Background for NodeBot Discussion
by John Wolf

In earlier sessions, I talked about the Arduino family of microcontrollers extensively. The supporting programming language was the C++. I also mentioned more complex microcontrollers that run from their own onboard operating systems, usually a derivative of Linux such as the Beaglebone Black and Raspberry-Pi boards. The Arduino is basically running a canned firmware program from flash memory. There's no underlying OS being used.

What makes a NodeBot distinctive from processors running C++ is a NodeBot is using JavaScript for the fundamental processing language. That may seem odd to you, since JavaScript is usually married to HTML webpages to provide a dynamic element to browser based client app or webpage when an user rolls over a button, pushes a button, rolls out of a particular place or used as a filter to check user entered data.

There are several other language choices for example Python that have become popular in robotics, but they all work like C++ in structure and operation. They all run a looping function to keep the program alive and suffer from program blocking if an interrupt comes in. JavaScript is an asynchronous language that is triggered to do something from a "listener" function that senses an event and acts on it immediately. The characteristic is not only appropriate for client-server apps, but is perfectly suited for robots. You want action to take place asynchronously and be event driven in a robot.

Classic programming runs in a polling interval that can be supported by an interrupt system that services events that generate the interrupt, but the issue is in the name. This interrupts the running program, blocking any further instructions until the interrupt is serviced. Not only that, but the process has to unload pipelined instructions, place the next instruction in a register past the current instruction, place all current referenced data being used into storage and push all of this information onto the stack to be undone by pulling all this information back into the mainstream after the interrupt is serviced and unblock the code to continue to run.

JavaScript just runs an anonymous function as part of the event handler code whenever an event is detected as described by the JavaScript script. In effect, a great number of events can happen all at the same time and all be running very quickly one after the other that appears to be serviced all at the same time. The Linux kernel doesn't actually run realtime operations (but can be modified to be a realtime processor). In general with the typical low bandwidth and very small payload package size of typical operations within the script, the instructions are run very close together. None of the classic overhead is involved. In fact, the main code is not blocked and is running amongst the event handler code instructions as well. All of these characteristics are exactly what robot designers need to smooth motions, react as quickly as possible, and not block time critical operations.

This all sounds good, but how do we take JavaScript from the client-server webpage environment to a robotic's environment. First a little history has to be laid out.

As web developers depended more and more on JavaScript, it was realized that the server-side could benefit from JavaScript running there as well. Then the client-server system could be more efficient and coherent. Node.js is a collection of JavaScript libraries that were created to manage the server-side operations. Then a new module (library) was created called node-serialport that would allow communication between processors outside the client-server realm. It didn't take long for frameworks were build around this code to connect normal node operations directly to specific microcontroller chips. This allowed JavaScript code to address the chips I/O suite from a command line or browser window. Combined with normal node.js functionality, you can use the Internet to gain client-side control over a distant socket

interconnection or obtain control a remote processor, gain data from a remote sensors and display it on a webpage with controls and data displays and fancy dashboard graphics.

This is a huge gain is operability, because from a smartphone or browser page a secure link can be establish and wirelessly connected to a remote device using very simple code provided by the node.js paradigm. This is much easier than developing unique class structures in C++ to do the same thing and avoids the latency timing and code blocking of classical techniques. But now we need to look at the framework to make this happen.

From the Disney robot movies called Short Circuit came a bot named Johnny-Five. This became the name of the framework for the use of the node-serialport module. If you go to the Johnny-Five.io website, you can see the current extensive connectivity available to robot builders.

One can learn Node.js quickly and it doesn't require and in-depth knowledge of programming legacy. You can take advantage of all the many node modules available and apply them to your robotics tasks immediately.

So this is what this session discussion is about. There are a lot of unmentioned pieces of code that have to be linked together to see the total impact of this approach. Not only is this approach basic to robotics, but is a great way to apply IoT to home automation, not to mention it has been field tested by the many users of Node.js and has a legacy of working well. Oh, did I mention - it's all open source!