

## Serial Communications

This Technical Document describes the SilverLode Controller/Drivers Serial Communications and Networking. Other Technical Documents referenced below can be found on our website.

Operating in a host configuration, or accessing the servo's serial communications, requires networking. Networking the SilverLode servos uses industry standard protocols and serial interfaces. The serial interface selected dictates the hardware configuration, while the protocol selected affects the programming necessary on the host side.

SilverLode servos support three communication protocols:

- 8-Bit ASCII
- 9-Bit binary
- Modbus®
- DMX 512 (Rev 34-1x) with option -D

The default, and most straightforward, is the 8-Bit ASCII protocol. It uses packets constructed from 8-Bit ASCII characters to send commands to and receive responses from the servo. This is a common protocol supported by most PLCs, HMIs, and other host devices. However, the 9-Bit binary protocol provides more robust communications. Each packet consists of binary values, and includes packet length and checksum information. The 9-Bit protocol has built-in error checking to determine the integrity of all commands received.

Two serial interfaces are also supported: RS-232 and RS-485. The default, RS-232, is a widely used, easy to wire communication system. The servo uses the simple three-wire version of the RS-232 standard. RS-485 is a more noise-immune interface that supports more devices and longer distances on a single network. The tradeoff for this improvement is increased wiring complexity.

This Technical Document covers the details of both protocols and interfaces and offers some guidance in choosing between them. Suggested hardware configurations and troubleshooting tips are provided for building systems. In addition to this document, there is a great deal of information on standard protocols and interfaces available on the Internet.

## Selecting a Protocol and Interface

In the 8-Bit ASCII protocol, packets are made up of standard ASCII characters. This protocol is supported by most equipment manufacturers, and is easy to generate in a PC host. The 8-Bit ASCII protocol is recommended for most applications.

Note: For SilverDust Rev 12, the 8-Bit ASCII protocol provides an option for error checking (checksum), as well as decimal or hex arguments and decimal or hex returned values.

The 9-Bit binary protocol supplies error checking capabilities built into each packet. This makes the protocol suitable for environments where interference and transmission errors are likely or expected. The 9-Bit protocol is also faster because it requires fewer bytes to transmit a given packet. This protocol is more complex, and requires greater control over the host serial port to implement, however the binary communications may be easier for some low level controllers to control than converting into and out of ASCII. The ninth bit is the parity bit, and is used only at the start of packets, and not to establish parity. It should be set high for the first byte of a packet, and low for other bytes in the packet. If this type of control of the parity bit is not available, then the 9-Bit protocol cannot be implemented.

The choice between RS-232 and RS-485 is dictated primarily by system size and environment. RS-232 is generally only suitable for systems with less than five nodes and less than 50 feet of total cable. RS-232 has the advantage of being easy to set up, requiring only three wires to each the servo. RS-485 is capable of connecting up to 64 servos and up to 1000-foot cable runs. RS-485 also improves the noise immunity of the system, making it the suitable choice for operation in electrically noisy environments. QCI also has several additional hardware components available for electrically noisy environments. RS-485 usually requires extra communications hardware, such as converters and termination resistors.

**Choose 8-Bit ASCII for:**

- An Easy To Implement Protocol
- Non Time Critical Communications
- Minimal Programming Requirements
- Human Interpretable Data
- Compatibility With Most Devices/ Drivers
- Error Checking And Hex Or Decimal Returned Data – SilverDust Rev 08

**Choose 9-Bit Binary for:**

- Time Critical Communications
- Error Checking For Electrically Noisy Environments
- Consistent Data Length/Transmission Time

**Choose Modbus® for:**

- Standardized communication to PLC, HMI, and existing data communications converters
- Ethernet/IP bridge for SilverDust IG8 units
- Modbus/IP bridge for SilverDust IG8 units

**Choose DMX512 for:**

- Streaming data to entertainment systems. See Application Note “QCI-AN045 DMX512 Protocol”

## Communications

### Communication Port Settings

Successful communication with the servo requires that both the host and the servo serial ports have matching settings. The host must be configured for 1 start bit and 1.5 or 2 stop bits, 2 being preferable as this improves the robustness of communications. This setting is fixed within the servo. The baud rates of the servo and the host must also be set to match. The servo baud rate is adjustable internally by the Baud Rate (BRT) command; the default is 57600 baud. The servo has an additional parameter, the Acknowledgement Delay (ADL), adjusted by the command of the same name. This parameter is a specified time delay between receipt of a command and the transmission of a response from the servo. This delay is used primarily in networks and is discussed in the networking section below.

### Packets

All device serial communications are accomplished using packets. A packet is a collection of bytes of information with a particular format. There are two kinds of packets: command and response.

#### Command Packet

Command packets are sent from a host to one or more devices. All device communication begins with a command packet sent from a host. SilverLode servos are slave devices and do not initiate communication. Regardless of the protocol chosen, the basic structure of a command packet is the same. Each command packet includes an ID, a command number, and parameters.

#### ID (Address)

The ID is sent as a value in the range of 1 to 255. The ID is the first piece of information in a command packet. Every device that receives the packet checks if the ID in the packet corresponds to its Unit, Group, or Global ID. Packets not containing one of these IDs are ignored. Packets sent to the Unit ID will be acted upon, and a response packet sent. Packets addressed to a Group or the Global ID will be acted upon, but no response packet will be sent. See the section on networking below.

#### Command Number

Each command has a unique number in the range of 0 to 255. The command number follows the ID in the packet. Refer to the Command Reference for command numbers.

#### Parameters

Many commands require parameters such as distance or I/O state. These parameters are numerical values that are included in a command packet following the command number.

#### Response Packet

Response packets are sent from the servo in response to command packets. They have three forms that differ significantly depending on the circumstances. The three types of response packets are: acknowledgement (ACK), data, and negative acknowledgement (NAK).

**Acknowledgement (ACK) Response Packet**

ACK packets contain only the ID of the responding servo, indicating that a command was successfully received. These packets are as short as possible since they are the most commonly transmitted.

**Data Response Packet**

Data packets are issued in response to commands requesting information from the servo. They include the ID of the responding servo, the command number being responded to, and the data itself.

**Negative Acknowledgement (NAK) Response Packet**

NAK packets are issued in response to command packets that contain invalid information. This can be due to a variety of reasons, such as improper command parameters, perhaps induced by communication errors. NAK packets include the servo ID, the command number being responded to, and a NAK code. The NAK code is an explanation for the rejection of the command packet. The following table describes the codes.

**NAK Codes**

NAK Code	Name	Description
1	Bad Command	The servo received a command number it did not recognize. If you are sure the command number is correct, ensure that the servo has the latest firmware revision. A table of commands verses firmware revisions can be found at the end of the Command Reference. The current revision can be found in the Control Panel.
2	Device Busy	The servo is actively executing a previous command or internal program. The received command cannot be executed at this time. See the Command Reference under Command Types for details.
3	Reserved	Reserved for back compatibility with pre E-series units.
4	Reserved	Reserved for back compatibility with pre E-series units.
5	Bad Format	Command number and number of parameters is not consistent.
6	Buffer Full	The servo's program buffer is full. Too many bytes were sent to the servo for its available program buffer space (RAM).
7	Bad Address	An address used in a command was out of range. This includes: trying to access Data Registers that do not exist, accessing registers not available to the command, accessing command buffer that does not exist (negative or larger than buffer) or trying to read or write zero words of data.
8	Bad Response Packet Request	The requested return data was too large for one packet. The serial communications buffer can transmit only 31 bytes at a time. Break the request into multiple packets.
9	Bad PUP Lockout Code	Lockout Code for Protect User Program (PUP) command was incorrect.
10	Bad Checksum	8-Bit ASCII protocol checksum was incorrect. Valid for SilverDust Rev 08.

## Protocols

### 8-Bit ASCII Protocol

Packets in the 8-Bit ASCII protocol consist of standard ASCII characters. A space character delimits the components of the packet.

#### Command Packets

8-Bit ASCII Command Packet Data Transmission Order – HEX returned data				
Start Character	ID	Command Num.	Parameters ...	Ending Character
1 Byte @	1 -3 Bytes	1 - 3 Bytes	1 - 10 bytes for each	1 Byte <CR>

#### Start Character

For the 8-Bit ASCII command packets, the start character is the @ symbol. Any characters sent before the @ are ignored, including responses from other units or communication between other devices. Following the @ symbol, one or more space characters may be inserted, but are not required. Alternate Start Characters for the SilverDust Series include & and | (Vertical bar) to request the response in word unsigned decimal or double word signed decimal. The SilverDust also supports optional packet checksum and support for hexadecimal input. See below.

#### Unit ID

The Unit ID is transmitted as the ASCII representation of the number. For example, to transmit the default unit address of 16, transmit the ASCII code for 1, followed by the ASCII code for 6.

#### Command Number

The command number is transmitted in the same fashion as the ID. The ASCII space character must be transmitted between the ID and command number.

#### Parameters

Any parameters are transmitted in the same fashion as the ID and command number. The ASCII space character must be transmitted between the command number and the first parameter. A space must also be included between multiple parameters.

#### Ending Character

The final character is an ASCII Carriage Return (ASCII code 13) represented in examples as <CR>. The carriage return signals the end of the Command Packet. The servo will begin processing the command once this character is received. A space may be included between the final parameter and <CR>, but is not required.

Example: Send a Read Register (RRG) command.

Command = RRG (Command Number = 12)  
 ID = 5  
 Data Register# = 1

Fields with data shown in ASCII.

Start Char	ID	Cmd	Data Register #	End Char
@	5	12	1	<CR> (dec. 13)

Byte stream with data shown as an ASCII String: **@5 12 1 <CR>**

Example: Send a MOVE RELATIVE, TIME BASED (MRT) command.

Command = MRT (Command Number = 177)  
 ID = 16  
 Distance = 4000 counts  
 Ramp Time = 833 ticks (1 tick=120us) = 0.100 sec  
 Total Time = 8333 ticks = 1.00 sec  
 No stop conditions

Fields with data shown in ASCII:

Start Char	ID	Cmd	Distance	Ramp Time	Total Time	Stop Enable	Stop State	End Char
@	16	177	4000	833	8333	0	0	<CR>

Byte stream with data shown as an ASCII string: **@16 177 4000 833 8333 0 0 <CR>**

### Poll Command

The Poll (POL, command number = 0) command is the most commonly transmitted command. A special packet may be used in place of the standard POL packet. The command number is dropped to make the packet shorter and help minimize communication overhead. Note: The Poll Response (POR) command returns the same poll data, except that it always returns HEX data even if it is zero.

**@5<CR> (shortened format)**

### Response Packets

All the information in a response packet is in ASCII hexadecimal. Each type of response packet has a different start character, but all use the <CR> as the ending character.

### ACK Packet

The ACK response uses the \* (asterisk) as the start character, followed by a space, the Unit ID in hexadecimal, and the <CR> end character.

8-Bit ASCII ACK Packet Data Transmission Order		
Start Character	ID (Hex)	Ending Character
1 Byte *	1 - 2 Bytes	1 Byte <CR>

Example: Acknowledge response from device #16 (which is 10 in Hexadecimal):

**\* 10<CR>**

**Data Packet**

Data packets use # as the start character. This is followed by the ID of the responding servo, the command number being responded to, the actual data requested, followed with the <CR> ending character.

8-Bit ASCII Returned Data Packet Data Transmission Order				
Start Character	ID (Hex)	Command Number (Hex)	Data Fields ... (Hex)	Ending Character
1 Byte #	2 Bytes	4 Bytes	4 Bytes each	1 Byte <CR>

Example: Poll command data response from device #27 (which is 1B in Hexadecimal). The actual data returned is the Polling Status Word.

**# 1B 0000 2000<CR>**

Example: Read Register command data response from device #10 (0x0A in Hexadecimal); the last 8 digits represent the 32-bits of data. The Read Register command number is 12 (0x0C in Hexadecimal).

**# 0A 000C 0005 06A3<CR>**

The current contents of this register are 0x506A3 or 329379 in decimal.

**NAK Packet**

The NAK response uses ! as the start character. It is followed by the ID, command number, NAK code, and a <CR>.

8-Bit ASCII NAK Response Packet data transmission order				
Start Character	ID	Command Number	NAK Code	Ending Character
1 Byte !	2 Bytes	4 Bytes	4 Bytes	1 Byte <CR>

Example: NAK response for a Read Register (RRG) (Command Number 0x0C) from device #10 (0x0A) with an invalid Data Register parameter.

**! 0A 000C 0007<CR>**

Example: NAK response for a Move Relative, Time Based (MRT) (Command Number 0xB1) if device #16 (0x10) is already in motion. The servo will NAK back Device Busy as follows:

**! 10 00B1 0002 <CR>**

### **8-Bit ASCII Protocol – Expanded \*(SD12)**

The 8 bit ASCII protocol has been expanded in the SilverDust Rev 12 code to allow the return of decimal data as well as hexadecimal data. The input parameters may be individually specified as decimal or HEX. The packet, in any of the above combinations, may also be sent with a checksum to detect communications errors.

#### **HEX Parameters**

HEX Parameters may be sent simply by prefixing the number with 0x - that is a 16 may be sent as 0x10. This applies equally to unit ID, command, parameters, and checksum.

A poll command using Hex parameters would be sent as

**@0x10 <CR>    or    @0x10 0<CR> or @0x10 0x0<CR>**

Each of these is interpreted the same as if

**@16 0<CR>    or    @16<CR> were sent**

The response might be either

**\* 10<CR>**

or

**# 10 0000 2001<CR> (bits 0 and 12 set)**

#### **Decimal Format Response Packet**

On previous SilverLode controllers, all response packets were in HEX format. With the new SilverDust, the response packet format for any given command packet can be changed to decimal by starting the packet with the “&” character instead of “@”. To eliminate any confusion, all response data is sent as unsigned. SilverDust Rev 31-1x and higher code additionally supports a Decimal Signed Long-Word Response by starting the packet with the “|” character instead of the “@” or “&” character.

The start characters of the decimal format response packets have been changed to aid in stateless decoding of the returned data.

Response Packet	Hex Format Response	Decimal Format Response	Decimal Long Response
Start Character	@	&	
ACK Packet	*	%	%
Data Packet	#	\$	/
NAK Packet	!	?	?

Response packets are formatted as groups of 16-bit unsigned words. For 32 bit data (i.e. response packet from a Read Register (RRG) command), the following formula can be used to combine two 16-bit unsigned words.

$$\begin{aligned}
 W_H &= \text{High 16 Bit Word} \\
 W_L &= \text{Low 16 Bit Word} \\
 R &= \text{Result} \\
 R &= W_H * 65536 + W_L
 \end{aligned}$$

If the number was signed, then the 2's complement math must be performed. See below for an example of converting from unsigned to signed using 2's complement math.

RRG example: Read 32 bit signed value from register 10.  
 Assume register 10 = -20 (0xFFFFFEC)

Command Packet  
 &16 12 10 <CR>  
 Response Packet  
 \$16 12 65535 65516<CR>

$$\begin{aligned}
 R &= W_H * 65536 + W_L \\
 R &= 65535 * 65536 + 65516 \\
 R &= 4294901760 + 65516 \\
 R &= 4294967276
 \end{aligned}$$

If the host computer has defined R as a 32 bit signed value,

$$R = 4294967276 = -20$$

If the host computer has defined R larger than 32 bits or a floating point number, then following calculation must still be done:

$$\begin{aligned}
 \text{If } R >= 268435456 \text{ (0x10000000 or } 2^{31}\text{)} \\
 R' &= R - 4294967296 \text{ (} 2^{32}\text{)} \\
 \text{Else} \\
 R' &= R
 \end{aligned}$$

A poll command using Hex parameters would be sent as

**&0x10 <CR> or  
&0x10 0<CR> or  
&0x10 0x0<CR>**

Each of these is interpreted the same as if

**&16 0<CR> or  
&16<CR>**

were sent.

The response might be either

**% 16<CR>**

or

**\$ 16 0 8193<CR>**  
(bits 0 and 12 set)

### **Decimal Long-Word Response Format (SD 31)**

The Decimal Long-Word format responds with the same ACK and NAK packets as does the Decimal response. Data response packets containing multi-word responses are parsed as signed long-words for each pair of data words in the response after the Unit ID and Command Number. Single word responses are parsed as unsigned numbers. This format is most useful with immediate commands Read Register (RRG), 12, and Read Register Write (RRW), 32, as these commands return the values of 32bit registers.

Example Register read of Register 11 with a value of -1,234,567,890, and Register 12 with a value of 5,000,000:

**|16 12 11 12<CR>**

**/ 16 12 -1234567890 5000000<CR>**

### **8-Bit Checksum**

An optional checksum field can be added to the command packets to ensure data integrity. A checksum is specified by inserting a "(" prior to the unit address, and ")" following the last parameter, followed by the modulo 256 checksum of all characters sent between the leading "(" and the trailing ")" including any spaces. Spaces outside of the parenthesis are not included in the checksum. Like all the parameters, the checksum may be sent as either decimal or as HEX.

The following example reads the revision (RVN, cmd=5) from device 16.

**@ (16 5) 188<CR>**

Checksum Calculation:

ASCII CHAR	DECIMAL VALUE
1	49
6	54
space	32
5	53
Checksum	188

Example response packet (depends on device's firmware revision). Note that a checksum and parenthesis are added to the response packet.

**#(10 0005 0809 2005 1111 FF15)14 <CR>**

A NAK code of 10 will be returned if a bad checksum is received.

The checksum, like any other parameter, may be sent in Hexadecimal by prepending a "0x":

**@(16 5) 0xBC<CR>**

**#(10 0005 1010 2006 E131 FF15)1C <CR>** (Varies with Code Revision)

The checksum may also be intermixed with return formats, such as Decimal response, with both the response and the checksum returned in decimal format:

**&(16 5) 188<CR>**

**\$(16 5 4112 8198 57649 65301)236**

or

**! &(16 5) 188<CR>**

**/(16 5 269492230 -516817131)207<CR>**

### 9-Bit Binary Protocol

The more advanced 9-Bit binary protocol uses hex values (rather than ASCII characters) and error checking to provide greater speed and reliability than the 8-Bit ASCII protocol. Each command and response packet includes length and checksum data. the servo will reject any packet in which the error checking data does not match.

The ninth bit used in this protocol is the parity bit available as part of standard serial communication drivers. In this protocol, however, the bit is not implemented to create even or odd parity, but rather to serve as a flag indicating the start of a packet.

All values are transmitted in hexadecimal form in this protocol.

### Command packet

9-Bit Binary Command Packet Data Transmission Order				
ID	Length	Command #	Parameters ...	Checksum
1 Byte + 9th bit set	1 Byte	1 Byte	2 or 4 Bytes each	1 Byte

### Unit ID

The target ID is the first byte transmitted in the 9-Bit protocol. The parity bit is set (or mark) for this byte only. This indicates that it is the start of a command packet. the servo will ignore any transmission prior to the byte with the parity bit set.

### Length

The second byte is the length. The length is defined as the number of bytes from the command number up to but not including the checksum. Put another way, it is one (the command number) plus the number of bytes used by all parameters. This value of the length is limited to the range 0-31.

### Command Number

The command number is the next byte, in binary form.

### Parameters

Command parameters are also sent in binary form. 16-bit parameters must be transmitted as two bytes, and 32-bit parameters as four. That is, all the bytes must be transmitted, even if a small value would only require a single byte. The size of a parameter is listed in the Command Reference.

### Checksum

The final byte of the packet is the checksum. This is used to verify that the rest of the packet arrived intact. The checksum is the 2's complement (1's complement + 1) of the sum of the rest of the previous bytes. In other words, add up all bytes from the ID through the final parameter and take the 2's complement.

Example: Calculate the following Read Register (RRG) command.

Address = 16 (0x10) – ignore 9th bit  
 Length = 03 (0x03)  
 Command = RRG (Command Number = 12, 0x0C)  
 Data Register# = 1 (0x0001)

Add up all the bytes (all numbers shown in hex)  
 0x10 + 0x03 + 0x0C + 0x00 + 0x01 = 0x20

1's Complement (invert)  
 Invert (0x20) = 0xDF

Add 1  
 0xDF + 0x1 = 0xE0

Checksum = 0xE0

**POL Command Packet**

The POL command packet is a special command packet that has been shortened to minimize communications overhead, as this is probably the most frequently sent packet. See Poll Response (POR) command; same as POL but always returns data even if zero.

9-Bit Binary POL Command Packet Data Transmission		
Address	Length	Checksum
1 Byte + 9th bit set	1 Byte = 0x00	1 Byte

- First byte is the ID with the ninth bit set.
- Second byte is the length - set to zero. This defines the packet as the POL Command packet.
- (Since the data count is zero, no command or parameters are included.)
- Third byte is the Checksum.

Example POL command to Unit ID 10 (Hex 0x0A), byte stream in hex: **0A 00 F6**

Example: Send a Read Register (RRG) command with the following:

Command = RRG (Command Number = 12)  
 ID = 16 (Hex 0x10)  
 Data Register # = 1

Fields with data shown in hex.

ID	Length	Command	Data Register #	Checksum
0x10	0x03	0x0C	0x0001	0xE0

Byte stream with data shown in hex:  
 note: "[ ]" around first byte signifies parity set high  
**[10] 03 0C 00 01 E0**

Example: Send the following Move Relative, Time Based (MRT) command:

Command = MRT (Command Number = 177)  
 ID = 16  
 Distance = -4000 counts (Hex 0xFFFFF060)  
 Ramp Time = 833 ticks (1 tick=120us) (0x00000341)  
 Total Time = 8333 ticks (0x0000208D)  
 No stop conditions

Fields with data shown in hex:

ID	Length	Command	Distance	Ramp Time	Total Time	Stop Enable	Stop State	Checksum
0x10	0x11	0xB1	0xFFFFF060	0x00000341	0x0000208D	0x0000	0x0000	0xEF

Byte stream with data shown in Hex:

**[10] 11 B1 FF FF F0 60 00 00 03 41 00 00 20 8D 00 00 00 00 EF**

**Response Packets**

All of the information returned from the servo is in binary format. The ID of the responding servo is returned as a single byte with the ninth bit set. The length of the return packet is returned as a single byte. All parameters are returned as bytes with 2 bytes for words (16-bit) and 4 bytes for long words (32-bit). The checksum is returned as a single byte.

There are three types of response packets: Acknowledge (ACK), Returned Data, and Negative Acknowledge (NAK). Each has a different form to allow them to be easily parsed by the host system.

**ACK Response**

The ACK response is the positive acknowledge of the receipt of a command packet. This response has also been shortened to minimize the load on the serial communications interface, as it is the most common response of the servo. It contains only the ID and a special code in the length slot to indicate an ACK.

9-Bit Binary ACK Response Packet data transmission order		
ID	Length (ACK Code)	Checksum
1 Byte + 9th bit	1 Byte = 0x80	1 Byte

- The first byte is the ID of the responding servo with the ninth bit set.

- The second byte is the length value of zero plus the eighth bit set to indicate an ACK (128 (0x80)). This is the equivalent of a length value of 128.
- The third byte is the checksum. This is calculated in the same fashion as a command packet.

Example: ACK response from device #16 (shown in hex): **[10] 80 70**

**Data Packet**

The returned data packet is sent when the servo receives a properly formatted command requesting data. The format is as follows:

<b>9-Bit Binary ACK Response Packet data transmission order</b>				
<b>Address</b>	<b>Length</b>	<b>Command Number</b>	<b>Data fields ....</b>	<b>Checksum</b>
1 Byte + 9th bit set	1 Bytes	1 Byte	2 or 4 Bytes each	1 Byte

- The first byte the address with the ninth bit set.
- The second byte is the length, which is the number of bytes starting with the command number and including all the data fields.
- The third byte is the command number that is being responded to.
- The requested data fields are next. 16-bit words are sent one byte at a time, upper byte first. 32-bit long words are also sent a byte at a time, upper byte first.
- The final byte is the checksum.

Example: Send a Read Register (RRG) command and receive the following response packet:

Command = RRG (Command Number = 12)  
 Address = 16  
 Data Register# = 1

Response Packet with field data shown in Hex:

<b>Address</b>	<b>Length</b>	<b>Command</b>	<b>Value of Register #1</b>	<b>Checksum</b>
0x10	0x05	0x0C	0x00000FA0	0x30

Byte stream with data shown in Hex: **[10] 05 0C 00 00 0F A0 30**

**NAK Response**

The NAK response packet is sent when the command issued is not a valid command. This occurs if the command packet is improperly formatted, when the parameters are invalid, or when a command is sent at an invalid time—such as requesting a motion when the servo is already busy. Note: some invalid parameters are only evaluated when a command is executed, so a command may be downloaded with out error, but may still error when run.

<b>9-Bit Binary NAK Response Packet Data Transmission Order</b>					
<b>Address</b>	<b>Length</b>	<b>NAK Indicator</b>	<b>NAK Response Code</b>	<b>Command Number</b>	<b>Checksum</b>
1 Byte + 9th bit set	1 Byte	1 Byte = 255 (0xFF)	1 Byte	1 Byte	1 Byte

- The first byte is the address of the responding the servo with the 9th bit set.
- The second byte is the length. This is defined as the number of bytes starting with the NAK indicator through the command number.
- The third byte is the NAK Indicator. The fixed value of 255 (0xFF) indicates this is a NAK response packet. This byte is always included to indicate a NAK.
- The fourth byte is the NAK Error Code. The code indicates the error. The codes are tabled at the beginning of this section.
- The fifth byte is the command number of the command that caused the NAK. Command numbers above 128 will be returned with the most significant bit removed. For example, Identity (IDT, command number 155) would be returned as 27.
- The final byte is the checksum.

Example: NAK response for a Read Register (RRG) (Command Number 0x0C) from device #10 (0x0A) with an invalid Data Register parameter.

**[0A] 04 FF 00 07 0C E0**

Example: NAK response for a Move Relative, Time Based (MRT) (Command Number 0xB1) if device #16 (0x10) is already in motion. The servo will NAK back “Device Busy” as follows:

**[10] 03 FF 00 02 B1 3B**

## **Modbus® Protocol**

For the SilverNugget, Modbus® requires a special firmware revision (see price list for ordering information). Note, the Modbus® version of the SilverNugget comes with QCI's 8-Bit ASCII protocol and does not include QCI's 9-Bit protocol or the Position Compare (PCP) command. SilverNugget Rev 43+, series 7-B. (i.e. Rev 43-7, 43-8,...).

Modbus® is included in SilverDust firmware Rev 02 and higher. Note, for the SilverDust; all three protocols are available in the same firmware revision.

See Application Note QCI-AN038 for details on QuickSilver's implementation of Modbus® and communicating with a Modbus® device including an example program.

Modbus® is a registered trademark of Modicon.

## **DMX512 Protocol**

For the SilverDust, firmware Rev 34-1x or higher is needed, as well as the –D option (adds an internal jumper to enable DMX). DMX512 protocol allows multiple contiguous data elements to be extracted from the data frame in big endian or small endian format, with or without sign extension to any of the writable registers. Note that Baud Rate 2500 (250k) is needed. DMX512 is receive only, and RS-485 only. See Application Note "QCI-AN045 DMX512 Protocol" for more details.

## **Serial Interface**

The servo has two communication hardware configurations available, RS-485, or RS-232. The serial interface is selected by the Serial Interface (SIF) command. The serial lines are used to program the servo, issue instructions to the servo, or query the servo for data. Choosing between the two available standards is a matter of application requirements and system complexity. This section covers the two protocols, their requirements, and how to choose between them.

When working with serial communications, it is important to remember that RS stands for "Recommended Standard." There can be a great deal of variation between manufacturers, sometimes making communication setup difficult. Wherever possible this section offers suggestions on avoiding these problems.

### **RS-232**

RS-232 mode is the default setting of the servo. This is a widely used standard for serial communication between devices. It is simple to set up, requiring only three wires, but is more susceptible to noise and has limited networking capabilities.

### **RS-485**

RS-485 is a more robust serial interface, with powerful networking capabilities. These capabilities require a more complicated setup, including termination and software

configuration. The extra effort is rewarded with support for bigger networks, longer runs at higher speeds, and better noise immunity.

### Comparing RS-485 and RS-232

The following terms need to be defined to compare the two standards:

**Balance** – Signals can be either balanced or unbalanced. Unbalanced interfaces use a voltage level on a single wire to transmit either a 1 or a 0. This level is in reference to the required ground wire. Balanced interfaces use two wires to transmit a signal. The state of the bit is determined by the difference in voltage between the two lines. This scheme is also called a differential signal. When one signal is driven high, the complement signal is driven low and visa versa; when using twisted pair wiring, this differential signaling both produces less radio frequency interference (RFI), and is more resistant to noise and ground offsets.

**Slave Device** –SilverLode servos are slave devices, meaning that they do not initiate communication on their own. They only respond after receiving a command.

**Half/Full-Duplex** – Full-duplex systems can transmit and receive data at the same time. They usually have a separate wire (or wires) for each signal. In a SilverLode network, this means that a transmission from a host and a response from a servo will not interfere with each other. Half-duplex systems use the same line(s) to transmit and receive. In these systems, only one node can be transmitting at any time. Half-duplex systems require the addition of a delay before transmitting a response. This delay gives the communication circuitry time to switch modes.

**ACK Delay** – This is a setting within the SilverLode that causes a time delay between the receipt of a command packet and the transmission of a response packet. This delay is required in half-duplex systems to give devices time to switch from transmit to receive, and vice-versa.

**Termination** – Differential communication systems usually require biasing resistors. The collection of resistors is referred to as termination. The networking section of this document covers termination in detail. The termination prevents signals from reflecting from the ends of a network, thus reducing “ringing” of signals on the network. Biased terminations bias the network to sustain a “marking” state on the network when no nodes are transmitting. The marking state is required for the nodes to recognize a falling edge start bit at the start of the first character of the packet. Some alternate methods pre-drive the network prior to starting transmission, or send null characters prior to the first required character to eliminate the need to bias the network. The SilverDust provides this option by means of the **ACK Delay Extended (ADX)** command. Following the receipt of a transmission, the Unit delays the number of 120uS time ticks specified in the **ACK DELAY (ADL)**, and then drives the bus active with a steady state signal for the number of time ticks specified in the **ACK Delay Extended (ADX)** command before responding with data. The time associated with the ADX command should be calculated to be at least 1.5 character periods at the selected baud rate.

This table lists the industry standard specifications for RS-232 and RS-485

	<b>RS-232</b>	<b>RS-485</b>
Cabling	Single ended	Balanced - differential
Number of Devices	1 transmit <sup>1</sup> 1 receive	64 transmitters (1/2 load Rx) <sup>2</sup> 64 receivers
Communication Mode	Full duplex	Half duplex
Maximum Line Distance	50 feet (at 19.2 kbps)	4000 feet (at 100 kbps)
Maximum Data Rate	19.2 kbps (for 50 feet) 115kps (for 6 feet)	10 MB/s (for 50 feet) <sup>3</sup>
Signaling Mode	Unbalanced	Balanced
Mark (data 1)	-5 VDC min. -15 VDC max.	1.5 VDC min. (B>A) 5 VDC max. (B>A)
Space (data 0)	5 VDC min. 15 VDC max.	1.5 VDC min. (A>B) 5 VDC max. (A>B)
Input Level (Minimum)	+/- 3 VDC	0.2 VDC difference
Output Current	500 mA max (Note: driver ICs normally used in PCs are limited to 10 mA) <sup>4</sup>	250 mA

<sup>1</sup> SilverLode servos support multiple units (up to 5) with RS-232; setting the ACK delay greater than 0 enables this multi-drop configuration.

<sup>2</sup> RS-485 specifications allow for 32 nodes/devices with full load receivers per network.  
A SilverLode servo use 1/2 load receivers and allows up to 64 nodes/devices per network.

<sup>3</sup> Actual transmission speed limited to the SilverLode baud rate

<sup>4</sup> See the Additional Information and Troubleshooting below.

## **Choosing RS-232 or RS-485**

### **Simplicity**

The simplicity of setting up RS-232 makes it the default choice for most applications. It is recommended, unless RS-485 is required for one of the following reasons.

### **Nodes**

RS-232 can support at most five nodes in one network. Further, this size can only be achieved with a high quality host serial port and cabling. An RS-485 QuickSilver network will support up to 64 nodes with standard equipment. Large networks require RS-485.

### **Cable Length**

RS-232 will support up to 50-foot cable runs at 19.2k baud, and 10-foot runs at 57.6k baud. This also requires high quality equipment. RS-485 will support up to 4000 feet, making it the choice for geographically large networks.

### **Noise**

RS-232 provides very little noise immunity. In applications with significant electro-magnetic interference (EMI), such as welding or other high-energy machinery, RS-232 communications is more easily disrupted. In these applications, the noise immunity provided by RS-485 may be required. In both cases, high quality shielded wiring reduces noise. RS-485 further should use shielded twisted pair. The twisting of the pairs reduces magnetic coupling into the signals while the shielding reduces the electric field coupling. QCI has additional filters available to improve the noise immunity of any system.

### **Industrial Standard**

Many industrial devices support RS-485 but do not support RS-232, usually for the above reasons. Obviously, in these cases RS-485 must be used. Many devices also use RS-422 interfaces, which use RS-485 signaling levels, but use separate terminals for transmit and receive signals. If the transmit signals are configured for multi-drop, then the two pairs may typically be connected together to be compatible with RS-485. See manufacturer specifications before making any connections to verify that they support this mode.

## Implementing a SilverLode Communications Network

### General requirements

#### Unit/Group/Global ID

Each SilverLode servo on a network must have a different Unit ID. This is set by the Identity (IDT) command. The command also sets the Group ID. The servo will react, but not transmit a response packet to the Group ID. Multiple servos can share a Group ID. A Group ID would typically be used to initiate coordinated action of several, but not all, servos in a system. (i.e. via a Run Program (RUN) command). The Global ID, fixed at 255, is a reserved value to which all units will react, but not respond. SilverLode servos will not send response packets when either the Global ID or the Group ID is used. The Global ID is typically used to stop all motions on a multi-axis system simultaneously, or to initialize a device with an unknown ID.

#### Shielding

Cable shielding is required for many technical reasons, commonly noise reduction in systems or environments where EMI is heavy. Typically the DC supply power wires do not need shielding. However, communication lines are generally shielded to insure reliable data transmissions. Shielding the communication lines & I/O lines will also provide some protection from stray external electro-static discharge (ESD) that can possibly harm the host controller or the SilverLode servo. RS-485 signals should further be twisted shielded pairs, as the differential mode of operation of these signals is improved with a balanced transmission line; longer runs should more carefully match impedances of transmission line segments and terminators, both typically selected for 120 ohms.

Analog inputs to the SilverLode servo have a few mV per step increment. These lines require shielding to minimize noise and produce accurate resolution. The servo digital inputs may also require shielding in applications with long wire lengths and/or electrically noisy environments. Most cable shields are connected at one end and not connected at the other end of the cable. This will reduce the likelihood of ground loop problems developing. The Motor cable use with the SilverLode units has the motor drive signals terminated at both ends to allow noise currents flowing from the drivers through the capacitance between the windings to the stator, to be conducted back to the drivers via a path that minimizes any radiation area so as to minimize EMI. A ferrite bead is included on these cables to break the AC ground loop by inserting a significant impedance to any currents circulating through alternate ground loops (not returning via the same cable).

#### Logic Ground

The logic or communication ground of all devices on a communications network should be connected. For RS-232, this connection is required to achieve communications, while strongly recommended under RS-485. A logic ground connection prevents the development of large voltages across communication lines which at various levels disable communication and can destroy interface circuitry.

### **Termination**

RS-485 termination can be difficult to calculate for system designers not familiar with the interface. The wiring diagram section and the additional information section provide details on implementing termination in standard SilverLode networks.

### **Power**

DC power must be routed to the QuickSilver network nodes using a two-wire connection scheme supplied by an external power source. Wire gauge must be sufficient to carry the required current with the allowable voltage drop (including regenerated voltage effects). The entire QuickSilver network can be supplied with power from a single pair wire buss networked to each servo.

For example, let us say your SilverNugget servo application has a peak operating current of 4 Amps. In a network of three such servos, a total of 12.0 Amps is required, if all units will be operating at peak power. The power wires, connectors, and power source should be reliable and rated for the application. In this example, assuming total network cable length is less than 10 ft., 16 to 18 gauge stranded wire can be used for power lines. A heavier gauge supply wire for long lengths will reduce voltage loss that can decrease high speed power output from the servos. The clamp (if needed) should be located in proximity to the controllers to minimize excess voltage due to regeneration.

## Hot Plugging

Power must be disconnected before attaching or removing connections to the SilverLode products, both motors and controllers. Making connections with power applied will degrade the gold plating on contacts, leaving contacts that are more prone to failure. Additionally, the spark caused when disconnecting an inductive component, such as a motor may produce voltages much higher than the supply voltage, causing component failure. Finally, hot plugging the power connector runs the risk that the positive supply will contact prior to the ground return. If power is applied with no return paths, I/O (including serial communications) with an alternate path to ground may supply the return path for the applied power, causing damage to the affected I/O. Make all connections with power disconnected. Secure (tighten screws on D-sub connectors) and double check connections prior to connecting power.

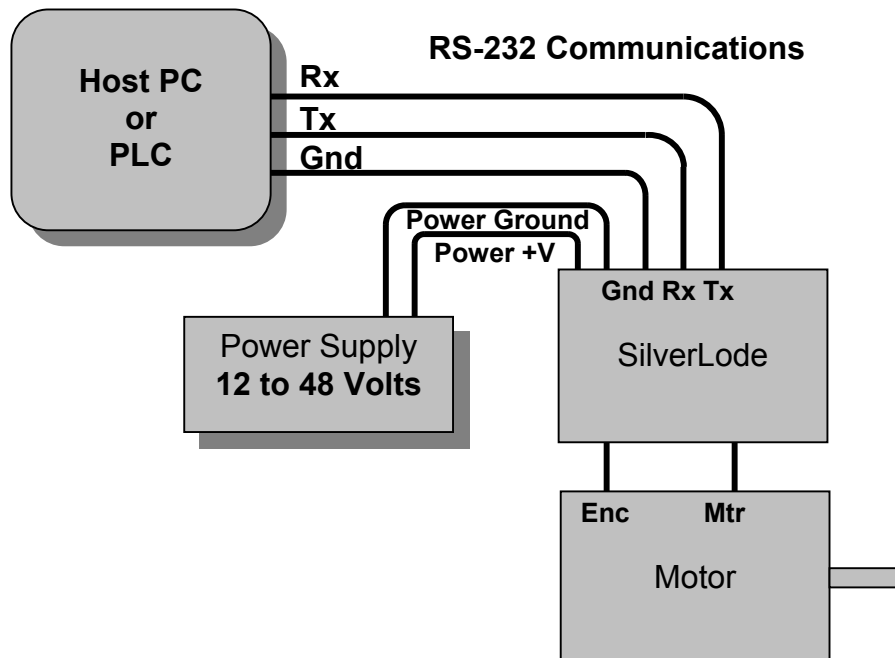
Damage caused by Hot Plugging is not covered under warranty.

**Note:** The “power” switch on the SilverDust IG and IGB only controls power to the local processor as an aid to initializing multiple motor configurations without disconnecting controllers from each other. It **WILL NOT** protect against hot plugging.

## Example Wiring Diagrams

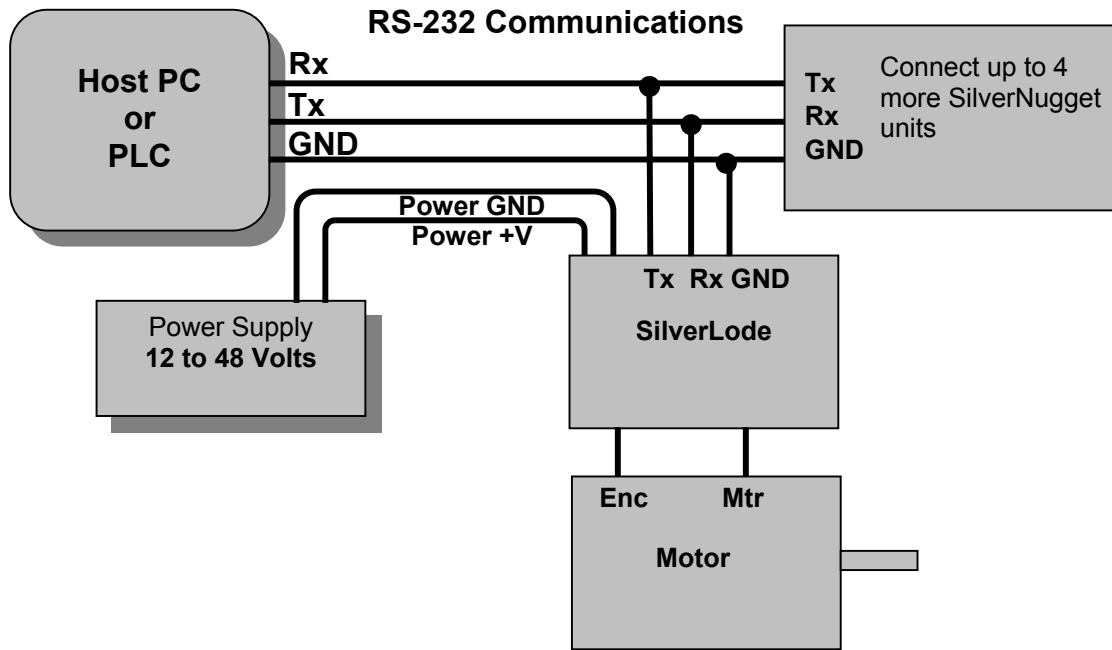
### RS-232

This interface can be directly connected to a SilverLode product. The wiring is called a null-modem configuration because the Tx line of the host is connected to the Rx line of the servo, and vice-versa. The following example show RS-232 for a SilverLode controller.



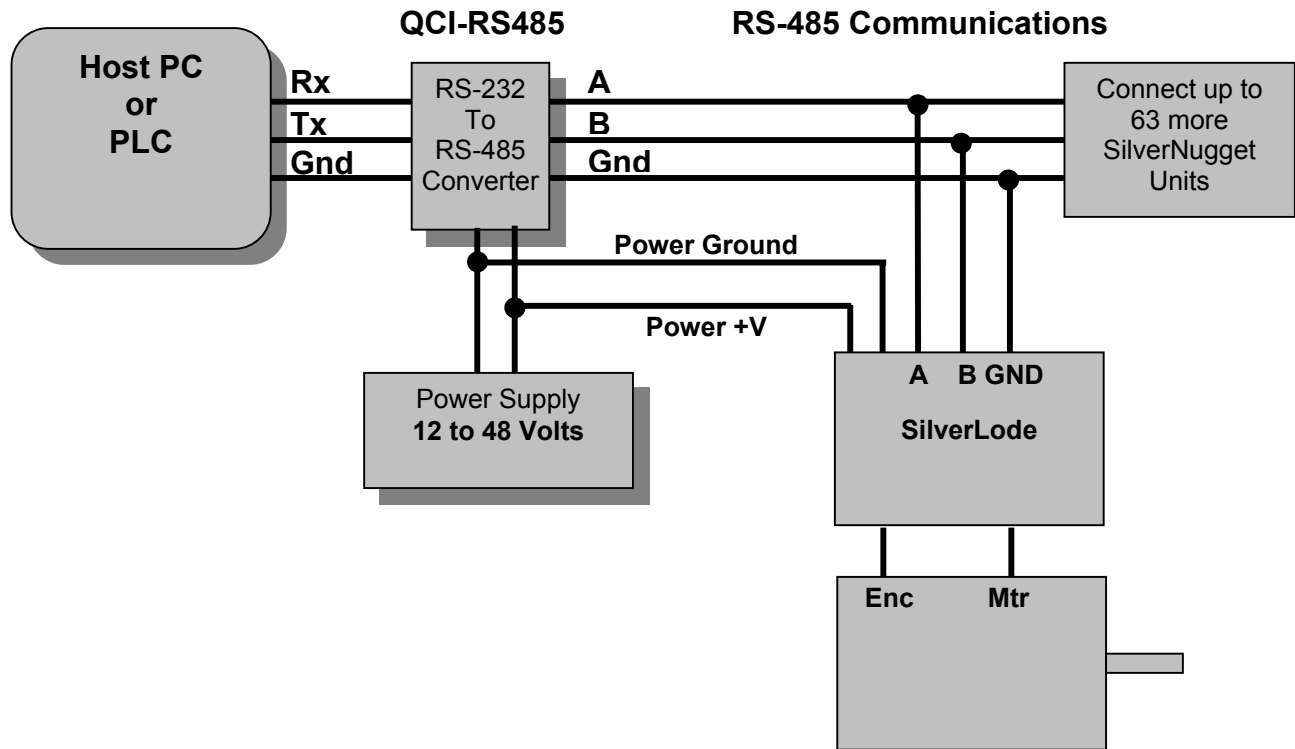
**RS-232 Network**

Up to five devices can typically be connected on a single network, according to the capability of the PC serial port (laptops often have low drive capability). This is available from the advanced communication chip within the servo. To implement this mode, use the ACK Delay (ACK) command with a non-zero value, typically 1 msec. This causes the servo to tri-state its transmit line when not in use, allowing other devices to transmit on the line. All SilverLode servo's Tx lines are tied together, as are all Rx lines. The collected lines are connected to the host using the same null-modem configuration as in a single servo system. The following examples shows multiple SilverLode controller/drivers with motors.



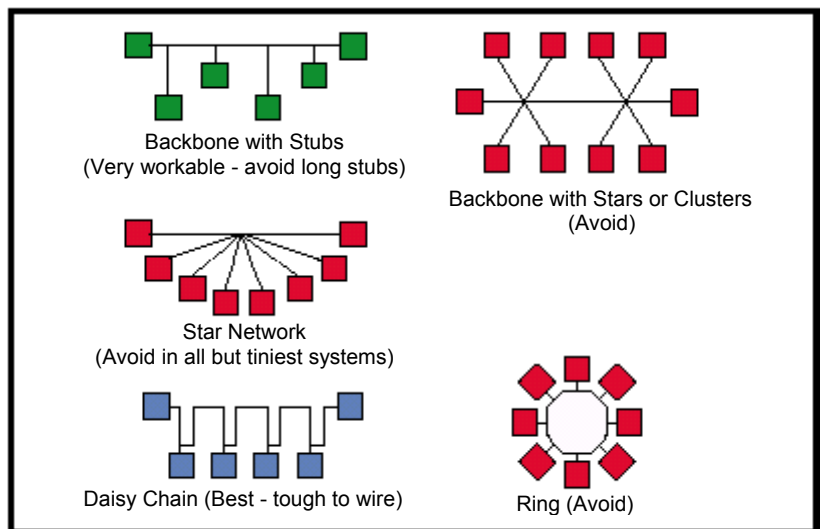
**RS-485 Network**

RS-485 is a more robust communication protocol designed specifically to work with multiple nodes. The three-wire system has three signals—"A", "B", and logic ground. All "A" lines are tied together, as are all "B" lines. Logic ground lines should be tied together to improve noise reduction. Standard RS-485 supports 32 nodes. The servo uses special half-power RS-485 chips that allow up to 64 nodes per network. NOTE: Both ends of the RS-485 bus should normally be terminated; at least one end should be a biased termination.



**Connection Topology**

RS-485 networks generally require termination. The recommended wiring method is a daisy chain style connection with both ends terminated. This type of network uses termination at extreme ends of the bus, commonly the host node and the last node (furthest from host). Backbone-stub type connections approximate the daisy chain configuration as long as the stubs are relatively short.

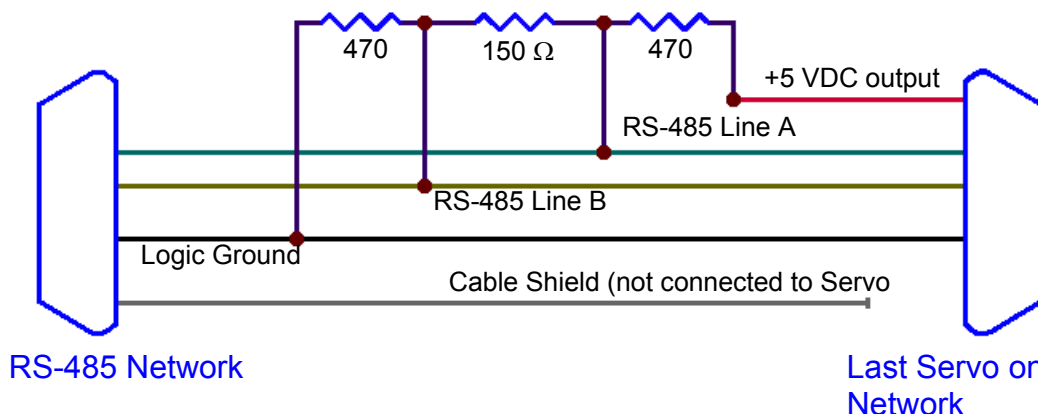


**Biased Termination**

At least one of the terminations should be a Biased Termination. The biasing allows the network to maintain a proper “Marking” level between transmissions. This allows the first start bit of the first characters sent on the bus to be properly identified for asynchronous communications. If the bus were to float to a “Space” level, the leading start bit might be missed, causing the first one or more characters to be missed. The biasing puts the bus in a known state.

The following resistor configuration should be implemented on at least one end of the RS-485 network. This will correctly bias the differential voltage between line A and B. The configuration provides both biasing and termination of the network. The termination impedance (effective impedance between A and B) should approximate the impedance of the wiring forming the backbone of the system; common 24-26ga twisted pair is commonly in the range of 100 to 150 ohms, so 120 ohms is a typical termination value. Long wiring runs may need to specify the impedance of the wire for optimal performance. The termination prevents the transmitted signal from reflecting or “echoing” from each end of the transmission line, causing ringing and loss of communications on the line. (This is much the same effect as removing the echo effect from a room by acoustical tiles, carpet and or drapes to absorb the unwanted sound energy rather than having it reflected back into the room. In the case of the network, the energy sent down the transmission line requires a means to be dissipated when it reaches the end of the wire or it will reflected back.)

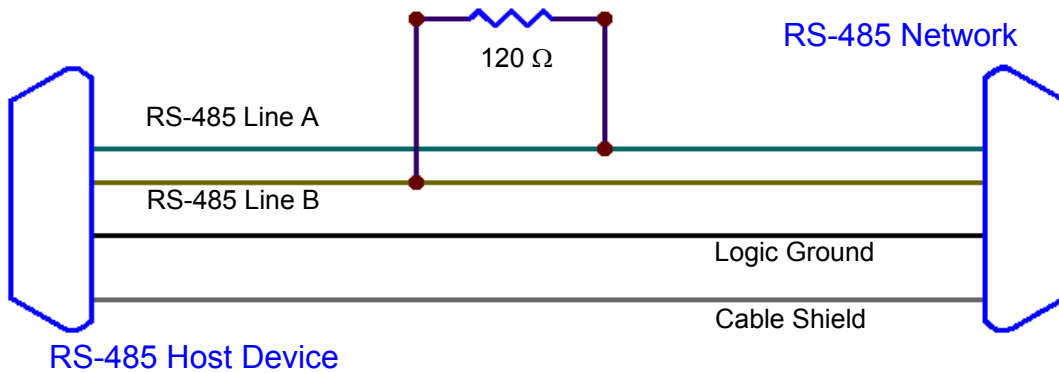
The effective impedance is  $940 // 150 = 129$  ohms. The biased voltage, with one biased termination and one unbiased termination is approximately 0.3v; ¼ watt resistors may be used.



Note: The SilverDust IG8, and IGC include biased terminators, accessible by jumpers on the front panel. See their documentation for details.

**Unbiased Termination**

One end of the network may be unbiased. The effect of this resistor has already been included in the calculations of the biased termination. The unbiased termination consists of resistor of 120 ohms connected between line A and line B at the one end of the network. Some RS-485 serial devices have internal jumpers available to add this terminating resistor. If an external termination, a standard 1/8 Watt 120 ohm resistor with 5% tolerance is generally effective. The function of a termination is to prevent the signal from reflecting back from the end of a transmission line, so as to prevent ringing which would disrupt communications.



## **Additional information and Troubleshooting**

A great deal of documentation on both the RS-232 and RS-485 standard is available on the Internet. This section attempts to cover some commonly needed details in setting up communication systems.

### **RS-422**

This is the 5-wire full-duplex version of RS-485. The four data lines have two naming conventions, and are usually labeled as follows: Tx+ or TxA, Tx- or TxB, Rx+ or RxA, Rx- or RxB. Tie the Tx+ and Rx+ lines together to create the RS-485 A channel, and the Tx- and Rx- to create the B channel.

### **RS-232 “compatible”**

Many devices, including many laptop computers, have serial ports listed as RS-232 “compatible”. These ports typically use low-power signals that may or may not meet the RS-232 specifications. As noted in the spec charts, the ports can sometimes supply as little as 10mA, which is insufficient to communicate with many industrial devices, such as the servo. There are accessories available for laptops and other devices to provide industrial-quality serial ports. These are usually available as conversions from other ports, such as USB, PCMCIA, or internal ISA/PCI type cards. These devices can also sometimes provide RS-485 ports.

### **QCI Accessories and RS-485 Termination**

Many of the communication accessories available from QCI include the basic termination resistors built in. The QCI RS-232 to RS-485 converter (QCI-RS485) provides both active and passive termination, selectable by jumpers. With smaller networks, this may be all the termination required. For larger networks, the QCI Optical Isolation Module (QCI-OPTM-24) and Training/Network Breakouts (QCI-BO-T, QCI-BO-N) provide optional (connected by jumper or solder blobs) node termination. The SilverDust IGB provides termination with the addition of a jumper between terminals; see the SilverDust data sheet for details.

## Advanced Serial Communication Commands

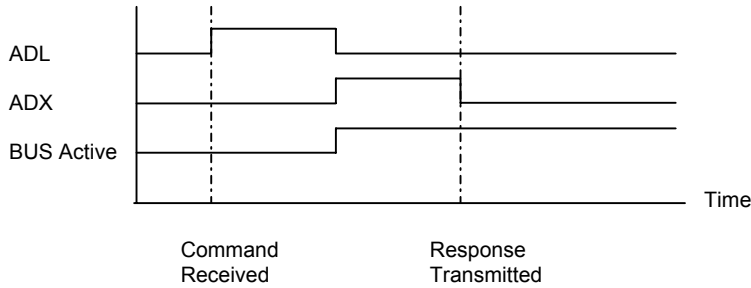
The following commands are for advanced users only with a thorough knowledge of serial communications.

### ADX:ACK Delay Extended

See Also: ADL:ACK Delay

#### Description

ADX sets a time delay in which the driver is actively driving the RS-485 serial bus at a "Marking" level before sending a response (i.e. ACK or data). ADX starts after ADL ends.



This command allows a passively terminated 485 bus to settle to the inter-character state prior to sending any response data.

The parameter is a count that equates to a number of 120 uSec "ticks". For greater resolution, use a negative Delay Count to specify 40 uSec ticks.

A two character delay allows the receiving UART to recover from an unspecified bus level (passive termination results in unknown state when no driver on the bus is active).

Baud Rate	ADX Delay	Delay (ms)
115200	-5	0.2
76800	-8	0.32
57600	-10	0.4
38400	-15	0.6
19200	-29	1.16
9600	-58	3.4

To effectively use the SilverDust on a passively terminated 485 bus, the Master must also assert the bus to the marking state for approximately 2 character periods prior to the command packet. Alternatively, two null characters (0xFF) may be sent prior to the command packet to allow the UART in the SilverDust to recover from a break/noisy line condition when no drivers are driving a passively terminated bus.

**Command Info**

Command	Command Type/Num	Parameters	Param Type	Parameter Range
ADX SN n/a SD 08	Program Class D 64 (0x40) 2 words Thread 1&2	Delay Count in ticks 1 tick= 120 uSec For negative values: 1 tick = 40 uSec	S16	-32767 to 21845

**Example**

ADX for 4 40uSec ticks

@16 64 -4 (CR)

**Response**

ACK only

**QuickControl Example**

