INTEG PROCESS GROUP, INC.

JNIOR DLL

JNIOR DLL and .NET Wrapper Developers Guide

Updated: 12/22/2009

This document is intended to familiarize a developer with the JNIOR DLL and its architecture. This document has sample snippets written in C++ and C#.

Contents

JNIOR DLL

The JNIOR DLL is designed to expedite the deployment of JNIORs in a custom Windows solution. The JNIOR DLL implements the JNIOR Protocol in a module that can be used by developers writing custom applications. This frees up the developer from having to implement communications between the JNIOR and their application and lets them focus on their solutions business logic.

The JNIOR Protocol is a binary protocol implemented over TCP. The DLL encapsulates the functionality of the protocol in a DLL and provides function calls to interact with the JNIOR. To learn more about the protocol please read the JNIOR Communications Protocol document. Describing the protocol is beyond the scope of this document.

The JNIOR DLL is written in C++ and implements the JNIOR communications. There is a .NET wrapper DLL written in C#. A developer that wants to implement JNIOR communications only needs the JNIOR DLL. .NET developers might want the additional JniorDll.Net.dll file as well as the necessary JniorDll.dll file. The .NET wrapper provides many useful benefits;

- Easy to use and understand classes in an object oriented approach
- Performs data marshalling between .NET and COM
- Defines delegates for creating callback functions
- Contains a few UI controls that are useful for JNIOR operations

NOTE: Before running any of the sample applications please be sure that doing so will not have any adverse effects. If a JNIOR is connected to a machine and a sample application commands a relay INTEG is not responsible for any action that is taken. Please only use the samples on a JNIOR that is not connected to any external equipment unless you, the developer, are fully aware of the logic behind the application.

Using JniorDll.dll

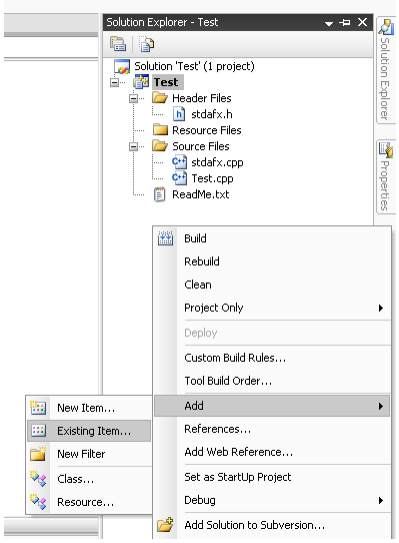
There are 4 files that you can include in your project.

- **JniorDll.h**
  Contains all of the function prototypes
- **JniorConstants.h**
  Contains the definitions of many constant values
- **JniorStructures.h**
  Contains definitions of the structures that are used in the DLL and that can be used by a developer

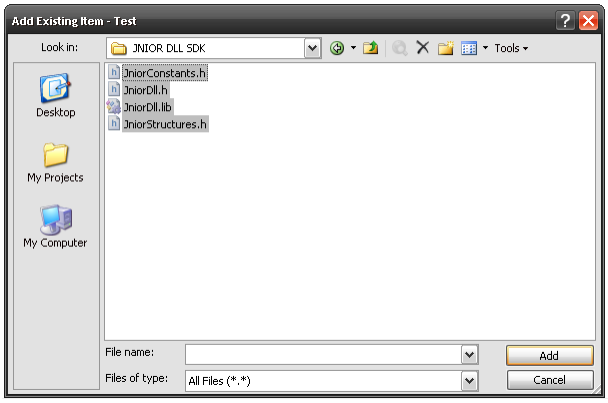
- **JniorDll.lib**
  Necessary for compiling your application

How to include the SDK files in Visual Studio

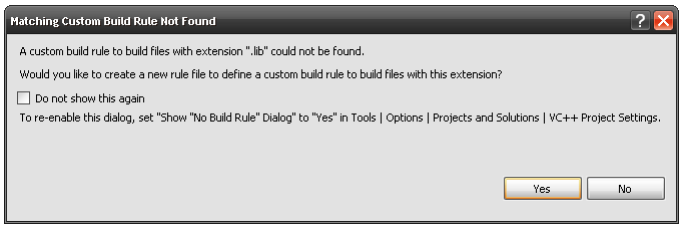
Right click on the solution explorer and go to add existing.



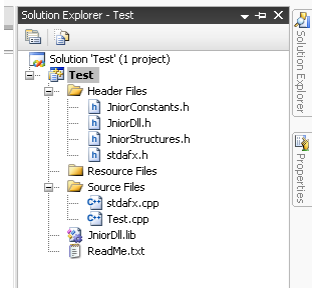
Navigate to the SDK package and select the four files listed above. Under the “Files of type” drop down select “All Files”.



You can say “No” if you see the following dialog “Matching Custom Build Rule Not Found”



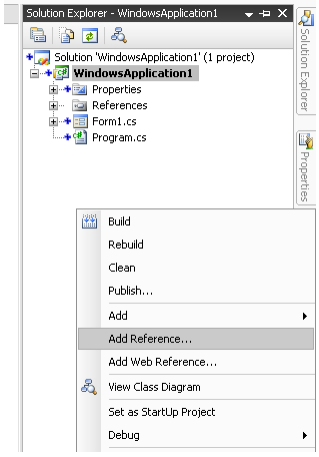
Your Solution Explorer will now look something like the following



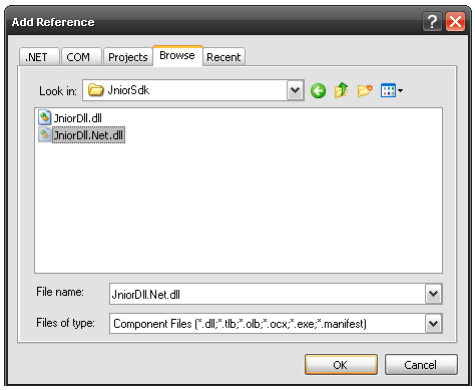
You are now ready to program for the JNIOR using the COM JNIOR DLL

Using JniorDll.Net.dll

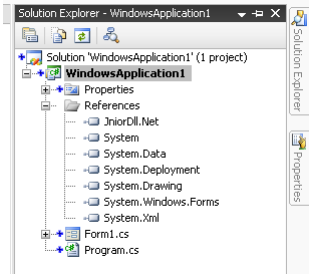
Add a reference to your .NET project. Right click in the solution explorer window and go to “Add Reference…”



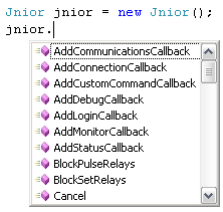
Navigate to the SDK package and select the JniorDll.Net.dll file



Click OK and your solution explorer will look like the following



You are now ready to program using the JNIOR .NET DLL. Once the .NET reference is associated IntelliSense will become available. IntelliSense is Microsoft's implementation of auto-completion. It will help you in writing applications for the JNIOR using the .NET wrapper.



Synchronous vs. Asynchronous

Most things we do in life are synchronous tasks. This means we have to complete the task before we can start on the next task. Take loading the dishwasher for example. We must fully unload the dishwasher before reloading it. If we don't fully unload the dishwasher then the clean dishes that are still in the machine will get dirty as the dirty dishes are added. Sticking with the kitchen theme, cooking is full of asynchronous tasks. We don't want to have to cook all of the sides before we start the main course or the sides would be cold be the time the main dish is ready. We want to use asynchronous processing when we have sections of code that will take a considerable amount of time to complete. UI programming is an area where we need to use asynchronous processing. We don't want the interface to lock up even for short periods of time. Having the interface lock up would lead to a poor user experience. The JNIOR DLL has taken this in account during its design and offers asynchronous functions. Connecting and Logging in can be quite time consuming.

Callbacks

There are callbacks defined for the following events. A callback is executable code that is passed as an argument to other code. It allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer. All callbacks are executed in a separate thread to guarantee that DLL processing does not stop if there is bad code in the client application. Multiple callbacks can be defined for the following events with the exception of the Accepted Connection callback.

- Connection Events
- Log In Events

- Monitor Packet Events
- Registry Key Events
- Communication Events
- Accepted Connection Events (DLL as a Server)

The callbacks use the `__stdcall` calling convention. The JNIOR DLL has defined a `CALLBACK` macro that can be used in place of `__stdcall` to produce the following function prototype.

```
return-type CALLBACK function-name[(argument-list)]
```

Logging

The JNIOR DLL has the option to enable logging. To enable logging simply create a “logs” directory in the same directory that contains the JNIORDLL.dll file.

Heartbeat

The JNIOR Connection should remain alive as long as it is required. There is overhead associated with establishing a new connection and logging in. The JNIOR protocol connection requires some sort of communication to be received from a client at least once every 15 minutes or the JNIOR will close the connection. Client communication could be in the form of a command, request, or any single byte other than 0x01. The JNIOR protocol recommends that you use the 0x06(ACK) byte.

The JNIOR DLL takes care of sending a “heartbeat” or keep alive once every 10 seconds. 10 seconds was chosen because the JNIOR OS was written to not use memory resources if client data is received at least every 15 seconds.

Features

General

Getting the DLL Version

It is wise in an application to show the version strings of not only your application but the version numbers of all third party tools as well. This is useful in error reporting. The JNIOR DLL provides the `GetDllVersion(char*)` function to expose the version string of the JNIOR DLL. Simply pass in a pointer to a string that is big enough to hold the version string. The .NET wrapper hides the need to pass in a string as a parameter and instead just returns the version string. The version string is in the format major.minor.date.time. For example version 3.1.1221.1421 is version 3.1 built on December 21st at 2:21pm.

C++

```
// show the version string
char* version = new char[32];
result = GetDllVersion(version);
```

C#

```
string version = Jnior.GetDllVersion();
```

You can see that the .NET wrapper simplifies coding. The actual code that the wrapper performs is as follows.

```
StringBuilder dllVersion = new StringBuilder();
jnior_dll_calls.GetDllVersion(dllVersion);
return dllVersion.ToString();
```

Getting a Description of a Status Code

When calling functions in the DLL, many of them return integer values that have some meaning. All of these values are defined in the JniorConstants.h file. To get a description of the meaning you can call the `GetStatusDescription(int)` function. It will return a string representing a user friendly description of the supplied status code.

C++

```
// return some status
int status = someFunction();
// get the description
char* version = GetStatusDescription(status);
```

C#

```
// return some status
int status = someFunction();
// get the description
jnior_dll_calls.GetStatusDescription(status).ToString();
```

Local Server for Incoming Connections

The JNIOR DLL also provides the ability to accept incoming connections. You can configure the JNIOR to attempt to establish a connection with a server that will host the JNIOR Protocol connection, in this case the JNIOR DLL. Once this connection is established it will behave as if the PC established the connection. If there is an anonymous login level set then the login acknowledgement packet is sent to the PC. The PC side application does not need to perform an additional login.

This architecture might be of interest if there are several JNIORs in an installation but may be behind a firewall or there IP Addresses are not visible. If the PC has a visible public IP Address then those JNIORs could be configured to initiate the connection with the JNIOR DLL. If the JNIOR and the server are on different networks then the gateway address must be set in the JNIOR. For further network help please contact your networks administrator.

To enable this functionality on the JNIOR set the JniorServer/RemoteIP registry key.

```
registry JniorServer/RemoteIP = 10.0.0.100
```

To enable this functionality in the JNIOR DLL the `StartServer(CONNECT_PARAMS*)` function must be called. The parameter passed to this function is a pointer to a CONNECT_PARAMS structure. The CONNECT_PARAMS structure contains an IP Address and the port number of the network adapter to bind to. **The other fields in the CONNECT_PARAMS structure are ignored. This is not the actual case but should it be? The If the connection is lost both the JNIOR and the DLL will try to reconnect. This is wrong, right?**

C++

```
// Create a connect params structure to define server information
CONNECT_PARAMS* cp = new CONNECT_PARAMS();
cp->host = "10.0.0.100";
cp->port = 9200;

// Start Server
StartServer(cp);
```

C#

```
// start the server
ConnectionProperties cp = new ConnectionProperties("10.0.0.100", 9200, 0, 0);
Jnior.StartServer(cp);
```

To alert the client application when a JNIOR successfully connects there is a Connection Accepted Callback. To set this callback use the `SetAcceptedConnectionCallback()` function.
The `SetAcceptedConnectionCallback()` function should be called before the `StartServer()` function. This will ensure that no incoming connections are missed.

C++

```
// define the callback function
void CALLBACK AcceptedConnectionCallback(int handle) {
        cout << "New JNIOR Connected, the handle is: " << handle << endl;
}

// set the callback function
SetAcceptedConnectionCallback(&AcceptedConnectionCallback);
```

C#

```
// define the callback function
private void onAcceptedConnection(Jnior jnior)
{
        Console.WriteLine("New JNIOR Connected, the handle is: " + handle);
}

// Set the callback function
jnior_dll_calls.AcceptedConnectionCallback callback;
callback = new jnior_dll_calls.AcceptedConnectionCallback(onAcceptedConnection));
Jnior.SetAcceptedConnectionCallback(callback);
```

Creating a Session

A session must be created before the JNIOR can be accessed. We create a session to let the DLL assign us a unique number that we can use in case we want to use multiple JNIORs with the DLL at one time. The unique number will be called a handle. The handle is just a 4 byte integer. The handle must be passed to the DLL with each call. The .NET Wrapper hides this from the developer and it is automatically passed bases on the object oriented nature of .NET.

C++

```
// create the jnior session
int handle = CreateJniorSession(NULL);
```

C#

```
Jnior m_jnior = new Jnior();
```

Connecting

As stated the JNIOR is connected to using TCP/IP. TCP is a connection oriented protocol. This means that we must establish a connection before sending or expecting to receive any commands. To connect we need to pass the HANDLE of the session to connect to and the pointer to the `CONNECT_PARAMS` object. The `CONNECT_PARAMS` object holds the information about the address of the JNIOR and some connection rules such as a retry count and a retry interval.

C++

```
// create our connection parameters
CONNECT_PARAMS* cp = new CONNECT_PARAMS();
cp->host = "10.0.0.146";
cp->port = 9200;
cp->retryCount = 0;
cp->retryInterval = 0;

// connect to the jnior using the handle
cout << "\r\nConnecting...\r\n";
int result = Connect(handle, cp);
```

C#

```
ConnectionProperties cp = new ConnectionProperties("10.0.0.146", 9200, 0, 0);
m_jnior.Connect(cp);
```

The result from calling this function will be one of the following values as defined in the JniorConstants.h file.

```
#define STATUS_CONNECTING                   4
#define STATUS_CONNECTED                    6
#define STATUS_CONNECTION_FAILED            7
#define STATUS_CANCELLED                    8
```

Connect Asynchronously

To perform the Connection asynchronously we must call the `ConnectAync()`function with the same signature. Behind the scenes the DLL will create another thread to handle the connection. The `ConnectAsync()` call will return right away reporting no error.

C++

```
// create our connection parameters
CONNECT_PARAMS* cp = new CONNECT_PARAMS();
cp->host = "10.0.0.146";
cp->port = 9200;
cp->retryCount = 0;
cp->retryInterval = 0;

// connect to the jnior using the handle
cout << "\r\nConnecting...\r\n";
int result = ConnectAsync(handle, cp);
```

C#

```
ConnectionProperties cp = new ConnectionProperties("10.0.0.146", 9200, 0, 0);
        m_jnior.ConnectAsync(cp);
```

Processing will continue in the background. In order to be alerted of the status of the connection attempt we can either poll the `IsConnected()` function or define a Callback that will be called when there is a status update. Polling the `IsConnected()` function is affectively the same as blocking.

C++

```
bool result = IsConnected(handle);
```

C#

```
Bool isConnected = m_jnior.IsConnected();
```

Define callback

To define the callback we must define a function with the `CALLBACKNOTIFY` signature. The `CALLBACKNOTIFY` is defined as follows:

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

The `args` parameter should be typecast to `STATUS_CALLBACK_ARGS`.

C++

We must first define a function to act as our callback.

```
void CALLBACK ConnectionCallback(void* args) {
        STATUS_CALLBACK_ARGS* statusArgs = (STATUS_CALLBACK_ARGS*)args;
        cout << statusArgs->message << endl;

        // if we are connected then we can login
        if (statusArgs->status == STATUS_CONNECTED) {
                // do something...
        }
}
```

Now we assign the callback to our session.

```
// add a callback to receive the status of the connection operation
AddConnectionCallback(handle, &ConnectionCallback);
```

C#

We must first define a function to act as our callback. Our .NET wrapper again hides some critical functionality and exposes the following delegate as our function signature.

```
public delegate void StatusCallback(Jnior jnior, StatusArgs args);
```

This means our callback function does not need to typecast the `args` parameter.

```
private void OnConnectionNotify(Jnior jnior, StatusArgs args)
{
        Console.WriteLine(args.messege);

        if (args.status == STATUS_CONNECTED) {
                // do something...
        }
}
```

Now we assign the callback to our session.

```
// add a connection callback
m_jnior.AddConnectionCallback(new jnior_dll_calls.StatusCallback(OnConnectionNotify));
```

Get Connection Properties

If it is ever desired to get the properties that were used in establishing the connection one can call the `GetConnectionProperties()` function.

C++

You will need to declare a pointer to a CONNECTION_PARAMS object that can be passed as the second parameter.

```
// declare a CONNECT_PARAMS object
CONNECT_PARAMS* cp = new CONNECT_PARAMS();
// call GetConnectionProperties
GetConnectionProperties(handle, cp);
```

C#

```
ConnectionProperties cp = m_jnior.GetConnectionProperties();
```

Disconnecting

To disconnect from the JNIOR call the `Disconnect()` function.

C++

```
Disconnect(handle);
```

C#

```
m_jnior.Disconnect();
```

Reboot

To reboot the JNIOR call the `Reboot()` function. When the `Reboot()` function is called and the connection is closed the DLL will wait for 90 seconds before trying to reestablish the connection. It is known that the JNIOR should take a little over 90 seconds to perform the reboot process.

C++

```
Reboot(handle);
```

C#

```
m_jnior.Reboot();
```

Logging In

Logging in is an essential operation in the JNIOR Protocol. The login can be optionally disabled but is enabled by default. You can also optionally leave the login enabled but specify an anonymous login level that will be granted to all connections. To login we need to pass the HANDLE of the session and the pointer to the `CREDENTIALS` object. The `CREDENTIALS` object holds the information about the login options.

Login Levels

There are three login levels on the JNIOR. The three levels are guest, user, and administrator. The user login level allows monitoring of I/O only and has a login level of 0. The user login level allows you to monitor and control the I/O and has a login level of 64. The administrator login level allows monitoring and controlling the I/O as well as configuring the JNIOR. The administrator login level is 128.

Anonymous

If the JniorServer/Anonymous key has been defined in the Registry an anonymous login will be allowed. An anonymous login request contains both blank 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).

Predefined Login Credentials

Each login level has at least one predefined credential. The guest credential is “guest” for the username and “guest” for the password. The user credential is “user”, “user”. Lastly the administrator login level has two credentials. The first one is, you guessed it “admin”, “admin”. The second predefined credential is “jnior”, “jnior”.

C++

```
// create our credentials object.  use the default jnior, jnior login
// credential for an admin login.
CREDENTIALS* creds = new CREDENTIALS();
creds->username = "jnior";
creds->password = "jnior";

// login using our handle obtained earlier
int result = Login(handle, creds);
```

C#

```
// create our credentials object.  use the default jnior, jnior login
// credential for an admin login.
LoginProperties lp = new LoginProperties(txtUserName.Text, txtPassword.Text);

// perform the login
m_jnior.Login(lp);
```

Encoding

By default the Username and Password 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 encode can be enabled. This renders the login information in a format that is not easily read by humans. 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. To enable encoding set the encode field of the `CREDENTIALS` object to true.

C++

```
// create our credentials object.  use the default jnior, jnior login
// credential for an admin login.
CREDENTIALS* creds = new CREDENTIALS();
creds->username = "jnior";
creds->password = "jnior";
creds->encode = true;

// login using our handle obtained earlier
int result = Login(handle, creds);
```

C#

```
// create our credentials object.  use the default jnior, jnior login
// credential for an admin login.
LoginProperties lp = new LoginProperties(txtUserName.Text, txtPassword.Text);
Lp.Encode = true;

// perform the login
m_jnior.Login(lp);
```

Log In Asynchronously

Logging in to the JNIOR can take a few seconds. This is a situation where if called from the UI thread the UI would become unresponsive. The JNIOR DLL provides an asynchronous call for logging in. The asynchronous login function has the same signature as the login function but the name in `LoginAsync`. Again we should define a callback to accept the login state events.

C++

```
// create our credentials object.  Use the default jnior, jnior login
// credential for an admin login.
CREDENTIALS* creds = new CREDENTIALS();
creds->username = "jnior";
creds->password = "jnior";

// login using our handle obtained earlier
int result = LoginAsync(handle, creds);
```

C#

```
// create our credentials object.  Use the default jnior, jnior login
// credential for an admin login.
LoginProperties lp = new LoginProperties(txtUserName.Text, txtPassword.Text);

// perform the login
m_jnior.LoginAsync(lp);
```

The status of the login can be checked by calling the `IsLoggedIn()` function. Calling this function will simply return a Boolean value indicating whether the session is successfully logged into a guest, user, or administrator account.

C++

```
bool result = IsLoggedIn(handle);
```

C#

```
Bool isLoggedIn = m_jnior.IsLoggedIn();
```

To determine the login level of the session call the `GetLoginStatus()`. This function returns on of the following values as defined in the JniorConstants.h file.

```
#define STATUS_NOT_LOGGED_IN                100
#define STATUS_LOGGED_OUT                   101
#define STATUS_LOGGING_IN                   102
#define STATUS_LOGGED_IN_GUEST              103
#define STATUS_LOGGED_IN_USER               104
#define STATUS_LOGGED_IN_ADMIN              105
#define STATUS_LOGIN_FAILED                 106
#define STATUS_LOGIN_INCORRECT_LEVEL        107
```

C++

```
int loginStatus = GetLoginStatus(handle);
```

C#

```
int loginStatus = m_jnior.GetLoginStatus();
```

Define callback

To define the callback we must define a function with the `CALLBACKNOTIFY` signature. The `CALLBACKNOTIFY` is defined as follows:

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

The `args` parameter should be typecast to `STATUS_CALLBACK_ARGS`.

C++

We must first define a function to act as our callback.

```
void CALLBACK LoginCallback(void* args) {
	STATUS_CALLBACK_ARGS* statusArgs = (STATUS_CALLBACK_ARGS*)args;
	cout << statusArgs->message << endl;

	// if we are connected then we can login
	if (statusArgs->status == STATUS_LOGGED_IN_ADMIN) {
		// do something...
	}
}

// add a callback to receive the status of the login operation
AddLoginCallback(handle, &LoginCallback);
```

C#

We must first define a function to act as our callback. Our .NET wrapper again hides some critical functionality and exposes the following delegate as our function signature.

```
public delegate void StatusCallback(Jnior jnior, StatusArgs args);
```

This means our callback function does not need to typecast the `args` parameter.

```
private void OnLoginNotify(Jnior jnior, StatusArgs args)
{
	Console.WriteLine(args.messege);

	if (args.status == STATUS_LOGGED_IN_ADMIN) {
		// do something...
	}
}

// Now we assign the callback to our session.
m_jnior.AddLoginCallback(new jnior_dll_calls.StatusCallback(OnLoginNotify));
```

Logging out

To logout we can try login but with “bad” credentials. A blank username and password is not always a bad credential. In the case where an anonymous login is set, blank username and passwords are sent.

Internal I/O

Internal I/O is represented by the Monitor Packet. The internal I/O of the JNIOR model 310 is 8 optically isolated digital inputs and 8 dry contact relays. The inputs can act as digital counters and can count accurately up to 2000 counts per second. Both the inputs and outputs have usage meters that are accurate to a resolution of a millisecond.

Monitor Packet

The monitor packet contains information pertaining to the states of the Internal I/O. The values for the Inputs are the state, counters and alarm states. The values for the outputs are just the state of the

output. A monitor packet is once the login in completed successfully and anytime there is a change to the internal I/O.

Define Callback

C++

```
void CALLBACK MonitorCallback(void* args) {
	if(WaitForSingleObject(hMonitorCallbackMutex, 10000)==WAIT_TIMEOUT) {
		return;
	}

	MONITOR_CALLBACK_ARGS* monitorArgs = (MONITOR_CALLBACK_ARGS*)args;

	char inputStates[9];
	char outputStates[9];

	inputStates[8] = outputStates[8] = '\0';

	for (int i = 7; i >= 0; i--) {
		inputStates[i] = monitorArgs->monitor->inputs[i].state + '0';
		outputStates[i] = monitorArgs->monitor->outputs[i].state + '0';
	}

	time_t time = (time_t)monitorArgs->monitor->jniorTime / 1000;
	char log[128];
	sprintf(log, "Inputs: %s, Outputs: %s, Time: %s\r\n", inputStates, outputStates,
	ctime(&time));
	cout << log;

	ReleaseMutex(hMonitorCallbackMutex);
}

// add a callback to receive the status of the connection operation
AddMonitorCallback(handle, &MonitorCallback);
```

C#

```
private void OnMonitorNotify(Jnior jnior, MonitorArgs args)
{
	StringBuilder sbInputs = new StringBuilder();
	StringBuilder sbOutputs = new StringBuilder();

	int inputs = jnior.GetInputs();
	int outputs = jnior.GetOutputs();

	for (int i = 7; i >= 0; i--)
	{
		sbInputs.Append(jnior.GetInput(i));
		sbOutputs.Append(jnior.GetOutput(i));
	}

	DateTime jniorTime = Jnior.EPOCH.AddMilliseconds(args.Monitor.jniorTime);
	string timeString = jniorTime.ToLocalTime().ToString("M/dd/yyyy HH:mm:ss.fff");

	Console.WriteLine("Inputs: " + sbInputs.ToString() + ", Outputs: " +
		sbOutputs.ToString() + ", Time : " + timeString);
}

// add the monitor callback
m_jnior.AddMonitorCallback(new jnior_dll_calls.MonitorCallback(OnMonitorNotify));
```

Disable / Enable Monitor Packets

When there are situations that cause changes to the internal I/O very rapidly it is recommended that

Monitor Packets are disabled. To do this we call the `DisableMonitorPackets()` for a JNIOR session. The Monitor Packets can be re-enabled at any time by using the `EnableMonitorPackets()` function.

C++

```
int result = DisableMonitorPackets(handle);
int result = EnableMonitorPackets(handle);
```

C#

```
m_jnior.DisableMonitorPackets();
m_jnior.EnableMonitorPackets();
```

Scheduling the Monitor Packet

The monitor packet can be scheduled to come on a specified interval. In some cases you might want to guarantee that you application hears from the JNIOR at least every 30 seconds for example. Using the `RequestMonitorPacket()` function can accomplish this task. The scheduled monitor packet can be used instead of the unsolicited monitor packet or in addition to it. The second parameter in the `RequestMonitorPacket()` function is the integer representing a time interval in milliseconds. If you specify a value of zero then you will receive a single monitor packet.

C++

```
int result = RequestMonitorPacket(handle, 30000);
```

C#

```
m_jnior.RequestMonitorPacket(30000);
```

Get Monitor Freshness

Sometimes you might want to know how long ago the last monitor packet was received. For this you would call the `GetMonitorFreshness()` function. This function simply returns the number of milliseconds since the last monitor packet was received.

C++

```
long long millis = GetMonitorFreshness(handle);
```

C#

```
long millis = m_jnior.GetMonitorFreshness();
```

Get Model of the JNIOR

Currently there is only the JNIOR Model 310. The model of the JNIOR is embedded in the monitor packet and can be retrieved via the `GetModel()` function. You need to pass a string pointer as the second parameter to this function.

C++

```
// some code
```

C#

```
// some code
```

Get OS Version of the JNIOR

The version of the operating system that is currently running on the JNIOR can be obtained by calling the `GetSoftwareVersion()` function. You need to pass a string pointer as the second parameter to this function.

C++

```
// some code
```

C#

```
// some code
```

Get Known JNIOR Time

If you want to know the time on the JNIOR you can call the `GetKnownJniorTime()` function. This function returns the number of milliseconds since January 1st 1970 as reported by the last monitor packet received. Note, the JNIOR is not asked for its time during this call. **SHOULD IT BE?**

C++

```
long long millis = GetKnownJniorTime(handle);
```

C#

```
long millis = m_jnior.GetKnownJniorTime();
```

Set JNIOR Time

The JNIOR can be configured to synchronize its time with a Network Time Protocol Server (NTP). In cases where this is not possible, the time can be set by calling the `SetClock()` function. This function takes a parameter that specifies the number of milliseconds since January 1st, 9170.

C++

```
// some code
```

C#

```
// some code
```

Get Inputs States

To get the status of the inputs you have three options. Your first option is to call the `GetMonitor()` function which returns a pointer to a `JNIOR_MONITOR` structure. You can then access the fields in the `JNIOR_MONITOR` structure to get the states. Option 2 is to call the `GetInputs()` function. This function returns a binary mask representing the state of all of the inputs in one value. Lastly you can call the

`GetInput()` function and pass in the channel of the input you want the state of. This parameter is a zero based index of the channels.

C++

```
// option 1
JNIOR_MONITOR* monitor = GetMonitor(handle);
for (int i = 0; i < 8; i++) {
	int state = monitor->inputs[i].state;
}

// option 2
int states = GetInputs(handle);

// option 3
int state = GetInput(handle, 0);
```

C#

```
// some code
```

Get Outputs

To get the status of the outputs you have three options. Your first option is to call the `GetMonitor()` function which returns a pointer to a `JNIOR_MONITOR` structure. You can then access the fields in the `JNIOR_MONITOR` structure to get the states. Option 2 is to call the `GetOutputs()` function. This function returns a binary mask representing the state of all of the outputs in one value. Lastly you can call the `GetOutput()` function and pass in the channel of the output you want the state of. This parameter is a zero based index of the channels.

C++

```
// option 1
JNIOR_MONITOR* monitor = GetMonitor(handle);
for (int i = 0; i < 8; i++) {
	int state = monitor->outputs[i].state;
}

// option 2
int states = GetOutputs(handle);

// option 3
int state = GetOutput(handle, 0);
```

C#

```
// some code
```

Close Output

This function will close a single output. If the output is already closed then no physical change occurs. This will cause the JNIOR to generate a monitor packet whether a change in I/O occurs or not.

C++

```
// some code
```

C#

```
// some code
```

Open Output

This function will open a single output. If the output is already open then no physical change occurs. This will cause the JNIOR to generate a monitor packet whether a change in I/O occurs or not.

C++

```
// some code
```

C#

```
// some code
```

Toggle Output

This function will toggle a single output. If the output is closed then it will open. If the output is currently open then it will be closed. Note that using toggle is not programmatically sound in many cases as the current state is not known and it is possible that the code on the client and the actual JNIOR output state can get "out of sync". It is best practice to explicitly set the state of the output.

C++

```
// some code
```

C#

```
// some code
```

Pulse Output

This function will pulse a single output closed for a given period of time defined in milliseconds. If the output is already closed then no physical change occurs but the pulse timer will still count down. Only one pulse can be processed at a time. If a pulse is in progress and another pulse is commanded the latter pulse will be queued and will commence once the original pulse is completed. A total of 31 pulses can be queued.

C++

```
// some code
```

C#

```
// some code
```

Set Output

This function is much like the `CloseOutput()` and `OpenOutput()` functions. It combines the functionality of those two functions into one. The third parameter in this function accepts either a zero or a one specifying if the output should the open (zero) or closed (one).

C++

```
// some code
```

C#

```
// some code
```

Block Set Relays

Multiple relays can be commanded to open or close simultaneously using the `BlockSetRelays()` function by specifying a channel mask, which channels are affected by this operation, and a states mask, the new states of the affected channels.

C++

```
// some code
```

C#

```
// some code
```

Block Pulse Relays

Multiple relays can be commanded to pulse open or closed simultaneously using the `BlockPulseRelays()` function by specifying a channel mask, which channels are affected by this operation, and a states mask, the new states of the affected channels. A third parameter specifies the duration in milliseconds that the pulse should last.

C++

```
// some code
```

C#

```
// some code
```

Reset Input Latch

Latching is the process of remembering the state of the input. If an input state must remain high for a long period of time but the actual state of the signal is brief then an input latch can be configured to provide this functionality. If an input state must be held high for some period of time until some software checks on the state the software can clear the latch once it is done processing the state. To accomplish clearing the latch we use the `ResetInputLatch()` function.

C++

```
// some code
```

C#

```
// some code
```

Clear Input Counter

As discussed before the inputs on the JNIOR can act as counters. Each counter is an unsigned 4 byte integer. The value will range from 0 – 4,294,967,296. Once the counter goes beyond 4,294,967,296

then rollover will occur and resume counting at 0. A custom application can reset the counter value to 0 at any time using the `CelarInputCounter()` function. The value can be set to a non-zero value using the jrmon command line command. Please refer to the Command Line Interface document.

C++

```
// some code
```

C#

```
// some code
```

Clear Input Usage

As discussed before the inputs on the JNIOR have usage meters. Each usage meter is a signed 8 byte integer. The value will range from 0 – 9,223,372,036,854,775,807. A custom application can reset the usage meter at any time using the `ClearInputUsage()` function.

C++

```
// some code
```

C#

```
// some code
```

Clear Output Usage

As discussed before the outputs on the JNIOR have usage meters. Each usage meter is a signed 8 byte integer. The value will range from 0 – 9,223,372,036,854,775,807. A custom application can reset the usage meter at any time using the `ClearInputUsage()` function.

C++

```
// some code
```

C#

```
// some code
```

External Devices

The External Devices are not implemented in this DLL.

External 4 Relay Output Module

This module has 4 relay outputs. They function much like the internal outputs in regards to open, close, and toggle. Pulsing is different since more than one pulse can happen concurrently. Pulses are not queued.

RTD Module

The RTD module has 4 channels of thermocouples.

10 Volt Module

The 10 Volt module has 4 input channels and 2 output channels.

4 – 20 Milliamp Module

The 4 – 20 Milliamp module has 4 input channels and 2 output channels.

Temperature Probe

The temperature probe reads temperatures from -40 degrees Celsius to +140 degrees Celsius.

Registry

Interaction with the JNIOR registry system is a little more complex than interacting with the I/O. The registry system is analogous to the registry system on a Windows PC. The registry system is a set of Name Value pairs. You can write and read registry keys. You can also subscribe to registry keys allowing your application to get updates when registry key values are changed.

The registry is saved to a jnior.ini file on the JNIOR in flash memory. Flash memory is a non-volatile computer storage that can be electrically erased and reprogrammed. Flash is slower than RAM memory. It is even slower on the JNIOR. Once every couple of minutes the JNIOR checks to see if there were any changes to the registry system. If there were changes then the jnior.ini file is regenerated.

Registry keys that have a node entry that starts with a '$' are called temporary registry keys. Temporary registry keys not saved to the jnior.ini file and are not counted as a change to the registry system. This means that temporary registry keys do not cause the file to be regenerated. Examples of temporary registry keys are *$BootTime*, *$SerialNumber*, *$Model*, and *$Version*. Custom applications can define temporary registry keys. *Demo/$Test* and *$Demo/Test* are both temporary registry keys. The difference is that the every registry key under the $Demo folder will be a temporary registry key.

If you are programming in .NET then it is wise to get the Registry instance from the JNIOR session.

```
Registry registry = jnior.GetRegistryInstance();
```

You can then use the registry object over and over without calling `jnior.GetRegistryInstance()` every time.

Get Registry Key Handle

The JNIOR DLL is responsible for creating and maintaining the registry key structures. To return a registry key structure you must call the `GetRegistryKeyHandle()` function. This function returns a pointer to a registry key structure.

Get Registry Key Unique Id

To get the unique id assigned to a registry key call the `GetRegistryKeyId()` function. By passing in the handle of the session and the registry key pointer the function will return an integer value that is the unique id associated with the registry key. The unique id is used for communications between the DLL and the JNIOR.

Get Registry Key

To get the registry key of the registry key structure call the `GetRegistryKey()` function. By passing in the handle of the session, the registry key pointer and a pointer to a string that will hold the registry key the function will fill the string with the registry key.

Get Registry Key Value

To get the value of the registry key structure call the `GetRegistryKeyValue()` function. By passing in the handle of the session, the registry key pointer and a pointer to a string that will hold the get registry key value the function will fill the string with the registry key value.

Set Registry Key Value

In addition to getting the registry key value of a registry key structure, you can set the value. To set the value of the registry key structure by calling the S`etRegistryKeyValue()` function. By passing in the handle of the session, the registry key pointer and a pointer to a string that is the new value the set registry key value the function will fill the structure with the given value.

Read Registry Keys

To read a registry key use the `ReadRegistryKeys()` function. You can read multiple registry keys with one call to this function. Reading the registry keys can also be performed in an asynchronous method. To do so call the `ReadRegistryKeysAsync()` function. There must be a callback function defined.

C++

```
// create an array of 4 registry keys.
REGISTRY_KEY** regKeys = new REGISTRY_KEY*[4];

// get the REGISTRY_KEY pointer for the $serialnumber registry key
regKeys[0] = GetRegistryKeyHandle(handle, "$SerialNumber");
// get the REGISTRY_KEY pointer for the $model registry key
regKeys[1] = GetRegistryKeyHandle(handle, "$Model");
// get the REGISTRY_KEY pointer for the $version registry key
regKeys[2] = GetRegistryKeyHandle(handle, "$Version");
// get the REGISTRY_KEY pointer for the $boottime registry key
regKeys[3] = GetRegistryKeyHandle(handle, "$BootTime");

// now that we have obtained the registry key handles, lets read the values.
// these values will never change while the JNIOR is running. so we will only
// read them.
result = ReadRegistryKeys(handle, regKeys, 4, "");

for (int i = 0; i < 4; i++) {
	cout << regKeys[i]->key << " = " << regKeys[i]->value << endl;
}
```

C#

```
RegistryKey[] regKeys = new RegistryKey[4];

Registry registry = m_jnior.GetRegistryInstance();
// get the REGISTRY_KEY pointer for the $serialnumber registry key
regKeys[0] = registry.GetRegistryKey("$SerialNumber");
// get the REGISTRY_KEY pointer for the $model registry key
regKeys[1] = registry.GetRegistryKey("$Model");
// get the REGISTRY_KEY pointer for the $version registry key
regKeys[2] = registry.GetRegistryKey("$Version");
// get the REGISTRY_KEY pointer for the $boottime registry key
regKeys[3] = registry.GetRegistryKey("$BootTime");

registry.ReadRegistryKeys(regKeys, "");

foreach (RegistryKey regKey in regKeys)
{
	Console.WriteLine("   {0} = {1}", regKey.Key, regKey.Value);
}
```

Subscribe Registry Keys

Registry keys can be subscribed to. This means that your application will be alerted of any changes through the Registry Callback. Registry key subscriptions add additional overhead on the JNIOR. Use subscriptions only when needed. To subscribe to a single registry key or block of registry keys use the `SubscribeRegistryKeys()` function. Subscribing to the registry keys can also be performed in an asynchronous method. To do so call the `SubscribeRegistryKeysAsync()` function. There must be a callback function defined.

C++

```
void CALLBACK RegistryCallback(void* args)
{
	REGISTRY_CALLBACK_ARGS* registryArgs = (REGISTRY_CALLBACK_ARGS*)args;

	cout << registryArgs->count << " registry keys returned" << endl;

	REGISTRY_KEY** regKeys = registryArgs->keys;
	for (int i = 0; i < registryArgs->count; i++) {
		cout << regKeys[i]->key << " = " << regKeys[i]->value << endl;
	}
}

…

// assign the callback
AddRegistryCallback(handle, &RegistryCallback);

// create an array of 1 registry key.
REGISTRY_KEY** regKeys = new REGISTRY_KEY*[1];

// get the REGISTRY_KEY pointer for the $hourmeter registry key
regKeys[0] = GetRegistryKeyHandle(handle, "IO/Outputs/rout1/$HourMeter");

// the hour meter will change.  Subscribe to this key.
result = SubscribeRegistryKeysAsync(handle, regKeys, 1, "");
```

C#

```
private static void RegistryCallback(Jnior jnior, RegistryArgs args)
{
	Console.WriteLine("{0} registry keys returned", args.Count);
	foreach (RegistryKey regKey in args.Keys)
	{
		Console.WriteLine("   {0} = {1}", regKey.Key, regKey.Value);
	}
}

…

registry.AddRegistryCallback(new jnior_dll_calls.RegistryCallback(RegistryCallback));

// get the REGISTRY_KEY pointer for the $hourmeter registry key
RegistryKey hourMeterRegKey = registry.GetRegistryKey("IO/Outputs/rout1/$HourMeter");

// the hour meter will change.  Subscribe to this key.
int result = registry.SubscribeRegistryKeyAsync(hourMeterRegKey, "");
```

Write Registry Keys

You can write a registry key or block of registry keys to the JNIOR by using the `WriteRegistryKeys()` function. Registry keys that are written to with a blank value are removed from the registry system. Also registry keys containing a space must be enclosed in quotes. Writing to the registry can also be performed in an asynchronous method. To do so call the `WriteRegistryKeysAsync()` function.

C++

```
REGISTRY_KEY** regKeys = new REGISTRY_KEY*[1];

// get the REGISTRY_KEY pointer for the demo registry key
regKeys[0] = GetRegistryKeyHandle(handle, "demo/test");

SetRegistryKeyValue(handle, regKeys[0], "Look at me");

// write the keys
WriteRegistryKeys(handle, regKeys, 1);
```

C#

```
// writing to a registry key
RegistryKey demoKey = registry.GetRegistryKey("demo/test");
demoKey.Value = "Look at me in CSharp";

// write the keys
registry.WriteRegistryKey(demoKey);
```

List Registry

You can list the children of a registry node with the `ListRegistry()` function. This function will NOT block until the list is returned, instead the registry listing callback will be notified. A registry listing callback must be defined. The path string must NOT contain the trailing '/' character.

C++

```
void CALLBACK RegistryListingCallback(void* args)
{
	REGISTRY_LISTING_CALLBACK_ARGS* registryListingArgs =
	(REGISTRY_LISTING_CALLBACK_ARGS*)args;

	cout << registryListingArgs->count << " registry keys returned" << endl;

	for (int i = 0; i < registryListingArgs->count; i++) {
		cout << "   " << registryListingArgs->keyNames[i] << endl;
	}
}

…

// add a registry listing callback
AddRegistryListingCallback(handle, &RegistryListingCallback);

// get a registry node listing for the root node
ListRegistry(handle, "");
```

C#

```
private static void RegistryListingCallback(Jnior jnior, RegistryListingArgs args)
{
	Console.WriteLine("{0} registry keys returned", args.Count);
	foreach (string regKey in args.KeyNames)
	{
		Console.WriteLine("   {0}", regKey);
	}
}

…

// add a registry listing callback
registry.AddRegistryListingCallback(new
	jnior_dll_calls.RegistryListingCallback(RegistryListingCallback));

// get a registry node listing for the root node
registry.ListRegistryAsync("");
```

Miscellaneous

Custom Command

The JNIOR has the ability to run a custom application written in Java. For more information on witting applications in Java to run on the JNIOR please refer to the Embedded Applications Programming manual. Sometimes it might be necessary to communicate between a PC application and the application running on the JNIOR. You can use registry keys at times to facilitate some interaction but other times a different approach is needed. The `CustomCommand()` function allows you to send a 64K byte array between a PC application and a JNIOR application.

C++

```
// some code
```

C#

```
// some code
```

Send Macro Name

Sending a macro name is used in INTEGs cinema.jnior application. This is a specific function dedicated to communicating with the cinema.jnior application.

C++

```
// some code
```

C#

```
// some code
```

Example Index

Simple C++

This sample is the minimum implementation of using the JNIOR DLL. Written in C++ this sample shows you synchronous usage of the `CreateJniorSession()`, `Connect()`, `Login()` and `ToggleOutput()` functions.

Connection Callback

This sample shows the same steps as the Simple C++ sample but with a asynchronous implementation. Callbacks are defined for the Connection and Login operations.

Jnior Server

This sample demonstrates how to instruct the DLL to start its internal local server to listen for incoming JNIOR connections.

Jnior I/O Usage

This example shows how to interact with the JNIORs internal I/O.

Registry Usage

This sample shows the basic usage of the registry system on the JNIOR. Reading registry key, subscribing to registry keys and writing registry keys are implemented in this example. Subscribing to registry keys uses registry callbacks to be alerted when registry key changes are received.