Appendicies

1 Device JSON Parser

```c
uint8_t configDevices(const char *configJSON) {
    // Clear previous device config
    clearDevices();

    // Setup devices according to JSON description
    // Create a JSON parser object
    MbedJSONValue jsonParser;

    // Parse the JSON string and store the result in jsonParser
    parse(jsonParser, configJSON);

    // Loop through the JSON, extracting the device configuration for each device ID
    for (uint8_t i = 0; jsonParser[i].hasMember((char *)"devID"); i++) {
        // Create variables to hold the extracted values
        uint16_t devID;
        uint16_t devPin1;
        short devPin2;
        std::string devType;
```
/Caution - always check if the object contains the requested value before attempting to access it, otherwise a hardfault occurs from trying to access invalid memory

if (jsonParser[i].hasMember((char *)"devID")) {
    // Have to get the value as a string and then convert it to an integer due to limitations with the JSON parser library
    devID = std::stoi(jsonParser[i]["devID"].get<std::string>());
}

if (jsonParser[i].hasMember((char *)"devPin1")) {
    // Have to get the value as a string and then convert it to an integer due to limitations with the JSON parser library
    devPin1 = std::stoi(jsonParser[i]["devPin1"].get<std::string>());
}

if (jsonParser[i].hasMember((char *)"devPin2")) {
    // Have to get the value as a string and then convert it to an integer due to limitations with the JSON parser library
    devPin2 = std::stoi(jsonParser[i]["devPin2"].get<std::string>());
}

if (jsonParser[i].hasMember((char *)"devType")) {
    // Get the device type string
    devType = jsonParser[i]["devType"].get<std::string>();
}

if (devType.find("perPump") != string::npos) {
    // Create perPump object
    devices[i] = new perPump(digitalOutputs[devPin1-1], devID);
} else if (devType.find("solValve") != string::npos) {
    //Create solValve object
    devices[i] = new solValve(digitalOutputs[devPin1-1], devID);
}

} else if (devType.find("sixValve") != string::npos) {
    //Create sixValve object
    devices[i] = new sixValve(digitalOutputs[devPin1-1],
    digitalOutputs[devPin2-1], devID);
}

} else if (devType.find("switchValve") != string::npos) {
    //Check if the switchValve is working in one or two pin mode
    if (devPin2 == -1) {
        //Create switchValve object with one pin
        devices[i] = new switchValve(digitalOutputs[devPin1-1], devID);
    } else {
        //Create switchValve object with two pins
        devices[i] = new switchValve(digitalOutputs[devPin1-1],
        digitalOutputs[devPin2-1], devID);
    }
}
} else {
    //Debugging, send the client information over serial
    serialQueue.call(printf, "Error creating object, could not determine
device type, device ID: %d, device pin 1: %d, device pin 2: %d\n", devID, devPin1, devPin2);

    //An error has occurred, signal failure to configure
    return 1;
}

} else {
    //Debugging, send the client information over serial
    serialQueue.call(printf, "Error reading device pin 2, device ID: %d, device pin 1: %d\n", devID, devPin1);

    //An error has occurred, signal failure to configure
    return 1;
}
} else {
    //Debugging, send the client information over serial
    serialQueue.call(printf, "Error reading device pin 1, device ID: %d, device pin 1: \n", devID, devPin1);

    //An error has occurred, signal failure to configure
    return 1;
}

} else {
    //Debugging, send the client information over serial
    serialQueue.call(printf, "Error reading device pin 1, device ID: %d\n", devID);

    //An error has occurred, signal failure to configure
    return 1;
}

} else {
    //Debugging, send the client information over serial
    serialQueue.call(printf, "Error reading device ID\n");

    //An error has occurred, signal failure to configure
    return 1;
}

// No errors, signal success
return 0;
}
2 Routine JSON Parser

```c
uint8_t configRoutine(const char *configJSON, uint16_t routineID) {
    // Clear the routine vector of any previous timing information
    routine.clear();

    // Setup routines according to JSON description
    // Create a JSON parser object
    MbedJSONValue jsonParser;

    // Parse the JSON string and store the result in jsonParser
    parse(jsonParser, configJSON);

    // Check for name/ID of routine
    // Check for timings array
    // Loop through array, extracting timing data

    // Loop through all of the routines that have a valid ID
    for (uint8_t i = 0; jsonParser[i].hasMember((char *)"routineID"); i++) {
```
//Caution - always check if the object contains the requested value before attempting to
access it, otherwise a hardfault occurs from trying to access invalid memory

//Get the current routine ID
if (jsonParser[i].hasMember((char *)"routineID")) {
    uint16_t currentRoutineID =
    std::stoi(jsonParser[i]["routineID"].get<std::string>());
}

//If the current iteration is the requested routine ID, load the data into the
vector
if (routineID == currentRoutineID) {

    //Check if the timings array is present
    if (jsonParser[i].hasMember((char *)"timings")) {

        //Loop through the timings array, extracting timing info
        for (uint16_t j = 0; jsonParser[i]["timings"][j].hasMember((char *)"devID"); j++) {

            //Create a variable to hold the extracted values
            deviceTimes time;
//Check for the deviceID
if (jsonParser[i]["timings"][j].hasMember((char *)"devID")) {
    //Have to get the value as a string and then convert it to an
    //integer due to limitations with the JSON parser library
    time.devID =
    std::stoi(jsonParser[i]["timings"][j]["devID"].get<std::string>());
}

//Check for the start time
if (jsonParser[i]["timings"][j].hasMember((char *)"timeStart")) {
    //Have to get the value as a string and then convert it to an
    //integer due to limitations with the JSON parser library
    time.startTime =
    std::stoi(jsonParser[i]["timings"][j]["timeStart"].get<std::string>());
}

//Check for the stop time
if (jsonParser[i]["timings"][j].hasMember((char *)"timeStop")) {
    //Have to get the value as a string and then convert it to an
    //integer due to limitations with the JSON parser library
    time.stopTime =
    std::stoi(jsonParser[i]["timings"][j]["timeStop"].get<std::string>());
}
/\Check for the device state

if (jsonParser[i]["timings"][j].hasMember((char *)"state"))
{
    // Have to get the value as a string and then convert it
to an integer due to limitations with the JSON parser library
    time.devState =
    std::stoi(jsonParser[i]["timings"][j]["state"].get<std::string>());

    // Add the timing information to the routines vector
    routine.emplace_back(time);

    } else {
        // Debugging, send the client information over serial
        serialQueue.call.printf("Error reading the device
state, device ID: %d, start time: %d, stop time: %d\n", time.devID, time.startTime,
        time.stopTime);

        // An error has occurred, signal failure to load the
        routine
        return 1;
    }
} else {
    // Debugging, send the client information over serial
    serialQueue.call.printf("Error reading the stop time,
device ID: %d, start time: %d\n", time.devID, time.startTime);
//An error has occurred, signal failure to load the routine
return 1;
}
} else {
//Debugging, send the client information over serial
serialQueue.call(printf, "Error reading the start time, device
ID: %d\n", time.devID);

//An error has occurred, signal failure to load the routine
return 1;
}
} else {
//Debugging, send the client information over serial
serialQueue.call(printf, "Error reading device ID\n");

//An error has occurred, signal failure to load the routine
return 1;
}
}

//Routine loaded, signal success
return 0;
else {
    // Debugging, send the client information over serial
    serialQueue.call(printf, "Error reading the timings array, Routine ID: %d\n", currentRoutineID);

    // An error has occurred, signal failure to load the routine
    return 1;
}

else {
    // Debugging, send the client information over serial
    serialQueue.call(printf, "Error reading routine ID\n");

    // Cannot read routine ID, signal failure
    return 1;
}
//Debugging, send the client information over serial
serialQueue.call(printf, "No routine found matching the requested ID, %d\n", routineID);

    //No routine found with given ID, signal failure
    return 1;
}
3 Routine Duration Function

```c
uint16_t routineDuration(void) {

    uint16_t duration = 0;

    //Loop through the timings array and look for the largest value of timeStop
    for (deviceTimes n : routine) {
        //If the stop time for the current step is greater than any previous stop time
        if (n.stopTime >= duration) {

            //Update the last time value with the new greatest value
            duration = n.stopTime;

        }
    }

    return duration;
}
```
4 Testing Devices Function

```javascript
$(document).on("click", "button.btnTstDevices", function (event) {

    //Get the ID of the selected routine
    var routineID = $('select[id="routines_dropdown"] option:selected').attr('class');

    //Check if a valid routine has been selected
    if (typeof routineID == "undefined") {
        //Tell the user to select a valid device
        alert("Please select a routine, or create one if none are available");
        //Return and do not execute the rest of the function as there is an invalid input
        return 0;
    }

    //Define the URL to hit
    var reqURL = '/testdevices/routineid=' + routineID;

});
```
//Send the AJAX request to the defined URL
$.ajax({
    type: "GET",
    url: reqURL,
    //On success, alert the user
    success: function (result) {
        alert('Device test successful');
    },
    //On failure, alert the user
    error: function (result) {
        alert('There is an issue with testing a device');
    }
});
function updateDeviceConfig() {

    //Declare a local variable to hold the JSON as a string
    let devicesJSON = [];

    //Open our object store
    let objectStore = db.transaction('devices').objectStore('devices');

    //Get a cursor list of all the different data items in the IDB to iterate through
    objectStore.openCursor().onsuccess = function (event) {
        //Get the cursor
        let cursor = event.target.result;

        //If there is still another device, keep running
        if (cursor) {

// Build the json for the device
let device = {
    devID: cursor.value.devID,
    devName: cursor.value.devName,
    devType: cursor.value.devType,
    devPin1: cursor.value.devPin1,
    devPin2: cursor.value.devPin2
};

// Push the devices onto the array
devicesJSON.push(device);

// Continue to the next item in the cursor
cursor.continue();

// If all the devices have been read, send the updated config to the MCU
} else {
    // Define the URL to hit with the updated configuration
    var reqURL = '/updatedevices/' + JSON.stringify(devicesJSON);
// Send the AJAX request to the defined URL
$.ajax({
    type: "GET",
    url: reqURL,
    // On success, refresh the page
    success: function (result) {
        window.reload();
    },
    // On failure, alert the user
    error: function (result) {
        alert('There was an issue adding the device');
    }
});
else if (address.find("updatedevices/") != string::npos) {

    //Find the start of the config JSON
    int configStart = address.find("updatedevices/");

    //Get the config string
    string newDevConfig = address.substr(configStart + 14);

    //Attempt to update the device configuration
    if (configDevices((char *)newDevConfig.c_str())) {
        //An error occurred with updating the configuration, add a 404 header code to the response
        response += HTTP_STATUS_LINE_404;
    }
    //Add a line feed and carriage return to the response
    response += "\r\n";
}
else {
    // Success, add a 200 header code to the response
    response += HTTP_STATUS_LINE_200;

    // Add a line feed and carriage return to the response
    response += "\r\n";
}
}
function genVisHTML(timings) {

  // Parse timings JSON
  // Sort the timings array by start time
  // Create a unique list of the deviceIDs used in the routine by using getUniqueDevices()
  // Create an array to hold the unique deviceID timing span blocks
  // Get the names of the devices using getDeviceName()
  // Loop through the timings array, generating the span blocks and appending to the specified deviceID row
  // Append closing div tags to each device row
  // Concatenate the html together and return as .html
try {
    times = JSON.parse(timings);
} catch (e) {
    // If there was an error parsing the JSON, return an error
    return {
        code: 1,
        msg: "There was a problem parsing the JSON string",
        devices: [],
    };
}

// Sort the timings by start time
times.sort(sortByProperty("timeStart"));

// Get a list of all devices used
var uniqueDevices = getUniqueDevices(timings);
//Check if failed - code non-zero
if (uniqueDevices.code) {

    //Alert that an error occurred
    alert("Error getting the list of devices used");

    //Log specific error
    console.log(duration.msg);
}

//Create an array to store the html row data
var htmlRows = [];
//Loop through the unique devices, and create the initial row html
for (ij in uniqueDevices.devices) {

    //Attempt to get the device name from the deviceID
    let devName = getDeviceName(uniqueDevices.devices[ij]);

    //Check if failed - code non-zero
    if (devName.code) {

        //Alert that an error occurred
        alert("Error getting device name");

        //Log specific error
        console.log(devName.msg);
    }

    //Initialise the row with the name of the device and the class for the chart
    htmlRows[ij] = '<div class="row"> <h6>' + devName.name + '</h6><div class="chart">';
}
// Attempt to get the duration of the routine
var duration = getDuration(timings);

// Check if failed - code non-zero
if (duration.code) {
   // Alert that an error occurred
   alert("Error getting the duration of the routine");

   // Log specific error
   console.log(duration.msg);
}

// Loop through the timings array and generate the span blocks
for (ik in times) {

    // Get the type of the device
    var deviceType = getDeviceType(times[ik].devID);

    // Check if failed - code non-zero
    if (deviceType.code) {

        // Alert that an error occurred
        alert("Error getting the type of the device");

        // Log specific error
        console.log(deviceType.msg);
    }
}
// Get the pretty name of the current state
var pName = getPrettyState(times[ik].state, deviceType.type);

// Check if failed - code non-zero
if (pName.code) {
  // Alert that an error occurred
  alert("Error getting the pretty name of the current state");

  // Log specific error
  console.log(pName.msg);
}
// Calculate the width of the block in the figure, in %
// Calculate the duration of the block
var stepDur = times[ik].timeStop - times[ik].timeStart;

// Convert this to the width of the block by multiplying by 100/duration
var blockWidth = stepDur * (100 / duration.dur);

// Create visualisation HTML span
var rowHTML = '<span style="width:' + blockWidth + '%;"class="block state" + pName.pState + " title="" + times[ik].timeStart + '-' + times[ik].timeStop + ' seconds"><span class="dspState">" + pName.pState + '</span></span>';

// If the deviceID can be found in the timings array, append the next span block to it
if (uniqueDevices.devices.indexOf(parseInt(times[ik].devID)) != -1) {
    // Get the index of the deviceID row specified in the timings block
    let index = uniqueDevices.devices.indexOf(parseInt(times[ik].devID));

    // Append the new block to the existing HTML
    htmlRows[index] += rowHTML;
}
//Loop through the unique devices, and append the closing div tags
for (ji in uniqueDevices.devices) {

    //Append the closing div tags to each of the rows
    htmlRows[ji] += '</div></div>';
}

//Join the rows together into 1 string to return
visHtml = htmlRows.join('"");

//Return the duration of the routine
return {
    code: 0,
    msg: "Success",
    html: visHtml,
};