
Industrial Automation Using Raw Port Drivers

Tech Note 10



Abstract:

This document describes the use of Raw Port drivers to send and receive ASCII data to compatible Red Lion Controls (RLC) products.

Products:

Red Lion Controls CR1000 Human Machine Interface (HMI), CR3000 HMI, Data Station Plus (Excludes LE), G3 HMI, G3 Kadet HMI, Graphite[®] Controller, Graphite HMI, Modular Controller (Excludes LE and V2), and ProductVity Station[™]

Use Case:

Raw Port drivers are used when Crimson does not have a driver for a particular device, or when ASCII data needs to be transferred.

Required Software:

Crimson[®] 3.0 or 3.1

Required Operating System:

Microsoft Windows 2000, or above

Introduction

The port number is used as an argument in all of the functions associated with sending and receiving data through a raw port. In order to identify the port number, start Crimson and click on the Communications section of the Navigation pane, as shown in Figure 1. After the Communications - RS-485 Comms Port popup appears, refer to the upper right of the Editing Pane or the lower left of the Crimson window in the status bar; *Port Number 2* in this example.

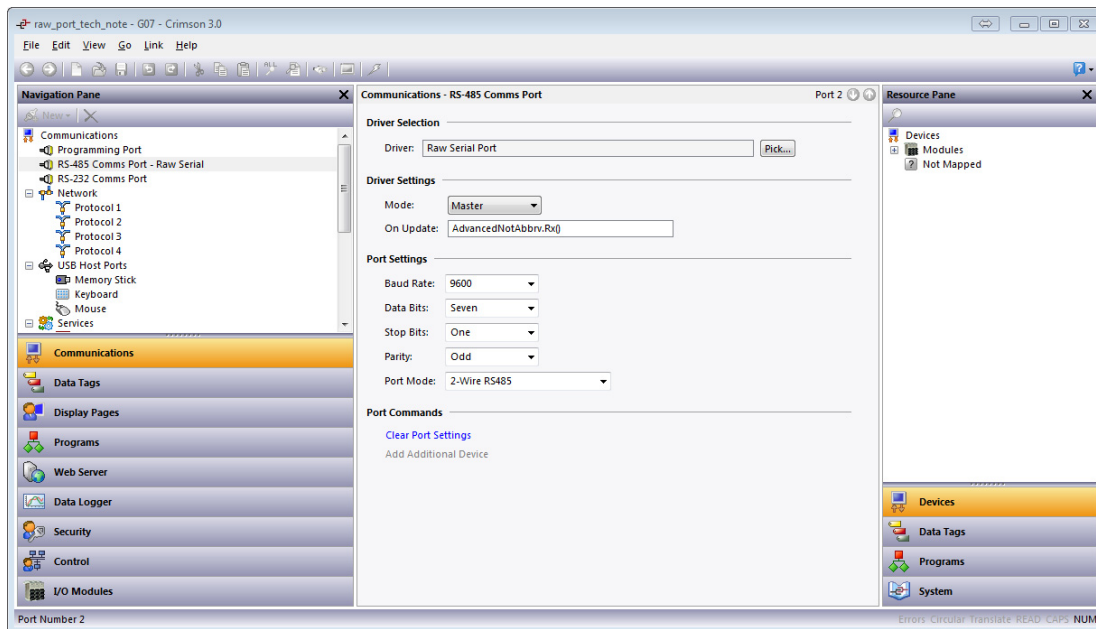


Figure 1.

NOTE: Crimson support for Raw Port drivers requires Crimson 3.0 or higher and is only available to Windows 2000 or above users. Please update your version of Crimson to the latest, available online at: www.redlion.net.

When a Raw Driver is assigned to a port, the On Update parameter is used to define an action that occurs during every communications scan; this is the location that calls a program that acts as a receive routine.

The majority of the code examples that follow were sourced from the corresponding Crimson 3.0 database file: http://files.redlion.net/filedepot_download/1378/7105.

The first three (3) sections use the RLC Instrument protocol for their examples, specifically communicating to a PAXI. See page 25 of the PAXI datasheet/manual for the protocol specification: http://files.redlion.net/filedepot_download/213/5263.

Basic

Transmitting Fixed Strings

Transmitting a fixed string requires only one function and one line of code:

```
Send N01TA$
```

```
PortPrint(2, "N01TA$");
```

```
Send 123<CR><LF> (123<carriage return><linefeed>)
```

```
PortPrint(2, "123\r\n");
```

Display Entire Received Message

To read a string from the port's buffer, use the PortInput function:

```
input = PortInput(2, 0, 13, 500, 0);
```

This will return any ASCII characters that precede a <CR>; if 500ms goes by without a <CR> then it will return a null string. In order to hold the last valid (non-null) received string, an IF statement will be required to verify what was received before populating a tag:

```
if(input != "") {  
    Basic.Response = input;  
}
```

Here is the code for a full receive routine:

```
cstring input;  
input = PortInput(2, 0, 13, 500, 0);  
if(input != "") {  
    Basic.Response = input;  
}
```

Intermediate

Transmitting Strings with Embedded Variables

In order to embed variables into a string transmission, the variable must be made into a string (if it is not already) and then concatenated with other strings before transmission:

```
PortPrint(2, "N1VM" + IntToText(Intermediate.Write, 10, 5) + "$");
```

or,

```
output = "N1VM" + IntToText(Intermediate.Write, 10, 5) + "$";  
PortPrint(2, output);
```

or,

```
output = "N1VM";  
output += IntToText(Intermediate.Write, 10, 5);  
output += "$";  
PortPrint(2, output);
```

Parsing Received String

As in the Basic section, the buffer needs to be read and checked for a non-null string. Depending on the data format, different parsing techniques may be required, including:

```
input = PortInput(2, 0, 13, 500, 0);  
if(input != "") {
```

1.The first X characters are a portion of the transmission that needs to be separated:

```
Node = Left(input, 2);
```

2.The next portion of the transmission is X characters after the first space:

```
space = Find(input, 32, 0);  
Mnemonic = Mid(input, space+1, 3);
```

3.The final portion is the last 12 characters, but is padded with spaces:

```
temp = Strip(Right(input, 12), ' ');  
Value = TextToInt(temp, 10);
```

Here is the code for a full receive routine:

```
cstring input;  
cstring temp;  
  
int space;  
  
input = PortInput(2, 0, 13, 500, 0);  
if(input != "") {  
    Intermediate.Node = TextToInt(Left(input, 2), 10);  
    space = Find(input, 32, 0);  
    Intermediate.Mnemonic = Mid(input, space+1, 3);  
    temp = Strip(Right(input, 12), ' ');  
    Intermediate.Value = TextToInt(temp, 10);  
}
```

Advanced

Cyclically Polling Parameters

If more than a single transmission is required to gather all of the needed data, a state machine can be used to cycle through the different transmissions:

```
switch(State) {  
    // poll counter A  
    case 0:  
        output = "N01TA$";  
        break;  
    // poll rate  
    case 1:  
        output = "N01TD$";  
        break;  
    // poll setpoint 1  
    case 2:  
        output = "N01TM$";  
        break;  
}
```

Interrupt Polling to Issue Write

If there is a need to write, the polling may need to wait while the write request is serviced. This can be accomplished by using an IF statement before entering the state machine, such as:

```
// need to write?  
if(NeedToWrite) {  
    NeedToWrite = 0;  
    // generate output  
    output = "N1VM";  
    output += IntToText(Write, 10, 5);  
    output += "$";  
  
    // clear the port's buffer  
    ClearRx(port);  
  
    // send command  
    PortPrint(2, output);  
    // exit program  
    return;  
}
```

Receive: Assuming Data Received Is What Was Requested

If the incoming data is assumed to be correct, or if there are no identifiers in the response, a similar state machine can be used:

```
// populate correct tag based on what was requested
switch(State) {

    // counter A
    case 0:
        CountA = value;
        break;
    // rate
    case 1:
        Rate = value;
        break;
    // setpoint 1
    case 2:
        Setpoint1 = value;
        State = -1;
        break;
}

// increment state to poll the next value
State ++;
```

Receive: Response Includes Identification of Data

If the incoming data includes identifiers in the response, they can be used to determine which tag to populate:

```
// parse response
// Node Address is the first 2 characters
node = TextToInt(Left(input, 2), 10);

// the Mnemonic follows a space after the node
// find the space
space = Find(input, 32, 0); // could also use find(input, ' ', 0)
mnemonic = Mid(input, space+1, 3);

// the data is the last 12 bytes, padded with spaces, which need to be removed
temp = Strip(Right(input, 12), ' '); // could also use strip(left(input, 12), 32);
value = TextToInt(temp, 10);

// populate correct tag based on what was received
if(mnemonic == "CTA") {
    CountA = value;
}
else if(mnemonic == "RTE") {
    Rate = value;
}
else if(mnemonic == "SP1") {
    Setpoint1 = value;
}
```

Receive: Byte-by-Byte

Receiving byte-by-byte is required for ASCII protocols, but can also be used for troubleshooting or if the format of the response is unknown. The received data can be loaded into an array, or a string can be built from it; the following code does both:

```
// this Rx routine will allow you to display all of the bytes received in a transmission
// declare locals
int i;
int j;
int in = PortRead(3, 100);

// loop while there is data in the buffer, populating the array
while(in != -1) {
    Array[i++] = in;
    in = PortRead(3, 100);
}
// if something was received, build a string
if(i > 0) {
    String = "";
    for(j = 0; j <= i; j++){
        String += Array[j];
    }
}
```


Raw UDP

Transmitting

Unlike the Raw TCP/IP drivers, the only configurable property is the port that it listens on. Part of the transmission includes both the target IP address, target UDP port, and byte count. Refer to the following:

```
// Target IP = 192.168.50.11
PortWrite(4, 192);
PortWrite(4, 168);
PortWrite(4, 50);
PortWrite(4, 11);

// Target Port = 0x1234
PortWrite(4, 0x12);
PortWrite(4, 0x34);

// Byte Count = 11
PortWrite(4, 0);
PortWrite(4, 11);

// UDP Data (0123456789<CR>)
PortPrint(4, "0123456789\r");
```

Receiving

Unlike the other raw port drivers, the UDP buffer contains more than just response data. It uses an “R” to identify the beginning of a response, followed by the source IP address, the port it came in on, and the byte count. Refer to the following:

```
// Check for Frame
if( PortRead(4, 0) == 'R' ) {

    // Read Source IP
    RxIP  = PortRead(4, 0) << 24;
    RxIP |= PortRead(4, 0) << 16;
    RxIP |= PortRead(4, 0) << 8;
    RxIP |= PortRead(4, 0) << 0;

    // Read Source Port
    RxPort = PortRead(4, 0) << 8;
    RxPort |= PortRead(4, 0) << 0;

    // Read Length
    RxCount = PortRead(4, 0) << 8;
    RxCount |= PortRead(4, 0) << 0;

    // Read Data
    Fill(RxData[0], 0, 40);

    int n;

    for( n = 0; n < RxCount; n++ ) {
        RxData[n] = PortRead(4, n);
    }

    // Increment Sequence
    RxSeq++;
}
```

Clearing Receive Buffer

Reading from the buffer removes the data from it. A while loop can be used to clear out old data, as follows:

```
// loop until buffer is empty  
while(PortRead(2, 0) != -1) {}
```

Disclaimer

It is the customer's responsibility to review the advice provided herein and its applicability to the system. Red Lion makes no representation about specific knowledge of the customer's system or the specific performance of the system. Red Lion is not responsible for any damage to equipment or connected systems. The use of this document is at your own risk. Red Lion standard product warranty applies.

Red Lion Technical Support

If you have any questions or trouble contact Red Lion Technical Support by emailing support@redlion.net or calling 1-877-432-9908.

For more information: <http://www.redlion.net/support/policies-statements/warranty-statement>