Processing signals received from sensors

Student: Laurentiu-Calin Grad

2025

Technical University of Cluj-Napoca

Structure of Computer Systems Project

Contents

[**1.** **Introduction** 1](#_Toc183461186)

[1.1. Context 1](#_Toc183461187)

[1.2. Objectives 1](#_Toc183461188)

[**2.** **Bibliographic Research** 2](#_Toc183461189)

[2.1. PMOD HYGRO 2](#_Toc183461190)

[2.1.1. The I²C interface (protocol) 2](#_Toc183461191)

[2.1.2. Interfacing with the Pmod 3](#_Toc183461192)

[2.2. PMOD ESP32 5](#_Toc183461193)

[2.2.1. AT command set 6](#_Toc183461194)

[2.2.2. Interfacing with the Pmod 7](#_Toc183461195)

[2.2.3. UART 8](#_Toc183461196)

[**3.** **Analysis** 9](#_Toc183461197)

[3.1. Project Proposal 9](#_Toc183461198)

[3.2. Project Analysis 9](#_Toc183461199)

[3.2.1. Component interfacing 9](#_Toc183461200)

[3.2.2. Control logic of the system 10](#_Toc183461201)

[3.3. Conclusions: 11](#_Toc183461202)

[3.3.1. PMOD HYGRO requirements: 11](#_Toc183461203)

[3.3.2. Other requirements: 11](#_Toc183461204)

[**4.** **Design** 12](#_Toc183461205)

[4.1. PMOD HYGRO 12](#_Toc183461206)

[4.1.1. The I2C interface 12](#_Toc183461207)

[4.1.2. Controlling the sensor 14](#_Toc183461208)

[4.2. Converting binary data into temperature, storing and managing it 16](#_Toc183461209)

[4.3. Redesigning the *temperature controller* & *temperature register* components 17](#_Toc183461210)

[4.3.1. First solution 17](#_Toc183461211)

[4.3.2. Second solution (best) 20](#_Toc183461212)

[4.4. The Graphical User Interface 20](#_Toc183461213)

[**5.** **Implementation** 21](#_Toc183461214)

[5.1. PMOD HYGRO 21](#_Toc183461215)

[5.2. Processing the sensor data 25](#_Toc183461216)

[5.2.1. First draft 25](#_Toc183461217)

[5.2.2. Second draft 29](#_Toc183461218)

[5.2.3. Third draft 30](#_Toc183461219)

[5.3. No ESP32 30](#_Toc183461220)

[5.4. Assembling the top-level module 30](#_Toc183461221)

[5.5. Linking the Graphical User Interface 31](#_Toc183461222)

[**6.** **Testing & Validation** 34](#_Toc183461223)

[**7.** **Conclusions & Further development** 34](#_Toc183461224)

[**8.** **Bibliography** 34](#_Toc183461225)

# **Introduction**

## Context

This project focuses on processing the signals coming from sensors in real time. Such data must be collected for further analysis. Any type of sensor should be able to fit into the configuration if the communication protocol it uses is the same as the one the project will be implemented with.

## Objectives

The main part of the project will be designed in VHDL; however, more programming languages may be required to implement the full functionality.

The expected functionality at design time is:

* Input data (e.g. temperature) is read from the sensors and sent to the development board.
* The development board processes this data and/or sends this to an emitter (e.g. Wi-Fi, Bluetooth).
* The emitter links the project (as if it was an all-in-one chip) and a receiver (a computer).
* The computer displays the visual representation of the data[[1]](#footnote-1).

The development stage will be based on the following components: Basys3 FPGA board, 2 compatible pmods (HYGRO and ESP32). These 3 are the core of the project, namely the receiving (temperature) sensor, the transmitter (Wi-Fi/Bluetooth) and the main controller. Some personal research must be done to understand the possibilities/requirements these components offer.

A computer and a small computer

Description automatically generated**A group of white text on a black background

Description automatically generated**The required software (written in VHDL or other programming languages) will be provided, as well as the results for some test cases. These aim to prove the correct functionality of the whole design.

Figure 2. The development diagram of the project.

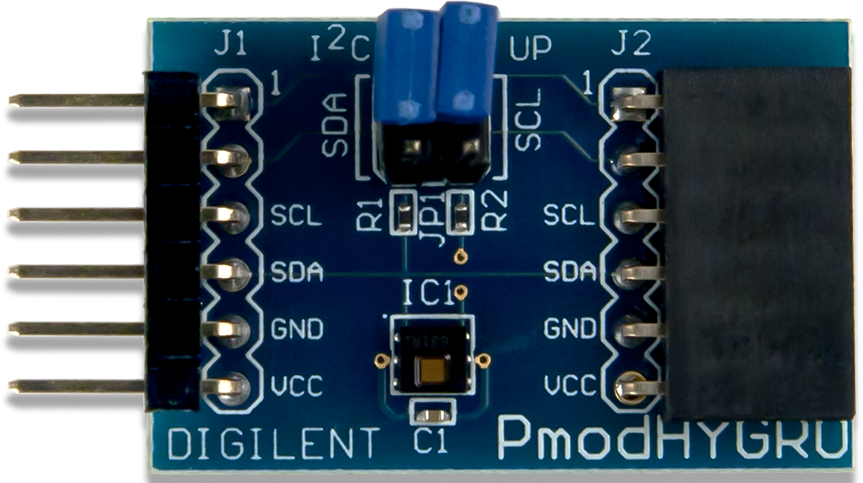
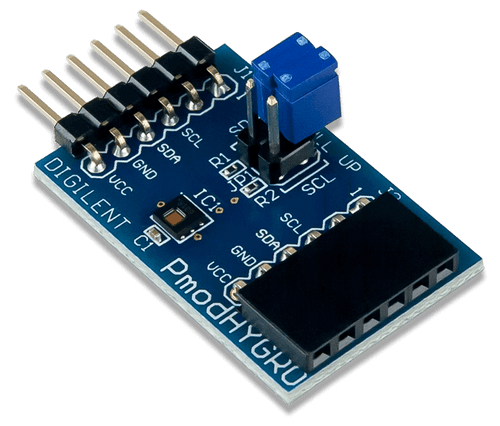
Figure1. The conceptual diagram of the project.

# **Bibliographic Research**

## PMOD HYGRO

Pmod Hygro is a module manufactured by Digilent equipped with the TI HDC1080, a humidity and temperature sensor produced by Texas Instruments.

Figure . PMOD HYGRO



The module offers up to 14 bits of resolution, good stability at high humidity and a great sensitivity – the error is bounded to ±2%, for humidity, and ±0.2ºC, for temperature. An internal resistive heating element is provided to ensure the condensation that may appear when it is used in high humidity environments is driven off. The J2 header passes through all the information in the J1 header to allow daisy chaining.

The Pmod can pe attached to the Basys3 board via the 6-pin connector. The voltage required is between 2.7 and 5.5, providing a 10-400 KHz clock frequency.

### The I²C interface (protocol)[[2]](#footnote-2)

The *Inter-Integrated Circuit* is a synchronous, multi-controller/multi-target, single-ended communication bus, widely used for attaching lower-speed peripheral integrated circuits to processors and microcontrollers in short-distance, intra-board communication.

I²C uses 2 bidirectional, pulled-up with resistors, signals: Serial Data Line (**SDA**) and Serial Clock Line (**SCL**). I²C defines 3 basic types of transactions that all begin with *START* and end with *STOP*:

1. Single message – controller writes data to a target.
2. Single message – controller reads data from a target.
3. Combined format – controller issues at least 2 reads/writes to one or more targets.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| START | First byte | | ACK | I²C message sequence  (X bytes) | STOP |
| I²C address field | Read/Write |
| 7 bits  MSB to LSB | 1 = Read |
| 0 = Write |

Table . I²C 7-bit addressing structure

|  |  |  |  |
| --- | --- | --- | --- |
| First byte | | | Description |
| 7-bit I²C address field | | **R/W value** |
| 0000 | 000 | 0 | General call |
| 0000 | 000 | 1 | Start byte |
| 0000 | 001 | X | CBUS address |
| 0000 | 010 | X | Reserved for different bus |
| 0000 | 011 | X | Reserved for future purpose |
| 0000 | 1XX | X | HS-mode controller mode |
| 1111 | 1XX | 1 | Device ID |
| 1111 | 0XX | X | 10-bit target addressing |

Table . Reserved addresses in 7-bit address space

**The transaction format** of the I²C protocol consists of one or more messages. Each message begins with a *START* and ends with a *STOP*. More *START* symbols placed at the beginning of a transaction are referred to as *repeated start* *symbols*.

A message is either a read or a write. A transaction may consist of a single message (read/write transaction) or multiple messages (combined transaction).

Some limitations of the I²C interface are the *small address space* (the 7-bit protocol is widely used, but does not prevent address collision when thousands of devices are available, while the 10-bit I²C is not supported by many operating systems), *automatic bus configuration* (given addresses may be used by protocol-incompatible devices, device cannot be detected at runtime), *limited range of speeds*, *starving bandwidth* (devices are allowed to stretch clock cycles to suit their particular needs, increasing latencies), being *fault prone* (a fault, error or exception can hang the entire bus, i.e. a device holding *SDL* or *SCL* low will prevent the controller from sending *START* or *STOP* commands).

### Interfacing with the Pmod[[3]](#footnote-3)

The Pmod HYGRO communicates with the host board by means of the I²C interface. Users can both read and configure from the module. Data is sent in such a sequence:

1. the 7-bit I²C address 0x40.
2. a read/write bit.
3. the register address of interest.

Multiple 16-bit registers are accessible: the *configuration register* (address 0x02) allows the user to control the resolution of the measurement, change the acquisition mode, enable or disable the heater etc., the *temperature* (address 0x00) and *humidity* (address 0x01) *registers* are both read-only. The result of the measurement is always stored (regardless of resolution) in the most significant bits, the least 2 significant bits are always 0 for both registers. The higher the resolution the more time the conversion takes.

Upon power-up, the pmod requires 15 ms prior to perform a measurement. To perform a measurement, users need to accept the settings in the *configuration register* and then trigger the measuring process by sending an I²C write transaction paired with the address pointer set to the appropriate register. After waiting for the appropriate time necessary for conversion, users may perform a read transaction. After a read transaction users must wait at least one full second before performing another read transaction to avoid internal heating of the sensor and the distortion of the result. Whenever a write transaction is performed on either *temperature* or *humidity register*, the current conversion will be aborted and a new one started. If a read is performed during the conversion, the pmod will respond back with an ‘unavailable’ signal (NACK).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bit name | Bit number | Bit description | Bit values | Functional description |
| RST | 15 | Reset | 0\* | Normal functioning (self-clearing) |
| 1 | Software reset |
| Reserved | 14 | Reserved | 0 | Must be 0 |
| HEAT | 13 | Heater | 0\* | Heater disabled |
| 1 | Heater enabled |
| MODE | 12 | Acquisition Mode | 0 | Temperature or humidity is acquired depending on which register you choose to read |
| 1\* | Temperature and humidity are acquired in sequence (temperature first) |
| BTST | 11 | Battery Status | 0\* | VDD > 2.8 V (read-only) |
| 1 | VDD < 2.8 V (read-only) |
| TRES | 10 | Temperature Measurement Resolution | 0\* | 14 bit (6.35 ms\*\*) |
| 1 | 11 bit (3.65 ms\*\*) |
| HRES | [9:8] | Humidity Measurement Resolution | 00\* | 14 bit (6.50 ms\*\*) |
| 01 | 11 bit (3.85 ms\*\*) |
| 10 | 8 bit (2.50 ms\*\*) |
| 11 | - |
| Reserved | [7:0] | Reserved | 0 | Must be 0 |

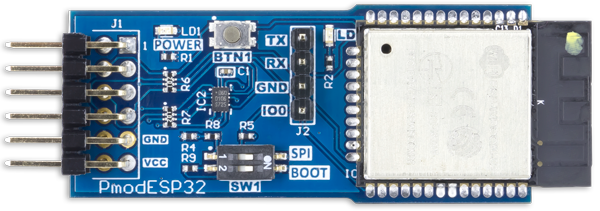
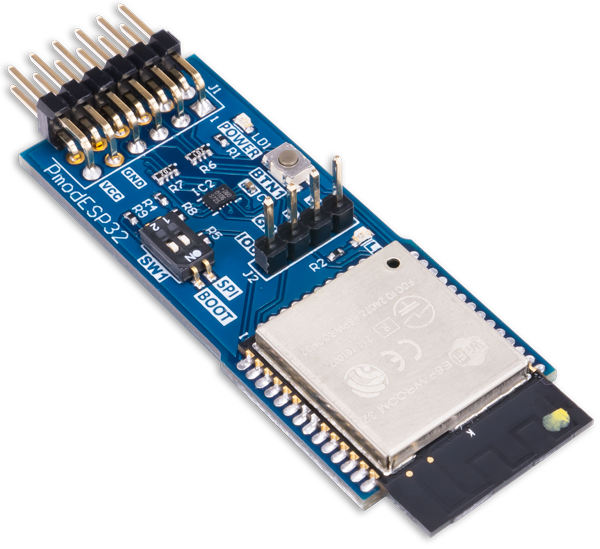
(\* Default value on power-up or reset, \*\* Time required to compute a conversion at that specific resolution)

Table . The Configuration Register (address 0x02)

## PMOD ESP32[[4]](#footnote-4)

The pmod ESP32 contains a Tensilica Xtensa microprocessor that allows operation in target mode over a UART interface. The module can work in a standalone mode as well. An additional UART port is provided on top of the module to make debugging easier.

Figure . PMOD ESP32



The module integrates Wi-Fi and Bluetooth 4.2 on a single chip. It possesses both the ability to be configured as an access point to its own network and to connect to an existing Wi-Fi. In target mode (SPI and GPIO are not used), the pmod responds to a set of AT commands. As a standalone device, a particular application can be uploaded via the J2 header UART connector.

Two switches are placed in the center of the module. The first one (SPI) switches between the SPI and UART interfaces. The second one (BOOT) controls whether the ESP32 boots into an application stored in memory or waits to be flashed with a new one. When the device is powered on, if the BOOT switch is on, the ESP32 will enter a mode where it waits to be flashed with a new application. If the switch is off, the ESP will boot and begin to run whatever application it has stored in memory. The behavior of the SPI switch can be controlled by a host board by means of the SELECT pin in the J1 header (the nineth pin): driving the SELECT pin high causes the top row of the connector to have SPI functionality, regardless of the value of the SPI switch; driving the same pin low causes the top row of the connector to have UART functionality. After flashing the ESP32 with a new application, or when switching between boot modes, RESET must be pressed. Pin 8 on the J1 header controls the functionality of the RESET button (BTN1 on the module).

The module requires between 2.7 and 3.6 V to operate. It offers an SPI serial clock frequency between 2 (typical) and 8.8 MHz and a UART serial clock frequency between 80 – 5 000 000 Baud (typically 115 200). Radio specifications: up to 150 Mbps 802.11 max data rate and up to 4 Mbps Bluetooth HCI max data rate.

|  |  |  |
| --- | --- | --- |
| Pin number | Signal | Description |
| 1 | RTS/SS | UART request to send/SPI target select |
| 2 | RXD/MOSI | UART receive data/SPI controller out target in |
| 3 | TXD/MOSI | UART transmit data/SPI controller in target out |
| 4 | CTS/SCK | UART clear to send/SPI serial clock |
| 5 | GND | Power supply ground |
| 6 | VCC | Power supply (3.3 V) |
| 7 | INT | Configurable GPIO/IO2 |
| 8 | EN | Reset enable |
| 9 | SELECT | UART or SPI mode select |
| 10 | GPIO | Configurable GPIO/IO32 |
| 11 | GND | Power supply ground |
| 12 | VCC | Power supply (3.3 V) |

Table . Pinout table diagram for ESP32

|  |  |  |
| --- | --- | --- |
| Switch | Value | Behavior |
| SPI (SW 1.1) | OFF | UART interface on pins  1 – 4 of Pmod header J1 |
| ON | SPI interface on pins 1 – 4 of Pmod header J1 |
| BOOT (SW 1.2) | OFF | Upon boot, the ESP32 will load the application that is currently stored in memory |
| ON | Upon boot, the ESP32 will not load anything, but wait for a new application to be flashed |

Table . PMOD ESP32 switch behavior

### AT command set[[5]](#footnote-5)

List . The types of AT commands

Rules for creating/using AT commands:

* Not all AT commands support all the above-mentioned four types.
* Only strings and integers are supported as parameters.
* **<>** designate parameters that cannot be omitted.
* []designate optional parameters. If missing, such a parameter is replaced by its default value.
* Multiple parameters are separated by commas.
* Strings need to be included between quotation marks.
* Escape character syntax is required if a string contains special characters.
* Input does not need to be escaped.
* The default baud rate is 115 200.
* The length of a command must not be greater than 256 bytes.
* All commands end with a new line (carriage return + line feed).

|  |  |
| --- | --- |
| ESP-AT response message | Description |
| OK | AT command process done and return OK. |
| ERROR | AT command error or error occurred during execution. |
| SEND OK | Data has been sent to the protocol stack. Data may not have reached the opposite end.  *(specific to AT+CIPSEND and AT+CIPSENDEX)* |
| SEND FAIL | Error occurred during sending the data to the protocol stack.  *(specific to AT+CIPSEND and AT+CIPSENDEX)* |
| SET OK | The URL has been set successfully.  *(specific to AT+HTTPURLCFG)* |
| +<command\_name>: … | Response to the sender that describes the AT command process results in details. |

Table . ESP-AT passive messages

|  |  |
| --- | --- |
| ESP-AT message report | Description |
| ready | The ESP-AT firmware is ready. |
| busy p… | Busy processing. The system is in process of handling the previous command, thus CANNOT accept the new input. |
| ERR CODE:<0x%08x> | Error code for different commands. |
| Will force to restart!!! | Module restart right now. |
| etc. |  |

Table . ESP-AT active messages

### Interfacing with the Pmod

The Pmod ESP32 is shipped to customers with the AT Instruction firmware preloaded into it, in target mode. The top pins are mapped to their UART functionality. It is important to keep in mind that the switches must be off when the device is powered on. To keep the device in target mode the SPI switch must be off. In this mode, AT commands are passed to the pmod via the UART interface on the top row of pins. The UART interface is set to work at 115 200 baud with 8 data bits, 1 stop bit, no parity or hardware control. These settings can be modified by the user.

To make it operate in standalone mode a couple of tools are required: the Xtensa toolchain, the Espressif ESP-IDF, Python and a USB-UART bridge device.

### UART[[6]](#footnote-6)

A universal asynchronous receiver-transmitter is a peripheral device used for serial communication in which the data format and transmission speeds cand be configured. A clock generator, input/output shift registers, transmit/receive FIFO buffers and the required functioning logic are the essential parts of a UART.

A UART takes bytes of data and sends them as individual bits in a sequential manner (usually from the least to the most significant) to another UART device that reassembles them. The shift register is the fundamental component that converts the sequential input into the original data. Three types of communication modes are possible: simplex (in one direction – transmitter to receiver), full duplex (both devices transmit and receive information), half duplex (devices take turns transmitting and receiving). For communication to work both devices must have the same: voltage level, baud rate[[7]](#footnote-7), parity bit, data bits size, stop bits size and flow control.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 bit | 5-9 bits | 0-1 bit | 1-2 |
| Start bit | Data frame | Parity bits | Stop bits |

Table . The UART frame.

A UART frame consists of:

* **Idle** bit (logic 1).
* **Start** bit (logic 0) → signals the receiver that data is coming.
* **Data** bits → usually a character.
* **Parity** bit → not mandatory but helps in error identification.
* **Stop** (logic 1) → end of transmission.

All UART operations are controlled by an internal clock that runs at a multiple of the data rate (typically 8 or 16 times the bit rate[[8]](#footnote-8)). The receiver tests the state of the incoming signal at each clock pulse. If the apparent start bit lasts for at least half of the bit time, it is valid and signals the beginning of a new character, if not it is ignored. After waiting for another bit time, the line is sampled again, and the result is stored in the shift register. After enough time passes (enough bit periods so that 5-8 bits pass), the result is made available to the receiving system. Usually, the UART will set a flag indicating that data is available and may even raise a processor interrupt to request the host processor to process that data. Most UART have a FIFO buffer to prevent losing data when communicating at high rates.

To transmit information, the device waits until a character is stored in the shift register. Once this is done, the start signal is generated, the data, the parity bit (optional) and the stop bit are sent. In full duplex mode, 2 shift registers are used. FIFO buffers are used for the same reasoning here as they were in the receiving process. To prevent communication crashes, an additional busy flag is added to the design.

Typical home computers connected to modems use 8 data bits, no parity and 1 stop bit. In such a configuration, the number of ASCII characters per second equals the bit rate divided by 10.

# **Analysis**

This section does not focus on the thought process that went into designing & implementing this project, but it provides its result.

## Project Proposal

My project aims to be a wireless thermometer simulator. Suppose the house you live in is thermally perfectly isolated from the outside world so that you have no idea what the temperature is outside. In addition to this, you want to optimize the time you have for getting dressed – you want to pick clothes that suit the weather, so you are neither cold nor hot. This is your lucky day because my project intends to let you know about what temperatures is outside without having to go outside.

The final intended features are:[[9]](#footnote-9)

* Temperature reading over a temperature sensor.
* Temperature classification into “buckets”.
* Temperature broadcasting over Wi-Fi.
* Temperature histogram visualization on a display.
* One or more test cases that prove the correctness of the design.
* The source code.

## Project Analysis

### Component interfacing

As the **Bibliographic Research** states, the development of the project is linked to the components that I have purchased from the market: a development board Artix7 - Basys3 and 2 compatible PMODS – a PMOD HYGRO and a PMOD ESP32. These are powerful components that can be programmed according to my requirements. Due to the lack of a built-in interface in the Basys3 board, the main issue[[10]](#footnote-10) regarding the implementation is how I can connect these 3 elements so that all of them are able to function properly. In other words, the PMODS are not plug and play, even though they are specifically crafted to connect to development boards such as Basys3. Each of them comes with a predetermined communication protocol, so to interface with them means to create finite state machines that convert serial information into useful, ready to use data.

The PMOD HYGRO uses the I2C protocol to pass information about the current registered temperature. This means an I2C state machine must be instantiated to fetch the recorded binary data. In addition to this, the manufacturer of the sensor states how to compute the temperature and humidity out of that binary number, meaning that further processing of the serial data will be required.

A glass jar with plants inside

Description automatically generatedThis temperature in Celsius degrees varies, however, due to stability errors and the high resolution of the sensor. Such details offer a lot of possibilities to design a more general, customizable thermometer-based device. Let’s go back to the specifications of the sensor: PMOD HYGRO offers up to 14 bits of resolution for temperature measurements. Thermometers that are available on the market, regardless of their structural details, come with a predefined range, e.g.: clinical thermometers have a range between 35 to 40 degrees Celsius, while industrial thermometers range between 0 to thousands of degrees Celsius. Based on this characteristic and its resolution (±0.1 degrees and couple of degrees for the mentioned examples), we deduce their use. The PMOD HYGRO offers a range from 0 up to over 120 degrees Celsius with a resolution of ±0.01 degrees. This makes the sensor useful in different kinds of measuring scenarios: controlling a self-sufficient environmental system such as a terrarium or simple household appliances (measuring tea/coffee/outside temperature). I intend to design a general measuring hardware system that uses threshold values to place different measuring outputs into corresponding bins. This classification-based approach enables a precision tuning possibility: the user can set his / her own threshold values to make the system more sensitive to temperature modifications. Although a great feature, it implies the user to enter and modify the source code.[[11]](#footnote-11)

Figure . A terrarium is a self-sustainable micro-environment.

### Control logic of the system

Even though the toughest challenge to overcome seems to be the communication between components, implementing this does not mean the job is done – there must be some control logic that receives data from the sensor, classifies it, and sends the result to the ESP32. This additional logic must solve the issue of fetching data from the sensor wrapper and managing it. The histogram computation starts with the “binning” process: input values are classified between predefined thresholds.

CUSUM is a sequential analysis technique used for monitoring change detection.[[12]](#footnote-12) It uses a cumulative sum to signal changes in input data. Samples are assigned weights and summed:

When the value of S exceeds a threshold a change in value has been detected.

Based on this technique, we can design and implement a rougher way to detect changes in the recorded data of the sensor: we modify the concept of bin to accommodate a wider range of relative values. In CUSUM we assign values to bins if they overcome a threshold h, but do not reach the threshold h + 1, while in my approach, we use thresholds to define the limit of a bin. This way, a bin becomes either precise, or general, depending on the thresholds we define. All values that fit between those two limits are placed in the same bin. The first and last bin are infinitely large in opposite directions (the first bin starts from and goes up to the first threshold, while the last bin starts at the last threshold and ends at ). We can do this because we expect the temperature sensor to operate between certain conditions. As presented in [this section](#_Converting_binary_data), the final implementation is designed with modularity and customization possibilities in mind. For a functional sensor and a well-defined partition[[13]](#footnote-13) of the bins, the resulting histogram should come close to a Gaussian bell-shaped curve.

Thus, we require additional logic to map the data into bins and store the number of values in each bin. These can be implemented as register-based components with some arithmetic-logic capabilities. For debugging reasons, the internal state of each such register should be easy to query and display on the seven-segment module of the Basys3 development board. This is an overhead in the designing process, but it can make the difference if bugs, implementation or synthetization errors are encountered in the long run.

## Conclusions:

### PMOD HYGRO requirements:

* An I2C state machine to fetch the data it sends over the serial port.
* A wrapper to handle transactions between the sensor and the board.

### Other requirements:

* An arithmetic-logic component that converts the binary data into valid temperatures as the provider’s formula states.[[14]](#footnote-14)
* A specialized register / register-file component to act as bins (buckets) for the converted binary data. This component must be customizable by the user to support different bin threshold values without much additional effort (using signals to create thresholds, not hard-coded values, generics to create a variable number of bins etc.)
* Additional logic to manage the temperature registers and update their state.
* *(DEBUGGING purposes)* Additional logic to visualize the current value in each temperature register by making use of the seven-segment display that is embedded in the Basys3 development board (a seven-segment display controller and a block of switches). These stored values can be used to compute the required histogram.

# **Design**

## PMOD HYGRO

### The I2C interface

The I2C interface follows a public design found on the internet.[[15]](#footnote-15) This article follows the traditional finite state machine design technique. First, the possible states of the automaton are written down: *READYY, STARTT, COMMANDD, ACKK1, WRITEE, ACKK2, M\_ACKK, STOPP*. These states appear as the designer sketches the normal functioning of the I2C protocol.

At start-up, the components must be put in functional state – *READYY*. The *STARTT* state means the transaction is starting, while the *COMMAND* state communicates the address and read-write command to the bus. *ACKK1* captures the acknowledge of the target component. At this point, the component either writes data to the bus – *WRITEE* – or reads from the bus – *READD*. Once this finishes, the master device captures and verifies the state of the target device – *ACKK2* – or issues its own response – M\_*ACKK*. These 2 pairs of states: *WRITEE* – *ACKK2* and *READD* – M\_*ACKK* loop from one to the another while an enable flag is set. When the flag is unset, the system goes into the STOPP state, that is linked to the *READYY* state, and the process reiterates. The state transition process uses the control signals that appear in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Port name | Bus size | Port mode | Data type | Description | Interface |
| clk | 1 | IN | std\_logic | System clock | User logic |
| reset\_n | 1 | IN | std\_logic | System reset | User logic |
| ena | 1 | IN | std\_logic | 0,  no transaction is initated | User logic |
| 1, latches in addr, wr, and data\_wr to initiate a transaction. If ena is high at the conclusion of a transaction then the new address, read/write command, and data are latched to continue the transaction. |
| addr | 7 | IN | std\_logic\_vector | Address to the target device. | User logic |
| rw | 1 | IN | std\_logic | 0, write command. | User logic |
| 1, read command. |
| data\_wr | 8 | IN | std\_logic\_vector | Data to transmit if we write on the bus. | User logic |
| data\_rd | 8 | IN | std\_logic\_vector | Data to read if we read from the bus. | User logic |
| busy | 1 | OUT | std\_logic | 0, I2C is idle and the last read data is available. | User logic |
| 1, command has been latched in and a transaction is in process. |
| ack\_error | 1 | BUFFER | std\_logic | 0, no acknowledge errors. | User logic |
| 1, acknowledge errors appeared during the transaction, self-clearing. |
| sda | 1 | INOUT | std\_logic | Serial data line | Target devices |
| scl | 1 | INOUT | std\_logic | Serial clock line | Target devices |

Table . Port description of I2C finite state machine.[[16]](#footnote-16)

Transactions can be initiated when the *busy* signal is LOW. At this moment the user can place the desired target device address, *addr*, the read-write command, *rw*, the data he / she wants to send on the *data\_wr* bus and set the *ena* flag. Thus, the transaction starts. Once it is completed, the automaton unsets the *busy* flag and reports errors via the *ack\_error*. To chain more read / write commands, the *ena* signal and the new inputs must be set no later than the last bit of the current command. If errors happened during the transaction, the command is not sent back over the protocol. The *reset\_n* user input provides the system reset functionality. Being active low, it assures that the system starts in a valid state.

A diagram of a computer

Description automatically generated

Figure . State transition diagram.[[17]](#footnote-17)

### Controlling the sensor

The PMOD HYGRO controller needs a separate state machine[[18]](#footnote-18) that makes use of the previous I2C component. Its purpose is to place data on the bus when needed, wait until the transaction is processed, tune the resolution of the sensor and fetch sensor data when told so.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Port | Bus width | Mode | Data type | Interface | Description |
| clk | 1 | IN | std\_logic | User logic | System clock |
| reset\_pmod | 1 | IN | std\_logic | User logic | Asynchronous active LOW reset |
| scl | 1 | INOUT | std\_logic | PMOD HYGRO | Serial clock of I2C bus |
| sda | 1 | INOUT | std\_logic | PMOD HYFRO | Serial data of I2C bus |
| i2c\_ack\_error | 1 | OUT | std\_logic | User logic | I2C communication error (acknowledge flag) |
| relative\_humidity | 8/11/14 | OUT | std\_logic\_vector | User logic | Sensor recorded data |
| temperature | 11/14 | OUT | std\_logic\_vector | User logic | Sensor recorded data |

Table . Controller port description.[[19]](#footnote-19)

A diagram of a diagram

Description automatically generated

Figure . The state diagram of the PMOD HYGRO controller.

The state diagram describes the process the sensor shall loop through. After a 100ms delay that is required by the producer to ensure the measurement accuracy, the system enters a configuration state, where the resolution and the internal values are initiated. Next, we enter an infinite cycle that performs the initiation of a measurement, a pause until the sensor can decide what value to output and send over the I2C protocol; we read that data, and output this to higher-level modules (or to the LEDs for debugging). This process starts over and over again as long as the sensor is powered.

## Converting binary data into temperature, storing and managing it

In designing a component that handles these operations, I used a top-down approach. First, I noticed that there is need of synchronization (values must be stored inside some kind of registers), so the sequential logic techniques must be used. The component begins as a black box with inputs – clock, reset (because it is a sequential circuit), select bin (for debugging purposes), raw data from sensor (the data the sensor sends over the I2C protocol), and outputs A white square with black text

Description automatically generated– anodes & cathodes (to seven segment displays).

Figure .The black box of the top-level component of the current design.

A screenshot of a computer

Description automatically generatedNow let’s start to deconstruct this component into smaller, more specialized systems: a seven-segment-display-responsible component, registers corresponding to the number of bins where temperature will be stored, and a way to choose between them.

Figure . The break-up of the top module in case of 2 bins.

This diagram[[20]](#footnote-20) is composed of low-level modules that can be implemented directly in VHDL, so no further break-up is required. Further analysis of the design shows that the temperature registers (bin0, bin1) can be implemented as counters with enable. That happens because once the comparator signals the demultiplexer to pick a bin, the internal state of the register is incremented just by one, equivalently, when enable is asserted, the counter increments by one. The comparator acts as the main control of the system: it is fed some reference (threshold) values and the data coming from sensor in its binary form. It classifies the measured value into the correct bin by means of multiple unsigned comparisons. A possible optimization of the design is removing the steps to compute the temperature value from the sensor data. This happens because the function we use to map the data into useful temperature data (in degrees Celsius) is linear, meaning that the order relation between two values is preserved. The last step is to create the seven-segment logic. Seven-segment displays come in several configurations (common anode, common cathode, etc.). The Basys3 development board uses the common anode configuration, but all the four displays are connected to the same cathodes. Thus, when displaying a 16-bit hexadecimal number, the anode of each segment (out of the four) must be powered on in quick succession, with the value placed on cathodes changing based on the digit we want to show. This implies that the seven-segment logic needs an internal state that controls the digit we are sending via the cathodes and the anode that should be powered on. A frequency divider may provide an internal clock that will handle this multiplexing problem.

For debugging purposes, the *bit selector* signal lets the user manually check the status of each temperature register (bin) over a binary encoding. The number of the bin in binary form picks the register from which the 16-bit value is fetched and displayed on the seven-segment display in hexadecimal format.

The reset signal (not shown on the diagram) links all the sequential components (all bins) to the global system reset. When this is set, the internal state of each component goes back to its initial state.

## Redesigning the *temperature controller* & *temperature register* components[[21]](#footnote-21)

Back to the drawing board we go.

Even though the expected result is different from what I imagined in the beginning, the thought process can stay the same. There is no need to modify the driving ideas as the binning process is not wrong, but its implementation is not the best. In fact, it is the bottleneck of the resulting system; the component that greatly limits the frequency of the system. This being said, let’s return to the core idea: I want to have a sub-system that can decide what bin should be incremented based on a value that is fed from outside. How can I optimize the comparison process? The initial solution (the one I despise now), can be looked at as a for-loop that compares each element to the result of the previous comparison, storing & returning the index of the bin that fits best. This yields linear complexity. A better way to assign the responsibilities in this sub-system is to transfer some of the load to the *temperature register*. Let each *temperature register* be able to perform one comparison.

### First solution

Such a register allows me to implement a sort of binary tree. The increasing complexity of the design is counterbalanced by the decrease of the component latency. The same number of components, but they are rearranged.

A black background with white rectangles

Description automatically generated The structure above solves out issue provided with a good heuristic. The solution I came up with is to use a modified hardware-based binary search tree that is presented in the figure below.

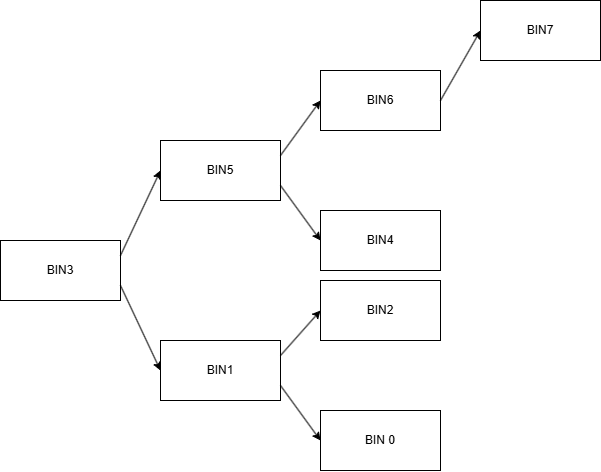


Figure . The distribution of bins.

Figure . The diagram of the new design.

Each bin is a register and is fed three inputs: two threshold values (min and max) and a data signal or a signal from the previous register. The result of the comparison between the threshold values and the data that enters the register is used to decide between two possibilities: either pass the value down to the next register (if the value is smaller, the one to the left, if the value is greater, the value to the right) or increment its internal state and pass NULL (0x0000) to its successors.

BIN0 and BIN7 store values with no precision, i.e. values between and . The rest of the bins simulate a binary search algorithm: the data from the sensor enters BIN3[[22]](#footnote-22) and passes through a pipeline of at most (in the demo that is ).

Let the following be the threshold values:

* 14 ºC 🡪 threshold\_value0
* 16 ºC 🡪 threshold\_value1
* 18 ºC 🡪 threshold\_value2
* 20 ºC 🡪 threshold\_value3
* 22 ºC 🡪 threshold\_value4
* 24 ºC 🡪 threshold\_value5
* 26 ºC 🡪 threshold\_value6

**Algorithm traces:**

1. Suppose the sensor measures 25ºC.
   1. *Greater* In the first clock cycle, data enters BIN3 (20ºC), BIN3 preserves its internal value and reroutes the input to BIN5 (24ºC).
   2. *Greater* In the second clock cycle BIN5 preserves its internal state and reroutes the input to BIN6(26ºC).
   3. *Suitable* In the third clock cycle BIN6 updates its internal state (+1) and does NOT reroute its data to its successors (if any).
2. Suppose the sensor measures 16.5ºC.
   1. *Smaller* In the first clock cycle data enters BIN3 (20ºC), BIN3 preserves its internal value and reroutes the input to BIN1(16ºC).
   2. A screen shot of a white rectangular object

      Description automatically generated*Suitable* In the second clock cycle data enters BIN1, BIN1 updates its internal state (+1) and does NOT reroute its data to its successors (if any)

Figure . Block diagram of a temperature register component.

### Second solution (best)

The first solution is overcomplicated and limits the modularity of the system. An even better approach is to use a simple queue of registers. Sensor data passes from one register to another (a pipeline), if the value is smaller than the threshold value corresponding to the register, its internal state is incremented.



Figure . The block diagram of the queue.

A white rectangular object with black text

Description automatically generatedIn such a scenario the black box of the temperature register becomes:

Figure . The black box of the temperature register

The queue implements a first-in-first-out policy meaning that the first value that enters is the first one that exits after it moves through all the stages. In my case, the temperature data coming from the sensor enters the first register (first bin) where it is compared to the smallest threshold value, if it is smaller the value of the bin increments and the value is NOT passed down the pipeline. To ensure all values fit inside the 7-bin demo configuration, the last register has the maximum readable sensor value (0xFFFF), i.e. there is no greater value the sensor can read that does not fit in that bin.

## The Graphical User Interface

Because I already know the basic functionality of the pair *raylib.h* – *C++* to create in-console graphics, I used them. I believe the GUI should be simple and clear, in other words, I want the histogram to be the focal point of the window. *raylib.h* offers basic kernel-level functions to interact with drawing primitives. Consequently, I must design everything that appears on the screen by means of simple 2D figures: the plot itself, as well as the axes. Regarding colors, I picked black for the background, white for the axes and labels to ensure a great contrast and red for the histogram to attract the view there.

The axes will be constructed each as 2 back-to-back right triangles placed at the end of a slim rectangle; some basic mathematics are required to compute midpoints, lengths etc. to ensure symmetry and proportionality. Once the axes are placed on the screen, the usable area must be computed to ensure columns with the same width that are evenly spaced. This was done on the spot (a few simple subtractions based on where the coordinates of the drawing primitives were placed). A sketch is presented below.

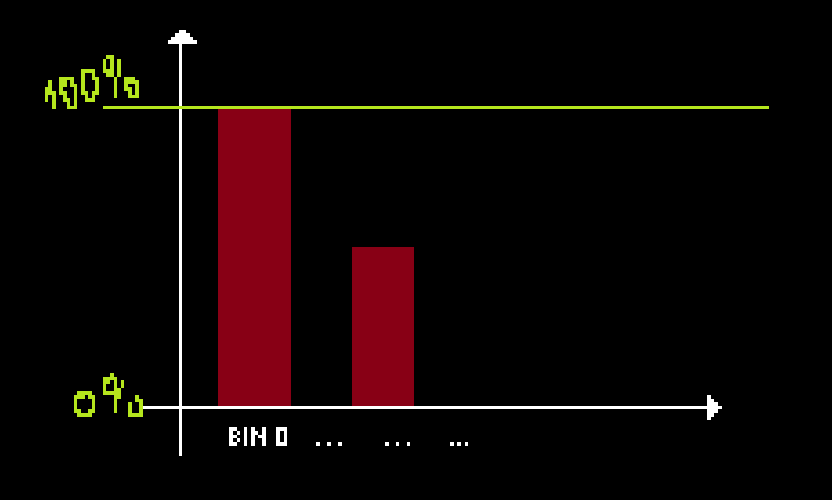


Figure . The sketch of the GUI.

To ensure that the columns do not reach above the height limit (a design-level bug), I intend to dynamically convert the values into percentages. I plan to set a fixed 100% checkpoint, i.e. 100 units (meaning that each sensor read is placed inside a bin with a weight of 1, that is converted into a greater rectangle height).

# **Implementation**

## PMOD HYGRO

As stated in the [previous section](#_PMOD_HYGRO), to make use of the PMOD HYGRO, a finite state machine that instantiates an I2C interface is required. Once the state transition diagram is drawn, all that is left is to decide which signals are asserted in one state. The choice of implementation is the canonical one: one big process that takes care of both state changes and signal assignments. Another process is created to generate the timing of the serial clock line.



Figure . The entity of the I2C component.



Figure . READYY state signal assignment



Figure . READD state signal assignments.[[23]](#footnote-23)

The i2c is used as a “black box” in a higher-level component *hygro*:



Figure . The entity of the PMOD HYGRO controller.[[24]](#footnote-24)

Figure . Multiplexing between resolution values based on the value of the generic parameter TEMPERATURE\_RESOLUTION.[[25]](#footnote-25)



Figure . INITIATEE state signal assertion.[[26]](#footnote-26)

## Processing the sensor data

### First draft

To process the data that is coming from the sensor we proceed by implementing the *temperature register*. As established in the [previous section](#_Control_logic_of), this component must act as a counter with enable. When the enable signal is asserted, the counter increments by one, otherwise it preserves its value. I opted for a 16-bit design because this is the maximum number I can comfortably display on the integrated seven-segment display module of the Basys3 development board. A clock signal ensures the synchronization requirements, and a reset signal lets the user link the internal state of the register to the functioning state of the whole system.



Figure . Structure of the temperature register.

The next component I implemented is the *seven-segment display*. This wraps up all the logic that goes into lighting up the corresponding anodes, sending the correct digit over the cathodes (i.e. first digit to the first seven-segment, and so on). Predefined functions of the ieee.numeric\_std library proved to be of real use. A frequency divider paired with a multiplexer does the job of powering up the anodes in quick succession and, thus, creating the illusion of a number being displayed (in fact, digits are selected from 4 pools and send to the only anode that is being active at a high enough frequency). In addition to these, a constant array that stores the seven-segment encodings of the hexadecimal digits is required.



Figure . The entity declaration and the seven-segment digit code look-up table.



Figure . The architecture of the seven-segment component.

These 2 components must be managed by a top module, called *temperature register controller.*  Here, the threshold values are instantiated, the number of bins is set and the binning process takes place. A demultiplexer enables the required *temperature register* to signal that a value corresponding to bin *x* (between the (*x-1*)stthreshold and the *x*-ththreshold) has been registered. A process with a chained if-elsif statement provides this functionality. As far as the debugging tools are concerned, a multiplexer selects what *temperature register*’s internal state (value) is sent to the seven-segment display.



Figure . The entity of the register controller component.



Figure . The threshold values for the demo design.

At this point I realized my design has a major flaw: I intended to create a modular system that can be expanded with high precision and low cost (i.e. add more registers in just a few lines of code to improve the efficiency of the binning process); this approach resulted in implementing a *temperature register* component that has minimal load because everything is supported by the *comparator* module. Thus, the *comparator* becomes the **bottleneck** of the system. Moreover, the on-board, physical implementation that is uploaded on the FPGA uses a chain of comparators where the output of the previous is compared to a threshold instead of the (supposed by me) 2-by-2 contest algorithm[[27]](#footnote-27). This moves us back to the [Redesigning the temperature controller & temperature register components](#_Redesigning_the_temperature).

### Second draft

The source code of the new implementation is presented below.



Figure 25. Entity declaration.



Figure 26. Updated counter process of the temperature register

### Third draft

The source code of the new implementation is presented down below.



Figure . The entity of the FIFO-type register.



Figure . The core of the temperature register component.

## No ESP32 😔

Due to the lack of time and the difficulties I faced with the integration of the ESP32 I must remove it from the design, replacing it with a UART connection over the power connection cable. The design of the UART module is available at [https://github.com/](https://github.com/jakubcabal/uart-for-fpga).

## Assembling the top-level module

Assembling the top-level felt like LEGO. I have just instantiated the lower-level modules as components and linked them with wires. By using a bottom-up technique to build each sub-system I ensured that they act as building blocks for higher-level modules. Consequently, not much additional logic was required.

For the moment the PMOD ESP32 is replaced by a simple UART communication handled by the Basys3 development board over its power connection by means of the integrated receiver and transmitter in addition to a VHDL UART controller.



Figure . Snipped from the architecture of the top-level module.

## Linking the Graphical User Interface

A quick walkthrough of the GUI implementation process. As I have stated previously, the first step was to create a working media: a *raylib.h* compatible C++ project. I opted for an object-oriented design. At first I created the plot class that manages the drawing of the axes, the spacing between the bins and the update of the bin values:

class Plot {

public:

    Plot(const int howManyBins);

    void setData(std::vector<float> newData);

    void incrementValue(const unsigned int binNumber, const float newValue);

    void draw();

    ~Plot();

private:

    void drawBinLabels();

    void drawColumns();

    int computeHeightOfColumn(const unsigned int binIndex);

    void drawAxes();

    void drawOY();

    void drawOX();

    std::vector<float> data;

    int numberOfBins = -1;

};

Following this, I had to dig into some Windows API calls to create a handle to a COM port so that I can read the data that my sensor passes over UART.

class ComPort

{

public:

    ComPort(std::string portName);

    int read();

    ~ComPort();

private:

    HANDLE createHandle() noexcept(false);

    void createProperties() noexcept(false);

    void createTimeouts() noexcept(false);

    char \*readBytes(const int numberOfBytes) noexcept(false);

    std::string convertBufferToString(char \*data);

    int convertCharsToInt(std::string data);

    int computeInput(std::string data);

    int convertHexaToInt(const char hexD);

    void clearBuffer(const int numberOfBytes, char \*buffer);

    HANDLE serialHandle;

    std::string portName;

};

const unsigned int NUMBER\_SIZE = 16;

The class above is my handle to the COM port, it manages, reads and converts the data so that I can use it in the GUI.

void ComPort::createProperties() noexcept(false)

{

    DCB dcbSerialParam = {0};

    dcbSerialParam.DCBlength = sizeof(dcbSerialParam);

    if (!GetCommState(serialHandle, &dcbSerialParam))

        throw std::runtime\_error("Cannot get the state of the port " + portName);

    dcbSerialParam.BaudRate = CBR\_115200;

    dcbSerialParam.ByteSize = 8;

    dcbSerialParam.StopBits = ONESTOPBIT;

    dcbSerialParam.Parity = NOPARITY;

    if (!SetCommState(serialHandle, &dcbSerialParam))

        throw std::runtime\_error("Cannot set the state of the port " + portName);

}

The main.cpp represents the entry point of the GUI, here the interfacing between the above-mentioned objects cand be noticed.

int main()

{

    ComPort \*myPort = new ComPort("COM5");

    InitWindow(WINDOW\_WIDTH, WINDOW\_HEIGHT, WINDOW\_NAME);

    Plot \*myPlot = new Plot(NUMBER\_OF\_BINS);

    std::vector<float> v = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f};

    myPlot->setData(v);

    SetTargetFPS(120);

    while (!WindowShouldClose())

    {

        BeginDrawing();

        ClearBackground(BACKGROUND\_COLOR);

        int x = myPort->read();

        if (x > 0)

        {

            int bin = getBinBasedOnValue(x);

            myPlot->incrementValue(bin, 0.3f);

        }

        myPlot->draw();

        EndDrawing();

    }

    return 0;

}

With this setup, all that is left to do is to write the bitstream onto the Basys3 development board after the sensor is plugged into the JB header, turn the reset on (active LOW), and run the GUI from an IDE such as VS Code. Tuning may be required on both ends; the GUI is designed to work on my current settings (COM5) with fixed threshold values on both the GUI and on the board (see *Parameters.hpp* and *temperature\_register\_controller.vhd*).

# **Testing & Validation**

# **Conclusions & Further development**

# **Bibliography**

|  |  |
| --- | --- |
| [1] | Digilent, "PMOD HYGRO reference manual," 22 February 2017. [Online]. Available: https://www.farnell.com/datasheets/2243501.pdf?\_gl=1\*63u4b2\*\_gcl\_aw\*R0NMLjE3Mjc4ODk0NDAuQ2owS0NRanczdk8zQmhDcUFSSXNBRVdibGNETnBoOU5faFVJT05mZk9RLVEyVUhYdDE1RjlXWnZGYUpVbHV0bEZJN3NkYlo1emtHOENUSWFBdVFuRUFMd193Y0I.\*\_gcl\_au\*NzM3NTE2MzMuMTcyNzg3MDcxMg... |
| [2] | Texas Instruments, "HDC 1080," January 2016. [Online]. Available: https://www.ti.com/lit/ds/symlink/hdc1080.pdf. |
| [3] | Scott\_1767, "Humidity and Temperature Sensor Pmod Controller (VHDL)," 6 March 2021. [Online]. Available: https://forum.digikey.com/t/humidity-and-temperature-sensor-pmod-controller-vhdl/13064?\_ga=2.24350906.1099841098.1728495616-27067341.1728495616. |
| [4] | wikipedia.org, "CUSUM," [Online]. Available: https://en.wikipedia.org/wiki/CUSUM. [Accessed 23 October 2024]. |
| [5] | "Difference between master and slave devices in communication network," March 2021. [Online]. Available: https://electronics.stackexchange.com/questions/554170/difference-between-master-and-slave-devices-in-communication-network. |
| [6] | wikipedia.org, "I²C," [Online]. Available: https://en.wikipedia.org/wiki/I%C2%B2C. [Accessed 25 October 2024]. |
| [7] | S. Campbell, "Basics of the I2C Communication Protocol," February 2016. [Online]. Available: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/. |
| [8] | Scott\_1767, "I2C Master (VHDL)," 11 March 2021. [Online]. Available: https://forum.digikey.com/t/i2c-master-vhdl/12797. |
| [9] | Digilent, "Digilent Pmod™ Interface Specification 1.2.0," 5 October 2017. [Online]. Available: https://digilent.com/reference/\_media/reference/pmod/pmod-interface-specification-1\_2\_0.pdf. |
| [10] | Espressif, "ESP32 Seris Datasheet," September 2024. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32\_datasheet\_en.pdf. |
| [11] | Digilent, "Pmod ESP32 Reference Manual," [Online]. Available: https://digilent.com/reference/pmod/pmodesp32/reference-manual. [Accessed 17 October 2024]. |
| [12] | espressif.com, "ESP-IDF Programming Guide," [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html. [Accessed 17 October 2024]. |
| [13] | R. Danescu, "Design with Microprocessors – 2023-2024," 2024. [Online]. Available: https://users.utcluj.ro/~rdanescu/teaching\_pmp.html. |
| [14] | R. Withu, "Tournament Tree (Winner Tree) and Binary Heap," geeksforgeeks, 2 June 2023. [Online]. Available: https://www.geeksforgeeks.org/tournament-tree-and-binary-heap/. [Accessed 12 November 2024]. |
| [15] | "raylib," [Online]. Available: https://www.raylib.com/. [Accessed 25 11 2024]. |

1. To facilitate data interpretation a visual representation (e.g. histogram) should be provided to the users. [↑](#footnote-ref-1)
2. Adapted from [6] [↑](#footnote-ref-2)
3. Adapted from [1] [↑](#footnote-ref-3)
4. Adapted from [10] [↑](#footnote-ref-4)
5. Adapted from [12] [↑](#footnote-ref-5)
6. https://en.wikipedia.org/wiki/Universal\_asynchronous\_receiver-transmitter [↑](#footnote-ref-6)
7. Unit of measurement for the symbol rate, measures the speed of communication over a data channel. [↑](#footnote-ref-7)
8. The number of bits that can be processed in a unit of time. [↑](#footnote-ref-8)
9. The implementation is based on certain, physical, available on the market components. [↑](#footnote-ref-9)
10. There are 2 such problems: one for each PMOD. [↑](#footnote-ref-10)
11. The same reasoning applies to relative humidity measuring. [↑](#footnote-ref-11)
12. Adapted from [4] [↑](#footnote-ref-12)
13. A well-defined partition for measuring temperature is created by setting a series of threshold values centered around an expected or estimated value. For instance, if we aim to measure the outside temperature in Cluj-Napoca during October, a reasonable estimate might be 20 degrees Celsius. We then establish a finite number of thresholds at 2-degree intervals both above and below this central value. [↑](#footnote-ref-13)
14. Because the formula is linear, the order relation between different values is preserved. This means that the conversion from binary to degrees Celsius may not be required at all by the design. [↑](#footnote-ref-14)
15. Adapted from [8] [↑](#footnote-ref-15)
16. Adapted from [8] [↑](#footnote-ref-16)
17. Adapted from [8] [↑](#footnote-ref-17)
18. Adapted from [3] [↑](#footnote-ref-18)
19. Adapted from [3] [↑](#footnote-ref-19)
20. For more bins, the design can be adapted, i.e. the selection signals of the multiplexer, demultiplexer and the number of registers increase (plus the additional wiring). However, the logic remains the same. [↑](#footnote-ref-20)
21. This section follows the first part of the implementation chapter. Please jump it for the moment. [↑](#footnote-ref-21)
22. BINx holds the x-th and (x+1)st threshold values. [↑](#footnote-ref-22)
23. Adapted from [3] [↑](#footnote-ref-23)
24. Adapted from [3] [↑](#footnote-ref-24)
25. Adapted from [3] [↑](#footnote-ref-25)
26. Adapted from [3] [↑](#footnote-ref-26)
27. [14] [↑](#footnote-ref-27)