**MeaowMeaow Technical**

June 25, 2015

Chris Stakutis

[chris.stakutis@gmail.com](mailto:chris.stakutis@gmail.com)

# Introduction

Talking to devices such as thermostats and medical sensors such as blood pressure cuffs and in particular robots is very cumbersome. The interfaces are either too low-level and require experts in microcode software (and a restricted environment) or too high level wrapped into some manufacturers application. Each device is different and has a different environment, different programming language or interface, and different abilities regarding CPU and memory speed as well as access to other world assets like the Internet.

The goal of MeaowMeaow is to bring the world’s vast resources of Internet “web” programmers to the up-and-coming Internet-of-Things (IoT). In short, this means shift the majority of the programming away from “on the device itself” to “something bigger and smarter nearby” and using all the familiar tools that web programmers embrace. As such, the horsepower climbs dramatically as does the ability to interact with numerous other data sources and available libraries (image and voice processing, mapping, weather and stocks, and so on).

Now, a MeaowMeaow-enabled solution can talk to potentially 100’s of devices and robots in parallel running difficult tasks and interfacing to many other repositories, while even updating its own behavior in real-time much like your applications update on your phone.

MeaowMeaow is a software-stack that attempts to simplify 3 problems:

* Writing code that is nearly 100% asynchronous in a nature (typical of programming robots and devices) using 100% javascript.
* Building full applications in either html (web pages) or node (server or embedded) or as part of some larger application.
* Talking to specific devices/robots using a small stack of libraries streamlining the process.

## Asynchronous Code

Let’s start with the first objective: Asynchronous code. Most consumer and business applications have the luxury that the majority of the software is written in a start-to-finish procedural manner. Of course activities are often started by some asynchronous event such as a mouse-button or receiving a message or a timer, but generally the work done within the task is serial. If a task is long-running, the operating system will break it up and allow other system tasks to occur.

Robot and device programming is dramatically different, although, not that much different than modern web programming. Modern web programming the author often deals with GUI events occurring sporadically or timer events maturing or communication messages arriving. More recently, web programmers now routinely call functions that complete in steps asynchronously (such as sending a message or writing to a file or gathering data from the web). This has led to an acceptance of asynchronous programming in web applications.

Controlling robots or devices is very similar. Suppose you wanted to move a robot forward 10 steps (unless stopped by a wall along the way), then turn right, then scan for heat/IR, then beep? There are perhaps 20 steps in doing all those activities and each one completes asynchronously and with or without errors. This can be very challenging to create readable software.

Various attempts to simplify such programming have existed over the years. Typically the solutions are some sort of configuration-language or scripting language where a master program will “step” through the execution. MeaowMeaow did not want to create “yet another language” or scripting environment. Instead, it combined Javascript JSON structures with Javascript to allow for the best of both worlds (structures allow for easy listing of activities and javascript is a superb scripting language all on its own).

The heart of a MeaowMeaow system is what is called the “Executioner”. This class will accept a JSON structure of steps and javascript code and steps through the execution understanding the asynchronous nature and in parallel to the rest of your program running (perhaps even running other Executioners). The “Executioner” chapter will describe this in much more detail. For now, think of it as a simple subroutine that executes a complicated set of tasks that you define and can change on the fly.

## Application Flexibility

MeaowMeaow is 100% Javascript code and as such allows you a wide variety of deployment paths. Often an application will have some sort of GUI and being able to use a web-based HTML environment is a welcome advancement. Using technology-wrappers such as Cordova/Phonegap your web application will run as true native application on all phones as well if you like. If your application is embedded (perhaps part of the brain of a dishwasher or pet robot or a vehicle), it can run as a pure node.js application.

There are some differences in each model, mostly around how the application is built and deployed, and access to hardware elements and security. See the following table:

|  |  |  |
| --- | --- | --- |
| Approach | Pro’s | Con’s |
| Regular html web page running on a client (no http server) | * Very simple starting point. * Works ANYWHERE. * Full HTML Gui capabilities | * Typically can’t access local devices or networks and thus not your robot or device but and easy to get started. |
| Chrome Application | * As above * Can access internet * Can access devices via bluetooth or other networks | * Runs only where Chrome apps can run (but that’s about everywhere). * Needs Chrome-provided drivers for networks like bluetooth (but they exist and work). |
| Mobile Application via Phonegap | * Allows your phone to be the master control. * Same UI as above. | * As above, needs drivers from low level networks like blue (but they exist). |
| Node.js application | * Super fast * Trivial to use/deploy | * No UI unless you create one * Must run on a node-capable system (but most are...even embedded boards) |

## 

## Software Stack / Libraries

MeaowMeaow is not overly complex. In short, examine this application hierarchy:

|  |
| --- |
| 1-Your full GUI application doing lots of things |
| 2-MeaowMeaow “Executioner” |
| 3-Logical Drivers for each device or robot |
| 4-Communication abstraction to the devices |

At the top most layer, Layer-1, you are in complete control. You build your application, you design the GUI (HTML/CSS/Javascript based), you process any interaction controls, you decide what Internet services to talk to, you decide on a graphics to update and pages to reveal, and so on.

You then have MeaowMeaow “programs” that you have either written or bought/borrowed from someone or modified. Maybe 10’s of them. You activate (“Execute”) those as and when you see fit (perhaps several at once, perhaps ones activating others, and so forth) and even update them when you wish.

The programs you write can do anything you desire, not simply interact with devices. They have free access to call any javascript that you have or is available from a library etc. The beauty of MeaowMeaow programs is that they are easy to string-together long-running and asynchronous tasks while your main program marches along.

In general, however, your MeaowMeaow programs will talk to MeaowMeaow-implemented device drivers. MeaowMeaow device drivers are agnostic of the network that will actually carry the transmissions (described next). Rather, they focus on providing a simple interface to the key activities of a device (e.g. start motor, check a sensor) without stopping your application or the MeaowMeaow program while activities are in progress (thus, you can, say, start 2 motors and turn on a LED at the same time, and then wait for the motors to stop later).

As just mentioned, the MeaowMeaow-implemented device drivers will likely talk to the device by a MeaowMeaow supplied communication driver. This is not necessary and of course you can use your own, but a goal of MeaowMeaow is to establish some standards which then allow for easier mixing and matching parts. Most of the devices support a “serial” type of interface where a stream of bytes/characters is written and a stream can be read if/when available. Often, these protocols are truly line-oriented and will work with a terminal emulator (that is, they terminate with a CRLF sequence). MeaowMeaow supplies a serial driver that works in a Chrome Application environment build. MeaowMeaow also (will) supply the same serial driver for a phonegap bluetooth serial. For node.js one can easily built on top of the bluetooth plugin as well. Drivers can also be built for near-field or ethernet technology. The goal of a properly configured software stack is that the upper layers (the robot-specific drivers) should not care about the transport below them.

## More on Communications

The ultimate goal of MeaowMeaow is rapidly support many different and complicated robots and devices in an easy-to-program pseudo language. Sometimes MeaowMeaow will build the lower level communication interfaces to the devices and sometimes even build the code that runs on the devices to talk more easily to MeaowMeaow.

Let’s go thru some uses cases to better explain. Sparki is a consumer robot. It has many motors and sensors and capabilities. It even has its own programming language and environment which you download programs onto. In fact, Sparki is an Arduino board underneath and uses the Arduino tools and languages. While the language is “C” and feels rich, there is only a tiny amount of memory and the programming model is single-threaded and very limited. Only the simplest of programs can be created and downloaded.

Enter MeaowMeaow. MeaowMeaow wants users to be able to write huge applications that are very powerful and can talk to the internet and other devices and disk/files etc. So, MeaowMeaow created a tiny program for Sparki that essentially makes the robot a “remote procedure call”. A very simple serial protocol sends commands to the Sparki and she responds with answers. Special care is taken to make the protocol tiny because Sparki does not have much memory or compute power for processing.

So, MeaowMeaow provides two pieces here: The actual code that needs to be downloaded to the robot (and stays persistent) that responds to this language, and the MeaowMeaow device driver that talks that language and coordinates the conversations properly. Then, an upper layer, a MeaowMeaow program, can string-together commands to do powerful tasks. In fact, a few additional functions are provided that simplify work for users, such as analyzing an ultra-sound scan of an area for objects. Other parties (other people, students, etc) might also supply other javascript library functions built on top of the Sparki Driver that do more creative work and expose those functions to the MeaowMeaow programming language.

Let’s now contrast this to devices that do NOT allow a code-load. In the medical device world, there is a handheld bluetooth-enabled PEF meter (breathing meter) called an AM1. There is a defined serial interface to the meter and that protocol can not be changed. Thankfully, it exposes most of the capabilities in a reasonable way. MeaowMeaow created a MeaowMeaow driver for it that understands that protocol and exposes the functions/data/sensors to the MeaowMeaow programming environment. Now, a grand program can be built that moves the Sparki robot around, picks up the PEF meter, hands it to a person, waits for reading from the PEF meter, and then returns home while updating a cloud-service with the new data en route!

# MeaowMeaow Programming Language

The goal of the MeaowMeaow programming language is primarily to make executing a long series of asynchronous tasks/steps easy. It is NOT a goal to completely introduce a new programming language. The user \*always\* has the ability to write their own logic (or pieces of logic when necessary) using all the glory of javascript.

That said, there are a number of very common “needs” that it make sense for the MeaowMeaow language to provide such things as: Branching, Subroutines, and Loops. That’s about it -- anymore and it would be simply creating a new language and the world does not need a new programming language. And to be clear, MeaowMeaow is not a language per-se, but rather a data structure that describes steps and actions (thus, a bit more script-like which makes it much easier for developers).

Let’s first look at the terminology in the MeaowMeaow language:

* Program
* Steps
* Commands
* Repeat Commands
* Branching

Some Programs can be quite tiny, perhaps the logic necessary at start-up, or shut-down, or to play a short musical tune, or to grab the current object. These can be thought of as “Subroutines” and can be “called” by higher order programs.

A Program has Steps. Sometimes just one step, sometimes dozens. Within a step, “Commands” are processed. Again, perhaps there is just one command, or many. Each command can either advance to the next command in the list, or, change the step (that is, go-to another step). Going to another step is quite common in robot programming -- it is more of a “state change”. It is \*not\* a subroutine call and will not resume the current command list.

It is very common for a Step to enter into a loop-mode where it is waiting for something specific to occur or end.

Let’s look a simple example. Suppose I want a program to start by getting the robot into a known settled state, and then monitor the z-axis sensor and when it changes (by being picked up) it screams a note and then finishes.

In the MeaowMeaow vernacular, we’d create a simple program with a couple of steps. The first step will have a series of simple commands to initialize the robot (say, turn the head straight, turn on an LED), and do this just once. Then, in that step, it advances to the Loop. Here, it continually reads the z-axis and maybe the clock. If the z-axis changes, it changes to a new step called “scream”. The “scream” step turns on the buzzer and does nothing else (not other commands) so the entire program-chain now ends.

## Language Structure

A “Program” is simply a JSON structure that has two attributes: name and steps. Name is a simple string, and steps is an array (of “step” structures). Let’s look a simple example:

var MyProgram = {

name: “MyFirstProgram”,

steps: [ … ]

};

There can be any number of “steps” and they are not directly connected to each other but can be referenced by steps. Their order in the program is not significant; think of the program as more or less a library of steps. The program will start, however, with the first step (future rev this will be specifiable). That step might end, or it might chain to a step 10-deep in the list or whatever.

Let’s now look at a “Step”. It too is just another JSON structure which also has a “name” (required) and optionally a list of commands and optionally a list of loop/repeated commands. For example:

{

name: “Step1”,

commands: [ … ],

repeat: [...]

}

If “commands” is present, each item in the array will be executed, one after the other, UNLESS the command has an branch target in it (more on this later). When all the commands are executed (or, if no “commands” present) the “repeat” item will be examined. If “repeat” is present, it will run essentially the same way as “commands” runs, except it will loop automatically. The loop can only end if a given command branches to another step (or the program is aborted).

This style of a step is quite useful. In robots and device control, very often you want to perform a few functions at the start, and just once. And then you enter the bulk of the work which is typically a loop. Keep in mind, any command can be subroutine call (which is essentially another whole program).

The transfer between steps is best thought of as a “state transition”. Let’s say we’re monitoring a door. If it is now open, we want to go into the “Door Is Open” state and do whatever is necessary there. If it is closed, we want to go into the “Door is Closed” state. You can have as many steps (“states”) as you desire.

You can build-up massively complex programs with this approach yet keep each individual task or group of tasks small. Also, Multiple programs can be in process at the same time (perhaps mid-way through a program you need to start another program to run in parallel for a period).

Let’s now take a look at a “command”. A “command” is also a structure, but a bit more colorful. It has these attributes:

* “func” (a javascript function -- either one you created or supplied or a system one)
* “program” (optional if you want to “call” another MeaowMeaow program)
* “ifTrue” (optional; if the func returns true, then GO TO the named step)
* “ifFalse” as above

Plus, you can add as many other attributes as you wish because your “func” can access them (like parameters). Let’s look at some simple examples:

{func: MyBeep}

This will simply call the “MyBeep()” function that must be defined (supplied by you or otherwise) and then the next command will be executed.

{func: MyBeep pitch:1000}

In this example, the “MyBeep” function is smart and checks to see if the “pitch” attribute is defined and if so uses that value for the pitch (details on how this works explained a bit later).

{func: MyBeep ifTrue:”Step2”}

Here, if MyBeep() returns a boolean value and the value is true, the program control will now start at “Step2” (state transition).

{func: function() {console.log(“Hi there.”);}}

You can define anonymous functions right in the JSON structure. They are executed exactly like any other function. This is often handy for quick/simple checks when you dont want to bother creating a full function.

{program: waitTilDone}

If “func” is not defined in your structure, the system will check for “program”. If “program” is defined, then that program is called as a subroutine (that is, the current program pauses until the subroutine program completes).

{func: sparki, cmd:”R45”}

This calls the MeaowMeaow supplied Sparki driver’s function called “sparki”. The Sparki robot has a variety of commands it can execute and they are all abbreviated for simplicity. The sparki() function looks for a “cmd” attribute in the command and will send that to the robot. In this particular case, “R45” means “Rotate head to the 45 degree angle”. Talking to devices gets interesting. First, it takes some “time” to send a command and to believe it was received. The MeaowMeaow program will not advance the program until that occurs. That is different, however, from the command actually fulfilling the request (to turn 45 degrees, which could take many milliseconds). The creator of this program may choose to do other work in the next commands or may choose to wait until the activity is completed (to do that, it will likely execute a subroutine call that loops checking on motor status or similar).

## More details on how commands run

You saw from the previous pages that a MeaowMeaow “command” typically has a “func” function that it invokes. MeaowMeaow passes three parameters to all functions which help them communicate with the program. The parameters are known as “Environment”, “Step”, and “Command” (all JSON structures).

Most typically, a function will look at “Command” which is the exact JSON command structure in your program. This is how it can see if there are other attributes (like parameters) such as “cmd” or “pitch” in the examples above. Your function could optionally “set” an attribute in the command which will be persistent (and thus available the next time that command executes) and this might help in keeping some sort of state or variable.

The “Step” structure is the exact JSON structure of the currently running step. The command may want to know its name or may want to store attributes at the Step level for other commands in that step to use. Also, MeaowMeaow will set a “stepStartTime” attribute in the step each time the step is started and a command may want make use of that for timing something.

The “Environment” structure is unique per running instance of a program (that is, if the program is running twice in parallel, the Environment struct is unique to each). This is a more common place for commands to store data that other commands may need and that is unique to this running instance of the program. Importantly, MeaowMeaow device drivers will store incoming data (such as sensor readings) in this structure (under “devices” and then the device name). Look at the following fragment:

**repeat**: [

{**func**: *sparki*, **cmd**: **"SAZ"**}, *// Wait until we've stopped moving*

{func: **function** (e,s,c) { **if** (e.**devices**.**sparki**.**sensors**.AZ > -10 && e.**devices**.**sparki**.**sensors**.AZ < -9) **return true**}, **ifTrue**:**"BeepOff"**}

]

In this repeat loop, we are asking Sparki to sample the Z axis accelerometer. In our anonymous function we are checking the value recorded and if the device seems “still” we chain (state transfer) to the “BeepOff” state to turn off the alarm.

## Controlling Asynchronous Functions

Any “func” called has an obligation to tell new MeaowMeaow executioner if there is going to be a delay. Ordinary and simple commands complete right away and MeaowMeaow will essentially advance to the next command on the next timer tick. But device and robot commands typically do \*not\*. They will have their own asynchronous completion (say, when a reply from the device comes back). In such cases, it is important to tell MeaowMeaow to NOT advance automatically. To do so, the func must set the Environment attribute “advanceOnTimer” to false.

When set to false, MeaowMeaow essentially pauses execution. When the device code receives the asynchronous event from the device, it is now appropriate for MeaowMeaow to resume, and the device code must call the Environment.advanceProgram() to allow MeaowMeaow to resume its scheduling loop.

## Some handy built-in functions

MeaowMeaow supplies a few handy functions right now, and many more forth coming. They are:

“MM.timedOut”: Looks for either a “stepMS” or “MS” attribute. If “stepMS” is set, this will return true if the current step has exceeded the setting. If “MS” is set, this will return true if the program has exceeded the setting.

“MM.sleepMS”: This looks for “ms” attribute set and will essentially pause for that amount of time. It does so by telling MeaowMeaow to not advance to the next step for that amount of time. This way, everything else in the overall system is still functioning and not blocked nor doing a tight-loop polling of time.

“MM.alwaysTrue”: Always returns true. This is handy to have a command simply change the step (state transfer) which is often desired at the end of a list of commands.

## A more complete example

Here is a more complete example involving the Sparki robot and scanning for an object and then fetching it. You can find all the comple Sparki examples in the installation (under MM/Robots/Sparki/Examples).

**var *ProgramDS*** = {

**name**: **"Depth Search"**,

**steps**: [

{

**name**: **"StartScan"**,

**commands**: [

{**func**: *sparki*, **cmd**: **"R0"**}, *// Put head straight*

{**func**: *sparki*, **cmd**: **"GO1"**}, *// Gripper-Open 1 cm*

{**func**: *sparki*, **cmd**: **"T1000|500"**},*// Beep at 1000hz for 500ms*

{**func**: ***sleepMS***, **ms**:500} *// Sleep a little*

],

**repeat**: [

{**func**: *sparki*, **cmd**: **"D-50|50|2"**}, *// Invoke a distance-scan from -50 to 50 degrees*

{**func**: *rangePlot*, **canvas**:**"myCanvas"**, **range**:55, **label**:**true**, **skipAbove**:55}, *// Plot it on the screen*

{**func**: *findTarget*, **closerThan**:40, **ifTrue**: **"ApproachTarget"**}, *// Output in e->targetAngle, e->targetDistance*

*// If we didn't find anything, beep, and keep trying! Loop!*

{**func**: *sparki*, **cmd**: **"T100|1000"**},

{**func**: ***sleepMS***, **ms**:5000}

]

},

{

**name**: **"ApproachTarget"**,

**commands**: [

*// MR==Turn Right, MF=Move Forward, Move Backward, Turn Left, etc...*

{func: **function** (e,s,c) {c.**cmd**=**"MR"**+ (e.**targetAngle** - 2); *sparki*(e,s,c);}},

{**program**: ***waitTilDone***},

{func: **function** (e,s,c) {c.**cmd**=**"MF"**+ (e.**targetDistance** -4); *sparki*(e,s,c);}},

{**program**: ***waitTilDone***},

{**func**: ***sleepMS***, **ms**:500},

{**func**: *sparki*, **cmd**: **"GC3"**}, *// Grab it!!*

{**program**: ***waitTilDone***}, *// Now lets return home*

{func: **function** (e,s,c) {c.**cmd**=**"MB"**+ (e.**targetDistance** -4); *sparki*(e,s,c);}},

{**program**: ***waitTilDone***},

{func: **function** (e,s,c) {c.**cmd**=**"ML"**+ (e.**targetAngle** - 2); *sparki*(e,s,c);}},

{**program**: ***waitTilDone***},

]

},

]

};