U-CON (User-Configurable) Driver Help

Table of Contents

U-CON (User-Configurable) Driver Help

Help version 1.078

CONTENTS

Overview

The U-CON (User-Configurable) Driver provides an easy and reliable way to connect U-CON (User-Configurable) Ethernet and Serial Devices to OPC Client applications, including HMI, SCADA, Historian, MES, ERP and countless custom applications. While other drivers are designed specifically for use with a single device type, or a small family of closely related devices, the U-CON (User-Configurable) Driver can be programmed to work with a very wide variety of serial and Ethernet devices. Driver profiles are created using the integrated Transaction Editor. Transaction elements are selected from context aware menus, thus eliminating the need to learn scripting languages and greatly reducing the possibility of errors.

Features

The U-CON (User-Configurable) Driver is completely integrated with the server. Custom drivers can be developed, debugged, and run from the server itself. Such tight integration with the server ensures that all of the important features users demand from other drivers are available to the custom driver projects. These features include full OPC 1.0 and 2.0 compliance, DDE support, tag browsing, automatic tag database generation, diagnostics and event logging. The server may also be configured to run as a Windows NT/2000/XP service.

The server's Ethernet Encapsulation feature is supported and may be used in solicited or unsolicited mode. This feature is used to communicate with serial devices connected to a terminal server such as the Digi One RealPort or the Lantronix CoBox over an Ethernet network. It also is used to develop driver profiles for native Ethernet devices.

Like any other serial driver for the server, custom driver projects will have modem support, communication port configuration and standard error handling features with configurable retries and timeouts. Furthermore, the server's built in diagnostics display is used to easily diagnose communications problems during driver profile development.

The U-CON (User-Configurable) Driver is based upon the same technology found in every other driver available for the server. With the U-CON (User-Configurable) Driver, users get all of the benefits of a true multi-threaded 32-bit environment without the need to learn the intricacies of Microsoft Windows development.

System Requirements

Operating System
Windows NT
Windows 2000 (recommended)

Windows XP (recommended)

Intel Pentium class Processor
200 MHz (minimum)
400 MHz or better recommended

Memory
32 MB (minimum)
64 MB or better recommended

Recommended (For Driver Development)
VGA monitor
Mouse

Protocol Requirements

Development of a driver profile requires access to the protocol documentation of the target device; thus, a basic understanding of device communications is highly recommended.

Engineering services

Custom enhancements and driver configuration services are available. Contact Technical Support for details.

Demo Mode

An unlicensed copy of this driver may be used for evaluation purposes. If a profile is being edited while the demo period expires, a chance will be given to save the changes made to the work. Any new tags or tag groups created since the last time the server was updated will not be visible in the server at this point. Save the server project. The next time the project is opened, the new tags and groups will appear.

Device Setup

Supported Devices

The U-CON (User-Configurable) Driver can be configured to work with a wide range of serial devices.

Communication Protocol

Most protocols can be accommodated.

Supported Communication Parameters*

Baud Rate: 300, 600, 1200, 2400, 9600, 19200 or 38400
Parity: None, Even or Odd
Data Bits: 5, 6, 7 or 8
Stop Bits: 1 or 2

*Not all devices support the listed configurations.

Device IDs

This driver can support any Device ID from 0 to 255.

Note: Not all devices recognize this entire range.

Ethernet Encapsulation

This driver supports Ethernet Encapsulation, in both solicited and unsolicited modes. Ethernet Encapsulation allows the driver to communicate with serial devices attached to an Ethernet network using a terminal server like the Lantronix DR1. Ethernet Encapsulation mode is invoked by selecting it from the COM ID dialog in Channel Properties. For more information, refer to the OPC Server's help documentation.

Flow Control

When using an RS232/RS485 converter, the type of flow control that is required will depend upon the needs of the converter. Some converters do not require any flow control and others will require RTS flow. Consult the converter's documentation in order determine its flow requirements. We recommend using an RS485 converted that provides automatic flow control.

Note: When using the manufacturer's supplied communications cable, it is sometimes necessary to choose a flow

control setting of **RTS** or **RTS Always** under the Channel Properties.

Inter-Request Delay

This option is used to limit how often requests are sent to a device. It will override the normal polling frequency of tags associated with the device, as well as one-shot reads and writes. Delays will not be used if the channel is in unsolicited mode. This delay can be useful when dealing with devices with slow turnaround times and in cases where network load is a concern. Be aware that configuring a delay for a device will affect communications with all other devices on the channel. Because of this, it is recommended that any device that requires an inter-request delay be segregated to a separate channel if possible. Users may set the inter-request delay from 0 to 300000 ms (5 minutes). The default setting of 0 disables this feature. **See Also:** Defining a Server Channel.

Cable Connections

Cable connections depend on the specific device being used.

Modem Setup

This driver supports modem functionality. For more information, please refer to the topic "Modem Support" in the OPC Server Help documentation.

Unsolicited Message Wait Time

The **Wait Time** parameter is used to specify the time (in milliseconds) that the device should wait for unsolicited messages before the _UnsolicitedPcktRcvdOnTime system tag is set to 1. The _UnsolicitedPcktRcvdOnTime tag, which is displayed by the client application, indicates whether or not an unsolicited message has been received for a given device within the amount of time that was specified in the Wait Time field.

In the client application, check the _UnsolicitedPcktRcvdOnTime tag's Value field:

- If the Value field displays 0 (zero), the message was received within the Wait Time amount.
- If the Value field displays 1, the message was not received within the Wait Time amount.
- For solicited communications, the _UnsolicitedPcktRcvdOnTime tag will always display 1 and can be ignored.

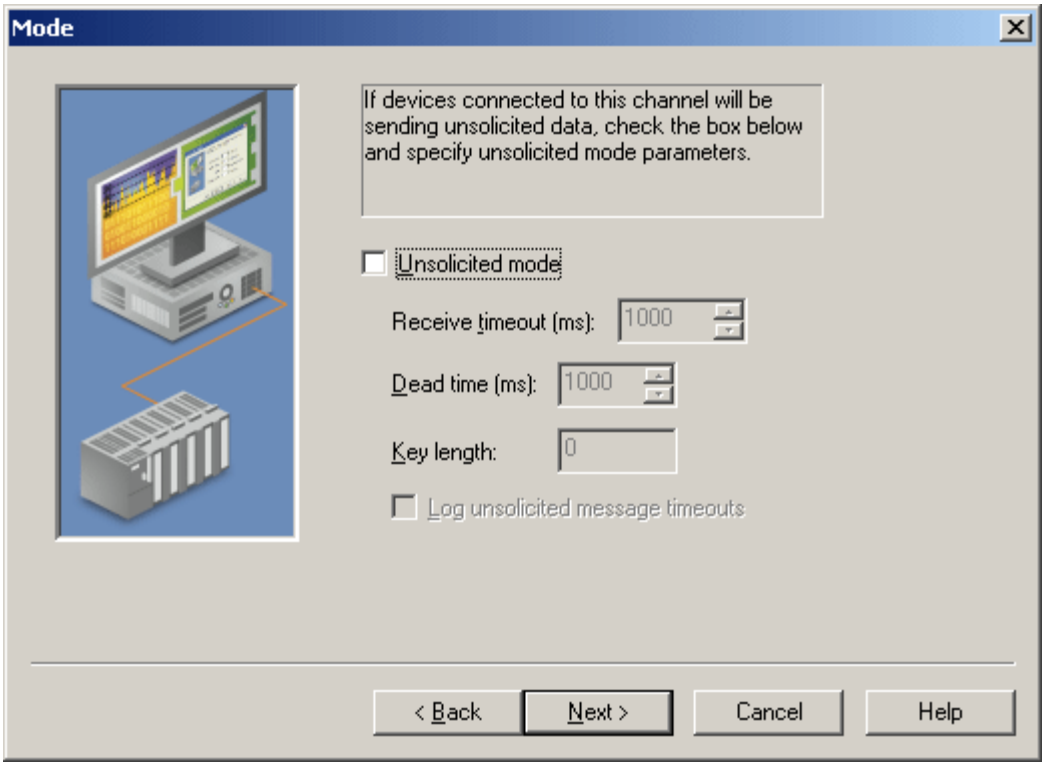
Driver Configuration

There are four steps required to configure the U-CON (User-Configurable) Driver Driver. Users must define a server channel, define a server device, define a device profile and then test and debug the configuration. Although the first two steps are relatively simple, the final two steps will most likely require a significant amount of effort and attention.

See Also:
Defining a Server Channel
Defining a Server Device
Defining a Device Profile
Test and Debug the Configuration

Defining a Server Channel - Step 1

The first step in creating any server project is to define a channel in the server. Many devices can be connected to a single channel as long as they can all use the same protocol and Driver. In this case, run the server's **Channel Creation Wizard** and choose **U-CON (User-Configurable) Driver** from the list of installed drivers. Next, specify the various communication parameters (such a baud rate, parity, number of data bits and etc.) that are required by the device(s) to be specified. In the final dialog, the U-CON (User-Configurable) Driver's mode is specified. This dialog should appear as shown below.



The **U-CON (User-Configurable) Driver** can operate in two modes: **normal** and **unsolicited**. By default, the driver will set itself up in normal mode. In normal mode, the driver will request data from each device periodically (up to 100 or more times per second per tag). The driver ignores all data that is not in response to a request. In unsolicited mode, the driver does not request data from the device. Instead, it waits for data to come in from the device. The device determines which mode will be chosen. Some devices are designed to work in unsolicited mode such as scanners and scales, while others only supply data when it is requested such as most controllers and PLCs. Once the driver's channel mode is set, it cannot be changed: any driver configuration beyond this point will likely be incompatible with the new mode of operation.

Note 1: It is necessary to segregate all devices that issue unsolicited data to one or more channels that are specific for unsolicited communication.

Note 2: If using **Ethernet encapsulation**, be sure to configure its mode of operation to match this setting. For more

information on Ethernet encapsulation, refer to the Server's help documentation.

If selecting unsolicited mode, three additional parameters must also be set: **Receive timeout**, **Dead time** and **Key length**. Before setting these parameters, it should be noted how the driver handles unsolicited data. Upon receipt of an unsolicited message, the driver must determine what user defined transaction should be used to interpret the message. To make this possible, the user must associate each transaction definition with some property unique to messages of a given type. For example, a device could report changes in input 1 as IN01xxxx where xxxx is a 4-byte value, and changes in input 2 as IN02xxxx. In this case, IN01 would tell the driver to use one transaction that updates an Input_1 tag, and IN02 would tell it to use another transaction that updates an Input_2 tag. The driver can lookup the appropriate transaction using the first four bytes of any message from this particular device as keys. If the protocol does not lend itself to the use of such keys, it is still possible to use this driver by specifying a **zero key length**.

- The **Receive Timeout** is the amount of time that the driver should wait to receive the full, unsolicited message. If a full message has not been received by this time (either due to a hardware problem or an incorrectly defined Read Response command) the driver will assume that the next received character is the start of another message.
- The **Dead Time** is necessary so that the driver may re-synchronize itself with the device(s) after receiving a message with an unknown key. If a message is unrecognizable, the driver will not know where that message ends and the next one begins. The way the driver handles this situation is to let the entire unrecognized message come in and will then wait for some period of time. This dead time must be such that it is safe to assume that the next byte received is the beginning of another message. Reasonable values depend upon the target device and should be as small as possible, but longer than the typical time between bytes in a message. The time, in milliseconds, between bytes in a message is **approximately** 8000/baud rate. Since the dead time period is started each time a byte is received, make sure not to define too large a value: the driver would see individual messages as a single unrecognizable stream.
- The **Transaction key length** tells the driver how many characters to use as transaction keys. These characters must be the first characters in a message. In the example above, this value would be 4. The protocol(s) used on a given channel must be such that keys of the same length can uniquely identify all possible messages. The key length may be between 0 and 32 characters.

  In cases where the protocol does not permit the use of such keys, the driver can still be used. A scanner that sends packets starting with the raw data values would be an example. In these cases, the transaction key length must be set to zero. This will force the driver to use the first unsolicited transaction defined on the channel to interpret all incoming packets. Because of this, there should be only one device on the channel. Furthermore, that device should have a single block tag or a single non-block tag defined. That tag or tag block may be placed in a group.

  Note: For more information about unsolicited transactions and transaction keys, refer to **Unsolicited Transactions**.

- The **Log unsolicited message timeouts** setting can be useful for diagnosing communications problems. When checked, a message is placed in the event log each time the Receive timeout period expires while receiving an unsolicited message. Such events may be caused by data delays due to network traffic or gateway devices, incorrectly configured transactions, or **Pause** commands in the transactions.

Important: It is generally necessary to place devices that use different protocols on separate channels. It is possible to mix protocols on an unsolicited channel, as long as the transaction keys can be the same length and are unique. Remember also that the channel mode cannot be changed after the channel has been defined. This precaution is necessary since any transactions that have previously been defined would likely be incompatible with the new mode. Make sure that the channel mode being selected is the correct one. Finally, users must not mix devices that send unsolicited data with those that do not on the same channel.

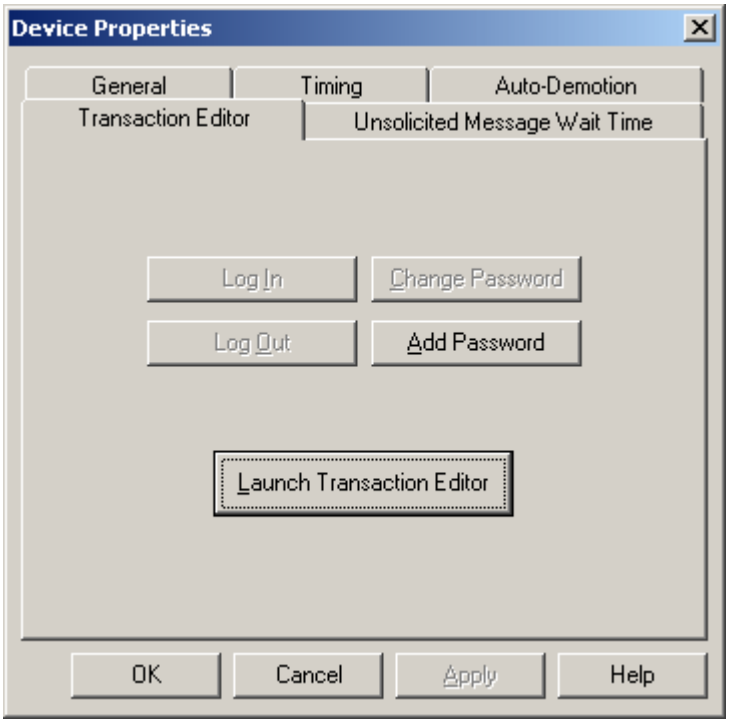
Defining a Server Device - Step 2

Next, a device must be defined. For more information, refer to the server documentation. When asked to set the Device ID, the number will only have meaning if the transactions use **Write Device ID** or **Test Device ID** commands.

Defining a Device Profile - Step 3

The U-CON (User-Configurable) Driver requires the user to define a device profile for each target device. A device profile includes a definition of each tag that the driver will serve as well as the sequence of commands necessary to carry out Read and Write requests for each tag. This work is done using the Transaction Editor, which is the graphical user interface of the U-CON (User-Configurable) Driver.

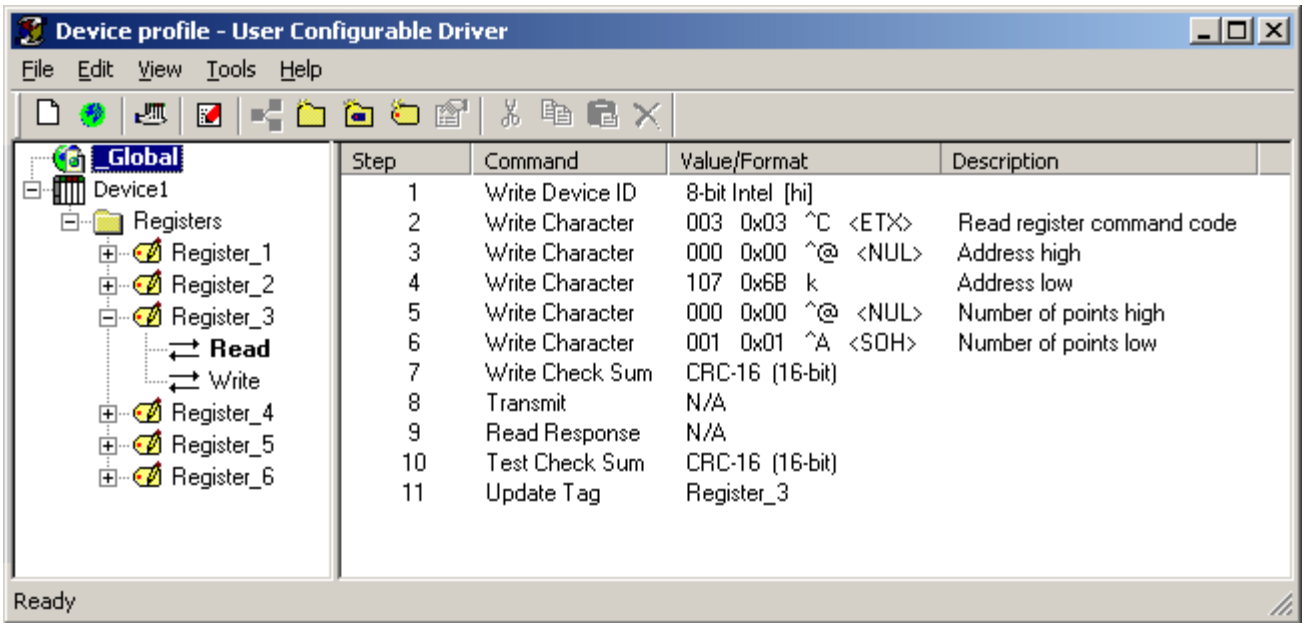
To invoke the Transaction Editor , double-click on the device and then select the **Transaction Editor** tab. Next, click **Launch Transaction Editor**.



Note 1: The Transaction Editor can not be started if the device is in use. Before accessing, disconnect all client applications.

Note 2: The device profile may be password protected. For more information, refer to **Password Protection**.

Note 3: The Transaction Editor can be used to construct groups of tags and transaction command sequences. Its user-defined profile is shown below.



For detailed information on how to define a driver profile, refer to **Transaction Editor**.

Once a device profile has been created, the tag and transaction definitions can be sent to the server by clicking **Update Server** on the toolbar or main menu. If the Transaction Editor is closed, users will be given the chance to update the server. The tags and groups previously defined with the Transaction Editor will automatically be generated in the server. Remember, the changes have not been saved to file at this point: save the server project every time one of the device profiles is edited. Device profiles are an extension of the standard server project and are saved as part of the server project file (.opf).

At this point, the driver project may be used. Once a driver profile has been created and loaded, the U-CON (User-Configurable) Driver should work just like any other driver plug-in for the server. Changes are made to the profile at any time by disconnecting the device from all client applications and then invoking the Transaction Editor. Remember to save the project in between edit sessions.

Testing and Debugging the Configuration - Step 4

Once a device profile has been created using the Transaction Editor, it should be tested. To do so, first connect the device(s) and client application and make sure that the data can be read and written correctly. If there are any problems, refer to the servers built in **Diagnostics Window**, which can be a very useful tool in debugging the profile. For more information on debugging, refer to **Tips and Tricks**.

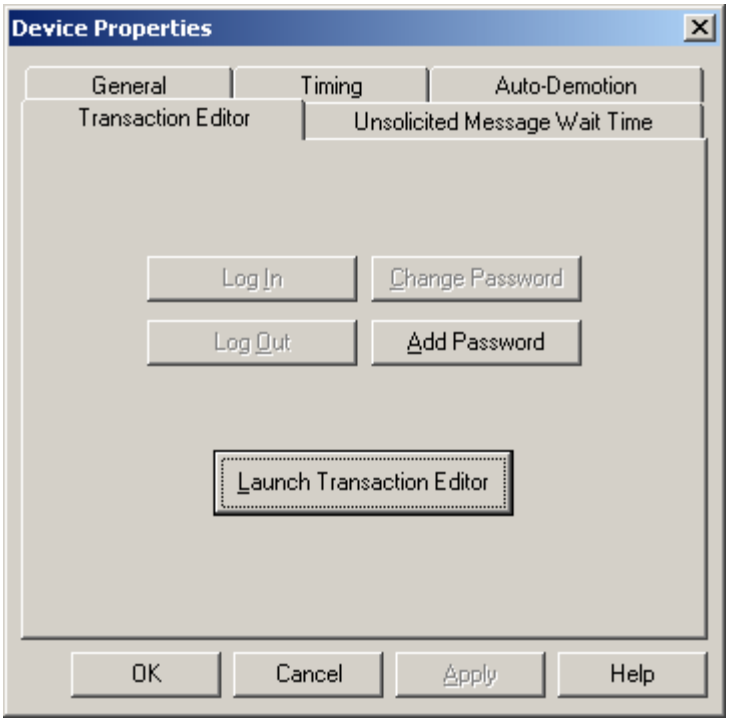
Caution: Although the U-CON (User-Configurable) Driver's run-time processor makes every reasonable effort to trap error conditions, it is still possible for certain poorly defined configurations to cause a driver failure. For this reason, development work should be completed on an isolated system (if possible) and the project should be tested thoroughly before going live. It is important to save work frequently.

Password Protection

Device profiles have the option of being password protected. Password protection prevents unauthorized users from launching the transaction editor and examining or modifying the profile. Each device profile can have its own password.

Note: This feature is not the same as the OPC Server's User Manager tool.

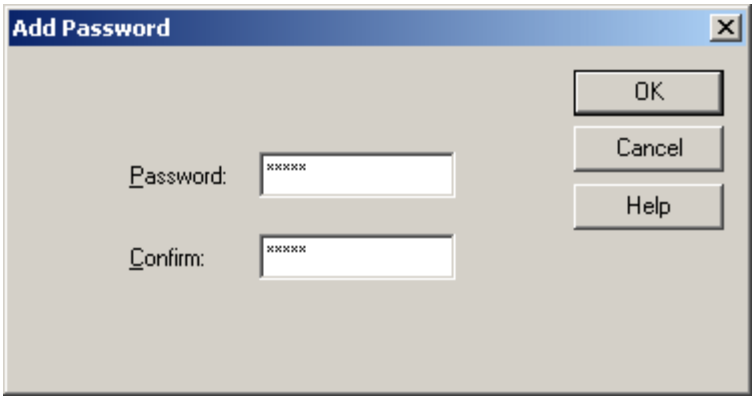
To enable password protection for a device, select the **Transaction Editor** tab in Device Properties.



Add Password

If the device does not currently have a password associated with it, the **Add Password** button will be enabled. Click

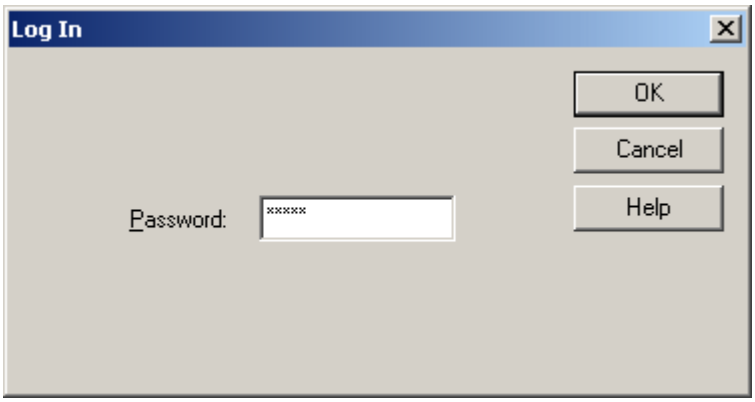
this button to invoke the Add Password dialog. The server project must be saved after a password has been added.



Note: The new password must be entered twice. Passwords are not case-sensitive and may be up to 15 characters long.

Log In

Click this button in order to enter the password.

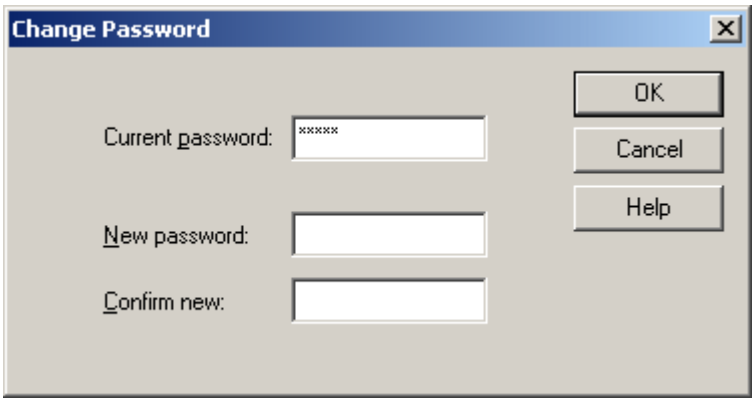


Log Out

Click this button in order to log out.

Change Password

Click this button in order to change or remove a password.



Users will be required to enter the current password and the new password twice. To disable password protection, simply leave **New password** and **Confirm new** blank. Passwords are not case-sensitive and may be up to 15

characters long. Users must save the server project after a password is changed.

Launch Transaction Editor

Click this button to launch the Transaction Editor. This button will be disabled if password protection for the device has been enabled and a user has not successfully logged in.

Transaction Editor

A transaction is a list of simple actions needed to Read data from or Write data to a device. Transactions come in three varieties: **Read**, **Write**, and **unsolicited**.

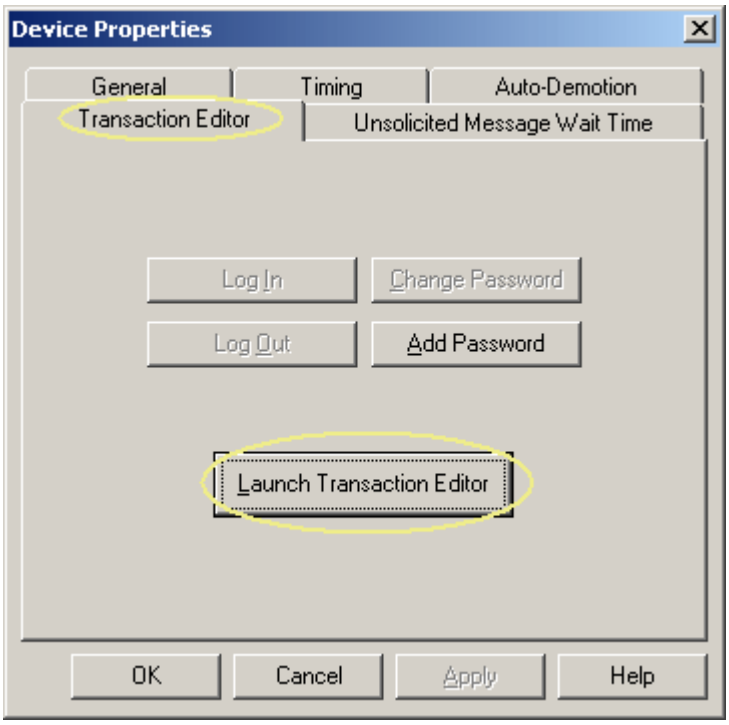
- **Read transactions** normally consist of a series of Write commands that build up a Read request string, a **Transmit** command that triggers the transmission of the request to the device, a **Read Response** command which waits for the expected response from the device, and an **Update Tag** command which parses and reformats the desired data from the response and updates the tag's value. A Read response may also employ other commands, including conditionals as the application dictates.
- **Write transactions** normally consist of a series of Write commands that build up a Write request string, a **Transmit** command, and possibly a Read response and **Update Tag** command. One of the Write commands will almost always be a **Write Data** command that takes the desired Write value from the client application and reformats it as required by the device.
- **Unsolicited transactions** are related to Read transactions, except that the first executable command must be a **Read Response** command. Each unsolicited transaction must also have a **Transaction Key** defined which will help the driver recognize what transaction should process a given message. When the driver is in unsolicited mode, it can only have Write and unsolicited transactions. In normal mode, it can only have Read and Write transactions. For more information on unsolicited transactions, refer to **Unsolicited Transactions**.

Transaction Editor

Normally, a driver is developed for a specific device type or family of closely related devices. The various transaction steps are programmed directly into the driver which allows users to simply select a driver and go. The U-CON (User-Configurable) Driver fills the need for a non-specific driver that can be used to communicate with a large number of devices for which targeted drivers have not yet been developed. It is up to the user to define the transactions necessary to communicate with the device. This work is done using the integrated Transaction Editor application.

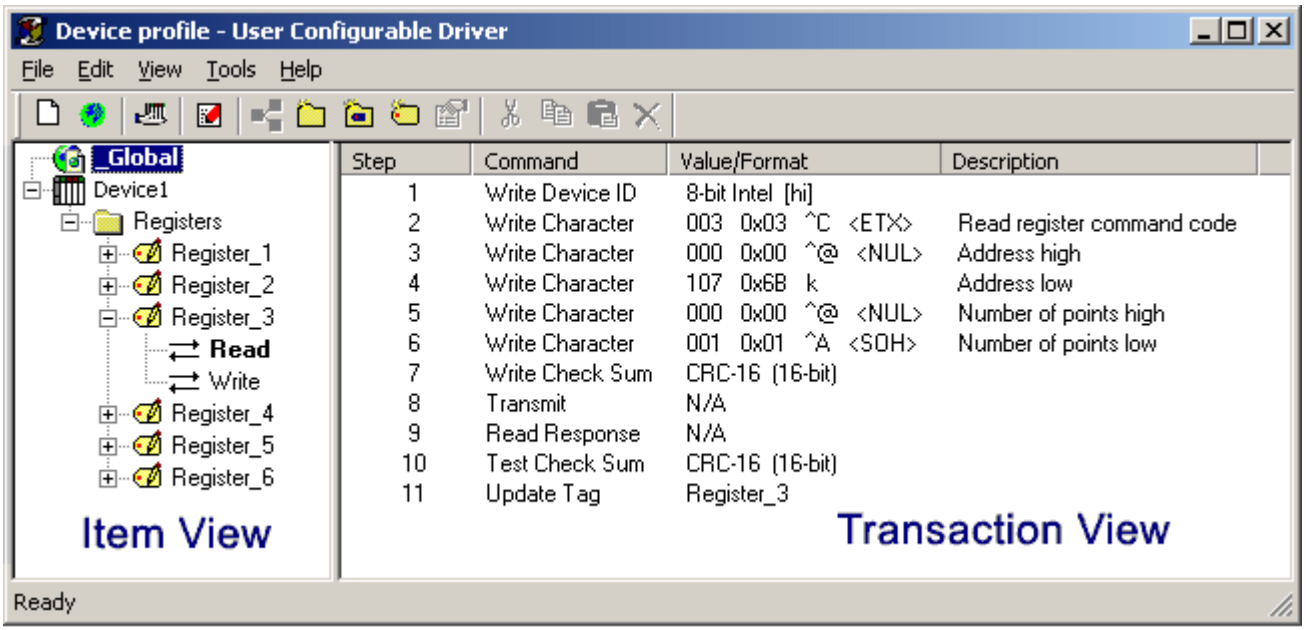
The Transaction Editor is the integrated development environment of the U-CON (User-Configurable) Driver. It provides an easy and intuitive means for configuring the driver. Tags, groupings of tags, and transactions are constructed using contextual pop-up menus. This graphical user interface approach eliminates the need to learn a driver programming or scripting language, and provides a degree of error prevention. All that is needed is an understanding of the particular device protocol in question.

In the OPC server main window, right-click on the desired device and then select **Properties**. Select the **Transaction Editor tab**, then click **Launch Transaction Editor**. The Device Properties dialog should appear as shown below.



In the Transaction Editor main window, a **Device Profile** can be both created and modified. A Device Profile refers to tags' groupings and transactions and may be password protected. For more information, refer to **Password Protection**.

As shown in the image below, the left pane shows the **Item View** and the right pane shows the **Transaction View**.



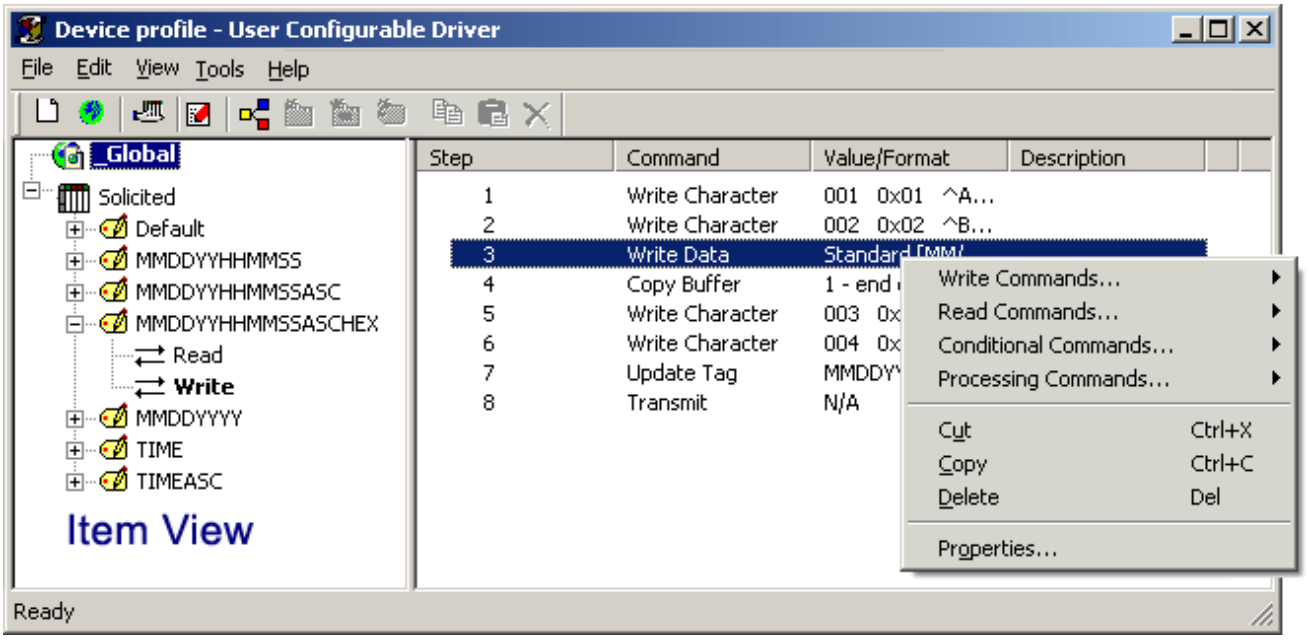
- The **Item View** displays the hierarchy of OPC items attached to a particular device. The fundamental item type is the tag. Associated with each tag are one or more transactions (represented by "to" and "from" arrow icons). These transactions can be for solicited Reads, Solicited Writes or Unsolicited Reads, and are created automatically whenever a tag is defined. Tags may be attached to the device, placed in tag groups (represented by plain folder icons) or in tag blocks (represented by folders with tags). A tag block is a special kind of group where all the contained tags are updated at once with a single Read or unsolicited transaction common to the block. Block Reads are much more efficient than the functionally equivalent series of individual Reads and should be used

whenever possible.

- When a transaction is selected in the item view, the **Transaction View** displays the currently defined sequence of commands that are to take place. When something other than a transaction is selected in the item view, the Transaction View is blank.
- The **Edit Option** at the top of the screen includes options for adding items, as well as options to cut, paste, delete or show the selected item's properties. The menu options commonly used are also represented on the toolbar for quick access.

Adding and Modifying Transactions in the Transaction View

Right-click in the Transaction View to invoke a submenu that provides access to all the available **transaction commands**.The dialog should appear as shown below.



For users without a mouse, individual commands can be selected from the Edit submenus with "**alt-character**" combinations.

Updating the OPC Server with the Device Profile

Once all of the groups, tags and transactions have been defined, the device profile must be sent to the server. This is initiated by clicking on the **Update Server** icon or by selecting **File**| **Update Server** from the main menu. The Transaction Editor also provides a chance upon its closing.

After the device profile has been transferred, the Transaction Editor will shut itself down and the driver will automatically initiate the OPC server's auto tag database generation function. All of the tags that have been defined will instantly appear in the OPC server project.

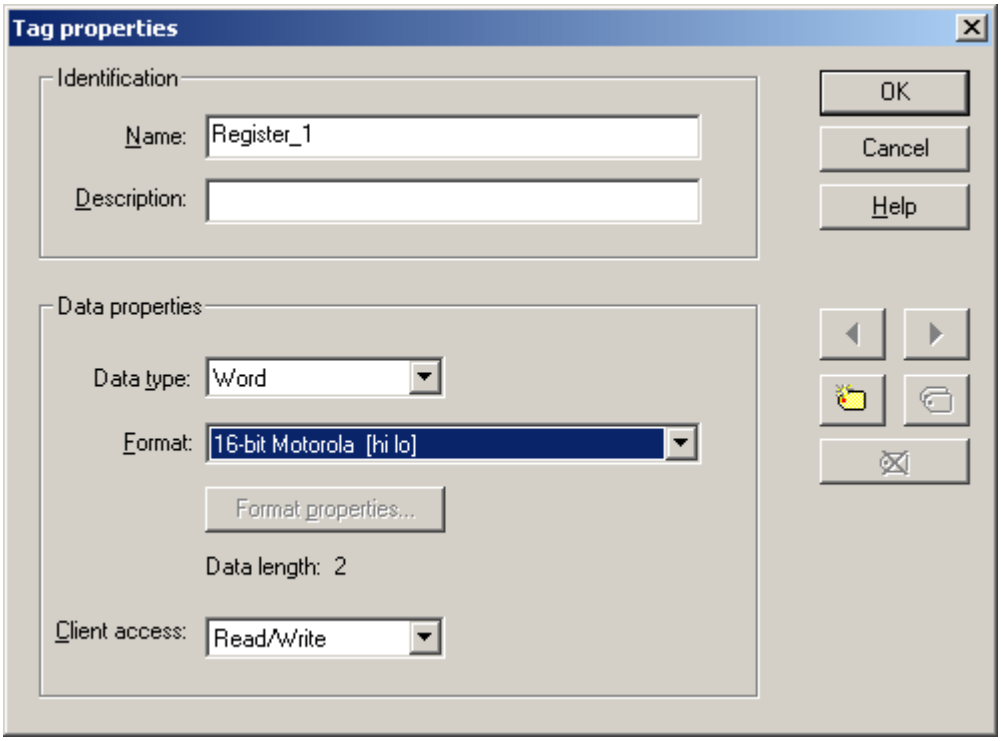
Note: At this point, the changes have not been saved to file. Click **File**| **Save** to save. Remember to save the OPC server project after each edit session.

Further Information

Click on any of the following links to learn more about the main help pages for the Transaction Editor.

Tags

A tag item can be added to the device, a tag group or a tag block. A tag can be added using the main menu, an item's pop-up menu or the toolbar. To edit an existing tag, users can either double-click on it or select it and then access **Properties** from the main menu. Alternatively, users can utilize the tag's pop-up menu or the toolbar. The Tag Properties dialog should appear as shown below.



- The **Name** must be set first. If the tag is new, the driver will offer a valid default name that can be changed to any valid name. **Valid names** must start with a letter, consist of only letters, digits and underscores, be less than 32 characters long and be unique to the parent device, group or tag block.
- The **Description** is an optional string that will be displayed along with the tag in the server. It serves no function other than to provide the user additional information about the tag.
- The **Data Type** is the representation of the data when it is exchanged between the server and client applications.

The U-CON (User-Configurable) Driver allows any one of the basic data types to be chosen, although the one that best suits the expected range of data values should be chosen.

- The **Format Property** determines the representation of the data as it is exchanged between the server and device. Some formats, such as ASCII Integer, ASCII Real and ASCII String, require additional properties to be set. When this is the case, the **Format Properties** button will be enabled. The format determines how many data bytes will be transferred between the server and device and is shown for reference below the Format Properties button.

  Note: Whenever the format selection is changed, the user defined Format Properties, if any, will be reset to default values appropriate for the format. Always check these settings when available. For more information on formats, refer to **Device Data Formats**.

By default, a tag is set with Read/Write access, although it can be changed to Read Only by using the drop list box at the bottom of the dialog. The tag will be created with all necessary transactions. Users must, however, define the sequence of commands necessary to carry out each transaction. The access permission can be at any time during an edit session; however, when changing from Read/Write to Read Only, all commands defined for the write transaction will be permanently lost.

Note: Users can create a Write Only tag by selecting Read/Write access and leaving the read transaction empty. In unsolicited mode, tags are created with an unsolicited transaction instead of a Read. For more information, refer to **Unsolicited Transactions**.

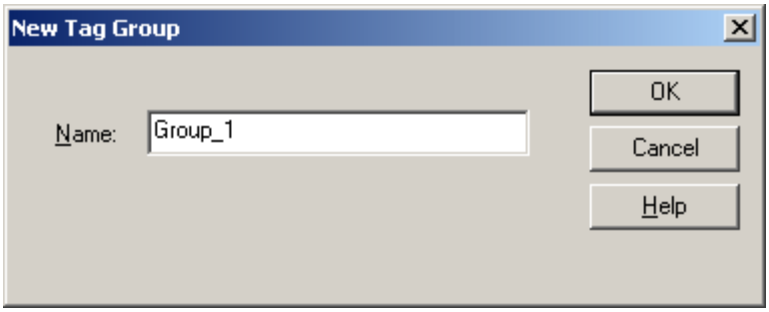
Like the server and many OPC Clients, the tag dialog can be used to browse the tags currently defined at the selected grouping level, duplicate tags and delete tags. This is especially useful when creating many similar tags. These functions can be accessed through the five buttons below the help button.

Note 1: The tag's properties can be changed at any time during an editing session.

Note 2: Event counter values are stored in 16 bit buffers. All tags updated from event counters must be configured with the 16 bit Intel (Lo Hi) device data format. For more information, refer to **Event Counters**.

Tag Groups

Tag groups are provided in order to organize tags. A tag group item can be added onto the device or onto another group through the main menu, item pop-up menu or the toolbar. An existing group can be edited by selecting and then clicking **Properties** from the main menu, the group's pop-up menu or with the toolbar.

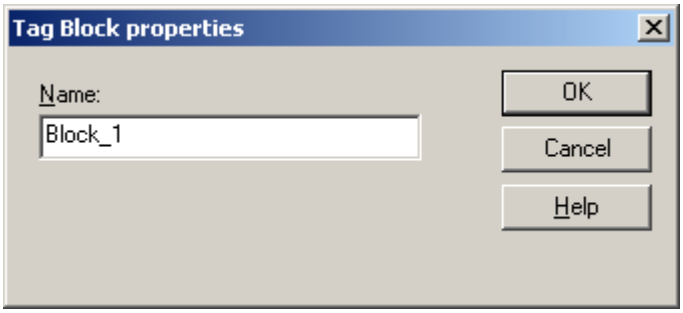


The only user-defined property that a tag group has is its name. Although a valid default name is generated when first creating a new group, it can be changed to any valid name. Valid names must start with a letter, consist of only letters, digits and underscores, be less than 32 characters long and be unique to the parent item. A tag group name may not be the same as a tag block at the same level since the server treats blocks as groups. The group's name can be changed at any time during the editing session. Groups may be nested up to three levels deep.

Tag Blocks

Tag blocks are a special type of group used by the **Transaction Editor** to contain all tags that can be updated by a common read or unsolicited transaction. The transaction common to all tags in the block is attached to the block item in the editor's item view. This common transaction should contain an **Update Tag** command for each tag in the group. Block tags with Read and Write client access permission will each have their own Write transaction. A **tag on group folder** icon in the Transaction Editor represents tag blocks only. The server represents tag blocks with the normal group folder icon.

A tag block item can be added to the device or a tag group. Tag blocks may be added using the main menu, the selected item's pop-up menu, or the toolbar. Existing blocks can be edited by selecting it then clicking **Properties** from the main menu, the pop-up menu or the toolbar.



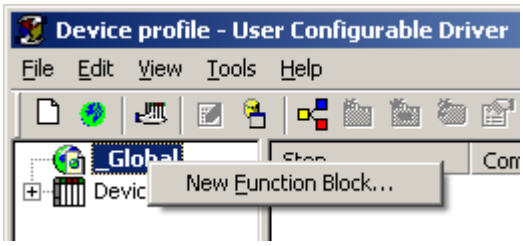
The only user-defined property that a tag block has is its name. Although a valid default name is generated when first creating a new block, it can be changed to any valid name. Valid names must start with a letter, consist of only letters, digits and underscores, be less than 32 characters long and be unique to the parent item. A tag block name may not be the same as a tag group at the same level since the server treats blocks as groups. The block's name can be changed at any time during an editing session. Groups and blocks may be nested up to three levels deep.

Function Blocks

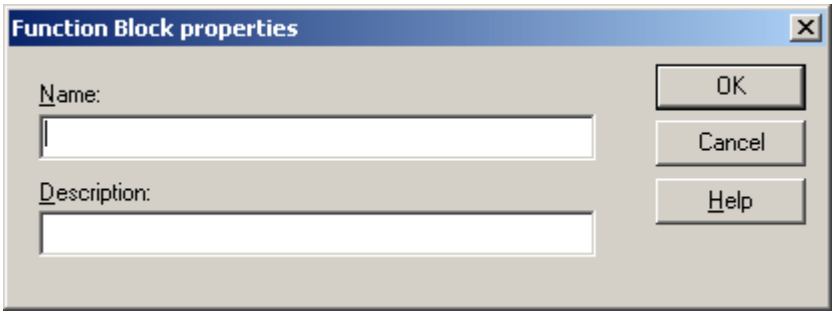
Function blocks can be used to define a series of commands that can be shared by any number of transactions, thus making projects more compact and easier to maintain. Function blocks reside in the U-CON global data store, and may be referenced by any device on any U-CON channel. To create a Function Block, follow the instructions below.

1. Invoke the **Transaction Editor** for any device on a U-CON channel. Select the **_Global item**.

2. Next, select **New Function Block** from the Edit menu or toolbar.



Note: The Function Block dialog should appear as shown below.



Descriptions of the parameters are as follows.

- **Name:** Valid names must start with a letter, consist of only letters, digits and underscores, be less than 32 characters long and be unique.
- **Description:** An optional description of the function block can be entered here.

3. Click **OK** to create the new function block. A new function block item will appear under the _Global node (item view,

left pane). "FBTransaction item" will be displayed under the new function block. Select the transaction item and enter the function block command in the Transaction View as would be done for any other transaction type. For more information, refer to **Insert Function Block Command**.

Note 1: Update Tag and **Insert Function Block** commands cannot be used in a function block. Update Tag commands can only be used in Read, Write and Unsolicited transactions that are explicitly associated to a particular tag or block of tags. Function blocks can not be used within function blocks.

Note 2: Be cautious when including **Go To** and **Label** commands in function blocks, as infinite loops can be created. When a Go To command is executed, the driver will scan all commands in the current Read, Write, or Unsolicited transaction from top to bottom looking for a matching Label. Commands in function blocks referenced in the transaction will be scanned in the order in which they appear.

Scratch Buffers

Each device has 256 scratch buffers associated with it. These buffers can be used to exchange information between transactions defined for that device. Data cannot be copied to a scratch buffer associated with a different device. Data stored in a scratch buffer will exist as long as the OPC server project is running or until the scratch buffer is overwritten in a transaction. **See Also:** **Global Buffers**.

When updating a tag from a scratch buffer, be aware that the value used will be the last value stored in the buffer. Depending on how the transaction is defined, this data may not necessarily represent the current state of a device. If no data has been stored in the scratch buffer at the time the Update Tag command is executed, the tag will be given a value of zero. **See Also:** **Update Tag Command**.

No special measures are taken when a **Copy Buffer Command** is executed when the buffer in question has not yet been initialized. If there is no data in the buffer, no bytes will be copied.

Note 1: For more information (and examples of how to use scratch buffers) refer to **Tips and Tricks**.

Note 2: For instructions on how to initialize a scratch buffer, refer to **Initialize Buffers**.

Global Buffers

Global buffers can be used to exchange information among devices. There are 256 global buffers. Each global buffer is associated with all devices under every channel. This is different from a **scratch buffer**, which is associated with only one device.

Important: Global buffers should be used with caution because they are associated with all devices for all channels. To exchange device-specific information (e.g., to make device-specific changes), use **scratch buffers**.

Note: For instructions on initializing a global buffer, refer to **Initialize Buffers**.

Rolling Buffer

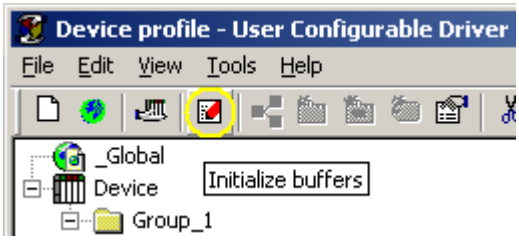
Rolling buffers are similar to scratch buffers but differ in that writes append data rather than replace it. Rolling buffers can be used to exchange information between transactions defined for that device. Data cannot be copied to a rolling buffer associated with a different device. Data stored in a rolling buffer will exist as long as the OPC server project is running or until the rolling buffer is overwritten in a transaction. Each device has an associated Rolling Buffer.

When updating a tag from a rolling buffer, be aware that the value used will be the last value stored in the buffer. Depending on how the transaction is defined, this data may not necessarily represent the current state of a device. If no data has been stored in the rolling buffer at the time the Update Tag command is executed, the tag will be given a value of zero. **See Also:** **Update Tag Command**.

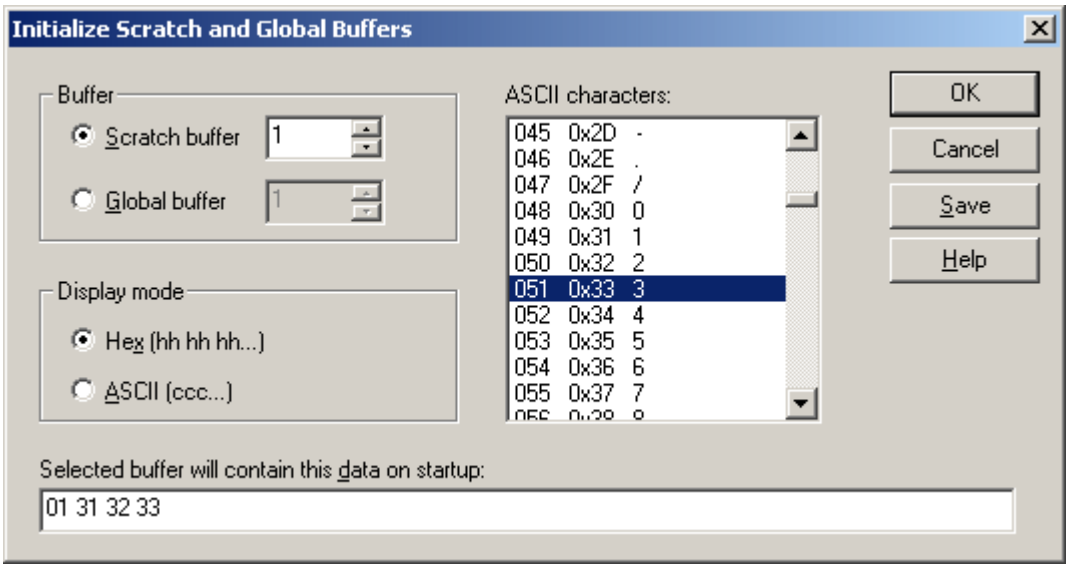
Initialize Buffers

A preset value for any **scratch buffer** and/or **global buffer** can be defined. The buffers will be loaded with these preset values on driver startup. To define buffer presets, follow the instructions below.

1. Click **Edit** | **Initialize Scratch and Global Buffers** or click on the toolbar icon as shown below.



2. The buffer initialization dialog should appear as shown below.



Descriptions of the parameters are as follows.

- **Buffer:** This parameter specifies the buffer for which a preset will be defined.
- **Display mode:** This parameter specifies how the preset data to be displayed in the edit box at the bottom of the dialog. In Hex mode, the hexadecimal value of each byte is displayed. When editing, each byte value must be entered as 2 hex digits (1-9, A-F) with a space separating each byte. If wishing to preset a buffer with an ASCII string, users will find it easier to work in ASCII mode where each data byte is displayed as the equivalent ASCII character. Users will not be able to view or edit preset data that contains non-printable characters in ASCII mode.
- **ASCII characters:** This scrolling list includes all data byte values in decimal (0-255) and hexadecimal (00 – FF), as well as the ASCII character mapped to each value. Users may utilize this as a reference. Items may be double-clicked in order to insert the byte into the preset data field at the bottom of the dialog.
- **Selected buffer will contain this data on startup:** This parameter displays the preset value for the selected buffer. It can be edited.
- **Save:** Clicking this button will save the preset value for the selected buffer without closing the dialog.
- **OK:** Clicking this button will save the preset value for the selected buffer and close the dialog.

Event Counters

Each transaction configured in the project automatically keeps track of how many times it is executed. These numbers are stored in special 16 bit buffers called Event Counters. All counter values are initialized to zero when a UCON project is first loaded. Counter values can reach 65535, at which point they wrap around back to 0. Tags from event counters can also be updated. **Transaction Event Counters** can be especially useful in scanner applications. For more information on their usage, refer to **Scanner Applications**.

Note: Event counter values are stored in 16 bit buffers. All tags updated from event counters must be configured with the 16 bit Intel (Lo Hi) device data format.

See Also: **Set Event Counter Command** and **Update Tag command**.

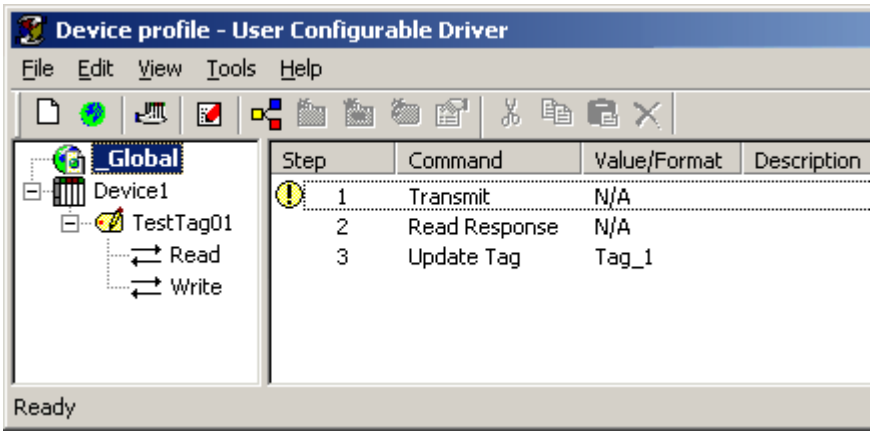
Buffer Pointers

The **read buffer**, **write buffer**, **scratch buffer** and **global buffer** each have an individual associated buffer pointer. The pointer is used to store the index or position of a byte in the associated buffer. Pointers can be moved to different bytes by using the **Seek Character** and **Move Buffer Pointer** commands. The **Update Tag** command has an option where data for a tag can be parsed starting at the current buffer pointer position. Buffer pointers are necessary when processing delimited lists. For an example, refer to **Tips and Tricks: Delimited Lists**.

For convenience, the read and write buffers are automatically reset to the first byte position at the start of each transaction. Since a major use of scratch and global buffers is to exchange data between transactions, scratch buffer pointers and global buffer pointers are not reset. Because of this, use care with relative moves of scratch and global buffer pointers.

Transaction Validation

The Transaction Editor performs a cursory inspection of the transaction after each edit is applied. Obvious errors are flagged with a yellow warning icon.



If **Verbose Transaction Validation** mode (located under the Transaction Editor's Tools option) is selected, a message box with a brief explanation of the problem will be shown. For the example above, the message would look a shown below.

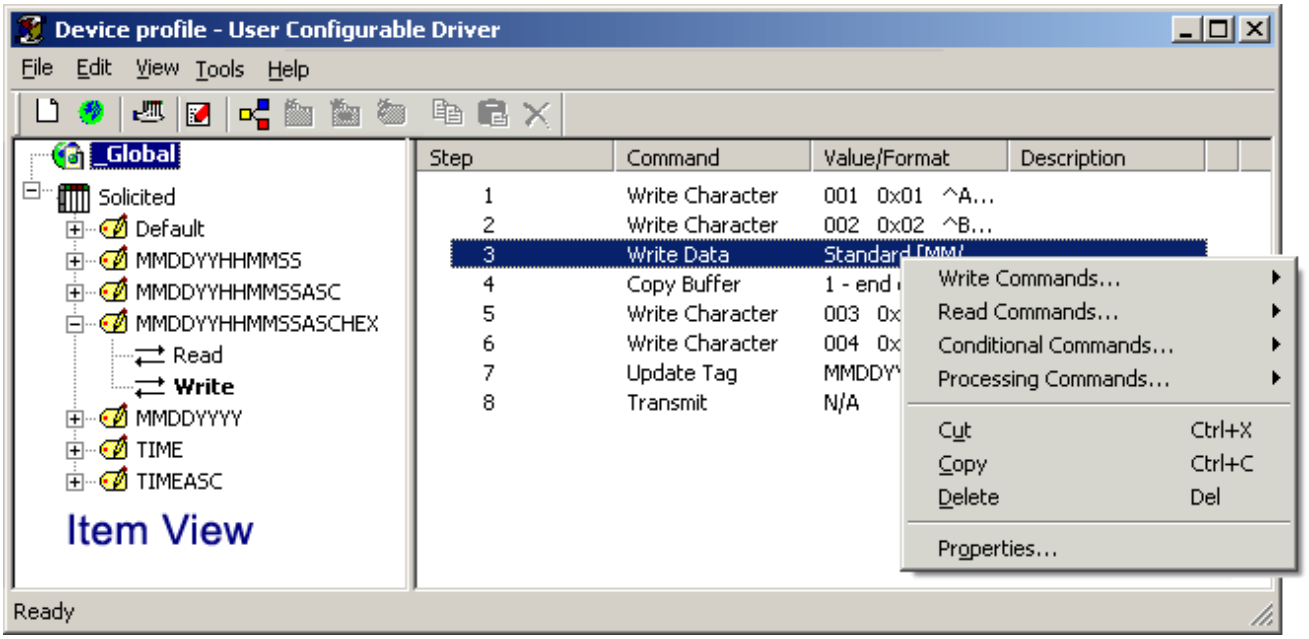
It should be emphasized that the Transaction Editor will only look for the most obvious problems. The absence of warnings is not a guarantee that the transaction definition will work. For more information on diagnosing problems, refer to **Tips and Tricks: Debugging**.

Transaction Commands

Each transaction must be defined so that the driver knows how to exchange data with a device. This is accomplished by constructing a list of commands that the driver should execute during a transaction. There are commands to construct request strings to be sent to the device, receive and store devices responses, validate responses, parse data from responses, convert data formats and update tag values (among others).

To define a transaction, first select the desired transaction in the **Transaction Editor**'s item view. Any currently

defined steps will be shown in the **Transaction View**. To add a command, right-click on the Transaction View. This will invoke a pop-up menu, as shown in the following screen sample.



If the mouse pointer is on a blank portion of the Transaction View when right-clicking, the new command will be added to the end of the list. If right-clicking on an existing command step, a new command will be inserted at that step. Alternatively, users can also use Edit to add commands.

Most commands have properties that must be specified. If this is the case, a command dialog will be presented before the new command is inserted into the transaction step list. To edit existing commands, users can double-click on them or select them and then click **Properties**. If users need to define other transactions that require similar command sequences, simply select and copy the commands of one transaction and paste them into the other.

For detailed information on a specific command, refer to the following links and information.

Write Commands

Write Device ID Command

Gets the Device ID set on the server's device property page, reformats it if needed, and places the result on the Write buffer.

Write Event Counter Command

Appends the value of the event counter to the Write buffer, which makes it possible to use of the event count value as a transaction ID in serial communication packets.

Write Character Command

Places a specified character on the Write buffer.

Write String Command

Places the specified string of characters on the Write buffer.

Write Data Command

Gets the Write value sent down from the client, reformats it if needed, and places the result on the Write buffer.

Write Check Sum Command

Computes the check sum, reformats it if needed, and places the result on the Write buffer.

Close Port Command

Closes the COM port associated with the current transaction.

Copy Buffer Command

Copies a portion of the Read buffer to the Write buffer.

Modify Byte Command
Sets one or more bits in a byte that was previously placed on the buffer, using the Write value sent down from the client. This is used to modify a byte in the Read, Write or scratch buffer.

Pause Command
Delays the execution of next command.

Control Serial Line
Controls the RTS and DTR lines to assert/de-assert the line manually.

Transmit Command
Sends the contents of the Write buffer to devices attached to channel.

Cache Write Value Command
Caches the value written after a transmit.

Read Commands

Read Response
Stores incoming data in Read buffer.

Update Tag Command
Parses data from Read buffer, reformats it if needed and updates the tag value accordingly.

Conditional Commands

Continue Command
The Continue command is one of several conditional actions available under the five test commands (Test String, Test Character, Test Device ID, Test Bit Within Byte, Test Check Sum, Test Frame Length). Continue tells the driver to do nothing as a result of the test, and proceed to the next command in the transaction. The Continue command has no user defined properties.

Test Device ID Command
Gets the Device ID set on the server's device property page, reformats it if needed and compares it with the Device ID in Read buffer. Executes different commands depending on the result.

Test Character Command
Compares a character in the Read or Write buffer with a specified character. Executes different commands depending on the result.

Test Bit within Byte Command
Compares a bit within a specified byte from the Read or Write buffer and compare it with a set value. Various actions can be taken depending on the result of the comparison.

Test Check Sum Command
Computes the check sum on portion of Read buffer, reformats it if needed, and compares it with the check sum in Read buffer. Executes different commands depending on the result.

Test String Command
Instructs the driver to parse a string from a buffer and compare it with a test value.

Test Frame Length Command
Instructs the driver to compare the length of the received frame with a test value.

Compare Buffer Command
Instructs the driver to compare two buffers. Executes different commands depending on the result.

Processing Commands

Clear Rolling Buffer Command
Sets all bytes in the rolling buffer to 0x00 and the length of the received frame to 0.

Clear RX Buffer Command
Sets all bytes in the Read buffer to 0x00 and the length of the received frame to 0.

Clear TX Buffer Command
Sets all bytes in the transmit butter to 0x00 and the current length of the Write frame to 0.

Set Event Counter Command
Sets the event counter of the current transaction to any valid number specified.

Deactivate Tag Command
Deactivates the tag. The transaction will not be executed again.

End Command
Terminates the transaction.

Go To Command
Processes the commands following the specified label command.

Invalidate Tag Command
Sets the tag's data as invalid. Client will report "bad quality" for tag data.

Label Command
Marks a transaction step for Go To commands.

Add Comment Command
Inserts a comment or a blank line in the Transaction Editor.

Log Event Command
Writes a message in the server's event log.

Seek Character Command
Instructs the driver to search for a given character in a specified buffer.

Move Buffer Pointer Command
Instructs the driver to change the current position of one of the buffer pointers. Pointers can be moved forward or backward.

Handle Escape Characters Command
Defines special handling of specific escape characters; for example, to add duplicate escape characters to Writes and to remove duplicates from Reads.

Serial Line Control Commands

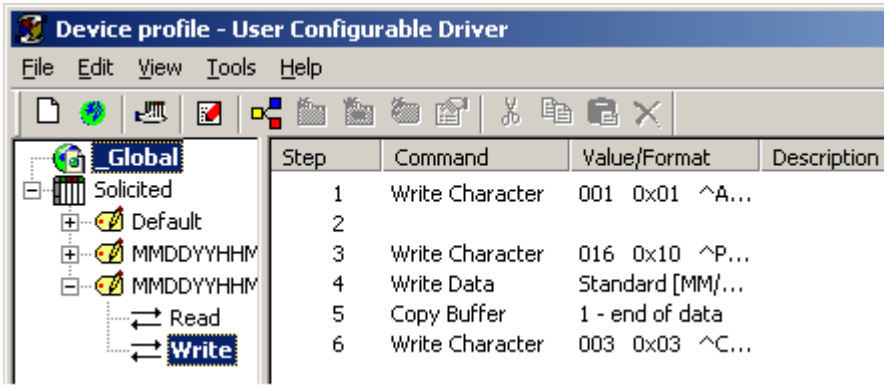
Control Serial Line Command
Controls the RTS and DTR lines to assert/de-assert the line manually.

Edit Menu Commands

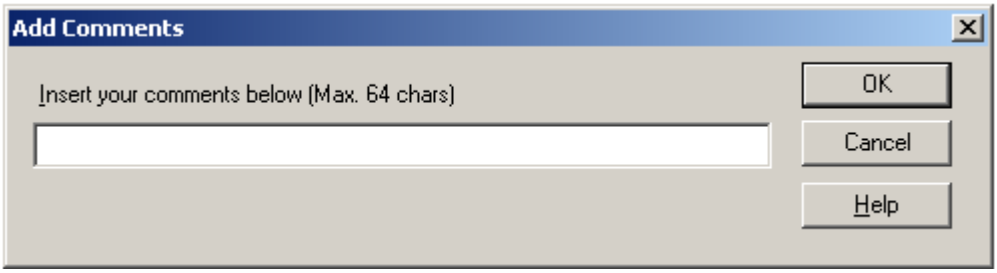
Insert Function Block Command
Inserts a previously defined **function block** into a Read, Write or Unsolicited transaction.

Add Comment Command

The **Add Comment** command can be used to insert a comment or a blank line in the **Transaction View**. For example, the screen shown below shows a blank line inserted above Step 3.



To add a **Add Comment** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Add Comment** from the pop-up menu. Alternatively, select **Edit** | **Add Comment** from the main menu. The comment (or blank line) will be inserted above the current step in the Transaction View. Comment lines have a maximum of 64 characters.



Note: In order to insert a blank line in the Transaction View, leave the Add Comments dialog field blank and simply click **OK**.

Cache Write Value Command

The Cache Write Value command tells the driver to cache the contents of the write buffer after transmit. It has no user defined properties.

To add a Cache Write Value command, right-click on the desired step in the Transaction View and then select **Write Commands** | **Cache Write Value**. Alternatively, click **Edit** | **New Write Command**. Then, select **Cache Write Value** from the main menu.

Caution: The command should be used for devices that are Write Only.

Clear Rolling Buffer Command

The **Clear Rolling Buffer** command tells the driver to set all bytes in the rolling buffer to 0x00 and the length of the received frame to 0. The command has no user-defined properties.

To add a Clear Rolling Buffer command, right-click on the desired step in the Transaction View and then select **Processing Commands** | **Clear Rolling Buffer** from the resulting pop-up menu. Alternatively, click **Edit** | **New Processing Command** | **Clear Rolling Buffer** from the main menu.

Caution: It is the user's responsibility to call the Clear Rolling Buffer Command. Failure to do so could result in buffer overflows.

Clear RX Buffer Command

The **Clear RX Buffer** command tells the driver to set all bytes in the read buffer to 0x00 and the length of the received frame to 0. The command has no user-defined properties.

To add a **Clear RX Buffer** command, right-click on the desired step in the **Transaction View** and then select

Processing Commands | **Clear RX Buffer** from the resulting pop-up menu. Alternatively, click **Edit** | **New Processing Command** and then select **Clear RX Buffer** from the main menu.

Note: The RX buffer is automatically cleared before each **Read Response** command is processed.

Clear TX Buffer Command

The **Clear TX Buffer** command tells the driver to set all bytes in the transmit buffer to 0x00 and the current length of the write frame to 0. The command has no user-defined properties.

To add a **Clear TX Buffer** command, right-click on the desired step the **Transaction View** and then select **Processing Commands** | **Clear TX Buffer** from the resulting pop-up menu. Alternatively, click **Edit** | **New Processing Command** and then select **Clear TX Buffer** from the main menu.

Note: The TX buffer is automatically cleared at the beginning of each transaction and after each Transmit and Read Response command.

Close Port Command

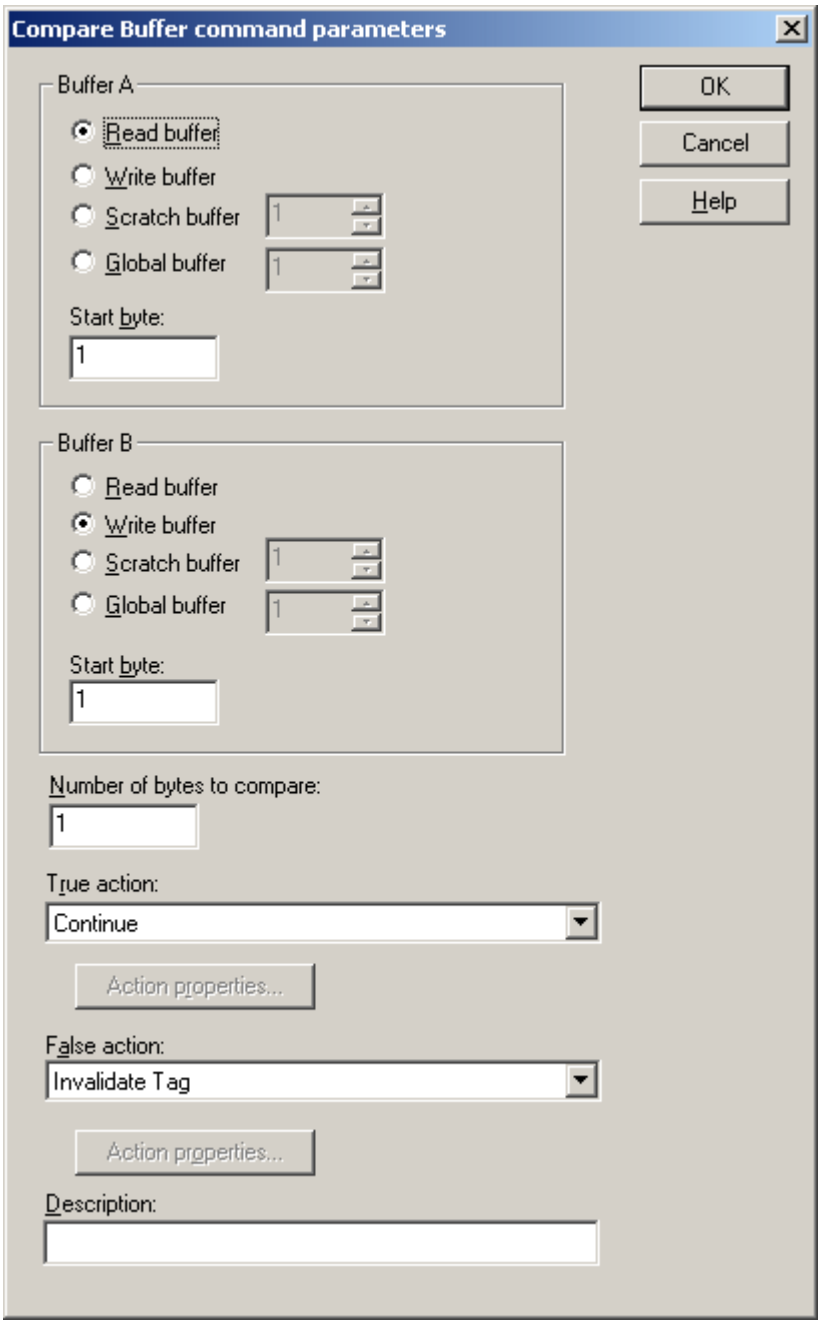
The **Close Port** command tells the driver to close the COM port associated with the current transaction. The port will be reopened automatically the next time something is written out that port. The Close Port command has no user defined properties.

To add a **Close Port** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Close Port** from the resulting pop-up menu. Alternatively, click **Edit** | **New Write Command** and then select **Close Port** from the main menu.

Compare Buffer Command

The Compare Buffer command tells the driver to compare specified sections of bytes in two buffers. Various actions can be taken depending on the result of that comparison.

To add a Compare Buffer command, right-click on the desired step in the Transaction View and then select **Conditional Commands** | **Compare Buffer**. Alternatively, click **Edit** | **New Conditional Command** and then select **Compare Buffer** from the main menu. In either case, the resulting dialog should appear as shown below.



- The **Read** buffer, **Write** buffer, **Scratch** buffer or **Global** buffer may be compared. When selecting the Scratch or Global buffer options, users must also specify the buffer indexes, data source buffers and also the **Start byte** within each buffer. The Start byte is the 1-based index of the first character to be parsed from the buffer.
- The **Number of bytes to compare control** is used to tell the driver the total number of bytes to compare from each buffer.
- As mentioned above, there are many actions that can be taken after a comparison is made; thus, the driver must be told what to do. Actions selected from the **True action** list will occur if the parse bytes from Buffer A equal the parsed bytes from Buffer B. Actions selected from the **False action** list specifies what the driver should do if the bytes do not agree. If the action requires that additional properties be defined, the **Action properties** button will become activated.
- The **Description** box is where a notation may be entered that will later be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very useful when reviewing the transaction definition later.

Note: The TX buffer is automatically cleared at the beginning of each transaction and after each Transmit and Read Response command. Following any of these conditions, the TX buffer must be copied to either a scratch buffer or a global buffer before being used in a comparison.

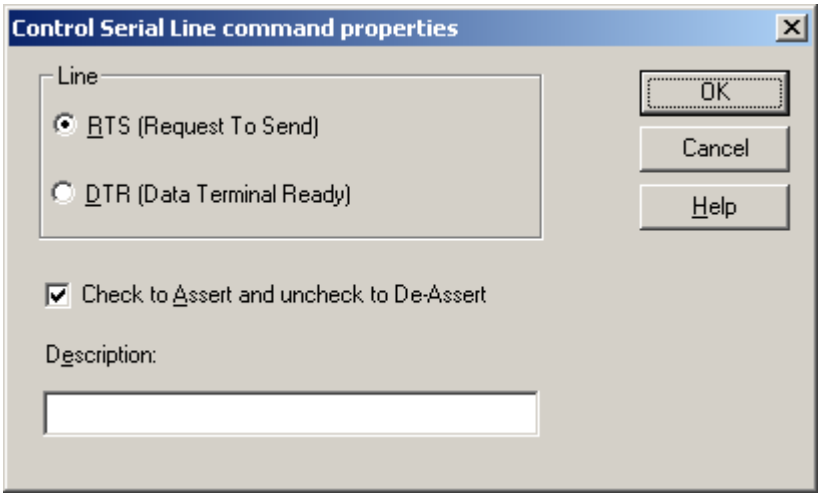
Continue Command

The **Continue** command is one of several conditional actions available under the five test commands (Test String, Test Character, Test Device ID, Test Bit Within Byte, and Test Check Sum). Continue tells the driver to do nothing as a result of the test, and proceed to the next command in the transaction. The Continue command has no user defined properties.

Control Serial Line Command

The **Control Serial Line** command allows for manual control of the RTS and DTR lines.

To add a **Control Serial Line** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Control Serial Line** from the resulting pop-up menu. Alternatively, click **Edit** | **New Write Command** and then select **Control Serial Line** from the main menu. In either case, the dialog should appear as shown below.



Important: This command should be used with caution. Before setting the RTS or DTR line high or low, be sure to set the line's default setting before the start of any transaction. Set the line back to default when the transaction completes and whenever there is a failure.

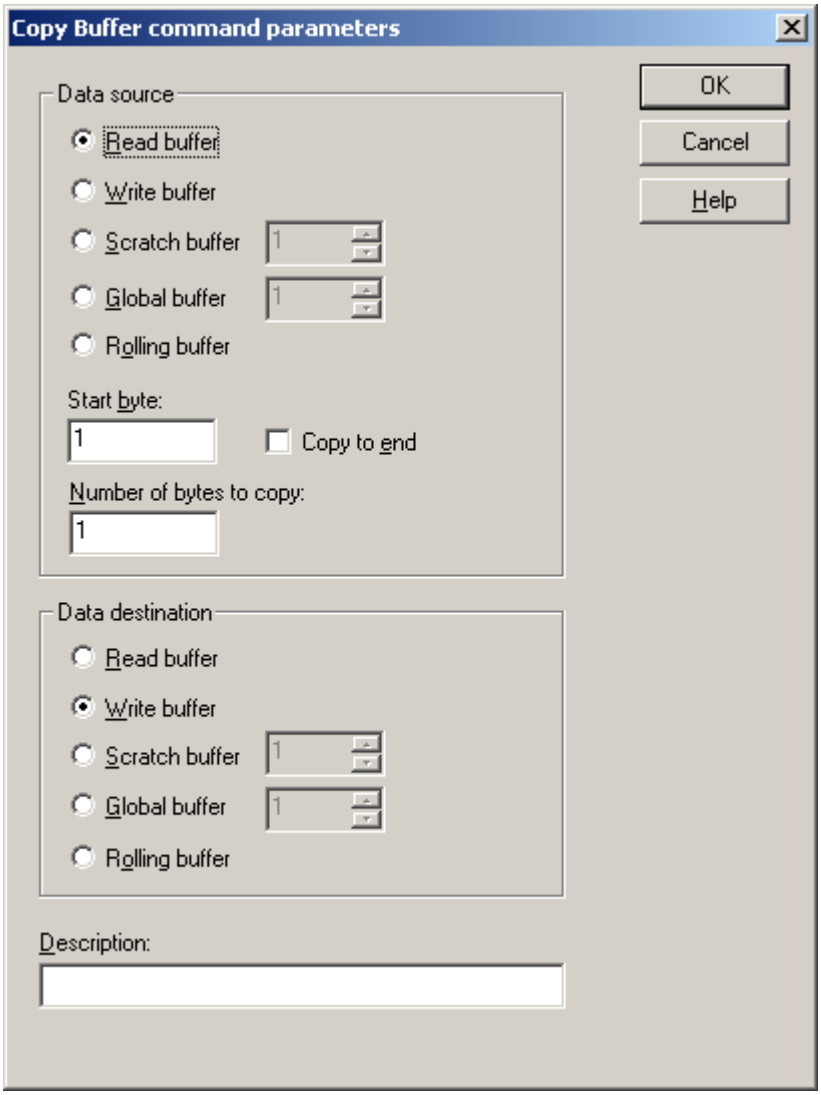
- In the **Line** box, select either **RTS** or **DTR**. Users must select only one at a time. After completing this dialog window for one line, it can be accessed again to select the other line.
- Enter a check next to **Check to Assert and uncheck to De-Assert** to assert the line. The checkbox should be unchecked to de-assert the line.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Copy Buffer Command

The **Copy Buffer** command tells the driver to copy a number of bytes from one buffer to another buffer. Bytes copied to the read, write or rolling buffers are placed after any data currently in that buffer. **Scratch buffers** and **global buffers** are flushed before new data is placed in them.

This command is normally used in conjunction with a **Modify Byte** command to construct a bit field or to store off data from a **Read Response** that will be used in subsequent transactions. Be careful that the selected source buffer will have valid data when using this command. For more information, refer to **Tips and Tricks**.

To add a **Copy Buffer** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Copy Buffer** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Copy Buffer** from the main menu. The dialog should appear as shown below.



- The **data source** is selected by checking the **Read buffer**, **Write buffer**, **Scratch buffer**, **Global buffer** or **Rolling Buffer** radio button. If either the scratch or global buffer is selected, the buffer index must be specified. If there are not enough bytes of data in the buffer, this command will be aborted and the transaction will fail. An error message will also be placed in the OPC server's event log. Users should be cautious of this when using scratch, global or rolling buffers as the data source.
- The **Start byte** control is used to tell the driver what byte in the source buffer to start the copy operation. The byte positions are addressed using a 1-based index.
- The **Copy to end** control is used to tell the driver to copy all of the data from the specified start byte to the last byte of data currently stored in the source buffer.
- The **Number of bytes to copy** control is used to tell the driver the total number of bytes to copy from the source buffer.
- The destination buffer is selected with the **Read buffer**, **Write buffer**, **Scratch buffer**, **Global buffer** or **Rolling buffer** radio buttons.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Deactivate Tag Command

The **Deactivate Tag** command tells the driver to set the tag's data quality to bad and to perform no more read or writes for that tag. It has no user-defined properties. Once a tag has been deactivated, it will stay deactivated. To reactivate a tag, the server project must be restarted, so use this command with care.

To add a Deactivate Tag command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Deactivate Tag** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Deactivate Tag** from the main menu.

Note: A Deactivate Tag command does not end the current transaction. The tag will remain active until the transaction has completed, thus giving users the chance to do any clean-up work such as logging a message or writing additional information to the device. To terminate the transaction at the time as tag deactivation, place an **End** command immediately after the Deactivate Tag command.

End Command

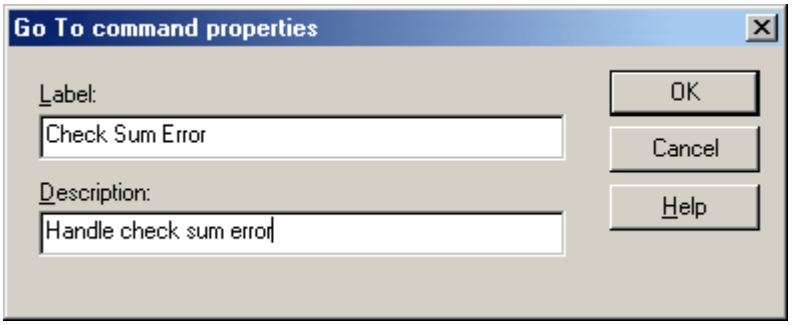
The **End** command tells the driver to stop processing the current transaction, and is generally used in conjunction with **Go To** and **Label** commands. A typical use of the end command is to prevent the driver from executing steps in a transaction that should only be executed as the result of a conditional command with a **Go To**. For more information, refer to the "Branching" section in **Tips and Tricks**. This command has no user-defined properties.

To add an End command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **End** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **End** from the main menu.

Go To Command

The **Go To** command tells the driver to search for the specified **Label** command in the current transaction and proceed from there. See the "**Branching**" section in **Tips and Tricks** for more information.

To add a **Go To** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Go To** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Go To** from the main menu. The dialog should appear as shown below.



- The **Label** identifies the Label command the driver will search for upon encountering this command. If the Label command is not found, an error message will be logged and the transaction will terminate.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Note: Go To commands should be used with caution. It is possible to set up "infinite loops" which will cause the driver to become stuck in a transaction. A simple example of an infinite loop would be as follows.

1. Label "Jump to here".
2. Go To "Jump to here".

It may be necessary to terminate the server in this event by pressing the "Ctrl-Alt-Del" key combination. Make sure that any transaction that uses a Go To command will always terminate, either by running to the last defined command

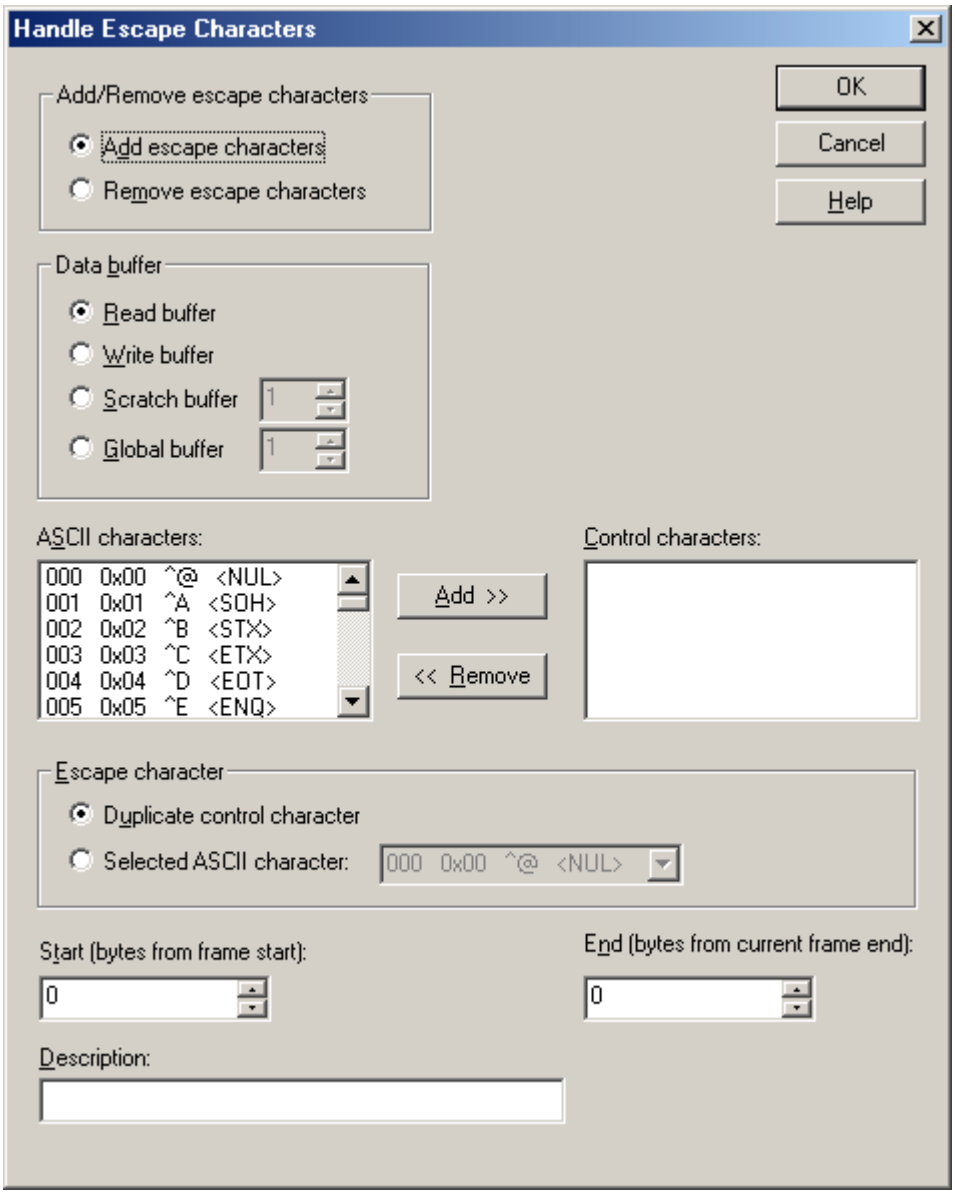
step or to an **End** command.

Handle Escape Characters Command

The **Handle Escape Characters** command is used to provide data transparency as required by some binary protocols. Some protocols assign a special meaning to certain character sequences. For example, the end of a variable length frame may be indicated by the sequence DLE ETX (0x10 0x03). A potential problem would exist if the data value 4099 (0x1003) must be transmitted in one of these frames. The receiving application would not know whether these two bytes are part of the data payload or indicate the end of the frame.

This type of ambiguity would typically be resolved or made "transparent" by doubling all occurrences of the DLE character within the data portion of the frame. Throughout the frame, DLE acts as an "escape" character, and must be interpreted in the context of what follows. In the example above, the value 4099 would be encoded as DLE DLE ETX. The receiving application would then interpret all doubled DLE characters as a single data byte with the value 0x10. The Handle Escape Characters command allows the U-CON (User-Configurable) Driver to add escape characters to outgoing frames, and remove them from received frames.

To add a Handle Escape Characters command, right-click on the desired step in the Transaction View and then select **Processing Commands** | **Handle Escape Characters** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Handle Escape Characters** from the main menu. In either case, the dialog should appear as shown below.



See Also: [Transaction View]

Add/Remove Escape Characters

Select **Add escape characters** to add escape characters as needed to the specified section of an outgoing frame. Select **Remove escape characters** to remove escape characters from the specified section of a received frame. Once the escape characters have been removed from a received frame, data values may be parsed by subsequent calls to the Update Tag command.

Data Buffer

Specify the buffer in which the frame to be processed by this command is stored. The Handle Escape Characters operation is done "in place." If choosing the **Scratch** or **Global Buffer** option, specify the buffer index in the box to the right.

ASCII Characters and Control Characters

Specify the control characters by selecting entries in the **ASCII characters** box and then clicking **Add>>**. If multiple control characters are selected, they will be processed independently: they will not be added or removed as a sequence. Users may select up to five control characters, although multiple Handle Escape Characters commands can be included in the transactions for protocols that require more.

Escape Character
Specify the escape character by either selecting **Duplicate control character** or **Selected ASCII character**.

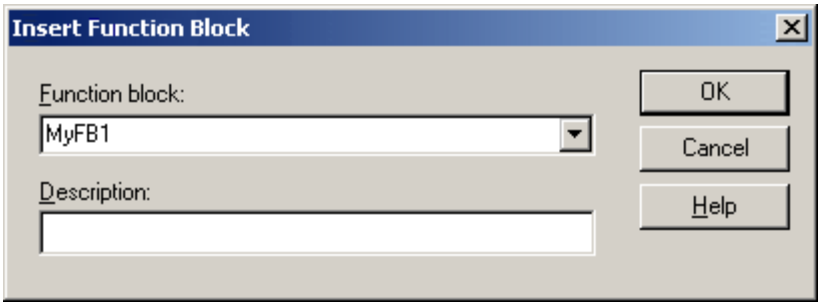
Start (bytes from frame start)
Specify the position of the first byte, relative to the start of the frame, of data to be processed by this command. The byte positions are addressed using a 1-based index. For example, specify 0 to include the first byte, specify 1 to skip the first byte, etc.

End (bytes from current frame end)
Specify the position of the last byte (relative to the end of the frame) of data to be processed by this command. The byte positions are address using a 1-based index. For example, specify 0 to include the last byte, specify 1 to process up to but not including the last byte, etc.

Description
This parameter is used to enter notations that will be displayed next to the command type in the Transaction View. Descriptions are optional, but can be very helpful when reviewing the transaction definition later.

Insert Function Block
Use the Insert Function Block command to include the commands defined in a previously defined function block in a Read, Write, or Unsolicited transaction.

To add an Insert Function Block command, right-click on the desired step in the **Transaction View** and then select **Insert Function Block** from the resulting pop-up menu. Alternatively, select **Edit** | **Insert Function Block** from the main menu. In either case, the dialog should appear as shown below.



- In the **Function block** drop-down list, select from the previously defined function blocks. **See Also:** **Function Blocks**.
- The **Description** box is used to enter a notation that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Important: Use caution when including Go To and Label commands in function blocks, as infinite loops can be created. When a Go To command is executed, the driver will scan all commands in the current Read, Write, or Unsolicited transaction from top to bottom looking for a matching Label. Commands in function blocks referenced in the transaction will be scanned in the order in which they appear.

Invalidate Tag Command
The **Invalidate Tag** command tells the driver to set the tag's data quality to bad. This command has no user-defined properties.

To add an **Invalidate Tag** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Invalidate Tag** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Invalidate Tag** from the main menu.

Note 1: An Invalidate Tag command does not end a transaction. In order to terminate the transaction when the tag is invalidated, place an **End** command immediately after the Invalidate Tag command.

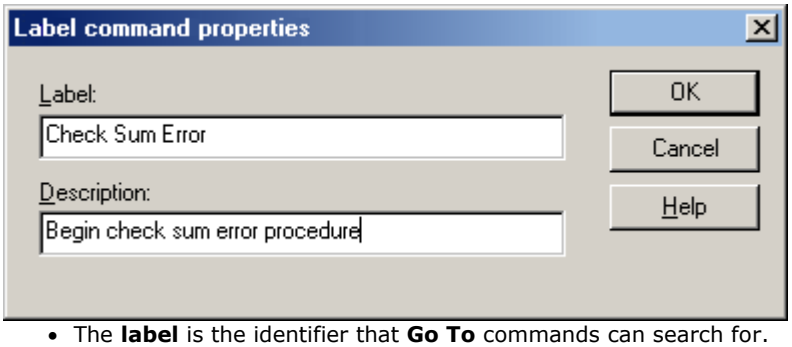
Note 2: The Invalidate Tag command is intended for use in a read transaction only. If an Invalidate Tag command is

included in a write transaction, it will have no effect on the quality of the tag.

Label Command

The **Label** command is used in conjunction with the **Go To** command. It does nothing other than serve as a target for **Go To** commands. See the "Branching" section in **Tips and Tricks** for more information.

To add a **Label** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Label** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Label** from the main menu. The dialog should appear as shown below.



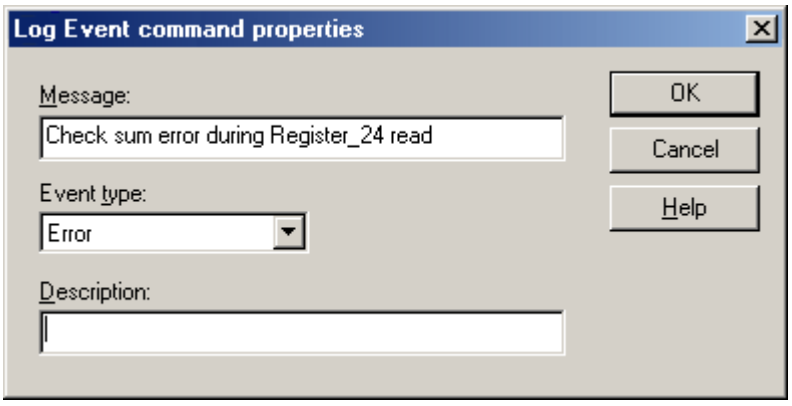
- The **label** is the identifier that **Go To** commands can search for.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Note: The Transaction Editor will not allow duplicate labels to be created in a transaction.

Log Event Command

The **Log Event** command tells the driver to send a message to the server's event log.

To add a **Log Event** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** |**Log Event** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Log Event** from the main menu. The dialog should appear as shown below.



The **Message** is the text the driver will write to the event log. The following special values may be entered in the Message field:

<tag>	<*tag*> will output the value of the tag
<RBuffer>	<*RBuffer*> will output the data in the read buffer
<WBuffer>	<*WBuffer*> will output the data in the

	write buffer

The **Event Type** sets the message-type icon, which will be associated with the message in the event log.

- Error
- Warning
- Information

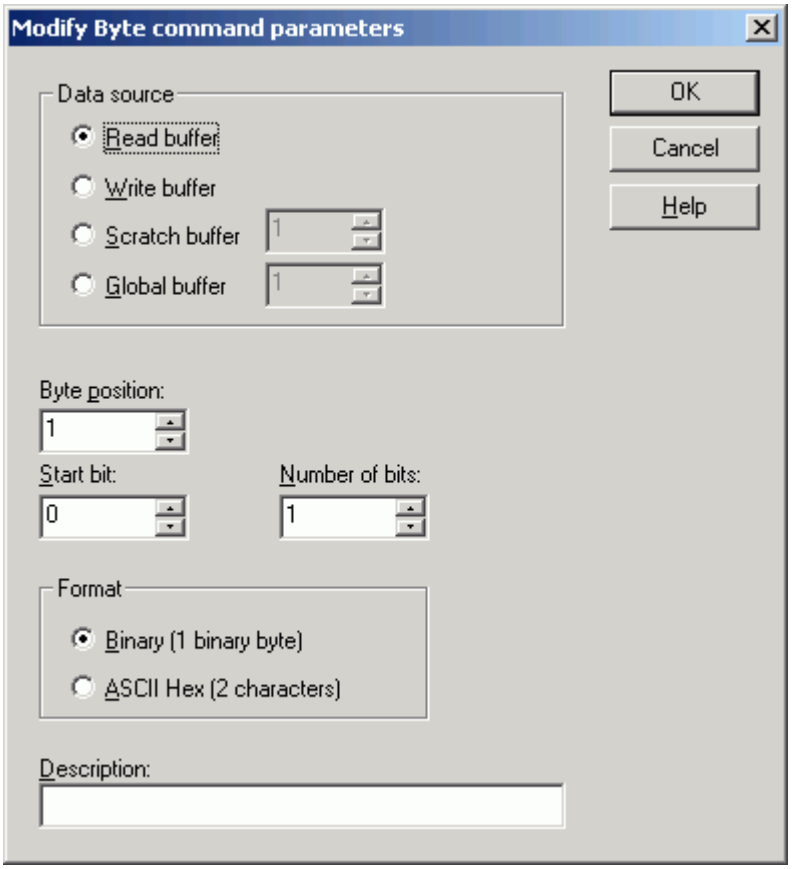
The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Modify Byte Command

The **Modify Byte** command tells the driver to modify a number of bits within a byte in the read buffer, write buffer, a scratch buffer, or a global buffer without changing the state of the other bits. The modified byte must have been placed in the buffer by a previous command in the transaction. The modified bits are set to zero or one depending on the write value sent down from the client (see example explanation below).

This command can be used in conjunction with the **Copy Buffer** command. The **Copy Buffer** and **Modify Byte** commands are used to change device settings that are represented by bit fields. For more information, refer to the "Bit Fields" section of **Tips and Tricks**.

To add a **Modify Byte** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Modify Byte** from the resulting pop-up menu. Alternatively, select **Edit** | **Write Commands** | **Modify Byte** from the main menu. The dialog should appear as shown below.



- The **data source** is selected by checking the **Read buffer**, **Write buffer**, **Scratch buffer** or **Global buffer** radio button. If either scratch or global buffer is chosen, the buffer index must also be specified.
- The **Byte position** control is used to specify what byte in the buffer will be modified. Byte positions are

addressed using a 1-based index.

- The **Start bit** control sets the index of the first bit to modify. As is customary, bits are numbered such that the least significant bit has index 0, and the most significant bit has index 7.
- The **Number of bits** control sets the number of bits that can be modified by this command.
- The **Format** option is used to modify binary or data in ASCII Hex data. If Binary is selected, this command will modify a single byte in the transmit buffer. If ASCII Hex is selected, two characters (assumed to be ASCII Hex "0" - "9", "A" - "F") are taken from the transmit buffer, converted to their binary equivalent, modified, then converted back to two ASCII Hex characters and copied back into the transmit buffer.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Note: Bits are changed to zero or one depending on the write value sent down from the client. The bits are set to the binary representation of the write value. If the write value exceeds the maximum value that can be represented by that number of bits, all changeable bits will be set to 1.

Example 1 (Binary data)

For this example, **Byte position** points to a byte in the write buffer with an initial value of 10110110, **Start bit** is 1 and **Number of bits** is 2. The table below shows what the byte value would be after this command is executed for various write values.

Initial byte value	Write value	Final byte value
10110110	0	1011**00**0
10110110	1	1011**01**0
10110110	2	1011**10**0
10110110	3 or greater	1011**11**0

Example 2 (ASCII Hex data)

For this example, **Byte position** points to the first of 2 ASCII hex characters in the write buffer with an initial value of "B6", **Start bit** is 1 and **Number of bits** is 2. The table below shows what the value would be after this command is executed for various write values. The actual ASCII Hex data in the transmit buffer is in quotes, the binary equivalent is in parenthesis.

Initial value	Write value	Final value
"B6" (10110110)	0	"B0" (1011**00**0)
"B6" (10110110)	1	"B2" (1011**01**0)
"B6" (10110110)	2	"B4" (1011**10**0)
"B6" (10110110)	3 or greater	"B6" (1011**11**0)

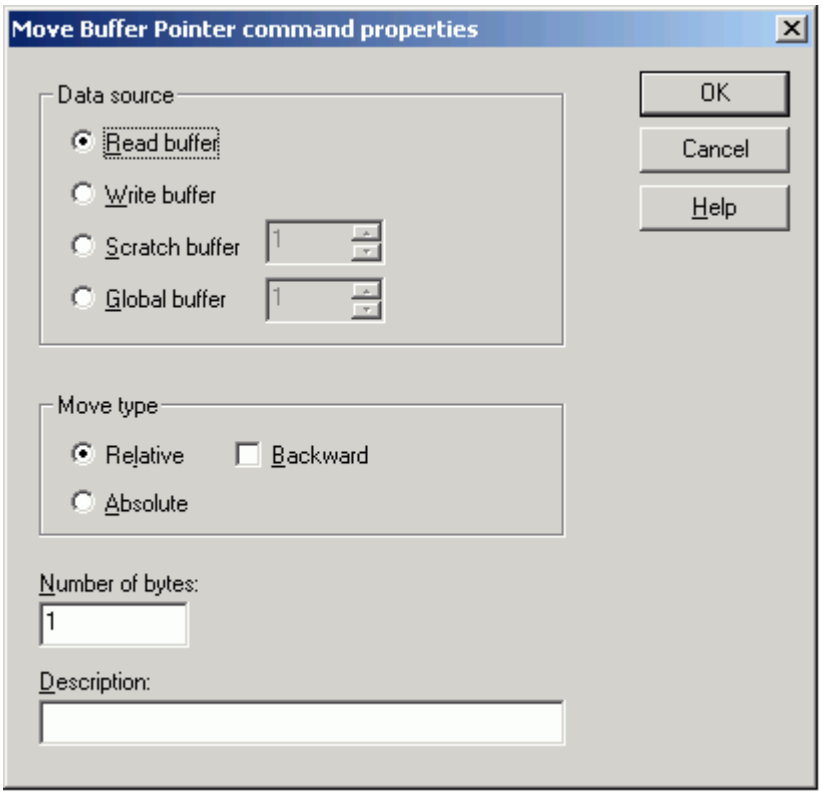
Move Buffer Pointer

Each buffer has its own, independent pointer which can be used to reference a particular byte in data processing commands such as **Update Tag**. **See Also:** **Buffer Pointers**.

The **Move Buffer Pointer** command tells the driver to change the current position of one of the buffer pointers. Pointers can be moved forward or backward.

The read and write buffer pointers are automatically reset to 1 at the start of each transaction. Scratch and global buffer pointers do not get reset automatically. The pointer position will not be changed if the specified move would place it beyond the current data content of the buffer. This command is especially useful for parsing delimited lists of variables. **See Also:** **Tips and Tricks: Delimited Lists**.

To add a **Move Buffer Pointer** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Move Buffer Pointer** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Move Buffer Pointer** from the main menu. In either case, the dialog should appear as shown below.



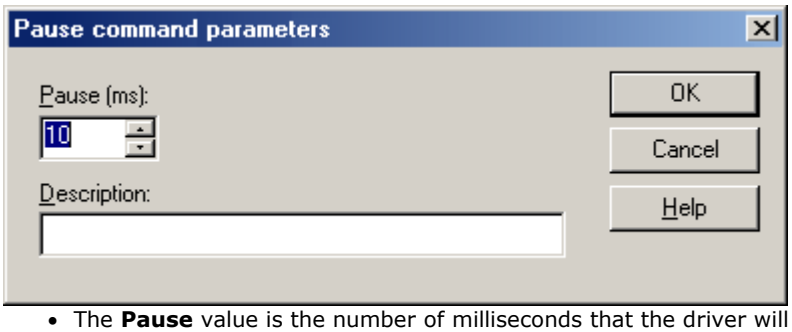
- The **data source** is selected by checking the **Read buffer**, **Write buffer**, **Scratch buffer** or **Global buffer** radio button. If selecting the Scratch or Global buffer option, users must also specify the buffer index in the box to the right.
- There are two types of moves: **Relative** and **Absolute**. A relative move is a specified number of bytes from the current pointer position. The default direction is forward. If **Backward** is checked, the pointer will move backward. An absolute move places the buffer pointer at the specified byte position, where the first byte is number 1, etc.
- The **Number of bytes** box is used to specify the number of bytes to advance the pointer in a relative move or the byte position in an absolute move.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.
- **Example:** Say that a read transaction receives the string "ABCDEFG" and places it into the read buffer. If no other buffer pointer operations have yet been performed during that transaction, the pointer will initially point to "A". Issuing the command illustrated above will place the read buffer pointer at "B". If the command is issued again, but with **Number of bytes** changed to 2, the pointer will end up at "D". An absolute move with Number of bytes set to 5 would place the pointer at "E".

Important: Use care with scratch and global buffer pointers. Unlike the read and write buffer pointers, scratch and global buffer pointers are not automatically reset at the start of each transaction.

Pause Command

The **Pause** command tells the driver to wait a specified period of time before processing the next command, which can be invaluable when communicating with slower devices. Normally, the **Pause** command is used in multiple Write Character/Transmit/Pause combinations. For more information on this technique, refer to the "Slowing Things Down" section of **Tips and Tricks**.

To add a **Pause** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Pause** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Pause** from the main menu. In either case, the dialog should appear as shown below.



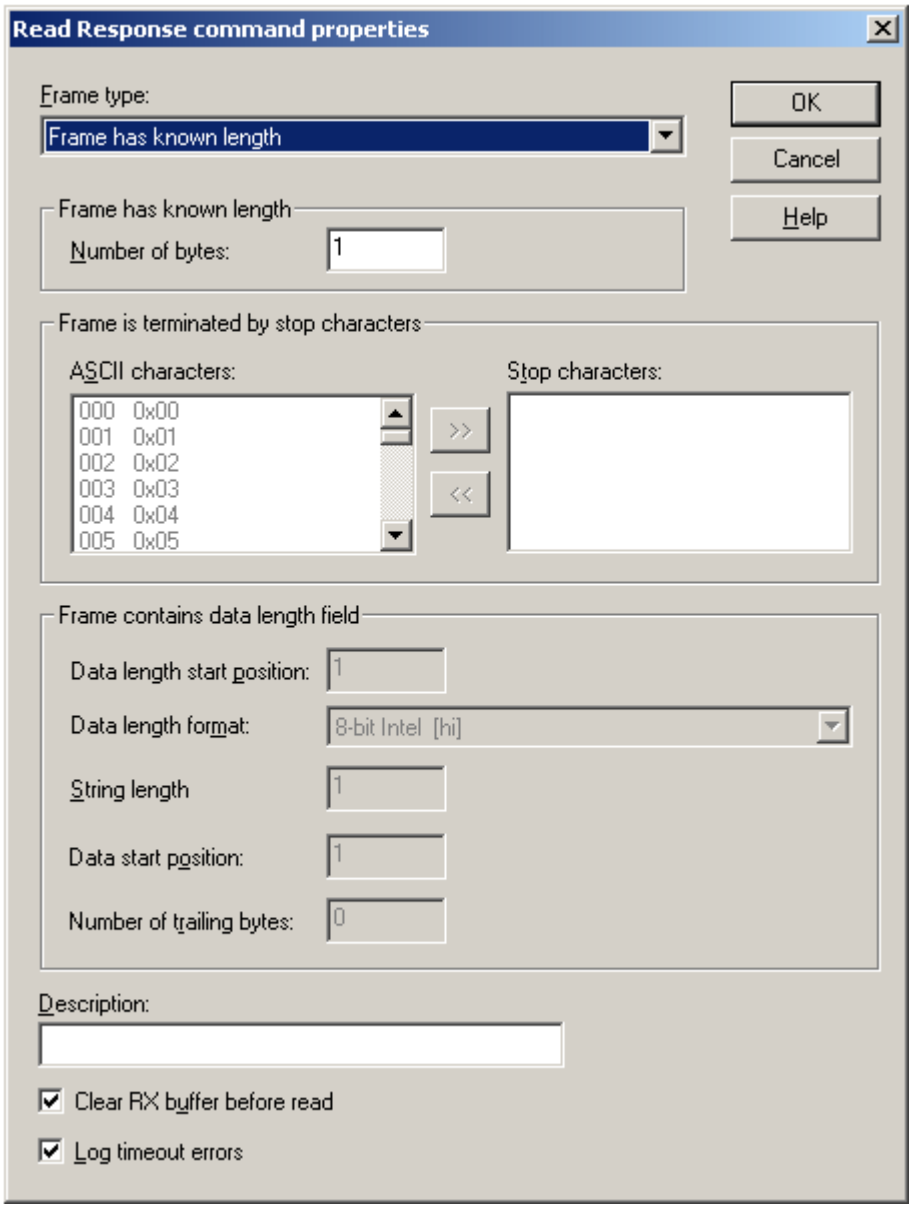
- The **Pause** value is the number of milliseconds that the driver will wait before processing the next command. Any value between 10 and 1000 ms, in 10 ms increments, can be selected.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Caution: The Pause command should not be used with Unsolicited UDP.

Read Response Command

The **Read Response** command tells the driver to receive data from the device and place it in the read buffer. The driver will continue to wait for data until either the user specified termination criteria has been met or the device timeout period (as set in the server's device property page) has elapsed.

To add a **Read Response** command, right-click on the desired step in the **Transaction View** and then select **Read Commands** | **Read Response** from the resulting pop-up menu. Alternatively, select **Edit** | **New Read Command** | **Read Response** from the main menu. In either case, the dialog should appear as shown below.



The driver must know when the last byte of the message has been received. There are three different frame type options which are distinguished by their receive termination methods: **Frame has known length**, **Frame is terminated by stop characters** and **Frame contains data length field**.

- When choosing the **Frame has known length** frame type, make sure to enter the correct **Number of bytes** the driver should wait for. Note that the amount of time the driver will wait for the specified number of bytes is set in the server's device property page under **Request timeout**. If the request times out, the driver will execute the transaction again up to the number of attempts that was specified in Device Properties. Any bytes in excess to that specified will be ignored.
- When choosing the **Frame is terminated by stop characters** frame type, make sure to define the character sequence that will mark the end of a response using the **ASCII characters** box and **Add >>** button. The driver will wait until the specified stop character sequence is received or the request times out (whichever occurs first).
- When choosing the **Frame contains data length field** frame type, make sure to specify where in the frame the data length field is located and what bytes are included in that count. The driver will try to receive bytes up to the end of the frame length field and then calculate how many more bytes to expect after that. The **Data length start position** is the 1-based byte position of the first byte in the data length field. The **Data length format** drop list box provides format options available for the data length field. The **String length** is the total number of characters in the selected data length field. The **Data start position** is the 1-based byte position of the first data

byte to include in the count. This will often be the first byte after the data length field. The **Number of trailing bytes** is the number of bytes the driver should expect after the indicated number of data bytes has been received. This might be used to handle cases where the check sum bytes are not included in the data length.

The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

By default, this command will automatically clear the read buffer before it accepts the next incoming byte. There are situations where this is not desired. The **Clear RX buffer before read** box can be used to disable this behavior. For example, say that a user needs to receive a frame that contains a variable number of data bytes, followed by an ETX byte that marks the end of the data, and a check sum byte. Such a frame must be received in two steps. First, issue a read response command configured to wait for an ETX stop character, and clear the RX buffer before read. This would get every thing except the check sum byte. To receive the check sum and append it to the read buffer, issue a second read response command configured to wait for a single byte, and not clear RX buffer before read.

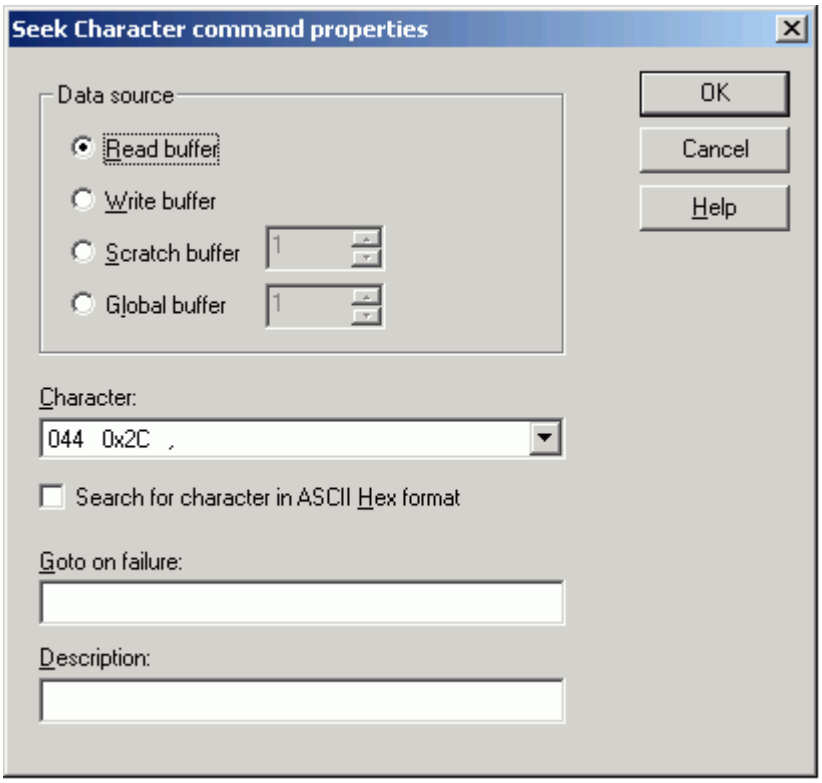
In some cases, a device will occasionally produce responses shorter than expected. Such a condition may occur if the device is in an error state, or if the protocol allows for headers of non-uniform length. The driver will timeout when attempting to read these short responses and will place a message to that effect in the server's event log. Over time, these messages can fill up the event log and obscure other log entries that may be of more interest. When in such a situation, suppress the logging of timeout errors by deselecting the **Log timeout errors** box at the bottom of the dialog. This should only be done when users understand why the timeouts occur and can show that they are relatively rare and insignificant.

Seek Character

Each buffer has its own, independent pointer that can be used to reference a particular byte in data processing commands such as the **Update Tag** command. The **Seek Character** command tells the driver to search for a given character in the specified buffer. The search will begin at the current buffer pointer position. The buffer pointer position will be relocated to the next instance of the specified character, if the character is found. If the character is not found, the pointer will not be changed. An optional Go To label may be executed on failure. **See Also:** **Buffer Pointers**.

Note: This command is especially useful for parsing delimited lists of variables. For more information, refer to **Tips and Tricks: Delimited Lists**.

To add a **Seek Character** command, right-click on the desired step in the **Transaction View** and then select **Processing Commands** | **Seek Character** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Seek Character** from the main menu. In either case, the dialog below will be shown.



- The **data source** is selected by checking the **Read buffer**, **Write buffer**, **Scratch buffer** or **Global buffer** radio button. If either the Scratch or Global buffer options are selected, the buffer index must also be specified.
- The **Character** drop list can be used to specify what character to search for. Any ASCII character 0x00 to 0xFF may be specified.
- The **Search for character in ASCII Hex format** check box can be used to specify if the data is in ASCII (default) or ASCII Hex format. For example, a comma in ASCII format will be a single byte with value 0x2C (","). A comma in ASCII Hex format will be two bytes with values 0x32 ("2") 0x43 ("C").

  Note: When searching for a character in ASCII Hex format, users must make sure that the search starts from the first byte of a string of ASCII Hex characters or an even number of bytes preceding them. The **Move Buffer Pointer** command may need to be used in order to initialize the pointer.
- The **Go To on failure** box can be used to specify a label that execution should proceed to if the specified characters are not found. This parameter is optional. If no label is specified, the buffer pointer will be left unchanged on seek failure and the driver will execute the next command in the transaction. If a label is specified but not found on seek failure, the current transaction will be aborted. The Transaction Editor will warn users of this condition. For more information, refer to **Label**.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

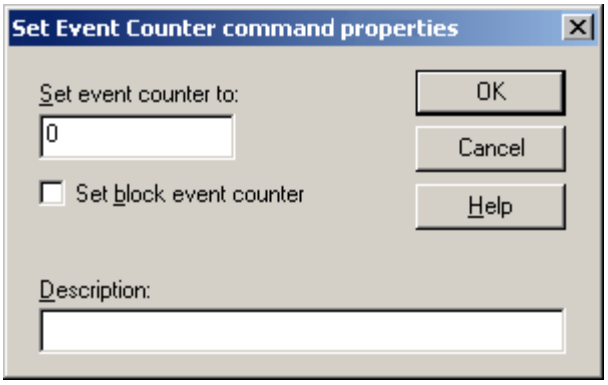
Example: A read transaction receives the string "ABC,DEFG" and places it into the read buffer. If no other buffer pointer operations have yet been performed, the pointer will point to "A". A "Seek Character" command as illustrated above would place the read buffer pointer at the comma ",". A second, identical "Seek Character" command would fail to find another comma. In this example, it would attempt to **Go To** the label "HandleParseFailure". The pointer would remain at the comma.

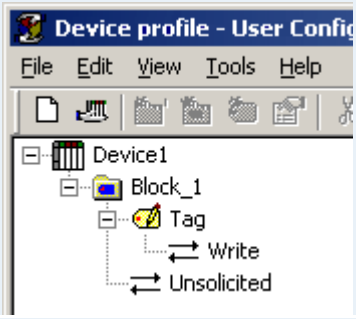
Important: Use care with scratch and global buffer pointers. Unlike the read and write buffer pointers, scratch and global buffer pointers are not automatically reset at the start of each transaction.

Set Event Counter Command

The **Set Event Counter** command is used to reset the value for the event counter of the current transaction.

To add a **Set Event Counter** command, right-click on the desired step in the **Transaction View** and then select **New Processing Commands** | **Set Event Counter** from the resulting pop-up menu. Alternatively, select **Edit** | **New Processing Command** | **Set Event Counter** from the main menu. The dialog should appear as shown below.



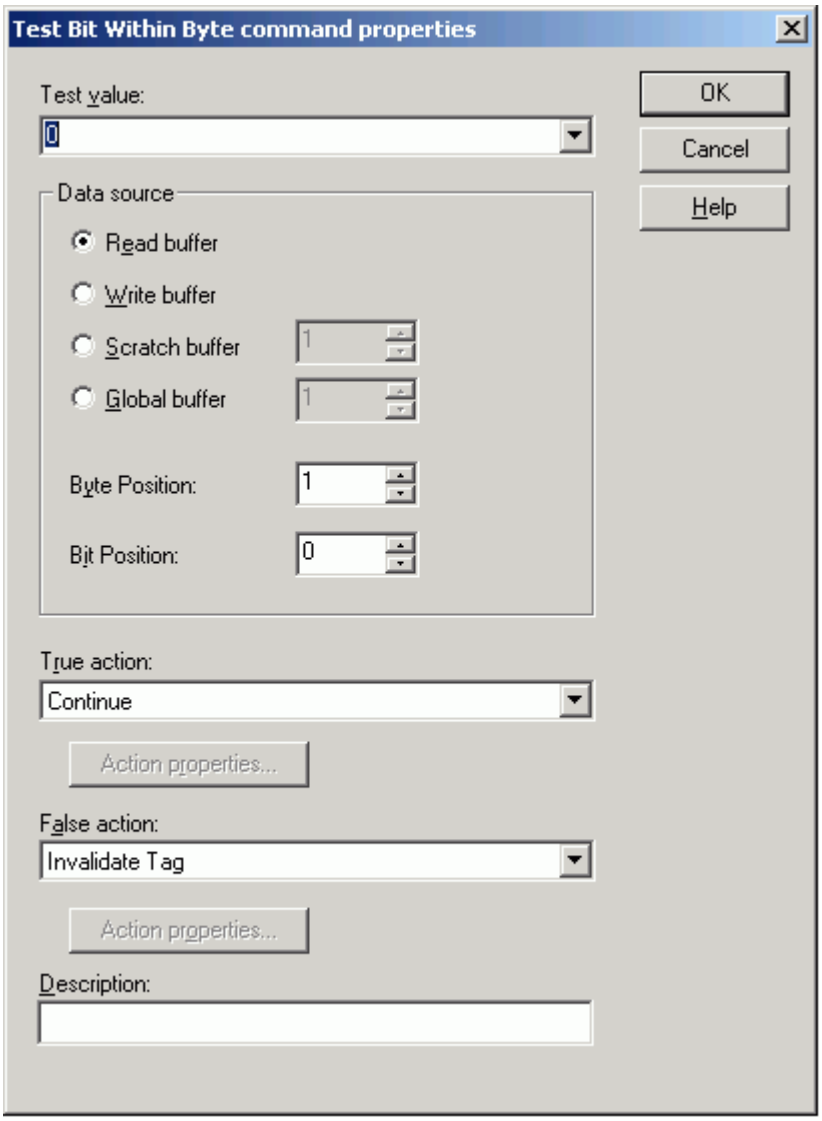
Parameter	Description
Set event counter to	Enter a number. The event counter of the current transaction will be reset to that number.
Set block event counter	The Set Event Counter command can reset the event counter of the transaction that the command is used in, or the counter of the read/unsolicited transaction of its parent block. Event counters are typically used in tag blocks, where one or more tags are updated from received data, and another tag is updated from the block's read or unsolicited transaction's event counter. **Unchecked:** The counter of the transaction that the command is used in will be reset to the number that was entered. **Checked:** The counter of the read/unsolicited transaction *of the parent block* will be reset. **Note:** The **Set block event counter** checkbox should be checked when the Set Event Counter Command is used in the event counter tag's write transaction. If this checkbox is left unchecked, the Set Event Counter Command will reset the *write transaction's* event counter. In the example shown below, the tag is within a parent block (Block_1). **Set block event counter** should be checked so that the event counter of Block_1's unsolicited transaction will be reset (i.e., the counter of the parent block transaction is reset). 
Description	Users can enter a notation that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

See Also: **Event Counters** and **Write Event Counter Command**.

Test Bit within Byte Command

The **Test Bit within Byte** command tells the driver to parse a bit within a specified byte from the read or write buffer (or one of the scratch or global buffers) and compare the bit value with a test value. Various actions can be taken depending on the result of that comparison. This command is useful for detecting communication errors in read transactions or for issuing different commands based on a write value in write transactions.

To add a **Test Bit within Byte** command, right-click on the desired step in the **Transaction View** and then select **Conditional Commands** | **Test Bit within Byte** from the resulting pop-up menu. Alternatively, select **Edit** | **New Conditional Command** | **Test Bit within Byte** from the main menu. In either case, the dialog should appear as shown below.



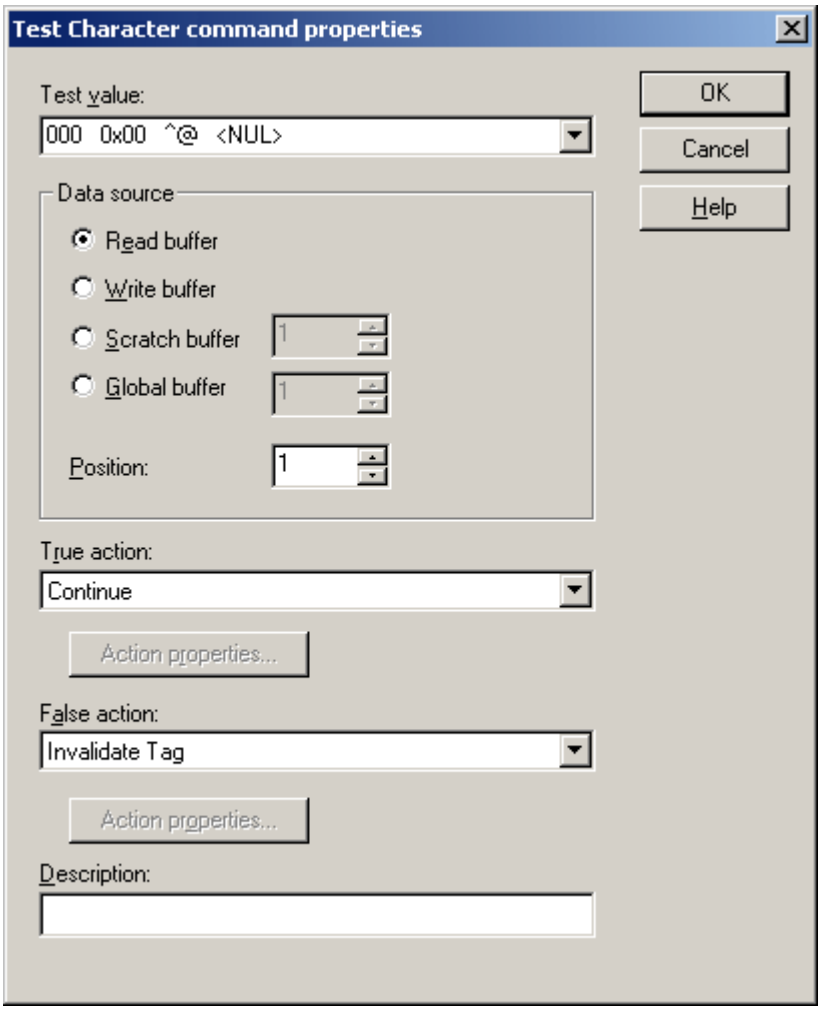
- Enter 0 or 1 in the **Test value** field. The test value will be compared with a bit within byte in the **Read buffer**, **Write buffer**, **Scratch buffer** or **Global buffer**.
- If either the Scratch or Global buffer options are selected, the buffer index must also be specified.
- In addition to the data source buffer, the **Byte Position** and **Bit Position** within that buffer must also be specified.
- The driver must be told what to do as a result of the comparison. Actions selected from the **True action** list define what will occur if the parsed bit within byte is the same as the test value. Actions selected from the **False action** list define what the driver should do if the values do not agree. Some actions require additional properties to be defined. If this is the case, the **Action properties** button will become activated.

- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can be very helpful when reviewing the transaction definition later.

Test Character Command

The **Test Character** command tells the driver to parse a character/byte from the read or write buffer, a scratch or a global buffer, and compare the character/byte with a test value. Various actions can be taken depending on the result of that comparison. This command is useful for detecting communication errors in read transactions or for issuing different commands based on a write value in write transactions.

To add a **Test Character** command, right-click on the desired step in the **Transaction View** and then select **Conditional Commands** | **Test Character** from the resulting pop-up menu. Alternatively, select **Edit** | **New Conditional Command** | **Test Character** from the main menu. In either case, the dialog below will be shown:



- The **Test value** drop list box provides the complete list of characters that may be added. The choices are listed with the decimal value, followed by the hex equivalent, and may be followed by the keyboard equivalent and mnemonic if applicable. Users may drop the list and select an item from it or take advantage of the **auto-complete** feature. The auto-complete feature is used to type in a decimal or hex value (in 0x?? format), or a character, and the indicated item will be selected from the list automatically. Clear the entry by pressing **Delete** or **Backspace** on the keyboard.
- The Test value may be compared with characters in the **Read buffer**, **Write buffer**, **Scratch buffer** or **Global buffer.** If either the Scratch or Global buffer options are selected, the buffer index must also be specified. In addition to the data source buffer, the **Position** within that buffer must also be specified. This is the 1-based

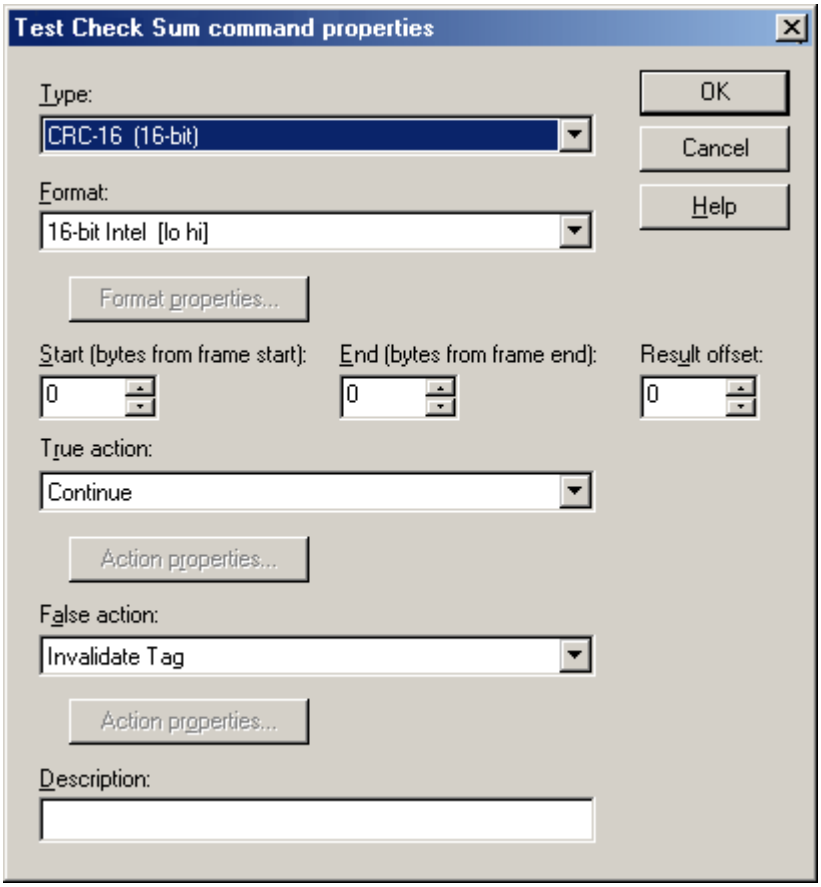
index of the character to be parsed from the buffer.

- The driver must be told what to do as a result of the comparison. Actions selected from the **True action** list define what will occur if the parsed byte is the same as the standard value. Actions selected from the **False action** list define what the driver should do if the bytes do not agree. Some actions require additional properties to be defined. If this is the case, the **Action properties** button will become activated.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Test Check Sum Command

The **Test Check Sum** command tells the driver to compute the check sum for a range of bytes in the read buffer, reformat it if necessary, and compare the result with the check sum value in the read buffer. Various actions can be taken depending on the result of that comparison. This command is useful for detecting communication errors.

To add a **Test Check Sum** command, right-click on the desired step in the **Transaction View**, and select **Conditional Commands** | **Test Check Sum** from the resulting pop-up menu. Alternatively, select **Edit** | **New Conditional Command** | **Test Check Sum** from the main menu. In either case, the dialog should appear as shown below.



- The **Type** drop list box provides the complete list of check sum algorithms supported. For more information of the check sum options, refer to **Check Sum Descriptions**.
- The **Format** drop list box provides the format options available for the selected check sum type. For a complete discussion of formats, refer to **Device data formats**. The Format Properties button will become enabled if the selected format has properties that must be set. Clicking this button will display the appropriate format configuration dialog.
- The **Start** and **End** boxes are used to tell the driver what bytes to include in the check sum calculation. The start value is given as a number of bytes from the beginning of the received frame. The end value is given as a number of bytes from the end of received frame. (Note the end value here has a different meaning than in the

Write Check Sum command. Here, it is defined relative to the frame end to allow for processing of variable length frames.) Generally, the check sum value will immediately follow the last byte included in the calculation, but not necessarily. The **Result offset** is used to indicate how many bytes are between the last byte included in the calculation and the check sum value.

- The driver must be told what to do depending on the result of the comparison. Actions selected from the **True action** list define what will occur if the received check sum is the same as the calculated value. Actions selected from the **False action** list define what the driver should do if the check sum values do not agree. Some actions require additional properties to be set. If this is the case, the **Action properties** button will be enabled. Clicking on these buttons will show an appropriate action configuration dialog.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Example

If wishing to test the check sum in a received frame with the following structure:

[SOH] [Data 1] [Data 2] … [Data N] [ETX] [BCC].

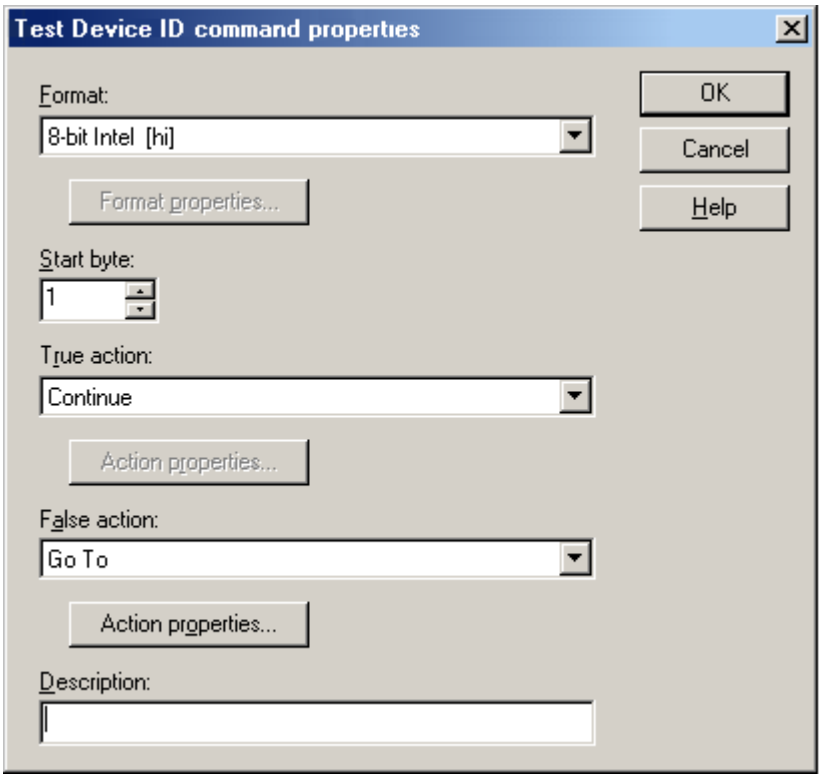
This frame contains an unknown number of data bytes, but has an ETX byte to mark the end of the data. The BCC is a single byte XOR check sum that includes just the data bytes, not the SOH and ETX characters. To test the BCC byte, users would configure a test check sum command to use the following:

Parameter	Setting
Type	XOR (8-bit)
Format	8-bit Intel
Format Properties	N/A
Start	1 (skip the SOH at start of frame)
End	2 (skip ETX and BCC at end of frame)
Result offset	1 (skip ETX between Data N and BCC)
True action	Action to take if calculated XOR of Data 1 to Data N is the same as received BCC.
Action properties (true)	Depends on True action selection.
False action	Action to take if calculated XOR of Data 1 to Data N is not the same as received BCC.
Action properties (false)	Depends on False action.
Description	Comment

Test Device ID Command

The **Test Device ID** command tells the driver to get the Device ID set in the server's device property page, reformat it if needed, and compare the result with the Device ID value in the read buffer. Various actions can be taken depending on the result of that comparison. This command is useful for detecting communication and physical device configuration errors.

To add a **Test Device ID** command, right-click on the desired step in the **Transaction View** and then select **Conditional Commands** | **Test Device ID** from the resulting pop-up menu. Alternatively, select **Edit** | **New Conditional Command** | **Test Device ID** from the main menu. In either case, the dialog should appear as it is shown below.

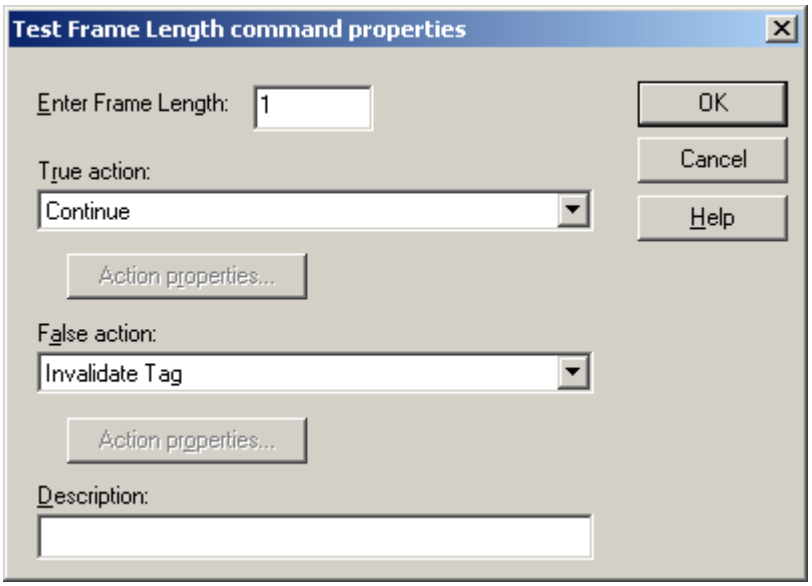


- The **Format** drop-down menu provides a list of the format options available. For a complete discussion of available formats, see **Device data formats**. The **Format Properties** button will become enabled if the selected format has properties that must be set.
- The **Start Byte** value tells the driver where in the read buffer the Device ID begins. This number is a 1-based index. The number of bytes parsed is based on the format specification.
- The driver must be told what to do as a result of the comparison. Actions selected from the **True action** list define what will occur if the Parsed ID is the same as the correct value. Actions selected from the **False action** list define what the driver should do if the IDs do not agree. Some actions require additional properties to be defined. If this is the case, the **Action properties** button will be enabled.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Test Frame Length Command

The **Test Frame Length** command tells the driver to compare the length of the received frame with a test value. Various actions can be taken depending on the result of that comparison. This command is especially useful when the incoming frame was received based on a sequence of stop characters.

To add a **Test Frame Length** command, right-click on the desired step in the **Transaction View** and then select **Conditional Commands** | **Test Frame Length** from the resulting pop-up menu. Alternatively, select **Edit** | **New Conditional Command** | **Test Frame Length** from the main menu. In either case, the dialog should appear as shown below.

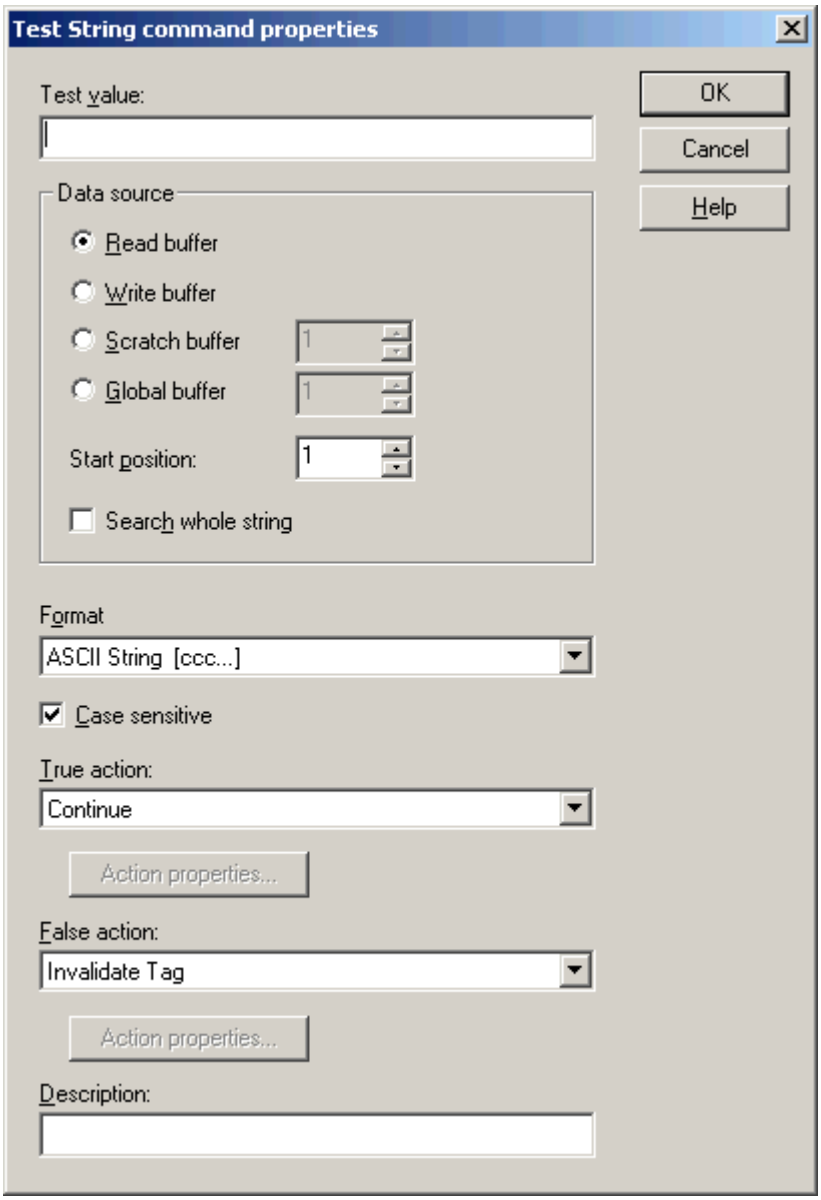


- The **Enter Frame Length** is the value that will be tested against.
- The driver must be told what to do as a result of the comparison. Actions selected from the **True action** list define what will occur if the received frame length is the same as the entered frame length value. Actions selected from the **False action** list define what the driver should do if the comparison fails. Some actions require additional properties to be defined. If this is the case, the **Action properties** button will become activated.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Test String Command

The **Test String** command tells the driver to parse a string from a buffer and compare it with a test value. Various actions can be taken depending on the result of that comparison. This command is useful for detecting communication errors in read transactions or for issuing different commands based on a write value in write transactions.

To add a **Test String** command, right-click on the desired step in the **Transaction View** and then select **Conditional Commands** | **Test String** from the resulting pop-up menu. Alternatively, select **Edit** | **New Conditional Command** | **Test String** from the main menu. In either case, the dialog should appear as shown below.



- The **Test Value** is the value that the string will be tested for. This string may be up to 64 characters in length. The test value may be compared with characters in the **Read buffer**, **Write buffer**, **Global buffer** or any **Scratch buffer** associated with the device. If the Scratch or Global buffer options are selected, the buffer index must also be specified.
- In addition to the data source buffer, the **Start position** within that buffer must also be specified. The Start position is the 1-based index of the first character to be parsed from the buffer. The number of characters parsed from the buffer will be the number of characters specified in the Test Value. If the buffer does not contain the required number of characters, the transaction will fail and an error message will be posted in the server's event log.
- In order to test or search the entire string, click to check **Search whole string**. This option ignores the value in the Start Position so that the whole string is tested for a string that matches the Test Value.
- The **Format** drop-down list is used to select the format (ASCII [default], Unicode, Alternating Byte ASCII), or ASCII Hex String From Nibbles.
- The string comparison will be **Case sensitive** by default. Uncheck this box to have the string comparison to not be case sensitive.
- Select from the **True Action** or **False Action** drop-down menus to tell the driver what to do as a result of the comparison. Actions selected from the **True action** list define what will occur if the string parsed from the buffer is the same as the Test value. Actions selected from the **False action** defines what will occur if the strings are

not the same. Some actions require additional properties to be defined. If this is the case, the **Action properties** button will become activated.

- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Transmit Command

The **Transmit** command tells the driver to output the contents of the write buffer. The **Transmit** command has no user defined properties.

To add a **Transmit** command, right-click on the desired step in the **Transaction View**, and select **Write Commands** | **Transmit** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Transmit** from the main menu.

Transmit Byte Command

The **Transmit Byte** command tells the driver to output a single byte from the write buffer. Only the byte transmitted is removed from the buffer: any other bytes will remain in the write buffer. The Transmit Byte command has no user-defined properties.

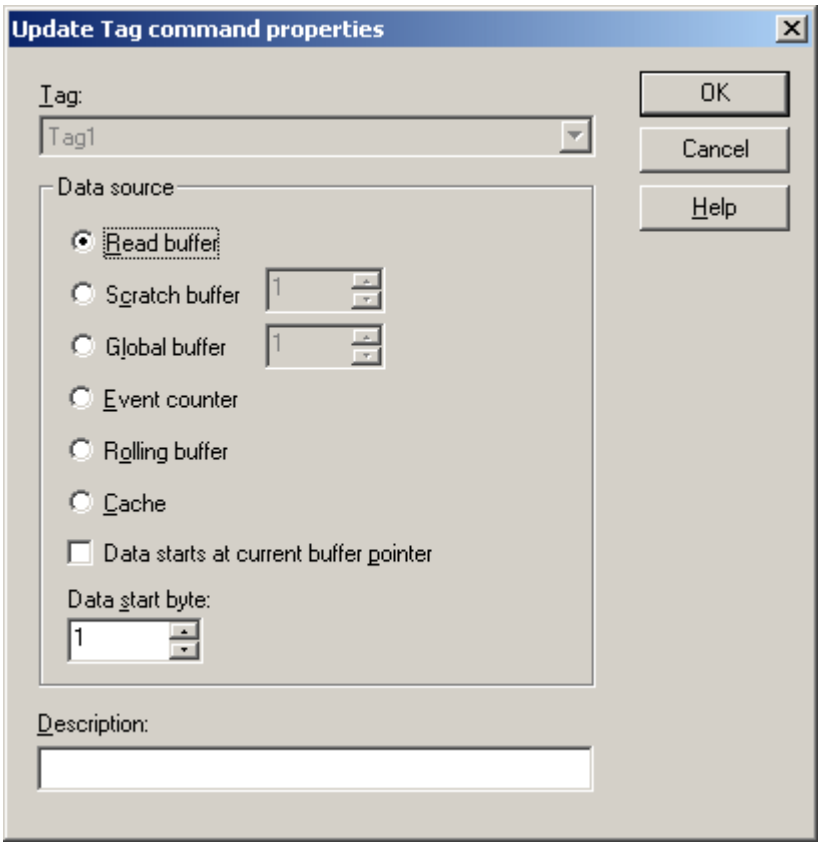
To add a Transmit Byte command, right-click on the desired step in the Transaction View and then select **Write Commands** | **Transmit Byte** from the resulting pop-up menu. Alternatively, click **Edit** | **New Write Command** | **Transmit Byte** from the main menu.

Note: The write buffer is not cleared after a transmit byte command.

Update Tag Command

The **Update Tag** command tells the driver to parse the data value from a read buffer, **scratch buffer**, **global buffer**, cache, the transaction's **event counter** or the **rolling buffer**. It then reformats as needed and updates the tag value accordingly.

To add an **Update Tag** command, right-click on the desired step in the **Transaction View** and then select **Read Commands** | **Update Tag**. Alternatively, select **Edit** | **New Read Command** | **Update Tag** from the main menu. In either case, the dialog should appear as shown below.



Note: If the transaction belongs to a tag block member, the tag must be selected to update from the **Tag** drop list. Otherwise, the transaction's parent tag will automatically be selected.

- The data source is selected by checking the **Read Buffer** (default), **Scratch Buffer**, **Global Buffer**, **Cache**, **Event Counter** or **Rolling Buffer** radio button. If the scratch or global buffer option is selected, users must specify which buffer index using the spin control to the right of the radio button. If no data has been stored in the scratch or global buffer when this command is executed, the tag value will be set to zero or a null string. If the event counter option is selected, the tag will be updated with the transaction's current event count. If the cache option is selected, the tag will be updated with the last value written to the tag.
- Check the **Data starts at current buffer pointer** box if the data for the selected tag begins at the current pointer position of the selected data source. The pointer must have been set prior to the execution of this command with either **Move Buffer Pointer** or **Seek Character** commands. For more information, refer to **Buffer Pointers** and **Tips and Tricks: Delimited Lists**.
- If the **Data starts at current buffer pointer** box is not checked, use the **Data start byte** box to specify where the tag's data begins. The first byte in the buffer is number 1.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can be very helpful when reviewing the transaction definition later.

Note: The format of the data to be parsed is taken from the selected tag's definition. For example, if the device data format 16 bit Intel [lo hi] was specified for the selected tag, the driver will attempt to parse two bytes from the specified source buffer and construct a 16 bit integer value from those bytes. The low byte of the integer will be the byte pointed to or given by the **Data start byte** setting. The high byte will be the following byte in the source buffer. This integer will then be converted to the tag's data type and stored. The stored value will be sent up to the client application as called.

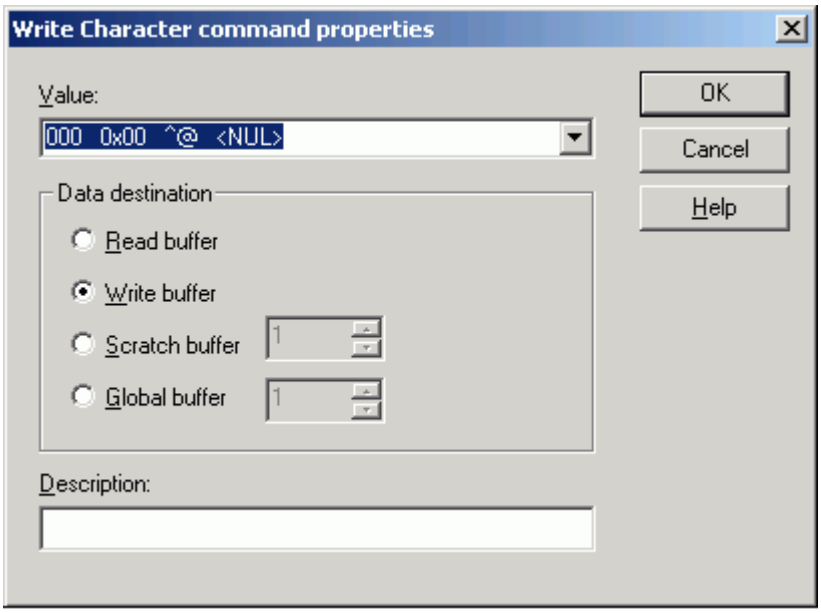
Caution: The cache option should only be selected for Write Only applications.

See Also: **Tags** and **Device Data Formats**.

Write Character Command

The Write Character command tells the driver to append a single byte character to the write, read, scratch or global buffer. The character need not be a printable ASCII character such as a letter, number, or punctuation mark. Anything with a binary equivalent of 0 to 255 is acceptable. If needing to write a sequence of printable characters, it may be easier to use the **Write String** command instead.

To add a Write Character command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Write Character** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Write Character** from the main menu. In either case, the dialog should appear as shown below.



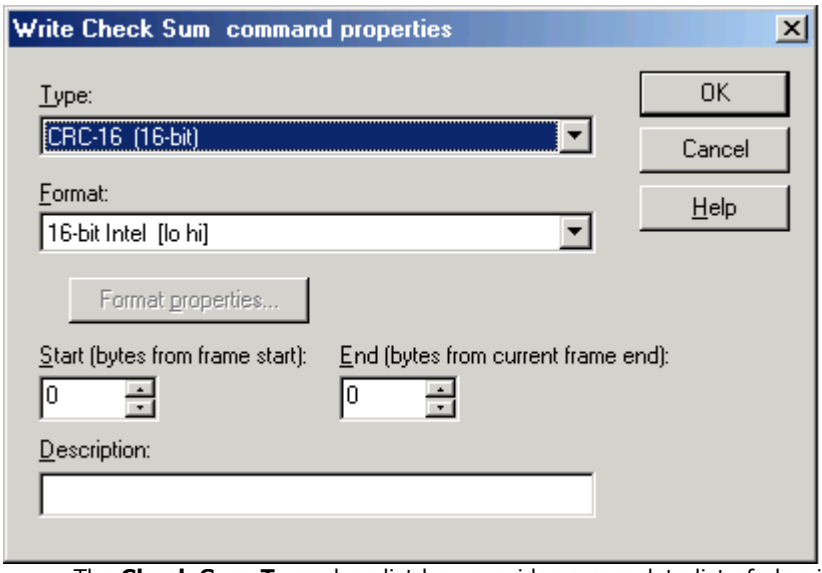
- The **Value** drop list box gives users a complete list of characters that may be added. Each entry in the list provides the ASCII character code in decimal followed by its hex equivalent. Some entries may have a third and forth column giving the keyboard equivalent and mnemonic when applicable. Users may drop the list and select an item from it. They can also take advantage of the **auto-complete** feature. The auto-complete feature is used to type in a decimal or hex value (in 0x?? format), or a character, and the indicated item will be selected from the list automatically. The entry can be cleared by pressing Delete or Backspace on the keyboard.
- Use the **Data destination** radio buttons to select the destination: **Read buffer**, **Write buffer**, **Scratch buffer**, or **Global buffer**. If the Scratch or Global buffer options are selected, users must also specify the buffer index in the box to the right. If there are not enough bytes of data in the buffer, this command will be aborted and the transaction will fail, and an error message will be placed in the OPC server's event log.

  Note: Data will be appended to TX and RX buffers, but not scratch or global buffers. To append data to the current contents of a scratch or global buffer, copy that data to either the RX or TX buffer, append that buffer, then copy the contents back to the scratch or global buffer. **See Also:** **Copy Buffer Command**.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Write Check Sum Command

The Write Check Sum command tells the driver to compute a check sum, reformat it if needed and append the result to the write buffer. There are several choices for common check sum types and device data formats.

To add a **Write Check Sum** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Write Check Sum** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command|Write Check Sum** from the main menu. In either case, the dialog should appear as shown below.



- The **Check Sum Type** drop list box provides a complete list of algorithms supported. For more information on check sum options, read **Check Sum Descriptions**.
- The **Format** drop list box provides users the format options available for the selected check sum type. For a complete discussion of available formats, see **Device data formats**. The **Format Properties** button will become enabled if the selected format has properties that must be set.

  All **check sum calculations** are performed over a range of bytes in a message frame. The **Start** and **End** fields are used to tell the driver what bytes to include in the calculation. The start value is given as a number of bytes from the beginning of the frame. The end value is given as a number of bytes from the current end of the frame, i.e. the last byte placed on the write frame before the Write Check Sum command is processed. (Note that the end value here has a slightly different meaning than in the **Test Check Sum** command.) The Start and End values will almost always be zero. For example, suppose the transaction consists of a **Write String** command followed by a Write Check Sum and a **Transmit**. Suppose the string is "0123456789ABC", and users need to compute a check sum over all of the characters in the string and place the result after the "C". In this case, both the Start and End values would have to be zero. Or, if the check sum calculation needs to go from the "1" to "9" inclusively, then the Start value must be 1 and the End value must be 3. Any additional characters added to the frame by commands placed after the Write Check Sum command cannot be included in the calculation.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Write Data Command

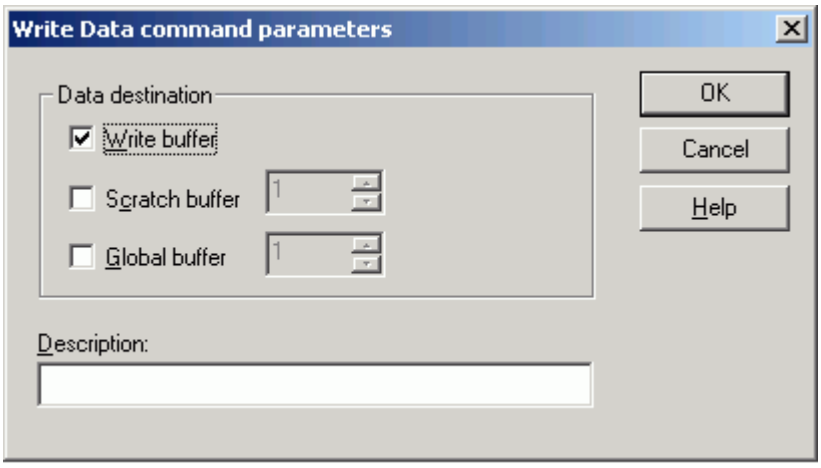
The Write Data command tells the driver to get the write value sent down from the client application, convert it to the specified **device data format** and do any of the following:

- Append the write buffer with the result.
- Store the result in a **scratch buffer** (the scratch buffer will be cleared first).
- Store the result in a **global buffer** (the global buffer will be cleared first).

or

- Any combination of the above actions.

To add a **Write Data** command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Write Data** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Write Data** from the main menu. The dialog should appear as shown below.

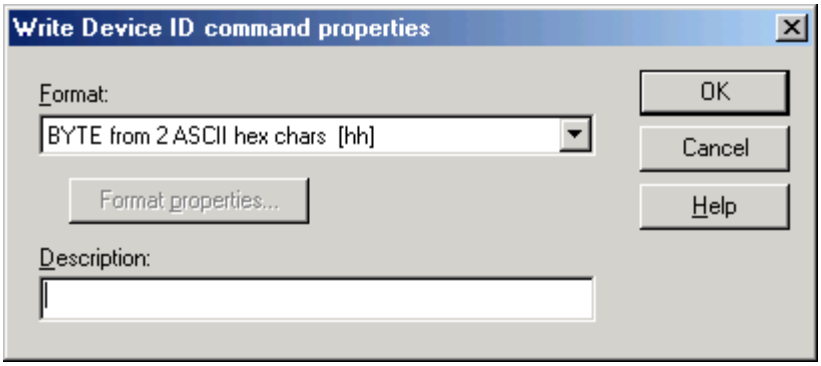


- The **Write buffer** check box tells the driver to append the write buffer with the formatted write value. This box will be checked by default. Click the **Scratch buffer** or **Global buffer** checkbox to place the formatted write value in a scratch or global buffer. The buffer index is selected with the spin control to the right of the check box. The scratch or global buffer chosen will be cleared before the formatted write value is stored.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Write Device ID Command

The Write Device ID command tells the driver to get the ID number set in the server's device property page, reformat it if needed, and append the result to the write buffer.

To add a Write Device ID command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Write Device ID** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Write Device ID** from the main menu. In either case, the dialog should appear as shown below.



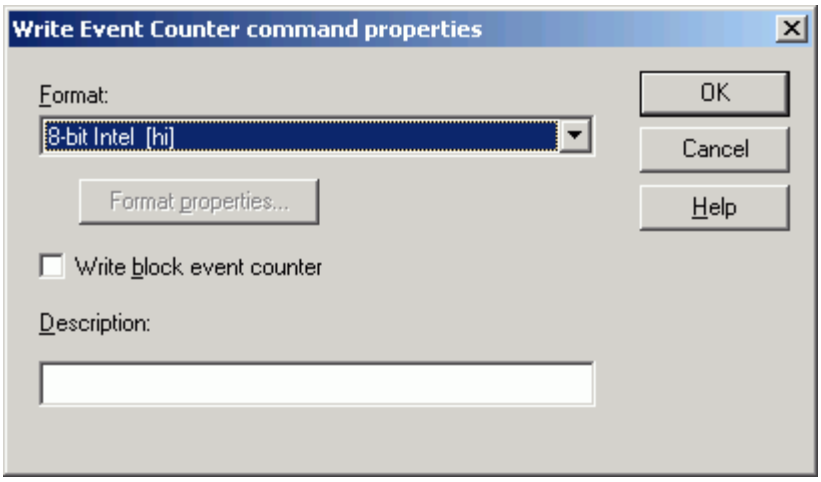
- The **Format** drop list is used to select the **device data format** in which the ID will be written. The **Format Properties** button will become enabled if the selected format has properties that must be set.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

Note: Users may **hard code** a Device ID using **Write Character** or **Write String** commands; however, those Device IDs would not be dynamic. If a **Write Device ID** command is used in all of the transactions, changing a Device ID is as simple as bringing up the server's Device Properties and changing the ID. The change will automatically take effect in all transactions associated with the device.

Write Event Counter Command

The **Write Event Counter** command tells the driver to append the value of the event counter to the write buffer. This makes possible the use of the event count value as a Transaction ID in serial communication packets.

To add a **Write Event Counter** command, simply right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Write Event Counter** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Write Event Counter** from the main menu. The dialog shown below will be displayed.



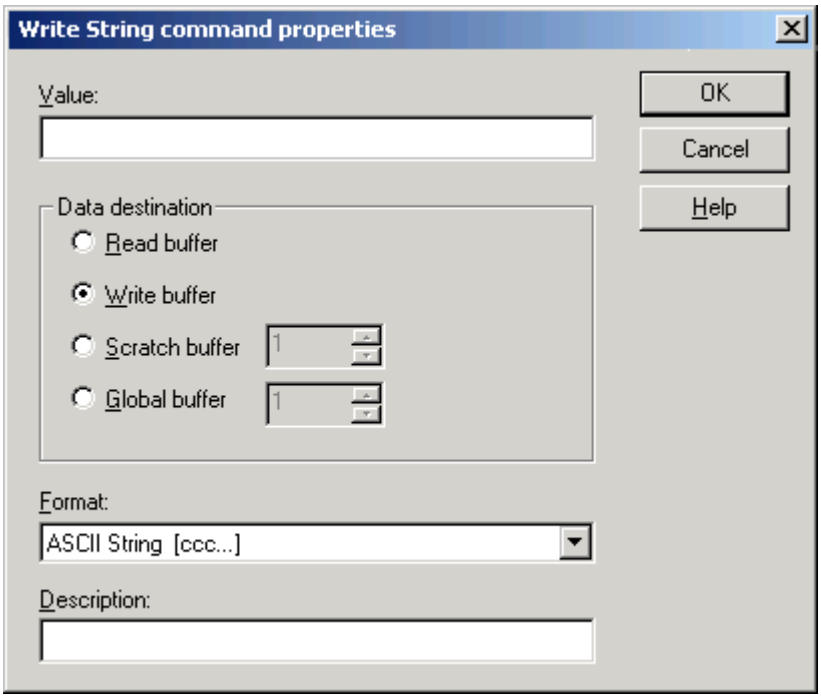
- The **Format** drop-down box is used to define the format of the Event Counter. The **Format Properties** button will become enabled if the selected format has properties that must be set.
- Click the **Write block event counter** checkbox if the event counter is from a block transaction. **Write block event counter** should be left unchecked if the event counter is from a regular transaction.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can also be very helpful when reviewing the transaction definition later.

See Also: **Event Counters** and **Set Event Counter Command**.

Write String Command

The Write String command tells the driver to append a string of ASCII characters to the write buffer, read buffer, a scratch buffer or a global buffer. Printable characters, such as letters, number and punctuation marks may be used only. To add a control-character or some other non-printable character, use the **Write Character** command.

To add a Write String command, right-click on the desired step in the **Transaction View** and then select **Write Commands** | **Write String** from the resulting pop-up menu. Alternatively, select **Edit** | **New Write Command** | **Write String** from the main menu. In either case, the dialog should appear as shown below.



- The **Value** box is where the series of characters to be appended to the buffer may be entered. The string may be of any length. A NULL terminator will not be assumed; only the characters explicitly entered will be appended to the buffer.
- The **Data** destination radio buttons is used to select the destination: **Read buffer**, **Write buffer**, **Scratch buffer** or **Global buffer**. If the Scratch or Global buffer options are selected, users must also specify the **buffer index** in the box to the right. If there are not enough bytes of data in the buffer, this command will be aborted and the transaction will fail, and an error message will be placed in the OPC Server's event log.

  Note: Data will be appended to TX and RX buffers, but not scratch or global buffers. To append data to the current contents of a scratch or global buffer, copy that data to either the RX or TX buffer. Then append that buffer and copy the contents back to the scratch or global buffer. **See Also:** **Copy Buffer Command**.

- The **Format** drop-down list is used to select the format (ASCII [default], Unicode, Alternating Byte ASCII), or ASCII Hex String From Nibbles*.
- The **Description** box is used to enter notations that will be displayed next to the command type in the Transaction View. Although descriptions are optional, they can be very helpful when reviewing the transaction definition later.

*For ASCII Hex String From Nibbles, only even numbers of characters are allowed. Furthermore, only hex characters ('0'-'9' and ''A'-'F') are allowed. Characters 'a'-'f' are automatically converted to valid hex 'A'-'F' by the Driver.

Unsolicited Transactions

An unsolicited transaction is a set of commands that is to be carried out when the driver receives a particular type of unsolicited message. (The driver will ignore unsolicited data unless it is configured to be in unsolicited mode.) Unlike with normal **query/receive** transactions, the driver does not have the benefit of knowing ahead of time what device and tag it is dealing with. Instead, the driver must determine from the message itself what device it came from and what tag the data should be sent to. To facilitate this, the user must define **unsolicited transaction keys**.

Unsolicited Transaction Keys

An unsolicited transaction key is a series of ASCII characters (or binary bytes) that match the first few characters of the message type the transaction is intended for. It is the user's responsibility to ensure that there is a one-to-one relationship between all of the transaction keys and all of the possible message types associated with a given channel.

For example, assume two devices are on a channel dedicated to unsolicited communication. Further, assume that these devices use the same, simple protocol. Suppose our hypothetical protocol has two possible unsolicited message types of the form:

[@] [A] [Device ID high digit] [Device ID low digit] [data] [data] [data] [data] [^M]
[@] [B] [Device ID high digit] [Device ID low digit] [data] [data] [^M]

where each character is surrounded with square brackets for notational clarity. If the two devices are configured as device 01 and 02, we have four possible message types that could be received on this channel:

[@] [A] [0] [1] [data] [data] [data] [data] [^M]
[@] [B] [0] [1].[data] [data] [^M]
[@] [A] [0] [2].[data] [data] [data] [data] [^M]
[@] [B] [0] [2].[data] [data] [^M]

To process all four possible message types, we need to define a channel using the U-CON (User-Configurable) Driver in unsolicited mode. Next, we need to add two devices to that channel. The transaction editor must be used to create two tags for each device, one tag for the @A messages and another for the @B messages. Each of these tags will be created with an unsolicited transaction that must be defined by the user. The complete definition of an unsolicited transaction consists of two things, the transaction key, and the series of commands that are required to receive and process the message. We will consider the transaction keys first.

For this hypothetical protocol, we need to look at the first four bytes of an incoming message to know which transaction should be used to process it. Thus, the four transaction keys need to be assigned as:

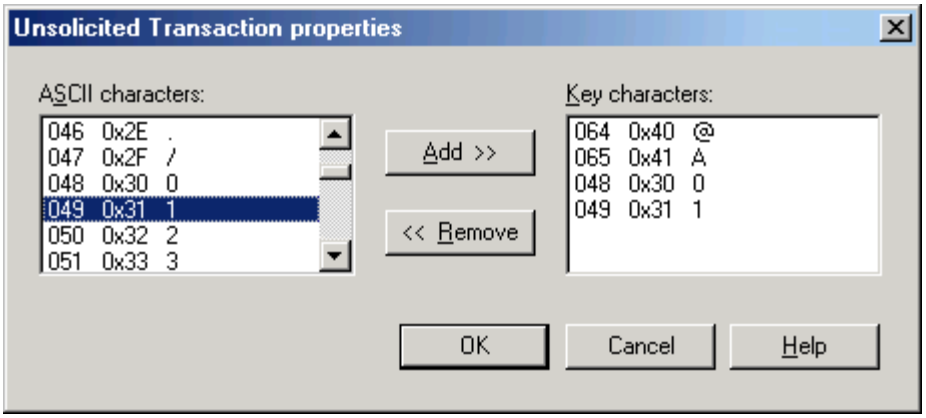
DEVICE TAG KEY
---
01 A @A01
01 B @B01
02 A @A02
02 B @B02

If we made our keys only three characters long in this example, there would be an ambiguous message-to-transaction relationship. The driver would have no way of knowing which device the data came from since this is indicated by the fourth character in the messages. If we made one of the keys longer than four bytes, it would extend into the variable data portion of some (in this case all) of the messages. Such a key would only be matched by pure coincidence depending on the data value.

With **normal** (not unsolicited) communication, it is generally not possible to place devices using different protocols on the same channel. It is possible to mix protocols on an unsolicited channel, so long as the transaction keys are of the same length and are unique.

In practice, a tag and its unsolicited transaction does not need to be defined for every possible message on a channel. The only constraint is that defined tags' unsolicited transactions have keys that are specific enough to match only the message types that users want to process.

To define the **unsolicited** transaction key, bring up the Transaction Editor and double-click on the unsolicited transaction item (or select the transaction and then select properties from the main menu, the transaction's pop-up menu, or the toolbar). The **unsolicited transaction key editor** should then appear as shown below.



To define the transaction key, simply double-click on the desired ASCII character in the left **ASCII characters** box or select it and click **Add >>**. The key character sequence will appear in the right **Key characters** box. If a mistake is

made, the **<< Remove** button can be used to remove selected items in key characters box. The number of characters that must be entered was set when the channel was defined. All unsolicited transaction associated with a given channel must have the same key length. **See Also:** Configuration.

Note: In the case of multiple unsolicited devices on a single channel, the Device ID must be **hard coded** into the transaction key. Therefore, the Device ID as set in the server's device property page has no bearing on how incoming data is sorted out to the various tags. Make sure that the IDs configured in the physical devices match the corresponding fields in the transaction keys at all times.

In cases where the protocol does not lend itself to use of such keys, this driver can still be used. A scanner that sends packets starting with the raw data values would be an example. In these cases, the transaction key length must be set to zero. This will force the driver to use the first unsolicited transaction defined on the channel to interpret all incoming packets. Because of this, there should be only one device on the channel. Furthermore, that device should have a single block tag or a single non-block tag defined. That tag or tag block may be placed in a group.

All tags belonging to an unsolicited channel will have an initial value of zero. Client applications will see this initial value until the first unsolicited update for that tag is received by the driver.

Commands in Unsolicited Transactions

Although an unsolicited transaction may start with comments and/or insert function block, the first executable command must be a **Read Response** command. This is so the driver will know where the end of the current message should be. After the **Read Response** command, almost any other command type can be placed. However, a second **Read Response** should not be issued in an unsolicited transaction, because it would imply that users know what the next message received on the channel will be. This is generally a bad assumption when dealing with unsolicited communications.

See Also: "Unsolicited Message Wait Time" in **Device Setup**.

Updating the Server

Once all work within the **Transaction Editor** is finished, users must transfer the updates to the server. To do so, select the Transaction Editor's main menu option **File | Update Server**. Alternatively, click on the **Update Server** icon on the toolbar. Users will be given a chance to update the server when the Transaction Editor is closed. After the server has received the device profile updates, it will automatically invoke the tag database generation feature. All of the new tags and groups will instantly appear on the server. Any tags and groups removed during the transaction edit session will be removed from the server. At this point, the Transaction Editor will shut itself down. To resume communication, reconnect the client application to the device.

Device Data Formats

The U-CON (User-Configurable) Driver offers a large set of device data format options which describe how data values will be transmitted between the driver and the device. This should not be confused with the data type, which describes the binary format of data as transmitted between the client and server applications. The device and protocol determine the device data format. Care should be taken to choose a compatible tag data type. The combination of data type and format determines the range of values that can be transmitted. Truncation errors are possible with many combinations.

Binary Formats
ASCII Formats
ASCII Hex Formats
Date
Legend

Binary Formats

Format	Data Length	Notes
8-bit Intel [hi]	1	**Example:** The value 10 (0x0A) would be encoded as a single byte 0x0A.
16 bit Intel [lo hi]	2	**Example:** The value 258 (0x0102) would be encoded as two bytes 0x02 0x01.
16 bit Motorola [hi lo]	2	**Example:** The value 258 (0x0102) would be encoded as two bytes 0x01 0x02.
24-bit Motorola [Hilo LOhi Lolo]	3	**Example:** The value 66051 (0x010203)

		would be encoded as three bytes 0x01 0x02 0x03.
32-bit Intel [LOlo LOhi HIlo HIhi]	4	**Example:** The value 16909060 (0x01020304) would be encoded as four bytes 0x04 0x03 0x02 0x01.
32-bit Intel (word swap) [HIlo HIhi LOlo LOhi]	4	**Example:** The value 16909060 (0x01020304) would be encoded as four bytes 0x02 0x01 0x04 0x03.
32-bit Motorola [HIhi HIlo LOhi LOlo]	4	**Example:** The value 16909060 (0x01020304) would be encoded as four bytes 0x01 0x02 0x03 0x04.
32-bit Motorola (word swap) [LOhi LOlo HIhi HIlo]	4	**Example:** The value 16909060 (0x01020304) would be encoded as four bytes 0x03 0x04 0x01 0x02.
32-bit IEEE float	4	Also known as single precision real. **Example:** The value 1.23456 would be encoded as four bytes 0x3F 0x9E 0x06 0x10
32-bit IEEE float (byte swap)	4	Similar to 32-bit IEEE float, but in byte swapped order. **Example:** The value 1.23456 would be encoded as four bytes 0x9E 0x3F 0x10 0x06
32-bit IEEE float (word swap)	4	Similar to 32-bit IEEE float, but in word swapped order. **Example:** The value 1.23456 would be encoded as four bytes 0x06 0x10 0x3F 0x9E
32-bit IEEE float (reversed)	4	Similar to 32-bit IEEE float, but with bytes in reverse order (word and byte swap). **Example:** The value 1.23456 would be encoded as four bytes 0x10 0x06 0x9E 0x3F
64-bit IEEE float	8	Also known as double precision real **Example:** The value 1.234567 would be encoded as eight bytes 0x0A 0x4A 0xD1 0xCA 0xBD 0xC0 0xF3 0x3F
1-byte packed BCD	1	Integers between 0-99 are encoded as Binary Coded Digits data. Behavior is undefined for values beyond this range. **Example:** The value 12 would be encoded as a single byte 0x12.
2 byte packed BCD	2	Integers between 0-9999 are encoded as Binary Coded Digits data. Behavior is undefined for values beyond this range. **Example:** The value 1234 would be encoded as two bytes 0x12 0x34.
2 byte packed BCD (byte swap)	2	Similar to 2 byte packed BCD, but in byte swapped order. **Example:** The value 1234 would be encoded as two bytes 0x34 0x12.
4 byte packed BCD	4	Integers between 0-99999999 are encoded as Binary Coded Digits data. Behavior is undefined for values beyond this range. **Example:** The value 12345678 would be encoded as four bytes 0x12 0x34 0x56 0x78.

4 byte packed BCD (byte swap)	4	Similar to 4 byte packed BCD, but in byte swapped order. **Example:** The value 12345678 would be encoded as four bytes 0x34 0x12 0x78 0x56.
4 byte packed BCD (word swap)	4	Similar to 4 byte packed BCD, but in word swapped order. **Example:** The value 12345678 would be encoded as four bytes 0x56 0x78 0x12 0x34.
4 byte packed BCD (reversed)	4	Similar to 4 byte packed BCD, but with bytes in reverse order (word and byte swap). **Example:** The value 12345678 would be encoded as four bytes 0x78 0x56 0x34 0x12.
Bit 0 from byte [00000001] Bit 1 from byte [00000010] Bit 2 from byte [00000100] Bit 3 from byte [00001000] Bit 4 from byte [00010000] Bit 5 from byte [00100000] Bit 6 from byte [01000000] Bit 7 from byte [10000000]	1	When reading, a whole byte is received from the device. The state (0 or 1) of the specified bit is then passed to the tag. When writing, a whole byte is sent to the device. The specified bit is set if the write value is non-zero, all other bits will be zero. **Example:** Receive the byte 0x01, would cause a tag with Bit 0 format take a value of 1 or TRUE. Tags with any other bit format would take a value of 0 or FALSE. **Example:** Write the value 1 (or any other non-zero value), would result in the byte 0x01 being sent if Bit 0 format, 0x02 if Bit 1 format, etc.
Multi-Bit Integer	1, 2, or 4	When reading, a whole 8, 16, or 32 bit integer is received from the device. The equivalent integer value of a subset of the bits within this data is then passed to the tag. When writing, a whole 8, 16, or 32 bit integer is sent to the device. The specified bits will be set to the binary equivalent of the write value, with all other bits set to zero. If the write value exceeds the maximum value that can be represented by the specified number of bits, the specified bits will all be set to one. For Boolean data types, all specified bits are set to one if the write value is non-zero with all other bits being a zero. **See Also:** **Format Multi-Bit Integer**

ASCII Formats

Format	Data Length	Notes
ASCII Integer [+ddd]	F/V/D	Integer values encoded as ASCII strings. **See Also:** **Format ASCII Integer**
ASCII Integer Hex [hhh]	F/V/D	Integer values encoded as ASCII hex strings. **See Also:** **Format ASCII HEX Integer**
ASCII Real [+ddd.dddE+ddd]	F/V/D	Real (or floating point) values encoded as ASCII strings.

		See Also: **Format ASCII Real**
ASCII String [ccc...]	F/V/D	Strings encoded as ASCII characters. **See Also:** **Format ASCII String**
ASCII Multi-Bit Integer [xxxxxxxx]	8	The 8 bits in a byte value are represented as a string of 8 ASCII "0" or "1" characters. **See Also:** **Format ASCII Multi-bit Integer**
ASCII String - Alternating Byte [0 c 0 c ....]	F/V/D	Strings encoded as ASCII characters where each of the characters is preceded by a character containing 0 (zero). For example, the string "TEST" will be 0x00 0x54 0x00 0x45 0x00 0x53 0x00 0x54 in this format. **See Also:** **Format Alternating Byte ASCII String**
ASCII Hex String From Nibbles [hh hh hh...]	F/V/D	Nibbles encoded as ASCII hex strings. **See Also:** **Format ASCII Hex String From Nibbles**
Unicode String [u1u2u3u4...]	F/V/D	Strings encoded in the Unicode format. **See Also:** **Format Unicode String**
Unicode String with Lo Hi Byte Order [u2u1u4u3...]	F/V/D	Strings encoded in the Unicode format with the order reversed – Lo Hi (Least significant byte first). **See Also:** **Format UnicodeLoHi String**
Byte from 2 Offset Nibble chars	2	The value is represented as two ASCII characters with values: [low nibble + 0x30] [high nibble + 0x40]. **Example:** The value 168 (0xA8) is represented as the characters "8J". (Low nibble = 0x08, 0x08 + 0x30 = 0x38 = "8". High nibble = 0x0A, 0x0A + 0x40 = 0x4A = "J".)
Float from 8 Offset Nibble chars	8	The value is represented as an IEEE float with reversed byte order, where each byte is encoded using the "Byte from 2 Offset Nibble chars" format described above. **Example:** The value 1.23456, which is 0x3F9E0610 in normal IEEE form and 0x10069E3F in reversed byte order form, would be encoded as the characters "0A6@>I?C". (Low nibble of first byte = 0x00, 0x00 + 0x30 = 0x30 = "0". High nibble of first byte = 0x01, 0x01 + 0x40 = 0x41 = "A". The other three bytes are encoded in a similar manner.)
Use dynamic ASCII format table	V	This format option is provided for devices that represent values as a fixed number of ASCII digits and a format character that specifies the decimal placement and sign. To use this option, the user must define a table of format characters. **See Also:** **Dynamic ASCII Formatting**
ASCII String (packed 6 bit) [cccc...]	F/V	Strings encoded as ASCII (packed 6 bit) characters.

		See Also: Format ASCII String (packed 6 bit)
ASCII Integer (packed 6 bit) [+dddd...]	F/V	Strings encoded as ASCII (packed 6 bit) characters. **See Also:** Format ASCII Integer (packed 6 bit)
ASCII Real (packed 6 bit) [+ddd.dddE+ddd]	F/V	Strings encoded as ASCII (packed 6 bit) characters. **See Also:** Format ASCII Real (packed 6 bit)

ASCII Hex Formats

Format	Data Length	Notes
NIBBLE from 1 ASCII Hex char [h]	1	**Example:** The value 10 (0x0A) would be sent as a single ASCII Hex character "A" (0x41).
Byte from 2 ASCII Hex chars [hh]	2	**Example:** The value 26 (0x1A) would be sent as two ASCII Hex characters "1A" (0x31 0x41).
Byte from 2 ASCII Hex chars (LC) [hh]	2	**Example:** The value 26 (0x1A) would be sent as two lower-case ASCII Hex characters "1a" (0x31 0x61).
Word from 4 ASCII Hex chars [hh hh]	4	**Example:** The value 4666 (0x123A) would be sent as four ASCII Hex characters "123A" (0x31 0x32 0x33 0x41).
Word from 4 ASCII Hex chars (LC BS) [hh hh]	4	**Example:** The value 4666 (0x123A) would be sent as four lower-case ASCII Hex characters with bytes swapped "3a12" (0x33 0x61 0x31 0x32).
DWORD from 8 ASCII Hex chars [hh hh hh hh]	8	**Example:** The value 305419898 (0x1234567A) would be sent as eight ASCII Hex characters "1234567A" (0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x41).
ASCII Hex String [hh hh hh...]	F/V/D	Strings encoded as ASCII Hex values. **Example:** The string "AB12" would be sent as eight ASCII Hex characters "AB12" (0x34 0x31 0x34 0x32 0x30 0x31 0x30 0x32). **See Also:** Format ASCII Hex String
Bit 0 from 2 ASCII Hex chars [hh] Bit 1 from 2 ASCII Hex chars [hh] Bit 2 from 2 ASCII Hex chars [hh] Bit 3 from 2 ASCII Hex chars [hh] Bit 4 from 2 ASCII Hex chars [hh] Bit 5 from 2 ASCII Hex chars [hh] Bit 6 from 2 ASCII Hex chars [hh] Bit 7 from 2 ASCII Hex chars [hh]	2	When reading, two ASCII Hex values are received from the device. They are converted to a Byte, and the state (0 or 1) of the specified bit is sent to the tag. When writing, the specified bit in a Byte is set if the write value is non-zero, all other bits will be zero. The Byte is converted to 2 ASCII Hex characters and sent to the device. **Example:** Receive the bytes 0x30 0x31 (the binary value 0x01), would cause a tag with Bit 0 format take a value of 1 or TRUE. Tags with any other bit format would take a value of 0 or FALSE. **Example:** Write the value 1 (or any other non-zero value), would result in the bytes 0x30 0x31 (the binary value 0x01) being sent if Bit 0 format, 0x30 0x32 (the binary value

		0x02) if Bit 1 format, etc.
ASCII coded IEEE float [hh hh hh hh]	8	This format option is provided for devices that encode each nibble of a 32-bit IEEE float value as an ASCII Hex character. **Example:** The binary representation of 1.0 is 0x3F800000. This value would be encoded as an 8 character string "3F800000". This value would be sent as eight ASCII Hex characters "3F800000" (0x33 0x46 0x38 0x30 0x30 0x30 0x30 0x30). This format is not to be confused with "ASCII Real" described above which would send this value as a 3 character string "1.0".
ASCII coded IEEE float (LC) [hh hh hh hh]	8	This is the same as ASCII Coded IEEE float, except lower-case ASCII hex characters are used. **Example:** The value 1.0 (0x3F800000) would be sent as: "3f800000" (0x33 0x66 0x38 0x30 0x30 0x30 0x30 0x30).
ASCII coded IEEE float (Rev) [hh hh hh hh]	8	This is the same as ASCII Coded IEEE float, except the byte order is reversed. **Example:** The value 1.0 (0x3F800000) would be sent as: "0000803F" (0x30 0x30 0x30 0x30 0x38 0x30 0x33 0x46).
ASCII coded IEEE float (LC Rev) [hh hh hh hh]	8	This is the same as above, except lower-case ASCII hex characters are used, and the byte order is reversed. **Example:** The value 1.0 (0x3F800000) would be sent as: "0000803f" (0x30 0x30 0x30 0x30 0x38 0x30 0x33 0x66).

Date

Format	Data Length	Notes
Short Date [MM/DD/YYYY]	6	*
Short Date [MM/DD/YY]	5	*
Short Date [DD/MM/YYYY]	6	*
Short Date [DD/MM/YY]	5	*
Time [HH:MM:SS]	5	*
Standard [MM/DD/YYYY HH:MM:SS]	12	*

*For more information, refer to **Format Date Time**.

LEGEND

h = ASCII Hex digit ("0" to "F")
d = ASCII decimal digit ("0" to "9")
x = ASCII binary digit ("0" or "1")
c = ASCII character
LO = Low Word
lo = Low byte in a Word
HI = High Word
hi = High byte in a Word
0 = low binary bit
1 = high binary bit
+ = Optional sign ("+" or "-")
F = Fixed data length support

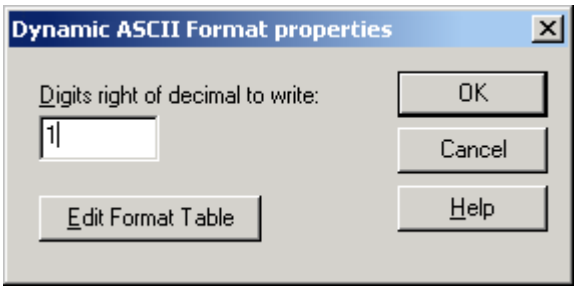
V = Variable data length support
D = Delimited list support

See Also: **Delimited lists**.

Dynamic ASCII Formatting

Many ASCII devices utilize a formatting scheme where values are represented by a fixed number of ASCII digits and a format character. No decimal point or sign characters are used. Instead, the format character determines decimal placement and sign. For example, a device may represent the value -12.3 as 0123D where D means multiply the transmitted integer value, 123 in this case, by -0.1. The format character is dynamic, meaning that it could be different for each read and write transaction, depending on the data value.

The **Use Dynamic ASCII Format Table** **device data format** option tells the driver to use this type of formatting. By clicking on the **Format Properties** button on the tag dialog, the following dialog will come up.



In this dialog, users can specify how many digits to the right of the decimal point should be used when writing to the device. Most devices that utilize this type of formatting, zero digits are expected for integer types, and a specific non-zero number is expected for real types. For example, the value 1.2 could possibly be represented as 1200A, 0120B, or 0012C, where A means multiply by 0.001, B 0.01, and C 0.1. However, the device may only accept 0012C for a particular register. In this case, users would set the number of digits to right of decimal to 1 to force the driver to choose the C format. In general, if the device is expecting an integer, this value should be 0. When attempting a read, the value has no significance. The driver simply parses the format character from the read buffer, looks up its corresponding multiplier and then converts the data digits accordingly.

In order for this option to work, the user must also define a table of format characters and their corresponding multipliers. Such a table must be defined for each device that uses this format option. To edit the table, click on the **Edit Format Table** button, or select **Edit Dynamic ASCII Format Table** from the main menu or device pop-up menu.

The **Dynamic ASCII Format Table editor**, shown below, includes a list of formats currently defined for the device. Clicking on a table entry will select it; double-clicking will bring up a dialog that can be used to edit the format item. To the left of the format list are three buttons. The top one is used to add a new format to the table. The middle and bottom ones allow users to edit or delete the selected format item respectively. There must be a one-to-one relationship between each format character and multiplier. In addition to the format characters, users must specify the number of data characters the device uses and whether the format character will precede or follow the data.

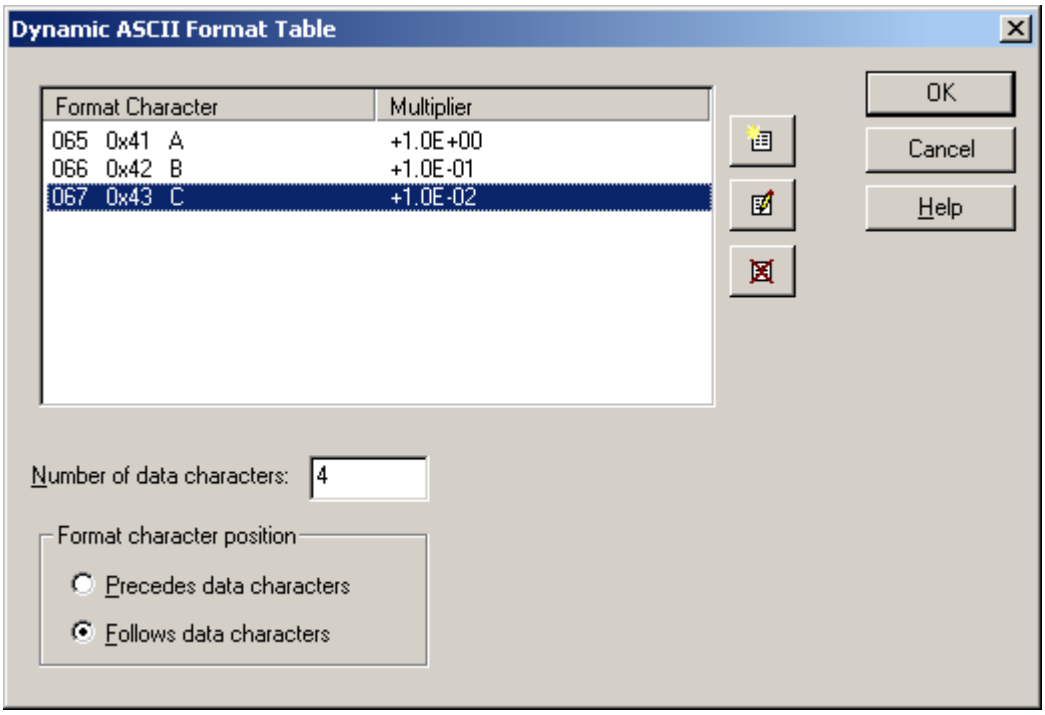
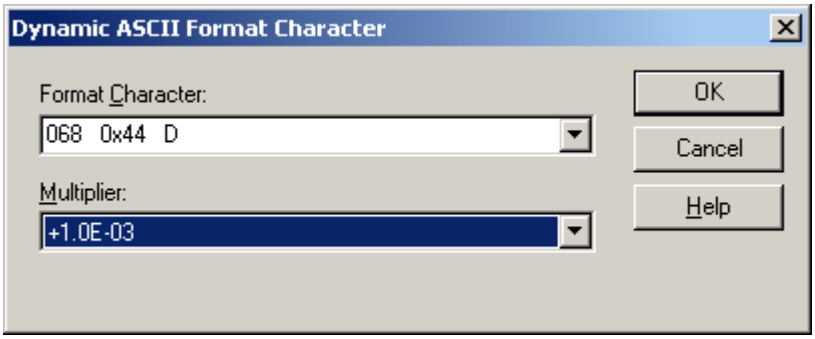
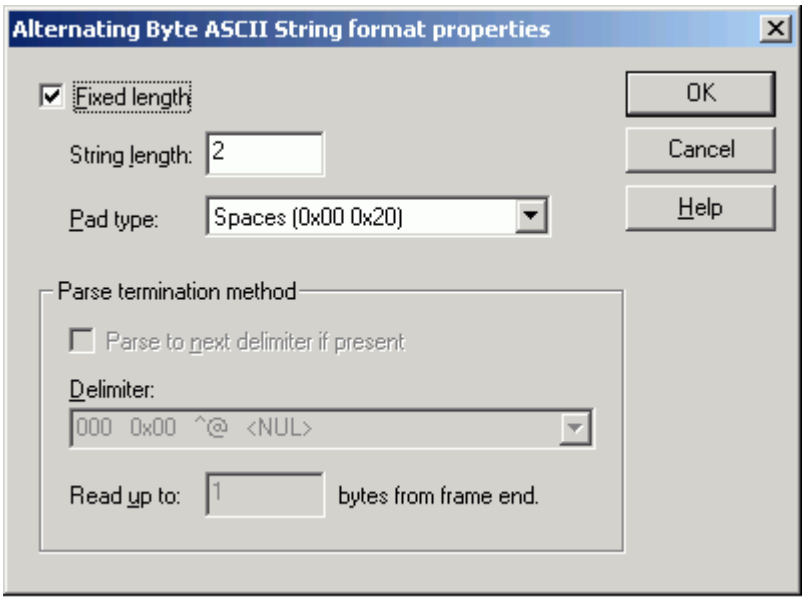


Table entries are edited using the following dialog.



Format Alternating Byte ASCII String

The Alternating Byte ASCII String **device data format** option can be used to define the format of string data. For example, when the **Alternating Byte ASCII String [0 c 0 c ....]** format is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

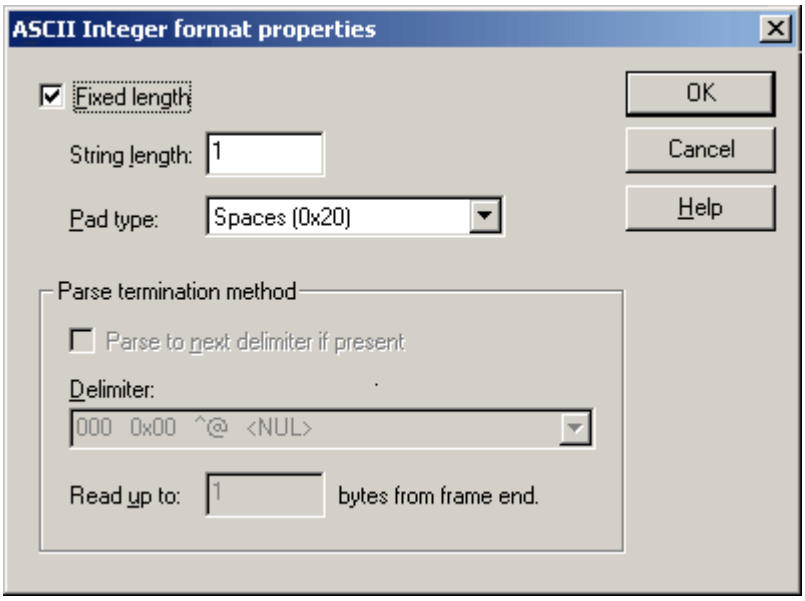
- For fixed length strings, the **String length** must be set. The number entered sets the total number of characters (two bytes per character) that will be written to or read from the device. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for values that do not require the full string length. Unlike ASCII strings, each pad character is encoded as two ASCII bytes (high byte 0). For example, if the string length was set to 8 and **Spaces** was chosen as the pad type, writing the string **ABC** would cause the driver to send 0x00 0x41, 0x00 0x42, 0x00 0x43, 0x00, 0x20. There are many options for pad characters: spaces (0x00 0x20), zeros (0x00 0x30), and NULL (0x00 0x00). The pad character option applies to writes only: the driver can read any valid ASCII hexadecimal string of the specified length.

For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. Either of the following can be used for variable length ASCII data:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF may be chosen. The driver will search for this character as ASCII hexadecimal data. For example, the two bytes 0x00 0x20 would be considered a space character.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

Format ASCII Integer

The ASCII integer **device data format** option allows the user to specify how ASCII integer data should be formatted. For example, when a format of **ASCII Integer [+ddd]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length ASCII integer strings, the **String length** must be specified. As its name suggests, this sets the total number of characters (one byte per character) that will be written to or read from the device. A minus sign counts as one character.
- For fixed length ASCII integer strings, the **Pad type** must be specified. Pad characters are used to fill out the string for integer values that do not require the full string length. For example, if the string length was set to 4, and a value of 12 is to be written to the device, the driver will create a string consisting of two pad characters, followed by 1 then 2. There are many options for pad characters: spaces (0x20), zeros (0x30), and NULL (0x00). The pad character option applies to writes only: the driver can read any valid ASCII integer string of the specified length.

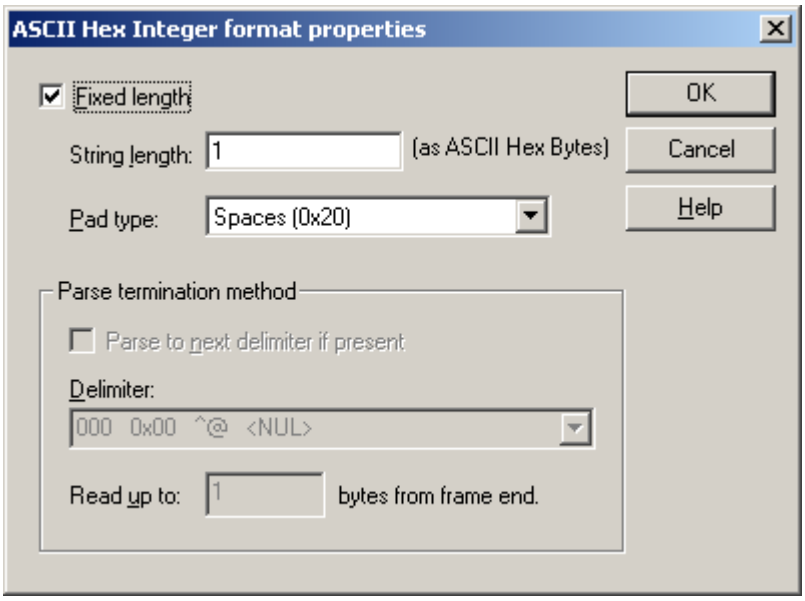
For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF can be chosen.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

ROUND OFF: When writing values that require more characters than allotted by String length, the driver will write the largest positive or smallest negative value that can be expressed in the allotted space. For example, if string length is set to 4, then writing 12345 results in the string 9999 and writing -1234 results in the string -999.

Format ASCII HEX Integer

The ASCII Hex Integer **device data format** option allows the user to specify how ASCII hex integer data should be formatted. For example, when a format of **ASCII Hex Integer [hhh]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines if the string data is fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length ASCII hex integer strings, the **String length** must be specified. As its name suggests, this sets the total number of characters (one byte per character) that will be written to or read from the device. A minus sign counts as one character.
- For fixed length ASCII hex integer strings, the **Pad type** must be specified. Pad characters are used to fill out the string for integer values that do not require the full string length. For example, if the string length was set to 4 and a value of 12 is to be written to the device, the driver will create a string consisting of two pad characters, followed by 1 then 2. There are many options for pad characters: spaces (0x20), zeros (0x30), and NULL (0x00). The pad character option applies to writes only: the driver can read any valid ASCII integer string of the specified length.

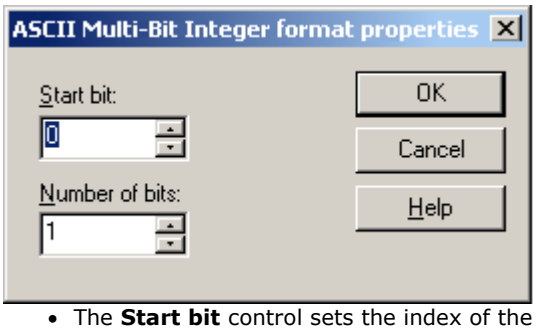
For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF can be chosen.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

ROUND OFF: When writing values that require more characters than allotted by String length, the driver will write the largest positive or smallest negative value that can be expressed in the allotted space. For example, if string length is set to 4, then writing 12345 results in the string FFFF and writing -1234 results in the string FFFF.

Format ASCII Multi-Bit Integer

The ASCII Multi-Bit Integer **device data format** option reads or writes a specified number of bit characters represented in an ASCII multi-bit integer. An ASCII multi-bit integer is an 8 character long string, where each character can be either 0 or 1. This format option requires the user to specify two Format Properties, the start bit, and number of bits. For example, when a format of **ASCII Multi-Bit Integer [xxxxxxxx]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



- The **Start bit** control sets the index of the first bit that the driver will read from or write to. As is standard practice, the least significant bit (LSB) is referred to as bit index 0, and the most significant bit (MSB) has a bit index of 7.
- The **Number of bits** control sets how many bits to read or write, starting at the start bit index.

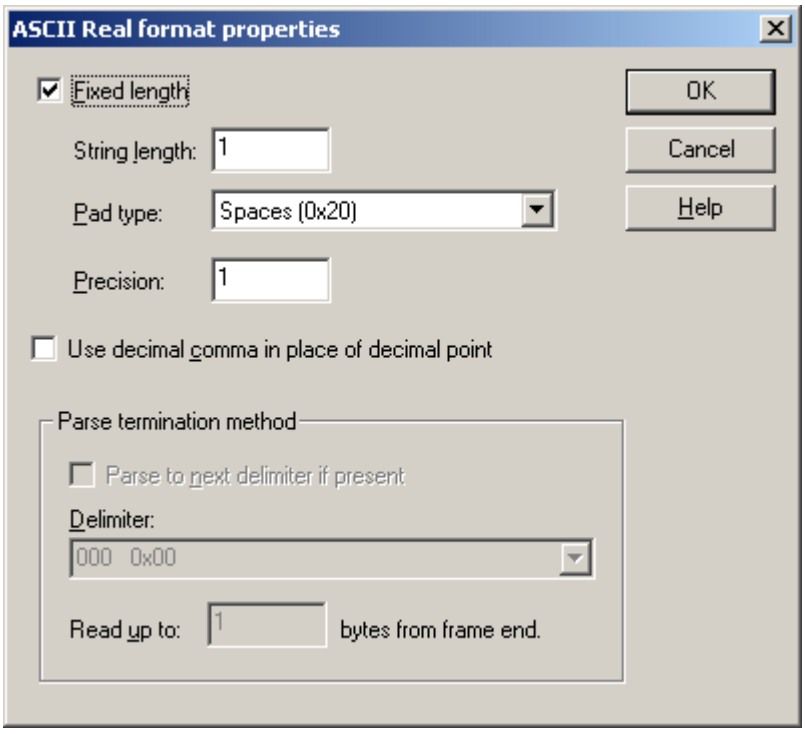
  If a value is to be written that exceeds the maximum value that the can be represented by the specified number of bits, then all of the specified bits will be set to one. All bits other than those specified by this format will be set to zero in writes. If this format is used with a Boolean data type, then all specified bits are set to one, if the write value is non-zero. If wishing to set a number of bits in a predefined byte, preserving the state of the other bits, use another device data format and the **Modify Byte** command.

Read Example
Say the device returns 11001010, and this format specifies a start bit of 3 and number of bits of 4. The value returned to the tag is 9 decimal (1001 binary).

Write Example
Say a value of 1 is to be written, and this format specifies a start bit of 3 and number of bits of 2. The value sent to the device will be 00001000. If a value of 3 or greater is to be written using the same Format Properties, then the value sent to the device will be 00011000.

Format ASCII Real

The ASCII Real **device data format** option allows the user to specify how ASCII Real data should be formatted. For example, when a format of **ASCII Real [+ddd.dddE+ddd]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines if string data is fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length ASCII real strings, the **String length** must be specified. As its name suggests, this sets the total number of characters (one byte per character) that will be written to or read from the device. The decimal point and possible minus sign each count as one character.
- For fixed length ASCII real strings, the **Pad type** must also be specified. Pad characters are used to fill out the left hand side of the string for real values that do not require the full string length. Zeros are added as needed to fill out the specified precision. For example, if the string length was set to 8, and the precision was set to 3, and a value of 12.3 is to be written to the device, the driver will create a string consisting of two pad characters, followed by "12.300". There are many options for pad characters: spaces (0x20), zeros (0x30), and NULL (0x00). The pad character option applies to writes only: the driver can read any valid ASCII real string of the specified length.

Precision sets the number of digits to the right of the decimal point. The precision property applies to writes only: the driver can read any valid ASCII real value of the specified length.

Use decimal comma in place of decimal point allows for use of a comma as the decimal separator.

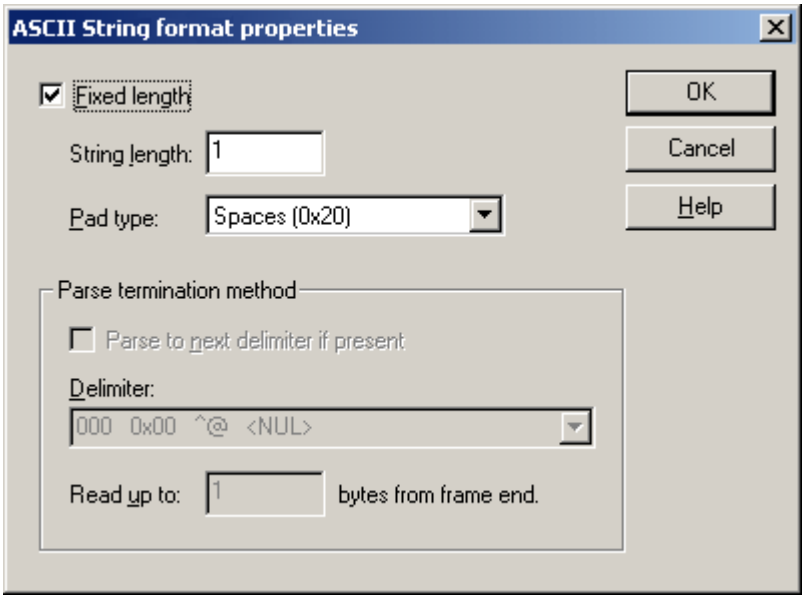
For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF can be chosen.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

ROUND OFF: When writing values that require more characters than allotted by String length, the driver will write the largest positive or smallest negative value that can be expressed in the allotted space. For example, if string length is set to 6 and the precision is set to 2, then writing 1234.567 results in the string "999.99" and writing -123.456 results in the string "-99.99".

Format ASCII String

The ASCII String **device data format** option allows the user to specify how string data should be formatted. When a format of **ASCII String [ccc...]** is selected, in the **tag dialog** for example, the **Format Properties** button will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length strings, the **String length** must be set. The number entered here sets the total number of characters (one byte per character) that will be written to or read from the device. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for values that do not require the full string length. Unlike ASCII integer and ASCII real formats, the pad characters are added as needed to the right. For example, if the string length was set to 4, and a value of **ABC** is to be written to the device, the driver will create a string consisting of the characters, ABC, followed by one pad character. There are many options for pad characters: spaces (0x20), zeros (0x30), and NULL (0x00). The pad character option applies to writes only: the driver can read any valid ASCII string of the specified length.

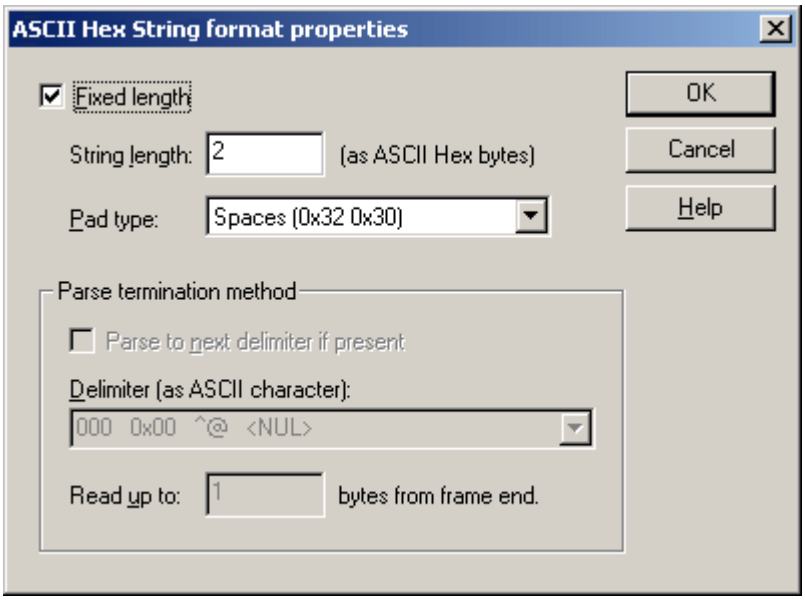
For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF can be chosen.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

See Also: **Tips and Tricks: Delimited Lists**

Format ASCII Hex String

The ASCII Hex String **device data format** option allows the user to specify how string data should be formatted. For example, when a format of **ASCII Hex String [hh hh hh...]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

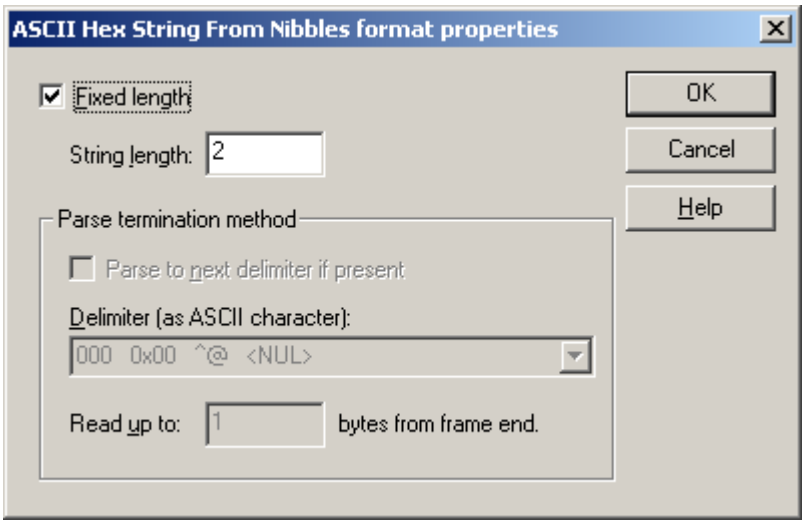
- For fixed length strings, the **String length** must be set. The number entered sets the total number of characters (two bytes per character) that will be written to or read from the device. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for values that do not require the full string length. Unlike ASCII strings, each pad character is encoded as two ASCII Hex bytes. For example, if the string length was set to 8 and **Spaces** was chosen as the pad type, writing the string **ABC** would cause the driver to send eight bytes 0x34 0x31 0x34 0x32 0x34 0x33 0x32 0x30. There are many options for pad characters: spaces (0x32 0x30), zeros (0x33 0x30), and NULL (0x30 0x30). The pad character option applies to writes only: the driver can read any valid ASCII Hex string of the specified length.

For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF can be chosen. The driver will search for this character as ASCII hexadecimal data. For example, the two bytes 0x32 0x30 would be considered a space character.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

Format ASCII Hex String From Nibbles

The ASCII Hex String From Nibbles **device data format** option allows the user to specify how string data should be formatted. For example, when a format of **ASCII Hex String From Nibbles [hh hh hh...]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. Clicking this button should invoke the dialog shown below.



The **Fixed length** check box determines if string data is fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length strings, the String length must also be set. The number entered sets the total number of bytes (one byte per two characters) that will be written to or read from the device. Only even lengths are allowed. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, when writing through a client, the driver adds pad character ('0':0x30) at the end of the string up to the set length. For example, if the string length was set to 8, writing the string **ABC** would cause the driver to send four bytes 0xAB 0xC0 0x00 0x00 (for a driver recreated string ABC00000). The pad character option applies to writes only: the driver can read any valid ASCII Hex string of the specified length.

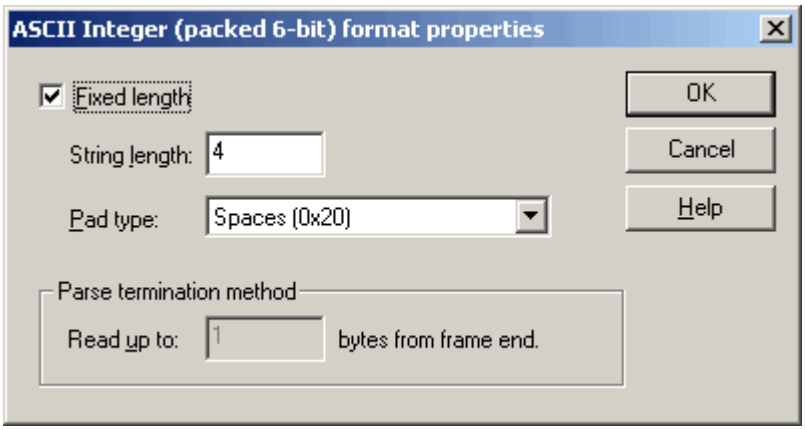
For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF can be chosen.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame. For variable length strings, when writing through a client, the driver adds a single pad character ('0':0x30) at the end of the string if the length of the string is odd. For example, writing the string ABC would cause the driver to send two bytes 0xAB 0xC0 (for a driver recreated string ABC0).

Note: When writing through a client only hex characters ('0'-'9' and ''A'-'F') are allowed. Characters 'a'-'f' are automatically converted to valid hex 'A'-'F' by the driver.

Format ASCII Integer (Packed 6 Bit)

The ASCII integer (packed 6 bit) **device data format** option can be used to specify how ASCII integer data should be formatted. For example, when a format of **ASCII Integer (packed 6 bit) [+dddd]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether the string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length ASCII integer (packed 6 bit) strings, the **String length** must be specified. As its name suggests, this sets the total number of characters (one byte per character) prior to conversion that will be written to or read from the device. A minus sign counts as one character. The number of bytes sent over the wire is equal to three fourths the String length.
- For fixed length ASCII integer (packed 6 bit) strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for integer values that do not require the full string length. For example, if the string length was set to 4 and a value of 12 is to be written to the device, the driver will create a string consisting of two pad characters followed by 1 then 2. There are many options for pad characters: spaces (0x20) and zeros (0x30). The pad character option applies to writes only: the driver can read any valid ASCII integer (packed 6 bit) string of the specified length.

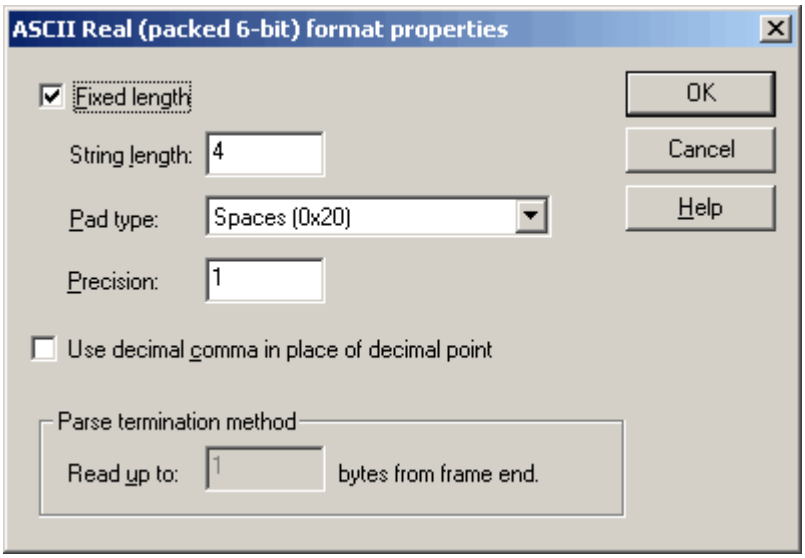
For **variable length ASCII integer (packed 6 bit) data,** the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This is accomplished by specifying an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

ROUND OFF: When writing values that require more characters than allotted by String length, the driver will write the largest positive or smallest negative value that can be expressed in the allotted space. For example, if string length is set to 4, then writing 12345 results in the string 9999 and writing -1234 results in the string -999.

Note: Due to packing, ASCII (packed 6 bit) data uses a reduced **ASCII (packed 6 bit) Character Table**. Attempting to use characters not in the **ASCII (packed 6 bit) Character Table** will result in data conversion failures.

Format ASCII Real (Packed 6 Bit)

The ASCII Real (Packed 6 Bit) **device data format** option can be used to specify how ASCII Real data should be formatted. For example, when a format of **ASCII Real (packed 6 bit) [+ddd.dddE+ddd]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether the string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length ASCII real (packed 6 bit) strings, the **String length** must be specified. As its name suggests, this sets the total number of characters (one byte per character) prior to conversion that will be written to or read from the device. The decimal point and possible minus sign each count as one character. The number of bytes sent over the wire is equal to three fourths the string length.
- For fixed length ASCII real (packed 6 bit) strings, the **Pad type** must also be specified. Pad characters are used to fill out the left hand side of the string for real values that do not require the full string length. Zeros are added as needed to fill out the specified precision. For example, if the string length is set to 8, the precision is set to 3, and a value of 12.3 is to be written to the device, the driver will create a string consisting of two pad characters followed by 12.300. There are many options for pad characters: spaces (0x20) and zeros (0x30). The pad character option applies to writes only: the driver can read any valid ASCII real (packed 6 bit) string of the specified length.

Precision sets the number of digits to the right of the decimal point. The precision property applies to writes only: the driver can read any valid ASCII real (packed 6 bit) value of the specified length.

Use decimal comma in place of decimal point allows a comma to be used as a decimal.

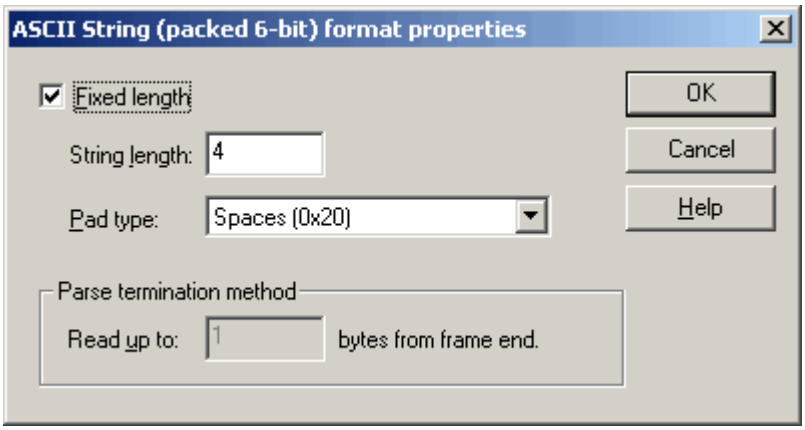
For **variable length ASCII real (packed 6 bit) data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This is accomplished by specifying an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

ROUND OFF: When writing values that require more characters than allotted by **String length**, the driver will write the largest positive or smallest negative value that can be expressed in the allotted space. For example, if string length is set to 6 and the precision is set to 2, then writing 1234.567 results in the string 999.99 and writing -123.456 results in the string -99.99.

Note: Due to packing, ASCII (packed 6 bit) data uses a reduced **ASCII (packed 6 bit) Character Table**. Attempting to use characters not in the **ASCII (packed 6 bit) Character Table** will result in data conversion failures.

Format ASCII String (Packed 6 Bit)

The ASCII String (Packed 6 Bit) **device data format** option allows the user to specify how string data should be formatted. For example, when a format of **ASCII String (packed 6 bit) [cccc...]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

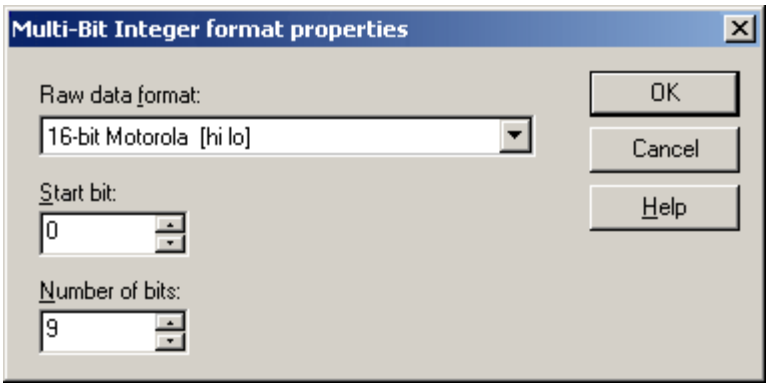
- For fixed length strings, the string length must also be set. The number entered sets the total number of characters (one byte per character) prior to conversion that will be written to or read from the device. The number of bytes sent over the wire is equal to three fourths the string length. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for values that do not require the full string length. Unlike ASCII integer (packed 6 bit) and ASCII real (packed 6 bit) formats, the pad characters are added as needed to the right. For example, if the string length was set to 4 and a value of ABC is to be written to the device, the driver will create a string consisting of the characters **ABC**, followed by one pad character. There are many options for pad characters: spaces (0x20), and zeros (0x30). The pad character option applies to writes only: the driver can read any valid ASCII (packed 6 bit) string of the specified length.

For **variable length ASCII (packed 6 bit) string data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This is accomplished by specifying an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

Note: Due to packing, ASCII (packed 6 bit) data uses a reduced **ASCII (packed 6 bit) Character Table**. Attempting to use characters not in the **ASCII (packed 6 bit) Character Table** will result in data conversion failures.

Format Multi-Bit Integer

The Multi-Bit Integer **device data format** option is used to associate the tag with a subset of bits in a longer integer value which is read from or written to the hardware. The tag's data type will determine how the integer equivalent of these bits will be communicated to or from the client application. For example, when a format of **Multi-Bit Integer** is selected, the **Format Properties** button in the **tag dialog** will become enabled. After clicking this button, the dialog should appear as shown below.



- The **Raw data format** control can be used to specify the length and byte order of the integer data as read from or written to the device. The quantity will be represented by one or more of the bits within this integer.
- The **Start bit** control sets the index of the first bit of interest with the integer. As is standard practice, the least significant bit (LSB) is referred to as bit index 0.
- The **Number of bits** control sets how many bits are within the integer, starting at the start bit index.

Read example
Say we have a device that measures an analog quantity which can range in value from 1 to 63. This value is reported by the device as the first 6 bits in a byte. The seventh bit in this byte indicates the status of the associated sensor, and the remaining bit is not used. We could create a **tag block** with a value tag using this Multi Bit Integer format, and a status tag using one of the single bit within byte formats. Both tags could be updated from a single block read transaction. For the value tag, we would set Raw data format to 8-bit Intel, Start bit to 0, and Number of bits to 6. If the device returned [01100111], the value tag would then be updated with a value of 39 (binary 100111).

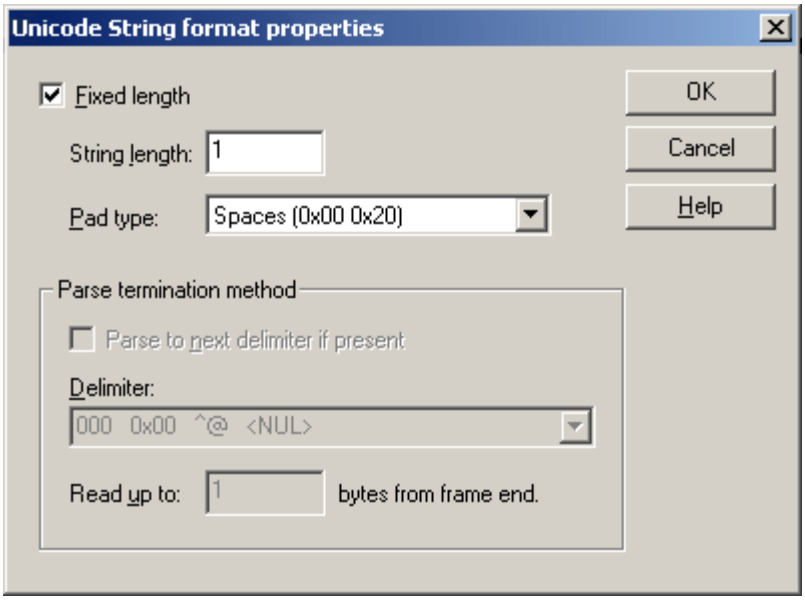
Write example
Assume we have a tag using this format with Raw data format set to 8-bit Intel, Start bit set to 3, and Number of bits set to 2. If a value of 1 is written to the tag, the device will receive the byte [00001000]. If a value of 3 or greater is written, the device will receive the byte [00011000].

Boolean Data types

The above examples assume the tag's data type is one of the integer types, Byte, Char, Word, etc. Boolean tags behave a bit differently. On reads, if any of the specified bits is set, the tag will receive a value of TRUE. All of the specified bits will be set if TRUE is written, and all bits will be cleared if FALSE is written.

Format Unicode String

The Unicode String **device data format** option allows the user to specify how string data should be formatted. For example, when **Unicode String [u1u2u3u4...]** is selected, the **Format Properties** button in the **tag dialog** will become enabled. Click **Format Properties** to display the **Unicode String Format Properties** dialog box, as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

- For fixed length strings, the **String length** must be set. The number entered here sets the total number of characters (two bytes per character) that will be written to or read from the device. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for

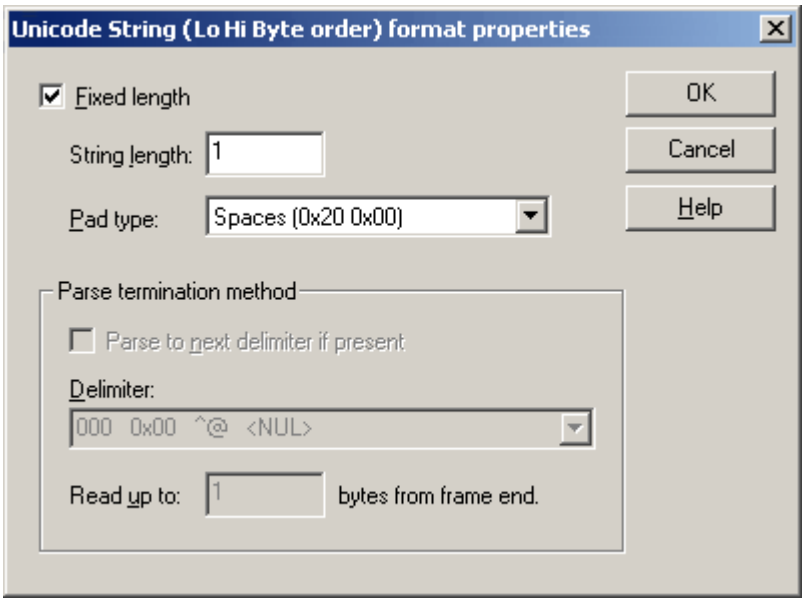
values that do not require the full string length. Unlike ASCII integer and ASCII real formats, the pad characters are added as needed to the right. For example, if the string length was set to 4 and a value of **ABC** is to be written to the device, the driver will create a string consisting of the characters, ABC in Unicode form, followed by one pad character. There are many options for pad characters: spaces (0x00 0x20), zeros (0x00 0x30), and NULL (0x00 0x00). The pad character option applies to writes only: the driver can read any valid ASCII string of the specified length.

For **variable length ASCII data**, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF may be chosen.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

Format UnicodeLoHi String

The Unicode String **device data format** option allows the user to specify how string data should be formatted. For example, when **Unicode String with Lo Hi Byte Order (u2u1u4u3...)** is selected, the **Format Properties** button in the **tag dialog** will become enabled. Click **Format Properties** to display the **UnicodeLoHi String Format Properties** dialog box, as shown below.



The **Fixed length** check box determines whether string data is a fixed or variable length. This box must be checked if a device will only accept strings of a given length in write transactions. If the length of a string returned from a read transaction cannot be anticipated, then this box should be unchecked.

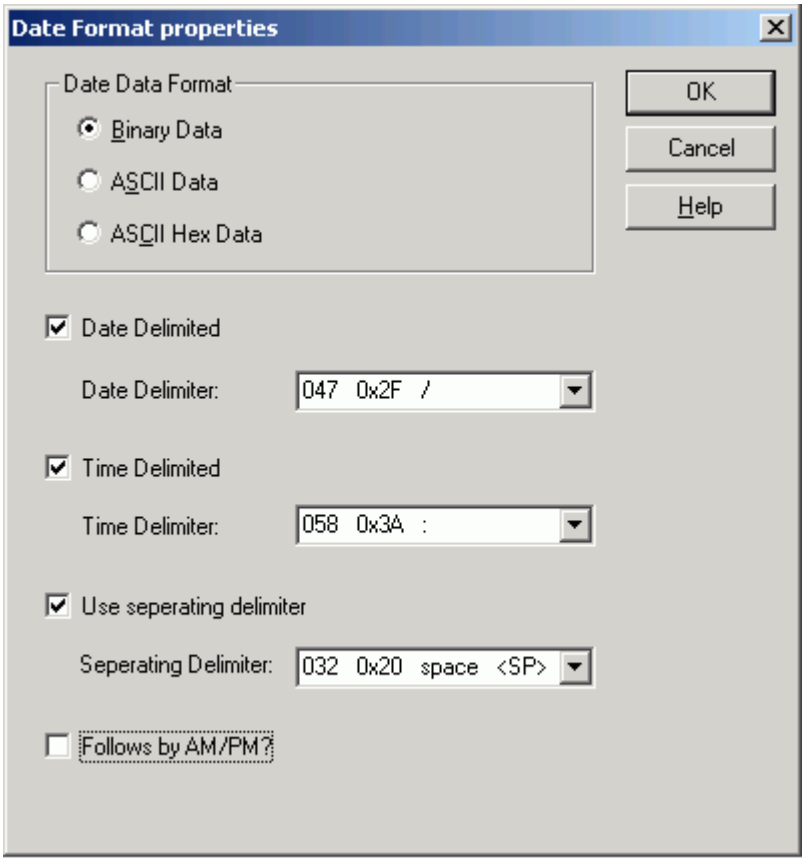
- For fixed length strings, the **String length** must be set. The number entered here sets the total number of characters (two bytes per character) that will be written to or read from the device. Null characters are not added to the end of strings written to the device, however: they are added to strings read from the device and passed to the client application.
- For fixed length strings, the **Pad type** must also be specified. Pad characters are used to fill out the string for values that do not require the full string length. Unlike ASCII integer and ASCII real formats, the pad characters are added as needed to the right. For example, if the string length was set to 4 and a value of **ABC** is to be written to the device, the driver will create a string consisting of the characters, ABC in Unicode form, followed by one pad character. There are many options for pad characters: spaces (0x20 0x00), zeros (0x30 0x00), and NULL (0x00 0x00). The pad character option applies to writes only: the driver can read any valid ASCII string of the specified length.

For variable length ASCII data, the driver must have some way of knowing where a tag's data ends when executing an **Update Tag** command. This can be accomplished in one of two ways:

- Specify a delimiter character. Check the **Parse to next delimiter if present** box if the end of the tag's data will be marked by a known character, as would be the case in a delimited list of values. For more information, refer to **Tips and Tricks: Delimited Lists**. When this box is checked, the **Delimiter** drop down list will be enabled. An ASCII character from 0x00 to 0xFF may be chosen. The driver will search for this character as ASCII hexadecimal data. For example, the two bytes 0x32 0x30 would be considered a space character.
- Give an end point relative to the frame end. The **Read up to xxx bytes from frame end** box can be used to define the end of a tag's data field relative to the end of a frame.

Format Date / Time

The Date **device data format** option allows the user to specify how date or date/time data will be formatted. When Date is selected, the **Format Properties** button in the **tag dialog** will become enabled. Click **Format Properties** to display the **Date Format Properties** dialog box as shown below.



First, select the **Date Data Format**.

- If **Binary Data** is selected, the date value will be sent as a binary value. For example, the value

  2006/17/10 10:09:12 AM

  would be sent as:

  0A 2F 11 2F 07 D6 20 0A 3A 09 3A 0C 20 41 4D
- If **ASCII Data** is selected, the date value will be sent as an ASCII string. Continuing with the example shown for Binary Data above, the value would be sent as:

  33 30 34 31 33 32 34 36 33 31 33 31 33 32 34 36 33 30 33 37 34 34 33 36 33 32 33 30 33 30 34 31 33 33 34

31 33 30 33 39 33 33 34 31 33 30 34 33 33 32 33 30 33 34 33 31 33 34 34 34

- If **ASCII Hex Data** is selected, the date value will be sent as an ASCII string converted to hexadecimal. Continuing with the example shown for Binary Data above, the value would be sent as:

  30 41 32 46 31 31 32 46 30 37 44 36 32 30 30 41 33 41 30 39 33 41 30 43 32 30 34 31 34 44

The remaining options in the dialog box can be used to further refine the date format.

- Check **Date Delimited** to include delimiters in the date value, and use the **Date Delimiter** drop-down list to choose the delimiter character (default: forward slash).
- Check **Time Delimited** to include delimiters in the time value, and use the **Time Delimiter** drop-down list to choose the delimiter character (default: colon).
- Check **Use separating delimiter** to include a separating delimiter in the date or date/time value, and use the **Separating Delimiter** drop-down list to choose the delimiter character (default: blank space).

Check the **Followed by AM/PM?** box in order to have the value be followed by "AM" or "PM".

Check Sum Descriptions

The U-CON (User-Configurable) Driver offers a variety of check sum options. Here is a brief description of each. Custom check sums can be created for a small fee. For more information, refer to Technical Support.

2's Complemented sum (8 bit)

The checksum is the 2's complement of the data received. The checksum is 8 bit.

2's Complemented sum (16 bit)

Same as 2's Complemented sum (8 bit) except the checksum is 16 bit.

CRC-CCITT (16 bit)

Cyclical Redundancy Check using: X^16 + X^12 + X^5 + 1 generator polynomial. Commonly used in XMODEM protocol.

CRC-CCITT-INITO (16 bit) (Reflected in/out)

Same as CRC-CCITT (16 bit) except input and output bytes are reflected and CRC is initialized to 0.

CRC-CCITT-INIT-0xFFFF

Same as CRC-CCITT (16 bit) except that the initialization = 0xFFFF.

CRC-CCITT-INIT-0xFFFF (16 bit)(Reflected)

Same as CRC-CCITT (16 bit) except input and output bytes are reflected and CRC is initialized to 0xFFFF.

CRC-16 (16 bit)

For more information, refer to **Custom #3 (16 bit)**.

CRC-16 (16 bit)(Reflected)

Cyclical Redundancy Check using: X^16 + X^15 + X^2 + 1 generator polynomial. Commonly used in MODBUS protocol.

CRC-16-INIT1 (16 bit) (Reflected)

Same as CRC-16 (16 bit) except that the initialization = 1.

CRC-32 (32 bit)

Cyclic Redundancy Check using x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + X^4 + x^2 + x^1 + 1 generator polynomial (Modbus "CRC-32" version)

CRC-32 (32 bit) (Reflected)

Same as CRC-32 (32 bit) except it uses reflected* polynomial.

LRC (8 bit)

Longitudinal Redundancy Check - two's complement of modulus 0xFF sum of all bytes.

LRC ASCII (8 bit)
Like LRC, but for ASCII Hex data. Pairs of bytes, assumed to be ASCII Hex values, are converted to their binary equivalent before being added to the modulus 0xFF sum.

MLEN (8 bit)
Adds the number of bytes in the message. The checksum is 8 bit.

MLEN (16 bit)
Same as MLEN (8 bit) except the checksum is stored in 16 bit. For example, if a message is received that has 4 bytes, MLEN (16 bit) would be 4 and it would be stored in a 16 bit field as 0x00 0x04 or 0x04 0x00 (depending on the format, Hi-Lo or Lo-Hi).

MLEN_INCL (8 bit)
Adds the number of bytes in the message including itself which is 1-byte long (8 bit). For example, if a message is received that has 4 bytes, then MLEN_INCL would be 4 + 1 = 5.

MLEN_INCL (16 bit)
Same as MLEN_INCL (8 bit) except the checksum is stored in 16 bit. For example, if a message of 4 bytes is received, MLEN_INCL (16 bit) would be 4 + 2 = 6. MLEN_INCL (16 bit) would be stored as 0x00 0x06 or 0x06 0x00 (depending on the format, Hi-Lo or Lo-Hi).

SUM (7 bit)
Adds the least 7 bits from each byte. The checksum is 8 bit.

SUM (8 bit)
Modulus 0xFF sum of all bytes.

SUM (16 bit)
Modulus 0xFFFF sum of all bytes.

Sum of [Hi Lo] Word Data (16 bit)
Modulus 0xFFFF sum of all words. Words are read in 16 bit Motorola [hi lo] format.

XOR (8 bit)
Bit wise exclusive OR of all bytes.

*__CRC Reflected:__ When reflected polynomials are used, the CRC is computed by processing data from the least significant bit to the most significant bit. Reflected or reciprocal polynomials are reversed. For example, if the regular polynomial is:

x^16 + x^15 + x^2 + 1 (0x8005) which in binary is 1000 0000 0000 0101

then the reflected polynomial will be:

1010 0000 0000 0001 x^16 + x^15 + x^13 + 1

Custom #1 (8 bit)
The C code used to calculate this custom check sum is as follows:

```
Byte CheckSumCustom_1 (Byte *pData, int nLength)
{
Byte byCS = 0xFF;

for (int nByte = 0; nByte < nLength; nByte ++)
 {
 Byte byTemp = pData [nByte];

 byCS = byCS ^ byTemp;
 byTemp = byCS;
 byTemp = (byTemp >> 3) & 0x1F;
 byCS = byCS ^ byTemp;
```

```
 byTemp = (byTemp >> 3) & 0x1F;
 byCS = byCS ^ byTemp;
 byTemp = byCS;
 byTemp = byTemp << 5;
 byCS = byCS ^ byTemp;
 }

return (byCS);
}
```

Custom #2 (8 bit)

This is a variation of the LRC (8 bit) check sum type - binary complement of the modulus 0xFF sum of all bytes. This can be expressed as:

byCS = ~bySum, or

byCS = 0xFF - bySum,

where byCS is the result and bySum is the modulus 0xFF sum of all bytes.

Custom #3 (16 bit)

This is a variation of the CRC-16 (16 bit) check sum type. Here, the sum is initialized to 0x0000, instead of 0xFFFF as it is in CRC-16 (16 bit)(Reflected).

Custom #4 (16 bit)

This is a variation of the CRC-16 (16 bit) check sum type. Here, the sum is initialized to 0x0000, instead of 0xFFFF as it is in CRC-16. Also, this check sum method searches the frame for a start sequence and end sequence. DLE characters are used for data transparency.

When using this check sum method, make sure that the whole frame is included in the calculation range. This driver will search for the start and end sequence within the frame. If the end points are not located, a check sum of 0x00 0x00 will be used.

The check sum calculation begins after <DLE><SOH> or <DLE><STX>. The characters of the start sequence are not included in the calculation. The calculation ends after <DLE><ETB>, <DLE><ETX>, or <DLE><ENQ>. The DLE characters in the end sequence are not included in the calculation.

Character Sequence	Included in CRC	Not Included in CRC
<DLE><SYN>	-	<DLE><SYN>
<DLE><SOH>	-	<DLE><SOH>
<DLE><STX>*	-	<DLE><STX>
<DLE><STX>**	<STX>	<DLE>
<DLE><ETB>	<ETB>	<DLE>
<DLE><ETX>	<ETX>	<DLE>
<DLE><DLE>	<DLE>	<DLE> (one)

*If not preceded in same block by transparent header data.
**If preceded in same block by transparent header data.

Custom #5 (8 bit)

This is a variation of the LRC (8 bit) check sum type. Here, control characters (0x00 - 0x1F) are not included in the summation.

Custom #6 (8 bit)

This is a variation of the SUM (8 bit) check sum type. Here, the raw data is assumed to be in lower-case ASCII Hex format (0 - 9, a - f). Each pair of ASCII Hex characters is converted to a byte value and summed modulus 0xFF. Users will typically want to select the "Byte from 2 ASCII Hex chars (lower-case) [hh]" **device data format** so that the resulting byte value is placed in lower-case ASCII Hex format.

Custom #7 (16 bit)

The C code used to calculate this custom check sum is as follows:

```
Word CheckSumCustom_7 (Byte *pData, int nLength)
{
C. CRC and checksum calculation
Use, including checksum:
void CheckSumCustom_7(unsigned char MessageOut[28])
{
unsigned char    i;
Word wCRCNChkSum = 0;
MessageOut[26] = 0xFF;
for (i = 0; i < 26; i ++)
 MessageOut[26] = CRC_Byte(MessageOut[26],
                           MessageOut[i]);
 MessageOut[27] = 0;

 for (i = 0; i < 27; i ++)
     MessageOut[27] += MessageOut[i]);

wCRCNChkSum = MessageOut [26];

wCRCNChkSum <<= 8;

wCRCNChkSum |= MessageOut [27];

return (wCRCNChkSum);

}
CRC algorithm:
unsigned char CRC_Byte(unsigned char Seed, unsigned char Data)
{
unsigned char    j;
for (j = 0; j < 8; j++)
{
 if (((Data ^ Seed) & 1)!= 0)
 {
    Seed ^= 0x18;
    Seed >>= 1;
    Seed |= 0x80;
 }
 else Seed >>= 1;
 Data >>= 1;
}
return (Seed);
}
```

Caution: If using a variable length data format, this custom check sum command requires an extra byte position for the CRC byte in the checksum field. Therefore, while setting up this check sum in the Transaction Editor, users must specify double the data length in the checksum data length field.

Custom #8 (16 bit)

This is a variation of the SUM (8 bit) check sum type where the output is 2 bytes: [0x30 + high nibble of sum] [0x30 + low nibble of sum]. For example, the 8 bit sum of the frame [1B 43 30 31] is 0xBF. Thus, the Custom #8 check sum of this frame would be [3B 3F].

Custom #9 (8 bit)

Takes the modulus 255 sum of data bytes, and bitwise OR's the result with 0x80 (i.e. set most significant bit to 1).

Custom #10 (16 bit)

16 bit version of LRC. Takes the modulus 0xFFFF sum of the data bytes, and returns the 2's complement of result.

Custom #11 (8 bit)

This is a variation of the XOR (8 bit) check sum. With this Custom #11, the intermediate result of each XOR operation is rotated left by 1 bit.

ASCII Character Table

Dec	Hex	ASCII	Key	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
0	0x00	NUL	Ctrl-@	32	0x20	Space	64	0x40	@	96	0x60	`
1	0x01	SOH	Ctrl-A	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	Ctrl-B	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	Ctrl-C	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	Ctrl-D	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	Ctrl-E	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	Ctrl-F	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	Ctrl-G	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	BS	Ctrl-H	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT	Ctrl-I	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	LF	Ctrl-J	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	VT	Ctrl-K	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	FF	Ctrl-L	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	CR	Ctrl-M	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	SO	Ctrl-N	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	SI	Ctrl-O	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	DLE	Ctrl-P	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	Ctrl-Q	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	Ctrl-R	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	Ctrl-S	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	Ctrl-T	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	Ctrl-U	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	Ctrl-V	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	Ctrl-W	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	Ctrl-X	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	Ctrl-Y	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	Ctrl-Z	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	Ctrl-[	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	Ctrl-\	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	GS	Ctrl-]	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	Ctrl-^	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	Ctrl-_	63	0x3F	?	95	0x5F	_	127	0x7F	Del

ASCII Character Table (Packed 6 Bit)

Dec	Hex	ASCII	Dec	Hex	ASCII
0	00	@	32	20	Space
1	01	A	33	21	!
2	02	B	34	22	`
3	03	C	35	23	#
4	04	D	36	24	$
5	05	E	37	25	%
6	06	F	38	26	&
7	07	G	39	27	`

8	08	H	40	28	(
9	09	I	41	29	)
10	0A	J	42	2A	*
11	0B	J	43	2B	+
12	0C	L	44	2C	,
13	0D	M	45	2D	-
14	0E	N	46	2E	.
15	0F	O	47	2F	/
16	10	P	48	30	0
17	11	Q	49	31	1
18	12	R	50	32	2
19	13	S	51	33	3
20	14	T	52	34	4
21	15	U	53	35	5
22	16	V	54	36	6
23	17	W	55	37	7
24	18	X	56	38	8
25	19	Y	57	39	9
26	1A	Z	58	3A	:
27	1B	[	59	3B	;
28	1C	\	60	3C	<
29	1D	\	61	3D	=
30	1E	^	62	3E	>
31	1F	_	63	3F	?

Tips and Tricks

Debugging: Using the Diagnostic Window and Quick Client

The server's Diagnostic Window and the Quick Client application are indispensable tools for debugging transactions. The Diagnostic Window shows users exactly what was sent and received by the driver during a transaction. Common errors (such as a **Read Response** command configured to receive an incorrect number of bytes or an incorrect device data format selection) are apparent with the Diagnostic Window. The Quick Client is tightly integrated with the server, so that users invoke a powerful test client with all of the tags automatically loaded with one click. With the Quick Client, users can manually control the execution of each transaction.

Follow the instructions below for the recommended method of debugging a new transaction. Note that the server project should be saved after each edit session.

1. Double-click on the desired channel in the server and make sure that the **Enable diagnostics** box is checked.
2. Next, click on the **Quick Client** icon on the server's toolbar.
3. Disable all tags in the Quick Client except for the ones in the "_System" and "_Statistics" groups. By doing this, the Diagnostic Window will not fill up with data from transactions that users are not interested in.

   Note: If users have a lot of tags, it may be easier to launch the Quick Client directly from Windows instead of

from the server. This way, users can manually add the tags they want to test and also specify when they are tested.

4. Return to the server and right-click on the channel. Select the **Diagnostics** item to bring up the Diagnostics Window. Then, return to the Quick Client and right-click on the tag to which the transaction belongs.
5. Issue a read or write request, depending on what type of transaction is being tested. The Diagnostic Window will show users the bytes the driver sent to the device and any response.

   Note: For more information, refer to the "Diagnostic Window" help topic in the OPC Server's help documentation.

   Important: If a change must be made to the transaction, users must disconnect the Quick Client from the server before invoking the Transaction Editor.
6. Next, minimize the Quick Client and perform the edits. Close the dialog only in order to disable the tags again.
7. After all changes have been made, users can bring the previous instance of the Quick Client back up and reconnect. The tags should not all need to be disabled again.
8. Check the transaction as before by issuing an asynchronous read or write.

Dealing with Echoes

Some devices operate in **echo mode**, which is when every byte sent to it is echoed back. Unless told otherwise, the U-CON (User-Configurable) Driver will ignore such echoes. It is usually perfectly okay to ignore these echoes. However, some devices will not accept the next byte sent to it until it has sent back the previous character. To make sure that the driver and device remain in sync in these cases, users must process each echoed byte. For example, if the command string "AB1" needs to be sent to such a device, it should then send a nine-character response. A transaction would need to be created like as is shown below.

STEP	COMMAND	COMMAND PARAMS	DESCRIPTION
1	Write Character	A	Place "A" in TX buffer
2	Transmit		Send the "A"
3	Read Response	Wait for 1 character	Wait for echoed "A"
4	Write Character	B	Place "B" in TX buffer
5	Transmit		Send the "B"
6	Read Response	Wait for 1 character	Wait for echoed "B"
7	Write Character	1	Place "1" in TX buffer
8	Transmit		Send the "1"
9	Read Response	Wait for 10 characters	Wait for echoed "1" and nine character command response
10	Update Tag	This tag	Parse response, accounting for echoed "1" at the beginning of the RX buffer, and update tag

Note: The reason some devices echo is to provide a means of error checking. To actually perform such error checking, a **Test Character** command will need to be included after each **Read Response** command to make sure that the returned character is what it is supposed to be. If it is not, users could "Go To" an error handling section of the transaction. Keep in mind that additional transaction commands will decrease the performance of the driver.

Branching: Using the conditional, Go To, Label and End commands

The U-CON (User-Configurable) Driver is used to create transactions that branch off and execute different sets of commands depending on the data received from a device. Error handling is the most common use for branching. If data is judged to be good, one set of commands will be executed; if it is judged to be bad, another set of commands will be executed. In the example below, the device is sent a read request of some sort. The device will return an error code of 0x00 or 0x01. If the error code is 0x00, the device successfully processed the read request but requires the driver to send back the acknowledgement code 0x06. If the error code is 0x01, the device failed to process the request. In the example transaction, the error code is examined, the tag is updated and the acknowledgment is sent if the read request succeeds. If it fails, the request is repeated. If it fails a second time, the tag is invalidated and requests are stopped.

STEP	COMMAND	COMMAND PARAMS	DESCRIPTION
1	Write String	AB1	Place read request string in TX buffer
2	Transmit		Send the request
3	Read Response	Wait for terminator 0x0D	Get response from device
4	Test Character	Position = 4 Test Character = 0x00 TRUE action = Go To "Good" FALSE action = Continue	Ack receipt of good data or retry
5	Log Event	"Retrying AB1 command"	Post message to server's event log
6	Transmit		Send the last command again
7	Read Response	Wait for terminator 0x0D	Get response from device
8	Test Character	Position = 4 Test Character = 0x00 TRUE action = Go To "Good" FALSE action = Invalidate Tag	Ack receipt of good data or invalidate tag and give up.
9	End		Do not proceed into next section
10	Label	Label = Good	Marks beginning of good data processing section
11	Update Tag	Tag = This tag	Update tag with good data
12	Write Character	0x06	Place acknowledgement code in TX buffer
13	Transmit		Send acknowledgement

Note: Steps 10-13 are executed only when the device returns an error code of 0x00 (success).

Slowing Things Down: Using the Pause command

Users may encounter devices that are not capable of operating at the same speed as the server. In these cases, **Pause** commands can be added to the transactions to slow things up. In the example below, the device requires a short pause between each character in the read request "AB1":

STEP	COMMAND	COMMAND PARAMS	DESCRIPTION
1	Write Character	"A"	
2	Transmit		Send first character
3	Pause	Time = 50 ms	Wait before sending next character
4	Write Character	"B"	
5	Transmit		Send second character
6	Pause	Time = 50 ms	Wait before sending next character
7	Write Character	"1"	
8	Transmit		Send third and last character of request
9	Read Response	Wait for 10 characters	Wait for 10 character response to AB1 command
10	UpdateTag	Tag = This tag	Parse response and update tag

Note: Omitting the **Transmit** commands in steps 2 and 5 would not produce the desired effect. In that case, the U-CON (User-Configurable) Driver would slowly build up the TX buffer internally, and then send all three characters in the usual rapid succession.

Bit Fields: Using the Modify Byte and Copy Buffer commands

For efficiency, sometimes protocols pack several device settings into a single byte, sometimes called a bit field. For example, consider a process control device that has four outputs R0, R1, R2, and R3. Each of these outputs can operate in either alarm mode or proportional control mode. It is typical for such devices to allow the mode of all four outputs to be read using a single command that returns all four settings in a single byte bit field. For example, bit 0 may represent output R0, bit 1 represents R1, etc. If a bit is 0, then the output is in alarm mode, and if the bit is 1 the output is in proportional mode. Likewise, the mode of all four outputs is usually set with a single command that takes a bit field as an argument.

To read the mode of each output, users should create a tag block with four tags: Mode_R0, Mode_R1, Mode_R2, and Mode_R3. These tags should have Read/Write access and have a data type of Boolean. The device data formats should be "Bit 0 from byte (00000001)" for Mode_R0, "Bit 1 from byte (00000010)" for Mode_R1, etc. The block read transaction must issue the appropriate read command and then update all four tags. All four of the update tag commands must have the same data "start position" which points to the byte containing the output mode settings.

Setting the mode of a single output requires a bit more work. Since our hypothetical set output mode function takes a bit field that sets the mode of all four outputs, users need to know what mode the other three outputs are in. This way, users can construct the bit field used in the set output mode command such that all other outputs are unchanged. For example, to be able to set the mode of output R0, users define the write transaction attached to the Mode_R0 tag. The first thing that must occur in this transaction is to issue the get output mode command string and receive the response. The current output mode settings are encoded somewhere in the RX buffer and are available to users for the remainder of the transaction. After this read response, users need to construct the set output mode command string in the TX buffer. Somewhere in that command string users will need to place the output mode bit field. Users get this by issuing a **Copy Buffer** command that will copy the current settings from the RX buffer to the TX buffer. Next, users need to modify bit 0 of this byte to set the mode of output R0. The "Modify Byte" function does just that. It will take the value to be written to the device and modify a bit or set of bits in the specified byte accordingly. In this case, users can use it to modify bit 0 of the byte by specifying the bit mask "00000001". Writing 0 to the Mode_R0 tag then results in bit 0 being set to 0, setting R0 to alarm mode. Writing 1 results in bit 0 being set to 1, setting R0 to proportional control mode. All other bits remain unchanged, and therefore outputs R1, R2, and R3 remain in the same mode.

Transferring Data Between Transactions: Using Scratch Buffers

Some protocols require that a special type of **Device Identifier** be used in all requests. This Identifier can be read directly from the device using a special command. A read transaction could be defined to issue this **Get Device Identifier** command, and store the returned value in a scratch buffer. All other transactions defined for that device could copy this value from the scratch buffer to the write buffer. The client application would have to make sure that the **Get Device Identifier** tag be read before any other read or write transaction takes place.

Scratch buffers can also be used in **Write Only** tags. The U-CON (User-Configurable) Driver does not support Write Only tags as such, but a tag can be created with both read and write transactions, where only the write transaction makes a request of the physical device. If the read transaction is empty, the client will report bad data quality for that tag. A better situation would be for the read transaction to return the last value written to the device. To do this, select both the **Write buffer** and a **Scratch buffer** in the write transaction's **Write Data** command. In the read transaction, simply use an **Update Tag** command with the data source being the scratch buffer. Keep in mind that this is not a value just read from the device, it is the last value written to the device. If an Update Tag command is executed before any data has been saved in the scratch buffer, the tag value will be set to zero.

Important: Unlike a scratch buffer which is associated with one device only, a global buffer is associated with multiple devices and should be used with caution.

Scanner Applications

Transaction **event counters** can be especially useful in scanner applications. Typically, scanners will issue a notification each time an item is scanned – they are not usually designed to be polled. The UCON can be configured to receive and process this sort of data with an **unsolicited transaction**. The primary function of this transaction would be to parse the data of interest from a message and update a tag with it.

This simple design works fine, unless it is possible for the same item or code to be scanned multiple times. The client will get no indication that multiple scans have occurred. All it knows is that the tag's value has not changed since the last timestamp. To get around this issue, event counters were introduced into the UCON. Each time an item is scanned, the unsolicited transaction that was defined for that scanner will be triggered and its event counter incremented. Users should update two tags in the transaction: one with the data parsed from the unsolicited message and the other with the transaction's event counter value. These tags must belong to a tag block. The client application will see the event counter tag change each time an item is scanned.

Note: Event counter values are stored in 16 bit buffers. All tags updated from event counters must be configured with the 16 bit Intel (Lo Hi) device data format.

Delimited Lists

Many protocols provide data for multiple values in a list format, generally providing a separate tag for each value. In these cases, it makes sense to create a **Tag Block**. A tag block will have a single, common read transaction that can be

used to read data for all its member tags in a single shot. This read transaction will contain a number of **Update Tag** commands, one for each of its member tags. If the number of bytes of each data field are fixed, then parsing the frame is easy. Users must specify the data start byte in each **Update Tag** command and the data length in the tag definition. It is more complicated if the length of the data fields is variable: in these cases, the protocol must provide some sort of delimiter character to mark the end of one field and the beginning of the next. The U-CON (User-Configurable) Driver driver provides **Buffer Pointers** and associated command options to aid in parsing delimited lists. **See Also:** **Tags** and **Device Data Formats**.

Example

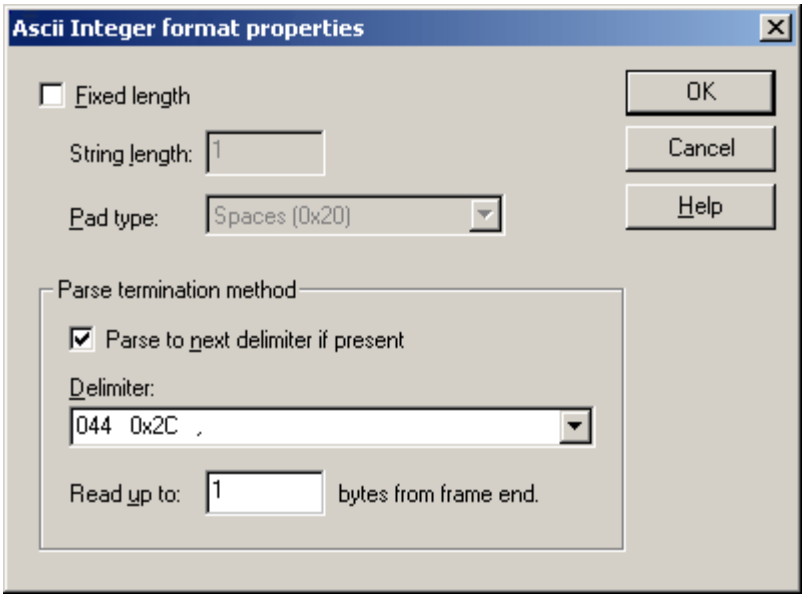
For example, users expect the response to a read request to be of the form:

[STX] [value 1 bytes], [value 2 bytes], [value 3 bytes] [ETX]

where the values are ASCII integers of unknown length and the values could range from -100 to 1000.

1. Start by creating a tag block with three tags in it: Tag_1, Tag_2, and Tag3 for values 1, 2, and 3 respectively. Choose a data type of short for each tag since its range is sufficient to cover the expected range of values. Next, select the **ASCII Integer** device data format for each tag. For more information, refer to **Tags** and **Tag Blocks**.

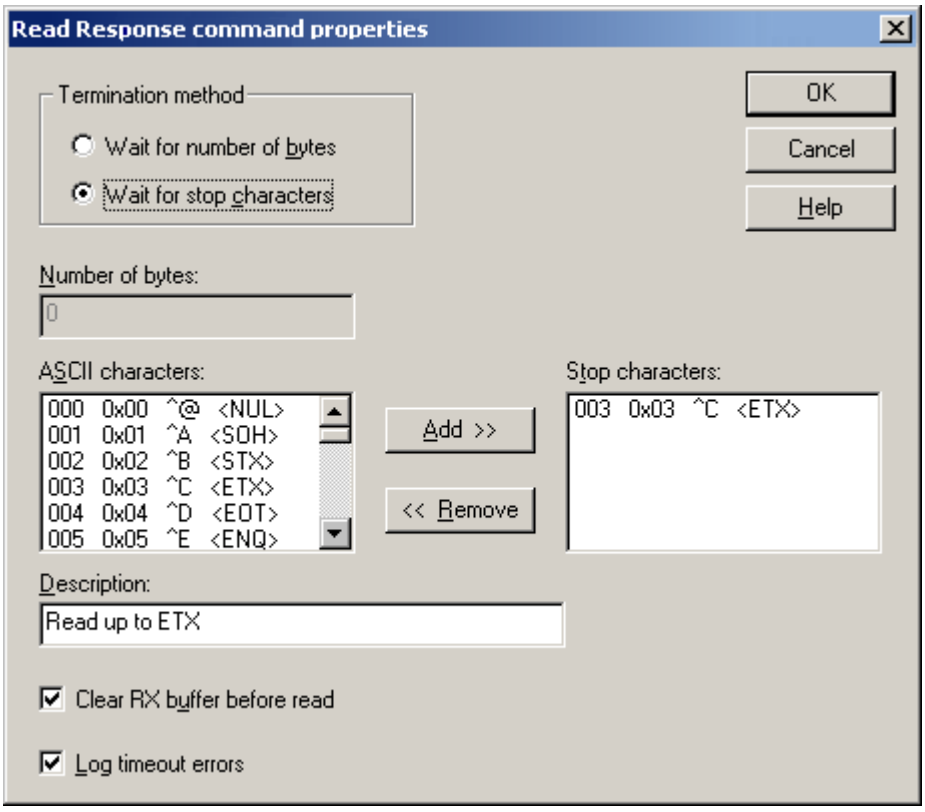
Some of the specialized options of the ASCII Integer device data format must be used in this case. For Tag_1 and Tag_2, choose the **Parse to next delimiter** format option and then choose the comma (0x2C) as the delimiter. The **Format Properties** should appear as shown below.



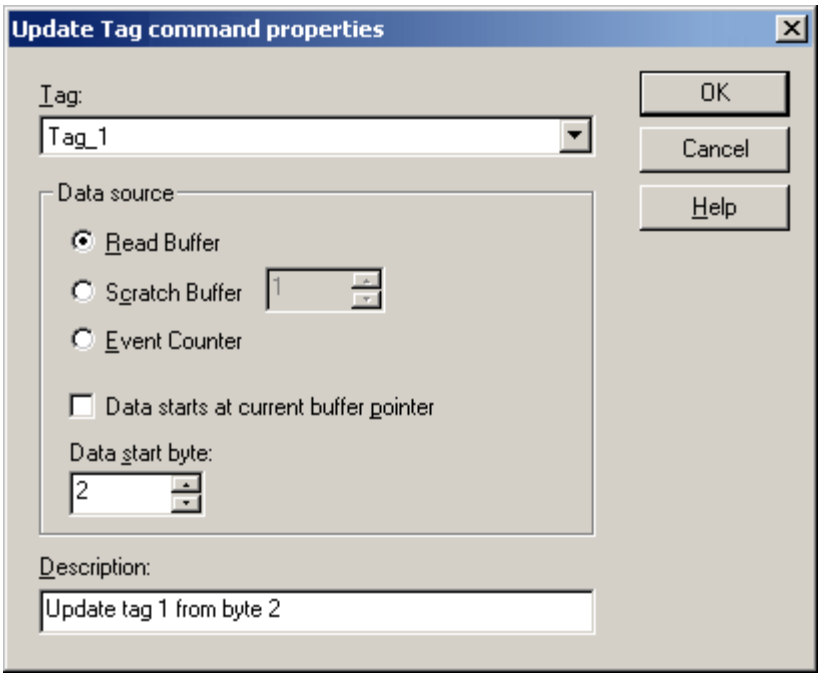
2. Since value 3 does not precede a comma, it must have a different termination method. Two equally good options exist here: users can choose to parse to the next delimiter, where this time the delimiter would be the end ETX character. Or, users could leave the "Parse to next delimiter" box unchecked and specify "Read up to..." 1 byte from frame end.

3. Next, define the block read transaction. The first set of commands in the transaction will build the read request in the write buffer. The details of the request are not important for this example. Following these commands will be a **Transmit** command to send the write buffer to the device.

4. Next, define a **Read Response** command to gather the response and store it in the read buffer. In this example, users do not know how many bytes to expect but they do know that the response will end with the ETX character. The command properties will look as shown below.

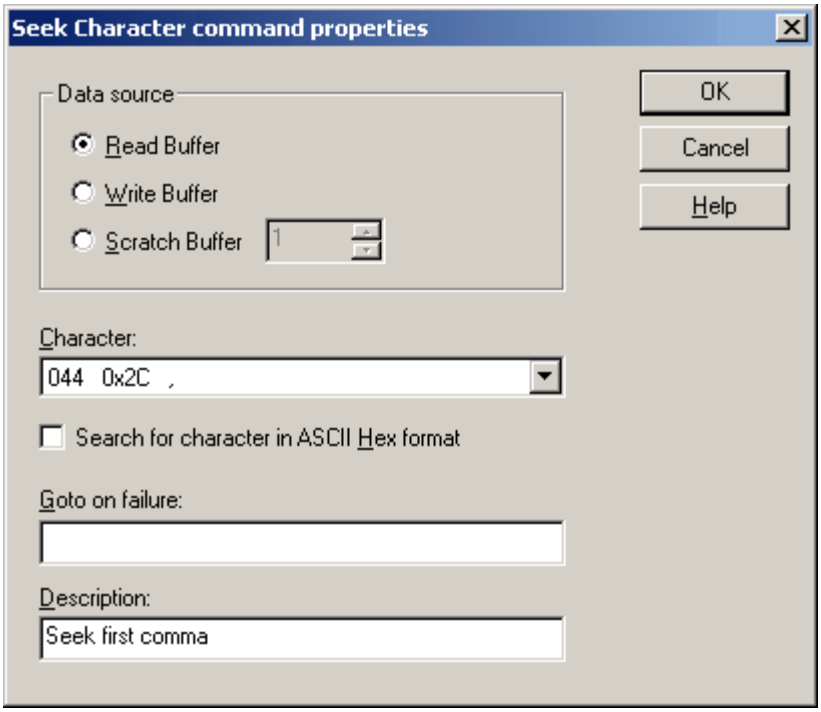


5. Once the response has been received and copied into the read buffer, commands must be added to parse the data and send the result to the appropriate tag. The **Update Tag** command does just that. There must be an Update Tag command for each tag in the block. For Tag_1, users know the data starts at byte 2 in the read buffer. The device data format defined for Tag_1 tells the driver to parse up to the next comma. The command properties for Tag_1 will look as shown below.



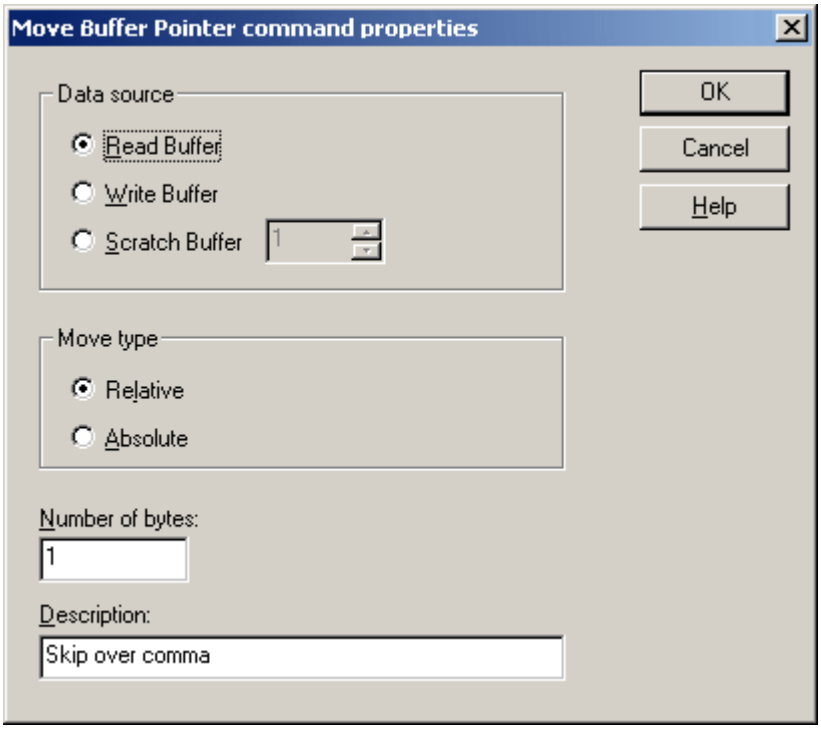
6. Users cannot predict what byte the data for Tag_2 will start on because of the variable length ASCII values, but they

do know value 2 will follow the first comma in the frame. This is where **buffer pointers** come into play. The objective is to move the read buffer pointer to the start of value 2. This is done in two steps, the first of which is accomplished with a **Seek Character** command. This command is used to move the pointer to the first comma in the frame.

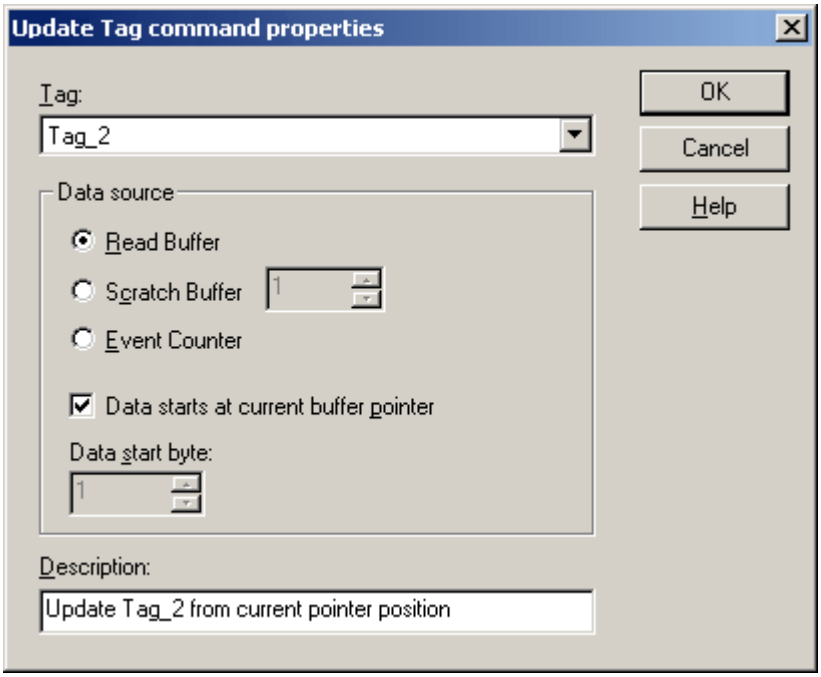


7. If there was some question of where the delimiter will be found, users can specify a "Go To on failure" **Label** to handle the situation.

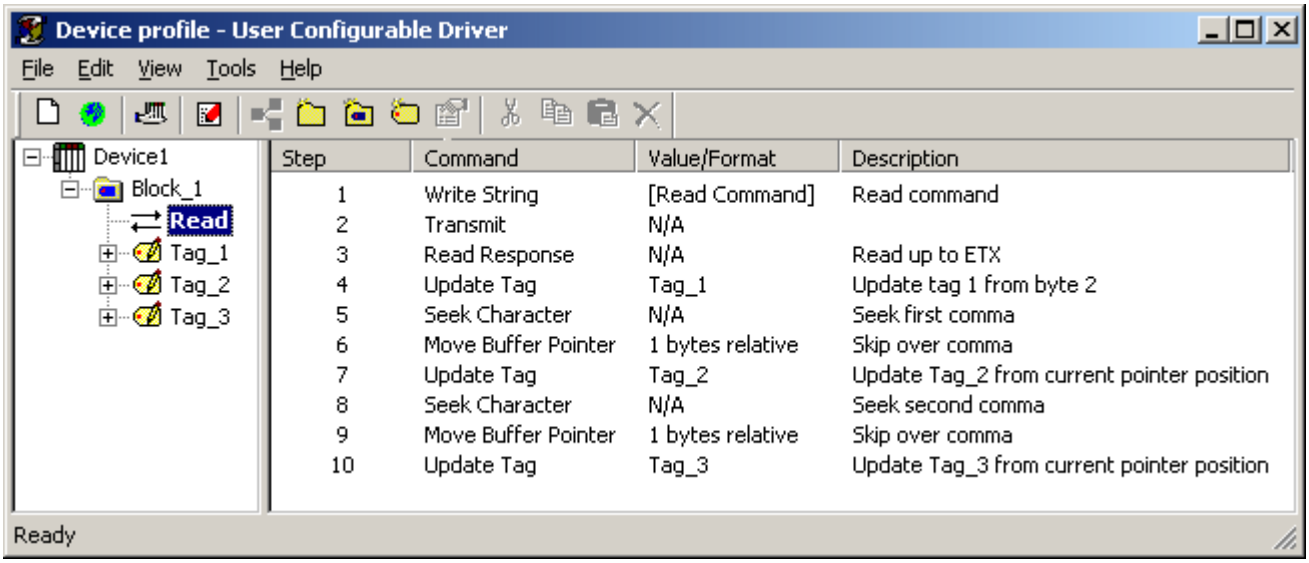
8. Next, move the pointer past the comma to the first byte of value 2. This is done using a **Move Buffer Pointer** command. In this case, users should perform a relative move one byte from the current position.



9. If users expected values to be separated by a comma space, then they would have entered 2 in **Number of bytes**. Now the read buffer pointer points to the first byte of value 2. The Update Tag command for Tag_2 should appear as shown below.



10. To parse value 3, issue another **Seek Character**, **Mover Buffer Pointer** and **Update Tag** sequence just like what was done for Tag_2. The full read transaction should appear as shown below.



Data Types Description

The U-CON (User-Configurable) Driver can be used to represent a tag's data as any one of the basic types described below. Choose a data type that is recognized by the client application and will accommodate the expected range of data values.

Data Type	Description

Boolean	Single bit
Byte	Unsigned 8 bit value bit 0 is the low bit bit 7 is the high bit
Char	Signed 8 bit value bit 0 is the low bit bit 6 is the high bit bit 7 is the sign bit
Word	Unsigned 16 bit value bit 0 is the low bit bit 15 is the high bit
Short	Signed 16 bit value bit 0 is the low bit bit 14 is the high bit bit 15 is the sign bit
DWord	Unsigned 32 bit value bit 0 is the low bit bit 31 is the high bit
Long	Signed 32 bit value bit 0 is the low bit bit 30 is the high bit bit 31 is the sign bit
BCD	Two byte packed BCD Value range is 0-9999. Behavior is undefined for values beyond this range.
LBCD	Four byte packed BCD Value range is 0-99999999. Behavior is undefined for values beyond this range.
Float	32 bit floating point value. The driver interprets two consecutive 16 bit registers as a floating point value by making the second register the high word and the first register the low word.
Double	64 bit floating point value
String	Zero terminated character array

Note: "Data Type" refers to the representation of data values that the server and client applications exchange. Data exchanged between the server and a device can be formatted in a wide variety of ways, depending on the data type. For more information, refer to **Device Data Formats**.

Address Descriptions

The U-CON (User-Configurable) Driver does not use the tag address in the usual manner. Normally, a driver "knows" how to interpret an address string specified by the user, and build read and write requests accordingly. This is not possible with the U-CON (User-Configurable) Driver since it was not developed for a specific device. It is up to the user to properly encode an address in each transaction defined in the driver profile. (In many devices a command code is sufficient, in others, a command code and memory location are required to access a given piece of data.) The U-CON (User-Configurable) Driver uses the "address" to describe the path relationship between the tag and device as defined in the Transaction Editor. "Group dot tag" notation is used.

Example
An address of "Group_1.Registers.Register_1" means the tag "Register_1" is in the group "Registers", which is in a group called "Group_1". "Group_1" is attached to the device. Thus, a user can manually add a tag to the server, so

long as it was previously defined with the Transaction Editor and the path is known. However, this is generally not necessary since the Transaction Editor automatically invokes the server's auto-tag database generation feature.

Error Descriptions

The following error/warning messages may be generated. Click on the link for a description of the message.

Address Validation

Missing address
Device address '<address>' contains a syntax error
Address '<address>' is out of range for the specified device or register
Device address '<address>' is not supported by model '<model name>'
Data Type '<type>' is not valid for device address '<address>'
Device address '<address>' is Read Only
Array size is out of range for address '<address>'
Array support is not available for the specified address: '<address>'

Serial Communications

COMn does not exist
Error opening COMn
COMn is in use by another application
Unable to set comm parameters on COMn
Communications error on COMn [<error mask>]
Unable to create serial I/O thread

Device Status Messages

Device '<device name>' is not responding
Unable to write to '<address>' on device '<device name>'

U-CON (User-Configurable) Driver Error Messages

RX buffer overflow. Stop characters not received
RX buffer overflow. Full variable length frame could not be received
Unable to locate Transaction Editor executable file
Copy Buffer command failed for address '<address.transaction>' - <source/destination> buffer bounds
Failed to load the global file
Go To command failed for address '<address.transaction>' - label not found
Mod Byte command failed for address '<address.transaction>' - write buffer bounds
Test Character command failed for address '<address.transaction>' - source buffer bounds
Test Check Sum command failed for address '<address.transaction>' - read buffer bounds
Test Check Sum command failed for address '<address.transaction>' - data conversion
Test Device ID command failed for address '<address.transaction>' - read buffer bounds
Test Device ID command failed for address '<address.transaction>' - data conversion
Update Tag command failed for address '<address.transaction>' - <read/scratch/event counter> buffer bounds
Write Character command failed for address '<address.transaction>' - write buffer bounds
Write Check Sum command failed for address '<address.transaction>' - write buffer bounds
Write Check Sum command failed for address '<address.transaction>' - data conversion
Write Data command failed for address '<address.transaction>' - write buffer bounds
Write Data command failed for address '<address.transaction>' - data conversion
Write Device ID command failed for address '<address.transaction>' - write buffer bounds
Write Device ID command failed for address '<address.transaction>' - data conversion
Write String command failed for address '<address.transaction>' - write buffer bounds
Tag update for address '<address>' failed due to data conversion error
Unsolicited message receive timeout
Unsolicited message dead time expired

XML Errors

Address Validation

The following error/warning messages may be generated. Click on the link for a description of the message.

Address Validation

Missing address

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically has no length.

Solution:
Re-enter the address in the client application.

Device address '<address>' contains a syntax error

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically contains one or more invalid characters.

Solution:
Re-enter the address in the client application.

Address '<address>' is out of range for the specified device or register

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically references a location that is beyond the range of supported locations for the device.

Solution:
Verify the address is correct; if it is not, re-enter it in the client application.

Device address '<address>' is not supported by model '<model name>'

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically references a location that is valid for the communications protocol but not supported by the target device.

Solution:
Verify that the address is correct; if it is not, re-enter it in the client application. Verify that the selected model name for the device is correct.

Data Type '<type>' is not valid for device address '<address>'

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically has been assigned an invalid data type.

Solution:
Modify the requested data type in the client application.

Device address '<address>' is Read Only

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically has a requested access mode that is not compatible with what the device supports for that address.

Solution:
Change the access mode in the client application.

Array size is out of range for address '<address>'

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically is requesting an array size that is too large for the address type or block size of the driver.

Solution:
Re-enter the address in the client application to specify a smaller value for the array or a different starting point.

Array support is not available for the specified address: '<address>'

Error Type:
Warning

Possible Cause:
A tag address that has been specified statically contains an array reference for an address type that doesn't support arrays.

Solution:
Re-enter the address in the client application to remove the array reference or correct the address type.

Serial Communications

The following error/warning messages may be generated. Click on the link for a description of the message.

Serial Communications
COMn does not exist
Error opening COMn
COMn is in use by another application
Unable to set comm parameters on COMn
Communications error on COMn [<error mask>]
Unable to create serial I/O thread

COMn does not exist

Error Type:
Fatal

Possible Cause:
The specified COM port is not present on the target computer.

Solution:
Verify that the proper COM port has been selected in the Channel Properties.

Error opening COMn

Error Type:
Fatal

Possible Cause:
The specified COM port could not be opened due to an internal hardware or software problem on the target computer.

Solution:
Verify that the COM port is functional and may be accessed by other Windows applications.

COMn is in use by another application

Error Type:
Fatal

Possible Cause:
The serial port assigned to a channel is being used by another application.

Solution:
1. Verify that the correct port has been assigned to the channel.
2. Close the other application that is using the requested COM port.

Unable to set comm parameters on COMn

Error Type:

Fatal

Possible Cause:
The serial parameters for the specified COM port are not valid.

Solution:
Verify the serial parameters and make any necessary changes.

Communications error on COMn [<error mask>]

Error Type:
Warning

Error Mask Definitions:
B = Hardware break detected.
F = Framing error.
E = I/O error.
O = Character buffer overrun.
R = RX buffer overrun.
P = Received byte parity error.
T = TX buffer full.

Possible Cause:
1. The serial connection between the device and the host PC is bad.
2. The communication parameters for the serial connection are incorrect.
3. There is a noise source disrupting communications somewhere in the cabling path between the PC and the device.

Solution:
1. Verify the cabling between the PC and the device.
2. Verify that the specified communication parameters match those of the device.
3. Reroute cabling to avoid sources of electrical interference such as motors, generators or high voltage lines.

Unable to create serial I/O thread

Error Type:
Warning

Possible Cause:
The OPC Server process has no more resources available to create new threads.

Solution:
Remember that each tag group takes up a thread, and that the typical limit for a single process is about 2000 threads. Reduce the number of tag groups in the project.

Device Status Messages

The following error/warning messages may be generated. Click on the link for a description of the message.

Device Status Messages
Device '<device name>' is not responding
Unable to write to '<address>' on device '<device name>'

Device <device name>' is not responding

Error Type:
Serious

Possible Cause:

1. The serial connection between the device and the host PC is broken.
2. The communication parameters for the serial connection are incorrect.
3. The named device may have been assigned an incorrect Network ID.
4. One or more transactions are not configured properly.

Solution:

1. Verify the cabling between the PC and the device.
2. Verify that the specified communication parameters match those of the device.
3. Verify that the Network ID given to the named device matches that of the actual device.
4. Check that all Read Response command properties are correct. A very common cause for "Device not responding" errors from this driver is a Read Response command set to wait for more bytes that the device actually sends. It may also be necessary to place a pause command at the end of transactions that write to the device but do not get a response. In such cases, the device may need a short period of time to process the write before it will accept the next request from the driver.

Unable to write to '<address>' on device '<device name>'

Error Type:

Serious

Possible Cause:

1. The serial connection between the device and the host PC is broken.
2. The communication parameters for the serial connection are incorrect.
3. The named device may have been assigned an incorrect Network ID.

Solution:

1. Verify the cabling between the PC and the device.
2. Verify that the specified communication parameters match those of the device.
3. Verify that the Network ID given to the named device matches that of the actual device.

U-CON (User-Configurable) Driver Error Messages

The following error/warning messages may be generated. Click on the link for a description of the message.

U-CON (User-Configurable) Driver Error Messages

RX buffer overflow. Stop characters not received

Error Type:
Serious

Possible Cause:
The read buffer filled to capacity while waiting for the stop characters specified in the transaction's Read Response command.

Solution:
Make sure that the correct stop characters are specified in the Read Response command. If the receive frame is of known length, use the "Wait for Number of Bytes" command option instead.

See Also:
Read Response Command

RX buffer overflow. Full variable length frame could not be received

Error Type:
Serious

Possible Cause:
The read buffer filled to capacity while receiving a frame containing a data length field described in the transaction's Read Response command.

Solution:
Make sure that the data length start position, format, and trailing bytes specified in the Read Response command are correct.

See Also:
Read Response Command

Unable to locate Transaction Editor executable file

Error Type:
Serious

Possible Cause:
The Transaction Editor executable file is not in the expected location.

Solution:
Make sure that the Transaction Editor executable (UserConfigDrv_GUI_u.exe) is located in the server's "utilities" subdirectory. Reinstall the driver if not.

Copy Buffer command failed for address '<address.transaction>' - <source/destination> buffer bounds

Error Type:

Serious

Possible Cause:
The combination of "start byte" and "number of bytes" properties of the **Copy Buffer** command have caused to driver to attempt to access non-existent source buffer elements.

Solution:
Make sure that the Copy Buffer command property settings are correct and that the source buffer contains valid data when the offending Copy Buffer command is executed.

Failed to load the global file

Error Type:
Serious

Possible Cause:
Driver was unable to create or open a temporary file used to transfer function block data between driver and Transaction Editor. The file may have become corrupted or was removed while driver was running.

Solution:
Restart the server and retry the last edits with the Transaction Editor.

Note:
Contact Technical Support if error occurs again.

Go To command failed for address '<address.transaction>' - label not found

Error Type:
Serious

Possible Cause:
The specified label does not exist in the present transaction.

Solution:
Make sure the transaction has a Label command of exactly the same name as that of the Go To command's label property. Labels are case sensitive.

See Also:
Label Command
Go To Command

Mod Byte command failed for address '<address.transaction>' - write buffer bounds

Error Type:
Serious

Possible Cause:
The byte position property of the Mod Byte command is not within the current bounds of the write buffer.

Solution:
This command can only operate on bytes placed on the write buffer prior to the execution of this command. Make sure that the byte position setting is within this range of bytes.

See Also:
Mod Byte Command

Test Character command failed for address '<address.transaction>' - source buffer bounds

Error Type:
Serious

Possible Cause:
The "Position" property of the Test Character command is not within the current bounds of the source buffer.

Solution:
This command can only operate on bytes received by the last Read Response command when the data source is specified as the read buffer. Make sure that the position value is not larger than the number of bytes received.

See Also:
Read Response Command
Test Character Command

Test Check Sum command failed for address '<address.transaction>' - read buffer bounds

Error Type:
Serious

Possible Cause:
The start byte or number of bytes properties of the Test Check Sum command are incorrect and have caused to driver to attempt to access non-existent read buffer elements.

Solution:
This command can only operate on bytes received by the last Read Response command. Make sure that the sum of start byte and number of bytes does not exceed the number of bytes received.

See Also:
Read Response Command
Test Check Sum

Test Check Sum command failed for address '<address.transaction>' - data conversion

Error Type:
Serious

Possible Cause:
A necessary data format conversion failed.

Solution:
If the problem is persistent, try to find another compatible data format. If dynamic ASCII formatting is used, make sure all necessary format characters are present in the table.

See Also:
Dynamic ASCII Formatting

Test Device ID command failed for address '<address.transaction>' - read buffer bounds

Error Type:
Serious

Possible Cause:

The "start byte" property of the Test Device ID command is incorrect and has caused to driver to attempt to access non-existent read buffer elements.

Solution:
This command can only operate on bytes received by the last Read Response command. Make sure that the start byte value does not exceed the number of bytes received.

See Also:
Test Device ID
Read Response Command

Test Device ID command failed for address '<address.transaction>' - data conversion

Error Type:
Serious

Possible Cause:
A necessary data format conversion failed.

Solution:
If the problem is persistent, try to find another compatible data format. If dynamic ASCII formatting is used, make sure all necessary format characters are present in the table.

See Also:
Dynamic ASCII Formatting

Test String command failed for address '<address.transaction>' - source buffer bounds

Error Type:
Serious

Possible Cause:
The data source buffer does not currently contain enough characters to perform the string comparison described in a Test String command.

Solution:
Verify that the transaction has been properly configured and that the driver is receiving the data as expected.

See Also:
Test String Command

Update Tag command failed for address '<address.transaction>' - <read/scratch/ event counter> buffer bounds

Error Type:
Serious

Possible Cause:
The combination of "data start byte" property of the Update Tag command and tag data size have caused to driver to attempt to access non-existent source buffer elements.

Solution:
This command can only operate on bytes received by the last Read Response command, previously stored in a Scratch buffer or global buffer, or the 16 bit values stored in event counter buffers. Make sure the sum of data start byte and the data length (2 for word, 4 for float, etc) does not exceed the number of bytes in the source buffer.

See Also:

Update Tag
Read Response Command
Scratch Buffer
Global Buffer
Event Counter

Write Character command failed for address '<address.transaction>' - destination buffer bounds

Error Type:
Serious

Possible Cause:
The command caused the driver to attempt to write past the maximum destination buffer limit of 8192 bytes.

Solution:
1. The destination buffer should be of ample size for all but the most unusual circumstance. Ensure that the byte count of the message being constructed is less than 8192 bytes. If it is, examine the command properties in the offending transaction. The most common cause of this sort of error is an incorrect Start Byte, End Byte or Number of Bytes value.
2. Make sure that the number of bytes written by a Write Data command are considered. This is set by the tag's device data format specification.

Write Check Sum command failed for address '<address.transaction>' - write buffer bounds

Error Type:
Serious

Possible Cause:
The command caused the driver to attempt to write past the maximum write buffer limit of 8192 bytes.

Solution:
1. The write buffer should be of ample size for all but the most unusual circumstance. Ensure that the byte count of the message being constructed is less than 8192 bytes. If it is, examine the command properties in the offending transaction. The most common cause of this sort of error is an incorrect Start Byte, End Byte or Number of Bytes value.
2. Make sure that the number of bytes written by a Write Data command are considered. This is set by the tag's device data format specification.

Write Check Sum command failed for address '<address.transaction>' - data conversion

Error Type:
Serious

Possible Cause:
A necessary data format conversion failed.

Solution:
If the problem is persistent, try to find another compatible data format. If dynamic ASCII formatting is used, make sure all necessary format characters are present in the table.

See Also:
Dynamic ASCII Formatting

Write Data command failed for address '<address.transaction>' - write buffer bounds

Error Type:
Serious

Possible Cause:
The command caused the driver to attempt to write past the maximum write buffer limit of 8192 bytes.

Solution:
1. The write buffer should be of ample size for all but the most unusual circumstance. Ensure that the byte count of the message being constructed is less than 8192 bytes. If it is, examine the command properties in the offending transaction. The most common cause of this sort of error is an incorrect Start Byte, End Byte or Number of Bytes value.
2. Make sure that the number of bytes written by a Write Data command are considered. This is set by the tag's device data format specification.

Write Data command failed for address '<address.transaction>' - data conversion

Error Type:
Serious

Possible Cause:
A necessary data format conversion failed.

Solution:
If the problem is persistent, try to find another compatible data format. If dynamic ASCII formatting is used, make sure all necessary format characters are present in the table.

See Also:
Dynamic ASCII Formatting

Write Device ID command failed for address '<address.transaction>' - write buffer bounds

Error Type:
Serious

Possible Cause:
The command caused the driver to attempt to write past the maximum write buffer limit of 8192 bytes.

Solution:
1. The write buffer should be of ample size for all but the most unusual circumstance. Ensure that the byte count of the message being constructed is less than 8192 bytes. If it is, examine the command properties in the offending transaction. The most common cause of this sort of error is an incorrect Start Byte, End Byte or Number of Bytes value.
2. Make sure that the number of bytes written by a Write Data command are considered. This is set by the tag's device data format specification.

Write Device ID command failed for address '<address.transaction>' - data conversion

Error Type:
Serious

Possible Cause:
A necessary data format conversion failed.

Solution:
If the problem is persistent, try to find another compatible data format. If dynamic ASCII formatting is used, make sure all necessary format characters are present in the table.

See Also:

Dynamic ASCII Formatting

Write String command failed for address '<address.transaction>' - destination buffer bounds

Error Type:
Serious

Possible Cause:
The command caused the driver to attempt to write past the maximum destination buffer limit of 8192 bytes.

Solution:
1. The destination buffer should be of ample size for all but the most unusual circumstance. Ensure that the byte count of the message being constructed is less than 8192 bytes. If it is, examine the command properties in the offending transaction. The most common cause of this sort of error is an incorrect Start Byte, End Byte or Number of Bytes value.
2. Make sure that the number of bytes written by a Write Data command are considered. This is set by the tag's device data format specification.

Tag update for address '<address>' failed due to data conversion error

Error Type:
Serious

Possible Cause:
A necessary data format conversion failed.

Solution:
If the problem is persistent, try to find another compatible data format. If **dynamic ASCII formatting** is used, make sure all necessary format characters are present in the table.

Unsolicited message receive timeout

Error Type:
Warning

Possible Cause:
The unsolicited mode "Receive timeout" expired while the channel was receiving a message. This could be caused by a delay in part of the message due to network traffic or gateway device, the data source, or an incorrectly configured transaction.

Solution:
Verify that the driver has been configured correctly for the expected messages. In particular, make sure the Read Response command at the beginning of each unsolicited transaction is set to terminate correctly. The use of Pause commands in the unsolicited transactions must be accounted for in the timeout setting. If the problem is due to wire time or hardware issues, increase the "Receive timeout" period accordingly. These messages can only be seen if the "Log unsolicited message timeouts" setting is checked.

See Also:
Define a Server Channel
Read Response Command
Pause Command

Unsolicited message dead time expired

Error Type:
Warning

Possible Cause:

This is caused when the driver receives an unsolicited message with an unknown key. Once the driver has received an unknown key, it waits one dead time period for the remainder of the message to come in.

Solution:
This is not necessarily a problem unless the driver was expected to process the message that caused this warning. If this is the case, users should check that the unsolicited transaction keys are properly defined. If choosing to ignore messages of this type, be aware that the driver will ignore all other incoming data for one dead time period after receiving each unhandled message. These messages can only be seen if the "Log unsolicited message timeouts" setting is checked.

See Also:
Unsolicited Transactions
Define a Server Channel

Move Pointer command failed for address '<address.transaction>'

Error Type:
Serious

Possible Cause:
An attempt was made to move a buffer pointer past the current frame bounds.

Solution:
Check the transaction definition.

Seek Character command failed for address '<address.transaction>' - label not found

Error Type:
Serious

Possible Cause:
The specified character was not found, and the given "Go to on failure" label was not found.

Solution:
Check the transaction definition. Make sure the label specified in the "Seek Character" command has been defined in that transaction.

Insert Function Block command failed for address '<address.transaction>' - Invalid FB

Error Type:
Serious

Possible Cause:
The function block inserted into the specified transaction may have since been deleted or renamed.

Solution:
Use the Transaction Editor to recreate the function block if necessary or to correct the name of the function block referenced in the transaction.

Unable to save password protected device profile in XML format

Error Type:
Serious

Possible Cause:
The device profile of one or more devices is password protected.

Solution:

The purpose of the password is to restrict unauthorized users from viewing and editing a device profile. Saving a project as XML will expose the information. Thus, save the project as an .opf file or remove all passwords in order to save as an XML file.

See Also:
Transaction Editor

XML Errors

The following error/warning messages may be generated. Click on the link for a description of the message.

XML Errors
XML Loading Error: The number of unsolicited transaction keys exceeds the set key length: <key length>
XML Loading Error: The two buffers of a <command> are the same. The buffers must be unique
XML Loading Error: The string '<string>' entered for a Write String command with format '<format>' is invalid
XML Loading Error: Range exceeds source buffer size of <max buffer size> bytes for a <command>

XML Loading Error: The number of unsolicited transaction keys exceeds the set key length: <key length>

Error Type:
Serious

Possible Cause:
1. The key length is incorrect.
2. There are extra unsolicited transaction keys in the XML.

Solution:
1. Verify that the key length is valid.
2. Verify that the keys are valid.

Note:
The project will not load.

XML Loading Error: The two buffers of a <command> are the same. The buffers must be unique

Error Type:
Serious

Possible Cause:
A buffer is being used twice in a single command.

Solution:
Verify that the buffers are unique.

Note:
The project will not load.

XML Loading Error: The string '<string>' entered for a Write String command with format '<format>' is invalid

Error Type:
Serious

Possible Cause:

1. Invalid ASCII Hex String from Nibble string.
2. Invalid ASCII String (packed 6 bit) string.

Solution:
1. For ASCII Hex String from Nibble string, only hex characters ('0' - '9' and 'A' - 'F') are allowed in the string. The string must be an even number of characters.
2. For ASCII String (packed 6 bit) string, the string must consist of characters supported in the ASCII Packed 6 bit table. The string length must be a multiple of four.

Note:
The project will not load.

XML Loading Error: Range exceeds source buffer size of <max buffer size> bytes for a <command>

Error Type:
Serious

Possible Cause:
The start byte plus the number of bytes exceeds the max buffer size.

Solution:
Verify that the sum of the start byte and the number of bytes is less than the max buffer size.

Note:
The project will not load.

Index

- _ -

- 1 -

- A -

- B -

- C -

- D -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- W -