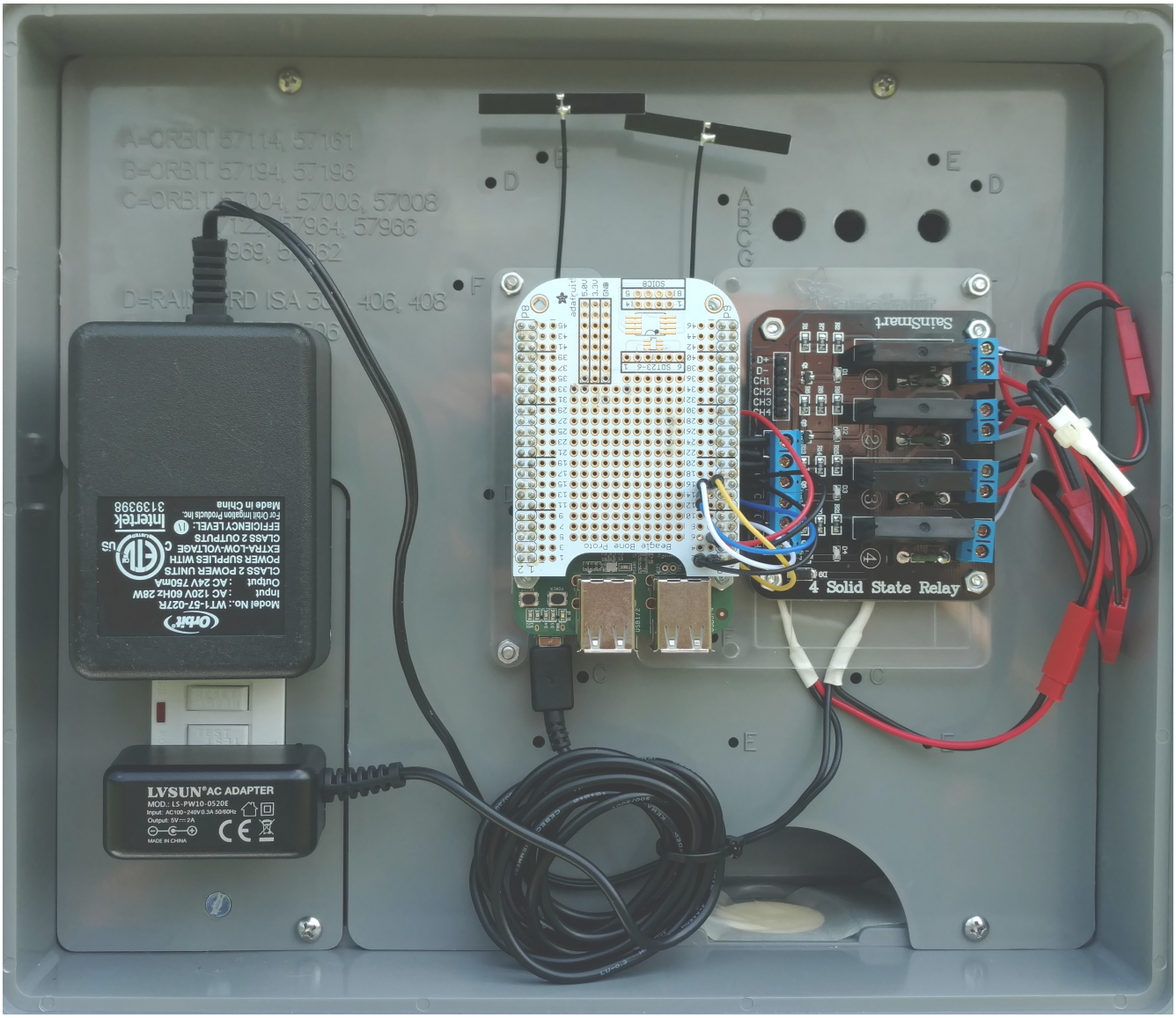

Irrigation Controller Using Beaglebone Green Wireless, Node.js, and Ecmascript 6

Gregory Raven

June 12, 2017

Irrigation Controller Using Beaglebone Green Wireless, Node.js, and Ecmascript 6

Contents

List of Figures

Chapter 1

Introduction

This is the documentation for an embedded GNU/Linux project using the Beaglebone Green Wireless (BBGW) development board. The project repository is located here:

https://github.com/Greg-R/irrigate-control

A listing of parts and other resources is found in the Resources chapter.

1.1 Project Goals

I've published two other projects on Hackster.io. These other projects are interesting, and good learning tools, but they do not do anything useful in the real world. After a couple of years of "going up the learning curve" on embedded device development, it was time to create something practical!

This project performs the simple home automation task of controlling a lawn irrigation system. For many years, I had been controlling the irrigation system manually by flipping a circuit breaker to turn the pump motor on and off. The old timing unit had failed years ago, and I had never replaced it. I wanted to be able to control the system via a web browser without having to leave my home office desk.

Although in principle the automation is simple, the underlying technology is complex! There was a significant investment in time learning the technology required to implement the project.

1.2 Technologies

The development board chosen is the Beagle Bone Green Wireless (BBGW). The board will be mounted remotely with access to power only (no wired ethernet is possible at that location).

The Debian-based GNU/Linux distribution used on the BBGW can be downloaded from this page:

http://beagleboard.org/latest-images

The "IOT" (non-GUI) image was chosen, as this provides most of the software features required to get the project up and running. As is, the board is configured as an "access point". This was changed to a conventional wireless LAN interface.

Node.js version v8.1.0 was used. This is a much later version than what is included in the image. A section on upgrading Node is included.

The project's Javascript code uses several "Ecmascript 6" constructs. In my opinion this release of Javascript is a significant improvement in the language. Some of the strange quirks of Javascript are eliminated!

Two-way communication between the web browser and the BBGW was done with "WebSockets". The client side WebSocket is built into the browser. Using the latest updates in Ubuntu 16.04, both Firefox and Chromium browsers include this capability. The Chrome browser of an Android phone was also found to work.

Here is a good reference on the client (browser) side WebSockets:

https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

The server side uses a Node.js package called "ws":

https://github.com/websockets/ws

Since this is a "real world" project, there has to be an interface with real actuators. This small "solid-state-relay" board proved to be ideal for taking the GPIO outputs of the BBGW and doing something real:

https://www.amazon.com/gp/product/B00ZZVQR5Q/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1

This board has four individual relays. This project uses only three.

Note that this type of relay can switch AC power only. This was ideal for this project, as the irrigation system controls are powered by 24VAC.

The other common components of the irrigation system are listed in the reference section at the end of this document.

"Universal IO" was used to set the pin multiplexer to GPIO mode:

https://github.com/cdsteinkuehler/beaglebone-universal-io

The irrigation system control components were manufactured by Orbit. These types of solenoid valves and relays seem to be standardized to 24VAC power, so other manufacturers products will work as well.

Chapter 2

System Diagrams

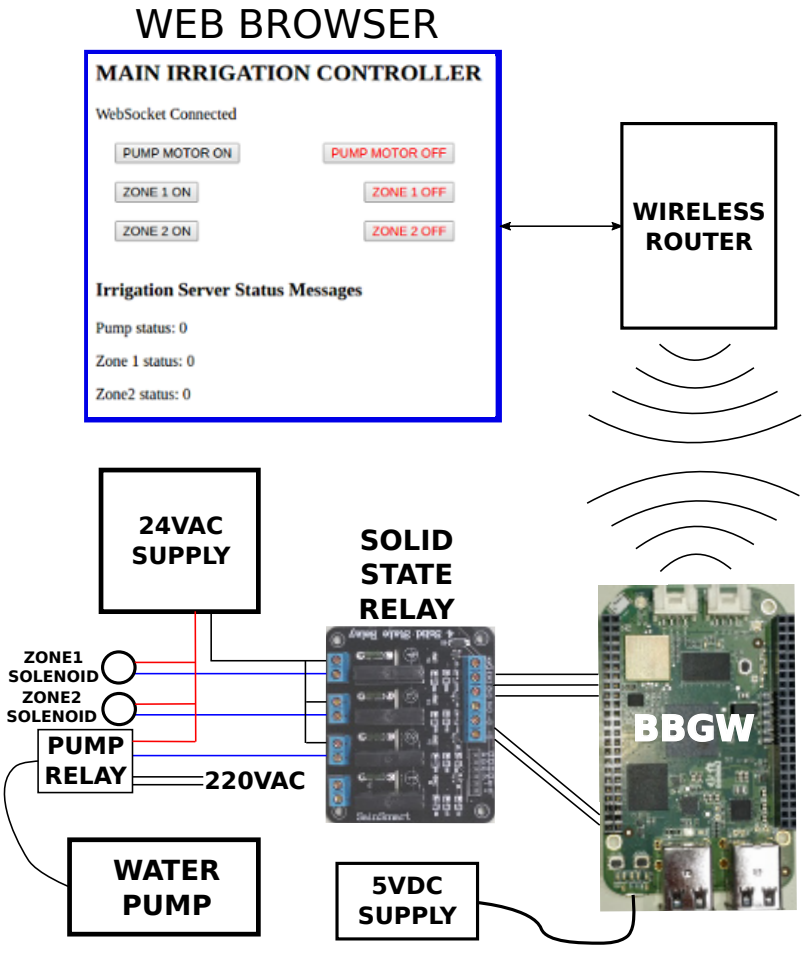

Figure 2.1: Beagle Bone Green Wireless Irrigation Control System

The above diagram shows the main components of the system. A reference section is included which has a complete parts list.

2.1 GNU/Linux Operating System on BBGW Host ARM Processor

The command uname -a on the BBGW used to develop this project reports this:

```
Linux beaglebone 4.4.48-ti-r88 #1 SMP Sun Feb 12 01:06:00 UTC 2017 armv7l
GNU/Linux
```

The hardware used in this project only requires GPIOs. So the actual distribution and kernel used should not be critical as long as it supports the WIFI of the BBGW and upgrading Node is possible.

Chapter 3

Ecmascript 6

This page shows which Ecmascript 6 features are implemented in major versions of Node:

http://node.green/

Javascript has become a feature rich and large language! Some of the ES6 constructs used in this project:

- class
- Map
- arrow function
- Proxy
- const and let

A good reference for ES6 is at the Mozilla Developer Network:

https://developer.mozilla.org/en-US/docs/Web/JavaScript

A quick summary of the ES6 constructs follows.

3.0.1 class

The new "class" keyword does not provide new functionality. What is does is allow a Javascript class object to be written in a more traditional object-oriented style. This does make things easier for a person used to other language's syntax for defining classes.

3.0.2 Map

The "Map" is a new data structure object which is similar to what is called a "hash", dictionary, or associative array. Prior to ES6, the Javascript object was used functionally as a Map, however, this was kind of a hack.

In this project, the Map object is used in the pumpActuator object.

3.0.3 Arrow Function

“Arrow Functions” are a simplification of the syntax used when defining a function. However, they are also important in determining the the scope of “this” within the function. The arrow functions capture the “this” of the enclosing scope. This was found to be convenient in the design of the pumpActuator class.

```
this.pumpMap = new Map([['pumpmotor', 0],
                        ['zone1', 0],
                        ['zone2', 0]]);
```

The “pumpMap” is a simple data structure to store the state of the pumpmotor and the two zone actuators.

3.0.4 Proxy and Object.assign

Of the above ES6 constructs, the only one requiring detailed explanation is the “Proxy”. The construct is used to implement the software “Observer” pattern.

The instantiation of the “pumpMapProxy” is done in the constructor of the pumpActuator class:

```
this.pumpMapProxy = new Proxy(this.pumpMap, this.pumpObserver());
```

The Proxy’s constructor takes two arguments, which in this case is this.pumpMap and this.pumpObserver(). The second argument requires some explanation.

```
pumpObserver() {
    return {
        set: (target, property, value, receiver) => {
            console.log(‘Setting ${property} to ${value}.‘);
            this.pumpControl(property, value);
            target[property] = value;
            return true;
        }
    };
}
```

The above class method is a little bizarre. This method merely returns a Javascript object. The object in this case has a single key “set” and the value is an arrow function with four parameters: target, property, value, and receiver.

The Proxy creates a sort of “watcher” or “observer” of pumpMap. The proxy intercepts changes written to or read from the target object (the first parameter of the Proxy constructor).

Using the key “set” causes the function (the set key’s value) to be executed when the value is written to. Note that the function has access to the intercepted property and value, and these

are used in call the function “this.pumpControl”. This does the physical setting of the GPIOs. The data structure is also updated (target[property]=value) and “true” is returned to indicate a successful set.

The proxy does a sort of “intercept” of writes to the object and then can perform custom actions based on the the function assigned to “set”. Similar functionality for reads can be done with the “get” key. The object can contain both and custom read and write functions can be used. This is very powerful!

Functionally what the Proxy does is intercept the write to the data structure which stores the state of the pumpmotor and the zone solenoids. The intercept runs the “set” function and changes the physical state of the GPIOs.

The write to the data structure is done by the server in file websocketserver.js:

```
Object.assign(pumpObject.pumpHashProxy, controlObject);
```

The “controlObject” in this case is an incoming WebSocket message (using JSON notation) from the browser which looks like this example:

```
{"pumpmotor":0}
```

The write to the data structure in the pumpActuator object is done by the server in file websocketserver.js:

```
Object.assign(pumpObject.pumpMapProxy, controlObject);
```

The “Object.assign” is a shortcut which simply overwrites the value in pumpMapProxy with the value from controlObject. The Proxy intercepts this write, and then executes the custom set function.

The Proxy allows a custom behavior to be executed when a data structure is written to or read from. In this particular project with only three controls it does not standout, however, this sort of “Observer” pattern is very scalable and could be very advantageous in a much larger and more complex system.

3.0.5 const and let

const creates a read-only reference to a block-scoped value. let is also block-scoped, but it is a variable.

let and const solve the crazy problem of hard to understand scope and “hoisting” of the var type variables of previous versions of Javascript. var was not used anywhere in this project.

const and let are a huge improvement to Javascript!

3.1 Upgrading the BBGW to Node Version 8

The Beaglebone image used in this project is built with Node version 4.8.0. This will need to upgrade to the latest Node.

Follow these instructions at the Node.js web site:

https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distribution

A couple of command lines is all it takes:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
```

After the upgrade completes, run this command:

```
node -v
```

The shell should respond with something like

```
v8.1.0
```

which was the current Node version at the time of this writing.

Chapter 4

GPIO Control with sysfs Virtual File System

The “sysfs” virtual file system is the core functionality which allows the “General Purpose Input Output” (GPIO) to be controlled from user space by Node.js Javascript. Three GPIOs are used in output mode, and these outputs control a solid-state relay board.

Note that the BBGW must be correctly configured for GPIO output mode on the three control pins used to control the irrigation devices.

A highly recommended resource which covers this topic is “Exploring Beaglebone” by Derek Molloy [1]. This book has an excellent chapter on controlling the GPIOs from Linux user-space.

The method of configuring the header pins to GPIO is covered in the chapter “Universal IO”. GPIO configuration must be complete before any of the commands shown below will function properly.

“POSIX” type operating systems, which includes Linux, are “file based”. That means the interface to everything is via writing to or reading from a file. In this case, the GPIO’s state is changed by writing a 0 or 1 to the appropriate file. Here is an example:

```
echo 1 > /sys/class/gpio/gpio50/value
```

The above changes the state of header pin P9.14 to “high” or an output of 3.3 volts. Echoing 0 changes the output to 0.0 volts. It’s that simple!

4.0.1 Controlling the GPIOs using Javascript

The command shown above is typed and executed in a bash shell. How is this done from Javascript? A module from Node.js is used to accomplish this:

https://nodejs.org/api/child_process.html

For example, the bash command shown above would be executed as follows in Javascript:

```
const exec = require('child_process').exec;
```

[1] http://exploringbeaglebone.com/

```
exec('echo 1 > /sys/class/gpio/gpio50/value');
```

This is the simplest possible usage; an optional callback function as a second option is possible. The callback option is used in the in the "pumpActuator" class, and the callback is used to emit an event from the pumpActuator Object. This event is subscribed to by the WebSocket server, and when the event fires it sends a message to the web page controller to indicate that the control function has changed state. The web page is updated to indicate the new state.

The class method looks like this:

```
pumpControl(pumpgpio, command) {
    const exec = require('child_process').exec;
    exec(`echo ${command} > ${this.pumpGpioMap.get(pumpgpio)}`, (error,
    stdout, stderr) => {
        // If error, do not update the status of the controls.
        if (error) {
            console.error(`exec error: ${error}`);
            return;
        } else {
            console.log(`Status message emitted from ledActuator:
            ${pumpgpio} is set to ${command}.`);
            //  Send a JSON object with the value being an array.
            this.emit('statusmessage', `["${pumpgpio}",${command}]`);
        }
    });
}
```

Chapter 5

An HTML5 Controller

The controller GUI is an HTML5 web page. The project was developed with the Chromium browser in Ubuntu 16.04.

There was no attempt to fix problems with cross-browser compatibility issues. Plain HTML5 and CSS was used throughout. The HTML was manipulated directly via the "Document Object Model" (DOM) technology in the web browser. This is a bare-bones interface with no fancy features.

Manual control buttons for the zone solenoids and the pump motor are at the top. The user can enter a schedule date and start and stop times. Clicking the "Schedule" button sends the requested irrigation schedule to the server. The server responds with a message which updates the displayed schedule. No rigorous checking of inputs is done.

The current date and time are shown in the last line of the controller page.

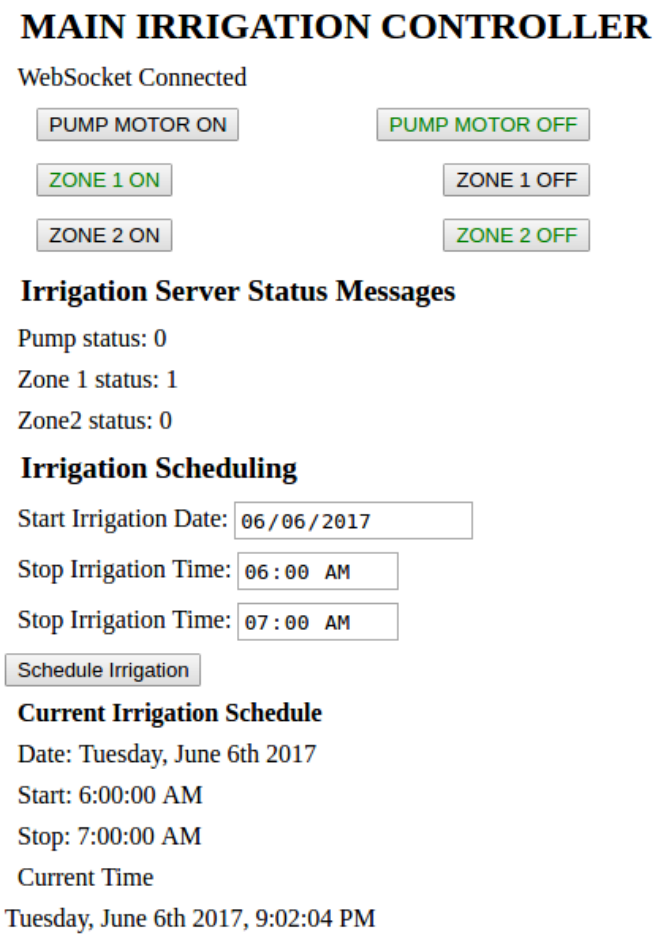

Figure 5.1: The Irrigation Controller as seen with a Chromium Browser

Chapter 6

Irrigation Scheduler Class

The Scheduler class is responsible for activating the irrigation system at the time specified by the user. The user enters the start and stop timing using the web browser controller. The watering times are equally split between the two zones.

A timing function such as this might seem simple. However, it proved challenging to implement with simple code.

Two excellent NPM packages came to the rescue: node-cron and moment.

https://www.npmjs.com/package/moment

The Node-cron package manager implements a functionality similar to a “cron job” in a POSIX operating system. The function provided is a time-delayed task set by the user. In this project, four jobs are required, and thus four node-cron tasks are created by a method of the Scheduler class.

The Moment package provides a robust data and time module which is more flexible than the native Javascript Date object. The module’s mode of operation is a little unusual, however, the documentation is excellent and the user should have no problem using the extensive feature set.

https://www.npmjs.com/package/node-cron

Figure 6.1: The Web Browser Irrigation Scheduler

Chapter 7

JSON Messaging with WebSocket

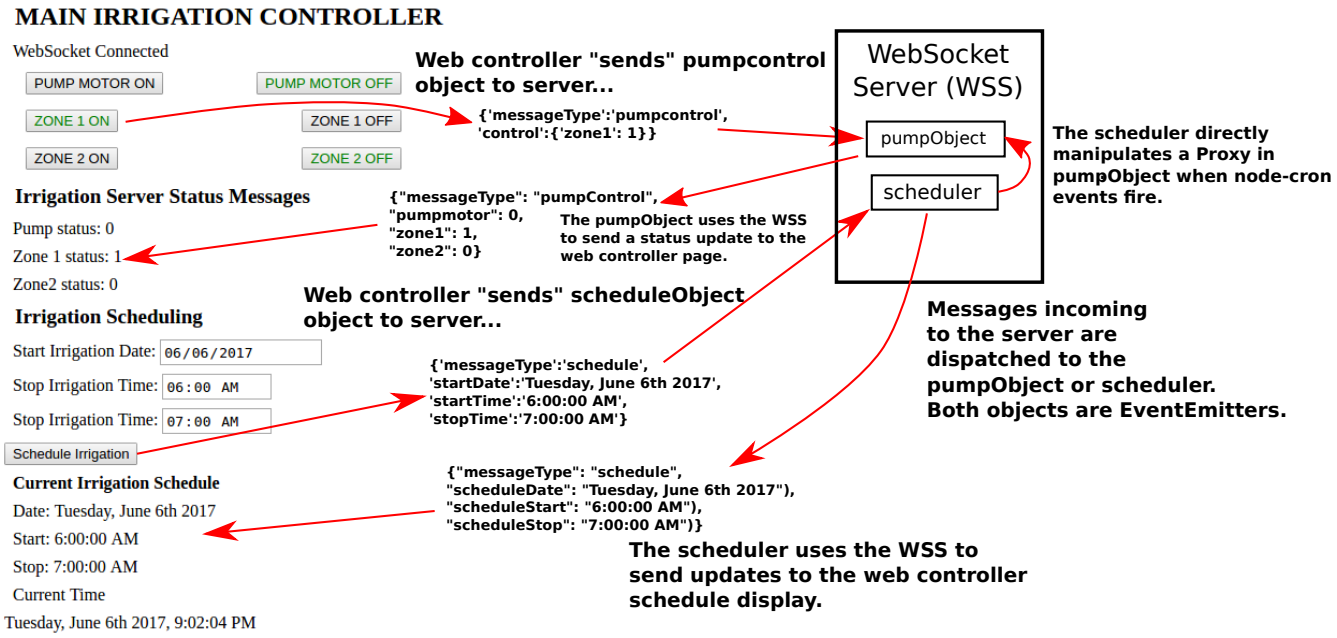

Figure 7.1: JSON Objects are passed between the Web Page Controller and the Server

The above diagram shows the flow of data between the web browser controller and the BBGW server.

"Javascript Object Notation" (JSON) is used the format used for the messages. These "Objects" are used as associative arrays would be used in other programming languages. In fact, ES6 includes a formal associative array called Map. However, JSON is a de-facto standard for this sort of simple message passing, and the JSON related tools are easy to use.

To send a JSON Object with WebSocket, it is first necessary to "stringify" the object. This is done with the JSON.stringify() method. On the receiving end, the JSON.parse() method is used to translate back to a Javascript Object. After that, the Object is used with the normal key/value syntax of Javascript.

7.1 Example JSON Messages

Thanks to WebSocket being bidirectional, it is easy to pass messages in both directions between the web browser controller and the server. There are a total of four message types; two for browser to server and two for server to browser.

7.1.1 Browser to Server Hardware Control Message

This message indicates the device to be controlled and the desired state:

```
{'messageType':'pumpControl', 'control':{'zone1':1}}
```

The key "messageType" indicates to the server that this message should be routed to the pumpActuator Object. The "control" key's value is another JSON object, and this Object has as its key the device to be controlled, and the value is the desired state.

A little bit of ES6 Javascript trickery is involved to use this piece of incoming data:

```
Object.assign(pumpObject.pumpMapProxy, dataObject.control);
```

The pumpActuator object has a data structure containing the current state of the zone solenoids and the pump motor. The data structure is an ES6 "Map", which is a true associative array.

So what the Object.assign(arg1, arg2) method does is overwrite key:value pair located in arg1 with the key:value pair in arg2.

Due to the write being done through a Proxy Object, another function is called which physically changes the state of the correct GPIO, and the function then sends another object back to the web browser to update its status display.

The ES6 Proxy Object appears to be useful for managing state in Javascript based embedded devices. The interested reader is encouraged to check out the documentation at this link:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

7.1.2 Server to Browser Hardware Status Message

When the project was first started, the buttons in the web browser to turn the hardware components on and off was simple DOM updates of the button colors.

However, this does not indicate the true state of the hardware in the case of a communications breakdown between browser and server.

The design was revised to update button indicators only upon receipt of a message from the server indicating a successful hardware state change.

The pumpActuator object emits “pumpStatusMessage”, and then only if a WebSocket is open the status message is sent to the browser. The web browser parses the message and uses DOM Javascript manipulation to update the displayed hardware status.

```
{'messageType':'pumpControl', 'pumpmotor':0, 'zone1':1, 'zone2':0}
```

7.1.3 Browser to Server Scheduling Message

The system user can input a date, start, and stop times for irrigation to happen automatically in the future. Simple HTML5 form fields are used to gather the user’s input, and then a “Schedule Irrigation” button is clicked. If a valid WebSocket is open, a scheduling JSON object is sent to the server.

```
{'messageType':'schedule', 'scheduleData':'Tuesday June 6th 2017',
'scheduleStart':'6:00:00 AM', 'scheduleStop':'7:00:00'}
```

The websocketserver determines the “messageType” is schedule, and then dispatches the message to the Scheduler Object.

7.1.4 Server to Browser Schedule Display Update Message

The Scheduler has the necessary logic to drive the hardware per the schedule.

The browser’s displayed schedule is updated by the server, not by the browser. Thus the server sends the scheduling message back to the browser, but only if the message is first processed by the Scheduler object and also a valid WebSocket is open. The message is identical to the browser to server message.

The scheduling display is updated using DOM Javascript methods.

Chapter 8

Solid State AC Relays

The BBGW GPIOs are low-current capability with logic voltage of 3.3 volts. This solid-state relay board allows the GPIOs to control the irrigation devices:

https://www.amazon.com/gp/product/B00ZZVQR5Q/ref=oh_aui_detailpage_o04_s00?ie=UTF8&psc=1

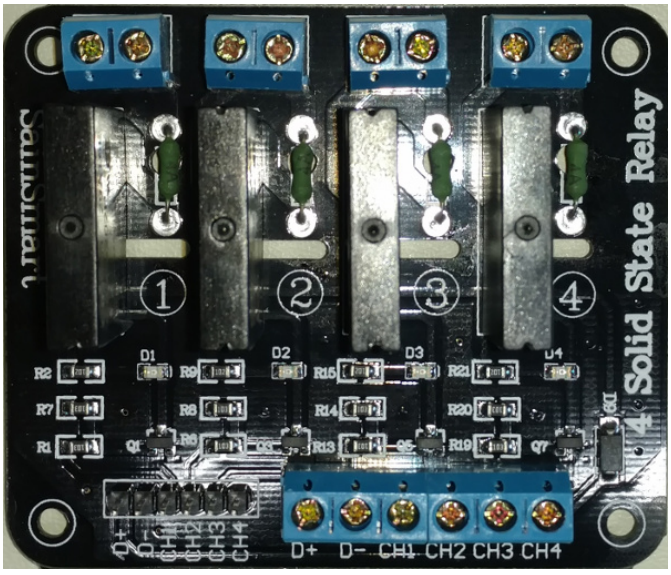

Figure 8.1: Solid-State Relay Board)

Note that in the photo above you can see that the board uses four independent solid-state relay modules. These modules use photo-electric isolation. This should allow excellent protection from voltage transients on the irrigation equipment side of the system from damaging the BBGW. Long-term testing needs to be done to see how well this works out.

The individual relay modules require 5 volt logic. However, there is some additional circuitry on the board which appears to allow less than 5 volt logic, as long as a 5 volt supply is available. Unfortunately a circuit diagram for this board could not be found. A "best guess" as to the hook-up seems to function well.

Viewing the photo of the board above, the screw terminals are labelled:

```
D+ D- CH1 CH2 CH3 CH4
```

The BBGW has 5 volts DC supply available on header pins P9-5 and P9-6 (VDD_5V), and also on P9-7 and P9-8 (SYS_5V).

VDD_5V is the same as the regulated 5 volt power supply which powers the BBGW via the USB connector. SYS_5V is switched on immediately after VDD_5V is applied to the board via the USB connector, so functionally it doesn't matter which supply is used to power the solid-state-relay input circuitry.

The suggested hook-up is to connect P9-5 to D+, and both of the ground pins (P9-1 or P9-2) to D-. The GPIOs are then connected to CH1, CH2, and CH3 as shown in the table:

Table 8.1: BBGW to Solid-State Relay Interconnect

Header Pin	Relay Terminal	Function
P9-1	D-	Ground
P9-2	D-	Ground
P9-5	D+	5 volts
P9-14	CH1	Pump Motor
P9-15	CH2	Zone 1 Solenoid
P9-16	CH3	Zone 2 Solenoid

Chapter 9

Configuration of the Beagle Bone Green Wireless

The default configuration of the Beagle Bone Green Wireless is as an “access point” (a wireless router). This is not a desired configuration for a dedicated embedded device as used in this project.

The following process re-configures the BBGW to a non-access point wireless mode.

The goal is to have a working wireless substitute for the Ethernet connector which does not exist on the BeagleBone Green Wireless. This is as a typical "headless" embedded project with the primary access using a terminal and ssh.

9.1 Get Latest IOT Image

Download and expand the IOT bone image per this link and flash to micro-sd. I used this one:

https://debian.beagleboard.org/images/bone-debian-8.7-iot-armhf-2017-03-19-4gb.img.xz

Write the image to the micro-sd (I put the micro-sd in a USB adapter plugged into my Ubuntu workstation):

```
xzcat bone-debian-8.7-iot-armhf-2017-03-19-4gb.img.xz | sudo dd of=/def/sdb
```

Eject the micro-sd from workstation and insert in the BBGW micro-sd slot. Connect a USB 3.3V serial device to the "debug serial header". The USB network connection could be substituted, however, my experience with this is that using the serial device is solid and will work consistently. Also, the BBGW doesn’t have a dedicated power connector. It uses the micro-USB. It is my preference to use a dedicated USB power supply and ignore USB as a network connection.

Power-up the BBGW and wait for the boot process to complete. Open a (bash) terminal and use the screen utility to connect via the serial USB device.

```
screen /dev/ttyUSB0 115200
```

I had to hit enter after the above command to get to the login prompt. The login user is debian and the password is temppwd.

You may have to install screen:

```
sudo apt-get install screen
```

After logging in the first time, a good thing to do first is to run this shell script:

```
cd /opt/scripts/tools
sudo ./grow_partition.sh
```

9.2 Wireless Network Configuration

Enter this command:

```
ifconfig
```

You should see 4 different network resources (not showing the full output here):

```
SoftAp0
lo
usb0
wlan0
```

The network resource SoftAp0 represents an "access point". The BBGW is configured as a wireless router! That is not the desired configuration, and fortunately this is easily removed. Edit the file:

```
/etc/default/bb-wl18xx
```

Change the line:

```
TETHER_ENABLED=yes
```

to

```
TETHER_ENABLED=no
```

Save and exit, and reboot, and login.

ifconfig should now show only 3: lo, usb0, and wlan0.

Now to configure WIFI! It is assumed you have a home wireless router and you know the SSID and passphrase. The router should be configured for DHCP (automatic assignment of IP addresses). From a terminal:

```
sudo connmanctl
connmanctl> scan wifi
Scan completed for wifi
connmanctl> services
    (your router broadcast)         (router info)
```

```
connmanctl> agent on
Agent registered
connmanctl> connect (copy router info here)
Agent RequestInput (router info)
  Passphrase = [ Type=psk, Requirement=mandatory, Alternates=[ WPS ] ]
  WPS = [ Type=wpspin, Requirement=alternate ]
Passphrase? (your passphrase)
Connected (router info)
connmanctl> quit
```

The above configuration is permanent and will survive reboot. An outstanding page with good info on connman:

https://wiki.archlinux.org/index.php/Connman

Another very good thing is to login to your router and use this to determine if your BBGW is successfully connected. And remember the router may have security settings which may block it from connecting. Also, rather than attempting to force a fixed IP address on the BBGW, I used the "address reservation" feature so that the IP address assigned by the router will be the same each time it connects. This is done using the MAC address of the BBGW.

After the above configuration is done, shutdown and remove the USB serial device. Power up the BBGW and wait for it to boot, and then using a terminal and ssh you should be able to connect to the BBGW as if an ethernet cable was connected:

ssh debian@(the assigned IP address)

After logging in you should have internet connectivity, so don't forget to:

```
sudo apt-get update
```

9.3 Copy Public SSH Key to BBGW

Another good thing to do is to copy your public ssh key to the BBGW:

```
ssh-copy-id -i id\_rsa.pub debian@192.168.1.3
```

where you need to substitute the IP address assigned to the BBGW by the router. This will allow you to bypass the usual login authentication routine.

9.4 USB Serial Device

Here is an example USB serial device. This should be on your tool kit list:

https://www.amazon.com/gp/product/B01AFQ00G2/ref=oh_aui_search_detailpage?ie=UTF8&psc=1

9.5 Change of Wireless Router

If the home wireless router is re-configured or replaced, this could require the use of the serial USB device again. That's a problem assuming the device is remote mounted.

One possibility is to configure the new WIFI router for deployment in the home network and place it in range of the BBGW. Using the above procedure for wireless network configuration, the router information for the new router is copied into the configuration while still connected with the old router.

As soon as the correct passphrase is entered the connection to the former router will drop. The new router can now be deployed and the BBGW will connect to it.

9.6 Flash the eMMC After Completing Development

Running the GNU/Linux OS and applications from the SD card is OK for development purposes. However, using an SD card is not a reliable long-term solution.

It is better to flash the contents of the SD card to the on-board eMMC flash drive. This device is specifically designed for use with an operating system and will adapt and survive as the memory cells wear out.

A simple change to a file on the SD card will cause the flash to commence upon the next boot-up. From the Beagleboard web site:

> To turn these images into eMMC flasher images, edit the /boot/uEnv.txt file on the Linux partition on the microSD card and remove the '#' on the line with 'cmdline=init=/opt/scripts/tools/eMMC/init-eMMC-flasher-v3.sh'. Enabling this will cause booting the microSD card to flash the eMMC. Images are no longer provided here for this to avoid people accidentally overwriting their eMMC flash.

Chapter 10

Device Tree Requirements

This project requires only three GPIOs. Rather than editing device tree files, and having to deal with potential bugs caused by this, the excellent config-pin utility was used. This utility is provided by the Universal IO project.

The Universal IO project is located at this Github repository:

https://github.com/cdsteinkuehler/beaglebone-universal-io

Universal IO is included with the most recent Debian-based IOT images.

The setting of the three GPIOs is done by the pumpActuator class. This is most appropriate, as the class uses these GPIOs to control the irrigation devices. The GPIO configuration is done by the class constructor:

```
setGpio(headerPin) {
    const exec = require('child_process').exec;
    console.log(`Setting header pin ${headerPin} to GPIO mode.`);
    exec(`config-pin ${headerPin} low_pd`);
}
```

Note the use of the Node.js "child_process" as described in the chapter on GPIO control with sysfs.

The three pins are set to GPIO low state and pull-down mode via these three lines in the class constructor using the above method:

```
this.setGpio('P9.14');
this.setGpio('P9.15');
this.setGpio('P9.16');
```

This partitioning of the GPIO mode set functioning into the pumpActuator class is appropriate and eliminated the requirement for an auxiliary shell script for control of header pin modes.

Chapter 11

Setting the BBGW to Local Time

The default time setting of the BBGW was found to be UTC. It was desired to have this set to local time.

Fortunately, there is a web page by Derek Molloy which covers this subject in detail:

http://derekmolloy.ie/automatically-setting-the-beaglebone-black-time-using-ntp/

CHECK IF DEFAULT IMAGE HAS ACTIVE NTP

The change to local time is done by removing an existing file, and then adding a symbolic link:

```
cd /etc
rm localtime
ln -s /usr/share/zoneinfo/America/New_York /etc/localtime
```

In the above example, the time zone is set to North America Eastern (New York). A complete listing of the possibilities is found in/usr/share/zoneinfo. Simply find the appropriate path for your time zone and create the link.

The time zone setting will take effect upon the next boot.

Chapter 12

Running the Project

Chapter 13

Setting up a systemd Irrigation Service

systemd is responsible for booting up the user space in Debian (and other) GNU/Linux distributions including the BBGW.

This system can be used to start the Node.js server as an “irrigation” service.

This is easy to accomplish and can be done by creating a single text file:

`/etc/systemd/system/irrigation.service`

Here is the contents of the file irrigation.service:

```
[Unit]
Description=Irrigation Control Server

[Service]
ExecStart=/usr/bin/node /home/debian/irrigate-control/software/node/server.js

[Install]
WantedBy=graphical.target
```

The [Unit] section provides a short description of the service which is printed out when the service is interrogated.

The [Service] section is the complete path to the node command followed by the path to the server.js file. This is the “service” which will be “daemonized” at boot.

The [Install] section indicates the default state in which the service should be started. The default state can be found by using this command:

systemctl get-default

In the case of the IOT distribution used by this project, the response is:

`graphical.target`

Once the service unit file is in place, enable the service like this:

`systemctl enable irrigation`

The irrigation service will now start at boot time! Set a bookmark in your browser, and simply click to go straight to the irrigation control page.

To permanently disable the service:

```
systemctl disable irrigation
```

When debugging, it may be necessary to temporarily stop the service. Use this command:

```
systemctl stop irrigation
```

To start the service again:

```
systemctl start irrigation
```

Chapter 14

Resources

14.1 Github repository for this project

https://github.com/Greg-R/

14.2 Beagle Bone Green Wireless

https://www.seeedstudio.com/SeeedStudio-BeagleBone-Green-Wireless-p-2650.html