Universidad de La Laguna

Sistemas Embebidos: ESP32

OCW

Introducción a JavaScript para Moddable

ESP32: Moddable JavaScript

JavaScript

- Sintaxis similar a C y C++
- Creado para dinamismo en páginas web.
- Incluye características de P Estructurada, PO Objetos, P Funcional, etc.
- Rápida evolución, compatible hacia atrás pero características desfasadas
- Orientado a eventos: funciones son invocadas cuando se produce evento
  - Muy adecuado para sistemas empotrados.
  - Único hilo de ejecución ⇒ no hay concurrencia
  - Ejecutor de tareas cíclico.
- Moddable trata de seguir JavaScript 2019
  - con ligeras diferencias y uso funcionalidades del 2020.

ESP32: JavaScript Sintaxis

Sintaxis básica:

- Soporte completo de UTF-8
  - cualquier carácter/símbolo en identificadores
- No necesario punto y coma para separa instrucciones
  - pero **muy recomendable**
- Variables no declaran tipo
- Opciones de declaración:
  - `var` → variable visible en todo el fichero, incluso fuera del bloque. No recomendado
  - `let` → variable visible dentro del bloque. Su valor puede cambiar
  - `const` → variable visible dentro del bloque. Su valor NO puede cambiar. Preferible
    - puede cambiar el objeto al que *apunta* la variable.
- Estructuras de control igual que en C
  - `if-else`, `switch`, `while`, `for`, `do-while`.
    - añade bucles `for` más cómodos

ESP32: JavaScript Tipos de datos

- Tipos básicos:
  - `Boolean`: valores `true` y `false`
  - `Number`: números en punto flotante de 64 bits
    - Optimizaciones para enteros
    - prefijo `0x` para constantes hexadecimales, `0b` para las binarias.
    - definidos el `Infinity` y el `NaN`.
    - Objeto `Math` contiene constantes y métodos matemáticos (`PI`, `sqrt`, `cos`, `min`, ...)
  - `String`: clase similar a `std::string` de C++
  - `Array`: lista ordenada de items de cualquier tipo
- Valor especial `undefined` cuando algo no definido
  - Variable declarada pero no inicializada → contiene `undefined`
  - Parámetro de función no indicado en la llamada → contiene `undefined`
  - Propiedad que no existe en un objeto → devuelve `undefinde`

ESP32: JavaScritp Operadores

Mismos operadores que en C

- Asignación: =
- Números: +, − , /, %
- Manipulación de bits: ~, &, |, ^, >>, <<, >>> (desplaza derecha sin signo → 0s)
- Comparaciones:
  - == convierte a enteros los operandos antes de comparar
    - 1 == "1" → true, "0" == false → true, 1 == true → true, 2 == true → false, Infinity == "Infinity" → true
  - === comparación estricta, sin cambio de tipos
    - 1 === "1" → false, "0" === false → false, 1 === true → false, Infinity === "Infinity" → false

ESP32: JavaScritp Strings

- Similares a las de C++ con algunas mejoras:
  - definición con comillas simples ('), dobles (")
  - comillas hacia atrás (`) → *template literals*
    - múltiples líneas
    - sustitución de expresiones usando `${}`: `` `La suma de a + b es ${a + b}` ``
  - longitud con atributo: `s.length`
  - son inmutables → no se pueden modificar su contenido.
  - obtener subcadenas → método `slice(inicial[,final+1])`:
    - `let a="hello, world!"`
      - `a.slice(0, 5)` → `"hello"`
      - `a.slice(7, 12)` → `"world"`
      - `a.slice(7)` → "world!"
  - borrar espacios al principio/final → métodos `trim(), trimStart(), trimEnd()`
  - buscar subcadenas → métodos `indexOf(buscado), lastIndexOf(buscado), startsWith(buscado), endsWith(buscado)`
  - repetir el string → método `repeat(cuantas)`

ESP32: JavaScript arrays

- Clase `Array` contiene lista ordenada de items de cualquier tipo
  - Se puede declarar con corchetes: `let a = [1, "Hola", true, {x: 3, y: undefined}];`
  - Se puede instanciar como objeto `let vacio = new Array();`
  - Conocer el número de elemento con propiedad `length` → `a.length`
  - Se indexa con corchetes: `a[0], a[1]`
    - Se devuelve `undefined` si se accede índice no existe: `a[100]` → `undefined`
  - Recorrer array:
    - `for (let i = 0; i < a.length; i++)` → accederemos con `a[i]`
    - `for (let value of a)` → `value` irá teniendo los valores
    - `for (let i in a)` → `i` irá teniendo los índice, accederemos con `a[i]`
    - `a.forEach(value => trace(value))` → ejecuta función indicada para cada elemento
    - `a.map(value => value * 2)` → devuelve array del mismo tamaño con resultado de aplicar función a cada elemento
    - `a.reduce((acumulado, actual, indice) => acumulado + actual * indice, inicial)` → devuelve acumulado final
  - Añadir elementos: `a.push(elem)` → añade al final; `a.unshift(elem)` → añade el principio
  - Eliminar elementos (y devuelve eliminado): `a.pop()` → elimina último; `a.shift()` → elimina el primero
  - Extraer sub-arrays con `slice()` como los Strings → devuelve nuevo array: `let b = a.slice(0,2)`
    - `splice()` extrae y borra lo extraído
  - Ordenación como strings `a.sort()`.
    - Se le puede pasar función para indicar criterio: `a.sort((x, y) => x.length - y.length)`
  - Búsqueda con métodos `indexOf(buscado)`, `lastIndexOf(buscado)`, `a.findIndex(v => v%2)`

ESP32: JavaScritp Objetos

JS es orientado a objetos pero numerosas diferencias con C++

- Mayoría de objetos sin clase: `let a = {one: 1, two: "two", object: {}, };`
  - se usan como diccionarios
- Los atributos se denominan propiedades
  - Añadir propiedades a objeto existente: `a.otra = [1, 2, "casa",];`
  - Borrar propiedades: `delete a.object;`
  - Acceder propiedades por corchetes: `a["one"]` → 1
  - Propiedades que no existen devuelven `undefined`: `a.noExiste == undefined`
    - pero puede existir y contener `undefined`: `let a = {missing: undefined};`
    - mejor usar `in`: `if ("noExiste" in a)`
- `const` al declarar objeto ⇒ indentificador siempre *apunta* al mismo objeto
  - pero objeto puede cambiar
    - `const a = { b: 1 }`
    - `a = 3` → error
    - `a.b = 2` → correcto
    - `a.c = 5` → correcto

ESP32: JavaScritp Objetos

- Usar Object.freeze(objeto) convierte objeto en solo lectura
  - más eficiente → almacenable en ROM
- Clases se pueden definir como en C++
  - Palabra reservada class
  - constructor de la clase → constructor() { }
  - propiedades y métodos se acceden con this.propiedad this.metodo()
  - propiedades privadas: comienzan por # y declaradas en cuerpo de la clase.
  - se instancian objetos de la clase con
    - new Clase; → sin parámetros al constructor
    - new Clase(p1, p2) → parámetros para el constructor
- Definir herencia con `class Hija extends Madre {}`
  - Invocar constructor de clase Madre con `super()`
  - Clase `Object` ancestro de todas las clases/objetos

```
class Hija extends Madre {
  #privado1;
  #privado2 = 3.1;

  constructor(p1, p2) {
    this.atrib3 = "Clase";
    …
  }

  metodo1 (p1) {
    this.atrb4 =this.#privado2
                * this.atrib3;
    …
  }

  #metPrivado() {
    …
  }
}
```

ESP32: JavaScritp Funciones

- Palabra reservada `function`: `function suma(a, b) { return a+b; }`
  - No hay que indicar tipo de salida ni de parámetros
- `return` para devolver algo
  - caso contrario devuelve `undefined`
- Parámetros de entrada
  - siempre por valor
    - pero si es objeto se puede modificar sus propiedades (se pasan los punteros)
  - Parámetros no pasados en la invocación se les asigna valor `undefined`
  - Posibilidad de parámetros por defecto
  - Número variable parámetros de entrada con `...resto` → `resto` array con parámetros.
- Posibilidad de funciones anónimas (lambda)
  - Definidas para se pasadas como parámetros a otras funciones
    - MUY, MUY, MUY usado en JavaScript

ESP32: JavaScript clausuras

Herramienta fundamental de JavaScript

- Funciones pueden acceder a variables locales del bloque donde estén definidas
  - Aunque el bloque/función contenedora **haya terminado de ejecutarse**.
  - MUY, MUY, MUY usado en JavaScript

```
function makeCounter(){
 let value = 0;
 return function() {
  value += 1;
  return value;
 }
}
```

```
let counterOne = makeCounter();
let counterTwo = makeCounter();
let a = counterOne(); // 1
let b = counterOne(); // 2
let c = counterTwo(); // 1
let d = counterTwo(); // 2
let e = counterOne(); // 3
let f = counterTwo(); // 3
```

ESP32: JavaScript módulos

Módulos → librerías de código

- **Creación de módulo**
  - Un fichero que define variables, funciones, clases y código a ejecutar
  - De todo lo definido se puede exportar
    - una sola cosa → `export default ClaseDefinida;`
    - varias cosas → `export {ClaseDefinida, FuncionDefinida, ConstanteDefinida};`
- **Uso de módulo → importación**
  - Con palabra reservada `import`
    - `import Nombre from "fichero"`
      - en `Nombre` el `default export` del módulo definido en `"fichero"`
    - `import {ClaseDefinida, ConstanteDefinida} from "fichero"`
      - solo algunas de las cosas exportadas por el módulo
    - `import {ClaseDefinida as Clase1, ConstanteDefinida as CTE} from "fichero"`
      - como el anterior pero cambiando el nombre de lo importado
    - `import * as Mod1 from "fichero"`
      - se importa todo pero hay que accederlo con `Mod1.FuncionDefinida()`
  - El código del módulo se ejecuta al ser importado → antes de comenzar modulo importador

ESP32: JavaScript excepciones

Para indicar errores

- Existe clase Error: se instancia objeto de esa clase y se lanza
  - if(valor < 0) throw new Error("No se admiten negativos")
  - Otras subclases más específicas: RangeError, TypeError y ReferenceError

Para detectar los errores

- construcción `try – catch – finally`:

```
try {
  instrucciones pueden lanzar excepción
} catch (e) {
  que hacer cuando salta excepción
  e contiene lo lanzado
} finally {
  se ejecuta siempre
  se capture o no la excepción
}
```

ESP32: JavaScript ArrayBuffer

- Clase `ArrayBuffer` (AB)permite manejo eficiente de datos binarios
  - Constructor numero de bytes indicados: `let ab = new ArrayBuffer(10);`
  - propiedad `byteLength` → número de bytes. No se puede cambiar
  - método `slice()` → obtener subarrays en nuevos ArrayBuffer
  - NO se pueden indexar para sacar su contenido → necesarios los Typed Arrays.
- Los Typed Arrays (ta) permiten ver contenido de `ArrayBuffer` como ciertos tipos de datos:
  - tipos disponibles: `Int8Array, Uint8Array, Uint16Array, Float32Array, Float64Array, Uint32Array,` etc.
    - `let ta = new Uint8Array(ab)`
    - Se puede crear ta sobre una parte del BA indicando bytes desplazamiento y nº elementos → `let ta = new Int16Array(ab, 5, 3)`
  - Se pueden indexar sus elementos: `ta[0] = 12; ta[1] = ta[2] + 8;`
  - Tienen propiedad `length` indicando número de elementos (según tamaño del tipo)
  - Es posible tener diferentes ta sobre el mismo BA → peligroso
  - Posible crear directamente un ta dando sus elementos con `of`: `let ta = Uint16Array.of(0, 1, 2, 3);`
    - se crea automáticamente el AB necesario. Se accede al mismo con `let ab = ta.buffer;`
  - `ta.copyWithin(destino, ini, fin)` copian dentro del propio `ta` desde `ini` hasta `fin` (sin corgerlo) en `destino`
  - `ta.set(ta2, destino)` escribe el contenido del `ta2` en `ta` a partir del índice `destino` (hasta que se acabe `ta2`)
  - `ta.subarray(ini, fin)` crea *vista* del array solo con los elementos entre `ini` y `fin`. No crea nuevo array como `slice()`
  - `ta.fill(valor, ini, fin)` rellena elementos desde `ini` a `fin` con `valor`.

ESP32: data views

Los DataView (DV) permiten acceso a ArrayBuffer con distintos tipos

- Constructor con todo AB o indicando rango: `let db = new DataView(ab, ini, fin);`
- métodos `setXX()` de los distintos tipos con indice y valor para escribir DV
  - `db.setUint16(ind, valor)  db.setInt8(ind, valor) db.setUint32(ind, valor)`
  - se guardan en big-endian, si añadimos tercer parámetro a `true` → se usará little-endian
    - `db.setUint32(4, 0x11223344, true)`
- métodos `getXX()` de los distintos tipos con indice para leer DV
  - `dv.getInt8(ind) db.getInt32(ind, true)`