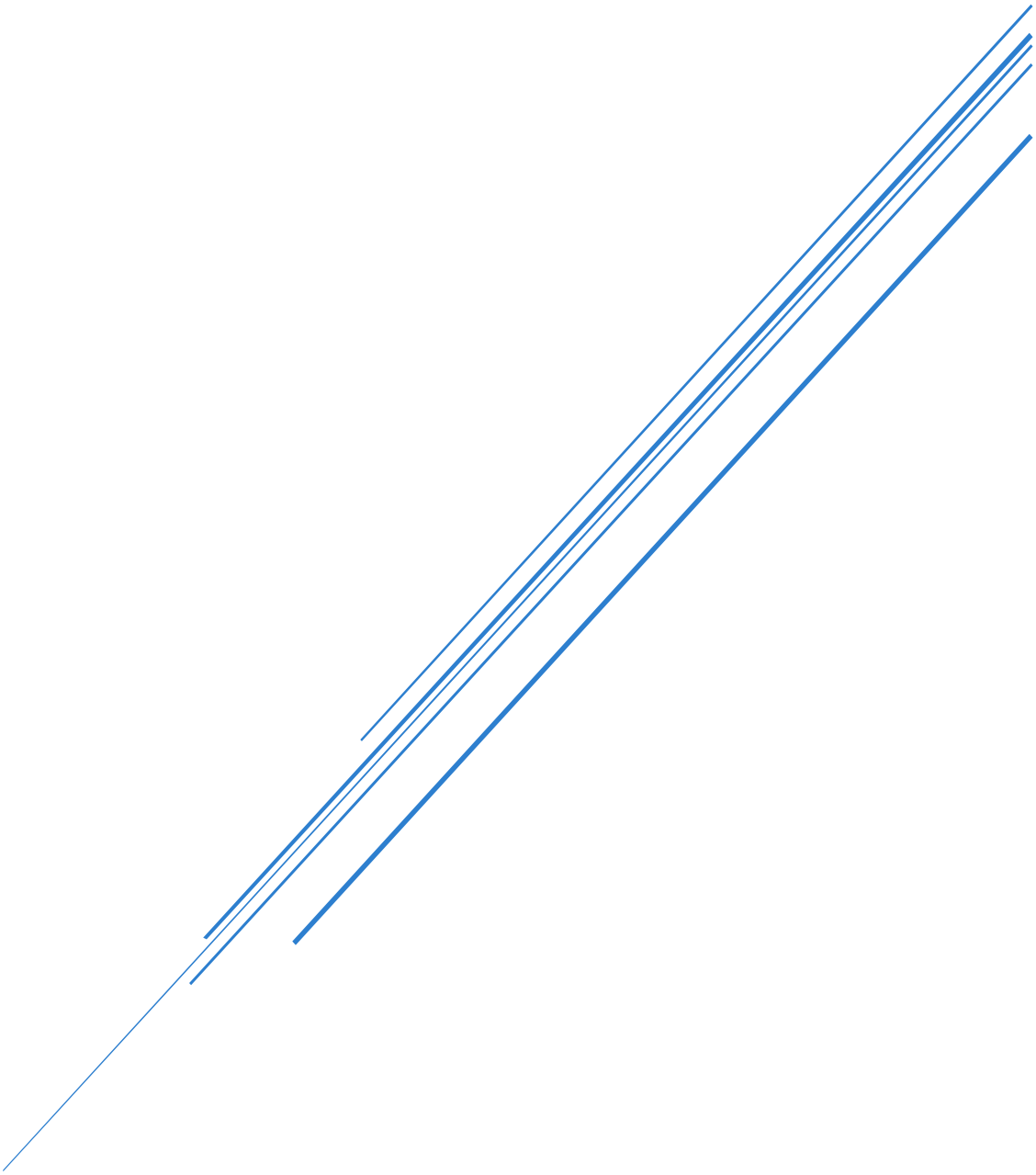
Processing signals received from sensors

Student: Laurentiu-Calin Grad

2025



Technical University of Cluj-Napoca
Structure of Computer Systems Project

Contents

1. Introduction

1.1. 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.

1.2. 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].

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.

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.

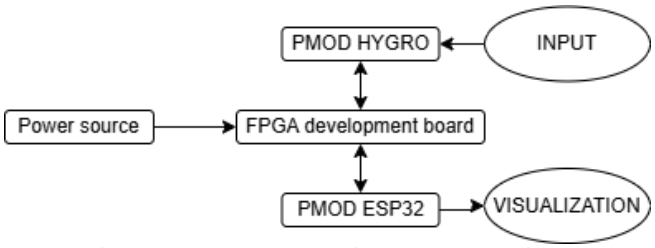


Figure1. The conceptual diagram of the project.

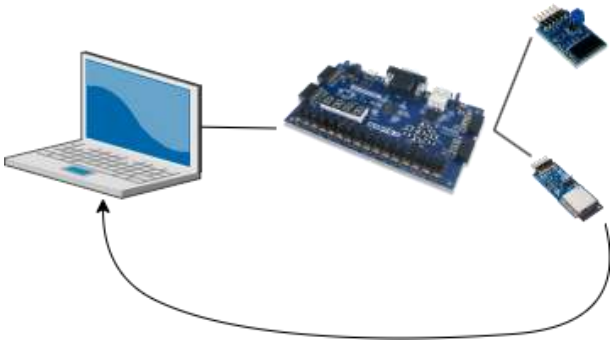


Figure 2. The development diagram of the project.

[1] To facilitate data interpretation a visual representation (e.g. histogram) should be provided to the users.

2. Bibliographic Research

2.1. PMOD HYGRO

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

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.

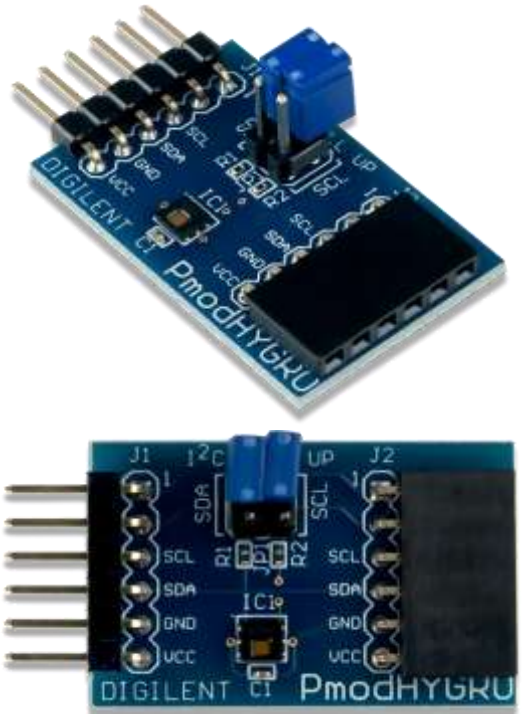


Figure 1. PMOD HYGRO

2.1.1. The I²C interface (protocol)[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*:

i. Single message – controller writes data to a target.
ii. Single message – controller reads data from a target.
iii. 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 1. I²C 7-bit addressing structure

First byte			Description
7-bit I²C address field		R/W value	
0000	000	0	General call

[2] Adapted from [6]

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 2. 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).

2.1.2. Interfacing with the Pmod[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:

i. the 7-bit I²C address 0x40.
ii. a read/write bit.
iii. 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

[3] Adapted from [1]

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 3. The Configuration Register (address 0x02)

2.2. PMOD ESP32[4]

Figure 2. PMOD ESP32

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.

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

[4] Adapted from [10]

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 4. 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 5. PMOD ESP32 switch behavior

2.2.1. AT command set[5]

TEST COMMAND	AT+<command_name>=?
• Query the internal parameters and their range of values of the command set.	
QUERY COMMAND	AT+<command_name>?
• Return the current values of the parameters.	
SET COMMAND	AT+<command_name>=<...>
• Set the value of the user-defined parameters in commands and run these commands.	
EXECUTE COMMAND	AT+<command_name>
• Run commands with no user-defined parameters.	

List 1. 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.

[5] Adapted from [12]

- 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 6. 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 7. ESP-AT active messages

2.2.2. 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.

2.2.3. UART[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], 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 8. 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]). 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.

[6] https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
[7] Unit of measurement for the symbol rate, measures the speed of communication over a data channel.
[8] The number of bits that can be processed in a unit of time.

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.

3. Analysis

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

3.1. 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]

- 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.

3.2. Project Analysis

3.2.1. 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] 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.

---

[9] The implementation is based on certain, physical, available on the market components.

[10] There are 2 such problems: one for each PMOD.

$$temperature\ (℃) = \frac{sensor\ data[15:0]}{2^{16}} \cdot 165 - 40$$

$$relative\ humidity\ (\%RH) = \frac{sensor\ data[15:00]}{2^{16}} \cdot 100\%RH$$

This 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]

Figure 3. A terrarium is a self-sustainable micro-environment.

3.2.2. 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

[11] The same reasoning applies to relative humidity measuring.

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] It uses a cumulative sum to signal changes in input data. Samples x_n are assigned weights ω_n and summed:

$$S_{n+1} = \max(S_n + x_{n+1} - \omega_n, 0)$$
$$S_0 = 0$$

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 $-\infty$ and goes up to the first threshold, while the last bin starts at the last threshold and ends at $+\infty$). We can do this because we expect the temperature sensor to operate between certain conditions. As presented in this section, the final implementation is designed with modularity and customization possibilities in mind. For a functional sensor and a well-defined partition[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.

3.3. Conclusions:

3.3.1. 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.

3.3.2. Other requirements:

- An arithmetic-logic component that converts the binary data into valid temperatures as the provider's formula states.[14]

[12] Adapted from [4]

[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.

[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.

- 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.

4. Design

4.1. PMOD HYGRO

4.1.1. The I2C interface

The I2C interface follows a public design found on the internet.[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	

[15] Adapted from [8]

				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 9. Port description of I2C finite state machine.[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.

[16] Adapted from [8]

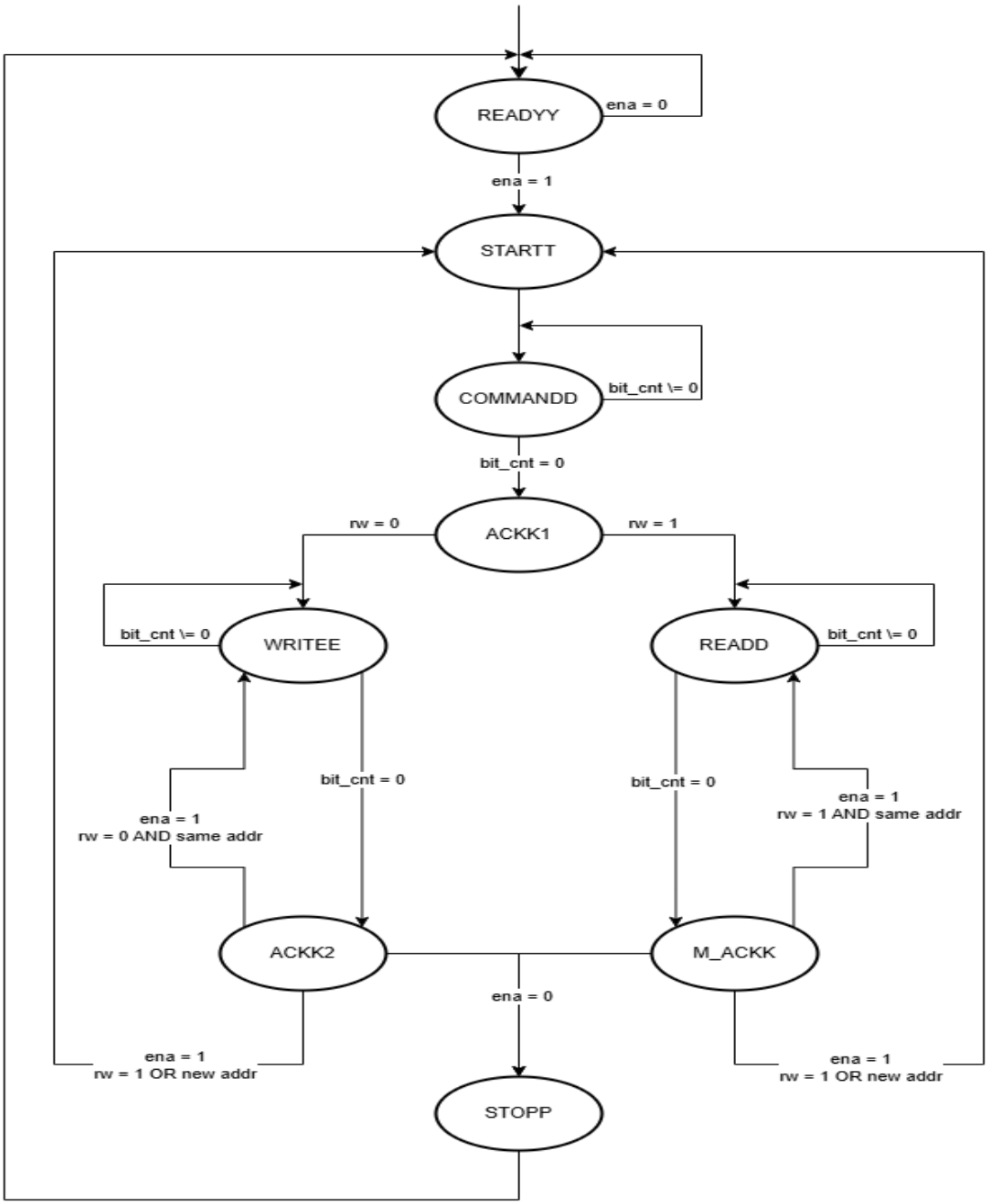


Figure 4. State transition diagram.[17]

4.1.2. Controlling the sensor

The PMOD HYGRO controller needs a separate state machine[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.

[17] Adapted from [8]
[18] Adapted from [3]

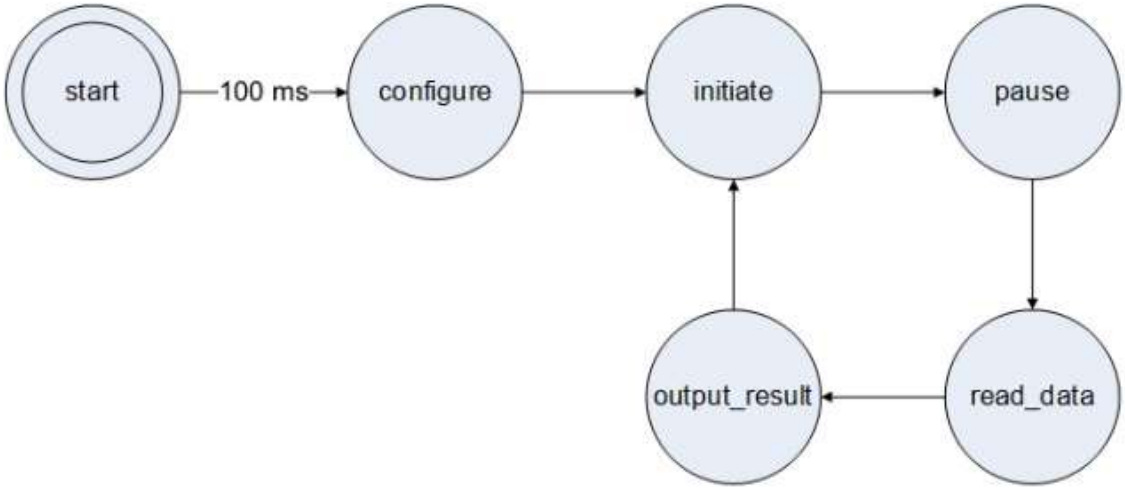


Figure 5. The state diagram of the PMOD HYGRO controller.

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 10. Controller port description.[19]

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.

[19] Adapted from [3]

4.2. 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

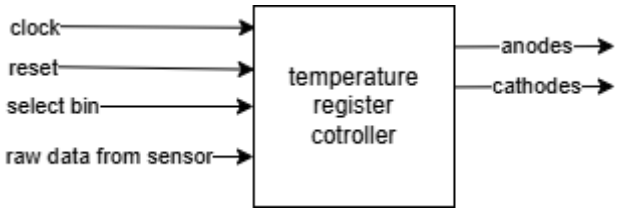


Figure 6.The black box of the top-level component of the current design.

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 – anodes & cathodes (to seven segment displays).

Now 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.

This diagram[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

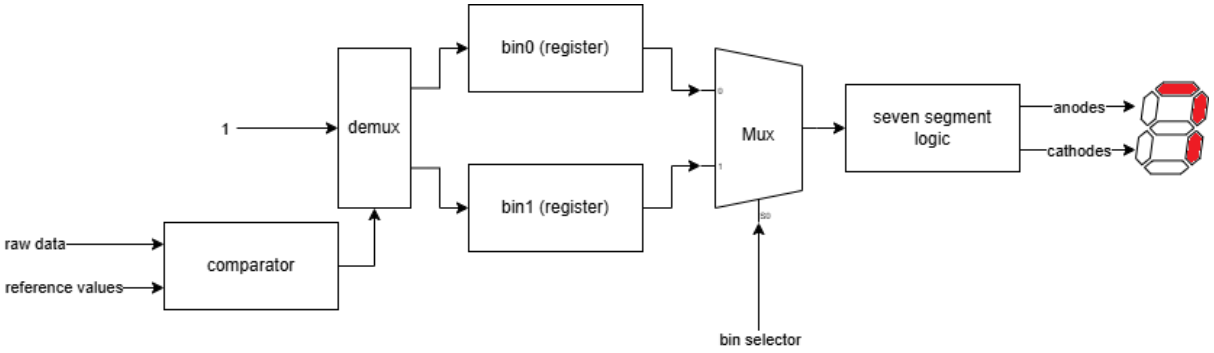


Figure 7. The break-up of the top module in case of 2 bins.

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

[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.

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.

4.3. Redesigning the *temperature controller* & *temperature register* components[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.

4.3.1. 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.

[21] This section follows the first part of the implementation chapter. Please jump it for the moment.

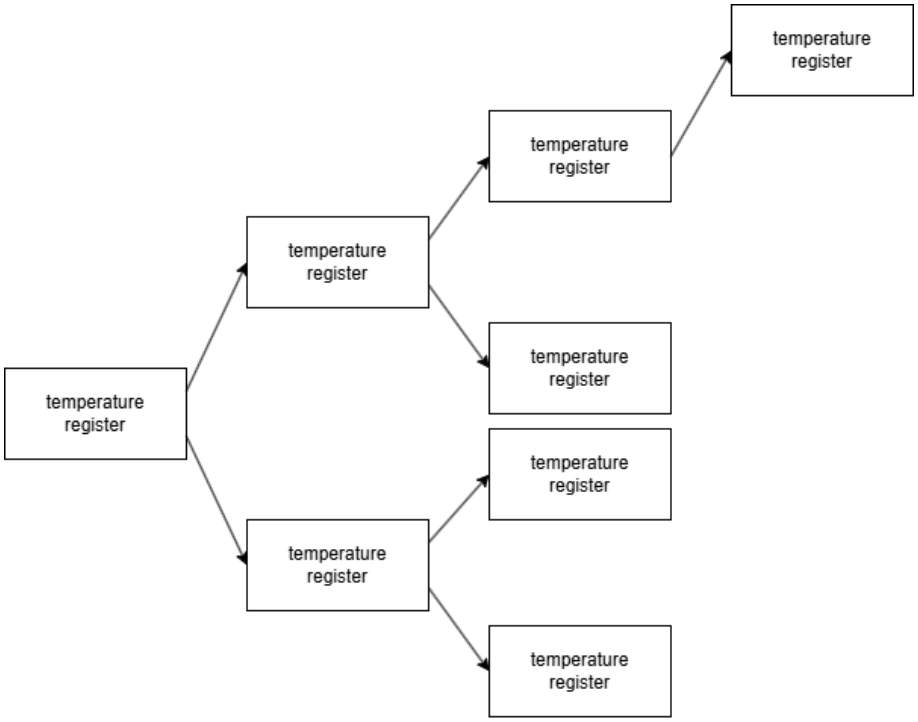


Figure 9. The diagram of the new design.

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.

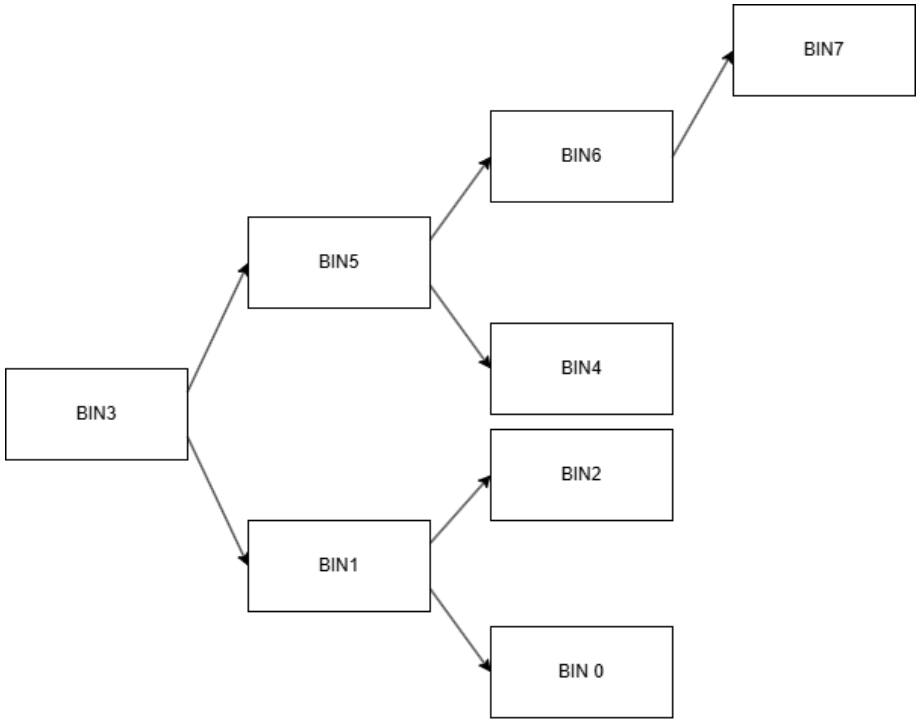


Figure 8. The distribution of bins.

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 $(-\infty; threshold\_value0]$ and $(threshold\_value6; +\infty)$. The rest of the bins simulate a binary search algorithm: the data from the sensor enters BIN3[22] and passes through a pipeline of at most $\log_2 NUMBER\_OF\_BINS$ (in the demo that is $\log_2 7 \cong 2.8$).

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:

i. Suppose the sensor measures 25ºC.
   a. *Greater* In the first clock cycle, data enters BIN3 (20ºC), BIN3 preserves its internal value and reroutes the input to BIN5 (24ºC).
   b. *Greater* In the second clock cycle BIN5 preserves its internal state and reroutes the input to BIN6(26ºC).
   c. *Suitable* In the third clock cycle BIN6 updates its internal state (+1) and does NOT reroute its data to its successors (if any).

ii. Suppose the sensor measures 16.5ºC.
   a. *Smaller* In the first clock cycle data enters BIN3 (20ºC), BIN3 preserves its internal value and reroutes the input to BIN1(16ºC).
   b. *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)

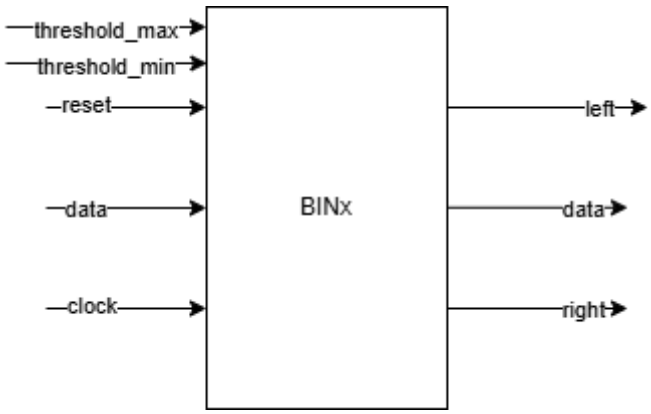


Figure 10. Block diagram of a temperature register component.

[22] BINx holds the x-th and (x+1)st threshold values.

4.3.2. 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.

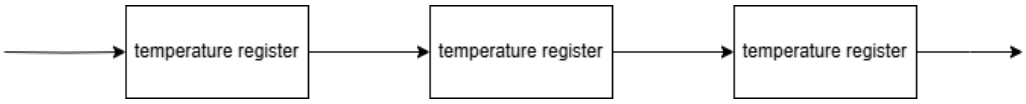


Figure 11. The block diagram of the queue.

In such a scenario the black box of the temperature register becomes:

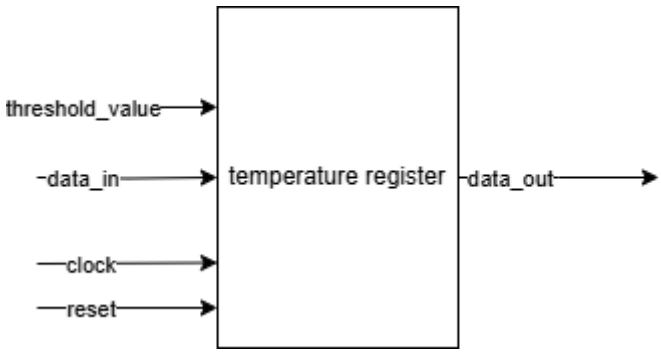


Figure 12. The black box of the temperature register

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.

4.4. 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.

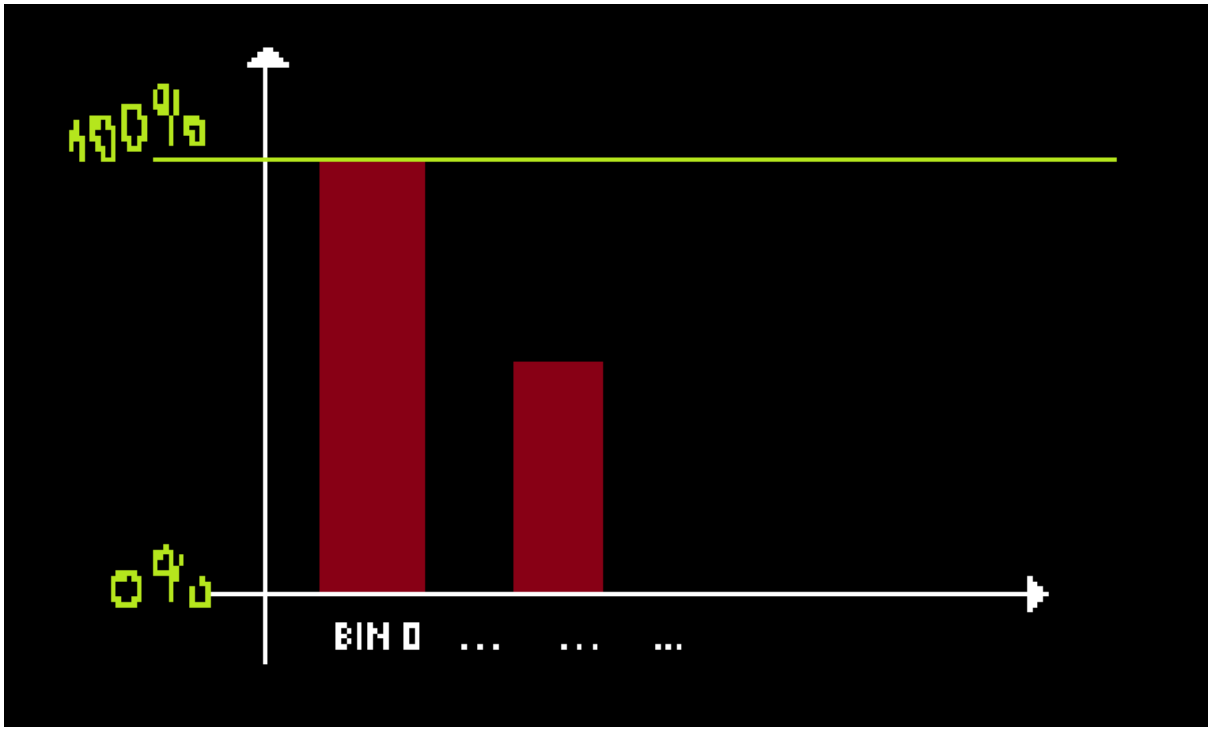


Figure 13. The sketch of the GUI.

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).

5. Implementation

5.1. PMOD HYGRO

As stated in the previous section, 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.

```
ENTITY i2c IS
GENERIC(
 INPUT_CLK : INTEGER;                    --input clock speed from user logic in Hz
 BUS_CLK   : INTEGER                     --speed the i2c bus (scl) will run at in Hz
);
PORT(
 clk          : IN    STD_LOGIC;         --system clock
 reset_i2c    : IN    STD_LOGIC;         --active low reset
 ena          : IN    STD_LOGIC;         --latch in COMMANDD
 addr         : IN    STD_LOGIC_VECTOR(6 DOWNTO 0);   --address of target device
 rw           : IN    STD_LOGIC;         --'0' is write, '1' is read
 write_data   : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);          --data to write to slave
 busy         : OUT   STD_LOGIC;         --indicates transaction in progress
 read_data    : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);          --data read from slave
 ack_error    : BUFFER STD_LOGIC;        --flag if improper acknowledge from slave
 sda          : INOUT  STD_LOGIC;        --serial data output of i2c bus
 scl          : INOUT  STD_LOGIC         --serial clock output of i2c bus
);
END ENTITY;
```

Figure 14. The entity of the I2C component.

```
CASE state IS
    WHEN READYY =>                      --idle state
     IF(ena = '1') THEN                 --transaction requested
      busy <= '1';                      --flag busy
      addr_rw <= addr & rw;             --collect requested slave address and
                                        --command
      data_tx <= write_data;            --collect requested data to write
      state <= STARTT;                  --go to STARTT bit
     ELSE                               --remain idle
      busy <= '0';                      --unflag busy
      state <= READYY;                  --remain idle
     END IF;
```

Figure 15. READYY state signal assignment

```
WHEN READD =>                                   --read byte of transaction
      busy <= '1';                              --resume busy if continuous mode
      IF(bit_cnt = 0) THEN                      --read byte receive finished
       IF(ena = '1' AND addr_rw = addr & rw) THEN --continuing with another read at
                                                 --same address
        sda_int <= '0';                         --acknowledge the byte has been
                                                 --received
       ELSE                                     --stopping or continuing with a
                                                 --write
        sda_int <= '1';                         --send a no-acknowledge (before
                                                 --STOPP or repeated STARTT)
       END IF;
       bit_cnt <= 7;                            --reset bit counter for "byte" states
       read_data <= data_rx;                    --output received data
       state <= M_ACKK;                         --go to master acknowledge
      ELSE                                      --next clock cycle of read state
       bit_cnt <= bit_cnt - 1;                  --keep track of transaction bits
       state <= READD;                          --continue reading
      END IF;
```

Figure 16. READD state signal assignments.[23]

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

```
ENTITY hygro IS
 GENERIC(
  SYS_CLK_FREQ        : INTEGER := 50_000_000;          --input clock speed from user
                                                        --logic in Hz
  HUMIDITY_RESOLUTION    : INTEGER RANGE 0 TO 14 := 14; --RH resolution in bits (must be
                                                        --14, 11, or 8)
  TEMPERATURE_RESOLUTION  : INTEGER RANGE 0 TO 14 := 14 --temperature resolution in bits
                                                        --(must be 14 or 11)
 );
 PORT(
  clk               : IN   STD_LOGIC;                 --system clock
  reset_pmod        : IN   STD_LOGIC;                 --asynchronous active-low reset
  scl               : INOUT STD_LOGIC;                --I2C serial clock
  sda               : INOUT STD_LOGIC;                --I2C serial data
  pmod_ack_err   : OUT   STD_LOGIC;                   --I2C target device acknowledge
                                                      --error flag
  relative_humidity : OUT   STD_LOGIC_VECTOR(HUMIDITY_RESOLUTION - 1 DOWNTO 0);
                                                      --relative humidity data obtained
  temperature : OUT   STD_LOGIC_VECTOR(TEMPERATURE_RESOLUTION - 1 DOWNTO 0)
                                                      --temperature data obtained
  );
END hygro;
```

Figure 17. The entity of the PMOD HYGRO controller.[24]

[23] Adapted from [3]
[24] Adapted from [3]

```
--determine the bits to set the temperature resolution in the sensor's configuration
--register
 WITH TEMPERATURE_RESOLUTION SELECT
  temp_res_bit <= '1' WHEN 11,
            '0' WHEN OTHERS;

--determine the number of clock cycles required for a temperature measurement at the
--given resolution
 WITH TEMPERATURE_RESOLUTION SELECT
  temp_time <= sys_clk_freq / 273 WHEN 11,          --3.65ms
          sys_clk_freq / 157 WHEN OTHERS;           --6.35ms
```

Figure 18. Multiplexing between resolution values based on the value of the generic parameter TEMPERATURE_RESOLUTION.[25]

```
WHEN INITIATEE =>
      busy_prev <= i2c_busy;                        --capture the value of the previous i2c
                                                    --busy signal
      IF(busy_prev = '0' AND i2c_busy = '1') THEN   --i2c busy just went high
       busy_cnt := busy_cnt + 1;                    --counts the times busy has gone from
                                                    --low to high during transaction
      END IF;
      CASE busy_cnt IS                              --busy_cnt keeps track of which
                                                    --command we are on
       WHEN 0 =>                                    --no command latched in yet
        i2c_enable <= '1';                          --initiate the transaction
        i2c_addr <= hygrometer_addr;                --set the address of the hygrometer
        i2c_rw <= '0';                              --command 1 is a write
        i2c_write_data <= "00000000";               --set the register pointer to the
                                                    --Temperature Register
       WHEN 1 =>                                    --1st busy high: command 1 latched
        i2c_enable <= '0';                          --deassert enable to stop transaction
                                                    --after command 1
        IF(i2c_busy = '0') THEN                     --transaction complete
         busy_cnt := 0;                             --reset busy_cnt for next transaction
         state <= PAUSEE;                           --advance to the PAUSEE state
        END IF;
       WHEN OTHERS => NULL;
      END CASE;
```

Figure 19. INITIATEE state signal assertion.[26]

[25] Adapted from [3]
[26] Adapted from [3]

5.2. Processing the sensor data

5.2.1. 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, 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.

```
ENTITY temperature_register IS
  PORT(
    reg_clk             : IN STD_LOGIC;
    reg_r               : IN STD_LOGIC;
    reg_add_one         : IN STD_LOGIC;
    reg_value           : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END temperature_register;

ARCHITECTURE arh OF temperature_register IS

SIGNAL internal_state: STD_LOGIC_VECTOR(15 DOWNTO 0) := x"0000";

BEGIN

  counter: PROCESS(reg_clk, reg_r)                    --counter with enable
  BEGIN
    IF reg_r = '0' THEN
      internal_state <= x"0000";
    ELSIF rising_edge(reg_clk) THEN
      IF reg_add_one = '1' THEN
        internal_state <= internal_state + 1;
      END IF;
    END IF;
  END PROCESS;
  reg_value <= internal_state;
END arh;
```

Figure 20. 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.

```
ENTITY ssd_display IS
  PORT(
    clk          : IN STD_LOGIC;
    number_in    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    anodes       : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
    cathodes     : OUT STD_LOGIC_VECTOR (0 TO 6)
  );
END ssd_display;

--start of the architecture
TYPE constant_array IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR (0 TO 6);
constant digits: constant_array := (
  0 => "1000000", -- 0
  1 => "1111001", -- 1
  2 => "0100100", -- 2
  3 => "0110000", -- 3
  4 => "0011001", -- 4
  5 => "0010010", -- 5
  6 => "0000010", -- 6
  7 => "1111000", -- 7
  8 => "0000000", -- 8
  9 => "0010000", -- 9
  10 => "0001000", -- A
  11 => "0000011", -- B
  12 => "1000110", -- C
  13 => "0100001", -- D
  14 => "0000110", -- E
  15 => "0001110"  -- F
);
```

Figure 21. The entity declaration and the seven-segment digit code look-up table.

```
--inside the architecture
SIGNAL lighting_anodes :  STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";

BEGIN
  freq_divider: PROCESS(clk)
    VARIABLE aux: STD_LOGIC_VECTOR (19 DOWNTO 0) := x"00000";
  BEGIN
    IF rising_edge(clk) THEN
      aux := aux + 1;
    END IF;
    lighting_anodes <= aux(19 DOWNTO 18);
  END PROCESS;

  WITH lighting_anodes SELECT anodes <=
    "1110" when "00",
    "1101" when "01",
    "1011" when "10",
    "0111" WHEN OTHERS;

  WITH lighting_anodes SELECT cathodes <=
    digits(TO_INTEGER(UNSIGNED(number_in(3 DOWNTO 0)))) WHEN "00",
    digits(TO_INTEGER(UNSIGNED(number_in(7 DOWNTO 4)))) WHEN "01",
    digits(TO_INTEGER(UNSIGNED(number_in(11 DOWNTO 8)))) WHEN "10",
    digits(TO_INTEGER(UNSIGNED(number_in(15 DOWNTO 12))))  WHEN OTHERS;
--end of architecture
```

Figure 22. 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*)st threshold and the x-th threshold) 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.

```
ENTITY temperature_register_controller IS
  GENERIC(
    RES                 : INTEGER := 14;
    NO_BINS             : INTEGER := 8
  );
  PORT(
    clockk              : IN STD_LOGIC;
    rs                  : IN STD_LOGIC;
    raw_data            : IN STD_LOGIC_VECTOR (RES – 1 DOWNTO 0);
    select_bin          : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
    as                  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
    cs                  : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
  );
END temperature_register_controller;
```

Figure 23. The entity of the register controller component.

```
SIGNAL increment: STD_LOGIC_VECTOR(0 TO NO_BINS - 1) := (OTHERS => '0');

TYPE matrix IS ARRAY (0 TO NO_BINS - 1) OF STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL m : matrix:= (OTHERS=> x"0000");
SIGNAL ref_value1 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"53C9";      --14 deg C
SIGNAL ref_value2 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"56E3";      --16 deg C
SIGNAL ref_value3 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"59FD";      --18 deg C
SIGNAL ref_value4 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"5D18";      --20 deg C
SIGNAL ref_value5 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"6032";      --22 deg C
SIGNAL ref_value6 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"634D";      --24 deg C
SIGNAL ref_value7 : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"6667";      --26 deg C
```

Figure 24. 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]. This moves us back to the Redesigning the temperature controller & temperature register components.

[27] [14]

5.2.2. Second draft

The source code of the new implementation is presented below.

```
ENTITY temperature_register IS
  PORT(
    reg_clk          : IN STD_LOGIC;
    reg_r            : IN STD_LOGIC;
    measured_data    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    threshold_min    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    threshold_max    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    reg_value        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    data_out_left    : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    data_out_right   : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
  );
END temperature_register;
```

Figure 25. Entity declaration.

```
  counter: PROCESS(clk, reg_r)
  BEGIN
    IF reg_r = '0' THEN
      internal_state <= x"0000";
    ELSIF rising_edge(clk) THEN
      IF (unsigned(threshold_min) < unsigned(measured_data) AND
        unsigned(measured_data) <= unsigned(threshold_max)) THEN
        internal_state <= internal_state + 1;
        data_out_left <= x"0000";
        data_out_right <= x"0000";
      ELSIF(unsigned(threshold_min) > unsigned(measured_data)) THEN
        data_out_left <= measured_data;
        data_out_right <= x"0000";
      ELSIF(unsigned(threshold_min) < unsigned(measured_data)) THEN
        data_out_right <= measured_data;
        data_out_left <= x"0000";
      ELSE
        data_out_right <= x"0000";
        data_out_left <= x"0000";
      END IF;
    END IF;
  END PROCESS;
  reg_value <= internal_state;
```

Figure 26. Updated counter process of the temperature register

5.2.3. Third draft

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

```
ENTITY temperature_register IS
	PORT(
		clk		: IN STD_LOGIC;
		reset		: IN STD_LOGIC;
		data_in		: IN STDLOGIC_VECTOR(15 DOWNTO 0);
		threshold	: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
		data_out	: OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
	);
END temperature_register
```

Figure 27. The entity of the FIFO-type register.

```
PROCESS(clk, reset)
BEGIN
	IF reset = '0' THEN
		data_out <= x"0000";
	ELSIF rising_edge(clk) THEN
		IF (data_in /= x"0000" AND data_in < threshold) THEN
			internal_state <= internal_state + 1;
			data_out <= x"0000";
		ELSE
			data_out <= data_in;
		END IF;
	END IF;
END PROCESS;
```

Figure 28. The core of the temperature register component.

5.3. 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/.

5.4. 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.

```
--ARCHITECURE arh OF top IS
--component declaration
BEGIN  --of architecture
  temp            <= t;
  n_reset         <= not system_reset;

  send_data_to : uart_top
  GENERIC MAP(
    RESOLUTION          => MY_TEMPERATURE_RESOLUTION)
  PORT MAP(
    clk                 => system_clock,
    reset               => n_reset,
    temperature          => t,
    rx                  => r_antenna,
    tx                  => t_antenna);

  hygro_connection : hygro
  GENERIC MAP(
    SYS_CLK_FREQ                => MY_SYSTEM_CLK_FREQ,
    HUMIDITY_RESOLUTION         => MY_HUMIDITY_RESOLUTION,
    TEMPERATURE_RESOLUTION => MY_TEMPERATURE_RESOLUTION)
  PORT MAP(
    clk                 => system_clock,
    reset_pmod          => system_reset,
    scl                 => hygro_scl,
    sda                 => hygro_sda,
    pmod_ack_err        => hygro_ack,
    relative_humidity   => h,
    temperature         => t);

  histogram_binning: temperature_register_controller
  GENERIC MAP(
    RES                 => MY_TEMPERATURE_RESOLUTION,
    NO_BINS             => 8
  )
  PORT MAP(
    clockk              => system_clock,
    rs                  => system_reset,
    raw_data            => t,
    select_bin          => debug_bins_selector,
    as                  => debug_anodes,
    cs                  => debug_cathodes
  );

END ARCHITECTURE;
```

Figure 29. Snipped from the architecture of the top-level module.

5.5. 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*).

6. Testing & Validation

6.1. Simulations & testbenches

Some of the components are not suitable for testbench testing (i.e. the PMOD HYGRO) because it requires too much work. How I decide if they work correctly was a matter of implementation (in VHDL), synthesizing, implementing & uploading (all done by Vivado) on the Basys3 development board. As I have already stated in the previous chapters, I used a bottom-up technique, meaning that I started small and made sure everything I did in one go works as intended on the board. Consequently, I did not face any unexpected issues when the bitstream was uploaded on the board (as I may have encountered if I relied only on testbenches, and the simulation tools provided by Vivado).

There are, however, modules that required some final simulation touches because things looked off at times. Implementing algorithms in hardware is not the most straightforward, so the histogram binning process was the first one to require special attention. Being a versatile component of the system and the subject of about three changes in its design (and, consequently, its implementation), a testbench[28] was a simple way to ensure that the module preserves its interface (i.e. input and output ports), but most importantly, its behavior.

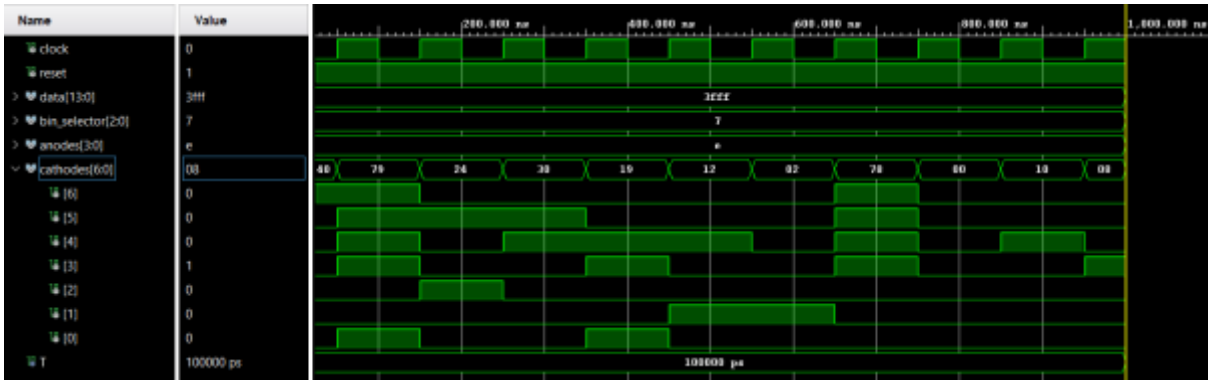


Figure 30. A testbench simulation record.

In the record above, the system clock is set at a frequency of 100 Hz, and the simulated registered temperature is *data* – 0xFFFF (14 bits resolution and 2 hardcoded zeroes at the end, as the manufacturer states, yield the searched value). The bin to be displayed is the seventh one (*bin_selector* = 0x7), the bin that groups values between 26ºC up to +∞. 0xFFFF converted with the formula provided in this section equals ≈ 125ºC. The seventh bin (register) being linked to the system clock for the simulation means that at each rising edge its internal state is incremented. This can be seen on the cathodes signal (that goes, in the real design, to the seven segment displays). The interpretation of the first 5 values (0x40, 0x79, 0x24, 0x30, 0x19) are shown in the image to the right. We see that the state of the register correctly updates with the rising edge of the system clock.

[28] The unit under test for the *binning_tb.vhd* testbench is not a single component, but a "specialized top-level" module that depends on how often the sensor reads data (about once every 6.35ms). This implies that a frequency divider must be included in the *temperature_register_controller.vhd* so that the same value is not counted twice (and thus, skewing or biasing the result). For simulating the behavior of the module, this frequency divider is redundant, so I have disabled it.

In the following simulation we decreased the clock frequency (to 10ns) so that more signal changes could fit.

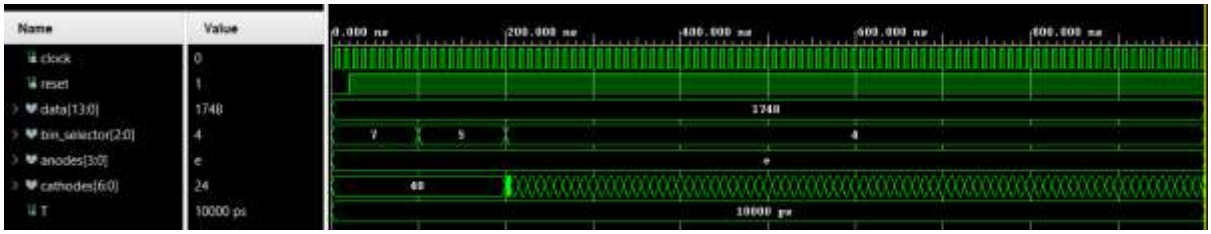


Figure 32. Perspective view of a testbench simulation record.

In the above diagram we observe that the data the sensor measures is constant (0x1748, converted to 0x5D20) and the displayed contents are changing. Let's begin by classifying the temperature into the correct bin: 0x5D20 is equivalent to ≈ 20.02ºC meaning that its place is in the fourth bin. The first part of the simulation above (up to 200ns) shows the contents of the seventh and fifth bins correctly preserving the state 0 (encoded as 0x40 on seven segment displays).

Figure 31. In-depth view of the previous simulation record.

A closer inspection of the interval 196ns to 310ns yields the above picture. We see that when the *bin_selector* changes to its correct value (i.e. 0x4) we start to see incrementing values. Because there is only one seven-segment-display active (the last one – 0xE), only the unit digit is visible. We note that it is visible that the internal state of the fourth bin has been changing even if the bin was not queried for display (as expected, the unit digit is non-zero). The decoding of the first six values in the fourth register last digit is shown in the image on the right.[29]

6.2. Real time testing

7. Conclusions & Further development

Working on this project was more demanding than I expected at first sight. I was ready to invest a lot of time in the bibliographical research and implementation process, but I found myself lost in too many complicated concepts (i.e. Microblaze, Vivado's intellectual properties repository, Vivado's way of interfacing components etc.). These are all interesting topics but

[29] We can compute the first digit based on how many rising clocks have happened until the 200ns mark: the last rising clock is placed 5 ns before the mark, at 195ns, each 10ns starting there down to 5ns a rising edge has happened. Two rising edges are invalidated by the reset signal being LOW (active), meaning there are (195-25) / 10 + 1 = 18 rising edges up to the 195ns mark, yielding the 0x12 internal state in the fourth register. Consequently, before the 19th (excluding the first 2 reset-disabled) rising clock signal happening at the 205ns mark, the fourth register displays the state 0x12. Thus, the first digit (the decimals) of the register value is 1, and the units update accordingly with the upcoming clock cycles.

trying to understand their ups and downs and how to use them became tiring quickly. The lack of good sources and documentation was another obstacle I had to overcome.

All in all, the project's initial, rough idea was to create a remote (wireless) digital thermometer. The project has been built, but the wireless part had to be excluded because it implied additional overhead that could not accommodate inside the imposed time frame. Consequently, the PMOD ESP32 has been replaced mid-development with a UART connection. This may seem like a small disaster, but thanks to the modular architecture paradigms I tried to follow the best as I could, the current design must not be rebuilt from scratch when the wireless capabilities shall be added. The graphical user interface can be extended with a new "connection handler", that should replace the ComPort handler with almost no effort if the interface is preserved. No modifications should be required in other parts of the GUI. The same reasoning applies to the hardware implementation (some twitches may be needed, but I am not qualified enough at this time to understand or explain them).

The positive aspects of working on the project are more important. I have developed my VHDL skills, I have grown more aware of the relation between complexity, time frames and deadlines, I discovered new topics: soft-core processors, intellectual properties, communication protocols, block designs, Windows API calls, handling serial communication in C++ and so on.

The final project does not accomplish all the proposed features presented in the Project Proposal section, but I am proud of its final form and plan to develop it in the future.

Promised feature	Final status
Temperature reading over a temperature sensor.	DONE
Temperature classification into "buckets".	DONE[30]
Temperature broadcasting over Wi-Fi.	NOT DONE[31]
Temperature histogram visualization on a display.	DONE[32]
One or more test cases that prove the correctness of the design.	DONE[33]
The source code.	DONE[34]

Table 11. The status of the features at the end of the current version of the project.

Further developments:

- Adding the PMOD ESP32 in the design to broadcast the measured temperature.
- Updating the system design to detect hardware errors or faulty input using algorithms like CUSUM, Z-score etc.
- Updating the GUI to handle different inputs (Wireless and wired), and display them one next to the other for comparison, validation etc.

[30] Multiple designs and implementations have been tested, but only the best one has been preserved in the final design.
[31] In the final design, the Wi-Fi broadcasting has been replaced by a UART connection over a wire to the device that wants to see the histogram (has GUI source code, and the required dependencies installed: C/C++ compiler, raylib.h, make, etc.).
[32] The C++ implementation of the GUI.
[33] Presented in the Testing & Validation section.
[34] Available at my github profile.

- Updating the hardware to handle more sensors and sent the data they read to the GUI.

8. 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, "I^{2}C," [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.

[Digilent, "Digilent Pmod™ Interface Specification 1.2.0," 5 October 2017. [Online].
9 Available: https://digilent.com/reference/_media/reference/pmod/pmod-interface-
] specification-1_2_0.pdf.

[Espressif, "ESP32 Seris Datasheet," September 2024. [Online]. Available:
1 https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.p
0 df.
]

[Digilent, "Pmod ESP32 Reference Manual," [Online]. Available:
1 https://digilent.com/reference/pmod/pmodesp32/reference-manual. [Accessed 17
1 October 2024].
]

[espressif.com, "ESP-IDF Programming Guide," [Online]. Available:
1 https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.
2 [Accessed 17 October 2024].
]

[R. Danescu, "Design with Microprocessors – 2023-2024," 2024. [Online]. Available:
1 https://users.utcluj.ro/~rdanescu/teaching_pmp.html.
3
]

[R. Withu, "Tournament Tree (Winner Tree) and Binary Heap," geeksforgeeks, 2 June
1 2023. [Online]. Available: https://www.geeksforgeeks.org/tournament-tree-and-
4 binary-heap/. [Accessed 12 November 2024].
]

["raylib," [Online]. Available: https://www.raylib.com/. [Accessed 25 11 2024].
1
5
]