[APPLICATION NOTE #168]

Modbus Installation and Troubleshooting for AP9635/AP9635CH Network Management Card

By Gary Ware

PROJECT AT A GLANCE

Project Type

Modbus installation

Applications

Data centers with products that use AP9635/AP9635CH Network Management Card 2

Equipment Installed

AP9635/AP9635CH Network Management Card 2

CUSTOMER BENEFITS

- **Centralized data center management from StruxureWare Data Center Expert or a customer-supplied building management system**

Abstract

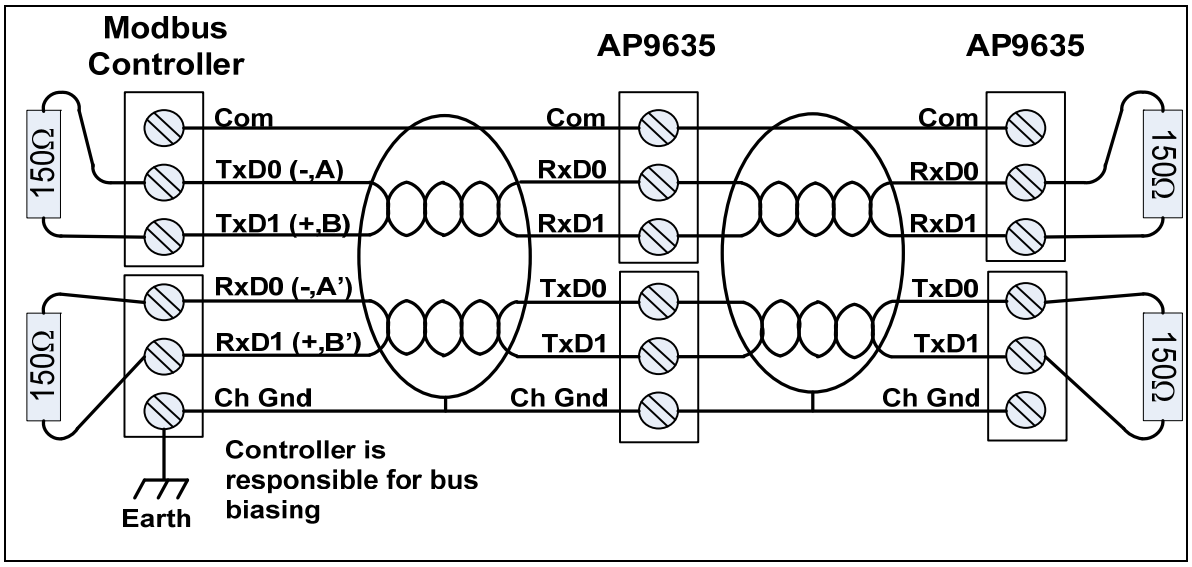
This application note explains how to install the Modbus wiring for an AP9635 Network Management Card, gives an overview of the Modbus protocol, describes the Modbus packet structure, and explains how to identify and resolve common problems.

Introduction

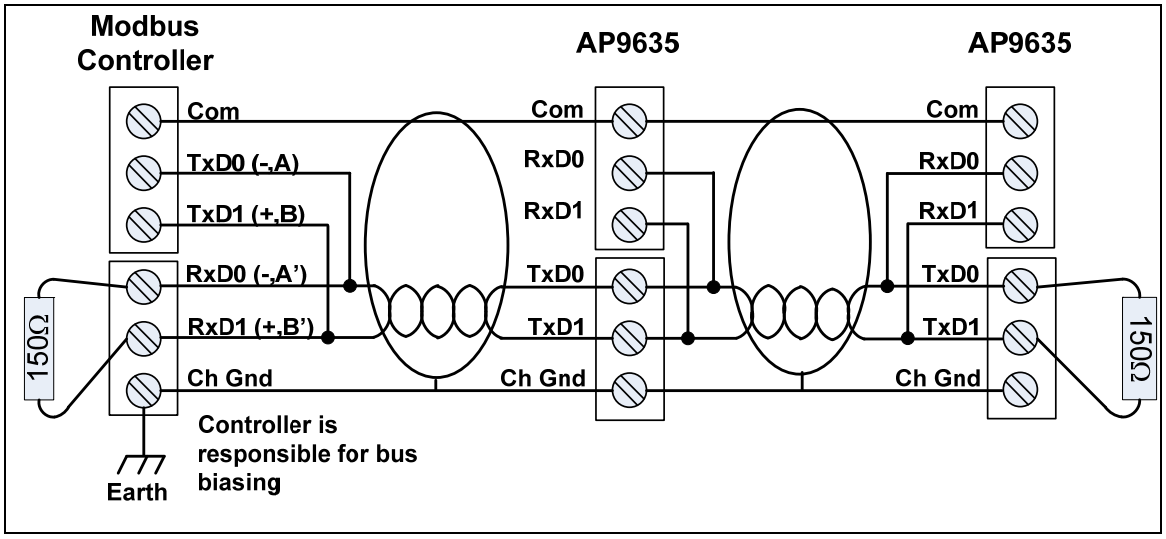
The Modbus standard defines both hardware and protocol. The hardware (media) is based on RS-485 with certain additional constraints.

RS-485 communications can be half-duplex over a 2-wire bus or full duplex over a 4-wire bus. RS-485 is differential signaling, multi-drop (typically up to 32 devices on a bus, however repeaters are available to extend this), and designed for long distances (up to 4000 ft / 1220 meters). It must be a linear bus, not a star. When interconnecting in environments where ground potential differences may exist (such as power devices), or over significant distances, devices with isolated communications ports should be used.

4-wire Connection Diagram



2-wire Connection Diagram



Wiring Guidelines

- Good quality shielded twisted pair cable should be used. Shielded cable is worth the small added cost.
- Wiring should be done in accordance with local wiring codes

- Do not put communications cables and power cables in the same raceways or conduits.
- Route communications cables to avoid potential noise sources such as high power equipment or motors.
- In addition to the 2 or 4 signal lines, the common terminal should be connected at each device and the shield should be connected to chassis at each device. Common should not be connected to chassis. The common connection requires an extra twisted pair in the cable – a total of 2 twisted pairs for a 2-wire bus and 3 twisted pairs for a 4-wire bus.
- The shield should be connected to each device, not just at one end. It should be earth grounded at one point in the system.
- The AP9635 represents 0.5 standard Modbus loads.
- The AP9635 Modbus port is optically isolated. The Modbus port's ground is not connected to any other ground.
- AP9635 requires two 3-contact connectors (provided). (APC Part #730-0532; Phoenix Contact part number: 1952270 FMC 1,5/ 3-ST-3,5)

The Modbus standard specifies 150 ohm termination resistors at each end of a bus. Unless the bus is very long and operating at high data rates these resistors are not needed, and in fact they reduce the signal swing and therefore reduce noise immunity. Busses under 2000 feet operating at 9600 baud or under 1000 feet operating at 19,200 baud should not require termination.

The standard also specifies bias resistors of 400 – 650 ohms, which are normally placed at the system controller or supplied inside the controller; one from D0 to ground and one from D1 to +5VDC.

Setup and Configuration

Modbus requires that every setting be correct for communications to take place. This includes:

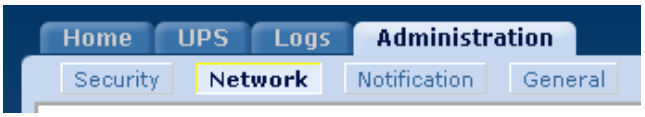
- Baud rate (typically either 9,600 or 19,200)
- Number of data bits (must be 8)
- Parity (typically either none or even)
- Number of stop bits (typically 1 for even parity or 2 for none)
- Device address (in the range of 1 to 247; every device must be different)

Every device on the bus must have exactly the same settings except the device address, which must be unique for every device. No two devices on the bus can have the same address. Serial settings are often abbreviated. For example: "9600,8,N,1" means "9600 bits/sec, 8 data bits, no parity, 1 stop bit".

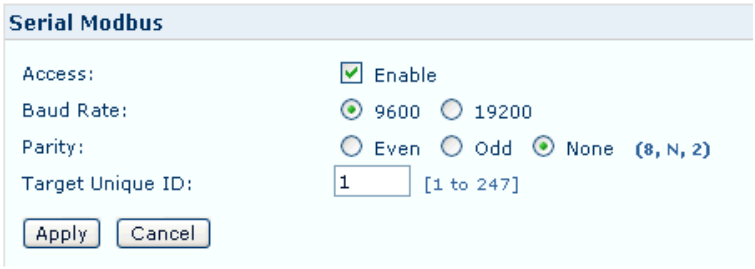
For the AP9635 card, the Modbus configuration can be found by accessing the web page. The IP address of the card is used for access to the web page, so the card must be connected to a network via the Ethernet port and must get an IP address either by static configuration or dynamically via DHCP. One way to view and set this information is via the console port on the

card. This configuration option uses a special cable (APC #940-0299A) supplied with the card. Connect the cable to the DB9 serial port on a laptop and plug the mini-plug into the "Console" port on the card. Use a dumb terminal program (e.g., HyperTerminal or PuTTy), set to 9600,N,8,1. Press "Enter" 2 or 3 times or until you see the prompt: ***Username:*** Enter your username (the default is ***apc***), and for the prompt ***Password:*** enter your password (the default is ***apc***). At the ***APC >*** prompt, type ***tcpip***. If the ***IPv4 Address*** is not all zeros, use that address and enter it into your web browser. If the device does not have an address, see the "AP9635 User Guide". From the ***APC >*** prompt, type ***Exit***.

Log in to the web page (the default user name and password are ***apc***). Navigate to Administration >> Network >> Modbus Serial.



Choose the serial settings and press "Apply": ("Target Unique ID" is the device's Modbus address.) Other devices on the bus must be configured similarly.



Protocol Introduction

This section explains the Modbus protocol for APC devices. The device performs Modbus communications according to the Modbus Application Protocol v1.1. It is assumed that the reader is familiar with the Modbus protocol and serial communications in general.

Purpose of the Modbus Protocol

The Modbus protocol allows data and setup information to be transferred between a Modbus Master and a Modbus Slave. This may include:

- interrogation of all device status, static data and dynamic data
- configuration and interrogation of device setup registers (not supported in some devices)
- control of the device (not supported in some devices)

Ground Rules

The device is capable of communicating via the RS-485 serial communication standard. The RS-485 medium allows for multiple devices on a serial bus network. The following rules define the protocol for information transfer between a Modbus Master device and the slave device:

- All communications on the network conform to a MASTER/SLAVE scheme. In this scheme, information and data is transferred between a Modbus MASTER device and one or more of up to 32 SLAVE devices.
- The MASTER initiates and controls all information transfer on the communications loop.
- A SLAVE device never initiates a communications sequence.
- All communications activity on the loop occurs in the form of “PACKETS.” A packet is a serial string of 8-bit bytes. The maximum number of bytes contained within one packet is 255.
- All PACKETS transmitted by the MASTER are REQUESTS. All PACKETS transmitted by a SLAVE device are RESPONSES.
- At most, one SLAVE can respond to a single request from a MASTER.

Modes of Transmission

The Modbus protocol uses TCP, RTU and ASCII modes of transmission. The AP9635 supports Modbus/TCP and Modbus/RTU protocols. Depending on firmware support the device will require 8 data bits, no parity, and two stop bits (8N2), or 8 data bits, even/odd parity, one stop bit (8E1 or 8O1) for the serial communication.

Description of the Modbus Packet Structure

Every Modbus packet consists of four fields:

- Slave Address Field
- Function Field
- Data Field
- Error Check Field (Checksum)

Slave Address Field

The slave address field of a Modbus packet is one byte in length and uniquely identifies the slave device involved in the transaction. Valid addresses range between 1 and 247. A slave device performs the command specified in the packet when it receives a request packet with the slave address field matching its own address. A response packet generated by the slave has the same value in the slave address field.

Function Field

The function field of a Modbus request packet is one byte in length and tells the addressed slave which function to perform. Similarly, the function field of a response packet tells the master what function the addressed slave has just performed. “Table 1: Modbus Functions Supported by the Device as Slave” on page 5 lists the Modbus functions supported by the device when acting as Slave.

Data Field

The data field of a Modbus request is of variable length, and depends on the function. This field contains information required by the slave device to perform the command specified in a request packet or data being passed back by the slave device in a response packet. Data in this field is contained in 16-bit registers. Registers are transmitted in the order of high-order byte first, low-order byte second.

Example:

A 16-bit register contains the value 12AB Hex. This register is transmitted:

- High order byte = 12 Hex
- Low order byte = AB Hex

This register is transmitted in the order 12 AB.

Error Check Field (Checksum)

The checksum field lets the receiving device determine if a packet is corrupted with transmission errors. In Modbus RTU mode, a 16-bit Cyclic Redundancy Check (CRC-16) is used.

The sending device calculates a 16-bit value, based on every byte in the packet, using the CRC-16 algorithm. The calculated value is inserted in the error check field.

The receiving device performs the calculation, without the error check field, on the entire packet it receives. The resulting value is compared to the error check field. Transmission errors are indicated when the calculated checksum does not equal the checksum stored in the incoming packet. The receiving device ignores a bad packet.

Packet Communications

This section illustrates the Modbus functions supported by the device.

Table 1– Modbus Functions Supported by the Device as Slave

Function	Meaning	Action
03	Read Holding Registers	Obtains the current value in 1 to 125 consecutive holding registers of the device
16	Preset Multiple Registers	Places specific values into 1 to 124 consecutive holding registers of the device. The holding registers that can be written to the device are shown in the register map.

Function 03: Read Holding Registers

To read device parameter values, a Master must send the Slave device a Read Holding Registers request packet.

The Read Holding Registers request packet specifies a start register and a number of registers to read. Up to 125 registers can be read. The start register is numbered from zero (40001 = zero, 40002 = one, etc.).
The device responds with a packet containing the values of the registers in the range defined in the request.

Table 2– Read Holding Registers Packet Structure:

Read Registers Request Packet (Master to Slave)	Read Registers Response Packet (Slave to Master)
Unit ID/Slave Address (1 byte)	Unit ID/Slave Address (1 byte)
03 (Function code) (1byte)	03 (Function code) (1 byte)
Start Register (sr) (2 bytes)	Byte Count (2 x nr) (1 byte)
# of Registers to Read (nr) (2 bytes)	First Register in range (2 bytes)
CRC Checksum	Second Register in range (2 bytes)
	…
	CRC Checksum (2 bytes)

Example:

A 3-phase device is configured as a Modbus slave device with slave address 100. The Master requests to read all three voltage phases (A, B, C) from Modbus registers 40011, 40012 and 40013, with a scaling factor of 10. In accordance with the Modbus protocol, register 40011 is numbered as 10 when transmitted. The request must read 3 registers starting at 10.

Slave address: 100 = 64 (hex) Start register 10 = 000A (hex)

Table 3– Request Packet (white background denotes the DATA field of the packet)

Slave	Function	Start Register (40011)		# of Registers (3)		CRC Checksum	
64	03	00	0A	00	03	2C	3C

Table 4– Response packet

Slave	Function	Byte Count	Register 1		Register 2		Register 3		CRC Checksum	
64	03	06	2E	CE	2E	E8	2F	13	0D	58

The Master station retrieves the data from the response:

Register 40011: 2ECE(hex) = 11982 (scaled: 1198.2)

Register 40012: 2EE8(hex) = 12008 (scaled: 1200.8)

Register 40013: 2F13(hex) = 12051 (scaled: 1205.1)

Note: The values shown in the packets illustrated above are in hexadecimal format.

Function 16: Preset Multiple Registers

The Preset Multiple Registers command packet allows a Modbus master to configure or control the slave device.

The AP9635 does not currently support this command.

A Preset Multiple Registers data-field request packet contains a definition of a range of registers to write to, and the values that are written to those registers. Up to 124 registers can be read. When writing data, you must begin writing data at the beginning of a data field and end it at the end of a data field. Not doing so will result in the receipt of an error response.

The slave device responds with a packet indicating that a write was performed to the range of registers specified in the request. The Preset Multiple Registers request and response packet formats are shown in the following example transaction.

Table 5–Preset multiple registers

Read Registers Request Packet (Master to Slave)	Read Registers Response Packet (Slave to Master)
Unit ID/Slave Address (1 byte)	Unit ID/Slave Address (1 byte)
16 (Function code) (1byte)	16 (Function code) (1 byte)
Start Register (sr) (2 bytes)	Start Register (sr) (2 bytes)
# of Registers to Write (nr) (2 bytes)	# of Registers Written (nr) (2 bytes)
Byte Count (2 x nr) (1 byte)	CRC Checksum (2 bytes)
First Register in range (2 bytes)	
Second Register in range (2 bytes)	
…	
CRC Checksum (2 bytes)	

Note: Except for the function field, the Preset Registers Response packet is identical in format to the Read Registers Request packet.

Example:

A device is configured as a Modbus slave device with slave address 200. The Master desires to write 1200 to a 32-bit register at address 46001/2 and 120 to a 32-bit register at address 46003/4 . Register 46001 is numbered 6000. The request must write 4 registers starting at 6000.

Slave address: 200 = C8(hex) Start register 6000 = 1770 (hex)

Value 1: 1200 = 0000 | 04B0 (hex) Value 2: 120 = 0000 | 0078 (hex)

Table 6– Request Packet (white background denotes the DATA field of the packet)

Slave	Function	Start Register (46001)		# of Registers (4)		Byte Count	Register 1		Register 2		Register 3		Register 4		CRC Checksum	
C8	10	17	70	00	04	08	00	00	04	B0	00	00	00	78	8B	F8

Table 7– Response packet

Slave	Function	Start Register (46001)		# of Registers (4)		CRC Checksum	
C8	10	17	70	00	04	D4	3C

Note: The values shown in the packets illustrated above are in hexadecimal format.

Invalid Registers

In the device Modbus register map, there are gaps between some registers. Invalid registers store no information. When an invalid register is read, the data field is FFFF(hex). When an invalid register is written, the data field is not stored. The device does not reject the request.

Modbus Data Format

This section covers the data format for the flexible Modbus Slave Module registers and the fixed Modbus registers.

Table 8– Data formats

Format	Datatype	Range
UINT16	16 bit unsigned integer	0 to 65,535
INT16	16 bit signed integer	-32,768 to +32,767
UINT32	32 bit unsigned integer	0 to 4,294,967,295
INT32	32 bit signed integer	-2,147,483,648 to +2,147,483,647
BOOLEAN	Boolean	
ENUM	16 bit unsigned integer corresponding to a fixed list of pre-defined meanings	0 to 65,535
FLOAT	32-bit single precision per IEEE St. 754-1985	
ASCII	Printable ASCII subset from 0x20 - 0x7E	

Note: The available formats vary depending on your device type and firmware.

16-bit Integer Format

Unsigned and Signed 16-bit Integer Formats are the simplest formats. Each register corresponds to one 16-bit value. If the format is unsigned, the value range for the output registers is 0 to 65535. If the format is signed, the value range is -32767 to +32767.

32-bit Integer Format

In Signed and Unsigned 32-bit Integer Formats, each value corresponds to two 16-bit Modbus Holding Registers. A 32-bit register represented in 32-bit integer format is passed via communications as two 16-bit registers:

- High-Order Register
  - o $register_{high}$=value/65536

- Low-Order Register
  - o $register_{low}$= value **modulus** 65536
  - o value = $register_{high}$ x 65536 + $register_{low}$ or

value = $register_{high}$ | $register_{low}$

Example (Unsigned 32-bit):

Value 12345678 is passed in unsigned 32-bit integer format:

- 12345678 = 00BC614E Hex
- $Register_{high}$ = 00BC Hex (unsigned) = 188
- $Register_{low}$ = 614E Hex (unsigned) = 24910
- Value = 188 x 65536 + 24910 = 12345678

In Unsigned 32-bit Integer Format, both the High-Order and Low-Order registers are unsigned 16-bit integers.

Example (Signed 32-bit):

Value -12345678 is passed in signed 32-bit integer format:

- -12345678 = FF439EB2 Hex
- $Register_{high}$ = FF43 Hex (signed) = -189
- $Register_{low}$ = 9EB2 Hex (unsigned) = 40626
- value = -189 x 65536 + 40626 = -12345678

In Signed 32-bit Integer Format, the High-Order register is a signed 16-bit number, but the Low-Order register is unsigned.

String Format

Strings are two characters per register, first character in high-order byte, second character in low-order byte. Printable ASCII only. For ASCII strings less than the maximum length, the unused characters are filled with nulls (zeros).

Example (8-character string):

- Word n (upper byte): S
- Word n (lower byte): t
- Word n+1 (upper): r
- Word n+1 (lower): i
- Word n+2 (upper): n
- Word n+2 (lower): g
- Word n+3 (upper): 0x0
- Word n+3 (lower): 0x0

Data is limited to printable characters, hex 0x20 – 0x7E

ASCII strings should be written using the same format shown for reading string data. When writing, there must be a null termination. A proper terminator is a byte-wide 0x0 value placed in the string. Any data written to the register after the terminator is ignored.

Decimal Format

Reading Decimal Data

Decimal formatted data is returned as a signed/unsigned binary value. The data is signed where the data range makes the sign necessary. A negative value is returned in the two's complement binary format.

Writing Decimal Data

Write decimal data as described in the "read" section.

Fixed-Point "S" Notation

Some data is formatted in a fixed-point format. Fixed-point format is a method for representing floating point numbers on integer based microprocessors. Fixed-point notation is based on the binary number system (base-2). The notation is written as S(X) where X defines the scaling factor and represents a power of two.

Reading Fixed-Point Data

When reading fixed-point data that is scaled S(X) you will need to convert the registers contents to a floating point number. To do this, divide the value by 2X.

Floating point value = Register Contents / 2X

Writing Fixed-Point Data

When writing fixed-point data that is scaled S(X), you will need to convert the desired floating point value to it's fixed point equivalent. To do this multiply the floating-point number by 2X, add 0.5, then truncate the fractional part of the result.

Register Contents = INT(Floating Point value * 2X + 0.5)

Device Modbus Registers

The device Modbus register map defines a set of parameters which are treated as HOLDING REGISTERS, having addresses 4xxxx. According to the Modbus protocol, in response to a request for register 4xxxx of a particular slave device, the Modbus master reads register xxxx-1 from the slave. For example, register 40011 corresponds to holding register 10.

Broadcast Packets

The Modbus protocol supports broadcast request packets. The purpose of a broadcast request packet is to allow all Slave devices to receive the same command from the Master.

A broadcast request packet is the same as a normal request packet, except the slave address field is set to zero (0). All Modbus slave devices receive and execute a broadcast request command, but no device will respond. The Preset Multiple Registers command is the only command supporting broadcast packets for Slaves.

Exception Responses

If a Modbus master device sends an invalid command to a device or attempts to read an invalid holding register, an exception response is generated. The exception response follows the standard packet format. The high order bit of the function code in an exception response is set to 1.

The data field of an exception response contains the exception error code. The table below describes the exception codes supported by the device and the possible causes.

Table 8–Exception codes supported

Code	Name	Meaning
01	Illegal Function	An invalid command is contained in the function field of the request packet. The device only supports Modbus functions 3 and 16.
02	Illegal Address	An Illegal Address error will be returned under the following conditions: 1. A read begins at an address other than the start address of a data variable. 2. A read begins outside the range of valid data variables. 3. A write begins at an address other than the start address of a data variable. 4. A write begins outside the range of valid data variables.
03	Illegal Value	The value referenced in the data field is not allowed for the referenced register on the device.
08	Write Error	A Write Error will be returned if the data is out of range for the register.
09	Field Overlap	A Field Overlap is returned when a write operation stops before the end of a data variable's address range. (Ex: Writing 5 words to a 10-word variable is not allowed.)

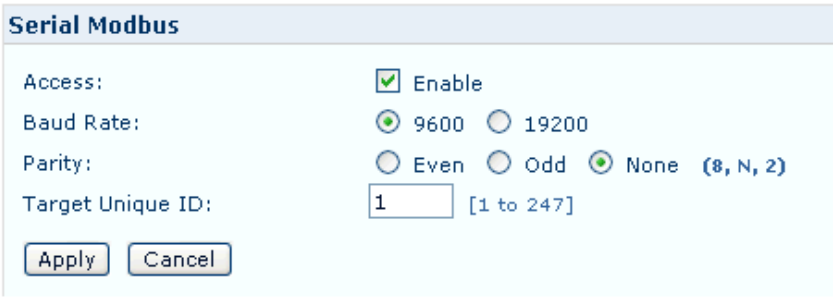
Troubleshooting

Troubleshooting Modbus can be very difficult made more complicated by the many possibilities of serial configurations, RS-232 to RS-485 converters, software, register numbers, etc. Even determining whether the issue is hardware/wiring or software is not easy. This section will attempt to offer suggestions.

This procedure assumes that an RS-232 to RS-485 converter is being used.

1. Double check that all devices have the same baud rate, number of data bits, parity, and number of stop bits. Double check that each device has a different address configured. Typically the RS-232 to RS-485 converter wants echo turned off, if there is a switch or jumper for that. The converter may also need to be configured for the baud rate being used. The software should be running in Modbus/RTU mode, not Modbus/ASCII mode. Try reading only single registers (vs. multiple register reads).
2. For troubleshooting purposes: try reducing the system to one device connected to the converter; get that working, then add other devices one at a time. Also, if possible, reduce the system to a 2-wire bus, this eliminates the possibility that Rx and Tx are swapped. Once you can talk to a single device, you can begin migrating back to your desired system in small steps.
3. The data line labeling can be confusing, double check the wiring:
   TxD– = A = TxD0 = TDA
   TxD+ = B = TxD1 = TDB
   RxD– = A' = RxD0 = RDA
   RxD+ = B' = RxD1 = RDB
4. For troubleshooting purposes: Remove all termination and bias resistors. The AP9635 does not require biasing, and most modern converters do not either. Some converters may have an internal termination resistor and double-terminating could lower the signal swing too much. Port powered converters may do better without any terminators. Some converters only drive the bus low and depend on a passive pull-up; these have about half the voltage swing of converters that drive the bus both high and low.

5. For troubleshooting purposes: set the AP9635 as below:



Set your software to 9600,8,N,2 (notice the 2 stop bits!) and poll device #1, attempting to read a single register continuously from register 40001 (Modcon format) or absolute address zero. In AP9635 this is a status register.

6. Your software should have a way to view the communications traffic. You should see:
   Transmit: 01 03 00 00 00 01 84 0A
   Receive: 01 03 02 00 08 B9 82
   The transmit data must be exactly as shown; if it is not, there is some software setting that needs to be changed, based on the following:

Byte #	Value	Description
1	01	Slave – the number we gave the AP9635
2	03	Function code:
3,4	00,00	Start register #, same as "40001" or "0001" if register # are set in software to be "base 1" or "PLC Addresses"
5,6	00,01	Number of registers to read
7,8	84,0A	CRC Checksum – this depends on the other data and will change automatically if anything else changes, you can not change this independently.

   The received data could be a little different for the last 4 bytes, because the device status could be different. This shows a status of 0x0008. The first 3 bytes must be 01 03 02

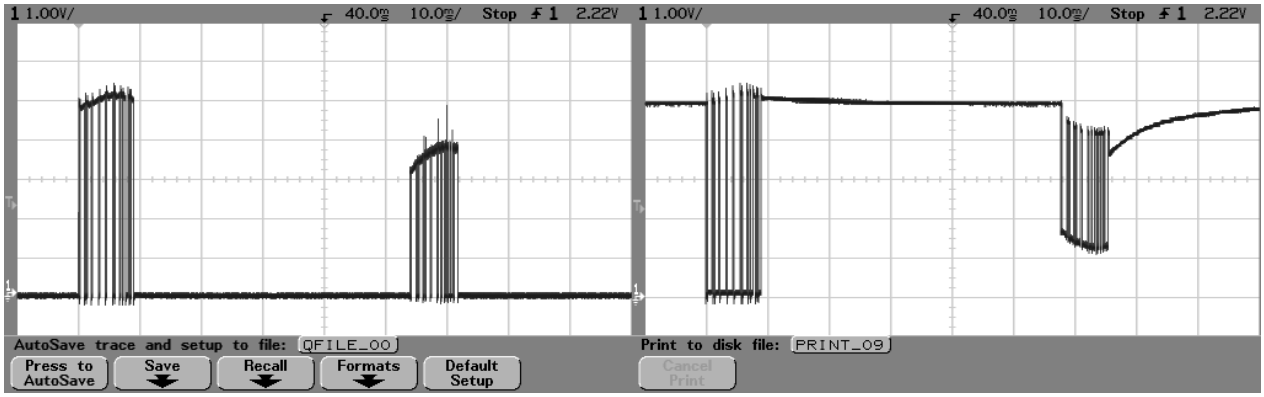
   If it looks like below (the key being that the received data is the same as the transmitted data), your converter box has echo turned on and your software does not have "Remove Echo" or similar enabled. Change the jumper on your converter box to turn echo off.

   Transmit: 01 03 00 00 00 01 84 0A
   Receive: 01 03 00 00 00 01 84
   (The error might show as "CRC Error")

   If there is no received data:

7. Be certain that common (ground) is connected from the device to the converter. The AP9635 Modbus interface is opto-isolated, other ground connections do not provide a ground for the Modbus interface.
8. Some RS-232 converters use null modem, some use straight-through cable. If the converter has a female DB9, you probably need a straight through; if it's male, you need a null modem.
9. Port powered converters generally get their power from RS-232 handshake lines, such as RTS (pin 7) and/or DTR (pin 4), if your cable doesn't pass those through, that could be a problem. If you can, connect the converter directly to the RS-232 port without a cable, or try powering the converter with an external supply if that is an option.

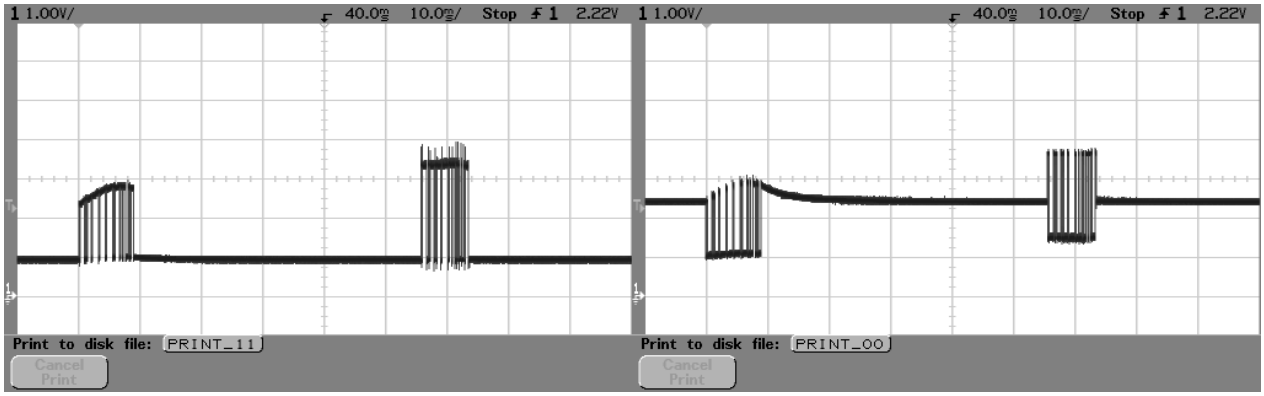
10. Some converters are baud-rate dependent (especially opto-isolated ones); check if jumpers need to be set inside the converter to agree with the baud-rate you are using. Some use RTS to control the RS-485 driver. If you have a separate opto-isolator in-line in your system, it probably has baud-rate jumpers that need to be set correctly.
11. Wiring: "Tx–" = "A" = "D0"; "Tx+ = "B" = "D1". Try swapping the + and – data wires at one end of the 2-wire cable. Still tie Tx+ to Rx+ and Tx– to Rx–; don't swap those. If that doesn't work, put them back the way you think they should be. Measuring with a volt meter, Tx+/Rx+ should read more positive than Tx–/Rx–.
12. At this point an oscilloscope is the best tool to figure out what's going on. Connect the scope, from common (ground) to D0, DC coupled, in auto trigger mode (continuous running), set the trigger level just above the steady state voltage. Set to 100uS/division, change triggering mode to "normal", try sending data and see if you can see activity. If not, go back and focus on the RS-232 and power connections.
13. Note the high and low voltages; the difference should be >1.5V. Common is a low of very close to zero and high of >3.2V (without termination or bias). Connect the scope to D1 and repeat the previous step. The high and low voltages should be almost the same as they were for D0. In practice, if your signal isn't swinging down to near zero and up to >3V, either there's something wrong, or you should consider a better converter box.
14. Baud-rate: notice the narrowest bit in the data capture, 50uS is 19.2k baud; 100 uS is 9600 baud. We set the AP9635 to 9600 baud, if you see 50uS bit times, your host software isn't set the way you think it is.
15. Slow the scope to 10mS/div. A full request packet (8 bytes of 10 bit-times each) at 9600 baud is a little more than 8mS (19.2k is a little more than 4mS). In 2-wire mode, since the bus is half duplex, both request and response are on the same wire. Attempt to determine if only a request is present, or if a response is being returned; the response would be 7 bytes, a little shorter than the request.



Unterminated Tx–/A/D0/TDA (1V/div; 10mS/div) **Unterminated Tx+/B/D1/TDB**

These scope shots were taken with a B&B Model 485BAT3 converter, port powered, connected as described, to an AP9635. Comment: One can see that the B&B converter is using 5V logic (signal swing peak is +5V) and the AP9635 uses 3.3V logic.

Our reference setup for debugging has no termination resistors. However for future reference here is what it looks like with termination:



Terminated Tx–/A/D0/TDA (1V/div; 10mS/div) **Terminated Tx+/B/D1/TDB**

16. If you see a response on the RS-485 wire but your software is not seeing any response, double check that you have Tx+ jumpered to Rx+ and Tx– jumpered to Rx– at both ends of the wire.
    Once this reference configuration is working, migrate back to your desired configuration in small steps, checking that the system is working after each small change.

17. If data is being received but there are problems with formatting/interpretation of the data or the values of the data, check the following notes which are included at the top of the APC Modbus map document. All the same issues apply to other vendor's devices, but may differ in some details.
    a. 16-bit registers are transmitted MSB first (i.e. big-endian).
    b. INT32 and UINT32 are most-significant word in n+0, least significant word in n+1 (i.e. big-endian).
    c. Function codes 3 and 4 are supported
    d. Modbus serial RTU and Modbus over TCP is supported.
    e. Signed numbers are twos-compliment
    f. Status bits are atomic within a single Modbus register. User should not look for consistency across multiple registers, only within a single register.
    g. For ASCII strings less than the maximum length, the unused characters are filled with nulls.
    h. Single-register reads of reserved or undefined registers will return an error. Block reads which begin with a valid register will not return an error but will return zeros for undefined registers.
    i. Strings are two characters per register, first character in high-order byte, second character in low-order byte. Printable ASCII only.
    j. Bit #0 is least significant bit.

k. Data Type column: "INT16"=signed 16-bit integer, "UINT16" = unsigned 16-bit integer, "INT32" = signed 32-bit integer, "UINT32" = unsigned 32-bit integer, "ENUM" is a UINT16 value which maps to a defined list of states, "ASCII" = the printable ASCII subset from 0x20 - 0x7E. BOOLEAN= a single bit, 0 or 1.

l. "Absolute Starting Register Address" = 0 (the column heading used in this table) is equivalent to "Register 40001" in Modicon terminology, which is address zero when transmitted over the wire.

18. Using the raw communications view, check the register address numbers in the transmitted and received packets. See step 6 above and the whole “Packet Communications” section above.

About the Author

Gary Ware is Director of Engineering for Shared Platforms in APC. He is responsible for technology platforms that are used across lines of business such as network and field bus communication interfaces, LCD displays, and protocols. Gary has been with APC since 1995.