JNIOR Communications Protocol

Dated October 21, 2009
(v3.4 and later)

Notice Every effort was made to make this document as accurate and useful as practical at the time of writing. INTEG Process Group reserves the right to alter the information presented herein as well as the function of the JNIOR product at any time without prior notice. All information is subject to change.

Trademarks Trademarks are the property of their respective holders.
1-Wire is a registered trademark of Dallas Semiconductor.

INTEG Process Group, Inc.
2919 East Hardies Road, First Floor
Gibsonia, PA 15044

www.integpg.com

JNIORsales@integpg.com

PH (724) 933-9350
FAX (724) 443-3553

Contents

Revision History

Date	OS Version	Change Description
6/3/2005	v2.01.655	Added the ability to request the Usage Meter content using the Status Request Message (Message Type 5 – Request #2), the ability to reset input and output Usage Meters using the Command Message (Message Type 10 – Commands #8 and #9), and the Usage Meter Response message (Message Type 8). The latter reporting all Usage Meters in millisecond long format.
7/18/2005	v2.02.3	Added JNIOR Protocol functions to return lists of Registry sections and keys. A ListRegistry (Message Type 16) request is used to obtain a list of the entries for any node in the Registry. Those are supplied by the ListRegistryResponse (Message Type 17). Added Reboot function to the Request (Message Type 5 – Request 3).
9/9/2005	v2.03.60	Added Read/Write/Subscribe Device protocol structure. Dropped support for the jr200. Added ability to disable/enable unsolicited Monitor Packet transmission to the Request message (type 5). Added block relay controls to the Command message (type 10) allowing relays to be commanded simultaneously. Support added for external sensor types Temperature (10), Dual Addressable Switch (12), Counters (1D), Quad A/D Converter (20), Temperature (28), and Digital Potentiometer (2C).
12/6/2005	v2.03.174	Added support for external EEPROM Memory (23). This component is used to store configuration information for INTEG I/O Net modules.
12/22/2005	v2.03.225	Added login extensions to LoginRequest (message type 126) supporting anonymous and encoded login options.
5/3/2006	v2.11.261	Extended Digital Input and Relay Output device Read Blocks to include the Usage Alarm status. This extends the each by one (1) byte.
6/9/2006	v2.11.269	Added device types for the 4-20ma and 10V analog modules.
9/17/2006	v2.12.51	Support added for DS2438 Smart Battery Monitor external device.
8/8/2007	v2.13.86	Added support for the 4-channel RTD analog temperature module.
4/8/2008	v2.14.17	Added support for the 4 relay digital output module with pulse capability.
3/20/2009	V3.1.3	ROUT9 – ROUT16 using 1 or 2 relay expansion boards was added to the Command Packet (type 10). Added Custom Command Support for communications from Ethernet applications using the JNIOR Protocol to external applications running on the JNIOR Added support for external device channel assignments in packets (types 29 – 32). Up to 2 external devices total may be connected.

Connection

The JNIOR protocol connection is connection oriented using the TCP/IP protocol. By default the connection is made over port 9200. This is configurable via the use of the `JniorServer/Port` registry key. Multiple connections can be made to the JNIOR. There is a performance hit for each connection made as well as the number of items subscribed to. Item subscriptions can be IO or registry keys.

Since 3.3. In addition to making a connection to the JNIOR protocol server, the JNIOR protocol server can be configured to make a connection to one remote host. This is configured through the `JniorServer/RemoteIP` and `JniorServer/RemotePort` registry keys. The connection is tried every 30 seconds until it is established. If the connection is lost then the connection attempts are resumed. This does not require a reboot.

Message Structure

Each of the following messages (regardless of its source) is transmitted with the following header. If the supplied CRC16 does not match that calculated from the included data then the message should be ignored.

Item	# of Bytes	Content	Defined Values
1	1	Start of header	0x01 (always)
2	2	Length (Short)	**N** - Count of data bytes in the message
3	2	CRC16 (Short)	CRC for the following message
4	**N**	Message as defined below	

Connection Maintenance (Keep Alive)

Connections that are not closed properly can linger. Under certain circumstances the server may not be aware that the client has improperly dropped the connection. This can happen if the client device or system is interrupted by the loss of power, reset or otherwise unplugged from the network.

The **JNIOR** continues to service such connections and if after 15 minutes there has not been any communications from the client (neither command nor acknowledgement) the connection will be dropped. This frees resources in the **JNIOR** system and readies the **JNIOR** for additional connections. It that time it is quite possible that the disconnected device or system has reconnected using a new socket.

In order to keep a quiet connection from dropping the client should transmit a single ACK byte (0x06) periodically. This is acceptable at any time when the **JNIOR** would otherwise be ready to receive a command from the client. The **JNIOR** must receive this ACK byte or some other message from the client within 15 minutes of any prior ACK or command message or the connection will be assumed dead and it will be dropped.

CRC16 Error Check

A 16-bit Circular Redundancy Check is used. This calculation is made using only the Message content (N bytes that follow the CRC16 entry).

As it is the implementer's option to ignore the CRC16 on incoming messages, it is also optional for outgoing messages. A CRC16 value of 0xFFFF (-1) will bypass the CRC verification in the **JNIOR**. This is highly useful during development where it may be desirable to first demonstrate function and latter insure accuracy by enabling the error check.

Since the **JNIOR** will happily ignore the CRC16 value if it is set to 0xFFFF (-1) the client should return the favor. A CRC16 value of 0xFFFF (-1) received from **JNIOR** should bypass the client’s error check. By design this should never occur as **JNIOR** carefully provides a valid CRC16 value with all transmissions. A symmetrical implementation is recommended for possible future compatibility.

It is highly recommended that the proper Circular Redundancy Check be implemented. An example Java program to generate the CRC is supplied later in this document.

Character Strings

All character strings are encoded with a single byte prefix defining the number of characters to follow. A US_ASCII character set is used. Where Strings appear in the following packets their length is indicated as being variable. In fact the encoded string uses one byte more than the number of characters. For example the string “Hello World” will be transmitted in order from left to right as follows:

0x0B	‘H’	‘e’	‘l’	‘l’	‘o’	‘ ‘	‘W’	‘o’	‘r’	‘l’	‘d’

Numeric Formatting

The header and messages may contain multiple-byte numeric values which are generally unsigned integer values although other formats may be used. The following formats are referenced:

byte Single byte representing

0x00 to 0xFF (hexadecimal)
0 to 255 (unsigned)
-128 to -127 (signed)

Used typically for string lengths.

short 2-Byte integer value representing

0x0000 to 0xFFFF (hexadecimal)
0-65535 (unsigned)
-32768 to 32767 (signed).

Used typically for message lengths, CRC16 error checks, integer counts and identifiers.

int 4-Byte integer value representing

0x00000000 to 0xFFFFFFFF (hexadecimal)
0 to 4294967295 (unsigned)
-2147483648 to 2147483647 (signed)

Used typically for long term tallies and time intervals.

long 8-Byte integer value representing

0x0000000000000000 to 0xFFFFFFFFFFFFFFFF (hexadecimal)
0 to 2^64-1 (unsigned)
-2^63 to 2^63-1 (signed)

Used typically for absolute time and date.

double 8-byte value in IEEE 754 64-bit double-precision binary floating-point format.

Care must be taken to properly read, write, promote and calculate integers supplied through this protocol using signed or unsigned values as appropriate. The automatic sign extension that might occur when casting a *byte* value into an *int* integer value or a *short* integer value into an *int* integer value can raise havoc with protocol processing.

Byte Ordering

When acquiring a multiple-byte numeric value the byte order as transmitted across the network is important. Clearly the bytes of an integer value when taken in reverse order will result in a dramatically different value. These numeric values must be properly assembled from the data being received byte-by-byte through the network connection. The **JNIOR** Protocol uses *big-endian* byte ordering.

> **big-endian**: adj.
> Describes a computer architecture in which, within a given multi-byte numeric representation, the most significant byte has the lowest address (the word is stored ‘big-end-first’). Most processors, including the IBM 370 family, the PDP-10, the Motorola microprocessor families, and most of the various RISC designs are big-endian. Big-endian byte order is also sometimes called network order.
>
> A format for storage or transmission of binary data in which the most significant bit (or byte) comes first. The term comes from "Gulliver’s Travels" by Jonathan Swift. The Lilliputians, being very small, had correspondingly small political problems. The Big-Endian and Little-Endian parties debated over whether soft-boiled eggs should be opened at the big end or the little end. [Source: The Network Working Group Internet Glossary RFC 1392]

Java data streams format these numeric values by writing the most significant byte first using big-endian as is common in the network world. Java applications and applets can easily manipulate values passed through the **JNIOR** Protocol. C/C++, Visual Basic and similar language programmers need to take extra care in constructing values as these languages use the *little-endian* byte order employed by Intel processors and the standard Personal Computer.

Device Identification (IDs)

Each JNIOR input and output can be envisioned as an independent device. This would also include any additional components connected to the external Sensor Port. Each device is assigned a permanent identification. This is an unsigned 64-bit (8 byte) value. For external devices the ID may be obtained from its labeling, documentation or through discovery using the EnumerateDevices request.

The IDs may convey additional information about the device. The least significant byte of the ID for those external devices connected through the sensor port provides an indication of the device type (one-wire devices). The internal device IDs (type FF) are assigned as follows:

jr310 Internal Devices	
Inputs	
din1	0x00000000000001FF
din2	0x00000000000002FF
din3	0x00000000000003FF
din4	0x00000000000004FF
din5	0x00000000000005FF
din6	0x00000000000006FF
din7	0x00000000000007FF
din8	0x00000000000008FF
Outputs	
rout1	0x00000000000101FF
rout2	0x00000000000102FF
rout3	0x00000000000103FF
rout4	0x00000000000104FF
rout5	0x00000000000105FF
rout6	0x00000000000106FF
rout7	0x00000000000107FF
rout8	0x00000000000108FF
rout9	0x00000000000109FF
rout10	0x0000000000010AFF
rout11	0x0000000000010BFF
rout12	0x0000000000010CFF
rout13	0x0000000000010DFF
rout14	0x0000000000010EFF
rout15	0x0000000000010FFF
rout16	0x00000000000110FF

Introduction to a JNIOR protocol Connection

This section is an introduction into the methods and logistics for communicating from an application residing on a personal computer (or other device) with the JNIOR 310 utilizing the JNIOR Protocol over an Ethernet network using TCP. This document is an introduction to the more detailed document describing the JNIOR Protocol. The JNIOR Protocol document describes the various communication packets that are sent back and forth between the JNIOR and an application over the Ethernet to read and write to the JNIOR I/O and other settings.

The JNIOR device acts as a server for the TCP communications. The default setting is for the JNIOR to listen and accept connections coming in over the Ethernet on port 9200. [NOTE: This port is a configurable setting that can be altered via the JNIOR registry. The registry key for this setting is JniorServer/Port. The registry key can be changed from the main JNIOR web page and/or via a Telnet window using the Registry editor. These are further described in the JNIOR Web Based Interface Manual and Command Line Communications Manual.] The remote application makes the initial contact with the JNIOR by creating a socket connection to the JNIOR. In order to receive a response from the JNIOR, you must either successfully login or disable login by modifying the appropriate JNIOR Registry setting. After a successful login, the JNIOR automatically sends out the Monitor packet with any changes in the I/O state. The Monitor packet contains the status of all the I/O and is described in the JNIOR Protocol document. The remote application can parse out this packet and get the desired information.

The following is a summary of what a programmer should initially expect when implementing the JNIOR protocol.

1. Establish a socket connection over port 9200 and login with a valid user name and password **(message type 126).** “jnior”, “jnior” is the default admin user name and password.
2. Based on the login user name and password, the JNIOR will reply with the login acknowledgement message **(message type 125)** alerting the application of the security level granted.
3. The JNIOR will issue a Monitor packet **(message type 1)**. This monitor packet has information about the JNIOR and the current I/O states.

Once connected and logged in, the application can communicate with the JNIOR by following the JNIOR protocol and issuing any of the commands. The login is necessary before any commands are sent to the JNIOR with the exception of registry reads **(message type 11)**.

Please note, that when you send an I/O command to the JNIOR, a Monitor packet **(message type 1)** does not automatically get returned. The Monitor packet **(message type 1)** only gets returned after I/O commands that actually cause the I/O states to alter. The Monitor packet is automatically issued with each change in I/O state unless the Monitor packet was disabled.

For those requiring finer control of the Ethernet packets, the user application can issue a command to the JNIOR disabling the Monitor packet using the Request packet **(message type 5)** for the current connection (it is not global for all connections, only for this particular connection because multiple connections can be made to the JNIOR at the same time). The user would then issue a command to Subscribe **(message type 15)** to various I/O points. The JNIOR will then notify this particular connection of the change in the subscribed to I/O points.

If you need to continuously interact with the JNIOR I/O, it is better to leave the connection open. However, the JNIOR has a 15 minute timeout such that if it doesn’t see any activity on a particular connection, it will close the connection. This is a failsafe so that unused connections do not build on the JNIOR should the Ethernet link be physically broken or the other application stop. In order to keep the connection active, a “keep alive” packet needs to be sent to the JNIOR approximately ever 10 minutes. This is described in the JNIOR Protocol document on page 5 – Connection Maintenance (Keep Alive).

Example Packets

The following example packets were captured using WireShark, formerly Ethereal. WireShark is an exceptional tool for debugging the implementations of communication protocols. This gives you a sample of the format of the packets you will be receiving and sending.

Monitor (message type 1)

```
01 00 60 68 85 01 0e 6a  72 33 31 30 20 76 32 2e ..`h...j r310 v2.
31 34 2e 31 37 00 00 00  00 00 00 00 00 00 00 00 14.17... ........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 01 ........ ........
19 33 ca 9f eb                                   .3...
```

Login Request Packet (message type 126)

```
01 00 0d 60 b7 7e 05 6a  6e 69 6f 72 05 6a 6e 69 ...`.~.j nior.jni
6f 72                                            or
```

Login Response Packet (message type 125)

```
01 00 02 f0 20 7d 80                             .... }.
```

Read Registry Keys Packet (message type 11)

```
01 00 13 be 61 0b 00 01  00 de 0d 24 53 65 72 69 ....a... ...$Seri
61 6c 4e 75 6d 62 65 72                          alNumber
```

Registry Response Packet (message type 12)

```
01 00 0f 9e d2 0c 00 01  00 de 09 31 30 35 31 30 ........ ...10510
30 33 32 38                                      0328
```

Message Specification

A number of messages (sometimes referred to as packets) are available as payload within the **JNIOR** Protocol. These messages will be either transmitted by the **JNIOR** or received by **JNIOR** following the Message Header. The following are available.

Message Type	Description
1	**Monitor Packet** [1] JNIOR sends this packet when a connection is logged in and then afterwards whenever the status changes (input or output changes state).
2	**External Monitor Packet** [1] JNIOR sends this packet when the digital expansion I/O is being used. This packet is sent whenever the status of any digital I/O (internal or external) changes.
5	**Request** [1] Requests action from JNIOR. JNIOR replies with the appropriate response.
6	**DateTime Response** JNIOR sends this packet in response to a DateTime Status request.
7	**Set Clock Command** [1] Provides a means to set the JNIOR clock.
8	**Usage Meter Response** Reports the high resolution usage meter content.
10	**Command Packet** [1] This message is sent to the JNIOR to request various actions such as, for example, a change in output state.
11	**ReadRegistryKeys** A request sent to JNIOR listing the Registry Keys for which values are needed. The client assigns IDs to each key to facilitate processing the response.
12	**ReadRegistryKeys Response** The list of Registry Key values along with the associated ID sent by JNIOR in response to a ReadRegistryKeys request. Note that values are returned ONLY for existing keys. It is the client's responsibility to assign default values for those keys not present. This message is also sent unsolicited when Registry subscriptions are used.
13	**WriteRegistryKeys** [2] A list sent to JNIOR containing pairs of Registry Keys and values. JNIOR updates the Registry with the new Key content.
14	**WriteRegistryKeys Response** JNIOR acknowledges a WriteRegistryKeys request with this packet.
15	**SubscribeRegistryKeys** A request sent to JNIOR listing the Registry Keys of which values are needed. The format of this request is identical to the ReadRegistryKeys request and JNIOR sends the ReadRegistryKeys Response. Subsequently should the values for any of those keys change, JNIOR will send another ReadRegistryKeys Response packet with the updated values. The subscriptions are cancelled when the connection is closed.
16	**ListRegistry** [2] A request sent to JNIOR to obtain the list of Registry Sub-sections and key names present at the specified section node.
17	**ListRegistryResponse** JNIOR responds to a ListRegistry request with this message.
21	**ReadDevices** [1] A request sent to JNIOR to obtain device status/content for one or more device IDs. Both internal and external I/O is supported.
22	**ReadDevicesResponse** Message contains the device status/content for each requested by ReadDevices. This will also be sent unsolicited when SubscribeDevices is used.
23	**WriteDevices** [1] Provides new status/content for one or more devices. This is used to command outputs or to configure devices.

Message Type	Description
24	**WriteDevicesResponse** JNIOR acknowledges a WriteDevices request with this packet.
25	**SubscribeDevices** [1] A request sent to JNIOR requesting the status/content of one of more devices. The format of this request is identical to the ReadDevices request and JNIOR sends the ReadDevices Response. Subsequently should the status for any of those devices change, JNIOR will send another ReadDevicesResponse packet with the updated status/content. The subscriptions are cancelled when the connection is closed.
26	**EnumerateDevices** [2] A request sent to JNIOR for an enumeration of available devices.
27	**EnumerateDevicesResponse** JNIOR responds to an EnumerateDevices with a list of active device IDs present in the current hardware configuration. This contains both internal and external devices.
28	**UnsubscribeDevices** This will unsubscribe the given devices from the current connection
29	**GetExternalValue** Each connected external device can be enumerated by the JNIOR OS and assigned channel numbers. This command allows a user to request a value by channel number for a device type
30	**GetExternalValueResponse** JNIOR responds with the scaled value of the desired IO. The scaling information can be set in the registry or by the JNIOR Web Page. Outputs are returned on a scale of 0 – 100%
31	**SetExternalValue** Each connected external device can be enumerated by the JNIOR OS and assigned channel numbers. This command allows a user to set a value by channel number for a device type. Outputs are entered on a scale of 0 – 100%
32	**SetExternalValueResponse** JNIOR responds with the scaled value of the desired IO. The scaling information can be set in the registry or by the JNIOR Web Page. The returned value mimics the set value.
125	**Login Acknowledgement** Sent by JNIOR in response to a LoginRequest.
126	**LoginRequest** [3] Provides a username and password (clear text) for login. The user must be valid previously assigned.

[1] By default requires successful Login.

[2] By default requires successful Login by administrator.

[3] Requirement may be disabled using the JniorServer/Login Registry key.

Monitor Packet – Message Type 1

Note that the Monitor Packet is NOT fixed-length. The version string is variable length and will change with software version. Therefore, the location of any particular I/O status relative to the start of the packet will vary. The location of the I/O information may be calculated using the length of the version string which is always the second byte in the packet. Recall that strings are sent with a preceding length byte.

Item	# of Bytes	Content	Defined Values
1	1	Message Type	(1 = Monitor packet from **JNIOR**)
2	Variable	**JNIOR** Software Version (String)	"jrNNN #.##.####" where NNN indicates the Model Number and # digits in the current OS version.
Following Block (Items 3-7) Repeated For Each DIN1 – DIN8			
3	1	Present State	0 = Off, 1 = On
4	1	Alarm State	1 = alarm
5	4	Count (int)	
6	1	Count Alarm 1	1 if Count exceeds Alarm 1
7	1	Count Alarm 2	1 if Count exceeds Alarm 2
Following Item (8) Repeated For Each ROUT1 – ROUT8			
8	1	Present State	0 = Open, 1 = Closed
9	8	Date and Time (long)	Number of milliseconds since January 1, 1970 00:00:00 GMT

External Monitor Packet – Message Type 2

The External Monitor Packet is only available when external digital expansion I/O modules are used. The packet is sent whenever the status of any digital I/O (internal or external) changes.

For the Count value, the JNIOR automatically determines the number of inputs or outputs available via the Expansion Modules based on the number and type of modules connected and operating properly.

The associated Relay Output number (9 through 16) is determined by the JNIOR based on the order the modules are recognized by the JNIOR on the initial boot-up. Here are several scenarios:

- If the user connects 1 Relay Output Expansion Module to the Sensor Port and then applies power to the JNIOR (or reboots it) the JNIOR will assign the 4 relay outputs as outputs 9 – 12.
- If the user then adds a second Relay Output Expansion Module and reboots the JNIOR, the JNIOR will assign the outputs on the second module as 13 – 16.
- If the user disconnects the first module and reboots the JNIOR, the JNIOR will continue to maintain the addresses of the second module as outputs 13 – 16.
- If the user re-connects the original first expansion module (or a different one) and reboots, the JNIOR will assign these outputs as 9 – 12.
- If the user unplugs all expansion modules and reboots, the numbering process will start over again the next time an expansion module is added.

This method was implemented to allow the user to add new modules or change failed modules without requiring any software changes.

The relay output numbering sequence being used for each expansion module is stored and displayed as a Registry Key in the JNIOR. For the Relay Output Expansion Modules, the keys are viewed in the IO/Outputs registry folder.

Item	# of Bytes	Content	Defined Values
1	1	Message Type	(2 = External Monitor packet from **JNIOR**)
2	1	Count	The number of digital INPUTS on the Expansion Modules. **FUTURE USE** so value should be 0.
Following Item (3) Repeated For Each DIN9 – DIN16 (FUTURE Digital Inputs)			
3	1	Present State	0 = Off, 1 = On (FUTURE)
4	1	Count	The number of relay OUTPUTS on the Expansion Modules. (value will be 4 or 8)
Following Item (5) Repeated For Each ROUT9 – ROUT16 (Digital Outputs)			
5	1	Present State	0 = Open, 1 = Closed
6	8	Date and Time (long)	Number of milliseconds since January 1, 1970 00:00:00 GMT

Request – Message Type 5

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	5 = Request packet
2	2	Request (short)	0 = Date and Time 1 = Monitor Packet Request 2 = Usage Meter Request 3 = Reboot Request 4 = Disable Monitor Packets 5 = Enable Monitor Packets
3	4	Interval (int) Required for Monitor Packet Request	Defines an interval in milliseconds. 0 indicates infinity.

The Interval field is optional and may or may not be included in messages. It is used with the Monitor Packet Request to specify a new interval in milliseconds at which Monitor Packets will be sent. By default a Monitor Packet is transmitted upon initial login (or connection if login disabled) and whenever the states of inputs/outputs have changed.

If external digital I/O modules are being used, whenever the user sends the Monitor Packet Request, the External Monitor Packet will also be sent.

Request 3 is available to Administrator logins. There is no response. The JNIOR communications connection is immediately terminated and a shutdown commences. You may reconnect in about 60 seconds.

Date & Time Response – Message Type 6

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	6 = DateTime response packet
2	8	Date and Time (long)	Number of milliseconds since January 1, 1970 00:00:00 GMT

Set Clock Message – Message Type 7

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	7 = Set Clock Message
2	8	Date and Time (long)	Number of milliseconds since January 1, 1970 00:00:00 GMT

Usage Meter Response – Message Type 8

Usage Meters report the number of milliseconds that the corresponding input or relay output has been in the “on” or “closed” state respectively.

Item	# of Bytes	Content	Defined Values
1	1	Message Type	8 = Usage Meter Response
2	8	din1 Usage Meter (long)	milliseconds “on”
3	8	din2 Usage Meter (long)	milliseconds “on”
4	8	din3 Usage Meter (long)	milliseconds “on”
5	8	din4 Usage Meter (long)	milliseconds “on”
6	8	din5 Usage Meter (long)	milliseconds “on”
7	8	din6 Usage Meter (long)	milliseconds “on”
8	8	din7 Usage Meter (long)	milliseconds “on”
9	8	din8 Usage Meter (long)	milliseconds “on”
10	8	rout1 Usage Meter (long)	milliseconds “closed”
11	8	rout2 Usage Meter (long)	milliseconds “closed”
12	8	rout3 Usage Meter (long)	milliseconds “closed”
13	8	rout4 Usage Meter (long)	milliseconds “closed”
14	8	rout5 Usage Meter (long)	milliseconds “closed”
15	8	rout6 Usage Meter (long)	milliseconds “closed”
16	8	rout7 Usage Meter (long)	milliseconds “closed”
17	8	rout8 Usage Meter (long)	milliseconds “closed”
18	8	Date and Time (long)	Number of milliseconds since January 1, 1970 00:00:00 GMT

Command Packet – Message Type 10

The Command Packet structure varies depending on the parameter requirements.

<table>
<tr><th>Item</th><th># of Bytes</th><th>Content</th><th>Defined Values</th></tr>
<tr><td>1</td><td>1</td><td>Message Type (byte)</td><td>10 = Command packet</td></tr>
<tr><td>2</td><td>1</td><td>Action (byte)</td><td>1 = close relay output
2 = open relay output
3 = toggle relay output
4 = reset input latch
5 = clear input counter
8 = clear input usage meter
9 = clear output usage meter</td></tr>
<tr><td>3</td><td>2</td><td>Channel (short)</td><td>selected digital input or relay output as appropriate.
<table>
<tr><th>Channel</th><th>2 Bytes</th><th>jr310</th></tr>
<tr><td>1</td><td>00 01</td><td>din1, rout1</td></tr>
<tr><td>2</td><td>00 02</td><td>din2, rout2</td></tr>
<tr><td>3</td><td>00 03</td><td>din3, rout3</td></tr>
<tr><td>4</td><td>00 04</td><td>din4, rout4</td></tr>
<tr><td>5</td><td>00 05</td><td>din5, rout5</td></tr>
<tr><td>6</td><td>00 06</td><td>din6, rout6</td></tr>
<tr><td>7</td><td>00 07</td><td>din7, rout7</td></tr>
<tr><td>8</td><td>00 08</td><td>din8, rout8</td></tr>
<tr><td>9</td><td>00 09</td><td>rout9</td></tr>
<tr><td>10</td><td>00 0A</td><td>rout10</td></tr>
<tr><td>11</td><td>00 0B</td><td>rout11</td></tr>
<tr><td>12</td><td>00 0C</td><td>rout12</td></tr>
<tr><td>13</td><td>00 0D</td><td>rout13</td></tr>
<tr><td>14</td><td>00 0E</td><td>rout14</td></tr>
<tr><td>15</td><td>00 0F</td><td>rout15</td></tr>
<tr><td>16</td><td>00 10</td><td>rout16</td></tr>
</table>
NOTE: Channels 9 – 16 require the 4 Relay Output Expansion Modules</td></tr>
</table>

The Command Packet specifying a **pulse** requires an additional parameter. This command will pulse the given output high for the specified duration. Note: If you wish to pulse an output low you must use the block pulse action parameter as part of the command packet.

<table>
<tr><th>Item</th><th># of Bytes</th><th>Content</th><th>Defined Values</th></tr>
<tr><td>1</td><td>1</td><td>Message Type (byte)</td><td>10 = Command packet</td></tr>
<tr><td>2</td><td>1</td><td>Action (byte)</td><td>6 = pulse relay output</td></tr>
<tr><td>3</td><td>2</td><td>Channel (short)</td><td>selected digital input or relay output as appropriate.
<table>
<tr><th>Channel</th><th>jr310</th></tr>
<tr><td>X
Where x = 1 through 16 (see previous table)</td><td>routx</td></tr>
</table></td></tr>
<tr><td>4</td><td>4</td><td>Pulse Duration (Int)</td><td>Pulse length in milliseconds</td></tr>
</table>

The Command Packet that is used to perform **block I/O** (changing relay states simultaneously) requires a combination of a mask byte to select the affected outputs and a byte defining the resulting state for those outputs. The changes may optionally be pulsed.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	10 = Command packet
2	1	Action (byte)	7 = block pulse relay states 10 = block change relay states
3	1	Channel Select Mask	Byte indicating channels affected by this command. Bits set to 1 will be changed. LSB represents channel 1 and MSB channel 8.
4	1	Relay State	Defines the state of the relays affected. A 1 indicates closed and a 0 open. LSB represents channel 1 and MSB channel 8.
5	4	Pulse Duration (int) Required for Pulse only	Pulse length in milliseconds

A maximum of 31 pulse requests may be queued at any one time. Each pulse will complete in its entirety before the next is begun. Relays return to their original state upon completion.

The Command Packet that is used with the **Relay Output Expansion Modules** to perform **block I/O** (changing relay states simultaneously) requires two bytes to implement the mask byte that selects the affected outputs and two bytes to define the resulting state for those outputs. This Command Packet can thus handle the 8 internal relay outputs and up to 2 external Relay Output Expansion Modules for a total of 16 relay outputs. The changes may optionally be pulsed. The JNIOR uses the packet length to determine which of the two Command Packets for pulsing are being sent because of the different lengths for the channel select mask and the relay state mask.

IMPORTANT: Please make sure that the length field is correct in the message header.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	10 = Command packet
2	1	Action (byte)	7 = block pulse relay states 10 = block change relay states
3	2	Channel Select Mask (short)	Byte indicating channels affected by this command. Bits set to 1 will be changed. LSB represents channel 1 and MSB channel 16.
4	2	Relay State (short)	Defines the state of the relays affected. A 1 indicates closed and a 0 open. LSB represents channel 1 and MSB channel 16.
5	4	Pulse Duration (int) Required for Pulse only	Pulse length in milliseconds

A maximum of 31 pulse requests may be queued at any one time. Each pulse request on the internal outputs will complete in its entirety before the next request is begun. The external relay outputs differ from the internal relay outputs in that the pulse duration can be changed by sending a new command and new pulse commands can be initiated on the other expansion relay outputs while one of the expansion relay outputs is already executing a pulse command. Relays return to their original state upon completion.

ReadRegistryKeys – Message Type 11

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	11 = ReadRegistryKeys packet
2	2	Count (short)	The number of Registry Keys requested
Following Block (Items 3 & 4) Repeated For Each Requested Key			
3	2	Unique ID (short)	A unique integer identifying the Registry Key.
4	Variable	Registry Key (String)	The Registry Key identification.

ReadRegistryKeys Response – Message Type 12

This message will be received in response to either the ReadRegistryKeys or SubscribeRegistryKeys request. When the SubscribeRegistryKeys request has been used this response message may be subsequently returned (unsolicited) whenever a subscribed key value changes. If a requested Registry Key does not exist, or in the case of a subscribed key and the key has been removed, this message will include the key’s Unique ID followed by a null string (string of length zero). No matter how the Registry Key is requested it will be returned at least once whether or not it exists in the Registry.

The returned key value is formatted as it appears in the jnior.ini file with multiple values (if present) separated by commas with double quotation marks and escaping used as required. It is recommended that the user’s application parse the returned string to retrieve the desired value. A key that may usually contain only one value would appear properly if used directly in many cases but if a second element happens to be added to the JNIOR Registry both elements may be inappropriately used unless properly handled.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	12 = ReadRegistryKeys Response packet
2	2	Count (short)	The number of Registry Key Values to follow
Following Block (Items 3 & 4) Repeated For Each Key Value			
3	2	Unique ID (short)	The unique identifier for the associated Registry Key copied from the ReadRegistryKeys request packet.
4	Variable	Registry Key Value (String)	The Registry Key value as obtained from the Registry.

WriteRegistryKeys – Message Type 13

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	13 = WriteRegistryKeys packet
2	2	Count (short)	The number of Registry Keys and Values to follow
Following Block (Items 3 & 4) Repeated For Each Key-Value Pair			
3	Variable	Registry Key (String)	The Registry Key Identification.
4	Variable	Registry Key Value (String)	The Registry Key value.

WriteRegistryKeys Response – Message Type 14

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	14 = WriteRegistryKeys Response packet
2	2	Count (short)	The number of Registry Keys successfully written.

SubscribeRegistryKeys – Message Type 15

Note: With the exception of the Message Type byte this packet is physically identical to the ReadRegistryKeys request (Message Type 11). JNIOR responds with a ReadRegistryKeys Response (Message Type 12). A subscription is entered for the connection and an additional ReadRegistryKeys Response packet will be spontaneously transmitted whenever a subscribed key changes value.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	15 = SubscribeRegistryKeys packet
2	2	Count (short)	The number of Registry Keys requested
Following Block (Items 3 & 4) Repeated For Each Requested Key			
3	2	Unique ID (short)	A unique integer identifying the Registry Key.
4	Variable	Registry Key (String)	The Registry Key identification.

ListRegistry – Message Type 16

Registry keys imply a structure much like the directory structure of the file system. Each *node* of the tree in the structure may contain the name of a Registry Key and the name of any sub-sections (folders). The ListRegistry request is used to obtain a list of sub-section and key names for any given node.

The Registry node is specified without leading '/' or trailing '/' and specifies the complete path to the specific Registry section from the Registry Root. The Registry Root is specified with a null or empty string (zero length).

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	16 = ListRegistry packet
2	Variable	Registry Node (String)	Specifies the sub-section or node of the Registry for which the list is requested.

ListRegistryResponse – Message Type 17

JNIOR responds to a ListRegistry request with a ListRegistryResponse in all cases even if the Registry section or node is empty or does not exist. The response contains the list of local entries. If an entry represents a node with further substructure, the returned name with include a trailing '/'. Otherwise the entry represents a Registry Key and the value of that Key may be obtained with the ReadRegistryKey or SubscribeRegistryKey requests.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	17 = ListRegistryResponse packet
2	2	Count (short)	The number of Registry names to follow. Zero (0) if there are no entries
Following Item 3 Repeated For Each Name Value			
3	Variable	Registry Name Value (String)	The Registry Name as it exists at the specified node. Nodes with further substructure are indicated by a trailing '/'.

The Registry names are returned in no specific order.

ReadDevices – Message Type 21

This requests the status/content for each of the devices listed. The returned status/content is device dependent.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	21 = ReadDevices packet
2	2	Count (short)	The number of Device IDs to follow.
Item #3 Repeated for the Count indicated			
3	8	Device ID (unsigned long)	

ReadDevicesResponse – Message Type 22

This message will be received in response to either the ReadDevices or SubscribeDevices request. When the SubscribeDevices request has been used this response message may be subsequently returned (unsolicited) whenever a subscribed device status changes. What actually constitutes a device status *change* is device dependent. In some cases that may be configurable.

The returned byte array contains a structure that is also device dependent. If a requested Device does not exist this message will include the Device ID followed by an empty byte array.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	22 = ReadDevicesResponse packet
2	2	Count (short)	The number of Device Reports to follow.
Following Items 3 - 5 Repeated For Each Device Report			
3	8	Device ID (unsigned long)	
4	2	Length (short)	Length in bytes of the following Device Block.
5	Variable	Device Block. (byte array)	Contains device dependent structure.

The Devices are reported in no specific order.

Read Device Block Structures

The byte array returned by the ReadDevicesResponse contains a structure that is device dependent. This varies depending on the type of device and its capabilities. The following defines those structures for the internal devices and known external devices. Note that the size of the byte array returned may vary with each ReadDevicesResponse and should no be assumed to be constant for any particular device.

jr310 Digital Inputs (din1 – din8) Read Block

Item	# of Bytes	Content	Defined Values
1	1	Present State	0 = Off, 1 = On
2	1	Alarm State	1 = alarm
3	4	Count (int)	
4	1	Count Alarm 1	1 if Count >= Alarm1 setpoint
5	1	Count Alarm 2	1 if Count >= Alarm 2 setpoint
6	8	Usage Meter (long)	milliseconds “on”
7	1	Usage Alarm	1 if Usage >= Alarm setpoint
17 bytes Total Length			

jr310 Relay Outputs (rout1 – rout8) Read Block

Item	# of Bytes	Content	Defined Values
1	1	Present State	0 = Open, 1 = Closed
2	8	Usage Meter (long)	milliseconds “on”
3	1	Usage Alarm	1 if Usage >= Alarm setpoint
10 bytes Total Length			

External Sensor Type 10 – Temperature Probe Read Block

Item	# of Bytes	Content	Defined Values
1	8	Temperature (double)	Current temperature in degrees Celsius. Conversions take from 500 to 750 milliseconds. Resolution is 0.0625 degrees Celsius.
8 bytes Total Length			

External Sensor Type 12 – Dual Addressable Switch Read Block

Item	# of Bytes	Content	Defined Values
1	1	Channel A Level (byte)	0=low, 1=high
2	1	Channel B Level (byte)	0=low, 1=high
2 bytes Total Length			

External Sensor Type 1D – 4kbit RAM with Counter Read Block

Item	# of Bytes	Content	Defined Values
1	4	Channel A Count (unsigned int)	
2	4	Channel B Count (unsigned int)	
8 bytes Total Length			

Note: RAM access not available.

External Sensor Type 20 – Quad A/D Converter Read Block

Item	# of Bytes	Content	Defined Values
1	2	Channel A (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFFF full scale.
2	2	Channel B (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFFF full scale.
3	2	Channel C (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFFF full scale.
4	2	Channel D (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFFF full scale.
8 bytes Total Length			

External Memory Type 23 – 512 Byte EEPROM Memory

Item	# of Bytes	Content	Defined Values
1	512	Memory content (byte array)	Contains Reserved and Free memory areas configured per the application.
512 bytes Total Length			

External Sensor Type 26 – Smart Battery Monitor Read Block

Item	# of Bytes	Content	Defined Values
1	1	Status Register **	See Write Block fir detail
2	1	Temperature LSB	Degrees C Combine to Signed Integer and divide by 256.0 for reading.
3	1	Temperature MSB	
4	1	Voltage LSB	Volts Combine to Unsigned Integer and divide by 100.0 for reading.
5	1	Voltage MSB	
6	1	Current LSB	Volts ** Combine to Signed Integer and multiply by 0.002441 for reading.
7	1	Current MSB	
8	1	Threshold **	read only
8 bytes Total Length			

** Refer to Dallas/Maxim DS2438 Datasheet for details.

External Sensor Type 28 – Temperature Probe Read Block

Item	# of Bytes	Content	Defined Values
1	8	Temperature (double)	Current temperature in degrees Celsius. Conversions take from 500 to 750 milliseconds. Resolution is 0.0625 degrees Celsius.
8 bytes Total Length			

External Sensor Type 2C – Digital Potentiometer Read Block

Item	# of Bytes	Content	Defined Values
1	1	Feature Register (byte)	(see below)
2	1	Control Register (byte)	(see below)
3	1	Wiper Position (byte)	
3 bytes Total Length			

Feature Register

D7	D6	D5	D4	D3	D2	D1	D0
PR		NWP		NP		WSV	PC

PC 0: logarithmic potentiometer element(s)
1: linear potentiometer element(s)

WSV 0: wiper position(s) non-volatile
1: wiper position(s) volatile

NP 00: device contains 1 potentiometer
01: device contains 2 potentiometers
10: device contains 3 potentiometers
11: device contains 4 potentiometers

NWP 00: 5-bit (32 positions)
01: 6-bit (64 positions)
10: 7-bit (128 positions)
11: 8-bit (256 positions)

PR 00: 5K Ohm resistance
01: 10K Ohm resistance
10: 50K Ohm resistance
11: 100K Ohm resistance

Control Register

D7	D6	D5	D4	D3	D2	D1	D0
X	CPC	X	X	IWN		WN	

WN 00: select potentiometer 1
01: select potentiometer 2
10: select potentiometer 3
11: select potentiometer 4

IWN 1’s complement of WN

CPC 0: charge pump OFF
1: charge pump ON

X don’t care

External Sensor Type FB – 4ROUT Digital Module Read Block

Item	# of Bytes	Content	Defined Values
1	1	Last Used Channel Select Mask	Bits D0 thru D3 correspond to Relay Outputs A thru D. (0 = No Change, 1 = Change)
2	1	Relay State	Bits D0 thru D3 correspond to Relay Outputs A thru D. (0 = Open, 1 = Closed)
3	2	Relay A Pulse Time Remaining. (unsigned short)	1 to 65535 milliseconds. (0 = static)
4	2	Relay B Pulse Time Remaining. (unsigned short)	1 to 65535 milliseconds. (0 = static)
5	2	Relay C Pulse Time Remaining. (unsigned short)	1 to 65535 milliseconds. (0 = static)
6	2	Relay D Pulse Time Remaining. (unsigned short)	1 to 65535 milliseconds. (0 = static)
10 bytes Total Length			

External Sensor Type FC – RTD Temperature Module Read Block

Item	# of Bytes	Content	Defined Values
1	2	Input Channel 1 (signed short)	Signed integer temperature in degrees Celsius X10.
2	2	Input Channel 2 (signed short)	Signed integer temperature in degrees Celsius X10.
3	2	Input Channel 3 (signed short)	Signed integer temperature in degrees Celsius X10.
4	2	Input Channel 4 (signed short)	Signed integer temperature in degrees Celsius X10.
8 bytes Total Length			

Note: Each channel requires the connection of a 2-wire or 3-wire PT100 RTD device. The value of 32767 (0x7FFF) is returned for unwired channels.

External Sensor Type FD – 10V Analog Module Read Block

Item	# of Bytes	Content	Defined Values
1	2	Input Channel 1 (short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
2	2	Input Channel 2 (short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
3	2	Input Channel 3 (short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
4	2	Input Channel 4 (short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
5	2	Output Channel 1 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
6	2	Output Channel 2 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
12 bytes Total Length			

Note: Inputs are +/- 10 volts. 0x0000 represents a -10V input and 0xFFF0 a +10V input. 0x8000 therefore represents the 0V input level. Outputs are 0 to 10 volts only. A setting of 0x0000 results in a 0V output and 0xFFF0 in +10V out. Only the most significant 12 bits are used for the output channels.

External Sensor Type FE – 4-20ma Analog Module

Item	# of Bytes	Content	Defined Values
1	2	Input Channel 1 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
2	2	Input Channel 2 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
3	2	Input Channel 3 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
4	2	Input Channel 4 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
5	2	Output Channel 1 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
6	2	Output Channel 2 (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
12 bytes Total Length			

Note: 0x0000 represents a 4 milliamp loop current and 0xFFF0 (thru 0xFFFF) represents 20 milliamps. Scaling is linear. Only the most significant 12 bits are used for the output channels.

WriteDevices – Message Type 23

This is used to change one or more device's status, configuration, or content. The requested change is conveyed as a byte array whose structure is device dependent.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	23 = WriteDevices packet
2	2	Count (short)	The number of Devices to be written.
Following Items 3 - 5 Repeated For Each Device			
3	8	Device ID (unsigned long)	
4	2	Length (short)	Length in bytes of the following Device structure.
5	Variable	Device Block (byte array)	Contains device dependent structure.

WriteDevices Response – Message Type 24

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	14 = WriteDevices Response packet
2	2	Count (short)	The number of Devices successfully written.

Write Device Block Structures

The byte array supplied with the WriteDevices message must contain the defined structure that is device dependent. This varies depending on the type of device and its capabilities. The following defines those structures for the internal devices and known external devices. Note that the *Flags* field is used to indicate which the items in the structure to be modified by the write request. The bit positions in *Flags* indicate the action and may contain any combination of bits allowing for any portion of the structure to be affected. The entire structure must be provided and those fields not being written are ignored.

jr310 Digital Inputs (din1 – din8) Write Block

Item	**# of Bytes**	**Content**	**Defined Values**
1	1	Flags (byte)	0x01 – Reset Count 0x02 – Write Count 0x04 – Reset Usage Meter
2	4	Count (int)	Required if Write Count flag set.
1 or 5 bytes Total Length			

jr310 Relay Outputs (rout1 – rout8) Write Block

Item	**# of Bytes**	**Content**	**Defined Values**
1	1	Flags (byte)	0x01 – Modify State 0x02 – Reset Usage Meter
2	1	Present State	0 = Open, 1 = Closed Required if Modify State flag set.
1 or 2 bytes Total Length			

External Sensor Type 12 – Dual Addressable Switch Write Block

Item	**# of Bytes**	**Content**	**Defined Values**
1	1	Channel A Level (byte)	0=low, 1=high
2	1	Channel B Level (byte)	0=low, 1=high
2 bytes Total Length			

External Memory Type 23 – 512 Byte EEPROM Memory

Item	# of Bytes	Content	Defined Values
1	2	Starting Address (short)	0 – 511
2	2	Count (short)	Count of bytes to be written. The sum of the Starting Address and the Count must not exceed 511.
3	Variable	Content to be written (byte array)	Array of Bytes to be written. Must be exactly the number of bytes defined by Count.

External Sensor Type 26 – Smart Battery Monitor Write Block

Item	# of Bytes	Content	Defined Values
1	1	Configuration Register **	See below
1 bytes Total Length			

** Refer to Dallas/Maxim DS2438 Datasheet for details.

Configuration Register

D7	D6	D5	D4	D3	D2	D1	D0
X	ADB	NVB	TB	AD	EE	CA	IAD

IAD	0: current A/D disabled 1: current A/D enabled
CA	0: CCA/DCA disabled (inaccessible) 1: CCA/DCA disabled
EE	0: CCA/DCA not shadowed 1: CCA/DCA shadowed
AD	0: Voltage reports general purpose input 1: Voltage reports supply voltage
TB	0: Temperature conversion complete 1: Temperature conversion in-process
NVB	0: Non-volatile memory not busy 1: Non-volatile memory busy
ADB	0: A/D conversion complete 1: A/D conversion in-process
X	Don’t care

External Sensor Type 2C – Digital Potentiometer Write Block

Item	**# of Bytes**	**Content**	**Defined Values**
1	1	Control Register (byte)	(see below)
2	1	Wiper Position (byte)	
2 bytes Total Length			

Control Register

D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
X	CPC	X	X	IWN		WN	

WN 00: select potentiometer 1
01: select potentiometer 2
10: select potentiometer 3
11: select potentiometer 4

IWN 1’s complement of WN

CPC 0: charge pump OFF
1: charge pump ON

X don’t care

External Sensor Type FB – 4ROUT Digital Module Write Block

Item	# of Bytes	Content	Defined Values
1	1	Channel Select Mask	Bits D0 thru D3 correspond to Relay Outputs A thru D. (0 = No Change, 1 = Change)
2	1	Relay States	Bits D0 thru D3 correspond to Relay Outputs A thru D. Only those relays selected in the Channel Select Mask are affected. (0 = Open, 1 = Closed)
3	2	Relay A Pulse Duration. (unsigned short)	1-65535 milliseconds. Enabled only if relay selected by the Channel Select mask. (0 = static/no pulse)
4	2	Relay B Pulse Duration. (unsigned short)	1-65535 milliseconds. Enabled only if relay selected by the Channel Select mask. (0 = static/no pulse)
5	2	Relay C Pulse Duration. (unsigned short)	1-65535 milliseconds. Enabled only if relay selected by the Channel Select mask. (0 = static/no pulse)
6	2	Relay D Pulse Duration. (unsigned short)	1-65535 milliseconds. Enabled only if relay selected by the Channel Select mask. (0 = static/no pulse)
10 bytes Total Length			

Note: Add 1-2 milliseconds to offset the mechanical response time of relays. Relays return to their prior state upon completion of a pulse except if a pulse is interrupted. In that case the new duration is started and the relay will return to its state prior to the original interrupted pulse.

External Sensor Type FD – 10V Analog Module Write Block

Item	# of Bytes	Content	Defined Values
1	2	Channel 1 Setting (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
2	2	Channel 2 Setting (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
4 bytes Total Length			

Note: 0x0000 results in a 0 volt output and 0xFFF0 in 10 volts out. Only the most significant 12 bits are used. Scaling is linear.

External Sensor Type FE – 4-20ma Analog Module Write Block

Item	# of Bytes	Content	Defined Values
1	2	Channel 1 Setting (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
2	2	Channel 2 Setting (unsigned short)	Raw 16-bit A/D reading 0x0000 to 0xFFF0 full scale.
4 bytes Total Length			

Note: 0x0000 results in a 4 milliamp loop current and 0xFFF0 in 20 milliamps. Only the most significant 12 bits are used. Scaling is linear.

SubscribeDevices – Message Type 25

Note: With the exception of the Message Type byte this packet is physically identical to the ReadDevices request (Message Type 21). JNIOR responds with a ReadDevices Response (Message Type 22). A subscription is entered for the connection and an additional ReadDevices Response packet will be spontaneously transmitted whenever a subscribed device's status changes. What constitutes a status *change* is device dependent and in some cases may be configurable. A device subscription remains in place until the connection is closed.

Note that a subscription typically returns the Read Device Block for a device when any content changes. The one exception being the A/D devices wherein noise levels guarantee that every sampling results in some change. The Type 20 Quad A/D Converter reports changes only when the delta exceeds a value roughly equivalent to 8 bits of resolution.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	25 = SubscribeDevices packet
2	2	Count (short)	The number of Device IDs to follow.
Item #3 Repeated for the Count indicated			
3	8	Device ID (unsigned long)	

EnumerateDevices – Message Type 26

Each JNIOR will contain internal I/O devices and possibly external I/O devices if the Sensor Port is available and additional devices are connected there. This request is used to obtain a list of active devices.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	26 = EnumerateDevices packet
2	1	Flags (byte)	0x01 – enumerate internal 0x02 – enumerate external 0x03 – enumerate all

EnumerateDevicesResponse – Message Type 27

JNIOR responds to an EnumerateDevices request with a EnumerateDevicesResponse. The response contains the list of the Device IDs available either internally, externally or both depending on the setting of the associated Flags bits. Note that JNIOR is prompted to scan its I/O device structures and physical networks and several seconds may pass before the response is returned.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	27 = EnumerateDevicesResponse packet
2	1	Flags (byte)	Flags provided in request.
3	2	Count (short)	The number of Devices found.
Following Item 4 Repeated For Each Device			
4	8	Device ID (unsigned long)	

The Device IDs are returned in no specific order.

UnsubscribeDevices – Message Type 28

This will unsubscribe the given devices from the current connection.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	28 = UnsubscribeDevices packet
2	2	Count (short)	The number of Device IDs to follow.
Item #3 Repeated for the Count indicated			
3	8	Device ID (unsigned long)	

GetExternalValue – Message Type 29

Each connected external device can be enumerated by the JNIOR OS and assigned channel numbers. This command allows a user to request a value by channel number for a device type.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	29 = GetExternalValue packet
2	1	Device Type (byte)	0x10 – Temperature Sensor 0x28 – Temperature Sensor 0xFB – 4 Relay Out Module 0xFC – RTD Module 0xFD – 10 Volt Module 0xFE – 4 – 20 ma Module
3	1	Input / Output (byte)	01 – Input Channel 02 – Output Channel
4	1	Channel Number (byte)	This can be device dependent. For example an RTD module has no outputs and temperature Sensors have 1 input.

GetExternalValueResponse – Message Type 30

JNIOR responds with the scaled value of the desired IO. The scaling information can be set in the registry or by the JNIOR Web Page. Outputs are returned on a scale of 0 – 100%.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	30 = GetExternalValueResponse packet
2	1	Device Type (byte)	The device type passed in
3	1	Input / Output (byte)	The IO selection passed in
4	1	Channel Number (byte)	The channel number passed in
5	8	Value (double)	The value of the selected channel on the desired device.

SetExternalValue – Message Type 31

Each connected external device can be enumerated by the JNIOR OS and assigned channel numbers. This command allows a user to set a value by channel number for a device type. Outputs are entered on a scale of 0% - 100%.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	31 = SetExternalValue packet
2	1	Device Type (byte)	0xFB – 4 Relay Out Module 0xFD – 10 Volt Module 0xFE – 4 – 20 ma Module
3	1	Channel Number (byte)	This can be device dependent. For example an RTD module has no outputs and temperature Sensors have 1 input.
4	8	Value (double)	The value to assign to the given devices channel

SetExternalValueResponse – Message Type 32

JNIOR responds with the scaled value of the desired IO. The scaling information can be set in the registry or by the JNIOR Web Page. The returned value mimics the set value.

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	32 = SetExternalValueResponse packet
2	1	Device Type (byte)	The device type passed in
3	1	Channel Number (byte)	The channel number passed in
4	8	Value (double)	The value of the selected channel on the desired device.

LoginRequest – Message Type 126

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	126 = LoginRequest packet
2	Variable	Username (String)	A valid username
3	Variable	Password (String)	The valid password for this user.

By default a successful login will be required to enable protocol operation. This requirement may be removed by setting JniorServer/Login to ‘disabled’ in the JNIOR Registry. In this case the protocol proceeds as if an administrator has logged in. Note that if login has been disabled a LoginRequest may still be used to effect a login if appropriate.

Anonymous Login

If the JniorServer/Anonymous key has been defined in the Registry an anonymous login will be allowed. An anonymous login request contains both blank (zero length) Username and blank Password strings. The anonymous login must be successfully performed to enable protocol operation. The Registry key defines the integer (0-254) User ID to be used for anonymous access. A value of 0 (zero) is recommended. User IDs of 128 or greater are equivalent to an administrator login. To disable anonymous usage the JniorServer/Anonymous key must not appear in the Registry (valid content or not).

Encoded Password Transfer

The UserName and Password in the above transactions are transferred in clear text. This means that someone able to monitor network traffic may view packet content and will be able to see your login information. This may be of concern when communicating with JNIOR over public networks.

Optionally one may encode the combined username:password string (for instance “jdoe:mypass”) using Base64 encoding as defined by IEC RFC 1521. This renders the login information in a format that is not easily read by humans. The base64 encoded login string is transferred as the Password and its use is signified by supplying a blank (zero length) Username string. Note that this a minimal step and by no means represents true security. It will however minimize the temptation associated with accidentally discovering a user’s password.

Login Acknowledgement – Message Type 125

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	125 = Login Acknowledgement response packet
2	1	User ID (byte)	The JNIOR uID assigned to the user or -1 if login failed. Note that uID less than zero (bytes 0x80 through 0xFE) indicate administrator privileges. A uID of 0 indicates a guest login (not always allowed).

The following custom commands allow an application on the Ethernet that is talking to the JNIOR via the JNIOR protocol to talk directly to an external application running on the JNIOR.

Custom Command Response – Message Type 254

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	254 = Custom Command response packet
2	1	Return Status (byte)	-1 = error. Other status codes will be external application defined.
3	2	Payload size (short)	This is the length of the payload that follows.
4	Variable	Payload (byte[])	This is an array of bytes that is the payload.

Custom Command – Message Type 255

Item	# of Bytes	Content	Defined Values
1	1	Message Type (byte)	255 = Custom Command packet
2	Variable	Command name (string)	This is the command that the external application running on the JNIOR would have registered with the OS.
3	1	Command Type (byte)	This is a byte that will be passed to the external application describing the payload to follow.
4	2	Payload size (short)	This is the length of the payload that follows.
5	Variable	Payload (byte[])	This is an array of bytes that is the payload.

CRC-16 Algorithm

A standard implementation of the CRC-16 (16-bit Cyclic Redundancy Check) is used. If a library function is not available for use the following algorithm will provide the calculation. This is a table look-up implementation and the table can either be used as supplied or generated on the fly at the programmer's option.

A Java implementation follows. This is readily adapted for use with C/C++, etc.

```
/* Static CRC16 lookup table. This table can be used as supplied or
 *  generated on the fly.
 */

// Omit if dynamically generated table used
private int[] crctab =    /* CRC lookup table */
{
 0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
 0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
 0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
 0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
 0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
 0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
 0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
 0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
 0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
 0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
 0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
 0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
 0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
 0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
 0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
 0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
 0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
 0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
 0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
 0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
 0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
 0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
 0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
 0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
 0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
 0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
 0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
 0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
 0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
 0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
 0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
 0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040
};

// Omit if static table used
private int[] crc_table = null; // dynamically generated table
```

```
/** Generates the CRC16 for the supplied byte array. Note that this
 *  will generate the lookup table on the fly the first time it is
 *  used. You may optionally use the static table. The initial CRC
 *  value should be 0.
 *
 * Note: ‘>>>’ is an unsigned right shift.
 */
public int crc16(byte[] data, int crc) {

    /* Omit this conditional block if the static version of the
     *  look-up table is to be used.
     */
    if (crc_table == null) { /* generate table */
        crc_table = new int[256];

        int n, i, c, d;
        for (n = 0; n < 256; n++) {
            c = 0;
            d = n;
            for (i = 0; i < 8; i++) {
                if (((c ^ d) & 1) != 0) c = (c >>> 1) ^ 0xA001;
                else c = c >>> 1;
                d = d >>> 1;
            }
            crc_table[n] = c;
        }
    } /* generate table */

    // Calculate the CRC16 of the data byte array.
    int i;
    for (i=0; i<data.length; i++)
        crc = ((crc >>> 8) & 0xff) ^ crc_table[(crc ^ data[i]) & 0xff];
    return crc;
}
```

Example Transactions

Here is an example login transaction for programming reference. This shows two complete messages byte-by-byte. The first is a Login Request and the second the Login Acknowledgement from **JNIOR**.

```
Msg Received by JNIOR:
    01 00 0d 60 b7 7e 05 6a 6e 69 6f 72 05 6a 6e 69 6f 72

Msg Sent by JNIOR:
    01 00 02 f0 20 7d 80
```

In the above the first 5 bytes of each transmission is the Message Header these are shown using **bold** characters. Each header starts with 0x01. This is always the case and all other byte values should be ignored. **JNIOR** may send the ACK 0x06 byte periodically in an attempt to demonstrate activity and to keep a connection alive. These should be ignored with the exception of the fact that they indicate that the **JNIOR** is still connected.

Following the Start of Header 0x01 byte the next two bytes form the *short* integer message length as transmitted used big-endian byte order. This would be 13 (0x000d) and 2 (0x0002) bytes respectively. This denotes the exact length of the message payload that follows the header. The message length may be zero (0x0000). Such a message could also be used as a keep-alive indication. The message should otherwise be ignored.

The last two bytes of the header represent the CRC16 and are 0x60b7 and 0xf020 respectively in this example. With one exception, messages in which the calculated CRC16 from the message payload does not match that reported in the header will be ignored by **JNIOR** and should be ignored by the user's application. The CRC16 calculation routine has been provided in a prior section.

The programmer may be anxious to demonstrate function prior to CRC implementation. This is understandable. It is for this purpose that the **JNIOR** will ignore the CRC16 check should the supplied value be exactly 0xFFFF. Although it will likely never happen, the programmer should return the favor and not perform the check upon reception of a message from the **JNIOR** with a 0xFFFF CRC value. So, for example, the following form of the above transaction is equally functional. Note where the 0xFFFF CRC has been used.

```
Msg Received by JNIOR:
    01 00 0d ff ff 7e 05 6a 6e 69 6f 72 05 6a 6e 69 6f 72

Msg Sent by JNIOR:
    01 00 02 f0 20 7d 80
```

In the above the messages the payload is shown in **bold** characters. The first payload is 13 bytes and the second just two bytes. Both are the proper length as defined by the length field in the header.

Each message serves a purpose and the very first byte defines the message type. The first message is Type 126 (0x7e) representing the LoginRequest and the second Type 125 being the associated LoginAcknowledgement. The login request message contains two strings of 5 character length. Note the string length appears first and the following characters represent 'jnior' the default username and also the default password (both being the same as shipped from the factory).

The acknowledgement returns the User ID which is 128 (0x80) in this case. The **JNIOR** would return 255 (0xFF) if the login failed. Note that User IDs in the range 128 (0x80) to 254 (0xFE) represent Administrative users and have rights to modify configuration and perform other functions.

The following transaction demonstrates a Registry key Subscription.

```
Msg Received by JNIOR:
    01 00 2c 2c 04 0f 00 03 00 00 0b 44 65 76 69 63 65
    2f 44 65 73 63 00 01 08 24 56 65 72 73 69 6f 6e 00
    02 0d 24 53 65 72 69 61 6c 4e 75 6d 62 65 72

Msg Sent by JNIOR:
    01 00 31 98 9a 0c 00 03 00 00 16 6a 72 33 31 30 20
    44 65 76 65 6c 6f 70 6d 65 6e 74 20 55 6e 69 74 00
    01 08 32 2e 30 31 2e 33 34 36 00 02 07 34 39 30 34
    30 30 34
```

This particular exchange requests the current value and notice of all future changes to the following three keys:

```
Device/Desc
$Version
$SerialNumber
```

And the appropriate values are returned in the response:

```
“jr 310 Development Unit”
“2.01.346”
“4904004”
```

Of these three Registry Keys only the first happens to be subject to change. If it were changed now through the Registry Editor an additional message would be sent by JNIOR containing the new value. This would be referenced against the key's Unique ID of 0 (0x0000). This *subscription* would be valid until the connection is closed. The detailed parsing of these two example transmissions will be left to the programmer.

CRC Test Strings

The following strings may be used to test the CRC16 algorithm. For each string the value should be returned as shown.

Text String	**CRC16 Value**
“0123456789”	0x443d
“ABCDEFG”	0x9e6c
“” (empty string – 0 bytes)	0x0000

Proper CRC16 values for more complex arrangements can be found in the example transmissions of the prior section.