JANOS Management Protocol (JMP)

1. Abstract

While the JNIOR Protocol remains a viable option for controlling and monitoring I/O on the JNIOR the JANOS Management Protocol or JMP (pronounced "JuMP") offers a single connection point wherein the JNIOR may be fully managed and monitored. This capability is new to the Series 4 JNIOR and requires JANOS v1.8 and later.

2. Introduction

2.1 Background

In order to remotely control the JNIOR you need the ability to obtain I/O status and to affect changes in I/O condition. In the earlier Series 3 JNIOR this was accomplished through the JNIOR Protocol[1] made available through a TCP/IP connection typically on port 9200. This is a documented binary protocol that requires special programming external to the JNIOR for its use. Care is also required to allow access to the specific port through routers and firewalls. Once successfully implemented the JNIOR Protocol not only provided I/O status and control mechanisms, it also opened access to the JNIOR Registry[2] and thereby the ability to configure and manage the product.

In addition to the JNIOR Protocol it was also necessary to access the JNIOR Command Line through Telnet in order to completely manage the product. Care again is required to allow access to the Telnet port (Port 23) through routers and firewalls. The Command Line is also accessible using a serial connection to the RS-232 port on the JNIOR and through the Dynamic Configuration Pages (DCP) Web interface . This *Console* connection provides tools for monitoring I/O status and affecting I/O conditions as well as use of various kinds of diagnostics. Furthermore in this environment the product can be fully configured in all aspects including the network parameters. In addition this is where application programs can be executed which extend the functionality of the JNIOR product.

Management of the JNIOR also requires the manipulation of files in the local file system. While files may be manipulated through Console connection transfer to/from an external system is done using FTP (File Transfer Protocol) or through drag and drop operations in the DCP. Again care must be taken to allow access to the FTP command port (Port 21) through routers and firewalls. FTP typically opens/accepts data connections which must also be accommodated by the network.

With the introduction of the Series 4 JNIOR running the JANOS operating system the various I/O and management requirements covered by these other protocols are additionally handled through a single Web Server connection. Access to the Web Server is typically through ports 80 and 443. The latter connection providing for TLS Transport Layer Security up to 256-bit. While these ports would also need to be accommodated by routers and firewalls this is a much more standard requirement and often routine request for IT personnel. This consolidation of functionality is accomplished using the Websocket Protocol. This can result in a fully functional browser-based dynamic website providing JNIOR monitoring and control. The example being the Dynamic Configuration Pages provided with the product. These Javascript[TM] based dynamic web pages have replaced the Java-based applets used by the Series 3 JNIOR products. This Websocket interface utilizes the very same JSON Objects associated with the JMP Server (as documented here) to achieve the desired performance.

The JANOS Management Protocol (JMP) exposes the underlying JSON interface previously only available through the Websockets connection. This is handled by the JMP Server.

2.2 JMP Protocol Overview

To access the JNIOR using the JANOS Management Protocol (JMP) the client need only make a TCP/IP connection to the JMP port. By default this is Port 9220. This port may be disabled or moved to another number through settings in the Registry. These settings can be made through the Dynamic Configurations Pages (DCP) using the browser or by using the Command Line Console. The JMP Server can support up to 16 simultaneous connections.

Once a successful connection is made to the JMP Server, messages may be exchanged. With one exception all messages conform to the JSON[4] format using the ASCII printable character set. The high-level message format must be as follows. These are to be 2-element JSON Array constructs.

```
[ number , object ]
```

Where *number* defines the exact size of the *object* in bytes excluding leading and trailing white-space if any. Leading and trailing white-space, which can include newline characters, may be present surrounding both the *number* value and the *object*. Here *object* must be a fully formed and valid JSON Object beginning with '{' and ending with '}'. Both these curly braces and characters in between are included in the length value defined by *number*. The leading '[' opening square bracket, ',' comma, and trailing ']' closing square bracket are required. The opening and closing JSON Object curly braces are also verified. If there is any violation to this

format the message will be simply ignored. There is no response or indication of error. All bytes outside of the square brackets are ignored as well.

A valid parsing strategy would be as follows:

1. Read and ignore bytes up to a '[' opening square bracket
2. Read and ignore white-space characters (space, tab, newline, etc.)
3. Accumulate a decimal length (must be digits 0-9, the result must be >= 2)
4. Read and ignore white-space
5. Read and confirm the presence of the ',' comma
6. Read and ignore white space
7. Extract the JSON Object of length defined by the numeric value
8. Read and ignore white-space
9. Read and confirm the ']' closing square bracket (no other character is acceptable)
10. Confirm that the JSON object is properly enclosed by '{' and '}'
11. Process the obtained object and repeat

It is important to note that the TCP/IP connection is a streaming channel and one or more network packets may be required to convey an entire message. Similarly a packet may include the final bytes of one message and those beginning the next. A reliable implementation will buffer incoming data until an entire message is received. Once the message is processed it is removed from the buffer leaving any additional data which will be required to form messages that follow.

JMP is not Master-Slave. Many requests do solicit a response but not all. There are also unsolicited messages produced by the server. These alert the client to I/O changes as well as many other events. The protocol allows you to specify any amount of META data with your request. That data is echoed in the associated response. This can be used to maintain synchronization between request and response. It is a very flexible means of synchronization and can be used to convey, for example, processing instructions for the response.

2.3 Security

Any protocol providing control and management functions must employ some form of security preventing unauthorized access and abuse. By default the JMP Server requires authentication (login). While the login requirement may be disabled it is not recommended. When the login is disabled an account must be specified for the anonymous login. This is configured through the Registry. We strongly urge you to accommodate the login requirement.

In addition to user authentication the JMP Server supports a TLSv1.2 secure connection. A secure connection is established by first connecting to the JMP Server port and issuing the following clear-text command exactly as shown below. This is the one exception to the 2-element JSON Array formatting mentioned earlier.

```
[STARTTLS]
```

Immediately following the ']' closing square bracket the JNIOR will begin a SSL/TLS negotiation. The client should expect to do the same. If successfully all further communications will be encrypted.

3. Connection

3.1 JMP Server

All communications utilize the 2-element JSON Array format for conveying valid JSON Objects. From this point forward in this document the 2-element JSON Array format will be assumed and we will only show the enclosed JSON Objects. The exchanged JSON Objects heretofore are referred to as "Messages".

To initialize communications the client should send a blank or empty message. The following is acceptable.

```
{
  "Message":""
}
```

This message properly formatted for JMP would be transmitted as follows. For the remainder of this document we will show only the JSON Object content. It will be assumed that proper JMP blocking is used in all cases.

```
[14,{"Message":""}]
```

The connection will proceed depending on the authentication requirements established by JNIOR configuration and the client environment for the connection.

With the login requirement the exchange will proceed as follows. In this example the client properly utilizes the supplied *Nonce* to properly calculate a digest inclusive of the login credentials for the username 'jnior'. Refer to this document's Appendix for details on the associated MD5 digest calculation. The response indicates successful login and that the account has *Administrator* and *Control* permissions. All Administrators have the ability to control the JNIOR. Not all accounts with *Control* permission are administrators.

We will use ">>>" to indicate objects transmitted by the client and "<<<" to indicate those transmitted by the server either in response or unsolicited. These are not part of the protocol. Recall that all of these JSON Objects are to be properly blocked into a 2-element JSON Array with the object length in communications.

```
>>> {
  "Message":""
}

<<< {
  "Message":"Error",
  "Text":"401 Unauthorized",
  "Nonce":"5d894efb48e1c3bc074fe78e7a5f"
}

>>> {
  "Auth-Digest":"jnior:65f2d1cb66ef63f7d17a764f3a2f2508"
}

<<< {
  "Message":"Authenticated",
  "Administrator":true,
  "Control":true
}
```

A "Monitor" message will likely immediately follow. This might even be received before the "Authenticated" message. That is the asynchronous nature of the connection. Please feel free to contact INTEG for assistance in implementing the digest calculation.

4. Messaging

This section describes the various bi-directional messages and replies supported by the JMP Server.

The JMP server implementation is not Master-Slave however there are a number of 'Requests' that have 'Responses' which is typical for such a server. In addition to this, unsolicited messages may be received from the server. These provide immediate notification for changes in I/O status and updates in configuration settings for instance. Any use of this implementation must handle the presence of unsolicited messages. Care is also required to pair responses with the associated requests as messaging order is not guaranteed. Optional Meta data supplied with a Request is returned with the Response unmodified. This can then be used to identify each response and the action it then requires.

4.1 Common Message Structure

All messages use JSON formatting. Each consists of a set of *members* enclosed by curly braces '{' and '}'. An empty set is acceptable '{}' although it would be ignored by the server and solicit no response. A set may consist of any number of members separated by commas. Each member represents a name/value pair where the *name* is separated from the *value* by a colon ':'. The value can be a *string*, *number*, *object*, *array*, *ture*, *false* or *null*. The members are referenced by name and therefore may appear in any order. An *array* however consists of 0 or more *elements* each of which are values separated by a commas and presented in sequence dependent order.

4.1.1 "Message" Member

JMP requires that each valid message contain a 'Message' member. This is a name/value pair where the *name* is exactly the string "Message" and the *value* separated by the colon be any one of the following.

- Client generated messages
  - "Status"
  - "Control"
  - "Registry List"
  - "Registry Read"
  - "Registry Write"
  - "Registry Write Encrypted"
  - "Enumerate Devices"
  - "Read Devices"

    - "Write Devices"
    - "Console Open"
    - "Console Stdin"
    - "Console Close"
- Server generated responses
    - "Registry List Response"
    - "Registry Response"
    - "Enumerate Devices Response"
    - "Read Devices Response"
    - "Write Devices Response"
    - "Console Response"
    - "Error"
    - "Authenticated"
- Server generated messages - unsolicited
    - "Monitor"
    - "Registry Update"
    - "Console Stdout"

Messages received by the server not containing a valid "Message" member are ignored. These will not cause an error or solicit any response.

4.1.2 "Meta" Message Member

The "Meta" message member is entirely optional and since its associated *value* may be an *object* it can contain any information and any amount of information. The value of this message pair is ignored by the server. However, the entire pair is returned unmodified with the associated Response. The "Meta" object then can contain application specific information that later can be used by the client to synchronize Responses and Requests or to determine any other appropriate course of action when the Response is received.

4.1.3 General Message Content

Any number of message members may appear in the message although only those appropriate for the specific request will be used. All others will be ignored. One possible use for any extra message members beyond those required by the request is in providing debug information when viewed/logged on the wire (using Wireshark[5] for instance).

4.2 "Monitor" Message

Here is an example of the "Monitor" message. In addition to the State and Count for each Digital Input in sequence and Relay Output in sequence, there is information about the JNIOR including a timestamp. The "Monitor" message will be sent by the server whenever any I/O status changes.


```
<<< {
  "Message":"Monitor",
  "Model":"410",
  "Version":"v1.4-b4.1",
  "Serial Number":614070500,
  "Inputs":[
    {"State":1,"Count":49},
    {"State":0,"Count":360},
    {"State":0,"Count":8},
    {"State":0,"Count":38},
    {"State":0,"Count":3},
    {"State":0,"Count":4},
    {"State":0,"Count":5},
    {"State":0,"Count":7}
  ],
  "Outputs":[
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0}
  ],
  "Timestamp":1444155435066
}
```

Note that the number of inputs and outputs vary depending on the model of JNIOR and number of 4ROUT external modules. The standard Model 410 has 8 inputs and 8 outputs. The Model 412 has an additional 4 outputs for 12 and correspondingly less inputs where there are only 4. Similarly the Model 414 has 4 additional inputs for 12 and correspondingly fewer outputs where there are only 4.

There may be additional outputs included. The JNIOR will include up to 8 additional Relay Outputs from up to 2 external 4ROUT modules in this message. The order in which the external relay modules are assigned into the output sequence is managed by the Registry and the EXTERN command based upon each external module's ID.

4.2.1 "Status" Command

The "Monitor" message is an unsolicited message however, the message may be requested using the "Status" request message. This is not typically required as a "Monitor" message is sent immediately after a connection is authenticated and whenever there is a change thereafter. A The initial "Monitor" message is sent immediately after establishing the Websockets connection. If for any reason this initial message is not processed you can request the information using the "Status" request.

```
>>> {
  "Message":"Status"
}

<<< {
  "Message":"Monitor",
  "Model":"410",
  "Version":"v1.4-b4.1",
  "Serial Number":614070500,
  "Inputs":[
    {"State":1,"Count":49},
    {"State":0,"Count":360},
    {"State":0,"Count":8},
    {"State":0,"Count":38},
    {"State":0,"Count":3},
    {"State":0,"Count":4},
    {"State":0,"Count":5},
    {"State":0,"Count":7}
  ],
  "Outputs":[
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0},
    {"State":0}
  ],
  "Timestamp":1444155435066
}
```

4.3 "Control" Messages

Each "Control" message must contain a "Command" member which may be one of the following valid values:

- "Toggle"
- "Close"
- "Open"
- "Reset Latch"
- "Reset Counter"
- "Reset Usage"

Each "Control" Message must contain a numeric "Channel" member specifying the input/output channel. This parameter is 1-based where the number '1' specifies either the first Digital Input or first Relay Output. This depends on the specific "Command".

There is no formal response to these command messages although a "Monitor" message will invariably follow some for obvious reasons.

4.3.1 "Toggle" Command

The "Toggle" command inverts the state of the defined output "Channel". If the relay is open it will be closed. If it is closed it will be opened. The optional "Duration" member parameter if positive and non-zero specifies the milliseconds before the relay is to be returned to its original state. Therefore the following will close Relay Output 1 assuming that it originally is open.

```
>>> {
  "Message":"Control",
  "Command":"Toggle",
  "Channel":1
}
```

Similarly the following will pulse Relay Output 2. Assuming that originally the relay is open, it will be closed for precisely 5000 milliseconds (5 seconds).

```
>>> {
  "Message":"Control",
  "Command":"Toggle",
  "Channel":2,
  "Duration":5000
}
```

4.3.2 "Close" Command

The "Close" command closes the defined output "Channel". If the relay is open it will be closed. If it is closed it will remain closed (state = 1). The optional "Duration" member parameter if positive and non-zero specifies the milliseconds before the relay is to be returned to its original state. Therefore the following will close Relay Output 1.

```
>>> {
  "Message":"Control",
  "Command":"Close",
  "Channel":1
}
```

Similarly the following will pulse Relay Output 2. It will be closed for precisely 5000 milliseconds (5 seconds). There will be no change if the relay is already closed.

```
>>> {
  "Message":"Control",
  "Command":"Close",
  "Channel":2,
  "Duration":5000
}
```

4.3.3 "Open" Command

The "Open" command opens the defined output "Channel". If the relay is open it will remain so (state = 0). If it is closed it will be opened. The optional "Duration" member parameter if positive and non-zero specifies the milliseconds before the relay is to be returned to its original state. Therefore the following will open Relay Output 1.

```
>>> {
  "Message":"Control",
  "Command":"Open",
  "Channel":1
}
```

Similarly the following will pulse Relay Output 2. It will be opened for precisely 5000 milliseconds (5 seconds). There will be no change if the relay is already open.

```
>>> {
  "Message":"Control",
  "Command":"Open",
  "Channel":2,
  "Duration":5000
}
```

4.3.4 "Block" Command

The "Block" command allows the state of one or more relays to be changed simultaneously. The "Mask" parameter selects the relay or relays to be affected. Here the presence of a '1' bit indicates that the associated relay's state is to be affected. The parameter's LSB represents Relay Output 1. The corresponding bit in the "States" parameter defines the new state of the associated relay where a '1' indicates that the relay is to be closed, a '0' it is to be opened. The optional "Duration" member parameter if positive and non-zero specifies the milliseconds before the relay is to be returned to its original state. Therefore the following will close Relay Outputs 1 and 3 and open Relay Output 2.

```
>>> {
  "Message":"Control",
  "Command":"Block",
  "Mask":7
  "States":5
}
```

Similarly the following will pulse Relay Outputs 1 and 2 for precisely 5000 milliseconds (5 seconds).

```
>>> {
  "Message":"Control",
  "Command":"Block",
  "Mask":3,
  "States":3
  "Duration":5000
}
```

4.3.5 "Reset Latch" Command

Latching may be enabled for any of the digital inputs. This is a form of event capture which can be very useful in monitoring pulsed signals. A latching input may be set to trigger on either a positive going or negative going signal edge. In waiting for the event the input is considered to be *armed*. When the trigger signal is detected the input changes state.

A *LatchTime* may be configured. This defines a timer setting. The timer starts when the event occurs and the input signal is automatically *reset* when it expires. This provides for a form of pulse stretching. With a latch time of 10 seconds, pulsing an input for a mere 1 millisecond results in the input being activated for 10 seconds. The very brief event is captured. The result is signaled for a period long enough to alert any monitoring system.

If *LatchTime* is not configured (default is 0) or configured for 0 seconds there will be no automatic reset. The input state indicating the capture of an event must be manually reset or reset by the monitoring system using the "Reset Latch" command. An example message follows.

```
>>> {
  "Message":"Control",
  "Command":"Reset Latch",
  "Channel":2
}
```

4.3.6 "Reset Counter" Command

Input transitions are tallied. The counter can be configured to tally positive going or negative going edges. This provides an indication of the total number of input pulses detected. The JNIOR can count signals up to 2 kHz but is typically employed to count more reasonable paced events. At some point there may be a need to reset the counts to 0. This might occur each time this "meter" is read for instance and perhaps on a monthly basis. The following command does the job.

```
>>> {
  "Message":"Control",
  "Command":"Reset Counter",
  "Channel":3
}
```

4.3.7 "Reset Usage" Command

Often it is necessary to keep track of how long for instance that a piece of equipment is in use. The JNIOR tallies the time that either an input or an output is active. Each I/O point can be configured to tally usage time for either the high/1/ON state or the low/0/OFF state. It is reported as a fraction of hours. At some point you may need to reset this Usage Meter. The following command does the job.

```
>>> {
  "Message":"Control",
```

```
  "Command":"Reset Usage",
  "Channel":11
}
```

The JNIOR maintains 16 separate usage meters representing the 16 internal I/O points. This covers a mixture of inputs and outputs that varies depending on JNIOR Model. In this example, if we are running a Model 410 with 8 inputs and 8 outputs, we are resetting the Usage Meter for Relay Output 3. Channels 1 through 8 are inputs and 9 through 16 then correspond to Relay Outputs 1 through 8. So for this example Channel 11 is Relay Output 3.

4.4 File System Commands

The JNIOR supports a file system comparable to that on the PC. It is not possible to support, maintain or program a JNIOR without access to the file system. The JMP Server provides access to files for reading and writing depending on login permissions. This then provides for the greatest flexibility in application development.

4.4.1 "File List" Message

The "File List" message is used to request a listing of files in a particular directory/folder within the file system. This solicits a "File List Response" message providing the content. The response echoes the requested "folder" specification and supplies the "Content" as an array of objects each specifying the "Name", "Size, and last modification timestamp "Mod" for the file or folder. Note that a *folder* is distinguished from a *file* by the inclusion of a trailing '/' in the name. The folder's size is a count of the items it contains. A trailing '/' is not necessary in the folder specification. A typical exchange follows. The response message can be quite extensive depending on the numbers of files your system stores.

```
>>> {
  "Message":"File List",
  "Folder":"/"
}

<<< {
  "Message":"File List Response",
  "Folder":"/",
  "Content":[
    {
      "Name":"etc/",
      "Size":1,
      "Mod":"07 Jul 2016 10:25"
    },
    {
      "Name":"temp/",
      "Size":0,
      "Mod":"14 Sep 2016 13:16"
    },
    {
      "Name":"flash/",
      "Size":38,
      "Mod":"23 Sep 2016 07:50"
    },
    {
      "Name":"manifest.json",
      "Size":32698,
      "Mod":"29 Jul 2016 10:26"
    },
    {
      "Name":"jniorsys.log.bak",
      "Size":65557,
      "Mod":"20 Sep 2016 15:49"
    },
    {
      "Name":"jniorsys.log",
      "Size":16526,
      "Mod":"23 Sep 2016 07:52"
    },
    {
      "Name":"jniorboot.log.bak",
      "Size":1056,
      "Mod":"23 Sep 2016 07:33"
```

```
    },
    {
      "Name":"jniorboot.log",
      "Size":1010,
      "Mod":"23 Sep 2016 07:50"
    }
  ]
}
```

4.4.2 "File Read" Message

The "File Read" operation is used to obtain the data for a single file. Data is returned using Base64 encoding. This allows for the transfer of files containing binary data. The "Encoding" parameter indicates "base64". At this time this is the only encoding that is supported. The "Size" parameter indicates the size of the file and the length of the decoded content of the "Data" parameter.

```
>>> {
  "Message":"File Read",
  "File":"/flash/www/config/folder.png"
}

<<< {
  "Message":"File Read Response",
  "File":"/flash/www/config/folder.png"
  "Size":329,
  "Encoding":"base64",
  "Data":"iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAIAAACQkWg2AAAABnRSTlMA/wD/AP83WBt9AAAA/klEQVR4Ab2Ss1+G
          UQBG+9eyXUtY0pRt27bXtsaWXGtYss2L533XejK/L3d/1/eca5urL4b/EfZGXB7jZwWIJugOVhaG7F5Fdr4nnFVA
          NEN3vhsJvBA4DS7r5GwgK9bjkyDG7DmNWoxSywlqJUmtpqjVVFaeRz2fSfJeIC0n/XFRhrNKXNaD25OtUO08GMxe
          RuNikAzjk6AWQvVxDk5KcFEN0QjZAtUG3Q6zh9E4bJXDHoSfCWvhei8Fx/m8gOeLkGYkwPhCEKuhajNGH2TgtATn
          1bhsgLhZhDQjGZIvhUVvuRqhd5NxxEXKcVHHx+EijGoymMCLd/jwQv/x810D+VqIFtAOzxIAAAAASUVORK5CYII=
",
  "Status":"Succeed"
}
```

Reading Large Files

For very large files the "File Read Response" can become quite huge. This can lead to memory and performance concerns. Fortunately you can optionally use "Limit" and "Offset" parameters to read sections of the file while limiting the "data" content size. Repeated "File Read" requests can then be used to retrieve the entire file. This is also useful if the application requires the retrieval of only a small amount of information from a certain offset in a file and not the entire file.

When an "Offset" is specified in the "File Read" request the content of "Data" reflects the bytes starting at the file offset. A value of "0" indicates the beginning of the file.

When the "Limit" parameter is specified, the read operation will return only that number of bytes or the balance of the file whichever is less.

When either "Limit" or "Offset" are specified the "File Read Response" will contain a "NumRead" parameter indicating the actual number of bytes read. The "Size" parameter will always reflect the total file size. The following is an example of a the exchanges needed to read a file limiting the message size. Note that you might likely be able to transfer files as large as 128KB in a single message.

```
>>> {
  "Message":"File Read",
  "File":"/flash/www/config/folder.png"
  "Limit":256,
}

>>> {
  "Message":"File Read Response",
  "File":"/flash/www/config/folder.png",
  "Size":329,
  "Offset":0,
  "Limit":256,
  "NumRead":256,
  "Encoding":"base64",
  "Data":"iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAIAAACQkWg2AAAABnRSTlMA/wD/AP83WBt9AAAA/klEQVR4Ab2Ss1+
```

```
          GUQBG+9eyXUtY0pRt27bXtsaWXGtYss2L533XejK/L3d/1/eca5urL4b/EfZGXB7jZwWIJugOVhaG7F5Fdr4nnF
          VANEN3vhsJvBA4DS7r5GwgK9bjkyDG7DmNWoxSywlqJUmtpqjVVFaeRz2fSfJeIC0n/XFRhrNKXNaD25OtUO08G
          MxeRuNikAzjk6AWQvVxDk5KcFEN0QjZAtUG3Q6zh9E4bJXDHoSfCWvhei8Fx/m8gOeLkGYkwPhCEKuhag==",
  "Status":"Succeed"
}

>>> {
  "Message":"File Read",
  "File":"/flash/www/config/folder.png"
  "Offset":256,
  "Limit":256,
}

<<< {
  "Message":"File Read Response",
  "File":"/flash/www/config/folder.png",
  "Size":329,
  "Offset":256,
  "Limit":256,
  "NumRead":73,
  "Encoding":"base64",
  "Data":
"M0YfZOC0BOfVuGyAuFmENCMZki+FRW+5GqF3k3HERcpxUcfH4SKMajKYwIt3+PBC//HzXQP5WogW0A7PEgAAAABJRU5ErkJgg
g==",
  "Status":"Succeed"
}
```

4.4.3 "File Write" Message

The "File Write" operation is used to transfer a file to the JNIOR. The write request specifies the target "File" from the root of the file system. The "File" parameter must be present for the request to be considered valid.

Since files may contain binary data the "Data" portion of the message is encoded with Base64 encoding. The "Encoding" parameter must be specified as precisely as "base64".

The "Size" parameter is required and must define the intended size of the file in bytes. It must match the decoded Base64 "Data" content in length. The data is decoded and the byte count compared to that specified before attempting to write the file.

You may optionally specify the *last modification* timestamp parameter "Mod" for the file. The timestamp is represented as a Linux time in milliseconds since midnight January 1st 1970 in Universal Coordinated Time (UTC). If present the last modification time for the resulting file will be as specified.

Once the file is written the "File Write Response" is returned. The "File" and "Size" are reflected in the response (as would any "Meta" data per Section 4.1.2). The formatted timestamp is also returned in a "Mod" parameter. The "NumWritten" parameter reflects the result of the file write. This should match the specified "Size" value if the write is to be successful. A value less than zero indicates an error. A typical exchange follows.

```
>>> {
  "Message":"File Write",
  "File":"/temp/main.c",
  "Size":144,
  "Mod":1310414726000,
  "Encoding":"base64",
  "Data":"DQojaW5jbHVkZSAiaW80MzAuaCINCg0KaW50IG1haW4oIHZvaWQgKQ0Kew0KICAvLyBTdG9wIHdhdGNoZG9nIHRp
          bWVyIHRvIHByZXZlbnQgdGltZSBvdXQgcmVzZXQNCiAgV0RUQ1RMID0gV0RUUFcgKyBXRFRIT0xEOw0KDQogIHJl
          dHVybiAwOw0KfQ0K"
}

<<< {
  "Message":"File Write Response",
  "File":"/temp/main.c",
  "Size":144,
  "Mod":"11 Jul 2011 16:05",
  "NumWritten":144,
  "Status":"Succeed"
}
```

Writing Large Files

For very large files the "File Write" message can become huge. This can lead to memory and performance concerns. Fortunately, you can optionally use the boolean parameter "Append" to break file writes into manageable blocks.

To append to an existing file you use the "File Write" message exactly as described above. You must include an additional parameter named "Append" set to the value of "true". In this case the file must previously exist and the data included with the "Data" parameter will be appended to it. The write operation will fail if the file is not present. So to transfer a large file using multiple messages the first must not indicate "Append". It would be included only in subsequent "File Write" messages. This will insure that the resulting file will be as you are expecting.

In this case the returned "Size" parameter will increase as the size of the target file increases by the "NumWritten" byte count.

4.4.4 "File Remove" Message

One or more files or folders can be removed/deleted using the "File Remove" request. The "Files" parameter is an array of file/folder names. You do not use a trailing '/' when specifying a folder in this request. The JNIOR will attempt to remove each file/folder specified in the array.

Each individual deletion may or may not succeed. The "File Remove Response" will enumerate the successful deletions in a "Succeed" array. Similarly any failures will be listed in a "Fail" array. Depending on the results the response message may contain either a "Succeed" array or a "Fail" array or both. Between the two arrays the results of each attempt for those items listed in the original "Files" array will be reported. A couple of examples follow.

```
>>> {
  "Message":"File Remove",
  "Files":[
    "/flash/image.txt",
    "/flash/main.c"
  ]
}

<<< {
  "Message":"File Remove Response",
  "Succeed":[
    "/flash/image.txt",
    "/flash/main.c"
  ]
}
```

Here we attempt to remove a folder and the request fails. In this case we expect that it would fail both because the folder contains files and sub-folders (it is not empty) and also because it is a special system folder. You cannot remove the /etc, /flash, or /temp folders. You also cannot remove any content from the /etc folder.

```
>>> {
  "Message":"File Remove",
  "Files":[
    "/flash"
  ]
}

<<< {
  "Message":"File Remove Response",
  "Fail":[
    "/flash"
  ]
}
```

4.4.5 "File Rename" Message

You may rename a file or folder using the "File Rename" request. In this case you specify the file/folder with the "old" parameter and the new file/folder name with the "New" parameter. The files must be specified from the root of the file system and both specifications must be in the same folder. You cannot "move" a file through a rename operation. A file/folder matching the "New" specification cannot already exist.

The "File Rename Response" reiterates the request and adds a "Result" parameter. The "Result" will be either "Succeed" or "Fail" reflecting the result of the rename operation. An example follows.

```
>>> {
  "Message":"File Rename",
  "Old":"/flash/main.c",
  "New":"/flash/test-prog.c"
}

<<< {
  "Message":"File Rename Response",
  "Old":"/flash/main.c",
  "New":"/flash/test-prog.c",
  "Result":"Succeed"
}
```

4.4.6 "File Mkdir" Message

The ability to create a folder completes the set of file system functions. Here you can create a new folder using the "File Mkdir" message. The new folder is specified from the root of the file system by the "Folder" parameter.

The "File Mkdir Response" reiterates the "Folder" and adds a "Result" which will be either "Succeed" or "Fail" depending on the outcome of the creation attempt.

```
>>> {
  "Message":"File Mkdir",
  "Folder":"/flash/testing"
}

<<< {
  "Message":"File Mkdir Response",
  "Folder":"/flash/testing",
  "Result":"Succeed"
}
```

4.5 Registry Commands

The JNIOR is configured by various parameter settings which are stored in the non-volatile Registry. In addition to configuration there are special keys (that start with the dollar sign '$') which record and report dynamic information. The input and output Usage Meter status is reported through a system Registry key named $HourMeter for example. The Registry then plays an important role in monitoring the status of a JNIOR.

4.5.1 "Registry Update" Message

The "Registry Update" message is an unsolicited message. It is transmitted through the JMP Server whenever there is a change in the Registry. This notifies the client when new keys are created and when they are removed (content is empty/null). It notifies the client whenever the content of a key is changed. This allows the client to respond to the changing configuration of a connected unit as well as to receive information stored in dynamic system keys. The following is a very typical update for a channel's usage meter.

```
<<< {
  "Message":"Registry Update",
  "Keys":{
    "IO/Inputs/din1/$HourMeter":"43.68"
  }
}
```

Note that the "Keys" member passes an object which may contain 0 or more name/value pairs where the *name* is the Registry Key and the *value* its content. Here the $HourMeter reports 43.68 hours of usage. These update every 100th of an hour. That is the resolution of the Usage Meter. In general, Registry Updates will report only one key per message since changes occur in sequence and each change generates an update message through the inter-process messaging system. The Web Server picks up the internal message and broadcasts the information to all active JMP connections.

4.5.2 "Registry List" Command

The Registry stores information that from time to time you may need to retrieve. This is easily done if you know precisely what Registry keys to read. A lot of work can be saved if you can determine easily what Registry keys have been defined and that have data available for reading. The "Registry List" command is used to obtain a listing similar to a file directory or folder listing for a node in the Registry.

The "Registry List" command summons a "Registry List Response" message. A complete exchange is shown below. The Client

sends the request and the server supplies the response message. Note how the "Meta" member might be used to pass information to the routine that eventually (and asynchronously) will receive the response.

```
>>> {
  "Message":"Registry List",
  "Meta":{"Op":"registry","Node":"/IO/Inputs/din1"},
  "Node":"/IO/Inputs/din1"
}

<<< {
  "Message":"Registry List Response",
  "Meta":{"Op":"registry","Node":"/IO/Inputs/din1"},
  "Keys":[
    "/IO/Inputs/din1/Enabled",
    "/IO/Inputs/din1/$HourMeter",
    "/IO/Inputs/din1/Conditioning",
    "/IO/Inputs/din1/LatchState",
    "/IO/Inputs/din1/Desc",
    "/IO/Inputs/din1/ClosedDesc",
    "/IO/Inputs/din1/OpenDesc",
    "/IO/Inputs/din1/Count/",
    "/IO/Inputs/din1/ShowCount",
    "/IO/Inputs/din1/ShowUsageMeter",
    "/IO/Inputs/din1/UsageState",
    "/IO/Inputs/din1/CountState",
    "/IO/Inputs/din1/ShowControls"
  ]
}
```

Here we note that a list (or array) of key names is returned in the "Keys" member. Note too that those that end in a forward slash '/' represent sub-nodes which will contain keys or additional nodes which can be retrieved with a subsequent request for that node. There are no empty sub-nodes (subdirectories or subfolders) in the JANOS Registry. Therefore if the node is listed it must have content within its structure somewhere.

4.5.3 "Registry Read" Command

The "Registry Read" command request is used to retrieve the content of one or more Registry keys. The request includes the "Keys" member which provides and array of Registry keys for which we want the content. Note that the optional "Meta" member is available for use but not employed in this example. The request solicits a "Registry Response" message which returns the "Keys" member which list time returns an *object* whose members are name/value pairs reporting each key and its content.

```
>>> {
  "Message":"Registry Read",
  "Keys":[
    "/IO/Inputs/din1/Enabled",
    "/IO/Inputs/din1/$HourMeter",
    "/IO/Inputs/din1/Conditioning",
    "/IO/Inputs/din1/LatchState",
    "/IO/Inputs/din1/Desc",
    "/IO/Inputs/din1/ClosedDesc",
    "/IO/Inputs/din1/OpenDesc",
    "/IO/Inputs/din1/ShowCount",
    "/IO/Inputs/din1/ShowUsageMeter",
    "/IO/Inputs/din1/UsageState",
    "/IO/Inputs/din1/CountState",
    "/IO/Inputs/din1/ShowControls"
  ]
}

<<< {
  "Message":"Registry Response",
  "Keys":{
    "/IO/Inputs/din1/Enabled":"true",
    "/IO/Inputs/din1/$HourMeter":"44.28",
    "/IO/Inputs/din1/Conditioning":"1",
    "/IO/Inputs/din1/LatchState":"1",
    "/IO/Inputs/din1/Desc":"Input 1",
```

```
    "/IO/Inputs/din1/ClosedDesc":"ON",
    "/IO/Inputs/din1/OpenDesc":"OFF",
    "/IO/Inputs/din1/ShowCount":"true",
    "/IO/Inputs/din1/ShowUsageMeter":"true",
    "/IO/Inputs/din1/UsageState":"0",
    "/IO/Inputs/din1/CountState":"0",
    "/IO/Inputs/din1/ShowControls":"true"
  }
}
```

Note that there is a name/value pair corresponding to each requested Registry key even if that key is undefined (does not exist). All of the keys requested here in this example have values. If a key is not present it will return the empty or null string value "".

4.5.4 "Registry Write" Command

An external application may need to alter the configuration of a JNIOR. In order to do so it is necessary to create or change the content of a Registry key. The "Registry Write" command is used for this purpose. There is no restriction as to what can be written to the Registry. Specific keys have specific purposes and some are recognized internally by the JANOS operating system. Others pertain to the formatting of the dynamic pages. Still others may be specific to custom applications and programs running on the JNIOR.

The "Keys" member of the "Registry Write" command message provides an *object* containing 1 or more name/value pairs. Each element represents a write request where the *name* is the Registry key and the *value* its intended content. Note that the JANOS Registry stores strings. Only strings can be written however they may encode practically anything. The "Registry Write" request solicits a "Registry Response" returning the keys successfully written.

If there is an error in writing a key, the key will be returned either with an empty or null string ("") or the prior and still valid content. Here is an example changing the description displayed by the configuration pages for Digital Input 2. The write was successful.

```
>>> {
  "Message":"Registry Write",
  "Keys":{
    "IO/Inputs/din2/Desc":"Part Produced"
  }
}

<<< {
  "Message":"Registry Response",
  "Keys":{
    "IO/Inputs/din2/Desc":"Part Produced"
  }
}
```

Not surprisingly this exchange is immediately followed by a "Registry Update" message. This signals to all who are listening that the key has been altered.

```
<<< {
  "Message":"Registry Update",
  "Keys":{
    "IO/Inputs/din2/Desc":"Part Produced"
  }
}
```

4.5.5 "Registry Write Encrypted" Command

The Registry may store user names and passwords for configured email accounts for example. The user's and administrator's account credentials defined in JANOS are stored very securely internal the processor chip itself. Passwords for other purposes are configured in the Registry and should not be stored in plain text. Note the result of the following "Registry Read" request.

```
>>> {
  "Message":"Registry Read",
  "Keys":[
    "/IpConfig/Password"
  ]
}

<<< {
```

```
  "Message":"Registry Response",
  "Keys":{
    "/IpConfig/Password":"Qrq5CQ/rYBPfyeHMg8VEWr/iFCXsPDD0dDVtXvZaHH8="
  }
}
```

This password for the default email account is not readable. This is not just obfuscated from view but securely encrypted by a secret key known only to the JANOS operating system and one that is unique to the unit. Nevertheless an external application (including the configuration pages) needs to be able to set a new password. This cannot be done without special handling as the encryption secret is not externally known and cannot be determined.

To make this possible, the "Registry Write Encrypted" command is available. This is used to write new password credentials for the default email account and indeed any other such account where JANOS later requires access to the plain text password. JANOS needs to be able to decrypt the content. If an application wants to store data securely it can encrypt the data using its own procedures and write the encrypted result using the normal "Registry Write" command. Later the content can be read and decrypted. The special form of write command is used only for information that JANOS stores with its own secure encryption. Data that only JANOS can then decrypt and use.

The "Registry Write Encrypted" command works exactly as does the "Registry Write" command. It also summons a "Registry Response" but one that shows only the encrypted password content. The password is provided in the request in combination with the username and in plain text. It is highly recommended that passwords not be configured through this protocol unless the connection used is secured by TLS/SSL. The procedure for setting a new password can be gleamed from the dynamic web pages supplied with the unit. The steps to handle it are in the Javascript. You can also contact INTEG Process Group, Inc. for assistance if you have trouble. Typically this password is set using the IPCONFIG command in the Console.

4.6 Console Session

A *Console Session* provides access to the JANOS Command Line interpreter. Practically every operating system has a command line interpreter. Windows(R) has the DOS Command Prompt. JANOS is no different and in fact provides a command line interpreter that recognizes many different commands some of which are similar to commands available in either the DOS or Linux environments. The command line Console provides the tools needed for JNIOR configuration, diagnostics and application development.

The *Console* can be accessed by 115,200 BAUD serial connection to the RS-232 port directly on the JNIOR. If the unit is configured for operation on the network the Console can also be opened by making a Telnet connection to the unit. The command line interpreter functions identically using either approach. The RS-232 diagnostic port provides some additional information such as a boot dialog chronicling the boot sequence and error messages should critical assertions occur.

The JMP Server also provides access to the command line interpreter. A JMP connection can open a Console Session. This is a separate command process under the control of the JMP Server on behalf of the JMP connection. The client can supply data simulating keystroke entry and consume characters output from the session perhaps for display. Only one session can be opened for each JMP connection although it may be closed and reopened any number of times while the JMP connection is active.

The dynamic configuration pages supplied with the unit support a "Console" tab through which the use can interact with the Console Session in a fashion virtually identical to any Telnet client or serial terminal client application. You can review the Javascript for more insight.

An application may use a Console session to accomplish some action only available through the command line interpreter. In such case the session my be opened, the command or commands executed, and then immediately closed.

4.6.1 "Console Open" Command

When a JMP connection is made there is no command session associated. If commands are to be fed to the command line interpreter or a console session supported it must be opened. The "Console Open" command is then required and this solicits a "Console Response" message whose "Status" member provides the status of the result. The outcome can be either "Established" or "Failed". Below a Console Session is started.

```
>>> {
  "Message":"Console Open"
}

<<< {
  "Message":"Console Response",
  "Status":"Established"
}
```

Note that while a Console session is open all other JMP requests and unsolicited messaging are still valid and active. The console

session can be supported in parallel with all other activity over the connection.

4.6.2 "Console Stdin" Message

The "Console Stdin" message passes character data to the command line interpreter through its **stdin** serial stream. These characters function exactly as if they were typed at the keyboard in a Telnet session. You use "\r" as the ENTER keystroke. An UP-ARROW or DN-ARROW keystroke is replaced by its VT-100 escape sequence which the Series 3 and Series 4 JNIORs have come to expect. Characters entered through the Console tab in the dynamic configuration pages are each sent immediately as typed one at a time to the stdin stream. Note that the console session command line interpreter echoes character input just as it does everywhere else.

```
{
  "Message":"Console Stdin",
  "Data":"dir\r"
}
```

4.6.3 "Console Stdout" Message

With every **stdin** stream there is likely a **stdout** and the Console Session is no exception. The "Console Stdout" message is transmitted by the server and it supplies data available for display. This may be echoed characters or command output. It is delivered asynchronously and therefore may contain 1 or more characters. It may contain the entire output of a command or only part depending on JANOS activity levels. In other words this is a character stream and a single "Console Stdout" message may contain multiple lines of output or the output from multiple commands. The output from a single console command may be spread across multiple messages. Applications must be coded with this in mind. For example this is data from the Console session tab where the command was typed in and executed.

```
>>> {"Message":"Console Stdin","Data":"d"}
>>> {"Message":"Console Stdin","Data":"i"}
<<< {"Message":"Console Stdout","Data":"d"}
>>> {"Message":"Console Stdin","Data":"r"}
<<< {"Message":"Console Stdout","Data":"i"}
>>> {"Message":"Console Stdin","Data":"\r"}
<<< {"Message":"Console Stdout","Data":"r"}

<<< {
  "Message":"Console Stdout",
  "Data":"\r\netc\r\nflash\r\njniorboot.log\r\njniorboot.log.bak\r\njniorsys.log\r\n
      jniorsys.log.bak\r\nmyfile.txt\r\nphp.log\r\ntemp\r\n\r\nBruce_Dev /> "
}
```

p(. This would be the same command executed by an application. The results may not be consistent although the output of the command certainly should.

```
>>> {
  "Message":"Console Stdin",
  "Data":"dir\r"
}

<<< {
  "Message":"Console Stdout",
  "Data":"dir\r\netc\r\nflash\r\njniorboot.log\r\njniorboot.log.bak\r\njniorsy"
}

<<< {
  "Message":"Console Stdout",
  "Data":"s.log\r\njniorsys.log.bak\r\nmyfile.txt\r\nphp.log\r\ntemp\r\n\r\nBruce_Dev /> "
}
```

An application would likely buffer all data until the command line prompt is detected. Only then can it interpret the list of files supplied reliably.

4.6.4 "Console Close" Command

A Console session will remain active until closed. It is automatically closed should the JMP connection terminate. It is good practice however to close the command session if there is no immediate need for it. This keeps the load on JANOS to a minimum and keeps the process slot open for other activities. The "Console Close" command solicits a "Console Response" message whose "Status" member indicates "Closed" in all cases.

```
>>> {
  "Message":"Console Close"
}

<<< {
  "Message":"Console Response",
  "Status":"Closed"
}
```

4.6.5 Example Session

Here is a example of opening a command session and logging in using the default credentials. The session is then closed once the prompt has been reached. Note how the entry of the password is not echoed. This is just as it is in any JNIOR Telnet session.

```
>>> {
  "Message":"Console Open"
}

<<< {
  "Message":"Console Response",
  "Status":"Established"
}

<<< {
  "Message":"Console Stdout",
  "Data":"\r\nWelcome to the JNIOR Model 410 (S/N 614070500) running JANOS v1.4-b4.1\r\n
      Copyright (c) 2012-2015 INTEG Process Group, Inc., Gibsonia PA USA.\r\n
      Local time: Wed Oct 07 13:45:38 EDT 2015    Process ID: 16\r\n\r\n
      Bruce_Dev login: "
}

>>> {
  "Message":"Console Stdin",
  "Data":"jnior\r"
}

<<< {
  "Message":"Console Stdout",
  "Data":"jnior\r\nBruce_Dev password: "
}

>>> {
  "Message":"Console Stdin",
  "Data":"jnior\r"
}

<<< {
  "Message":"Console Stdout",
  "Data":"*****\r\n\r\nBruce_Dev /> "
}

>>> {
  "Message":"Console Close"
}

<<< {
  "Message":"Console Response",
  "Status":"Closed"
}
```

While access to the Console offers a great amount of flexibility for any application it should not be abused. If there is some action that logic suggests should itself be a built-in function of the server, it would be greatly appreciated if you would let us know. We are always looking to improve the product. Even though you may have gotten the job done there is no reason why we shouldn't make life easier going forward. So please do make us aware of the need.

4.7 External Devices

There are a number of *external modules* that can be used with JNIOR. These attach to the Sensor Port and can be daisy-chained. The most popular of these are the 4ROUT and Power 4ROUT modules each adding an additional 4 relay outputs to the JNIOR I/O set. Up to two 4ROUT modules can be used which will logically extend the number of relay outputs reported in the "Monitor" message. But additional 4ROUT and other modules can be used limited only by the power load on the sensor port/network. Modules are read and written using their ID string as an address.

Each interaction with an external module involves the exchange of a *Data Block*. The data blocks will differ depending on whether a device is being read or written. These blocks define a structure of fields. The definitions for the device blocks are provided as part of the JNIOR Protocol Specification[1].

4.7.1 "Enumerate Devices" Command

Each external module has a unique ID. This is a 16 character hexadecimal string representing 8 bytes. The least significant byte or rightmost 2 characters always specify the type of module. This would be 'FB' for a standard 4ROUT external module. The 5 bytes or 10 characters immediately preceding the type can be considered a Serial Number of sorts. Typically these are constrained to the digits 0 through 9. The first byte or 2 characters is a check byte and the byte following a software code (typically but not always a '11').

The "Enumerate Devices" command is used to retrieve a list of the active modules connected to the JNIOR. This solicits an "Enumerate Devices Response" which includes a "Devices" list of 0 or more module IDs. Note that the "Meta" member can be included in the request and will be returned unmodified in the response. This can be used to pass information to the routine that will process the response. For example we have this exchange.

```
>>> {
  "Message":"Enumerate Devices"
}

<<< {
  "Message":"Enumerate Devices Response",
  "Devices":[
    "CD111090708109FB",
    "16111100125011FE"
  ]
}
```

This tells us that the JNIOR has two connected modules. One is type 0xFB which is a 4ROUT module. The other a type 0xFE which is the Analog 4-20 ma module. The device types are described in the JNIOR Protocol Specification document.

4.7.2 "Read Devices" Command

The "Read Devices" command is used to obtain the current data block from one or more devices. The format of the data block is specific to the device type. This solicits the "Read Devices Response" which includes only those devices successfully read and the data block content encoded in a "Hex" string. Here we read both of the devices reported in the previous enumeration.

```
>>> {
  "Message":"Read Devices",
  "Devices":[
    "CD111090708109FB",
    "16111100125011FE"
  ]
}

<<< {
  "Message":"Read Devices Response",
  "Devices":[
    {
      "Address":"CD111090708109FB",
      "Hex":"0F000000000000000000"
    },
    {
      "Address":"16111100125011FE",
      "Hex":"00000000000000000000000000"
    }
  ]
}
```

The content of these blocks can be interpreted using the formats defined in the JNIOR Protocol Specification document. From this

response we can see that all of the relays on the 4ROUT device are open and not activated. The 4-20 module is connected but since in this instance it is not wired to any current loop devices it reports all inputs at 4 ma (0x0000) and its two outputs are set to 4 ma (0x0000).

4.7.3 "Write Devices" Command

The "Write Devices" command is used to write to an external module. Here we pass a properly formatted data block to the 4ROUT module reported in the prior example. The goal is to close the 3rd relay (Relay Output C). This is achieved by setting the mask (first byte) to 0x04 informing the module that we will only be setting the state of the 3rd relay. We define the state (second byte) as 0x04 to close that relay. The command solicits the "Write Devices Response" which returns the result of each write attempt. The "Result" member will be 'true' if the write is successful and 'false' otherwise.

```
>>> {
  "Message":"Write Devices",
  "Devices":[
    {
      "Address":"CD111090708109FB",
      "Hex":"0404000000000000000"
    }
  ]
}

<<< {
  "Message":"Write Devices Response",
  "Devices":[
    {
      "Address":"CD111090708109FB",
      "Result":true
    }
  ]
}
```

Note that the relays in the 4ROUT module can be pulsed. Here we simply turned Relay C on. The value for its pulse duration being 0x0000 in the block.

4.8 Realtime Clock

Access to the JNIOR's realtime clock is provided. This can be used to obtain and display the clock as it is maintained by the JNIOR. This exchange can be useful as a tick allowing you to detect the loss of connection.

```
>>> {
  "Message":"Clock Read"
}

<<< {
  "Message":"Clock Response",
  "Time":1452012668787,
  "Date":"Tue, 05 Jan 2016 16:51:08 GMT"
}
```

An Administrator may adjust the JNIOR's realtime clock. There is no response.

```
>>> {
  "Message":"Clock Set"
  "Time":1452012668787
}
```

4.9 Shutdown/Reboot Notification

This message is sent when the JNIOR is shutting down for a reboot.

```
<<< {
  "Message":"Device Shutdown"
}
```

4.10 System Logging (Syslog)

JANOS logs system events to the jniorsys.log file. When this file reaches a certain size it is aged to the jniorsys.log.bak file. The content of the latter is discarded. As a result there can be as much as 128KB of system logs.

The "Syslog Read" request will return the log history in sequence from oldest to latest. This includes both the content of both files, as much as 128KB worth of log information.

```
>>> {
  "Message":"Syslog Read"
}

<<< {
  "Message":"Syslog Read Response",
  "Data":[
    "10/10/16 10:28:16.645, -- JANOS 410 v1.5.1-rc0.3 initialized (POR: 3050)",
    "10/10/16 10:28:16.683, Registry exported to: /flash/jnior.ini (pid 2)",
    "10/10/16 10:28:17.791, Added: WebServer/Locator/Serial = /flash/serialcontrol/www (pid 0)",
        .
        .
        .
    "10/20/16 12:55:26.582, -- JANOS 410 v1.5.1-rc2.1 initialized (POR: 3200)",

"10/20/16 12:55:26.596, ** Warning: Password change for 'jnior' administrator account recommended.
",
    "10/20/16 12:55:49.467, Requesting time sync from pool.ntp.org (108.59.2.24)",
    "10/20/16 12:55:55.000, Clock synchronized via NTP (+9)",
    "10/20/16 13:26:15.698, Starting session Command/Serial (pid 7)",
    "10/20/16 13:26:15.939, Successful login for 'jnior' (pid 7)",

"10/20/16 14:02:33.633, FTP/10.0.0.20:63066 transferred /flash/www/config/config.js [318.4 kbps]",
    "10/20/16 14:02:40.130, FTP/10.0.0.20:63072 uploaded /flash/www/config/config.js [333.8 kbps]"

  ]
}
```

Note that the "Syslog Read Response" can be quite lengthy. Each line of the log is supplied in sequence in the "Data" string array.

As new entries are posted to the jniorsys.log file the JMP Server will supply them. This is a real-time update and these messages are unsolicited. Note here the the "Data" is simply a string and not an array. These messages supply one line at a time.

```
<<< {
  "Message":"Syslog Update",
  "Data":"10/20/16 14:11:10.561, [logger] This is a new log entry"
}
```

4.11 Application Messaging

JANOS supports an inter-process messaging mechanism in the form of a message loop or pump. Messages are identified with a *Message Number* or *Type*. This JMP request processes messages associated with application message numbers of 1024 or greater. Reserved system messages, those with message numbers less than 1024, cannot be posted and any such request is ignored.

The "Post Message" request routes the accompanying message onto the process messaging loop. The "Number" entry must be present and contain a number greater than or equal to 1024. The assumption is that an application program is running which is monitoring the messaging system for communications associated with this number. The "Content" entry must also be present and contain a text string of non-zero length. This string would contain information appropriate to the application that can be later parsed by the receiver. There is no immediate response to the request. No response is provided to indicate whether or not the message is received or if there is anyone even listening. An acknowledgement would be the responsibility of the listener. For example

```
>>> {
  "Message":"Post Message",
  "Number":1024,
  "Content":"Some Textual Data"
}
```

The JMP connection can receive replies from the application. The application can send any number of messages at any time back to the JMP connection once the channel has been established. These must all have the same message number as in the initial posting and contain textual content. In order for a message of a particular message number to be routed back through a JMP connection the

client must first post a message with that very number. This then sets up the necessary routing for replies.

```
<<< {
  "Message":"Reply Message",
  "Number":1024,
  "Content":"Textual Response Data"
}
```

A protocol can then be created between a webpage and a running program to achieve whatever purpose the application requires.

5. Appendices

5.1 Auth-Digest Calculation

If the JMP connection requires a login it will respond with a "401 Unauthorized" error text. The server provides a unique "Nonce" string as part of this message. This can be used in conjunction with the username and password to calculate the appropriate Authorization Digest. This requires a MD5 message digest calculation which generates a 16 byte digest represented as 32 hexadecimal characters. The calculation proceeds as follows:

```
Digest = Username + ":" + MD5(Username + ":" + Nonce + ":" + Password)
```

Where Username, Password, Nonce and Digest are all strings. The resulting Digest string is returned in the "Auth-Digest" member. Here is an example login with the default administrator's account.

```
>>> {
  "Message":""
}

<<< {
  "Message":"Error",
  "Text":"401 Unauthorized",
  "Nonce":"bc581a9683d3e1857218db135e4b"
}

>>> {
  "Auth-Digest":"jnior:6b7b418f223e7e0dc600c41c7b6644b3"
}

<<< {
  "Message":"Authenticated",
  "Administrator":true,
  "Control":true
}
```

5.2 External Module Types

The following module types are typically used with JNIOR. The type is represented in hexadecimal. This appears as the last two characters in a module's ID string.

- Type 10 -- Temperature Probe
- Type 26 -- Temperature Probe
- Type F9 -- 3-Channel LED Dimmer
- Type FA -- Rack Mounted User Panel
- Type FB -- 4ROUT Quad Relay Output Module
- Type FC -- RTD Temperature Module
- Type FD -- 10V Analog Module
- Type FE -- 4-20ma Analog Module

5.3 Example - 4ROUT Read/Write Blocks

The read and write data blocks appropriate for each module are defined in the JNIOR Protocol Specification. The data blocks for the 4ROUT Quad Relay Output module are represented here as an example of translation between the binary descriptions and that required for this protocol.

4ROUT Read Data Block

```
"Hex":"0000000000000000000"
       | | |   |   |   |
       | | |   |   |   0000  Relay D Pulse Time Remaining (0 to FFFF hexadecimal milliseconds)
       | | |   |   0000  Relay C Pulse Time Remaining (0 to FFFF hexadecimal milliseconds)
       | | |   0000  Relay B Pulse Time Remaining (0 to FFFF hexadecimal milliseconds)
       | | 0000  Relay A Pulse Time Remaining (0 to FFFF hexadecimal milliseconds)
       | 00  Bit mapped relay status (0-open 1-closed)
       00  Bit mapped last relay mask used (1-selected)
```

```
  Bit mappings (mask and status)
  +---------+---------+---------+---------+---------+---------+---------+---------+
  |    0    |    0    |    0    |    0    | Relay D | Relay C | Relay B | Relay A |
  +---------+---------+---------+---------+---------+---------+---------+---------+
```

Of most importance here are the last 4 bits of the second byte. This is basically the 4th character of the "Hex" string encoding which relays are closed and which are open. '0' indicating that all of OFF. 'F' indicating all are ON.

4ROUT Write Data Block

```
"Hex":"0000000000000000000"
       | | |   |   |   |
       | | |   |   |   0000  Relay D Pulse Time (0 to FFFF hexadecimal milliseconds)
       | | |   |   0000  Relay C Pulse Time (0 to FFFF hexadecimal milliseconds)
       | | |   0000  Relay B Pulse Time (0 to FFFF hexadecimal milliseconds)
       | | 0000  Relay A Pulse Time (0 to FFFF hexadecimal milliseconds)
       | 00  Bit mapped relay state (0-open 1-closed)
       00  Bit mapped relay selection mask (1-selected)
```

```
  Bit mappings (mask and state)
  +---------+---------+---------+---------+---------+---------+---------+---------+
  |    0    |    0    |    0    |    0    | Relay D | Relay C | Relay B | Relay A |
  +---------+---------+---------+---------+---------+---------+---------+---------+
```

The state of the relays corresponding to the '1' bits in the 'mask' are changed to the desired 'state'. For a permanent/static change the corresponding *Pulse Time* must be 0000. To pulse Relay A ON for 5 seconds the Pulse Time field would be set to 5000 milliseconds which is represented as 1388 hexadecimal. The "Hex" string for this command would be "01010000000000001388". Note that the mask indicates the target relay. The state indicates the desired change and the length of the pulse in milliseconds is defined.

References

[1] JNIOR Protocol Specification, INTEG Process Group, Inc.

[2] Registry Key Assignments, INTEG Process Group, Inc.

[3] RFC 6455, Internet Engineering Task Force Standard Track, ISSN: 2070-1721, Dated December 2011. https://tools.ietf.org/html/rfc6455#ref-WSAPI

[4] The JSON Data Interchange Standard, http://www.json.org/

[5] Wireshark Network Protocol Analyzer, https://www.wireshark.org/