Technical Document

NiagaraAX FlexSerial Driver Guide

Updated: June 12, 2008

NiagaraAX FlexSerial Driver Guide

Copyright © 2008 Tridium, Inc.
All rights reserved.
3951 Westerre Pkwy, Suite 350
Richmond
Virginia
23233
U.S.A.

NiagaraAX FlexSerial Driver Guide

June 12, 2008

This documents usage of the FlexSerial driver for the NiagaraAX framework.

FlexSerial Driver Overview

The purpose of the FlexSerial driver is to provide a generic serial driver that can be configured to communicate to a wide range of "simple" serial communicating devices. The driver allows the user to model the native device message structures. This includes message framing detail and message payload content. The message framing detail is pushed down to the low-level, send and receive processing threads, so that properly framed messages can be sent and received. Once messages have been defined, they can be assigned to request-response pairs that then, can be used in components that require communications to a device.

Installation

To use the NiagaraAX FlexSerial driver, you must have a target host that is licensed for "flexSerial". In addition, other device limits or proxy point limits may exist in your license.

From your PC, use the Niagara Workbench 3*.n.nn* installed with the "installation tool" option (checkbox "This instance of Workbench will be used as an installation tool"). This option installs the needed distribution files (*.dist* files) for commissioning various models of remote JACE platforms. The dist files are located under your Niagara install folder in various revision-named subfolders under the "sw" folder.

Apart from installing the 3*.n.nn* version of the Niagara distribution files in the JACE, make sure to install the **flexSerial** module too (if not already present, or upgrade if an older revision). For more details, see "About the Commissioning Wizard" in the *JACE NiagaraAX Install and Startup Guide*.

Following this, the station is now ready for flexSerial software integration, as described in the rest of this document.

Message Definition Concepts

This section discusses the message definition terminology as used in defining native device message structures within the FlexSerial driver.

A native device message is modeled as a FlexMessage. A FlexMessage is defined as a collection of FlexMessageBlocks and FlexMessageElements. A FlexMessageBlock is defined as a collection of FlexMessageElements.

FlexMessageElement

The FlexMessageElement is the basic building block for defining a FlexMessage. A FlexMessageElement is used to define a single element of a message. FlexMessageElements are used to define FlexMessageBlocks and FlexMessages.

FlexMessageBlock

A FlexMessageBlock is used as a wrapper for a collection of FlexMessageElements, which can then be use in the definition of multiple FlexMessages.

FlexMessage

A FlexMessage is used to model a complete native device message. It is defined by a collection of FlexMessageBlocks, and FlexMessageElements. Once defined the FlexSerial driver will have enough information to generate a serialized byte array that can be sent to the serial port and to also decode a serialized byte array that has been received from the serial port.

Message Framing

All serial protocols must have a way to define how a message begins and ends and is usually common to all messages. Typically it is defined as a sequence of one or more fixed byte values, but this is not always the case. The FlexSerial driver supports the definition of the message framing detail with special “frameStart” and “frameEnd” FlexMessageBlocks and a “maxReceiveSilentTime” property of the FlexSerialNetwork.

If a message always begins with the same sequence of bytes, this collection of bytes is defined in the “frameStart” FlexMessageBlock. If a message always ends with the same sequence of bytes, this collection of bytes is defined in the “frameEnd” FlexMessageBlock. In some cases a message frame is defined by silent time on the serial port. In this case, the silent time in milliseconds is defined in the “maxReceiveSilentTime” property of the FlexSerialNetwork and the frameStart and frameEnd FlexMessageBlocks are left undefined.

Defining FlexMessageElements

As previously stated FlexMessageElements are used to define FlexMessageBlocks and FlexMessages. The purpose of the FlexMessageElement is to completely define a primitive element of a message. It must contain enough information to serialize this element value to and from a native byte array. The FlexSerial driver provides a special view and editor that can be used to create FlexMessageElements.

There are six different types of FlexMessageElements. They are:

1. Flex**Byte**Element – defines a single byte message element.
2. Flex**Word**Element – defines a 2 byte message element.
3. Flex**Integer**Element – defines a 4 byte message element.
4. Flex**Float**Element – defines a 4 byte IEEE floating point element.
5. Flex**String**Element – defines a string element. It uses facets to define exactly how the string is stored.
6. Flex**Marker**Element **-** a marker is a special named place holder and consumes no space in the message but defines an offset into the message. “cksumStart” is a special named marker that defines the message offset at which checksum calculation starts. It defaults to 0 if this marker is not defined.

Each of the six FlexMessageElement types is a component, and has properties as discussed in the next section, “FlexMessageElement Properties”.

FlexMessageElement Properties

Common among all FlexMessageElement component types are properties listed in Table 1.

Table 1 FlexMessageElement properties

Properties	**Descriptions**
name	user defined name for this message element. Unique names for use across different message blocks are recommended (See “Notes on Building Messages” on page 11).
offset	byte offset of this message element in parent FlexMessageBlock or FlexMessage. Information only, automatically calculated.
size	byte size of this element. Information only, automatically set based on dataType.

Properties	Descriptions
dataType	Data types supported are byte, word, integer, float, string, and marker. **byte**: a single byte (8 bits) of data **word**: two bytes of data **integer**: four bytes of data **float**: a four byte IEEE floating point number **string**: string data can be variable length. It uses the facets property to define exactly how string is stored. **marker**: a marker is a special named placeholder and consumes no space in the message but defines an offset into the message. "cksumStart" is a special named marker that defines the message offset at which checksum calculation starts. It defaults to 0 if this marker is not defined.
value	The actual value of this data item, and is dependent on the dataType specified. The value can be a constant or an indirect value from the object defined by the source property. ***NOTE:*** If entering a constant, it must be in *decimal* format, regardless of "encode" property setting. Often, a calculator or conversion chart is useful when building message elements.
encode	To select special encoding used. Primarily, this applies to the encoding of received (Response) messages. Encodes supported are none, ascii, and asciiHex, where: **none**: No special encoding is applied, raw binary. **ascii**: The value data is converted to an ascii string. **asciiHex**: The value data is converted to an ascii hex string. ***NOTE:*** See "FlexMessageElement Encode property usage" for details about encode in Request and Response Message operation, which varies by FlexMessageElement type.
facets	This is used to define other special name-value pairs used to support this message element. The following is a list of facet name-value pairs that are used by the FlexSerial driver "**bigEndian**"- boolean: defines how multi-byte numeric values are placed in the byteArray. "**activeValue**"- integer: defines the numeric value used to represent a boolean active value in the byteArray. "**inactiveValue**"-integer: defines the numeric value used to represent a boolean inactive value in the byteArray. "**padWidth**"-integer: used with strings to define a value to be used to pad out a fixed length string message element. "**nullTerminate**"- boolean: used with strings to define that the string value is terminated with a null character. "**fieldWidth**"-integer: used with strings to specify a fixed length string message element. The padWidth facet defines the value used for padding. "**trueText**"-string: used with boolean values and ascii encoding. "**falseText**"-string: used with boolean values and ascii encoding.
source	This is an ord that points to another value within the station database. The value of this source object will be copied into the value of this message element. If the source is null, then the value of this message element will be a constant. Special consideration for Program object source. If the source is a Program object, it is assumed that the Program object is used to calculate an error-check message element value such as a checksum or CRC value. See the "Checksum or CRC Message Element" section for related details.
exposeInParent	This is a boolean value and is used to cause this message element's value to be exposed as a slot in the parent FlexResponseMessage

FlexMessageElement Encode property usage

Among the properties of any FlexMessageElement type (see Table 1 for properties) is an "encode" property. Three selections are available: "None", "Ascii", and "AsciiHex". Usages are described below in Table 2 by FlexMessageElement type, in Request and Response Messages.

Table 2 FlexMessageElement encode property usage with Request and Response Messages

Encode	FlexElement Type	Request Message	Response Message
None	String	Writes string value. Uses fieldWidth, nullTerminate, and delimiter facets.	Reads input string value. Uses fieldWidth, nullTerminate, and delimiter facets .
	Byte	Writes raw binary byte value.	Reads raw byte value.
	Word	Writes raw binary 2 byte value. Uses bigEndian facet to define byte order.	Reads raw 2 byte value. Uses bigEndian facet to define byte order
	Integer	Writes raw binary 4 byte value. Uses bigEndian facet to define byte order.	Reads raw 4 byte value. Uses bigEndian facet to define byte order.
	Float	Writes raw binary 4 byte float value. Uses bigEndian facet to define byte order.	Reads raw 4 byte float value. Uses bigEndian facet to define byte order.
Ascii	String	Not used (same as None).	Not used (same as None).
	Byte	Not used (same as None).	Treats the input data as a decimal string and attempts to convert it to an integer value. May need to set fieldWidth, nullTerminate, or delimiter facet in order to parse the received string properly.
	Word	Not used (same as None).	
	Integer	Not used (same as None).	
	Float	Not used (same as None).	Same as above except it will convert the string to a float value.
AsciiHex	String	Not used (same as None).	Not used.
	Byte	Converts the least significant 8 bits of the value to an asciiHex string. For example: 255 —> "ff" 256 —> "00" 257 —> "01"	Treats the input data as a decimal string and attempts to convert it to an integer value. May need to set fieldWidth, nullTerminate, or delimiter facet in order to parse the received string properly.
	Word	Same as above, except uses 16 bits or 2 bytes. Uses bigEndian facet to define byte order.	Same as above. Uses bigEndian facet to define byte order.
	Integer	Same as above, except uses 32 bits or 4 bytes.	
	Float	Converts the 4 byte float value to an asciiHex string. Uses bigEndian facet to define byte order.	Same as above except it will convert the string to a float value. Uses bigEndian facet to define byte order.

Defining FlexMessageBlocks

FlexMessageBlocks are defined by creating a collection of FlexMessageElements under an instance of a FlexMessageBlock. All FlexMessageBlocks are defined under the frozen "messageBlocks" slot of the FlexSerialNetwork.

Defining FlexMessages

FlexMessages are defined by creating a collection of FlexMessageBlock references and FlexMessageElements under an instance of a FlexMessage. All FlexMessages are defined under the frozen "messages" slot of the FlexSerialNetwork.

Checksum or CRC Message Element

Many serial protocols contain a checksum or CRC field that contains a calculated one or two-byte value. This value is calculated from some or all of the preceding bytes of the message. The FlexSerial driver supports the calculation of this error-checking value with the use of a Program object. The user must understand the protocol algorithm used to calculate the error-checking value and implement this algorithm in the "onExecute" method of the Program object. The Program object must be defined with the following slots:

- "offset" – Integer: This is the offset within the byte array where the error-checking calculation starts. This is automatically set based on any "cksumStart"-named marker message elements defined in the message (see dataType on page 6).
- "byteArray" – Blob: This is the byte array of the message up to the point of this error-checking value. This will automatically be set when the FlexSerial driver is generating the byte array for this message.
- "results" – Integer: This slot will contain the results of the error-checking algorithm implemented in the "onExecute" method of the Program object.

A FlexMessageElement can be defined to be an error check message element simply by specifying the message element source ord as this Program object.

FlexRequestResponse component

A FlexRequestResponse component ties a FlexRequestMessage and a FlexResponseMessage pair together as a request-response pair.

FlexRequestMessage component

A FlexRequestMessage component allows the user to select one of the defined FlexMessages under the FlexSerialNetwork/messages component to be a request message. When FlexMessage is selected, an instance of the message will be created as a child with any FlexMessageBlocks defined in the FlexMessage expanded FlexMessageElements so that this FlexMessage instance only contains FlexMessageElements.

Table 3 FlexRequestMessage slots

Frozen Slots	Descriptions
message	Allows user to select a defined FlexMessage to be this request message
facets	Allows user to specify a "showAscii" facet to allow ascii formatted display of the following byteArray slot.
byteArray	This is the byte array form of this request message

FlexResponseMessage component

A FlexResponseMessage component allows the user to select one of the defined FlexMessages under the FlexSerialNetwork/messages component to be a response message. When FlexMessage is selected an instance of the message will be created as a child with any FlexMessageBlocks defined in the FlexMessage expanded FlexMessageElements so that this FlexMessage instance only contains FlexMessageElements.

Table 4 FlexResponseMessage slots

Frozen Slots	Descriptions
message	Allows user to select a defined FlexMessage to be this response message.
facets	Allows user to specify a "showAscii" facet to allow ascii formatted display of the following byteArray slot.
byteArray	This is the byte array form of this response message
elementSelect	This allows the user to select one of the message elements of this response, to be the "primary" value of the response. This is primarily used when this response is a part of a FlexProxyExt "pollMessage" FlexRequestResponse component. It will be this response message element that will be stored in the readValue slot of the proxyExt.
errorCheck	This is an ord that will point to an "errorCheck" Program object. If this ord is null, no error checking will be done on the response message. The defined Program object will be initialized with both the request and response byte arrays prior to invoking the "onExecute" method of the Program object. The "onExecute" method will perform required logic to validate that the response is valid for the request. The Program object must have a "results" String slot that will contain the results of this error check algorithm. If everything is "Ok" the results slot should be set to "Ok". If there is an error of some type, the results slot should be set to a string that describes the error. This error string will be displayed in the faultCause slot of the FlexProxyExt.

FlexUnsolicitedMessage component

A FlexUnsolicitedMessage is typically used to decode an unsolicited message byte array exposed in the FlexSerialNetwork. This message component must be used as a child of a FlexSerialNetwork.

Table 5 FlexUnsolicitedMessage slots

Frozen Slots	Descriptions
message	Allows user to select a defined FlexMessage to be used to decode this message.
facets	Allows user to specify a "showAscii" facet to allow ascii formatted display of the following byteArray slot.
byteArray	This is the byte array that is to be decoded. It is typically linked to the unsolicitedByteArray of the parent FlexSerialNetwork.
elementSelect	This property is not used in this application.

Frozen Slots	**Descriptions**
messageValidate	This is an ord that will point to an "errorCheck" Program object. If this ord is null, no error checking will be done on the message and the byte array will be decoded. The defined Program object's responseByteArray will be initialized prior to invoking the "onExecute" method of the Program object. The "onExecute" method will perform required logic to validate that the response is valid for the request. The Program object must have a "results" String slot that will contain the results of this error check algorithm. If everything is "Ok" the results slot should be set to "Ok". If there is an error of some type, the results slot should be set to a string that describes the error. If the results is "Ok" then the byte array will be decoded.
unsolicitedMessage Received	This is a Topic that is fired when this object decodes a valid message.

FlexSendMessage component

A FlexSendMessage is typically used to send a message either manually or on an event. This message component must be used as a child of a FlexSerialNetwork. It can be used in conjunction with the FlexUnsolicitedMessage to send a reply to an unsolicited message.

Table 6 FlexSendMessage slots

Frozen Slots	**Descriptions**
message	Allows user to select a defined FlexMessage will be sent when the send action is invoked.
facets	Allows user to specify a "showAscii" facet to allow ascii formatted display of the following byteArray slot.
byteArray	This is the sent byte array.
enable	This is a StatusBoolean input that will enable/disable the sending of this message.
Send	This is an Action that will cause the message to be sent if the enable input is true.

SerialRequest component

A SerialRequest can be used to send a string request and receive a string response.
This message component must be used as a child of a FlexSerialNetwork.

Table 7 SerialRequest slots

Frozen Slots	**Descriptions**
Request	This is a StatusString input. When this input changes the message will automatically be sent.
addFrameStart	If set to true, the driver will add bytes defined in the frameStart MessageBlock to the request message before sending.
addFrameEnd	If set to true, the driver will add bytes defined in the frameEnd MessageBlock to the request message before sending.
Response	This is a StatusString output. The response received will be exposed here.
responseExpected	If set to true, the driver will wait for a response to the request message sent.
addFrameStart	If set to true, the driver will remove the bytes defined in the frameStart MessageBlock from the response message.

Frozen Slots	Descriptions
stripFrameEnd	If set to true, the driver will remove the bytes defined in the frameEnd MessageBlock from the response message.
SendRequest	Invoking this Action that will cause the message to be sent.

SerialSend component

A SerialSend can be used to send a string request.
This message component must be used as a child of a FlexSerialNetwork.

Table 8 SerialSend slots

Frozen Slots	Descriptions
In	This is a StatusString input. When this input changes the message will automatically be sent.
addFrameStart	If set to true, the driver will add bytes defined in the frameStart MessageBlock to the request message before sending.
addFrameEnd	If set to true, the driver will add bytes defined in the frameEnd MessageBlock to the request message before sending.
Send	Invoking this Action that will cause the message to be sent.

Unsolicited Messages

The FlexSerial driver can also be used to receive and process unsolicited messages. When an unsolicited message is received it is exposed in the unsolicitedMessage property of the FlexSerialNetwork as a StatusString. It is also exposed in the unsolicitedByteArray property as a Blob byte array. After the message is written to these two properties it fires the FlexSerialNetwork's unsolicitedMessageReceived topic. The FlexUnsolicitedMessage component can be used to decode the unsolicited message. KitControl's string processing components may also be used to decode the unsolicited message.

Notes on Building Messages

The following notes relate to building messages, and include the following topics

- Naming of Elements
- Updating Message Instances
- Show Instance Action
- AsciiHex Encode with showAscii Facet

Naming of Elements

MessageBlocks can be used to build up reusable fragments to later piece together by defining messages in the Message Manager. When you **Add** a new element (FlexMessageElement) to a MessageBlock, the *name* of the element defaults to the *type* of element that you are adding, for example, "FlexByteElement".

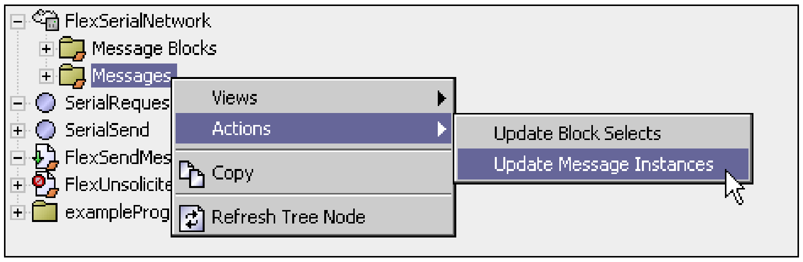
Within the boundaries of a single MessageBlock, autonaming is OK, as duplicate names are prevented (e.g., another added FlexByteElement is autonamed "FlexByteElement1"). However, be aware that *issues* can occur *when element names* across MessageBlocks *are not unique*—as actual messages are comprised of one or more MessageBlocks and one or more additional elements. So, when a message is created, it takes copies of MessageBlocks and adds their components (elements), which may result in a “duplicate name exception” and incomplete message being sent.

Therefore, it is recommended that when creating elements (FlexMessageElements), that you give them **unique names**, that is unique across the various MessageBlocks.

Updating Message Instances

After updating the configuration of a message in the Message Manager (including but not limited to element name, value or encoding changes), be aware that any objects that are configured to use that message are using a stored *instance* of that message—and therefore updates are *not* automatic.

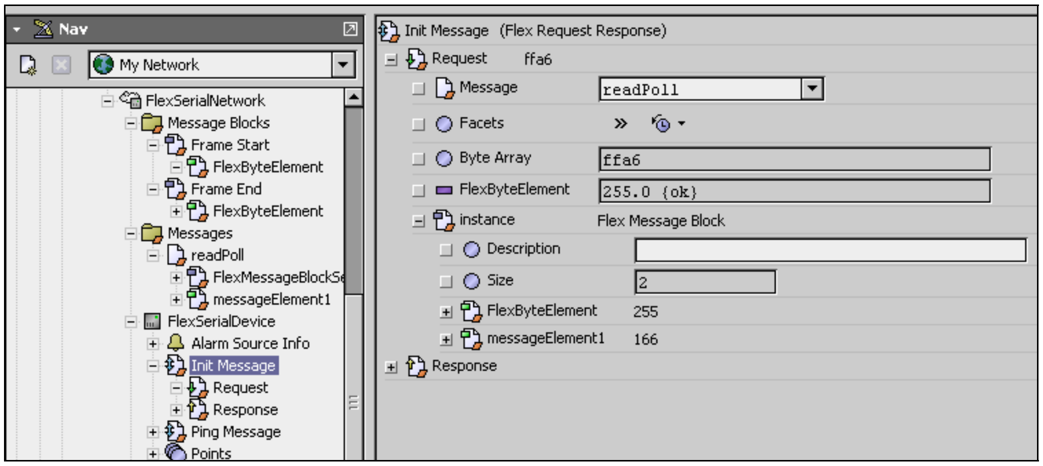
For this reason, after updating one or more messages, you should invoke the right-click “Update Message Instances” action on its parent MessageFolder, or click the equivalent “Update Messages” button in the Message Folder Manager view.



In the same way, if you have modified a MessageBlock that has already been added to a message, you should invoke the “Update Block Selects” action on its MessageFolder.

Show Instance Action

Note that Request and Response messages include an *action* that allows either showing or hiding the message *Instance*. If you invoke the “Show Instance” action, then the instance of the message (as sent out on the wire) is shown in the message’s property sheet.



Request	ffa6
Message	readPoll
Facets	»
Byte Array	ffa6
FlexByteElement	255.0 {ok}
instance	Flex Message Block
Description	
Size	2
FlexByteElement	255
messageElement1	166
Response	

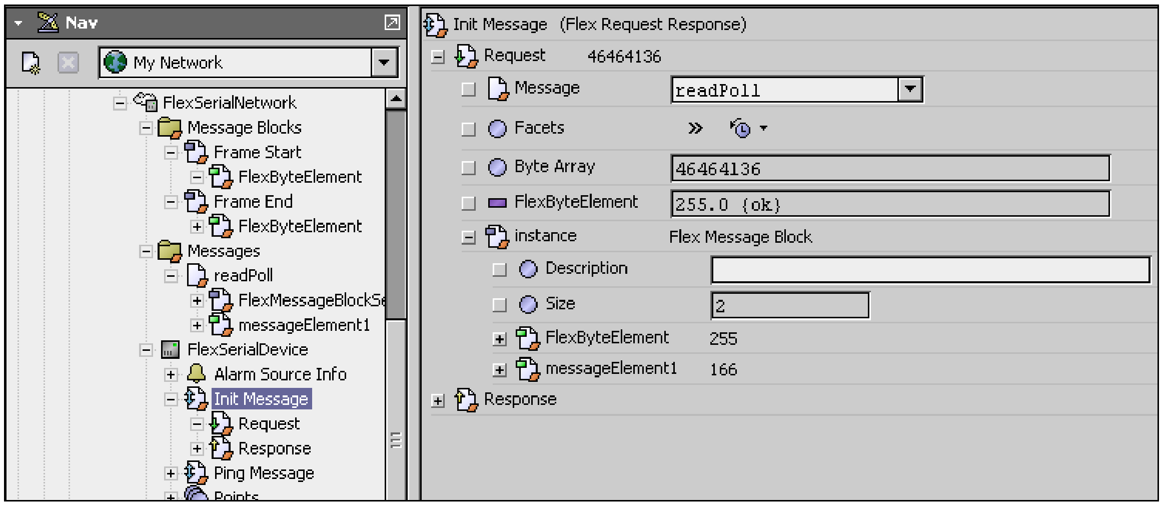
The Instance is broken down into each of the individual message elements, each one as a slot. This feature can be helpful for both troubleshooting and verifying how the message is created from the various MessageBlocks and FlexMessageElements.

AsciiHex Encode with showAscii Facet

Note that in some cases a manufacturer may provide message examples in hex format, but require messages to be sent on the wire in "ASCII hex" format. Using defaults, this can be somewhat confusing to verify or debug in the message property sheet.

For example, say the message "FF A6" must be sent out in ASCII hex. When you build each of the two message elements, you must enter a decimal value, which you pre-converted to the decimal values 255 and 166, respectively. Since the vendor requires ASCII hex, you set the "encode" property in each of the elements to "AsciiHex".

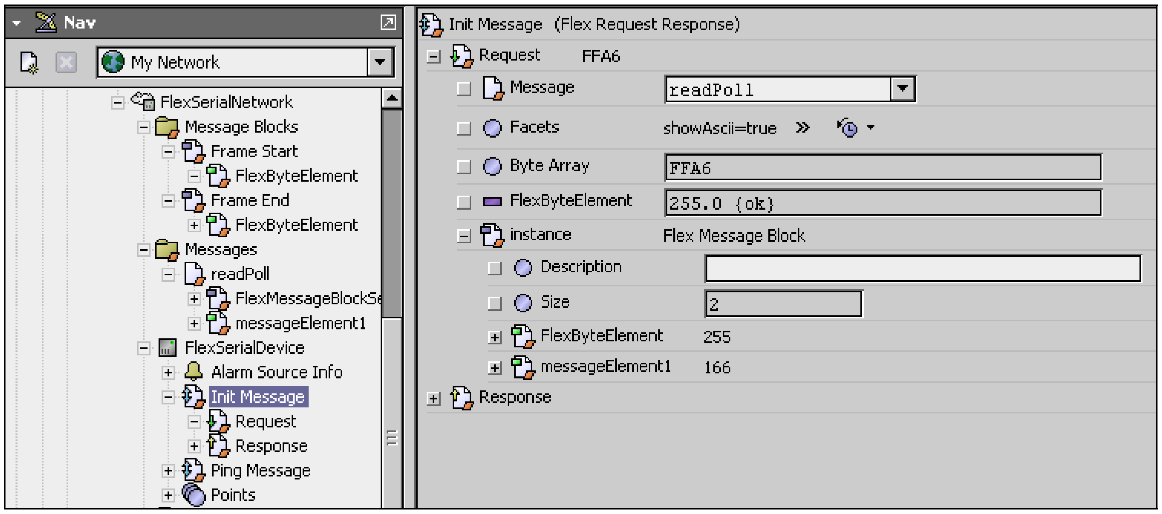
When the property sheet of that message is viewed, the information sent (ByteArray) reflects this extra encoding, instead of the expected hex value of "FF A6". So in this case you see the more cryptic "46464136", as shown below.



```
Init Message  (Flex Request Response)
Request          46464136
  Message          readPoll
  Facets
  Byte Array       46464136
  FlexByteElement  255.0 {ok}
  instance         Flex Message Block
    Description
    Size           2
    FlexByteElement    255
    messageElement1    166
Response
```

In order to verify, you must use a calculator or conversion chart to make sure that "46464136" reflects the desired hex number (in this case FF A6) in ASCII format.

However, a *better solution* is to add a facet named "showAscii", of type boolean, to the message, which perform this same conversion—provided that you set this facet's value to true. Note that "showAscii" does not appear in the drop-down list of facets—you must type it in. Below is the property sheet of this same example message after adding the "showAscii" facet.



```
Init Message  (Flex Request Response)
Request          FFA6
  Message          readPoll
  Facets           showAscii=true
  Byte Array       FFA6
  FlexByteElement  255.0 {ok}
  instance         Flex Message Block
    Description
    Size           2
    FlexByteElement    255
    messageElement1    166
Response
```

If this "showAscii" facet is missing or set to false, then the ASCII hex format is shown instead.

Examples

Modbus RTU Protocol

Note: The examples include here are from a station named “flexmodbusrtu” that is also provided along with this document. This example station uses a ModbusSlave driver using COM4. The FlexSerial driver is using COM1. A RS-232 cable is connected between COM1 & COM4.

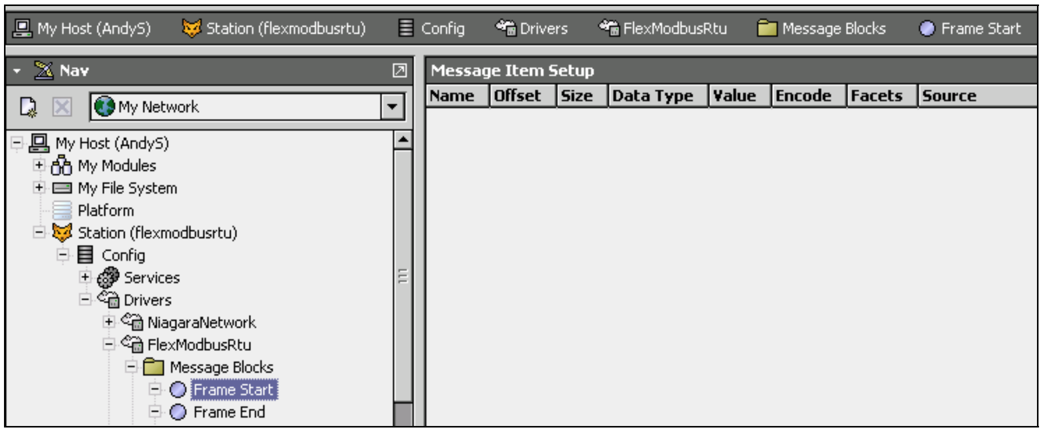
Modbus RTU protocol is a request-response binary protocol and message framing is defined by silent time between messages. A message start is defined as the first byte received after a silent time of at least 3.5 character times. A message end is defined as 3.5 character silent time after data has been on the wire. Message is then defined as follows:

start	device addr	function	data	crc	end
silent	8 bits	8 bits	n bytes	16 bits	silent

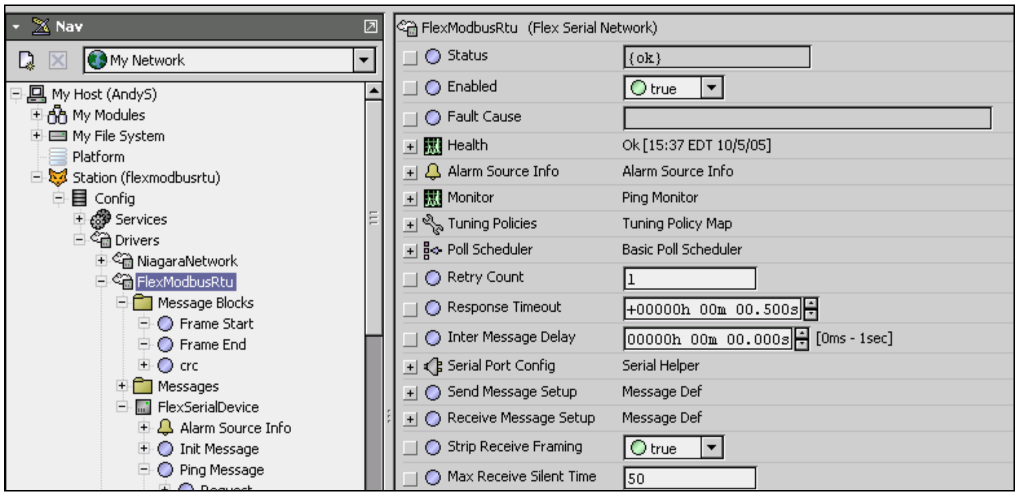
As can be seen this protocol does not have a fixed sequence of bytes that frame the message but is framed by silent time on the wire.

Define Message framing in FlexSerialNetwork

Since there is no fixed sequence of bytes that frame the message leave the “frameStart” and “frameEnd” messageBlocks empty.



And since the message is framed by receive silent time, set the “maxReceiveSilentTime” property of the FlexSerialNetwork to 50 milliseconds.



FlexModbusRtu (Flex Serial Network)

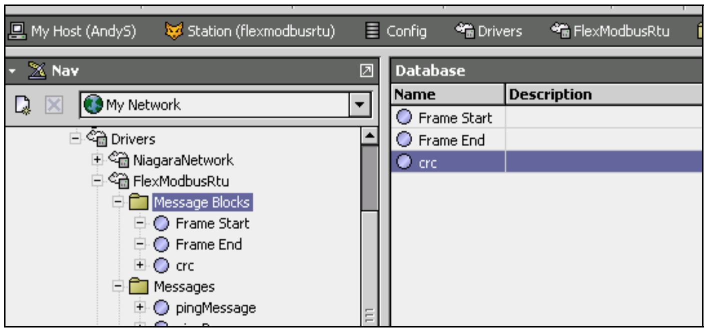
Status	{ok}
Enabled	true
Fault Cause	
Health	Ok [15:37 EDT 10/5/05]
Alarm Source Info	Alarm Source Info
Monitor	Ping Monitor
Tuning Policies	Tuning Policy Map
Poll Scheduler	Basic Poll Scheduler
Retry Count	1
Response Timeout	+00000h 00m 00.500s
Inter Message Delay	00000h 00m 00.000s [0ms - 1sec]
Serial Port Config	Serial Helper
Send Message Setup	Message Def
Receive Message Setup	Message Def
Strip Receive Framing	true
Max Receive Silent Time	50

Define Crc message block

Modbus RTU protocol message uses a 16-bit CRC error-checking field in the message. Code examples of the CRC algorithm are provided in the *Modbus Protocol Reference Guide* provided on the modbus.org web site. In this example this algorithm has been implemented in the Program object at: station:|slot:/Drivers/FlexModbusRtu/messages/crcCalc

Since the CRC is part of all messages and it will always be associated with the same Program object it is best to define this as a MessageBlock. To do this use the MessageBlockManager view of the MessageBlocks component under the FlexSerialNetwork, use the New button of the view to add a new message block named "crc" and give it a description if you desire.

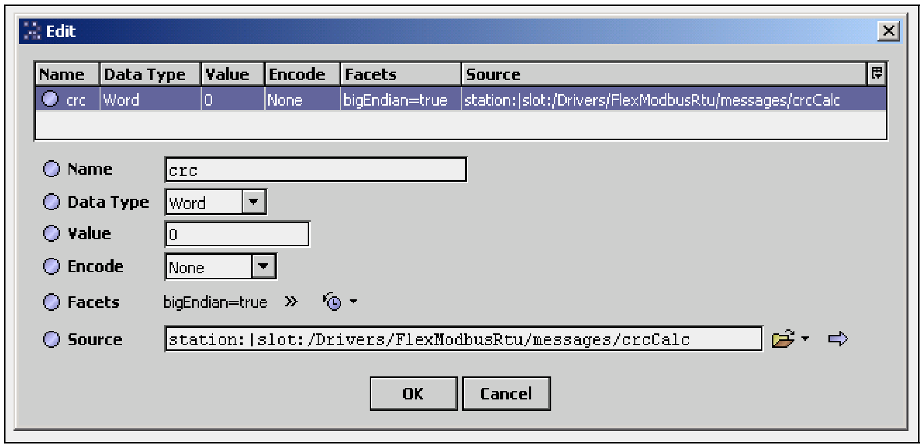
The view should now look something like this:



Name	Description
Frame Start	
Frame End	
crc	

Now double-click on the crc message block in the Nav tree to bring up the MessageManager view.

Use the New button to create a new FlexMessageElement define this message element as follows:



Name	Data Type	Value	Encode	Facets	Source
crc	Word	0	None	bigEndian=true	station:|slot:/Drivers/FlexModbusRtu/messages/crcCalc

Name	crc
Data Type	Word
Value	0
Encode	None
Facets	bigEndian=true
Source	station:|slot:/Drivers/FlexModbusRtu/messages/crcCalc

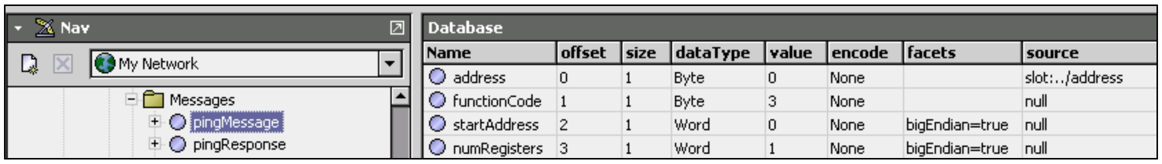
Data type is chosen to be word because this is a 16 bit value. Value is a "don't care" because it will be filled in from the source Program object. Encode is none because this will be transmitted as a raw binary value. A "bigEndian" = true facet is set because sent with most significant 8 bits of the crc sent first. Use the Component Chooser to select desired Program object.

Define a PingMessage

This ping message will be used by the FlexSerialDevice to monitor the communication status of the device. In this example, we will be reading holding register 0. The Modbus message used to read a holding register has the following format:

device address	**Address of this device**
function	03
register address hi byte	00
register address lo byte	00
no. of registers hi byte	00
no. of registers lo byte	01
crc hi byte	xx
crc lo byte	xx

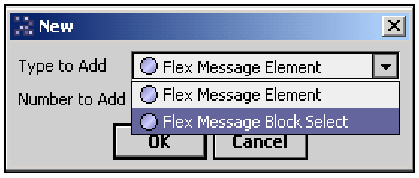
To create this message double-click on the FlexSerialNetwork/messages component in the Nav tree to open the MessageFolderManager view. Use the New button to create a new FlexMessage named "pingMessage". Then double-click on the "pingMessage" in the Nav tree to open the FlexMessageManager view. Use the New button to create FlexMessageItems as shown below:

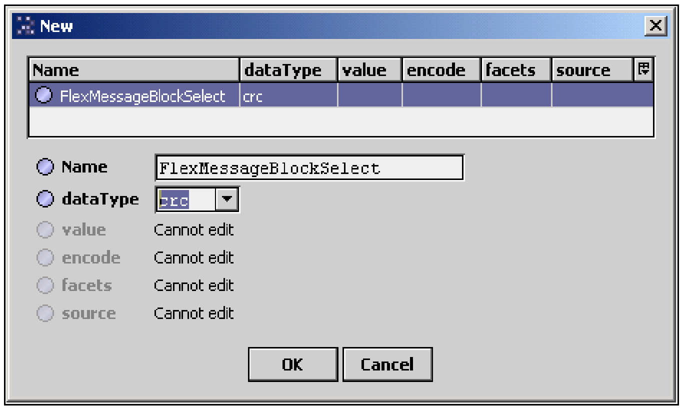


Name	offset	size	dataType	value	encode	facets	source
address	0	1	Byte	0	None		slot:../address
functionCode	1	1	Byte	3	None		null
startAddress	2	1	Word	0	None	bigEndian=true	null
numRegisters	3	1	Word	1	None	bigEndian=true	null

Note on the address message element: The address element is defined with a relative ord definition. Since an instance of this ping message is going actually reside as a child of the FlexSerialDevice, and this same ping message will be likely be used in all FlexSerialDevice instances, the source for the address is the address property of the parent FlexSerialDevice, which is defined an ord as: "slot:../address"

Next use the New button to add a new FlexMessageBlock Select named "crc".



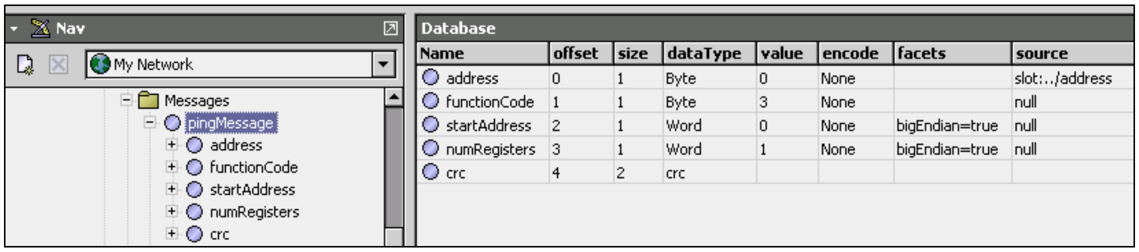


Name	dataType	value	encode	facets	source
FlexMessageBlockSelect	crc				

Name	FlexMessageBlockSelect
dataType	crc
value	Cannot edit
encode	Cannot edit
facets	Cannot edit
source	Cannot edit

Select "crc" dataType and press OK.

Now the pingMessage should look like this:



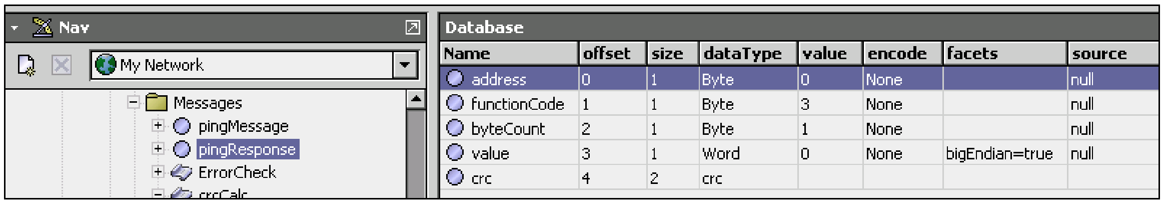
Name	offset	size	dataType	value	encode	facets	source
address	0	1	Byte	0	None		slot:../address
functionCode	1	1	Byte	3	None		null
startAddress	2	1	Word	0	None	bigEndian=true	null
numRegisters	3	1	Word	1	None	bigEndian=true	null
crc	4	2	crc				

Define a PingResponse Message

The response to a Modbus read holding register message has the following format:

device address	should be same as request
function	03 same as request
byte count	02 in this case
data hi byte	nn
data lo byte	nn
crc hi byte	xx
crc lo byte	xx

Using a procedure similar to what was used to define the PingMessage create a "pingResponse" message defined as follows:

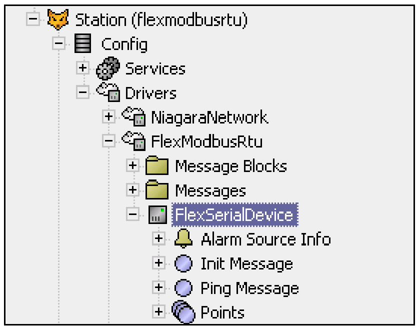


Name	offset	size	dataType	value	encode	facets	source
address	0	1	Byte	0	None		null
functionCode	1	1	Byte	3	None		null
byteCount	2	1	Byte	1	None		null
value	3	1	Word	0	None	bigEndian=true	null
crc	4	2	crc				

Since this message is going to be used as a response message definition, the value of each message element doesn't matter. These values will actually be set when a real response message is decoded into this message structure.

Create a FlexSerialDevice and define its PingMessage

Use the FlexDeviceManager view of the FlexSerialNetwork to add a new FlexSerialDevice with an address of "1". The Nav tree should now look something like this:

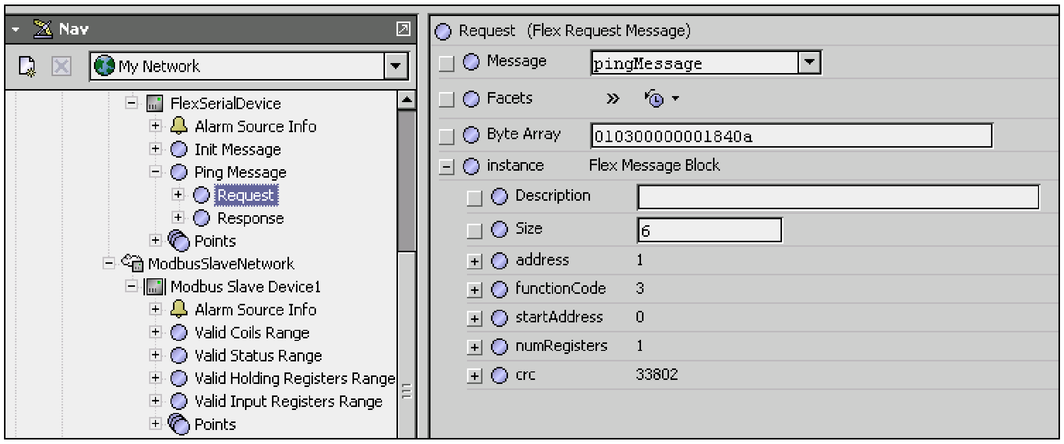


The Init Message and Ping Message are FlexRequestResponse frozen component slots of the device.

- The Init Message is used to define a message that needs to be sent to the device to before other communications can be processed. In this case, no Init Message is needed and left un-initialized.
- The PingMessage is used to define the request and response FlexMessages to be used when this device is to be pinged.

Define the PingMessage Request message

Expand the Ping Message and double-click on the request in the Nav tree. This will open the property sheet of the request FlexRequestMessage.



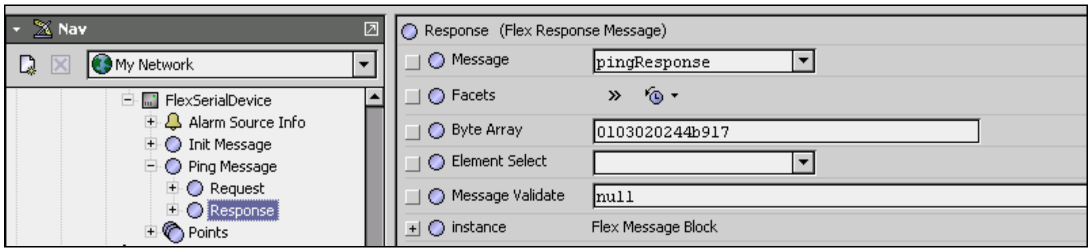
Request (Flex Request Message)
Message pingMessage
Facets
Byte Array 010300000001840a
instance Flex Message Block
Description
Size 6
address 1
functionCode 3
startAddress 0
numRegisters 1
crc 33802

Initially the Message and Byte Array will be blank and the instance will not exist.
Click the pull-down of the Message property and select the pingMessage created in the earlier step. Note: all of the messages defined in the FlexSerialNetwork Messages folder will appear in this pull-down.

Once the message is selected and saved an instance FlexMessageBlock should appear in the property sheet. At some point data in the Byte Array property will appear indicating that a ping message has actually been sent to the device.

Define the PingMessage Response message

Using a procedure similar to the above double-click on the Response and select the "pingResponse" as the response message.



Response (Flex Response Message)
Message pingResponse
Facets
Byte Array 0103020244b917
Element Select
Message Validate null
instance Flex Message Block

You will notice that the Response structure also has Element Select and Message Validate properties. These properties are primarily used in defining Poll Message responses under FlexProxyExt and can be left un-initialized here. If the device responds to a ping message, it is typically considered to be OK.

Assuming that the station is running in a JACE that is actually connected to a Modbus RTU device with this address you should be able to force a Ping message to the device and see a request and response and the device will have the {down} flag cleared.

In the example “flexmodbusrtu” station, the there is a modbusSlave driver also running using COM4. This FlexSerialNetwork is using COM1, so connecting COM1 to COM4 should complete the communications path.

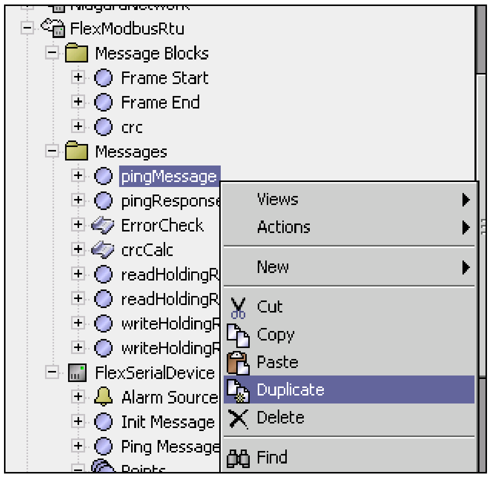
Control Points and FlexProxyExt Usage

Most of the register data from a Modbus device will be exposed in Niagara using control points with FlexProxyExts. The method to create and setup these control points is consistent with other AX driver implementations. In this example we will need to setup the request and response messages used by the FlexProxyExt to read the holding register defined by the address property of the FlexProxyExt. This request-response pair is selected by the FlexProxyExt’s “pollMessage” property.

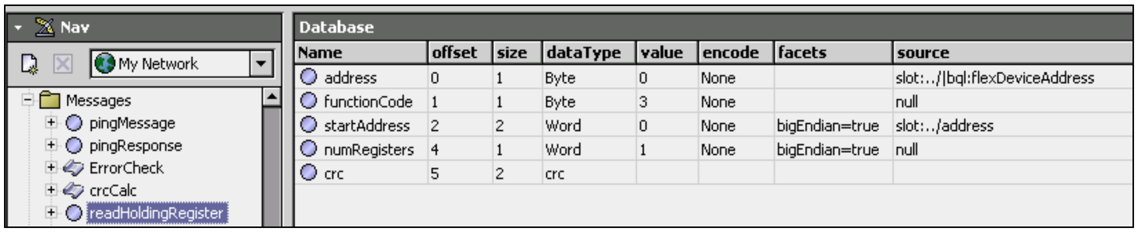
Define Read Holding Register Message

The read holding register message that will be used by a proxy extension is almost identical to the ping message defined earlier. The Modbus message structure is the same but the source ord used to get the device address is different since the proxy extension is several layers deeper in the station database. Another difference is the fact that the register address to be read needs to be sourced from a property of the proxy extension.

Since this read holding register message almost identical to the ping message, using the Nav tree, select the “pingMessage” and duplicate it, renaming it to “readHoldingRegister”.



Next double click on “readHoldingRegister” in the Nav tree to open the FlexMessageManager. Change the source property of the address and startAddress message elements as shown below.



Name	offset	size	dataType	value	encode	facets	source
address	0	1	Byte	0	None		slot:../|bql:flexDeviceAddress
functionCode	1	1	Byte	3	None		null
startAddress	2	2	Word	0	None	bigEndian=true	slot:../address
numRegisters	4	1	Word	1	None	bigEndian=true	null
crc	5	2	crc				

Note the address message element source ord is defined as a relative ord and makes use of a Bql statement. This message element’s value needs to be replaced with the address property value of the parent FlexSerialDevice. The FlexProxyExt has a "getFlexDeviceAddress" method, which will return content of the address property of the parent FlexSerialDevice. The "slot:../bql:flexDeviceAddress" ord will invoke the "getFlexDeviceAddress" method on the parent FlexProxyExt.

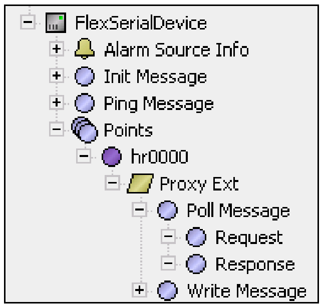
The startAddress message element source ord is also defined as a relative ord. This message element's value needs to be replaced with the address property of the parent FlexProxyExt.

Define Read Holding Register Response Message

The read holding register response message is actually identical to the pingResponse defined earlier, but to keep things consistent we are going to duplicate the pingResponse message and name it "readHoldingResponse".

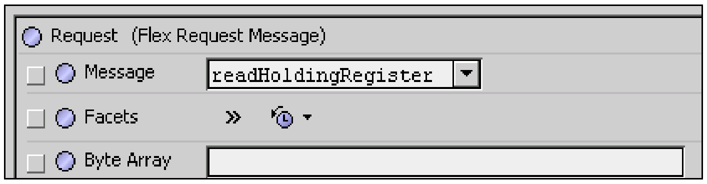
Create a NumericWritable Point and Setup PollMessage

Using the Point Manager view of the FlexSerialDevice create a NumericWritable point and name it "hr0000" with an address of 0. The Nav tree should now appear like this:



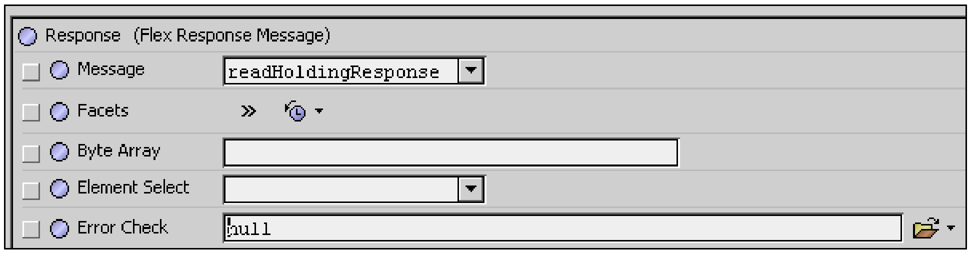
The next step is to select the message to be used as the pollMessage/request and the pollMessage/response.

Double-click on the pollMessage/request in the Nav tree to open its property sheet. In the message pull-down select "readHoldingRegister".


Request (Flex Request Message)

Message	readHoldingRegister
Facets	
Byte Array	

Double-click on the pollMessage/response in the Nav tree to open its property sheet. In the message pull-down select "readHoldingResponse" and press the save button.

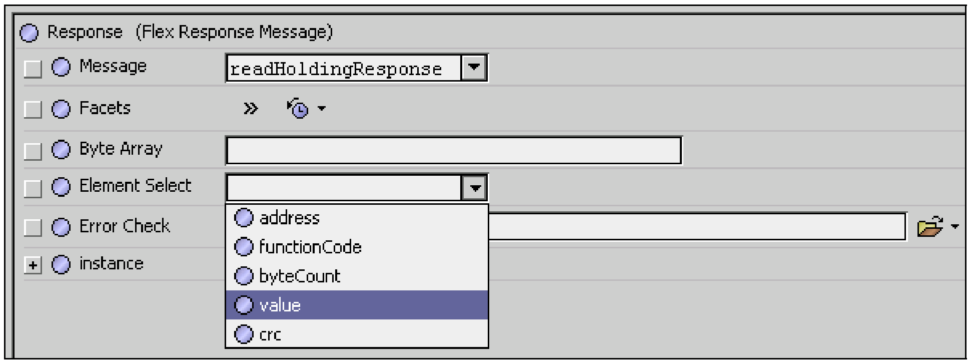

Response (Flex Response Message)

Message	readHoldingResponse
Facets	
Byte Array	
Element Select	
Error Check	null

Now press the "refresh" button. This will allow selecting one of the message elements of the readHoldingResponse message to be the value that will be processed as the readValue of the

FlexProxyExt. This value in turn will be exposed as the out property of the parent control point. For the readHoldingResponse message the "value" message element is this value.

Use the Element Select pull-down and select "value".



Response (Flex Response Message)

Message	readHoldingResponse
Facets	
Byte Array	
Element Select	address / functionCode / byteCount / value / crc
Error Check	
instance	

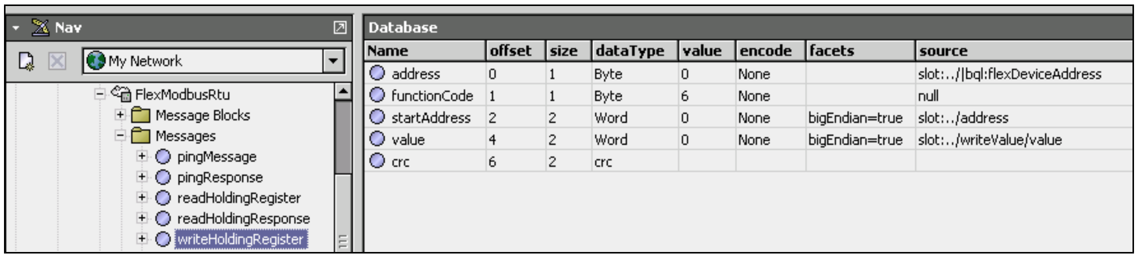
The pollMessage is now setup and the driver should be able to generate a valid Modbus RTU message to read holding register 0 when the parent control point is subscribed.

Define a Write Holding Register Message

The write holding register message that will be used by a proxy extension will model a Modbus RTU Preset Single Register message. This message has the following byte format:

device address	**address of this Modbus device**
function	06 preset single register function
register hi byte	nn
register lo byte	nn
preset data hi byte	dd
preset data lo byte	dd
crc hi byte	xx
crc lo byte	xx

Using methods described earlier create a "writeHoldingRegister" message like this:



Name	offset	size	dataType	value	encode	facets	source
address	0	1	Byte	0	None		slot:../|bql:flexDeviceAddress
functionCode	1	1	Byte	6	None		null
startAddress	2	2	Word	0	None	bigEndian=true	slot:../address
value	4	2	Word	0	None	bigEndian=true	slot:../writeValue/value
crc	6	2	crc				

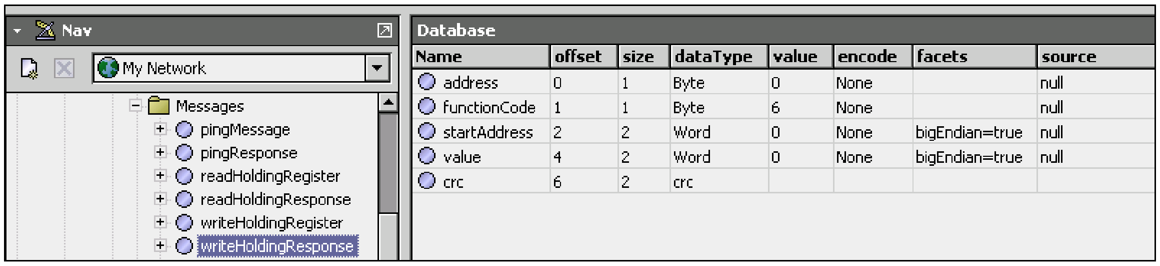
Note that the value message element is sourced from "slot:../writeValue/value". The value to be written to the Modbus register will come from the FlexProxyExt's writeValue property. The writeValue in this case is a StatusNumeric type. Specifying "writeValue/value" in the source ord will extract only the numeric value.

Define a Write Holding Register Response Message

The write holding register response message used by a proxy extension will model a Modbus RTU Preset Single Register Response message. This message has the following byte format:

device address	**address of this Modbus device**
function	06 preset single register function
register hi byte	nn
register lo byte	nn
preset data hi byte	dd
preset data lo byte	dd
crc hi byte	xx
crc lo byte	xx

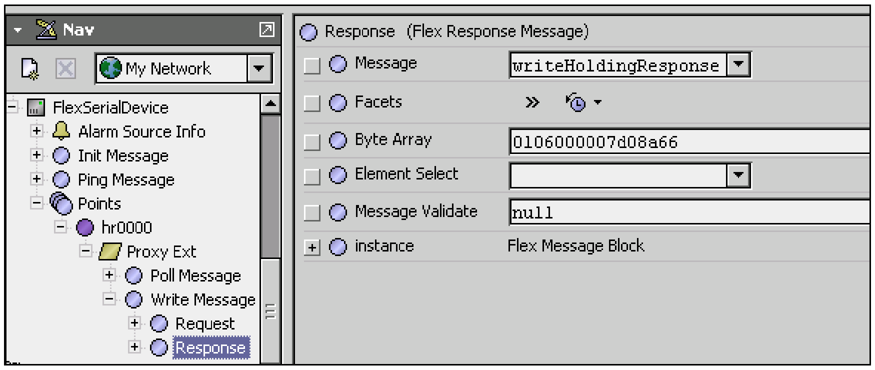
Using methods described earlier create a "writeHoldingResponse" message like this:



Name	offset	size	dataType	value	encode	facets	source
address	0	1	Byte	0	None		null
functionCode	1	1	Byte	6	None		null
startAddress	2	2	Word	0	None	bigEndian=true	null
value	4	2	Word	0	None	bigEndian=true	null
crc	6	2	crc				

Setup NumericWritable Point's WriteMessage

Open the property sheet view of the Hr0000 NumericWritable point's WriteMessage's Response component. Select "writeHoldingResponse" as the Message property value.



Response (Flex Response Message)

Message	writeHoldingResponse
Facets	
Byte Array	0106000007d08a66
Element Select	
Message Validate	null
instance	Flex Message Block

Now the "hr0000" NumericWritable point is setup to write values to the addressed Modbus register. If you invoke the "set" action on the point, you should see a Preset Single Register message being sent to the Modbus device.

Add Message Validation to Response Messages

The “messageValidate” property of the Flex Response Message is the hook that is used to provide error checking of response messages. This “messageValidate” property is an ord will point to a Program object.

The Program object must have the following slots:

1. “requestByteArray” of type Blob
2. “responseByteArray” of type Blob
3. “results” of type String

This Program object must be coded to determine if the response byte array is valid for the request byte array. If the response is valid, the “results” slot must be set to “Ok”. If the response is not valid, it must set the “results” slot to a short string that describes the error.

The FlexSerial driver will automatically set the request and response byte array values in the messageValidate Program object, invoke its execute method, and read the results value. If the results is not “Ok”, the control point’s status will be set to fault and the results string will appear in the faultCause property of the FlexProxyExt.

In the “flexModbusRtu” example station, there is an “messageValidate” Program object at: ”slot:/Drivers/FlexModbusRtu/messages/ErrorCheck”.

Modbus ASCII Protocol

Modbus ASCII protocol is a request-response Ascii protocol and message framing is defined by a “:” start character and a 0x0d, 0x0a (carriage return, line feed) frame end sequence. All data between the frame start and frame end is encoded as Ascii Hex data. Message is then defined as follows:

Start	device addr	function	data	lrc	end
‘:’	“nn”	“ff”	n ascii hex bytes	“nn”	0x0d, 0x0a

NOTE: This document does not detail the application of the FlexSerial driver to communicate to interface to a ModbusAscii device. There is an example implementation of a station, “flexModbusAscii”, that does provide an implementation that communicates to ModbusAscii devices. This example station uses a ModbusSlave driver using one serial port and the FlexSerial driver is using another serial port. A RS-232 cable is connected between the two serial ports.

FlexWeatherStationExample

This example uses two FlexSerialNetworks. One network is used to simulate a weather station device that periodically sends the current temperature, humidity, wind speed, and wind direction out the serial port. The other network then receives this message unsolicited and exposes the received values.

The format of the message is a string as that starts with a “#” followed by the temperature value, humidity value, wind speed value, and wind direction with a comma delimiter between each value. The message is terminated with a carriage return (0x0d).

Example: #72.8,78.4,12.8,230.4

Document Change Log

- Updated: June 12, 2008

  Added more details and notes about the “value” and “encode” properties for FlexMessageElement type components, as given in Table 1 starting on page 5. Related to this, a new section was added: “FlexMessageElement Encode property usage” on page 7. Added a “Notes on Building Messages” section starting on page 11. Removed “BETA DRAFT” from cover and page headers. Document page count increased from 20 to 24, and document currently remains in PDF format only.

- BETA DRAFT: May 2, 2007

  Initial draft document, available in PDF format only.