Table of Contents

Preparacion entorno de programacion del ESP32

esp32, ide, arduino

Preparación del entorno

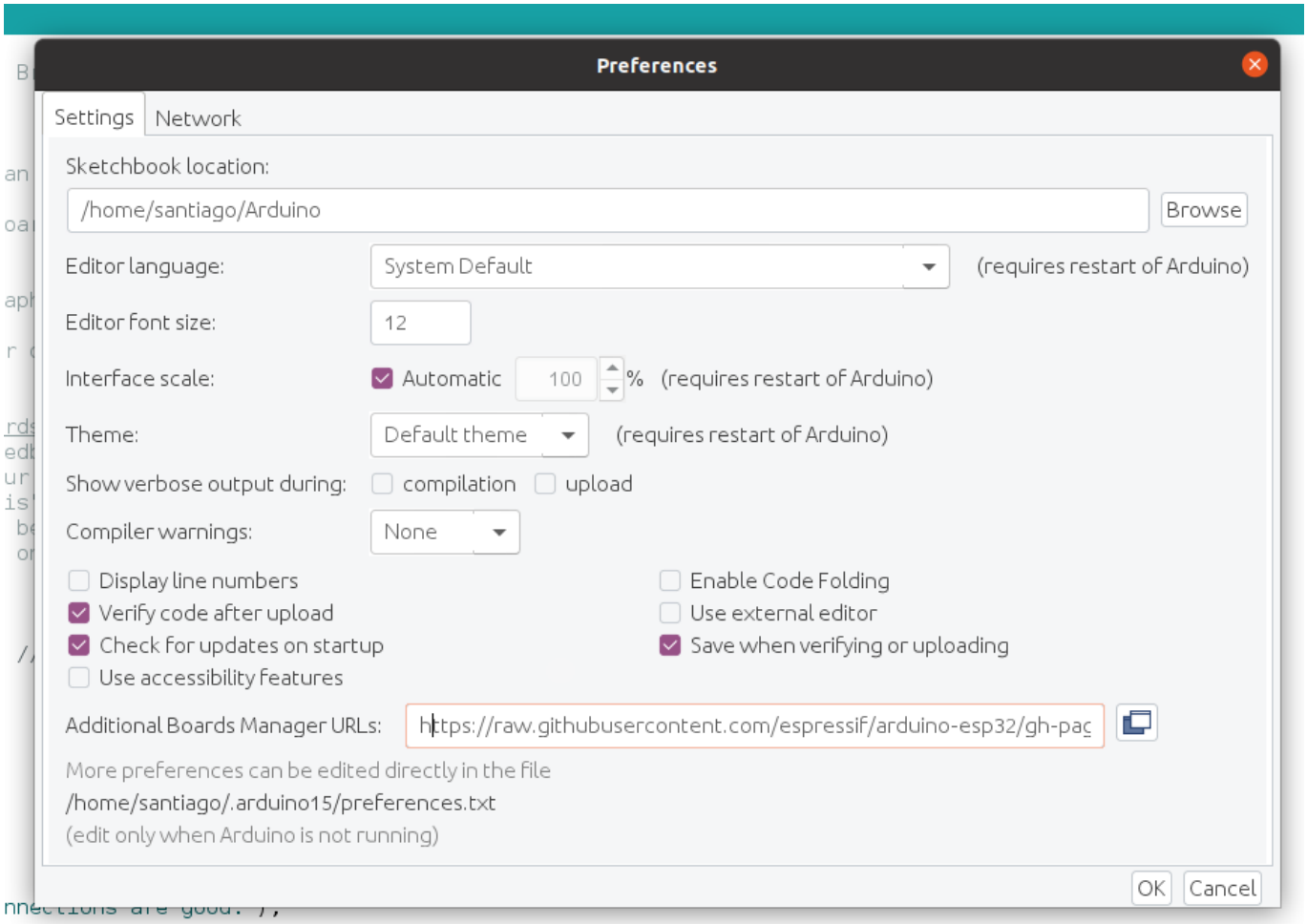
La siguiente descripción de una instalación ha sido llevada a cabo en un sistema operativo Linux, concretamente en la distribución Ubuntu 20.04. En el caso de Windows, quizá sean necesarios otros pasos.

Instalación IDE arduino

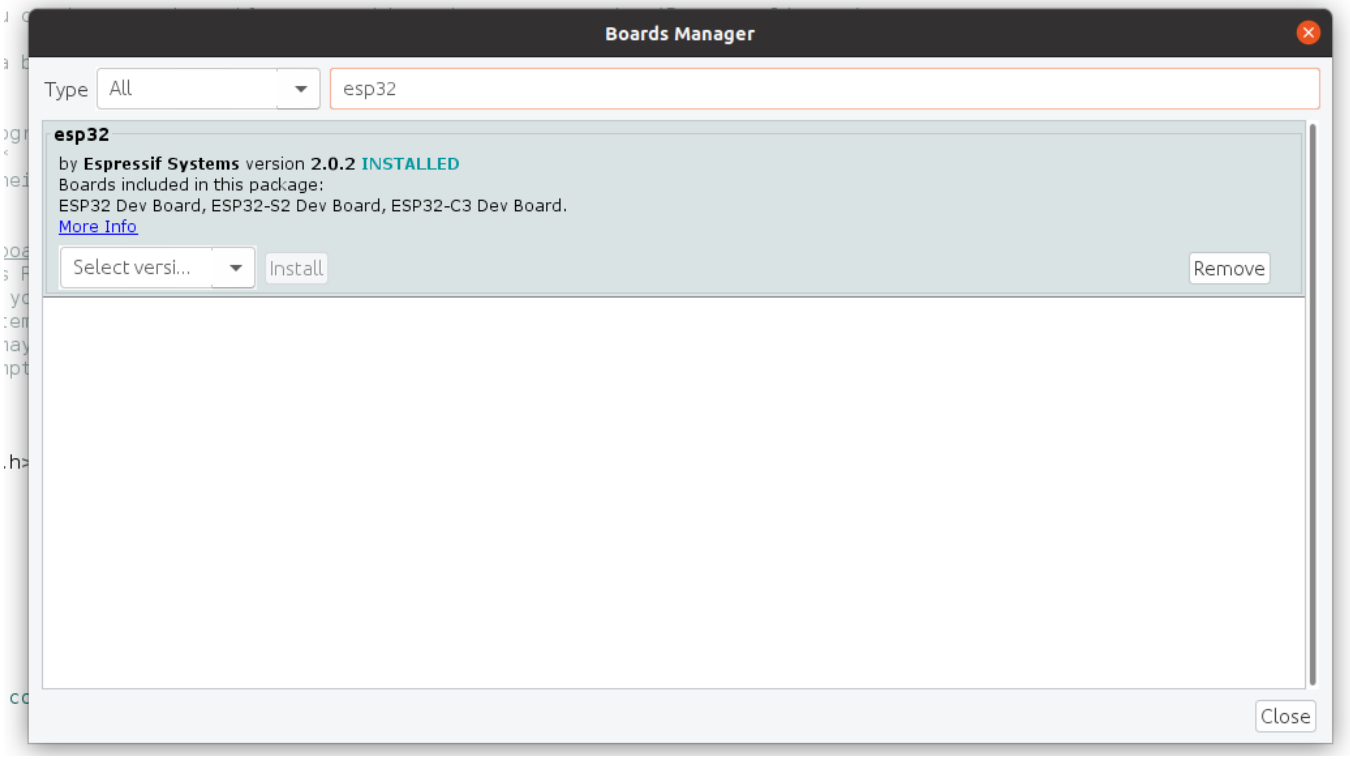
Una de las maneras de programar la placa ESP32 es usando el IDE de arduino. Para ello, acceder a https://docs.arduino.cc/software/ide-v1/tutorials/Linux y descargar la versión deseada. Una vez extraído el fichero descargado, ejecutar como super usuario el script install.sh.

Configuración para la programación del ESP32

Para poder usar el IDE con el ESP32, es necesario acceder a preferencias y en Gestor de URLs adicionales de tarjetas copias la siguiente URL
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json



Pulsamos el botón ok, en el apartado de herramientas, placas, seleccionamos el administrador de placas (board manager), buscamos por esp32 y lo instalamos. Una vez instalado, en el apartado de placas debe aparecer ESP32 Arduino y dentro de ese apartado seleccionamos nuestra placa en concreto (en el caso de la placa ESP32 ESP-WROOM-32 NodeMCU v2 la opcion ESP32 DEV Module ha funcionado correctamente).



Una vez esté seleccionado, para testear el funcionamiento es interesante probar un hola mundo. Para ello, copiamos este código y lo ejecutamos.

```
void setup() {
  Serial.begin(115200);
}

void loop() {
  Serial.println("Hello from DFRobot ESP-WROOM-32");
  delay(1000);
}
```

Resolución de errores

Si aparece un error de compilación indicando *ImportError: No module named serial* significa que hay un problema la librería pyserial de python. Si tras instalarla con
`Pip install pyserial`

el problema persiste, puede ser un problema respecto a la version de python y sus librerias. Una solucion es ejecutar el comando

`sudo apt install python-is-python3`

Una vez hecho esto, ya debería poderse compilar el programa sin problema. A la hora de subir el programa a la placa, puede aparecer el siguiente error.

`Permission denied on /dev/ttyXXX`

Para solucionarlo, hay que consultar la terminal y localizar el grupo al que pertenece el puerto con

```
ls -l /dev/ttyXXX
```

Obteniendo algo como lo siguiente

```
crw-rw---- 1 root dialout 188, 0 5 apr 23.01 ttyACM0
```

Para añadir nuestro usuario al grupo ejecutamos el comando

```
sudo usermod -a -G dialout <username>
```

Otra opcion es ejecutar el comando

```
sudo chmod a+rw /dev/ttyXXX
```

Con esto, ya se debería subir el programa de prueba a la placa ESP32. Para consultar la salida, abrir el monitor serie y poner los baudios al número indicado en el Serial.begin, en este caso 115200.

Detección del dispositivo conectado por bus I2C

Para comprobar si el ESP32 detecta el chip criptográfico (o cualquier otro elemento) conectado a través del bus I2C, el siguiente código es útil para obtener la dirección hexadecimal de los chips con los que se comunica.

```
#include <Wire.h>

void setup() {
  Wire.begin();
  Serial.begin(115200);
  Serial.println("\nI2C Scanner");
}

void loop() {
  byte error, address;
  int nDevices;
  Serial.println("Scanning...");
  nDevices = 0;
  for(address = 1; address < 127; address++ ) {
    Wire.beginTransmission(address);
    error = Wire.endTransmission();
    if (error == 0) {
      Serial.print("I2C device found at address 0x");
      if (address<16) {
        Serial.print("0");
      }
      Serial.println(address,HEX);
```

```
      nDevices++;
    }
    else if (error==4) {
      Serial.print("Unknow error at address 0x");
      if (address<16) {
        Serial.print("0");
      }
      Serial.println(address,HEX);
    }
  }
  if (nDevices == 0) {
    Serial.println("No I2C devices found\n");
  }
  else {
    Serial.println("done\n");
  }
  delay(5000);
}
```

La salida particular en nuestro caso es la siguiente:

```
Scanning...

I2C device found at address 0x60

done
```

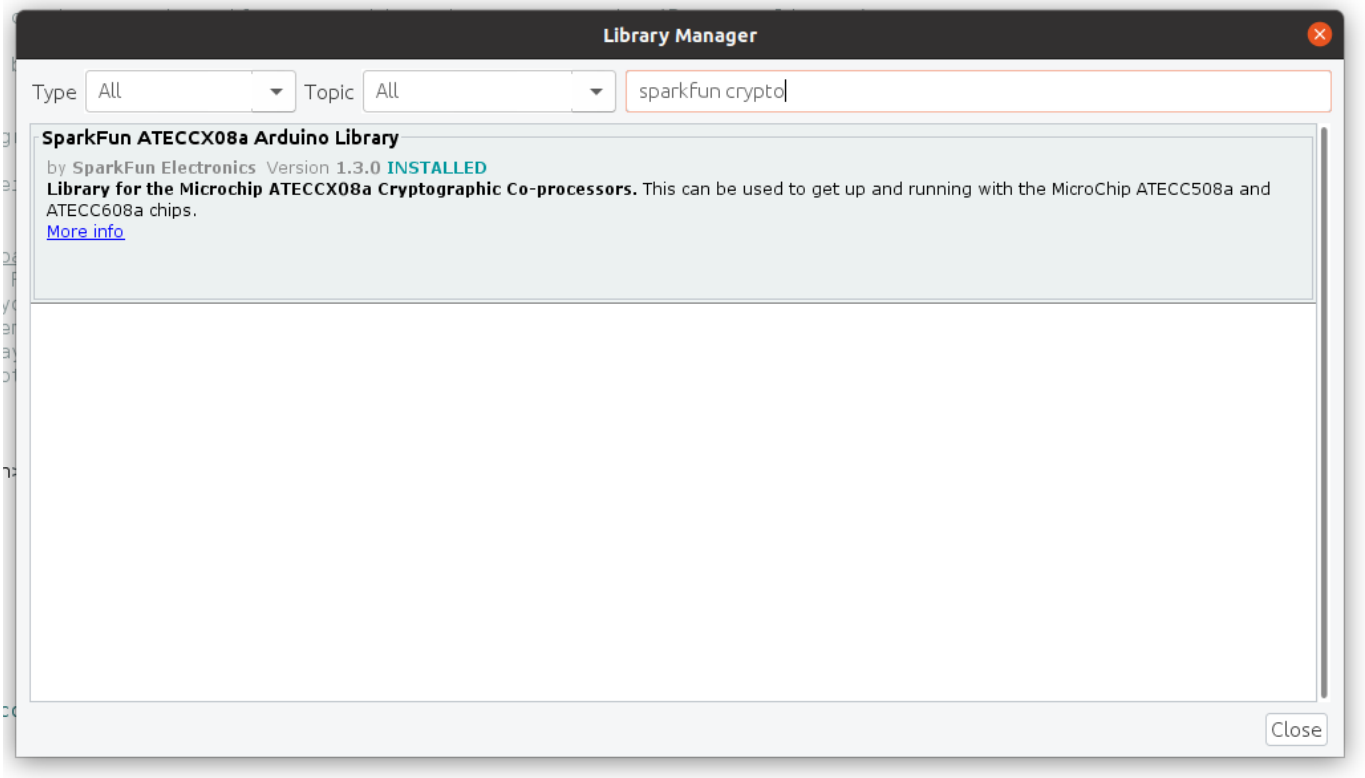
Por lo que se detecta el chip criptografico.

NOTA: Si al tratar de inyectar el código para la detección de dispositivos conectado al bus I2C se obtiene el error A fatal error occurred: Failed to connect to ESP32: Invalid head of packet puede significar que sea necesario pulsar el botón de reset antes de cargar el programa.

Uso de librería criptográfica con el chip ATECC508A

Una opción de firmar y verificar mensajes es haciendo uso de la librería criptográfica de Sparkfun.

Para usarla, a través del gestor de librerías, instalamos la relativa a Sparkfun Cryptographic, como podemos ver en la imagen.



Una vez está la librería correctamente instalada, se pueden usar los ejemplos para comprobar el funcionamiento del chip, yendo al apartado Ejemplos del IDE de Arduino.

Ejemplo 1 - Configuración del chip criptográfico

Antes de realizar cualquier operación con el chip, es necesario configurarlo para bloquear el acceso a la clave privada, entre otras cosas. Una vez ejecutado el programa de configuración, esta se queda bloquedada para siempre.

Al finalizar la ejecución se imprimirá un mensaje indicando la clave pública del chip, la clave privada nunca se va a poder consultar.

A continuación vemos un ejemplo de salida:

```
Configuration beginning.

Write Config:   Success!

Lock Config:    Success!

Key Creation:   Success!

Lock Data-OTP:  Success!

Lock Slot 0:    Success!
```

```
Configuration done.



Serial Number:  0123C5C7349171DDEE

Rev Number:     00005000

Config Zone:    Locked

Data/OTP Zone:  Locked

Data Slot 0:    Locked


This device's Public Key:


uint8_t publicKey[64] = {

0xB2, 0xAA, 0xE7, 0x84, 0x1D, 0x43, 0x5C, 0xE6, 0x49, 0xFD, 0x26, 0x3B,
0x8D, 0xC2, 0xF8, 0x2A,

0x20, 0x49, 0x9A, 0xFC, 0xAE, 0xFE, 0x25, 0x1C, 0x6A, 0x90, 0x26, 0xC6,
0x40, 0xC3, 0x4C, 0x5F,

0x3A, 0x98, 0xAA, 0xA4, 0x2B, 0xFE, 0x46, 0x40, 0x99, 0xB4, 0xC5, 0x26,
0x81, 0x94, 0x6B, 0x18,

0xDF, 0x3D, 0xE6, 0x18, 0x6D, 0x4C, 0x61, 0xE0, 0x1F, 0xD2, 0x4F, 0x73,
0x80, 0xB2, 0x2E, 0x68

};
```

Ejemplo 2 - Firmar un mensaje

En este ejemplo, dado un mensaje y usando la clave privada almacenada en el chip, se procederá a realizar la firma digital.

Cosas importantes que hay que saber sobre las firmas ECC:

- Las firmas ECC se crean utilizando una clave privada secreta y el algoritmo de la curva elíptica. Para el propósito de este ejemplo, podemos ignorar la impresión de la clave pública de las llaves en la parte superior - esto es parte de la función printInfo(). Como nota adicional, nunca conoceremos la clave privada del dispositivo. La clave privada de este dispositivo se crea, se almacena y se bloquea dentro del dispositivo.
- Las firmas ECC tienen una longitud de 64 bytes.
- Contrariamente a la mayoría de las definiciones de las firmas, las firmas ECC son diferentes cada vez. Las ECC incluyen aleatoriedad intencionada en el cálculo, por lo que siempre producirán una nueva firma única. Sin embargo, cada firma que cree pasará la verificación

(como veremos más adelante en el ejemplo 3). Pruebe a pulsar “reset” en su Artemis unas cuantas veces y observe cómo cambian los valores de la firma. Mismo mensaje + misma clave privada = nueva firma.

- Con el ATECC508A, sólo podemos enviarle 32 bytes de datos para firmar. Si necesitas firmar algo más grande, entonces se recomienda crear primero un hash de 32 bytes de los datos, y luego puedes firmar el hash. Básicamente, cuando envías un montón de datos a un algoritmo de hash (como SHA256), éste siempre responderá con un resumen único de 32 bytes. Es muy similar a la creación de una firma digital, sin embargo no requiere una clave, por lo que cualquiera puede calcular el hash de cualquier dato. Puedes encontrar que para muchos proyectos, 32 bytes son suficientes datos, así que por ahora, no hay necesidad de buscar hashes.

Al firmar el mensaje, obtendremos la siguiente salida:

```
This device's Public Key:


uint8_t publicKey[64] = {

0xB2, 0xAA, 0xE7, 0x84, 0x1D, 0x43, 0x5C, 0xE6, 0x49, 0xFD, 0x26, 0x3B,
0x8D, 0xC2, 0xF8, 0x2A,

0x20, 0x49, 0x9A, 0xFC, 0xAE, 0xFE, 0x25, 0x1C, 0x6A, 0x90, 0x26, 0xC6,
0x40, 0xC3, 0x4C, 0x5F,

0x3A, 0x98, 0xAA, 0xA4, 0x2B, 0xFE, 0x46, 0x40, 0x99, 0xB4, 0xC5, 0x26,
0x81, 0x94, 0x6B, 0x18,

0xDF, 0x3D, 0xE6, 0x18, 0x6D, 0x4C, 0x61, 0xE0, 0x1F, 0xD2, 0x4F, 0x73,
0x80, 0xB2, 0x2E, 0x68

};


uint8_t message[32] = {

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B,
0x1C, 0x1D, 0x1E, 0x1F

};


uint8_t signature[64] = {

0xD6, 0xC6, 0x53, 0xB5, 0x8E, 0xD3, 0xFA, 0x4A, 0xD8, 0xB6, 0xAE, 0x9A,
0x0F, 0x71, 0xD3, 0x17,
```

```
0xBB, 0x87, 0x0C, 0xF4, 0xE9, 0xDB, 0xFE, 0x44, 0x05, 0x06, 0x3B, 0xD2,
0x09, 0x6A, 0x68, 0x5C,

0xC1, 0xF5, 0x1F, 0xB8, 0xD8, 0x44, 0xB3, 0x64, 0x75, 0xEC, 0xE1, 0xD5,
0x8C, 0xFA, 0x26, 0x8A,

0x89, 0x9B, 0xC2, 0x27, 0x85, 0x94, 0x51, 0xC3, 0xB9, 0xBF, 0xE3, 0x40,
0x57, 0x8E, 0x6D, 0x3B

};
```

La función de la librería usada para realizar la firma digital de un mensaje es **createSignature(mensaje)**, usando por defecto la clave privada localizada y bloqueada en el chip.

Ejemplo 3 - Verificación de Firma Digital

En este caso, se dará un mensaje, una clave pública y una firma. Debido a la naturaleza de las claves asimétricas, partiendo de un mensaje firmado por una clave privada, con la clave pública asociada a dicha clave privada podemos verificar que el mensaje está firmado por un usuario particular.

La función de la librería usada para realizar la firma digital de un mensaje es **verifySignature(mensaje, mensajeFirmado, clavePublica)**

Ejemplo 5 - Generación de números aleatorios

Otra función aportada por la librería de Sparkfun y soportada por el chip criptográfico ATECC508A es la generación de números aleatorios de 32 bytes.

Un ejemplo de la salida obtenida es la siguiente:

```
Random number: 46

Random number2: 467

Random Byte: 0x1D

Random Int: 28715

Random Long: 1027057910

atecc.random32Bytes[32]:
F7BC4EAE52BC6B34946B45FAA6EF8E0BBBB8726363F01190BB8F0CAE46F6028D


Random number: 15

Random number2: 201
```

```
Random Byte: 0x94

Random Int: 31296

Random Long: -193108556

atecc.random32Bytes[32]:
2BADC5F66F215490697DE8836C85C391088DB2227B76F9A72D9EF5E10F6A71E0


Random number: 43

Random number2: 441

Random Byte: 0x8

Random Int: 22112

Random Long: -408265483

atecc.random32Bytes[32]:
E453A7D19375CFA3A72689B045FC2E1E8C25D3F5217F4607AC4D15F53E2DBCEB
```

Como podemos ver en los ejemplos, esta librería tiene varias opciones de generar números aleatorios.

- Generación de número en un intervalo con **random(intervalo)**
- Generación de byte aleatorio con **getRandomByte()**
- Generación de entero (int) aleatorio con **getRandomInt()**
- Generación de long aleatorio con **getRandomLong()**
- Generación de valor de 32 Bytes aleatorio con **updateRandom32Bytes()**

Esta última opción es muy interesante ya que por su naturaleza es prácticamente imposible que se repita un valor generado aleatoriamente de dichas características, por lo que puede ser usado para acompañar mensajes firmados o como parámetro en el intercambio de las claves criptográficas.

Resolución de errores

Si durante la ejecución de algún código donde imprimamos algo por el Serial no se ven los datos o se ven extraños caracteres, en primer lugar hay que comprobar si los baudios concuerdan con los especificados en el **Serial.begin(numBaudios)**.

Si el número es el mismo entre el monitor serial y el serial begin y sigue sin verse nada, es posible que la placa esté tardando demasiado en abrir el serial, sigue la ejecución del código y cuando el serial está abierto ya no hay nada que imprimir por él. Para evitar esto, si se da, hay dos opciones:

- Poner un **delay()** después del **Serial.begin()** con un tiempo lo suficientemente alto para que dé tiempo a iniciarse el serial antes de seguir ejecutando el resto del código. Tiene el

inconveniente de no ser una buena práctica que puede dar lugar a código no paralelizable.
- La otra opción es usar un **while (!Serial)** después del serial.begin, con el problema de que si no se abre un serial monitor el código no avanzará.

Otra opción de depuración es usar el comando **dmesg –w** es util al trabajar con embebidos, muestra si ha habido errores al subir el codigo a la placa.

Instalación del ACE Framework en el ESP32

Para comenzar este proceso, es conveniente instalar en el ordenador donde se trabaje el framework Mongoose OS, utilizado para el desarrollo de firmware.

Para la instalación de Mongoose OS en Linux se siguen los siguientes comando:

```
sudo add-apt-repository ppa:mongoose-os/mos
sudo apt-get update
sudo apt-get install mos
```

Y ejecutando **mos** en la terminal, se abrirá en nuestro navegador en la dirección http://127.0.0.1:1992/ una interfaz gráfica para trabajar con él.

Desde esta interfaz, después de configurarla correctamente escogiendo el puerto y la placa con la que se está trabajando, nos tenemos que situar en la carpeta donde se encuentre descargado el proyecto y ejecutar **mos build**.

Una vez termine, ya flasheamos la memoria del ESP32 con el comando **mos flash**. Tras esto, y como último paso tenemos que configurar la placa para que se conecte a una dirección WiFi, usando el comando **mos wifi \<NOMBREDELARED> <CONSTRASEÑA>**. Una vez se haya conectado, veremos por la terminal la dirección IP del dispositivo, por lo que con esta dirección y sabiendo que es el puerto 8000 el que se encuentra abierto (se puede comprobar haciendo uso del comando **nmap** o bien buscando en el código que hemos flasheado en la placa), sólo falta instalar el Authorization Server y el Client en nuestra máquina local, y hacer que ambas instancias apunten al ESP32 con el Authorization Server.

Problemas en Mongoose

Para construir el firmware usando el comando **mos build** se usa un servidor cloud proporcionado por Mongoose. En algunas ocasiones, el certificado x509 expira, por lo que si se ejecuta el comando se devolverá el siguiente mensaje de error:

```
Error: Post https://build.mongoose-os.com/api/fwbuild/2.20.0/build: x509:
certificate has expired or is not yet valid
/build/mos-Wt49Ss/mos-2.20.0+0278853~focal0/cli/build_remote.go:329:
/build/mos-Wt49Ss/mos-2.20.0+0278853~focal0/cli/build.go:278:
/build/mos-Wt49Ss/mos-2.20.0+0278853~focal0/cli/build.go:221:
/build/mos-Wt49Ss/mos-2.20.0+0278853~focal0/cli/main.go:194: build failed
```

```
exit status 1
```

Para abordar este problema hay dos enfoques, el primero construir el firmware localmente con el comando **mos build –local –platform esp32** desde la terminal de nuestro ordenador. La parte negativa es que para construir el firmware localmente se necesitan muchos recursos por lo que cada ejecución es muy lenta. Existe otra manera de construir el firmware localmente siguiendo las instrucciones del enlace:

```
https://github.com/v-kiniv/mos-native
```

Si se quiere evitar tener que construir las imágenes localmente, con el esfuerzo y el tiempo que conlleva, la otra opción es contactar con los administradores de la nube de mongoose para que la vuelvan a levantar los certificados, ya que en muchas ocasiones puedes ser el primero en darte cuenta de que han caducado. Para ello, acceder al siguiente grupo y avisar del fallo.

```
https://gitter.im/cesanta/mongoose-os?at=5a7e25adce68c3bc74690082
```