Processing signals received from sensors

Student: Laurentiu-Calin Grad

2025

Technical University of Cluj-Napoca

Structure of Computer Systems Project

Contents

[**1.** **Introduction** 1](#_Toc179735488)

[1.1. Context 1](#_Toc179735489)

[1.2. Objectives 1](#_Toc179735490)

[**2.** **Bibliographic Research** 2](#_Toc179735491)

[2.1. PMOD HYGRO 2](#_Toc179735492)

[2.1.1. The I²C interface (protocol) 2](#_Toc179735493)

[2.1.2. Interfacing with the Pmod 3](#_Toc179735494)

[2.2. PMOD ESP32 5](#_Toc179735495)

[2.2.1. AT command set 6](#_Toc179735496)

[2.2.2. Interfacing with the Pmod 7](#_Toc179735497)

[2.2.3. The UART interface (protocol) 8](#_Toc179735498)

[**3.** **Analysis** 9](#_Toc179735499)

[3.1. Project Proposal 9](#_Toc179735500)

[3.2. Project Analysis 9](#_Toc179735501)

[**4.** **Design** 9](#_Toc179735502)

[**5.** **Implementation** 9](#_Toc179735503)

[**6.** **Testing & Validation** 9](#_Toc179735504)

[**7.** **Conclusions** 9](#_Toc179735505)

[**8.** **Bibliography** 9](#_Toc179735506)

[PMOD HYGRO 9](#_Toc179735507)

[PMOD ESP32 9](#_Toc179735508)

[I²C protocol 9](#_Toc179735509)

[AXI Protocol 9](#_Toc179735510)

[UART Protocol 10](#_Toc179735511)

[Miscellaneous 10](#_Toc179735512)

# **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.

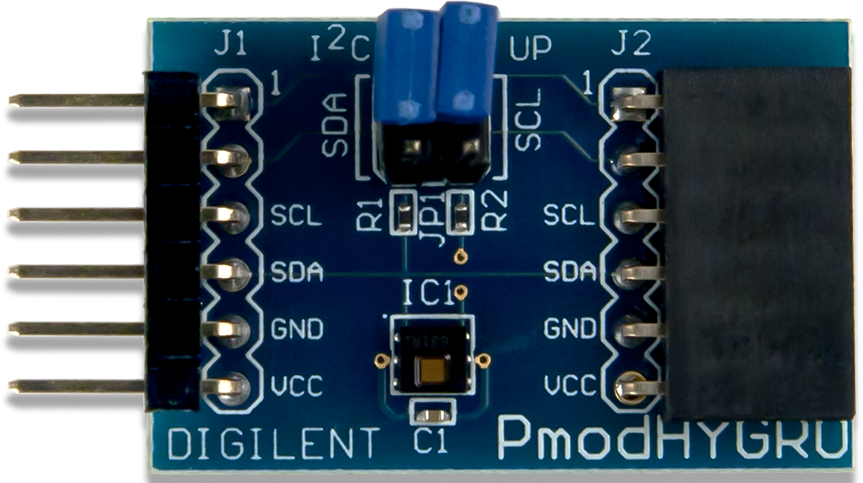
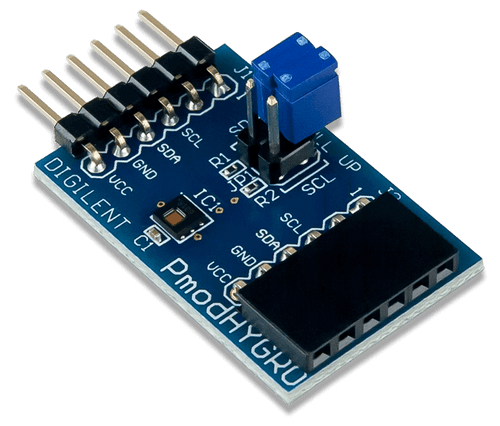
Figure1. The conceptual diagram of the project.

# **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 |

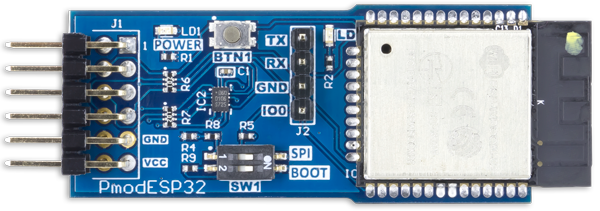
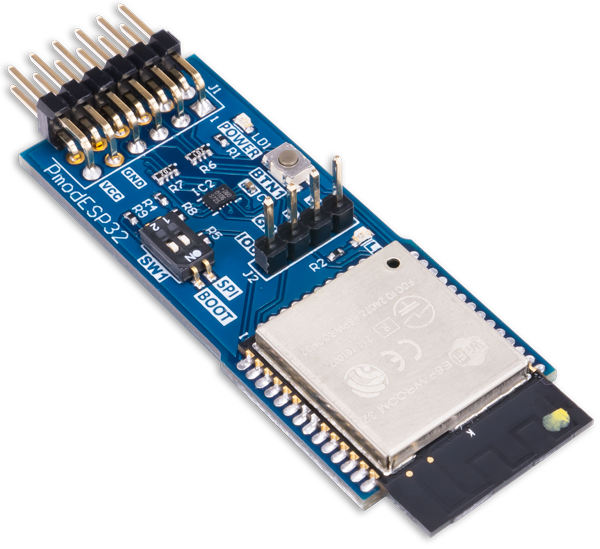
(\* Default value on power-up or reset, \*\* Time required to compute a conversion at that specific resolution)

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.

Conclusions:

* PMOD HYGRO requirements:
  + An I2C state machine to fetch the data it sends over the serial port.
  + An arithmetic-logic component that converts the binary data into valid temperatures as the provider’s formula states.
  + 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.
* PMOD ESP32 requirements:
  + In progress…
* Other requirements:
  + In progress…

# **Design**

## PMOD HYGRO

### The I2C interface

The I2C interface follows a public design found on the internet.[[12]](#footnote-12) 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.[[13]](#footnote-13)

A diagram of a computer

Description automatically generated

Figure . State transition diagram.

# **Implementation**

# **Testing & Validation**

# **Conclusions**

# **Bibliography**

### PMOD HYGRO

* [2243501.pdf (farnell.com)](https://www.farnell.com/datasheets/2243501.pdf?_gl=1*63u4b2*_gcl_aw*R0NMLjE3Mjc4ODk0NDAuQ2owS0NRanczdk8zQmhDcUFSSXNBRVdibGNETnBoOU5faFVJT05mZk9RLVEyVUhYdDE1RjlXWnZGYUpVbHV0bEZJN3NkYlo1emtHOENUSWFBdVFuRUFMd193Y0I.*_gcl_au*NzM3NTE2MzMuMTcyNzg3MDcxMg..)
* [HDC1080 Low Power, High Accuracy Digital Humidity Sensor with Temperature Sensor datasheet (Rev. A) (ti.com)](https://www.ti.com/lit/ds/symlink/hdc1080.pdf)
* https://forum.digikey.com/t/humidity-and-temperature-sensor-pmod-controller-vhdl/13064?\_ga=2.24350906.1099841098.1728495616-27067341.1728495616

### PMOD ESP32

* [pmod-interface-specification-1\_2\_0.pdf (digilent.com)](https://digilent.com/reference/_media/reference/pmod/pmod-interface-specification-1_2_0.pdf)
* [esp32\_datasheet\_en.pdf (espressif.com)](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)
* [Pmod ESP32 Reference Manual - Digilent Reference](https://digilent.com/reference/pmod/pmodesp32/reference-manual)
* [Get Started - ESP32 - — ESP-IDF Programming Guide latest documentation (espressif.com)](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html)
* [vscode-esp-idf-extension/docs/tutorial/basic\_use.md at master · espressif/vscode-esp-idf-extension (github.com)](https://github.com/espressif/vscode-esp-idf-extension/blob/master/docs/tutorial/basic_use.md)
* [vscode-esp-idf-extension/docs/tutorial/install.md at master · espressif/vscode-esp-idf-extension (github.com)](https://github.com/espressif/vscode-esp-idf-extension/blob/master/docs/tutorial/install.md)
* [Proiectare cu Microprocesoare (utcluj.ro)](https://users.utcluj.ro/~rdanescu/teaching_pmp.html)

### I²C protocol

* [Basics of the I2C Communication Protocol (circuitbasics.com)](https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/)
* [I²C - Wikipedia](https://en.wikipedia.org/wiki/I%C2%B2C)
* https://forum.digikey.com/t/i2c-master-vhdl/12797

### AXI Protocol

* [Advanced eXtensible Interface - Wikipedia](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface)
* [AMBA AXI4 Interface Protocol (xilinx.com)](https://www.xilinx.com/products/intellectual-property/axi.html)
* [bing.com/ck/a?!&&p=36a560511f23f71fJmltdHM9MTcyODM0NTYwMCZpZ3VpZD0wNzBjMDc1My0xMDhiLTYwYzgtMjZlMy0xMzlmMTE3MTYxZDQmaW5zaWQ9NTE5Mw&ptn=3&ver=2&hsh=3&fclid=070c0753-108b-60c8-26e3-139f117161d4&psq=amba+axi-stream+protocol+specification&u=a1aHR0cHM6Ly9kb2N1bWVudGF0aW9uLXNlcnZpY2UuYXJtLmNvbS9zdGF0aWMvNjQ4MTlmMTUxNmYwZjIwMWFhNmI5NjNj&ntb=1](https://www.bing.com/search?pglt=673&q=amba+axi-stream+protocol+specification&cvid=8ca7a00867fa4fe48039b8fc6f2a7ec3&gs_lcrp=EgZjaHJvbWUqBggBEAAYQDIGCAAQRRg5MgYIARAAGEAyBggCEAAYQDIGCAMQABhAMgYIBBAAGEAyBggFEAAYQDIGCAYQABhAMgYIBxAAGEAyBggIEEUYPNIBCDQyMjNqMGoxqAIAsAIA&FORM=ANNTA1&PC=U531)

### UART

* https://en.wikipedia.org/wiki/Universal\_asynchronous\_receiver-transmitter

### Miscellaneous

* [Difference between master and slave devices in communication network - Electrical Engineering Stack Exchange](https://electronics.stackexchange.com/questions/554170/difference-between-master-and-slave-devices-in-communication-network)
* <https://insights.sigasi.com/tech/vhdl-resets/>
* <https://nandland.com/uart-serial-port-module/>

cusum, z score

histograma hardware input output analiza

1. To facilitate data interpretation a visual representation (e.g. histogram) should be provided to the users. [↑](#footnote-ref-1)
2. [I²C - Wikipedia](https://en.wikipedia.org/wiki/I%C2%B2C) [↑](#footnote-ref-2)
3. [HYGRO reference manual](https://www.farnell.com/datasheets/2243501.pdf?_gl=1*63u4b2*_gcl_aw*R0NMLjE3Mjc4ODk0NDAuQ2owS0NRanczdk8zQmhDcUFSSXNBRVdibGNETnBoOU5faFVJT05mZk9RLVEyVUhYdDE1RjlXWnZGYUpVbHV0bEZJN3NkYlo1emtHOENUSWFBdVFuRUFMd193Y0I.*_gcl_au*NzM3NTE2MzMuMTcyNzg3MDcxMg..) [↑](#footnote-ref-3)
4. [Pmod ESP32 Reference Manual - Digilent Reference](https://digilent.com/reference/pmod/pmodesp32/reference-manual) [↑](#footnote-ref-4)
5. [AT Command Set - ESP32 - — ESP-AT User Guide latest documentation (espressif.com)](https://docs.espressif.com/projects/esp-at/en/latest/esp32/AT_Command_Set/index.html#at-command-types) [↑](#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. https://forum.digikey.com/t/i2c-master-vhdl/12797 [↑](#footnote-ref-12)
13. https://forum.digikey.com/t/i2c-master-vhdl/12797 [↑](#footnote-ref-13)