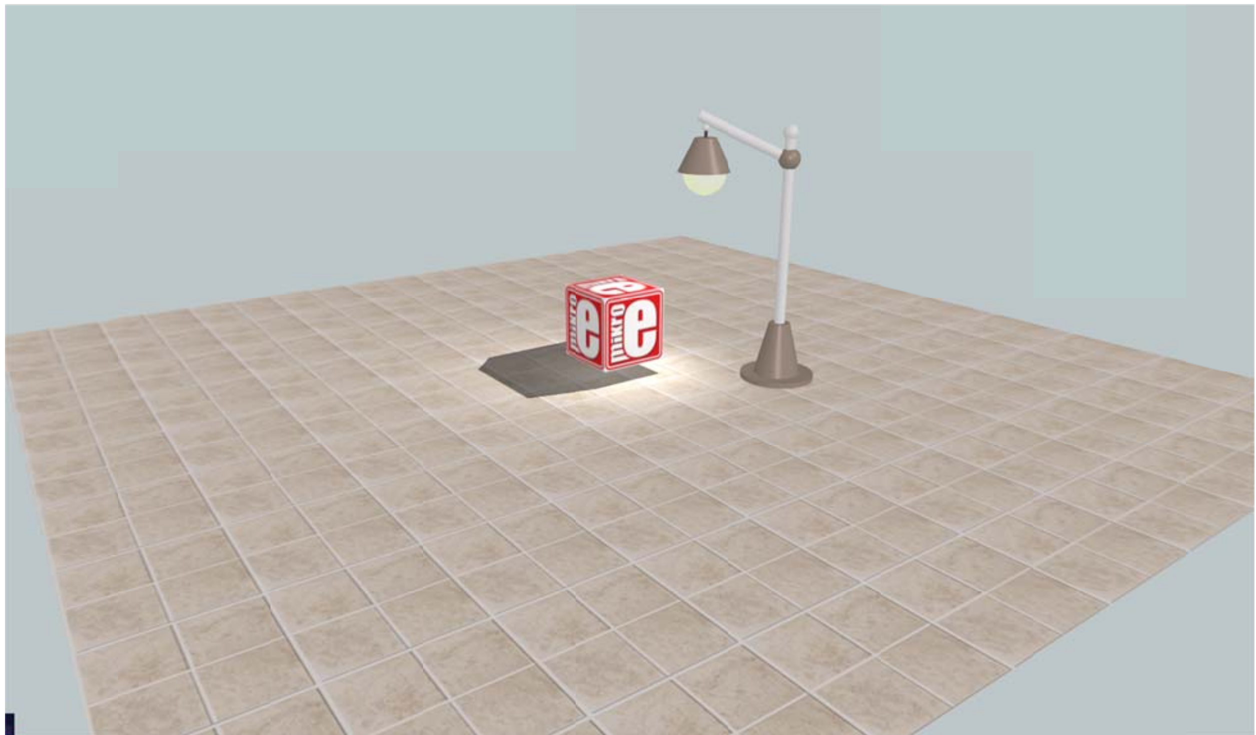
WIFI WebGL (three.js) Lamp control with PIC micro



Content

Page
1. Introduction……………………………………………………………………………….. 3
2. WebGL (Three.js) program ………………………………………………………… 4
3. PIC communication ………………………………………………………………….. 11
4. MikroC program ………………………………………………………………………. 13
5. Triac control …………………………………………………………………………… 16
6. Additional considerations ………………………………………………………….. 22
7. License …………………………………………………………………………………. 25

Introduction

Traditionally the 3d creation for the web was not natively possible and plugins had to be used for such a task as for example in the case of Java3D. Now the release of WebGL (Web graphics library) by the non-profit Khronos Group have opened new possibilities that have been used in this project to build an embedded electronic control (and monitoring) system.

It is yet to gain more support among the browsers, especially internet explorer (not supported at all except by using third party plugins) and mobile browsers, but hopefully the near future will be a history of success.

One of the features that allowed the embedded project with a PIC microcontroller was the fact that WebGL is programmed using JavaScript (and shader code), the native script language of all mayor browsers and its ability of executing on the client side, using the client computer GPU for graphics processing, allowing the MCU to delegate the graphics intensive tasks. Also the ability of making Ajax calls to the server allowed the WebGL virtual world to behave as a desktop-like program.

However the goods, this graphic library for the web will not be suitable for those who want to protect the programming code when making a marketable product, as it is not what JavaScript was intended for.

This project is composed of 4 main parts:

1- The WebGL programming (**main.js**), which was developed from ground up using the library "**Three.js**", an open sourced project aimed to abstract the common web programmer from the complexities of raw WebGL.
2- The PIC communication script (**PicComm.js**), as I've called it, which is in charge of the Ajax communication between the PIC micro and the WEBGL program.
3- The mikroC program which was developed using the mikroC Pro for PIC v6 compiler by Mikroelektronika. This program is based in the "**Ajax Wifi Http demo with microSD storage**" (available on libstock.com), adapted for this project specific needs and is the one (for those who don't know) that will ultimately be programmed into the PIC microcontroller ROM.
4- The last part explains how to control a 100W/120Vrms light bulb (resistive load) utilizing a triac.

To test the project, the mikroC program has to be configured for the local Wifi network SSID and password. All files provided are to be included in the root directory of the microSD card.

Then, after the EasyPICv7 board and its parts are properly configured (see mikroC program headers or mikroC program section below for instructions) and interfaced to the Triac circuit (Warning! Life hazard when working with this part) and the external files loaded in the microSD, the web page served by the PIC web server can be viewed in any computer in the local network, (even through internet with proper port forwarding router configuration) using the address: http://192.168.1.100 in Google Chrome or Mozilla Firefox. In my testing the Opera browser did not displayed the 3D geometries, even though WebGL was previously enabled and it had (at least) basic WebGL functionalities. I couldn't test it with Safari in the Mac platform but when I do it will surely post it here.

WebGL (Three.js) program

WebGL is a low level API that has brought real-time, hardware-accelerated 3D rendering capability to desktop and mobile web browsers, opening up possibilities for new kinds of connected visual applications. [1]

Low level means that it is designed to give access from JavaScript to as much of the power of the user's graphics hardware as it is compatible with, being possible to run on a broad spectrum of devices, like PCs with NVidia or ATI graphics, all the way to smartphones and with security, with the minimum number of functions and data types. This facts have made it difficult to code than average web standards, however, to date multiple JavaScript libraries have been written to abstract the most difficult and repetitive tasks. Among them **Three.js** seems to be prevailing and that's why it has been chosen for this project. [2]

Three.js on its side is a little hard to learn due to lack of documentation. The problem probably lies in the fact that since its first release, not long ago, has been quickly developing and with many constant changes, the documentation hasn't been updated at the same pace, some features have not ever been described, so it has been learned mainly by reading the code and questions asked in the http://stackoverflow.com forum.

However, recently the course "Interactive 3D Graphics" was released on Udacity.com for free, what has been a big push for the Three.js learning process. [3]

Another good resource, even used by the Udacity course instructor, Eric Haines, is the Lee Stemkoski Three.js examples on the Github repository that are more newbie friendly than the Threejs.org official examples. [5]

In this project I created a 3D world composed of a floor, a street lamp, and a cube. It also has different types of lights to make the objects visible. Moreover it features the Three.js Orbit Controls so when loaded, the 3D world can be controlled in the following way:

1. Orbit 360 degrees around Y axis and 180 degrees in the imaginary line traced horizontally from side to side of the display by holding left click and moving the mouse.
2. Pan the view holding the right click and moving the mouse.
3. Zoom in/out scrolling.

Initially the 3D world will capture the lamp light bulb status (on/off) (the PORTA0 status actually) and display it on the web page with the lights On or Off accordingly. At the very beginning (when just connected to the local network) the PORTA0 of the PIC will be off, but after some different users have been connected from different computers it could be in any state, which will be properly captured by the application when loading the web page the first time, what proves that not only control is performed but also monitoring.

The on/off switching will be done clicking on the lamp glass and only there, which features a Three.js picking implementation.

To make it more realistic, when on, the light hitting the cube will project a shadow of it on the floor.

Any Three.js implementation has to load and execute the core first, before anything else can run. It has been called ***threemin.js*** in this project (originally three.min.js) as it is minified.

Also included are the **orbit controls** and the **stats** archives as part of the Three.js implementation with the name ***OrbitC.js*** (originally OrbitControls.js) and ***statsmin.js*** (originally stats.min.js) respectively.

Finally the actual project source code was called ***main.js.*** It has to be loaded after the three libraries described above are loaded and executed and it has probably been commented enough to make somebody with some knowledge of the library understand it, but following, some parts will be explained for further clarification.

Initially the program will load the following functions:

```
//Call all functions

try {

WebGLdetector();

        init();

        fillScene();

        statistics();

        animate();

}

//In case of error provide appropriate message

catch(e) {

        if ( ! Detector.webgl )Detector.addGetWebGLMessage();//If WebGL is not supported

        else{//If the problem is other

        varerrorReport = "Your program encountered an unrecoverable error,

cannot draw on canvas.  Error was:<br/><br/>";

        $('#ErrorContainer').append(errorReport+e);

        }

}
```

What is done above is try to execute the functions, if they fail then show an appropriate message to the user.

The very first function is WebGLdetector() where browser WebGL support is checked, so if the answer is negative we won't go any further as another browser, update to the graphics card or any other action will be required for the WebGL to work. At that point the addGetWebGLMessage from the object Detector will be thrown.

Also if any other error was found in the other following functions the message: - ***Your program encountered an unrecoverable error, cannot draw on canvas. Error was:***- will be displayed with a hint of what is it the issue. (This last idea was taken from udacity.com course)

In the happy event that everything goes well, in the init() function we create some Three.js (and WebGL) must have, like create the *scene,* create the camera and its configuration, obtain and set up the WebGL context and start rendering content, etcetera.

The fillscene() fuction as its name implies is in charge of filling the 3D world with all its content. There the lights are created and added to the scene:

```
ambient = new THREE.AmbientLight( 0x4c4a4a );

        scene.add(ambient);

vardirectionalLight = new THREE.DirectionalLight(0xffffff, 0); //Always ON

        directionalLight.position.set(320, 250, -80);

        directionalLight.intensity = 0.15;

scene.add(directionalLight);
```

Also the floor geometry and its tile material what makes it finally look as a house floor:

```
varfloorTexture = new THREE.ImageUtils.loadTexture( 'floor.png', new THREE.UVMapping(), ...

varfloorGeometry = new THREE.PlaneGeometry(1000, 1000, 100, 100);
```

The cube geometry and its mikroe logo material:

(I got written permission from mikroe to use their logo)

```
varCubeimgTexture = THREE.ImageUtils.loadTexture('cube.png', new THREE.UVMapping() …

varCubeimgMaterial = new THREE.MeshLambertMaterial( { map : CubeimgTexture } );

varcubeGeometry = new THREE.CubeGeometry( 70, 70, 70 );
```

Note that in the above cases actual images are used as textures (e.g. cube.png, floor.png) to build the materials and give a more realistic skin to the geometries, but it is not required; materials can be entirely made upon coding.

This materials and geometries are embedded in what is called a **MESH** and that way added to the main **SCENE** object.

```
cube = new THREE.Mesh( cubeGeometry, CubeimgMaterial );

            scene.add(cube);
```

The lamp on its side is a multi-mesh object so what is done is that all these smaller objects are first created and then embedded into another object called object3D in Three.js, that way if for example a translation, rotation or scale is needed; no individual parameters have to be adjusted, but just manipulate the object3D.

```
LampGroup = new THREE.Object3D();//create an empty container

LampGroup.add(LampBase );//add a mesh with geometry to it

LampGroup.add(LampBaseCap );

LampGroup.add(LampPost );

LampGroup.add(LampArm );

LampGroup.add( sphere );

LampGroup.add( sphere2 );

LampGroup.add(LampCap );

LampGroup.add( Hanger );

LampGroup.add( sphere3);

LampGroup.add(LightBulb);

LampGroup.add( spotlight);

scene.add(LampGroup );//when done adding meshes, add the group to the scene
```

One thing to note here is how the cube shadow is produced on the floor.

A spotlight has been used to make our cube the center of attention and cast shadow. This light has been made coincide in the same point of space that is our light bulb (actually viceversa).

```
LightBulb.position.copy( spotlight.position);
```

Then the spotlight has to be enabled to cast shadows.

```
spotlight.castShadow = true;
```

Same thing has to be done with the cube.

```
cube.castShadow = true;
```

And the floor has to be enabled to receive the shadow.

```
floor.receiveShadow = true;
```

Object picking

There is not something like DOM in WebGL, so world calculations have to be made to detect what object was actually interacted with. This is done through a proceeding called **RayCasting**, utilizing the un-projection of rays from the 2D screen to the 3d world. For more info on this refer to the Object picking tutorial by Soledad Penades (http://soledadpenades.com/articles/three-js-tutorials/object-picking).

At the end of the fillscene() the objects that will be check upon in an onDocumentMouseDown event are pushed to the targetList array. Pushed means added to the end of the array.

```
targetList.push(LightBulb);

targetList.push(LampCap);    // to exclude (later in code) all additional objects that may turn on the lamp

targetList.push(floor);      // because are in the ray path.
```

The only one that actually mater to us is the ***LightBulb*** (that is the only that will have clicking detection capabilities), but ***LampCap*** and ***floor*** are added so that when the light cap and the floor are in front of the light bulb they will block it from detecting clicking even if in the ray path. Try commenting these two last lines and you will see what I'm talking about.

In the following piece of code of onDocumentMouseDown() function, is where all of the clicking checking actually happens. Here any matching with the objects contained in targetList will evaluate true the first ***if*** statement. If the object clicked is the first element in targetList (our light bulb) then click detection is confirmed and the lamp (virtual and real) will switch state calling the *updateSceneLighting_And_CommandPIC(); function .*

```
// create an array containing all objects in the scene with which the ray intersects

        var intersects = ray.intersectObjects( targetList );

        // if there is one (or more) intersections

        if ( intersects.length> 0 ){

        // only the if first object intersected is the LightBulb

        // the lamp will be turned on/off.

        if ( targetList[0] == intersects[ 0 ].object ) {

        callChangeStatus = 0; // only with mouse click, the change_Lamp_Status() function will be called.

        updateSceneLighting_And_CommandPIC();

        }

        }
```

The last two functions called on startup are statistics() and animate().

statistics() is used to show the left down corner box showing the frames per second actually rendered by our scene in real time.

animate() is key for the Three.js world. It contains the requestAnimationFrame() method that will update the stats and controls and continuously repaint (render) our scene. This repaint may occur up to 60 times per second for foreground tabs (the exact rate is up to the browser to decide), but may be reduced to a lower rate in background tabs, consequently releasing our system resources, improving performance and literally saving energy.

To finish with ***main.js*** let's talk about the function updateSceneLighting_And_CommandPIC(), it is essential for our invention. In this function is where the scene parameters will be set to turn on/off the 3D world.

For example: if turning off, it will darken the ambient light color, drive to zero the spotlight intensity and shadow darkness etc.

Also there the communication with a function in an external JavaScript file will be established, it is the ***PicComm.js*** (of whom will talk about later) calling the function change_Lamp_Status() that is the one that tells the PIC micro to change its output on PORTA.

In the code below you may note that I've used two flags that may confuse a little bit but following will try to briefly explain them. They are ***callChangeStatus*** and ***toggle***.

Initially ***callChangeStatus*** and ***toggle*** will be equal to 1 as it is the set value at the time of the variable declaration and also the scene will be dark, signaling ***lamp off***. Then when the light bulb is clicked, ***callChangeStatus*** is set to 0 and updateSceneLighting_And_CommandPIC() is invoked from inside onDocumentMouseDown(). Once in updateSceneLighting_And_CommandPIC() the ***if*** statement will evaluate to true as ***toogle*** is still 1 and the scene will remain dark, but as ***callChangeStatus** = 0, change_Lamp_Status()* will be invoked. This last function will try to set high the PIC PORTA0 and if succeeded it will set the variable ***toggle*** = 0 and call back updateSceneLighting_And_CommandPIC(), but at this time ***callChangeStatus*** = 1 (to prevent and infinite loop) , so with **toggle** = 0 the ***If*** statement will evaluate false and continue to the ***else*** consequently turning on the scene. Similar process applies to turn it back off.

All the above process turns to be somewhat cumbersome but it is necessary to keep the PIC port status synchronized with the scene status.

In case my last explanation is felt senseless (what wouldn't surprise me) trying to analyze the code itself may be an easier election.

```
functionupdateSceneLighting_And_CommandPIC(){

if (toggle){

        ambient.color.set(0x4c4a4a);

spotlight.intensity = 0;

spotlight.shadowDarkness = 0;

LightBulbMaterial.emissive.setHSL( 0.17, 1, 0);

skyBoxMaterial.color.set (0x4e4b4b);

        If (!callChangeStatus){
```

```
            callChangeStatus = 1;

            change_Lamp_Status();//PIC communication function located in PicComm.js file

              }

}

else{

            ambient.color.set(0xaba8a8);

            spotlight.intensity = 0.5;

spotlight.shadowDarkness = 0.5;

LightBulbMaterial.emissive.setHSL( 0.17, 1, 0.5);

skyBoxMaterial.color.set (0xbdc7c9);

            if (!callChangeStatus){

            callChangeStatus = 1;

            change_Lamp_Status();

              }

}

}
```

WEBGL section references

1. "Khronos Releases Final WebGL 1.0 Specification". Retrieved 3 March 2011.
2. "WebGL: Frequently Asked Questions." *Learning Webgl FAQs*. N.p., n.d. Web. 28 July 2013.
3. Haines, Eric, and GundegaDekena. "Interactive 3D Graphics." *Course with Three.js &WebGL*. Udacity, n.d. Web. 28 July 2013.
4. "WebGL." *Wikipedia*. Wikimedia Foundation, 27 July 2013. Web. 28 July 2013.
5. Stemkoski, Lee. "Three.js - Examples." *Three.js - Examples*. Github, n.d. Web. 28 July 2013.
6. Cabello, Ricardo. "Three.js R59." *Three.js*. N.p., n.d. Web. 28 July 2013.

PIC communication program

The ***PicComm.js*** file which we have talk about in the previous section is the actually container of the code in charge of communicating with the PIC micro from the web side.

It will essentially make ajax calls (using jQuery library) to the PIC http server, requesting it to toogle the port state through a GET request: "… ***?port="+checkboxIndex*** and at the same time requesting the ***portScript.js*** file which will contain the final PORTA0 status.

```
PIC_Request = function (){
$.getScript("portScript.js?port="+checkboxIndex)
                .done(function (data) {
portaStatus();
            ajax_rqst_status =0;
                })
                .fail(function() {
portaStatus();
            ajax_rqst_status==1;
              });
      }
```

If the call succeed or fail, whichever is the case, portaStatus() will be invoked and ***toggle*** variable updated acordingly and sent back to the updateSceneLighting_And_CommandPIC() in the ***main.js*** file.

```
portaStatus = function (){     //Retrieve PORTA status and update
var i=0;
if(PORTA&(1<<i)){
              //code to turn on the lamp
toggle = 0;
            updateSceneLighting_And_CommandPIC();
              }
else {
            //code to turn off the lamp
toggle = 1;
```

```
        updateSceneLighting_And_CommandPIC();

          }

    }
```

This entire script (PicComm.js) is a simplification of the one used in the **Ajax Wifi Http demo with microSD storage** (available on libstock.com) as for example here the scene and PIC status can be kept synchronized when clicking the light bulb, but this synchronization won't happen automatically if another user in another computer change the lamp status, neither we can detect server or network down here as it is possible with the mentioned Ajax demo.

MikroC program

The program of this section was again another modification of the ***Ajax Wifi Http demo with microSD storage,*** which on its side is a son of the ***Mikroelektronika's Wifi Http Demo.***

For this project the EasyPICv7 board shown below was used to facilitate programming connections and others, but if not available, connecting the single parts like the PIC, the LCD, the buzzer etc. is still a possibility. The Wifi plus click, microSD click and an actual microSD card are also required.

Figure. EasyPIC v7 board from Mikroelektronika

The following configuration parameters and hardware have to be observed:

Defaults on the program:

IP: 192.168.1.100 (change if desired or required)

SSID of AP is: mikroe_net (change for yours)

Password: mikroe_key (change for yours)

Encryption: WPA2_PSK, the MRF24WB0M will calculate the PSK key (which can take up to 30 seconds).

Hardware:

MCU: PIC18LF46K22 (The PIC18F45K22 came short of RAM for this project)

Oscillator: HS-PLL, 32.00000 MHz 4xPLL, 8.00000 MHz crystal

External modules: wifi_plus_click on mikrobus_2 and microSD click on mikrobus_1 on development board.

A microSD card that can be formatted as FAT16 (up to 2GB)

Put jumper J5 in 3.3V position!

Turn ON PORTA LEDs at SW3.

Turn ON LCD through SW4.5 and SW4.6

Place jumper J21 in RE1 position for buzzer.

Also having a full version of the mikroC Pro for PIC is required to compile, but to test this example the pre-compiled provided HEX file could be used if parameters like the SSID or network password are to be kept.

The mikroC has, on its side, to include the **Network Wifi libraries** from Mikroelektronika, available for free here: http://www.libstock.com/projects/view/356/network-wifi-library.

The program

The first thing to note in the program itself is that I've eliminated the HTML and JavaScript content (that otherwise would have been stored in the PIC program memory) and placed them in the microSD in a gzip format to save program memory that could be used for other purposes in case of a bigger project. The file names are stored and retrieved from two **Arrays of Pointers to Char**:

One to compare upon the files requested by the client web browser:

```
char *fileNames[10] = {"threemin.js", "statsmin.js", "OrbitC.js", "jquery.js",

              "main.js", "PicComm.js", "cube.png", "floor.png",

              "portScript.js", "portScript.js?port="};
```

And a second to retrieve the same gzipped counterpart stored in the microSD card:

```
char *GzipfileNames[9] = {"threemin.gz", "statsmin.gz", "OrbitC.gz",

                "jquery.gz", "main.gz", "PicComm.gz", "cube.png",

                "floor.png", "index.gz"};
```

So when the above files are requested and the program land into the **While loop** of the SendHttpResponse() function, they will be retrieved from the microSD and sent back to the client browser for display and control/monitor operations. This way we achieve a multi file WIFI HTTP server.

Important part of this adaptation of the original **Wifi HTTP demo** is the addition of more headers:

```
const code  char     httpCachecontrol[] = "Cache-Control: public, max-age=28800\n" ;   // cache control

const code  char              httpGzip[] = "Content-Encoding: gzip\n" ;              // Content gzip compressed

const code  char  httpMimeTypeHTML[] = "Content-type: text/html\n\n" ;               // HTML MIME type

const code  char  httpMimeTypeScript[] = "Content-type: text/javascript\n\n" ;       // javaScript MIME type
```

```
const code   char      httpMimeTypePNG[] = "Content-type: image/png\n\n" ;               // png MIME type
```

The cache control was used to speed up the application loading in case of repeated use and the MIME type headers are necessary to tell the browser which file is being sent so it can react appropriately.

In the void Main() a microSD card detection has been established

```
while(Mmc_Card_Detect)          // Loop until card is detected

   ;
```

It will stall the program until the card is inserted as nothing could be done in this project without the files it contains.

Furthermore, using the fat16 library Mmc_Fat_Init() routine, SPI module proper initialization will be checked before allowing the program to go ahead.

Many other parts are basically same as the **Wifi http demo** by Mikroe.

MikroC section references

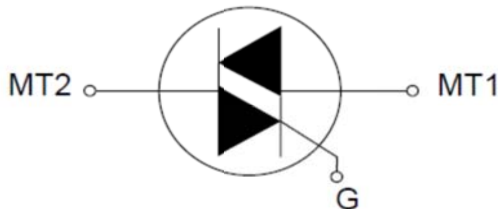
1. "Wifi Http Demo." *LibStock*. Mikroelektronika, 18 June 2012. Web. 28 July 2013.
2. Gan Cuba, Wilson A. "Ajax WIFI HTTP Demo with MicroSD Storage." *LibStock*. Mikroelektronika, 07 May 2013. Web.

Triac control

Warning! The Triac controlled circuit is connected to the mains and could be life threatening. Extreme care must be taken when handling and if you are unsure of the procedure, make you a favor, do not connect this section.

Triac

A Triac is an electronic component that can conduct relative high current in either direction when it is triggered (turned on), allowing it to control very large power flows with milliampere-scale gate currents.



TRIACs can be triggered by either a positive or a negative current applied to its gate. In order to create a triggering current, a positive or negative voltage has to be applied to the gate with respect to the MT1 terminal. Once triggered, the device continues to conduct until the current flowing between MT2 and MT1 drops below a certain threshold, called the holding current, no matter if the gate is completely opened thereafter.

A TRIAC starts conducting when a current flowing into or out of its gate is sufficient to turn on the relevant junctions in the quadrant of operation. The minimum current able to do this is called gate threshold current and is generally indicated by IGT. In a typical TRIAC, the IGT is generally a few milliamperes, but one has to take into account also that:

- IGT depends on the temperature: The higher the temperature, the higher the reverse currents in the blocked junctions. This implies the presence of more free carriers in the gate region, which lowers the gate current needed.
- IGT depends on the quadrant of operation, because a different quadrant implies a different way of triggering. See figure 1.
- When turning on from an off-state, IGT depends on the voltage applied on the two main terminals MT1 and MT2. Higher voltage between MT1 and MT2 cause greater reverse currents in the blocked junctions requiring less gate current similar to high temperature operation. Generally, in datasheets, IGT is given for a specified voltage between MT1 and MT2.

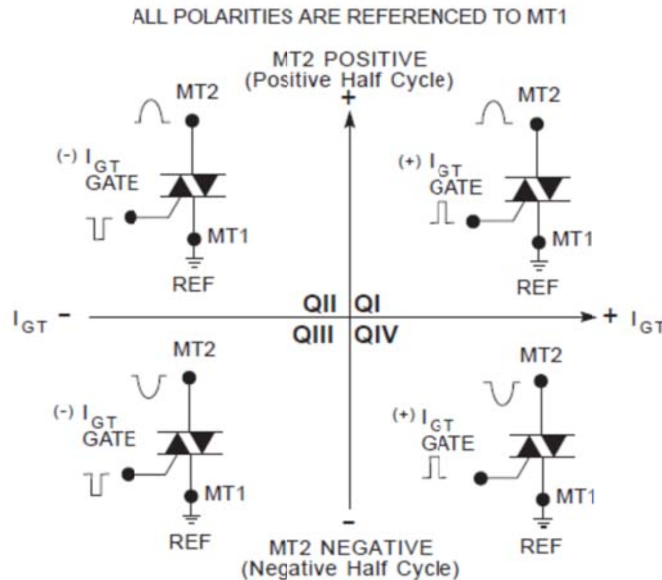


Figure 1.Triac Quadrants of operation

In this project the gate voltage (and current) is in phase with MT2 voltage (and current) (in phase with the mains voltage) so the Triac will function in the first and third quadrant.

Interfacing the PIC micro with the Triac

The specific Triac used for this application is of a sensitive gate with an IGT = 5mA in QI and IGT = 10mA in QIV that does not exceed the 25mA maximum current of the PIC18LF46K22 used in this project, so it can be interfaced directly to the Triac gate, however for life security reasons and to protect the mikroe easyPICv7 board from damaging due to human error, we have used a much safer optoisolator interface approach, which is electrically isolated, as show in figure 2.

The PIC RA0 port will be connected to the pin 1 of the optoisolator which will ultimately control the lamp through the triac.

Optoisolator

The MOC3041M consist of aAlGaAs infrared emitting diode optically coupled to a silicon detector performing the function of a zero voltage crossing bilateral triac driver.

It is designed for use with a Triac in the interface of logic systems to equipment powered from 120 VAC lines, such as teletypewriters, CRTs, solid-state relays, industrial controls, printers, motors, solenoids and consumer appliances, etc.

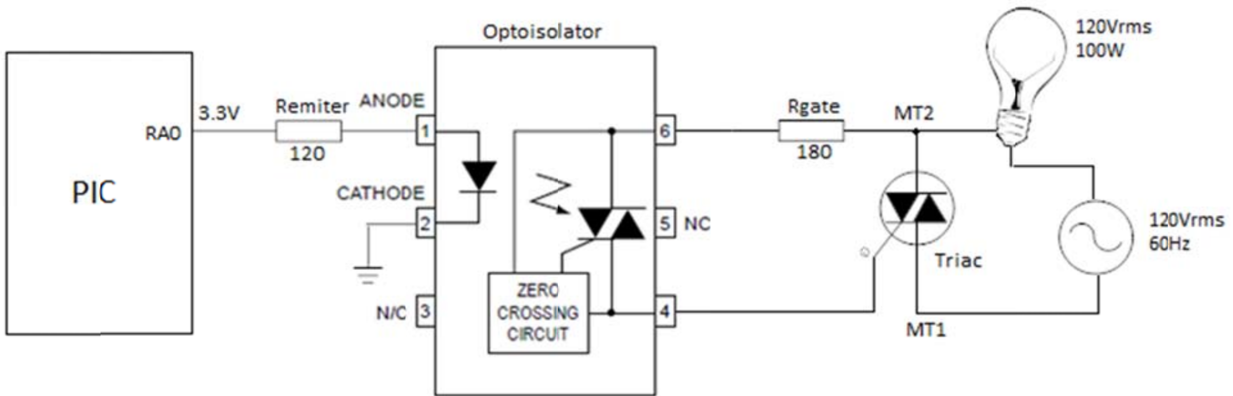


Figure 2. PIC and Triac interfaced through optoisolator.

The above is the typical circuit for use when hot line switching is required. In this circuit the "hot" side of the line is switched and the load connected to it. The load may be connected to either the neutral or hot line.

For this example we have used the Triac L4008L6 and the optoisolator MOC3041 as well as a regular 100W/120V lightbulb.

Even when the Thyristor theory is a huge topic by itself, below are summarized some considerations that should be enough to control the incandescent resistive load.

The optoisolator and Triac where had in stock before beginning the project, so the calculations were made to make sure their specs were sufficient to control the lightbulb, (i.e. I had the devices and adjusted the calculations to make them work as needed) but no doubt that by taking the specific project needs in advance, one could make a more refined (often cheaper) pair opto-triac election, to control just one lightbulb.

MOC3041 specs:

Emiter:

IF = 60mA, Absolute maximum forward current

VF = 1.25, Typical forward voltage

IFT = 15 mA, LED Trigger Current (Main terminal voltage = 3V (see note below) MOC3041)

Note: All devices are guaranteed to trigger at an IF value less than or equal to max IFT. Therefore, recommended operating IF lies between max IFT (15 mA for MOC3041M) and absolute max IF (60 mA).

Detector:

VTM = 1.8V, Typical Peak On-State Voltage, Either Direction (ITM = 100 mA peak, IF = 0)

ITSM = 1 A, Peak Repetitive Surge Current

L4008L6 Triac specs:

IT(rms) = 8A, ON−STATE RMS CURRENT. The maximum value of on−state rms current that can be applied to the device through the two main terminals of a Triac on a continuous basis.

VDRM = 400 V, Repetitive peak off-state/blocking voltage. The maximum peak voltage (note that is peak, not rms) that can be applied to the main terminals of the Triac in non-conductive state.

IGT = QI = 5mA, QIII = 5mA @ TC = 25 °C (case or lead temperature); GATE TRIGGER CURRENT. The maximum value of gate current required to switch the device from the off−state to the on−state under specified conditions. The designer should consider the maximum gate trigger current as the minimum trigger current value that must be applied to the device in order to assure its proper triggering. It is specified by 4 Quadrants

IGTM = 1.6A, PEAK GATE CURRENT (Triac) .The maximum peak gate current which may be safely applied to the device to cause conduction.

VGT = 2V, GATE TRIGGER VOLTAGE The gate dc voltage required to produce the gate trigger current.

ITSM = 80 amps at 60hz, PEAK NON−REPETITIVE SURGE CURRENT The maximum allowable non−repetitive surge current the device will withstand at a specified pulse width, usually specified at 60 Hz.

General calculations

Emiter calculations:

Remiter is calculated so that IF is equal to the rated IFT of the part, which is 15 mA for the MOC3041M.

Remiter = (Vsupply-VF)/IFT The PIC micro source a voltage Vsupply = 3.3V

Remiter = (3.3 – 1.25)/0.015 = 137 ohms,

137 ohms is not a typical value. A lower typical value chosen is 120 ohms. The reason for this election is that a greater value than 137 ohms would produce a current <IFT which is the minimum value recommended.

Detector calculations:

From the Fairchild Application Note AN-3003:

The max surge current rating (ITSM) of the optoisolator sets the minimum value of Rgate through the equation:

Rgate (min) = Vin(pk)/ITSM where Vin(pk) is the mains sine wave peak voltage.

If we are operating on the 120 VAC(rms) nominal line voltage,

Vin(pk) = 120VAC x 1.414 = 170 V, then

Rgate (min) = 170V/1A = 170 ohms.

In practice, this would be a 150 or 180 ohm resistor.

Now, the minimum voltage needed between MT1 and MT2 to turn on the triac is determined by adding up the gate current through the resistor IGT, the triac gate voltage VGT, and the opto on-state output voltage VTM.

Rgate x IGT + VGT + VTM = 180 ohms x 5 mA + 2 V + 1.8 V = 4.7 V.

Therefore, the supply voltage for a 120Vrms circuit must reach 4.7V before the TRIAC will trigger. This is well below the minimum voltage required to get the lights to illuminate (about 70V).

Other considerations:

Current consumed by the lightbulb:

I_{bulb} = P/V = 100W/120Vac(rms) = 0.83A(rms) , it is < 8A(rms) which is the IT(rms) of this specific Triac, so it is under safe current parameters.

Peak voltage of mains supply: As demonstrated above it is 170V < this Triac VDRM = 400 V, which is also under safe parameters.

Triac maximum peak gate current: As the Rgate resistor was calculated by the ITSM of the optoisolator which is 1A < the Triac IGTM = 1.6A is also again under safe parameters.

PIC18LF46K22 maximum sink/source current:

The optoisolatoremiter IF as being = 15mA does not exceed this PIC sink/source maximum current of 25mA.

As more load is added (more lamps for example) and the current by the Triac increases getting closer to the IT(rms) value of the Triac, a heatsink may be required to dissipate the extra heat.

Triac vs relay?

Advantages of the Triac

1. No mechanical wear-out
2. Easier to switch on zero-crossing. (Can also be done with a relay, but less accurate due to the switch-on delay)
3. Can be used in dangerous environment, particularly in explosive sensitive environments where sparking relay contacts are absolutely out
4. No EMI due to switching sparks/arcs (However EMI can still be produced with Triacs)
5. No contacts which can weld
6. Often more compact

Advantages of the relay

1. Can handle DC
2. Can handle any signal: low and high current, low and high frequency, low and high voltage
3. Insulation between control and switched sides
4. No leakage when off
5. Very low voltage drop when on
6. Therefore high currents possible without cooling

Triac control section references

3. Microchip. *X-10 Home Automation Using the PIC16F877A*. N.p.: Microchip, 2002.*Microchip.com*. Web. 27 July 2013.
4. "TRIAC." *Wikipedia*. Wikimedia Foundation, 27 July 2013. Web. 27 July 2013.
5. "Applications of Random Phase Crossing Triac Drivers." *Www.fairchildsemi.com*. N.p., 2003. Web. 27 July 2013.
6. "Thyristor Theory and Design Considerations." *Onsemi.com*. N.p., Nov. 2006. Web. 27 July 2013.
7. Stevenvh. "Triac versus Relay." *Stackexchange.com*. N.p., 21 July 2010. Web. 27 July 2013.

Other considerations

Being the 8-bit PIC18 family so useful, its CPU is really powerless compared with computer microprocessors, having multiple cores and speeds in the order of the gigahertz while the PIC used in this project (PIC18LF46K22) can support up to 64 MHz using 4X Phase Lock Loop (PLL), and a CPU speed of up to 16 MIPS. That said, high file transfer speeds couldn't be expected and this issue affected this project in a way that action had to be taken and some adjustments were required to avoid a complete failure.

The first problem was that (as noted experimentally in the Google Chrome **Developer Tools** and **Firebug** under Mozilla Firefox) when JavaScript external files are linked in the html page **head** section they tend to be requested in parallel independently on how are delivered by the server. Usually this is not a problem because the computer web servers operative systems are capable of multi-tasking and all files will tend to be delivered in parallel and at high speeds. The last assumption doesn't take in account the network latency, but we could say that will be pretty much parallel. The problem lies in that if the browser request the files in parallel but they are delivered serially, will happen, if the files are too long or too many, that the last ones will time-out (especially with the PIC low processing speed) and the browser will not try to retrieve them anymore unless refresh is performed, but refresh will neither work because the process will repeat endlessly.

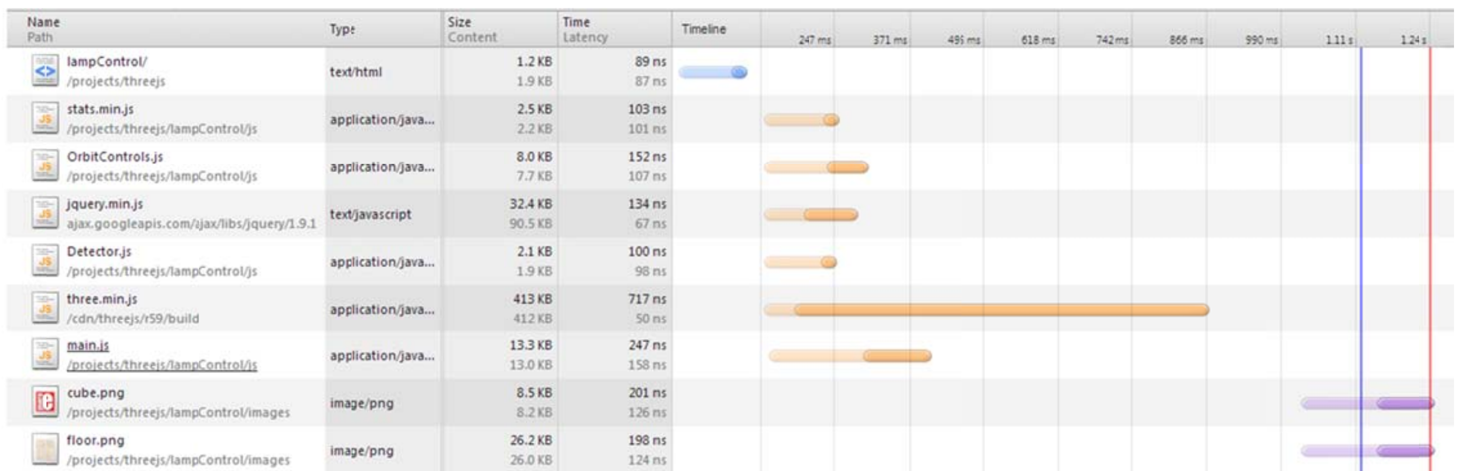


Figure. Pararell request⇔parallel deliver (computer based web server)

The PIC micros have not operating system from factory and as so, no multi-tasking available, even though an RTOS or any logic to simulate it could be used, but this is not the case with this project and the serial deliver nightmare will be produced.

The easier solution I found was to not only make a serial file deliver but also a serial file request by the browser, that way when the first file requested finish loading, the second will be requested and on and

on and no more time-outs even though the web page loading speed is lower than in the [parallel-request]⇔ [parallel-deliver] approach.

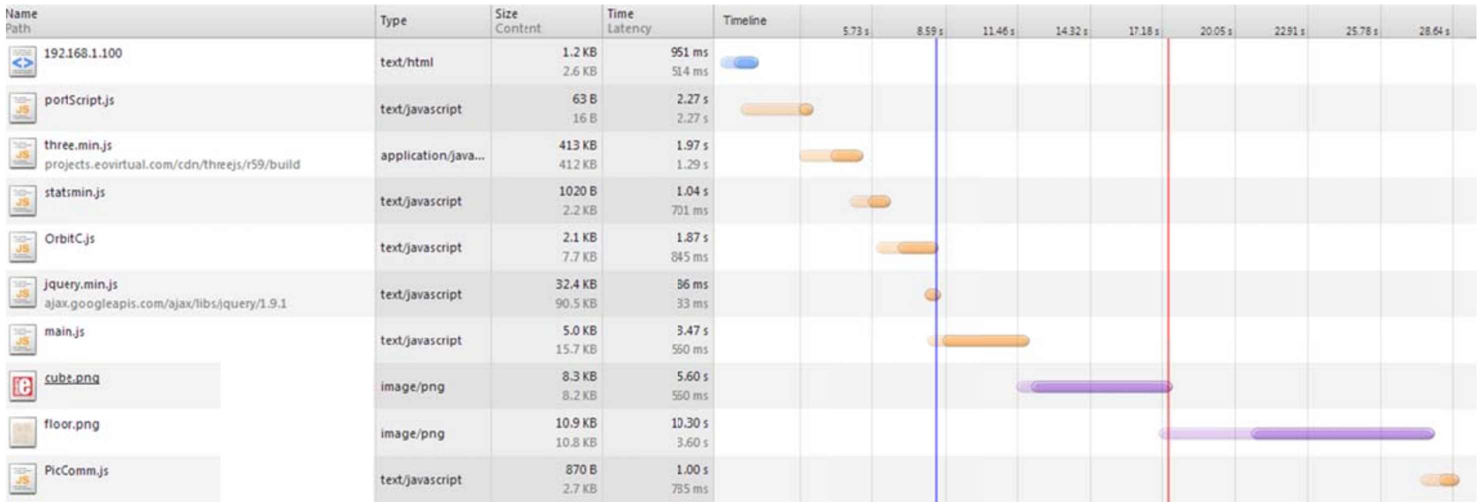


Figure. Serial request⇔serial deliver (PIC based web server)

The serialization was in part achieved by using **document.write()** (write method) to load the JavaScript external files:

```
<script> document.write("<script src='portScript.js'><\/script>");                              </script>

<!-- Grab three.js from internet. Fall back to local if necessary

Comment - Uncomment the line below to test with or without internet connection  -->

<script> document.write("<script src='//projects.eovirtual.com/cdn/threejs/r59/build/three.min.js'><\/script>");
</script>

<script>!window.THREE && document.write("<script src='threemin.js'><\/script>");     </script>

<script> document.write("<script src='statsmin.js'><\/script>");                              </script>

<script> document.write("<script src='OrbitC.js'><\/script>");                              </script>

Etcetera…
```

Note in the code above that I've created a sort of CDN (that you could use) where I've made available online the minified three.js released, so that it is actually loaded from internet for speed purposes, but if no internet is available on the client browser, fall back to retrieve it from the microSD is provided.

Now, for the images I used in the Three.js application, the document.write() approach did not work, so I did the following:

```
var CubeimgTexture = THREE.ImageUtils.loadTexture('cube.png', new THREE.UVMapping(), function() { }//Call back
function used for synchronous
```

Here the loadTexture property from ImageUtils can accept a callback function that will fire when the image texture has been loaded and that's what I used to continue serializing the file loading process.

Performance improvement

Above the browser requests were serialized to match the PIC behavior, but this is incompatible with loading performance and speed, fortunately there are some steps that can contribute to improve this deficiency.

- First thing is compress the files in a format understood by the browsers, the winner is gzip, supported by all mayor browsers. In the mikroC program section the **Arrays of Pointers to Char ***GzipfileNames[] was used to achieve this, plus every single external file was manually gzipped with the free compressor 7-Zip.
- A cache header is send by the server requesting the browser to enable cache for files that never change, that way the first loading after the cache has expired may be slow but all other times it will happen instantaneously as no real server request is made but a retrieval of locally stored files.
- As mentioned above, the Three.js library, but also the jQuery, are first tried to be retrieved from internet and if this fails then use the microSD stored version which will take longer to load. In fact all static content could be placed on internet as first source.
- Minify all JavaScript files to decrease their size. I haven't done this here as this makes the code unreadable, however I've used the Three.js and jQuery already minified versions.
- Optimize images. Decreasing the image size will surely speed up the loading.

For more on this visit: http://developer.yahoo.com/performance/rules.html

Last words

However this is a good first step, is not the end. There are many clear improvements that could be done like those we discussed in the **PIC communication program** section, as for example implement automatic synchronization if another user in another computer change the lamp status, detect server or network down as it is possible with the mentioned **Ajax Wifi http demo with microSD storage**, develop some kind of access control, because nobody wants to have its equipment publicly available on internet, make a more sophisticated WebGL design that could even include more realistic imported meshes. Imagination has no limits.

Note that English is not my native language; if you find any redaction error, do not hesitate to contact me at kvantumax@gmail.com

I hope you enjoy this project as I did, good luck!

License

WIFI WebGL (three.js) Lamp control with PIC micro by Wilson A. Gan Cuba is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.