ctrlX - CORE

- **NodeRed – Part 02**

- Communicate XM with NodeRed in ctrlX Core

Jordi Laboria (DCET/SLF4-ES)

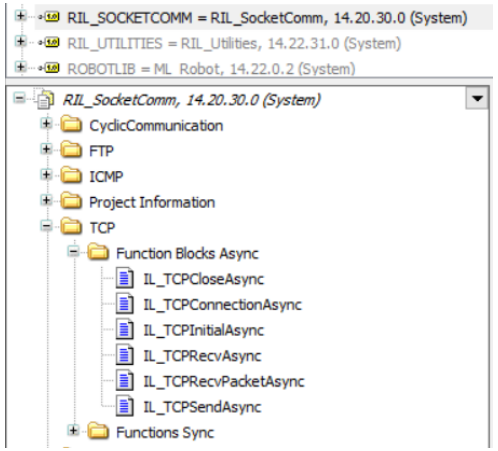
rexroth
A Bosch Company

rexroth
A Bosch Company

GOALS:

- *Data Injection Example (Visualization and Formatting)*
- *Data reading in NodeRed (ctrlX Core)*
- *Send Data from the NodeRed of the ctrlX Core and read them in the XM*

ctrlX



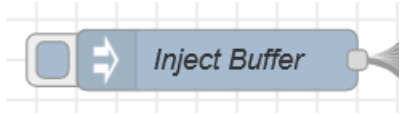
XM

Data Injection Example (Visualization and Formatting)

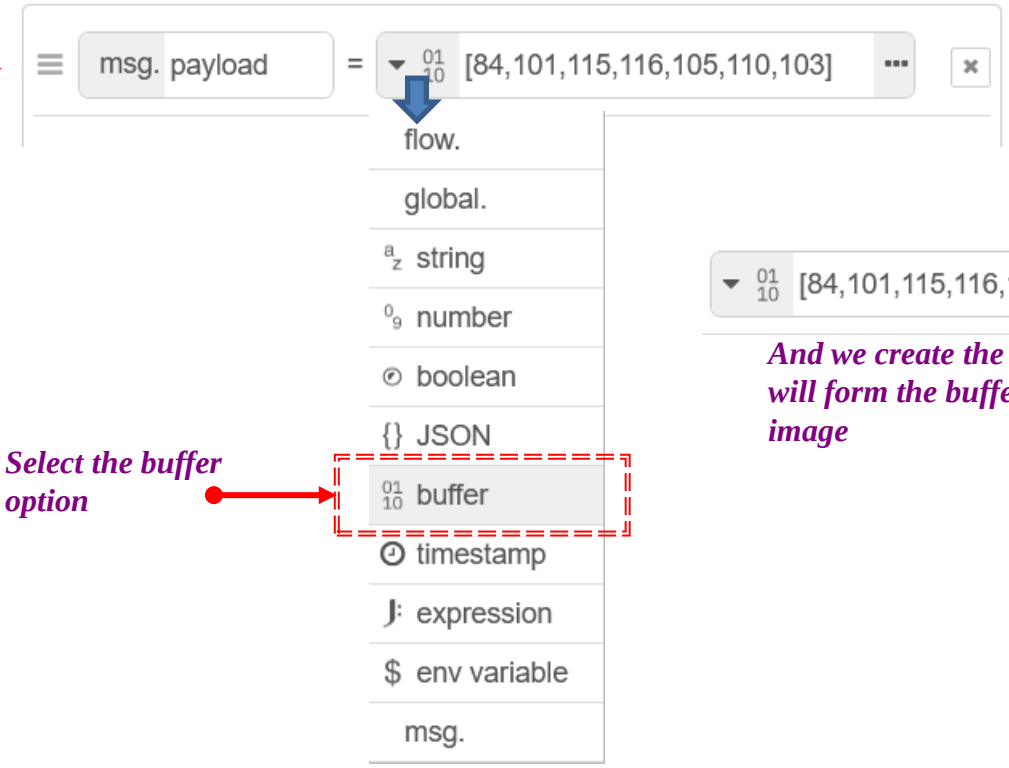
- Before we start manipulating the data coming from the XM, let's see how the management of the data sent and received by communications works. These data, in our case, will be received in buffer format, which means that we will see them all grouped and then we will have to manipulate them to separate them.

- *In order to understand the operation and before activating the communications we will see how we generate a Buffer to be able to simulate the data that we will receive and that of course we can send to the XM from or to the NodeRed*
-

The "inject" function will allow us to generate the test buffer



We will modify the name by entering the properties of the function

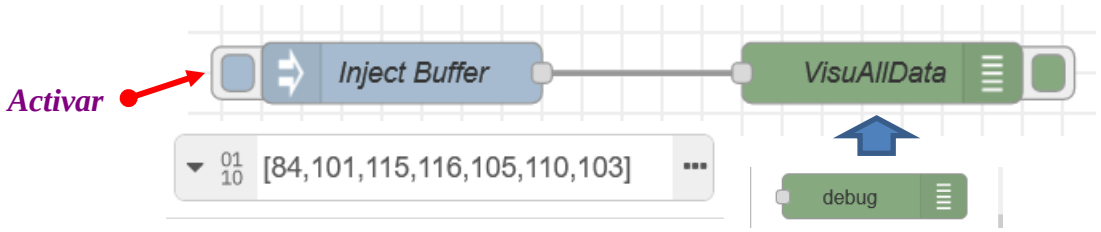


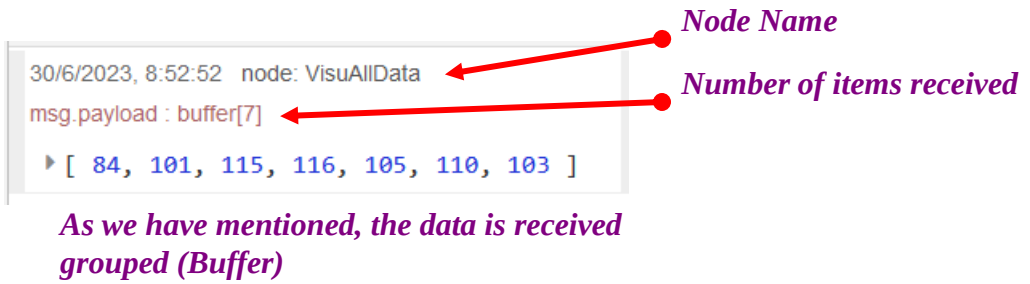
Select the buffer option

And we create the 7 elements that will form the buffer as shown in the image

Do not forget that the modifications made must be activated with "Deploy"

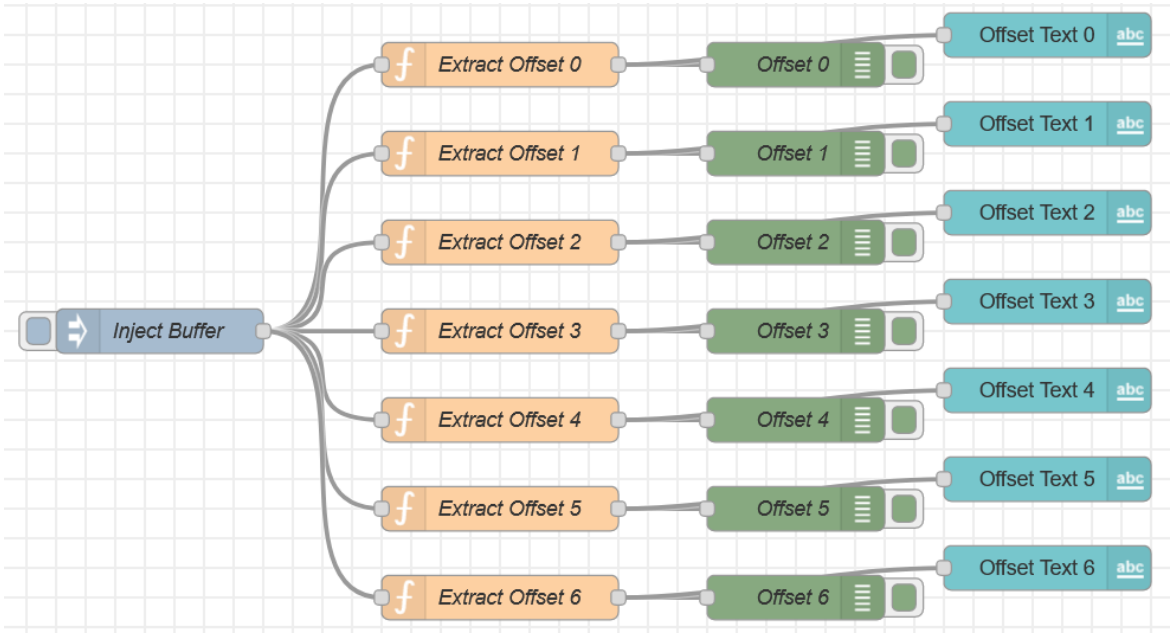
- If we then add a "debug" function and activate the "Inject Buffer" we will see the result we receive





As we have mentioned, the data is received grouped (Buffer)

- To see how the system works and "separate" the 7 elements of the Buffer received, we will use the following example



Debug

30/6/2023, 9:06:02 node: Offset 0
msg.payload : number
84

30/6/2023, 9:06:02 node: Offset 1
msg.payload : number
101

30/6/2023, 9:06:02 node: Offset 2
msg.payload : number
115

30/6/2023, 9:06:02 node: Offset 3
msg.payload : number
116

30/6/2023, 9:06:02 node: Offset 4
msg.payload : number
105

30/6/2023, 9:06:02 node: Offset 5
msg.payload : number
110

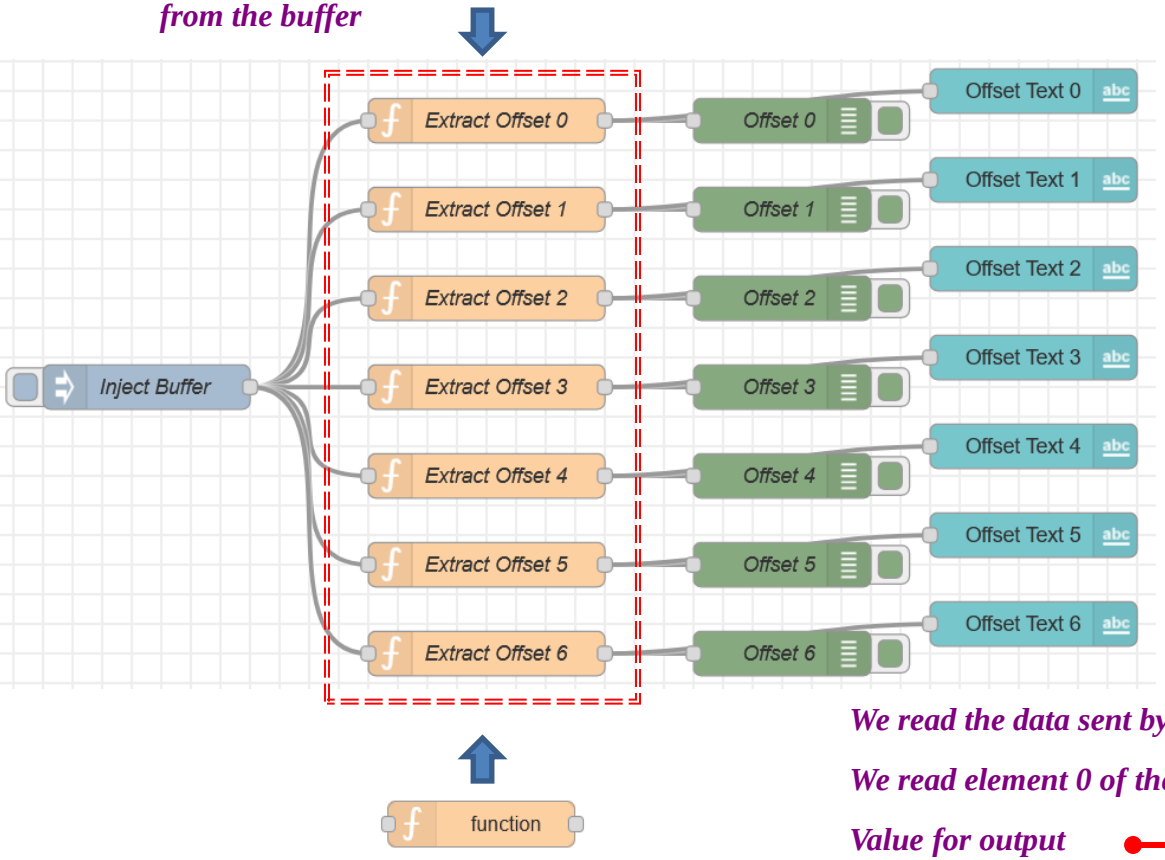
30/6/2023, 9:06:02 node: Offset 6
msg.payload : number
103

Visualization on the DashBoard

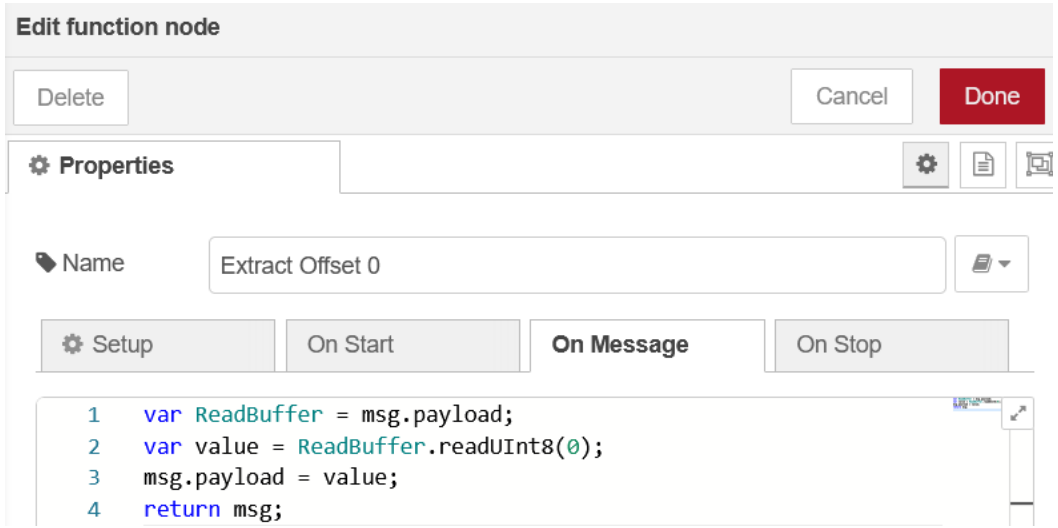
Offset Text 0	84
Offset Text 1	101
Offset Text 2	115
Offset Text 3	116
Offset Text 4	105
Offset Text 5	110
Offset Text 6	103

- The "inject Buffer" we have already seen how it should be configured, therefore, the next step is to configure the extraction functions of the different elements of the buffer received

Functions for extracting data from the buffer



To extract the data from the buffer individually we will do it as follows



We read the data sent by the previous instruction

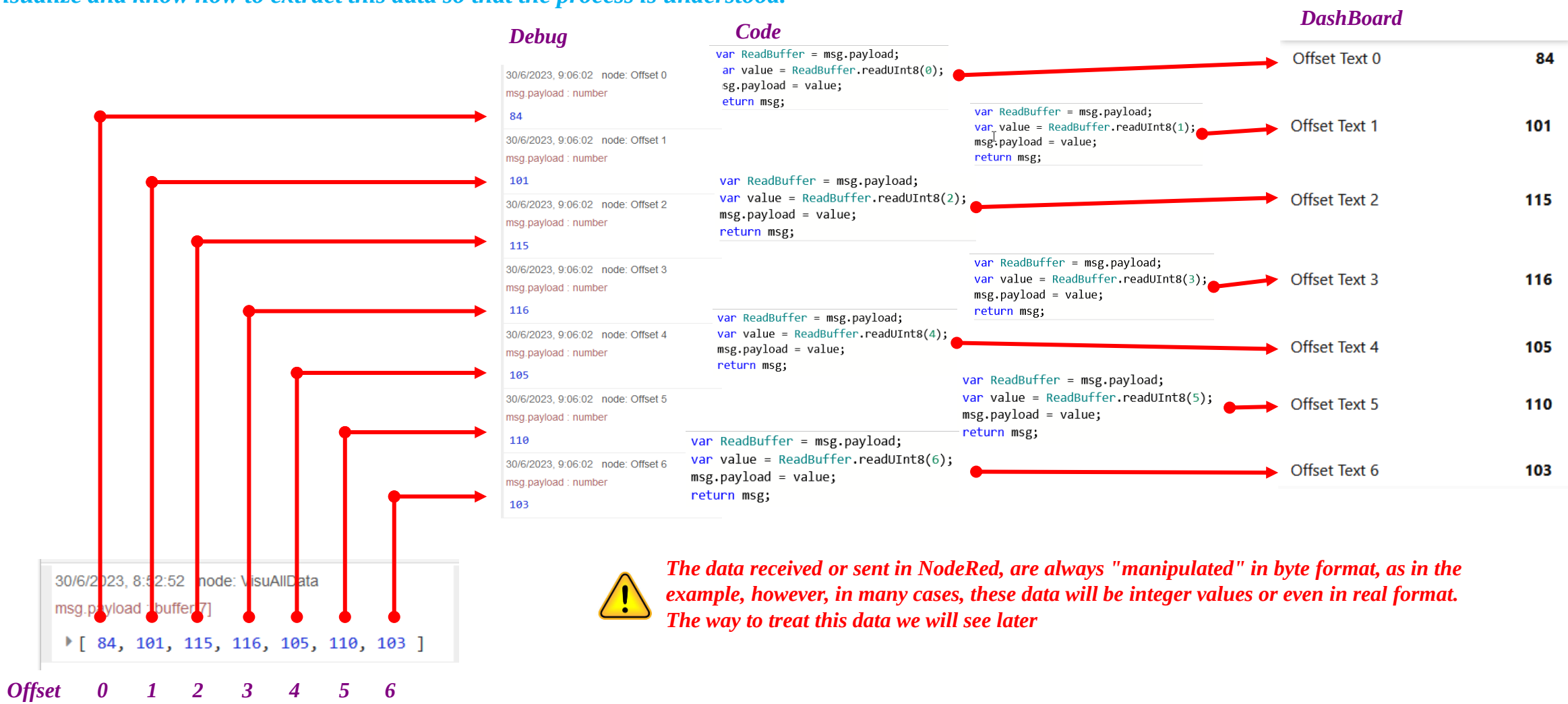
We read element 0 of the Buffer

Value for output

Message to the next statement

```
var ReadBuffer = msg.payload;
var value = ReadBuffer.readUInt8(0);
msg.payload = value;
return msg;
```

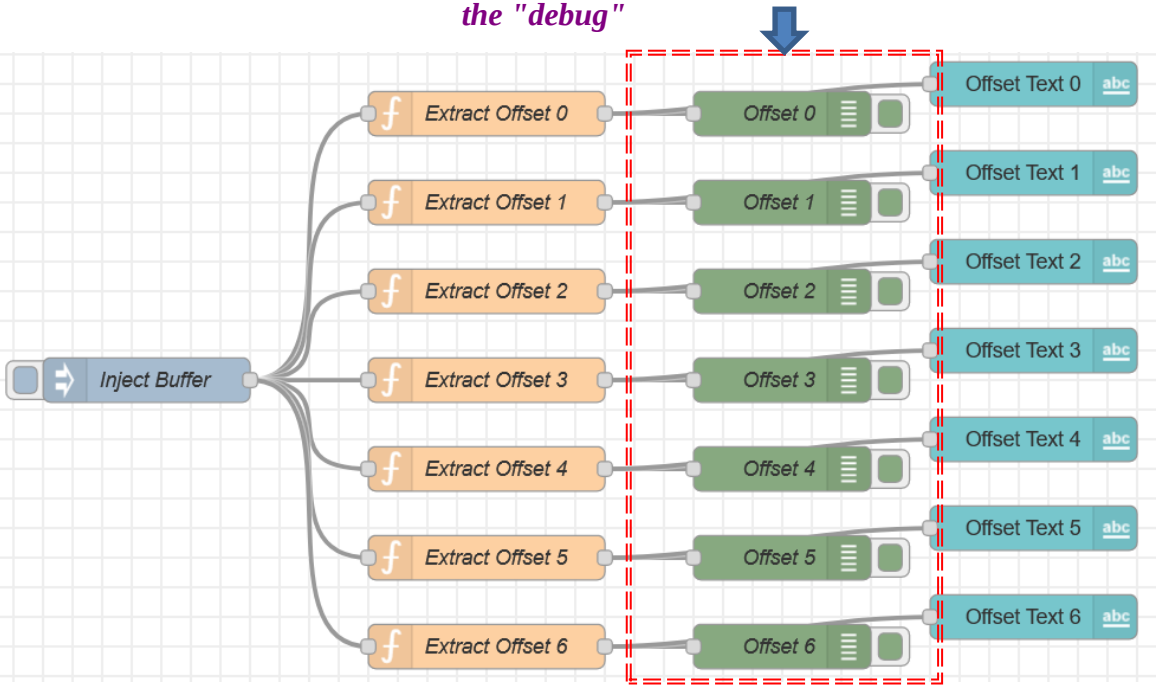
- The extraction of the buffer can be done in different ways depending on the data received or sent. In the case of the example we are using the byte option to visualize and know how to extract this data so that the process is understood.



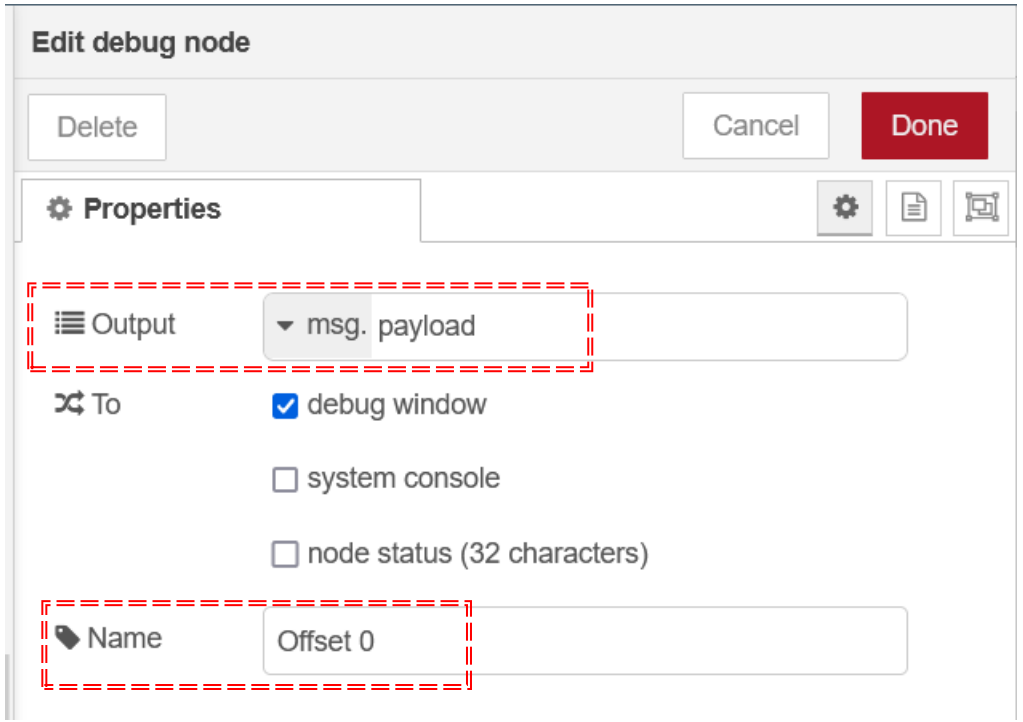
The data received or sent in NodeRed, are always "manipulated" in byte format, as in the example, however, in many cases, these data will be integer values or even in real format. The way to treat this data we will see later

- The next group of instructions are the ones we will use to visualize in the debug. It is always important to have the help of this element to be able to see what we are receiving or if the data is of the type we want to use.

Functions for data visualization in the "debug"

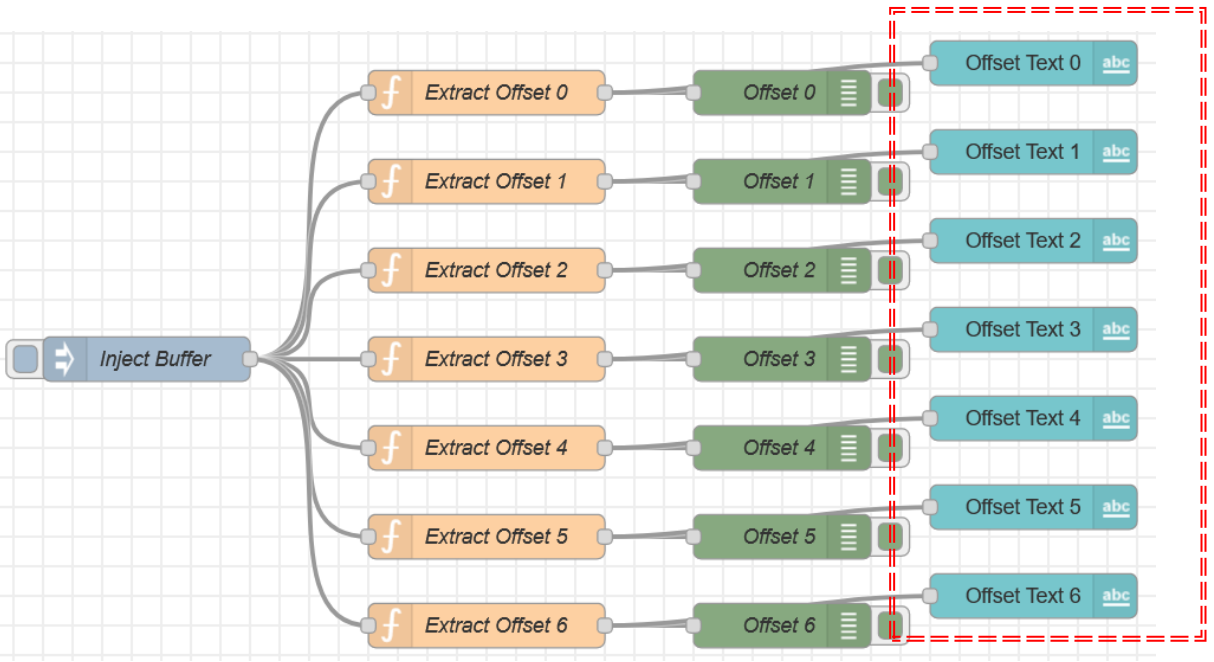


In the debug functions and given that the message received is of the type msg.payload the only thing that we will have to modify in each of them is the name



- The last function of this part of the example is the visualization in the "DashBoard"

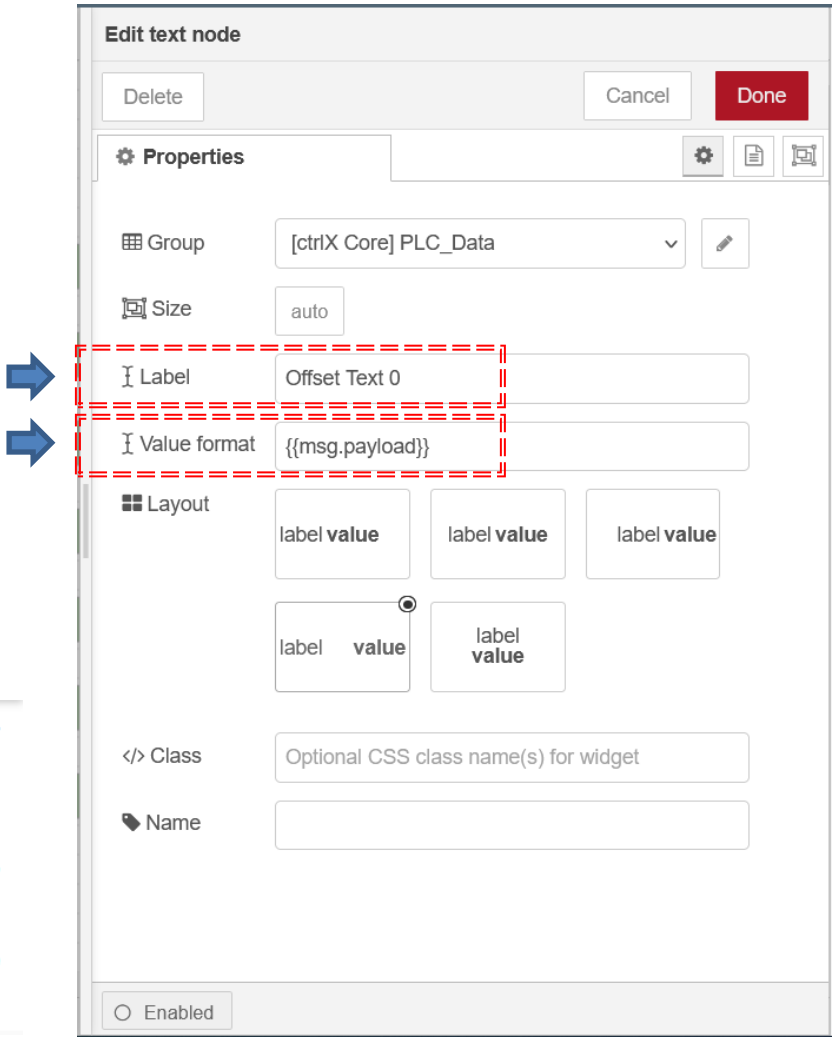
Functions for data visualization in the "DashBoard"



Visualization on the DashBoard

Offset Text 0	**84**
Offset Text 1	**101**
Offset Text 2	**115**
Offset Text 3	**116**
Offset Text 4	**105**
Offset Text 5	**110**
Offset Text 6	**103**

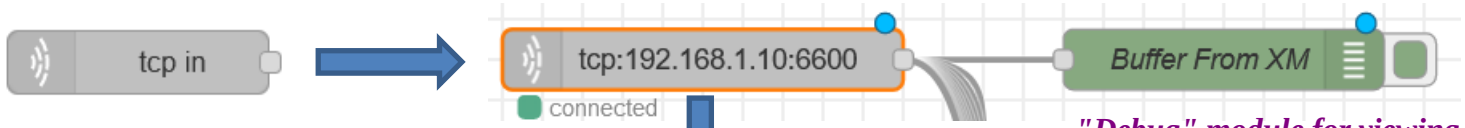
In the "text" functions and since the message received is of the type msg.payload the only thing that we must modify in each of them is the name



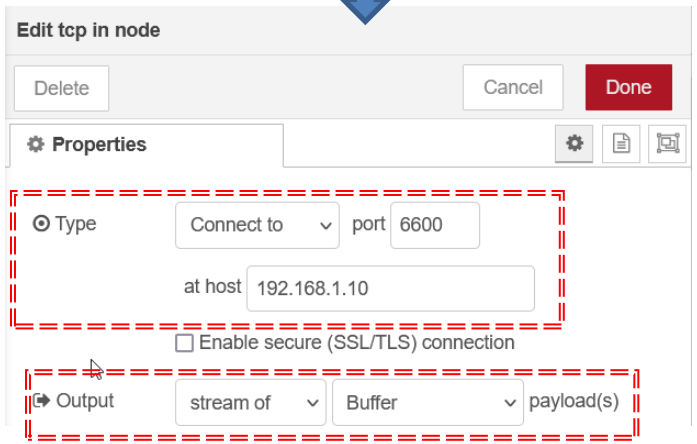
Reading data in the Node Red (ctrlX Core)

- At this point we will see how to establish communication with the XM and send and receive data using the NodeRed of the ctrlX Core

To establish communication with the XM we will use the function "tcp in"

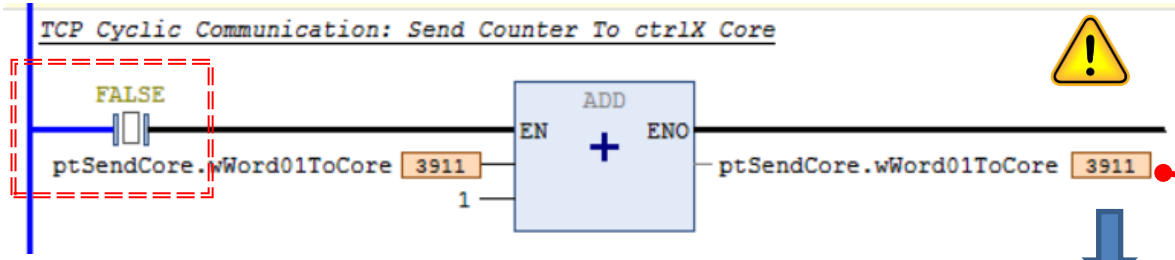


"Debug" module for viewing the received buffer

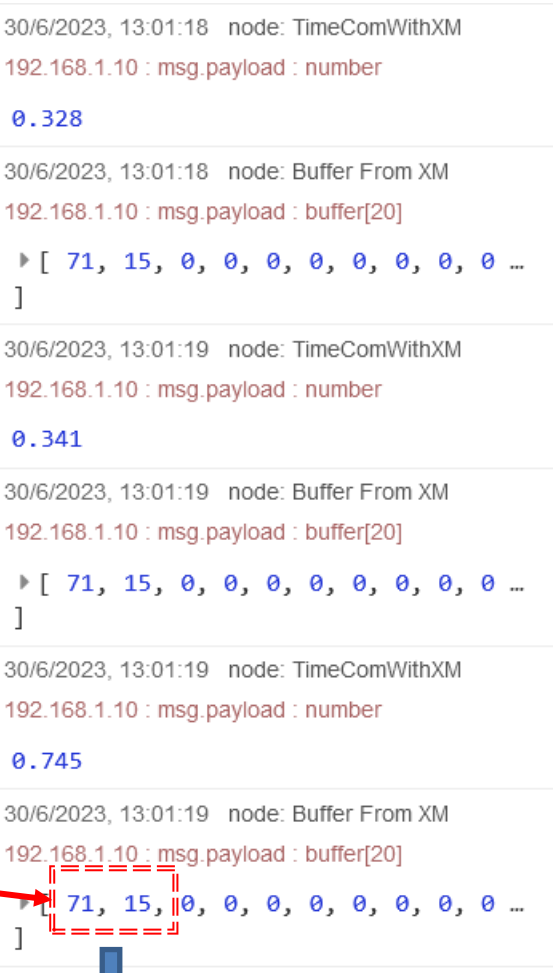


IP address of the computer with which we want to communicate

Choose the "Stream of" and "Buffer" option



The value sent by the first word of the communications from the XM arrives at the NodeRed as we have commented in buffer mode therefore in two bytes. We have placed the contact FALSE to be able to visualize the value arrived in a fixed way and to be able to see how it arrives to us.



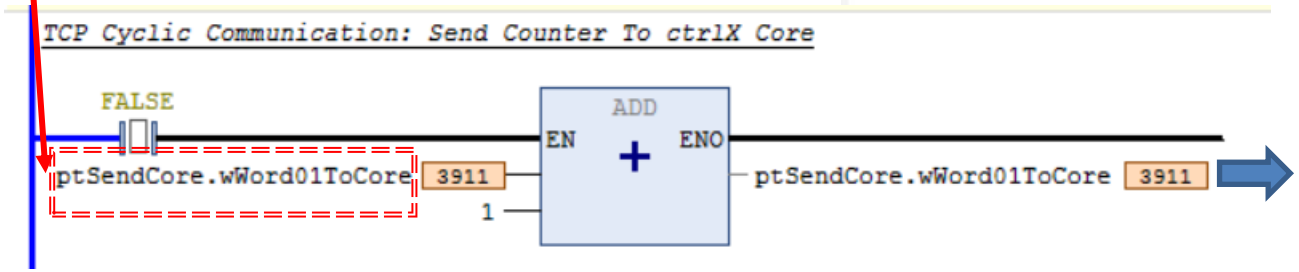
As you can see the value comes to us with the order of the buffer changed

3911	DEC	15	71
	BIN	0000 1111	0100 0111

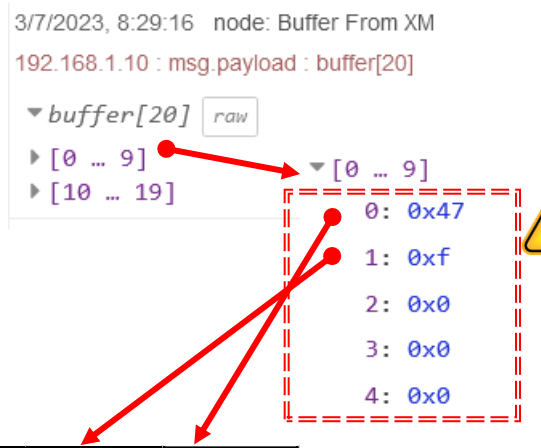
- Although we have not yet commented from the XM the structure sent is as follows. Therefore what is being sent is a total of 10 words

ptSendCore	DUT_SendToCore	
wWord01ToCore	WORD	3911
wWord02ToCore	WORD	0
wWord03ToCore	WORD	0
wWord04ToCore	WORD	0
wWord05ToCore	WORD	0
wWord06ToCore	WORD	0
wWord07ToCore	WORD	0
wWord08ToCore	WORD	0
wWord09ToCore	WORD	0
wWord10ToCore	WORD	0

```
TYPE DUT_SendToCore :
STRUCT
    wWord01ToCore: WORD;
    wWord02ToCore: WORD;
    wWord03ToCore: WORD;
    wWord04ToCore: WORD;
    wWord05ToCore: WORD;
    wWord06ToCore: WORD;
    wWord07ToCore: WORD;
    wWord08ToCore: WORD;
    wWord09ToCore: WORD;
    wWord10ToCore: WORD;
END_STRUCT
END_TYPE
```



However, and without any type of treatment, the data received in the NodeRed appear in a Buffer of 20 Bytes (10 Words)

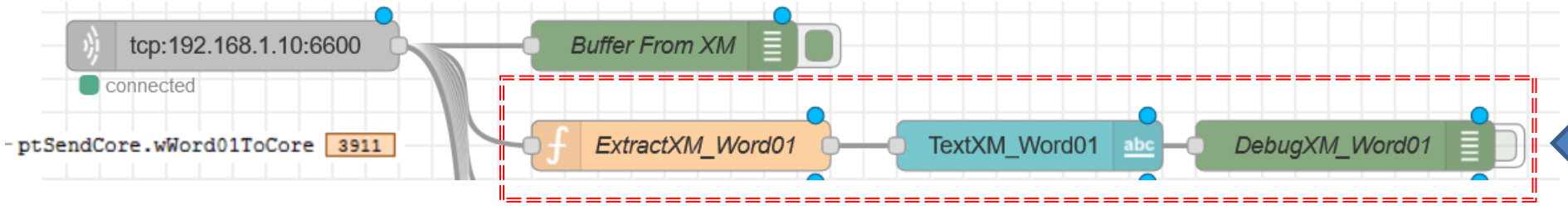


Hexadecimal value viewer

3911	DEC	15	71
	BIN	0000 1111	0100 0111

- Therefore this part comes to us with a value that we can not use directly and we must clear the value using a function as we have already seen previously

- To extract the first Word and correctly visualize the result we will use the function with the following instructions:



OK

Reading the buffer with .readInt16LE

```
var ReadBuffer = msg.payload;
var value = ReadBuffer.readUInt16LE(0);
msg.payload = value;
return msg;
```

```
3/7/2023, 8:49:35   node: DebugXM_Word01
192.168.1.10 : msg.payload : number
3911
```

The result is correct and the value we are sending from the XM appears

3911	DEC	15	71
	BIN	0000 1111	0100 0111

NOK

Reading the buffer with .readInt16BE

```
1  var ReadBuffer = msg.payload;
2  var value = ReadBuffer.readUInt16BE(0);
3  msg.payload = value;
4  return msg;
```

```
3/7/2023, 8:51:13   node: DebugXM_Word01
192.168.1.10 : msg.payload : number
18191
```

The result is NOT correct and the value appears with the order of the bytes changed

18191	DEC	71	15
	BIN	0100 0111	0000 1111

It is important to check the correct reception of the values received

- In this list we have the different methods for the conversion of the values (Read, Write, To String)

- buf.readBigInt64BE([offset])
- buf.readBigInt64LE([offset])
- buf.readBigUInt64BE([offset])
- buf.readBigUInt64LE([offset])
- buf.readDoubleBE([offset])
- buf.readDoubleLE([offset])
- buf.readFloatBE([offset])
- buf.readFloatLE([offset])
- buf.readInt8([offset])
- buf.readInt16BE([offset])
- buf.readInt16LE([offset])
- buf.readInt32BE([offset])
- buf.readInt32LE([offset])
- buf.readIntBE(offset, byteLength)
- buf.readIntLE(offset, byteLength)
- buf.readUInt8([offset])
- buf.readUInt16BE([offset])
- buf.readUInt16LE([offset])
- buf.readUInt32BE([offset])
- buf.readUInt32LE([offset])
- buf.readUIntBE(offset, byteLength)
- buf.readUIntLE(offset, byteLength)

Methods used in the previous example

- buf.write(string[, offset[, length]][, encoding])
- buf.writeBigInt64BE(value[, offset])
- buf.writeBigInt64LE(value[, offset])
- buf.writeBigUInt64BE(value[, offset])
- buf.writeBigUInt64LE(value[, offset])
- buf.writeDoubleBE(value[, offset])
- buf.writeDoubleLE(value[, offset])
- buf.writeFloatBE(value[, offset])
- buf.writeFloatLE(value[, offset])
- buf.writeInt8(value[, offset])
- buf.writeInt16BE(value[, offset])
- buf.writeInt16LE(value[, offset])
- buf.writeInt32BE(value[, offset])
- buf.writeInt32LE(value[, offset])
- buf.writeIntBE(value, offset, byteLength)
- buf.writeIntLE(value, offset, byteLength)
- buf.writeUInt8(value[, offset])
- buf.writeUInt16BE(value[, offset])
- buf.writeUInt16LE(value[, offset])
- buf.writeUInt32BE(value[, offset])
- buf.writeUInt32LE(value[, offset])
- buf.writeUIntBE(value, offset, byteLength)
- buf.writeUIntLE(value, offset, byteLength)

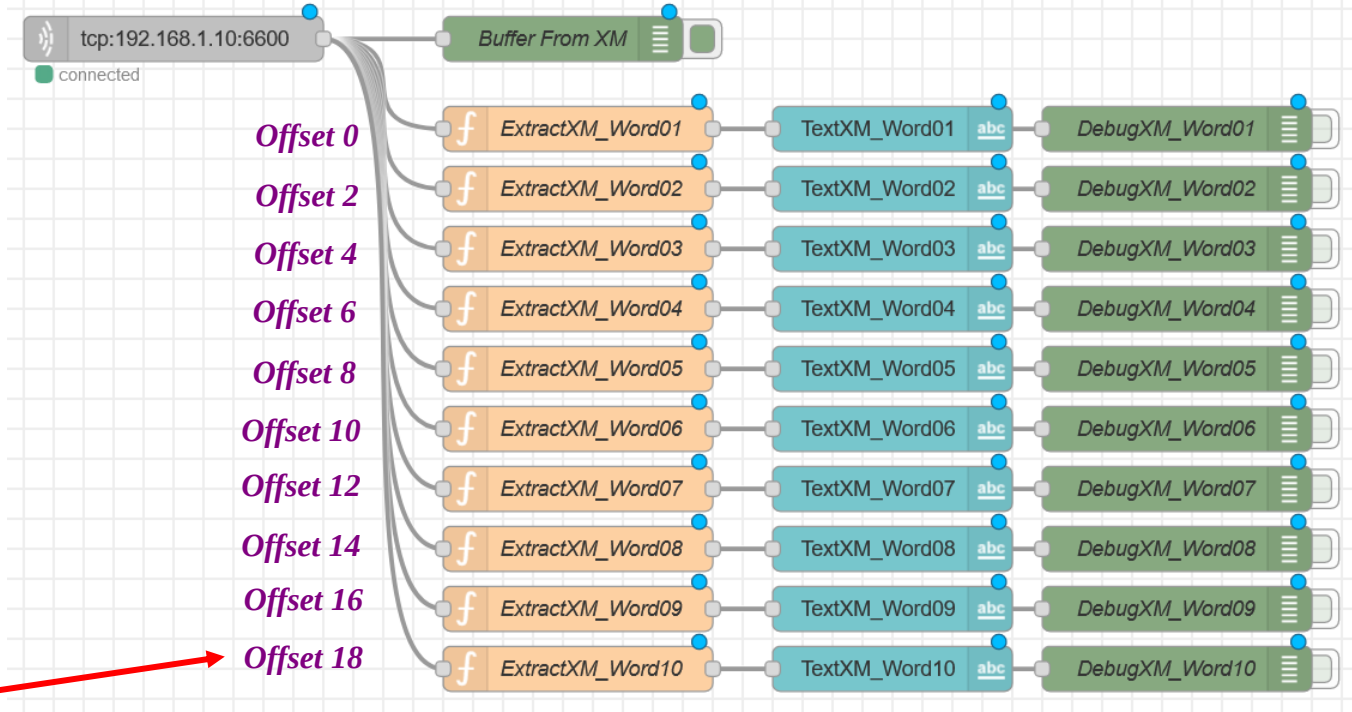
- buf.toString([encoding[, start[, end]]])

- Obviously, if we want to receive all the data we must include in the frame, the structures of functions of each of the elements

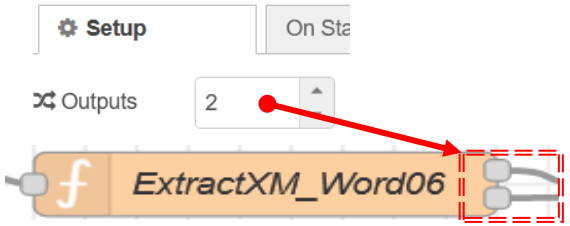
ptSendCore	DUT_SendToCore	
wWord01ToCore	WORD	3911
wWord02ToCore	WORD	2523
wWord03ToCore	WORD	3450
wWord04ToCore	WORD	4734
wWord05ToCore	WORD	5212
wWord06ToCore	WORD	6453
wWord07ToCore	WORD	7865
wWord08ToCore	WORD	8234
wWord09ToCore	WORD	9123
wWord10ToCore	WORD	10234

Values sent by XM

NodeRed with the relevant processing of the data received



In the function module we can modify the number of output points if necessary




```
var ReadBuffer = msg.payload;
var value = ReadBuffer.readUInt16LE(0);
msg.payload = value;
return msg;
```

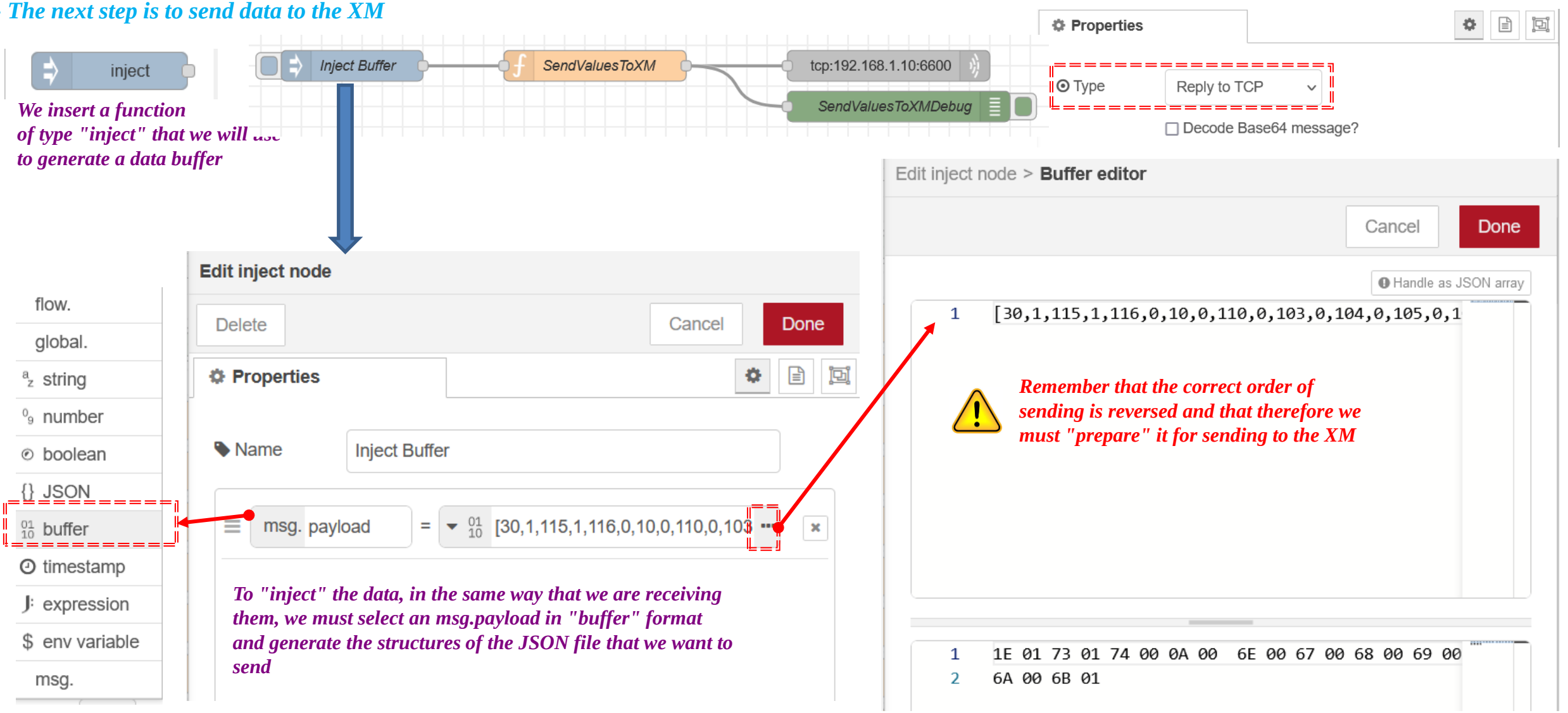
Offset number corresponding to the buffer

Values received in the Dashboard

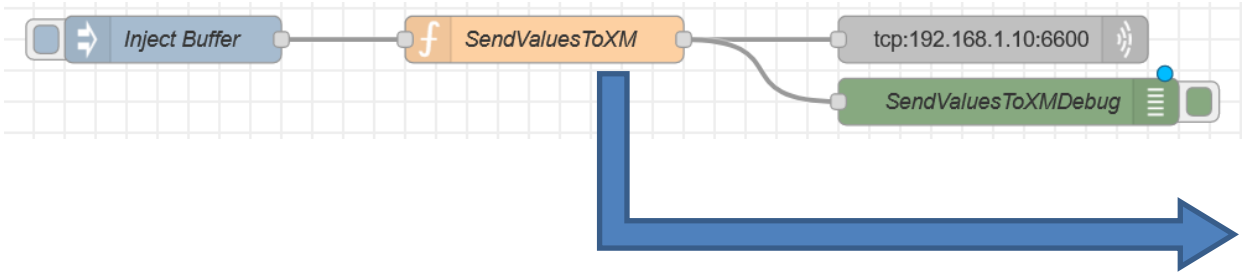
TextXM_Word01	3911
TextXM_Word02	2523
TextXM_Word03	3450
TextXM_Word04	4734
TextXM_Word05	5212
TextXM_Word06	6453
TextXM_Word07	7865
TextXM_Word08	8234
TextXM_Word09	9123
TextXM_Word10	10234

Sending data from the Node Red (ctrlX Core) to the XM

- The next step is to send data to the XM



- To convert the generated buffer data to the correct values in its visualization in the XM we must implement a new function between the buffer and the data sent to perform the conversion.



Reading the "injected" values"

The auxiliary values are extracted, in LE format (following the example of the data received), arranging them on the auxiliary variables, once the value has been extracted from the offset that the Word that we want to send must generate.

Reading of the "injected" values that we will modify for writing

The offset values read earlier are now written to the temporary buffer "testing" and this variable is the one that is sent to the XM

Variable that is sent to the XM

Edit function node

Delete

Properties

Name: SendValuesToXM

Setup | On Start | **On Message**

```
1  var test1 = msg.payload;
2  var valueWord01 = test1.readUInt16LE(0);
3  var valueWord02 = test1.readUInt16LE(2);
4  var valueWord03 = test1.readUInt16LE(4);
5  var valueWord04 = test1.readUInt16LE(6);
6  var valueWord05 = test1.readUInt16LE(8);
7  var valueWord06 = test1.readUInt16LE(10);
8  var valueWord07 = test1.readUInt16LE(12);
9  var valueWord08 = test1.readUInt16LE(14);
10 var valueWord09 = test1.readUInt16LE(16);
11 var valueWord10 = test1.readUInt16LE(18);
12 // Write Values
13 var testing = msg.payload;
14 testing.writeUInt16LE(valueWord01,0);
15 testing.writeUInt16LE(valueWord02, 2);
16 testing.writeUInt16LE(valueWord03, 4);
17 testing.writeUInt16LE(valueWord04, 6);
18 testing.writeUInt16LE(valueWord05, 8);
19 testing.writeUInt16LE(valueWord06, 10);
20 testing.writeUInt16LE(valueWord07, 12);
21 testing.writeUInt16LE(valueWord08, 14);
22 testing.writeUInt16LE(valueWord09, 16);
23 testing.writeUInt16LE(valueWord10, 18);
24 msg.payload = testing;
25 return msg;
26
```

- The treated values are sent to the XM as follows

JSON with the data sent from the "injector"

[30,1,115,1,116,0,10,0,110,0,103,0,104,0,105,0,106,0,107,1]

Offset 0

[30,1,

286	**DEC**	*1*	*30*
	BIN	0000 0001	0001 1110

```
14    testing.writeUInt16LE(valueWord01,0);
```

Data received at XM

ptReceiveCore	DUT_ReceiveFromCore	
wWord01FromCore	WORD	286
wWord02FromCore	WORD	371
wWord03FromCore	WORD	116
wWord04FromCore	WORD	10
wWord05FromCore	WORD	110
wWord06FromCore	WORD	103
wWord07FromCore	WORD	104
wWord08FromCore	WORD	105
wWord09FromCore	WORD	106
wWord10FromCore	WORD	363

115,1,

[30,1,115,1,116,0,10,0,110,0,103,0,104,0,105,0,106,0,107,1]

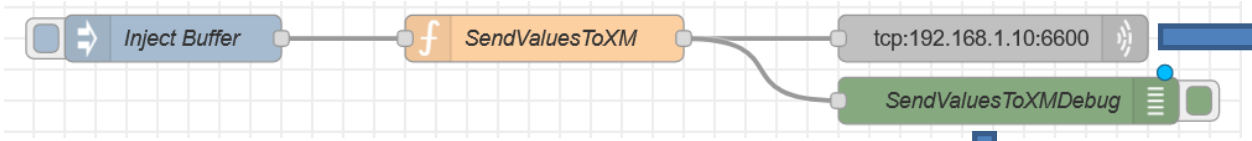
Offset 2

371	**DEC**	*1*	*115*
	BIN	0000 0001	01110011

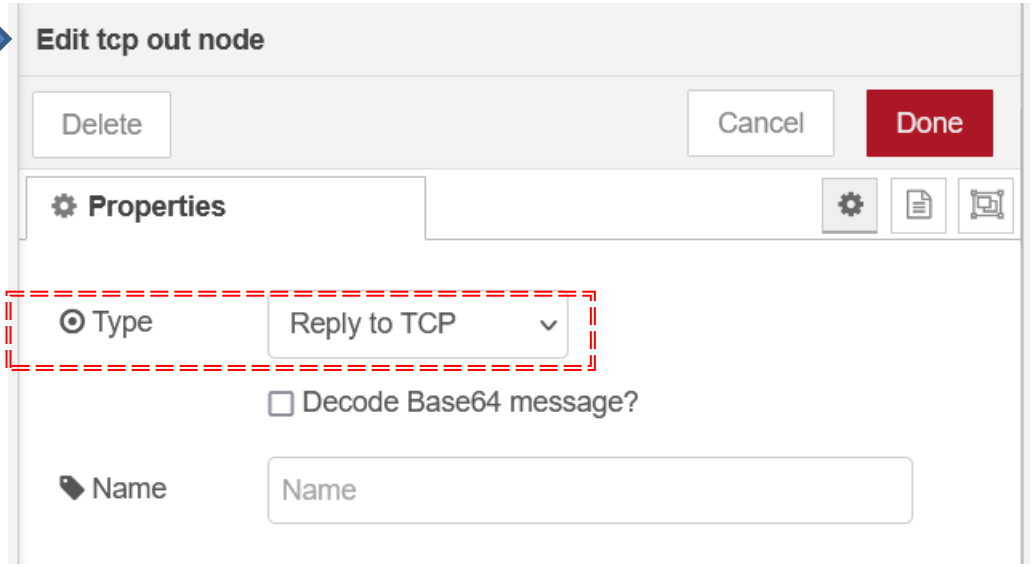
```
15    testing.writeUInt16LE(valueWord02, 2);
```

ptReceiveCore	DUT_ReceiveFromCore	
wWord01FromCore	WORD	286
wWord02FromCore	WORD	371
wWord03FromCore	WORD	116
wWord04FromCore	WORD	10
wWord05FromCore	WORD	110
wWord06FromCore	WORD	103
wWord07FromCore	WORD	104
wWord08FromCore	WORD	105
wWord09FromCore	WORD	106
wWord10FromCore	WORD	363

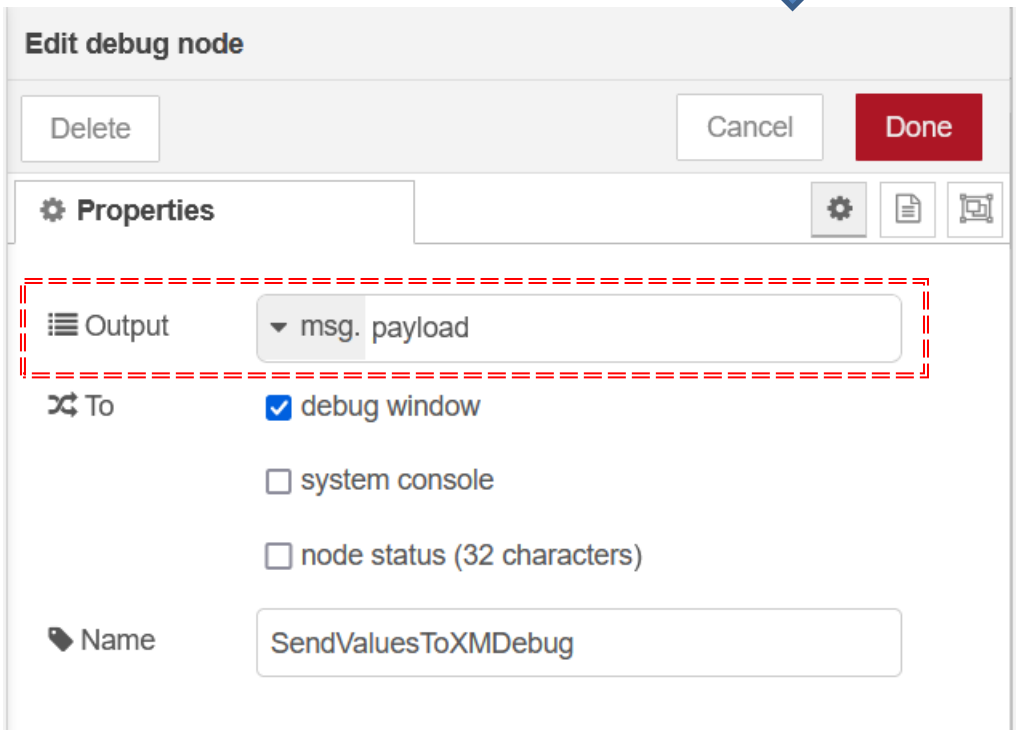
- The next two blocks allow us to send the data to the XM on the one hand and visualize them in the "Debug" on the other.



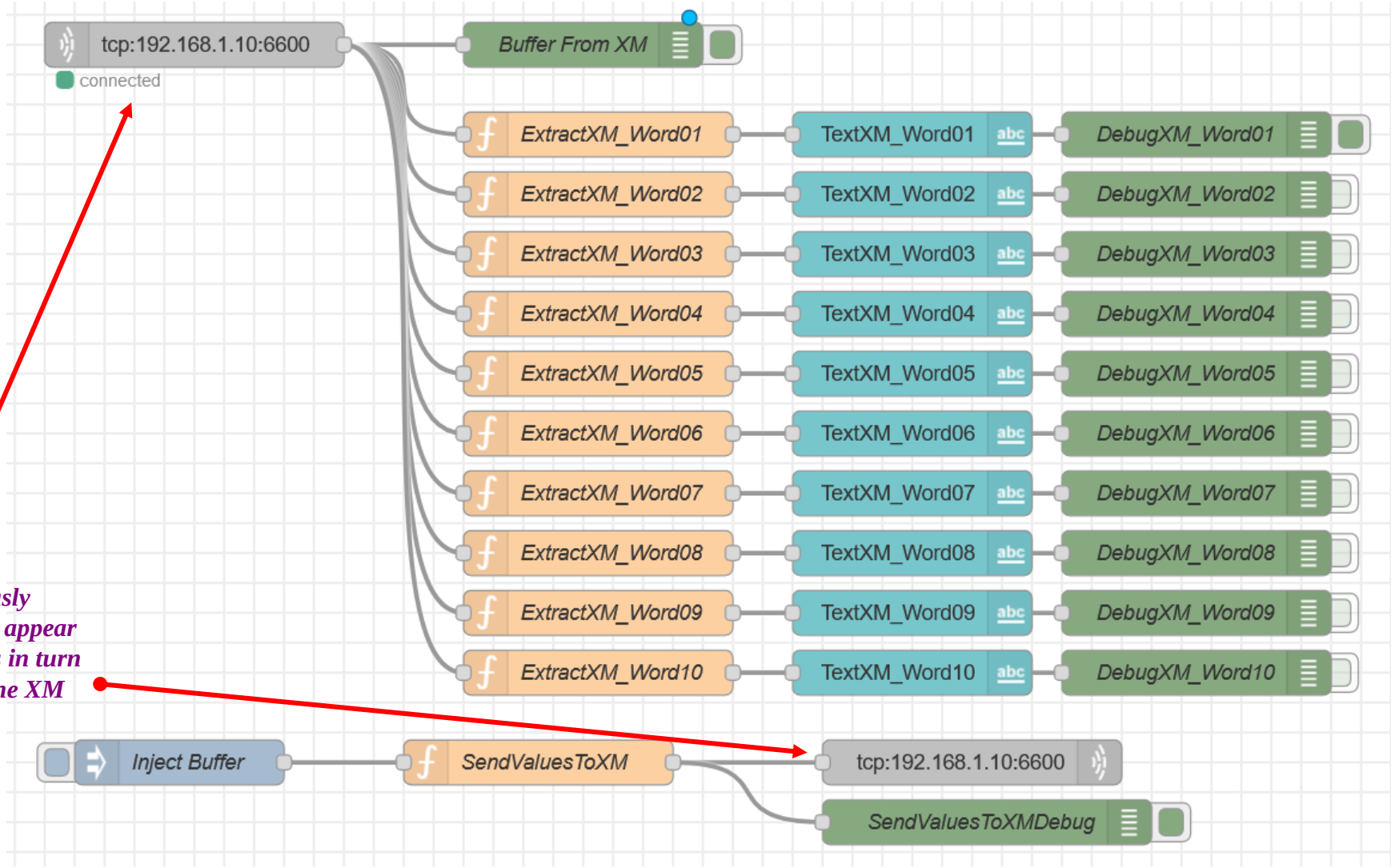
Data "Injected" into the XM



Data to Visualizer "Debug"



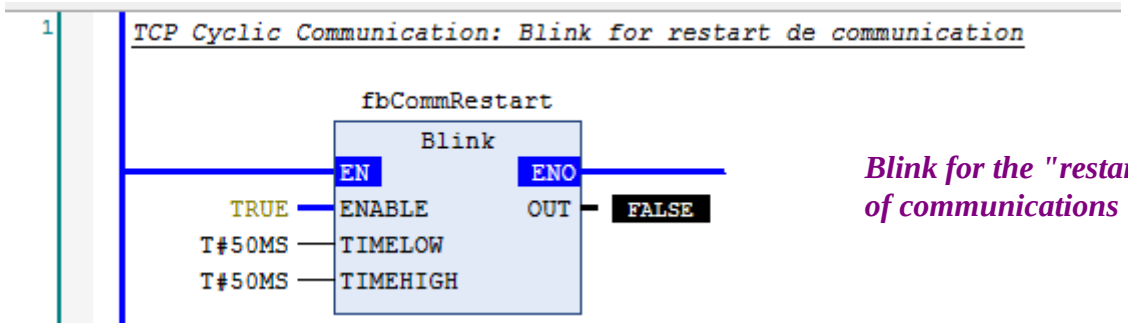
- The sending of data is activated from the element with which we establish the initial connection



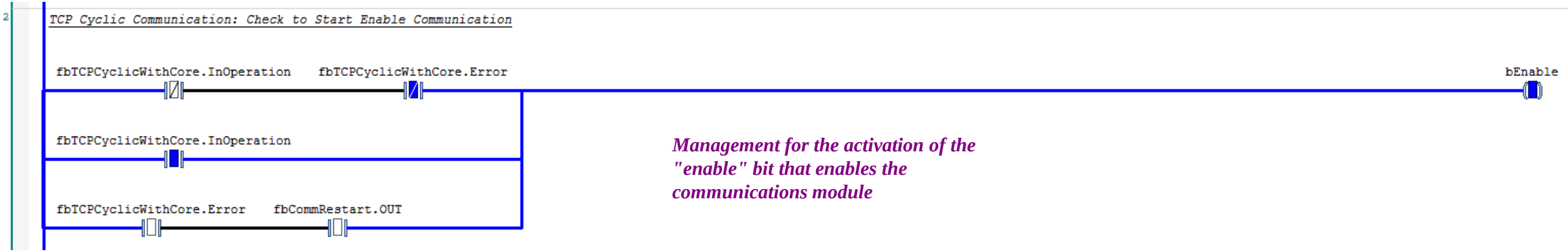
The connection previously opened and that should appear as "connected" enables in turn the sending of data to the XM

Program at XM

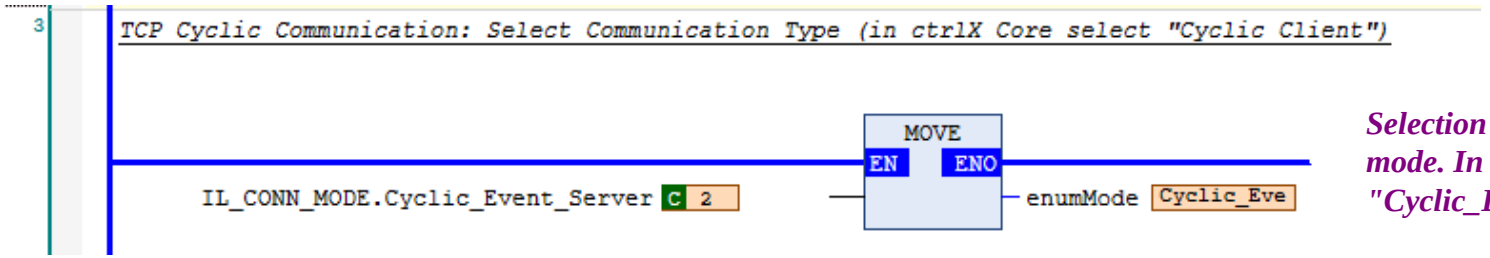
- For the example program located in the XM we will use practically the same structure used in the example of cyclic communications. Only in this case the "mode" used will be of the Cyclic Event Server type



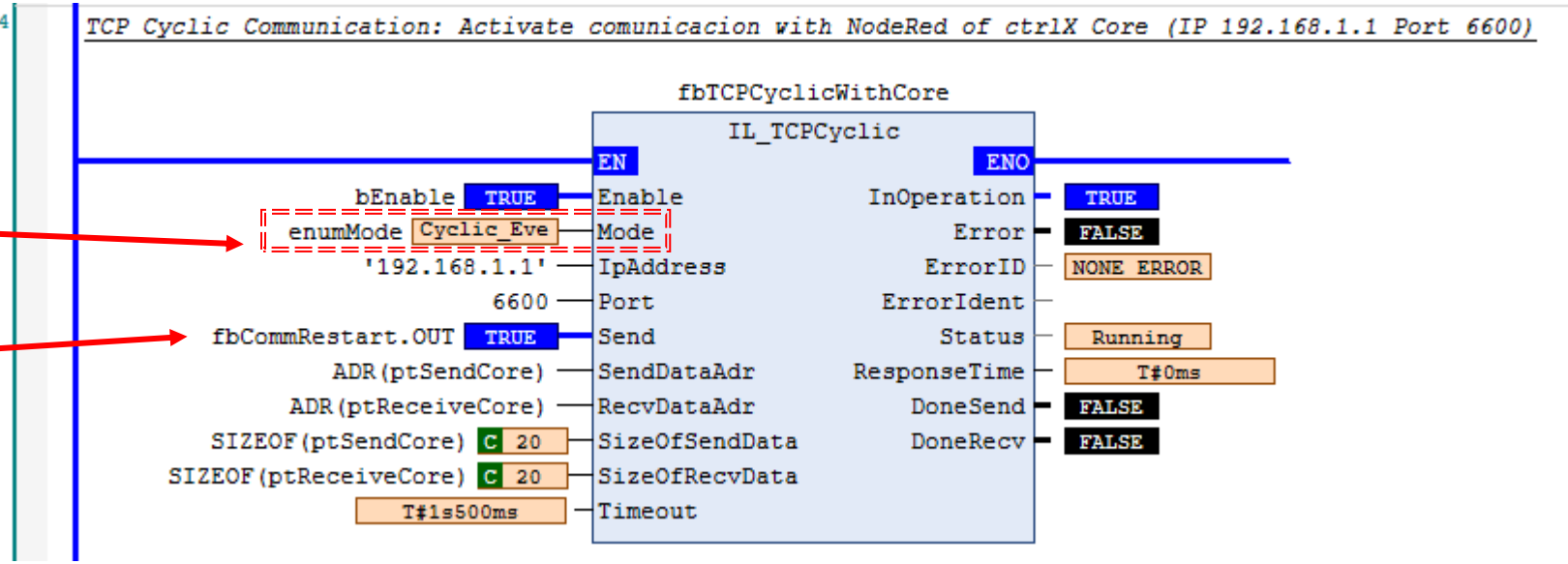
Blink for the "restart" of communications



Management for the activation of the "enable" bit that enables the communications module



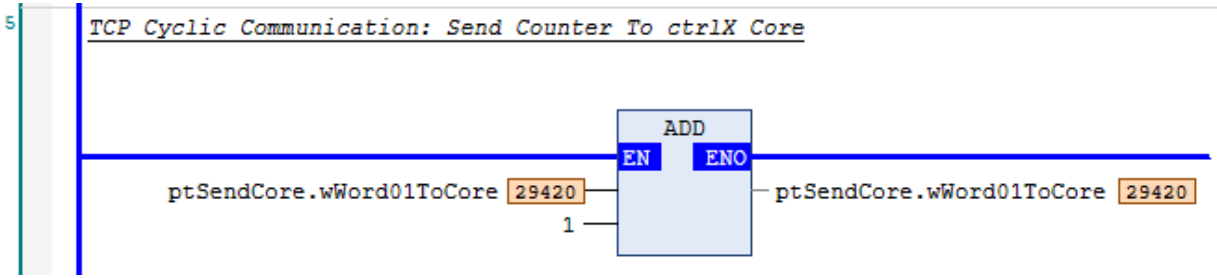
Selection of the communication mode. In the example "Cyclic_Event_Server"



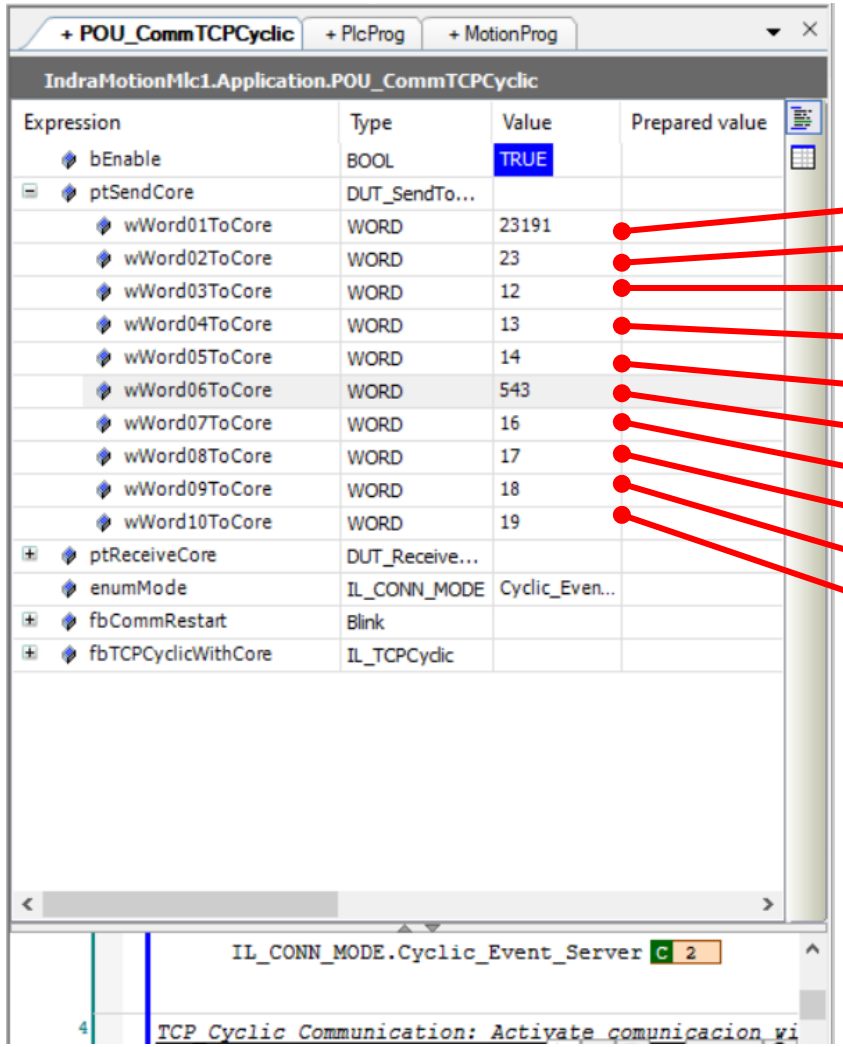
Mode

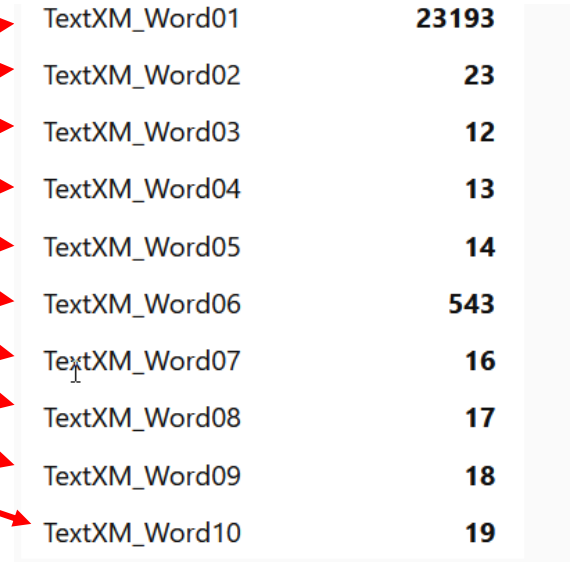
Impulse-controlled sending mode. Required in "Cyclic Event Server" mode"

Using this example we can restore the connection in any of the cases.



Value for test sent to ctrlX Core





Thank you for your attention

rexroth
A Bosch Company