# JNIOR Simulator

## Overview

The JNIOR Simulator is designed to simulate some of the I/O functionality of the JNIOR. The JNIOR Simulator was developed as a backend DLL that can be connected to various frontends. Frontends can be developed by INTEG or its clients.

Note that a PC running the simulator will be much faster than a JNIOR. This can cause the monitor packets to arrive more frequently than would be seen in normal JNIOR usage. Also monitor packets will arrive almost immediately in response to an I/O change where the speed of the JNIOR would introduce some delay.

The JNIOR Simulator starts a TCP Server for each JNIOR it is going to simulate. The number of JNIORs that are going to be simulated is passed in as an argument in the `StartServer()` function described later. When a new client is connected the JNIOR Simulator creates a new Connection object to handle the client communications. Each JNIOR being simulated can have multiple client connections.

As mentioned, only some of the JNIOR functionality was simulated. Specifically the I/O monitor packet delivery is simulated. In order to receive the monitor packet a client must be connected and logged in with valid credentials. Once the login is successful a monitor packet will be delivered. A monitor packet will also be delivered whenever there is a change in I/O.

## Use Cases

A user wants to implement the JNIOR Protocol or utilize the JNIOR DLL without the physical JNIOR.

A user wants to simulate $x$ JNIORs in the field without the presence of those JNIORs.

## Requirements

**Server**

1. Must utilize TCP/IP
2. Must impersonate the JNIOR side of the JNIOR Protocol

3. Must be able to simulate multiple JNIORs
4. Must be able to accept multiple connections per simulated JNIOR
5. Must assign each simulated JNIOR a new port number starting at the port number specified during the startup of the server

**JNIOR Functions**

1. Must be able to set a single input on a simulated JNIOR
2. Must be able to set multiple inputs on a simulated JNIOR
3. Must be able to get the status of a single input on a simulated JNIOR
4. Must be able to get the status of all of the inputs on a simulated JNIOR
5. Must be able to set a single output on a simulated JNIOR
6. Must be able to set multiple outputs on a simulated JNIOR
7. Must be able to pulse a single output on a simulated JNIOR
8. Must be able to pulse multiple outputs on a simulated JNIOR
9. Must queue pulses as the JNIOR would
10. Must be able to get the status of a single output on a simulated JNIOR
11. Must be able to get the status of all of the outputs on a simulated JNIOR
12. Must be able to keep track of counters
13. Must be able to keep track of usage meters
14. Must be able to simulate a JNIOR disconnecting

User Interface

15. Must allow the user to select which network adapter to bind to
16. Must allow the user to select a starting port for a block of ports
17. Must allow the user to select the number of JNIORs to simulate
18. Must allow the user to set the state of a single input
19. Must allow the user to set the state of multiple inputs
20. Must allow the user to set the state of a single output
21. Must allow the user to set the state of multiple outputs

## Structure Definitions

**NOTE: All structures are 1-byte aligned**

**CONNECT_PARAMS**

The `CONNECT_PARAMS` provides an object that holds the information on how to set up the server. The host field indicates the network adapter that the server will bind to. The port field is the starting port number for the block of ports used for all simulated JNIORs. For example, if you start with port 9200 and are simulating 8 JNIORs then ports 9200 – 9207 will be used.

```
#pragma pack(1)
struct CONNECT_PARAMS {
	char* host;
	int port;
};
```

**SERVER_INFO**

This structure is passed in via the `SetServerInfoCallback()` function described later. The `SERVER_INFO` structure contains the information that is passed back as the `void* args` argument in the `ServerInfoNotify()` method. It contains information aobut the simulated JNIOR as well as the `CONNECT_PARAMS` structure and how many clients are connected.

```
#pragma pack(1)
struct SERVER_INFO {
	CONNECT_PARAMS* cp;
	int connectedClients;
	int handle;
	int inputs;
	int outputs;
};
```

## Function Prototypes

**Start Server**

Starts the Simulator servers according to the `CONNECT_PARAMS` and the count parameters

```
extern "C" JNIORDLL_API int StartServer(
	CONNECT_PARAMS* cp,
	int count);
```

**Set Input Count**

Sets the number of inputs on the simulated JNIOR.

```
extern "C" JNIORDLL_API int SetInputCount(
      int handle,
      int inputCount);
```

**Get Input Count**

Gets the number of inputs on the simulated JNIOR.

```
extern "C" JNIORDLL_API int GetInputCount(
      int handle);
```

**Get Input**

Returns the status of a single input. Either 0 indicating low or off or 1 indicating high or on.

```
extern "C" JNIORDLL_API int GetInput(
      int handle,
      int channel);
```

**Get Inputs**

Gets the status of all of the 8 inputs. A binary representation of the inputs states is returned. Input 8 is the MSB.

```
extern "C" JNIORDLL_API int GetInputs(
      int handle);
```

**Set Input**

Sets a single input channel to the new desired state. 0 is returned if the command is successful. Once an input state is changed a new monitor packet will be sent out to all connected clients.

```
extern "C" JNIORDLL_API int SetInput(
      int handle,
      int channel,
      int state);
```

**Set Inputs**

Sets multiple inputs according the the channel mask and states mask. Inputs not defined by the channel mask are not affected. 0 is returned if the command is successful. Once an input state is changed a new monitor packet will be sent out to all connected clients.

```
extern "C" JNIORDLL_API int SetInputs(
      int handle,
      int channelMask,
      int stateMask);
```

**Set Output Count**

Sets the number of outputs on the simulated JNIOR.

```
extern "C" JNIORDLL_API int SetOutputCount(
      int handle,
      int outputCount);
```

**Get Output Count**

Gets the number of outputs on the simulated JNIOR.

```
extern "C" JNIORDLL_API int GetOutputCount(
      int handle);
```

**Get Output**

Returns the status of a single output. Either 0 indicating low or off or 1 indicating high or on.

```
extern "C" JNIORDLL_API int GetOutput(
      int handle,
      int channel);
```

**Get Outputs**

Gets the status of all of the 8 outputs. A binary representation of the inputs states is returned. Output 8 is the MSB.

```
extern "C" JNIORDLL_API int GetOutputs (
      int handle);
```

**Set Output**

Sets a single output channel to the new desired state. 0 is returned if the command is successful. Once an output state is changed a new monitor packet will be sent out to all connected clients.

```
extern "C" JNIORDLL_API int SetOutput(
      int handle,
      int channel,
      int state);
```

**Set Outputs**

Sets multiple outputs according the the channel mask and states mask. Outputs not defined by the channel mask are not affected. 0 is returned if the command is successful. Once an output state is changed a new monitor packet will be sent out to all connected clients.

```
extern "C" JNIORDLL_API int SetOutputs(
      int handle,
      int channelMask,
      int stateMask);
```

**Disconnect**

This will cause the server for the specified JNIOR handle to stop and then restart. The net effect will be that all connected clients will be disconnected from the simulated JNIOR.

```
extern "C" JNIORDLL_API int Disconnect (
      int handle);
```

**Set Server Info Callback**

This function sets the listener method for server events. This functionality is optional but if desired this function **MUST** be called before the `StartServer()` function.

```
extern "C" JNIORDLL_API int SetServerInfoCallback(
      CALLBACKNOTIFY lProcAddress);
```

The `CALLBACKNOTIFY` tpye is defined as follows

```
typedef void (CALLBACK* CALLBACKNOTIFY)(void* args);
```

## Parameter definitions

| | |
|---|---|
| handle | The index of the simulated JNIOR. 0 based index. |
| channel | The channel of the I/O point |
| state | The state of the I/O point |
| channelMask | The binary representation of which 8 inputs or outputs are affected by the command. This value will be 0 – 255. Channel 8 is the MSB. |
| stateMask | The binary representation of the 8 input or output states. This value will be 0 – 255. Channel 8 is the MSB. |