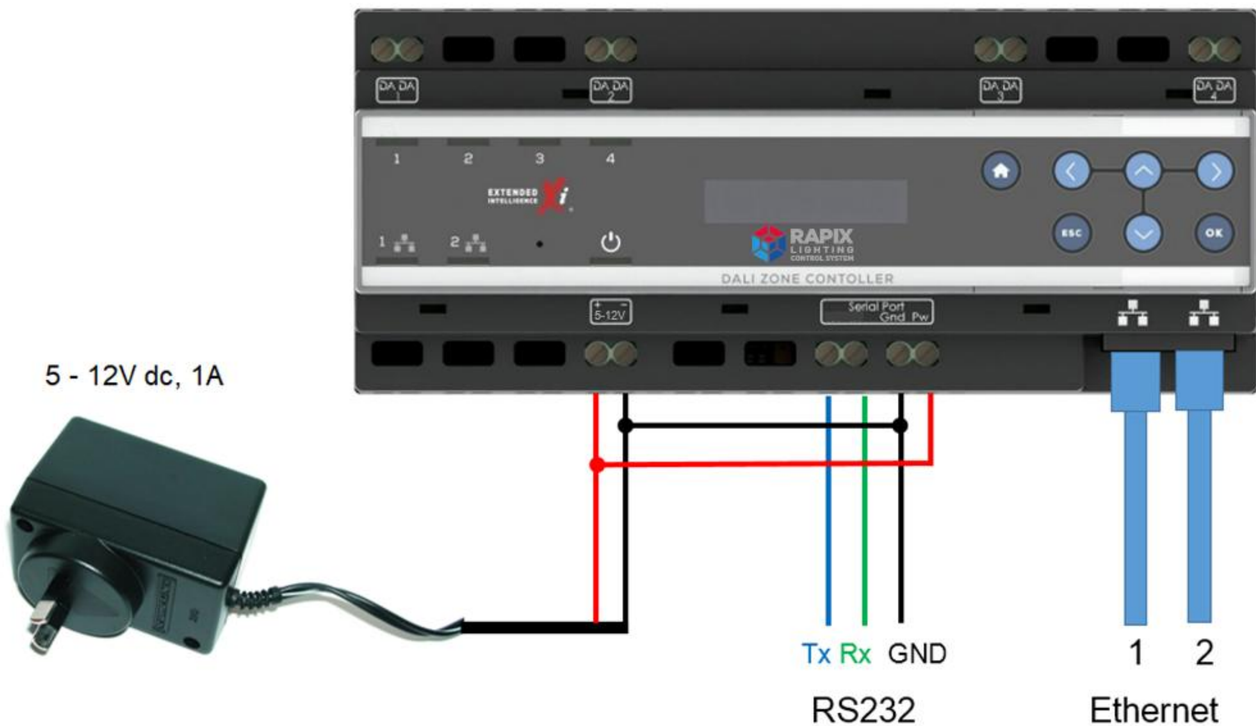
Introduction

This application note describes the use of the Zone Controller serial ports for general-purpose control and monitoring.

The Zone Controller has an RS-232 serial port that can be connected to another device as shown in the wiring diagrams below.



Zone Controller Connections

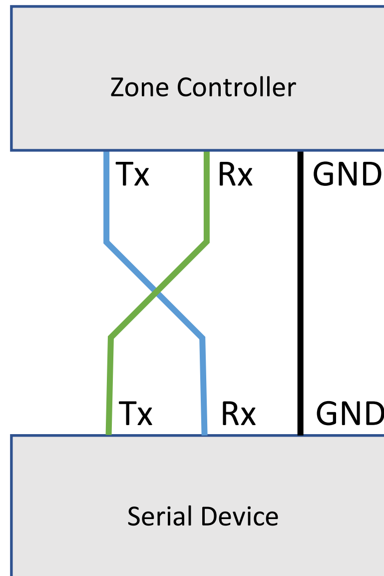
Communication through the serial port is done using logic code, as described in the following sections.

For details of using RAPIX Logic, please refer to the *RAPIX Logic Programming Guide*[1]

[1] This guide can be opened from the RAPIX Integrator software LOGIC tab.

Physical Connection

The Zone Controller is equipped with a 3-wire RS-232 physical interface.



RS-232 Connections between the Zone Controller and Serial Device

Zone Controller Serial Port Options

The Zone Controller serial port can be configured with a variety of options to suit most serial devices. The options are set in the logic code when opening the serial port.

Feature	Options
Speed	Standard serial port speeds: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600 and 115200 bits / sec.
Data Bits	8
Stop Bits	1, 1.5 or 2
Parity	None, Even, Off, Mark, Space
Flow Control (aka "Handshaking")	None or XON/XOFF

Serial Port Options

Cable Length

The RS-232 standard states that for reliable communications at 9600 bits/sec, the maximum cable length is 15m, or the cable length equal to a capacitance of 2500pF. If using a low-capacitance cable, then the length can be greater than 15m. For example, if CAT-5 cable is used with a capacitance of 52 pF/m, the maximum allowed cable length is 48m.

The cable length can be extended by using a lower speed. A rough guide can be obtained by reference to the table shown below:

Speed	Maximum Cable Length (m)
2400	60
4800	30
9600	15
19200	7.6
38400	3.7
56000	2.6

RS-232 Standard maximum recommended cable length for various speeds

For reliable operation, use the lowest speed and use the shortest, best quality cable available.

Communication Protocols

All devices that use serial communications have a "protocol" which defines the format of the messages sent between the two devices. It is important that the protocol is clearly understood before trying to write the serial port logic code. The important factors are:

- Low-level communication settings (e.g. speed, parity and stop bits);
- Message structure (e.g. binary or ASCII data, message terminators);
- Message content and format; and
- High-level communication architecture (e.g. whether messages are acknowledged).

There are too many variations of communication protocols to cover in this document, so just the most common will be described.

Message Separation

The data received from a serial port will be a stream of bytes. It is usually necessary to be able to determine where one message ends and the next starts. The main methods used for this are:

- **Fixed Length**: for simple messages, the length can be always the same. To read these messages, it is simply a matter of reading the expected number of bytes from the serial port, then process the message.
- **Message Length Field**: for more complex messages, some protocols include the length of the message in the data. To read these messages, it is necessary to keep reading from the serial port until the length field has been read. Once the length of the message is known, the rest of the message data can then be read and processed.
- **Terminator Characters**: a specific character (byte value) can be used to separate the messages. This character is not allowed to appear within a message. To read these messages, it is necessary to keep reading from the serial port until the terminator character is read. The message can then be processed.

Message Data

Once the bytes of a message have been read, they need to be interpreted. The most common message data formats are:

- **ASCII**: In ASCII messages, each byte in the message represents an ASCII character. These messages are generally human-readable.
- **UTF-8**: For messages that contain non-English characters, UTF-8 encoding is often used instead of ASCII. These messages are generally used for human-readable information.
- **Binary**: In binary data, each byte of the message is just a binary number. Binary messages are not generally human-readable. Binary message structures generally don't use terminator characters.

ASCII with CR/LF terminator

The most common message structure is to have ASCII encoded data with a Carriage Return and/or Line Feed character as a terminator (to separate the messages).

The RAPIX Logic has methods designed to simplify reading and writing these types of messages. These types of messages will be used in the examples for simplicity.

The terminator characters cannot generally be written directly in a C# string, so "escape sequences" are used to allow them to be represented. The most common ones are shown in the table below.

Character	Byte value	C# Escape Sequence
Carriage Return	13	\r
Line Feed/New Line	10	\n
Tab	9	\t
Null	0	\0

Common Message Terminator Characters

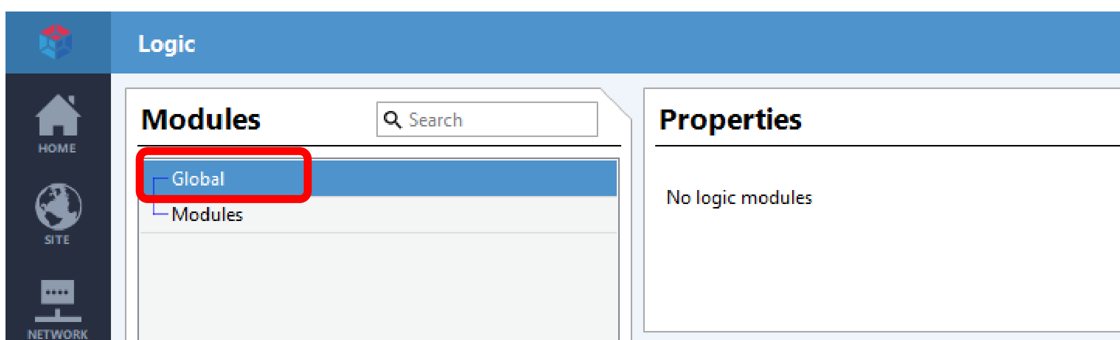
An ASCII string containing the text "hello" and terminated by a Carriage Return and Line Feed is written in C# as:

```
"hello\r\n"
```

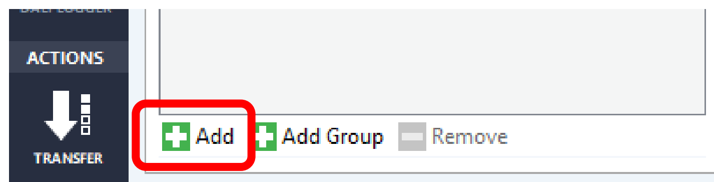
Using the Serial Port

A serial port will normally be opened when the logic first starts, then remain open. The best way to implement this is to declare the serial port variable in the Global script then use it in the Modules as required. The process of declaring the serial port variable in the Global script is:

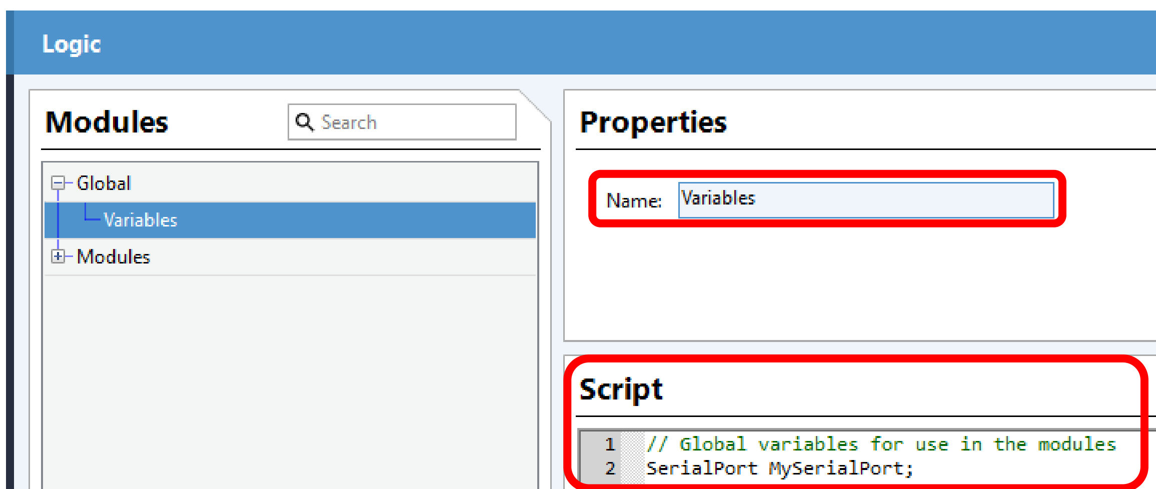
1. Select **Global** in the Modules tree



2. Click on the **Add** button



3. Enter a name (for example "Variables")
4. Type the declaration into the "Variables" script



Declaring a global serial port variable.

Opening the Serial Port

The serial port needs to be opened before it can be used. The two options for doing this are:

1. Create and open the serial port when logic first starts; or
2. Check that the serial port exists each time before use, then create and open if required.

The code in the examples below assumes that the device communicating with the Zone Controller serial port has the following properties:

- **9600 bits / second**
- **No parity**
- **One stop bit**
- **No flow control / hand-shaking**

Option 1

In a logic module that is set to run once only on start-up, add this code:

```
if (IsFirstRun)
{
  MySerialPort = new SerialPort(1, 9600, SerialPort.PARITY_NONE,
                              SerialPort.STOP_BITS_ONE, false);
  MySerialPort.Open();
}
```

Option 2

In a logic module that is set to run periodically, add this code before the serial port is used in the module:

```
if (MySerialPort == null)
{
  MySerialPort = new SerialPort(1, 9600, SerialPort.PARITY_NONE,
                              SerialPort.STOP_BITS_ONE, false);
  MySerialPort.Open();
}
```

There is a method for closing the serial port, but this should only be required in extremely rare cases.

Writing to the Serial Port

To write data to the serial port, three methods are provided. The correct one to use depends on the message format.

Method	Description
bool SendAsciiString(string message)	Send an ASCII string
bool SendUtf8String(string message)	Send a UTF-8 encoded string
bool SendBytes(byte[] bytes)	Send binary data (bytes)

To send an ASCII string "hello" which is terminated by CR/LF:

```
MySerialPort.SendAsciiString("hello\r\n");
```

To send a UTF-8 string "こんにちは" which is terminated with a null character:

```
MySerialPort.SendUtf8String("こんにちは\0");
```

To send a series of bytes with values 1, 2 and 3:

```
byte[] bytes = new byte[3] { 1, 2, 3 };
MySerialPort.SendBytes(bytes);
```

Reading from the Serial Port

To read data from the serial port, eight methods are provided. The correct one to use depends on the message format.

Method	Description
string ReadAsciiLine()	Read an ASCII string that is terminated by a carriage return and/or end of line (not including the terminators).
string ReadAsciiString()	Read an ASCII string (everything in the buffer, including any terminators)
string ReadUtf8Line()	Read a UTF string that is terminated by a carriage return and/or end of line (not including the terminators).
string ReadUtf8String()	Read a UTF string (everything in the buffer, including any terminators)
byte[] ReadBytes()	Read bytes from the serial port
byte[] ReadBytesUpToTerminator(char terminatorChar)	Read bytes from the serial port up to the terminator character
byte[] ReadBytesUpToTerminator(char terminatorChar1, char terminatorChar2)	Read bytes from the serial port up to the terminator characters
byte[] ReadBytesUpToTerminator(char terminatorChar1, char terminatorChar2, char terminatorChar3)	Read bytes from the serial port up to the terminator characters

If the message format is ASCII terminated with a Carriage Return and/or Line Feed, then use the ReadAsciiLine method:

```
string message = MySerialPort.ReadAsciiLine();
```

If the message format is UTF-8 terminated with a Carriage Return and/or Line Feed, then use the ReadUtf8Line method:

```
string message = MySerialPort.ReadUtf8Line();
```

For most other requirements, one of the methods for reading bytes will be required. For example:

```
byte[] message = MySerialPort.ReadBytesUpToTerminator('\0');
```

The serial port should be read sufficiently often to ensure that a back-log of messages does not build up. For example, if messages are expected every few seconds, then the logic module should run with a period of a second or less.

The response time also needs to be considered. If the logic needs to respond within a certain time of the serial message being sent to the Zone Controller, then the logic module period needs to be equal to this time or less.

How the read buffer works

Incoming serial data is stored in a buffer. If the incoming data is not read regularly, the buffer will fill up and any new data will be discarded.

Some examples are provided below showing what happens to the data in the buffer when read using different methods.

Buffer Content (before being read)	**Data Returned by the ReadAsciiLine method**	**Buffer Content (after being read)**
"HELLO"		"HELLO"
"HELLO\r\nTESTING\r\n123\r\n"	"HELLO"	"TESTING\r\n123\r\n"
"TESTING\r\n123\r\n"	"TESTING"	"123\r\n"
"123\r\n"	"123"	

Reading the serial port using the ReadAsciiLine method

Buffer Content (before being read)	**Data Returned by the ReadAsciiString method**	**Buffer Content (after being read)**
"HELLO"	"HELLO"	
"HELLO\r\nTESTING\r\n123\r\n"	"HELLO\r\nTESTING\r\n123\r\n"	

Reading the serial port using the ReadAsciiString method

Buffer Content (before being read)	**Data Returned by the ReadUtf8Line method**	**Buffer Content (after being read)**
"HELLO\r\n"	"HELLO"	
"HELLO\r\nこんにちは\r\n"	"HELLO"	"こんにちは\r\n"
"こんにちは\r\n	"こんにちは"	

Reading the serial port using the ReadUtf8Line method

Buffer Content (before being read)	Data Returned by the ReadBytes method	Buffer Content (after being read)
Bytes: [1, 2, 3, 4, 5]	Bytes: [1, 2, 3, 4, 5]	
Bytes: [1, 2, 3, 4, 5, 0, 1, 2, 3]	Bytes: [1, 2, 3, 4, 5, 0, 1, 2, 3]	

Reading the serial port using the ReadBytes method

Buffer Content (before being read)	Data Returned by ReadBytesUpToTerminator('\0')	Buffer Content (after being read)
Bytes: [1, 2, 3, 4, 5]		Bytes: [1, 2, 3, 4, 5]
Bytes: [1, 2, 3, 4, 5, 0, 1, 2, 3, 0]	Bytes: [1, 2, 3, 4, 5]	Bytes: [1, 2, 3, 0]
Bytes: [1, 2, 3, 0]	Bytes: [1, 2, 3]	

Reading the serial port using the ReadBytesUpToTerminator('\0') method

Example 1 – Controlling a Serial Device

For this example, the Zone Controller needs to control a serial device to turn it on or off so that its state matches a Zone.

The device serial protocol is:

- RS232 Options
  - 9600 bits / second
  - 1 stop bit
  - No parity or flow control
- Data format
  - ASCII
  - "on" to turn the device on
  - "off" to turn the device off
- Message terminator
  - Carriage return followed by line feed

The code could be done in many ways, but a simple and easy to understand solution is:

Global script, containing the following code:

```
SerialPort MySerialPort;
```

Module containing the following code, running periodically every 0.5 second:

```
// Open the serial port if needed
if (MySerialPort == null)
{
  MySerialPort = new SerialPort(1, 9600, SerialPort.PARITY_NONE,
                                SerialPort.STOP_BITS_ONE, false);
  MySerialPort.Open();
}

// If the Zone state has changed, send a serial message.
if (HasChanged(Zones["Control"].State))
{
  if (Zones["Control"].State == ON)
  {
    MySerialPort.SendAsciiString("on\r\n");
  }
  else
  {
    MySerialPort.SendAsciiString("off\r\n");
  }
}
```

Example 2 –Controlling from a Serial Device

For this example, the Zone Controller is being controlled by a serial device to turn a Zone on or off.

The device serial protocol is:

- RS232 Options
  - 9600 bits / second
  - 1 stop bit
  - No parity or flow control
- Data format
  - ASCII
  - "on" to turn the Zone on
  - "off" to turn the Zone off
- Message terminator
  - Carriage return followed by line feed
- Other
  - Zone Controller must reply with "OK" or "ERROR"

The code could be done in many ways, but a simple and easy to understand solution is:

Global script, containing the following code:

```
SerialPort MySerialPort;
```

Module containing the following code, running periodically every 0.5 second:

```
// Open the serial port if needed
if (MySerialPort == null)
{
  MySerialPort = new SerialPort(1, 9600, SerialPort.PARITY_NONE,
                              SerialPort.STOP_BITS_ONE, false);
  MySerialPort.Open();
}

// Read from the serial port.
// If no message, exit.
string input = MySerialPort.ReadAsciiLine();
if (input.Length == 0)
{
  return;
}

// Act on the message.
if (input == "on")
{
  // Turn Zone on.
  Zones["ABC"].On();

  // Send OK Reply.
  MySerialPort.SendAsciiString("OK\r\n");
}
```

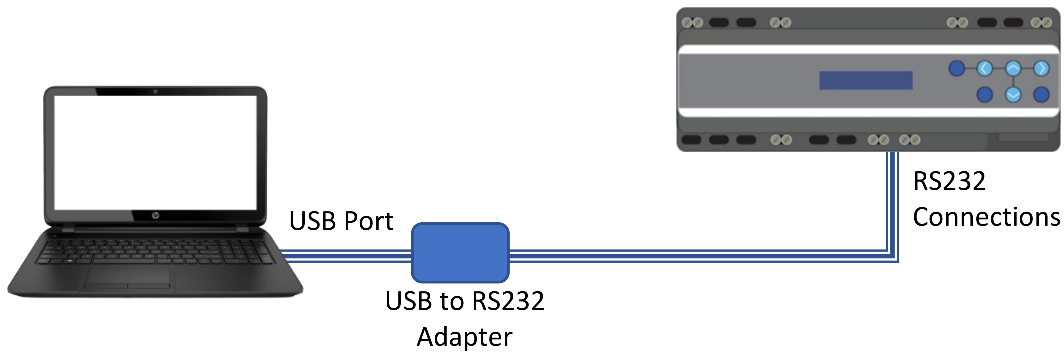
```
else if (input == "off")
{
  // Turn Zone off.
  Zones["ABC"].Off();

  // Send OK Reply.
  MySerialPort.SendAsciiString("OK\r\n");
}
else
{
  // Send error reply for any other input.
  MySerialPort.SendAsciiString("ERROR\r\n");
}
```

Testing and debugging

Before using your logic code with a real serial device, test it using a serial "terminal" program which will provide a better environment for testing and debugging. This is simplest when using ASCII format protocols.

To do this, connect the Zone Controller to your computer serial port. Many modern computers do not have serial ports, so a USB to RS-232 adapter may be required.

OR



Connecting a Computer to the Zone Controller

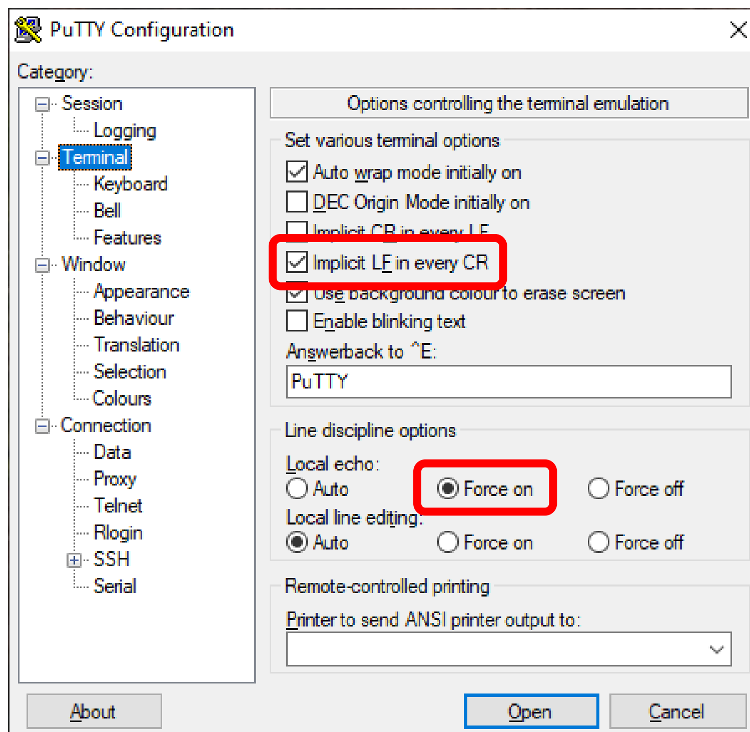
Using PuTTY

A variety of programs that can be used to communicate through a computer serial port. A typical one (for Windows PCs) is PuTTY, available from https://www.putty.org/

By default, you will only see the characters received from the Zone Controller. To see the text that you have typed, select the **Local Echo** "**Force on**" option.

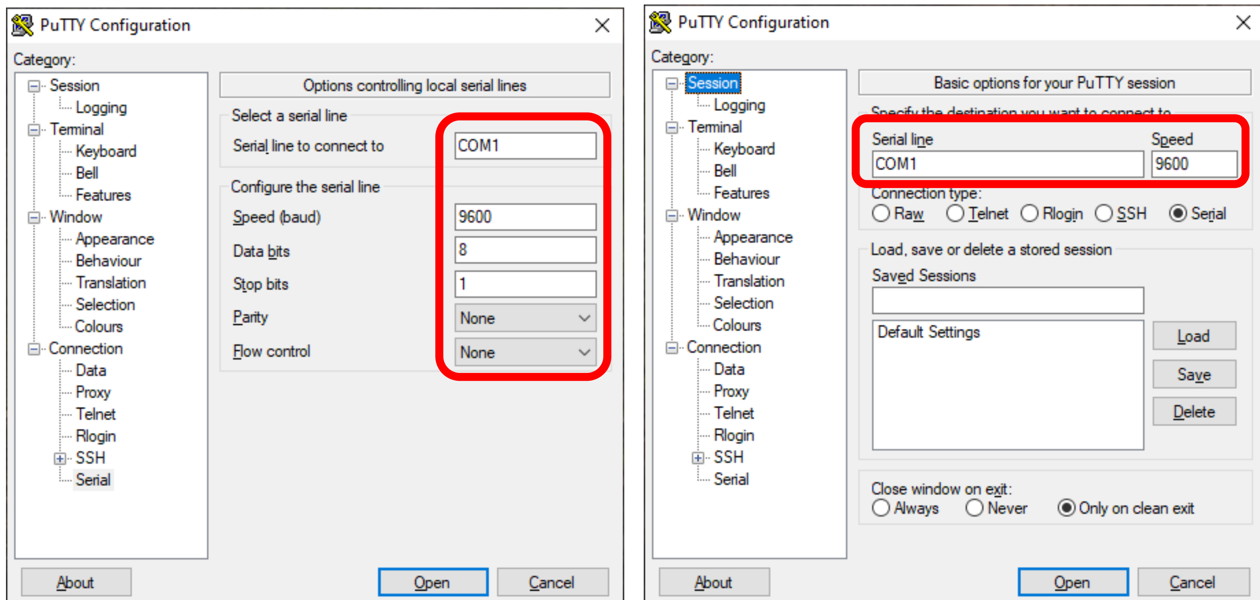
To prevent received messages from being written over the top of what you have typed, select the "**Implicit LF in every CR**" option.

These options need to be selected before opening the connection.



PuTTY Terminal Settings

The serial port needs to be configured to match the Zone Controller logic code. For the examples above, the PuTTY configuration will be:



PuTTY Serial Port Settings

Click on **Open** to start the session. The terminal will be shown:

The PuTTY Terminal

Anything typed into the PuTTY terminal window will be sent to the Zone Controller. Anything sent from the Zone Controller will appear in the terminal window.

Change History

Rev	Date	Updated By	Comment
1	5 May 2021	D S	First Release
2	26 May 2026	A Q	Updated contact details.

Contact Information

Web www.ozuno.com

All Enquiries +61 8 8362 7584 sales@ozuno.com

Ozuno Trading Pty Ltd

ABN: 96 621 194 483

RAPIX is a trademark of Ozuno Holdings Pty Ltd.

DALI and **DALI-2** are trademarks of the Digital Illumination Interface Alliance (DiiA).

COPYRIGHT © 2021-2026 This document is copyright by Ozuno Holdings Pty Ltd. Except as permitted under relevant law, no part of this application note may be reproduced by any process without written permission of and acknowledgement to Ozuno.

DISCLAIMER. Ozuno Holdings Pty Ltd (Ozuno) reserves the right to alter the specifications, designs or other features of any items and to discontinue any items at any time without notice and without liability. While every effort is made to ensure that all information in this application note is correct, no warranty of accuracy is given and Ozuno shall not be liable for any error.

TRADEMARKS. The identified trademarks and copyrights are the property of Ozuno Holdings Pty Ltd unless otherwise noted.

APN-RAPIX-018-02 May 2026