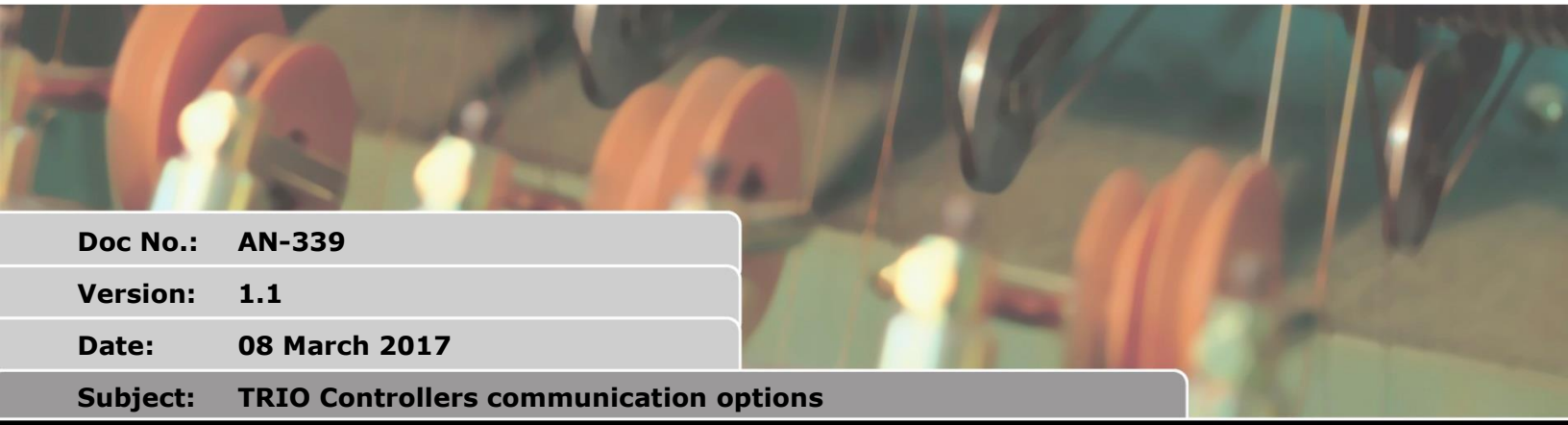
Shannon Way,
Tewkesbury,
Gloucestershire. GL20 8ND
United Kingdom
Tel: +44 (0)1684 292 333
Fax: +44 (0)1684 297 929

187 Northpointe Blvd,
Suite 105
Freeport, PA 16229
United States of America
Tel: +1 724-472-4100
Fax: +1 724-472-4101

Tomson Centre
188 Zhang Yang Rd., B1602
Pudong New Area, Shanghai,
Postal code: 200122
CHINA
Tel/Fax: +86 21 587 97659

SCMC House
16/6 Vishal Nagar
Pimpale Nilakh, Wakad, Pune
PIN 411027
INDIA
Tel: +91 827 506 5441



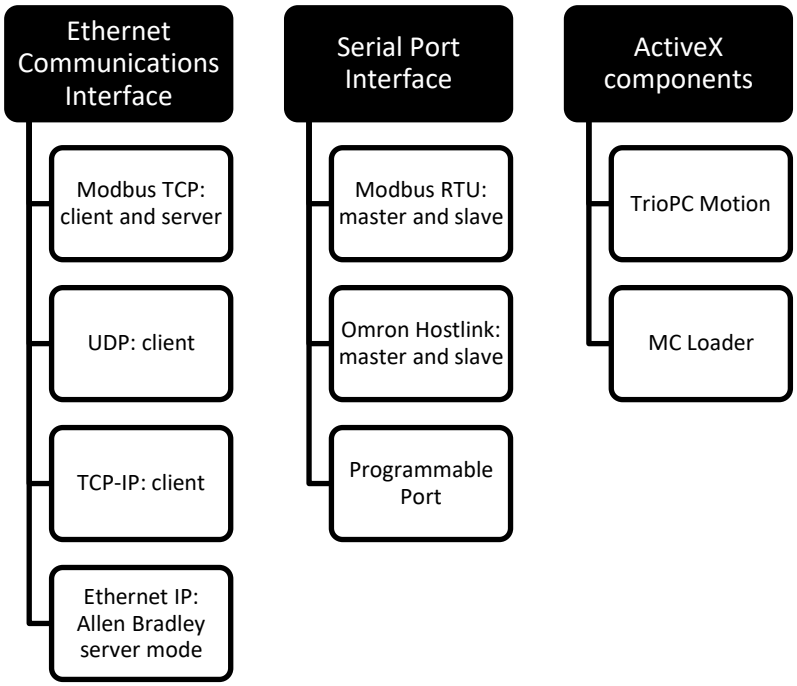
Doc No.: AN-339
Version: 1.1
Date: 08 March 2017
Subject: TRIO Controllers communication options

1. Version history

V 0.1	15 Feb 2016	First Draft
V 1.1	14 Feb 2017	Corrections to endpoint ID in section 2.1.5

2. TRIO Controllers communications options

This application note consists on a summarized guide of the whole range of communication protocols supported by Trio motion controllers. After reading this application note the user would be able to communicate with the controller through its different interfaces and built in communication protocols, as well as building a custom application layer by using BASIC code.



2.1. Ethernet Communications Interface

TrioBASIC Ethernet Command

The TrioBASIC Ethernet command is used to configure 'end-points' on the controller with which clients can communicate, using the Modbus TCP or EtherNet IP protocols. The 'Ethernet' command function index '14' is used to access the endpoints, as follows:

Ethernet - Function 14 - read/write endpoints:

Read:

```
ETHERNET(0, slot, function, endpoint_id, parameter_index)
```

Write:

```
ETHERNET(1,slot,function,endpoint_id, parameter_index, parameter_value)
```

Where endpoint_id is:

Index	Value	Description
0	Modbus TCP	
1	Ethernet IP – Assembly Object, Instance 100	Input to the controller
2	Ethernet IP – Assembly Object, Instance 101	Output from the controller

Where endpoint_id is:

Index	Value	Description
0	Address	
1	Data type	Register (hence configurable using register target described below), VR, table, digital IO, analogue IO.
2	Data format	Integer 16 bit, integer 32 bit, floating point 32, floating point 64
3	Length	
4	Class	
5	Instance	
6	Address mode	Changes the Modbus TCP address decode. 0: standard, 1: half address for 32 bit data.

If the parameter_index is data type, then the parameter_value is:

Index	Value	Description
0	Register	
1	IO input	
2	IO output	
3	VR	
4	Table	
5	Digital IO Input	

6	Digital IO Output	
7	Analogue IO Input	
8	Analogue IO Input	

If the parameter_index is data format, then the parameter_value is:

Index	Value	Description
0	Integer 16 bit	
1	Integer 32 bit	
2	Floating point 32 bit	
3	Floating point 64 bit	

NOTES

- Assembly Object class 4, instance 100 = input to the controller, 101 = output from the controller.
- You cannot set the datatype to register, since the controller needs to know whether to access the VRs or table when a get/set register request is received.

EXAMPLES

- Set endpoint 1 (Ethernet IP input) to VR Data type, and Data format to Floating point 32 bit:

```
ETHERNET(1,-1,14,1,1,3)
ETHERNET(1,-1,14,1,2,2)
```

- Change endpoint 1 (Ethernet IP input) input address to 100:

```
ETHERNET(1,-1,14,1,0,100)
```

2.1.1. Modbus TCP: client and server

By using *MODBUS* and *ETHERNET* TRIO Basic commands, you can configure the TRIO controllers as a client (Master) or server (Slave) Modbus node. The following BASIC code illustrates how to do a Modbus TPC client against another node working as a server.

To use Trio controllers as servers, the *ETHERNET* command should be used to choose between either 16 bits (default) or 32 bits registers, and also if the registers will point to VRs or TABLE. Furthermore, the halving addressing feature could be used in case of choosing 32 bits.

The halving addressing is set by default in the MC2X generation, it cannot be modified.

The example below consists on a client application running on a Trio controller and opening a connection against another Trio controller doing the Modbus TCP server side.

EXAMPLE:

```
currenthandle=100
nrerrors=101
currenterror=102
connectionstatus=103

'Virtual IO
opencon=104
closecon=105
stopprogram=106
```

```
allerrors=107
writecoils=108
readcoils=109

slotnumber=-1

MODBUS( 1, slotnumber, VR(currenthandle)) 'close a previous connection is opened
MODBUS($12, slotnumber, VR(currenthandle)) 'Reset the entire error log

main:
    PRINT "SELECT OPERATION TO BE PERFORMED:"
    GOSUB printmenu

    WAIT UNTIL IN(104,111)>0
    'Open Conection
    IF IN(opencon) THEN
        OP(opencon,OFF)
        GOSUB checkconectionstatus
        IF VR(connectionstatus) =0 THEN
            IF MODBUS( 0, slotnumber,192,168,0,190,502,currenthandle) THEN
                PRINT "Connection successfully open, now select the next
operation"
                GOSUB blankline
                GOTO main
            ELSE
                GOSUB errorhandler
                GOTO main
            ENDIF

        ELSE
            PRINT "The connection is already open"
            GOSUB blankline
            GOTO main
        ENDIF

        'Write coils
    ELSEIF IN(writecoils) THEN
        OP(writecoils,OFF)

        IF MODBUS( 3, slotnumber, VR(currenthandle),$0F,100,3,203) THEN
            PRINT "Coils written:", VR(203).2, VR(203).1, VR(203).0
            GOSUB blankline
                GOTO main
            ELSE
                GOSUB errorhandler
                GOTO main
            ENDIF

            'Read coils
        ELSEIF IN(readcoils) THEN
            OP(readcoils,OFF)

            IF MODBUS( 3, slotnumber, VR(currenthandle),1,100,3,200) THEN
                PRINT "Single coil", VR(200).2, VR(200).1, VR(200).0
                GOSUB blankline
                GOTO main
            ELSE
                GOSUB errorhandler
                GOTO main
```

```
            ENDIF

        'Close conection
        ELSEIF IN(closecon) THEN
        OP(closecon,OFF)

        IF MODBUS( 1, slotnumber, VR(currenthandle) ) THEN
                    PRINT "Connection succesfully closed, now select the next
operation"
                    GOSUB blankline
                    GOTO main
                ELSE
                    GOSUB errorhandler
                    GOTO main
                ENDIF

            'Print all errors
            ELSEIF IN(allerrors) THEN
            OP(allerrors,OFF)
        GOSUB getallerrors
        GOTO main
        'Stop Program
    ELSEIF IN(stopprogram) THEN
        OP(stopprogram,OFF)
        HALT
    ENDIF

getallerrors:
    MODBUS($11,slotnumber, VR(currenthandle), 101) 'Get the number of errors for
the given handle
FOR x=1 TO nrerrors
    MODBUS($10, slotnumber, VR(currenthandle),x-1 ,102) 'Store the error in
VR(102)
    GOSUB printerror
NEXT x
RETURN

errorhandler:
MODBUS($10, slotnumber, VR(currenthandle),0 ,102) 'Store the error in VR(102)
GOSUB printerror
RETURN

printerror:
IF HEX(VR(currenterror)) = "1" THEN
    PRINT "MB_ERRCODE_ILLEGAL_FUNCTION"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "2" THEN
    PRINT "MB_ERRCODE_ILLEGAL_DATA_ADDRESS"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "3" THEN
    PRINT "MB_ERRCODE_ILLEGAL_DATA_VALUE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "4" THEN
    PRINT "MB_ERRCODE_SLAVE_DEVICE_FAILURE"
    GOSUB blankline
    RETURN
```

```
ELSEIF HEX(VR(currenterror)) = "5" THEN
    PRINT "MB_ERRCODE_ACKNOWLEDGE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "6" THEN
    PRINT "MB_ERRCODE_SLAVE_DEVICE_BUSY"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "7" THEN
    PRINT "MB_ERRCODE_NEGATIVE_ACKNOWLEDGE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "8" THEN
    PRINT "MB_ERRCODE_MEMORY_PARITY_ERROR"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "10" THEN
    PRINT "MB_ERRCODE_INVALID_PROTOCOL_IDENTIFIER"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "11" THEN
    PRINT "MB_ERRCODE_INVALID_MSG_LEN"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "12" THEN
    PRINT "MB_ERRCODE_CNX_CLOSED"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "13" THEN
    PRINT "MB_ERRCODE_BUSY"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "14" THEN
    PRINT "MB_ERRCODE_TIMEOUT"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "15" THEN
    PRINT "MB_ERRCODE_INVALID_RESPONSE_VALUE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "16" THEN
    PRINT "MB_ERRCODE_INVALID_REQUEST_PARAMETERS"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "17" THEN
    PRINT "MB_ERRCODE_INSUFFICIENT_RESOURCES"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "18" THEN
    PRINT "MB_ERRCODE_INVALID_CONFIG_MODE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "19" THEN
    PRINT "MB_ERRCODE_FAILED_OPEN_CNX"
    GOSUB blankline
RETURN
ELSEIF HEX(VR(currenterror)) = "1A" THEN
    PRINT "MB_ERRCODE_INVALID_PARAMETER"
    GOSUB blankline
```

```
RETURN
ELSEIF HEX(VR(currenterror)) = "1B" THEN
    PRINT "MB_ERRCODE_INVALID_CMD_HANDLE"
    GOSUB blankline
RETURN
ELSEIF HEX(VR(currenterror)) = "1C" THEN
    PRINT "MB_ERRCODE_INVALID_CNX_HANDLE"
    GOSUB blankline
RETURN
ELSEIF HEX(VR(currenterror)) = "1D" THEN
    PRINT "MB_ERRCODE_INVALID_CHECKSUM"
    GOSUB blankline
    RETURN
ELSE
    PRINT "UNKNOWN ERROR: "; VR(currenterror)
    GOSUB blankline
    RETURN
ENDIF

printmenu:
PRINT "Press INPUT 104 to Open connection"
PRINT "Press INPUT 105 to Close connection"
PRINT "Press INPUT 106 to Stop the program"
PRINT "Press INPUT 107 to Print all errors"
PRINT "Press INPUT 108 to write multiple Coils"
PRINT "Press INPUT 109 to read multiple Coils"
GOSUB blankline
RETURN

checkconectionstatus:
IF MODBUS( 2, slotnumber, VR(currenthandle), connectionstatus) THEN
    RETURN
ELSE
    GOSUB errorhandler
    GOTO main
ENDIF

blankline:
PRINT ""
RETURN
```

2.1.2. UDP: client

The Trio UDP client implementation is used for the transport of ASCII and Binary data bytes between devices over the Internet Protocol (IP), in a manner similar to the use of a serial port. So that a user application program can send and receive strings of data using OPEN, CLOSE, PRINT, GET, CHANNEL_READ and CHANNEL_WRITE commands.

The UDP packets can be sent from and read into strings, arrays of integer values or using VRs.

Some constraints to take into account are:

- UDP is supported only by the ARM based controllers (not MIPS/MC464)

The following example illustrates how TRIO controllers can establish a UDP communication, send and receive packets.

The example below consists on a client application running on a Trio controller and opening a connection against another Trio controller working as a server. As the server is a trio controller and port 23 is being opened, characters are sent to the terminal (channel 0) of the server controller, this

is the reason why an “echo” response is obtained in the client controller.

EXAMPLE:

```
'Program Data
comms = 5
chan = 20
DIM str_output AS STRING(64)
DIM str_input AS STRING(64)

' begin
WA(120)
GOSUB close_port
WA(120)
GOSUB open_port
str_output = "?version" + CHR(13)
GOSUB write_string
WA(100)
GOSUB rx_scheduler
str_output = "?serial_number" + CHR(13)
GOSUB write_string_chwrite
WA(100)
GOSUB rx_scheduler_chread

STOP

' ********************************************************
' API
'

'Program to close the connection in channel 21
close_port:
CLOSE#chan
RETURN

'Program to open the communication with the server through channel 21
open_port:
OPEN#chan AS "dgram:192.168.0.190(23)" FOR READ_WRITE
RETURN

'Program to send a string to the server controller
write_string:
PRINT#chan, str_output;
RETURN

'Program to send a string to the server controller
write_string_chwrite:
CHANNEL_WRITE(chan, str_output)
RETURN

'Program to read all data received from the server controller (storing ascii code
received in a variable), and write
'to channel 5
rx_scheduler:
WHILE KEY#chan
    GET#chan,value
    PRINT#comms,CHR(value);
WEND
RETURN
```

```
'Program to read all data received from the server controller (storing ascii code
received in a variable), and write
'to channel 5
rx_scheduler_chread:
PRINT CHANNEL_READ(chan,str_input)
PRINT#comms,str_input;
RETURN

'Program to read all data received from the server controller (storing ascii code
received in a VR), and write
'to channel 5
rx_scheduler_vr:
WHILE KEY#chan
    GET#chan,VR(100)
    PRINT#comms,CHR(VR(100));
WEND
RETURN
```

2.1.3. TCP-IP: client

The Trio TCP client implementation is used for the transport of ASCII and Binary data bytes between devices over the Internet Protocol (IP), in a manner similar to the use of a serial port. So that a user application program can send and receive strings of data using OPEN, CLOSE, PRINT, GET CHANNEL_READ and CHANNEL_WRITE commands. It basically works in the same manner as the UDP client but it is safer since the delivery of packages are guaranteed. It is also slower compared to UDP, but more suitable for certain applications.

The TCP packets can be sent from and read into strings, arrays of integer values or using VRs.

Some constraints to take into account are:

- UDP is supported only by the ARM based controllers (not MIPS/MC464)

The following example illustrates how TRIO controllers can establish a TCP communication, send and receive packets.

The example below consists on a client application running on a Trio controller and opening a connection against another Trio controller working as a server. As the server is a trio controller and port 23 is being opened, characters are sent to the terminal (channel 0) of the server controller, this is the reason why an “echo” response is obtained in the client controller.

EXAMPLE:

```
'Program Data
comms = 5
chan = 21
DIM str_output AS STRING(64)
DIM str_input AS STRING(64)

' begin
WA(120)
GOSUB close_port
WA(120)
GOSUB open_port
str_output = "?version" + CHR(13)
GOSUB write_string
WA(100)
GOSUB rx_scheduler
str_output = "?serial_number" + CHR(13)
GOSUB write_string_chwrite
WA(100)
```

```
GOSUB rx_scheduler_chread

STOP

' ********************************************************
' API
'

'Program to close the connection in channel 21
close_port:
CLOSE#chan
RETURN

'Program to open the communication with the server through channel 21
open_port:
OPEN#chan AS "tcp:192.168.0.190(23)" FOR READ_WRITE
RETURN

'Program to send a string to the server controller
write_string:
PRINT#chan, str_output;
RETURN

'Program to send a string to the server controller
write_string_chwrite:
CHANNEL_WRITE(chan, str_output)
RETURN

'Program to read all data received from the server controller (storing ascii code
received in a variable), and write
'to channel 5
rx_scheduler:
WHILE KEY#chan
    GET#chan,value
    PRINT#comms,CHR(value);
WEND
RETURN

'Program to read all data received from the server controller (storing ascii code
received in a variable), and write
'to channel 5
rx_scheduler_chread:
PRINT CHANNEL_READ(chan,str_input)
PRINT#comms,str_input;
RETURN

'Program to read all data received from the server controller (storing ascii code
receiver in a VR), and write
'to channel 5
rx_scheduler_vr:
WHILE KEY#chan
    GET#chan,VR(100)
    PRINT#comms,CHR(VR(100));
WEND
RETURN
```

2.1.4. Ethernet IP: Allen Bradley server mode

Ethernet/IP is an application layer protocol designed for use in process control and industrial automation. It is built on the Common Industrial Protocol (CIP) - as used in Devicenet and Controlnet. However, Ethernet IP uses the standard TCP/IP stack, and exchanges information over the Ethernet physical layer. The specification is managed by the ODVA.

The Trio Motion Coordinator is an Ethernet IP server which waits, listening for an incoming message from a client. It will then act on this message – writing to controller memory, returning controller memory values, or setting up a connection for exchanging fast cyclic data.

The Motion Coordinator supports the two fundamental message types defined by this protocol. The first is the explicit protocol, used for acyclic one-time request-response type messages, in which the message defines its own meaning and contains all necessary data to perform the requested operation. The second is the implicit protocol, defining regular cyclic messages which contain only a connection identifier and the message data. These messages are used to transport high frequency IO type data. The meaning of the data is defined when the connection between the client and server end-points is established.

2.2. Serial Port Interface

2.2.1. Modbus RTU: master and slave

Modbus RTU is a widely known protocol running over the serial communication interface (RS-232, RS-485). Trio controllers can run this protocol as a slave or master by properly setting the SETCOM command. This command will allow the user to configure the serial port of the controller, setting parameters like the baud rate, parity, choosing the physical interface, the communications protocol, etc.

If the Trio controller is configured to work as a slave, using SETCOM and ADDRESS will be enough. ADDRESS will just set the address of the node on the Modbus network.

```
ADDRESS=1
SETCOM(38400,8,1,2,1,4,0,3)
```

If the Trio controller is configured to work as a Modbus master, besides configuring the serial port to work as a Modbus master (**mode 11**), the MODBUS command must be used to execute the different Modbus functions. As shown in the example below, the MODBUS functions works exactly in the same way as in Modbus TCP (see Modbus TCP: client and server).

As in previous examples, the following consists on a client application running on a Trio controller and opening a connection against another Trio controller doing the Modbus RTU slave side.

EXAMPLE

```
currenthandle=100
nrerrors=101
currenterror=102
connectionstatus=103

'Virtual IO
opencon=104
closecon=105
stopprogram=106
allerrors=107
writecoils=108
readcoils=109

slotnumber=-1

'set the rs232 serial PORT 1 parameters, AND start MODBUS rtu client
PROTOCOL
SETCOM(38400,8,1,2,1,11,0,10) 'Modbus RS232 2 wires mode 11
```

```
MODBUS( 1, slotnumber, VR(currenthandle)) 'close a previous connection is
opened
MODBUS($12, slotnumber, VR(currenthandle)) 'Reset the entire error log

main:
PRINT "SELECT OPERATION TO BE PERFORMED:"
GOSUB printmenu

WAIT UNTIL IN(104,111)>0
'Open Conection
IF IN(opencon) THEN
    OP(opencon,OFF)
    GOSUB checkconectionstatus
    IF VR(connectionstatus) =0 THEN
        IF MODBUS( 0, slotnumber,-1,0,currenthandle) THEN
            PRINT "Connection succesfully open, now select the next
operation", VR(currenthandle)
            GOSUB blankline
            GOTO main
        ELSE
            GOSUB errorhandler
            GOTO main
        ENDIF

    ELSE
        PRINT "The connection is already open"
        GOSUB blankline
        GOTO main
    ENDIF

    'Write coils
ELSEIF IN(writecoils) THEN
    OP(writecoils,OFF)
    IF MODBUS( 3, slotnumber, VR(currenthandle),$0F,1,100,3,203) THEN
        PRINT "Coils written:", VR(203).2, VR(203).1, VR(203).0
        GOSUB blankline
        GOTO main
    ELSE
        GOSUB errorhandler
        GOTO main
    ENDIF

    'Read coils
ELSEIF IN(readcoils) THEN
    OP(readcoils,OFF)

    IF MODBUS(3,slotnumber,VR(currenthandle),1,1,100,3,200) THEN
        PRINT "Single coil", VR(200).2, VR(200).1, VR(200).0
        GOSUB blankline
        GOTO main
    ELSE
        GOSUB errorhandler
        GOTO main
    ENDIF

    'Close conection
ELSEIF IN(closecon) THEN
    OP(closecon,OFF)
```

```
    IF MODBUS( 1, slotnumber, VR(currenthandle)) THEN
        PRINT "Connection succesfully closed, now select the next operation"
        GOSUB blankline
        GOTO main
    ELSE
        GOSUB errorhandler
        GOTO main
    ENDIF

    'Print all errors
ELSEIF IN(allerrors) THEN
    OP(allerrors,OFF)
    GOSUB getallerrors
    GOTO main
    'Stop Program
ELSEIF IN(stopprogram) THEN
    OP(stopprogram,OFF)
    HALT
ENDIF

getallerrors:
MODBUS($11,slotnumber,VR(currenthandle), 101) 'Get the number of errors for
the given handle
FOR x=1 TO nrerrors
    MODBUS($10, slotnumber, VR(currenthandle),x-1 ,102) 'Store the error in
VR(102)
    GOSUB printerror
NEXT x
RETURN

errorhandler:
MODBUS($10, slotnumber, VR(currenthandle),0 ,102) 'Store the error in
VR(102)
GOSUB printerror
RETURN

printerror:
IF HEX(VR(currenterror)) = "1" THEN
    PRINT "MB_ERRCODE_ILLEGAL_FUNCTION"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "2" THEN
    PRINT "MB_ERRCODE_ILLEGAL_DATA_ADDRESS"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "3" THEN
    PRINT "MB_ERRCODE_ILLEGAL_DATA_VALUE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "4" THEN
    PRINT "MB_ERRCODE_SLAVE_DEVICE_FAILURE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "5" THEN
    PRINT "MB_ERRCODE_ACKNOWLEDGE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "6" THEN
```

```
    PRINT "MB_ERRCODE_SLAVE_DEVICE_BUSY"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "7" THEN
    PRINT "MB_ERRCODE_NEGATIVE_ACKNOWLEDGE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "8" THEN
    PRINT "MB_ERRCODE_MEMORY_PARITY_ERROR"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "10" THEN
    PRINT "MB_ERRCODE_INVALID_PROTOCOL_IDENTIFIER"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "11" THEN
    PRINT "MB_ERRCODE_INVALID_MSG_LEN"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "12" THEN
    PRINT "MB_ERRCODE_CNX_CLOSED"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "13" THEN
    PRINT "MB_ERRCODE_BUSY"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "14" THEN
    PRINT "MB_ERRCODE_TIMEOUT"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "15" THEN
    PRINT "MB_ERRCODE_INVALID_RESPONSE_VALUE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "16" THEN
    PRINT "MB_ERRCODE_INVALID_REQUEST_PARAMETERS"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "17" THEN
    PRINT "MB_ERRCODE_INSUFFICIENT_RESOURCES"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "18" THEN
    PRINT "MB_ERRCODE_INVALID_CONFIG_MODE"
    GOSUB blankline
    RETURN
ELSEIF HEX(VR(currenterror)) = "19" THEN
    PRINT "MB_ERRCODE_FAILED_OPEN_CNX"
    GOSUB blankline
RETURN
ELSEIF HEX(VR(currenterror)) = "1A" THEN
    PRINT "MB_ERRCODE_INVALID_PARAMETER"
    GOSUB blankline
RETURN
ELSEIF HEX(VR(currenterror)) = "1B" THEN
    PRINT "MB_ERRCODE_INVALID_CMD_HANDLE"
    GOSUB blankline
RETURN
```

```
ELSEIF HEX(VR(currenterror)) = "1C" THEN
    PRINT "MB_ERRCODE_INVALID_CNX_HANDLE"
    GOSUB blankline
RETURN
ELSEIF HEX(VR(currenterror)) = "1D" THEN
    PRINT "MB_ERRCODE_INVALID_CHECKSUM"
    GOSUB blankline
    RETURN
ELSE
    PRINT "UNKNOWN ERROR: "; VR(currenterror)
    GOSUB blankline
    RETURN
ENDIF

printmenu:
PRINT "Press INPUT 104 to Open connection"
PRINT "Press INPUT 105 to Close connection"
PRINT "Press INPUT 106 to Stop the program"
PRINT "Press INPUT 107 to Print all errors"
PRINT "Press INPUT 108 to write multiple Coils"
PRINT "Press INPUT 109 to read multiple Coils"
GOSUB blankline
RETURN

checkconectionstatus:
IF MODBUS( 2, slotnumber, VR(currenthandle), connectionstatus) THEN
    RETURN
ELSE
    GOSUB errorhandler
    GOTO main
ENDIF

blankline:
PRINT ""
RETURN
```

2.2.2. Omron Hostlink: master and slave

Hostlink is a serial communications protocol used by many devices. This protocol is a built-in feature of most of TRIO controllers and it allows the Hostlink Master to exchange data with the VR and TABLE of the Slaves.

The following pieces of code shows how to stablish the communication and make the basic data exchange operations with the TRIO controllers.

2.2.2.1. Slave TRIO configuration

Each slave Motion Coordinator must be set to run on power-up. The first slave should be set as HLS_NODE=0, the second one as HLS_NODE=1, third as HLS_NODE=2, etc.

```
'''SLAVE CONFIGURATION'''
baud_rate = 9600
data_bits = 7
stop_bits = 2
parity = 2 ' Even parity
option =5 ' Host Link Slave
SETCOM(baud_rate,data_bits,stop_bits,parity,2,option)
' Host Link Slave settings
'-----------------------------
```

```
HLS_NODE=1
HLS_MODEL=48
```

2.2.2.2. Master TRIO configuration

The master is configured with a single SETCOM command like this:

```
'set up host link master for PORT 2
SETCOM(9600,7,2,2,2,6)
```

After configuration there are various Hostlink Master Commands that can be used in the BASIC. For example: HLM_READ and HLM_WRITE.

2.2.2.3. Read 4 VRs from slave at node 1

4 x 16 bit words are read from VR(2) to VR(5) in slave 1 and put the data into VR(100) to VR(103) in the master.

```
' HLM_READ(port, node, pc_area, pc_offset, length, mc_area, mc_offset)
' source ADDRESS: VR(2)
' amount of data: 4 words
' destination ADDRESS: VR(100)
HLM_READ(2,1 ,PLC_IR,2,4,MC_VR,100)
```

2.2.2.4. Read 4 VRs from slave at node 5

4 x 16 bit words are read from TABLE(22) to TABLE(25) in slave 1 and put the data into VR(1200) to VR(1203) in the master.

```
' source ADDRESS: TABLE(22)
' amount of data: 4 words
' destination ADDRESS: VR(1200)
HLM_READ(2,5,PLC_DM,22,4,MC_VR,1200)
```

2.2.2.5. Write 10 values from TABLE in master to TABLE in slave node 0

10 x 16 bit words from TABLE(18) to TABLE(27) in the master are written to the slave 0, into TABLE(14) to TABLE(23).

```
' Source address: TABLE(18)
' Amount of data: 10 words
' Destination address: TABLE(14)
HLM_WRITE(2,0,PLC_DM,14,10,MC_TABLE,18)
```

2.2.2.6. Error checking routine

The Hostlink protocol has some error checking. HLM_STATUS returns the value of the error number if there is an error, or 0 if there is no error. The following example is a typical error routine.

```
' Source address: Table(18)
' Amount of data: 2 words
' Destination address: TABLE(14)
HLM_WRITE(2,0,PLC_DM,14,2,MC_TABLE,18)
IF HLM_STATUS<>0 THEN GOSUB report_status

report_status:
' decodes the HLM_STATUS word and prints to terminal
hst = HLM_STATUS
IF hst AND $200 THEN
    PRINT #term,"HLM: Command not Recognised"
ENDIF
```

```
IF hst AND $100 THEN
    PRINT #term,"HLM: Timeout error"
ENDIF
hst=hst AND $ff
IF hst=0 THEN
    PRINT #term,"HLM: Normal completion"
ELSEIF hst=1 THEN
    PRINT #term,"HLM: Not executable in RUN mode"
ELSEIF hst=13 THEN
    PRINT #term,"HLM: FCS error"
ELSEIF hst=14 THEN
    PRINT #term,"HLM: Format error"
ELSEIF hst=15 THEN
    PRINT #term,"HLM: Entry number data error"
ELSEIF hst=18 THEN
    PRINT #term,"HLM: Frame length error"
ELSEIF hst=19 THEN
    PRINT #term,"HLM: Not executable"
ELSEIF hst=21 THEN
    PRINT #term,"HLM: CPU error"
ELSE
    PRINT #term,"HLM: Unknown error"
ENDIF
RETURN
```

2.2.3. Programmable Port

The programmable port utility is used for the transport of ASCII and Binary data bytes between devices over a serial communication link. So that a user application program can send and receive strings of data using OPEN, CLOSE, PRINT, GET CHANNEL_READ and CHANNEL_WRITE commands.

The packets can be sent from and read into strings, arrays of integer values or using VRs.

The following example illustrates how TRIO controllers can establish a serial communication over an RS-232 connection, send and receive packets.

The example below consists on a point to point serial connection between two Trio controllers. One of the controllers send and receive an echo response from the other.

EXAMPLE

Controller 1

```
DIM output_str AS STRING(100)
DIM input_str AS STRING(100)
chann = 1

output_str = "This is the string to be sent and read back through the serial
port"

PRINT #chann, output_str

WA(30)

IF KEY #chann THEN
    CHANNEL_READ(chann,input_str)
    PRINT #5, input_str
ENDIF
```

Controller 2

```
chann = 1
WHILE TRUE
    IF KEY #chann THEN
        GET #chann, ascii_code
        PRINT#chann, CHR(ascii_code);
    ENDIF
WEND
```

2.3. ActiveX Components

2.3.1. TrioPC Motion

TrioPC Motion is an ActiveX component that can be used to create windows client applications in any of the windows programming languages (VB.NET, C#, Visual C/C++, Delphi) as well as other applications supporting ActiveX (OCX) such as LabView.

This component contains wide function library, allowing the user to perform most of the motion commands as well as managing variables among other things. The connection can be done via a USB or Ethernet link.

For more information about installation and examples, you could consult our support section on our website. The application note AN-215 might be also really useful.

2.3.2. MC Loader

MC Loader is an ActiveX component that can be used to create windows client applications in any of the windows programming languages (VB.NET, C#, Visual C/C++, Delphi) as well as other applications supporting ActiveX (OCX) such as LabView.

This component can load projects (produced with *Motion* Perfect 2 or *Motion* Perfect v3) and programs onto a Trio Motion Coordinator. Communication with the Motion Coordinator can be via Serial link, USB, Ethernet or PCI depending on the Motion Coordinator.