KM_Medulla

1.0

Generated by Doxygen 1.9.8

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 km_coms_msg Struct Reference

Structure representing a KM_COMS message.

```
#include <km_coms.h>
```

Public Attributes

- uint8_t len
- uint8_t type
- uint8_t payload [KM_COMS_MSG_MAX_LEN-5]
- uint8_t crc

3.1.1 Detailed Description

Structure representing a KM_COMS message.

Message format: | SOF | LEN | TYPE | PAYLOAD | CRC |

- SOF: 1 byte Start-of-Message (0xAA)
- LEN: 1 byte payload length
- TYPE: 1 byte message type
- PAYLOAD: variable-length data
- CRC: 1 byte checksum

Definition at line 129 of file km_coms.h.

3.1.2 Member Data Documentation

3.1.2.1 crc

```
uint8_t km_coms_msg::crc
```

CRC checksum of the message

Definition at line 133 of file km_coms.h.

3.1.2.2 len

```
uint8_t km_coms_msg::len
```

Length of the payload

Definition at line 130 of file km_coms.h.

3.1.2.3 payload

```
uint8_t km_coms_msg::payload[KM_COMS_MSG_MAX_LEN-5]
```

Payload data

Definition at line 132 of file km_coms.h.

3.1.2.4 type

```
uint8_t km_coms_msg::type
```

Message type (see message_type_t)

Definition at line 131 of file km_coms.h.

The documentation for this struct was generated from the following file:

- km_coms.h

3.2 PID_Controller Struct Reference

Structure that reperesents a PID.

```
#include <km_pid.h>
```

Public Attributes

- float kp
- float ki
- float kd
- float integral
- float lastError
- uint64_t lastTime
- float outputMin
- float outputMax
- float integralMin
- float integralMax

3.2.1 Detailed Description

Structure that reperesents a PID.

Definition at line 24 of file km_pid.h.

3.2.2 Member Data Documentation

3.2.2.1 integral

```
float PID_Controller::integral
```

Integral accumulator

Definition at line 28 of file km_pid.h.

3.2.2.2 integralMax

```
float PID_Controller::integralMax
```

Definition at line 36 of file km_pid.h.

3.2.2.3 integralMin

```
float PID_Controller::integralMin
```

Anti-windup limits

Definition at line 35 of file km_pid.h.

3.2.2.4 kd

```
float PID_Controller::kd
```

Derivative gain

Definition at line 27 of file km_pid.h.

3.2.2.5 ki

```
float PID_Controller::ki
```

Integral gain

Definition at line 26 of file km_pid.h.

3.2.2.6 kp

```
float PID_Controller::kp
```

Proportional gain

Definition at line 25 of file km_pid.h.

3.2.2.7 lastError

```
float PID_Controller::lastError
```

Previous error for derivative

Definition at line 29 of file km_pid.h.

3.2.2.8 lastTime

```
uint64_t PID_Controller::lastTime
```

Last update time

Definition at line 30 of file km_pid.h.

3.2.2.9 outputMax

```
float PID_Controller::outputMax
```

Definition at line 34 of file km_pid.h.

3.2.2.10 outputMin

```
float PID_Controller::outputMin
```

Definition at line 33 of file km_pid.h.

The documentation for this struct was generated from the following file:

- km_pid.h

3.3 RTOS_Task Struct Reference

Structure that reperesents a task in FreeRTOS.

```
#include <km_rtos.h>
```

Public Attributes

- TaskHandle_t handle
- const char ∗ name
- KM_RTOS_TaskFunction_t taskFn
- void ∗ context
- uint32_t period_ms
- uint16_t stackSize
- UBaseType_t priority
- uint8_t active

3.3.1 Detailed Description

Structure that reperesents a task in FreeRTOS.

Definition at line 54 of file km_rtos.h.

3.3.2 Member Data Documentation

3.3.2.1 active

```
uint8_t RTOS_Task::active
```

Task active flag

Definition at line 65 of file km_rtos.h.

3.3.2.2 context

```
void* RTOS_Task::context
```

User context pointer

Definition at line 59 of file km_rtos.h.

3.3.2.3 handle

```
TaskHandle_t RTOS_Task::handle
```

FreeRTOS task handle

Definition at line 55 of file km_rtos.h.

3.3.2.4 name

```
const char* RTOS_Task::name
```

Task name

Definition at line 56 of file km_rtos.h.

3.3.2.5 period_ms

```
uint32_t RTOS_Task::period_ms
```

Execution period in milliseconds

Definition at line 61 of file km_rtos.h.

3.3.2.6 priority

```
UBaseType_t RTOS_Task::priority
```

Task priority

Definition at line 63 of file km_rtos.h.

3.3.2.7 stackSize

```
uint16_t RTOS_Task::stackSize
```

Stack size in words

Definition at line 62 of file km_rtos.h.

3.3.2.8 taskFn

```
KM_RTOS_TaskFunction_t RTOS_Task::taskFn
```

Logical task function

Definition at line 58 of file km_rtos.h.

The documentation for this struct was generated from the following file:

- km_rtos.h

3.4 sensor_struct Struct Reference

Structure that reperesents the direction sensor.

```
#include <km_sdir.h>
```

Public Attributes

- uint8_t connected
- uint8_t errorCount
- uint8_t max_error_count
- uint64_t lastReadTime
- uint16_t lastRawValue
- uint16_t centerOffset

3.4.1 Detailed Description

Structure that reperesents the direction sensor.

Definition at line 30 of file km_sdir.h.

3.4.2 Member Data Documentation

3.4.2.1 centerOffset

```
uint16_t sensor_struct::centerOffset
```

Definition at line 36 of file km_sdir.h.

3.4.2.2 connected

```
uint8_t sensor_struct::connected
```

Definition at line 31 of file km_sdir.h.

3.4.2.3 errorCount

```
uint8_t sensor_struct::errorCount
```

Definition at line 32 of file km_sdir.h.

3.4.2.4 lastRawValue

```
uint16_t sensor_struct::lastRawValue
```

Definition at line 35 of file km_sdir.h.

3.4.2.5 lastReadTime

```
uint64_t sensor_struct::lastReadTime
```

Definition at line 34 of file km_sdir.h.

3.4.2.6 max_error_count

```
uint8_t sensor_struct::max_error_count
```

Definition at line 33 of file km_sdir.h.

The documentation for this struct was generated from the following file:

- km_sdir.h

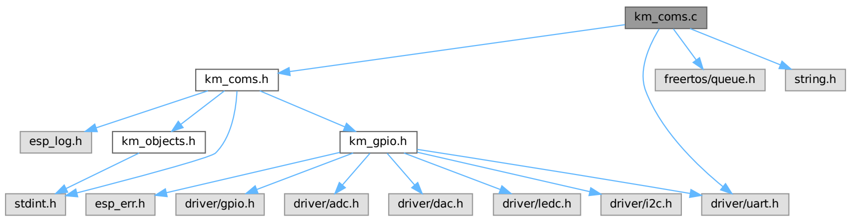
Chapter 4

File Documentation

4.1 km_coms.c File Reference

```
#include "km_coms.h"
#include "driver/uart.h"
#include "freertos/queue.h"
#include <string.h>
```

Include dependency graph for km_coms.c:



Macros

- #define KM_COMS_WAIT_SEM_AVAILABLE 5

Functions

- static void KM_COMS_ProccessPayload (km_coms_msg msg)

  Processes the payload of the incoming message and updates the corresponding application objects.
- static uint8_t KM_COMS_crc8 (uint8_t len, uint8_t type, const uint8_t ∗data)

  Calculates an 8-bit CRC checksum for a message.
- esp_err_t KM_COMS_Init (uart_port_t uart_port)

  Initializes the KM_COMS communication library for a given UART port.
- int KM_COMS_SendMsg (message_type_t type, uint8_t ∗payload, uint8_t len)

  Sends a communication message over UART.
- void km_coms_ReceiveMsg (void)

  Reads bytes from the UART and stores them into the internal RX buffer.
- void KM_COMS_ProccessMsgs (void)

  Processes all received messages from the RX buffer.

Variables

- static uint8_t rx_buffer [KM_COMS_MSG_MAX_LEN - 1]
- static size_t rx_buffer_len = 0
- static SemaphoreHandle_t km_coms_mutex
- static uart_port_t km_coms_uart = UART_NUM_0

4.1.1 Macro Definition Documentation

4.1.1.1 KM_COMS_WAIT_SEM_AVAILABLE

```
#define KM_COMS_WAIT_SEM_AVAILABLE 5
```

Definition at line 65 of file km_coms.c.

4.1.2 Function Documentation

4.1.2.1 KM_COMS_crc8()

```
static uint8_t KM_COMS_crc8 (
             uint8_t len,
             uint8_t type,
             const uint8_t * data )  [static]
```

Calculates an 8-bit CRC checksum for a message.

This function computes a CRC-8 checksum using the polynomial 0x07. The calculation includes the message length (`len`), message type (`type`), and the actual payload data. The CRC is computed sequentially:

1. XOR with the message length and process 8 bits.
2. XOR with the message type and process 8 bits.
3. XOR each byte of the payload data and process 8 bits per byte.

This checksum is used for detecting errors in communication messages.

Parameters

len	Length of the payload data in bytes.
type	Message type identifier.
data	Pointer to the payload data array.

Returns

The computed 8-bit CRC value.

Definition at line 358 of file km_coms.c.

```
00358                                                                         {
00359     uint8_t crc = 0x00;
```

```

    crc ^= len;
    for (int j = 0; j < 8; j++) {
        crc = (crc & 0x80) ? (crc « 1) ^ 0x07 : (crc « 1);
    }

    crc ^= type;
    for (int j = 0; j < 8; j++) {
        crc = (crc & 0x80) ? (crc « 1) ^ 0x07 : (crc « 1);
    }

    for (uint8_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (int j = 0; j < 8; j++) {
            crc = (crc & 0x80) ? (crc « 1) ^ 0x07 : (crc « 1);
        }
    }

    return crc;
}
```

4.1.2.2 KM_COMS_Init()

```
esp_err_t KM_COMS_Init (
              uart_port_t uart_port )
```

Initializes the KM_COMS communication library for a given UART port.

This function sets up the internal UART interface for the communication library, creates a mutex for thread-safe access to the RX buffer, and installs/configures the UART driver with the appropriate TX/RX pins and buffer sizes.

Steps performed by the function:

1. Stores the UART port to be used by the library (`km_coms_uart`).
2. Creates a mutex (`km_coms_mutex`) to protect access to the internal RX buffer.
    - Returns `ESP_ERR_NO_MEM` if mutex creation fails.
3. Installs the UART driver with predefined RX and TX buffer sizes.
4. Configures UART parameters (baud rate, data bits, parity, stop bits, etc.).
5. Sets the TX/RX pins based on the selected UART port.

Parameters

uart_port	The UART port to use (e.g., UART_NUM_1, UART_NUM_2).

Returns

ESP_OK if initialization succeeded.

ESP_ERR_NO_MEM if the mutex could not be created.

Note

This function should be called once at the start of the system before sending or receiving messages.

Thread-safety for the RX buffer is ensured via `km_coms_mutex`.

Definition at line 80 of file km_coms.c.

```
                                                {
    km_coms_uart = uart_port;

    km_coms_mutex = xSemaphoreCreateMutex();
    if(km_coms_mutex == NULL)
        return ESP_ERR_NO_MEM;

    // Instala el driver UART con buffers TX/RX
    uart_driver_install(km_coms_uart, BUF_SIZE_RX, BUF_SIZE_TX, 0, NULL, 0);
    uart_param_config(km_coms_uart, &uart_config);

    // Asigna pines según el puerto
    if(km_coms_uart == UART_NUM_2) {
        uart_set_pin(km_coms_uart, PIN_ORIN_UART_TX, PIN_ORIN_UART_RX,
                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    } else {
        uart_set_pin(km_coms_uart, PIN_USB_UART_TX, PIN_USB_UART_RX,
                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    }

    return ESP_OK;
}
```

4.1.2.3 KM_COMS_ProccessMsgs()

```
void KM_COMS_ProccessMsgs (
            void )
```

Processes all received messages from the RX buffer.

This function iterates through the `rx_buffer` to extract and process complete communication messages. Each message is expected to follow the protocol: [SOM | LEN | TYPE | PAYLOAD... | CRC].

Steps performed by the function:

1. Acquires the `km_coms_mutex` semaphore to ensure thread-safe access to the RX buffer.
2. Iterates through the buffer while enough bytes remain for at least a minimal message.
3. Searches for the Start-of-Message (SOM) byte. Skips invalid bytes.
4. Reads the payload length (`LEN`) and calculates the total message length.
5. If a complete message is available:
   - Constructs a `km_coms_msg` structure with `len`, `type`, `payload`, and `crc`.
   - Verifies the CRC using `KM_COMS_crc8()`.
   - If the CRC is valid, calls `KM_COMS_ProccessPayload()` to handle the message.
6. After processing, compacts the buffer by moving unprocessed bytes to the beginning.
7. Releases the `km_coms_mutex` semaphore.

This function ensures that partially received messages are retained in the buffer until complete data arrives.

Note

The minimum message length is 4 bytes: SOM + LEN + TYPE + CRC.

Thread-safety is provided by `km_coms_mutex`.

This function is intended to be called periodically from a FreeRTOS task.

Definition at line 181 of file km_coms.c.

```
                          {
    size_t processed = 0;

    if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) != pdTRUE)
        return;

    while(rx_buffer_len - processed >= 4) { // Mínimo: SOM + LEN + TYPE + CRC
        if(rx_buffer[processed] != KM_COMS_SOM) {
            processed++;
            continue;
        }

        uint8_t payload_len = rx_buffer[processed + 1];
        size_t total_len = 4 + payload_len; // SOM + LEN + TYPE + PAYLOAD + CRC

        if(rx_buffer_len - processed < total_len)
            break; // Mensaje incompleto, esperar más bytes

        // Construir mensaje
        km_coms_msg msg;
        msg.len = payload_len;
        msg.type = rx_buffer[processed + 2];
        memcpy(msg.payload, rx_buffer + processed + 3, payload_len);
        msg.crc = rx_buffer[processed + 3 + payload_len];

        // Verificar CRC
        uint8_t crc_calc = KM_COMS_crc8(msg.len, msg.type, msg.payload);
        if(crc_calc == msg.crc) {
            KM_COMS_ProccessPayload(msg);
        }

        processed += total_len;
    }

    // Compactar buffer: mover bytes no procesados al inicio
    if(processed > 0 && processed < rx_buffer_len) {
        memmove(rx_buffer, rx_buffer + processed, rx_buffer_len - processed);
        rx_buffer_len -= processed;
    } else if(processed >= rx_buffer_len) {
        rx_buffer_len = 0;
    }

    xSemaphoreGive(km_coms_mutex);
}
```

4.1.2.4 KM_COMS_ProccessPayload()

```
static void KM_COMS_ProccessPayload (
            km_coms_msg msg )  [static]
```

Processes the payload of the incoming message and updates the corresponding application objects.

This function interprets the payload of a received message (`km_coms_msg`) based on its `type`. Depending on the message type, it extracts the relevant data from the payload, converts it to a 64-bit integer (`int64_t`), and updates the appropriate shared object using `KM_OBJ_SetObjectValue()`.

Supported message types include:

- ORIN_TARG_THROTTLE: 1-byte payload representing the target throttle.
- ORIN_TARG_BRAKING: 1-byte payload representing the target braking.
- ORIN_TARG_STEERING: 2-byte payload where the first byte indicates direction and the second byte the magnitude. 0->positive turn, 1->negative turn

- ORIN_MISION: 1-byte payload representing the current mission state.
- ORIN_MACHINE_STATE: 1-byte payload representing the machine's current state.
- ORIN_HEARTBEAT: message indicating heartbeat; currently not processed.
- ORIN_SHUTDOWN: 1-byte payload indicating shutdown command.
- ORIN_COMPLETE: 7-byte payload updating all above objects in a single message.

Invalid payloads (wrong length or unknown direction for steering) are ignored.

Parameters

msg	The incoming message to process.

Definition at line 251 of file km_coms.c.

```
                                                    {
    ESP_LOGI("KM_coms", "RX msg type=0x%02X len=%d crc=0x%02X", msg.type, msg.len, msg.crc);

    int64_t object_value = 0;
    int8_t direction;

    switch (msg.type)
    {
    case ORIN_TARG_THROTTLE:
        if (msg.len != 1) return; // Invalid payload

        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);
        break;

    case ORIN_TARG_BRAKING:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);
        break;

    case ORIN_TARG_STEERING:
        if (msg.len != 2) return; // Invalid payload
        direction = msg.payload[0];
        if (direction == 0) {
            object_value = (int64_t)msg.payload[1];
        } else if (direction == 1){
            object_value -= (int64_t)msg.payload[1];
        } else {
            return; // Invalid payload
        }

        KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);
        break;

    case ORIN_MISION:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(MISION_ORIN, object_value);
        break;

    case ORIN_MACHINE_STATE:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);
        break;

    case ORIN_HEARTBEAT:
        // Ns si guardarlo en la libreria de variables o reinicar algo. ns
        break;

    case ORIN_SHUTDOWN:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
        break;

    case ORIN_COMPLETE:
        if (msg.len != 7) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);
```

```

            object_value = (int64_t)msg.payload[1];
            KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);

            direction = msg.payload[2];
            if (direction == 0) {
                object_value = (int64_t)msg.payload[3];
            } else if (direction == 1){
                object_value -= (int64_t)msg.payload[3];
            } else {
                return; // Invalid payload
            }
            KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);

            object_value = (int64_t)msg.payload[4];
            KM_OBJ_SetObjectValue(MISION_ORIN, object_value);

            object_value = (int64_t)msg.payload[5];
            KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);

            object_value = (int64_t)msg.payload[6];
            KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
            break;

        default:
            break;
    }
}
```

4.1.2.5 km_coms_ReceiveMsg()

```
void km_coms_ReceiveMsg (
              void  )
```

Reads bytes from the UART and stores them into the internal RX buffer.

This function retrieves available data from the UART interface (`km_coms_uart`) and appends it to the internal communication library buffer (`rx_buffer`). It ensures thread-safe access using the `km_coms_mutex` semaphore.

Steps performed by the function:

1. Checks how many bytes are currently available in the UART buffer.
2. Reads up to `KM_COMS_RX_CHUNK` bytes from the UART.
3. If bytes are read successfully, acquires the `km_coms_mutex` semaphore.
4. Appends the read bytes to the internal RX buffer.
   - If the buffer would overflow, it is reset to prevent memory corruption.
5. Releases the semaphore after updating the buffer.

Note

Thread-safety is provided via `km_coms_mutex`.

The internal RX buffer size is limited; excessive incoming data may reset the buffer.

This function is intended to be called periodically from a FreeRTOS task.

Definition at line 148 of file km_coms.c.

```
                                                     {
    uint8_t uart_chunk[KM_COMS_RX_CHUNK];
    size_t len_read = 0;
    uint8_t bytes2read = 0;

    // 1. Verificar cuantos bytes hay en el buffer UART
    size_t uart_len = 0;
    uart_get_buffered_data_len(km_coms_uart, &uart_len);
    if(uart_len == 0)
        return;

    // 2. Leer hasta KM_COMS_RX_CHUNK bytes de la UART
    if(uart_len > KM_COMS_RX_CHUNK)
        bytes2read = KM_COMS_RX_CHUNK;
    else
        bytes2read = uart_len;

    len_read = uart_read_bytes(km_coms_uart, uart_chunk, bytes2read, 0);
    if(len_read == 0)
        return;

    if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) == pdTRUE) {
        // 3. Copiar bytes al buffer interno
        if(rx_buffer_len + len_read > sizeof(rx_buffer)) {
            // Overflow, reiniciar buffer
            rx_buffer_len = 0;
        }
        memcpy(rx_buffer + rx_buffer_len, uart_chunk, len_read);
        rx_buffer_len += len_read;
        xSemaphoreGive(km_coms_mutex);
    }
}
```

4.1.2.6 KM_COMS_SendMsg()

```
int KM_COMS_SendMsg (
            message_type_t type,
            uint8_t * payload,
            uint8_t len )
```

Sends a communication message over UART.

This function constructs and sends a message frame according to the KM_COMS protocol: [SOM | LEN | TYPE | PAYLOAD | CRC]. It calculates the CRC for the message and attempts to send it via UART up to 5 times if the UART buffer is full.

Steps performed by the function:

1. Validates that the payload length does not exceed the maximum allowed.
2. Fills a `km_coms_msg` structure with length, type, payload, and calculated CRC.
3. Builds the UART frame: Start-of-Message (SOM), LEN, TYPE, PAYLOAD, CRC.
4. Attempts to send the entire frame via `uart_write_bytes`, retrying up to 5 times if the UART buffer is temporarily full, with a short delay between attempts.
5. Returns success if the full message is transmitted, or failure if it could not be sent.

Parameters

type	The type of message to send (message_type_t).
payload	Pointer to the payload data to send.
len	Length of the payload in bytes.

Returns

1 if the message was sent successfully, 0 if sending failed, -1 if payload is too long.

Note

This function can be called from a FreeRTOS task periodically or on-demand.

Definition at line 103 of file km_coms.c.

```
00103                                                                              {
00104     km_coms_msg msg;
00105     uint8_t frame[KM_COMS_MSG_MAX_LEN - 1];
00106     size_t total_sent = 0;
00107     int sent, attempts = 0;
00108 
00109     if(len > KM_COMS_MSG_MAX_LEN)
00110         return -1;
00111 
00112     msg.len = (uint8_t)len;
00113     msg.type = (uint8_t)type;
00114 
00115     memcpy(msg.payload, payload, len);
00116     msg.crc = KM_COMS_crc8(msg.len, msg.type, msg.payload); // LEN + TYPE + PAYLOAD
00117 
00118     // Armar frame
00119     frame[0] = (uint8_t)KM_COMS_SOM;
00120     frame[1] = (uint8_t)msg.len;
00121     frame[2] = (uint8_t)msg.type;
00122     memcpy(&frame[3], msg.payload, msg.len);
00123     frame[3 + msg.len] = (uint8_t)msg.crc;
00124 
00125     // Enviar mensaje, se intenta 5 veces
00126     while(total_sent < len+4 && attempts < 5) {
00127         sent = uart_write_bytes(km_coms_uart, frame + total_sent, (len + 4) - total_sent);
00128         total_sent += sent;
00129 
00130         if(sent < (len - total_sent)) {
00131             // buffer lleno, espera a que se vacíe
00132             attempts++;
00133             vTaskDelay(5 / portTICK_PERIOD_MS);
00134         }
00135     }
00136 
00137     // NO se ha podido enviar el mensaje correctamente
00138     if(attempts >= 5 || total_sent < 4 + len){
00139         ESP_LOGE("KM_coms", "El msg no se ha enviado, bytes enviados: %d, bytes que habia que enviar:
      %d", total_sent, 4+len);
00140         return 0;
00141     }
00142 
00143     // Se ha enviado el mensaje correctamente
00144     return 1;
00145 }
```

4.1.3 Variable Documentation

4.1.3.1 km_coms_mutex

```
SemaphoreHandle_t km_coms_mutex  [static]
```

Definition at line 71 of file km_coms.c.

4.1.3.2 km_coms_uart

```
uart_port_t km_coms_uart = UART_NUM_0  [static]
```

Definition at line 72 of file km_coms.c.

4.1.3.3 rx_buffer

```
uint8_t rx_buffer[KM_COMS_MSG_MAX_LEN - 1]  [static]
```

Definition at line 69 of file km_coms.c.

4.1.3.4 rx_buffer_len

```
size_t rx_buffer_len = 0  [static]
```

Definition at line 70 of file km_coms.c.

4.2 km_coms.c

Go to the documentation of this file.

```
/******************************************************************************
 * @file    km_coms.c
 * @brief   Implementation of the KM_COMS communication library for ESP32.
 *
 * This file implements the KM_COMS library, providing UART-based communication
 * between an ESP32 and an NVIDIA Orin (or other host system). The library handles:
 *  - Initialization and configuration of the UART peripheral
 *  - Sending and receiving framed messages
 *  - CRC8 validation for data integrity
 *  - Processing incoming payloads and updating shared application objects
 *  - FreeRTOS task integration for periodic message handling
 *
 * ---------------------------------------------------------------------------
 * PROTOCOL DETAILS
 * ---------------------------------------------------------------------------
 * Each message follows a custom framing format:
 *
 *   | SOF | LEN | TYPE | PAYLOAD | CRC |
 *
 * Fields:
 *   - SOF     : Start-of-frame byte (0xAA)
 *   - LEN     : 1-byte payload length
 *   - TYPE    : 1-byte message type (see message_type_t in km_coms.h)
 *   - PAYLOAD: N bytes of data up to 251 bytes, content depends on message type
 *   - CRC     : 1-byte CRC8 (XOR-based) over LEN, TYPE, and PAYLOAD
 *
 * Total size of message is 255 bytes
 *
 * The `message_type_t` enum defines all possible message types. Conceptually:
 *   - ESP32 → Orin messages contain telemetry data such as current speed, throttle,
 *     braking, steering, mission state, machine state, and shutdown status.
 *   - Orin → ESP32 messages carry commands such as target throttle, braking,
 *     steering, mission state, machine state, and heartbeat.
 *
 * Using the enum ensures type safety and consistency when sending or processing messages.
 *
 * The CRC ensures the integrity of the message. Invalid messages are discarded.
 *
 * The library uses internal buffers and a FreeRTOS mutex to provide thread-safe
 * access to shared objects and message queues.
 *
 * ---------------------------------------------------------------------------
 * FREE RTOS INTEGRATION
 * ---------------------------------------------------------------------------
 * Periodic tasks should call km_coms_ReceiveMsg() and KM_COMS_ProccessMsgs() to
 * read bytes from UART, assemble messages, validate CRC, and process payloads.
 * KM_COMS_SendMsg() can be used to transmit messages safely from tasks.
 *
 * Author: Adrian Navarredonda Arizaleta
 * Date:   14-02-2026
 * Version: 1.0
 ******************************************************************************/

#include "km_coms.h"

/******************************* INCLUDES INTERNOS ****************************/
// Headers internos opcionales, dependencias privadas
#include "driver/uart.h"
```

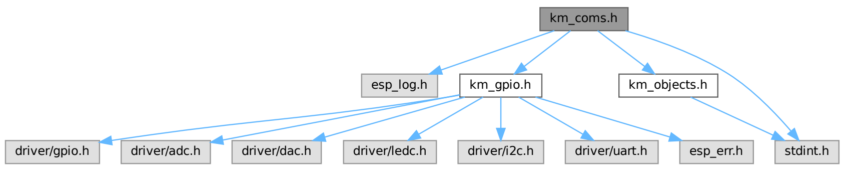
```c
#include "freertos/queue.h"
#include <string.h>

/******************************** MACROS PRIVADAS ********************************/
// Constantes internas, flags de debug

#define KM_COMS_WAIT_SEM_AVAILABLE 5

/******************************** VARIABLES PRIVADAS **************************/
// Variables globales internas (static)
static uint8_t rx_buffer[KM_COMS_MSG_MAX_LEN - 1]; //[0-255]
static size_t rx_buffer_len = 0;
static SemaphoreHandle_t km_coms_mutex;
static uart_port_t km_coms_uart = UART_NUM_0;

/******************************** DECLARACION FUNCIONES PRIVADAS ***************/
static void KM_COMS_ProccessPayload(km_coms_msg msg);
static uint8_t KM_COMS_crc8(uint8_t len, uint8_t type, const uint8_t *data);

/******************************** FUNCIONES PÚBLICAS **************************/

esp_err_t KM_COMS_Init(uart_port_t uart_port) {
    km_coms_uart = uart_port;

    km_coms_mutex = xSemaphoreCreateMutex();
    if(km_coms_mutex == NULL)
        return ESP_ERR_NO_MEM;

    // Instala el driver UART con buffers TX/RX
    uart_driver_install(km_coms_uart, BUF_SIZE_RX, BUF_SIZE_TX, 0, NULL, 0);
    uart_param_config(km_coms_uart, &uart_config);

    // Asigna pines según el puerto
    if(km_coms_uart == UART_NUM_2) {
        uart_set_pin(km_coms_uart, PIN_ORIN_UART_TX, PIN_ORIN_UART_RX,
                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    } else {
        uart_set_pin(km_coms_uart, PIN_USB_UART_TX, PIN_USB_UART_RX,
                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    }

    return ESP_OK;
}

int KM_COMS_SendMsg(message_type_t type, uint8_t *payload, uint8_t len) {
    km_coms_msg msg;
    uint8_t frame[KM_COMS_MSG_MAX_LEN - 1];
    size_t total_sent = 0;
    int sent, attempts = 0;

    if(len > KM_COMS_MSG_MAX_LEN)
        return -1;

    msg.len = (uint8_t)len;
    msg.type = (uint8_t)type;

    memcpy(msg.payload, payload, len);
    msg.crc = KM_COMS_crc8(msg.len, msg.type, msg.payload); // LEN + TYPE + PAYLOAD

    // Armar frame
    frame[0] = (uint8_t)KM_COMS_SOM;
    frame[1] = (uint8_t)msg.len;
    frame[2] = (uint8_t)msg.type;
    memcpy(&frame[3], msg.payload, msg.len);
    frame[3 + msg.len] = (uint8_t)msg.crc;

    // Enviar mensaje, se intenta 5 veces
    while(total_sent < len+4 && attempts < 5) {
        sent = uart_write_bytes(km_coms_uart, frame + total_sent, (len + 4) - total_sent);
        total_sent += sent;

        if(sent < (len - total_sent)) {
            // buffer lleno, espera a que se vacíe
            attempts++;
            vTaskDelay(5 / portTICK_PERIOD_MS);
        }
    }

    // NO se ha podido enviar el mensaje correctamente
    if(attempts >= 5 || total_sent < 4 + len){
        ESP_LOGE("KM_coms", "El msg no se ha enviado, bytes enviados: %d, bytes que habia que enviar: %d", total_sent, 4+len);
        return 0;
    }

    // Se ha enviado el mensaje correctamente
    return 1;
```

```
}


void km_coms_ReceiveMsg(void) {
    uint8_t uart_chunk[KM_COMS_RX_CHUNK];
    size_t len_read = 0;
    uint8_t bytes2read = 0;

    // 1. Verificar cuantos bytes hay en el buffer UART
    size_t uart_len = 0;
    uart_get_buffered_data_len(km_coms_uart, &uart_len);
    if(uart_len == 0)
        return;

    // 2. Leer hasta KM_COMS_RX_CHUNK bytes de la UART
    if(uart_len > KM_COMS_RX_CHUNK)
        bytes2read = KM_COMS_RX_CHUNK;
    else
        bytes2read = uart_len;

    len_read = uart_read_bytes(km_coms_uart, uart_chunk, bytes2read, 0);
    if(len_read == 0)
        return;

    if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) == pdTRUE) {
        // 3. Copiar bytes al buffer interno
        if(rx_buffer_len + len_read > sizeof(rx_buffer)) {
            // Overflow, reiniciar buffer
            rx_buffer_len = 0;
        }
        memcpy(rx_buffer + rx_buffer_len, uart_chunk, len_read);
        rx_buffer_len += len_read;
        xSemaphoreGive(km_coms_mutex);
    }
}

void KM_COMS_ProccessMsgs(void) {
    size_t processed = 0;

    if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) != pdTRUE)
        return;

    while(rx_buffer_len - processed >= 4) { // Mínimo: SOM + LEN + TYPE + CRC
        if(rx_buffer[processed] != KM_COMS_SOM) {
            processed++;
            continue;
        }

        uint8_t payload_len = rx_buffer[processed + 1];
        size_t total_len = 4 + payload_len; // SOM + LEN + TYPE + PAYLOAD + CRC

        if(rx_buffer_len - processed < total_len)
            break; // Mensaje incompleto, esperar más bytes

        // Construir mensaje
        km_coms_msg msg;
        msg.len = payload_len;
        msg.type = rx_buffer[processed + 2];
        memcpy(msg.payload, rx_buffer + processed + 3, payload_len);
        msg.crc = rx_buffer[processed + 3 + payload_len];

        // Verificar CRC
        uint8_t crc_calc = KM_COMS_crc8(msg.len, msg.type, msg.payload);
        if(crc_calc == msg.crc) {
            KM_COMS_ProccessPayload(msg);
        }

        processed += total_len;
    }

    // Compactar buffer: mover bytes no procesados al inicio
    if(processed > 0 && processed < rx_buffer_len) {
        memmove(rx_buffer, rx_buffer + processed, rx_buffer_len - processed);
        rx_buffer_len -= processed;
    } else if(processed >= rx_buffer_len) {
        rx_buffer_len = 0;
    }

    xSemaphoreGive(km_coms_mutex);
}

/******************************* FUNCIONES PRIVADAS ***************************/

static void KM_COMS_ProccessPayload(km_coms_msg msg) {
    ESP_LOGI("KM_coms", "RX msg type=0x%02X len=%d crc=0x%02X", msg.type, msg.len, msg.crc);

    int64_t object_value = 0;
    int8_t direction;

    switch (msg.type)
    {
    case ORIN_TARG_THROTTLE:
        if (msg.len != 1) return; // Invalid payload

        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);
        break;

    case ORIN_TARG_BRAKING:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);
        break;

    case ORIN_TARG_STEERING:
        if (msg.len != 2) return; // Invalid payload
        direction = msg.payload[0];
        if (direction == 0) {
            object_value = (int64_t)msg.payload[1];
        } else if (direction == 1){
            object_value -= (int64_t)msg.payload[1];
        } else {
            return; // Invalid payload
        }

        KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);
        break;

    case ORIN_MISION:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(MISION_ORIN, object_value);
        break;

    case ORIN_MACHINE_STATE:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);
        break;

    case ORIN_HEARTBEAT:
        // Ns si guardarlo en la libreria de variables o reinicar algo. ns
        break;

    case ORIN_SHUTDOWN:
        if (msg.len != 1) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
        break;

    case ORIN_COMPLETE:
        if (msg.len != 7) return; // Invalid payload
        object_value = (int64_t)msg.payload[0];
        KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);

        object_value = (int64_t)msg.payload[1];
        KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);

        direction = msg.payload[2];
        if (direction == 0) {
            object_value = (int64_t)msg.payload[3];
        } else if (direction == 1){
            object_value -= (int64_t)msg.payload[3];
        } else {
            return; // Invalid payload
        }
        KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);

        object_value = (int64_t)msg.payload[4];
        KM_OBJ_SetObjectValue(MISION_ORIN, object_value);

        object_value = (int64_t)msg.payload[5];
        KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);

        object_value = (int64_t)msg.payload[6];
        KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
        break;

    default:
        break;
    }
}

static uint8_t KM_COMS_crc8(uint8_t len, uint8_t type, const uint8_t *data) {
    uint8_t crc = 0x00;

    crc ^= len;
    for (int j = 0; j < 8; j++) {
        crc = (crc & 0x80) ? (crc « 1) ^ 0x07 : (crc « 1);
    }

    crc ^= type;
    for (int j = 0; j < 8; j++) {
        crc = (crc & 0x80) ? (crc « 1) ^ 0x07 : (crc « 1);
    }

    for (uint8_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (int j = 0; j < 8; j++) {
            crc = (crc & 0x80) ? (crc « 1) ^ 0x07 : (crc « 1);
        }
    }

    return crc;
}

/******************************* FIN DE ARCHIVO ********************************/
```

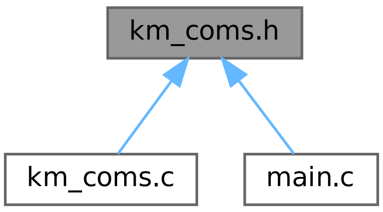
4.3 km_coms.h File Reference

```
#include "esp_log.h"
#include "km_gpio.h"
#include "km_objects.h"
#include <stdint.h>
```

Include dependency graph for km_coms.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct km_coms_msg

  Structure representing a KM_COMS message.

Macros

- #define KM_COMS_SOM 0xAA
- #define KM_COMS_MSG_MAX_LEN 256
- #define KM_COMS_RX_CHUNK 64
- #define BUF_SIZE_TX 2048
- #define BUF_SIZE_RX 2048

Enumerations

- enum message_type_t {
  ESP_ACT_SPEED = 0x01 , ESP_ACT_THROTTLE = 0x02 , ESP_ACT_BRAKING = 0x03 , ESP_ACT_STEERING = 0x04 ,
  ESP_MISION = 0x05 , ESP_MACHINE_STATE = 0x06 , ESP_ACT_SHUTDOWN = 0x07 , ESP_HEARTBEAT = 0x08 ,
  ESP_COMPLETE = 0x09 , ORIN_TARG_THROTTLE = 0x20 , ORIN_TARG_BRAKING = 0x21 , ORIN_TARG_STEERING = 0x22 ,
  ORIN_MISION = 0x23 , ORIN_MACHINE_STATE = 0x24 , ORIN_HEARTBEAT = 0x25 , ORIN_SHUTDOWN = 0x26 ,
  ORIN_COMPLETE = 0x27 }

  Enum defining all message types supported by KM_COMS.

Functions

- esp_err_t KM_COMS_Init (uart_port_t uart_port)

  Initializes the KM_COMS communication library for a given UART port.
- int KM_COMS_SendMsg (message_type_t type, uint8_t ∗payload, uint8_t len)

  Sends a communication message over UART.
- void km_coms_ReceiveMsg (void)

  Reads bytes from the UART and stores them into the internal RX buffer.
- void KM_COMS_ProccessMsgs (void)

  Processes all received messages from the RX buffer.

4.3.1 Macro Definition Documentation

4.3.1.1 BUF_SIZE_RX

```
#define BUF_SIZE_RX 2048
```

Definition at line 75 of file km_coms.h.

4.3.1.2 BUF_SIZE_TX

```
#define BUF_SIZE_TX 2048
```

Definition at line 74 of file km_coms.h.

4.3.1.3 KM_COMS_MSG_MAX_LEN

```
#define KM_COMS_MSG_MAX_LEN 256
```

Definition at line 72 of file km_coms.h.

4.3.1.4 KM_COMS_RX_CHUNK

```
#define KM_COMS_RX_CHUNK 64
```

Definition at line 73 of file km_coms.h.

4.3.1.5 KM_COMS_SOM

```
#define KM_COMS_SOM 0xAA
```

Definition at line 71 of file km_coms.h.

4.3.2 Enumeration Type Documentation

4.3.2.1 message_type_t

```
enum message_type_t
```

Enum defining all message types supported by KM_COMS.

Values are split between ESP32 -> Orin and Orin -> ESP32 messages.

Enumerator

ESP_ACT_SPEED	Actual speed telemetry (m/s)
ESP_ACT_THROTTLE	Actual throttle telemetry (0-100)
ESP_ACT_BRAKING	Actual braking telemetry (0-100)
ESP_ACT_STEERING	Actual steering telemetry (-100 to 100)
ESP_MISION	Current mission/state
ESP_MACHINE_STATE	Current machine sub-state
ESP_ACT_SHUTDOWN	Shutdown state
ESP_HEARTBEAT	ESP32 heartbeat message
ESP_COMPLETE	Full telemetry message
ORIN_TARG_THROTTLE	Target throttle command (0-100)
ORIN_TARG_BRAKING	Target braking command (0-100)
ORIN_TARG_STEERING	Target steering command (-100 to 100)
ORIN_MISION	Mission command/state
ORIN_MACHINE_STATE	Machine state command
ORIN_HEARTBEAT	Orin heartbeat message
ORIN_SHUTDOWN	Shutdown command
ORIN_COMPLETE	Complete command with all fields

Definition at line 85 of file km_coms.h.

```
{
    // ==========================
    // ESP32 --> Orin (0x01 - 0x1F)
    // ==========================
    ESP_ACT_SPEED              = 0x01,
    ESP_ACT_THROTTLE           = 0x02,
    ESP_ACT_BRAKING            = 0x03,
    ESP_ACT_STEERING           = 0x04,
    ESP_MISION                 = 0x05,
    ESP_MACHINE_STATE          = 0x06,
    ESP_ACT_SHUTDOWN           = 0x07,
    ESP_HEARTBEAT              = 0x08,
    ESP_COMPLETE               = 0x09,
    // ==========================
    // Orin --> ESP32 (0x20 - 0x3F)
    // ==========================
    ORIN_TARG_THROTTLE         = 0x20,
    ORIN_TARG_BRAKING          = 0x21,
    ORIN_TARG_STEERING         = 0x22,
    ORIN_MISION                = 0x23,
    ORIN_MACHINE_STATE         = 0x24,
    ORIN_HEARTBEAT             = 0x25,
    ORIN_SHUTDOWN              = 0x26,
    ORIN_COMPLETE              = 0x27,
    // ==========================
    // Others (0x40 - 0xFF)
    // ==========================
} message_type_t;
```

4.3.3 Function Documentation

4.3.3.1 KM_COMS_Init()

```
esp_err_t KM_COMS_Init (
              uart_port_t uart_port )
```

Initializes the KM_COMS communication library for a given UART port.

This function sets up the internal UART interface for the communication library, creates a mutex for thread-safe access to the RX buffer, and installs/configures the UART driver with the appropriate TX/RX pins and buffer sizes.

Steps performed by the function:

1. Stores the UART port to be used by the library (`km_coms_uart`).
2. Creates a mutex (`km_coms_mutex`) to protect access to the internal RX buffer.
    - Returns `ESP_ERR_NO_MEM` if mutex creation fails.
3. Installs the UART driver with predefined RX and TX buffer sizes.
4. Configures UART parameters (baud rate, data bits, parity, stop bits, etc.).
5. Sets the TX/RX pins based on the selected UART port.

Parameters

uart_port	The UART port to use (e.g., UART_NUM_1, UART_NUM_2).

Returns

ESP_OK if initialization succeeded.

ESP_ERR_NO_MEM if the mutex could not be created.

Note

This function should be called once at the start of the system before sending or receiving messages.

Thread-safety for the RX buffer is ensured via `km_coms_mutex`.

Definition at line 80 of file km_coms.c.

```
                                             {
    km_coms_uart = uart_port;

    km_coms_mutex = xSemaphoreCreateMutex();
    if(km_coms_mutex == NULL)
        return ESP_ERR_NO_MEM;

    // Instala el driver UART con buffers TX/RX
    uart_driver_install(km_coms_uart, BUF_SIZE_RX, BUF_SIZE_TX, 0, NULL, 0);
    uart_param_config(km_coms_uart, &uart_config);

    // Asigna pines según el puerto
    if(km_coms_uart == UART_NUM_2) {
        uart_set_pin(km_coms_uart, PIN_ORIN_UART_TX, PIN_ORIN_UART_RX,
                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    } else {
        uart_set_pin(km_coms_uart, PIN_USB_UART_TX, PIN_USB_UART_RX,
                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    }

    return ESP_OK;
}
```

4.3.3.2 KM_COMS_ProccessMsgs()

```
void KM_COMS_ProccessMsgs (
            void   )
```

Processes all received messages from the RX buffer.

This function iterates through the `rx_buffer` to extract and process complete communication messages. Each message is expected to follow the protocol: [SOM | LEN | TYPE | PAYLOAD... | CRC].

Steps performed by the function:

1. Acquires the `km_coms_mutex` semaphore to ensure thread-safe access to the RX buffer.
2. Iterates through the buffer while enough bytes remain for at least a minimal message.
3. Searches for the Start-of-Message (SOM) byte. Skips invalid bytes.
4. Reads the payload length (`LEN`) and calculates the total message length.
5. If a complete message is available:
    - Constructs a `km_coms_msg` structure with `len`, `type`, `payload`, and `crc`.
    - Verifies the CRC using `KM_COMS_crc8()`.
    - If the CRC is valid, calls `KM_COMS_ProccessPayload()` to handle the message.
6. After processing, compacts the buffer by moving unprocessed bytes to the beginning.
7. Releases the `km_coms_mutex` semaphore.

This function ensures that partially received messages are retained in the buffer until complete data arrives.

Note

The minimum message length is 4 bytes: SOM + LEN + TYPE + CRC.

Thread-safety is provided by `km_coms_mutex`.

This function is intended to be called periodically from a FreeRTOS task.

Definition at line 181 of file km_coms.c.

```
                              {
    size_t processed = 0;

    if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) != pdTRUE)
        return;

    while(rx_buffer_len - processed >= 4) { // Mínimo: SOM + LEN + TYPE + CRC
        if(rx_buffer[processed] != KM_COMS_SOM) {
            processed++;
            continue;
        }

        uint8_t payload_len = rx_buffer[processed + 1];
        size_t total_len = 4 + payload_len; // SOM + LEN + TYPE + PAYLOAD + CRC

        if(rx_buffer_len - processed < total_len)
            break; // Mensaje incompleto, esperar más bytes

        // Construir mensaje
        km_coms_msg msg;
        msg.len = payload_len;
        msg.type = rx_buffer[processed + 2];
        memcpy(msg.payload, rx_buffer + processed + 3, payload_len);
        msg.crc = rx_buffer[processed + 3 + payload_len];

        // Verificar CRC
        uint8_t crc_calc = KM_COMS_crc8(msg.len, msg.type, msg.payload);
        if(crc_calc == msg.crc) {
            KM_COMS_ProccessPayload(msg);
        }

        processed += total_len;
    }

    // Compactar buffer: mover bytes no procesados al inicio
    if(processed > 0 && processed < rx_buffer_len) {
        memmove(rx_buffer, rx_buffer + processed, rx_buffer_len - processed);
        rx_buffer_len -= processed;
    } else if(processed >= rx_buffer_len) {
        rx_buffer_len = 0;
    }

    xSemaphoreGive(km_coms_mutex);
}
```

4.3.3.3 km_coms_ReceiveMsg()

```
void km_coms_ReceiveMsg (
            void )
```

Reads bytes from the UART and stores them into the internal RX buffer.

This function retrieves available data from the UART interface (`km_coms_uart`) and appends it to the internal communication library buffer (`rx_buffer`). It ensures thread-safe access using the `km_coms_mutex` semaphore.

Steps performed by the function:

1. Checks how many bytes are currently available in the UART buffer.
2. Reads up to `KM_COMS_RX_CHUNK` bytes from the UART.
3. If bytes are read successfully, acquires the `km_coms_mutex` semaphore.
4. Appends the read bytes to the internal RX buffer.
    - If the buffer would overflow, it is reset to prevent memory corruption.
5. Releases the semaphore after updating the buffer.

Note

Thread-safety is provided via `km_coms_mutex`.

The internal RX buffer size is limited; excessive incoming data may reset the buffer.

This function is intended to be called periodically from a FreeRTOS task.

Definition at line 148 of file km_coms.c.

```
                                  {
    uint8_t uart_chunk[KM_COMS_RX_CHUNK];
    size_t len_read = 0;
    uint8_t bytes2read = 0;

    // 1. Verificar cuantos bytes hay en el buffer UART
    size_t uart_len = 0;
    uart_get_buffered_data_len(km_coms_uart, &uart_len);
    if(uart_len == 0)
        return;

    // 2. Leer hasta KM_COMS_RX_CHUNK bytes de la UART
    if(uart_len > KM_COMS_RX_CHUNK)
        bytes2read = KM_COMS_RX_CHUNK;
    else
        bytes2read = uart_len;

    len_read = uart_read_bytes(km_coms_uart, uart_chunk, bytes2read, 0);
    if(len_read == 0)
        return;

    if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) == pdTRUE) {
        // 3. Copiar bytes al buffer interno
        if(rx_buffer_len + len_read > sizeof(rx_buffer)) {
            // Overflow, reiniciar buffer
            rx_buffer_len = 0;
        }
        memcpy(rx_buffer + rx_buffer_len, uart_chunk, len_read);
        rx_buffer_len += len_read;
        xSemaphoreGive(km_coms_mutex);
    }
}
```

4.3.3.4 KM_COMS_SendMsg()

```
int KM_COMS_SendMsg (
            message_type_t type,
            uint8_t * payload,
            uint8_t len )
```

Sends a communication message over UART.

This function constructs and sends a message frame according to the KM_COMS protocol: [SOM | LEN | TYPE | PAYLOAD | CRC]. It calculates the CRC for the message and attempts to send it via UART up to 5 times if the UART buffer is full.

Steps performed by the function:

1. Validates that the payload length does not exceed the maximum allowed.
2. Fills a `km_coms_msg` structure with length, type, payload, and calculated CRC.
3. Builds the UART frame: Start-of-Message (SOM), LEN, TYPE, PAYLOAD, CRC.
4. Attempts to send the entire frame via `uart_write_bytes`, retrying up to 5 times if the UART buffer is temporarily full, with a short delay between attempts.
5. Returns success if the full message is transmitted, or failure if it could not be sent.

Parameters

type	The type of message to send (message_type_t).
payload	Pointer to the payload data to send.
len	Length of the payload in bytes.

Returns

1 if the message was sent successfully, 0 if sending failed, -1 if payload is too long.

Note

This function can be called from a FreeRTOS task periodically or on-demand.

Definition at line 103 of file km_coms.c.

```
                                                                    {
    km_coms_msg msg;
    uint8_t frame[KM_COMS_MSG_MAX_LEN - 1];
    size_t total_sent = 0;
    int sent, attempts = 0;

    if(len > KM_COMS_MSG_MAX_LEN)
        return -1;

    msg.len = (uint8_t)len;
    msg.type = (uint8_t)type;

    memcpy(msg.payload, payload, len);
    msg.crc = KM_COMS_crc8(msg.len, msg.type, msg.payload); // LEN + TYPE + PAYLOAD

    // Armar frame
    frame[0] = (uint8_t)KM_COMS_SOM;
    frame[1] = (uint8_t)msg.len;
    frame[2] = (uint8_t)msg.type;
    memcpy(&frame[3], msg.payload, msg.len);
    frame[3 + msg.len] = (uint8_t)msg.crc;

    // Enviar mensaje, se intenta 5 veces
    while(total_sent < len+4 && attempts < 5) {
        sent = uart_write_bytes(km_coms_uart, frame + total_sent, (len + 4) - total_sent);
        total_sent += sent;

        if(sent < (len - total_sent)) {
            // buffer lleno, espera a que se vacíe
            attempts++;
            vTaskDelay(5 / portTICK_PERIOD_MS);
        }
    }

    // NO se ha podido enviar el mensaje correctamente
    if(attempts >= 5 || total_sent < 4 + len){
        ESP_LOGE("KM_coms", "El msg no se ha enviado, bytes enviados: %d, bytes que habia que enviar:
%d", total_sent, 4+len);
        return 0;
    }

    // Se ha enviado el mensaje correctamente
    return 1;
}
```

4.4 km_coms.h

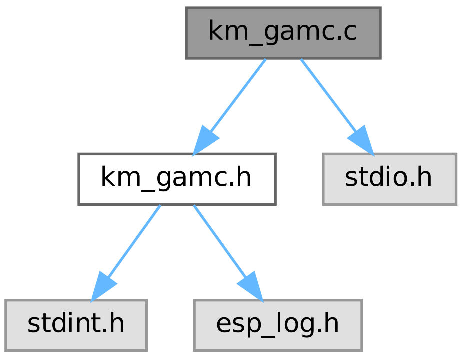
Go to the documentation of this file.

```
/*****************************************************************************
 * @file    km_coms.h
 * @brief   Public API for the KM_COMS communication library.
 *
 * This header defines the interface for the KM_COMS library, which provides:
 *  - UART-based communication between an ESP32 and NVIDIA Orin (or other host)
 *  - Message framing, CRC checks, and payload handling
 *  - Functions to send, receive, and process messages
 *
 * The library supports periodic tasks that handle incoming/outgoing messages
 * safely in a FreeRTOS environment.
 *
 * @author  Adrian Navarredonda Arizaleta
 * @date    14-02-2026
 * @version 1.0
 ******************************************************************************/

#ifndef KM_COMS_H
#define KM_COMS_H

/******************************* INCLUDES ***********************************/
// Includes necesarios para la API pública
#include "esp_log.h" // Para log
#include "km_gpio.h"
#include "km_objects.h"
#include <stdint.h>

// Estructura mensaje, | es solo para visualizar los disintintos campos, no esta
// en el mensaje
//  | SOF | LEN | TYPE | PAYLOAD | CRC |

// Donde:

//     SOF → 1 byte (ej: 0xAA)

//     LEN → 1 byte (longitud del payload)

//     TYPE → 1 byte (tipo de mensaje)

//     PAYLOAD → N bytes

//     CRC → 1 byte (checksum simple XOR)

// -------------------------- Tipos de mensajes --------------------------------
// ## Orin <-> ESP32 Messaging (UTF-8)

// All inter-computer messages are UTF-8 text. Payloads are defined below; line framing and exact
// field ordering should follow the agreed message definitions when implemented.

// **ESP32 -> Orin (telemetry):**
// - `actual_speed` (m/s, float)
// - `actual_acc` (m/s^2, float, signed)
// - `actual_braking` (0-1, float; interpreted as brake pedal or hydraulic pressure)
// - `actual_steering` (rad, float)
// - `mission` (enum; state machine owner TBD)
// - `machine_state` (enum; mission sub-state)
// - `actual_shutdown` (0/1, end of SDC loop state)
// - `esp32_heartbeat`

// **Orin -> ESP32 (commands):**
// - `target_throttle` (0-1, float)
// - `target_braking` (0-1, float)
// - `target_steering` (-1 to 1, float)
// - `mission` (enum; state machine owner TBD)
// - `machine_state` (enum; mission sub-state)
// - `orin_heartbeat`

/******************************* DEFINES PÚBLICAS ****************************/
// Constantes, flags o configuraciones visibles desde fuera de la librería

#define KM_COMS_SOM 0xAA
#define KM_COMS_MSG_MAX_LEN 256
#define KM_COMS_RX_CHUNK 64 // Lectura de la UART por bloques
#define BUF_SIZE_TX 2048
#define BUF_SIZE_RX 2048

/******************************* TIPOS PÚBLICOS ********************************/
// Estructuras, enums, typedefs públicos

typedef enum
{
    // ==========================
    // ESP32 --> Orin (0x01 - 0x1F)
    // ==========================
    ESP_ACT_SPEED            = 0x01,
    ESP_ACT_THROTTLE         = 0x02,
    ESP_ACT_BRAKING          = 0x03,
    ESP_ACT_STEERING         = 0x04,
    ESP_MISION               = 0x05,
    ESP_MACHINE_STATE        = 0x06,
    ESP_ACT_SHUTDOWN         = 0x07,
    ESP_HEARTBEAT            = 0x08,
    ESP_COMPLETE             = 0x09,
    // ==========================
    // Orin --> ESP32 (0x20 - 0x3F)
    // ==========================
    ORIN_TARG_THROTTLE      = 0x20,
    ORIN_TARG_BRAKING       = 0x21,
    ORIN_TARG_STEERING      = 0x22,
    ORIN_MISION             = 0x23,
    ORIN_MACHINE_STATE      = 0x24,
    ORIN_HEARTBEAT          = 0x25,
    ORIN_SHUTDOWN           = 0x26,
    ORIN_COMPLETE           = 0x27,
    // ==========================
    // Others (0x40 - 0xFF)
    // ==========================
} message_type_t;

typedef struct {
    uint8_t len;
    uint8_t type;
    uint8_t payload[KM_COMS_MSG_MAX_LEN-5];
    uint8_t crc;
} km_coms_msg;

/******************************* VARIABLES PÚBLICAS ***************************/
// Variables globales visibles (si realmente se necesitan)

// extern int ejemplo_variable_publica;

/******************************* FUNCIONES PÚBLICAS ***************************/
esp_err_t KM_COMS_Init(uart_port_t uart_port);

int KM_COMS_SendMsg(message_type_t type, uint8_t *payload, uint8_t len);

void km_coms_ReceiveMsg(void);

void KM_COMS_ProccessMsgs(void);

#endif /* KM_COMS_H */
```

4.5 km_gamc.c File Reference

```
#include "km_gamc.h"
#include <stdio.h>
```

Include dependency graph for km_gamc.c:



4.6 km_gamc.c

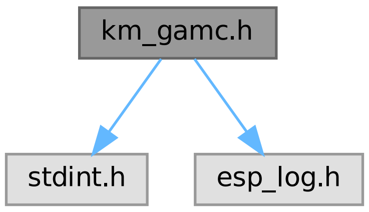
Go to the documentation of this file.

```
/******************************************************************************
 * @file    nombre_libreria.c
 * @brief   Implementación de la librería.
 ******************************************************************************/

 #include "km_gamc.h"
 #include <stdio.h>   // solo si es necesario para debug interno

 /******************************* INCLUDES INTERNOS ***************************/
 // Headers internos opcionales, dependencias privadas

 /******************************* MACROS PRIVADAS *******************************/
 // Constantes internas, flags de debug
 // #define LIBRERIA_DEBUG 1

 /******************************* VARIABLES PRIVADAS **************************/
 // Variables globales internas (static)

 /******************************* DECLARACION FUNCIONES PRIVADAS ***************/


 /******************************* FUNCIONES PÚBLICAS **************************/
 // int libreria_operacion(int valor) {
 //     ...
 // }

 /******************************* FUNCIONES PRIVADAS **************************/
 // static void funcion_privada(void);

 /******************************* FIN DE ARCHIVO *******************************/
```

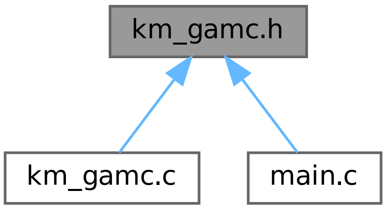
4.7 km_gamc.h File Reference

```
#include <stdint.h>
#include "esp_log.h"
```

Include dependency graph for km_gamc.h:



This graph shows which files directly or indirectly include this file:



4.8 km_gamc.h

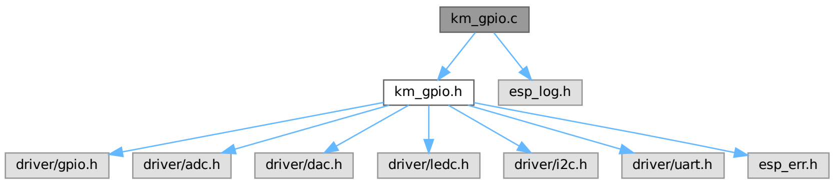
Go to the documentation of this file.

```
/***************************************************************************
 * @file    nombre_libreria.h
 * @brief   Interfaz pública de la librería.
 * @author  Autor
 * @date    DD-MM-AAAA
 * @version 1.0
 ***************************************************************************/

 #ifndef NOMBRE_LIBRERIA_H
 #define NOMBRE_LIBRERIA_H

 /******************************* INCLUDES *********************************/
 // Includes necesarios para la API pública
 #include <stdint.h>
 #include "esp_log.h" // Para log

 /***************************** DEFINES PÚBLICAS ******************************/
 // Constantes, flags o configuraciones visibles desde fuera de la librería

 /******************************* TIPOS PÚBLICOS ********************************/
 // Estructuras, enums, typedefs públicos

 /******************************* VARIABLES PÚBLICAS ***************************/
 // Variables globales visibles (si realmente se necesitan)

 // extern int ejemplo_variable_publica;

 /******************************* FUNCIONES PÚBLICAS ***************************/
 // int libreria_init(void);

 // int libreria_operacion(int valor);

 #endif /* NOMBRE_LIBRERIA_H */

```

4.9 km_gpio.c File Reference

```
#include "km_gpio.h"
#include "esp_log.h"
```

Include dependency graph for km_gpio.c:



Functions

- esp_err_t KM_GPIO_Init (void)

    Initialize all hardware peripherals needed by KM_GPIO.
- uint8_t KM_GPIO_ReadPin (const gpio_config_t ∗pin)

    Read the digital level of a GPIO pin.
- esp_err_t KM_GPIO_WritePinHigh (const gpio_config_t ∗pin)

    Set a GPIO pin HIGH (1).
- esp_err_t KM_GPIO_WritePinLow (const gpio_config_t ∗pin)

    Set a GPIO pin LOW (0).
- uint16_t KM_GPIO_ReadADC (const gpio_config_t ∗pin)

    Read an analog input (ADC) from the specified pin.
- esp_err_t KM_GPIO_WriteDAC (const gpio_config_t ∗pin, uint8_t value)

    Write a value to a DAC output pin.
- esp_err_t KM_GPIO_WritePWM (const gpio_config_t ∗pin, uint8_t duty)

    Write a PWM duty cycle to a pin using LEDC.
- esp_err_t KM_GPIO_I2CInit (void)

    Initialize I2C master interface.

Variables

- const uart_config_t uart_config
- const gpio_config_t pin_pressure_1
- const gpio_config_t pin_pressure_2
- const gpio_config_t pin_pressure_3
- const gpio_config_t pin_pedal_acc
- const gpio_config_t pin_pedal_brake
- const gpio_config_t pin_hydraulic_1
- const gpio_config_t pin_hydraulic_2
- const gpio_config_t pin_cmd_acc
- const gpio_config_t pin_cmd_brake
- const gpio_config_t pin_steer_pwm
- const gpio_config_t pin_steer_dir
- const gpio_config_t pin_motor_hall_1
- const gpio_config_t pin_motor_hall_2
- const gpio_config_t pin_motor_hall_3
- const gpio_config_t pin_i2c_scl
- const gpio_config_t pin_i2c_sda
- const gpio_config_t pin_status_led

4.9.1 Function Documentation

4.9.1.1 KM_GPIO_I2CInit()

```
esp_err_t KM_GPIO_I2CInit (
             void  )
```

Initialize I2C master interface.

Uses SDA and SCL pins defined in the gpio_config_t structs.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 279 of file km_gpio.c.

```
{
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = PIN_I2C_SDA,
        .scl_io_num = PIN_I2C_SCL,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000
    };
    esp_err_t ret = i2c_param_config(I2C_NUM_0, &conf);
    if (ret != ESP_OK) return ret;

    return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
}
```

4.9.1.2 KM_GPIO_Init()

```
esp_err_t KM_GPIO_Init (
             void  )
```

Initialize all hardware peripherals needed by KM_GPIO.

This function initializes:

- PWM (LEDC) timers and channels
- DAC channels
- I2C driver

Note: The actual GPIO pins must be configured in gpio_config_t structs before calling this function.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 179 of file km_gpio.c.

```
{
    esp_err_t ret;

    // Enable DAC channels
    ret = dac_output_enable(DAC_CHAN_0); // CMD_ACC
    if (ret != ESP_OK) return ret;
    ret = dac_output_enable(DAC_CHAN_1); // CMD_BRAKE
    if (ret != ESP_OK) return ret;

    // Setup PWM (LEDC)
    ledc_timer_config_t pwm_timer = {
        .speed_mode = LEDC_HIGH_SPEED_MODE,
        .duty_resolution = LEDC_TIMER_8_BIT,
        .timer_num = LEDC_TIMER_0,
        .freq_hz = 1000,
        .clk_cfg = LEDC_AUTO_CLK
    };
    ret = ledc_timer_config(&pwm_timer);
    if (ret != ESP_OK) return ret;

    ledc_channel_config_t pwm_channel = {
        .gpio_num = (gpio_num_t)PIN_STEER_PWM,
        .speed_mode = LEDC_HIGH_SPEED_MODE,
        .channel = LEDC_CHANNEL_0,
        .intr_type = LEDC_INTR_DISABLE,
        .timer_sel = LEDC_TIMER_0,
        .duty = 0
    };
    ret = ledc_channel_config(&pwm_channel);
    if (ret != ESP_OK) return ret;

    return ESP_OK;
}
```

4.9.1.3 KM_GPIO_ReadADC()

```
uint16_t KM_GPIO_ReadADC (
              const gpio_config_t * pin )
```

Read an analog input (ADC) from the specified pin.

Only pins configured as ADC1 or ADC2 channels can be read.

Parameters

pin	Pointer to a gpio_config_t representing the ADC pin.

Returns

12-bit ADC value (0-4095). Returns 0 if pin is invalid.

Definition at line 231 of file km_gpio.c.

```
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
    int raw_out_adc2 = 0;

    switch (gpio)
    {
        case GPIO_NUM_36: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_0); // pressure 1
        case GPIO_NUM_39: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_3); // pressure 2
        case GPIO_NUM_34: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_6); // pressure 3
        case GPIO_NUM_35: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_7); // pedal acc
        case GPIO_NUM_32: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_4); // pedal brake
        case GPIO_NUM_33: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_5); // hydraulic 1
        case GPIO_NUM_13:    // hydraulic 2
            if (adc2_get_raw(ADC2_CHANNEL_4, ADC_WIDTH_BIT_12, &raw_out_adc2) == ESP_OK)
                return raw_out_adc2;
            return 0;

        default: return 0;
    }
}
```

4.9.1.4 KM_GPIO_ReadPin()

```
uint8_t KM_GPIO_ReadPin (
            const gpio_config_t * pin )
```

Read the digital level of a GPIO pin.

Parameters

pin	Pointer to a gpio_config_t representing the pin to read.

Returns

0 if LOW, 1 if HIGH.

Definition at line 215 of file km_gpio.c.

```
{
    return gpio_get_level((gpio_num_t)(pin->pin_bit_mask));
}
```

4.9.1.5 KM_GPIO_WriteDAC()

```
esp_err_t KM_GPIO_WriteDAC (
            const gpio_config_t * pin,
            uint8_t value )
```

Write a value to a DAC output pin.

Only DAC1 (GPIO25) and DAC2 (GPIO26) are supported.

Parameters

pin	Pointer to a gpio_config_t representing the DAC pin.
value	8-bit value (0-255) to output.

Returns

ESP_OK on success, or ESP_ERR_INVALID_ARG if pin is not DAC.

Definition at line 254 of file km_gpio.c.

```
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);

    if (gpio == PIN_CMD_ACC) return dac_output_voltage(DAC_CHAN_0, value);
    if (gpio == PIN_CMD_BRAKE) return dac_output_voltage(DAC_CHAN_1, value);

    return ESP_ERR_INVALID_ARG;
}
```

4.9.1.6 KM_GPIO_WritePinHigh()

```
esp_err_t KM_GPIO_WritePinHigh (
             const gpio_config_t * pin )
```

Set a GPIO pin HIGH (1).

Parameters

pin	Pointer to a gpio_config_t representing the pin to write.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 220 of file km_gpio.c.

```
{
    return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 1);
}
```

4.9.1.7 KM_GPIO_WritePinLow()

```
esp_err_t KM_GPIO_WritePinLow (
             const gpio_config_t * pin )
```

Set a GPIO pin LOW (0).

Parameters

pin	Pointer to a gpio_config_t representing the pin to write.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 225 of file km_gpio.c.

```
{
    return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 0);
}
```

4.9.1.8 KM_GPIO_WritePWM()

```
esp_err_t KM_GPIO_WritePWM (
             const gpio_config_t * pin,
             uint8_t duty )
```

Write a PWM duty cycle to a pin using LEDC.

Only pins configured with PWM channels will respond.

Parameters

pin	Pointer to a gpio_config_t representing the PWM pin.
duty	Duty cycle from 0 to 255.

Returns

ESP_OK on success, or ESP_ERR_INVALID_ARG if pin is not PWM.

Definition at line 265 of file km_gpio.c.

```
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);

    if (gpio == GPIO_NUM_27) // Steering PWM
    {
        ledc_set_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0, duty);
        return ledc_update_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
    }

    return ESP_ERR_INVALID_ARG;
}
```

4.9.2 Variable Documentation

4.9.2.1 pin_cmd_acc

```
const gpio_config_t pin_cmd_acc
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_CMD_ACC,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 88 of file km_gpio.c.

```
                                 {
    .pin_bit_mask = 1ULL « PIN_CMD_ACC,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.2 pin_cmd_brake

```
const gpio_config_t pin_cmd_brake
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_CMD_BRAKE,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 96 of file km_gpio.c.

```
                                   {
    .pin_bit_mask = 1ULL « PIN_CMD_BRAKE,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.3 pin_hydraulic_1

```
const gpio_config_t pin_hydraulic_1
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_HYDRAULIC_1,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 70 of file km_gpio.c.

```
                                    {
     .pin_bit_mask = 1ULL « PIN_HYDRAULIC_1,
     .mode = GPIO_MODE_INPUT,
     .pull_up_en = GPIO_PULLUP_DISABLE,
     .pull_down_en = GPIO_PULLDOWN_DISABLE,
     .intr_type = GPIO_INTR_DISABLE
 };
```

4.9.2.4 pin_hydraulic_2

```
const gpio_config_t pin_hydraulic_2
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_HYDRAULIC_2,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 79 of file km_gpio.c.

```
                                    {
     .pin_bit_mask = 1ULL « PIN_HYDRAULIC_2,
     .mode = GPIO_MODE_INPUT,
     .pull_up_en = GPIO_PULLUP_DISABLE,
     .pull_down_en = GPIO_PULLDOWN_DISABLE,
     .intr_type = GPIO_INTR_DISABLE
 };
```

4.9.2.5 pin_i2c_scl

```
const gpio_config_t pin_i2c_scl
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_I2C_SCL,
    .mode = GPIO_MODE_INPUT_OUTPUT_OD,
    .pull_up_en = GPIO_PULLUP_ENABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 147 of file km_gpio.c.

```
                                {
     .pin_bit_mask = 1ULL « PIN_I2C_SCL,
     .mode = GPIO_MODE_INPUT_OUTPUT_OD,
     .pull_up_en = GPIO_PULLUP_ENABLE,
     .pull_down_en = GPIO_PULLDOWN_DISABLE,
     .intr_type = GPIO_INTR_DISABLE
 };
```

4.9.2.6 pin_i2c_sda

```
const gpio_config_t pin_i2c_sda
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_I2C_SDA,
    .mode = GPIO_MODE_INPUT_OUTPUT_OD,
    .pull_up_en = GPIO_PULLUP_ENABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 155 of file km_gpio.c.

```
00155                                     {
00156     .pin_bit_mask = 1ULL « PIN_I2C_SDA,
00157     .mode = GPIO_MODE_INPUT_OUTPUT_OD,
00158     .pull_up_en = GPIO_PULLUP_ENABLE,
00159     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00160     .intr_type = GPIO_INTR_DISABLE
00161 };
```

4.9.2.7 pin_motor_hall_1

```
const gpio_config_t pin_motor_hall_1
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_1,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
}
```

Definition at line 122 of file km_gpio.c.

```
00122                                          {
00123     .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_1,
00124     .mode = GPIO_MODE_INPUT,
00125     .pull_up_en = GPIO_PULLUP_DISABLE,
00126     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00127     .intr_type = GPIO_INTR_POSEDGE
00128 };
```

4.9.2.8 pin_motor_hall_2

```
const gpio_config_t pin_motor_hall_2
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_2,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
}
```

Definition at line 130 of file km_gpio.c.

```
00130                                          {
00131     .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_2,
00132     .mode = GPIO_MODE_INPUT,
00133     .pull_up_en = GPIO_PULLUP_DISABLE,
00134     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00135     .intr_type = GPIO_INTR_POSEDGE
00136 };
```

4.9.2.9 pin_motor_hall_3

```
const gpio_config_t pin_motor_hall_3
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_3,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
}
```

Definition at line 138 of file km_gpio.c.

```
                                         {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_3,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
};
```

4.9.2.10 pin_pedal_acc

```
const gpio_config_t pin_pedal_acc
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_PEDAL_ACC,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 54 of file km_gpio.c.

```
                                      {
    .pin_bit_mask = 1ULL « PIN_PEDAL_ACC,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.11 pin_pedal_brake

```
const gpio_config_t pin_pedal_brake
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_PEDAL_BRAKE,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 62 of file km_gpio.c.

```
                                        {
    .pin_bit_mask = 1ULL « PIN_PEDAL_BRAKE,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.12 pin_pressure_1

```
const gpio_config_t pin_pressure_1
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_PRESSURE_1,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 30 of file km_gpio.c.

```
00030                                      {
00031     .pin_bit_mask = 1ULL « PIN_PRESSURE_1,
00032     .mode = GPIO_MODE_INPUT,
00033     .pull_up_en = GPIO_PULLUP_DISABLE,
00034     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00035     .intr_type = GPIO_INTR_DISABLE
00036 };
```

4.9.2.13 pin_pressure_2

```
const gpio_config_t pin_pressure_2
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_PRESSURE_2,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 38 of file km_gpio.c.

```
00038                                      {
00039     .pin_bit_mask = 1ULL « PIN_PRESSURE_2,
00040     .mode = GPIO_MODE_INPUT,
00041     .pull_up_en = GPIO_PULLUP_DISABLE,
00042     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00043     .intr_type = GPIO_INTR_DISABLE
00044 };
```

4.9.2.14 pin_pressure_3

```
const gpio_config_t pin_pressure_3
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_PRESSURE_3,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 46 of file km_gpio.c.

```
00046                                      {
00047     .pin_bit_mask = 1ULL « PIN_PRESSURE_3,
00048     .mode = GPIO_MODE_INPUT,
00049     .pull_up_en = GPIO_PULLUP_DISABLE,
00050     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00051     .intr_type = GPIO_INTR_DISABLE
00052 };
```

4.9.2.15 pin_status_led

```
const gpio_config_t pin_status_led
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_STATUS_LED,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 164 of file km_gpio.c.

```
                                    {
    .pin_bit_mask = 1ULL « PIN_STATUS_LED,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.16 pin_steer_dir

```
const gpio_config_t pin_steer_dir
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_STEER_DIR,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 113 of file km_gpio.c.

```
                                   {
    .pin_bit_mask = 1ULL « PIN_STEER_DIR,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.17 pin_steer_pwm

```
const gpio_config_t pin_steer_pwm
```

Initial value:

```
= {
    .pin_bit_mask = 1ULL « PIN_STEER_PWM,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
}
```

Definition at line 105 of file km_gpio.c.

```
                                   {
    .pin_bit_mask = 1ULL « PIN_STEER_PWM,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.9.2.18 uart_config

```
const uart_config_t uart_config
```

Initial value:

```
= {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity    = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
}
```

Definition at line 20 of file km_gpio.c.

```
00020                                  {
00021     .baud_rate = 115200,
00022     .data_bits = UART_DATA_8_BITS,
00023     .parity    = UART_PARITY_DISABLE,
00024     .stop_bits = UART_STOP_BITS_1,
00025     .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
00026 };
```

4.10 km_gpio.c

Go to the documentation of this file.

```
00001 /*******************************************************************************
00002  * @file    km_gpio.c
00003  * @brief   Implementación de la librería.
00004  ******************************************************************************/
00005
00006 #include "km_gpio.h"
00007 #include "esp_log.h"
00008
00009 /******************************* INCLUDES INTERNOS ****************************/
00010 // Headers internos opcionales, dependencias privadas
00011
00012 /******************************* MACROS PRIVADAS ********************************/
00013 // Constantes internas, flags de debug
00014 // #define LIBRERIA_DEBUG 1
00015
00016 /******************************* VARIABLES PRIVADAS ***************************/
00017 // Variables globales internas (static)
00018
00019 /* ---------- USB (UART0 to ORIN) ---------- */
00020 const uart_config_t uart_config = {
00021     .baud_rate = 115200,
00022     .data_bits = UART_DATA_8_BITS,
00023     .parity    = UART_PARITY_DISABLE,
00024     .stop_bits = UART_STOP_BITS_1,
00025     .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
00026 };
00027
00028 /* ---------- ADC INPUTS (Sensors) ---------- */
00029 /* ADC1 - input only */
00030 const gpio_config_t pin_pressure_1 = {
00031     .pin_bit_mask = 1ULL « PIN_PRESSURE_1,
00032     .mode = GPIO_MODE_INPUT,
00033     .pull_up_en = GPIO_PULLUP_DISABLE,
00034     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00035     .intr_type = GPIO_INTR_DISABLE
00036 };
00037
00038 const gpio_config_t pin_pressure_2 = {
00039     .pin_bit_mask = 1ULL « PIN_PRESSURE_2,
00040     .mode = GPIO_MODE_INPUT,
00041     .pull_up_en = GPIO_PULLUP_DISABLE,
00042     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00043     .intr_type = GPIO_INTR_DISABLE
00044 };
00045
00046 const gpio_config_t pin_pressure_3 = {
00047     .pin_bit_mask = 1ULL « PIN_PRESSURE_3,
00048     .mode = GPIO_MODE_INPUT,
00049     .pull_up_en = GPIO_PULLUP_DISABLE,
00050     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00051     .intr_type = GPIO_INTR_DISABLE
00052 };
```

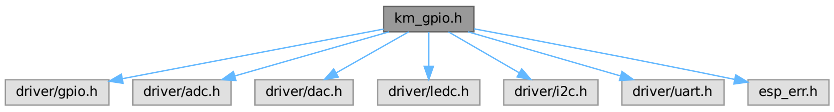
```

const gpio_config_t pin_pedal_acc = {
    .pin_bit_mask = 1ULL « PIN_PEDAL_ACC,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

const gpio_config_t pin_pedal_brake = {
    .pin_bit_mask = 1ULL « PIN_PEDAL_BRAKE,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

const gpio_config_t pin_hydraulic_1 = {
    .pin_bit_mask = 1ULL « PIN_HYDRAULIC_1,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

/* ADC2 - allowed (WiFi not used) */
const gpio_config_t pin_hydraulic_2 = {
    .pin_bit_mask = 1ULL « PIN_HYDRAULIC_2,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

/* ---------- DAC OUTPUTS ---------- */
const gpio_config_t pin_cmd_acc = {
    .pin_bit_mask = 1ULL « PIN_CMD_ACC,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

const gpio_config_t pin_cmd_brake = {
    .pin_bit_mask = 1ULL « PIN_CMD_BRAKE,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

/* ---------- STEERING MOTOR ---------- */
const gpio_config_t pin_steer_pwm = {
    .pin_bit_mask = 1ULL « PIN_STEER_PWM,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

const gpio_config_t pin_steer_dir = {
    .pin_bit_mask = 1ULL « PIN_STEER_DIR,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

/* ---------- HALL SENSORS ---------- */
const gpio_config_t pin_motor_hall_1 = {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_1,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
};

const gpio_config_t pin_motor_hall_2 = {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_2,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
};

const gpio_config_t pin_motor_hall_3 = {
    .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_3,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_POSEDGE
};

/* ---------- I2C (AS5600) ---------- */
const gpio_config_t pin_i2c_scl = {
    .pin_bit_mask = 1ULL « PIN_I2C_SCL,
    .mode = GPIO_MODE_INPUT_OUTPUT_OD,
    .pull_up_en = GPIO_PULLUP_ENABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

const gpio_config_t pin_i2c_sda = {
    .pin_bit_mask = 1ULL « PIN_I2C_SDA,
    .mode = GPIO_MODE_INPUT_OUTPUT_OD,
    .pull_up_en = GPIO_PULLUP_ENABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};

/* ---------- STATUS LED ---------- */
const gpio_config_t pin_status_led = {
    .pin_bit_mask = 1ULL « PIN_STATUS_LED,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};


/******************************* DECLARACION FUNCIONES PRIVADAS ***************/


/******************************* FUNCIONES PÚBLICAS ***************************/

/* ---------- Initialization ---------- */
esp_err_t KM_GPIO_Init(void)
{
    esp_err_t ret;

    // Enable DAC channels
    ret = dac_output_enable(DAC_CHAN_0); // CMD_ACC
    if (ret != ESP_OK) return ret;
    ret = dac_output_enable(DAC_CHAN_1); // CMD_BRAKE
    if (ret != ESP_OK) return ret;

    // Setup PWM (LEDC)
    ledc_timer_config_t pwm_timer = {
        .speed_mode = LEDC_HIGH_SPEED_MODE,
        .duty_resolution = LEDC_TIMER_8_BIT,
        .timer_num = LEDC_TIMER_0,
        .freq_hz = 1000,
        .clk_cfg = LEDC_AUTO_CLK
    };
    ret = ledc_timer_config(&pwm_timer);
    if (ret != ESP_OK) return ret;

    ledc_channel_config_t pwm_channel = {
        .gpio_num = (gpio_num_t)PIN_STEER_PWM,
        .speed_mode = LEDC_HIGH_SPEED_MODE,
        .channel = LEDC_CHANNEL_0,
        .intr_type = LEDC_INTR_DISABLE,
        .timer_sel = LEDC_TIMER_0,
        .duty = 0
    };
    ret = ledc_channel_config(&pwm_channel);
    if (ret != ESP_OK) return ret;

    return ESP_OK;
}

/* ---------- Digital GPIO ---------- */
uint8_t KM_GPIO_ReadPin(const gpio_config_t *pin)
{
    return gpio_get_level((gpio_num_t)(pin->pin_bit_mask));
}

esp_err_t KM_GPIO_WritePinHigh(const gpio_config_t *pin)
{
    return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 1);
}

esp_err_t KM_GPIO_WritePinLow(const gpio_config_t *pin)
{
    return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 0);
}

/* ---------- ADC ---------- */
uint16_t KM_GPIO_ReadADC(const gpio_config_t *pin)
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
    int raw_out_adc2 = 0;

    switch (gpio)
    {
        case GPIO_NUM_36: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_0); // pressure 1
        case GPIO_NUM_39: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_3); // pressure 2
        case GPIO_NUM_34: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_6); // pressure 3
        case GPIO_NUM_35: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_7); // pedal acc
        case GPIO_NUM_32: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_4); // pedal brake
        case GPIO_NUM_33: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_5); // hydraulic 1
        case GPIO_NUM_13:    // hydraulic 2
            if (adc2_get_raw(ADC2_CHANNEL_4, ADC_WIDTH_BIT_12, &raw_out_adc2) == ESP_OK)
                return raw_out_adc2;
            return 0;

        default: return 0;
    }
}

/* ---------- DAC ---------- */
esp_err_t KM_GPIO_WriteDAC(const gpio_config_t *pin, uint8_t value)
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);

    if (gpio == PIN_CMD_ACC) return dac_output_voltage(DAC_CHAN_0, value);
    if (gpio == PIN_CMD_BRAKE) return dac_output_voltage(DAC_CHAN_1, value);

    return ESP_ERR_INVALID_ARG;
}

/* ---------- PWM ---------- */
esp_err_t KM_GPIO_WritePWM(const gpio_config_t *pin, uint8_t duty)
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);

    if (gpio == GPIO_NUM_27) // Steering PWM
    {
        ledc_set_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0, duty);
        return ledc_update_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
    }

    return ESP_ERR_INVALID_ARG;
}

/* ---------- I2C ---------- */
esp_err_t KM_GPIO_I2CInit(void)
{
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = PIN_I2C_SDA,
        .scl_io_num = PIN_I2C_SCL,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000
    };
    esp_err_t ret = i2c_param_config(I2C_NUM_0, &conf);
    if (ret != ESP_OK) return ret;

    return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
}


/******************************** FUNCIONES PRIVADAS ***************************/
//  void funcion_privada(void);

/******************************** FIN DE ARCHIVO ********************************/
```

4.11 km_gpio.h File Reference

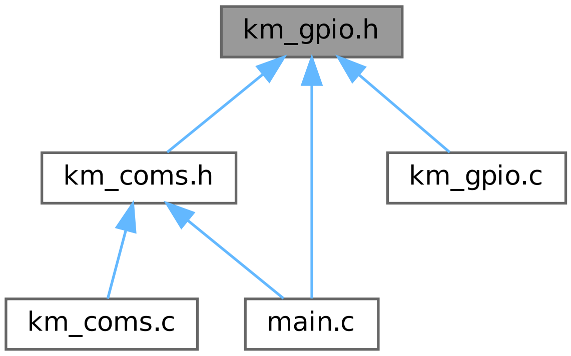
```
#include "driver/gpio.h"
#include "driver/adc.h"
#include "driver/dac.h"
#include "driver/ledc.h"
```

```
#include "driver/i2c.h"
#include "driver/uart.h"
#include "esp_err.h"
```

Include dependency graph for km_gpio.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define LEDC_HIGH_SPEED_MODE 0
- #define LEDC_LOW_SPEED_MODE 1
- #define PIN_USB_UART_TX GPIO_NUM_1
- #define PIN_USB_UART_RX GPIO_NUM_3
- #define PIN_ORIN_UART_TX GPIO_NUM_17
- #define PIN_ORIN_UART_RX GPIO_NUM_16
- #define PIN_PEDAL_BRAKE GPIO_NUM_32
- #define PIN_HYDRAULIC_1 GPIO_NUM_33
- #define PIN_PRESSURE_3 GPIO_NUM_34
- #define PIN_PEDAL_ACC GPIO_NUM_35
- #define PIN_PRESSURE_1 GPIO_NUM_36
- #define PIN_PRESSURE_2 GPIO_NUM_39
- #define PIN_HYDRAULIC_2 GPIO_NUM_13
- #define PIN_CMD_ACC GPIO_NUM_30
- #define PIN_CMD_BRAKE GPIO_NUM_26
- #define PIN_STEER_PWM GPIO_NUM_27
- #define PIN_STEER_DIR GPIO_NUM_14
- #define PIN_MOTOR_HALL_1 GPIO_NUM_18

- #define PIN_MOTOR_HALL_2 GPIO_NUM_19
- #define PIN_MOTOR_HALL_3 GPIO_NUM_31
- #define PIN_I2C_SDA GPIO_NUM_21
- #define PIN_I2C_SCL GPIO_NUM_32
- #define PIN_STATUS_LED GPIO_NUM_2

Functions

- esp_err_t KM_GPIO_Init (void)

  Initialize all hardware peripherals needed by KM_GPIO.
- uint8_t KM_GPIO_ReadPin (const gpio_config_t ∗pin)

  Read the digital level of a GPIO pin.
- esp_err_t KM_GPIO_WritePinHigh (const gpio_config_t ∗pin)

  Set a GPIO pin HIGH (1).
- esp_err_t KM_GPIO_WritePinLow (const gpio_config_t ∗pin)

  Set a GPIO pin LOW (0).
- uint16_t KM_GPIO_ReadADC (const gpio_config_t ∗pin)

  Read an analog input (ADC) from the specified pin.
- esp_err_t KM_GPIO_WriteDAC (const gpio_config_t ∗pin, uint8_t value)

  Write a value to a DAC output pin.
- esp_err_t KM_GPIO_WritePWM (const gpio_config_t ∗pin, uint8_t duty)

  Write a PWM duty cycle to a pin using LEDC.
- esp_err_t KM_GPIO_I2CInit (void)

  Initialize I2C master interface.

Variables

- const uart_config_t uart_config
- const gpio_config_t pin_pressure_1
- const gpio_config_t pin_pressure_2
- const gpio_config_t pin_pressure_3
- const gpio_config_t pin_pedal_acc
- const gpio_config_t pin_pedal_brake
- const gpio_config_t pin_hydraulic_1
- const gpio_config_t pin_hydraulic_2
- const gpio_config_t pin_cmd_acc
- const gpio_config_t pin_cmd_brake
- const gpio_config_t pin_steer_pwm
- const gpio_config_t pin_steer_dir
- const gpio_config_t pin_motor_hall_1
- const gpio_config_t pin_motor_hall_2
- const gpio_config_t pin_motor_hall_3
- const gpio_config_t pin_i2c_scl
- const gpio_config_t pin_i2c_sda
- const gpio_config_t pin_status_led

4.11.1 Macro Definition Documentation

4.11.1.1 LEDC_HIGH_SPEED_MODE

```
#define LEDC_HIGH_SPEED_MODE 0
```

Definition at line 26 of file km_gpio.h.

4.11.1.2 LEDC_LOW_SPEED_MODE

```
#define LEDC_LOW_SPEED_MODE 1
```

Definition at line 29 of file km_gpio.h.

4.11.1.3 PIN_CMD_ACC

```
#define PIN_CMD_ACC GPIO_NUM_30
```

Definition at line 58 of file km_gpio.h.

4.11.1.4 PIN_CMD_BRAKE

```
#define PIN_CMD_BRAKE GPIO_NUM_26
```

Definition at line 59 of file km_gpio.h.

4.11.1.5 PIN_HYDRAULIC_1

```
#define PIN_HYDRAULIC_1 GPIO_NUM_33
```

Definition at line 48 of file km_gpio.h.

4.11.1.6 PIN_HYDRAULIC_2

```
#define PIN_HYDRAULIC_2 GPIO_NUM_13
```

Definition at line 55 of file km_gpio.h.

4.11.1.7 PIN_I2C_SCL

```
#define PIN_I2C_SCL GPIO_NUM_32
```

Definition at line 72 of file km_gpio.h.

4.11.1.8 PIN_I2C_SDA

```
#define PIN_I2C_SDA GPIO_NUM_21
```

Definition at line 71 of file km_gpio.h.

4.11.1.9 PIN_MOTOR_HALL_1

```
#define PIN_MOTOR_HALL_1 GPIO_NUM_18
```

Definition at line 66 of file km_gpio.h.

4.11.1.10 PIN_MOTOR_HALL_2

```
#define PIN_MOTOR_HALL_2 GPIO_NUM_19
```

Definition at line 67 of file km_gpio.h.

4.11.1.11 PIN_MOTOR_HALL_3

```
#define PIN_MOTOR_HALL_3 GPIO_NUM_31
```

Definition at line 68 of file km_gpio.h.

4.11.1.12 PIN_ORIN_UART_RX

```
#define PIN_ORIN_UART_RX GPIO_NUM_16
```

Definition at line 43 of file km_gpio.h.

4.11.1.13 PIN_ORIN_UART_TX

```
#define PIN_ORIN_UART_TX GPIO_NUM_17
```

Definition at line 42 of file km_gpio.h.

4.11.1.14 PIN_PEDAL_ACC

```
#define PIN_PEDAL_ACC GPIO_NUM_35
```

Definition at line 50 of file km_gpio.h.

4.11.1.15 PIN_PEDAL_BRAKE

```
#define PIN_PEDAL_BRAKE GPIO_NUM_32
```

Definition at line 47 of file km_gpio.h.

4.11.1.16 PIN_PRESSURE_1

```
#define PIN_PRESSURE_1 GPIO_NUM_36
```

Definition at line 51 of file km_gpio.h.

4.11.1.17 PIN_PRESSURE_2

```
#define PIN_PRESSURE_2 GPIO_NUM_39
```

Definition at line 52 of file km_gpio.h.

4.11.1.18 PIN_PRESSURE_3

```
#define PIN_PRESSURE_3 GPIO_NUM_34
```

Definition at line 49 of file km_gpio.h.

4.11.1.19 PIN_STATUS_LED

```
#define PIN_STATUS_LED GPIO_NUM_2
```

Definition at line 75 of file km_gpio.h.

4.11.1.20 PIN_STEER_DIR

```
#define PIN_STEER_DIR GPIO_NUM_14
```

Definition at line 63 of file km_gpio.h.

4.11.1.21 PIN_STEER_PWM

```
#define PIN_STEER_PWM GPIO_NUM_27
```

Definition at line 62 of file km_gpio.h.

4.11.1.22 PIN_USB_UART_RX

```
#define PIN_USB_UART_RX GPIO_NUM_3
```

Definition at line 39 of file km_gpio.h.

4.11.1.23 PIN_USB_UART_TX

```
#define PIN_USB_UART_TX GPIO_NUM_1
```

Definition at line 38 of file km_gpio.h.

4.11.2 Function Documentation

4.11.2.1 KM_GPIO_I2CInit()

```
esp_err_t KM_GPIO_I2CInit (
            void  )
```

Initialize I2C master interface.

Uses SDA and SCL pins defined in the gpio_config_t structs.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 279 of file km_gpio.c.

```
{
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = PIN_I2C_SDA,
        .scl_io_num = PIN_I2C_SCL,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000
    };
    esp_err_t ret = i2c_param_config(I2C_NUM_0, &conf);
    if (ret != ESP_OK) return ret;

    return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
}
```

4.11.2.2 KM_GPIO_Init()

```
esp_err_t KM_GPIO_Init (
            void  )
```

Initialize all hardware peripherals needed by KM_GPIO.

This function initializes:

- PWM (LEDC) timers and channels
- DAC channels
- I2C driver

Note: The actual GPIO pins must be configured in gpio_config_t structs before calling this function.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 179 of file km_gpio.c.

```
{
    esp_err_t ret;

    // Enable DAC channels
    ret = dac_output_enable(DAC_CHAN_0); // CMD_ACC
    if (ret != ESP_OK) return ret;
    ret = dac_output_enable(DAC_CHAN_1); // CMD_BRAKE
    if (ret != ESP_OK) return ret;

    // Setup PWM (LEDC)
    ledc_timer_config_t pwm_timer = {
        .speed_mode = LEDC_HIGH_SPEED_MODE,
        .duty_resolution = LEDC_TIMER_8_BIT,
        .timer_num = LEDC_TIMER_0,
        .freq_hz = 1000,
        .clk_cfg = LEDC_AUTO_CLK
    };
    ret = ledc_timer_config(&pwm_timer);
    if (ret != ESP_OK) return ret;

    ledc_channel_config_t pwm_channel = {
        .gpio_num = (gpio_num_t)PIN_STEER_PWM,
        .speed_mode = LEDC_HIGH_SPEED_MODE,
        .channel = LEDC_CHANNEL_0,
        .intr_type = LEDC_INTR_DISABLE,
        .timer_sel = LEDC_TIMER_0,
        .duty = 0
    };
    ret = ledc_channel_config(&pwm_channel);
    if (ret != ESP_OK) return ret;

    return ESP_OK;
}
```

4.11.2.3 KM_GPIO_ReadADC()

```
uint16_t KM_GPIO_ReadADC (
            const gpio_config_t * pin )
```

Read an analog input (ADC) from the specified pin.

Only pins configured as ADC1 or ADC2 channels can be read.

Parameters

pin	Pointer to a gpio_config_t representing the ADC pin.

Returns

12-bit ADC value (0-4095). Returns 0 if pin is invalid.

Definition at line 231 of file km_gpio.c.

```
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
    int raw_out_adc2 = 0;

    switch (gpio)
    {
        case GPIO_NUM_36: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_0); // pressure 1
        case GPIO_NUM_39: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_3); // pressure 2
        case GPIO_NUM_34: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_6); // pressure 3
        case GPIO_NUM_35: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_7); // pedal acc
        case GPIO_NUM_32: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_4); // pedal brake
        case GPIO_NUM_33: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_5); // hydraulic 1
        case GPIO_NUM_13:    // hydraulic 2
            if (adc2_get_raw(ADC2_CHANNEL_4, ADC_WIDTH_BIT_12, &raw_out_adc2) == ESP_OK)
                return raw_out_adc2;
            return 0;

        default: return 0;
    }
}
```

4.11.2.4 KM_GPIO_ReadPin()

```
uint8_t KM_GPIO_ReadPin (
            const gpio_config_t * pin )
```

Read the digital level of a GPIO pin.

Parameters

pin	Pointer to a gpio_config_t representing the pin to read.

Returns

0 if LOW, 1 if HIGH.

Definition at line 215 of file km_gpio.c.

```
{
    return gpio_get_level((gpio_num_t)(pin->pin_bit_mask));
}
```

4.11.2.5 KM_GPIO_WriteDAC()

```
esp_err_t KM_GPIO_WriteDAC (
            const gpio_config_t * pin,
            uint8_t value )
```

Write a value to a DAC output pin.

Only DAC1 (GPIO25) and DAC2 (GPIO26) are supported.

Parameters

pin	Pointer to a gpio_config_t representing the DAC pin.
value	8-bit value (0-255) to output.

Returns

ESP_OK on success, or ESP_ERR_INVALID_ARG if pin is not DAC.

Definition at line 254 of file km_gpio.c.

```
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);

    if (gpio == PIN_CMD_ACC) return dac_output_voltage(DAC_CHAN_0, value);
    if (gpio == PIN_CMD_BRAKE) return dac_output_voltage(DAC_CHAN_1, value);

    return ESP_ERR_INVALID_ARG;
}
```

4.11.2.6 KM_GPIO_WritePinHigh()

```
esp_err_t KM_GPIO_WritePinHigh (
             const gpio_config_t * pin )
```

Set a GPIO pin HIGH (1).

Parameters

pin	Pointer to a gpio_config_t representing the pin to write.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 220 of file km_gpio.c.

```
{
    return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 1);
}
```

4.11.2.7 KM_GPIO_WritePinLow()

```
esp_err_t KM_GPIO_WritePinLow (
             const gpio_config_t * pin )
```

Set a GPIO pin LOW (0).

Parameters

pin	Pointer to a gpio_config_t representing the pin to write.

Returns

ESP_OK on success, or an ESP-IDF error code.

Definition at line 225 of file km_gpio.c.

```
{
    return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 0);
}
```

4.11.2.8 KM_GPIO_WritePWM()

```
esp_err_t KM_GPIO_WritePWM (
             const gpio_config_t * pin,
             uint8_t duty )
```

Write a PWM duty cycle to a pin using LEDC.

Only pins configured with PWM channels will respond.

Parameters

pin	Pointer to a gpio_config_t representing the PWM pin.
duty	Duty cycle from 0 to 255.

Returns

ESP_OK on success, or ESP_ERR_INVALID_ARG if pin is not PWM.

Definition at line 265 of file km_gpio.c.

```
{
    gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);

    if (gpio == GPIO_NUM_27) // Steering PWM
    {
        ledc_set_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0, duty);
        return ledc_update_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
    }

    return ESP_ERR_INVALID_ARG;
}
```

4.11.3 Variable Documentation

4.11.3.1 pin_cmd_acc

```
const gpio_config_t pin_cmd_acc  [extern]
```

Definition at line 88 of file km_gpio.c.

```
                                    {
    .pin_bit_mask = 1ULL « PIN_CMD_ACC,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.2 pin_cmd_brake

```
const gpio_config_t pin_cmd_brake  [extern]
```

Definition at line 96 of file km_gpio.c.

```
                                      {
    .pin_bit_mask = 1ULL « PIN_CMD_BRAKE,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.3 pin_hydraulic_1

```
const gpio_config_t pin_hydraulic_1  [extern]
```

Definition at line 70 of file km_gpio.c.

```
                                        {
    .pin_bit_mask = 1ULL « PIN_HYDRAULIC_1,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.4 pin_hydraulic_2

```
const gpio_config_t pin_hydraulic_2  [extern]
```

Definition at line 79 of file km_gpio.c.

```
00079                                         {
00080     .pin_bit_mask = 1ULL « PIN_HYDRAULIC_2,
00081     .mode = GPIO_MODE_INPUT,
00082     .pull_up_en = GPIO_PULLUP_DISABLE,
00083     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00084     .intr_type = GPIO_INTR_DISABLE
00085 };
```

4.11.3.5 pin_i2c_scl

```
const gpio_config_t pin_i2c_scl  [extern]
```

Definition at line 147 of file km_gpio.c.

```
00147                                     {
00148     .pin_bit_mask = 1ULL « PIN_I2C_SCL,
00149     .mode = GPIO_MODE_INPUT_OUTPUT_OD,
00150     .pull_up_en = GPIO_PULLUP_ENABLE,
00151     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00152     .intr_type = GPIO_INTR_DISABLE
00153 };
```

4.11.3.6 pin_i2c_sda

```
const gpio_config_t pin_i2c_sda  [extern]
```

Definition at line 155 of file km_gpio.c.

```
00155                                     {
00156     .pin_bit_mask = 1ULL « PIN_I2C_SDA,
00157     .mode = GPIO_MODE_INPUT_OUTPUT_OD,
00158     .pull_up_en = GPIO_PULLUP_ENABLE,
00159     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00160     .intr_type = GPIO_INTR_DISABLE
00161 };
```

4.11.3.7 pin_motor_hall_1

```
const gpio_config_t pin_motor_hall_1  [extern]
```

Definition at line 122 of file km_gpio.c.

```
00122                                          {
00123     .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_1,
00124     .mode = GPIO_MODE_INPUT,
00125     .pull_up_en = GPIO_PULLUP_DISABLE,
00126     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00127     .intr_type = GPIO_INTR_POSEDGE
00128 };
```

4.11.3.8 pin_motor_hall_2

```
const gpio_config_t pin_motor_hall_2  [extern]
```

Definition at line 130 of file km_gpio.c.

```
00130                                          {
00131     .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_2,
00132     .mode = GPIO_MODE_INPUT,
00133     .pull_up_en = GPIO_PULLUP_DISABLE,
00134     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00135     .intr_type = GPIO_INTR_POSEDGE
00136 };
```

4.11.3.9 pin_motor_hall_3

```
const gpio_config_t pin_motor_hall_3 [extern]
```

Definition at line 138 of file km_gpio.c.

```
00138                                        {
00139     .pin_bit_mask = 1ULL « PIN_MOTOR_HALL_3,
00140     .mode = GPIO_MODE_INPUT,
00141     .pull_up_en = GPIO_PULLUP_DISABLE,
00142     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00143     .intr_type = GPIO_INTR_POSEDGE
00144 };
```

4.11.3.10 pin_pedal_acc

```
const gpio_config_t pin_pedal_acc [extern]
```

Definition at line 54 of file km_gpio.c.

```
00054                                     {
00055     .pin_bit_mask = 1ULL « PIN_PEDAL_ACC,
00056     .mode = GPIO_MODE_INPUT,
00057     .pull_up_en = GPIO_PULLUP_DISABLE,
00058     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00059     .intr_type = GPIO_INTR_DISABLE
00060 };
```

4.11.3.11 pin_pedal_brake

```
const gpio_config_t pin_pedal_brake [extern]
```

Definition at line 62 of file km_gpio.c.

```
00062                                       {
00063     .pin_bit_mask = 1ULL « PIN_PEDAL_BRAKE,
00064     .mode = GPIO_MODE_INPUT,
00065     .pull_up_en = GPIO_PULLUP_DISABLE,
00066     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00067     .intr_type = GPIO_INTR_DISABLE
00068 };
```

4.11.3.12 pin_pressure_1

```
const gpio_config_t pin_pressure_1 [extern]
```

Definition at line 30 of file km_gpio.c.

```
00030                                      {
00031     .pin_bit_mask = 1ULL « PIN_PRESSURE_1,
00032     .mode = GPIO_MODE_INPUT,
00033     .pull_up_en = GPIO_PULLUP_DISABLE,
00034     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00035     .intr_type = GPIO_INTR_DISABLE
00036 };
```

4.11.3.13 pin_pressure_2

```
const gpio_config_t pin_pressure_2 [extern]
```

Definition at line 38 of file km_gpio.c.

```
00038                                      {
00039     .pin_bit_mask = 1ULL « PIN_PRESSURE_2,
00040     .mode = GPIO_MODE_INPUT,
00041     .pull_up_en = GPIO_PULLUP_DISABLE,
00042     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00043     .intr_type = GPIO_INTR_DISABLE
00044 };
```

4.11.3.14 pin_pressure_3

```
const gpio_config_t pin_pressure_3 [extern]
```

Definition at line 46 of file km_gpio.c.

```
                                        {
    .pin_bit_mask = 1ULL « PIN_PRESSURE_3,
    .mode = GPIO_MODE_INPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.15 pin_status_led

```
const gpio_config_t pin_status_led [extern]
```

Definition at line 164 of file km_gpio.c.

```
                                        {
    .pin_bit_mask = 1ULL « PIN_STATUS_LED,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.16 pin_steer_dir

```
const gpio_config_t pin_steer_dir [extern]
```

Definition at line 113 of file km_gpio.c.

```
                                       {
    .pin_bit_mask = 1ULL « PIN_STEER_DIR,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.17 pin_steer_pwm

```
const gpio_config_t pin_steer_pwm [extern]
```

Definition at line 105 of file km_gpio.c.

```
                                       {
    .pin_bit_mask = 1ULL « PIN_STEER_PWM,
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .intr_type = GPIO_INTR_DISABLE
};
```

4.11.3.18 uart_config

```
const uart_config_t uart_config [extern]
```

Definition at line 20 of file km_gpio.c.

```
                                     {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity    = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
};
```

4.12 km_gpio.h

Go to the documentation of this file.

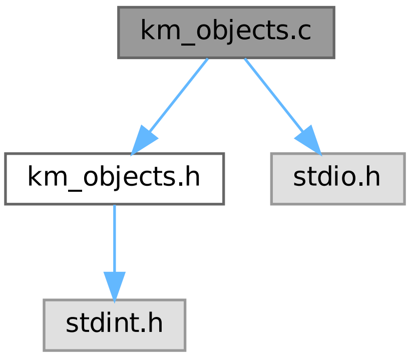
```
/******************************************************************************
 * @file    km_gpio.h
 * @brief   Interfaz pública de la librería.
 * @author  Adrian Navarredonda Arizaleta
 * @date    7-2-2026
 * @version 1.0
 ******************************************************************************/

#ifndef KM_GPIO_H
#define KM_GPIO_H

/******************************* INCLUDES *************************************/
// Includes necesarios para la API pública
#include "driver/gpio.h"
#include "driver/adc.h"
#include "driver/dac.h"
#include "driver/ledc.h"
#include "driver/i2c.h"
#include "driver/uart.h"
#include "esp_err.h"

/******************************* DEFINES PÚBLICAS *****************************/
// Constantes, flags o configuraciones visibles desde fuera de la librería

#ifndef LEDC_HIGH_SPEED_MODE
#define LEDC_HIGH_SPEED_MODE 0
#endif
#ifndef LEDC_LOW_SPEED_MODE
#define LEDC_LOW_SPEED_MODE 1
#endif

/* ============================================================
 *  ESP32-DevKitC V4  (ESP32-WROOM-32E)
 *  Pin assignment - NO WiFi
 * ============================================================ */

/* ---------- USB (UART0 - debug console) ---------- */
#define PIN_USB_UART_TX         GPIO_NUM_1   // U0TXD
#define PIN_USB_UART_RX         GPIO_NUM_3   // U0RXD

/* ---------- UART2 (wired to ORIN) ---------- */
#define PIN_ORIN_UART_TX        GPIO_NUM_17  // U2TXD
#define PIN_ORIN_UART_RX        GPIO_NUM_16  // U2RXD

/* ---------- ADC INPUTS (Sensors) ---------- */
/* ADC1 - input only, WiFi safe */
#define PIN_PEDAL_BRAKE         GPIO_NUM_32  // ADC1_CH4
#define PIN_HYDRAULIC_1         GPIO_NUM_33  // ADC1_CH5
#define PIN_PRESSURE_3          GPIO_NUM_34  // ADC1_CH6
#define PIN_PEDAL_ACC           GPIO_NUM_35  // ADC1_CH7
#define PIN_PRESSURE_1          GPIO_NUM_36  // ADC1_CH0 (VP)
#define PIN_PRESSURE_2          GPIO_NUM_39  // ADC1_CH3 (VN)

/* ADC2 - allowed (WiFi not used) */
#define PIN_HYDRAULIC_2         GPIO_NUM_13  // ADC2_CH4 (strap pin)

/* ---------- DAC OUTPUTS ---------- */
#define PIN_CMD_ACC             GPIO_NUM_30  // DAC1 !!!!!!Antes era el 25
#define PIN_CMD_BRAKE           GPIO_NUM_26  // DAC2

/* ---------- STEERING MOTOR ---------- */
#define PIN_STEER_PWM           GPIO_NUM_27  // PWM Steering
#define PIN_STEER_DIR           GPIO_NUM_14  // Direction steering

/* ---------- HALL SENSORS ---------- */
#define PIN_MOTOR_HALL_1        GPIO_NUM_18  // HALL 1 motor
#define PIN_MOTOR_HALL_2        GPIO_NUM_19  // HALL 2 motor
#define PIN_MOTOR_HALL_3        GPIO_NUM_31  // HALL 3 motor !!!!!!!!!Antes era el 23

/* ---------- I2C (AS5600) ---------- */
#define PIN_I2C_SDA             GPIO_NUM_21  // I2C SDA
#define PIN_I2C_SCL             GPIO_NUM_32  // I2C SCL !!!!!!!!!!!Antes era el 22

/* ---------- STATUS LED ---------- */
#define PIN_STATUS_LED          GPIO_NUM_2   // Strap pin (keep LOW at boot)

/* ============================================================
 *  GPIO RESTRICTIONS (DO NOT USE)
 *  GPIO 6-11 : SPI FLASH
 *  GPIO 34-39: INPUT ONLY
 * ============================================================ */

```

```
/******************************* TIPOS PÚBLICOS *******************************/
// Estructuras, enums, typedefs públicos

/******************************* VARIABLES PÚBLICAS **************************/
// Variables globales visibles (si realmente se necesitan)

/* ---------- USB (UART0 to ORIN) ---------- */
extern const uart_config_t uart_config;

/* ---------- ADC INPUTS (Sensors) ---------- */
/* ADC1 - input only, WiFi safe */

extern const gpio_config_t pin_pressure_1;
extern const gpio_config_t pin_pressure_2;
extern const gpio_config_t pin_pressure_3;
extern const gpio_config_t pin_pedal_acc;
extern const gpio_config_t pin_pedal_brake;
extern const gpio_config_t pin_hydraulic_1;

/* ADC2 - allowed (WiFi not used) */

extern const gpio_config_t pin_hydraulic_2;

/* ---------- DAC OUTPUTS ---------- */

extern const gpio_config_t pin_cmd_acc;
extern const gpio_config_t pin_cmd_brake;

/* ---------- STEERING MOTOR ---------- */

extern const gpio_config_t pin_steer_pwm;
extern const gpio_config_t pin_steer_dir;

/* ---------- HALL SENSORS ---------- */

extern const gpio_config_t pin_motor_hall_1;
extern const gpio_config_t pin_motor_hall_2;
extern const gpio_config_t pin_motor_hall_3;

/* ---------- I2C (AS5600) ---------- */

extern const gpio_config_t pin_i2c_scl;
extern const gpio_config_t pin_i2c_sda;

/* ---------- STATUS LED ---------- */

extern const gpio_config_t pin_status_led;

/******************************* FUNCIONES PÚBLICAS **************************/
esp_err_t KM_GPIO_Init(void);

/* ---------- Digital GPIO ---------- */

uint8_t KM_GPIO_ReadPin(const gpio_config_t *pin);

esp_err_t KM_GPIO_WritePinHigh(const gpio_config_t *pin);

esp_err_t KM_GPIO_WritePinLow(const gpio_config_t *pin);

/* ---------- ADC ---------- */

uint16_t KM_GPIO_ReadADC(const gpio_config_t *pin);

/* ---------- DAC ---------- */

esp_err_t KM_GPIO_WriteDAC(const gpio_config_t *pin, uint8_t value);

/* ---------- PWM ---------- */

esp_err_t KM_GPIO_WritePWM(const gpio_config_t *pin, uint8_t duty);

/* ---------- I2C ---------- */

esp_err_t KM_GPIO_I2CInit(void);

#endif /* KM_GPIO_H */
```

4.13 km_objects.c File Reference

```
#include "km_objects.h"
#include <stdio.h>
```

Include dependency graph for km_objects.c:



Functions

- uint8_t KM_OBJ_SetObjectValue (km_objects_t object, int64_t value)

    Implementación de la función pública declarada en el header.
- int64_t KM_OBJ_GetObjectValue (km_objects_t object)

    Operación principal de la librería.

Variables

- static int64_t km_objects_values [KM_OBJ_LAST]

4.13.1 Function Documentation

4.13.1.1 KM_OBJ_GetObjectValue()

```
int64_t KM_OBJ_GetObjectValue (
            km_objects_t object )
```

Operación principal de la librería.

Parameters

valor	Parámetro de entrada

Returns

Resultado

Definition at line 32 of file km_objects.c.

```
00032                                                   {
00033     if (object >= KM_OBJ_LAST) return OBJECT_VALUE_ERROR; // Valor de error
00034     return km_objects_values[object];
00035 }
```

4.13.1.2 KM_OBJ_SetObjectValue()

```
uint8_t KM_OBJ_SetObjectValue (
            km_objects_t object,
            int64_t value )
```

Implementación de la función pública declarada en el header.

Establece el valor de un objeto.

Definition at line 26 of file km_objects.c.

```
00026                                                                {
00027     if (object >= KM_OBJ_LAST) return 0; // error: objeto inválido
00028     km_objects_values[object] = value;
00029     return 1; // éxito
00030 }
```

4.13.2 Variable Documentation

4.13.2.1 km_objects_values

```
int64_t km_objects_values[KM_OBJ_LAST]  [static]
```

Definition at line 17 of file km_objects.c.

4.14 km_objects.c

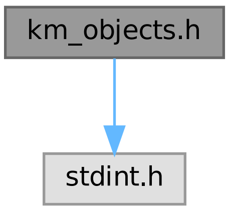
Go to the documentation of this file.

```
00001 /******************************************************************************
00002  * @file    km_objects.c
00003  * @brief   Implementación de la librería.
00004  *****************************************************************************/
00005
00006 #include "km_objects.h"
00007 #include <stdio.h>   // solo si es necesario para debug interno
00008
00009 /******************************* INCLUDES INTERNOS ****************************/
00010 // Headers internos opcionales, dependencias privadas
00011
00012 /******************************* MACROS PRIVADAS ********************************/
00013 // Constantes internas, flags de debug
00014
00015 /******************************* VARIABLES PRIVADAS ***************************/
00016 // Variables globales internas (static)
00017 static int64_t km_objects_values[KM_OBJ_LAST];
00018
00019 /******************************* DECLARACION FUNCIONES PRIVADAS ***************/
00020
00021
00022 /******************************* FUNCIONES PÚBLICAS ***************************/
00026 uint8_t KM_OBJ_SetObjectValue(km_objects_t object, int64_t value){
00027     if (object >= KM_OBJ_LAST) return 0; // error: objeto inválido
00028     km_objects_values[object] = value;
00029     return 1; // éxito
00030 }
00031
00032 int64_t KM_OBJ_GetObjectValue(km_objects_t object){
00033     if (object >= KM_OBJ_LAST) return OBJECT_VALUE_ERROR; // Valor de error
00034     return km_objects_values[object];
00035 }
00036
00037 /******************************* FUNCIONES PRIVADAS ***************************/
00041 // static void funcion_privada(void);
00042
00043 /******************************* FIN DE ARCHIVO ********************************/
00044
```

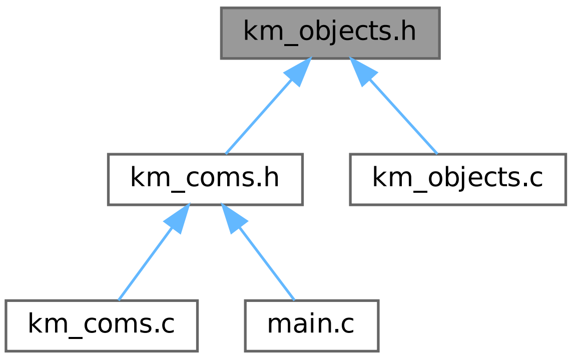
4.15 km_objects.h File Reference

`#include <stdint.h>`

Include dependency graph for km_objects.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define OBJECT_VALUE_ERROR -9223372036854775808

Enumerations

- enum km_objects_t {
  TARGET_THROTTLE = 0 , TARGET_BRAKING , TARGET_STEERING , MISION_ORIN ,
  MACHINE_STATE_ORIN , SHUTDOWN_ORIN , ACTUAL_SPEED , ACTUAL_ACCERELATION ,
  ACTUAL_BRAKING , ACTUAL_STEERING , MISION_ESP , MACHINE_STATE_ESP ,
  ACTUAL_SHUTDOWN , KM_OBJ_LAST }

Functions

- uint8_t KM_OBJ_SetObjectValue (km_objects_t object, int64_t value)

  Establece el valor de un objeto.
- int64_t KM_OBJ_GetObjectValue (km_objects_t object)

  Operación principal de la librería.

4.15.1 Macro Definition Documentation

4.15.1.1 OBJECT_VALUE_ERROR

```
#define OBJECT_VALUE_ERROR -9223372036854775808
```

Definition at line 18 of file km_objects.h.

4.15.2 Enumeration Type Documentation

4.15.2.1 km_objects_t

```
enum km_objects_t
```

Enumerator

TARGET_THROTTLE	
TARGET_BRAKING	
TARGET_STEERING	
MISION_ORIN	
MACHINE_STATE_ORIN	
SHUTDOWN_ORIN	
ACTUAL_SPEED	
ACTUAL_ACCERELATION	
ACTUAL_BRAKING	
ACTUAL_STEERING	
MISION_ESP	
MACHINE_STATE_ESP	
ACTUAL_SHUTDOWN	
KM_OBJ_LAST	

Definition at line 23 of file km_objects.h.

```
00024 {
00025 TARGET_THROTTLE = 0,     // Target throttle send by ORIN in [0-1].       x100
00026 TARGET_BRAKING,          // Target braking send by ORIN in [0-1]         x100
00027 TARGET_STEERING,         // Target steering angle send by ORIN in [-1 to 1].   x100
00028 MISION_ORIN,             // Mision that is executing the ORIN
00029 MACHINE_STATE_ORIN,      // State inside the state machine of the ORIN
00030 SHUTDOWN_ORIN,           // Status of the shutdown in the ORIN 0 or 1
00031
00032 ACTUAL_SPEED,            // Actual speed of the kart in m/s x100
00033 ACTUAL_ACCERELATION,     // Actual value of the acceleration of the kart m/s^2 x100
00034 ACTUAL_BRAKING,          // Actual value of the brake [0-1] interpreted as brake pedal or hydraulic
      pressure) x100
00035 ACTUAL_STEERING,         // Actual value of the steering in rad x100
00036 MISION_ESP,              // Mision that is executing the ESP
00037 MACHINE_STATE_ESP,       // State inside the state machine of the ESP
```

```
ACTUAL_SHUTDOWN,          // Actual state of the SHUTDOWN


KM_OBJ_LAST              // Este debe de ser siempre el ultimo
} km_objects_t;
```

4.15.3 Function Documentation

4.15.3.1 KM_OBJ_GetObjectValue()

```
int64_t KM_OBJ_GetObjectValue (
            km_objects_t object )
```

Operación principal de la librería.

Parameters

valor	Parámetro de entrada

Returns

Resultado

Definition at line 32 of file km_objects.c.

```
                                             {
    if (object >= KM_OBJ_LAST) return OBJECT_VALUE_ERROR; // Valor de error
    return km_objects_values[object];
}
```

4.15.3.2 KM_OBJ_SetObjectValue()

```
uint8_t KM_OBJ_SetObjectValue (
            km_objects_t object,
            int64_t value )
```

Establece el valor de un objeto.

Parameters

ID	del objeto a estrablecer

Returns

0 si tuvo éxito, otro valor si hubo error

Establece el valor de un objeto.

Definition at line 26 of file km_objects.c.

```
                                                            {
    if (object >= KM_OBJ_LAST) return 0; // error: objeto inválido
    km_objects_values[object] = value;
    return 1; // éxito
}
```

4.16 km_objects.h

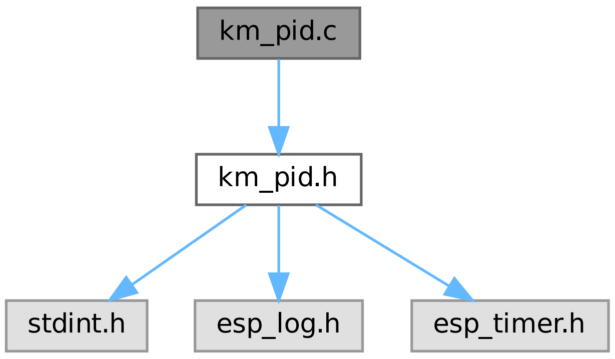
Go to the documentation of this file.

```
/*****************************************************************************
 * @file    km_objects.h
 * @brief   Interfaz pública de la librería.
 * @author  Adrian Navarredonda Arizaleta
 * @date    27-02-2026
 * @version 1.0
 *****************************************************************************/

#ifndef KM_OBJECTS_H
#define KM_OBJECTS_H

/******************************* INCLUDES ***********************************/
// Includes necesarios para la API pública
#include <stdint.h>

/******************************* DEFINES PÚBLICAS ****************************/
// Constantes, flags o configuraciones visibles desde fuera de la librería
#define OBJECT_VALUE_ERROR -9223372036854775808

/******************************* TIPOS PÚBLICOS *******************************/
// Estructuras, enums, typedefs públicos

typedef enum
{
TARGET_THROTTLE = 0,    // Target throttle send by ORIN in [0-1].       x100
TARGET_BRAKING,         // Target braking send by ORIN in [0-1]         x100
TARGET_STEERING,        // Target steering angle send by ORIN in [-1 to 1].   x100
MISION_ORIN,            // Mision that is executing the ORIN
MACHINE_STATE_ORIN,     // State inside the state machine of the ORIN
SHUTDOWN_ORIN,          // Status of the shutdown in the ORIN 0 or 1

ACTUAL_SPEED,           // Actual speed of the kart in m/s x100
ACTUAL_ACCERELATION,    // Actual value of the acceleration of the kart m/s^2 x100
ACTUAL_BRAKING,         // Actual value of the brake [0-1] interpreted as brake pedal or hydraulic
pressure) x100
ACTUAL_STEERING,        // Actual value of the steering in rad x100
MISION_ESP,             // Mision that is executing the ESP
MACHINE_STATE_ESP,      // State inside the state machine of the ESP
ACTUAL_SHUTDOWN,         // Actual state of the SHUTDOWN


KM_OBJ_LAST             // Este debe de ser siempre el ultimo
} km_objects_t;

/******************************* VARIABLES PÚBLICAS ***************************/
// Variables globales visibles (si realmente se necesitan)

// extern int ejemplo_variable_publica;

/******************************* FUNCIONES PÚBLICAS ***************************/
uint8_t KM_OBJ_SetObjectValue(km_objects_t object, int64_t value);


int64_t KM_OBJ_GetObjectValue(km_objects_t object);

#endif /* KM_OBJECTS_H */

```

4.17 km_pid.c File Reference

```
#include "km_pid.h"
```

Include dependency graph for km_pid.c:



Functions

- PID_Controller KM_PID_Init (float kp_val, float ki_val, float kd_val)
- float KM_PID_Calculate (PID_Controller ∗controller, float setpoint, float measurement)
- void KM_PID_SetTunings (PID_Controller ∗controller, float kp, float ki, float kd)
- void KM_PID_SetOutputLimits (PID_Controller ∗controller, float min, float max)
- void KM_PID_SetIntegralLimits (PID_Controller ∗controller, float min, float max)
- void KM_PID_Reset (PID_Controller ∗controller)
- void KM_PID_GetTunings (PID_Controller ∗controller, float kp, float ki, float kd)
- float KM_PID_GetIntegral (PID_Controller ∗controller)

4.17.1 Function Documentation

4.17.1.1 KM_PID_Calculate()

```
float KM_PID_Calculate (
            PID_Controller ∗ controller,
            float setpoint,
            float measurement )
```

Definition at line 37 of file km_pid.c.

```
00037                                                                                  {
00038     unsigned long currentTime = esp_timer_get_time();
00039     float dt = (currentTime - controller->lastTime) / 1000000.0f;  // Convert to seconds
00040 
00041     // Avoid division by zero
00042     if (dt <= 0.0f) {
00043         dt = 0.001f;  // Minimum 1ms
00044     }
00045 
00046     // Calculate error
00047     float error = setpoint - measurement;
00048 
00049     // Proportional term
00050     float pTerm = controller->kp * error;
00051 
00052     // Integral term with anti-windup
00053     controller->integral += error * dt;
00054 
00055     if (controller->integral < controller->integralMin) {
00056         controller->integral = controller->integralMin;
00057     }
00058 
00059     if (controller->integral > controller->integralMax) {
00060         controller->integral = controller->integralMax;
00061     }
00062 
00063     float iTerm = controller->ki * controller->integral;
00064 
00065     // Derivative term
00066     float derivative = (error - controller->lastError) / dt;
00067     float dTerm = controller->kd * derivative;
00068 
00069     // Calculate total output
00070     float output = pTerm + iTerm + dTerm;
00071 
00072     // Apply output limits
00073     if (output < controller->outputMin) output = controller->outputMin;
00074     if (output > controller->outputMax) output = controller->outputMax;
00075 
00076 
00077     // Update state
00078     controller->lastError = error;
00079     controller->lastTime = currentTime;
00080 
00081     return output;
00082 }
```

4.17.1.2 KM_PID_GetIntegral()

```
float KM_PID_GetIntegral (
              PID_Controller * controller )
```

Definition at line 113 of file km_pid.c.

```
00113                                                    {
00114     return controller->integral;
00115 }
```

4.17.1.3 KM_PID_GetTunings()

```
void KM_PID_GetTunings (
              PID_Controller * controller,
              float kp,
              float ki,
              float kd )
```

Definition at line 106 of file km_pid.c.

```
00106                                                                                              {
00107     controller->kp = kp;
00108     controller->ki = ki;
00109     controller->kd = kd;
00110 }
```

4.17.1.4 KM_PID_Init()

```
PID_Controller KM_PID_Init (
              float kp_val,
              float ki_val,
              float kd_val )
```

Definition at line 22 of file km_pid.c.

```
00022                                                                            {
00023
00024     PID_Controller controller;
00025
00026     controller.kp = kp_val;
00027     controller.ki = ki_val;
00028     controller.kd = kd_val;
00029
00030     controller.integral = 0.0f;
00031     controller.lastError = 0.0f;
00032     controller.lastTime = esp_timer_get_time();
00033
00034     return controller;
00035 }
```

4.17.1.5 KM_PID_Reset()

```
void KM_PID_Reset (
            PID_Controller * controller )
```

Definition at line 100 of file km_pid.c.

```
00100                                                  {
00101     controller->integral = 0.0f;
00102     controller->lastError = 0.0f;
00103     controller->lastTime = esp_timer_get_time();
00104 }
```

4.17.1.6 KM_PID_SetIntegralLimits()

```
void KM_PID_SetIntegralLimits (
            PID_Controller * controller,
            float min,
            float max )
```

Definition at line 95 of file km_pid.c.

```
00095                                                                                      {
00096     controller->integralMin = min;
00097     controller->integralMax = max;
00098 }
```

4.17.1.7 KM_PID_SetOutputLimits()

```
void KM_PID_SetOutputLimits (
            PID_Controller * controller,
            float min,
            float max )
```

Definition at line 90 of file km_pid.c.

```
00090                                                                                    {
00091     controller->outputMin = min;
00092     controller->outputMax = max;
00093 }
```

4.17.1.8 KM_PID_SetTunings()

```
void KM_PID_SetTunings (
            PID_Controller * controller,
            float kp,
            float ki,
            float kd )
```

Definition at line 84 of file km_pid.c.

```
00084                                                                                       {
00085     controller->kp = kp;
00086     controller->ki = ki;
00087     controller->kd = kd;
00088 }
```

4.18 km_pid.c

Go to the documentation of this file.

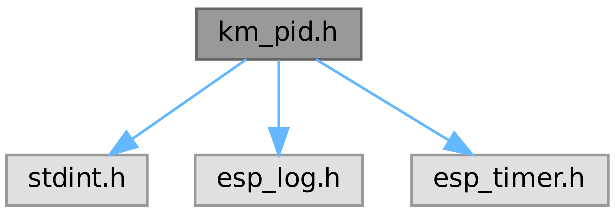
```
/******************************************************************************
 * @file    KM_PID.c
 * @brief   Implementación de la librería.
 * @author Adrian Navarredonda Arizaleta
 ******************************************************************************/

#include "km_pid.h"

/******************************* INCLUDES INTERNOS ****************************/
// Headers internos opcionales, dependencias privadas

/******************************* MACROS PRIVADAS ******************************/
// Constantes internas, flags de debug
// #define LIBRERIA_DEBUG 1

/******************************* VARIABLES PRIVADAS ***************************/
// Variables globales internas (static)

/******************************* DECLARACION FUNCIONES PRIVADAS ***************/

/******************************* FUNCIONES PÚBLICAS ***************************/
PID_Controller KM_PID_Init(float kp_val, float ki_val, float kd_val) {

    PID_Controller controller;

    controller.kp = kp_val;
    controller.ki = ki_val;
    controller.kd = kd_val;

    controller.integral = 0.0f;
    controller.lastError = 0.0f;
    controller.lastTime = esp_timer_get_time();

    return controller;
}

float KM_PID_Calculate(PID_Controller *controller, float setpoint, float measurement) {
    unsigned long currentTime = esp_timer_get_time();
    float dt = (currentTime - controller->lastTime) / 1000000.0f;  // Convert to seconds

    // Avoid division by zero
    if (dt <= 0.0f) {
        dt = 0.001f;  // Minimum 1ms
    }

    // Calculate error
    float error = setpoint - measurement;

    // Proportional term
    float pTerm = controller->kp * error;

    // Integral term with anti-windup
    controller->integral += error * dt;

    if (controller->integral < controller->integralMin) {
        controller->integral = controller->integralMin;
    }

    if (controller->integral > controller->integralMax) {
        controller->integral = controller->integralMax;
    }

    float iTerm = controller->ki * controller->integral;

    // Derivative term
    float derivative = (error - controller->lastError) / dt;
    float dTerm = controller->kd * derivative;

    // Calculate total output
    float output = pTerm + iTerm + dTerm;

    // Apply output limits
    if (output < controller->outputMin) output = controller->outputMin;
    if (output > controller->outputMax) output = controller->outputMax;


    // Update state
    controller->lastError = error;
    controller->lastTime = currentTime;

    return output;
}
```

```

void KM_PID_SetTunings(PID_Controller *controller, float kp, float ki, float kd) {
    controller->kp = kp;
    controller->ki = ki;
    controller->kd = kd;
}

void KM_PID_SetOutputLimits(PID_Controller *controller, float min, float max) {
    controller->outputMin = min;
    controller->outputMax = max;
}

void KM_PID_SetIntegralLimits(PID_Controller *controller, float min, float max) {
    controller->integralMin = min;
    controller->integralMax = max;
}

void KM_PID_Reset(PID_Controller *controller){
    controller->integral = 0.0f;
    controller->lastError = 0.0f;
    controller->lastTime = esp_timer_get_time();
}

void KM_PID_GetTunings(PID_Controller *controller, float kp, float ki, float kd) {
    controller->kp = kp;
    controller->ki = ki;
    controller->kd = kd;
}

// Get integral value (for debugging)
float KM_PID_GetIntegral(PID_Controller *controller){
    return controller->integral;
}

 /******************************* FUNCIONES PRIVADAS ***************************/

 /******************************* FIN DE ARCHIVO ********************************/
```

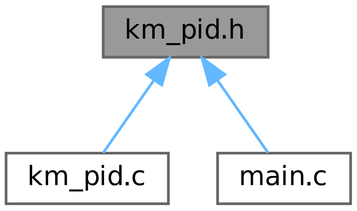
4.19 km_pid.h File Reference

```
#include <stdint.h>
#include "esp_log.h"
#include "esp_timer.h"
```

Include dependency graph for km_pid.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct PID_Controller

  Structure that reperesents a PID.

Functions

- PID_Controller KM_PID_Init (float kp, float ki, float kd)
- float KM_PID_Calculate (PID_Controller ∗controller, float setpoint, float measurement)
- void KM_PID_SetTunings (PID_Controller ∗controller, float kp, float ki, float kd)
- void KM_PID_SetOutputLimits (PID_Controller ∗controller, float min, float max)
- void KM_PID_SetIntegralLimits (PID_Controller ∗controller, float min, float max)
- void KM_PID_Reset (PID_Controller ∗controller)
- void KM_PID_GetTunings (PID_Controller ∗controller, float kp, float ki, float kd)
- float KM_PID_GetIntegral (PID_Controller ∗controller)

4.19.1 Function Documentation

4.19.1.1 KM_PID_Calculate()

```
float KM_PID_Calculate (
            PID_Controller * controller,
            float setpoint,
            float measurement )
```

Definition at line 37 of file km_pid.c.

```
                                                                                       {
    unsigned long currentTime = esp_timer_get_time();
    float dt = (currentTime - controller->lastTime) / 1000000.0f;  // Convert to seconds

    // Avoid division by zero
    if (dt <= 0.0f) {
        dt = 0.001f;  // Minimum 1ms
    }

    // Calculate error
    float error = setpoint - measurement;

    // Proportional term
    float pTerm = controller->kp * error;

```

```
00052     // Integral term with anti-windup
00053     controller->integral += error * dt;
00054 
00055     if (controller->integral < controller->integralMin) {
00056         controller->integral = controller->integralMin;
00057     }
00058 
00059     if (controller->integral > controller->integralMax) {
00060         controller->integral = controller->integralMax;
00061     }
00062 
00063     float iTerm = controller->ki * controller->integral;
00064 
00065     // Derivative term
00066     float derivative = (error - controller->lastError) / dt;
00067     float dTerm = controller->kd * derivative;
00068 
00069     // Calculate total output
00070     float output = pTerm + iTerm + dTerm;
00071 
00072     // Apply output limits
00073     if (output < controller->outputMin) output = controller->outputMin;
00074     if (output > controller->outputMax) output = controller->outputMax;
00075 
00076 
00077     // Update state
00078     controller->lastError = error;
00079     controller->lastTime = currentTime;
00080 
00081     return output;
00082 }
```

4.19.1.2 KM_PID_GetIntegral()

```
float KM_PID_GetIntegral (
              PID_Controller * controller )
```

Definition at line 113 of file km_pid.c.

```
00113                                                        {
00114     return controller->integral;
00115 }
```

4.19.1.3 KM_PID_GetTunings()

```
void KM_PID_GetTunings (
              PID_Controller * controller,
              float kp,
              float ki,
              float kd )
```

Definition at line 106 of file km_pid.c.

```
00106                                                                                  {
00107     controller->kp = kp;
00108     controller->ki = ki;
00109     controller->kd = kd;
00110 }
```

4.19.1.4 KM_PID_Init()

```
PID_Controller KM_PID_Init (
              float kp,
              float ki,
              float kd )
```

Definition at line 22 of file km_pid.c.

```
00022                                                                        {
```

```
00023
00024     PID_Controller controller;
00025
00026     controller.kp = kp_val;
00027     controller.ki = ki_val;
00028     controller.kd = kd_val;
00029
00030     controller.integral = 0.0f;
00031     controller.lastError = 0.0f;
00032     controller.lastTime = esp_timer_get_time();
00033
00034     return controller;
00035 }
```

4.19.1.5 KM_PID_Reset()

```
void KM_PID_Reset (
            PID_Controller * controller )
```

Definition at line 100 of file km_pid.c.

```
00100                                          {
00101     controller->integral = 0.0f;
00102     controller->lastError = 0.0f;
00103     controller->lastTime = esp_timer_get_time();
00104 }
```

4.19.1.6 KM_PID_SetIntegralLimits()

```
void KM_PID_SetIntegralLimits (
            PID_Controller * controller,
            float min,
            float max )
```

Definition at line 95 of file km_pid.c.

```
00095                                                                               {
00096     controller->integralMin = min;
00097     controller->integralMax = max;
00098 }
```

4.19.1.7 KM_PID_SetOutputLimits()

```
void KM_PID_SetOutputLimits (
            PID_Controller * controller,
            float min,
            float max )
```

Definition at line 90 of file km_pid.c.

```
00090                                                                             {
00091     controller->outputMin = min;
00092     controller->outputMax = max;
00093 }
```

4.19.1.8 KM_PID_SetTunings()

```
void KM_PID_SetTunings (
            PID_Controller * controller,
            float kp,
            float ki,
            float kd )
```

Definition at line 84 of file km_pid.c.

```
00084                                                                                {
00085     controller->kp = kp;
00086     controller->ki = ki;
00087     controller->kd = kd;
00088 }
```

4.20 km_pid.h

Go to the documentation of this file.

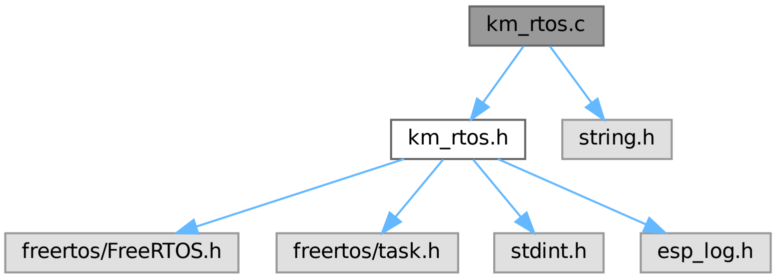
```
/*******************************************************************************
 * @file    km_pid.h
 * @brief   Interfaz pública de la librería.
 * @author  Adrian Navarredonda Arizaleta
 * @date    25-01-2026
 * @version 1.0
 ******************************************************************************/

#ifndef KM_PID_H
#define KM_PID_H

/******************************* INCLUDES ***********************************/
// Includes necesarios para la API pública
#include <stdint.h>
#include "esp_log.h" // Para log
#include "esp_timer.h"

/******************************* DEFINES PÚBLICAS ***************************/
// Constantes, flags o configuraciones visibles desde fuera de la librería

typedef struct {
    float kp;
    float ki;
    float kd;
    float integral;
    float lastError;
    uint64_t lastTime;
    // Limits
    float outputMin;
    float outputMax;
    float integralMin;
    float integralMax;
} PID_Controller;

/******************************* TIPOS PÚBLICOS *****************************/
// Estructuras, enums, typedefs públicos

/******************************* VARIABLES PÚBLICAS *************************/
// Variables globales visibles (si realmente se necesitan)

/******************************* FUNCIONES PÚBLICAS *************************/

// Initialize with gains
PID_Controller KM_PID_Init(float kp, float ki, float kd);

// Calculate PID output
float KM_PID_Calculate(PID_Controller *controller, float setpoint, float measurement);

// Set tuning parameters at runtime
void KM_PID_SetTunings(PID_Controller *controller, float kp, float ki, float kd);

// Set output limits
void KM_PID_SetOutputLimits(PID_Controller *controller, float min, float max);

// Set integral limits (anti-windup)
void KM_PID_SetIntegralLimits(PID_Controller *controller, float min, float max);

// Reset controller state
void KM_PID_Reset(PID_Controller *controller);

// Get current gains
void KM_PID_GetTunings(PID_Controller *controller, float kp, float ki, float kd);

// Get integral value (for debugging)
float KM_PID_GetIntegral(PID_Controller *controller);

#endif // KM_PID_H
```

4.21 km_rtos.c File Reference

Implementation of the KM_RTOS task management library.

```
#include "km_rtos.h"
#include <string.h>
```

Include dependency graph for km_rtos.c:



Macros

- #define RTOS_DEFAULT_STACK_SIZE 1024
- #define RTOS_DEFAULT_PRIORITY 1
- #define RTOS_DEFAULT_PERIOD_MS 50

Functions

- int8_t KM_RTOS_FindTask (TaskHandle_t handle)

  Finds the index of a task in the internal task array by its FreeRTOS handle.
- static void KM_RTOS_TaskWrapper (void ∗params)

  FreeRTOS wrapper function for executing a periodic task.
- void KM_RTOS_Init (void)

  Initializes the RTOS task system.
- void KM_RTOS_Destroy (void)

  Destroys all tasks managed by the RTOS task system and resets the task array.
- RTOS_Task KM_COMS_CreateTask (char ∗name, KM_RTOS_TaskFunction_t taskFn, void ∗context, uint32_t period_ms, uint16_t stackSize, UBaseType_t priority, uint8_t active)

  Creates and initializes an RTOS_Task structure.
- esp_err_t KM_RTOS_AddTask (RTOS_Task task)

  Creates a new FreeRTOS task and stores it in the internal task array.
- esp_err_t KM_RTOS_DeleteTask (TaskHandle_t handle)

  Deletes a specific FreeRTOS task and clears its slot in the internal task array.
- esp_err_t KM_RTOS_SuspendTask (TaskHandle_t handle)

  Suspends a specific FreeRTOS task and marks it as inactive in the internal task array.
- esp_err_t KM_RTOS_ResumeTask (TaskHandle_t handle)

  Resumes a specific FreeRTOS task and marks it as active in the internal task array.
- esp_err_t KM_RTOS_RestartTask (TaskHandle_t handle)

  Restarts a specific FreeRTOS task by deleting and recreating it.
- esp_err_t KM_RTOS_ChangePriority (TaskHandle_t handle, uint8_t newPriority)

  Changes the priority of a specific FreeRTOS task and updates the internal task array.

Variables

- static RTOS_Task tasks [RTOS_MAX_TASKS]

4.21.1 Detailed Description

Implementation of the KM_RTOS task management library.

This source file contains the implementation of the KM_RTOS library, which provides a lightweight abstraction for managing FreeRTOS tasks in a structured and safe manner.

The library handles:

- Initialization and cleanup of the task management subsystem.
- Creation, deletion, suspension, resumption, and restart of tasks.
- Periodic task execution using a task wrapper to enforce timing.
- Internal management of the static task array, ensuring consistency and safe access.

Both public API functions and private helper functions are implemented in this file.

Usage:

- Call KM_RTOS_Init() before creating tasks.
- Use the provided KM_RTOS_* functions to manipulate tasks safely.

Author: Adrian Navarredonda Arizaleta Date: 24-01-2026 Version: 1.0

Definition in file km_rtos.c.

4.21.2 Macro Definition Documentation

4.21.2.1 RTOS_DEFAULT_PERIOD_MS

```
#define RTOS_DEFAULT_PERIOD_MS 50
```

Definition at line 35 of file km_rtos.c.

4.21.2.2 RTOS_DEFAULT_PRIORITY

```
#define RTOS_DEFAULT_PRIORITY 1
```

Definition at line 34 of file km_rtos.c.

4.21.2.3 RTOS_DEFAULT_STACK_SIZE

```
#define RTOS_DEFAULT_STACK_SIZE 1024
```

Definition at line 33 of file km_rtos.c.

4.21.3 Function Documentation

4.21.3.1 KM_COMS_CreateTask()

```
RTOS_Task KM_COMS_CreateTask (
            char * name,
            KM_RTOS_TaskFunction_t taskFn,
            void * context,
            uint32_t period_ms,
            uint16_t stackSize,
            UBaseType_t priority,
            uint8_t active )
```

Creates and initializes an RTOS_Task structure.

This function fills a `RTOS_Task` structure with the provided parameters, preparing it to be added to the task scheduler or created with `KM_RTOS_CreateTask`.

Parameters

name	Name of the task (used by FreeRTOS for debugging/logging)
taskFn	Function pointer to the task implementation (of type KM_RTOS_TaskFunction_t)
context	Pointer to a user-defined context or data structure passed to the task
period_ms	Task period in milliseconds (used for periodic scheduling)
stackSize	Stack size for the FreeRTOS task
priority	FreeRTOS task priority
active	1 to mark the task as active, 0 to mark as inactive

Returns

A fully-initialized `RTOS_Task` structure ready to be added to the scheduler.

Note

This function does not create the FreeRTOS task itself; it only prepares the task structure for use with `KM_RTOS_CreateTask`.

Definition at line 62 of file km_rtos.c.

```
                                                                        {

    RTOS_Task task;

    task.name = name;
    task.taskFn = taskFn;
    task.context = context;
    task.period_ms = period_ms;
    task.stackSize = stackSize;
    task.priority = priority;
    task.active = active;

    return task;
}
```

4.21.3.2 KM_RTOS_AddTask()

```
esp_err_t KM_RTOS_AddTask (
            RTOS_Task task )
```

Creates a new FreeRTOS task and stores it in the internal task array.

This function searches for a free slot in the internal `tasks` array and attempts to create a new FreeRTOS task using the provided `RTOS_Task` configuration. It ensures that the task does not already exist and fills in default values for priority, stack size, and period if they are not specified.

Steps performed by the function:

1. Iterate through the `tasks` array to find an empty slot (`handle == NULL`).
2. Assign the provided `task` to the slot and mark it as active.
3. Fill default parameters if `priority`, `stackSize`, or `period_ms` are zero.
4. Call `xTaskCreate` with `KM_RTOS_TaskWrapper` as the task function, passing the task structure pointer as the argument.
5. On success, store the FreeRTOS task handle in the task structure.
6. On failure, clear the slot and return 0.

Parameters

task	The RTOS_Task structure containing task configuration (name, priority, stack size, period, etc.)

Returns

1 if the task was successfully created

0 if task creation failed or there was no free slot in the array

Note

This function manages tasks in a static array of size `RTOS_MAX_TASKS`.

Task execution is wrapped by `KM_RTOS_TaskWrapper`.

Default values are applied for any missing parameters:

- priority: `RTOS_DEFAULT_PRIORITY`
- stack size: `RTOS_DEFAULT_STACK_SIZE`
- period_ms: `RTOS_DEFAULT_PERIOD_MS`

Definition at line 78 of file km_rtos.c.

```
                                               {

    // Buscar hueco libre en array y comprobar que no exista ya esa tarea
    for (uint8_t i = 0; i < RTOS_MAX_TASKS; i++) {
        // Hueco ocupado
        if (tasks[i].handle != NULL) continue;

        tasks[i] = task;
        tasks[i].active = 1;

        if (tasks[i].priority == 0) tasks[i].priority = RTOS_DEFAULT_PRIORITY;
        if (tasks[i].stackSize == 0) tasks[i].stackSize = RTOS_DEFAULT_STACK_SIZE;
        if (tasks[i].period_ms == 0) tasks[i].period_ms = RTOS_DEFAULT_PERIOD_MS;

        BaseType_t result = xTaskCreate(
            KM_RTOS_TaskWrapper,  // Wrapper
            task.name,
            task.stackSize,
            &tasks[i],            // Pointer to this task
            task.priority,
            &tasks[i].handle
        );

        if (result != pdPASS) {
            memset(&tasks[i], 0, sizeof(RTOS_Task));
            return ESP_FAIL; // Error creating task
        }

        return ESP_OK; // Success
    }

    return ESP_FAIL; // Fail, no space in array
}
```

4.21.3.3 KM_RTOS_ChangePriority()

```
esp_err_t KM_RTOS_ChangePriority (
               TaskHandle_t handle,
               uint8_t newPriority )
```

Changes the priority of a specific FreeRTOS task and updates the internal task array.

This function searches the internal `tasks` array for the task corresponding to the provided handle. If found, it calls `vTaskPrioritySet` to update the task's FreeRTOS priority and updates the `priority` field in the internal `tasks` array.

Parameters

handle	The FreeRTOS task handle whose priority should be changed.
newPriority	The new priority value to assign to the task.

Returns

ESP_OK if the priority was successfully updated.

ESP_FAIL if the task handle was not found in the internal array.

Note

Only affects the specified task; other tasks remain unchanged.

Safe under normal operation, as it modifies only the targeted task entry.

The change is immediate from the scheduler's perspective.

Definition at line 181 of file km_rtos.c.

```
                                                                        {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    vTaskPrioritySet(tasks[index].handle, newPriority);

    // Actualizar informacion sobre tarea
    tasks[index].priority = newPriority;

    return ESP_OK;
}
```

4.21.3.4 KM_RTOS_DeleteTask()

```
esp_err_t KM_RTOS_DeleteTask (
            TaskHandle_t handle )
```

Deletes a specific FreeRTOS task and clears its slot in the internal task array.

This function searches the internal `tasks` array for a task that matches the provided FreeRTOS task handle. If found, the task is deleted using `vTaskDelete` and the corresponding slot in the array is cleared with `memset`, marking it as free.

Parameters

handle	The FreeRTOS task handle to delete.

Returns

ESP_OK if the task was successfully deleted.

ESP_FAIL if the task handle was not found in the internal array.

Note

After this function, the task slot is fully reset and can be reused for new tasks.

Thread-safd calling this while other tasks may concurrently access the `tasks` array.

Definition at line 113 of file km_rtos.c.

```
                                             {
    // Search for task in array

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    if (tasks[index].handle == handle) {
        vTaskDelete(handle);
        memset(&tasks[index], 0, sizeof(RTOS_Task));
        return ESP_OK;
    }

    // NO se ha encontrado la tarea
    return ESP_FAIL;
}
```

4.21.3.5 KM_RTOS_Destroy()

```
void KM_RTOS_Destroy (
            void )
```

Destroys all tasks managed by the RTOS task system and resets the task array.

This function iterates through the internal `tasks` array and deletes all active FreeRTOS tasks using `vTask←Delete`. After deleting tasks, it clears the `tasks` array to mark all slots as free and inactive.

The function wraps the entire operation in a critical section (`taskENTER_CRITICAL` / `taskEXIT_CRITICAL`) to prevent the scheduler from switching tasks while the array is being modified. This ensures thread-safety during the destruction process.

Note

All tasks created through `KM_RTOS_CreateTask` will be stopped.

After calling this function, the internal task array is fully reset and ready for re-initialization.

Definition at line 52 of file km_rtos.c.

```
                                 {
    for (int i = 0; i < RTOS_MAX_TASKS; i++) {
        if (tasks[i].name != NULL) {
            vTaskDelete(tasks[i].handle);
        }
    }

    memset(tasks, 0, sizeof(tasks));
}
```

4.21.3.6 KM_RTOS_FindTask()

```
int8_t KM_RTOS_FindTask (
             TaskHandle_t handle )
```

Finds the index of a task in the internal task array by its FreeRTOS handle.

This function searches the internal `tasks` array for a task that matches the given FreeRTOS `TaskHandle_t`. It is used to identify tasks for monitoring, activation/deactivation, or debugging purposes.

Parameters

handle	The FreeRTOS task handle to search for.

Returns

The index (0 to RTOS_MAX_TASKS-1) of the task in the internal array if found.

-1 if the task handle was not found.

Note

The function iterates over a static array of size `RTOS_MAX_TASKS`.

Does not modify the task array or the tasks themselves.

Definition at line 210 of file km_rtos.c.

```
                                                 {

    for (int8_t i = 0; i < RTOS_MAX_TASKS; i++) {
        if (tasks[i].handle == handle) {
            return i;
        }
    }

    return -1;
}
```

4.21.3.7 KM_RTOS_Init()

```
void KM_RTOS_Init (
             void  )
```

Initializes the RTOS task system.

This function resets the internal `tasks` array by zeroing all entries, effectively marking all task slots as free and inactive. It should be called once at system startup before creating or managing any tasks.

Note

This function does not create any FreeRTOS tasks; it only prepares the internal task array for use.

Definition at line 47 of file km_rtos.c.

```
00047                      {
00048     memset(tasks, 0, sizeof(tasks));
00049 }
```

4.21.3.8 KM_RTOS_RestartTask()

```
esp_err_t KM_RTOS_RestartTask (
            TaskHandle_t handle )
```

Restarts a specific FreeRTOS task by deleting and recreating it.

This function searches the internal `tasks` array for the task matching the provided handle. If found, it performs the following steps:

1. Makes a copy of the task structure to preserve its configuration.
2. Deletes the task using `KM_RTOS_DeleteTask`.
3. Re-adds the task using `KM_RTOS_AddTask` with the preserved configuration.

Parameters

handle	The FreeRTOS task handle to restart.

Returns

ESP_OK if the task was successfully restarted.

ESP_FAIL if the task handle was not found, deletion failed, or creation failed.

Note

This operation temporarily stops the task and replaces it with a new instance.

Definition at line 163 of file km_rtos.c.

```
00163                                         {
00164
00165     int8_t index = KM_RTOS_FindTask(handle);
00166     if (index == -1) return ESP_FAIL;
00167
00168     RTOS_Task copy = tasks[index];
00169
00170     // Destruir tarea
00171     if(KM_RTOS_DeleteTask(handle) == ESP_FAIL) return ESP_FAIL;
00172
00173     // Crear tarea
00174     if(KM_RTOS_AddTask(copy) == ESP_FAIL) return ESP_FAIL;
00175
00176     // Reiniciado correctamente
00177     return ESP_OK;
00178 }
```

4.21.3.9 KM_RTOS_ResumeTask()

```
esp_err_t KM_RTOS_ResumeTask (
            TaskHandle_t handle )
```

Resumes a specific FreeRTOS task and marks it as active in the internal task array.

This function searches the internal `tasks` array for the task corresponding to the provided FreeRTOS handle. If found and the task is currently suspended, it calls `vTaskResume` to restart the task's execution and sets its `active` flag to 1.

Parameters

handle	The FreeRTOS task handle to resume.

Returns

ESP_OK if the task was successfully resumed.

ESP_FAIL if the task handle was not found or the task was already active.

Note

Only affects the specified task; other tasks and array slots remain untouched.

Safe to call under normal operation, as it modifies only the targeted task entry.

Definition at line 146 of file km_rtos.c.

```
                                                {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return 0;

    if (tasks[index].handle == handle && !tasks[index].active) {
        vTaskResume(handle);
        tasks[index].active = 1;

        return ESP_OK;
    }

    // No se ha encontrado la tarea o ya estaba activa
    return ESP_FAIL;
}
```

4.21.3.10 KM_RTOS_SuspendTask()

```
esp_err_t KM_RTOS_SuspendTask (
            TaskHandle_t handle )
```

Suspends a specific FreeRTOS task and marks it as inactive in the internal task array.

This function searches the internal `tasks` array for the task corresponding to the provided FreeRTOS handle. If found and the task is currently active, it calls `vTaskSuspend` to pause the task's execution and sets its `active` flag to 0.

Parameters

handle	The FreeRTOS task handle to suspend.

Returns

ESP_OK if the task was successfully suspended.

ESP_FAIL if the task handle was not found or the task was already suspended.

Note

Only affects the specified task; other tasks and array slots remain untouched.

Safe to call under normal operation, as it modifies only the targeted task entry.

Definition at line 130 of file km_rtos.c.

```
                                                  {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    if (tasks[index].handle == handle && tasks[index].active) {
        vTaskSuspend(handle);
        tasks[index].active = 0;
        return ESP_OK;
    }

    // No se ha encontrado la tarea o ya estaba suspendida
    return ESP_FAIL;
}
```

4.21.3.11 KM_RTOS_TaskWrapper()

```
static void KM_RTOS_TaskWrapper (
              void * params )  [static]
```

FreeRTOS wrapper function for executing a periodic task.

This function serves as a generic wrapper for all RTOS tasks created through the `KM_RTOS_CreateTask` system. It manages periodic execution and ensures that the user-defined task function is called at the configured interval.

Steps performed by the function:

1. Casts the `params` pointer to an `RTOS_Task` structure.
2. Retrieves the current tick count to use as a reference for periodic delays.
3. Continuously loops:
   - Checks if the task is active and has a valid function pointer.
   - Calls the task's logical function, passing the user-defined context.
   - Delays until the next scheduled execution using `vTaskDelayUntil` to maintain a fixed period.

Parameters

params	Pointer to the `RTOS_Task` structure representing this task.

Note

This function is intended to be passed to `xTaskCreate` as the FreeRTOS task function.

Handles the periodic execution of tasks based on `task->period_ms`.

If `task` is NULL, the wrapper deletes itself immediately.

Task execution is controlled by the `active` flag in the `RTOS_Task` structure.

Definition at line 243 of file km_rtos.c.

```
00243                                                {
00244     RTOS_Task *task = (RTOS_Task *)params;
00245     TickType_t xLastWakeTime = xTaskGetTickCount();
00246
00247     if (!task || task->taskFn) vTaskDelete(NULL);
00248
00249     while (1) {
00250         if (task->active && task->taskFn != NULL) {
00251             task->taskFn(task->context);  // Call the logical function
00252         }
00253
00254         vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(task->period_ms));
00255     }
00256 }
```

4.21.4 Variable Documentation

4.21.4.1 tasks

`RTOS_Task tasks[RTOS_MAX_TASKS]  [static]`

Definition at line 39 of file km_rtos.c.

4.22 km_rtos.c

Go to the documentation of this file.

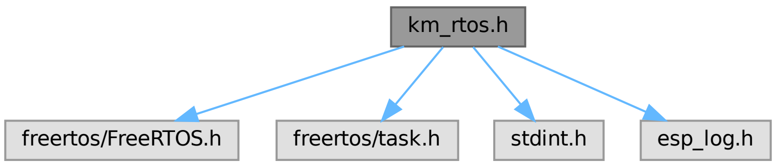
```
00001
00025 /******************************** INCLUDES INTERNOS ****************************/
00026 // Headers internos opcionales, dependencias privadas
00027 #include "km_rtos.h"
00028
00029 #include <string.h>
00030
00031 /******************************** DEFINES PRIVADAS *****************************/
00032 // Constantes internas
00033 #define RTOS_DEFAULT_STACK_SIZE 1024// Default size stack
00034 #define RTOS_DEFAULT_PRIORITY   1   // Default priority for each task
00035 #define RTOS_DEFAULT_PERIOD_MS 50   // Default time for executing each task
00036
00037 /******************************** VARIABLES PRIVADAS ***************************/
00038 // Variables globales internas (static)
00039 static RTOS_Task tasks[RTOS_MAX_TASKS];
00040
00041 /******************************** DECLARACION FUNCIONES PRIVADAS ***************/
00042 int8_t KM_RTOS_FindTask(TaskHandle_t handle);
00043 static void KM_RTOS_TaskWrapper(void *params);
00044
00045 /******************************** FUNCIONES PÚBLICAS ***************************/
00046
00047 void KM_RTOS_Init(void){
00048     memset(tasks, 0, sizeof(tasks));
00049 }
00050
00051 // Destruye todas las tareas
00052 void KM_RTOS_Destroy(void){
00053     for (int i = 0; i < RTOS_MAX_TASKS; i++) {
00054         if (tasks[i].name != NULL) {
00055             vTaskDelete(tasks[i].handle);
00056         }
00057     }
00058
00059     memset(tasks, 0, sizeof(tasks));
00060 }
00061
00062 RTOS_Task KM_COMS_CreateTask(char *name, KM_RTOS_TaskFunction_t taskFn, void *context,
00063         uint32_t period_ms, uint16_t stackSize, UBaseType_t priority, uint8_t active){

    RTOS_Task task;

    task.name = name;
    task.taskFn = taskFn;
    task.context = context;
    task.period_ms = period_ms;
    task.stackSize = stackSize;
    task.priority = priority;
    task.active = active;

    return task;
}

esp_err_t KM_RTOS_AddTask(RTOS_Task task){

    // Buscar hueco libre en array y comprobar que no exista ya esa tarea
    for (uint8_t i = 0; i < RTOS_MAX_TASKS; i++) {
        // Hueco ocupado
        if (tasks[i].handle != NULL) continue;

        tasks[i] = task;
        tasks[i].active = 1;

        if (tasks[i].priority == 0) tasks[i].priority = RTOS_DEFAULT_PRIORITY;
        if (tasks[i].stackSize == 0) tasks[i].stackSize = RTOS_DEFAULT_STACK_SIZE;
        if (tasks[i].period_ms == 0) tasks[i].period_ms = RTOS_DEFAULT_PERIOD_MS;

        BaseType_t result = xTaskCreate(
            KM_RTOS_TaskWrapper,  // Wrapper
            task.name,
            task.stackSize,
            &tasks[i],            // Pointer to this task
            task.priority,
            &tasks[i].handle
        );

        if (result != pdPASS) {
            memset(&tasks[i], 0, sizeof(RTOS_Task));
            return ESP_FAIL; // Error creating task
        }

        return ESP_OK; // Success
    }

    return ESP_FAIL; // Fail, no space in array
}

// Destruir una tarea existente, devuelve 0 en caso de error, 1 en caso correcto
esp_err_t KM_RTOS_DeleteTask(TaskHandle_t handle){
    // Search for task in array

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    if (tasks[index].handle == handle) {
        vTaskDelete(handle);
        memset(&tasks[index], 0, sizeof(RTOS_Task));
        return ESP_OK;
    }

    // NO se ha encontrado la tarea
    return ESP_FAIL;
}

// Suspender una tarea, devuelve 0 en caso de error, 1 en caso correcto
esp_err_t KM_RTOS_SuspendTask(TaskHandle_t handle) {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    if (tasks[index].handle == handle && tasks[index].active) {
        vTaskSuspend(handle);
        tasks[index].active = 0;
        return ESP_OK;
    }

    // No se ha encontrado la tarea o ya estaba suspendida
    return ESP_FAIL;
}

// Reanudar una tarea, devuelve 0 en caso de error, 1 en caso correcto
esp_err_t KM_RTOS_ResumeTask(TaskHandle_t handle){

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return 0;

    if (tasks[index].handle == handle && !tasks[index].active) {
        vTaskResume(handle);
        tasks[index].active = 1;

        return ESP_OK;
    }

    // No se ha encontrado la tarea o ya estaba activa
    return ESP_FAIL;
}

// Reiniciar una tarea (destruir + volver a crear), devuelve 0 en caso de error, 1 en caso correcto
esp_err_t KM_RTOS_RestartTask(TaskHandle_t handle){

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    RTOS_Task copy = tasks[index];

    // Destruir tarea
    if(KM_RTOS_DeleteTask(handle) == ESP_FAIL) return ESP_FAIL;

    // Crear tarea
    if(KM_RTOS_AddTask(copy) == ESP_FAIL) return ESP_FAIL;

    // Reiniciado correctamente
    return ESP_OK;
}

// Cambiar prioridad, devuelve 0 en caso de error, 1 en caso correcto
esp_err_t KM_RTOS_ChangePriority(TaskHandle_t handle, uint8_t newPriority){

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    vTaskPrioritySet(tasks[index].handle, newPriority);

    // Actualizar informacion sobre tarea
    tasks[index].priority = newPriority;

    return ESP_OK;
}

/******************************* FUNCIONES PRIVADAS **************************/

int8_t KM_RTOS_FindTask(TaskHandle_t handle) {

    for (int8_t i = 0; i < RTOS_MAX_TASKS; i++) {
        if (tasks[i].handle == handle) {
            return i;
        }
    }

    return -1;
}

static void KM_RTOS_TaskWrapper(void *params) {
    RTOS_Task *task = (RTOS_Task *)params;
    TickType_t xLastWakeTime = xTaskGetTickCount();

    if (!task || task->taskFn) vTaskDelete(NULL);

    while (1) {
        if (task->active && task->taskFn != NULL) {
            task->taskFn(task->context);  // Call the logical function
        }

        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(task->period_ms));
    }
}

/******************************* FIN DE ARCHIVO *******************************/

```

4.23 km_rtos.h File Reference

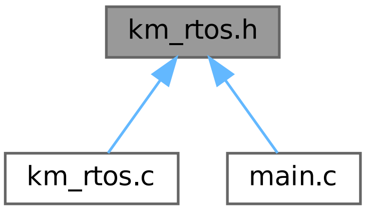
```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <stdint.h>
```

```
#include "esp_log.h"
```

Include dependency graph for km_rtos.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct RTOS_Task

  Structure that reperesents a task in FreeRTOS.

Macros

- #define RTOS_MAX_TASKS 10

Typedefs

- typedef void(∗ KM_RTOS_TaskFunction_t) (void ∗context)

  Logical task function type (RTOS-agnostic)

Functions

- void KM_RTOS_Init (void)

  Initializes the RTOS task system.
- void KM_RTOS_Destroy (void)

  Destroys all tasks managed by the RTOS task system and resets the task array.
- RTOS_Task KM_COMS_CreateTask (char ∗name, KM_RTOS_TaskFunction_t taskFn, void ∗context, uint32_t period_ms, uint16_t stackSize, UBaseType_t priority, uint8_t active)

  Creates and initializes an RTOS_Task structure.
- esp_err_t KM_RTOS_AddTask (RTOS_Task task)

  Creates a new FreeRTOS task and stores it in the internal task array.
- esp_err_t KM_RTOS_DeleteTask (TaskHandle_t handle)

  Deletes a specific FreeRTOS task and clears its slot in the internal task array.
- esp_err_t KM_RTOS_SuspendTask (TaskHandle_t handle)

  Suspends a specific FreeRTOS task and marks it as inactive in the internal task array.
- esp_err_t KM_RTOS_ResumeTask (TaskHandle_t handle)

  Resumes a specific FreeRTOS task and marks it as active in the internal task array.
- esp_err_t KM_RTOS_RestartTask (TaskHandle_t handle)

  Restarts a specific FreeRTOS task by deleting and recreating it.
- esp_err_t KM_RTOS_ChangePriority (TaskHandle_t handle, uint8_t newPriority)

  Changes the priority of a specific FreeRTOS task and updates the internal task array.

4.23.1 Macro Definition Documentation

4.23.1.1 RTOS_MAX_TASKS

```
#define RTOS_MAX_TASKS 10
```

Definition at line 39 of file km_rtos.h.

4.23.2 Typedef Documentation

4.23.2.1 KM_RTOS_TaskFunction_t

```
typedef void(* KM_RTOS_TaskFunction_t) (void *context)
```

Logical task function type (RTOS-agnostic)

This function is executed periodically by the RTOS wrapper. It must NOT contain an infinite loop or call vTaskDelay().

Parameters

context	User-defined context pointer

Definition at line 49 of file km_rtos.h.

4.23.3 Function Documentation

4.23.3.1 KM_COMS_CreateTask()

```
RTOS_Task KM_COMS_CreateTask (
            char * name,
            KM_RTOS_TaskFunction_t taskFn,
            void * context,
            uint32_t period_ms,
            uint16_t stackSize,
            UBaseType_t priority,
            uint8_t active )
```

Creates and initializes an RTOS_Task structure.

This function fills a `RTOS_Task` structure with the provided parameters, preparing it to be added to the task scheduler or created with `KM_RTOS_CreateTask`.

Parameters

name	Name of the task (used by FreeRTOS for debugging/logging)
taskFn	Function pointer to the task implementation (of type KM_RTOS_TaskFunction_t)
context	Pointer to a user-defined context or data structure passed to the task
period_ms	Task period in milliseconds (used for periodic scheduling)
stackSize	Stack size for the FreeRTOS task
priority	FreeRTOS task priority
active	1 to mark the task as active, 0 to mark as inactive

Returns

A fully-initialized `RTOS_Task` structure ready to be added to the scheduler.

Note

This function does not create the FreeRTOS task itself; it only prepares the task structure for use with `KM_RTOS_CreateTask`.

Definition at line 62 of file km_rtos.c.

```
                                                                             {

    RTOS_Task task;

    task.name = name;
    task.taskFn = taskFn;
    task.context = context;
    task.period_ms = period_ms;
    task.stackSize = stackSize;
    task.priority = priority;
    task.active = active;

    return task;
}
```

4.23.3.2 KM_RTOS_AddTask()

```
esp_err_t KM_RTOS_AddTask (
            RTOS_Task task )
```

Creates a new FreeRTOS task and stores it in the internal task array.

This function searches for a free slot in the internal `tasks` array and attempts to create a new FreeRTOS task using the provided `RTOS_Task` configuration. It ensures that the task does not already exist and fills in default values for priority, stack size, and period if they are not specified.

Steps performed by the function:

1. Iterate through the `tasks` array to find an empty slot (`handle == NULL`).
2. Assign the provided `task` to the slot and mark it as active.
3. Fill default parameters if `priority`, `stackSize`, or `period_ms` are zero.
4. Call `xTaskCreate` with `KM_RTOS_TaskWrapper` as the task function, passing the task structure pointer as the argument.
5. On success, store the FreeRTOS task handle in the task structure.
6. On failure, clear the slot and return 0.

Parameters

task	The RTOS_Task structure containing task configuration (name, priority, stack size, period, etc.)

Returns

1 if the task was successfully created

0 if task creation failed or there was no free slot in the array

Note

This function manages tasks in a static array of size `RTOS_MAX_TASKS`.

Task execution is wrapped by `KM_RTOS_TaskWrapper`.

Default values are applied for any missing parameters:

- priority: `RTOS_DEFAULT_PRIORITY`
- stack size: `RTOS_DEFAULT_STACK_SIZE`
- period_ms: `RTOS_DEFAULT_PERIOD_MS`

Definition at line 78 of file km_rtos.c.

```
                                       {

    // Buscar hueco libre en array y comprobar que no exista ya esa tarea
    for (uint8_t i = 0; i < RTOS_MAX_TASKS; i++) {
        // Hueco ocupado
        if (tasks[i].handle != NULL) continue;

        tasks[i] = task;
        tasks[i].active = 1;

        if (tasks[i].priority == 0) tasks[i].priority = RTOS_DEFAULT_PRIORITY;
        if (tasks[i].stackSize == 0) tasks[i].stackSize = RTOS_DEFAULT_STACK_SIZE;
        if (tasks[i].period_ms == 0) tasks[i].period_ms = RTOS_DEFAULT_PERIOD_MS;

        BaseType_t result = xTaskCreate(
            KM_RTOS_TaskWrapper,  // Wrapper
            task.name,
            task.stackSize,
            &tasks[i],            // Pointer to this task
            task.priority,
            &tasks[i].handle
        );

        if (result != pdPASS) {
            memset(&tasks[i], 0, sizeof(RTOS_Task));
            return ESP_FAIL; // Error creating task
        }

        return ESP_OK; // Success
    }

    return ESP_FAIL; // Fail, no space in array
}
```

4.23.3.3 KM_RTOS_ChangePriority()

```
esp_err_t KM_RTOS_ChangePriority (
            TaskHandle_t handle,
            uint8_t newPriority )
```

Changes the priority of a specific FreeRTOS task and updates the internal task array.

This function searches the internal `tasks` array for the task corresponding to the provided handle. If found, it calls `vTaskPrioritySet` to update the task's FreeRTOS priority and updates the `priority` field in the internal `tasks` array.

Parameters

handle	The FreeRTOS task handle whose priority should be changed.
newPriority	The new priority value to assign to the task.

Returns

ESP_OK if the priority was successfully updated.

ESP_FAIL if the task handle was not found in the internal array.

Note

Only affects the specified task; other tasks remain unchanged.

Safe under normal operation, as it modifies only the targeted task entry.

The change is immediate from the scheduler's perspective.

Definition at line 181 of file km_rtos.c.

```
                                                                {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    vTaskPrioritySet(tasks[index].handle, newPriority);

    // Actualizar informacion sobre tarea
    tasks[index].priority = newPriority;

    return ESP_OK;
}
```

4.23.3.4 KM_RTOS_DeleteTask()

```
esp_err_t KM_RTOS_DeleteTask (
              TaskHandle_t handle )
```

Deletes a specific FreeRTOS task and clears its slot in the internal task array.

This function searches the internal `tasks` array for a task that matches the provided FreeRTOS task handle. If found, the task is deleted using `vTaskDelete` and the corresponding slot in the array is cleared with `memset`, marking it as free.

Parameters

handle	The FreeRTOS task handle to delete.

Returns

ESP_OK if the task was successfully deleted.

ESP_FAIL if the task handle was not found in the internal array.

Note

After this function, the task slot is fully reset and can be reused for new tasks.

Thread-safd calling this while other tasks may concurrently access the `tasks` array.

Definition at line 113 of file km_rtos.c.

```
                                                   {
    // Search for task in array

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    if (tasks[index].handle == handle) {
        vTaskDelete(handle);
        memset(&tasks[index], 0, sizeof(RTOS_Task));
        return ESP_OK;
    }

    // NO se ha encontrado la tarea
    return ESP_FAIL;
}
```

4.23.3.5 KM_RTOS_Destroy()

```
void KM_RTOS_Destroy (
              void   )
```

Destroys all tasks managed by the RTOS task system and resets the task array.

This function iterates through the internal `tasks` array and deletes all active FreeRTOS tasks using `vTask←Delete`. After deleting tasks, it clears the `tasks` array to mark all slots as free and inactive.

The function wraps the entire operation in a critical section (`taskENTER_CRITICAL` / `taskEXIT_CRITICAL`) to prevent the scheduler from switching tasks while the array is being modified. This ensures thread-safety during the destruction process.

Note

All tasks created through `KM_RTOS_CreateTask` will be stopped.

After calling this function, the internal task array is fully reset and ready for re-initialization.

Definition at line 52 of file km_rtos.c.
```
00052                         {
00053     for (int i = 0; i < RTOS_MAX_TASKS; i++) {
00054         if (tasks[i].name != NULL) {
00055             vTaskDelete(tasks[i].handle);
00056         }
00057     }
00058
00059     memset(tasks, 0, sizeof(tasks));
00060 }
```

4.23.3.6 KM_RTOS_Init()

```
void KM_RTOS_Init (
            void )
```

Initializes the RTOS task system.

This function resets the internal `tasks` array by zeroing all entries, effectively marking all task slots as free and inactive. It should be called once at system startup before creating or managing any tasks.

Note

This function does not create any FreeRTOS tasks; it only prepares the internal task array for use.

Definition at line 47 of file km_rtos.c.
```
00047                      {
00048     memset(tasks, 0, sizeof(tasks));
00049 }
```

4.23.3.7 KM_RTOS_RestartTask()

```
esp_err_t KM_RTOS_RestartTask (
            TaskHandle_t handle )
```

Restarts a specific FreeRTOS task by deleting and recreating it.

This function searches the internal `tasks` array for the task matching the provided handle. If found, it performs the following steps:

1. Makes a copy of the task structure to preserve its configuration.
2. Deletes the task using `KM_RTOS_DeleteTask`.
3. Re-adds the task using `KM_RTOS_AddTask` with the preserved configuration.

Parameters

handle	The FreeRTOS task handle to restart.

Returns

ESP_OK if the task was successfully restarted.

ESP_FAIL if the task handle was not found, deletion failed, or creation failed.

Note

This operation temporarily stops the task and replaces it with a new instance.

Definition at line 163 of file km_rtos.c.

```
                                                  {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return ESP_FAIL;

    RTOS_Task copy = tasks[index];

    // Destruir tarea
    if(KM_RTOS_DeleteTask(handle) == ESP_FAIL) return ESP_FAIL;

    // Crear tarea
    if(KM_RTOS_AddTask(copy) == ESP_FAIL) return ESP_FAIL;

    // Reiniciado correctamente
    return ESP_OK;
}
```

4.23.3.8 KM_RTOS_ResumeTask()

```
esp_err_t KM_RTOS_ResumeTask (
              TaskHandle_t handle )
```

Resumes a specific FreeRTOS task and marks it as active in the internal task array.

This function searches the internal `tasks` array for the task corresponding to the provided FreeRTOS handle. If found and the task is currently suspended, it calls `vTaskResume` to restart the task's execution and sets its `active` flag to 1.

Parameters

handle	The FreeRTOS task handle to resume.

Returns

ESP_OK if the task was successfully resumed.

ESP_FAIL if the task handle was not found or the task was already active.

Note

Only affects the specified task; other tasks and array slots remain untouched.

Safe to call under normal operation, as it modifies only the targeted task entry.

Definition at line 146 of file km_rtos.c.

```
                                                 {

    int8_t index = KM_RTOS_FindTask(handle);
    if (index == -1) return 0;

    if (tasks[index].handle == handle && !tasks[index].active) {
```

```
00152         vTaskResume(handle);
00153         tasks[index].active = 1;
00154 
00155         return ESP_OK;
00156     }
00157 
00158     // No se ha encontrado la tarea o ya estaba activa
00159     return ESP_FAIL;
00160 }
```

4.23.3.9 KM_RTOS_SuspendTask()

```
esp_err_t KM_RTOS_SuspendTask (
              TaskHandle_t handle )
```

Suspends a specific FreeRTOS task and marks it as inactive in the internal task array.

This function searches the internal `tasks` array for the task corresponding to the provided FreeRTOS handle. If found and the task is currently active, it calls `vTaskSuspend` to pause the task's execution and sets its `active` flag to 0.

Parameters

handle	The FreeRTOS task handle to suspend.

Returns

ESP_OK if the task was successfully suspended.

ESP_FAIL if the task handle was not found or the task was already suspended.

Note

Only affects the specified task; other tasks and array slots remain untouched.

Safe to call under normal operation, as it modifies only the targeted task entry.

Definition at line 130 of file km_rtos.c.

```
00130                                                 {
00131 
00132     int8_t index = KM_RTOS_FindTask(handle);
00133     if (index == -1) return ESP_FAIL;
00134 
00135     if (tasks[index].handle == handle && tasks[index].active) {
00136         vTaskSuspend(handle);
00137         tasks[index].active = 0;
00138         return ESP_OK;
00139     }
00140 
00141     // No se ha encontrado la tarea o ya estaba suspendida
00142     return ESP_FAIL;
00143 }
```

4.24 km_rtos.h

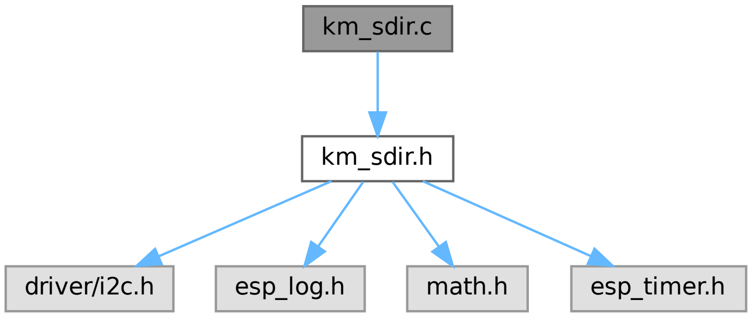
Go to the documentation of this file.

```
/*****************************************************************************
 * @file    Km_rtos.h
 * @brief   Public API for the KM_RTOS task management library.
 *
 * This header provides the public interface for the KM_RTOS library, which
 * offers a structured and safe way to manage FreeRTOS tasks. Through this API,
 * users can create, delete, suspend, resume, restart tasks, and change task
 * priorities without directly manipulating internal data structures.
 *
 * The library ensures:
 *  - Controlled access to a static internal task array.
 *  - Consistent task creation with default parameters if needed.
 *  - Safe periodic execution of tasks via the KM_RTOS_TaskWrapper.
 *  - Simplified FreeRTOS task management while maintaining thread safety.
 *
 * Usage:
 *  1. Call KM_RTOS_Init() before creating any tasks.
 *  2. Use KM_RTOS_CreateTask / KM_RTOS_AddTask to define and add tasks.
 *  3. Manage tasks using KM_RTOS_DeleteTask, KM_RTOS_SuspendTask, KM_RTOS_ResumeTask,
 *     KM_RTOS_RestartTask, and KM_RTOS_ChangePriority.
 *
 * Author: Adrian Navarredonda Arizaleta
 * Date: 24-01-2026
 * Version: 1.0
 *****************************************************************************/

#ifndef KM_RTOS_H
#define KM_RTOS_H

/******************************* INCLUDES *************************************/
// Includes necesarios para la API pública
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <stdint.h>
#include "esp_log.h" // Para log

/******************************* DEFINES PUBLICOS *****************************/
// Constantes, flags o configuraciones visibles desde fuera de la librería
#define RTOS_MAX_TASKS          10  // Número máximo de tareas que puede manejar la librería

typedef void (*KM_RTOS_TaskFunction_t)(void *context);

typedef struct {
    TaskHandle_t handle;
    const char *name;
    KM_RTOS_TaskFunction_t taskFn;
    void *context;
    uint32_t period_ms;
    uint16_t stackSize;
    UBaseType_t priority;
    uint8_t active;
} RTOS_Task;

/******************************* VARIABLES PÚBLICAS ***************************/
// Variables globales visibles (si realmente se necesitan)

// extern int ejemplo_variable_publica;

/******************************* FUNCIONES PÚBLICAS ***************************/
void KM_RTOS_Init(void);

void KM_RTOS_Destroy(void);

RTOS_Task KM_COMS_CreateTask(char *name, KM_RTOS_TaskFunction_t taskFn, void *context,
        uint32_t period_ms, uint16_t stackSize, UBaseType_t priority, uint8_t active);

esp_err_t KM_RTOS_AddTask(RTOS_Task task);

esp_err_t KM_RTOS_DeleteTask(TaskHandle_t handle);

esp_err_t KM_RTOS_SuspendTask(TaskHandle_t handle);

esp_err_t KM_RTOS_ResumeTask(TaskHandle_t handle);

esp_err_t KM_RTOS_RestartTask(TaskHandle_t handle);

esp_err_t KM_RTOS_ChangePriority(TaskHandle_t handle, uint8_t newPriority);

#endif /* NOMBRE_LIBRERIA_H */

```

4.25 km_sdir.c File Reference

```
#include "km_sdir.h"
```

Include dependency graph for km_sdir.c:



Macros

- #define TAG "KM_SDIR"
- #define I2C_MASTER_TIMEOUT_MS 1000

Functions

- static int8_t KM_SDIR_ReadRegisters (sensor_struct ∗sensor, uint8_t reg, uint8_t ∗data, uint8_t len)
- static float KM_SDIR_ReadAngle (sensor_struct ∗sensor)
- sensor_struct KM_SDIR_Init (int8_t max_error_count)

    Inicializa una conexion i2c.
- int8_t KM_SDIR_Begin (sensor_struct ∗sensor, gpio_num_t sda, gpio_num_t scl)
- uint16_t KM_SDIR_ReadRaw (sensor_struct ∗sensor)
- float KM_SDIR_ReadAngleRadians (sensor_struct ∗sensor)
- float KM_SDIR_ReadAngleDegrees (sensor_struct ∗sensor)
- int8_t KM_SDIR_isConnected (sensor_struct ∗sensor)
- int8_t KM_SDIR_ResetI2C (sensor_struct ∗sensor)
- void KM_SDIR_setCenterOffset (sensor_struct ∗sensor, uint16_t offset)

4.25.1 Macro Definition Documentation

4.25.1.1 I2C_MASTER_TIMEOUT_MS

```
#define I2C_MASTER_TIMEOUT_MS 1000
```

Definition at line 12 of file km_sdir.c.

4.25.1.2 TAG

```
#define TAG "KM_SDIR"
```

Definition at line 11 of file km_sdir.c.

4.25.2 Function Documentation

4.25.2.1 KM_SDIR_Begin()

```
int8_t KM_SDIR_Begin (
            sensor_struct * sensor,
            gpio_num_t sda,
            gpio_num_t scl )
```

Definition at line 34 of file km_sdir.c.

```
00034                                                                              {
00035     i2c_config_t conf = {
00036         .mode = I2C_MODE_MASTER,
00037         .sda_io_num = sda,
00038         .sda_pullup_en = GPIO_PULLUP_ENABLE,
00039         .scl_io_num = scl,
00040         .scl_pullup_en = GPIO_PULLUP_ENABLE,
00041         .master.clk_speed = 400000
00042     };
00043
00044     i2c_param_config(I2C_NUM_0, &conf);
00045     if (i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0) != ESP_OK) {
00046         ESP_LOGE(TAG, "Failed to install I2C driver");
00047         return 0;
00048     }
00049
00050     uint8_t testData[2];
00051     sensor->connected = KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, testData, 2);
00052     if (sensor->connected) {
00053         ESP_LOGI(TAG, "AS5600 sensor initialized successfully");
00054     } else {
00055         ESP_LOGW(TAG, "AS5600 sensor not found!");
00056     }
00057
00058     return sensor->connected;
00059 }
```

4.25.2.2 KM_SDIR_Init()

```
sensor_struct KM_SDIR_Init (
            int8_t max_error_count )
```

Inicializa una conexion i2c.

Returns

retorna el struct conection i2c inicializado

Definition at line 20 of file km_sdir.c.

```
00020                                                    {
00021     sensor_struct sensor;
00022
00023     sensor.centerOffset = SENSOR_CENTER;
00024     sensor.connected = 0;
00025     sensor.errorCount = 0;
00026     sensor.lastRawValue = 0;
00027     sensor.lastReadTime = 0;
00028     sensor.max_error_count = max_error_count;
00029
00030     return sensor;
00031 }
```

4.25.2.3 KM_SDIR_isConnected()

```
int8_t KM_SDIR_isConnected (
              sensor_struct * sensor )
```

Definition at line 92 of file km_sdir.c.

```
00092                                              {
00093     return sensor->connected && (sensor->errorCount < sensor->max_error_count);
00094 }
```

4.25.2.4 KM_SDIR_ReadAngle()

```
static float KM_SDIR_ReadAngle (
              sensor_struct * sensor )  [static]
```

Definition at line 161 of file km_sdir.c.

```
00161                                                  {
00162     uint16_t raw = KM_SDIR_ReadRaw(sensor);
00163
00164     int16_t centered = (int16_t)raw - sensor->centerOffset;
00165     float angle = ((float)centered / (float)SENSOR_MAX) * 2.0f * MAX_RAD;
00166
00167     return angle;
00168 }
```

4.25.2.5 KM_SDIR_ReadAngleDegrees()

```
float KM_SDIR_ReadAngleDegrees (
              sensor_struct * sensor )
```

Definition at line 87 of file km_sdir.c.

```
00087                                                  {
00088     return KM_SDIR_ReadAngle(sensor) * 180.0f / PI;
00089 }
```

4.25.2.6 KM_SDIR_ReadAngleRadians()

```
float KM_SDIR_ReadAngleRadians (
              sensor_struct * sensor )
```

Definition at line 82 of file km_sdir.c.

```
00082                                                  {
00083     return KM_SDIR_ReadAngle(sensor);
00084 }
```

4.25.2.7 KM_SDIR_ReadRaw()

```
uint16_t KM_SDIR_ReadRaw (
              sensor_struct * sensor )
```

Definition at line 62 of file km_sdir.c.

```
00062                                            {
00063     uint8_t data[2];
00064
00065     if (KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, data, 2)) {
00066         sensor->lastRawValue = ((uint16_t)data[0] « 8) | data[1];
00067         sensor->lastReadTime = esp_timer_get_time(); // microsegundos desde boot
00068         sensor->errorCount = 0;
00069         return sensor->lastRawValue;
00070     } else {
00071         sensor->errorCount++;
00072         if (sensor->errorCount >= sensor->max_error_count) {
00073             ESP_LOGW(TAG, "Error reading sensor, resetting I2C");
00074             KM_SDIR_ResetI2C(sensor);
00075             sensor->errorCount = 0;
00076         }
00077         return sensor->lastRawValue;
00078     }
00079 }
```

4.25.2.8 KM_SDIR_ReadRegisters()

```
static int8_t KM_SDIR_ReadRegisters (
               sensor_struct * sensor,
               uint8_t reg,
               uint8_t * data,
               uint8_t len )  [static]
```

Definition at line 132 of file km_sdir.c.

```
                                                                                                    {
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (AS5600_ADDR « 1) | I2C_MASTER_WRITE, true);
    i2c_master_write_byte(cmd, reg, true);
    i2c_master_stop(cmd);
    esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
    i2c_cmd_link_delete(cmd);

    if (ret != ESP_OK) {
        return 0;
    }

    cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (AS5600_ADDR « 1) | I2C_MASTER_READ, true);
    if (len > 1) {
        i2c_master_read(cmd, data, len - 1, I2C_MASTER_ACK);
    }
    i2c_master_read_byte(cmd, data + len - 1, I2C_MASTER_NACK);
    i2c_master_stop(cmd);

    ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
    i2c_cmd_link_delete(cmd);

    return (ret == ESP_OK) ? 1 : 0;
}
```

4.25.2.9 KM_SDIR_ResetI2C()

```
int8_t KM_SDIR_ResetI2C (
               sensor_struct * sensor )
```

Definition at line 97 of file km_sdir.c.

```
                                                    {
    i2c_driver_delete(I2C_NUM_0);
    vTaskDelay(pdMS_TO_TICKS(100));

    // Reinstala I2C con los mismos parámetros guardados en el sensor
    // Necesitas que sensor tenga guardados i2c_port, sda, scl
    // Por simplicidad vamos a usar I2C_NUM_0 y pines por defecto
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = 21,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_io_num = 22,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000
    };
    i2c_param_config(I2C_NUM_0, &conf);
    if (i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0) != ESP_OK) {
        ESP_LOGE(TAG, "Failed to reinstall I2C driver");
        return 0;
    }

    uint8_t testData[2];
    sensor->connected = KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, testData, 2);
    return sensor->connected;
}
```

4.25.2.10 KM_SDIR_setCenterOffset()

```
void KM_SDIR_setCenterOffset (
            sensor_struct * sensor,
            uint16_t offset )
```

Definition at line 124 of file km_sdir.c.

```
                                                                   {
    sensor->centerOffset = offset;
    ESP_LOGI(TAG, "AS5600 center offset set to: %d", offset);
}
```

4.26 km_sdir.c

Go to the documentation of this file.

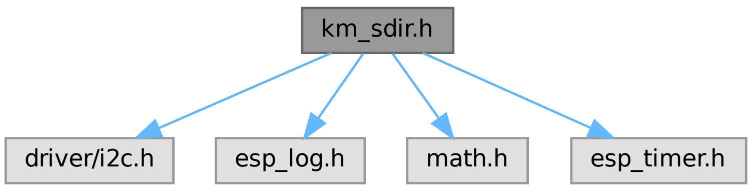
```
/*****************************************************************************
* @file    km_sdir.c
* @brief   Librería para usar el sensor de dirección (ESP-IDF)
* @author Adrian Navarredonda
* @date 31-01-2026
****************************************************************************/

#include "km_sdir.h"

/******************************* MACROS PRIVADAS *****************************/
#define TAG "KM_SDIR"
#define I2C_MASTER_TIMEOUT_MS 1000

/******************************* DECLARACION FUNCIONES PRIVADAS ***************/
static int8_t KM_SDIR_ReadRegisters(sensor_struct *sensor, uint8_t reg, uint8_t* data, uint8_t len);
static float KM_SDIR_ReadAngle(sensor_struct *sensor);

/******************************* FUNCIONES PÚBLICAS **************************/

sensor_struct KM_SDIR_Init(int8_t max_error_count) {
    sensor_struct sensor;

    sensor.centerOffset = SENSOR_CENTER;
    sensor.connected = 0;
    sensor.errorCount = 0;
    sensor.lastRawValue = 0;
    sensor.lastReadTime = 0;
    sensor.max_error_count = max_error_count;

    return sensor;
}

// Inicializa I2C
int8_t KM_SDIR_Begin(sensor_struct *sensor, gpio_num_t sda, gpio_num_t scl) {
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = sda,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_io_num = scl,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000
    };

    i2c_param_config(I2C_NUM_0, &conf);
    if (i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0) != ESP_OK) {
        ESP_LOGE(TAG, "Failed to install I2C driver");
        return 0;
    }

    uint8_t testData[2];
    sensor->connected = KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, testData, 2);
    if (sensor->connected) {
        ESP_LOGI(TAG, "AS5600 sensor initialized successfully");
    } else {
        ESP_LOGW(TAG, "AS5600 sensor not found!");
    }

    return sensor->connected;
}

// Leer valor crudo (0-4095)
uint16_t KM_SDIR_ReadRaw(sensor_struct *sensor) {
```

```c
    uint8_t data[2];

    if (KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, data, 2)) {
        sensor->lastRawValue = ((uint16_t)data[0] « 8) | data[1];
        sensor->lastReadTime = esp_timer_get_time(); // microsegundos desde boot
        sensor->errorCount = 0;
        return sensor->lastRawValue;
    } else {
        sensor->errorCount++;
        if (sensor->errorCount >= sensor->max_error_count) {
            ESP_LOGW(TAG, "Error reading sensor, resetting I2C");
            KM_SDIR_ResetI2C(sensor);
            sensor->errorCount = 0;
        }
        return sensor->lastRawValue;
    }
}

// Leer ángulo en radianes
float KM_SDIR_ReadAngleRadians(sensor_struct *sensor) {
    return KM_SDIR_ReadAngle(sensor);
}

// Leer ángulo en grados
float KM_SDIR_ReadAngleDegrees(sensor_struct *sensor) {
    return KM_SDIR_ReadAngle(sensor) * 180.0f / PI;
}

// Verifica si el sensor está conectado
int8_t KM_SDIR_isConnected(sensor_struct *sensor) {
    return sensor->connected && (sensor->errorCount < sensor->max_error_count);
}

// Reset de I2C
int8_t KM_SDIR_ResetI2C(sensor_struct *sensor) {
    i2c_driver_delete(I2C_NUM_0);
    vTaskDelay(pdMS_TO_TICKS(100));

    // Reinstala I2C con los mismos parámetros guardados en el sensor
    // Necesitas que sensor tenga guardados i2c_port, sda, scl
    // Por simplicidad vamos a usar I2C_NUM_0 y pines por defecto
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = 21,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_io_num = 22,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000
    };
    i2c_param_config(I2C_NUM_0, &conf);
    if (i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0) != ESP_OK) {
        ESP_LOGE(TAG, "Failed to reinstall I2C driver");
        return 0;
    }

    uint8_t testData[2];
    sensor->connected = KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, testData, 2);
    return sensor->connected;
}

// Ajuste de offset
void KM_SDIR_setCenterOffset(sensor_struct *sensor, uint16_t offset) {
    sensor->centerOffset = offset;
    ESP_LOGI(TAG, "AS5600 center offset set to: %d", offset);
}

/******************************* FUNCIONES PRIVADAS ***************************/

// Leer registros I2C
static int8_t KM_SDIR_ReadRegisters(sensor_struct *sensor, uint8_t reg, uint8_t* data, uint8_t len) {
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (AS5600_ADDR « 1) | I2C_MASTER_WRITE, true);
    i2c_master_write_byte(cmd, reg, true);
    i2c_master_stop(cmd);
    esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
    i2c_cmd_link_delete(cmd);

    if (ret != ESP_OK) {
        return 0;
    }

    cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (AS5600_ADDR « 1) | I2C_MASTER_READ, true);
    if (len > 1) {
        i2c_master_read(cmd, data, len - 1, I2C_MASTER_ACK);
    }
    i2c_master_read_byte(cmd, data + len - 1, I2C_MASTER_NACK);
    i2c_master_stop(cmd);

    ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
    i2c_cmd_link_delete(cmd);

    return (ret == ESP_OK) ? 1 : 0;
}

// Devuelve ángulo en radianes (-PI a PI)
static float KM_SDIR_ReadAngle(sensor_struct *sensor) {
    uint16_t raw = KM_SDIR_ReadRaw(sensor);

    int16_t centered = (int16_t)raw - sensor->centerOffset;
    float angle = ((float)centered / (float)SENSOR_MAX) * 2.0f * MAX_RAD;

    return angle;
}

/****************************** FIN DE ARCHIVO ******************************/
```

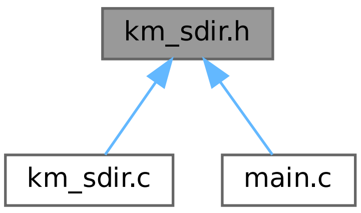
4.27 km_sdir.h File Reference

```
#include "driver/i2c.h"
#include "esp_log.h"
#include <math.h>
#include "esp_timer.h"
```

Include dependency graph for km_sdir.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct sensor_struct

  Structure that reperesents the direction sensor.

Macros

- #define PI 3.1415

Enumerations

- enum conection_constans { AS5600_ADDR = 0x36 , AS5600_ANGLE_MSB = 0x0C , AS5600_ANGLE_LSB = 0x0D }
- enum sensor_constans { SENSOR_MIN = 0 , SENSOR_MAX = 4095 , SENSOR_CENTER = 2048 }

Functions

- sensor_struct KM_SDIR_Init (int8_t max_error_count)

  Inicializa una conexion i2c.
- int8_t KM_SDIR_Begin (sensor_struct ∗sensor, gpio_num_t sdaPin, gpio_num_t sclPin)
- uint16_t KM_SDIR_ReadRaw (sensor_struct ∗sensor)
- float KM_SDIR_ReadAngleRadians (sensor_struct ∗sensor)
- float KM_SDIR_ReadAngleDegrees (sensor_struct ∗sensor)
- int8_t KM_SDIR_isConnected (sensor_struct ∗sensor)
- int8_t KM_SDIR_ResetI2C (sensor_struct ∗sensor)
- void KM_SDIR_setCenterOffset (sensor_struct ∗sensor, uint16_t offset)

Variables

- const float MAX_RAD = PI

4.27.1 Macro Definition Documentation

4.27.1.1 PI

```
#define PI 3.1415
```

Definition at line 23 of file km_sdir.h.

4.27.2 Enumeration Type Documentation

4.27.2.1 conection_constans

```
enum conection_constans
```

Enumerator

AS5600_ADDR	
AS5600_ANGLE_MSB	
AS5600_ANGLE_LSB	

Definition at line 40 of file km_sdir.h.

```
                 {
         AS5600_ADDR = 0x36,
         AS5600_ANGLE_MSB = 0x0C,
         AS5600_ANGLE_LSB = 0x0D
 } conection_constans;
```

4.27.2.2 sensor_constans

```
enum sensor_constans
```

Enumerator

SENSOR_MIN	
SENSOR_MAX	
SENSOR_CENTER	

Definition at line 46 of file km_sdir.h.

```
                 {
         SENSOR_MIN = 0,
         SENSOR_MAX = 4095,
         SENSOR_CENTER = 2048,
 } sensor_constans;
```

4.27.3 Function Documentation

4.27.3.1 KM_SDIR_Begin()

```
int8_t KM_SDIR_Begin (
              sensor_struct * sensor,
              gpio_num_t sdaPin,
              gpio_num_t sclPin )
```

Definition at line 34 of file km_sdir.c.

```
                                                                                 {
     i2c_config_t conf = {
         .mode = I2C_MODE_MASTER,
         .sda_io_num = sda,
         .sda_pullup_en = GPIO_PULLUP_ENABLE,
         .scl_io_num = scl,
         .scl_pullup_en = GPIO_PULLUP_ENABLE,
         .master.clk_speed = 400000
     };

     i2c_param_config(I2C_NUM_0, &conf);
     if (i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0) != ESP_OK) {
         ESP_LOGE(TAG, "Failed to install I2C driver");
         return 0;
     }

     uint8_t testData[2];
     sensor->connected = KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, testData, 2);
     if (sensor->connected) {
         ESP_LOGI(TAG, "AS5600 sensor initialized successfully");
     } else {
         ESP_LOGW(TAG, "AS5600 sensor not found!");
     }

     return sensor->connected;
 }
```

4.27.3.2 KM_SDIR_Init()

```
sensor_struct KM_SDIR_Init (
              int8_t max_error_count )
```

Inicializa una conexion i2c.

Returns

retorna el struct conection i2c inicializado

Definition at line 20 of file km_sdir.c.

```
                                                  {
    sensor_struct sensor;

    sensor.centerOffset = SENSOR_CENTER;
    sensor.connected = 0;
    sensor.errorCount = 0;
    sensor.lastRawValue = 0;
    sensor.lastReadTime = 0;
    sensor.max_error_count = max_error_count;

    return sensor;
}
```

4.27.3.3 KM_SDIR_isConnected()

```
int8_t KM_SDIR_isConnected (
              sensor_struct * sensor )
```

Definition at line 92 of file km_sdir.c.

```
                                                 {
    return sensor->connected && (sensor->errorCount < sensor->max_error_count);
}
```

4.27.3.4 KM_SDIR_ReadAngleDegrees()

```
float KM_SDIR_ReadAngleDegrees (
              sensor_struct * sensor )
```

Definition at line 87 of file km_sdir.c.

```
                                                     {
    return KM_SDIR_ReadAngle(sensor) * 180.0f / PI;
}
```

4.27.3.5 KM_SDIR_ReadAngleRadians()

```
float KM_SDIR_ReadAngleRadians (
              sensor_struct * sensor )
```

Definition at line 82 of file km_sdir.c.

```
                                                     {
    return KM_SDIR_ReadAngle(sensor);
}
```

4.27.3.6 KM_SDIR_ReadRaw()

```
uint16_t KM_SDIR_ReadRaw (
            sensor_struct * sensor )
```

Definition at line 62 of file km_sdir.c.

```
00062                                               {
00063     uint8_t data[2];
00064 
00065     if (KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, data, 2)) {
00066         sensor->lastRawValue = ((uint16_t)data[0] « 8) | data[1];
00067         sensor->lastReadTime = esp_timer_get_time(); // microsegundos desde boot
00068         sensor->errorCount = 0;
00069         return sensor->lastRawValue;
00070     } else {
00071         sensor->errorCount++;
00072         if (sensor->errorCount >= sensor->max_error_count) {
00073             ESP_LOGW(TAG, "Error reading sensor, resetting I2C");
00074             KM_SDIR_ResetI2C(sensor);
00075             sensor->errorCount = 0;
00076         }
00077         return sensor->lastRawValue;
00078     }
00079 }
```

4.27.3.7 KM_SDIR_ResetI2C()

```
int8_t KM_SDIR_ResetI2C (
            sensor_struct * sensor )
```

Definition at line 97 of file km_sdir.c.

```
00097                                              {
00098     i2c_driver_delete(I2C_NUM_0);
00099     vTaskDelay(pdMS_TO_TICKS(100));
00100 
00101     // Reinstala I2C con los mismos parámetros guardados en el sensor
00102     // Necesitas que sensor tenga guardados i2c_port, sda, scl
00103     // Por simplicidad vamos a usar I2C_NUM_0 y pines por defecto
00104     i2c_config_t conf = {
00105         .mode = I2C_MODE_MASTER,
00106         .sda_io_num = 21,
00107         .sda_pullup_en = GPIO_PULLUP_ENABLE,
00108         .scl_io_num = 22,
00109         .scl_pullup_en = GPIO_PULLUP_ENABLE,
00110         .master.clk_speed = 400000
00111     };
00112     i2c_param_config(I2C_NUM_0, &conf);
00113     if (i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0) != ESP_OK) {
00114         ESP_LOGE(TAG, "Failed to reinstall I2C driver");
00115         return 0;
00116     }
00117 
00118     uint8_t testData[2];
00119     sensor->connected = KM_SDIR_ReadRegisters(sensor, AS5600_ANGLE_MSB, testData, 2);
00120     return sensor->connected;
00121 }
```

4.27.3.8 KM_SDIR_setCenterOffset()

```
void KM_SDIR_setCenterOffset (
            sensor_struct * sensor,
            uint16_t offset )
```

Definition at line 124 of file km_sdir.c.

```
00124                                                                    {
00125     sensor->centerOffset = offset;
00126     ESP_LOGI(TAG, "AS5600 center offset set to: %d", offset);
00127 }
```

4.27.4 Variable Documentation

4.27.4.1 MAX_RAD

```
const float MAX_RAD = PI
```

Definition at line 55 of file km_sdir.h.

4.28 km_sdir.h

Go to the documentation of this file.

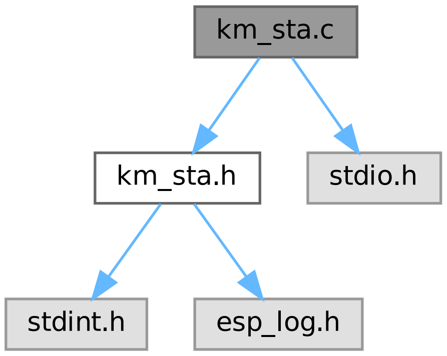
```
00001 /*********************************************************************************
00002  * @file     km_dsen.h
00003  * @brief    Libreria para usar el sensor de la direccion.
00004  * @author   Adrian Navarredonda Arizaleta
00005  * @date     27-01-2026
00006  * @version 1.0
00007  *******************************************************************************/
00008 
00009 #ifndef KM_SDIR_H
00010 #define KM_SDIR_H
00011 
00012 /******************************** INCLUDES ***********************************/
00013 // Includes necesarios para la API pública
00014 #include "driver/i2c.h"
00015 #include "esp_log.h"
00016 #include <math.h>
00017 #include "esp_timer.h"
00018 
00019 /******************************** DEFINES PÚBLICAS ***************************/
00020 // Constantes, flags o configuraciones visibles desde fuera de la librería
00021 
00022 
00023 #define PI 3.1415
00024 
00025 /******************************** TIPOS PÚBLICOS *****************************/
00026 // Estructuras, enums, typedefs públicos
00030 typedef struct {
00031     uint8_t connected;
00032     uint8_t errorCount;
00033     uint8_t max_error_count;
00034     uint64_t lastReadTime;
00035     uint16_t lastRawValue;
00036     uint16_t centerOffset;
00037 } sensor_struct;
00038 
00039 // Represents the conection constans
00040 typedef enum {
00041     AS5600_ADDR = 0x36,
00042     AS5600_ANGLE_MSB = 0x0C,
00043     AS5600_ANGLE_LSB = 0x0D
00044 } conection_constans;
00045 
00046 typedef enum {
00047     SENSOR_MIN = 0,
00048     SENSOR_MAX = 4095,
00049     SENSOR_CENTER = 2048,
00050 } sensor_constans;
00051 
00052 /******************************** VARIABLES PÚBLICAS ***************************/
00053 // Variables globales visibles (si realmente se necesitan)
00054 
00055 const float MAX_RAD = PI;
00056 
00057 // extern int ejemplo_variable_publica;
00058 
00059 /******************************** FUNCIONES PÚBLICAS ***************************/
00060 
00065 sensor_struct KM_SDIR_Init(int8_t max_error_count);
00066 
00067 // Initialize the sensor
00068 int8_t KM_SDIR_Begin(sensor_struct *sensor, gpio_num_t sdaPin, gpio_num_t sclPin);
00069 
00070 // Read raw sensor value (0-4095)
00071 uint16_t KM_SDIR_ReadRaw(sensor_struct *sensor);
00072 
00073 // Read angle in radians (-PI to PI)
```

```
float KM_SDIR_ReadAngleRadians(sensor_struct *sensor);

// Read angle in degrees (-180 to 180)
float KM_SDIR_ReadAngleDegrees(sensor_struct *sensor);

// Check if sensor is connected
int8_t KM_SDIR_isConnected(sensor_struct *sensor);

// Reset I2C communication if errors occur
int8_t KM_SDIR_ResetI2C(sensor_struct *sensor);

// Get center offset for calibration
void KM_SDIR_setCenterOffset(sensor_struct *sensor, uint16_t offset);

#endif /* KM_SDIR_H */
```

4.29 km_sta.c File Reference

```
#include "km_sta.h"
#include <stdio.h>
```

Include dependency graph for km_sta.c:



4.30 km_sta.c

Go to the documentation of this file.

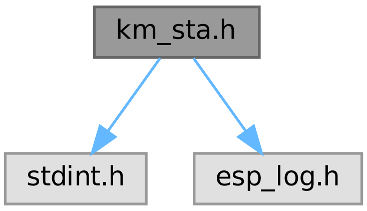
```
/******************************************************************************
 * @file    nombre_libreria.c
 * @brief   Implementación de la librería.
 ******************************************************************************/

 #include "km_sta.h"
 #include <stdio.h>   // solo si es necesario para debug interno

 /******************************** INCLUDES INTERNOS ***************************/
 // Headers internos opcionales, dependencias privadas

 /******************************** MACROS PRIVADAS ********************************/
 // Constantes internas, flags de debug
 // #define LIBRERIA_DEBUG 1

 /******************************** VARIABLES PRIVADAS ***************************/
 // Variables globales internas (static)

 /******************************** DECLARACION FUNCIONES PRIVADAS ***************/

```

```

/******************************* FUNCIONES PÚBLICAS ***************************/
// int libreria_operacion(int valor) {
//     ...
// }

/******************************* FUNCIONES PRIVADAS ***************************/
// static void funcion_privada(void);

/******************************* FIN DE ARCHIVO ********************************/

```

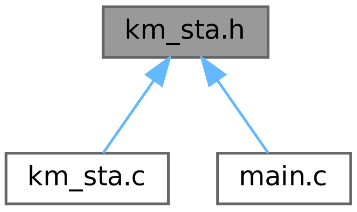
4.31 km_sta.h File Reference

```
#include <stdint.h>
#include "esp_log.h"
```

Include dependency graph for km_sta.h:



This graph shows which files directly or indirectly include this file:



4.32 km_sta.h

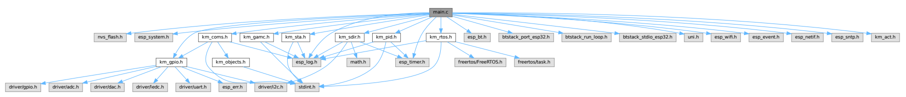
Go to the documentation of this file.

```
/*****************************************************************************
 * @file     nombre_libreria.h
 * @brief    Interfaz pública de la librería.
 * @author   Autor
 * @date    DD-MM-AAAA
 * @version 1.0
 ******************************************************************************/

 #ifndef NOMBRE_LIBRERIA_H
 #define NOMBRE_LIBRERIA_H

 /******************************* INCLUDES ***********************************/
 // Includes necesarios para la API pública
 #include <stdint.h>
 #include "esp_log.h" // Para log

 /******************************* DEFINES PÚBLICAS ***************************/
 // Constantes, flags o configuraciones visibles desde fuera de la librería

 /******************************* TIPOS PÚBLICOS *****************************/
 // Estructuras, enums, typedefs públicos

 /******************************* VARIABLES PÚBLICAS *************************/
 // Variables globales visibles (si realmente se necesitan)

 // extern int ejemplo_variable_publica;

 /******************************* FUNCIONES PÚBLICAS *************************/
 // int libreria_init(void);

 // int libreria_operacion(int valor);

 #endif /* NOMBRE_LIBRERIA_H */
```

4.33 main.c File Reference

```
#include "nvs_flash.h"
#include "esp_system.h"
#include "esp_log.h"
#include "esp_bt.h"
#include <btstack_port_esp32.h>
#include <btstack_run_loop.h>
#include <btstack_stdio_esp32.h>
#include <uni.h>
#include "esp_timer.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_netif.h"
#include "esp_sntp.h"
#include "km_act.h"
#include "km_coms.h"
#include "km_gamc.h"
#include "km_pid.h"
#include "km_rtos.h"
#include "km_sdir.h"
#include "km_sta.h"
#include "km_gpio.h"
```

Include dependency graph for main.c:



Macros

- #define MAX_ERROR_COUNT_SDIR 10

Functions

- void system_init (void)
- void app_main (void)

Variables

- static const char ∗ TAG = "MAIN"

4.33.1 Macro Definition Documentation

4.33.1.1 MAX_ERROR_COUNT_SDIR

```
#define MAX_ERROR_COUNT_SDIR 10
```

Definition at line 72 of file main.c.

4.33.2 Function Documentation

4.33.2.1 app_main()

```
void app_main (
            void )
```

Definition at line 121 of file main.c.

```
                     {

    // Habilitar logs por UART0 (USB) para debug
    esp_log_level_set("*", ESP_LOG_INFO);

    ESP_LOGI(TAG, "ESP iniciando...");

    // Inicializa Bluetooth, Bluepad32, NVS, etc.
    // init_bluetooth();

    // Inicia todas las librerias que se necesitan
    system_init();

    // Creacion de toda las tareas de FreeRTOS
    // Tareas de libreria de comunicaciones
    // Tareas de libreria de maquina de estado

    // Ejecutar loop de BTstack (bloquea dentro de app_main, pero otras tareas siguen)
    //btstack_run_loop_execute();
}
```

4.33.2.2 system_init()

```
void system_init (
            void  )
```

Definition at line 74 of file main.c.

```
00074                        {
00075
00076     // Initialize hardware
00077     if(KM_GPIO_Init() != ESP_OK)
00078         ESP_LOGE(TAG, "Error inicializando libreria gpio\n");
00079
00080     // Initialize I2C
00081     if (KM_GPIO_I2CInit() != ESP_OK)
00082         ESP_LOGE(TAG, "Error inicializando libreria i2c\n");
00083
00084     // Initialise tasks
00085     KM_RTOS_Init();
00086
00087     // Initialise comunications on UART0 (USB to Orin)
00088     if (KM_COMS_Init(UART_NUM_0) != ESP_OK)
00089         ESP_LOGE(TAG, "Error inicializando libreria de comunicaciones");
00090
00091     // KM_SDIR_Init(MAX_ERROR_COUNT_SDIR);
00092
00093     // KM_ACT_Begin();
00094     // KM_PID_Init();
00095
00096     // Test bidirectional: send heartbeat + receive from Orin
00097     uint8_t payload[4] = {0xDE, 0xAD, 0xBE, 0xEF};
00098     uint32_t loop_count = 0;
00099
00100     while (true) {
00101         // TX: send heartbeat to Orin every 10 iterations (~1s)
00102         if (loop_count % 10 == 0) {
00103             KM_COMS_SendMsg(ESP_HEARTBEAT, payload, sizeof(payload));
00104             ESP_LOGI(TAG, "TX: heartbeat sent");
00105         }
00106
00107         // RX: read bytes from UART2 into internal buffer
00108         km_coms_ReceiveMsg();
00109
00110         // RX: parse complete frames and dispatch to objects
00111         KM_COMS_ProccessMsgs();
00112
00113         loop_count++;
00114         vTaskDelay(pdMS_TO_TICKS(100));
00115     }
00116 }
```

4.33.3 Variable Documentation

4.33.3.1 TAG

```
const char* TAG = "MAIN"  [static]
```

Definition at line 35 of file main.c.

4.34 main.c

Go to the documentation of this file.

```
00001 #include "nvs_flash.h"          // Obligatorio para almacenar parámetros BT/WiFi
00002 #include "esp_system.h"         // Funciones básicas de ESP32
00003 #include "esp_log.h"            // Logging
00004
00005 #include "esp_bt.h"             // Core BT
00006 // #include "esp_bt_main.h"        // Funciones de inicio BT
00007 // #include "esp_bt_device.h"      // Información del dispositivo
00008 // #include "esp_gap_ble_api.h"    // GAP BLE (escaneo, publicidad)
00009 // #include "esp_gatts_api.h"      // Servidor GATT
```

```c
// #include "esp_gatt_common_api.h"// Funciones GATT comunes

#include <btstack_port_esp32.h>           // Puerto BTstack para ESP32
#include <btstack_run_loop.h>             // Loop de BTstack
#include <btstack_stdio_esp32.h>          // Consola BTstack
#include <uni.h>                           // Core Bluepad32 (unijoysticle)
// #include <platform/esp32/uni_platform_esp32.h> // Adaptación a ESP32

#include "esp_timer.h"       // Temporizadores de alta resolución
#include "esp_wifi.h"        // Para WiFi
#include "esp_event.h"       // Event loop de ESP-IDF
#include "esp_netif.h"       // Configuración de interfaces de red
#include "esp_sntp.h"        // Para sincronizar tiempo
#include "esp_system.h"      // Información de sistema

// Librerias propias
#include "km_act.h"
#include "km_coms.h"
#include "km_gamc.h"
#include "km_pid.h"
#include "km_rtos.h"
#include "km_sdir.h"
#include "km_sta.h"
#include "km_gpio.h"

static const char *TAG = "MAIN";

// ===========================
// Funciones auxiliares de Bluetooth
// ===========================
// void init_bluetooth(void) {
//     esp_err_t ret;

//     // Inicializar NVS (necesario para BT/WiFi)
//     ret = nvs_flash_init();
//     if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
//         ESP_ERROR_CHECK(nvs_flash_erase());
//         ret = nvs_flash_init();
//     }
//     ESP_ERROR_CHECK(ret);

//     // Inicializar controlador BT
//     esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
//     ESP_ERROR_CHECK(esp_bt_controller_init(&bt_cfg));
//     ESP_ERROR_CHECK(esp_bt_controller_enable(ESP_BT_MODE_BTDM)); // BLE + Classic

//     // Inicializar Bluedroid
//     ESP_ERROR_CHECK(esp_bluedroid_init());
//     ESP_ERROR_CHECK(esp_bluedroid_enable());

//     // Inicializar consola BTstack
//     btstack_stdio_init();

//     // Configurar plataforma ESP32 para Bluepad32
//     uni_platform_set_custom(get_esp32_platform());

//     // Inicializar Bluepad32
//     uni_init(0, NULL);

//     ESP_LOGI(TAG, "Bluetooth y Bluepad32 inicializados");
// }

#define MAX_ERROR_COUNT_SDIR 10

void system_init(void) {

    // Initialize hardware
    if(KM_GPIO_Init() != ESP_OK)
        ESP_LOGE(TAG, "Error inicializando libreria gpio\n");

    // Initialize I2C
    if (KM_GPIO_I2CInit() != ESP_OK)
        ESP_LOGE(TAG, "Error inicializando libreria i2c\n");

    // Initialise tasks
    KM_RTOS_Init();

    // Initialise comunications on UART0 (USB to Orin)
    if (KM_COMS_Init(UART_NUM_0) != ESP_OK)
        ESP_LOGE(TAG, "Error inicializando libreria de comunicaciones");

    // KM_SDIR_Init(MAX_ERROR_COUNT_SDIR);

    // KM_ACT_Begin();
    // KM_PID_Init();

    // Test bidirectional: send heartbeat + receive from Orin
    uint8_t payload[4] = {0xDE, 0xAD, 0xBE, 0xEF};
    uint32_t loop_count = 0;

    while (true) {
        // TX: send heartbeat to Orin every 10 iterations (~1s)
        if (loop_count % 10 == 0) {
            KM_COMS_SendMsg(ESP_HEARTBEAT, payload, sizeof(payload));
            ESP_LOGI(TAG, "TX: heartbeat sent");
        }

        // RX: read bytes from UART2 into internal buffer
        km_coms_ReceiveMsg();

        // RX: parse complete frames and dispatch to objects
        KM_COMS_ProccessMsgs();

        loop_count++;
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}

// ===========================
// app_main - punto de entrada
// ===========================
void app_main(void) {

    // Habilitar logs por UART0 (USB) para debug
    esp_log_level_set("*", ESP_LOG_INFO);

    ESP_LOGI(TAG, "ESP iniciando...");

    // Inicializa Bluetooth, Bluepad32, NVS, etc.
    // init_bluetooth();

    // Inicia todas las librerias que se necesitan
    system_init();

    // Creacion de toda las tareas de FreeRTOS
    // Tareas de libreria de comunicaciones
    // Tareas de libreria de maquina de estado

    // Ejecutar loop de BTstack (bloquea dentro de app_main, pero otras tareas siguen)
    //btstack_run_loop_execute();
}

// ESP_LOGI(TAG, "Mensaje: %d", valor); → Información normal

// ESP_LOGW(TAG, "Mensaje: %s", cadena); → Advertencia

// ESP_LOGE(TAG, "Mensaje: %s", cadena); → Error

// ESP_LOGD(TAG, "Mensaje debug"); → Debug (solo si está habilitado)
```

Index