Universal Internet Connected Remote Control



This article discusses an IoT project that allows its users to send IR signals from any computer within a network. For that purpose, an ESP8266-enabled development board is connected to an IR sender module, and the finished device hosts a website that allows users to send out codes. The device can be useful, for example, to remote-control several devices from within a network without having to replace the existing electronic equipment with connected items.

### BOM

The following parts can be used to build this project:

* [940nm IR LED](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [ESP8266 based development board](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [68 Ohm resistor](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [NPN transistor](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [330 Ohm resistor](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [CR2032 battery holder](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [CR2032 battery](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [Perfboard (optional)](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [Breadboard (optional)](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)
* [M/F Jumper Wires (optional)](https://www.digikey.in/en/maker/blogs/2021/how-to-build-a-universal-internet-connected-remote-control)

Note that these are recommended parts. It is, however, possible to use different parts if a project requires a more specific IR wavelength to function, or if you’d like to employ a different MCU development board.

### The IR Sender Circuit with an ESP8266



This project doesn’t require a complicated circuit as the software running on the ESP8266 does most of the work.

The above diagram shows a [simple circuit](https://www.digikey.at/schemeit/project/nodemcu-ir-led-fb05e1b6e1344fe39a9e41dd84ae803e/) that allows the ESP8266 to control an infrared LED. Note that it would be possible to connect the LED directly to the development board. That approach, however, isn’t recommended. Instead, one common tactic is to use current-limiting resistors to avoid damaging the components.

This project uses an added NPN transistor to the circuit because IR LEDs might require more current to function properly. A development board might not be able to supply enough current for the LED to reach its maximum brightness, which limits the effective range of the device. To avoid that, the circuit includes a CR2032 battery that powers the LED, and the transistor acts as a switch that can be turned on or off by the ESP8266.

### Arduino’s IDE Library and the Software for the Project

Now to the more complicated part of the project: the software. As in the previous parts, the software of this project uses the IRremote library for Arduino. For the ESP8266, however, there’s a version that programmers can install with the Arduino IDE’s library manager:



And here’s the code for the entire project (an explanation of the program code is given below):

#**include** <IRremoteESP8266.h>

#**include** <IRsend.h>

#**include** <ESP8266WiFi.h>

#**define** LED\_PIN 5

#**define** STORAGE\_SIZE 5

**const** **char**\* ssid = "YOUR\_NETWORK\_NAME";

**const** **char**\* pass = "NETWORK\_PASSWORD";

WiFiServer **server**(80);

IRsend **sender**(LED\_PIN);

**long** codes[STORAGE\_SIZE] = {0x00123,0x02,0xAB,0x0F,0x0};

**bool** stored[STORAGE\_SIZE] = {true,false,false,false,false};

**int** protocols[STORAGE\_SIZE] = {0,1,1,0,1};

// Function Prototypes

**int** **containsValidIndex**(String);

String **findParameterValue**(String, String);

**void** **setup**()

{

 Serial.begin(9600);

 **while**(!Serial)

 delay(50);

 sender.begin();

 Serial.println("");

 Serial.print("Connecting to ");

 Serial.print(ssid);

 WiFi.begin(ssid, pass);

 **while**(WiFi.status() != WL\_CONNECTED)

 {

 Serial.print(".");

 delay(500);

 }

 Serial.println("Done!");

 Serial.println("Starting server...");

 server.begin();

 Serial.print("Server started with address ");

 Serial.print("http://");

 Serial.print(WiFi.localIP());

 Serial.println("/");

}

**void** **loop**()

{

 // Check for incoming connections

 WiFiClient c = server.available();

 **if**(c)

 {

 **while**(!c.available())

 delay(10);

 // Read the first line of the HTTP request

 // It contains something similar to the following line:

 // METHOD /requested\_url HTTP\_VERSION

 // for example:

 // GET /send?value=0x0085&protocol=NEC HTTP/1.1

 // However, for the sake of simplicity this device only accepts

 // GET requests as they can be sent with any web browser.

 // Updating values this way is not the 'correct' way according to

 // the HTTP standard but it makes using the device easier.

 String request = c.readStringUntil('\r');

 c.flush();

 **int** error = 0;

 **int** value = request.indexOf("value=");

 **int** protocol = request.indexOf("protocol=");

 **int** index = request.indexOf("index=");

 **int** snd = request.indexOf("/send");

 **int** str = request.indexOf("/store");

 **int** del = request.indexOf("/delete");

 Serial.println(request);

 // The following few if/else-statements parse the incoming request

 // and then execute the action that the user requested.

 // The user requested the send page and did not include /store or /delete in their request

 **if** (snd != -1 && str == -1 && del == -1)

 {

 // The user requested the send page and included a value for the index

 **if**(index != -1)

 {

 // Check if the supplied index is valid (i.e. at least zero and less than STORAGE\_SIZE)

 // And check if the values array contains an entry at the requested position

 // First, get the value of the parameter as a string

 String index\_string = findParameterValue(request, "index");

 // Next, convert it to an integer and check whether the value is valid

 **int** i = containsValidIndex(index\_string);

 **if**(i > -1 && stored[i])

 {

 // The parameter was valid. Repeat the stored value!

 // Make sure to verify the bit length for your remotes!

 // I used 32 and 14 in this program, but yours might vary

 **if**(protocols[i] == 0)

 sender.sendNEC(codes[i], 32);

 **else**

 sender.sendRC5(codes[i], 14);

 Serial.print("Repeat the value stored at position ");

 Serial.println(i);

 }

 }

 // The user supplied the wrong parameters for this request.

 **else**

 {

 error = 1;

 Serial.print("Unknown request: ");

 Serial.println(request);

 }

 }

 // The user requested the /store page and the request didn't include

 // the /send or /delete page

 **else** **if**(str != -1 && snd == -1 && del == -1)

 {

 // The user supplied the wrong parameters for this request.

 // (Either of the three parameters is missing)

 **if**(index == -1 || protocol == -1 || value == -1)

 {

 error = 1;

 Serial.print("BAD REQUEST. Missing field: ");

 Serial.println(request);

 }

 **else**

 {

 String p = findParameterValue(request, "protocol");

 String v = findParameterValue(request, "value");

 **int** i = containsValidIndex(findParameterValue(request, "index"));

 // Check if the supplied index is valid

 **if**(i > -1)

 {

 Serial.print("Store ");

 Serial.print(v);

 Serial.print(" for protocol ");

 Serial.print(p);

 Serial.print(" on position ");

 Serial.println(i);

 codes[i] = strtol(v.c\_str(), NULL, 16);

 protocols[i] = (p == "NEC") ? 0 : 1;

 stored[i] = true;

 }

 }

 }

 // The user requested the /delete page and the request didn't include

 // the /send or /delete page

 **else** **if**(del != -1 && str == -1 && snd == -1)

 {

 // The /delete page requires the index to work. It ignores the other parameters.

 // So if the request didn't include the index, then answer with an error

 **if**(index == -1)

 {

 error = 1;

 Serial.print("BAD REQUEST. Missing field: ");

 Serial.println(request);

 }

 **else**

 {

 **int** i = containsValidIndex(findParameterValue(request, "index"));

 **if**(i > -1 && stored[i])

 {

 stored[i] = false;

 Serial.print("Delete the value stored on position ");

 Serial.println(i);

 }

 }

 }

 // Return the response

 // If no error occurred, send an HTML page that lists the stored codes

 // and that contains two forms for managing them (add new ones and delete existing codes).

 **if**(error == 0)

 {

 /[\* HTML code omitted! \*/](https://drive.google.com/file/d/1vRBnl22yDNS6Ikdl1ZPPHkncYnn6ZZ3F/view)

 }

 **else**

 {

 // Return a response header

 /\* HTML error response omitted! \*/

 }

 }

}

**int** containsValidIndex(String index\_string)

{

 // First check whether the string only contains digits

 **for**(**int** i = 0; i < index\_string.length(); i++)

 {

 **if**(!isDigit(index\_string.charAt(i)))

 **return** -1;

 }

 // Then convert the string to an int and repeat the stored valud at that

 // position in the array (if the index is valid)

 **int** index\_num = index\_string.toInt();

 **if**(index\_num >= 0 && index\_num < STORAGE\_SIZE)

 **return** index\_num;

 **else**

 **return** -1;

}

String **findParameterValue**(String request, String param\_name)

{

 **int** param\_name\_length = param\_name.length()+1;

 // First, dispose of everything in the request that comes prior to the index parameter

 String tmp = request.substring(request.indexOf(param\_name));

 // Then check whether the remaining part of the request (everything that comes after

 // the request=... part of the URL) contains more paramters (they are separated by

 // ampersand characters (&)).

 **if**(tmp.indexOf("&") != -1)

 // If there are more parameters, the value of the index sits between the

 // parameter name (index=) and the next ampersand (&).

 // For example: URL?index=5&other\_param=24 (...)

 **return** tmp.substring(param\_name\_length, tmp.indexOf("&"));

 **else**

 // Otherwise, the value of interest comes right before the next

 // whitespace character like this:

 // URL?index=5 HTTP(...)

 **return** tmp.substring(param\_name\_length, tmp.indexOf(" "));

}

### Breaking Down the Code — What’s Happening Here, Exactly?

The previous articles made use of the [IRremote library for Arduino](https://github.com/Arduino-IRremote/Arduino-IRremote%22%20%5Ct%20%22_blank). Luckily, a ported version for ESP8266-based development boards exists. There are, however, a few minor differences between the versions. The ESP8266 port requires programmers to include different header files. Furthermore, it’s necessary to initialize the sender class. This happens in the setup method of the code for this project (right after the while loop that waits for the serial port to open).

Right after that, the setup function prints a few logging messages before it tries to establish a connection to a local wireless network. The SSID and password reside in two variables above the setup method. Once a connection is established, the setup method starts an HTTP server.

The server is defined at the top of the sketch. It listens to incoming requests on port 80. Below that are three arrays that store information about the codes a user entered on the website that the development board hosts.

The loop method periodically checks for a client waiting to establish a connection to the web server. If there is a client, the ESP8266 retrieves the most recent request and parses it. In this example, we’re only dealing with the first line of simple HTTP requests. The loop method attempts to parse each incoming request. If the request contains ‘/send’, then the device knows that the user requested to send one of the previously stored IR codes. For that, the request must also include the index of the code to send. If the request contains these parts, the program further analyzes the value of the index parameter. It must be between zero and the length of the arrays. The arrays must also contain a valid value at the requested position.

The ‘containsValidIndex’ function towards the end of the sketch does the parsing of parameters. This function takes a string that represents a number. The method inspects every character of the string, and if it finds a non-numeric character, it returns a negative number. The function then converts the string to an integer and returns it if the integer represents a valid position in any of the arrays.

Another helper function resides below the ‘containsValidIndex’ method, called ‘findParameterValue’. This function scans the request for a given parameter name. Note that this function assumes the request contains the parameter in question. It then returns the value of the parameter as a string.

These two helper methods together allow the program to extract parameter values from the user’s requests. Back in the loop method, the next part of the if/else blocks checks whether the user requested the store function. This functionality allows users to input new codes that the ESP8266 can store as long as it’s powered. This block, again, checks whether the request contains the required parameters and then extracts the parameter values with the previously discussed helper functions.

The last ‘if/else’ block checks whether the user wants to delete a previously stored value. It then parses the request in the same ways as previously described. If the supplied value for the index parameter is valid, this part of the loop method deletes a stored code at the given position by setting the ‘stored’ array to false at that index.

Once all the parsing is done, the loop method returns an HTML page that contains a table with the stored codes and two forms that allow the users to manage the previously recorded codes. Note that the code above omits that HTML part to save some space.



Note that you can easily add other parameters, such as a name for each IR code like ‘Living room TV OFF’ and include more sophisticated input checks to increase the security and functionality of the program. It’s also possible to support more than two protocols. Consult the [official documentation of the IRremote library](https://github.com/Arduino-IRremote/Arduino-IRremote/wiki/IRremote-library-API) for further details.

### Summary

This simple project allows you to administer almost any device that employs an IR remote control from a web browser. This technique can be a cost-effective way to connect your existing devices to a home network and make them more practical. The circuit for this project is not complicated. It consists of an ESP8266-based development board, a transistor, an IR LED, an external power source, and a couple of resistors. The software that runs on the ESP8266 allows users to visit a website as long as they are connected to their home network. On that website, they can enter IR remote controller codes and instruct the ESP8266 to repeat them. Doing so allows them to control multiple of their existing devices without having to change remotes.