Networking and Interfacing IoT Platforms
Lab 1
Low power platforms and mcu/sensor communication

There are two parts to this assignment. Part one will focusses on basic electronics followed by part two which consists of a deep dive into commonly used communication protocols for microcontrollers and sensors.

Important: Do <u>NOT</u> connect any circuits you create to a power source without permission from the teaching team! When you have finished creating your circuits, always have a teaching assistant come and inspect your work.

Note: Labs can be answered in Dutch or English, the important part is the contents. Paste lots of pictures/diagrams/… instead of writings walls of text. Do include all your failures in the description.

Part 1: Electronics

In the following assignments you will become acquainted with the basics of electronics. The goal is to have a basic understanding of circuits and how to build them. You are expected to write out your findings and solutions as detailed as possible (i.e., use pictures, screenshots, schematics, diagram, … but only if it enhances your explanation).

Assignment 1.1: The basics

Provided with the following parameters, calculate the requested values:

A. *A 100Ω resistor rated for 0.5W hooked in a series circuit connected to a 12V power supply. How much power does the resistor consume and why does this circuit (not) work? What happens to the power consumed by the resistor ?*

   From the formula of Ohm's law (V = I*R), we can conduct the value of I. Which gives us a power consument of 0.12A. This would mean the resistor consumes 1.44 W (P = I*I*R), which is higher than its rated value, so the resistor will not work in the given situation.

B. *Your circuit requires a 350Ω resistor to protect its components, based on calculations. However, resistors are only available in certain values (e.g. E12 series). Which resistor in the E12 series would you use here and why? What happens if we use a lower/higher resistor value than the one we require?*

   An E12 series resistor has a 10% tolerance on its value, so it would be advisable to use a 39 value resistor (390Ω). The actual resistance will be in the range 350Ω-429Ω. A higher value would limit the current further, whereas a lower value would potentially harm the components.

C. *Two leds with a forward voltage of 2V and a rated current of 25mA are connected to a 5V power supply in series. What value resistor do we at least require?*
   As both the LEDs consume 2V, we have to eliminate 1V in the circuit. This can be done by adding a resistor of 40 Ohm (R = V/I = 1V / 0.025A)

D. *A USB power bank is rated at 10.000mAh. This is based on an internal battery voltage of 3.7V (normalized). The power bank outputs 5V instead of 3.7V. How much mAh can we expect to have available ?*

   We know that P = IV, so the wattage of the power bank would be 37Wh. To determine the current in the circuit, we divide 37Wh with 5V. This gives us a value of 7400mAh.

E. *Given the previous set of parameters, assume that we attach a device consuming 500mA. How long can the device be expected to function (substantiate !)*

   As the power bank gives us 7.400mA per hour, we can conclude that the device can be powered for almost 15 hours (14h 48').

Assignment 1.2: Getting started with PyCom

Read the PyCom quickstart provided by the teaching staff. If you encounter any difficulties, write them down and how you fixed them. Once you are connected to your PyCom you should see a generic welcome message. Download the contents from the PyCom device and change the main.py script to say your name name instead of the generic welcome message.

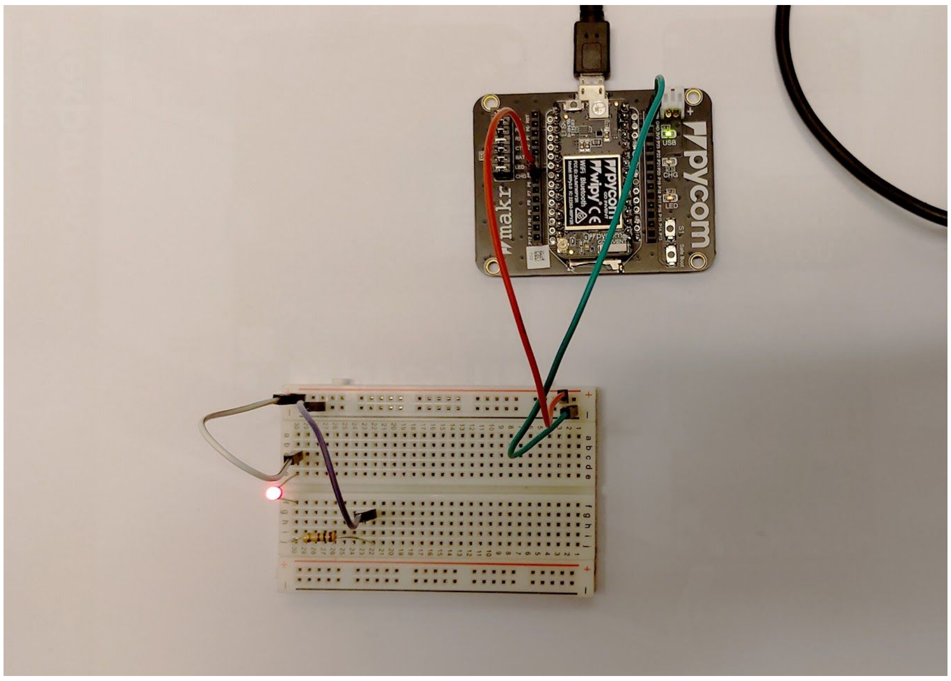
By using the PyMakr plugin for Visual Studio Code, we were able to download the existing code from the microcontroller. We then changed the value of the name variable, which gives us the following output:

```
-------------------------------------------------------------------------------
Hello there William & Bram!
It seems you were able to connect your PyCom and see its UART output.
Download the sourcecode, change the name variable to your own and reupload it.
-------------------------------------------------------------------------------
```

Assignment 1.3: Let there be light

Using the knowledge gained from previous questions, build a simple circuit consisting out of a red or green LED that you can turn on or off using python code on a PyCom. Take pictures and describe how it works.

With the same formula we used above, we determined that we need a resistor of 52 Ohm. The smallest resistor available is one of 100 Ohm, which still works (although the LED will not light up at its maximum). The code lets the LED toggle once every 2 seconds (period), with a duty cycle of 1 second.



Part 2: MCU/sensor communication protocols

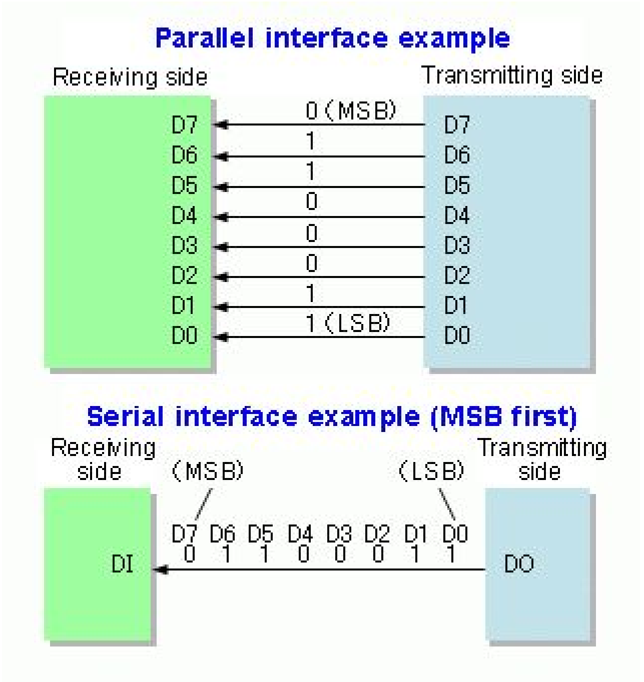
In the following assignments you will become acquainted with the most commonly used communication protocols for digital electronics. Just like in the previous assignment, make use of diagrams, pictures, screenshots, … if and only if it enhances your writings.

Assignment 2.1: Communication protocols

Look into the following list of communication protocols (besides those already explained in the slide sets on Classroom) and summarize their inner workings. For each serial communication protocol, describe in enough detail how the signalling of data works and what the protocol looks like.

- *Parallel communication*
- *Serial communication*
  - *1-Wire*
  - *CAN*

Parallel vs Serial interface



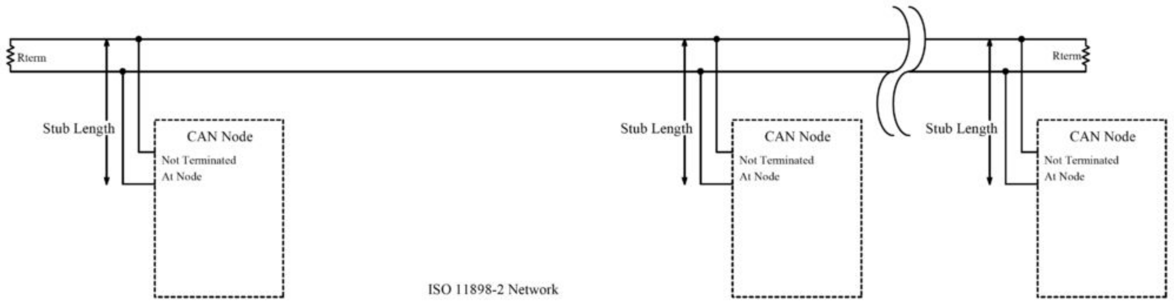
A parallel link transmits several streams of data simultaneously along multiple channels, whereas a serial link transmits only a single stream of data.
At first, it seems that serial communication would be slower, since it can transmit less data per clock cycle. However, it is often the case that serial links can be clocked considerably faster than parallel links in order to achieve a higher data rate.

1-Wire communication

1-Wire is a serial protocol using a single data line plus ground reference for communication. It initiates and controls the communication with one or more 1-Wire slave devices on the 1-Wire bus. Each slave device has a unique 64-bit identifier, which serves as device address on the bus. The 8-bit family code (the

least significant bits of the 64-bit identifier), identifies the device type and functionality.
Most 1-wire devices have no pin for power supply, as they take their energy from the 1-Wire bus.

CAN-Communication
The Controller Area Network (CAN) is a serial communication bus designed for robust and flexible performance in harsh environments, and particularly for industrial and automotive applications.
CAN is a multi-master serial bus standard for connecting ECUs (= nodes). Two or more nodes are required on the CAN network to communicate. All the nodes are connected to each other through a two wire bus.

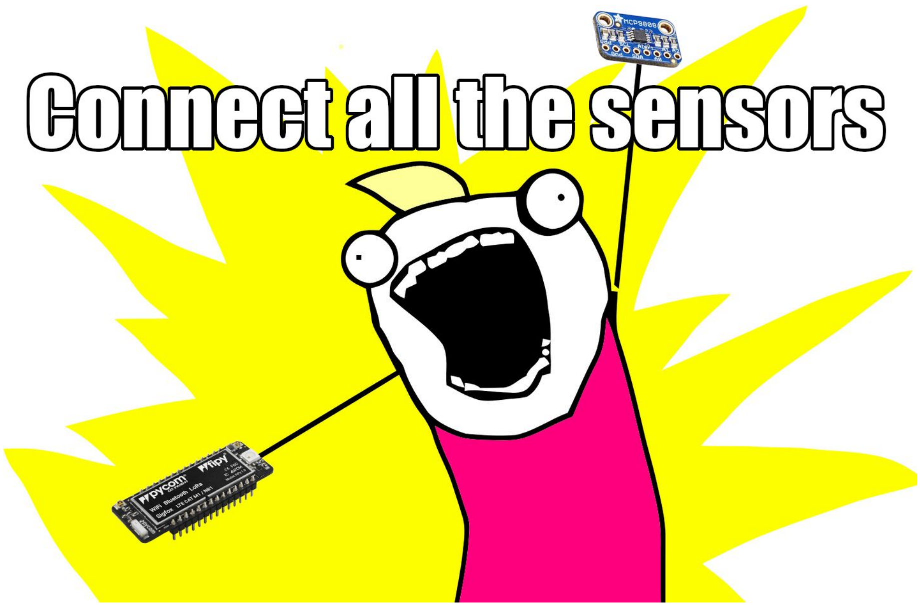


Assignment 2.2: Connect all the sensors

Following sub-assignments consist of practical setups you will have to implement using the mentioned communication protocol. We encourage you to be creative in your creations (unless mentioned otherwise). More hardware is available upon request.

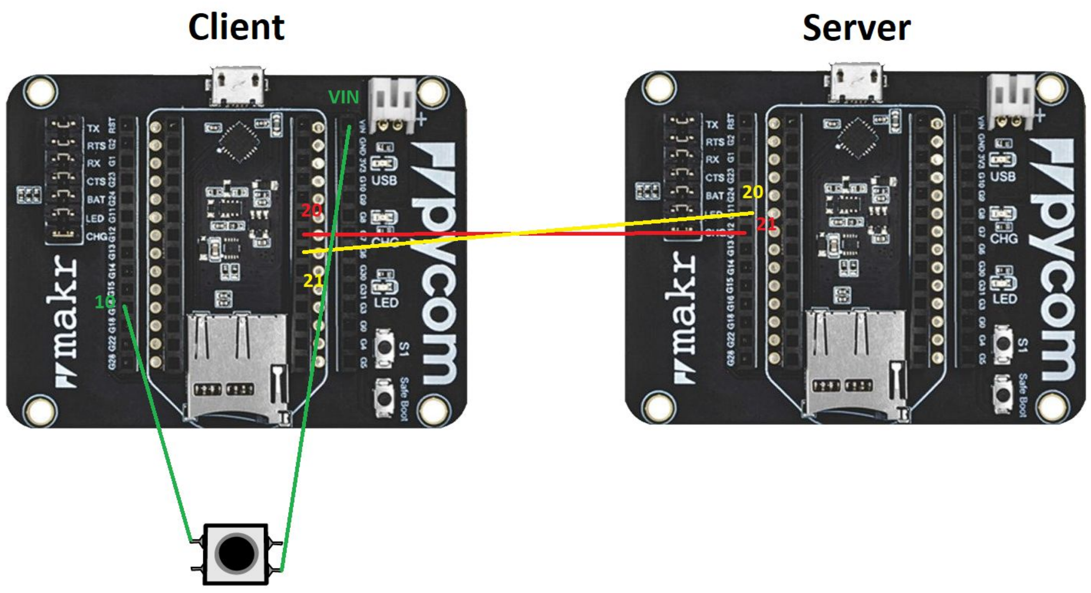
After you finish, take pictures and/or screenshots of your working creation and document it below the corresponding sub-assignment.

As a final check, hook up the Saleae Logic 8 to your setup and analyze the exchanged data. Save your capture, take a screenshot of at least 1 interesting data exchange and annotate it with your findings about serial communication from assignment 2.1 (i.e., annotate the protocol on a captured example). Make sure that you have the teaching team check the connections before powering on, each logic analyzer costs upwards of 400 euros !



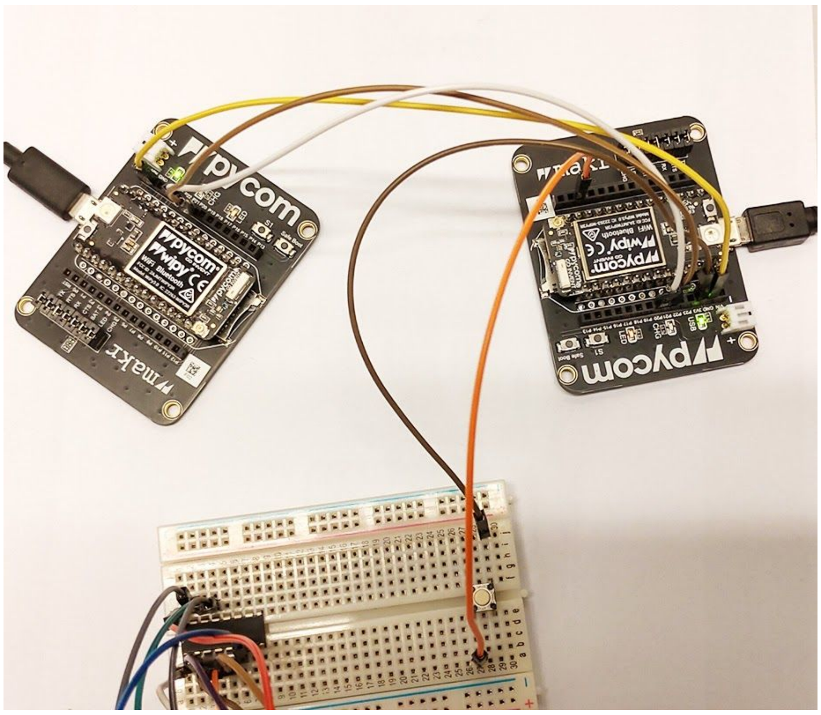
UART: *Create a slot machine by connecting two PyComs using UART. The goal is to create a 3 wheel slot machine with at least a horizontal payout line. One PyCom will act as the slot machine (server) whilst the other PyCom acts as the input (client). The client should at least use 1 button. Be creative with your implementation (i.e., do not necessarily limit yourself to console interaction).*

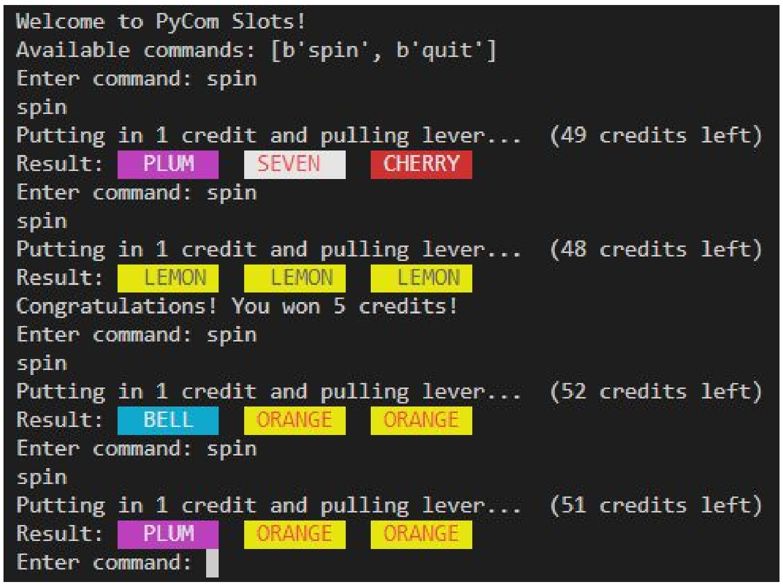
Schematic:



At first we wired the button manually with a pull-down resistor. As this is not required for the PyCom, we preferred the "vanilla" setup with the built in PULL_DOWN option.
One PyCom acts as a server and receives a command (`spin` or `quit`), while responding with the current game state. The other PyCom will send the `spin` command over uart and displays the response from the server. At first a lot of thrash bytes were sent through the connection and the actual data did not seem to come through. After a quick verification from one of the assistants, it became apparent we forgot to make the two system share a common ground.

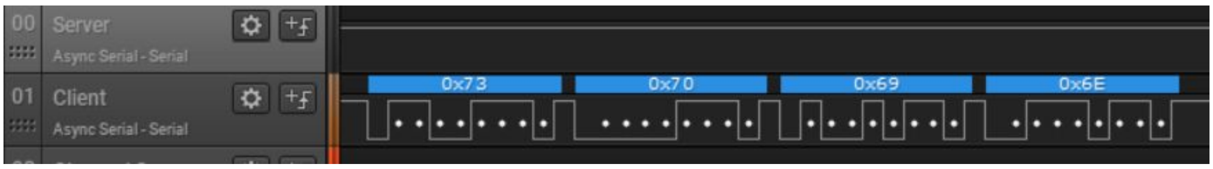
Example output can be seen below.



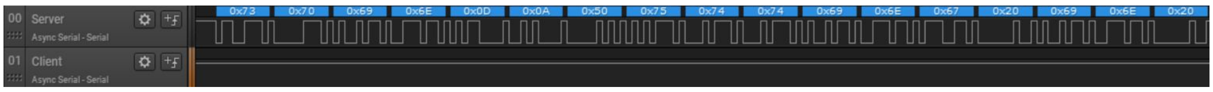


```
Welcome to PyCom Slots!
Available commands: [b'spin', b'quit']
Enter command: spin
spin
Putting in 1 credit and pulling lever...  (49 credits left)
Result:  PLUM    SEVEN   CHERRY
Enter command: spin
spin
Putting in 1 credit and pulling lever...  (48 credits left)
Result:  LEMON   LEMON   LEMON
Congratulations! You won 5 credits!
Enter command: spin
spin
Putting in 1 credit and pulling lever...  (52 credits left)
Result:  BELL    ORANGE  ORANGE
Enter command: spin
spin
Putting in 1 credit and pulling lever...  (51 credits left)
Result:  PLUM    ORANGE  ORANGE
Enter command:
```

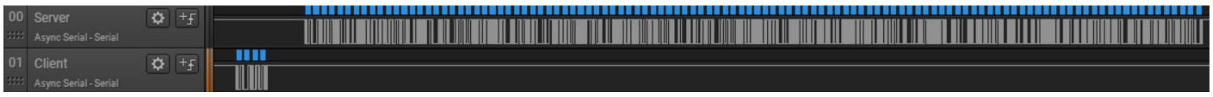
Client sends b"spin" (== ('s', 'p', 'i', 'n') == b"\x73\x70\x69\x6E"):



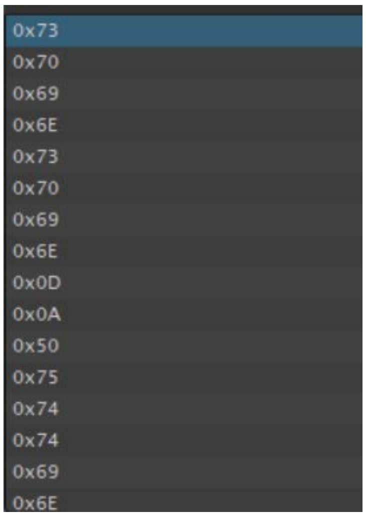
Server responds with state (long string with game telling the lever got pulled, the amount of credits left and the result with color coding):



Overview: client sends "spin" (bottom graph) and server responds (top graph)



Raw data: The first 4 bytes are the client sending "spin" (== b"\x73\x70\x69\x6E"), the server echoes this command and start returning data from the game state.

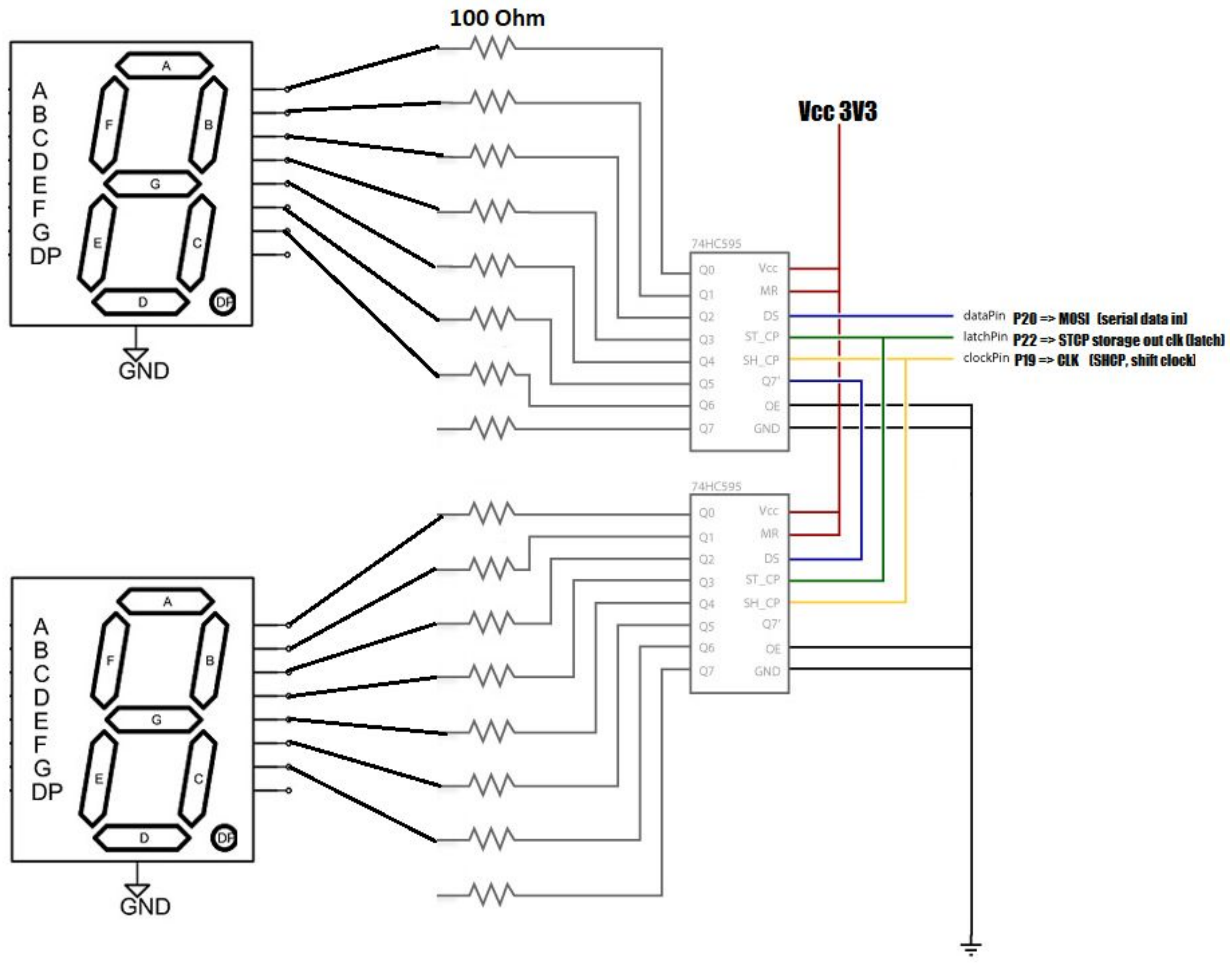


```
0x73
0x70
0x69
0x6E
0x73
0x70
0x69
0x6E
0x0D
0x0A
0x50
0x75
0x74
0x74
0x69
0x6E
```

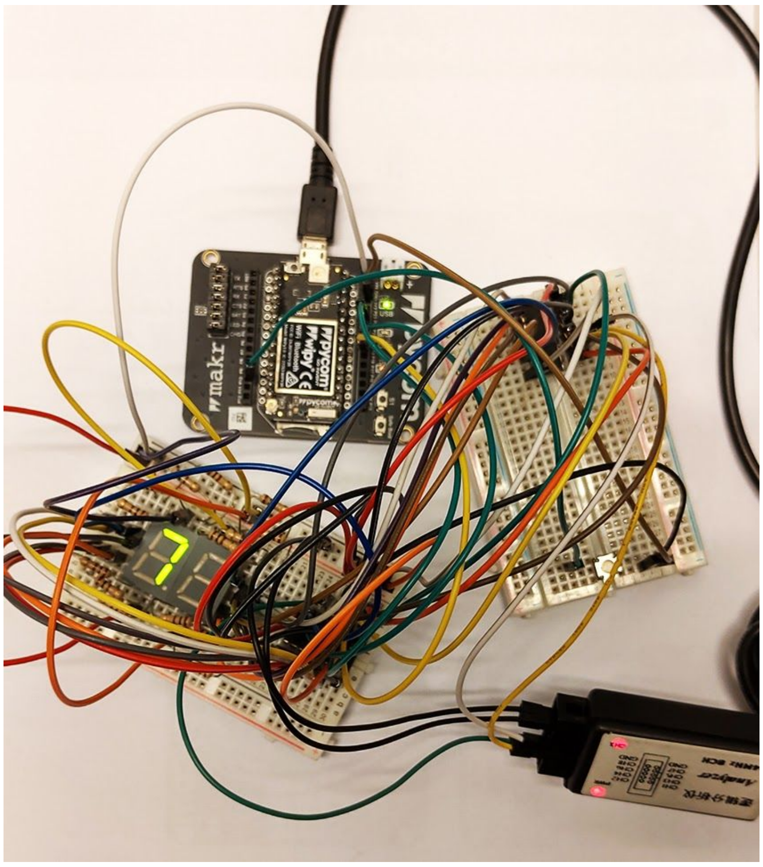
SPI: *Create a digital Dungeons and Dragons dice using the 74HCN595N SPI shift register in combination with a SBA56-21EGWA 7-segment LED display. Be creative in your setup.*

We first wanted to continuously shift the next value (increased by 1) to the shift register, causing the 7-segment displays to change constantly, but the clock speed was too high to see any changes (we didn't add a delay). So instead we used machine.rng() to get a random number and clamp it between the required dice values. When the user presses the button, the random function will be called and numbers are displayed with a decreasing delay, so it looks like the "die" comes to a stop. The values range from "01" to "20", or 1 to 20 as a D&D die is a 1d20. (see video)

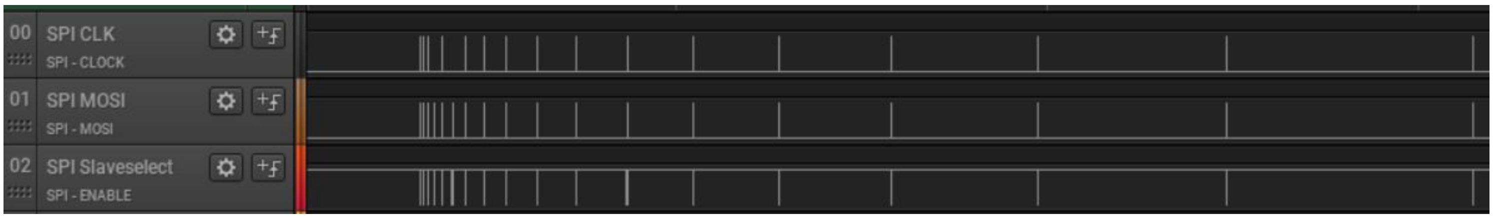
Schematic:


100 Ohm
Vcc 3V3
A
B
C
D
E
F
G
DP
GND
74HC595
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Vcc
MR
DS
ST_CP
SH_CP
Q7'
OE
GND
dataPin P20 => MOSI (serial data in)
latchPin P22 => STCP storage out clk (latch)
clockPin P19 => CLK (SHCP, shift clock)

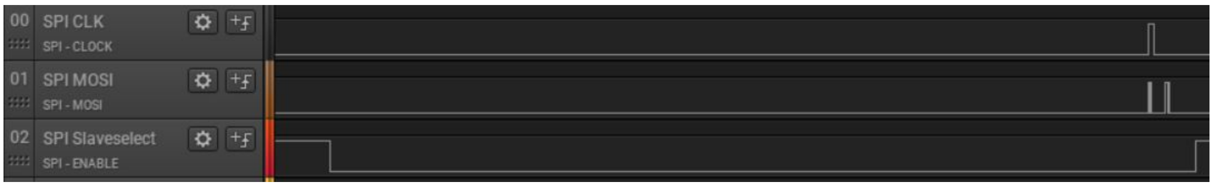
In the picture below, we used a 1d100 instead (values between 00 and 99) to check if the second display also works properly. (below is the value "71" written out, but the "1" was not visible due to the extensive cabling required, refer to the video to see the full result.)



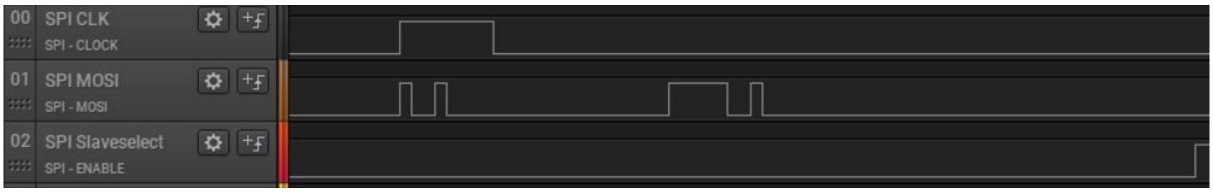
SPI overview, the increased delays after the button press are clearly visible: it takes longer and longer to send the next value, until it stops.



Here it is clear that the slave select pin gets pulled down, after which communication starts.



The actual data written, to shift in. The Logic software did not decode this part, as it is essentially a raw setting of pins (even though the SPI protocol was selected).

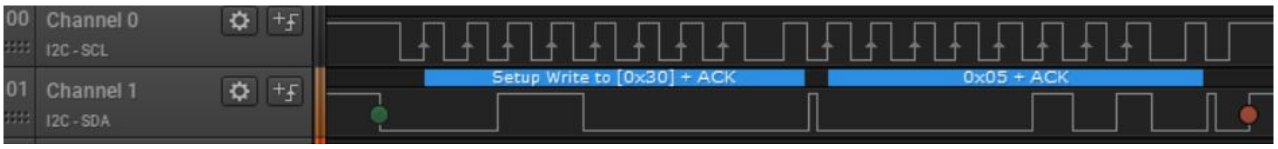


I2C part 1: *Hook up a MCP9808 temperature sensor to a PyCom using I2C. Scan the I2C bus for devices and confirm that the device you hooked up is present. Once connected, use the provided library to read out the ambient temperature. No creative element is expected in this assignment.*

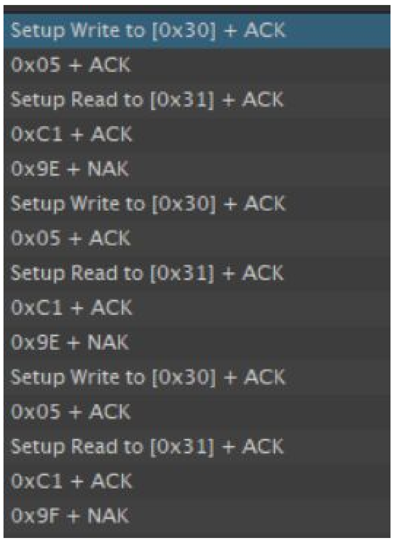
(see next part for a picture of the wiring)
When the bus is scanned (`i2c.scan()`), the array `[24]` is returned where 24 (== 0x18) is the address of the sensor. However this is the 7-bit address. When shifting the value to the left by 1, we get 48 == 0x30 which is the value reported in the logic analyser. This shift is required as the LSB indicates if the I2C call is for reading ('1') or writing ('0') from the sensor.
It turns out that PyCom requires the 7-bit address for I2C calls and will shift the value by its own when reading or writing.

Output: Temperature: 25.69°C

Logic Analyser output:



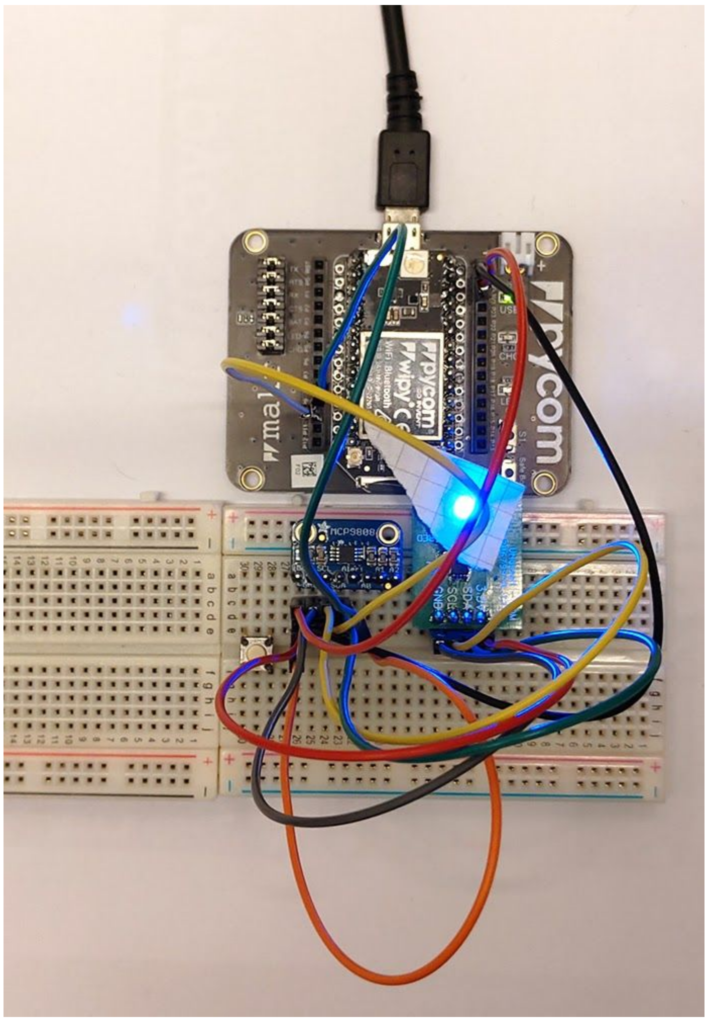
Decoded protocol:
First the address `((0x18 << 1) | 1) = 0x31` is sent on the bus and ACKed by the sensor, indicating writing. Next, 0x05 is sent, we are requesting the register 0x05 to be read from to get the latest temperature data. Subsequently we request to read from address `((0x18 << 1) | 0) = 0x30` and get b"\xC1\x9E" as response. These bytes are then converted to the value 25.875 °C.

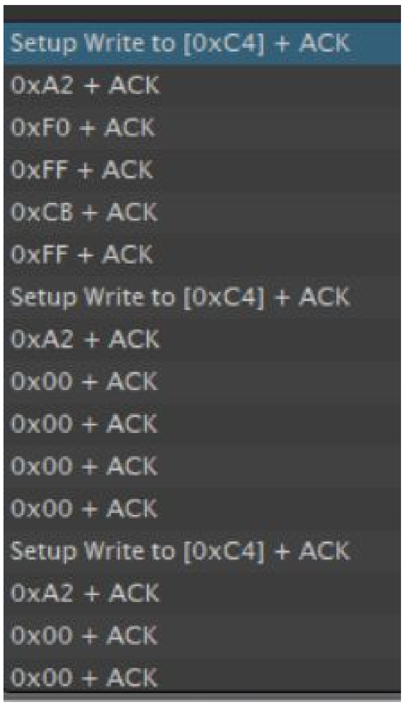


```
Setup Write to [0x30] + ACK
0x05 + ACK
Setup Read to [0x31] + ACK
0xC1 + ACK
0x9E + NAK
Setup Write to [0x30] + ACK
0x05 + ACK
Setup Read to [0x31] + ACK
0xC1 + ACK
0x9E + NAK
Setup Write to [0x30] + ACK
0x05 + ACK
Setup Read to [0x31] + ACK
0xC1 + ACK
0x9F + NAK
```

I2C part 2: *Develop a library to control an in-house developed I2C PCA9632 LED driver board with built in RGB LEDs. The library should be able to control all three LED colors their brightness individually as well as the group brightness. It should also be able to set blinking mode to be on or off (Hint: Read the manual through first before you begin with this assignment).*

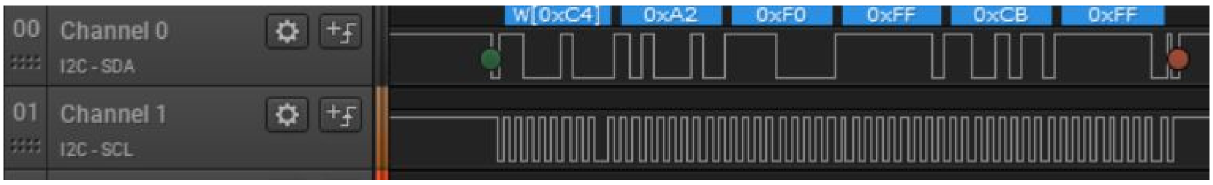
LED is controllable with written driver code. The I2C clock SCL is connected to the default pin P10 on the PyCom, while the I2C data SDA is connected to P9. Of course both the temperature sensor and the led driver board share the same SCL and SDA lines, as they have different addresses on the same bus, as well as the same Vcc and GND.





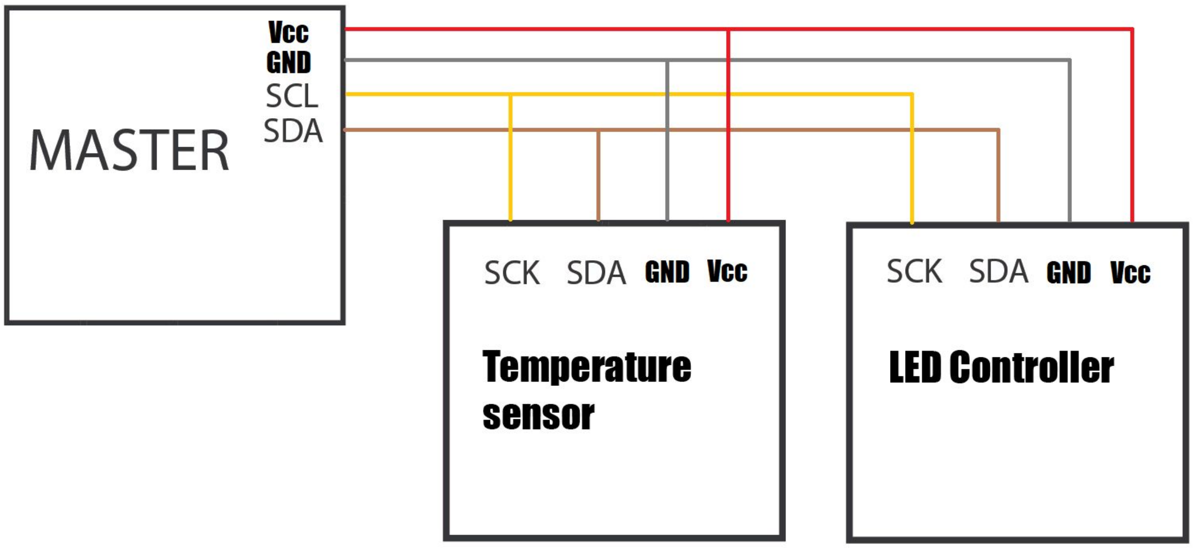
```
Setup Write to [0xC4] + ACK
0xA2 + ACK
0xF0 + ACK
0xFF + ACK
0xCB + ACK
0xFF + ACK
Setup Write to [0xC4] + ACK
0xA2 + ACK
0x00 + ACK
0x00 + ACK
0x00 + ACK
0x00 + ACK
Setup Write to [0xC4] + ACK
0xA2 + ACK
0x00 + ACK
0x00 + ACK
```

Send address (`0xC4`, auto increment options `0xA0` (option `0b101` or increment individual regs only) and start register `0x02`, and 4 bytes for BGRA). By setting the auto increment option, we only need to send the start register. All subsequent byte writes cause the register pointer to be automatically increased, as the option indicates.

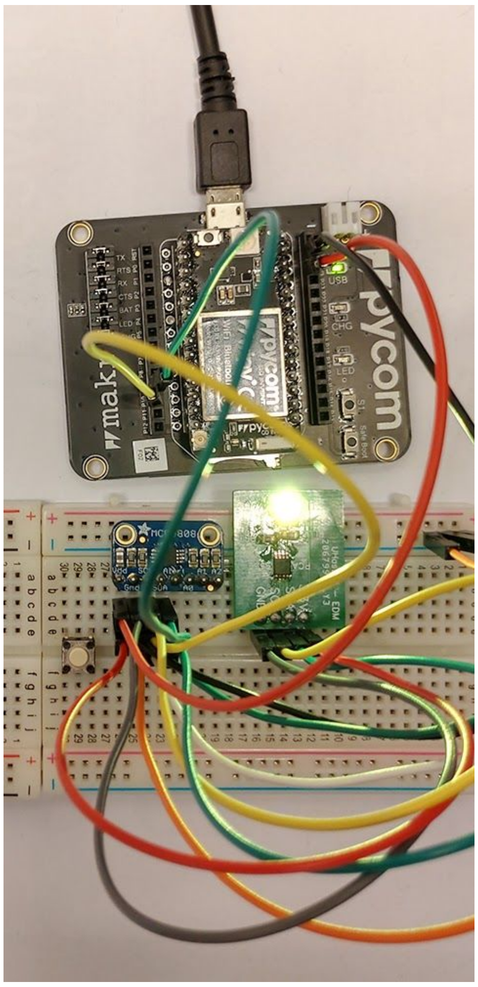


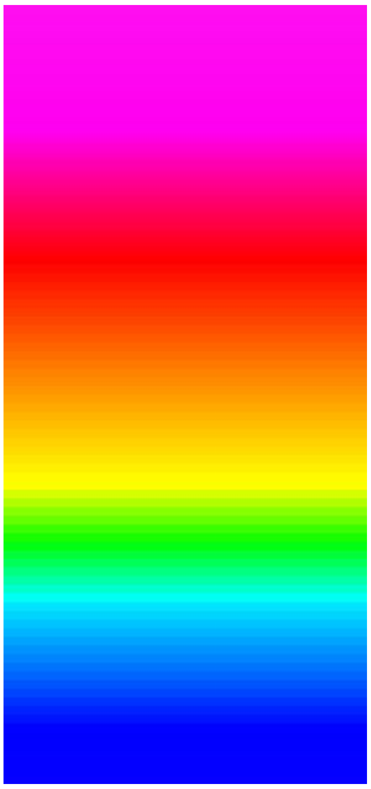
I2C part 3: *Combine part one and two into one circuit. The end goal is to create a setup where the RGB leds reflect the ambient temperature as measured by the MCP9808. E.g., hot ambient temperature is reflected by warm colors whilst colder ambient temperature is reflected by a colder color.*

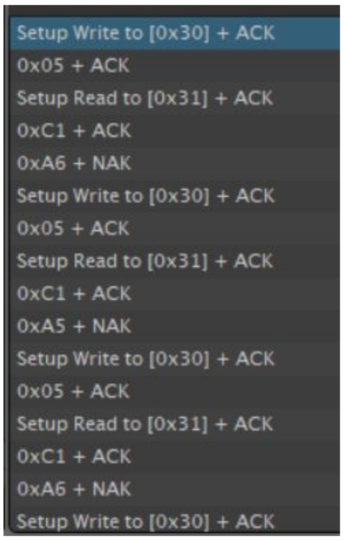
Schematic (analog for all I2C parts):



LED is controllable with written driver code, and a LUT determines the colour by reading the temperature sensor value. Here approx 26.5°C, which gives #d7ff00. The led is lighted with the colour according to the palette below (purple starts from 100 °C up to -10 for blue at the bottom).

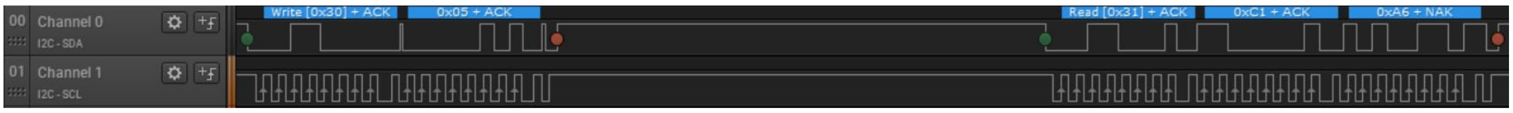






Setup Write to [0x30] + ACK
0x05 + ACK
Setup Read to [0x31] + ACK
0xC1 + ACK
0xA6 + NAK
Setup Write to [0x30] + ACK
0x05 + ACK
Setup Read to [0x31] + ACK
0xC1 + ACK
0xA5 + NAK
Setup Write to [0x30] + ACK
0x05 + ACK
Setup Read to [0x31] + ACK
0xC1 + ACK
0xA6 + NAK
Setup Write to [0x30] + ACK

Send temp sensor address 0x30 and request data from reg 0x05, 3 bytes data (0x31, 0xC1, 0xA6). (no data written to led in this case).



In the video we changed the step just above 26.5°C to a purple colour, so the change when the next temperature interval is reached, is clearly visible.