Computación Ubicua

Sistema de gestión de puestos de estudio (STUBIA)

Grado en Ingeniería Informática
Universidad de Alcalá

Carlos Javier Prado Vázquez
Guillermo González Martínez
Pablo García García
Robert Petrisor

14 de diciembre de 2022

Índice general

Índice de figuras

Resumen

Introducción

Capítulo 1

Objetivos del proyecto

Capítulo 2

Arquitectura del proyecto

2.1. Capa de percepción

En esta capa, vamos a explanar la manera de proceder que se ha realizado para implementar el entorno.

En cuanto a la fase de instalación, se tiene que partir del caso base de tener incorporado la última versión instalada del IDE Arduino en nuestro equipo junto con la placa de desarrollo NodeMCU (ESP32). Cabe destacar que necesitaremos un cable microUSB de transferencia de datos para la conexión entre ambos dispositivos. Dado que, si dicho cable tiene la única función de alimentación, meramente no se podrá realizar la conexión.

Figura 2.1: Conexión ESP-PC

Es interesante conocer que, en la instalación del IDE Arduino, no se dispone de la placa ESP32, aunque sí contiene una gran variedad y tipos de placas de desarrollo de Arduino. Seguidamente, una vez abierto el IDE Arduino, vamos a acceder a la siguiente opción: *Archivo* > *Preferencias*. Otra forma similar de acceder sería tecleando simultáneamente las teclas: "`Ctrl + ,`".

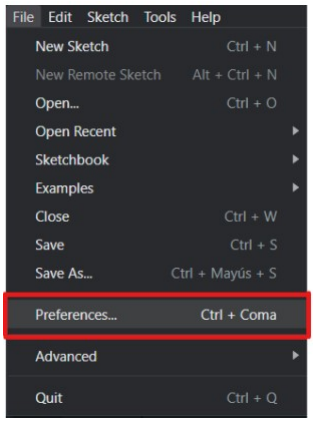


Figura 2.2: Menú preferencias

Inmediatamente se nos abrirá una ventana de preferencias automáticamente. En la sección de *Ajustes*, hacemos la búsqueda en la sección que aparezca la opción de *Gestor de URLs adicionales de tarjetas*, donde nos aparecerá un campo de texto para insertar una librería URL. En nuestro caso, como necesitaremos usar una ESP32, emplazaremos la siguiente librería: `https://dl.espressif.com/dl/package_esp32_index.json` y cerraremos dicha ventana, haciendo clic en "*OK*".

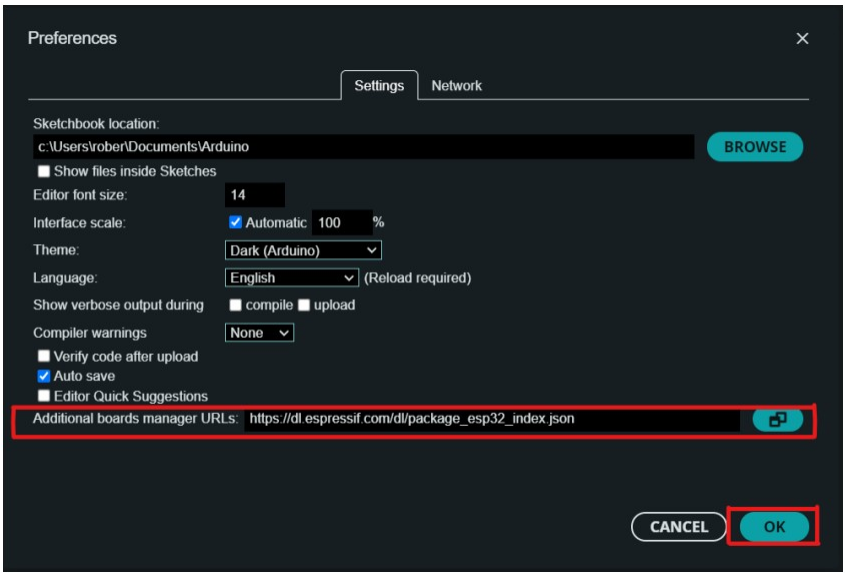


Figura 2.3: Importar datos para ESP-32

A continuación, vamos a acceder a la siguiente opción: *Herramientas* > *Placas* > *Gestor de placas (Tarjetas)*.

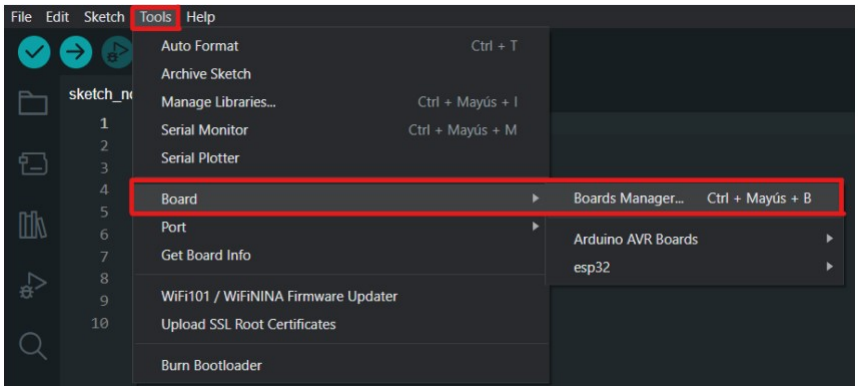


Figura 2.4: Gestión de placas

En la opción que aparece como filtro por búsqueda, escribiremos en el campo de texto: ESP32. Instalaremos el único paquete que viene por defecto de la placa ESP (ESP32 by Espressif Systems v1.0.6). Una vez instalado el paquete, cerramos la ventana de gestor de tarjetas.

Figura 2.5: Elegir ESP-32

A continuación, vamos a seleccionar una placa ESP32 específica. Hay un montón de ellas, pero en nuestro caso, vamos a elegir la nuestra. Para esto, accedemos a *Herramientas > Placa > ESP32 > ESP Dev Module*.

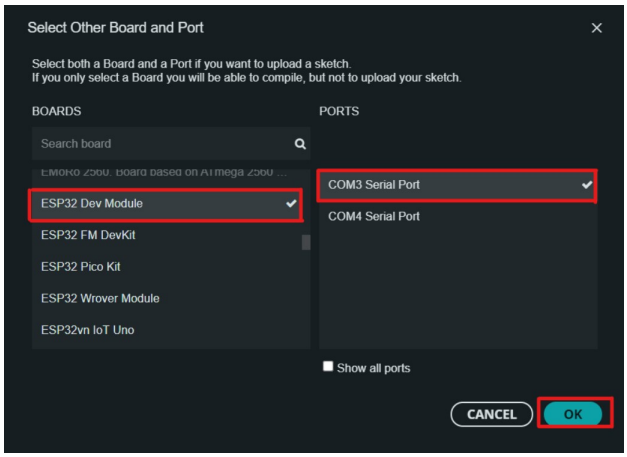


Figura 2.6: Selecionar puerto correcto

A continuación, nos pide seleccionar un puerto, el puerto al que esté conectado para la conexión con la placa de desarrollo física (ESP). En caso de que la ESP esté conectada al puerto USB del ordenador a través del cable microUSB, le pondrá por defecto uno (`COMX`, siendo X un entero positivo asignado por el sistema). En caso de no saber qué puerto seleccionar, abriremos el administrador de dispositivos. Seleccionaremos Puertos (`COM` y `LPT`), desplegamos y buscamos una entrada de este estilo: "*Silicon Labs CP210x USB to UART Bridge (COMX)*". El valor que nos aparezca en el COMX, ese será el valor del puerto al que se realizará la conexión entre nuestro programa con la ESP.

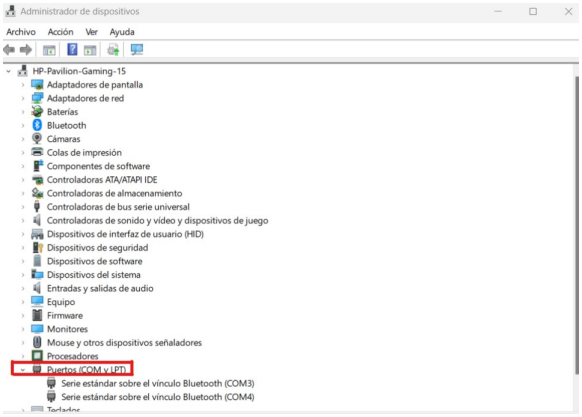


Figura 2.7: Puertos `COM` en administrador de dispositivos

En caso de que no nos aparezca ese puerto, habría que comprobar si el cable microUSB sirve para la transferencia de datos, y no solo para la alimentación. Otro posible caso, sería que se necesitaría instalar un driver específico en nuestro sistema que pueda detectar nuestra placa ESP. En nuestro caso, sería el driver `CP210x` para Windows x64 bits.

Figura 2.8: Driver a instalar para Windows

Una vez escogida la placa ESP Dev Module y el puerto COMX, ya queda terminado el proceso de instalación de los recursos necesarios para empezar a implementar el montaje tanto físico como simulado y dicha codificación de los ficheros con extensión `ino`.

2.1.1. Sensores

Tenemos dos tipos de sensores que vamos a implementar, uno es un sensor de ultrasonidos, y el otro es un sensor de inclinación. Vamos a explicar brevemente cómo se podría implementar estos tipos de sensores a partir de una simulación. El simulador que vamos a emplear para dicho montaje virtual se llama Tinkerkad, donde se dispone de un Arduino Uno. Cabe destacar que, en nuestro proyecto empleamos una placa de desarrollo NodeMCU (ESP32), pero que nos sirve para la funcionalidad de comprobación de cada uno de los circuitos que vamos a montar a continuación. Además, de que el código es el mismo para ambos dispositivos. También tiene pines tanto analógicos como digitales para abarcar diferentes conexiones.

2.1.1.1. Sensor de ultrasonidos

El sensor de ultrasonidos está basado en determinar el tiempo que la señal sonora tarda en ir y volver hasta un obstáculo. Para ello, tenemos dos pines importantes: el `triggerPin` y el `echoPin`. El `triggerPin`, es el que da la señal de salida de un pulso. Este sonido rebota en el obstáculo y vuelve hasta el otro sensor que lo recoge a través del `echoPin`.

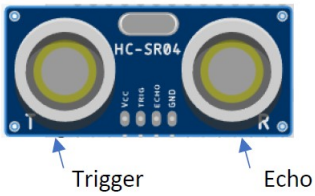
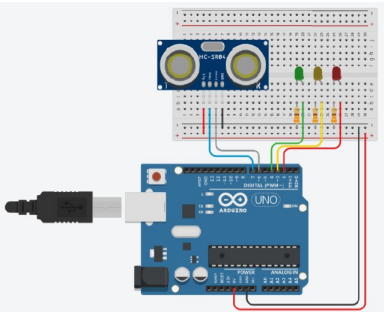


Figura 2.9: Sensor de ultrasonidos

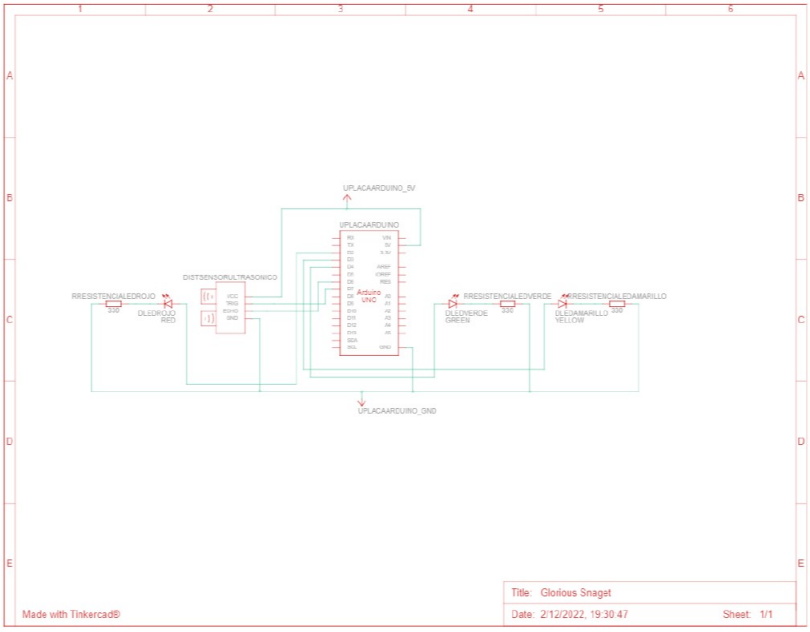
Para empezar con el montaje, necesitaremos: una placa del microcontrolador (Arduino Uno, la que viene por defecto en el simulador online Tinkercad), una protoboard, el sensor de ultrasonidos, un semáforo led, resistencias para cada led (330 Ω), y cables para la conexión. Cabe destacar que, en el modelo físico, ya tenemos un semáforo con resistencias incorporadas dentro del mismo. Para ello, partimos de introducir el sensor en la protoboard, enlazamos la corriente (VCC) del sensor al polo negativo (−), después enlazamos la tierra (GND) del sensor al polo positivo (+), y el pin Trigger a un pin digital de la placa Arduino. Finalmente, enlazamos el pin Echo a un pin digital de la placa Arduino.

A continuación, introducimos los leds RGB en la protoboard. En cada uno de ellos, enlazamos el cátodo al polo negativo (−), y el ánodo al polo positivo (+). Además, le hemos aplicado unas resistencias de 330 Ω enlazado con el cátodo de cada led. Por último, solo hace falta enlazar el polo positivo (+) de la protoboard con la corriente (5V) de la placa Arduino,

y el polo negativo (−) con la tierra (GND) de la placa Arduino. El esquema nos quedaría de la siguiente forma:



(a) Simulación



(b) Esquema

Figura 2.10: Modelo tipo laboratorio

Como hemos mencionado anteriormente, el sensor de ultrasonidos tiene un emisor y un receptor. El emisor va a emitir un pulso en ultrasonido (es decir, insonoro) a la velocidad del sonido, y lo que va a hacer ese pulso es rebotar contra un obstáculo si es que lo hay. Entonces, rebota, lo devuelve, y lo detecta el receptor.

Vamos a explicar cómo podemos obtener la distancia para el sensor de ultrasonidos. Como ya sabemos, la velocidad a la que viaja el sonido, en las condiciones que nos encontramos es de 344 m/s. Sabiendo el tiempo que tarda y la velocidad a la que viaja el sonido, podemos obtener la distancia a la que se encuentra el obstáculo (en caso de que haya) con respecto al sensor. Despejando, $s = v \cdot t$. Es decir, cuanto mayor sea el tiempo, más lejos se encontraría el obstáculo. A continuación necesitaremos hacer una serie de cambios de unidades, pues nos interesa expresar el tiempo en μs y la distancia en cm.

$$344 \text{ m/s} = 34400 \text{ cm/s} = 0{,}0344 \text{ cm/}\mu\text{s} \Longrightarrow s = 0{,}0344 \cdot t$$

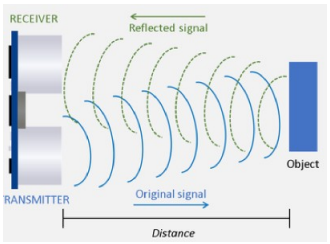


Figura 2.11: Señales en un sensor de ultrasonidos

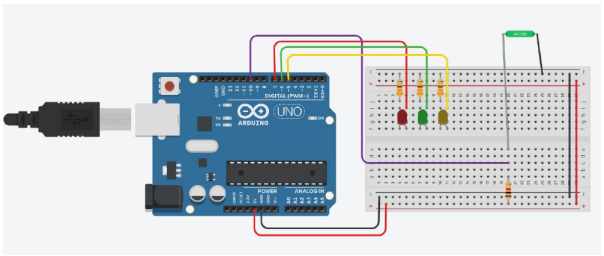
Como la distancia al obstáculo es la mitad del espacio que recorre, porque el tiempo que tarda es el tiempo que tarda en ir y volver, tenemos que dividir ese factor entre dos

$$d = \frac{s}{2} \Longrightarrow d = 0{,}01723 \cdot t$$

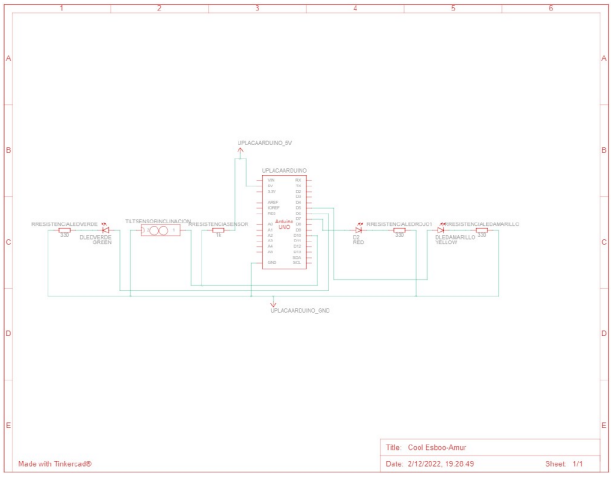
2.1.1.2. Sensor de inclinación

Un sensor de inclinación es un dispositivo que proporciona una señal digital en caso de que su inclinación supere un umbral. Este tipo de sensor no permite saber el grado de inclinación del dispositivo, sino que simplemente actúa como un sensor que se cierra a partir de una cierta inclinación. Cabe destacar que en el simulador tiene el sensor de inclinación (modelo `SW200D`), diferente al nuestro (`SW520D`). Pero que, en este caso, funciona con la misma dinámica.

Para empezar con el montaje, necesitaremos: una placa del microcontrolador (Arduino Uno, la que viene por defecto en el simulador online, Tinkercad), una protoboard, el sensor de inclinación, un semáforo led, resistencias para cada led (330 Ω), y cables para la conexión. Como comentamos previamente, en el modelo físico ya tenemos un semáforo con resistencias integradas. Partimos de introducir el sensor en la protoboard, enlazamos una terminal del sensor al polo negativo (−) de la protoboard, enlazamos la otra terminal del sensor en línea con una resistencia de 1kΩ, donde la resistencia la conectamos al polo positivo (+) de la protoboard. Ahora creamos un conector de voltaje en línea con la terminal del sensor y la resistencia. Enlazamos donde hemos puesto ese divisor de voltaje con un pin digital de la placa Arduino. A continuación, introducimos los leds RGB en la protoboard. En cada uno de ellos, enlazamos el cátodo al polo negativo (−) de la protoboard; y enlazamos el ánodo al polo positivo (+) de la protoboard. Además, le hemos aplicado unas resistencias de 330 Ω enlazado con el cátodo de cada led. Por último, solo hace falta enlazar el polo positivo (+) de la protoboard con la corriente (5V) de la placa Arduino, y el polo negativo (−) con la tierra (GND). Además, enlazamos los ambos polos (positivos y negativos) de los extremos de la protoboard, para que circule la corriente en toda la placa. EL esquema queda de la siguiente manera:



(a) Simulación



(b) Esquema

Figura 2.12: Modelo tipo teoría

2.1.2. ESP-32

Vamos ahora a tratar cómo se ha montado y programado esta simulación previamente comentada en la realidad. Para ello hemos hecho uso de un ESP-32, en el que simulamos tener, por ejemplo, dos puestos de tipo laboratorio y uno de tipo teoría, aunque como veremos ahora en la programación, el código está hecho de manera que podamos poner tantos puestos de cada tipo como queramos, siempre que no nos quedemos sin pines, obviamente.

Comencemos con la programación. Si bien Arduino utiliza C++ como lenguaje para programarlo, también pueden usarse expresiones válidas en C. Nuestro ESP-32 funciona exactamente igual con el tema de los lenguajes. Nuestro proyecto va a constar de tres ficheros:

- `constantes.h`
- `config.h`
- `stubia.ino`

2.1.2.1. Archivos de cabecera

Empecemos por `constantes.h`. Aquí se definen valores que es posible que se necesiten cambiar con cierta frecuencia, como nombre y contraseña de la red WiFi a conectarse, dirección IP del servidor principal, dirección URL del *webservice*, en qué aula de teoría o laboratorio se encuentra el ESP, y las calibraciones del sensor de ultrasonidos.

En `config.h` definiremos tres constantes fundamentales y las estructuras de datos con las que trabajaremos. Si empezamos por las constantes, la primera de ellas será `portMAX_delay`, que deberemos convertir a tipo `TickType_t`. ¿De qué se trata? El nombre del tipo al que queremos convertir, y su librería nos dan una pista. Más adelante cuando expliquemos cómo

funciona el código, veremos que vamos a aprovechar los dos *cores* de los que cuenta nuestro ESP, de forma que trabajaremos de forma concurrente pero con una memoria común. Como podremos recordar, esto nos trae los clásicos problemas de condiciones de carrera, comunicación, sincronización, y comunicación sincronizada del paradigma concurrente. Veremos que para resolver nuestro problema haremos uso de elementos mutex para acceder a los recursos compartidos en exclusión mutua, y es aquí donde entra en juego nuestro `portMAX_delay`. Según `freertos` define los mutex, cuando intentamos "cogerlos", la tarea que queda pausada, llamémosla T_1, debe estarlo solo durante cierto tiempo τ, pero nosotros queremos que esto sea hasta que la tarea que había cogido previamente el mutex, llamémosla T_2, lo suelte.

La solución más sencilla será definir τ como un número tan grande que parezca que es un tiempo indefinido. Ya que nuestro ESP trabaja con palabras de 32 bits, podemos definir como tiempo máximo de espera $2^{32} - 1$, lo que es equivalente a un número de 32 bits con todo unos. Podemos recordar que esta igualdad es cierta aplicando la suma de una serie geométrica de razón 2:

$$\sum_{k=0}^{n-1} 2^k = \frac{2^0 - 2 \cdot 2^{n-1}}{1 - 2} = 2^n - 1$$

En lenguaje C, podemos definir el mayor `long` sin signo como `~0UL`, lo que es equivalente a $2^{64} - 1$, pero como decíamos, como nuestro ESP trabaja con 32 bits, por lo que lo ve como $2^{32} - 1$, que es justo el valor que buscábamos. Teniendo en cuenta que se hace un *tick* cada milisegundo, podríamos quedarnos esperando casi dos meses, tiempo que para nuestro prototipo es más que de sobra.

Después definimos otras dos constantes, que definen cada cuántos milisegundos se vuelve a repetir el bucle principal de cada tarea. Ahora ya nos encontramos con la definición de cuatro estructuras diferentes.

- `puesto_fijo`: define un puesto de tipo teoría, guarda el aula, número de puesto, y pin de la placa a la que está conectado el cable de datos del sensor de inclinación respectivo. También tiene un puntero a una estructura que representa el semáforo respectivo del puesto.
- `puesto_movible`: al igual que `puesto_fijo`, guarda lo mismo excepto el pin de datos, que esta vez serán dos, el de `trigger` y el de `echo`.
- `semaforo`: representa un semáforo led de un puesto. Guarda los números de los pines de cada uno de los leds del semáforo. Tiene un puntero a una estructura `estado_semaforo`.
- `estado_semaforo`: serán recursos compartidos entre T_1 y T_2 que indiquen si cada led de cada semáforo debe estar estar encendido o apagado, es decir, estado `LOW` o `HIGH`.

Después, en el programa principal tendremos un array de puestos fijos, y otro de puestos movibles. Veamos un esquema de estas estructuras para que sea más sencillo de entender cómo vamos a almacenar los datos.

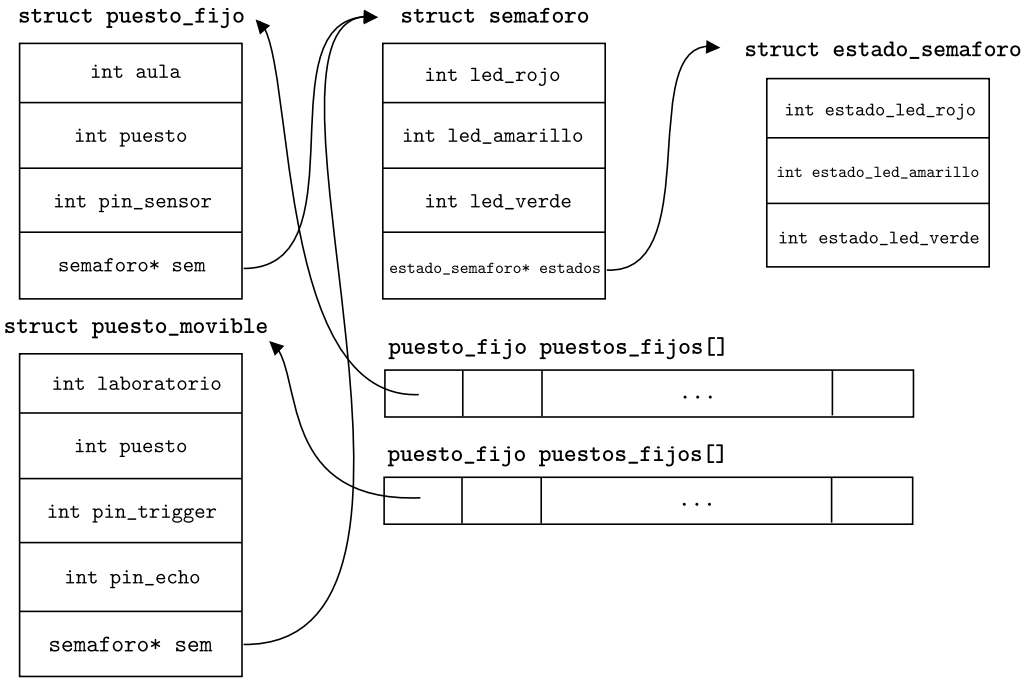


Figura 2.13: Esquema de las estructuras usadas

2.1.2.2. Programa principal

Si ahora pasamos a comentar el programa principal, `stubia.ino`, veremos que lo primero que hacemos es incluir todos los archivos de cabecera que hemos mencionado previamente, y las librerías que usaremos para controlar WiFi, llamadas HTTP, y tareas. Después, empezamos a declarar variables que representan las estructuras de datos representadas en la figura 2.13, siendo aquí donde ponemos los pines en los que hemos hecho las respectivas conexiones.

En nuestro caso hemos puesto en la placa un puesto de teoría y dos de laboratorio (en la realidad todos serían de teoría o todos de laboratorio), ocupando el de teoría los pines 35 para el sensor, y 18, 19, 21 para su semáforo; el primero de laboratorio los pines 23, y 22 para el ultrasonidos, y 12, 13, 14 para su semáforo; y el segundo de teoría ocupa el 25, y 26 para su ultrasonidos, y 15, 16, y 17 para su semáforo. También declararemos dos variables relacionadas con la tarea que se ejecutará en el segundo core, `comunicar_servidor`, que representa la propia tarea, y mutex, que será el objeto que permita sincronizar a T_1 y T_2 de forma que no existan condiciones de carrera.

Ahora definimos tres funciones auxiliares, `setup_wifi`, `setup_pines`, y `lectura_ultrasonidos`. La primera de ellas hace uso de la librería `WiFi.h`, y los métodos `begin()` y `status()` para que mediante las constantes definidas en `constantes.h` el ESP pueda conectarse a una red WiFi a través de la cual poder intercambiar información con el servidor.

La siguiente, `setup_pines`, se ocupará de ir recorriendo las estructuras de datos previa-

mente comentadas para ir fijando cada uno de los pines en modo entrada o salida. Un detalle muy importante a tener en cuenta es que en esta función, los dos arrays que pasamos deben pasarse por referencia, pues si los pasamos como parámetros, aunque en una posición de memoria tengamos el número 10, `pinMode(10, INPUT)` no va a saber poner el pin en modo entrada, provocando un fallo en el sistema. Vemos aquí qué modo corresponde a cada pin según su uso, y cómo queda perfectamente organizado gracias al uso de `structs`:

```
35 void setup_pines(puesto_fijo* puestos_teoria, puesto_movible* puestos_lab)
       {
36   for(int i = 0; i<size_fijos; i++){
37     pinMode(puestos_teoria[i].pin_sensor, INPUT);
38     pinMode(puestos_teoria[i].sem->led_rojo, OUTPUT);
39     pinMode(puestos_teoria[i].sem->led_amarillo, OUTPUT);
40     pinMode(puestos_teoria[i].sem->led_verde, OUTPUT);
41   }
42
43   for(int i = 0; i<size_movibles; i++){
44     pinMode(puestos_lab[i].pin_trigger, OUTPUT);
45     pinMode(puestos_lab[i].pin_echo, INPUT);
46     pinMode(puestos_lab[i].sem->led_rojo, OUTPUT);
47     pinMode(puestos_lab[i].sem->led_amarillo, OUTPUT);
48     pinMode(puestos_lab[i].sem->led_verde, OUTPUT);
49   }
50 }
```

Código 2.1: `setup_pines`

Para calcular el número de elementos de cada array, dividimos el número de bytes que ocupa el array, entre el número de bytes que ocupa una estructura del tipo que almacena. La siguiente función que nos encontramos es `lectura_ultrasonidos()`, que dados un pin de `trigger` y otro de `echo`, nos devuelve una distancia. Funciona y realiza lo ya explicado en el punto 2.1.1.1. La última función creada por nosotros que encontramos es `loop2()`, pero la dejaremos para más adelante, pues es el código de T_2.

Ahora ya nos encontramos con la función `setup()`, función que se ejecutará una sola vez y hará las inicializaciones necesarias, en nuestro iniciar la comunicación serial (a 115200 bits por segundo, por ejemplo), crear nuestro mutex con `xSemaphoreCreateMutex()`, iniciar la conexión WiFi con `setup_wifi()`, poner el modo de los pines con `setup_pines()`, y por último crear y lanzar nuestra segunda tarea en paralelo con `xTaskCreate()`, aunque dejaremos los parámetros que esta toma, y demás asuntos relacionados para la sección 2.1.2.3.

Pasemos a hablar de qué hace la tarea T_1 o principal de nuestro código. Lo que haremos cada tantos milisegundos como estén definidos en `WAIT_SENSOR_CORE` (en `config.h`), será, para cada puesto de teoría, leer el valor booleano que nos envíe el sensor diciendo si el asiento está libre o no, y ahora apuntaremos en cada uno de los estados de los semáforos del puesto si el led rojo y verde deben estar cada uno a `HIGH` o `LOW`. En la sección 2.1.2.3 veremos que esta parte de "apuntar" `LOW` o `HIGH` se trata de una región crítica que deberemos acceder en exclusión mutua. Una vez leído y apuntado el valor del estado, haremos uso de `digitalWrite()`, pasando los pines de los semáforos y sus estados para así iluminar

correctamente los leds. Después, nos encontraremos con otro bucle que hace exactamente lo mismo pero con los puestos de laboratorio, por lo que en vez de consultar un valor booleano, hacemos uso de `lectura_ultrasonidos()` y comprobamos si dicho valor es mayor o menor que `CALIBRACION_DIST`, también definido en los archivos de cabecera para poder ser modificado a gusto del usuario. Observemos que como cada ESP irá destinado a una clase de teoría o a un laboratorio, en la práctica solo se ejecutará uno de los bucles comentados.

2.1.2.3. Programación concurrente y mutex

Si bien hemos comentado que la tarea principal de nuestro ESP es recoger valores de los sensores e iluminar en rojo y verde los semáforos, no debemos olvidar que disponemos de una aplicación web que ofrece multitud de servicios pero que necesita la ayuda de nuestros sensores para poder realizarlos de forma correcta. Aprovecharemos la propia arquitectura del ESP-32 para ejecutar en el primero de los núcleos lo comentado en la sección 2.1.2.2 para de manera casi instantánea preguntar valores a los sensores, guardarlos, e iluminar ciertos leds; y en el segundo núcleo cada cierto tiempo "hablar" con el servidor comunicándole el estado de los sensores para que pueda ser mostrado en la web y almacenarlo en la base de datos, y que el nos diga si nuestro puesto está reservado o no, lo que implicaría encender o apagar el led amarillo.

Para lanzar una segunda tarea en paralelo, en el setup hacemos uso de la función `xTaskCreate()`. A esta debemos pasarle como parámetros:

- Nombre de la función a ejecutar
- Nombre asignado a la tarea
- Tamaño de pila en palabras
- Puntero a parámetros
- Prioridad
- Puntero a la tarea

Nosotros asignamos como nombre y función, la que hemos definido como `loop2()`. Como tamaño de pila `configMINIMAL_STACK_SIZE * 5`, pues al no haber una forma determinista de hallar cuánto espacio varios a necesitar, hemos ido acotando de forma heurística a partir de qué tamaño se nos quedaba pequeña (en dichos casos el comportamiento del ESP era extraño). Como parámetros pasamos un puntero a `NULL`, pues no tenemos, como prioridad 1, y como tarea pasamos un puntero a la tarea definida al principio del código.

Pasemos a explicar la parte de concurrencia y recursos compartidos. Recordemos que estamos ejecutando diferente código en diferentes cores, pero que seguimos teniendo una memoria común donde compartimos variables entre diferentes tareas, por lo que se pueden dar las conocidas como condiciones de carrera. Supongamos que tenemos dos tareas y un recurso compartido, y que ambas pueden leer y escribirlo (no es tal cual nuestro caso pero nos ayudará a entenderlo), veamos qué sucede en diferentes situaciones:

- T_1 lee y T_2 lee $\Longrightarrow$ no hay problema, pues ambas leerán siempre el mismo valor.
- T_1 lee y T_2 escribe, o al revés $\Longrightarrow$ **Condiciones de carrera**: mientras T_1 lee, T_2 puede estar cambiando el valor, de T_2 de forma que T_1 esté tomando un valor erróneo.
- T_1 escribe y T_2 escribe $\Longrightarrow$ **Condiciones de carrera**: como ambas pueden escribir a la vez, no tenemos forma de saber cuál de los dos valores quedará en memoria, será inconsistente.

De manera ingenua, una solución sencilla aunque rompería el paradigma, sería hacer que no sucedan varias cosas a la vez, sin embargo, podemos seguir haciendo que varias cosas trabajen a la vez, pero que a la hora de acceder a un recurso compartido lo hagan de uno en uno, o lo que es lo mismo, en exclusión mutua, evitando así estados inesperados. Nuestra solución son los mutex, que nos solucionarán este problema de sincronización entre tareas. Crearemos uno en el `setup` con `xSemaphoreCreateMutex()` y lo compartiremos con ambas tareas, al igual que nuestros arrays.

A la hora de acceder a la región crítica, lo que haremos será "*echar el pestillo*" con `xSemaphoreTake(mutex, portMAX_delay)`[1], que nos devolverá `pdTRUE` en caso de que todo haya ido de forma correcta, lo que significará que podemos entrar. Haremos diversas operaciones con el recurso compartido, en nuestro caso apuntaremos en los estados del semáforo si cada led ha de estar en `LOW` o en `HIGH`, y una vez hemos hemos terminado, "*quitamos el pestillo y salimos del baño*" con la instrucción `xSemaphoreGive(mutex)`. Veamos la forma que esto tiene en nuestra función principal al querer actualizar los estados de los semáforos:

```
161     if(xSemaphoreTake(mutex, portMAX_delay) == pdTRUE){
162       if(!inclinado){
163         puestos_fijos[i].sem->estados->estado_led_verde = LOW;
164         puestos_fijos[i].sem->estados->estado_led_rojo = HIGH;
165       }
166       else{
167         puestos_fijos[i].sem->estados->estado_led_verde = HIGH;
168         puestos_fijos[i].sem->estados->estado_led_rojo = LOW;
169       }
170     }
171     xSemaphoreGive(mutex);
```

Código 2.2: `Acceso a sección crítica en exclusión mutua`

Podemos realizar una observación, y es que accedemos solo en exclusión mutua a la hora de actualizar en la estructura el valor leído, pero no a la hora de hacer `digitalWrite`, ¿a qué se debe esto? Como ahora veremos, nuestra T_2, solo va a leer el recurso compartido, por lo que si recordamos lo que explicábamos antes, lectura con lectura no entra en conflictos, en nuestro caso tenemos conflictos cuando T_1 está escribiendo (actualizando valores en las estructuras) y T_2 lee valores para mandárselos al servidor.

Expliquemos ahora el funcionamiento de nuestro `loop2()` que funciona en paralelo a `loop()`, tal y como estamos viendo. De la misma manera que `loop()` hará un bucle para

[1] Ya se ha explicado en la sección 2.1.2.1 qué significa `portMAX_delay`.

los puestos de teoría y otro para los de laboratorio (en la práctica solo se ejecuta uno), y en cada uno de ellos, en exclusión mutua va a consultar el estado del semáforo para generar una URL para llamar al webservice que se encuentra en nuestro servidor, será de la forma `http://stubia.com/stubia/ws/ws_setestadopuesto.php?aula=a&puesto=p&estado=e`, donde en aula se pone el valor de `AULA_TEORIA` o `AULA_LABORATORIO`, en puesto el valor ubicado en la estructura de datos, y en estado 1 si está libre o 2 si está ocupado. Al hacer esta llamada HTTP (ya fuera de la sección crítica), el servidor nos contesta en un HTML con `SI` o `NO`, dependiendo de si nuestro puesto está reservado o no, con lo que actualizaremos en nuestras estructuras de datos el estado del led amarillo y lo encenderemos o apagaremos. Otra peculiaridad a comentar es que si bien en `loop()` llamábamos a `delay()`, aquí debemos hacer uso de `vTaskDelay()`.

2.2. Capa de transporte

2.3. Capa de procesamiento

2.4. Servidor web

2.5. Base de datos

2.6. Capa de aplicación

Capítulo 3

Conclusiones

Apéndice A

Manual de instalación

Apéndice B

Manual de usuario de la aplicación web

Apéndice C

Simulación de datos

Bibliografía