**What is RS-485?**

RS-485 is an industrial specification that defines the electrical interface and physical layer for point-to-point communication of electrical devices. The RS-485 standard allows for long cabling distances in electrically noisy environments and can support multiple devices on the same bus.

### Modbus RTU Format

|  |  |  |  |
| --- | --- | --- | --- |
| Slave Address | Function Code | Data | CRC |
| 00-FF | Hex Value 8-bits | Hex value 16 bits | Hex Value 16-bits |

Modbus RTU messages are composed of the following parts:

1. **Slave Address**: The address of the slave device.
2. **Function Code**: Indicates the type of action to perform.
3. **Data**: Data being transmitted or received.
4. **CRC**: Cyclic Redundancy Check for error-checking.

### [**Industrial USB to RS485 Converter**](https://www.seeedstudio.com/Industrial-USB-to-RS485-Converter-p-4371.html?utm_source=blog&utm_medium=blog)



* Clueing in from its name, this is a **USB to RS485** converter with industrial-grade features.
* It converts a **USB port to use with RS485**.
* Best known for the simplicity of usage it provides, it also features embedded circuits protection features such as**lightning-proofing, resettable fuse, ESD protection and TVS diode.**
* It allows for automatic transceiving, providing convenience.
* It is also an **ideal choice** for industrial equipment use due to its**high speed, reliability and safety**.

Testing Software : QModMaster

**Read Digital Inputs:**

Sent: b'\x01\x02\x03\xe8\x00\x08\xf9\xbc'

* **01**: Slave address (0x01)
* **02**: Function code (0x02), which is "Read Discrete Inputs".
* **03 E8**: Starting address (0x03E8 in hexadecimal, which is 1000 in decimal)
* **00 08**: Quantity of inputs to read (0x0008, which is 8 in decimal)
* **F9 BC**: CRC (Cyclic Redundancy Check) for error-checking.

Received: b'\x01\x02\x01\x00\xa1\x88'

* **01**: Slave address (0x01), echoing the address of the device that responded.
* **02**: Function code (0x02), echoing the function code of the request.
* **01**: Byte count (0x01), indicating one byte of data follows.
* **00**: Data (0x00), indicating the status of the 8 discrete inputs (all are off or false).
* **A1 88**: CRC (Cyclic Redundancy Check) for error-checking.

Here Received Response index from 0 to 7 hence, Data Received that indicating 8-bits input states position at index 3, Response[3] to Separate individual bits AND operation by shifting N times for Nth bit.

**#Code to Separate Individual Inputs:**

def get\_inputs\_state(response):

# Assuming the response contains input states in the format of a single byte

input\_states = response[3] # Change the index based on actual Modbus RTU response structure

states = [(input\_states >> i) & 1 for i in range(8)]

return states

**Write Mltiple Coils:**

**Sent: b'\x01\x0f\x0b\xb8\x00\x08\x01\x00\_\xf5'**

* **01**: Slave address (0x01).
* **0F**: Function code (0x0F), which is "Write Multiple Coils".
* **0B B8**: Starting address (0x0BB8 in hexadecimal, which is 3000 in decimal).
* **00 08**: Quantity of coils to write (0x0008, which is 8 in decimal).
* **01**: Byte count (0x01), indicating the number of bytes of coil data.
* **FF**: Coil data (0xFF), setting all 8 coils to ON (binary 11111111).
* **FF**: Coil data (0x00), setting all 8 coils to OFF (binary 00000000).
* **1F B5**: CRC (Cyclic Redundancy Check) for error-checking.

**Received: b'\x01\x0f\x0b\xb8\x00\x08\xd6\x0c'**

* **01**: Slave address (0x01), echoing the address of the device that responded.
* **0F**: Function code (0x0F), echoing the function code of the request.
* **0B B8**: Starting address (0x0BB8 in hexadecimal, which is 3000 in decimal)
* **00 08**: Quantity of coils written (0x0008, which is 8 in decimal)
* **D6 0C**: CRC (Cyclic Redundancy Check) for error-checking.

**To set Individual Outputs Bits should be Manipulated in DATA:**

# Define variable to give individual output bits value for 8 coils  
coil\_states = [True, False, True, False, True, False, True, False]   
byte\_value = set\_outputs\_state(coil\_states)

#Operate Individual Coils

def set\_outputs\_state(coil\_states):

byte\_value = 0

for i, state in enumerate(coil\_states):

if state:

byte\_value |= (1 << i)

return byte\_value

### Defining Coil States

coil\_states = [True, False, True, False, True, False, True, False]

* coil\_states is a list of boolean values, where each value represents the state of a corresponding coil.
* True indicates the coil is on (1), and False indicates the coil is off (0).

### Function to Set Output State

def set\_outputs\_state(coil\_states):

byte\_value = 0

for i, state in enumerate(coil\_states):

if state:

byte\_value |= (1 << i)

return byte\_value

* The function set\_outputs\_state takes a list of coil states (coil\_states) as input.
* byte\_value is initialized to 0. This variable will store the final byte value representing the state of all coils.
* The for loop iterates over each coil state and its corresponding index (i).
  + enumerate(coil\_states) provides both the index (i) and the state (state) of each coil.
* Inside the loop, there is a check if state: to determine if the current coil is on (True).
  + If the coil is on, the expression (1 << i) shifts the bit 1 to the left by i positions.
  + The |= (1 << i) operation performs a bitwise OR between byte\_value and the shifted bit, setting the corresponding bit in byte\_value to 1.
* Finally, the function returns the computed byte\_value.

### Example Execution

byte\_value = set\_outputs\_state(coil\_states)

* set\_outputs\_state(coil\_states) is called with the defined coil\_states list.
* For the given coil\_states [True, False, True, False, True, False, True, False], the function computes the byte\_value.

### Calculation Breakdown

1. Initial byte\_value: 00000000 (binary)
2. For the first True (i=0): 00000001 (binary)
3. For the second False (i=1): no change
4. For the third True (i=2): 00000101 (binary)
5. For the fourth False (i=3): no change
6. For the fifth True (i=4): 00010101 (binary)
7. For the sixth False (i=5): no change
8. For the seventh True (i=6): 01010101 (binary)
9. For the eighth False (i=7): no change

So, the final byte\_value is 01010101 in binary, which is 85 in decimal.

Add = int(input("Enter Address Value (3000-3007): "))

Rly = int(input("Enter State Value (0 or 1): "))

if Add < 3000 or Add > 3007:

print("Invalid address. Please enter a value between 3000 and 3007.")

step = Add - 3000 # Map address to coil\_states index

coil\_states[step] = True if Rly == 1 else False

byte\_value = set\_outputs\_state(coil\_states)

**Read Coils:**

**Sent: b'\x01\x01\x0b\xb8\x00\x08\xbf\xcd'**

* **01**: Slave address (0x01)
* **01**: Function code (0x01), which is "Read Coils".
* **0B B8**: Starting address (0x0BB8 in hexadecimal, which is 3000 in decimal)
* **00 08**: Quantity of Coils to read (0x0008, which is 8 in decimal)
* **BF CD**: CRC (Cyclic Redundancy Check) for error-checking.

**Received: b'\x01\x01\x01\x00Q\x88'**

* **01**: Slave address (0x01), echoing the address of the device that responded.
* **01**: Function code (0x01), echoing the function code of the request.
* **01**: Byte count (0x01), indicating one byte of data follows.
* **00**: Data (0x00), indicating the status of the 8 Output Coils (all are off or false).
* **0Q 88**: CRC (Cyclic Redundancy Check) for error-checking.

**Here also use same logic of Discrete Read Inputs to get individual coil states**

**Output States: [0, 0, 0, 0, 0, 0, 0, 0]**

**16-bits CRC Calculator:**

#CRC Calculator

def calculate\_crc16(data):

crc = 0xFFFF

for pos in data:

crc ^= pos

for \_ in range(8):

if (crc & 1) != 0:

crc >>= 1

crc ^= 0xA001

else:

crc >>= 1

high\_byte = (crc & 0xFF00) >> 8

low\_byte = crc & 0x00FF

return high\_byte, low\_byte

**Simple CODE to Send Command and get Response**

import socket

import time

# Configuration

converter\_ip = '192.168.1.200' # IP address of your Ethernet to RS485 converter

converter\_port = 4196 # Port number (Modbus TCP default port)

# Create a TCP/IP socket

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

rs485\_command = b'\x01\x02\x03\xE8\x00\x08'

high\_byte, low\_byte = calculate\_crc16(rs485\_command\_read)

rs485\_command\_read += bytes([low\_byte, high\_byte])

sock.sendall(rs485\_command\_read)

print(f"Sent: {rs485\_command\_read}")

# Wait for the response

response = sock.recv(1024)

print(f"Received: {response}")

**This code is for how to send a command to an Ethernet to RS485 converter and receive a response using a TCP/IP socket:**

### Importing Required Modules

import socket

import time

- socket: This module provides access to the BSD socket interface for network communication.

-time: This module provides various time-related functions, though it's not used in the provided code.

### Configuration

converter\_ip = '192.168.1.200' # IP address of your Ethernet to RS485 converter

converter\_port = 4196 # Port number (Modbus TCP default port)

- converter\_ip : The IP address of the Ethernet to RS485 converter.

- converter\_port: The port number to connect to. Port `4196` is commonly used for Modbus TCP communication.

### Creating a TCP/IP Socket

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

- This line creates a new socket object using the `AF\_INET` address family (IPv4) and `SOCK\_STREAM` socket type (TCP).

### Defining the RS485 Command

rs485\_command = b'\x01\x02\x03\xE8\x00\x08'

-rs485\_command is a byte string representing the command to be sent over the RS485 interface. The specific command here is an example and its meaning depends on the RS485 protocol being used.

### Calculating and Adding the CRC16 Checksum

high\_byte, low\_byte = calculate\_crc16(rs485\_command)

rs485\_command += bytes([low\_byte, high\_byte])

- The function returns two values: the high byte and the low byte of the CRC16 checksum.

- rs485\_command += bytes([low\_byte, high\_byte]): The calculated CRC16 bytes are appended to the original command to ensure data integrity during transmission.

### Sending the Command

sock.sendall(rs485\_command)

print(f"Sent: {rs485\_command}")

```

- sock.sendall(rs485\_command): This method sends the complete command, including the CRC16 checksum, over the socket connection. `sendall` ensures that all data is sent.

### Receiving the Response

response = sock.recv(1024)

print(f"Received: {response}")

- response = sock.recv(1024): This line waits for a response from the converter and reads up to 1024 bytes of data. The received data is stored in the `response` variable.