Universidad de La Laguna

CC BY NC SA

Sistemas Embebidos: ESP32

OCW

Programación Moddable

ESP32: Moddable SDK

Permite programar en JavaScript: libro disponible

Instalación

- Primero instalar esp-idf v5.3
  - Probar ejemplo sencillo
- Después instalar Moddable SDK.
- Instalar las ESP32 tools.

- **Estructura de aplicación:**
  - Carpeta con al menos ficheros:
    - `main.js` → código JavaScritp de nuestra programa
    - `manifest.json` → fichero de configuración, incluye otros `manifest.json`
- **Compilación e instalación:**
  - Aplicación mcconfig → mcconfig -m -d -p esp32
    - -d para usar depurador xsbug
    - -i sin depuración, solo mensajes de istrumentación por la serial.
    - -p plataforma
      - -p lin ó -p win para simulador en Linux ó Windows
      - -p esp32 para subir a la placa de desarrollo

ESP32: Manifest y trace

Típico manifest.json

```
{
  "include": [
    "$(MODDABLE)/examples/manifest_base.json",
     <otros manifest según módulos usados>
  ],
  "modules": {
    "*": "./main"
  }
}
```

Función trace

- salida mesajes: `trace(string)`, `trace(objeto)`
  - por xsbug
  - por serial

ESP32: Temporización

Clase Timer

- módulo: import Timer from "timer";
- Timer.set(callback[, intervalo, repetición])
  - ejecuta función callback pasado intervalo msg, y se repite cada repetición msg.
  - devuelve id del temporizador creado
  - callback recibe como parámetro id del temporizador
- Timer.repeat(callback, repetición)
  - ejecución repetitiva del callback
- Timer.schedule(id [, intervalo[, repetición]])
  - reprograma temporizador ya existente id, cambiando intervalo y repetición.
- Timer.clear(id)
  - cancela temporizador id
- Timer.delay(intervalo)
  - retrasa ejecución el número de msg. pero **bloquea** ejecución del resto ⇒ NO USAR

ESP32: Acceso a pines

Digitales

- Clase `Digital` de módulo `pins/digital` → `import Digital from "pins/digital";`
  - Se puede crear objeto → `let pata = new Digital([port], pin, mode)`
    - `port`: opcional, nombre del puerto
    - `pin`: número de pin
    - `mode`: `Digital.Input, Digital.InputPullUp, Digital.InputPullDown, Digital.InputPullUpDown, Digital.Output, Digital.OutputOpenDrain`
    - se puede pasar todo en un diccionario
  - Métodos
    - `pata.read()`: devuelve valor leido (0 ó 1)
    - `pata.write(valor)`: saca en valor indicado (0 ó 1)
    - `pata.mode(modo)`: cambia a modo indicado
- Métodos estáticos
  - `Digital.read(pin)`: fija pin como entrada y devuelve valor
  - `Digital.write(pin, valor)`: fija pin como salida y saca valor

ESP32: cambios en pines

Clase Monitor

    - manifest.json → "$(MODULES)/pins/digital/monitor/manifest.json"
- modulo: `import Monitor from "pins/digital/monitor";`
- Constructor diccionario con: `pin, port, mode, edge`
  - `edge`: `Monitor.Rising`, `Monitor.Falling`. Ambos → `Monitor.Rising | Monitor.Falling`
- Propiedades y Métodos:
  - `onChanged`: asignar función callback para cuando se de evento
  - `read()`: valor actual del pin
  - `close()`: termina con la monitorización
  - `rises` y `falls`: contador interno de flancos que se han producido
    - más exacto que llamdas a `onChange`

ESP32: ADC y PWM

Entrada analógica: clases Analog

- modulo: `import Analog from "pins/analog";`
- Solo método estático: `Analog.read(pin)`
  - `pin`: es número pin del conversor ADC1 del ESP32, no el pin GPIO
  - devuelve valor entre 0 y 1023

Salida PWM: clase PWM

- modulo: `import PWM from "pins/pwm";`
- constructor diccionario con: `pin`, `port` (opcional)
- métodos:
  - `write(valor)` ciclo de trabajo entre 0 y 1023
  - `close()` detiene PWM y libera el pin.

ESP32: Wifi

Clase WiFi en módulo "wifi" → import WiFi from "wifi";

`manifest.json` → `"$(MODDABLE)/examples/manifest_net.json"`

- `WiFi.mode`: 1 → cliente; 2 → punto de acceso (AP); 3 → ambas cosas
- Escanear WiFis diponible → `WiFi.scan(opciones, callback)`
  - `opciones`: `channel` solo en canal indicado; `hidden` si `true` también AP ocultos
  - `callback`: recibe `encontrado` por cada AP y `null` cuando termina escaneo
    - `encontrado.ssid` → nombre del AP
    - `encontrado.rssi` → potencia señal
    - `encontrado.authentication` → tipo autenticación, `"none"` si es AP abierto
- Conectarse a AP → `Wifi.connect(opciones)`
  - opciones: `ssid/bssid` → indentifica WLAN/AP; `password` → si necesario
  - Para saber si conectado preguntar a `Net.get("IP")`
  - Desconectar con `WiFi.disconect()`
  - Si se desconecta de AP no hay reconexión automática.

ESP32: Wifi

- Monitorizar conexión a AP → instanciar objeto `WiFi`
  - `let monitor = new WiFi(opciones, callback)`
    - `opciones` → como las de connect: `ssid`/`bssid` y `password` si necesario
    - `callback` recibe `msg` indicando la fase:
      - `WiFi.connected` → cuando conectado al AP
      - `WiFi.gotIP` → ha recibido IP
      - `WiFi.disconnected` → ha perdido conexión, permite lanzar otro intento de reconexión.
    - `monitor.close()` → detiene la monitorización, pero deja Wifi conectada
- Crear un AP → `WiFi.accessPoint(opciones)`
  - `ssid` → identificador de red obligatorio
  - `channel` → canal
  - `hidden` → si queremos que se oculto
  - `interval` → ms entre anuncios del AP
  - `max` → máximo número de conexiones simultáneas
  - `station` → activar modo cliente simultáneamente

EPS32: Net

Clase Net gestiona conexión de red → `import Net from "net";`

- `"$(MODDABLE)/examples/manifest_net.json"`
- Si en `manifest` - `config` o línea comando pasa datos wifi → hace conexión automática
  - no es necesario preocuparse por módulo `WiFi`
  - `mcconfig -d -m -p esp32 ssid="my wi-fi" password="secret"`
  - se puede deshabilitar el auto setup en `manifest.json`

```
"config": {
     "ssid": "embebidos",
     "password": "22embebidos23"
}
```

```
"~": [
   "$(BUILD)/devices/esp32/setup/network"
 ]
```

- Contienes propiedades sobre estado red:
  - `Net.IP, Net.MAC, Net.SIID, Net.BSSID, Net.RSSI, Net.CHANEL, Net.DNS`
- Resolución de nombres
  - `Net.resolve("www.ull.es", (nombre, dirIP) => trace(`${nombre} tiene IP ${dirIP}\n`);.`

ESP32: Peticiones HTTP

Clase Request de "http" → `import {Request} from "http";`

- Constructor con diccionario con las siguientes opciones:
  - `hosts/adrress`: string con el nombre/dirección ip del servidor
  - `port`: puerto, por defecto 80.
  - `method`: GET, POST, etc. Por defecto GET.
  - `headers`: array con cabeceras a mandar: nombres(posiciones pares) y valores(posiciones impares)
  - `body`: si se quiere proporcionar un body en el request. Se puede poner String/ArrayBuffer o `true` para irlo enviando según lo pida el callback. Por defecto `false`.
  - `response`: como se quiere la respuesta
    - `String` → en un único String cuando esté completa
    - `ArrayBuffer` → en único ArrayBuffer cuando esté completa
    - `undefined` → por fragmentos en cuanto vayan llegando (por defecto).
- Método `close()` → cancela la petición.

ESP32: Peticiones HTTP

- Propiedad `callback`: función que recibe (message, val1, val2). Según message:
  - error → error
  - requestFragment → se pide fragmento de body. val1 máximo de bytes a enviar. Devolver String o ArrayBuffer
  - status → se ha recibido status. val1 contiene el valor (ej 200 ⇒ OK)
  - header → recibida nueva cabecera. val1 = nombre, val2 = valor
  - headersComplete → ya están todas las cabeceras
  - responseFragment → fragmento de la respuesta (si response == undefined). val1 = cuantos bytes disponibles. Leerlos con read()
  - responseFragment → respuesta completa (si response != undefined). val1 = la respuesta completa
- Método read(tipo [, hasta])
  - tipo: tipo de datos en el que se devuelve lo leído → String, ArrayBuffer o Number (solo 1 byte)
    - si tipo == null se descarta lo leído y se devuelve el número de bytes descartados.
  - hasta:
    - si no figura → se devuelve todo lo disponible
    - si entero → con cuantos bytes a leer
    - si string → hasta que se encuente el carácter indicado

ESP32: Servidor HTTP

Clase Server de "http" → **`import {Server} from "http";`**

- Constructor con diccionario: `port`
  - `port`: puerto, por defecto 80.
- Método `close()` → cancela la petición.

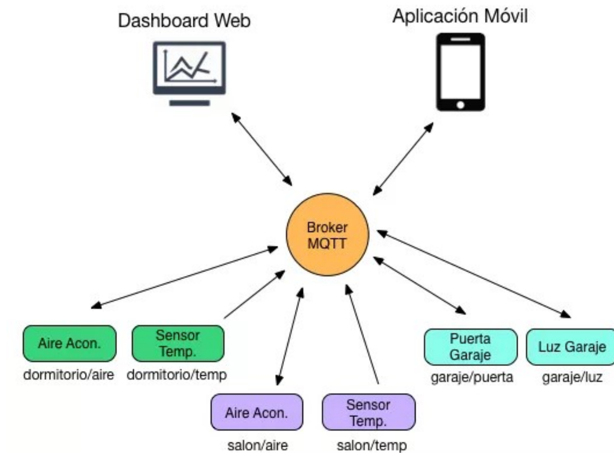
ESP32: Servidor HTTP

- Propiedad callback: función que recibe `(message, val1, val2)`. Según message:
  - error → error
  - connection → nueva conexionón.
  - status → val1 = path pedido, val2 = método (ej. GET, POST, ...)
  - header → recibida nueva cabecera. val1 = nombre, val2 = valor
  - headersComplete → ya están todas las cabeceras. El cb debe indicar como recibir el body de la petción:
    - String /ArrayBuffer → en llamada requestComplete
    - True → en sucesivas llamadas requestFragment
    - False/undefined → ignorar el body
  - perpareResponse → listo para enviar la respuesta: devolver diccionario con
    - status → estado de la respuesta, por defecto 200 (OK)
    - headers → array de las cabeceras (Content-Length, Content-type, etc.)
    - body → cuerpo de la respuesta:
      - String/ArrayBuffer la respuesta completa → se añade automaticamente Content-Length
      - true si se prefiere enviar poco a poco con llamadas responseFragment
  - ResponseComplete → éxito al enviar la respuesta
- `this` apunta a objeto de la petición → se le pueden añadir propiedades necesarias

ESP32: MQTT

MQ Telemetry Transport → MQTT

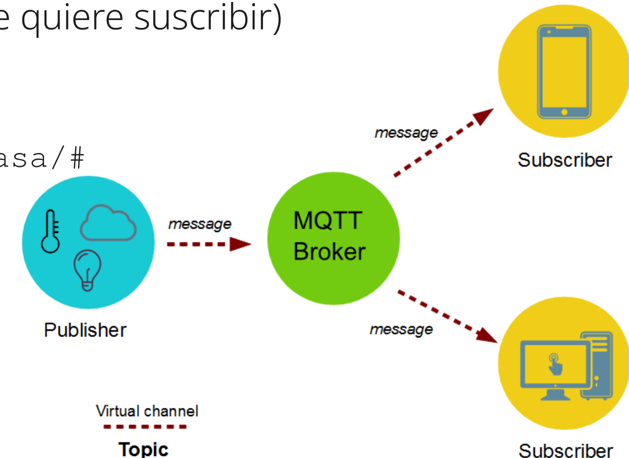
- originariamente → Message Queuing Telemetry Transport
  - diseñado para la telemetría de los oleoductos por IBM
- Estándar OASIS desde 2014
- Protocolo de mensajes ligero basado en publicación y suscripción
  - protocolo más común en IoT: comunicación máquina a máquina
- Versiones:
  - MQTT v3.1.1 – la más extendida (TCP/IP)
  - MQTT v5 – Nueva versión (TCP/IP)
  - MQTT-SN → versión sobre UDP/IP (sin éxito)
- Necesita un Broker (o Server)
  - intermedia entre publicadores (publisher) y subscriptores (subscriber)
  - un dispositivo puedes publicador y subscriptor a la vez.



ESP32: MQTT tópicos

Mensajes organizados en tópicos

- Identificadores organizados en árbol → / es el separador de niveles
  - `casa/plantaBaja/salon/temperatura`
  - `casa/plantaAlta/baño/humedad`
- El publisher debe indicar tópico concreto al que mandar mensaje
- El Server reenvía mensaje a todos los subscribers suscritos a ese tópico
  - susbcribers deben indicar al server los tópicos en los que está interesado (se quiere suscribir)
  - puede usar comodines:
    - + un nivel → `casa/plantaBaja/+/temperatura`
    - # varios los subniveles → `casa/#/temperatura, /casa/plantaAlta/#, /casa/#`
- Tópicos sobre estadísticas de servidor → `$SYS/`
  - `$SYS/broker/clients/connected`
  - `$SYS/broker/messages/sent`
  - `$SYS/broker/uptime`



ESP32: MQTT Herramientas

- **Servidor MQTT software libre → Mosquitto**
  - Última versión utiliza autenticación y cerrado al exterior
    - para abrir en fichero configuración `mosquitto.conf`:
      - `listener 1883`
      - `allow_anonymous true`
- **Clientes en línea de comandos → Mosquitto**
  - subscriber → `mosquitto_sub -v -d -h broker -t prueba/test`
  - publisher → `mosquitto_pub -t -h broker -m "Hola"  -t prueba/test`
- **Cliente para depuración →**
  - MQTT Explorer → gratis pero no SL
  - MQTT X → SL pero más básico

ESP: MQTT moddable

Clase → `import Client from "mqtt";`

  - `"$(MODDABLE)/modules/network/mqtt/manifest.json"`
- Constructor con diccionario:
  - `host, port` → del server
  - `id` → identificador único para el dispositivo
  - `user, password` → si server exige autenticación
  - `timeout` → intervalo del keep-alive
  - `will` → mensaje testamento. Lo enviará el server si cliente se desconecta
  - `path` → si se usa websocket
  - `Socket, secure` → para conexiones seguras usando TLS
- `onReady` → callback que se invoca cuando se ha establecido conexión con server
- `subscribe(topico)` → suscribirse a un tópico
- `unsubscribe(topico)` → desuscribirse del tópico
- `onMessage(topico, datos)` → callback que se invoca al recibir tópico al que se está suscrito
  - `datos` → se recibe en `ArrayBuffer`. Si se quiere convertir a `String` → `String.fromArrayBuffer(mensaje)`
- `publish(topico, mensaje)` → publicar mensaje
  - mensaje puede ser `String` o `ArrayBuffer`
- `onClose` → callback que se invoca si se cierra la comunicación con server.

ESP32: otras clases de red

Otras clases para comunicaciones por Internet

- Socket → base para las otras comunicaciones
- Listener → manejar un servidor TCP
- WebSocket Client y Server → comunicaciones websocket.
- DNS → para peticiones complejas al DNS.
  - Net.Resolve hace las normales nombre a ip de manera transparente
- MDNS → para trabajar con Multicast DNS.
- Telnet → consola remota para dar comandos de texto
- Ping → Para detectar si otro dispositivo en la red está activo.

ESP32: Almacenamiento

Acceso como fichero

- Sistemas de ficheros en la flash o SD → módulo `"files"`
  - Leer o escribir ficheros
  - Recorrer directorio → sistema de archivos de un solo nivel

Preferencias

- Solo pequeña cantidad de datos → preferencias del usuario
  - `import Preference from "preference";`
- `Preference.set(dominio, nombre, valor)`
  - `valor` puede ser booleano, entero, `String` o `ArrayBuffer`
- `Preference.get(dominio, nombre)` → devuelve el valor almacenado
- `Preference.delete(dominio, nombre)` → borra preferencia
- `Preference.keys(dominio)` → todos los nombres definidos en `dominio`

ESP32: Recursos

Para datos de solo lectura: datos, imágenes, ...

- Se declaran en el `manifest.json → resources`
  - poner nombre del fichero a incluir pero sin extensión
- `import Resource from "resource";`
  - `let data = new Resource("mydata.dat");`
  - devuelve `HostBuffer` que es tipo de `ArrayBuffer` de solo lectura.
    - para convertir a `String` → `String.fromArrayBuffer(data)`
    - para otros tipos de datos Arrays con tipo o DataViews

```
"resources": {
    "*": [
        "./mydata"
    ],
},
```