Chipkin Modbus Explorer

User Manual

Product Code: CAS-1000-20

Manual Version: 1.0.0

Application Version: 1.2.0

Last Updated: January 22, 2026

Table of Contents

Quick Links

Part I: Chipkin Modbus Explorer

In This Section

This section covers using Chipkin Modbus Explorer for commissioning, testing, and troubleshooting Modbus devices.

Jump to the technical reference: Part II: Modbus Protocol Reference

---

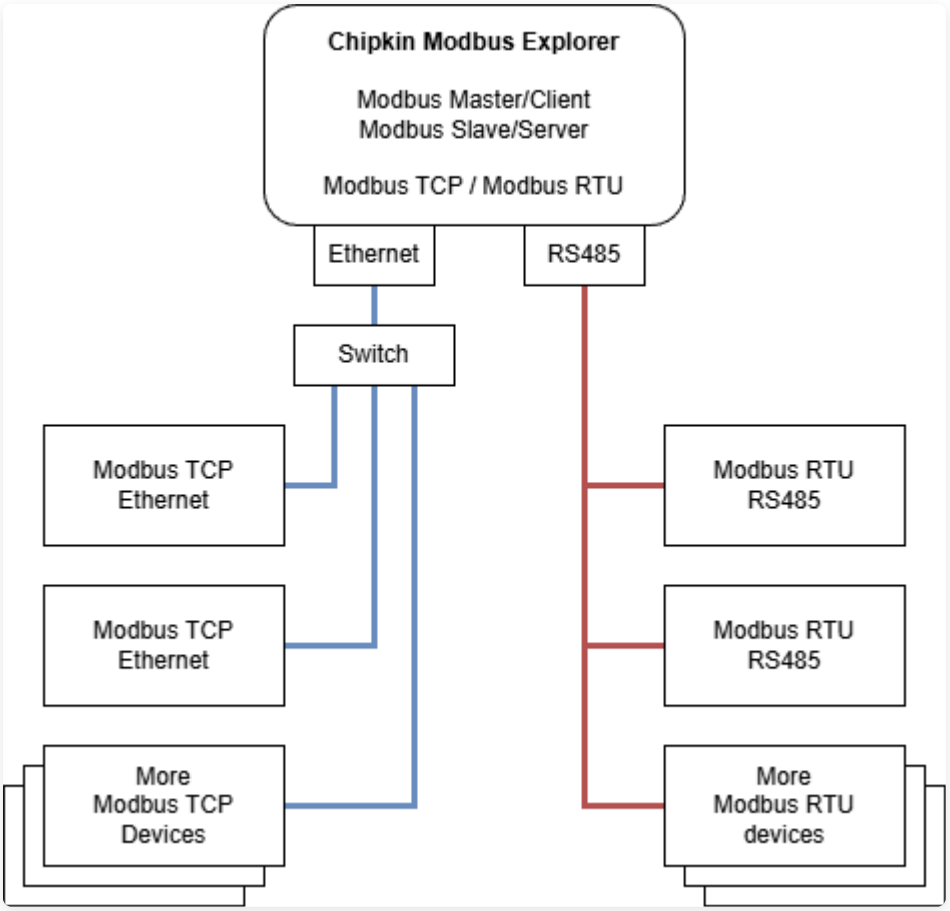
Introduction

Welcome to the Chipkin Modbus Explorer, a powerful and user-friendly tool designed to simplify your work with the Modbus protocol. Whether you're commissioning a new device, troubleshooting a complex communication issue, or developing a Modbus-

enabled application, this tool provides a comprehensive solution.

This manual serves as both a guide to the features of the Chipkin Modbus Explorer and a practical resource for understanding and resolving real-world Modbus challenges.

Block Diagram



Core Features

The Chipkin Modbus Explorer is organized around the three primary tasks you'll encounter when working with Modbus:

- **Modbus Client**: Manually send requests, write data, and interact directly with any Modbus TCP or RTU device. This is your main tool for testing, commissioning, and debugging.
- **Discovery**: Automatically scan your network to find Modbus devices. This feature can save significant time by identifying devices and their available data points without manual configuration.
- **Simulator**: Create a virtual Modbus slave on your computer. This is invaluable for developing and testing a Modbus master application when physical hardware is not available.

Who Is This For?

This tool is designed for a diverse audience, including:

- **Controls Technicians and System Integrators**: Quickly commission devices, verify installations, and troubleshoot communication problems on-site.
- **Software and Hardware Developers**: Test your Modbus master or slave implementations against a reliable and compliant tool. Use the Simulator to create various test cases and validate error handling.
- **Facility Managers and Engineers**: Diagnose network issues and confirm that devices are reporting data correctly.
- **Anyone New to Modbus**: Learn the protocol interactively in a clear and forgiving environment. See requests and responses in real-time to understand how Modbus works.

If you encounter any issues, the **Troubleshooting Guide** is an excellent starting point.

Key Capabilities

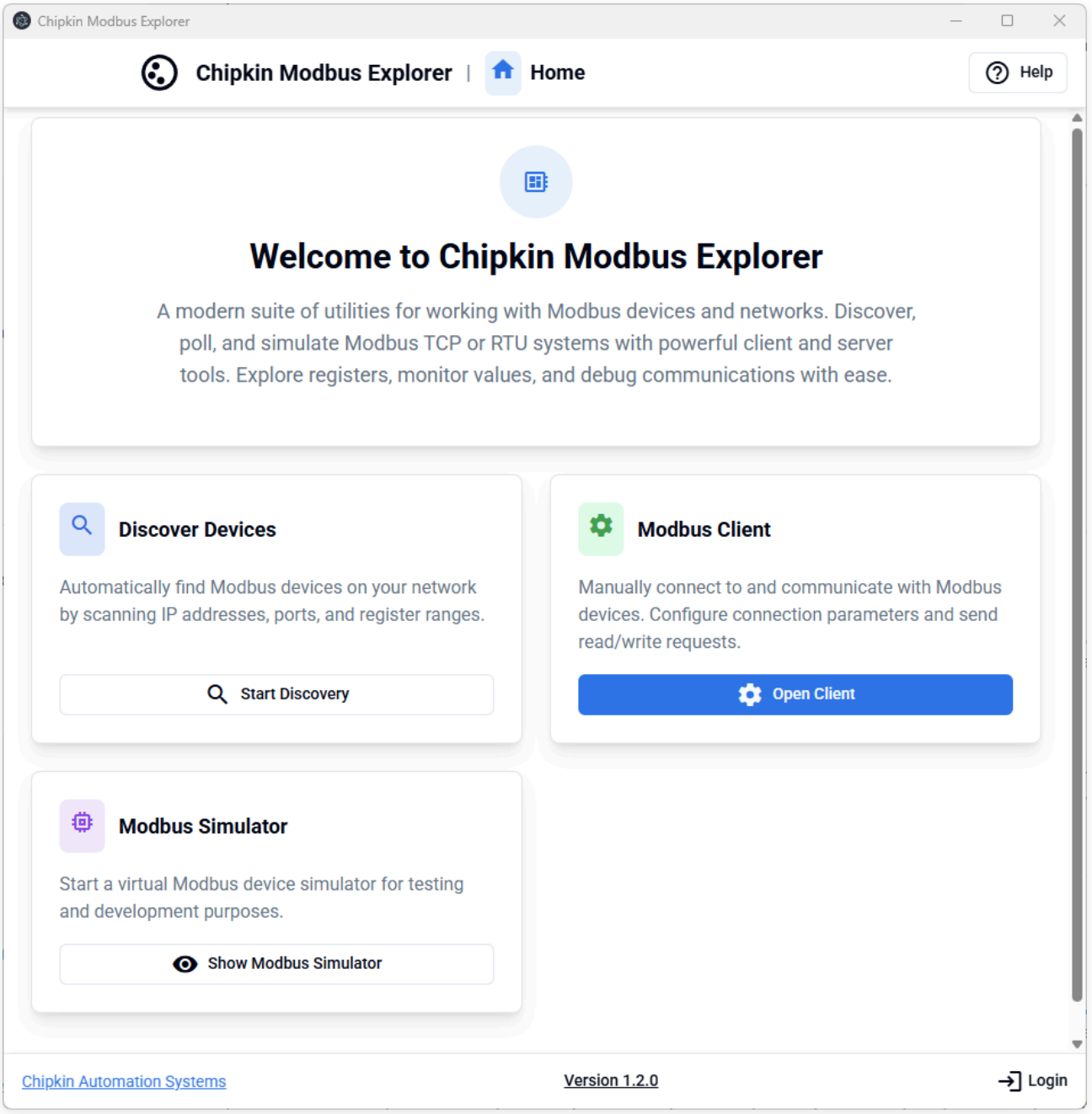
- **Universal Protocol Support**: Connect to both **Modbus TCP** (over Ethernet) and **Modbus RTU** (over serial RS-232/RS-485) devices.
- **Complete Function Code Coverage**: Supports all standard Modbus function codes for reading and writing data. For a detailed reference, see the Modbus Protocol section.
- **Flexible Data Interpretation**: Automatically decodes 16-bit registers into 32-bit and 64-bit integers and floating-point numbers. It provides full control over byte and word order (endianness) to resolve common data formatting issues.
- **Advanced Logging**: A clear, color-coded message log shows every byte sent and received, making it easy to diagnose communication problems.
- **Persistent Configurations**: Save device configurations and settings to quickly resume your work.
- **Import/Export Functionality**: Load and save register lists from CSV files, which is ideal for documenting and sharing device configurations.

Getting Started

This guide provides a practical walkthrough of the Chipkin Modbus Explorer, covering the fundamental concepts you need to start communicating with Modbus devices.

Application Layout

The user interface is designed for simplicity and quick access to the main features.



- **Main Tools**: The home screen features three prominent buttons that navigate to the core functionalities of the application:
  - Modbus Client: For sending manual Modbus requests and analyzing responses.
  - Discovery: To automatically find and identify Modbus devices on your network.
  - Simulator: To create a virtual Modbus slave for testing purposes.
- **Header Navigation**: A persistent header at the top of the application allows you to return to the home screen from anywhere and access this help manual.

Core Concepts: A Practical Example

The most common task in Modbus is reading data from a device. Let's walk through a typical scenario.

Imagine a device manual states: "*The current temperature is stored as a 16-bit integer in Holding Register 40108.*"

Here's how to read this value using the Chipkin Modbus Explorer:

1. **Navigate to the Modbus Client**: From the home screen, click on "Modbus Client".
2. **Establish a Connection**:
    - For a **Modbus TCP** device, enter its IP Address (e.g., `192.168.1.100`) and ensure the port is set to `502`.
    - For a **Modbus RTU** device, select the correct COM port and configure the serial parameters (baud rate, parity, etc.) to match the device.
    - Click **Connect**. If you encounter issues, consult the Troubleshooting Guide.
3. **Construct the Read Request**:
    - **Function Code**: Select `03: Read Holding Registers`.
    - **Starting Address**: Enter `107`.
    - **Quantity**: Enter `1`.
4. **Send the Request**: Click the "Send Request" button.

The response from the device will appear in the log, and the value read from register `107` will be displayed in the results table.

Why Address 107? The #1 Point of Confusion

Modbus has two addressing schemes:

- **Public Addressing (1-based)**: Used in device manuals (e.g., `40108`). It's human-friendly.
- **Protocol Addressing (0-based)**: Used in the actual Modbus message sent over the wire. It's a zero-indexed offset.

You must always convert the public address to a protocol address. The Chipkin Modbus Explorer exclusively uses **protocol addresses**.

The conversion rule is: `Protocol Address = Public Address - Register Type Offset`

Register Type	Public Address Range	Offset	Example (Public)	Protocol Address
Coils	`00001` - `09999`	1	`00012`	`11`
Discrete Inputs	`10001` - `19999`	10001	`10055`	`54`
Input Registers	`30001` - `39999`	30001	`30200`	`199`
Holding Registers	`40001` - `49999`	**40001**	`40108`	`107`

If you were to enter the public address `40108` into the address field, the device would correctly return an **Exception 02: Illegal Data Address**, as this address is far outside its valid memory range.

What Do These Bytes Mean? The #2 Point of Confusion

Now, let's say the manual specifies: "*The temperature is a 32-bit floating-point value stored across registers 40108 and 40109.*"

You read two registers starting at address `107` and the response contains two raw 16-bit values, for example, `17224` and `2052`. How do you interpret this as a temperature?

This is an **endianness** (or byte/word order) problem. The Modbus standard is big-endian, but not all manufacturers follow it. A 32-bit value can be arranged in four different ways.

The Solution: The Chipkin Modbus Explorer automatically interprets multi-register values for you.

1. Read the two registers (`107` and `108`).
2. Examine the **32-bit Data Types** view in the results section.
3. Look for the "Float" interpretations.

FLOAT32 (ABCD)	FLOAT32 (DCBA)
3.765255e-39	1.470803e-41
0.0	0.0
88.900002	-4.292711e+8

The tool displays the floating-point value for all four common byte order combinations. One of them will match the expected value (e.g., `123.45`). This instantly tells you the correct endianness for that device.

Quick Reference Cheatsheet

When the Manual Says...	In Modbus Explorer, You Do This...
Read **Holding Register 40010**	Function: `03` , Address: `9` , Quantity: `1`
Read **Input Register 30005**	Function: `04` , Address: `4` , Quantity: `1`
Read **Coil 00025**	Function: `01` , Address: `24` , Quantity: `1`
Write `750` to **Holding Register 40050**	Function: `06` , Address: `49` , Value: `750`
Turn **Coil 25** ON	Function: `05` , Address: `24` , Value: `ON`

Next Steps

With a solid grasp of addressing and data representation, you are now equipped to use the full capabilities of the Chipkin Modbus Explorer.

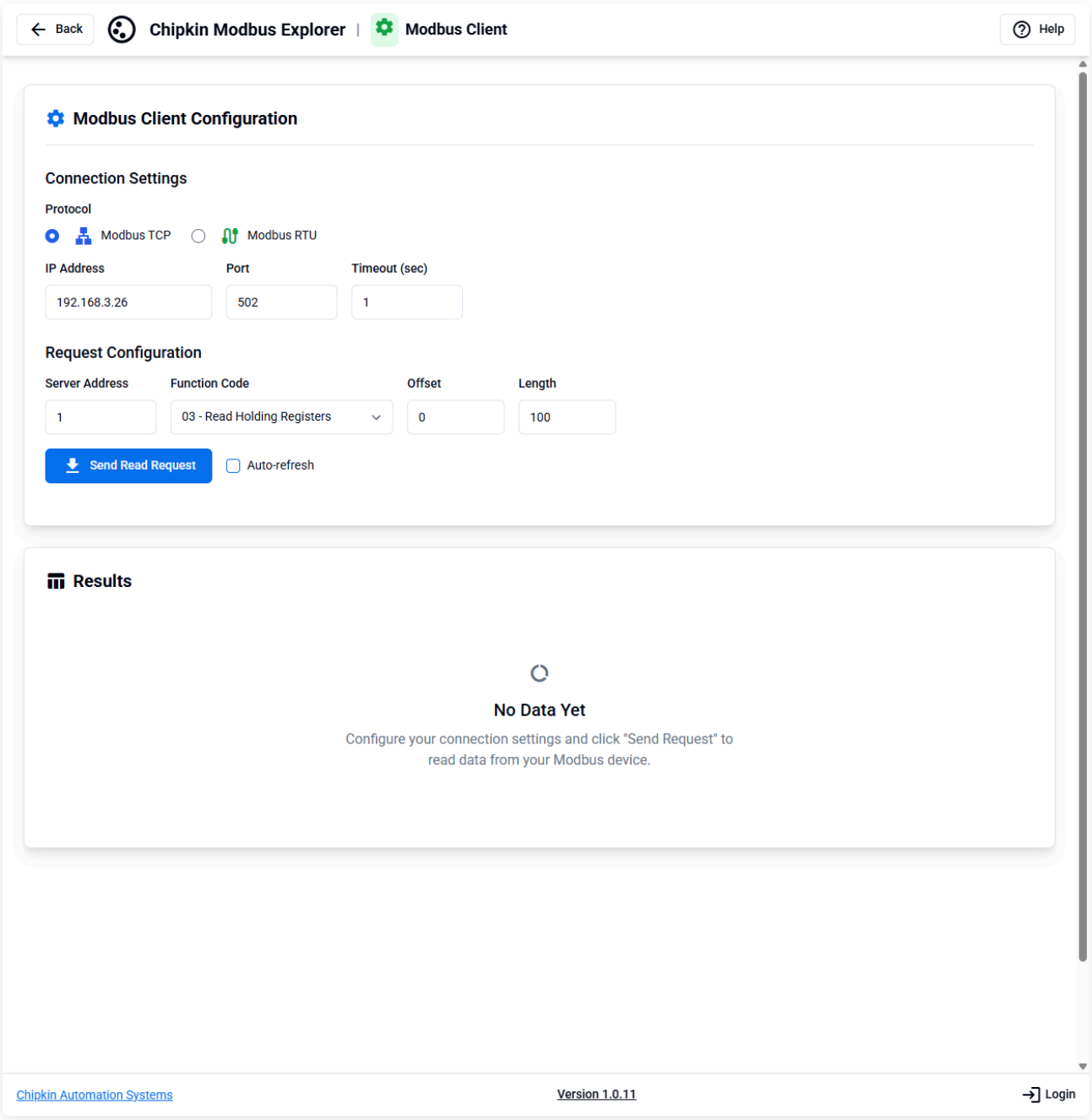
- **Modbus Client**: Explore advanced features for manual testing and analysis.
- **Simulator**: Learn to create a virtual Modbus device to test your own client applications without needing physical hardware.

- **Modbus Protocol**: Take a deep dive into the technical specifications of every function code and exception.

Modbus Client

The Modbus Client is your direct interface to any Modbus device. Use it to verify connections, read and write data, and diagnose communication problems before integrating devices into a larger system.

This guide will walk you through connecting to devices, reading and writing data, and solving the most common issues you'll encounter.



What Problems Does It Solve?

- **"Is this device even working?"** - Instantly verify connectivity and get a response.
- **"What data is in register 40108?"** - Read any register and see its value in multiple formats.
- **"My data looks like garbage."** - Fix scrambled data by correcting data types and byte order.
- **"I need to change a setpoint."** - Write new values to holding registers to control your device.
- **"Why am I getting a timeout?"** - Diagnose connection issues with clear error messages and integrated troubleshooting.

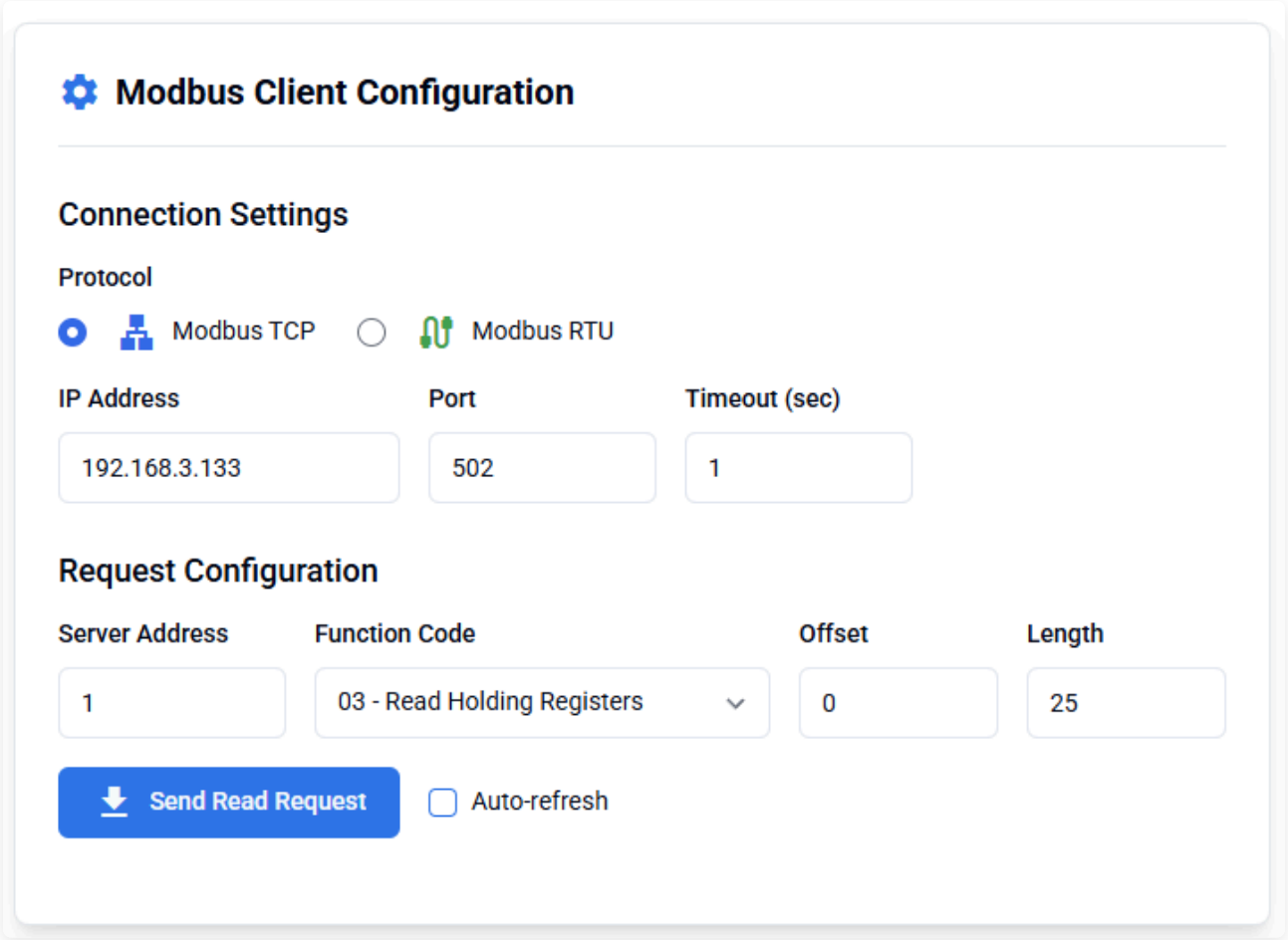
Quick Start: Read a Temperature Sensor in 5 Steps

Let's solve the most common task: reading a temperature value from a Modbus TCP device.

Scenario: Your documentation says "Temperature is at Holding Register **40108**" on a device with IP **192.168.1.50**.

Step 1: Set Connection Details

- **Protocol**: TCP
- **IP Address**: `192.168.1.50`
- **Port**: `502` (the standard)
- **Unit ID**: `1` (a safe default)



Step 2: Connect to the Device

- Click the **Connect** button.
- The status indicator should turn 🟢 **Green**.
- **Not green?** Your first problem to solve. Jump to our Connection Troubleshooting section.

Step 3: Configure the Register to Read

This is where most users get stuck. Here's the secret:

- **Function Code**: `03` (for Reading Holding Registers, `4xxxx` addresses).
- **Address**: `107` . **Wait, what?** See the note below.
- **Quantity**: `2` . Most floating-point values need two registers.
- **Data Type**: `FLOAT32` . Temperature is rarely a whole number.

> ***The Golden Rule of Modbus Addressing***
>
> *Device manuals use **Public Addresses** (like `40108`), but the protocol uses **Protocol Addresses** (`107`).*

> ***The Formula***: `Protocol Address = Public Address - Table Start - 1`
>
> - *For* `4xxxx` *registers:* `40108 - 40000 - 1 = 107`
> - *For* `3xxxx` *registers:* `30050 - 30000 - 1 = 49`
>
> *Our tool uses the* ***Protocol Address****. For a deep dive, see our Getting Started Cheatsheet.*

Step 4: Add the Register and Read

- Click **Add Row**. The register appears in the table below.
- Click the main **Read** button at the top.

Step 5: Interpret the Result

You'll see a value. Does it make sense?

- **Result is** `72.5` : **Success!** You've correctly read the temperature.
- **Result is** `1.12e-35` **or some other nonsense**: **You have a byte order problem.** This is the second most common Modbus issue.
  - In the row you added, find the **Byte Order** dropdown.
  - Try each of the four options (`ABCD` , `BADC` , `CDAB` , `DCBA`) and click **Read** again after each change. One of them will show the correct value.
  - For a full explanation, see our guide on Fixing Garbled Data.

You have now mastered the two biggest hurdles in Modbus: **addressing** and **byte order**.

Connecting to a Device

First, choose your protocol: **TCP** for network devices or **RTU** for serial (RS-232/RS-485).

TCP Connection

Use for devices connected via Ethernet.

Field	Description	Example
IP Address	The network address of your device.	`192.168.1.100`
Port	The TCP port. Almost always `502` .	`502`
Unit ID	The device's address on the Modbus link.	`1` or `255`
Timeout	How long to wait for a response (in ms).	`5000`

RTU Connection

Use for devices connected via a COM port (e.g., a USB-to-Serial adapter).

Field	Description	Common Values
COM Port	The serial port on your computer.	`COM3` , `COM4`
Baud Rate	Communication speed. **Must match device.**	**9600**, 19200
Parity	Error checking. **Must match device.**	**None**, Even
Data Bits	Bits per character. Almost always `8` .	**8**
Stop Bits	End of character marker. Almost always `1` .	**1**

> ***Pro Tip**: The most common RTU setting is **9600 8-N-1** (9600 baud, 8 data bits, None parity, 1 stop bit). If that doesn't work, try **19200 8-E-1**.*

Connection Troubleshooting

If the indicator isn't 🟢 **Green**, here's what to do.

Problem	TCP Solution	RTU Solution
Connection Timeout	1. Can you `ping` the IP? 2. Check firewall rules. 3. Is the device powered on?	1. **Baud Rate and Parity must be exact.** 2. Is the COM port correct? Check in Device Manager. 3. Are TX/RX wires swapped?
Connection Refused	1. Is the port correct? (Try 502). 2. Is another application using the port?	1. Is another program (like PuTTY) using the COM port? 2. Did you install the USB-to-Serial driver?
No Response	1. Is the Unit ID correct? Try `1` , then `255` . 2. See our detailed Timeout Errors Guide.	1. Is the Unit ID correct? 2. For RS-485, are termination resistors installed? 3. See our Triage Checklist.

Reading and Writing Data

The main table is where you build a list of registers to read or write.

Core Concepts for Reading Data

1. **Function Code (FC)**: The operation you want to perform.
    - `01 Read Coils` : Read digital ON/OFF outputs (`0xxxx`).
    - `02 Read Discrete Inputs` : Read digital ON/OFF inputs (`1xxxx`).
    - `03 Read Holding Registers` : Read configuration/setpoint values (`4xxxx`). **Most common.**
    - `04 Read Input Registers` : Read sensor/measurement values (`3xxxx`). **Second most common.**
2. **Address**: The **Protocol Address** you calculated (e.g., `107`).
3. **Quantity**: How many registers to read. For a 32-bit float, you need **2** registers. For a 64-bit value, you need **4**.

4. **Data Type & Byte Order**: How to interpret the raw bytes. If your value looks wrong, change the Byte Order. See Fixing Garbled Data.

How to Read

1. Fill in the configuration form (Function Code, Address, Quantity, etc.).
2. Click **Add Row**.
3. Click **Read**.
4. The `Value` and `Raw Hex` columns will populate.

How to Write

You can only write to **Coils (FC05, FC15)** and **Holding Registers (FC06, FC16)**.

1. First, **read** the register to see its current value.
2. Click the **pencil icon (🖍)** in the row you want to change.
3. Enter the new value in the dialog and click **Write**.
4. The app automatically uses the correct function code (e.g., `FC06` for a single register, `FC16` for a float).
5. **Best Practice**: Click **Read** again immediately to confirm the device accepted the new value.

Understanding the Results Table

Column	Description
Address	The 0-based protocol address.
Public Address	The 1-based address you see in manuals (e.g., `40101`).
Value	The final, interpreted value based on your Data Type and Byte Order settings.
Raw Hex	The actual hexadecimal bytes returned by the device. Invaluable for debugging.
Status	Shows `✅ OK`, `⚠️ Exception`, or `❌ Error`.

If you get an exception, the device is talking, but it doesn't like your request. See our Exception Code Handler to translate the code into a solution.

Import and Export (CSV)

Don't configure the same device twice. Use the **Import** and **Export** features to save your register maps to a CSV file.

- **Export**: Saves the current table (addresses, data types, etc.) to a file. Use this to **document** a device's configuration or create a **template**.
- **Import**: Loads a register map from a CSV file. Use this to quickly configure the client for a known device.

Pro Workflow: Create a Device Template

1. Connect to a new device and configure all its important registers.

2. Verify all values are reading correctly.
3. **Export** the table to `[DeviceModel]_template.csv`.
4. The next time you work with this model, simply **Import** the template to load the entire configuration instantly.

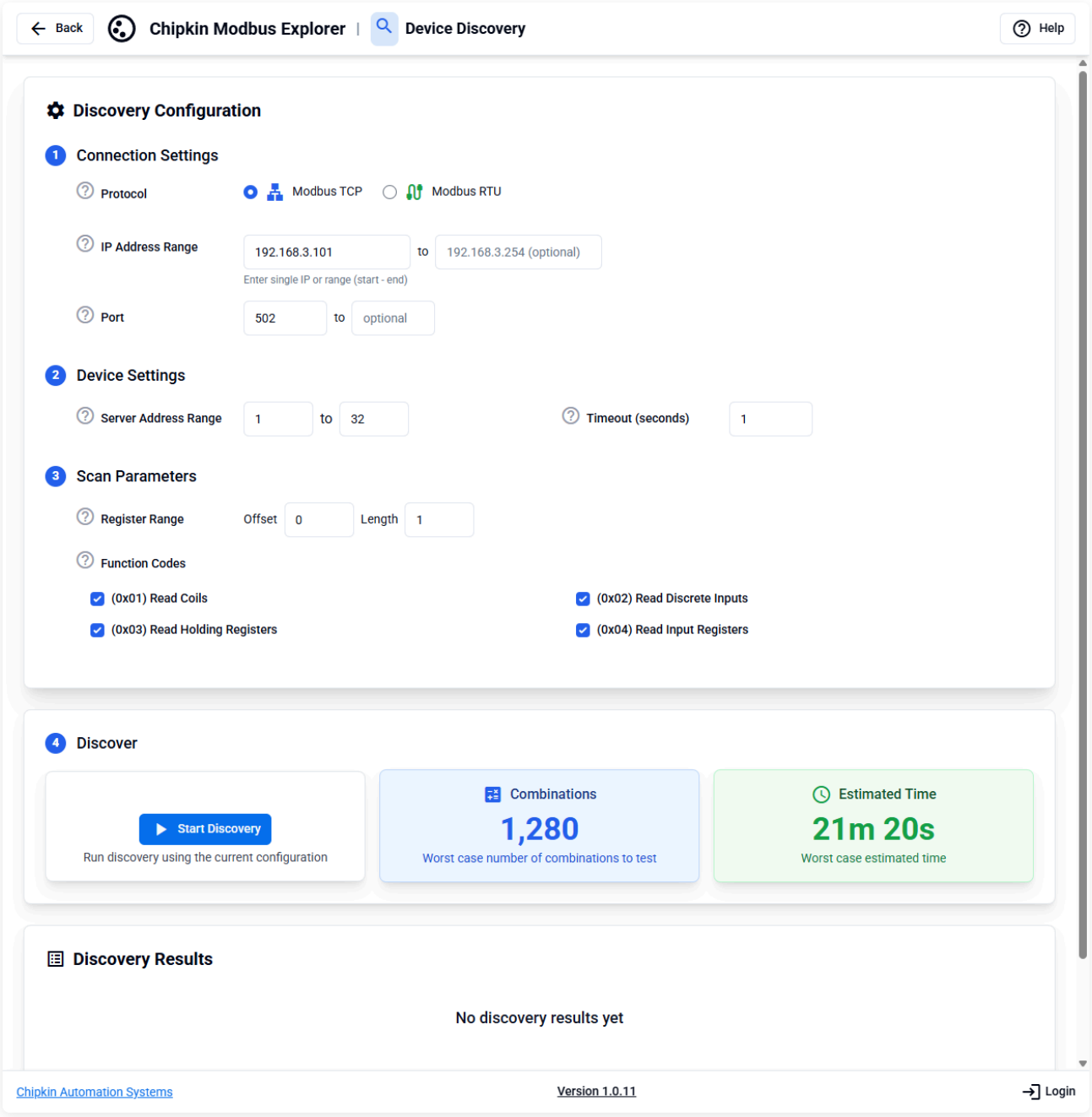
Related Guides

- **Getting Started**: The essentials of Modbus, including the addressing and byte order cheatsheet.
- **Discovery**: Automatically find registers on an unknown device.
- **Simulator**: Create a virtual Modbus device for safe testing.
- **Modbus Protocol**: A reference for function codes and message structures.
- **Troubleshooting Section**: Your first stop for any problem.

Device Discovery

Device Discovery is your scanner for the Modbus world. Use it to automatically find devices, map out their registers, and figure out what a device can do, especially when you have no documentation.

It's the first tool to use when you're faced with an unknown device or a new network.



What Problems Does It Solve?

- **"What devices are on this network?"** - Scan an IP range to find all responsive Modbus TCP devices.
- **"Which Unit IDs are active on this serial line?"** - Scan a serial port to find all connected RTU devices.
- **"I have no documentation. What registers does this device have?"** - Scan a device to create a complete map of its readable registers.
- **"Is this device even a Modbus device?"** - A successful discovery scan provides definitive proof.
- **"I need to create a device template."** - Discover all registers, then export them as a CSV to create a reusable template for the Modbus Client.

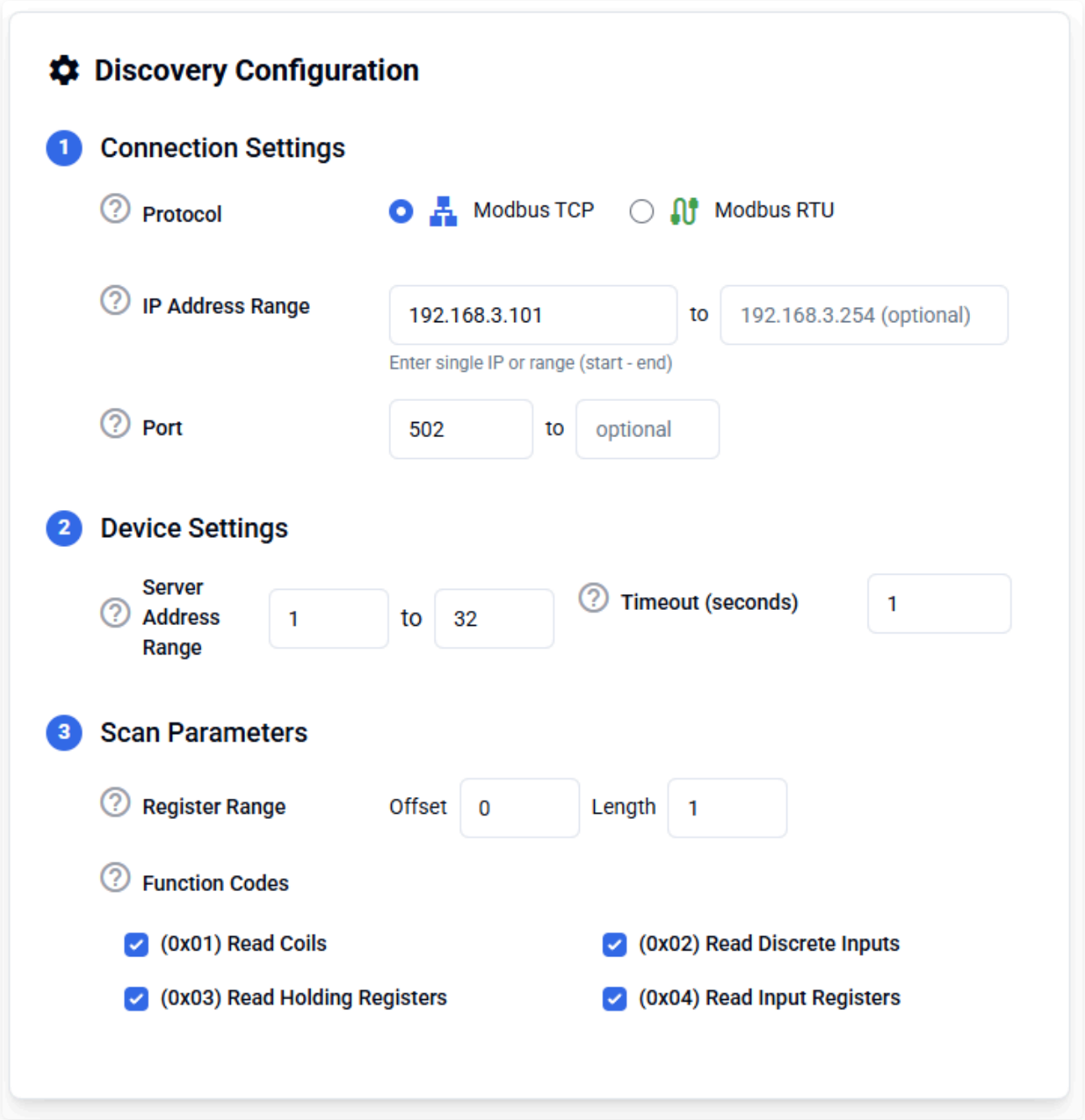
Quick Start: Find All Registers on a New Device

Let's solve the most common problem: you have a new device on your network at **192.168.1.120**, but you have no idea what registers it has.

Scenario: A Modbus TCP device is connected to your network. You need to find all its holding registers (`4xxxx` addresses).

Step 1: Set Connection Details

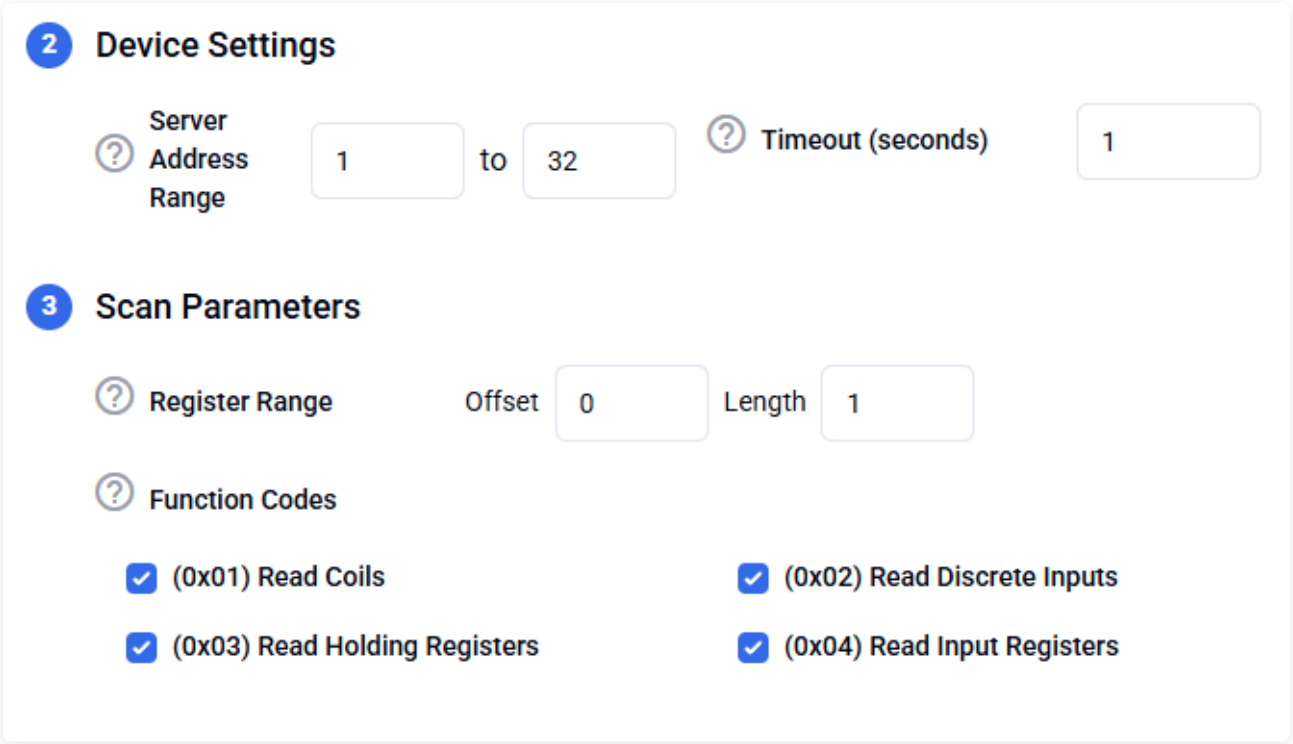
- **Protocol**: TCP
- **IP Address**: `192.168.1.120` (we'll scan just this one device)
- **Port**: `502` (the standard)
- **Unit ID**: `1` (a safe default for a single device)



Step 2: Configure the Scan Parameters

This tells Discovery what to look for.

- **Function Codes**: Select `03 Read Holding Registers`. This is the most common type.
- **Address Range**: Set it to `0` to `100`. Most devices put their most important data in the first 100 registers. This keeps the scan fast.
- **Timeout**: Leave it at `1000` ms. This is a safe default.



Step 3: Start the Scan and Watch the Results

- Click the **Start Discovery** button.
- The progress bar will move, and results will appear in the table in real-time.

You are looking for rows with a **Status** of ✅ `OK`.

- ✅ `OK`: Success! A register was found at this address.
- ⚠️ `Exception`: The device responded, but it doesn't have a register at this address. This is normal and helps you find the boundaries of the register map.
- ❌ `Timeout`: The device didn't respond at all. If you get all timeouts, you have a connection problem. See Troubleshooting below.

Step 4: Analyze and Use the Results

Once the scan is complete, you'll have a list of all holding registers found between addresses 0 and 100.

- **Review the `Value` column**: Do the numbers look like temperatures, pressures, or status codes? This gives you clues about what each register does.
- **Click "Send to Modbus Client"**: This is the magic button. It takes all the discovered registers and automatically sets them up in the Modbus Client for you.
- **Continue in the Modbus Client**: Now you can experiment with different data types (e.g., `FLOAT32`, `INT32`) and byte orders to interpret the data correctly.

You have successfully mapped an unknown device.

How Discovery Works

Discovery is a brute-force testing tool, but a smart one. It systematically sends a read request for every combination of settings you provide.

The Process:

1. **You define the scope**:
    - Which devices to test (IP range or Unit ID range).
    - Which function codes to try (`03` , `04` , etc.).
    - Which addresses to scan (`0-100` , `0-9999` , etc.).
2. **Discovery builds a test plan**: It creates a list of every possible request.
3. **It executes the plan**: It sends requests one by one (or in parallel for TCP).
4. **It records the outcome**: `OK` , `Exception` , or `Timeout` .
5. **It displays the results**: You get a complete map of what responded.

The tool includes optimizations like running TCP requests in parallel to dramatically speed up network scans.

Discovery Configuration In-Depth

Connection: TCP vs. RTU

First, choose your protocol.

- **TCP**: For devices on an Ethernet network. **Much faster** because it can scan multiple IPs at once.
- **RTU**: For devices on a serial line (RS-232/RS-485). **Slower** because it must test devices one at a time.

Scan Scope: Who to Scan

This defines the devices you want to test.

For TCP:

Field	Description	Example
IP Address	The network address(es) to scan. Supports ranges, CIDR, and lists.	`192.168.1.1-192.168.1.254`
Port	The TCP port(s) to test. `502` is standard.	`502` or `502, 1502`
Unit ID	The device address. For TCP, usually `1` or `255` . For gateways, use a range.	`1` or `1-10`

For RTU:

Field	Description	Example
COM Port	The serial port on your computer.	`COM3`

Baud Rate	Communication speed. **Must match the device.**	`9600` , `19200`
Parity	Error checking. **Must match the device.**	`None` , `Even`
Unit ID	The range of device addresses to scan on the serial bus.	`1-10`

> ***Pro Tip**: For RTU, always start with a small Unit ID range like `1-10` . Scanning the full `1-247` range is very slow.*

Scan Parameters: What to Look For

This defines the register data you want to find.

Field	Description	Recommendation
Function Codes	The read operations to perform. `03` and `04` are the most common.	Start with `03` , then add `04` .
Address Range	The register addresses to scan. The tool uses **Protocol Addresses** (0-based).	Start with `0-100` for a quick scan.
Timeout (ms)	How long to wait for a response.	`1000` is a good balance.
Concurrent (TCP)	How many TCP connections to run in parallel. Higher is faster.	`20` for a local network. `5` for a slow one.

Analyzing the Results

The results table tells you everything Discovery found.

Column	Description
Device	The IP address or COM port of the device.
Unit ID	The address of the device that responded.
Address	The 0-based protocol address that was found.
Public Address	The 1-based address you see in manuals (e.g., `40101`).
Value	The raw value read from the register. This is your starting point for interpreting the data.
Status	✅ `OK` (found), ⚠️ `Exception` (address doesn't exist), or ❌ `Timeout` (no response).

From Discovery to Action

1. **Filter for Success**: Use the table filters to show only rows with `Status = OK` . This is your list of valid registers.
2. **Look for Patterns**:
    - Are the addresses in a solid block (e.g., `100` to `150`)? This often means related data.
    - Are there two registers together that look like a floating-point number?
3. **Send to Modbus Client**: This is the most important step. It saves you from manually typing every address.
4. **Refine in the Client**: In the Modbus Client, you can now:

- Test different **Data Types** (`FLOAT32` , `INT32` , etc.).
- Fix garbled values by changing the **Byte Order**.
- Add descriptions to each register.

5. **Export as a Template**: Once you have the register map perfected in the Modbus Client, **Export** it as a CSV. You now have a reusable template for that device model.

Troubleshooting

Problem: The scan finds nothing (all timeouts).

This is a connection issue.

For TCP:

1. **Can you `ping` the device?** Open a command prompt and type `ping 192.168.1.120` . If you get no reply, the device is not on the network or your PC can't reach it.
2. **Is a firewall blocking the port?** Windows Firewall or a network firewall might be blocking port `502` .
3. **Is the IP address correct?** Double-check for typos.
4. **Is the device powered on?**

For RTU:

1. **Are the serial settings correct?** Baud rate, parity, data bits, and stop bits **must exactly match** the device's configuration. The most common setting is **9600 8-N-1**.
2. **Is the COM port correct?** Check in Windows Device Manager to see which COM port your USB-to-Serial adapter is using.
3. **Are the wires swapped?** For RS-485, try swapping the A/B (or +/-) wires.
4. **Is another program using the port?** Close any other serial communication tools.

Problem: The scan is taking forever.

You've made the scope too large. Click **Cancel** and reduce your settings.

1. **Address Range**: Change `0-9999` to `0-100` .
2. **Unit ID Range (RTU)**: Change `1-247` to `1-10` .
3. **Function Codes**: Scan for one at a time, starting with `03` .
4. **Concurrent (TCP)**: If your network is slow, a high number of concurrent connections can actually slow things down. Try reducing it from `50` to `10` .

Problem: The app crashes or the device stops responding during a scan.

The device can't handle the rapid-fire requests.

1. **Reduce Concurrent Connections (TCP)**: Lower it to `5` or even `1` .
2. **Add a Delay (RTU)**: Set a small delay (e.g., `50` ms) between requests to give the device time to breathe.

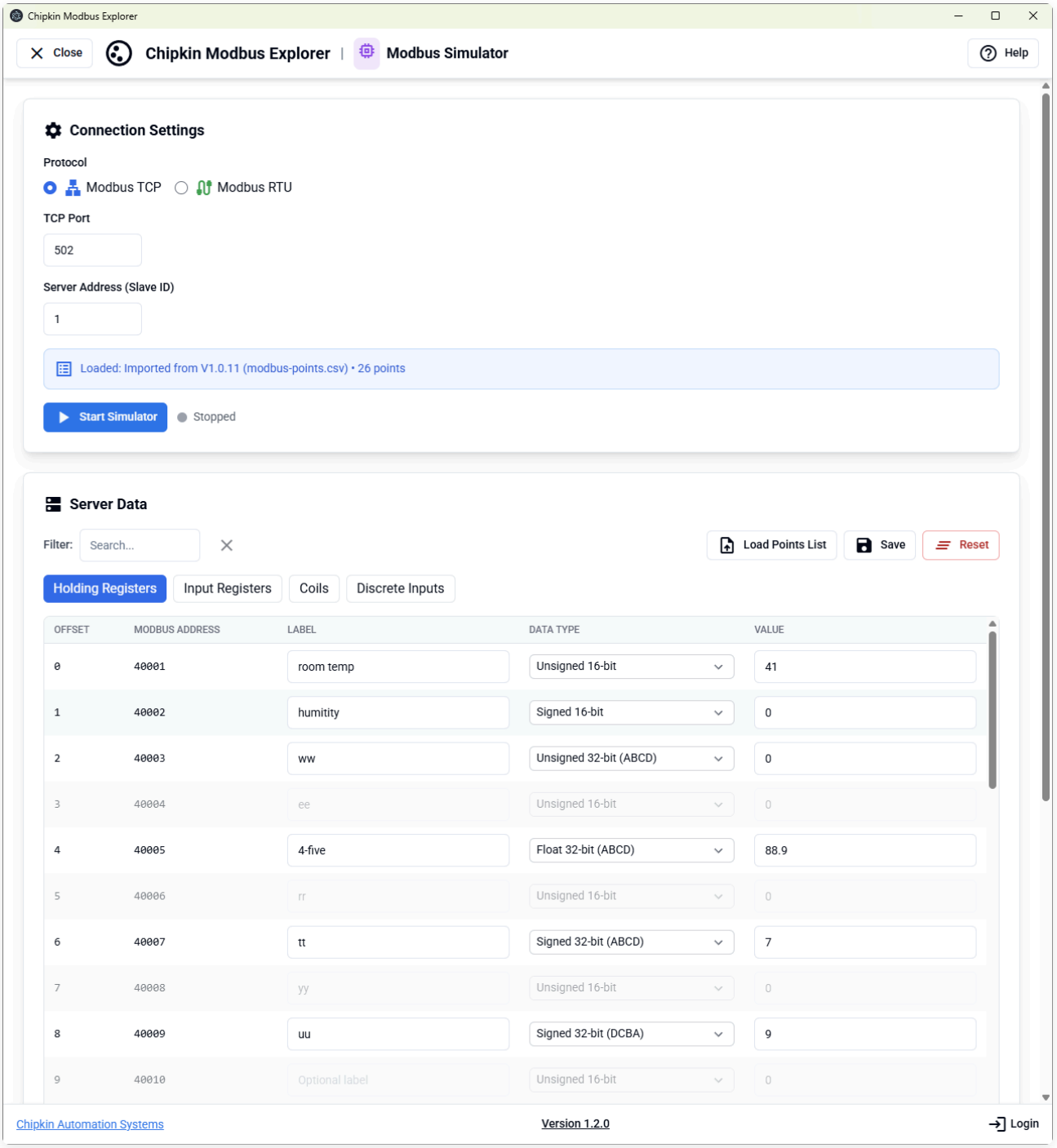
Related Guides

- **Modbus Client**: Where you'll analyze the data you discover.
- **Getting Started**: For a refresher on Modbus basics like addressing.
- **Troubleshooting Section**: For a deep dive into connection and data issues.

Modbus Simulator

The Modbus Simulator creates a virtual Modbus device on your computer. It's a powerful tool for developing and testing Modbus client applications (like an HMI or SCADA system) without needing any physical hardware.

Think of it as a "stunt double" for a real Modbus device. You can't break it, it does exactly what you tell it to, and it's always available.



What Problems Does It Solve?

- **"I need to develop an HMI, but the hardware won't arrive for weeks."** - Build your entire client application against the simulator. When the real device arrives, just change the IP address.
- **"How can I test if my client application handles Modbus errors correctly?"** - Configure the simulator to send specific exception codes to see how your app reacts.
- **"I need to train a new technician on Modbus, but I don't want them breaking expensive equipment."** - Let them practice reading and writing data in a safe, virtual environment.

- **"My client can't talk to a device in the field. Is the problem my app or the device?"** - Test your client against the simulator. If it works, the problem is likely with the field device or network.
- **"I need to demonstrate my application to a customer, but I'm not at the job site."** - Run the simulator on your laptop to create a live, interactive demo.

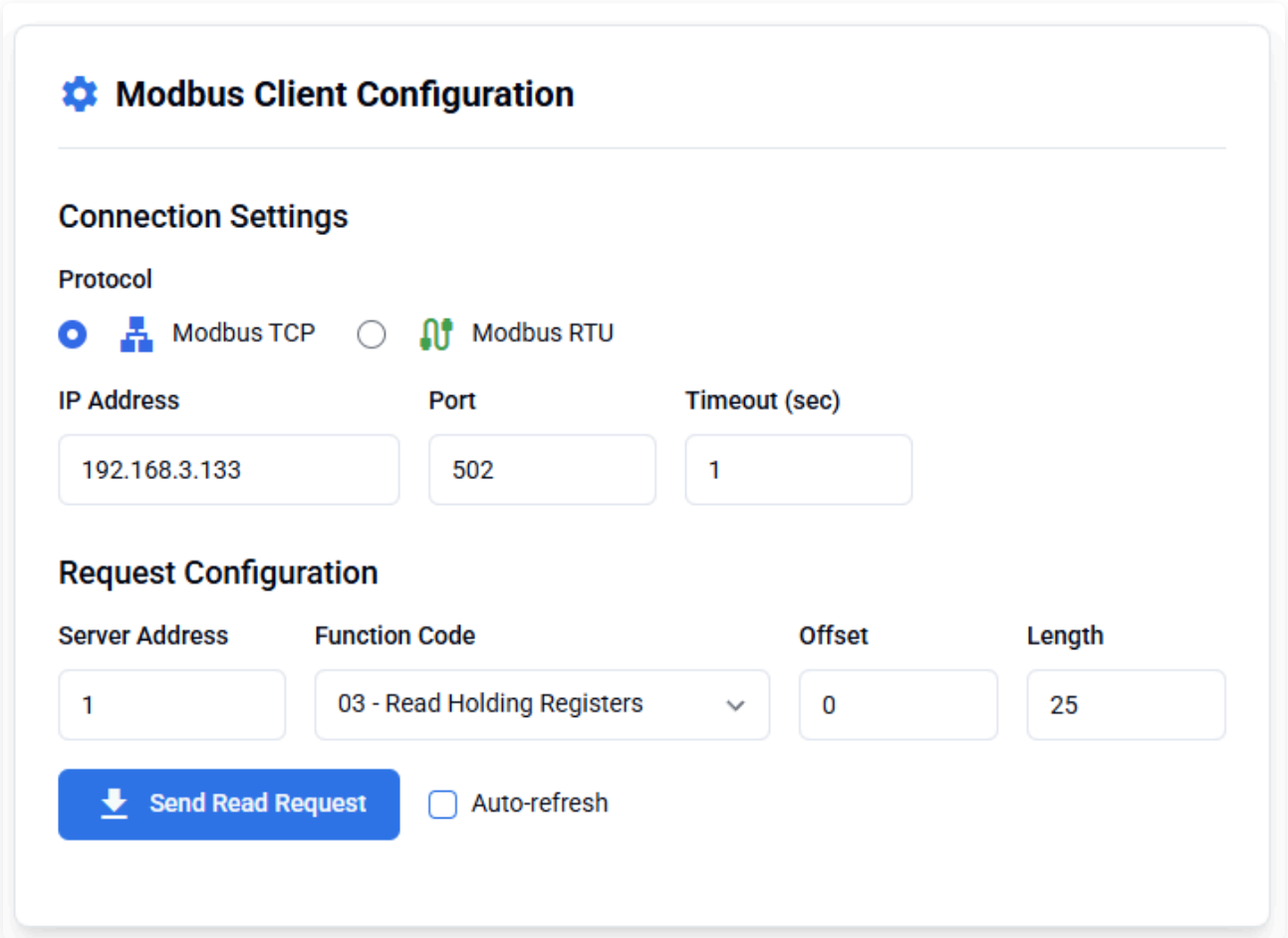
Quick Start: Create a Virtual Temperature Sensor in 4 Steps

Let's simulate a simple Modbus TCP device with a single temperature sensor.

Scenario: You want to create a virtual device that has a temperature reading at Holding Register `40101`.

Step 1: Set Connection Details

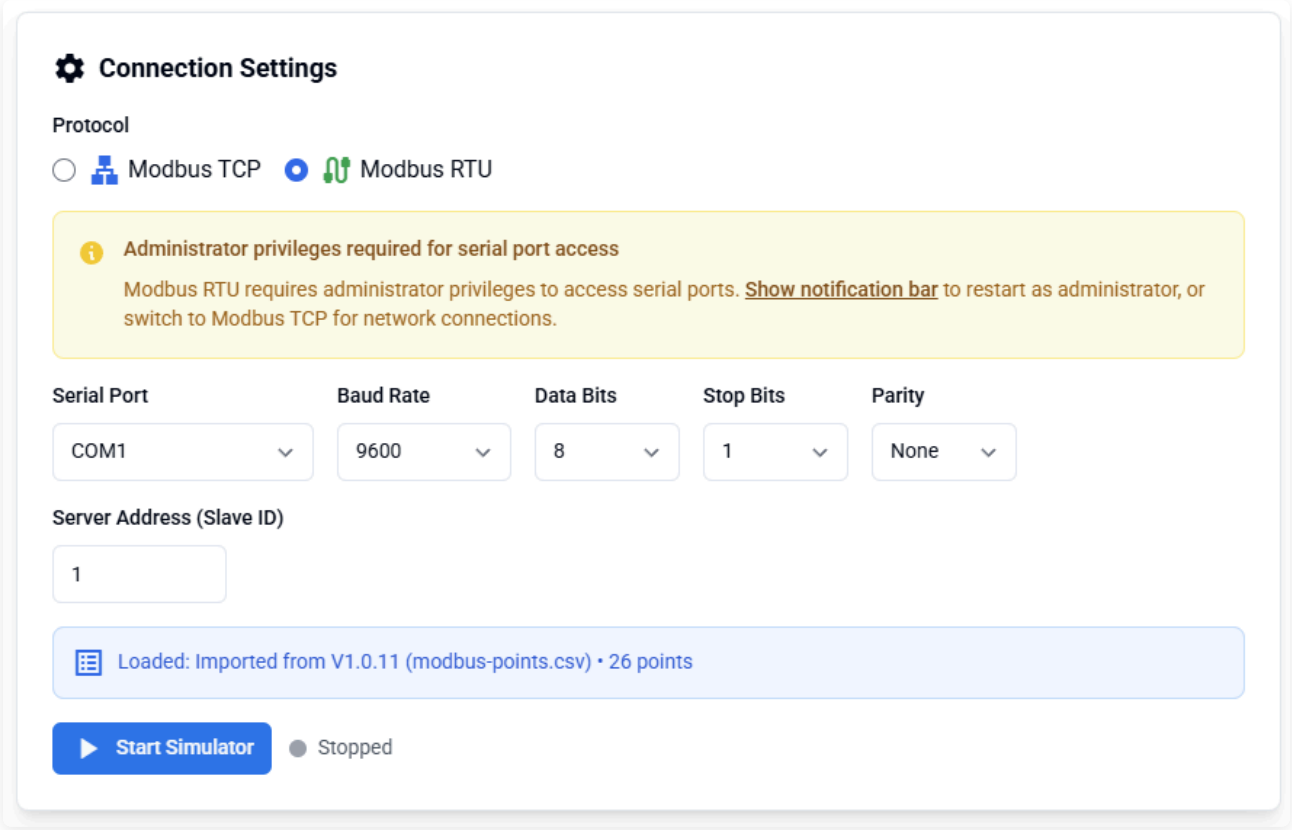
- **Protocol**: TCP
- **Listen Port**: `502` (the standard). If you get a permission error, use a port above 1024, like `1502`.
- **Unit ID**: `1`



Step 2: Define the Virtual Register

This is where you create the data for your virtual device.

- In the **Holding Registers** table, click **Add Row**.
- **Address**: `100` (This is the protocol address for `40101`. See our addressing guide).
- **Data Type**: `FLOAT32`. Temperatures are rarely whole numbers.
- **Value**: `72.5`. This is the starting temperature.
- **Name**: "Room Temperature".



Step 3: Start the Simulator

- Click the **Start Simulator** button.
- The status indicator should turn **Green**.
- **Not green?** The most common issue is another application using port `502`. Try a different port like `1502` or `5020`.

Your virtual device is now running and waiting for a client to connect.

Step 4: Connect to it with the Modbus Client

1. Go to the **Modbus Client** tab.

2. Set the connection to **TCP** and the IP Address to `127.0.0.1` (which means "this computer").
3. Use the same **Port** and **Unit ID** you configured in the simulator.
4. Click **Connect**.
5. Add a row to read Holding Register `100` with a data type of `FLOAT32`.
6. Click **Read**.

Success! The Modbus Client will show the value `72.5`. You have successfully created a virtual device and communicated with it. You can even write a new value from the client, and you will see it update in the simulator's table.

How the Simulator Works

The simulator opens a network port (for TCP) or a serial port (for RTU) on your computer and listens for incoming Modbus requests. When it receives a request, it looks up the address in its internal register tables and sends back a response, just like a real device would.

- **For a `Read` request**: It sends the value from its table.
- **For a `Write` request**: It updates the value in its table and sends a success response.
- **For a request to an unknown address**: It sends an `Illegal Data Address` exception.

This allows you to simulate a complete Modbus device with thousands of data points, all running locally on your machine.

Simulator Configuration In-Depth

Connection: TCP vs. RTU

- **TCP**: The easiest and most common choice. It simulates a network device. Use this unless you have a specific need to test serial communication.
- **RTU**: Simulates a serial device. This requires a virtual COM port pair (e.g., using `com0com`) to connect a client. Use this only when you need to test RTU-specific timing or wiring scenarios.

Register and Coil Tables

The simulator provides four tables, one for each type of Modbus data.

Table	Public Address Range	Read/Write?	Use For
Coils	`0xxxx`	Read/Write	Simulating digital outputs (relays, lights).
Discrete Inputs	`1xxxx`	Read-Only	Simulating digital inputs (switches, alarms).
Input Registers	`3xxxx`	Read-Only	Simulating analog sensors (temperature, pressure).
Holding Registers	`4xxxx`	Read/Write	Simulating setpoints and configuration values.

Creating Your Device Profile (Import/Export)

Manually adding hundreds of registers is tedious. The **Import** and **Export** features are your best friends.

- **Export**: Saves the current set of all registers and coils to a CSV file. Use this to create a reusable template of a device.

- **Import**: Loads a register map from a CSV file, instantly configuring your simulator.

Pro Workflow: Create a Reusable Device Template

1. Configure the simulator with all the registers for a specific device model (e.g., a VFD or a power meter).
2. **Export** the configuration to a file named `[DeviceModel]_template.csv`.
3. The next time you need to test against that model, simply **Import** the template to load the entire configuration in seconds.

CSV File Format

The CSV file is simple. At a minimum, it needs `Address` and `DataType`.

```
Address,Name,DataType,Value,ByteOrder
100,Supply Temp,FLOAT32,72.5,ABCD
102,Return Temp,FLOAT32,68.0,ABCD
200,Setpoint,FLOAT32,75.0,ABCD
0,Fan Enable,BIT,1
```

Advanced Simulation Techniques

Simulating Errors and Exceptions

This is a key feature for testing the robustness of your client application.

How to Simulate an "Illegal Data Address" (Exception 02):

1. In the simulator, define registers only from address `0` to `100`.
2. In your client, try to read address `200`.
3. The simulator will respond with **Exception 02**.
4. **Verify**: Does your client handle this gracefully? Does it show an error message without crashing?

How to Simulate a "Slave Device Failure" (Exception 04):

Some simulators allow you to configure a specific register to always return an exception. This is useful for testing how your client handles a critical device fault.

Simulating Dynamic Data

Real-world data changes. A good simulator can mimic this.

- **Manual Updates**: You can click on any value in the simulator's tables and change it while it's running. The next time your client reads that register, it will get the new value. This is great for testing HMI updates.
- **Automatic Changes**: Some simulators support scripting or pre-defined functions (e.g., sine wave, random walk, counter) to make data change automatically over time. This is perfect for testing trend logs and alarm thresholds.

Testing Multiple Devices

The simulator can act as multiple devices at once.

- **TCP**: By responding to multiple **Unit IDs** on the same IP address and port, the simulator can mimic a gateway with several devices behind it.

- **RTU**: It can respond to different Unit IDs on the same serial line.

Simply enter a comma-separated list of IDs in the **Unit ID** field (e.g., `1, 2, 5, 10`).

Troubleshooting

Problem: The simulator won't start.

Cause	Solution
"Port already in use"	Another application (or another instance of the simulator) is using that port. Change to a different port like `1502`.
"Permission denied" (for ports < 1024)	On Windows, low-numbered ports require admin rights. Run the application as an administrator, or use a port > 1024.
Invalid COM Port (RTU)	The selected COM port doesn't exist or is in use. Check Device Manager and close other serial tools.

Problem: A client can't connect to the simulator.

1. **Is the simulator running?** Check for the 🟢 **Green** status.
2. **Firewall**: The Windows Firewall might be blocking the connection. Allow the application or the specific port through the firewall.
3. **Wrong IP Address**: If connecting from the same machine, use `127.0.0.1`. If connecting from another machine, use the full IP address of the computer running the simulator.
4. **Wrong Port or Unit ID**: Double-check that the client's settings exactly match the simulator's.

Problem: The client reads the wrong value.

This is almost always a data type or byte order issue.

1. **Data Type Mismatch**: Is the client trying to read a `FLOAT32` as a `UINT16`? The raw bytes will be misinterpreted.
2. **Byte Order Mismatch**: This is the most common issue for 32-bit and 64-bit values. There are four possible byte orders. Try changing the byte order in the client until the value reads correctly. The simulator lets you set up the same value with all four byte orders to make this easy to test.

Related Guides

- **Modbus Client**: The perfect partner for testing your simulator.
- **Device Discovery**: Use the discovery tool to scan your own simulator.
- **Getting Started**: A refresher on Modbus fundamentals.

Troubleshooting

This section provides solutions to common issues encountered when using the Chipkin Modbus Explorer.

Triage Checklist

Before diving into detailed troubleshooting, quickly review these common points of failure:

- [ ] **Physical Connection**: Are all cables securely connected? Is the device powered on?
- [ ] **Connection Parameters**: Do the IP address (for TCP) or COM port settings (for RTU) match the device?
- [ ] **Protocol Settings**: Do the baud rate, parity, data bits, and stop bits exactly match the device's configuration?
- [ ] **Unit ID**: Does the slave/unit ID in the explorer match the device's configured ID?
- [ ] **Addressing**: Are you using **0-based protocol addresses** (e.g., `107`) instead of 1-based public addresses (e.g., `40108`)?
- [ ] **Function Code**: Is the function code you are using supported by the device?
- [ ] **Address Validity**: Does the register address you are trying to access actually exist on the device?
- [ ] **Data Format**: Is the byte/word order correct for 32-bit or 64-bit data types?

Connection Issues

Modbus TCP Connection Failed

Symptoms: "Connection refused," "Connection timeout," or the application fails to connect to a network device.

Solutions:

1. **Verify Network Reachability**: Use the `ping` command to check if the device is responsive on the network.

   ```
   ping 192.168.1.100
   ```

   If the ping fails, investigate the device's power, network cable, and IP address.

2. **Check the Port**: The standard Modbus TCP port is `502`. Some devices may use a custom port. Verify the correct port in the device's documentation. Use a tool like `Test-NetConnection` in PowerShell to check if the port is open.

   ```
   Test-NetConnection -ComputerName 192.168.1.100 -Port 502
   ```

3. **Firewall Rules**: A firewall on your computer or network may be blocking the connection. Temporarily disable the firewall to test, and if successful, create an inbound rule to allow traffic on the Modbus port.

4. **Unit ID**: For TCP, the Unit ID is often `1` or `255`. Some devices are strict about this, while others are not. Try both if you are unsure.

Modbus RTU (Serial) Connection Failed

Symptoms: "Port already in use," "Access denied," or failure to connect to a serial device.

Solutions:

1. **Verify COM Port**: Ensure you have selected the correct COM port. Use the Windows Device Manager to see a list of available serial ports. If you are using a USB-to-serial adapter, make sure the correct drivers are installed.
2. **Match Serial Parameters EXACTLY**: The baud rate, parity, data bits, and stop bits must be identical on both the client and the slave device. The most common configuration is **9600 baud, 8 data bits, no parity, and 1 stop bit (9600, 8, N, 1)**.
3. **Check Physical Wiring**:
    - **RS-232**: This is a point-to-point connection. Ensure the transmit (TX) pin of one device is connected to the receive (RX) pin of the other.
    - **RS-485**: This is a multi-drop bus.
        - **Polarity**: Ensure the A/+ and B/- lines are connected consistently across all devices. Incorrect polarity is a common failure point.
        - **Termination**: The two devices at the physical ends of the RS-485 bus must have a 120Ω termination resistor connected across the A and B lines.
4. **Port in Use**: If you get a "Port already in use" error, another application (like a terminal emulator or another Modbus tool) has control of the COM port. Close the other application and try again.

Exception Responses

Exception responses are messages from the slave device indicating that it could not process the master's request.

Exception 01: Illegal Function

Meaning: The function code sent in the request is not supported by the device.

Solution: Consult the device's documentation to find out which function codes it supports. Stick to the basic function codes (`01` , `02` , `03` , `04` , `05` , `06` , `15` , `16`) if you are unsure.

Exception 02: Illegal Data Address

Meaning: The requested address does not exist on the device. This is the most common exception.

Causes and Solutions:

- **Using Public Address**: You are likely using a 1-based public address (e.g., `40108`) instead of the 0-based protocol address (e.g., `107`). **The Chipkin Modbus Explorer requires 0-based protocol addresses.**
- **Address Out of Range**: The address is simply not within the device's memory map. Check the device manual for the valid address ranges.
- **Requesting Too Many Registers**: A request for a block of registers that extends beyond the valid memory range will fail. For example, if a device has registers `0-99` , a request for 10 registers starting at address `95` will fail because it attempts to read up to address `104` .

Exception 03: Illegal Data Value

Meaning: A value in the data field of the request is invalid.

Solution: This is often caused by an incorrectly formatted request, such as a byte count that does not match the quantity of registers. This is rare when using a tool like the Chipkin Modbus Explorer but can occur if the request is manually crafted.

Data and Display Issues

Garbled or Incorrect Values

Symptom: You receive a response, but the values are nonsensical (e.g., a temperature of `3.45e-41`).

Cause: This is almost always a **byte or word order (endianness)** issue when reading 32-bit or 64-bit values.

Solution: The Chipkin Modbus Explorer automatically interprets multi-register values in all four common byte order combinations. Simply look at the **32-bit Data Types** or **64-bit Data Types** view in the results section and find the interpretation that matches the expected value. This will tell you the correct byte order for the device.

Values Off by a Factor of 10, 100, or 1000

Symptom: The value is numerically correct but seems to be missing a decimal point (e.g., you read `255` but expect `25.5`).

Cause: The Modbus protocol itself does not handle floating-point numbers or scaling. It is common for devices to transmit a value as an integer that must be multiplied or divided by a factor (usually a power of 10) to get the actual value.

Solution: **Consult the device manual** for the correct scaling factor. The Chipkin Modbus Explorer displays the raw integer value; you must apply the scaling factor yourself.

Triage Checklist: First Steps in Troubleshooting

When you encounter a problem with Modbus communication, it's tempting to suspect a complex protocol issue. However, the vast majority of problems are caused by simple, easy-to-fix issues. This checklist is your first line of defense. Before you dive into deep diagnostics, quickly run through these steps to rule out the most common culprits.

1. Is Everything Plugged In and Powered On?

It sounds obvious, but it's the most frequent cause of failure.

- **Slave Device**: Ensure the Modbus slave device (your sensor, actuator, PLC, etc.) is powered on. Look for status lights.
- **Cabling**:
    - **For Modbus TCP**: Is the Ethernet cable securely plugged into both the device and your network switch or computer? Are there link lights on the Ethernet ports?
    - **For Modbus RTU (Serial)**: Is the RS-485 or RS-232 cable firmly connected to both the device and your USB-to-Serial adapter?

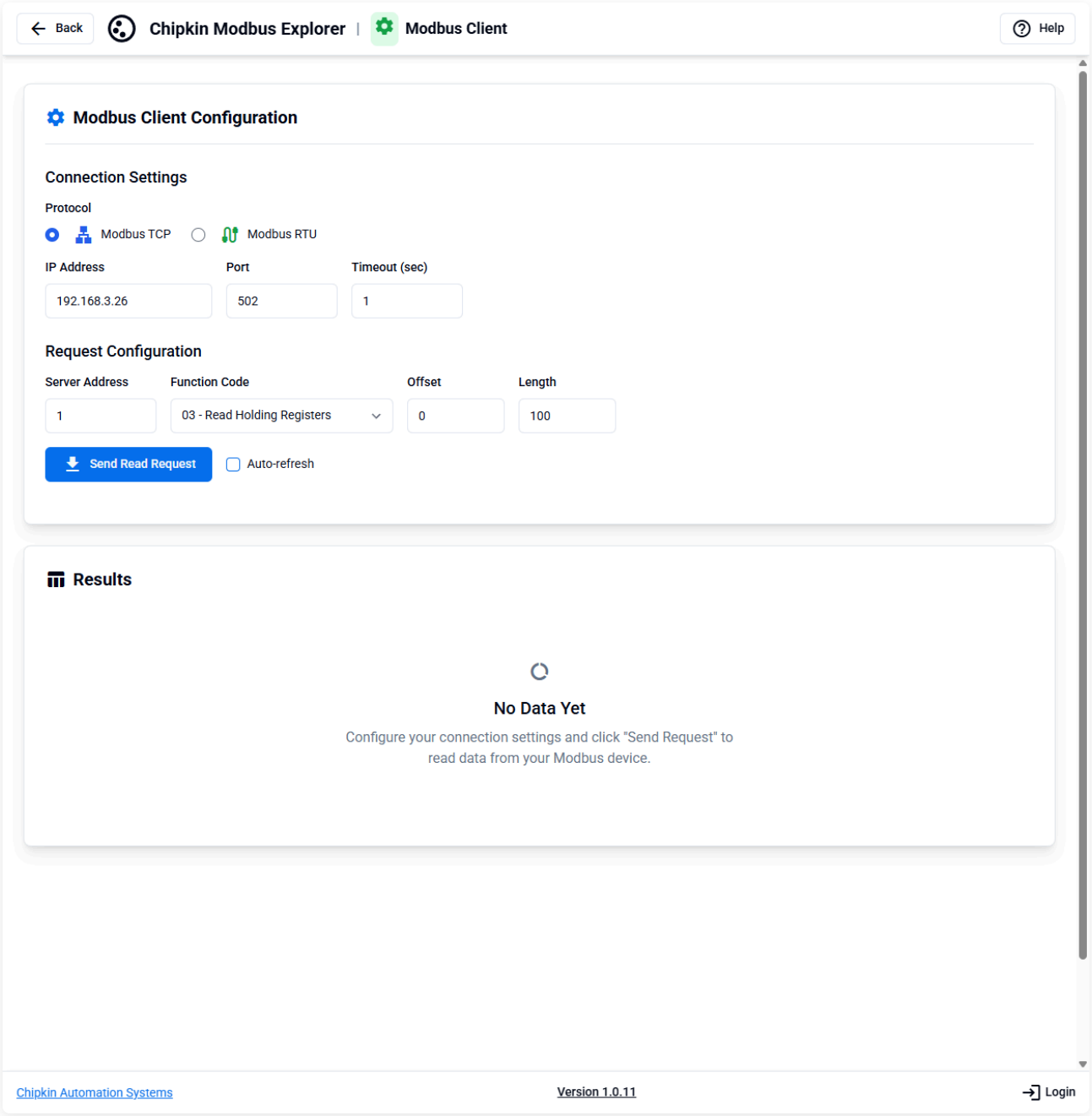
2. Are You Using the Correct Settings in Modbus Explorer?

The Chipkin Modbus Explorer needs to know how to contact your device. Double-check the fundamental connection settings.

- **Correct Mode**: Have you selected the right communication mode?
    - Use **TCP** for network devices.
    - Use **RTU** for serial devices (RS-232 or RS-485).
- **Correct Address/Port**:
    - **For TCP**: Is the **IP Address** and **Port** (usually 502) correct? You can find this in the device's manual or network configuration.
    - **For RTU**: Have you selected the correct **COM Port** from the dropdown list? If you're unsure which COM port your adapter is using, you can check in the Windows Device Manager under "Ports (COM & LPT)".
- **Correct Slave ID**:
    - Every Modbus slave device on a bus needs a unique address, called a Slave ID (or Unit ID). This is a number from 1 to 247.
    - **This is the single most common point of failure.** Ensure the Slave ID in the Modbus Explorer matches the ID configured on the physical device.

Tool Spotlight: The Connection Panel

All of these critical settings are located in the main connection panel of the Modbus Client tab in the Chipkin Modbus Explorer.



Before proceeding to more advanced troubleshooting, make sure every setting in this panel has been verified.

Next Steps

If you have confirmed all the points on this checklist and still cannot communicate with your device, your next step depends on the symptom:

- **If you are getting no response at all (a timeout error)**: Proceed to Troubleshooting Timeout Errors.
- **If you are getting an error message back from the device**: Proceed to Decoding Modbus Exceptions.
- **If you are getting a response, but the data looks wrong**: Proceed to Investigating Garbled Data.

Troubleshooting Timeout Errors

A "timeout" is the most common error in Modbus. It means the master (Chipkin Modbus Explorer) sent a request, but the slave device never sent a reply. The master waited for a specified amount of time and then gave up.

This is "The Silent Treatment." Your device isn't even saying "no"—it's saying nothing at all. This almost always points to a fundamental problem with the physical connection or the core addressing.

Diagnosing Timeouts on Modbus RTU (Serial)

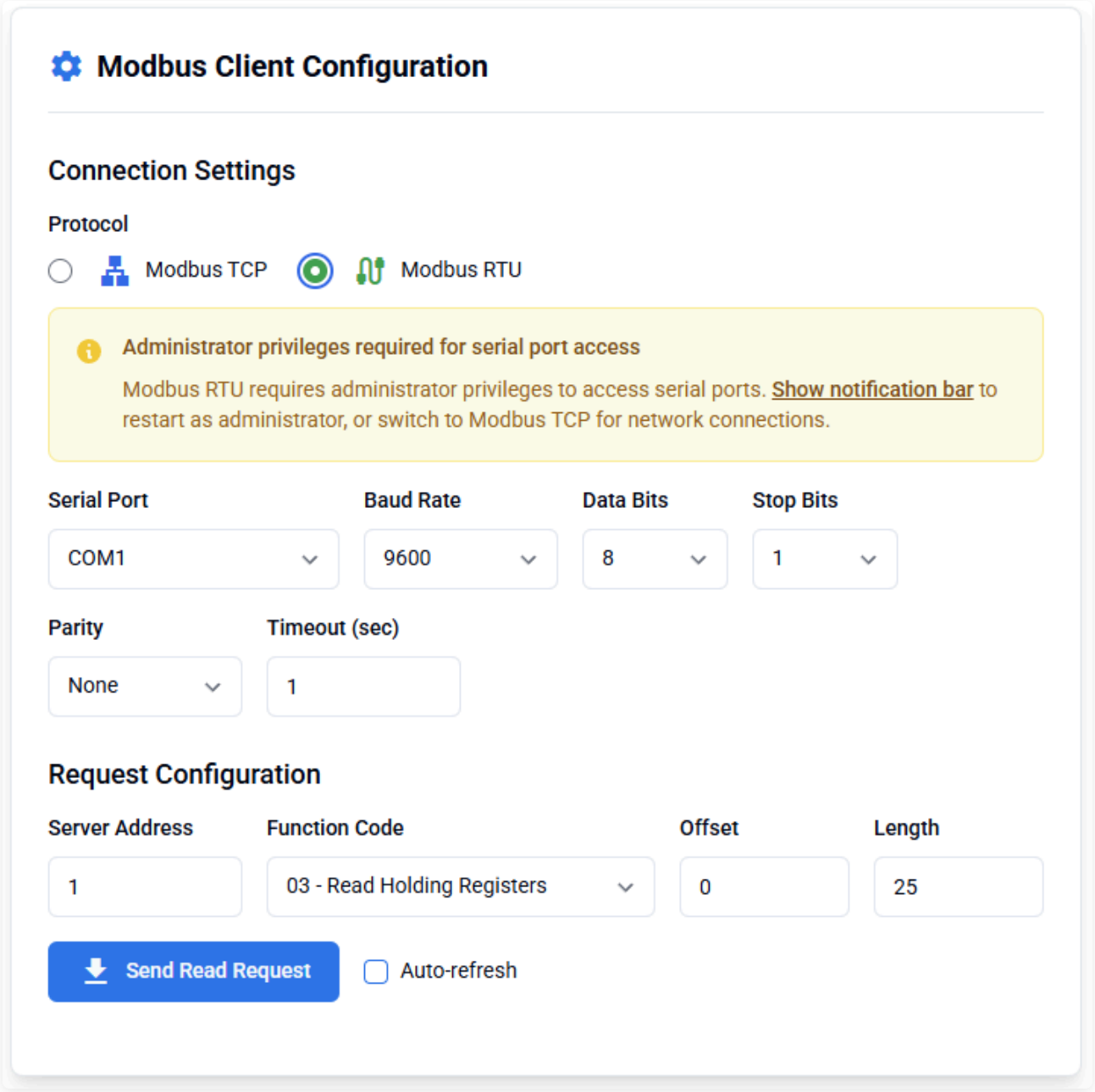
Serial communication is prone to physical layer issues. If you're getting timeouts, the problem is likely in the wiring or the COM port settings.

1. Incorrect COM Port Settings

For two serial devices to talk, they must agree on the "language" of the electrical signals. If their settings don't match, the receiving device will not understand the message.

- **Symptoms**:
    - Consistent timeout errors.
    - In the Message Log, you might see your request (TX) but no response (RX).
    - Occasionally, you might see a response, but it's garbled or has a CRC error.

- **Solution**:
    1. Find the device manual for your slave device.
    2. Locate the section on serial communication settings. It will specify the required **Baud Rate**, **Parity**, **Data Bits**, and **Stop Bits**.
    3. In the Chipkin Modbus Explorer, carefully set these parameters to match the device's requirements *exactly*.
    4. **Common Default**: Many devices default to **9600 baud, 8 data bits, no parity, 1 stop bit** (often written as 9600, 8, N, 1).



2. RS-485 Wiring Problems

RS-485 uses a two-wire system (plus a ground reference). It's robust, but sensitive to correct wiring.

- **Symptoms**:
  - Intermittent timeouts, especially if multiple devices are on the same bus.
  - Communication works with one device but fails when you add more.

- **Solutions**:
  - **Check Polarity**: The two wires are typically labeled A and B (or D+ and D-). The 'A' on your adapter must connect to the 'A' on all slave devices, and 'B' must connect to 'B'. If you swap them, communication will fail. If you're unsure, try swapping the two wires.

- **Use Termination Resistors**: In RS-485, the devices at the physical start and end of the bus need a termination resistor (usually 120 Ohm) connected across the A and B terminals. This prevents signal reflections. Without proper termination, the bus can become unreliable, especially on long cable runs or at high baud rates.

3. RS-232 Wiring Problems

RS-232 is simpler as it's point-to-point, but wiring is still a common issue.

- **Symptoms**: Consistent timeouts.
- **Solution**: RS-232 requires at least three wires: Transmit (TX), Receive (RX), and Ground (GND). The TX pin of the master must connect to the RX pin of the slave, and the RX of the master must connect to the TX of the slave. If a direct cable doesn't work, you may need a **null modem adapter**, which performs this crossover for you.

Diagnosing Timeouts on Modbus TCP/IP

Modbus TCP runs over standard computer networks, so troubleshooting involves common network diagnostics.

1. Basic Network Connectivity Issues

- **Symptoms**: The Modbus Explorer shows a "Connection Refused" or "Timeout" error immediately.
- **Solution**: Use the `ping` command to test basic connectivity.
    1. Open a Command Prompt (`cmd.exe`).
    2. Type `ping <device_ip_address>` (e.g., `ping 192.168.1.55`).
    3. **If you get a reply**: The device is on the network and reachable. The problem is likely a firewall or an incorrect Modbus setting.
    4. **If you get "Destination host unreachable" or "Request timed out"**: The device is not on the network, is powered off, or there is a fundamental network configuration problem (e.g., your PC and the device are on different subnets).

2. Firewall Blocking

- **Symptoms**: The device responds to `ping` , but the Modbus Explorer still times out.
- **Solution**: Modbus TCP uses port **502**. Firewalls on your PC or on the network can block this port.
    - **Windows Firewall**: Temporarily disable the Windows Firewall to see if communication starts working. If it does, you need to add an inbound and outbound rule to allow traffic on TCP port 502.
    - **Network Firewall**: Contact your IT department to ensure that port 502 is not being blocked by a network firewall between your computer and the device.

3. Incorrect Slave ID on Gateways

- **Symptoms**: You are connecting to a Modbus TCP-to-RTU gateway. The connection to the gateway itself seems to work, but any request for data times out.
- **Solution**: A gateway acts as a bridge. When you send a Modbus TCP request to it, the gateway forwards that request onto its serial bus. The **Slave ID (or Unit ID)** in the TCP message tells the gateway which serial device to talk to.

- Ensure the Slave ID in the Modbus Explorer matches the ID of the target device on the serial side of the gateway. If you set it to 0 or 255, some gateways may not forward the message correctly.

Next Steps

If you have resolved your timeout errors but are now receiving specific error codes from the device, proceed to Decoding Modbus Exceptions.

Decoding Modbus Exception Responses

Congratulations! If you're on this page, it means you have established successful communication with your slave device. The "Silent Treatment" is over. Your device is now talking back.

However, it's not sending you data. Instead, it's sending you a specific **Exception Response**. This is a message with a special error code that tells you *why* it cannot fulfill your request. Understanding these codes is the key to solving the next layer of Modbus problems.

Tool Spotlight: The Message Log

The Chipkin Modbus Explorer's **Message Log** is your best friend for diagnosing exceptions. A normal request/response pair looks like this:

- `TX: 01 03 00 00 00 01 84 0A` (Request to read 1 register)
- `RX: 01 03 02 00 0A F8 44` (Response with the data)

An exception response is different. The function code in the response will be the original function code plus 128 (0x80).

- `TX: 01 03 00 00 00 01 84 0A` (Request to read 1 register)
- `RX: 01 83 02 41 30` (Exception Response)

Notice the `83` in the response. This is `03` (Read Holding Registers) + `128` (`0x80`). The next byte, `02` , is the **Exception Code**.

Common Exception Codes and How to Fix Them

Here are the most common exception codes you will encounter and what they mean.

Exception Code 01: Illegal Function

- **What it means**: You have asked the device to perform an action it does not support. For example, you sent a "Write Multiple Coils" (Function Code 15) request to a device that only has registers.
- **How to Fix**:
    1. **Consult the Device Manual**: The device's documentation is the ultimate source of truth. It will contain a "Modbus Register Map" or similar section that lists all supported function codes.

2. **Check Your Action**: Are you trying to write to a read-only value? For example, you can't "write" to an Input Register (Function Code 04). You must use "Read Input Registers."
3. **Cross-reference our protocol documentation**:
    - 01: Read Coils
    - 03: Read Holding Registers
    - 05: Write Single Coil
    - 06: Write Single Register

Exception Code 02: Illegal Data Address

- **What it means**: This is the most common exception. You are trying to read from or write to a register or coil that does not exist on the device.
- **The Cause: 0-based vs. 1-based Addressing**:
    - Device manuals often list registers starting from 1 (e.g., "Holding Register 40001").
    - The Modbus protocol itself addresses registers starting from 0. So, register 40001 is actually at **protocol address 0**.
    - If you try to read register `40001` by requesting address `40001`, you are asking for a register that is far outside the device's memory map, resulting in an "Illegal Data Address" exception.
- **How to Fix**:
    1. **Do the Math**: Remember that the protocol address is `Register Number - Offset`.
        - For Holding Registers (4xxxx), the offset is 40001. So, register 40001 is address 0.
        - For Input Registers (3xxxx), the offset is 30001. So, register 30001 is address 0.
    2. **Use the Chipkin Modbus Explorer**: The Explorer handles this for you! Simply enter the register number (e.g., `40001`) and it will automatically use the correct protocol address (`0`) in the background.
    3. **Probe the Device**: If you're unsure of the exact register map, start at address 0 and try to read one register. If it works, try address 1, and so on, until you get an exception. This can help you discover the valid range of addresses.

Exception Code 03: Illegal Data Value

- **What it means**: The address and function code are valid, but the *data* you sent in a "write" request is not acceptable to the device.
- **How to Fix**:
    - **Check the Device Manual**: The documentation for the specific register may define an allowable range of values. For example, a register for a fan's speed might only accept values from 0-5. Trying to write a value of 10 would cause an "Illegal Data Value" exception.
    - **Check Data Type**: Are you trying to write a value that doesn't make sense? For example, writing the value `2` to a single coil (which can only be `0` for OFF or `1` for ON, represented as `0x0000` or `0xFF00`).

Exception Code 04: Slave Device Failure

- **What it means**: This is a more serious, generic error. The slave device encountered an unrecoverable error while trying to process the request. This is not a problem with your request, but a problem within the slave device itself.
- **How to Fix**:
    - **Power Cycle**: The first step is to turn the slave device off and on again.
    - **Check Device Status**: Look for any fault lights or error messages on the device's physical display.
    - **Consult Manufacturer**: If the problem persists, it may indicate a hardware fault, and you should contact the device manufacturer for support.

Next Steps

If you are now able to read data but the values seem incorrect or nonsensical, it's time to investigate data formatting issues.

- **Proceed to Investigating Garbled Data**.

Investigating Garbled Data

You've established communication and you're getting responses without exceptions. You're in the home stretch! However, the data you're reading doesn't make sense. A temperature sensor is reading `1078024704` or a pressure value is `-2.35e+38`.

This is a classic data interpretation problem. The raw data is being transmitted correctly, but the master (Chipkin Modbus Explorer) is not interpreting the bytes in the way the slave device intended. Modbus itself only transmits a series of 16-bit registers; it has no concept of a "float" or a "signed integer." The master must know how to reassemble and interpret these registers.

Tool Spotlight: The 32-Bit Data View

The Chipkin Modbus Explorer has a powerful feature specifically for solving these problems. When you read a set of registers, the main view shows you the raw 16-bit values. Below that, the **32-bit Data Types** view shows you what those same registers represent when interpreted as different 32-bit and 64-bit data types.

FLOAT32 (ABCD)	FLOAT32 (DCBA)
3.765255e-39	1.470803e-41
0.0	0.0
88.900002	-4.292711e+8

This view is the key to solving all of the following common data issues.

Common Data Interpretation Problems

1. Wrong Endianness (Byte and Word Swapping)

This is, by far, the most common cause of "garbled" 32-bit values (like floats or large integers).

- **What it is**: A 32-bit value is stored across two 16-bit Modbus registers. There is no universal standard for which order the bytes and words should be in. **Endianness** refers to this ordering.
- **The Symptom**: You read a value you know should be `25.5`, but it appears as a massive, nonsensical number.
- **The Solution**:
    1. Read the two registers that hold your 32-bit value.
    2. Look at the **32-bit Data Types** view in the Modbus Explorer.
    3. The view displays four common byte/word orderings (endian formats) simultaneously:
        - **Big Endian** (ABCD)
        - **Little Endian** (DCBA)
        - **Mid-Big Endian** (BADC)
        - **Mid-Little Endian** (CDAB)
    4. Scan down the list. One of these values will almost certainly be the correct, expected number. This instantly tells you the endianness of your device.

2. Incorrect Data Type

- **What it is**: You are interpreting the registers as the wrong kind of number.
- **The Symptom**: The value looks like a valid number, but it's not what you expect. For example, you expect a floating-point temperature like `25.5`, but you see a large integer like `2550`.
- **The Solution**:

1. Look at the device's Modbus map. It should specify the data type for the register (e.g., "32-bit float", "16-bit signed integer", "32-bit unsigned integer").
2. In the Modbus Explorer's **32-bit Data Types** view, find the row that corresponds to the correct data type.
3. This is common with devices that use scaling factors. They may store the temperature `25.5` as the integer `255` and expect the master to know to divide by 10.

3. Signed vs. Unsigned Integers

- **What it is**: You are reading a negative number, but it's showing up as a very large positive number.
- **The Symptom**: A known negative value, like `-1`, is displayed as `65535` (for a 16-bit integer) or `4294967295` (for a 32-bit integer).
- **The Solution**:
  - This is a classic sign of interpreting a **signed** integer (which can be positive or negative) as an **unsigned** integer (which can only be positive).
  - The Modbus Explorer's data view shows both interpretations. Simply look at the "Signed" value instead of the "Unsigned" one. For example, the raw 16-bit register `0xFFFF` is `65535` when interpreted as unsigned, but `-1` when interpreted as signed.

You've Mastered Modbus

If you've followed the guides this far—from making the initial connection, to decoding exceptions, to correctly interpreting the data—you have successfully troubleshot the vast majority of issues you will ever encounter with Modbus.

Support

This page provides information on how to get help, report issues, and stay updated with the latest versions of the Chipkin Modbus Explorer.

Technical Support

If you encounter any issues or have questions, our support team is here to help.

- **Website**: www.chipkin.com
- **Email**: support@chipkin.com
- **Phone**: +1 (866) 383-1657

Our support hours are Monday to Friday, 9:00 AM to 5:00 PM Pacific Time.

Reporting Bugs

If you believe you have found a bug, please help us by providing as much detail as possible.

1. **Describe the Issue**: Clearly explain what happened and what you expected to happen.
2. **Steps to Reproduce**: Provide a step-by-step account of how to trigger the bug.
3. **Include Logs**: The message log is invaluable for debugging. Copy and paste the relevant portion of the log from the Modbus Client.
4. **Screenshots**: A picture is often worth a thousand words. Include screenshots of the application state if possible.

Please send all bug reports to **support@chipkin.com**.

Feature Requests

We are always looking to improve the Chipkin Modbus Explorer. If you have an idea for a new feature or an improvement to an existing one, please let us know.

- **Email**: Send your suggestions to support@chipkin.com with the subject line "Feature Request".
- **Describe Your Use Case**: Explain the problem you are trying to solve and how the proposed feature would help.

Software Updates

The Chipkin Modbus Explorer includes an auto-update feature to ensure you always have the latest version.

- **Automatic Checks**: The application will automatically check for updates on startup.
- **Manual Checks**: You can manually check for updates by navigating to `Help > Check for Updates` in the application menu.

When an update is available, you will be prompted to download and install it. Release notes for each version are available within the application and on our website.

Part II: Modbus Protocol Reference

In This Section

This section is a protocol-focused appendix you can use as a reference while using Chipkin Modbus Explorer.

Back to the user guide: Part I: Chipkin Modbus Explorer

Modbus Protocol Reference

Complete technical documentation for understanding and implementing the Modbus protocol.

What is Modbus?

Modbus is an industrial communication protocol for connecting electronic devices:

- **Open**: Freely available specification
- **Simple**: Easy to implement
- **Mature**: Industry standard since 1979
- **Versatile**: Works over serial and TCP/IP networks

Originally developed by Modicon (now Schneider Electric) for programmable logic controller (PLC) communication, Modbus has become one of the most widely used industrial protocols worldwide. It is commonly used in SCADA systems, building automation, industrial control, and energy management.

Protocol Reference Sections

Fundamentals

Data Model - Understanding the four tables (coils, discrete inputs, input registers, holding registers)

Message Structure - PDU, ADU, framing, and error checking

Addressing - Public vs protocol addressing, the off-by-one error

Protocol Variants

Modbus TCP/IP

- Transport: Ethernet network (TCP/IP)
- Port: 502 (standard IANA assigned)
- Model: Client/server with multiple client support
- Error Checking: TCP checksums
- See: Modbus Technical Reference - TCP/IP Section

Modbus RTU (Remote Terminal Unit)

- Transport: Serial communication (RS-232/RS-485)
- Model: Master/slave (poll-response)
- Encoding: Binary with CRC-16 error checking
- See: Modbus Technical Reference - RTU Section

Modbus ASCII

- Transport: Serial communication
- Encoding: ASCII characters (human-readable)

- Error Checking: LRC (Longitudinal Redundancy Check)
- Usage: Rare in modern systems, legacy support only

> **Tip — Protocol Variant Compatibility**
>
> *Modbus RTU and Modbus ASCII are **not interoperable**. An RTU master cannot communicate with an ASCII slave. Always verify which variant your device supports.*

Read Functions

Function	Name	Page	Data Type
FC01	Read Coils	Details	Boolean outputs
FC02	Read Discrete Inputs	Details	Boolean inputs
FC03	Read Holding Registers	Details	16-bit words (most common)
FC04	Read Input Registers	Details	16-bit words (read-only)

Write Functions

Function	Name	Page	Operation
FC05	Write Single Coil	Details	Force coil ON/OFF
FC06	Write Single Register	Details	Write 16-bit value
FC15	Write Multiple Coils	Details	Write 1-1968 coils
FC16	Write Multiple Registers	Details	Write 1-123 registers

Diagnostic Functions

Function	Name	Page	Purpose
FC08	Diagnostics	Details	Communication health testing
FC11	Get Comm Event Counter	Details	Message counter (serial only)

Advanced Functions

Function	Name	Page	Purpose
FC23	Read/Write Multiple Registers	Details	Atomic read and write
FC43	Read Device Identification (MEI)	Details	Device vendor/model info

Exception Handling

Exception Overview - How Modbus reports errors

Individual Exception Codes:

Code	Name	Page
01	Illegal Function	Details
02	Illegal Data Address	Details
03	Illegal Data Value	Details
04	Server Device Failure	Details
05	Acknowledge	Details
06	Server Device Busy	Details
08	Memory Parity Error	Details
0A	Gateway Path Unavailable	Details
0B	Gateway Target Failed to Respond	Details

Data Types and Encoding

For information about data types, byte ordering, scaling, and floating-point numbers:

- **Modbus Technical Reference - Data Types**
- **Modbus Technical Reference - Byte Ordering**

Physical Layer and Connection

For information about serial connections (RS-232, RS-485), network topologies, and physical wiring:

- **Modbus Technical Reference - Serial Communications**
- **Modbus Technical Reference - Multi-Drop Networks**

Best Practices

For Implementing a Modbus Stack

If you're building a Modbus stack from scratch, follow this learning path:

1. **Understand the Data Model** - Read the data model page
2. **Learn Message Structure** - Study PDU and ADU formats
3. **Master Addressing** - Understand the off-by-one error
4. **Implement Read Functions** - Start with FC03 (Read Holding Registers)
5. **Add Write Functions** - Implement FC06 (Write Single Register)
6. **Handle Exceptions** - Learn exception response format
7. **Test with Real Devices** - Use Chipkin Modbus Explorer's simulator

8. **Add Advanced Functions** - Implement additional function codes as needed

Quick Reference for Common Tasks

Reading Process Data:

- Use FC03 - Read Holding Registers for most data
- Use FC04 - Read Input Registers for read-only sensor data
- Batch multiple registers in single request (max 125 registers)

Writing Control Values:

- Use FC06 - Write Single Register for single values
- Use FC16 - Write Multiple Registers for atomic updates
- Always read back to confirm critical writes

Controlling Digital Outputs:

- Use FC05 - Write Single Coil for single relay/output
- Use FC15 - Write Multiple Coils for multiple outputs

Troubleshooting Communication:

- Check for exception responses
- Use FC08 - Diagnostics for loopback tests
- Review Troubleshooting Guide

Related Resources

- **Getting Started Guide** - Quick start for using Chipkin Modbus Explorer
- **Modbus Client** - Using the client/master features
- **Modbus Simulator** - Creating virtual Modbus devices
- **Glossary** - Modbus terminology reference

External References

- Modbus Organization Official Site
- Modbus Application Protocol Specification V1.1b3
- Modbus over Serial Line Specification V1.02
- Modbus Messaging Implementation Guide V1.0b

Modbus Addressing

Modbus addressing is one of the most confusing aspects of the protocol. There are **two different addressing schemes** used simultaneously, leading to the notorious "off-by-one error" that trips up both beginners and experienced developers.

The Two Addressing Schemes

Public Address (Display Address)

This is how addresses appear in documentation, user interfaces, and device manuals:

5-Digit Format (Traditional - supports up to 9,999 points per table):

Table	Public Range	Example
Coils	00001 - 09999	00027
Discrete Inputs	10001 - 19999	10105
Input Registers	30001 - 39999	30009
Holding Registers	40001 - 49999	40108

6-Digit Format (Extended - supports up to 65,535 points):

Table	Public Range	Example
Coils	000001 - 065535	000027
Discrete Inputs	100001 - 165535	110105
Input Registers	300001 - 365535	330009
Holding Registers	400001 - 465535	440108

How to Read Public Addresses:

```
Example: 40108

First digit(s): Table identifier
  0 = Coils
  1 = Discrete Inputs
  3 = Input Registers
  4 = Holding Registers


Remaining digits: Point number within table
  0108 = The 108th holding register
```

Protocol Address (Wire Address)

This is what actually goes inside Modbus messages. It's a **0-based offset** from the start of each table:

Table	Protocol Range	Example
Coils	0x0000 - 0xFFFF	0x001A (26)
Discrete Inputs	0x0000 - 0xFFFF	0x0068 (104)
Input Registers	0x0000 - 0xFFFF	0x0008 (8)
Holding Registers	0x0000 - 0xFFFF	0x006B (107)

Key Insight: Protocol addresses are **0-based**, meaning they start at 0, not 1.

The Off-By-One Error

The Problem

Public address and protocol address differ by 1:

```
Public Address:   40108
Table:            Holding Registers (indicated by '4')
Point Number:     108 (the 108th holding register)
Protocol Address: 107 (because first point is 0, not 1)
```

The Conversion:

```
Public → Protocol:  Subtract 1 from the point number
Protocol → Public:  Add 1 to the point number
```

Real-World Examples

Example 1: Holding Register 40001

```
Public Address:   40001
Point Number:     1 (first holding register)
Protocol Address: 0 (0x0000)

Modbus Request to read 40001:
  Function Code: 0x03
  Address: 0x0000 ← NOT 1!
  Quantity: 0x0001
```

Example 2: Holding Register 40108

```
Public Address:   40108
Point Number:     108
Protocol Address: 107 (0x006B)
```

```
Modbus Request to read 40108:
  Function Code: 0x03
  Address: 0x006B ← This is what goes on the wire
  Quantity: 0x0001
```

Example 3: Coil 00027

```
Public Address:   00027
Point Number:     27
Protocol Address: 26 (0x001A)

Modbus Request to read coil 00027:
  Function Code: 0x01
  Address: 0x001A
  Quantity: 0x0001
```

Example 4: Input Register 30009

```
Public Address:   30009
Point Number:     9
Protocol Address: 8 (0x0008)

Modbus Request to read input register 30009:
  Function Code: 0x04
  Address: 0x0008
  Quantity: 0x0001
```

Conversion Process

Public to Protocol Address

Step-by-Step:

1. **Identify the table type** from first digit(s)
2. **Extract the point number** (remaining digits)
3. **Subtract 1** to get protocol address
4. **Select appropriate function code** based on table

Example: Read "40256"

```
Step 1: First digit '4' → Holding Register table
Step 2: Point number = 256
Step 3: Protocol address = 256 - 1 = 255 (0x00FF)
Step 4: Function code = 0x03 (Read Holding Registers)
```

```
Result Request:
  [03][00 FF][00 01]
   FC  Addr   Qty
```

Protocol to Public Address

Step-by-Step:

1. **Function code** determines table type
2. **Add 1** to protocol address
3. **Prepend table identifier**

Example: Response to FC03, address 0x006B

```
Step 1: FC03 → Holding Registers → Prefix '4'
Step 2: 0x006B = 107 decimal → 107 + 1 = 108
Step 3: Public address = 40108
```

Conversion Table Reference

Quick Reference:

Public Address	Table	Point #	Protocol Address (Hex)	Function Code
00001	Coil	1	0x0000	FC01, FC05, FC15
00100	Coil	100	0x0063	FC01, FC05, FC15
10001	Discrete Input	1	0x0000	FC02
10100	Discrete Input	100	0x0063	FC02
30001	Input Register	1	0x0000	FC04
30100	Input Register	100	0x0063	FC04
40001	Holding Register	1	0x0000	FC03, FC06, FC16, FC23
40100	Holding Register	100	0x0063	FC03, FC06, FC16, FC23
40108	Holding Register	108	0x006B	FC03, FC06, FC16, FC23

Why This Confusion Exists

Historical Reasons

Original Modicon PLCs (1970s):

- Physical I/O points numbered starting at 1 (human-friendly)
- Documentation used 1-based numbering

- First relay coil was "Coil 1", first register was "Register 1"

Modbus Protocol Design:

- Computer science convention: Arrays start at 0
- First element has offset 0 from start of memory
- More efficient for processors

Result:

- **User-facing documentation** uses 1-based (public address)
- **Protocol messages** use 0-based (protocol address)
- **Translation required** at the interface

Modern Impact

This dual-addressing persists today because:

- Backwards compatibility with existing documentation
- Industry familiarity with 5/6-digit format
- Standards maintain historical conventions

Common Mistakes

Mistake 1: Using Public Address Directly

Wrong:

```
Want to read: Holding Register 40001
Send: [03][00 01][00 01]
      FC  Addr=1  Qty
```

Result: Reads holding register 40002 instead!

Correct:

```
Want to read: Holding Register 40001
Send: [03][00 00][00 01]
      FC  Addr=0  Qty
```

Mistake 2: Forgetting Table Independence

Wrong Assumption: "Address 0 means the same thing for all function codes"

Reality:

```
FC01, Address 0 → Coil 00001
FC02, Address 0 → Discrete Input 10001
```

```
FC03, Address 0 → Holding Register 40001
FC04, Address 0 → Input Register 30001

All different memory locations!
```

Mistake 3: Off-By-One in Range Calculations

Scenario: Read holding registers 40100-40109 (10 registers)

Wrong:

```
Start: 40100 → Protocol 100
End: 40109 → Protocol 109
Quantity: 109 - 100 = 9 ← WRONG!

Reads only 9 registers (40100-40108)
```

Correct:

```
Start: 40100 → Protocol 99
Quantity: 10 (explicit count)

OR


Start point: 40100 (point #100)
End point: 40109 (point #109)
Quantity: (109 - 100) + 1 = 10 registers
Protocol start: 100 - 1 = 99
```

Using Chipkin Modbus Explorer

Chipkin Modbus Explorer uses **0-based protocol addressing** (what's actually on the wire).

Reading Holding Register 40108

Option 1: Direct Protocol Address

```
1. Function Code: 03 (Read Holding Registers)
2. Starting Address: 107 ← Protocol address
3. Quantity: 1
```

Option 2: Using Conversion (if UI provides it)

Some tools offer conversion calculators:

```
Enter Public Address: 40108
Tool converts to: 107
Automatically sets function code: 03
```

Example: Multiple Register Read

Want to read: 40100 through 40109 (10 registers)

```
Start public: 40100
Start protocol: 100 - 1 = 99
```

```
Function Code: 03
Starting Address: 99
Quantity: 10
```

Response will contain:

```
Byte Count: 20 (10 registers × 2 bytes)
Data: [Register 40100][Register 40101]...[Register 40109]
```

Address Calculation Formulas

For Implementation

Public to Protocol:

```
FUNCTION publicToProtocol(publicAddress):
  addressString = CONVERT_TO_STRING(publicAddress)

  // Determine table from first digit
  firstDigit = GET_FIRST_CHAR(addressString)


  IF firstDigit == '0':
    functionCode = 1 // Coils
  ELSE IF firstDigit == '1':
    functionCode = 2 // Discrete Inputs
  ELSE IF firstDigit == '3':
    functionCode = 4 // Input Registers
  ELSE IF firstDigit == '4':
    functionCode = 3 // Holding Registers
  END IF


  // Extract point number (remaining digits)
  pointNumberString = SUBSTRING(addressString, 1)
  pointNumber = CONVERT_TO_INTEGER(pointNumberString)
```

```
  // Convert to protocol address (0-based)
  protocolAddress = pointNumber - 1


  RETURN { functionCode, protocolAddress }
END FUNCTION


// Example usage:
result = publicToProtocol(40108)
// Returns: { functionCode: 3, address: 107 }
```

Protocol to Public:

```
FUNCTION protocolToPublic(functionCode, protocolAddress):
  // Determine table prefix
  IF functionCode IN (1, 5, 15):
    prefix = '0' // Coils
  ELSE IF functionCode == 2:
    prefix = '1' // Discrete Inputs
  ELSE IF functionCode IN (3, 6, 16, 23):
    prefix = '4' // Holding Registers
  ELSE IF functionCode == 4:
    prefix = '3' // Input Registers
  END IF

  // Convert protocol address to point number (1-based)
  pointNumber = protocolAddress + 1


  // Format as 5 or 6 digit address
  pointString = FORMAT_STRING(pointNumber, "0000") // Pad with leading zeros
  publicAddressString = prefix + pointString
  publicAddress = CONVERT_TO_INTEGER(publicAddressString)


  RETURN publicAddress
END FUNCTION


// Example usage:
publicAddr = protocolToPublic(3, 107)
// Returns: 40108
```

Best Practices

For Documentation

1. **Always specify which addressing you're using**

```
Good: "Read holding register 40108 (protocol address 107)"
Bad:  "Read address 107" (which addressing scheme?)
```

2. **Use public addressing for user documentation**
    - More intuitive for end users
    - Matches device manuals
    - Standard industry practice

3. **Use protocol addressing for implementation**
    - Matches actual wire protocol
    - Easier to validate against captures
    - Less conversion errors

For Implementation

1. **Convert at the boundary**

```
User Input (Public) → Convert → Internal (Protocol) → Wire
Wire → Internal (Protocol) → Convert → Display (Public)
```

2. **Store addresses in protocol format internally**
    - Matches Modbus messages directly
    - Simpler message construction
    - Convert only for display

3. **Validate ranges before conversion**

```
if (address < 40001 || address > 449999) {
  throw new Error("Invalid holding register address");
}
```

4. **Test boundary conditions**
    - First address (40001 → 0)
    - Last address (65535 → 65534)
    - Maximum quantity reads

Related Topics

Data Organization:

- Data Model - The four tables and their characteristics
- Glossary - Address - Terminology definitions

Protocol Details:

- Message Structure - How addresses fit into messages
- FC03 - Read Holding Registers - Address usage example
- FC06 - Write Single Register - Writing to addresses

Error Handling:

- Exception 02 - Illegal Data Address - Invalid address errors

Implementation:

- Modbus Client Guide - Using addresses in Chipkin Modbus Explorer
- Troubleshooting - Addressing problems

Modbus Data Model

Modbus was originally designed for programmable logic controller (PLC) programming and data transfer. It organizes all data into **four separate memory tables**, each with specific characteristics and usage patterns.

The Four Tables

Table Name	Data Type	Access	Addressing	Function Codes	Original Use
Coils (0xxxx)	Boolean	Read/Write	0-65535	01, 05, 15	Binary outputs (relay coils)
Discrete Inputs (1xxxx)	Boolean	Read-only	0-65535	02	Binary inputs
Input Registers (3xxxx)	16-bit word	Read-only	0-65535	04	Analog inputs
Holding Registers (4xxxx)	16-bit word	Read/Write	0-65535	03, 06, 16, 23	Analog outputs, configuration

Historical Context

Understanding why these four tables exist helps clarify when to use each one.

Original Modicon PLC Design

When Modbus was created in 1979, Modicon programmable logic controllers had different types of physical I/O modules:

Coils (Outputs)

- Named after electromagnetic relay coils
- Controlled physical relay outputs
- Could energize or de-energize coils to open/close contacts
- Typical use: Turning motors, valves, lights ON/OFF

Discrete Inputs (Digital Inputs)

- Connected to binary sensors and switches
- Read-only from Modbus perspective
- Reflected physical state of input terminals
- Typical use: Limit switches, push buttons, proximity sensors

Input Registers (Analog Inputs)

- Connected to analog input modules (4-20mA, 0-10V)
- Read-only from Modbus perspective
- 16-bit values representing sensor readings
- Typical use: Temperature sensors, pressure transducers, flow meters

Holding Registers (General Storage)

- Writeable memory areas
- Used for setpoints, configuration parameters
- Could be connected to analog output modules
- Also used as PLC scratch pad memory
- Typical use: PID setpoints, timer presets, counters, configuration

Modern Usage Evolution

While the original hardware distinctions have blurred, the four-table model persists:

Modern Devices May:

- Implement all four tables in software (no actual physical I/O)
- Use holding registers exclusively (most flexible)
- Map the same physical I/O to multiple tables
- Use tables for logical organization rather than hardware mapping

Common Modern Patterns:

```
Coils:
- Digital control points
- Alarm acknowledgments
- Mode selection (Auto/Manual)

Discrete Inputs:
```

- Alarm status flags
- Equipment running status
- Digital sensor states

Input Registers:

- Process measurements (temperature, pressure, flow)
- Calculated values (averages, totals)
- Status information

Holding Registers:

- Configuration parameters
- Setpoints and limits
- Counters and accumulators
- Any read/write data

Tip — Modern Best Practice

*Most modern devices primarily use **holding registers** for maximum flexibility. Many devices allow both reading and writing holding registers, even if the data represents "input" values. Always consult device documentation.*

Table Characteristics

Boolean Tables (Coils and Discrete Inputs)

Data Type: Single bit (ON/OFF, TRUE/FALSE, 1/0)

Storage:

- Bits packed into bytes for transmission

- LSB (Least Significant Bit) first
- Partial bytes padded with zeros

Example: Reading 10 coils returns 2 bytes (10 bits + 6 padding bits)

Maximum Quantity per Request:

- Read: Up to 2000 bits (FC01, FC02)
- Write: Up to 1968 bits (FC15)

Common Uses:

- Relay outputs (coils)
- Alarm indicators
- Mode flags
- Digital sensor states

See also: FC01 - Read Coils, FC02 - Read Discrete Inputs

Register Tables (Input Registers and Holding Registers)

Data Type: 16-bit unsigned integer (0-65535)

Byte Order: Big-endian (high byte first) per Modbus specification

- First byte: Most significant 8 bits
- Second byte: Least significant 8 bits
- See: Byte Ordering

Maximum Quantity per Request:

- Read: Up to 125 registers (FC03, FC04)
- Write: Up to 123 registers (FC16)

Why These Limits?

- Modbus PDU maximum size: 253 bytes
- Function code (1) + byte count (1) + data = 253 bytes
- For reads: 1 + 1 + (125 × 2) = 251 bytes ✓
- For writes: 1 + address (2) + quantity (2) + byte count (1) + (123 × 2) = 252 bytes ✓

Multi-Register Data Types:

While individual registers are 16-bit, consecutive registers can represent larger values:

Data Type	Registers	Size	Purpose
UINT16	1	16-bit	Unsigned 0-65535
INT16	1	16-bit	Signed -32768 to 32767
UINT32	2	32-bit	Unsigned long integers
INT32	2	32-bit	Signed long integers

FLOAT32	2	32-bit	IEEE 754 floating point
UINT64	4	64-bit	Very large unsigned integers
INT64	4	64-bit	Very large signed integers
FLOAT64	4	64-bit	Double precision float

See also: Data Types Reference

Scaling and Engineering Units

Important: Modbus registers are just numbers. The protocol has no concept of engineering units or scaling.

Example Problem:

- Device measures 58.5°C
- Modbus register is 16-bit integer (no decimal point)
- How to represent 58.5?

Solution: Scaling convention

```
Device measures: 58.5°C
Device multiplies by 10: 58.5 × 10 = 585
Device stores in register: 585
Modbus transmits: 585

Client receives: 585
Client divides by 10: 585 ÷ 10 = 58.5
Client displays: 58.5°C
```

Critical: Scaling is a **convention** between device and client, documented by device manufacturer. It is not part of the Modbus protocol.

Common Scaling Factors:

- ×10 (one decimal): Temperature, voltage, current
- ×100 (two decimals): Precise measurements, currency
- ×1000 (three decimals): Scientific measurements
- ×1000000 (six decimals): High-precision applications

See also: Modbus Technical Reference - Scaling

Table Address Ranges

Original Specification (9,999 Points)

The original Modbus specification supported up to 9,999 data points per table:

- Coils: 00001 - 09999
- Discrete Inputs: 10001 - 19999

- Input Registers: 30001 - 39999
- Holding Registers: 40001 - 49999

Modern Extended Addressing (65,535 Points)

Modern implementations support the full 16-bit address space:

- Coils: 000001 - 065535
- Discrete Inputs: 100001 - 165535
- Input Registers: 300001 - 365535
- Holding Registers: 400001 - 465535

The first digit (or first two digits) identifies the table type.

> **Warning — Address Confusion**
>
> *The addressing scheme is one of the most confusing aspects of Modbus. See Addressing for complete explanation of public vs protocol addresses and the off-by-one error.*

Choosing the Right Table

Decision Matrix

For Digital/Boolean Data:

Requirement	Use This Table
Read-only status (sensors, alarms)	**Discrete Inputs** (FC02)
Read/write control (relays, outputs)	**Coils** (FC01, FC05, FC15)

For Numeric Data:

Requirement	Use This Table
Read-only values (measurements)	**Input Registers** (FC04)
Read/write values (setpoints, config)	**Holding Registers** (FC03, FC06, FC16, FC23)
Maximum flexibility	**Holding Registers** (most versatile)

Real-World Examples

Temperature Sensor:

```
Device: Temperature transmitter
Measurement: 72.5°F
Storage: Input Register 30001 (value = 725, scale ÷10)
Access: Read-only via FC04
```

Setpoint:

```
Device: Temperature controller
Setpoint: 70.0°F
Storage: Holding Register 40001 (value = 700, scale ÷10)
Access: Read/write via FC03 (read), FC06 (write)
```

Pump Control:

```
Device: Pump controller
Control: Start/Stop command
Storage: Coil 00001
Access: Read/write via FC01 (read status), FC05 (write command)
```

Alarm Status:

```
Device: Safety system
Status: High temperature alarm
Storage: Discrete Input 10001
Access: Read-only via FC02
```

Table Independence

Important Concept: The four tables are completely independent.

This means:

- Coil 00001 is different from Holding Register 40001
- You can have Coil 5, Discrete Input 5, Input Register 5, and Holding Register 5 all in the same device
- Each table has its own address space (0-65535)
- Function codes determine which table is accessed

Example:

```
FC01, Address 0x0005 → Reads Coil 6 (public address 00006)
FC02, Address 0x0005 → Reads Discrete Input 6 (public address 10006)
FC03, Address 0x0005 → Reads Holding Register 6 (public address 40006)
FC04, Address 0x0005 → Reads Input Register 6 (public address 30006)
```

All four operations access **different memory locations** in the device.

Device Implementation Variations

Not All Devices Implement All Tables

Common Patterns:

Simple Devices:

- May only implement holding registers
- All data accessed via FC03/FC06/FC16

Legacy Devices:

- May strictly separate read-only (input registers) from read/write (holding registers)
- Attempting to write input registers returns Exception 01 (Illegal Function)

Modern Flexible Devices:

- May allow reading from all four tables
- May allow writing to "input" registers (device-specific)
- May mirror same data across multiple tables

Sparse Address Maps

Devices typically don't implement all 65,535 addresses:

Typical Device:

```
Coils:
  00001-00050: Control outputs (50 coils)

Discrete Inputs:
  10001-10032: Digital inputs (32 inputs)


Input Registers:
  30001-30100: Measurements (100 registers)


Holding Registers:
  40001-40100: Process data (100 registers)
  40201-40300: Configuration (100 registers)
  40501-40550: Diagnostic counters (50 registers)
```

Accessing undefined addresses results in Exception 02 (Illegal Data Address).

> **Tip — Finding Valid Addresses**
>
> *Always consult device documentation (register map or memory map) to find valid addresses. Chipkin Modbus Explorer's Discovery feature can help identify available addresses automatically.*

Related Topics

Understanding Addressing:

- Addressing - Public vs protocol addresses, off-by-one error

Reading Data:

- FC01 - Read Coils

- FC02 - Read Discrete Inputs
- FC03 - Read Holding Registers
- FC04 - Read Input Registers

Writing Data:

- FC05 - Write Single Coil
- FC06 - Write Single Register
- FC15 - Write Multiple Coils
- FC16 - Write Multiple Registers

Data Representation:

- Modbus Technical Reference - Data Types
- Modbus Technical Reference - Byte Ordering
- Glossary - Data Model Terms

Error Handling:

- Exception 02 - Illegal Data Address
- Exceptions Overview

Modbus Message Structure

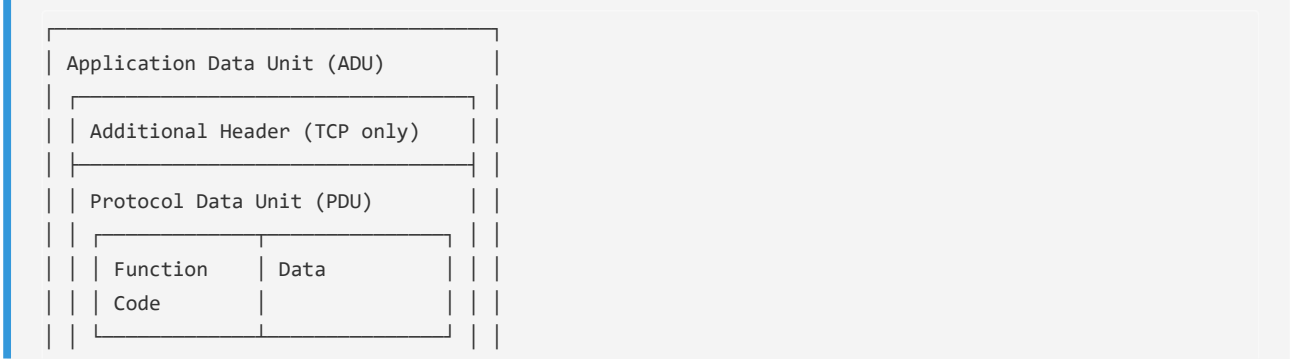
Understanding how Modbus messages are constructed is essential for implementing a Modbus stack or troubleshooting communication issues.

Message Hierarchy

Every Modbus transaction consists of two parts:

```
Client Request  →  [MESSAGE]  →  Server/Slave
Server Response ←  [MESSAGE]  ←  Server/Slave
```

Each message has a layered structure:



```
│ ├──────────────────────────────────┤ │
│ │ Error Check (RTU/ASCII only)     │ │
│ └──────────────────────────────────┘ │
└──────────────────────────────────────┘
```

Protocol Data Unit (PDU)

The **PDU** is the core Modbus message, independent of underlying communication layer.

PDU Structure

```
┌──────────────┬──────────────────────────┐
│ Function Code│ Data                     │
│ (1 byte)     │ (0-252 bytes)            │
└──────────────┴──────────────────────────┘
```

Maximum PDU Size: 253 bytes

- 1 byte function code
- Up to 252 bytes of data

Function Code (1 byte)

The function code tells the server what operation to perform:

Range	Usage
1-64	Public function codes (defined in specification)
65-72	User-defined function codes
73-99	Public function codes (defined in specification)
100-110	User-defined function codes
111-127	Reserved for future use

Exception Response: If function code ≥ 128 (0x80), it's an exception response

- Exception FC = Normal FC + 0x80
- Example: FC03 exception = 0x83 (0x03 + 0x80)

See: Function Code Reference, Exceptions

Data Field (0-252 bytes)

Content varies by function code. Common patterns:

Read Request:

```
┌──────────────┬──────────────┬──────────────┐
│ Function Code│ Start Address │ Quantity     │
│ (1 byte)     │ (2 bytes)     │ (2 bytes)    │
└──────────────┴──────────────┴──────────────┘
```

Read Response:

```
┌──────────────┬──────────────┬──────────────────┐
│ Function Code│ Byte Count   │ Register Values  │
│ (1 byte)     │ (1 byte)     │ (N bytes)        │
└──────────────┴──────────────┴──────────────────┘
```

Write Request:

```
┌──────────────┬──────────────┬──────────────┬──────────────┬──────────────┐
│ Function Code│ Address      │ Quantity     │ Byte         │ Values       │
│ (1 byte)     │ (2 bytes)    │(2 bytes)     │Count         │(N bytes)     │
│              │              │              │(1 byte)      │              │
└──────────────┴──────────────┴──────────────┴──────────────┴──────────────┘
```

Exception Response:

```
┌──────────────┬──────────────────┐
│ Function Code│ Exception Code   │
│ + 0x80       │ (1 byte)         │
│ (1 byte)     │                  │
└──────────────┴──────────────────┘
```

See individual function pages for exact data field formats.

Application Data Unit (ADU)

The **ADU** wraps the PDU with additional addressing and error-checking information. The ADU format differs between Modbus TCP and Modbus RTU/ASCII.

Modbus TCP ADU

```
┌──────────────────────────────────────────────┐
│ MBAP Header                                  │
│ ┌──────────────┬──────────────┬─────┬────────┐│
│ │ Transaction  │ Protocol     │ Len │ Unit   ││
│ │ ID (2 bytes) │ ID (2 bytes) │(2)  │ID (1)  ││
│ └──────────────┴──────────────┴─────┴────────┘│
├──────────────────────────────────────────────┤
│ PDU (Function Code + Data)                   │
└──────────────────────────────────────────────┘
```

MBAP Header Fields:

Field	Size	Description
Transaction ID	2 bytes	Request/response matching identifier
Protocol ID	2 bytes	Always 0x0000 for Modbus
Length	2 bytes	Number of following bytes (Unit ID + PDU)
Unit ID	1 byte	Slave address (1-247, or 255 for broadcast)

Total TCP ADU Size:

- MBAP header: 7 bytes
- PDU: 1-253 bytes
- Total: 8-260 bytes

Example - Read Holding Register:

```
MBAP Header:
  Transaction ID: 0x0001
  Protocol ID:    0x0000
  Length:         0x0006 (6 bytes to follow)
  Unit ID:        0x01

PDU:
  Function:       0x03 (Read Holding Registers)
  Start Address:  0x006B (register 107)
  Quantity:       0x0001 (1 register)


Complete TCP ADU:
[00 01][00 00][00 06][01] [03][00 6B][00 01]
 Trans  Proto  Len   Unit  FC  Addr   Qty
```

> **Tip — No CRC in Modbus TCP**
>
> *Modbus TCP does **not** include CRC or error checking in the ADU. Error detection is handled by the TCP layer (TCP checksums).*

See: Modbus Technical Reference - TCP/IP

Modbus RTU ADU

```
┌──────────┬──────────────────────────┬──────────┐
│ Unit ID  │ PDU                      │ CRC-16   │
│ (1 byte) │ (Function Code + Data)   │ (2 bytes)│
└──────────┴──────────────────────────┴──────────┘
```

RTU ADU Fields:

Field	Size	Description

Unit ID	1 byte	Slave address (1-247, 0 = broadcast)
PDU	1-253 bytes	Function code + data
CRC-16	2 bytes	Error checking (LSB first)

Total RTU ADU Size:

- Unit ID: 1 byte
- PDU: 1-253 bytes
- CRC: 2 bytes
- Total: 4-256 bytes

Example - Same Read Request in RTU:

```
Unit ID:        0x01
PDU:
  Function:     0x03
  Start Addr:   0x006B
  Quantity:     0x0001
CRC-16:         0xC5B4 (calculated)
```

```
Complete RTU ADU:
[01][03][00 6B][00 01][B4 C5]
 Unit FC  Addr   Qty   CRC(LSB,MSB)
```

CRC Calculation:

The CRC-16 is calculated over all bytes except the CRC itself:

```
CRC Input: [01 03 00 6B 00 01]
CRC Output: 0xC5B4
Transmitted: [B4 C5] (LSB first)
```

> **Warning — CRC Byte Order**
>
> *The CRC is transmitted **LSB first** (low byte, then high byte), which is the opposite of data values in Modbus.*

Silent Interval:

Modbus RTU requires silence on the line before and after each message:

- Minimum gap: 3.5 character times
- At 9600 baud: ~3.6ms
- At 19200 baud: ~1.8ms
- Marks the beginning and end of frames

See: Modbus Technical Reference - RTU

Modbus ASCII ADU

```
┌─────┬────────┬──────────────────┬─────┬──────┐
│ ':' │ Unit ID│ PDU              │ LRC│ CRLF │
│     │(2 char)│(ASCII hex chars) │(2) │      │
└─────┴────────┴──────────────────┴─────┴──────┘
```

ASCII ADU Fields:

Field	Size	Description
Start	1 byte	Colon character ':' (0x3A)
Unit ID	2 bytes	Slave address as ASCII hex (e.g., "01")
Function Code	2 bytes	Function as ASCII hex (e.g., "03")
Data	Variable	Data as ASCII hex pairs
LRC	2 bytes	Longitudinal Redundancy Check as ASCII hex
End	2 bytes	Carriage Return + Line Feed (0x0D 0x0A)

Example - Same Read Request in ASCII:

```
Start:     ':'
Unit ID:   "01" (ASCII '0' '1')
Function:  "03"
Address:   "006B"
Quantity:  "0001"
LRC:       "F6" (calculated)
End:       CR LF

Complete ASCII ADU:
:01 03 00 6B 00 01 F6 <CR><LF>
```

LRC Calculation:

```
1. Add all bytes (as binary): 01 + 03 + 00 + 6B + 00 + 01 = 0x70
2. Two's complement: 0x100 - 0x70 = 0x90
3. Result: 0x90
4. Transmit as ASCII: "90"
```

Character Encoding:

- Each byte becomes 2 ASCII hex characters
- Example: Byte 0x6B → ASCII "6B" (characters '6' and 'B')
- Message size doubles compared to RTU

Warning — ASCII Rarely Used

Modbus ASCII is rarely used in modern systems due to inefficiency (double size) and slower speed. Most implementations use Modbus RTU or Modbus TCP.

Byte Ordering

Register Values (Data)

Modbus Standard: Big-Endian (Network Order)

Multi-byte values are transmitted **most significant byte first**:

```
16-bit Register Value: 0x022B (555 decimal)

Transmission order:
  Byte 1: 0x02 (high byte)
  Byte 2: 0x2B (low byte)
```

For 32-bit Values (two registers):

Standard requires high register first, but device implementations vary:

```
Value: 0x00 01 E2 40 (123,456 decimal)

Standard (Big-Endian ABCD):
  Register 1: 0x0001
  Register 2: 0xE240
  Bytes: [00][01][E2][40]
```

However, non-compliant devices may use other byte orders. See: Byte Ordering Reference

CRC in RTU (Error Check)

Exception: LSB First

The CRC-16 value is transmitted **least significant byte first**:

```
CRC Value: 0xC5B4

Transmission order:
  Byte 1: 0xB4 (low byte)
  Byte 2: 0xC5 (high byte)
```

This is the **only** place in Modbus where LSB-first order is used.

Message Examples

FC03 Read Holding Registers - Complete Transaction

Scenario: Read 2 registers starting at address 40108 (protocol address 107)

Modbus TCP Request:

```
┌──────────────────── MBAP Header ────────────────────┬─── PDU ───┐
[00 01][00 00][00 06][01] [03][00 6B][00 02]
 Trans  Proto  Len   Unit  FC  Addr   Qty
  ID     ID
```

```
Breakdown:
  Transaction ID: 0x0001 (for matching response)
  Protocol ID:    0x0000 (always 0 for Modbus)
  Length:         0x0006 (6 bytes follow)
  Unit ID:        0x01   (device 1)
  Function:       0x03   (Read Holding Registers)
  Start Address:  0x006B (107 decimal)
  Quantity:       0x0002 (2 registers)
```

Modbus TCP Response:

```
┌──────────────────── MBAP Header ────────────────────┬──── PDU ────┐
[00 01][00 00][00 07][01] [03][04][02 2B][00 00]
 Trans  Proto  Len   Unit  FC  ByteCt  Reg1   Reg2
```

```
Breakdown:
  Transaction ID: 0x0001 (matches request)
  Protocol ID:    0x0000
  Length:         0x0007 (7 bytes follow)
  Unit ID:        0x01
  Function:       0x03
  Byte Count:     0x04 (4 bytes of data = 2 registers)
  Register 107:   0x022B (555 decimal)
  Register 108:   0x0000 (0 decimal)
```

Same Transaction in Modbus RTU:

RTU Request:

```
[01][03][00 6B][00 02][B4 C5]
 Unit FC  Addr   Qty   CRC
```

```
Breakdown:
  Unit ID:       0x01
  Function:      0x03
  Start Address: 0x006B
  Quantity:      0x0002
  CRC-16:        0xC5B4 (transmitted LSB first as B4 C5)
```

RTU Response:

```
[01][03][04][02 2B][00 00][63 3A]
 Unit FC  ByteCt  Reg1   Reg2   CRC

Breakdown:
  Unit ID:       0x01
  Function:      0x03
  Byte Count:    0x04
  Register 107:  0x022B
  Register 108:  0x0000
  CRC-16:        0x3A63 (transmitted as 63 3A)
```

FC06 Write Single Register - Complete Transaction

Scenario: Write value 3 to register 40002 (protocol address 1)

Modbus TCP Request:

```
[00 02][00 00][00 06][01] [06][00 01][00 03]
 Trans  Proto  Len   Unit  FC  Addr   Value
```

Modbus TCP Response (Echo):

```
[00 02][00 00][00 06][01] [06][00 01][00 03]
 (Exact echo of request confirms write)
```

Exception Response Example

Scenario: Request invalid address, receive exception 02

Modbus TCP Request:

```
[00 03][00 00][00 06][01] [03][04 A1][00 01]
 Trans  Proto  Len   Unit  FC  BadAddr Qty
```

Modbus TCP Exception Response:

```
[00 03][00 00][00 03][01] [83][02]
 Trans  Proto  Len   Unit  ErrFC ExCode

Breakdown:
  Transaction ID: 0x0003 (matches request)
  Function:       0x83 (0x03 + 0x80 = exception)
  Exception Code: 0x02 (Illegal Data Address)
```

See: Exception 02 - Illegal Data Address

Implementing Message Construction

Building a Request

Algorithm:

```
1. Determine function code (FC01-FC43, etc.)
2. Build data field according to function specification
3. Combine FC + Data = PDU
4. Add framing:
   - For TCP: Add MBAP header
   - For RTU: Add Unit ID, calculate and append CRC
   - For ASCII: Convert to ASCII hex, add ':', LRC, CR-LF
5. Transmit
```

Parsing a Response

Algorithm:

```
1. Receive message
2. Validate framing:
   - For TCP: Check MBAP header, verify length
   - For RTU: Verify silent interval, check CRC
   - For ASCII: Verify ':', LRC, CR-LF
3. Extract PDU
4. Check function code:
   - If FC < 128: Normal response
   - If FC >= 128: Exception response
5. Parse data field according to function code
6. Return parsed result to application
```

Error Detection

TCP:

- TCP layer handles error detection with checksums
- Application validates MBAP header fields
- Check for exception responses

RTU:

- Calculate CRC-16 over received bytes
- Compare with received CRC
- If mismatch: Discard message (no response)
- If match: Process message

ASCII:

- Calculate LRC over received bytes

- Compare with received LRC
- Validate start ':' and end CR-LF
- If error: Discard message

Related Topics

Protocol Basics:

- Data Model - The four tables
- Addressing - Public vs protocol addressing
- Protocol Index - All function codes

Function Code Specifications:

- FC03 - Read Holding Registers - Most common function
- FC06 - Write Single Register - Write example
- All Function Codes

Error Handling:

- Exceptions Overview - How errors are reported
- Exception Code List

Implementation Details:

- Modbus Technical Reference - TCP
- Modbus Technical Reference - RTU
- Glossary - Protocol Terms

Modbus Exception Responses

When a Modbus server cannot process a request normally, it returns an **exception response** instead of a normal response. Exception responses provide diagnostic information about what went wrong.

Exception Response Format

How to Recognize an Exception

Normal Response:

```
[03][04][00 0A][00 14]
 FC  ByteCt  Data...
```

Exception Response:

```
[83][02]
 FC  ExCode
```

```
FC = 0x83 = 131 decimal = 0x03 + 0x80
     ^^^^ MSB set to 1
```

Rule: If the function code in the response ≥ 128 (0x80), it's an exception response.

Exception PDU Structure

```
┌──────────────────────┬──────────────────┐
│ Exception Function   │ Exception Code   │
│ Code (1 byte)        │ (1 byte)         │
│ = FC + 0x80          │                  │
└──────────────────────┴──────────────────┘
```

Exception Function Code:

- Original function code with MSB set to 1
- Calculated as: Original FC + 0x80 (or Original FC | 0x80)

Examples:

Normal FC	Function Name	Exception FC	Calculation
0x01	Read Coils	0x81	0x01 + 0x80 = 0x81
0x03	Read Holding Registers	0x83	0x03 + 0x80 = 0x83
0x06	Write Single Register	0x86	0x06 + 0x80 = 0x86
0x10	Write Multiple Registers	0x90	0x10 + 0x80 = 0x90

Exception Codes

Standard Exception Codes

Code	Name	When It Occurs	Typical Cause
01	Illegal Function	Function code not supported	Device doesn't implement this function
02	Illegal Data Address	Address doesn't exist	Reading/writing undefined memory
03	Illegal Data Value	Value in query invalid	Malformed request, bad quantity
04	Server Device Failure	Unrecoverable error	Hardware fault, internal error
05	Acknowledge	Long operation accepted	Programming command in progress
06	Server Device Busy	Device processing long operation	Retry later

08	Memory Parity Error	Extended memory error	File record parity check failed
0A	Gateway Path Unavailable	Gateway misconfigured	No route to target device
0B	Gateway Target Failed to Respond	No response from target	Device offline or not responding

Reserved Codes:

- **07, 09**: Reserved (not used)
- **0C-FF**: Reserved for future use or vendor-specific

Complete Exception Example

Example: Reading Invalid Address

Request: Read holding register at non-existent address

Modbus TCP Request:

```
[00 01][00 00][00 06][01] [03][04 A1][00 01]
 Trans  Proto  Len   Unit  FC  Addr   Qty

Request Details:
  Transaction ID: 1
  Function: 03 (Read Holding Registers)
  Address: 0x04A1 (1185 decimal = register 41186)
  Quantity: 1


Assumption: Device only has registers 0-999
```

Modbus TCP Exception Response:

```
[00 01][00 00][00 03][01] [83][02]
 Trans  Proto  Len   Unit  ErrFC ExCode

Response Details:
  Transaction ID: 1 (matches request)
  Length: 3 (1 Unit ID + 1 FC + 1 ExCode)
  Function: 0x83 (0x03 + 0x80 = exception)
  Exception Code: 0x02 (Illegal Data Address)


Meaning: Address 1185 doesn't exist in this device
```

Modbus RTU Request:

```
[01][03][04 A1][00 01][XX XX]
```

```
 Unit FC  Addr   Qty   CRC
```

Modbus RTU Exception Response:

```
[01][83][02][XX XX]
 Unit ErrFC ExCode CRC
```

Exception Response Flow

Normal Response Flow

```
Client Request → [FC=03][Address][Quantity]
                 ↓
Server Processing → ✓ Address exists
                    ✓ Quantity valid
                    ✓ Read successful
                 ↓
Server Response → [FC=03][ByteCount][Data...]
```

Exception Response Flow

```
Client Request → [FC=03][Address][Quantity]
                 ↓
Server Processing → ✗ Address doesn't exist
                 ↓
Server Exception → [FC=83][ExCode=02]
```

Handling Exceptions in Client Code

Detection Algorithm

```
FUNCTION parseResponse(response)
  // The function code is the first byte of the PDU
  functionCode = response[0]

  // Check if the most significant bit is set (value >= 128)
  IF functionCode >= 0x80 THEN
    // This is an exception response
    originalFC = functionCode - 0x80
    exceptionCode = response[1]
```

```
    // Raise an error or return a special exception object
    THROW NEW ModbusException(originalFC, exceptionCode)


  ELSE
    // This is a normal response, process it accordingly
    RETURN parseNormalResponse(functionCode, response)
  END IF
END FUNCTION
```

Exception Handling Strategy

Level 1: Automatic Retry (for transient errors)

```
IF exceptionCode IS 0x06 THEN // Server Device Busy
  // Wait for a short period and then retry the request
  SLEEP(100 milliseconds)
  RETURN retry(request)
END IF
```

Level 2: User Notification (for configuration errors)

```
IF exceptionCode IS 0x02 THEN // Illegal Data Address
  // Inform the user that the request is invalid
  SHOW_ERROR("The requested address does not exist. Please check the device's memory map.")
END IF
```

Level 3: Fatal Error (for unrecoverable errors)

```
IF exceptionCode IS 0x04 THEN // Server Device Failure
  // Notify the user of a critical hardware problem and stop communication
  SHOW_ERROR("The device reported a hardware failure. Please contact support.")
  disconnectDevice()
END IF
```

Exception Code Details

Exception 01 - Illegal Function

When: Function code not supported or not allowed in current state

Examples:

- Device doesn't implement FC20 (Read File Record)
- Device in wrong mode for this function
- Function only available in newer firmware

See: Exception 01 Details

Exception 02 - Illegal Data Address

When: The data address is not valid for this device

Examples:

- Reading address 1000 when device only has 0-999
- Start address exists but start + quantity exceeds range
- Address not implemented (sparse memory map)

See: Exception 02 Details

Exception 03 - Illegal Data Value

When: Value in the query data field is not allowable

Examples:

- Quantity = 0 (must be ≥ 1)
- Quantity > maximum (e.g., 126 registers for FC03)
- Invalid sub-function code
- Malformed message structure

Important: NOT used for out-of-range data values in writes. Protocol is unaware of data semantics.

See: Exception 03 Details

Exception 04 - Server Device Failure

When: Unrecoverable error occurred while processing

Examples:

- Internal hardware fault
- Memory corruption detected
- Watchdog timer reset
- Critical system error

See: Exception 04 Details

Exception 05 - Acknowledge

When: Long-duration operation has been accepted

Examples:

- Program download initiated
- Configuration write started
- Use FC08 sub-function or another polling mechanism to check completion

Not an error: Server is processing, client should poll for completion

See: Exception 05 Details

Exception 06 - Server Device Busy

When: Device is busy processing a long-duration command

Examples:

- Still processing previous program command
- Performing internal calibration
- Busy with higher-priority task

Action: Retry request after delay

See: Exception 06 Details

Exception 08 - Memory Parity Error

When: Extended file area parity check failed

Examples:

- Used with FC20/FC21 (Read/Write File Record)
- Reference type 6 (extended file)
- Memory corruption detected

Action: Can retry, but device may need service

See: Exception 08 Details

Exception 0A - Gateway Path Unavailable

When: Gateway cannot allocate communication path

Examples:

- Gateway misconfigured
- Gateway communication channels full
- No routing from input port to output port

Only for gateways

See: Exception 0A Details

Exception 0B - Gateway Target Device Failed to Respond

When: Target device didn't respond to gateway

Examples:

- Target device offline
- Target device powered off
- Communication link failure
- Wrong address

Only for gateways

See: Exception 0B Details

Best Practices

Always Check for Exceptions

Don't assume success:

```
// Bad: Assumes the request will always succeed
response = sendModbusRequest(request)
value = response.registers[0]
```

```
// Good: Wraps the request in a try-catch block
TRY
  response = sendModbusRequest(request)
  value = response.registers[0]
CATCH ModbusException AS e
  handleException(e)
END TRY
```

Log Exceptions for Debugging

```
FUNCTION handleException(e)
  LOG "Modbus Exception Occurred:"
  LOG "  Original Function: " + e.functionCode
  LOG "  Exception Code: " + e.exceptionCode
  LOG "  Description: " + getExceptionDescription(e.exceptionCode)

  // ... take appropriate action based on the exception ...
END FUNCTION
```

Implement Appropriate Retry Logic

Retry for transient errors:

- Exception 06 (Server Busy) - Retry after delay
- No response (timeout) - Retry with backoff

Don't retry for permanent errors:

- Exception 02 (Illegal Address) - Fix configuration
- Exception 01 (Illegal Function) - Use different function

Provide User-Friendly Messages

Instead of:

> *"Exception 02 received"*

Provide:

> *"Address 40500 doesn't exist. This device only supports addresses 40001-40100. Check the device manual for valid addresses."*

Using Chipkin Modbus Explorer

Chipkin Modbus Explorer automatically:

1. **Detects exceptions** in responses
2. **Decodes exception codes** with descriptions
3. **Logs exceptions** for troubleshooting
4. **Displays human-readable messages**

Example Display:

```
❌ Exception Response

Function Code: 03 (Read Holding Registers)
Exception Code: 02 (Illegal Data Address)


Description:
The data address received in the query is not an allowable
address for the device. The address may not exist, or the
combination of address + quantity may exceed the available range.


Suggestion:
Check the device memory map to verify address 1185 exists.
```

Related Topics

Individual Exception Details:

- Exception 01 - Illegal Function
- Exception 02 - Illegal Data Address
- Exception 03 - Illegal Data Value
- Exception 04 - Server Device Failure
- Exception 05 - Acknowledge
- Exception 06 - Server Device Busy
- Exception 08 - Memory Parity Error
- Exception 0A - Gateway Path Unavailable
- Exception 0B - Gateway Target Failed

Protocol Basics:

- Message Structure - How exceptions fit into PDUs
- Data Model - Understanding data organization
- Addressing - Avoiding address errors

Troubleshooting:

- Troubleshooting Guide - Solving common problems
- Glossary - Exception Code - Terminology

Function References:

- FC03 - Read Holding Registers
- All Function Codes

Exception 01 - Illegal Function

Exception Code: 0x01

Name: ILLEGAL FUNCTION

Severity: ⚠️ Configuration Error - Request must be corrected

What It Means

The function code received in the query is not an action that the server can perform. This is a fundamental configuration mismatch between the client and the server.

This exception indicates that:

- The server does not support the requested function code at all.
- The server is not in a state where it can execute the function.
- The client is attempting to write to a read-only data block.

When It Occurs

Scenario 1: Function Not Supported

The most common cause is that the device simply does not implement the requested function code.

Example:

```
Device is a simple sensor that only supports:
- FC03 (Read Holding Registers)
- FC04 (Read Input Registers)
```

```
Client attempts to write a register:
Request:  [06][00 01][00 7B]
           FC=06 Addr=1 Value=123
Response: [86][01]
          ErrFC ExCode


Why: The device is read-only and does not implement FC06 (Write Single Register).
```

Scenario 2: Device in Wrong State

Some devices only allow certain functions in specific modes (e.g., "configuration mode" vs. "run mode").

Example:

```
A programmable logic controller (PLC) might only allow FC16 (Write Multiple Registers) when it is in
"Program" or "Stop" mode.

If the PLC is in "Run" mode:
Request:  [10][...data...]
           FC=16
Response: [90][01]
          ErrFC ExCode


Why: The function is illegal in the current state.
```

Scenario 3: Writing to a Read-Only Address Block

Some devices use Exception 01 to reject write attempts to an entire table that is considered read-only.

Example:

```
A client attempts to write to an Input Register (address 30005). Input registers are always read-only.

Request:  [06][00 04][12 34]
           FC=06 Addr=4 (for 30005)
Response: [86][01]
          ErrFC ExCode


Why: The device rejects any write attempt to the Input Register table with "Illegal Function".
```

Tip — Device Behavior Varies

Other devices might return Exception 02 - Illegal Data Address in this scenario. The Modbus standard allows for either interpretation.

Function Codes That Can Return Exception 01

Any function code can theoretically return this exception if it's not supported by the server.

Complete Example

FC06 Write Single Register - Not Supported

Modbus TCP Request:

```
[00 0A][00 00][00 06][01] [06][00 01][00 7B]
 Trans  Proto  Len   Unit  FC  Addr   Value

Transaction ID: 10
Function: 06 (Write Single Register)
Address: 1 (public 40002)
Value: 123


Assumption: The server device is a simple sensor and does not support writing.
```

Modbus TCP Exception Response:

```
[00 0A][00 00][00 03][01] [86][01]
 Trans  Proto  Len   Unit  ErrFC ExCode

Transaction ID: 10 (matches request)
Exception Function: 0x86 (0x06 + 0x80)
Exception Code: 0x01 (Illegal Function)


Meaning: The server does not know how to perform Function Code 06.
```

Troubleshooting Steps

Step 1: Verify Function Code Support

Check device documentation:

- Look for a "Supported Function Codes" table in the manual.
- Confirm that the function code you are using is listed.

Example from device manual:

FC	Description	Supported
01	Read Coils	Yes

03	Read Holding Registers	Yes
04	Read Input Registers	Yes
06	Write Single Register	No
16	Write Multiple Registers	No

Step 2: Check the Data Table

Are you reading/writing to the correct block?

- **Coils (0xxxx):** Read with FC01, Write with FC05/FC15.
- **Discrete Inputs (1xxxx):** Read with FC02. Read-only.
- **Input Registers (3xxxx):** Read with FC04. Read-only.
- **Holding Registers (4xxxx):** Read with FC03, Write with FC06/FC16.

An attempt to write to a read-only block (Discrete Inputs or Input Registers) can cause this exception.

Step 3: Check Device State

- If the device has different operating modes, ensure it is in the correct mode to accept your request.
- This may require physical interaction with the device or a separate command to change its state.

Prevention Strategies

Client-Side Capability Profile

Before communicating, configure the client with the known capabilities of the server.

```
{
  "device": "Simple Sensor XYZ",
  "supportedFunctions": [1, 2, 3, 4]
}
```

```
FUNCTION sendRequest(functionCode, payload)
  // Get the list of functions supported by the target device
  supported = getDeviceProfile().supportedFunctions

  // Check if the requested function is in the list
  IF functionCode NOT IN supported THEN
    // Abort the request before it's even sent
    THROW ERROR "Function " + functionCode + " is not supported by this device."
  END IF
```

```
  // If supported, proceed with sending the request
  // ...
END FUNCTION
```

Use Discovery Tools

Some advanced Modbus clients or discovery tools can query a device (e.g., using FC43 - Read Device Identification) to learn about its capabilities, though support for this is not universal.

Related Topics

Other Exceptions:

- Exception 02 - Illegal Data Address
- Exception 03 - Illegal Data Value
- Exceptions Overview - All exception codes

Data Model:

- Data Model - The four data tables.

Exception 02 - Illegal Data Address

Exception Code: 0x02

Name: ILLEGAL DATA ADDRESS

Severity: ⚠️ Configuration Error - Request must be corrected

What It Means

The data address received in the query **is not an allowable address** for the server device.

This exception indicates that:

- The specified address doesn't exist in the device
- The address exists but is not implemented
- The combination of starting address + quantity exceeds the available range

When It Occurs

Scenario 1: Address Doesn't Exist

Example:

```
Device implements registers: 0-99 (public 40001-40100)
Request asks for: Register 100 (public 40101)
Result: Exception 02
```

```
Request:  [03][00 64][00 01]
           FC  Addr=100 Qty=1
Response: [83][02]
          ErrFC ExCode
```

Scenario 2: Range Exceeds Available Memory

Example:

```
Device implements registers: 0-99 (public 40001-40100)
Request asks for: Start at 96, read 5 registers (96-100)
Result: Exception 02

Why: Registers 96, 97, 98, 99 exist (4 registers)
     Register 100 does NOT exist
     Therefore: Cannot read 5 registers
```

```
Request:  [03][00 60][00 05]
           FC  Addr=96 Qty=5
Response: [83][02]
```

Correct request:

```
Request:  [03][00 60][00 04]
           FC  Addr=96 Qty=4
Response: [03][08][data for 4 registers...]
          ✓ Success - reads registers 96-99 only
```

Scenario 3: Sparse Memory Map

Many devices have non-contiguous address ranges:

Example Device:

```
Implemented addresses:
  0-49:    Process values (40001-40050)
  200-249: Configuration (40201-40250)

Unimplemented addresses:
  50-199:  NOT IMPLEMENTED
  250+:    NOT IMPLEMENTED
```

Valid request:

```
Read 40001: [03][00 00][00 01] → ✓ OK
Read 40050: [03][00 31][00 01] → ✓ OK
Read 40201: [03][00 C8][00 01] → ✓ OK
```

Invalid request:

```
Read 40100: [03][00 63][00 01] → ✗ Exception 02
(Address 99 not implemented)
```

Scenario 4: Write to Read-Only Address

Some devices return Exception 02 when attempting to write to read-only addresses:

Example:

```
Input Registers are read-only (FC04)
Attempt to write: [06][00 10][00 05]
                  FC06 Addr=16 Value=5

Some devices return: [86][02]
                     ^^^^ Exception 02


Others return: [86][01] (Exception 01 - Illegal Function)
```

> **Tip — Device Behavior Varies**
>
> *Different manufacturers handle read-only write attempts differently. Some use Exception 01, others use Exception 02. Check device documentation.*

Function Codes That Can Return Exception 02

Any function that specifies addresses:

- **FC01** - Read Coils
- **FC02** - Read Discrete Inputs
- **FC03** - Read Holding Registers *(most common)*
- **FC04** - Read Input Registers
- **FC05** - Write Single Coil
- **FC06** - Write Single Register
- **FC15** - Write Multiple Coils
- **FC16** - Write Multiple Registers
- **FC23** - Read/Write Multiple Registers

Complete Example

FC03 Read Holding Registers - Invalid Address

Modbus TCP Request:

```
[00 05][00 00][00 06][01] [03][04 A1][00 01]
 Trans  Proto  Len   Unit  FC  Addr   Qty

Transaction ID: 5
Function: 03 (Read Holding Registers)
Address: 0x04A1 (1185 decimal → public 41186)
Quantity: 1 register


Assumption: Device only has registers 0-999
```

Modbus TCP Exception Response:

```
[00 05][00 00][00 03][01] [83][02]
 Trans  Proto  Len   Unit  ErrFC ExCode

Transaction ID: 5 (matches request)
Exception Function: 0x83 (0x03 + 0x80)
Exception Code: 0x02 (Illegal Data Address)


Meaning: Address 1185 doesn't exist
```

Modbus RTU Request:

```
[01][03][04 A1][00 01][XX XX]
 Unit FC  Addr   Qty   CRC
```

Modbus RTU Exception Response:

```
[01][83][02][XX XX]
 Unit ErrFC ExCode CRC
```

Troubleshooting Steps

Step 1: Verify Address Exists

Check device documentation (memory map or register map):

```
Example from device manual:

Holding Registers:
  40001-40100: Process Data
```

```
  40201-40250: Configuration
  40501-40510: Diagnostic Counters


Valid addresses: 0-99, 200-249, 500-509
Invalid examples: 100, 150, 300, 511
```

Step 2: Check Address Conversion

Verify public ↔ protocol conversion:

```
Want to read: Holding Register 40108
Conversion:
  Public: 40108
  Table: Holding Register (first digit '4')
  Point: 108
  Protocol: 108 - 1 = 107 (0x006B)

Correct request: [03][00 6B][00 01]
Wrong request:   [03][00 6C][00 01] ← Address 108 = register 40109
```

See: Addressing for conversion details

Step 3: Validate Range Calculation

For multi-register reads:

```
Want: Registers 40096-40100 (5 registers)

Calculation:
  Start public: 40096
  Start protocol: 96 - 1 = 95 (0x005F)
  Quantity: 5


  Reads: 95, 96, 97, 98, 99 (protocol)
  Public: 40096, 40097, 40098, 40099, 40100 ✓


Request: [03][00 5F][00 05]
```

Common mistake:

```
Start: 40096 → Protocol 96 (WRONG - forgot -1)
Quantity: 5

Reads: 96, 97, 98, 99, 100
Public: 40097, 40098, 40099, 40100, 40101
```

```
If device ends at 40100, address 40101 causes Exception 02
```

Step 4: Check for Sparse Memory Map

Test boundary addresses:

```
If getting Exception 02 unexpectedly:

1. Try reading first known address: 40001
   Request: [03][00 00][00 01]

2. Try reading last known address: 40100
   Request: [03][00 63][00 01]

3. Try addresses in between to find gaps

4. Document which addresses work and which don't
```

Step 5: Use Discovery Tools

Chipkin Modbus Explorer has Discovery feature:

1. Go to Discovery page
2. Select address range to scan
3. Click Start Discovery
4. Tool automatically finds implemented addresses
5. Generates memory map showing valid addresses

See: Discovery Guide

Prevention Strategies

Always Consult Device Documentation

Before implementation:

1. Obtain device manual or datasheet
2. Locate memory map or register map

3. Identify implemented address ranges
4. Note any gaps in addressing
5. Document findings

Validate Addresses Before Sending

Client-side validation:

```
FUNCTION validateReadRequest(functionCode, startAddress, quantity)
  // Retrieve the known valid address ranges for the device
  // based on the function code (e.g., holding registers vs. coils)
  validRanges = getDeviceMemoryMap(functionCode)

  // Iterate through every address in the requested block
  FOR i FROM 0 TO quantity - 1
    address = startAddress + i


    // Check if the calculated address falls within any of the valid ranges
    IF isAddressInRanges(address, validRanges) IS FALSE THEN
      // If not, abort and report the error
      THROW ERROR &quot;Address &quot; + address + &quot; is not valid. Valid ranges
    are: &quot; + validRanges
    END IF


  END FOR


  // If all addresses are valid, the request is safe to send
  RETURN TRUE
END FUNCTION
```

Use Conservative Quantity Values

When in doubt, request fewer registers:

```
Uncertain if registers 40090-40100 all exist?

Option 1 (risky): Read 40090-40100 (11 registers)
  If 40100 doesn't exist → Exception 02


Option 2 (safe): Read 40090-40095 (6 registers)
  Then read 40096-40100 (5 registers) separately
  If second request fails, only lose 5 registers worth of attempt
```

Document Sparse Maps

Create configuration files:

```
{
  "device": "Power Meter Model X",
  "holdingRegisters": {
    "process": { "start": 0, "end": 49 },
    "config": { "start": 200, "end": 249 },
    "diagnostic": { "start": 500, "end": 509 }
  }
}
```

Handling in Application Code

Detect and Report

```
TRY
  // Attempt to read the requested registers
  response = readHoldingRegisters(address, quantity)

  // If successful, process the data
  processData(response)


CATCH ModbusException AS e
  // Check if the specific error is "Illegal Data Address"
  IF e.exceptionCode IS 0x02 THEN
    // Provide a clear, user-friendly error message
    LOG ERROR "The requested address (" + address + ") is not valid for this device."
    LOG INFO "Please check the device's documentation for its memory map."
```

```
    // Optionally, send structured error data to a monitoring system
    REPORT_ERROR(
      type: &amp;quot;MODBUS_ILLEGAL_ADDRESS&amp;quot;,
      details: {
        &amp;quot;device&amp;quot;: deviceInfo,
        &amp;quot;address&amp;quot;: address,
        &amp;quot;quantity&amp;quot;: quantity
      }
    )
```

```
  ELSE
    // If it's a different Modbus exception, re-throw it
    THROW e
```

```
  END IF
END TRY
```

Automatic Discovery Fallback

```
FUNCTION readWithFallback(address, quantity)
  TRY
    // First, try to read the entire block as requested
    RETURN readHoldingRegisters(address, quantity)

  CATCH ModbusException AS e
    // If it fails with "Illegal Data Address" and we were reading more than one point
    IF e.exceptionCode IS 0x02 AND quantity > 1 THEN
      LOG WARN "Address range is invalid. Attempting to read each address individually."


    results = NEW LIST

    // Loop through each requested address one by one
    FOR i FROM 0 TO quantity - 1
      currentAddress = address + i
      TRY
        // Read a single register
        value = readHoldingRegisters(currentAddress, 1)
        ADD value TO results
      CATCH
        // If a single read fails, mark it as non-existent (null)
        ADD NULL TO results
      END TRY
    END FOR

    // Return the list of results, which may contain nulls for invalid addresses
    RETURN results

  ELSE
    // For other errors or single-point reads, re-throw the original exception
    THROW e
  END IF


  END TRY
END FUNCTION
```

Common Causes by Function Code

FC03 / FC04 (Read Registers)

- **Address doesn't exist** - Most common cause
- **Quantity too large** - Address + quantity > max address
- **Sparse memory map** - Gap in middle of requested range

FC05 / FC06 (Write Single)

- **Read-only address** - Attempting to write to input table
- **Address not writable** - Device-specific read-only addresses
- **Address doesn't exist** - Invalid address

FC15 / FC16 (Write Multiple)

- **Same as read** - Address or range invalid
- **Additional:** Partially valid range (some addresses exist, others don't)

Related Topics

Understanding Addresses:

- Addressing - Public vs protocol addressing, off-by-one
- Data Model - The four tables and their ranges

Other Exceptions:

- Exception 01 - Illegal Function
- Exception 03 - Illegal Data Value
- Exceptions Overview - All exception codes

Function Code Details:

- FC03 - Read Holding Registers
- FC06 - Write Single Register
- All Function Codes

Tools and Features:

- Discovery - Automatically find valid addresses
- Modbus Client - Testing addresses
- Troubleshooting - Solving communication problems

Reference:

- Glossary - Address
- Glossary - Exception Code

Exception 03 - Illegal Data Value

Exception Code: 0x03

Name: ILLEGAL DATA VALUE

Severity: ⚠️ Configuration Error - Request must be corrected

What It Means

A value contained in the request's data field is invalid. This is different from an illegal *address* (Exception 02); it refers to the *values* being sent, such as quantity, byte counts, or the data to be written.

This exception indicates that:

- A quantity or count field is outside its valid range (e.g., too large or zero).
- A byte count does not match the expected value based on the quantity.
- A value to be written to a register is invalid for that specific register's function (application-level validation).

When It Occurs

Scenario 1: Invalid Quantity

Many function codes have limits on the number of points that can be read or written.

Example: FC03 (Read Holding Registers)

```
Valid quantity range: 1 to 125
Client requests 200 registers.

Request:  [03][00 00][00 C8]
           FC  Addr=0  Qty=200
Response: [83][03]
          ErrFC ExCode


Why: The quantity (200) exceeds the maximum of 125 for this function.
```

Scenario 2: Mismatched Byte Count

For functions that write data, the `Byte Count` field must correctly correspond to the `Quantity` field.

Example: FC16 (Write Multiple Registers)

```
Rule: Byte Count = Quantity of Registers × 2

Client wants to write 2 registers, but sends an incorrect byte count.
Request:  [10][00 00][00 02][03][...data...]
           FC  Addr=0  Qty=2  ByteCt=3 (WRONG!)
Response: [90][03]
          ErrFC ExCode


Why: For 2 registers, the byte count must be 4 (2 × 2).
```

Correct Request:

```
Request:  [10][00 00][00 02][04][...4 bytes of data...]
           FC  Addr=0  Qty=2  ByteCt=4 (CORRECT)
Response: [10][00 00][00 02]
          ✓ Success
```

Scenario 3: Invalid Application-Level Value

A server can perform its own validation on the data it receives. Even if the request is structurally correct, the value itself might be unacceptable.

Example: Writing to an Enumerated Register

```
A device has a register (40100) that controls its operating mode.
Valid values:
- 0 = Off
- 1 = On
- 2 = Standby

Client attempts to write an unsupported value.
Request:  [06][00 63][00 05]
           FC=06 Addr=99 Value=5
Response: [86][03]
          ErrFC ExCode


Why: The value '5' is not a valid operating mode for this register.
```

Scenario 4: Invalid Sub-function

For functions that use sub-function codes, like FC08 - Diagnostics, an unsupported sub-function will trigger Exception 03.

Example: FC08 (Diagnostics)

```
Device supports sub-functions 0 (Return Query Data) and 1 (Restart).
Client requests a non-existent sub-function.
```

```
Request:  [08][00 FF][DE AD]
           FC  SubFunc=255 Data
Response: [88][03]
          ErrFC ExCode


Why: Sub-function 255 is not implemented by the server.
```

Function Codes That Can Return Exception 03

Any function with data fields that have constraints can return this exception.

- **FC03 / FC04:** Invalid quantity.
- **FC06:** Invalid register value (application-level).
- **FC08:** Invalid sub-function code.
- **FC15 / FC16:** Invalid quantity or mismatched byte count.
- **FC23:** Invalid quantity or mismatched byte count for either read or write part.

Complete Example

FC16 Write Multiple Registers - Mismatched Byte Count

Modbus TCP Request:

```
[00 0B][00 00][00 08][01] [10][00 01][00 02][03][12 34]
 Trans  Proto  Len   Unit  FC  Addr   Qty    ByteCt Data

Transaction ID: 11
Function: 16 (Write Multiple Registers)
Address: 1
Quantity: 2 registers
Byte Count: 3 (This is incorrect for 2 registers)
Data: 0x1234 (This part is irrelevant as the byte count is checked first)
```

Modbus TCP Exception Response:

```
[00 0B][00 00][00 03][01] [90][03]
 Trans  Proto  Len   Unit  ErrFC ExCode

Transaction ID: 11 (matches request)
Exception Function: 0x90 (0x10 + 0x80)
Exception Code: 0x03 (Illegal Data Value)


Meaning: The byte count of 3 is invalid for a request to write 2 registers.
```

Troubleshooting Steps

Step 1: Check Quantity Limits

- Consult the documentation for the specific function code being used.
- Ensure the `Quantity` field is within the allowed min/max range.
    - **Reads (FC01-04):** 1-2000 (coils/inputs), 1-125 (registers)
    - **Writes (FC15-16):** 1-1968 (coils), 1-123 (registers)

Step 2: Verify Byte Count Calculation

- For write functions, double-check the formula for the byte count.
    - **FC15 (Coils):** `Byte Count = ceil(Quantity of Coils / 8)`
    - **FC16 (Registers):** `Byte Count = Quantity of Registers × 2`

Step 3: Validate Application-Level Data

- If quantities and byte counts are correct, the issue is likely the data itself.
- Check the device manual for the valid range or accepted values for the target register.
- For enumerated registers, ensure you are using one of the listed values.
- For numerical registers, check for min/max limits (e.g., a setpoint might be limited to 0-1000).

Prevention Strategies

Client-Side Validation

Implement checks in the client application before sending a request.

```
FUNCTION createWriteMultipleRegistersRequest(startAddress, values)
  quantity = length(values)

  // Check quantity limit for FC16
  IF quantity < 1 OR quantity > 123 THEN
    THROW ERROR "Invalid quantity for FC16. Must be between 1 and 123."
  END IF

  // Calculate the correct byte count
  byteCount = quantity * 2

  // Validate each value before adding it to the payload
  FOR each value IN values
```

```
    // Get the valid range for the target register from a device profile
    validRange = getRegisterProfile(startAddress).validRange
    IF value < validRange.min OR value > validRange.max THEN
      THROW ERROR "Value " + value + " is out of range for register " + startAddress
    END IF
    startAddress = startAddress + 1
  END FOR

  // If all checks pass, build and send the request
  // ...
END FUNCTION
```

Use High-Level Libraries

Use Modbus libraries that handle the calculation of quantities and byte counts automatically. This eliminates a common source of errors.

Related Topics

Other Exceptions:

- Exception 01 - Illegal Function
- Exception 02 - Illegal Data Address
- Exceptions Overview - All exception codes

Function Code Details:

- FC16 - Write Multiple Registers
- FC08 - Diagnostics

Exception 04 - Server Device Failure

Exception Code: 0x04

Name: SERVER DEVICE FAILURE

Severity: 🔥 Critical Error - May require device intervention

What It Means

The server encountered an unrecoverable error while attempting to perform the action requested in the query. This is a catch-all exception for internal problems within the server device that are not related to the format of the request itself.

This exception indicates a problem with the server device, such as:

- A hardware fault (e.g., memory parity error, I/O failure).
- A firmware crash or internal software error.

- The device is stuck in an unrecoverable state.

This is one of the most serious exceptions, as it implies the request was valid, but the device was physically unable to complete it.

When It Occurs

This exception is not tied to specific request values but rather to the internal state of the server.

Scenario 1: Hardware Fault

A common cause is a failure in the device's non-volatile memory or I/O subsystem.

Example:

```
A client sends a valid FC16 (Write Multiple Registers) request to save new configuration parameters.
The server attempts to write these to its flash memory, but the write operation fails due to a
hardware fault.

Request:  [10][01 00][00 0A][14][...data...]
           FC  Addr=256 Qty=10 ByteCt=20
Response: [90][04]
          ErrFC ExCode


Why: The server's flash memory is corrupted or has failed, preventing it from saving the new
configuration. The request was valid, but the hardware could not execute it.
```

Scenario 2: Firmware Error

An internal bug or unexpected state in the server's firmware can lead to this exception.

Example:

```
A client requests a diagnostic report using FC08. The server's firmware enters an infinite loop or
encounters a null pointer exception while trying to gather the diagnostic data. After a timeout, it
returns a failure exception.

Request:  [08][00 02][00 00]
           FC  Sub-func=2
Response: [88][04]
          ErrFC ExCode


Why: A software bug prevented the device from successfully completing an otherwise valid request.
```

Function Codes That Can Return Exception 04

Any function code can return this exception. It is not specific to any particular operation but rather reflects the overall health of the server device.

Complete Example

FC03 Read Holding Registers - Device Failure

Modbus TCP Request:

```
[00 0C][00 00][00 06][01] [03][00 00][00 01]
 Trans  Proto  Len   Unit  FC  Addr   Qty

Transaction ID: 12
Function: 03 (Read Holding Registers)
Address: 0
Quantity: 1


Assumption: The server's I/O board that reads the sensor connected to this register has failed.
```

Modbus TCP Exception Response:

```
[00 0C][00 00][00 03][01] [83][04]
 Trans  Proto  Len   Unit  ErrFC ExCode

Transaction ID: 12 (matches request)
Exception Function: 0x83 (0x03 + 0x80)
Exception Code: 0x04 (Server Device Failure)


Meaning: The server could not read the register due to an internal hardware or software fault.
```

Troubleshooting Steps

Troubleshooting this exception is difficult from the client's perspective because the problem lies within the server.

Step 1: Retry the Request

- Send the exact same request again after a short delay (e.g., 1-5 seconds).
- If the error was transient, the subsequent request might succeed.
- If the error persists after several retries, it indicates a hard fault.

Step 2: Check Device Status Indicators

- Look at the physical device for any fault LEDs (e.g., red or amber lights).
- Consult the device manual to interpret the meaning of any status lights.

Step 3: Power Cycle the Device

- A hard reset (powering the device off and on again) can sometimes clear internal faults.
- **Caution:** This may result in the loss of non-persistent data or configuration.

Step 4: Query Basic Information

- Try to read a simple, static value, such as the vendor name via FC43 - Read Device Identification.
- If even the most basic requests fail with Exception 04, the device is likely in a critical failure state.

Step 5: Contact Manufacturer

- If the device consistently returns Exception 04 and cannot be recovered via a power cycle, it likely requires repair or replacement.
- Provide the manufacturer with the context of the failure (what command was being sent, any physical status indicators).

Handling in Application Code

Retry Logic

Implement a retry mechanism with a limited number of attempts.

```
FUNCTION readWithRetry(address, quantity, maxRetries = 3)
  retries = 0
  WHILE retries < maxRetries
    TRY
      // Attempt the read operation
      RETURN readHoldingRegisters(address, quantity)

  CATCH ModbusException AS e
    // Check for Server Device Failure
    IF e.exceptionCode IS 0x04 THEN
      retries = retries + 1
      LOG WARN &amp;quot;Server device failure detected. Retry &amp;quot; + retries +
  &amp;quot;/&amp;quot; + maxRetries
      WAIT 2 seconds // Wait before retrying
    ELSE
      // For any other exception, fail immediately
      THROW e
    END IF
  END TRY


  END WHILE
```

```
  // If all retries fail, throw a final exception
  THROW ERROR "Server device failed to respond after " + maxRetries + " retries."
END FUNCTION
```

Alerting and Monitoring

Since this is a critical error, the client application should generate a high-priority alert.

```
// Inside the catch block for Exception 04
LOG CRITICAL "Unrecoverable error reported by Modbus server at address " + serverAddress
CREATE_HIGH_PRIORITY_ALARM(
  type: "MODBUS_SERVER_FAILURE",
  message: "Server " + serverAddress + " reported Exception 04: Server Device Failure.",
  details: {
    "request": lastRequestSent
  }
)
```

Related Topics

Other Exceptions:

- Exception 05 - Acknowledge - For long-duration commands.
- Exception 06 - Server Device Busy - A recoverable "busy" state.
- Exceptions Overview - All exception codes.

Exception 05 - Acknowledge

Exception Code: 0x05

Name: ACKNOWLEDGE

Severity: informational - Part of a normal workflow for long-running commands

What It Means

The server has accepted the request and has started processing it, but the operation will take a significant amount of time to complete. This is **not an error**. It is a special acknowledgement that prevents the client from timing out while waiting for a long-duration task to finish.

The server sends this exception to say: "I got your request and it's valid. Come back later to check on the progress."

When It Occurs

This exception is specifically used for commands that take a long time, such as:

- Programming a device.
- Erasing or writing to large blocks of flash memory.
- Running a complex self-test or calibration routine.

Example: Device Programming

```
A client sends a command to a PLC to download a new program. This operation might take 30 seconds.
```

1. Client sends "Program Device" command (e.g., using a proprietary function code).
   Request: [41][...programming data...]

2. Server receives the request, validates it, and begins the programming sequence. It immediately responds with Exception 05.
   Response: [C1][05]
         ErrFC ExCode
   This tells the client that the command was accepted.

3. The client now knows not to expect an immediate success response. Instead, it should start polling the server's status.

4. Client periodically sends FC11 (Get Comm Event Counter) to check if the server is still busy.
   Request: [0B]
   Response (while busy): [8B][FF FF][...event count...]
                      Status=Busy

5. After 30 seconds, the server finishes programming. The next time the client polls with FC11, the server indicates it is ready.
   Response (when done): [8B][00 00][...new event count...]
                     Status=Ready

6. The client can now resume normal communication.

How to Handle It

This exception requires a specific workflow in the client application.

Client Workflow

1. **Send Long-Duration Command:** The client sends the initial request that is expected to take a long time.
2. **Receive Exception 05:** The client's Modbus stack should be configured to treat Exception 05 not as an error, but as a trigger for a polling loop.
3. **Enter Polling State:** Upon receiving Exception 05, the client should:
    - Stop waiting for a final response to the original command.
    - Start a periodic polling timer (e.g., every 1-2 seconds).
4. **Poll for Status:** In the polling loop, the client should send a status-checking command. The standard function for this is **FC11 - Get Comm Event Counter**.
    - The `Status` field in the FC11 response will be `0xFFFF` (Busy) while the operation is in progress.
5. **Exit Polling State:** The client continues polling until the `Status` field in the FC11 response is `0x0000` (Ready).
6. **Resume Normal Operation:** Once the server is ready, the client can continue with other Modbus transactions.

Pseudocode for Client Logic

```
FUNCTION executeLongCommand(command)
  TRY
 // Send the initial command
 sendModbusRequest(command)

  CATCH ModbusException AS e
    // Check for the "Acknowledge" exception
    IF e.exceptionCode IS 0x05 THEN
      LOG INFO "Server acknowledged long-running command. Entering polling mode."


    // Start polling until the server is no longer busy
    WHILE isServerBusy()
      WAIT 2 seconds
    END WHILE

    LOG INFO &amp;quot;Server has completed the command.&amp;quot;
    RETURN &amp;quot;SUCCESS&amp;quot;

  ELSE
    // If it&amp;#39;s a different exception, treat it as an error
    THROW e
  END IF


END TRY
```

```
FUNCTION isServerBusy()
  // Use FC11 to get the server's status
  response = readCommEventCounter() // Sends FC11

  // The status is 0xFFFF if busy, 0x0000 if ready
  RETURN response.status IS 0xFFFF
END FUNCTION
```

Related Topics

Other Exceptions:

- Exception 04 - Server Device Failure - An unrecoverable error.
- Exception 06 - Server Device Busy - A temporary busy state where the server asks the client to retry the *same* request later.
- Exceptions Overview - All exception codes.

Associated Function Codes:

- FC11 - Get Comm Event Counter - The standard way to poll for status after receiving Exception 05.

Exception 06 - Server Device Busy

Exception Code: 0x06

Name: SERVER DEVICE BUSY

Severity: Informational - Client should retry later

What It Means

The server is busy processing a long-duration program command. The server returns this exception to indicate that it cannot handle a new request at this moment. The client should retry the same request again after a short delay.

This exception is closely related to Exception 05 - Acknowledge, but they occur at different stages of a long-duration command sequence.

When It Occurs

This exception occurs when a client sends a request to a server that is already in the middle of executing a long-running task initiated by a previous command.

Typical Workflow

1. **Client A sends a long-duration command** (e.g., "Program Device").

2. **Server accepts the command** and responds to Client A with **Exception 05 - Acknowledge**. The server is now in a "busy" state.
3. **Client B (or Client A) sends a new request** (e.g., a simple `FC03 - Read Holding Registers`) while the server is still busy programming.
4. **Server rejects the new request** by responding with **Exception 06 - Server Device Busy**.

This tells the requesting client: "I cannot process your request right now because I am busy with something else. Please ask again later."

Example:

```
// Step 1 & 2: A long command is already running
Client A -> [Program Command] -> Server
Server   -> [Exception 05]    -> Client A

// The Server is now busy for the next 30 seconds.


// Step 3 & 4: A new request arrives while the server is busy
Client B -> [FC03 Request] -> Server
Server   -> [83][06]       -> Client B (FC03 + 0x80, Exception 06)
```

How to Handle It

The client's response to Exception 06 should be simple: **wait and retry**.

Client Workflow

1. **Send Request:** The client sends any Modbus request.
2. **Receive Exception 06:** The client receives the "Server Device Busy" exception.
3. **Wait:** The client should wait for a short, predefined period (e.g., 500ms to 2 seconds).
4. **Retry:** The client should send the **exact same request** again.
5. **Repeat:** Continue this wait-and-retry loop until the request is successful or a maximum number of retries is reached.

Pseudocode for Client Logic

```
FUNCTION sendRequestWithBusyHandling(request, maxRetries = 10)
  retries = 0
  WHILE retries < maxRetries
    TRY
      // Attempt to send the request
      response = sendModbusRequest(request)

      // If it succeeds, return the response
      RETURN response
```

```
    CATCH ModbusException AS e
      // Check if the specific error is &amp;quot;Server Device Busy&amp;quot;
      IF e.exceptionCode IS 0x06 THEN
        retries = retries + 1
        LOG INFO &amp;quot;Server is busy. Retrying in 1 second... (Attempt &amp;quot; +
    retries + &amp;quot;)&amp;quot;
        WAIT 1 second
      ELSE
        // For any other exception, fail immediately
        THROW e
      END IF
    END TRY


  END WHILE


  // If all retries fail, throw a final timeout error
  THROW ERROR "Request failed after " + maxRetries + " retries. Server remained busy."
END FUNCTION
```

Difference Between Exception 05 and 06

These two exceptions can be confusing, but they serve different purposes in the same overall process.

Feature	Exception 05 (Acknowledge)	Exception 06 (Server Device Busy)
Meaning	"I have **accepted** your long-running command and am starting it now."	"I **cannot accept** your new command because I am already busy."
Client Action	Start polling for status (e.g., with FC11). Do **not** resend the original command.	Wait and **retry the same command** later.
Who Receives It	The client that initiated the long-running command.	Any client that sends a request while the server is busy.
Analogy	You order a custom-made pizza and are told, "It will be ready in 20 minutes."	You call a restaurant, and the line is busy, so you hang up and call again.

Related Topics

Other Exceptions:

- Exception 05 - Acknowledge - The initial response to a long-duration command.
- Exception 04 - Server Device Failure - A critical, unrecoverable error.
- Exceptions Overview - All exception codes.

Associated Function Codes:

- FC11 - Get Comm Event Counter - Used to poll for status after an "Acknowledge" response.

Exception 07 - Negative Acknowledge

Exception Code: 0x07

Name: NEGATIVE ACKNOWLEDGE

Severity: ⚠️ Warning - Command rejected

What It Means

This exception is specialized for programming commands. It means the server has received a request to perform a programming function but cannot currently perform it. This is different from Exception 05 - Acknowledge, which indicates the command has been accepted.

"Negative Acknowledge" means "I understand your request to program me, but I cannot do it right now."

When It Occurs

This exception is returned only in response to programming-related function codes. The reasons for rejection are device-specific but can include:

- The device is not in the correct mode for programming.
- A required prerequisite step has not been completed.
- The programming request is invalid for reasons other than data value or address (which would be Exception 02 or 03).

Example:

```
A client sends a "Write Program" command to a device, but the device's memory is locked or write-protected.

Request:  [Program Command]
Response: [ErrFC][07]


Why: The server understood the command but rejected it because its memory is locked. The client must first send an "Unlock Memory" command before retrying.
```

How to Handle It

The client's action depends on the specific device's programming protocol.

1. **Consult Device Manual:** The manual is essential to understand why a programming command would be rejected.
2. **Check Prerequisites:** The client may need to send one or more preliminary commands (e.g., "Enter Program Mode", "Unlock Memory").

3. **Correct and Retry:** After performing the required prerequisite steps, the client can send the original programming command again.

Related Topics

- Exception 05 - Acknowledge - When the programming command is accepted.
- Exceptions Overview - All exception codes.

Exception 08 - Memory Parity Error

Exception Code: 0x08

Name: MEMORY PARITY ERROR

Severity: 🔥 Critical Error - Potential hardware fault

What It Means

The server attempted to read data from its internal memory as part of processing the request, but it detected a memory parity error. This indicates a hardware-level problem with the server's memory (RAM or ROM).

This is a more specific version of Exception 04 - Server Device Failure.

When It Occurs

This error happens when the server's hardware self-diagnostics detect data corruption.

Example:

```
A client sends a valid FC03 request to read a holding register. The server's CPU attempts to fetch the
value from the specified memory address. The memory hardware reports a parity error, meaning the data
at that location is corrupt.

Request:  [03][00 0A][00 01]
           FC  Addr=10 Qty=1
Response: [83][08]
          ErrFC ExCode


Why: The server could not reliably read the data from its own memory to fulfill the request.
```

How to Handle It

From the client's perspective, this should be treated as a serious server fault.

1. **Retry:** The client should retry the request a few times. If the error was transient, it might succeed.
2. **Log Critical Error:** If the error persists, the client should log a critical alarm. This is not a client or configuration issue.
3. **Device Intervention:** The server device likely has a hardware fault. It may need to be power-cycled, reset to factory defaults, or replaced.

Client Logic

```
TRY
  // Send a read request
  response = readHoldingRegisters(address, quantity)
CATCH ModbusException AS e
  IF e.exceptionCode IS 0x08 THEN
    // Log a high-priority alarm for operator intervention
    LOG CRITICAL "Server at " + serverAddress + " reported Memory Parity Error."
    // Optionally, attempt a limited number of retries before failing
  END IF
END TRY
```

Related Topics

- Exception 04 - Server Device Failure - A more general server error.
- Exceptions Overview - All exception codes.

Exception 10 (0x0A) - Gateway Path Unavailable

Exception Code: 0x0A (10 decimal)

Name: GATEWAY PATH UNAVAILABLE

Severity: ⚠️ Warning - Network/Gateway issue

What It Means

This exception is specific to Modbus gateways (e.g., Modbus TCP to Modbus RTU). It means the gateway received the request but could not establish a communication path to the final target device on the downstream serial network.

This is **not** a problem with the end device itself, but rather with the gateway's ability to reach it.

When It Occurs

This typically happens in a Modbus TCP to RTU gateway scenario.

Example:

```
Client (TCP) -> Gateway (TCP to RTU) -> End Device (RTU)

The client sends a valid request to the gateway, addressed to Unit ID 5 on the serial link.


Request: [MBAP Header Unit=5][FC03 Request]


The gateway receives this, but its serial port is misconfigured, disconnected, or already busy
handling another request on a different network path. The gateway cannot forward the request to Unit
ID 5.


The gateway then responds to the client with Exception 10.


Response: [MBAP Header Unit=5][83][0A]
                            ErrFC ExCode
```

Common Causes:

- The gateway's serial port is not properly configured (baud rate, parity, etc.).
- The physical serial cable (e.g., RS-485) is disconnected from the gateway.
- The gateway is overloaded and has no available internal resources to handle the request.
- A misconfigured router or switch is preventing the gateway from accessing the required network segment.

How to Handle It

This is a network infrastructure problem.

1. **Check Gateway Configuration:** Verify that the gateway's network settings and serial port settings are correct.
2. **Inspect Physical Connections:** Ensure all network cables and serial lines are securely connected to the gateway.
3. **Reduce Gateway Load:** If the gateway is handling many requests, the client may need to slow down its polling rate.
4. **Retry:** The client should implement a retry mechanism with a delay, as the path may become available.

Difference Between Exception 10 and 11

Exception	Problem Location	Meaning
10 - Path Unavailable	**Gateway**	"I (the gateway) cannot even send the message to the end device."
11 - Target Failed to Respond	**End Device**	"I (the gateway) sent the message, but the end device never answered."

Related Topics

- Exception 11 - Gateway Target Device Failed to Respond
- Exceptions Overview - All exception codes.

Exception 11 (0x0B) - Gateway Target Device Failed to Respond

Exception Code: 0x0B (11 decimal)

Name: GATEWAY TARGET DEVICE FAILED TO RESPOND

Severity: ⚠️ Warning - End device or network issue

What It Means

This exception is specific to Modbus gateways. It means the gateway successfully forwarded the request to the target device on the downstream network (e.g., a serial RTU link), but the target device did not send a response back within the expected timeout period.

This indicates a problem with the **end device** or the **network segment between the gateway and the end device**, not with the gateway itself.

When It Occurs

This is a very common error in Modbus TCP-to-RTU setups.

Example:

```
Client (TCP) -> Gateway (TCP to RTU) -> End Device (RTU, Unit ID 5)


  1. The client sends a valid request to the gateway, addressed to Unit ID 5.
     Request: [MBAP Header Unit=5][FC03 Request]


  2. The gateway receives the request and successfully transmits it onto the serial RTU network.


  3. The gateway waits for a response from Unit ID 5, but nothing is received before its timeout
     period expires.


  4. The gateway gives up and sends Exception 11 back to the original client.
     Response: [MBAP Header Unit=5][83][0B]
                           ErrFC ExCode
```

Common Causes:

- The end device with the specified Unit ID does not exist on the serial link.
- The end device is powered off or has failed.
- The end device's serial configuration (baud rate, parity) does not match the gateway's.
- The serial wiring is faulty (e.g., swapped A/B lines on RS-485).
- The end device is too slow to respond, and the gateway's timeout is too short.

How to Handle It

This error points to a problem "downstream" from the gateway.

1. **Verify End Device:**
    - Ensure the device with the target Unit ID exists and is powered on.
    - Check for any fault indicators on the device itself.
2. **Check Serial Configuration:**
    - Confirm that the baud rate, parity, and stop bits match between the gateway and all devices on the serial link.
3. **Inspect Serial Wiring:**
    - Check for loose connections or damage to the cable.
    - For RS-485, ensure the A/B lines are connected correctly.
4. **Adjust Gateway Timeout:**
    - If the end device is known to be slow, increase the "Slave Response Timeout" or similar setting in the gateway's configuration.
5. **Isolate the Device:**
    - If there are multiple devices on the serial link, disconnect all but the problematic one to see if communication can be established. This helps rule out conflicts.

Difference Between Exception 10 and 11

Exception	Problem Location	Meaning
10 - Path Unavailable	**Gateway**	"I (the gateway) couldn't even send the message."

11 - Target Failed to Respond	**End Device**	"I (the gateway) sent the message, but the target device ignored me."

Related Topics

- Exception 10 - Gateway Path Unavailable
- Exceptions Overview - All exception codes.

FC01 - Read Coils

Function Code: 0x01 (1 decimal)

Purpose: Reads the ON/OFF status of 1 to 2000 contiguous discrete outputs (coils) in a server device.

Use Case: Used to get the current state of digital outputs, such as relays, switches, or lamps.

Overview

FC01 reads the status of coils, which are single-bit read/write values. Each coil can be either ON (1) or OFF (0).

See: Data Model - Coils

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Starting Address │ Quantity of Coils│
│ 0x01         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x01	Fixed value
Starting Address	2 bytes	0x0000 - 0xFFFF	First coil to read (0-based)
Quantity of Coils	2 bytes	1 - 2000	Number of consecutive coils

Valid Quantity Range: 1 to 2000 coils (0x07D0)

Response Format

PDU Structure:

```
┌──────────────┬──────────────┬──────────────────────────┐
│ Function Code│ Byte Count   │ Coil Status              │
│ 0x01         │ (1 byte)     │ (N bytes)                │
│ (1 byte)     │              │                          │
└──────────────┴──────────────┴──────────────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x01 (echoed from request)
Byte Count	1 byte	Number of data bytes to follow (ceil(Quantity / 8))
Coil Status	N bytes	Coil data packed into bytes

Bit Packing:

- Coils are packed into bytes, with the first coil at the least significant bit (LSB) of the first byte.
- If the quantity is not a multiple of 8, the remaining bits in the last byte are padded with zeros.

Example Packing (10 coils):

```
Byte 1: [Coil 8, Coil 7, Coil 6, Coil 5, Coil 4, Coil 3, Coil 2, Coil 1]
Byte 2: [  0,     0,     0,     0,     0,     0,   Coil 10, Coil 9]
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
         (MSB)                                                (LSB)
```

Complete Example

Example: Read 10 Coils

Scenario: Read 10 coils starting at public address 1 (protocol address 0).

Address Conversion:

```
Public Address:   1
Point Number:     1
Protocol Address: 0 (0x0000)
```

See: Addressing

Modbus TCP Request:

```
┌─── MBAP Header ──────┬──────── PDU ────────┐
[00 01][00 00][00 06][01] [01][00 00][00 0A]
 Trans  Proto  Len   Unit  FC  Addr   Qty=10

Breakdown:
  Function:       0x01
  Start Address:  0x0000 (Coil 1)
  Quantity:       0x000A (10 coils)
```

Modbus TCP Response:

Assume the following coil states:

- Coils 1, 3, 5, 8 are ON
- Coils 2, 4, 6, 7, 9, 10 are OFF

```
┌─── MBAP Header ──────┬──────── PDU ────────┐
[00 01][00 00][00 05][01] [01][02][85][01]
 Trans  Proto  Len   Unit  FC  ByteCt Data

Breakdown:
  Function:       0x01
  Byte Count:     0x02 (ceil(10 / 8) = 2 bytes)
  Data Byte 1:    0x85 (binary 10000101)
  Data Byte 2:    0x01 (binary 00000001)


Data Interpretation:
  Byte 1 (0x85): 1000 0101
    Bit 0 (Coil 1): 1 (ON)
    Bit 1 (Coil 2): 0 (OFF)
    Bit 2 (Coil 3): 1 (ON)
    Bit 3 (Coil 4): 0 (OFF)
    Bit 4 (Coil 5): 1 (ON)
    Bit 5 (Coil 6): 0 (OFF)
    Bit 6 (Coil 7): 0 (OFF)
    Bit 7 (Coil 8): 1 (ON)


  Byte 2 (0x01): 0000 0001
    Bit 0 (Coil 9):  1 (ON) -> Wait, this should be OFF. Let's correct the example.
    Bit 1 (Coil 10): 0 (OFF)
```

Corrected Example Response: Let's assume Coils 1, 3, 5, 8 are ON, and the rest are OFF.

- Coils 1-8: `10000101` binary = `0x85`
- Coils 9-10: `00000000` binary = `0x00` (since 9 and 10 are OFF)

Correct Modbus TCP Response:

```
[00 01][00 00][00 05][01] [01][02][85][00]
```

Exception Responses

FC01 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC01. **Response:** `[81][01]`

Exception 02 - Illegal Data Address

Cause:

- Starting address doesn't exist.
- Starting address + quantity exceeds available coils. **Response:** `[81][02]`

Exception 03 - Illegal Data Value

Cause:

- Quantity is 0 or > 2000. **Response:** `[81][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[81][04]`

Best Practices

- **Batch Reads:** Read multiple coils in a single request to reduce network traffic.
- **Validate Quantity:** Ensure the requested quantity is within the valid range of 1-2000.
- **Handle Byte/Bit Unpacking:** Be careful when unpacking the bit data from the response bytes. Remember the LSB of the first byte corresponds to the first coil address.

Related Topics

- **Data Model:** Coils
- **Writing Coils:** FC05 - Write Single Coil, FC15 - Write Multiple Coils
- **Read-Only Bits:** FC02 - Read Discrete Inputs
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC02 - Read Discrete Inputs

Function Code: 0x02 (2 decimal)

Purpose: Reads the ON/OFF status of 1 to 2000 contiguous discrete inputs in a server device.

Use Case: Used to get the current state of digital inputs, such as sensors, contacts, or buttons. This data is read-only.

Overview

FC02 reads the status of discrete inputs, which are single-bit read-only values. Each input can be either ON (1) or OFF (0). This function is nearly identical to FC01 - Read Coils, but it reads from a different data table.

See: Data Model - Discrete Inputs

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Starting Address │ Quantity of Inputs│
│ 0x02         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x02	Fixed value
Starting Address	2 bytes	0x0000 - 0xFFFF	First input to read (0-based)
Quantity of Inputs	2 bytes	1 - 2000	Number of consecutive inputs

Valid Quantity Range: 1 to 2000 inputs (0x07D0)

Response Format

PDU Structure:

```
┌──────────────┬──────────────┬──────────────────────────┐
│ Function Code│ Byte Count   │ Input Status             │
│ 0x02         │ (1 byte)     │ (N bytes)                │
│ (1 byte)     │              │                          │
└──────────────┴──────────────┴──────────────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x02 (echoed from request)
Byte Count	1 byte	Number of data bytes to follow (ceil(Quantity / 8))
Input Status	N bytes	Input data packed into bytes

Bit Packing:

- The packing is identical to FC01 - Read Coils. Inputs are packed into bytes, with the first input at the least significant bit (LSB) of the first byte.
- If the quantity is not a multiple of 8, the remaining bits in the last byte are padded with zeros.

Complete Example

Example: Read 12 Discrete Inputs

Scenario: Read 12 discrete inputs starting at public address 10001 (protocol address 0).

Address Conversion:

```
Public Address:   10001
Point Number:     1
Protocol Address: 0 (0x0000)
```

See: Addressing

Modbus TCP Request:

```
┌─── MBAP Header ──────┬──── PDU ─────┐
[00 02][00 00][00 06][01] [02][00 00][00 0C]
 Trans  Proto  Len   Unit  FC  Addr   Qty=12

Breakdown:
  Function:       0x02
  Start Address:  0x0000 (Input 10001)
  Quantity:       0x000C (12 inputs)
```

Modbus TCP Response:

Assume the following input states:

- Inputs 10001, 10002, 10005, 10011 are ON
- All others are OFF

```
┌─── MBAP Header ──────┬──────── PDU ─────┐
[00 02][00 00][00 05][01] [02][02][13][04]
 Trans  Proto  Len   Unit  FC  ByteCt Data
```

```
Breakdown:
  Function:       0x02
  Byte Count:     0x02 (ceil(12 / 8) = 2 bytes)
  Data Byte 1:    0x13 (binary 00010011)
  Data Byte 2:    0x04 (binary 00000100)


Data Interpretation:
  Byte 1 (0x13): 0001 0011
    Bit 0 (Input 10001): 1 (ON)
    Bit 1 (Input 10002): 1 (ON)
    Bit 2 (Input 10003): 0 (OFF)
    Bit 3 (Input 10004): 0 (OFF)
    Bit 4 (Input 10005): 1 (ON)
    Bit 5 (Input 10006): 0 (OFF)
    Bit 6 (Input 10007): 0 (OFF)
    Bit 7 (Input 10008): 0 (OFF)


  Byte 2 (0x04): 0000 0100
    Bit 0 (Input 10009):  0 (OFF)
    Bit 1 (Input 10010): 0 (OFF)
    Bit 2 (Input 10011): 1 (ON)
    Bit 3 (Input 10012): 0 (OFF)
```

Exception Responses

FC02 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC02. **Response:** `[82][01]`

Exception 02 - Illegal Data Address

Cause:

- Starting address doesn't exist.
- Starting address + quantity exceeds available inputs. **Response:** `[82][02]`

Exception 03 - Illegal Data Value

Cause:

- Quantity is 0 or > 2000. **Response:** `[82][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[82][04]`

Best Practices

- **Use for Read-Only Data:** This function is intended for reading the status of physical inputs that cannot be changed by the Modbus master.
- **Batch Reads:** Like FC01 - Read Coils, read multiple inputs in a single request to improve efficiency.
- **Verify Device Support:** Ensure the device you are communicating with uses the discrete input table. Some devices map all data to holding registers.

Related Topics

- **Data Model:** Discrete Inputs
- **Read/Write Bits:** FC01 - Read Coils
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC03 - Read Holding Registers

Function Code: 0x03 (3 decimal)

Purpose: Reads the contents of contiguous holding registers from a server device.

This is the most commonly used Modbus function code because holding registers are versatile and can represent any type of data.

Overview

FC03 is used to read 16-bit register values from the holding register table. Holding registers typically contain:

- Process measurements and values
- Setpoints and configuration parameters
- Counters and accumulators
- Any data that can be read or written

See: Data Model - Holding Registers

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
| Function Code| Starting Address  | Quantity of Regs |
| 0x03         | (2 bytes)         | (2 bytes)        |
```

```
| (1 byte)     |                  |                  |
└──────────────┴──────────────────┴──────────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x03	Fixed value
Starting Address	2 bytes	0x0000 - 0xFFFF	First register to read (0-based)
Quantity of Registers	2 bytes	1 - 125	Number of consecutive registers

Valid Quantity Range: 1 to 125 registers

Why 125 Maximum?

- Maximum PDU size: 253 bytes
- Response overhead: 1 (FC) + 1 (byte count) = 2 bytes
- Data per register: 2 bytes
- Maximum data: 253 - 2 = 251 bytes
- Maximum registers: 251 ÷ 2 = 125.5 → **125 registers**

Response Format

PDU Structure:

```
┌──────────────┬──────────────┬──────────────────────────┐
| Function Code| Byte Count   | Register Values          |
| 0x03         | (1 byte)     | (N × 2 bytes)            |
| (1 byte)     |              |                          |
└──────────────┴──────────────┴──────────────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x03 (echoed from request)
Byte Count	1 byte	Number of data bytes to follow (Quantity × 2)
Register Values	N bytes	Register data (2 bytes per register, big-endian)

Byte Encoding:

- Each register value: 2 bytes, big-endian (high byte first)
- Register N is followed by register N+1, etc.

Complete Example

Example 1: Read Two Holding Registers

Scenario: Read 2 holding registers starting at 40108 (public address)

Address Conversion:

```
Public Address:   40108
Point Number:     108
Protocol Address: 107 (0x006B)
```

See: Addressing for conversion details

Modbus TCP Request:

```
┌─── MBAP Header ──────┬──── PDU ─────┐
[00 01][00 00][00 06][01] [03][00 6B][00 02]
 Trans  Proto  Len   Unit  FC  Addr   Qty
  ID     ID

Breakdown:
  Transaction ID: 0x0001
  Protocol ID:    0x0000
  Length:         0x0006 (6 bytes follow)
  Unit ID:        0x01
  Function:       0x03
  Start Address:  0x006B (107 decimal = register 40108)
  Quantity:       0x0002 (2 registers)
```

Modbus TCP Response:

```
┌─── MBAP Header ──────┬──────── PDU ─────────┐
[00 01][00 00][00 07][01] [03][04][02 2B][00 00]
 Trans  Proto  Len   Unit  FC  ByteCt Reg107 Reg108

Breakdown:
  Transaction ID: 0x0001 (matches request)
  Protocol ID:    0x0000
  Length:         0x0007 (7 bytes follow)
  Unit ID:        0x01
  Function:       0x03
  Byte Count:     0x04 (4 bytes = 2 registers × 2)
  Register 107:   0x022B (555 decimal)
  Register 108:   0x0000 (0 decimal)

Results:
  Holding Register 40108 = 555
  Holding Register 40109 = 0
```

Modbus RTU Request:

```
[01][03][00 6B][00 02][B4 C5]
 Unit FC  Addr   Qty   CRC

Breakdown:
  Unit ID:        0x01
  Function:       0x03
  Start Address:  0x006B
  Quantity:       0x0002
  CRC-16:         0xC5B4 (transmitted LSB first as B4 C5)
```

Modbus RTU Response:

```
[01][03][04][02 2B][00 00][63 3A]
 Unit FC  ByteCt Reg107 Reg108 CRC

Breakdown:
  Unit ID:        0x01
  Function:       0x03
  Byte Count:     0x04
  Register 107:   0x022B (555)
  Register 108:   0x0000 (0)
  CRC-16:         0x3A63 (transmitted as 63 3A)
```

Example 2: Read Ten Registers

Scenario: Read registers 40001-40010 (10 registers)

Address Conversion:

```
Public Address:   40001
Point Number:     1
Protocol Address: 0 (0x0000)
```

Modbus TCP Request:

```
[00 02][00 00][00 06][01] [03][00 00][00 0A]
 Trans  Proto  Len   Unit  FC  Addr   Qty=10
```

Modbus TCP Response (assuming all registers contain increasing values):

```
[00 02][00 00][00 17][01] [03][14]
  [00 01][00 02][00 03][00 04][00 05]
  [00 06][00 07][00 08][00 09][00 0A]
 Trans  Proto  Len   Unit  FC  ByteCt=20
  Reg0   Reg1   Reg2   Reg3   Reg4
  Reg5   Reg6   Reg7   Reg8   Reg9
```

```
Results:
  40001 = 1
  40002 = 2
  40003 = 3
  ...
  40010 = 10
```

Example 3: Maximum Quantity (125 Registers)

Modbus TCP Request:

```
[00 03][00 00][00 06][01] [03][00 00][00 7D]
 Trans  Proto  Len   Unit  FC  Addr   Qty=125
```

Modbus TCP Response:

```
[00 03][00 00][00 FB][01] [03][FA][...250 bytes of data...]
 Trans  Proto  Len   Unit  FC  ByteCt=250

Length: 0x00FB = 251 bytes (1 Unit ID + 1 FC + 1 ByteCt + 250 data)
ByteCt: 0xFA = 250 bytes (125 registers × 2 bytes/register)
```

Data Interpretation

16-bit Unsigned Integer (UINT16)

Default interpretation - treat each register as unsigned 0-65535:

```
Register value: 0x022B
Decimal: (0x02 × 256) + 0x2B = 512 + 43 = 555
```

16-bit Signed Integer (INT16)

For negative values, interpret as two's complement:

```
Register value: 0xFFFE
Binary: 1111 1111 1111 1110
Signed: -2 (two's complement)

Calculation:
  Bit 15 = 1 → negative number
  Invert bits: 0000 0000 0000 0001
  Add 1:       0000 0000 0000 0010 = 2
  Result: -2
```

32-bit Values (Two Registers)

Combining two consecutive registers:

```
Register 40108: 0x0001
Register 40109: 0xE240

Combined (big-endian ABCD):
  0x0001E240 = 123,456 decimal
```

Warning: Byte order varies by device. See: Byte Ordering

Floating Point (FLOAT32)

IEEE 754 representation using two registers:

```
Register 40108: 0x42F6
Register 40109: 0xE666

Combined: 0x42F6E666
IEEE 754 decode: 123.45 (approximately)
```

See: Data Types - Floating Point

Scaled Values

Many devices use scaling for values with decimal points:

```
Register value: 0x0249 (585 decimal)
Device scaling: ÷10
Actual value: 58.5°C

Always check device documentation for scaling factors!
```

See: Data Types - Scaling

Using in Chipkin Modbus Explorer

Step-by-Step:

1. **Select Function Code**
    - Choose "03 - Read Holding Registers" from dropdown
2. **Enter Starting Address**
    - Use protocol address (0-based)
    - Example: To read 40108, enter **107**

- Or use address conversion calculator if available

3. **Enter Quantity**
   - Number of consecutive registers (1-125)
   - Example: 10 registers

4. **Select Data Type** (for interpretation)
   - UINT16: Unsigned integer (default)
   - INT16: Signed integer
   - UINT32: 32-bit unsigned (uses 2 registers)
   - FLOAT32: IEEE 754 float (uses 2 registers)
   - Other types available

5. **Choose Byte Order** (for multi-register types)
   - Big-Endian (ABCD) - Try this first (standard)
   - Little-Endian (DCBA)
   - Big-Endian Byte Swap (BADC)
   - Little-Endian Byte Swap (CDAB)

6. **Click Send Request**

7. **View Response**
   - Response automatically decoded
   - Values displayed in selected data type
   - Raw hex also shown

Exception Responses

FC03 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC03 or not available in current state

Response:

```
[83][01]
 FC  ExCode
0x83 = 0x03 + 0x80
```

See: Exception 01 - Illegal Function

Exception 02 - Illegal Data Address

Cause:

- Starting address doesn't exist
- Starting address + quantity exceeds available registers
- Address outside device's implemented range

Example: Device only has registers 0-99, but request asks for address 100

Response:

```
[83][02]
 FC  ExCode
```

See: Exception 02 - Illegal Data Address

Exception 03 - Illegal Data Value

Cause:

- Quantity is 0
- Quantity > 125
- Data field values invalid

Example: Quantity = 0x0000 or Quantity = 0x0100 (256)

Response:

```
[83][03]
 FC  ExCode
```

See: Exception 03 - Illegal Data Value

Exception 04 - Server Device Failure

Cause: Internal device error while processing request

Response:

```
[83][04]
 FC  ExCode
```

See: Exception 04 - Server Device Failure

Best Practices

Optimize Read Requests

Good - Single Request:

```
Read registers 40001-40010 (10 registers)
Request: [03][00 00][00 0A]
```

```
Transactions: 1
```

Bad - Multiple Requests:

```
Read register 40001 → Request 1
Read register 40002 → Request 2
...
Read register 40010 → Request 10
Transactions: 10 (slow!)
```

Benefit: Batching reduces network overhead and improves performance

Stay Within Limits

Maximum safe quantity: 125 registers

Don't try to read more:

```
Bad: Quantity = 200 → Exception 03 (Illegal Data Value)
```

Handle Sparse Address Maps

Many devices don't implement consecutive addresses:

```
Device has:
  40001-40050: Process values
  40201-40250: Configuration

Don't read 40001-40250 (wastes bandwidth)


Instead:
  Request 1: Read 40001-40050
  Request 2: Read 40201-40250
```

Verify Range Before Reading

```
FUNCTION readHoldingRegisters(startAddress, quantity)
  // Validate the number of registers requested
  IF quantity < 1 OR quantity > 125 THEN
    THROW ERROR "Invalid quantity: must be between 1 and 125."
  END IF

  // Validate the starting address
  IF startAddress < 0 OR startAddress > 65535 THEN
    THROW ERROR "Invalid start address: must be between 0 and 65535."
  END IF
```

```
  // Ensure the read operation does not exceed the total address space
  IF (startAddress + quantity) > 65536 THEN
    THROW ERROR "The requested range (start address + quantity) exceeds the maximum address of 65535."
  END IF


  // If all checks pass, proceed with sending the Modbus request
  // ...
END FUNCTION
```

Read-Back After Write

For critical values, confirm writes:

```
1. Write value using FC06 or FC16
2. Wait briefly (10-50ms)
3. Read back using FC03
4. Verify value matches what was written
```

See: FC06 - Write Single Register

Common Use Cases

Process Monitoring

Reading sensor values:

```
Temperature:   Register 40001 (UINT16, scale ÷10)
Pressure:      Register 40002 (UINT16, scale ÷100)
Flow Rate:     Register 40003-40004 (FLOAT32)
Status Word:   Register 40010 (UINT16, bit flags)
```

Single request reads all:

```
FC03, Address 0, Quantity 11 → Reads 40001-40011
```

Configuration Reading

Reading device settings:

```
Setpoint:        Register 40101
High Alarm:      Register 40102
Low Alarm:       Register 40103
Operating Mode:  Register 40104
```

Status and Diagnostics

Reading device state:

```
Running Hours:    Registers 40201-40202 (UINT32)
Error Count:      Register 40203
Last Error Code:  Register 40204
Firmware Version: Register 40210 (BCD encoded)
```

Related Topics

Data Organization:

- Data Model - Holding Registers
- Addressing - Converting public to protocol addresses

Similar Functions:

- FC04 - Read Input Registers - Read-only registers
- FC23 - Read/Write Multiple Registers - Combined read and write

Writing Data:

- FC06 - Write Single Register - Write one register
- FC16 - Write Multiple Registers - Write many registers

Message Format:

- Message Structure - PDU and ADU details

Error Handling:

- Exceptions Overview
- Exception 02 - Illegal Data Address
- Exception 03 - Illegal Data Value

FC04 - Read Input Registers

Function Code: 0x04 (4 decimal)

Purpose: Reads the contents of 1 to 125 contiguous input registers from a server device.

Use Case: Used to read read-only data from a device, such as sensor readings, measurements, or status information.

Overview

FC04 reads 16-bit register values from the input register table. This function is nearly identical to FC03 - Read Holding Registers, but it reads from a different, read-only data table.

See: Data Model - Input Registers

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Starting Address │ Quantity of Regs │
│ 0x04         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x04	Fixed value
Starting Address	2 bytes	0x0000 - 0xFFFF	First register to read (0-based)
Quantity of Registers	2 bytes	1 - 125	Number of consecutive registers

Valid Quantity Range: 1 to 125 registers

Response Format

PDU Structure:

```
┌──────────────┬────────────┬──────────────────────────┐
│ Function Code│ Byte Count │ Register Values          │
│ 0x04         │ (1 byte)   │ (N × 2 bytes)            │
│ (1 byte)     │            │                          │
└──────────────┴────────────┴──────────────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x04 (echoed from request)
Byte Count	1 byte	Number of data bytes to follow (Quantity × 2)
Register Values	N bytes	Register data (2 bytes per register, big-endian)

Complete Example

Example: Read One Input Register

Scenario: Read 1 input register at public address 30001 (protocol address 0).

Address Conversion:

```
Public Address:   30001
Point Number:     1
Protocol Address: 0 (0x0000)
```

See: Addressing

Modbus TCP Request:

```
┌─── MBAP Header ──┬─── PDU ──┐
[00 03][00 00][00 06][01] [04][00 00][00 01]
 Trans  Proto  Len   Unit  FC  Addr   Qty=1

Breakdown:
  Function:       0x04
  Start Address:  0x0000 (Input Register 30001)
  Quantity:       0x0001 (1 register)
```

Modbus TCP Response:

Assume the register contains the value 1234 (0x04D2).

```
┌─── MBAP Header ──┬─── PDU ──┐
[00 03][00 00][00 05][01] [04][02][04 D2]
 Trans  Proto  Len   Unit  FC  ByteCt Data

Breakdown:
  Function:       0x04
  Byte Count:     0x02 (1 register × 2 bytes)
  Register Value: 0x04D2 (1234 decimal)
```

Data Interpretation

The data in input registers is interpreted in the same way as holding registers (FC03 - Read Holding Registers). Common formats include:

- 16-bit Unsigned/Signed Integers
- 32-bit Unsigned/Signed Integers (using two registers)
- 32-bit Floating Point (using two registers)
- Scaled values

Refer to the FC03 Data Interpretation section for detailed examples.

Exception Responses

FC04 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC04. **Response:** `[84][01]`

Exception 02 - Illegal Data Address

Cause:

- Starting address doesn't exist.
- Starting address + quantity exceeds available registers. **Response:** `[84][02]`

Exception 03 - Illegal Data Value

Cause:

- Quantity is 0 or > 125. **Response:** `[84][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[84][04]`

Best Practices

- **Use for Read-Only Registers:** This function is specifically for reading registers that are not writable by the Modbus master.
- **Batch Reads:** Combine multiple reads into a single request to minimize network overhead.
- **Check Device Documentation:** Always verify which data tables the device uses. Some devices place all data, including read-only values, in holding registers (readable with FC03 - Read Holding Registers).

Related Topics

- **Data Model:** Input Registers
- **Read/Write Registers:** FC03 - Read Holding Registers
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC05 - Write Single Coil

Function Code: 0x05 (5 decimal)

Purpose: Writes a single discrete output (coil) to either ON or OFF in a server device.

Use Case: Used to control a single digital output, such as turning a relay on or off.

Overview

FC05 is used to set the state of a single coil. The value to be written is specified directly in the request.

See: Data Model - Coils

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Output Address   │ Output Value     │
│ 0x05         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x05	Fixed value
Output Address	2 bytes	0x0000 - 0xFFFF	Address of the coil to write (0-based)
Output Value	2 bytes	0xFF00 or 0x0000	Value to write: ON or OFF

Valid Output Values:

- **ON:** 0xFF00
- **OFF:** 0x0000
- Any non-zero value in the high byte is treated as ON. The low byte is ignored.

Response Format

The normal response is an echo of the request, confirming that the coil was written.

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Output Address   │ Output Value     │
│ 0x05         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x05 (echoed from request)
Output Address	2 bytes	Echoed from request

Output Value	2 bytes	Echoed from request

Complete Example

Example: Turn a Coil ON

Scenario: Turn ON the coil at public address 173 (protocol address 172).

Address Conversion:

```
Public Address:   173
Point Number:     173
Protocol Address: 172 (0x00AC)
```

See: Addressing

Modbus TCP Request:

```
┌─── MBAP Header ──────┬──── PDU ──────┐
[00 04][00 00][00 06][01] [05][00 AC][FF 00]
 Trans  Proto  Len   Unit  FC  Addr   Value=ON

Breakdown:
  Function:       0x05
  Output Address: 0x00AC (Coil 173)
  Output Value:   0xFF00 (ON)
```

Modbus TCP Response:

The response is an exact copy of the request PDU, confirming the write.

```
┌─── MBAP Header ──────┬──── PDU ──────┐
[00 04][00 00][00 06][01] [05][00 AC][FF 00]
 Trans  Proto  Len   Unit  FC  Addr   Value=ON
```

Exception Responses

FC05 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC05. **Response:** `[85][01]`

Exception 02 - Illegal Data Address

Cause: The specified coil address does not exist. **Response:** `[85][02]`

Exception 03 - Illegal Data Value

Cause: The `Output Value` is not 0xFF00 or 0x0000. (Note: some devices may ignore this and treat any non-zero value as ON).
Response: `[85][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[85][04]`

Best Practices

- **Read-Back Verification:** For critical operations, it's good practice to follow a write (FC05) with a read (FC01 - Read Coils) to confirm the state of the coil has changed as expected.
- **Use for Single Points:** This function is efficient for changing a single coil. For multiple coils, use FC15 - Write Multiple Coils to reduce transactions.

Related Topics

- **Data Model:** Coils
- **Reading Coils:** FC01 - Read Coils
- **Writing Multiple Coils:** FC15 - Write Multiple Coils
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC06 - Write Single Register

Function Code: 0x06 (6 decimal)

Purpose: Writes a single 16-bit value to a holding register in a server device.

Use Case: Used to change a single configuration parameter, setpoint, or any other writable 16-bit data point.

Overview

FC06 is used to set the value of a single holding register. The value to be written is specified directly in the request. This function is analogous to FC05 - Write Single Coil but operates on 16-bit registers instead of single-bit coils.

See: Data Model - Holding Registers

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Register Address │ Register Value   │
│ 0x06         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x06	Fixed value
Register Address	2 bytes	0x0000 - 0xFFFF	Address of the register to write (0-based)
Register Value	2 bytes	0x0000 - 0xFFFF	Value to write (0-65535)

Response Format

The normal response is an echo of the request, confirming that the register was written.

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Register Address │ Register Value   │
│ 0x06         │ (2 bytes)        │ (2 bytes)        │
│ (1 byte)     │                  │                  │
└──────────────┴──────────────────┴──────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x06 (echoed from request)
Register Address	2 bytes	Echoed from request
Register Value	2 bytes	Echoed from request

Complete Example

Example: Write to a Holding Register

Scenario: Write the value 100 to the holding register at public address 40001 (protocol address 0).

Address Conversion:

```
Public Address:   40001
Point Number:     1
Protocol Address: 0 (0x0000)
```

See: Addressing

Modbus TCP Request:

```
┌─── MBAP Header ──────┬──── PDU ─────┐
[00 05][00 00][00 06][01] [06][00 00][00 64]
 Trans  Proto  Len   Unit  FC  Addr   Value=100

Breakdown:
  Function:         0x06
  Register Address: 0x0000 (Register 40001)
  Register Value:   0x0064 (100 decimal)
```

Modbus TCP Response:

The response is an exact copy of the request PDU.

```
┌─── MBAP Header ──────┬──── PDU ─────┐
[00 05][00 00][00 06][01] [06][00 00][00 64]
 Trans  Proto  Len   Unit  FC  Addr   Value=100
```

Exception Responses

FC06 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC06. **Response:** `[86][01]`

Exception 02 - Illegal Data Address

Cause: The specified register address does not exist or is read-only. **Response:** `[86][02]`

Exception 03 - Illegal Data Value

Cause: The value to be written is invalid for the specified register (e.g., out of range for the device's application logic). Note that this is application-level validation; the Modbus protocol itself does not validate the data value beyond its 16-bit range. **Response:** `[86][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[86][04]`

Best Practices

- **Read-Back Verification:** For critical setpoints or configuration, follow a write (FC06) with a read (FC03 - Read Holding Registers) to ensure the value was written correctly.
- **Use for Single Registers:** This function is ideal for changing one register at a time. For multiple registers, use FC16 - Write Multiple Registers to improve efficiency.

- **Data Type Awareness:** Remember that you are writing a 16-bit integer. If this register is part of a larger data type (like a 32-bit float), you must perform two separate FC06 writes or a single FC16 - Write Multiple Registers write to update the entire value correctly.

Related Topics

- **Data Model:** Holding Registers
- **Reading Registers:** FC03 - Read Holding Registers
- **Writing Multiple Registers:** FC16 - Write Multiple Registers
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC08 - Diagnostics

Function Code: 0x08 (8 decimal)

Purpose: Provides a series of diagnostic tests and queries for checking the communication status between a client and a server.

Use Case: Used for troubleshooting network issues, monitoring device health, and gathering communication statistics.

Overview

FC08 is a multi-purpose function that operates using a two-byte **sub-function code** specified in the request. The server's response depends on the sub-function requested.

Request/Response Format

The PDU for FC08 is generic, but the `Data` field's content and meaning change based on the sub-function.

General Request PDU:

```
┌──────────────┬──────────────────┬──────────────────┐
| Function Code| Sub-function     | Data             |
| 0x08         | (2 bytes)        | (N bytes)        |
| (1 byte)     |                  |                  |
└──────────────┴──────────────────┴──────────────────┘
```

General Response PDU:

The response is an echo of the request's function and sub-function, followed by data specific to the sub-function.

```
┌──────────────┬──────────────────┬──────────────────┐
| Function Code| Sub-function     | Data             |
| 0x08         | (2 bytes)        | (N bytes)        |
```

```
| (1 byte)     |                  |                |
└──────────────┴──────────────────┴────────────────┘
```

Standard Sub-functions

Here are some of the most common standard sub-function codes.

Sub-function 0000: Return Query Data

This is a simple loopback test. The server echoes back the exact data it received in the request.

- **Purpose:** To verify that the communication path is working and that the server is processing requests.
- **Request Data:** Any data, up to 250 bytes.
- **Response Data:** An exact copy of the request data.

Example:

Request:

```
[08][00 00][DE AD]  // Sub-function 0000, Data: 0xDEAD
```

Response:

```
[08][00 00][DE AD]  // Echoes the request
```

Sub-function 0001: Restart Communications Option

This sub-function commands a server to restart its communication port.

- **Purpose:** To reset a device's communication stack remotely.
- **Request Data:** Optional data. If the data is `0x0000`, the server performs a cold restart, clearing its communication event log. If `0xFF00`, it performs a warm restart, leaving the log intact.
- **Response:** Echoes the request.

Example (Cold Restart):

Request:

```
[08][00 01][00 00]
```

Response:

```
[08][00 01][00 00]
```

Sub-function 0002: Return Diagnostic Register

Retrieves the contents of the server's 16-bit diagnostic register.

- **Purpose:** To get a quick status overview. The bits of this register are device-specific.
- **Request Data:** None (2 bytes of 0x0000).
- **Response Data:** The 16-bit value of the diagnostic register.

Example:

Request:

```
[08][00 02][00 00]
```

Response:

```
[08][00 02][01 05] // Diagnostic register contains 0x0105
```

Sub-function 0011 (0x0B): Return Bus Message Count

Retrieves a 16-bit counter of the total number of messages the server has detected on the bus since its last restart or counter reset.

- **Purpose:** To monitor network traffic and health.
- **Request Data:** None (2 bytes of 0x0000).
- **Response Data:** A 16-bit count of messages.

Example:

Request:

```
[08][00 0B][00 00]
```

Response:

```
[08][00 0B][04 D2] // 1234 messages detected
```

Sub-function 0012 (0x0C): Return Bus Communication Error Count

Retrieves a 16-bit counter of communication errors (e.g., parity, framing, CRC errors) detected by the server.

- **Purpose:** To diagnose physical layer or noise issues on the network.
- **Request Data:** None (2 bytes of 0x0000).

- **Response Data:** A 16-bit count of communication errors.

Example:

Request:

```
[08][00 0C][00 00]
```

Response:

```
[08][00 0C][00 05] // 5 communication errors detected
```

Sub-function 0013 (0x0D): Return Bus Exception Error Count

Retrieves a 16-bit counter of the number of exception responses the server has sent.

- **Purpose:** To identify configuration or application-level errors.
- **Request Data:** None (2 bytes of 0x0000).
- **Response Data:** A 16-bit count of exceptions sent.

Example:

Request:

```
[08][00 0D][00 00]
```

Response:

```
[08][00 0D][00 1A] // 26 exception responses sent
```

Sub-function 0010 (0x0A): Clear Counters and Diagnostic Register

Resets all diagnostic counters and the diagnostic register to zero.

- **Purpose:** To establish a new baseline for monitoring.
- **Request Data:** None (2 bytes of 0x0000).
- **Response:** Echoes the request.

Example:

Request:

```
[08][00 0A][00 00]
```

Response:

```
[08][00 0A][00 00]
```

Exception Responses

FC08 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC08. **Response:** `[88][01]`

Exception 03 - Illegal Data Value

Cause: The requested sub-function code is not supported by the device. **Response:** `[88][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[88][04]`

Best Practices

- **Use for Troubleshooting:** FC08 is an invaluable tool for diagnosing communication problems without needing physical access to the device.
- **Check for Support:** Not all devices implement all sub-functions. Always consult the device's documentation to see which diagnostics are available.
- **Establish Baselines:** When monitoring a network, use sub-function 0x0A to clear counters, then poll the other counter sub-functions periodically to measure error rates over time.

Related Topics

- **Error Handling:** Exceptions Overview
- **Other Diagnostic Functions:** FC11 - Get Comm Event Counter
- **Troubleshooting:** Troubleshooting Guide

FC11 - Get Comm Event Counter

Function Code: 0x0B (11 decimal)

Purpose: Fetches a status word and a count of communication events from a server device.

Use Case: To monitor the health of a device's communication port and track the number of successful messages processed.

Overview

FC11 provides a way to get two key pieces of information:

1. **Status Word:** A 16-bit word indicating the current state of the communication port (e.g., busy, ready).
2. **Event Counter:** A 16-bit counter of the number of successful communication events.

This function is useful for clients that need to coordinate with a server that may be busy with long-running tasks.

Request Format

The request PDU for FC11 is empty. It only contains the function code.

PDU Structure:

```
┌──────────────┐
│ Function Code│
│ 0x0B         │
│ (1 byte)     │
└──────────────┘
```

Response Format

PDU Structure:

```
┌──────────────┬──────────────┬──────────────┐
│ Function Code│ Status       │ Event Count  │
│ 0x0B         │ (2 bytes)    │ (2 bytes)    │
│ (1 byte)     │              │              │
└──────────────┴──────────────┴──────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x0B (echoed from request)
Status	2 bytes	`0x0000` if ready, `0xFFFF` if busy
Event Count	2 bytes	Number of successful communication events

Status Word Values:

- `0x0000` **:** The server is ready to accept a new request.
- `0xFFFF` **:** The server is busy processing a long-duration command. The client should not send a new request until the server is ready.

Complete Example

Scenario: A client polls a server to check its status.

Modbus TCP Request:

```
┌─── MBAP Header ──────┬── PDU ──┐
[00 08][00 00][00 02][01] [0B]
 Trans  Proto  Len   Unit  FC

Breakdown:
  Function: 0x0B
```

Modbus TCP Response (Server is Ready):

```
┌─── MBAP Header ──────┬──────── PDU ────────┐
[00 08][00 00][00 05][01] [0B][00 00][01 F4]
 Trans  Proto  Len   Unit  FC  Status EventCt

Breakdown:
  Function:    0x0B
  Status:      0x0000 (Ready)
  Event Count: 0x01F4 (500 events)
```

Modbus TCP Response (Server is Busy):

```
┌─── MBAP Header ──────┬──────── PDU ────────┐
[00 08][00 00][00 05][01] [0B][FF FF][01 F4]
 Trans  Proto  Len   Unit  FC  Status EventCt

Breakdown:
  Function:    0x0B
  Status:      0xFFFF (Busy)
  Event Count: 0x01F4 (500 events)
```

Exception Responses

FC11 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC11. **Response:** `[8B][01]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[8B][04]`

Best Practices

- **Polling for Status:** This function is designed to be polled. A client can use it to wait for a server to become available after sending a long-running command (e.g., a programming command acknowledged with Exception 05).
- **Event Counter Monitoring:** The event counter can be used to detect if a device has restarted. If the counter value is less than the previously recorded value, the device has likely reset.
- **Alternative to FC08:** This function provides a subset of the information available through FC08 - Diagnostics but in a more direct and compact format.

Related Topics

- **Error Handling:** Exceptions Overview
- **Other Diagnostic Functions:** FC08 - Diagnostics
- **Long-Running Commands:** Exception 05 - Acknowledge
- **Troubleshooting:** Troubleshooting Guide

FC15 - Write Multiple Coils

Function Code: 0x0F (15 decimal)

Purpose: Writes a sequence of coils to either ON or OFF in a server device.

Use Case: Used to control multiple digital outputs at once, such as setting a pattern on a bank of lights or relays.

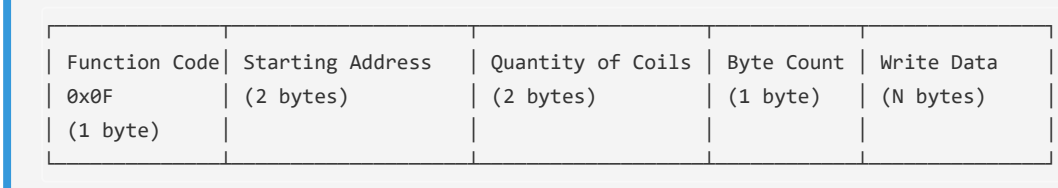
Overview

FC15 allows a client to set the state of multiple contiguous coils in a single transaction. The coil states are packed into bytes in the request message.

See: Data Model - Coils

Request Format

PDU Structure:



Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x0F	Fixed value

Starting Address	2 bytes	0x0000 - 0xFFFF	First coil to write (0-based)
Quantity of Coils	2 bytes	1 - 1968	Number of consecutive coils to write
Byte Count	1 byte	1 - 246	Number of data bytes to follow (ceil(Quantity / 8))
Write Data	N bytes	Coil states packed into bytes	

Valid Quantity Range: 1 to 1968 coils (0x07B0)

Bit Packing:

- The coil states are packed into bytes, with the first coil corresponding to the LSB of the first data byte. This is the same packing format used in the response of FC01 - Read Coils.

Response Format

The normal response confirms the starting address and the quantity of coils written.

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
│ Function Code│ Starting Address │ Quantity of Coils │
│ 0x0F         │ (2 bytes)        │ (2 bytes)         │
│ (1 byte)     │                  │                   │
└──────────────┴──────────────────┴──────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x0F (echoed from request)
Starting Address	2 bytes	Echoed from request
Quantity of Coils	2 bytes	Echoed from request

Complete Example

Example: Write 10 Coils

Scenario: Write to 10 coils starting at public address 20 (protocol address 19).

- Coils 20, 22, 24 ON
- All others OFF

Address Conversion:

```
Public Address:   20
Point Number:     20
```

```
Protocol Address: 19 (0x0013)
```

See: Addressing

Data to Write:

- Coils 20-27: `00010101` binary = `0x15`
- Coils 28-29: `00000000` binary = `0x00`

Modbus TCP Request:

```
┌─── MBAP Header ──┬──────────── PDU ──────────┐
[00 06][00 00][00 08][01] [0F][00 13][00 0A][02][15 00]
 Trans  Proto  Len   Unit  FC  Addr   Qty    ByteCt Data
```

```
Breakdown:
  Function:         0x0F
  Start Address:    0x0013 (Coil 20)
  Quantity of Coils:0x000A (10 coils)
  Byte Count:       0x02 (ceil(10 / 8) = 2 bytes)
  Write Data:       0x1500
```

Modbus TCP Response:

```
┌─── MBAP Header ──┬─────── PDU ──────┐
[00 06][00 00][00 06][01] [0F][00 13][00 0A]
 Trans  Proto  Len   Unit  FC  Addr   Qty
```

Exception Responses

FC15 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC15. **Response:** `[8F][01]`

Exception 02 - Illegal Data Address

Cause:

- The starting address does not exist.
- The range (start address + quantity) exceeds the available coils. **Response:** `[8F][02]`

Exception 03 - Illegal Data Value

Cause:

- The quantity is 0 or > 1968.
- The byte count does not match the quantity (e.g., `Byte Count != ceil(Quantity / 8)`). **Response:** `[8F][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[8F][04]`

Best Practices

- **Efficiency:** Use FC15 instead of multiple FC05 - Write Single Coil requests to write to several coils at once. This significantly reduces network overhead.
- **Read-Back Verification:** For critical control sequences, follow an FC15 write with an FC01 - Read Coils read to verify that all coils have been set to their intended states.
- **Data Packing:** Pay close attention to the bit packing order (LSB first) when constructing the `Write Data` field.

Related Topics

- **Data Model:** Coils
- **Reading Coils:** FC01 - Read Coils
- **Writing a Single Coil:** FC05 - Write Single Coil
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC16 - Write Multiple Registers

Function Code: 0x10 (16 decimal)

Purpose: Writes a block of contiguous holding registers (1 to 123 registers) in a server device.

Use Case: Used to change multiple configuration parameters, download a recipe, or write any block of 16-bit data in a single transaction.

Overview

FC16 allows a client to set the values of multiple contiguous holding registers. This is one of the most powerful and commonly used functions for configuring devices.

See: Data Model - Holding Registers

Request Format

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┬──────────────┬──────────────────┐
| Function Code| Starting Address  | Quantity of Regs  | Byte Count | Register Values|
| 0x10         | (2 bytes)         | (2 bytes)         | (1 byte)   | (N × 2 bytes)  |
```

```
| (1 byte)     |                  |                  |           |               |
└──────────────┴──────────────────┴──────────────────┴───────────┴───────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x10	Fixed value
Starting Address	2 bytes	0x0000 - 0xFFFF	First register to write (0-based)
Quantity of Registers	2 bytes	1 - 123	Number of consecutive registers to write
Byte Count	1 byte	2 - 246	Number of data bytes to follow (Quantity × 2)
Register Values	N bytes	Register data (2 bytes per register, big-endian)	

Valid Quantity Range: 1 to 123 registers (0x007B)

Response Format

The normal response confirms the starting address and the quantity of registers written.

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┐
| Function Code| Starting Address  | Quantity of Regs  |
| 0x10         | (2 bytes)         | (2 bytes)         |
| (1 byte)     |                   |                   |
└──────────────┴──────────────────┴──────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x10 (echoed from request)
Starting Address	2 bytes	Echoed from request
Quantity of Registers	2 bytes	Echoed from request

Complete Example

Example: Write Two Registers

Scenario: Write two holding registers starting at public address 40001 (protocol address 0).

- Register 40001 = 10
- Register 40002 = 20

Address Conversion:

```
Public Address:   40001
Point Number:     1
Protocol Address: 0 (0x0000)
```

See: Addressing

Modbus TCP Request:

```
┌─── MBAP Header ──┬──────────── PDU ──────────┐
[00 07][00 00][00 09][01] [10][00 00][00 02][04][00 0A 00 14]
 Trans  Proto  Len   Unit  FC  Addr   Qty    ByteCt Data

Breakdown:
  Function:         0x10
  Start Address:    0x0000 (Register 40001)
  Quantity of Regs: 0x0002 (2 registers)
  Byte Count:       0x04 (2 registers × 2 bytes)
  Register Values:  0x000A (10), 0x0014 (20)
```

Modbus TCP Response:

```
┌─── MBAP Header ──┬─────── PDU ──────┐
[00 07][00 00][00 06][01] [10][00 00][00 02]
 Trans  Proto  Len   Unit  FC  Addr   Qty
```

Exception Responses

FC16 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC16. **Response:** `[90][01]`

Exception 02 - Illegal Data Address

Cause:

- The starting address does not exist.
- The range (start address + quantity) exceeds the available registers.
- Any register in the range is read-only. **Response:** `[90][02]`

Exception 03 - Illegal Data Value

Cause:

- The quantity is 0 or > 123.
- The byte count is incorrect (`Byte Count != Quantity * 2`).

- Any data value in the `Register Values` field is invalid for the corresponding register (application-level validation). **Response:** `[90][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[90][04]`

Best Practices

- **Efficiency:** This is the preferred function for writing multiple registers. Use it instead of multiple FC06 - Write Single Register requests to reduce network traffic and latency.
- **Data Integrity:** When writing multi-register values (e.g., 32-bit floats), use a single FC16 request to ensure the entire value is updated atomically. Using multiple FC06 - Write Single Register requests could lead to a temporary inconsistent state if the connection is interrupted between writes.
- **Read-Back Verification:** For critical data, follow an FC16 write with an FC03 - Read Holding Registers read to confirm that all registers have been updated correctly.

Related Topics

- **Data Model:** Holding Registers
- **Reading Registers:** FC03 - Read Holding Registers
- **Writing a Single Register:** FC06 - Write Single Register
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC23 - Read/Write Multiple Registers

Function Code: 0x17 (23 decimal)

Purpose: Performs a combined read and write operation in a single Modbus transaction, which avoids the possibility of a race condition.

Use Case: Ideal for applications where a client needs to write new values to a set of registers and simultaneously read back a different set of registers, ensuring the write operation completes before the read begins.

Overview

FC23 combines the functionality of a read (like FC03 - Read Holding Registers) and a write (like FC16 - Write Multiple Registers) into a single, atomic operation on the server side. The server is guaranteed to execute the write first, and then the read.

This is particularly useful in control loops where, for example, a new setpoint is written and the resulting process variable is read back immediately.

Request Format

The request PDU specifies both the read and write parameters.

PDU Structure:

```
┌──────────────┬──────────────────┬──────────────────┬──────────────────┬──────────────────┬──────────────┐
| Function Code| Read Start Addr  | Read Quantity    | Write Start Addr | Write Quantity   | Write
Byte | Write Data     |
| 0x17         | (2 bytes)        | (2 bytes)        | (2 bytes)        | (2 bytes)        | Count
(1B) | (N × 2 bytes)  |
| (1 byte)     |                  |                  |                  |                  |
|              |
└──────────────┴──────────────────┴──────────────────┴──────────────────┴──────────────────┴──────────────┘
```

Request Fields:

Field	Size	Range	Description
Function Code	1 byte	0x17	Fixed value
Read Start Addr	2 bytes	0x0000 - 0xFFFF	First holding register to read
Read Quantity	2 bytes	1 - 125	Number of registers to read
Write Start Addr	2 bytes	0x0000 - 0xFFFF	First holding register to write
Write Quantity	2 bytes	1 - 121	Number of registers to write
Write Byte Count	1 byte	2 - 242	`Write Quantity` × 2
Write Data	N bytes	Values to write (2 bytes per register)	

Response Format

The response contains the data from the read portion of the operation.

PDU Structure:

```
┌──────────────┬──────────────┬──────────────────┐
| Function Code| Read Byte    | Read Data        |
| 0x17         | Count (1B)   | (N × 2 bytes)    |
| (1 byte)     |              |                  |
└──────────────┴──────────────┴──────────────────┘
```

Response Fields:

Field	Size	Description
Function Code	1 byte	0x17 (echoed from request)
Read Byte Count	1 byte	`Read Quantity` × 2
Read Data	N bytes	Data from the read registers

Complete Example

Scenario:

- **Write:** Write the value `3` to register `40001`.
- **Read:** Read the value of register `40011`.

Modbus TCP Request:

```
[10][00 00][00 0B][01] [17][00 0A][00 01][00 00][00 01][02][00 03]
 Trans Proto  Len   Unit  FC ReadAddr ReadQty WriteAddr WriteQty ByteCt WriteData

Breakdown:
  Function:         0x17
  Read Start Addr:  0x000A (Register 40011)
  Read Quantity:    0x0001
  Write Start Addr: 0x0000 (Register 40001)
  Write Quantity:   0x0001
  Write Byte Count: 0x02
  Write Data:       0x0003
```

Modbus TCP Response:

Assume register 40011 contains the value `123` (0x007B).

```
[10][00 00][00 05][01] [17][02][00 7B]
 Trans Proto  Len   Unit  FC ByteCt ReadData

Breakdown:
  Function:       0x17
  Read Byte Count:0x02
  Read Data:      0x007B (Value of register 40011)
```

Exception Responses

FC23 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC23. **Response:** `[97][01]`

Exception 02 - Illegal Data Address

Cause:

- Any address in the read or write range is invalid or does not exist.
- Any register in the write range is read-only. **Response:** `[97][02]`

Exception 03 - Illegal Data Value

Cause:

- Invalid quantity for read or write.
- Incorrect `Write Byte Count`.
- Invalid data value in the `Write Data` field. **Response:** `[97][03]`

Exception 04 - Server Device Failure

Cause: Internal device error during the read or write operation. **Response:** `[97][04]`

Best Practices

- **Atomic Operations:** Use FC23 to ensure that a write operation is completed before a subsequent read, preventing race conditions where a client might read stale data.
- **Efficiency:** While slightly more complex, FC23 is more efficient than sending separate FC16 - Write Multiple Registers (write) and FC03 - Read Holding Registers (read) requests, as it halves the number of required transactions.
- **Device Support:** This is a less common function. Always verify that the target device supports FC23 before implementing it in an application.

Related Topics

- **Reading Registers:** FC03 - Read Holding Registers
- **Writing Registers:** FC16 - Write Multiple Registers
- **Error Handling:** Exceptions Overview
- **Addressing:** Addressing

FC43 / MEI 14 - Read Device Identification

Function Code: 0x2B (43 decimal) **MEI Type:** 0x0E (14 decimal)

Purpose: To read identifying information from a server, such as vendor name, product code, and version.

Use Case: Essential for auto-discovery, device verification, and asset management. It allows a client to programmatically identify a device without prior knowledge.

Overview

This function provides a structured way to access a device's identity. It operates on a "category" of objects, each with a specific Object ID. The client requests one or more of these objects, and the server returns their values.

The function code is `0x2B`, and it uses a **Modbus Encapsulated Interface (MEI)** type `0x0E`.

Request Format

PDU Structure:

```
┌──────────────┬──────────┬──────────────┬──────────┐
│ Function Code│ MEI Type │ Read Device  │ Object ID │
│ 0x2B         │ 0x0E     │ ID Code      │ (1 byte)  │
│ (1 byte)     │ (1 byte) │ (1 byte)     │           │
└──────────────┴──────────┴──────────────┴──────────┘
```

Request Fields:

Field	Size	Value	Description
Function Code	1 byte	0x2B	Fixed value
MEI Type	1 byte	0x0E	Specifies "Read Device Identification"
Read Device ID Code	1 byte	0x01-0x04	Type of access
Object ID	1 byte	0x00-0xFF	The specific piece of information requested

Read Device ID Codes:

- `0x01` : Request to get basic device identification (stream access).
- `0x02` : Request to get regular device identification (stream access).
- `0x03` : Request to get extended device identification (stream access).
- `0x04` : Request to get one specific identification object (individual access).

Response Format

The response is complex and includes a list of objects, each with an ID, length, and value.

PDU Structure (Simplified):

```
[FC][MEI][ID Code][Conformity][More][NextObjID] [Num Objs] [Obj1 ID][Obj1 Len][Obj1 Val]...
```

Key Response Fields:

- **Conformity Level:** Indicates how well the device conforms to the specification.
- **More Follows:** `0x00` if this is the last response fragment, `0xFF` if more fragments are coming.
- **Next Object ID:** The ID of the next available object, used for streaming.
- **Number of Objects:** How many objects are included in this response.
- **Object List:** A series of `(ID, Length, Value)` triplets.

Standard Object IDs

The specification defines three categories of objects:

Basic Device Identification (ID Code 0x01)

- `0x00` **VendorName:** The manufacturer's name (ASCII string).
- `0x01` **ProductCode:** The device's model number (ASCII string).
- `0x02` **MajorMinorRevision:** The device's firmware version (ASCII string).

Regular Device Identification (ID Code 0x02)

Includes Basic objects, plus:

- `0x03` **VendorUrl:** Manufacturer's website (ASCII string).
- `0x04` **ProductName:** The device's brand name (ASCII string).
- `0x05` **ModelName:** The specific model name (ASCII string).
- `0x06` **UserApplicationName:** User-defined application name (ASCII string).

Extended Device Identification (ID Code 0x03)

- Includes all Basic and Regular objects.
- `0x07-0x7F` : Reserved for public definition.
- `0x80-0xFF` : Vendor-specific objects.

Complete Example

Scenario: Request the `VendorName` (Object ID 0x00) from a device.

Modbus TCP Request (Individual Access):

```
[00 09][00 00][00 04][01] [2B][0E][04][00]
 Trans  Proto  Len   Unit  FC  MEI ID-Code ObjID

Breakdown:
  Function:         0x2B
  MEI Type:         0x0E
  Read Device ID:   0x04 (get one specific object)
  Object ID:        0x00 (VendorName)
```

Modbus TCP Response:

Assume the vendor is "Chipkin".

```
[00 09][00 00][00 0F][01] [2B][0E][04][81][00][00][01] [00][07]["Chipkin"]
 Trans  Proto  Len   Unit  FC  MEI ...               ObjID Len  Value

Breakdown (PDU only):
  [2B 0E 04 81 00 00 01] - Header info
  [00] - Object ID: VendorName
```

```
[07] - Length: 7 bytes
[43 68 69 70 6B 69 6E] - "Chipkin" in ASCII
```

Exception Responses

FC43 can return the following exceptions:

Exception 01 - Illegal Function

Cause: Device doesn't support FC43. **Response:** `[AB][01]`

Exception 02 - Illegal Data Address

Cause: The requested Object ID does not exist. **Response:** `[AB][02]`

Exception 03 - Illegal Data Value

Cause: The `Read Device ID Code` is not supported. **Response:** `[AB][03]`

Exception 04 - Server Device Failure

Cause: Internal device error. **Response:** `[AB][04]`

Best Practices

- **Use for Discovery:** This function is the standard way to identify Modbus devices on a network. A discovery tool can iterate through addresses and send an FC43 request to build a map of the network.
- **Handle Streamed Responses:** When requesting Basic, Regular, or Extended information, be prepared to handle fragmented responses where `More Follows` is `0xFF`.
- **Check Conformity Level:** The conformity level in the response tells you how much you can rely on the device's implementation of this function.

Related Topics

- **Discovery:** Discovery Guide
- **Error Handling:** Exceptions Overview
- **Troubleshooting:** Troubleshooting Guide

Glossary of Modbus Terms

This glossary defines all key Modbus terms as specified in the Modbus Application Protocol Specification and provides cross-references to relevant sections in this manual.

A

ADU (Application Data Unit)

The complete Modbus message frame that includes transport-specific headers plus the PDU. The ADU format differs between transport types:

- **Modbus RTU**: `[Unit ID][PDU][CRC-16]`
- **Modbus TCP**: `[MBAP Header][PDU]`

The ADU represents the full packet transmitted over the physical medium.

See also: PDU, MBAP, CRC

Referenced in:

- Modbus Technical Reference - Modbus TCP
- Modbus Technical Reference - Modbus RTU
- Modbus Protocol - Message Structure

Addressing Model

The Modbus addressing scheme for accessing data elements. Modbus uses two parallel addressing systems:

1. **Protocol Addressing**: 0-based addresses used in actual messages (0-65535)
2. **Public Addressing**: 1-based addresses with data type prefix used in documentation (e.g., 40001-465535 for holding registers)

Important: Protocol address = Public address - prefix - 1

See also: Off-By-One Error, Protocol Addressing, Public Addressing

Referenced in:

- Modbus Technical Reference - Off-By-One Addressing
- Modbus Protocol - Addressing

Asynchronous Serial Transmission

Serial communication method where data is transmitted without a shared clock signal. Each character is framed with start and stop bits. Used in Modbus RTU over RS-232, RS-485, and RS-422.

Common parameters:

- Baud rate: 9600, 19200, 38400, 57600, 115200
- Data bits: 8
- Parity: None, Even, Odd
- Stop bits: 1 or 2

Referenced in:

- Modbus Technical Reference - RS-485

- Modbus Technical Reference - Modbus RTU
- Troubleshooting - Serial Communication

B

Baud Rate

The rate at which symbols (bits) are transmitted over a serial line, measured in bits per second (bps). Higher baud rates allow faster communication but require better quality cabling and termination.

Common Modbus baud rates:

- 9600: Default, most reliable, long distances
- 19200: Good balance
- 38400: Faster, requires quality wiring
- 57600: High speed, shorter distances
- 115200: Maximum speed, excellent installation required

Referenced in:

- Modbus Technical Reference - Baud Rate Selection
- Getting Started - Connection Settings
- Troubleshooting - RS-485

Big Endian

Byte ordering where the most significant byte (MSB) is stored or transmitted first. Modbus specification mandates big endian byte order for addresses and data values.

Example: Value 0x1234 is transmitted as 0x12, 0x34

For 32-bit values, word order also matters - see Byte Ordering.

Referenced in:

- Modbus Technical Reference - Byte Ordering
- Modbus Protocol - Data Encoding

Broadcast

A Modbus message sent to all devices on a network (address 0). Slaves execute the command but do not respond. Used for time synchronization and parameter updates to multiple devices.

Limitations:

- No response expected or given
- Only valid for write operations
- Client has no confirmation of success
- Rarely used in Modbus TCP

Referenced in:

- Modbus Technical Reference - Multi-Drop Networks
- Modbus Protocol - Function Codes

Byte Count

A field in Modbus messages indicating the number of data bytes that follow. Used in responses and multi-byte write requests to specify the payload length.

Example: In Read Holding Registers response, Byte Count = Quantity of Registers × 2

Referenced in:

- Modbus Protocol - Function Codes

Byte Ordering (Endianness)

The sequence in which bytes are arranged in multi-byte data types. Modbus has two aspects of byte ordering:

1. **Byte Order within each register**: Big endian (MSB first) per spec
2. **Word Order across registers**: For 32-bit/64-bit values

Four possible 32-bit combinations:

- **ABCD** (Big/Big): Modbus standard
- **DCBA** (Little/Little): Fully reversed
- **BADC** (Big/Little): Byte swap
- **CDAB** (Little/Big): Word swap

See also: Big Endian, Little Endian

Referenced in:

- Modbus Technical Reference - Byte Ordering
- Modbus Protocol - Data Types

C

Client

The device that initiates Modbus requests (formerly called "master"). The client sends queries to servers and processes their responses. In most systems, there is one client and one or more servers.

Typical clients: SCADA systems, PLCs (when reading from other devices), HMIs

See also: Server, Master

Referenced in:

- Modbus Client - Overview
- Modbus Protocol - Client-Server Model

Coil

A single-bit read/write data element in the Modbus data model. Historically represented discrete outputs in PLCs.

- **Function Code 01** (0x01): Read Coils
- **Function Code 05** (0x05): Write Single Coil
- **Function Code 15** (0x0F): Write Multiple Coils
- **Address Range**: 0-65535 (protocol addressing)
- **Public Address**: 00001-065536 (prefix 0)
- **Values**: 0 (OFF) or 1 (ON), represented as 0x0000 or 0xFF00 for writes

See also: Discrete Input, Holding Register

Referenced in:

- Modbus Protocol - Function Code 01
- Modbus Protocol - Function Code 05
- Modbus Protocol - Function Code 15
- Modbus Client - Reading Coils

CRC (Cyclic Redundancy Check)

A 16-bit error-detection code used in Modbus RTU to verify message integrity. The CRC-16 algorithm uses polynomial 0xA001.

Calculation:

- Computed over: Unit ID + Function Code + Data
- Not included in CRC: The CRC itself
- Byte order: Low byte first, high byte second
- Position: Last 2 bytes of RTU frame

Note: Modbus TCP does not use CRC; TCP's own checksums provide error detection.

See also: Modbus RTU, LRC

Referenced in:

- Modbus Technical Reference - Modbus RTU
- Troubleshooting - CRC Errors

D

Data Model

The Modbus data organization consisting of four primary tables:

Data Type	Access	Size	Public Address Prefix
Coils	Read/Write	1 bit	0xxxx

Discrete Inputs	Read-Only	1 bit	1xxxx
Input Registers	Read-Only	16 bits	3xxxx
Holding Registers	Read/Write	16 bits	4xxxx

Each table can contain up to 65,536 elements (0-65535 in protocol addressing).

Referenced in:

- Modbus Protocol - Data Model
- Modbus Technical Reference - Data Types

Diagnostic Function

Function Code 08 (0x08) provides tests for checking communication systems and internal device status. Serial line devices only.

Sub-functions include:

- 00: Return Query Data (loopback test)
- 01: Restart Communications
- 04: Force Listen Only Mode
- 10: Clear Counters and Diagnostic Register
- 11-18: Return various counters (message count, error count, etc.)

Referenced in:

- Modbus Protocol - Function Code 08

Discrete Input

A single-bit read-only data element in the Modbus data model. Historically represented physical inputs in PLCs.

- **Function Code 02** (0x02): Read Discrete Inputs
- **Address Range**: 0-65535 (protocol addressing)
- **Public Address**: 10001-165536 (prefix 1)
- **Values**: 0 (OFF) or 1 (ON)

See also: Coil, Input Register

Referenced in:

- Modbus Protocol - Function Code 02
- Modbus Client - Reading Discrete Inputs

E

Exception Code

A 1-byte value returned in an exception response indicating the specific error condition. Defined in the Modbus specification.

Common exception codes:

Code	Name	Meaning
01	Illegal Function	Function code not supported
02	Illegal Data Address	Address doesn't exist
03	Illegal Data Value	Value in query data field invalid
04	Server Device Failure	Unrecoverable error occurred
05	Acknowledge	Long operation in progress
06	Server Device Busy	Busy processing command
08	Memory Parity Error	Extended file area parity error
0A	Gateway Path Unavailable	Gateway cannot allocate path
0B	Gateway Target Device Failed to Respond	No response from target

Referenced in:

- Modbus Protocol - Exception Responses
- Troubleshooting - Exception Codes

Exception Response

A Modbus response indicating an error occurred while processing a request. An exception response has:

1. **Function Code**: Original function code + 0x80 (sets MSB to 1)
2. **Exception Code**: 1-byte error code

Example: Request with function code 0x03, error → Response with 0x83 + exception code

See also: Exception Code

Referenced in:

- Modbus Protocol - Exception Responses
- Troubleshooting - Understanding Exceptions

F

FIFO (First-In-First-Out) Queue

A data buffer where the first value written is the first value read. Function Code 24 (0x18) reads the contents of a FIFO queue without clearing it.

- Maximum 31 queued data registers
- Queue count register returned first
- Then queued data registers

- If queue exceeds 31, exception code 03 returned

Referenced in:

- Modbus Protocol - Function Code 24

Function Code

A 1-byte field in the Modbus PDU identifying the requested action. Valid codes are 1-255, with 128-255 reserved for exception responses.

Function code categories:

- **Public Function Codes**: Well-defined, documented, standardized
- **User-Defined**: 65-72 and 100-110
- **Reserved**: Used by some vendors for legacy products

Common function codes:

Code	Name	Operation
01 (0x01)	Read Coils	Read 1-2000 coils
02 (0x02)	Read Discrete Inputs	Read 1-2000 discrete inputs
03 (0x03)	Read Holding Registers	Read 1-125 holding registers
04 (0x04)	Read Input Registers	Read 1-125 input registers
05 (0x05)	Write Single Coil	Write 1 coil
06 (0x06)	Write Single Register	Write 1 register
15 (0x0F)	Write Multiple Coils	Write 1-1968 coils
16 (0x10)	Write Multiple Registers	Write 1-123 registers

Referenced in:

- Modbus Protocol - Function Codes
- Modbus Technical Reference - Modbus Protocol Variants

G

Gateway

A device that translates between different Modbus networks or protocols. Common scenarios:

- **Modbus TCP ↔ Modbus RTU**: Bridge Ethernet to serial
- **Multi-client aggregation**: Multiple TCP clients to single RS-485 master
- **Protocol translation**: BACnet ↔ Modbus, OPC ↔ Modbus

Challenges:

- Data staleness when RTU device offline
- Performance bottlenecks
- Configuration complexity
- Exception codes 0A and 0B indicate gateway issues

See also: Exception Code

Referenced in:

- Modbus Technical Reference - Gateways
- Troubleshooting - Gateway Issues

H

HDLC (High-Level Data Link Control)

A bit-oriented code-transparent synchronous data link layer protocol. Used in Modbus Plus networks.

Referenced in:

- Modbus Technical Reference - Modbus Protocol Variants

Holding Register

A 16-bit read/write data element in the Modbus data model. The most commonly used register type for storing configuration and process values.

- **Function Code 03** (0x03): Read Holding Registers
- **Function Code 06** (0x06): Write Single Register
- **Function Code 16** (0x10): Write Multiple Registers
- **Function Code 22** (0x16): Mask Write Register
- **Function Code 23** (0x17): Read/Write Multiple Registers
- **Address Range**: 0-65535 (protocol addressing)
- **Public Address**: 40001-465536 (prefix 4)
- **Size**: 16 bits (2 bytes)
- **Range**: 0-65535 (unsigned) or -32768 to +32767 (signed)

See also: Input Register, Data Types

Referenced in:

- Modbus Protocol - Function Code 03
- Modbus Protocol - Function Code 06
- Modbus Protocol - Function Code 16
- Modbus Client - Reading/Writing Registers

I

Input Register

A 16-bit read-only data element in the Modbus data model. Typically used for analog inputs and measurements.

- **Function Code 04** (0x04): Read Input Registers
- **Address Range**: 0-65535 (protocol addressing)
- **Public Address**: 30001-365536 (prefix 3)
- **Size**: 16 bits (2 bytes)
- **Access**: Read-only

See also: Holding Register, Discrete Input

Referenced in:

- Modbus Protocol - Function Code 04
- Modbus Client - Reading Input Registers

Inter-Character Timeout

The maximum time allowed between characters within a single Modbus RTU message. If exceeded, the message is considered incomplete and discarded by the slave.

Typical value: 1.5 character times

- At 9600 baud: ~1.5 ms
- At 19200 baud: ~0.75 ms

See also: Inter-Frame Delay

Referenced in:

- Modbus Technical Reference - Message Timing
- Troubleshooting - Timing Issues

Inter-Frame Delay

The minimum silence time required between Modbus RTU messages. Allows slaves to process the previous message before receiving the next.

Minimum value: 3.5 character times

- At 9600 baud: ~3.5 ms
- At 19200 baud: ~1.75 ms

Master must wait this long before sending the next message.

See also: Inter-Character Timeout

Referenced in:

- Modbus Technical Reference - Message Timing
- Troubleshooting - Timing Issues

L

Little Endian

Byte ordering where the least significant byte (LSB) is stored or transmitted first. Opposite of Modbus specification, but some devices use it.

Example: Value 0x1234 is transmitted as 0x34, 0x12

Note: Not standard Modbus. Causes compatibility problems.

See also: Big Endian, Byte Ordering

Referenced in:

- Modbus Technical Reference - Byte Ordering
- Troubleshooting - Data Interpretation

LRC (Longitudinal Redundancy Check)

A simple checksum used in Modbus ASCII. Calculated as the two's complement of the sum of all bytes, keeping only the least significant byte.

Note: Modbus ASCII is rarely used. Use Modbus RTU or TCP instead.

See also: CRC

Referenced in:

- Modbus Technical Reference - Modbus ASCII

Listen Only Mode

A special diagnostic mode (Function Code 08, Sub-function 04) where a device monitors Modbus traffic but does not respond to any messages. Used to isolate malfunctioning devices.

Characteristics:

- Device monitors all traffic
- No responses sent (even for addressed messages)
- No actions taken
- Can only exit via Restart Communications Option (FC 08, Sub-function 01)

Referenced in:

- Modbus Protocol - Diagnostics

M

Master (Deprecated)

Legacy term for the device that initiates Modbus communication. In modern terminology, use **Client** instead.

The master/slave terminology has been replaced with client/server to better reflect the request/response nature and avoid problematic language.

See also: Client, Server

Referenced in:

- Modbus Protocol - Terminology

MBAP (Modbus Application Protocol Header)

The 7-byte header used in Modbus TCP to encapsulate the PDU:

Field	Size	Description
Transaction ID	2 bytes	Matches request with response
Protocol ID	2 bytes	Always 0x0000 for Modbus
Length	2 bytes	Bytes to follow (Unit ID + PDU)
Unit ID	1 byte	Modbus slave address

Total MBAP header: 7 bytes + PDU

See also: ADU, PDU

Referenced in:

- Modbus Technical Reference - Modbus TCP
- Modbus Protocol - Modbus TCP

MEI (Modbus Encapsulated Interface)

A transport mechanism for tunneling service requests inside Modbus PDUs. Function Code 43 (0x2B) with MEI Type field.

MEI Types:

- **Type 13 (0x0D)**: CANopen General Reference
- **Type 14 (0x0E)**: Read Device Identification
- **Types 0-12, 15-255**: Reserved

Referenced in:

- Modbus Protocol - Function Code 43

Modbus ASCII

A variant of Modbus RTU where all data is encoded as ASCII hexadecimal characters. Each byte becomes two ASCII characters.

Characteristics:

- Messages start with `:` (colon)
- Messages end with `\r\n` (CR+LF)
- Uses LRC instead of CRC

- Double the message length of RTU
- Human-readable but inefficient

Recommendation: Don't use Modbus ASCII. Use Modbus RTU or TCP instead.

Referenced in:

- Modbus Technical Reference - Modbus ASCII

Modbus RTU

The binary serial implementation of Modbus. Most common serial protocol variant.

Characteristics:

- Binary encoding (compact)
- CRC-16 for error detection
- Physical layer: RS-232, RS-485, RS-422
- 1 master + up to 247 slaves (RS-485)
- Baud rates: typically 9600-115200

Message structure: `[Unit ID][Function Code][Data][CRC]`

See also: ADU, CRC

Referenced in:

- Modbus Technical Reference - Modbus RTU
- Getting Started - RTU Configuration

Modbus TCP

The Ethernet/IP implementation of Modbus. Uses TCP port 502.

Characteristics:

- Standard TCP/IP transport
- MBAP header instead of Unit ID and CRC
- No CRC (TCP provides error detection)
- Multiple clients possible
- Fast (millisecond response times)
- Long distance (global via Internet)

Message structure: `[MBAP Header][Function Code][Data]`

See also: MBAP, ADU

Referenced in:

- Modbus Technical Reference - Modbus TCP
- Getting Started - TCP Configuration
- Modbus Client - TCP Client

Multi-Drop Network

A network topology where multiple devices share the same communication bus. Used in RS-485 Modbus RTU.

Characteristics:

- One master, multiple slaves (up to 247)
- Each slave has unique address (1-247)
- Address 0 is broadcast
- All slaves hear all messages
- Only addressed slave responds
- Bandwidth shared among all devices

See also: RS-485, Unit ID

Referenced in:

- Modbus Technical Reference - Multi-Drop Networks
- Troubleshooting - Multi-Drop Issues

O

Off-By-One Error

The common confusion between Modbus public addressing (1-based with prefix) and protocol addressing (0-based without prefix).

Example: Holding Register public address **40001** = protocol address **0** (with function code 0x03)

Conversion formula:

```
Protocol Address = Public Address - Prefix - 1

Public 40108 → Remove "40" prefix → 108 → Subtract 1 → Protocol 107
```

See also: Addressing Model, Protocol Addressing, Public Addressing

Referenced in:

- Modbus Technical Reference - Off-By-One
- Getting Started - Understanding Addresses

P

Parity

An error-detection method in serial communication. A parity bit is added to each character.

Parity settings:

- **None**: No parity bit (most common for Modbus)
- **Even**: Parity bit set so total 1s is even
- **Odd**: Parity bit set so total 1s is odd

Important: All devices on network must use same parity setting.

Referenced in:

- Getting Started - Serial Settings
- Troubleshooting - Serial Issues

PDU (Protocol Data Unit)

The core Modbus message independent of transport layer. Contains:

Structure: `[Function Code][Data]`

Maximum PDU size: 253 bytes

- Inherited from first Modbus serial line implementation
- Maximum RS-485 ADU = 256 bytes
- PDU = 256 - Unit ID (1) - CRC (2) = 253 bytes

Modbus TCP: MBAP (7 bytes) + PDU (253 bytes) = 260 bytes maximum

See also: ADU, Function Code

Referenced in:

- Modbus Protocol - PDU Structure
- Modbus Technical Reference - Message Structure

Protocol Addressing

The 0-based addressing scheme used in actual Modbus messages. Addresses range from 0 to 65535.

No data type prefix - the function code determines data type.

Example: To access holding register public address 40001, use protocol address 0 with function code 0x03.

See also: Public Addressing, Off-By-One Error

Referenced in:

- Modbus Technical Reference - Protocol Addressing
- Modbus Protocol - Addressing

Public Addressing

The 1-based addressing scheme with data type prefix used in documentation and HMI screens. Human-friendly but not used in actual messages.

Format: `[Data Type Prefix][Address starting from 1]`

Prefixes:

- **0xxxx**: Coils (00001-065536)
- **1xxxx**: Discrete Inputs (10001-165536)
- **3xxxx**: Input Registers (30001-365536)
- **4xxxx**: Holding Registers (40001-465536)

5-digit vs 6-digit: Original 5-digit format supports up to 9,999 elements. 6-digit format (with leading zero) supports full 65,536 range.

See also: Protocol Addressing, Off-By-One Error

Referenced in:

- Modbus Technical Reference - Public Addressing
- Getting Started - Address Formats

Q

Quantity

The number of data elements (coils, inputs, or registers) to read or write in a single operation.

Limits vary by function code:

- Read Coils/Discrete Inputs: 1-2000
- Read Holding/Input Registers: 1-125
- Write Multiple Coils: 1-1968
- Write Multiple Registers: 1-123

Note: Limited by maximum PDU size of 253 bytes.

Referenced in:

- Modbus Protocol - Function Codes

R

Register

A 16-bit data storage element in Modbus. Two types:

- **Holding Registers**: Read/write (Function codes 03, 06, 16, 22, 23)
- **Input Registers**: Read-only (Function code 04)

See also: Holding Register, Input Register

Referenced in:

- Modbus Protocol - Data Model
- Modbus Client - Working with Registers

Request

A Modbus message sent from client to server asking for an operation to be performed.

Contains:

- Function code (what operation)
- Data address (where)
- Quantity (how many)
- Additional data (for write operations)

See also: Response, Client

Referenced in:

- Modbus Protocol - Request/Response

Response

A Modbus message sent from server to client in reply to a request.

Two types:

1. **Normal Response**: Contains requested data or confirmation
    - Function code echoed from request
    - Response data
2. **Exception Response**: Indicates error
    - Function code = request function code + 0x80
    - Exception code

See also: Request, Exception Response

Referenced in:

- Modbus Protocol - Request/Response
- Modbus Protocol - Exception Responses

RS-232

A point-to-point serial communication standard. Originally designed for modem connections.

Characteristics:

- Maximum distance: ~50 feet (15 m)
- Point-to-point only (2 devices)
- Voltage levels: ±3V to ±15V
- Connectors: DB9 or DB25
- Minimum 3-wire: TX, RX, GND

Limitations:

- Cannot multi-drop
- Limited distance
- Susceptible to noise

Referenced in:

- Modbus Technical Reference - RS-232
- Troubleshooting - RS-232 Issues

RS-422

A differential serial standard similar to RS-485 but full-duplex with separate TX/RX pairs.

Characteristics:

- 4-wire (separate TX and RX pairs)
- Full-duplex
- Distance: up to 4000 feet
- One transmitter, up to 10 receivers

Note: Less common for Modbus; most use RS-485 (half-duplex).

Referenced in:

- Modbus Technical Reference - RS-422

RS-485

The workhorse serial standard for industrial multi-drop Modbus networks.

Characteristics:

- Differential signaling (2 wires: A and B)
- **Actually needs 3 conductors**: A, B, and ground reference
- Distance: up to 4000 feet (1200 m)
- Multi-drop: up to 32-128 devices
- Termination: 120Ω resistors at both ends only
- Topology: Linear trunk (no star!)

Common issues:

- Wrong polarity (A/B reversed)
- Missing termination
- Star topology (forbidden)
- Missing ground reference

See also: Multi-Drop Network, Termination

Referenced in:

- Modbus Technical Reference - RS-485

- Troubleshooting - RS-485 Problems

S

Scaling

The practice of multiplying values by a factor to represent decimals using integer registers.

Common scaling factors:

- Temperature: ×10 (285 = 28.5°C)
- Voltage: ×10 (2450 = 245.0V)
- Current: ×100 (1234 = 12.34A)
- Energy: ×100 or ×1000

Important: Scaling is device-specific. Always check the device manual.

Referenced in:

- Modbus Technical Reference - Scaling
- Troubleshooting - Wrong Values

Server

The device that responds to Modbus requests (formerly called "slave"). The server executes commands from clients and returns responses.

Typical servers: PLCs, drives, meters, I/O modules, sensors

See also: Client, Slave

Referenced in:

- Modbus Protocol - Client-Server Model
- Simulator - Server Configuration

Slave (Deprecated)

Legacy term for the device that responds to Modbus communication. In modern terminology, use **Server** instead.

The master/slave terminology has been replaced with client/server to better reflect the request/response nature and avoid problematic language.

See also: Server, Unit ID

Referenced in:

- Modbus Protocol - Terminology

T

TCP/IP (Transmission Control Protocol/Internet Protocol)

The standard Internet protocol suite. Modbus TCP uses TCP/IP for reliable, connection-oriented communication over Ethernet.

Modbus TCP specifics:

- Standard port: 502
- Connection-oriented (TCP)
- Reliable delivery
- Error detection via TCP checksums

See also: Modbus TCP, MBAP

Referenced in:

- Modbus Technical Reference - Modbus TCP
- Getting Started - TCP Connection

Termination Resistor

A 120Ω resistor placed at both ends of an RS-485 trunk to absorb signal reflections and prevent ringing.

Critical requirements:

- Value: 120Ω (matches cable characteristic impedance)
- Power: 0.25W sufficient
- Location: **Both ends of trunk ONLY**
- Never in the middle
- Never more than 2 total

Without termination:

- Signals reflect off cable ends
- Reflections cause errors
- Especially problematic at high baud rates

See also: RS-485

Referenced in:

- Modbus Technical Reference - Termination
- Troubleshooting - RS-485 Wiring

Timeout

The maximum time a client waits for a server response before considering the request failed.

Typical timeout values:

- Modbus TCP: 1-5 seconds
- Modbus RTU: 100ms - 2 seconds (depends on baud rate and network size)

Timeout too short: False failures due to legitimate delays **Timeout too long**: Slow detection of actual failures

Referenced in:

- Modbus Client - Timeout Settings
- Troubleshooting - Timeout Issues

Transaction

A complete Modbus communication cycle: request from client → processing by server → response to client.

Components:

1. Client sends request
2. Server receives and validates
3. Server performs action
4. Server sends response (or exception)
5. Client receives and processes response

Transaction ID: In Modbus TCP, a 2-byte identifier that matches requests with responses, allowing multiple outstanding transactions.

See also: MBAP, Request, Response

Referenced in:

- Modbus Protocol - Transaction Processing
- Modbus Technical Reference - Modbus TCP

U

Unit ID (Slave Address)

A 1-byte field identifying the target Modbus device.

Address ranges:

- **0**: Broadcast (all devices, no response)
- **1-247**: Individual device addresses
- **248-255**: Reserved

In Modbus RTU: Part of the ADU, included in CRC calculation

In Modbus TCP: Part of MBAP header, not in CRC (no CRC in TCP)

- Typically 1 for single device
- Allows multiple Modbus devices behind single IP (gateway scenario)
- 0xFF (255) for broadcast (rarely used in TCP)

Important: Each device on multi-drop network must have unique address.

See also: Multi-Drop Network, Broadcast

Referenced in:

- Modbus Protocol - Unit ID
- Getting Started - Device Addressing
- Discovery - Address Scanning

V

Vendor-Specific Function Codes

Function codes in ranges 65-72 and 100-110 reserved for user/vendor-defined functions.

Characteristics:

- Not standardized
- Device manufacturer specific
- No guarantee of uniqueness
- Consult device manual for details

Public function codes: 1-64, 73-99, 111-127 (standardized or reserved)

Referenced in:

- Modbus Protocol - Function Code Categories

W

Word

A 16-bit data unit (2 bytes). In Modbus:

- Registers are word-sized (16 bits)
- Big endian: High byte first, low byte second
- Range: 0-65535 (unsigned) or -32768 to +32767 (signed)

Word order: For multi-word data types (32-bit, 64-bit), determines which register contains high word vs low word.

See also: Byte Ordering, Register

Referenced in:

- Modbus Technical Reference - Data Types
- Modbus Technical Reference - Byte Ordering

Write Operation

A Modbus function that modifies data on the server (coils or registers).

Write function codes:

- FC 05: Write Single Coil
- FC 06: Write Single Register

- FC 15: Write Multiple Coils
- FC 16: Write Multiple Registers
- FC 22: Mask Write Register
- FC 23: Read/Write Multiple Registers

Response: Server echoes the request (or returns exception)

See also: Coil, Holding Register

Referenced in:

- Modbus Protocol - Write Functions
- Modbus Client - Writing Data

Cross-Reference Index

By Manual Section

Getting Started

- ADU
- Baud Rate
- Client
- Modbus RTU
- Modbus TCP
- Off-By-One Error
- Parity
- Protocol Addressing
- Public Addressing
- Server
- TCP/IP
- Unit ID

Modbus Protocol

- Coil
- Data Model
- Discrete Input
- Exception Code
- Exception Response
- Function Code
- Holding Register
- Input Register
- MBAP
- PDU
- Request

- Response
- Transaction
- Write Operation

Modbus Technical Reference

- Addressing Model
- Asynchronous Serial Transmission
- Big Endian
- Byte Ordering
- CRC
- Gateway
- Inter-Character Timeout
- Inter-Frame Delay
- Little Endian
- Modbus ASCII
- Multi-Drop Network
- Off-By-One Error
- RS-232
- RS-422
- RS-485
- Scaling
- Termination Resistor
- Word

Modbus Client

- Client
- Coil
- Discrete Input
- Holding Register
- Input Register
- Timeout
- Write Operation

Simulator

- Exception Code
- Function Code
- Server

Discovery

- Broadcast
- Multi-Drop Network
- Unit ID

Troubleshooting

- Baud Rate
- CRC
- Exception Code
- Exception Response
- Gateway
- Inter-Character Timeout
- Inter-Frame Delay
- Parity
- RS-232
- RS-485
- Scaling
- Termination Resistor
- Timeout

Abbreviations

Abbreviation	Full Term
ADU	Application Data Unit
ASCII	American Standard Code for Information Interchange
CRC	Cyclic Redundancy Check
FC	Function Code
FIFO	First-In-First-Out
HDLC	High-Level Data Link Control
HMI	Human-Machine Interface
I/O	Input/Output
IP	Internet Protocol
LRC	Longitudinal Redundancy Check
LSB	Least Significant Bit/Byte
MAC	Media Access Control
MBAP	Modbus Application Protocol (Header)
MEI	Modbus Encapsulated Interface
MSB	Most Significant Bit/Byte
OSI	Open Systems Interconnection
PDU	Protocol Data Unit

PLC	Programmable Logic Controller
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
TCP	Transmission Control Protocol
VFD	Variable Frequency Drive

Document version: 1.0
Last updated: Based on Modbus Application Protocol Specification V1.1b3 (April 26, 2012)

Appendix: Indexes

This appendix is generated automatically from the manual manifest.

Explorer Guides

- Introduction
- Getting Started
- Modbus Client
- Device Discovery
- Modbus Simulator
- Troubleshooting
- Support

Troubleshooting Topics

- Triage Checklist: First Steps in Troubleshooting
- Troubleshooting Timeout Errors
- Decoding Modbus Exception Responses
- Investigating Garbled Data

Protocol Fundamentals

- Modbus Protocol Reference
- Modbus Addressing
- Modbus Data Model

- Modbus Message Structure
- Modbus Exception Responses

Function Codes

- FC01 - Read Coils
- FC02 - Read Discrete Inputs
- FC03 - Read Holding Registers
- FC04 - Read Input Registers
- FC05 - Write Single Coil
- FC06 - Write Single Register
- FC08 - Diagnostics
- FC11 - Get Comm Event Counter
- FC15 - Write Multiple Coils
- FC16 - Write Multiple Registers
- FC23 - Read/Write Multiple Registers
- FC43 / MEI 14 - Read Device Identification

Exception Codes

- Exception 01 - Illegal Function
- Exception 02 - Illegal Data Address
- Exception 03 - Illegal Data Value
- Exception 04 - Server Device Failure
- Exception 05 - Acknowledge
- Exception 06 - Server Device Busy
- Exception 07 - Negative Acknowledge
- Exception 08 - Memory Parity Error
- Exception 10 (0x0A) - Gateway Path Unavailable
- Exception 11 (0x0B) - Gateway Target Device Failed to Respond

Other Appendices

- Glossary of Modbus Terms