SIMPLE MODBUS RTU MASTER

This document provides some hint to my Modbus RTU master code that I use instead of the Modbus commands built in to TBASIC. This document will make extensive reference to the PLC program, “ModbusMaster V2.PC6”.

The TBASIC statements ReadModbus, ReadMB2, WriteModbus, and WriteMB2 allow read and write access to one or more 16-bit registers. The function Status(2) must be called to determine if the success/failure of the TBASIC statements. Additionally the SetSystem 6,n statement is used to modify the Modbus Function code that are used with the TBASIC Modbus statements.

The TBASIC support for Modbus RTU has several limitations:

1. The TBASIC Modbus statements only use 4 of the 126 possible Modbus function codes. Only 16-bit data register access is supported by TBASIC. There is no support for bit data access to the slave device. There is no support for device specific Modbus function codes.
2. The TBASIC Modbus statements are blocking. This means that when you execute one of the statements the PLC ladder logic is suspended until the Modbus statement completes. If the slave device does not respond to the Modbus request, the PLC waits approximately 3 seconds before giving up. If your custom function uses 10 TBASIC Modbus functions and the slave device does not respond, then the PLC will be suspended for 30 seconds. This sort of behavior is unacceptable to me and my clients.
3. When a TBASIC Modbus statement is executed the only status information that you can get comes from execution of the Status(2) function., the only information that you can get is 0 and 1 that indicates if the Modbus transaction failed or succeeded. If the Modbus transaction fails you have no information to use to determine the nature of the failure.

MODBUS RTU RESOURCES

Modbus is 40+ years old. It was designed before the “0” was invented. There are some excellent resources available on the internet to help you understand Modbus. I have found the following to be indispensable:

“Modbus for Field Technicians” is an article written by Peter Chipkin that provides the best tutorial on Modbus that I have found. You can find it at this link:

http://chipkin.com/files/liz/MODBUS_2010Nov12.pdf

CAS Modbus Scanner is a utility program that runs on a PC that I use to characterize Modbus devices. Don’t even try and write PLC code for a Modbus device until you get it to work with CAS Modbus Scanner.

It works with Modbus RTU devices via a serial cable or Modbus TCP/IP via Ethernet. You can download it free from Chipkin Automation Systems:

https://store.chipkin.com/products/tools/cas-modbus-scanner

“Modbus Application Protocol V1.1b3”. This is the official documentation for Modbus and is available from modbus.org at this location:

http://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

MY APPROACH TO FIXING TBASIC'S SUPPORT FOR MODBUS

The issue with only supporting a small subset of Modbus function codes is solved by allowing you to build your own Modbus request packet. You can use any Modbus function code that you desire.

The problem with the TBASIC Modbus statements being blocking is solved by breaking the Modbus transaction into 3 parts:

- Build and transmit a Modbus request packet over the serial link.
- Wait for a response.
- Process the response

The building of the Modbus request packet is handled by a user-written custom function that fills in the details that are unique to the specific request. This code executes very quickly and once the Modbus request packet has been sent the TIMER, MBDelay, is started and the PLC ladder logic scanner continues running.

The ladder logic scanner continues running until the TIMER, MBDelay, goes active. This event invokes a custom function to handle the response to the Modbus request.

The use of the TIMER, MBDelay, ensures that the PLC does not lockup waiting for the response to arrive from the slave device. This makes my version of Modubs RTU non-blocking.

The problem with debugging Modbus queries and responses. The complete Modbus query and its response is visible in DM[] when using On-Line Monitoring. Nothing is visible when using the TBASIC Modbus statements. My code maintains a status variable, MBStatus, for the Modbus transaction that provides a lot more information then pass/fail.

DETAILS OF THIS MODBUS RTU IMPLEMENTATION

I make extensive use of the TBASIC #Define macros. I use this mechanism to help make my TBASIC code a bit easier to maintain and to minimize the opportunities to write buggy code.

If you click on the "#Define" menu button in the custom function editor, the "Define Variable Names" window will open. This is the first chunk of the define tables and it describes how and where the Modbus request is built in 16-bit DM[] memory:

#	Label Name	Variable	
1	__Modbus__	Definitions	
2	ModbusPort	2	<-- PLC Serial port #
3	SlaveAddr	2	<-- Modbus Slave Address
4	__Query__		
5	MBQuery	801	<-- Modbus Request buffer starts at DM[801]
6	MBQuerySize	30	<-- Size of buffer in characters
7	MBQSlaveAddr	DM[801]	<--Modbus Slave Address goes here
8	MBQFunctionCode	DM[802]	<-- Modbus Function code
9	MBQStartAddrMSB	DM[803]	<-- Modbus Starting Address Argument (16-bit)
10	MBQStartAddrLSB	DM[804]	
11	MBQArg1MSB	DM[805]	<-- Modbus argument to specify how many
12	MBQArg1LSB	DM[806]	
13	MBQByteCount	DM[807]	<-- Number of data bytes to follow *
14	MBQData	808	<-- Start of data for variable length requests *
15	MBQDataEnd	830	<-- End of buffer space for Modbus request

"*" The exact layout of the Modbus request packet is determined by the Modbus function code.

There is a 2nd, independent buffer setup in DM[] for the response packet that the slave Modbus device is expected to send. The #Defines for the response buffer look like this:

16	__Response__		
17	MBResponse	831	<-- Start of Modbus response buffer, DM[831]
18	MBResponseSize	30	<-- Size of buffer in characters
19	MBRSlaveAddr	DM[831]	<--Modbus Slave Address is echoed here
20	MBRFunctionCode	DM[832]	<-- Modbus Function code is echoed here **
21	MBRByteCount	DM[833]	<-- Number of data bytes to follow **
22	MBRData	834	<-- Start of data for variable length responses *
23	MBRDataEnd	860	<-- End of buffer space for Modbus response

"*" The exact layout of the Modbus request packet is determined by the Modbus function code.

"**" The slave may set the most significant bit of the function code to a "1". This is used by the slave to indicate that the response is an error message. If this is an error response then the byte that follows the function code is interpreted as the error value.

When a Modbus request/response sequence completes, the PLC code updates the variable MBStatus with the final status . The #Define layout for the status value is as follows:

24	MBStatus	DM[861]	<-- Modbus status variable stored here
25	__ModbusDefs__	MB Constants	
26	EndOfMessage	&h7fff	<-- end of buffer marker
27	MBSuccess	0	<-- possible Modbus status values
28	MBFCErr	1	
29	MBArgErr	2	
30	MBBigRspErr	3	
31	MBCRCErr	4	
32	MBSlaveIDErr	5	
33	MBFCodeErr	6	
34	MBTimeout	7	
35	MBErrRsp	8	

The "MBSuccess" value of 0 indicates that the Modbus request was sent, the slave device responded and everything when just as planned.

The "MBErrRsp" indicates that the slave device received the request but objected to something in the request and has returned an error response packet to give you some clue as to what was unacceptable. This is a sign of a well-designed Modbus slave. This response can help you correct your PLC coding.

The "MBTimeout" indicates that no response was received. This could be a communication issue, a cabling issue, a non-functional slave device, a slave device that has not been configured for the proper Modbus ID or the result of space aliens. In any case, this is a hint of a problem that you need to solve.

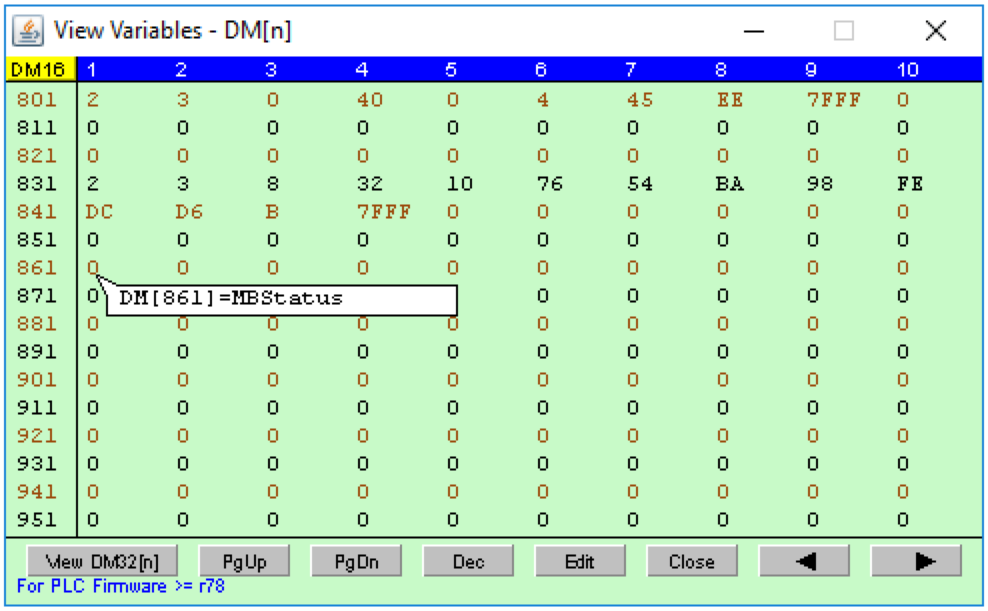
The other responses are going to be very rare. Search through the CFs to see how they are defined.

The last bit of the #Define tables are the Modbus function codes. Yes I could have just used the constant value of "3" instead of using "MBReadHoldingRegs". But, I find that if I have to look at this code a year later, I can never guess what a "3" means.

36	__MBFunctionCodes__	MB Defines	
37	MBReadCoils	&h01	<-- Modbus function code values
38	MBReadHoldingRegs	&h03	
39	MBReadInputRegs	&h04	
40	MBWriteSingleCoil	&h05	
41	MBWriteSingleReg	&h06	
42	MBWriteMultiCoils	&h0f	
43	MBWriteMultiRegs	&h10	

On-line Monitoring

My Modbus code was designed to help debug both PLC code and to characterize Modbus slave device behavior. This is screen shot shows the the Modbus request, Modbus response and the final status for a successful Modbus transaction:



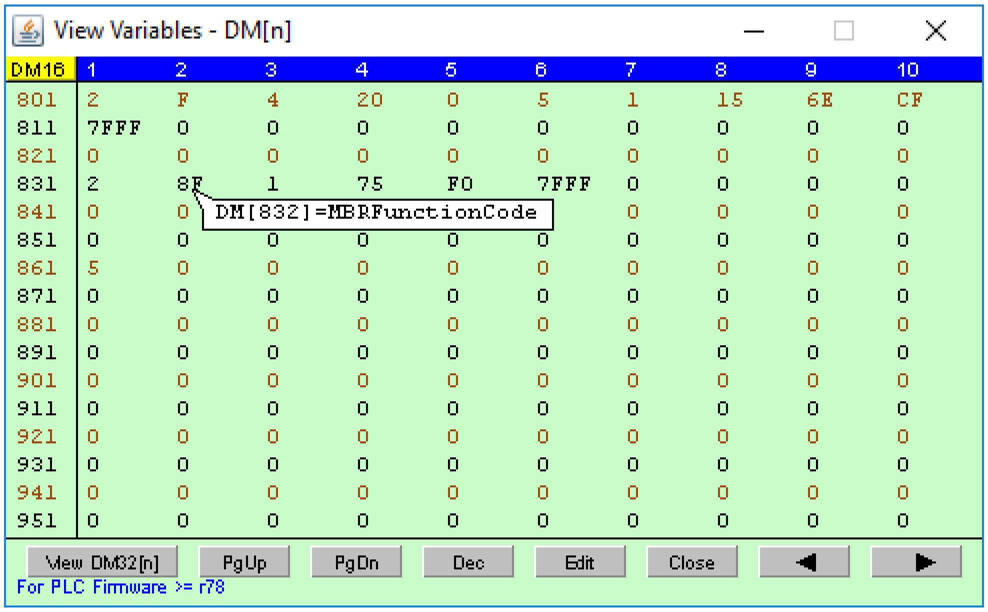
DM16	1	2	3	4	5	6	7	8	9	10
801	2	3	0	40	0	4	45	EE	7FFF	0
811	0	0	0	0	0	0	0	0	0	0
821	0	0	0	0	0	0	0	0	0	0
831	2	3	8	32	10	76	54	BA	98	FE
841	DC	D6	B	7FFF	0	0	0	0	0	0
851	0	0	0	0	0	0	0	0	0	0
861	0	0	0	0	0	0	0	0	0	0
871	0	DM[861]=MBStatus				0	0	0	0	0
881	0	0	0	0	0	0	0	0	0	0
891	0	0	0	0	0	0	0	0	0	0
901	0	0	0	0	0	0	0	0	0	0
911	0	0	0	0	0	0	0	0	0	0
921	0	0	0	0	0	0	0	0	0	0
931	0	0	0	0	0	0	0	0	0	0
941	0	0	0	0	0	0	0	0	0	0
951	0	0	0	0	0	0	0	0	0	0

The Modbus request packet starts at DM[801] the value of &h7fff at DM[809] marks the end of the Modbus request. The data in DM[807] and DM[808] represents the 16-bit CRC for the request packet. The &H7fff marks the end of the request packet.

Note that when you move the mouse cursor over the View Variables - DM[n] window that those locations that are referenced in the #Define table will be annotated.

The Modbus response packet starts at DM[831] at the &h7fff at DM[844] marks the end of the response. Note the value at DM[861] this is the status for the overall transaction. The value of 8 in DM[833] is the byte count for the data returned from the slave. These 8 bytes represent four 16-bit register values returned by the slave device. The data at DM[842] and DM[843] is the CRC for the response packet. The &H7fff marks the end of the response packet.

OK. Now I am going to show you what a Modbus response looks like when the Slave device objects to the format of the request:



DM16	1	2	3	4	5	6	7	8	9	10
801	2	F	4	20	0	5	1	15	6E	CF
811	7FFF	0	0	0	0	0	0	0	0	0
821	0	0	0	0	0	0	0	0	0	0
831	2	8F	1	75	F0	7FFF	0	0	0	0
841	0	0	DM[832]=MBRFunctionCode				0	0	0	0
851	0	0	0	0	0	0	0	0	0	0
861	5	0	0	0	0	0	0	0	0	0
871	0	0	0	0	0	0	0	0	0	0
881	0	0	0	0	0	0	0	0	0	0
891	0	0	0	0	0	0	0	0	0	0
901	0	0	0	0	0	0	0	0	0	0
911	0	0	0	0	0	0	0	0	0	0
921	0	0	0	0	0	0	0	0	0	0
931	0	0	0	0	0	0	0	0	0	0
941	0	0	0	0	0	0	0	0	0	0
951	0	0	0	0	0	0	0	0	0	0

Notice that the Function code in the response packet at DM[832] does not match the Function code in the request packet at DM[803]. The most significant bit of the 8-bit function code has been set to a "1" in the response. This indicates that the response is an error response. The "1" value at DM[833] indicates that the slave does not support the function code of &h0f, write multiple coils. This tells you something very useful about the slave device behavior.

Custom Functions

The custom functions are written as simply as I know how. I have tried to comment them in some useful manner. They are written to execute quickly.

InitModbus

This custom function is called on the first scan of the ladder logic to initialize the PLC hardware. This function sets up the serial port that will be sued to communicate with the slave device.

This function informs the PLC low-level firmware of my attempt to implement my own communication protocol. If you don't disable the low-level firmware, it will respond to the response from the slave as if it were a Modbus request. Two things will happen, the response from the slave will be extracted from the PLC receive buffer and be lost and the PLC will attempt then respond to the slave as if it was a master device. The result is a mess. I have seen this happen and I took me a while to sober up enough to figure out what was happening.

The HSTIMER statement modifies the behavior of TIMER #1, MBDelay. This configure MBDelay to work with time internals of 10ms instead of 100ms. This allows me to "tune" the SV of this TIMER to be close to the time for the Modbus request to be transmitted and for the slave to respond.

SendMBQuery

This is the generic handing of a Modbus request (query). This code handles generating the CRC for the request packet and enqueuing the request packet to be transmitted by the low level PLC firmware.

If you need to add support for a new Modbus function code, then you may have to edit this function. This issue is that SendMBQuery needs to "know" how many bytes are in the request so that it can do its work.

SendMBQuery is called from user-written custom functions, only. It is the responsibility of the user-written custom function to enter the slave address, function code and all arguments before calling this CF. Please refer to the user-written CF, RdHldRegs, as an example of what you need to do.

ProcessMBRsp

This is the custom function that provides the generic handling of the response (or lack of response) from the slave device. The CF extracts all of the data bytes in the serial port's receive buffer and copies them into DM[] memory. This custom function verifies that the response is valid by checking the CRC and then examining the first few bytes in the response packet. This CF updates the MBStatus variable and exits.

ProcessMBRsp is called from user written custom functions only. It is the responsibility of the user-written responsible to process the data returned by the slave. Please refer to the user-written CF, RDhldRegsPsp, as an example of what you need to do.

Test Environment for this Demo Code

This code will execute on any of the modern TRI PLCs. I have run it on Nano-10s, FMD1616-10s, and Fx series PLCs. Right this moment is running on a Fx1616-BA PLC.

To make this demo a bit more educational, the Modbus Slave that I am using is, also, a TRI PLC. At this moment I am using a 2nd Fx1616-BA as the slave device. The are interconnected with a single twisted pair for the RS-485 port and share a common DC ground.

This is a copy of the CF function that I am running on the Slave PLC and this code executes on the first scan of that PLC:

```
' Init - CF to initialize PLC for Modbus Data Test Patterns for slave
device
'
SetBAUD 2,6              ' RS-485 port 38.4K 8,1,N (Modbus interface)
SetProtocol 2, 1         ' this port should be Modbus RTU

' Fill DM[] with incrementing word patterns
'
'     DM[1] = 1
'     DM[2] = 2
'     DM[1023] = 1023 = &03FF
'     DM[1024] = 1024 = &0400
'
' If you look at the same memory as DM32[] this is what you will see:
'
'     DM32[1] = &H0010002 = 65,538
'     DM32[2] = &H0030004 = 19,612
'     DM32[511] = &H03FD03FE = 66,913,278
'     DM32[512] = &h03FF0400 = 67,044,352
'
for i = 1 to 1024
      DM[i] = i
next

' Test patterns for RELAYS
'
j = 0
for i = 1 to 16          ' for each RELAY[i]

      n = j+3     : n = n * 16
      n = n+j+2   : n = n * 16
      n = n+j+1   : n = n * 16
      n = n+j
      j = j + 4

      RELAY[i] = n
next
```

Notes on Zero Based Modbus Register Addressing

If you haven't read through the PLC code custom functions then you missed by rant on "zero-based" addressing and Modbus.I will repeat just enough of it so that you will learn something useful about Modbus and TRI PLCs.

Notes on Zero Based Modbus Register Addressing

Modbus was apparently invented before the "0" was discovered. The documentation for most Modbus devices, including the TRI PLCs publish the device registers address as being +1 greater than what will be sent in the Modbus request and response packets.

However, device documents describe the internal registers addresses as the exact values that need to be sent in the request/response packets. I prefer this approach. Just take the published register address and add "0" (nothing) to it and use it. Zero-based.

Remember I said that TRI documents their PLC's Modbus behaver using the +1 notation. If you have attempted to use their TBASIC ReadModbus, ReadMB2, WriteModbus or WriteMB2 statements, then you will know that these statements do not work with the +1 notation but require you to provide the zero-based register addresses! Not very consistent.

OK, now I am going to let you in on a little secret, the register address that TRI PLCs respond to as Modbus slave devices. First the PLC data that you can address as 16-bit registers:

16-bit Data Items	Starting Address	Solve for y Modbus address
INPUT[x]	0	$y = x - 1$
OUTPUT[x]	16	$y = x + 15$
TIMERBIT[x]	32	$y = x + 31$
COUNTERBIT[x]	48	$y = x + 47$
RELAY[x]	64	$y = x + 63$
TimerPV[x]	128	$y = x + 127$
CtrPV[x]	256	$y = x + 255$
TIME[x]	512	$y = x + 511$
DATE[x]	516	$y = x + 515$
DM[x]	1000	$y = x + 999$

OK, but there are 32-bit things that you can access via Modbus, DM32[x] and FP[x] on the PLCs that support floating point math. In all cases, you must request 2 sequential 16-bit registers for each 32-bit item. TRI PLCs sends/receives the most significant word (16-bits) of the 32-bit value first. The least significant word follows. Here's the rules for computing the Modbus address:

32-bit Data Items	Starting Address	Solve for y Modbus address
DM32[x]	1000	$y = 2x + 998$
FP[x]	1000	$y = 2x + 998$

OK. I am feeling very generous. Let me clue you in on what you can address as bit-oriented data in the TRI PLCs. I will just give you the math to figure out the starting address for Modbus. I won't lecture you on the details of multiple coil reads/writes as I consider this sort of knowledge to be too dangerous to share.

Bit Addressable Data Items	Starting Address	Solve for y Modbus address
INPUT x	0	$y = x - 1$
OUTPUT x	256	$y = x + 255$
TIMER x	512	$y = x + 511$
COUNTER x	768	$y = x + 767$
RELAY x	1024	$y = x + 1023$

There are some additional details about how the TRI PLCs respond to Modbus requests. Both the 16-bit registers and the bit items will respond to more that a single Modbus function code. As an example, you read DM[x] as either a Holding register or as an Input register. Most bit data items can be read as either a Coil or a discrete input. TRI documented these facts!

Gary Dickinson 6/11/2019

garysdickinson@me.com