Creating/Editing Your Own Product File via XML

The "**Product File Creator**" tool found under the **"Tools" menu of WinScriptLive** can be used to get started creating a product file. This can be used for simple product files.

The following tutorial is only in reference to directly editing the resulting text product file rather than using the "Product File Creator" tool.

Product files can be created or edited on any non-document mode word processor including Notepad. Word processors such as Word, WordPad, and Word Perfect can be used as long as the files are exported to non-rich, straight ASCII text.

XML document creation tools can be very useful due to easy color-coding, error-checking and highlighting.

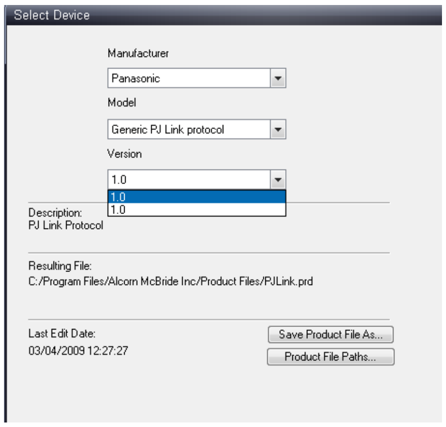
Examples in this section will show screen shots using "**XML Notepad**" with "**Shell.prd**" open. "Shell.xsd" is a schema that can be used for validation with any XML editor.

Protocol File Storage

Product files are stored **inside** the .ami script after a device has been added to the Script.

When testing, be sure to re-launch WinScriptLive, click on "**edit device**" in the devices screen, and select the new version. New protocol files are **not** automatically updated.

When selecting a version, two options will usually appear. One will have a "**Resulting File**" as "stored in .ami file". The other will have the "Resulting File" as a location on the hard drive.



Select Device

Manufacturer: Panasonic

Model: Generic PJ Link protocol

Version: 1.0

1.0
1.0

Description:
PJ Link Protocol

Resulting File:
C:/Program Files/Alcorn McBride Inc/Product Files/PJLink.prd

Last Edit Date:
03/04/2009 12:27:27

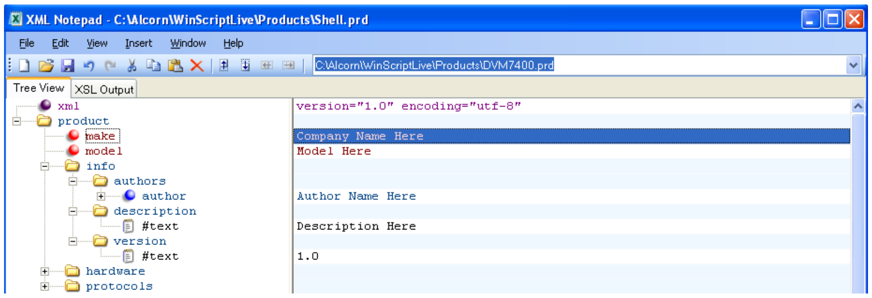
Getting Started

The easiest way to get started writing the protocol file is to go to the "Tools" menu in WinScript Live. There are several options under this menu. The "Product File Creator" Tool can be used to generate a very simple product file. The "Product File Editor" launches XML Notepad. The "Product File Tester" can help test the resulting files.

For the following section examples, open "Shell.prd" located in the "product files" directory of "Program Files\Alcorn McBride". Opening this in XML Notepad ("Product File Editor" on the "Tools" menu) is a good place to start.

Product Section

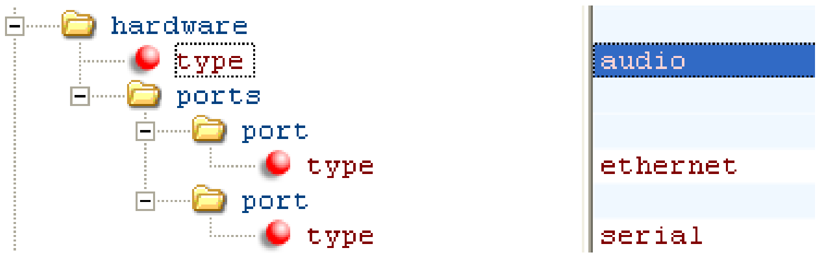
The beginning of the product section is for information and WinScript display only. This includes Make, Model, Author, Description and Version.


XML Notepad - C:\Alcorn\WinScriptLive\Products\Shell.prd
File
Edit
View
Insert
Window
Help
C:\Alcorn\WinScriptLive\Products\DVM7400.prd
Tree View
XSL Output

```
xml                     version="1.0" encoding="utf-8"
product
  make                  Company Name Here
  model                 Model Here
  info
    authors
      author            Author Name Here
    description
      #text             Description Here
    version
      #text             1.0
  hardware
  protocols
```

Hardware Section

This section defines information about the actual product’s available communication ports and type.



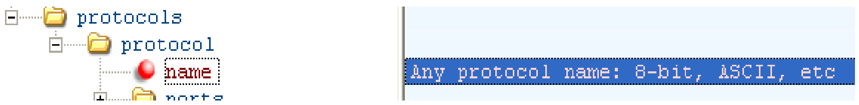
```
hardware
  type          audio
  ports
    port
      type      ethernet
    port
      type      serial
```

Here’s a brief description of what each field here is used for

- **"hardware" section’s "type"**: this is used to indicate a special type such as FlexIO or Show Controller
- **"port" sections’s "type" :** type ethernet or serial. Delete or add additional ‘ports’ sections to match your hardware.

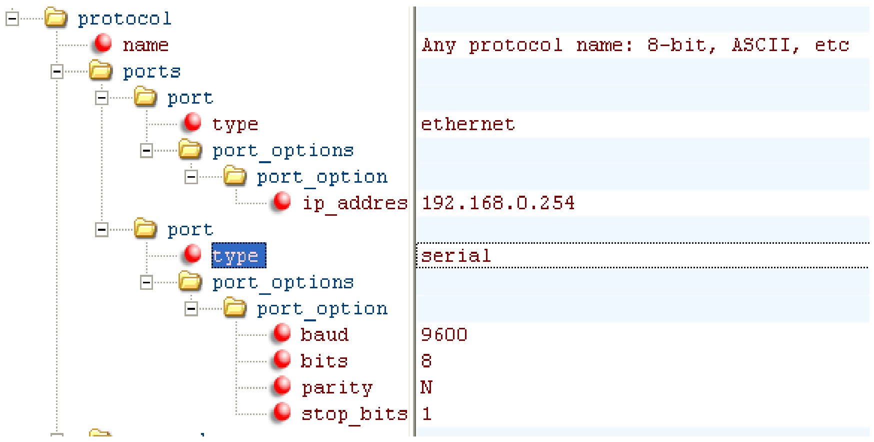
Protocol Section

Products can have multiple protocol types. For example, our DVM2 accepts both a Sony Protocol and a Pioneer Protocol. Some products have both an ASCII protocol and a Hexadecimal protocol. You can choose to implement one or more protocols by making a "protocol" section and giving it an appropriate name.



```
protocols
  protocol
    name        Any protocol name: 8-bit, ASCII, etc
    ports
```

After picking a name, define which ports use that protocol and give details about that port:

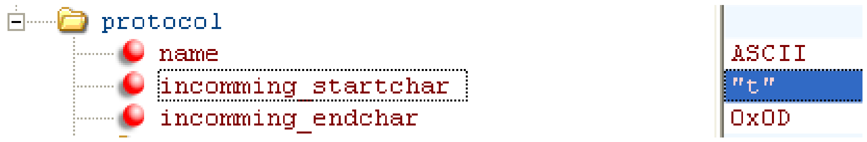


```
protocol
  name                          Any protocol name: 8-bit, ASCII, etc
  ports
    port
      type                      ethernet
      port_options
        port_option
          ip_addres             192.168.0.254
    port
      type                      serial
      port_options
        port_option
          baud                  9600
          bits                  8
          parity                N
          stop_bits             1
```

The details about each port are stored as a "port_option". If a device's port uses multiple baud rates, add a "port option" for each baud rate configuration. In the case of ethernet, the IP address and UDP port is used only as a default parameter and not as a required connection IP.

Copy and Paste additional "port" sections as needed.

Optionally, you can add an incoming start character or an incoming end character to aid in the processing of Incoming messages**.**



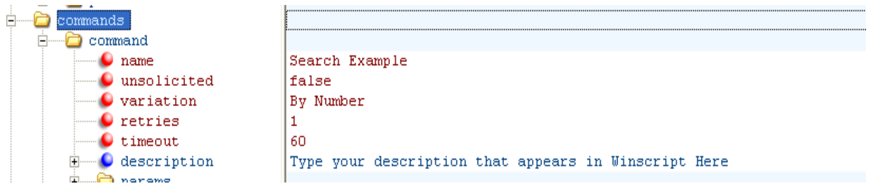
```
protocol
  name                    ASCII
  incomming_startchar     "t"
  incomming_endchar       0x0D
```

These can be specified as a single ASCII character in quotes, or any other character represented with a 0x prefix.

Commands Section

This all-important section defines the actual commands and reponses.

The "commands" section defines a list of Incoming and outgoing commands to/from the device. The image below shows an example command called "Search Example."



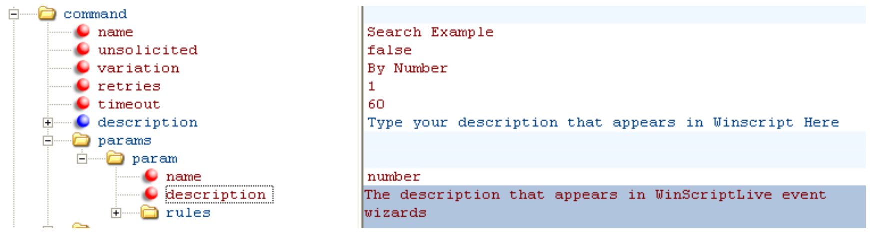
```
commands
  command
    name            Search Example
    unsolicited     false
    variation       By Number
    retries         1
    timeout         60
    description     Type your description that appears in Winscript Here
```

The basic attributes of a command are:

- **Name**: The name of the command that appears in WinScript
- **Unsolicited**: If the command is only an unsolicited, Incoming message, make this boolean value true
- **Variation**: A variation of the command - use if different parameters are needed for a command of the same name.
- **Retries:** Number of times the show controller will re-send the message after not receiving a response. Default is zero.
- **Timeout:** Number of Frames before the show controller will re-send the message after not receiving a response. Default is 30 frames.

Command Parameters

Each command will have a set of parameters. These correspond to Data1, Data2, Data3, etc, in WinScript Live. The "data" number is determined by the param's location in "params" section.

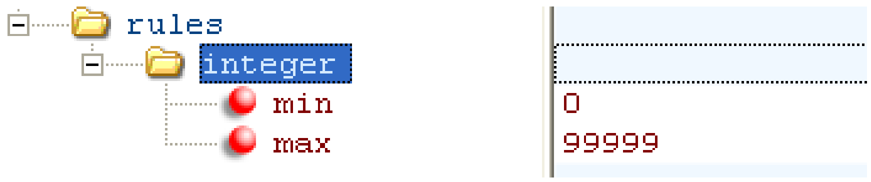


```
command
    name                 Search Example
    unsolicited          false
    variation            By Number
    retries              1
    timeout              60
    description          Type your description that appears in Winscript Here
    params
        param
            name         number
            description  The description that appears in WinScriptLive event
                         wizards
            rules
```

- **Name**: The parameter's name can be used later to have operations performed on it or to be placed into the final message
- **Description:** The description will appear in WinScript Live's event wizards to aid the user in knowing what to type into Data1, Data2, etc.

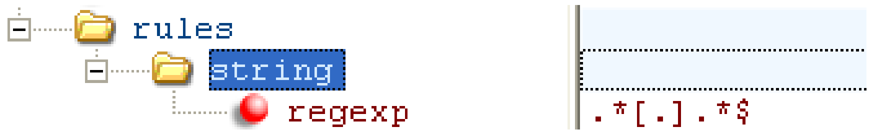
Parameter Rules

A rule restricts what is considered a valid entry in WinScript Live. Rules are not required, but do aid the user in entering the correct value when using a custom command. They are also used to help determine the output in the final outgoing message.
The most common rules are **integer** restrictions and **string** restrictions.
An example of a **integer rule** is shown below:



```
rules
    integer
        min      0
        max      99999
```

This allows a user to enter 0-99999 into a "Data" column in WinScript.
Another example is a **string rule**:



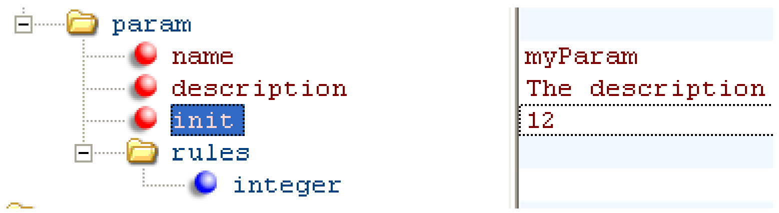
```
rules
    string
        regexp   .*[.].*$
```

This string rule has an optional parameter of "regexp". This indicates that a regular expression defines a valid entry for this string. The above string defines a valid filename.
Documentation on Regular Expressions can be found in many locations on the web. The Regex Buddy, found at http://www.regexbuddy.com/, provides a tool for learning regular expressions.
The table below lists the available "Rules" to use with custom protocol files along with their optional parameters.

Rule	Description	Parameters
Integer	*An integer value*	*Min and Max (optional)*
Decimal	*A decimal value*	*Min and Max (optional)*
String	*A string value in double quotes OR in the style h01*	*Regexp (optional)*
Bool	*True, false, yes, no, 1 or 0*	
DateTime	*Real time and/or date*	
Timecode	*In the form 00:00:00.01*	
Percent	*0-100% (sign required)*	
Option	*Match the value the user typed in exactly – No quotes required*	*Match – the string to match* *Replace (optional) – the string, timecode, int, etc to replace the entire match with*

Optional Parameters (Init Values)

Add an initial value to make a parameter optional. This way, if the parameter isn't entered by the user, the init value will be used instead.

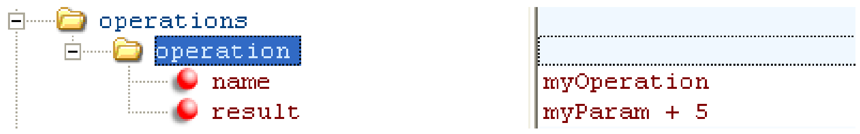


```
param
    name            myParam
    description     The description
    init            12
    rules
        integer
```

init: the value that will be used if the parameter isn't entered in WinScript.
Note: parameters with "String" rules should have "init" values placed in double quotes.
Note: parameters with "init" must be placed at the END of the list of parameters.

Operations on Parameters

If you'd like to **change the user entered data** before it goes into an outgoing message, you can perform an "operation" on a parameter.
A simple operation to add 5 to the parameter "myParam" is shown below:



```
operations
    operation
        name        myOperation
        result      myParam + 5
```

Name: The name that can be referenced in the final message

Result: The actual operation functions or operators in combination with a parameter or operation name.

Note: Any hex values can have a 0x prefix. For example, "myParam + 0x05" is valid.
Below is a list of operators that can be performed on a particular parameter or other operation.

Operator	Function
(	*Open Parenthesis*

)	*Closed Parenthesis*
!	*Logical Not (use for bool types)*
***	*Multiply*
/	*Divide*
-	*Subtract*
+	*Add*
&	*Concatinate*
>>	*Shift Right*
<<	*Shift Left*
>	*Greater Than*
<	*Less Than*

These are in order of operator priority. In other words, in the operation:

MyParam + 2 *3

The multiplication of 2*3 would occur before the addition of "MyParam".

In addition to operators, functions can also be used.

In the table below, the values in brackets, < >, can be a static value or reference another operation name or a parameter name.

Function	Parameters	Description
Mod (<dividend>, <divisor>)	*<dividend> integer value* *<divisor> integer value to divide by*	*Returns remainder from division*
Left(<string>, <length>)	*<length> how many characters from the left to retain in the string*	*Returns the first (or leftmost) character or characters in a text string*
Right(<string>, <length>)	*<length> how many characters from the right to maintain*	*Returns the last (or rightmost) character or characters in a text string*
Mid(<string>, <starting point>, <length>)	*<starting point> the zero indexed starting point in the string* *<length> the number of characters to maintain*	*Returns a specific number of characters from a text string starting at the position you specify*
Length(<string>)		*Returns the number of characters in a text string*
Pad(<byte>, <length>)	*<byte> integer or hex byte to duplicate to create string* *<length> total length that the string should be.*	*Returns a string of characters equal to the length*
bitand(<parameter>, <parameter>, ….,<parameter>)	*<parameter>a parameter name or a number written as an integer or hex value written as 0x01 where 01 is the hex value of 1*	*Returns the bitwise "and" of all the parameters*
bitor(<parameter>, <parameter>, ….,<parameter>)	*Same as above*	*Returns the bitwise "or" of all the parameters*

bitxor(<parameter>, <parameter>, ….,<parameter>)	*Same as above*	*Returns the bitwise "xor" of all the parameters*
not(<parameter>)	*<parameter> logical value*	*Returns the reverse of a logical argument. For a boolean value, false returns true.*
compl(<parameter>)	*<parameter>a parameter name or a number written as an integer or hex value written as 0x01 where 01 is the hex value of 1*	*Returns the bitwise complement of a value*
atoi(<string>)	*<string> an ASCII string such as "1234"*	*Converts ASCII to an integer. Returns the actual number 1,234 (ready for mathematical operations)*
btoi(<string>,<number of bytes>)	*Convert a "Big Endian" byte order string (high-order byte comes first) string into an integer value*	*Returns the corresponding integer (up to 4 bytes long)*
ltoi(<string>,<number of bytes>)	*Convert a "Little Endian" byte order string(high-order byte comes first) string into an integer value*	*Returns the corresponding integer (up to 4 bytes long)*
htoi(<string>)	*<string> an ASCII hex string such as "0x12"*	*Converts ASCII to an integer. Returns the actual number 12 (ready for mathematical operations)*
ByteChecksum(<parameter>)	*<parameter> a string of characters, ASCII or other*	*Returns the LSB of the byte checksum for all the characters in the parameter*
Checksum(<parameter>)	*Same as above*	*Returns a bytewise checksum (result up to 4 bytes) for all the characters in the parameter*
AMINetChecksum(<parameter>)	*Same as above*	*Returns an AMI Net checksum for all the characters in the parameter*
MD5(<string>)	*String of any length*	*Returns the 32 byte MD5 algorithm result*
BCC(<string>)	*String of any length*	*Returns the byte BCC checksum (the xor of all the bytes)*
LSB(<integer>)	*(<integer>) A four byte integer value or variable*	*Returns the least significant byte in the 4 byte integer value.*
MSB(<integer>)	*Same as above*	*Returns the most significant byte in the 4 byte integer value.*
GetByte(<integer>, <index>)	*(<integer>) A four byte integer value or variable* *(<index>) The index of the byte we want. 1=lsb, 4=msb*	*Returns the byte referenced byte the index from the 4 byte integer value*
Byte(<integer>)	*Same as LSB*	*Same as LSB*
Word(<integer>)	*(<integer>) A four byte integer value or variable*	*Returns the 2 Least significant bytes from the 4 byte integer value*
sprintf(<format>, <param>, <param>, …. <param>)	*The format string (using %s, %f, %d, or %p)*	*Returns a string with the parameters stuffed in as needed*
Hours(<timecode>)	*<timecode> a string timecode or timecode parameter in the format 00:00:00.01*	*Returns an integer value of the hours portion of the string*
Minutes(<timecode>)	*Same as above*	*Returns an integer value of the minutes portion of the string*
Seconds(<timecode>)	*Same as above*	*Returns an integer value of the seconds portion of the string*

Frames(<timecode>)	*Same as above*	*Returns an integer value of the frames portion of the string (not total frames)*
TotalFrames(<timecode>)	*Same as above*	*Returns the total number of frames equivalent to the timecode*
BER(<oid string>)	*An Object ID string in the form: "1.3.6.1.2.3.5.4.45.5"*	*Returns he binary string needed for outgoing messages such as SNMP*
Replace(<string>,<find>,<replace>)	*<string> to find in* *<find> value to find* *<replace> value to stuff in*	*A string with the values replaced accordingly. Use spaces between commas when using this function:* *ie: Replace("test", "st","ll")* *NOT* *Replace("test","st","ll")*
ScanList(<value>,<length>, *<list variable>, <size>, <endian>)*	*<value> to parse and store* *<length> total length to store* *<list variable> destination list (array) variable* *<size> number of bytes per array slot* *<endian> either "big" or "little" in quotes. Big is highest value number first.*	*Takes a stream of data (ie: 00239102) and parses words (or bytes) of data into a list (array) WinScriptLive variable.* *ie: for ScanArray(00239102, 4, myList, 2,"little") the result would be:* *myList[0]=00, myList[1]=23, myList[2] =91, myList[3]=02*
PrintList(<length>,<list variable>, *<size>, <endian>)*	*<length> total length to store* *<list variable> destination list (array) variable* *<size> number of bytes per array slot* *<endian> either "big" or "little" in quotes. Big is highest value number first*	*Outputs a stream of data (not ASCII) using a list variable.*
GetAt(<list variable>, <index>)	*<list variable> destination list (array) variable* *<index> location in the list variable*	*Gets the value of a particular position in a list (array) variable.*
SetAt(<list variable>, <index>, <value>	*<list variable> destination list (array) variable* *<index> location in the list variable* *<value> value to store*	*Sets the value at a particular position in a list (array) variable.*
If(<logical test>,<value if true>, <value if false>)	*<logical test> test statement such as myVar = "1"* *<value if true> resulting value to use if above logical test is true* *<value if false> resulting value to use if above logical test is true*	*Returns one of two values based on logical test.* *Note: If statements can be "nested." ie:* *if (myVar=1, "varIsOne", if(myVar=2, "varIsTwo", "unknown"),*

Note: the Printf function uses a method similar to the "C-style" printf.

The table of characters that can be used are shown below:

Letter	Function
%s	*String print*
%d	*Integer ASCII Print*
%f	*Decimal ASCII Print*
%x	*Hex ASCII Print*
%p	*Value Print (non-ASCII values)*

Precision is specified by using a number before the "d" or "f" to indicate how many characters will be printed. If the value to be printed is shorter than this number, the result is padded. The value is not truncated even if the result is larger. Placing a "0" before d or f indicates leading zeros.

For example:

printf("Color %s, number1 %d, number2 %05d, hex %X, float %5.2f", "red", 123456, 89, 255, 3.14);

will create following line:

Color red, number1 123456, number2 00089, hex FF, float 3.14

The "%p" is a special type created by alcorn to send out individual bytes of data without any formatting.

For example, to type in h22,h23,h00,h04 into a data field and send out exactly the corresponding characters with no formmating, use:

printf("%p", paramName)

When using %p, leading placing a "0" before the number indicates added nulls

Note: %p is little endian byte order. In other words, using printf("%4p", h03) will result in

h03 h00 h00 h00

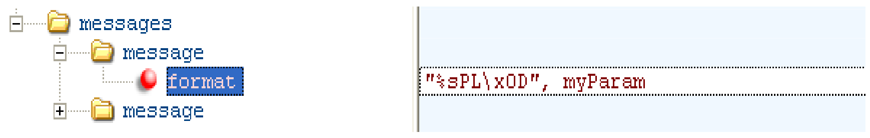
%p will also automatically truncate to the lowest size of the byte. It WILL also truncate the most significant bytes of a value if a number is specified.

For example: printf("%1p", 259) would result in just h03 . But printf("%2p", 259) would result in h03 h01.

Outgoing Message

The outgoing message is what is send out of the V16’s serial, ethernet or MIDI port to the remote device.

Outgoing messages are formulated in way similar to a "C style" printf or sprintf statement.



```
"%sPL\x0D", myParam
```

Format: The "printf" style statement that defines the outgoing message. (see table in the Functions section for details)
Here’s a breakdown of the format: "%sPL\x0d", myParam
The **%s** defines the spot that the parameter, "myParam" will be inserted. The "s" means that "myParam" is a string and will be inserted as a string.

The **PL** is simply the characters ‘P’ and ‘L’.
The **\x0D** is the hex character 0D, more commonly known as a carriage return.

Hex Characters

Hexidecimal characters are represented in quotes with a \x preceeding them.

ie: "\xFF"

Inserting Parameters/Operations

Insert parameters by using the % sign followed by the character that matches both the parameter’s type and the form in which you’d like the outgoing message to be created.

The below table lists the characters you may use:

Operator	Function
%s	*String input, string (ASCII) output*
%d	*Integer input, string (ASCII) output*
%p	*Integer or string input, acutal character value output (Use when hex output is requried)*
%f	*Decimal input, string (ASCII) output*

Escape Characters and Quotes

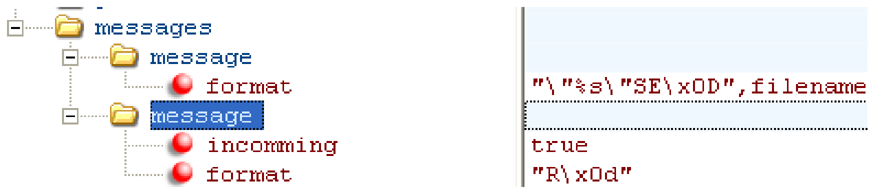
To escape any character, such as a % sign or a quotation mark, use the backslash character ‘\’.

For example, to actually send "hello world", including the quotes, write:

"\"hello world\""

Incoming Message (Response To Command)

You can write both the correct response to a message and responses that are considered to be error responses to a command. These messages are included under the same "command" section as the outgoing message.
To indicate that a message is Incoming, set the message’s parameter "incoming" to "true" as shown below



```
messages
    message
        format          "\"%s\"SE\xOD",filename
    message
        incomming       true
        format          "R\xOd"
```

Incoming: indicates that this is an Incoming message. If this parameter is not present, outgoing is assumed.

Format: **Using this param indicates that you are using the printf style**

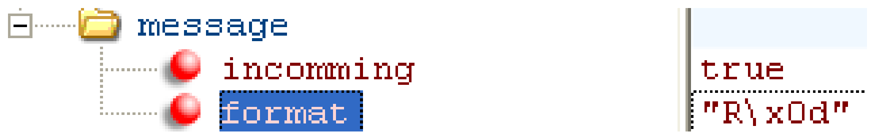
Regexp_format**: using this param indicates that you are using the regular expression style**
An Incoming message’s format can be written in one of two ways: "printf style" or "regular expressions" style.

Printf style

The printf style is a "c" style formatting of a message which can include data parameters that a user has typed into WinScript. This format is identical to the style shown in more detail in the "Outgoing Messages" section.

Important Note: if using this style for Incoming messages, use either the "Incoming char" or "outgoing char" in the "protocol" section to indicate the start or end of a message.

The example below shows an Incoming message of the character ‘R’ followed by a carriage return.

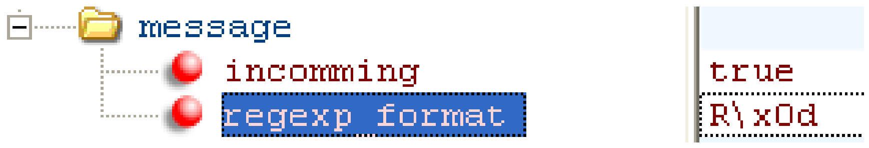


```
message
    incomming    true
    format       "R\x0d"
```

Regular Expression Style

Perl compatible regular expressions can also be used to define an Incoming message. Documentation on Regular Expressions can be found in many locations on the web. The Regex Buddy, found at http://www.regexbuddy.com/, provides a tool for learning regular expressions.

Note that regular expressions do NOT require quotes around them like the printf style format does.



```
message
    incomming       true
    regexp format   R\x0d
```

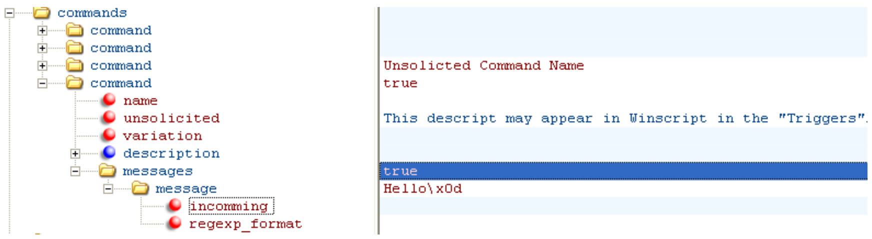
A brief table of regular expressions is shown below:

Operator Type	Example	Description
Literal Characters *Match a character exactly*	*a A y 6 % @*	*Letters, digits and many special characters match exactly*
	\$ \^ \+ \\ \?	*Precede other special characters with a \ to cancel their regex special meaning*
	\n \t \r	*Literal new line, tab, return*
	\cJ \cG	*Control Codes*
	\xa3	*Hex codes for any character*
Anchors and assertions	^	*Starts With*
	$	*Ends With*
	\b \B	*on a word boundary,*
		NOT on a word boundary
Character groups *any 1 character from the group*	*[aAeEiou]*	*any character listed from [to]*
	[^aAeEiou]	*any character except aAeEio or u*
	[a-fA-F0-9]	*any hex character (0 to 9 or a to f)*
	.	*any character at all*
Counts *Applies to previous element*	+	*1 or more ("some")*
	*	*0 or more ("perhaps some")*
	?	*0 or 1 ("perhaps a")*
	{4}	*exactly 4*
	{4,}	*4 or more*
	{4,8}	*between 4 and 8*
Alternation	|	*either, or*

Grouping	*( )*	*Group for saving to variable* *(Maximum of 6 sets of parenthesis per expression)*

Incoming Message (Unsolicited Command)

An Incoming message that is NOT sent in response to an outgoing message is considered to be "Unsolicited".
These Incoming messges can be used to trigger a Sequence in WinScript or to store the contents in a variable (See Incoming Message Variable Storage for more detail)
A **separate command** must be created for an unsolicted message. The image below shows an unsolicited message "Hello" followed by a carriage return.

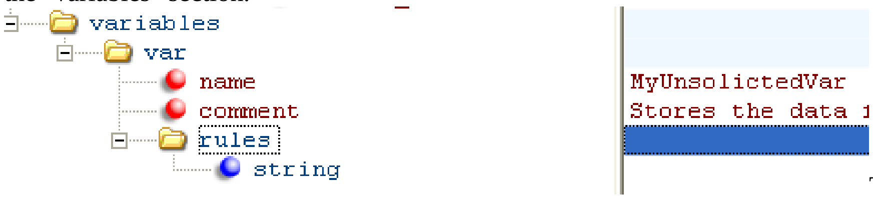


```
commands
  command
  command
  command
  command
    name              Unsolicted Command Name
    unsolicited       true
    variation
    description       This descript may appear in Winscript in the "Triggers".
    messages
      message         true
        incomming     Hello\x0d
        regexp_format
```

Unsolicited: Set this parameter to "true" to indicate that this is an unsolicited message.

Incoming Message Variable Storage

A response (Incoming message) can also be stored into a device variable which can be accessed in a Script.
First, setup the device variable in the "variables" section. Multiple "Var" sections can be added to the "variables" section.



```
variables
  var
    name      MyUnsolictedVar
    comment   Stores the data i
    rules
      string
```

The "var" is setup in a similar way to a "parameter".
Name: The name that will appear in WinScript and will be used to reference this variable in the protocol file
Init: An initial value for the variable (optional)
Setup: If “True” specifies that this will appear in the device setup screen

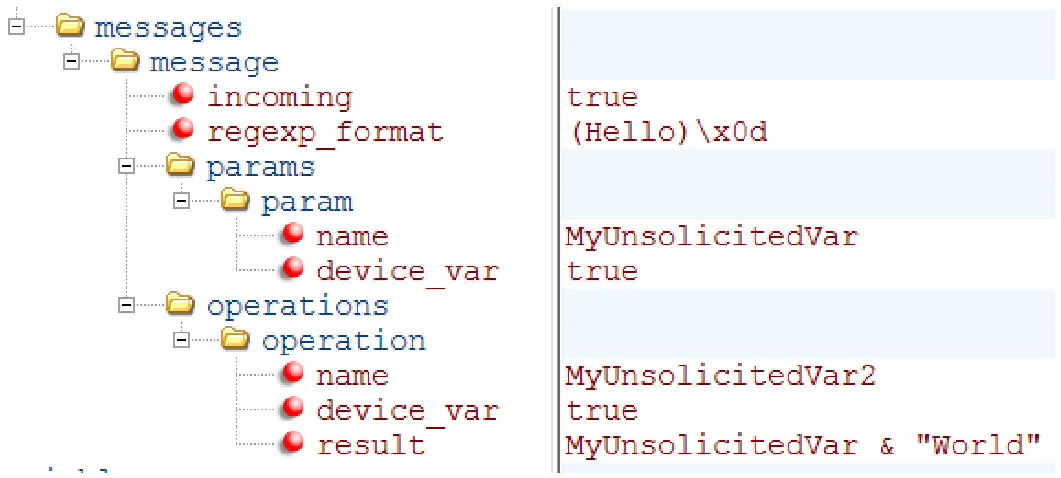
Comment: The comment that will appear in WinScript.

Rules: The "Rules" are identical to the "Rules" for a "Parameter". See "Parameter Rules" section above for more details.
Next, specify what portion of the Incoming message you would like to store and in what variable.

The section of the message to store can be specified by placing parenthesis around that portion. These sections (starting at the most outward section) are stored in subsequent "param" sections listed below the message.

A **maximum of 6 sections** can be pulled from a single message using parenthesis. However, an unlimited number of variables can be "stuffed" by using operations on these 6.

The below example shows an unsolicited message of "Hello" followed by a carriage return. This message shows storage in two different device variables.



messages	
message	
incoming	true
regexp_format	(Hello)\x0d
params	
param	
name	MyUnsolicitedVar
device_var	true
operations	
operation	
name	MyUnsolicitedVar2
device_var	true
result	MyUnsolicitedVar & "World"

device_var: When set to "true", this indicates that the value in "name" is actually referencing an existing Device Variable.
For the first "param" named "MyUnsolicitedVar", only the word "Hello" is stored because that is what is in the parentheses in "regexp_format".

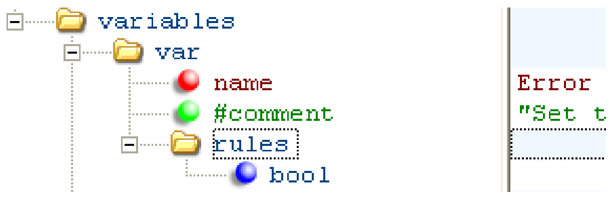
An operation is done upon the "param" named "MyUnsolicitedVar".

This operation concatenates the word: "World" with what was stored in the "param" called "MyUnsolicitedVar". This results in the string "Hello World" being stored in the variable "MyUnsolicitedVar2".

Error Variable

In addition to creating unlimited device variables in the protocols "variables" section, you can create a special, "error" variable that will be set when a device fails to receive a valid response.

Simply setting the variable name to "error" and the type to "Boolean" creates this variable as shown below



variables	
var	
name	Error
#comment	"Set t
rules	
bool	

A comment may also be set if desired.

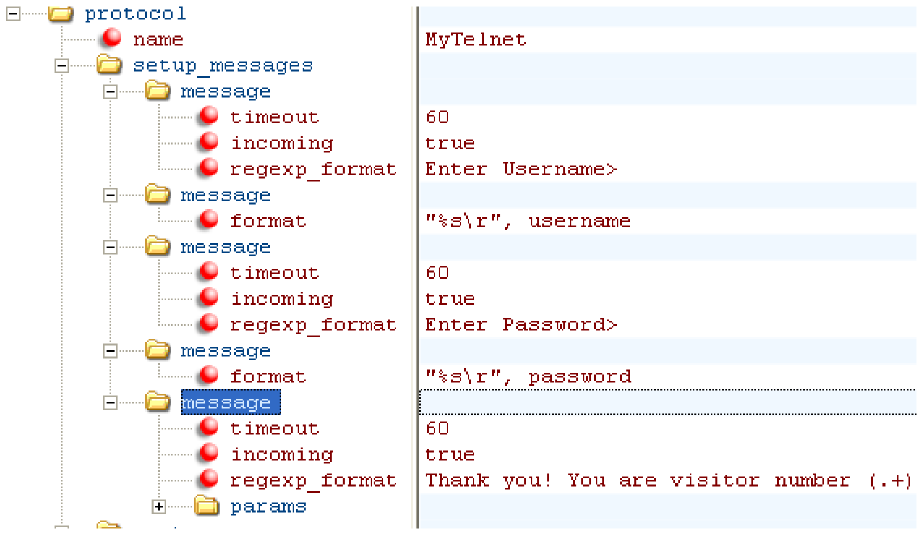
TCP Status Variable

In addition to the "Error" special device variable, you can also create a special "TCPStatus" variable that will be set when the state of the TCP connection for that device changes. Simply add the "TCPStatus" variable to your protocol file, and it will fill automatically.

Setup Messages (TCP Only)

Some TCP protocols require a series of "login" message that must take place once before the controller can send commands to the device. These setup messages take place immediately after a TCP connection has been made. Telnet, for example, often specifies a username and password login process.

The example below shows a series of login messages followed by a visitor number response. You can find this example in the "shell.prd" file under the second "Protocol" tag.



```
protocol
  name                    MyTelnet
  setup_messages
    message
      timeout             60
      incoming            true
      regexp_format       Enter Username>
    message
      format              "%s\r", username
    message
      timeout             60
      incoming            true
      regexp_format       Enter Password>
    message
      format              "%s\r", password
    message
      timeout             60
      incoming            true
      regexp_format       Thank you! You are visitor number (.+)
      params
```

The prompt for the above example would look like this:

Username> myname

Password> password

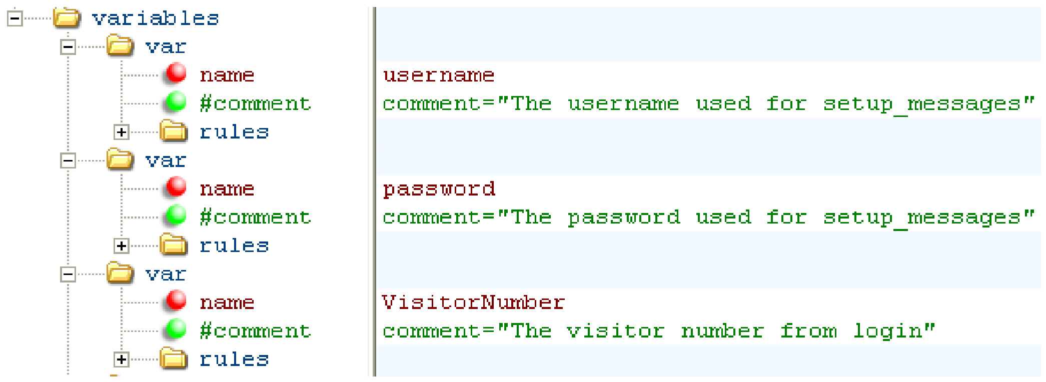
Thank you! You are visitor number 243!

Each "message" above specifies either an incoming or outgoing message. For incoming messages, "timeout" and "regexp_format" are valid parameters. For outgoing messages, "format" is the only valid parameter. Operations (such as +, -, left(), right(), etc) may be included in the "format" field directly.

In the above example, to specify sending the outgoing message of the username, use a format function of:

"%s\r", username

The value of "username" is pulled from device variables section as shown below.



```
variables
  var
    name        username
    #comment    comment="The username used for setup_messages"
    rules
  var
    name        password
    #comment    comment="The password used for setup_messages"
    rules
  var
    name        VisitorNumber
    #comment    comment="The visitor number from login"
    rules
```

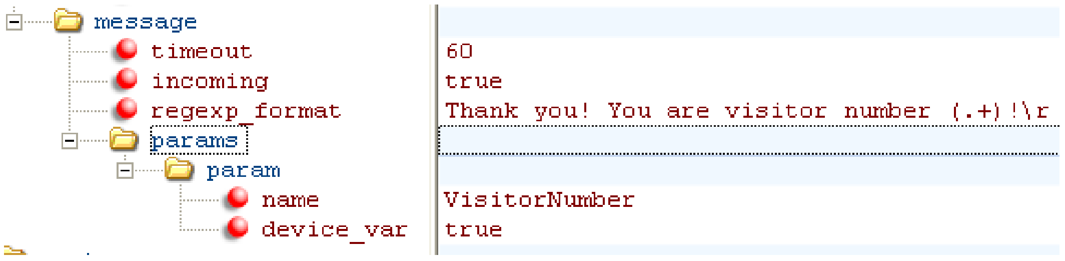
The user can modify these device variables in the WinScript Live script.

If you would like to pull some of the prompt information, basically an "incoming" message, use a regular expression just as you would any other incoming message in the product file.

In this case, the device variable "Visitor Number" (as seen above) can be filled using the incoming regexp_format:

"Thank you! You are visitor number (.+)!"

The contents of the parenthesis, .+, means any character for any length. The "params" tag below the regexp_format message indicates where the contents of the parenthesis, in this case the visitor number, should go.



```
message
  timeout          60
  incoming         true
  regexp_format    Thank you! You are visitor number (.+)!\r
  params
    param
      name         VisitorNumber
      device_var   true
```

Any number of device variables can be "pulled out" of any message within the "setup_messages" section.

This sequence of setup_messages will occur ever time a new TCP connection is made. If the incoming messages received are not as expected, the TCP connection is closed and the device’s error Boolean Type Variable is set. If the command attempting to be sent has retries, the TCP connection will try to connect again and re-send any setup_messages data. The error Boolean Type Variable will only be set after the final command retry

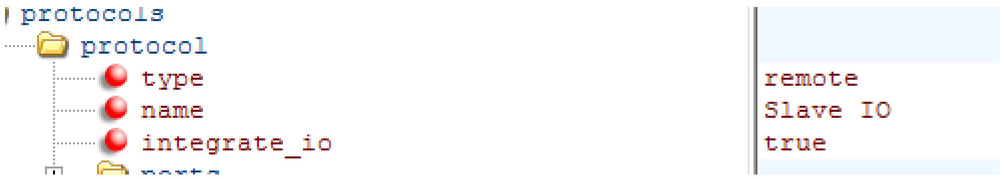
Integrated IO Protocols

Integrated IO protocols are those where IO appears "automatically" in the Inputs and Outputs windows. You can use the V16Pro's "on" or "off" commands to control these outputs, rather than using the "device" column with your custom device name. The inputs are automatically "polled" at a certain rate, and therefore do not require extra sequences to request the status of the inputs.

For all of the screen shots below, the Advantech_Adam6060.prd protocol will be used.

Setting Protocol as Integrate IO Type

In order for WinScriptLive and the V16Pro to attempt to process this file for special command names, the attribute "inegrate_io" must be set to "true" in the "protocol" section of the product file. The name of the protocol can be anything you'd like.

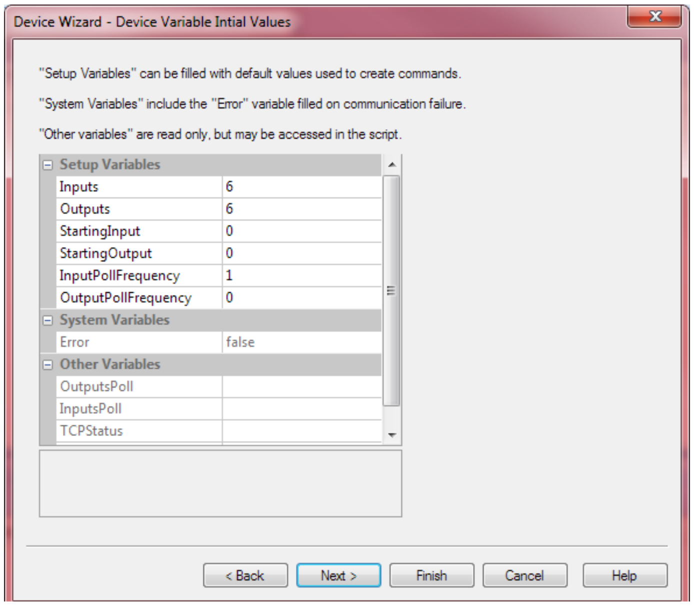


```
protocols
  protocol
    type            remote
    name            Slave IO
    integrate_io    true
```

Figure 1 - integrate IO attribute

Special Device Variables

In order for the Inputs and Outputs to appear automatically in the "Inputs" and "Outputs" windows, you need to define variables that allow the user to enter the number of inputs and number of outputs in the device wizard screen. (shown below).



Device Wizard - Device Variable Intial Values

"Setup Variables" can be filled with default values used to create commands.

"System Variables" include the "Error" variable filled on communication failure.

"Other variables" are read only, but may be accessed in the script.

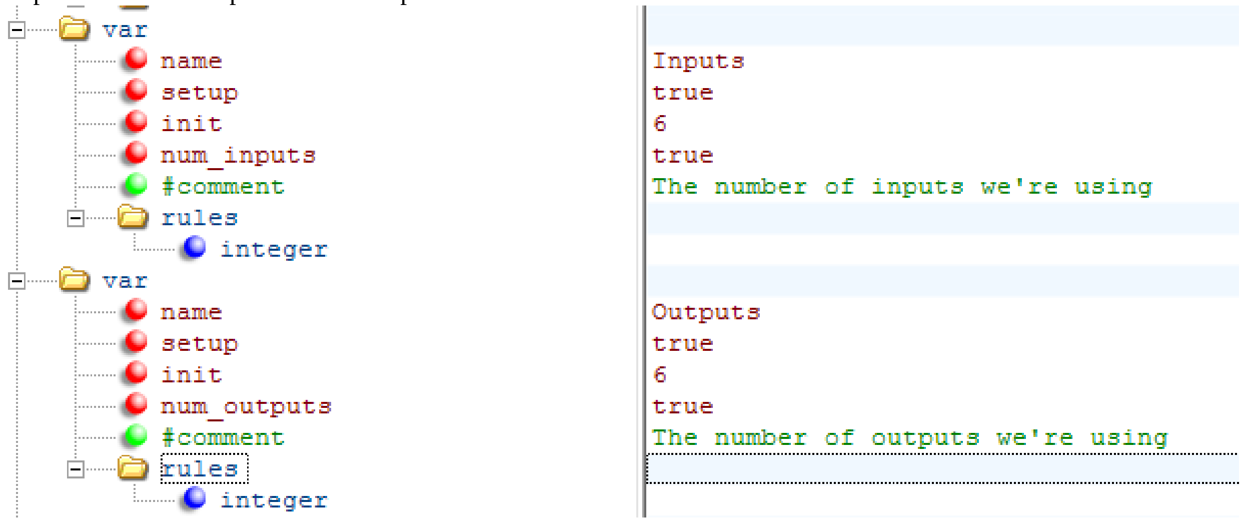
Setup Variables	
Inputs	6
Outputs	6
StartingInput	0
StartingOutput	0
InputPollFrequency	1
OutputPollFrequency	0
System Variables	
Error	false
Other Variables	
OutputsPoll	
InputsPoll	
TCPStatus	

Figure 2 - Device Wizard Setup Variables

Other variables include those that allow the user to control the frequency of the polling. As long as the "setup" attribute is present, these variables will appear in the device wizard screen.

Number of Inputs and Outputs

These variables have the special attribute "num_inputs" or "num_outputs." The screen shot below shows the XML setup from XML Notepad with these special variables.

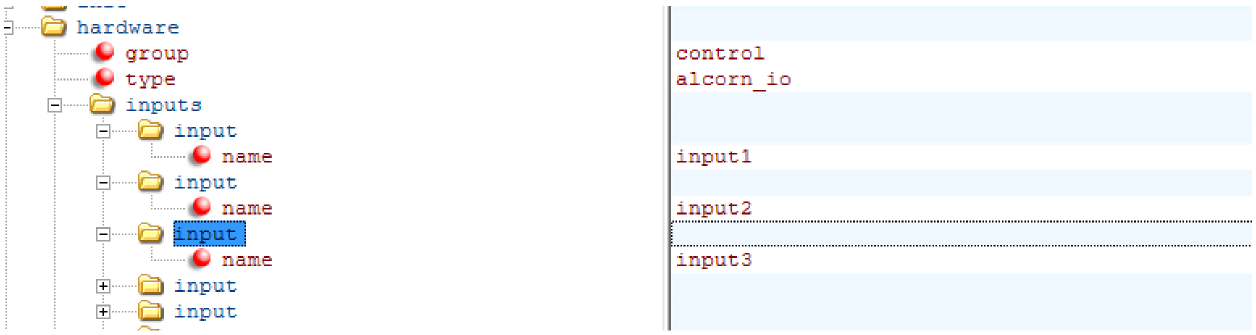


```
var
    name            Inputs
    setup           true
    init            6
    num_inputs      true
    #comment        The number of inputs we're using
    rules
        integer
var
    name            Outputs
    setup           true
    init            6
    num_outputs     true
    #comment        The number of outputs we're using
    rules
        integer
```

Figure 3- speical num_outputs and num_inputs attribute

Alternative Method: Number of Inputs and Outputs - NOT using num_outputs

If you would rather specify a specific number of inputs/outputs that is not able to be changed by the user, you can do so in the "Hardware" section of the product file. This is the way the AMI_io64.prd file is setup.

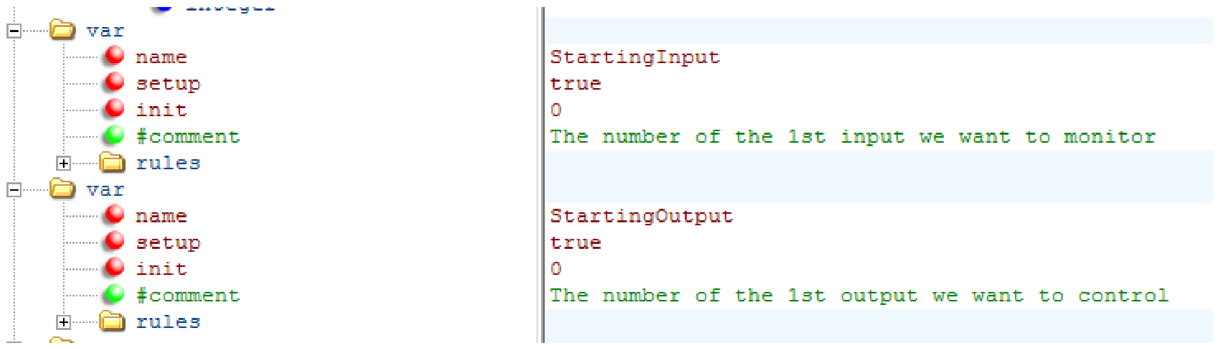


```
hardware
    group           control
    type            alcorn_io
    inputs
        input
            name    input1
        input
            name    input2
        input
            name    input3
        input
        input
```

Figure 4- Setting fixed number of inputs in AMI_IO64

Starting Input and Starting Output Location

In addition to these device variables, it is often required to have a "starting output" of the 1st output we'd like to control. This is useful in the case where you only want the V16Pro to "take control" of only half of a block of outputs. Once you define the outputs that the V16Pro will control, it will always attempt to "set" the outputs to the desired state according to WinScriptLive. In other words, it will not allow other devices to control the same set of outputs. If you require a certain starting output, these will be "normal" device variables (as shown below).

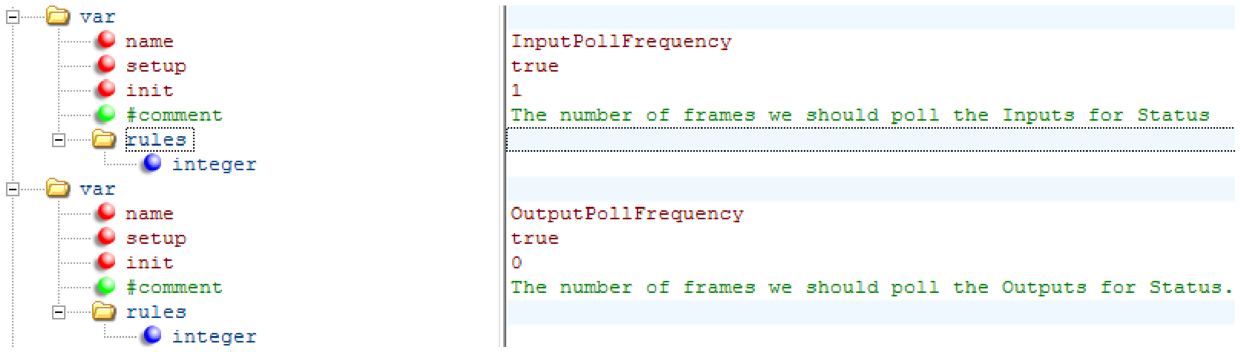


```
var
    name          StartingInput
    setup         true
    init          0
    #comment      The number of the 1st input we want to monitor
    rules
var
    name          StartingOutput
    setup         true
    init          0
    #comment      The number of the 1st output we want to control
    rules
```

Figure 5 - Standard Device variables for Starting Output Number

Polling Frequency

The frequency by which the inputs are polled can hard coded as a fixed variable in the actual event settings; however , it is nice to allow the user to be able to control this value. This can be accomplished using a standard device variable. (As shown below).



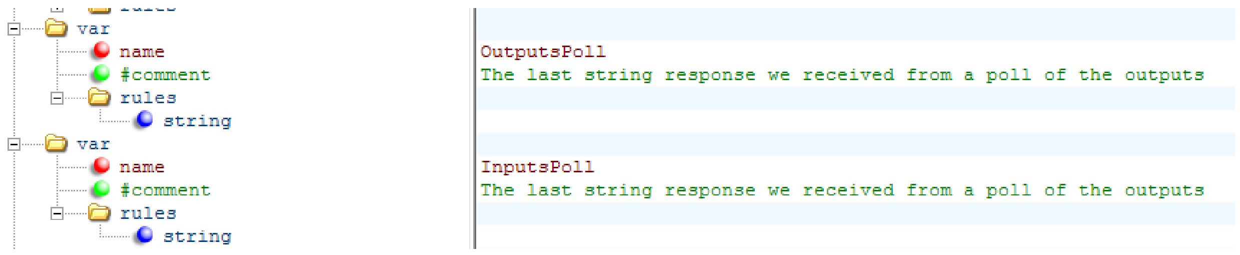
```
var
    name          InputPollFrequency
    setup         true
    init          1
    #comment      The number of frames we should poll the Inputs for Status
    rules
        integer
var
    name          OutputPollFrequency
    setup         true
    init          0
    #comment      The number of frames we should poll the Outputs for Status.
    rules
        integer
```

Figure 6 - Polling Frequency Variables

If a "0" is placed as the poll frequency value for the inputs by the user in the WinScriptLive device wizard, the inputs will never be automatically polled. If a "0" is placed as the poll frequency for the outputs, the outputs will only be polled immediately following a change of state ("set" command that changes the output value).

Temporary "Storage" Variables

In order for any information to be set for the input or output values, the V16Pro needs to believe it is storing the information for a variable. So, temporary variables are created to use to store the responses we get from the device. These are usually just strings, but can be very handy for debug purposes of your product file. These can be any names you wish and have no special properties.



```
var
    name          OutputsPoll
    #comment      The last string response we received from a poll of the outputs
    rules
        string
var
    name          InputsPoll
    #comment      The last string response we received from a poll of the outputs
    rules
        string
```

Figure 7 - Temporary device response variables

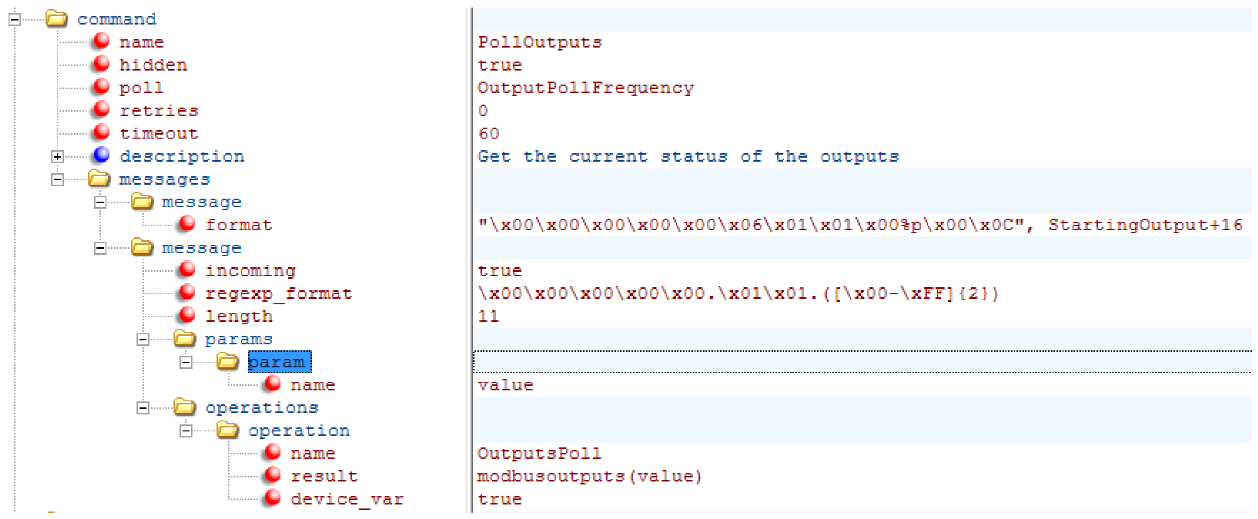
The Output's "On" and "Off" commands

In order for the V16Pro to change the state of the output using WinScriptLive's default "On" and "Off" commands, "special" commands must be in place. At a minimum, these commands are "PollOutputs" and "SetAll." Alternatively, if a "SetAll" command is not available, you can define "On" and "Off" commands for the individual outputs. However, this method is less efficient and not recommended.

Note: Unlike the device variables above, these commands must have EXACTLY the names "PollOutputs" and "SetAll"

PollOutputs Command - ModbusOutputs

Aside from the special name of "PollOutputs", this command also has the special property "poll." The value for this attribute is the number of frames that the output's state can be polled. If this value is zero, the output's state will only be requested when after a change of state command (ie: "on" or "off" command). This value can also be a variable. In this example, we'll use the variable we created called "OutputPollFrequency."



```
command
    name                    PollOutputs
    hidden                  true
    poll                    OutputPollFrequency
    retries                 0
    timeout                 60
    description             Get the current status of the outputs
    messages
        message
            format          "\x00\x00\x00\x00\x00\x06\x01\x01\x00%p\x00\x0C", StartingOutput+16
        message
            incoming        true
            regexp_format   \x00\x00\x00\x00\x00.\x01\x01.([\x00-\xFF]{2})
            length          11
            params
                param
                    name    value
            operations
                operation
                    name        OutputsPoll
                    result      modbusoutputs(value)
                    device_var  true
```

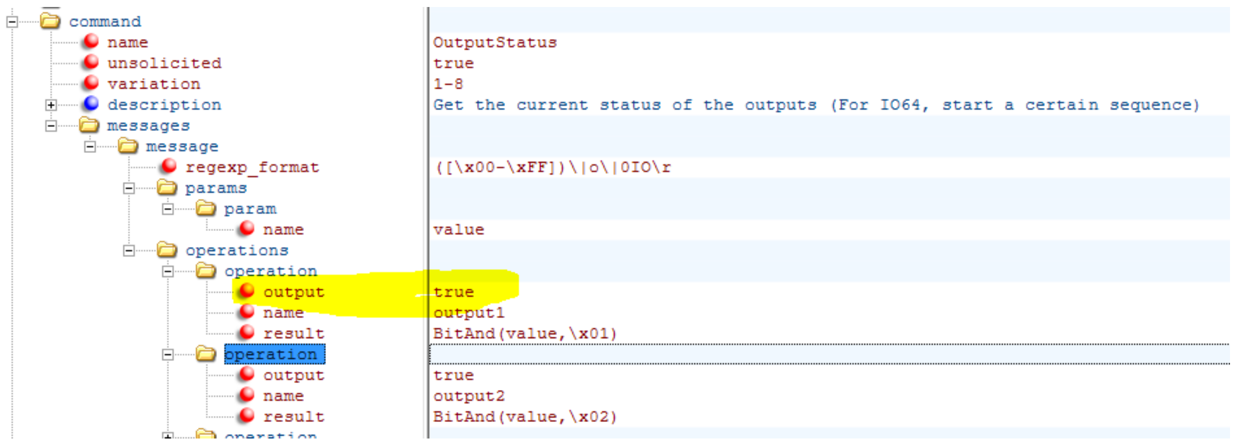
Figure 8 - Poll Outputs with "modbus" style return value

A portion of the incoming status message (noted by the parenthesis) is stored in the temporary variable "value."

From there, the message is parsed into the status of the outputs using the function "modbusoutputs." This function assumes that there are bits for each output in the variable "value." It separates these bits and stores the "on" or "off" state sequentially into the each "output" in WinScriptLive.

Note: the device variable "OutputsPoll" is used so that the V16Pro will process the response. Without this "device_var" tag and variable, the function "modbusoutputs()" would never be called.

Alternatively, any math operation (such as BitAnd()) may be used instead of the "modbusoutputs()". To finally store the value in an individual output, the parameter can have the attribute "output" to note that this is the value that should be stored in the WinScriptLive output. This method is used in the AMI_io64 product file (section shown below). In the IO64 product file, the responses are separate 'unsolicited messages' rather than a direct response to the "PollOutputs" command.



```
command
    name                    OutputStatus
    unsolicited             true
    variation               1-8
    description             Get the current status of the outputs (For IO64, start a certain sequence)
    messages
        message
            regexp_format   ([\x00-\xFF])\|o\|0IO\r
            params
                param
                    name    value
            operations
                operation
                    output  true
                    name    output1
                    result  BitAnd(value,\x01)
                operation
                    output  true
                    name    output2
                    result  BitAnd(value,\x02)
                operation
```

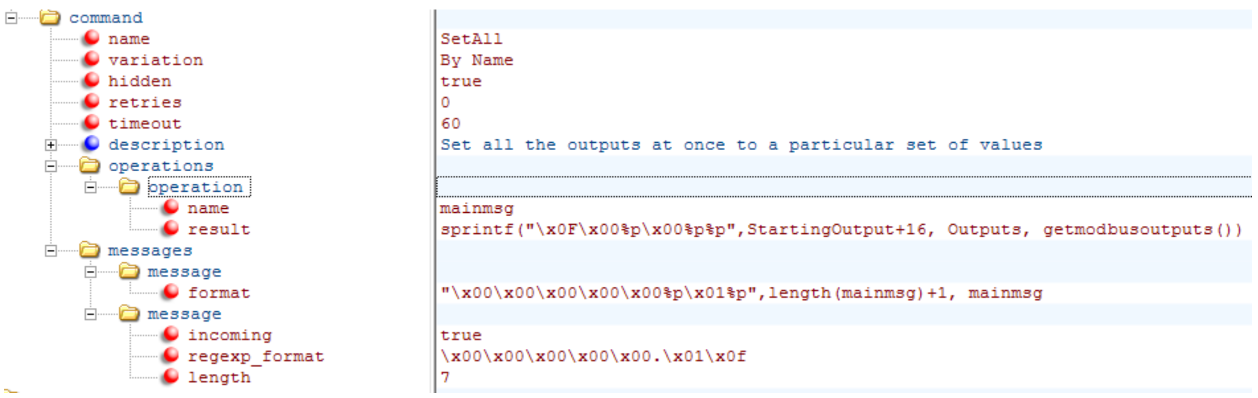
Figure 9 - IO64 Output Status Store Method

SetAll Command for Output Set

When the internal "on" or "off" commands are used, the controller keeps track of the changes to the outputs until the sequences are all processed. Then, if any changes have taken place in any of the output states, the event "SetAll" is called. Since the "SetAll" command needs no other data than the state of all the outputs, it can be sent without a "data" parameter.

The SetAll command may need to perform some additional operations in order to print the final string (as in this example), but the important function is the "**getmodbusoutputs()**" function. This generates bytes of data where each individual bit represents the state of the output. The lowest bit of the first byte sent out represents "output1."

Alternatively, use "**getoutputs**(<starting output>, <number of outputs>)" function to get a single byte for each output. For Example, use getoutputs(0, 4) generates 0x01 0x00 0x01 0x00 when the first 4 outputs are "on, off, on, off".

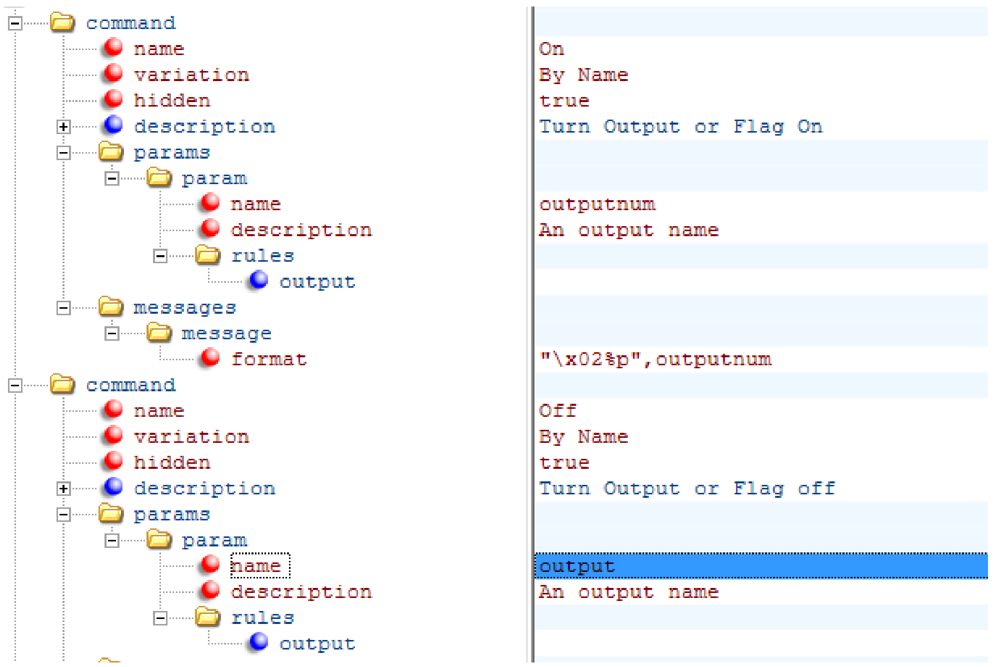


```
command
    name                SetAll
    variation           By Name
    hidden              true
    retries             0
    timeout             60
    description         Set all the outputs at once to a particular set of values
    operations
        operation
            name        mainmsg
            result      sprintf("\x0F\x00%p\x00%p%p",StartingOutput+16, Outputs, getmodbusoutputs())
    messages
        message
            format      "\x00\x00\x00\x00\x00%p\x01%p",length(mainmsg)+1, mainmsg
        message
            incoming        true
            regexp_format   \x00\x00\x00\x00\x00.\x01\x0f
            length          7
```

Figure 10 - SetAll command

Alternate Method: Individual Commands for Output Set

You can directly use the commands with the exact name "On" or "Off" to set individual outputs one at a time. This method requires that the outputs are listed by under the "hardware" section (see Alternative Method: Number of Inputs and Outputs). This method is not recommended if there is a way to set all the outputs state at one time. This method can NOT be used if there is a "SetAll" command in place. The "SetAll" method is usually more efficient and will result in more accurate timing. The IO64 product file using this method is shown below.



```
command
    name                    On
    variation               By Name
    hidden                  true
    description             Turn Output or Flag On
    params
        param
            name            outputnum
            description     An output name
            rules
                output
    messages
        message
            format          "\x02%p",outputnum
command
    name                    Off
    variation               By Name
    hidden                  true
    description             Turn Output or Flag off
    params
        param
            name            output
            description     An output name
            rules
                output
```

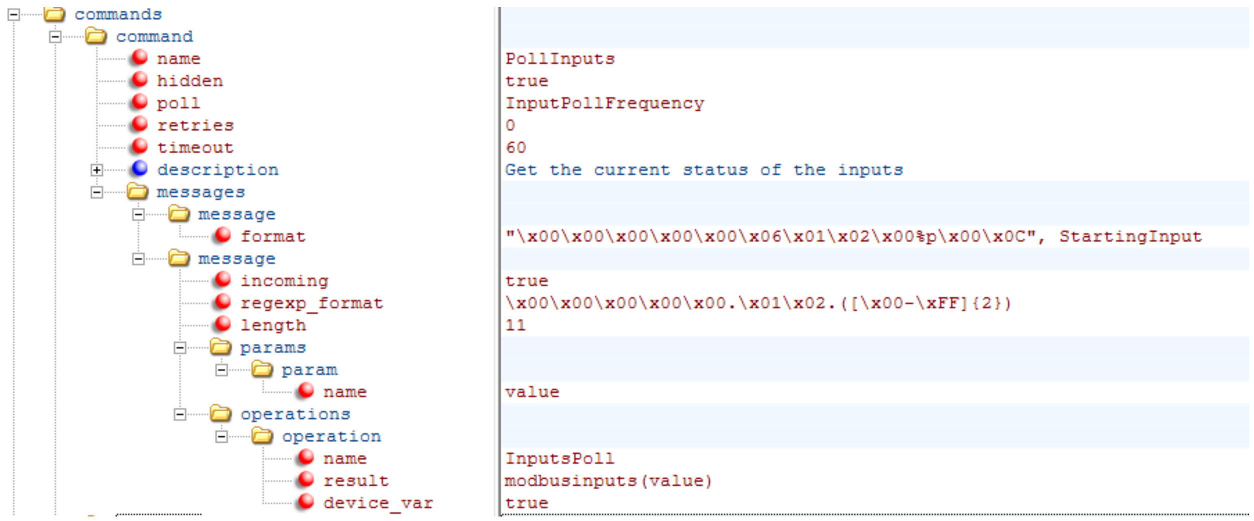
Figure 11 - Alternate Method IO64 Style "On" and "Off" individual

Getting Input Status with Constant Polling

There is only 1 command required for polling the inputs, and that is "PollInputs." The response to this can be parsed directly, or set as a separate unsolicited response.

PollInputs Command - ModbusInputs

Aside from the special name of "PollInputs", this command also has the special property "poll." The value for this attribute is the number of frames that the Input's state can be polled. If this value is zero, the Input's state will never be obtained. This value can also be a variable. In this example, we'll use the variable we created called "InputPollFrequency."



```
commands
  command
    name                    PollInputs
    hidden                  true
    poll                    InputPollFrequency
    retries                 0
    timeout                 60
    description             Get the current status of the inputs
    messages
      message
        format              "\x00\x00\x00\x00\x00\x06\x01\x02\x00%p\x00\x0C", StartingInput
      message
        incoming            true
        regexp_format       \x00\x00\x00\x00\x00.\x01\x02.([\x00-\xFF]{2})
        length              11
        params
          param
            name            value
        operations
          operation
            name            InputsPoll
            result          modbusinputs(value)
            device_var      true
```

Figure 12 - PollInputs Command

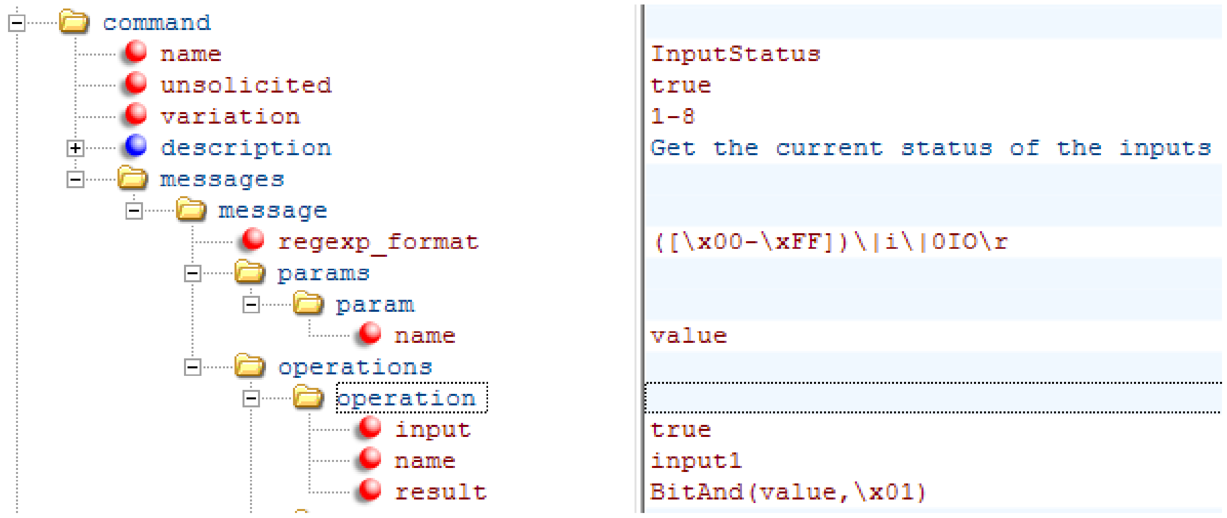
A portion of the incoming status message (noted by the parenthesis) is stored in the temporary variable "value."

From there, the message is parsed into the status of the Inputs using the function "modbusInputs." This function assumes that there are bits for each Input in the variable "value." It separates these bits and stores the "on" or "off" state sequentially into the each "Input" in WinScriptLive.

Alternatively, use “**storeinputs**(<data>, <starting input>)” function to store an input value for each single byte of data (0x00 or 0x01).

Note: the device variable "InputsPoll" is used so that the V16Pro will process the response. Without this "device_var" tag and variable, the function "modbusInputs()" would never be called.

As an alternative to using "modbusinputs", any math operation (such as BitAnd()) may be used instead of the "modbusInputs()". To finally store the value in an individual Input, the parameter can have the attribute "Input" to note that this is the value that should be stored in the WinScriptLive Input. This method is used in the AMI_io64 product file (section shown below). In the IO64 product file, the responses are separate 'unsolicited messages' rather than a direct response to the "PollInputs" command.



```
command
  name                      InputStatus
  unsolicited               true
  variation                 1-8
  description               Get the current status of the inputs
  messages
    message
      regexp_format         ([\x00-\xFF])\|i\|0I0\r
      params
        param
          name              value
      operations
        operation
          input             true
          name              input1
          result            BitAnd(value,\x01)
```

Figure 13 - IO64 method of storing inputs

Multiple Solutions for IO Product Files

There are many methods to writing product files that involve inputs and outputs. For many files, it may be more efficient to store the responses from polling into "Device Variable" Lists (like arrays), rather than setting up the "integrate_io" method shown above. While the above method is complex, it does offer the tightest integration into the WinScriptLive software.