

# MULTIdrive™ 2000

## USER GUIDE

---

For Windows 98/NT

NewAge Automation Inc.  
1700 Varsity Estates Drive NW.  
Calgary • Alberta • Canada • T3B 2W9  
Phone 403.247.4236 • Fax 403.547.7495  
[www.NewAgeAutomation.com](http://www.NewAgeAutomation.com)

printed 2000.01.03



# Table of Contents

## CHAPTER 1

### INTRODUCTION

Installation	3
Overview	4
Where to Go From Here	4

## CHAPTER 2

### Tutorial I - Your Modem as an Industrial Device

Creating a New Project	6
To Create a New Project	6
Adding Tags to Your OPC Server	6
To Create an OPC Item	7
Creating Your First Transaction	7
To create the required transaction:	8
Adding Transaction Triggers	9
To Add a Trigger to our Transaction	10
Initializing Your OPC Server	11
To Initialize Your OPC Server.	11
Testing Your OPC Server	11
To Test Your OPC Server	11
To Configure You Serial Analyzer.	12
Transaction Error Handling	14
Summary of Changes	15
Adding a Read/Write Item	16
To Add the Read/Write Item	16
To Add the Read Register S0 Transaction	16
To Assign Triggers to the Transaction	17
To Add the Write Register S0 Transaction	18
Summary	19

## CHAPTER 3

### TUTORIAL II - Modbus RTU

Prerequisites	21
Getting Started	22
Adding Tags to Your OPC Server	23
Modbus 101	28
Modbus Transactions	28
Modbus Commands	29
Read Input Status (1XXXX)	29
Read Output Status (0XXXX)	29
Read Input Registers (3XXXX)	30
Read Holding Registers (4XXXX)	30
Modbus CRC-16	30
Adding Transactions	32
Defining the Transaction Steps	32
Adding Transaction Triggers	36
Exercise I	36
Exercise II	36
Writing 4XXXX Registers	37

## CHAPTER 4

### SERIAL ANALYZER

Setting up the Serial Analyzer	40
--------------------------------	----

## CHAPTER 5

### REFERENCE

Transaction Steps	44
Transaction Triggers	53



## Introduction

### *OPC Support for Any Serial Device*

**M**ULTIdrive is a next-generation driver development kit that allows technical non-programmers to develop & deploy OPC compliant drivers for almost any serial device. Using the features & functionality of MULTIdrive™, almost any device can be controlled and communicated to regardless of protocol, speed or communications format. Once your MULTIdrive™ OPC Server has been developed, it can be deployed as a runtime where it will provide seamless, transparent device communications.

#### Installation

The system can be installed by running **SETUP.EXE** on **Disk 1**. The system will be installed immediately with a 30 day evaluation license. While it is important to activate the license as soon as possible, the evaluation version is fully compatible with the licensed version and so no effort will be lost developing your drivers immediately.

Depending upon the type of MULTIdrive™ licensed purchased, the licensing information may or may not be included with the package. If it is not, please contact NewAge Automation immediately and your licensing information will be provided via FAX.

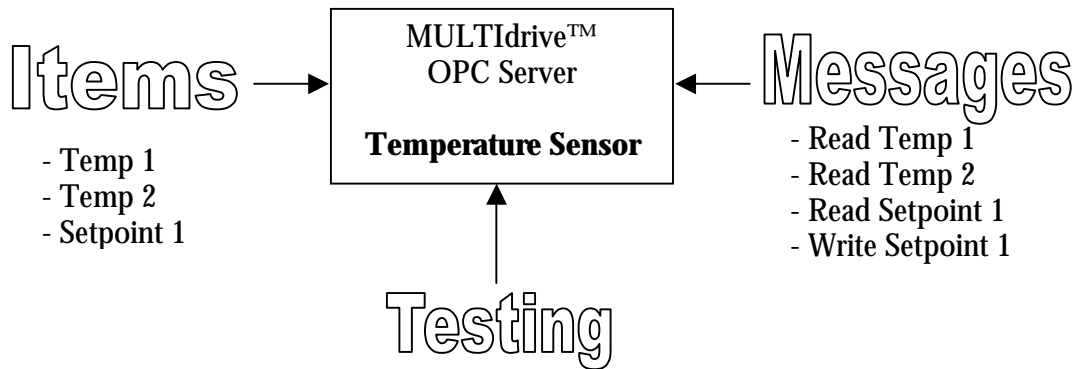
To activate your license, select the menu option **Help Authorization...** and enter the licensing information provided to you from NewAge Automation Inc.

The screenshot shows a standard Windows-style dialog box titled "Authorization". It features a blue title bar with a close button (X) on the right. The dialog contains several text input fields and buttons. The fields are: "Name" with the value "John Doe", "Company" with "ABC Manufacturing Co.", "City / State" with "Ottawa, Canada", "Serial Number" with "113756", and "Key Code" with "1022-3236". On the right side, there are three buttons: "OK", "Cancel", and "Change...". At the bottom right, the "Site Code" is displayed as "FF2E-7599".

## Overview

Developing a MULTIdrive™ OPC Server for your device consists of 3 basic steps.

1. Defining all the OPC items (or tags) your server will provide to OPC clients.
2. Defining each of the communication messages used to communicate with your device.
3. Using the Serial Analyzer & OPC Client functionality to test your server.



Prior to starting a new development project it is essential you have the following materials.

1. The actual device that will be interfaced to, including any necessary cables, equipment, etc. *With each communications message developed, it is essential to test the results to help guide further development.*
2. All the device's user documentation including a specification of the protocol used to communicate to the device. *The protocol specification will be an invaluable reference when re-implementing the protocol in MULTIdrive™.*
3. Any existing software or hardware tools used to configure, monitor or otherwise communicate to the device. *Existing utilities can be used to verify internal settings are properly being read or written by MULTIdrive™.*

## Where to Go From Here

Prior to starting your own device development project it is recommended you perform the tutorial found in Chapter 2. This tutorial will guide you through each of the steps required to define, develop and test your driver interface.

From here, you should be able to start your own project while referencing the reference material found in Chapter 3. The Glossary and Index located at the end of this User Guide will help define terminology or provide more detail about a particular aspect of MULTIdrive™.

## Tutorial I – Your Modem as an Industrial Device

*Quickly develop a MULTIdrive™ OPC Server using your modem as an industrial device.*

**T**he following tutorial exercises some of MULTIdrive's functionality by communicating over your serial port directly to your modem. Although you might not consider your modem an industrial device, it does share many similarities to an industrial device.

- Communicates over a standard Serial port.
- Communicates via an ASCII protocol
- Accepts commands & settings

For first time users this tutorial provides an excellent starting point prior to developing your own driver for your next industrial device.

If neither an internal or external modem is installed on your computer, you will not be able to complete this tutorial.

## Creating a New Project

Prior to developing any new device interface it is necessary to start a new MULTIdrive™ Definition File (MDF). In addition to creating a new file, it is recommended you immediately provide a filename via the **File Save As...** command.

To Create a New Project

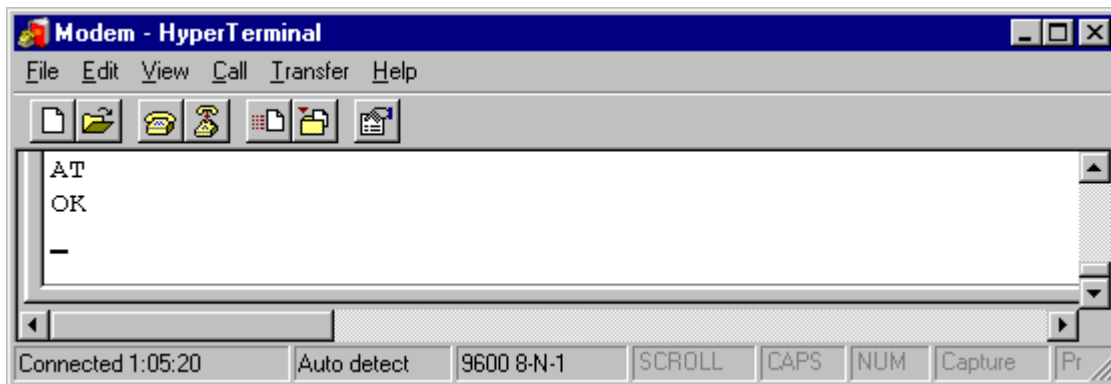
1. From the main menu select **File New**.
2. Save the file under a new filename via the **File Save As...** command. Provide a filename of MODEM.MDF.

## Adding Tags to Your OPC Server

Regardless of what type of industrial device your communicating with, it is necessary to define early on, all the information that will be read or written to the device. It is this information that will eventually define what types of messages will be required to communicate with the device.

A *Tag* refers to a single data point within an industrial sensor, controller or SCADA system. For example, if you were to read 3 temperatures from a temperature sensor, 3 tags would be needed to store the information in a database. MULTIdrive™ follows the terminology outlined by the OPC specification and refers to tags as *OPC Items* (or *Items*).



Regardless of the number of items defined, some of the first items to define are those concerned with device communications and status. It is always important to know your device is properly connected, communicating and healthy. With respect to our modem, by simply typing **AT**, the modem responds with 'OK'. You can try this using any standard terminal software connected to your modem.

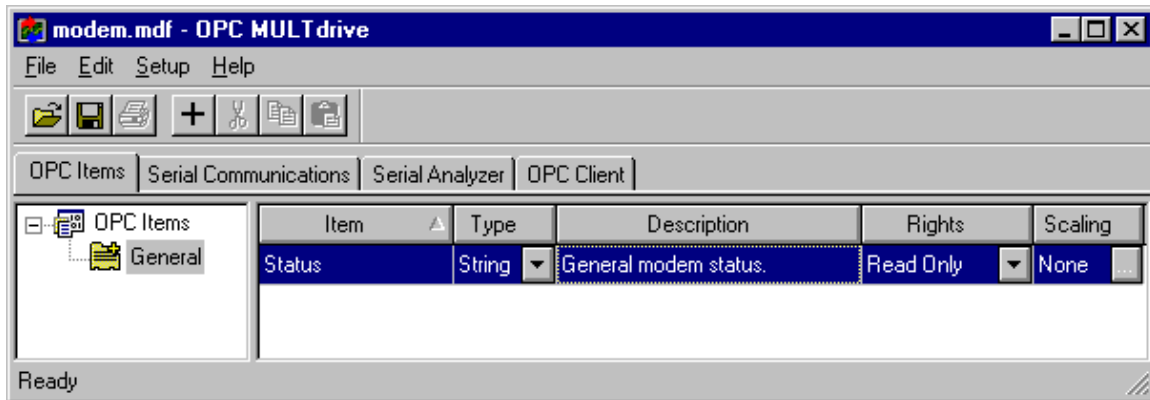




While on the surface 'OK' doesn't tell us much, it does in fact eliminate many possibilities for a user diagnosing a communications problem. For example, with a single reply of 'OK' we know; a modem is installed, it's turned on, the cable is connected, and our communication settings are OK.

Using HyperTerminal or any other terminal software, attempt to connect to your modem (e.g. COM2: 9600, N, 8, 1) and issue a simple **AT** command.

To Create an OPC Item

1. From within MULTIdrive, click  to add a new *OPC Group*. Rename the group to 'General'. You can change the name of a group by clicking on the group itself and renaming it.
2. With the group selected, click  again and add a new *OPC Item*. Rename the item to 'Status' and define it as follows...



 The  toolbar icon along with the menu option **Edit Add**, and the keystroke <Ctrl-A> are all used to add (or append) groups, items, transactions, steps or triggers. What is added depends upon what is currently selected.

## Creating Your First Transaction

*Transactions* within MULTIdrive™ are the key to communicating. Each transaction contains a list of *transaction steps* that define *how* to communicate, and a list of *transaction triggers* that define *when* communications should take place. Under normal circumstances, a transaction is responsible for sending a command and receiving a reply. More formally, a transaction contains the following steps...




- Formatting the transmission message.
- Transmitting the message
- Receiving the reply
- Extracting information from the reply.

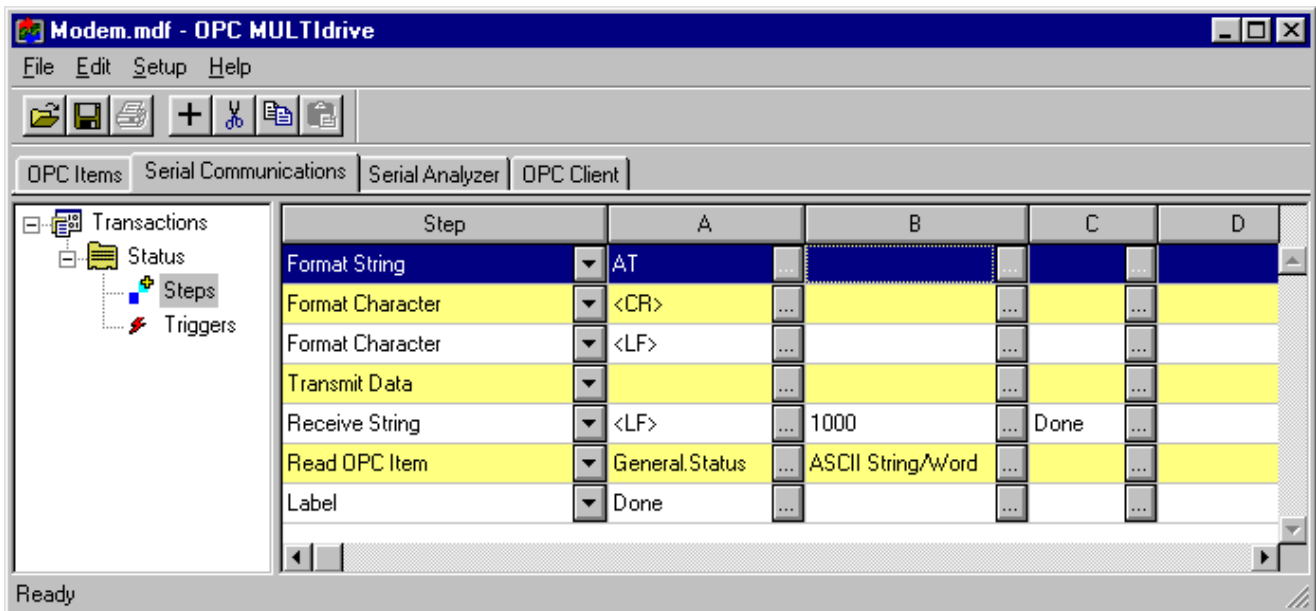
Each transaction stands alone as a complete entity, independent of other transactions. With respect to our new 'Status' item added above, we require a single transaction to retrieve the necessary information from our modem. The transaction will consist of these steps...

- Formatting of the transmit message "AT<CR><LF>"
- Transmission of the message.

- Reception of the reply message
- Setting the contents of the 'Status' item to that of the reply.

To create the required transaction:

1. Select the *Serial Communications* tab.
2. Click on the root *Transaction*.
3. Click  to add a new transaction, and rename it to 'Status'.
4. Expand the 'Status' transaction by clicking .
5. Add your first transaction step by selecting *Steps* and clicking .
6. Add and format the steps as seen below.



The step *Format String* places any user defined string on to the transmit buffer. The prefix *Format* is used to identify all functions used for formatting the transmit buffer. The string 'AT' is our ASCII command message that will retrieve the modem's status.

The step *Format Character* is used to place any ASCII character on to the transmit buffer. While *Format String* could perform the same function, this command has the added ability of allowing any ASCII character to be selected, including non-keyboard type characters such as carriage return <CR>, line feed <LF>, etc.

The step *Transmit Data* has no parameters and performs the function of transmitting all characters (or bytes) in the transmit buffer out the currently open serial port.

The step *Receive String* receives an entire line or string of data up to and including a user defined termination character. Most ASCII protocols will terminate their reply with a carriage return <CR>, linefeed <LF>, end of text <EOT> character, or something similar. In our case, data will be received until a line feed <LF> character is received (A). In the event the characters are not received, and/or a timeout occurs, a timeout setting in

milliseconds (B) and a jump label (C) are defined. The jump location in our case is simply the bottom of the step list.

The step *Read OPC Item* is used to read values out of the receive buffer and place into their respective OPC Item (A). There are many different types of extraction formats for both ASCII and Binary protocols. When selecting the format (B), first select ASCII, then select *ASCII String/Word*. This will extract a single English word from the receive buffer.



The prefix *Read.* in any command refers to a step that *reads* data from the receive buffer. The data is read from the buffer's current location. Once the data is read, the current location is incremented to the next byte/character not read yet. ***It is extremely important to understand this concept, and always be aware of the receive buffer's current location before reading data.***

## Adding Transaction Triggers

Each transaction is defined with a list of *Steps* and *Triggers*. The steps define *what* is to be done, the *triggers* define *when* it is to be done. If triggers are not defined for a given transaction, that transaction will never execute, and hence never transmit or receive any information.

MULTIdrive™ supports many different types of triggers that can be used individually or combined together to execute the transaction as required. The table below provides a quick summary of the different triggers supported.

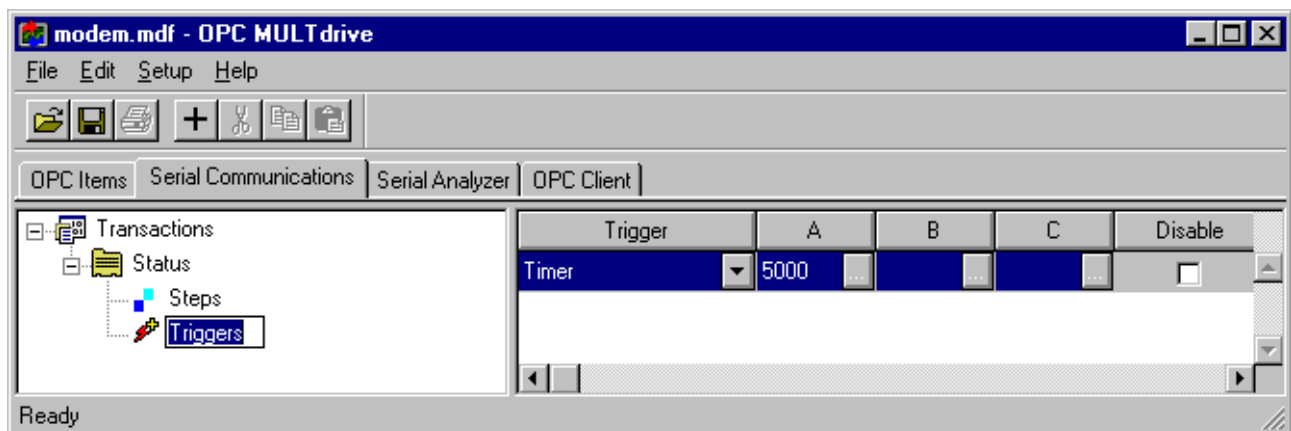
Break Signal	Execute the transaction whenever an RS232 break signal is received.
Carrier Detect	Execute the transaction when the RS232 CD signal goes high (or low).
Clear to Send	Execute the transaction when the RS232 CTS signal goes high (or low).
Data Set Ready	Execute the transaction when the RS232 DSR signal goes high (or low).
Error Detected	Execute the transaction when a checksum, overrun, or framing error is detected.
Event Character	Execute the transaction when a user defined 'event character' is received.
OPC Item Read	Execute the transaction whenever an attached OPC client request a specified item be read.
OPC Item Update	Execute the transaction whenever the MULTIdrive™ server performs a periodic update.

OPC Item Update	Execute the transaction whenever an attached OPC client requests a specified item be written.
Receive Character	Execute the transaction whenever any character is received.
Ring	Execute the transaction whenever the RS232 ring signal goes high (or low).
Shutdown	Execute the transaction whenever MULTIdrive shuts-down. This also includes each time a <b>File Save</b> operation is performed and the old runtime is shutdown.
Startup	Execute the transaction whenever MULTIdrive starts-up. This includes each time a <b>File Save</b> operation is performed and a new runtime is initialized.
Timer	Execute the transaction at a specified interval in milliseconds.
Tx Buffer Empty	Execute the transaction whenever the transmit buffer empties out.

With respect to our newly added transaction we would like to execute the transaction every 5 seconds.

To Add a Trigger to our Transaction


1. Expand the transaction as before and select *Triggers*.
2. Click **+** to add the first trigger.
3. Change the trigger type to *Timer* and enter a 5000 milliseconds (5 sec) delay for parameter (A).
4. Your new trigger should look like this...

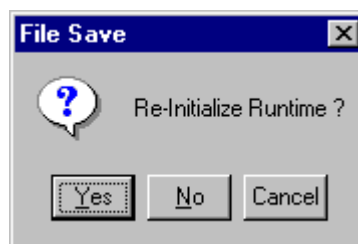


## Initializing Your OPC Server

The MULTIdrive™ OPC Server is initialized and updated each time the **File Save** operation is performed. With each save operation, the user has the option of updating the server or not updating the server and simply saving the file.

To Initialize Your OPC Server.

1. Select **File Save** or click  to save your definition file.
2. If there are any syntax errors detected, fix the errors and return to step 1. Otherwise, continue to step 3.
3. At the prompt below, select yes to initialize your runtime system (or OPC Server).



Once the save / initialize operation is performed, any connecting OPC Clients will now view the new groups and items created.

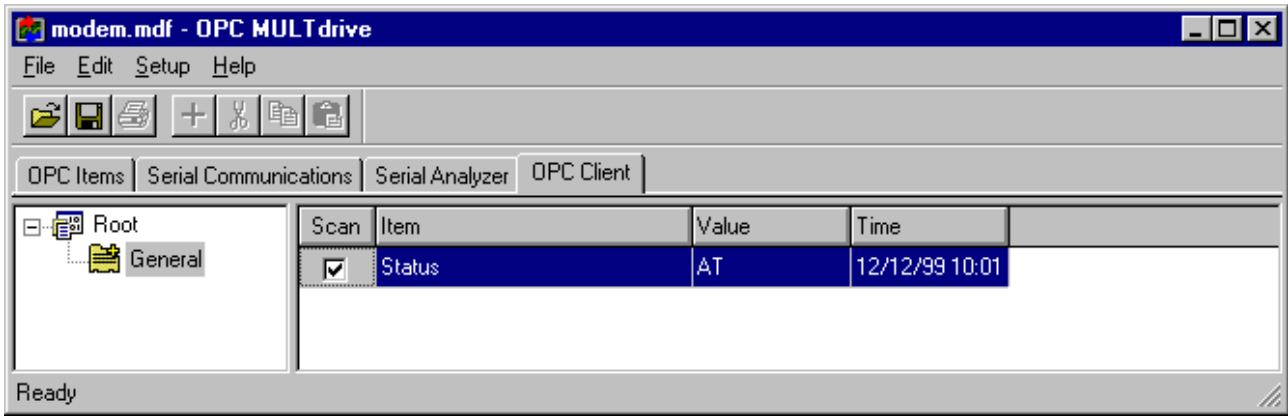
If clients are connected prior to re-initializing the server, these clients will be disconnected before the re-initialization occurs.

## Testing Your OPC Server

It is always important to test your OPC Server for proper functionality. This can be done using MULTIdrive's built-in *OPC Client* in conjunction with the *Serial Analyzer*. The OPC client functionality allows you to connect to your OPC server in the exact same manner as an external client. The serial analyzer allows you to monitor communications and verify the device is being properly communicated to.

To Test Your OPC Server

1. Select the *OPC Client* tab.
2. Select the OPC Group 'General' and enable the scanning of the 'Status' item by checking the scan checkbox.
3. The following display should result.



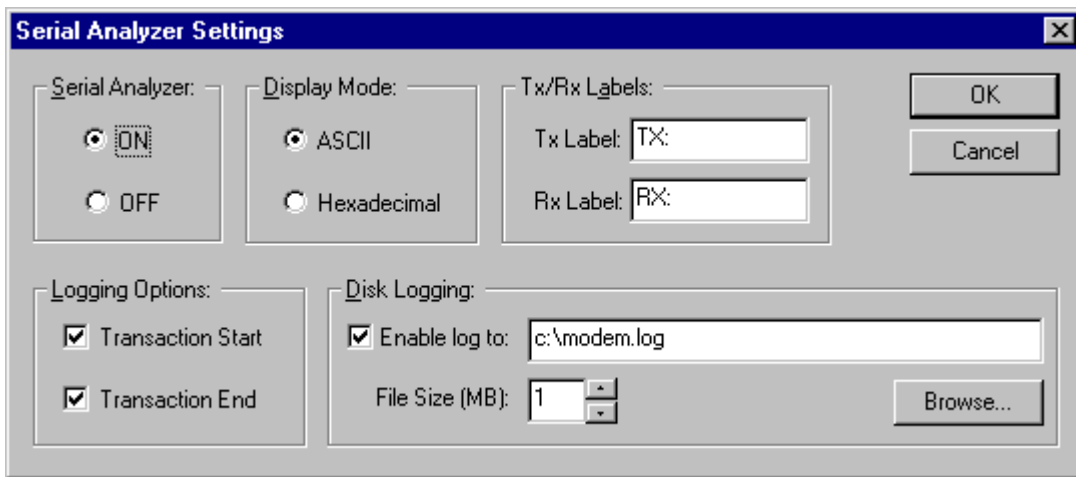
The connected client immediately tries to update the ‘Status’ tag, whose result for some reason toggles between ‘AT’ and ‘OK’. *Note, this is normal behavior for this point in the tutorial.*

MULTIdrive’s *Serial Analyzer* can be used to debug communication problems. Under almost all circumstances communications problems found during the development stage are due to misunderstood protocols or incorrect programming by the user. Fortunately the analyzer can quickly identify and assist in correcting these problems.

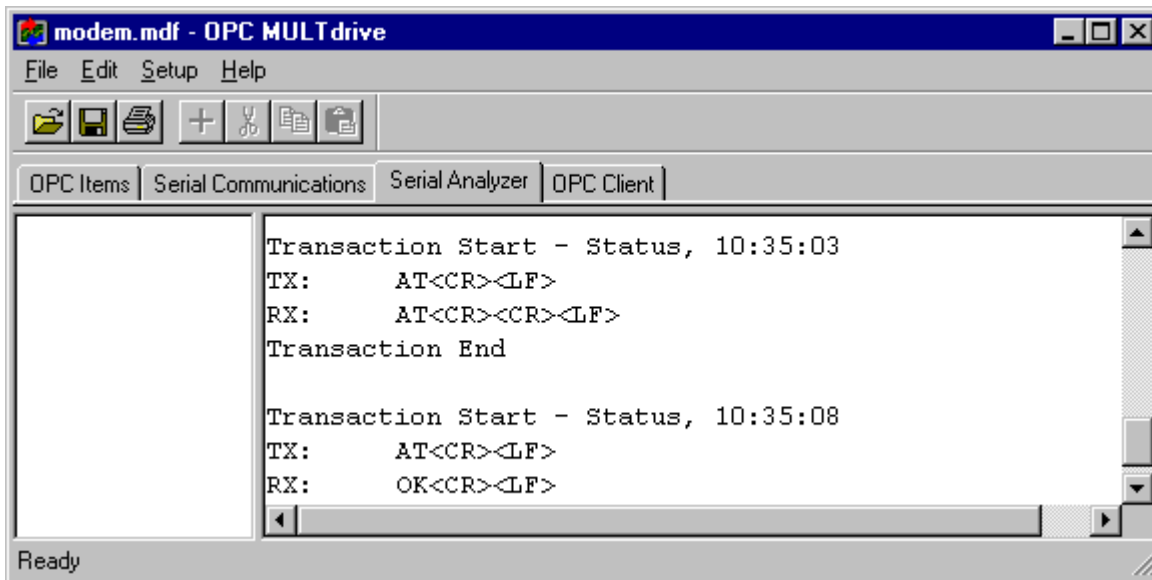
Before proceeding with this step, please ensure your Serial Analyzer is configured as seen below.

To Configure You Serial Analyzer.

1. Select menu option **Setup Serial Analyzer...**



Now that the analyzer has been turned on and configured for ASCII communications, select the ‘Serial Analyzer’ tab and view the resulting communications.

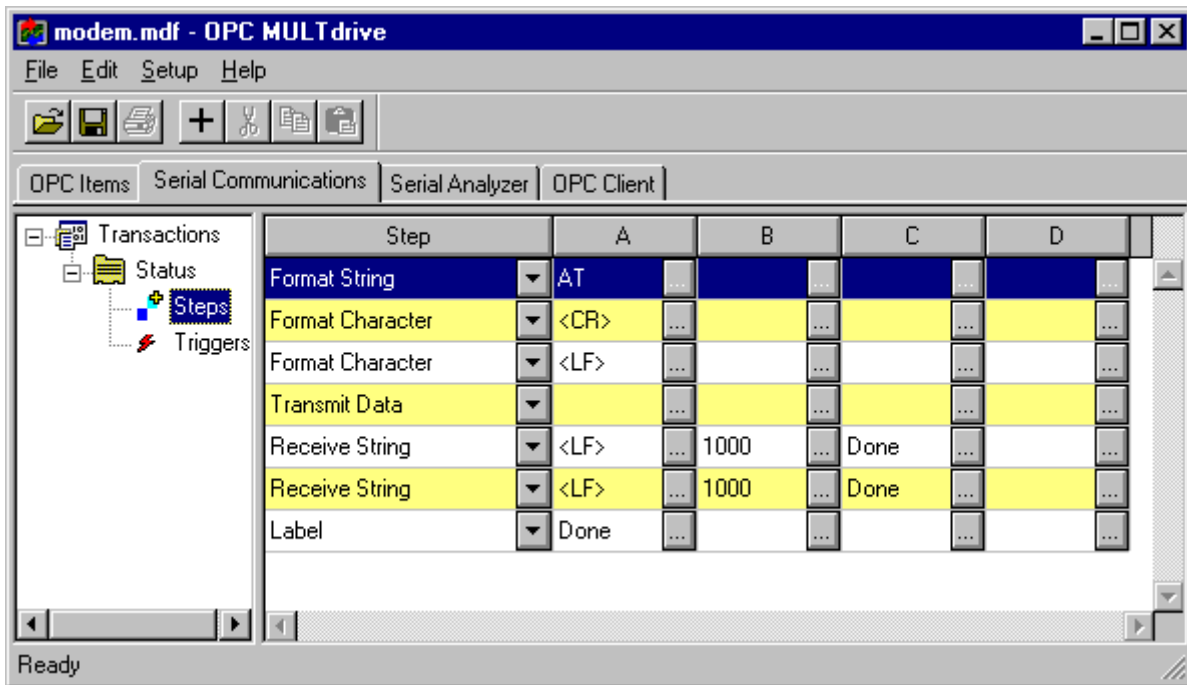


If you do not see the communications as above, chances are either the trigger has not been properly setup and is not executing the transaction, or more likely the communications port, baud rate, etc. are not correctly set. Be sure to use HyperTerminal to issue an **AT** command and ensure your properly communicating to your modem.

Our expectation is that when we send the **AT** command, we would receive a reply of 'OK' or something similar. After viewing the analyzer results we see that after sending the command, we either receive the command sent **AT<CR><LF>** or the actual reply **OK<CR><LF>**.

Actually, our problem is not a problem at all, simply a misunderstanding of full-duplex vs. half-duplex communications. Our modem is operating in full-duplex mode, meaning it echoes every character we send, prior to sending its reply. If this were not the case, we would not see our 'AT' command displayed on the screen when entering it from HyperTerminal.

To solve the problem we simply receive and discard the first line (our echoed command), and then continue as normal and receive our actual reply. To fix the problem add a second 'Receive String' command as seen below, save, and re-initialize the runtime system.



Once the change is made, the OPC item *General.Status* will display a constant 'OK' and your analyzer will display the proper communications everytime the transaction is executed.

## Transaction Error Handling

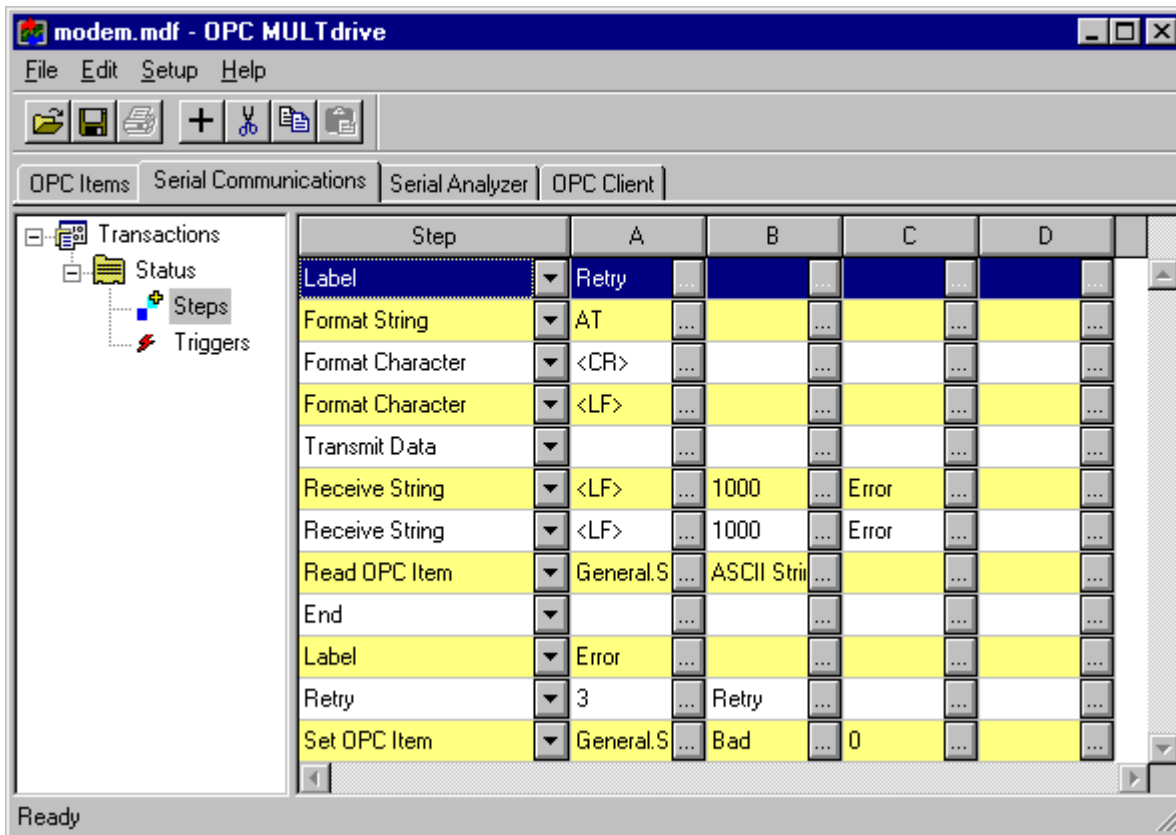
Error handling is an important part of all transactions. If the transaction is not executing properly, its up to the transaction itself to perform the necessary retries and/or set its OPC items to invalid. OPC items only reflect how they've been set by their related transaction, and so if the transaction fails and doesn't change the item, the item will continue to display its last value, along with a good status indication.

With respect to the steps in our transaction, we can see that if either of the two *Receive String* commands fail, execution will resume at the 'Done' label, and no further processing is performed. Since no OPC items are changed the item will remain "OK".

To fix up this transaction it is necessary to add the following additional steps...

- Retry logic to attempt a retry if a timeout occurs.
- Change the OPC item's status if a permanent failure occurs.

The steps below display the new transaction with the additional steps required to guard against failures. Add the new steps as seen below.



#### Summary of Changes

1. Insert a new **Label** at the top of our transaction. This provides a label to jump to when performing a retry.
2. Rename our existing 'Done' label to 'Error'. This is a little more descriptive of what's really happening.
3. Insert an **End** statement to properly terminate the transaction when correct processing is complete.
4. Add a **Retry** statement that will automatically jump to 'Retry' a maximum of 3 times before failing and dropping through.
5. Add a **Set OPC Item** statement to change the item's value and quality should the retry fail after 3 retries.

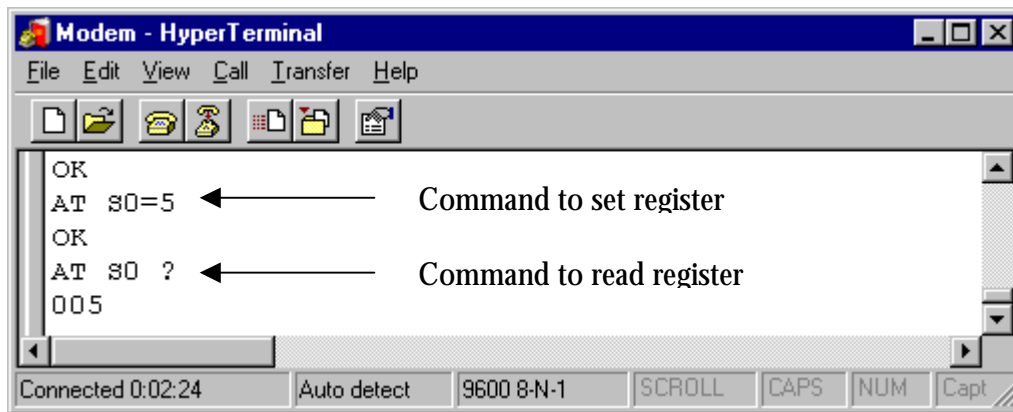
By re-initializing and testing this runtime, you can see its much more robust and changes the item's status if the modem is turned-off or not present. You can change the COM port via **Setup COM Settings...** to simulate the modem not being present

## Adding a Read/Write Item

Now that we have gone through the basic steps of creating an OPC item and its corresponding transaction, it is time to add a more complex OPC item; a read/write item. Most industrial devices contain one or more read only or read/write registers. The standard modem also has these types of items in the form of S registers (S1..Sxx).

For our next example we will add a new item called *AutoAnswerRings* and have it read or write the modem's S0 register, which corresponds to the modems auto-answer ring setting.

Using HyperTerminal, attempt to read & write the S0 register as seen below.

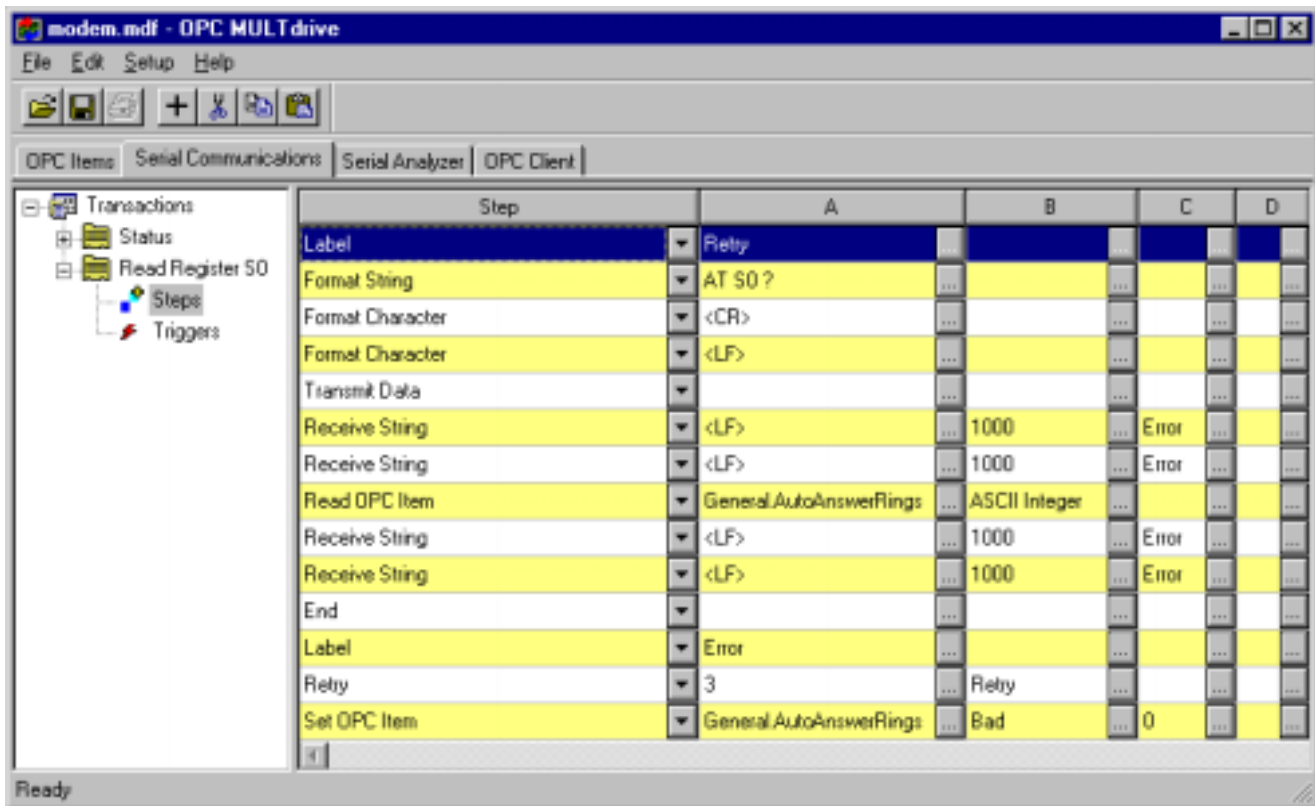


To Add the Read/Write Item

1. Select the OPC Items tab.
2. Select the group *General* or add a new group if desired.
3. Press <Ctrl-A> to add (or append) an item to the group.
4. Change the item's name to *AutoAnswerRings*
5. Change its **Type** to *Integer*
6. Change its **Rights** to *Read/Write*

To Add the Read Register S0 Transaction

1. Select the **Serial Communications** tab
2. Click Transaction, and then select **Edit Add Transaction.**
3. Name the transaction *Read Register S0*
4. Study and add each of the following steps...



### Comments:

1. Notice the step commands could have easily been copied & pasted from the *Status* transaction. Try this next time.
2. After retrieving the value from your modem (i.e. **Read OPC Item**) it may be necessary to read one or two more lines depending on what your modem replies with. The Serial Analyzer will immediately reveal this potential problem.

Had you attempted to re-initialize the runtime, you would have noticed it doesn't work. As mentioned before no transactions will execute until one of their triggers fire. Of course if there's no triggers, the transaction never executes.

We could add the *Timer* trigger again and have the item read periodically from the modem, or we can add an *OPC Item Read* or *OPC Item Update* trigger and have the item read only when requested by an attached OPC Client. This provides the benefit of communicating only when requested, increasing overall performance. Of course this strategy could backfire if many clients continually read the same register. Only the designer will know which approach is best suited for his/her application.

To Assign Triggers to the Transaction

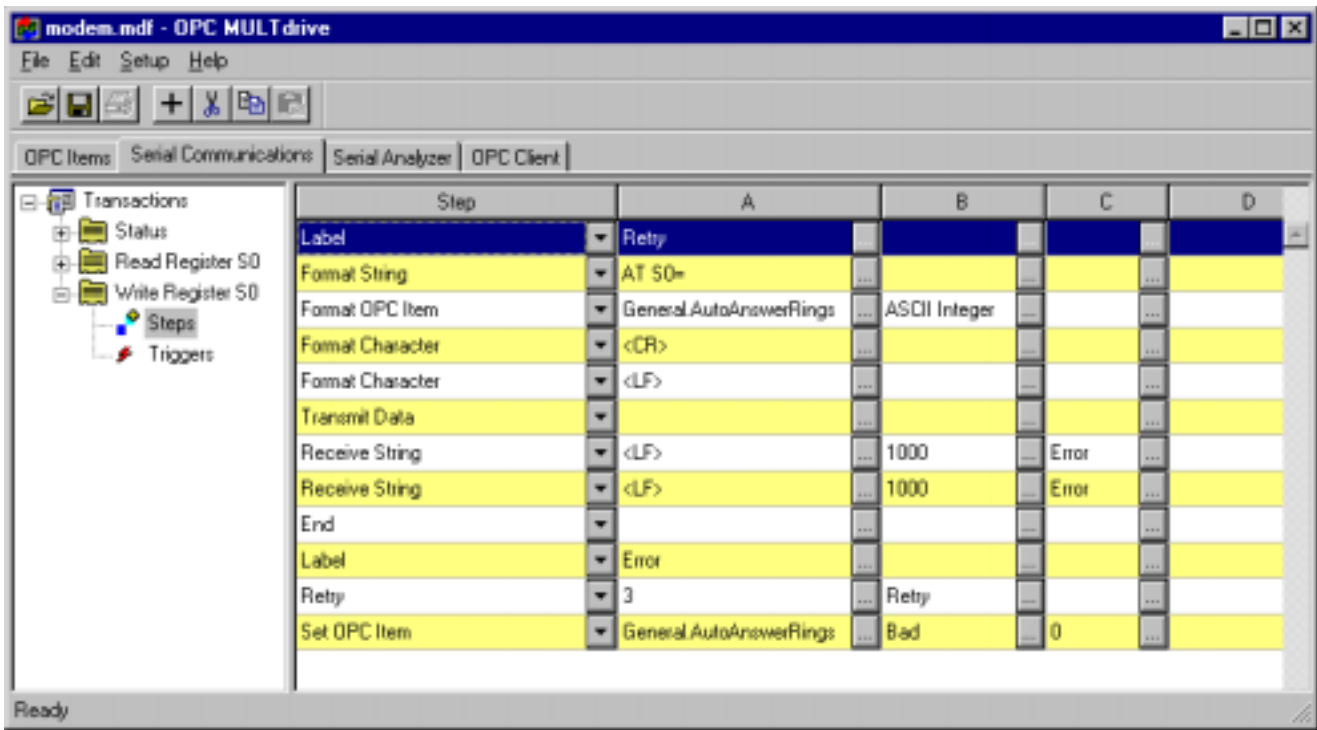
1. Expand the Read Register S0 Transaction
2. Select Triggers and add two new triggers.
3. Format the first trigger as *OPC Item Read*
4. Format the second trigger as *OPC Item Update*

Re-Initialize the runtime and attempt to debug communications. You may find it necessary to temporarily disable the *Status* transaction trigger. To do this expand the trigger view of the *Status* transaction and disable the *Timer* trigger. This will eliminate the possibility of this transaction communicating while debugging your new transaction.

Once you are satisfied with the read operation, continue on to add the write transaction.

To Add the Write Register S0 Transaction

1. Select the **Serial Communications** tab
2. Click Transaction, and then select **Edit Add Transaction.**
3. Name the transaction *Write Register S0*
4. Study and add each the following steps...



Once the following steps have been added, be sure to add an *OPC Item Write* trigger to ensure the trigger is executed each time a write operation is performed by an OPC Client.

Using the Serial Analyzer & OPC Client tabs, test the newly created transaction. Once tested, re-enable the *Status* transaction and re-test.

**Comment**

- Keep in mind that the modem is operating in full-duplex and anything transmitted must be received prior to processing the actual reply.

- Depending on the device/application it may not be necessary (or safe) to execute retries. For example, if the write fails and no retries were performed, the OPC item simply wouldn't change, and the operator would know something is immediately wrong.

## Exercises

1. Go back to the *Status* transaction and using the *Set OPC Item* step, mark the *AutoAnswerRings* item as being bad whenever communications fails. This way if communications fails, all tags/items will reflect a bad state.
2. Using your modem reference or standard AT command reference, add a new item for taking the phone on/off hook (**AT H0 and AT H1**). Add another for dialing a phone number (**ATD #**).

## Summary

This concludes our Tutorial of developing an OPC Modem Server. While a modem server may not be particularly useful in real life, it does provide an excellent example for learning. If your industrial equipment, RTU, etc. is located over a modem link, MULTIdrive™ could be programmed to provide additional functionality for managing the phone line.

Using the skills acquired in this tutorial, along with the reference material provided toward the end of this guide, you should have enough background knowledge to begin developing your first ASCII device interface. When developing an ASCII device driver, please keep in mind the following points.

- Define all OPC items (or tags) at the start of the project. Defining all the items together will produce a more cohesive list of items and groups.
- If your device accepts standard ASCII commands via a dumb terminal, use the terminal to test each command prior to implementing under MULTIdrive™. This will eliminate costly mistakes.
- Always keep in mind the mode of operation (full duplex or half duplex). It is very easy to forget about the echoed command coming back and not all that easy to identify the problem when it occurs.
- Develop transactions one at a time. Develop & test a single transaction before moving on to the next. This will build a knowledge base along with re-usable code that can be pasted into future transactions.
- Add error handling and retry-logic into each transaction. The time and effort spent early on testing these transactions, will payoff as this logic can be pasted into future transactions.

-



## Tutorial II – Modbus RTU

### *Developing Your First Binary Protocol.*

**N**ow that we have developed an ASCII protocol, it is time to develop a binary protocol with a few more complexities. Unfortunately, to successfully work with this tutorial it will be necessary to acquire an industrial device that supports the Modbus RTU protocol. We highly recommend locating a modbus device and completing this tutorial before moving on to your own protocol implementation.

### Prerequisites

Successful completion of Tutorial I

Modbus device set for RTU communications at device address 1.

Since we have completed Tutorial I, this tutorial runs at a faster pace and no longer provides detailed instructions on adding/editing OPC items, transactions, steps, etc. As well, this tutorial only explains the Modbus protocol to a level required by the tutorial. If the reader requires a more complete description of the Modbus protocol, copies of the specification are readily available on the internet. At the time of development of this tutorial, two Modbus protocol specifications were found at the following http addresses.

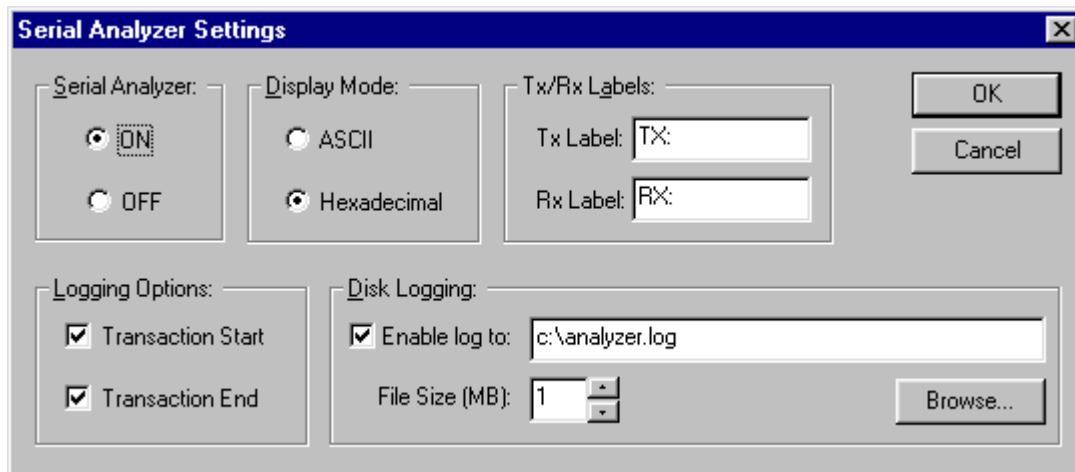
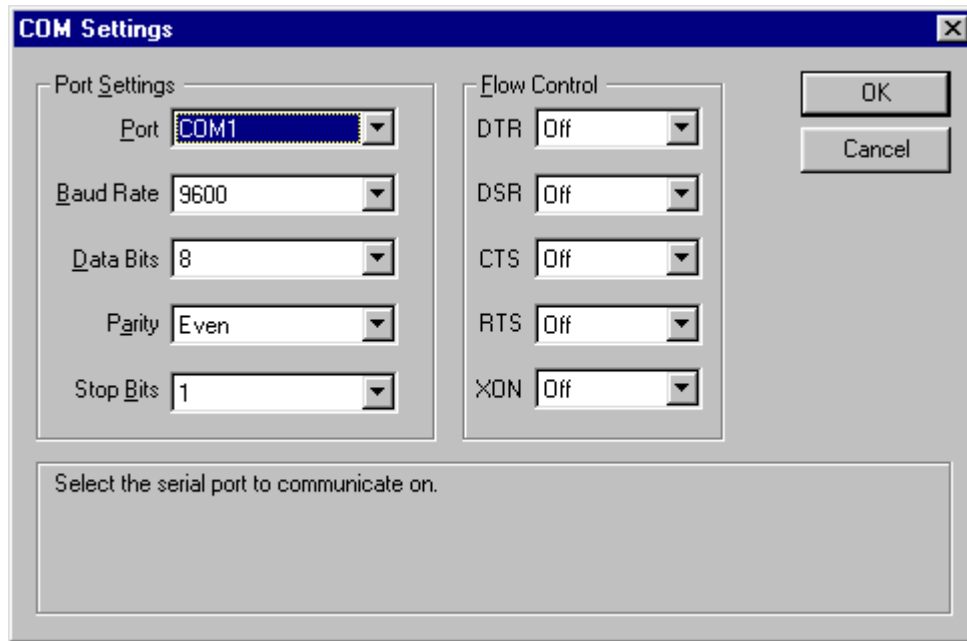
[http://www.modicon.com/TECHPUBS/TechPubNew/PI\\_MBUS\\_300.pdf](http://www.modicon.com/TECHPUBS/TechPubNew/PI_MBUS_300.pdf)

<http://www.gefanuc.com/support/plc/downloads/S9070/cksumrun.txt>

## Getting Started

Assuming you have successfully located a Modbus device, cable, etc. The first step is to properly create a new project, configure MULTIdrive's COM settings and configure the Serial Analyzer for proper *binary* communications. A MULTIdrive™ definition file (MDF) of ModbusRTU.MDF has been installed in your MULTIdrive™ directory to provide further assistance should it be required.

The following dialogs below display the recommended settings. The reader should only deviate from these settings if they are confident with their changes. Of course the port may have to be changed to COM2, COM3, etc.



## Adding Tags to Your OPC Server

As described in Tutorial I, the first step required when developing a protocol is to develop the OPC tag (or item) list early on. MULTIdrive™ is intended to solve driver problems, where drivers don't currently exist. Its power & flexibility allow it to communicate to almost any device quickly & efficiently. Unfortunately there is a price to be paid, and this price comes in the form of a static protocol & tag definition. For example a normal Modbus OPC Server, would allow a user to select the device address, the registers to read and then build the necessary packets behind the scenes. If the user changes their mind, they simply specify a new set of registers. With MULTIdrive™, the user must know ahead of time exactly what is to be read from the device, and then build the appropriate protocol packets. It is therefore even more important to ensure the information to be read from the device is well defined (i.e. in the form of OPC items) before any protocol development starts.

For purposes of this tutorial we have chosen to read a sample set of 16 discrete inputs (10001-10016), 16 discrete outputs (00001-00016), 16 input registers (30001-30016) and 16 holding registers (40001-40016). For clarity we have decided to use the actual register address as the item name. So for example, the item that holds the contents of register 40014 is simply called '40014'. This addressing scheme provides a universal modbus protocol interface. In a real life application, the user may choose to use a more descriptive item name, describing the actual point. This way connecting OPC Clients can immediately identify the contents of the tag. For example, if register 40014 held the engine speed of an engine in RPM, the item name may be 'EngineSpeed', with a description of 'Engine speed (RPM).' Of course the user will know which naming convention makes the most sense.

Using MULTIdrive™, enter the following *OPC Item* definitions. Be sure to use MULTIdrive's *Cut/Copy/Paste* functionality.

The screenshot shows the 'ModbusRTU.mdf - OPC MULTIdrive' application window. The interface includes a menu bar (File, Edit, Setup, Help), a toolbar with icons for file operations, and a tabbed interface with 'OPC Items' selected. On the left, a tree view shows 'OPC Items' expanded to 'DiscreteInputs'. The main area contains a table with the following data:

Item	Type	Description	Rights	Scaling
10001	Discrete	Modicon discrete input.	Read Only	None
10002	Discrete	Modicon discrete input.	Read Only	None
10003	Discrete	Modicon discrete input.	Read Only	None
10004	Discrete	Modicon discrete input.	Read Only	None
10005	Discrete	Modicon discrete input.	Read Only	None
10006	Discrete	Modicon discrete input.	Read Only	None
10007	Discrete	Modicon discrete input.	Read Only	None
10008	Discrete	Modicon discrete input.	Read Only	None
10009	Discrete	Modicon discrete input.	Read Only	None
10010	Discrete	Modicon discrete input.	Read Only	None
10011	Discrete	Modicon discrete input.	Read Only	None
10012	Discrete	Modicon discrete input.	Read Only	None
10013	Discrete	Modicon discrete input.	Read Only	None
10014	Discrete	Modicon discrete input.	Read Only	None
10015	Discrete	Modicon discrete input.	Read Only	None
10016	Discrete	Modicon discrete input.	Read Only	None

The status bar at the bottom left of the window displays 'Ready'.

***Modbus Discrete Inputs***

Item	Type	Description	Rights	Scaling
00001	Discrete	Modicon discrete output.	Read Only	None
00002	Discrete	Modicon discrete output.	Read Only	None
00003	Discrete	Modicon discrete output.	Read Only	None
00004	Discrete	Modicon discrete output.	Read Only	None
00005	Discrete	Modicon discrete output.	Read Only	None
00006	Discrete	Modicon discrete output.	Read Only	None
00007	Discrete	Modicon discrete output.	Read Only	None
00008	Discrete	Modicon discrete output.	Read Only	None
00009	Discrete	Modicon discrete output.	Read Only	None
00010	Discrete	Modicon discrete output.	Read Only	None
00011	Discrete	Modicon discrete output.	Read Only	None
00012	Discrete	Modicon discrete output.	Read Only	None
00013	Discrete	Modicon discrete output.	Read Only	None
00014	Discrete	Modicon discrete output.	Read Only	None
00015	Discrete	Modicon discrete output.	Read Only	None
00016	Discrete	Modicon discrete output.	Read Only	None

### *Modbus Discrete Outputs*

The screenshot shows the 'ModbusRTU.mdf - OPC MULTIdrive' application window. The interface includes a menu bar (File, Edit, Setup, Help), a toolbar with icons for file operations, and a tabbed interface with 'OPC Items', 'Serial Communications', 'Serial Analyzer', and 'OPC Client' tabs. On the left, a tree view shows 'OPC Items' expanded to 'RegisterInputs'. The main area contains a table with the following data:

Item	Type	Description	Rights	Scaling
30001	Integer	Modicon register input.	Read Only	None
30002	Integer	Modicon register input.	Read Only	None
30003	Integer	Modicon register input.	Read Only	None
30004	Integer	Modicon register input.	Read Only	None
30005	Integer	Modicon register input.	Read Only	None
30006	Integer	Modicon register input.	Read Only	None
30007	Integer	Modicon register input.	Read Only	None
30008	Integer	Modicon register input.	Read Only	None
30009	Integer	Modicon register input.	Read Only	None
30010	Integer	Modicon register input.	Read Only	None
30011	Integer	Modicon register input.	Read Only	None
30012	Integer	Modicon register input.	Read Only	None
30013	Integer	Modicon register input.	Read Only	None
30014	Integer	Modicon register input.	Read Only	None
30015	Integer	Modicon register input.	Read Only	None
30016	Integer	Modicon register input.	Read Only	None

The status bar at the bottom left indicates 'Ready'.

***Modbus Input Registers***

Item	Type	Description	Rights	Scaling
40001	Integer	Modicon register.	Read/Write	None
40002	Integer	Modicon register.	Read/Write	None
40003	Integer	Modicon register.	Read/Write	None
40004	Integer	Modicon register.	Read/Write	None
40005	Integer	Modicon register.	Read/Write	None
40006	Integer	Modicon register.	Read/Write	None
40007	Integer	Modicon register.	Read/Write	None
40008	Integer	Modicon register.	Read/Write	None
40009	Integer	Modicon register.	Read/Write	None
40010	Integer	Modicon register.	Read/Write	None
40011	Integer	Modicon register.	Read/Write	None
40012	Integer	Modicon register.	Read/Write	None
40013	Integer	Modicon register.	Read/Write	None
40014	Integer	Modicon register.	Read/Write	None
40015	Integer	Modicon register.	Read/Write	None
40016	Integer	Modicon register.	Read/Write	None

Ready

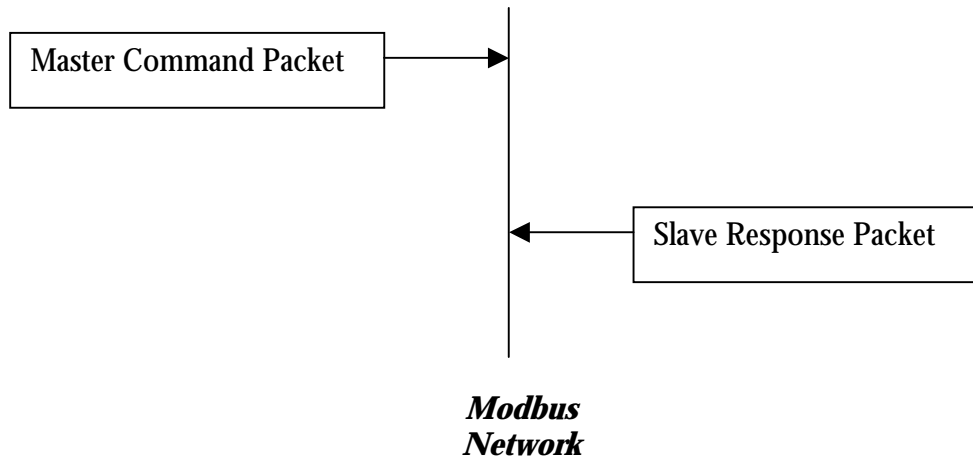
### *Modbus Holding Registers*

## Modbus 101

In order to define the necessary transactions required to satisfy our OPC item list, it is necessary to understand the modbus RTU protocol. The following paragraphs briefly describe the protocol along with the commands we'll be using. If the reader is having difficulty following, we ask they locate a more in depth protocol specification from the internet. The search words '*modbus specification*' will most likely yield volumes of descriptive explanations.

### Modbus Transactions

All modbus communications is performed between a single master device and one or more slave devices. Outside of some basic broadcast functionality, only two devices are ever communicating at one time. The master (in our case MULTIdrive™) issues commands that will be answered by a single slave device (in our case the acquired modbus device). If the slave device understands the command and can properly execute it, it sends back a reply. If the command reaches the slave in error, the slave simply does not respond.



Both the command & reponse packets are similar in structure. They both start with a modbus device address, command, data , and then end with a 16 bit CRC.

<b>Modbus Address</b>	<b>Modbus Command</b>	<b>Data</b>	<b>CRC-16</b>
01 (BYTE)	02 (BYTE)	.... (1...250 BYTES)	LO HI (WORD)

### ***Modbus Message Structure***

Depending on the command sent, the message packet contains a mix of 8 bit (byte) and 16 bit (word) data items. As soon as the binary data becomes multiple byte lengths (2, 3, 4, etc.) the programmer needs to become concerned with the order the bytes are sent so he/she can properly inform MULTIdrive™ of how the data is formatted. For example a 16 bit word can be sent hi byte first (MSB) followed by its low byte (LSB). Or, it can

be sent low byte first (LSB) followed by its high byte (MSB). The explanation of why this happens is an entirely different story that we are not going to get into, only to say, always double and triple check the data formats.

With respect to the modbus protocol, 16 bit data is sent MSB LSB, referred to within MULTIdrive as *Motorola 16 Bit Unsigned* data. Unfortunately the 16 bit CRC is sent the opposite way of LSB MSB, referred to as *Intel 16 Bit Unsigned*. If the user incorrectly identifies these formats, the resulting data will be corrupt.

#### Modbus Commands

A different Modbus command is required to read each of the 4 data types... 1XXXX input coils, 0XXXX output coils, 3XXXX input registers and 4XXXX holding registers. Fortunately multiple items can be read within a single message transaction and so a single transaction of each type can satisfy all our data needs.

The following tables quickly define the format of each command and response packet for each of the modbus commands we need to execute for this tutorial. Where possible actual sample data is plugged in as it will be sent by us. The tables are displayed in hexadecimal.

#### Read Input Status (1XXXX)

	<b>CMD</b>		<b>RESP</b>	
Slave Address	01		01	Slave Address
Function	02		02	Function
Starting Address HI	00		02	Byte Count
Starting Address LO	00		00	Data (10001-10008)
# of Points HI	00		00	Data (10009-10016)
# of Points LO	16		XX	CRC - LO
CRC - LO	10		XX	CRC - HI
CRC - HI	79			

#### Read Output Status (0XXXX)

	<b>CMD</b>		<b>RESP</b>	
Slave Address	01		01	Slave Address
Function	01		01	Function
Starting Address HI	00		02	Byte Count
Starting Address LO	00		00	Data (10001-10008)
# of Points HI	00		00	Data (10009-10016)
# of Points LO	10		XX	CRC - LO
CRC - LO	3D		XX	CRC - HI
CRC - HI	C6			

Read Input Registers (3XXXX)

	CMD		RESP	
Slave Address	01		01	Slave Address
Function	04		04	Function
Starting Address HI	00		02	Byte Count
Starting Address LO	00		00	Register HI (30001)
# of Points HI	00		00	Register LO (30001)
# of Points LO	10		00	Register HI (30002)
CRC - LO	F1		00	Register LO (30002)
CRC - HI	C6		...	Remaining registers
			XX	CRC - LO
			XX	CRC - HI

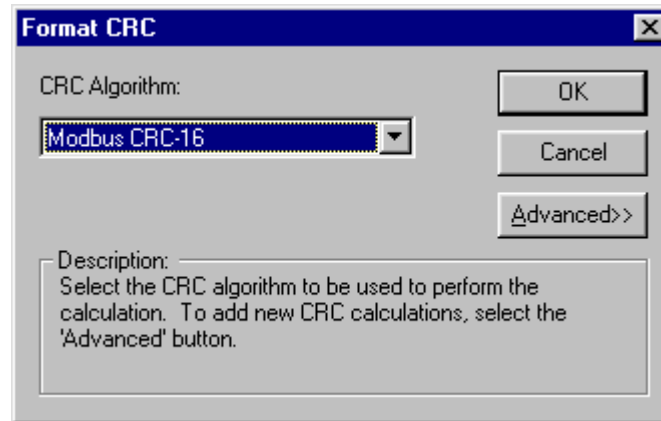
Read Holding Registers (4XXXX)

	CMD		RESP	
Slave Address	01		01	Slave Address
Function	03		03	Function
Starting Address HI	00		02	Byte Count
Starting Address LO	00		00	Register HI (30001)
# of Points HI	00		00	Register LO (30001)
# of Points LO	16		00	Register HI (30002)
CRC - LO	44		00	Register LO (30002)
CRC - HI	06		...	Remaining registers
			XX	CRC - LO
			XX	CRC - HI

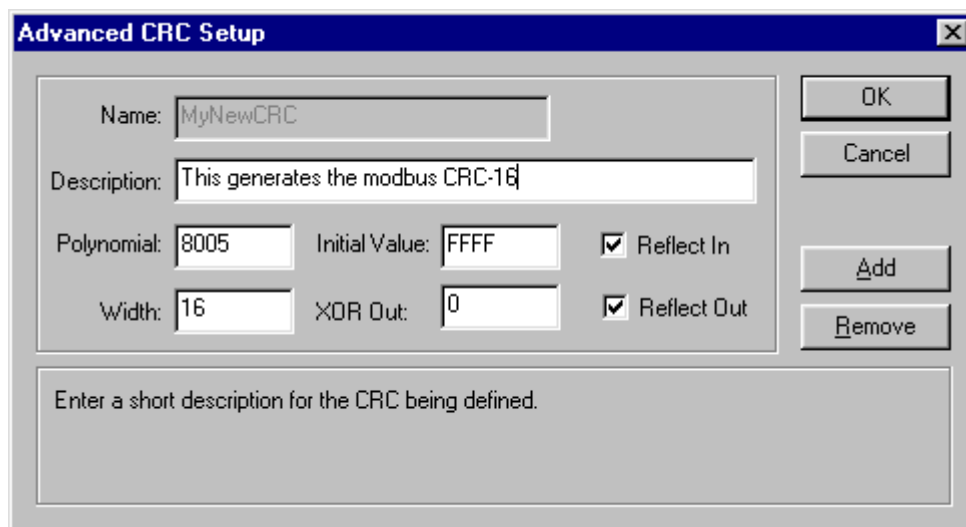
Modbus CRC-16

Whenever a protocol employs a CRC, the *protocol specification document* typically provides a detailed explanation of the CRC generation along with sample source code. This information is vital when having MULTIdrive™ generate the CRC. While most CRCs are created equal, variations exist in their input parameters (initial CRC values, polynomials, XORing, etc.) The CRC generation capabilities of MULTIdrive™ should be sufficient to generate most CRCs used today.

The CRC-16 found within the modbus protocol can be defined in one of two ways. First, it can simply be selected from the list of available CRCs when using commands such as *Format OPC & Jump Bad CRC...*



Or, using the **Advanced** button, the CRC can be custom defined with detailed knowledge of the CRC...



We recommend using the first method, as it is much faster than the user defined version.

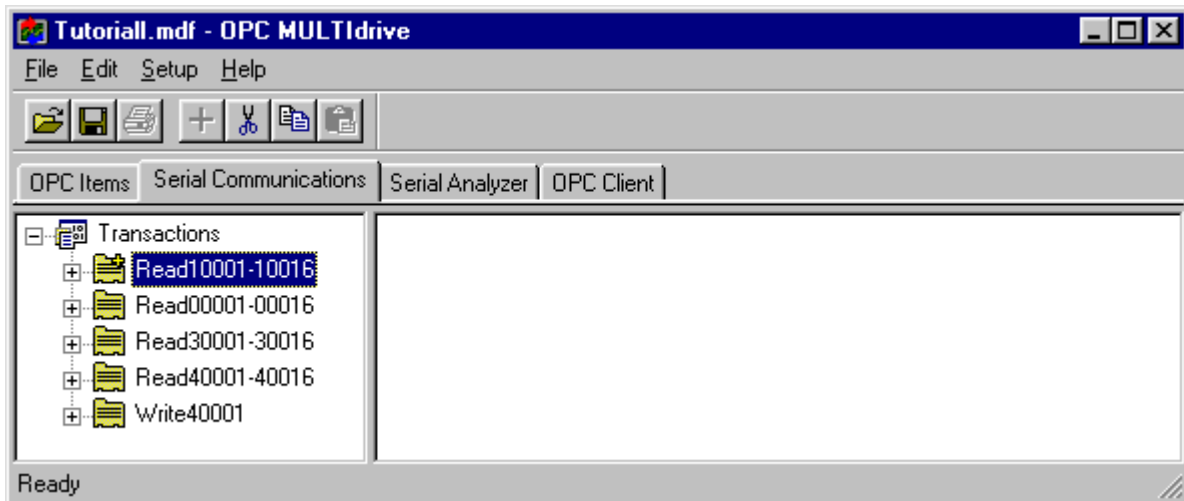


Generating the right CRC can sometimes be a difficult task when provided sketchy documentation. The best remedy to this problem, is to have a known CRC value for a given packet. Have MULTIdrive™ send the same packet, and then quickly monitor the CRC results with the *Serial Analyzer*. This provides an extremely quick method to modify algorithms until the correct settings are found. Without a sample packet/CRC, you'll never know if the calculated CRC is correct. And of course the target slave device will never respond to a command packet with an incorrect CRC.

## Adding Transactions

The transaction list is implicitly defined by the OPC item list, so once our items are defined, the necessary transactions will fall out. Of course to define the transaction list it is necessary to have a complete understanding of the communications protocol and how it will be applied. With respect to our tutorial, we have to send a single command for each type of register; 1XXXX, 0XXXX, 3XXXX, and 4XXXX, resulting in 4 transactions. A 5<sup>th</sup> transaction will be require to perform a sample write operation of register 40001.

Define the following transactions...



## Defining the Transaction Steps

Once each of the transactions has been created, it is necessary to define each of the steps within the transactions. As described in other areas of this manual it is important to focus on getting a single transaction working before moving on to a second or third transaction.

Under normal circumstances, the user will first create a skeleton set of transaction steps. Once the transaction is properly working, the user may then begin to add additional steps to convey good/bad OPC item status, and finally add necessary retry logic to automatically perform retries. If the user concentrates on a single transaction during these phases, the transaction can be copied to all other transactions and provide a basic (proven) framework for all communications. If the user develops a skeleton version of each transaction first, then each transaction will have to be edit/upgraded individually whenever enhancements are made.

Define the following transaction steps for the *Read10001-10016* transaction...

Step	A	B	C	D
Label	Start			
Format Data (8)	01 02 00 00 00 16			
Format CRC	0	0	Modbus CRC-16	Intel 16 Bit Unsigned
Transmit Data				
Receive Data	7	500	Retry	
Jump Bad CRC	0, 4	Modbus CRC-16	Intel 16 Bit Unsigned	Retry
Jump Not Equal	1	Intel 8 Bit Unsigned	Retry	
Read Data (8):	1			
Jump Not Equal	2	Intel 8 Bit Unsigned	Retry	
Read Data (8):	1			
Jump Not Equal	2	Intel 8 Bit Unsigned	Retry	
Read Data (8):	1			
Read OPC Item	DiscreteInputs.10001	Bit 0		
Read OPC Item	DiscreteInputs.10002	Bit 1		
Read OPC Item	DiscreteInputs.10003	Bit 2		
Read OPC Item	DiscreteInputs.10004	Bit 3		
Read OPC Item	DiscreteInputs.10005	Bit 4		
Read OPC Item	DiscreteInputs.10006	Bit 5		
Read OPC Item	DiscreteInputs.10007	Bit 6		
Read OPC Item	DiscreteInputs.10008	Bit 7		
Read Data (8):	1			
Read OPC Item	DiscreteInputs.10009	Bit 0		
Read OPC Item	DiscreteInputs.10010	Bit 1		
Read OPC Item	DiscreteInputs.10011	Bit 2		
Read OPC Item	DiscreteInputs.10012	Bit 3		
Read OPC Item	DiscreteInputs.10013	Bit 4		
Read OPC Item	DiscreteInputs.10014	Bit 5		
Read OPC Item	DiscreteInputs.10015	Bit 6		
Read OPC Item	DiscreteInputs.10016	Bit 7		
End				
Label	Retry			
Clear				
Retry	3	Start		
Set OPC Item	DiscreteInputs.10001	0	0	
Set OPC Item	DiscreteInputs.10002	0	0	
Set OPC Item	DiscreteInputs.10003	0	0	
Set OPC Item	DiscreteInputs.10004	0	0	
Set OPC Item	DiscreteInputs.10005	0	0	
Set OPC Item	DiscreteInputs.10006	0	0	
Set OPC Item	DiscreteInputs.10007	0	0	
Set OPC Item	DiscreteInputs.10008	0	0	
Set OPC Item	DiscreteInputs.10009	0	0	
Set OPC Item	DiscreteInputs.10010	0	0	
Set OPC Item	DiscreteInputs.10011	0	0	
Set OPC Item	DiscreteInputs.10012	0	0	
Set OPC Item	DiscreteInputs.10013	0	0	
Set OPC Item	DiscreteInputs.10014	0	0	
Set OPC Item	DiscreteInputs.10015	0	0	
Set OPC Item	DiscreteInputs.10016	0	0	

***Label***

The *Label* command provides a location for the top (or Start) position of the transaction. Without this label, there is no way to jump to the top of the transaction, as in the *Retry* command found later in the transaction.

***Format Data (8)***

The Format Data (8) command is used to define the bulk of the command message. The data defined, is defined one byte at a time, and therefore can be entered exactly as it will be sent out on the command message. Care must be taken understanding the base the numbers were entered in. For example, these numbers are specified in decimal (base 10), however the *Serial Analyzer* will display these values in hexadecimal (base 16) when it is transmitted. This may confuse the weary programmer. Of course the data entered with the Format Data (8) command can be specified in any base (e.g. binary, octal, decimal, and hexadecimal). Typically you'll want to use the same base as used in the protocol specification documentation.

Another small comment should be made here. Within this tutorial the entire command message is created within this single command. It is very easy to have the same command message generated with multiple transaction step. For example, a *DeviceAddress* tag could format the modbus address, while a *BaseAddress* tag could format the base address to read, etc., providing a much more versatile transaction. For simplicity however, we choose to specify the entire command.

***Format CRC***

After the command has been placed into the transmit buffer, the CRC can be calculated for the message. Care must be taken specifying the algorithm to use and the method to format the resultant bytes. The *Serial Analyzer* will always tell you how the CRC was finally transmitted.

***Transmit Data******Receive Data***

The transmit and receive data commands transmit the command message in the transmit buffer and receive the resulting response message in the receive buffer. If the specified number of bytes is not read within 500 milliseconds, the *Retry* label will be jumped to.

It is important to know and specify the exact number of bytes that will be returned, otherwise MULTIdrive™ will not know when the message is complete. Too few bytes, and the CRC will not be received, too many, and MULTIdrive™ will timeout waiting for more bytes that aren't coming. In other applications outside this tutorial some devices may have one or more responses to a given command. Under almost all circumstances however, one response represents the desired response, while all others represent some form of an error response, in which case something is wrong and we can continue anyway.

***Jump Bad CRC***

Once the response is received we immediately check the CRC and validate the packet. If the packet is in error, we immediately attempt a retry.

***Jump Not Equal******Read Data (8)***

Once the CRC is checked we can assume we are looking at good data, unfortunately we can't assume it's the correct response. It is entirely possible for the receive buffer to contain the previous response message. This is unlikely, but never the less possible. To confirm we have the right response we continue to check the device address, function code and number of bytes fields via pairs of *Jump Not Equal* and *Read Data (8)* commands. If any of these checks fail, we are not looking at the right response.

It should be noted here that the *Jump Not Equal* command performs a comparison operation, but does not move the receive buffer position forward, while the *Read Data (8)* command moves the receive buffer position ahead a single byte.

To drive home the necessity to validate the response beyond the basic CRC check is the following scenario: Had we sent this same command message to our mouse (say COM2), interesting enough the mouse would echo back the transmit message. With our transmit message echoed back, our CRC would compute fine, the device address and function codes would also check out fine. We would then start setting tags that don't reflect actual process data.

### ***Read OPC Item***

Once we have confirmed our response, we can begin to extract data from the message into our items. *Read OPC Item* can read almost any data format and update any OPC item defined. It is important to understand the incoming data format so the appropriate format can be selected. Once the data is read, the receive buffer position is incremented by the number of bytes in the item. If the item is less than 1 byte in length (i.e. 1 or more bits) the position is **NOT** incremented. This way additional *Read OPC Item* commands can be executed upon the same byte.

### ***Read Data (8)***

After reading 8 bits of discrete data from the same byte, it is necessary to manually increment the receive buffer position by 1 byte to the next 8 discrete inputs. Had the *Read OPC Item* been reading something bigger like a 16 bit register the receive position would have automatically been incremented with each *Read OPC Item* command.

### ***End***

Once the response message has been successfully parsed, it is necessary to end the transaction. Under normal circumstances transaction execution ends at the bottom of the step list. However due to our retry steps below this point, it is necessary for us to explicitly end transaction execution.

### ***Label***

The label *Retry* basically provides a location to jump to if a retry is to be executed. Similar to the *Start* label at the top.

### ***Clear***

If the *Retry* label is jumped to, something is wrong. The *Clear* command clears the transmit and receive buffers so we may start over. If the receive buffer is not cleared, it is possible we may begin processing remnants of our last unsuccessfully received message.

### ***Retry***

The *Retry* command performs a loop operation jumping to the *Start* label a maximum of 3 times, before falling through and executing the steps below it. This essentially implements our retry sequence.

### ***Set OPC Item***

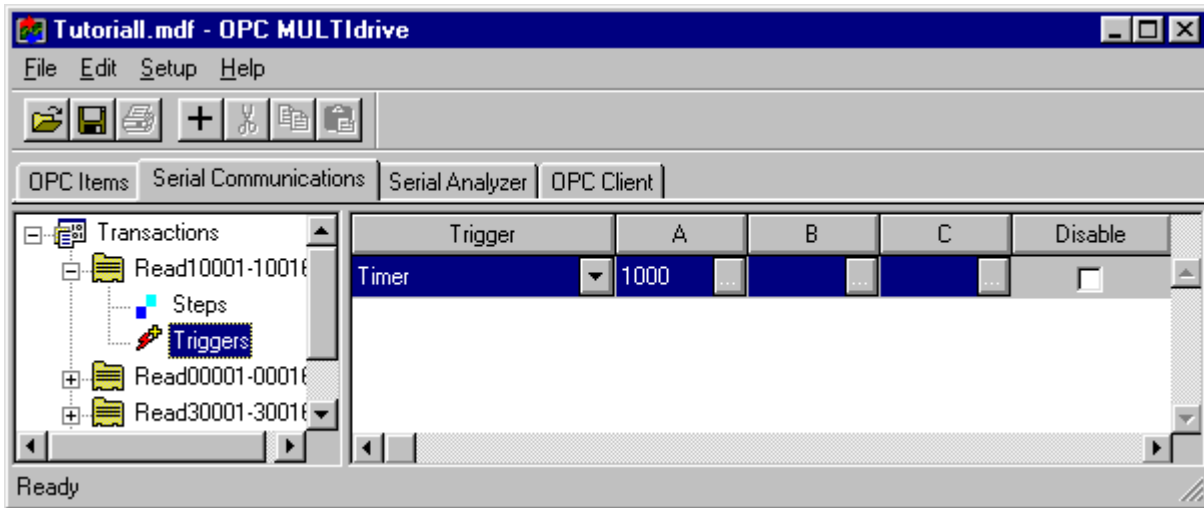
After 3 failed retries it is time for us to manually set each OPC item in question to a bad status (or quality). In this particular example, we set the item's value to 0 and its quality to bad (0). Should the client be connected to any of these items, they will instantly be notified of their status change.

In some situations it may be desirable to change the item's quality to bad, but leave its value intact as the last know value rather than setting to zero. This is possible by simply not specify a value in the (A) parameter.

## Adding Transaction Triggers

Before the transaction will execute it will need to be activated by a trigger. For this application we've decided to poll the Modbus device every second. To do this we only have to add a single *Timer* trigger to the transaction we wish to execute every second.

Add the trigger as below...



***Transaction Trigger***

## Exercise I

Using the *Serial Analyzer* & *OPC Client* tabs verify & test the operation of the first transaction. If you are having trouble try the supplied *ModbusRTU.MDF* file. When the transaction is working, move on to Exercise II.

## Exercise II

Using the first working transaction, copy and define the other 3 read transactions. Each new transaction will need adjustments to support the new transaction. Below is a list of the adjustments that will be needed.

Each transaction sends a similar, but different modbus command.

The number of bytes received will vary for each of the different commands.

The checks after the message is received along with the CRC will need some minor adjustments.

The *Read/Set OPC Item* commands will all have to be adjusted to use the right items. As well, there data will have to be changed to *Motorola 16 Bit Unsigned* for 3XXXX & 4XXXX registers.

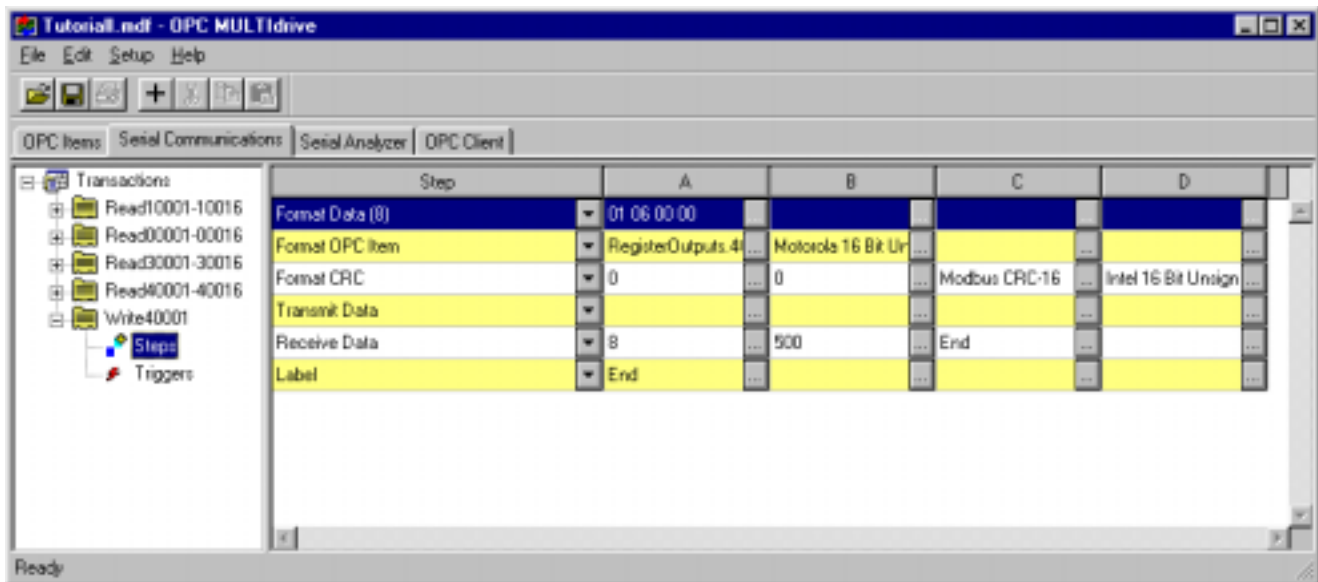
If you encounter problems building the new transactions, refer to the supplied *ModbusRTU.MDF* definition file for verification.

## Writing 4XXXX Registers

Once all the read transactions are complete & operational, it is time to add the write operations. The only *read/write* items in our tutorial are the 4XXXX registers. This tutorial only adds a single write transaction for register 40001. The supplied MDF file however, has write transactions for each of the 4XXXX registers. More can be added if desired simply by copying the first transaction to 40002-40016 and making some minor alterations.

A single write transaction is required for each item to be written to the device. While a block of registers can be written in a single transaction, we have decided not to perform write operations that way.

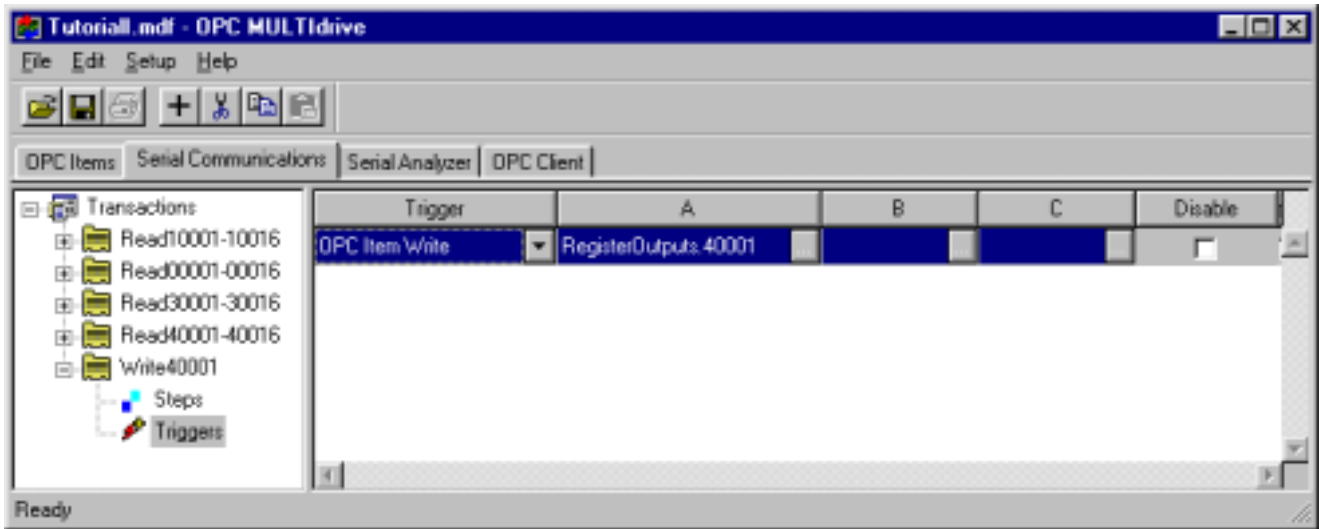
Build the write transaction as follows...



### ***Write Transaction***

The first thing you'll notice is the write transaction is much smaller than the other read transactions, this is because we are only dealing with a single OPC item. You'll also notice the transaction performs no retries, and so if the write fails, nothing happens. We have chosen this method as all write operations will be performed by an operator and so each write operation can be visually verified as it is performed. Another school of thought may say we should have added the retry logic.

This transaction will only need to be executed when a write request comes in for OPC item '40001'. Using the display below, add the following trigger...



***Write Transaction Trigger***

Using the *Serial Analyzer* & *OPC Client* tabs verify the write operation works as desired.

## Serial Analyzer

### *Monitor, Log and Analyze Serial Communications.*

**T**he Serial Analyzer allows you to monitor, log, and analyze serial communications at any time during or after protocol development. This tool is essential during the development process as it provides instant feedback as to the communications between MULTIdrive™ and the end device.

The MULTIdrive™ Serial Analyzer has the following features:

- Displays both ASCII and Hexadecimal formats.

- Works with both ASCII & binary protocols.

- Logs data to disk in a standard text file from one to several mega-bytes.

- Logs supplemental information; such as the transaction executing and the time it was executed.

- Operates in both development & runtime environments.

The Serial Analyzer works in concert with executing MULTIdrive™ transactions, and not independently. By this we mean if no transactions are transmitting or receiving information, no data will be logged. On the other hand each byte/character transmitted or received by a transaction will be logged. This has the benefit of logging communications exactly as being processed by the transactions, while having the negative effect of not logging communications not processed by a transaction. For example, if a device continually sends its pressure reading, and this reading is not expected or read by any transactions, the data will be received by the PC, but never logged by the Serial Analyzer.



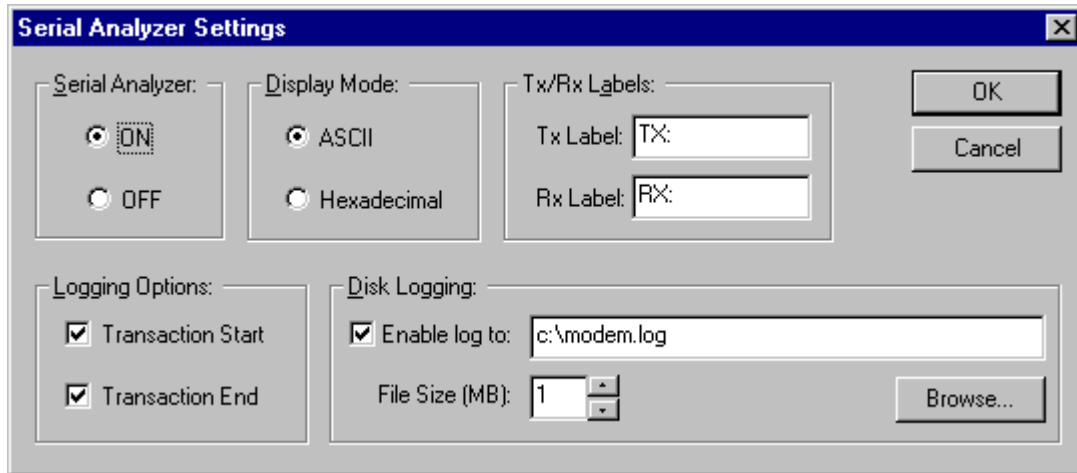
In situations where it is not known exactly how much data will be received from a device, it is recommended the programmer attempt to read more data than is expected. This way the Serial Analyzer will properly log all data received, allowing the programmer to see exactly what was received. If less data is read by a transaction than is received by the PC, the data will not be logged by the analyzer and will be left for another transaction to read/log.

Setting up the Serial Analyzer

All settings for the serial analyzer are performed through a single setup dialog.

To setup the serial analyzer:

1. Select **Setup Serial Analyzer...** from the main menu.
2. Configure any of the analyzer settings.
3. Click **OK** to save the settings



### Serial Analyzer

Determines whether the serial analyzer is ON and logging data, or OFF and not logging data. The serial analyzer is typically left ON during the development & test phase, and OFF for final testing and deployment. When OFF, no analyzer functions are performed and therefore optimum MULTIdrive™ performance is achieved

### Display Mode

Determines whether data will be logged in ASCII or hexadecimal formats. This setting should be set to **ASCII** for those devices using an ASCII protocol, and to Hexadecimal for those devices using a binary (or non-ASCII) protocol. The logged data will be illegible if the setting is incorrect. Changing this setting changes how new data is logged, and not how old data is displayed.

### Tx/Rx Labels:

Defines labels to prefix each outgoing (Tx) or incoming (Rx) message logged by the analyzer. The log becomes more readable by setting the Tx label to something like; *MULTIdrive, PC, COM1*, etc., and the Rx label to the name of the device; *PLC, PID, sensor*, etc

### Logging Options

Display a list of supplementary information that can be logged or not. *Transaction Start* logs the both the name of the transaction about to execute, along with the time it was executed. *Transaction End*, logs the completion of a transaction. This information informs the user of what transaction executed, so the user knows the type of communications that will be performed

**Disk Logging**

When enabled via the *Enable log to:* setting, the analyzer automatically logs data to disk. The log file is capped at a maximum size of *File Size (MB)*. To allow for maximum performance while logging, the file log is retained in memory until shutdown, and then saved to disk. If disk logging is not enabled, logging occurs in memory and is lost upon shutdown.



## Reference

### *Alphabetic listing of all Steps & Triggers*

**T**he following reference section provides a detailed explanation of each of the steps and triggers provided within MULTIdrive™. Use this as a supplement to the information provide on-line.

## Transaction Steps

### Clear

A	B	C	D

Clears both the transmit and receive buffers to a known (empty) state. Clear is typically used in error situations where the state of the buffers must be cleared before continuing.

### End

A	B	C	D

Terminates or completes transaction processing when encountered. Transaction processing automatically ends when the last step is executed, but can be forced to end anywhere else in the transaction with this command. Usually an *End* command is used to complete normal communications, when followed by a block of error processing steps.

No parameters are required.

### Format CRC

A	B	C	D
Start Index	End Index	CRC Algorithm	

Computes and writes a CRC value on the end of the transmit buffer. The CRC value is computed from bytes (A) to (B) inclusive. A start index of 0 indicates the first byte in the buffer. An end index of 0 indicates the last byte in the buffer, wherever that byte may be. An end index of 1, 2, and 3 indicate the 2<sup>nd</sup>, 3<sup>rd</sup>, and 4<sup>th</sup> bytes in the buffer respectively. The typical use of these settings is (0,0) directing the step to compute the CRC across the entire buffer. A value of (1, 5) computes the CRC from bytes 1 to 5 (or the 2<sup>nd</sup> byte to the 6<sup>th</sup> byte).

The algorithm used to calculate the CRC is selected in (C). Custom CRC algorithms can be defined as long as the user knows the exact specifications of the CRC calculation.

The resulting CRC calculated will be placed at the end of the transmit buffer in the format designated by (D).

---

**Format Character**

A	B	C	D
Character			

Places a single ASCII character on to the transmit buffer. *Format Character* can be used to place any ASCII character on to the buffer. Typically *Format String* is used to define ASCII characters or strings while *Format Character* is used to place non-printable characters on to the buffer such as carriage return <CR>, line feed <LF>, start of text <SOT>, etc.

---

**Format Data (16)**

A	B	C	D
WORDS	Format		

Places a list of 16 bit WORDs (A) on to the transmit format buffer in format (B). This command is typically used in conjunction with binary protocols and allows the user to format WORD values directly on to the buffer. In instances where the device's protocol is defined using WORDs, this command makes defining the transaction easier and more readable.

The list of WORDs can be defined in binary, octal, decimal, or hexadecimal bases by adding the suffixes b, o, d and h respectively.

---

**Format Data (8)**

A	B	C	D
BYTEs			

Places a list of 8 bit BYTEs (A) on to the transmit buffer. This command is typically used in conjunction with binary protocols and allows the user to format BYTE values directly on to the buffer. In instances where the device's protocol is defined with individual BYTEs, this command makes defining the transaction easier and more readable.

---

**Format XOR Checksum (16)**

A	B	C	D

Start Index	End Index	Format	
-------------	-----------	--------	--

Computes and writes a 16 bit checksum on to the end of the transmit buffer. The checksum value is computed from bytes (A) to (B) inclusive. A start index of 0 indicates the first byte in the formatting buffer. An end index of 0 does not indicate the first byte, but rather denotes the last byte, whatever that index may be. An end index of 1, 2, and 3 indicates the 2<sup>nd</sup>, 3<sup>rd</sup>, and 4<sup>th</sup> bytes in the buffer respectively. The typical use of this settings is (0,0), indicating to compute the checksum across the entire buffer. A value of (1, 5) computes the checksum from bytes 1 to 5 (or the 2<sup>nd</sup> byte to the 6<sup>th</sup> byte).

Parameter C defines how the resultant 16 bit checksum will be placed on to the transmit buffer.

An XOR checksum is computed by XORing each successive 16 bit word. This approach varies slightly from a normal checksum that adds each successive 16 bit word.

For example, the XOR checksum of the two values ABCD and EF91 is 445C hexadecimal. If this were a standard (adding) checksum the result would be 9B5E hexadecimal.

**Caution:** (A) and (B) are specified by byte index, not word index.

### Jump

A	B	C	D
Label			

Immediately jumps to label (A). Step execution will resume at the step following the label (A).

An immediate jump is typically only used in situation where a transaction is divided up into smaller blocks of steps.

### Jump Bad XOR Checksum (16)

A	B	C	D
Start Index	End Index	Format	Label

Using the receive buffer, computes the 16 bit (XOR) checksum from (A) to (B) inclusive according to format (C). Should the computed checksum not equal the actual checksum located at (B)+1, the step jumps to (D).

The actual checksum must be located in the two bytes following (B).

An XOR checksum is computed by XORing each successive 16 bit word. This varies slightly from a normal checksum that adds each successive 16 bit word.

---

### Jump Equal (16)

A	B	C	D
WORD	Label		

Compares the user supplied word (A) to the current word in the receive buffer and jumps to label (B) if the values are equal.

**Caution:** The data format is not specified here, and therefore defaults to Intel format (i.e. low byte, high byte). An incoming byte stream of 09 08 would be interpreted as a value of 0809 hexadecimal (or 2057 decimal). If this is mis-understood, the value (A) may never match the incoming word value.

---

### Jump Equal (8)

A	B	C	D
BYTE	Label		

Compares byte (A) to the current byte in the receive buffer and jumps to label (B) if the values are equal.

Step execution resumes at the step following label (B).

---

### Jump Not Equal (16)

A	B	C	D
WORD	Label		

Compares a user supplied 16 bit word (A) to the current word in the receive buffer and jumps to label (B) if the values are not equal.

**Caution:** The data format is not specified here, and therefore defaults to Intel format (i.e. low byte, high byte). An incoming byte stream of 05 06 would be interpreted as a value of 0605 hexadecimal (or 1541 decimal). If this is mis-understood, the value (A) may never match the incoming word value.

**Jump Not Equal (8)**

A	B	C	D
BYTE	Label		

Compares a user supplied 8 bit byte (A) to the current byte in the receive buffer, and jumps to label (B) if the values are not equal.

**Label**

A	B	C	D
Label Name			

Performs no operation other than to label the step with the name found in (A). When this step is executed, it does nothing and simply continues on to the next step.

The label step provides a jump location for all steps requiring a label as an input parameter.

**Pause**

A	B	C	D
Time (ms)			

Pause step execution for (A) milliseconds. Used to introduce small delays between steps. The step may be used in situations where a device requires a specific delay between messages, retries, settings, etc.

The step essentially gives up CPU processing to other programs and will not affect PC performance. No other transactions will execute while a delay is being performed.

**Read Data (16)**

A	B	C	D
---	---	---	---

# of WORDs			
------------	--	--	--

Removes 16 bit word data from the receive buffer.

As the receive buffer is parsed and interpreted the current buffer position is moved ahead with each step. This command performs no function other than to move the buffer position ahead by (A) words, or (A) \* 2 bytes.

The command is used to skip over words that exist within the reply message, but are not be extracted for use.

In situations where the device's protocol is defined in WORDs, this step provides a more readable definition.

---

### Read Data (8)

A	B	C	D
# of BYTEs			

Removes (or reads) 8 bit byte data from the receive buffer.

As the receive buffer is parsed and interpreted the current buffer position is moved ahead with each step. This command performs no function other than to move the buffer position ahead by (A) bytes.

The command is used to skip over bytes that exist within the reply message, but are being ignored.

---

### Read OPC Item

A	B	C	D
OPC Item	Format		

Reads a value from the receive buffer in format (B) and places the value into OPC item (A). When complete, increments the current receive buffer position to the next byte following the byte(s) processed by this command.

When the specified input format B is of a discrete nature, the byte position will not be moved. This will allow multiple Read OPC Item steps to extract the necessary discrete information out of a single byte. When extracted, the user must issue a Read Data (8) to manually move the byte position to the next byte in the receive buffer. When reading all other types of formats (B), the position is automatically updated to the next byte in line.

**Receive Data**

A	B	C	D
# of BYTEs	Timeout	Label	

Receives (A) bytes from the device. If the bytes cannot be read before the timeout value (B) elapses, then step execution jumps to (C).

If (A) is set to 0, receives all the bytes currently available from the device.

Used to receive data from devices where a binary protocol is being used. If an ASCII protocol is being used, use *Receive String*

**Receive String**

A	B	C	D
Term Char.	Timeout (ms)	Label	

Receives a multi-character string, terminating reception after receiving character (A). If the time (B) elapses before character (A) is received, step execution jumps to label (C).

*Receive String* is used with ASCII devices to receive replies terminated with a defined character (e.g. carriage return <CR>, linefeed <LF>, end of text <EOT>, etc.).

**Retry**

A	B	C	D
# of Retries	Label		

Each time this step is executed jumps to label (B) up to a maximum of (A) times before falling through and executing the next step.

This step is similar to a Do-While or For-Next loop, whereby label (B) is executed (A) times.

Typically used in situations where a retry sequence is required for a user define maximum number of times

**Seek Byte**

A	B	C	D
BYTE	Label		

Searches through the receive buffer and re-positions the current position to the first byte that matches (A). If byte (A) is not found, jumps to label (B).

Used in conjunction with binary protocols to position the current receive position to a specific location in the buffer. If the location cannot be found, jumps to label (B).

When interpreting binary data, devices have a very defined response. This step can be used to locate a specific location for further processing incoming values. When the byte is found the resulting position is the byte itself and not the next position. Therefore a Read Data (8) will most likely be required to move the current position over the required data to be read.

### Seek String

A	B	C	D
String	Label		

Searches through the receive buffer and re-positions the current position to the string that matches (A). If the string (A) is not found, jumps to label (B). The string comparison is case sensitive.

Used in conjunction with ASCII protocols to move the current receive position to a specific location. For example, if a device returns “*The Temperature is 22 degrees*”, seeking to the string “*is*”, would place the receive position just before the value of 22. A following step command of Read OPC Item would then read the value 22.

### Set OPC Item

A	B	C	D
OPC Item	Value	Quality	

Sets the OPC item (A) to value (B) and quality (C). If the value (B) is omitted, only the quality is set.

An OPC item’s value is usually set via a response message from a device (via *Read OPC Item*). Set OPC Item is used in situations where the user must set, reset or change the value or quality of an item.

For example, if communications fails during a specific transaction due to a checksum/CRC failure, it may be necessary for the transaction to mark all OPC items as having a bad value (or bad quality).

Another example may be to preset all items to a known state upon startup.

---

**Transmit Data**

A	B	C	D

Transmits all the bytes or characters currently in the transmit buffer. If there are no flow control options enabled, transmits the data immediately. If flow control options are enabled, the transmit data may not be transmitted immediately.

There are no other settings for this command.

## Transaction Triggers

---

### Break Signal

A	B	C

The Break Signal trigger executes its transaction whenever a break signal is received on the input of the open serial port.

If multiple transactions are configured to use this trigger, each transaction will be guaranteed to execute, however the order of their execution is not defined.

---

### Carrier Detect

A	B	C
Level		

Triggers its transaction to execute if the Carrier Detect signal (CD) changes state to (A). The transaction is only executed once per state change of the CD pin.

If multiple transactions are configured to use this trigger, each transaction will be guaranteed to execute, however the order of their execution is not defined.

---

### Clear to Send

A	B	C
Level		

Triggers its transaction to execute if the Clear-to-Send signal (CTS) changes state to (A). The transaction is only executed once per state change of the CTS pin.

If multiple transactions are configured to use this trigger, each transaction will be guaranteed to execute, however the order of their execution is not defined.

**Data Set Ready**

A	B	C
Level		

Triggers its transaction to execute each time the Data-Set-Ready (DSR) pin changes to state (A).

If multiple transactions are configured to used this trigger, each transaction will be guaranteed to execute, however the order of their execution is not defined.

**Error Detected**

A	B	C

Triggers its transaction to execute if a line status error is detected. A line status error includes a parity error (if enabled), a buffer overrun or a framing error when assembling an incoming byte. The transaction is executed one per error detected.

If multiple transactions are configured to used this trigger, each transaction will be guaranteed to execute, however the order of their execution is not defined.

**Event Character**

A	B	C
Character		

Triggers its transaction to execute each time character (A) is received. The trigger can be used in situations where an unsolicited communication messages may be sent from the device.

Due to operating system limitations, only one Event Character trigger can be used within the entire MULTIdrive™ Definition File (MDF). Using this trigger more than once, may produce unpredictable results.

When a transaction executes with this trigger, the event character (A) may not be the first character in the receive buffer. Using the Seek Byte step will assist in locating the event character.

**OPC Item Read**

A	B	C
OPC Item		

Triggers its transaction to execute each time OPC item (A) has been requested to read its value from an attached OPC client. If the client requests a cached value, this transaction is not executed and the item's current value is simply returned.

The trigger has both pros and cons when being used to trigger read operations. A pro is that attached clients have better control over what communications take place and can optimize thru-put by controlling when the item is read. A con is that if multiple clients continually read the same item, wasted messages will be sent as each read request will result in this transaction executing. If this were the case, using the *Timer* trigger would yield better performance than this trigger. Only the user will know which approach is best suited to the application.

---

### OPC Item Update

A	B	C
Item		

Triggers its transaction to execute each time OPC Item (A) has been requested to read its value from a request generated internally by MULTIdrive™. This request occurs when an attached OPC client has placed item (A) into an OPC group list, and has specified an overall update rate for the group. MULTIdrive™ automatically scans all the items in the group and only informs the client of changes.

MULTIdrive™ optimizes this trigger by only trigger the transaction when absolutely necessary. For example, if 25 attached OPC clients each have item (A) placed into their own group, updating at rates between 100 and 900 milliseconds, the transaction will only trigger once every 100 milliseconds satisfy the fastest request. All other clients will simply be provided a cached value, which of course is only 100 milliseconds old.

When a transaction is defined that reads OPC Item (A), usually both an *OPC Item Update* trigger and an *OPC Item Read* trigger are added. This way the transaction is executed regardless of how the read request came in. It is important to note that if only one of these triggers is defined, it is possible that some OPC clients will work properly, while others may not. This is a direct result of how they request their values (i.e. item read vs. group update).

Most OPC Client applications rely heavily on the OPC server scanning groups and sending changes, rather than reading items individually. MULTIdrive™ is optimized for this situation and will perform the minimum amount of device communications to satisfy all clients.

---

### OPC Item Write

A	B	C
---	---	---

Item		
------	--	--

Triggers its transaction to execute each time a write request has been received for OPC Item (A). A single write occurs for each request received.

Currently, any OPC client can perform a write operation and there is no policing over which clients have read/write access and which only have read access.

**Receive Character**

A	B	C

Triggers its transaction to execute each time any character is received. The trigger can be used in situations where unsolicited messages are received without any requests. It is up to the transaction to make sense out of whatever character was received and parse the incoming message accordingly.

**Ring**

A	B	C
Level		

Triggers its transaction to execute if the *Ringsignal* changes to level (A). The transaction is only executed once per change of the ring status input to level (A). Under normal circumstances, only modems use this input to indicate the phone is ringing. Of course this may not be the case for all devices.

If multiple transactions are configured to used this trigger, each transaction will be guaranteed to execute, however the order of their execution is not defined.

**Shutdown**

A	B	C
Priority		

Triggers its transaction to execute each time the MULTIdrive™ OPC Server is shutdown. This includes each time the software is exited or shutdown, as well as each time a **File Save** operation is performed and the old runtime is shutdown prior to the new one being initialized.

The priority value (A) helps define the order of execution upon shutdown. For example, those triggers with a lower value of (A) defined (e.g. 0, 1, 2...) will execute before those triggers with a priority of (3, 4, 5... etc.).

A possible use of this trigger is to perform some sort of log-out command upon exiting.

---

## Startup

A	B	C
Priority		

Triggers its transaction to execute each time the MULTIdrive™ OPC Server is started. This includes each time the software is started, as well as each time the **File Save** operation is performed resulting in a new runtime being initialized.

The priority value (A), helps define the order of execution upon startup. For example those triggers with a lower value of (A) 0, 1, 2 will be executed before those triggers with a higher value of (A) 3, 4, 5.

This trigger is used in situations where it is necessary to log into a device upon startup, reset it, or otherwise establish communications. It may also be used to preset values within the server to a known value.

---

## Timer

A	B	C
Period (ms)		

Triggers its transaction to execute periodically every (A) milliseconds. This is the most popular trigger used, and is used to periodically scan a device for updates.

Rather than waiting for an incoming read/update request from an attached OPC client, this trigger can be used to read a device based upon a defined update rate.

In some situations where many OPC clients are attached, it may be advantageous to read the device periodically and simply provide the clients with cached values. This way, the number of attached clients, or their requests will not effect the communications performed to the end device.

However, if performance is of paramount importance, using the trigger *OPC Item Read* and *OPC Item Update* may provide a higher overall thru-put.

---

**Tx Buffer Empty**

A	B	C

Triggers its transaction to execute each time the transmit buffer runs out of characters to transmit. Transmission of data occurs everytime the *Transmit Data* step is performed. This trigger would fire sometime after this step, when all the data has been transmitted.

One possible use of this trigger maybe to provide some form of message termination or signaling after a message has been sent. For example, some devices may require signaling of DTR or CTS to inform the device they are done transmitting, waiting for a reply, etc.

# Glossary

## **OPC**

OLE for Process Control

## **OPC Client**

A COM/DCOM client application with the ability to connect to an OPC server application, such as MULTIdrive™.

## **OPC Group**

Within MULTIdrive, an OPC Group is a parent item containing one or more child items. Here a group simply formats the display shown to any connected clients. An OPC Group for a OPC client application is a collection of any items within the server that can be controlled together.

## **OPC Item**

A single database point that lives within the MULTIdrive™ OPC Server and reflects a value within an industrial device/sensor.

## **OPC Server**

A COM/DCOM server application that provides the necessary methods as defined by the OPC Specification.

## **Step**

A single operation or step within a transaction that is performed each time the transaction is executed. A Transaction Step can perform many operations include sending/receiving data, formatting inputs/outputs, controlling step execution, etc.

## **Transaction**

A complete cycle of (1) format transmission message, transmit message, receive reply message, and extract receive information. Each transaction is executed independently of the other.

## **Trigger**

A trigger (or transaction trigger) defines when its parent transaction executes. There are various types of triggers all of which may be defined for a transaction. If a transaction has no triggers defined, the transaction will never execute.



# Index

## I

Item. *See* OPC Item

## M

Modbus Protocol, 28

## O

OPC Client, 59  
OPC Group, 7, 59  
OPC Item, 59  
OPC Items, 6  
OPC Server, 59

## S

Serial Analyzer, 12  
Steps  
  Clear, 35, 44  
  End, 15, 35, 44  
  Format Character, 45  
  Format CRC, 34, 44  
  Format Data (16), 45

Format Data (8), 34, 45  
Format XOR Checksum (16), 46  
Jump, 46  
  Jump Bad CRC, 34  
  Jump Bad XOR Checksum (16), 46  
  Jump Equal (16), 47  
  Jump Equal (8), 47  
  Jump Not Equal, 34  
  Jump Not Equal (16), 47  
  Jump Not Equal (8), 48  
Label, 15, 34, 48  
Pause, 48  
Read Data (16), 49  
Read Data (8), 49  
Read OPC Item, 17, 35, 49  
Receive Data, 34, 50  
Receive String, 50  
Retry, 15, 35, 50  
Seek Byte, 51  
Seek String, 51  
Set OPC Item, 15, 19, 35, 51  
Transmit Data, 34, 52

## T

Tag. *See* OPC Item  
Transaction, 59  
Transaction Step, 59  
Transaction Trigger, 59  
Transactions, 7  
*Triggers. See* Transaction Trigger  
  Break Signal, 53  
  Carrier Detect, 53  
  Clear to Send, 53  
  Data Set Ready, 54  
  Error Detected, 54  
  Event Character, 54  
  OPC Item Read, 17, 55  
  OPC Item Update, 17, 55  
  OPC Item Write, 18, 56  
  Receive Character, 56  
  Ring, 56  
  Shutdown, 57  
  Startup, 57  
  Timer, 57  
  Tx Buffer Empty, 58