Programación en C para Sistemas Embebidos

Guía de Estilo

Esta guía proporciona un conjunto de convenciones y buenas prácticas para escribir código en C claro, consistente y mantenible. Está inspirada en la ["Google C++ Style Guide"](https://google.github.io/styleguide/cppguide.html) pero modificada para alinearse con prácticas comunes de C y los requisitos específicos para la arquitectura ARM.

# Nomenclatura

La nomenclatura busca evitar colisiones de nombres en ausencia de namespaces y alinearse con el estilo tradicional de C.

### Prefijos

Todos los símbolos en un archivo de cabecera (.h) deben compartir un **prefijo único y compartido**. Esto actúa como un namespace.

// En el archivo teclado\_4x4.h

// El prefijo es "tec4x4"

#define TEC4X4\_PUERTO\_FILAS LPC\_GPIO5

static const char tec4x4\_matriz\_teclas\_arriba[4][4]**;**

void tec4x4\_inicializar(uint8\_t tipo);

## Archivos

Los nombres de los archivos siempre serán en **sin caracteres especiales y en snake\_case** (minúsculas unidas por guiones bajos).

## Variables

Utiliza nombres descriptivos y en **snake\_case** (minúsculas unidas por guiones bajos). Evita abreviaturas a menos que sean muy comunes (como i para un índice de bucle).

int numero\_de\_elementos**;**

char**\*** nombre\_usuario**;**

## Funciones

Utiliza nombres descriptivos de su acción y en **snake\_case**.

int calcular\_suma**(**int a**,** int b**);**

void imprimir\_reporte**(**const char**\*** reporte**);**

## Enumerados

Utiliza nombres descriptivos y en **snake\_case**. Si se define un tipo se terminará en **\_t**.

### Constantes de una enumeración (enum)

Los nombres de las constantes de una enumeración deben llevar como prefijo el nombre de su tipo de enumeración para evitar colisiones. Se recomienda usar como una macro en **MAYUSCULAS\_CON\_GUIONES\_BAJOS** o **k seguido de PascalCase** para las constantes.

// Los valores de un Enum pueden verse como una macro

**typedef** enum **{**

PIEZA\_PALO**,**

PIEZA\_CUADRADO**,**

PIEZA\_ESE**,**

PIEZA\_ZETA**,**

PIEZA\_ELE**,**

PIEZA\_ELEREVES**,**

PIEZA\_TE

**}** pieza\_tipo\_t**;**

// o como constantes

**typedef** enum **{**

kPiezaPalo**,**

kPiezaCuadrado**,**

kPiezaEse**,**

kPiezaZeta**,**

kPiezaEle**,**

kPiezaEleReves**,**

kPiezaTe

**}** pieza\_tipo\_t**;**

## Estructuras

Utiliza nombres descriptivos y en **snake\_case**. Si se define un tipo se terminará en **\_t**.

**typedef** struct **{**

pieza\_tipo\_t tipo**;**

int32\_t tamano**;**

int8\_t matriz**[**PIEZA\_MAX\_TAM\_MATRIZ**][**PIEZA\_MAX\_TAM\_MATRIZ**];**

int32\_t color**;**

**}** pieza\_t**;**

## Macros

Escribe los nombres en **MAYUSCULAS\_CON\_GUIONES\_BAJOS**. Esto ayuda a distinguirlas de las variables y funciones en tiempo de ejecución.

#define MAX\_CONEXIONES 100

#define PI 3.14159

Si la macro no representa únicamente un literal, enciérralo entre paréntesis

#define PIN3 (1u << 3)

# Declaraciones y Definiciones

## Constantes

El manejo de constantes y macros en C requiere una atención especial. **Se prefiere usar enum** para definir constantes con nombre en lugar de macros o const int.

* Aparecen en la tabla de símbolos, lo que facilita la depuración.
* Pueden usarse como etiquetas case en una instrucción switch.
* No ocupan memoria.
* **Excepción:** Si la constante se va a utilizar en código ensamblador, una macro de preprocesador (#define) es la mejor opción.

**typedef** enum estado\_proceso **{**

INICIADO**,**

EN\_EJECUCION**,**

FINALIZADO

**}** estado\_proceso**;**

## Macros

A diferencia de C++, el uso de macros es una práctica razonable en C. Se permite exportar macros como parte de una API pública (por ejemplo, para definiciones de registros).

// Registros del LM75B

#define LM75B\_REG\_TEMP 0

#define LM75B\_REG\_CONF 1

#define LM75B\_REG\_THYST 2

### **Macros tipo Función**

Deben evitarse. Si son necesarias, deben ser "higiénicas", es decir:

* Encierra los argumentos de la macro entre paréntesis ().
* **No** termines la macro con un punto y coma ;
* Las macros pueden ocupar más de una línea utilizando **\** para indicarlo.
  + **Si la macro devuelve un valor**, envuélvela en un bloque ({…}).

#define CEIL(x) \

({ \

\_\_typeof\_\_(x) \_x = (x); \

/\* La última expresión es el valor de retorno de la macro \*/ \

(\_x > 0) ? (long long)(\_x + 0.9999999999999999) : (long long)\_x; \

})

* + En caso contrario, envuélvela en un bloque do {...} while (0).

#define ASSERT(expr, mensaje) \

do { \

if (expr) { \

; \

} else { \

parar\_con\_error(\_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_, mensaje); \

} \

} while (0)

## Funciones

* Los nombres de los parámetros en la declaración deben coincidir con los de la definición.
* Para una función sin argumentos, usa void como parámetro

bool\_t mi\_funcion**(**void**);**

### Funciones Inline

Al declarar una función como inline, puedes indicarle al compilador que haga que las llamadas a esa función sean más rápidas. Una forma en que el compilador puede lograr esto es integrando ("copiando") el código de esa función en el código de sus llamadores. Esto hace que la ejecución sea más rápida al eliminar la sobrecarga de la llamada a la función; además, si alguno de los valores de los argumentos reales es constante, sus valores conocidos pueden permitir simplificaciones durante el tiempo de compilación para que no sea necesario incluir todo el código de la función inline.

La definición coninline **va en el .h** y se recomienda que exista una única declaración extern en el fichero .c correspondiente**.**

**En funcion\_inline.h**

inline uint32\_t funcion\_inline**(**uint8\_t param1**)** **{**

// Implementación

**}**

**En funcion\_inline.c**

#include "funcion\_inline.h"

extern uint32\_t funcion\_inline**(**uint8\_t param1**);**

## Declaraciones Static

Las declaraciones marcadas como static no deben aparecer en los archivos de cabecera (.h). Los archivos de cabecera son declaraciones de interfaces públicas, y las definiciones static se copian, no se comparten, entre las unidades de compilación, lo que puede generar errores. Las funciones marcadas como static no deben ser marcadas como inline. El compilador es capaz de incluir funciones estáticas en línea sin la anotación inline.

## Punteros

Al declarar punteros, el asterisco (\*) debe colocarse junto al nombre de la variable, no al tipo.

int **\***ptr**;** // Correcto

int**\*** ptr**;** // Incorrecto

## Inicialización de estructuras

Al inicializar un struct o union, los inicializadores deben ser designados, es decir, indicados explícitamente.

pieza\_t pieza\_palo **=** **{**

**.**tipo **=** PIEZA\_PALO**,**

**.**tamano **=** 4**,**

**.**matriz **=** **{{**0**,** 1**,** 0 **,**0**},**

**{**0**,** 1**,** 0**,** 0**},**

**{**0**,** 1**,** 0**,** 0**},**

**{**0**,** 1**,** 0**,** 0**}},**

**.**color **=** ROJO

**};**

# Archivos y Directivas del Preprocesador

## Archivos de Cabecera

Deben tener guardas de inclusión con un nombre único basado en la ruta del proyecto.

// Guarda del archivo con ruta: mi\_proyecto/modulo/mi\_cabecera.h

#ifndef MI\_PROYECTO\_MODULO\_MI\_CABECERA\_H\_

#define MI\_PROYECTO\_MODULO\_MI\_CABECERA\_H\_

// ...

#endif // MI\_PROYECTO\_MODULO\_MI\_CABECERA\_H\_

## Directivas #include

En los archivos de cabecera se incluirán aquellos archivos que sean necesarios para que este funcione. No se debe incluir ningún otro archive de cabecera bajo ningún otro concepto. Al realizar las inclusiones deben indicarse siempre desde el directorio base del programa.

No contemplar las inclusiones heredadas. Por ejemplo, si mi archivo sonido.c requiere de la librería gpio\_lpc40xx.h, se debe incluir dicha librería en sonido.c aun cuando esta ya se haya incluido en sonido.h.

Las inclusiones deberán indicarse en el siguiente orden:

1. Si estamos implementando una librería (.c), inclusión del archivo .h con el mismo nombre
2. Librerías y cabeceras globales del microcontrolador (<LPC407x\_8x\_177x\_8x.h >)
3. Librerías estándar de C (<stdio.h>)
4. Librerías y cabeceras propias de la asignatura de la más general a la más específica ("tipos.h"/ "gpio\_lpc40xx.h"/ "timer\_lpc40xx.h"/ "i2c\_lpc40xx.h")

#include "joystick.h" // Archivo de cabecera de la librería

#include <LPC407x\_8x\_177x\_8x.h> // Librería principal del μc

#include "tipos.h" // Librería propia más general

#include "gpio\_lpc40xx.h" // Librería propia

#include "leds.h" // Librería propia más específica

# Consideraciones Específicas para ARM (C)

Al programar en C para la arquitectura ARM, aplican todas las reglas anteriores, con las siguientes adiciones cruciales:

## Conformidad con AAPCS

El código debe adherirse al **"Procedure Call Standard for the Arm Architecture"**. Esto define cómo se pasan los argumentos (ej. los primeros 4 en **R0-R3**), cómo se devuelven los valores y qué registros debe preservar una función.

## Tipos de Datos de Ancho Fijo

**Usa siempre los tipos definidos en <stdint.h>** o "tipos.h" (como uint32\_t, int16\_t, uint8\_t) en lugar de int, char o long. Esto garantiza la portabilidad y que el tamaño de las variables sea exactamente el que esperas en la arquitectura ARM.

**typedef** signed char int8\_t**;**

**typedef** signed short int **(**int**)** int16\_t**;**

**typedef** signed int **(**long int**)** int32\_t**;**

**typedef** signed long long int int64\_t**;**

**typedef** unsigned char uint8\_t**;**

**typedef** unsigned short int **(**int**)** uint16\_t**;**

**typedef** unsigned int **(**long int**)** uint32\_t**;**

**typedef** unsigned long long int uint64\_t**;**

**typedef** float float32\_t**;** // IEEE-754 de simple precisión

**typedef** double float64\_t**;** // IEEE-754 de doble precisión

## Alineación de Datos

El acceso a datos no alineados en ARM puede causar un **HardFault** o degradar el rendimiento. Asegúrate de que las estructuras de datos estén correctamente alineadas. Si es necesario, fuerza la alineación con \_\_attribute\_\_((aligned(X))), siendo X el tamaño del bloque de memoria del microcontrolador.

## Calificador *volatile*

Es **imprescindible** usar volatile al acceder a registros de periféricos mapeados en memoria (MMIO – Memory Mapped Input Output). Esto prohíbe al compilador optimizar (reordenar, agrupar o eliminar) las lecturas y escrituras a esa dirección de memoria, garantizando que cada acceso al hardware ocurra tal y como está escrito en el código.

volatile uint32\_t **\***uart\_status\_reg **=** **(**uint32\_t **\*)**0x40001000**;**

// La lectura de este registro siempre se realizará sin optimizar.

**while** **((\***uart\_status\_reg **&** **(**1 **<<** 5**)))** **{**

; // Espera a que el bit de "ocupado" se ponga a cero.

**}**

**En core\_cm4.h**

#define \_\_I volatile const /\*!< Defines 'read only' permissions \*/

#define \_\_O volatile /\*!< Defines 'write only' permissions \*/

#define \_\_IO volatile /\*!< Defines 'read / write' permissions \*/

**En LPC407x\_8x\_177x\_8x.h**

//------- General Purpose Input/Output (GPIO) --------------------

**typedef** struct **{**

\_\_IO uint32\_t DIR**;**

uint32\_t RESERVED0**[**3**];**

\_\_IO uint32\_t MASK**;**

\_\_IO uint32\_t PIN**;**

\_\_IO uint32\_t SET**;**

\_\_O uint32\_t CLR**;**

**}** LPC\_GPIO\_TypeDef**;**

# Formato y Estructura

Un formato consistente es clave para la legibilidad.

* **Llaves ({}):** Usa el estilo "K&R". La llave de apertura ({) se coloca en la misma línea que la sentencia de la estructura de control o la declaración de la función.

**if** **(**condicion**)** **{**

// Código aquí

**} else {**

// Otro código

**}**

* **Bloques Obligatorios:** **Todos** los bucles y condicionales **deben usar llaves**, incluso si el cuerpo es una sola línea.

|  |  |
| --- | --- |
| // Correcto  **if** **(**foo**)** **{**  do\_something**();**  **}** | // Incorrecto  **if** **(**foo**)**  do\_something**();** |

* **Bucles Infinitos:** Prefiere while (true) o while(1) en lugar de for (;;).
* **Identación:** Usa **2 o 4 espacios**. Sé consistente. **No uses tabuladores**. El código del profesor se encuentra identado normalmente en 2 espacios.
* **Longitud de Línea:** Máximo de **80 o 100 caracteres**. Esto es configurable en la mayoría de los entornos de desarrollo y editores de código.

En μVision se pueden configurar estas opciones en el menú "Edit→Configuration" y configurar los apartados como la Figura 1.  
Además, es posible añadir palabras clave personalizadas como los tipo estándar o propios del LPC en la pestaña "User Keywords" como en la Figura 2.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Figura – Configuración de la longitud de línea e identación y tabulaciones como espacios en μVision

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Figura – Palabras clave personalizadas para C incluyendo los tipos estándar de C y algunos propios del LPC4088.

## Espacio horizontal

1. Usa un espacio alrededor de los operadores binarios (=, +, -, \*, /, ==, !=, etc.). Se permite eliminarlos en los factores de una multiplicación o división. No utilizar espacios al abrir o cerrar los paréntesis
2. No pongas espacios entre el nombre de una función y su paréntesis de apertura
3. Usa un espacio después de las comas en las listas de argumentos
4. Agregar 1 espacio antes de una apertura de llave ({)
5. Dos espacios antes de un comentario de final de línea
6. Dentro del bucle for, siempre agregar un espacio tras ;
7. En una estructura switch, no se agrega un espacio antes de :

int resultado **=** **(**a + b**) \*** c; // (1.)

v **=** w **\*** x **+** y **/** z**;** // (1.)

v **=** w**\***x **+** y**/**z**;** // (1.)

mi\_funcion**(**bool\_t arg1**,** bool\_t arg2**);** // (2., 3.)

mi\_funcion**(**bool\_t arg1**,** bool\_t arg2**)** **{** // (4.)

a **=** 0**;** // (5.)

**for** **(**i **=** 0**;** i **<** 5**;** i**++)** **{** // (6.)

// ...

**}**

**switch** **(**i**)** **{** // (7.)

**case** 1**:**

// ...

**}**

## Espacio vertical

Utiliza una (o varias) líneas en blanco entre funciones, estructuras, etc. así como entre secciones de código dentro de una función (similar a como utilizas los párrafos normalmente).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* **\brief** Leer el valor del registro TC de un timer.

\* **\param[in]** timer\_regs puntero al bloque de registros del timer.

\* **\return** Valor actual del registro TC de un timer.

\*/

uint32\_t timer\_leer**(**LPC\_TIM\_TypeDef **\***timer\_regs**){**

ASSERT**(**timer\_regs **==** TIMER0 **||** timer\_regs **==** TIMER1 **||**

timer\_regs **==** TIMER2 **||** timer\_regs **==** TIMER3**,**

"timer\_regs incorrecto"**);**

**return** timer\_regs**->**TC**;**

**}**

# Guía de Estilo para Ensamblador ARM

Si necesitas escribir código en ensamblador, sigue estas directrices para mantener la claridad.

* **Formato de Cuatro Columnas**
  + **Etiquetas:** Al inicio de la línea, terminadas en dos puntos (:)
  + **Mnemónicos:** Indentados 4 espacios (ej. MOV, LDR)
  + **Operandos:** Los registros y valores
  + **Comentarios:** Empiezan con ; o @ y explican el *porqué* de la instrucción
* **Nomenclatura de Registros:** Usa los nombres simbólicos definidos en la AAPCS (sp, lr, pc, etc.) para mejorar la legibilidad.
* **Comentarios de Bloque:** Antes de una función en ensamblador, incluye un comentario de cabecera que describa su propósito, los registros de entrada/salida y los que se modifican.

esperar\_bandera**:**

**LDR** **r1,** **[r0,** #0x04**]** ;@ Carga el registro de estado desde la base R0

**TST** **r1,** #0x10 ;@ Comprueba si el bit 4 está activo

**BEQ** esperar\_bandera ;@ Si es cero, vuelve a esperar

# Documentación – Doxygen

Para la escritura de la documentación del código realizado se utilizará la herramienta [Doxygen](https://www.doxygen.nl/index.html) por ser ampliamente utilizada en el desarrollo de aplicaciones en C, C++ o Python entre otros.

Doxygen es una herramienta de generación de documentación ampliamente utilizada en el desarrollo de software. Automatiza la generación de documentación a partir de comentarios en el código fuente, analizando información sobre clases, funciones y variables para producir resultados en formatos como HTML y PDF. Al simplificar y estandarizar el proceso de documentación, Doxygen mejora la colaboración y el mantenimiento en diversos lenguajes de programación y escalas de proyectos.

Además, nos permite generar diagramas mediante código lo que puede facilitar la generación de diagramas de flujo o relación entre clases.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Figura – Ejemplo de diagrama de flujo generado por Doxygen

## Comentarios Generales

Los comentarios deben aclarar la intención, no parafrasear el código.

Usa el estilo de C99 (//) para comentarios que no son de documentación. Las variables mencionadas en los comentarios deben delimitarse con acentos graves (`).

// `ptr` nunca puede ser NULL por esta razón.

## Licencia del código

Comienza cada archivo con el texto estándar de la licencia que utilices en tu programa.

### Ejemplo (GNU v3)

/\*\*

\* <Descripción del programa y un resumen de lo que realiza>

\* Copyright (C) <año> <nombre del autor/es>

\*

\* This program is free software: you can redistribute it and/or

\* modify it under the terms of the GNU General Public License as

\* published by the Free Software Foundation, either version 3 of

\* the License, or (at your option) any later version.

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\* You should have received a copy of the GNU General Public License

\* along with this program. If not, see https://www.gnu.org/licenses/

 \*/

## Comentarios de Documentación

Usa bloques /\*\* ... \*/ para documentar el código tanto en los archivos de cabecera como en los de implementación.

* La primera línea es un resumen corto.
* Después de una línea en blanco, puede seguir una descripción más detallada.
* Usa @ o \ para introducir un comendo especial ([enlace](https://www.doxygen.nl/manual/commands.html)).
* Puedes usar Markdown para el formato y #nombre\_funcion para referencias cruzadas.

/\*\*

\* Datos de arranque almacenados en la partición de información

\* del flash.

\*/

**typedef** struct boot\_data **{**

/\*\*

\* Resumen SHA-256 de los datos de arranque.

\*

\* La región cubierta por este resumen comienza inmediatamente

\* después de este campo y termina al final de la entrada.

\*/

hmac\_digest\_t digest**;**

/\*\* Identificador de los datos de arranque. \*/

uint32\_t identifier**;**

**}** boot\_data\_t**;**

## Comentarios de archivo

A nivel de archivo, la documentación principal es un bloque de comentario en la cabecera que identifica el fichero y resume su propósito general. Sirve como la "portada" del módulo, permitiendo a cualquiera entender su contenido sin necesidad de leer el código.

Es fundamental incluir este bloque al principio de cada fichero (.c o .h) para describir el propósito general del archivo. **Qué incluir:**

**Apartados Esenciales**

* **@file:** Identifica el nombre del fichero. Es la etiqueta principal que le dice a Doxygen que este comentario describe al archivo completo.
* **@brief:** Un resumen corto y directo del propósito del archivo. Responde a la pregunta: "¿Qué hace el código de este fichero?".

**Apartados Recomendables**

* **@author:** El nombre y, opcionalmente, el correo electrónico del autor o mantenedor principal. Es útil para saber a quién contactar con dudas.
* **@version:** La versión actual del fichero, que puede seguir la del proyecto o ser independiente.
* **@date:** La fecha de creación o de la última modificación significativa.
* **@details** (Opcional): Si el resumen (@brief) no es suficiente, aquí puedes añadir una descripción más detallada sobre la arquitectura, las dependencias o el funcionamiento del módulo.
* **@note** (Opcional): Para añadir notas importantes sobre el fichero, como dependencias de hardware específicas o la necesidad de configurar ciertos defines.
* **@copyright:** Una breve nota de copyright.

### Ejemplo completo para la cabecera de un Archivo .h

/\*\*

\* **@file** DS3231M.cpp

\* **@brief** Define la estructura básica de la clase DS3231M.

\* **@details** Este archivo implementa las funcionalidades para

\* el módulo de reloj en tiempo real (RTC) DS3231M. Proporciona

\* métodos para leer y establecer la hora y fecha, gestionar

\* alarmas y otras configuraciones del RTC.

\* - Datasheet del DS3231M: https://dfimg.dfrobot.com/nobody/wiki/

\* 456426e32d698477163ee658755f4d05.pdf

\*

\* **@author** [ALD-DSL/ATARI\_RESEARCH\_LAB]

\* **@date** [2025-07-22]

\* **@version** 1.0

\*

\* **@copyright** GNU General Public License version 3 or later

\*

\* **@note** Este módulo se basa en la librería de DFRobot para el DS3231M

\* https://github.com/DFRobot/DFRobot\_DS3231M

\* Copyright 2010 DFRobot Co.Ltd

\* Licencia: Licencia MIT

\*/

## Comentarios de Funciones

La mejor práctica es poner el bloque de documentación completo en el fichero de cabecera (.h), junto a la declaración de la función. El fichero .h define la API pública (la "interfaz"), y es lo que otros desarrolladores consultarán para saber cómo usar tu función. **Que incluir:**

* **@brief**: Un resumen corto y directo de una sola línea.
* **@details**: (Opcional) Una explicación más larga si la función es compleja.
* **@param[in]**: Describe un parámetro que la función solo **lee** (entrada).
* **@param[out]**: Describe un parámetro que la función **escribe** para devolver un valor (salida).
* **@param[in,out]**: Describe un parámetro que la función **lee y también modifica**.
* **@return** o **@retval**: Describe el valor que devuelve la función. Se prefiere **@retval** para documentar diferentes códigos de error.
* **@note** o **@warning**: Para añadir notas importantes o advertencias.

### Ejemplo de cabecera de función en fichero .h

/\*\*

\* **@brief** Lee el valor crudo de un sensor y lo convierte a unidades calibradas.

\*

\* **@details** Esta función accede al hardware a través del objeto

\* sensor, lee el valor ADC, y luego le aplica los factores de escala

\* y offset definidos en la calibración del sensor.

\*

\* **@param[in]** sensor Ptr. al objeto sensor a leer. No puede ser NULL.

\* **@param[out]** output\_value Ptr. a una variable float donde se

\* almacenará el resultado final ya calibrado.

\*

\* **@retval** 0 Si la lectura y calibración fueron exitosas.

\* **@retval** -1 Si el puntero 'sensor' o 'output\_value' es NULL.

\* **@retval** -2 Si el sensor no respondió (error o timeout).

\*

\* **@note** Esta función puede tardar hasta 5 ms en completarse debido al

\* tiempo de conversión del sensor.

\*/

int32\_t sensor\_read\_calibrated**(**const sensor\_t **\***sensor**,**

float32\_t **\***output\_value**);**

## Macros

Para las macros, es crucial documentar qué hacen, sus parámetros (si los tienen) y, muy importante, cualquier efecto secundario como la doble evaluación de argumentos.

* **@def (Opcional):** Nombre de la macro.
* **@brief**: Describe la funcionalidad de la macro.
* **@param**: Documenta cada parámetro en macros tipo función.
* **@return**: Explica el valor al que se expande la macro, si aplica.
* **@warning**: Advierte sobre cualquier comportamiento inesperado o peligroso.

### Ejemplo

/\*\*

\* **@def** CONEXIONES\_MAX

\* **@brief** Valor máximo de conexiones simultáneas permitidas.

\*/

#define CONEXIONES\_MAX 16

#define GLCD\_TAMANO\_X 480u //!< Pixeles horizontales GLCD

/\*\*

\* **@brief** Calcula el cuadrado de un número.

\* **@param[in]** x El número a elevar al cuadrado.

\* **@return** El valor de x \* x.

\* **@warning** El argumento 'x' se evalúa dos veces. No uses

\* expresiones con efectos secundarios como `SQUARE(i++)`.

\*/

#define SQUARE(x) ((x) \* (x))

## Tipos (typedef)

La documentación de un typedef debe aclarar qué representa el nuevo tipo de dato, especialmente si se trata de tipos complejos como punteros a función.

* **@brief**: Explica el propósito o el significado del nuevo tipo.

### Ejemplo

/\*\*

\* **@brief** Códigos de estado para operaciones de dispositivo.

\*/

**typedef** enum **{**

DEVICE\_OK **=** 0**,** //!< La operación se completó con éxito

DEVICE\_BUSY**,** //!< El dispositivo está ocupado

DEVICE\_ERROR**,** //!< Ocurrió un error no especificado

DEVICE\_TIMEOUT**,** //!< La op. no finalizó en el tiempo esperado

**}** device\_status\_t**;**

/\*\*

\* **@brief** Almacena la configuración de un puerto serie (UART).

\*/

**typedef** struct **{**

uint32\_t baud\_rate**;** //!< Velocidad de transmisión en baudios

uint8\_t data\_bits**;** //!< Número de bits de datos

bool enable\_parity**;** //!< Habilita el bit de paridad

**}** uart\_config\_t**;**

/\*\*

\* **@brief** Identificador único para un usuario en el sistema.

\*/

**typedef** uint32\_t user\_id\_t**;**

## Enumeraciones (enum)

El objetivo es explicar el propósito de la enumeración en su conjunto y el significado de cada uno de sus valores.

* **@enum (opcional):** Nombre de la enumeración.
* **@brief**: Se usa para dar un resumen corto del propósito del enum.
* **Comentario en línea (//!< ...)**: La mejor manera de documentar cada valor es con un comentario especial justo después de él, en la misma línea. Es posible realizar comentarios de más de una línea utilizando **/\*!< … \*/**.

### Ejemplo

/\*\*

\* **@brief** Códigos de estado para operaciones de dispositivo.

\*/

enum device\_status **{**

DEVICE\_OK **=** 0**,** //!< La operación se completó con éxito

DEVICE\_BUSY**,** //!< El dispositivo está ocupado

DEVICE\_ERROR**,** //!< Ocurrió un error no especificado

DEVICE\_TIMEOUT**,** //!< La op. no finalizó en el tiempo esperado

**};**

## Estructuras (struct)

Al igual que con los enum, se debe documentar el propósito general de la estructura y luego cada uno de sus miembros de forma individual.

* **@struct (opcional):** Nombre de la estructura.
* **@brief**: Resume para qué se utiliza la estructura.
* **Comentario en línea (//!< ...)**: Se usa después de cada miembro para describir qué dato almacena, sus unidades o su rango de valores.

### Ejemplo

/\*\*

\* **@brief** Almacena la configuración de un puerto serie (UART).

\*/

struct uart\_config **{**

uint32\_t baud\_rate**;** //!< Velocidad de transmisión en baudios

uint8\_t data\_bits**;** //!< Número de bits de datos

bool enable\_parity**;** //!< Habilita el bit de paridad

**};**

# Instrucciones/Etiquetas de Doxygen

## @brief

Este es un **resumen corto y directo** de lo que hace el elemento (función, fichero, etc.). Debe ser una única frase, concisa y empezar con un verbo en tercera persona. Es lo que aparece en las listas y resúmenes generales. **Qué incluir:**

* La acción principal y su objetivo.

/\*\*

\* **@brief** Calcula el factorial de un número entero no negativo.

\*/

## @details

Esta es la **descripción detallada**. Aquí puedes explayarte sobre el funcionamiento, el algoritmo utilizado, los casos de uso, las limitaciones o cualquier otra cosa que el resumen no pueda cubrir. **Qué incluir:**

* Explicación del algoritmo o la lógica interna.
* Contexto de uso o propósito más amplio.
* Posibles efectos secundarios.
* Referencias a documentación externa o especificaciones.

/\*\*

\* **@details** Esta función utiliza un bucle iterativo para calcular el

\* factorial. No maneja números negativos; para ellos, el

\* comportamiento es indefinido. El cálculo se realiza usando

\* `uint64\_t` para evitar desbordamientos con valores de entrada

\* relativamente pequeños (hasta 20!).

\*/

## @param[in]

Describe un **parámetro de entrada**. Cada parámetro de entrada de la función debe tener su propia etiqueta @param[in] .

**Qué incluir:**

* Nombre del parámetro.
* Descripción de lo que representa.
* Unidades (si aplica, ej. "milisegundos", "bytes").
* Rango de valores válidos o restricciones (ej. "no puede ser NULL", "debe ser mayor que cero").

/\*\*

\* **@param[in]** n Número entero no negativo para el cálculo.

\*/

## @param[out]

Describe un **parámetro de salida**. Se usa para argumentos que la función modifica para devolver un resultado (normalmente punteros).

**Qué incluir:**

* Nombre del parámetro.
* Qué valor almacenará la función en él al finalizar.
* Condiciones bajo las cuales se modifica.

/\*\*

\* **@param[out]** result Ptr. a una variable donde se almacenará

\* el resultado.

\*/

## @param[in,out]

Describe un **parámetro de entrada y también salida**. Se usa para argumentos que la función lee y también modifica para devolver un resultado (normalmente punteros).

**Qué incluir:**

* Nombre del parámetro.
* Qué valor almacenará la función en él al finalizar.
* Condiciones bajo las cuales se modifica.

/\*\*

\* **@param[in,out]** array\_origen Ptr. a la variable de tipo array que se

\* ordenará.

\*/

## @return

Describe el **valor de retorno** de la función. No debe confundirse con @param[out].

**Qué incluir:**

* Qué significa el valor devuelto.
* Si se devuelven diferentes valores para indicar éxito o error, descríbelos todos (ej. "0 en caso de éxito, -1 si hay error").

/\*\*

\* **@return** El factorial del número `n`, o 1 si `n` es 0.

\*/

## @retval

Es una versión más específica y estructurada que @return. Es ideal para funciones que devuelven diferentes valores (generalmente códigos de error) para indicar distintos resultados.

**Qué incluir:**

* Una lista de los posibles valores de retorno y el significado exacto de cada uno.

/\*\*

\* **@retval** 0 Si la operación fue exitosa.

\* **@retval** -1 Si el puntero de entrada era NULL.

\* **@retval** -2 Si no se pudo obtener el bloqueo del mutex.

\*/

## @note

Esta etiqueta se usa para añadir una **nota o comentario importante** que el desarrollador debe tener en cuenta. Es útil para resaltar aspectos que no son parte de la descripción principal, pero son cruciales.

**Qué incluir:**

* Advertencias sobre el rendimiento.
* Detalles de implementación específicos de una plataforma.
* Información sobre la reentrada o seguridad en hilos (thread-safety).
* Recordatorios o advertencias.

/\*\*

\* **@note** Esta función comparte recursos globales.

\*/

## @todo

Doxygen puede generar una lista centralizada de todas las tareas pendientes marcadas con esta etiqueta. Es más visible que un simple *// TODO:.*

**Qué incluir:**

* Una descripción clara de la tarea pendiente.

/\*\*

\* **@todo** Optimizar el algoritmo para reducir el uso de memoria.

\*/

## @see o @ref

Se usa para crear referencias cruzadas a otras funciones, estructuras o documentos relacionados. Ayuda a navegar por la documentación. Se indica mediante # delante del nombre de la función, macro, variable, etc.

**Qué incluir:**

* Nombres de otras funciones, macros o páginas que aporten contexto.

/\*\*

\* **@see** #device\_power\_on(), #estado\_actual

\*/

## @warning

Se usa para advertir sobre posibles efectos secundarios importantes, riesgos o situaciones que podrían no ser obvias.

**Qué incluir:**

* Consecuencias de un mal uso, problemas de rendimiento, advertencias sobre la no reentrada (si una función no es thread-safe), etc.

/\*\*

\* **@warning** Esta función modifica el buffer de entrada directamente.

\* Asegúrese de tener una copia si necesita los datos originales.

\*/

## @pre (Precondición)

Define una condición que debe ser verdadera antes de llamar a la función. Si no se cumple, el comportamiento de la función es indefinido.

**Qué incluir:**

* Suposiciones sobre el estado del sistema, o validaciones que la función *no* realiza internamente por motivos de rendimiento.

/\*\*

\* **@pre** El hardware del temporizador debe estar inicializado y

\* corriendo antes de invocar esta función.

\*/

## @post (Postcondición)

Define una condición que será verdadera después de que la función se ejecute con éxito.

**Qué incluir:**

* El estado en el que queda el sistema o los datos después de la ejecución.

/\*\*

\* **@post** El pin GPIO estará configurado en modo de salida y

\* en estado bajo.

\*/