Welcome to Yaskawa America's Training Café Express

- To make this Café enjoyable for all, please follow these tips.
  - Please do not put us on hold. Others will hear the hold music.
  - Do not use a speaker phone. Background noise can be heard.
  - We welcome comments and questions.
    You can type questions into the "Chat" window. Please send to 'All Panelists'
  - Questions not answered during the Café can be e-mailed to training@yaskawa.com or can be entered into the survey sent to you at the end of the class.

YASKAWA

Driving value

EtherNet/IP Explicit Messaging with MPiec Controllers

Nishant Unnikrishnan
Motion Application Engineer

YASKAWA

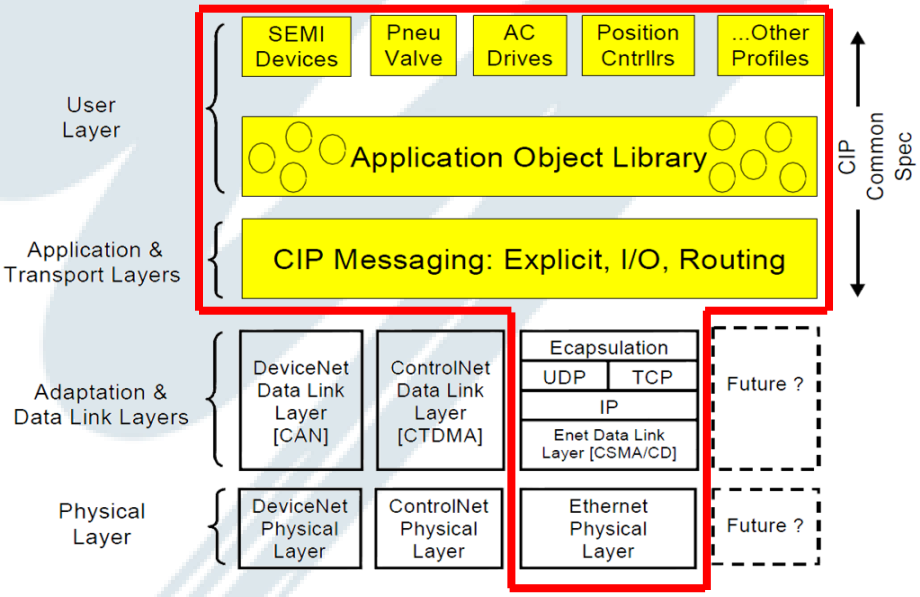
Topics

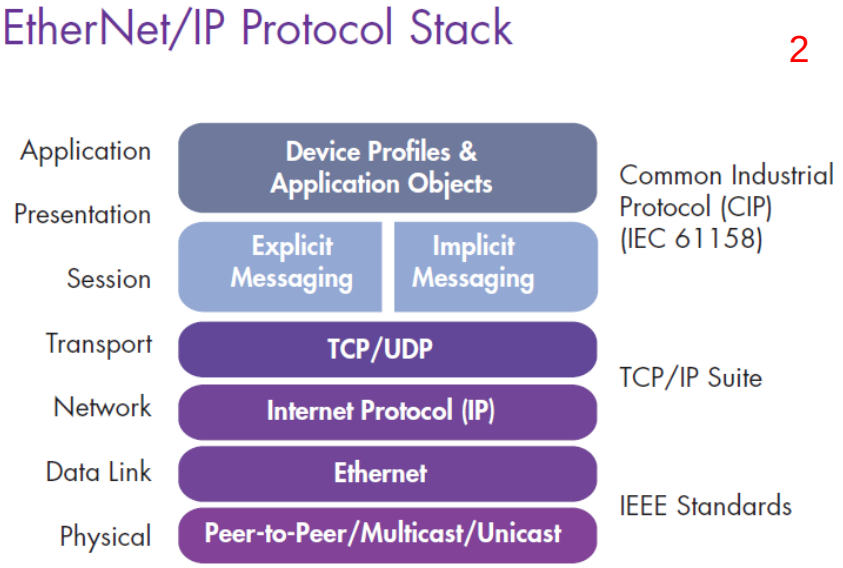
- *Explicit Messaging: Definition*
- *Explicit vs. Implicit messaging*
- *YDeviceComm library*
- *How to set up explicit messaging with YDeviceComm library*
- *Examples*
- *Function Block*

 |

EtherNet/IP

EtherNet/IP uses CIP (Control and Information Protocol), the common network, transport and application layers also shared by ControlNet and DeviceNet. EtherNet/IP then makes use of standard Ethernet and TCP/IP technology to transport CIP communications packets. The result is a common, open application layer on top of open and highly popular Ethernet and TCP/IP protocols.[1]





1 Recommended Functionality for EtherNet/IP Devices, Version 1.2, Feb 16, 2006, EtherNet/IP Implementors Workshop, ODVA

2 http://literature.rockwellautomation.com/idc/groups/literature/documents/ar/5058h-ar103_-en-p.pdf

Explicit vs. Implicit messaging

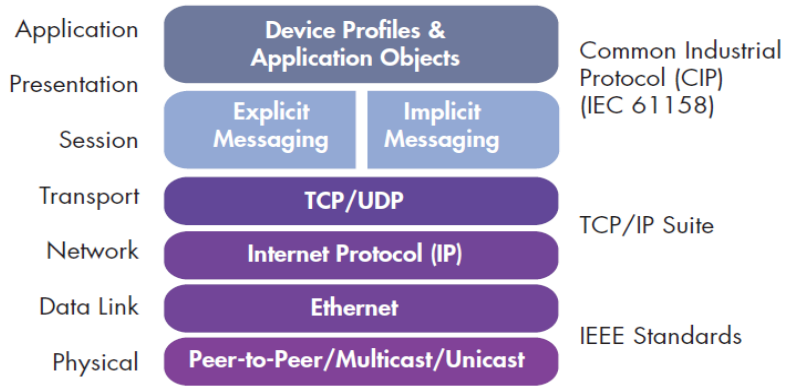
Explicit Messaging

- Uses TCP/IP for messaging
- Unscheduled
- Suited for less frequent operations
- Use request/response structure

Implicit Messaging

- Uses UDP/IP for messaging
- Packets are time critical, scheduled (use RPI)
- Suited for frequent operations
- The UDP packets are usually multicast if more than one data consumer exists

EtherNet/IP Protocol Stack



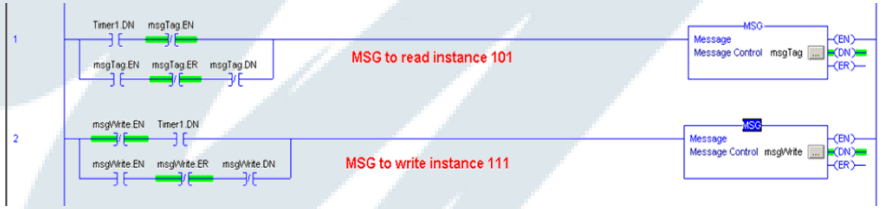
 |

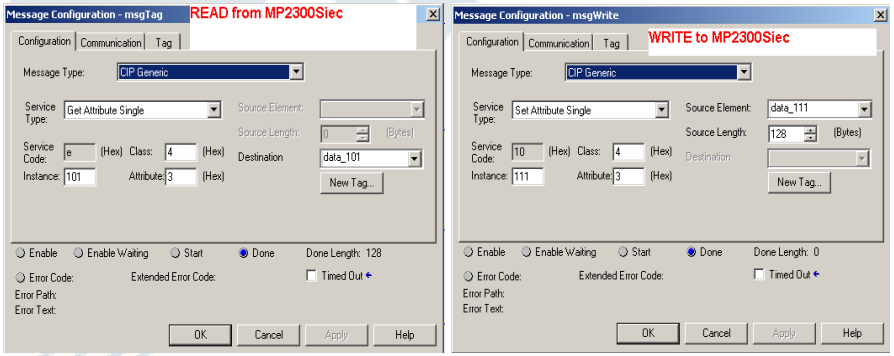
Explicit vs. Implicit messaging

Explicit Messaging (Examples)

- Sending a CAM table
- Changing jobs in a vision system
- Setting ranges in encoders
- Reading S/W versions on adapters

MSG instruction in RSLogix

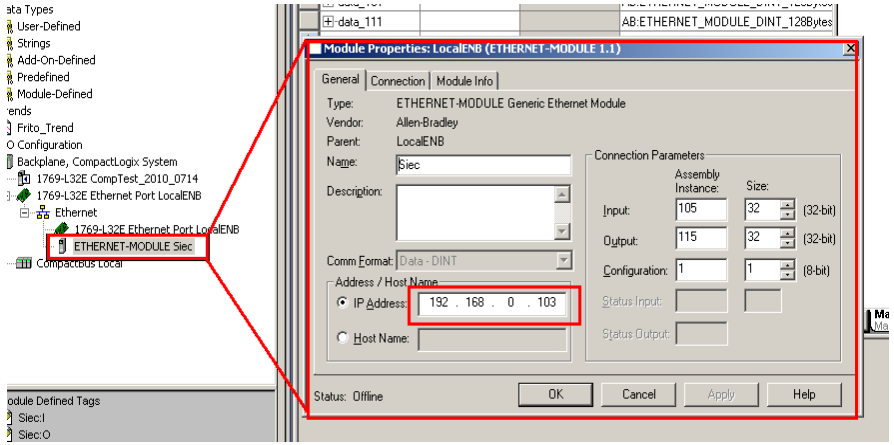




Implicit Messaging (Examples)

- Monitoring sensors
- HMI interfacing
- Multiple consumers on the network
- Heartbeat monitoring

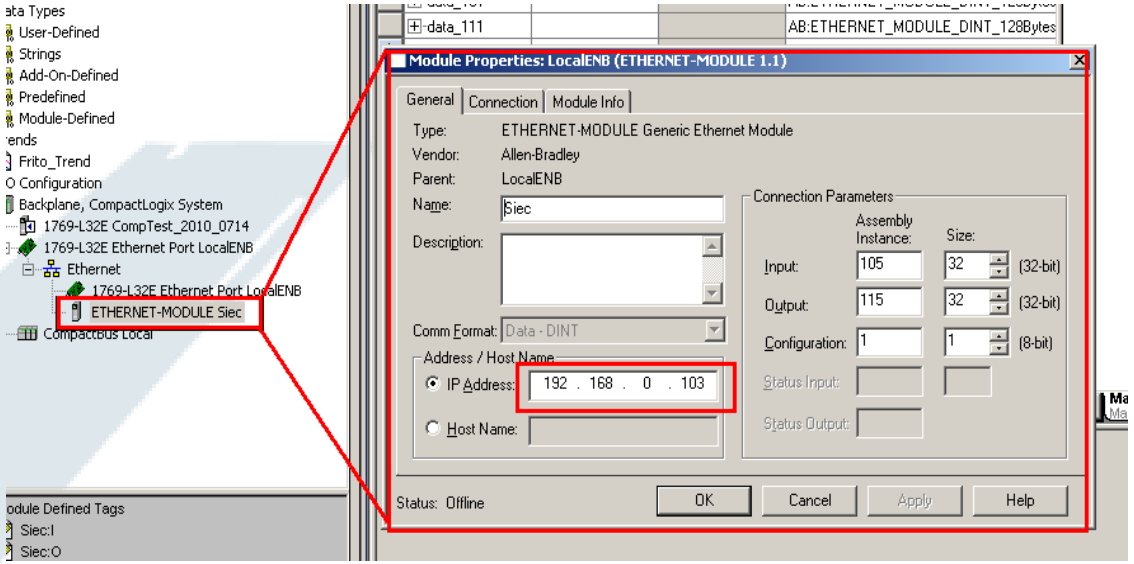
IO polling in RSLogix



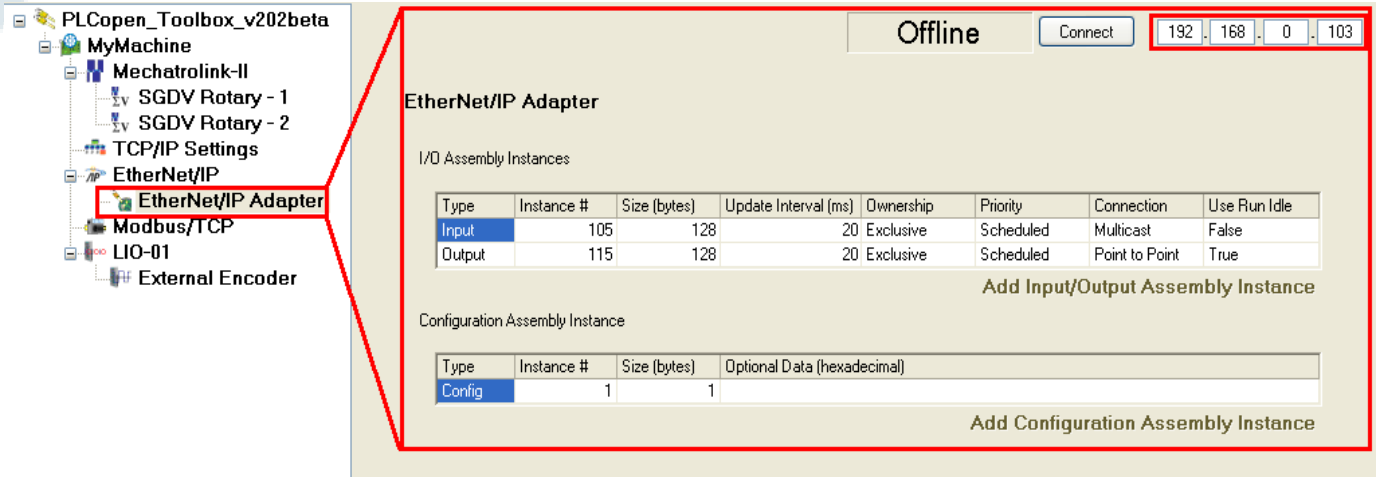
 |

Implicit messaging (comparison)

RS LOGIX

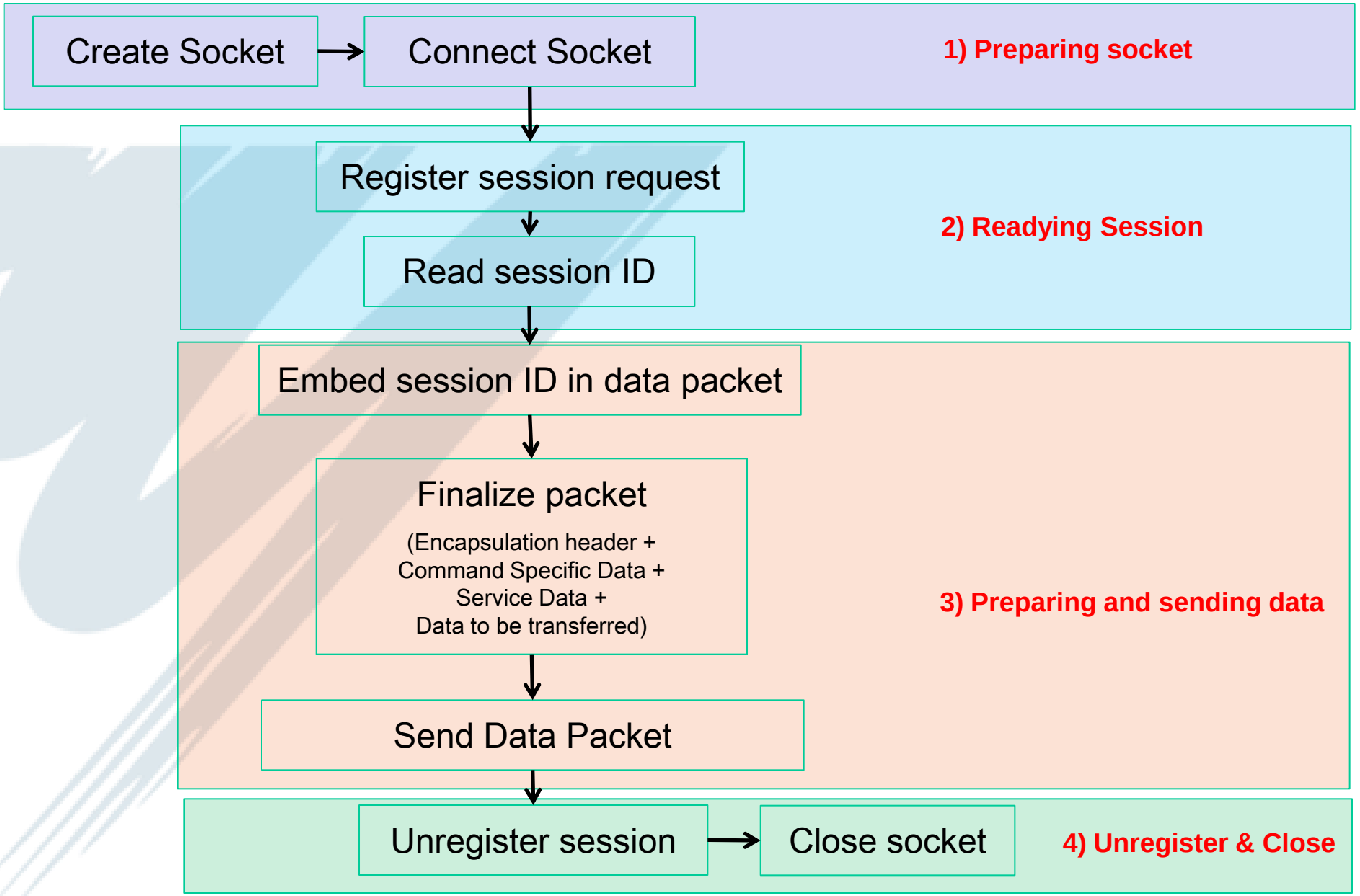


MOTIONWORKS IEC



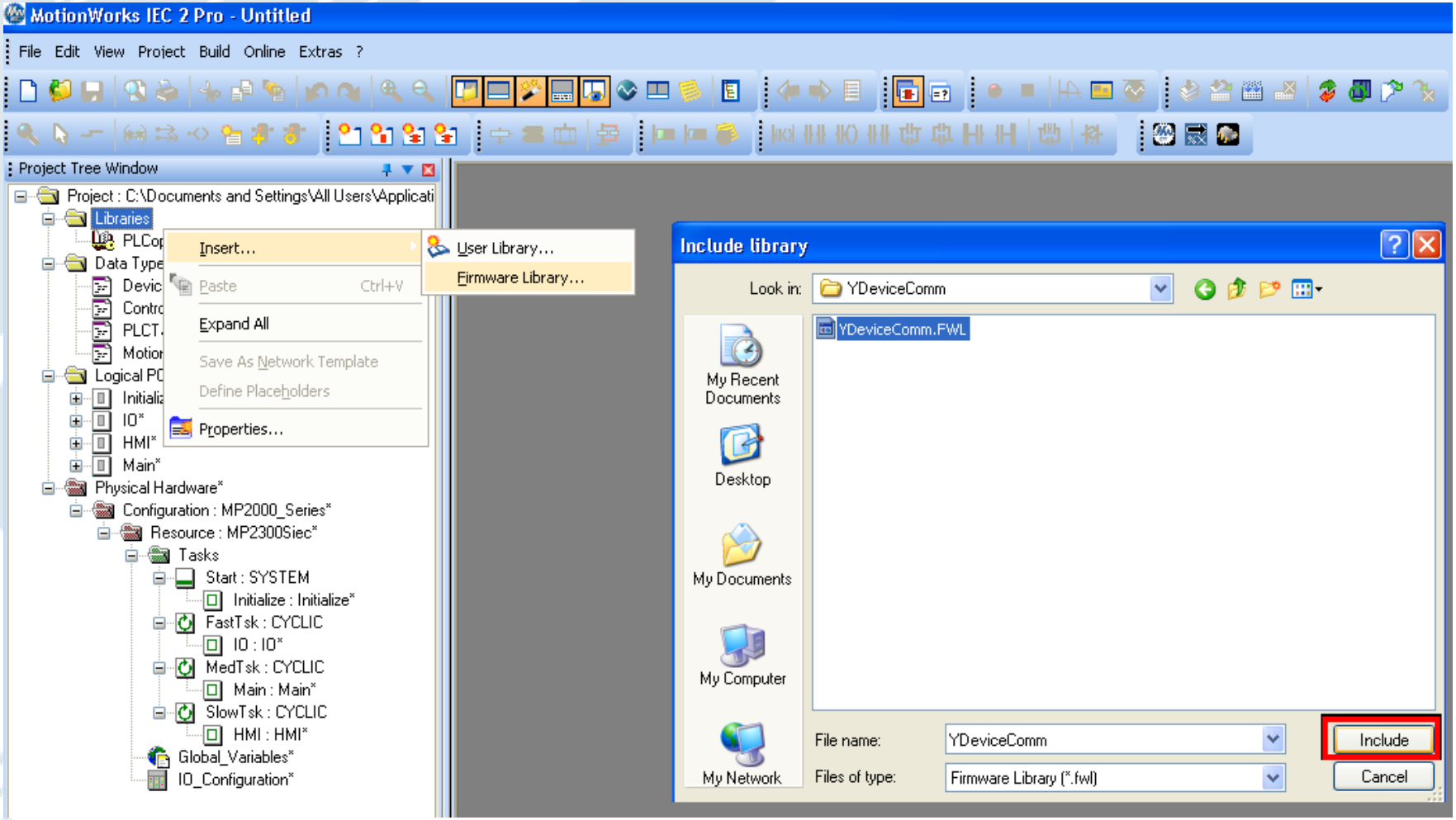
 |

Explicit Messaging with MPiec

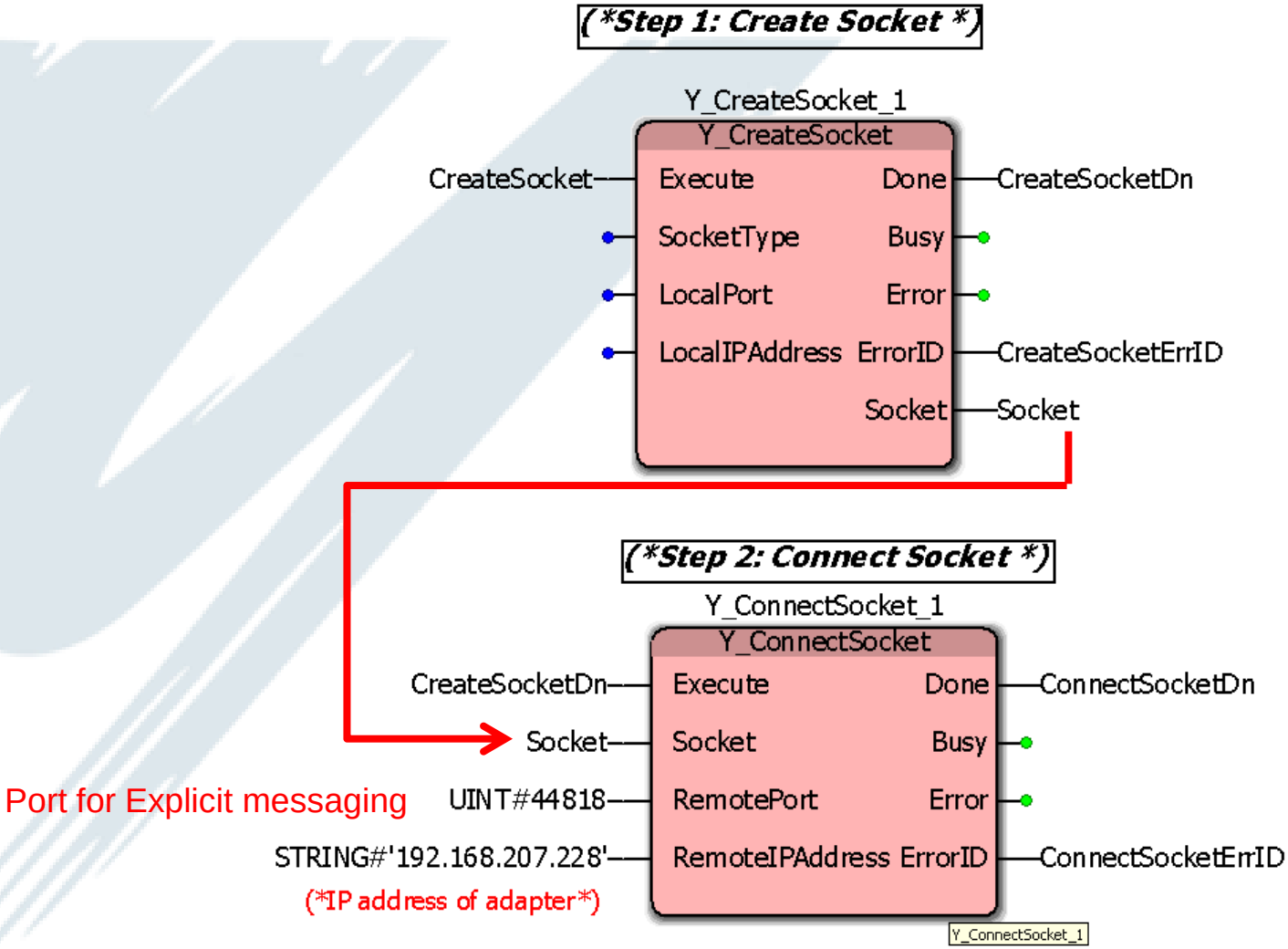


YDeviceComm Firmware Library

YDeviceComm firmware library:
Firmware version: 2.1
Software version: 2.1



Step1: Preparing socket



Step 2: Registering session


```
(*================================================================*)
(*================ REGISTER SESSION REQUEST ======================*)
(*================================================================*)
(*Encapsulation Header*)
RegSessionReqData[0] := BYTE#16#65;
RegSessionReqData[1] := BYTE#16#00; (*Command: Register session*)
RegSessionReqData[2] := BYTE#16#04;
RegSessionReqData[3] := BYTE#16#00; (*Length*)
FOR i := 4 TO 27 BY 1
    DO
    RegSessionReqData[i] := BYTE#16#00;
END_FOR;
RegSessionReqData[24] := BYTE#16#01; (*Protocol version*)
```

Array of Bytes

45	5.999271	192.168.207.222	192.168.207.78	TCP	20547 > apx500api-2 [PSH, ACK] Seq=489 Ack=213 Win=17640 Len=26
46	6.055533	192.168.207.222	192.168.207.228	ENIP	Register Session (Req), Session: 0x00000000
47	6.056004	192.168.207.228	192.168.207.222	ENIP	Register Session (Rsp), Session: 0x00000006
48	6.056535	192.168.207.222	192.168.207.228	CIP CM	Forward Open
49	6.057442	192.168.207.228	192.168.207.222	CIP CM	Success

Step 3: Preparing and sending data

```
347 1.315317 192.168.207.241 192.168.207.228 CIP Set Attribute Single
⊞ Frame 347: 242 bytes on wire (1936 bits), 242 bytes captured (1936 bits)
⊞ Ethernet II, Src: Rockwell_21:bd:49 (00:00:bc:21:bd:49), Dst: YaskawaE_26:64:de (00:20:b5:26:64:de)
⊞ Internet Protocol, Src: 192.168.207.241 (192.168.207.241), Dst: 192.168.207.228 (192.168.207.228)
⊞ Transmission Control Protocol, Src Port: omnisky (2056), Dst Port: EtherNet/IP-2 (44818), Seq: 1, Ack: 1, Len: 176
⊟ EtherNet/IP (Industrial Protocol), Session: 0x00000001, Send RR Data
  ⊟ Encapsulation Header
      Command: Send RR Data (0x006f)
      Length: 152
      Session Handle: 0x00000001
      Status: Success (0x00000000)
      Sender Context: c0a8cff10000b81a
      Options: 0x00000000
  ⊟ Command Specific Data
      Interface Handle: CIP (0x00000000)
      Timeout: 0
    ⊟ Item Count: 2
      ⊟ Type ID: Null Address Item (0x0000)
          Length: 0
      ⊟ Type ID: Unconnected Data Item (0x00b2)
          Length: 136
      [Response In: 348]
⊟ Common Industrial Protocol
  ⊟ Service: Set Attribute Single (Request)
      0... .... = Request/Response: Request (0x00)
      .001 0000 = Service: Set Attribute Single (0x10)
    Request Path Size: 3 (words)
  ⊟ Request Path: Assembly Object, Instance: 0x6F, Attribute: 0x03
    ⊟ 8-Bit Logical Class Segment (0x20)
        Class: Assembly Object (0x04)
    ⊟ 8-Bit Logical Instance Segment (0x24)
        Instance: 0x6f
    ⊟ 8-Bit Logical Attribute Segment (0x30)
        Attribute: 0x03
⊟ CIP Class Generic
  ⊟ Command Specific Data
      Data: 0100000000000000000000000000000000...
```

```
0000  00 20 b5 26 64 de 00 00  bc 21 bd 49 08 00 45 00   . .&d... .!.I..E.
0010  00 e4 1e 6f 40 00 40 06  fa 7d c0 a8 cf f1 c0 a8   ...o@.@. .}......
0020  cf e4 08 08 af 12 6c bf  67 14 c7 3c 59 eb 80 18   ......l. g..<Y...
0030  10 00 2b 46 00 00 01 01  08 0a 00 00 7b a9 00 02   ..+F.... ....{...
0040  df a4 6f 00 98 00 01 00  00 00 00 00 00 00 c0 a8   ..o..... ........
0050  cf f1 00 00 b8 1a 00 00  00 00 00 00 00 00 00 00   ........ ........
0060  02 00 00 00 00 00 b2 00  88 00 10 03 2c 04 24 6f   ........ .... .$o
0070  30 03 01 00 00 00 00 00  00 00 00 00 00 00 00 00   0....... ........
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00f0  00 00                                             ..
```

Step 3: Preparing and sending data

YASKAWA

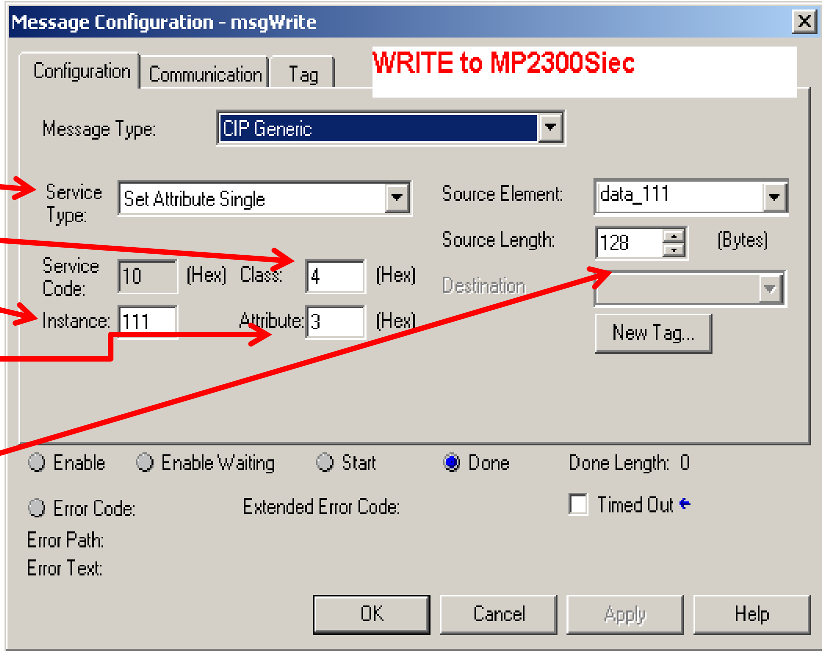
```

(*Encapsulation Header*)

SendData[0]  := BYTE#16#6f;
SendData[1]  := BYTE#16#00; (* Command: Send RR Data (006f)*)
SendData[2]  := BYTE#16#98;
SendData[3]  := BYTE#16#00; (*Length: 152  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*)
SendData[4]  := BYTE#16#01;
SendData[5]  := BYTE#16#00;
SendData[6]  := BYTE#16#00;
SendData[7]  := BYTE#16#00; (*Session Handle: Will have to be set after session is registered. See Main program*)
SendData[8]  := BYTE#16#00;
SendData[9]  := BYTE#16#00;
SendData[10]  := BYTE#16#00;
SendData[11]  := BYTE#16#00; (*Status: success*)
SendData[12]  := BYTE#16#c0;
SendData[13]  := BYTE#16#a8;
SendData[14]  := BYTE#16#cf;
SendData[15]  := BYTE#16#f1;
SendData[16]  := BYTE#16#00;
SendData[17]  := BYTE#16#00;
SendData[18]  := BYTE#16#b8;
SendData[19]  := BYTE#16#1a; (*Sender context: (Not important?????)*)
SendData[20]  := BYTE#16#00;
SendData[21]  := BYTE#16#00;
SendData[22]  := BYTE#16#00;
SendData[23]  := BYTE#16#00; (*Options*)

(*Command Specific Data*)

SendData[24]  := BYTE#16#00;
SendData[25]  := BYTE#16#00;
SendData[26]  := BYTE#16#00;
SendData[27]  := BYTE#16#00; (*Interface Handle: CIP*)
SendData[28]  := BYTE#16#00;
SendData[29]  := BYTE#16#00; (*Timeout*)
SendData[30]  := BYTE#16#02;
SendData[31]  := BYTE#16#00; (*Item Count*)
SendData[32]  := BYTE#16#00;
SendData[33]  := BYTE#16#00; (*Type ID: Null Address Item*)
SendData[34]  := BYTE#16#00;
SendData[35]  := BYTE#16#00; (*Length*)
SendData[36]  := BYTE#16#b2;
SendData[37]  := BYTE#16#00; (*Type ID: Unconnected Data item*)
SendData[38]  := BYTE#16#88;
SendData[39]  := BYTE#16#00; (*Length 136 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*)

```

Variables

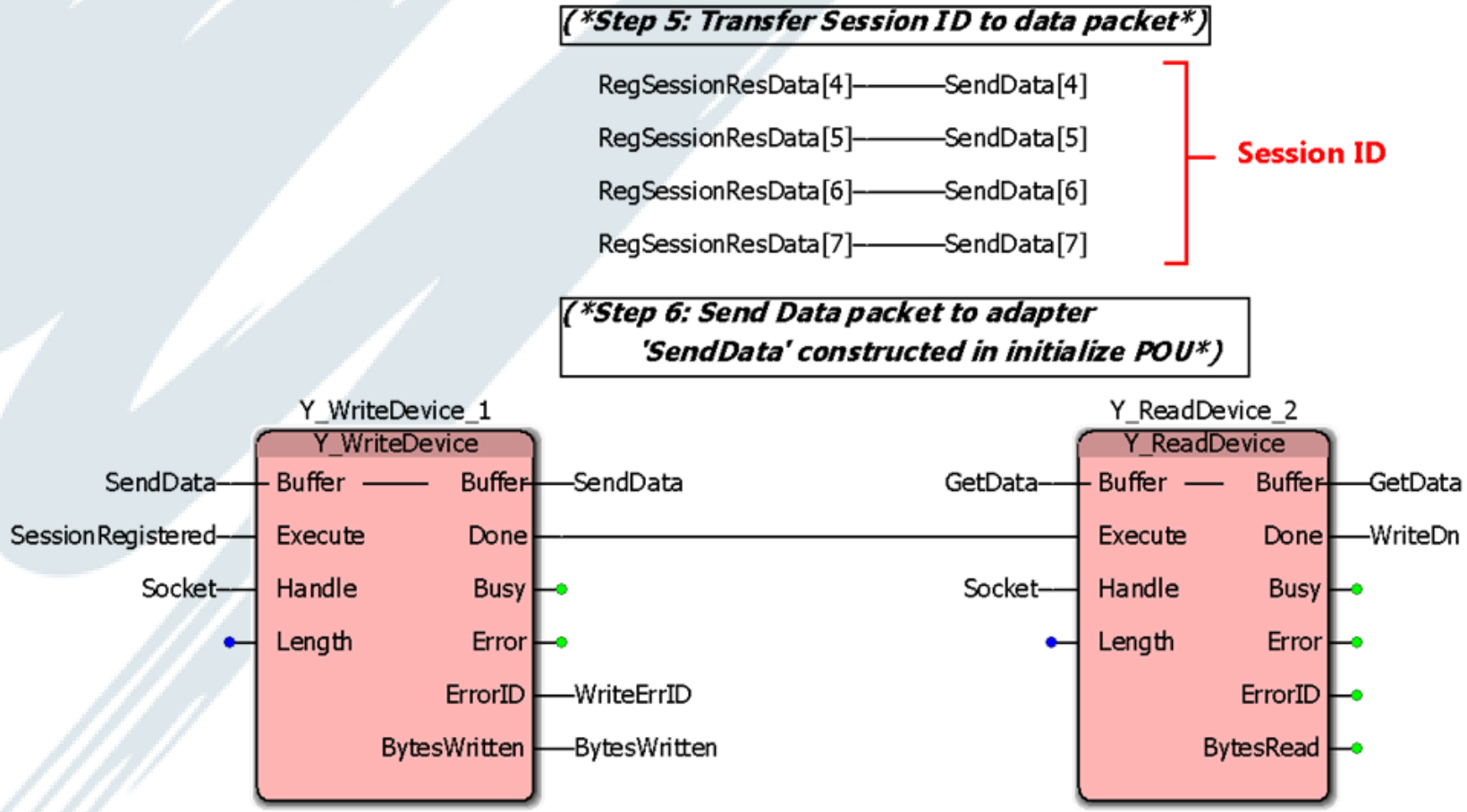
Step 3: Preparing and sending data

```
(*Common Industrial Protocol*)
(*=========================*)

(*Service*)

SendData[40] := BYTE#16#10; (*Set Attribute SIngle*)
SendData[41] := BYTE#16#03; (*Request Path size*)
SendData[42] := BYTE#16#20; (*Logical Class Segment*)
SendData[43] := BYTE#16#04; (*Class: Assembly Object <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*)
SendData[44] := BYTE#16#24; (*Logical Instance Segment*)
SendData[45] := BYTE#16#6f; (*Instance <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*)
SendData[46] := BYTE#16#30; (*Logical attribute Segment*)
SendData[47] := BYTE#16#03; (*Attribute <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*)

(*Data starts here*)
(*================*)
SendData[48] := BYTE#16#01; (* first byte of data*)


FOR k := 49 TO 175 BY 1
    DO
    SendData[k] := BYTE#16#0;
END_FOR;
```

Bytes [48..175]

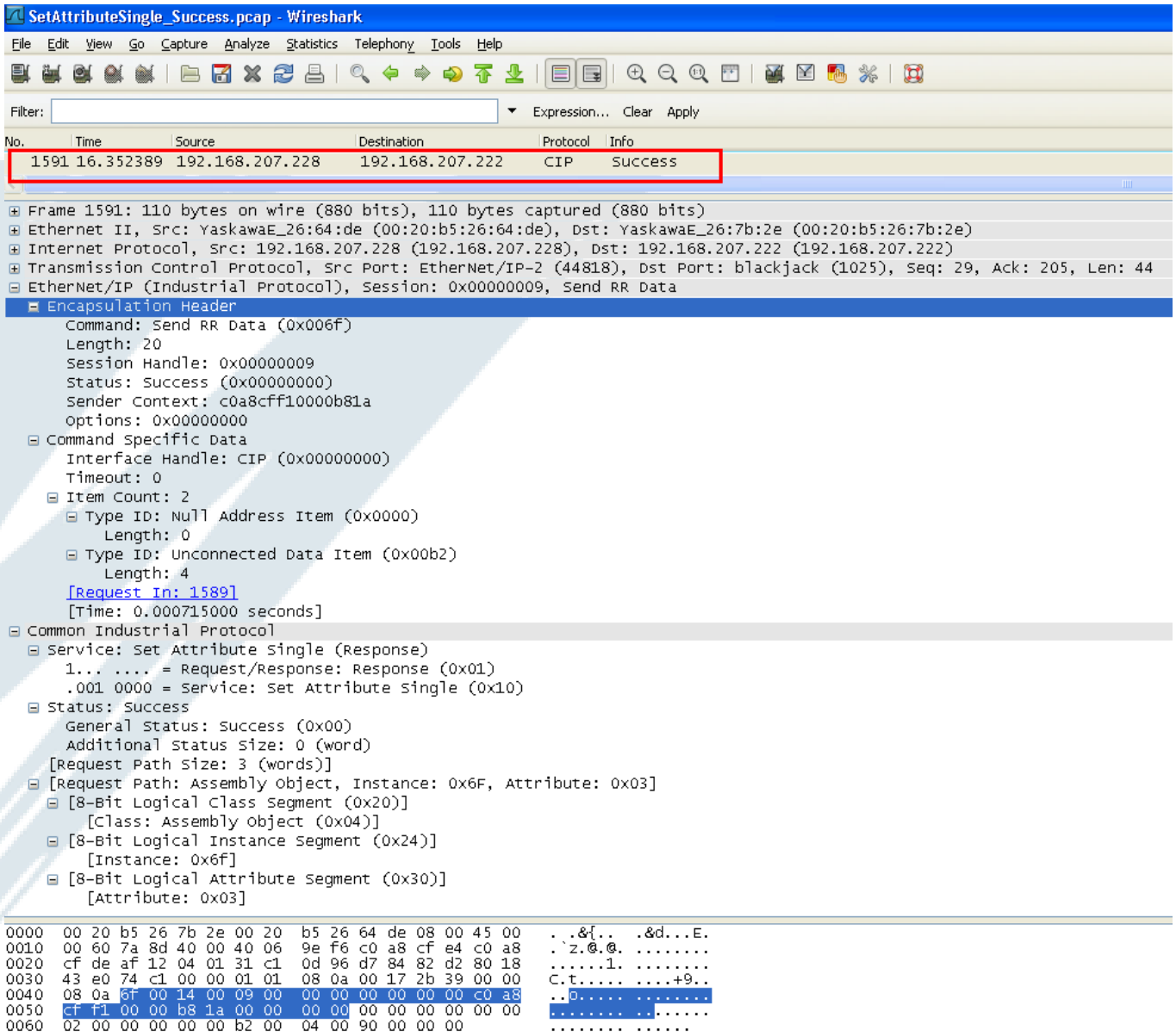


Step 3: Preparing and sending data



 |

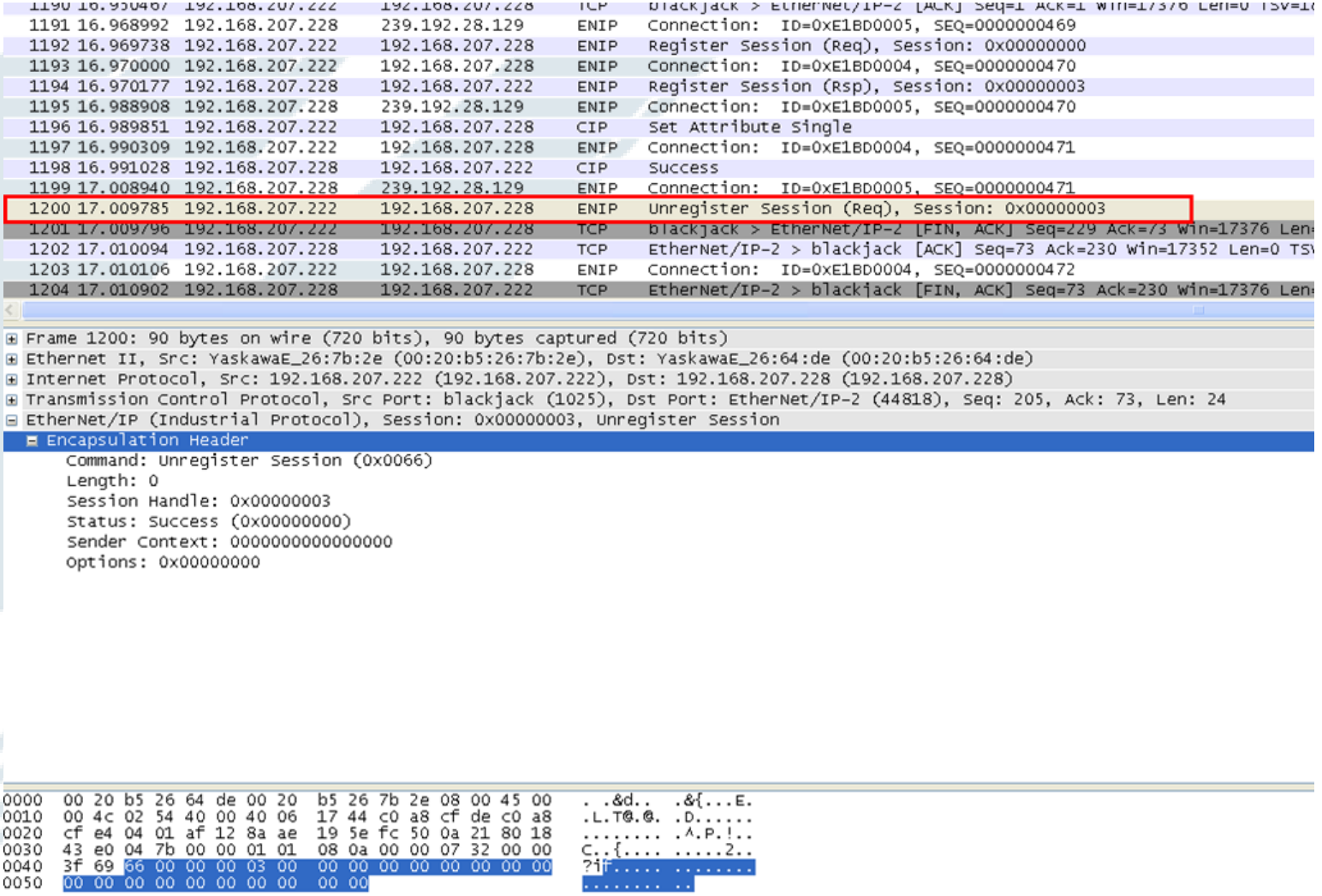
Step 3: Preparing and sending data



Step 4: Unregister and close

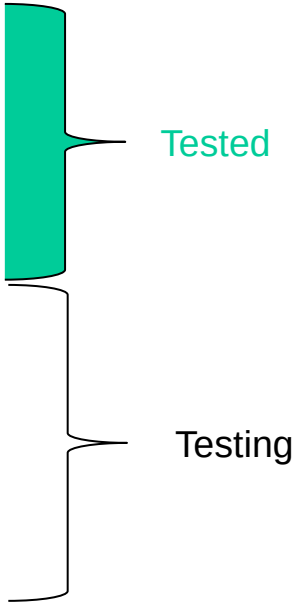
 |

Step 4: Unregister and close



Tests

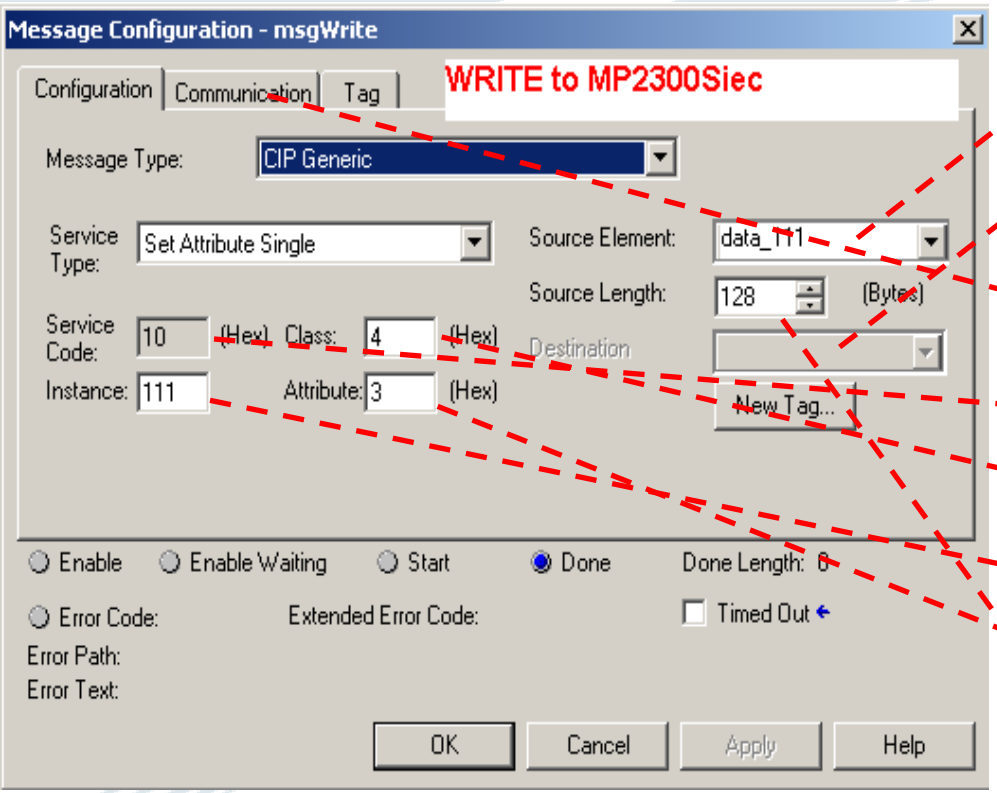
Client	Server
MP2000iec	MP2000iec
MP2000iec	Cognex InSight sensor (used to change a job)
MP2000iec	Yaskawa V1000 VFD
MP2000iec	Baumer encoder

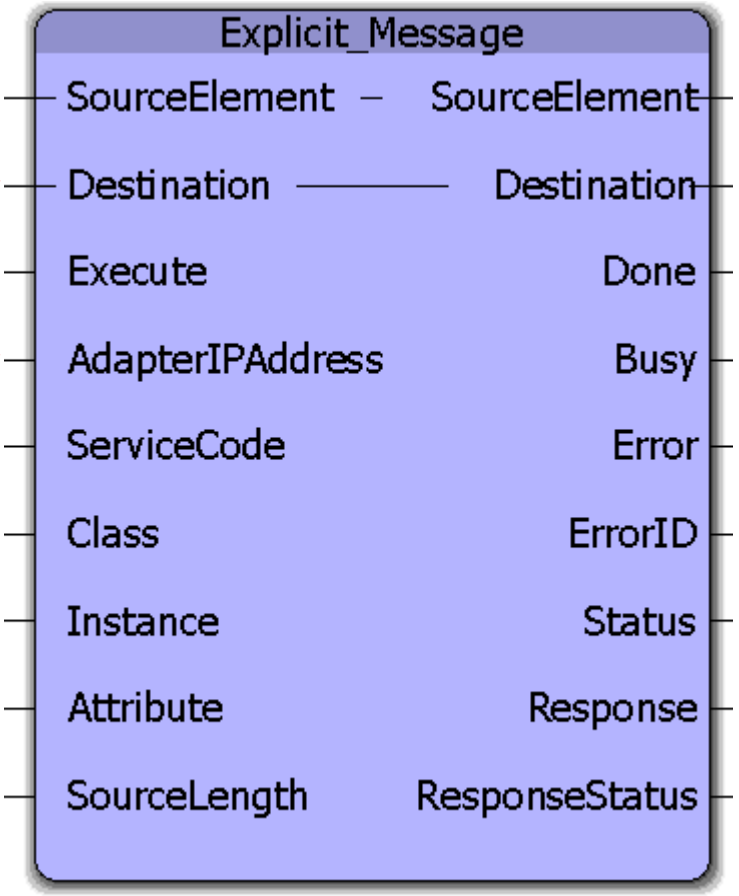


Function Block in Yaskawa Toolbox v202

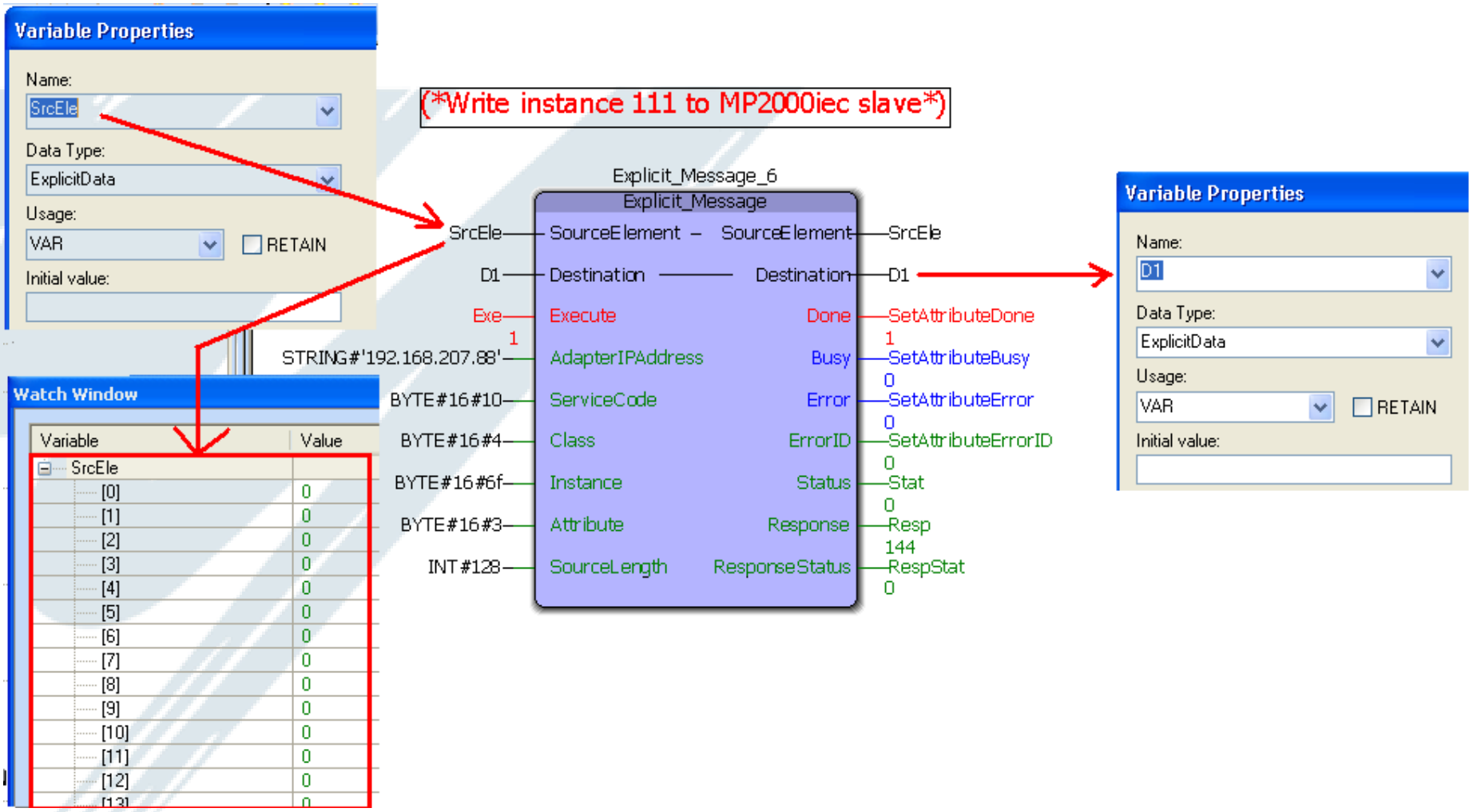
Q: What does a user have to do for explicit messaging with the MPiec controller as a scanner (master) ?

A: Use firmware v 2.1 and software 2.1. Use the Explicit_Message function block from Yaskawa Toolbox v 202. Enter parameters as entered in Message Configuration in RSLogix

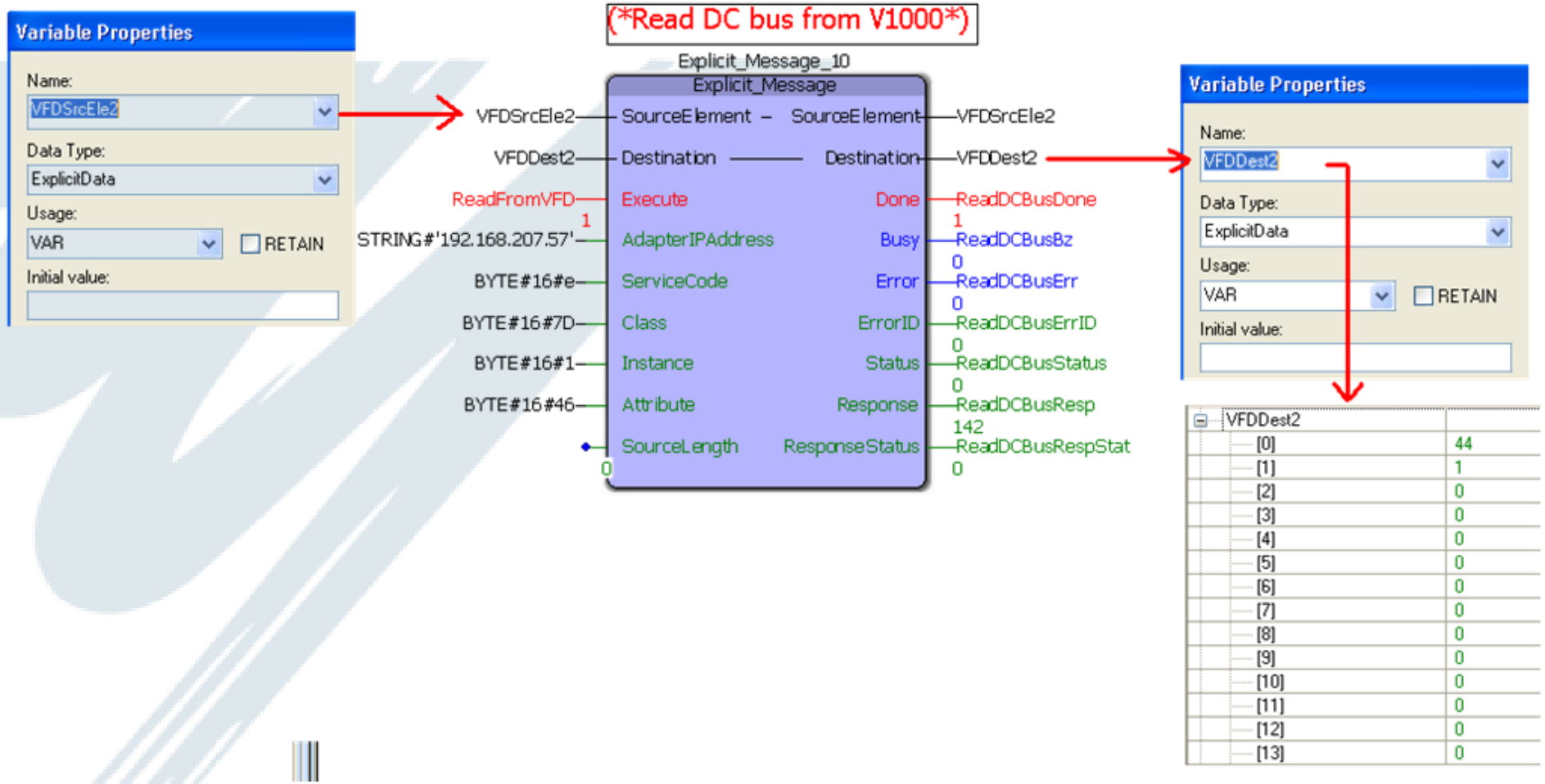




Example: Set Attribute Single (Write) to MP2000iec



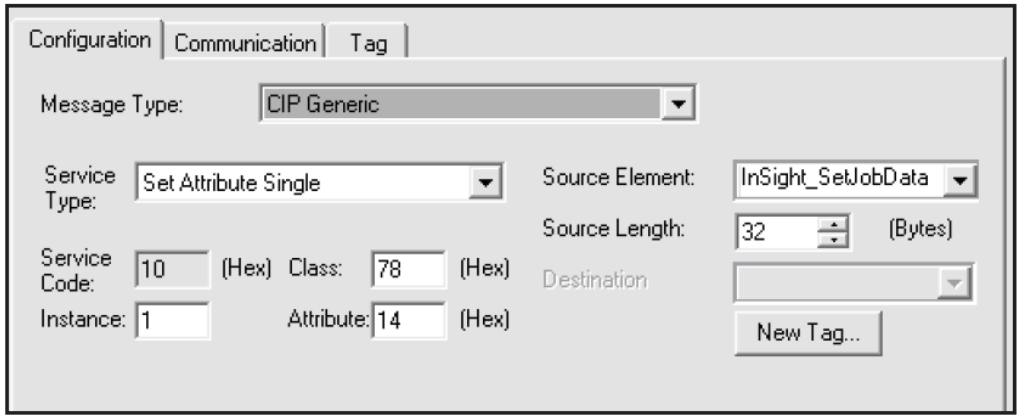
Example: Get Attribute Single (Read) from V1000



Troubleshooting tools

Wireshark

Hub (not switch)



Configuration using RSLogix

 |

3. Using Explicit Messaging with ControlLogix

Unlike implicit messages, explicit messages are sent to a specific device and that device always responds with a reply to that message. As a result, explicit messages are better suited for operations that occur less frequently. Explicit messages can be used to read and write the *Attributes* in the Ethernet/IP *Vision Object* of the In-Sight Vision System, which may be used for changing jobs, acquiring images, sending Native Mode commands and retrieving result data.

3.1. Changing a Job on an In-Sight Sensor

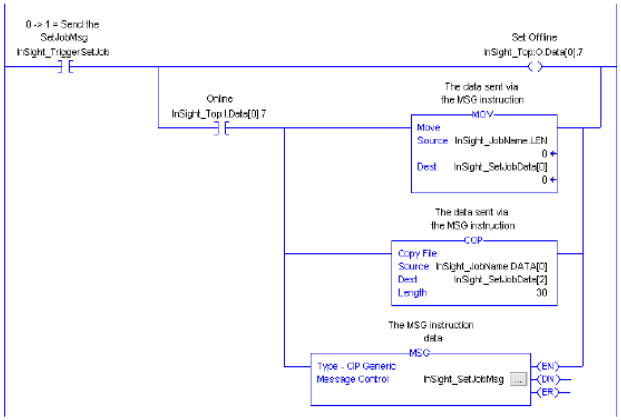
The most common explicit messaging operation performed between an In-Sight sensor and a ControlLogix PLC is the changing of jobs. Within the ControlLogix PLC, explicit messages are sent using the MSG instructions.

The following steps illustrate how to add a MSG instruction in RSLogix to change the current job on an In-Sight sensor:

1. Add the following tags to the ControlLogix *Controller Tags* dialog:

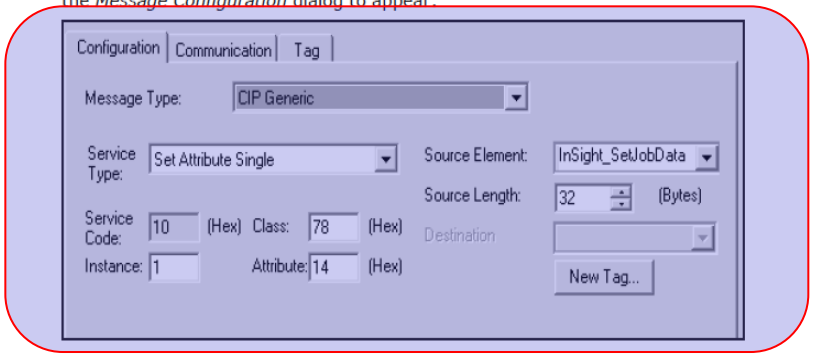
Name	Data Type	Description
⊞ InSight_SetJobMsg	MESSAGE	The MSG instruction data
⊞ InSight_JobName	STRING	The new job name
⊞ InSight_SetJobData	SINT[32]	The data sent via the MSG instruction
InSight_TriggerSetJob	BOOL	0 -> 1 = Send the SetJobMsg

2. Create the following rung in the MainRoutine of your ControlLogix project:



This rung uses the *Set Offline* bit in the implicit connection to force the In-Sight sensor Offline, because the *JobName* attribute of the *Vision Object* requires the In-Sight sensor to be Offline before it will change the job. The *Set Offline* bit waits for the In-Sight sensor to bring the Online bit low, then sets up the data containing the new job name and sends the MSG instruction to the In-Sight sensor. After the job change is completed, the falling edge of the *TriggerSetJob* tag will cause the In-Sight sensor to go back Online.

3. To setup the MSG instruction, click on the *InSight_SetJobMsg* ... button. This will cause the *Message Configuration* dialog to appear:



The following fields need to be configured:

- **Message Type:** This is the type of message that will be sent; choose *CIP Generic* to send a message to vendor specific objects like the *Vision Object*.
- **Service Type:** This is the service code that will be sent to the object; select *Set Attribute Single* to change a single attribute of the *Vision Object*.
- **Service Code:** This field allows any type of service to the object; this field is disabled if a predefined service type is selected.
- **Class:** This is the identifier of the class that the message will be sent to; the ID of the *Vision Object* is *Hex 78*.
- **Instance:** This field specifies the instance number of the object that the message will be sent to; for In-Sight sensors, the *Vision Object* only has one instance, so this field should be set to *1*.
- **Attribute:** The attribute number that the message will be sent to; in this case, the *JobName* attribute has the ID of *14 Hex*.
- **Source Element:** This field indicates the source for the data that is being sent with the message. For the *JobName* attribute, a *String* with a 2 byte length header needs to be sent. This string was formatted earlier using the COP and MOV instructions in the run that was created in the previous steps. This field should be set to *InSight_SetJobData* to send the new job name to the *JobName* attribute.
- **Source Length:** This field indicates the number of bytes of the source element that will be sent to the object. In this instance, because all of the data in the source element needs to be sent, the bytes should be set to *32*.

Thank you

 |