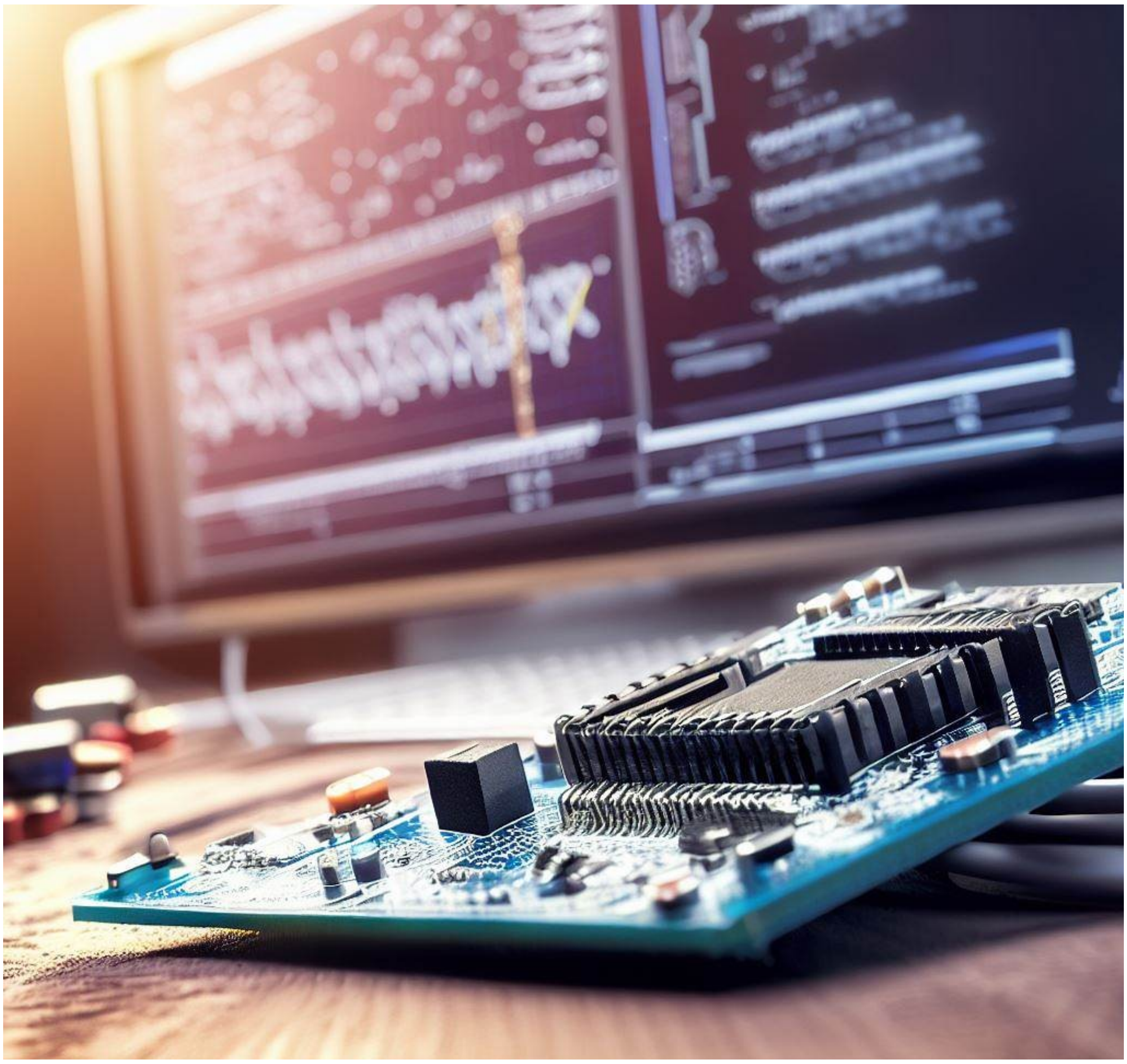


Table of content

1. Prologue

Welcome to the topic of hardware protocols in the Arduino environment! In this book you will learn all about the different communication protocols that play an important role in the world of microcontrollers and especially in the Arduino ecosystem. From serial communication to I2C and SPI we explore the basics, structure and possible applications of each protocol. Whether you are a beginner looking to learn the basics or an experienced Arduino developer looking to deepen your knowledge, this book will help you understand and use the many possibilities of hardware communication in the Arduino environment.

It's nice that there are more people who want to understand more about the world of 0s and 1s. I am sure you have used and applied various protocols many times, but now we want to dive one level deeper together and analyse, understand and even write our own evaluations for temperature sensors. Why all this? Besides training our understanding, it is also a good exercise in reading and understanding data sheets. We also broaden our horizons and learn a bit about C++, bit operators and analysing hardware components.

So let's get started and start analysing the hardware protocols in the Arduino environment.

2. Basics Logic Analyzer

2.1. Oscilloscope vs. Logic Analyzer

An oscilloscope and a logic analyser are two different devices used in electronics and signal analysis. Here are the differences between them:

Oscilloscope:
An oscilloscope is a measuring instrument used to graphically display electrical signals. It measures and shows the variation of a voltage over time. It allows you to visualise signals in the form of waveforms so that you can read information such as amplitude, frequency, phase shift and signal stability. An oscilloscope is particularly useful for analysing periodic or repetitive signals.

Logic Analyzer:
A logic analyser is a measuring instrument specially designed for analysing digital signals. It enables the observation and recording of digital signals as they occur in computer systems, microcontrollers or logic circuits. The logic analyser displays the logic levels (high or low) of the individual signal lines and enables the analysis of digital protocols, such as I2C, SPI, UART, etc. With a logic analyser you can investigate timing problems, signal delays, bus communication and logic states.

In summary, an oscilloscope enables the temporal representation of analogue signals, while a logic analyser specialises in digital signals and analyses logic levels and digital protocols. Both devices have different areas of application and can be used depending on the requirements of the specific measurement task.

2.2. Features of Logic Analyzern

The serious difference between inexpensive and more expensive logic analysers lies in the quality, functions and performance of the devices. Here are some of the main differences:

1. Sampling rate and number of channels: More expensive logic analysers usually offer a higher sampling rate and a larger number of channels. A higher sampling rate enables the acquisition of fast signals with more detail, while a larger number of channels enables the simultaneous acquisition of multiple signals.
2. Memory depth: Memory depth refers to the amount of data the logic analyser can capture and store. More expensive devices often have a greater memory depth, which allows for longer recording time, especially when analysing long signal traces.
3. Real-time analysis: While cheaper logic analysers are often limited to capturing data and analysing it afterwards, more expensive devices often offer real-time analysis

functions. This means that the acquired data can be processed and displayed in real time, enabling rapid diagnosis and analysis.

4. Protocol decoding: More expensive logic analysers often offer more extensive protocol decoding functions for various serial communication protocols such as UART, I2C, SPI, CAN, etc.. This makes it easier to analyse and interpret the communication data.
5. Software functions: More expensive logic analyzers often come with more comprehensive and powerful software that offers advanced analysis functions, custom trigger options, protocol analysis and other advanced tools.
6. Build quality and durability: More expensive logic analysers tend to have better build quality and are more robust. They are often designed for professional use and offer greater reliability and durability.

It is important to note that the choice of Logic Analyzer depends on the specific requirements of the project and the available budget. For simple tasks with low requirements, an inexpensive logic analyser may be sufficient, while for demanding projects and professional applications, a more expensive device with advanced features may be preferred.

2.3. Practical applications

1. Understanding protocols and data transmission
2. Signal acquisition: A logic analyser can be used to record the digital signals in a system. This makes it possible to capture the exact timing of the signals and analyse them. By examining the recorded signals, irregularities or errors can be identified.
3. Trigger conditions: A logic analyser allows the configuration of trigger conditions that determine when the acquisition of signals starts. By setting appropriate trigger conditions, the logic analyser can automatically record the signals when certain events or conditions occur. This makes troubleshooting more efficient because only relevant signals are recorded.
4. Temporal analysis: With a logic analyser, the temporal sequence of signals can be analysed. By examining signal propagation times and changes, temporal relationships between different signals can be determined. This can help identify timing problems or synchronisation errors.
5. Protocol analysis: A logic analyser enables decoding and analysis of various communication protocols such as UART, SPI, I^{2}C, CAN, etc. By capturing and decoding the transmitted data, the logic analyser can help identify faulty or incorrect communication. This is particularly useful when problems occur during data transmission.

6. Analysis of the captured data: After the signals have been captured, they can be analysed in detail. This can include looking at signal levels, checking signal quality, finding patterns or identifying errors. By visualising the captured data and understanding the context, it is easier to detect errors or irregularities.

By combining these techniques, a logic analyser can be used effectively in troubleshooting to identify and correct problems in digital circuits or communication protocols. However, it is important to have a basic understanding of the system and the signals to be analysed in order to use the logic analyser effectively and interpret the data correctly.

3. Bitwise operations

```
byte x = 0b10000111;
 Serial.println(x, BIN);
 Serial.println(x, HEX);
 Serial.println(x, DEC);
```

3.1.Bitwise AND (&)

The bitwise AND operation in the Arduino environment performs a logical AND operation on the bits of two variables or constants. Each bit of the result is 1 if both corresponding bits in the inputs are also 1, otherwise it is 0. The result is often used to manipulate individual bits or to check certain bit patterns.

An example of the bitwise AND operation in the Arduino environment is to check certain conditions in a status register. Suppose we have a status register with 8 bits and want to check whether a certain bit is set. By applying the bitwise AND operation between the status register and a mask containing only the desired bit, we get either 0 (if the bit is not set) or a value greater than 0 (if the bit is set). This allows us to check conditions and perform appropriate actions.

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

  byte statusRegister = 0b11011011;
  byte mask = 0b00000100;          // Mask third bit

  // Bitweise AND-Operation
  byte result = statusRegister & mask;

  // Überprüfen, ob das dritte Bit gesetzt ist
  if (result > 0) {
    Serial.println("The third bit is set!");
  } else {
    Serial.println("The third bit is not set!.");
  }
}

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

In this example, the status register is initialised with the value B11011011. We want to check whether the third bit (counting from the right) is set. For this, we use a mask B00000100 that contains only the third bit and sets all other bits to 0. Through the bitwise AND operation between the status register and the mask, we get either 0 or a value greater than 0. If the third bit is set, the corresponding message is output via the serial interface

3.2.Bitwise OR (|)

The bitwise OR operation in the Arduino environment performs a logical OR operation on the bits of two variables or constants. Each bit of the result is 1 if at least one of the corresponding bits in the inputs is 1. The result is often used to set or combine bits.

Example of the bitwise OR operation in the Arduino environment:

Suppose we have a variable a with value B101010 (170 in binary representation) and a variable b with value B11001100 (204 in binary representation). We want to perform the bitwise OR operation between a and b and store the result in a variable result. The code looks like this:

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

  byte a = B10101010; // 170 in binary
  byte b = B11001100; // 204 in binary

  // Bitweise OR-Operation
  byte result = a | b;

  //Result
  Serial.println(result, BIN);
}

void loop() {
  // nothing
}
```

In this example, the bitwise OR operation between the variables a and b is performed. The result is stored in the variable result. The binary representation of the result is output via the serial interface. The result of the bitwise OR operation is B11101110 (238 in binary representation), since each bit of the result is set if at least one of the corresponding bits in a or b is set.

3.3.Bitwise Linksverschiebung (<<)

The bitwise left shift (<<) in the Arduino environment shifts the bits of a number a certain number of places to the left. In the process, the empty places at the right end are filled with zeros. This operation corresponds to multiplying the number by 2 to the power of the number of shift places.

Example of bitwise left shifting in the Arduino environment:

Suppose we have a variable a with the value 5 (B00000101 in binary notation) and we want to shift the bits 2 places to the left. The code looks like this:

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

  byte a = 5; // binary: B00000101

  // Bitweise left
  byte result = a << 2;

  //  resultn
  Serial.println(result, BIN); // 10100
}

void loop() {
  // nothing
}
```

In this example, the bitwise left shift of the variable a by 2 places is carried out. The result is stored in the variable result. The result of the shift is 20 (B00010100 in binary representation) because the bits were shifted 2 places to the left and the empty places at the right end were filled with zeros.

Use cases

The bitwise left shift (<<) in the Arduino environment is used in various situations, for example:

- Multiplication by a power of two: by shifting left by a certain number of digits, multiplication by a power of two can be done more efficiently. Instead of multiplying the number by 2 to the power, it can simply be shifted to the left by the corresponding number of digits.
- Manipulation of individual bits: By shifting left, certain bits in a number can be set or deleted. By shifting a mask with a set bit a certain number of places to the left and then performing the bitwise OR operation with the original number, the desired bit can be set.
- Bit-based storage: When working with memory-intensive applications, such as LED controls or display drivers, bitwise left shifting can be used to efficiently utilise memory space. By positioning bits at specific positions within a number and shifting them left accordingly, multiple states or options can be stored in a single byte or larger data structure.

Bitwise left shift is a useful operation to achieve efficiency and compactness in programming microcontrollers in the Arduino environment.

3.4. Bitwise right shift (>>)

The bitwise right shift (>>) in the Arduino environment shifts the bits of a number a certain number of places to the right. In the process, the empty places at the left end are filled with zeros. This operation corresponds to dividing the number by 2 to the power of the number of shift places (for positive numbers).

Example of bitwise right shifting in the Arduino environment:

Suppose we have a variable a with a value of 16 (B00010000 in binary notation) and we want to shift the bits 3 places to the right. The code looks like this:

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

  byte a = 16; // binary: B00010000

  // Bitwise shift right
  byte result = a >> 3;

  // result ausgeben
  Serial.println(result, BIN);
}

void loop() {
  // nothing
}
```

In this example, the bitwise right shift of the variable a by 3 places is carried out. The result is stored in the variable result. The result of the shift is 2 (B00000010 in binary notation) because the bits were shifted 3 places to the right and the empty places at the left end were filled with zeros. Note that for unsigned numbers, as in this example, the right shift is a logical right shift. For signed numbers, an arithmetic right shift is performed that preserves the sign bit.

Anwendungsfälle

The bitwise right shift (>>) in the Arduino environment is also used in various situations, including:

- Division by a power of two: By shifting to the right by a certain number of places, division by a power of two can be done more efficiently. Instead of dividing the number by 2 to the power, it can simply be shifted to the right by the corresponding number of digits.
- Extracting bit information: By shifting to the right, certain bits can be extracted from a number. By shifting a number to the right by a certain number of places and then performing the bitwise AND operation with a mask, specific bits can be isolated and checked.
- Signed right shift: When working with signed numbers, right shift is used to perform an arithmetic right shift. In this case, the sign bit is retained and the empty places are filled with the corresponding sign bit.

Bitwise right shift is a useful operation to optimise numerical calculations, isolate certain bits and correctly process signed values. It enables efficient code and better use of resources in microcontroller applications in the Arduino environment.

3.5. Hex to Decimal

- Assign decimal values to each digit according to its position. The rightmost digit represents the units place, the next digit to the left of it the 16s place, the one after that the 256s place and so on.
- For the digit 7 at the ones place, its decimal value is 7 × 16^0 = 7.
- For the digit 8 in the 16s place, its decimal value is 8 × 16^1 = 128.
- Add the decimal values of the digits together. In this case the result is7 + 128 = 135

3.6.Binary to Decimal

1. Determine the value of each digit in the binary number. The rightmost digit represents the ones place, the next digit to the left of that represents the twos place, the one after that represents the fours place, and so on.
2. Multiply the value of each digit by the corresponding power of 2 and add the results together.

Für die binäre Zahl 10000111:

1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0

= 128 + 0 + 0 + 0 + 0 + 4 + 2 + 1

= 135

Therefore, the binary number 10000111 corresponds decimally to the number 135.

4. PulseView

4.1.Why PulseView?

PulseView is a popular choice for a logic analyser for several reasons:

- Ease of use: PulseView offers a user-friendly interface that makes it easy to acquire, display and analyse signals. The software is intuitively designed and has a wide range of functions that facilitate signal analysis.
- Support for various protocols: PulseView supports a wide range of protocols, including well-known standards such as I2C, SPI, UART, CAN, USB and many others. This allows you to analyse and interpret different types of signals and buses.
- Hardware compatibility: PulseView is compatible with various logic analyser hardware devices and supports a wide range of interfaces such as USB, Ethernet and Wi-Fi. This means you can use PulseView with a wide range of hardware options, depending on your requirements and preferences.
- Real-time analysis: PulseView enables real-time analysis of signals, which is particularly useful for capturing time-critical events and fast signal trajectories. You can monitor and record signals in real time for accurate and detailed information.

- Open source software: PulseView is open source software, which means that the source code is freely available. This allows developers to customise and extend the software to meet their specific needs.

4.2.Configuration

https://www.sigrok.org/doc/pulseview/0.4.1/manual.html#_data_acquisition
Hier nochmal die basics wie fit to zoom etc durchgehen

4.3.Frequency and milliseconds

To calculate the number of switching operations in a time interval of 200 ms at a frequency of 5 Hz, proceed as follows:

- Determine the duration of a switching cycle: The frequency of 5 Hz indicates how many switching cycles take place per second. Since 1 Hz corresponds to one cycle per second, the duration of a switching cycle at 5 Hz is 1/5 seconds or 0.2 seconds (200 ms).

- Divide the duration of the time interval (200 ms) by the duration of one switching cycle (200 ms) per switching operation: 200 ms / 200 ms = 1.

The result is 1, which means that a switching operation takes place in a time interval of 200 ms at a frequency of 5 Hz.

In this case, if a sketch is used that switches between HIGH and LOW every 200 ms, the frequency is 5 Hz, and you have one switching operation in that time interval.

4.4.Samples and MHz settings

- • Sampling rate: A logic analyzer has a limited sampling rate that indicates how many data points per second it can capture. If the sampling rate is too low, fast signal changes or fine details may be lost. At 20 kHz sampling rate, the Logic Analyzer may not be fast enough to capture all relevant information, while at 12 MHz it is fast enough.
- • Ex: Arduino Uno and 20 kHz does not work.
- • Samples: To accurately capture and analyze the signal, you should capture a sufficient number of samples per signal period. To achieve this, you can increase the sampling rate. Suppose you want to capture the signal

at a frequency of 5 Hz (one signal every 200 ms). A good rule of thumb is to make the sampling rate at least ten times the frequency you want to capture. In this case, you could use a sampling rate of 50 Hz (50 samples per second) to analyze the signal in detail.

- • MHz setting: The MHz setting determines the maximum sampling rate at which PulseView can capture the signal. Since the Arduino Uno operates at a maximum clock frequency of 16 MHz, you should make sure that the MHz setting in PulseView is below this value. For example, you could choose a sample rate of 1 MHz (1 million samples per second) to accurately capture the signal from the Arduino Uno.

It is important to note that the exact settings of PulseView may also depend on other factors, such as the desired resolution, the number of channels to be analyzed, and the memory and resource limitations of your system. The above values are guidelines that can help you acquire and analyze a sufficiently accurate signal with PulseView based on the given situation.

4.5.Baud rate

n an Arduino environment, the baud rate can be figuratively described as the speed at which information is exchanged between the Arduino and other devices.

Imagine you are a student in a class and your teacher wants to communicate information about an assignment to you. The baud rate in this case would be the speed at which the teacher communicates the information. Suppose your teacher can say 4 words per second. That means the baud rate is 4 words per second.

If the baud rate is low, the teacher would speak slowly and you would have to listen carefully to catch all the information. In this case, it may take longer for you to fully understand the task.

When the baud rate is high, the instructor speaks very quickly and conveys the information to you at a fast pace. You would have to increase your concentration to keep up with the high speed and catch all the information.

The baud rate in the Arduino environment works similarly. When the baud rate is low, it takes longer to transmit data, but it can be easier to receive the data reliably. When the baud rate is high, data transfer is faster, but there is a higher risk of transmission errors if the connection is not stable.

In the Arduino environment, the baud rate is measured in bits per second (bps). For example, if the baud rate is 9600 bps, it means that the Arduino can send or receive 9600 bits per second. You can think of it like a stream of data flowing at a certain speed.

The higher the baud rate, the faster data can be transferred between the Arduino and other devices. A higher baud rate allows faster communication, but it also requires a more stable and reliable connection to ensure that data can be transferred correctly.

Common baud rates in the Arduino environment are 9600 or 115200.

4.6.PWM example

What is the difference in these three lines?

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

1. 1. analogWrite(pwm1, 20);: This instruction uses the analogWrite function to generate an analog signal on a PWM (Pulse-Width Modulation) pin. The parameter pwm1 specifies the pin on which the PWM signal is to be generated, and the value 20 determines the strength of the signal. The value range is normally between 0 (off) and 255 (full strength). In this case the PWM signal is generated on pin pwm1 with a strength of 20.
2. 2. analogWrite(pwm2, 200);: This instruction is similar to the previous one, but generates a PWM signal on a different pin (pwm2) with a strength of 200. The value 200 indicates that the PWM signal in this case is much stronger than in the previous example.
3. 3. digitalWrite(digital, HIGH);: This instruction uses the digitalWrite function to set a digital pin to a specific state. The parameter digital specifies the pin on which the change is to be made and the value HIGH means that the pin is set to the logical state "HIGH" (one). This command switches the digital pin to "HIGH".

In summary, the analogWrite and digitalWrite functions of the Arduino Uno control the output of analog and digital signals on the corresponding pins. The values you specify for signal strength or state affect how much signal is output or what state the pin is set to.

4.7.Trigger

Sketch with infrared module

D0 setting -> rising edge and then trigger by hand

Keep hand over SEnsor longer and then move back and forth faster to better see changes from 0 and 1.

4.8.Export

- • As ASCI art logic ... 2 er entry
- • 0/ 1 digits
- • and *.sr to copy

4.9.Links

- PulseView from Sigrok https://sigrok.org/wiki/Main_Page
- Manual https://www.sigrok.org/doc/pulseview/0.4.1/manual.html#_data_acquisition

Explanation Data transmission rate https://www.eetimes.com/How-Much-Bandwidth-Does-Your-Logic-Analyzer-Need-/

5. UART

UART stands for Universal Asynchronous Receiver Transmitter and is a serial communication technique commonly used to transfer data between microcontrollers, computers and other electronic devices.

UART transmits data in the form of bits serially, which means that the individual bits are sent one after the other. There is no clock line to synchronize the transmission, so UART is referred to as asynchronous.

UART enables bidirectional serial communication between a transmitter and a receiver via two data lines: the TX line (Transmit) and the RX line (Receive). It is an asynchronous communication, which means that the transmitter and receiver do not need to be synchronized by a common clock signal.

The transmitter converts the serial data into an electrical voltage and sends it over the TX line. The receiver detects the voltage levels on the RX line and reconstructs the serial data. The start bit signals the beginning of a data packet, followed by the data bits and the stop bit, which marks the end of the data packet. The parity bit, if used, is used to check data integrity.

Communication via UART normally takes place at a specific baud rate (transmission rate), which determines the speed of the data transmission. Typical baud rates are for example 9600, 19200, 38400, 115200 etc.

To enable data transfer in UART, start and stop bits are used.

- Start bit: The start bit signals the beginning of a data packet. It is always a logical 0 bit (low). The start bit helps the receiver to synchronize the data flow and to recognize the start of the data packet.
- Data length: The data length in UART determines the number of bits used to transmit a single character. Typical data lengths are 5, 6, 7 or 8 bits. The data length must match between sender and receiver, otherwise the data cannot be interpreted correctly.
- Stop bit: The stop bit marks the end of a data packet and separates it from the subsequent start bits of the next data packet. It is always a logical 1 bit (High). The stop bit enables the receiver to synchronize the data flow and to recognize the end of the data packet.

The combination of start bit, data length and stop bit forms a frame, which represents a single data character and is transmitted between sender and receiver.

It is important to note that the exact configuration of UART (baud rate, data length, parity, number of stop bits) depends on the application, the devices and the selected UART implementation. These parameters must be correctly matched between the transmitter and receiver to ensure successful data transmission.

5.1.LSB & MSB

LSB and MSB are terms used in digital data processing to describe the place value system of binary numbers.

• LSB (Least Significant Bit) refers to the least significant bit of a binary number. It is the rightmost bit in the representation of the number.

• MSB (Most Significant Bit), on the other hand, refers to the most significant bit of a binary number. It is the leftmost bit in the representation of the number.

To illustrate the concept of LSB and MSB, here is an example:

Let's take an 8-bit binary number: 10101011.

In this case the LSB is the rightmost bit, i.e. 1, while the MSB is the leftmost bit, i.e. 1.

The position of each bit in the binary number has a certain value corresponding to the position of LSB and MSB.

If we calculate the value of the number, it would look like this:

Value = (1 2^7) + (0 2^6) + (1 2^5) + (0 2^4) + (1 2^3) + (0 2^2) + (1 2^1) + (1 2^0) = 128 + 0 + 32 + 0 + 8 + 0 + 2 + 1 = 171

In this example the LSB is 1 and has the value 1(2^0) while the MSB is also a 1 and has the value 128 (1 ^7).

LSB and MSB play an important role in the interpretation and manipulation of binary data because they represent the relative importance of each bit in a binary number.

5.2.Ikea air quality sensor Vindriktning

Data sheet:

https://pdf.directindustry.com/pdf/cubic-sensor-instrument-co-ltd/pm1006k-led-particle-sensor-module/54752-927719.html

5 byte *256 + 6 byte

LSB also 11001 => 10011 = 19

5.3.Find baud rate

9600 Baudrate = 1/9600 = 104µs

UART is widely used in many microcontrollers and peripherals and enables communication with a variety of external devices, such as sensors, displays, GPS modules, Bluetooth modules, and many others.

Example with ASCII

- Arduino Uno sends pixelEDI
- With decoder output view hex -> ascii
- set the correct baud rate
- needs only one channel because we do not return anything
- first several recordings and then importantly only one "pixelEDI" as *.sr export
- load configuration file
- still open - how file should be named so people know_d0=rx
  - https://www.rapidtables.com/convert/number/hex-to-ascii.html

5.4.Advantages and disadvantages of UART

UART (Universal Asynchronous Receiver-Transmitter) is a widely used serial communication technology that is used to transfer data between different devices. Here are some advantages and disadvantages of UART:

5.4.1. Advantages of UART

1. Simplicity: UART is a simple and widely used communication technology. Implementing UART in hardware and software is comparatively simple and requires fewer resources than complex protocols such as Ethernet or USB.
2. Widely used: UART is present in a variety of microcontrollers, SoCs (system-on-chip), and other integrated circuits. It is a commonly used interface for communication with sensors, displays, peripherals, and other devices.
3. Speed: UART can operate at different baud rates and allows data to be transferred at different speeds. It provides a flexible data rate that can be adjusted according to application requirements.
4. Real-time communication: UART enables asynchronous serial communication where data can be transferred in real time. This is particularly useful for real-time applications where fast transmission and processing of data is required.
5. Compatibility: UART is compatible in many different systems and platforms due to its widespread use and simplicity. It can be easily used between different devices and microcontrollers as long as the communication parameters (such as baud rate and parity bit) match.

5.4.2. Disadvantages of UART:

1. Point-to-point communication: UART only allows communication between two devices (point-to-point). It does not inherently support multidrop communication, where multiple devices are connected to the same bus.

2. Limited transmission distance: UART is designed for short to medium transmission distances. It is not suitable for long transmission paths or communication over long distances, as it is susceptible to signal loss and interference.
3. No error detection or correction: UART does not provide built-in error detection or correction functions. If data errors occur, for example due to noise or signal interference, they are not automatically detected or corrected.
4. Limited flexibility: UART offers limited options for extending or configuring protocols. It is less flexible than more complex protocols such as Ethernet or USB, which offer more extensive functionality and configuration options.
5. Limited bandwidth: Due to the serial nature of communication, UART has limited bandwidth compared to parallel communication technologies. It is not suitable for high data rate applications where a large amount of data must be transmitted simultaneously.

Despite some limitations, UART is still a popular communication technology due to its simplicity, flexibility and widespread use. It is well suited for a variety of applications requiring reliable serial data transmission at moderate speeds.

It is important to note that UART usually serves as a basic serial interface and may not be sufficient for more specific requirements. In such cases, more advanced protocols such as Ethernet, USB, or SPI can be used to provide advanced features such as higher data rates, multidrop communication, error detection and correction, and increased bandwidth.

Ultimately, the choice of communication technology depends on the specific requirements of the application. However, UART remains a valuable option for many applications due to its simplicity, compatibility and availability in a wide variety of microcontrollers and integrated circuits.

6. OneWire

- https://github.com/villeneuve/libsigrokdecode-ds18b20
- Download file and copy the ds18b20 folder to
- (Appimage) /usr/share/libsigrokdecode/decoders/
- Alternatively ~/.local/share/libsigrokdecode/decoders/
- Under Linux
- move with admin rights (sudo cp -r)
- Adjust permissions chmod 755 ds1812b
- PulseView restart and the decoder should be available

6.1.One Wire protokol

The OneWire protocol is a serial communication standard developed by Dallas Semiconductor (now Maxim Integrated). It allows communication with devices connected by only one data line. Here are some basics about the OneWire protocol:

1. Data line: The OneWire protocol uses a single data line that is used both for the transmission of data and for the power supply of the connected devices. This data line is often referred to as the "DQ" (Data Line).
2. Time multiplexing: Since only one data line is used, communication in the OneWire protocol is done by time multiplexing. This means that data bits and control signals are transmitted over the same line at different times.
3. Pull-up resistor: For the OneWire protocol to work, the data line must normally be connected to a voltage via a pull-up resistor. The pull-up resistor ensures that the data line is kept at a logical high level (e.g. 5V) in the idle state.
4. Reset: To start communication, the master device (e.g. a microcontroller) sends a reset pulse on the data line. This pulse consists of a low voltage for a certain period of time, followed by a return to the voltage of the pull-up resistor.
5. Presence pulse: After the reset pulse, the slave devices (e.g. sensors) must respond to the data bus to indicate their presence. For this purpose, the slave device pulls the data line to a logical low level for a short time and then lets it return to the high level.
6. Data transmission: The actual data is transmitted bit by bit. A logical "0" is represented by a low voltage and a logical "1" by maintaining the voltage of the pull-up resistor. The data is transmitted in a certain time interval and can be read by a master device or sent to a slave device.

7. CRC checksum: To ensure the integrity of the transmitted data, the OneWire protocol uses a cyclic redundancy check (CRC). The CRC checksum is added to the end of the transmitted data and allows the receiver to detect errors.

The OneWire protocol is widely used in communication with various devices, including temperature sensors (e.g. DS18B20), real-time clocks and memory devices. It provides a simple and cost-effective way to connect multiple devices with just one data line.

6.2.Nine or 12-bit resolution

The DS18B20 is a digital temperature sensor that is frequently used in various applications. The terms "9-bit resolution" and "12-bit resolution" refer to the accuracy with which the DS18B20 can measure temperature.

Resolution refers to the number of bits the sensor uses to represent the measured temperature. A bit is the smallest unit of information and can have either the value 0 or 1. The higher the number of bits, the more different combinations of zeros and ones can be used to represent the measured values.

With a 9-bit resolution, the DS18B20 uses 9 bits to represent the temperature. This means that it can represent 2^9 (2 to the 9th power) or 512 different values. This results in a coarser accuracy because the temperature values are represented in larger increments. For example, the sensor could have an accuracy of 0.5 degrees Celsius, which means that it measures temperatures in steps of 0.5 degrees Celsius.

With a 12-bit resolution, the DS18B20 uses 12 bits to represent the temperature. This means that it can represent 2^12 (2 to the power of 12) or 4096 different values. This increases accuracy because the temperature values can be represented in smaller increments. For example, with a 12-bit resolution, the sensor could have an accuracy of 0.0625 degrees Celsius.

Overall, the higher the bit resolution, the more accurately the DS18B20 can measure temperature, but more memory is needed to store the additional bits. It is important to note that the actual accuracy of the sensor can also be affected by other factors such as ambient temperature, calibration, and electrical noise.

7. SPI

SPI stands for Serial Peripheral Interface and is a serial communication standard commonly used in microcontroller programming, including the Arduino platform. SPI communication is usually between a master device and one or more slave devices.

The operation of SPI is based on a synchronous serial protocol. As a rule, four lines are used:

- SCLK (Serial Clock): This line is controlled by the master and generates the clock signal that enables synchronization of communication.
- MOSI (Master Output, Slave Input): This line is used by the master to send data to the slave.
- MISO (Master Input, Slave Output): This line is used by the slave to send data to the master.
- SS (Slave Select): This line is used by the master to select the appropriate slave to communicate with. If there are several slaves, a separate SS line is normally used for each slave.

<u>The sequence of SPI communication is as follows:</u>

1. The master activates the SS line of the desired slave.
2. The master sends a clock pulse on the SCLK line.
3. With each clock pulse the master transmits a bit via the MOSI line and the slave receives the bit via the MISO line.
4. At the same time the slave transmits a bit via the MISO line and the master receives the bit via the MOSI line.
5. This process is repeated for each bit to be transferred.

Data is normally transmitted in full-duplex mode, which means that both the master and the slave can send and receive simultaneously.

The exact configuration and handling of SPI can vary depending on the microcontroller and library. However, the Arduino platform usually provides functions and libraries to facilitate and abstract SPI communication.

7.1.Data transmission in Detail

With SPI, the MOSI line (Master Out Slave In) is used from the master to the slave to transfer data. The data is transferred synchronously to the rising edge of the clock signal. Here is a short explanation how the MOSI line is evaluated at a rising edge of the clock:

- Preparation: Before data transmission starts, the Slave Select (SS) pin of the corresponding slave device is set to LOW to select the slave for communication.
- Data transmission: At a rising edge of the clock signal the master sets the data bit values on the MOSI line. This means that the master puts the first bit of the data on the MOSI line while the clock changes from LOW to HIGH.
- Data acquisition: The slave detects the rising edge of the clock and perceives the data bit on the MOSI line. The slave reads the state of the MOSI line and interprets the data bit accordingly. When the MOSI line is HIGH (logic 1), the slave senses a logic 1 for the corresponding bit. If the MOSI line is LOW (logical 0), the slave detects a logical 0.
- Continuation of the data transmission: After the data bit is acquired, the clock continues to be clocked and the master transmits the next bit on the MOSI line. The slave repeats the data acquisition process for each data bit while the clock is clocked.
- Completion of the transmission: After all data bits have been transmitted, the Slave Select (SS) pin is set HIGH again to disable the slave for further communications.

This process is repeated for each data transmission in the SPI protocol. The rising edge of the clock and the acquisition of the data bit on the MOSI line allows the slave to correctly receive and process the sent data from the master.

7.2.Full Duplex

PI (Serial Peripheral Interface) is a full-duplex communication protocol. This means that data can be transmitted in both directions simultaneously. Both the master and the slave can send and receive data while communicating with each other.

In full duplex mode SPI works with two data lines: MOSI (Master Out Slave In) and MISO (Master In Slave Out). The master sends data to the slave via the MOSI line, while the slave sends data to the master via the MISO line. The clock (SCK) is provided by the master and serves as a synchronization mechanism for data transmission.

This simultaneous transmission in both directions enables SPI to provide efficient and fast communication between the master and the slave device. It is important that both the master and slave have separate MOSI and MISO lines to support full-duplex communication.

8. I2C

8.1.BH1750

Values of binary in DEC and then / 1.2 see data sheet = LUX

8.2.DHT20

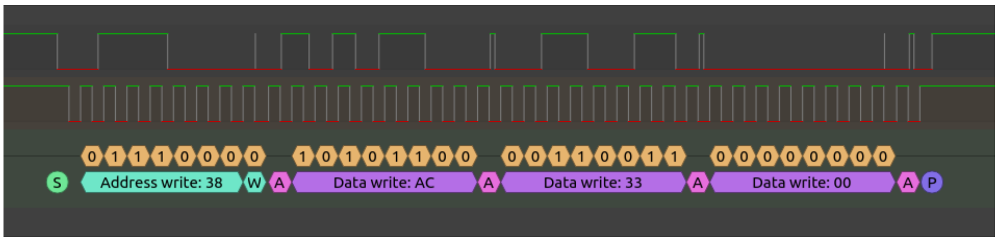


Figure 1: Trigger measurement DHT20

According to the information in the data sheet, the transmission of four bytes takes place: "Sending a write command to the address 0x38", "Request for measurement 0xAC" with the parameters 0x33 and 0x00.

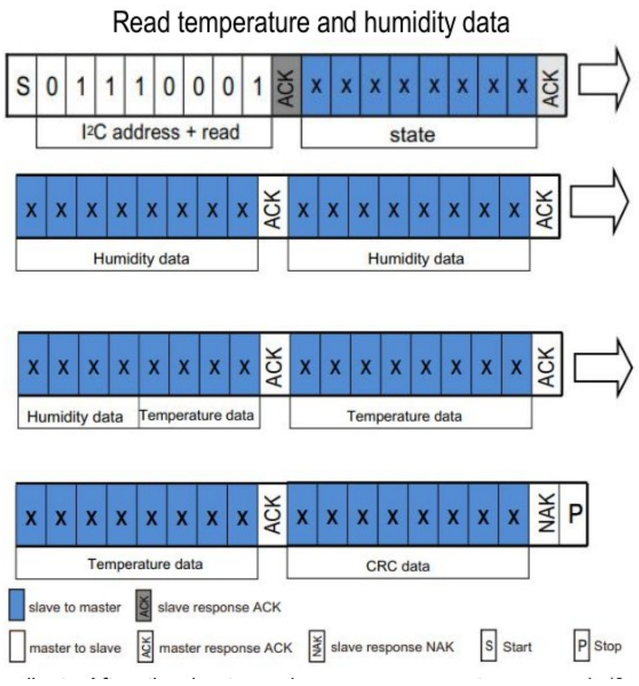


Figure 2: Excerpt data sheet Read Temperature and humidity

Afterwards the instruction to read out the measured data is given. For clarification: The I2C address consists of 7 bits, here represented by 0x38 = 0b0111000. If the eighth bit is a 0, the sensor will remain in "listening mode" as shown in the figure above. If the eighth bit is a 1, the sensor will send data as shown in the following figure:

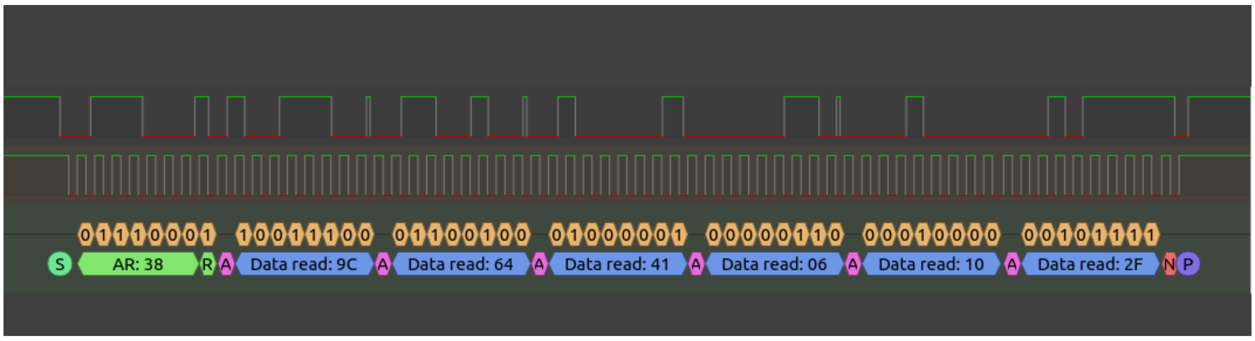


Figure 3: second part DHT20 evaluation

The evaluation looks as follows:

8.2.1. Humidity

9c = State. Byte 64, 41 und die Hälfte von 06 = SRH.

SRH = 1100100010000010000 = 410640

$$\frac{SRH}{2^{20}} * 100\,\%$$

$$\frac{410640}{2^{20}} * 100\,\% = 39{,}161682129$$

8.2.2. Temperature:

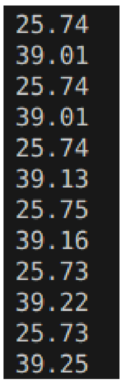
ST = 01100001000000101111= 397359

$$\frac{ST}{2^{20}} * 200 - 50$$

$$\frac{397359}{2^{20}} * 200 - 50 = 25{,}790214539$$

The serial monitor outputs the following:

First the degrees in °C and then the humidity in %.



```
25.74
39.01
25.74
39.01
25.74
39.13
25.75
39.16
25.73
39.22
25.73
39.25
```

9. Conclusio

Here is a brief summary of the analyzed hardware protocols in the Arduino environment and their application areas:

2C (Inter-Integrated Circuit):

Usage: short range communication with multiple devices.

Example application: Sensors and actuators that transmit data to an Arduino, e.g. temperature sensors, humidity sensors or LCD displays.

Advantages:

- Simple cabling with only two lines (data and clock).
- Multiple devices can be connected to the same data and clock lines.
- Low power consumption due to pull-up resistors.
- Relatively slow data traffic, suitable for applications with low data volume.

Disadvantages:

- Limited cable length due to resistance in the data lines.
- Lower data rates compared to other protocols like SPI.

SPI (Serial Peripheral Interface):

Usage: High speed data transfer between microcontrollers and peripherals.

Example application: communication with external memory chips, TFT displays or radio modules.

Advantages:

- High data rates for fast and efficient data transmission.

- Full duplex communication enables simultaneous sending and receiving of data.
- Separate data lines for each device allow flexible configuration.

Disadvantages:

- Requires more pins for cabling compared to I2C and UART.
- More complex cabling than I2C and UART.

UART (Universal Asynchronous Receiver/Transmitter):

Usage: Simple serial communication with a single device.

Example application: communication between Arduino and a computer, Bluetooth modules or GSM modules.

Advantages:

- Easy implementation and cabling with only two lines (TX and RX).
- Supports serial communication with a small number of devices.
- Wide distribution and compatibility with many devices and protocols.

Disadvantages:

- Punkt-zu-Punkt-Kommunikation, keine natürliche Unterstützung für mehrere Geräte.
- Relativ langsamere Datenraten im Vergleich zu SPI.

OneWire:

Usage: Communication with multiple sensors via a single data pin.

Example application: Digital temperature sensors (e.g. DS18B20) or other sensors where simple wiring is desired.

Advantages

- Simple cabling with only one data pin.
- Supports communication with multiple sensors via the same data pin.
- Low resource consumption, as no additional hardware modules are required

Disadvantages:

- Limited availability of OneWire-enabled sensors compared to other protocols.
- Relatively slower data rates compared to I2C and SPI.

10. USB, Ethernet and JTAG

10.1. USB (Universal Serial Bus):

USB is a widely used communication protocol that is used to connect electronic devices. It enables the exchange of data and power between a host (such as a computer) and a peripheral device (such as an Arduino board). USB offers different transfer speeds and supports different types of data transfers such as mass storage, keyboards, mice, and serial communication. Arduino boards can be connected to a computer via USB to upload programs and use serial communication to transfer data.

10.2. Ethernet:

Ethernet is a protocol for networking computers and other devices in local area networks (LANs). It enables the exchange of data packets over a wired network. Arduino boards can use Ethernet to connect to the Internet or other devices on a network. This opens up possibilities for IoT (Internet of Things) and communication with web services or remote control applications.

The POE protocol allows both data and power to be transmitted over the same Ethernet cable. It is based on the IEEE 802.3af standard or the extended IEEE 802.3at standard. This allows you to connect an Arduino board with POE function directly to a POE-capable Ethernet switch or injector that provides power over the Ethernet cable.

The advantage of the POE function is that you do not need a separate power supply for the Arduino board. This can be especially handy if the board is installed in a place where there is no power outlet nearby or if you want to avoid cable clutter. It also makes it easier to install Arduino-based projects in network environments, especially in building automation, surveillance systems, or other IoT applications.

10.3. JTAG (Joint Test Action Group):

TAG is a protocol that is mainly used for debugging and programming integrated circuits (ICs). It allows access to the internal logic of an IC for diagnostic and test purposes. For Arduino, JTAG is typically used to connect the Arduino board directly to a programming device to program or debug the microcontroller on the board. JTAG provides an efficient way to perform firmware updates on Arduino boards or to find errors.