Proyecto

Gómez Cruz Erick Iván

6 de diciembre de 2023

1. Objetivo

- Desarrollar un sistema de cómputo distribuido usando un RP2040 y un ESP32 que cumpla con un propósito específico y solucione una problemática.

2. Introducción

El acuarismo es una actividad recreativa y estética donde se busca crear paisajes y a su vez mantener el cuidado adecuado de los peces y plantas que se piensan obtener.

Por otro lado, en México el acuarismo es una actividad productiva que ha crecido notablemente en los últimos 15 años. Ello se refleja en los aproximadamente mil 650 millones de pesos que genera la comercialización de más de 43 millones de peces de ornato de agua dulce, y en los cerca de 41 mil empleos que existen entre las 250 granjas de producción, los 5 mil establecimientos comerciales registrados y los 15 mil puntos de venta informales. El 46 por ciento de los peces que se comercializan en el país son importados y el resto se produce en las granjas distribuidas en todo el territorio, pero concentradas fundamentalmente en el estado de Morelos (Julia Carabias Lilo, Reforma, 2009).

Debido al crecimiento del acuarismo, se busca un sistema que automatice las necesidades de los peces y/o las plantas que se tengan en un acuario. Entre los parámetros que se desean automatizar son:

1. Cambios de Agua.
2. Control de iluminación.
3. Control de Temperatura.
4. Control del Filtro.

Tales actividades son las más indispensables para el acuarismo.

Una forma de automatizar es usando algún microcontrolador comercial y relativamente accesible, de forma más sencilla usarlo en una tarjeta de desarrollo. Dos opciones de microprocesadores es el RP2040 y el ESP32.

El ESP32 es un SOC que ofrece una amplia variedad de especificaciones que lo hacen de amplio uso en ciertas aplicaciones, como en IoT. Dentro de sus características, tenemos:

- Conectividad vía Wifi y Bluetooth;
- Potencial computacional vía la CPU y la memoria, además de soporte para baja potencia;
- Soporte para diversos protocolos de comunicación, como SPI, I2C, I2S, etc.;
- I/Os, y
- RTC

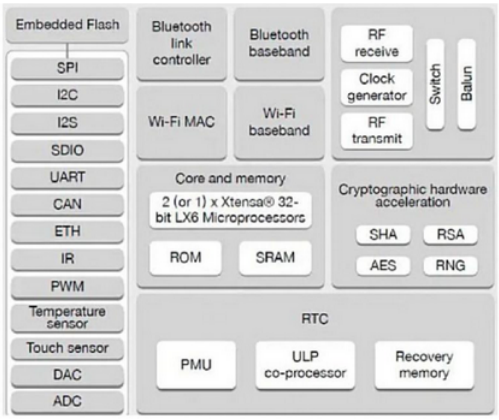

Figura 1: Diagrama a bloques ESP32 [1].

El diagrama de bloques de la figura 1nos muestra con qué cuenta el ESP32. Se dice que con aproximadamente 20 componentes externos es posible aprovechar todas sus funcionalidades [1], todo esto bajo el beneficio de ocupar un área muy pequeña dentro del circuito impreso.

La forma más sencilla de encontrar el ESP32 es como un módulo que ofrece el shield metálico y la posibilidad de soldarlo.

El RP2040 fue el primer microcontrolador diseñado por Raspberry Pi Ltd en 2021. Este se puede programar en ensamblador, C, C++, MicroPython, entre otros. Algunas de sus características son:

- Dos Cortex-M0+ a 133 MHz;
- 264 KB de SRAM;
- Controlador DMA;
- 30 pines de propósito general (GPIO), y
- Periféricos como: UART, SPI, I2C, PWM, etc.

3. Lista de Materiales

- Tarjeta de desarrollo Dual MCU;
- Conectores varios (jumpers en cualquiera de sus variantes, conexiones a la acometida eléctrica, etc.);
- Puente H;
- Fuente de 5 V;
- Filtro de AC;
- Calentador de AC;
- Tira de leds neopixel;
- Servomotor;
- Sensor de nivel;
- Bombas de DC
- Ventilador de DC;
- Sensor de temperatura sumergible Pt100 de dos terminales;

- MOC3011, y
- Resistencias varias a $\frac{1}{4}[W]$.

4. Dispositivos, su funcionamiento y cuidados

Si bien en la sección 3 se enlistan todos los materiales desarrollados en el proyecto, aquí solo se especificará de aquellos usado por mi.

- Tarjeta de desarrollo Dual MCU: Es un módulo que combina el microcontrolador Raspberry Pi RP2040 y el chip Espressif ESP32 en una sola placa, la cual es compatible con el entorno de desarrollo integrado (IDE) Micropython, como Thonny, y también con el entorno de desarrollo Arduino IDE.
- Puente H: Se trata de un circuito usado para hacer girar y controlar motores de DC. El circuito se muestra en la figura 2 y con la cual podemos mencionar diferentes elementos:
  - Contamos con dos borneras (Vcc y GND) por donde se conectará el voltaje que se desea llegue a las cargas, puede ir de 5 V hasta 35 V;
  - Las entradas E se encargan de indicar el sentido de giro de la carga, mientras que en las entradas C se suele conectar una señal PWM para regular la potencia promedio que llega a la carga. En nuestro caso no usaremos éstas entradas. Para que funcionen, el voltaje que debe suministrarse a las entradas E debe ser 5 V.
  - Las entradas E_1, E_2 y C_1 sirven para controlar a la carga 1 mientras que entradas E_3, E_4 y C_2 a la carga dos.

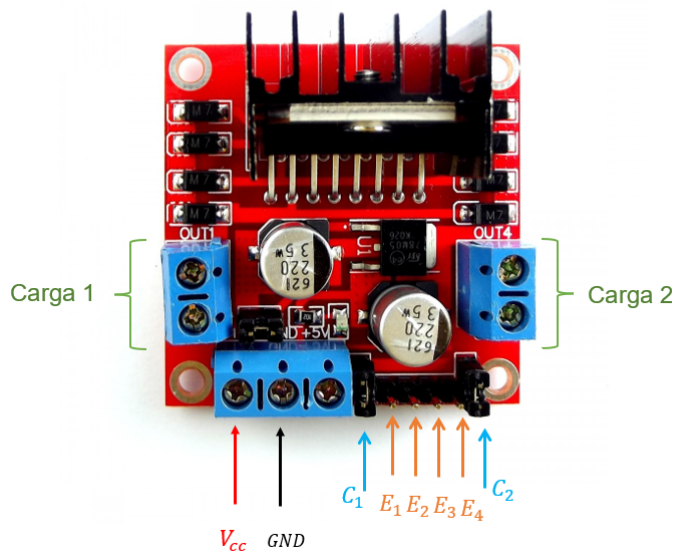

Figura 2: Diagrama del puente H [3].

- Bombas de DC: Una bomba de agua es una máquina rotatoria que mueve un elemento conocido como impulsor, que es una rueda con aspas, y que se encarga de generar energía centrífuga que desplaza el agua de un lugar a otro.

  Para su operación basta con sumergirla y alimentarla con el voltaje que especifique el fabricante, el que usualmente es de 5 V a 12 V. Se sugiere que no se opere sin agua ya que se podría llegar a quemar.

  El tipo de bomba empleado se muestra en la figura 3

Figura 3: Ejemplo de la bomba de DC empleada.

- Ventilador DC: consiste en un motor DC. Transforma energía eléctrica en energía mecánica al hager girar el eje del motor. A este eje se le acoplan las aspas del ventilador. Su principio de funcionamiento se basa en una espira de material conductor inmerso en un campo magnético. Un ejemplo de ventilador de DC se muestra en la figura 4.

Figura 4: Ejemplo de un ventilador de DC empleado.

- Sensor de nivel: Se encarga de detectar si hay líquido. Trabaja con una alimentación entre 3 y 5 V de DC a una corriente menor a 20 mA. Tiene una resolución de 10 bits. Este sensor se observa en la figura 5.

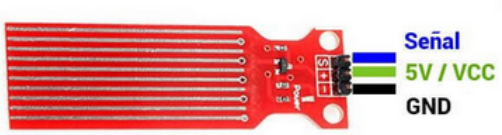

Figura 5: Sensor de nivel.

5. Advertencias

Se debe tener cuidado con aquellas conexiones con la acometida eléctricas ya que se podría recibir una descarga, en menor medida aquellas que incluyen la fuente de 5 V. Se recomienda que cualquier cambio que se haga sobre el circuito se haga sin las fuentes conectadas. Se recomienda que se manipule el agua con el cuidado necesario para no mojar y dañar los componentes electrónicos. En caso de mojar alguno dispositivo no encender el sistema.

6. Configuración: Dual MCU

Para trabajar con la tarjeta de desarrollo necesitamos tener la capacidad de correr MicroPython en ambos núcleos, el proceso seguido para cada uno se muestra a continuación:

- RP2040:

  Para ello necesitamos ingresar al siguiente enlace y descargar la versión más resiente. Debe ser un archivo con extensión uf2. Una vez realizado esto deberemos cargar el archivo al RP2040, para ello debemos presionar el botón Boot, como se muestra en la figura 6, y sin soltarlo conectarlo al ordenador, podemos soltarlo después de un par de segundos. Si esto se realizó de forma exitosa se detectará un dispositivo de almacenamiento masivo, con lo cual solo habrá que mover el archivo descargado dentro del directorio correspondiente al RP2040.

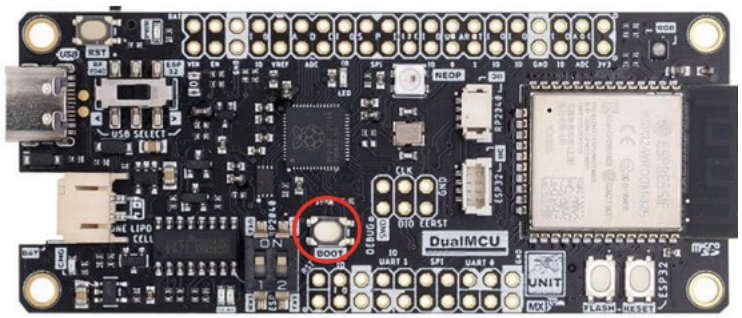

Figura 6: Modo boot para el RP2040 en Dual MCU

- ESP32 Aquí dependerá del sistema operativo en el cual se esté trabajando:

  - Windows:

    Partimos instalando la última versión de Python del sitio oficial, durante la instalación, en la primera ventana debemos seleccionar la casilla **Add Python to PATH** y continuar de forma normal.

    Hecho esto, abrimos la terminal de Windows y colocamos el siguiente comando.

```
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.
c:\Users\algo> pip install esp
```

    una vez terminado esto, debemos borrar lo que se tenga en la flash del ESP32 por defecto, esto lo hacemos con el siguiente comando, notése que en com# se debe colocar el puerto COM que está asociado a la placa de desarrollo. Revise el administrados de dispositivos para esto.

```
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.
c:\Users\algo> esptool --chip esp32 --port com# erase_flash
```

    Acabado el proceso, descargaremos la versión más reciente del firmware de MicroPython de su página oficial. Hecho esto, el archivo descargado lo pasamos a la carpeta donde se encuentra por defecto la terminal; ahora colocamos el siguiente comando para flashear MicroPython en el ESP32

```
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.
c:\Users\algo>esptool --chip esp32 --port com# --baud 460800 write_flash -z 0x1000
    ESP32_GENERIC-20230426-v1.20.0.bin
```

    Un aspecto a considerar es que se debe colocar el nombre del archivo .bin que se descargó, en mi caso el que viene en el ejemplo de arriba.

    Finalmente, solo necesitamos descargar Thonny de su sitio oficial. Llegado a este punto, tenemos todo para comenzar a trabajar.

  - Ubuntu:

    Abrimos una terminal y colocamos lo siguientes comandos

```
$ sudo apt install python3 python3-pin
$ pip install -U pip esptool
```

y podemos verificar la instalación mediante

```
$ python3 -m esptool
```

Ahora, descargamos MicroPython de si página oficial
Ahora, borraremos la flash del núcleo mediante el siguiente comando

```
$ esptool.py --chip esp32 --port /dev/ttyUSB0 erase_flash
```

y ahora cargamos el archivo de firmware mediante

```
$ esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0
    x1000 ESP32_GENERIC-20230426-v1.20.0.bin
```

Recuerde que el nombre del archivo con terminación bin puede variar dependiendo la versión, por lo cual es importante asegurarse que sea el mismo que se descargó.
Hecho esto solo bastará con usar Thonny para poder cargar los programas. Desde terminal basta con ingresar el comando

```
$ sudo apt install Thony
```

7. Desarrollo de software

Para el desarrollo de software tenemos tres códigos: la página web, el archivo para el ESP32 y el archivo para el RP2040.

7.1. index

El index es el nombre del archivo donde se almacena la página web. Esta se desarrolló en html y tiene varios apartados a considerar. Primero, se definen las características del cuerpo y del contenedor, en ambos se especifica su color de fondo, margenes y posición. Posteriormente, se definieron los botones, uno para indicar que se hará una modifición (para el color e intensidad de los leds) y otros para el encendido y apagado de algunos dispositivos (como el ventilador, calentador, etc.).

Habiendo definido esto, en el contenedor se especificará lo que habrá en cada línea, que será un texto indicando el estado de cada elemento y la opción de encenderlo o apagarlo (si corresponde) o bien se incluye el cuadro de texto para modificar un parámetro. A cada línea se le agreda un identificador para, por medio de funciones de java sustituir valores allí.

Para las funciones de java, podemos definir tres tipos de funciones: aquellas que capturan un valor y lo envían, aquellas que dependiendo del botón presionado enviarán uno u otro valor y aquellas que solo pidan la actualización de datos.

- Para las primeras, tenemos un código de ejemplo en 14, donde se muestra cómo se recupera el dato ingresado en la caja de texto. Usamos **document.getElementById()** para obtener en forma de cadena lo que tiene la caja de texto y lo almacenamos en la variable **valor**. Posteriormente en la variable **aux** almacenamos el resultado de convertir a entero **valor**. Con ello, enviamos a **aux** para que sea procesado en el archivo del ESP32 y posteriormente este nos regrese un resultado, el cual se sustituirá en el respectivo ID. Esta función opera junto al párrafo en html mostrado en 5. Este formato se aplica en la función del brillo y del color para los neopixeles.

```
function brightness(on){
  var valor = document.getElementById('duty')
  let aux = parseInt(valor.value,10)
  let req = new XMLHttpRequest();
  req.open('GET', 'brightness?on=' + aux);
  req.onload = function() {
  if (req.status == 200)
    document.getElementById("result").innerHTML = "Brightness defined
    at " + valor.value + "%."
  else
    console.log("Error: " + req.status);
```

```
  }
  req.send();
}
```

Listing 1: Función que guarda un valor ingresado

```
<p>
  <p>Brightness at <input id="duty" type="text" placeholder="[0 -
   100]" size= "6" maxlength="3" pattern="{0-9}"> %.</p>
  <button class="btnCalc"onclick="brightness();">Modify Brightness</
   button>
</p>
```

Listing 2: Párrafo que pide ingresar un valor

- Para el segundo, tenemos como código de ejemplo el mostrado en 12. Observamos que el proceso es similar en cuanto a mandar el valor **on** hacia el programa del ESP32, en este caso ese valor solo es 0 o 1 dependiendo del estado en que se necesite el actuador en cuestión, mismo que se reflejará posteriormente en el respectivo ID, en este caso en "led". Esta función va acompañada de un párrafo como el que se muestra en 6. Este formato de párrafo y/o botones se aplica en las funciones para controlar el encendido y apagado de los leds, del filtro, de la bomba de salida y la subida y bajada de los leds.

```
function updateLed(on){
  let req = new XMLHttpRequest();
  req.open('GET', 'led?on=' + on);
  req.onload = function() {
  if (req.status == 200)
      document.getElementById("led").innerHTML = req.responseText;
    else
      console.log("Error: " + req.status);
  }
  req.send();
}
```

Listing 3: Función que define el estado de un periférico

```
<p>
  Leds  <span id="led"><!--led--></span>.
  <button class="btnOn" onclick="updateLed(1);">On</button>
  <button class="btnOff" onclick="updateLed(0);">Off</button>
</p>
```

Listing 4: Párrafo donde se puede definir el estado de un periférico

- Para el último caso donde sólo se pide el estado o un valor leído para desplegarlo tomamos como ejemplo el código mostrado en 12, donde simplemente enviamos el çomando"lvl para indicarle al ESP32 qué acción queremos que realice y su respuesta la sustituiremos en el ID "lvl". El párrafo del que se acompaña se muestra en 2. Otro aspecto a considerar es que este tipo de peticiones las realizamos de forma periódica, la forma de declarar esto se muestra en 8, donde además mostramos todas las declaraciones para cada función de este tipo. Este formato de solo peticiones se hace para la temperatura, el nivel, el estado de la bomba de entrada, del calentador y del ventilador.

```
function updatelvl(){
  let req = new XMLHttpRequest();
  req.open('GET', "lvl");
  req.onload = function() {
    if (req.status == 200)
      document.getElementById("lvl").innerHTML = req.responseText;
    else
```

```
        console.log("Error: " + req.status);
    }
    req.send();
}
```

Listing 5: Función que actualiza un dato

```
<p><span id="lvl"><!--lvl--></span>.</p>
```

Listing 6: Sustitución de resultados por ID

```
window.onload = (event) => {
    window.setInterval(updateTemperature, 1000);
  window.setInterval(updatelvl,1100);
  window.setInterval(updateIP,1200);
  window.setInterval(updateHeat,1300);
  window.setInterval(updateFan,1400);
}
```

Listing 7: Función periódicas

7.2. ESP32.py

Aquí, destacamos algunas funciones en las cuales realicé modificaciones. La primera de ellas, mostrada en 46,se encarga de evaluar que petición realiza el index. Dependiendo de con qué empiece la petición realizará una acción.

```
def serveWeb():
  cnn, addr = s_web.accept()
  print(f'Client connected from { str(addr) }')
  request = cnn.recv(1024)
  sreq = request.decode('utf-8')
  print(f'Request: { sreq[:indexOf(sreq,'\n')] }')

  if not sreq.startswith('GET '):
    cnn.send('HTTP400: Bad request\n')
    cnn.close()
    return

  sreq = sreq[4:indexOf(sreq,' ', 5)]

  if sreq.startswith('/?') or sreq.startswith('/index.htm'):
    payload = webpage()
  elif sreq.startswith('/led'):
    payload = serveLed(sreq)
  elif sreq.startswith('/filt'):
    payload = serveFilt(sreq)
  elif sreq.startswith('/fan'):
    payload = serveFan(sreq)
  elif sreq.startswith('/heat'):
    payload = serveHeat(sreq)
  elif sreq.startswith('/inP'):
    payload = serveInP(sreq)
  elif sreq.startswith('/outP'):
    payload = serveOutP(sreq)
  elif sreq.startswith('/temp'):
    payload = serveTemp(sreq)
  elif sreq.startswith('/brightness'):
    payload = serveBrightness(sreq)
  elif sreq.startswith('/color'):
    payload = serveColor(sreq)
  elif sreq.startswith('/lvl'):
    payload = serveLvl(sreq)
  elif sreq.startswith('/lamp'):
```

```
    payload = serveLamp(sreq)
  elif sreq == '/':
    payload = webpage()
  else:
    payload = 'HTTP404: Not found\n'
  print(f'Response: {payload.strip()[:200]}')
  cnn.send(payload)
  cnn.close()
```

Listing 8: Función que gestiona la petición

Ahora, de las funciones que se encargan de cada petición podemos agrupar algunas entre si y solo mostrar un ejemplo.

La primera de ellas se muestra en (11), esta es la función para modificar el color y es, en esencia, la misma para **serveBrightness**, **serveOutP**, **serveFilt**, **serveLamp** y **serveLed**. De ella obtenemos la cadena enviada por el index y evalúa si se encuentra «on» en ella. En caso afirmativo realiza la petición al RP2040 y espera su respuesta enviando como parámetro al propio «on»; en caso negativo realiza la petición sin el parámetro. El valor que regresa 1 ó 0, por lo cual en las funciones que enciende u apagan un dispositivo al final evaluamos esto para sustituirlo por un on u off respectivamente. Esto se mantiene igual entre las funciones excepto por aquella que cambia la altura de los leds, ahí se sustituyó por up ó down.

```
def serveBrightness(sreq):
  up = fetchUriParams(sreq)
  if 'on' in up:
    status = rpc('brightness', up['on'])
  else:
    status = rpc('brightness')
  if status is None:
    return 'HTTP500: Internal server error\n'
    # return 'HTTP400: Bad request\n'
  return status
```

Listing 9: Función que envía parámetros al RP2040

El último grupo engloba las funciones que solo realizan la petición sin argumento. De aquí podemos desprender dos casos.

El primero de ellos se muestra en 6, se trata de la función que atiende la petición de nivel de agua y **serveHeat** y **serveInP**. Observamos que hacemos la petición, esperamos la respuesta y dependiendo de la respuesta se regresa una cadena al index. Aquí el retorno del valor es directo dado que en el programa RP2040.py se regresa una cadena, como veremos más adelante en 7.3.

```
def serveLvl(sreq):
  lvl = rpc('lvl')
  if lvl is None:
    return ''
  return lvl
```

Listing 10: Función que envía petición sin parámetros al RP2040

El segundo caso es cuando no recibimos una cadena sino un número. Para ello simplemente mediante format regresamos en forma de string el valor, en este caso se trata de un flotante y yo especifiqué que fueran 3 decimales. Este se muestra en 6. Este formato solo se usa en esta función.

```
def serveTemp(sreq):
  temp = rpc('temp')
  if temp is None:
    return ''
  return f'{float(temp):.3f}'
```

Listing 11: Función que envía petición sin parámetros al RP2040

7.3. RP2040

En este archivo, se definen los pines con los que se trabajará en la función setup() como sigue

```
neoD = NeoPixel(Pin(22),96)
filt = Pin(23, Pin.OUT)
fan = Pin(21, Pin.OUT)
heater= Pin(19, Pin.OUT)
inPump = Pin(15, Pin.OUT)
outP = Pin(11, Pin.OUT)
rpcQ = ucollections.deque((), 10)
```

En esta misma función se inicializa el uart, el adc, se fuerza el apagado de los leds y se hacen definiciones para el motor a pasos. La explicación del porqué corresponde a quienes lo realizaron.

Le función medular de este programa se muestra en (27). Se trata de la función serveRPC() que se encarga de atender la petición dependiendo de la cadena que reciba.

```
def serveRPC(s):
  i, f, p = splitRPC(s)
  if f == 'temp':
    serveTemp(i, f, p)
  elif f == 'led':
    serveLed(i, f, p)
  elif f == 'filt':
    serveFilt(i,f,p)
  elif f == 'fan':
    serveFan(i,f,p)
  elif f == 'heat':
    serveHeat(i,f,p)
  elif f == 'inP':
    serveInP(i,f,p)
  elif f == 'outP':
    serveOutP(i,f,p)
  elif f=='brightness':
    serveBrightness(i,f,p)
  elif f=='color':
    serveColor(i,f,p)
  elif f == 'lvl':
    serveLvl(i,f,p)
  elif f == 'lamp':
    serveLamp(i,f,p)
  else:
    serveNot(s)
```

Listing 12: Función que evalúa las peticiones en el RP2040

Aquí, podemos agrupar algunas funciones que, en esencia, operan de la misma forma. El primer grupo engloba a la función **serveHeat()**, **serveInP()** y **serveFan()** que se encargan simplemente determinar el estado del puerto asociado y regresar su valor. Una de esas funciones se muestra en (7). Observamos que guardamos en una variable el valor del pin mediante el parámetro **value()** (es importante recalcar que no lleva argumento) y con un if separamos los casos (1 de 0) y dependiendo de cual cumpla regresa un «on» u «off» respectivamente.

```
def serveHeat(i, f, p):
  heatStat = heater.value()
  if bool(heatStat):
    returnRPC(i, f, 'on')
  else:
    returnRPC(i, f, 'off')
```

Listing 13: Función que determina el estado de un pin y lo retorna

Otro grupo engloba a las funciones **serveColor()** y **serveBrightness**. Un ejemplo se muestra en (23). Aquí evaluaremos que el argumento **p** tenga una longitud mayor a 0, en caso afirmativo lo convertimos a entero y en otro caso se le asigna al estado un None (esto con el fin de primero obtener el entero que se ingresó el cuadro de texto del index).

Ahora, mediante el bloqueo del hilo y sí el estado no es None evaluamos el número que se recibió. Para la función del color el rango admitido es de 0 a 360 y en caso de ingresar un número mayor se asigna el 360 y de ser menor el 0; para la función del brillo el rango admitido es de 0 a 100, si se ingresa un número mayor al rango se define en 100 y si es menor un 0. Posteriormente se usa la función **hsv2rgb()** para con base en los argumentos de color y brillo se obtenga el respectivo valor en rgb y se escriba este para cada uno de los leds del arreglo. Finalmente regresamos el valor del color o del brillo.

```
def serveBrightness(i, f, p):
  global svBrightnessOn, brightness, color
  try:
    if len(p) > 0:
      brightnessStat = int(p[0])
    else:
      brightnessStat = None
  except:
    brightnessStat = None
  with lkBrightnessOn:
    if brightnessStat is not None:
      if brightnessStat>100: brightnessStat=100
      elif brightnessStat<0: brightnessStat=0
      brightness = brightnessStat/100
      NL = 0
      while NL < 96:
        neoD[NL] = hsv2rgb(color, 1, brightness) # set the first pixel to white
        neoD.write()              # write data to all pixels
        NL+=1
    else:
      brightnessStat = svBrightnessOn
  returnRPC(i, f, brightnessStat)
```

Listing 14: Función que cambia el brillo de los leds

Un punto a recalcar es que en ambas funciones antes mencionadas se utiliza una variable global para almacenar el color y brillo, se esta forma se mantiene uno en caso de cambiarse otro.

Un grupo más de funciones engloba aquellas donde dependiendo del botón presionado en el index (y del valor asociado que llega hasta el RP2040) enciende o apaga un pin. Las funciones comprendidas son **serveOutP** y **serveFilt**. Una de ellas se muestra en (20), aquí se evalúa que haya un valor en el parámetro **p** y mediante un bloqueo de hilo evaluamos si se trata de un true o un false (un 1 o 0 respectivamente) y encendemos o pagamos el puerto respectivamente. Al final regresamos ese estado al ESP32.

```
def serveFilt(i, f, p):
  global svFiltOn
  try:
    if len(p) > 0:
      FiltStat = int(p[0])
    else:
      FiltStat = None
  except:
    FiltStat = None
  with lkFiltOn:
    if FiltStat is not None:
      svFiltOn = bool(FiltStat)
      if svFiltOn:
        filt.on()
      else:
        filt.off()
    else:
      FiltStat = svFiltOn
  returnRPC(i, f, int(FiltStat))
```

Listing 15: Función que enciende o apaga puertos

Dos funciones que recaen en este mismo grupo con algunos extras son **serveLed** y **serveLamp**.

- En la primera se agrega el respectivo ciclo while para encender o apagar cada uno de los leds del arreglo, el código se muestra en

```
def serveLed(i, f, p):
  global svLedOn
  try:
    if len(p) > 0:
      ledStat = int(p[0])
    else:
      ledStat = None
  except:
    ledStat = None
  with lkLedOn:
    if ledStat is not None:
      svLedOn = bool(ledStat)
      if svLedOn:
        NL = 0
        while NL < 96:
          neoD[NL] = hsv2rgb(color, 1, brightness) # set the first pixel to
    white
          neoD.write()              # write data to all pixels
          NL+=1
      else:
        NL = 0
        while NL < 96:
          neoD[NL] = hsv2rgb(color, 1, 0) # set the first pixel to white
          neoD.write()              # write data to all pixels
          NL+=1
    else:
      ledStat = svLedOn
  returnRPC(i, f, int(ledStat))
```

Listing 16: Función que enciende o apaga los leds

- En la segunda se agrega una variable a modo de bandera para determinar si los leds se encuentra arriba o abajo, dependiendo de esto y de la instrucción que llegue se enciende o se apaga el motor. La explicación del funcionamiento de estas funciones será hecho por quien corresponda. El código se muestra en

```
def serveLed(i, f, p):
  global svLedOn
  def serveLamp(i, f, p):
  global svLampOn, realizar
  try:
    if len(p) > 0:
      LampStat = int(p[0])
    else:
      LampStat = None
  except:
    LampStat = None
  with lkLampOn:
    if LampStat is not None:
      svLampOn = bool(LampStat)
      if svLampOn:
        if realizar:
          onmotor()
          realizar = False
      else:
        if realizar ==False:
          offmotor()
          realizar= True
    else:
      LampStat = svLampOn
  returnRPC(i, f, LampStat)
```

Listing 17: Función que sube o baja los leds

Finalmente tenemos las funciones que usando los ADC miden tanto el nivel como la temperatura.

- La función que mide el nivel se muestra en (23), aquí medimos el nivel y almacenamos el resultado en la variable **lvl**. Definimos un límite sobre el cual se harán diferentes acciones: por encima de ese se apaga la bomba de entrada y envía el mensaje de que el tanque está a más de la mitad y por debajo de eso se enciende la bomba de entrada y se manda el mensaje de que el tanque está por debajo de la mitad. Esto ocurre de esta manera siempre y cuando la bomba de salina no esté activada.

```
def readLvl():
  lvl=adc2.read_u16()
  print(f'Level: {lvl}')
  return lvl

def serveLvl(i,f,p):
  lvl = readLvl()
  print('Nivel: {}'.format(lvl))
  outpStat = outP.value()
  if 32000<lvl:
    state = "Tank over half full"
    if bool(outpStat):
      inPump.off()
    else:
      inPump.off()
  else:
    state = "Tank less than a half full"
    if bool(outpStat):
      inPump.off()
    else:
      inPump.on()
  returnRPC(i,f,state)
```

Listing 18: Funciones para medir el nivel

Para la función **serveTemp** usamos la función **readTemp** la cual usa 2 ADC para medir la temperatura con ayuda del sensor Pt100 y la configuración del puente de Wheatstone. Especificará más al respecto a quien le correspondió este apartado. El código de la función se muestra en ()

```
def readTemp():
  aux = 0
  Sum=0
  while(aux<1000):
    x1 = adc1.read_u16()*3.3/65535
    x3 = adc3.read_u16()*3.3 /65535
    Vab= x3-x1
    Sum += Vab
    aux+=1
  Sum = Sum/1000
  temp=Sum/0.00047
  if (temp>=17):
    heater.off()
    fan.on()
  elif (temp<17):
    heater.on()
    fan.off()
  return temp

def serveTemp(i, f, p):
  temp = readTemp()
  returnRPC(i, f, temp)
```

Listing 19: Funciones para medir la temperatura

8. Integración software-hardware

Las conexiones hechas para integrar todo el sistema se muestra en 7. Observamos los pines respectivos nombrados y conectados entre los puentes H, la dual MCU, los sensores, entre otros. Finalmente, el resultado se muestra en el siguiente video y los códigos se pueden acceder mediante el siguiente repositorio de Github.

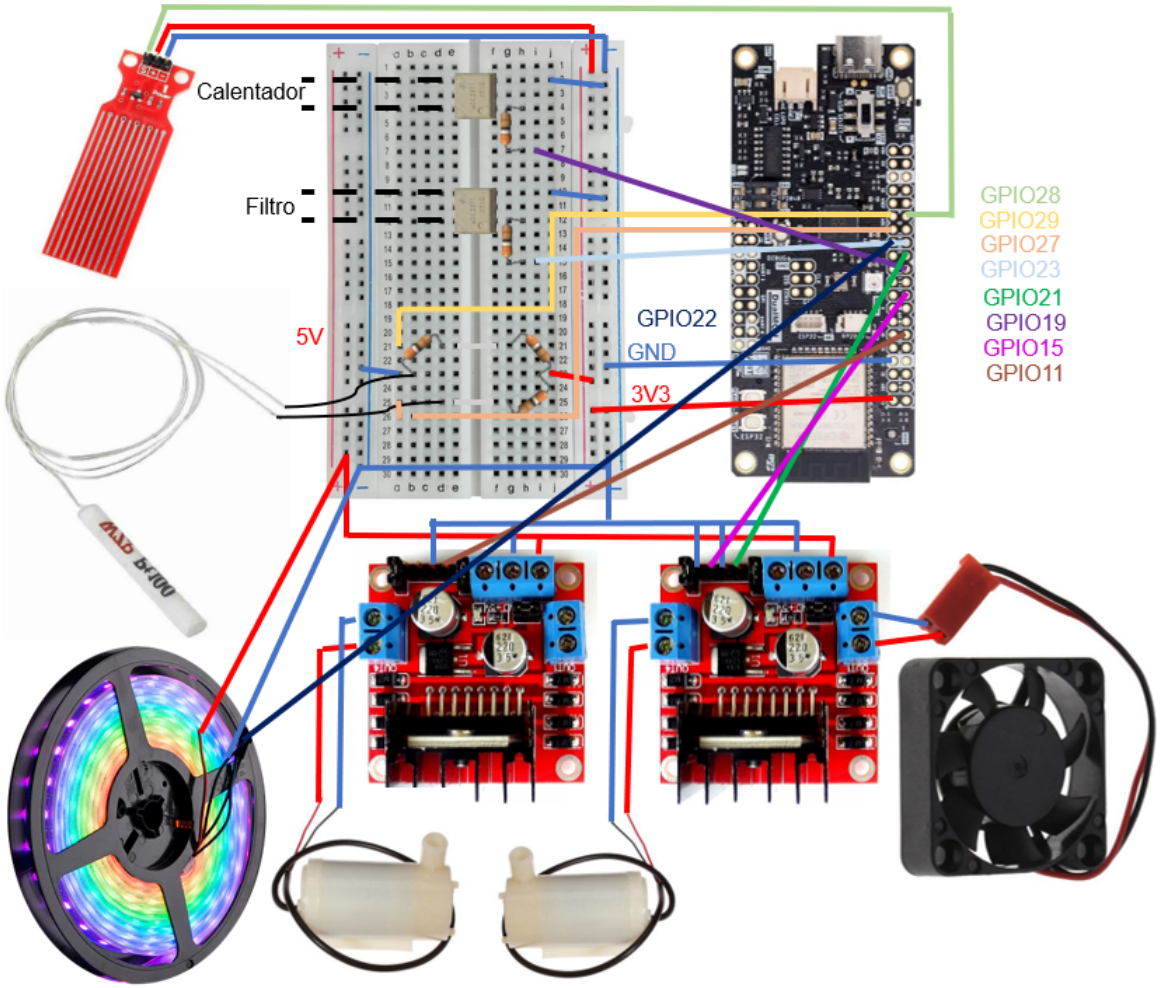

Figura 7: Diagrama de conexión entre hardware.

9. Conclusiones

Por medio del proyecto terminé de aplicar los conocimientos adquiridos a lo largo del curso. Logré unificar un sistema distribuido usando Micropython, lo que no solo me ayuda a entender el concepto y funcionamiento del computo distribuido sino que también fortalece mi conocimiento sobre Python y representa mi primer acercamiento a trabajar con el ESP32 y el RP2040, lo cual me ayuda a tener una visión más amplia sobre el diseño en tarjetas de desarrollo y la forma de programación (ya que antes solo lo había hecho a nivel registro). Por otro lado,la integración entre hardware y software me permite ver todas las posibilidades y oportunidades para desarrollar proyectos en un futuro.

Referencias

[1] Bertoleti, P. (2019). Proyectos com ESP32 y LoRa. Editora NCB.

[2] RP2040. (2023). En Wikipedia. https://en.wikipedia.org/w/index.php?title=RP2040&oldid=1182853225

[3] Driver Puente H L298N 2A. (s.f.). Recuperado 3 de diciembre de 2023, de https://naylampmechatronics.com/drivers/11-driver-puente-h-l298n.html

[4] Acuarismo y riesgos. (2009, julio 11). CeIBA. https://ceiba.org.mx/que-hacemos/blogs-opinion-editorial/julia-carabias-lillo/acuarismo-y-riesgos/

[5] UNIT DualMCU ESP32 + RP2040. (s. f.). UNIT Electronics. Recuperado 6 de diciembre de 2023, de https://uelectronics.com/producto/unit-dualmcu-esp32-rp2040-tarjeta-de-desarrollo/