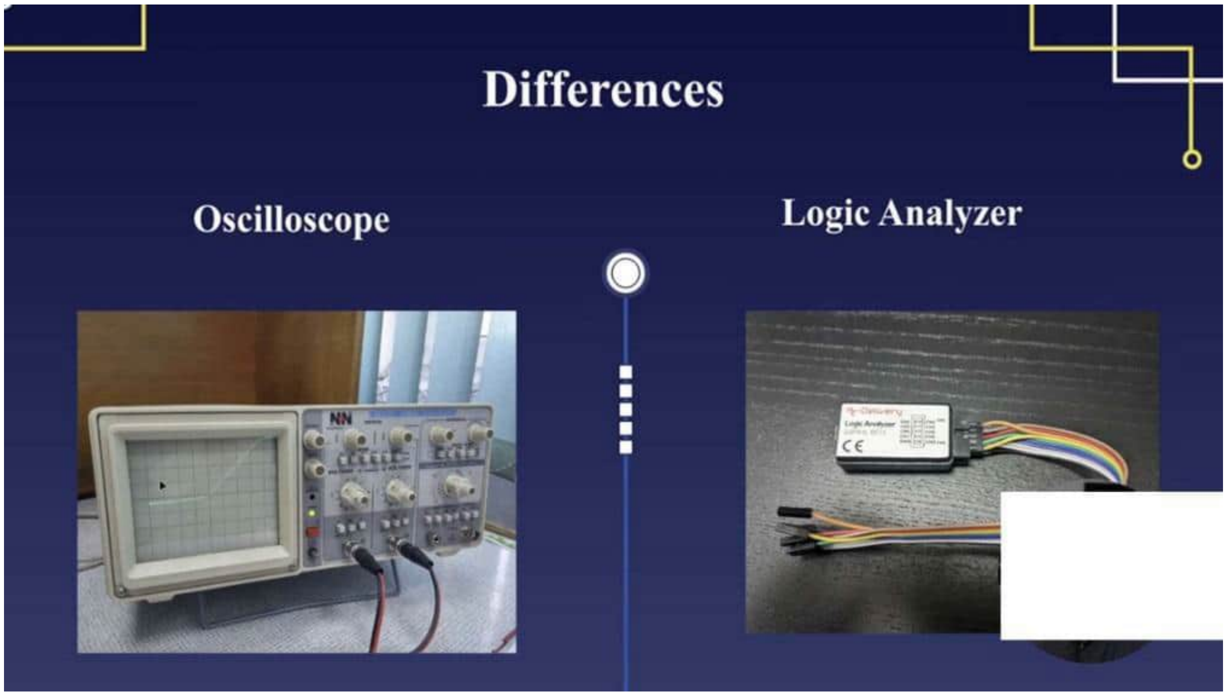
So as you can see here the code is working right away. And this is what you can do to transfer your code.

Another option is you take the CPP, copy to a new folder for example and then you're changing the context to INNO and then you can open it also with the Arduino IDE. As you can see, double click on it and then it should be opened. Need a little bit of time. Then I get here a prompt that I have to change it into a folder. And there we go, we have it. Deleting the Arduino.h and that's it. So this is how you can easily transfer the code from my Platform I.O. to the Arduino IDE.

DIFFERENCE OSCILLOSCOPE VS. LOGIC ANALYZER

In this project, I would like to show the main differences between an oscilloscope, which is on the left side, and a logic analyzer, which we have on the right side. And as you can see here, I have this tiny device. And the main one of the main advantages of this device is that price. It costs you around €20.20. It depends on where you get it.

I bought mine for around €12. An oscilloscope is really a big measurement device. It costs you a couple of €100, it depends on what you need. And we will dig a little bit deeper later on in the next project about all of the features of on logic analyzer. But to give you here a short summary, we have here 8 channels, which is completely sufficient for us in the Hopi area.



Differences

Oscilloscope

Logic Analyzer

And with an Oscilloscope, of course, you have to watch how many levels you would like to capture. But what is now the main difference? The main difference is that we can measure with an oscilloscope analog data. That means you can measure on your circuit, for example, how is the voltage between and diodes. How is the voltage or some, references according to time to the voltage in, upside and and capacitor.

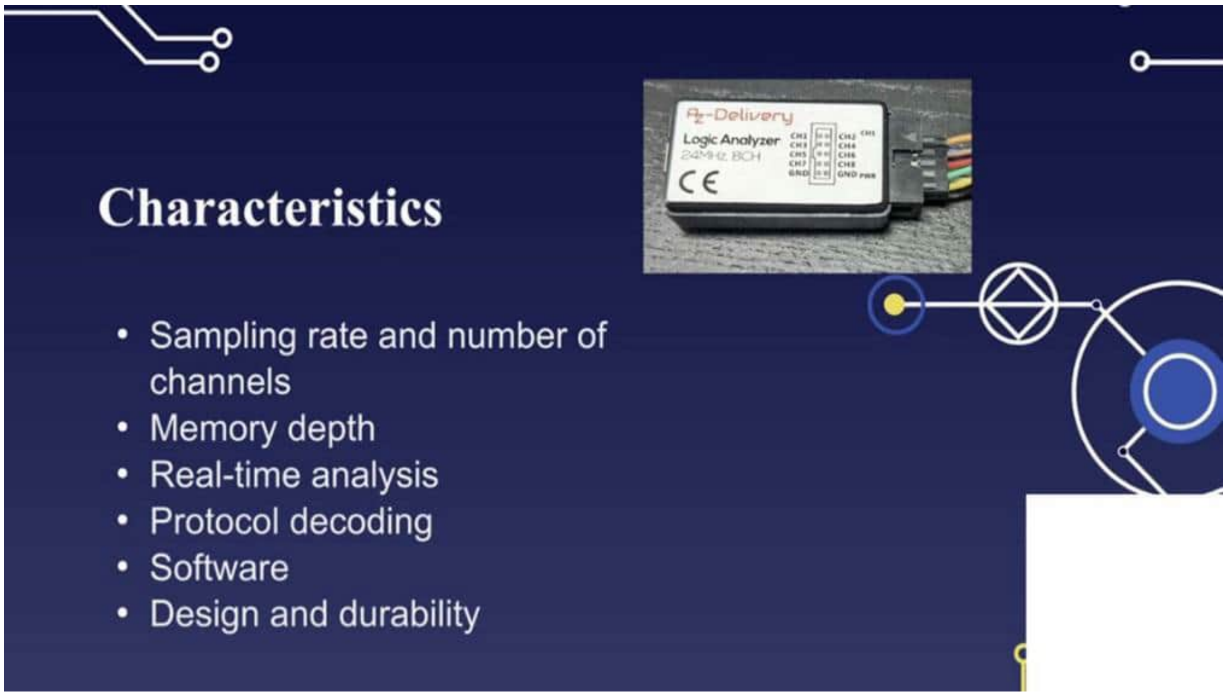
So it's more accurate, and you can granulate your information in much more detail. On the other hand, the logic analyzer is only able to convert the voltage into lead levels from 1 to 0. That means when we have 3.3 volts or 5 volts, which we normally get from our early inos and ESPs, Then this will be shown as in 1. And if we have no voltage, then it will be shown as a 0. And this is also what we talk about TTL.

This is the transistor logic level. And in this case, our logic analyzer is only able to get series and ones converted from our spectrum from the voltage. And here, we can measure, of course, also this kind of logic levers. but much more. And you also see more analog values as well.

So as you can see, much more detail, much, much more I would say is a little bit complicated, but it's a good handheld and good measurement device if you want to dig a little bit deeper in the world of electronics. What we are focusing on is investigating and analyzing all of the different kinds of protocols, and therefore, a logic analyzer will be here our best advice at a really good price. And later on, we talk about the features. The main difference is that you can also buy a logic analyzer for €10200, and then we'll receive the specs later on.

LOGIC ANALYZER CHARACTERISTICS

So before you buy a logic analyzer, you go through your mind and think, which one is the best one? Of course, always a cheaper one if it works efficiently. But there are main differences, and I would like to go through all of them so that you have a good overview. What is the best logic analyzer for you? So the first thing I think is one of the important points is the sampling rate.



Characteristics

- Sampling rate and number of channels
- Memory depth
- Real-time analysis
- Protocol decoding
- Software
- Design and durability

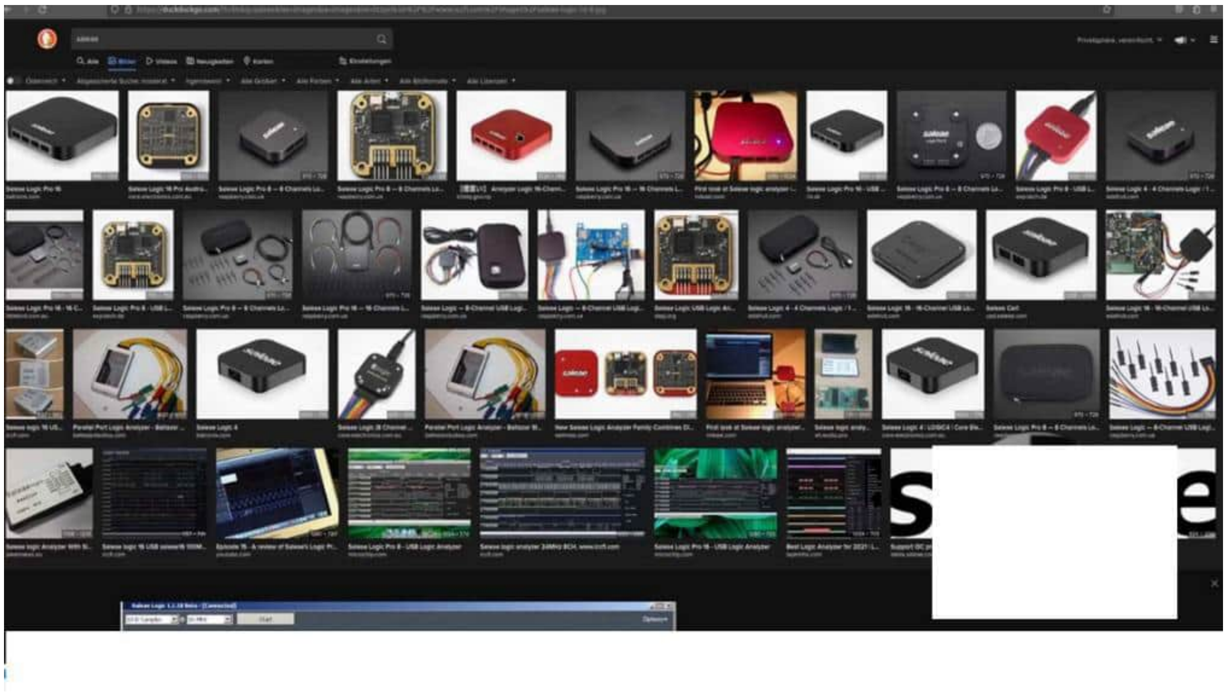
AZ-Delivery
Logic Analyzer
24MHz 8CH
CE

The sampling rate tells you how quickly the logic analyzer can capture and store digital samples of the signals that we are analyzing. Here in our case, we have 24 Megahertz. And in my opinion, this is absolutely enough for the hobby area. We can capture e squared SPI, UWART, One Wire. Also, some non standard sized for DHT and temperature sensors, etcetera, works absolutely fine and accurately.

And higher sampling rate, allows you a little bit also more precise capturing and fast changing signals. So the number of channels indicating how many digital signals can be captured simultaneously. And a logic analyzer can hear a very bright range. So now you have to think, okay, my ESP, etcetera, the adenos have much more and megahertz, maybe also the protocols. have it.

But as I told you before, with those protocols, which I mentioned, we have 24 Megahertz here. Absolutely. No problem. The number of channels. So here we have 8 is also absolutely enough because the maximum we are using in this Project will be, I think, 5 channels for SBI.

The other ones e squared c, u r does etcetera only needs 2 channels, one wire. Of course, one etcetera, and also this non standardized, I think also two channels. That's it. Plus we don't need a normal hobby area with the otherinos and ESPs. Memory depth.



Yeah. Memory depth is in point. how men, what is the amount of storage capacity of this logic analyzer? And often, this logic analyzer comes also with a device so that you don't really need, PC or notebook, etcetera. But in our case, we are just capturing the data with this logic analyzer and putting it into a Siggroke pulse view and part 2 will be our main, yeah, the main program where we're analyzing the signals afterwards and therefore also this could be neglected in my opinion.

Also, the real time analysis and protocol decoding, these are all software related issues in my case. And we will go through all of these points, but, this depends on which software you're using. We are using, and we come to this also later on, open source software, which is available for Mac OS Linux, nearly every distribution of Linux, and, of course, Windows. And therefore, it's easy to use, free to use, and we really use them. But I would like to show you here also an auto variant.

So one of the best logic analyzes on the market in my opinion is Salia. And those devices cost, I think it starts at $1.50. It's at a but those are really high precision and really reliable devices. And This Salia software has or the logic analyzer has also its own software. It's called Salia Logic. You can also download this logic software for free.

But if you take a closer look at the license of this software, you can see It's, attached that you have bought an original cell geologic logic analyzer. And therefore, we always conform regarding the license. Therefore, we are not using this as we are using the open source software. And also because we can also have an external decoder, and external decoder is really an important point because, with Python, you can adhere to your own programs and can, implemented in sick rock, pulse fuel. One point, maybe this is also something for you. This is a really, really, cheap product, and you see it in the whole handling, etcetera.

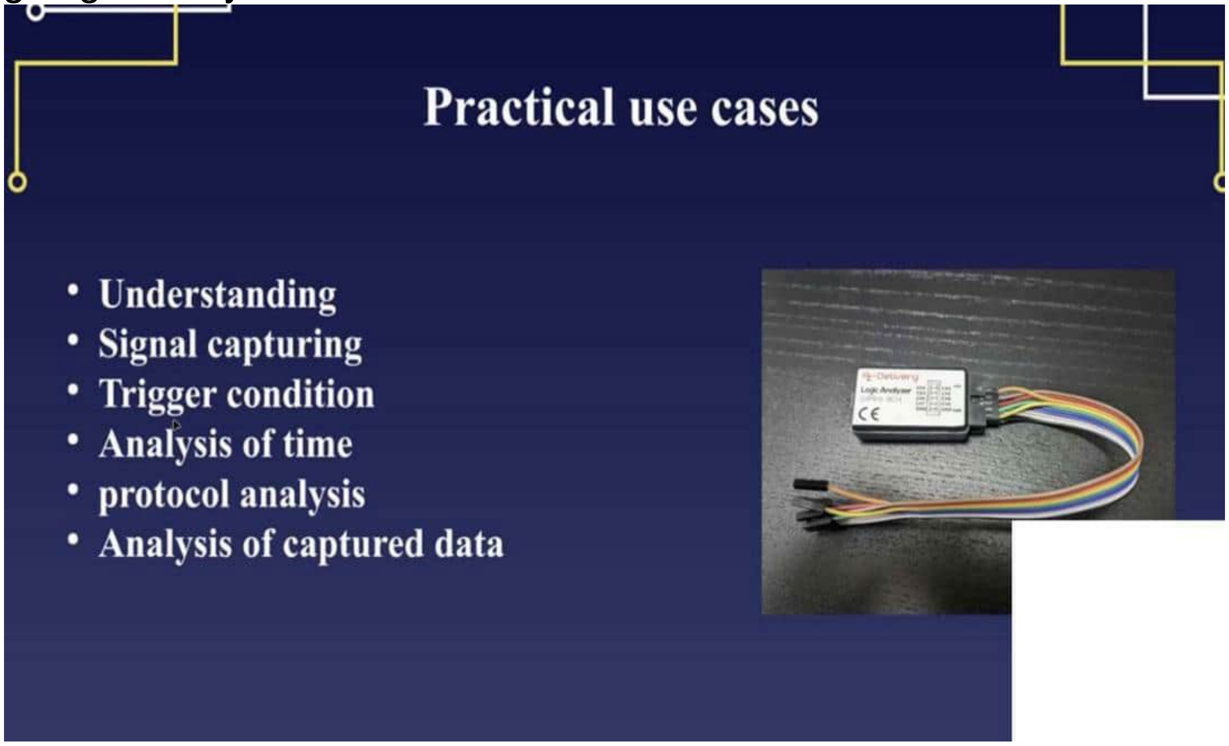
It's not very, really durable. And I think in a few months, etcetera, this will fall apart maybe, but maybe not. And as you can see here, this is just a good product design. So if you like product design, etcetera, then maybe you should avoid those cheap ones. If you are just wanting an logic analyzer, which is working and is cheap, then go for example from asset delivery.

PRACTICAL USE CASES AT A GLANCE

In this project I would like to give you an overview of practical use cases where you can use your logic analyzer in a hobby area. And we start with, of course , understanding. We often use some external libraries to get data from a sensor but do we really understand what is happening there? And also if we want to use sensors etc with for example an ATtiny or an ATmega where we have limited resources then it's often the case that we have to capture the data by ourselves because the external libraries are too big.

And therefore it's a really good way to understand once you have understood the principle you can fetch the data on your own. And also the signal capturing that you are able to know what is a logic level, how you can transform a logic level into ASCII code into real life sensor data etc. And this is really interesting because it gives you a deeper understanding how all of this ecosystem is really working together. Trigger connection.

Really fascinating point because often when you are searching for some errors in your electrical devices etc then you have to know when something is triggered and therefore we could also make an example later on that we start the measurement when something is triggered. And then you can see okay in this time area. This certain package will arrive then I have to change something etc and this gives you another perspective of the whole conditions of what is going on on your circuit.

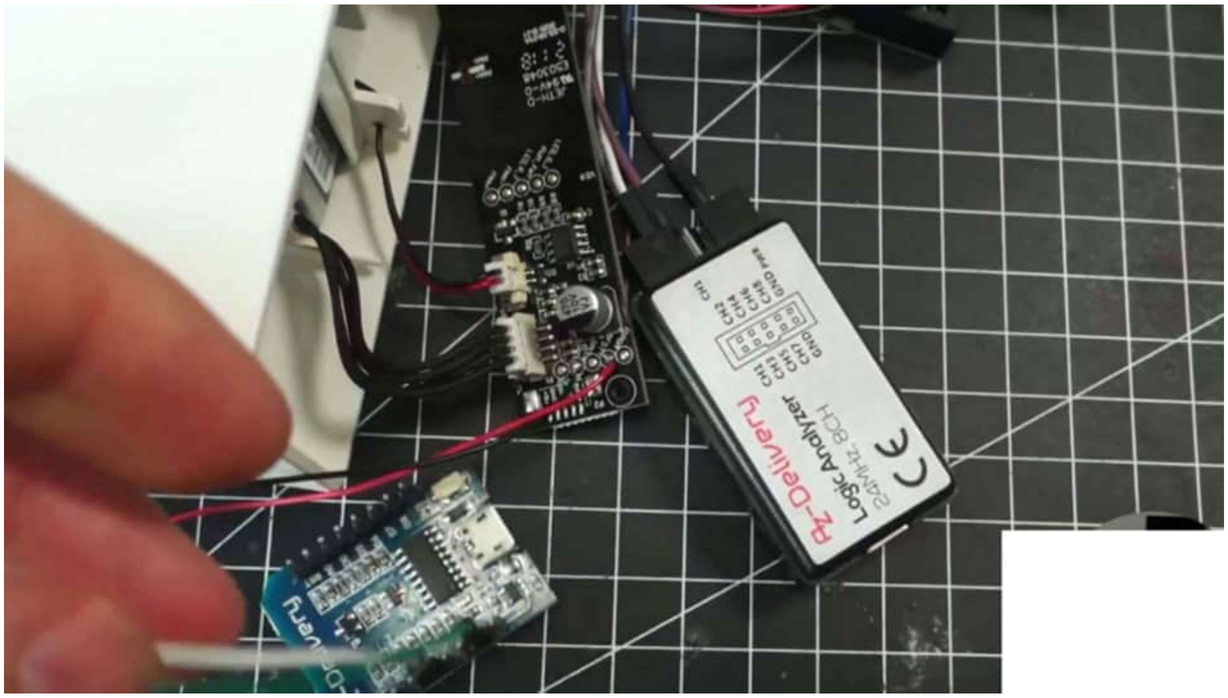


Practical use cases

- Understanding
- Signal capturing
- Trigger condition
- Analysis of time
- protocol analysis
- Analysis of captured data

Protocol analysis and we will go through all of these different kinds of protocols and you will see if you have once understood the main principle it doesn't matter what or which protocol it is because nearly the other approach is nearly the same. There are nuances but we go through all of them. And of course analyzing the captured data this will be our final point that we are able to fetch the data directly from the sensor without any external libraries.

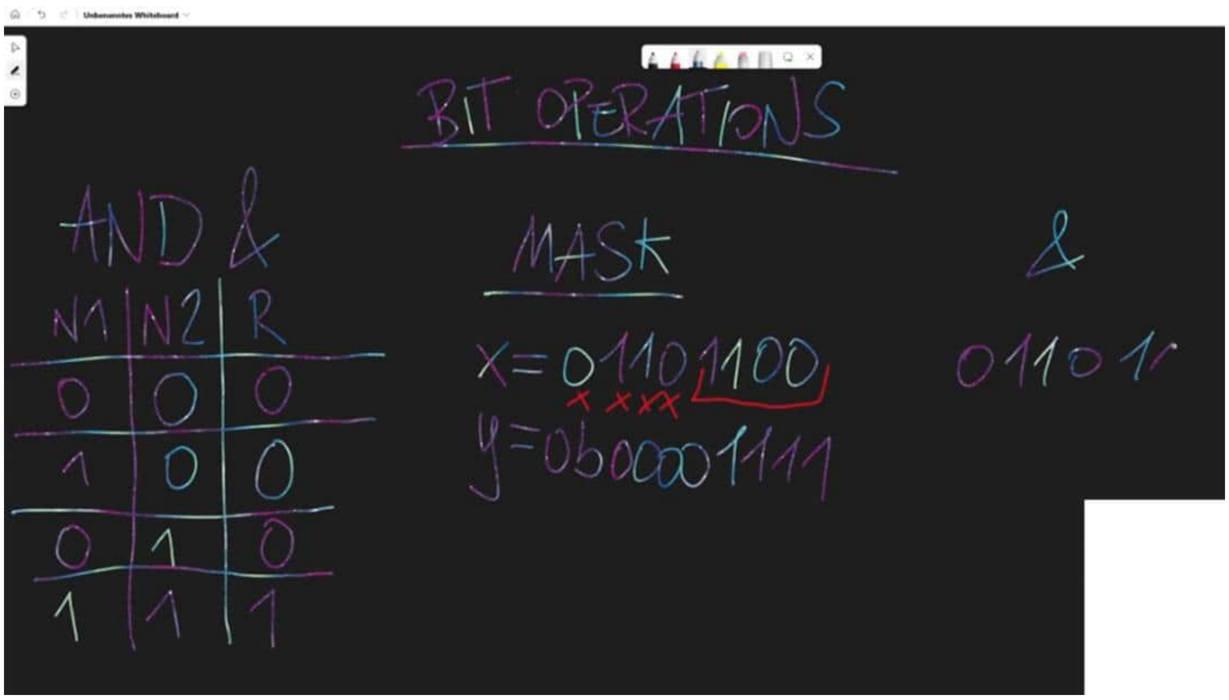
And to see what you can do with that so here is my sensor let's get here for example this one this is a CO2 sensor and it will measure various data but I just got the sensor and of course you can see which one it is but I see okay this is an SCL and an SDA so you can here think about okay it's an E2C and then you can fetch here the E2C address with a tiny sketch this is also one of the things what we do an E2C scanner and then you can capture the data what everything will be what we get from the sensor and in the data sheet then you can see how the data is processed and so you can make your own library because this library for example is so big that I can't use it for a tiny device and therefore this is an USB32 of course have enough space but not for the ATtiny for example.



Another use case is that this is an IKEA sensor it's an air quality sensor and when you open it up you can see here the PCB there are some points some looks like UART and therefore you are soldering here to jumper cables and connecting to the logic analyzer and this is exactly what we are doing then later on in the UART chapter and then we are analyzing what is this microprocessor sensor this microprocessor sending data from the sensor and we are capturing this data and try and reverse engineering and later on also process this data with an ESP8266 a smaller device and send this data for example to a database or visualize it in our home assistant and with all of these projects afterwards I think you have a very good understanding how the protocols are working what you can do with a logic analyzer and how to investigate your own electrical PCBs.

MASK WITH AND

Now we are in the second part of our basic chapter and I would go through some bit operations. And we're using all of this content later on in UART and E2C to get and capture data directly from the sensor. And also we do all of these inputs what we are going through in this chapter in the last two projects in the Arduino environment. So this is the theoretical part and then we do exactly the same examples in the Arduino environment.



N1	N2	R
0	0	0
1	0	0
0	1	0
1	1	1

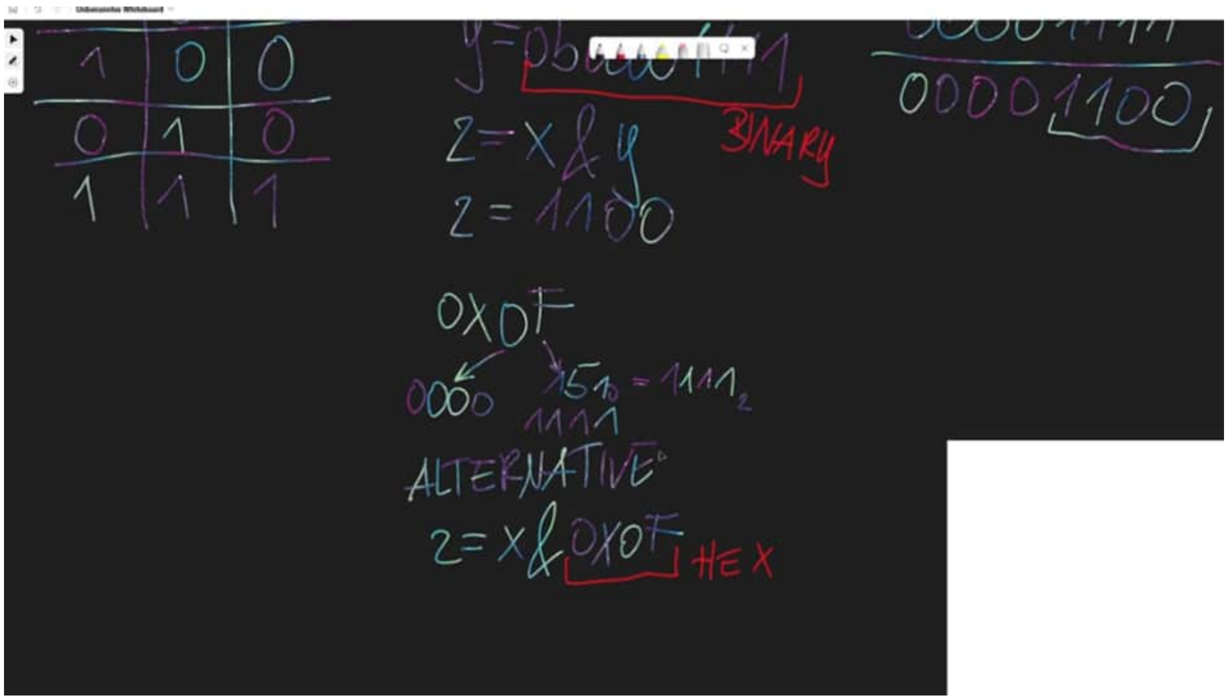
And I would like to start with the first concept and this first concept is the AND. And it will be used with the sign of the ampersand. Don't mix up the logic AND with the Arduino environment. So if you're using an IF condition you have the first condition and a second condition you're using two times this ampersand. At a bit level we are only using one ampersand. And the AND condition of the bit level means that we are having two numbers and each of these two numbers have to be true and then the result is also true. So let's check it in a truth table.

I'm sure a few of you know a truth table. It's some basic concepts but you may do it also in school. So we have here a number one then we have a number two and this is our result. So now when we have the first number and the second number these are two zeros and of course our result with the AND is also zero. If we have a one and a zero this will also lead to a zero. The same exact example is if number one is zero and number two is one then we have a zero.

And now you see the last example is if we have two numbers one then it also the result is here one. So this special case is absolutely necessary that we understand because why do we need such an input? Because we want to mask some numbers here. What do I mean with a mask? So for example if you have here a number I store it in the variable x. So just an example: some binary value what we get from a sensor for example. So it's a 0 1 1 0 and I have 1 1 0 0. So eight bits is one byte.

This is a typical order of what we are doing later on in all of this evaluation. And now the case could be and this will be exactly a case. I think in the DHT20 that we only want the last four bits. So this one should be deleted , this one and this one. And we do not do any string operations here. We make it in a mathematical way and therefore we could say here our second variable and we are indicating in the Arduino environment with a zero b that now a binary value is coming. And then we say four times a zero and four times the one.

So let's see what I want to do here. I'm writing here those numbers once again so that it is really really clear what I'm going to do here 0 0 0 0 1 1 1 and 1 so now the idea is so let me go here a little bit up that my face is here something covering as we did here in the truth table if we have 0 0 then it's a 0 0 1 0 0 0 and now everywhere where is a 1 in our original number there will be also a 1 so that means the last four digits will be 0 0 1 1 0 0 so what we did now is we are masking out the last four digits by making an AND operation with these binary level so in the Arduino environment we are doing later on x and y and the set value is 1 1 0 0 why is that so because the leading series doesn't need has any values so we only have here the last four digits and we are copy the digits because here we have the four ones and with this operation by each single digit... digits what we want and this is exactly what we are using later on in evaluating here our sensors one special case when you when we're investigating some libraries we see another variant of this so for example we see here some hexadecimal values and this 0x indicates there's an hex value coming and 0f means that we... f means



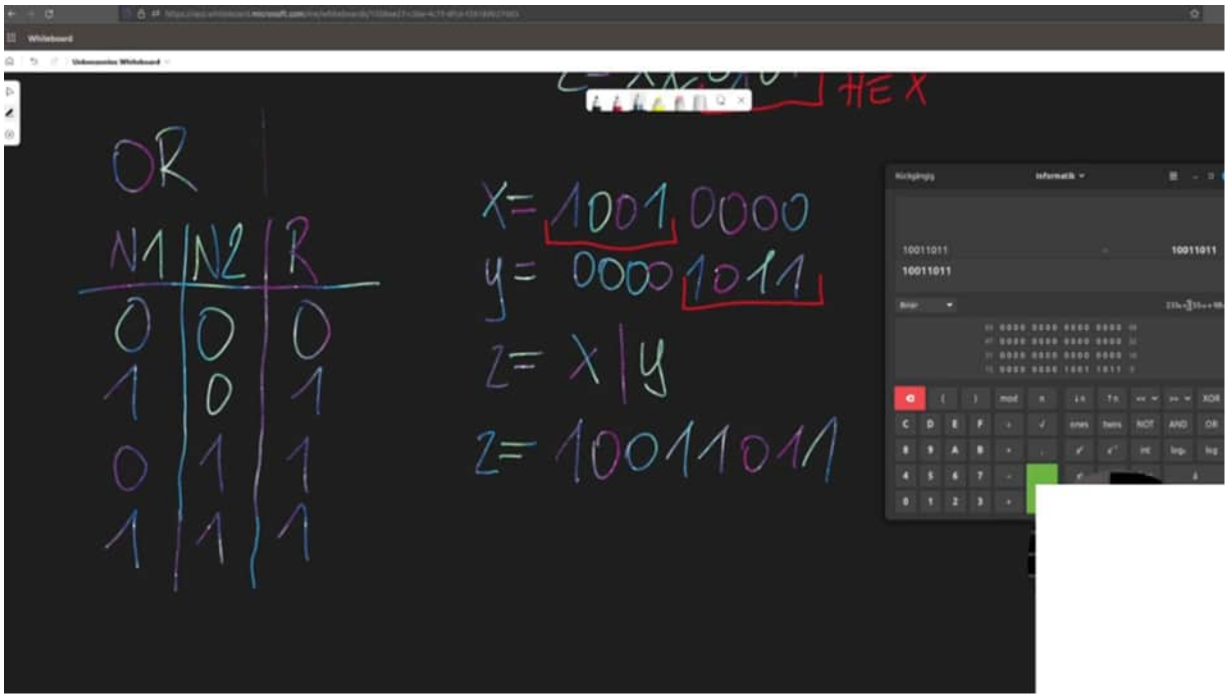
15 with the potential 10 and if we convert this to binary then we have here 1 1 1 1 why is that so because this is 1 2 4 8 12 14 15 and so we could also manage this alternative so let me write this here alternative set equals x and 0 x 0 f is the same is the same thing like we did here but it has a different approach because here we have an hex and the other was was binary binary so as you can see different approaches same output and this is necessary because maybe you are seeing those kind a little bit often because it's a shorter writing yeah and this is what i would say a bit operational level the end operation what we are using with masking.

MASK WITH OR

In the previous projects we talked about the and now I would like to introduce you also to the logical or with this vertical pipe. And also here don't mix it up with the logical or in the Arduino environment. So if you have an if condition with the first expression is something true or false or the second expression and then we are using two of these vertical pipes of this sign and the logical or a function in that way that we are comparing two values and each or one of the value is true then also the result is true.

So let's face with some truth table we have here number one as before then we have here number two and we have a result. So if let's give us some space here if we have a zero in the first number and a zero in the second number of course the result will also be zero. But if we have one and zero here, this leads us to a true result.

If we have here and true in the second number or one in this sorry second the second number is one or true both could could be used then it's also on one in the result and also one and one will lead to one or true to the result. Very good and now you're asking you know maybe why do we need those example and this is also a use case what we have later on because in the sensors often the result will be saved in 8-bit registers and one 8-bit register is not enough so we have to combine two of them so that we get one major number out of it.



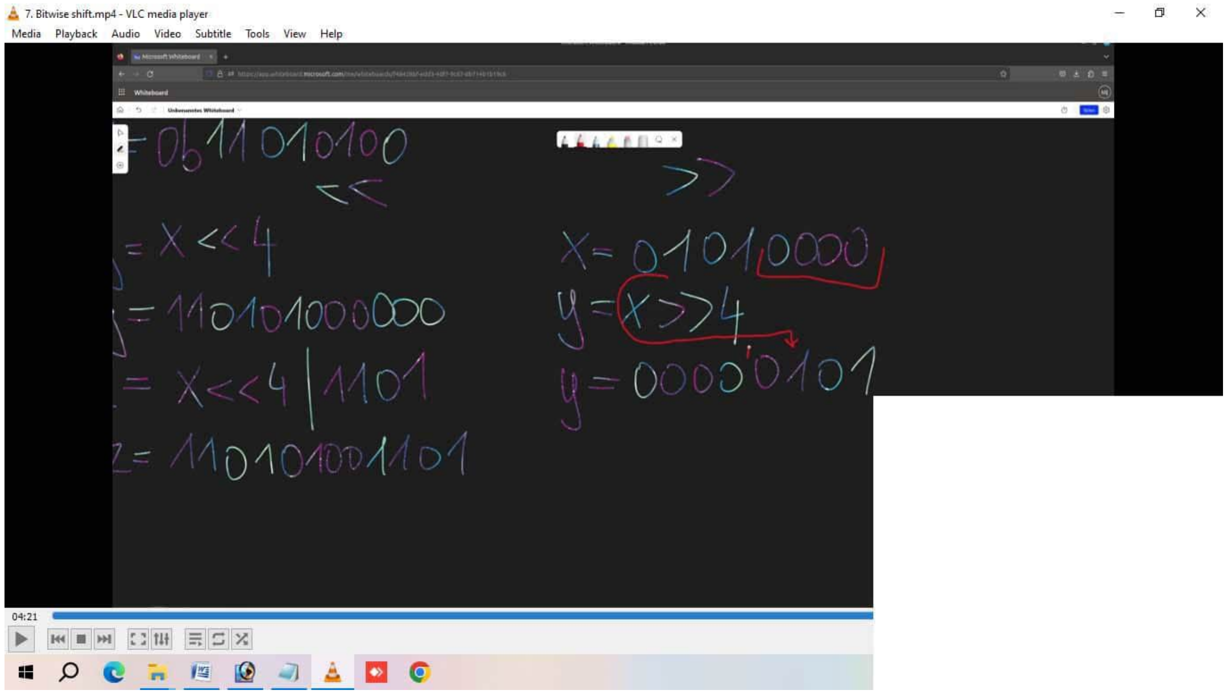
N1	N2	R
0	0	0
1	0	1
0	1	1
1	1	1

Speaker 1 (00:02:05) - So that could be that we have here the first number so the first byte could be one one and four zeros because we're shifting here around some bits we come to this later on we just take this for granted and have here a second number so that I can explain you what the OR operation is doing here and now I would like combine those two values that this will be here here this first part I would like to have in my number and the second part and this could be done with an OR operation so my new variable set will be x or y and this is be done let's write it once more so that we can compare with our truth table 0 0 0 1 0 1 1 and now we can compare with our truth table if there is an and one in our expression then we can do here the result also 1 0 0 is 0 0 there's a 1 1 0 1 and 1 and this will be now our new value and as you can see here now we combined both binary levels with the first expression and the last four bits and this is now our new value 1 0 0 1 1 0 1 1 and this could be then translated for example if you take the calculator we can change here to binary also under windows I think you have to change here to scientific and then you can also type in here binary values so 1 0 0 1 1 0 1 1 and then you have here the potential of 10 or we can change here to decimal and then we have the value 155 and this will be done with two registers a little bit of bit shifting and then with the or expression yeah with this operation we are using later on also to calculating here some values and I think now we have the two major operations with the and and the or of course there are a few more with XOR and NAND etc but we are just keeping it simple because this is what we are using later on in the other chapters.

BITWISE SHIFT

One more concept we need to know before we can start with the practical example, and this will be the bitwise shift. So a bitwise shift could be, for example, we're starting right again with one variable, and this variable holds a binary value from example 1101, and then we have 1 00. So let me see. Is everything good? Okay.

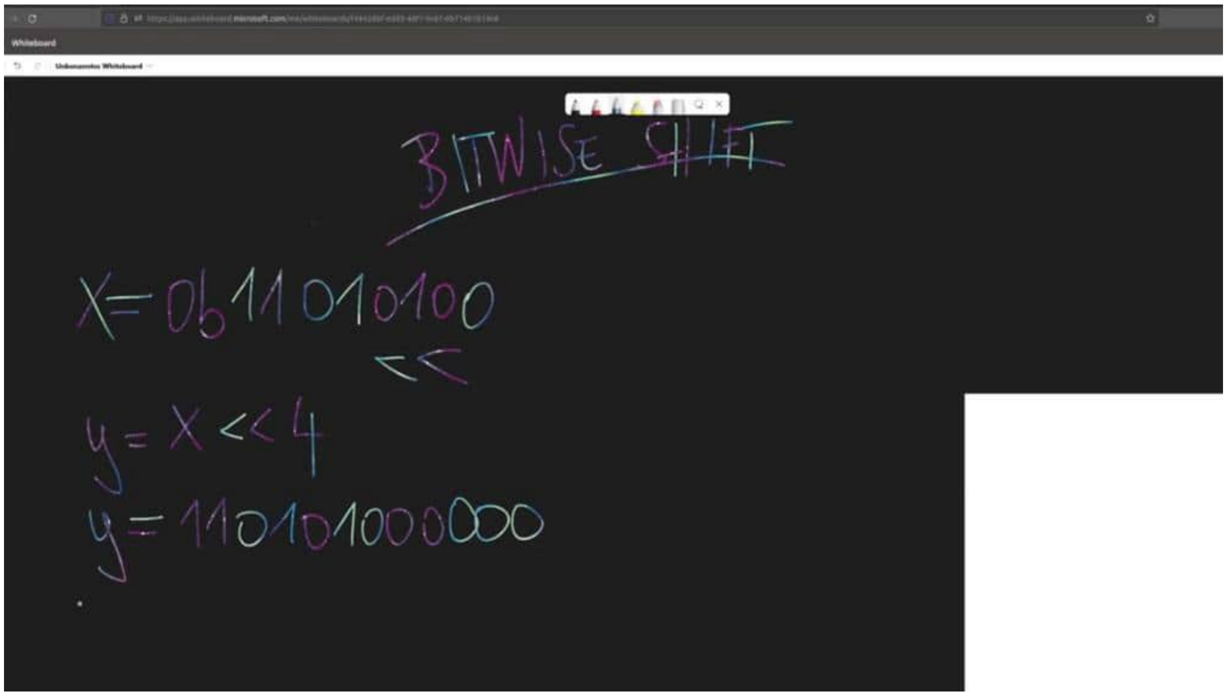
And now I would like to shift here to the left. with those 2 signs. So y equals to X, then I would like to pitch shift to left and how many spaces I would like to shift. And for example, I would like to shift four times to the left. That means now my new y level could be, 11 010100.



And now four times the new is wrong because we shifted it to the left. A few things to that. We have to change the wire here. Now to binary. It should be long or something like that because a binary can only hold 8 bits.

So if you're not changing the data type here, we are not adding 4, or, a pan tier 4 of these zeros. because this will be shifted to the left. So those 4 will be deleted, and this will be the new sign. But we want to have a long because the practical example here is that we're combining 2 registers from ABSENCE. So for example, we're reading the first eight bits, then the last four bits sort of only half of the and register, which is in the DHT 20 example.

And, therefore, we only need the first four bits, and this will now be our new number, for example. And this, you can also write another variable so you could say for example, x, bit shift, 4 times, and then we could use an or and I would like to add here those 4 values. And now my set value could be Zed equals 1101 011 0, 0, and now the 4 last digits, and this will now be my new number with this expression. And this is exactly what we are doing later on in our evaluation. So it means x, pitch shift to 4 so that we get those values.



And then we're making an or with those 4 digits and adding those 4 digits with this operation to this number. If it's an example, we have here, of course, also the second example with shifting it to the right. same principle, a little bit different in the output. So let's see. We have here our x And we have 0101, and we have, yeah, 4 of the 0s. And now we are seeing our y variable is x pitch shift to the right four times So what the result will be now is 0000.

The new 40s And then adding the number here. So what have we done now? We have shifted those 4 to the right, that means they will be deleted. Nevertheless, what kind of data type we have. And those 4, we are putting here into the place of this one.

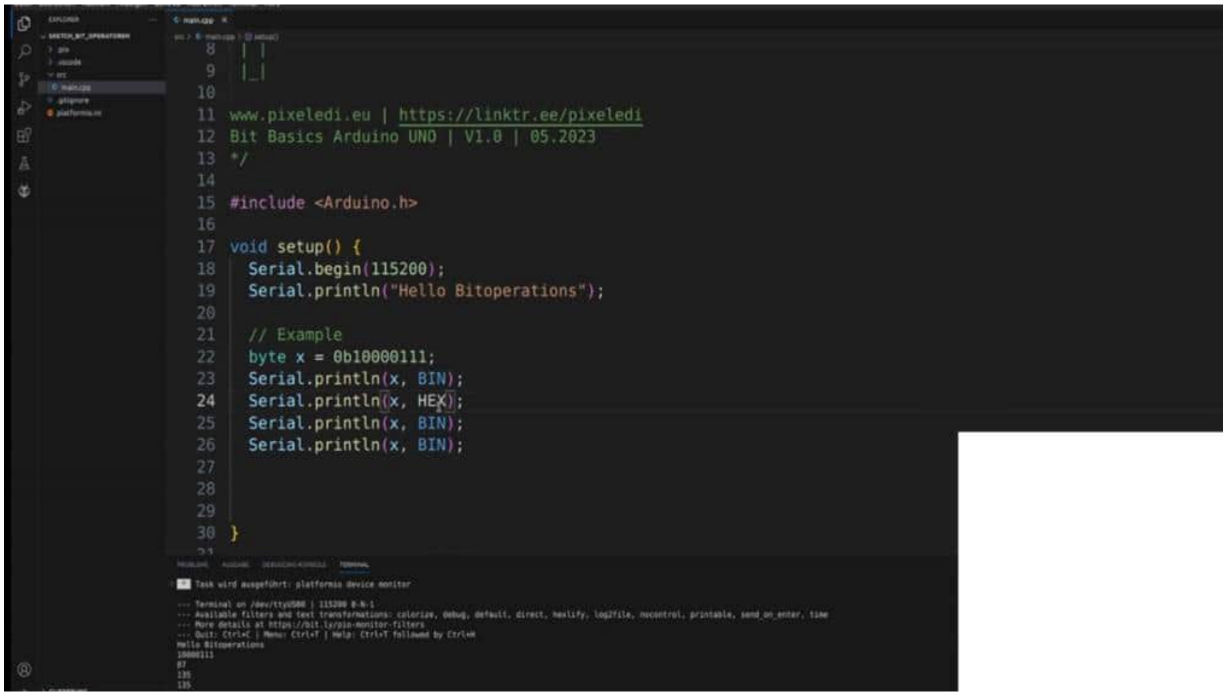
And those 4 will be here. And this is exactly what we're doing here, and the result is now because we are deleting all of the series in front because they have no value. And then if it's 1, 2, 4 equals to 5 in the 10a, 10 potential. Okay. So now we are also aware of the concept of bit shifting to the left and to the right.

We also have some use cases here later on. especially this one will be used because this one, we're using later on also for deleting some numbers, etcetera. And now we can use it also later on in the arena sketch.

PRACTICAL EXERCISE EXAMPLES 1

Let us start now in the practical part. I'm using a platform you own with Visual Studio Court, and if you like the audio ID, that's completely okay. You can use all of my codes also in the audio ID. I've made a project on how you can transfer the code. And now we can start with the bit operations.

I'm using an Arduino Uno and clone it. And this is where we execute the codes now. And now we are starting here in the setup part. We don't really need any external code because I only want to use it in the setup because the code should only execute once. So now, we start with the first example, and we are creating the co step by step together so you can follow me along with all of the examples.



```
 | |
 |_|

www.pixeledi.eu | https://linktr.ee/pixeledi
Bit Basics Arduino UNO | V1.0 | 05.2023
*/

#include <Arduino.h>

void setup() {
  Serial.begin(115200);
  Serial.println("Hello Bitoperations");

  // Example
  byte x = 0b10000111;
  Serial.println(x, BIN);
  Serial.println(x, HEX);
  Serial.println(x, BIN);
  Serial.println(x, BIN);



}
```

```
Task wird ausgeführt: platformio device monitor

--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocontrol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
Hello Bitoperations
10000111
87
135
135
```

So I'm creating here a new variable in the data type with byte, and we are indicating that this is really a byte with the 0b. And now 8 bits can follow. And for example, we are saying here the following number. So this one with 1000 in the first four bits and then 0 111 in the second 4 bits. 4 bits are also called nimble, and therefore, it's common, really common to, to use half of the bits also.

Yeah. Okay. And then I would like to show you how we can print out such numbers. So for example, you could say print And this is also what we are using later on because I want to print out, what is the binary value. And, also, I would like to have the same value in hex as in decimal.

And what will happen if I'm not indicating here the second So then let's upload the quote. And what we got, we the handler operations 0 prints with the baud rate 115200. Then I got my binary value as we type it in on line 22. Then I got the hex value, the decimal value, and also a decimal value because if I'm not prompting here, second part of the, parameter of the function, then I will also get the decimal value So far, so good. We are using this example very often because it's a kind of debugging and I really like it.

So, therefore, just commenting out all of the lines before so that we go on do not get an inquisition about the reused variables with control and the hash example, you can make here in Visual Studio Code, commenting out many lines at once. Then I would say we start with the first and and or example. So for example, we have here invite x, just with 3 bits that we can see here in the example, 111 And now I would like to make here an, yeah, for example, we make here value 1. and value 1 equals to X And Y, and then we copy this 1 and say byte value 2, is x or y just to see what happens if we're printing out those values Then we say well 1 in binary. Copy this 1 and say well, you too.

So just before we hit your upload, we think about what will happen. And if you can reproduce the result. Then we upload the codes, and we're getting out here, halibut operations, and only one radio. Why is that so? Have you expected those results?

Because when we are making our end operation here, we are always getting 0 because 010101 is 0 and 0 is not printed out because it's just 0. And here, we're getting the value, from our own. This is three times the one. So a little bit of, funny experiment. Then next example, what we're gonna do is we are creating a new byte here, and we'll call it wall 3.

And, before we do that, we are needing here some new variables that called bytex2 equals 0 BM. And now we have 1000, and then we have white and this I would like to have in this case. So we have here 10 and also here so that we can get a proper example, then we're making an order here, I would say, with my tool. And now we are printing out this example in binary and this should be a value 3. And before we hit upload, just, go room, you mind what will be the finished results.

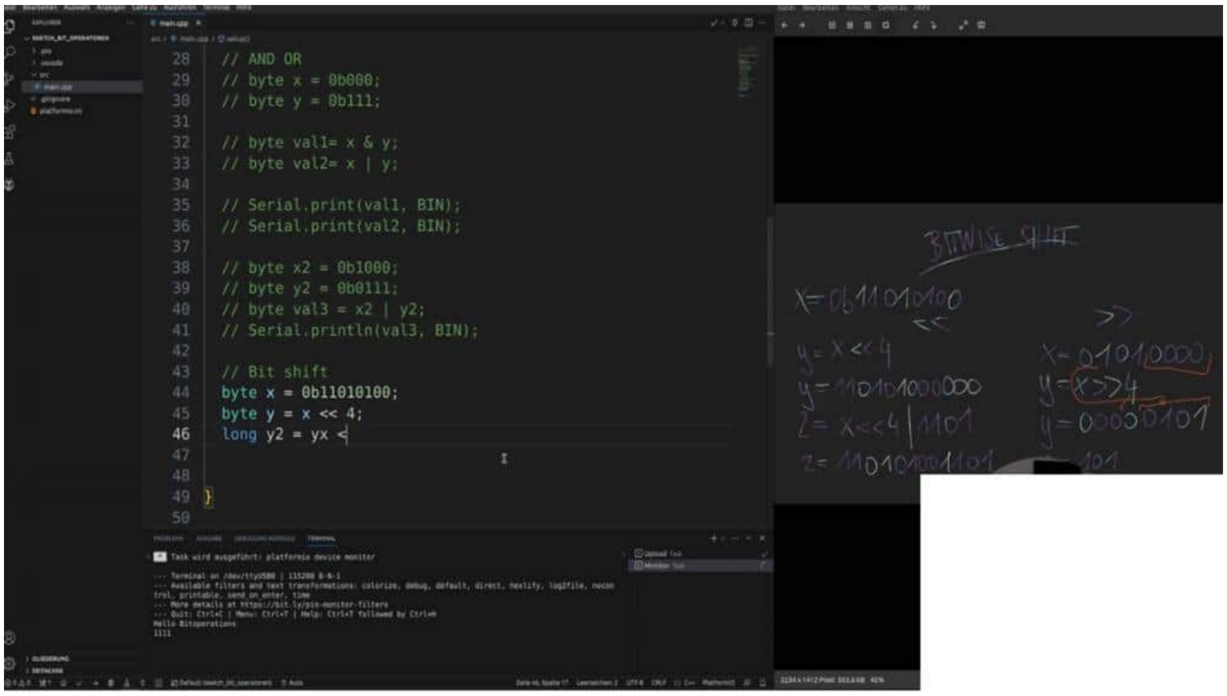
And, of course, it's an all operation, and we have here on both sides, and each digit on 1, and therefore, we got here the result 4 times 1. Nothing that we don't expect. So now I would say we are trying to solve the wise operations, what we did in our theoretical parts. And therefore, we are starting with the new x. So let's indicate here the example, pitch shift, the byte x 0 b, then I copy here the values.

This is the value from the left side. Now I would say the byte y is x times 4. So we do it, firstly, in the byte way. And then I would say long y2isx bit shifting 4, and then I say long. set is x, but this is also an example so that we could do it in one term.

So now we're printing our different kinds of values here. So we have first the y as some binary, then we have the y2 as some binary, and I would like to have here set value as a binary. So as you can see here, this is the 3rd example. Let's upload the code. So can you reproduce the results? The first example and what we have here are those numbers.

Why is that so? We have here the data type bytes. And this is our saved X And now we are shifting four times to the left. That means we are creating here the value 0000 and shifting all of them to the left. But in and by it, it only has room for 8 bits.

That means that those 4 values will be deleted, and we only have those values. And the first 0 will be deleted, then I have 1, 2, 3, 4, 5, 6, is the correct 1, 2, 3, 4, 5, 6. And this is the reason why we have this value. In the second example, I changed the data type to long and I shifted the x to the left with 4 digits. And as you can see now, we have exactly the number, what we have here, but also the 4 digits that we added to this number.



```
// AND OR
// byte x = 0b000;
// byte y = 0b111;

// byte val1= x & y;
// byte val2= x | y;

// Serial.print(val1, BIN);
// Serial.print(val2, BIN);

// byte x2 = 0b1000;
// byte y2 = 0b0111;
// byte val3 = x2 | y2;
// Serial.println(val3, BIN);

// Bit shift
byte x = 0b11010100;
byte y = x << 4;
long y2 = yx <

}
```

```
Task wird ausgeführt: platformio device monitor

--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocon
trol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
Hello Bitoperations
1111
```

And this is what we are using later on because we want to have the first register and the second register definitely had to make the space that we can add the second number because it's different. Different approaches. Maybe we should add 2 registers to each other. Maybe we have to combine it. This is what the datasheet tells us, and therefore, we're going through here.

SpaceX. And the third one is all in 1. That means we'll make a bit of a shift in and long. 2 for, 4 bits, and we made an or we made an or. That means the last 4 digits are those values and the first 8 digits are those values.

So art is correct. Yes. And this is a common way to approach how we use the results from a sensor, for example, in a way that we're using here the bit shift and then also, in the same line, the or operation. So now we can proceed with our bid shifting operations. And we have one left, and this is the bit shift to the right.

So let's focus here. Can I place it a little bit up? I don't think so. So I have to zoom out. So, now, okay, pitch shift to the right.

That means we have to use it here. Bit shift to the right. Our byte x, for example, is 0b, this one, And now we want to have the byte y is x, pitch shift to the right four times. And now we're printing out our Y, 0 print line Y in binary. Let's see.

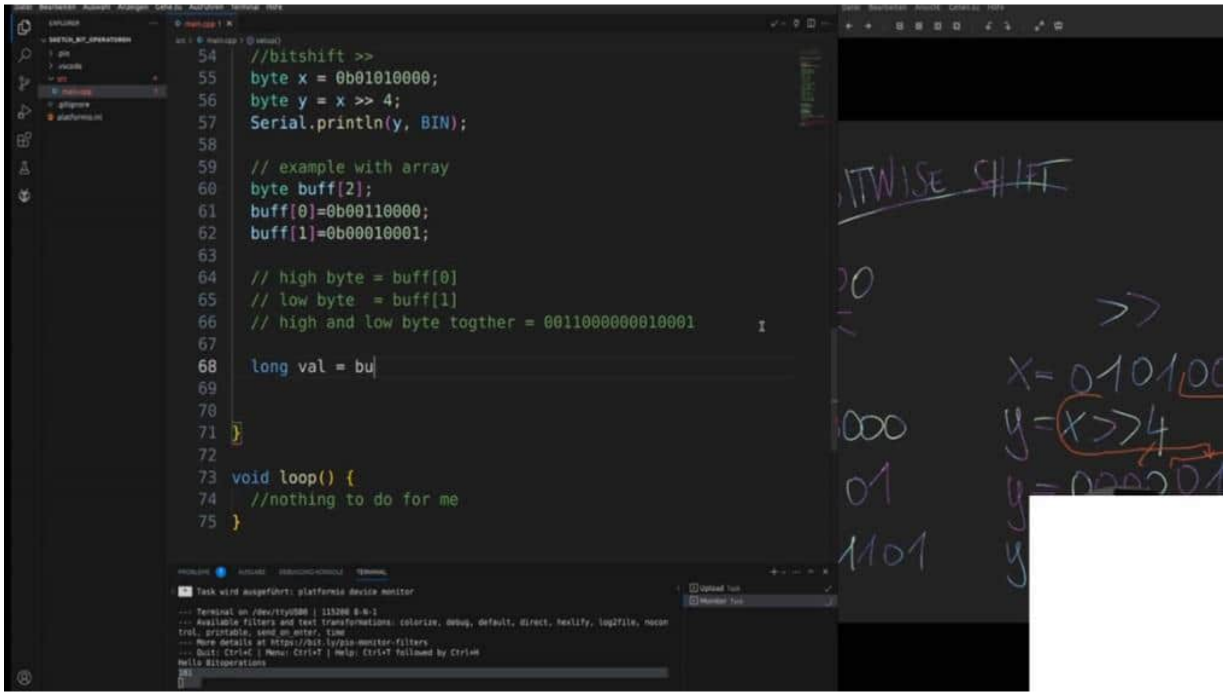
And as our calculation was, let's do it a little bit on the big side here, we have shifted to the right. That means all of these will be to the right and replaced with 0s, and all the 0s are deleted because there is no value inside. So we have 010, and this is exactly what we get here out of our operation. And now I would like to give you and show you an example what is really also common when it comes to evaluating, the values out from and sensor. because often we are using the output with serial read or with e squared t read, and then we are saving the data inside an area, and then we are fetching and working with the area.

So how could we do that? Next example, with an array. So we have here an byte error, and this is called buffer, for example, with the place of 2 arrays. Then now I am making a static because normally we are using here in 4 and reading flags out the values out, but now, I place in the index 0 the first value on just for training purposes. And then on the 1st index.

So we have 2, 2 arrays with index 0 and index 1. And the second one has this one. And now with this buffer, what we would like to do now is we want to work with it, and we want to create a high end low byte. So that means for example, let me show you, the high part, just for explanation, The high byte is the buffer 0. And this is something that stands in the datasheet.

And the low byte, for example, is in the buffer 1. So how can we know what should be the result now? The result is high and low bytes together. So not operations is the first value and the second value. This should be the result.

And how can we do that? So for example, I could say long value is buffer, 0, and just hold on here a little bit and try to think how you could create those numbers with Bit Operator. because when you think on your own, it's much more that you comprehend all of the concepts. And now what we are doing is we are creating space. We are shifting eight spaces to the left because it's long, We are not deleting everything.



```
  //bitshift >>
  byte x = 0b01010000;
  byte y = x >> 4;
  Serial.println(y, BIN);

  // example with array
  byte buff[2];
  buff[0]=0b00110000;
  buff[1]=0b00010001;

  // high byte = buff[0]
  // low byte  = buff[1]
  // high and low byte togther = 0011000000010001

  long val = bu


}

void loop() {
  //nothing to do for me
}
```

```
 * Task wird ausgeführt: platformio device monitor

--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocon
trol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
Hello Bitoperations
101
```

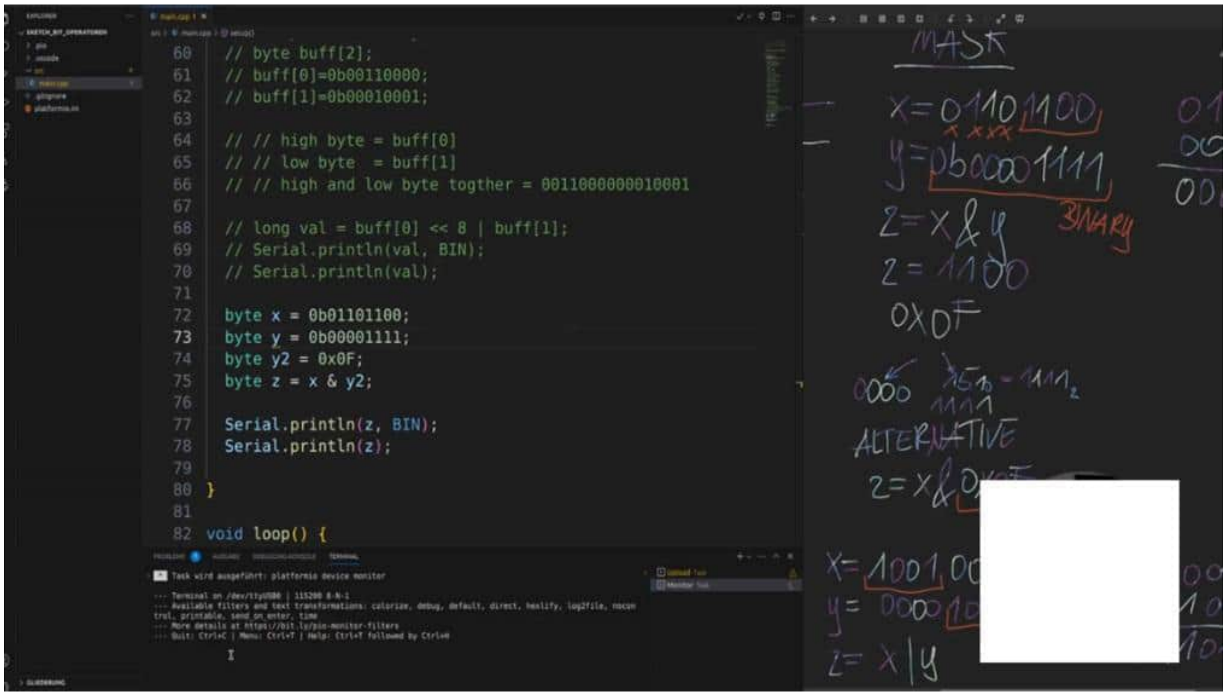
We are appended, and then we are making a logical or with the buffer size of 1 because this is the low byte. And when we are printing out now, the value of value in binary and also in decimal. This could be our desired value, what we are getting from a sensor, like the pH 1750, 1750, I think the light sensor. So this is one before. We don't need this one.

So this is now our combined value. As you can see, the first two are printed here. This is this one with a high bite. And this will be, indeed, tomorrow, those values from, 12,305. And this is a way how we can access data from 2 or 3 or more registers in an 8 bit space, 1 byte space, and we can then have higher values like those one, as I show you here.

PRACTICAL EXERCISE EXAMPLES 2

And as a last example, we of course want to check if our theoretical masking was right or not or correct. And, therefore, we start with spike x equals. So this, I would like to copy and paste because I'm not really good at typing the serious ones. So this should be our first, is that right? Yes.

And here 0011, this is the y part. Then can I zoom in here with yes. So now I would like to create the byte here. The set is x and y, and we're printing out 0 print line zed. And let's take a look.



```
    // byte buff[2];
    // buff[0]=0b00110000;
    // buff[1]=0b00010001;

    // // high byte = buff[0]
    // // low byte  = buff[1]
    // // high and low byte togther = 0011000000010001

    // long val = buff[0] << 8 | buff[1];
    // Serial.println(val, BIN);
    // Serial.println(val);

    byte x = 0b01101100;
    byte y = 0b00001111;
    byte y2 = 0x0F;
    byte z = x & y2;

    Serial.println(z, BIN);
    Serial.println(z);

}

void loop() {
```

```
Task wird ausgeführt: platformio device monitor

--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocon
trol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
```

If everything works as expected, we should get the value. 12 is 12 right. We have here 1, 2, 4, and 8, 8, and 4 is 12. Sounds good to me. We can also then give out here the binary and as well as the decimal.

But before we do that, I would like to show you the second variant that is called with an 0x at those values with the hex value 0f. And now we are creating here. Those values should be exactly the same result as we did here. Let me see. Here we go. here with the binary value so we can replace it both.

Then we have some ores left here. Therefore, are they the same values? No. Biotech is 01101234. and also he applied for 1111.

Set is xory And then we have a serial print. There we go. and tack and here, our pie, oop. I missed this one, and now also here and byte is missing, but now it should work. Then upload the code and the result should be an 11.

It's not correct because I missed something. I have to enter the 0b in front because otherwise, it will be converted from an integer to binary. And now it's a binary value, and now we should have, yeah, the same exact results as before. And this is what we calculated. Not really because I made a mistake here.

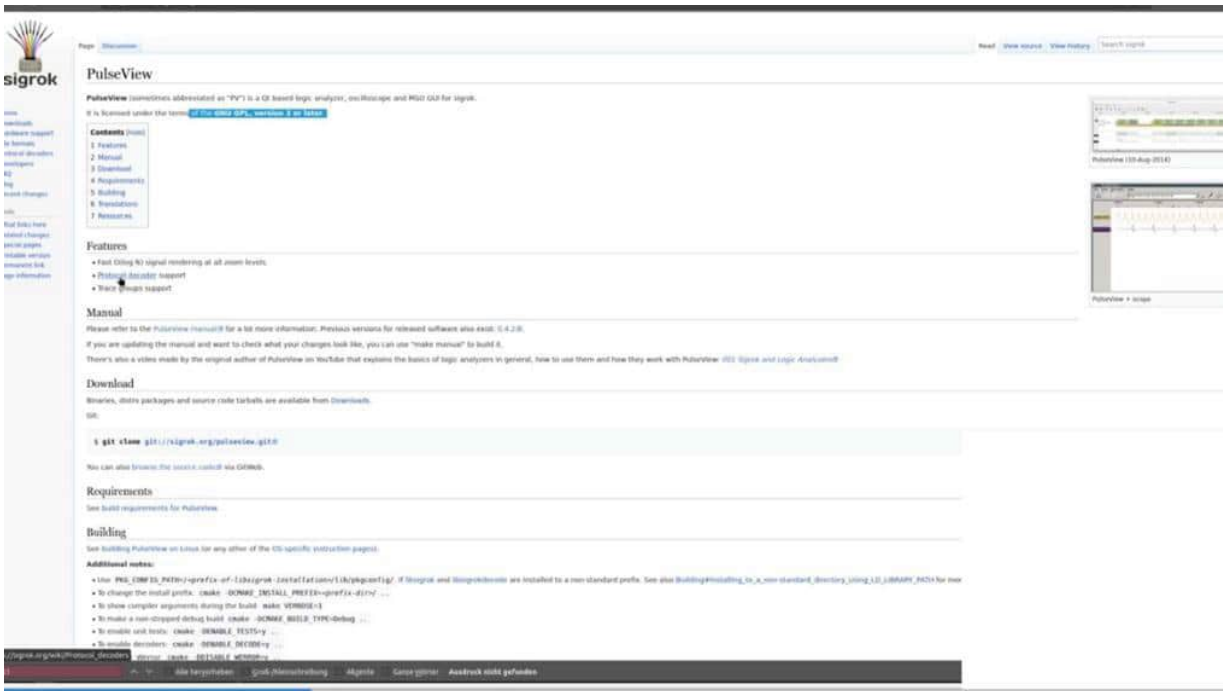
but now the value should be as here. Now it looks the same. Yes. Okay. An error in and the byte.

y, but now we have the same examples. We have to check if our theoretical parts are correct, and it is. And now we are well prepared because we did a lot of, or a few examples in the audio environment, and now we can jump right into Paul's view.

WHY PULSEVIEW

We started this chapter by discussing why we should use pulse fuel from sickrock. And one of the first points is it's open so I really love this whole fuss philosophy, free and open source. It means as you see, you can see in the license, we can also use it for commercial purposes. And, also, we can add our personal programs and decoders to it. But to this, we'll come later.

PulseHire has a really good, graphical user interface as well as a command line interface, but we only use the GUI here. And as you can see here, it's really, really easy in the overview. We do not have too many functions and this is what I think it's really an advantage to other software from logic analyzers, like logic, etcetera, because it keeps you to the main function of what we want to do, and this is we want to decode the signals. And, we can jump you a little bit back to the protocol decoder. Sigrok supports a lot of internal decoders.



sigrok

PulseView

PulseView (sometimes abbreviated as "PV") is a Qt based logic analyzer, oscilloscope and MSO GUI for sigrok.

It is licensed under the terms of the GNU GPL, version 3 or later.

Contents [hide]
1 Features
2 Manual
3 Download
4 Requirements
5 Building
6 Translations
7 Resources

Features

- Fast O(log N) signal rendering at all zoom levels.
- Protocol decoder support
- Trace groups support

Manual

Please refer to the PulseView manual for a lot more information. Previous versions for released software also exist: 0.4.2.

If you are updating the manual and want to check what your changes look like, you can use "make manual" to build it.

There's also a video made by the original author of PulseView on YouTube that explains the basics of logic analyzers in general, how to use them and how they work with PulseView: [illegible]

Download

Binaries, distro packages and source code tarballs are available from Downloads.

Git:

```
$ git clone git://sigrok.org/pulseview.git
```

You can also browse the source code via GitWeb.

Requirements

See build requirements for PulseView.

Building

See Building PulseView on Linux (or any other of the OS-specific instruction pages).

Additional notes:

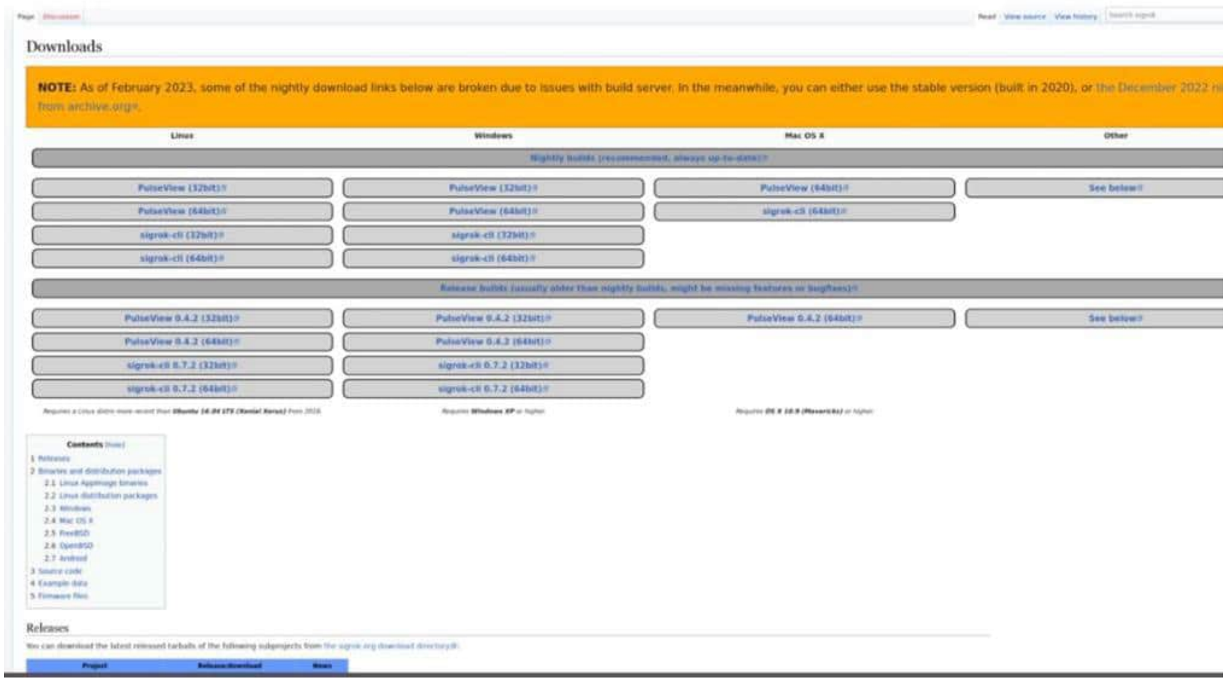
- Use PKG_CONFIG_PATH=/<prefix-of-libsigrok-installation>/lib/pkgconfig/ if libsigrok and libsigrokdecode are installed to a non-standard prefix. See also [illegible]
- To change the install prefix: cmake -DCMAKE_INSTALL_PREFIX=<prefix-dir>/ ..
- To show compiler arguments during the build: make VERBOSE=1
- To make a non-stripped debug build: cmake -DCMAKE_BUILD_TYPE=Debug ..
- To enable unit tests: cmake -DENABLE_TESTS=y ..
- To enable decoders: cmake -DENABLE_DECODE=y ..
- [illegible] cmake -DDISABLE_WERROR=y ..

And when we haven't found anyone that's suitable for our signals, Then we can program with Python by ourselves, or we can find someone, for example, on GitHub. And this is exactly what we are doing in the, I think, 1 wire chapter. We are downloading an external decoder, which was created by someone in the community, and this is what you can do with open source software. Use it and find other software. absolutely cool. Yeah.

Cross platform, supportive or compatibility because this is also what I like in the sickroke environment, you find here analytics version for nearly every, this Revolution, Windows, and, of course, Mac or Mac OS. And if you have trouble downloading the actual version here is also a reminder. Just use the older version, for example, the 0 point 4.2. This is the version I use. I think it's an app image for Linux.

So, therefore, just download this one. And, of course, on the Windows and Mac OS just downloaded, install it right away. This is not really something, rockets sense. And as you can see here, there's also a command line tool, but this is not what we cover here in this online course. Yeah.

And then what also is really, really necessary that we check on is the hardware support. If you buy a logic analyzer, keep an eye on if the logic analyzer is covered here. In my case, this is the ACET delivery. You can get it on Amazon for really a few bucks, and it's covered here. But, also, all of these cheap Chinese clones from eBay, etcetera should work because there is the same chip inside, but I will show you how you can, choose the right hardware here in Palmsville later on.


Page Discussion Read View source View history Search sigrok

Downloads

NOTE: As of February 2023, some of the nightly download links below are broken due to issues with build server. In the meanwhile, you can either use the stable version (built in 2020), or the December 2022 ni
from archive.org.

Linux	Windows	Mac OS X	Other
Nightly builds (recommended, always up-to-date)			
PulseView (32bit)	PulseView (32bit)	PulseView (64bit)	See below
PulseView (64bit)	PulseView (64bit)	sigrok-cli (64bit)	
sigrok-cli (32bit)	sigrok-cli (32bit)		
sigrok-cli (64bit)	sigrok-cli (64bit)		
Release builds (usually older than nightly builds, might be missing features or bugfixes)			
PulseView 0.4.2 (32bit)	PulseView 0.4.2 (32bit)	PulseView 0.4.2 (64bit)	See below
PulseView 0.4.2 (64bit)	PulseView 0.4.2 (64bit)		
sigrok-cli 0.7.2 (32bit)	sigrok-cli 0.7.2 (32bit)		
sigrok-cli 0.7.2 (64bit)	sigrok-cli 0.7.2 (64bit)		
Requires a Linux distro more recent than **Ubuntu 16.04 LTS (Xenial Xerus)** from 2016	Requires **Windows XP** or higher	Requires **OS X 10.9 (Mavericks)** or higher	

Contents [hide]

Releases

You can download the latest released tarballs of the following subprojects from the sigrok.org download directory.

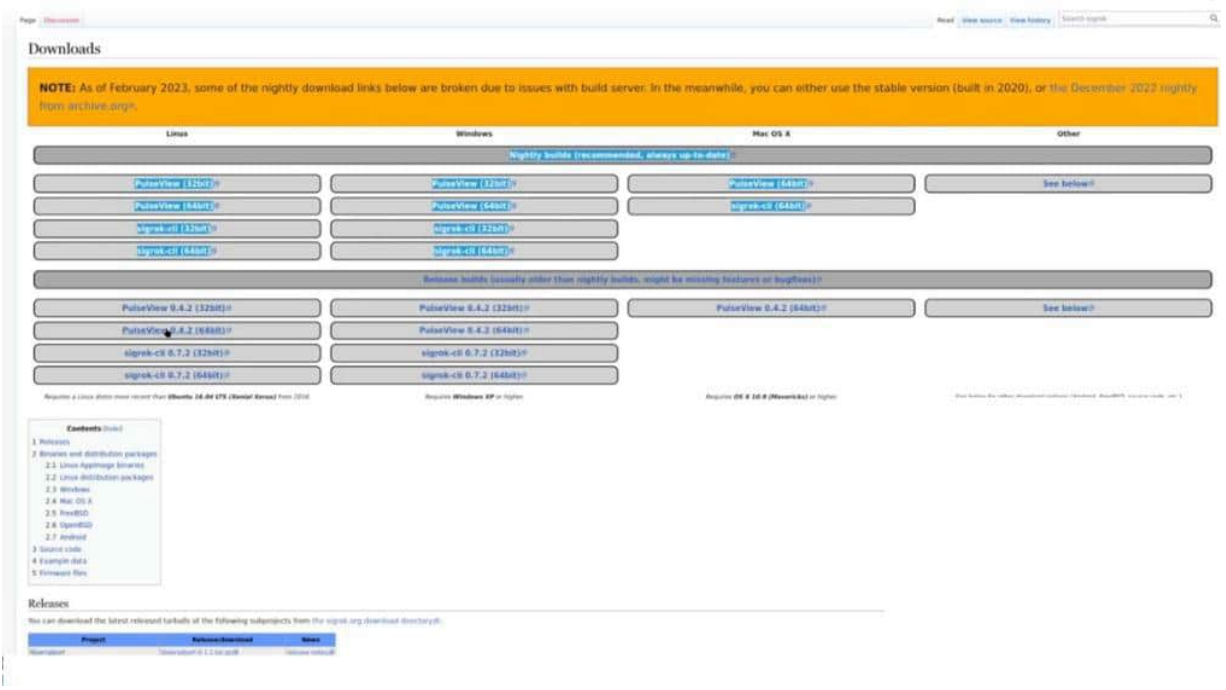
Project	Release/download	News

And I think a lot of these loans have the same chip inside, so you can follow me along. But also other and expensive ones, like the sallyas, etcetera, and This is why I really like it here in the past year because it has a broad understanding and support of hardware. Yeah. And also the community is really good. As I told you before, there are a lot of external decoders which you can find in GitHub, etcetera, and implement 2 year projects and have fun with analyzing different kinds of protocols.

DOWNLOAD

For the sake of completeness, let's go through where you can download all of the things. So on sickrock.org on the left side, there are downloads. And here, you can find a different kind of platform. For example, for Linux, you get an app image for Windows and normal Instala on for Mac I think also an installer and container. So here, you see some notes.

So the first links are somewhere broken or what else, doesn't matter. Just use the above, the below 1 of the pulse view 0.4.2. It works also the same as the newer version, and they're good enough to go. If you install here are some, as you can see, a few more inputs about all of the binaries. It's that, but just install the container here and you're ready to go.



Downloads

NOTE: As of February 2023, some of the nightly download links below are broken due to issues with build server. In the meanwhile, you can either use the stable version (built in 2020), or the December 2022 nightly from archive.org.

Linux	Windows	Mac OS X	Other
Nightly builds (recommended, always up-to-date)			
PulseView (32bit)	PulseView (32bit)	PulseView (64bit)	See below
PulseView (64bit)	PulseView (64bit)	sigrok-cli (64bit)	
sigrok-cli (32bit)	sigrok-cli (32bit)		
sigrok-cli (64bit)	sigrok-cli (64bit)		
Release builds (usually older than nightly builds, might be missing features or bugfixes)			
PulseView 0.4.2 (32bit)	PulseView 0.4.2 (32bit)	PulseView 0.4.2 (64bit)	See below
PulseView 0.4.2 (64bit)	PulseView 0.4.2 (64bit)		
sigrok-cli 0.7.2 (32bit)	sigrok-cli 0.7.2 (32bit)		
sigrok-cli 0.7.2 (64bit)	sigrok-cli 0.7.2 (64bit)		
Requires a Linux distro more recent than **Ubuntu 16.04 LTS (Xenial Xerus)** from 2016	Requires **Windows XP** or higher	Requires **OS X 10.9 (Mavericks)** or higher	[illegible]

Contents [hide]

1 Releases
2 Binaries and distribution packages
2.1 Linux AppImage binaries
2.2 Linux distribution packages
2.3 Windows
2.4 Mac OS X
2.5 FreeBSD
2.6 OpenBSD
2.7 Android
3 Source code
4 Example data
5 Firmware files

Releases

You can download the latest released tarballs of the following subprojects from the sigrok.org download directory.

Project	Release/download	News
[illegible]	[illegible]	[illegible]

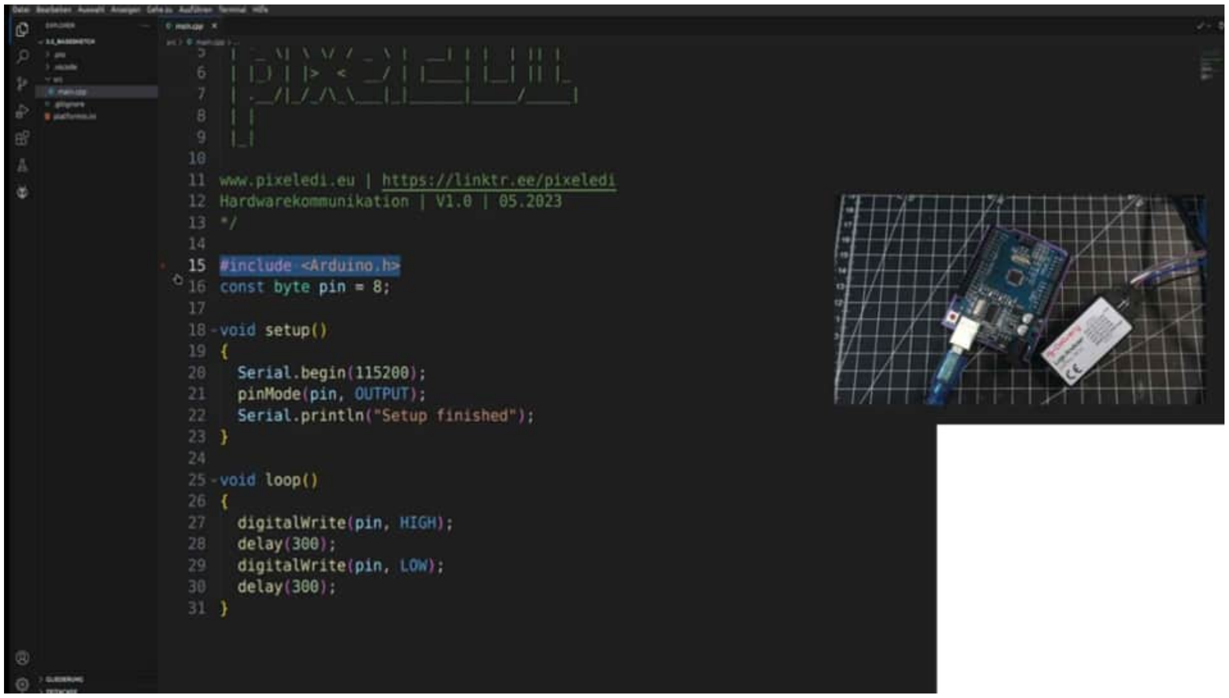
For Windows, I just installed it. Just clicked through the installer. There's nothing special. And you can see here this is the Windows version and this is the Linux version with the app image. I have exactly the same features.

So nevertheless on which platform you choose and what working, you can do it, the exact same things as I did it. And also here, just switch to a talk stream, talk theme, but all of the rest is the same.

BASE SKETCH FOR ALL EXAMPLES

In this project I would like to introduce you to our base sketch. The base sketch will be that code that we are using for every other project and this will be our initial starting point where we code together and build up step by step our code. Therefore we have here just a basic one and we are using the platform EO with Visual Studio Code. So if you are with the Arduino IDE then take a closer look at chapter one where I explain how you can transform my code to the Arduino IDE so that you can follow me along with the given code.

So I start with including the Arduino.h. This is the library that is necessary that you can use, strings etc. for the C++ environment and especially for the Arduinos.



```
| '_ \| \ \/ / _ \ | |__| | | | || |
| |_) | |>  < __/ | |___| |_| || |_
| .__/|_/_/\_\___|_|_____|_____/____|
| |
|_|

www.pixeledi.eu | https://linktr.ee/pixeledi
Hardwarekommunikation | V1.0 | 05.2023
*/

#include <Arduino.h>
const byte pin = 8;

void setup()
{
  Serial.begin(115200);
  pinMode(pin, OUTPUT);
  Serial.println("Setup finished");
}

void loop()
{
  digitalWrite(pin, HIGH);
  delay(300);
  digitalWrite(pin, LOW);
  delay(300);
}
```

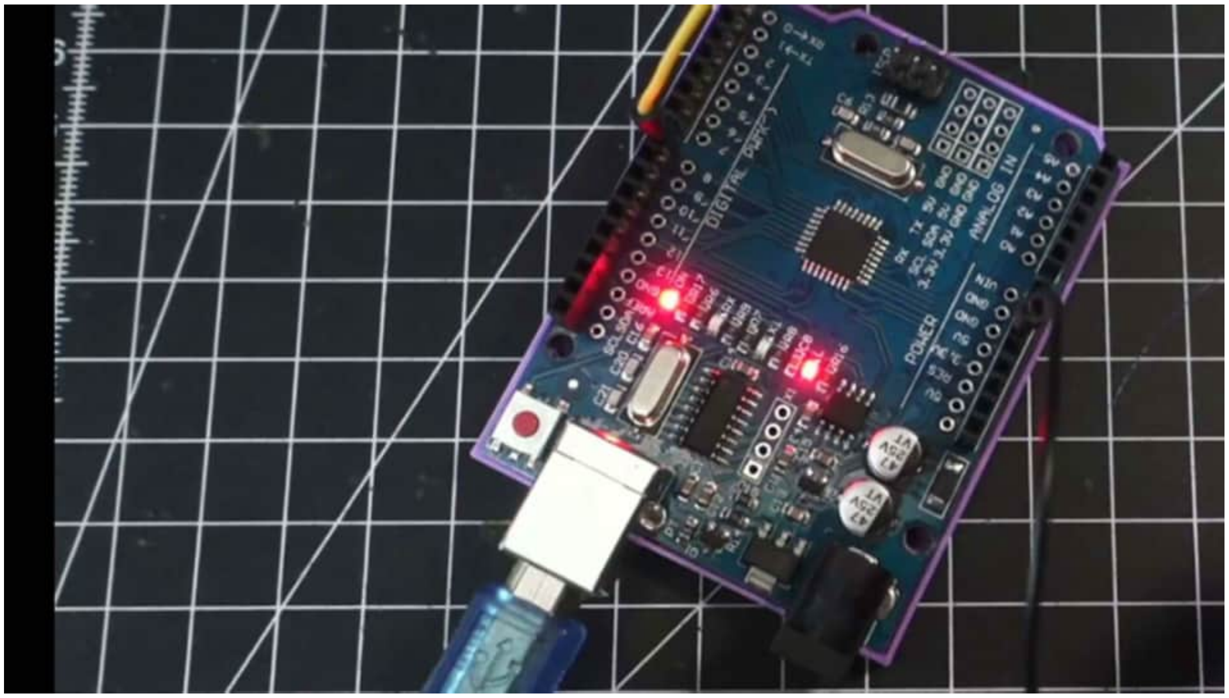
And I am defining here a pin, for example pin 8, because our main purpose in the next project is to make our first measurement with the logic analyzer and therefore I just want to make one pin on a signal output and then we will measure those pin 8 for example.

But those pin 8 could be any kind of pin, it just matters that we are using a pin mod that is an output, setup finish that we know, ok setup is through and this could be now our pin 8 that we have here and digital high and digital low for around about 300 ms and exactly this signal is our main purpose for the next project.

CONFIGURATION AND CONNECTING LOGIC ANALYZER

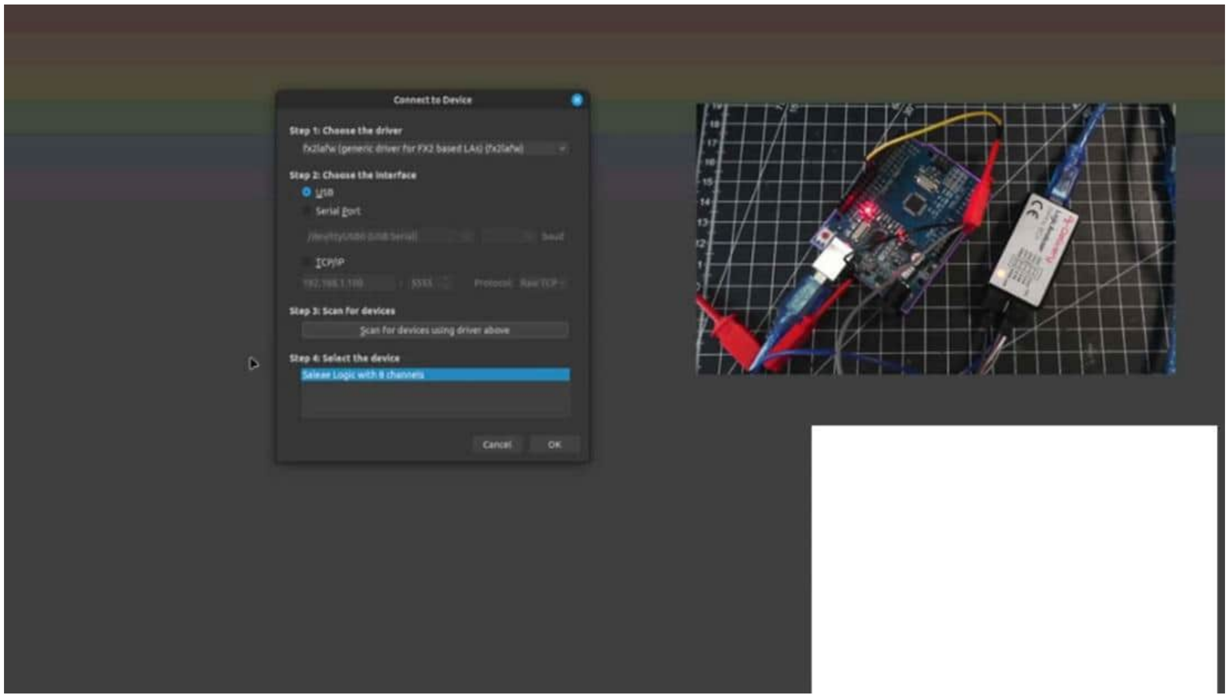
Now I've uploaded the codes to the Arduino UNO and what I would like to do now is, so let's switch here to embed a few, I have here an extra jumper cable because it's easier for us to grab the signals. So I have pin 8 here and I also need a ground here. Then we can go to our logic analyzer and with the logic analyzer I need the first one is the ground. So ground is always necessary in each measurement so that we have the same potential.

So just grab this one, be aware that those two don't get to each other and then channel what you would like to use.



I have a few more channels here so I grab one of the next and connect it to the Now I can connect my logic analyzer as well to So let's see if we have a little bit more cable here. Two wires are connected to the logic analyzer, one here and one here and what we do now is we're switching back to our PulseView and in PulseView now, so let's turn on, we are seeing here for example on the left side new window, new analyze window.

So new session if you have another language you can change it here on the general settings and what we want to do now is the first point is we click here on Saleae logic because we want to use here another device and for the main part with this cheap logic analyzer we can use here the FX2LAFW generic driver. Click on it then scan for the device and here we go. Here stands Salea logic.


Connect to Device

Step 1: Choose the driver
fx2lafw (generic driver for FX2 based LAs) (fx2lafw)

Step 2: Choose the interface
[✓ USB]
Serial Port
/dev/ttyUSB0 (USB Serial) Baud
TCP/IP
192.168.1.100 : 5555 Protocol: Raw TCP

Step 3: Scan for devices
Scan for devices using driver above

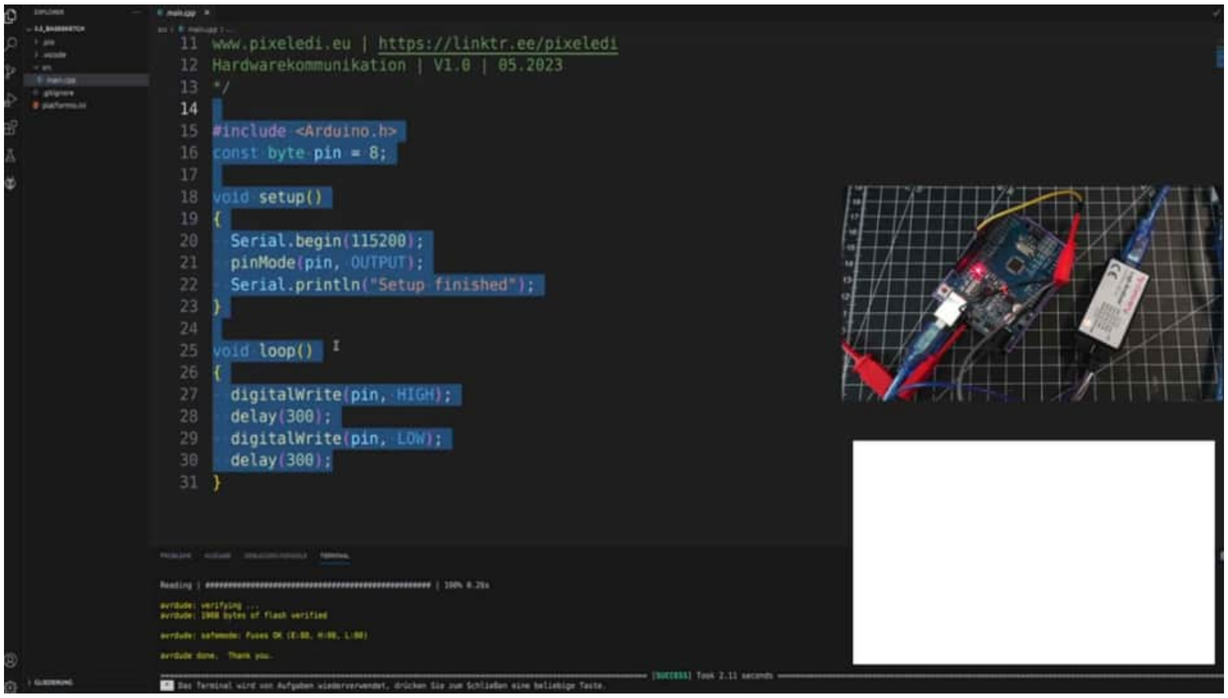
Step 4: Select the device
Saleae Logic with 8 channels

Cancel OK

I don't know why but I'm using it and now we are ready to go. That means our sketch is performing right now. We have connected our logic analyzer.

We can see it with the red dot and what we can do now is we have to change both settings above. That means I would like to sample now one gig sample and I would like to have here for example 12 megahertz. That should be fine. Then let's click on run and let's see if we got here some output and as you can see I have obviously attached the signal to D1. Then I click on stop and here is my 300 millisecond signal.

With the mouse wheel you can zoom in and zoom out and with this red clamp we could say that I only want the channel where I have my signal and this is in my case D1. So now I have only my signal and what is also very convenient is the measuring tool. Show cursors and now I can grab the left side and the right side and can for example measure here the time where there is no signal so a low signal and here you can see 300 milliseconds. If I push it to the high signal it should also be 300 milliseconds.



```
www.pixeledi.eu | https://linktr.ee/pixeledi
Hardwarekommunikation | V1.0 | 05.2023
*/

#include <Arduino.h>
const byte pin = 8;

void setup()
{
  Serial.begin(115200);
  pinMode(pin, OUTPUT);
  Serial.println("Setup finished");
}

void loop()
{
  digitalWrite(pin, HIGH);
  delay(300);
  digitalWrite(pin, LOW);
  delay(300);
}
```

```
Reading | ################################################## | 100% 0.25s

avrdude: verifying ...
avrdude: 1908 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.

=========================== [SUCCESS] Took 2.11 seconds ===========================
Das Terminal wird von Aufgaben wiederverwendet, drücken Sie zum Schließen eine beliebige Taste.
```

We got here the 300 milliseconds which we have here defined in our loop. So pin high 300 and low 300 and if I would like to have here a shorter time frame I could change here so let's get rid of that one gig for example to one amp sample. I click on run and then you can see I only get here a high signal so I need a more sample rate 10 m to get here more of the signal out and this is the reason why I would like to change here always at a minimum for one gene that I have here and write a wider span for capturing the data.

The change of the megahertz is how fast the signal will be captured and if we have a signal which is very fast and we choose here in our measurement device a very low capturing signal then of course the whole capturing process will be not really reliable and therefore it's necessary that we often go above that what it is sent so that we can be sure that it is captured and this is normally round about 12, 16 or 24. I often give it just a try and click on run so this is also always my first purpose, try and error and see if something comes out.

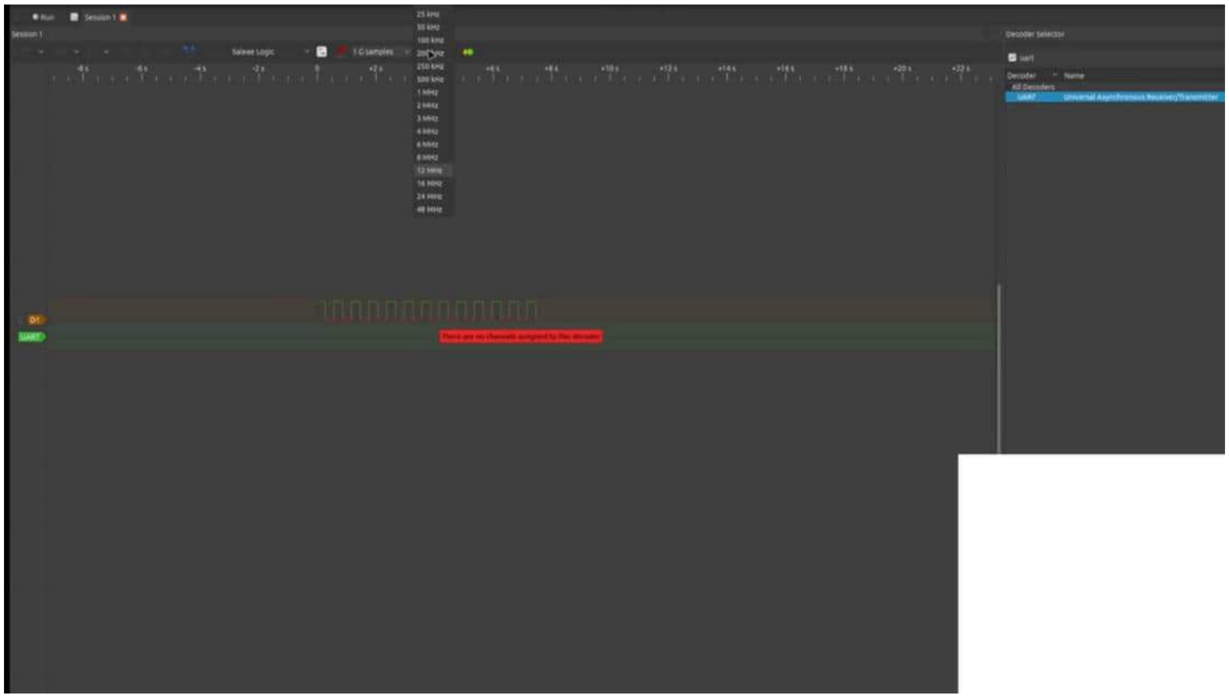
So this was our first test now with the logic analyzer and the first settings and that's it nearly all of these settings are used later on also in all of these examples. We can use also some external decoders or protocol decoders click on add protocol decoder and here we have a lot of different predefined decoders we are using later on for example the UART and just by clicking on it we have here in second channel and can make all of the settings here but we are using those settings a little later on in the chapter UART 1Y etc.

This was just a first glimpse of the logic analyzer, how it works and if the connection is right with the PC and the Arduino UNO.

SAMPLES AND MHZ SETTING

Let us discuss in this project once again the concept behind the sample rate and also the time period what we can set up in the logic analyzer or in the pulse field. So our logic analyzer is now connected and I would like to change here from 1G to 2G. What is the main change here? And if I keep the mouse over the setting you can see here that I have a total sampling time from 167 seconds. If I change it to 1G then of course this will be reduced by half and I only have 83 seconds. And this is the first concept. How long would I like to capture my data?

And the next point is here the sampling rate. And this is a really important concept because the sampling rate of a logic analyzer is the number of data points that we would like to record.



So that means if the sampling rate is too low, for example in kHz, then fast signal changes can't be captured. And we will do this example later on. That means if we are sending data on the serial monitor or sending some ASCII's and we change here to some kHz setting, then the logic analyzer is not able to capture all of the data in a reliable way.

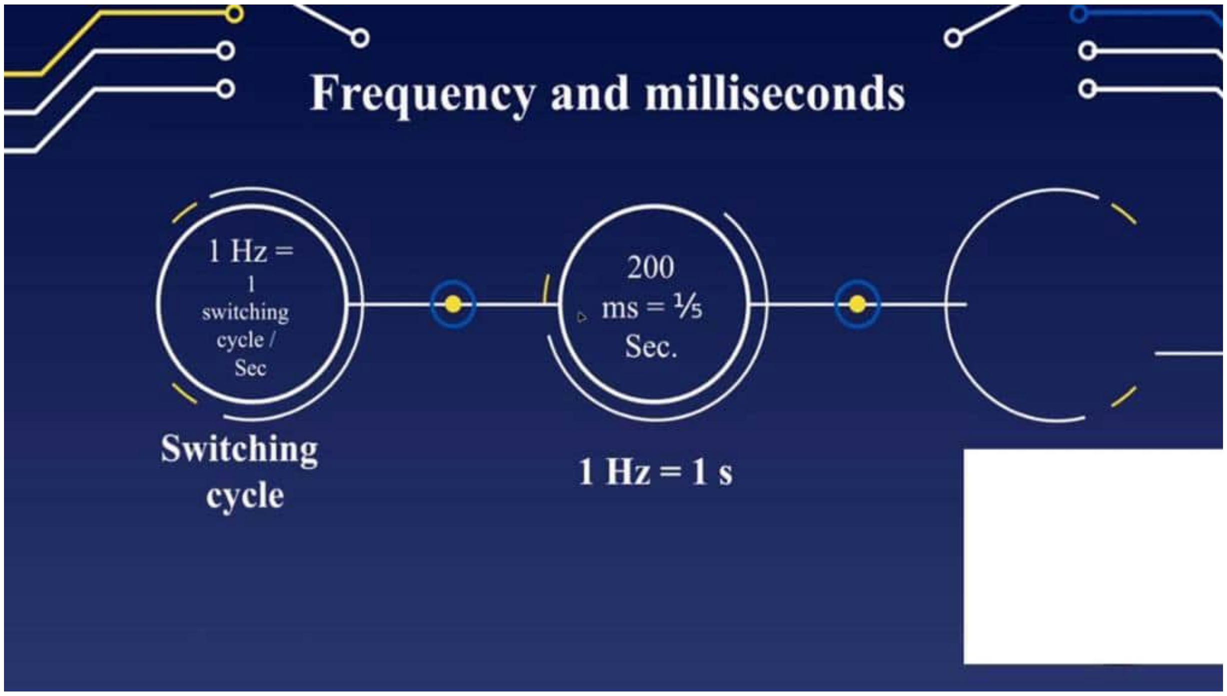
And therefore it's always a good point to use something in the spectrum of what the Arduino Uno is, a little bit above, so that we can be sure that the logic analyzer can capture all of the data. So here is our example, so let's try it out. If I use the first 20 kHz here, it doesn't matter because here we have 300 milliseconds. This is of course a signal that I can capture here.

But with faster protocols, E-squares, C, SPI, etc., it's really necessary that we are changing here according to the specs to the Arduino or ESP, so that we can then have a reliable signal. So often when you get here in the signal processing and you find some strange values, keep in mind if you have also changed the sample rate.

FREQUENCY AND MILLISECONDS

In this project, I would like to discuss with you the concept between frequency milliseconds and what is the relationship to it. because often, we can find in data sheets, the term and the data frequency, and not the milisack. And also when we want to calculate or find out the right bar rate, it's necessary that we can recalculate from milliseconds to frequency etcetera. And therefore, we can use here 1 Hertz is 1 switching cycle per second. And 1 switching cycle is defined you can find, for example, this is one data process, etcetera.

So, in our example, that means, Let me see here. Yes. 200 milliseconds, but we use 300 milliseconds. So just assume we had 200 milliseconds. is 1 5th of a second.



And with the basis of 1 hertz is 1 second, we can riff or transform it to 5 hertz is 200 milliseconds. And this is what we keep in mind. So when we are using some analyzing techniques, then, of course, you can now use the Hertz data, or you can also use here the milliseconds data. And when we are switching back to our previous example, we had here 300 milliseconds. And you can also find here the data 300 milliseconds and also here the value of the Hertz.

So let's reduce. So let's reduce the cursor here and go to the left. It's around 200 milliseconds. That's good enough, and you can find here the 5 hertz. And, therefore, it's easier to recalculate, to transform it so that we know how many seconds do we have here in our capturing data?

BAUD RATE

Let us talk about the bot rate. And I'm sure all of you have already used the bot rate or the settings, but are you aware what the bot rate is? thread is. So one part is the speed when one symbol is transmitted per second. An example.

So let's assume you are right now in a lecture. Good to know. We are already in a lecture. And therefore, when I'm speaking for example, 4 words per second, that means the bar rate is 4. If I'm talking a little bit faster, for example, 20 words per second, Then, of course, I will transmit more words, more information, more data, and this could lead to you not really being able to comprehend all of the things that I am talking about.

Baud rade

1 baud is the speed when
1 symbol is transmitted per

4 words

per second

20 words

But with four words per seconds or let it be 2 words, then it's easier for you to understand, but I need more time to transmit all of my content. And this is the same with the bot rate. And normally with the ESP cert the 2 or with the Adi and Uno, we have some predefined alterations. So, for example, for Uno, we often use 9600. That means 9600 symbols could be transmitted per second.

And you might have mixed up the pulse, right, with the serial monitor. In the end, you saw just some squiggly lines or some mixed symbols. And this is because the serial monitor and the Arden Uno have a different bot rate, and then they can't communicate because you have to be here at the same bot rate. that the sender and the transmitter can really communicate in the right way. And the portrait is also necessary for us to understand Okay.

The ESPs. So the 2, for example, have, normally, we're using the portrait 1152, 0 0. And there are a lot more different portraits. And we are using, different kinds of, the portraits that later on, for example, you are or in the E squared C chapter. And we are also trying to, and this will be also covered in the UI section.

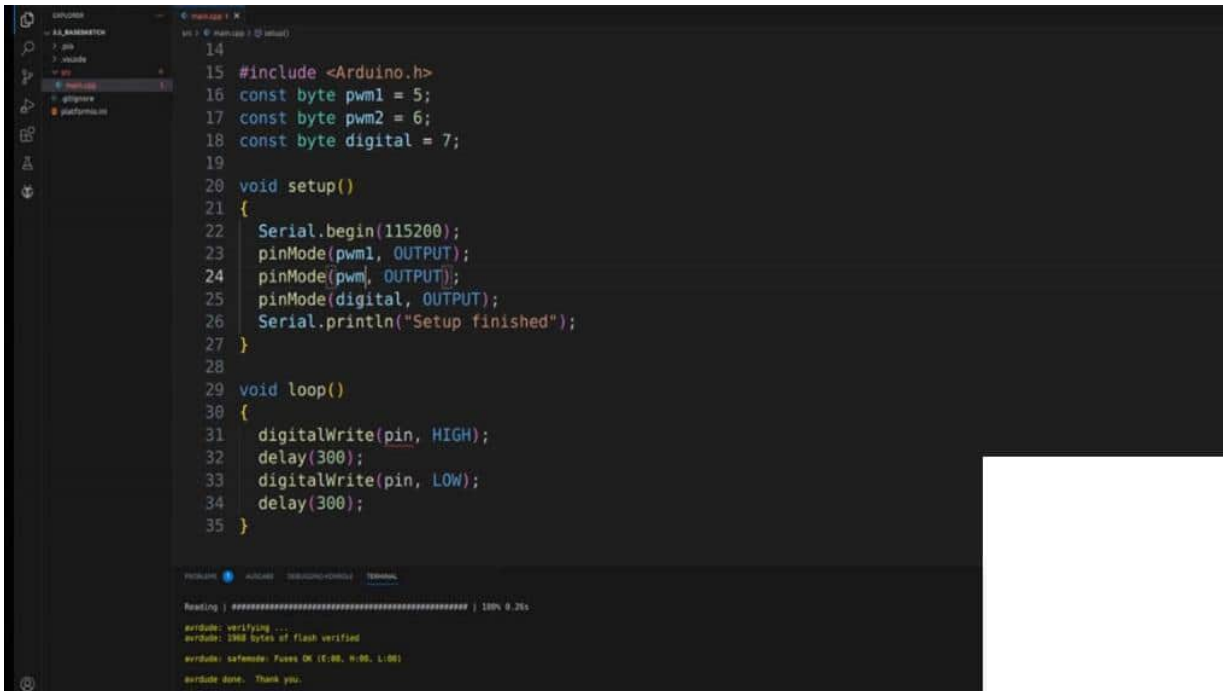
trying to measure is, and then we try to recalculate what is this, what kind of boundary is this? Why do we need that in some, logic or error inspecting or when we have an external PCB, we are capturing data. And then we want to use this data, for example, with an outer uniform. And therefore, we have to know what part rate is this MCU communicating or this sensor. And of course, we can take a closer look also in the datasheet.

But often, we only can capture data and therefore, it's necessary that we also can recalculate from the time which power rate it is.

PRODUCE AND MEASURE PWM SIGNAL

In this project, we want to analyze PBM signals. PBM stands for pulse with model and it's used in the arduino environment for simulating analog output with digital signals. And therefore, it's a perfect use case for our logic analyzer to test the logic analyzer and to inspect such PBM Signers. And we are using our basic sketch here, and I will edit it here. the base sketch.

And therefore, we say, for example, the PBM 1 is on the pin. Let's see. We are using pin 5 because it has to be a PBM pin, and you will see it with this squiggly line. on the board, then we have PVM 2, and we would say here in digital output. It's on the 6th pin, and we have it here on the 7th pin.



```

#include <Arduino.h>
const byte pwm1 = 5;
const byte pwm2 = 6;
const byte digital = 7;

void setup()
{
  Serial.begin(115200);
  pinMode(pwm1, OUTPUT);
  pinMode(pwm, OUTPUT);
  pinMode(digital, OUTPUT);
  Serial.println("Setup finished");
}

void loop()
{
  digitalWrite(pin, HIGH);
  delay(300);
  digitalWrite(pin, LOW);
  delay(300);
}
```

```
Reading | ################################################## | 100% 0.25s

avrdude: verifying ...
avrdude: 1968 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.
```

Then we say in the output, of course, in the setup, we have to also output all of these 3 signals. pvm1, pvm2, and the teacher. And now in the loop, we are making a delay here. And normally, we are skipping delays because they are really, really bad because of the breakup or whole loop. But in this case, we can use it.

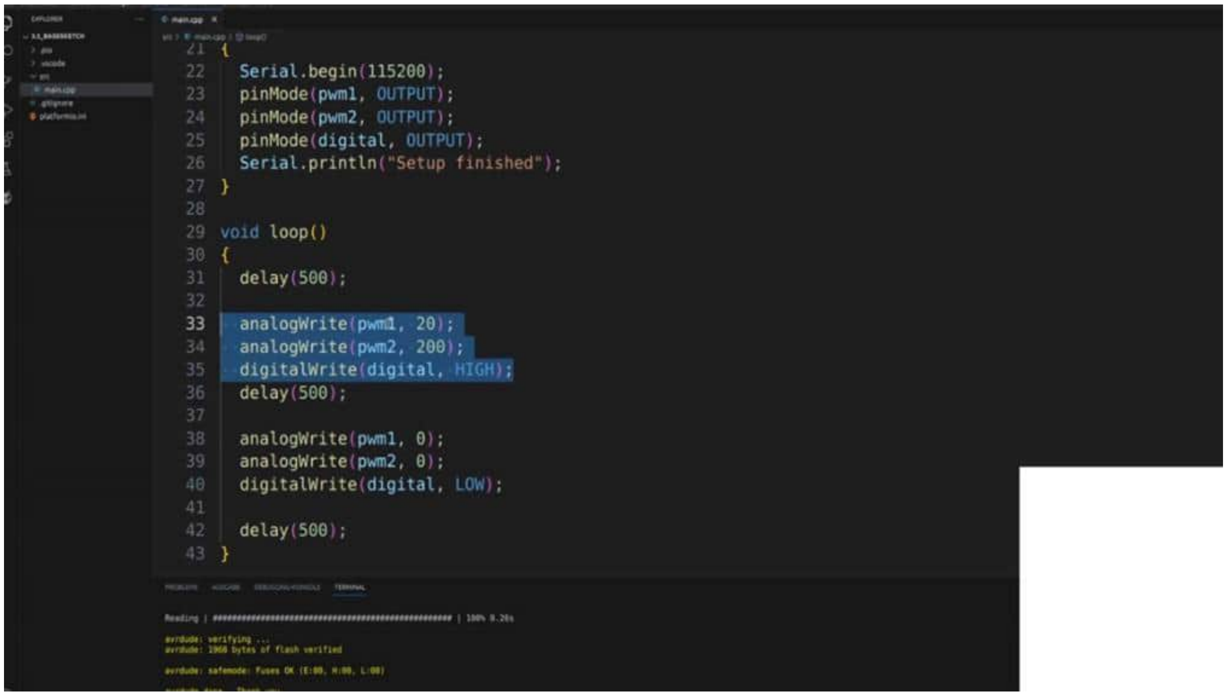
So let's see. we start and we end with some 500 milliseconds delay. And then we could say, for example, analog. Right? And if you're doing this with an ESP32, you have to include an external library because in the standard core, There is not really an analog write function implemented.

So analog write, PBM 1, and I would say 20. Then I will explain in a minute what we are doing here. So now it doesn't work. Analog write PWM tool. Here, I will have a duty cycle from roundabout 200.

And then I made a digital write that we can compare these 2 PBM signals, and I would say a digital higher. Then I will copy all of that. We're making a delay of 500 pasting in this block, and then we say 0, 0, and low. So what we are doing here in this loop, we are producing here 3 signals on 3 different channels. 1 the PBM 1 would have a short duty cycle by 20.

Then we have, bigger duty cycle by 200, and we have a digital signal that is the whole duty cycle, of high. Then we have a delay from 500 milliseconds. Then we put all three channels low, and then all again, it started. I think, yeah, we'll let you know. So it's one second I could do here.

So then, I keep the last one. So there's a little bit more delay between then uploading the code. And then we switch here to this view. And I've attached 3 jumper cables here. on pin 7, 6, and 5.

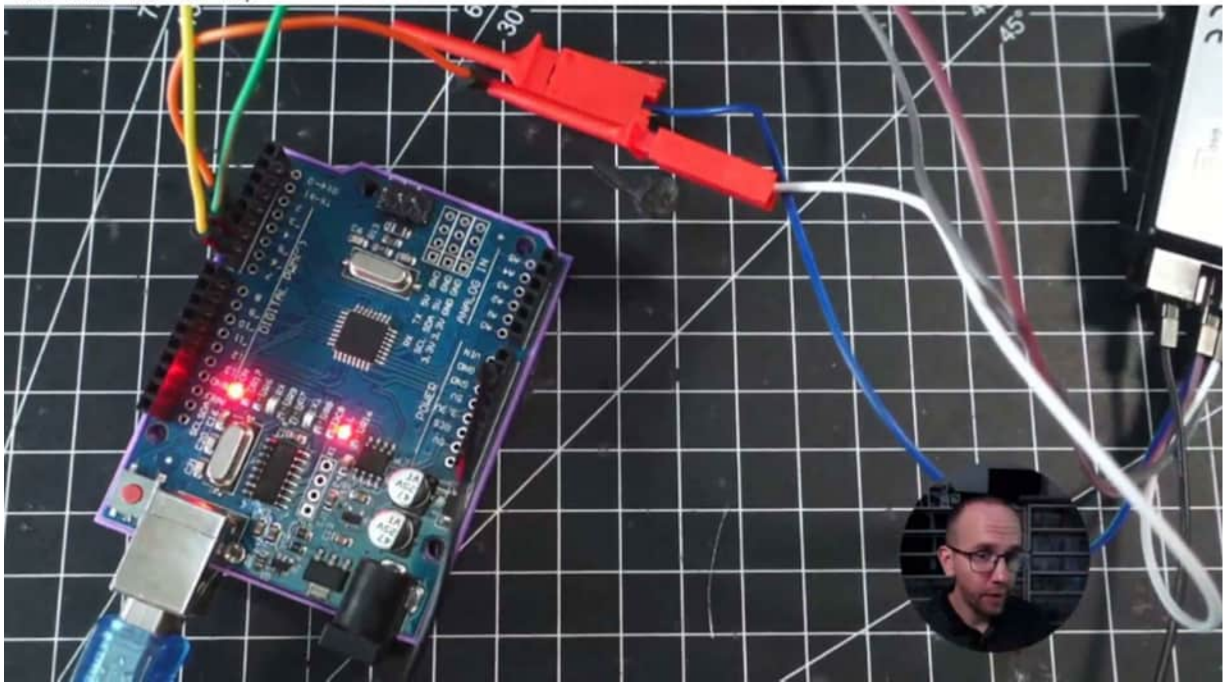


```
{
  Serial.begin(115200);
  pinMode(pwm1, OUTPUT);
  pinMode(pwm2, OUTPUT);
  pinMode(digital, OUTPUT);
  Serial.println("Setup finished");
}

void loop()
{
  delay(500);

  analogWrite(pwm1, 20);
  analogWrite(pwm2, 200);
  digitalWrite(digital, HIGH);
  delay(500);

  analogWrite(pwm1, 0);
  analogWrite(pwm2, 0);
  digitalWrite(digital, LOW);

  delay(500);
}
```

```
Reading | ################################################## | 100% 0.26s

avrdude: verifying ...
avrdude: 1968 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)
```

And now I'm using my logic analyzer. The ground, of course, is already connected as we did before. And then I'll see if I find channel 1 here, this should be channel 1. Channel 1, get the first Then channel 2 should be the gray 1. This is the 2nd PVM with 200 duty cycles.

And channel 3, we are connecting channel 3. Then let's see if I've connected everything right because we can change now to pulse view, making a new analyzer. been done selecting our device. Is it connected? I don't think so.



It's connected to the USB. Then we click on the connected device. We are setting up ethics, scan, yes. Then I'm setting the level to 1 g, and we're saying, for example, 16 Megahertz. That's it.

We don't need anything more. Our audio is running right now. You can see it here on the Also, the logic analyzer is working. And now we click on run, and we're capturing here some data. Then we click here to stop.

And I can get rid of 7 to 3 because we don't need it. This is starting by 0 and the logic analyzer starting by 1. So it's a little bit of, different name. But, let us focus now on one of the signals. First of all, I would like to capture and measure what is the time between the signals.

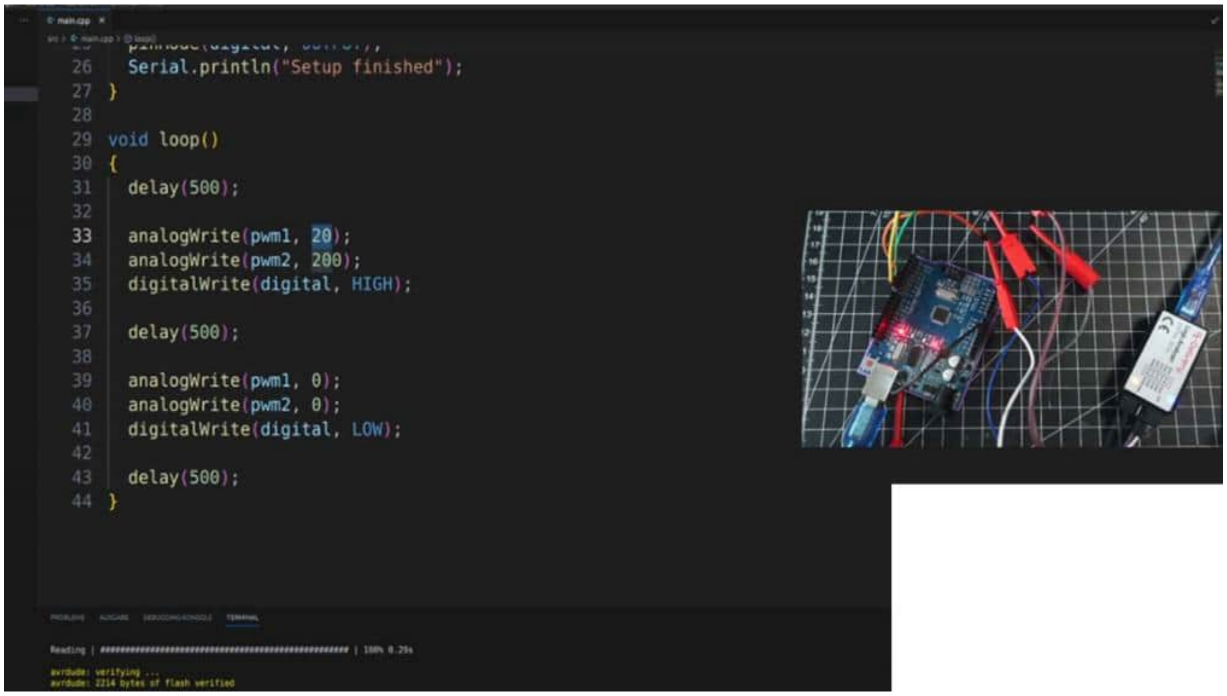
And before you measure, think on, according to the codes, what would you say? how much time is between one signal or one loop. And we have here one second, 1000 milliseconds. Why is it so? Because I have a delay here and a delay here.

Of course, you can only use one delay. But this is just for, and a thinking process. Then I would like to focus on this one. So let's focus on the new width. And we have here roundabout 500 milliseconds Also, this is perfect.

So this is channel 3, and this is our digital signal. You see one straight, high signal. And when we zoom a little bit more in, now we can see here our PVM signal. and let's measure the first signal. And this should be it.

84 microseconds, and we have 20 here. So 20 is the duty cycle. We have in the summary 255. So one byte is 1 digital high, and I only have 20 from this 255. And therefore, as we can see here, we have here this little aspect of one duty cycle.

And one duty cycle is round about this one because we have 200 here. And in the second channel or D1, we have 200 in the duty section. And this will be 1, duty section with 250 5 units. So what we did here is we skipped the first part and had only 200 from the range of 250 5. That means if we add here an LED with a resistor, the LED will not have his full voltage.



```
  Serial.println("Setup finished");
}

void loop()
{
  delay(500);

  analogWrite(pwm1, 20);
  analogWrite(pwm2, 200);
  digitalWrite(digital, HIGH);

  delay(500);

  analogWrite(pwm1, 0);
  analogWrite(pwm2, 0);
  digitalWrite(digital, LOW);

  delay(500);
}
```

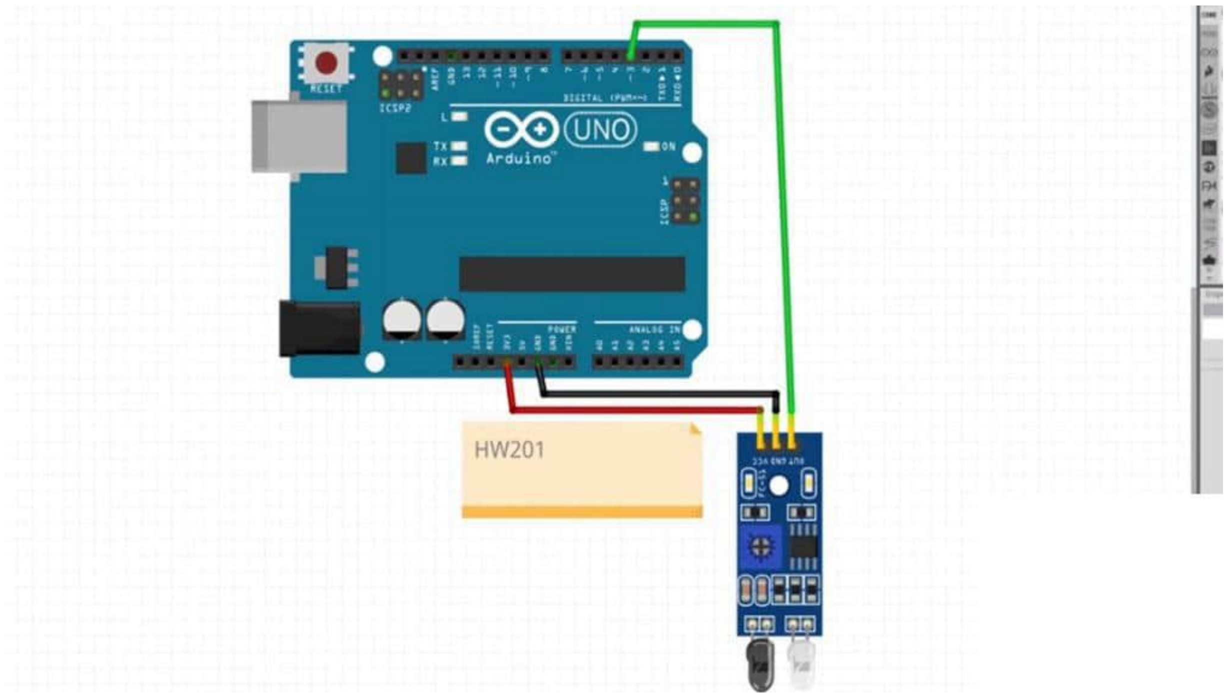
```
Reading | ################################################## | 100% 0.29s

avrdude: verifying ...
avrdude: 2214 bytes of flash verified
```

And, therefore, the light will be reduced. And even more reduced, it will be when we are, attach it to the D 0. And this is a typical way we can control motors or we can control LEDs or some other lights with PVM because the voltage or the whole supply will be reduced by just this tiny pulse. And as you can see in the D 2, we have a complete high level through all of the time, and this is the normal, high signal. And here, we have this 200 and this 20 PVM.

And in this project, I would like to show you how we can measure here signals and also that we can produce it from the other inu and what we are able to do with this logic analyzer.

TRIGGER ON LEVEL WITH IR SENSOR

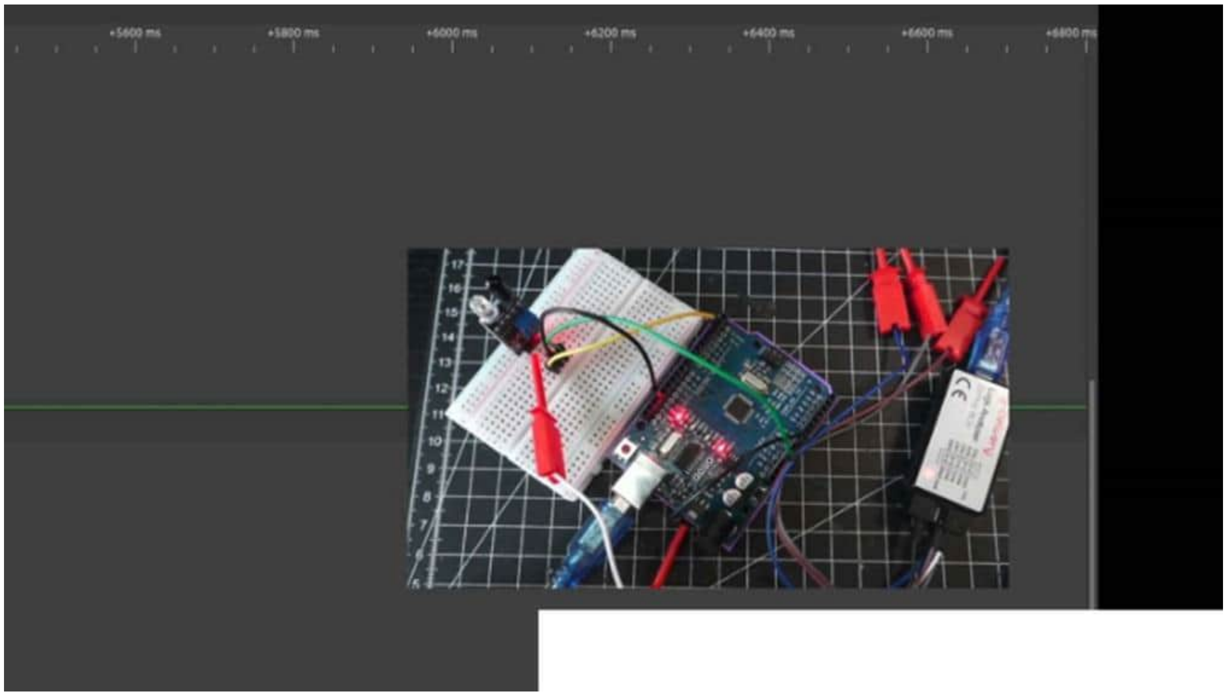
In this project, we talk about what we can do with external triggers. And we often use external triggers for Paul's view when we are inspecting or investigating our circuit and they're, you know, searching for some errors. And for example, if we want to test and turn the system and want to see if the voltage which he passed through is correct or not correct, we can also use a logic level, and therefore, some triggering measurements would be nice. And we do this in this scenario with, infrared sensor and hw201. And we are simply connecting here the 3 word ground and to unpin, but this pin is not really used because we don't attach any sketched to it, but we could do it.



And, here, you can see my setting. So this is the sensor, the infrared and infrared sensor. And I connect the ground VCC to pin 3. Ground for the logic analyzer is on an auto pin. And here with this clamp, I directly connect to the breadboard or to the module to the out point.

And now, we can open and pause the view again. Start a new analyzing window. Then we click on devices, and we select our device. Okay. I have attached it to channel 1.

Here it's channel D 0. Then I'm changing to one gig and, for example, 60 megahertz. So now, let's run. And we, in this use case, have a high signal here. If nothing is detected, If I put my hand in front of it, then we get a low signal. So you can see it works pretty well.

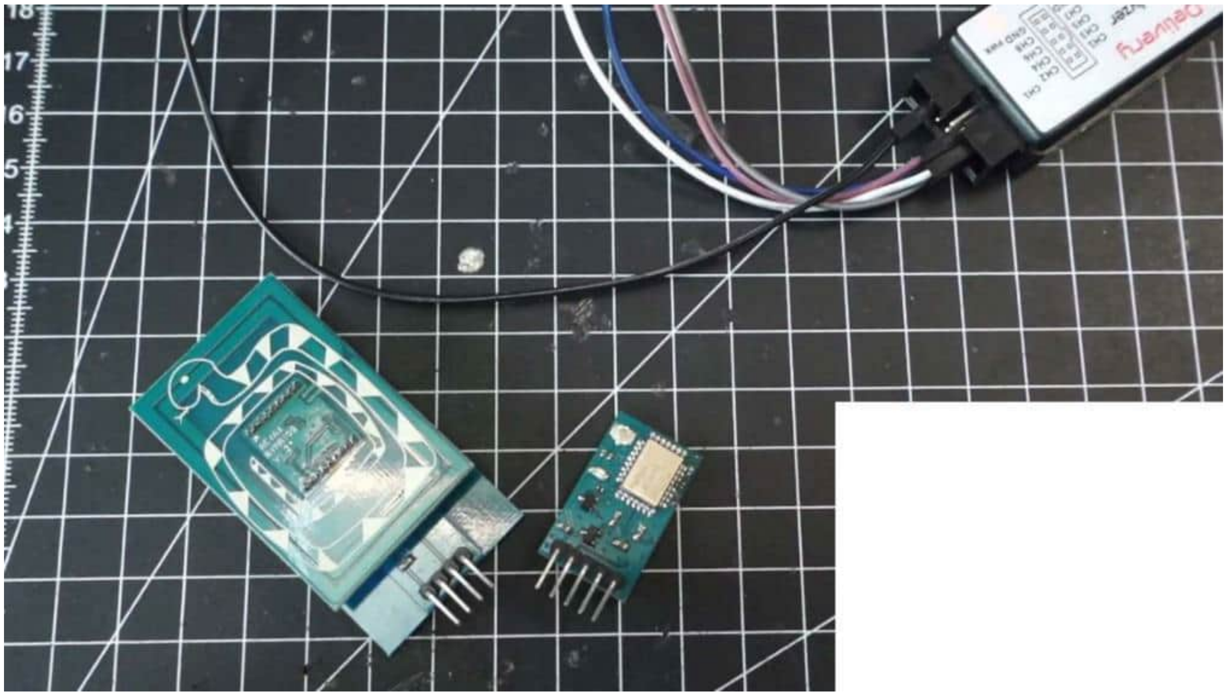


But what I would like to do now is, not that we're getting here straight the values out of it, I would like to start the measurement when something is triggered. For example, you could click here on the left side. And then I click here, maybe I would like to change here if the flag is changing. Or, as you can see here, trigger on rising edge, trigger on rising or falling edge. And I would like to trigger on a low level in this specific scenario.

And then I click on run, and you will see nothing, nothing is captured until I now put my hand in front of the sensor, and then the whole measurement will be started. And this is really convenient when it comes to investigating all of the PCPs, etcetera, or you're circled because otherwise, you will be spammed out of the capturing data.

OVERVIEW OF SERIAL COMMUNICATION WITH UART

We start in this chapter by defining and discussing what is u art. 1st of all, u art stands for universal asynchronous receiver transmitter. And this is a communication protocol commonly used for serial communication. and you are unable to see a serial communication between a transmitter and a receiver via 2 data lines. That means microcontroller sensor sensor microcontroller.

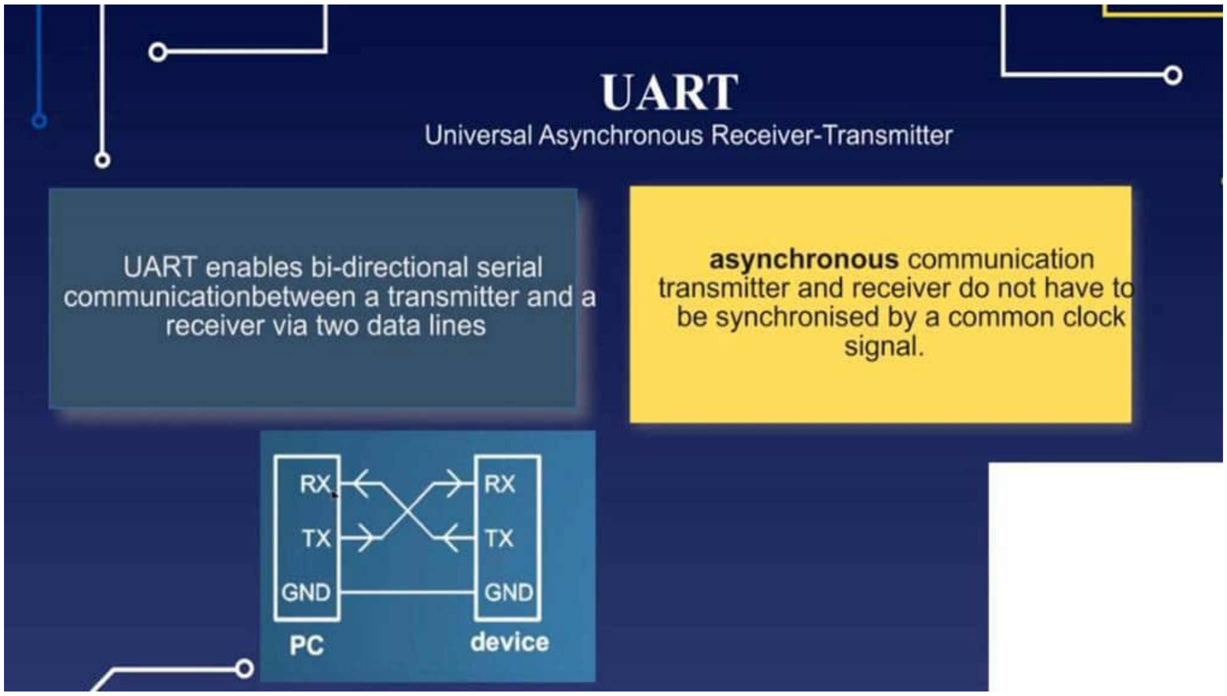


And UAT allows asynchronous transmission. That means that the data is sent without any clock signal. Later on, we will also discuss SPI and e squared. see where we have a clock line, which indicates when data is transmitted. This is not the case in UHARD.

And this is also an advantage and disadvantage which we also uncover later on. EODs widely used in different kinds of applications. So for example, I've had an NFC and RFID module or a GNSS module. So it's not really commonly used in the arena environment, but you will find several components available within your interface And this is when the Arlenino comes in place or an ESP because we can use those devices. And nevertheless, what kind of MCU are they using?

This could also be an STM32, etcetera, you will probably find a UI interface. also e squared c very, very commonly in the area of microcontrollers. If you're using a typical 80 tiny, for example, you also will find a new route. And also e squared c is commonly used in these cases. So Uart is also a kind of asynchronous communication in which the transmitter receiver does not have to be synchronized by incoming clock signal.

And the disadvantage is that we can't send data in a fast way. because if we set up a clock signal, we can be very sure and correct when we are transmitting data. Also a common misconception, or I would say pitfall is that when it comes to the connection, we have to crosswise implement it. That means, for example, if we have here our microcontroller, then the Eric line comes to the t t x line from the sensor and vice versa. Because the sensor transmits data and we will receive it.



UART

Universal Asynchronous Receiver-Transmitter

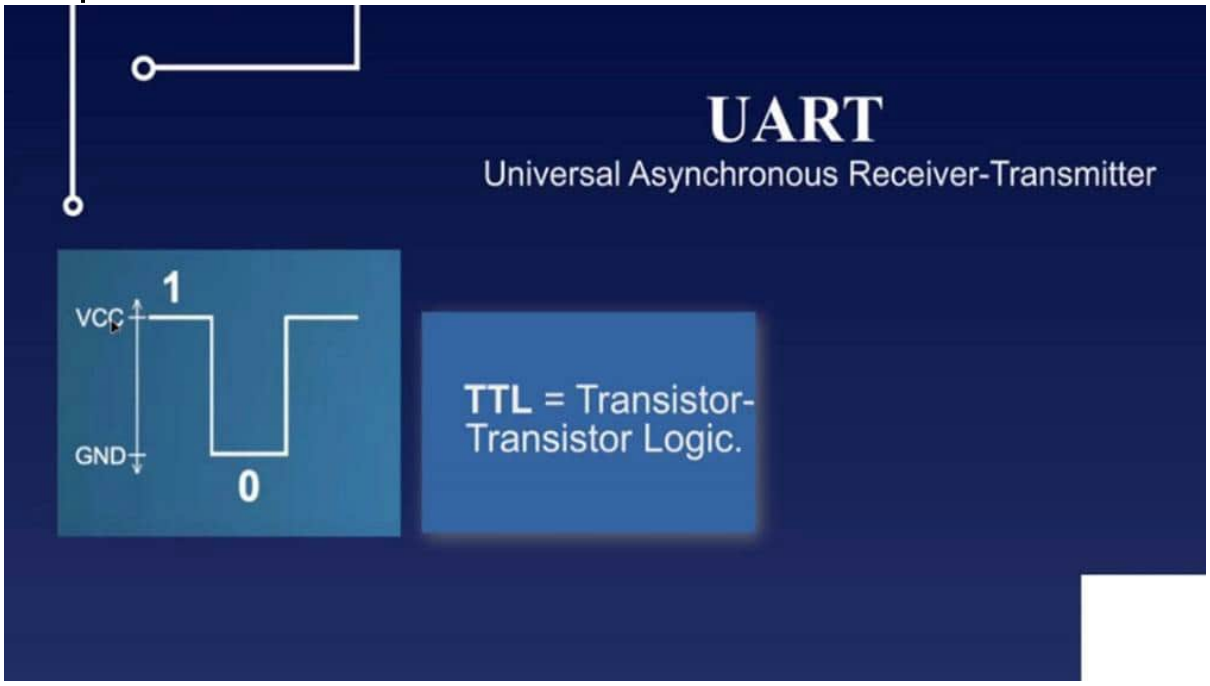
UART enables bi-directional serial communicationbetween a transmitter and a receiver via two data lines

asynchronous communication transmitter and receiver do not have to be synchronised by a common clock signal.

And also when we transmit data, the sensor will receive it. And this is the reason why we change here Eric's and T X. So what does it mean to have here a TTL and transistor transistor logic level? That means we are sending analog datas, for example, and transforming it to once and 0s. And this is what we can capture.

That means when we are sending here via our GPIO and with the Arlene Unum, we have a logic level from 5 worlds and the GPIO. And in the ESP environment, we have a 3 point sweet world. Also, AT Times, etcetera, have 3 point sweet world logic level. And with this 3.3 or 5 volt, we're indicating in 1. And if we have, yeah, on ground level, this indicates 0.

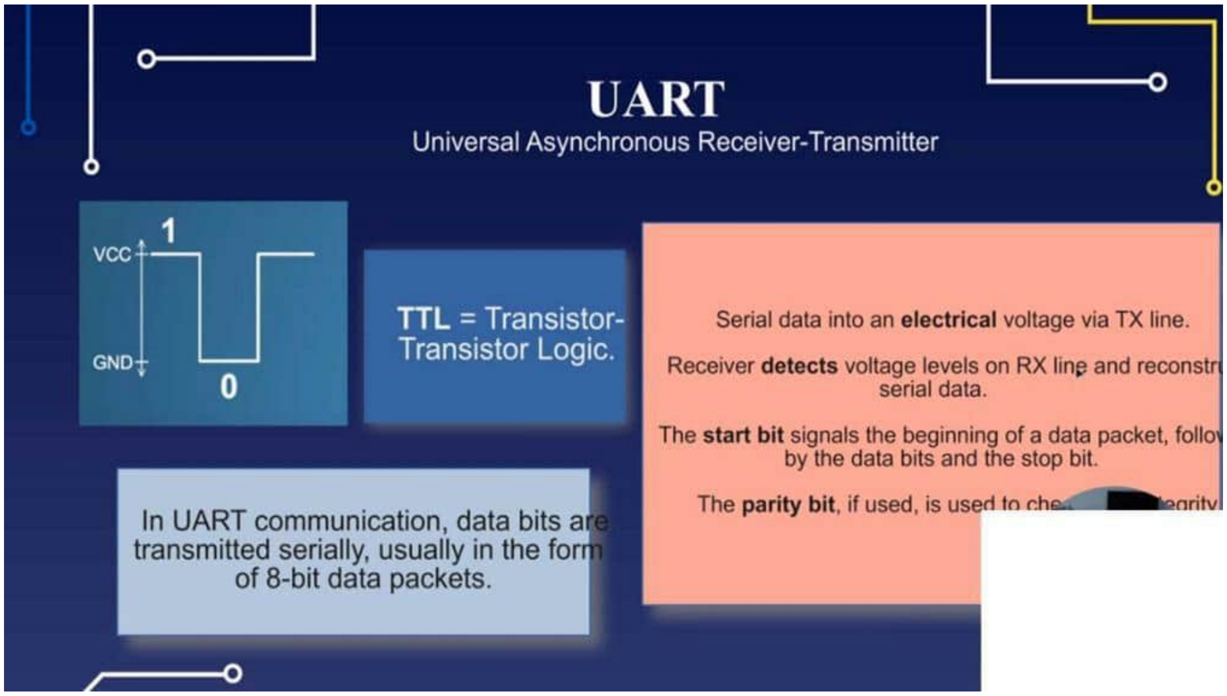
And this is translated later on into bigger numbers, and these numbers could be, for example, a sensor value from a DHT 20 or from a light sensor. And this is exactly what we want to do. We want to send data from our MCU to a component . Why is that so? Because we want to say that they send the commands to the components. Please now start with communi


UART
Universal Asynchronous Receiver-Transmitter
1
VCC
GND
0
TTL = Transistor-Transistor Logic.

cating with me, or please start measuring.

And if you're finished measuring, send me back the value. This is the reason why we use both lines, the Erickson TX line. In the example of the IKEA sensor, we only need to receive data with only one line inclusive of g g n d as a ground, so two lines are needed. In your communication, data bits are transmitted serially, usually in the form of 8 bit data packets. So when we only have 8 bits of data, we can't really transmit bigger numbers.

And, therefore, we have often the case that we have 1 regular register with 8 bits, a second register with 8 bits and we can combine it, for example, and then with a place for a bigger number. So in a summary, serial data is into electrical voltage via T where the T X line receiver detects the voltage levels on the Eric's line and reconstructs the serial data. And this is what we do later on in the auto enos catch that we read out all of the signals, and then calculate from binary to decimal. And this is why we also need the first chapter with this whole bit of operations because This is the case in which we later on also get our data out of it without any external libraries. And then The start bit signals the beginning of a data package and is followed by the data bits and then stops.



UART

Universal Asynchronous Receiver-Transmitter

TTL = Transistor-Transistor Logic.

Serial data into an **electrical** voltage via TX line.

Receiver **detects** voltage levels on RX line and reconstru serial data.

The **start bit** signals the beginning of a data packet, follo by the data bits and the stop bit.

The **parity bit**, if used, is used to che tegrity

In UART communication, data bits are transmitted serially, usually in the form of 8-bit data packets.

And this is very good shown later on also in the policy with the start and stop bit, etcetera. The parity bit is used to check the data in credit, and this will be a separate project to play a little bit around with this parity bit. Yeah. And this was a short summary about you art, and now we would like to start a little bit more with the practical part

THEORY OF DATA TRANSMISSION

One more theoretical project or inputs before we start with the practical example, I promise. But I would like to start with the TTL logic. So transistor transistor logic so that we really understand what is going on here. And I would like to show you here an example, what we do later on also in pulse view. So, for example, we have the following measurement.

So, this could be our logic level. Then we have such a thing a little bit longer on serial. Then we have such a small one. And that's it. So on this will is now our output, for example, on the pulse view.

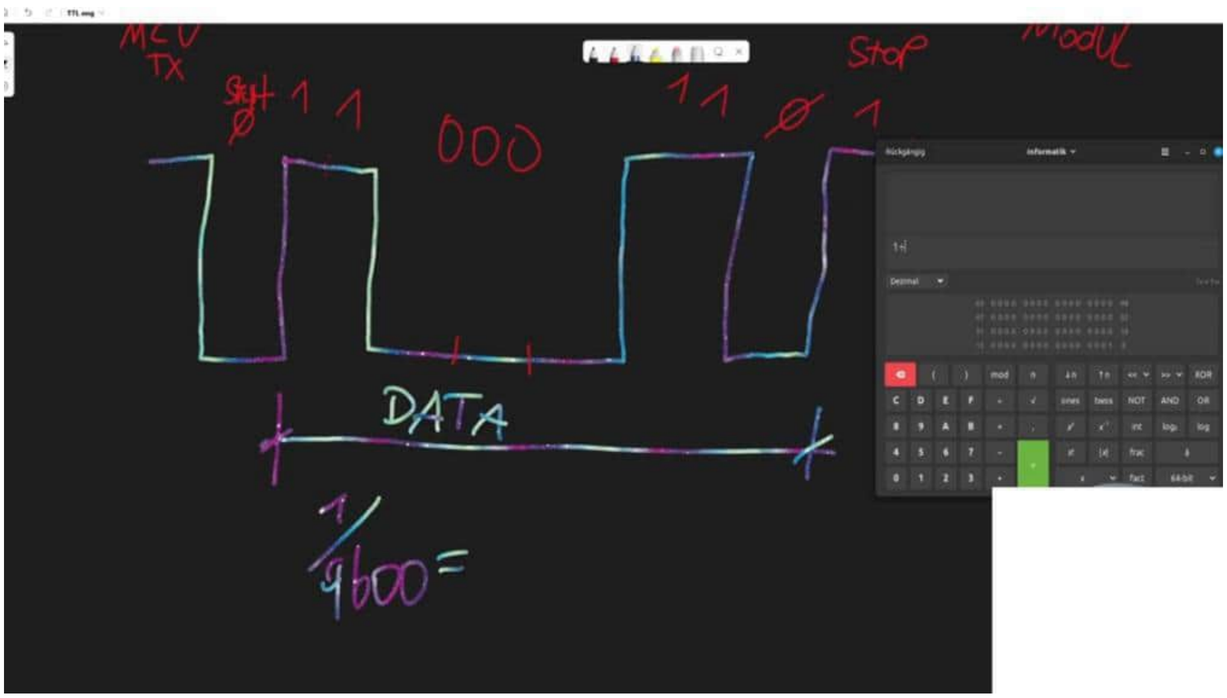
And what we have here is, for example, this is our MCU, and this is our line. We are sending here commands to a module component of what else. Module. So what we have here is a serum, and this will be our start. is indicated in UART when we, heavier and high level.

And then with and 0 will be indicated. Okay. The transmission could be started. Now, we have a high line, and this high line could be, for example, because this is 2, 2 times measured. So 1 and 1.

Then we have 3 0s here. k. This is not really the same measurement, but maybe we have three times here. And this two times And here we have a serum. And then we have one stop, That means we have 1, 2, 3, 4, 5, 6, 7, 8.

So we're transmitting here the data 1100 0110 because this is the stop bit. Maybe there's a parity bit, but this will be optional. And this is exactly what we are doing later on sending data. That means also that In this time frame, here, this is our data, our transmitted data, And this is the start and the stop indication. And now, we can also interpret how long a signal should be transmitted.

And therefore, we could say, let's go a little bit up. we could say, for example, in an hour, by an early noon, we have some power rate from 9600 Divided Nope. That was false. 1, I would say it's, in the other way. So one divided by 9600 should be roundabout.



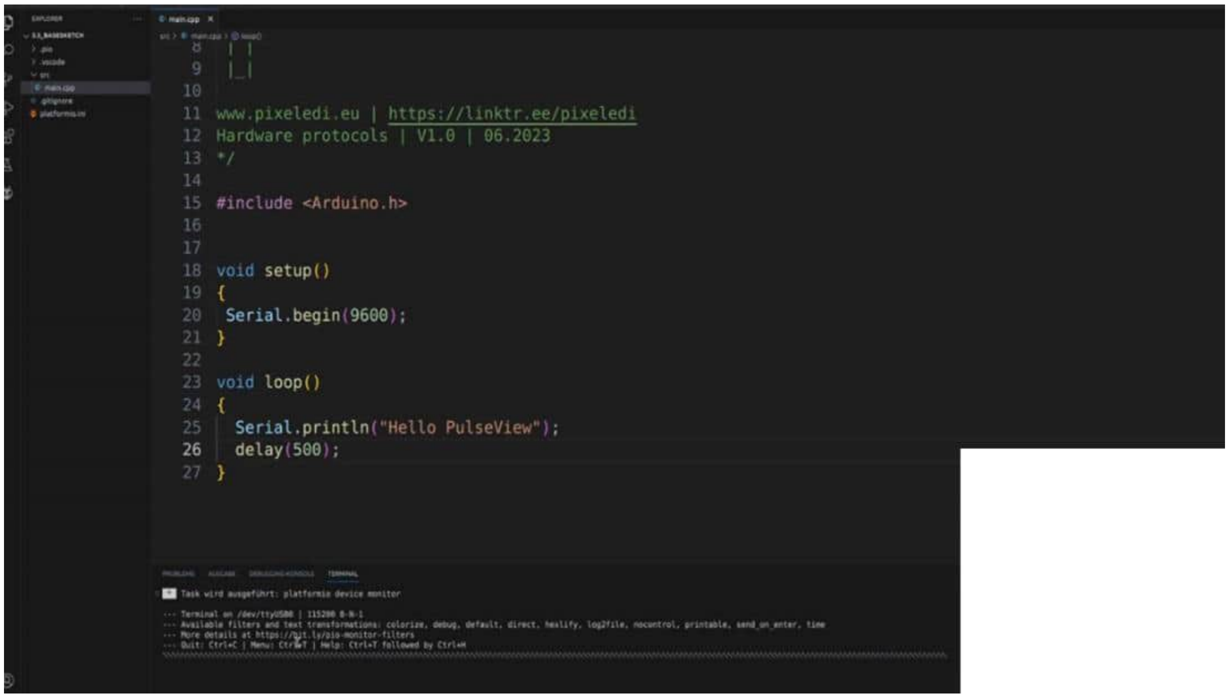
I think Let's see. 1 divided 9600 should be roundabout 104 microseconds Here we go. That means we have 104 microseconds, and this is such a time frame, what we have here. And this is also what we can measure later on. 104 seconds.

And this is, it's on a theoretical level that you understand what it means transistor, transistor, logic level, when we are talking about one's and ones and zeros, which will be transmitted, and this is our separation on a bit level so that we can transmit and receive data.

HARDWARE SERIAL ARDUINO UNO

In this project I would like to make with you the first Hewlett communication and we are not interpreting the results, we just want to be sure if the communication works or not. And therefore I have here an Arduino UNO and clone from it. I have connected here a ground and also on the RX and TX line which is here on the upper side. And I just connected the TX line because I want to send here some data and will capture it with the logic analyzer. So therefore we are connecting those two lines to our logic analyzer.

And then we can switch back to our IDE and I have here a blank sketch and what we are doing now is just in the setup we are starting in serial begin. For example with 9600 or 1152 or whatever baud rate you would like to do. And then we are printing out some text on the serial monitor and we could say hello pulse view and we are making a delay from 500 ms or half of a second.



```
 | |
 |_|

www.pixeledi.eu | https://linktr.ee/pixeledi
Hardware protocols | V1.0 | 06.2023
*/

#include <Arduino.h>


void setup()
{
 Serial.begin(9600);
}

void loop()
{
  Serial.println("Hello PulseView");
  delay(500);
}
```

```
Task wird ausgeführt: platformio device monitor

--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocontrol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
```

Then we are uploading the sketch... will get here some squiggly lines because I have in the platform.io I have managed here monitor speed 1152.00 so I have to change here also to 1152.00 and now we should see here hello pulse view exactly. And now we are starting pulse view, new session, device, then we are seeing only the first channel I would like to have, 1 GHz and as before for example 60 MHz. Now I click on run and we see here some data is transmitted.

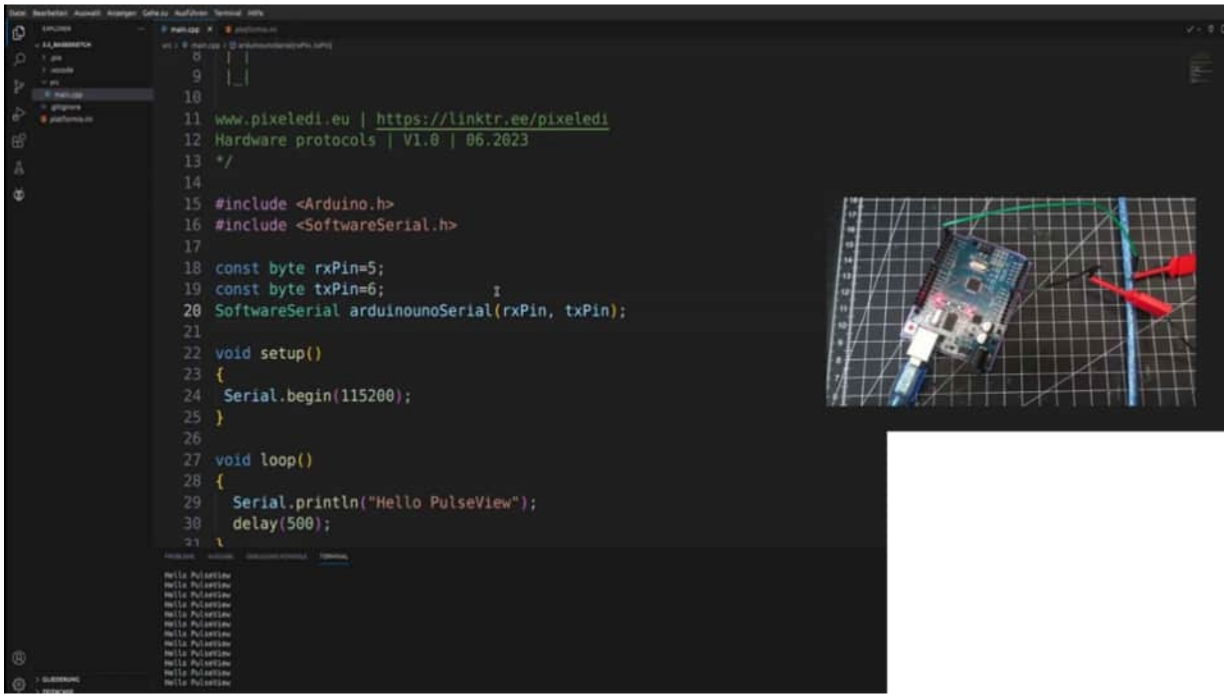
Click on stop, then we are zooming in with the mouse wheel and now we see here some data is transmitted. And as we did before we can see now that there are some 1's and 0's and now the first thing we can do is just calculate 1 divided by 1152.00. This will be 86 microseconds for one logic level, for one trigger logic level. And let me see if this is correct, 8.5 microseconds. And now we have the right baud rate and 8.5 or 8.6 yes. And we are not recalculating now the S-keys from this binary radio, this will be done later in a later project.

This project has only the purpose if we manage the communication via UART from an Arduino for example and capture the data with pulse view.

SOFTWARE SERIAL ARDUINO UNO

Now we have the case that for example the hardware serial connection and GPIOs are not available for some reasons or maybe there is already a sensor on these GPIOs and therefore we also have the possibilities to make a software serial. Therefore we have to include here the software serial library. This is included in the Arduino UNO core. That means we don't have to install something here, we just have to include it. With the Arduino UNO we can go here... some GPIOs. and... 6. Then we are starting the

software serial and now I can choose any name for example Arduino UNO serial and I overhand here the RX pin I think and this should be the TX pin. Yes, TX pin. This is why I love Visual Studio Code with Platform.io because here all of these parameters will be shown in the right way so the intelligence is really really good.



```
 | |
 |_|

www.pixeledi.eu | https://linktr.ee/pixeledi
Hardware protocols | V1.0 | 06.2023
*/

#include <Arduino.h>
#include <SoftwareSerial.h>

const byte rxPin=5;
const byte txPin=6;
SoftwareSerial arduinounoSerial(rxPin, txPin);

void setup()
{
 Serial.begin(115200);
}

void loop()
{
  Serial.println("Hello PulseView");
  delay(500);
}
```

```
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
Hello PulseView
```

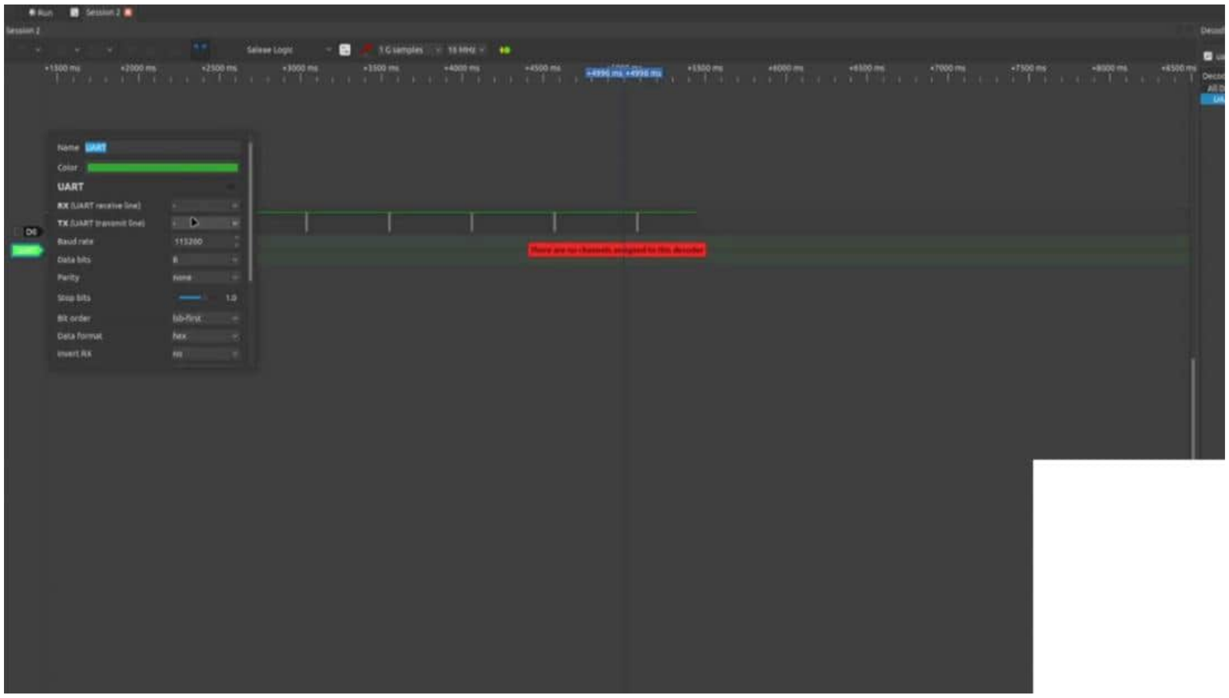
Then in the setup we could start now the serial begin but also as well we are starting the Arduino serial and we could change here the baud rate for example begin 9600 and in the loop we're printing out on our serial monitor PulseView. So for example hello PulseView software serial and also because this is what I would like to capture later on is Arduino serial. Then we're seeing here print line same thing hello PulseView software serial. Then upload the code. Before we can measure anything here we have to change the pin.

The sending pin TX pin is number 6. We don't receive anything here therefore we left this one empty. Here we see we're getting in 500 milliseconds the new hello PulseView software serial and zooming out click on run and also now on the on the new pin on the pin number what was it five we're receiving here the data from our new software serial. So a convenient way we can switch here from the hardware serial pins GPIOs to a software serial.

DISPLAY ASCII WITH DECODER

And therefore I would like to change the text here. So for example a smaller text that is easier to interpret. So let's take this one here, upload it. And I have the same pin as before with the software serial. You can use the hardware serial, doesn't matter. It's the same output. So here we have the new text. I click on run and I also get the new data. I click on stop because it's enough for now. And what I do now is I am adding here a decoder. This little sign here.

And now we are typing in UART, double clicking on it. And we have attached a second line here. And when we click on UART the option menu will be opened. And now we have to do a few settings. So first of all the RX and TX pins. So the TX pin is on line 6, on GPIO 6 but on line D0. Because we are not receiving anything therefore I leave this empty.

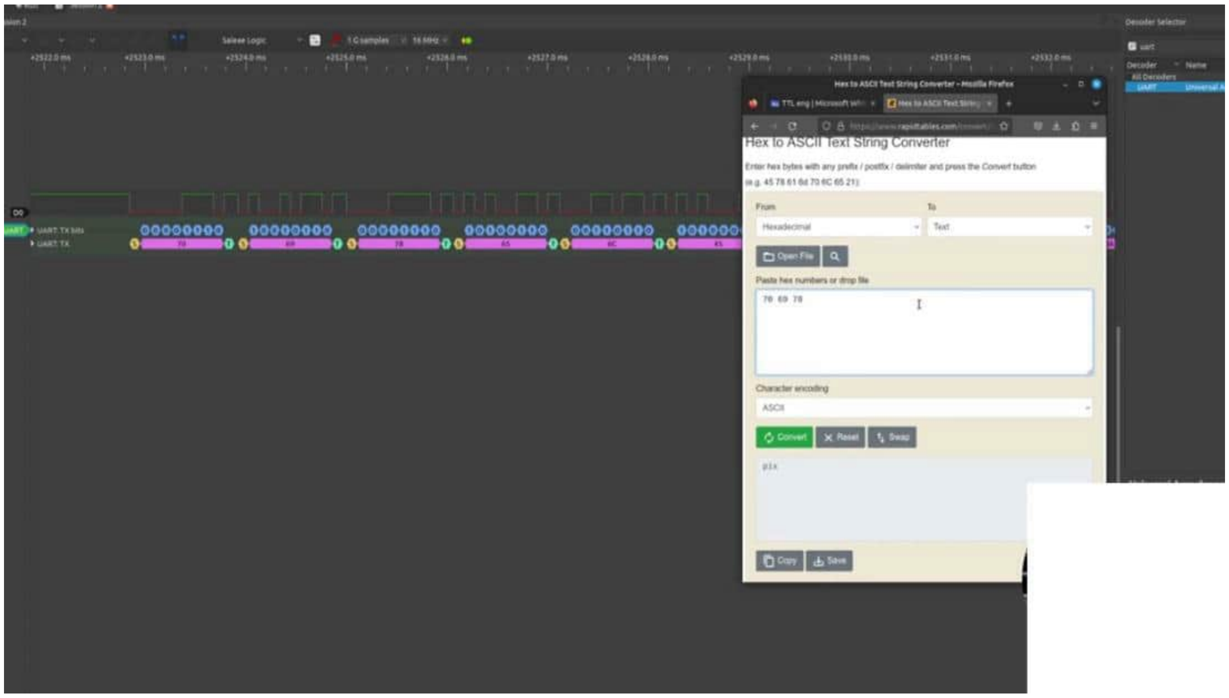


Name: UART
UART
RX (UART receive line):
TX (UART transmit line):
Baud rate: 115200
Data bits: 8
Parity: none
Stop bits: 1.0
Bit order: lsb-first
Data format: hex
Invert RX: no

The baud rate, let me see what we have set up, is 9600. So I am zooming a little bit in. As you can see here I receive the data but not really something that can be interpreted.

So therefore we have to change here to 9600. Enter. And this looks a little bit more like data. Because now I can see here, ah, here is my start bit. Then I have the data 0, 0, 0. The logic level changes to 1, 1, 1, 1, 1, 0 and then stops. Then the start bit follows and the next data is transmitted. And this will be one time frame and this should be the text Pixlady. But we can't see it because this is just, here we have the binary data, the raw binary. And this is some hex data. And now of course we can search the internet for decoders here.

So for example we are typing in here 76978. What do we get out here? Pix, that looks promising. 65, 60 and so on and so on. Pixl, that's good, we can also get all of the data when we click here on the binary decoder output view. Then we have UART here. I would like to have a TX dump. And I would like to have it here, is it RX dump, let me see, TX dump. Here we go.


Hex to ASCII Text String Converter

Enter hex bytes with any prefix / postfix / delimiter and press the Convert button (e.g. 45 78 61 6d 70 6C 65 21)

From: Hexadecimal
To: Text

Paste hex numbers or drop file
```
70 69 78
```
Character encoding: ASCII

```
pix
```

But I don't want to have all of the data. The last one is 49, so maybe I can copy this one. Copy and paste it, yes. Paste it here, convert it and I get Pixlady here out of Pixlady.

But there is an even more convenient way. So we click here on UART and we go a little bit down. So we have a data bit set of 8, this is ok. Priority bit is not set. We have a stop bit. The bit order, we come to this in a later chapter. And here we have the data format. We can change it to ASCII. And as you can see here, I close this window. Now we have the ASCII characters directly in PulseView. Here we have some checksums etc. But here this will be transmitted. Also in lower and upper case. In this menu we can also invert the RX and TX line.

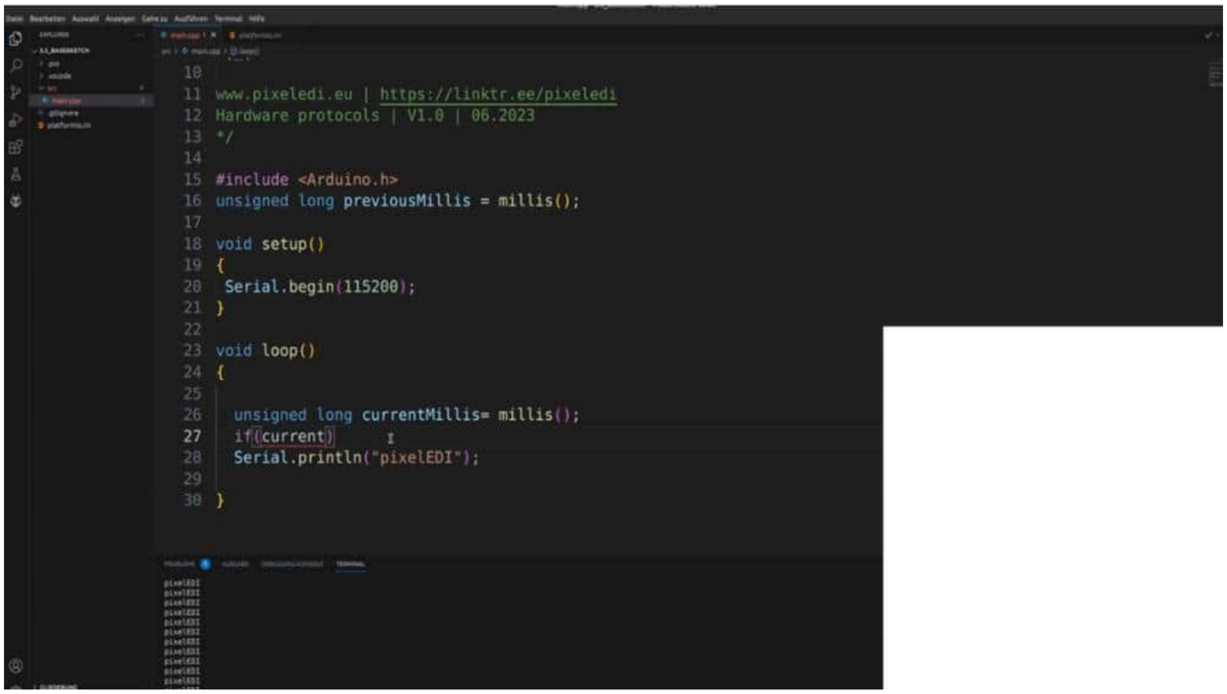
This is often used in circuits where we have added some transistors. So maybe we have changed the logic level here. So that we have the right logic level. Then we can invert it here in the program. So now I think we have discussed nearly every step here in this menu. And also how we can translate UART text into PulseView.

HEX TO ASCII WITH ARDUINO UNO

And to get a little bit of practice, I would like to show you how we can send hex data from the audio new scratch to Palsphere, so that we get the ASCII from HEX. So in the other way around. Therefore, we are changing our setup a little bit just for training purposes. We are changing the softness to hardware serial. So, yeah, to KX pin, then I'm getting rid of all of the software serial stuff.

And I'm adding here an order variable, and it's called unsigned long previous milliseconds, and I'm assigning here the current run time in milliseconds. And this is the proper way to get rid of the delays because delays are bad. They are blocking our whole sketch, and I'm sure you'll be familiar with the logic blink without delay. If not, just search. Make a quick Google search.

The AidenO CC site has a good explanation here before. For it. So here we have to begin. That's okay. Now, we're getting rid of the delay.



```
www.pixeledi.eu | https://linktr.ee/pixeledi
Hardware protocols | V1.0 | 06.2023
*/

#include <Arduino.h>
unsigned long previousMillis = millis();

void setup()
{
 Serial.begin(115200);
}

void loop()
{

  unsigned long currentMillis= millis();
  if(current)
  Serial.println("pixelEDI");

}
```

```
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
pixelEDI
```

And Now we have if the current mill is, so we have to define it here. So it's also an unsigned long because it's really a long number. The current millis is also millis. And if the current millis minus the previous milliseconds. Greater than.

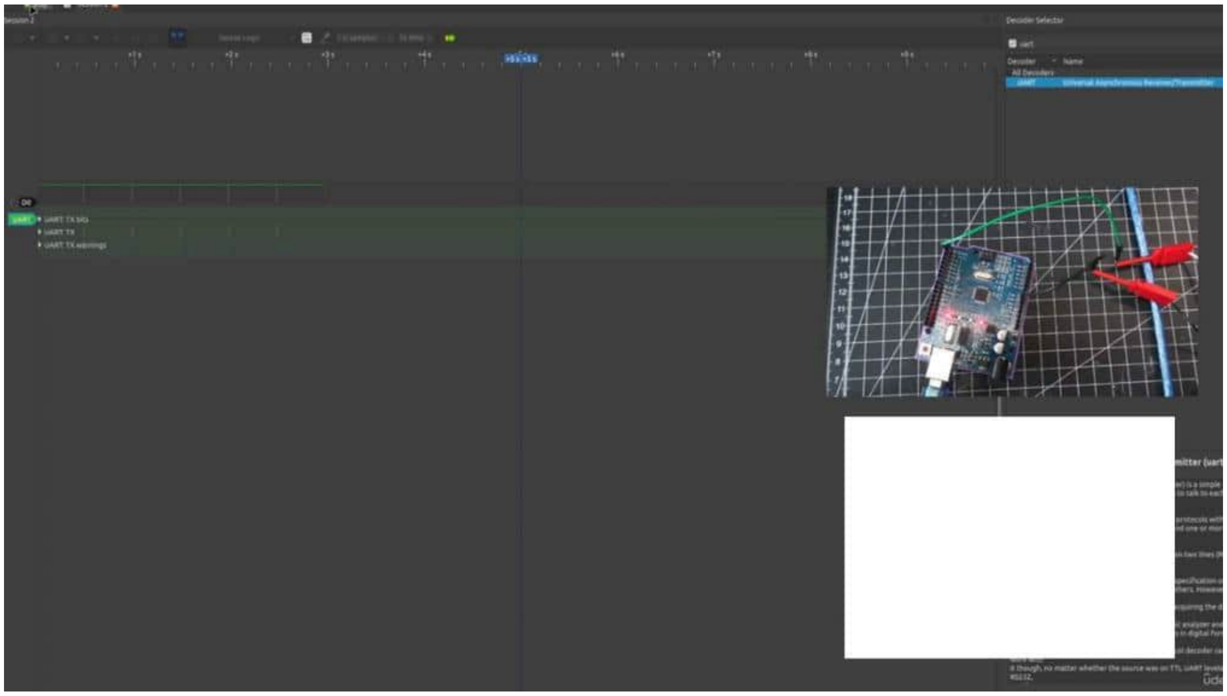
For example, let's make 300 milliseconds or 500 milliseconds Then, something should happen. And inside this one, we have to set the previous millis to the current milliseconds so that in the next loop, my function will work. So that means every 500 milliseconds this condition will be true because the current millis minus the previous millis has a differentiation from more than 500 milliseconds, then it will be set to the current milliseconds. So when we have the next loop, this will not be true and so on and so on after 500 milliseconds, then it's true. This is really a good way to code in our arena environment.

So now I'm not want to send the raw ASCII jar here. I would like to send here the hex values and then, see on policy what we get. So that means I am sending here an 0.write And now I'm indicating here with an 0 and x that I will send here an HEX value. And let me see what I have here, those values. So, let me see. Could I have your text?

Then I would like to change something here, but not ask you, could I change it to, for example, so there we go. We have to change the text here to hex editor. And before I had here the bigger case, bigger letters for EDI and now I would like to change it so that we have here, other outputs. So the last three should be changed to this one. I copy paste it.

coming back to the audio code. And now, I would like to add here those hex So let's paste in here a few times. Then we have 69 here, filling out all of the points And I have already uploaded the code. As you can see here, I have no line break anymore. I'm just sending the serial data out, to the monitor, and it will be converted directly from the ceremony tone to ask this, but let me see what will happen on our Are doing Wono?

So I click on run, and we capture here some data, click on stop, zooming in, And it doesn't work. Why? What have I changed? I have before, I had a software 0 with 9600, and this is what I have to change here as well. So 1100.

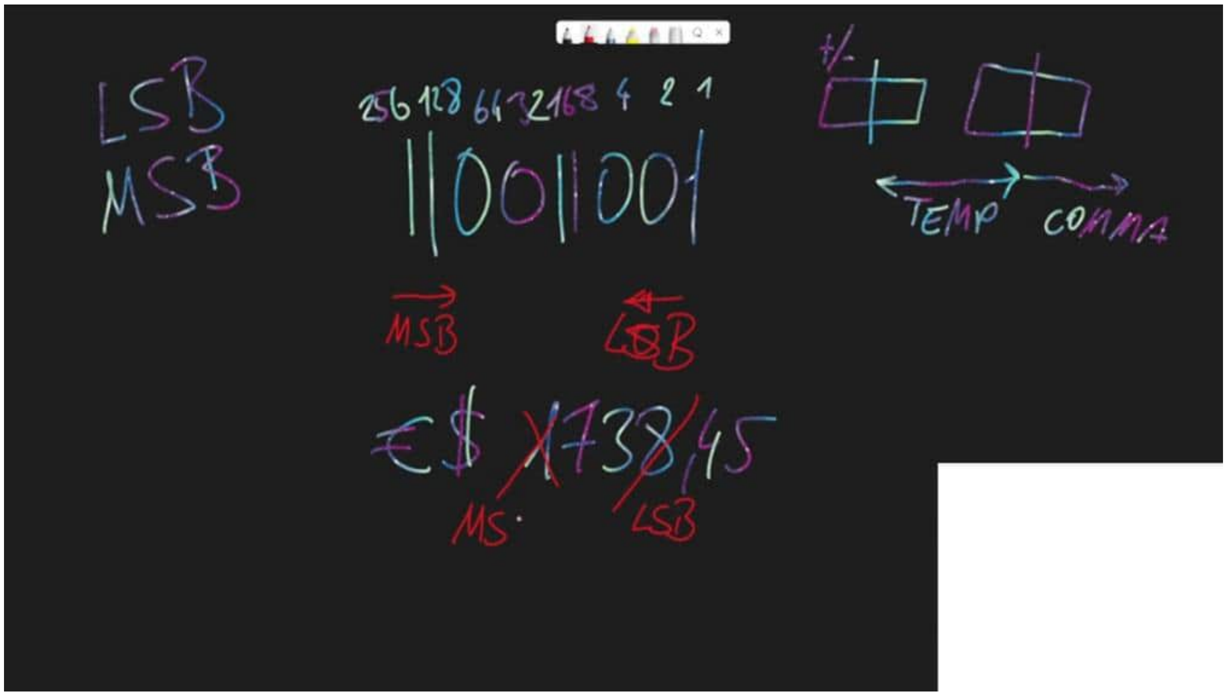


And if I change the portrait, it also shows, we can see here the mask is converted. So we are sending here 1, 2, 3, 4, 5, 678 signs from hex, and this will be translated into ASCII's as well with star and a stop bit. And this is how we can transfer data from binary hex, etcetera, to an auto module via u art.

LSB AND MSB

Let us now discuss the concept of LSB and MSP, and we have already heard these terms before. And it's called least significant bit and most significant bits. So, for example, if we have the binary level, the binary number of 11001100 and also n The terms of least significant or most significant bit refers now to the position and the significance of bits within a binary number. So what does that mean? That mean?

For example, if we got this number, we can read it from this order, or we can get the values in this order, but we have to read it in another order because it's called the least significant bit. And this depends on the datasheet and the manufacturer, how they will transmit the data to us. And when we are switching back to pulse view and click on UART, we can see here the Bit order. Is it LSB first? or is it the most significant bit first?



And if you change the order here, we will also see that something will change in our outputs. And therefore, this is a necessary event that we know what is our least and most significant bit. And normally, we are translating our values here. So for example, this is 124, 860, 32, 64, 128, and 200. 56.

And now we would add here on each position where is in 1 at the numbers, and this is our final number, what we are transmitted. But always keep in mind in which order we have to read it. And why is that so? For example, if we have here an register from a module 8 bit, and this is split it in half because here we have the plus minus sign so that maybe we will get values with and sign. And now also the second register is split in half.

And now This one, this content is our temperature value, and then the comma value will be attached to the last four bits and the next register. So it's not every time it's so clear what kind of information we get. And we will see this later on, especially in the eastward seas chapter with the DHT 20 evaluation of the sensor that the datasheet shows us where and in which order we have to read it. And just keep in mind what is most at least significant, but keep in mind when you have some dollars or euros in your bank account, And, for example, I am deleting you on the least significant bit. It has not so much input into your life.

But if I'm deleting you on the first number in your bank account, so the most significant bit 20. This has a tremendous impact on your life. And therefore, the least significant bit should always be considered in order to read out the right radius.

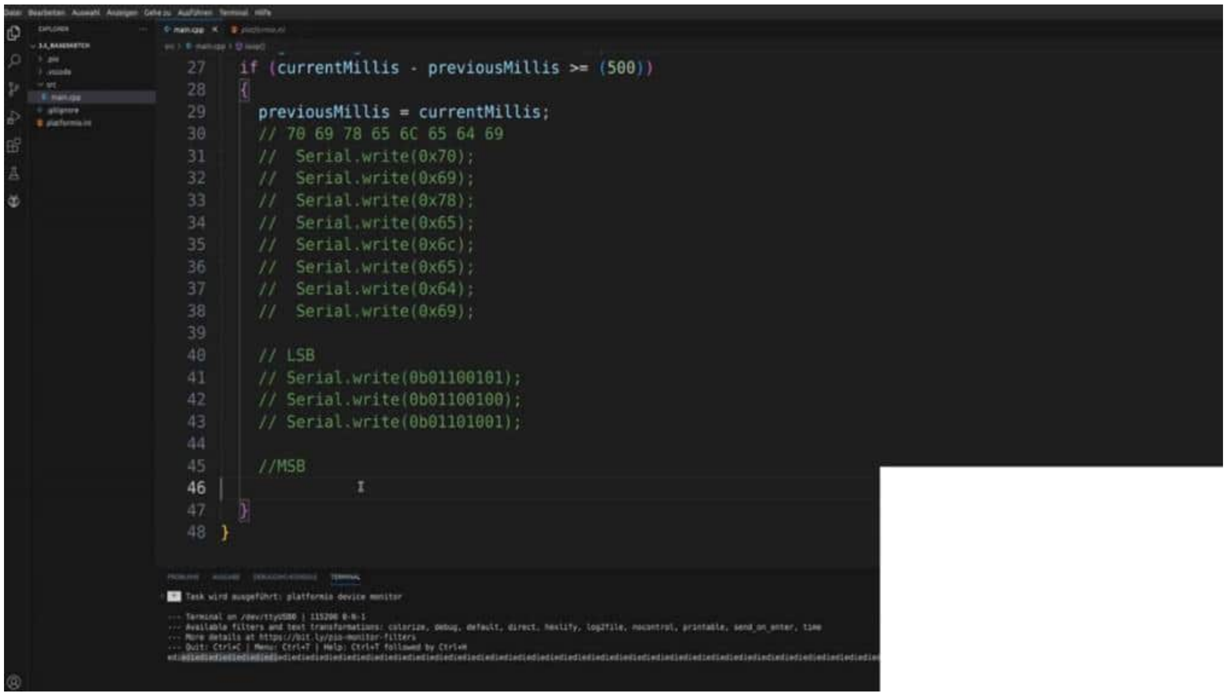
BINARY LSB AND MSB WITH ARDUINO UNO

Let's now see a real time example with all of this LSB and MSP topic. And I would like to send binary data here. Only three characters. So my name, Eddie, and these binaries, have already prepared. So let's switch back to our code from before.

uncommented all of these hex things. And now I would like to send the least significant bit before as well as first. And now I would like to send via 0 dot write. And now I'm indicating within 0 and MB that I'm sending binary values. And my first binary value will be this 1.

So 0, we also, I think, could we use Let me see if we can extract it here. This was not right. And decoder, I would like to have t x. It's only the hex values that I am getting here. Okay.

hexdump. No binaries. Nevertheless, I'm typing all of the stuff in. So let's see the least significant bit ready here. Let's upload the code.



```
    if (currentMillis - previousMillis >= (500))
    {

      previousMillis = currentMillis;
      // 70 69 78 65 6C 65 64 69
      //  Serial.write(0x70);
      //  Serial.write(0x69);
      //  Serial.write(0x78);
      //  Serial.write(0x65);
      //  Serial.write(0x6c);
      //  Serial.write(0x65);
      //  Serial.write(0x64);
      //  Serial.write(0x69);

      // LSB
      // Serial.write(0b01100101);
      // Serial.write(0b01100100);
      // Serial.write(0b01101001);

      //MSB

    }
}
```

```
Task wird ausgeführt: platformio device monitor

--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocontrol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
ediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediediedie
```

The serial monitor should already, converted to, yes, converted. Now we click on run because we have already set everything up from before, zooming a little bit out, and you can see I receive you some data. Then we click on stop zooming a little bit in, And of course, we're getting these 3 characters as before. What changed is that we, not the hex values, we are using binary levels. Now, I would like to change the order.

So what do we do now? copy this one, then the value from the first character will now be changed in the order. That means now I have 011 at the beginning, and now I start with the ending. So my new number will be from this direction, from the, 1. So let's see.

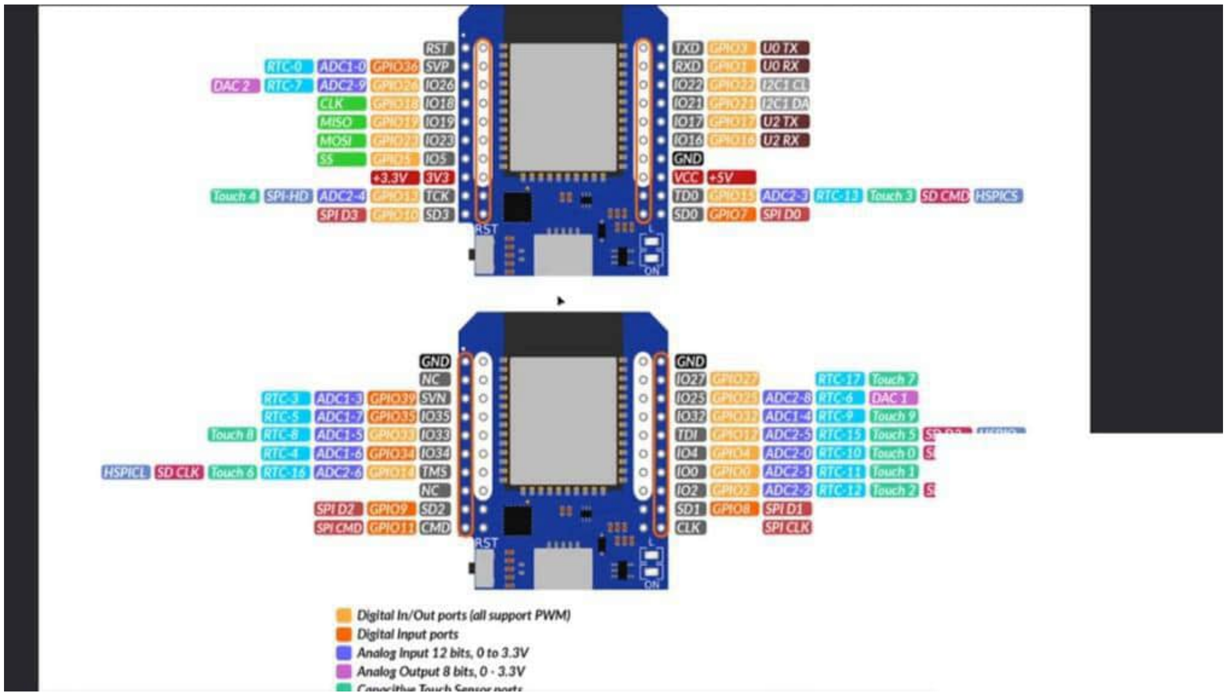
0101. 1 1. Then I have 000110. Then on the second number, I have here 1, then 0011 0. And in the last one, I start with 1 0010 110.

Then we're uploading, once again, the code. Let's see what the serial monitors are printing out. If we convert it, no, because the most significant bit order is now not recognized. Let's see here pulse view does, assuming we are out, we're getting here the new data, assuming a little bit in. And now I haven't changed anything.

So you can see the same thing as we have here some data which is not recognized and the ampersand. And now I'm changing to the most significant bit first. And the order will be changed. And also, the pulse you can now read the values. And I hope with this practical example, it's now clear from which side we are reading the data and that we can change it also in Pile Sphere or in the other Unity, how we want to read it.

Hardware Serial with ESP32

In this project, we're gonna try to use the u art interface with an ESP32. And I've here the ESP32D1 mini, And it's nearly the same as the development board, just another layout and size of the board, but you can do it with any kind of ESP32. Be aware the pinout can also differentiate between the development port or another ESP rum, etcetera, because often the manufacturer or the company which is delivering you the ESP, changes the layout. I don't know why, but there are minor changes. So be aware, especially from asset delivery, I saw some different pinouts out there.



RST
RTC-0 ADC1-0 GPIO36 SVP
DAC 2 RTC-7 ADC2-9 GPIO26 IO26
CLK GPIO18 IO18
MISO GPIO19 IO19
MOSI GPIO23 IO23
SS GPIO5 IO5
+3.3V 3V3
Touch 4 SPI-HD ADC2-4 GPIO13 TCK
SPI D3 GPIO10 SD3
TXD GPIO3 U0 TX
RXD GPIO1 U0 RX
IO22 GPIO22 I2C1 CL
IO21 GPIO21 I2C1 DA
IO17 GPIO17 U2 TX
IO16 GPIO16 U2 RX
GND
VCC +5V
TD0 GPIO15 ADC2-3 RTC-13 Touch 3 SD CMD HSPICS
SD0 GPIO7 SPI D0
RST
ON
GND
NC
RTC-3 ADC1-3 GPIO39 SVN
RTC-5 ADC1-7 GPIO35 IO35
Touch 8 RTC-8 ADC1-5 GPIO33 IO33
RTC-4 ADC1-6 GPIO34 IO34
HSPICL SD CLK Touch 6 RTC-16 ADC2-6 GPIO14 TMS
NC
SPI D2 GPIO9 SD2
SPI CMD GPIO11 CMD
GND
IO27 GPIO27 RTC-17 Touch 7
IO25 GPIO25 ADC2-8 RTC-6 DAC 1
IO32 GPIO32 ADC1-4 RTC-9 Touch 9
TDI GPIO12 ADC2-5 RTC-15 Touch 5
IO4 GPIO4 ADC2-0 RTC-10 Touch 0
IO0 GPIO0 ADC2-1 RTC-11 Touch 1
IO2 GPIO2 ADC2-2 RTC-12 Touch 2
SD1 GPIO8 SPI D1
CLK SPI CLK
Digital In/Out ports (all support PWM)
Digital Input ports
Analog Input 12 bits, 0 to 3.3V
Analog Output 8 bits, 0 - 3.3V

And when we take a closer look here on this pinout, we can see we have a serial interface here. And we can find it here on the upper right side, TXRx. This is our interface 0 because the ESP cert user has 3 different hardware series. And we can find here the uart 2. So where is uart 1?

This will be used also for the internal connection between the USB and the ESP. So this is not available for us, but we can use the second one here because the With the index 0, it's pretty straightforward. We can also find the Rx and TX. So let me see. it's here on where it is?

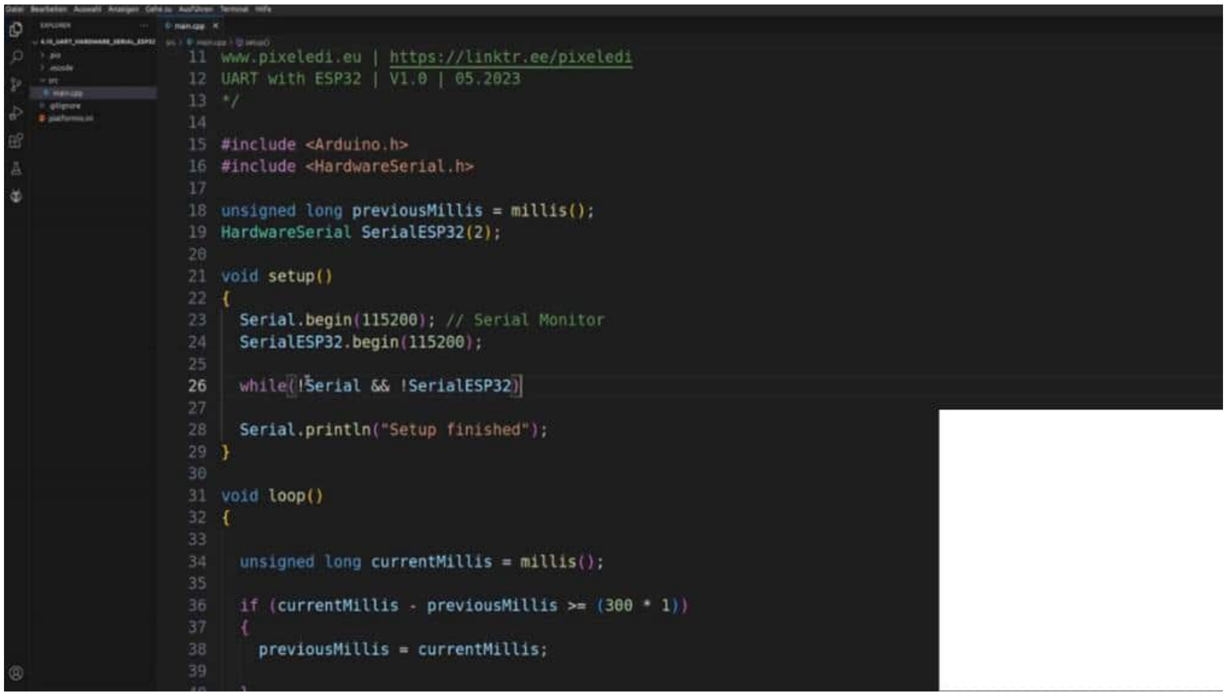
rxdx. We can find it also on the pin out here. But I would like to use another kind here, the u 2, and, therefore, we are using the pin number 17, GPO 17, for our T X communication. So let's jump into our basic sketch. And what we do at the first point is we are including the hardware serial library.

And as always, this is in the core from the ESP 30 to include it. So we don't have to install some external libraries here. We just have to include And then we're starting with hardware, serial. Then we give the name of, for example, serial ESP 32. And here in the brackets, we have to type in which number we want to use.

And according to our pinout, we are using the 2nd u at interface. If we want to use the normal hardware, serial port, then we have to enter this 0. So far, so good. Nothing special until now. And now we start with 0 ESP32 begin, and we can also use here the faster bulk rate from 115200.

And as you can see, I have a normal serial beginning and a hardware serial. And then we could, for example, here, type in 0 print lines, set up finished. What we also can do It's just show you all of the possibilities. We can say, while, if the 0 and the serial ESP. So if they are not available because if we use these terms, then we're getting back here and true.

But if there is not, So if it's false, then we keep here in a while loop. This gives us the proper ability to check if the serial communication could be established or not. And if so, we go through the loop. Here's our link with our delay. construct.



```
www.pixeledi.eu | https://linktr.ee/pixeledi
UART with ESP32 | V1.0 | 05.2023
*/

#include <Arduino.h>
#include <HardwareSerial.h>

unsigned long previousMillis = millis();
HardwareSerial SerialESP32(2);

void setup()
{
  Serial.begin(115200); // Serial Monitor
  SerialESP32.begin(115200);

  while(!Serial && !SerialESP32)

  Serial.println("Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (300 * 1))
  {
    previousMillis = currentMillis;

```

And we want to have here, for example, each 300 milliseconds. I would like to print out a serial print line on the monitor. Hello, serial monitor. And I'm the serial s p 32. Should print our handler, pulse view.

Then let's see. If we have some typos or it's the compiler, yes, we get the success, then we upload the code. So then we can open up the ceremony time. And we can see the output from the serial monitor. And now we are changing to our past view.

I've already inserted my device here. changing here the sample rate. We click on run. Then we're adding here just one channel. Then adding in a decoder, it's an URL decoder as we did before.

Then new art that takes because we are sending something. 11 looks good. And I would like to have the ASCII. Then we're tuning into one of the data frames. and we're getting here Hello, pulse view.

Perfect. So in this project, we learned how we can use the hardware serial on an ESP32, and we can change the channels or which hardware serial we would like to use and also that we can combine it with the serial monitor, and the hardware serial on an ESP32 level.

WHAT IS A PARITY BIT

Now, it's time that we talk about error detection in UART. And therefore, we have the parity bits. And we are focusing on the even parity. So what does that mean? The parity bit is a bit that is added before the stop.

and tells to receive what is the number of ones in the whole transmitted number, the binary numbers, is, odd or even. The possible settings for the parity bit are odd or even. So odds the parity bits are one if there is an odd number of ones in the data frame. Even this is what we are using, the parity bit is 0 if there is an even number of ones in the data frame. So let's take an example.

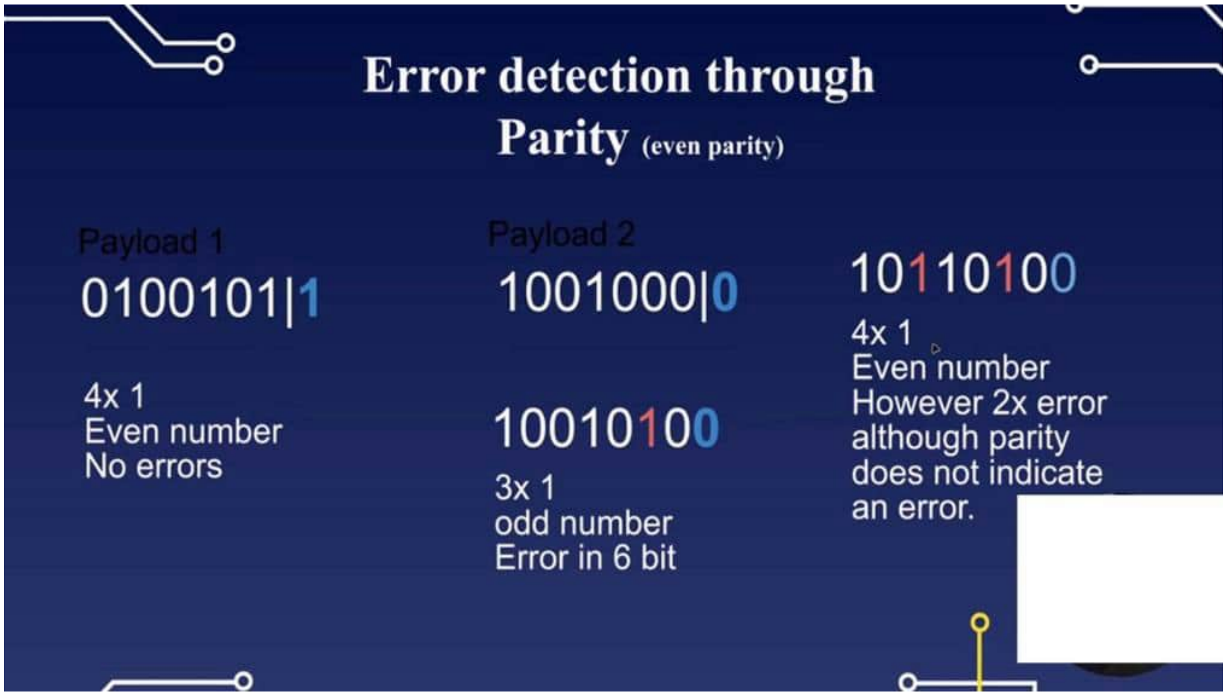
For example, we have the following payload. We have here 1 byte 12345 67. And this is, the parity, the bit. So that means we have in our data frame 1, 2, 3 ones. And, therefore, we are in an even parity bit order.

We need to add in the parity bit on 1 so that we have four times on 1, and that's an even number. Let's take a closer look at the payload to them. Here we have 2 ones in my data frame. And I'm adding here an 0 to this is the parity bit. This is an additional bit, and therefore, we're using here and 0 because we have 2 of the ones already here.

This is an even number, and they're SSN 1. So in this case, we have now something in the signal, some some arrows, and therefore, we have here 31 So this is my arrow. And the priority bit is 0 because this was intentionally sent. Then we can detect an error. So three times 1, odd number, error is in the 6 bit.

Cool and easy stuff. But hold on. There is a little, I don't want to say misconception, but there is, this error detection is not really reliable because what if we have some case. That means we have sent

100100 here. That means we have an even number of ones. Therefore, the priority bid is 0.



Error detection through Parity (even parity)

Payload 1

0100101|1

4x 1
Even number
No errors

Payload 2

1001000|0

10010100

3x 1
odd number
Error in 6 bit

10110100

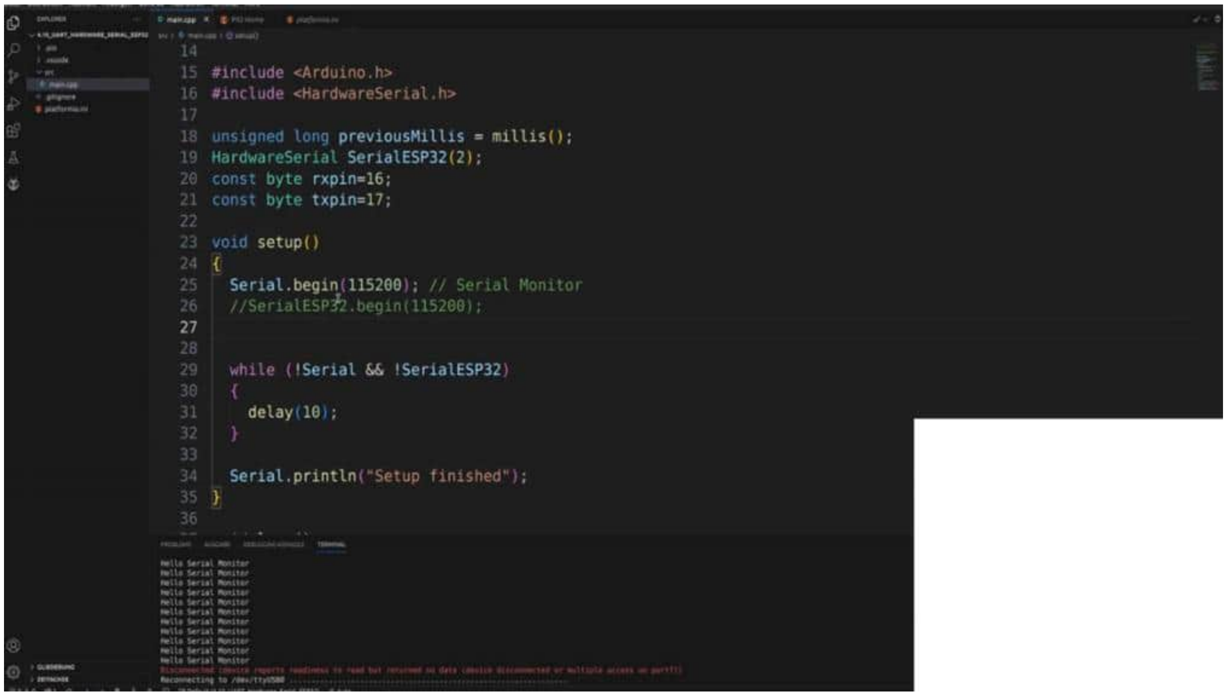
4x 1
Even number
However 2x error although parity does not indicate an error.

But what if we have such an incident that when we have here 2 arrows at once and now we have here four times and 1, that's even as also an even number. And therefore, however, 2 errors are in our data frame, but the parity bit task does not indicate an error because it's an even number. So you see here the limitation of the parity bit But nevertheless, it's good when we implement it because we have a first inside view if this could happen. In other structures, which we will see later on, like in, some non standardized protocols with this th HD, for example, the they are using, checksum. That means we are adding you from the humidity and the temperature and in the checksum register, those two values will be edited.

And if we have the summary of those values, then we can recalculate it if the values are correct or not. This is also an interesting more reliable check and error detection. But it is good to implement it. And in the next project, I would like to show you how we can do that in the ESP32.

EXAMPLE PARITY BIT SERIAL_8E1 WITH ESP32

In this project now, we want to implement the priority bit in our sketch from the previous project. And I would like to show you once again, pulse here, And you can see here that the priority bit is missing because we haven't got implemented it. And therefore, this will now be our main goal from this project. and therefore on the Sketch from before. And although we are using here unfixed, you are the interface, I would like to define the pins because it's a little bit more readable and the whole setup from the code.



```
#include <Arduino.h>
#include <HardwareSerial.h>

unsigned long previousMillis = millis();
HardwareSerial SerialESP32(2);
const byte rxpin=16;
const byte txpin=17;

void setup()
{
  Serial.begin(115200); // Serial Monitor
  //SerialESP32.begin(115200);


  while (!Serial && !SerialESP32)
  {
    delay(10);
  }

  Serial.println("Setup finished");
}
```

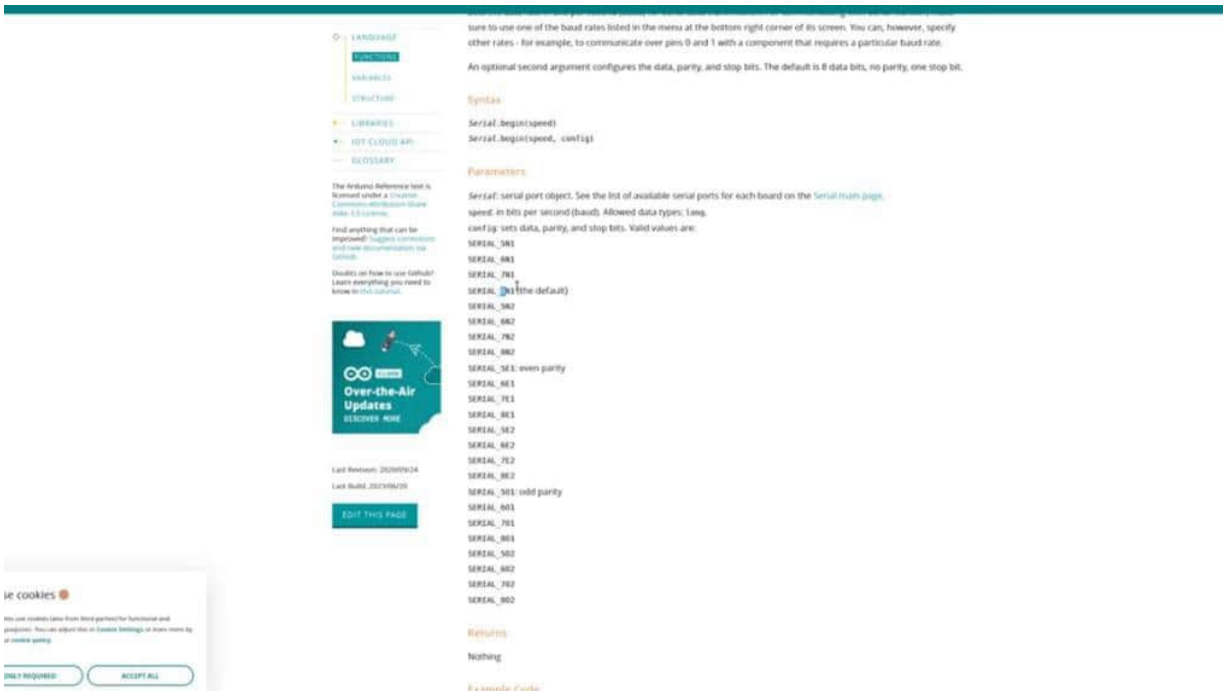
```
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Hello Serial Monitor
Disconnected (device reports readiness to read but returned no data (device disconnected or multiple access on port?))
Reconnecting to /dev/ttyUSB0 ......................................
```

And therefore, we are using here some variables, const byte RUXpin. And this should be 16. And we are from KonSpITE T X PIN, and this is 17. This is where our clamp is on for the logic analyzer. So now what we are doing is, all of the magic will happen in the beginning.

And therefore, it's a really good website from the audio in OCC reference with the serial begin explanation. and we can't find different kinds of 0 here. So, what we can do here with the 0 begins. This is like a function. And normally, we are just adding here one parameter, and the parameter is the bounce rate speed.

But we have even more possibilities here. For example, set data, parity, and stop it. And these are all the things that we can do. So for example, this is the default. The default one means we have 8 bits in length, no parity bits, and 1 stop bit. So in our case now, I'm switching back to the code.

What we do now is 0 ESP32. dot begin. And now with this intelli sense, you can see what parameters I can fit in. And the first one is the board rate. So you can see here in an unsigned long portrait.



sure to use one of the baud rates listed in the menu at the bottom right corner of its screen. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate.

An optional second argument configures the data, parity, and stop bits. The default is 8 data bits, no parity, one stop bit.

Syntax

```
Serial.begin(speed)
Serial.begin(speed, config)
```

Parameters

Serial: serial port object. See the list of available serial ports for each board on the Serial main page.
speed: in bits per second (baud). Allowed data types: long.
config: sets data, parity, and stop bits. Valid values are:

```
SERIAL_5N1
SERIAL_6N1
SERIAL_7N1
SERIAL_8N1 (the default)
SERIAL_5N2
SERIAL_6N2
SERIAL_7N2
SERIAL_8N2
SERIAL_5E1: even parity
SERIAL_6E1
SERIAL_7E1
SERIAL_8E1
SERIAL_5E2
SERIAL_6E2
SERIAL_7E2
SERIAL_8E2
SERIAL_5O1: odd parity
SERIAL_6O1
SERIAL_7O1
SERIAL_8O1
SERIAL_5O2
SERIAL_6O2
SERIAL_7O2
SERIAL_8O2
```

Returns

Nothing

Example Code

The Arduino Reference text is licensed under a Creative Commons Attribution-Share Alike 3.0 License.

Find anything that can be improved? Suggest corrections and new documentation via GitHub.

Doubts on how to use Github? Learn everything you need to know in this tutorial.

Last Revision: 2020/09/24

Last Build: 2023/06/20

se cookies

[illegible]

Then I would like to add the serial and also you can see what I can add here. And the standard was 88 and 1. But I would like to add here an even for the even priority bit. 1 stop bit and it's 8 bit long. as we can see here, 8 bits long, and one stop bit.

So, therefore, we also have to add here that we have a complete parameter. All the parameters complete, Rx and TX PIN. And that's it. That's all of the change, what we are making. The rest should be enough so then upload the codes.

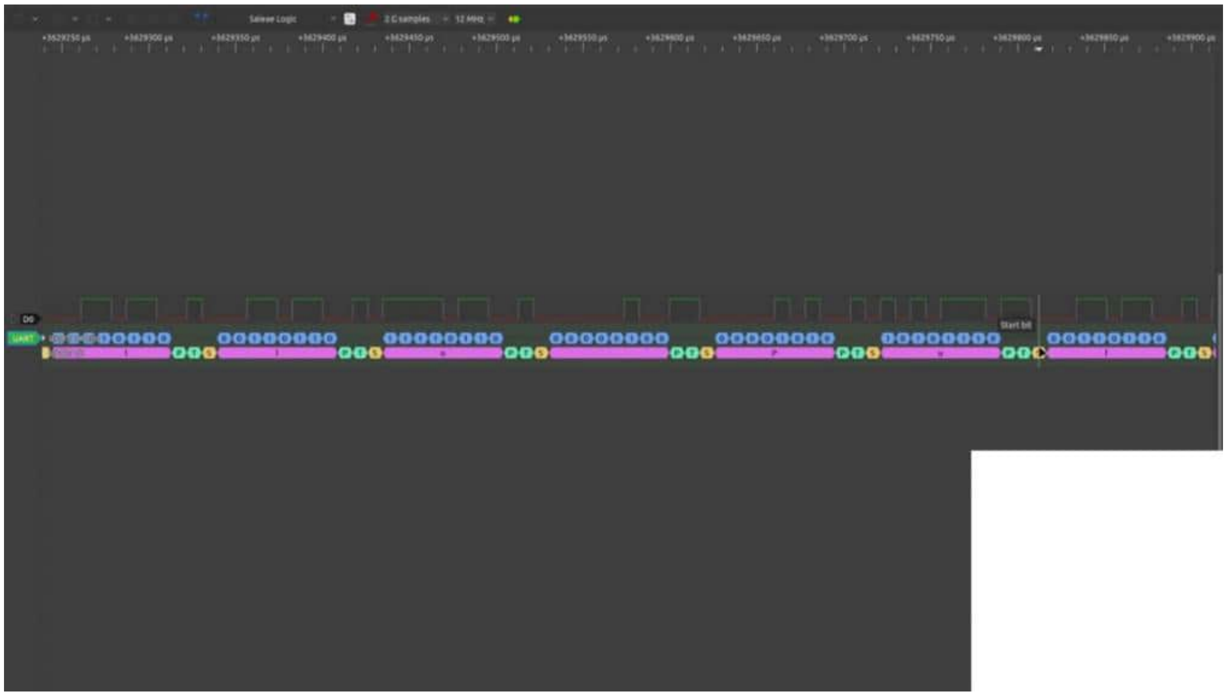
And before we run the pulse view, I would like to add the pulse view from the front. And let's see here in the options, when we go a little bit down, where is the parity bit here stands on. When I'm clicking an event, Then you can see that nothing will happen because we have not really impaired a bit. Here, parity error because we haven't transmitted any parity bits. And, therefore, we get an error here because 1, 2, 3, 4, And we haven't got any parity bits here.

That's an error. So let's see if we have uploaded it here. Yes. Looking good. Therefore, I'm just clicking on run because the setting should be the same.

a lot of data coming in, zooming in a little bit to one of the new data frames. And I have already entered the parity bit. So let's investigate if we are true. If we have everything set up here. So, We have 1111, so no parity bit is here, 0 as we can see in the line.

That's the check. 123456 is also even. He's only 1 So the parity bit has to add 1 because then we have 2 ones and this should be an even number. And as you can see, the parity bit is here 1. and, therefore, we have a positive check.

Let's see here. 11. So the parity bit is 0. And as you can see here, 1, 2, 3, 4, 5, that means an odd number. We have to add here the parity bit to 1, and we also have an even number.

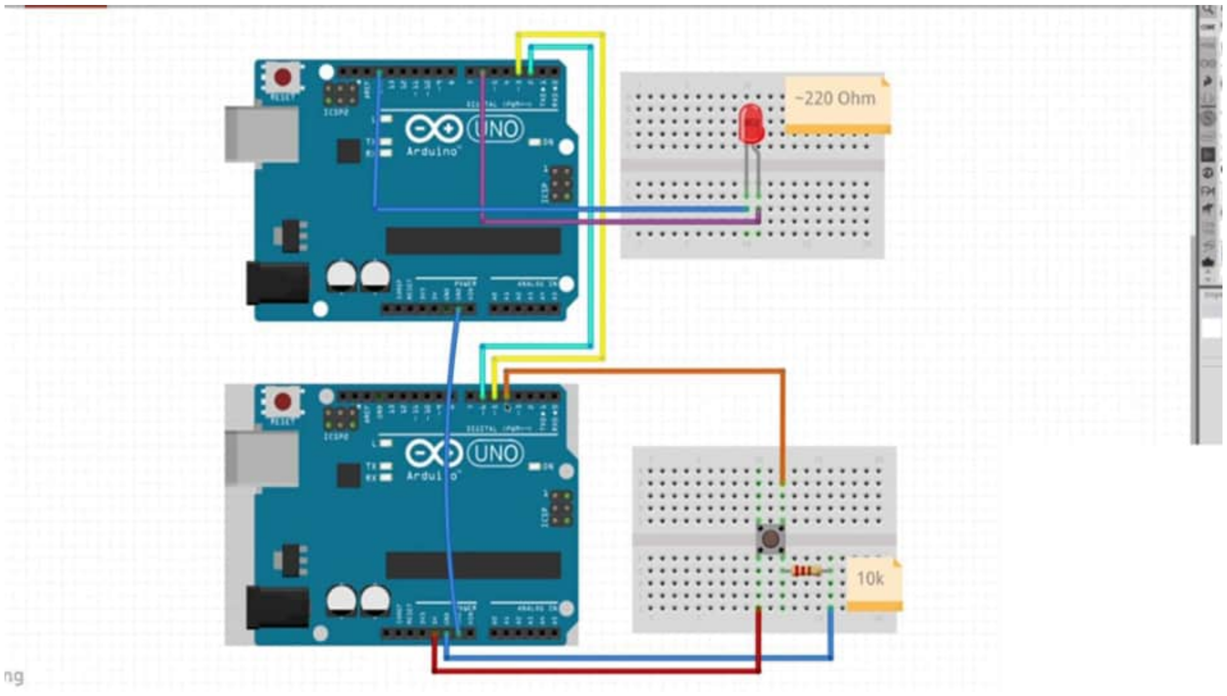


Very good. And this is the implementation of what we could do with the ESP32, for example, to add the parity bit to have here a little bit more control of our data frames if everything is working well. And afterwards, for example, if we're transmitting it the data from an ESP or from a module, we could do in the code and check if the parity bit has an error or not, or if, we have to calculate the numbers of the ones, and then we can check if the parity bit is here and even or not number in summary. And this is why we are using the parity bit also in the code.

UART COMMUNICATION BETWEEN TWO ARDUINO UNOS

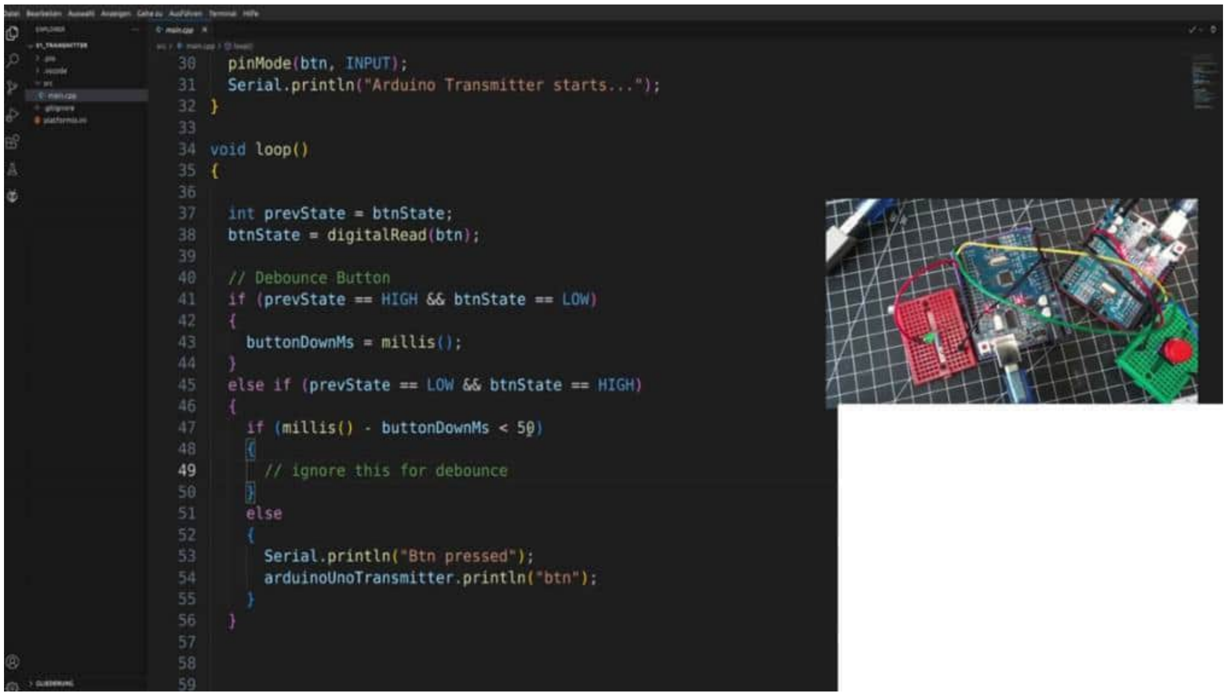
Let us now make some practical examples. And therefore, I have 2 otherinos, which are connected via UART, and the main goal of this little project is now that when we are pressed the button from this auto in order, we're sending and you add a message. This will be received by the second order, Uno and an LED will be turned on. I'm clicking the next one, turned off, turned on, turned off. And this should give us now the knowledge that we are able to send data via UART, what we already know, and receive data on an ARENA basis.

Yeah. And therefore, I would like to start with the wiring part. And we have here the 2 other owners. could also be implemented with ESPs, but be aware, not UNO and ESP, this will be covered in the next project. So first of all, we have a fivefold and ground to the button.



We have here a pull down button, resistor And then we are connecting the button to the GPO 4. 56 will be our uart lines. which are connecting to 3 and to the 2nd owner, this is the receiver. We have to connect ground and ground from the auto in owners. And here, we have an LEDM.

Also, insert a resistor for the LEDM. That's, for roundabout 200 ohms. And this will be connected to this GPO number 6 and to ground. And that's it. So we are simulating that we are, sending data and what often is the case that we are receiving data.



```
  pinMode(btn, INPUT);
  Serial.println("Arduino Transmitter starts...");
}

void loop()
{

  int prevState = btnState;
  btnState = digitalRead(btn);

  // Debounce Button
  if (prevState == HIGH && btnState == LOW)
  {
    buttonDownMs = millis();
  }
  else if (prevState == LOW && btnState == HIGH)
  {
    if (millis() - buttonDownMs < 50)
    {
      // ignore this for debounce
    }
    else
    {
      Serial.println("Btn pressed");
      arduinoUnoTransmitter.println("btn");
    }
  }
```

So our receiver is now our main focus in this project. So, therefore, I have it. I have 2 sketches, and I would like to focus now on the transmitter because this is really an easy one and we already deal with it. So the receiver is here on the right side. I've already implemented the sketch here. And I have 2 USB ports here.

So, with the platform you own, don't forget to edit here the USB numbers and the sender. has now on software 0, all what we did before, and input for the button. And what I have here is a little debunking logic. And with this debounce logic, I tried to find out if there is some flattering with the buttons And if I get a signal under 50 milliseconds, it will be ignored. And if it's really a button click like here, Then I will wire you out the string button.

That's it. Nothing special. Everything we did before. But now I would like to focus on the audio on the left side where the LEDs are. And this is often a use case where we want to receive data from a module, but often also send and receive, but now we are just taking the receiver site.

I also did a software serial here. Whispin 2 and 3. This is what we have here. 2 and 3. Eric's and T X.

So you can see here the Rx pin is on 2. And here, the Rx pin is on 5. So 2 and 5. Let me see. 2 and 6 are now combined because we have the receiver to the transmitter and the receiver to the transmitter.

So let's keep on the receiver module. We have an LED on the pin gpio 6. We are starting with the baud rate of 9600. This has, of course, the same value as here. And we're setting the pin mode on output.

And now we are coming to the loop, and this is what we are, want to want to do now. So the first step is that we are checking if there is some if we are getting here some data. Therefore, we're making a while. And let's see if we received some UR data. And we are seeing dots available.

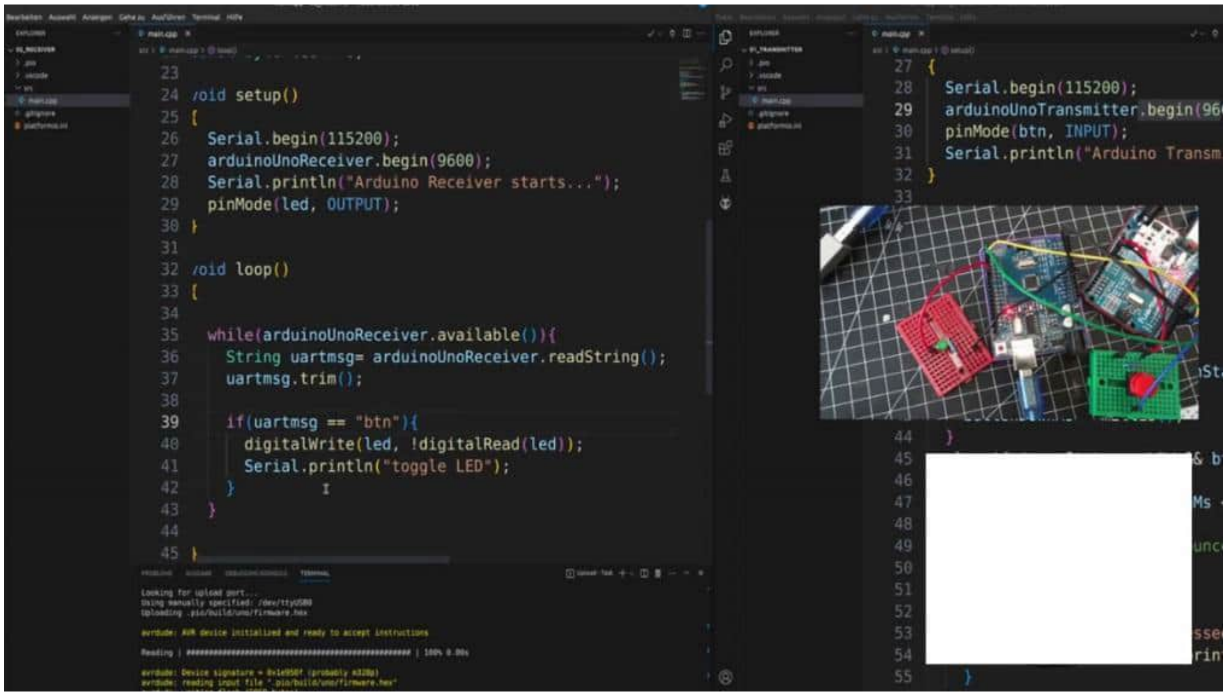
And then we are seeing, string uart message, for example. is serial.read string, and then we're printing out this message. Then we're uploading this code. And of course, we don't have to use a serial. We have to use it here. How do you know uno receive 1?

So now we can upload the code. And it's up. Then let's test. receive a start so that the setup message will be received, clicking one time, clicking 2 times, 3 times yes, it works. But what we see here, we have here some white spaces and line breaks.

we can get rid of them by saying, uart message.trim So now we should only get here the button as in string. And now we can, for example, say if we receive an uart message. uart message equals to button. because we are transmitting and stringing. And, therefore, we can check if this is on string.

And then we make a digital write here. And I also want a serial print line toggled LED. This will be LEDM and, for example, high or low. But what I do now here is I make a digital read, digital read, LED. So what I'm doing here, I'm reading out what is the current state now is false because it's not on.

And if I get here and false, it will be turned out of So I am transferred to true because I make a note in front. So I received it here. This is false with this line, it will be true. So I'm clicking on it, and now it's true. If it's true, the LED lights, then I get back from the digital read true, but with this negation, I make it unfold.



```
void setup()
{
  Serial.begin(115200);
  arduinoUnoReceiver.begin(9600);
  Serial.println("Arduino Receiver starts...");
  pinMode(led, OUTPUT);
}

void loop()
{

  while(arduinoUnoReceiver.available()){
    String uartmsg= arduinoUnoReceiver.readString();
    uartmsg.trim();

    if(uartmsg == "btn"){
      digitalWrite(led, !digitalRead(led));
      Serial.println("toggle LED");
    }
  }

}
```

```
Looking for upload port...
Using manually specified: /dev/ttyUSB0
Uploading .pio/build/uno/firmware.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: reading input file ".pio/build/uno/firmware.hex"
```

```
{
  Serial.begin(115200);
  arduinoUnoTransmitter.begin(96
  pinMode(btn, INPUT);
  Serial.println("Arduino Transm
}
```

So I can toggle the LED with only one line. Okay. Then upload. Let's see if it works. And we can do this in the next with you exactly in the same way, with an ESP32.

So we are clicking on our button. We are getting the toggle LED. Now when I'm clicking once again on the button, this expression will be true, but with the note, it will be false. So false is now the whole expression, and therefore, it will be turned off. Let's see.

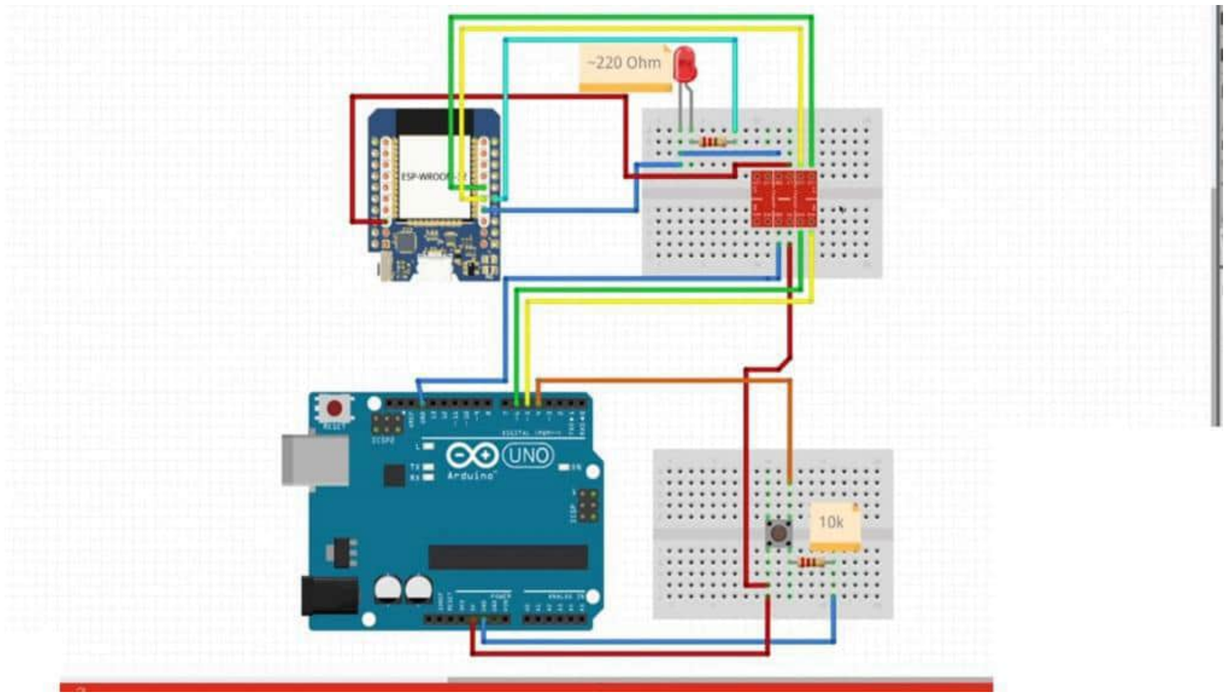
it's off clicking, clicking. And what I would like to show you in this project was or is that how we are able to communicate and receive data. Because often, we want to receive data from an UI module. This is exactly what we can do here. We also can add here and equal greater signs, greater than 0, and then, is also in a convenient way.

how you can fetch and get data from a module via UART. And in the next project, we take a look on how we can communicate with 2 different logic levels.

NOTE THE LOGIC LEVEL WITH ESP32 TO ARDUINO UNO

In this project, I would like to show you what we can do when we have different logic levels. So, the artery in the Oulu has some logic level from 5 worlds. What does that mean? If we measure here with a multimeter, the GPR outage, then we see the UNO has 5 worlds. Besides that, the ESP 32 has 3.3 volts.

That means we can't not connect directly the GPOs from an Audi in Uno to an ESP32 because the ESP32 is only able to handle 3.3 volts. So if we are using 5 volts and the GPO from the ESP32, It could happen that we destroy those ESP, GPOs, or even the whole balls. So therefore, we have to use logic levels. And this is a common case because often we have other kinds of voltage levels. For example, 1.8 volts It's very common with some power cons, reducing, modules, for example, in GNSS module where you can use just 2, double a or triple a batteries.



So therefore, we have to slow down, reduce the voltage, and reduce the power consumption. And when we have want to use UniN USB 32, then we have also used here, leverage And a l a level shifter works in a way that we are connecting the higher voltage and the lower voltage and, the transistor on it will then transform, I would say, the logics, to to the desired voltage And we, can't use directly here voltage dividers because we want to correlate and want to have the nuances and the details from our PDM signal, for example. And, therefore, these logic level shifters are not really expensive and easy to use. And in this case, we want to use the same example as before. We want to have a push button. We want to turn on an LED, and the LED is powered by the ESP32.

And we wire you out. We're sending as before and string. So how can we do that? We're using his software serial of the auto in the Uno site. And here, you can see We have a high voltage, for example, 1, 2.

We also can use the pins we inform. And this is the UART signal. Here, we have the five ports in the ground from the Uno, and, therefore, we have not connected or we don't need to directly connect the grounds together because we do it with the level shifters. You can see you with the ground connected. And we have 3.3 volts here.

So, therefore, the logic level shifter knows, 5.3.3 volt, what they have to do. And the UART We're now connected here to Eric's and T X. This is the 2nd channel of the hardware serial from the ESP32. And that's it. The LED with the resistor and the push button with the resistor is the same as in the level before, in this example before.

So now I have here 2 sketches. And as we had before, I just want to go through the transmitter very fastly. I have a software serial with Eric's and TX PIN. I created a software serial, The button is on the gp04. We have an input.

We're starting the ESP, the softer serum. And here's the logic for the debounce as we did before, and we are sending here our wire URLs the string button. But now to the more interesting part, And the more interesting part is the ESP side. So we're using heat hardware serum. This is nothing new for us.

We already did this on the channel tool. And now we already ah, let me see. We have and we should use here also, unconced bytes for the LED. And we have to use it in pin mode. The LED is in output.

Otherwise, it won't work. And also this kind of, checking if the serial is available, we already did before. So there we have to do nothing more. But now in the loop, the interesting part is how we can get the signal via UART. And we could say, for example, while, and in this while, but curly brackets.

In this while, we're seeing, serial. esp32 dot available, greater than 0. Then we will be getting actual information here. And then we could say, for example, string. We see the data is equal to, oop, that was consuming equals to, serial ESP32 dot read string.

I think this would be the proper function. And now we will get all of the information here in this string because we are sending a string. And I would say I would like to print out at the first step the received data, and we'll take a closer look if this will be our desired data. We also could do just for training purposes and that we are seeing what we are getting here. and wind, 8.

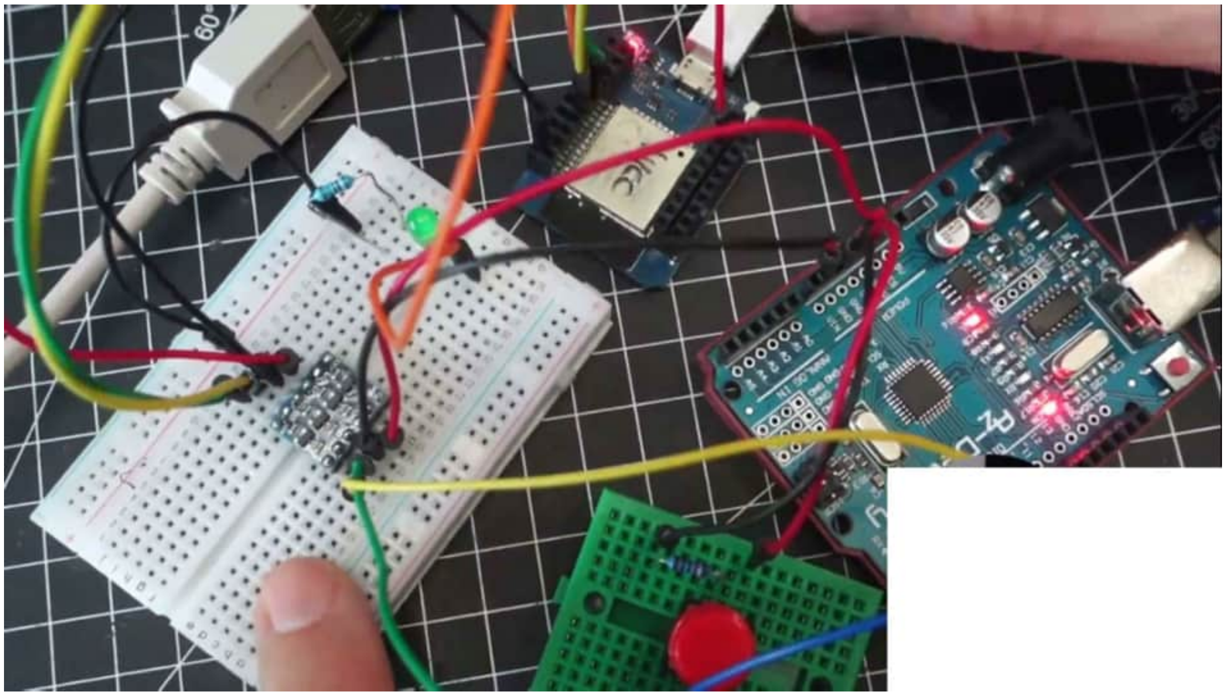
It's also in bite, but you can see both of them. bite from Syria equals serialesp32 dot read. So we're getting the raw bytes in this case. and it's an int8. Then, we can print out those bytes.

This will be the first step. Then upload the code here. And here is my setting with the audio in Uno and the push button. Here is the ESP32 with the logic level shifter as we talked before. And now I would like to click here on the button and in the serial monitor.

I'm starting the ceremony to write again that we see here something happened. And here is the raw data from our serial monitor. Now, we are changing back. I would like to have this string then upload the code. There we go.

Once again, button click, and we're getting the button. Nicely. Oh, but we got here in one line break. So, therefore, I would suggest that we, at the Receive data. We could say receive data dot trim.

That will delete all of the white spaces and line breaks. And then we could say if received data equals the string, what we want to to check. In our case, it's the button that we are sending out with the audio. If this is so, Then, we could switch and turn on the LED. And, therefore, we are toggling the LED.



And we could say, digital write, LED. And now we make a little trick. and we are saying Digital Read LED. So we are not finished yet, but I would like to explain what I'm doing here. Normally, I would write here in high or low.

But now, I would like to read the current state of the LED. If I do that now, I will get here in law or enforce because it's not on. So if it's turned off, I get back here and false, and I make a knot here an So I convert it, I make a knot in front of it. So if this statement is false, Then I convert it to true with this knot. If it's turned on, then I'm getting here and true and with this knot, I make a false out of it.

So we always invert the current state and therefore, we are using it on one liner and toggle the LED. And that's what we have to want to do. And then I could add a serial print here so that we see that something happens to toggle LED. Let's see if our function works. So, it's uploaded.

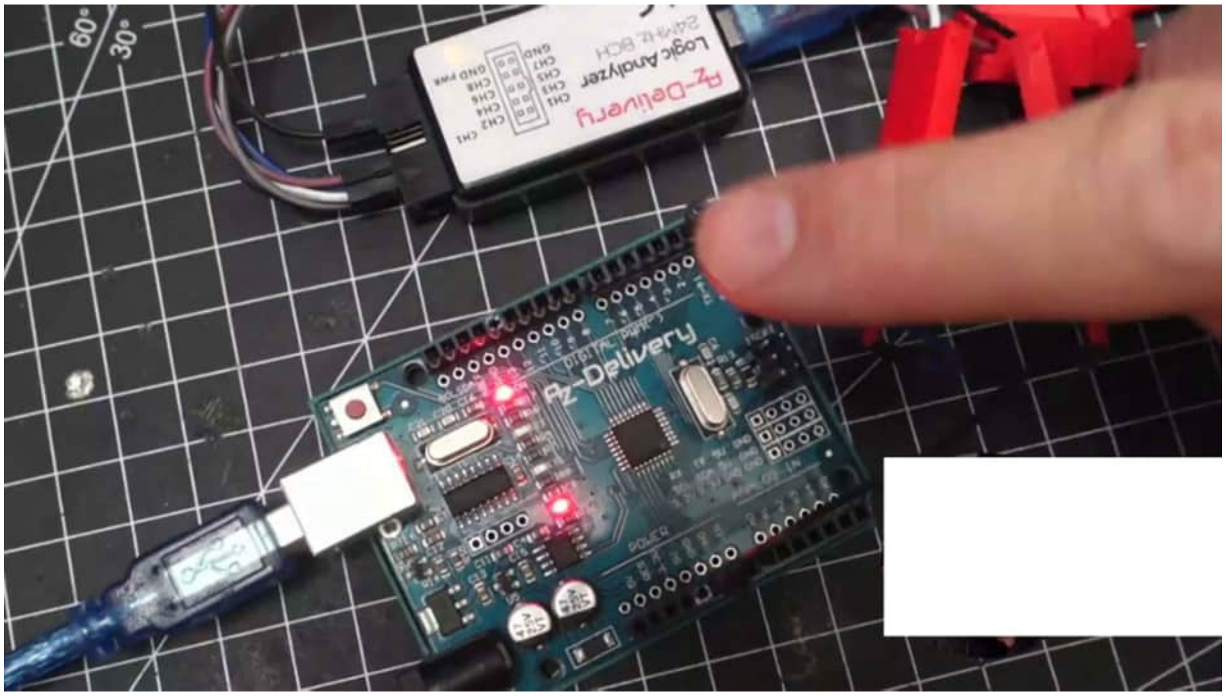
Then one click. And it's turned on. Next click? Turned off. Next click?

Yes. So we are sending now through a logic level shifter from the UNO to the ESP32 via UART and string, which will trigger some LEDs here. And of course, you're not communicating often with 2 MCUs, but you can use this, especially this available when you are receiving data from a module, and this is exactly what we are doing also later on. And therefore, it's a nice use case that you can see what you can do with a logic level shifter with serial available. and also how you can transmit data via your art.

FINDING OR CALCULATING THE BAUD RATE

In this project, I would like to talk with you about the common vault rates. So imagine you have the following setting. You have here an existing circuit, which has some external components and you want to capture the data. But you really don't get any valuable data out of it, what you can read and interpret. So, therefore, we capture the data and then we measure the time frame from one logic level and we can calculate the bot rate.

And this is often used because maybe we can't find any data sheets. So we are trying to reverse engineer it. And the first step is of course, we can make an arrow with some common portraits listed on Wikipedia. We already know the 9600 115200. And also 576 alone are really common in the audio environment.

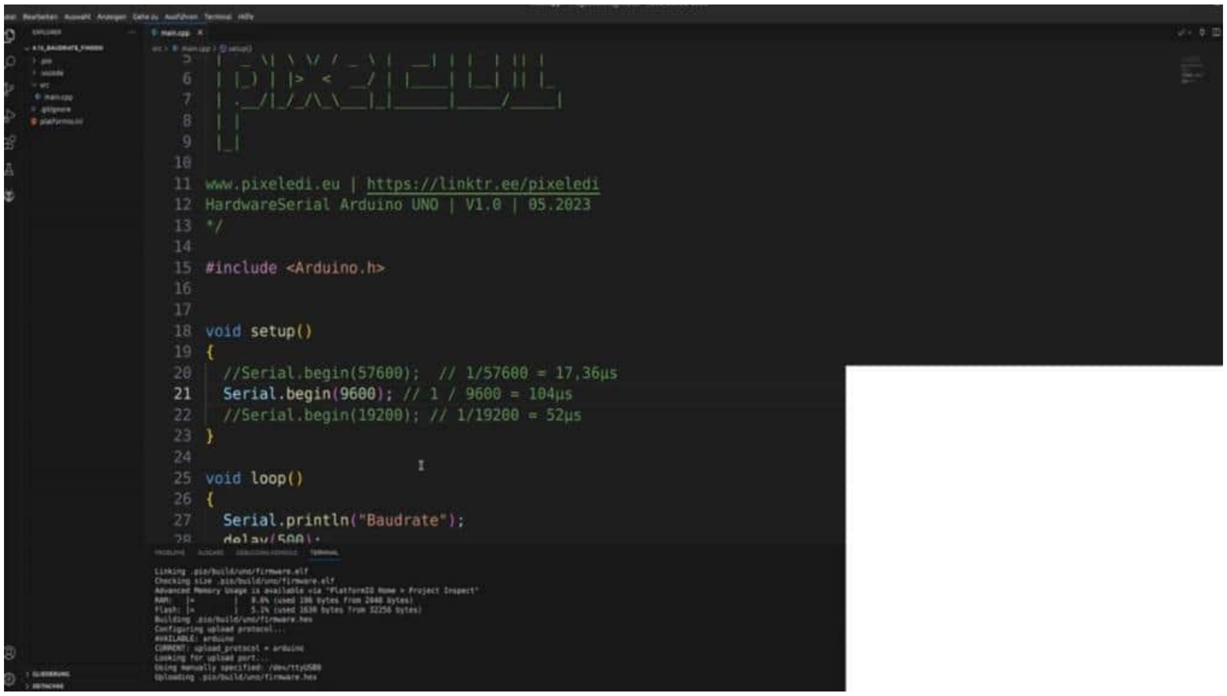


So What we then do is we don't know the portrait, but we want to have it. And the hertz is equal to a portrait. And it's in microseconds when we make the calculation 1 divided by Hertz. So, we have 1 divided to 57,000 is 17.36 microseconds. So let's do this with our sketch.

Therefore, I've already grabbed a connected to hardware serial and the sketch from close to this one from the beginning where we're just sending out here, portraits via serial print. And now I'm starting with 57600. And we should get a pulse here and signal length from the data from roundabout 17 microseconds. So let's check it out. I've already set it up here, the u rts with the serial, 2 gigs, 16 megahertz, and let's run it.

And as you can see, we get here all of the data, it should be synchronized with the. So let's stop it. zoom in on one of the data frames. And I have already entered here 5600. So if we are stuck at 99,600, we are not able to read anything here.

So I'm zooming in and trying to get here on one of the logic levels. So I think this could be one lever. Let's zoom in. And there we go. We have 17 microseconds.



```
www.pixeledi.eu | https://linktr.ee/pixeledi
HardwareSerial Arduino UNO | V1.0 | 05.2023
*/

#include <Arduino.h>


void setup()
{
  //Serial.begin(57600);  // 1/57600 = 17,36µs
  Serial.begin(9600); // 1 / 9600 = 104µs
  //Serial.begin(19200); // 1/19200 = 52µs
}

void loop()
{
  Serial.println("Baudrate");
  delay(500);
```

```
Linking .pio/build/uno/firmware.elf
Checking size .pio/build/uno/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [=         ]   9.8% (used 200 bytes from 2048 bytes)
Flash: [=         ]   5.1% (used 1630 bytes from 32256 bytes)
Building .pio/build/uno/firmware.hex
Configuring upload protocol...
AVAILABLE: arduino
CURRENT: upload_protocol = arduino
Looking for upload port...
Using manually specified: /dev/ttyUSB0
Uploading .pio/build/uno/firmware.hex
```

And then I could change it to 57600. enter. And now you can see, this fits perfectly to my one to one of the signals 11. And when we change it, for example, to 9600, which is commonly used in the arena owner environment. So we have changed it.

Then we click on the run. Tum a little bit out, stop, zoom in to the new data frame. And now we have to change it to 9600. enter. And there we go.

We can read it. But let's measure the new data frame. So I assume this is one data frame, And we get here the length of 104 microseconds, and the calculation shows you also 104 microseconds. So the serial monitor gives out here some undefined imports because I have set up here the monitor speeds to 115200. That's the reason why the monitor can't read it the correct way.

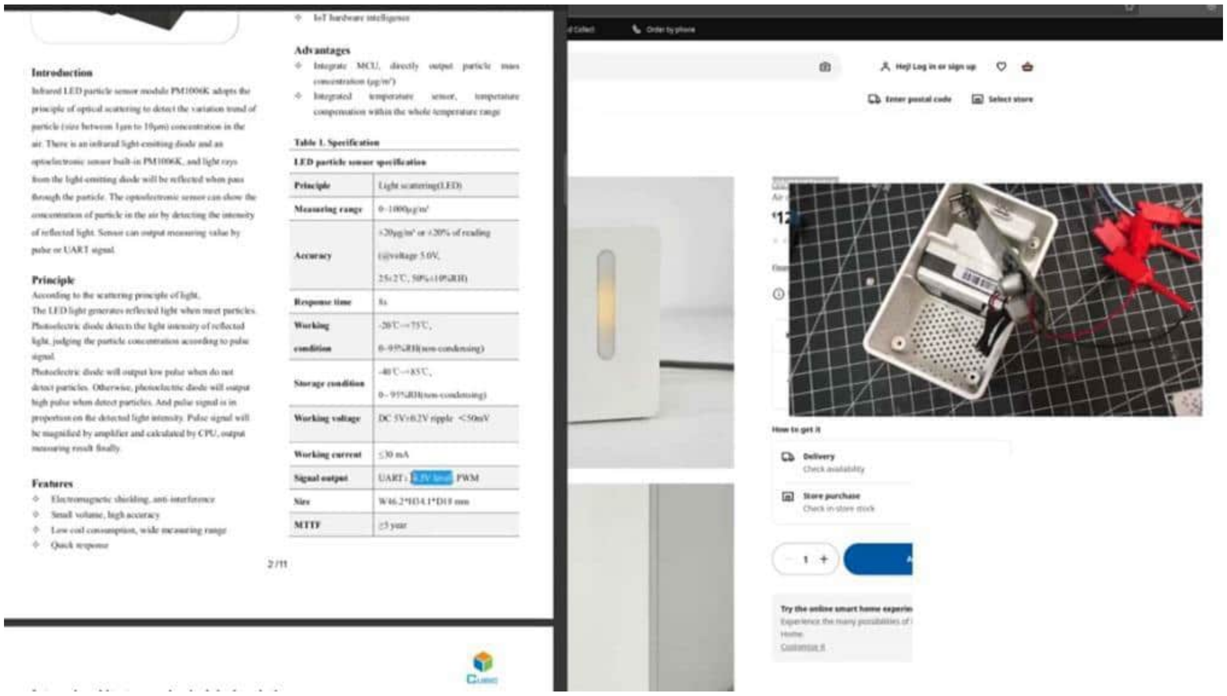
Yeah. And this is a way how we could recalculate or reverse engineer the portrait according to the speed of the logic level.

ANALYZING THE IKEA AIR QUALITY SENSOR VINDRIKTNING

In this project, we're going to make a reverse engineering approach with an example of an air quality sensor. This is the name, which no one can really pronounce correctly, but this air quality sensor or this device has a really good air quality sensor in it. And our approach is now, if we can't capture, here's some data. And I would like to show you my approach with all of the datasheet, analyzing, etcetera. But before you say stop is a finished solution with Tasmotor, etcetera.

Everything is correct. There are already tons of solutions. But not and this is why we are here. We want to train a little bit more, and we want to get the knowledge how could we investigate all of this information, how we can use it, and then also write our own library because this library, which is on GitHub, etcetera, is good, but maybe we can reduce it a little bit on space so that we can make our own IoT device out of So the first point was opening up and take a look on the PCP if we can find here some something which is, commonly known by us. And I found here a reset and also some BCC endpoints And I attached here and sold it to wires on it, so it's easier to capture hidden data and later on also for the connection with the DSP.

Then my next approach was I took the quality sensor out and tried to find it here, and data sheets. And I found here the PM 100 and 6 k data sheets. And when I scroll a little bit down, you can see here the Here it is, a picture of the sensor. And when you take a closer look at me here. So it looks very, very similar.



Introduction

Infrared LED particle sensor module PM1006K adopts the principle of optical scattering to detect the variation trend of particle (size between 1μm to 10μm) concentration in the air. There is an infrared light-emitting diode and an optoelectronic sensor built-in PM1006K, and light rays from the light-emitting diode will be reflected when pass through the particle. The optoelectronic sensor can show the concentration of particle in the air by detecting the intensity of reflected light. Sensor can output measuring value by pulse or UART signal.

Principle

According to the scattering principle of light.
The LED light generates reflected light when meet particles. Photoelectric diode detects the light intensity of reflected light, judging the particle concentration according to pulse signal.
Photoelectric diode will output low pulse when do not detect particles. Otherwise, photoelectric diode will output high pulse when detect particles. And pulse signal is in proportion on the detected light intensity. Pulse signal will be magnified by amplifier and calculated by CPU, output measuring result finally.

Features

- Electromagnetic shielding, anti-interference
- Small volume, high accuracy
- Low coil consumption, wide measuring range
- Quick response

- IoT hardware intelligence

Advantages

- Integrate MCU, directly output particle mass concentration (μg/m³)
- Integrated temperature sensor, temperature compensation within the whole temperature range

Table 1. Specification

LED particle sensor specification	
Principle	Light scattering(LED)
Measuring range	0-1000μg/m³
Accuracy	±20μg/m³ or ±20% of reading (@voltage 5.0V, 25±2℃, 50%±10%RH)
Response time	8s
Working condition	-20℃~+75℃, 0-95%RH(non-condensing)
Storage condition	-40℃~+85℃, 0~95%RH(non-condensing)
Working voltage	DC 5V±0.2V ripple <50mV
Working current	≤30 mA
Signal output	UART (4.5V level) PWM
Size	W46.2*H34.1*D18 mm
MTTF	≥5 year

2/11

And also the stick on it, shows that it's nearly the same generation, not exactly the same we can give it a try. So my approach now was to read all of the stuff in the data sheets. And the first thing that I mentioned here was the UR signal output. 4.5 volt level. That means perfectly usable.

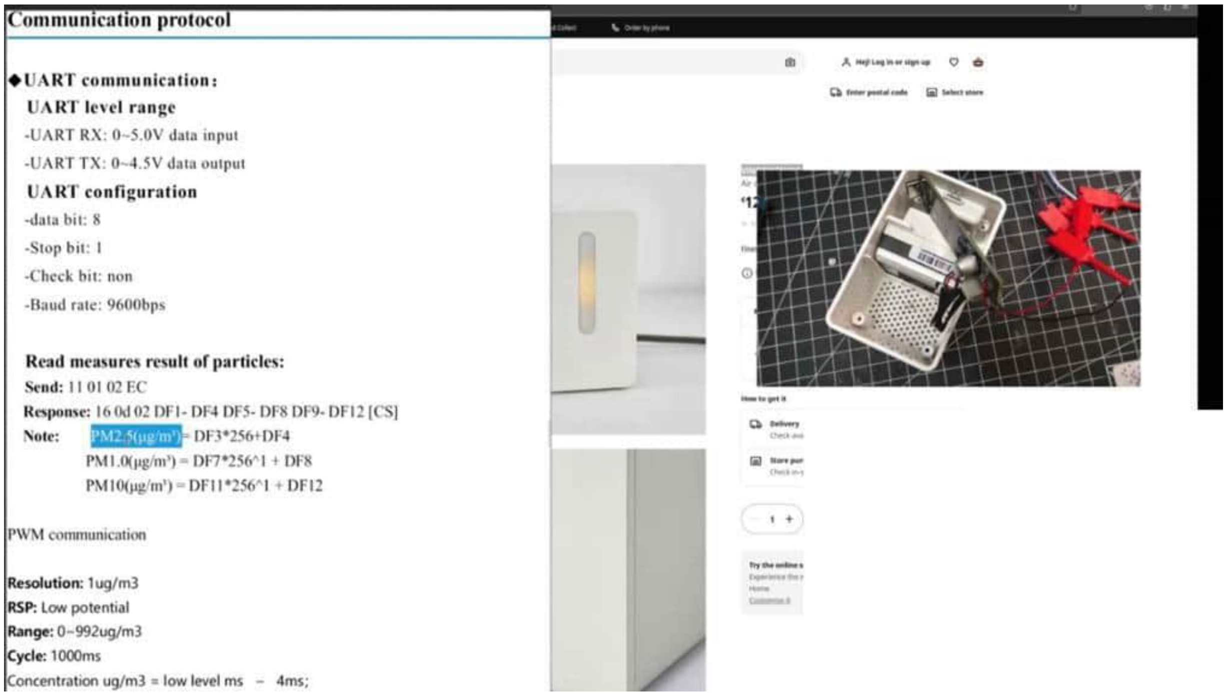
for our Arlene Uno, but maybe not for the ESP year, but I come to this later. So going a little bit down because I want to know in which register the values are the values stored. Here once again, I find out 5 fold ground, RX T X perfectly for the sensor itself. So In this case, I'm not using the sensor directly because I need an MCU, and this is somewhere here, which already communicates with the sensor. So I'm just capturing data and using this finished data.

Because otherwise, I have to write the whole library by myself, and this is not really what I want to do here. So I go a little bit down. Here's the characteristic for the sensor itself. So it gives us some feedback about the air quality inside and room. And the maximum value is here, 100 micrograms per cubic meters and we will test this later on.

Also, here, once again, I have, the T X level is what I want because I'm not sending. Yes. I won't send anything. So this will be the thing.

And here, you would create configuration.

That's it. Data bit We have a stop bit, we have no check bit, and the ball thread is already, set it here. And also here, read measures result of particulars So we are sending those values. So we could connect our ESP directly to the sensor because this is what sends to the sensor, and this should be the response. So we should get here 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16.



Communication protocol

◆**UART communication:**

UART level range

-UART RX: 0~5.0V data input

-UART TX: 0~4.5V data output

UART configuration

-data bit: 8

-Stop bit: 1

-Check bit: non

-Baud rate: 9600bps

Read measures result of particles:

Send: 11 01 02 EC

Response: 16 0d 02 DF1- DF4 DF5- DF8 DF9- DF12 [CS]

Note: PM2.5(μg/m³) = DF3*256+DF4

PM1.0(μg/m³) = DF7*256^1 + DF8

PM10(μg/m³) = DF11*256^1 + DF12

PWM communication

Resolution: 1ug/m3

RSP: Low potential

Range: 0~992ug/m3

Cycle: 1000ms

Concentration ug/m3 = low level ms - 4ms;

So a lot of bytes should we get back? And this is our sensor and pm2.5 sensor. And in the registrar between there, we could then calculate our value, our air quality value. And now the question could be, how could we check if we are right? This is a good question, but I have a second device here.

And this second device is just a few centimeters behind this camera. This is in my office. Here are some more values which were transmitted to Grafana. So for example, I have here humidity, I have here something which indicates my light. pressure and here, are the air quality.

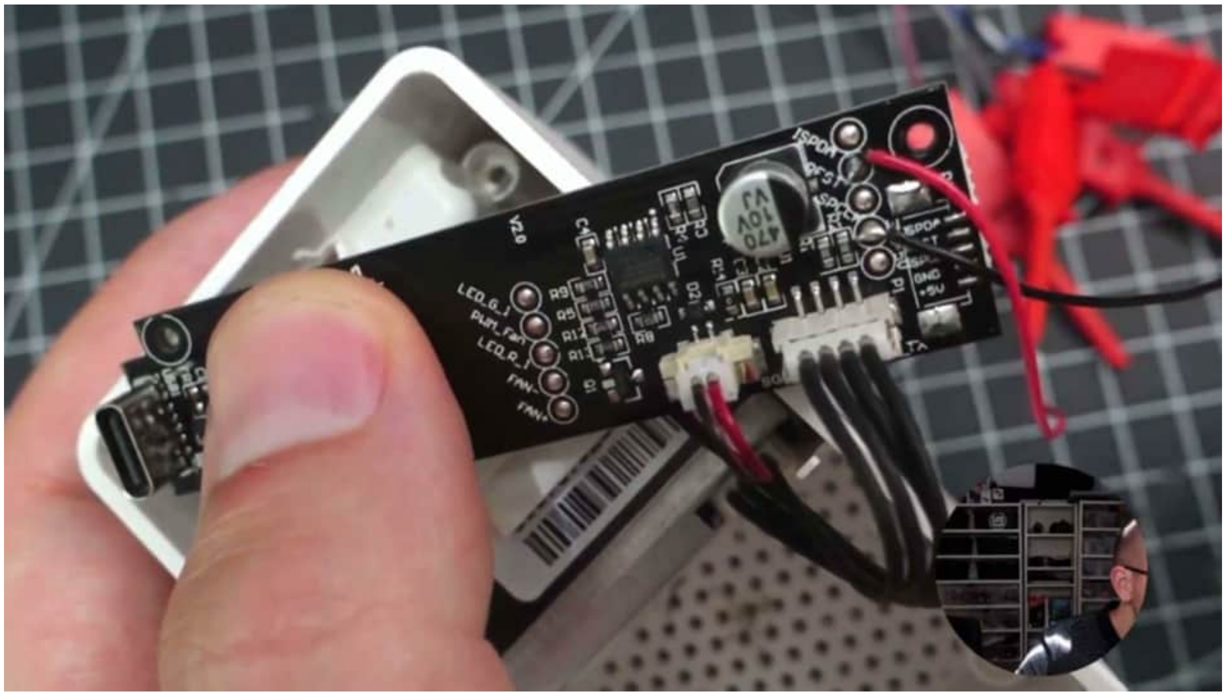
So I have been here in Raleigh from 12, 12 is moderate and could be better. And here's my temperature. I'm melting, but, with all of the light, etcetera, and equipment, the room got here really, really hot.

So here, this quality, this air quality level should be read out. This is our desired state.

no change to 14. So then, I have already attached the clips, clamps to the PCPM. That means I have one ground and one reset. And is it reset? So let's take a close look.

It's one of the reset and one of So here you can see it in a closer look. So the second one, and also the second one from the right and from the left. Those are my 2 desired values. This one is the ground. This one is the data bin.

So with this setting, we could change here to pulse view, click on new. As we did before, we're selecting our device. Okey dokey. I'm changing here to 100 gigs. 60 megahertz should be the channel 0 or channel 1, then an UR decoder, UART decoder, We already know the bot rates.



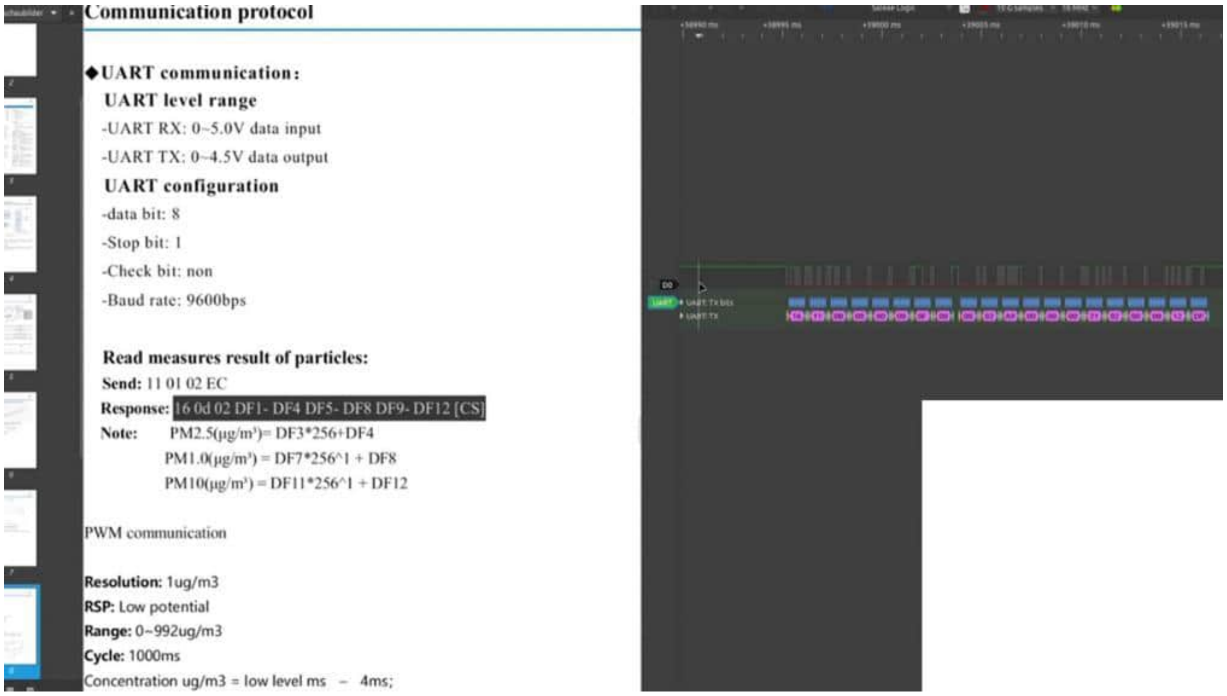
It's 9600. Data bits, no parity bits, stop bits, The order is not really mentioned, so we'll leave it on default. And now it should be the takes because we transceiver. Let's test it out if we get something. But before, we have to also power up the PCB with an USB c cable because the internal MCU should read out all of the stuff here.

So as we can see here, it's starting up. And if it's finished with starting with studying, then I need a 20 seconds time frame, and then we get some data. So I clicked on start measurement, and we can see we're getting here the first data 1, 2, 3, 4, 5, 6. I think it should be 7. And now there should be an 8, 1, 2, 3, 4, getting a lot of data now.

Now it is finished. Now, it should be 20 seconds, for example, no data is transmitted and then it starts all over again with 7 or 6 data frames. This is good to know for the later purpose when we are using the ESP to get the data out of it. So 1, 2, 3, 4, 5, 6, 7, and now the next time frame starts. Okay.

I click and stop. Then turning into one of the data frames, which we got here. And now the interesting part starts because we have here now all of these bytes should be our response. And what we have here is now 1, 2, 3, 4, 5 is TF2. So the 61 should be DF 3 because it's index 0.

So, the 6th one should it be? So 012 4, 5. There is 0 because this is good that this is 0 because otherwise I have some health issues here and 6 because this is our df 4. So we have to calculate df 3 multiplied 256 plus the F 4. So 0 is the first value, and this is our desired second value.


Communication protocol

◆UART communication:

UART level range

-UART RX: 0~5.0V data input

-UART TX: 0~4.5V data output

UART configuration

-data bit: 8

-Stop bit: 1

-Check bit: non

-Baud rate: 9600bps

Read measures result of particles:

Send: 11 01 02 EC

Response: 16 0d 02 DF1- DF4 DF5- DF8 DF9- DF12 [CS]

Note:	PM2.5(μg/m³)= DF3*256+DF4
	PM1.0(μg/m³) = DF7*256^1 + DF8
	PM10(μg/m³) = DF11*256^1 + DF12

PWM communication

Resolution: 1ug/m3

RSP: Low potential

Range: 0~992ug/m3

Cycle: 1000ms

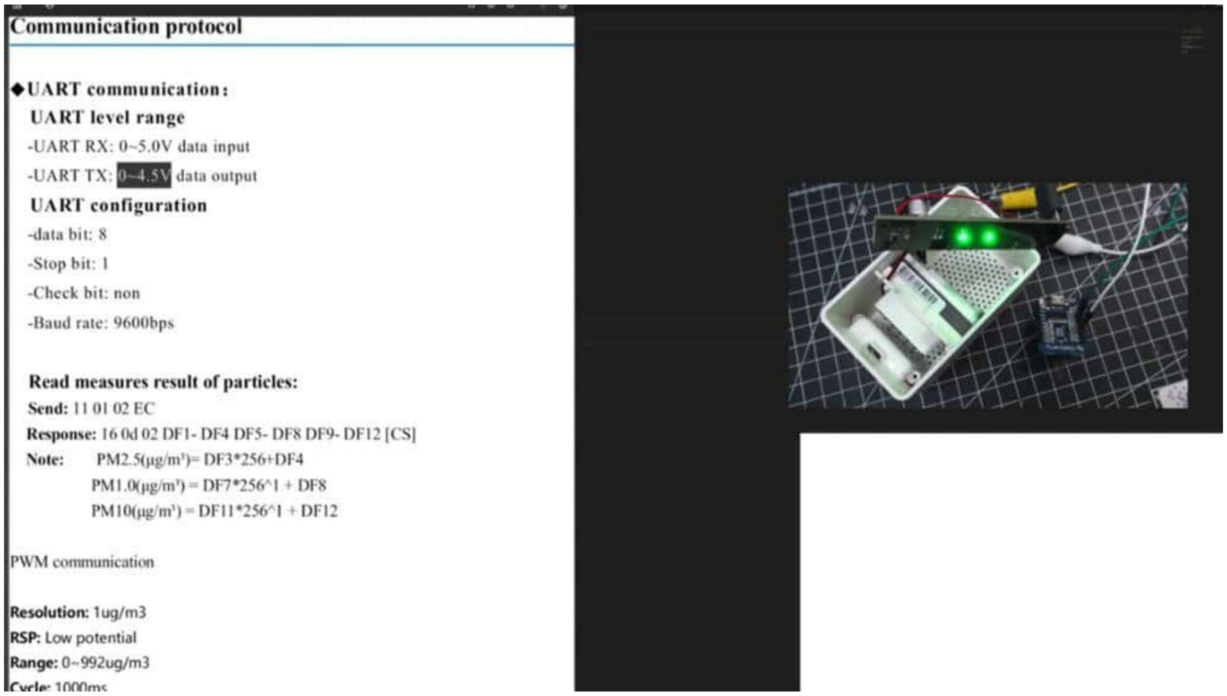
Concentration ug/m3 = low level ms − 4ms;

And it's in hex 4, and this should be 15, I think. if you change it to hex and as you can see, it's 15. And my internal value shows here, 14. As I can see here, 14 is my second device, and we calculated 15. So this gives us, and first clue, how we could approach such a new device and external device because The MCU will now make all of the work by communicating with the sensor.

We are just interfering with it by capturing the data. and reuse this data. But you can also send those commands via you out to the sensor and get out the response. In the next project, we're gonna try to make here our own library for the ESP 8266 to get out the equality values.

SELF-MADE ANALYSIS WITH ESP32 FOR PM2,5 SENSOR FROM IKEA

So then I would say let's do some programming, and I have already attached an ESPN to the board. So here, we have our PCB. And I have attached some jumper cables and some clamps here because it's easier to use. That means I have one cable ground to ground of this tiny twin, 8266, and one goes to an D2, just an input and digital input. We don't use, no more, but with the ability of a wifi, where I can now do this little ESP inside this device.



Communication protocol

◆**UART communication:**

UART level range

-UART RX: 0~5.0V data input

-UART TX: 0~4.5V data output

UART configuration

-data bit: 8

-Stop bit: 1

-Check bit: non

-Baud rate: 9600bps

Read measures result of particles:

Send: 11 01 02 EC

Response: 16 0d 02 DF1- DF4 DF5- DF8 DF9- DF12 [CS]

Note: PM2.5(μg/m³)= DF3*256+DF4

PM1.0(μg/m³) = DF7*256^1 + DF8

PM10(μg/m³) = DF11*256^1 + DF12

PWM communication

Resolution: 1ug/m3

RSP: Low potential

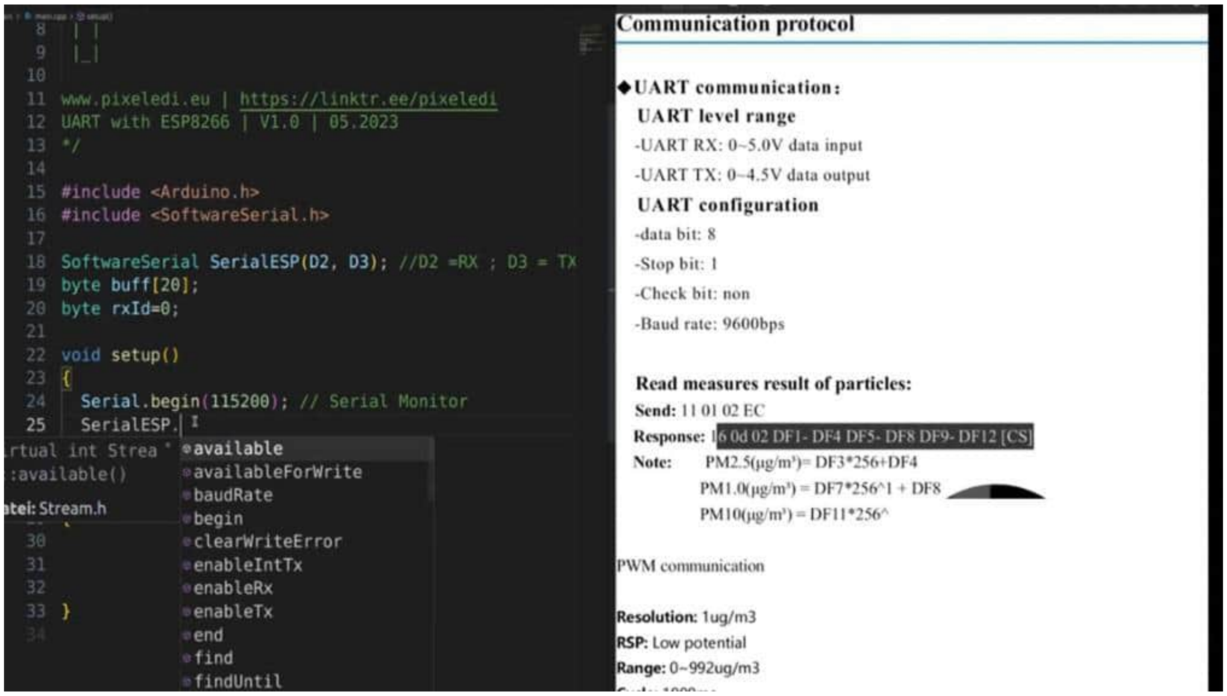
Range: 0~992ug/m3

Cycle: 1000ms

could add maybe some BME 280 sense of humidity and also temperature, and then we have a perfectly IoT device. I also have attached, on the other side, the logic analyzer so that we can also control the values that we are trying to get with this ESP. 8266. And therefore, I've created a fresh new sketch here . And be aware, we are now using the 8266, but you can do the same approach also with the ESP32. And one of the major issues, what we are running into is that we have here a logic level from roundabout 4.5 volts.

And as we did in the previous projects, we know this could damage our ESPN. So what can we do now? First thing is we could use a level shifter. You should use a level shifter. My approach is I'm taking the lazy way and taking the risk, to damage my ESP, but the risk is worth it because we are not really by five volts, a little bit bizarre down.

Maybe this is in the tolerance, and I have my device running for more than a year, and it works really, really well. but it's not so reliable. So if you want to have a really reliable device, use a level shifter because otherwise it really can't, could get damaged. So I am accessing directly, and therefore, we can focus now on some cooling stuff. And I would like to start with some software series.



```
 | |
 |_|

www.pixeledi.eu | https://linktr.ee/pixeledi
UART with ESP8266 | V1.0 | 05.2023
*/

#include <Arduino.h>
#include <SoftwareSerial.h>

SoftwareSerial SerialESP(D2, D3); //D2 =RX ; D3 = TX
byte buff[20];
byte rxId=0;

void setup()
{
  Serial.begin(115200); // Serial Monitor
  SerialESP.

}
```

Communication protocol

◆**UART communication:**

UART level range

-UART RX: 0~5.0V data input

-UART TX: 0~4.5V data output

UART configuration

-data bit: 8

-Stop bit: 1

-Check bit: non

-Baud rate: 9600bps

Read measures result of particles:

Send: 11 01 02 EC

Response: 16 0d 02 DF1- DF4 DF5- DF8 DF9- DF12 [CS]

Note: PM2.5(μg/m³)= DF3*256+DF4

PM1.0(μg/m³) = DF7*256^1 + DF8

PM10(μg/m³) = DF11*256^

PWM communication

Resolution: 1ug/m3

RSP: Low potential

Range: 0~992ug/m3

So we include here, software serial.h Then we are saying software, serial, serial, ESP, And with the ESP 8266, we are declaring it in with the GPIOs. And D2 was my Eriks and B3 is taken and not used, but, keep it there. So what I also need is a buffer array with 20 spaces because we have here So let's change here the there we go. It's better. So here, because we have arm here, 20 bytes what we are getting as response, and this will be saved here in this array.

And I think I need an helper variable, which is an IRX ID index, and we assign it with 0. So let's go. The next one is we have a serial begin for the serial monitor, of course. And then, I would like to have my serial ESP dot begin, and we have here the bar rate from 9600 according to the datasheet. And then we are saying 0 print line, set up finished.

I like it because then I see that everything works, as I expected. Okay. Now we start with the first reading of the content. and you could say if serialesp.available And then we make here a while. And of course, you could only use the while, but I would like to have this approach and as always, many, many ways leads to success.

And then we say buff, Eric's ID, and this is our helper value starting at 0. So 01, 2, 3, 4, 5, etcetera. And I would like to add here my serial ESP sirror.rit. We want the raw bytes and the raw bits And I print out here something so that I can see that there is something happening. So it should print out 20 of these signs, and then I know, okay, there is some data communication right away.

Yeah. And then let's see. We have to do something here, because in the first row, let's open here. So we see we can 1, 2, 3, 4, 5, 6, 7 outputs. We could also take here calculating the mean value from the 7, but just keep it simple in this way.

Now I'm only getting the first value here. So when it comes to the first 20 values, I have to reset my helper value, so that I can get the second cert etcetera as well. So therefore, I say if the Rx ID is greater than 20, then I'm resetting it to 0. I think this should work. And what I would like to do now is print lines.

And we have the buffer array. It should be 6 because 5, nothing is now inside. I just want to check if it's working in the correct way. And the first one, I would like to have here in hex and then in DATIMI. Looking good so far.

Let's see if the compiler goes along with our first approach. Success, then we upload it and it takes some time or when you're used to working with an ESP32, the 8266 is really slow. So two minor things we missed here, we have to increment the value because Now we start with 0 because it has been assigned to 0. Then, the first value will be 0. and it will increase it.

Then 1, 2, 3, 4, 5, 6 until 20. And then we have all of the data safe here in this buff array. And if the Eric's is over 20, then it will print out our values but we should also add this inside the first if because otherwise, we are not getting here value of the data. So, therefore, next try. So let's check if everything works.

Now I have already started viewing parts. And also the serial monitor is on, and then we can check if some value is coming. Yes. 12, 12, let's see here. 1313 So we memorized that this is the actual data frame.

The last one is 19. So let's check if the last one is 19 because the serial monitor will go further on. Here is our hex value, and this is the decimal value because I have forgotten to make an 0 print line. So last well is 19 in this last data frame. Then we're making the UART. So now let's check if something is coming in, it always needs a little bit of time.

So I'm tuning out a little bit. on the data, on the pulse view. So 20 seconds between each measurement, now we're getting the first data. 21, 21, 19, 7, sh, 7 data frames should be received. There we go.

So I click here to stop because we are finished. So let's memorize the last value as 17. This is the hack dismissal. The hex value, this is the decimal value. 17 in the last data frame because this will capture the next data as well.

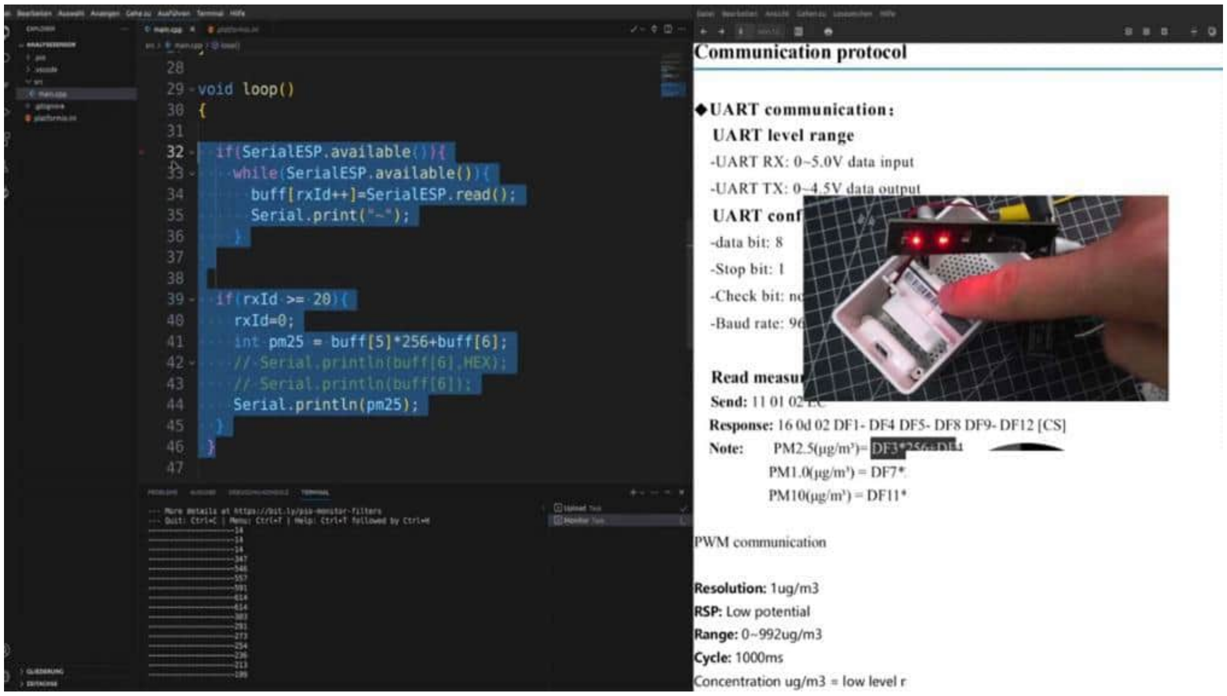
So let's check if we are here on the right path. That means 1, 2, 3, 4, 5, and 6 is the value 111, what is in hex, and 17 in decimal. This is exactly what we want. So it looks promising. That means that our first approach works very well.

So we could, at here, and PM 25 value. And let's take a look in the datasheet. So we have here, DF3. That means Puffer 5, should that be multiplied by 256 plus Buff 6 should be our desired value, what we want to have. And then we could print out here the value on a 0 print line, PM25.

And of course, when we know that there are 7 circles inside in a measurement, we could calculate here the meanwhile you're but for our case, this will be perfect. And now we could use this PM value and send via M quantity to influxdb, for example, and also visualize it on Kafana. But this is not covered in this Project. You can check out my note RED Project, for example, where I'm dealing with all of these IoT devices and aspects as well. So let's see if this also works, then, upload the code.

And in the meantime, I will have quality tests here so I'm burning here a little bit of paper so that I can use the air quality so that we can change here or see a little bit of change inside the values. So, therefore, I am adding a little bit of smoke here. Maybe this is already enough. So this was the 1st cycle, then let's see that should make a significant change normally in the air quality. There we go.

Nice. We know 1000 is the maximum we can receive. And now I have 5 611. And we also can see here that the LED is changed. But when you search a little bit about the BM 2.5, I think it should be changed at the level 30.



```
void loop()
{

  if(SerialESP.available()){
    while(SerialESP.available()){
      buff[rxId++]=SerialESP.read();
      Serial.print("~");
    }


  if(rxId >= 20){
    rxId=0;
    int pm25 = buff[5]*256+buff[6];
    // Serial.println(buff[6],HEX);
    // Serial.println(buff[6]);
    Serial.println(pm25);
  }
}
```

```
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
~~~~~~~~~~~~~~~~~~~~14
~~~~~~~~~~~~~~~~~~~~14
~~~~~~~~~~~~~~~~~~~~14
~~~~~~~~~~~~~~~~~~~~347
~~~~~~~~~~~~~~~~~~~~546
~~~~~~~~~~~~~~~~~~~~557
~~~~~~~~~~~~~~~~~~~~591
~~~~~~~~~~~~~~~~~~~~614
~~~~~~~~~~~~~~~~~~~~614
~~~~~~~~~~~~~~~~~~~~383
~~~~~~~~~~~~~~~~~~~~291
~~~~~~~~~~~~~~~~~~~~273
~~~~~~~~~~~~~~~~~~~~254
~~~~~~~~~~~~~~~~~~~~236
~~~~~~~~~~~~~~~~~~~~213
~~~~~~~~~~~~~~~~~~~~199
```

Communication protocol

◆**UART communication:**

UART level range

-UART RX: 0~5.0V data input

-UART TX: 0~4.5V data output

UART conf

-data bit: 8

-Stop bit: 1

-Check bit: no

-Baud rate: 96

Read measu

Send: 11 01 02

Response: 16 0d 02 DF1- DF4 DF5- DF8 DF9- DF12 [CS]

Note: PM2.5(μg/m³)= DF3*256+DF4

PM1.0(μg/m³) = DF7*

PM10(μg/m³) = DF11*

PWM communication

Resolution: 1ug/m3

RSP: Low potential

Range: 0~992ug/m3

Cycle: 1000ms

Concentration ug/m3 = low level r

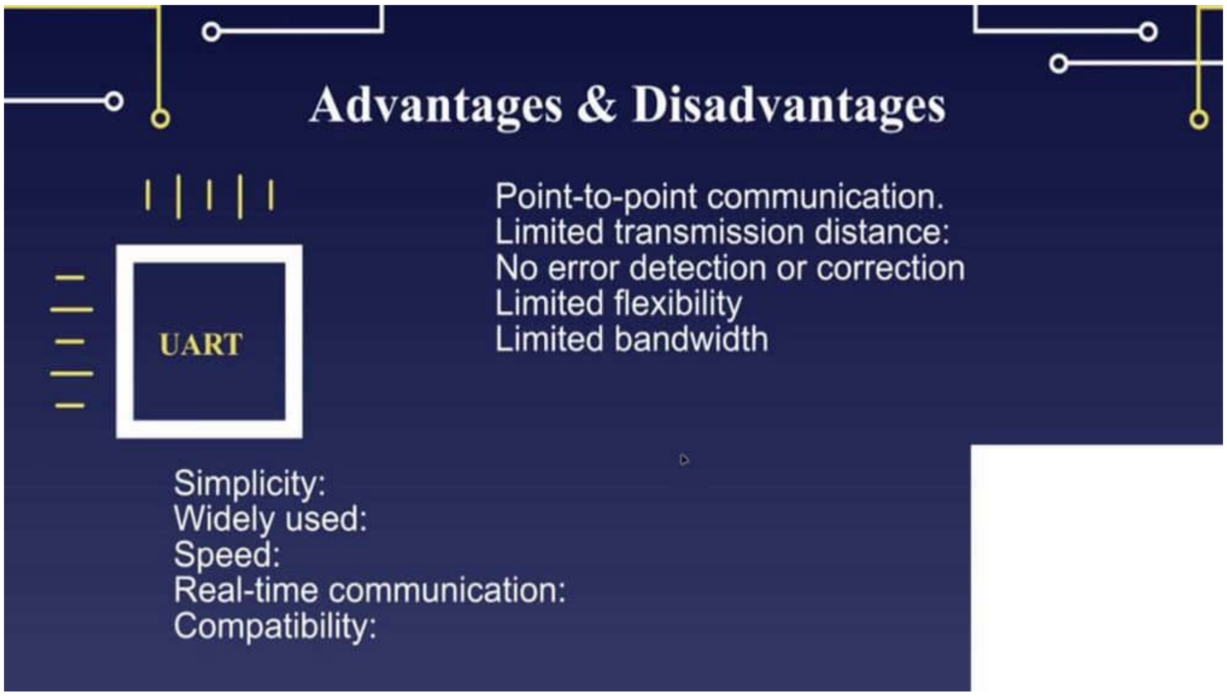
This is the first indication that you should, open the windows, etcetera. So first test, I would say absolutely amazing. And with just a few lines of codes, we were able to capture the data from this, PCP. Otherwise, we can get rid of the PCP, send the data directly to the sensor and this is even more of an approach to what you can do. But I would like to have this signal and start to slide the Ikea from itself and, therefore, keep it and also attach a temperature sensor.

And now we have successfully reversed engineered here on the UI's communication between the sensor and PCPM and have made our own IoT device with a little ESP 8266.

ADVANTAGES AND DISADVANTAGES OF UART

We are coming to the end of our first protocol chapter. And I would like to take the chance to discuss with you a little bit more the advantages and disadvantages. So one of the first points is that we only can establish your point to point communication. In the other protocols, we have a bus system where we can add many other modules to GPS I also, for example, but in this case, we only can use 1 module with 1 Rx and Tx level. Of course, we have more uarts, like hardware uarts in the ESP32, or we can use software serial but this is one limitation we have.

We also have some limited transmission distance, which means from, the cable length, etcetera. So otherwise, we will get a lot of arrows in the communication. We don't really have error detection. or correction. It's just an indication with this parity bit.



Advantages & Disadvantages

Point-to-point communication.
Limited transmission distance:
No error detection or correction
Limited flexibility
Limited bandwidth

Simplicity:
Widely used:
Speed:
Real-time communication:
Compatibility:

We have to do our own error detection, like checksums, etcetera, to be very sure that the data is transmitted correctly. Limited flexibility and also limited bandwidth, but nevertheless, it's one of the, yeah, often used in our Reno environment because it's easy to use, and nearly every microcontroller has it. The simplicity and widely used speed, of course, real time communication is also, as we did before with this air quality sensor, and the compatibility of STM, AT tiny, RT Mega, ASPs, what do you call it? Nearly every microcontroller has an interface. and, therefore, it's widely used and also very commonly used.

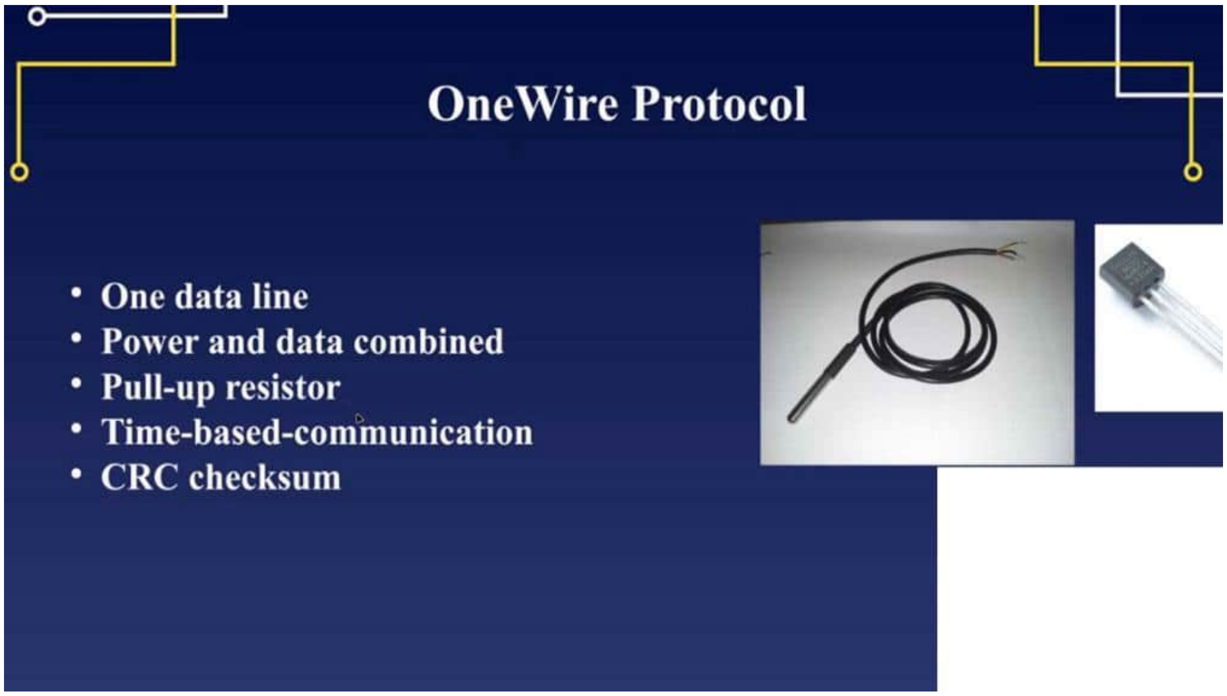
And as you can see, easy, easy to learn the principles, the main principles, how we can send data from and to, MCU or turn material.

ONEWIRE PROTOCOL BASICS

Let us start in this chapter by discussing what is the 1 wire protocol. On the right side, we see here 2 different housings from the same sensor. And when it comes to 1 wire, the DS ATB 20 is one of the mainly used sensors. And on the right side, we see the typical housing, what we are using for our RD new projects on, on the left side, we see the same sensor but with another housing so that you can use it also with some liquids. And the reason why this is so popular is because it's easy to get.

It's very cheap, and we only need one data line. There is not really, library needed. Of course, we use the 1 wire library, but all of the other aspects we can do very easily on our own. And the next real advantage is that we can combine the VCC line and the data line So we only need 2 GPS that means ground and one line for power and data communication. And this is really cool.

You can go even further. You can use, many of the dealers yeah, 18b20 simultaneously on the same GPU. How does that work? Each of this temperature sensor has this unique address And later on in the sketch, we can address the sensor bytes unique address and then can fetch your temperature data. Yeah.



OneWire Protocol

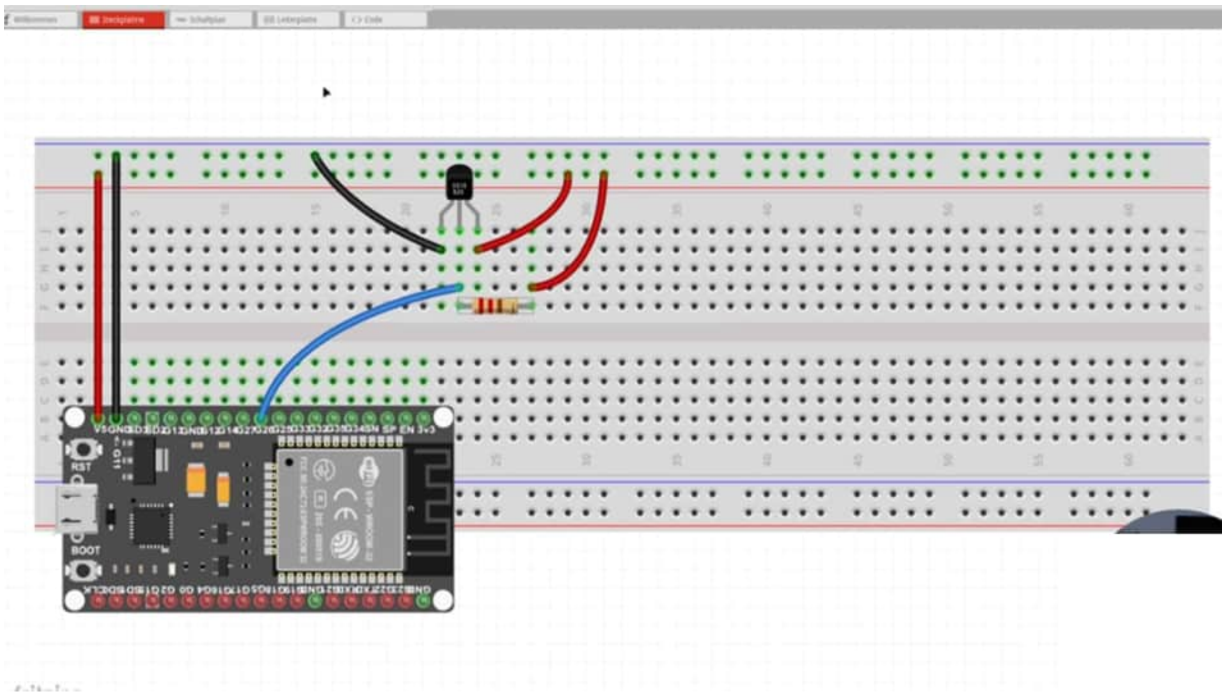
- One data line
- Power and data combined
- Pull-up resistor
- Time-based-communication
- CRC checksum

We need a pull up resistor. That's how this whole communication works, and it's like, time based communication. And as the last point, we also get here in CRC, check some that we can, and have a reliable temperature setting in our MCO. So in summary, this is really cool or a cool protocol. It's a single bidirectional data line for communication, to make it simple.

too and caustic cost effective to use. This is the reason why it's really popular, and I would like to cover these one wire protocols as well because it's good training for us on how we can access data on a bit level.

DS18B20 SKETCH AND WIRING

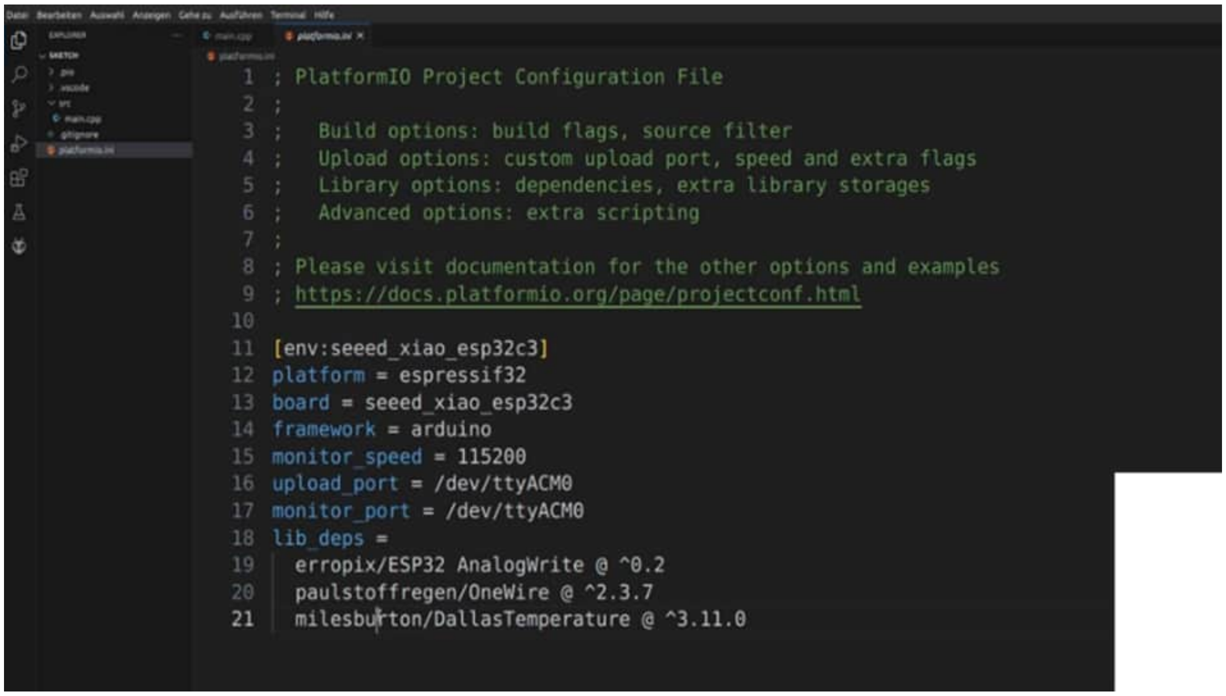
One of our first steps will be to create a simple sketch so that we can communicate from MCU to the module. And then we start with the wiring part. So first of all, we are connecting W5 and the ground to our breadboards. can also use 3.3 volts. So on the left pin of our DS 18b20 sensor, we're connecting ground.



The middle pin, we connected to some of your GPIOs. You can also use another Uni or an ESP 866 if you would like. Then we have a 4.7 kilo ohm resistor, and this is our pull up resistor. So we need to put in a slap. And therefore, afterwards, before we are connecting to the GPO, afterwards, we are connecting the line to the VCC part so that we have a resistor and pull up logic.

And on the right pin, we have here our BCC. So this is in a short summary, our wiring part. I have implemented it now with, tiny ESP32. This is one from the studio, but you can do it as well with an d 1, with a dev board. What do you have in your equipment?

So the clamps from the logic analyzer are connected directly to the center. And that's it. That's all that we need to do here. So let's switch back to Visual Studio Code And I've prepared here my basic sketch, nothing special in it. And let's take a closer look at that platform for him.



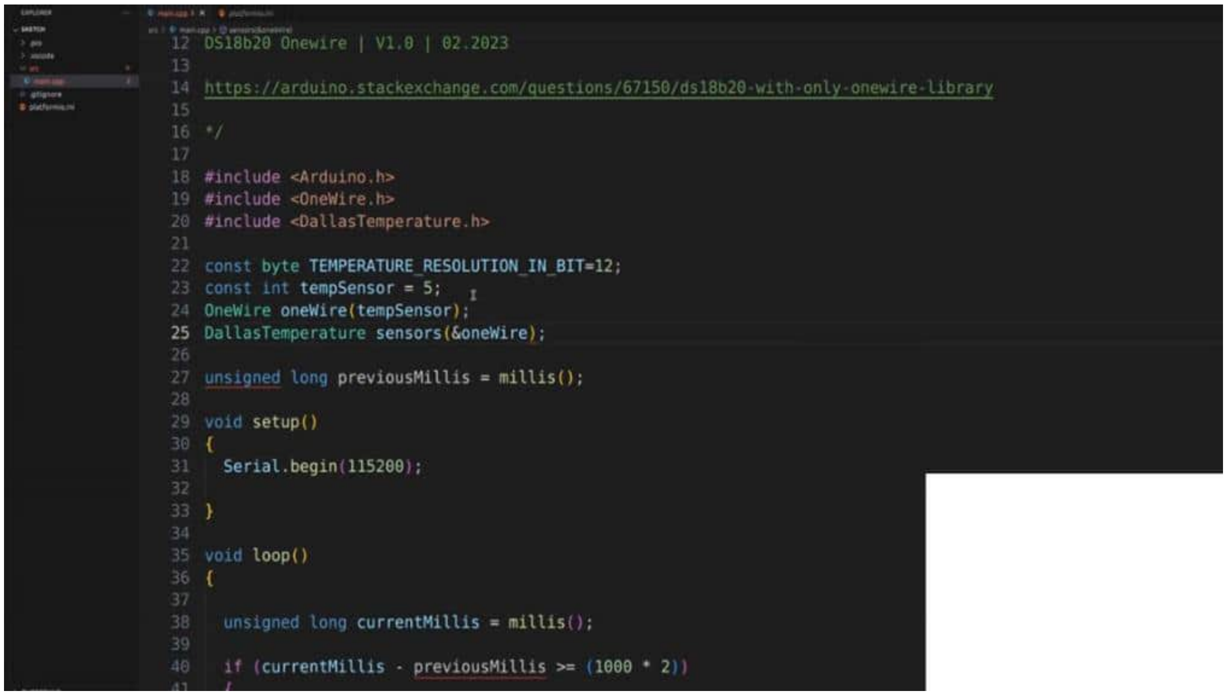
```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:seeed_xiao_esp32c3]
platform = espressif32
board = seeed_xiao_esp32c3
framework = arduino
monitor_speed = 115200
upload_port = /dev/ttyACM0
monitor_port = /dev/ttyACM0
lib_deps =
  erropix/ESP32 AnalogWrite @ ^0.2
  paulstoffregen/OneWire @ ^2.3.7
  milesburton/DallasTemperature @ ^3.11.0
```

Here is my information from the board. But what's really interesting is that we need external libraries. So the first one is the 1 wire, and then we need the delist temperature. And both libraries can also be installed in the Ardeno ID. Just search for 1 wire in Dallas temperature.

These are the names who are created, the libraries, then installed, and then we are ready to go. So first thing, we have to include those libraries, which we have just downloaded. And therefore, we're typing in include, and it's 1y.h. And this library will mainly be used mainly to use the 1 wire protocol. And the second one is primarily used for the Dallas 18b20 sensor on.h.

Very good. Then we need some special points here. So I would like to set it up here in constant byte temperature resolution. because later on, we will use this and we will change it. So at default value, we have here, resolution in twelve bits, and we come to this in the next project.

What does that mean? So then we have a constant temp sensor, and I have used the gpio 5 on a tiny ESP32. So just, take a look at your pinout from the ESP port and then type in here, the GPIO. Then we need 1 wire object, 1 wire, 1 wire, and we are referring here to the sensor. This was the first step.



```
DS18b20 Onewire | V1.0 | 02.2023

https://arduino.stackexchange.com/questions/67150/ds18b20-with-only-onewire-library

*/

#include <Arduino.h>
#include <OneWire.h>
#include <DallasTemperature.h>

const byte TEMPERATURE_RESOLUTION_IN_BIT=12;
const int tempSensor = 5;
OneWire oneWire(tempSensor);
DallasTemperature sensors(&oneWire);

unsigned long previousMillis = millis();

void setup()
{
  Serial.begin(115200);

}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000 * 2))
```

And then we need a Dallas temperature object. I call it a sensor. And now we make a reference to the 1 wire object. So the 1 wire object will be passed to the constructor of the delast temperature object. Okay.

So far so good. Just some basics. In the setup part, we are starting the sensor now. So this is our desired object where we can get all of our data and we say sensors begin, sensors. No.

I was a sensor. Let's call it sensors because maybe you would like to add here a few more of them and the same GPIO, then you can do that with this one. Okay. Then, let's jump into the loop part and we want to fetch the data each 2 seconds because it's also connected with the resolution. I think you can do one second as well.

So then let's see sense or set resolution. is one point. So we do not set it each time, but we can BTR as well. So then the temperature resolution is a bit. Then we say sensors dot request, I think, should be a request temperature.

So this is just a function call. And so now we are creating some flaws here. For example, temperature in Celsius because since the Celsius is the default value, And then we say sensors get temperature by index. and this is, how we get here the temperature value. The same thing could be done for the recalculation for Fahrenheit, and then it's not get temperature c.

It's called Let's see. Get temperature f by index 0. Okeydoke. That looks good so far. Then we can print out the values.

So I have already prepared the serial print here. And now, it's time for the first check. Let's see if you have some typos, no, then upload the code. And there we go. We have the values here.

And as you can see here, it refreshes every 2 seconds. And I am here in 29 degrees. It's mid June 2023. And now I would like to have a little bit of compression because I'm really melting here. My temperature sensor behind me says it's around 20, 32 degrees.

This is because of the light, many monitors, the PC, etcetera, and, it's not the perfect condition in the summertime to make some projects here. But back to the 1 wire protocol, as we can see, the first step we have already accomplished so we can get here the temperature in scratch, degrees Celsius and also in Fahrenheit.

EXAMPLE DUMMY CODE

Wiring:

Connect the DS18B20 sensor to your Arduino as follows:

DS18B20 VCC pin to Arduino 5V.
DS18B20 GND pin to Arduino GND.
DS18B20 Data pin (DQ or DATA) to an Arduino digital pin (e.g., 2).
Make sure to add a pull-up resistor (4.7k ohm) between the DS18B20's data pin and the 5V line.

Arduino Sketch:

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

// Data wire is connected to Arduino digital pin 2
#define ONE_WIRE_BUS 2

// Create OneWire instance to communicate with DS18B20
OneWire oneWire(ONE_WIRE_BUS);

// Pass oneWire reference to DallasTemperature library
DallasTemperature sensors(&oneWire);

void setup() {
  // Start serial communication
  Serial.begin(9600);

  // Initialize the DS18B20 sensor
  sensors.begin();
}
```

```
void loop() {
  // Call sensors.requestTemperatures() to issue a global
temperature request to all devices on the bus
  sensors.requestTemperatures();

  // Get temperature from the DS18B20 sensor
  float temperatureC = sensors.getTempCByIndex(0); // 0
refers to the first DS18B20 on the bus

  // Check if a valid temperature was obtained
  if (temperatureC != DEVICE_DISCONNECTED_C) {
    // Print temperature to the Serial Monitor
    Serial.print("Temperature: ");
    Serial.print(temperatureC);
    Serial.println(" °C");

    // You can also convert to Fahrenheit if needed
    // float temperatureF =
sensors.toFahrenheit(temperatureC);
    // Serial.print("Temperature: ");
    // Serial.print(temperatureF);
    // Serial.println(" °F");
  } else {
    Serial.println("Error: Could not read temperature
data");
  }

  // Wait a moment before taking the next reading
  delay(1000);
}
```

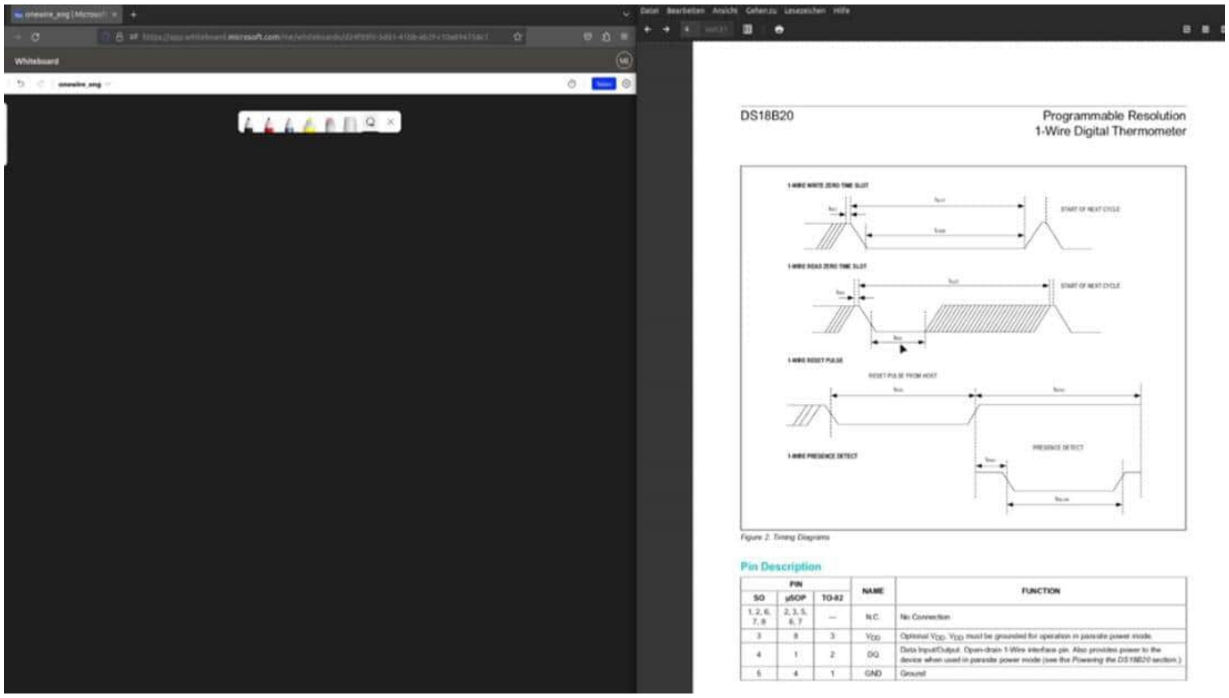
In this code, we use the DallasTemperature and OneWire libraries to interface with the DS18B20 sensor. The temperature is read in Celsius, and you can convert it to Fahrenheit if needed.

Upload this code to your Arduino, open the Serial Monitor, and you should see temperature readings from the DS18B20 sensor. Make sure the DS18B20 sensor is correctly wired to your Arduino before running this code.

INFO FROM THE DATA SHEET

Before we make the analysis in PulseView, I would like to go through the datasheet, so the first few pages, and the DS18B20 one-wire digital thermometer. As you can see here the pin configuration, because we have other housings here as well, so we did a good job in our wiring part so far, very good. So let's see what's interesting for us. The measured temperature, minus 55°C to plus 125°C, is really cool. And we can program a resolution, this is what we focus on now, only two pins, everything what we talked about before.

So let's go a little bit down. Here is also the supply voltage, minimum plus 3V, maximum 5.5V, so we are in the range from 3.3V to 5V, so ESP32 and Arduino UNO logic level would work here. Then a little bit down. Here we can see the temperature conversion time, that means in 9-bit resolution we can fetch data in 100ms frame, but in 12-bit resolution, which is in the default setting, we have to wait 750 ms for each circle. Then here we can see a little bit more in detail how the communication process is set up.



DS18B20

Programmable Resolution
1-Wire Digital Thermometer

Figure 2. Timing Diagrams

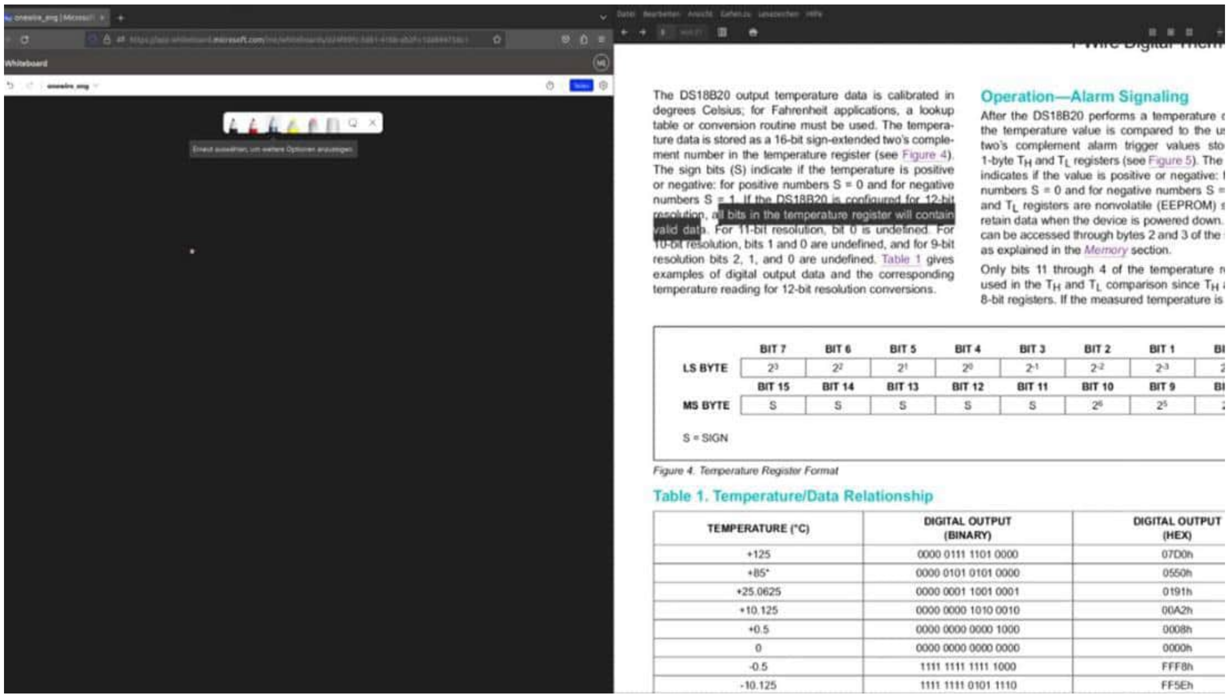
Pin Description

PIN			NAME	FUNCTION
SO	µSOP	TO-92		
1, 2, 6, 7, 8	2, 3, 5, 6, 7	—	N.C.	No Connection
3	8	3	V_{DD}	Optional V_{DD}. V_{DD} must be grounded for operation in parasite power mode.
4	1	2	DQ	Data Input/Output. Open-drain 1-Wire interface pin. Also provides power to the device when used in parasite power mode (see the *Powering the DS18B20* section.)
5	4	1	GND	Ground

will be investigated later when we are starting PulseView, because I would like to scroll a little bit more down, this is what I would like to do now, because a little bit of the basics, how we can recalculate the data in temperature, because this is what we need later on.

So let's focus here, the output temperature is calibrated in degree Celsius, for Fahrenheit application look up, ok we have to recalculate it, is stored in a 16-bit signed extended two complement numbers, ok, the sign bit indicates if the temperature is positive or negative, for positive numbers, S is 0, negative 1. If the sensor is configured for 12-bit resolution, all bits in the temperature contain valid data.

Ok, so now we have here some examples, which I would like to focus on, but before we do that, let's see what does it mean to have 9-bit or 12-bit resolution. So if we have here 9-bit and here we have 12-bit, then let's see, we have here by 9-bit we could say 2 to the power of 9 and then we have 2 to the power of 12, so when we calculate this, this should be 512 and this should be 4096.


The DS18B20 output temperature data is calibrated in degrees Celsius; for Fahrenheit applications, a lookup table or conversion routine must be used. The temperature data is stored as a 16-bit sign-extended two's complement number in the temperature register (see Figure 4). The sign bits (S) indicate if the temperature is positive or negative: for positive numbers S = 0 and for negative numbers S = 1. If the DS18B20 is configured for 12-bit resolution, all bits in the temperature register will contain valid data. For 11-bit resolution, bit 0 is undefined. For 10-bit resolution, bits 1 and 0 are undefined, and for 9-bit resolution bits 2, 1, and 0 are undefined. Table 1 gives examples of digital output data and the corresponding temperature reading for 12-bit resolution conversions.

Operation—Alarm Signaling

After the DS18B20 performs a temperature
the temperature value is compared to the u
two's complement alarm trigger values sto
1-byte T_H and T_L registers (see Figure 5). The
indicates if the value is positive or negative:
numbers S = 0 and for negative numbers S =
and T_L registers are nonvolatile (EEPROM)
retain data when the device is powered down.
can be accessed through bytes 2 and 3 of the
as explained in the *Memory* section.

Only bits 11 through 4 of the temperature
used in the T_H and T_L comparison since T_H
8-bit registers. If the measured temperature is

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1
LS BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9
MS BYTE	S	S	S	S	S	2^6	2^5

S = SIGN

Figure 4. Temperature Register Format

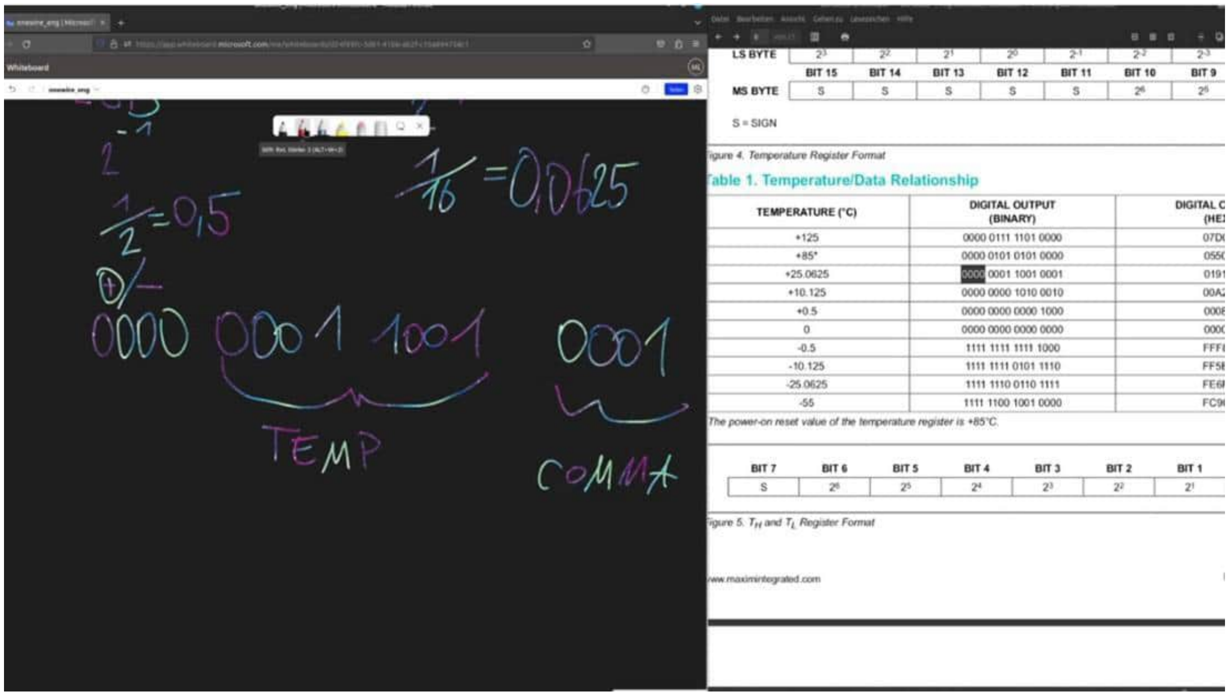
Table 1. Temperature/Data Relationship

TEMPERATURE (°C)	DIGITAL OUTPUT (BINARY)	DIGITAL OUTPUT (HEX)
+125	0000 0111 1101 0000	07D0h
+85*	0000 0101 0101 0000	0550h
+25.0625	0000 0001 1001 0001	0191h
+10.125	0000 0000 1010 0010	00A2h
+0.5	0000 0000 0000 1000	0008h
0	0000 0000 0000 0000	0000h
-0.5	1111 1111 1111 1000	FFF8h
-10.125	1111 1111 0101 1110	FF5Eh

So nothing special so far, but when we take here once again a closer look on the calculation, that means we have to see here that this least significant bit first and when we are calculating here, so that means 2 to the power of minus 1 and this is 1 divided by 2 is 0.5 in the granulation and here we have much more behind the comma, so here we have 2 to the power of minus 4 and this means 1 divided by 16 and this will lead to a resolution, let's check the calculator, 1 divided by 16 is 0.0625.

So in this detail we can now fetch the data when we are setting the resolution to 12-bits and we just get steps from half a degree Celsius when we set it to 9-bits. So far so good, this is what the resolution is and how we can interpret it, but let's focus now on the calculation of those examples. So later in the next project we are getting some data from the register, that means this is the poor bit that we want to interpret. So let's do this one.

So that means if we have that data, let's type it in once again. So we have here four times and zero, and that is indicating the sign. That means if we have a plus or minus, and as we can see before, it indicates the sign bit, positive means S is zero. So we have a positive value here. And then we have the temperature value. So we have here 0, 0, 0, 1, 1, 0, 0, 1, and we have here 0, 0, 0, 1. So this is the temperature value. And the last four bits are the comma. And now we can calculate the values here.



LS BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9
MS BYTE	S	S	S	S	S	2^6	2^5

S = SIGN

igure 4. Temperature Register Format

able 1. Temperature/Data Relationship

TEMPERATURE (°C)	DIGITAL OUTPUT (BINARY)	DIGITAL C (HE
+125	0000 0111 1101 0000	07D
+85*	0000 0101 0101 0000	055
+25.0625	0000 0001 1001 0001	0191
+10.125	0000 0000 1010 0010	00A
+0.5	0000 0000 0000 1000	000
0	0000 0000 0000 0000	000
-0.5	1111 1111 1111 1000	FFF
-10.125	1111 1111 0101 1110	FF5
-25.0625	1111 1110 0110 1111	FE6
-55	1111 1100 1001 0000	FC9

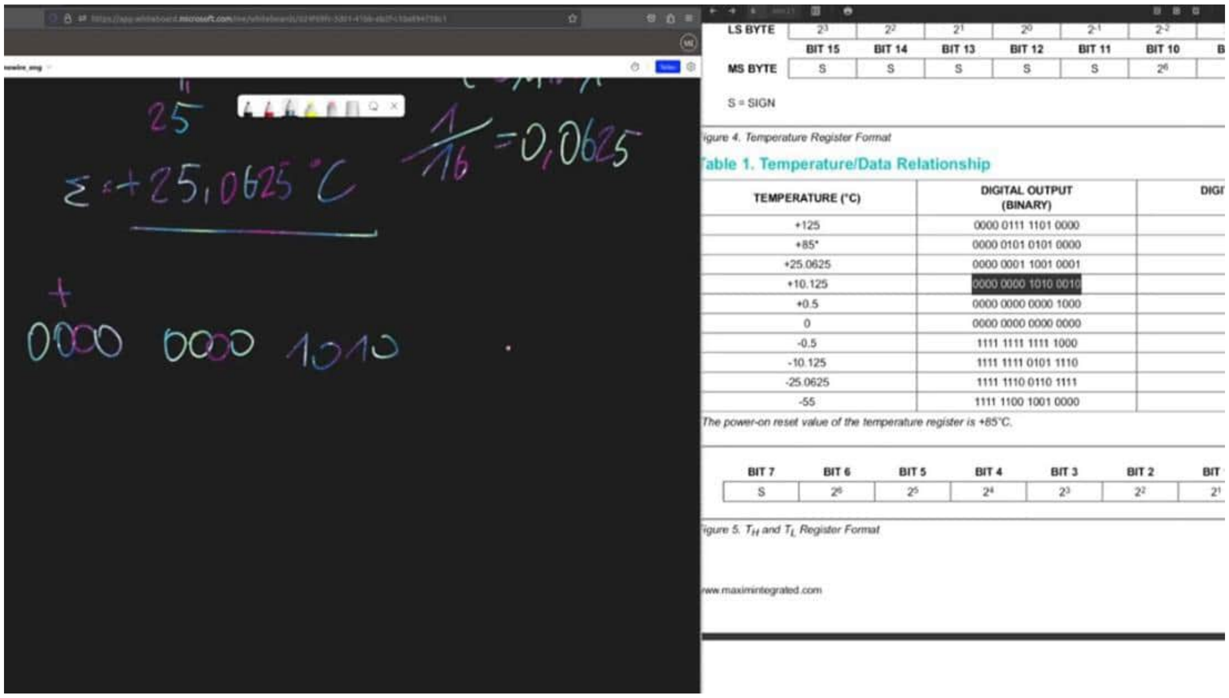
The power-on reset value of the temperature register is +85°C.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1
S	2^6	2^5	2^4	2^3	2^2	2^1

igure 5. T_H and T_L Register Format

ww.maximintegrated.com

Speaker 1 (00:05:58) - So we have 1, 2, 4, 8, 16 here, and this doesn't matter because it's zero. And here we have just the one. So how can we calculate it now? So that means if we add here 16 and 8 are 24, 25. So that means this is 25. And to the comma, we have to use the value of the comma. And we have a 12-bit resolution, and this will be as before, 0, 0, 6, 2, 5. If we get another value here, then we set it here, for example, five divided by 16. So in summary, we have here the value of plus 25.0625 degrees Celsius.



LS BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10
MS BYTE	S	S	S	S	S	2^6

S = SIGN

igure 4. Temperature Register Format

able 1. Temperature/Data Relationship

TEMPERATURE (°C)	DIGITAL OUTPUT (BINARY)	DIGI
+125	0000 0111 1101 0000	
+85*	0000 0101 0101 0000	
+25.0625	0000 0001 1001 0001	
+10.125	0000 0000 1010 0010	
+0.5	0000 0000 0000 1000	
0	0000 0000 0000 0000	
-0.5	1111 1111 1111 1000	
-10.125	1111 1111 0101 1110	
-25.0625	1111 1110 0110 1111	
-55	1111 1100 1001 0000	

The power-on reset value of the temperature register is +85°C.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT
S	2^6	2^5	2^4	2^3	2^2	2^1

igure 5. T_H and T_L Register Format

ww.maximintegrated.com

Let's do another example so that we can really be sure that we have understood this kind of values, what we are getting. So let's focus on the 10-degree example here. So we have four times the zero. This means plus. Then we have four times zero. Doesn't really matter. Then we have one, zero, one, zero, and zero, zero, one, zero. Let's do it step-by-step. One, two, doesn't matter, one, two, four, eight. Then as we did before, this is our temperature, and this is our comma.

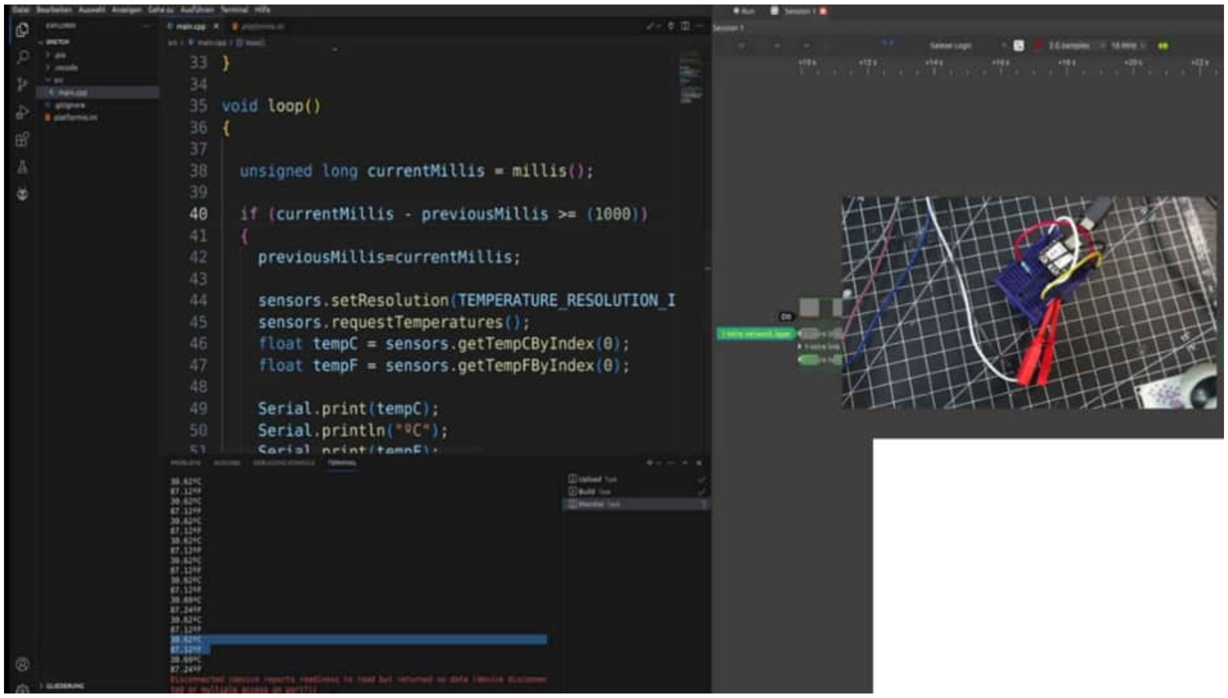
So here in the temperature, we have 10, we have plus, and the comma is two divided by 16, and this should be, the double of this should be, let me think, one, two, five. Is this right? Sounds good to me. So we have, in summary, 10, one, two, five degree Celsius. And this is what we are doing later on when we want to get the data out of the sensor with an Arduino sketch. We have to fetch all of these bits here, or when we want to have the bytes, or in this case, nibbles, because it's all four bits.

And this was just a first analysis so that we can really calculate the values here. And also necessary is that we are having here the least significant bit to read at first, and I will show you in the next project why this is so important.

DATA ANALYSIS WITH LOGIC ANALYZER

So now let's do some analyzing. So my ESP32 is running and giving me the data. I already have the clamps on it so both of the systems are working correctly. As you can see here we're getting the fresh data. Then I clicked on the decoder. I have here on one wire decoder as maybe we already discussed before the zero and I use here on stack decoder one wire network layer so that I get here something out as sample rate 2G and 60 megahertz for the time. Then we click on run and we're getting here some data.

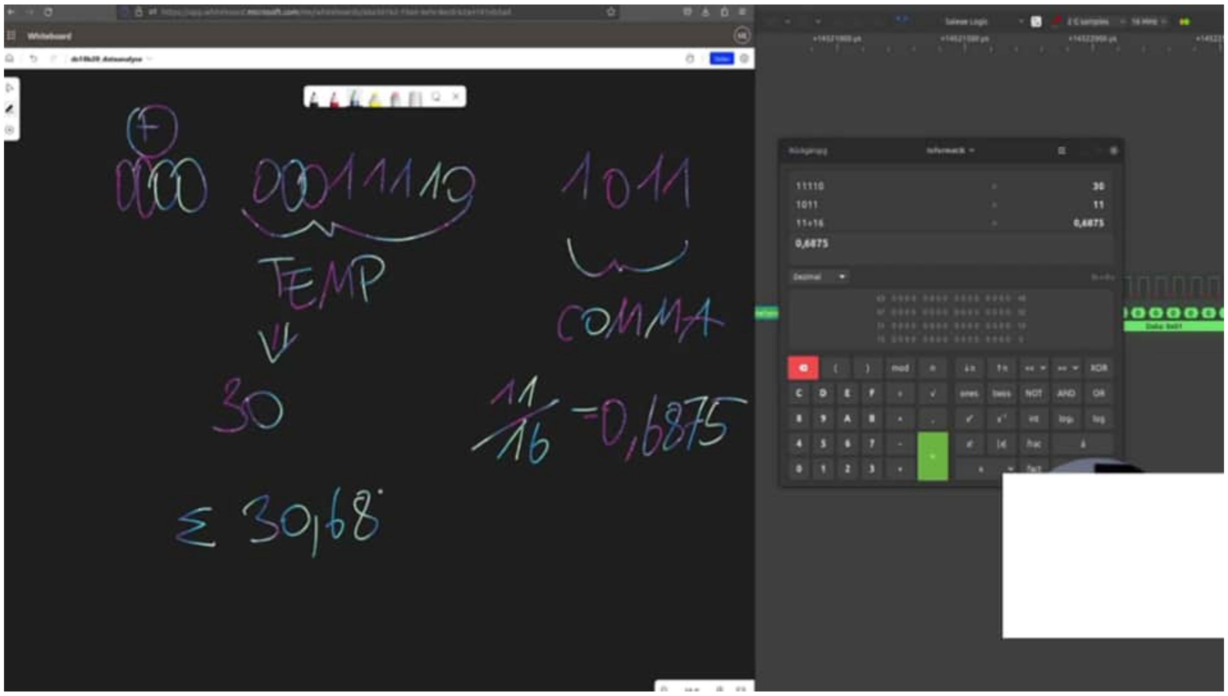
So now that we have here the same results I would say I click here now on stop and also disconnect the ESP and this one was the last value because I think this is not captured anymore as you can see here stop and therefore we are focusing on the last data frame and this should be this value and this would be nice so if you can see here this is the last data frame and this is here the last captured data from the Arduino platform.



```
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000))
  {
    previousMillis=currentMillis;

    sensors.setResolution(TEMPERATURE_RESOLUTION_I
    sensors.requestTemperatures();
    float tempC = sensors.getTempCByIndex(0);
    float tempF = sensors.getTempFByIndex(0);

    Serial.print(tempC);
    Serial.println("ºC");
    Serial.print(tempF);
```

```
30.62ºC
87.12ºF
30.62ºC
87.12ºF
30.62ºC
87.12ºF
30.62ºC
87.12ºF
30.62ºC
87.12ºF
30.62ºC
87.12ºF
30.69ºC
87.24ºF
30.62ºC
87.12ºF
30.62ºC
87.12ºF
30.69ºC
87.24ºF
[illegible]
```

So let's focus and zoom in a little bit in this last data frame and now it would be nice if we can get the data out of it. So 0xBE is the command which sends the MCU to the sensor to say let's start the measurement and now we can calculate our values.



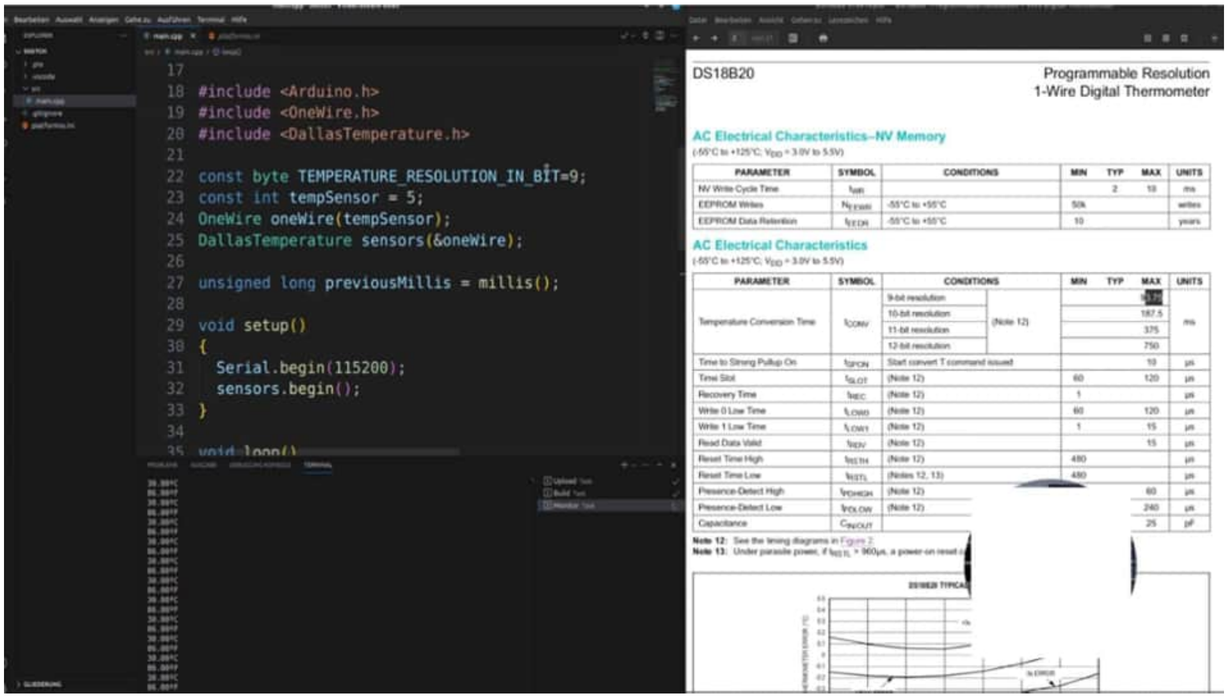
```
11110    =    30
1011     =    11
11÷16    =    0,6875
0,6875
```

So therefore we start with the least significant bit first so the read reading order is from right to left and the first four bits are the sign so zero zero zero zero indicates here and plus sign as we trained before then we have 0 0 0 1 0 0 0 1 then we have 1 1 1 0 1 1 1 0 and the last four bits here is where we are here let's see 0 0 0 1 there we go we have 1 0 1 1 these are the last four bits so this is our plus sign this will be our temperature value and this should be our comma value and if we are calculating everything right we should get the value 30.62 so then let's try it out but now i would like to use here the calculator if you're running a windows machine then you should change here the calculator to scientific or to code or programming modes then you can also add here some binary values and then we are typing in one because the zeros are not relevant for us two three four four times one zero and you can see here it indicates me the decimal value as well or we can hit enter and change to decimal and then we get here the value 30 looks promising to me and the comma value changing back to binary then we have 1 0 1 1 enter we got the value 11 divided by the resolution of 16 we should get the value let's see a decimal 11 divided by 16 16 equals to 0.6875 and we have now a new value from 30.6875 degree celsius this is our desired value is it right let's see 36.875 36.8 so 62 i think with a little bit of um ah this is the last value it rounded to 69 and this could be on 69 so it's roughly here 30.69 cool so this was the first analysis of the register from our one wire protocol easy way i would say just focus a little bit on the data sheet a little bit of bit calculation and then we have our value as we have also in the platform.

CHANGE RESOLUTION TO 9 BIT

And for the sake of completeness, I would like to show you here that we also can change here the resolution For example to 9-bit and in 9-bit resolution, we can fetch data and get data here much much quicker.

So for example when we change to 9-bit We could say here for example Let's do 200 milliseconds Save it uploads the code And now we should get here the data with not so and detailed Value, we're just getting here 30 degrees But much much faster and this is what maybe the use case could be in in in your project But would when you would like the more detailed value, then let's see.



```cpp
#include <Arduino.h>
#include <OneWire.h>
#include <DallasTemperature.h>

const byte TEMPERATURE_RESOLUTION_IN_BIT=9;
const int tempSensor = 5;
OneWire oneWire(tempSensor);
DallasTemperature sensors(&oneWire);

unsigned long previousMillis = millis();

void setup()
{
  Serial.begin(115200);
  sensors.begin();
}

void loop()
```

```
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
30.00°C
86.00°F
```

DS18B20

Programmable Resolution
1-Wire Digital Thermometer

AC Electrical Characteristics–NV Memory

(-55°C to +125°C; V_{DD} = 3.0V to 5.5V)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
NV Write Cycle Time	t_{WR}			2	10	ms
EEPROM Writes	N_{EEWR}	-55°C to +55°C	50k			writes
EEPROM Data Retention	t_{EEDR}	-55°C to +55°C	10			years

AC Electrical Characteristics

(-55°C to +125°C; V_{DD} = 3.0V to 5.5V)

PARAMETER	SYMBOL	CONDITIONS		MIN	TYP	MAX	UNITS
Temperature Conversion Time	t_{CONV}	9-bit resolution	(Note 12)			93.75	ms
		10-bit resolution				187.5	
		11-bit resolution				375	
		12-bit resolution				750	
Time to Strong Pullup On	t_{SPON}	Start convert T command issued				10	µs
Time Slot	t_{SLOT}	(Note 12)		60		120	µs
Recovery Time	t_{REC}	(Note 12)		1			µs
Write 0 Low Time	t_{LOW0}	(Note 12)		60		120	µs
Write 1 Low Time	t_{LOW1}	(Note 12)		1		15	µs
Read Data Valid	t_{RDV}	(Note 12)				15	µs
Reset Time High	t_{RSTH}	(Note 12)		480			µs
Reset Time Low	t_{RSTL}	(Notes 12, 13)		480			µs
Presence-Detect High	t_{PDHIGH}	(Note 12)				60	µs
Presence-Detect Low	t_{PDLOW}	(Note 12)				240	µs
Capacitance	$C_{IN/OUT}$					25	pF

Note 12: See the timing diagrams in Figure 2.
Note 13: Under parasite power, if t_{RSTL} > 960µs, a power-on reset

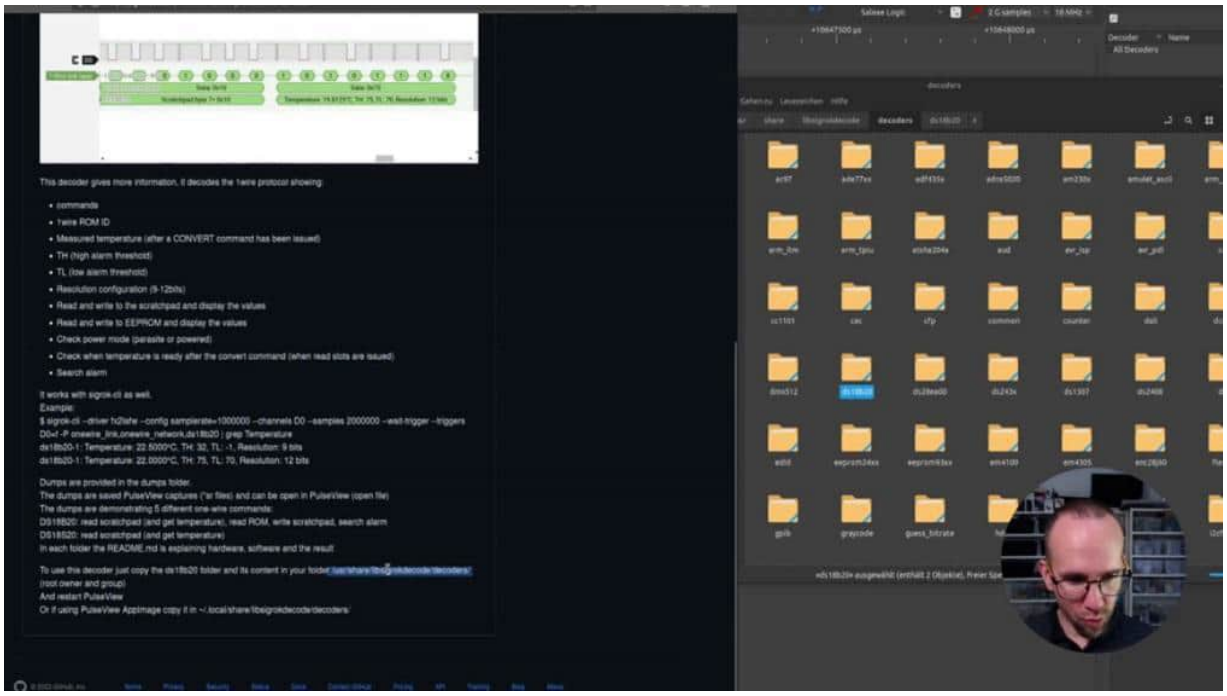
Let's make here one second Then it makes sense to Leave here a little bit more time a little bit higher resolution and get the accurate value as we can see here But this is a matter of what you want to do with your sensor but I just want to show you how easily you can change it with this kind of set resolution function and that you know that the values are Really be rough enough with 9-bit solution Resolution and it was much more detailed with the 12-bit resolution.

ADDING AN EXTERNAL DECODER

In this project, I would like to demonstrate to you how we can import external decoders from third parties in Python. And I found here an external decoder for the DS 18b20¢. So on. And although, or maybe it's already implemented in PowerS viewer, Let's use this use case to show you how we can do that. So there's, guy who wrote this external library, really Chrome, and we can download this folder.

In this folder, there are 2 Python scripts in, which do all of the magic. And where should we copy and paste this folder? To use this decoder, just copy the folder and its contents to user share lip lip sync rock decode decoders. This is exactly what I have here, but I'm using a Linux system here. So if you are running a Windows machine, the first thing is maybe you should consider changing to an analytics distribution in your private little area because it's free, it's open source, and it protects your data much more than Windows does.

And it's, and more the way we are using software in the audio environment. Just, and show notch. And also, I have a good Linux Project, also on this platform. So maybe you are interested in this also. But back to the topic, when you are running a Linux machine, maybe you have the same folder structure.


This decoder gives more information, it decodes the 1wire protocol showing:

- commands
- 1wire ROM ID
- Measured temperature (after a CONVERT command has been issued)
- TH (high alarm threshold)
- TL (low alarm threshold)
- Resolution configuration (9-12bits)
- Read and write to the scratchpad and display the values
- Read and write to EEPROM and display the values
- Check power mode (parasite or powered)
- Check when temperature is ready after the convert command (when read slots are issued)
- Search alarm

It works with sigrok-cli as well.
Example:

```
$ sigrok-cli --driver fx2lafw --config samplerate=1000000 --channels D0 --samples 2000000 --wait-trigger --triggers D0=f -P onewire_link,onewire_network,ds18b20 | grep Temperature
ds18b20-1: Temperature: 22.5000°C, TH: 32, TL: -1, Resolution: 9 bits
ds18b20-1: Temperature: 22.0000°C, TH: 75, TL: 70, Resolution: 12 bits
```

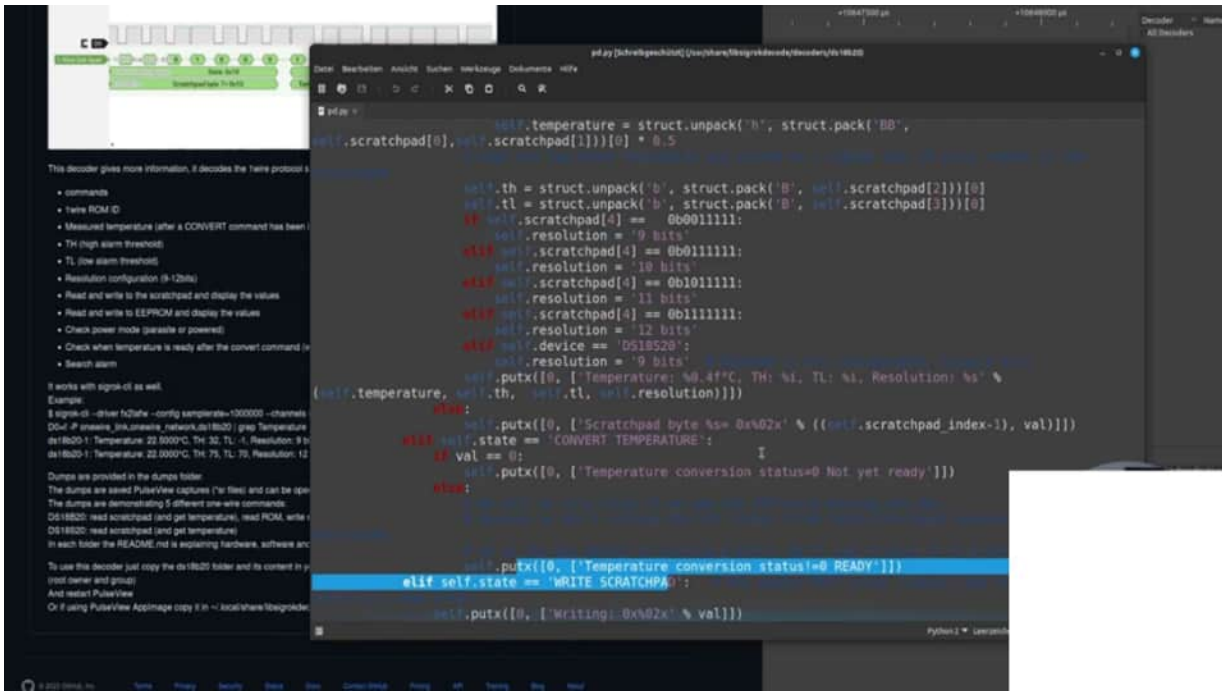
Dumps are provided in the dumps folder.
The dumps are saved PulseView captures (*sr files) and can be open in PulseView (open file)
The dumps are demonstrating 5 different one-wire commands:
DS18B20: read scratchpad (and get temperature), read ROM, write scratchpad, search alarm
DS18S20: read scratchpad (and get temperature)
In each folder the README.md is explaining hardware, software and the result

To use this decoder just copy the ds18b20 folder and its content in your folder /usr/share/libsigrokdecode/decoders/ (root owner and group)
And restart PulseView
Or if using PulseView AppImage copy it in ~/.local/share/libsigrokdecode/decoders/

It could also be the case that this is a hidden folder. So just try out in the explorer, typing in user, share, etcetera. And then maybe you find here in the folder structure, this lip sick rock decode, and then you can copy the files in it. So let's open one of these first, Python script to see what is in it. So first of all, we can find here all of these commands and, what we've seen in the datasheet, scrolling a little bit down, then read scratch cut.

This is also what we have read before. And this is really cool. You can see here, the 0.0625 gives us feedback for the resolution. highest resolution, lowest resolution, we are reading here, the resolution out. And here is where the converter temperature is buried now.

And This is exactly the same as what we did before with our blackboards, but here it is done via Python. Really cool. And now when we have copied this folder to the structure, then we reopen our pulse view so that it could be initialized. And then, so restart, pulse view. And then we can search the decoder here.



```
                self.temperature = struct.unpack('h', struct.pack('BB',
self.scratchpad[0], self.scratchpad[1]))[0] * 0.5

            self.th = struct.unpack('b', struct.pack('B', self.scratchpad[2]))[0]
            self.tl = struct.unpack('b', struct.pack('B', self.scratchpad[3]))[0]
            if self.scratchpad[4] ==   0b0011111:
                self.resolution = '9 bits'
            elif self.scratchpad[4] == 0b0111111:
                self.resolution = '10 bits'
            elif self.scratchpad[4] == 0b1011111:
                self.resolution = '11 bits'
            elif self.scratchpad[4] == 0b1111111:
                self.resolution = '12 bits'
            elif self.device == 'DS18S20':
                self.resolution = '9 bits'
            self.putx([0, ['Temperature: %0.4f°C, TH: %i, TL: %i, Resolution: %s' %
(self.temperature, self.th,  self.tl, self.resolution)]])
        else:
            self.putx([0, ['Scratchpad byte %s= 0x%02x' % ((self.scratchpad_index-1), val)]])
    elif self.state == 'CONVERT TEMPERATURE':
        if val == 0:
            self.putx([0, ['Temperature conversion status=0 Not yet ready']])
        else:

            self.putx([0, ['Temperature conversion status!=0 READY']])
    elif self.state == 'WRITE SCRATCHPAD':

        self.putx([0, ['Writing: 0x%02x' % val]])
```

by typing in DSO18, double click on it, and then we can change the one here, for example, and we should see something here. So let's see here. Let's go a little bit out. And we see here our data, the temperature into 30.375 as we had it before. in our audio, you know, IDE.

It will be in the last byte because you also see here the resolution in 12 bits, etcetera, really, really cool what is made. But we did before the approach by reading it ourselves, but as you can see here, with this externally It's much more convenient. And maybe for your purpose, you need another external decoder, search for it or even write your own decoder with the past few. This is no problem.

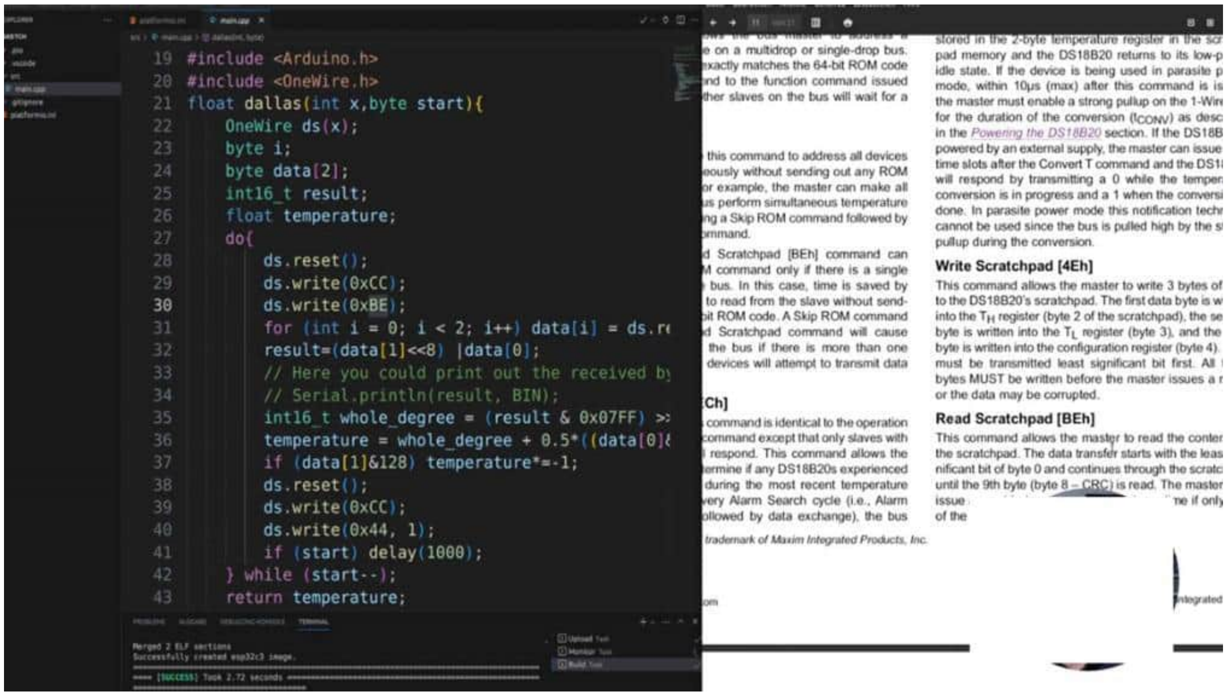
GET DATA ONLY WITH ONEWIRE LIBRARY

In this project, we are just using the 1 wire library to get the temperature data. And therefore, here and finished example because it's a little bit more of programming work. And I go through the main concept, and we also do an analysis with the logic liver to show you how we could do that. So here is the finished example we are just needing here: The one wire in Dallas we could get rid of. So let's compile it in the meantime.

And we are creating, including the one wire because this is our main communication to the whole module. And what we are needing here, let's go through setup. We have here an and function which is called Dellus. And with this Dellus, we just initialize it and then we are fetching the data. So let's see what this tells us.

We are just, pass here 1, that means is it in start or not in the start. So let's see. We're making a 1 wire option here. We are getting an array with 2 spots and an integer with result and also in float temperature. Then we are set, we are resetting the object which indicates that we are starting to communicate. And then we are sending the first command and it's the cc command.

So then we are combining both of the values. And let's see CCH. What is this register that skips RAM? The master can use this command to address all devices on the bus simultaneously. without sending any rom code information. For example, a master can make all DS on the bus perform simultaneous temperature conversion by using a skip wrong command.



```
#include <Arduino.h>
#include <OneWire.h>
float dallas(int x,byte start){
    OneWire ds(x);
    byte i;
    byte data[2];
    int16_t result;
    float temperature;
    do{
        ds.reset();
        ds.write(0xCC);
        ds.write(0xBE);
        for (int i = 0; i < 2; i++) data[i] = ds.r
        result=(data[1]<<8) |data[0];
        // Here you could print out the received by
        // Serial.println(result, BIN);
        int16_t whole_degree = (result & 0x07FF) >:
        temperature = whole_degree + 0.5*((data[0]&
        if (data[1]&128) temperature*=-1;
        ds.reset();
        ds.write(0xCC);
        ds.write(0x44, 1);
        if (start) delay(1000);
    } while (start--);
    return temperature;
```

```
Merged 2 ELF sections
Successfully created esp32c3 image.
===== [SUCCESS] Took 2.72 seconds =====
```

e on a multidrop or single-drop bus. exactly matches the 64-bit ROM code nd to the function command issued ther slaves on the bus will wait for a

this command to address all devices eously without sending out any ROM or example, the master can make all us perform simultaneous temperature ng a Skip ROM command followed by ommand.

d Scratchpad [BEh] command can M command only if there is a single bus. In this case, time is saved by to read from the slave without send- bit ROM code. A Skip ROM command d Scratchpad command will cause the bus if there is more than one devices will attempt to transmit data

Ch]

command is identical to the operation command except that only slaves with l respond. This command allows the ermine if any DS18B20s experienced during the most recent temperature very Alarm Search cycle (i.e., Alarm ollowed by data exchange), the bus

trademark of Maxim Integrated Products, Inc.

stored in the 2-byte temperature register in the scr pad memory and the DS18B20 returns to its low-p idle state. If the device is being used in parasite p mode, within 10µs (max) after this command is is the master must enable a strong pullup on the 1-Wir for the duration of the conversion (t_{CONV}) as desc in the Powering the DS18B20 section. If the DS18B powered by an external supply, the master can issue time slots after the Convert T command and the DS1 will respond by transmitting a 0 while the temper conversion is in progress and a 1 when the conversi done. In parasite power mode this notification techr cannot be used since the bus is pulled high by the s pullup during the conversion.

Write Scratchpad [4Eh]

This command allows the master to write 3 bytes of to the DS18B20's scratchpad. The first data byte is w into the T_H register (byte 2 of the scratchpad), the se byte is written into the T_L register (byte 3), and the byte is written into the configuration register (byte 4). must be transmitted least significant bit first. All bytes MUST be written before the master issues a r or the data may be corrupted.

Read Scratchpad [BEh]

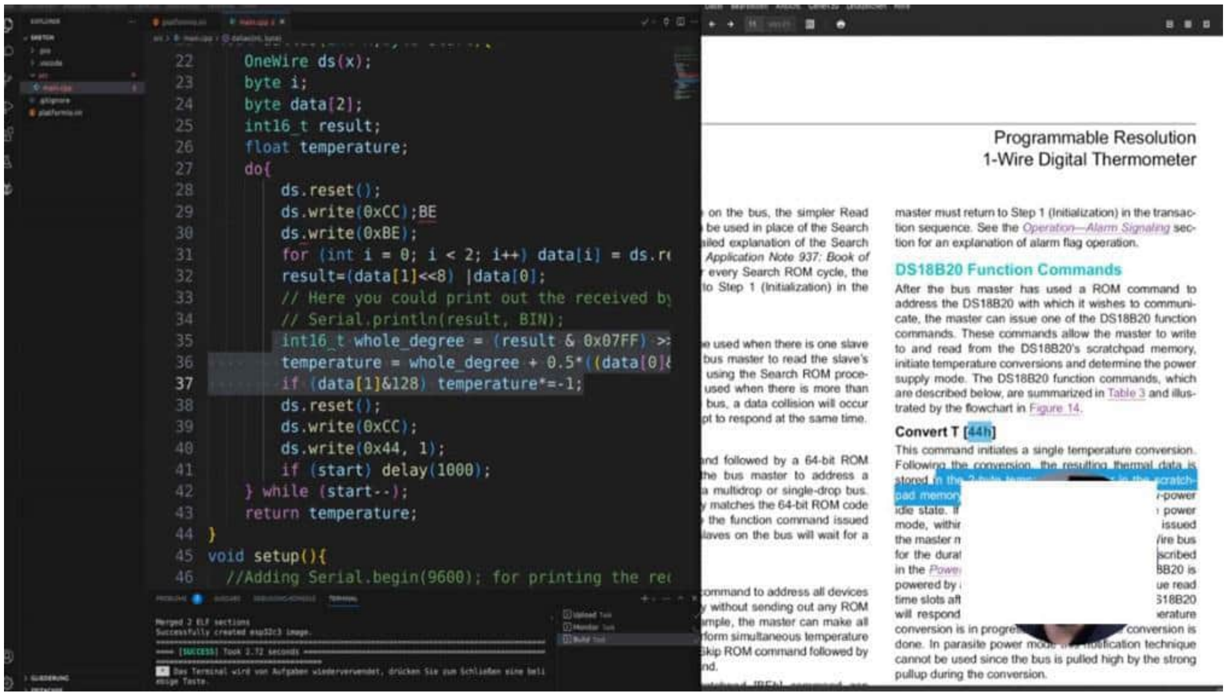
This command allows the master to read the conter the scratchpad. The data transfer starts with the leas nificant bit of byte 0 and continues through the scratc until the 9th byte (byte 8 – CRC) is read. The master issue ne if only of the

This is what we do here. We're just addressing all of the temperature sensors, what are here. Then note that the read scratch bat command can follow the skip RAM only if there's a single slave device on the In this case, time is saved by allowing the master to read from the slave. Okay. In this case, we are using the BE room, the next one.

This is the weeds scratch pad as we did also in the previous examples. And then we're reading out the data. And I'm sure you can read this line because we have the least significant bit in the first place. So that means first, we're getting: do we have the pass view open? Yes.

So that means we're reading from right to left, but we are getting this is in the index 0 and this is in the 1. So first, we are using index 1. This is this 1. Pitch shifting to the left, the result is an integer 16, not in bytes. And we make an all operation so that this will be, a both bytes together, and then we can calculate, here is some calculation going on with, in the code.

And afterwards, we are writing here in CCR. Let's see what the CC is. as we did before, skip RAM, and then we have 44 h. What is this command? This command initiates a single temperature conversion. Following the conversion, the resulting terminal data is in a 2 byte temperature register from this catch bug memory, this is exactly what we did before by reading out those 2 bytes.



```
    OneWire ds(x);
    byte i;
    byte data[2];
    int16_t result;
    float temperature;
    do{
        ds.reset();
        ds.write(0xCC);BE
        ds.write(0xBE);
        for (int i = 0; i < 2; i++) data[i] = ds.r
        result=(data[1]<<8) |data[0];
        // Here you could print out the received by
        // Serial.println(result, BIN);
        int16_t whole_degree = (result & 0x07FF) >>
        temperature = whole_degree + 0.5*((data[0]&
        if (data[1]&128) temperature*=-1;
        ds.reset();
        ds.write(0xCC);
        ds.write(0x44, 1);
        if (start) delay(1000);
    } while (start--);
    return temperature;
}
void setup(){
  //Adding Serial.begin(9600); for printing the re
```

Programmable Resolution
1-Wire Digital Thermometer

on the bus, the simpler Read be used in place of the Search iled explanation of the Search Application Note 937: Book of every Search ROM cycle, the to Step 1 (Initialization) in the

e used when there is one slave bus master to read the slave's using the Search ROM proce- used when there is more than bus, a data collision will occur pt to respond at the same time.

nd followed by a 64-bit ROM he bus master to address a a multidrop or single-drop bus. y matches the 64-bit ROM code the function command issued laves on the bus will wait for a

ommand to address all devices y without sending out any ROM mple, the master can make all form simultaneous temperature kip ROM command followed by nd.

master must return to Step 1 (Initialization) in the transaction sequence. See the Operation—Alarm Signaling section for an explanation of alarm flag operation.

DS18B20 Function Commands

After the bus master has used a ROM command to address the DS18B20 with which it wishes to communicate, the master can issue one of the DS18B20 function commands. These commands allow the master to write to and read from the DS18B20's scratchpad memory, initiate temperature conversions and determine the power supply mode. The DS18B20 function commands, which are described below, are summarized in Table 3 and illustrated by the flowchart in Figure 14.

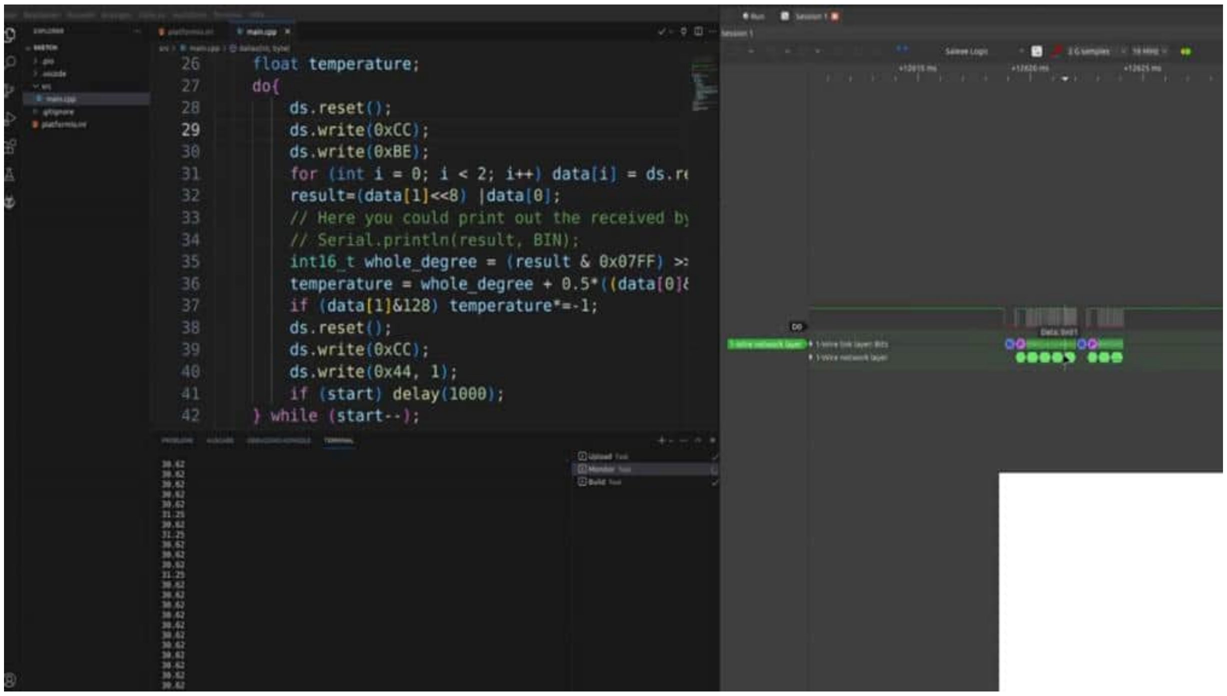
Convert T [44h]

This command initiates a single temperature conversion. Following the conversion, the resulting thermal data is stored in the scratch- pad memory -power idle state. power mode, withir issued the master n /ire bus for the dura scribed in the Powe B20 is powered by ue read time slots aft 318B20 will respond erature conversion is in progre conversion is done. In parasite power mo fication technique cannot be used since the bus is pulled high by the strong pullup during the conversion.

Okay. Cool thing. That's all we need to perform those actions. And this is what we do here. So no, not starting.

a 3 is the pin, the GPIO, where the sensor is on, and that's it. So I've already uploaded it. So let's see what the serial monitor prints us out. Now we can open up the pulse view again, and let's inspect what we get here. We should now get here and a shorter time data frame because we are just sending here.

Let's see a few commands out, and that's it. So we click on stop, zooming in to the time frame or to the date frame. And we can see here we have here our skip RAM, our BEA command, and then we have these two bytes where our data from the sensor is stored. Then we're reset. We're sending CCM and, next reading command out to the sensor.



```
    float temperature;
    do{
        ds.reset();
        ds.write(0xCC);
        ds.write(0xBE);
        for (int i = 0; i < 2; i++) data[i] = ds.r
        result=(data[1]<<8) |data[0];
        // Here you could print out the received b
        // Serial.println(result, BIN);
        int16_t whole_degree = (result & 0x07FF) >
        temperature = whole_degree + 0.5*((data[0]&
        if (data[1]&128) temperature*=-1;
        ds.reset();
        ds.write(0xCC);
        ds.write(0x44, 1);
        if (start) delay(1000);
    } while (start--);
```

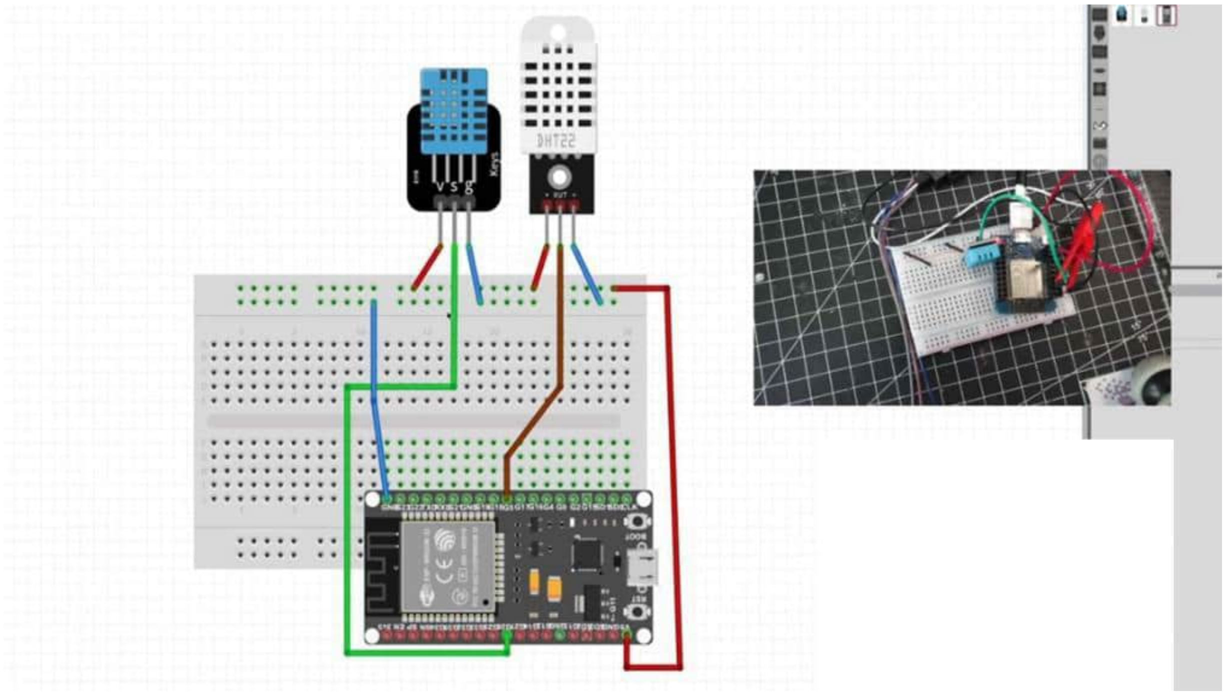
```
30.62
30.62
30.62
30.62
30.62
31.25
30.62
31.25
30.62
30.62
30.62
31.25
30.62
30.62
30.62
30.62
30.62
30.62
30.62
30.62
30.62
30.62
30.62
30.62
```

That's it. That's all that we need. And this code can be fulfilled also. For example, with an 80 tiny with limited space, etcetera, we just need an and 1 wire library. And there are also some tiny 1 wire libraries so that we don't have to do all the things by ourselves.

and not really more is needed for getting temperature values with an arduino or ESP.

CIRCUIT AND WIRING

And as always, in a new chapter, we start with some basic sketch and the wiring part so that we then lay down and can't catch the data. The DHT 11 or DHT 22 is also communicating via such icon 1 wire protocol. It's not really standardized, but, therefore, we're only needing also here one data line. So VCC and ground is to Bradbot, you can choose 5 fold or 3 point three volt, and then choose 1 GPI at home for the data communication with your USB 32, for example, per. I'm using here, as you can see here, on prep, so, a suction module, and this module has already and re system, pull up our system at the data line.

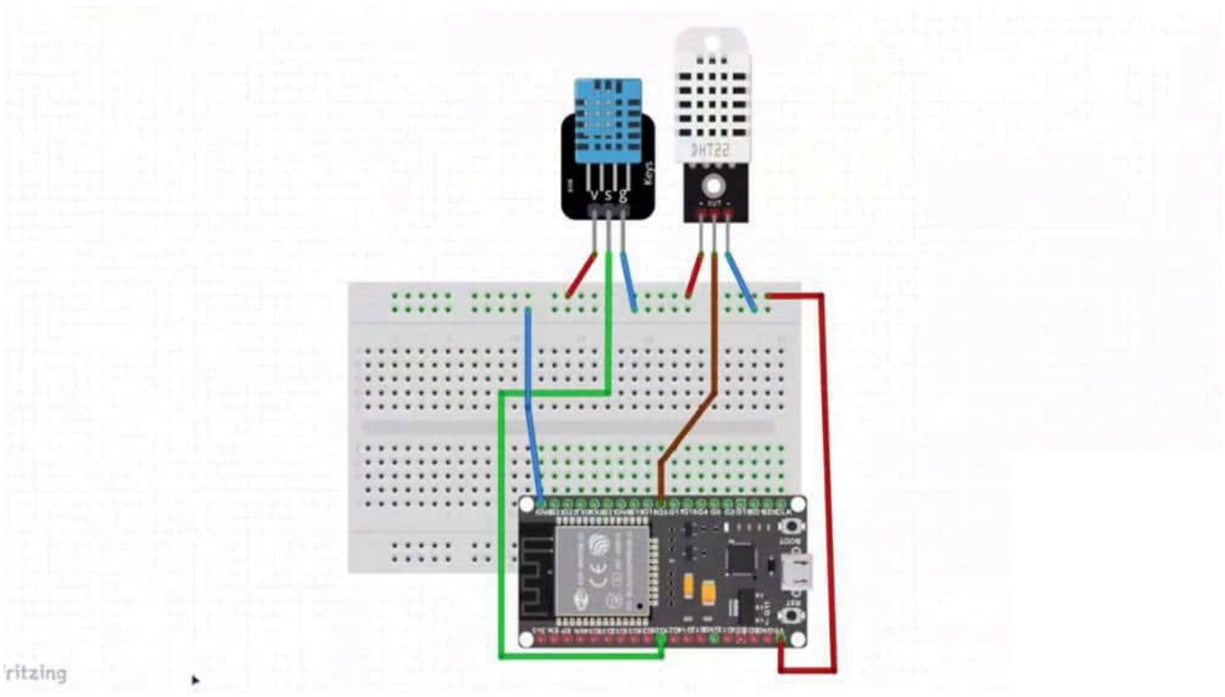


If you are using the raw temperature sensor. Then you need, I think, 10 kilo ohms, 4.7 kilo ohms resistor and pull up resistor to get here the data line. high. Also, I fear the wiring for both of the sensors, but I'm only using the DHT11. It's the cheaper version, but not so accurate.

But for the most part, it's absolutely sufficient enough. But if you would like to have more range here because I think that DHT 11 is can't or is not able to be measured here, a minus degrees. So, therefore, you need a DHT 22. much more, range of the temperature and humidity and also, more accurate, but costs a little bit more. The approach is nearly the same as we see later on.

you have to calculate it a little bit differently, but you can follow me along, but have to adapt certain steps then later also in the calculation. Yeah. That's it. That's the first step of the rhyme part, and I also connected here the clamps for the logic analyzer directly to the pins on the GPIO, and that's it. All that we need.

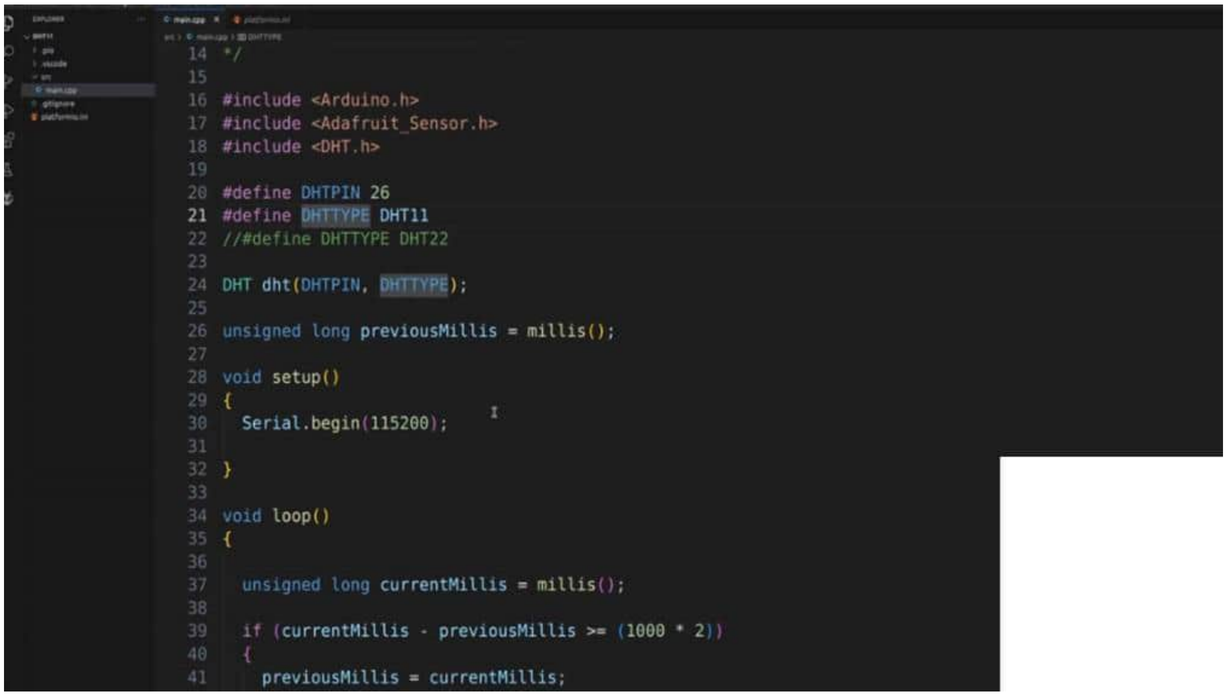
Nothing more to do here. And then we are switching to our basic sketch. And what we are doing here is, we are adding here either food sensor and the DHT library. That's the first thing we need to do. And it is under Arlino IDE, then you can just search for the DHT Center Library for Adafruit, and both of them can be used for the 22 and the 11.



So then let's start by defining, yeah, the pin. And I think the example does it with some defines and not with const bytes. So let's keep it like in the example from the library. define the HD type, and this is in my case, DHT 11. If you use 22, then you have to change the type here.

would use here defined HD type DHT 22. Then I'm using here an unsigned long previous Millies. That's correct. And I'm creating an object here. A DHT object, write this, with the name DHT in some lower letters.

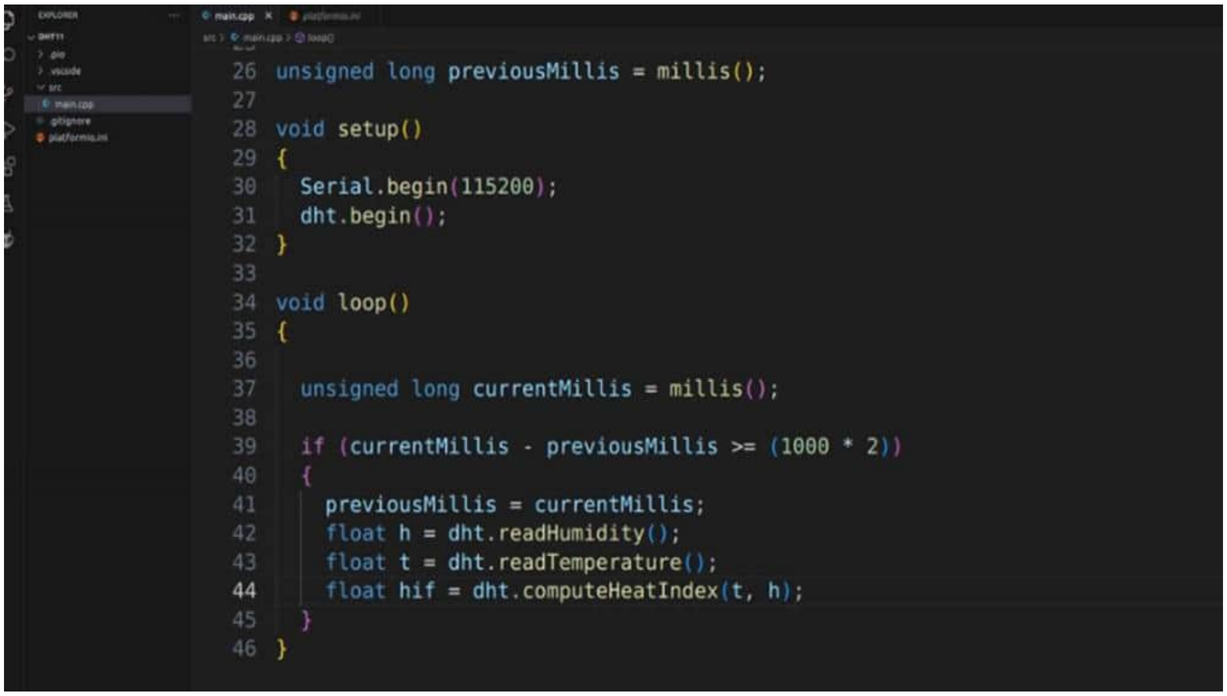
and I have to pass here the DHT pin, which we defined and the DHT type. So it's much more readable if we're using the definitions. Okay. Then dht dot Begin And now we can start in our loop with getting the data. And therefore, we are using some floats here for humidity, DHT read immunity, I think IntelliSense is such a good thing, or you just start typing and getting the name of the function.



```
*/

#include <Arduino.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>

#define DHTPIN 26
#define DHTTYPE DHT11
//#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);

unsigned long previousMillis = millis();

void setup()
{
  Serial.begin(115200);

}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000 * 2))
  {
    previousMillis = currentMillis;
```

Very good. Then we have a float, t is DHT reads a temperature. Nice one. And the DHT library has some special things in it. It's called the heat index, I think, heat index. DHTR Compute heat index.

What is the heat index? I have to pass here the temperature and the humidity. The heat index is something like, apparent temperature that's combining the temperature and humidity, and it should tell you how it's be, it takes into account a combination of these effects of the temperature and humidity on human comfort. Do you feel it? Is it, more, is it lower in the feeling, or is it higher?



```
unsigned long previousMillis = millis();

void setup()
{
  Serial.begin(115200);
  dht.begin();
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000 * 2))
  {
    previousMillis = currentMillis;
    float h = dht.readHumidity();
    float t = dht.readTemperature();
    float hif = dht.computeHeatIndex(t, h);
  }
}
```

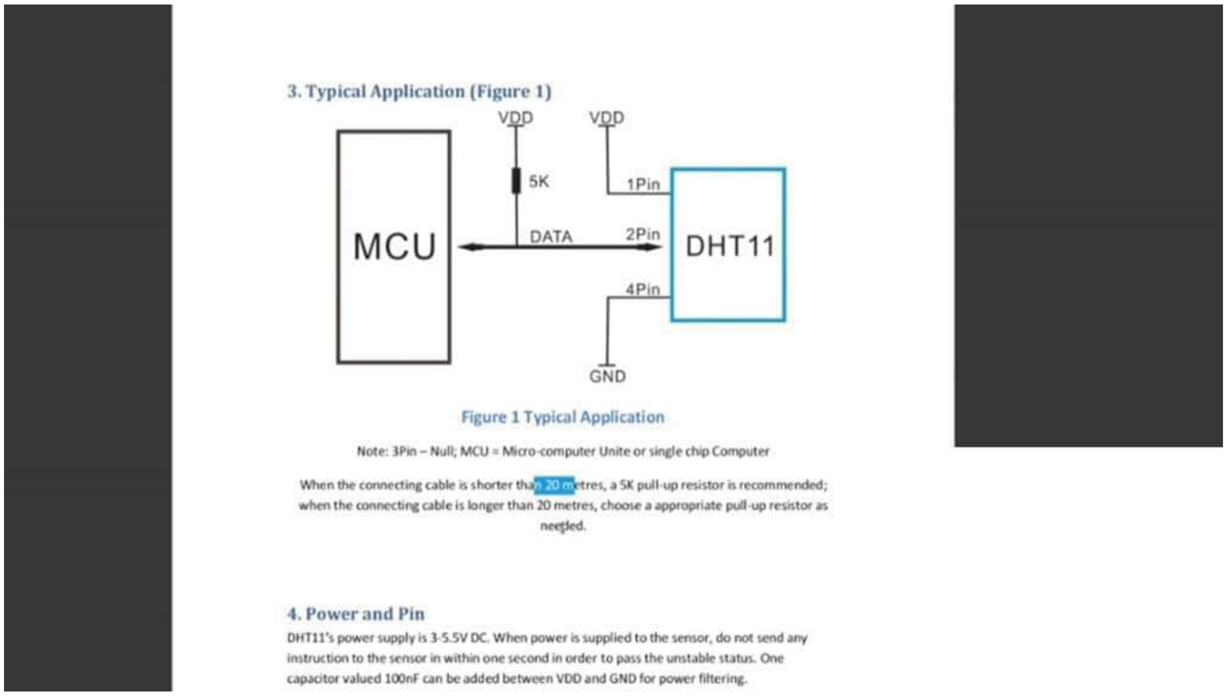
And this will be heated . So just that, you know, that you can also calculate it. And we take a closer look also at this computing heat index function later on when we're inspecting the library. So now we are printing out all of the stuff. I've already prepared the server printout, and then we're uploading the code and opening the serial monitor, and we're getting 30 degrees straight and, humidity 57, 58 heat index, 25. Okay.

Sounds good to me. For the first tests that we get here, the humidity and temperature data, that we can proceed.

UNDERSTANDING THE PROTOCOL AND SIGNAL ANALYSIS

Now let us start by inspecting the protocol a little bit more in detail from the DHT 22 or DHT 11. So I have both data sheets here, but we are focusing only on a DHT 11. You can follow me along with the DHT 22, but have to adapt here. on, details. So, therefore, let's close this one.

Open up the DHT 11 data sheet. scrolling a little bit down so that we can see our first in force what we are needing. The first thing is that the temperature and humidity accuracy is not really in a way that we could say is an industry standard. But in our spare time hobby project, thickering projects, it would be accurate enough. But you can see M 2 degrees is some significant change and it could make a significant difference.



3. Typical Application (Figure 1)

Figure 1 Typical Application

Note: 3Pin – Null; MCU = Micro-computer Unite or single chip Computer

When the connecting cable is shorter than 20 metres, a 5K pull-up resistor is recommended; when the connecting cable is longer than 20 metres, choose a appropriate pull-up resistor as needed.

4. Power and Pin

DHT11's power supply is 3-5.5V DC. When power is supplied to the sensor, do not send any instruction to the sensor in within one second in order to pass the unstable status. One capacitor valued 100nF can be added between VDD and GND for power filtering.

Scrolling a little bit down. We can see here our resolution has 8 bits. It's also not really accurate, in comparison with the DS ATB 20, for example, also the DHT 22. typically application with this, pull up resistor. And also you can see here, if you have longer distances, you can reduce the pull up resistor, for example.

Here, power and pins are also interesting. We can use here the logic level from the ESP32, the 3.3 volt, and also the 5 volt Ardino, both of them are sufficiently enough. And you could add here a capacitor for filtering out any noises. So here, the communication process, this is what I would like to see here. Singapore, why are two ways interesting?

So it's just not a standard format. It's a single bus data form from RTCs to communicate and synchronize between MCU and DHT sensor. It is about 4 milliseconds, the whole process. And the data consists of 40 bits. Okay.

Interesting. and start with the higher data bit. Okay. That means we can read from left to right. Here the first eight bits are the relative formality data, the decimal data, then we have the temperature and the decimal temperature.

And then check some If the data is trans transmission is right, the checksum should be the last 8 bits. Okay. The sum of this I see. I see. And here we have a good overview of the communication process.

And this is now exactly what I would like to see. We can find out the same thing, with our example here. So I've, my sketch running. You can see a certain degree Celsius. And I have here, the humidity by 58. The PulseViewer is already set up.

1 channel, no decoder. I have added 1 gm here and 8 Megahertz could change it also for 60 megahertz, for example, and let's run it. And now it would be interesting if we cut the communication because then we can be sure that we have the right values here. So Stop. Here we have.

Here we have the last value. So the last value should be 30 57, not 58. And this is what I'm now would like to go a little bit more in detail with the communication process. So we're focusing and zooming in, then let's open the communication process here. Let's go a little bit down.

So let's have here the senior bus free status is high voltage level. This is all before what we have here. You have the high line. And then when the communication between the MCU and DHT 11 begins, the program sets the data voltage from high to low. This is this area.

5.1 Overall Communication Process (Figure 2, below)

When MCU sends a start signal, DHT11 changes from the low-power-consumption mode to the running-mode, waiting for MCU completing the start signal. Once it is completed, DHT11 sends a response signal of 40-bit data that include the relative humidity and temperature information to MCU. Users can choose to collect (read) some data. Without the start signal from MCU, DHT11 will not give the response signal to MCU. Once data is collected, DHT11 will change to the low-power-consumption mode until it receives a start signal from MCU again.

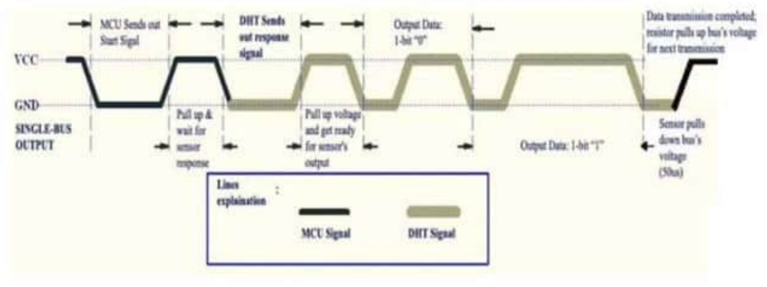

MCU Sends out Start Sigal
DHT Sends out response signal
Output Data: 1-bit "0"
Data transmission completed; resistor pulls up bus's voltage for next transmission
VCC
GND
SINGLE-BUS OUTPUT
Pull up & wait for sensor response
Pull up voltage and get ready for sensor's output
Output Data: 1-bit "1"
Sensor pulls down bus's voltage (50us)
Lines explaination :
MCU Signal
DHT Signal

Figure 2 Overall Communication Process

5.2 MCU Sends out Start Signal to DHT (Figure 3, below)

Data Single-bus free status is at high voltage level. When the communication between MCU an

And this process must take at least 80 milliseconds. So let's measure. What do we have here? bum. bum.

Okeydoke. Then we have 20 milliseconds here. And this is because the light Bree is set here at 20 milliseconds, and we will see this in one of the next projects when we also investigate the library. So this is exactly what we want. 20 milli at least 80 milliseconds to low.

And then the MCU pulls up the voltage for 20 microseconds. So we're zooming in. We have here the next high, and this should be 20 mil microseconds, not seconds and with your 13 micros 13. So I don't know why we are not following the standards here. It could be that this is some intentional purpose inside the library.

I don't know, but it works. So here, we have a little glitch. the DHT sends out a response signal and keeps it for 80 microseconds. So this is now, if both can communicate with each other, and we have here 80 3 microseconds. And then, DHT pulls up voltage and keeps it for 80 microseconds.

And this is where we have started with intolerance. So we have here, the rest of the 40 bits should be here. So let's go down a little bit. Here's the explanation in text form. And now, something really interesting in this kind of protocol is that we have between each data and spare time from around 50 microseconds, as we can see here, or, for example, here, these are the 50 microseconds.

And now, how can we interpret the data? We have here some short impulses and some longer pulses. So the short pulses which run about 26 to 28 microseconds, are logically 0. So here, we have 23. Here, we have 23.

so this focuses on the Data length, is, yeah, 28 microseconds and it is in 0. And if we have 17, 17 microseconds, then we got here and logical 1. And this is now how we can calculate our values. So going up a little bit because we want to have here first one is the relative humidity and then the decimal values. So let's focus on the first eight bits.

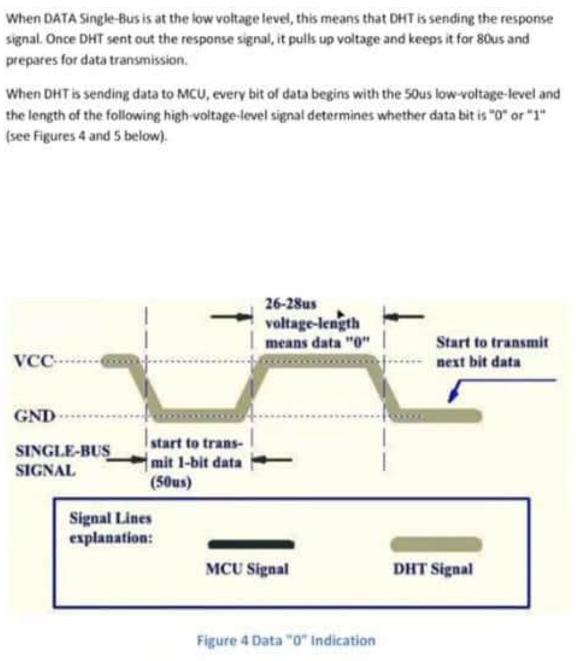
1, 2, 3, 4 of 5, 6, 7, 8. Open up the calculator. Change to binary, and we're typing in from left to right is 00, and this one is three times a 1, then we have 1. 1. Hit enter.

And then we are changing to decimal. And we got here at 57. 57 is the last value we got here, humidity, because 58 was the previous value. Amazing. It works.

So jumping back to our calculator. The next eight bits are the comma value. So 1, 2, 3, 4, 5. 678. As we can see, 0, we have 8 bit resolution.

So not really anything here. To mention, then the temperature data should start. 345678. changing to binary, typing in the binary data, 3 times in serum, 4 times 1, and 0. Hitting enter, we can see here the 30 in the decimal values and this is exactly what we have here.

Then we have 1, 2, 3, 4, 5, 6, 7, 8. 1, 2, 3, 4, 5, 6, 7, 8. And this is, the comma values from the temperature. And now the last 1, 2, 3, 4, 5, 6, 7, 8 is also interesting because this should be the checksum and the checksum should be the sum of all of the values. So that means 30+57 should be 87.

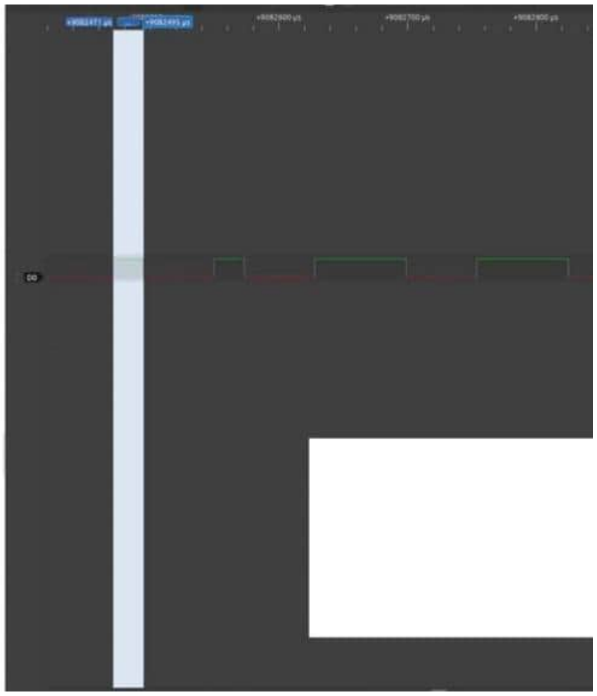


When DATA Single-Bus is at the low voltage level, this means that DHT is sending the response signal. Once DHT sent out the response signal, it pulls up voltage and keeps it for 80us and prepares for data transmission.

When DHT is sending data to MCU, every bit of data begins with the 50us low-voltage-level and the length of the following high-voltage-level signal determines whether data bit is "0" or "1" (see Figures 4 and 5 below).

26-28us voltage-length means data "0"
Start to transmit next bit data
VCC
GND
SINGLE-BUS SIGNAL
start to transmit 1-bit data (50us)
Signal Lines explanation:
MCU Signal
DHT Signal

Figure 4 Data "0" Indication



If I'm calculating your rights, changing to binary, we have and 1. Now, 0, 10, 10, and 3 times the one hitting enter. And we see 87 is the desired value for our checksum. And now we can be sure that we have read the values in the correct way. The last spare time and then the data line is set to high And this means the communication process is over.

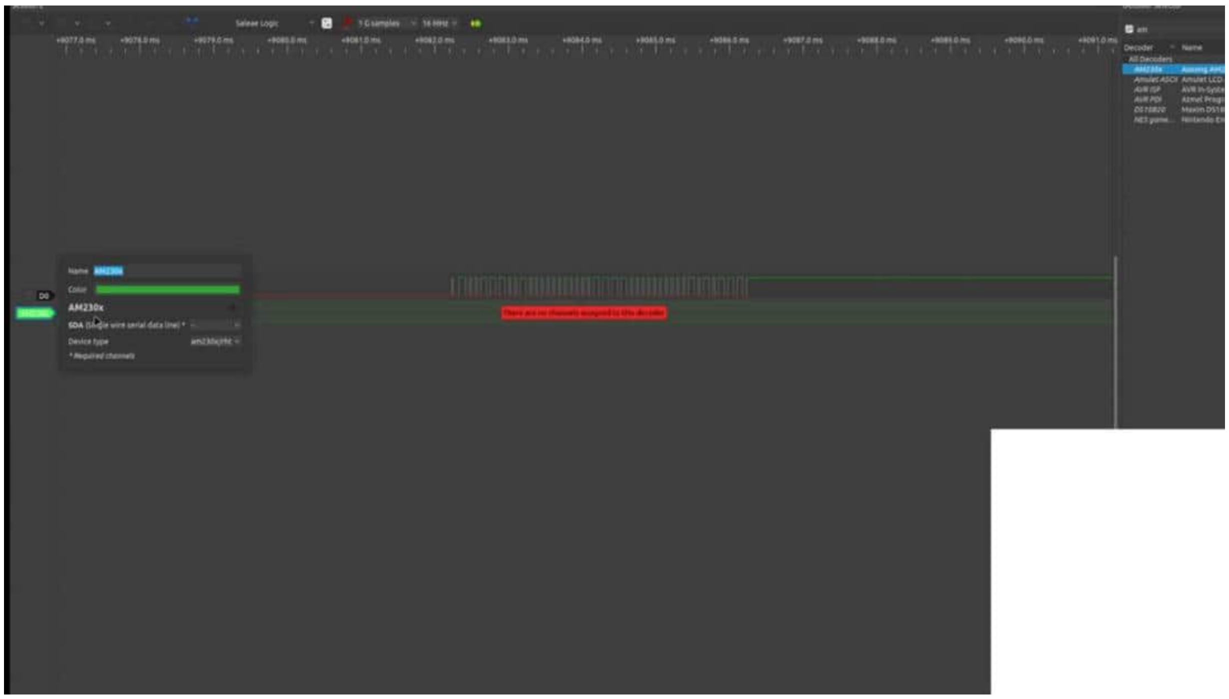
Very good. And this is how the DHT 11 sensor communicates with an MCU, for example, so let's see how long the whole data process is. It's only 20 milliseconds? No. It's 20.4.

Ah, not really 24 milliseconds and what was in the datasheet around about 4 milliseconds. Yes. 4 milliseconds is the whole data processing. Yes. 4 milliseconds.

That works really well. So we have now checked everything on a bit level, on a database level, and In the next projects, we're trying to find out if we have it here also in the decoder.

USING THE DHT11 DECODER

And of course, there are also some decoders out there. But to understand the protocol and all of the necessary content of the datasheet, it's necessary that we are not jumping too fast into some decoders. Also, it's beneficial and helps us, supports us. But at the first point, just try to understand all of the meanings from a datasheet. Then we open up here and we are typing in, it's an AM230, because this is the sensor inside this DHT. We click here on Enter.

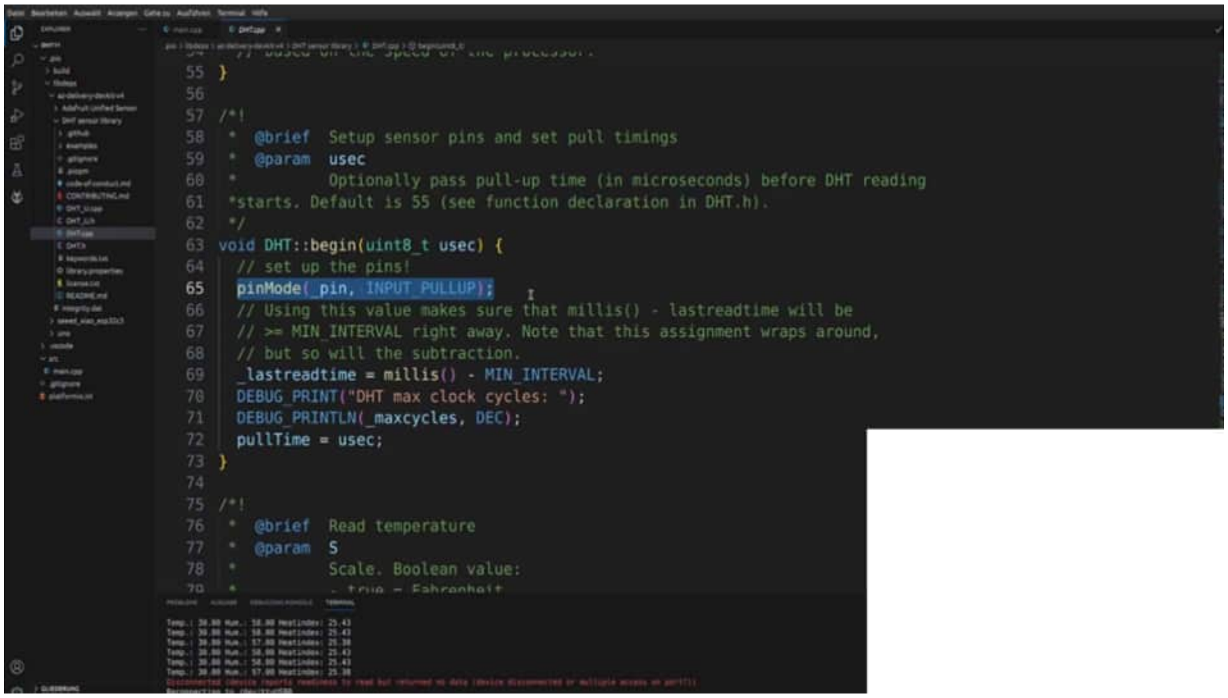


Speaker 1 (00:00:38) - And now we can add here the data line, this is D0. And now here we can change between the DHT11 and the DHT22. And if I'm choosing here the DHT11, now we can get here in a convenient way all of this stuff that we did before by counting the binary values. So as you can see, easy doing, easy as an overview, but not really helping us to understand all of the content. But why not use it when you are trying to find out if there is some error, for example, in your circuit.

INSIGHT INTO THE DHT LIBRARY

And if you are more interested in how the library is working, you can inspect it by clicking with CTRL and, for example, a mouse click. You can jump into the DHT folder, you can also use the Arduino IDE, you jump into the folder structure where all of your libraries are saved. And then you can always find a CPP and .h because this is in an object-oriented way. That means here in the header data is mentioned all functions and variables which are used. And in the CPP you find all of the code from this library.

And you can go through this library to understand how they managed to read out those values in Arduino code. And I would like to go a little bit down so that you can see, for example, that the DHT library implemented the pin mode so that we can read the values.



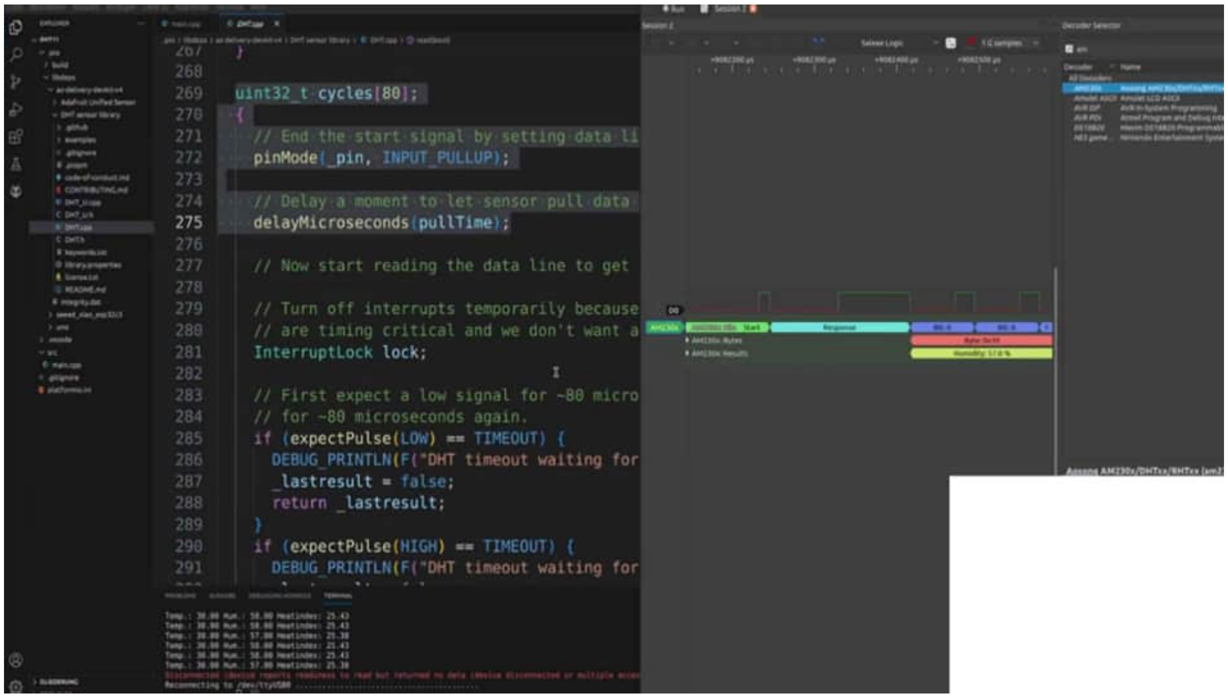
```
  // based on the speed of the processor.
}

/*!
 *  @brief  Setup sensor pins and set pull timings
 *  @param  usec
 *          Optionally pass pull-up time (in microseconds) before DHT reading
 *starts. Default is 55 (see function declaration in DHT.h).
 */
void DHT::begin(uint8_t usec) {
  // set up the pins!
  pinMode(_pin, INPUT_PULLUP);
  // Using this value makes sure that millis() - lastreadtime will be
  // >= MIN_INTERVAL right away. Note that this assignment wraps around,
  // but so will the subtraction.
  _lastreadtime = millis() - MIN_INTERVAL;
  DEBUG_PRINT("DHT max clock cycles: ");
  DEBUG_PRINTLN(_maxcycles, DEC);
  pullTime = usec;
}

/*!
 *  @brief  Read temperature
 *  @param  S
 *          Scale. Boolean value:
 *          - true = Fahrenheit
```

```
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 57.00 HeatIndex: 25.38
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 57.00 HeatIndex: 25.38
Disconnected (device reports readiness to read but returned no data (device disconnected or multiple access on port?))
Reconnecting to /dev/ttyUSB0
```

Because often we have to do it on our own, but here, when we are starting in the setup to begin, this will be done here. Then going a little bit out, reading the temperature.

Then we can see, for example, that the sensor is now read out with some helper function, converting the Celsius to Fahrenheit, for example. I'm going a little bit down because I have the DHT11 here, this will be something for us. But it's not really interesting here. I would like to see if we find a little bit more here, compute the heat index. Also interesting how this works, you see a lot of mathematics is going on here. Pow wow wow, much rocket science. Here, read. This is what was the interesting part.

Because we have the 40 bits to receive all of the data. So we have an array here. And now the magic begins with, for example, let me see, DHT11, delay 20. Datasheet says at least 80 milliseconds, 20 to be safe. This was the first start. So this line, no, here we go. This 20 milliseconds is done here with this delay. And then some magical things will happen where the start conditions start. So that means this was this part with the, what was it, 80 seconds down and high.



```
  }

  uint32_t cycles[80];
  {
    // End the start signal by setting data li
    pinMode(_pin, INPUT_PULLUP);

    // Delay a moment to let sensor pull data
    delayMicroseconds(pullTime);

    // Now start reading the data line to get

    // Turn off interrupts temporarily because
    // are timing critical and we don't want a
    InterruptLock lock;

    // First expect a low signal for ~80 micro
    // for ~80 microseconds again.
    if (expectPulse(LOW) == TIMEOUT) {
      DEBUG_PRINTLN(F("DHT timeout waiting for
      _lastresult = false;
      return _lastresult;
    }
    if (expectPulse(HIGH) == TIMEOUT) {
      DEBUG_PRINTLN(F("DHT timeout waiting for
```

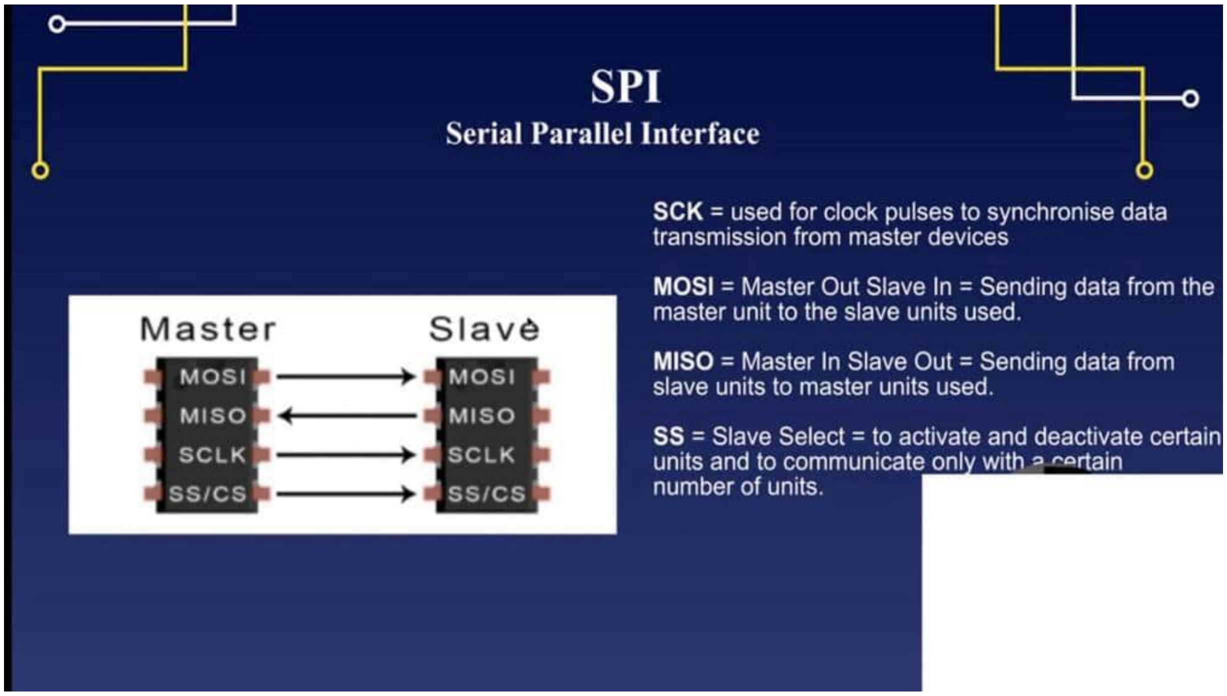
```
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 57.00 HeatIndex: 25.38
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 58.00 HeatIndex: 25.43
Temp.: 30.00 Hum.: 57.00 HeatIndex: 25.38
Disconnected (device reports readiness to read but returned no data (device disconnected or multiple acce
Reconnecting to /dev/ttyUSB0 ..............................
```

And then the real magic starts with interpreting all of the zeros and ones by measuring how long are the pulses. So it's not really easy to measure here without any interruptions to the pulses. But you can inspect it if you want to go a little bit deeper in this whole thing, how this library will do it. So they make a change if it's short or not short. And then, yeah, they will calculate the temperature and the humidity. So you see, simple to understand, not so simple on the Arduino code basis.

But we will do our own libraries later on with the E2C. It's a more common way. It's a more, I would say, convenient approach to write your own libraries. But here you can investigate how it's done for the DHT11.

BASICS

Now, we come to, next protocol, which is called SPI serial parallel interface. And this SPI is also widely used in the audio environment. So if you're using an AT tiny, RT Mega, and ESP, you will find general GPIOs. The thing is, if you are using a microcontroller, which does not have so many in GPIOs, SPI wouldn't be the 1st protocol with what you are using because we are needing 4 data lines, 1 ground. So 5 lines, jumper cables, 5 GPS.



SPI
Serial Parallel Interface

SCK = used for clock pulses to synchronise data transmission from master devices

MOSI = Master Out Slave In = Sending data from the master unit to the slave units used.

MISO = Master In Slave Out = Sending data from slave units to master units used.

SS = Slave Select = to activate and deactivate certain units and to communicate only with a certain number of units.

And this is not often very suitable for all circumstances. Also, we are not there's no need for a full duplex communication. And I come to this later on because in one of the next chapters, we are talking about the e squared c, which is widely used and only needs tools. lines, jumper, cables, connections. And, e squared, he can also only talk half duplex, and here we can talk in full duplex.

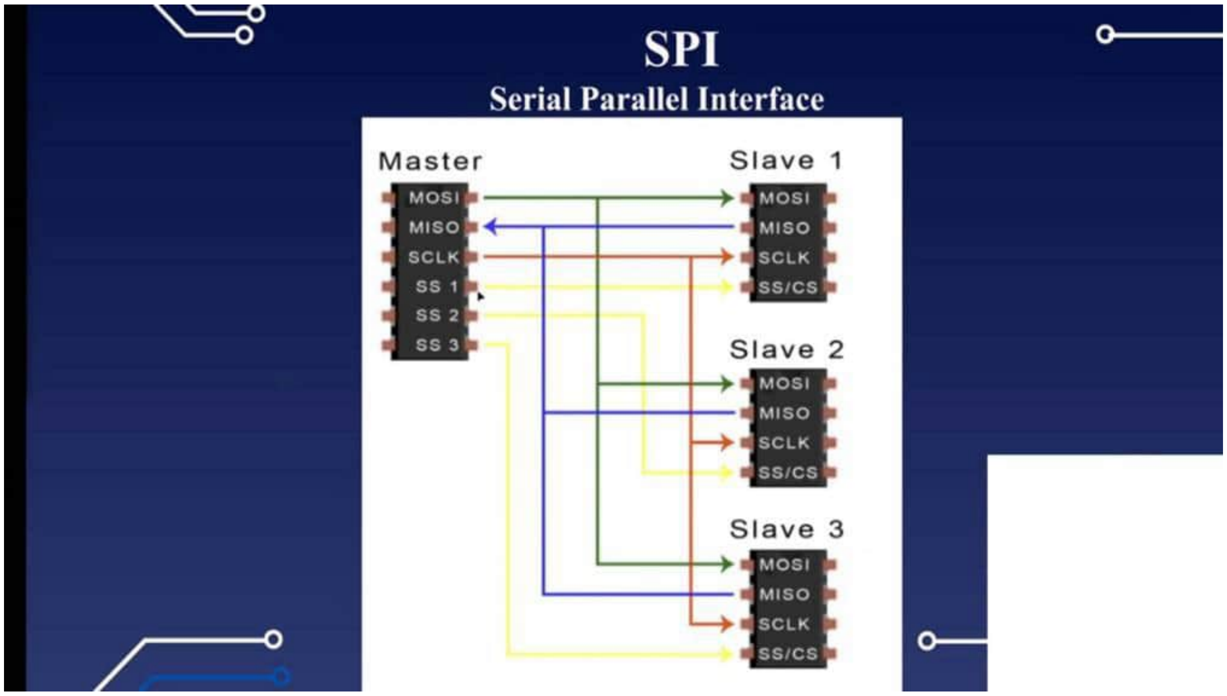
So that means we can transceive and transmit and receive data at the same time simultaneously. And this is used for example, ink displays and other modules were NFC, for example, RFIDM. And Of course, it's a convenient way with this full duplex, but we have to think about, is it really necessary? Because I also have RFID and NFC modules where I use your art, for example. And this is much more convenient when it comes to tiny microprocessors.

So therefore, it's used, but I'm not really a fan of. So this is also the reason why we keep this SPI a little bit shorter in the dynamic. So let's focus here on what the SPI have, on what the GPOs are. First of all, we have a clock SCL. And this is a clock input to synchronize data.

transmission for master device. Same as in the e squared team. So that means if the pulse goes high, then we send the data. going low. This is the space.

And with the next pulse, we can send the next data, and this is how we can set the clock into nanoseconds, for example, and can transmit data very fastly. The mostly stands for master out and slave in. So we are transmitting data. and the meansu stands for master in slave out for receiving data. And this is because it's in a separate line, full duplex.

Then we have slaves selected to activate and deactivate certain units and communicate only with a certain number of units because we also can use this as some parallel bus system. So that means I can use several slave devices here for 1, master. And I have to change the SS or CS pin here. Of course, if I have 2 slaves, I have to use 2 different SS or CS pins. So this is a variable pin often, but the clock meso and mostly are predefined in the outer, you know, or ESP, for example.

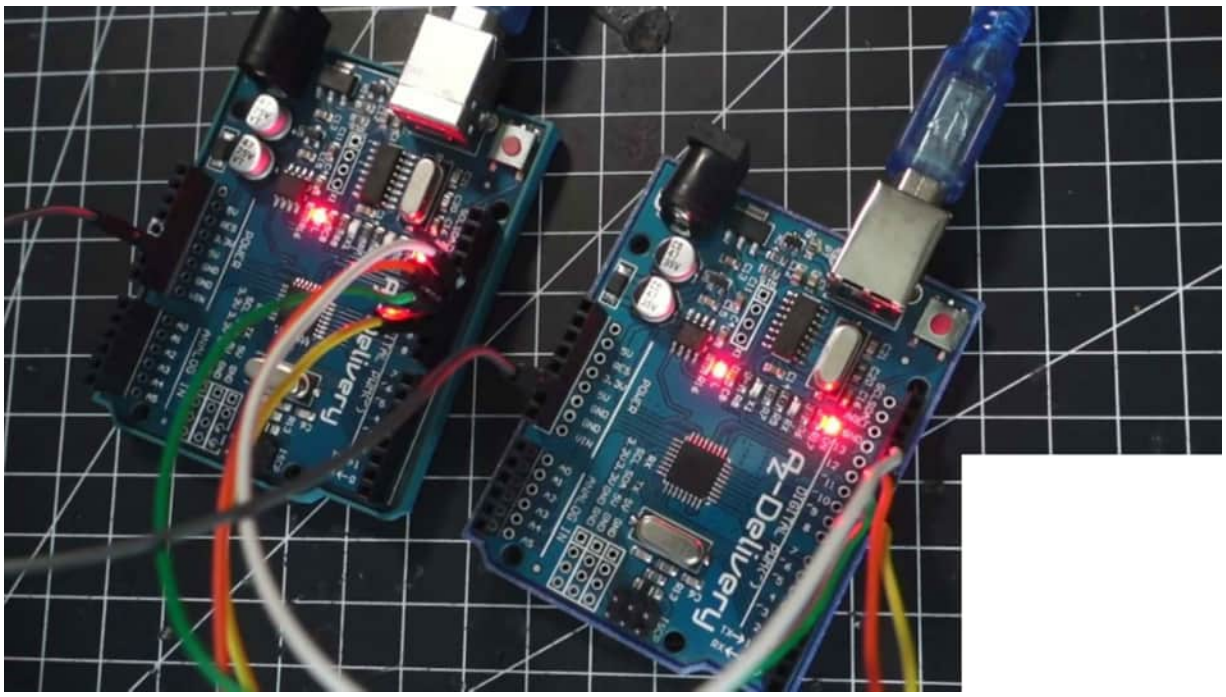


Yes. And this is how on such a parallel interface you could set up. As you can see here, we have mostly mesone here. connected in a parallel way, and then we have 3 in the master different slaves selected. So then I can say, I would like now to communicate with slave 1 when I'm putting high, for example, to SS 1. And as you can see in parallel devices, many different MOSIS, SPI devices can talk to 1 master. Yes. It is and where it's a I would say it's a the SBI has his justification it's widely used, but as you can see, you need a lot of wiring, therefore, So next project, we take a closer look on what we can do with the SBI.

SKETCH ARDUINO UNO TO UNO

In this project now, we want to make our own SPI environment. And therefore, I have FEM 21 here. Audio clones which should communicate to each other with our over SPI. And the connection is really simple here. So we are using the predefined GPS, what you can find here on your reference, your pinout from the manufacturer, And normally, it should be, on the pin GPS open 13, the clock pin, 12 should be a meso on.

11 should be mostly. And 10 probably you could set up the SS pin for example. but there's always checking your pinout. I've mentioned here in the pins also in the codes. So what I did is I connected all of them together, that means 10 to 10, 11 to 11, 12 to 12, and 13 to 13.



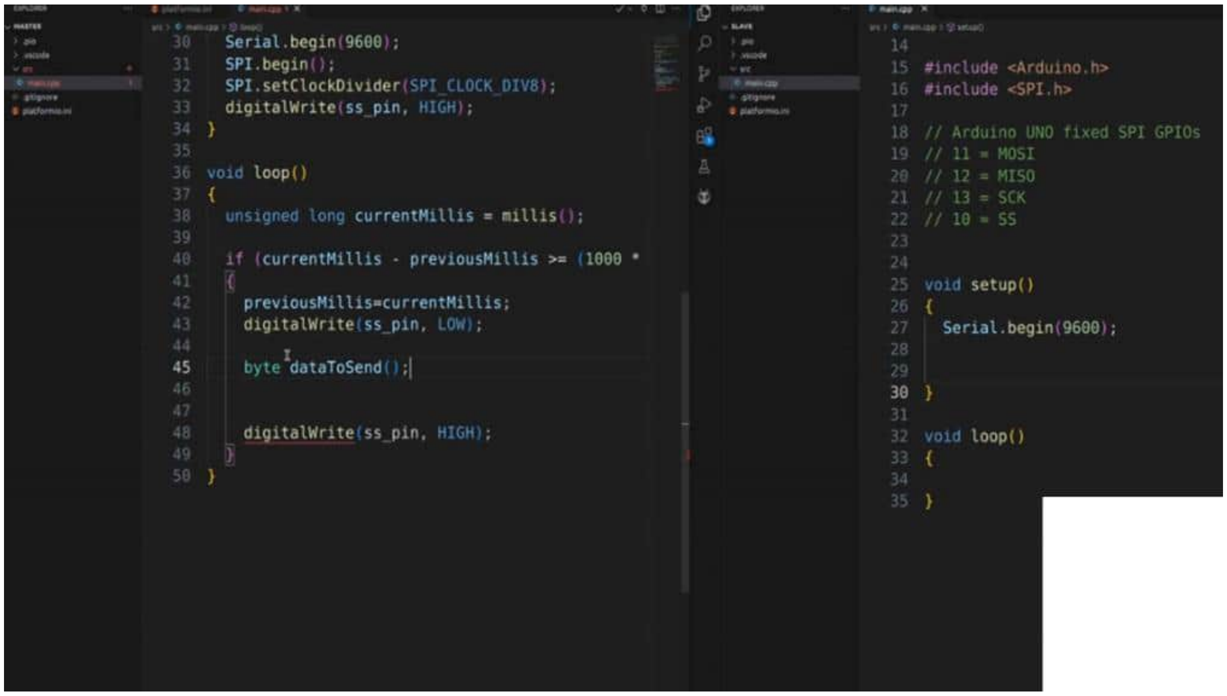
And that's it for our first catch. And afterwards, we also add, here, some sensors, but now we just want to transmit here one data that we can see if the communication process is working in the right way. So I have here the master on the left side and the slave on the right side. So that means the master sensing command, and we want to get your data from the slave, for example. And therefore, I would say that we start with the master.

And the first thing, what we are doing here is we are setting the SS pin. So we say const integer, for example, SSpin is on the pin 10 because this could be variable. And then we have to include. This is also one thing. in the standard repertoire in the core from the Ardeno is the SPI library already implemented also in the USB environment So, therefore, just include it.

And then we have access here. For example, to sbi. Begin. Then in the next line in the master, we are setting here some speed variables. That means we can say here, SPI set clock divider, and you can search about this term.

You find different kinds of explanations here. What we are doing here, we are setting the clock so you can see a different kind of dividers by 8. This will be probably the best speed setting for our purpose now. Then we are setting, the digi a digital write, and we are setting up the SS pin. too high because this is the normal default value.

And when we want to start the communication, We have to set it to low and then make the communication process, and then we are setting it again too high. This is also the first step of what we are doing now in the loop. So that means inside those 2 digital rights, we have to make our communication process. So we are setting ourselves low. And afterwards to hire, this is where the communication can start.



```
  Serial.begin(9600);
  SPI.begin();
  SPI.setClockDivider(SPI_CLOCK_DIV8);
  digitalWrite(ss_pin, HIGH);
}

void loop()
{
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000 *
  {
    previousMillis=currentMillis;
    digitalWrite(ss_pin, LOW);

    byte dataToSend();


    digitalWrite(ss_pin, HIGH);
  }
}
```

```

#include <Arduino.h>
#include <SPI.h>

// Arduino UNO fixed SPI GPIOs
// 11 = MOSI
// 12 = MISO
// 13 = SCK
// 10 = SS


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


}

void loop()
{

}
```

So what we are doing now, we want to send, for example, one to buy it. And despite being able to have the following information, a byte data to send, for example, and we are sending here, yeah, let's send here, for example, this value. Then spedot transfer and we are transferring the bytes, which we just initialized. And then we could say digital, digital. Alright.

Because I also want to see what we are printing out. SSIN. It's digital. Yes. That's what we want to see here, the data to send.

in hex, and I also would like to add here and serial print, not done. Print line, and here we say serial print. We could say there were no leaks. So it just looks very promising, then we try to upload this code. Very good.

And our first test was successful. That means we are sending out every 2 seconds here this value, but no one can really get it so our slave can't be able to receive it because we have to program it. Therefore, jumping to our slave, we have to do nearly the same. That means we have to define or include the SBI, and we have to define here their SPIN as well. Then we are starting with the SPI.

And we are making a pinmode here because, here, we haven't really defined it any. So that means it's an input pull up. We don't need this in the master, and now we would like to configure the SBI settings. in the slave. So, therefore, we say SPI set. Begin transaction.

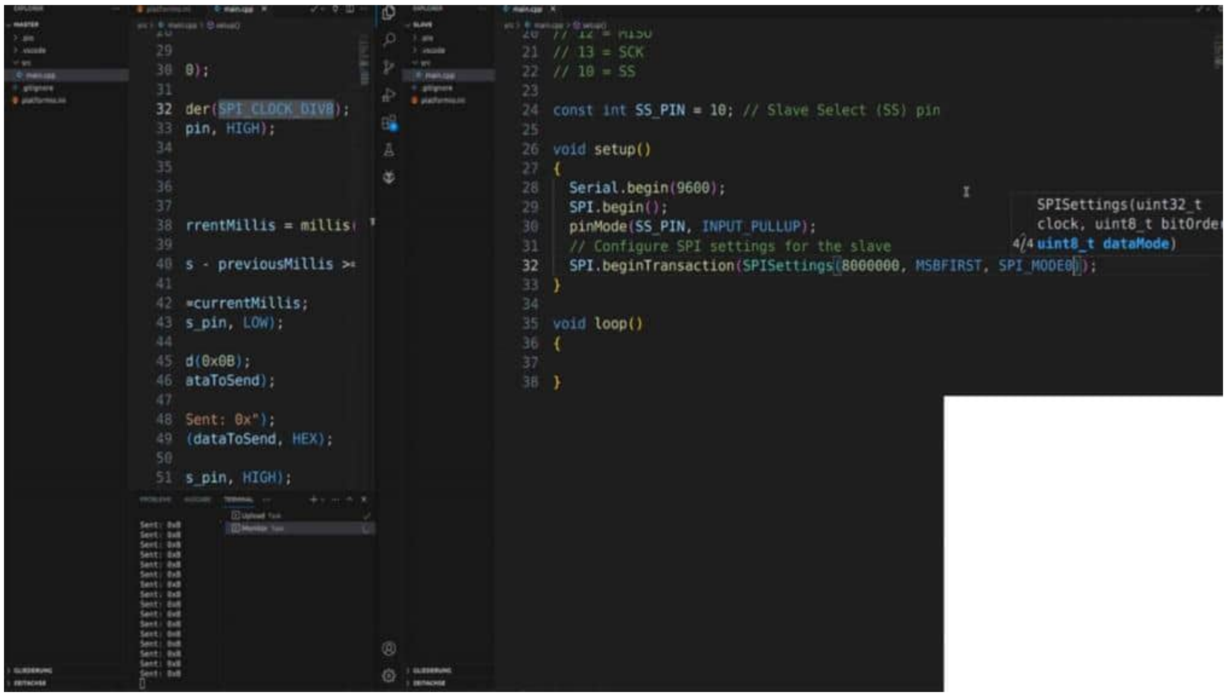
And now we have to fill in our SPI. I think it's called settings. And inside this function, we have to hand over some values here. First of all, I would like to have 8 Megahertz here, and this should be in accordance with where the SP clock divider is? Then we are seeing the MS B should be transmitted first, and I have E and SPI mode.

And this is also where we have different kinds of modes where, in this mode, the slave selection should be lower when we are communicating. This could be changed by SBE mode 1. I think then it's, we're communicating by high, otherwise, by low. So we can change the direction here as Okay. And that means in the loop, we say if, for example, the digit term, read of the SS pin, and this is the reason why we set the pin mode.

If this is low, then the communication could start. And then we say, for example, we are receiving here the data from the master. So byte, receive data equals to, spe.tran sphere 0 because we are sending here back Justin 0, and it's in full duplex. That means we can receive something here and send back data. But I would like only to receive the data here.

Nothing sent back. But otherwise, we could also see here, for example, here, we could receive the data as well. So then, we want to print out the data, which we have just received. So received 0x, and we're saying 0 print line, receive data in hex. I have already changed the USB settings here.

So this should be here on USB 0, and this one should be run on ESP 1. Then let's see if we have some first connection here. Then let's see. Open up the terminal again. And there we go.



```
0);

der(SPI_CLOCK_DIV8);
pin, HIGH);




rrentMillis = millis(

s - previousMillis >=

=currentMillis;
s_pin, LOW);

d(0x0B);
ataToSend);

Sent: 0x");
(dataToSend, HEX);

s_pin, HIGH);
```

```
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
Sent: 0x8
```

```
// 12 = MISO
// 13 = SCK
// 10 = SS

const int SS_PIN = 10; // Slave Select (SS) pin

void setup()
{
  Serial.begin(9600);
  SPI.begin();
  pinMode(SS_PIN, INPUT_PULLUP);
  // Configure SPI settings for the slave
  SPI.beginTransaction(SPISettings(8000000, MSBFIRST, SPI_MODE0));
}

void loop()
{

}
```

We're already getting some data here. Okay. It Looks like it works perfectly. So that means we are now able to send here data in bytes level via SPE from one device to the other device. And we also can send back some information here as well.

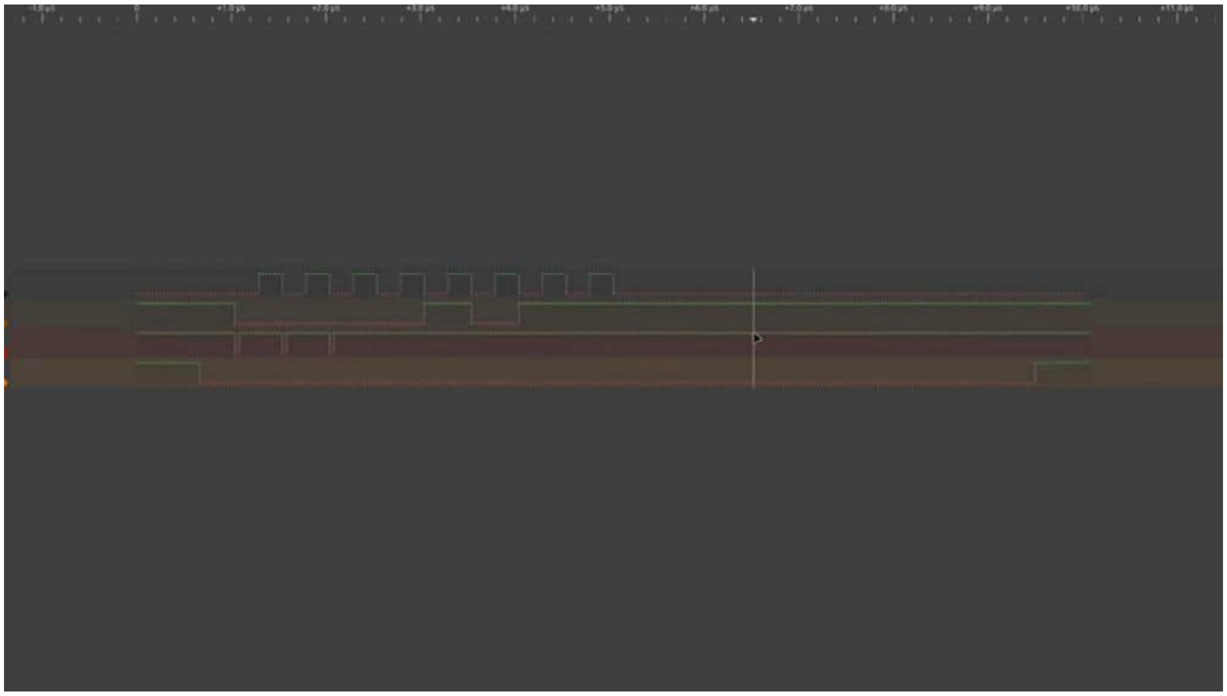
We just have to grab it here. and catch it, in this line, for example, and then we have here in full duplex. Way. One of the next projects we're analyzing this data communication on the logic analyzer. And then we also want to send you some real life data, for example, from a temperature sensor.

ANALYSIS WITH LOGIC ANALYZER

And now let's focus on the PulseView part so that we can understand the SPI protocol a little bit more in detail. Therefore I've already inserted the clamps on the Arduino UNO clone. So I've just pushed up the jumper wire a little bit so that I get the clamps inside and also that the data communication is working reliably. And now I've already started communicating here. I've added 10 GHz here and also in the sample speed I've changed to 24 MHz. And this is what we got here on these four channels.

Now let's focus here on the data frame that we captured here. And it's pretty obvious what kind of channel we have here in our SPI communication. So the D0 channel has obviously the clock values here. So you can see here the same width for low and high. So let's focus here and measure what is the time frame from one of these values. And we have here 250 ns which is equivalent to 4 MHz. And this is pretty fast what we can do here and transmit and receive data. On our D1 channel we now have the data that we are transmitted.

The D2 channel is the receiving line and this is some serial noise that I captured here because we do not really receive any data here. So I'm not really evaluating this channel. The D3 channel is our chip select. This is what we are set to low in our Arduino sketch so that we know what kind of module we are communicating with. And now let's focus on this one. This is our transmitted data. And when we are focusing in here we can see our signal is starting on a falling edge. But this is not really interesting for us.



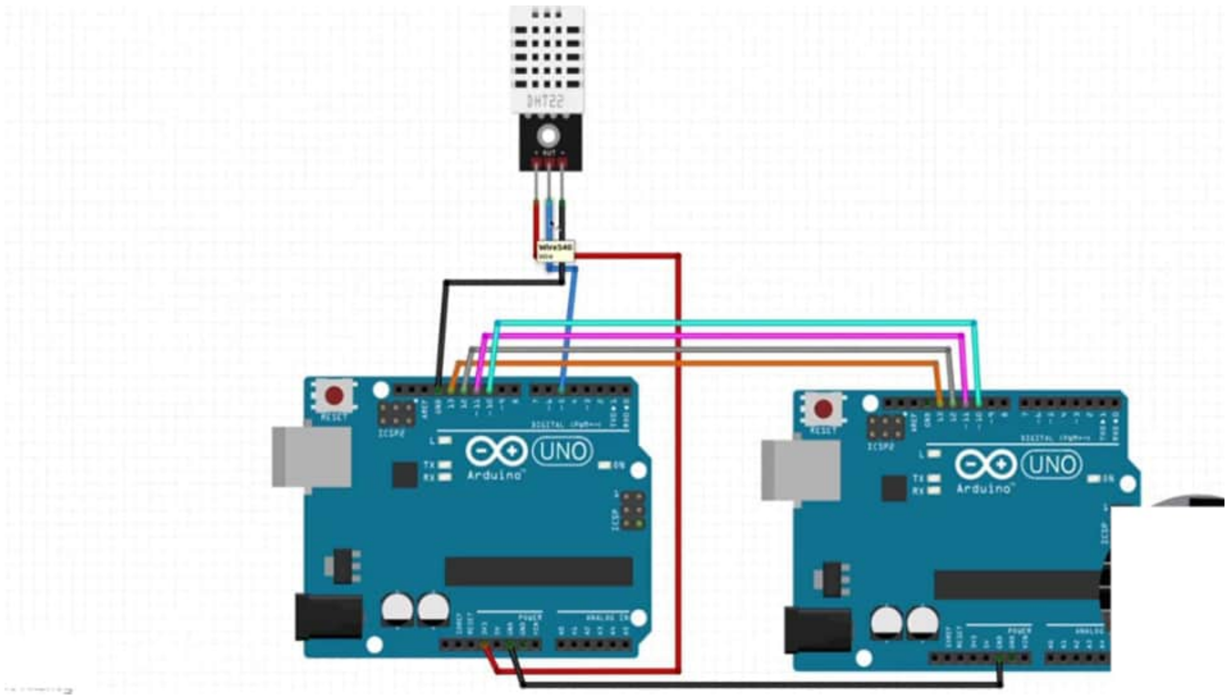
Interesting is the rising edge of the clock and this is where we can say here we have a logical 1. Here we have the next rising edge and here we have a logical 0. And then we have here a logical 1 and a logical 1. So in summary we have now when we are changed to binary 1011. This will be in decimal the number 11 or in hex what we send B. And this is exactly what we did before in our Arduino sketch. So this is just the poor raw data but we can also use some external decoders here.

So just click here on the decoder and then we are searching SPI. Then we are adding the SPI decoder by double clicking on it. Click on the settings and here we are referencing the different kinds of channels. We have the clock on D0. D1 is our transmitted data that means this is master in and slave out. This is not what we want because here we are still receiving data lines. We have the master out and slave in. This is the D1. And then we have the chip selected. This is the D3 line.

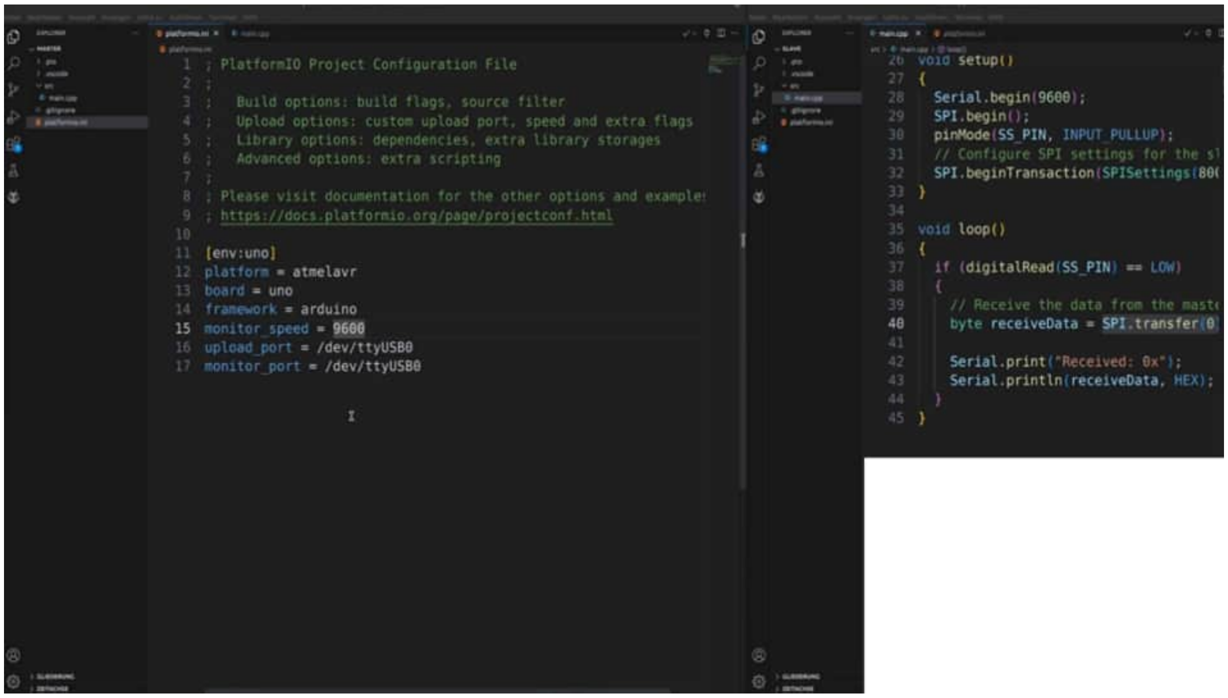
And this is now what we have before made on our own with evaluating all of these values. And as you can see here we have the bits and the data that we transfer with the decoder. So a simple and effective protocol. As you can see very very fast and we have a full duplex here. That means we can send and receive and transmit data at the same time simultaneously.

DHT22 VALUES VIA SPI

Now one of our last examples will be that we want to send real life data from a DHT22 for example via SPI. So the wiring part is very easy in this case we are just connecting the DHT22 to for example 3.3 volts ground to ground and the output goes to 5 to the GPIO5 for example and the connection what we had before keeps the same 30 to 13, 12 to 12, 11 to 11 and the chip select is on 10. So of course you can also use another sensor and if you're using not the breakout board don't forget to use here a pull-up resistor for the data line.



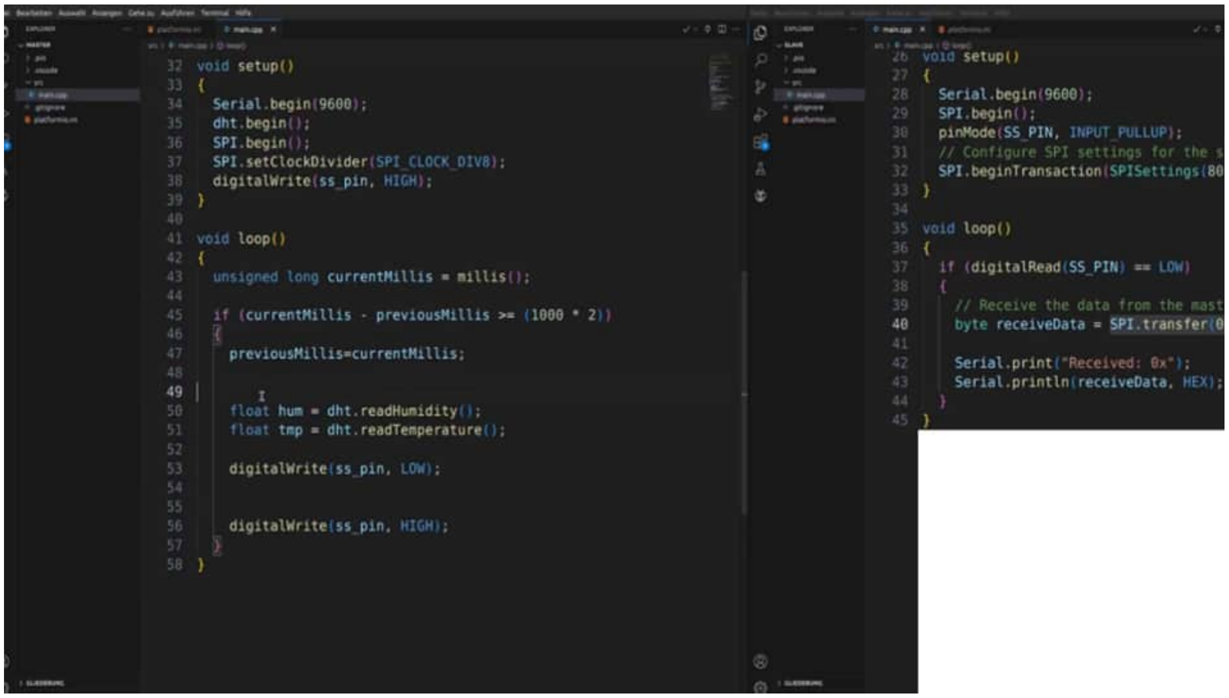
Okay so our existing sketch was here. Here on the left side we have the slave and I've already had the sensor. So what we are doing now is we want to adapt the first sketch from the master. So make it a little bit bigger here. Closing the serial monitors for now. So let's see this is just everything the same but what I would like to do here is we have to add the library for the DHT sensor. So the first step will be adding the DHT sensor library and now we could add here all the things that we need.



```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and example
; https://docs.platformio.org/page/projectconf.html

[env:uno]
platform = atmelavr
board = uno
framework = arduino
monitor_speed = 9600
upload_port = /dev/ttyUSB0
monitor_port = /dev/ttyUSB0
```

```
void setup()
{
  Serial.begin(9600);
  SPI.begin();
  pinMode(SS_PIN, INPUT_PULLUP);
  // Configure SPI settings for the sl
  SPI.beginTransaction(SPISettings(80(
}

void loop()
{
  if (digitalRead(SS_PIN) == LOW)
  {
    // Receive the data from the maste
    byte receiveData = SPI.transfer(0
    
    Serial.print("Received: 0x");
    Serial.println(receiveData, HEX);
  }
}
```

I'm just copy-pasting it because we already did such an example with the non-controversial or non-standard protocols including here the DHT.h library. So nothing special until now. In the setup don't forget to also start here the DHT sensor and now in the loop part we are getting rid of this kind of text and we are making a float which is DHT read humidity. It's a function then we have float temperature. I call it temp DHT read temperature and then I should write the float in the correct way and then yeah looking good so far.

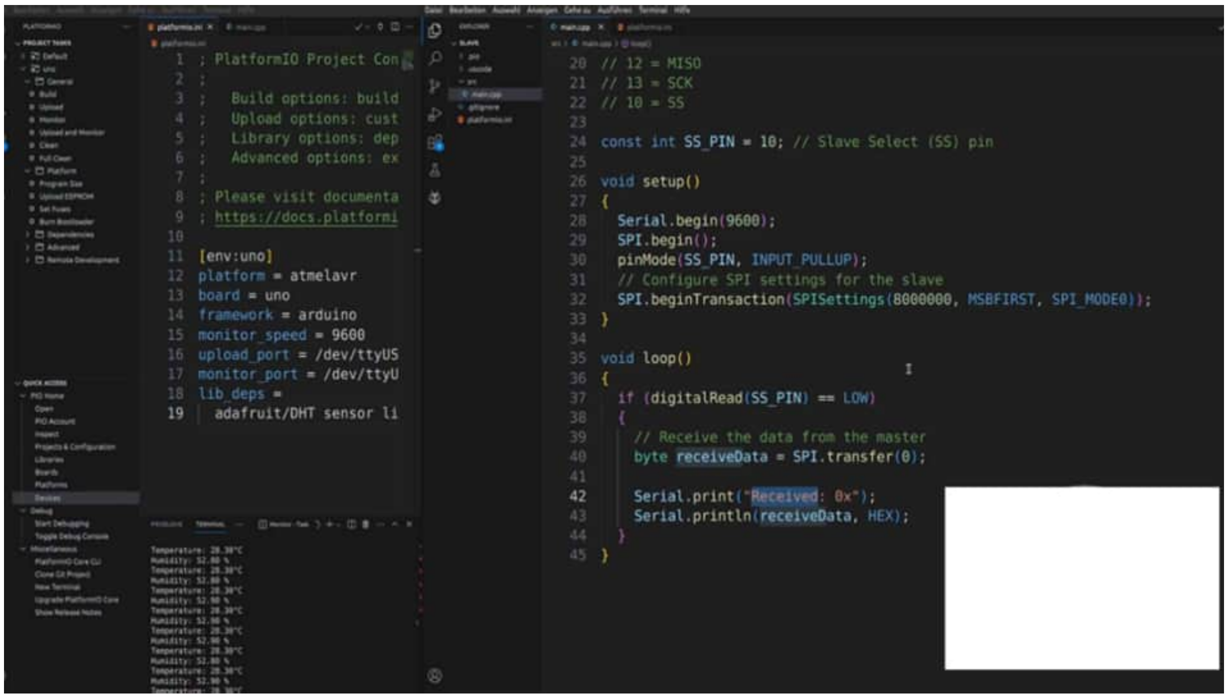


```
void setup()
{
  Serial.begin(9600);
  dht.begin();
  SPI.begin();
  SPI.setClockDivider(SPI_CLOCK_DIV8);
  digitalWrite(ss_pin, HIGH);
}

void loop()
{
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000 * 2))
  {
    previousMillis=currentMillis;


    float hum = dht.readHumidity();
    float tmp = dht.readTemperature();

    digitalWrite(ss_pin, LOW);


    digitalWrite(ss_pin, HIGH);
  }
}
```

```
void setup()
{
  Serial.begin(9600);
  SPI.begin();
  pinMode(SS_PIN, INPUT_PULLUP);
  // Configure SPI settings for the s
  SPI.beginTransaction(SPISettings(80
}

void loop()
{
  if (digitalRead(SS_PIN) == LOW)
  {
    // Receive the data from the mast
    byte receiveData = SPI.transfer(0

    Serial.print("Received: 0x");
    Serial.println(receiveData, HEX);
  }
}
```

We are setting up the digital write pin only when we are really transmitting data, not when we are reading the sensor value and now we could say the SPI transfer button transfers the temperature for example. And that's it for the communication process and I would like to print out the data here so that we can see if everything works right. One thing is missing: we have also included here the Adafruit sensor.h library as well. This is not really included here and now we can upload the code.

So the first test of the master looks promising and then we can switch to the slave. Here don't forget to adapt the USB port so when we want to have two serial monitors at the same time we have to use different kinds of USBs. So in the slave we do not really need to change here so much the only thing what we have to change here is DHT22 temp value and this shouldn't be in hex this should be in decimal and that's it we can upload it and when we did everything right then we get here back our values as we can see here.



```
; PlatformIO Project Con
;
;   Build options: build
;   Upload options: cust
;   Library options: dep
;   Advanced options: ex
;
; Please visit documenta
; https://docs.platformi

[env:uno]
platform = atmelavr
board = uno
framework = arduino
monitor_speed = 9600
upload_port = /dev/ttyUS
monitor_port = /dev/ttyU
lib_deps =
  adafruit/DHT sensor li
```

```
Temperature: 28.30°C
Humidity: 52.80 %
Temperature: 28.30°C
Humidity: 52.80 %
Temperature: 28.30°C
Humidity: 52.90 %
Temperature: 28.30°C
Humidity: 52.90 %
Temperature: 28.30°C
Humidity: 52.90 %
Temperature: 28.30°C
Humidity: 52.80 %
Temperature: 28.30°C
Humidity: 52.90 %
Temperature: 28.30°C
```

```
// 12 = MISO
// 13 = SCK
// 10 = SS

const int SS_PIN = 10; // Slave Select (SS) pin

void setup()
{
  Serial.begin(9600);
  SPI.begin();
  pinMode(SS_PIN, INPUT_PULLUP);
  // Configure SPI settings for the slave
  SPI.beginTransaction(SPISettings(8000000, MSBFIRST, SPI_MODE0));
}

void loop()
{
  if (digitalRead(SS_PIN) == LOW)
  {
    // Receive the data from the master
    byte receiveData = SPI.transfer(0);

    Serial.print("Received: 0x");
    Serial.println(receiveData, HEX);
  }
}
```

We have here made a byte that's the reason why we only get here not on float etc but as you can see we have here received successful values from left to right from master to slave. If you would like to also have the comma then we maybe could send here a second value first is the high byte low byte for example this is also possible. But this should just visualize how easy it is also to use the SPI for your own purpose for your own project to transmit data.

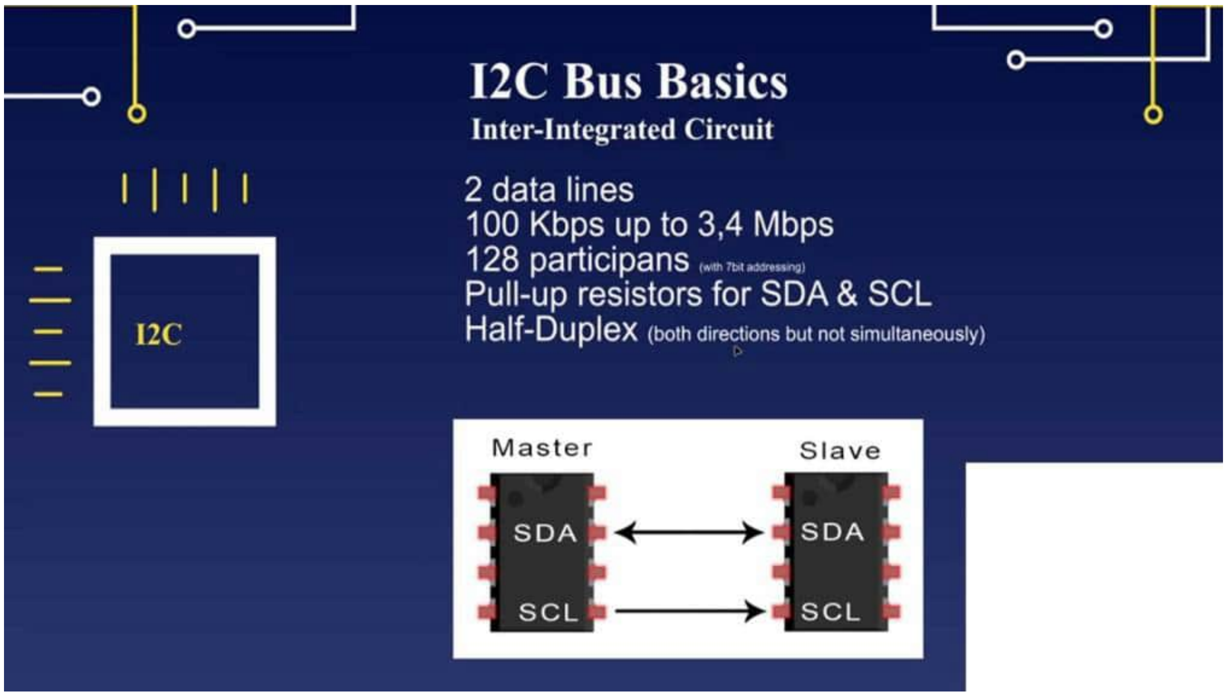
I2C INTRODUCTION

e squared c bus basics. e squared c stands for inter integrated circuit. And the e squared c also, in my opinion, one of the populous serial communication protocols in the Adelino environment. Why is that so? Because it's easy to use, and it's a bus system.

That means we have master and slave connections here, which we can see here. And that means that we have 1 master, for example, our audio, or USB, AT, tiny, which communicates to many multiple devices and in the case that each slave has this unique address, we can address, for example, one sensor specifically, then the order and in the next one. And this can be done only by 2 wires. We have an SDA here. This is the data line, and we have a clock line.

And with this clock line, we synchronize the data transmission. And this is one of the main differences between the e squared c and the u r because here, we can transmit faster our data because the clock line just indicates when it comes to next data. And therefore, yeah, we only use 2 data lines. Although we have multiple devices, this is a bus system. So all of them are in parallel.

That is exactly what we do afterwards. So you can see here, 3 devices, all of them, e squared Centimeters, and we can use them because each of them has a unique address. So we can use from, according to the speed from 1 100 kilobits per second up to 3.5 megabits per second. So this is, I would say, in the environment enough for exchanging his sensor data. Also 128 participants 127, booth and 7 bit addressing.



I2C Bus Basics

Inter-Integrated Circuit

2 data lines
100 Kbps up to 3,4 Mbps
128 participans (with 7bit addressing)
Pull-up resistors for SDA & SCL
Half-Duplex (both directions but not simultaneously)

I2C
Master
Slave
SDA
SCL
SDA
SCL

To address, we come later on. We also have the ability to make a 10 bit addressing here, and then we have up to 1000 participants. It all it always, you know, should ask yourself, is this really necessary to have so many participants here to 1 MCO, but maybe this could be the case. We need a polar resistor for SDA and SCL, but this is often done by the internal inputs from the GPIOs. So you can set here in p mode and with this input, pull up, for example, this could be done, and also the libraries do this by demo.

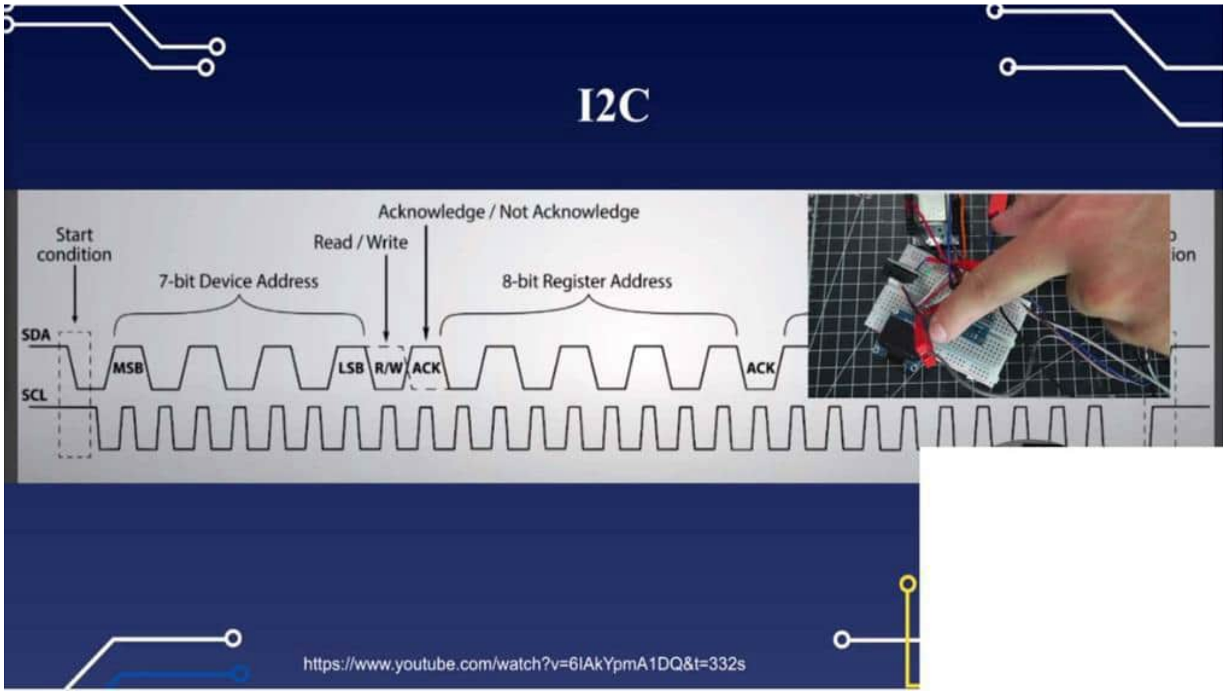
Half duplex means both directions, but not simultaneously. So we send it from the master, for example, address it if we get and acknowledge back, then we know, okay, we can communicate. As you can see here, in both directions we can communicate, but not simultaneously. This is, for example, done by SPI, where we have here a full duplex communication And here, once again, to visualize what e squared c means in the bus system, so we have here the master, and we are connecting the SD a parallel to each component and also the SEL, the clock line, and then we can communicate from our auto, you know, or ESP to all of the devices. And in my case, what we do later on is we have a sensor here in sensor and on OLED and in second sensor.

That means we are getting data from those 2 sensors, and we're sending data to this device. And this could be done all, only by 2 GPOs. Really great. I'm sure a lot of you already used e squared c, but now we whittled dig a little bit deeper into the protocol and also tried to make our own libraries on a bit level so that we have really a deep understanding of how this protocol works.

PROTOCOL IN DETAIL

And because it's so important that we understand the whole flow of this protocol, I would like to take a few seconds to go through some problems. So we start with this protocol by setting up the SDA line at high. If it changed to low, then the communication could start. This is the starting condition. And then the first date that is transmitted from the master is the device address on which I would like to communicate, because in my case, I have 3 devices on my breadboard, and I'm sending, for example, out the address for the OLED display so that I can communicate with the OLED display.

And this is a 7 bit address. And the last one is a read or write indication. If there is any one inside the last bits, then we know the master would only like to read something out here, is there in 0, then the slave knows, okay, there is something to be right on it. And what do we want to write? But before, we got here and acknowledged back.



So if the device is found with this address, I'm getting back an acknowledged bit. And this is necessary to know because in the end of the chapter, we are making our own E squared C scanner. And with this E squared C scanner, we can, and with this flow, we are sending out some addresses and looking if someone is sending us back and acknowledge, so we know, this device is in our circuit. But more to this later. So if we got back this acknowledged, we now have here an 8 bit register address.

And nevertheless, it depends on what we want to do. If we want to write something, for example, we want to send the model. Please start by measuring the temperature, setting the resolution offense. So all we want to transmit is some text, etcetera, we have to know on which address, register address we have to address our data. And this could be done here.

We're getting back and acknowledged, and then the data transmission is started, and this could be many, many bits. And then, as the last one, we're getting back here and acknowledged, or it's inverted. We will also see an inverted acknowledgement because this means now the stop condition is here. And that's it. And you can see here, this is our clock signal.

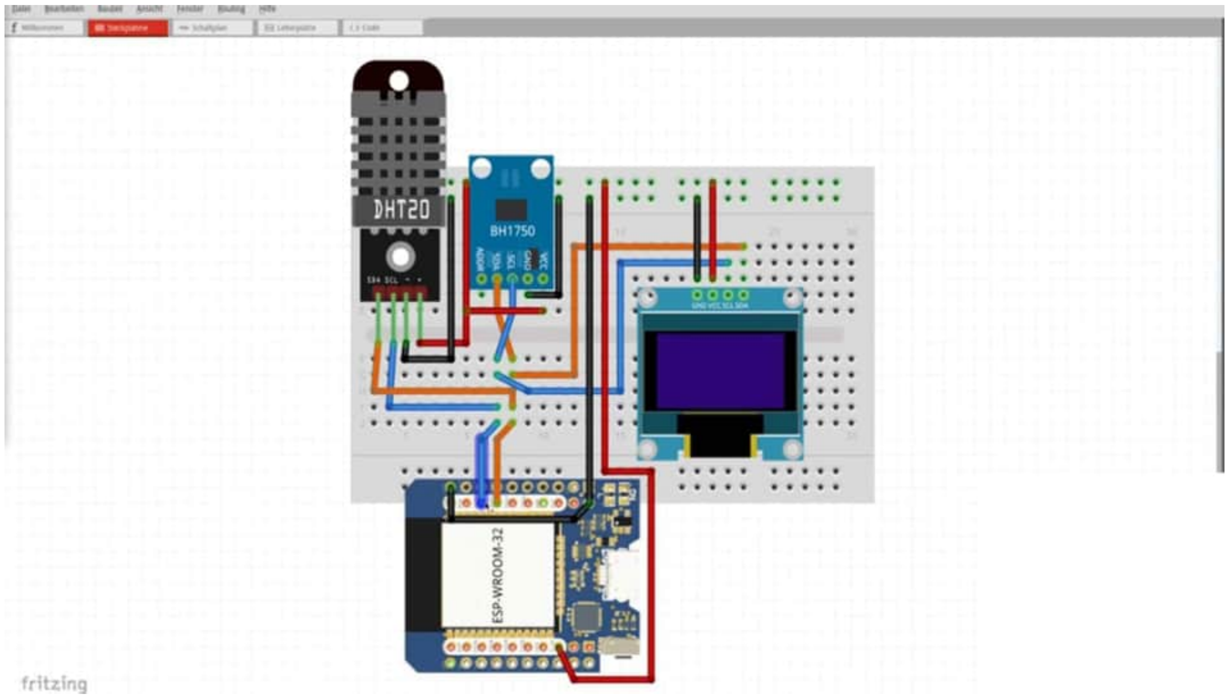
And on each clock pulse, we can transmit here on 1 orange 0. And this is in the basic form, the E squared C protocol, also very easy in the setup, and we will see this by one of the first syncs what we do is this PH175, 0, which it has really an easier, I would say you see strap of the data, and, therefore, this will be the perfect start in our e squared c journey.

SKETCH AND WIRING BH1750

We start with our first example and as I mentioned before I have chosen the BH1750. It's an ambient light sensor which gives us the value of the current lux in the environment. And this is exactly what we want to do because it has a really easy structure to read out. But in a whole general approach we will also use the DHT20 temperature sensor later on. It's a little bit more complex and I think it's a good way to understand the whole process.

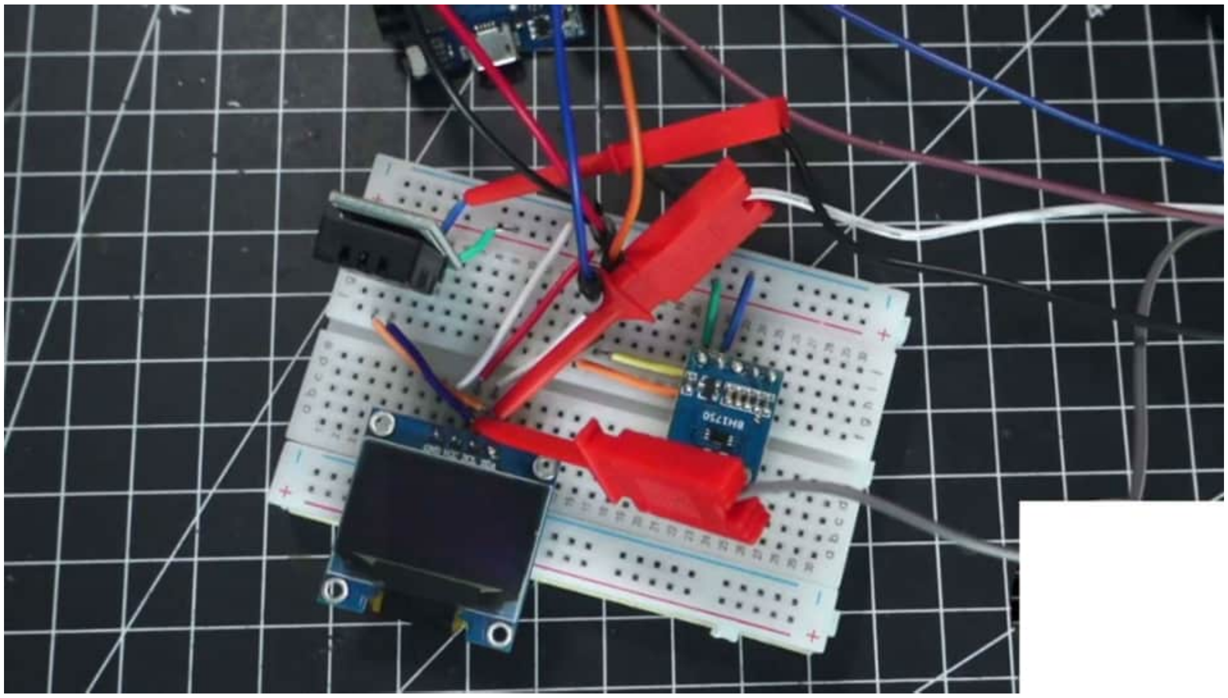
And as well as an OLED display this is just a placeholder for inserting E2C modules because later on we're also writing our own E2C scanner and therefore it's perfectly suited. Yeah and also the whole wiring part is very easy because we connect the 3.3 volts to the breadboard here because be aware I'm not sure we have taken a closer look into the datasheet. I think this is only compatible with 3.3 volts. Then we connect the ground also to the breadboard so that every of this module has sufficient power supply.

And now we're making a connection point on the breadboard where we connect each of the SDA ports to the breadboard as you can see here. And then we are connecting the SDA to the pin. There are some predefined GPIOs and in my case this is the GPIO number 21. So it's on the dev kit on the D1 nearly on every ESP32 the GPIO 21 is the SDA predefined GPIO. But take a look on your datasheet according to your manufacturer because this could be really often also unchanged there. And the clock line we did the same thing and here we have the GPIO 22.



that's it. That's our whole wiring part and in my case this looks like here I have the setting. So the OLED, the DHT20 and the BH17050. And what I did here is I set up the clamps somewhere where I have enough space. In this case I am using the lines of the SDA and SCAE directly from the OLED display because they are all connected together. So it doesn't matter where I use them. And I also use the ground pin. Where's the ground pin? Let me see. Here.

Here I have connected the ground to the logic analyzer because this has to be the same potential. Yeah and that's it. That's it for the wiring part. And now we can switch to our IDE. In my case Platform.io. And the first thing we are doing here is installing the external library for the BH1750. It's from Klos. And now we have here our basic sketch with nothing in it. And we can start now with our program.



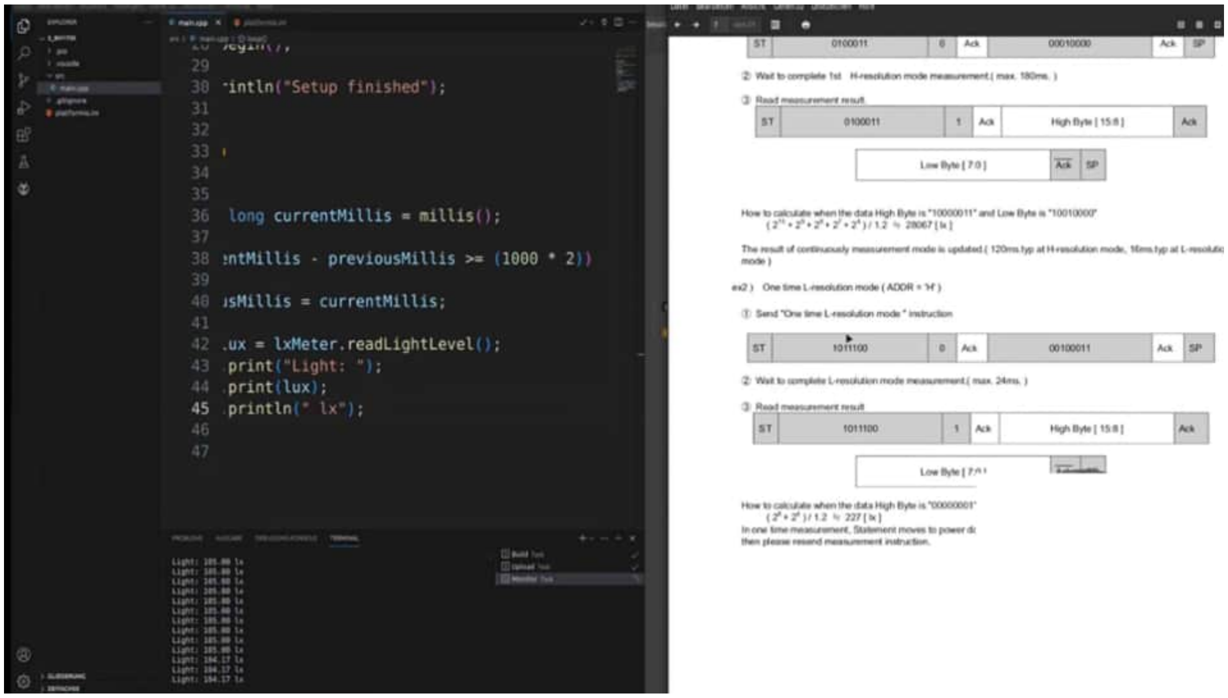
Speaker 1 (00:03:27) - And one of the first things that we do is whenever we are using E2C we include the wire because this is the main part where we can easily connect the devices into the code. So include wire.h and we don't have to use any external libraries here. This is inside the core of the ESP32. That means it is already implemented but we have to include it. Then we are using the include with the BH1750 what we have already included in the Platform.io. So then we can create a new object here. And this is the BH1750. And I call it Luxmeter.

Speaker 1 (00:04:17) - Then we start the first thing in the setup. And this is a little bit different from device to device. But in general we should start with E2C communication. And this could be done by wire.begin. And often the libraries do this by themselves. So if you are not sure then take a look at the example from the library. But normally we should start the wire.begin. And also we could now start according to the library the object also starts here with Lux Meter.begin. Just for us a short note that the setup is finished.

That both of them can establish the connection. And now we could focus on getting data out of the sensor. And therefore we are creating a Luxmeter. This is our object.readLightLevel. That's it. And then we can print out here the current Lux. There we go. Uploading the whole stuff. And there we go. We got here at our current Lux level. And the experts will now say 120 Lux is way too less. Normally we have here on an office desk beginning at 500 Lux. So therefore just check if I'm holding the hand above. Yes, it's reacting. So I think the values are accurate enough for this case. And as you can see we got the first levels out of it. And this is the precondition for the analysis which we do in the next project.

ANALYSIS BH1750

Now it's time to investigate the whole communication process via E2C a little bit more in detail and therefore I have opened up the datasheet for the BH175 and this is already the sketch is already running as you can see here we have 105 lux in this time so in the next step we would like to capture the data and also compare it with the datasheet so this is our main goal now and therefore I am here on the page what is it page seven and we can see here all of the necessary information what we need but before I have to jump a little bit up because one question is open is the VCC voltage as we mentioned before we are connected all of the supplies via 3.3 volts and this is necessary because this module only is able to get here maximum 3.6 volts so do not use 5 volts it will damage your device here are a lot of other specs what you can see also this is interesting so the spectrum where the light can fall on the sensor and where is working effectively but we want to go a little bit down and want to see here the resolution and what is also interesting is let me see so there we go here we have the communication process so first we send from the master here our address I think this should be the E2C address so let's check it it's called zero one zero zero zero one one and this is 35 and this is the E2C address from this sensor so we are sending out the E2C address if we're getting back and acknowledged then we know we can start and then read the measurement results this is what we want and this is what we should capture so this is the start bit then we're sending out once again the address from the sensor getting back and acknowledge here's a read statement as you see seven bit address one read bit acknowledged and here we got the high and the low byte and inverted acknowledged and then stop it and this is our data what we should now get on our pulse view because this one is continuously resolution mode don't use that so therefore let's check if we see something so I've already connected the clamps and also set up here two channels channel one and two or zero and one I changed here to two gigs and we can change here 60 megahertz and then let's go so we see here the data comes here in and what I would like to do now is once again open up the connection so that we get here the same result click on stop and this one should now our desired value here we go so probably you know what is the clock line and what is the data line we can see perfectly matching here the clock line so adding here and decoder e squared c double click on it then we can change here the clock line is d1 in my case and sda is d0 and.



```
intln("Setup finished");



 long currentMillis = millis();

ntMillis - previousMillis >= (1000 * 2))

sMillis = currentMillis;

.ux = lxMeter.readLightLevel();
.print("Light: ");
.print(lux);
.println(" lx");
```

```
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 185.00 lx
Light: 184.17 lx
Light: 184.17 lx
Light: 184.17 lx
```

ST	0100011	0	Ack	00010000	Ack	SP

② Wait to complete 1st H-resolution mode measurement.(max. 180ms.)

③ Read measurement result

ST	0100011	1	Ack	High Byte [15:8]	Ack

Low Byte [7:0]	$\overline{\text{Ack}}$	SP

How to calculate when the data High Byte is "10000011" and Low Byte is "10010000"
$(2^{15} + 2^{9} + 2^{8} + 2^{7} + 2^{4}) / 1.2 \fallingdotseq 28067$ [lx]

The result of continuously measurement mode is updated.(120ms.typ at H-resolution mode, 16ms.typ at L-resolutio mode)

ex2) One time L-resolution mode (ADDR = 'H')

① Send "One time L-resolution mode " instruction

ST	1011100	0	Ack	00100011	Ack	SP

② Wait to complete L-resolution mode measurement.(max. 24ms.)

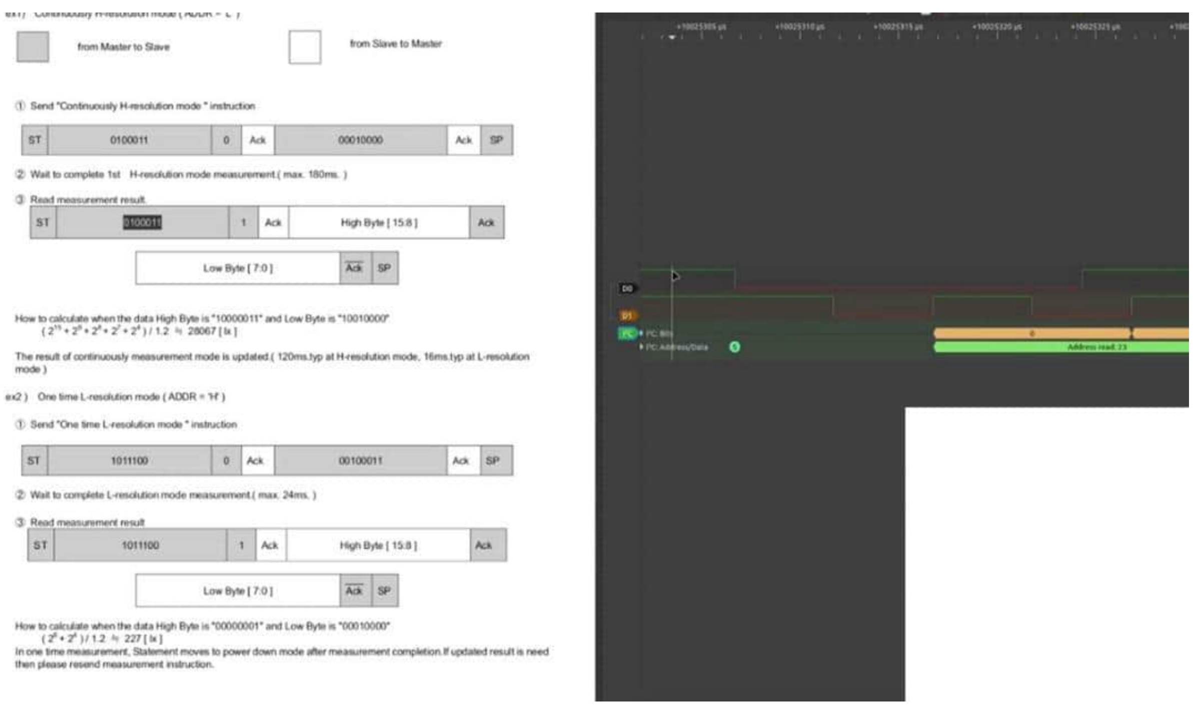
③ Read measurement result

ST	1011100	1	Ack	High Byte [15:8]	Ack

Low Byte [7:0]

How to calculate when the data High Byte is "00000001"
$(2^{8} + 2^{5}) / 1.2 \fallingdotseq 227$ [lx]
In one time measurement, Statement moves to power d
then please resend measurement instruction.

We don't need any further information because we do it all by one. So on the left side here, we have our measurement. So as we can see here, we have our start bit When the signal goes from high to low This indicates that we are starting and this is our Address and I said before 35.



ex1) Continuously H-resolution mode (ADDR = 'L')

▨	from Master to Slave	☐	from Slave to Master

① Send "Continuously H-resolution mode " instruction

ST	0100011	0	Ack	00010000	Ack	SP

② Wait to complete 1st H-resolution mode measurement.(max. 180ms.)

③ Read measurement result.

ST	0100011	1	Ack	High Byte [15:8]	Ack

Low Byte [7:0]	$\overline{\text{Ack}}$	SP

How to calculate when the data High Byte is "10000011" and Low Byte is "10010000"
$(2^{15} + 2^9 + 2^8 + 2^7 + 2^4) / 1.2 \fallingdotseq 28067$ [lx]

The result of continuously measurement mode is updated.(120ms.typ at H-resolution mode, 16ms.typ at L-resolution mode)

ex2) One time L-resolution mode (ADDR = 'H')

① Send "One time L-resolution mode " instruction

ST	1011100	0	Ack	00100011	Ack	SP

② Wait to complete L-resolution mode measurement.(max. 24ms.)

③ Read measurement result

ST	1011100	1	Ack	High Byte [15:8]	Ack

Low Byte [7:0]	$\overline{\text{Ack}}$	SP

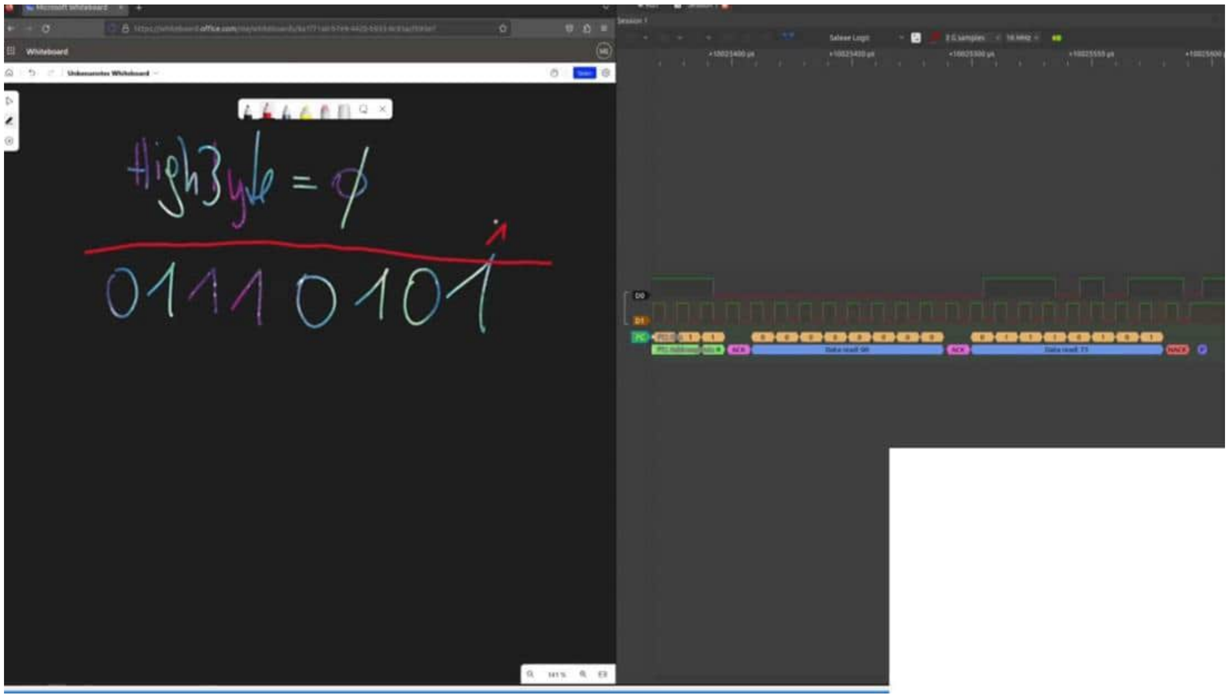
How to calculate when the data High Byte is "00000001" and Low Byte is "00010000"
$(2^8 + 2^4) / 1.2 \fallingdotseq 227$ [lx]
In one time measurement, Statement moves to power down mode after measurement completion.If updated result is need then please resend measurement instruction.

This is in decimal But normally you give the E squared C address in hex and this is 23 So this 7-bit indicates the address and now the 8-bit shows me that the master will Read now.

Here we are Then we're getting back in acknowledge that means the communication from my ESP to the module was successful and Then we got in getting here two Bytes and this is the high byte and this is the low byte and as we can see here nothing is in the high byte because normally when we have here and higher brightness level and we can Simulate it lay down with some flashlight.

Then we will also have here some data in it But now I would like to get and check if we get the value of from 97.50 Acknowledge and then we have a low byte And how can we calculate it to calculate when the high bit is and the low bit is so we are taking Both together. This is why we need later on also the bit shifting and the or operation and then we Divided by 1.2. This is necessary and then we get the approximate Lux level So, let's see.

Yeah, we open up our Board so that we can practice here a little bit more to calculate with all of these data So therefore we have on our high byte our high byte Is Zero we have nothing to do here and then our low byte is 0 1 1 1 1 1 1 Then we have 0 1 0 and 1 then we have here and not acknowledged. This is Also indicated here not acknowledged and a stop it so that we know.



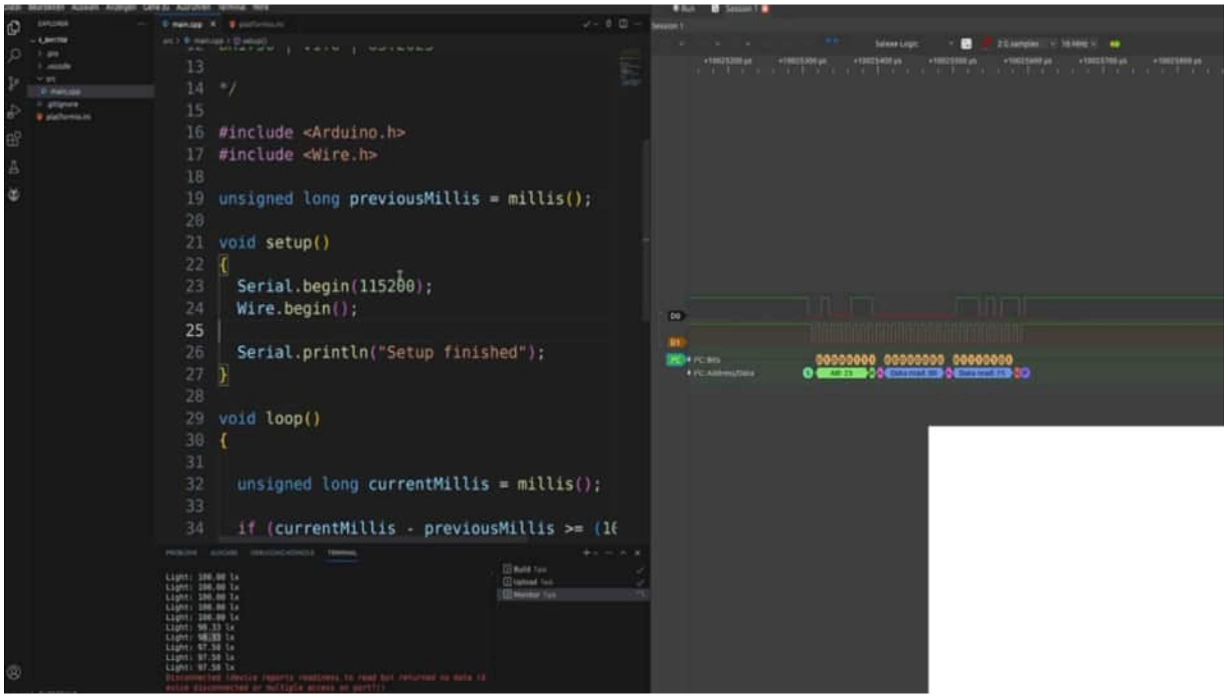
Okay, the whole transferring process is now over so what we can do is here Let's practice this a little bit 1 2 4 8 16 32 64 and now we can add here 64 plus 32 plus 16 plus 4 plus 1 and now a little bit of calculation 6 4 plus 32 plus 16 plus 4 plus 1 is Is 118 that have to be divided by 1.2 and This looks promising. I would say divided by 1.2 and this will result in 98 Lux So, let's see. What was our result? 98 and where is the visual studio code? 97 98 point three three.

So this is the last point what we fetched here in the Pulse view and what we got here. So 79.5 wasn't captured here with the logic analyzer, but 98.33 we get and this is Exactly what we wanted to know so this is the first inspection of the protocol and In the next project we are trying to get this data out of the ESP without any external library.

SKETCH BH1750 WITHOUT EXTERNAL LIBRARY

Are you ready for some code action? And if so, then we get rid of all of the things that we did before in our code, except the wire begin, but we get rid of the BH17050. And also here in the platform IO, we can get rid of it, because we just need the wire. The wire is necessary because it helps us, it's that we can communicate with the E2C modules in an easy way, because we don't have to reinvent the wheel by its own.

But it's easier to use here directly the wire.h and not using an external library when it comes to, for example, using an ATtiny and ATtiny where we have limited space or some other tiny microcontroller. And with a few lines of code, we will get the message here.



```
*/

#include <Arduino.h>
#include <Wire.h>

unsigned long previousMillis = millis();

void setup()
{
  Serial.begin(115200);
  Wire.begin();

  Serial.println("Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1(
```

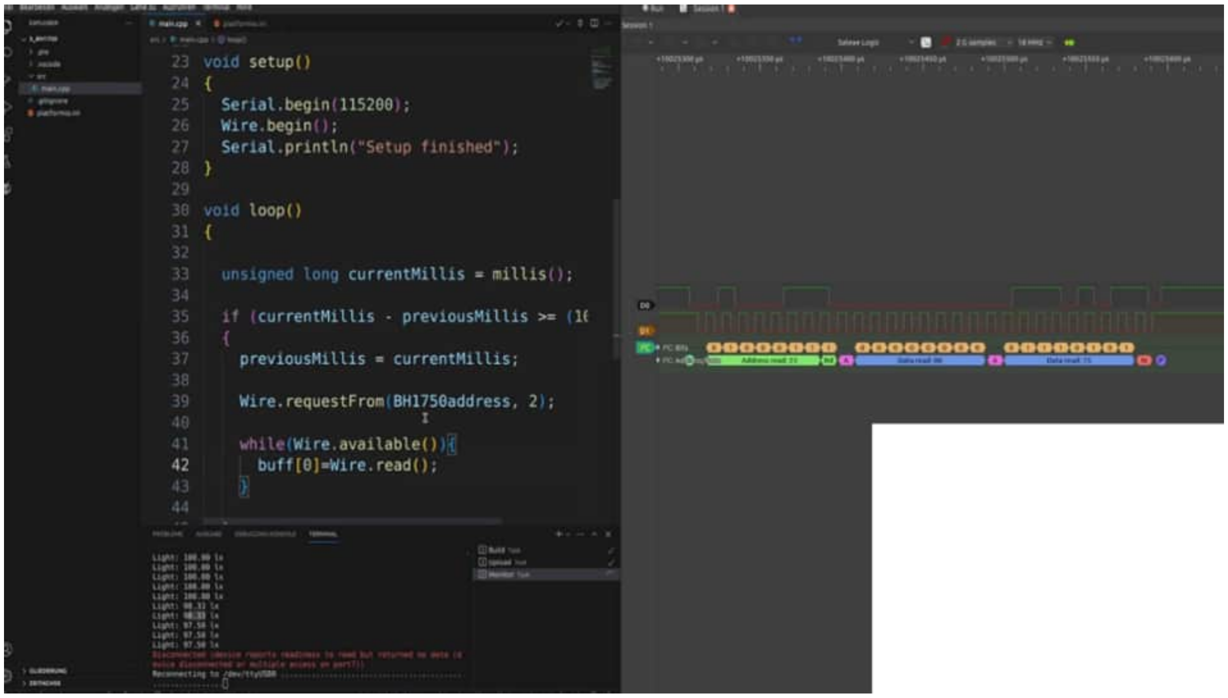
```
Light: 100.00 lx
Light: 100.00 lx
Light: 100.00 lx
Light: 100.00 lx
Light: 100.00 lx
Light: 98.33 lx
Light: 98.33 lx
Light: 97.50 lx
Light: 97.50 lx
Light: 97.50 lx
Disconnected (device reports readiness to read but returned no data (d
evice disconnected or multiple access on port?))
```

There's also a wire and tiny wire library out there when you're using an ATtiny, for example, because you can also use E2C components and you can use your own code as well. So what we want to do now is we want to start where the wire begins, because this is what we have to do.

Then in the loop, now the magic can happen. And we start by wire.request, I think request from, and we have to add here the address. So let's define the module address by saying integer bh1 70 50 address is, and that we do in hex x 23. So this is the address. And then we are declaring, let me see here and byte buffer, as we already did in the UART example. And this should hold here two bytes, this, the higher byte and the lower byte. So therefore we have enough space here. And I think that's it, that's it.

Speaker 1 (00:02:19) - So we could wire requests from, we handle and pass here our bh1 70 50 address. And this is what we saw before when we were transmitting the data here. We are not ready because we also have to set the size here, what we are expecting to get back. And we want to get back two bytes here. This is also necessary that we give them. And this is just a function call. Now the wire library does all of this. So that means if we got here and acknowledged back, then we can get the data here. And this could be done as nearly the same as in the UART chapter.

We have a wire dot available, and then we can get the data here. For example, in buff zero, I would like to get the first data out of it.



```
void setup()
{
  Serial.begin(115200);
  Wire.begin();
  Serial.println("Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1(

  {
    previousMillis = currentMillis;

    Wire.requestFrom(BH1750address, 2);

    while(Wire.available()){
      buff[0]=Wire.read();
    }

```

```
Light: 100.00 lx
Light: 100.00 lx
Light: 100.00 lx
Light: 100.00 lx
Light: 100.00 lx
Light: 98.33 lx
Light: 98.33 lx
Light: 97.50 lx
Light: 97.50 lx
Light: 97.50 lx
Disconnected (device reports readiness to read but returned no data (device disconnected or multiple access on port?))
Reconnecting to /dev/ttyUSB0 ..........................................
...............
```

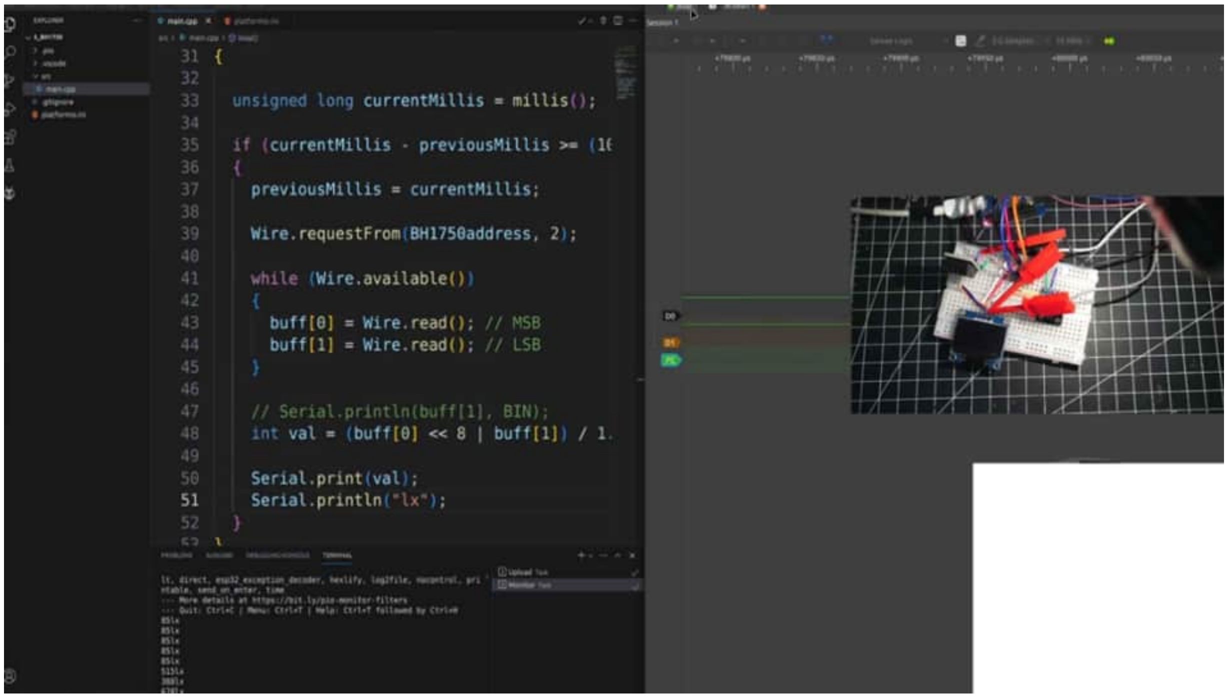
This is the most significant, if we could say so, the higher byte. And then we have here the lower byte, because here we are getting back both of them in one single connection as we have here. So therefore one wire read, second wire read. And here we have the first one in and here the second one. So let's test this. I would say I would like to see if this is already working.

And let's print out the buffer one because I have not so much, so much light in here. And I would like to have this in binary. Then let's print it out. So we are already getting some data out of it. That looks promising to me. Let's see if we can check if this is right. Zooming a little bit out, stop. Zooming in. And this should be one, one, one, two, three, four times and zero and one. Yes, okay. Looking good to me. That means we have our first communication established just with these few lines of code. Isn't that amazing? I think so.

So then the next step is the calculation because we are not finished yet. We are implementing the high byte, the low byte divided by 1.2. So now we need our basics from the bit operation and also the bit shifting. So we are creating an integer. You can create a long as well. Then what we do now is we are saying we want to have here the buff from index zero. And now try to find your own solution and then click on pause and then you can proceed. So it will help you much, much more in the learning process.

So what we are doing now is the first high byte will be shifted eight to the left. And then we are making an operation here, an OR operation with the buffer one. And all of these will then be divided by one.

by 1.2. So, let's see if this is working and then I will give a more detailed explanation. Just check if this will work. Okay, we are getting 85 lux here, looking good to me. Then I will start the flashlight here so that I'm seeing a little bit more.



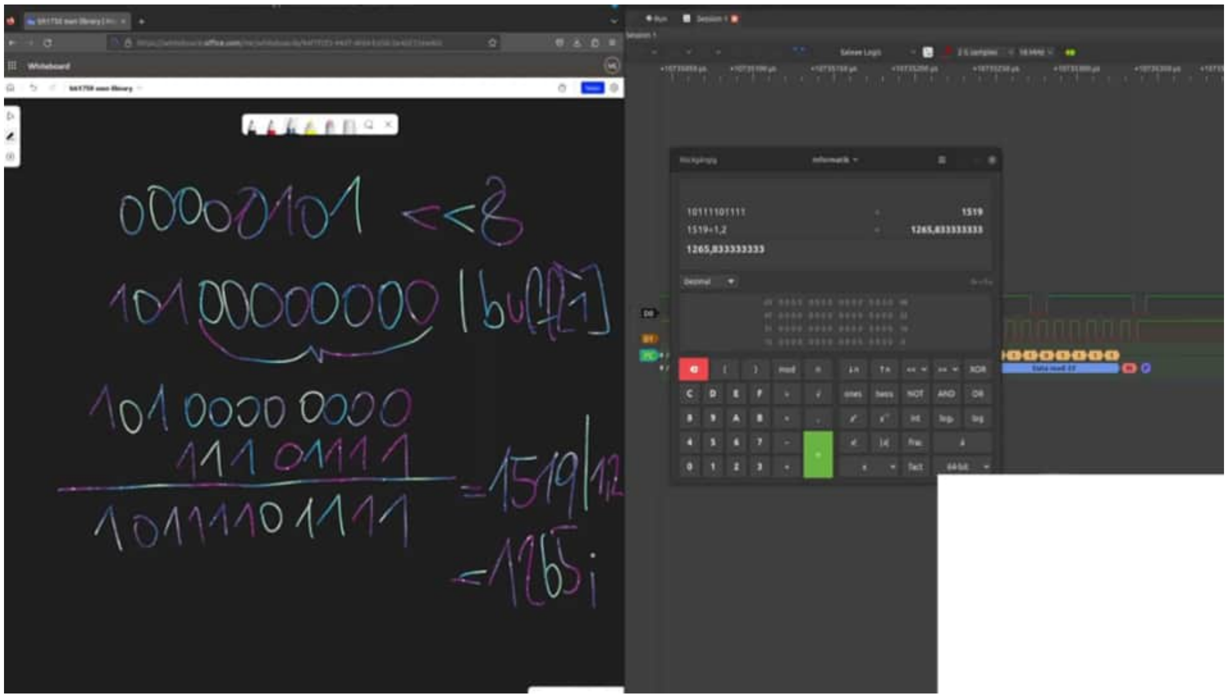
```
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1
  {
    previousMillis = currentMillis;

    Wire.requestFrom(BH1750address, 2);

    while (Wire.available())
    {
      buff[0] = Wire.read(); // MSB
      buff[1] = Wire.read(); // LSB
    }

    // Serial.println(buff[1], BIN);
    int val = (buff[0] << 8 | buff[1]) / 1.

    Serial.print(val);
    Serial.println("lx");
  }
}
```

```
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
85lx
85lx
85lx
85lx
85lx
515lx
388lx
678lx
```

As you can see here with the flashlight I also start the evaluation with the logic analyzer because here is the clamp exactly before. So, then I would like to stop here as well so that I can show you what I was doing here.

Zooming a little bit in and we're getting out one of these values should be stored here. So, once again open the blackboard and what we did now is we're getting the first in the index 0 we're getting those data and this is the... 4 5 zeros then we have a 1 0 1 and this will be shifted to the right. So, what we now have is 1 0 1 because the zeros are not interesting for us then we have 1 2 3 4 5 6 7 and 8. Then the logical OR with the buff 1 means that we are adding here on this space the low byte.

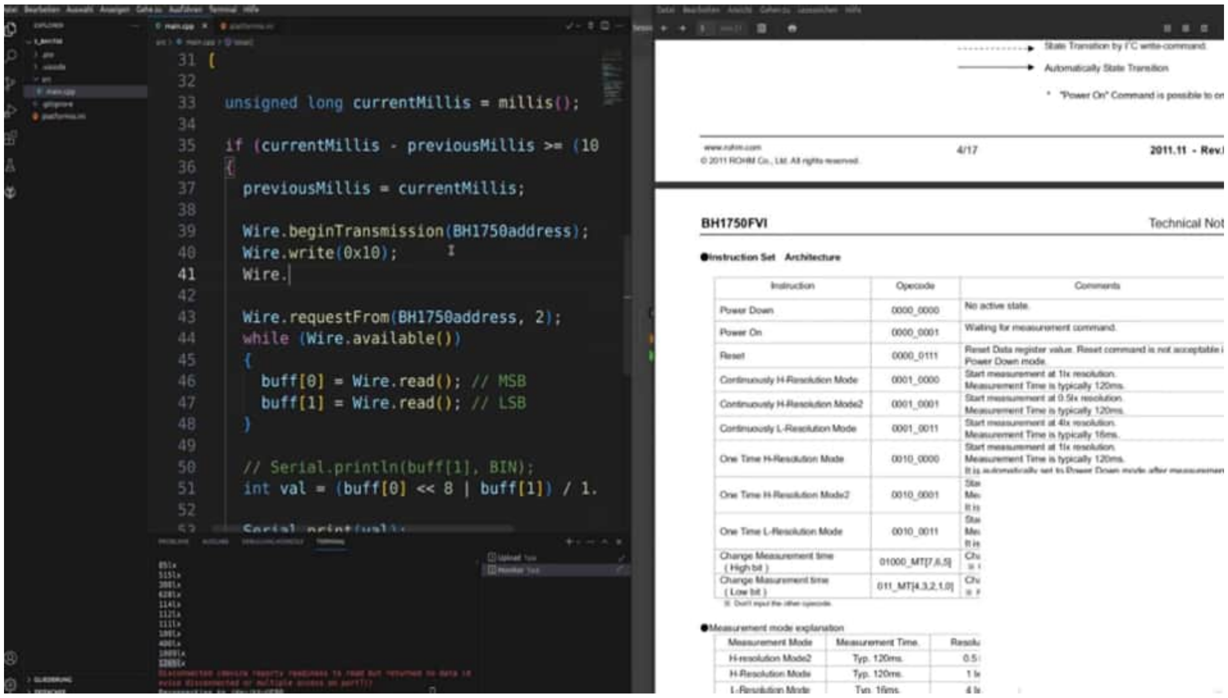
So, that means we have here 0 1 0 and this 1 2 3 4 and we are adding here the 1 1 0 and 1 1 1 1. So, our new value will be this one. So, in each place where we have a 1 we also have here the 1. Here we have a 0 and then we have 1 1 1 1 and if I don't mix up anything here this should be our new value. So, let's type in in the calculator or change to binary then we have 1 0 1 1 1 1 0 1 1 1 1. So, this should be 1519 1519 divided by 1.2 should be and decimal divided by 1.2 is 1265.83. Let's see if we have that. We have 65.83 and here we have 1265.



```
10111101111        =        1519
1519÷1,2           =        1265,833333333
1265,833333333
```

We have no commas therefore we have already implemented here our own library with just a few lines of code. Isn't that great and I really find of course these libraries aren't too big but you have to think of having a circuit which has really limited space on our microcontroller and therefore with these few lines of code we always have space for those few lines. And in addition to this, so that we have here really an incomplete example, I would also like to show you how we could add here in our code also the resolution. So, going a little bit up.

So, here for example for the continuous age resolution mode we have to set this data to start with one lux resolution. So, in this case we have to start here once again in transmission because this is in separate transmission. So, via begin transmission, we have to set up the address here once again and now we are saying why a write with this code and this code should be also translated into hex. So, therefore we have here the decimal where are the binary values from 1, 1, 2, 3, 4 and this will be in hex 10.



```
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (10
  {
    previousMillis = currentMillis;

    Wire.beginTransmission(BH1750address);
    Wire.write(0x10);
    Wire.

    Wire.requestFrom(BH1750address, 2);
    while (Wire.available())
    {
      buff[0] = Wire.read(); // MSB
      buff[1] = Wire.read(); // LSB
    }

    // Serial.println(buff[1], BIN);
    int val = (buff[0] << 8 | buff[1]) / 1.
```

BH1750FVI Technical Not

●Instruction Set Architecture

Instruction	Opecode	Comments
Power Down	0000_0000	No active state.
Power On	0000_0001	Waiting for measurement command.
Reset	0000_0111	Reset Data register value. Reset command is not acceptable i Power Down mode.
Continuously H-Resolution Mode	0001_0000	Start measurement at 1lx resolution. Measurement Time is typically 120ms.
Continuously H-Resolution Mode2	0001_0001	Start measurement at 0.5lx resolution. Measurement Time is typically 120ms.
Continuously L-Resolution Mode	0001_0011	Start measurement at 4lx resolution. Measurement Time is typically 16ms.
One Time H-Resolution Mode	0010_0000	Start measurement at 1lx resolution. Measurement Time is typically 120ms. It is automatically set to Power Down mode after measuremen
One Time H-Resolution Mode2	0010_0001	Sta Me It is
One Time L-Resolution Mode	0010_0011	Sta Me It is
Change Measurement time (High bit)	01000_MT[7,6,5]	Ch ※
Change Measurement time (Low bit)	011_MT[4,3,2,1,0]	Ch ※

※ Don't input the other opecode.

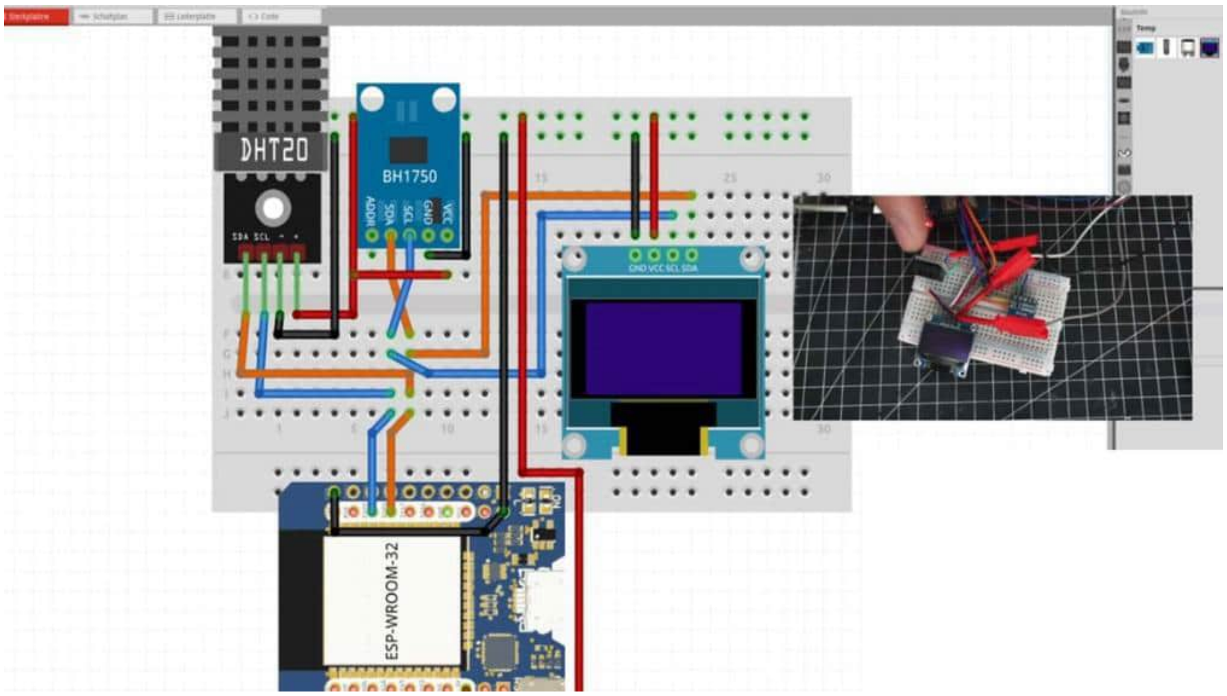
●Measurement mode explanation

Measurement Mode	Measurement Time.	Resolu
H-resolution Mode2	Typ. 120ms.	0.5
H-Resolution Mode	Typ. 120ms.	1 lx
L-Resolution Mode	Typ. 16ms.	4 lx

So, we are transmitting here 0x10 and then we are saying via end transmission we're making a short delay I think this is also written there that we have to make here and pause between where was it here wait to complete the first measurement roundabout maximum 180 milliseconds this means maximum does this measurement means so let's do this one then re-upload it again and there we go we don't really see an exchange and change but let's see here in the logic analyzer we can see that we sent here the resolution making 200 milliseconds pause and then we're getting the data and this is how we could also send here to and register some commands changing something and then getting data in the read modus and also here was just an easy line and we will see in the next chapter with this dht 20 for example that it's also easier but a little bit more tricky to evaluate but this will capture in the next chapter.

SKETCH AND WIRING DHT20

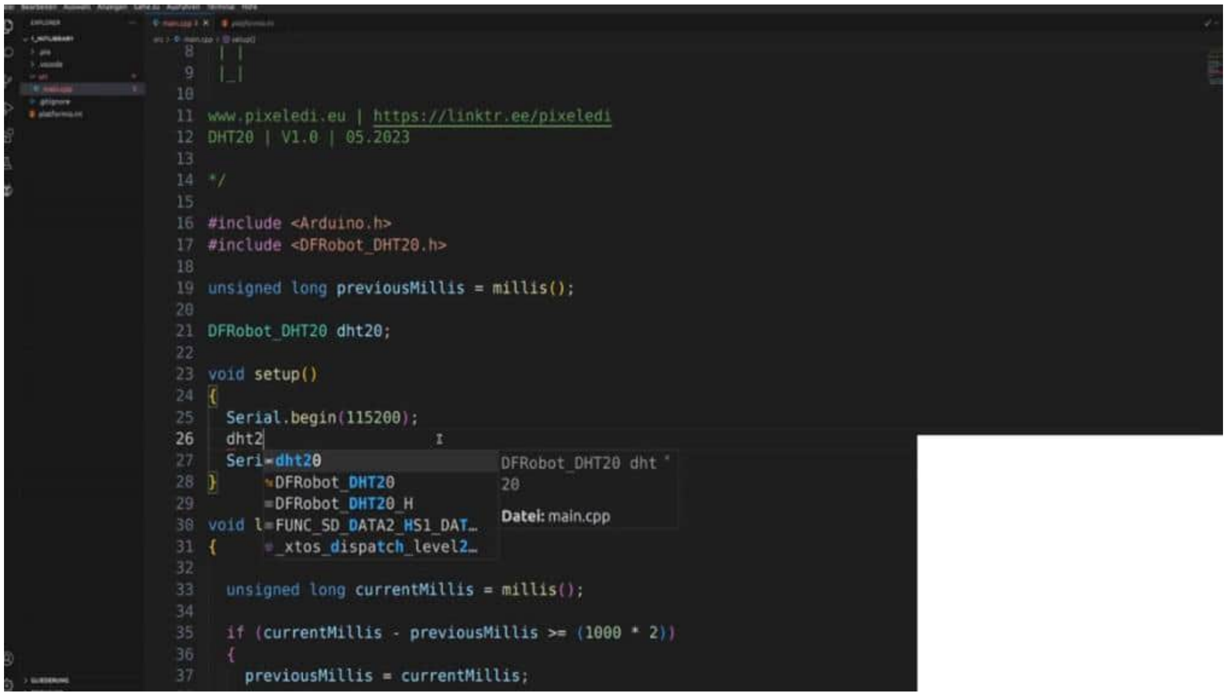
Let's examine the next sensor, and this will be the DHT 20 and temperature sensor similar to the T2, but here with E squared C. You can identify it with a black housing most of the time. And I already had it here. So also my connection points to the e squared t pass are I have to I haven't changed because, they are all connected together. So, therefore, we open up a fresh sketch.



And what we are doing now is we are using the DF Robert Library for the DHT 20. Then let's start. Let's start by including the library. So our main goal will be that we have a simple sketch, which gives us the temperature and humidity so that we then later on capture in pulse fuel all of the data. And then setting up this use case, writing our own library or our own sketch for getting the data.

Okay. That's it. Then we are creating the object it's called dfrobot, I think, DHT 20 equals to DHT 20. This is the name of the object. Then we start it, dht20.

Begin Not Rocket Science. And as you already know, in the loop, we are using 2 floats, I would say, float DHT 20 temperature, temperature equals to DHT 20, and call it temperature. Easy as it is. Then we have DHT 20, humidity equals to DHT 20, get humidity, from measurement range 0 to 100%. So I think I'm not really sure that we have to multiply it by 100 so that we get really exact values out of it.



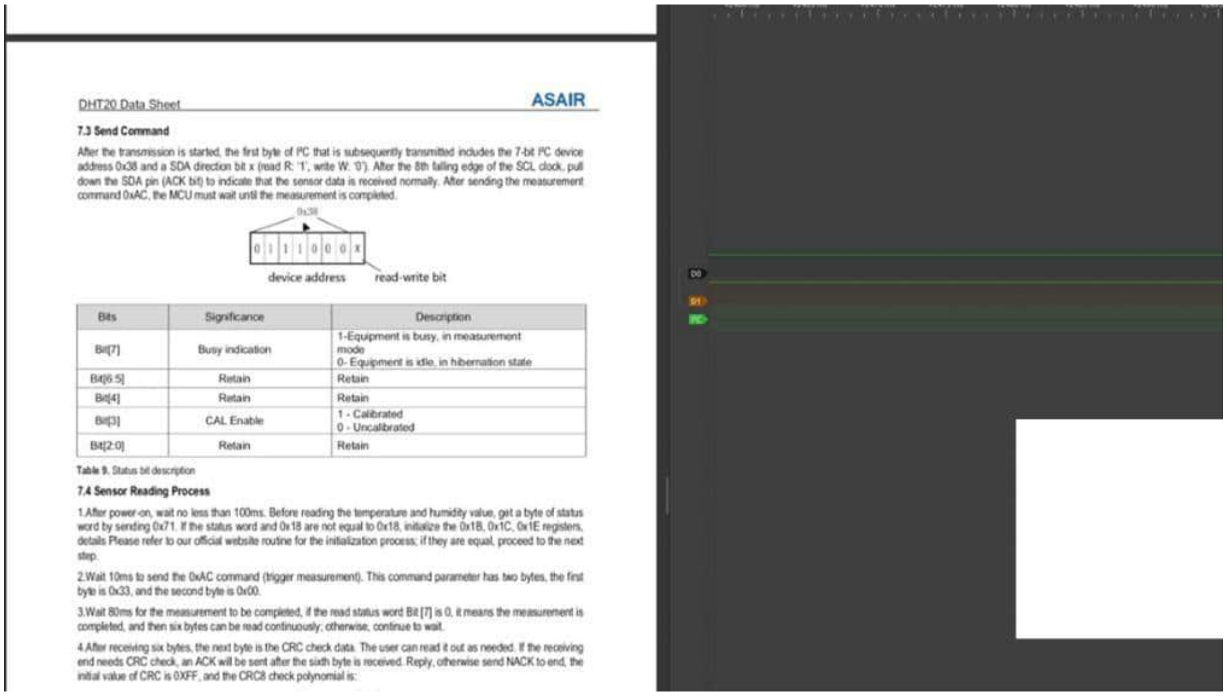
```
  | |
  |_|

www.pixeledi.eu | https://linktr.ee/pixeledi
DHT20 | V1.0 | 05.2023

*/

#include <Arduino.h>
#include <DFRobot_DHT20.h>

unsigned long previousMillis = millis();

DFRobot_DHT20 dht20;

void setup()
{
  Serial.begin(115200);
  dht2
  Seri dht20                        DFRobot_DHT20 dht
}      DFRobot_DHT20                20
       DFRobot_DHT20_H
void l FUNC_SD_DATA2_HS1_DAT...     Datei: main.cpp
{      _xtos_dispatch_level2...

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1000 * 2))
  {
    previousMillis = currentMillis;
```

So let's test it out. And then we are printing out both of the values and uploading the code again. And there we go. We have nearly 29 degrees Celsius inside my office and 47% humidity. Okay.

These are the values, what we are getting currently out of this sensor. So we have set up the first step to evaluate the Sequencing communication now.

ANALYSIS DHT20

Ready for some analyzing. Let's start by analyzing the DHT 20 data sheets and then we are focusing on pulse fields so that we calculate our values. So here you can see it's an upgraded product for the DHT 11, and it has a little bit more accuracy than the DHT11 and also the voltage support is really good because we can use here an ESP or an Alrino owner, for example, So let's go down to business and also to page 1010 because here we have more about our communication process. So here we can see the address. It's the 0 X38 that just some e squared c standards, blah blah, what we and what I would like to focus is those two parts.



DHT20 Data Sheet ASAIR

7.3 Send Command

After the transmission is started, the first byte of I²C that is subsequently transmitted includes the 7-bit I²C device address 0x38 and a SDA direction bit x (read R: '1', write W: '0'). After the 8th falling edge of the SCL clock, pull down the SDA pin (ACK bit) to indicate that the sensor data is received normally. After sending the measurement command 0xAC, the MCU must wait until the measurement is completed.

Bits	Significance	Description
Bit[7]	Busy indication	1-Equipment is busy, in measurement mode 0- Equipment is idle, in hibernation state
Bit[6:5]	Retain	Retain
Bit[4]	Retain	Retain
Bit[3]	CAL Enable	1 - Calibrated 0 - Uncalibrated
Bit[2:0]	Retain	Retain

Table 9. Status bit description

7.4 Sensor Reading Process

1.After power-on, wait no less than 100ms. Before reading the temperature and humidity value, get a byte of status word by sending 0x71. If the status word and 0x18 are not equal to 0x18, initialize the 0x1B, 0x1C, 0x1E registers, details Please refer to our official website routine for the initialization process; if they are equal, proceed to the next step.

2.Wait 10ms to send the 0xAC command (trigger measurement). This command parameter has two bytes, the first byte is 0x33, and the second byte is 0x00.

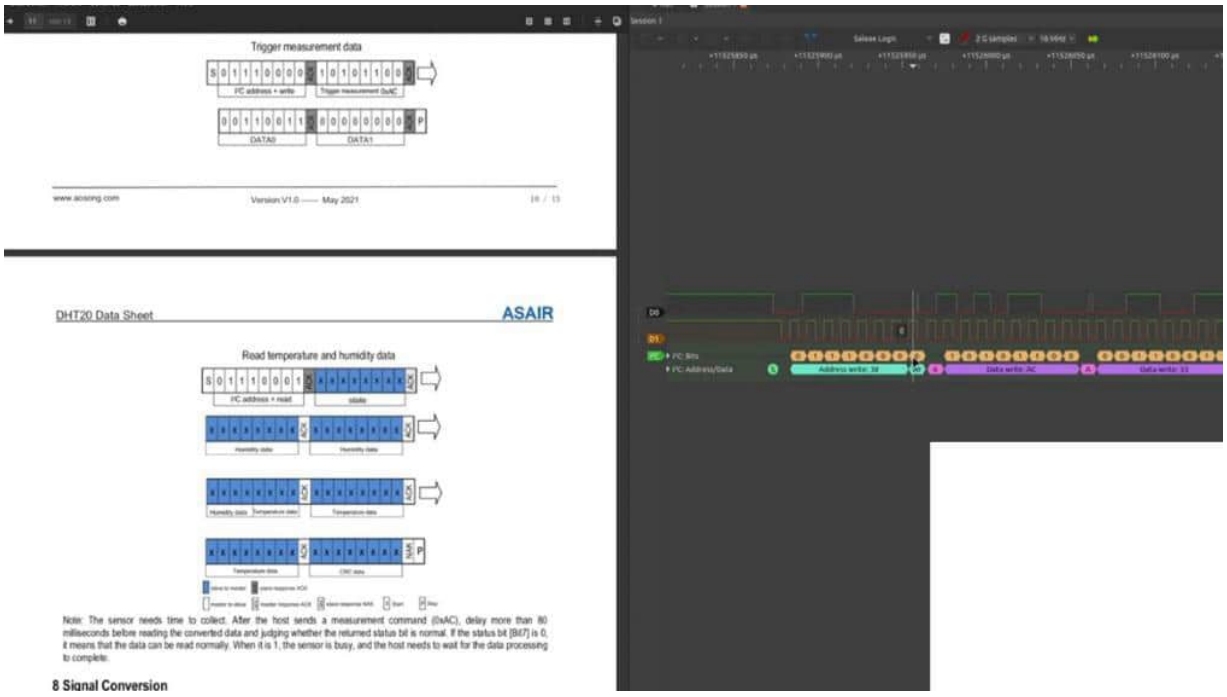
3.Wait 80ms for the measurement to be completed, if the read status word Bit [7] is 0, it means the measurement is completed, and then six bytes can be read continuously; otherwise, continue to wait.

4.After receiving six bytes, the next byte is the CRC check data. The user can read it out as needed. If the receiving end needs CRC check, an ACK will be sent after the sixth byte is received. Reply, otherwise send NACK to end, the initial value of CRC is 0XFF, and the CRC8 check polynomial is:

So we have 2 packages here. 1 is to trigger the measurement. And the second one, what we are getting back is here, all of the reading of humanity, and temperature. So I would suggest, before we go further on the datasheet, the sketch is working here. properly.

And now I can click here on run because I haven't changed anything here because it's also in e squared c logic from before, the 1, 0 and 1 and e squared c. Then I would like to do so. We are cupping the line here so that we see the same results. So these two values should be some here. And as we can see here, we have here a little bit more data.

So now let's focus here on the first one. What do we have here? Here we see the address and we see here that we are writing something because here, the last bit is the 0 bit. Then we got an acknowledgement, so the communication started. And then we have 3 bytes here.



www.aosong.com Version V1.0 —— May 2021

DHT20 Data Sheet ASAIR

Note: The sensor needs time to collect. After the host sends a measurement command (0xAC), delay more than 80 milliseconds before reading the converted data and judging whether the returned status bit is normal. If the status bit [Bit7] is 0, it means that the data can be read normally. When it is 1, the sensor is busy, and the host needs to wait for the data processing to complete.

8 Signal Conversion

with the register AC3300. So then let's see. Can we find those values here? Yes. Sense of reading process.

So after power on, when no less than 100 milliseconds before reading the temperature, blah blah blah. Wait 10 milliseconds to send the AC command is the trigger remeasurement. This is what we do here. And this command parameter has 2 bytes. The first by this, 33 and 0.

That is our trigger command. And this is also what we have here, the AC and data. Perfect. And then wait 80 meg, milliseconds for the measurement to be complete. And if the restart does 7 means it is completed.

So 80 milliseconds. Let's check if we can also see if this is correct. What do we have here? 20 milliseconds. No.

really the 80 milliseconds. What do we have here? Roundabout, 20 milliseconds. And then the next process started. So let's see what we have here.

This is not really according to the datasheet. I don't know why it's not on the initial purpose, but nevertheless, it will function because it's the DF Robert library. So then we get the next few inputs. We have here the e squared c address as well, of course, then and acknowledged. And now we have a read bit.

and we got here. Well, let's see here. so now. No. No.

1, 2, 3456. 123456. Okay. The CRC is missing then because here we have the state And then we have 1, 2a half. And this is a little bit trickier afterwards to get the data out of it.

1, 2a half. and 2a half for the temperature. Okay. So this is the signal flow, and here is a calculation of what we need, then let's start with some analyzing process. So this is the state byte.

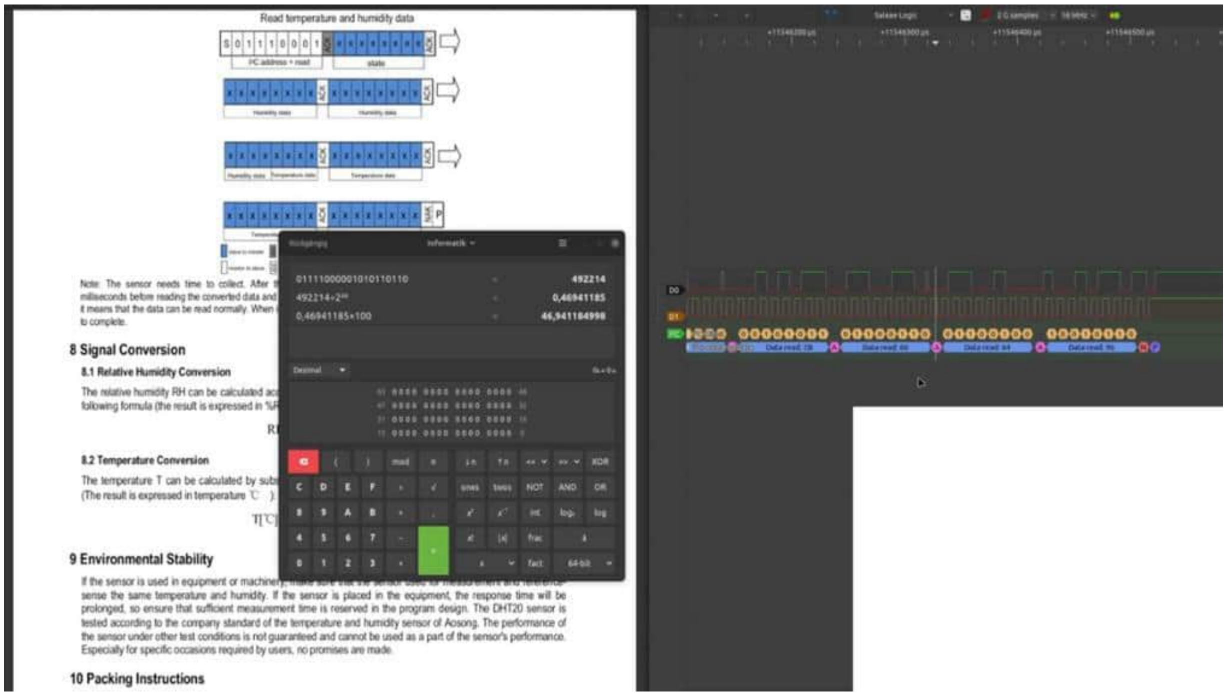
I don't care what the state means, but I care the next two bytes are humidity and the first nipple for pits is also immunity. So let's check that those values should be the humidity value. We're starting here on the left side by typing in the first byte. 0. So first, I have to change to buy a rim, then 011110000 is the first byte.

Second bytes 00101011, and the first four bits here are 1 10. Enter changing to decimal value. And now I have to end here. The relative humidity can be calculated by dividing by 2 to the power of 20. Enter multiplies 100 should be something like this.

Let's see what our last value was. 46.94 Wow. Cool. It works. Okay.

Let's do the same thing for the temperature. So that means we have here 2a half. Let's go a little bit out and zoom in, that means this is the first half, pick the first liberum, and then the last 2. Starting here with 0110, and then despite, or we have to change to binary. Once again, then starting with the sec, the first full pipes, this one, 011s 00100.

Last byte, 10010110. And done, Digital, then we have the temperature divided by 2 to the power of twin no, 20. Multiply 200-50 is 29 Degrees Celsius point 91. Not really. Here we have a little bit of, it's not in-depth detail, but it's, some comma.



```
01111000001010110110          =          492214
492214÷2²⁰                    =      0,46941185
0,46941185×100                =    46,941184998
```

Note: The sensor needs time to collect. After t… milliseconds before reading the converted data and… it means that the data can be read normally. When… to complete.

8 Signal Conversion

8.1 Relative Humidity Conversion

The relative humidity RH can be calculated ac… following formula (the result is expressed in %R…

8.2 Temperature Conversion

The temperature T can be calculated by sub… (The result is expressed in temperature ℃).

9 Environmental Stability

If the sensor is used in equipment or machinery, make sure that the sensor used for measurement and reference sense the same temperature and humidity. If the sensor is placed in the equipment, the response time will be prolonged, so ensure that sufficient measurement time is reserved in the program design. The DHT20 sensor is tested according to the company standard of the temperature and humidity sensor of Aosong. The performance of the sensor under other test conditions is not guaranteed and cannot be used as a part of the sensor's performance. Especially for specific occasions required by users, no promises are made.

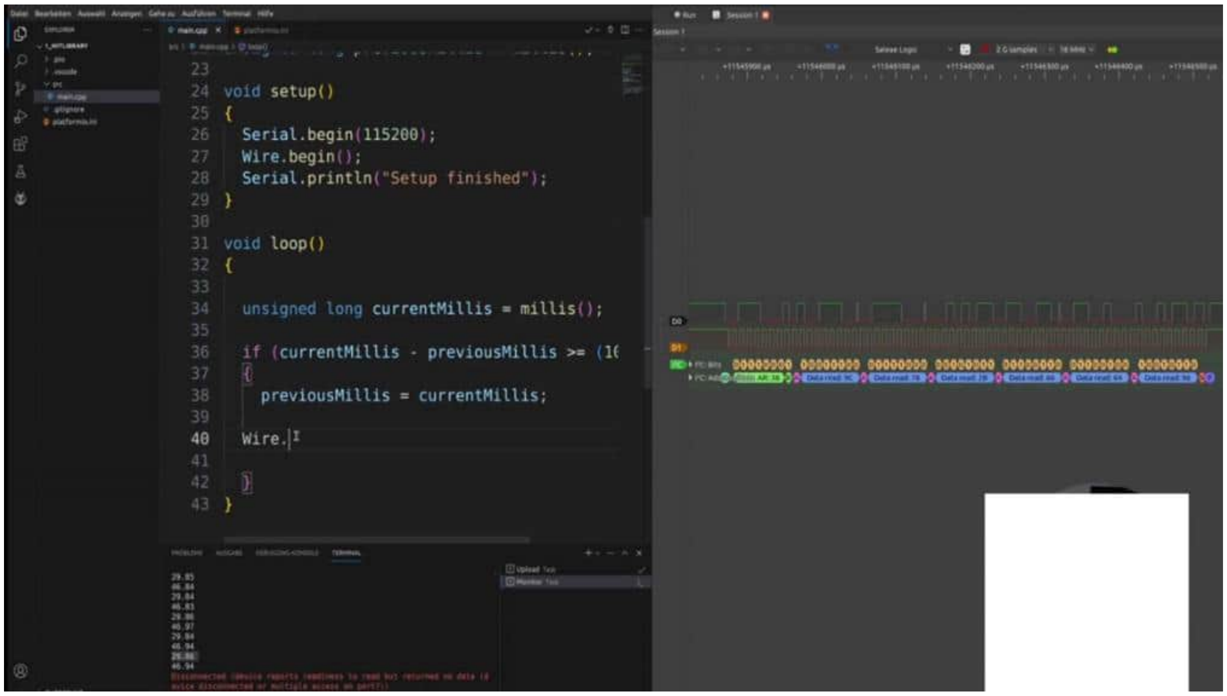
10 Packing Instructions

Let me see. But we can lift with this accuracy. And, therefore, I would say we have already mentioned here, yeah, could comprehend the data flow from the DHT 20 sensor, and wouldn't it be nice in the next project on we write our own library with this kind of knowledge so that we can get rid of the DF robot library

SKETCH DHT20 WITHOUT EXTERNAL LIBRARY

ready for the next coding challenge. So now, we want to have this flow inside our arduino sketch. And we go step by step and also how we can evaluate and also we are investigating then afterwards our own library with the pulse view. So first step, I would say, let's get rid of all of the things that we did before because we don't need any extra library. So also the library here, we can pick out then we include here the wire because this is our helper library.

And then dot h. And then we have to define 2 variables here, 2 global variables. I would say VHD address, and this was, in hex 38, I think, if I remember correctly. And we make a buffer here as we did before with the PH1750. And we need 6 indexes here, I think.



```
void setup()
{
  Serial.begin(115200);
  Wire.begin();
  Serial.println("Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (1(
  {
    previousMillis = currentMillis;

  Wire.

  }
}
```

```
29.85
46.84
29.84
46.83
29.86
46.97
29.84
46.94
29.86
46.94
Disconnected (device reports readiness to read but returned no data (device disconnected or multiple access on port?))
```

Let me see if this is correct. 1, 2, 3, 4, 5, 6. Yes. The 7th is not interesting for us. we could implement it, but, also the DF robots didn't do it.

So let's skip this one. Okay. Then in the setup, as we did before, wire dots begin so that the E squared C communication could be started. Now let's do some magic. The first magic is also in the previous example, you should know that wire dot, begin transmission, and then we are sending out the DHT address. And this should be here, let me see this first example.

So we're sending out the DHT in the right models, then if we are getting back here to acknowledge, this is all done by the wire dot edge, then we are writing out here the 0xac commands. wirewire. Right, then I copy and paste this two times because I want to write 33, and I would like to write 00. afterwards, we have to wire and transmission. And we are not finished.

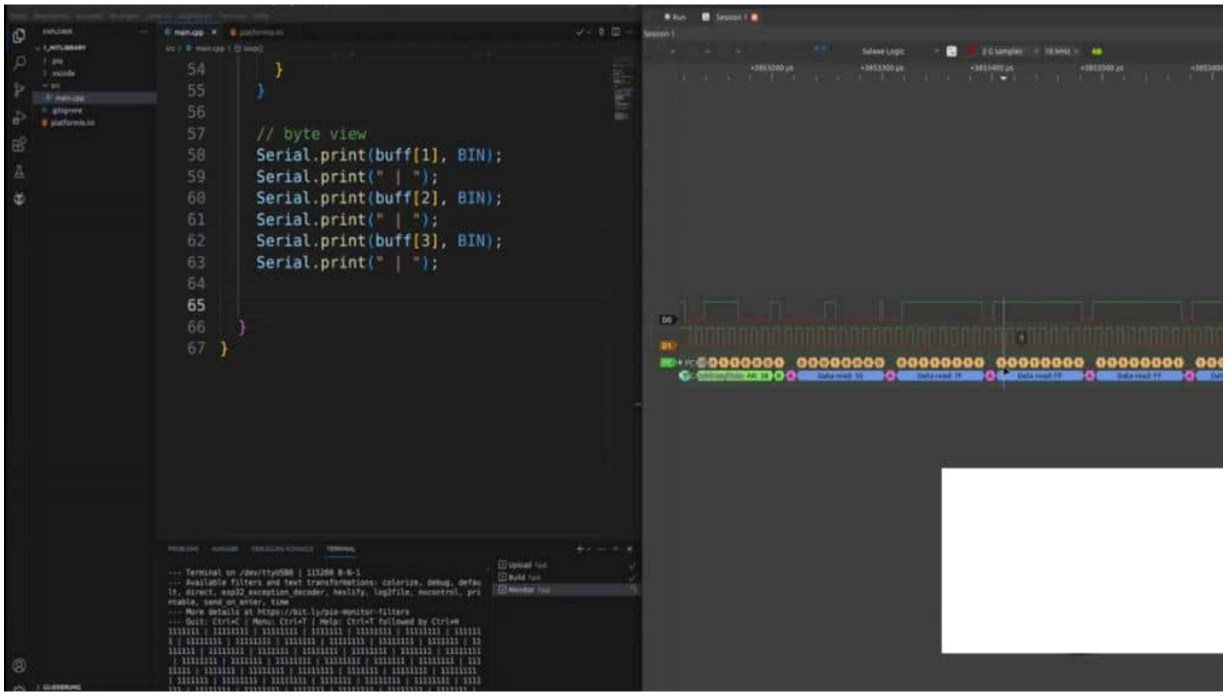
checking back to the flow because yes, it's stated out. Weights 80,000,000 seconds for the measurement to be called bleed. And this is, we should also do that, and I would like to use 90 milliseconds here, and then the next flow can be started by reading out the values. And this is also an easy part because we already did this one. So we are starting the wire request from the DHT address.

Then we are now in the reading part because, we are not not what is, request from? So it should work now. Let me check the compiler. I missed something here because we have to adhere to the size. So, 6 bytes, I am expecting it.

Now, this should be looking a little bit better. So just run the compiler, then also the intelli sense shows you that there is no error anymore. And now we want to get here 6, 6 bytes. So we could do that, 1 by 1, or we may hear a wire dot available, for example, and inside this while we could make 4 here. So let's see if we can make an auto-completion.

Completion. Let's check it out here. 4. Yes. And we're counting this one.

For example, e, should be less and 6 so that we get 6 values. Size could be the same here, and this is our code. And then we say buffer. So we don't have to write it six times here with the index, I am equal to wire dot read. it's a more convenient way and not writing here six times here, the buffer element.



```
        }
    }

    // byte view
    Serial.print(buff[1], BIN);
    Serial.print(" | ");
    Serial.print(buff[2], BIN);
    Serial.print(" | ");
    Serial.print(buff[3], BIN);
    Serial.print(" | ");


  }
}
```

```
--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, defau
lt, direct, esp32_exception_decoder, hexlify, log2file, nocontrol, pri
ntable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
[illegible]
```

Okay. So let's test the first thing if we are really making progress here or not, and I have already prepared this. So just checking out if we're getting out here the first three bytes not with the serial index because I just want to have the humidity level here. Then, uploading the code. So I'm getting a lot of them here.

So let's check what's going on here. So then I'm clicking on stop. Let's see if we get the, really only once and once and once and once. So First one is skipped. Then I got here, okay, this looks promising.

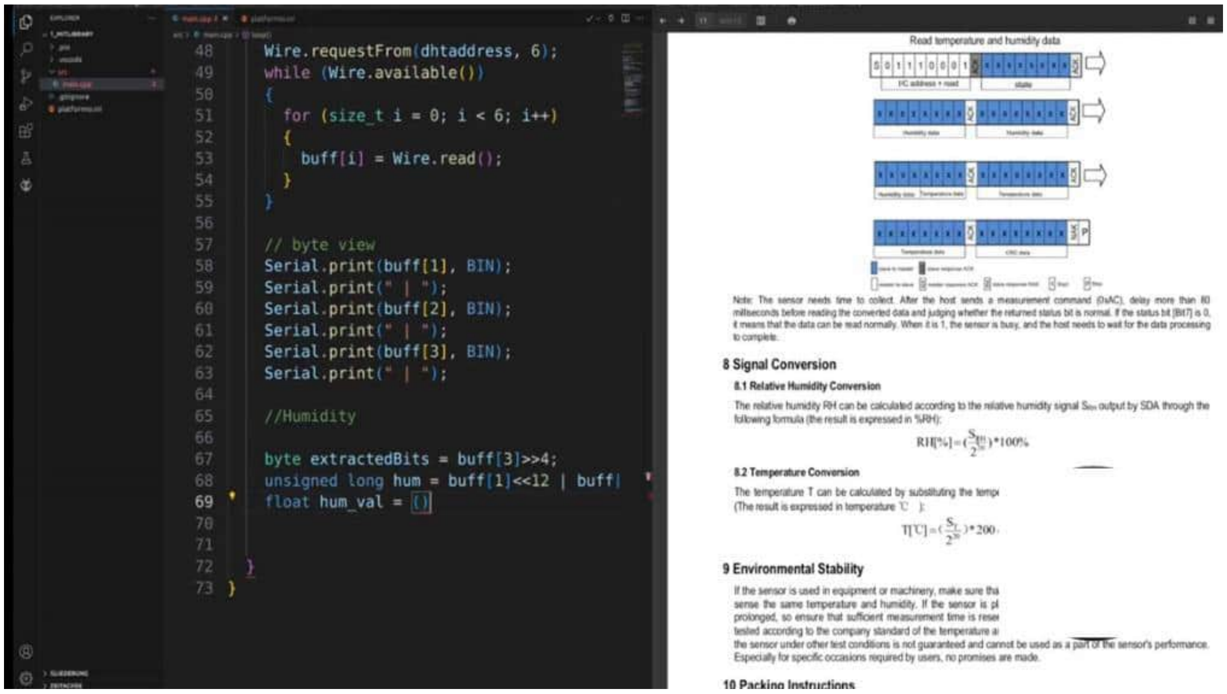
So here, I only have 7 once. And then 1111. Okay. This looks good. I was a little bit distracted while I got here only once out, but this looks really promising.

So now, let's close it so that we don't get distracted. Now we need some special operations because we have here the setting that we need here half of the bytes. And this is what we also trained here in the seek chapter. So I would like to show you the code and then we're going through it step by step. First of all, we are getting the humidity and home media.

And this is done by it. First of all, we have to direct the byte from the index 012 3. It's the 3rd index. There we have to extract the byte. So the, byte, I call it directed bits and, buff 3 with it, with the 3 index, and I only want the first 4 bits.

Therefore, we'll make a bit of a shift to the right. Yeah. That should work. Then, we have an unsigned long. And now I would like to do this in one line.

So the humidity is buffer 1, this 1, then we have to pitch shift 12 to the left and I explain, you know, the next step of what I'm doing here, then this 012, the index, path 2nd index, we have to shift now, shift to bitch shift to the left by 4 because we have 4 left, and then we have the extracted bits. And this should be our new value. So, we can use an unsigned long because humidity should be in the minus. And then we are printing out or we say a float. We are not finished yet.



```
    Wire.requestFrom(dhtaddress, 6);
    while (Wire.available())
    {
      for (size_t i = 0; i < 6; i++)
      {
        buff[i] = Wire.read();
      }
    }

    // byte view
    Serial.print(buff[1], BIN);
    Serial.print(" | ");
    Serial.print(buff[2], BIN);
    Serial.print(" | ");
    Serial.print(buff[3], BIN);
    Serial.print(" | ");

    //Humidity

    byte extractedBits = buff[3]>>4;
    unsigned long hum = buff[1]<<12 | buff|
    float hum_val = ()


  }
}
```

Read temperature and humidity data

Note: The sensor needs time to collect. After the host sends a measurement command (0xAC), delay more than 80 milliseconds before reading the converted data and judging whether the returned status bit is normal. If the status bit [Bit7] is 0, it means that the data can be read normally. When it is 1, the sensor is busy, and the host needs to wait for the data processing to complete.

8 Signal Conversion

8.1 Relative Humidity Conversion

The relative humidity RH can be calculated according to the relative humidity signal S_{RH} output by SDA through the following formula (the result is expressed in %RH):

$$RH[\%] = \left(\frac{S_{RH}}{2^{20}}\right) * 100\%$$

8.2 Temperature Conversion

The temperature T can be calculated by substituting the temp
(The result is expressed in temperature ℃):

$$T[℃] = \left(\frac{S_T}{2^{20}}\right) * 200$$

9 Environmental Stability

If the sensor is used in equipment or machinery, make sure tha
sense the same temperature and humidity. If the sensor is pl
prolonged, so ensure that sufficient measurement time is rese
tested according to the company standard of the temperature a
the sensor under other test conditions is not guaranteed and cannot be used as a part of the sensor's performance. Especially for specific occasions required by users, no promises are made.

10 Packing Instructions

Whom well year equals to, what was the calculation? Calculation was the come volume, this will automatically translate it or transferred into, decimal volume divided to the power, 2 of 20 here. So we have the first part, and then we are seeing multiply 100 looking good. Then we have a serial print here. And let's test it out, and then we are going through the details of this very sexy line.

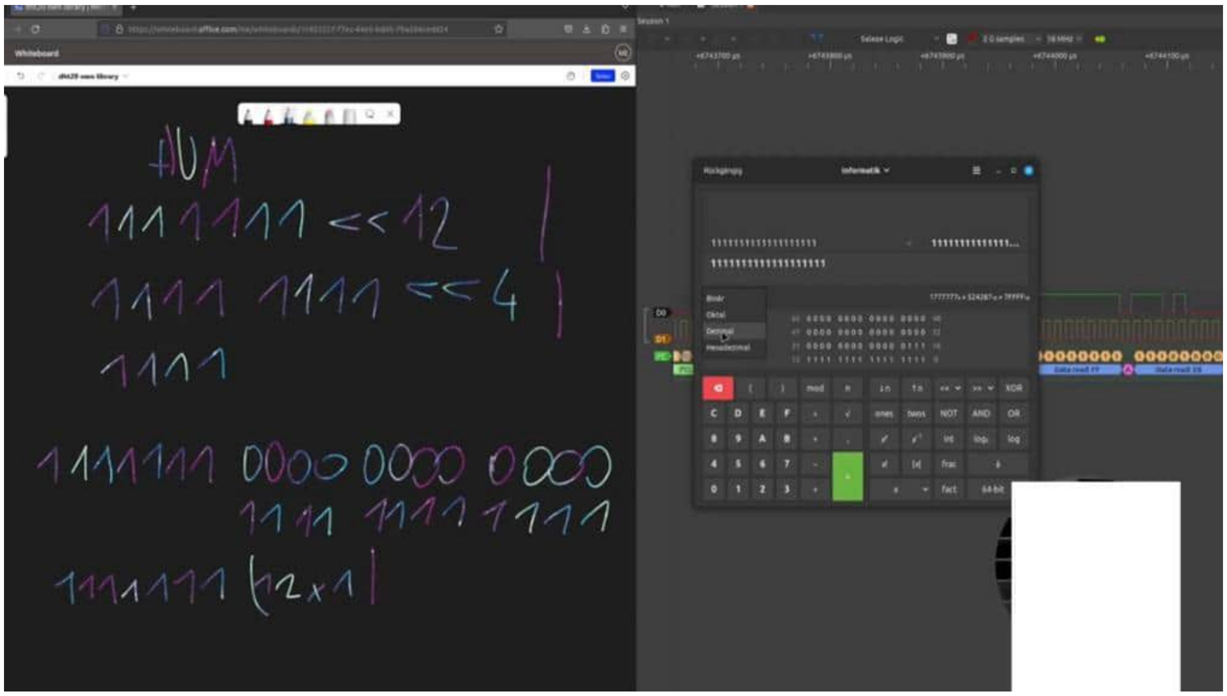
So let's check it out. Then I also would like to run a pulse. Then once again, open up the line so that we have really the same data, check, click on stop, So we have the humidity here by 50. Looks promising, but let's see what pulse fuel says here. So here we have the first command, the trigger commands, that, nothing new for us. And here we have the state, and this should be now our new value.

So let's focus on what I did with this line. So, grab here on the Blackboard once again so that we can do this step by step. So that means those two bytes and this nimble should be now combined to one value. And how could I do that with the humidity in three steps? The first step is we are getting the first byte.

So these are 7 ones because the first one is a 0. So here we are. 1, 2, 3, 4, 5, 6, 7. And I have to shift it twelve times to the left because I have 8 and 4 is 12, I need space for 12 more numbers. Then there comes an or.

For the second number, these are 4, 8 ones. 341234. And I have to shift those values four times to the left because I need space for the extracted bits for the 4 of this one. And these are also 4 ones. And this is now my new value.

So that means I have, in the first line, these 7 ones, 4, 5, 6, 7, and then 12 0s. So that means for 8 and the other 4. And I made an or with those values. So that means I have 1111 111 1. And those 4 should also have space for the next one.



So I am here at 1111. A lot of ones look really a little bit strange, but it will give us the right value. So in summary, I have here 71 1, 2, 3, 4, 5, 6, 7. Then speeding up here, I have 12 times 1. So let's try out here in the binary.

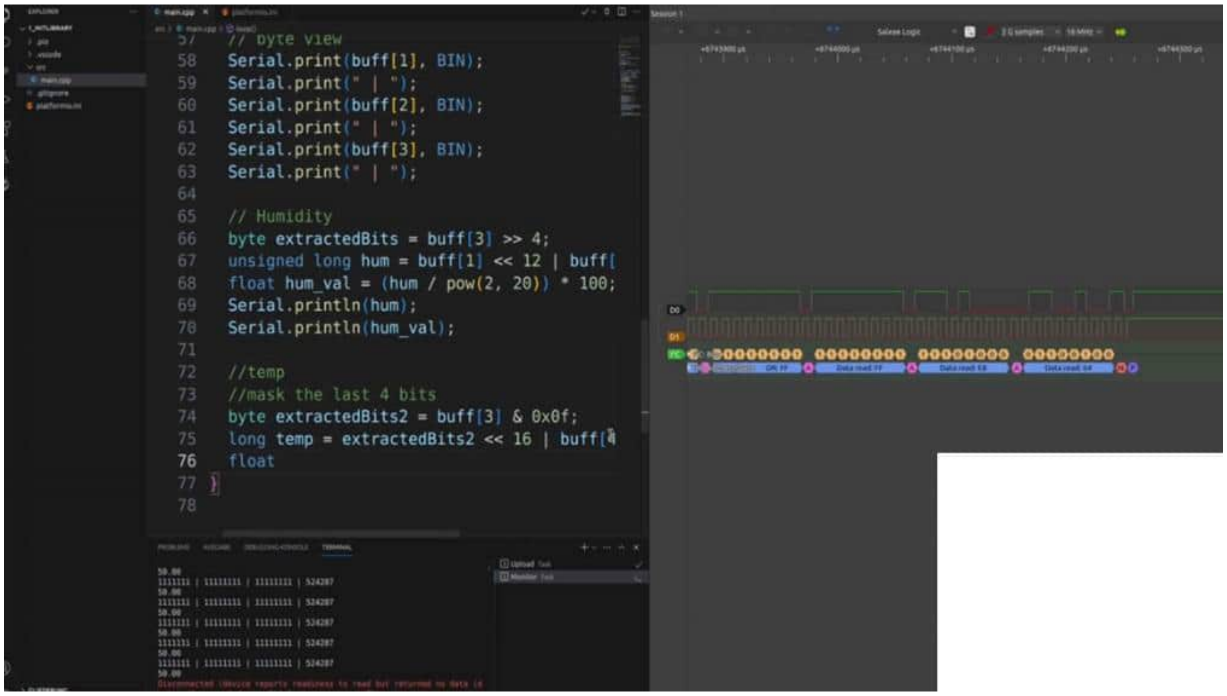
We have here 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. 112. Enter binary to decimal divided by 2 What was the calculation it is divided by? Was this right to the power 20 divided by 2 to the power of 20 and 100. So we have here the value of 50 in the humidity percent of humidity.

Nice one. Really nice one. Was it really a little bit distracted by all of the ones here in our explanation, but it looks promising so far. Okay. Now, we could do the same thing with the temperature.

And I would invite you to click here on pause and try it on your own because it's a little bit different now. So I hope you tried your own approach. And now, I would like to show you how to do it with the temperature values. So first of all, we have to mask the last four. This is also covered in our general, in our 1st general chapter because the first four refer to the humidity and days last 4 will be needed for our temperature. So masking out is extracted bits 2, for example, we need the buffers here.

Should it be a 012 3. And now, we make a mask with an ant, 0f, in hex, tilt a tiny version of it. And then we'll make the long temperature equal to first, we start with the extracted bits, then we make a bit shift and how much do we need space here. So these are the first extracted bits. Then we have 88 that means 16 spaces we need before 2 full bytes, then we make an or, then we have the buffer 4, and we shift the 8 to the left because we need 8 more spaces.

Then we made an or, and then we say Buff 5. what do you think? Seems legit and comprehensible. Then we are making a float here, the temperature value equals the temperature divided by the power of 2 20 multiplies. Let's do it outside the brackets.



```
  // byte view
  Serial.print(buff[1], BIN);
  Serial.print(" | ");
  Serial.print(buff[2], BIN);
  Serial.print(" | ");
  Serial.print(buff[3], BIN);
  Serial.print(" | ");

  // Humidity
  byte extractedBits = buff[3] >> 4;
  unsigned long hum = buff[1] << 12 | buff[
  float hum_val = (hum / pow(2, 20)) * 100;
  Serial.println(hum);
  Serial.println(hum_val);

  //temp
  //mask the last 4 bits
  byte extractedBits2 = buff[3] & 0x0f;
  long temp = extractedBits2 << 16 | buff[4
  float
}
```

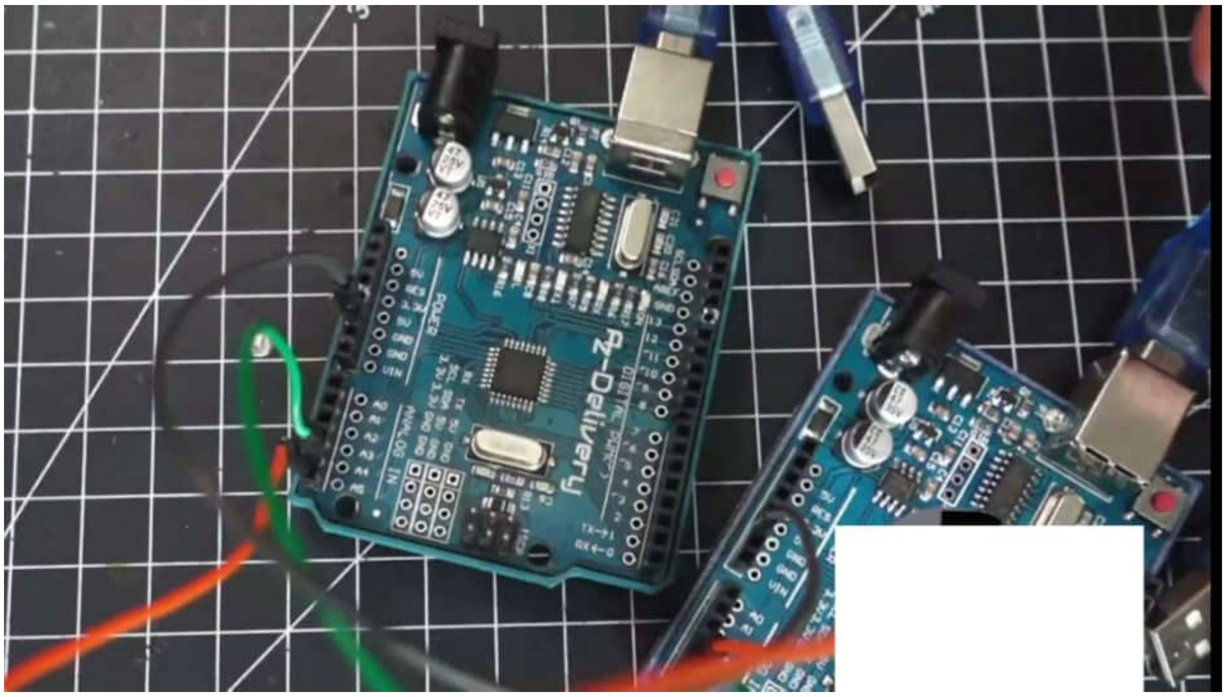
```
50.00
1111111 | 11111111 | 11111111 | 524287
50.00
1111111 | 11111111 | 11111111 | 524287
50.00
1111111 | 11111111 | 11111111 | 524287
50.00
1111111 | 11111111 | 11111111 | 524287
50.00
1111111 | 11111111 | 11111111 | 524287
50.00
[illegible]
```

multiplies by 200-50 should give us here the temperature value on 0 print line temp. Where are you? Here you are. So let's do one, and let's check it out if we got some really good values here. And we're getting here the humidity values and here we have the values from our temperature.

It is 30 degrees, 30.03 degrees, and that is what we also want to see here. And that was the goal of this project to show you that it's not really rocket science to communicate via e squared to the modules which your own written code, and you don't need every time and finished library because here, it also makes fun helps you to understand how this whole library or whole sensor is working and often you don't know what is in this black box from this library. So often you hear many more possibilities to be more accurate, etcetera. So try it out the next time when you are, having here an e squared c module and grab the data sheet. And, yeah, try a few things out, investigate.

ARDUINO UNO TO ARDUINO UNO

Let's create our own e squared c network with 2 other owners. And the main tasks will be now, make it a little bit bigger, that we have here one master that the master sends to the slave. The data, for example, gives me what you have and the slave just creates any kind of random number and sends them back to the master and the master the data. So this should simulate that the master sends to a module and commands, and this module is sending something back. It also can be used for some internal, controlling, etcetera, because you can have here not only one slave, you can have many slaves, 128, for example, with the 7 bit addressing, and then you could, getting a data them, etcetera, as we did it also with you at our SBI examples.

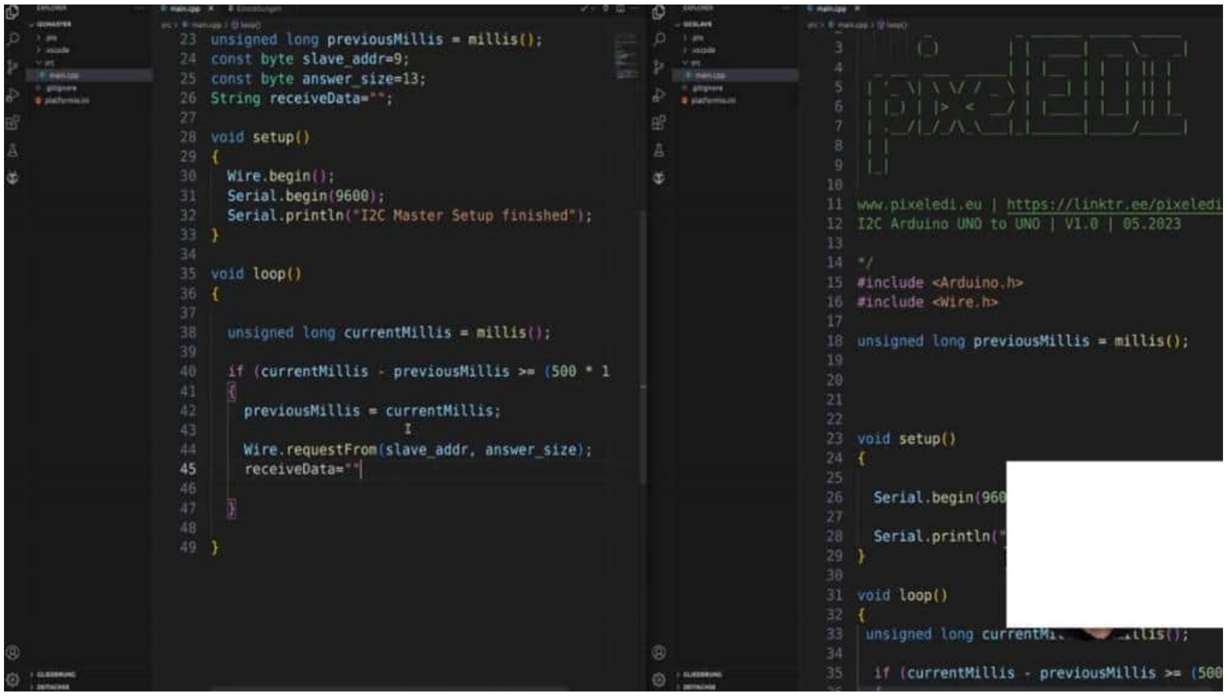


So the wiring part is also very easy because we are using A5 to A5, A 4 to A 4, and ground to ground. That's it. Nothing more is needed. Easy as it is. So on the left side, we have here our master sketch, and I would like to start with the master.

So we are implementing the wire dot h. And, wires begin, and we have seen 0 begin from 9600. This is sent out, you know, in an environment very common. And we squared our master setup here so that we know we are ready to go. So before going further on, we have to declare here some variables, some global variables that I would say, constraint, we are making in slave address. For example, 9, you could also use here an hex, but I would like to also, just use here the address line.

Then we could say const byte answer size. This should be defined here, and I set it to fix to some random string with certain lengths of 13. Then we have here and string, received data so that this is also globally. Content will be stored in our data from the slave. Okay.

And now we are fetching data each 500 milliseconds. And we are saying wire requests from. Here we have the slave address, nothing new for us, and this should be the answer size. In this case, 16 in the lengths, function call, then we received the data. So we have to delete it because we are getting each loop, new data, then we are saying, while a wire dot is available, also this should be known from before.



```cpp
unsigned long previousMillis = millis();
const byte slave_addr=9;
const byte answer_size=13;
String receiveData="";

void setup()
{
  Wire.begin();
  Serial.begin(9600);
  Serial.println("I2C Master Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (500 * 1
  {
    previousMillis = currentMillis;

    Wire.requestFrom(slave_addr, answer_size);
    receiveData=""

  }

}
```

```cpp
www.pixeledi.eu | https://linktr.ee/pixeledi
I2C Arduino UNO to UNO | V1.0 | 05.2023

*/
#include <Arduino.h>
#include <Wire.h>

unsigned long previousMillis = millis();



void setup()
{

  Serial.begin(960

  Serial.println("
}

void loop()
{
 unsigned long currentMi   illis();

  if (currentMillis - previousMillis >= (500
```

And now we are creating character for character. So let's try it out when we get a character. This is wire read. And, we are converting the character. So the receive date date is, we are concatenating it with some string.

So is it correct? Yes. So we're making a conversion to a string and saving it to the receive data so that we have a string. And outside these wires we can print out this value. serial print line receives data.

So, and now, nothing new is, for comes to, let me see. No. Nothing new is mentioned here because everything we did also before. So let's check and let's upload it. So there is nothing special because they can't receive anything because the slave isn't programmed yet.

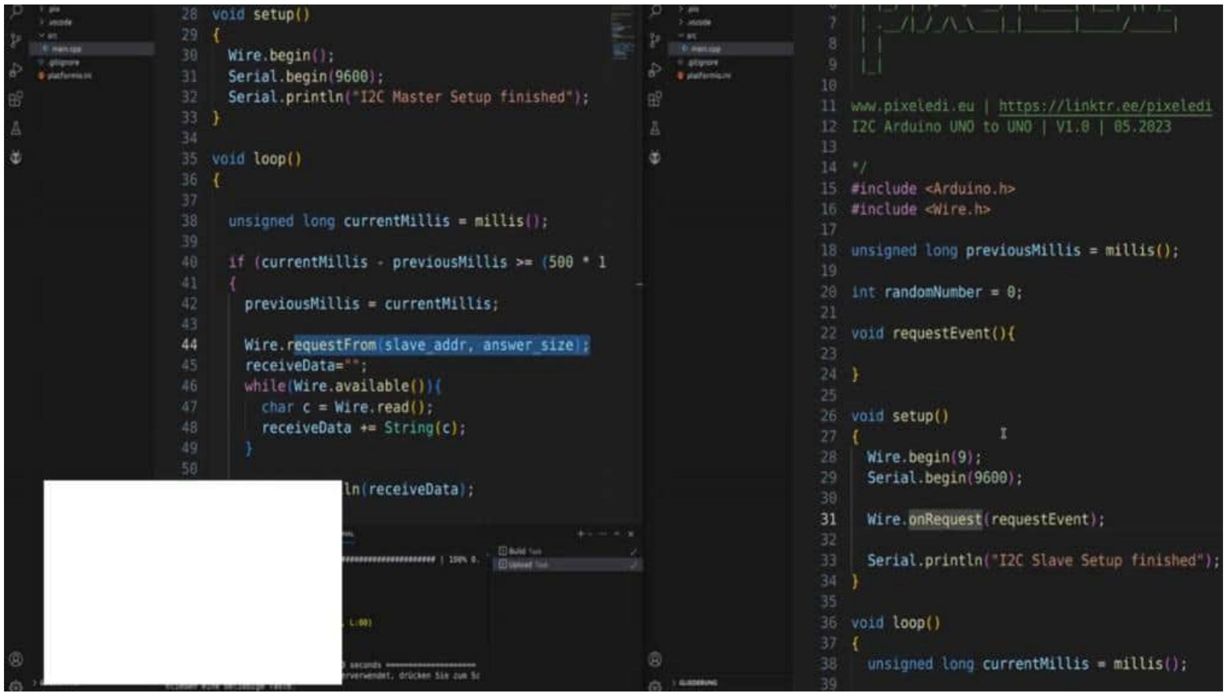
So I'm switching back to the left side and now we are coding here on the slave. So the slave side is pretty much the same, but we have to do here, let's make a global variable for a random number because we want to send you random numbers that we see that really each communication is sent. Therefore, we start here, the wire begins. And this is now our module, and we are starting it with getting here, inserting in the number and the address. We defined here the address with 9.

So we are typing in here. The wire begins on the address 9. So we'll begin, 9600, and then we are seeing here in the setup. Why are there requests? That means if we are getting all the slaves and requests, then the following function should be invoked. request event.

And this request event is great. until now. So we are creating a new function here, and this function will be called when there is a request like here. So then I would like to set here and random seats we have really each time if we start our new, random and then slave setup finished. I think that's it for the setup.

Let's focus a little bit more on the loop part. And now we are saying, for example, previous milliseconds equals to current milliseconds, and we are saying random now equals to random, let's make 3 numbers here because we have set the size fixed to 13. So we have to do three numbers here. Another request event now, we make a string, and we are saying pixeladienpepperpoint plus string random number. So let me see.

1, 2, 3, 4, 5, 6, 7, 8, 9, 10 plus three numbers are 13 perfect. So we are running that in an overflow. Then we are saying wire, right, and we are making it here, and text 2. And I create here not a string, but we are transmitting chars. That's the reason why I'm reading here also the chars, but it should be in texts.



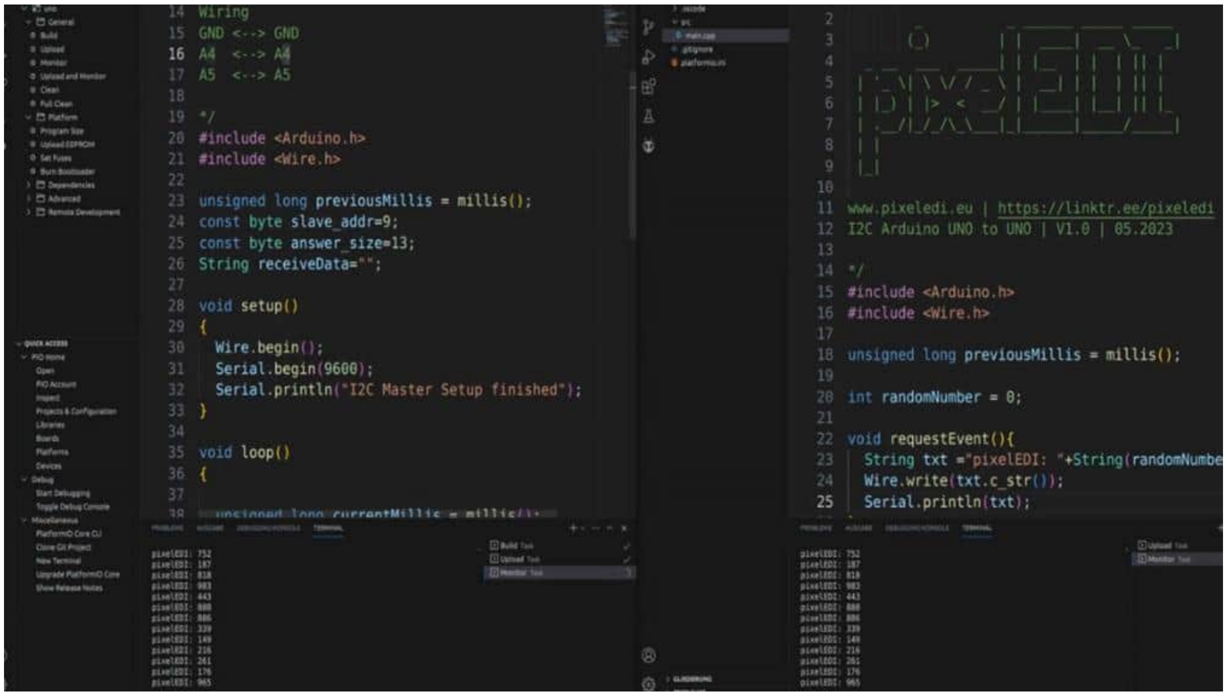
```
void setup()
{
  Wire.begin();
  Serial.begin(9600);
  Serial.println("I2C Master Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (500 * 1
  {
    previousMillis = currentMillis;

    Wire.requestFrom(slave_addr, answer_size);
    receiveData="";
    while(Wire.available()){
      char c = Wire.read();
      receiveData += String(c);
    }

                 ln(receiveData);
```

```
| .__/|_/_/\_\___|_|_____|_____/_____|
| |
|_|

www.pixeledi.eu | https://linktr.ee/pixeledi
I2C Arduino UNO to UNO | V1.0 | 05.2023

*/
#include <Arduino.h>
#include <Wire.h>

unsigned long previousMillis = millis();

int randomNumber = 0;

void requestEvent(){

}

void setup()
{
  Wire.begin(9);
  Serial.begin(9600);

  Wire.onRequest(requestEvent);

  Serial.println("I2C Slave Setup finished");
}

void loop()
{
  unsigned long currentMillis = millis();
```

And we are printing out our text because we want to check if everything works with our transmission. Okeydoke. Let's check and upload the code. And when we have uploaded everything, then we can see, our slave is transmitting the data. And before it's not a 4, a 4, we have of course, switched them.

So I've shown you this before of course. So, a 5 to a 5, a 4 to a 4, and then it should work properly. So here is the slave. The slave sends out 996, and we're getting here. on the master.

So I'm restarting it once again so that we see that it works. Here's the e squared c master. Here's the slave set up. And I've changed the USB, USB 0, USB 1, and then I can open here, both serial monitors here. And as you can see, it works quite well when we transmit some data here.



```
Wiring
GND <--> GND
A4  <--> A4
A5  <--> A5

*/
#include <Arduino.h>
#include <Wire.h>

unsigned long previousMillis = millis();
const byte slave_addr=9;
const byte answer_size=13;
String receiveData="";

void setup()
{
  Wire.begin();
  Serial.begin(9600);
  Serial.println("I2C Master Setup finished");
}

void loop()
{

  unsigned long currentMillis = millis();
```

```
pixelEDI: 752
pixelEDI: 187
pixelEDI: 818
pixelEDI: 983
pixelEDI: 443
pixelEDI: 888
pixelEDI: 886
pixelEDI: 339
pixelEDI: 149
pixelEDI: 216
pixelEDI: 261
pixelEDI: 176
pixelEDI: 965
```

```
www.pixeledi.eu | https://linktr.ee/pixeledi
I2C Arduino UNO to UNO | V1.0 | 05.2023

*/
#include <Arduino.h>
#include <Wire.h>

unsigned long previousMillis = millis();

int randomNumber = 0;

void requestEvent(){
  String txt ="pixelEDI: "+String(randomNumbe
  Wire.write(txt.c_str());
  Serial.println(txt);
```

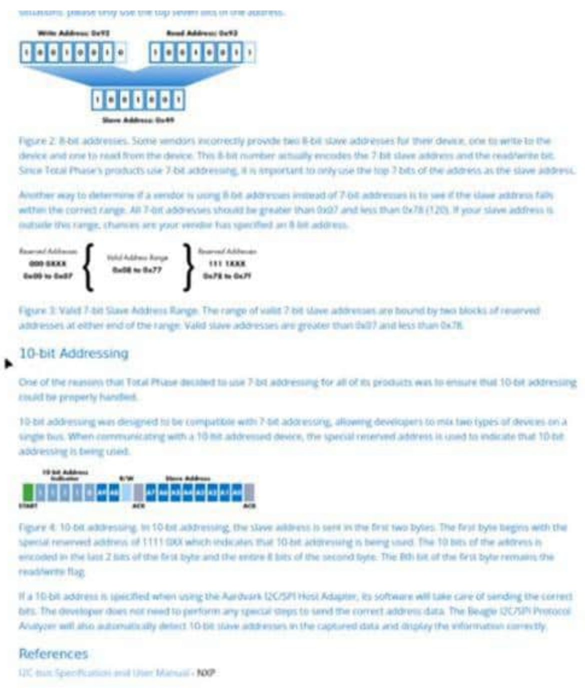
```
pixelEDI: 752
pixelEDI: 187
pixelEDI: 818
pixelEDI: 983
pixelEDI: 443
pixelEDI: 888
pixelEDI: 886
pixelEDI: 339
pixelEDI: 149
pixelEDI: 216
pixelEDI: 261
pixelEDI: 176
pixelEDI: 965
```

Yeah. And this is also done with the modules. So, this should just show you what it's, how can we do this, how is it possible, And when you want to send you some commands, just receive the commands, Kambulia, etcetera, you can turn on your LEDs lights. you can also, send here some text, for example, for visualization as you might like to do with e squared c.

OVERVIEW OF I2C ADDRESSES

Let's talk a little bit more about the e squared c addresses. So until now, we only used 7 bit addresses here because the last one was redone. Right? And this is absolutely enough for our area and the arena area because why should we use more than 128 devices. But if you would like to go a little bit deeper in the rabbit hole of e square t addresses and if it is a good internet address for you.

So you can check out what's all about it. And keep in mind with 10 bits you have the ability to go up to 1024 theoretical devices. You always have to think about why this is necessary in which use case? Is it really necessary that I have 1000 devices in, my Arlino command addressed, but maybe you have some use cases. And, therefore, you can check it out.



situations, please only use the top seven bits of the address.

Figure 2: 8-bit addresses. Some vendors incorrectly provide two 8-bit slave addresses for their device, one to write to the device and one to read from the device. This 8-bit number actually encodes the 7-bit slave address and the read/write bit. Since Total Phase's products use 7-bit addressing, it is important to only use the top 7 bits of the address as the slave address.

Another way to determine if a vendor is using 8-bit addresses instead of 7-bit addresses is to see if the slave address falls within the correct range. All 7-bit addresses should be greater than 0x07 and less than 0x78 (120). If your slave address is outside this range, chances are your vendor has specified an 8-bit address.

Figure 3: Valid 7-bit Slave Address Range. The range of valid 7-bit slave addresses are bound by two blocks of reserved addresses at either end of the range. Valid slave addresses are greater than 0x07 and less than 0x78.

10-bit Addressing

One of the reasons that Total Phase decided to use 7-bit addressing for all of its products was to ensure that 10-bit addressing could be properly handled.

10-bit addressing was designed to be compatible with 7-bit addressing, allowing developers to mix two types of devices on a single bus. When communicating with a 10-bit addressed device, the special reserved address is used to indicate that 10-bit addressing is being used.

Figure 4: 10-bit addressing. In 10-bit addressing, the slave address is sent in the first two bytes. The first byte begins with the special reserved address of 1111 0XX which indicates that 10-bit addressing is being used. The 10 bits of the address is encoded in the last 2 bits of the first byte and the entire 8 bits of the second byte. The 8th bit of the first byte remains the read/write flag.

If a 10-bit address is specified when using the Aardvark I2C/SPI Host Adapter, its software will take care of sending the correct bits. The developer does not need to perform any special steps to send the correct address data. The Beagle I2C/SPI Protocol Analyzer will also automatically detect 10-bit slave addresses in the captured data and display the information correctly.

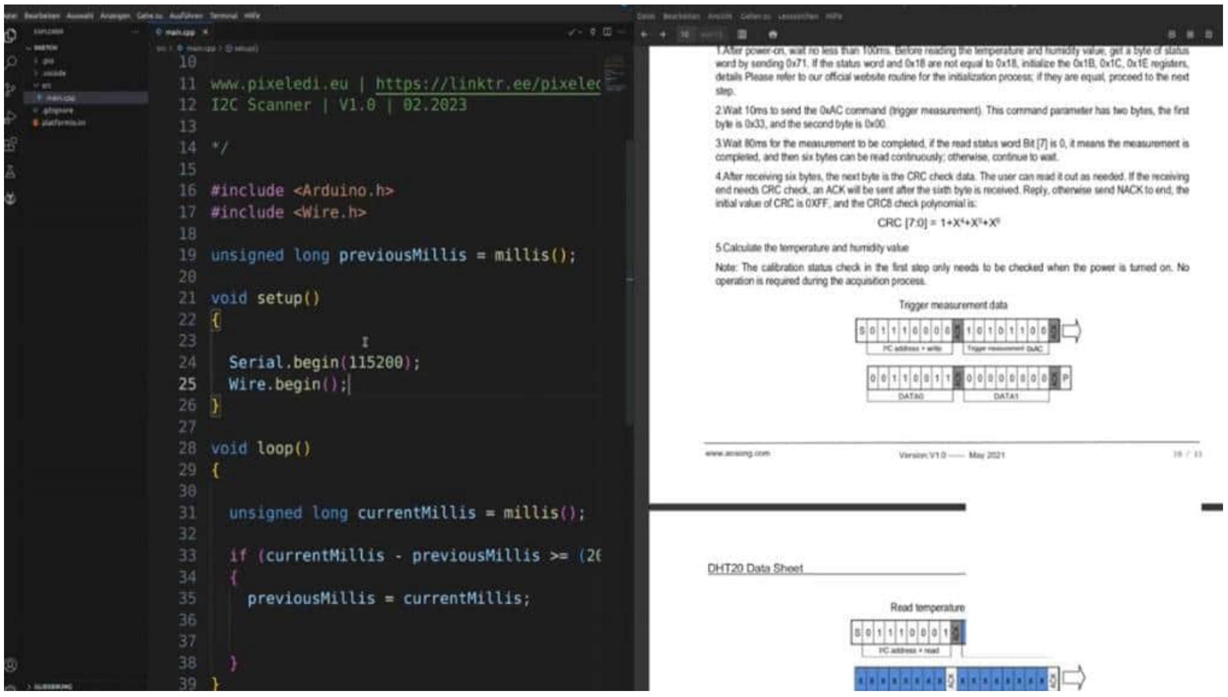
References

I2C-bus Specification and User Manual - NXP

Also, another reference I would like to show you is the adafruit square dc address list, which ator food makes really a lot of sensors. And also the clones from that are using the same e square address and defragments scroll a little bit down and can see where and what e squared c scheme is used for what kind of purpose. You can define your own e squared to address, what we did in the example before with the arduino and arduino communication. And in the next step, we want to create our own sketch where we find out e squared c addresses in an existing sketch because this could probably be handy. when we want to investigate an existing circuit, what is this, what kind of e squared c device is that, what is the address, etcetera.

CREATE OWN SKETCH FOR READING I2C ADDRESSES

In this project, I would like to create with you and Sketch so that we can find out all e squared c addresses in an existing circuit. So here we have our Sketchform before with 3 devices, and I would like to get here all of the e squared c addresses. And this could be handy, as I mentioned before. Maybe you have an existing circuit. You don't really know the e squared c address and just want to use the library, etcetera.



```

www.pixeledi.eu | https://linktr.ee/pixelec
I2C Scanner | V1.0 | 02.2023

*/

#include <Arduino.h>
#include <Wire.h>

unsigned long previousMillis = millis();

void setup()
{

  Serial.begin(115200);
  Wire.begin();
}

void loop()
{

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (2(
  {
    previousMillis = currentMillis;


  }
}
```

1.After power-on, wait no less than 100ms. Before reading the temperature and humidity value, get a byte of status word by sending 0x71. If the status word and 0x18 are not equal to 0x18, initialize the 0x1B, 0x1C, 0x1E registers, details Please refer to our official website routine for the initialization process; if they are equal, proceed to the next step.

2.Wait 10ms to send the 0xAC command (trigger measurement). This command parameter has two bytes, the first byte is 0x33, and the second byte is 0x00.

3.Wait 80ms for the measurement to be completed, if the read status word Bit [7] is 0, it means the measurement is completed, and then six bytes can be read continuously; otherwise, continue to wait.

4.After receiving six bytes, the next byte is the CRC check data. The user can read it out as needed. If the receiving end needs CRC check, an ACK will be sent after the sixth byte is received. Reply, otherwise send NACK to end, the initial value of CRC is 0XFF, and the CRC8 check polynomial is:

$$CRC\ [7:0] = 1+X^4+X^5+X^8$$

5.Calculate the temperature and humidity value

Note: The calibration status check in the first step only needs to be checked when the power is turned on. No operation is required during the acquisition process.

www.aosong.com Version: V1.0 —— May 2021

DHT20 Data Sheet

So we can do this on our own very, very easily. Therefore, we have to, at here, the wire.h And we are starting the communication after the serial begins. For example, with wire dots begin. Nothing special until now. And what we are doing now is this: I have just opened the

DHT 20 data sheet again because We're starting the communication, and then we are sending out different kinds of addresses.

We are iterating through 1, example, 1, 2, 3, 4, 5, 6, etcetera, and we are checking if we're getting an acknowledgement. If we get an acknowledged back, we know, there is such a device out there. And therefore, this is an easy task for So let's start by, in the loop by defining here two variables, for example, with byte and result. And then we are making a 4 loop here. So let's see if I can make it here.

Yes. 4. And we are saying here 4. Address is 0 because we won't inter iterate through all of the addresses. And we are seeing that the address should be lower than 127.

We want to iterate through 128. So 7 bits And we are saying address plus plus. This should be our for loop. And then we could say there, why don't we begin transmission as we did before, and we're posting the address here. So we don't need here, this will be interpreted as hex and therefore, the works correctly.

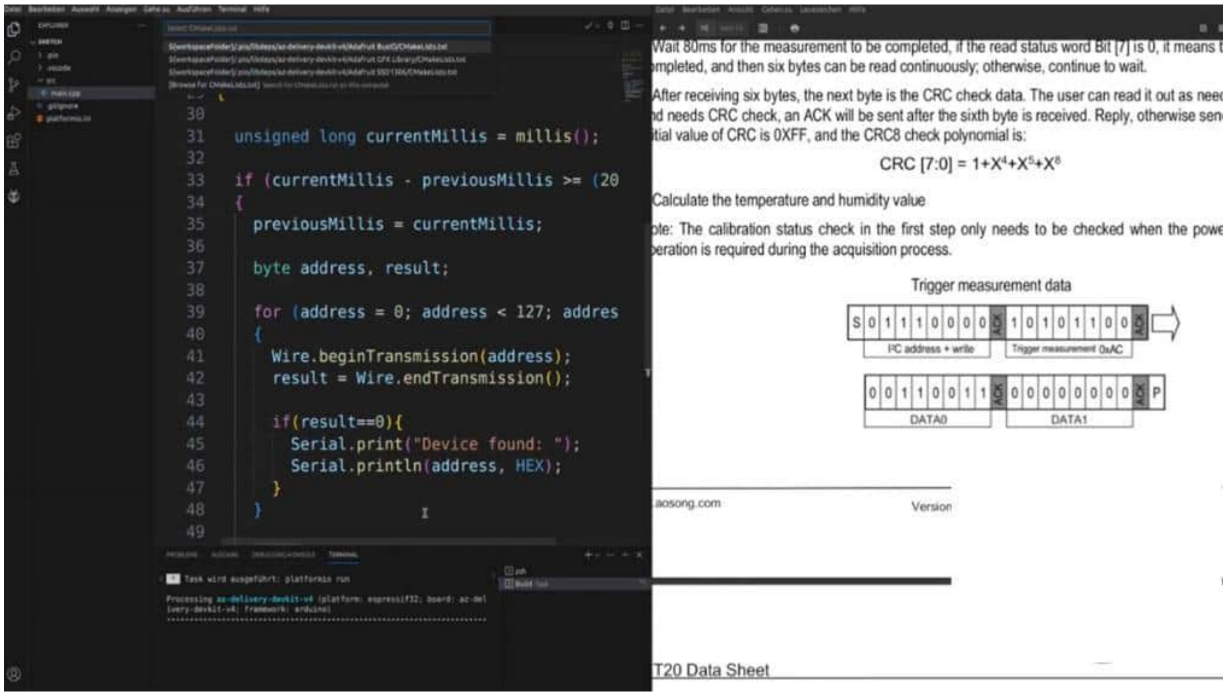
And then we're getting back and results so they acknowledge, and this will be sent back by the wire and transmission. And now we could say, and this should have to be done here. If the result is 0, then we know that there should be an address. And this could be printed out by address as hex radio. So, we could say 0 print device found.

And with some double d, and this is now our main sketch. So let me see if it's correct inside the form. We should have the if and outside the 4 we made and delay. This is a bad delay, but it works quite well. In this case, we don't need anything more.

Let's check if the compiler found some errors. some typos. No. Everything is good. Then let's upload it.

Let's check it out. And there we go. We found a device. 2338 3c. So 3c is, if I don't mix it up, the OLED, then we have 30 should be this 1 and 23 this or what else we are?

I have it here. 38 is the DHT 20 and then is 20 3, the BH1750, looking good, then let's check to hear the pulse for the logic analyzer or if we get some data and what it looks like, then we're zooming a little bit in. So Here is the full range of data that we are transmitting starting at the left side. We are sending out the address 00, and we are writing that, and we're getting back and not acknowledged because there is no device with the address 00 in our circuit. also is 1, 2, etcetera.



```
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= (20
  {
    previousMillis = currentMillis;

    byte address, result;

    for (address = 0; address < 127; addres
    {
      Wire.beginTransmission(address);
      result = Wire.endTransmission();

      if(result==0){
        Serial.print("Device found: ");
        Serial.println(address, HEX);
      }
    }
```

```
Task wird ausgeführt: platformio run

Processing az-delivery-devkit-v4 (platform: espressif32; board: az-delivery-devkit-v4; framework: arduino)
--------------------------------------------------------------------------------
```

Wait 80ms for the measurement to be completed, if the read status word Bit [7] is 0, it means t
mpleted, and then six bytes can be read continuously; otherwise, continue to wait.

After receiving six bytes, the next byte is the CRC check data. The user can read it out as nee
d needs CRC check, an ACK will be sent after the sixth byte is received. Reply, otherwise sen
tial value of CRC is 0XFF, and the CRC8 check polynomial is:

$$CRC\ [7:0] = 1+X^4+X^5+X^8$$

Calculate the temperature and humidity value

ote: The calibration status check in the first step only needs to be checked when the powe
eration is required during the acquisition process.

Trigger measurement data
S 0 1 1 1 0 0 0 0 ACK 1 0 1 0 1 1 0 0 ACK
I²C address + write
Trigger measurement 0xAC
0 0 1 1 0 0 1 1 ACK 0 0 0 0 0 0 0 0 ACK P
DATA0
DATA1

aosong.com Version

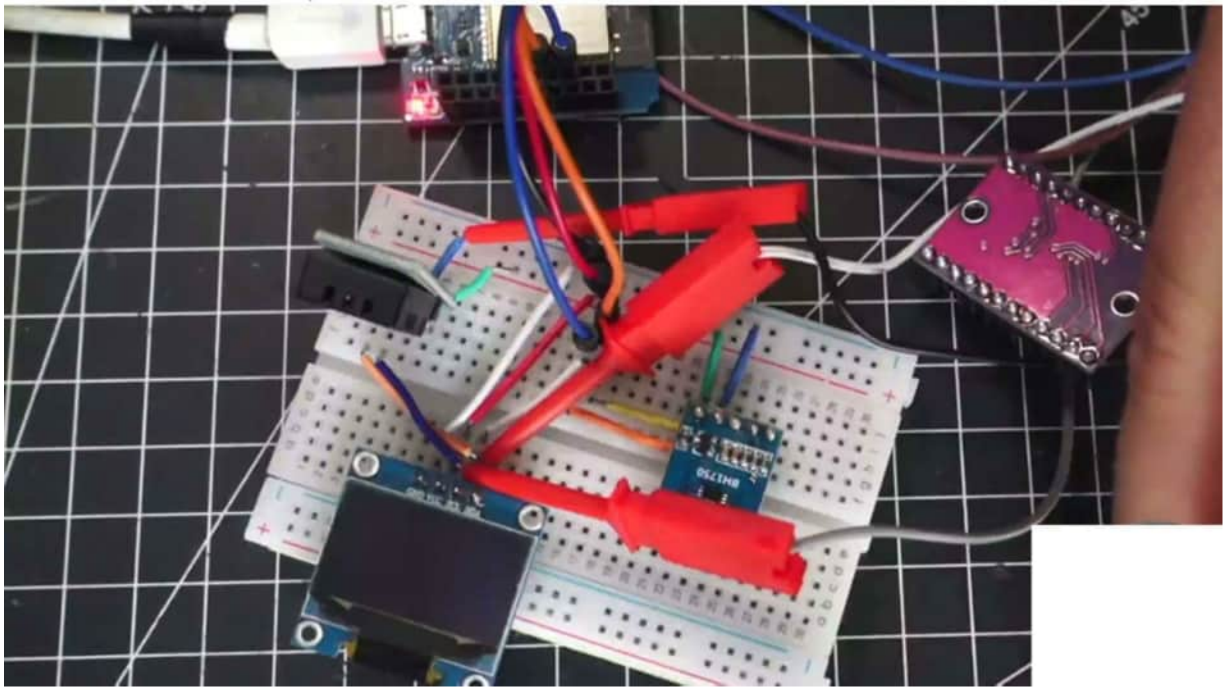
T20 Data Sheet

So our first device was 23. So let's switch here a little bit to the right. 22 is not acknowledged. 23 is acknowledged because it's there. And this same should be done by 38.

Yes, Otis is not acknowledged. And this is what we do here. We are sending you the raw bytes for the address for the e squared c address and waiting if we are getting back and acknowledged or not. Easy as it is to understand how the whole E squared C communication works and how easily we can create -- our own e squared scanner.

I2C MULTIPLEXER

One last thing, and then we are at the end of this chapter, I would like to show you an e squared c multiplexer. And this can be used when you need from the same device, a few of them in 1 of your circuits. Because if you want to use 2 of these, OLEDs, for example, then this is not really easy to do because so let's see if they're, if I'm getting in focus because the e squared c address stands here on this side, 0, 3 c, we can solder out this little resistor to the other side and then we're getting here an circuit because they have the same e squared c address. With this multiplexer, as the multiplexer says, they can handle the communication for you, giving them an internal address. And so you can use 7 devices with 1 multiplex here and you can cascade them.



So you could have many more devices here. And how you can use it here, I refer to a really good site from random notes to terrors. And this will show you how you can use this multiplex or this DCA. You can find all of these devices in my component list with some links on it. And when I go down, for example, creating multiple OLED displays or also using, for example, multiple BM sensors, like we can see here and with this multiplexer, it's easy to do it with only one microcontroller.

So keep in mind, you need more e squared c from the same type, use a multiplexer.