Explicación código ESP32.

Declaración de elementos:

**1 declaración de librerías.**

#include <WiFi.h>

#include <string.h>

#include <MD\_Parola.h>

#include <MD\_MAX72xx.h>

#include <Adafruit\_NeoPixel.h>

#include <Preferences.h>

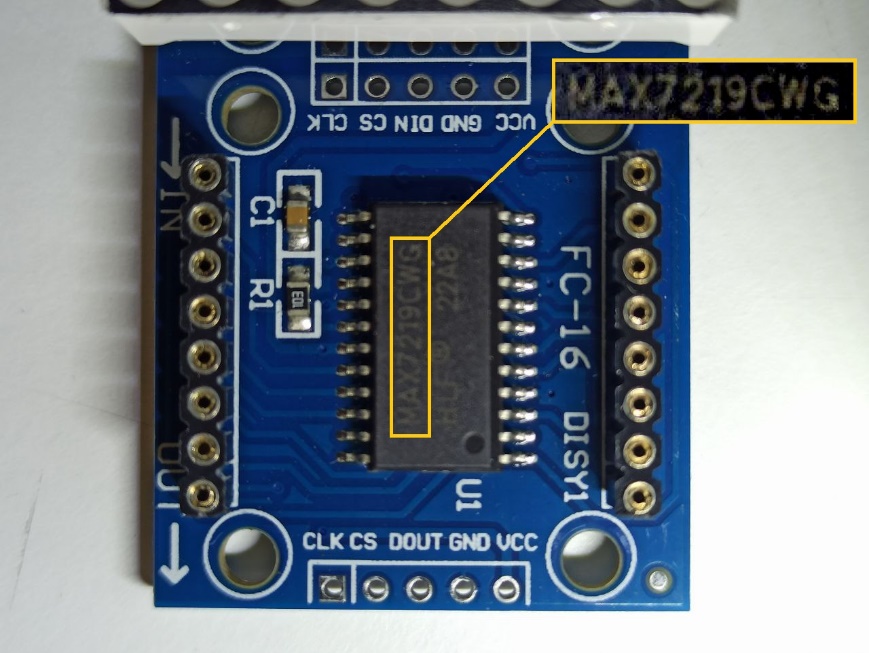
Por orden:

* Librería principal para la creación de objetos WiFiServer, WiFiClient, y para la conexión por wifi.
* Librería para el manejo más en profundidad de strings.
* Librería principal para el control del display.
* Librería para el control del display a la hora de desplegar íconos.
* Librería para el control de los leds de modelo WS2812, comúnmente conocidos como neopixel. Necesarios para la funcionalidad del conteo regresivo binario.

1. **Definición de constantes.**

#define HARDWARE\_TYPE MD\_MAX72XX::FC16\_HW

Declaración del tipo de hardware del display. FC16\_HW es el modelo del display, MAX7219CWG es la referencia del integrado de control del mismo.



**3 declaración de constantes.**

const int SECOND = 1000;

const int ARRAY\_SIZE = 4;

const int NO\_INDEX\_FOUND = -1;

const int RELAY = 21;

const int STATE\_LED = 17;

const int RESET\_BUTTON = 18;

const int KEY = 16;

const int TAMPER = 22;

const int BINARY\_PIN = 5;

const int NUMPIXELS = 13;

const int MAX\_DEVICES = 5;

const int DATA\_PIN = 15;

const int CS\_PIN = 2;

const int CLK\_PIN = 4;

// const int TIME\_COUNTER\_DEFAULT = 6539;

const int TIME\_COUNTER\_DEFAULT = 69;

const int FIRST\_STAGE = 300;

const int SECOND\_STAGE = 60;

const int THIRD\_STAGE = 10;

const int END\_STAGE = 0;

const int ALARM\_MODIFIER\_DEFAULT = 3;

* **SECOND** Declaración del tiempo en milisegundos de un segundo.
* **ARRAY\_SIZE** Declaración del tamaño del array de octetos de la ip. El tamaño es de 4 porque una ip está compuesta de 4 octetos.
* **NO\_INDEX\_FOUND** Declaración del valor obtenido en caso de consultar si en un string existe otro string, dado como argumento del método indexOf() del primer String. Al lanzar dicho método se devolverá un int indicando la posición del segundo string en el primer string, o se devolverá el valor -1.
* **RELAY** declaración del pin de salida encargado de activar el transistor que cerrará el circuito para la gestión del circuito de 230V.
* **STATE\_LED** declaración del pin de salida para el led encargado de mostrar cuando el sistema se encuentra sin conexión.
* **RESET\_BUTTON** declaración del pin de entrada para el botón encargado de la función de reinicio de fábrica de los parámetros de red.
* **KEY** declaración del pin de entrada del switch, del tipo llave, encargado de la función standby.
* **TAMPER** declaración del pin de entrada para el switch del tamper.
* **BINARY\_PIN** Declaración del pin al que se conecta el pin de datos de la tira de leds WS2812.
* **NUMPIXELS** declaración del número de leds en la tira de leds.
* **MAX\_DEVICES** número máximo de módulos de segmentos del display. Se entiende como segmento de display cada módulo de 8x8 pixeles.
* **DATA\_PIN** declaración de la entrada 4 para el pin CLK\_PIN del display. El pin DIN (Data In) del módulo de display FC-16 se utiliza para enviar datos al controlador MAX7219, que a su vez controla la matriz de LEDs en el módulo.
* **CS\_PIN** declaración de la entrada 2 para el pin CS\_PIN del display. Este pin se utiliza para seleccionar el controlador MAX7219 cuando se envían datos desde el microcontrolador. Cuando CS está en un estado bajo (normalmente LOW), el controlador MAX7219 está activo y listo para recibir datos. Cuando CS está en un estado alto (normalmente HIGH), el controlador MAX7219 se desactiva y no responde a los datos enviados desde el microcontrolador.
* **CLK\_PIN** declaración de la entrada 15 para el pin DATA\_PIN del display. Este pin se utiliza para sincronizar la transmisión de datos entre el microcontrolador y el controlador MAX7219. Cada vez que se envía un bit de datos (ya sea para configurar un registro interno o para controlar los LEDs), se envía un pulso de reloj en este pin para indicar al controlador que se ha enviado un nuevo bit.
* **TIME\_COUNTER\_DEFAULT** valor por defecto de la cuenta total de los segundos que hay en 108 minutos. Esta es la variable que se usará para resetear el contador de tiempo en segundos cuando se haya introducido la contraseña de forma correcta.
* **FIRST\_STAGE** declaración del primer escenario en la cuenta regresiva. Cuando timeCounter llegue al valor de esta variable se entrará en el primer escenario. Durante el primer escenario, la cuenta regresiva de tiempo en el display se mostrará con segundos incluidos, la alarma sonora comenzará y se abrirá la opción al usuario para poder ingresar la contraseña. Esta etapa abarca del minuto 4 al 1, se produce la primera señal sonora cada 2 segundos.
* **SECOND\_STAGE** declaración del segundo escenario en la cuenta regresiva. Cuando timeCounter llegue al valor de esta variable se entrará en el segundo escenario. Durante el segundo escenario la señal sonora cambiará. Esta etapa abarca del minuto 1 al segundo 11 incluido, se produce la segunda señal sonora cada 3 segundos, es decir, cada dos periodos en silencio.
* **THIRD\_STAGE** declaración del tercer escenario en la cuenta regresiva. Cuando timeCounter llegue al valor de esta variable se entrará en el tercer escenario.

Durante el tercer escenario la frecuencia a la que se produce la señal sonora aumenta. Esta etapa abarca del minuto 10 al segundo 0 incluido, se produce la segunda señal sonora cada 1 segundos.

* **END\_STAGE** declaración del último escenario en la cuenta regresiva. Cuando timeCounter llegue al valor de esta variable se entrará en el último escenario.

Llegado el último escenario, si no se ha introducido la contraseña de forma correcta, cesará toda actividad. En esta situación se iniciará el proceso gráfico y la activación del relé de 230.

* **ALARM\_MODIFIER\_DEFAULT** el valor de esta variable es el que se usará para establecer la periocidad de la señal sonora cada 3 segundos.

**4 declaración de elementos int.**

int alarmModifier = ALARM\_MODIFIER\_DEFAULT;

int timeCounter = TIME\_COUNTER\_DEFAULT;

int previousCounter = timeCounter;

int currentTime = 0;

int previousTime = 0;

int missConnectionCounter = 0;

int connectionLost = 0;

* **alarmModifier** establece el tiempo, en segundos, que ha de transcurrir entre cada señal sonora. El valor por defecto, ALARM\_MODIFIER\_DEFAULT, será de 3 segundos.
* **timeCounter** es la variable encargada de contar el tiempo restante en segundos. El usuario ha de realizar una serie de acciones durante los cuatro últimos minutos de un total de 108, timeCounter recogerá la cuenta en segundos de dicho tiempo. Se le restará una unidad al contador cada segundo. En función del valor de
* **previousCounter** variable encargada de representar un instante previo. Esta variable es necesaria para poder tener un tiempo de referencia con el que comparar cada instante. Esto permitirá saber si ha pasado un tiempo igual o mayor al valor de alarmModifier. Según lo explicado anteriormente, alarmModifier establecerá cada cuanto tiempo se ha de reproducir la señal sonora. Restandole a previousCounter, es decir, restándole a un instante previo, el tiempo restante actual, timeCounter, obtendremos un valor que, de ser superior al valor de alarmModifier, reportará un resultado true en un condicional que será el que realice la señal sonora. Ejemplo:

if ((previousCounter - timeCounter) >= alarmModifier){}

((10 – 7 = 3) >= 3 ) = True

En el momento de la declaración el valor de previousTime será igual al de timeCounter porque no ha existido ningún momento previo.

* **currentTime** tendrá un comportamiento y una finalidad similar a la de timeCounter. currentTime se encargará de representar un valor actual en el tiempo. Esta variable trabaja junto a previousTime, que se explica a continuación.
* **previousTime** representa un instante previo en el tiempo. Esta variable trabaja en conjunto con currentTime. Ambas variables se encargan de establecer el acceso al condicional principal del sistema, cuando se trabaja junto a Python, cada un segundo. Esto se logra tomando medidas de tiempo mediante la función millis() con la variable currentTime en diferentes partes del código. Llegada la ejecución del programa al condicional principal, su condición será la siguiente:

((currentTime - previousTime) >= SECOND)

Restándole al instante de tiempo actual, el instante de tiempo previo, podremos saber si el tiempo transcurrido será superior o igual al valor de SECOND, que como ya hemos visto anteriormente, tendrá el valor de 1000 milisegundos, es decir, un segundo.

Recordemos que, según la documentación oficial, la función millis() devuelve el número de milisegundos transcurridos desde que la placa inició el programa actual. Pasados aproximadamente 50 días la cuenta será restaurada a 0 y volverá a iniciarse. Es por esto que cualquier instante de tiempo actual tendrá un valor mayor que cualquier instante previo. Por ello, se iterará sobre el anterior condicional hasta que el resultado de la resta sea mayor o igual a 1 segundo. Llegados a este escenario se resetearán ambas variables.

(currentTime - previousTime) >= SECOND)

(150 millis – 100 millis >= 1000 millis) == False

(401 millis – 100 millis >= 1000 millis) == False

(823 millis – 100 millis >= 1000 millis) == False

(1105 millis – 100 millis >= 1000 millis) == TRUE

* **missConnectionCounter** esta variable se encarga del conteo de periodos de no comunicación con el cliente. Durante el proceso de comunicación con Python la comunicación se basará en el envío de un mensaje a Python y la recepción de un mensaje desde Python, cada un segundo. Esta variable aumentará en uno su valor cuando no se reciba el mensaje de Python. Se interpretará como conexión perdida cuando esta variable llegue al máximo número de mensajes no recibidos por parte de Python. En este escenario se ejecutará un bloque de instrucciones encargadas de manejar el intento de reconexión.
* **connectionLost** El valor de esta variable se actualizará cada vez que se pierda la conexión. Servirá únicamente para llevar un conteo del número de ocasiones en las que se perdió la conexión.

**5 declaración de elementos bool.**

bool exitLoop = false;

bool failureState = true;

bool resetValues = true;

* **exitLoop** Variable encargada de controlar el bucle while principal, el búcle que se inicia tras conectarse un cliente al servidor.
* **failureState** Declaración de la variable que se usará para activar o no el relé de control de 230V una vez que se llame al método systemFailureOn. Si el valor de esta variable es true, el relé de activación de 230V se activará. Si su valor es false cuando se llame al método systemFailureOn se desplegarán solo los gráficos, pero no se activará el relé.
* **resetValues** Esta variable servirá para tener una referencia de en qué estado se encuentra el sistema. Cuando se pierde la sincronía entre Python y el ESP32, esta variable cambia su valor inicial de true a false. Esto permite que se mantengan los valores de las variables relacionadas con el flujo del tiempo y con el control del contador de tiempo regresivo cuando se recupera dicha sincronía. Esto quiere decir que, cuando se recupera la sincronía tras una perdida, gracias a que el valor de esta variable es false en ese especifico momento, al detectarse un nuevo cliente, en lugar de resetear todos los valores como pasaría cuando se inicia una nueva conexión, los mantendrá. Así mismo, esta variable permitirá controlar el flujo de ejecución cuando se está intentando recuperar la sincronía en un escenario en el que la placa ESP32 tiene conexión, es decir, cuando la perdida de conexión la tiene Python. Todo esto se explicará en detalle en los métodos initiativeReset y connectionStateLed

**6 declaración de elementos String.**

String logText = "";

String octets[ARRAY\_SIZE];

String pSsid;

String pNetworkPassword;

String ipChangePassword = "admin";

String stringHelp;

* **logText** será la variable que almacene la concatenación de mensajes que deberán ser impresos en el campo de registro de log en el webservice. Cada vez que se realice alguna acción por medio de la interfaz web, esta quedará registrada en el log mostrado en la misma interfaz. Se mostrarán datos de interés de la acción realizada.
* **octets** será el array de números, en formato string, que forme una nueva ip cuando se solicite un cambio de los parámetros de red. Se utiliza un array de elementos string porque la nueva ip se recoge desde la interfaz web, como un string con el formato de una ip, que posteriormente se divide en los 4 números sin los puntos que separan cada número de la ip. Esto se debe a que la función encargada de establecer los parámetros de red según la necesidad del usuario será el método config del objeto Wifi, WiFi.config(staticIP, gateway, subnet). Los parámetros de este método son objetos IPAddress cuyos parámetros son números. Por ello obtenemos el string, del string obtenemos cada número de forma individual, y le pasamos estos números convertidos de strings a objetos int como parámetros a cada objeto IPAddress.

  IPAddress staticIP(octets[0].toInt(), octets[1].toInt(),

                     octets[2].toInt(), octets[3].toInt());

  IPAddress gateway(192, 168, 17, 254);

  IPAddress subnet(255, 255, 0, 0);

  WiFi.config(staticIP, gateway, subnet);

* **pSsid** y **pNetworkPassword** serán los strings del nombre y contraseña de la red a la que se va a conectar el dispositivo. La *p* hace referencia al termino preferences.
* **ipChangePassword** es la declaración de la contraseña necesaria para poder cambiar la ip del dispositivo por medio del webservice. Esta función no pretende hacer las veces de sistema de seguridad máxima, es solo un acercamiento por diversión.
* **stringHelp** constituye el texto a imprimir en caso de que se solicite ayuda desde el webservice.

**7 creación de los objetos necesarios para trabajar con el display y con la tira de led de cuenta atrás binaria.**

MD\_Parola parola = MD\_Parola(HARDWARE\_TYPE, DATA\_PIN, CLK\_PIN, CS\_PIN, MAX\_DEVICES);

char character[2];

textPosition\_t positions[] = {

  PA\_LEFT, PA\_LEFT, PA\_LEFT, PA\_RIGHT, PA\_RIGHT

};

MD\_MAX72XX mx = MD\_MAX72XX(HARDWARE\_TYPE, DATA\_PIN, CLK\_PIN, CS\_PIN, MAX\_DEVICES);

Adafruit\_NeoPixel pixels = Adafruit\_NeoPixel(NUMPIXELS, BINARY\_PIN, NEO\_GRB + NEO\_KHZ800);

En los tres casos, los objetos requerirán de las constantes definidas anteriormente para indicar los pines del esp32 a usar para recibir la información específica de cada uno de los pines de los dispositivos, así como el número de módulos del display, y el hardware del mismo.

Es posible que usar dos clases diferentes para el control de un mismo elemento sea una mala práctica, no obstante, el tiempo apremia durante el desarrollo del proyecto y la optimización de esta parte, así como el estudio de ambas librerías, no era la prioridad. Parola se usará para imprimir objetos string en el display, mientras que mx se usará para imprimir gráficos. Parola brinda una forma más rápida de imprimir un string que mx, mientras que mx permite imprimir gráficos.

No se incluye la librería MD\_MAX72xx porque la librería MD\_Parola ya la incluye internamente. Al instalar la librería MD\_Parola por medio del administrador de librerías, la librería MD\_MAX72xx se instala automáticamente como una dependencia. Es por ello que, aun sin incluir dicha librería, podemos acceder a sus definiciones y funciones, porque el compilador de Arduino ya tiene acceso a la la librería. Podemos comprobar esto inspeccionando el archivo MD\_Parola.h con un editor de texto cualquiera, por ejemplo con el bloc de notas de windows. Ya al inicio de la librería se empieza enunciando lo siguient:

*“The Parola library is implemented to work with the MD\_MAX2XX library. It  
depends on the MD\_MAX72xx library for hardware control and will run on all  
hardware supported by that library.”*

*La librería Parola está implementada para funcionar con la librería MD\_MAX2xx. Depende de la librería MD\_MAX2xx para el control del hardware y ejecutará todo el hardware asistida por esa librería.”*

Si seguimos inspeccionando la librería, después de un gran bloque de indicaciones y apuntes, encontramos la invocación a la misma.

El contador regresivo que aparece en la serie distribuye cada número agrupándolos por minutos y segundos. Los minutos están más juntos entre sí y ubicados a la izquierda; por otra parte, los segundos están ubicados a la derecha, dejando un espacio más obvio entre estos conjuntos que el espacio que existe entre cada número. Para lograr esto se establecen los tres primeros displays, los display de los minutos, con una orientación de lo impreso de tipo PA\_LEFT para que se impriman pegados al borde izquierdo de cada display, y los 2 últimos, los display de los segundos, con una orientación de tipo PA\_RIGHT para que se impriman pegados al borde derecho de cada display. Esto genera una separación pronunciada entre el tercer y cuarto display.

La orientación de cada display se logrará asignándole un valor de tipo textPosition\_t al apartado de alineación de cada display, por medio de la variable i de un bucle for y el array positions[]. Más adelante en el código, cuando se proceda a la configuración de cada zona por medio del método displayZoneText(), se le asignará a cada zona i un valor positions[i] para la alineación del mismo. Como se observa en el array, los tres primeros valores son de tipo PA\_LEFT, y los dos últimos son de tipo PA\_RIGHT.

Char character[2] es el array de caracteres que se usará para facilitarle un puntero a un char a la función displayZoneText.

|  |  |
| --- | --- |
| |  | | --- | | **Nota 10:**  Recordemos que un array en C y C++ decae en un puntero al primer elemento del array de caracteres, es decir, puede ser usado como un pointer al primer elemento del array. |   Al usar la función displayZoneText necesitaremos un puntero a un char para imprimirlo de forma individual en cada display. El array character se encargará de albergar el carácter correspondiente en su posición [0], y el terminador null en la posición [1] para indicarle al compilador dónde termina la cadena de caracteres. El terminador null no se imprimirá en el display por lo citado anteriomente, el array de elementos char será tratado como un puntero al primer elemento del array únicamente. |

**8 creación del objeto preferences.**

Preferences preferences;

Creación del objeto preferences. Preferences facilita el almacenamiento y la recuperación de datos en la memoria flash del dispositivo, es decir, en la memoria no volátil. Este objeto nos permitirá almacenar datos para que sobrevivan a un corte de corriente. Todo aquel dato que almacenemos en el podrá ser recuperado incluso si se reinicia la placa.

**9 creación del servidor.**

WiFiServer server(80);

Creación de un objeto servidor que atenderá las conexiones que reciba desde el puerto especificado. Se puede establecer cualquier puerto y se producirá la conexión con otros dispositivos que se comporten como clientes, siendo aconsejable no usar los puertos más comunes. No obstante, no se podrá tener acceso al webservice si no se establece el puerto 80, por el HTML request.

Void setup()

**1 inicialización de cada uno de los elementos display, y ajuste de intensidad.**

Serial.begin(115200);

  character[1] = '\0';

  parola.setIntensity(0);

  parola.begin(5);

  parola.setZone(0, 4, 4);

  parola.setZone(1, 3, 3);

  parola.setZone(2, 2, 2);

  parola.setZone(3, 1, 1);

  parola.setZone(4, 0, 0);

  for (int i = 0; i < MAX\_DEVICES; i++) {

    parola.displayZoneText(i, character, positions[i], 0, 0, PA\_PRINT, PA\_NO\_EFFECT);

  }

  mx.begin();

  mx.control(MD\_MAX72XX::INTENSITY, 2);

  pixels.begin();

  preferences.begin("migration");

  pSsid = preferences.getString("ssid", "network");

  pNetworkPassword = preferences.getString("networkPassword", "network12345");

  preferences.end();

* **Serial.begin(115200);** Inicialización de la consola serial para la impresión de todos los comandos de seguimiento.
* **parola.setIntensity(0);** Establece la intensidad de todas las zonas del display. Existen 16 niveles de intensidad. El nivel 0 será el nivel mínimo de intensidad sin encontrarse apagado.
* **character[1] = '\0';** Establece el carácter en la posición 1 del array character como un elemento null. Para una explicación más detallada de este elemento consultad la sección [array character[]](#Array character[]).
* **parola.begin(5);** Inicialización del objeto display parola así como el número de zonas que van a usarse.
* **parola.setZone();** Se establece el número de zona de acuerdo al display, o rango de displays que se establecen como parámetros. En el caso “ parola.setZone(0, 4, 4);” se establece como zona 0 el rango de displays que va desde el display 4 hasta el display 4.
* **for (int i = 0; i < MAX\_DEVICES; i++) {**

**parola.displayZoneText(i, character, positions[i], parola.getSpeed(), 0, PA\_PRINT, PA\_NO\_EFFECT);**

**}**

Bucle for que iterará tantas veces cómo número máximo de dispositivos existan. Por cada iteración se invocará una vez el método displayZoneText. Este método se encarga de configurar todos los parámetros necesarios para mostrar la animación y la inicia. Cuando se dice que la “Inicia” se refiere a que la maraca como lista para empezar, para ser actualizada frame a frame por el método displayAnimate(). Este método consta de varios parámetros que se explican a continuación:

- **i:** indice de la zona a configurar.

- **Character:** char a mostrar en el display.

- **positions[i]:** este parámetro permite configurar la posición del char a imprimir en el display. Encontramos las opciones PA\_LEFT para obtener una alineación a la izquierda,

es decir, que el texto se muestre una alineación a la izquierda; PA\_CENTER para que el texto semuestre centrado; y PA\_RIGHT para que el texto muestre una alineación a la derecha.

La alineación de cada display será la correspondiente a la que se encuentre en cada posición del array. Las alineaciones positions[0-2] será igual a PA\_LEFT, y las alineaciones positions[3-4] serán igual a PA\_RIGHT. La explicación al por qué de estas alineaciones se puede encontrar en la explicación a la declaración del array positions en apartado anterior de la presente documentación.

- **0:** Este parámetro establecerá la velocidad a la que se ha de mostrar la animación, es decir, el tiempo que ha de pasar entre frame y frame, independientemente de cuantas veces se ejecute el método displayAnimate(). En este caso lo establecemos a 0 porque no se trata de una animación con efectos de entrada y salida. Es decir, el texto no entra desde una dirección y sale por otra, en cuyo caso se debería ajustar la velocidad para que sea perceptible al ojo humano, sino que más bien simplemente se imprime directamente el texto, sin efectos de entrada o salida. Por lo tanto no necesitamos ningún tipo de delay.

- **0:** Con este parámetro establecemos el tiempo de espera que ha de producirse una vez que acaba la animación de entrada y antes de que empiece la de salida.

- **PA\_PRINT** y **PA\_NO\_EFFECT:** Estos dos apartados configuran el tipo de animación con la que entra y sale, respectivamente, del display el carácter a imprimir. Existen múltiples efectos que pueden ser consultados mediante el línk [librería MD\_Parola](https://majicdesigns.github.io/MD_Parola/_m_d___parola_8h.html#acf3b849a996dbbe48ca173d2b0b82eda). Si establecemos el ejecto “PA\_SCROLL\_DOWN”, tanto en la entrada como en la salida, el texto empezará a mostrarse dede la parte superior del display y saldrá por la parte de abajo del mismo. En el presente caso se usa el efecto PA\_PRINT como efecto de entrada, que simplemente imprime el carácter sin effecto alguno, y el PA\_NO\_EFFECT, que se usa como un objeto con el que rellenar el espacio de configuración de efecto de salida para no ejecutar ninguna acción.

* **mx.begin();**

**mx.control(MD\_MAX72XX::INTENSITY, 2);**

Inicialización del objeto display mx, el encargado del despliegue de elementos gráficos en el display, y configuración de la intensidad en el nivel 2.

Se tiene que establecer la intensidad en cada uno de los objetos display porque cada uno trabaja con su propia configuración, aunque actúen sobre el mismo display.

* **pixels.begin();** Inicialización del objeto pixels, el objeto encargado de controlar la tira de leds WS2812.
* **preferences.begin("migration"**); Inicialización del objeto preferences. Iniciamos el objeto con el nombre de las preferencias que deseamos cargar.
* **pSsid = preferences.getString("ssid", "network");**

**pNetworkPassword = preferences.getString("networkPassword", "network12345");**

Declaración de las variables string pSsid y pNetworkPassword. En estas variables se guardarán los strings de nombre de red y contraseña, con los que iniciaremos la conexión wifi. Para ello castearemos el método getString de la clase preferences, con dos parámetros. El primero será el nombre con el que se ha guardó la ssid o la networkPassword si acaso se solicitó un cambio de red previamente; el segundo parámetro será el valor por defecto que se cargará en caso de que no exista ningún dato guardado en las preferencias establecidas, para las referencias dadas, (en caso de que no existan datos ssid y networkPassword).

* **preferences.end();** Cerramos las preferencias para tener un mejor control sobre el ciclo de guardado de las mismas, ya que sin esta sentencia no se produciría el guardado de nuevos datos. De esta forma sabemos que se tendrán que iniciar las preferencias cuando se vaya a actualizar algún dato de las mismas.

**2 impresión del nombre y contraseña de la red a la que se intentará conectar el ESP32.**

  Serial.println("----\*\* Network \*\*----");

  Serial.println(pSsid);

  Serial.println(pNetworkPassword);

  Serial.println("----\*\*\*\*\*\*\*\*\*\*\*\*\*----");

**3 declaración de salidas y entradas.**

  pinMode(RELAY, OUTPUT);        //Action Led

  pinMode(STATE\_LED, OUTPUT);    //Wifi conexion state

  pinMode(RESET\_BUTTON, INPUT);  //Reset button

  pinMode(KEY, INPUT);           //Key switch

  pinMode(TAMPER, INPUT);        //Tamper

* **pinMode(RELAY, OUTPUT);** declaración del pin de relé de control de 230V como pin de salida.
* **pinMode(STATE\_LED, OUTPUT);** declaración del pin del estado del de control de 230V como pin de salida.
* **pinMode(RESET\_BUTTON, INPUT);** declaración del pin del botón de reseteo de parámetros de red como pin de entrada.
* **pinMode(KEY, INPUT);** declaración del pin de la llave de safe-mode como pin de entrada.
* **pinMode(TAMPER, INPUT);** declaración del pin del tamper como pin de entrada.

**4 inicio de la conexión a la red wifi.**

  Serial.print("Connecting to " + String(pSsid) + "\n");

  preferencesIpConfig();

  tryConnection(pSsid, pNetworkPassword);

  digitalWrite(STATE\_LED, LOW);

  setHelp();

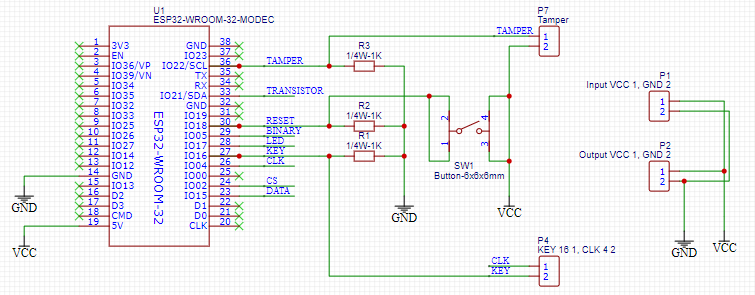
  server.begin();

* **digitalWrite(STATE\_LED, HIGH);** Se establece el led de estado de la conexión como HIGH. Permanecerá encendido mientras no haya conexión.
* **Serial.print("Connecting to " + String(pSsid) + "\n");** Impresión del nombre de la red a la que se va a conectar el dispositivo.
* **preferencesIpConfig();** Llamamos a este método para establecer la ip del dispositivo en caso de que haya alguna configuración previa. Cuando se ha producido un cambio de la ip desde el webservice, dicha ip queda guardada en las preferencias. Este método establecerá la configuración wifi como estática y con la ip guardada, cuando se arranca el ESP32, y siempre y cuando la red a la que se conecta sea la misma que aquella red para la que se solicitó el cambio de ip
* **tryConnection(pSsid, pNetworkPassword);** Llamada al método tryConnexion, el cuál intentará iniciar una conexión con la red de nombre y contraseña establecidos como parámetros del mismo.
* **digitalWrite(STATE\_LED, LOW);** Una vez que se haya producido la conexión, se establece el led de estado de conexión como LOW. Así permanecerá hasta que se pierda la conexión, en cuyo caso habrá otra función del código que se encargará de su gestión en función del estado de la red.
* **setHelp();** Llamada al método setHelp para asignarle a la variable stringHelp el texto, con formato HTTP, a imprimir cuando se solicite ayuda desde el webservice. Para mayor información acerca de este método haced [clic](#setHelp).
* **server.begin();** Se inicia el servidor para permitir la escucha de conexiones de cliente entrantes.

Void loop()

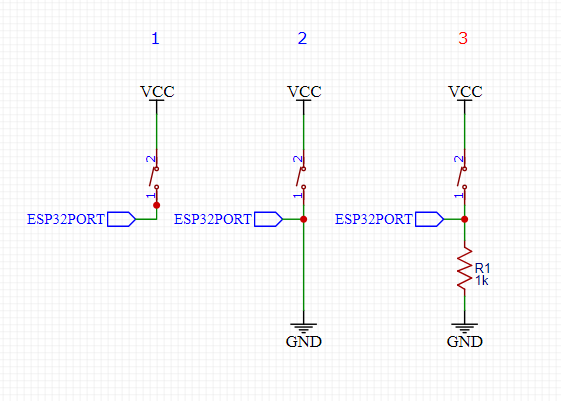
**1 detección de inputs.**

Procedemos a explicar unas nociones básicas de electrónica antes de comenzar con el desglose del código. Parte de las explicaciones dadas a continuación se basarán en la documentación oficial presente en <https://www.arduino.cc/>.



En la imagen anterior nos encontramos el esquema eléctrico del sensor de final de carrera, encargado de la función de tamper de la caja donde se montará todo el equipo, a fin de detectar cuando se manipula; del pulsador que permitirá el reseteo de los parámetros de red; y del interruptor de tipo llave que servirá para que el usuario pueda activar el modo seguro del sistema, es decir, para que pueda manipular el dispositivo libremente sin que se active el modo “System Failure”.

En los tres casos el principio de funcionamiento es el mismo, lo único que varía es la construcción. Mientras que el pulsador encargado del reseteo de los parámetros de red está soldado en placa, el sensor de final de carrera y el interruptor de tipo llave no lo están, se conectarán a la placa por medio de clemas soldadas a la misma. Adicionalmente, la conexión a placa de estos dos últimos es diferente. Mientras que ambos terminales del sensor de final de carrera se conectarán con las dos entradas de la clema P7, uno de los pines del interruptor de tipo llave se conectará a la entrada 2 de la clema P4, y el segundo pin se conectará a la entrada 1 de la clema P2. El sensor de final de carrera se cableará en sus terminales normalmente cerrados para que el circuito quede abierto cuando se cierre la tapa. Por otra parte, el interruptor de tipo llave se cableará en sus terminales normalmente abiertos.



En la anterior imagen se muestran tres escenarios. En el primero se observa la conexión directa entre la entrada y VCC, con un interruptor intercalado en la línea; en el segundo escenario se ha añadido, a continuación del interruptor y en paralelo con la conexión a la entrada del ESP32, una conexión con GND; en el tercer caso tenemos la misma construcción que en el caso dos, pero con una resistencia a continuación del interruptor.

Si usamos la primera construcción obtendremos un valor alto siempre que el pulsador esté cerrado. Cuando nos referimos a un pulso alto al hablar de una señal electrónica, lo que tenemos es una lectura del voltaje en la entrada de la placa que supera un valor predeterminado que la placa considera como HIGH. En el caso del presente proyecto, VCC es igual a cinco voltios. Por lo tanto, mientras que el pulsador esté cerrado, tendremos un valor cercano a los cinco voltios en la entrada, que serán interpretados como la variable de tipo int “**HIGH**”, cuyo valor es 1. No obstante, si el pulsador no está pulsado, la entrada del ESP32 quedará “al aire”, sin conexión. Este estado de conexión “al aire” es conocido como “**FLOAT**”. Es por esta razón que no se puede usar esta construcción. Cuando una entrada queda sin conexión y es leída mediante digitalRead(), el resultado de la lectura será arbitrario, fluctuará, debido a que dicha entrada “al aire” es susceptible de variaciones producidas por campos electromagnéticos y ruidos.

int pushButton = 5;

void setup() {

  Serial.begin(9600);

  pinMode(pushButton, INPUT);

}

void loop() {

  if (digitalRead(pushButton) == HIGH) {

    Serial.println("High");

  } else {

    Serial.println("Low");

  }

}

21:13:54.080 -> Low

21:13:54.080 -> Low

21:13:54.080 -> Low

21:13:54.080 -> Low

21:13:54.114 -> High

21:13:54.114 -> High

21:13:54.114 -> High

21:13:54.114 -> Low

21:13:54.114 -> Low

En el ejercicio anterior tenemos un código que, tras iniciar el puerto serie para la impresión de valores, imprimirá en la consola “HIGH” o “LOW” en función del valor leído en la entrada digital cinco. Vemos como, en tan solo unos pocos milisegundos y sin presionar el pulsador, obtenemos cuatro lecturas de valor LOW, tres de valor HIGH, y dos de valor LOW. Como se ve, cualquier ruido en la línea ocasionado por otros dispositivos generará falsos positivos.

En la segunda construcción, cuando el pulsador no sea accionado, el valor de lectura de la entrada en la placa será de 0, o “**LOW**”, puesto que estará conectado con GND. Esto solventaría el problema generado cuando la entrada queda “al aire” si no se está presionando el pulsador, sin embargo, cuando se presionase, lo que estaríamos ocasionando es un cortocircuito puesto que estaríamos conectando VCC a GND directamente. Esto provocaría que se dañasen diversas partes de la placa si se usa la alimentación de los terminales “5V” y “GND”, o la fuente de alimentación si el corto se produce entre los terminales de las clemas P1 o P2.

La resistencia añadida en el escenario tres será la encargada de evitar el cortocircuito. De esta forma cuando se presiona el pulsador obtenemos un valor HIGH, y cuando no se presiona obtenemos un valor LOW.

|  |
| --- |
| Nota 1: Existen dos tipos de construcciones con resistencias para la lectura de un pulsador en una entrada, la construcción “pull-up” y la construcción “pull-down”. En este proyecto se está usando la construcción de tipo pull-down.  Figura 1    En las construcciones de tipo pull-up tendremos un estado “HIGH” por defecto. Se conecta a vcc una resistencia, y al otro lado de la resistencia se conecta a una entrada de la placa en paralelo con un pulsador. Mientras que no se presione el pulsador, este permanecerá abierto, y por lo tanto tendremos un voltaje alto a la entrada de la placa. Cuando el pulsador sea presionado, se cerrará el circuito con GND generando un pulso LOW en la misma.  Por otra parte, en construcciones de tipo pull-down tenemos un estado “LOW” por defecto. En este caso, uno de los terminales del pulsador se sitúa a la salida de la fuente y el otro se conecta a la entrada de la placa en paralelo con uno de los lados de la resistencia. Cerraremos el circuito conectando el otro lado de la resistencia a GND. Esta construcción genera que tengamos un valor de cero voltios en la entrada mientras que el pulsador permanezca abierto. Cuando el circuito se cierre pasaremos a tener un valor cercano al de VCC.  El valor de las resistencias escogido para estas construcciones es de 10k ohm. Se toma este valor porque es suficiente como para eliminar el efecto “FLOAT”, siendo al mismo tiempo lo suficientemente alto como para no generar mucha corriente en la entrada cuando se está presionando el pulsador.      Tenemos el ESP32 conectado, por usb, a un power bank. Con el pin de salida de 5V de la placa marcando un voltaje de 4.76V, la corriente leída cuando se usa una resistencia de 1kΩ es de 4,72 mA. Cuando sustituimos la resistencia de 1kΩ por una resistencia de 10kΩ, la corriente cae hasta los 474 µA. Siguiendo la ley de ohm podremos comprobar de forma teórica los valores obtenidos en el polímetro.  V  I = ----  R  I = 4,76 V / 1000 ohm = 0,00476 A = 4.76 mA.  I = 4,76 V / 10000 ohm = 0,000476 A = 476 µA. |

|  |
| --- |
| Nota 2: En el esquema eléctrico de la figura 1, se aprecia la existencia de un condensador de 0.1uF en paralelo con la resistencia de 10k. Este condensador se encarga de eliminar los rebotes del pulsador. Cuando se presiona un pulsador, las placas metálicas internas del mismo entran en contacto permitiendo la continuidad del circuito. No obstante, aunque parece que la unión de ambas superficies es inmediata si se observa a simple vista, la realidad es diferente. El contacto entre ambas superficies no es regular, es decir, no se unen ambas superficies inmediatamente y de forma plena, existen fluctuaciones, los llamados rebotes, que generan el cierre y la apertura del circuito, de forma repetida, en periodos de microsegundos a milisegundos. Estas fluctuaciones pueden ser leídas por la placa, lo que generará falsos positivos en el código. Ilustremos lo dicho con imágenes.      En el ejercicio anterior tenemos un código que se encarga de imprimir la palabra “High”, y el valor de la variable counter tras incrementar en una unidad su valor, cada vez que se presiona el pulsador en la entrada 12. Tenemos un bloque while que se encargará de no enviar múltiples mensajes en caso de que se mantenga presionado el pulsador.  En el caso anterior el circuito se encuentra sin el condensador, y en paralelo con la resistencia. En este punto, se ha presionado el pulsador repetidas veces hasta llegar a 40 como valor de la variable counter. No obstante, los últimos mensajes reportados, es decir, los valores de la variable counter que van del 36 al 39 responden a presionar el pulsador únicamente dos veces. Se presionó una vez reportando las líneas “High” y counter cambiando de 36 a 37, y la segunda vez imprimiendo “High 38”, “High 39”. Esto se debe al rebote producido por el pulsador, rebote que se puede apreciar claramente en la captura de pantalla tomada en el osciloscopio.      Lo mismo ocurre en el ejemplo anterior. Se presiona una primera vez ocasionando la impresión de counter igual a 0. Acto seguido se presiona una segunda vez reportando counter igual a 1 y counter igual a 2. Como podemos observar, por los cursores del osciloscopio, este rebote sucede en un periodo de tiempo de 1,890 milisegundos, sin embargo, reportar ambos pulsos altos le lleva a la placa 109 milisegundos, (270 – 161). Esto es debido al tiempo de procesamiento de los eventos sucedidos en la placa, tanto la lectura de la entrada, como el análisis y ejecución del bloque condicional.    En esta ocasión, tras agregar el condensador en paralelo con la resistencia, vemos que el rebote queda parcialmente mitigado por la descarga del condensador. Es decir, debido al tiempo de descarga del condensador, el voltaje durante el momento en que la placa interna del pulsador está rebotando, no es cero. Siendo que no es cero, y siendo que el voltaje en ese instante es superior al valor mínimo necesario para que el procesador lo considere como un pulso alto, se pasa de la impresión “High 7” en la anterior vez que se presionó el pulsador, a la impresión de “High 8” en la ocasión actual, sin que se imprima ninguna línea adicional.      Problema  Lo anteriormente explicado funciona sin problema cuando el tiempo transcurrido desde que se deja de presionar el pulsador, hasta que la placa interna del mismo rebota, es pequeño. No obstante, si el proceso de rebotar toma un tiempo más prolongado, el condensador hará su función, pero la placa podrá realizar lecturas adicionales.  En la anterior captura del osciloscopio vemos como el condensador empieza a descargarse tan pronto como se deja de presionar el pulsador, sin embargo, transcurrido un tiempo se produce un rebote. Ese rebote realiza una pequeña recarga del condensador y, cuando la placa interna del pulsador termina de rebotar y vuelve a su estado de reposo, el condensador sigue descargándose hasta quedarse vacío.  El problema viene si el rebote ocurre cuando la carga del condensador está por debajo del valor umbral que el ESP32 considera que es un pulso bajo.  **HIGH**   * a voltage greater than 3.0V is present at the pin (5V boards) * a voltage greater than 2.0V is present at the pin (3.3V boards)   **LOW**   * a voltage less than 1.5V is present at the pin (5V boards) * a voltage less than 1.0V (Approx) is present at the pin (3.3V boards)   La información anterior se extrae de la página oficial de Arduino. Indica los valores para los que la lectura de un input es considerada como HIGH o LOW. Voltajes superiores a 3 voltios en placas de 5V, o superiores a 2V en placas de 3.3V, serán considerados como HIGH. Voltajes inferiores a 1.5 voltios en placas de 5V, o inferiores a 1V en placas de 3.3V, serán considerados como LOW.  La placa usada para esta prueba es un Arduino nano. Si nos fijamos en la captura del osciloscopio, el rango de voltaje está configurado a 1 voltio por división. La onda cae desde algo menos de 5V hasta más o menos la primera línea de división de voltaje. Esto implica que en ese punto el voltaje es de 1 voltio. Teniendo en cuenta que, como hemos dicho, el voltaje desde el cual Arduino considera como LOW una señal en un rango de 5V es de 1.5 o menos, toda lectura que se produzca desde que el voltaje cae desde 1.5V hasta 0, mientras se descarga el condensador, será considerada como LOW. Al considerarse como LOW el bucle while de control de pulsación deja de iterar, se realiza la acción del condicional y se vuelve al inicio del void loop. Cuando se produce el rebote a posteriori, se vuelve a acceder al condicional, repitiendo lo anteriormente explicado, generando así un falso positivo.  Si se incrementa la capacidad del condensador los resultados mejoran, pero no se eliminan por completo, y adicionalmente obtenemos nuevas lecturas sin que se produzcan rebotes. Estas nuevas lecturas suceden  El motivo de incluir toda esta teoría acerca de los condensadores y del rebote de los pulsadores es, en primer lugar, el de documentar el estudio realizado al respecto durante el desarrollo del dispositivo. Si bien ofrece buenos resultados, no se consolida como la solución definitiva y viable.      Tras rediseñar el esquema eléctrico, así como el diseño de la placa, las pruebas indican que no merece la pena lanzar a producción dicha modificación.  Solución  La finalidad de las entradas de este proyecto es la de ofrecer un tipo de funcionalidad que no será requerida de forma constante por la naturaleza de la misma. Es decir, el usuario no necesitará presionar constantemente y a gran velocidad el pulsador de reseteo de los parámetros de red; de la misma forma, no necesitará abrir la tapa de la caja del dispositivo o activar y desactivar rápidamente la función llave. Dado que el uso de dichas funciones va a ser esporádico, nos podemos permitir perder precisión en la lectura.  Si la finalidad de cualquiera de los trigger del dispositivo fuese la de captar todas y cada una de las activaciones que se produjeran, sin duda necesitaríamos que se mitigasen todos los rebotes en un escenario en el que se pudiesen realizar tantas lecturas de estado como fueran posibles. En un escenario como el presente, en el que la acción a realizar en caso de lectura positiva no responde a criterios de velocidad de lectura y acción, podemos agregar una simple pausa al finalizar la lectura de la entrada pulsada, a fin de que el programa quede en “pausa” mientras se producen dichos rebotes.    if (digitalRead(pushButton) == HIGH){      while(digitalRead(pushButton)){}      Serial.println("High");      Serial.println(counter++);      delay(50);    }  Agregando la línea delay y un valor simbólico, le estaremos añadiendo a la ejecución del código un breve retraso de milisegundos, totalmente despreciables para el usuario teniendo en cuenta que los rebotes de un pulsador suceden en periodos de tiempo medidos en microsegundos o unos pocos milisegundos. Durante este periodo de tiempo en el que se pausa la ejecución del código, cualquier rebote quedará fuera de toda lectura.  Se establecen 50 milisegundos de retraso a fin de fijar un valor en exceso. Se podría recortar ese valor puesto que la media de tiempo que le toma al rebote, tras una sucesión de varias pruebas, no suele superar los 10 a 20 milisegundos. No obstante, existen casos puntuales en los que se exceden estas medidas. |

* 1. **Detección de llave activada.**

  if (digitalRead(KEY)) {

    Serial.println("Inside KEY");

    parola.displayClear();

    if (digitalRead(RELAY)) {

      digitalWrite(RELAY, LOW);

    }

    while (digitalRead(KEY)) {

      WiFiClient client = server.available();

      if (client) {

        client.write("key\_activated");

      }

      if (digitalRead(RESET\_BUTTON)) {

        while (digitalRead(RESET\_BUTTON)) {}

        Serial.println("\nNetwork reset requested.");

        Serial.println("\n----\*\* Network \*\*----\nnetwork\nnetwork12345\n----\*\*\*\*\*\*\*\*\*\*\*\*\*----");

        saveIpPreferences(false);

        saveNetworkPreferences("network", "network12345");

        ESP.restart();

      }

    }

delay(50);

  }

Se procede a leer la entrada KEY, si el resultado de la lectura es un pulso alto se accede al interior del bloque condicional. Una vez dentro del condicional se empieza imprimiendo en la consola un mensaje indicando que se ha accedido al interior del condicional. A continuación, se llama al método displayClear para borrar lo que esté impreso en el display. Hay que recordar que la función de llave elimina cualquier estado previo, es decir, si el estado previo es la activación del modo systemFailure, al activar la llave el display borrará los jeroglíficos impresos por el modo failure. Así mismo, desactivará el relé de control de 230V en caso de que se encuentre activo. El siguiente bucle while se encargará de realizar la acción del condicional una única vez independientemente de cuánto tiempo permanezca presionado el pulsador.

|  |
| --- |
| Nota 3: La finalidad de este bucle es la de asegurar una sola acción cada vez que se active el pulsador. Esto es necesario porque cuando se acciona un pulsador, el acto de pulsarlo en condiciones normales dura escasamente un segundo, no obstante, durante ese segundo se ha producido la lectura positiva y la ejecución del bloque interior del condicional varias veces debido a la velocidad de procesamiento de la placa.  Ejemplo:  int pushButton = 18;  int counter = 0;  void setup() {    Serial.begin(9600);    pinMode(pushButton, INPUT);  }  void loop() {    if (digitalRead(pushButton) == HIGH) {      // while(digitalRead(pushButton) == HIGH){}      counter++;      Serial.println("High " + String(counter));    }  }  15:39:01.858 -> �1  15:39:01.858 -> 2  15:39:01.858 -> 3  15:39:01.858 -> 4  15:39:01.858 -> 5  15:39:01.858 -> 6  15:39:01.858 -> 7  15:39:01.890 -> 8  15:39:01.890 -> 9  15:39:01.890 -> 10  15:39:01.890 -> 11  15:39:01.890 -> 12  15:39:01.890 -> 13  15:39:01.890 -> 14  15:39:01.922 -> 15  15:39:01.922 -> 16  15:39:01.922 -> 17  15:39:01.922 -> 18  15:39:01.922 -> 19  15:39:01.922 -> 20  15:39:01.955 -> 21  15:39:01.955 -> 22  15:39:01.955 -> 23  15:39:01.955 -> 24  15:39:01.955 -> 25  15:39:01.955 -> 26  15:39:02.001 -> 27  15:39:02.001 -> 28  El anterior código de ejemplo imprimirá el valor de la variable counter en una línea nueva, cada vez que se acceda al condicional, siendo que el condicional tiene como requisito para su resultado true la lectura de un pulso alto en la entrada 18, la entrada del pulsador. El valor de esta variable se incrementará en una unidad cada vez que se acceda al condicional. Como se observa en el resultado, en tan solo 143 milisegundos que se ha tardado físicamente en presionar y liberar el pulsador, se ha accedido al condicional y se ha realizado la acción del interior del mismo un total de 28 veces.  Sin embargo, cuando agregamos la línea “while(digitalRead(pushButton) == HIGH){}” al inicio del condicional, podemos presionar el botón por tanto tiempo como queramos sin que se realice la acción más de una vez. En este segundo ejemplo se ha agregado la línea ”Serial.println("Time stamp.");” antes del bucle while a fin de imprimir un mensaje que sirva de referencia de tiempo de inicio, para que al compararse dicha referencia con la referencia de tiempo final marcada por la impresión de la propia variable de conteo al liberar el pulsador, se pueda determinar el tiempo que ha permanecido cerrado el pulsador.    if (digitalRead(pushButton) == HIGH) {      Serial.println("Time stamp.");      while(digitalRead(pushButton) == HIGH){}      counter++;      Serial.println(String(counter) + " ");    }  15:46:54.605 -> Time stamp.  15:46:58.239 -> 1  Como vemos, se ejecuta una única vez el bloque de acción del condicional, aun cuando se ha presionado el pulsador durante algo menos de 4 segundos. |

Cuando activamos la llave, el código entra en el condicional, ejecuta lo que haya en el interior, y vuelve al void loop. Como la llave sigue activada porque no es un pulsador que permanece presionado únicamente unos milisegundos, el ciclo de ejecución se vuelve a repetir, y esto ocurrirá de forma constante hasta que la llave deje de estar activada. Esta situación supondría un exceso de ejecución de líneas que solo precisan ser ejecutadas una vez. Es decir, es necesario que se imprima el mensaje en consola, que se reinicie el display, o que se desconecte la salida del relé, una única vez. Si se ejecuta constantemente el condicional de la llave mientras esta permanece activa, las acciones anteriormente citadas serían ejecutadas, innecesariamente, de forma constante.

Una vez dentro del bucle, se creará un objeto cliente a partir del método server.available(), y analizará si existe una nueva conexión entre el servidor y el cliente y, en caso de que exista dicha conexión, el servidor le enviará al cliente un mensaje indicándole que la llave se encuentra activada. El condicional (digitalRead(RESET\_BUTTON)) se encargará de analizar si se presiona el pulsador de reseteo. Este es el único momento en el que se puede leer el estado del pulsador de reseteo de los parámetros de red, cuando aún no se ha establecido la conexión servidor – cliente pero sí que hay red (dado que cuando se está intentando establecer la conexión mediante el método ‘tryConnection’ sí que se podrían reiniciar los parámetros de red), puesto que la idea de fondo es que sólo se puedan reiniciar dichos parámetros si se ha accedido a la placa de forma autorizada, es decir, con la llave.

Tras acceder al bloque de acción del pulsador de reseteo, iniciamos con un bucle de control de pulsación como el que se ha explicado anteriormente. Imprimiremos en la consola dos mensajes indicando la operación solicitada y el nombre y la contraseña de la red por defecto a la que se conectará una vez que se reinicie el dispositivo. A continuación, estableceremos el valor de la variable anySavedIp, guardada en las preferencias a través del método saveIpPreferences, como false, para evitar que se ejecute el código interno del mismo método cuando se reinicie el equipo. Cambiaremos los valores de nombre de red y contraseña guardados en las preferencias, por los parámetros de red por defecto, por medio del método saveNetworkPreferences. Y por último reiniciamos la placa.

Click en [saveIpPreferences](#saveIpPreferences) o [saveNetworkPreferences](#saveNetworkPreferences) para una explicación más detallada del método de cada método.

|  |
| --- |
| Nota 4: Cuando se produce un cambio en la ip del dispositivo por medio de la función “WiFi.config(staticIP, gateway, subnet);”, dicha función establece la configuración ip del objeto wifi de la placa como ip estática. Si la configuración ip de la placa es estática, no se podrá establecer en modo DHCP de nuevo en caso de que se necesite que el rúter le asigne una ip al dispositivo. Es decir, si como ocurre en el presente caso, lo que se pretende es cambiar la red a la que se conecta el dispositivo, y que sea el rúter de la nueva red el que le asigne la ip al dispositivo, una vez que se haya logrado la conexión con éxito el dispositivo mantendrá esta nueva ip configurada con el citado método aun si se vuelve a solicitar un cambio de red. Esto puede suponer un problema puesto que es posible que la última configuración de red establecida no esté en el rango ip de la nueva red.  Siguiendo un ejemplo real, durante el proceso de desarrollo de este proyecto se ha utilizado principalmente una red wifi compartida desde un teléfono Android, la opción tethering del mismo. El teléfono usado para ello asigna la ip de los dispositivos que se conectan a la red mediante el protocolo DHCP, con una configuración de máscara de subred de tipo /24. Esto quiere decir que de los cuatro octetos de bits que forman la ip, es decir, de los cuatro números de la ip, los tres primeros identifican la red y el último identifica el host. El tercer octeto será asignado de forma dinámica por el protocolo DHCP cada vez que se inicia el tethering. El último octeto será asignado en función del cada dispositivo.  La placa ESP32 usada en el proyecto recibe siempre el número 138 como número de host en la ip asignada por el DHCP al conectarse a la red compartida. Cuando se conecta otro teléfono Android que se dispone para pruebas, este siempre recibe el número 156 como número de host. Será el tercer número de la ip el que variará cada vez que encendamos el tethering, siendo que el mismo dispositivo podrá obtener, por ejemplo, la ip “192.168.**23**.138” y, a continuación, obtener la ip “192.168.**100**.138” tras apagar y encender la red compartida.  El proyecto está destinado a ser instalado en un hogar, donde se conectará a una red doméstica en la que el rúter le asignará una ip del tipo “192.168.1.XXX”. El método que establece la conexión, sustituirá la asignación del host establecida por el rúter, para asignarle a la placa el número de host 138 tan pronto como detecte este tipo de construcción ip. Esto se hace para que se pueda establecer la conexión con el software en el pc, dado que dicho software está programado para conectarse a un servidor en la red local “192.168.1.138”. En este instante ya se ha establecido la configuración ip del dispositivo como estática al usar el método “config()” de la clase WiFi para lograr dicho cambio del número de host. En este escenario, si se precisa que el dispositivo se conecte a otra red por cualquier razón, o como era el caso durante el proceso de desarrollo, si se necesita que se vuelva a conectar a la red compartida a través del dispositivo android, la ip de la placa seguirá siendo “192.168.1.138”. Según lo explicado acerca de la asignación de direcciones ip por parte del teléfono, esta ip solo estaría en rango con la red del teléfono si da la casualidad de que este establezca como identificador de red la ip “192.168.1.XXX”. Lo cual es altamente improbable teniendo en cuenta que tiene 254 opciones, es decir, puede establecer desde el rango indicado, hasta el rango “192.168.254.XXX”. |

Cuando le realizamos un reinicio a la placa, bien sea mediante el botón reset de la misma, bien sea por medio del corte de la alimentación, la configuración estática del objeto WiFi se pierde pasando a tener una configuración de tipo DHCP. Esto es así porque al reiniciar la placa se carga todo el código de nuevo en su estado por defecto, salvo aquellos valores guardados en las preferencias. Se precisa este tipo de acción porque no existe ningún método en la librería que permita configurar el WiFi nuevamente como DHCP.

Por último, si no se presiona el botón de reset mientras la llave está activada, y se cierra la llave, el bucle while dejará de iterar y llegará a la última línea del condicional. El delay(50), tal y como se explicó en la [nota 2](#_Nota_2:), eliminará los efectos de los rebotes del pulsador.

* 1. **Detección de tamper no presionado.**

La función de tamper servirá para conocer si alguien ha manipulado la tapadera del dispositivo accediendo al mismo. En caso de que esto ocurra se iniciará el modo systemFailure hasta que se vuelva a cerrar dicha tapa.

else if (digitalRead(TAMPER)) {

    Serial.println("Tamper triggered.");

    systemFailureOn();

    while (digitalRead(TAMPER) == HIGH) {

      WiFiClient client = server.available();

      if (client) {

        client.write("stopSystem");

      }

    }

    systemFailureOff();

delay(50);

  }

Este bloque es muy similar al anterior. Cuando se presiona el tamper se accede al interior del mismo. Se envía un mensaje informativo a la consola indicando que el tamper se ha activado y se carga la función “systemFailureOn()”. A continuación, nos encontramos el mismo bloque de control de pulsación. El bucle permanecerá iterando mientras se mantenga abierto el sensor de final de carrera. Si mientras esto ocurre un cliente se conecta al servidor, se le enviará al mismo un reporte de estado indicándole que se debe activar el systemFailure del cliente al estar el tamper activado.

Cuando el sensor de final de carrera vuelva a estar presionado se saldrá del bucle y se llamará al método systemFailureOff() para desactivar tanto el relé de 230V como el cartel luminoso.

Para finalizar, nos encontramos nuevamente con la sentencia delay para depurar los efectos del rebote del pulsador.

1. **Bloque else.**

  else {

    WiFiClient client = server.available();

    if (client) {

  }

    connectionStateLed();

    if (getExitLoop() == true) {

      exitLoop = false;

      binaryClose();

    }

  }

|  |
| --- |
| Nota 5: El bloque principal del código, el encargado de establecer la conexión cliente-servidor, se engloba en el apartado else del condicional principal de la sección void loop, porque ha de ejecutarse únicamente si tanto la llave como el tamper están en su condición de reposo. |

WiFiClient client = server.available();

Creación de un objeto cliente a partir del resultado del método “.available()” de la clase server. Este método comprobará si existe algún cliente conectado al servidor. Devolverá un objeto WiFiClient si existe algún cliente conectado, o un objeto null si no existe ningún cliente conectado. Se accederá al interior del bloque condicional en caso de que exista una conexión al servidor por parte de un cliente.

|  |
| --- |
| Nota 6: No podrá existir la comunicación simultánea del servidor con el sistema Dharma Initiative y con el servicio web del servidor. Esto es debido a que la conexión entre el servidor y el sistema Dharma Initiative se mantiene activa mientras no se den las condiciones necesarias, como son un error a la hora introducir la clave numérica, la ausencia de una clave numérica antes de que se agote el tiempo, o un error de conexión, para cerrarla. Esto hace que el canal de comunicación se mantenga ocupado de forma constante impidiendo que otros sistemas, la petición de web-service, pueda acceder al mismo. |

Dado que el interior del bloque condicional es el apartado más largo de todo el código, y con diferencia, se procede a explicar primero lo que sucede a continuación del condicional, lo que supone el final del apartado void loop, antes de explicar el condicional en sí.

* 1. **connectionStateLed();**

void connectionStateLed() {

  int signal = WiFi.RSSI();

  currentTime = millis();

  if (digitalRead(STATE\_LED) == HIGH && (signal < 0)) {

    digitalWrite(STATE\_LED, LOW);

  } else if (signal < 0 && getResetValues() == false) {

    if ((currentTime - previousTime) >= 10000) {

      systemFailureOn();

      setResetValues(true);

    }

  } else if (signal >= 0) {

    digitalWrite(STATE\_LED, HIGH);

    tryConnection(pSsid, pNetworkPassword);

  }

}

El método RSSI devuelve el nivel de señal que tiene la placa con respecto al dispositivo que le suministra red. Devuelve un valor numérico de tipo long cuya unidad de medida son los dBm (decibelio-milivatio). Llamar a este método tras una pérdida de conexión reportará un valor de 0 dBm. Cuanta más distancia haya entre la placa y la fuente de red, menor será el valor reportado, siendo siempre inferior a 0.

|  |
| --- |
| **Nota 7:**  Compartiendo conexión con un teléfono móvil, y situando el teléfono justo encima de la placa, concretamente por la zona de la antena de la placa de red del teléfono, el valor obtenido indicando el “mayor” nivel de señal será de -6dBm. Este valor se hace cada vez menor a medida que se aleja el teléfono de la placa, llegando a los -93, -94 dBm, con algún muro entre la fuente y la placa, antes de que se pierda la conexión. |

En la librería WiFi.h existe el método WiFi.status(). Este método devuelve el estado de la conexión. A continuación, se listan los valores devueltos:

* **WL\_IDLE\_STATUS**, value = 0, es un estado temporal reportado cuando se llama a WiFi.begin(), y permanece activo hasta que se alcanza el número máximo de intentos, en cuyo caso se reportará WL\_CONNECT\_FAILED, o hasta que se produce la conexión, en cuyo caso reportará WL\_CONNECTED.
* **WL\_NO\_SSID\_AVAIL**, value = 1, reportado cuando la ssid no se puede encontrar o no está disponible
* **WL\_SCAN\_COMPLETED**, value = 2, reportado cuando el escaneo de redes se ha completado.
* **WL\_CONNECTED**, value = 3, cuando la placa está conectada a la red wifi.
* **WL\_CONNECT\_FAILED**, value = 4, reportado cuando un intento de conexión ha fallado.
* **WL\_CONNECTION\_LOST**, value = 5, reportado cuando se pierde la conexión.
* **WL\_DISCONNECTED**, value = 6, reportado cuando se produce la desconexión de la red.
* **WL\_NO\_SHIELD** cuando no existe un módulo wifi presente.

Aun cuando el método devuelve diversos estados que pueden resultar de gran utilidad, en este caso se usa el nivel de señal para conocer el estado de la conexión porque el reporte de estado es inmediato a diferencia de lo que ocurre con WiFi.status(). Cuando se pierde la conexión se reporta nivel de señal cero inmediatamente. En ocasiones la solicitud de estado por medio de del método status tarda en reportar la desconexión o no la reporta. Se desconocen las causas.

  if (digitalRead(STATE\_LED) == HIGH && (signal < 0)) {

    digitalWrite(STATE\_LED, LOW);

  }

Primer escenario. Si el led de estado está encendido, y el nivel de señal entre la placa y la fuente de red wifi es inferior a 0, entonces se establecerá la salida del led de estado como LOW. Este escenario es el que se da cuando se recupera la conexión tras haberla perdido, es decir, el led estaba encendido porque no había red hasta que se ha recuperado, que es cuando se apaga el led.

else if (signal < 0 && getResetValues() == false) {

    if ((currentTime - previousTime) >= 10000) {

      systemFailureOn();

      setResetValues(true);

    }

  }

Segundo escenario. Si el nivel de señal es inferior a cero y el valor de la variable resetValues es igual a false. Si el valor de la variable resetValues es false cuando el nivel de señal es inferior a cero, la sincronía se ha perdido. La placa sigue conectada a la red, pero no existe sincronía con la aplicación del pc, lo que implica que el pc perdió la conexión.

Al acceder al interior del condicional nos encontramos con otro condicional anidado cuya función es mantener la placa a la espera, durante diez mil milisegundos, es decir, durante diez segundos. Este tiempo se debe al número de intentos de reconexión que se efectúan tras haber perdido la sincronía durante cinco ciclos. Ambas partes, la placa y el pc, están preparadas para esperar cinco ciclos de perdida de sincronía. Tras pasar por los ciclos de perdida de sincronía, aquella parte que haya perdido la conexión pasará por un proceso intento de reconexión, que se repetirá diez veces en un periodo de diez segundos, quedando la otra parte a la espera.

currentTime = millis();

Después de tomar el valor de la intensidad de la señal de red al inicio del método, se toma una referencia de tiempo mediante el método millis(). El método millis() devuelve el número de milisegundos transcurridos desde que se empezó a ejecutar el programa. Este valor se reinicia a cero después de aproximadamente cincuenta días.

Se le asigna un valor a previousTime cada segundo en el interior del bloque principal una vez se conecta un cliente al servidor. Restándole al valor de currentTime, el valor de la variable previousTime, obtenemos el tiempo transcurrido desde que se tomó la referencia previousTime hasta el momento presente. Si el resultado de la resta es igual o superior a diez segundos, accedemos al interior del condicional. Esta estructura nos asegura que la carga de los gráficos en el display así como la activación del relé por parte de la placa, sucederán justo en el momento indicado, con respecto al mensaje repetido “system failure” por parte del software de pc, aun sin sincronía ni conexión.

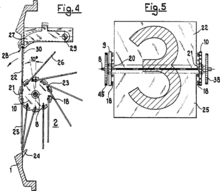
|  |
| --- |
| **Nota 8:**  No se ha implementado ningún mecanismo de control de overflow del método millis(). Si el dispositivo se mantuviese encendido por más de cincuenta días, y justo en el momento en el que se produce el overflow el pc pierde la conexión, podría darse un escenario en el que la variable “previousTime” tomase su valor antes de dicho overflow, y la variable currentTime después. En dicho caso, el valor de previousTime será mayor que el valor de la variable currentTime. Esta situación derivaría en que la resta de “currentTime – previousTime” reportaría un valor negativo, cuando lo normal es que reporte un valor positivo. Habitualmente dicha resta genera un valor positivo porque una medida de tiempo actual supone una cantidad de milisegundos transcurridos desde el inicio del programa, mayor, que una medida de tiempo pasada.  En este caso la placa nunca cargaría los métodos “systemFailureOn()” y “setResetValues(true)”, ocasionando que la próxima vez que se iniciase el programa en el pc se empezase con la cuenta regresiva en el punto en el que se hubiese quedado la vez anterior.  Se ha optado por no implementar ninguna corrección para evitar este resultado no deseado, debido a que el dispositivo no va a pasar a un escenario en el que se requiera su uso constante. Será más bien de carácter lúdico el uso que se le dará al mismo, resultando en que no permanecerá activo por tanto tiempo. Por ende, y debido otras cuestiones relacionadas con el desarrollador, no merecía la pena dedicarle más tiempo a dicho escenario, que el empleado en escribir esta nota. |

Una vez dentro del condicional nos encontramos con dos métodos.

* + 1. **systemFailureOn();**

El método systemFailureOn se encargará de ejecutar toda la secuencia de gráficos del display, así como de la activación del relé.

En la serie, la cuenta regresiva de tiempo, así como los dibujos / jeroglíficos, se muestran en un cartel cuyos dígitos están formados por varios letreros de tipo “split-flap display”.

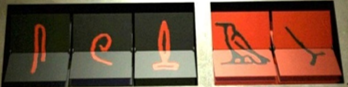


Ejemplos de letreros tipo “split-flap display”

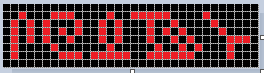
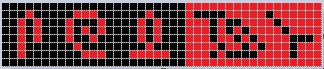
 

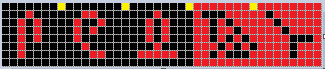
Durante el desarrollo del dispositivo se pensó en adaptar una serie de varios display de este tipo para hacer la experiencia más parecida a la serie. Sin embargo, pese a que se buscó en profundidad en internet, apenas había opciones y las opciones existentes o eran excesivamente caras, o simplemente eran otros dispositivos incluyendo algún display de este tipo. Existía la opción de diseñar los display en 3D para imprimirlos, pero no se cuenta con el conocimiento de diseño 3D, ni con la necesidad tan acentuada de usar este tipo de display en el proyecto como para dedicarle el tiempo al aprendizaje que ello conllevaría. Se pensó también, en tomar de internet algún modelo ya diseñado, sin embargo, se prefirió que el proyecto se elaborase con los conocimientos y materiales de los que se pudiese disponer según la necesidad. Llegados a este punto se optó por un display formado por 4 módulos dot matrix de 8\*8, controlados cada uno por un driver de display MAX7219CWG.

Como es obvio, los gráficos se tuvieron que adaptar a un formato de 8\*8. En un primer momento, el conjunto constó de cuatro displays, lo cual limitaba bastante la forma de organizar los números y gráficos dado que no todos ocupan lo mismo, con lo que no se podía dividir el conjunto de una forma equilibrada. No obstante, durante el proceso documentación se contempló la idea de incluir un quinto diplay para lograr una inclusión mejor tanto de los números como de los gráficos, pudiendo asignarle un display a cada uno.



Modelo 4 display, diseño actual – Modelo 5 display con modificaciones.

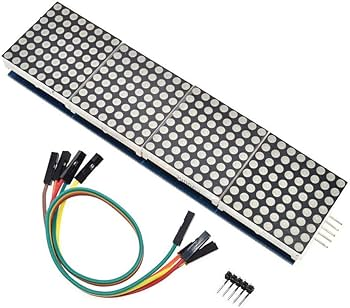
 



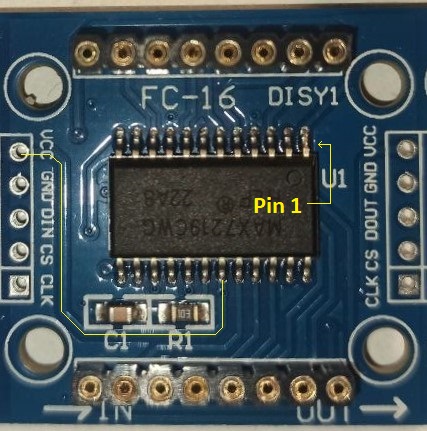
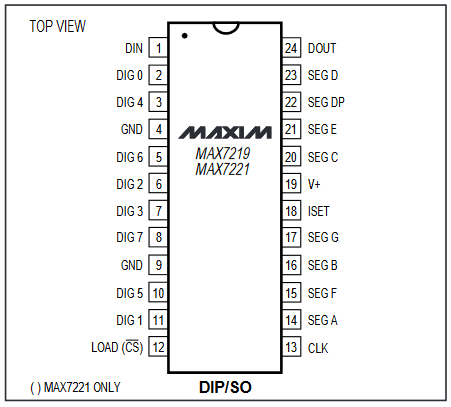
Modelo 5 display con modificaciones y mejor ajuste de jeroglíficos por cada sección de 8\*8

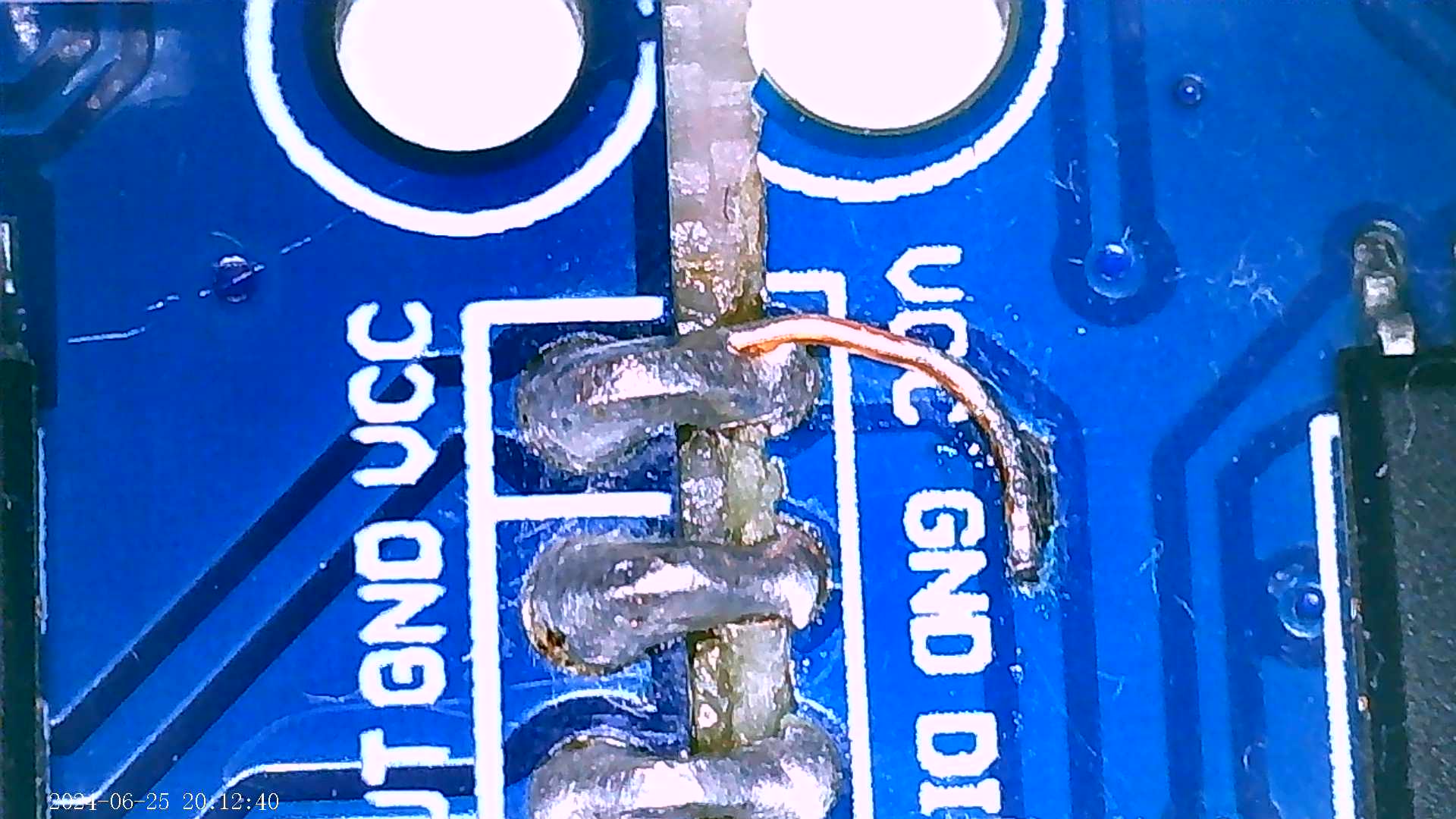
Modelo definitivo.

En la siguiente imagen se muestra el display formado por los cuatro módulos dot matrix de 8\*8.



Se usa una unidad como base del display, y uno de los módulos de otra unidad como quinto elemento. Durante el proceso de corte del último elemento de la segunda unidad, la continuidad de la pista de alimentación que conecta el pin 19 del integrado con el pin VCC de salida del cuarto módulo, se vio afectada, provocando que el quinto módulo no se encendiese. Se corrige la falta de continuidad con un puente hecho con un hilo.





El despliegue de las imágenes ocurre después de una transición de los números a las imágenes. Esta transición sucede como el giro a gran velocidad de cada display. Este efecto es el que se trata de emular en el método displayRollEffect().

### displayRollEffect();

void displayRolEffect() {

  mx.clear();

  byte fullLine = 0xFF;

  byte emptyLine = 0x00;

  int randomInt[MAX\_DEVICES];

  randomInt[0] = random(0, 8);

  for (int i = 1; i < MAX\_DEVICES;) {

    randomInt[i] = random(0, 8);

    for (int j = 0; i < MAX\_DEVICES;) {

      if (i != j) {

        if (randomInt[i] != randomInt[j]) {

          if (j++ == MAX\_DEVICES) {

            i++;

          }

        } else {

          break;

        }

      } else {

        j++;

      }

    }

  }

  for (int j = 0; j < 128; j++) {

    mx.setRow(4, 4, randomInt[0], fullLine);

    mx.setRow(3, 3, randomInt[1], fullLine);

    mx.setRow(2, 2, randomInt[2], fullLine);

    mx.setRow(1, 1, randomInt[3], fullLine);

    mx.setRow(0, 0, randomInt[4], fullLine);

    delay(15);

    mx.clear();

    for (int i = 0; i < MAX\_DEVICES; i++) {

      if (randomInt[i] == 7) {

        randomInt[i] = 0;

      } else {

        randomInt[i]++;

      }

    }

  }

}

Usamos la librería “MD\_MAX72xx.h” porque nos permite establecer el estado individual de cada led usando el método “setRow()”. El método setRow() puede constar de 2 a 4 parámetros en función de si trabajamos sobre todo el display o sobre un número de módulos en concreto. En este caso estaremos usando 4. El primer parámetro indica la dirección del display sobre el que se realizarán los cambios. Cuando se emplean 4 argumentos en este método la dirección vendrá dada por un rango de direcciones. Es decir, en lugar de usar un número como dirección del display sobre el que se trabaja, lo que se hace es establecer el primer número de los parámetros como dirección del primer display y el segundo número como dirección del último display del array de display sobre los que se trabajará. El tercer argumento constituye la fila del display que se va a modificar. El último parámetro es el que establece el estado de los leds de cada fila.

Este método establece el estado de los leds en función de un conjunto de 8 bit. Esto se logra a partir de facilitarle al método un número, en formato binario, de 8 dígitos. Un ejemplo sería el número “10101010”, donde los unos indicarán que los leds en esas posiciones de la fila se establecen como encendidos, y los ceros serán los leds que se encuentren apagados. Este número decimal ha de escribirse precedido o bien de la notación “0b” que indica que lo que se encuentra a continuación de dicha notación es un número binario, propia de diferentes idiomas de programación, o bien de la notación “B” propia de Arduino, cuya función es la misma que la de “0b”. No obstante, también podemos usar objetos de tipo byte dado que un byte es la representación de un conjunto de 8 bit. Veamos un ejemplo.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Figura 1 | | | | | | | |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Supongamos que deseamos mostrar la figura anterior en uno de los display. Lo primero que haremos será asignarles a los cuadrados grises, que simbolizan los leds apagados, el número 0, y a los cuadrados rojos que simbolizan los leds encendidos, el número 1. Hecho esto obtendremos la siguiente matriz.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Si se desea dibujar en un display la anterior figura, tendremos que llamar al método setRow 8 veces, y asignarle a cada fila cada uno de los números anteriores.

  byte rowThree = 0x18;

  byte rowTwo = 0x18;

  byte rowSecond = 0x18;

  byte rowOne = 0x18;

  mx.setRow(0, 7, 0b00000000);

  mx.setRow(0, 6, 0b01000010);

  mx.setRow(0, 5, 0b00100100);

  mx.setRow(0, 4, 0b00011000);

  mx.setRow(0, 3, rowThree);

  mx.setRow(0, 2, rowTwo);

  mx.setRow(0, 1, rowSecond);

  mx.setRow(0, 0, rowOne);

|  |  |
| --- | --- |
| 0x + Hexadecimal | Binario |
| 18 | 00011000 |
| 24 | 00100100 |
| 42 | 01000010 |
| 00 | 00000000 |

Las anteriores líneas imprimirán la figura 1. Como se observa se puede usar el valor binario con la notación “0b”, o el dato tipo byte cuyo valor es el número binario convertido a hexadecimal, precedido de la notación “0x”, (indicador hexadecimal).

|  |
| --- |
| **Nota 9:**  Es importante usar la notación binaria “0b” o “B” antes de escribir el valor en binario, de lo contrario el programa interpretará el valor introducido como un número decimal. Realizará la conversión de dicho número decimal a binario, y encenderá los leds en función de los últimos 8 dígitos del nuevo número binario.  Supongamos que introducimos el número 10101010 en el método sin la notación binaria:  10101010 interpretado como decimal = 100110100010000100010010 convertido a binario  00010010  0 1 2 3 4 5 6 7  Se encenderán los leds en las posiciones 3 y 6 de la fila. |

  mx.clear();

  byte fullLine = 0xFF;

  byte emptyLine = 0x00;

El efecto de transición desde la cuenta atrás hasta los gráficos se emula haciendo correr una línea horizontal de leds encendidos, de arriba abajo, a gran velocidad, como si estuvieran corriendo las placas del display split-flap. Empezamos borrando todo lo que esté impreso en el display. A continuación, declarando las variables de tipo byte que establecen una línea encendida y una línea apagada.

00000000 binario = 0x00 Hexadecimal

11111111 binario = 0xFF Hexadecimal

  int randomInt[MAX\_DEVICES];

  randomInt[0] = random(0, 8);

A continuación, creamos el array de elementos int que se usarán como posición inicial de cada una de las líneas de cada módulo. El tamaño del array será igual al número de displays. Continuamos declarando el elemento número 0 del array como el resultado de la función random(0, 8). Generamos así el elemento 0 como un número aleatorio.

La intención de que el efecto de transición empiece desde posiciones diferentes. Si se estableciese la misma posición para cada display, el efecto general en el conjunto de display sería una única línea horizontal corriendo de arriba abajo. A continuación, se borra todo lo que se esté mostrando en los displays.

  for (int i = 1; i < MAX\_DEVICES;) {

    randomInt[i] = random(0, 8);

    for (int j = 0; i < MAX\_DEVICES;) {

      if (i != j) {

        if (randomInt[i] != randomInt[j]) {

          if (j++ == MAX\_DEVICES) {

            i++;

          }

        } else {

          break;

        }

      } else {

        j++;

      }

    }

  }

El anterior bloque se encarga de generar x cantidad de números aleatorios, no coincidentes, donde x es el número de módulos -1. Se dice que la cantidad de números a generar es igual al número de módulos -1, porque el número aleatorio para la posición del array número 0 ya ha sido creado previamente.

Se empieza con un bucle for sin incremento fijado en la declaración, empezando por i = 1. A continuación, se genera un número aleatorio y se guarda en el array randomInt[i]. El incremento del primer array ocurrirá cuando el número aleatorio para la posición i no coincida con ninguno de los otros números aleatorios generados previamente. El segundo bucle for se encargará de iterar por todo el array buscando alguna coincidencia entre los elementos del mismo.

El cuerpo del segundo bucle for constará de una estructura condicional if-else. Se accederá al interior del condicional siempre que el valor actual de i y el valor actual de j no sean iguales, de lo contrario el valor de j incrementará. Esta estructura evita comparar randomInt[i] con randomInt[j] cuando i y j tienen el mismo valor. De hacerlo, la ejecución quedaría en bucle en la sentencia break que se explica a continuación.

Una vez dentro del condicional, se evaluará si los valores de randomInt[i] y randomInt[j] son iguales. Si son iguales quiere decir que existen dos números aleatorios coincidentes, en cuyo caso se lanza la sentencia break para poder iterar de nuevo en el primer bucle sobre el mismo valor de i. Con esto se logra poder generar un nuevo número aleatorio para dicho valor de i, para el que se lanzará, de nuevo, el bloque de comprobación de aleatoriedad. En caso de que los valores de randomInt[i] y randomInt[j] no sean iguales, se procede a comprobar si el valor de j, después de haber sufrido un incremento, es igual al número máximo de dispositivos. Si el valor de j es igual al número máximo de dispositivos, quiere decir que ya se ha comprobado si existen coincidencias entre los elementos del array y el elemento [i] del array, en cuyo caso se procede a incrementar el valor de i.

El siguiente bloque será el encargado de iterar x número de veces para que se produzca el desplazamiento de las líneas de leds encendidos en los displays. Encontramos un bucle for que iterará 128 veces. El número 128 surge de repetir 16 veces un ciclo completo de desplazamiento de los leds a lo largo de las 8 filas del display. La primera iteración del bucle empieza imprimiendo en cada display del conjunto, en la fila aleatoria generada con anterioridad para cada display, una línea completa. Después de esperar 15 milisegundos, se borra la totalidad de cada display mediante el método mx.clear(), y por último se comprueba el valor de cada variable de posición aleatoria de cada display.

Las filas de un display se cuentan desde la primera fila por la parte de superior, siendo dicha línea la línea 0, hasta la última por la parte inferior, siendo esta la línea 7. Cada display tendrá un bloque condicional que evaluará, después de haber sido impresa cada línea, si la posición de esta última línea es la posición 7, en cuyo caso se establecerá como 0 el valor de la posición. En caso de que la última posición no sea la posición 7, se le añadirá una unidad al contador de posición permitiendo que la línea de leds encendidos se desplace por el display en cada iteración. Esto permite que haya una continuidad, como si hubiera un eje interior que está girando y que lo último que sale por la parte inferior del display, vuelve a la parte frontal y superior del mismo.

Iteraciones:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

La anterior imagen muestra un ciclo completo para un display, cuya posición inicial aleatoria es la 3, es decir, la cuarta fila empezando por arriba. Al llegar a la quinta iteración del bucle for, es decir, cuando el valor de la posición de la línea es igual a 7, el valor de esta posición se reiniciará a 0. En la sexta iteración del bucle for, la línea de leds encendidos se encontrará en la posición 0. De esta forma, en las siguientes dos iteraciones se habrá conseguido mostrar la línea de leds encendidos a lo largo de las ocho posiciones del display.

**2.1.1.1 Fin de la explicación de displayRollEffect();**

Después de llamar al método displayRolEffect(), crearemos cinco arrays de elementos de tipo byte. Cada uno de estos arrays está formado por ocho elementos de tipo byte, cada uno de estos arrays supone la configuración de leds encendidos y apagados de cada línea.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **byte firstHieroglyph[8]** | | | | | | | | | **byte secondHieroglyph[8]** | | | | | | | | |
|  |  |  |  |  |  |  |  | **[0] = 0x00** |  |  |  |  |  |  |  |  | **[0] = 0x00** |
|  |  |  |  |  |  |  |  | **[1] = 0x10** |  |  |  |  |  |  |  |  | **[1] = 0x30** |
|  |  |  |  |  |  |  |  | **[2] = 0x28** |  |  |  |  |  |  |  |  | **[2] = 0x48** |
|  |  |  |  |  |  |  |  | **[3] = 0x28** |  |  |  |  |  |  |  |  | **[3] = 0x50** |
|  |  |  |  |  |  |  |  | **[4] = 0x28** |  |  |  |  |  |  |  |  | **[4] = 0x40** |
|  |  |  |  |  |  |  |  | **[5] = 0x20** |  |  |  |  |  |  |  |  | **[5] = 0x20** |
|  |  |  |  |  |  |  |  | **[6] = 0x20** |  |  |  |  |  |  |  |  | **[6] = 0x18** |
|  |  |  |  |  |  |  |  | **[7] = 0x00** |  |  |  |  |  |  |  |  | **[7] = 0x00** |
| **byte thirdHieroglyph[8]** | | | | | | | | | **byte fourthHieroglyph[8]** | | | | | | | | |
|  |  |  |  |  |  |  |  | **[0] = 0x00** |  |  |  |  |  |  |  |  | **[0] = 0x00** |
|  |  |  |  |  |  |  |  | **[1] = 0x10** |  |  |  |  |  |  |  |  | **[1] = 0x70** |
|  |  |  |  |  |  |  |  | **[2] = 0x28** |  |  |  |  |  |  |  |  | **[2] = 0x28** |
|  |  |  |  |  |  |  |  | **[3] = 0x28** |  |  |  |  |  |  |  |  | **[3] = 0x24** |
|  |  |  |  |  |  |  |  | **[4] = 0x28** |  |  |  |  |  |  |  |  | **[4] = 0x2a** |
|  |  |  |  |  |  |  |  | **[5] = 0x10** |  |  |  |  |  |  |  |  | **[5] = 0x15** |
|  |  |  |  |  |  |  |  | **[6] = 0x7c** |  |  |  |  |  |  |  |  | **[6] = 0x2e** |
|  |  |  |  |  |  |  |  | **[7] = 0x00** |  |  |  |  |  |  |  |  | **[7] = 0x00** |
| **byte fifthHieroglyph [8]** | | | | | | | | |  | | | | | | | | |
|  |  |  |  |  |  |  |  | **[0] = 0x00** |
|  |  |  |  |  |  |  |  | **[1] = 0x40** |
|  |  |  |  |  |  |  |  | **[2] = 0x20** |
|  |  |  |  |  |  |  |  | **[3] = 0x10** |
|  |  |  |  |  |  |  |  | **[4] = 0x0e** |
|  |  |  |  |  |  |  |  | **[5] = 0x08** |
|  |  |  |  |  |  |  |  | **[6] = 0x08** |
|  |  |  |  |  |  |  |  | **[7] = 0x00** |

  for (int i = 0; i < 8; i++) {

    mx.setRow(4, 4, i, firstHieroglyph[i]);

    mx.setRow(3, 3, i, secondHieroglyph[i]);

    mx.setRow(2, 2, i, thirdHieroglyph[i]);

    mx.setRow(1, 1, i, fourthHieroglyph[i]);

    mx.setRow(0, 0, i, fifthHieroglyph[i]);

  }

A continuación, le asignaremos a cada línea de los display, cada uno de los elementos byte de cada array. Para ello empleamos un bucle for que itere 8 veces usando el método setRow, sobre cada línea “i”, aplicando el elemento byte de posición “i” dentro de cada array.

  if (getFailureState() == true) {

    digitalWrite(RELAY, HIGH);

  }

El siguiente condicional servirá para evitar que se active el relé en caso de que se lance el método systemFailureOn, en un contexto en el que el usuario lanzó, con anterioridad, el comando “failureDisabled”, en la consola de la aplicación del pc. Cuando se envía este comando, así como el comando “failureEnabled”, desde el bloque de texto de la aplicación de pc, donde el usuario escribe la contraseña para reiniciar la cuenta atrás, lo que se logrará será que la placa active el relé de control de luces o no, cuando se cargue el método.

ATENCIÓN

COMPRUEBA SI LA SIGUIENTE LÍNEA exitLoop = true; SIRVE PARA ALGO EN ESTE CONTEXTO

**2.1.1 Fin de la explicación de systemFailureOn();**

**2.1.2** **setResetValues();**

setResetValues(true);

void setResetValues(bool newResetValues) {

  resetValues = newResetValues;

}

Con este método establecemos como true el valor la variable resetValues. Esto se hace para poder reiniciar las variables principales tras acceder al método initiativeReset(), la próxima vez que un cliente se conecte al servidor. Este método será explicado más adelante.

LINKEA INITIATIVE RESET() CON SU MÉTODO

**2.1.2 Fin de la explicación de setResetValues();**

Tercer escenario. Cuando el valor de la variable signal es mayor o igual a 0. Este caso trata el escenario en el que la placa ha perdido la conexión. En primer lugar, se enciende el led de estado.

digitalWrite(STATE\_LED, HIGH);

A continuación, se llama al método tryConnection().

**2.1.3 tryConnection(pSsid, pNetworkPassword);**

void tryConnection(String ssidString, String wifiPasswordString) {

  String states[] = { "WL\_IDLE\_STATUS", "WL\_NO\_SSID\_AVAIL", "WL\_SCAN\_COMPLETED", "WL\_CONNECTED", "WL\_CONNECT\_FAILED",

                      "WL\_CONNECTION\_LOST", "WL\_DISCONNECTED" };

  int present = millis(), past = present, pastDot = present;

  bool iterateLoop = true;

  const char\* ssidAttempt = ssidString.c\_str();

  const char\* wifiPasswordAttempt = wifiPasswordString.c\_str();

  int networkState = WiFi.begin(ssidAttempt, wifiPasswordAttempt);

  bool firstAttempt = true;

  while (iterateLoop) {

    if (present - past >= 5000) {

      if (getResetValues() == false & firstAttempt == false) {

        Serial.println("Testtest2");

        systemFailureOn();

        setResetValues(true);

      }

      firstAttempt = false;

      present = millis();

      past = present;

      pastDot = present;

      networkState = WiFi.begin(ssidAttempt, wifiPasswordAttempt);

      Serial.print(":");

    } else {

      if (digitalRead(RESET\_BUTTON)) {

        Serial.println("\nNetwork reset requested.");

        ssidString = "network";

        wifiPasswordString = "network12345";

        Serial.println("\n----\*\* Network \*\*----\n" + ssidString + "\n" + wifiPasswordString + "\n----\*\*\*\*\*\*\*\*\*\*\*\*\*----");

        saveIpPreferences(false);

        saveNetworkPreferences(ssidString.c\_str(), wifiPasswordString.c\_str());

        ESP.restart();

      }

      if (present - pastDot >= SECOND) {

        pastDot = present;

        Serial.print(".");

      }

      present = millis();

      networkState = WiFi.status();

      if ((networkState == WL\_CONNECTED) && (WiFi.localIP().toString() != "0.0.0.0")) {

        String ip = WiFi.localIP().toString();

        if (ip.startsWith("192.168.1.") && !(ip.equals("192.168.1.138"))) {

          Serial.println("\nDefault ip: " + ip);

          IPAddress staticIP(192, 168, 1, 138);

          IPAddress gateway(192, 168, 17, 254);

          IPAddress subnet(255, 255, 255, 0);

          WiFi.config(staticIP, gateway, subnet);

        }

        iterateLoop = false;

      }

    }

  }

  Serial.println("\nConexion status = " + states[WiFi.status()] + ".\nIP address: " + WiFi.localIP().toString() + "\nSsid = " + ssidAttempt + ".\n");

  saveNetworkPreferences(ssidAttempt, wifiPasswordAttempt);

}

  Serial.println("\nConexion status = " + states[WiFi.status()] + ".\nIP address: " + WiFi.localIP().toString() + "\nSsid = " + ssidAttempt + ".\n");

  saveNetworkPreferences(ssidAttempt, wifiPasswordAttempt);

}

Este método solicita dos parámetros de tipo string que serán usados como nombre de la red a la que intentará conectarse, y como contraseña de la red. El objetivo del método es el de iterar constantemente hasta lograr la conexión, teniendo en cuenta todos los posibles escenarios que se pueden dar.

  String states[] = { "WL\_IDLE\_STATUS", "WL\_NO\_SSID\_AVAIL", "WL\_SCAN\_COMPLETED", "WL\_CONNECTED", "WL\_CONNECT\_FAILED",

                      "WL\_CONNECTION\_LOST", "WL\_DISCONNECTED" };

Empezamos declarando un array llamado states que recogerá los posibles estados de la conexión reportados por el método WiFi.status(). Es necesario declarar este array porque al final del tryConnection se imprime el resultado de la lectura de status(), sin embargo este método reporta un valor integer. Usaremos este valor integer como índice del array puesto que los elementos del mismo están colocados en el orden correspondiente a los valores reportados por status().

int present = millis(), past = present, pastDot = present;

Declaramos tres variables de tipo integer que nos servirán para poder medir la cantidad de tiempo transcurrido desde un determinado instante. Estas tres variables se comportarán entre sí de la misma manera que las variables [‘currentTime’ y ‘previousTime’](#currentTime_y_previousTime_explicación) descritas al inicio del documento, en la sección de explicación de elementos integer. ‘past’ será usada para medir el tiempo transcurrido con respecto a present para conocer si han pasado 5 segundos desde que se intentó establecer la conexión sin éxito, para poder realizar un nuevo intento en caso afirmativo. ‘pastDot’ será usada para medir el tiempo transcurrido con respecto a present para poder imprimir en la consola del IDE un punto por cada segundo transcurrido, o dos puntos llegado a los 5 segundos, cuando se vuelve a lanzar un intento.

bool iterateLoop = true;

La variable ‘iterateLoop’ servirá como elemento true para que el bucle principal del método se mantenga iterando hasta que se logre la conexión.

  const char\* ssidAttempt = ssidString.c\_str();

  const char\* wifiPasswordAttempt = wifiPasswordString.c\_str();

Declaración de los elementos ‘const char\*’ solicitados por la función ‘WiFi.begin(ssidAttempt, wifiPasswordAttempt);’ como nombre y contraseña de la red a la que se intenta conectar la placa. Usando el método ‘c\_str()’ de la clase String convierte el contenido del String en un String de tipo C-style acabado en el carácter null ‘\0’. Recordemos que en C no existe un tipo de dato específico para las cadenas de caracteres, sino que se declaran como arrays formados por los caracteres del String. Adicionalmente al array se le añade el carácter null ‘\0’ al final del de este. Este carácter es el que indica el tamaño del array y hasta donde se ha de leer el mismo.

Por otro lado, un puntero en C es una dirección en la memoria en la que se encuentra un dato del mismo tipo que el puntero, sin importar la dimensión del mismo. Es decir, un puntero de tipo char puede ser un puntero a una dirección en la memoria en la que se encuentra un char, o un array de char. Dado que un array del tipo C-style puede ser considerado, a su vez, como un puntero al primer elemento del mismo, en este apartado es indiferente si le facilitamos al método ‘WiFi.begin()’ un argumento de tipo ‘const char\*’ o un array de tipo C-style mediante una estructura como la siguiente.

Ejemplo:

  char wifiPasswordAttempt[wifiPasswordString.length()+1];

  for (int i = 0; i < wifiPasswordString.length(); i++){

    wifiPasswordAttempt[i] = wifiPasswordString.charAt(i);

  }

  wifiPasswordAttempt[wifiPasswordString.length()] = '\0';

El anterior bloque de código sustituiría la siguiente línea.

const char\* wifiPasswordAttempt = wifiPasswordString.c\_str();

Se crea un array de caracteres de tamaño igual a la dimensión del string que se desea convertir en string C-style. Al tamaño del array le sumamos uno para poder albergar todos los caracteres del string más el carácter null ‘\o’. A continuación, con un loop de tipo for iteramos n veces, donde n es la dimensión del string, y le asignamos a cada elemento del array cada carácter del string. Por último, agregamos al final del mismo el carácter null ‘\0’.

El carácter null se encontrará en la posición igual al tamaño del string. Esto es así porque si tenemos un string de dimensión 5 caracteres, y le sumamos 1, el array albergará todos los caracteres del string en las posiciones del 0 al 4 mediante el bucle for, y el carácter null en la posición tamaño del string, es decir, en la posición quinta.

En resumen, el método ‘WiFi.begin()’ espera una cadena de caracteres y para ello solicita un puntero de tipo char, es decir, una dirección en la memoria en la que se guarda una cadena de caracteres de tipo C-style.

int networkState = WiFi.begin(ssidAttempt, wifiPasswordAttempt);

Creamos la variable que servirá para almacenar el resultado devuelto del actual intento de conexión por parte de la función begin. Este número será el que sirva como índice del array states[] del que obtendremos el estado de red a imprimir.

bool firstAttempt = true;

Usaremos la variable booleana ‘firstAttempt’ para evitar que el método ‘tryConnection’ itere más de dos veces en caso de que haya una pérdida sincronía entre pc y placa. Se ha considerado dos veces como número máximo de iteraciones porque cada iteración dura en 5 segundos, e intentar recuperar la conexión durante más de 5 segundos es excesivo.

while (iterateLoop) {

[…]

}

Bucle principal, iterará mientras que iterateLoop sea igual a true.

    if (present - past >= 5000) {

      if (getResetValues() == false & firstAttempt == false) {

        Serial.println("Testtest2");

        systemFailureOn();

        setResetValues(true);

      }

      firstAttempt = false;

      present = millis();

      past = present;

      pastDot = present;

      networkState = WiFi.begin(ssidAttempt, wifiPasswordAttempt);

      Serial.print(":");

    }

En el bucle principal encontramos una estructura if cuya condición para que se acceda al interior del mismo es que el tiempo transcurrido de restarle la referencia de tiempo ‘past’ a la referencia actual del tiempo, sea mayor o igual a 5000 milisegundos. La función del condicional es la de reiterar el intento de conexión, así como controlar las circunstancias en las que se debe producir. Se establecen cinco segundos como condición de acceso debido a que es un periodo de tiempo óptimo para lograr una conexión. Es aconsejable repetir el intento de conexión pasado este periodo de tiempo debido a que en ocasiones un intento nuevo es más propenso a obtener un resultado positivo, que mantener un intento durante un periodo de tiempo largo. Si se da la condición, se accede al interior del condicional donde encontramos otra estructura if.

El primer condicional que encontramos, una vez dentro del if principal del bucle while, evaluará si el valor de ‘resetValues’ es igual a false, y si la variable ‘firstAttempt’ es igual a false. Se accede al interior del condicional en caso de que se cumplan ambas condiciones. La variable ‘resetValues’ se establece como false cuando se han perdido 5 ciclos de sincronía en la comunicación entre el cliente, el pc, y el servidor, la placa. El objetivo de esta condición en este contexto es el de indicar que no se está intentando establecer conexión en una situación normal, es decir, no es una situación en la que se alimenta la placa y el método tryConnection se lanza desde el ‘void setup()’, lo que se está haciendo es intentar recuperar la conexión perdida en la placa en un escenario en el que había un dialogo entre servidor y cliente.

Siguiendo este planteamiento, cuando resetValues sirve de bandera para indicar que se está intentando recuperar una conexión perdida, ‘firstAttempt’ sirve para indicar el número de intentos lanzados. El valor de la variable ‘firstAttempt’ se establece como false inmediatamente después de este condicional.

      firstAttempt = false;

De esta forma, al acceder al interior del condicional ‘if ((present - past >= 5000)‘ por primera vez después de que la placa haya perdido la conexión mientras existía un dialogo servidor - cliente, se cumplirá la condición ‘getResetValues() == false’ pero no se cumplirá la condición ‘firstAttempt == false’. Esto provocará que se siga ejecutando el resto del condicional padre, estableciendo ‘firstAttempt’ como false, quedando ambas condiciones, ‘resetValues’ y ‘firstAttempt’, listas para generar un resultado true en el condicional ‘(getResetValues() == false & firstAttempt == false)’ y así acceder al interior del mismo al acabar el segundo intento de conexión (el primer intento fue el que se lanzó al declarar ‘networkState’, el segundo intento fue el lanzado al entrar por primera vez en el condicional ‘ if ((present - past >= 5000)’ sin llegar a entrar en el bloque ‘if (getResetValues() == false & firstAttempt == false) {‘).

Como ya se ha indicado, si el valor de resetValues y firstAttempt es igual a false se accederá al interior del condicional. Una vez dentro se lanzará el método [systemFailureOn](#systemFailureOn) para iniciar la secuencia de fallo del display y el relé, y el método [setResetValues](#setResetValues) para que la próxima vez que se conecte el cliente al servidor lo haga con los parámetros ‘timeCounter’, ‘previousCounter’ y ‘alarmModifier’ reiniciados a los valores por defecto.

Si no se da la condición necesaria para entrar en el condicional anteriormente citado, el programa seguirá ejecutándose y llegará a las siguientes líneas.

      present = millis();

      past = present;

      pastDot = present;

      networkState = WiFi.begin(ssidAttempt, wifiPasswordAttempt);

      Serial.print(":");

Actualizamos present mediante ‘millis()’, y past y pastDot asignándoles el valor de present. Se actualiza pastDot en este contexto porque de no hacerlo, se actualizaría en cuanto ‘while (iterateLoop) {‘ iterase de nuevo y se entrase en el apartado else del condicional ‘if ((present - past >= 5000) || networkReset) {‘. Es decir, la idea de fondo acerca de los puntos es, como ya se ha indicado anteriormente, que se imprima en la consola un punto por cada segundo transcurrido, y dos puntos “:” cuando se alcancen los 5 segundos y se lance un intento. El sistema empieza a contar tiempo mediante los mecanismos explicados, al llegar al quinto segundo se imprimirían dos puntos por el condicional “if ((present - past >= 5000) || networkReset) {“, saldría de condicional, el bucle while iteraría de nuevo, se entraría en la cláusula else, y sería entonces cuando se imprimiría el primer punto. Esto sucede porque la anterior vez que se actualizó pastDot fue cuando la cuenta iba por el cuarto segundo. Visualmente generaría secuencias “:…..” concatenadas, en lugar de la secuencia “:….” que es la que se espera.

A continuación, se actualiza el valor de networkState lanzando un nuevo intento de conexión, y se imprimen dos puntos en la consola “:”.

} else {

      if (digitalRead(RESET\_BUTTON)) {

        Serial.println("\nNetwork reset requested.");

        ssidString = "network";

        wifiPasswordString = "network12345";

        Serial.println("\n----\*\* Network \*\*----\n" + ssidString + "\n" + wifiPasswordString + "\n----\*\*\*\*\*\*\*\*\*\*\*\*\*----");

        saveIpPreferences(false);

        saveNetworkPreferences(ssidString.c\_str(), wifiPasswordString.c\_str());

        ESP.restart();

      }

      if (present - pastDot >= SECOND) {

        pastDot = present;

        Serial.print(".");

      }

      present = millis();

      networkState = WiFi.status();

      if ((networkState == WL\_CONNECTED) && (WiFi.localIP().toString() != "0.0.0.0")) {

        String ip = WiFi.localIP().toString();

        if (ip.startsWith("192.168.1.") && !(ip.equals("192.168.1.138"))) {

          Serial.println("\nDefault ip: " + ip);

          IPAddress staticIP(192, 168, 1, 138);

          IPAddress gateway(192, 168, 17, 254);

          IPAddress subnet(255, 255, 255, 0);

          WiFi.config(staticIP, gateway, subnet);

        }

        iterateLoop = false;

      }

    }

En caso de que no se cumpla la condicion necesaria para acceder al condicional “if (present - past >= 5000) {“ se accederá al bloque else. Lo primero que encontramos es un condicional que evaluará si el tiempo transcurrido desde el momento actual hasta el momento en el que se tomó la última referencia de tiempo para pastDot es igual o superior a un segundo.

       if (digitalRead(RESET\_BUTTON)) {

          Serial.println("\nNetwork reset requested.");

          ssidString = "network";

          wifiPasswordString = "network12345";

          Serial.println("\n----\*\* Network \*\*----\n" + ssidString + "\n" + wifiPasswordString + "\n----\*\*\*\*\*\*\*\*\*\*\*\*\*----");

          saveIpPreferences(false);

          saveNetworkPreferences(ssidString.c\_str(), wifiPasswordString.c\_str());

          ESP.restart();

        }

Una vez dentro del condicional encontramos otro if que leerá la entrada del botón de reseteo buscando un valor HIGH. Se accederá a este condicional cuando el valor de la lectura de la entrada del pulsador de reseteo sea igual a true.

Empezamos declarando los valores por defecto de las variables ‘ssidString’ y ‘wifiPasswordString’. A continuación, se imprime un mensaje en la consola indicando la red por defecto a la que se conectará en el siguiente intento de conexión. A continuación, lanzamos el método ‘saveIpPreferences’.

**2.1.3.1** **saveIpPreferences(false);**

void saveIpPreferences(bool anySavedIp) {

  preferences.begin("migration");

  preferences.putBool("anySavedIp", anySavedIp);

  if (anySavedIp) {

    preferences.putInt("firstOctec", octets[0].toInt());

    preferences.putInt("secondOctec", octets[1].toInt());

    preferences.putInt("thirdOctet", octets[2].toInt());

    preferences.putInt("fourthOctet", octets[3].toInt());

  }

  preferences.end();

}

Este método facilita la opción de guardar la ip actual en las preferencias del sistema para que, junto a otros mecanismos del código, que se explican en otros apartados, se pueda cargar la ip configurada manualmente como ip estática.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Nota 11:**  La librería **‘Preferences’** es una librería de Arduino enfocada a ESP32. Se le considera como un reemplazo de la librería EEPROM de Arduino.  Usa una parte de la memoria no volátil del ESP32 para almacenar información persistente a reinicios y eventos de perdida de alimentación.  Esta librería trabaja mejor almacenando grandes cantidades de datos pequeños, que almacenando cantidades más pequeñas de datos más grandes.  La forma de operar de esta librería es la siguiente:   * Primero se lanza el método.   bool begin(const char \* name, bool readOnly, const char\* partition\_label)  Se abre un espacio de almacenamiento en la memoria no volátil con el nombre facilitado como parámetro, desde una partición NVS, (non-volatile storage).   * El parámetro **‘name’** constituye el nombre del espacio en la memoria con el que se desea trabajar. * El parámetro **‘readOnly’** se encarga de abrir el dicho espacio con la configuración read-write, es decir, lectura y escritura, si se establece como false; y con la configuración read-only, es decir, solo lectura si se establece como true. Si se omite este parámetro el espacio se abrirá con la configuración de lectura y escritura. * El parámetro **‘partition\_label’** indica el nombre de la partición de la NVS en la que se ha de abrir el espacio. En caso de omitirse este parámetro se abrirá en la partición “nvs”.   Este método devuelve true si el espacio se abrió correctamente, y false en caso contrario.  Si el nombre del espacio facilitado no existe en la partición, se creará. Cualquier intento de escribir un dato en un espacio de almacenamiento configurado como read-only reportara un fallo.   * A continuación, se realizan las operaciones necesarias. En el presente proyecto se trabaja principalmente con los métodos put y get.   El método put se encarga de almacenar un dato bajo el nombre especificado.  size\_t putInt(const char\* key, int32\_t value)   * **‘Key’** constituye el nombre que se usará para identificar el valor que deseamos guardar, y qué más tarde se usará para recuperar dicho dato. Si el identificador ya existe sobrescribirá el valor del mismo, de lo contrario lo creará. * **‘value’** dato a almacenar que debe coincidir con el tipo de dato del método.   Este método devuelve el número de bytes almacenados para el tipo de dato si se llamó al método con éxito, o 0 si el resultado de llamar al método falló.  El método get devuelve el dato almacenado bajo el identificador facilitado como argumento del método.  int32\_t getInt(const char\* key, int32\_t defaultValue)   * **‘key’** nombre del elemento que se desea recuperar. * **‘defaultValue’** valor que se devolverá en caso de que no se logre recuperar ningún valor para el identificador facilitado porque no exista, o porque no se haya especificado ningún identificador. El tipo de este dato debe coincidir con el tipo del método.   El método reportará fallo en caso de que se pretenda obtener un dato sin indicar un identificador, o en caso de que se indique un identificador que no exista.  Para obtener un listado completo de los tipos de datos con los que se puede trabajar, o el resto de operaciones que se pueden realizar, se aconseja consultar la referencia al final de la nota.   * Por último, una vez hemos terminado de trabajar con las preferencias, debemos cerrarlas.   void end()  Después de cerrar el espacio de almacenamiento en la memoria, todo intento de acceso al mismo fallará.  Es altamente necesario invocar este método después de trabajar con las preferencias porque de no hacerlo pueden generarse comportamientos no deseados, y problemas varios. | | Información obtenida de esta [página](https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/api/preferences.html). | |

Empezamos abriendo las preferencias de nombre “migration”. A continuación, guardamos el valor de la variable booleana facilitada como parámetro del método, con el nombre anySavedIp.

Al alimentar la placa por primera vez, durante la ejecución del ‘void setup()’ y antes de llamar a ‘tryConnection()’, se llama a ‘preferencesIpConfig()’. En este método se consulta el valor de la variable ‘anySavedIp’. Si el valor de la misma es true se procede a un cambio en la configuración ip de la placa de acuerdo a las preferencias relacionadas con la ip guardada, para asignarle una ip estática; si el valor de ‘anySavedIp’ es igual a false, no se realizará ningún cambio y la asignación de la ip será dinámica, corriendo a cargo del router configurado en formato DHCP. Por ello, cuando se le realiza un cambio de ip a la placa, esta variable se establece como true, mientras que cuando se solicita un reseteo de parámetros de red, (ip y red), o cuando se solicita un cambio de red, esta variable toma el valor false.

La razón de porqué se establece este tipo de lógica de programación para pasar de una configuración de ip estática a una configuración de ip dinámica se explica más adelante dentro de este mismo apartado 2.1.3.1, en la sección en la que se explica la línea **‘ESP.restart();’**.

A continuación, encontramos un condicional cuya condición sea que el valor de ‘anySavedIp’ sea true. En caso de serlo, guardaremos cada elemento del array octets, casteado a tipo int, bajo las llaves ‘firstOctec’, ‘secondOctec’, ‘thirdOctet’ y ‘fourthOctet’ en las preferencias. Por último, cerramos las preferencias actualmente abiertas para evitar problemas

**2.1.3.1 Fin de la explicación de saveIpPreferences(false);**

Después de lanzar ‘saveIpPreferences’ se lanza el método ‘saveNetworkPreferences’,

**2.1.3.2** **saveNetworkPreferences(ssidString.c\_str(), wifiPasswordString.c\_str());**

void saveNetworkPreferences(const String newSsid, const String newNetworkPassword) {

  preferences.begin("migration");

  preferences.putString("ssid", newSsid);

  preferences.putString("networkPassword", newNetworkPassword);

  preferences.end();

}

Método que se usará para guardar en las preferencias un nuevo nombre y una nueva contraseña de red.

Iniciamos las preferencias con el método begin y la llave “migration”. A continuación lanzamos el método putString dos veces, guardando el nombre del nuevo ssid, newSsid, en la llave “ssid”, y la contraseña de dicha nueva red, newNetworkPassword, en la llave “networkPassword”. Finalizaremos cerrando las preferencias con el método end().

**2.1.3.2 Fin de la explicación saveNetworkPreferences(ssidString.c\_str(), wifiPasswordString.c\_str());**

ESP.restart();

Por último, reiniciamos el ESP. La razón por la que se reinicia el ESP es porque no se ha encontrado en la librería WiFi.h ninguna manera de pasar de un tipo de configuración de IP estática a una configuración de IP dinámica.

Es necesario poder pasar de una configuración de IP estática a una configuración de IP dinámica. Cuando se solicita el cambio de red es posible que la nueva red a la que se va a conectar la placa no se encuentre en el mismo rango que la red en la que se encontraba conectada. En este caso, aun si se lanza el método ‘saveIpPreferences(false);’ con el parámetro false, la placa seguiría en modo dirección IP estática, y mantendría la IP que se hubiese configurado con anterioridad. Para una explicación mas profunda de acerca de este tema se aconseja leer la [nota 4](#nota4).

Si deseamos, como es el presente caso, reiniciar los parámetros de red, lo mejor es reiniciar la placa y forzar la carga del código de cero. En este escenario, dado que se guardó en las preferencias el nombre y la contraseña de la red por defecto y se estableció ‘anySavedIp‘ como false, la placa se iniciará intentando conectarse a la red por defecto sin realizar ninguna configuración IP a través del método ‘preferencesIpConfig();’. Existe la posibilidad de repasar la librería para asegurar que no existe dicha opción, e incluso revisar si existen otras librerías que lo permitan. No obstante, por el momento se deja así.

El programa seguirá su curso si la lectura del pulsador de reseteo es LOW, en cuyo caso analizará si el tiempo transcurrido desde que se imprimió el anterior punto es superior a un segundo. De ser así, se actualizará el valor de la variable ‘pastDot’ al valor actual de la variable present, y se imprimirá un punto en la consola.

Al salir del anterior condicional se actualizará el valor de present y el valor de networkState. Si el valor de networkState es igual WL\_CONNECTED, (recordemos que WL\_CONNECTED es un elemento de la librería WiFi.h cuyo valor es 3, y que el método status() devuelve uno de todos los elementos indicadores de estado como WL\_CONNECTED, como valor de tipo int), y la IP actual de la placa, (en este momento, si networkState es igual a WL\_CONNECTED, la placa tiene una IP asignada), es distinta de “0.0.0.0” se accede al interior del condicional.

|  |
| --- |
| **Nota 12:**  Es necesario incluir en la estructura if la condición de que la ip sea distinta de 0.0.0.0 porque se ha probado que en ocasiones se le asigna dicha ip a la placa. No sería válido establecer únicamente la condición de que la placa reporte el estado “conectado”, porque en dicho escenario, donde la ip es la anteriormente citada, la comunicación sería nula. |

Una vez dentro del condicional crearemos un String ‘ip’ cuyo valor será la representación en formato String del reporte de IP del método ‘localIp()’. Se verificará si dicha IP está en el rango “192.168.1.” y si no es exactamente la IP “192.168.1.138”. Si la IP asignada cumple los requisitos, se lanzará un mensaje indicando la IP por defecto y se procederá a modificar la IP para asignar el host número 138.

Por último, se establece como false la variable iterateLoop, lo que permitirá salir del bucle while.

Una vez fuera del bucle while, se imprime el estado de la conexión, la dirección IP y el nombre de la Red. Por último, guardamos el nombre y la contraseña de la red en las preferencias por medio del método [saveNetworkPreferences](#saveNetworkPreferences).

* 1. **Fin de la explicación de connectionStateLed();**
  2. **Reseteo final.**

    if (getExitLoop() == true) {

      exitLoop = false;

      binaryClose();

    }

Se sale del apartado else comprobando si el valor de la variable ‘exitLoop’ es igual a true. Esto solo puede ocurrir cuando se ha finalizado una comunicación entre servidor y cliente. En tal caso, hay que reiniciar dicha variable para que vuelva a ser false en caso de que se conecte un nuevo cliente. De lo contrario, al aparecer un nuevo cliente, el bucle no iteraría y la comunicación servidor-cliente terminaría de forma abrupta. Por último, apagamos los leds del contador binario por medio del método binaryClose LINKEA BINARYCLOSE, dado que si se lanza el método [systemFailureOn](#systemFailureOn), lo único que debe quedar activo es el relé de control de luces y el display.

Métodos:

Getters y setters típicos:

Los setters típicos son aquellos que le asignan un valor a una variable por medio del parámetro del método. De igual manera, un getter típico será aquel que retorna el valor de dicha variable. No se produce ninguna modificación adicional sobre los mismos, y no se devuelve más que el valor de dicha variable.

A continuación, se muestran los getters y setters presentes en el proyecto.

* **int getMissConnectionCounter(** y **void setMissConnectionCounter(int newMissConnectionCounter) {**

int getMissConnectionCounter() {

  return missConnectionCounter;

}

void setMissConnectionCounter(int newMissConnectionCounter) {

  missConnectionCounter = newMissConnectionCounter;

}

* **int getAlarmModifier() {** y **void setAlarmModifier(int newAlarmModifier) {**

int getAlarmModifier() {

  return alarmModifier;

}

void setAlarmModifier(int newAlarmModifier) {

  alarmModifier = newAlarmModifier;

}

* **getConnectionLost() {** y **void setConnectionLost(int newConnectionLost) {**

int getConnectionLost() {

  return connectionLost;

}

void setConnectionLost(int newConnectionLost) {

  connectionLost = newConnectionLost;

}

* **bool getResetValues() {** y **void setResetValues(bool newResetValues) {**

bool getResetValues() {

  return resetValues;

}

void setResetValues(bool newResetValues) {

  resetValues = newResetValues;

}

* **bool getFailureState() {** y **void setFailureState(bool newFailureState) {**

bool getFailureState() {

  return failureState;

}

void setFailureState(bool newFailureState) {

  failureState = newFailureState;

}

* **getExitLoop() {** y **void setExitLoop(bool newExitLoop) {**

bool getExitLoop() {

  return exitLoop;

}

void setExitLoop(bool newExitLoop) {

  exitLoop = newExitLoop;

}

* **String getLogText() {** y **void setLogText(String newlogText) {**

String getLogText() {

  return logText;

}

void setLogText(String newlogText) {

  logText = newlogText;

}

* **String getIpChangePassword() {** y **void setIpChangePassword(String newIpChangePassword) {**

String getIpChangePassword() {

  return ipChangePassword;

}

void setIpChangePassword(String newIpChangePassword) {

  ipChangePassword = newIpChangePassword;

}

Getters y setters no típicos.

Los getters no típicos serán aquellos que devuelven un valor tras haber realizado algún tipo de operación.

* **int getMinutes() {**

Devuelve un int representando el resultado de dividir timeCounter, es decir, la cuenta total de segundos restantes, entre 60, para obtener el número de minutos que quedan para que termine la cuenta atrás.

El resultado de dicha cuenta se castea a int para que la parte restante de segundos, la parte decimal del resultado, sea igual a 0.

int getMinutes() {

  return int(timeCounter / 60);

}

* **int getSeconds() {**

Devuelve un int representando el resultado de restarle a timeCounter, la cuenta total de segundos restantes, el resultado de multiplicar los minutos restantes por 60. El int resultante representa la cantidad de segundos que quedan para termine la cuenta atrás.

int getSeconds() {

return timeCounter - (getMinutes() \* 60);

}

Valor de time counter 130:

130 – ((timeCounter / 60) \* 60)

130 – ((130 / 60) \* 60)

130 – (**2**~~.16~~ \* 60)

130 – 120 = 10

Segundos que quedan de acuerdo con la cantidad de minutos que quedan: 10

* **String getTimeString() {**

String getTimeString() {

  String stringSeconds = "00";

  String stringMinutes = "";

  String stringToPrint = "";

  int minutesCount = getMinutes();

  int secondsCount = getSeconds();

  int minTriggerRange = 5;

  if (minutesCount < minTriggerRange) {

    if (secondsCount < 10) {

      stringSeconds = "0" + String(secondsCount);

    } else {

      stringSeconds = String(secondsCount);

    }

  }

  if (minutesCount < 10) {

    stringMinutes = "00" + String(minutesCount);

  } else if (minutesCount < 100) {

    stringMinutes = "0" + String(minutesCount);

  } else {

    stringMinutes = String(minutesCount);

  }

  stringToPrint = stringMinutes + stringSeconds;

  return stringToPrint;

}

Devuelve un String cuyo valor es un número, de cinco dígitos, cuyo valor es la concatenación del número de minutos y segundos restantes, precedidos por la cantidad necesaria de ceros, siempre y cuando el número de minutos restantes sea inferior a 5.

Si quedan 5 minutos y 3 segundos, el valor devuelto por este método será un String cuyo valor será “00500”. Si el tiempo restante son 4 minutos y 2 segundos, el valor devuelto será “00402”. Gracias a este método se obtiene cada dígito a imprimir en cada módulo del display, por medio del método charAt(), de la clase String, en otros métodos.

  String stringSeconds = "00";

  String stringMinutes = "";

  String stringToPrint = "";

Empezamos declarando los elementos String. Se declara el valor de la variable stringSeconds como “00” dado que, en caso de que no se cumplan las condiciones necesarias, el valor de esta no se verá modificado. Esta variable representa la parte del string final encargada de los segundos, es decir, los dos últimos dígitos del string. stringMinutes se encarga de los minutos, representados por los tres primeros dígitos del string final. A diferencia de stringSeconds, esta variable, así como stringToPrint, van a ver su valor modificado en cualquiera de los casos, por eso no se inicializa con ningún valor. stringToPrint es el string compuesto por la concatenación del string stringMinutes más stringSeconds, que será devuelto al final del método.

  int minutesCount = getMinutes();

  int secondsCount = getSeconds();

  int minTriggerRange = 5;

Se continua por la declaración de los elementos int. ‘minutesCount’ y ‘secondsCount’ se inicializarán con el número minutos y segundos restantes, respectivamente, de acuerdo al valor actual de timeCounter. Usaremos estas variables, junto a minTriggerRange, para realizar las operaciones pertinentes que harán posibles que la concatenación de minutos y segundos se presente con la cantidad de ceros correspondientes. minTriggerRange servirá de bandera para indicar desde qué minuto ha de mostrarse la cuenta atrás de los segundos. En la serie Lost, los dos dígitos de los contadores de segundos solo empiezan a correr cuando el contador de tiempo pasa de mostrar 5 minutos a mostrar 4 minutos y 59 segundos, y hasta el segundo 0.

  if (minutesCount < minTriggerRange) {

    if (secondsCount < 10) {

      stringSeconds = "0" + String(secondsCount);

    } else {

      stringSeconds = String(secondsCount);

    }

  }

En el primer condicional evaluaremos si el valor actual de “minutesCount” es menor al valor a “minTriggerRange”, es decir, si es menor a 5 minutos accedemos al interior del condicional. Una vez dentro, encontramos un segundo condicional que evaluará si el valor actual de “secondsCount” es inferior a 10. El primer condicional sirve para evaluar si se debe imprimir el valor de los segundos o no, siempre que la cuenta actual de minutos restantes sea inferior a 5, se imprimirán los segundos. El segundo condicional sirve para imprimir de una u otra forma los segundos en función del valor actual de los mismos. Si la cuenta de segundos se forma a partir de dos dígitos, es decir, desde el segundo 59 hasta el segundo 10, se casteará a string y se le asignará a la variable stringSeconds el valor actual de los segundos, por medio del apartado else. Si el valor de los segundos es inferior a 10, quiere decir que la cuenta de segundos se forma a partir de un único digito, es decir, del segundo 0 al 9. En este escenario accedemos al interior del segundo condicional. Le asignaremos a la variable stringSeconds la concatenación del valor de la variable secondsCount, casteada a String, más el carácter 0. Esto se hace para que los segundos del 0 al 9 se muestren con un cero delante, para que en el display se muestre un dígito en cada módulo de pixels.

  if (minutesCount < 10) {

    stringMinutes = "00" + String(minutesCount);

  } else if (minutesCount < 100) {

    stringMinutes = "0" + String(minutesCount);

  } else {

    stringMinutes = String(minutesCount);

  }

La estructura anterior es igual a la anterior pero aplicada a la cuenta de minutos y con un condicional adicional.

Si la cuenta de minutos es inferior a 10 minutos, la variable stringMinutes tomará el valor de la concatenación del valor del casteo a string de la cuenta actual de minutos, más los caracteres “00”. Esto se hace para que la cuenta de minutos se muestre con dos ceros delante a fin de que los minutos cubran los tres módulos de pixeles asignados a los minutos.

|  |  |
| --- | --- |
| Tiempo restante | Tiempo a mostrar en el display |
| 101 minutos y 23 segundos. | 1 0 1 0 0 |
| 93 minutos y 40 segundos | 0 9 3 0 0 |
| 5 minutos y 0 segundos | 0 0 5 0 0 |
| 4 minutos y 50 segundos | 0 0 4 5 0 |
| 0 miuntos y 9 segundos | 0 0 0 0 9 |

stringToPrint = stringMinutes + stringSeconds;

  return stringToPrint;

Por último, le asignaremos a stringToPrint la concatenación de stringMinutes más stringSeconds, y retornaremos el valor de stringToPrint.

Para un tiempo de 4 minutos y 50 segundos devolveremos el string “00450”.

* **String** **setHelp(){**

String setHelp() {

  String stringHelp = "<br>~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~";

  stringHelp = stringHelp + "<br><br>&ensp;Type the keyword followed by a '-' character, and the right command, to perform an action:<br><br>";

stringHelp = stringHelp + "&ensp;&ensp;\* Password change:<br>&ensp;&ensp;&ensp;&ensp;- keyword = 'setpassword'; command = new password.<br><br>";

stringHelp = stringHelp + "&ensp;&ensp;\* Ip change:<br>&ensp;&ensp;&ensp;&ensp;- keyword = current password; command = new ip.<br><br>";

stringHelp = stringHelp + "&ensp;&ensp;\* System failure state:<br>&ensp;&ensp;&ensp;&ensp;- keyword = 'systemfailure'; command = on - off.<br><br>";

stringHelp = stringHelp + "&ensp;&ensp;\* Network set up:<br>&ensp;&ensp;&ensp;&ensp;- keyword = 'changenetwork'; command = [network ssid]-[network password].<br><br>";

stringHelp = stringHelp + "&ensp;&ensp;\* To clear the log:<br>&ensp;&ensp;&ensp;&ensp;- keyword = 'cls'.<br><br>&ensp;&ensp;Example: changenetwork-myhomenetwork-password1234<br><br>";

stringHelp = stringHelp + "~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~\*~<br><br><br>";

return stringHelp;

}

Este método se lanzará en la fase de carga de funciones en el void setup. Su función es la de asignarle a la variable stringHelp el texto a imprimir en el servicio webservice, tras aplicarle un formato de tipo HTML.

Configuración html

A continuación, se procede a explicar las etiquetas y estructuras usadas en el proyecto.

* **<HTML>…</HTML>**

Establece la totalidad de la página. La página está formada por la cabecera y el cuerpo.

* **<HEAD> … </HEAD>**

Delimita y engloba la cabecera de la página. El contenido de la cabecera no se muestra en la ventana, sirve para establecer diferentes ajustes de la página.

* **<TITLE>…</TITLE>**

El contenido de esta etiqueta se mostrará en la barra del título de la pestaña del navegador.

* **<BODY>…</BODY>**

Define el cuerpo de la página. Es el cuerpo el que contiene todos los elementos visibles del documento HTML. Como el texto, imágenes, hipervínculos, listas etc.

* **<p>…</p>**

El texto en su interior se establece como un párrafo.

* **style="font-size:30px; font-family:courier; color:red; background:#000000; padding:0; border:0; text-decoration: underline"**

**Font-size:Xpx;** estructura CSS que establece el tamaño del texto al tamaño de X pixel. También se puede emplear la sentencia “**<FONT SIZE="+3"></FONT>”**, no obstante, esta fórmula es incompatible con HTML5.

**Font-family:X;** Establece el tipo de fuente a la fuente X.

**Color:red/#ff0000** Establece el color de la fuente al indicado como parámetro mediante el string con el nombre del color en inglés, o mediante el código de color hexadecimal.

**Background:black/#000000** Establece el color de la fuente al indicado como parámetro mediante el string con el nombre del color en inglés, o mediante el código de color hexadecimal.

**border:X** Establece el borde del elemento al tamaño especificado. Establecer dicho parámetro a 0 eliminará el borde.

**text-decoration: X** Es una propiedad del tipo shorthand CSS, que establece la apariencia de líneas decorativas en el texto. Existen múltiples parámetros asociados a esta propiedad.

|  |
| --- |
| **Nota 14:**  Las propiedades shorthand CSS son propiedades CSS que permiten establecer a la vez varias  propiedades CSS. |

**Underline** genera una línea por debajo del texto.

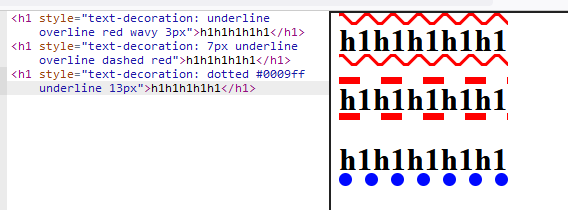
**Overline** genera una línea por encima del texto.

**Dotted** establece la línea asociada al parámetro como línea de puntos.

**Dashed** establece la línea asociada al parámetro como línea de rayas.

**Color/HexColor** establece la línea asociada al parámetro con el color facilitado como string, o con el color facilitado como valor hexadecimal.

**Xpx** Establece el grosor de la línea asociada a dicho parámetro.



Como se observa en el anterior ejemplo, da igual en qué orden se establezcan los parámetros del atributo. De igual forma, no importa si se establecen al inicio o al final del campo de parámetros del atributo. La única consideración a tener en cuenta será la de establecer los parámetros underline y overline juntos y sin ningún atributo entre medias.

* **<U>…</U>**

Subraya el texto en su interior.

* **<B>…</B>**

Resalta el texto en negrita.

* **<hr>**

Crea una línea divisoria horizontal que atraviesa la pantalla de lado a lado. Si se le añade el parámetro **color=""**, podremos establecer el color de la misma añadiendo un color en formato string o en formato hexadecimal. En los siguientes ejemplos se genera una línea de color verde de distinta tonalidad.

<hr color="#00FF00">

<hr color="green">

* **<br>**

Salto de línea.

* **<a href=”link”>…</a>**

Define un hipervínculo a otra página. El hipervínculo se fija en el espacio entre comillas. Se le puede asignar parámetros de estilo mediante el tag style.

Click <a href="/?ui=systemfailure-on" style="color: #08fcc3">here</a> to triguer the failure system mode on.

El anterior ejemplo establece la palabra here como un hipervínculo a la url relativa ="/?ui=systemfailure-on”, y establece su color al valor hexadecimal #08fcc3. Imagino que el término “href” viene de la **H**yperlink **REF**erence.

|  |
| --- |
| **Nota 13:**  Cuando se usa la estructura **<a href=”link”>…</a>**, sise establece un comando del tipo “/?ui=systemfailure-on”, se está pasando una url relativa. Es decir, se facilita una url parcial que no incluye ni dominio, ni protocolo y el redireccionamiento se produce en relación a la url de la página actual. Si la dirección actual del servidor es, por ejemplo, la “192.168.1.1”, al introducir la dirección relativa anteriormente citada, la redirección se produciría a la url final “192.168.1.1/?ui=systemfailure-on”.  Si el comando es una url completa, como por ejemplo “https://www.google.com/”, estaremos tratando una url absoluta. A continuación, tenemos una url completa que incluye el protocolo “http://” o “https://” marcado en amarillo, el nombre del dominio o la ip marcado en verde, y la ruta completa de recursos marcado en azul.  https://www.google.com/imghp?hl=es&authuser=0&ogbl |

* **<form action="X" method="X"></form>**

Unelemento form es una sección de un documento html encargado de recoger la información facilitada por un usuario. La información recogida es enviada a un servidor. Un form contiene elementos interactivos que sirven para que el usuario pueda facilitar su información. Estos elementos pueden ser bloques de texto, checkboxes, menús desplegables, botones, etc. Encontramos dos atributos imprescindibles en un elemento form:

**Action:**

Define la dirección a la que enviar la información lanzada por el usuario. Por lo general el valor de action es una URL del servidor que tratará dicha información.

Si establecemos una dirección relativa en este campo, la información se enviará a la concatenación de la dirección actual más la dirección relativa facilitada, tal y como se ha comentado en la nota 13. Si deseamos que la información se envíe se gestione desde la propia página, habrá que dejar este campo vacío o no incluirlo. De esta forma, si la información se envía desde la dirección principal del servidor, siguiendo el ejemplo de la nota 13, desde la dirección 192.168.1.1, la página que se cargará será esta misma.

**Method:**

Atributo usado para especificar el método HTTP usado para enviar la información. Existen dos tipos de métodos HTTP. El método **GET** concatena la información a continuación de la ip absoluta establecida por el atributo action. Si la dirección establecida en el atributo action es una ip relativa, por ejemplo, “192.168.1.1/user/data”, y el objeto input de la estructura form tiene como parámetro name “id” y de value “help”, la url resultante al presionar el botón help será: 192.168.138.138/192.168.138.138/user/data/?ui=help. En caso de presionar una segunda vez el botón help la url resultante será: 192.168.138.138/192.168.138.138/user/data/192.168.138.138/192.168.138.138/user/data/?ui=help. Esto ocurre por la razón que ya se ha explicado anteriormente en la nota 13. Get concatena el input enviado por el usuario a la url que aparece en el atributo action, que en este caso es una url relativa que se resuelve en torno a la url actual de la página, generando una cadena de concatenaciones.

Los datos enviados con el método GET tienen un límite de 2048 caracteres y han de ser de tipo ASCII. Si se excede dicho límite, los datos se verían recortados y no se enviaría todo el contenido al backend. Todo dato que se envíe será visible desde la URL, incluso los datos password que aparecen censurados en los inputs de tipo password. Get permite que el navegador agregue a la caché los resultados del envío de datos, así como la posibilidad de que el usuario guarde dicha búsqueda en marcadores.

El método **POST** envía los datos al servidor para que sea este quien los procese. No tiene limitación de caracteres ni restricción en el tipo de los mismos. La gestión de datos resulta más segura que con el método get, dado que envía los datos al servidor y no los concatena a ninguna url donde quedarían expuestos. No obstante, enviar una forma dos veces generará el envío de datos de forma duplicada, lo cual puede ser un problema si el elemento form está vinculado a un proceso de compra o a una membresía o a cualquier otra acción que solo deba ocurrir una única vez. Es por ello que cuando se usa este tipo de método no se puede guardar los resultados de una búsqueda en los marcadores de un navegador.

* **Inputs**

Son elementos usados que permiten la creación de controles interactivos para que el usuario pueda introducir y enviar datos al servidor. Hay una gran variedad de ellos. Algunos de los inputs con los que se puede trabajar son los elementos para desplegar un menú de selección de color dentro de una gama cromática, de tipo campo de fecha, de tipo oculto para que no se muestren al usuario, de tipo numérico para que se desplegue un teclado numérico en aquellos teclados dinámicos que lo permitan, etc.

**<button type="submit" name="ui" value="systemfailure-on">here</button>**

Input de tipo button. Genera un botón con el **string** que figura entre el tag de apertura y el tag de cierre. El atributo type establecerá el comportamiento del botón. Este campo puede tomar el valor “button” haciendo que se comporte como un botón clicable; el valor “submit” enviando toda la información del form al servidor; o el valor “reset” reseteando todos los datos presentes en el elemento form. El comando que se enviará al presionar el botón se establecerá en el campo value, y dicho comando se identificará en la url o en el cuerpo del mensaje enviado al servidor con el identificador establecido en el campo name.

**<input type="text" name="ui" placeholder="Enter your command" autofocus required>**

Input de tipo texto de una sola línea. Creará un campo de texto donde el usuario podrá introducir datos que enviar mediante un elemento submit. El elemento name cumplirá la misma función que en el anterior elemento button explicado. El campo placeholder establece un texto a modo de breve indicación para el usuario, en el campo de texto, para indicar qué tipo de información espera el campo.

Existen varios atributos adicionales de gran utilidad como **readonly** que establece el campo a modo de solo lectura; **size** que establece la cantidad de caracteres del campo; **spellcheck** que habilita la opción de comprobación de ortografía.

* **<label for="X">X</label>**

**RECORDATORIO PARA MI: Realiza los cálculos de la resistencia del transistor, teniendo en cuenta que la carga en el colector es la bobina del relé, que se alimenta a 6V y cuya resistencia es 88Ohms. Realiza la ley de ohm para sacar la Ic. Usa una de las formulas:**

* Sin aplicar el factor de protección sobre Ib:

RBE = (Voltaje en base - 0.7) / (IC / β (También conocido como hFE)

* Con factor de protección sobre Ib:

RBE = (Voltaje en base - 0.7) / 3 \* (IC / β (También conocido como hFE)

**<head>**

**<title>Dharma Initiative's web service</title>**

**<style>**

**body {**

**background-color: #000000;**

**color: #bdbcbc;**

**font-family: courier**

**}**

**.customButton {**

**background: #000000;**

**color: #bdbcbc;**

**padding: 0;**

**border: 0;**

**text-decoration: underline**

**}**

**.customTextBox {**

**background: #000000;**

**color: #bdbcbc;**

**border-color: #bdbcbc**

**}**

**hr {**

**color: #ababab;**

**}**

**</style>**

**</head>**

**<body>**

**<h1>Dharma initiative project web service. Welcome.</h1>**

**Low performance web service.**

**<hr>**

**<form method="POST">**

**<label for="getHelp">To get help enter 'help' in the text box, or click </label>**

**<button class="customButton" name="ui" value="help">here</button>**

**<label for="getHelp">.</label>**

**<br>**

**</form>**

**<hr>**

**<form method="POST">**

**<label for="Click">Click </label>**

**<button class="customButton" type="submit" name="ui" value="systemfailure-on">here</button>**

**<label for="failureSystemOff"> to triguer the system failure mode on.</label>**

**<br>**

**<label for="Click">Click </label>**

**<button class="customButton" type="submit" name="ui" value="systemfailure-off">here</button>**

**<label for="failureSystemOff"> to triguer the system failure mode off.</label>**

**</form>**

**<hr>**

**<form method="POST">**

**<input class=customTextBox type="text" id="textbox" name="ui" placeholder="Enter your command" autofocus required>**

**<input class="customButton" type="submit" value="Send">**

**</form>**

**<p align="left"> )====="; html += getLogText() + R"=====(**

**</p>**

**</body>**

<label for="clicked">Click in this text to trigger the checkbox.</label>

<input type="checkbox" id="clicked" name="checkbox">

<br>

<label accesskey="c">Click in this text to trigger the checkbox.<input type="checkbox" name="checkbox"></label>

**Add the lang Attribute**

You should always include the lang attribute inside the <html> tag, to declare the language of the Web page. This is meant to assist search engines and browsers.

**Example**

<!DOCTYPE html>  
<html lang="en-us">  
<head>  
  <title>Page Title</title>  
</head>  
<body>  
  
<h1>This is a heading</h1>  
<p>This is a paragraph.</p>  
  
</body>  
</html>

**Meta Data**

To ensure proper interpretation and correct search engine indexing, both the language and the character encoding <meta charset="*charset*"> should be defined as early as possible in an HTML document:

<!DOCTYPE html>  
<html lang="en-us">  
<head>  
  <meta charset="UTF-8">  
  <title>Page Title</title>  
</head>

**Continúa revisando esta página en relación a los form sin action y sin method, para poder explicar correctamente la utilidad de action**

**https://html.form.guide/action/html-form-without-action-and-method/**

[**https://html.onlineviewer.net/**](https://html.onlineviewer.net/) **Visualizador HTML**

HTTP/1.1 200 OK

Content-type:text/html

<head>

  <title>Dharma Initiative's web service</title>

</head>

<body style="background-color:#000000; color: #bdbcbc; font-family:courier">

  <p style="font-size:30px"> Dharma initiative project web service. Welcome. </p> Low performance web service.

  <hr color="#ababab">

  <form method="POST">

    <label for="getHelp">To get help enter 'help' in the text box, or click </label>

    <button style="background:#000000; color:#bdbcbc; padding:0; border:0; text-decoration: underline" type="submit" name="ui" value="help">here</button>

    <label for="getHelp">.</label>

    <br>

  </form>

  <hr color="#ababab">

  <form method="POST">

    <label for="Click">Click </label>

    <button style="background:#000000; color:#bdbcbc; padding:0; border:0; text-decoration: underline" type="submit" name="ui" value="systemfailure-on">here</button>

    <label for="failureSystemOff"> to triguer the system failure mode on.</label>

    <br>

    <label for="Click">Click </label>

    <button style="background:#000000; color:#bdbcbc; padding:0; border:0; text-decoration: underline" type="submit" name="ui" value="systemfailure-off">here</button>

    <label for="failureSystemOff"> to triguer the system failure mode off.</label>

  </form>

  <hr color="#ababab">

  <form method="POST">

    <input type="text" name="ui" placeholder="Enter your command" autofocus required style="background:#000000; color:#bdbcbc; border-color:#bdbcbc">

    <input style="background:#000000; color:#bdbcbc; border:0; text-decoration: underline" type="submit" value="Send">

  </form>

  <p align="left">

  </p>

</body>