

**Facultad de  
Informática**



**UNIVERSIDAD  
NACIONAL  
DE LA PLATA**

## INFORME FINAL

Remera LED controlado por web

Taller de Proyecto 2 (I118)

García, Agustín (824/0)  
Sanchez, Agustín (939/2)  
Ternouski, Nahuel (756/6)

Diciembre de 2017

# Índice

<b>1. Propuesta original</b>	<b>1</b>
<b>2. Correcciones/cambios de la propuesta</b>	<b>1</b>
2.1. Indicadas por la Cátedra . . . . .	1
2.2. Definidas por el avance y disponibilidad . . . . .	1
<b>3. Descripción y documentación general</b>	<b>2</b>
3.1. Wemos D1 . . . . .	2
3.2. MAX7219 . . . . .	3
3.2.1. Descripción general . . . . .	3
3.2.2. Configuración de pines . . . . .	4
3.3. Alimentación . . . . .	5
3.4. Conexión del Wemos D1 con el MAX7219 . . . . .	5
3.5. Conexión del MAX7219 con su módulo de 8x8 LEDs . . . . .	7
3.6. Concatenación de varios módulos de LEDs . . . . .	8
3.7. Conexionado de LEDs y armado de la matriz . . . . .	9
3.8. Prototipo final . . . . .	11
<b>4. Descripción funcional</b>	<b>13</b>
4.1. Descripción del sistema general . . . . .	13
4.2. Comunicación entre el cliente y el Wemos D1 . . . . .	14
4.3. Comunicación entre el administrador y el Wemos D1 . . . . .	16
4.4. Comunicación entre el Wemos D1 y el MAX7219 . . . . .	17
4.5. Protocolo de comunicación SPI entre los MAX7219 . . . . .	17
4.6. Sucesiones de eventos . . . . .	18
<b>5. Descripción software</b>	<b>19</b>
5.1. Archivo principal . . . . .	20
5.2. Clase WebServer . . . . .	21
5.2.1. init . . . . .	21
5.2.2. httpHandlers . . . . .	23
5.2.3. tick . . . . .	24
5.3. Clase Letter . . . . .	24
5.3.1. init . . . . .	25
5.3.2. Método setMessage . . . . .	26
5.3.3. setMap . . . . .	26
5.3.4. setPredefined . . . . .	27
5.3.5. setPartyOn . . . . .	27
5.3.6. setEnabled . . . . .	27
5.3.7. setIntensity . . . . .	28
5.3.8. clearScreen . . . . .	28
5.3.9. tick . . . . .	28
<b>6. Guías</b>	<b>29</b>
6.1. Ambiente de desarrollo e instalación . . . . .	29
6.1.1. Git . . . . .	29
6.1.2. Visual Studio Code . . . . .	29
6.1.3. Plugins necesarios para Visual Studio Code . . . . .	30
6.2. Instalación . . . . .	30
6.3. Compilación . . . . .	30

<b>Apéndice</b>	<b>33</b>
<b>A. Propuesta original</b>	<b>33</b>
A.1. Introducción . . . . .	33
A.2. Objetivo . . . . .	33
A.3. Dispositivos a utilizar . . . . .	33
A.4. Esquema gráfico . . . . .	34
A.5. Identificación de partes . . . . .	34
<b>B. Código fuente Servidor Web</b>	<b>37</b>
B.1. main.cpp . . . . .	37
B.2. WebServer.h . . . . .	38
B.3. WebServer.cpp . . . . .	40
<b>C. Código Fuente controlador matriz</b>	<b>44</b>
C.1. Letter.h . . . . .	44
C.2. Letter.cpp . . . . .	47

## **1. Propuesta original**

La propuesta original de este proyecto, se basa en el desarrollo de un programa para el microcontrolador Wemos D1 que le permita controlar una matriz de luces. La matriz está compuesta por módulos de 8x8 LEDs cada uno. Adicionalmente, el Wemos D1 debe ser capaz de hostear un servidor web, del cual extrae las peticiones HTTP que le realizan los clientes en relación al contenido que desean mostrar sobre la matriz.

Respecto de las peticiones, éstas se clasifican en tres categorías: frases, mapeos led a led (explicitando cuáles de ellos deben estar encendidos y cuáles no), y por último la representación de sprites animados almacenados dentro de la memoria del microcontrolador.

Como parte de la propuesta original es necesario, además, que el servidor previamente mencionado, corra sobre una red WiFi que el propio microcontrolador hostea y a la que se conectan los clientes. En otras palabras, el Wemos D1 debe actuar como access point. La tecnología implementada sobre el microcontrolador restringe una conexión máxima de cuatro clientes simultáneos.

Para este proyecto se propusieron dos objetivos secundarios. En primer lugar una funcionalidad para que el dueño del sistema filtre las publicaciones que no desea mostrar en la matriz. El otro objetivo consiste en que el programa debe ser capaz de poder deslizar el contenido publicado en la matriz, al estilo marquesina, en caso de que el usuario así lo desease, por ejemplo si el mensaje fuese más largo que la cantidad de módulos de LEDs disponibles. En el apéndice A se encuentra la propuesta aprobada por la cátedra.

## **2. Correcciones/cambios de la propuesta**

### **2.1. Indicadas por la Cátedra**

Las aclaraciones fueron, por un lado utilizar el Wemos D1 como Access Point y por otro, que la implementación del producto sobre la remera no era necesario; siendo prioritario el funcionamiento de la matriz de leds individualmente.

### **2.2. Definidas por el avance y disponibilidad**

Para armar la matriz de LEDs se optó por utilizar dispositivos integrados shifters MAX7219 en vez de los 74HC595 ya que facilitan la multiplexación. Se compraron tres en total, los cuales tienen un costo de \$45 pesos cada uno. Por otro lado se optó por la utilización de LEDs individuales en sustitución de la tira de LEDs debido a la sugerencia de la cátedra respecto de construir un cartel aislado en vez de hacerlo sobre una remera.

Cabe destacar que los objetivos principales descriptos en la sección anterior se desarrollaron de manera completa y satisfactoriamente, en tiempo y forma.

Respecto del objetivo secundario relacionado a la funcionalidad que permite administrar los posteos, se diseñó una interfaz para habilitar y deshabilitar las publicaciones que realicen los clientes. Adicionalmente, en la misma página, se agregaron interfaces gráficas para cambiar las intensidades de los LEDs y limpiar la pantalla, eliminando la última publicación. Estas herramientas se desarrollaron en tiempo y forma. El otro objetivo secundario que permite deslizar el contenido de la publicación a modo de marquesina se desarrolló completamente, en tiempo y forma. Adicionalmente se agregó un servidor DNS que corre sobre el microcontrolador a fin de mejorar la experiencia de usuario. Esta funcionalidad evita que el cliente deba recordar la dirección IP del servidor web hosteado por el Wemos D1.

Por otra parte, respecto de la construcción de la matriz; se desarrolló un prototipo de dos módulos de 8x8 LEDs cada uno. La idea de crear más de un módulo tiene como objetivo hacer un especial énfasis respecto de la escalabilidad del proyecto, a fin de mostrar la facilidad, a nivel de software y hardware, para agregar más módulos de LEDs. El prototipo se construyó a modo de shield para que se integre pin a pin en el Wemos D1.

### 3. Descripción y documentación general

#### 3.1. Wemos D1

El Wemos D1 está compuesto de un chip ESP8266 que posee conexión WiFi. El chip está montado sobre una placa Arduino compatible. En la figura 1 se puede observar una imagen del microcontrolador provisto por la cátedra.

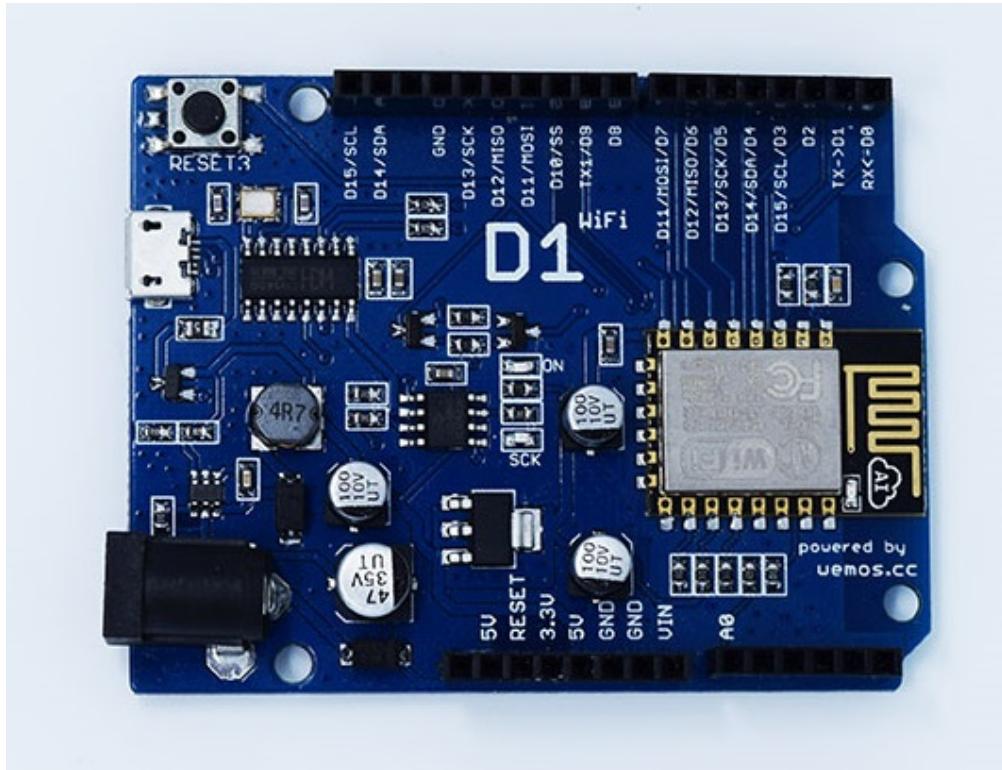


Figura 1: Microcontrolador Wemos D1.

Las características principales de la placa se listan a continuación.

- Once entradas/salidas digitales.
- Una entrada analógica.
- Conector micro USB
- Distribución de pines tipo Arduino UNO.
- PWM en todos sus pines digitales.
- Voltaje de operación de 3.3 V
- Once pines digitales de entrada y salida.
- Frecuencia de clock de 80Mhz.
- Flash de 4MB.
- Dentro de la flash, 1MB reservado para el almacenamiento de archivos permanentes (SPIFFS).

En la tabla 1 se muestra la distribución de pines de la placa. Como se puede observar, soporta los protocolos de comunicación serie I2C y SPI, siendo este último, el utilizado en este proyecto para establecer la transmisión de mensajes entre el microcontrolador y los chips MAX7219 que manejan la matriz de LEDs.

Tabla 1: Tabla pines Wemos D1.

Pin	Función	ESP8266 pin
D0	RX	GPIO3
D1	TX	GPIO1
D2	IO	GPIO16
D3(D15)	IO, SCL	GPIO5
D4(D14)	IO, SDA	GPIO4
D5(D13)	IO, SCK	GPIO14
D6(D12)	IO, MISO	GPIO12
D7(D11)	IO, MOSI	GPIO13
D8	IO, Pull up	GPIO0
D9	IO, Pull up, BUILDTIN LED	GPIO12
D10	IO, Pull down, SS	GPIO15
A0	Entrada analógica	A0

Tanto el convertidor analógico digital que posee la placa, como los pines de salida, tienen un valor lógico alto de 3.3 V.

Por otra parte, el microcontrolador es capaz de trabajar en tres modos de operación. Puede actuar como cliente WiFi, como access point o como ambas. Adicionalmente esta placa soporta la programación OTA (Over The Air).

### 3.2. MAX7219

#### 3.2.1. Descripción general

El MAX7219 es un controlador compacto, de entrada y salida en serie de cátodo común que conectan microprocesadores a LEDs numéricos de siete segmentos de hasta ocho dígitos, pantallas de gráfico de barras o 64 LED individuales. Se incluyen en el chip, un decodificador BCD, circuitos de multiplexación, controladores de segmentos y dígitos, y una RAM estática de 8x8 que almacena cada número. Solo se necesita una resistencia externa para configurar la corriente de segmento para todos los LEDs.

El MAX7219 es compatible con los protocolos SPI, QSPI y Microwire, y tiene controladores de segmento de velocidad limitada para reducir el EMI.

Los dígitos individuales se pueden actualizar sin reescribir toda la pantalla. El MAX7219 también permite al usuario seleccionar el código de decodificación o no decodificación para cada dígito.

Su alimentación V+ debe estar entre 4 y 5.5 Volts para su correcto funcionamiento. En cambio los voltaje de las entradas lógicas tienen una restricción de que un valor alto como mínimo debe ser 3.5V y un valor bajo como máximo 0.8V. Como la placa usada tiene valores digitales de 3.3V, se realizaron previamente las pruebas necesarias para llegar a la conclusión de que, mas allá de la restriccion del datasheet, la placa utilizada logra comunicarse con el MAX7219 de forma correcta en todos los casos.

Las características principales que posee el chip integrado se enumeran a continuación:

- Interfaz serie de 10 MHz.
- Control de segmento LED individual.
- Selección de dígitos Decode / No-Decode.
- Apagado de baja potencia de 150 microA (datos retenidos).
- Control de brillo digital y analógico.
- Pantalla borrada al encenderse.
- Unidad de visualización LED de cátodo común.
- SPI, QSPI, interfaz serie Microwire paquetes DIP y SO de 24 pines.

### 3.2.2. Configuración de pines

La disposición de los pines del MAX7219 se puede observar en la figura 2 y la descripción de cada uno en la tabla 2.

Es necesario prestar atención en el conexionado con respecto a los pines de tierra (GND) ya que ambos deben estar conectados para el driver pueda funcionar correctamente, ambas estan al lado izquierdo de la figura 2 (Pin 4 y 9).

Por otro lado, el MAX7219 tiene un pin denominado DOUT (pin 24) se utiliza para encadenar varios MAX7219 y de esta forma pasar la información al que esta directamente conectado, éste pin nunca tiene alta impedancia.

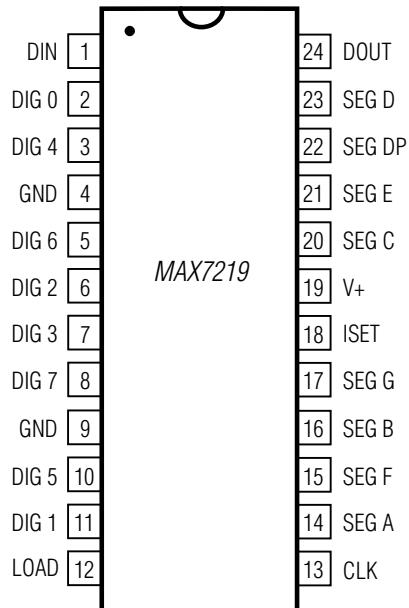


Figura 2: Configuración de pines del chip MAX7219.

Tabla 2: Descripción de los pines del MAX7219

Pin	Nombre	Función
1	DIN	Pin de datos seriales. Los datos son cargados en el registro de 16 bits en cada flanco ascendente del clock.
2, 3, 5-8, 10, 11	DIG 0 - DIG 7	Líneas de transmisión de ocho dígitos que absorben corriente del cátodo común de la pantalla. El MAX7219 deja en V+ cuando está apagado. Los dígitos están en alta impedancia cuando se apaga.
4, 9	GND	Tierra.
12	LOAD	Pin de control. Los últimos 16 bits del Serial Data son cargados en el flanco ascendente.
13	CLK	Pin de clock serial. En cada flanco ascendente, los datos sin shifteados dentro de un registro interno. En cada flanco descendente los datos salen de DOUT. En el MAX7221, la entrada CLK está activa solo mientras LOAD está baja.
14-17, 20-23	SEG A-SEG G, DP	Las unidades de siete segmentos y el punto decimal impulsan la fuente de corriente a la pantalla. Cuando un controlador de segmento está apagado, se conecta a GND.
18	ISET	Conectar a $V_{DD}$ a través de una resistencia ( $R_{SET}$ ) para configurar la corriente que pueda entregar a los dígitos y segmentos.
19	V+	Fuente positiva de corriente, conectar a 5 V.
24	DOUT	Salida de datos en serie. Los datos en DIN son válidos en DOUT 16.5 ciclos de reloj más tarde.

### 3.3. Alimentación

Para la alimentación del microcontrolador se utilizará un PowerBank conectado vía USB (5V) a la placa del Wemos D1. La batería mencionada será la única fuente de alimentación del sistema.

La alimentación de 5V para los integrados MAX7219 se extraerá del Wemos D1, ya que posee pines para dicho propósito. Este integrado se conecta a 5V por el pin VCC (ver figura 3), y es recomendado incluir capacitores electrolíticos para ayudar a desacoplar la línea de alimentación y absorber los picos transitivos de energía necesarios para encender y apagar los LEDs.

### 3.4. Conexión del Wemos D1 con el MAX7219

El microcontrolador se conecta con un chip shifter MAX7219 de la forma en que se muestra en la figura 3.

El microcontrolador Wemos D1 soporta el protocolo de comunicación serie SPI, como ya se describió en la sección 3.1. Para ello se utiliza los pines D13 (SCK), D11 (MOSI) y D10 (SS) para el clock, datos y load respectivamente que se conectan directamente a las entradas del shifter MAX7219 tal como se puede observar en la figura 3.

Adicionalmente, en la figura 4 se observa una foto del Wemos D1 y sus conexiones con el MAX7219. A fin de facilitar la comprensión del circuito, se utilizan cables de colores donde cada uno de ellos representa una funcionalidad. Se utiliza el verde para la conexión con Vcc y el rojo para ground. Mientras que para el clock, datos y load se utilizan los cables amarillo, gris y azul respectivamente.

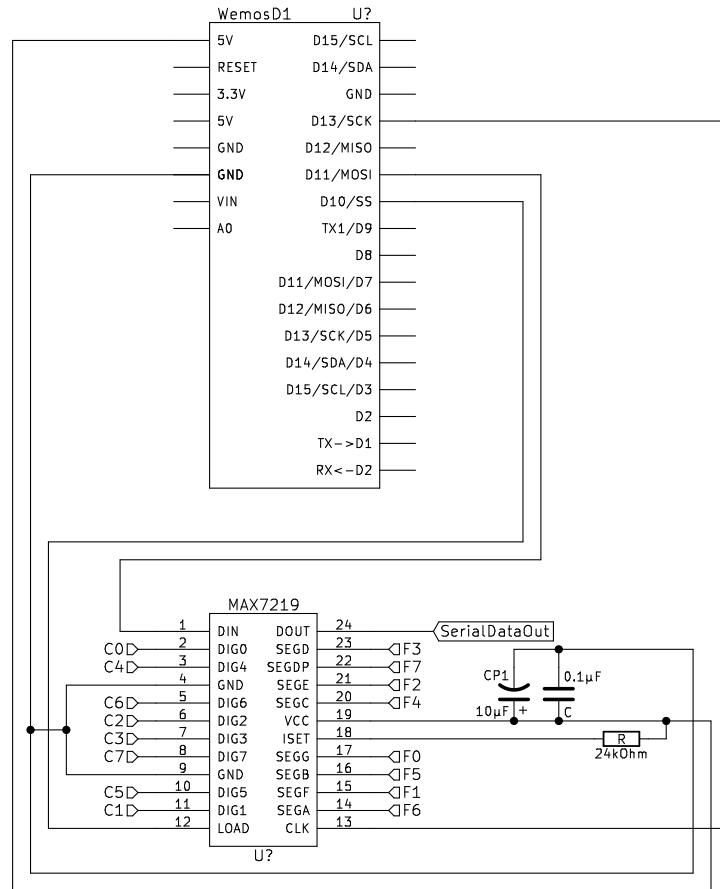


Figura 3: Conexión del microcontrolador Wemos D1 con MAX7219.

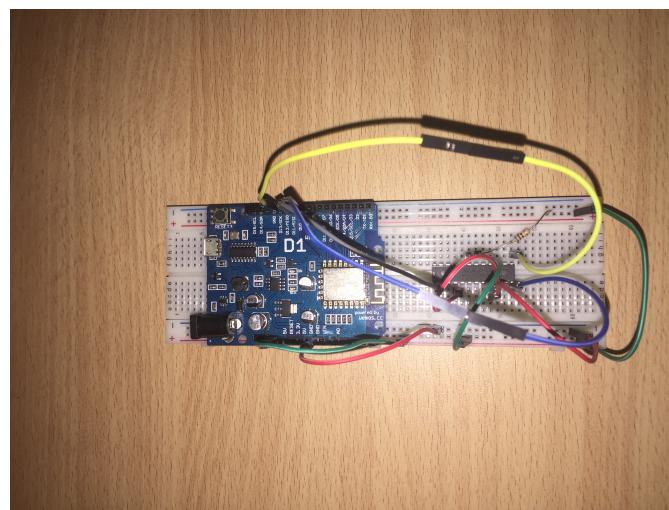


Figura 4: Fotografía de interconexión del microcontrolador Wemos D1 con MAX7219.

### 3.5. Conexión del MAX7219 con su módulo de 8x8 LEDs

Cada uno de los MAX7219 está conectado a un módulo de 8x8 LEDs, sus conexiones se pueden observar en el diagrama 5. Adicionalmente, en la figura 6 se observa el prototipo que complementa el esquema de conexión de la figura previamente mencionada.

A la hora de efectuar las conexiones, se debe prestar principal atención a la orientación de la matriz. En la figura 6 se indica claramente cuáles son las filas (y su orden) y cuáles son las columnas. Con dicha información el proceso de conexionado se simplifica.

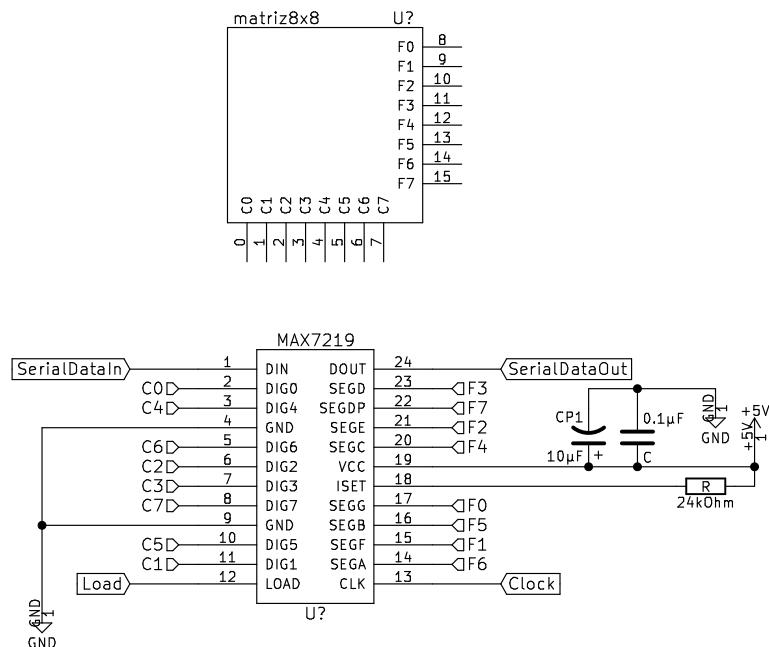


Figura 5: Conexión entre MAX7219 y su módulo de LEDs.

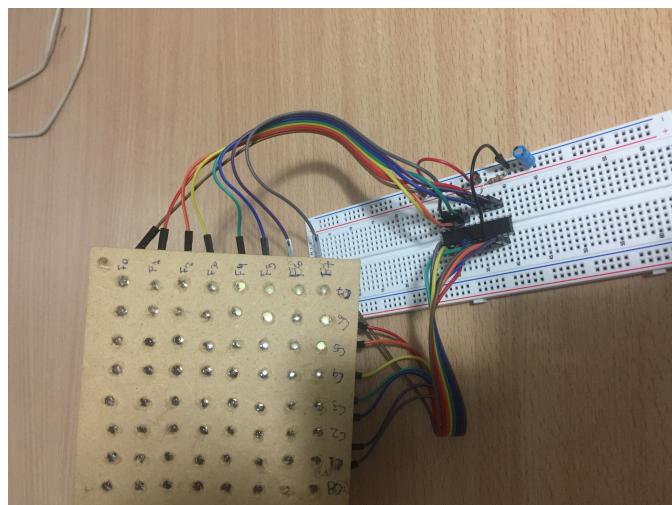


Figura 6: Fotografía de conexión entre MAX7219 y su módulo de LEDs.

### 3.6. Concatenación de varios módulos de LEDs

Anteriormente se hizo referencia a la forma en que se conectan el Wemos D1 con un MAX7219 y a la manera en que éste lo hace con la matriz de LEDs. Sin embargo al inicio del informe se mencionó la posibilidad de interconectar más de un módulo de 8x8 LEDs. En particular, para este proyecto se conectan dos módulos, generando una matriz de 8 filas por 16 columnas, dando un total de 128 luces.

Para realizar esta operación, es necesario que los MAX7219 se interconecten, uniendo el pin Data-Out del primero, hacia el pin Data-In del siguiente. Ésto permite que la información circule entre un shifter y el otro, de manera serial. Cabe mencionar que cada módulo funcional, está conectado al pin de clock y load del Wemos D1 para la transferencia de datos de forma serial.

Es importante aclarar que la forma en la que se comunica un MAX7219 con el siguiente se detalla en la sección 4.5. En esta parte solo se especifica el conexionado para que pueda establecerse el pasaje de los datos entre un chip y el siguiente.

Por otra parte, a fin de conocer el consumo máximo de un chip shifter MAX7219 conectado con su respectivo módulo de 8x8 LEDs, se utilizaron las ecuaciones proporcionadas en el [link](#) del chip. Éstas fórmulas contemplan las variables donde puede llegar a producirse mayor incertidumbre. Se obtuvo un consumo de 8.5 mA por módulo.

En la figura 7 se observa la forma en que se debe realizar la interconexión de los dispositivos integrados previamente mencionados mientras que la figura 8 complementa el esquemático mostrando una fotografía real del prototipo funcional.

Para la fotografía, se quitaron determinados cables a fin de simplificar el esquema. Sólo se dejaron los necesarios para mostrar como se conectan dos MAX7219. Los colores se mantienen iguales respecto de la figura 4. Se puede observar que el cable blanco se conecta desde el pin DataOut del primer MAX7219 hacia el pin DataIn del segundo. Por otra parte el clock y load se comparten con el primer shifter cuyos colores son amarillo y azul respectivamente.

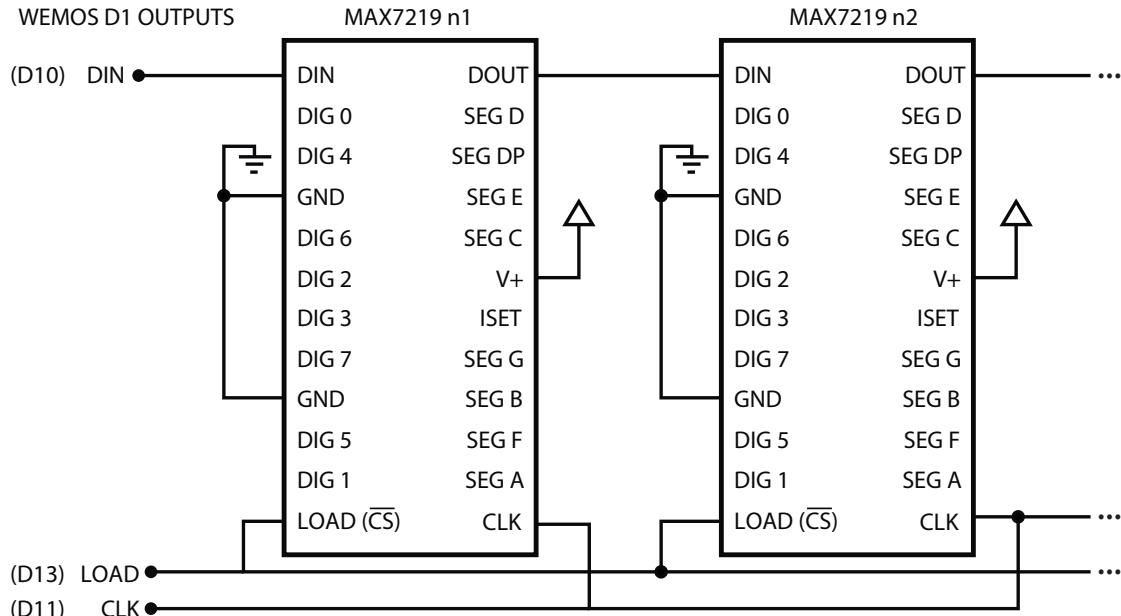


Figura 7: Conexión entre dos MAX7219.

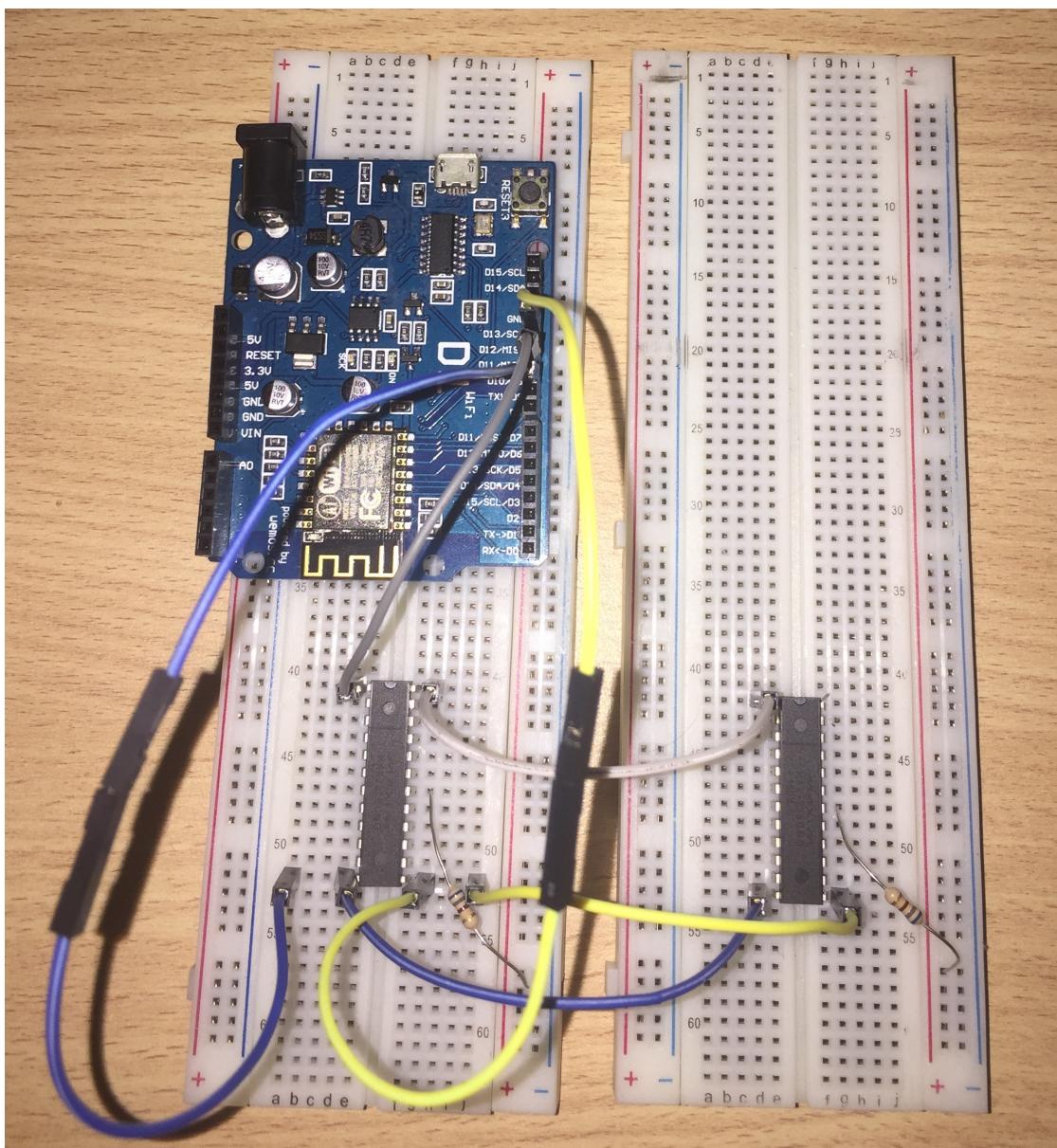


Figura 8: Fotografía de conexión entre dos MAX7219.

### 3.7. Conexionado de LEDs y armado de la matriz

Los LEDs color blanco soportan entre 2.8V a 4.2V con una corriente de 20mA, por lo que se eligió un tensión promedio de 3V para alimentarlos. La hoja de datos del dispositivo integrado MAX7219 indica que es necesario colocar una resistencia  $R_{SET}$  entre el pin  $I_{SET}$  y la alimentación para controlar la tensión y la corriente entregada en sus pines de salida. A fin de obtener más claridad, respecto del nombre de los pines, se puede observar la figura 2 la configuración de los pines del shifter. La tabla 3 permite elegir  $R_{SET}$  según las necesidades de cada aplicación o sistema; para este proyecto se eligió  $R_{SET}$  con valor de 24.5KOhm aproximadamente debido a las características previamente enunciadas de los LEDs utilizados para cada módulo.

Tabla 3: Tabla extraída del [datasheet](#) del MAX7219, que relaciona la resistencia  $R_{SET}$  con la salida de sus pines.

$I_{SET}$ (mA)	$V_{LED}$ (V)				
	1.5	2.0	2.5	3.0	3.5
40	12.2	11.8	11.0	10.6	9.69
30	17.8	17.1	15.8	15.0	14.0
20	29.8	28.0	25.9	24.5	22.6
10	66.7	63.7	59.3	55.4	51.2

Resumiendo, la matriz se compone de dos módulos. Cada uno contiene 64 LEDs dispuestos en forma de ocho filas por ocho columnas. Además posee un chip controlador MAX7219 para controlar las luces y componentes pasivos varios (ver figuras 5 y 9).

La matriz de 128 LEDs se interconecta con el microcontrolador Wemos D1. Éste es el encargado de recibir los pedidos del usuario, vía HTTP, procesar la información y transmitir los resultados a los MAX7219 para que representen la información solicitada en la matriz. A modo complementario se muestra una fotografía de un prototipo físico terminado, donde se observan las interconexiones de los LEDs. La misma se puede observar en la figura 10.

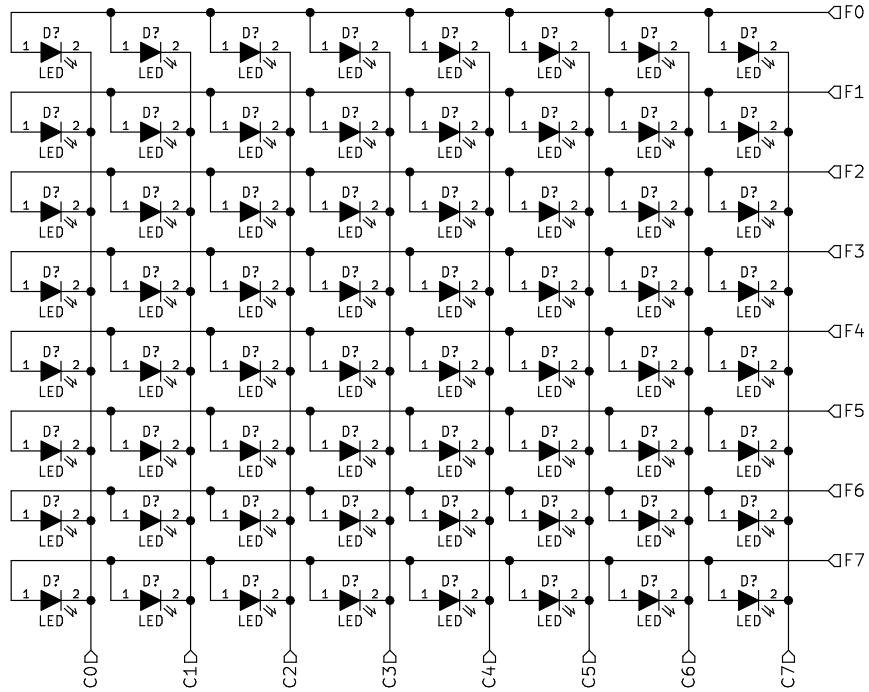


Figura 9: Esquema de conexiones de la matriz de LEDs.

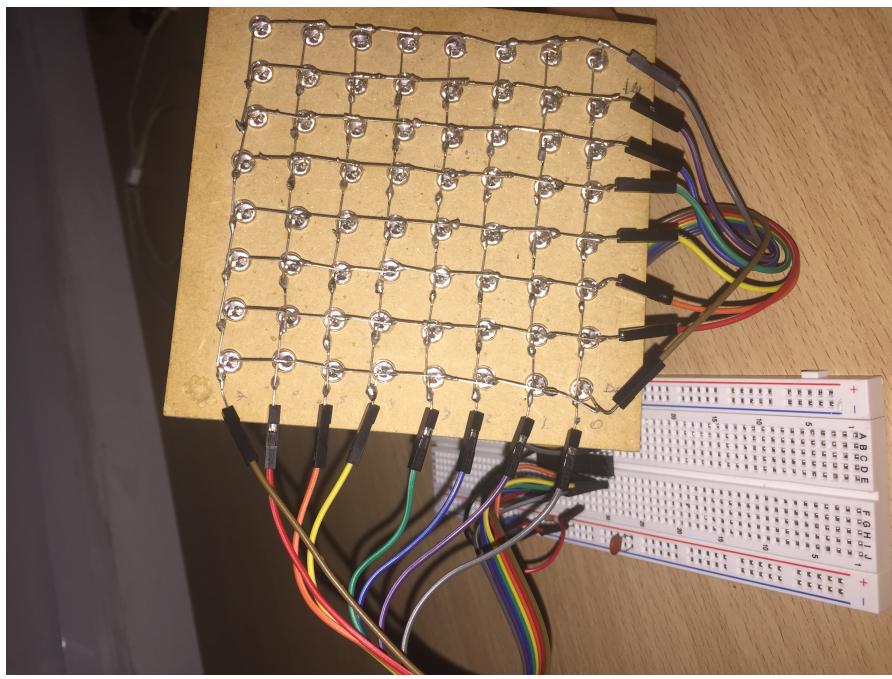


Figura 10: Fotografía de esquema de conexiones de la matriz de LEDs.

### 3.8. Prototipo final

A modo de conclusión de esta sección, en la figura 11 se presenta el diagrama de conexiones final, donde se incluye el Wemos D1 y dos chips MAX7219 conectados en cascada con el microcontrolador. A su vez, cada shifter está conectado con su respectivo módulo de 8x8 LEDs tal cual se describió a lo largo de esta sección.

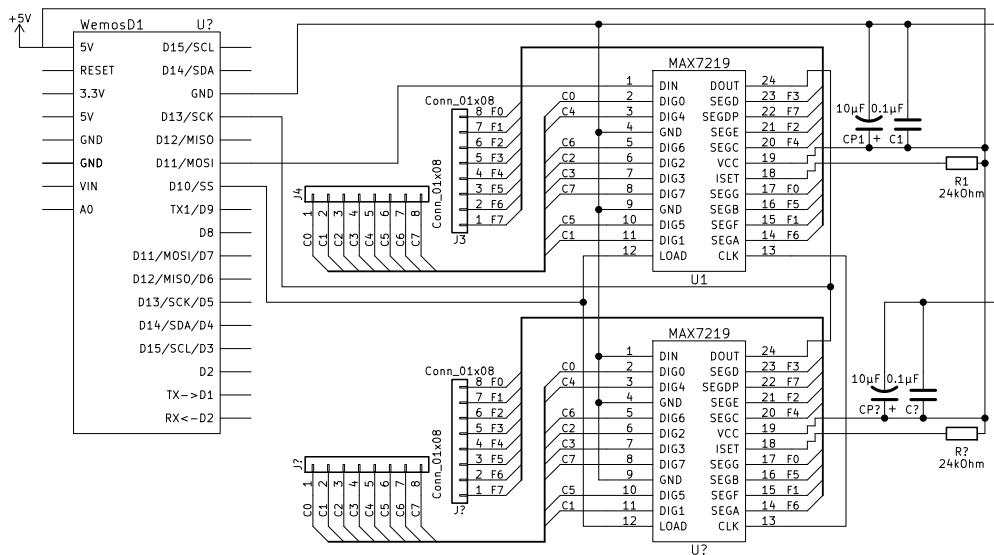


Figura 11: Conexión del sistema completo.

Adicionalmente se presenta de la figura 12 a la 14 fotografías del prototipo final terminado, implementado como modulo acoplable al Wemos D1.

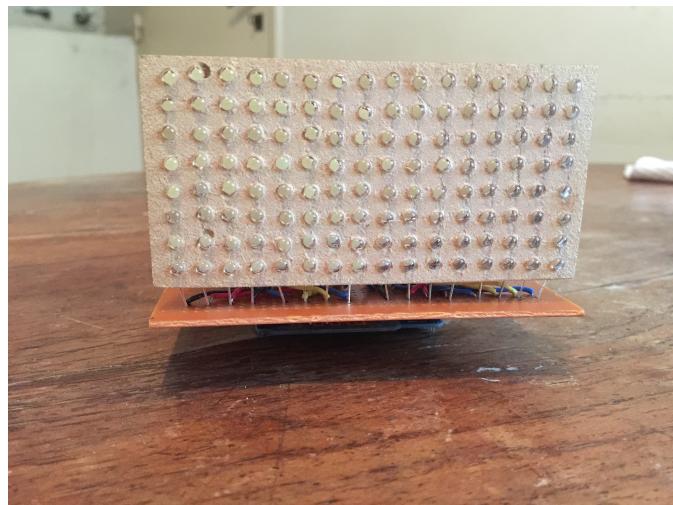


Figura 12: Prototipo final: vista frontal.



Figura 13: Prototipo final: vista superior.

En el siguiente [link](#), se adjunta un video en el que se muestra y se explica brevemente las conexiones sobre el prototipo final. Este archivo multimedia complementa los esquemáticos previamente mostrados en el informe. Cabe destacar que el prototipo final que se observa en el video no esta construido sobre protoboard, si no, a modo de shield para poder integrarse sin problemas con el Wemos D1.

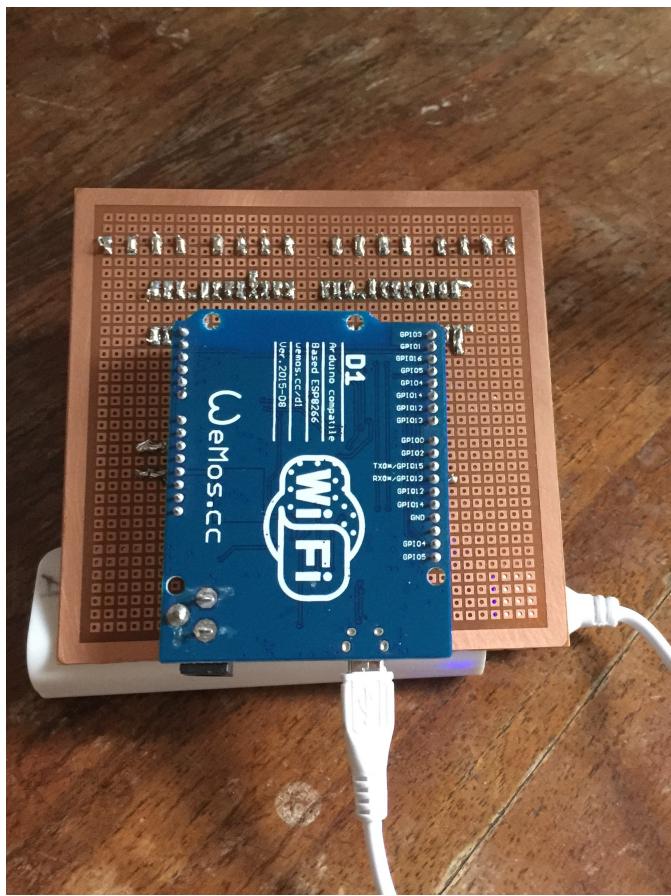


Figura 14: Prototipo final: vista por debajo.

## 4. Descripción funcional

### 4.1. Descripción del sistema general

La comunicación de la placa Wemos D1 con los clientes se realiza vía WiFi, accediendo a un servidor hosteado por el microcontrolador, a partir de una red creada también por el mismo. A fin de facilitar el acceso al sistema, se implementa un servicio de DNS que permite enlazar un nombre de dominio a la dirección IP del servidor web.

Por su parte, una vez conectados a la red, los clientes acceden al servidor web y envían sus requerimientos vía HTTP. Entre las posibles peticiones se encuentra: representar una frase, establecer LED a LED cuál debe prenderse y cuál no, y por último, mostrar sprites animados previamente almacenados en la memoria del programa. Adicionalmente se provee mecanismos para limpiar la pantalla, cambiar la intensidad de los LEDs y deshabilitar las posteriores publicaciones que se realicen. Estas últimas funcionalidades son realizadas por usuarios que ingresan al sistema en modo administrador.

La información es capturada por el sistema y procesada. El Wemos D1 es el encargado de llevar a cabo dicha tarea, transformando el pedido en una secuencia de bytes que posteriormente son transmitidos vía SPI hacia el primer MAX7219 que está conectado directamente al microcontrolador. En la figura 15 se puede observar el esquema general, en forma de diagrama en bloques, de los diferentes subsistemas que se comunican entre sí.

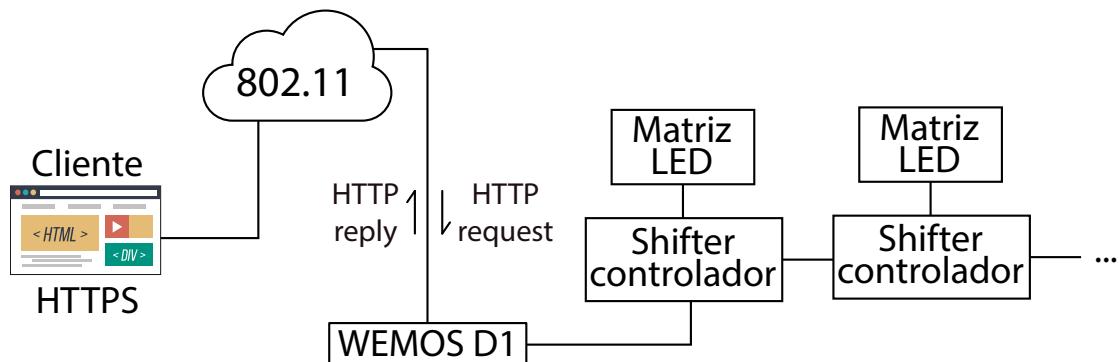


Figura 15: Diagrama en bloque del sistema.

#### 4.2. Comunicación entre el cliente y el Wemos D1

Como ya se dijo anteriormente el microcontrolador hostea una página web de manera tal que permita la comunicación entre los clientes y el Wemos D1. En la figura 16 se observa un campo de texto para que el usuario pueda ingresar una frase, es decir caracteres alfanumericos ASCII. Cabe aclarar que las frases pueden ser elegidas como estáticas o que se desplieguen a modo de marquesina para el caso de que el mensaje supere la cantidad de módulos que se tienen. Ese campo no tiene un límite definido para la cantidad de caracteres.

También, es posible especificar la velocidad en que los caracteres serán mostrados en el input inferior al de texto. Este campo esta en unidades de milisegundos y representa la velocidad que los datos de una columna de LED pasan a la siguiente columna. Si no se especifica ningún valor, los datos no se mueven.

Figura 16: Interfaz para ingresar un texto.

Por otra parte, si el cliente lo desea, puede manipular los leds individualmente a través de una matriz de puntos que se encuentra en la página web. De esta forma, puede encender y apagar las luces que deseé. En la figura 17 se muestra la interfaz gráfica a través de la cual se implementa dicha funcionalidad. De la misma manera en que se puede controlar la velocidad de movimiento en la frase, es posible en este caso, generar un corrimiento de los estados de los LEDs.

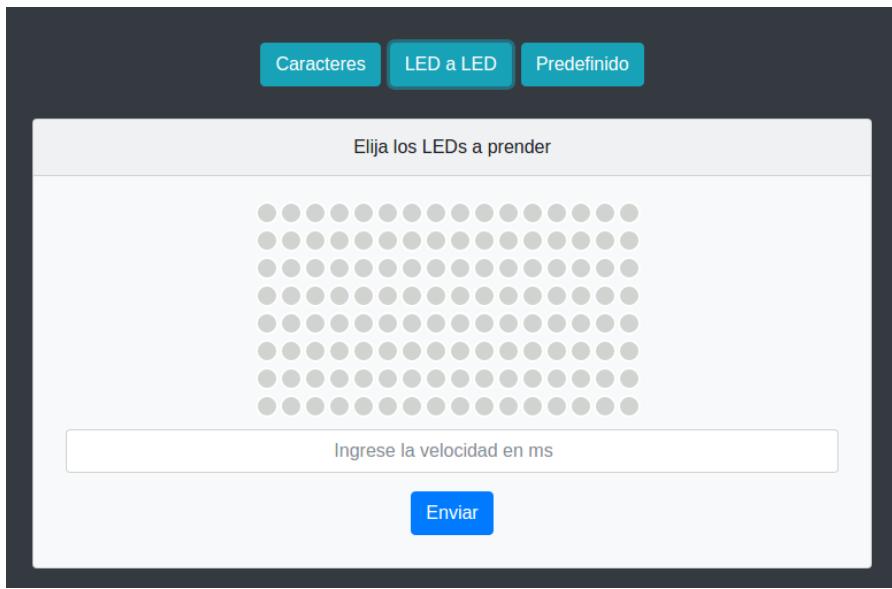


Figura 17: Interfaz para manipular LEDs individualmente.

Y por último, los usuarios pueden elegir diferentes secuencias animadas que se encuentran precargados en la memoria del microcontrolador; dichas animaciones se pueden observar en la figura 18.



Figura 18: Interfaz para elegir imágenes precargadas en memoria.

La página web esta pensado para mostrarse en tanto en ordenadores de escritorio, como notebooks, tablets y dispositivos móviles por lo que se desarrolló sobre un framework *responsive* denominado Bootstrap 4 Beta. En la figura 19 se puede apreciar como se integra a las pantallas reducidas.

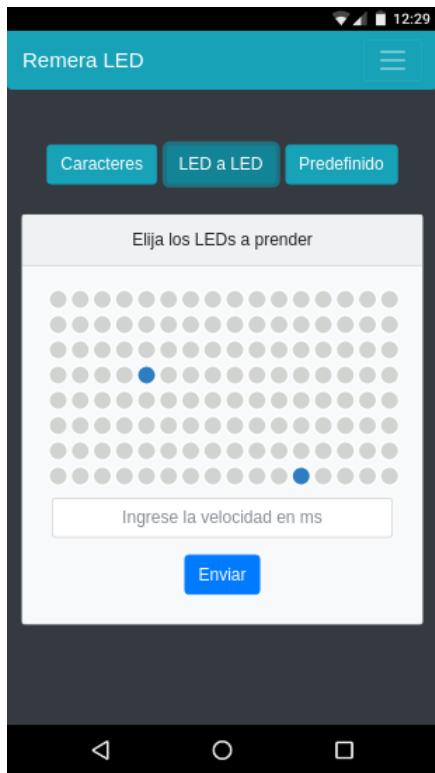


Figura 19: Interfaz para manipular LEDs en un móvil.

#### 4.3. Comunicación entre el administrador y el Wemos D1

El sistema provee una funcionalidad para ingresar como administrador del sitio. Esto permite realizar operaciones privilegiadas que un usuario normal no podría ejecutar. Entre ellas se destacan operaciones para eliminar la publicación actual, cambiar la intensidad de los LEDs y habilitar/deshabilitar los futuros posteos.

Para acceder como administrador se debe ingresar un token de seguridad tal cual se observa en la figura 20. Cabe aclarar que dicho token se corresponde con la frase “tres tristes tigres”.

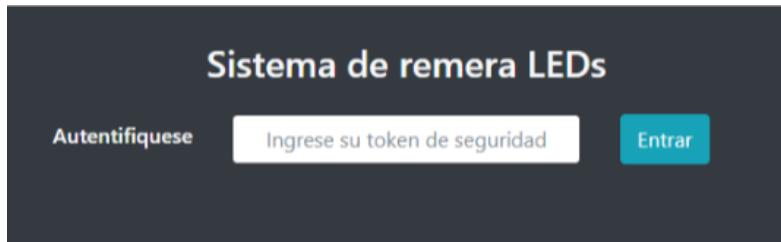


Figura 20: Interfaz para la autenticación de usuarios admin.

Una vez dentro, se observa una interfaz como se muestra en la figura 21. En ella se pueden observar las funcionalidades que sólo un administrador puede realizar. Las mismas se corresponden con las listadas previamente en esta sección.

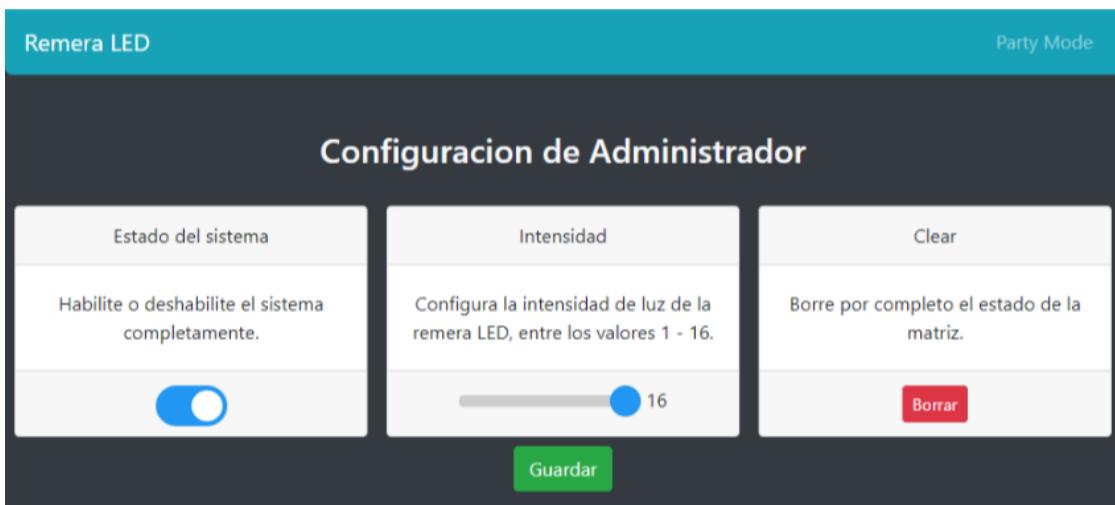


Figura 21: Interfaz de la configuración del lado administrador.

#### 4.4. Comunicación entre el Wemos D1 y el MAX7219

Como se detalló anteriormente, el chip shifter MAX7219 recibe una cadena de bytes de tamaño fijo vía SPI. Para ello el Wemos D1 actúa como maestro del canal e impone un clock, una señal de load y los datos que quiere transmitir (ver figura 3). El MAX7219 posee ocho dígitos en los cuales almacena un byte de datos en cada uno. Para este proyecto, cada dígito representa una columna, y los datos que cada uno almacena se utilizan para representar qué filas de la columna debe encenderse. La trama del paquete se observa en la tabla 4.

Tabla 4: Trama SPI y su significado.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	ADDRESS				MSB	DATA				LSB		

En la tabla 4 se observan los diferentes campos de interés. El primero es ADDRESS que representa la columna hacia la cuál está dirigido el mensaje. El valor va desde uno hasta ocho inclusive y corresponde a las ocho columnas que tiene cada módulo. Por otra parte, DATA, representa ocho bits de información dirigida a dicha columna. Siendo D0 la fila 0 en la columna indicada por address.

#### 4.5. Protocolo de comunicación SPI entre los MAX7219

Es importante destacar la forma en la que se comunican el microcontrolador con el primer chip shifter, y a su vez, el protocolo que utilizan para comunicarse entre los distintos MAX7219.

La idea general consiste en indicarle a cada chip shifter los leds que debe encender y apagar en cada columna. Para ello debe enviar palabras de dos bytes de manera serial. Es decir, primero envía la información de la columna 1, después la 2, y así siguiendo hasta la 8. El protocolo utilizado es SPI y corresponde a la trama previamente detallada (ver figura 4).

Para realizar este proceso, la figura 22 muestra un diagrama a lo largo del tiempo de la forma de enviar cada bit. En ella se puede observar que el primer paso consiste en bajar la señal de LOAD y esperar un instante de tiempo (aproximadamente un microsegundo). Luego se debe generar una señal de CLCK de onda cuadrada y de frecuencia de 1Mhz con la mitad de ciclo de trabajo.

El pin conectado al DATAIN del micro, se utiliza para enviar los datos, empezando por el bit más significativo primero. Los datos van a ser leídos por los MAX7219 en el flanco ascendente del CLOCK. Cuando finaliza el envío de los 16 bits, se debe subir la señal de LOAD. En ese momento, el MAX2719 almacena, en sus registros internos, el comando recibido. Todos los comandos son de dos bytes, sin embargo, se puede enviar más de esa cantidad. Esta funcionalidad se utiliza para enviar instrucciones a los demás MAX7219 que están conectados en serie. Lo que ocurre es que la información se recibe en el flanco ascendente de CLK y se envía, hacia el siguiente chip, por el pin DATAOUT en el flanco descendente. De esta forma, en una iteración se pueden configurar una columna de cada módulo de 8x8 LEDs.

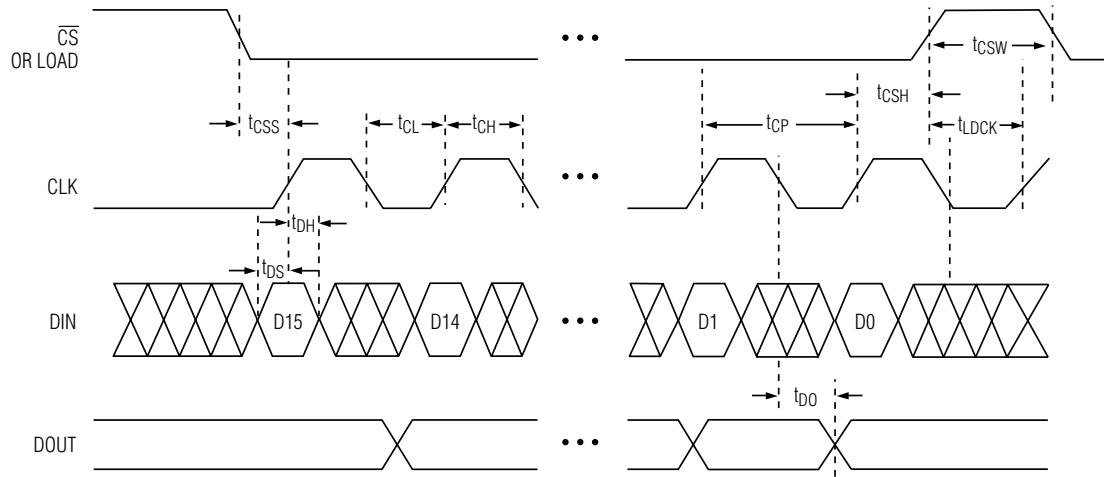


Figura 22: Diagrama de tiempo de las señales del MAX7219.

## 4.6. Sucesiones de eventos

El sistema al iniciar, crea una red WiFi cuyo SSID corresponde a “Remera LED” y su password “12345678” (excluir las comillas). Posteriormente, monta un servidor web utilizando la dirección IP que le fue asignada, y el puerto 80.

Como la dirección a nivel de red podría cambiar a lo largo del tiempo, el sistema monta inmediatamente a continuación un servidor DNS que enlaza la dirección IP del servidor web con un nombre. En este caso es «www.ledcartel.com». Este nombre es el que deben colocar los clientes en el navegador para ingresar al sistema. El programa, tras haber iniciado el servidor DNS, se queda a la espera de las peticiones que los clientes realicen.

El siguiente paso lo realiza el cliente. En algún momento determinado, el mismo accede a la red y coloca la contraseña. Luego abre su navegador de preferencia e ingresa al sistema escribiendo el nombre «www.ledcartel.com». El servidor DNS le proporciona la IP correspondiente del servicio web que hostea el microcontrolador Wemos D1.

A partir de ese momento el usuario puede realizar alguna de las siguientes tres acciones: escribir una frase, indicar uno a uno, el estado de cada led o, incluso, elegir alguno de los sprites animados que el sistema tiene precargado en su memoria. Las figuras 16, 17 y 19 muestran, respectivamente, las páginas para realizar las acciones previamente mencionadas. Cualquiera sea la petición, el usuario puede optar por deslizar el contenido, al estilo marquesina, ingresando en el campo de texto de abajo, la velocidad por pixel que desea. El rango de valores es desde -10.000 a 10.000 milisegundos. Un valor negativo, indica que se desliza hacia la izquierda, y uno positivo hacia la derecha. Mientras que 0 indica que el mensaje no realiza desplazamiento alguno.

El cliente a su vez, conociendo el token de seguridad puede acceder al sistema en modo administrador o privilegiado, tal como se muestra en la figura 20. Una vez dentro, se pueden realizar cualquiera de las 4 opciones que se presentan a continuación.

En la figura 20 se observa una captura de pantalla de la pestaña administrador. El usuario en modo privilegiado puede eliminar la última publicación realizada presionando el botón “clear”. Adicionalmente puede cambiar la intensidad de las luces o habilitar/deshabilitar la matriz para futuras publicaciones. Cualquiera de éstas dos acciones se efectúan apretando el botón de “Guardar”. Por último, el administrador puede activar un modo fiesta haciendo click en el botón “Party mode” que se encuentra en la esquina superior derecha de la pestaña.

Tras haber realizado una petición, el programa, por medio del protocolo HTTP recibe los datos, los procesa y almacena los resultados en un buffer de salida. El buffer posee la información de todas las columnas a setear en la matriz. Finalmente recorre el buffer y envía serialmente cada byte por medio del protocolo SPI comenzando primero, por las columnas del último MAX7219 conectado (ver Sección 4.5).

La figura 23 muestra un diagrama de casos de uso que representa las interacciones del cliente y de la matriz de LEDs con el sistema que corre sobre el Wemos D1, tal como se detalló previamente.

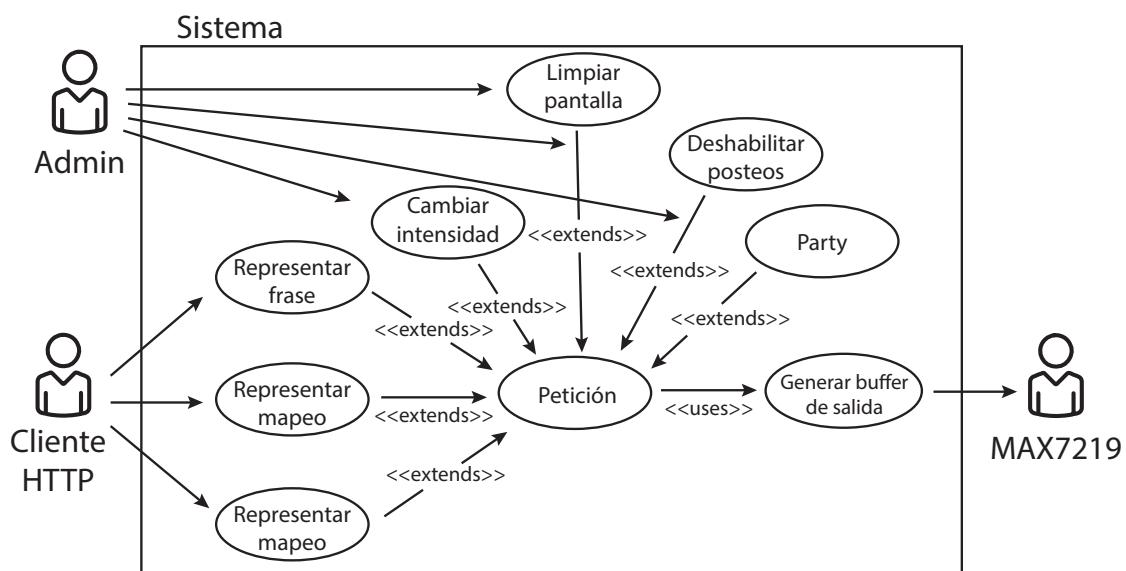


Figura 23: Diagrama de uso de interacción entre el cliente y el driver MAX7219 con el sistema.

Adicionalmente en el siguiente [link](#) se encuentra un vídeo que muestra la secuencia completa, previamente detallada a lo largo de esta sección.

## 5. Descripción software

La arquitectura de software se compone de dos importantes módulos. El primero es WebServer, esta clase modela todas las funcionalidades relacionadas al manejo del WiFi, del servidor DNS y del servidor web. Adicionalmente se encarga de interpretar las peticiones HTTP que recibe de los clientes y enviarlas casi sin procesamiento alguno, hacia la clase Letter.

La clase Letter, corresponde al segundo módulo que compone el sistema. La misma tiene como objetivo recibir los parámetros de las peticiones y en función de los valores obtenidos, arma un buffer de salida, con la información necesaria para encender los leds correspondientes en la matriz.

En la figura 24 se observa un diagrama en bloque de los componentes de software previamente detallados, mostrando además su interacción con el exterior, es decir, con clientes y con el chip MAX7219.

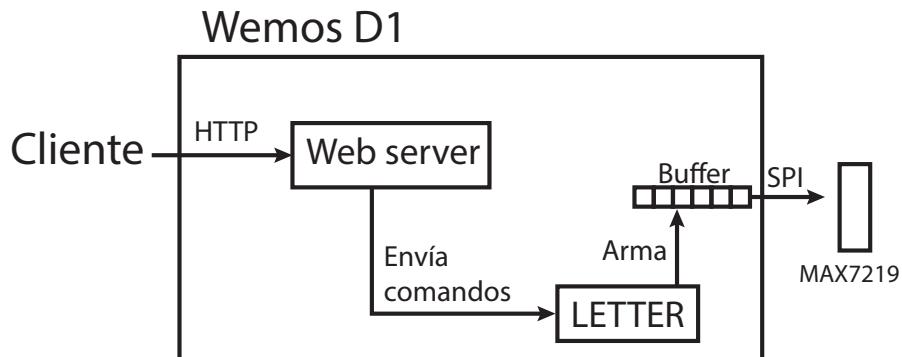


Figura 24: Flujo de información a través de las clases WebServer y Letter.

## 5.1. Archivo principal

```

0 #include <Arduino.h>
1 #include <WebServer.h>
2 #include <Letter.h>
3
4 void setup()
5 {
6     Letter::init();
7     WebServer::init();
8 }
9
10 void loop()
11 {
12     WebServer::tick();
13     Letter::tick();
14 }
  
```

Fragmento 1: Archivo `main.cpp`.

El archivo principal o punto de entrada en este programa es el main. El trabajo que realiza es muy simple. En principio incluye las clases WebServer y Letter previamente enunciadas.

Además tiene un método que se ejecuta una sola vez denominado `setup()` en donde inicializa los componentes de software previamente listados: WebServer y Letter.

A continuación un método `loop()` que se ejecuta indefinidamente. Este último permite realizar un tick a cada módulo del sistema. A lo largo de esta sección se detallan las características de cada componente mencionado. Lo importante a destacar es que la función `tick()` de cada elemento no debe ser bloqueante ni quedarse a la espera de algún evento externo que demore. Tanto el `tick()` de WebServer como el de Letter se han implementado siguiendo esa condición.

## 5.2. Clase WebServer

La clase WebServer es la encargada de establecer una red WiFi, hostear un servidor DNS y uno WEB para atender las peticiones que los clientes realicen. Por otra parte, posee métodos privados para manejar cada request que se envian por medio del protocolo HTTP.

En primer lugar se presenta el diagrama UML de la clase WebServer. La misma se observa en la figura 25 y contiene tanto los métodos públicos como privados que la clase posee.

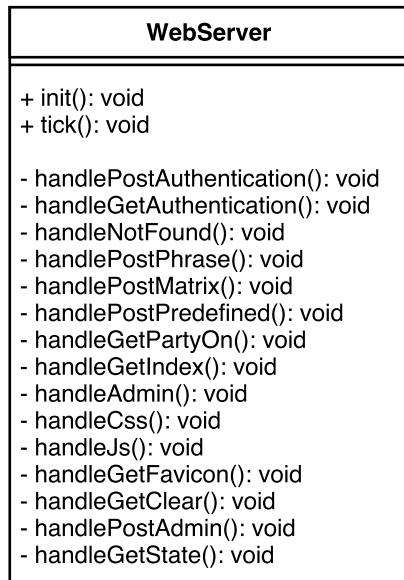


Figura 25: UML de la clase WebServer.

### 5.2.1. init

El primer método de interés, que ya se declaró en la subsección 5.1, es el init( ). A continuación se presenta un pseudocódigo para comprender su funcionamiento.

```
#include <FS.h>

void WebServer::init()
{
    SPIFFS.begin();
    initWifi(ssid, password);
    initDnsServer();
    initWebServer();
}
```

Fragmento 2: Pseudocódigo `WebServer.cpp`.

En primer lugar se importa la librería de Arduino “FS.h”, la misma provee funcionalidades para manejar el sistema de archivos que ofrece el módulo ESP8266 dentro del Wemos D1. Este

file System se denomina SPIFFS y se encuentra almacenado en una zona reservada de la flash. La cantidad de memoria depende del modelo del microcontrolador. En el caso de este proyecto es de 1MB. En el siguiente [link](#) se puede acceder al código fuente de la misma.

Entre las funciones que ofrece esta librería se encuentran las de leer y escribir sobre memoria no volátil. En este proyecto se utiliza para leer los archivos html almacenados en la flash a fin de msotrarlos en el navegador del cliente. Para poder utilizar estas funciones, la documentación exige que primero se ejecute la instrucción SPIFFS.begin( ).

A continuación el método initWifi( ) se encarga de colocar al WemosD1 como Access Point, generar la red WiFi deseada y asignarse una dirección IP. La firma de dicha función recibe como parámetro dos punteros a char que son el nombre de la red y la contraseña.

```
void initWifi(const char* ssid, const char* password);
```

El método initDnsServer( ) inicia un servidor DNS. Se utiliza la librería “DNSServer.h” que corresponde al framework de Arduino. La misma provee funcionalidades para abrir un servidor DNS que corra sobre el Wemos D1, en el puerto 53. Además permite enlazar la dirección IP asignada, en el método previo, a un nombre de dominio. En nuestro caso «www.ledcartel.com». El código fuente de la librería puede encontrarse en el siguiente [link](#).

```
void initDnsServer();
```

El último método que aparece en el init( ) de la clase WebServer, es initWebServer( ). Éste es el que se encarga propiamente de iniciar un servidor web, en el puerto 80.

```
#include <ESP8266WebServer.h>

void initWebServer()
{
    ESP8266WebServer server(80);
    server.onNotFound(handleNotFound);
    server.on("/authentication", HTTP_POST, handlePostAuthentication);
    server.on("/authentication", HTTP_GET, handleGetAuthentication);
    server.on("/phrase", HTTP_POST, handlePostPhrase);
    server.on("/matrix", HTTP_POST, handlePostMatrix);
    server.on("/predefined", HTTP_POST, handlePostPredefined);
    server.on("/party", HTTP_GET, handleGetPartyOn);
    server.on("/", HTTP_GET, handleGetIndex);
    server.on("/static/favicon.png", HTTP_GET, handleGetFavicon);
    server.on("/admin", HTTP_POST, handlePostAdmin);
    server.on("/clear", HTTP_GET, handleGetClear);

    server.begin();
}
```

Inicialmente se incluye la librería “ESP8266WebServer.h” la misma provee funcionalidades para iniciar y manejar un servidor web. Los métodos utilizados en el proyecto se explican a lo largo de esta sección.

En primer lugar se instancia un objeto del tipo ESP8266WebServer, esta clase es la que tiene las funciones que proporciona la librería previamente mencionada. El constructor toma un parámetro que es el puerto donde se van a escuchar las conexiones. En nuestro caso el 80.

El siguiente método es on( ). Éste permite enlazar una petición a un callback, es decir que cuando el cliente realice una acción, por ejemplo un GET REQUEST, se ejecuta la función especificada en el método on( ). En el fragmento 5.2.1 se puede observar que los callbacks que se pasan como parámetros corresponden a los métodos privados de la clase WebServer (ver figura 25). Su funcionamiento se detalla en la sección que se encuentra a continuación.

Posterior a la etapa de configuración, la librería “ESP8266WebServer.h” posee un método para iniciar efectivamente el server y permitir que los clientes accedan a él. Eso se realiza ejecutando la función begin( ).

### 5.2.2. httpHandlers

En la sección anterior se mencionó que la clase WebServer posee diferentes métodos privados denominados callbacks. Éstos se ejecutan ante las diferentes peticiones HTTP que el cliente realiza a través del servidor web hosteado por el Wemos D1 (ver figura 25).

Estos callbacks se clasifican en dos: los que antienden peticiones del tipo GET y los que son del tipo POST. La declaración de sus encabezados permite identificar al grupo al que pertenece cada uno. Por ejemplo handleGetIndex( ) y handlePostMatrix( ). El primero se ejecuta cuando se accede a la dirección “/index” a través de un HTTP GET REQUEST mientras que el segundo cuando se accede a “/matrix” por POST. El comportamiento es muy similar de acuerdo a la categoría donde pertenecen. En este apartado se explica en pseudocódigo las dos categorías previamente enunciadas para estos callbacks.

En primer lugar se explica el comportamiento de los callbacks privados de la clase WebServer que son del tipo GET. Se puede observar que la implementación es muy simple. Cuando se pida “/index” por GET, se va a usar la librería FS.h previamente descripta para cargar en RAM el archivo index.html alojado en la memoria no volátil del microcontrolador.

Luego se envía el HTML como respuesta, hacia el cliente con la respuesta 200 que significa OK. El fragmento 3 muestra el pseudocódigo del comportamiento del callback del tipo GET.

```
void WebServer::handleGetIndex()
{
    String file = SPIFFS.open("/index.html", "r");
    sendResponse(200, file);
}
```

Fragmento 3: Pseudocódigo de un callback del tipo GET

Para el caso de un callback de la categoría POST (ver pseudocódigo del fragmento 4), en primer lugar se realiza la obtención de los argumentos enviados por parte del cliente. Se realiza el procesamiento de esos argumentos, luego se obtiene el archivo HTML que corresponde y se envia por el socket establecido con el usuario cuya respuesta posee el valor 200.

La acción de procesar los argumentos, consiste en determinar qué tipo de pedido está realizando el cliente. Recordar las tres categorías de los mismos: publicar una frase, representar un sprite animado o determinar led a led el estado de cada uno.

Adicionalmente el administrador puede realizar cualquiera de las siguiente 4 opciones: eliminar la última publicación, habilitar/deshabilitar posteos, cambiar la intensidad de los LEDs y activar el modo fiesta.

Una vez que se conoce cuál es el pedido, se ejecuta el método de la clase Letter correspondiente para que envíe la información vía SPI a los MAX7219.

```
void WebServer::handlePostMatrix()
{
    Args args = getArgs();
    process(args);
    String file = SPIFFS.open("/response.html", "r");
    sendResponse(200, file);

}
```

Fragmento 4: Pseudocódigo de un callback del tipo POST

### 5.2.3. tick

Se debe recordar que en la sección 5.1, en el archivo principal, se describe un método denominado loop( ). En cada llamdo, se realiza un tick a cada componente de software que integran el sistema por lo tanto resulta fundamental que dicha función sea no-bloqueante.

La clase WebServer implementa el método tick( ) (ver UML de la figura 25) y en su cuerpo simplemente se encarga de ejecutar la función handleClient de la librería “ESP8266WebServer.h” previamente mencionada. En el siguiente [link](#) se puede observar la documentación completa. Sin embargo alcanza con saber que su principal tarea es comprobar si algún cliente ha realizado alguna petición HTTP. En caso de que la respuesta sea afirmativa, ejecuta el callback correspondiente, si no, retorna inmediatamente sin realizar acción alguna.

```
void WebServer::tick();
```

## 5.3. Clase Letter

La clase Letter es el segundo módulo de software más importante en la solución de este problema. Su principal función consiste en recibir los datos que le pasa por parámetro la clase WebServer, procesarlos y armar un buffer de salida con la información necesaria para encender las luces de la matriz. Para la comunicación con los MAX7219 se utiliza el protocolo de transmisión SPI como se observa en la figura 24.

Los métodos que componen la clase se describen en el diagrama UML de la figura 26. Se puede observar que las funciones públicas son init( ) y tick( ) al igual que en WebServer. Cabe recordar, que este último método no debe ser bloqueante ni quedarse a la espera de un evento asíncrono externo. En cualquier caso, si no cuenta con la información necesaria para ejecutar ciertas acciones, debe retornar inmediatamente.

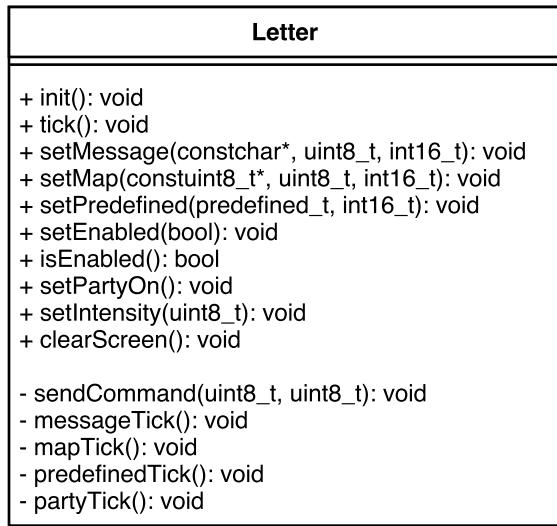


Figura 26: UML de la clase Letter.

### 5.3.1. init

El método `init()` se encarga de configurar todos los registros necesarios del MAX7219 para iniciar su funcionamiento. Entre las acciones que realiza, el programa se encarga de sacarlo del modo test, setearle la intensidad de los leds, indicarle que se van a usar las ocho columnas que posee el chip, limpiarle la pantalla apagando todos los leds y encenderlo. En el fragmento 5 se observa un pseudocódigo de la etapa de inicialización de los registros del MAX7219 para configurarlo de la manera detallada anteriormente.

```

#include <SPI.h>

void Letter::init()
{
    SPI.begin();
    sendCommand(MAX7219_REG_TESTMODE, 0x00); // No test mode
    sendCommand(MAX7219_REG_SHUTDOWN, 0x00); // Shutdown
    sendCommand(MAX7219_REG_DECODEMODE, 0x00); // No decode
    sendCommand(MAX7219_REG_INTENSITY, 0x06); // Medium intensity
    sendCommand(MAX7219_REG_SCANLIMIT, 0x07); // Scan all columns
    clearScreen();
    sendCommand(MAX7219_REG_SHUTDOWN, 0x01); // Turn on
}

```

Fragmento 5: Pseudocódigo del método `init` de la clase `Letter.cpp`

En primer lugar se incluye la librería “`SPI.h`”. La misma corresponde al framework de Arduino y se puede consultar su documentación completa en el siguiente [link](#). Su función principal es abstraer el manejo del protocolo de transmisión serial, a partir de determinadas funciones que la librería provee. Para comenzar a utilizarla se debe ejecutar el método `SPI.begin()`.

Las siguientes instrucciones consisten en configurar todos los MAX7219 a través del método sendCommand( ). La función toma dos parámetros, en el primero se especifica el registro a configurar y en el segundo el valor que se desea establecer. Para más información respecto de la configuración de los registros consultar el datasheet del MAX7219 haciendo click en el siguiente [link](#).

### 5.3.2. Método setMessage

El siguiente método público que aparece en el diagrama UML de la figura 26 es setMessage( ). Esta función es la encargada de procesar las peticiones relacionadas a las frases que el usuario publica en la matriz (ver el diagrama de casos de uso de la figura 23). El método recibe tres parámetros. En primer lugar el mensaje a setear en la matriz de LEDs. La longitud máxima del mensaje a representar está definido por la variable global MESSAGE SIZE.

El siguiente parámetro es strLen, este valor indica la cantidad de caracteres que posee el string message. No debe incluirse el carácter nulo ni tampoco superar el valor de la variable MESSAGE SIZE.

El último parámetro de interés es srate. Su valor indica la velocidad por píxel al que se debe mover el mensaje. Acepta valores negativos y en ese caso el contenido se desliza hacia la izquierda. Caso contrario hacia la derecha.

El comportamiento de este método es muy simple. Toma el mensaje que se recibe, y busca la representación de cada carácter en un arreglo que el microcontrolador posee. Ese arreglo le indica los valores que tiene que enviar a las diferentes columnas de la matriz para representar cada letra. Cada valor es cargado en orden en el búffer de salida.

```
#define MESSAGE_SIZE 6

void Letter::setMessage(const char* message, uint8_t strlen, int16_t srate);
```

### 5.3.3. setMap

Este método es el encargado de procesar las peticiones cuando el usuario quiere encender y apagar determinados LEDs de manera individual (ver el diagrama de casos de uso de la figura 23). El método recibe tres parámetros. En primer lugar un arreglo de bytes donde cada posición representa una columna de la matriz de LEDs y el valor (el byte en esa posición del vector) indica qué leds deben encenderse y cuáles no. Siendo el bit menos significativo la fila cero de la columna en cuestión. El segundo parámetro es la cantidad de columnas a setear. La misma no puede exceder MAX LETTERS \* 8. Si no el comportamiento se torna indefinido. En caso de que el tamaño fuese menor que la cantidad de matrices que hay, se rellena con columnas en 0. Es decir, apagadas. El último valor es srate. Su valor indica la velocidad por píxel al que se debe mover el contenido. El comportamiento de este método consiste en tomar el arreglo de columnas que se recibe por parámetro y copiar dichos valores en el buffer de salida. De esta forma la información queda lista para ser enviada bajo el protocolo de SPI.

```
#define MAX LETTERS 32

void Letter::setMap(const uint8_t* cols, uint8_t columnsCnt, int16_t srate);
```

### 5.3.4. setPredefined

Este método se asocia con el caso de uso relacionado a que el usuario puede elegir representar en la matriz de LEDs una serie de sprites animados previamente cargados en la flash del microcontrolador Wemos D1 (ver el diagrama de casos de uso de la figura 23). El método recibe dos parámetros. El primero corresponde con un enumerativo que determina cuál de todos los sprites precargados se quiere representar. En caso de que se enviara información que no corresponda con ningún sprite, la función retorna inmediatamente. Actualmente los sprites precargados son una sonrisa, un pacman y un corazón animado.

El segundo parámetro es srate. Su valor indica la velocidad por píxel al que se debe mover el sprite. Acepta valores negativos y en ese caso el contenido se desliza hacia la izquierda. Caso contrario hacia la derecha.

El comportamiento de este método consiste en analizar qué sprite se quiere representar, almacenar dicha información en una variable interna y luego llenar el buffer de salida con la información precargada que se tiene de ese sprite. Dichos datos corresponden a los leds que se deben encender en cada columna de la matriz a fin de representar el sprite determinado por el usuario.

```
enum predefined_t{noPredefined = 0, smile, pacman, newLife};  
  
void Letter::setPredefined(predefined_t pre, int16_t slideRate);
```

### 5.3.5. setPartyOn

Este método provee funcionalidad para encender aleatoriamente las luces de la matriz de un modo divertido. Se diseñó especialmente para la demostración del proyecto y tampoco recibe ningún parámetro extra. El comportamiento consiste en llenar el buffer de salida con números generados aleatoriamente con la función de Arduino random( ). Los números van desde 0 a 255 inclusive y representa todas las combinaciones de estados que pueden tomar los LEDs. Siendo 0 todos apagados y 255 todos encendidos. Cada cierto intervalo de tiempo, se renueva los valores del buffer de salida enviando los nuevos estados de los LEDs.

```
void Letter::setPartyOn();
```

### 5.3.6. setEnabled

El funcionamiento de este método consiste en habilitar o deshabilitar las publicaciones que se realicen a la matriz. Recibe un solo parámetro booleano que determina si el sistema acepta nuevas publicaciones. Un valor falso indica que no se pueden enviar nuevos posteos hasta que el flag de enabled vuelva a ser verdadero.

```
void Letter::setEnabled(bool enabled);
```

### 5.3.7. **setIntensity**

Este método provee funcionalidad para cambiar, en tiempo real, la intensidad con la que se encienden los LEDs de la matriz.

Recibe un parámetro de configuración que es un número representado en 8 bits sin signo. El mismo especifica el brillo de las luces y va desde 0 hasta 15 inclusive. En caso de que se introduzca un número superior, se tomarán los 4 bits menos significativos del parámetro.

```
void Letter::setIntensity(uint8_t intesity);
```

### 5.3.8. **clearScreen**

Esta funcionalidad de la clase Letter se utiliza para limpiar el último posteо que realizó el cliente, limpiando el buffer de salida. A fin de eliminar la publicación actual, se coloca cero en cada índice del arreglo del buffer. Luego se envía esa información a cada columna de la matriz.

El método no posee argumentos.

```
void Letter::clearScreen();
```

### 5.3.9. **tick**

Se debe recordar que el método tick( ) de la clase Letter es llamado sólo en el archivo principal main.cpp en su función loop( ). El funcionamiento es muy simple, y solo tiene sentido cuando se elige una velocidad de desplazamiento del contenido distinta a cero. La clase Letter posee determinadas variables internas que le permiten saber cuál es el primer índice del buffer de salida que se debe enviar por serial hacia los MAX7219.

El método, entonces, determina si la velocidad por píxel es positiva o negativa y en función del resultado mueve el índice hacia la izquierda o derecha respectivamente. Luego actualiza la información de los shifters para que representen la nueva imagen desplazada.

```
void Letter::tick();
```

## 6. Guías

### 6.1. Ambiente de desarrollo e instalación

Para los requisitos que se listan a continuación se recomienda utilizar el sistema operativo GNU/Linux. Cabe destacar que el presente proyecto se realizó sobre máquinas con la distribución Ubuntu 16.04.

#### 6.1.1. Git

En primer lugar se necesita la herramienta de software denominada git. La misma está disponible tanto para Linux, como para Windows y macOS. La instalación se puede realizar a través de un instalador binario, en general se utiliza la herramienta básica de administración de paquetes que trae la distribución. En el caso de distribución basada en Debian como Ubuntu, se puede utilizar `apt-get`.

```
$ apt-get install git
```

En caso de un ambiente Windows, se recomienda utilizar [git bash](#). Mientras que para computadoras con sistema operativo macOS se recomienda descargar la imagen en el siguiente [link](#).

#### 6.1.2. Visual Studio Code

A continuación se requiere el editor Visual Studio Code. El mismo se puede descargar de su página principal, haciendo click [aquí](#). Luego se debe elegir la extensión de acuerdo al sistema operativo que se desee. Una vez descargado el archivo, se debe descomprimir su contenido y almacenarlo en alguna carpeta determinada. En el caso de linux, ejecutar el siguiente comando.

```
$ sudo unzip VSCode-linux-x64.zip -d /opt/vscode
```

Lo que hace este comando es descomprimir el contenido del archivo VSCode-linux-x64.zip en la carpeta /opt/vscode. Con el parámetro -d se indica que cree la carpeta en caso de que no exista.

Para ejecutar el programa, se necesita acceder a la carpeta donde se descargó el contenido y luego escribir `./Code`.

```
$ cd /opt/vscode  
$ ./Code
```

Adicionalmente si se quiere tener un acceso directo al escritorio, se debe dirigir a la carpeta donde se descargó el programa, en este caso /opt/vscode y abrir con algún editor de texto con la extensión .desktop. Allí dentro añadimos el siguiente contenido.

```
[Desktop Entry]
Name=Visual Studio Code
Comment=Editor de código Visual Studio Linux
Exec=/opt/vscode/Code
Icon=/opt/vscode/resources/app/vso.png
Terminal=false
Type=Application
```

### 6.1.3. Plugins necesarios para Visual Studio Code

Para el desarrollo de este proyecto se requirió la instalación de los plugins que se listan a continuación en el orden en que son mencionados.

- C++.
- Native Debug.
- PlatformIO IDE.

Para la instalación se puede ingresar en los diferentes links listados, los cuales abrirán una página web y presionando en el botón instalar, automáticamente se añadirá la extensión a Visual Studio Code.

Otra forma de agregar los plugins es abriendo el programa y seleccionado, en la barra de tareas, la pestaña ver-*i* extensiones. Esto desplegará un campo de texto donde se debe escribir el nombre de la extensión a instalar. En nuestro caso se deben ingresar, en mismo orden, los nombres de los plugins listados al inicio de esta sección y apretar en el botón instalar.

## 6.2. Instalación

Todos los archivos fuente que se encuentran en los apéndices B y C, corresponden al programa que se ejecuta dentro del Wemos D1. En este proyecto no se utiliza la PC como parte del sistema.

A fin de brindar más claridad a esta sección se listan todos los nombres de los archivos que se agregan en el microcontrolador. En sus respectivas carpetas (ver listado de la figura 27).

## 6.3. Compilación

Se decidió confeccionar un video ilustrativo que muestre los pasos ordenados que se deben realizar a fin de poder reproducir el proyecto. El enlace se encuentra en el haciendo click en el siguiente [link](#). Adicionalmente, se listan los pasos a continuación de forma que se complementen con el video.

En primer lugar se debe clonar el repositorio: <https://github.com/trorik23/tpii.git> ejecutando el siguiente comando

```
$ git clone https://github.com/trorik23/tpii.git
```

Posteriormente se debe acceder a la carpeta *web* dentro del repositorio, en él se encontrara un script llamado *script.sh*. Una vez localizado, se debe ejecutar el script: *./script.sh*, esto hará

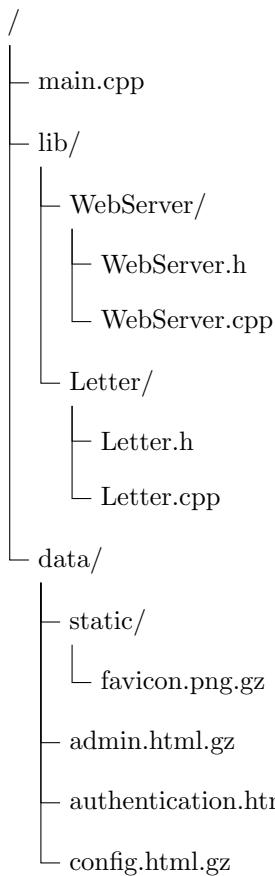


Figura 27: Árbol de archivos del proyecto.

que se carguen los archivos html (comprimidos con gzip) dentro de la carpeta *wemos/data* listo para ser flasheado al Wemos D1.

```
$ cd tpii/web
./script.sh
```

Para realizar pasos listados a continuación, se debe conectar el Wemos D1 a la PC de desarrollo y abrir Visual Studio Code.

- En home de PlatformIO seleccionar «Open Project» y buscar la carpeta «wemos» (donde se encuentra el archivo `platformio.ini`).
  - Dentro de Visual Studio Code, con el proyecto ya importado, seleccionar abajo a la izquierda, el ícono «PlatformIO: Run a Task». Eso desplegará una serie de funcionalidades que se pueden realizar con el Wemos D1. Ahí se debe elegir «PlatformIO: Upload SPIFFS Image». Eso cargará en la memoria del micro los archivos HTML que utilizará el servidor web.
- Este paso solo debe repetirse cada vez que se modifique los archivos HTML.

**ACLARACIÓN:** El proceso de cargar en la SPIFFS del Wemos D1 se realiza mediante serial por lo que no se puede estar ejecutando ninguna otra tarea que involucre serial tales como «SerialMonitor» o «Upload».

- Una vez cargada los archivos HTML, ingresar en la opción «Build».
- Si la compilación del paso anterior fue exitosa, realizar la tarea de «Upload» que se encuentra en las opciones anteriormente mencionadas o en la barra inferior izquierda haciendo click sobre el icono de la flecha.

**ACLARACIÓN:** Cada vez que se modifique el programa se debe rehacer los últimos dos paso.

# Apéndice

## A. Propuesta original

### A.1. Introducción

Los nuevos avances tecnológicos hicieron posible que se pueda incorporar LEDs en casi cualquier parte. De este modo el proyecto se basa en diseñar un cartel de LEDs que pueda ser portable en una remera, controlado de forma remota a través de una página web.

Una remera con cartel LED posee funcionalidades diversas según el ámbito en el que se aplique tales como: en entretenimiento, en festivales de música, en deporte mostrando información de las calorías o pulsaciones del corredor, o incluso identificar a ciclistas y motociclistas para evitar accidentes de tránsito producto de baja visibilidad.

### A.2. Objetivo

A nivel hardware se desarrollará un cartel luminoso, el mismo contendrá 8 LEDs de alto y 24 LEDs de ancho, lo que permitirá representar en un determinado tiempo hasta 3 caracteres del formato ASCII. Esta matriz de LEDs se integrará sobre una remera convirtiéndolo en un sistema portátil. La información a representar será determinada por el usuario el cuál, entre sus opciones, podrá elegir escribir un texto estático, manipular led a led la matriz de forma de determinar cuáles estarán prendidos y cuáles no o por último podrá elegir una animación predefinida que se encontrará cargada previamente en la memoria del microcontrolador.

La comunicación entre el usuario y la remera de LEDs se realizará a partir de un servidor web que mostrará, en una interfaz visual, las opciones anteriormente mencionadas. El sistema web y la red donde estará disponible, será manejado por el microcontrolador; por lo tanto el cliente deberá pertenecer a la misma red. No se permitirá manipular la matriz desde afuera de la red local.

El programa residirá dentro del microcontrolador y su función consistirá en crear una red wifi local de manera que el cliente pueda conectarse a la misma. Por otra parte hosteará un servidor web para que el usuario, dentro de la red, elija la información que se representará en la remera. Por último, se añadirán funcionalidades que permitirán escuchar las peticiones del cliente y procesarlas de modo de obtener, como salida, la información de cuáles leds estarán prendidos y cuáles no en un determinado tiempo. El programa entonces, deberá ser capaz de comunicarse con la matriz y encender los leds acorde a la información proporcionada.

Debido a la complejidad a nivel de hardware, que el proyecto posee, se deja como objetivos secundarios agregar una funcionalidad que permita que el texto ingresado se desplace a modo de marquesina. Por otra parte, cabe recordar que la información ingresada por el cliente se mostrará automáticamente en la remera de LEDs. De esta forma también se propone la implementación de un subsistema donde los mensajes no se muestren de manera instantánea si no que se encuenen hasta que el administrador se autentique y decida cuáles mensajes mostrar y cuáles no.

### A.3. Dispositivos a utilizar

El proyecto requiere lo básico para hacer funcionar una serie de LEDs de tal manera de poder controlar el barrido de lo que se mostrará. A continuación se menciona con detalle el hardware y elementos varios a utilizar:

- Microcontrolador Wemos D1 que posee un chip ESP8266. Costo \$250. (provisto por la cátedra).
- Tira de LEDs simple de color blanco para implementar una matriz de 8x24. Se opta por una tira de LEDs para permitir la flexibilidad de la remera. Costo \$130.

- Tres chips de interfaz serial MAX7219 o similar, el cual permite el control de matrices de LEDs de 8x8. Costo \$135.
- Una Remera negra talle M. Precio \$40.

En resumen, el presupuesto total que se deberá tener para el desarrollo de este proyecto es de \$555.

#### A.4. Esquema gráfico

El esquema general consiste en un sistema web que permita el control de la matriz de LEDs. Y un sistema back-end para administrar las solicitudes del usuario implementado en el microcontrolador utilizando librerías de arduino. Estos subsistemas estarán comunicados a través de el protocolo 802.11, de forma de poder establecer una conexión HTTP. El esquema se puede observar en la figura A.1.

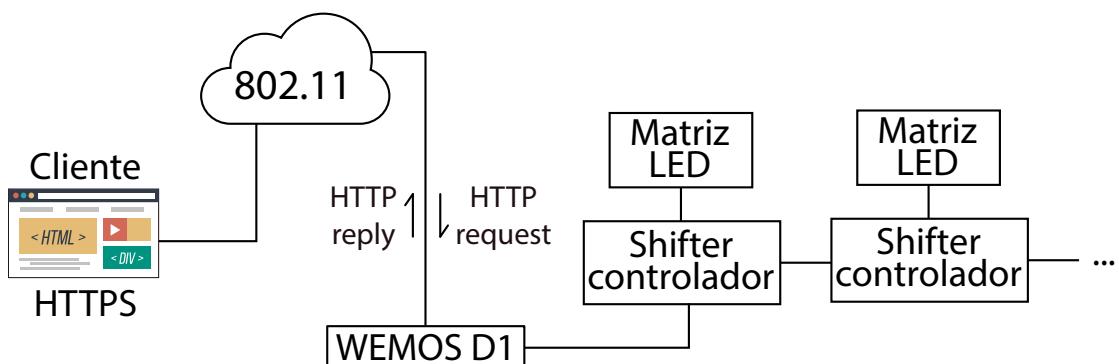


Figura A.1: Diagrama en bloques del sistema.

#### A.5. Identificación de partes

- E/S del controlador con el exterior:** debido a la baja de disponibilidad de pines que ofrece el microcontrolador, es indispensable la multiplexación con los chips de interfaz serial para el control individual de los LEDs. Para ello el microcontrolador se conecta a los Shifters MAX7219 de forma de poder controlar las matrices de LEDs de 8x8. En la figura A.2 se observa el esquema de conexión para 64 leds dispuestos en 8 filas por 8 columnas. El driver MAX7219 implementa una interfaz SPI que consiste en un bus de comunicación a nivel de circuitos integrados en el que la transmisión de datos se realiza en serie. La comunicación con el WEMOS se realiza exclusivamente por medio de 3 pines digitales de salida (DIN, CS y CLK), además de dos pines para la alimentación (+5V) y la masa (GND).
- Comunicaciones:** utilizando las librerías existentes del modulo ESP8266, se configurará el microcontrolador como Access Point y como Web Server, de manera de independizar al sistema de una conexión a Internet. Entre las librerías a utilizar se encuentran “ESP8266WebServer.h”, “WiFiClient.h” y “ESP8266Wifi.h”. Esto permitirá montar una red WIFI local y hostear un servidor desde el microcontrolador. El programa se desarrollará bajo el framework de Arduino utilizando el lenguaje c++. Por otra parte, el manejo de las peticiones que el cliente realice, se determinarán a nivel de capa de enlace mediante el protocolo WiFi (802.11) y no será posible comunicarse con el sistema a través de un dispositivo conectado en una red externa. Dichas peticiones y respuestas se transmitirán bajo

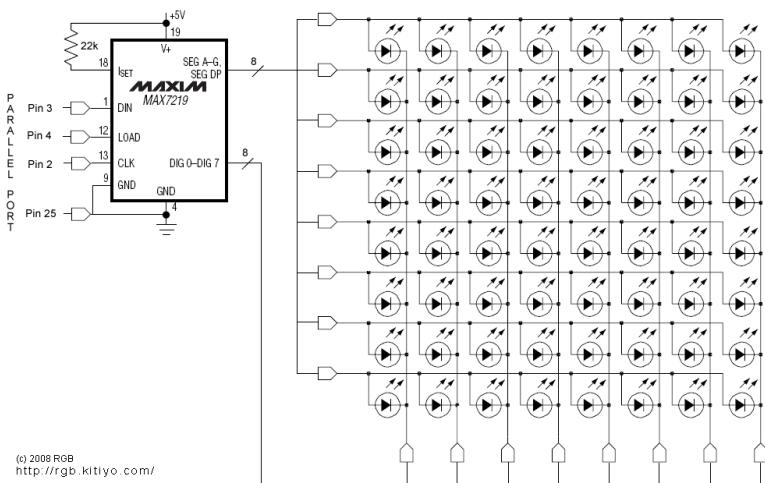


Figura A.2: Conexión de un Shifter con una matriz de 8x8 LEDs.

el protocolo HTTP. Las librerías previamente mencionadas permiten la implementación de este protocolo sobre el microcontrolador.

- c. **Sistema web:** el microcontrolador hostea una página web de manera tal que permita la comunicación entre los clientes y el WEMOS. En la figura A.3 se observa un campo de texto para que el usuario pueda ingresar una frase. En principio las letras serán estáticas, sin embargo, se plantea como objetivo secundario implementar una opción para mostrarlas a modo de marquesina.

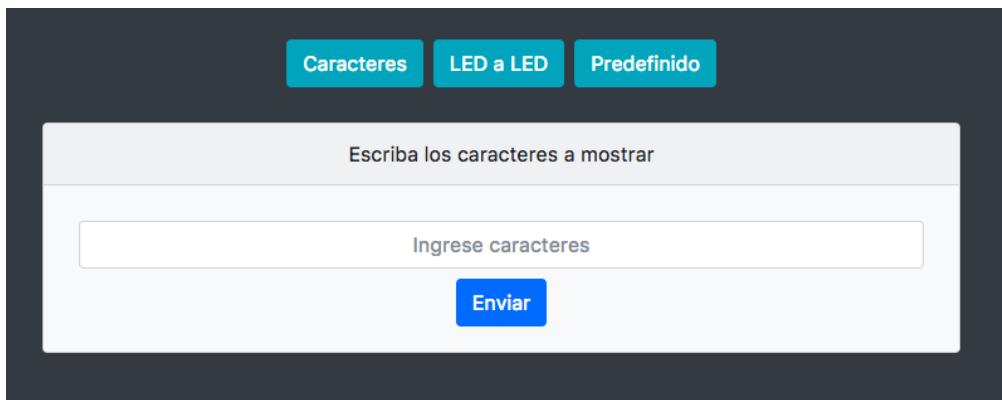


Figura A.3: Interfaz para ingresar un texto.

Por otra parte, si el cliente lo desea, puede manipular los leds individualmente a través de una matriz de puntos que se encuentra en la página web. De esta forma, puede encender y apagar las luces que desee. En la figura A.4 se muestra la interfaz gráfica a través de la cual se implementa dicha funcionalidad.

Además los usuarios pueden elegir diferentes imágenes precargadas en la memoria del microcontrolador; dichas animaciones se pueden observar en la figura A.5

En principio las peticiones del usuario se plasman automáticamente sobre la remera de

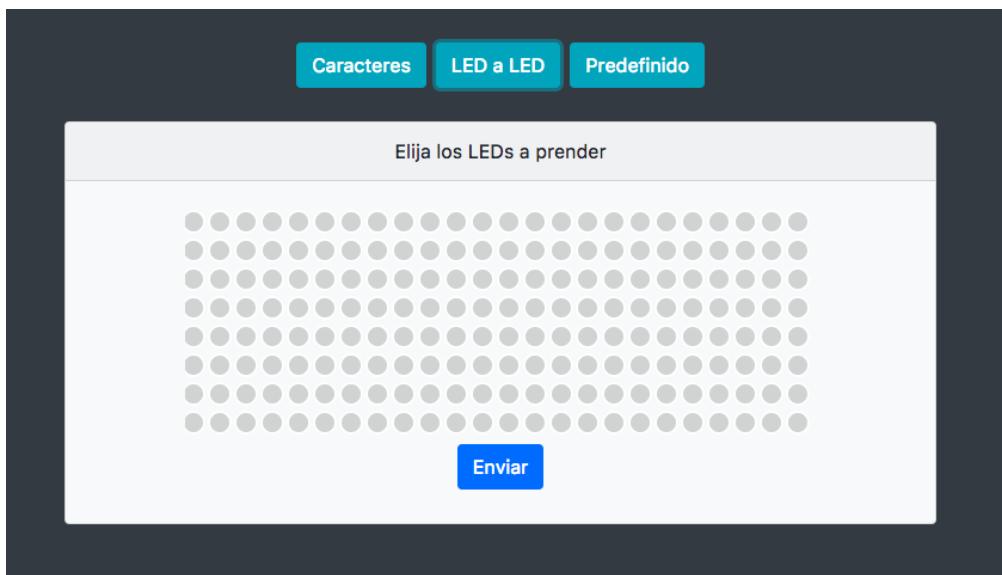


Figura A.4: Interfaz para manipular LEDs individualmente.



Figura A.5: Interfaz para elegir imágenes precargadas en memoria.

LEDs. Sin embargo se propone un objetivo secundario donde dichos pedidos se encuen-  
tran en la memoria del microcontrolador. De esta forma, debería implementarse un subsistema  
para que el administrador, como por ejemplo, el dueño de la remera, pueda determinar qué  
mensajes se muestran y cuáles son descartados. Para ello se dispone de una interfaz donde  
el dueño pueda autenticarse y aceptar mensajes.

## B. Código fuente Servidor Web

El programa entero que se escribió para este proyecto se encuentra en el repositorio de github haciendo click en el [link](#). Adicionalmente se adjunta el material en el presente apéndice.

### B.1. main.cpp

```
1 #include <Arduino.h>
2 #include <WebServer.h>
3 #include <Letter.h>
4
5 void setup()
6 {
7     Letter::init();
8     WebServer::init();
9 }
10
11 void loop()
12 {
13     WebServer::tick();
14     Letter::tick();
15 }
```

## B.2. WebServer.h

```
1 #pragma once
2
3
4 #define WEB_NAME "www.ledcartel.com"
5 #define WEB_SSID "Remera.LED"
6 #define WEB_PASSWORD "12345678"
7
8 #define TOKEN "tres_tristes_tigres"
9
10
11
12 // Esta clase se encarga de establecer una red Wifi, con las credenciales
13 // definidas en el encabezado de este archivo. Además inicia un servidor DNS
14 // para asociar el nombre del dominio con la dirección IP que el Wemos D1
15 // posee. De esta forma se puede acceder al servidor WEB sin tener que recordar
16 // la dirección IP del microcontrolador.
17 class WebServer {
18
19 public:
20
21     // Inicializa la red Wifi, con las credenciales definidas en este archivo.
22     // Además inicia el servidor DNS y el servidor WEB seteando los callbacks,
23     // para cada petición que realice el cliente,
24     static void init();
25
26     // Atiende los pedidos que realice el cliente ya sea para el servidor DNS o
27     // para el servidor WEB.
28     static void tick();
29
30 private:
31
32     // Maneja el POST del proceso de autenticación.
33     static void handlePostAuthentication();
34     // Maneja el GET del proceso de autenticación.
35     static void handleGetAuthentication();
36     // Maneja el evento, cuando no se encuentra la dirección solicitada.
37     static void handleNotFound();
38     // Maneja el POST del evento de postear una frase.
39     static void handlePostPhrase();
40     // Maneja el POST del proceso de postear una matriz.
41     static void handlePostMatrix();
42     // Maneja el POST del proceso de postear un sprite predefinido.
43     static void handlePostPredefined();
44     // Maneja el POST del proceso de partyOn.
45     static void handleGetPartyOn();
46     // Maneja el GET del proceso de index.
47     static void handlegetIndex();
48     // Maneja el POST del proceso de autenticación.
49     static void handleGetAdmin();
50     // Devuelve los archivos css.
51     static void handleCss();
52     // Devuelve los archivos js.
53     static void handleJs();
```

```
54 // Devuelve el ícono.  
55 static void handleGetFavicon();  
56 // Maneja el GET del clear de la pantalla  
57 static void handleGetClear();  
58 // Maneja el POST del proceso de configuracion de admin  
59 static void handlePostAdmin();  
60 // Devuelve el estado actual del sistema  
61 static void handleGetState();  
62  
63 };
```

### B.3. WebServer.cpp

```
1 #include <WebServer.h>
2 #include <ESP8266WiFi.h>
3 #include <DNSServer.h>
4 #include <ESP8266WebServer.h>
5 #include <Arduino.h>
6 #include <FS.h>
7 #include <Letter.h>
8
9 static IPAddress apIP(192, 168, 0, 128);
10 static DNSServer dnsServer;
11 static ESP8266WebServer server(80);
12
13 void WebServer::init()
14 {
15     SPIFFS.begin();
16
17     WiFi.mode(WIFI_AP);
18     WiFi.softAPConfig(apIP, apIP, IPAddress(255, 255, 255, 0));
19     WiFi.softAP(WEB_SSID, WEB_PASSWORD);
20     dnsServer.setTTL(30);
21     dnsServer.setErrorReplyCode(DNSReplyCode::ServerFailure);
22     dnsServer.start(53, WEB_NAME, apIP);
23
24     server.onNotFound(handleNotFound);
25     server.on("/authentication", HTTP_POST, handlePostAuthentication);
26     server.on("/authentication", HTTP_GET, handleGetAuthentication);
27     server.on("/phrase", HTTP_POST, handlePostPhrase);
28     server.on("/matrix", HTTP_POST, handlePostMatrix);
29     server.on("/predefined", HTTP_POST, handlePostPredefined);
30     server.on("/party", HTTP_GET, handleGetPartyOn);
31     server.on("/", HTTP_GET, handleGetIndex);
32     server.on("/static/favicon.png", HTTP_GET, handleGetFavicon);
33     server.on("/admin", HTTP_POST, handlePostAdmin);
34     server.on("/clear", HTTP_GET, handleGetClear);
35     server.on("/state", handleGetState);
36
37     server.begin();
38 }
39
40 void WebServer::tick()
41 {
42     dnsServer.processNextRequest();
43     server.handleClient();
44 }
45
46 void WebServer::handleGetState()
47 {
48     server.send(200, "text/plain", String(Letter::getIntensity() + 1) + "/" +
49     ↪ String(Letter::isEnabled()));
50 }
51 void WebServer::handleGetFavicon()
52 {
```

```

53     File f = SPIFFS.open("/static/favicon.png.gz", "r");
54     server.streamFile(f, "image/png");
55     f.close();
56 }
57
58 void WebServer::handleGetIndex()
59 {
60     File f = SPIFFS.open("/config.html.gz", "r");
61     server.streamFile(f, "text/html");
62     f.close();
63 }
64
65 void WebServer::handleGetAdmin()
66 {
67     File f = SPIFFS.open("/admin.html.gz", "r");
68     server.streamFile(f, "text/html");
69     f.close();
70 }
71
72 void WebServer::handleNotFound()
73 {
74     server.sendHeader("Location", String("/"), true);
75     server.send(302, "text/plain", "");
76 }
77
78 void WebServer::handlePostAuthentication()
79 {
80     if (server.arg("token").equals(TOKEN)) {
81         handleGetAdmin();
82     } else {
83         handleGetAuthentication();
84     }
85 }
86
87 void WebServer::handleGetAuthentication()
88 {
89     File f = SPIFFS.open("/authentication.html.gz", "r");
90     server.streamFile(f, "text/html");
91     f.close();
92 }
93
94 void WebServer::handleGetPartyOn()
95 {
96     Letter::setPartyOn();
97     handleGetAdmin();
98 }
99
100 void WebServer::handlePostPhrase()
101 {
102     String value = server.arg("phrase");
103     uint8_t size = value.length() + 1;
104     char message[size];
105     value.toCharArray(message, size);
106
107     Letter::setMessage(message, size - 1, server.arg("sliderate").toInt());
108
109     server.sendHeader("Location", String("/"), true);

```

```

110     server.send(302, "text/plain", "");
111 }
112
113 void WebServer::handlePostMatrix()
114 {
115     uint8_t columns[2 * MAX_COLUMNS];
116
117     for (uint8_t x = 0; x < 2 * MAX_COLUMNS; x++) {
118         columns[x] = 0;
119         for (int8_t y = MAX_COLUMNS - 1; y >= 0 ; y--) {
120             String args = String(y) + "-" + String(x);
121             columns[x] |= (server.arg(args) == "1" ? 1 : 0) << y;
122         }
123     }
124
125     Letter::setMap(columns, 2 * MAX_COLUMNS, server.arg("sliderate").toInt())
126     ↗ ;
127
128     server.sendHeader("Location", String("/"), true);
129     server.send(302, "text/plain", "");
130 }
131
132 void WebServer::handlePostPredefined()
133 {
134     String value = server.arg("image-predif");
135     uint16_t srate = server.arg("sliderate").toInt();
136
137     Serial.println("hola");
138
139     if (value.equals("smile-face"))
140         Letter::setPredefined(Letter::predefined_t::smile, srate);
141     else if (value.equals("pacman"))
142         Letter::setPredefined(Letter::predefined_t::pacman, srate);
143     else if (value.equals("new-life"))
144         Letter::setPredefined(Letter::predefined_t::newLife, srate);
145     else
146         Letter::setPredefined(Letter::predefined_t::noPredefined, 0);
147
148     server.sendHeader("Location", String("/"), true);
149     server.send(302, "text/plain", "");
150 }
151
152 void WebServer::handlePostAdmin(){
153     uint16_t brightness = server.arg("brightness").toInt();
154     String enabled = server.arg("enabled");
155
156     Letter::setIntensity(brightness - 1);
157     if(enabled.equals("on"))
158         Letter::setEnabled(true);
159     else
160         Letter::setEnabled(false);
161     handleGetAdmin();
162 }
163
164 void WebServer::handleGetClear(){
165     Letter::clearScreen();
166     handleGetAdmin();

```

166 }

## C. Código Fuente controlador matriz

### C.1. Letter.h

```
1 #pragma once
2
3 #include <Arduino.h>
4
5 // Constante que define la cantidad de matrices de 8x8 se conectan al sistema.
6 #define LETTERS_COUNT 2
7
8 // La cantidad de columnas que tiene cada matriz de LEDs.
9 #define MAX_COLUMNS 8
10 // La máxima cantidad de matrices que se pueden conectar en el sistema.
11 #define MAX LETTERS 32
12 // La longitud máxima del mensaje que se puede escribir. Este valor no incluye al
13 // carácter nulo.
14 #define MESSAGE_SIZE 25
15
16 #define RAW_DATA_SIZE 33
17
18
19
20 // Esta clase es la encargada de controlar la matriz de LEDs, su principal
21 // función consiste en configurar cada MAX7219 que haya conectado. Luego se
22 // encarga de recibir los datos desde el servidor, procesarlos y armar un
23 // buffer de salida. Dicho arreglo de bytes son los que se envían vía SPI hacia
24 // los chips shifters.
25 class Letter {
26
27 public:
28
29     // Tipo de enumerativo que se utiliza para indicar cuál sprite precargado
30     // se quiere utilizar para representar.
31     enum predefined_t{noPredefined = 0, smile, pacman, newLife};
32
33     // Configura todos los MAX7219, y apaga todos los LEDs. Luego se queda a la
34     // espera de las peticiones que vienen desde el servidor WEB.
35     static void init();
36
37     // Setea el mensaje a mostrar. La variable 'strlen' no incluye al 0.
38     // Tampoco debe superar el máximo definido por MESSAGE_SIZE. Si eso
39     // → sucede,
40     // el comportamiento es indefinido.
41     static void setMessage(const char* message, uint8_t strlen, int16_t srate);
42
43     // La cantidad de columnas a setear debe ser inferior a MAX_COLUMNS *
44     // MAX LETTERS. En caso de que columnsCnt fuese inferior a
45     // → mLetterCount *
46     // MAX_COLUMNS, se rellena, las columnas restantes con blanco. Si fuese
47     // superior, debe asegurarse que columnsCnt sea un múltiplo de
48     // MAX_COLUMNS, en caso contrario, podría imprimirse basura en las
49     // → columnas
50     // no seteadas.
51     static void setMap(const uint8_t* cols, uint8_t columnsCnt, int16_t srate);
```

```

49      // El predefinido que se establece debe ser del tipo predefined_t definido
50      // en este archivo. En caso de que de alguna forma se enviara informaci n
51      // que no corresponda, este m todo no realiza ninguna funci n.
52      static void setPredefined(predefined_t pre, int16_t slideRate);
53
54      // Inicia un modo donde enciende luces al azar y prende y apaga la pantalla
55      // de la matriz. En caso de que se ejecute por segunda vez, mantiene el
56      // mismo efecto.
57      static void setPartyOn();
58
59      // Habilita o deshabilita los posteos en la matriz. Deshabilitar implica
60      // mantener el estado actual de la matriz pero no permite realizar ninguna
61      // otra operaci n.
62      static void setEnabled(bool enabled);
63
64      // Devuelve el valor respectivo a si el sistema est  habilitado.
65      static bool isEnabled();
66
67      // Setea la intensidad de todos los LED de los MAX2719. El valor 'intensity'
68      // que se recibe por par metro especifica la intensidad de los leds en sus
69      // cuatro bits menos significativos. Los otros cuatro bits no producen
70      // efecto alguno.
71      static void setIntensity(uint8_t intesity);
72
73      // Devuelve el valor respectivo a la intensidad.
74      static uint8_t getIntensity();
75
76      // Limpia la pantalla de todos los MAX7219 conectados en el circuito.
77      // Adem s limpia el \'ltimo posteo realizado, de forma que no se siga
78      // ejecutando sus ticks.
79      static void clearScreen();
80
81      // Genera un tick al sistema, en base a la configuraci n de sus variables
82      // privadas, determina si tiene que actualizar o no su buffer de salida. En
83      // caso de que la respuesta fuese s , entonces debe enviar nuevamente la
84      // informaci n v a SPI hacia los distintos MAX7219.
85      static void tick();
86
87
88
89
90 private:
91
92     // Este tipo de dato determina el tipo de petici n que se ha recibido desde
93     // el servidor WEB. En caso de no haber recibido ninguna, el valor por
94     // defecto es noType.
95     enum type_t{noType = 0, message, map, predefined, party};
96
97     // La cantidad de matrices de LEDs de 8x8 conectadas en el sistema.
98     static uint8_t mLetterCount;
99
100    // Determina si permite nuevas operaciones.
101    static bool mEnabled;
102
103    // Determina la intensidad de la matriz.
104    static uint8_t mIntensity;
105

```

```

106 // El buffer de salida que se envía mediante SPI hacia los chips MAX7219.
107 static uint8_t mCommandBuffer[MAX_COLUMNS * MAX LETTERS];
108
109 // El tipo de petición que se está procesando. En caso de no haber recibido
110 // ninguna, el valor por defecto es noType.
111 static type_t mType;
112
113 // Un arreglo de bytes que posee toda la configuración necesaria que el
114 // controlador necesita.
115 static char mRaw[RAW DATA SIZE];
116
117 // El comando se envia a todos los MAX7219 conectados en el circuito. No se
118 // debe utilizar para setear individualmente una columna de un MAX7219.
119 static void sendCommand(uint8_t address, uint8_t value);
120
121 // Es el método que controla el cartel, cuando lo que se seteo fue un
122 // texto.
123 static void messageTick();
124
125 // Es el método que controla el cartel, cuando lo que se seteo fue un map.
126 static void mapTick();
127
128 // Es el método que controla el cartel, cuando lo que se seteo fue un
129 // sprite predefinido.
130 static void predefinedTick();
131
132 // Es el método que controla el cartel, cuando lo que se seteo fue activar
133 // la fiesta.
134 static void partyTick();
135
136 };

```

## C.2. Letter.cpp

```
1 #include <Letter.h>
2 #include <SPI.h>
3 #include <Font.h>
4
5 // La cantidad de milisegundos que debe esperarse hasta ejecutarse un nuevo
6 // tick del sistema.
7 #define TICKS 10
8
9
10 // Registros del MAX7219.
11 static constexpr uint8_t MAX7219_REG_NOOP = 0x0;
12 static constexpr uint8_t MAX7219_REG_DIGIT0 = 0x1;
13 static constexpr uint8_t MAX7219_REG_DIGIT1 = 0x2;
14 static constexpr uint8_t MAX7219_REG_DIGIT2 = 0x3;
15 static constexpr uint8_t MAX7219_REG_DIGIT3 = 0x4;
16 static constexpr uint8_t MAX7219_REG_DIGIT4 = 0x5;
17 static constexpr uint8_t MAX7219_REG_DIGIT5 = 0x6;
18 static constexpr uint8_t MAX7219_REG_DIGIT6 = 0x7;
19 static constexpr uint8_t MAX7219_REG_DIGIT7 = 0x8;
20 static constexpr uint8_t MAX7219_REG_DECODEMODE = 0x9;
21 static constexpr uint8_t MAX7219_REG_INTENSITY = 0xA;
22 static constexpr uint8_t MAX7219_REG_SCANLIMIT = 0xB;
23 static constexpr uint8_t MAX7219_REG_SHUTDOWN = 0xC;
24 static constexpr uint8_t MAX7219_REG_TESTMODE = 0xF;
25
26 // La estructura que se utiliza para almacenar la información de configuración
27 // cuando la petición corresponde a una frase.
28 struct text_t {
29     // No incluye al carácter 0.
30     uint8_t textLength;
31     // El mensaje incluyendo al carácter 0.
32     char message[MESSAGE_SIZE + 1];
33     // Representa la última letra a dibujar. Va desde 0 hasta textLength
34     // exclusive.
35     uint8_t letterIndex;
36     // Representa la última columna de la letra a dibujar. Va desde 0 hasta
37     // MAX_COLUMNS exclusive.
38     uint8_t columnIndex;
39 } __attribute__((packed));
40
41
42 // La estructura que se utiliza para almacenar la información de configuración
43 // cuando la petición corresponde a una mapeo de leds.
44 struct map_t {
45     // La cantidad de columnas que tiene el mapeo.
46     uint8_t columnsCount;
47     // Representa la primera columna que se dibuja. Va desde 0 hasta
48     // columnsCount
49     // exclusive.
50     uint8_t columnIndex;
51 } __attribute__((packed));
52 // La estructura que se utiliza para almacenar la información de configuración
```

```

53 // cuando la petición corresponde a un sprite predefinido.
54 struct predefined_t {
55     // Determina cual animación se está representando.
56     uint8_t sprite;
57     // Determina cuál sprite de la animación se está representando.
58     uint8_t spriteIndex;
59     // Determina cuantos sprites tiene la animación seleccionada.
60     uint8_t spritesCount;
61     // Determina la cantidad de ms que debe pasar hasta cambiar el indice.
62     int16_t fpms;
63     // Contador de ticks, que determina cuantos ticks faltan para ejecutar un
64     // refreshado.
65     int16_t remainingRefreshTicks;
66     // La cantidad de columnas que tiene la animación.
67     uint8_t columnsCount;
68     // Representa la primera columna que se dibuja. Va desde 0 hasta
69     // columnsCount exclusive.
70     uint8_t columnIndex;
71 } __attribute__((packed));
72
73 struct party_t {
74     // Flag que determina si esta encendido o apagado.
75     uint8_t on;
76     // La cantidad de ticks restantes para tooglear.
77     int16_t remainingTicks;
78     // El valor por defecto.
79     int16_t ticks;
80 } __attribute__((packed));
81
82
83 // La estructura base, que contiene información de configuración general a
84 // todas las peticiones.
85 struct base_t {
86     // El valor esta en ms. El signo determina hacia donde se mueve. Negativo
87     // implica hacia la izquierda.
88     int16_t srate;
89     // Contador de ticks, que determina cuantos ticks faltan para ejecutar un
90     // slide.
91     int16_t remainingSlideTicks;
92     union {
93         text_t text;
94         map_t map;
95         predefined_t predefined;
96         party_t party;
97     };
98 } __attribute__((packed));
99
100 static_assert(sizeof(base_t) <= RAW_DATA_SIZE, "Error_en_la_longitud_de_
101     ↪ rawData");
102
103 uint8_t Letter::mLetterCount;
104 uint8_t Letter::mCommandBuffer[MAX_COLUMNS * MAX LETTERS];
105 char Letter::mRaw[RAW_DATA_SIZE];
106 Letter::type_t Letter::mType;
107 bool Letter::mEnabled;
108 uint8_t Letter::mIntensity;

```

```

109
110 // Realiza la copia desde el puntero src, hasta dst, indicando cuantos bytes se
111 // quieren copiar. Mientras se realiza la copia, se busca en src un terminador
112 // 0. En caso de que no se encuentre se coloca en dst un terminador 0 al inicio.
113 static int strcpy_s(char *dest, unsigned int dmax, const char *src);
114
115 // Hace la cuenta del valor que se pasa por referencia. Si excede el máximo
116 // inicia en 0 nuevamente. En cambio si restando se llega por debajo de 0,
117 // la cuenta arranca en maxValue - 1.
118 static void countWithModule1(uint8_t &value, uint8_t maxValue, bool ascnd);
119
120 // Realiza dos cuentas con módulo recursivamente.
121 static void countWithModule2(uint8_t &s, uint8_t mxs, uint8_t &mn, uint8_t
122     ↪ mxmn, bool ascnd);
123
124 void Letter::init()
125 {
126     mLetterCount = LETTERS_COUNT;
127     pinMode(SS, OUTPUT);
128     SPI.begin();
129     sendCommand(MAX7219_REG_TESTMODE, 0x00); // No test mode
130     sendCommand(MAX7219_REG_SHUTDOWN, 0x00); // Shutdown
131     sendCommand(MAX7219_REG_DECODEMODE, 0x00); // No decode
132     setIntensity(0x0F); // High intensity
133     sendCommand(MAX7219_REG_SCANLIMIT, 0x07); // Scan all columns
134     clearScreen();
135     sendCommand(MAX7219_REG_SHUTDOWN, 0x01); // Turn on
136     randomSeed(analogRead(0));
137     mType = noType;
138     mEnabled = true;
139 }
140
141 void Letter::sendCommand(uint8_t address, uint8_t value)
142 {
143     digitalWrite(SS, LOW);
144     for (uint8_t i = 0; i < mLetterCount; i++) {
145         SPI.transfer(address);
146         SPI.transfer(value);
147     }
148     digitalWrite(SS, HIGH);
149 }
150
151 void Letter::clearScreen()
152 {
153     for(uint8_t i = 0; i < MAX_COLUMNS; i++) {
154         sendCommand(MAX7219_REG_DIGIT0 + i, 0x00);
155     }
156     mType = noType;
157 }
158
159 void Letter::setMessage(const char* message, uint8_t strLen, int16_t srate)
160 {
161     if (!mEnabled)
162         return;
163     base_t* base = reinterpret_cast<base_t*>(mRaw);

```

```

165     strcpy_s(base->text.message, strLen + 1, message);
166
167     uint8_t maxSize = (strLen > mLetterCount) ? strLen : mLetterCount;
168
169     for(int8_t j = strLen; j < maxSize; j++)
170         base->text.message[j] = '_';
171
172     base->text.message[maxSize] = '_';
173     base->text.message[maxSize+1] = 0;
174
175     base->srate = srate;
176     base->remainingSlideTicks = 0;
177     base->text.textLength = maxSize+1;
178     base->text.letterIndex = mLetterCount - 1;
179     base->text.columnIndex = MAX_COLUMNS - 1;
180
181     mType = type_t::message;
182 }
183
184 void Letter::setMap(const uint8_t* cols, uint8_t colCnt, int16_t srate)
185 {
186     if (!mEnabled)
187         return;
188
189     for(uint8_t i = 0; i < colCnt; i++) {
190         mCommandBuffer[i] = cols[i];
191     }
192     uint8_t maxColumns = mLetterCount * MAX_COLUMNS;
193     for(uint8_t i = colCnt; i < maxColumns; i++) {
194         mCommandBuffer[i] = 0x00;
195     }
196
197     base_t* base = reinterpret_cast<base_t*>(mRaw);
198     base->srate = srate;
199     base->remainingSlideTicks = 0;
200     base->map.columnsCount = (colCnt < maxColumns) ? maxColumns :
201         ↪ colCnt;
202     base->map.columnIndex = 0;
203
204     mType = type_t::map;
205 }
206
207 void Letter::setPredefined(Letter::predefined_t pre, int16_t srate)
208 {
209     if (!mEnabled)
210         return;
211
212     if (pre == noPredefined) {
213         mType = type_t::noType;
214         return;
215     }
216
217     base_t* base = reinterpret_cast<base_t*>(mRaw);
218     base->predefined.sprite = predefined_config[pre - 1][2];
219     base->predefined.spriteIndex = 0;
220     base->predefined.spritesCount = predefined_config[pre - 1][0];
221     base->predefined.fmps = predefined_config[pre - 1][1];

```

```

221     base->predefined.remainingRefreshTicks = 0;
222     base->remainingSlideTicks = (srate < 0) ? -srate : srate;
223     base->predefined.columnsCount = mLetterCount * MAX_COLUMNS;
224     base->predefined.columnIndex = 0;
225     base->srate = srate;
226
227     mType = type_t::predefined;
228 }
229
230 void Letter::setPartyOn()
231 {
232     if (!mEnabled)
233         return;
234
235     if (mType == type_t::party)
236         return;
237
238     base_t* base = reinterpret_cast<base_t*>(mRaw);
239     base->party.on = 0;
240     base->party.remainingTicks = 0;
241     base->party.ticks = 100;
242
243     mType = type_t::party;
244 }
245
246 void Letter::setEnabled(bool enabled)
247 {
248     mEnabled = enabled;
249 }
250
251 bool Letter::isEnabled()
252 {
253     return mEnabled;
254 }
255
256 void Letter::setIntensity(uint8_t intensity)
257 {
258     sendCommand(MAX7219_REG_INTENSITY, intensity & 0x0F);
259     mIntensity = intensity & 0x0F;
260 }
261
262 uint8_t Letter::getIntensity()
263 {
264     return mIntensity;
265 }
266
267 void Letter::tick()
268 {
269     switch(mType) {
270     case type_t::message:
271         messageTick();
272         break;
273
274     case type_t::map:
275         mapTick();
276         break;
277

```

```

278     case type_t::predefined:
279         predefinedTick();
280         break;
281
282     case type_t::party:
283         partyTick();
284         break;
285
286     default:
287         break;
288     }
289
290     delay(TICKS);
291 }
292
293 void Letter::messageTick()
294 {
295     base_t* base = reinterpret_cast<base_t*>(mRaw);
296
297     base->remainingSlideTicks -= TICKS;
298     if (base->remainingSlideTicks > 0)
299         return;
300
301     base->remainingSlideTicks = (base->srate < 0) ? -base->srate : base->
302         ↳ srate;
303
304     uint8_t i = base->text.columnIndex;
305     uint8_t I = base->text.letterIndex;
306
307     for (uint8_t j = 0; j < mLetterCount * MAX_COLUMNS; j++) {
308         mCommandBuffer[j] = font[base->text.message[I] * MAX_COLUMNS + i
309             ↳ ];
310         countWithModule2(i, MAX_COLUMNS, I, base->text.textLength, false);
311     }
312
313     for (uint8_t j = 1; j <= MAX_COLUMNS; j++) {
314         digitalWrite(SS, LOW);
315
316         for (uint8_t k = 0; k < mLetterCount; k++) {
317             uint16_t word = ((MAX_COLUMNS - (j - 1)) << 8);
318             uint8_t value = mCommandBuffer[(j - 1) + k * MAX_COLUMNS];
319             SPI.transfer16(word | value & 0x00FF);
320         }
321
322         digitalWrite(SS, HIGH);
323     }
324
325     if (base->srate) {
326         countWithModule2(base->text.columnIndex, MAX_COLUMNS,
327                         base->text.letterIndex, base->text.textLength,
328                         (base->srate < 0));
329     } else {
330         mType = type_t::noType;
331     }
332 }
```

```

333 void Letter::mapTick()
334 {
335     base_t* base = reinterpret_cast<base_t*>(mRaw);
336
337     base->remainingSlideTicks -= TICKS;
338     if (base->remainingSlideTicks > 0)
339         return;
340
341     base->remainingSlideTicks = (base->srate < 0) ? -base->srate : base->
342         ↪ srate;
343
344     for (uint8_t j = 1; j <= MAX_COLUMNS; j++) {
345         digitalWrite(SS, LOW);
346
347         for (int8_t k = mLetterCount - 1; k >= 0; k--) {
348             uint16_t word = (j << 8);
349             uint8_t value = mCommandBuffer[((j - 1) + k * MAX_COLUMNS +
350                 base->map.columnIndex) % base->map.columnsCount];
351
352             SPI.transfer16(word | value & 0x00FF);
353         }
354
355         digitalWrite(SS, HIGH);
356     }
357
358     if (base->srate)
359         countWithModule1(base->map.columnIndex, base->map.columnsCount,
360                         (base->srate < 0));
361     else
362         mType = type_t::noType;
363 }
364
365 void Letter::predefinedTick()
366 {
367     base_t* base = reinterpret_cast<base_t*>(mRaw);
368     bool hasToRefresh = false;
369     bool hasToSlide = false;
370
371     base->predefined.remainingRefreshTicks -= TICKS;
372     if (base->predefined.remainingRefreshTicks <= 0) {
373         base->predefined.remainingRefreshTicks = base->predefined.fpm;
374         hasToRefresh = true;
375     }
376
377     base->remainingSlideTicks -= TICKS;
378     if (base->srate && base->remainingSlideTicks <= 0) {
379         base->remainingSlideTicks = (base->srate < 0) ? -base->srate :
380             base->srate;
381         hasToSlide = true;
382     }
383
384     if (hasToRefresh | hasToSlide) {
385         uint8_t i = base->predefined.columnIndex;
386
387         for(uint8_t j = 0; j < mLetterCount * MAX_COLUMNS; j++)
388             mCommandBuffer[j] = 0;
389     }
390 }
```

```

389     for (uint8_t j = 0; j < MAX_COLUMNS; j++) {
390         mCommandBuffer[i] = predefined_values[base->predefined.sprite +
391             ↪ base->predefined.spriteIndex][j];
392         countWithModule1(i, base->predefined.columnsCount, true);
393     }
394
395     for(uint8_t j = 0; j < MAX_COLUMNS; j++) {
396
397         digitalWrite(SS, LOW);
398         for(int8_t k = mLetterCount - 1; k >= 0; k--) {
399
400             uint8_t address = j + 1;
401             uint8_t value = mCommandBuffer[k * MAX_COLUMNS + j];
402
403             SPI.transfer16(address << 8 | (value & 0x00FF));
404         }
405         digitalWrite(SS, HIGH);
406     }
407
408     if (hasToRefresh)
409         countWithModule1(base->predefined.spriteIndex, base->predefined.
410             ↪ spritesCount, true);
411
412     if(hasToSlide)
413         countWithModule1(base->predefined.columnIndex, base->predefined.
414             ↪ columnsCount, (base->srate > 0));
415     }
416
417     void Letter::partyTick()
418     {
419         base_t* base = reinterpret_cast<base_t*>(mRaw);
420         base->party.remainingTicks -= TICKS;
421         if (base->party.remainingTicks > 0)
422             return;
423
424         base->party.remainingTicks = base->party.ticks;
425
426         if (base->party.on) {
427             for(uint8_t i = 0; i < MAX_COLUMNS; i++)
428                 sendCommand(MAX7219_REG_DIGIT0 + i, 0x00);
429
430             base->party.on = 0;
431         } else {
432             for (uint8_t j = 1; j <= MAX_COLUMNS; j++) {
433
434                 digitalWrite(SS, LOW);
435                 for(int8_t k = mLetterCount - 1; k >= 0; k--) {
436                     SPI.transfer16(j << 8 | (random(256) & 0x00FF));
437                 }
438                 digitalWrite(SS, HIGH);
439             }
440             base->party.on = 1;
441         }
442     }

```

```

443
444 static void countWithModule1(uint8_t &value, uint8_t maxValue, bool ascnd)
445 {
446     if (ascnd) {
447         if (value < maxValue - 1) {
448             value++;
449         } else {
450             value = 0;
451         }
452     } else {
453         if (value > 0) {
454             value--;
455         } else {
456             value = maxValue - 1;
457         }
458     }
459 }
460
461 static void countWithModule2(uint8_t &seconds, uint8_t maxSeconds, uint8_t &
462     ↪ minutes, uint8_t maxMinutes, bool ascnd)
463 {
464     if (ascnd) {
465         if (seconds < maxSeconds - 1) {
466             seconds++;
467         } else {
468             seconds = 0;
469             if (minutes < maxMinutes - 1)
470                 minutes++;
471             else
472                 minutes = 0;
473     } else {
474         if (seconds > 0) {
475             seconds--;
476         } else {
477             seconds = maxSeconds - 1;
478             if (minutes > 0)
479                 minutes--;
480             else
481                 minutes = maxMinutes - 1;
482     }
483 }
484 }
485
486 int strcpy_s(char *dest, unsigned int dmax, const char *src)
487 {
488     char *dst = dest;
489
490     if (dmax)
491         dmax--;
492     else
493         return 1;
494
495     while (*src && dmax) {
496         *(dest++) = *(src++);
497         dmax--;
498     }

```

```
499
500     *dest = '\0';
501
502     if (*src) {
503         reinterpret_cast<char*>(dst)[0] = '\0';
504         return 1;
505     }
506
507     return 0;
508 }
```