IS4310-232M4

RS232 Modbus RTU Slave Module

Presentation

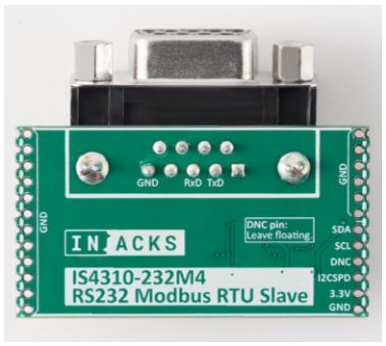
The IS4310-232M4 is a ready-to-operate module integrating the Modbus RTU Slave stack chip IS4310 with an RS232 transceiver. This solution reduces to the minimum expression the design effort of integrating a Modbus RTU Slave with an RS232 electrical interface.

The module can be directly soldered to your PCB with its castellated holes, or you can solder a pin header and use it as a module.

Module Characteristics

Electrical Characteristics	
Module Voltage	3.3V
I2C Compatible Voltage Levels	3.3V, 5V

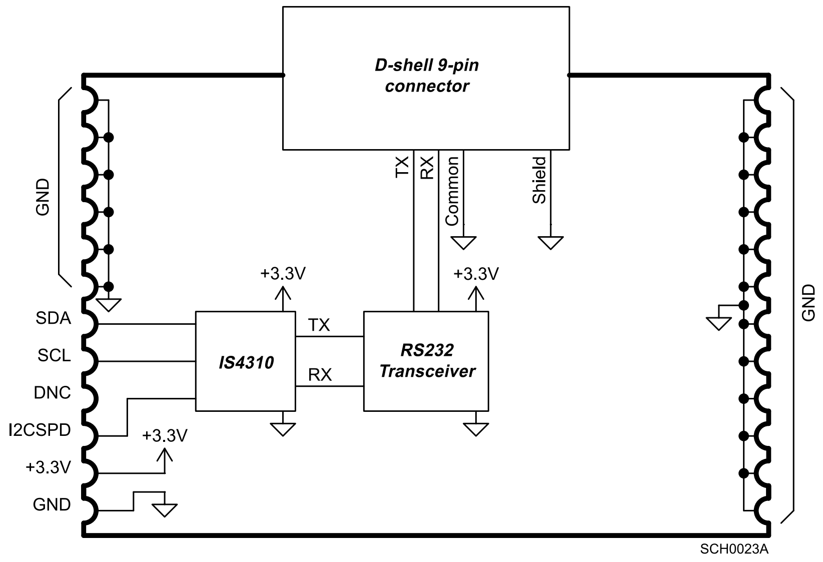
Modbus Characteristics	
Supported Function Codes:	3 (0x03) - Read Holding Registers 6 (0x06) - Write Single Register 16 (0x10) - Write Multiple Registers
Holding Registers:	500
Operating Mode:	RTU
Electrical Interface:	RS232



Product Selection Guide

Category	Part Number		Form Factor	Physical Layer	Stack	Description
Only Stack	IS4310-S8		SO8N	UART 3.3V	Modbus RTU Server	Modbus RTU Slave Stack Chip. View Product
Stack with Physical Layer	IS4310-485M2		Castellated Holes Module	RS485	Modbus RTU Server	IS4310 with RS485 Transceiver. Industrial communications. View Product
Stack with Physical Layer	IS4310-ISO485M6		Castellated Holes Module	Isolated RS485	Modbus RTU Server	IS4310 with Isolated RS485 Transceiver. The isolation offers more robust communications and longer RS485 bus distances. View Product
Stack with Physical Layer	IS4310-232M4		Castellated Holes Module	RS232	Modbus RTU Server	IS4310 with RS232 Transceiver. View Product
Evaluation Boards	Kappa4310Ard		Arduino Compatible	RS485	Modbus RTU Server	IS4310 Evaluation Board with RS485 Transceiver. Compatible with Arduino. View Product
Evaluation Boards	Kappa4310Rasp		Raspberry Pi Compatible	RS485	Modbus RTU Server	IS4310 Evaluation Board with RS485 Transceiver. Compatible with Raspberry Pi. View Product

1. Description



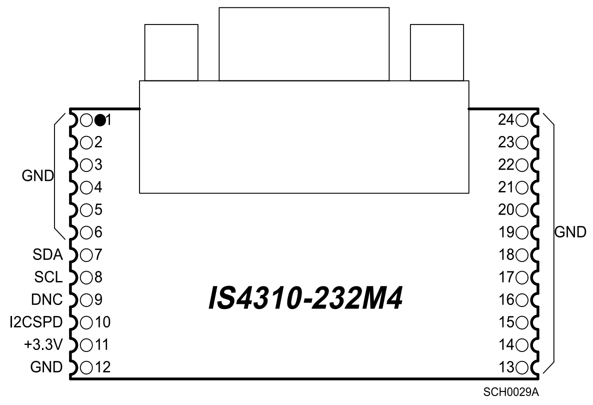
The IS4310-232M4 is a compact (44 × 24 mm) module with castellated holes, designed for PCB mounting to function as an RS232 Modbus RTU Slave. It features two key components: the Modbus RTU Slave Stack (IS4310) and an RS232 transceiver.

The module features a female D-shell 9 pin connector, the popular and wrongly known as DB9. The Shield pin connected to GND. Ensure that you use shielded cables.

The module operates at a fixed voltage of 3.3V. The I2C pads (SDA and SCL) are open-drain and compatible with both 3.3V and 5V. The DNC pad must be left unconnected. The I2CSPD pin is used for I2C speed selection. The module includes multiple GND pads, and proper soldering of all pads is essential for mechanical stability and durability when attaching the module to the main PCB.

The module also features a green LED to indicate power status, a yellow LED to signal data transmission and another to signal reception. A cable disconnected LED is also present.

1.1. Module Pinout



Pad	Name	Type	Description
7	SDA	Open Drain 3.3V (5V Tolerant)	SDA pin of the IS4310: Open drain, it requires pull-up.
8	SCL	Open Drain 3.3V (5V Tolerant)	SCL pin of the IS4310: Open drain, it requires pull-up.
9	DNC	Do Not Connect	This pad must be left floating.
10	I2CSPD	Analog Input 0 to 3.3V	I2CSPD pin of the IS4310: I2C-Serial Interface Speed Selection. • For 100kHz pull to GND. • For 400kHz make a voltage divider of +3.3V/2 (1.65V). • For 1MHz pull to 3.3V. Attention: Voltage in this pin above 4V will damage the IS4310.
11	+3.3V	Module Power (Power In)	Power supply for the module.
1-6, 12, 13-24	GND	Ground	Ground reference pad. GND pads are connected to the “Common” of the RS232 bus. GND pads are connected to the shield of the connector RS232 bus.

SCL and SDA Pads

I2C-Compatible Bus Interface Pads.

Both pads are open-drain and must be pulled up to 3.3V or 5V. The pull-up resistor value should be chosen based on the bus speed and capacitance. Typical values are 4.7kΩ for Standard Mode (100kbps) and 2.2kΩ for Fast Mode (400kbps) at both 3.3V and 5V.

+3.3V Pad

Module Power Supply Pad.

This pad is the power input for the entire module. 3.3V must be supplied to this pad. Bypass capacitors are included on the module, no need to place them outside of the module.

GND Pads

Module Ground.

These pads are also connected to the Common signal of the RS232 bus.

GND pads are connected to the shield of the DB9 connector.

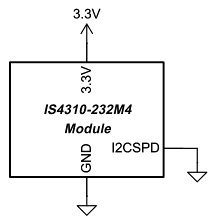
The module has multiple GND pads, all of which must be soldered to ensure proper mechanical attachment of the module to the main PCB. This is especially important when the RJ45 connectors are plugged in, as a significant amount of force is applied to the module. Failing to solder all pads or poor soldering can cause the module to detach from the main PCB.

I2CSPD Pad

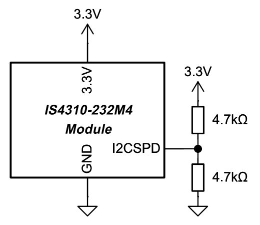
I2C-Serial Interface Speed Selection Pad.

This pad is directly connected to the pin I2CSPD of the IS4310. It configures the IS4310 internal I2C-Serial Interface timings and filters to properly work with the selected bus speed.

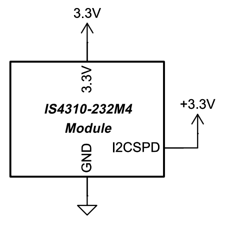
- For a **100kHz** setting, set the I2CSPD pad to GND.



- For a **400kHz** setting, set the I2CSPD to 1.65V (+3.3V/2) using a balanced voltage divider. This can be achieved by placing two 4.7kΩ resistors from the I2CSPD pin: one to VDD and the other to VSS.



- For a **1000MHz** setting, set the I2CSPD pad to 3.3V. Please note that pulling I2CSPD to 5V would permanently damage the IS4310 IC.



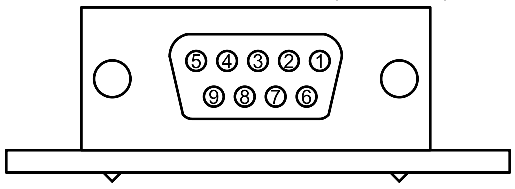
Important Remark:
A mismatch between the configured I2C speed and the actual operating I2C speed (e.g., configuring the bus for 100kHz but operating at 1MHz) can lead to an inconsistent state where some I2C messages are processed while others are not.

Ensure a proper match between the actual operating speed and the configured speed at the I2CSPD pad: If your bus works at 100kHz, ensure the I2CSPD pad is tied to VSS. If it works at 400kHz ensure the pad is at 1.65V. If it works at 1000MHz, ensure the pad is at 3.3V.

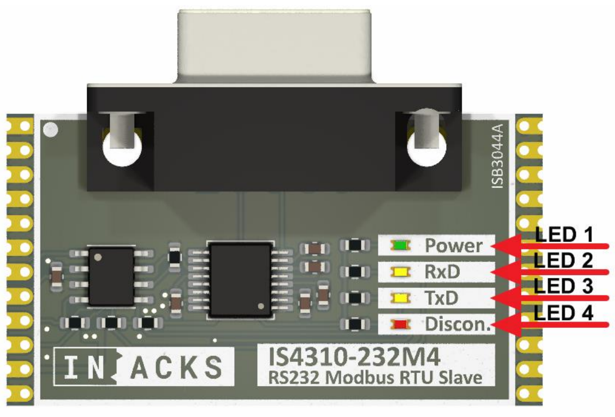
1.2. DB9 Connector

Typical Modbus Serial Line connectors include Screw Terminals, RJ45, and D-Sub 9-pin (commonly known as DB9), among others. The device-side connector must be female, while the cable-side connector must be male.

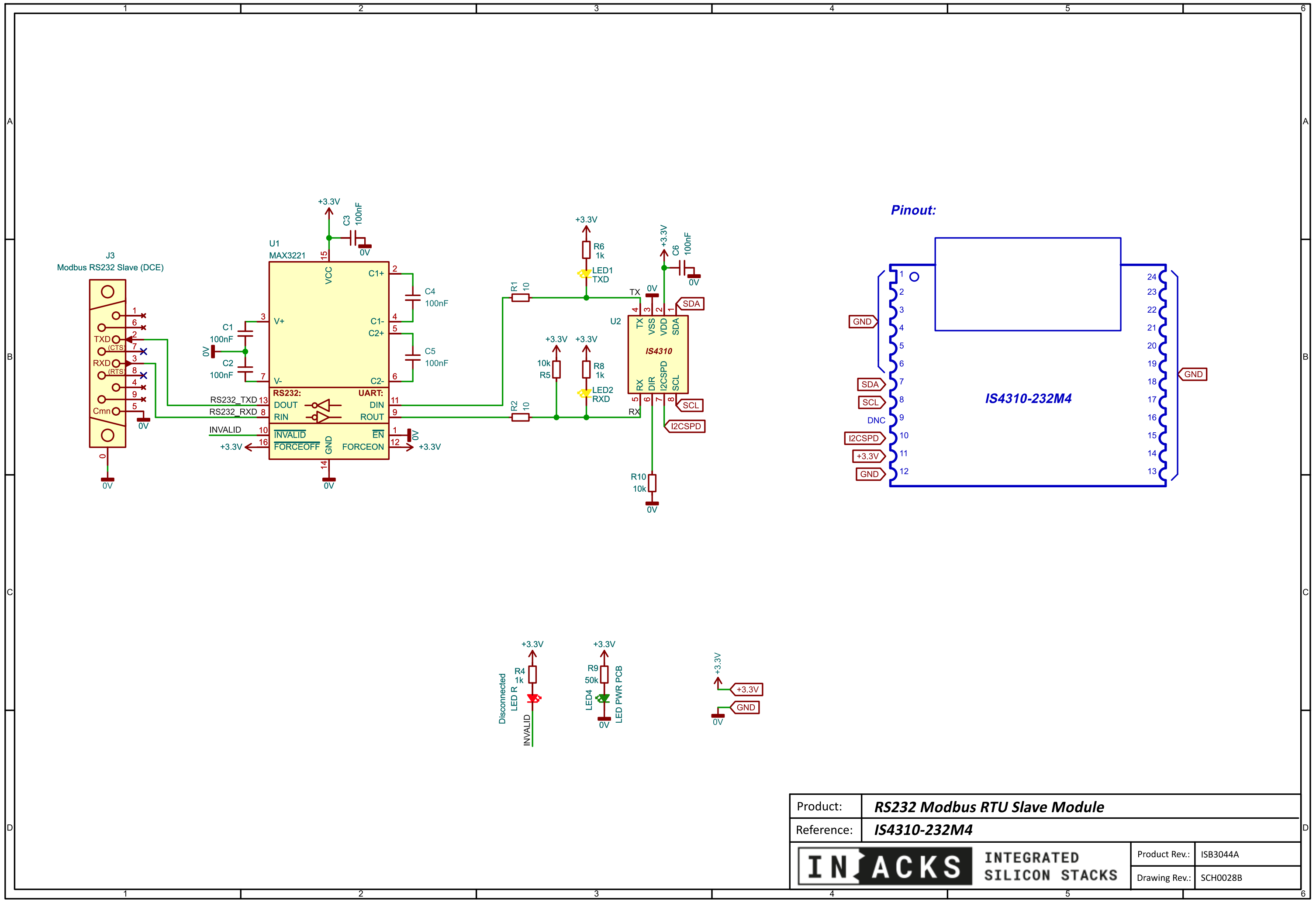
When selecting a cable, ensure it has shield and make sure to connect the cable shield to the connector shield to ensure proper electrical continuity.

Module's DB9 Connector for Modbus	
Module Front View (Female) 	1: NC
	2: TXD (Module's TXD pin)
	3: RXD (Module's RXD pin)
	4: NC
	5: Common
	6: NC
	7: NC
	8: NC
	9: NC
	Shield: Cable Shield

1.3. LEDs

LED Location	LED #	Description
	1	**Power Indicator** - Green: Module On - Off: Module Off
	2	**RxD Data Indicator** - Yellow Flashing: Receiving Data - Off: No Data
	3	**TxD Data Indicator** - Yellow Flashing: Transmitting Data - Off: No Data
	4	**Disconnected** - Red Steady: Error: the module is not properly plugged to a RS232 port. - Off: Connection Ok.

2. Schematic



Product:	RS232 Modbus RTU Slave Module
Reference:	IS4310-232M4

Product Rev.:	ISB3044A
Drawing Rev.:	SCH0028B

3. Firmware Implementation Guide

The following chapter presents firmware examples for different platforms for demonstration purposes only and is not part of the product standard. Customers must develop their own firmware, perform all necessary tests, and validate the final product according to applicable regulations and Modbus specifications.

3.1. Arduino Example

Coding for the IS4310 requires no dedicated library, making it easy to maintain and port to new Arduino boards or other microcontrollers.

This code reads the Modbus Slave ID and prints it to the terminal. Then, it stores a humidity variable in Modbus Holding Register address 0. This variable can be accessed by a Modbus Master device, such as a PC, PLC, or other controller.

You can download the Arduino project from the IS4310 product page.

This example uses the Kappa4310Ard Evaluation Board. Check the Kappa4310Ard product folder for more information.

```
#include <Wire.h>

void writeHoldingRegister(uint16_t holdingRegisterAddress, uint16_t data) {
  Wire.beginTransmission(0x11); // This is the I2C Chip Address of the IS4310. Never changes.
  // A Holding Register address is 16-bits long, so we need to write 2 bytes to indicate the address.
  Wire.write((holdingRegisterAddress >> 8) & 0xFF); // Send high 8-bits of the Holding Register Address we want to write.
  Wire.write(holdingRegisterAddress & 0xFF); // Send low 8-bits of the Holding Register Address we want to write.
  // A Holding Register data register is 16-bits long. So we need to write 2 bytes to make a full Holding Register Write:
  Wire.write((data >> 8) & 0xFF); // Send high 8-bits of the data we want to write to the Holding Register.
  Wire.write(data & 0xFF); // Send low 8-bits of the data we want to write to the Holding Register.
  Wire.endTransmission();
}

uint16_t readHoldingRegister(uint16_t holdingRegisterAddress) {
  uint16_t result; // This is the variable where the read data will be saved.
  Wire.beginTransmission(0x11); // This is the I2C Chip Address of the IS4310. Never changes.
  // A Holding Register address is 16-bits long, so we need to write 2 bytes to indicate the address.
  Wire.write((holdingRegisterAddress >> 8) & 0xFF);  // Send high 8-bits of the Holding Register Address we want to read.
  Wire.write(holdingRegisterAddress & 0xFF);         // Send low 8-bits of the Holding Register Address we want to read.
  Wire.endTransmission(false);
  // A Holding Register data register is 16-bits long. So we need to read 2 bytes to make a full Holding Register Read:
  Wire.requestFrom(0x11, 2);  // From the IS4310, request 2 bytes (2 bytes make a full Holding Register).
  result = Wire.read(); // Read the first byte.
  result = result << 8; // Make space for the second byte.
  result = result | Wire.read(); // Read the second byte.
  return result; // Return the read 16-bit register.
}

void setup() {
  uint16_t ModbusSlaveID;

  Wire.begin(); // Initialize the I2C.
  Serial.begin(9600); // Initialize  the Serial for the prints.

  // The Modbus Slave ID is stored in the Holding Register Address 500 of the IS4310, let's read it:
  ModbusSlaveID = readHoldingRegister(500);

  // Let's print the read Modbus Slave ID:
  Serial.println("");
  Serial.print("The Modbus Slave Address is: ");
  Serial.println(ModbusSlaveID);
}

void loop() {
  uint16_t humidity = 47; // Let's imagine a humidity sensor that reads a level of 47% RH.

  // Let's write the humidity to the Holding Register Address 0:
  writeHoldingRegister(0, humidity);
  delay(1000);
}
```

3.2. STM32 Example

Coding for the IS4310 requires no dedicated library, making it easy to maintain and port to new STM32 or other microcontrollers

The following code is an abstraction of the main.c file from the ISXMPL4310ex9 example. All external HAL routines and function calls have been removed for explanation proposals.

This example demonstrates:

1. How to read a potentiometer (simulating a sensor) and store its state in Holding Register 0.
2. How to control an RGB LED (simulating an actuator) using GPIO pins based on values in Holding Registers 1, 2, and 3.

You can download the full STM32 project from the IS4310 product page.

This example uses the Kappa4310Ard Evaluation Board. Check the Kappa4310Ard product folder for more information.

```
uint16_t readHoldingRegister(uint16_t registerAdressToRead) {
    uint8_t IS4310_I2C_Chip_Address; // This variable stores the I2C chip address of the IS4310.
    IS4310_I2C_Chip_Address = 0x11; // The IS4310's I2C address is 0x11.
    // The STM32 HAL I2C library requires the I2C address to be shifted left by one bit.
    // Let's shift the IS4310 I2C address accordingly:
    IS4310_I2C_Chip_Address = IS4310_I2C_Chip_Address << 1;

    // The following array will store the read data.
    // Since each holding register is 16 bits long, reading one register requires reading 2 bytes.
    uint8_t readResultArray[2];

    // This variable will contain the final result:
    uint16_t readResult;

    /*
     * This is the HAL function to read from an I2C memory device. The IS4310 is designed to operate as an I2C memory.
     *
     * HAL_I2C_Mem_Read parameters explained:
     * 1. &hi2c1: This is the name of the I2C that you're using. You set this in the CubeMX. Don't forget the '&'.
     * 2. IS4310_I2C_Chip_Address: The I2C address of the IS4310 (must be left-shifted).
     * 3. registerAdressToRead: The holding register address to read from the IS4310.
     * 4. I2C_MEMADD_SIZE_16BIT: You must indicate the memory addressing size. The IS4310 memory addressing is 16-bits.
     * This keyword is an internal constant of HAL libraries. Just write it.
     * 5. readResultArray: An 8-bit array where the HAL stores the read data.
     * 6. 2: The number of bytes to read. Since one holding register is 16 bits, we need to read 2 bytes.
     * 7. 1000: Timeout in milliseconds. If the HAL fails to read within this time, it will skip the operation
     * to prevent the code from getting stuck.
     */
    HAL_I2C_Mem_Read(&hi2c1, IS4310_I2C_Chip_Address, registerAdressToRead, I2C_MEMADD_SIZE_16BIT, readResultArray, 2, 1000);

    // Combine two bytes into a 16-bit result:
```

```
    readResult = readResultArray[0];
    readResult = readResult << 8;
    readResult = readResult | readResultArray[1];

    return readResult;
}

void writeHoldingRegister(uint16_t registerAdressToWrite, uint16_t value) {
    uint8_t IS4310_I2C_Chip_Address;  // I2C address of IS4310 chip (7-bit).
    IS4310_I2C_Chip_Address = 0x11; // IS4310 I2C address is 0x11 (7-bit).
    // STM32 HAL expects 8-bit address, so shift left by 1:
    IS4310_I2C_Chip_Address = IS4310_I2C_Chip_Address << 1;

    // The HAL library to write I2C memories needs the data to be in a uint8_t array.
    // So, lets put our uint16_t data into a 2 registers uint8_t array.
    uint8_t writeValuesArray[2];
    writeValuesArray[0] = (uint8_t) (value >> 8);
    writeValuesArray[1] = (uint8_t) value;

    /*
     * This is the HAL function to write to an I2C memory device. To be simple and easy to use, the IS4310 is designed to operate as an I2C
memory.
     *
     * HAL_I2C_Mem_Write parameters explained:
     * 1. &hi2c1: This is the name of the I2C that you're using. You set this in the CubeMX. Don't forget the '&'.
     * 2. IS4310_I2C_Chip_Address: The I2C address of the IS4310 (must be left-shifted).
     * 3. registerAdressToWrite: The holding register address of the IS4310 we want to write to.
     * 4. I2C_MEMADD_SIZE_16BIT: You must indicate the memory addressing size. The IS4310 memory addressing is 16-bits.
     * This keyword is an internal constant of HAL libraries. Just write it.
     * 5. writeValuesArray: An 8-bit array where we store the data to be written by the HAL function.
     * 6. 2: The number of bytes to write. Since one holding register is 16 bits, we need to write 2 bytes.
     * 7. 1000: Timeout in milliseconds. If the HAL fails to write within this time, it will skip the operation
     * to prevent the code from getting stuck.
     */
    HAL_I2C_Mem_Write(&hi2c1, IS4310_I2C_Chip_Address, registerAdressToWrite, I2C_MEMADD_SIZE_16BIT, writeValuesArray, 2, 1000);
}

while (1) {
    // This will store the potentiometer value:
    uint16_t potentiometerValue;
    // This will store the read value of the Holding Registers 1, 2 and 3:
    uint16_t holdingRegister1;
    uint16_t holdingRegister2;
    uint16_t holdingRegister3;

    // Read Holding Registers 1, 2 and 3:
    holdingRegister1 = readHoldingRegister(1);
    holdingRegister2 = readHoldingRegister(2);
    holdingRegister3 = readHoldingRegister(3);
    // If the value of each read Holding register is different from 0,
    // let's turn on the corresponding LED:
    if (holdingRegister1 >= 1) {
        HAL_GPIO_WritePin(RGB_Red_GPIO_Port, RGB_Red_Pin, GPIO_PIN_SET);
    } else {
        HAL_GPIO_WritePin(RGB_Red_GPIO_Port, RGB_Red_Pin, GPIO_PIN_RESET);
    }

    if (holdingRegister2 >= 1) {
        HAL_GPIO_WritePin(RGB_Green_GPIO_Port, RGB_Green_Pin, GPIO_PIN_SET);
    } else {
        HAL_GPIO_WritePin(RGB_Green_GPIO_Port, RGB_Green_Pin, GPIO_PIN_RESET);
    }

    if (holdingRegister3 >= 1) {
        HAL_GPIO_WritePin(RGB_Blue_GPIO_Port, RGB_Blue_Pin, GPIO_PIN_SET);
    } else {
        HAL_GPIO_WritePin(RGB_Blue_GPIO_Port, RGB_Blue_Pin, GPIO_PIN_RESET);
    }

    /*
     * Read ADC value from potentiometer (0-4095),
     * and write it to Holding Register 0.
     */
    HAL_ADC_Start(&hadc1); // Start the HAL ADC
    HAL_ADC_PollForConversion(&hadc1, 400); // Perform an ADC read
    // Get the ADC value:
    potentiometerValue = HAL_ADC_GetValue(&hadc1);
    // Store the ADC value to the Holding Register 0:
    writeHoldingRegister(0, potentiometerValue);
    // Stop the HAL ADC
    HAL_ADC_Stop(&hadc1);
}
```

3.3. Raspberry Pi Example

Coding for the IS4310 requires no dedicated library, making it easy to maintain and port to new Raspberry Pi boards or other single board computers (SBC).

This Python script communicates with the IS4310 Modbus RTU chip via I2C using a Raspberry Pi.

It demonstrates:

1. How to read a push button (simulating a sensor) and store its state in Holding Register 0.
2. How to control an RGB LED (simulating an actuator) using PWM on GPIO pins 12, 13, and 19, based on values in Holding Registers 1, 2, and 3.

A value of 0 turns off the LEDs, and a value of 100 sets them to maximum brightness.

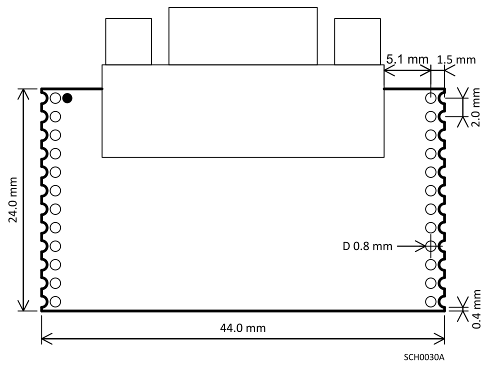
This example uses the Kappa4310Rasp Evaluation Board. Check the Kappa4310Ard product page for more information.

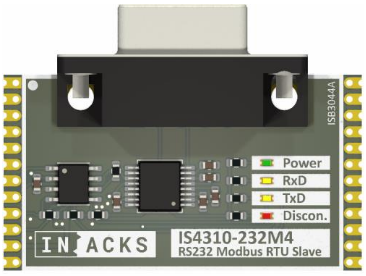
You can download the full Raspberry Pi Python project from the IS4310 product page.

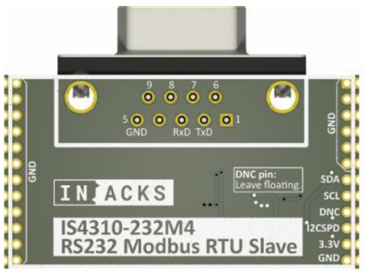
```
# IS4310 Modbus Code Example for Raspberry Pi
# ----------------------------------------------------------
# This Python script communicates with the IS4310 Modbus RTU chip via I²C using a Raspberry
Pi.
# It demonstrates how to read a push button (simulating a sensor) and store its value in
Holding Register 0.
# It also controls an RGB LED (simulating an actuator) using PWM pins 12, 13, and 19, based on
the values in Holding Registers 1, 2, and 3.
# A value of 0 turns off the LEDs, and a value of 100 sets them to maximum brightness.
#
#  You can test this code using the **Kappa4310Rasp Evaluation Board**.
# Buy it at: [www.inacks.com/kappa4310rasp](https://www.inacks.com/kappa4310rasp)
# Download the IS4310 datasheet at: www.inacks.com/is4310

from smbus2 import SMBus, i2c_msg
import RPi.GPIO as GPIO
import time

I2C_BUS = 1  # I2C bus number on Raspberry Pi (usually 1)
DEVICE_ADDRESS = 0x11  # 7-bit I2C address of the IS4310 Modbus RTU chip
GPIO.setmode(GPIO.BCM)  # Use BCM pin numbering scheme

# Define GPIO pins for three LEDs and push button
led_pin1 = 12
led_pin2 = 13
led_pin3 = 19
push_button_pin = 26

# Setup push button pin as input with internal pull-down resistor enabled
GPIO.setup(push_button_pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

# Setup LED pins as outputs
GPIO.setup(led_pin1, GPIO.OUT)
GPIO.setup(led_pin2, GPIO.OUT)
GPIO.setup(led_pin3, GPIO.OUT)

# Initialize PWM on LED pins at 1 kHz frequency
pwm1 = GPIO.PWM(led_pin1, 1000)
pwm2 = GPIO.PWM(led_pin2, 1000)
pwm3 = GPIO.PWM(led_pin3, 1000)

# Start PWM with 0% duty cycle (LEDs off initially)
pwm1.start(0)
pwm2.start(0)
pwm3.start(0)

def write_register(register, data):
    """
    Write a 16-bit data value to a 16-bit register address on the I2C device.

    :param register: 16-bit register address (split into high and low bytes)
    :param data: 16-bit data to write (split into high and low bytes)
    """
    high_addr = (register >> 8) & 0xFF  # Extract high byte of register address
    low_addr = register & 0xFF          # Extract low byte of register address
    data_high = (data >> 8) & 0xFF      # Extract high byte of data
    data_low = data & 0xFF              # Extract low byte of data

    # Open I2C bus, send write message: [register high, register low, data high, data low]
    with SMBus(I2C_BUS) as bus:
        msg = i2c_msg.write(DEVICE_ADDRESS, [high_addr, low_addr, data_high, data_low])
        bus.i2c_rdwr(msg)

def read_register(start_register):
    """
    Read a 16-bit value from a 16-bit register address on the I2C device.

    :param start_register: 16-bit register address to read from
    :return: 16-bit integer value read (big-endian)
    """
    high_addr = (start_register >> 8) & 0xFF  # High byte of register address
    low_addr = start_register & 0xFF          # Low byte of register address

    with SMBus(I2C_BUS) as bus:
        # Write register address first to set internal pointer
        write_msg = i2c_msg.write(DEVICE_ADDRESS, [high_addr, low_addr])
        # Prepare to read 2 bytes from the device
        read_msg = i2c_msg.read(DEVICE_ADDRESS, 2)
        bus.i2c_rdwr(write_msg, read_msg)

        data = list(read_msg)  # Read bytes as list of ints
        # Combine high and low bytes into 16-bit integer (big-endian)
        value = (data[0] << 8) | data[1]
        return value

try:
    while True:
        # Read push button state (0 or 1)
        button_value = GPIO.input(push_button_pin)

        # Write button state to register 0 of the device
        write_register(0, button_value)

        # Read PWM values from registers 1, 2, and 3
        pwm_val1 = read_register(1)
        pwm_val2 = read_register(2)
        pwm_val3 = read_register(3)

        # Cap PWM values at max 100 to avoid invalid duty cycles
        if pwm_val1 > 100:
            pwm_val1 = 100
        if pwm_val2 > 100:
            pwm_val2 = 100
        if pwm_val3 > 100:
            pwm_val3 = 100

        # Calculate duty cycles by inverting the PWM value (100 - value)
        # abs() used to ensure positive duty cycle, just in case
        duty1 = abs(pwm_val1 - 100)
        duty2 = abs(pwm_val2 - 100)
        duty3 = abs(pwm_val3 - 100)

        # Print duty cycle values for debugging (tab-separated)
        print(f"{duty1}\t{duty2}\t{duty3}")

        # Update PWM duty cycles to control LED brightness
        pwm1.ChangeDutyCycle(duty1)
        pwm2.ChangeDutyCycle(duty2)
        pwm3.ChangeDutyCycle(duty3)

        # Small delay to avoid excessive CPU load
        time.sleep(0.05)

except KeyboardInterrupt:
    # Gracefully handle Ctrl+C exit
    print("Exiting...")

finally:
    # Stop all PWM signals and cleanup GPIO pins on exit
    pwm1.stop()
    pwm2.stop()
    pwm3.stop()
    GPIO.cleanup()
```

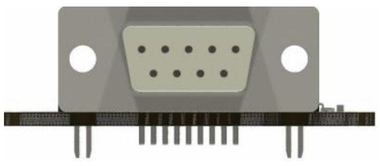
4. Mechanical Dimensions

M4 Package







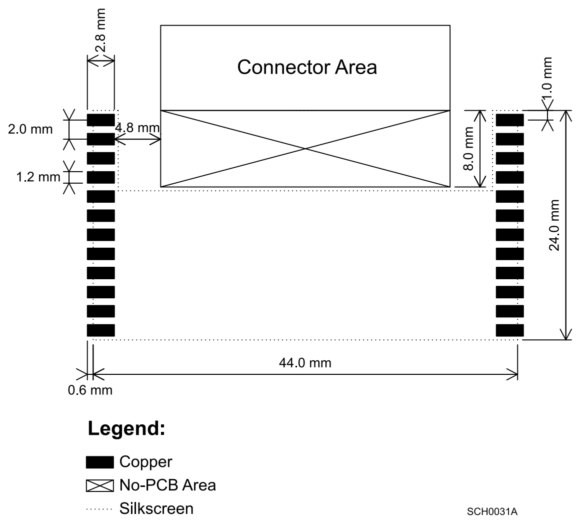


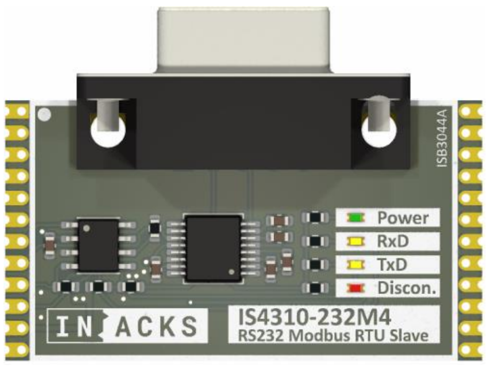
INACKS
INTEGRATED
SILICON STACKS

Units
Millimeters

Notes
This drawing is for general information only.
Not to scale.

M4 Recommended Footprint





Units
Millimeters

Notes
This drawing is for general information only.
Not to scale.

Content

Appendix

Revision History

Document Revision

Date	Revision Code	Description
June 2025	ISDOC128**B**	Changed renders for pictures. Added Arduino, STM32 and Raspberry Pi examples. Schematic in color.
February 2025	ISDOC128**A**	Initial Release

Module Revision

Date	Revision Code	Description
February 2025	ISB3044**A**	Initial Release

Documentation Feedback

Feedback and error reporting on this document are very much appreciated.

feedback@inacks.com

Sales Contact

For special order requirements, large volume orders, or scheduled orders, please contact our sales department at:

sales@inacks.com

Customization

INACKS can develop new products or customize existing ones to meet specific client needs. Please contact our engineering department at:

engineering@inacks.com

Trademarks

This company and its products are developed independently and are not affiliated with, endorsed by, or associated with any official protocol or standardization entity. All trademarks, names, and references to specific protocols remain the property of their respective owners.

Disclaimer