Advancing Plasma-Based Technologies

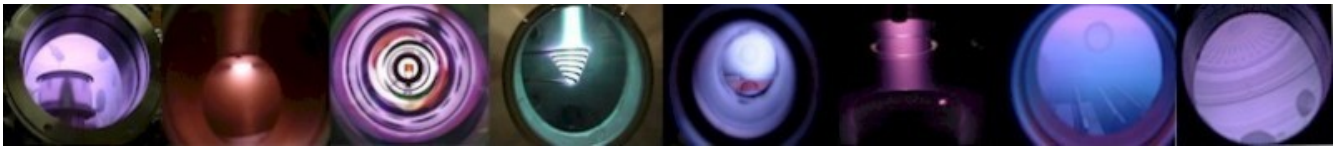
PLASMIONIQUE

À l'Avant-Garde des Technologies Plasmas

Modbus Master Library

User Guide

Rev. 1.3



Contents

1 Introduction

The Plasmionique Modbus Master Library is an open source add-on package for LabVIEW. It implements the Modbus Application Protocol Specification V1.1b3 for communicating with Modbus devices (slaves) over Asynchronous Serial or TCP/IP networks. It has been developed as a replacement for NI's Modbus V1.2.1 and to provide an open source alternative to the Modbus API released by NI labs.

This document describes the system requirements, installation procedure and usage of the API.

2 System Requirements

Software:

- National Instruments LabVIEW 2012
- JKI VI Package Manager 2017/2018

3 Installation

Download the latest version of the library from: https://lavag.org/files/file/286-plasmionique-modbus-master/

Install the ".vip" file using VI Package Manager.

4 Examples

Examples are included in "<LabVIEW>\examples\Plasmionique\MB Master\":

- **MB_Master Comm Tester.vi**: Demonstrates usage of API to open/close connection and communicate with a Modbus slave device.
- **MB_Master Multiple Sessions.vi**: Demonstrates usage of API to open concurrent Modbus sessions.
- **MB_Master Simple Serial.vi**: Demonstrates polling of a single input register over serial line.

These examples can also be found via Example Finder within LabVIEW.

5 Support

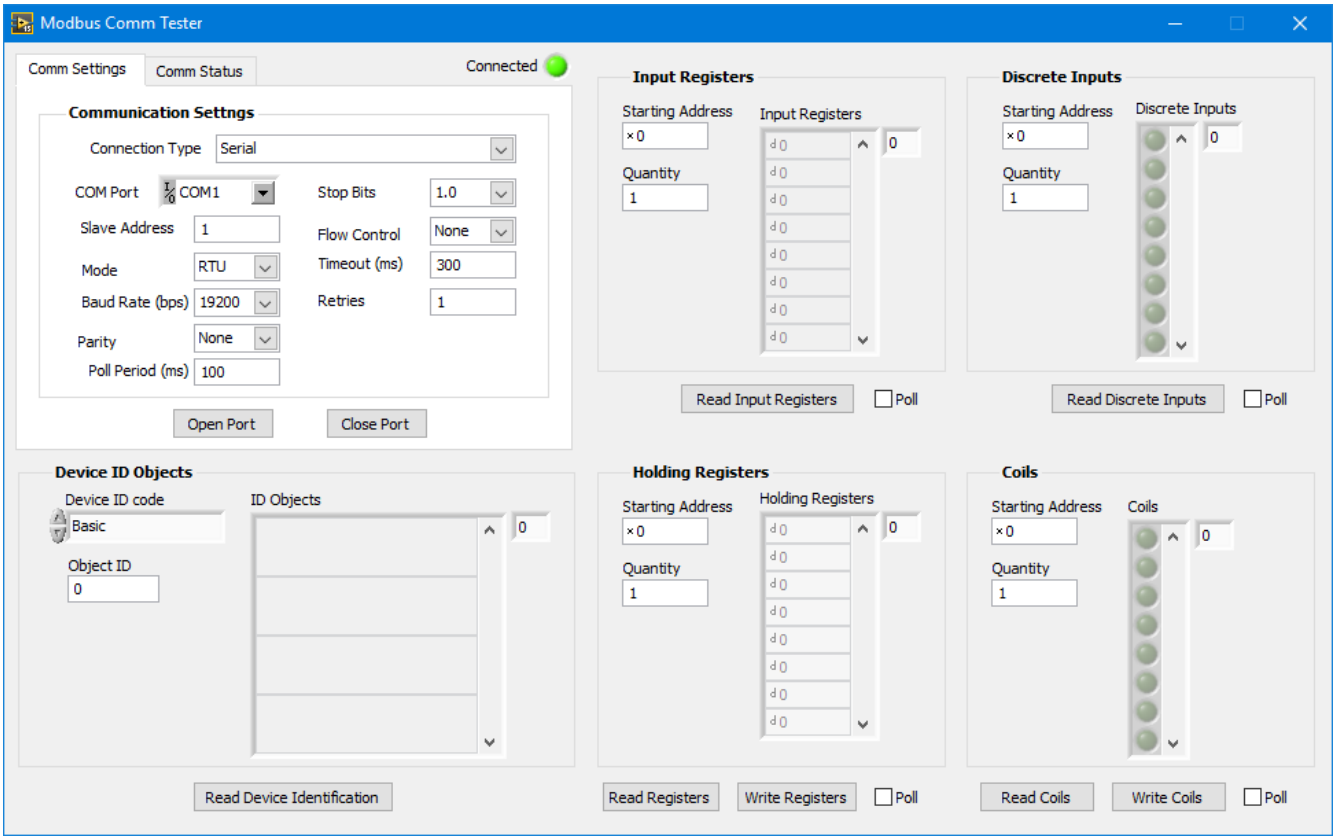
If you have any problems with this library or want to suggest changes contact Porter via PM on lavag.org or post your comment on the support forum: https://lavag.org/topic/19544-cr-plasmionique-modbus-master/

The development source code is available on GitHub: https://github.com/rfporter/Modbus-Master

6 Modbus Comm Tester

Included with this library is a tool for testing Modbus communication with your device, allowing you to determine the correct communication parameters and data addresses before writing any code.

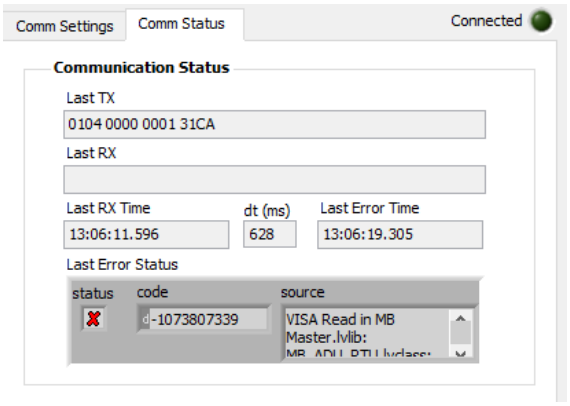
The Modbus Comm Tester can be launched from the “Tools > Plasmionique” menu of the LabVIEW development environment.



6.1 Usage

- Run the VI
- Enter the Connection Type (Serial or TCP) then fill in the connection parameters.
- To start the Modbus session, click the Open Port button.
- To close the Modbus session, click the Close Port button.
- The green LED indicator will be lit when the session is open.
- The data starting addresses are input in hexadecimal notation. If decimal notation is preferred, click the radix selector (x) and change to d for decimal notation.
- Note that input registers, discrete inputs, holding registers, and coils can be polled at the specified polling period by checking their “Poll” box.

6.2 Communication Status



- Last TX: The last Modbus message, in hexadecimal notation, sent from the master.
- Last RX: The last Modbus message, in hexadecimal format, received by the master.
- Last RX Time: Time that the last message was received by the mater.
- dt: Duration, in milliseconds, of the last Modbus action or transaction.
- Last Error Time: Time that the last error occurred.
- Last Error Status: Display the error code and source of the last error. Right-click and select explain error for a more information about the error.

7 Modbus Palette

The Modbus Master API is located in the LabVIEW functions palette under “Data Communication > Modbus Master”.

The top row contains functions for managing the Modbus Session. The property node can be used to access session data.

Modbus transaction functions are listed on subsequent rows. Each one implements a particular function code from the Modbus Application Protocol Specification. See: (MODBUS Application Protocol Specification V1.1b3). They encapsulate sending the request to the slave, waiting for a response, and validating and interpreting the response. Exception codes and timeouts are placed on their error out terminal.

Function codes currently supported are:

- 0x01 - Read Coils
- 0x02 - Read Discrete Inputs
- 0x03 - Read Holding Registers
- 0x04 - Read Input Registers
- 0x05 - Write Single Coil
- 0x06 - Write Single Register
- 0x07 - Read Exception Status
- 0x0F - Write Multiple Coils
- 0x10 - Write Multiple Registers
- 0x16 - Mask Write Register
- 0x17 - Read/Write Multiple Registers
- 0x2B/0x0E - Read Device Identification

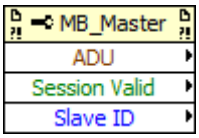
8 Modbus Session

The Modbus session keeps track of the type of connection, slave ID and manages the communication bus. A Modbus session must be opened in order to establish a connection with a slave device. It should be closed when it is no longer needed in order to release system resources.

Two types of Modbus sessions are implemented:

- **Asynchronous Serial**: Modbus over RS-232, RS-422 or RS-485 serial line. Can be configured for ASCII or RTU mode. See: (MODBUS over Serial Line Specification & Implementation Guide V1.02)
- **TCP/IP**: Modbus over TCP/IP network. See: (MODBUS Messaging on TCP/IP Implementation Guide V1.0b)

8.1 Session Properties

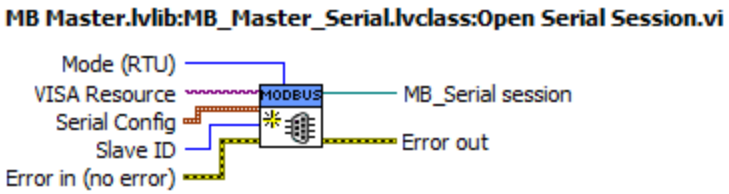


Placing the property node on a Modbus session wire will reveal the following properties:

- **ADU** (Read Only): The Application Data Unit (Sent and received) for the last Modbus transaction. Use an additional property node on the ADU wire to access the RX/TX data and timestamps.
- **Session Valid** (Read Only): Indicates if the Modbus session is open and properly initialized.
- **Slave ID** (Read/Write): The address of the slave device (1 to 247). A slave ID of zero specifies broadcast mode for sending commands to all slaves on the communication bus.

8.2 Opening a Modbus Serial Session

To open an Asynchronous Serial Modbus Session, use “Open Serial Session.vi” from the functions palette.



Inputs:

- **Mode**: Select either ASCII or RTU mode. RTU is default.
- **VISA Resource**: Specify the COM port that the slave is connected to.
- **Serial Config**: Cluster of serial port configuration parameters.
    - **Baud rate**: Baud rate in bps. Commonly used values include 9600, 19200, 38400, 57600 and 115200 bps.
    - **Stop bits**: Number of stop bits.

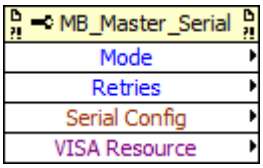
Number of Stop Bits	Value
1.0	10
1.5	15
2.0	20

  - **Parity**: Type of parity bit {None, Odd, Even, Mark, Space}.
  - **Flow control**: Type of flow control {None, XON/XOFF, RTS/CTS, XON/XOFF&RTS/CTS, DTR/DSR, XON/XOFF&DTR/DSR}.
  - **Timeout**: Communication timeout in ms. If the slave device does not respond to a request within this period, a timeout error is generated.
  - **Retries**: Number of times to retry a Modbus transaction before aborting and reporting the error.
- **Slave ID**: The address of the slave device (1 to 247). A slave ID of zero specifies broadcast mode for sending commands to all slaves on the communication bus. Note that the Slave ID can be changed later using the property node.

Outputs:

- **MB_Serial Session**: Asynchronous Serial Modbus Session object. Use this wire to perform Modbus transactions.

8.2.1 Serial Session Properties

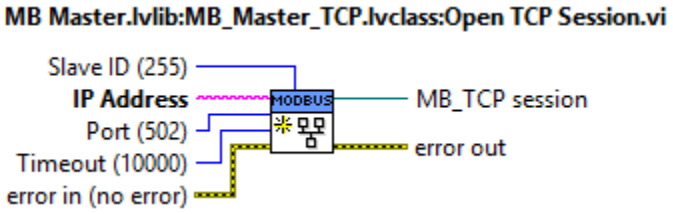


Placing the property node on the Asynchronous Serial Modbus Session wire reveals additional parameters:

- **Mode** (Read Only): Indicates ASCII or RTU mode.
- **Retries** (Read/Write): Number of times to retry a Modbus transaction before aborting and reporting the error.
- **Serial Config** (Read Only): Cluster of serial port configuration parameters.
- **VISA Resource** (Read/Write): VISA session reserved by the Modbus Session. This is provided in case some additional configuration of the serial port is required after the session has been opened.

8.3 Opening a Modbus TCP/IP Session

To open TCP/IP Modbus Session, use “Open TCP Session.vi” from the functions palette.



Inputs:

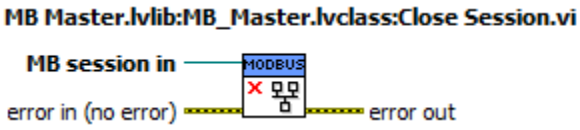
- **IP Address**: IP address or hostname of the slave device (Modbus server).
- **Port**: Port that the slave device is listening on. Default port is 502.
- **Timeout**: Communication timeout in ms to use for TCP read/write functions.
- **Slave ID**: Equivalent to the serial slave ID. This setting may be required when communicating with a serial slave through a Modbus gateway. By default slave ID is set to 255. Note that the Slave ID can be changed later using the property node.

Outputs:

- **MB_TCP Session**: TCP/IP Modbus Session object. Use this wire to perform Modbus transactions.

8.4 Closing a Modbus Session

To close any Modbus Session, use “Close Session.vi” from the functions palette.



8.5 Concurrent Modbus Sessions

Any number of independent Modbus Master Sessions can be opened and transactions can run concurrently without conflict.

Sessions are not independent if:

a. **Multiple sessions communicate with the same slave device.** For TCP/IP sessions this is not an issue. Each session is assigned a unique local port number. For serial sessions, transactions are forced to run consecutively due to an exclusive VISA lock on the comm port (with a timeout of 10 seconds). That is, if a session's transaction locks the comm port for more than 10 seconds, pending transactions from other sessions may time out.

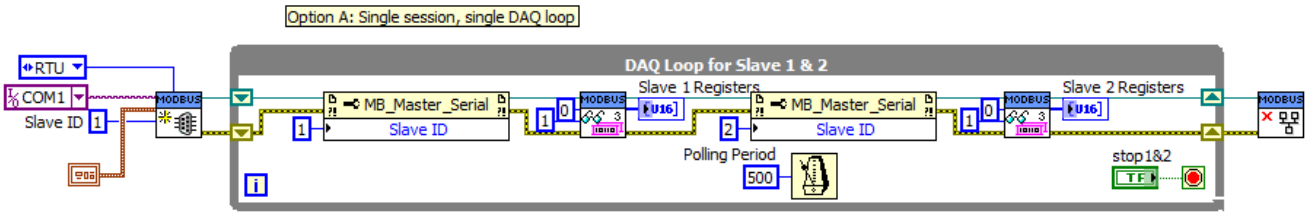
b. **Multiple sessions are connected to the same communication bus.** For TCP/IP sessions this is not an issue. Each session is assigned a unique local port number. For serial sessions, transactions are forced to run consecutively due to an exclusive VISA lock on the comm port (with a timeout of 10 seconds). That is, if a session's transaction locks the comm port for more than 10 seconds, pending transactions from other sessions may time out.

c. **A session wire is branched, allowing multiple copies of the session to run concurrently.** In this situation, transactions from all copies are forced to run consecutively due to a session-specific mutex lock. That is, of one if one session is performing a transaction, pending transactions from other sessions are forced to wait (indefinitely) until the transaction is complete.

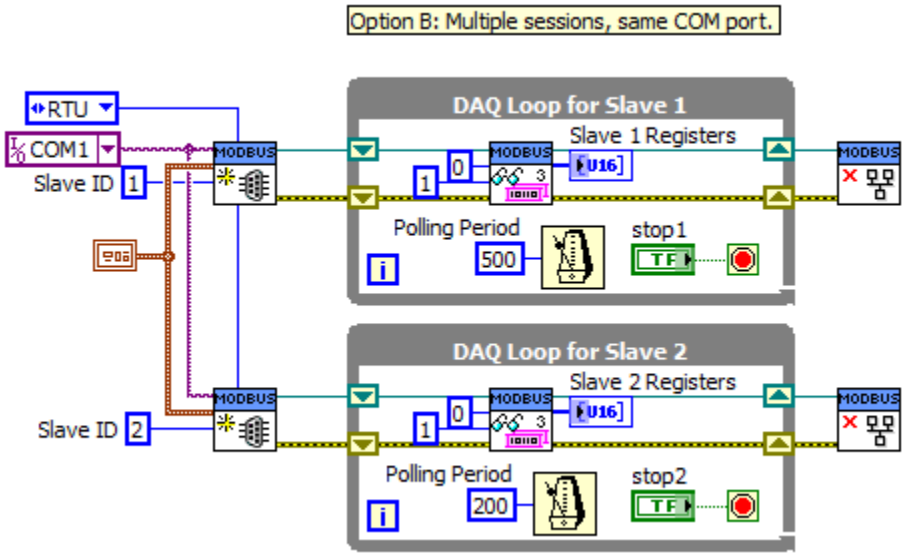
   Note that calling "Close Session" on any one of the copies will close all copies of the session. Further requests on these session wires will return error 403483 "session invalid".

Although interdependent Modbus sessions require some special consideration, there are situations where they are very useful. For example, if you need to communicate with multiple slaves connected to a single RS-485 bus, you could:

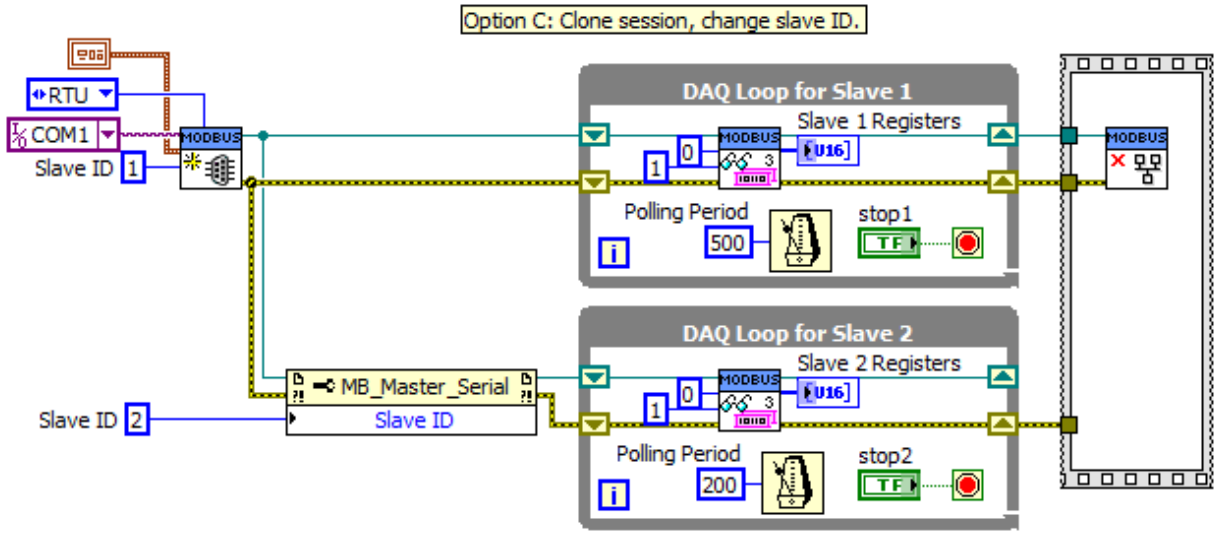
a. Open a single session and use the property node to change the Slave ID for each transaction. This avoids interdependent sessions but does not scale very well. Communication with all slave devices must be implemented in the same data acquisition loop. It is more difficult to implement different data polling rates for each slave. It also becomes very messy to handle device-specific data types or error cases.



b. Open multiple sessions, each using the same “Mode”, “Serial Config” and “VISA Resource” but unique Slave IDs (this is an example of case B above). Each session can run in its own, unique, data acquisition loop as long as the 10 second transaction time limit is respected. This is the preferred solution but care must be taken to ensure that all sessions are configured identically.



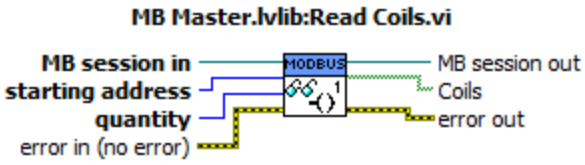
c. Open a single session then branch the session wire and use the property node to set the Slave ID of each branch (this is an example of case C above). Each branched session can run in its own, unique, data acquisition loop. This method guarantees that all sessions have the same configuration however, if one session is closed, the other sessions will be closed as well.



9 Transaction Functions

The Modbus Master API has one VI for each function code of the Modbus Application Specification.

9.1 Read Coils



Function Code 1: Reads **quantity** number of coils starting from **starting address**.

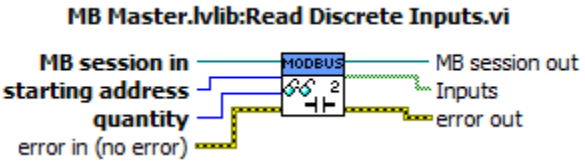
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Starting address** (U16): Address of first coil to read.
- **Quantity** (U16): Number of coils to read.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **Coils**: Array of coil values (Boolean). The value of the first coil is at index zero.

9.2 Read Discrete Inputs



Function Code 2: Reads **quantity** number of discrete inputs starting from **starting address**.

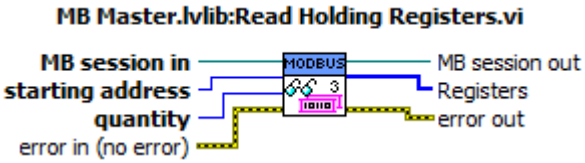
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Starting address** (U16): Address of first discrete input to read.
- **Quantity** (U16): Number of discrete inputs to read.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **Inputs**: Array of discrete input values (Boolean). The value of the first input is at index zero.

9.3 Read Holding Registers



Function Code 3: Read **quantity** number of holding registers starting from **starting address.**

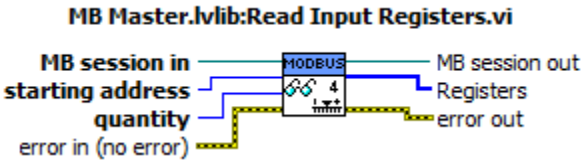
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Starting address** (U16): Address of first holding register to read.
- **Quantity** (U16): Number of holding registers to read.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **Registers**: Array of holding register values (U16). The value of the first holding register is at index zero.

9.4 Read Input Registers



Function Code 4: Read **quantity** number of input registers starting from **starting address.**

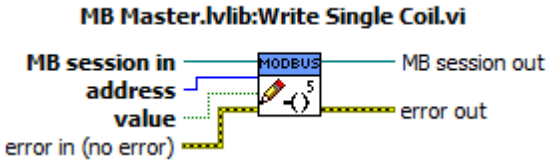
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Starting address** (U16): Address of first input register to read.
- **Quantity** (U16): Number of input registers to read.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **Registers**: Array of input register values (U16). The value of the first input register is at index zero.

9.5 Write Single Coil



Function Code 5: Write **value** to coil at **address.**

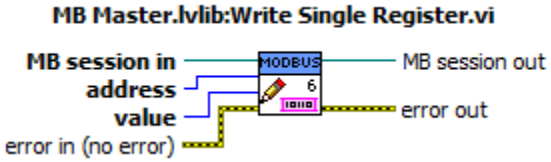
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Address** (U16): Address of the coil.
- **Value** (Boolean): Value to write to the coil.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.

9.6 Write Single Register



Function Code 6: Write **value** to holding register at **address.**

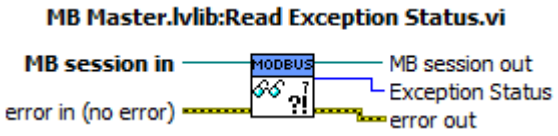
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Address** (U16): Address of the register.
- **Value** (U16): Value to write to the register.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.

9.7 Read Exception Status



Function Code 7: Reads slave exception status.

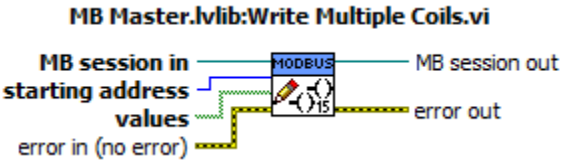
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **Exception Status** (U8): Value of the slave's exception status register.

9.8 Write Multiple Coils



Function Code 15: Writes coil **values** starting at **starting address**.

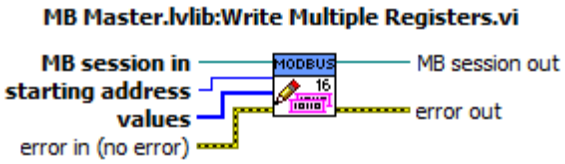
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Address** (U16): Address of the first coil to write.
- **Value**s: Array of values (Boolean) to write. Value for the first coil is at index zero. The number of coils to write to is specified by the size of the array.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.

9.9 Write Multiple Registers



Function Code 16: Writes holding register values starting at **starting address**.

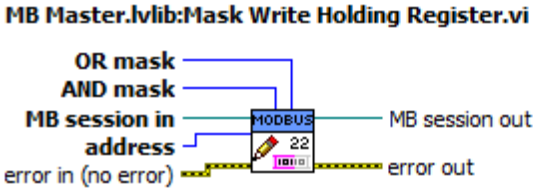
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Address** (U16): Address of the first register to write.
- **Value**s: Array of values (U16) to write. Value for the first register is at index zero. The number of register to write to is specified by the size of the array.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.

9.10 Mask Write Holding Registers



Function Code 22: Modifies value of holding register at **address** based on the **AND mask** and **OR mask**. The slave device implements the following formula to modify the register's value:

Result = (Current Value & And_Mask) | (Or_Mask & !And_Mask)

Set bits to 0 in **AND mask** to specify which bits to modify. Use **OR mask** to specify the value of bits.

Example:

Register Value: 1100 0001
AND Mask: 1111 1100
OR Mask: 0000 0010
Result: 1100 0010

Inputs:

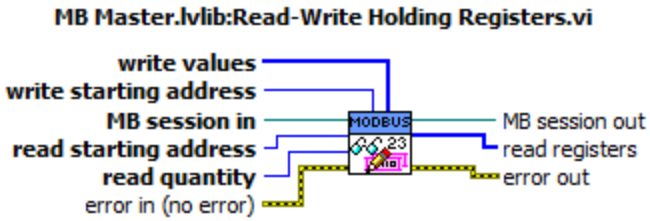
- **MB Session in**: Modbus session on which to perform the transaction.
- **Address** (U16): Address of the register to modify.
- **AND mask** (U16): Set bits of AND mask to zero to specify which bits of the register to modify.

- **OR mask** (U16)**:** Set bits of OR mask to specify the value of the modified bits of the register.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.

9.11 Read-Write Holding Registers



Function Code 23: Writes **write values** to holding registers starting at **write starting address** then reads **read quantity** number of holding registers starting at **read starting address**.

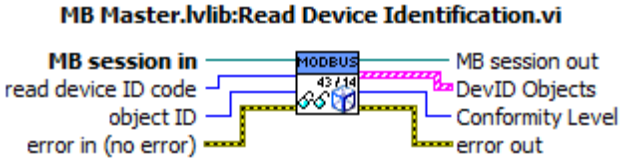
Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Read starting address** (U16): Address of the first register to read.
- **Read quantity** (U16): Number of registers to read.
- **Write starting address** (U16): Address of the first register to write.
- **Value**s: Array of values (U16) to write. Value for the first register is at index zero. The number of register to write to is specified by the size of the array.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **Read registers**: Array of register values (U16). The value of the first register is at index zero.

9.12 Read Device Identification



Function Code 43 / 14: Read device ID objects. Specify the type of access using the **read device ID code** and the id of the starting object using **object ID**.

Inputs:

- **MB Session in**: Modbus session on which to perform the transaction.
- **Object ID** (U8): ID number of first device ID object to read.
- **Read device ID code** (U8): Specifies type of read operation.
    - 0x01 - Basic: Request the basic stream of devID objects (0x00 to 0x02) starting from the specified object ID.
    - 0x02 - Regular: Requests the regular stream of devID objects (0x00 to 0x7F) starting from the specified object ID.
    - 0x03 - Extended: Requests the extended stream of devID objects (0x00 to 0xFF) starting from the specified object ID.
    - 0x04 - Single: Requests a single devID object specified by object ID.

Outputs:

- **MB Session out**: Modbus session updated with transaction's ADU data.
- **DevID Objects**: Array of DevID objects. Each element contains the object ID number and string value.
- **Conformity Level** (U8): Indicates the type of access supported by the device.
    - 0x01 - Basic (stream only)
    - 0x02 - Regular (stream only)
    - 0x03 - Extended (stream only)
    - 0x81 - Basic (stream and single)
    - 0x82 - Regular (stream and single)
    - 0x83 - Extended (stream and single)

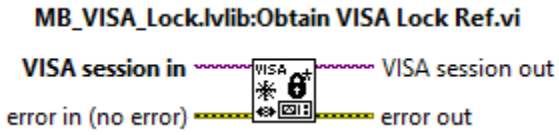
10 MB VISA Locks

The MB VISA Lock library was developed after a flaw was discovered in the behavior of the LabVIEW's VISA lock primitive. This flaw was causing the Modbus Master to behave erratically when communicating with multiple slaves over a shared serial port. The MB VISA Lock library wraps the VISA Lock primitive and forces it to behave as originally expected.

The discussion about the flaw and fix can be found here: https://lavag.org/topic/19871-visa-lock-behavior/

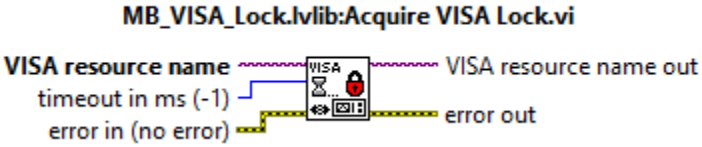
The Modbus Master library calls the MB VISA Lock library internally. For advanced users, the library has been included in the Modbus Palette.

10.1 Obtain VISA Lock Ref



Obtain the reference for the VISA resource lock. This reference is stored in the UserData property of the output VISA session. This VI should be called just after the VISA session is opened.

10.2 Acquire VISA Lock



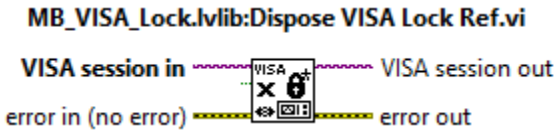
Attempt to acquire an exclusive lock on the VISA resource. If the lock cannot be obtained before the specified timeout, error -1073807345 will be returned.

10.3 Release VISA Lock

Release the session's exclusive lock on the VISA resource.

Input errors do not affect the behavior of this VI.

10.4 Dispose VISA Lock Ref



Dispose of the VISA resource lock reference. The output VISA session will have an invalid reference stored in the User Data property. This VI should be called just before closing the VISA session.

Input errors do not affect the behavior of this VI.

11 Error Codes

The following custom error codes have been defined. They are included in custom error codes file "Plasmionique-MB Master-errors.txt" located in the "<LabVIEW>\project\errors" directory.

Error Code	Description
403461	Modbus Exception Code 1: Illegal function
403462	Modbus Exception Code 2: Illegal data address
403463	Modbus Exception Code 3: Illegal data value
403464	Modbus Exception Code 4: Slave device failure
403465	Modbus Exception Code 5: Slave acknowledge
403466	Modbus Exception Code 6: Slave device busy
403467	Modbus Exception Code 7: Slave NACK
403468	Modbus Exception Code 8: Memory parity error
403470	Modbus Exception Code 10: Gateway path unavailable
403471	Modbus Exception Code 11: Gateway target device failed to respond
403481	Modbus slave ID mismatch
403482	Modbus CRC/LRC error
403483	Invalid Modbus session
403484	Modbus TCP invalid protocol ID
403485	Modbus TCP transaction ID mismatch

12 References

Modbus Organization. (2006). *MODBUS over Serial Line Specification & Implementation Guide V1.02.* Retrieved from Modbus.org: http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf

Modbus Organization. (2012). *MODBUS Application Protocol Specification V1.1b3.* Retrieved from Modbus.org: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

Modbus-IDA. (2006). *MODBUS Messaging on TCP/IP Implementation Guide V1.0b.* Retrieved from Modbus.org: http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf