

KM_Medulla

1.0

Generated by Doxygen 1.9.8

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 km_coms_msg Struct Reference	5
3.1.1 Detailed Description	5
3.1.2 Member Data Documentation	6
3.1.2.1 crc	6
3.1.2.2 len	6
3.1.2.3 payload	6
3.1.2.4 type	6
3.2 PID_Controller Struct Reference	6
3.2.1 Detailed Description	7
3.2.2 Member Data Documentation	7
3.2.2.1 integral	7
3.2.2.2 integralMax	7
3.2.2.3 integralMin	7
3.2.2.4 kd	7
3.2.2.5 ki	8
3.2.2.6 kp	8
3.2.2.7 lastError	8
3.2.2.8 lastTime	8
3.2.2.9 outputMax	8
3.2.2.10 outputMin	8
3.3 RTOS_Task Struct Reference	9
3.3.1 Detailed Description	9
3.3.2 Member Data Documentation	9
3.3.2.1 active	9
3.3.2.2 context	9
3.3.2.3 handle	9
3.3.2.4 name	10
3.3.2.5 period_ms	10
3.3.2.6 priority	10
3.3.2.7 stackSize	10
3.3.2.8 taskFn	10
3.4 sensor_struct Struct Reference	10
3.4.1 Detailed Description	11
3.4.2 Member Data Documentation	11
3.4.2.1 centerOffset	11
3.4.2.2 connected	11

3.4.2.3 errorCount	11
3.4.2.4 lastRawValue	11
3.4.2.5 lastReadTime	11
3.4.2.6 max_error_count	11
4 File Documentation	13
4.1 km_coms.c File Reference	13
4.1.1 Macro Definition Documentation	14
4.1.1.1 KM_COMS_WAIT_SEM_AVAILABLE	14
4.1.2 Function Documentation	14
4.1.2.1 KM_COMS_crc8()	14
4.1.2.2 KM_COMS_Init()	15
4.1.2.3 KM_COMS_ProcessMsgs()	16
4.1.2.4 KM_COMS_ProcessPayload()	17
4.1.2.5 km_coms_ReceiveMsg()	19
4.1.2.6 KM_COMS_SendMsg()	20
4.1.3 Variable Documentation	21
4.1.3.1 km_coms_mutex	21
4.1.3.2 km_coms_uart	21
4.1.3.3 rx_buffer	22
4.1.3.4 rx_buffer_len	22
4.2 km_coms.c	22
4.3 km_coms.h File Reference	26
4.3.1 Macro Definition Documentation	27
4.3.1.1 BUF_SIZE_RX	27
4.3.1.2 BUF_SIZE_TX	27
4.3.1.3 KM_COMS_MSG_MAX_LEN	28
4.3.1.4 KM_COMS_RX_CHUNK	28
4.3.1.5 KM_COMS_SOM	28
4.3.2 Enumeration Type Documentation	28
4.3.2.1 message_type_t	28
4.3.3 Function Documentation	29
4.3.3.1 KM_COMS_Init()	29
4.3.3.2 KM_COMS_ProcessMsgs()	30
4.3.3.3 km_coms_ReceiveMsg()	31
4.3.3.4 KM_COMS_SendMsg()	32
4.4 km_coms.h	33
4.5 km_gamc.c File Reference	35
4.6 km_gamc.c	35
4.7 km_gamc.h File Reference	36
4.8 km_gamc.h	37
4.9 km_gpio.c File Reference	37

4.9.1 Function Documentation	39
4.9.1.1 KM_GPIO_I2CInit()	39
4.9.1.2 KM_GPIO_Init()	39
4.9.1.3 KM_GPIO_ReadADC()	40
4.9.1.4 KM_GPIO_ReadPin()	41
4.9.1.5 KM_GPIO_WriteDAC()	41
4.9.1.6 KM_GPIO_WritePinHigh()	42
4.9.1.7 KM_GPIO_WritePinLow()	42
4.9.1.8 KM_GPIO_WritePWM()	42
4.9.2 Variable Documentation	43
4.9.2.1 pin_cmd_acc	43
4.9.2.2 pin_cmd_brake	43
4.9.2.3 pin_hydraulic_1	44
4.9.2.4 pin_hydraulic_2	44
4.9.2.5 pin_i2c_scl	44
4.9.2.6 pin_i2c_sda	45
4.9.2.7 pin_motor_hall_1	45
4.9.2.8 pin_motor_hall_2	45
4.9.2.9 pin_motor_hall_3	46
4.9.2.10 pin_pedal_acc	46
4.9.2.11 pin_pedal_brake	46
4.9.2.12 pin_pressure_1	47
4.9.2.13 pin_pressure_2	47
4.9.2.14 pin_pressure_3	47
4.9.2.15 pin_status_led	48
4.9.2.16 pin_steer_dir	48
4.9.2.17 pin_steer_pwm	48
4.9.2.18 uart_config	49
4.10 km_gpio.c	49
4.11 km_gpio.h File Reference	52
4.11.1 Macro Definition Documentation	54
4.11.1.1 LEDC_HIGH_SPEED_MODE	54
4.11.1.2 LEDC_LOW_SPEED_MODE	55
4.11.1.3 PIN_CMD_ACC	55
4.11.1.4 PIN_CMD_BRAKE	55
4.11.1.5 PIN_HYDRAULIC_1	55
4.11.1.6 PIN_HYDRAULIC_2	55
4.11.1.7 PIN_I2C_SCL	55
4.11.1.8 PIN_I2C_SDA	55
4.11.1.9 PIN_MOTOR_HALL_1	55
4.11.1.10 PIN_MOTOR_HALL_2	56
4.11.1.11 PIN_MOTOR_HALL_3	56

4.11.1.12 PIN_ORIN_UART_RX	56
4.11.1.13 PIN_ORIN_UART_TX	56
4.11.1.14 PIN_PEDAL_ACC	56
4.11.1.15 PIN_PEDAL_BRAKE	56
4.11.1.16 PIN_PRESSURE_1	56
4.11.1.17 PIN_PRESSURE_2	56
4.11.1.18 PIN_PRESSURE_3	57
4.11.1.19 PIN_STATUS_LED	57
4.11.1.20 PIN_STEER_DIR	57
4.11.1.21 PIN_STEER_PWM	57
4.11.1.22 PIN_USB_UART_RX	57
4.11.1.23 PIN_USB_UART_TX	57
4.11.2 Function Documentation	58
4.11.2.1 KM_GPIO_I2CInit()	58
4.11.2.2 KM_GPIO_Init()	58
4.11.2.3 KM_GPIO_ReadADC()	59
4.11.2.4 KM_GPIO_ReadPin()	60
4.11.2.5 KM_GPIO_WriteDAC()	60
4.11.2.6 KM_GPIO_WritePinHigh()	61
4.11.2.7 KM_GPIO_WritePinLow()	61
4.11.2.8 KM_GPIO_WritePWM()	61
4.11.3 Variable Documentation	62
4.11.3.1 pin_cmd_acc	62
4.11.3.2 pin_cmd_brake	62
4.11.3.3 pin_hydraulic_1	62
4.11.3.4 pin_hydraulic_2	63
4.11.3.5 pin_i2c_scl	63
4.11.3.6 pin_i2c_sda	63
4.11.3.7 pin_motor_hall_1	63
4.11.3.8 pin_motor_hall_2	63
4.11.3.9 pin_motor_hall_3	64
4.11.3.10 pin_pedal_acc	64
4.11.3.11 pin_pedal_brake	64
4.11.3.12 pin_pressure_1	64
4.11.3.13 pin_pressure_2	64
4.11.3.14 pin_pressure_3	65
4.11.3.15 pin_status_led	65
4.11.3.16 pin_steer_dir	65
4.11.3.17 pin_steer_pwm	65
4.11.3.18 uart_config	65
4.12 km_gpio.h	66
4.13 km_objects.c File Reference	67

4.13.1 Function Documentation	68
4.13.1.1 KM_OBJ_GetObjectValue()	68
4.13.1.2 KM_OBJ_SetObjectValue()	69
4.13.2 Variable Documentation	69
4.13.2.1 km_objects_values	69
4.14 km_objects.c	69
4.15 km_objects.h File Reference	70
4.15.1 Macro Definition Documentation	71
4.15.1.1 OBJECT_VALUE_ERROR	71
4.15.2 Enumeration Type Documentation	71
4.15.2.1 km_objects_t	71
4.15.3 Function Documentation	72
4.15.3.1 KM_OBJ_GetObjectValue()	72
4.15.3.2 KM_OBJ_SetObjectValue()	72
4.16 km_objects.h	73
4.17 km_pid.c File Reference	74
4.17.1 Function Documentation	74
4.17.1.1 KM_PID_Calculate()	74
4.17.1.2 KM_PID_GetIntegral()	75
4.17.1.3 KM_PID_GetTunings()	75
4.17.1.4 KM_PID_Init()	75
4.17.1.5 KM_PID_Reset()	76
4.17.1.6 KM_PID_SetIntegralLimits()	76
4.17.1.7 KM_PID_SetOutputLimits()	76
4.17.1.8 KM_PID_SetTunings()	76
4.18 km_pid.c	77
4.19 km_pid.h File Reference	78
4.19.1 Function Documentation	79
4.19.1.1 KM_PID_Calculate()	79
4.19.1.2 KM_PID_GetIntegral()	80
4.19.1.3 KM_PID_GetTunings()	80
4.19.1.4 KM_PID_Init()	80
4.19.1.5 KM_PID_Reset()	81
4.19.1.6 KM_PID_SetIntegralLimits()	81
4.19.1.7 KM_PID_SetOutputLimits()	81
4.19.1.8 KM_PID_SetTunings()	81
4.20 km_pid.h	82
4.21 km_rtos.c File Reference	82
4.21.1 Detailed Description	84
4.21.2 Macro Definition Documentation	84
4.21.2.1 RTOS_DEFAULT_PERIOD_MS	84
4.21.2.2 RTOS_DEFAULT_PRIORITY	84

4.21.2.3 RTOS_DEFAULT_STACK_SIZE	84
4.21.3 Function Documentation	85
4.21.3.1 KM_COMS_CreateTask()	85
4.21.3.2 KM_RTOS_AddTask()	85
4.21.3.3 KM_RTOS_ChangePriority()	87
4.21.3.4 KM_RTOS_DeleteTask()	88
4.21.3.5 KM_RTOS_Destroy()	88
4.21.3.6 KM_RTOS_FindTask()	89
4.21.3.7 KM_RTOS_Init()	89
4.21.3.8 KM_RTOS_RestartTask()	90
4.21.3.9 KM_RTOS_ResumeTask()	91
4.21.3.10 KM_RTOS_SuspendTask()	91
4.21.3.11 KM_RTOS_TaskWrapper()	92
4.21.4 Variable Documentation	93
4.21.4.1 tasks	93
4.22 km_rtos.c	93
4.23 km_rtos.h File Reference	95
4.23.1 Macro Definition Documentation	97
4.23.1.1 RTOS_MAX_TASKS	97
4.23.2 Typedef Documentation	97
4.23.2.1 KM_RTOS_TaskFunction_t	97
4.23.3 Function Documentation	98
4.23.3.1 KM_COMS_CreateTask()	98
4.23.3.2 KM_RTOS_AddTask()	98
4.23.3.3 KM_RTOS_ChangePriority()	100
4.23.3.4 KM_RTOS_DeleteTask()	101
4.23.3.5 KM_RTOS_Destroy()	101
4.23.3.6 KM_RTOS_Init()	102
4.23.3.7 KM_RTOS_RestartTask()	102
4.23.3.8 KM_RTOS_ResumeTask()	103
4.23.3.9 KM_RTOS_SuspendTask()	104
4.24 km_rtos.h	104
4.25 km_sdir.c File Reference	106
4.25.1 Macro Definition Documentation	106
4.25.1.1 I2C_MASTER_TIMEOUT_MS	106
4.25.1.2 TAG	107
4.25.2 Function Documentation	107
4.25.2.1 KM_SDIR_Begin()	107
4.25.2.2 KM_SDIR_Init()	107
4.25.2.3 KM_SDIR_isConnected()	108
4.25.2.4 KM_SDIR_ReadAngle()	108
4.25.2.5 KM_SDIR_ReadAngleDegrees()	108

4.25.2.6 KM_SDIR_ReadAngleRadians()	108
4.25.2.7 KM_SDIR_ReadRaw()	108
4.25.2.8 KM_SDIR_ReadRegisters()	109
4.25.2.9 KM_SDIR_ResetI2C()	109
4.25.2.10 KM_SDIR_setCenterOffset()	110
4.26 km_sdir.c	110
4.27 km_sdir.h File Reference	112
4.27.1 Macro Definition Documentation	113
4.27.1.1 PI	113
4.27.2 Enumeration Type Documentation	113
4.27.2.1 conection_consts	113
4.27.2.2 sensor_consts	114
4.27.3 Function Documentation	114
4.27.3.1 KM_SDIR_Begin()	114
4.27.3.2 KM_SDIR_Init()	115
4.27.3.3 KM_SDIR_isConnected()	115
4.27.3.4 KM_SDIR_ReadAngleDegrees()	115
4.27.3.5 KM_SDIR_ReadAngleRadians()	115
4.27.3.6 KM_SDIR_ReadRaw()	116
4.27.3.7 KM_SDIR_ResetI2C()	116
4.27.3.8 KM_SDIR_setCenterOffset()	116
4.27.4 Variable Documentation	117
4.27.4.1 MAX_RAD	117
4.28 km_sdir.h	117
4.29 km_sta.c File Reference	118
4.30 km_sta.c	118
4.31 km_sta.h File Reference	119
4.32 km_sta.h	119
4.33 main.c File Reference	120
4.33.1 Macro Definition Documentation	121
4.33.1.1 MAX_ERROR_COUNT_SDIR	121
4.33.2 Function Documentation	121
4.33.2.1 app_main()	121
4.33.2.2 system_init()	122
4.33.3 Variable Documentation	122
4.33.3.1 TAG	122
4.34 main.c	122

Chapter 1

Class Index

1.1 Class List

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

km_coms_msg	Structure representing a KM_COMS message	5
PID_Controller	Structure that reperesents a PID	6
RTOS_Task	Structure that reperesents a task in FreeRTOS	9
sensor_struct	Structure that reperesents the direction sensor	10

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

km_coms.c	13
km_coms.h	26
km_gamc.c	35
km_gamc.h	36
km_gpio.c	37
km_gpio.h	52
km_objects.c	67
km_objects.h	70
km_pid.c	74
km_pid.h	78
km_rtos.c	
Implementation of the KM_RTOS task management library	82
km_rtos.h	95
km_sdir.c	106
km_sdir.h	112
km_sta.c	118
km_sta.h	119
main.c	120

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 represents 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 represents 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 represents 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 represents 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 represents 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 represents 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](#)

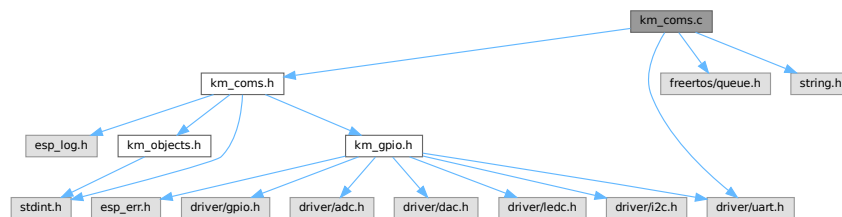
Chapter 4

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

<i>len</i>	Length of the payload data in bytes.
<i>type</i>	Message type identifier.
<i>data</i>	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; }
```

```

00360
00361     crc ^= len;
00362     for (int j = 0; j < 8; j++) {
00363         crc = (crc & 0x80) ? (crc < 1) ^ 0x07 : (crc < 1);
00364     }
00365
00366     crc ^= type;
00367     for (int j = 0; j < 8; j++) {
00368         crc = (crc & 0x80) ? (crc < 1) ^ 0x07 : (crc < 1);
00369     }
00370
00371     for (uint8_t i = 0; i < len; i++) {
00372         crc ^= data[i];
00373         for (int j = 0; j < 8; j++) {
00374             crc = (crc & 0x80) ? (crc < 1) ^ 0x07 : (crc < 1);
00375         }
00376     }
00377
00378     return crc;
00379 }

```

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

<code>uart_port</code>	The UART port to use (e.g., <code>UART_NUM_1</code> , <code>UART_NUM_2</code>).
------------------------	--

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](#).

```

00080                                     {
00081     km_coms_uart = uart_port;
00082
00083     km_coms_mutex = xSemaphoreCreateMutex();
00084     if(km_coms_mutex == NULL)
00085         return ESP_ERR_NO_MEM;
00086
00087     // Instala el driver UART con buffers TX/RX
00088     uart_driver_install(km_coms_uart, BUF_SIZE_RX, BUF_SIZE_TX, 0, NULL, 0);
00089     uart_param_config(km_coms_uart, &uart_config);
00090
00091     // Asigna pines según el puerto
00092     if(km_coms_uart == UART_NUM_2) {
00093         uart_set_pin(km_coms_uart, PIN_ORIN_UART_TX, PIN_ORIN_UART_RX,
00094                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
00095     } else {
00096         uart_set_pin(km_coms_uart, PIN_USB_UART_TX, PIN_USB_UART_RX,
00097                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
00098     }
00099
00100     return ESP_OK;
00101 }
```

4.1.2.3 KM_COMS_ProcessMsgs()

```

void KM_COMS_ProcessMsgs (
    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_ProcessPayload()` 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`.

```

00181         {
00182             size_t processed = 0;
00183
00184             if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) != pdTRUE)
00185                 return;
00186
00187             while(rx_buffer_len - processed >= 4) { // Mínimo: SOM + LEN + TYPE + CRC
00188                 if(rx_buffer[processed] != KM_COMS_SOM) {
00189                     processed++;
00190                     continue;
00191                 }
00192
00193                 uint8_t payload_len = rx_buffer[processed + 1];
00194                 size_t total_len = 4 + payload_len; // SOM + LEN + TYPE + PAYLOAD + CRC
00195
00196                 if(rx_buffer_len - processed < total_len)
00197                     break; // Mensaje incompleto, esperar más bytes
00198
00199                 // Construir mensaje
00200                 km_coms_msg msg;
00201                 msg.len = payload_len;
00202                 msg.type = rx_buffer[processed + 2];
00203                 memcpy(msg.payload, rx_buffer + processed + 3, payload_len);
00204                 msg.crc = rx_buffer[processed + 3 + payload_len];
00205
00206                 // Verificar CRC
00207                 uint8_t crc_calc = KM_COMS_crc8(msg.len, msg.type, msg.payload);
00208                 if(crc_calc == msg.crc) {
00209                     KM_COMS_ProcessPayload(msg);
00210                 }
00211
00212                 processed += total_len;
00213             }
00214
00215             // Compactar buffer: mover bytes no procesados al inicio
00216             if(processed > 0 && processed < rx_buffer_len) {
00217                 memmove(rx_buffer, rx_buffer + processed, rx_buffer_len - processed);
00218                 rx_buffer_len -= processed;
00219             } else if(processed >= rx_buffer_len) {
00220                 rx_buffer_len = 0;
00221             }
00222
00223             xSemaphoreGive(km_coms_mutex);
00224 }

```

4.1.2.4 KM_COMS_ProcessPayload()

```

static void KM_COMS_ProcessPayload (
    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

<i>msg</i>	The incoming message to process.
------------	----------------------------------

Definition at line 251 of file `km_coms.c`.

```

00251                                     {
00252     ESP_LOGI("KM_coms", "RX msg type=0x%02X len=%d crc=0x%02X", msg.type, msg.len, msg.crc);
00253
00254     int64_t object_value = 0;
00255     int8_t direction;
00256
00257     switch (msg.type)
00258     {
00259     case ORIN_TARG_THROTTLE:
00260         if (msg.len != 1) return; // Invalid payload
00261
00262         object_value = (int64_t)msg.payload[0];
00263         KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);
00264         break;
00265
00266     case ORIN_TARG BRAKING:
00267         if (msg.len != 1) return; // Invalid payload
00268         object_value = (int64_t)msg.payload[0];
00269         KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);
00270         break;
00271
00272     case ORIN_TARG STEERING:
00273         if (msg.len != 2) return; // Invalid payload
00274         direction = msg.payload[0];
00275         if (direction == 0) {
00276             object_value = (int64_t)msg.payload[1];
00277         } else if (direction == 1) {
00278             object_value -= (int64_t)msg.payload[1];
00279         } else {
00280             return; // Invalid payload
00281         }
00282
00283         KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);
00284         break;
00285
00286     case ORIN_MISION:
00287         if (msg.len != 1) return; // Invalid payload
00288         object_value = (int64_t)msg.payload[0];
00289         KM_OBJ_SetObjectValue(MISION_ORIN, object_value);
00290         break;
00291
00292     case ORIN_MACHINE_STATE:
00293         if (msg.len != 1) return; // Invalid payload
00294         object_value = (int64_t)msg.payload[0];
00295         KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);
00296         break;
00297
00298     case ORIN_HEARTBEAT:
00299         // No si guardarlo en la libreria de variables o reinicar algo. ns
00300         break;
00301
00302     case ORIN_SHUTDOWN:
00303         if (msg.len != 1) return; // Invalid payload
00304         object_value = (int64_t)msg.payload[0];
00305         KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
00306         break;
00307
00308     case ORIN_COMPLETE:
00309         if (msg.len != 7) return; // Invalid payload
00310         object_value = (int64_t)msg.payload[0];
00311         KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);

```

```

00312
00313     object_value = (int64_t)msg.payload[1];
00314     KM_OBJ_SetObjectValue(TARGET BRAKING, object_value);
00315
00316     direction = msg.payload[2];
00317     if (direction == 0) {
00318         object_value = (int64_t)msg.payload[3];
00319     } else if (direction == 1){
00320         object_value -= (int64_t)msg.payload[3];
00321     } else {
00322         return; // Invalid payload
00323     }
00324     KM_OBJ_SetObjectValue(TARGET STEERING, object_value);
00325
00326     object_value = (int64_t)msg.payload[4];
00327     KM_OBJ_SetObjectValue(MISION_ORIN, object_value);
00328
00329     object_value = (int64_t)msg.payload[5];
00330     KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);
00331
00332     object_value = (int64_t)msg.payload[6];
00333     KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
00334     break;
00335
00336     default:
00337         break;
00338 }
00339 }

```

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`.

```

00148     {
00149         uint8_t uart_chunk[KM_COMS_RX_CHUNK];
00150         size_t len_read = 0;
00151         uint8_t bytes2read = 0;
00152
00153         // 1. Verificar cuantos bytes hay en el buffer UART
00154         size_t uart_len = 0;
00155         uart_get_buffered_data_len(km_coms_uart, &uart_len);
00156         if(uart_len == 0)
00157             return;
00158
00159         // 2. Leer hasta KM_COMS_RX_CHUNK bytes de la UART
00160         if(uart_len > KM_COMS_RX_CHUNK)
00161             bytes2read = KM_COMS_RX_CHUNK;
00162         else
00163             bytes2read = uart_len;
00164
00165         len_read = uart_read_bytes(km_coms_uart, uart_chunk, bytes2read, 0);
00166         if(len_read == 0)
00167             return;
00168
00169         if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) == pdTRUE) {
00170             // 3. Copiar bytes al buffer interno
00171             if(rx_buffer_len + len_read > sizeof(rx_buffer)) {
00172                 // Overflow, reiniciar buffer
00173                 rx_buffer_len = 0;
00174             }
00175             memcpy(rx_buffer + rx_buffer_len, uart_chunk, len_read);
00176             rx_buffer_len += len_read;
00177             xSemaphoreGive(km_coms_mutex);
00178         }
00179     }

```

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

<i>type</i>	The type of message to send (<code>message_type_t</code>).
<i>payload</i>	Pointer to the payload data to send.
<i>len</i>	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:
00140         %d", total_sent, 4+len);
00141         return 0;
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.](#)

```
00001 /*****
00002  * @file      km_coms.c
00003  * @brief     Implementation of the KM_COMS communication library for ESP32.
00004  *
00005  * This file implements the KM_COMS library, providing UART-based communication
00006  * between an ESP32 and an NVIDIA Orin (or other host system). The library handles:
00007  * - Initialization and configuration of the UART peripheral
00008  * - Sending and receiving framed messages
00009  * - CRC8 validation for data integrity
00010  * - Processing incoming payloads and updating shared application objects
00011  * - FreeRTOS task integration for periodic message handling
00012  *
00013  * -----
00014  * PROTOCOL DETAILS
00015  * -----
00016  * Each message follows a custom framing format:
00017  *
00018  * | SOF | LEN | TYPE | PAYLOAD | CRC |
00019  *
00020  * Fields:
00021  * - SOF      : Start-of-frame byte (0xAA)
00022  * - LEN      : 1-byte payload length
00023  * - TYPE     : 1-byte message type (see message_type_t in km_coms.h)
00024  * - PAYLOAD  : N bytes of data up to 251 bytes, content depends on message type
00025  * - CRC      : 1-byte CRC8 (XOR-based) over LEN, TYPE, and PAYLOAD
00026  *
00027  * Total size of message is 255 bytes
00028  *
00029  * The `message_type_t` enum defines all possible message types. Conceptually:
00030  * - ESP32 → Orin messages contain telemetry data such as current speed, throttle,
00031  *   braking, steering, mission state, machine state, and shutdown status.
00032  * - Orin → ESP32 messages carry commands such as target throttle, braking,
00033  *   steering, mission state, machine state, and heartbeat.
00034  *
00035  * Using the enum ensures type safety and consistency when sending or processing messages.
00036  *
00037  * The CRC ensures the integrity of the message. Invalid messages are discarded.
00038  *
00039  * The library uses internal buffers and a FreeRTOS mutex to provide thread-safe
00040  * access to shared objects and message queues.
00041  *
00042  * -----
00043  * FREE RTOS INTEGRATION
00044  * -----
00045  * Periodic tasks should call km_coms_ReceiveMsg() and KM_COMS_ProcessMsgs() to
00046  * read bytes from UART, assemble messages, validate CRC, and process payloads.
00047  * KM_COMS_SendMsg() can be used to transmit messages safely from tasks.
00048  *
00049  * Author: Adrian Navarredonda Arizaleta
00050  * Date:   14-02-2026
00051  * Version: 1.0
00052  *****/
00053
00054 #include "km_coms.h"
00055
00056 /***** INCLUDES INTERNOS *****/
00057 // Headers internos opcionales, dependencias privadas
00058 #include "driver/uart.h"
```

```

00059 #include "freertos/queue.h"
00060 #include <string.h>
00061
00062 /***** MACROS PRIVADAS *****/
00063 // Constantes internas, flags de debug
00064
00065 #define KM_COMS_WAIT_SEM_AVAILABLE 5
00066
00067 /***** VARIABLES PRIVADAS *****/
00068 // Variables globales internas (static)
00069 static uint8_t rx_buffer[KM_COMS_MSG_MAX_LEN - 1]; //[0-255]
00070 static size_t rx_buffer_len = 0;
00071 static SemaphoreHandle_t km_coms_mutex;
00072 static uart_port_t km_coms_uart = UART_NUM_0;
00073
00074 /***** DECLARACION FUNCIONES PRIVADAS *****/
00075 static void KM_COMS_ProcessPayload(km_coms_msg msg);
00076 static uint8_t KM_COMS_crc8(uint8_t len, uint8_t type, const uint8_t *data);
00077
00078 /***** FUNCIONES PÚBLICAS *****/
00079
00080 esp_err_t KM_COMS_Init(uart_port_t uart_port) {
00081     km_coms_uart = uart_port;
00082
00083     km_coms_mutex = xSemaphoreCreateMutex();
00084     if(km_coms_mutex == NULL)
00085         return ESP_ERR_NO_MEM;
00086
00087     // Instala el driver UART con buffers TX/RX
00088     uart_driver_install(km_coms_uart, BUF_SIZE_RX, BUF_SIZE_TX, 0, NULL, 0);
00089     uart_param_config(km_coms_uart, &uart_config);
00090
00091     // Asigna pines según el puerto
00092     if(km_coms_uart == UART_NUM_2) {
00093         uart_set_pin(km_coms_uart, PIN_ORIN_UART_TX, PIN_ORIN_UART_RX,
00094                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
00095     } else {
00096         uart_set_pin(km_coms_uart, PIN_USB_UART_TX, PIN_USB_UART_RX,
00097                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
00098     }
00099
00100     return ESP_OK;
00101 }
00102
00103 int KM_COMS_SendMsg(message_type_t type, uint8_t *payload, uint8_t len) {
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 }
00146
00147
00148 void km_coms_ReceiveMsg(void) {
00149     uint8_t uart_chunk[KM_COMS_RX_CHUNK];
00150     size_t len_read = 0;
00151     uint8_t bytes2read = 0;
00152
00153     // 1. Verificar cuantos bytes hay en el buffer UART
00154     size_t uart_len = 0;
00155     uart_get_buffered_data_len(km_coms_uart, &uart_len);
00156     if(uart_len == 0)
00157         return;
00158
00159     // 2. Leer hasta KM_COMS_RX_CHUNK bytes de la UART
00160     if(uart_len > KM_COMS_RX_CHUNK)
00161         bytes2read = KM_COMS_RX_CHUNK;
00162     else
00163         bytes2read = uart_len;
00164
00165     len_read = uart_read_bytes(km_coms_uart, uart_chunk, bytes2read, 0);
00166     if(len_read == 0)
00167         return;
00168
00169     if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) == pdTRUE) {
00170         // 3. Copiar bytes al buffer interno
00171         if(rx_buffer_len + len_read > sizeof(rx_buffer)) {
00172             // Overflow, reiniciar buffer
00173             rx_buffer_len = 0;
00174         }
00175         memcpy(rx_buffer + rx_buffer_len, uart_chunk, len_read);
00176         rx_buffer_len += len_read;
00177         xSemaphoreGive(km_coms_mutex);
00178     }
00179 }
00180
00181 void KM_COMS_ProccessMsgs(void) {
00182     size_t processed = 0;
00183
00184     if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) != pdTRUE)
00185         return;
00186
00187     while(rx_buffer_len - processed >= 4) { // Mínimo: SOM + LEN + TYPE + CRC
00188         if(rx_buffer[processed] != KM_COMS_SOM) {
00189             processed++;
00190             continue;
00191         }
00192
00193         uint8_t payload_len = rx_buffer[processed + 1];
00194         size_t total_len = 4 + payload_len; // SOM + LEN + TYPE + PAYLOAD + CRC
00195
00196         if(rx_buffer_len - processed < total_len)
00197             break; // Mensaje incompleto, esperar más bytes
00198
00199         // Construir mensaje
00200         km_coms_msg msg;
00201         msg.len = payload_len;
00202         msg.type = rx_buffer[processed + 2];
00203         memcpy(msg.payload, rx_buffer + processed + 3, payload_len);
00204         msg.crc = rx_buffer[processed + 3 + payload_len];
00205
00206         // Verificar CRC
00207         uint8_t crc_calc = KM_COMS_crc8(msg.len, msg.type, msg.payload);
00208         if(crc_calc == msg.crc) {
00209             KM_COMS_ProccessPayload(msg);
00210         }
00211
00212         processed += total_len;
00213     }
00214
00215     // Compactar buffer: mover bytes no procesados al inicio
00216     if(processed > 0 && processed < rx_buffer_len) {
00217         memmove(rx_buffer, rx_buffer + processed, rx_buffer_len - processed);
00218         rx_buffer_len -= processed;
00219     } else if(processed >= rx_buffer_len) {
00220         rx_buffer_len = 0;
00221     }
00222
00223     xSemaphoreGive(km_coms_mutex);
00224 }
00225
00226 /***** FUNCIONES PRIVADAS *****/
00227
00228 static void KM_COMS_ProccessPayload(km_coms_msg msg) {
00229     ESP_LOGI("KM_coms", "RX msg type=0x%02X len=%d crc=0x%02X", msg.type, msg.len, msg.crc);
00230
00231     int64_t object_value = 0;

```

```

00255     int8_t direction;
00256
00257     switch (msg.type)
00258     {
00259     case ORIN_TARG_THROTTLE:
00260         if (msg.len != 1) return; // Invalid payload
00261
00262         object_value = (int64_t)msg.payload[0];
00263         KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);
00264         break;
00265
00266     case ORIN_TARG BRAKING:
00267         if (msg.len != 1) return; // Invalid payload
00268         object_value = (int64_t)msg.payload[0];
00269         KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);
00270         break;
00271
00272     case ORIN_TARG STEERING:
00273         if (msg.len != 2) return; // Invalid payload
00274         direction = msg.payload[0];
00275         if (direction == 0) {
00276             object_value = (int64_t)msg.payload[1];
00277         } else if (direction == 1) {
00278             object_value -= (int64_t)msg.payload[1];
00279         } else {
00280             return; // Invalid payload
00281         }
00282
00283         KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);
00284         break;
00285
00286     case ORIN_MISION:
00287         if (msg.len != 1) return; // Invalid payload
00288         object_value = (int64_t)msg.payload[0];
00289         KM_OBJ_SetObjectValue(MISION_ORIN, object_value);
00290         break;
00291
00292     case ORIN_MACHINE_STATE:
00293         if (msg.len != 1) return; // Invalid payload
00294         object_value = (int64_t)msg.payload[0];
00295         KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);
00296         break;
00297
00298     case ORIN_HEARTBEAT:
00299         // No se si guardarlo en la libreria de variables o reiniciar algo. ns
00300         break;
00301
00302     case ORIN_SHUTDOWN:
00303         if (msg.len != 1) return; // Invalid payload
00304         object_value = (int64_t)msg.payload[0];
00305         KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
00306         break;
00307
00308     case ORIN_COMPLETE:
00309         if (msg.len != 7) return; // Invalid payload
00310         object_value = (int64_t)msg.payload[0];
00311         KM_OBJ_SetObjectValue(TARGET_THROTTLE, object_value);
00312
00313         object_value = (int64_t)msg.payload[1];
00314         KM_OBJ_SetObjectValue(TARGET_BRAKING, object_value);
00315
00316         direction = msg.payload[2];
00317         if (direction == 0) {
00318             object_value = (int64_t)msg.payload[3];
00319         } else if (direction == 1) {
00320             object_value -= (int64_t)msg.payload[3];
00321         } else {
00322             return; // Invalid payload
00323         }
00324         KM_OBJ_SetObjectValue(TARGET_STEERING, object_value);
00325
00326         object_value = (int64_t)msg.payload[4];
00327         KM_OBJ_SetObjectValue(MISION_ORIN, object_value);
00328
00329         object_value = (int64_t)msg.payload[5];
00330         KM_OBJ_SetObjectValue(MACHINE_STATE_ORIN, object_value);
00331
00332         object_value = (int64_t)msg.payload[6];
00333         KM_OBJ_SetObjectValue(SHUTDOWN_ORIN, object_value);
00334         break;
00335
00336     default:
00337         break;
00338     }
00339 }
00340
00358 static uint8_t KM_COMS_crc8(uint8_t len, uint8_t type, const uint8_t *data) {

```

```

00359     uint8_t crc = 0x00;
00360
00361     crc ^= len;
00362     for (int j = 0; j < 8; j++) {
00363         crc = (crc & 0x80) ? (crc << 1) ^ 0x07 : (crc << 1);
00364     }
00365
00366     crc ^= type;
00367     for (int j = 0; j < 8; j++) {
00368         crc = (crc & 0x80) ? (crc << 1) ^ 0x07 : (crc << 1);
00369     }
00370
00371     for (uint8_t i = 0; i < len; i++) {
00372         crc ^= data[i];
00373         for (int j = 0; j < 8; j++) {
00374             crc = (crc & 0x80) ? (crc << 1) ^ 0x07 : (crc << 1);
00375         }
00376     }
00377
00378     return crc;
00379 }
00380
00381 /***** FIN DE ARCHIVO *****/

```

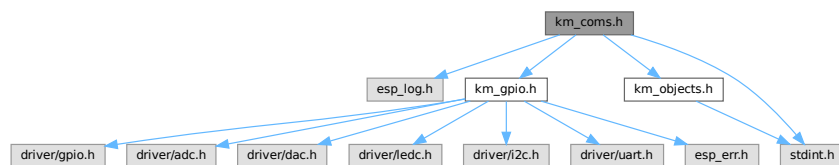
4.3 km_coms.h File Reference

```

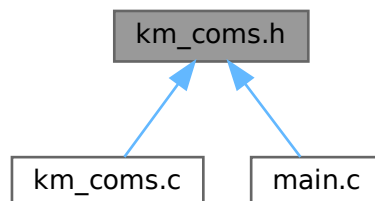
#include "esp_log.h"
#include "km_gpio.h"
#include "km_objects.h"
#include <stdint.h>

```

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_MISSION = 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_MISSION = 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_ProcessMsgs (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](#).

```

00086 {
00087     // =====
00088     // ESP32 --> Orin (0x01 - 0x1F)
00089     // =====
00090     ESP_ACT_SPEED           = 0x01,
00091     ESP_ACT_THROTTLE       = 0x02,
00092     ESP_ACT BRAKING        = 0x03,
00093     ESP_ACT STEERING       = 0x04,
00094     ESP_MISION             = 0x05,
00095     ESP_MACHINE_STATE      = 0x06,
00096     ESP_ACT_SHUTDOWN       = 0x07,
00097     ESP_HEARTBEAT          = 0x08,
00098     ESP_COMPLETE           = 0x09,
00100     // =====
00101     // Orin --> ESP32 (0x20 - 0x3F)
00102     // =====
00103     ORIN_TARG_THROTTLE     = 0x20,
00104     ORIN_TARG BRAKING      = 0x21,
00105     ORIN_TARG STEERING     = 0x22,
00106     ORIN_MISION            = 0x23,
00107     ORIN_MACHINE_STATE     = 0x24,
00108     ORIN_HEARTBEAT         = 0x25,
00109     ORIN_SHUTDOWN          = 0x26,
00110     ORIN_COMPLETE         = 0x27,
00112     // =====
00113     // Others (0x40 - 0xFF)
00114     // =====
00115 } 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

<code>uart_port</code>	The UART port to use (e.g., <code>UART_NUM_1</code> , <code>UART_NUM_2</code>).
------------------------	--

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`.

```

00080                                     {
00081     km_coms_uart = uart_port;
00082
00083     km_coms_mutex = xSemaphoreCreateMutex();
00084     if(km_coms_mutex == NULL)
00085         return ESP_ERR_NO_MEM;
00086
00087     // Instala el driver UART con buffers TX/RX
00088     uart_driver_install(km_coms_uart, BUF_SIZE_RX, BUF_SIZE_TX, 0, NULL, 0);
00089     uart_param_config(km_coms_uart, &uart_config);
00090
00091     // Asigna pines según el puerto
00092     if(km_coms_uart == UART_NUM_2) {
00093         uart_set_pin(km_coms_uart, PIN_ORIN_UART_TX, PIN_ORIN_UART_RX,
00094                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
00095     } else {
00096         uart_set_pin(km_coms_uart, PIN_USB_UART_TX, PIN_USB_UART_RX,
00097                     UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
00098     }
00099
00100     return ESP_OK;
00101 }
```

4.3.3.2 KM_COMS_ProcessMsgs()

```

void KM_COMS_ProcessMsgs (
    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_ProcessPayload()` 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`.

```

00181     {
00182         size_t processed = 0;
00183
00184         if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) != pdTRUE)
00185             return;
00186
00187         while(rx_buffer_len - processed >= 4) { // Mínimo: SOM + LEN + TYPE + CRC
00188             if(rx_buffer[processed] != KM_COMS_SOM) {
00189                 processed++;
00190                 continue;
00191             }
00192
00193             uint8_t payload_len = rx_buffer[processed + 1];
00194             size_t total_len = 4 + payload_len; // SOM + LEN + TYPE + PAYLOAD + CRC
00195
00196             if(rx_buffer_len - processed < total_len)
00197                 break; // Mensaje incompleto, esperar más bytes
00198
00199             // Construir mensaje
00200             km_coms_msg msg;
00201             msg.len = payload_len;
00202             msg.type = rx_buffer[processed + 2];
00203             memcpy(msg.payload, rx_buffer + processed + 3, payload_len);
00204             msg.crc = rx_buffer[processed + 3 + payload_len];
00205
00206             // Verificar CRC
00207             uint8_t crc_calc = KM_COMS_crc8(msg.len, msg.type, msg.payload);
00208             if(crc_calc == msg.crc) {
00209                 KM_COMS_ProcessPayload(msg);
00210             }
00211
00212             processed += total_len;
00213         }
00214
00215         // Compactar buffer: mover bytes no procesados al inicio
00216         if(processed > 0 && processed < rx_buffer_len) {
00217             memmove(rx_buffer, rx_buffer + processed, rx_buffer_len - processed);
00218             rx_buffer_len -= processed;
00219         } else if(processed >= rx_buffer_len) {
00220             rx_buffer_len = 0;
00221         }
00222
00223         xSemaphoreGive(km_coms_mutex);
00224     }

```

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`.

```

00148     {
00149         uint8_t uart_chunk[KM_COMS_RX_CHUNK];
00150         size_t len_read = 0;
00151         uint8_t bytes2read = 0;
00152
00153         // 1. Verificar cuantos bytes hay en el buffer UART
00154         size_t uart_len = 0;
00155         uart_get_buffered_data_len(km_coms_uart, &uart_len);
00156         if(uart_len == 0)
00157             return;
00158
00159         // 2. Leer hasta KM_COMS_RX_CHUNK bytes de la UART
00160         if(uart_len > KM_COMS_RX_CHUNK)
00161             bytes2read = KM_COMS_RX_CHUNK;
00162         else
00163             bytes2read = uart_len;
00164
00165         len_read = uart_read_bytes(km_coms_uart, uart_chunk, bytes2read, 0);
00166         if(len_read == 0)
00167             return;
00168
00169         if(xSemaphoreTake(km_coms_mutex, pdMS_TO_TICKS(KM_COMS_WAIT_SEM_AVAILABLE)) == pdTRUE) {
00170             // 3. Copiar bytes al buffer interno
00171             if(rx_buffer_len + len_read > sizeof(rx_buffer)) {
00172                 // Overflow, reiniciar buffer
00173                 rx_buffer_len = 0;
00174             }
00175             memcpy(rx_buffer + rx_buffer_len, uart_chunk, len_read);
00176             rx_buffer_len += len_read;
00177             xSemaphoreGive(km_coms_mutex);
00178         }
00179     }

```

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

<i>type</i>	The type of message to send (message_type_t).
<i>payload</i>	Pointer to the payload data to send.
<i>len</i>	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:
00140 %d", total_sent, 4+len);
00141         return 0;
00142     }
00143
00144     // Se ha enviado el mensaje correctamente
00145     return 1;
00146 }

```

4.4 km_coms.h

[Go to the documentation of this file.](#)

```

00001 /*****
00002  * @file    km_coms.h
00003  * @brief   Public API for the KM_COMS communication library.
00004  *
00005  * This header defines the interface for the KM_COMS library, which provides:
00006  * - UART-based communication between an ESP32 and NVIDIA Orin (or other host)

```

```

00007  * - Message framing, CRC checks, and payload handling
00008  * - Functions to send, receive, and process messages
00009  *
00010  * The library supports periodic tasks that handle incoming/outgoing messages
00011  * safely in a FreeRTOS environment.
00012  *
00013  * @author  Adrian Navarredonda Arizaleta
00014  * @date    14-02-2026
00015  * @version 1.0
00016  *****/
00017
00018 #ifndef KM_COMS_H
00019 #define KM_COMS_H
00020
00021 /***** INCLUDES *****/
00022 // Includes necesarios para la API pública
00023 #include "esp_log.h" // Para log
00024 #include "km_gpio.h"
00025 #include "km_objects.h"
00026 #include <stdint.h>
00027
00028 // Estructura mensaje, | es solo para visualizar los disintintos campos, no esta
00029 // en el mensaje
00030 // | SOF | LEN | TYPE | PAYLOAD | CRC |
00031
00032 // Donde:
00033
00034 //     SOF → 1 byte (ej: 0xAA)
00035
00036 //     LEN → 1 byte (longitud del payload)
00037
00038 //     TYPE → 1 byte (tipo de mensaje)
00039
00040 //     PAYLOAD → N bytes
00041
00042 //     CRC → 1 byte (checksum simple XOR)
00043
00044 // ----- Tipos de mensajes -----
00045 // ## Orin ↔ ESP32 Messaging (UTF-8)
00046
00047 // All inter-computer messages are UTF-8 text. Payloads are defined below; line framing and exact
00048 // field ordering should follow the agreed message definitions when implemented.
00049
00050 // **ESP32 -> Orin (telemetry):**
00051 // - `actual_speed` (m/s, float)
00052 // - `actual_acc` (m/s^2, float, signed)
00053 // - `actual_braking` (0-1, float; interpreted as brake pedal or hydraulic pressure)
00054 // - `actual_steering` (rad, float)
00055 // - `mission` (enum; state machine owner TBD)
00056 // - `machine_state` (enum; mission sub-state)
00057 // - `actual_shutdown` (0/1, end of SDC loop state)
00058 // - `esp32_heartbeat`
00059
00060 // **Orin -> ESP32 (commands):**
00061 // - `target_throttle` (0-1, float)
00062 // - `target_braking` (0-1, float)
00063 // - `target_steering` (-1 to 1, float)
00064 // - `mission` (enum; state machine owner TBD)
00065 // - `machine_state` (enum; mission sub-state)
00066 // - `orin_heartbeat`
00067
00068 /***** DEFINES PÚBLICAS *****/
00069 // Constantes, flags o configuraciones visibles desde fuera de la librería
00070
00071 #define KM_COMS_SOM 0xAA
00072 #define KM_COMS_MSG_MAX_LEN 256
00073 #define KM_COMS_RX_CHUNK 64 // Lectura de la UART por bloques
00074 #define BUF_SIZE_TX 2048
00075 #define BUF_SIZE_RX 2048
00076
00077 /***** TIPOS PÚBLICOS *****/
00078 // Estructuras, enums, typedefs públicos
00079
00085 typedef enum
00086 {
00087     // =====
00088     // ESP32 --> Orin (0x01 - 0x0F)
00089     // =====
00090     ESP_ACT_SPEED           = 0x01,
00091     ESP_ACT_THROTTLE        = 0x02,
00092     ESP_ACT_BRAKING         = 0x03,
00093     ESP_ACT_STEERING        = 0x04,
00094     ESP_MISION              = 0x05,
00095     ESP_MACHINE_STATE       = 0x06,
00096     ESP_ACT_SHUTDOWN        = 0x07,
00097     ESP_HEARTBEAT           = 0x08,
00098     ESP_COMPLETE            = 0x09,

```

```

00100 // =====
00101 // Orin --> ESP32 (0x20 - 0x3F)
00102 // =====
00103 ORIN_TARG_THROTTLE = 0x20,
00104 ORIN_TARG BRAKING = 0x21,
00105 ORIN_TARG STEERING = 0x22,
00106 ORIN_MISION = 0x23,
00107 ORIN_MACHINE_STATE = 0x24,
00108 ORIN_HEARTBEAT = 0x25,
00109 ORIN_SHUTDOWN = 0x26,
00110 ORIN_COMPLETE = 0x27,
00112 // =====
00113 // Others (0x40 - 0xFF)
00114 // =====
00115 } message_type_t;
00116
00129 typedef struct {
00130     uint8_t len;
00131     uint8_t type;
00132     uint8_t payload[KM_COMS_MSG_MAX_LEN-5];
00133     uint8_t crc;
00134 } km_coms_msg;
00135
00136 /***** VARIABLES PÚBLICAS *****/
00137 // Variables globales visibles (si realmente se necesitan)
00138
00139 // extern int ejemplo_variable_publica;
00140
00141 /***** FUNCIONES PÚBLICAS *****/
00165 esp_err_t KM_COMS_Init(uart_port_t uart_port);
00166
00189 int KM_COMS_SendMsg(message_type_t type, uint8_t *payload, uint8_t len);
00190
00211 void km_coms_ReceiveMsg(void);
00212
00239 void KM_COMS_ProcessMsgs(void);
00240
00241 #endif /* KM_COMS_H */

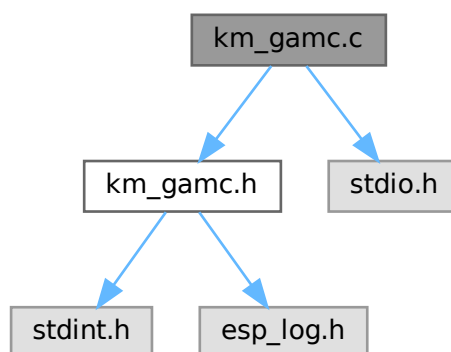
```

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.](#)

```

00001 /*****
00002  * @file    nombre_libreria.c
00003  * @brief   Implementación de la librería.
00004  *****/
00005
00006 #include "km_gamc.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 // #define LIBRERIA_DEBUG 1
00015
00016 /***** VARIABLES PRIVADAS *****/
00017 // Variables globales internas (static)
00018
00019 /***** DECLARACION FUNCIONES PRIVADAS *****/
00020
00021
00022 /***** FUNCIONES PÚBLICAS *****/
00026 // int libreria_operacion(int valor) {
00027 //     ...
00028 // }
00029
00030 /***** FUNCIONES PRIVADAS *****/
00034 // static void funcion_privada(void);
00035
00036 /***** FIN DE ARCHIVO *****/
00037

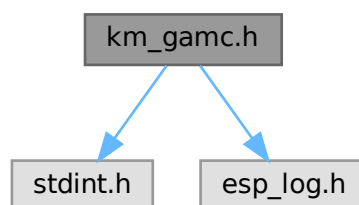
```

4.7 km_gamc.h File Reference

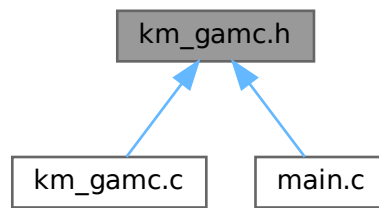
```
#include <stdint.h>
```

```
#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.](#)

```

00001 /*****
00002  * @file    nombre_libreria.h
00003  * @brief   Interfaz pública de la librería.
00004  * @author   Autor
00005  * @date    DD-MM-AAAA
00006  * @version 1.0
00007  *****/
00008
00009 #ifndef NOMBRE_LIBRERIA_H
00010 #define NOMBRE_LIBRERIA_H
00011
00012 /***** INCLUDES *****/
00013 // Includes necesarios para la API pública
00014 #include <stdint.h>
00015 #include "esp_log.h" // Para log
00016
00017 /***** DEFINES PÚBLICAS *****/
00018 // Constantes, flags o configuraciones visibles desde fuera de la librería
00019
00020 /***** TIPOS PÚBLICOS *****/
00021 // Estructuras, enums, typedefs públicos
00022
00023 /***** VARIABLES PÚBLICAS *****/
00024 // Variables globales visibles (si realmente se necesitan)
00025
00026 // extern int ejemplo_variable_publica;
00027
00028 /***** FUNCIONES PÚBLICAS *****/
00033 // int libreria_init(void);
00034
00040 // int libreria_operacion(int valor);
00041
00042 #endif /* NOMBRE_LIBRERIA_H */
00043

```

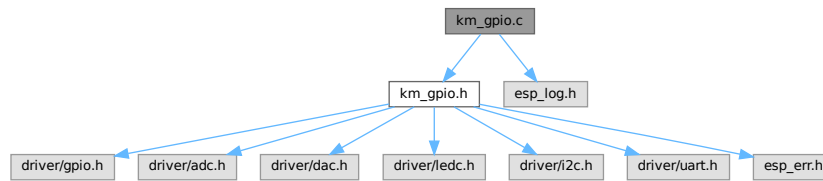
4.9 km_gpio.c File Reference

```

#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](#).

```
00280 {
00281     i2c_config_t conf = {
00282         .mode = I2C_MODE_MASTER,
00283         .sda_io_num = PIN_I2C_SDA,
00284         .scl_io_num = PIN_I2C_SCL,
00285         .sda_pullup_en = GPIO_PULLUP_ENABLE,
00286         .scl_pullup_en = GPIO_PULLUP_ENABLE,
00287         .master.clk_speed = 400000
00288     };
00289     esp_err_t ret = i2c_param_config(I2C_NUM_0, &conf);
00290     if (ret != ESP_OK) return ret;
00291
00292     return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
00293 }
```

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](#).

```
00180 {
00181     esp_err_t ret;
00182
00183     // Enable DAC channels
00184     ret = dac_output_enable(DAC_CHAN_0); // CMD_ACC
00185     if (ret != ESP_OK) return ret;
00186     ret = dac_output_enable(DAC_CHAN_1); // CMD_BRAKE
00187     if (ret != ESP_OK) return ret;
00188
00189     // Setup PWM (LEDC)
00190     ledc_timer_config_t pwm_timer = {
00191         .speed_mode = LEDC_HIGH_SPEED_MODE,
00192         .duty_resolution = LEDC_TIMER_8_BIT,
00193         .timer_num = LEDC_TIMER_0,
00194         .freq_hz = 1000,
00195         .clk_cfg = LEDC_AUTO_CLK
00196     };
00197     ret = ledc_timer_config(&pwm_timer);
00198     if (ret != ESP_OK) return ret;
00199
00200     ledc_channel_config_t pwm_channel = {
00201         .gpio_num = (gpio_num_t)PIN_STEER_PWM,
00202         .speed_mode = LEDC_HIGH_SPEED_MODE,
00203         .channel = LEDC_CHANNEL_0,
00204         .intr_type = LEDC_INTR_DISABLE,
00205         .timer_sel = LEDC_TIMER_0,
00206         .duty = 0
00207     };
00208     ret = ledc_channel_config(&pwm_channel);
00209     if (ret != ESP_OK) return ret;
00210
00211     return ESP_OK;
00212 }
```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> 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](#).

```
00232 {
00233     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00234     int raw_out_adc2 = 0;
00235
00236     switch (gpio)
00237     {
00238         case GPIO_NUM_36: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_0); // pressure 1
00239         case GPIO_NUM_39: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_3); // pressure 2
00240         case GPIO_NUM_34: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_6); // pressure 3
00241         case GPIO_NUM_35: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_7); // pedal acc
00242         case GPIO_NUM_32: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_4); // pedal brake
00243         case GPIO_NUM_33: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_5); // hydraulic 1
00244     }
```

```

00244         case GPIO_NUM_13:    // hydraulic 2
00245             if (adc2_get_raw(ADC2_CHANNEL_4, ADC_WIDTH_BIT_12, &raw_out_adc2) == ESP_OK)
00246                 return raw_out_adc2;
00247             return 0;
00248
00249         default: return 0;
00250     }
00251 }

```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> representing the pin to read.
------------	---

Returns

0 if LOW, 1 if HIGH.

Definition at line 215 of file [km_gpio.c](#).

```

00216 {
00217     return gpio_get_level((gpio_num_t)(pin->pin_bit_mask));
00218 }

```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> representing the DAC pin.
<i>value</i>	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](#).

```

00255 {
00256     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00257
00258     if (gpio == PIN_CMD_ACC) return dac_output_voltage(DAC_CHAN_0, value);
00259     if (gpio == PIN_CMD_BRAKE) return dac_output_voltage(DAC_CHAN_1, value);
00260
00261     return ESP_ERR_INVALID_ARG;
00262 }

```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> 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](#).

```
00221 {
00222     return gpio_set_level((gpio_num_t) (pin->pin_bit_mask), 1);
00223 }
```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> 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](#).

```
00226 {
00227     return gpio_set_level((gpio_num_t) (pin->pin_bit_mask), 0);
00228 }
```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> representing the PWM pin.
<i>duty</i>	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`.

```
00266 {
00267     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00268
00269     if (gpio == GPIO_NUM_27) // Steering PWM
00270     {
00271         ledc_set_duty(LEDCH_HIGH_SPEED_MODE, LEDC_CHANNEL_0, duty);
00272         return ledc_update_duty(LEDCH_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
00273     }
00274
00275     return ESP_ERR_INVALID_ARG;
00276 }
```

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`.

```
00088     {
00089         .pin_bit_mask = 1ULL << PIN_CMD_ACC,
00090         .mode = GPIO_MODE_OUTPUT,
00091         .pull_up_en = GPIO_PULLUP_DISABLE,
00092         .pull_down_en = GPIO_PULLDOWN_DISABLE,
00093         .intr_type = GPIO_INTR_DISABLE
00094     };
```

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`.

```
00096     {
00097         .pin_bit_mask = 1ULL << PIN_CMD_BRAKE,
00098         .mode = GPIO_MODE_OUTPUT,
00099         .pull_up_en = GPIO_PULLUP_DISABLE,
00100         .pull_down_en = GPIO_PULLDOWN_DISABLE,
00101         .intr_type = GPIO_INTR_DISABLE
00102     };
```

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](#).

```
00070     {
00071     .pin_bit_mask = 1ULL < PIN_HYDRAULIC_1,
00072     .mode = GPIO_MODE_INPUT,
00073     .pull_up_en = GPIO_PULLUP_DISABLE,
00074     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00075     .intr_type = GPIO_INTR_DISABLE
00076 };
```

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](#).

```
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 };
```

4.9.2.5 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](#).

```
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 };
```

4.9.2.6 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](#).

```
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 };
```

4.9.2.10 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](#).

```
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 };
```

4.9.2.11 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](#).

```
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 };
```

4.9.2.12 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](#).

```
00164     {
00165     .pin_bit_mask = 1ULL << PIN_STATUS_LED,
00166     .mode = GPIO_MODE_OUTPUT,
00167     .pull_up_en = GPIO_PULLUP_DISABLE,
00168     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00169     .intr_type = GPIO_INTR_DISABLE
00170 };
```

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](#).

```
00113     {
00114     .pin_bit_mask = 1ULL << PIN_STEER_DIR,
00115     .mode = GPIO_MODE_OUTPUT,
00116     .pull_up_en = GPIO_PULLUP_DISABLE,
00117     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00118     .intr_type = GPIO_INTR_DISABLE
00119 };
```

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](#).

```
00105     {
00106     .pin_bit_mask = 1ULL << PIN_STEER_PWM,
00107     .mode = GPIO_MODE_OUTPUT,
00108     .pull_up_en = GPIO_PULLUP_DISABLE,
00109     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00110     .intr_type = GPIO_INTR_DISABLE
00111 };
```

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  };
```

```

00053
00054 const gpio_config_t pin_pedal_acc = {
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 };
00061
00062 const gpio_config_t pin_pedal_brake = {
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 };
00069
00070 const gpio_config_t pin_hydraulic_1 = {
00071     .pin_bit_mask = 1ULL < PIN_HYDRAULIC_1,
00072     .mode = GPIO_MODE_INPUT,
00073     .pull_up_en = GPIO_PULLUP_DISABLE,
00074     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00075     .intr_type = GPIO_INTR_DISABLE
00076 };
00077
00078 /* ADC2 - allowed (WiFi not used) */
00079 const gpio_config_t pin_hydraulic_2 = {
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 };
00086
00087 /* ----- DAC OUTPUTS ----- */
00088 const gpio_config_t pin_cmd_acc = {
00089     .pin_bit_mask = 1ULL < PIN_CMD_ACC,
00090     .mode = GPIO_MODE_OUTPUT,
00091     .pull_up_en = GPIO_PULLUP_DISABLE,
00092     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00093     .intr_type = GPIO_INTR_DISABLE
00094 };
00095
00096 const gpio_config_t pin_cmd_brake = {
00097     .pin_bit_mask = 1ULL < PIN_CMD_BRAKE,
00098     .mode = GPIO_MODE_OUTPUT,
00099     .pull_up_en = GPIO_PULLUP_DISABLE,
00100     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00101     .intr_type = GPIO_INTR_DISABLE
00102 };
00103
00104 /* ----- STEERING MOTOR ----- */
00105 const gpio_config_t pin_steer_pwm = {
00106     .pin_bit_mask = 1ULL < PIN_STEER_PWM,
00107     .mode = GPIO_MODE_OUTPUT,
00108     .pull_up_en = GPIO_PULLUP_DISABLE,
00109     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00110     .intr_type = GPIO_INTR_DISABLE
00111 };
00112
00113 const gpio_config_t pin_steer_dir = {
00114     .pin_bit_mask = 1ULL < PIN_STEER_DIR,
00115     .mode = GPIO_MODE_OUTPUT,
00116     .pull_up_en = GPIO_PULLUP_DISABLE,
00117     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00118     .intr_type = GPIO_INTR_DISABLE
00119 };
00120
00121 /* ----- HALL SENSORS ----- */
00122 const gpio_config_t pin_motor_hall_1 = {
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 };
00129
00130 const gpio_config_t pin_motor_hall_2 = {
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 };
00137
00138 const gpio_config_t pin_motor_hall_3 = {
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 };
00145
00146 /* ----- I2C (AS5600) ----- */
00147 const gpio_config_t pin_i2c_scl = {
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 };
00154
00155 const gpio_config_t pin_i2c_sda = {
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 };
00162
00163 /* ----- STATUS LED ----- */
00164 const gpio_config_t pin_status_led = {
00165     .pin_bit_mask = 1ULL < PIN_STATUS_LED,
00166     .mode = GPIO_MODE_OUTPUT,
00167     .pull_up_en = GPIO_PULLUP_DISABLE,
00168     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00169     .intr_type = GPIO_INTR_DISABLE
00170 };
00171
00172
00173 /***** DECLARACION FUNCIONES PRIVADAS *****/
00174
00175
00176 /***** FUNCIONES PÚBLICAS *****/
00177
00178 /* ----- Initialization ----- */
00179 esp_err_t KM_GPIO_Init(void)
00180 {
00181     esp_err_t ret;
00182
00183     // Enable DAC channels
00184     ret = dac_output_enable(DAC_CHAN_0); // CMD_ACC
00185     if (ret != ESP_OK) return ret;
00186     ret = dac_output_enable(DAC_CHAN_1); // CMD_BRAKE
00187     if (ret != ESP_OK) return ret;
00188
00189     // Setup PWM (LEDC)
00190     ledc_timer_config_t pwm_timer = {
00191         .speed_mode = LEDC_HIGH_SPEED_MODE,
00192         .duty_resolution = LEDC_TIMER_8_BIT,
00193         .timer_num = LEDC_TIMER_0,
00194         .freq_hz = 1000,
00195         .clk_cfg = LEDC_AUTO_CLK
00196     };
00197     ret = ledc_timer_config(&pwm_timer);
00198     if (ret != ESP_OK) return ret;
00199
00200     ledc_channel_config_t pwm_channel = {
00201         .gpio_num = (gpio_num_t)PIN_STEER_PWM,
00202         .speed_mode = LEDC_HIGH_SPEED_MODE,
00203         .channel = LEDC_CHANNEL_0,
00204         .intr_type = LEDC_INTR_DISABLE,
00205         .timer_sel = LEDC_TIMER_0,
00206         .duty = 0
00207     };
00208     ret = ledc_channel_config(&pwm_channel);
00209     if (ret != ESP_OK) return ret;
00210
00211     return ESP_OK;
00212 }
00213
00214 /* ----- Digital GPIO ----- */
00215 uint8_t KM_GPIO_ReadPin(const gpio_config_t *pin)
00216 {
00217     return gpio_get_level((gpio_num_t)(pin->pin_bit_mask));
00218 }
00219
00220 esp_err_t KM_GPIO_WritePinHigh(const gpio_config_t *pin)
00221 {
00222     return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 1);
00223 }
00224
00225 esp_err_t KM_GPIO_WritePinLow(const gpio_config_t *pin)
00226 {

```

```

00227     return gpio_set_level((gpio_num_t)(pin->pin_bit_mask), 0);
00228 }
00229
00230 /* ----- ADC ----- */
00231 uint16_t KM_GPIO_ReadADC(const gpio_config_t *pin)
00232 {
00233     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00234     int raw_out_adc2 = 0;
00235
00236     switch (gpio)
00237     {
00238         case GPIO_NUM_36: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_0); // pressure 1
00239         case GPIO_NUM_39: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_3); // pressure 2
00240         case GPIO_NUM_34: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_6); // pressure 3
00241         case GPIO_NUM_35: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_7); // pedal acc
00242         case GPIO_NUM_32: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_4); // pedal brake
00243         case GPIO_NUM_33: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_5); // hydraulic 1
00244         case GPIO_NUM_13: // hydraulic 2
00245             if (adc2_get_raw(ADC2_CHANNEL_4, ADC_WIDTH_BIT_12, &raw_out_adc2) == ESP_OK)
00246                 return raw_out_adc2;
00247             return 0;
00248         default: return 0;
00249     }
00250 }
00251
00252 /* ----- DAC ----- */
00253 esp_err_t KM_GPIO_WriteDAC(const gpio_config_t *pin, uint8_t value)
00254 {
00255     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00256
00257     if (gpio == PIN_CMD_ACC) return dac_output_voltage(DAC_CHAN_0, value);
00258     if (gpio == PIN_CMD_BRAKE) return dac_output_voltage(DAC_CHAN_1, value);
00259
00260     return ESP_ERR_INVALID_ARG;
00261 }
00262
00263 /* ----- PWM ----- */
00264 esp_err_t KM_GPIO_WritePWM(const gpio_config_t *pin, uint8_t duty)
00265 {
00266     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00267
00268     if (gpio == GPIO_NUM_27) // Steering PWM
00269     {
00270         ledc_set_duty(LEDCH_HIGH_SPEED_MODE, LEDC_CHANNEL_0, duty);
00271         return ledc_update_duty(LEDCH_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
00272     }
00273
00274     return ESP_ERR_INVALID_ARG;
00275 }
00276
00277 /* ----- I2C ----- */
00278 esp_err_t KM_GPIO_I2CInit(void)
00279 {
00280     i2c_config_t conf = {
00281         .mode = I2C_MODE_MASTER,
00282         .sda_io_num = PIN_I2C_SDA,
00283         .scl_io_num = PIN_I2C_SCL,
00284         .sda_pullup_en = GPIO_PULLUP_ENABLE,
00285         .scl_pullup_en = GPIO_PULLUP_ENABLE,
00286         .master.clk_speed = 400000
00287     };
00288     esp_err_t ret = i2c_param_config(I2C_NUM_0, &conf);
00289     if (ret != ESP_OK) return ret;
00290
00291     return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
00292 }
00293
00294
00295
00296 /***** FUNCIONES PRIVADAS *****/
00300 // void funcion_privada(void);
00301
00302 /***** FIN DE ARCHIVO *****/

```

4.11 km_gpio.h File Reference

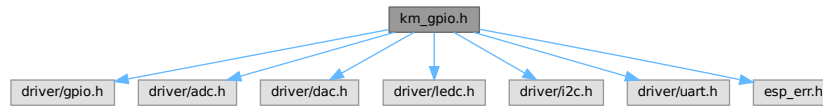
```

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

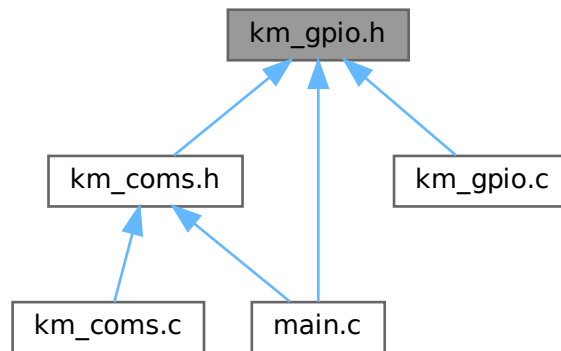
```

```
#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`.

```
00280 {
00281     i2c_config_t conf = {
00282         .mode = I2C_MODE_MASTER,
00283         .sda_io_num = PIN_I2C_SDA,
00284         .scl_io_num = PIN_I2C_SCL,
00285         .sda_pullup_en = GPIO_PULLUP_ENABLE,
00286         .scl_pullup_en = GPIO_PULLUP_ENABLE,
00287         .master.clk_speed = 400000
00288     };
00289     esp_err_t ret = i2c_param_config(I2C_NUM_0, &conf);
00290     if (ret != ESP_OK) return ret;
00291
00292     return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
00293 }
```

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](#).

```
00180 {
00181     esp_err_t ret;
00182
00183     // Enable DAC channels
00184     ret = dac_output_enable(DAC_CHAN_0); // CMD_ACC
00185     if (ret != ESP_OK) return ret;
00186     ret = dac_output_enable(DAC_CHAN_1); // CMD_BRAKE
00187     if (ret != ESP_OK) return ret;
00188
00189     // Setup PWM (LEDC)
00190     ledc_timer_config_t pwm_timer = {
00191         .speed_mode = LEDC_HIGH_SPEED_MODE,
00192         .duty_resolution = LEDC_TIMER_8_BIT,
00193         .timer_num = LEDC_TIMER_0,
00194         .freq_hz = 1000,
00195         .clk_cfg = LEDC_AUTO_CLK
00196     };
00197     ret = ledc_timer_config(&pwm_timer);
00198     if (ret != ESP_OK) return ret;
00199
00200     ledc_channel_config_t pwm_channel = {
00201         .gpio_num = (gpio_num_t)PIN_STEER_PWM,
00202         .speed_mode = LEDC_HIGH_SPEED_MODE,
00203         .channel = LEDC_CHANNEL_0,
00204         .intr_type = LEDC_INTR_DISABLE,
00205         .timer_sel = LEDC_TIMER_0,
00206         .duty = 0
00207     };
00208     ret = ledc_channel_config(&pwm_channel);
00209     if (ret != ESP_OK) return ret;
00210
00211     return ESP_OK;
00212 }
```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> 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](#).

```
00232 {
00233     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00234     int raw_out_adc2 = 0;
00235
00236     switch (gpio)
00237     {
00238         case GPIO_NUM_36: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_0); // pressure 1
00239         case GPIO_NUM_39: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_3); // pressure 2
00240         case GPIO_NUM_34: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_6); // pressure 3
00241         case GPIO_NUM_35: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_7); // pedal acc
00242         case GPIO_NUM_32: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_4); // pedal brake
00243         case GPIO_NUM_33: return (uint16_t)adc1_get_raw(ADC1_CHANNEL_5); // hydraulic 1
00244     }
```

```

00244         case GPIO_NUM_13:    // hydraulic 2
00245             if (adc2_get_raw(ADC2_CHANNEL_4, ADC_WIDTH_BIT_12, &raw_out_adc2) == ESP_OK)
00246                 return raw_out_adc2;
00247             return 0;
00248
00249         default: return 0;
00250     }
00251 }

```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> representing the pin to read.
------------	---

Returns

0 if LOW, 1 if HIGH.

Definition at line 215 of file [km_gpio.c](#).

```

00216 {
00217     return gpio_get_level((gpio_num_t)(pin->pin_bit_mask));
00218 }

```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> representing the DAC pin.
<i>value</i>	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](#).

```

00255 {
00256     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00257
00258     if (gpio == PIN_CMD_ACC) return dac_output_voltage(DAC_CHAN_0, value);
00259     if (gpio == PIN_CMD_BRAKE) return dac_output_voltage(DAC_CHAN_1, value);
00260
00261     return ESP_ERR_INVALID_ARG;
00262 }

```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> 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](#).

```
00221 {
00222     return gpio_set_level((gpio_num_t) (pin->pin_bit_mask), 1);
00223 }
```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> 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](#).

```
00226 {
00227     return gpio_set_level((gpio_num_t) (pin->pin_bit_mask), 0);
00228 }
```

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

<i>pin</i>	Pointer to a <code>gpio_config_t</code> representing the PWM pin.
<i>duty</i>	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`.

```

00266 {
00267     gpio_num_t gpio = (gpio_num_t)(pin->pin_bit_mask);
00268
00269     if (gpio == GPIO_NUM_27) // Steering PWM
00270     {
00271         ledc_set_duty(LEDCH_HIGH_SPEED_MODE, LEDC_CHANNEL_0, duty);
00272         return ledc_update_duty(LEDCH_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
00273     }
00274
00275     return ESP_ERR_INVALID_ARG;
00276 }
```

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`.

```

00088     {
00089         .pin_bit_mask = 1ULL < PIN_CMD_ACC,
00090         .mode = GPIO_MODE_OUTPUT,
00091         .pull_up_en = GPIO_PULLUP_DISABLE,
00092         .pull_down_en = GPIO_PULLDOWN_DISABLE,
00093         .intr_type = GPIO_INTR_DISABLE
00094     };
```

4.11.3.2 pin_cmd_brake

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

Definition at line 96 of file `km_gpio.c`.

```

00096     {
00097         .pin_bit_mask = 1ULL < PIN_CMD_BRAKE,
00098         .mode = GPIO_MODE_OUTPUT,
00099         .pull_up_en = GPIO_PULLUP_DISABLE,
00100         .pull_down_en = GPIO_PULLDOWN_DISABLE,
00101         .intr_type = GPIO_INTR_DISABLE
00102     };
```

4.11.3.3 pin_hydraulic_1

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

Definition at line 70 of file `km_gpio.c`.

```

00070     {
00071         .pin_bit_mask = 1ULL < PIN_HYDRAULIC_1,
00072         .mode = GPIO_MODE_INPUT,
00073         .pull_up_en = GPIO_PULLUP_DISABLE,
00074         .pull_down_en = GPIO_PULLDOWN_DISABLE,
00075         .intr_type = GPIO_INTR_DISABLE
00076     };
```

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](#).

```
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 };
```

4.11.3.15 pin_status_led

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

Definition at line 164 of file [km_gpio.c](#).

```
00164     {
00165     .pin_bit_mask = 1ULL < PIN_STATUS_LED,
00166     .mode = GPIO_MODE_OUTPUT,
00167     .pull_up_en = GPIO_PULLUP_DISABLE,
00168     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00169     .intr_type = GPIO_INTR_DISABLE
00170 };
```

4.11.3.16 pin_steer_dir

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

Definition at line 113 of file [km_gpio.c](#).

```
00113     {
00114     .pin_bit_mask = 1ULL < PIN_STEER_DIR,
00115     .mode = GPIO_MODE_OUTPUT,
00116     .pull_up_en = GPIO_PULLUP_DISABLE,
00117     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00118     .intr_type = GPIO_INTR_DISABLE
00119 };
```

4.11.3.17 pin_steer_pwm

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

Definition at line 105 of file [km_gpio.c](#).

```
00105     {
00106     .pin_bit_mask = 1ULL < PIN_STEER_PWM,
00107     .mode = GPIO_MODE_OUTPUT,
00108     .pull_up_en = GPIO_PULLUP_DISABLE,
00109     .pull_down_en = GPIO_PULLDOWN_DISABLE,
00110     .intr_type = GPIO_INTR_DISABLE
00111 };
```

4.11.3.18 uart_config

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

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 };
```

4.12 km_gpio.h

[Go to the documentation of this file.](#)

```

00001 /*****
00002  * @file      km_gpio.h
00003  * @brief     Interfaz pública de la librería.
00004  * @author    Adrian Navarredonda Arizaleta
00005  * @date      7-2-2026
00006  * @version   1.0
00007  *****/
00008
00009 #ifndef KM_GPIO_H
00010 #define KM_GPIO_H
00011
00012 /***** INCLUDES *****/
00013 // Includes necesarios para la API pública
00014 #include "driver/gpio.h"
00015 #include "driver/adc.h"
00016 #include "driver/dac.h"
00017 #include "driver/ledc.h"
00018 #include "driver/i2c.h"
00019 #include "driver/uart.h"
00020 #include "esp_err.h"
00021
00022 /***** DEFINES PÚBLICAS *****/
00023 // Constantes, flags o configuraciones visibles desde fuera de la librería
00024
00025 #ifndef LEDC_HIGH_SPEED_MODE
00026 #define LEDC_HIGH_SPEED_MODE 0
00027 #endif
00028 #ifndef LEDC_LOW_SPEED_MODE
00029 #define LEDC_LOW_SPEED_MODE 1
00030 #endif
00031
00032 /* =====
00033  * ESP32-DevKitC V4 (ESP32-WROOM-32E)
00034  * Pin assignment - NO WiFi
00035  * ===== */
00036
00037 /* ----- USB (UART0 - debug console) ----- */
00038 #define PIN_USB_UART_TX      GPIO_NUM_1    // U0TXD
00039 #define PIN_USB_UART_RX      GPIO_NUM_3    // U0RXD
00040
00041 /* ----- UART2 (wired to ORIN) ----- */
00042 #define PIN_ORIN_UART_TX     GPIO_NUM_17   // U2TXD
00043 #define PIN_ORIN_UART_RX     GPIO_NUM_16   // U2RXD
00044
00045 /* ----- ADC INPUTS (Sensors) ----- */
00046 /* ADC1 - input only, WiFi safe */
00047 #define PIN_PEDAL_BRAKE      GPIO_NUM_32   // ADC1_CH4
00048 #define PIN_HYDRAULIC_1      GPIO_NUM_33   // ADC1_CH5
00049 #define PIN_PRESSURE_3       GPIO_NUM_34   // ADC1_CH6
00050 #define PIN_PEDAL_ACC        GPIO_NUM_35   // ADC1_CH7
00051 #define PIN_PRESSURE_1        GPIO_NUM_36   // ADC1_CH0 (VP)
00052 #define PIN_PRESSURE_2        GPIO_NUM_39   // ADC1_CH3 (VN)
00053
00054 /* ADC2 - allowed (WiFi not used) */
00055 #define PIN_HYDRAULIC_2      GPIO_NUM_13   // ADC2_CH4 (strap pin)
00056
00057 /* ----- DAC OUTPUTS ----- */
00058 #define PIN_CMD_ACC          GPIO_NUM_30   // DAC1 !!!!!Antes era el 25
00059 #define PIN_CMD_BRAKE        GPIO_NUM_26   // DAC2
00060
00061 /* ----- STEERING MOTOR ----- */
00062 #define PIN_STEER_PWM        GPIO_NUM_27   // PWM Steering
00063 #define PIN_STEER_DIR        GPIO_NUM_14   // Direction steering
00064
00065 /* ----- HALL SENSORS ----- */
00066 #define PIN_MOTOR_HALL_1      GPIO_NUM_18   // HALL 1 motor
00067 #define PIN_MOTOR_HALL_2      GPIO_NUM_19   // HALL 2 motor
00068 #define PIN_MOTOR_HALL_3      GPIO_NUM_31   // HALL 3 motor !!!!!!!!Antes era el 23
00069
00070 /* ----- I2C (AS5600) ----- */
00071 #define PIN_I2C_SDA          GPIO_NUM_21   // I2C SDA
00072 #define PIN_I2C_SCL          GPIO_NUM_32   // I2C SCL !!!!!!!!!!!!!Antes era el 22
00073
00074 /* ----- STATUS LED ----- */
00075 #define PIN_STATUS_LED        GPIO_NUM_2    // Strap pin (keep LOW at boot)
00076
00077 /* =====
00078  * GPIO RESTRICTIONS (DO NOT USE)
00079  * GPIO 6-11 : SPI FLASH
00080  * GPIO 34-39: INPUT ONLY
00081  * ===== */
00082

```

```

00083 /***** TIPOS PÚBLICOS *****/
00084 // Estructuras, enums, typedefs públicos
00085
00086 /***** VARIABLES PÚBLICAS *****/
00087 // Variables globales visibles (si realmente se necesitan)
00088
00089 /* ----- USB (UART0 to ORIN) ----- */
00090 extern const uart_config_t uart_config;
00091
00092 /* ----- ADC INPUTS (Sensors) ----- */
00093 /* ADC1 - input only, WiFi safe */
00094
00095 extern const gpio_config_t pin_pressure_1;
00096 extern const gpio_config_t pin_pressure_2;
00097 extern const gpio_config_t pin_pressure_3;
00098 extern const gpio_config_t pin_pedal_acc;
00099 extern const gpio_config_t pin_pedal_brake;
00100 extern const gpio_config_t pin_hydraulic_1;
00101
00102 /* ADC2 - allowed (WiFi not used) */
00103
00104 extern const gpio_config_t pin_hydraulic_2;
00105
00106 /* ----- DAC OUTPUTS ----- */
00107
00108 extern const gpio_config_t pin_cmd_acc;
00109 extern const gpio_config_t pin_cmd_brake;
00110
00111 /* ----- STEERING MOTOR ----- */
00112
00113 extern const gpio_config_t pin_steer_pwm;
00114 extern const gpio_config_t pin_steer_dir;
00115
00116 /* ----- HALL SENSORS ----- */
00117
00118 extern const gpio_config_t pin_motor_hall_1;
00119 extern const gpio_config_t pin_motor_hall_2;
00120 extern const gpio_config_t pin_motor_hall_3;
00121
00122 /* ----- I2C (AS5600) ----- */
00123
00124 extern const gpio_config_t pin_i2c_scl;
00125 extern const gpio_config_t pin_i2c_sda;
00126
00127 /* ----- STATUS LED ----- */
00128
00129 extern const gpio_config_t pin_status_led;
00130
00131 /***** FUNCIONES PÚBLICAS *****/
00145 esp_err_t KM_GPIO_Init(void);
00146
00147 /* ----- Digital GPIO ----- */
00148
00155 uint8_t KM_GPIO_ReadPin(const gpio_config_t *pin);
00156
00163 esp_err_t KM_GPIO_WritePinHigh(const gpio_config_t *pin);
00164
00171 esp_err_t KM_GPIO_WritePinLow(const gpio_config_t *pin);
00172
00173 /* ----- ADC ----- */
00174
00183 uint16_t KM_GPIO_ReadADC(const gpio_config_t *pin);
00184
00185 /* ----- DAC ----- */
00186
00196 esp_err_t KM_GPIO_WriteDAC(const gpio_config_t *pin, uint8_t value);
00197
00198 /* ----- PWM ----- */
00199
00209 esp_err_t KM_GPIO_WritePWM(const gpio_config_t *pin, uint8_t duty);
00210
00211 /* ----- I2C ----- */
00212
00220 esp_err_t KM_GPIO_I2CInit(void);
00221
00222 #endif /* KM_GPIO_H */

```

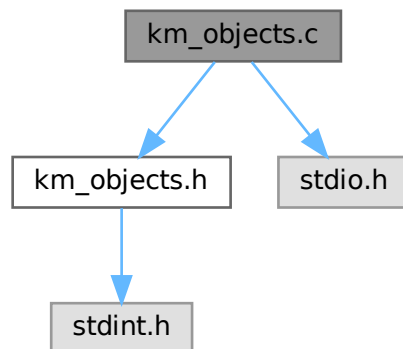
4.13 km_objects.c File Reference

```

#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

<i>valor</i>	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.](#)

```

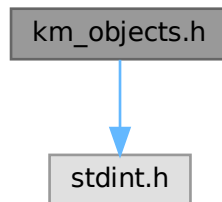
00001 /*****
00002  * @file    km_objects.c
00003  * @brief   Implementación de la librería.
00004  *****/
00005
00006 #include "km_objects.h"
00007 #include <stdio.h> // solo si es necesario para debug interno
00008
00009 /***** INCLUDES INTERNOS *****/
00010 // Headers internos opcionales, dependencias 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

```

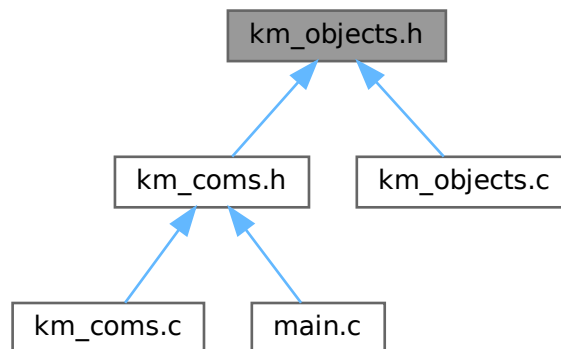
4.15 km_objects.h File Reference

```
#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
00035 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
```

```

00038 ACTUAL_SHUTDOWN,          // Actual state of the SHUTDOWN
00039
00040
00041 KM_OBJ_LAST                 // Este debe de ser siempre el ultimo
00042 } 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

<i>valor</i>	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 }

```

4.15.3.2 KM_OBJ_SetObjectValue()

```

uint8_t KM_OBJ_SetObjectValue (
    km_objects_t object,
    int64_t value )

```

Establece el valor de un objeto.

Parameters

<i>ID</i>	del objeto a establecer
-----------	-------------------------

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](#).

```

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

```

4.16 km_objects.h

[Go to the documentation of this file.](#)

```

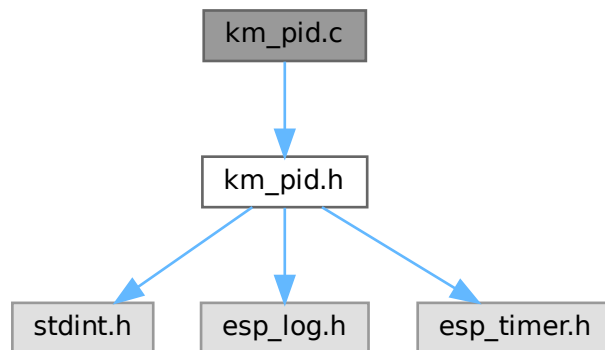
00001 /*****
00002  * @file      km_objects.h
00003  * @brief     Interfaz pública de la librería.
00004  * @author    Adrian Navarredonda Arizaleta
00005  * @date      27-02-2026
00006  * @version   1.0
00007  *****/
00008
00009 #ifndef KM_OBJECTS_H
00010 #define KM_OBJECTS_H
00011
00012 /***** INCLUDES *****/
00013 // Includes necesarios para la API pública
00014 #include <stdint.h>
00015
00016 /***** DEFINES PÚBLICAS *****/
00017 // Constantes, flags o configuraciones visibles desde fuera de la librería
00018 #define OBJECT_VALUE_ERROR -9223372036854775808
00019
00020 /***** TIPOS PÚBLICOS *****/
00021 // Estructuras, enums, typedefs públicos
00022
00023 typedef enum
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_ACCELERATION,   // 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
00035                             // pressure) x100
00036     ACTUAL_STEERING,       // Actual value of the steering in rad x100
00037     MISION_ESP,            // Mision that is executing the ESP
00038     MACHINE_STATE_ESP,     // State inside the state machine of the ESP
00039     ACTUAL_SHUTDOWN,       // Actual state of the SHUTDOWN
00040
00041     KM_OBJ_LAST            // Este debe de ser siempre el ultimo
00042 } km_objects_t;
00043
00044 /***** VARIABLES PÚBLICAS *****/
00045 // Variables globales visibles (si realmente se necesitan)
00046
00047 // extern int ejemplo_variable_publica;
00048
00049 /***** FUNCIONES PÚBLICAS *****/
00050 uint8_t KM_OBJ_SetObjectValue(km_objects_t object, int64_t value);
00051
00052 int64_t KM_OBJ_GetObjectValue(km_objects_t object);
00053
00054 #endif /* KM_OBJECTS_H */
00055
00056

```

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     // Update state
00077     controller->lastError = error;
00078     controller->lastTime = currentTime;
00079
00080     return output;
00081 }
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.](#)

```

00001 /*****
00002  * @file      KM_PID.c
00003  * @brief     Implementación de la librería.
00004  * @author    Adrian Navarredonda Arizaleta
00005  *****/
00006
00007 #include "km_pid.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 /***** DECLARACION FUNCIONES PRIVADAS *****/
00020
00021 /***** FUNCIONES PÚBLICAS *****/
00022 PID_Controller KM_PID_Init(float kp_val, float ki_val, float kd_val) {
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 }
00036
00037 float KM_PID_Calculate(PID_Controller *controller, float setpoint, float measurement) {
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     // Update state
00077     controller->lastError = error;
00078     controller->lastTime = currentTime;
00079
00080     return output;
00081 }
00082 }

```

```

00083
00084 void KM_PID_SetTunings(PID_Controller *controller, float kp, float ki, float kd) {
00085     controller->kp = kp;
00086     controller->ki = ki;
00087     controller->kd = kd;
00088 }
00089
00090 void KM_PID_SetOutputLimits(PID_Controller *controller, float min, float max) {
00091     controller->outputMin = min;
00092     controller->outputMax = max;
00093 }
00094
00095 void KM_PID_SetIntegralLimits(PID_Controller *controller, float min, float max) {
00096     controller->integralMin = min;
00097     controller->integralMax = max;
00098 }
00099
00100 void KM_PID_Reset(PID_Controller *controller){
00101     controller->integral = 0.0f;
00102     controller->lastError = 0.0f;
00103     controller->lastTime = esp_timer_get_time();
00104 }
00105
00106 void KM_PID_GetTunings(PID_Controller *controller, float kp, float ki, float kd) {
00107     controller->kp = kp;
00108     controller->ki = ki;
00109     controller->kd = kd;
00110 }
00111
00112 // Get integral value (for debugging)
00113 float KM_PID_GetIntegral(PID_Controller *controller){
00114     return controller->integral;
00115 }
00116
00117 /***** FUNCIONES PRIVADAS *****/
00118
00119 /***** FIN DE ARCHIVO *****/

```

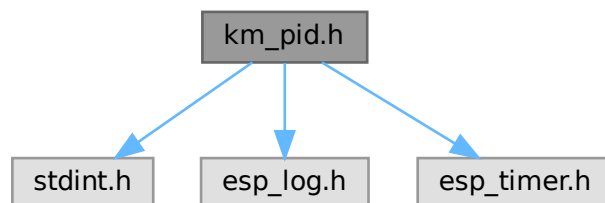
4.19 km_pid.h File Reference

```

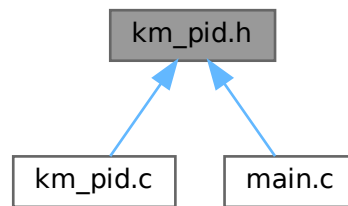
#include <stdint.h>
#include "esp_log.h"
#include "esp_timer.h"

```

Include dependency graph for km_pid.h:



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



Classes

- struct [PID_Controller](#)
Structure that represents 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](#).

```
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     // Update state
00077     controller->lastError = error;
00078     controller->lastTime = currentTime;
00079
00080     return output;
00081 }
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.](#)

```

00001 /*****
00002  * @file      km_pid.h
00003  * @brief     Interfaz pública de la librería.
00004  * @author    Adrian Navarredonda Arizaleta
00005  * @date      25-01-2026
00006  * @version   1.0
00007  *****/
00008
00009 #ifndef KM_PID_H
00010 #define KM_PID_H
00011
00012 /***** INCLUDES *****/
00013 // Includes necesarios para la API pública
00014 #include <stdint.h>
00015 #include "esp_log.h" // Para log
00016 #include "esp_timer.h"
00017
00018 /***** DEFINES PÚBLICAS *****/
00019 // Constantes, flags o configuraciones visibles desde fuera de la librería
00020
00024 typedef struct {
00025     float kp;
00026     float ki;
00027     float kd;
00028     float integral;
00029     float lastError;
00030     uint64_t lastTime;
00032     // Limits
00033     float outputMin;
00034     float outputMax;
00035     float integralMin;
00036     float integralMax;
00037 } PID_Controller;
00038
00039 /***** TIPOS PÚBLICOS *****/
00040 // Estructuras, enums, typedefs públicos
00041
00042 /***** VARIABLES PÚBLICAS *****/
00043 // Variables globales visibles (si realmente se necesitan)
00044
00045 /***** FUNCIONES PÚBLICAS *****/
00046
00047 // Initialize with gains
00048 PID_Controller KM_PID_Init(float kp, float ki, float kd);
00049
00050 // Calculate PID output
00051 float KM_PID_Calculate(PID_Controller *controller, float setpoint, float measurement);
00052
00053 // Set tuning parameters at runtime
00054 void KM_PID_SetTunings(PID_Controller *controller, float kp, float ki, float kd);
00055
00056 // Set output limits
00057 void KM_PID_SetOutputLimits(PID_Controller *controller, float min, float max);
00058
00059 // Set integral limits (anti-windup)
00060 void KM_PID_SetIntegralLimits(PID_Controller *controller, float min, float max);
00061
00062 // Reset controller state
00063 void KM_PID_Reset(PID_Controller *controller);
00064
00065 // Get current gains
00066 void KM_PID_GetTunings(PID_Controller *controller, float kp, float ki, float kd);
00067
00068 // Get integral value (for debugging)
00069 float KM_PID_GetIntegral(PID_Controller *controller);
00070
00071 #endif // KM_PID_H

```

4.21 km_rtos.c File Reference

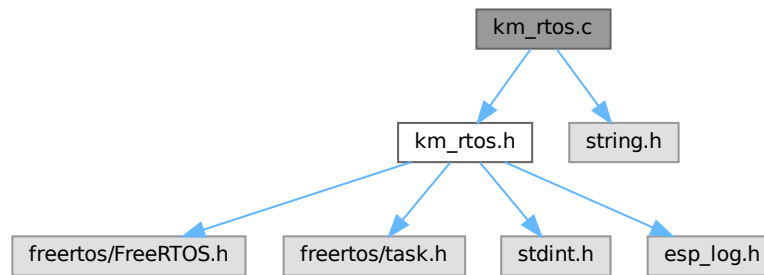
Implementation of the KM_RTOS task management library.

```

#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

<i>name</i>	Name of the task (used by FreeRTOS for debugging/logging)
<i>taskFn</i>	Function pointer to the task implementation (of type KM_RTOS_TaskFunction_t)
<i>context</i>	Pointer to a user-defined context or data structure passed to the task
<i>period_ms</i>	Task period in milliseconds (used for periodic scheduling)
<i>stackSize</i>	Stack size for the FreeRTOS task
<i>priority</i>	FreeRTOS task priority
<i>active</i>	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](#).

```
00063                                     {
00064
00065     RTOS_Task task;
00066
00067     task.name = name;
00068     task.taskFn = taskFn;
00069     task.context = context;
00070     task.period_ms = period_ms;
00071     task.stackSize = stackSize;
00072     task.priority = priority;
00073     task.active = active;
00074
00075     return task;
00076 }
```

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

<code>task</code>	The <code>RTOS_Task</code> 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`.

```

00078                                     {
00079
00080     // Buscar hueco libre en array y comprobar que no exista ya esa tarea
00081     for (uint8_t i = 0; i < RTOS_MAX_TASKS; i++) {
00082         // Hueco ocupado
00083         if (tasks[i].handle != NULL) continue;
00084
00085         tasks[i] = task;
00086         tasks[i].active = 1;
00087
00088         if (tasks[i].priority == 0) tasks[i].priority = RTOS_DEFAULT_PRIORITY;
00089         if (tasks[i].stackSize == 0) tasks[i].stackSize = RTOS_DEFAULT_STACK_SIZE;
00090         if (tasks[i].period_ms == 0) tasks[i].period_ms = RTOS_DEFAULT_PERIOD_MS;
00091
00092         BaseType_t result = xTaskCreate(
00093             KM_RTOS_TaskWrapper, // Wrapper
00094             task.name,
00095             task.stackSize,
00096             &tasks[i],           // Pointer to this task
00097             task.priority,
00098             &tasks[i].handle

```

```

00099     );
00100
00101     if (result != pdPASS) {
00102         memset(&tasks[i], 0, sizeof(RTOS_Task));
00103         return ESP_FAIL; // Error creating task
00104     }
00105
00106     return ESP_OK; // Success
00107 }
00108
00109 return ESP_FAIL; // Fail, no space in array
00110 }

```

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

<i>handle</i>	The FreeRTOS task handle whose priority should be changed.
<i>newPriority</i>	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`.

```

00181
00182
00183     int8_t index = KM_RTOS_FindTask(handle);
00184     if (index == -1) return ESP_FAIL;
00185
00186     vTaskPrioritySet(tasks[index].handle, newPriority);
00187
00188     // Actualizar informacion sobre tarea
00189     tasks[index].priority = newPriority;
00190
00191     return ESP_OK;
00192 }

```

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

<i>handle</i>	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-safe calling this while other tasks may concurrently access the `tasks` array.

Definition at line 113 of file [km_rtos.c](#).

```
00113                                     {
00114     // Search for task in array
00115
00116     int8_t index = KM_RTOS_FindTask(handle);
00117     if (index == -1) return ESP_FAIL;
00118
00119     if (tasks[index].handle == handle) {
00120         vTaskDelete(handle);
00121         memset(&tasks[index], 0, sizeof(RTOS_Task));
00122         return ESP_OK;
00123     }
00124
00125     // NO se ha encontrado la tarea
00126     return ESP_FAIL;
00127 }
```

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 `vTaskDelete`. 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     memset(tasks, 0, sizeof(tasks));
00059 }
00060 }
```

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

<i>handle</i>	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`.

```
00210     {
00211
00212     for (int8_t i = 0; i < RTOS_MAX_TASKS; i++) {
00213         if (tasks[i].handle == handle) {
00214             return i;
00215         }
00216     }
00217     return -1;
00218 }
00219 }
```

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

<i>handle</i>	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

<i>handle</i>	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](#).

```
00146                                     {
00147
00148     int8_t index = KM_RTOS_FindTask(handle);
00149     if (index == -1) return 0;
00150
00151     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 }
```

4.21.3.10 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

<i>handle</i>	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 }
```

4.21.3.11 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

<i>params</i>	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     memset(tasks, 0, sizeof(tasks));
00059 }
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) {
```

```

00064
00065     RTOS_Task task;
00066
00067     task.name = name;
00068     task.taskFn = taskFn;
00069     task.context = context;
00070     task.period_ms = period_ms;
00071     task.stackSize = stackSize;
00072     task.priority = priority;
00073     task.active = active;
00074
00075     return task;
00076 }
00077
00078 esp_err_t KM_RTOS_AddTask(RTOS_Task task){
00079
00080     // Buscar hueco libre en array y comprobar que no exista ya esa tarea
00081     for (uint8_t i = 0; i < RTOS_MAX_TASKS; i++) {
00082         // Hueco ocupado
00083         if (tasks[i].handle != NULL) continue;
00084
00085         tasks[i] = task;
00086         tasks[i].active = 1;
00087
00088         if (tasks[i].priority == 0) tasks[i].priority = RTOS_DEFAULT_PRIORITY;
00089         if (tasks[i].stackSize == 0) tasks[i].stackSize = RTOS_DEFAULT_STACK_SIZE;
00090         if (tasks[i].period_ms == 0) tasks[i].period_ms = RTOS_DEFAULT_PERIOD_MS;
00091
00092         BaseType_t result = xTaskCreate(
00093             KM_RTOS_TaskWrapper, // Wrapper
00094             task.name,
00095             task.stackSize,
00096             &tasks[i],           // Pointer to this task
00097             task.priority,
00098             &tasks[i].handle
00099         );
00100
00101         if (result != pdPASS) {
00102             memset(&tasks[i], 0, sizeof(RTOS_Task));
00103             return ESP_FAIL; // Error creating task
00104         }
00105
00106         return ESP_OK; // Success
00107     }
00108
00109     return ESP_FAIL; // Fail, no space in array
00110 }
00111
00112 // Destruir una tarea existente, devuelve 0 en caso de error, 1 en caso correcto
00113 esp_err_t KM_RTOS_DeleteTask(TaskHandle_t handle){
00114     // Search for task in array
00115
00116     int8_t index = KM_RTOS_FindTask(handle);
00117     if (index == -1) return ESP_FAIL;
00118
00119     if (tasks[index].handle == handle) {
00120         vTaskDelete(handle);
00121         memset(&tasks[index], 0, sizeof(RTOS_Task));
00122         return ESP_OK;
00123     }
00124
00125     // NO se ha encontrado la tarea
00126     return ESP_FAIL;
00127 }
00128
00129 // Suspender una tarea, devuelve 0 en caso de error, 1 en caso correcto
00130 esp_err_t KM_RTOS_SuspendTask(TaskHandle_t handle) {
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 }
00144
00145 // Reanudar una tarea, devuelve 0 en caso de error, 1 en caso correcto
00146 esp_err_t KM_RTOS_ResumeTask(TaskHandle_t handle){
00147
00148     int8_t index = KM_RTOS_FindTask(handle);
00149     if (index == -1) return 0;
00150

```

```

00151     if (tasks[index].handle == handle && !tasks[index].active) {
00152         vTaskResume(handle);
00153         tasks[index].active = 1;
00154     }
00155     return ESP_OK;
00156 }
00157 // No se ha encontrado la tarea o ya estaba activa
00158 return ESP_FAIL;
00159 }
00160 }
00161 // Reiniciar una tarea (destruir + volver a crear), devuelve 0 en caso de error, 1 en caso correcto
00162 esp_err_t KM_RTOS_RestartTask(TaskHandle_t handle){
00163     int8_t index = KM_RTOS_FindTask(handle);
00164     if (index == -1) return ESP_FAIL;
00165     RTOS_Task copy = tasks[index];
00166     // Destruir tarea
00167     if(KM_RTOS_DeleteTask(handle) == ESP_FAIL) return ESP_FAIL;
00168     // Crear tarea
00169     if(KM_RTOS_AddTask(copy) == ESP_FAIL) return ESP_FAIL;
00170     // Reiniciado correctamente
00171     return ESP_OK;
00172 }
00173 // Cambiar prioridad, devuelve 0 en caso de error, 1 en caso correcto
00174 esp_err_t KM_RTOS_ChangePriority(TaskHandle_t handle, uint8_t newPriority){
00175     int8_t index = KM_RTOS_FindTask(handle);
00176     if (index == -1) return ESP_FAIL;
00177     vTaskPrioritySet(tasks[index].handle, newPriority);
00178     // Actualizar informacion sobre tarea
00179     tasks[index].priority = newPriority;
00180     return ESP_OK;
00181 }
00182 /***** FUNCIONES PRIVADAS *****/
00183 int8_t KM_RTOS_FindTask(TaskHandle_t handle) {
00184     for (int8_t i = 0; i < RTOS_MAX_TASKS; i++) {
00185         if (tasks[i].handle == handle) {
00186             return i;
00187         }
00188     }
00189     return -1;
00190 }
00191 static void KM_RTOS_TaskWrapper(void *params) {
00192     RTOS_Task *task = (RTOS_Task *)params;
00193     TickType_t xLastWakeTime = xTaskGetTickCount();
00194     if (!task || task->taskFn) vTaskDelete(NULL);
00195     while (1) {
00196         if (task->active && task->taskFn != NULL) {
00197             task->taskFn(task->context); // Call the logical function
00198         }
00199         vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(task->period_ms));
00200     }
00201 }
00202 /***** FIN DE ARCHIVO *****/

```

4.23 km_rtos.h File Reference

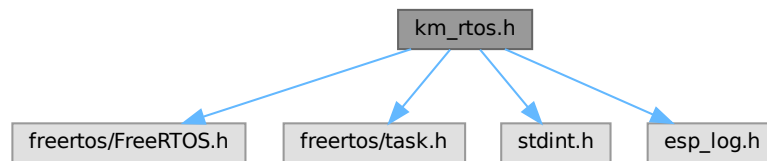
```

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

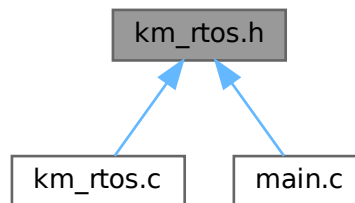
```

```
#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 represents 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, BaseType_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

<i>context</i>	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

<i>name</i>	Name of the task (used by FreeRTOS for debugging/logging)
<i>taskFn</i>	Function pointer to the task implementation (of type KM_RTOS_TaskFunction_t)
<i>context</i>	Pointer to a user-defined context or data structure passed to the task
<i>period_ms</i>	Task period in milliseconds (used for periodic scheduling)
<i>stackSize</i>	Stack size for the FreeRTOS task
<i>priority</i>	FreeRTOS task priority
<i>active</i>	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](#).

```
00063                                     {
00064
00065     RTOS_Task task;
00066
00067     task.name = name;
00068     task.taskFn = taskFn;
00069     task.context = context;
00070     task.period_ms = period_ms;
00071     task.stackSize = stackSize;
00072     task.priority = priority;
00073     task.active = active;
00074
00075     return task;
00076 }
```

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

<code>task</code>	The <code>RTOS_Task</code> 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`.

```

00078                                     {
00079
00080     // Buscar hueco libre en array y comprobar que no exista ya esa tarea
00081     for (uint8_t i = 0; i < RTOS_MAX_TASKS; i++) {
00082         // Hueco ocupado
00083         if (tasks[i].handle != NULL) continue;
00084
00085         tasks[i] = task;
00086         tasks[i].active = 1;
00087
00088         if (tasks[i].priority == 0) tasks[i].priority = RTOS_DEFAULT_PRIORITY;
00089         if (tasks[i].stackSize == 0) tasks[i].stackSize = RTOS_DEFAULT_STACK_SIZE;
00090         if (tasks[i].period_ms == 0) tasks[i].period_ms = RTOS_DEFAULT_PERIOD_MS;
00091
00092         BaseType_t result = xTaskCreate(
00093             KM_RTOS_TaskWrapper, // Wrapper
00094             task.name,
00095             task.stackSize,
00096             &tasks[i],           // Pointer to this task
00097             task.priority,
00098             &tasks[i].handle

```

```

00099         );
00100
00101         if (result != pdPASS) {
00102             memset(&tasks[i], 0, sizeof(RTOS_Task));
00103             return ESP_FAIL; // Error creating task
00104         }
00105
00106         return ESP_OK; // Success
00107     }
00108
00109     return ESP_FAIL; // Fail, no space in array
00110 }

```

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

<i>handle</i>	The FreeRTOS task handle whose priority should be changed.
<i>newPriority</i>	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`.

```

00181
00182
00183     int8_t index = KM_RTOS_FindTask(handle);
00184     if (index == -1) return ESP_FAIL;
00185
00186     vTaskPrioritySet(tasks[index].handle, newPriority);
00187
00188     // Actualizar informacion sobre tarea
00189     tasks[index].priority = newPriority;
00190
00191     return ESP_OK;
00192 }

```

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

<i>handle</i>	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-safe calling this while other tasks may concurrently access the `tasks` array.

Definition at line 113 of file [km_rtos.c](#).

```
00113                                     {
00114     // Search for task in array
00115
00116     int8_t index = KM_RTOS_FindTask(handle);
00117     if (index == -1) return ESP_FAIL;
00118
00119     if (tasks[index].handle == handle) {
00120         vTaskDelete(handle);
00121         memset(&tasks[index], 0, sizeof(RTOS_Task));
00122         return ESP_OK;
00123     }
00124
00125     // NO se ha encontrado la tarea
00126     return ESP_FAIL;
00127 }
```

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 `vTaskDelete`. 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

<i>handle</i>	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 }
```

4.23.3.8 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

<i>handle</i>	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`.

```
00146                                     {
00147
00148     int8_t index = KM_RTOS_FindTask(handle);
00149     if (index == -1) return 0;
00150
00151     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

<i>handle</i>	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.](#)

```

00001 /*****
00002  * @file    Km_rtos.h
00003  * @brief   Public API for the KM_RTOS task management library.
00004  *
00005  * This header provides the public interface for the KM_RTOS library, which

```

```

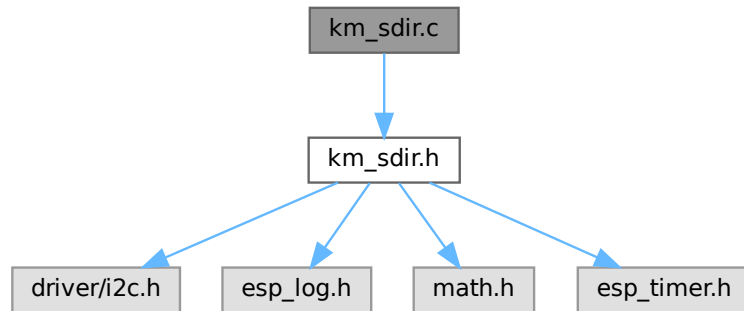
00006  * offers a structured and safe way to manage FreeRTOS tasks. Through this API,
00007  * users can create, delete, suspend, resume, restart tasks, and change task
00008  * priorities without directly manipulating internal data structures.
00009  *
00010  * The library ensures:
00011  * - Controlled access to a static internal task array.
00012  * - Consistent task creation with default parameters if needed.
00013  * - Safe periodic execution of tasks via the KM_RTOS_TaskWrapper.
00014  * - Simplified FreeRTOS task management while maintaining thread safety.
00015  *
00016  * Usage:
00017  * 1. Call KM_RTOS_Init() before creating any tasks.
00018  * 2. Use KM_RTOS_CreateTask / KM_RTOS_AddTask to define and add tasks.
00019  * 3. Manage tasks using KM_RTOS_DeleteTask, KM_RTOS_SuspendTask, KM_RTOS_ResumeTask,
00020  *    KM_RTOS_RestartTask, and KM_RTOS_ChangePriority.
00021  *
00022  * Author: Adrian Navarredonda Arizaleta
00023  * Date: 24-01-2026
00024  * Version: 1.0
00025  *****/
00026
00027 #ifndef KM_RTOS_H
00028 #define KM_RTOS_H
00029
00030 /***** INCLUDES *****/
00031 // Includes necesarios para la API pública
00032 #include "freertos/FreeRTOS.h"
00033 #include "freertos/task.h"
00034 #include <stdint.h>
00035 #include "esp_log.h" // Para log
00036
00037 /***** DEFINES PUBLICOS *****/
00038 // Constantes, flags o configuraciones visibles desde fuera de la librería
00039 #define RTOS_MAX_TASKS 10 // Número máximo de tareas que puede manejar la librería
00040
00041 typedef void (*KM_RTOS_TaskFunction_t)(void *context);
00042
00043 typedef struct {
00044     TaskHandle_t handle;
00045     const char *name;
00046     KM_RTOS_TaskFunction_t taskFn;
00047     void *context;
00048     uint32_t period_ms;
00049     uint16_t stackSize;
00050     UBaseType_t priority;
00051     uint8_t active;
00052 } RTOS_Task;
00053
00054 /***** VARIABLES PÚBLICAS *****/
00055 // Variables globales visibles (si realmente se necesitan)
00056
00057 // extern int ejemplo_variable_publica;
00058
00059 /***** FUNCIONES PÚBLICAS *****/
00060 void KM_RTOS_Init(void);
00061 void KM_RTOS_Destroy(void);
00062
00063 RTOS_Task KM_COMS_CreateTask(char *name, KM_RTOS_TaskFunction_t taskFn, void *context,
00064                             uint32_t period_ms, uint16_t stackSize, UBaseType_t priority, uint8_t active);
00065
00066 esp_err_t KM_RTOS_AddTask(RTOS_Task task);
00067
00068 esp_err_t KM_RTOS_DeleteTask(TaskHandle_t handle);
00069
00070 esp_err_t KM_RTOS_SuspendTask(TaskHandle_t handle);
00071
00072 esp_err_t KM_RTOS_ResumeTask(TaskHandle_t handle);
00073
00074 esp_err_t KM_RTOS_RestartTask(TaskHandle_t handle);
00075
00076 esp_err_t KM_RTOS_ChangePriority(TaskHandle_t handle, uint8_t newPriority);
00077
00078 #endif /* NOMBRE_LIBRERIA_H */
00079

```

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](#).

```
00132
00133     i2c_cmd_handle_t cmd = i2c_cmd_link_create();
00134     i2c_master_start(cmd);
00135     i2c_master_write_byte(cmd, (AS5600_ADDR << 1) | I2C_MASTER_WRITE, true);
00136     i2c_master_write_byte(cmd, reg, true);
00137     i2c_master_stop(cmd);
00138     esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
00139     i2c_cmd_link_delete(cmd);
00140
00141     if (ret != ESP_OK) {
00142         return 0;
00143     }
00144
00145     cmd = i2c_cmd_link_create();
00146     i2c_master_start(cmd);
00147     i2c_master_write_byte(cmd, (AS5600_ADDR << 1) | I2C_MASTER_READ, true);
00148     if (len > 1) {
00149         i2c_master_read(cmd, data, len - 1, I2C_MASTER_ACK);
00150     }
00151     i2c_master_read_byte(cmd, data + len - 1, I2C_MASTER_NACK);
00152     i2c_master_stop(cmd);
00153
00154     ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
00155     i2c_cmd_link_delete(cmd);
00156
00157     return (ret == ESP_OK) ? 1 : 0;
00158 }
```

4.25.2.9 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 }
```

4.25.2.10 KM_SDIR_setCenterOffset()

```
void KM_SDIR_setCenterOffset (
    sensor_struct * sensor,
    uint16_t offset )
```

Definition at line 124 of file [km_sdir.c](#).

```
00124                                     {
00125     sensor->centerOffset = offset;
00126     ESP_LOGI(TAG, "AS5600 center offset set to: %d", offset);
00127 }
```

4.26 km_sdir.c

[Go to the documentation of this file.](#)

```
00001 /*****
00002 * @file      km_sdir.c
00003 * @brief     Librería para usar el sensor de dirección (ESP-IDF)
00004 * @author    Adrian Navarredonda
00005 * @date      31-01-2026
00006 *****/
00007
00008 #include "km_sdir.h"
00009
00010 /***** MACROS PRIVADAS *****/
00011 #define TAG "KM_SDIR"
00012 #define I2C_MASTER_TIMEOUT_MS 1000
00013
00014 /***** DECLARACION FUNCIONES PRIVADAS *****/
00015 static int8_t KM_SDIR_ReadRegisters(sensor_struct *sensor, uint8_t reg, uint8_t* data, uint8_t len);
00016 static float KM_SDIR_ReadAngle(sensor_struct *sensor);
00017
00018 /***** FUNCIONES PÚBLICAS *****/
00019
00020 sensor_struct KM_SDIR_Init(int8_t max_error_count) {
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 }
00032
00033 // Inicializa I2C
00034 int8_t KM_SDIR_Begin(sensor_struct *sensor, gpio_num_t sda, gpio_num_t scl) {
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 }
00060
00061 // Leer valor crudo (0-4095)
00062 uint16_t KM_SDIR_ReadRaw(sensor_struct *sensor) {
```

```

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 }
00080
00081 // Leer ángulo en radianes
00082 float KM_SDIR_ReadAngleRadians(sensor_struct *sensor) {
00083     return KM_SDIR_ReadAngle(sensor);
00084 }
00085
00086 // Leer ángulo en grados
00087 float KM_SDIR_ReadAngleDegrees(sensor_struct *sensor) {
00088     return KM_SDIR_ReadAngle(sensor) * 180.0f / PI;
00089 }
00090
00091 // Verifica si el sensor está conectado
00092 int8_t KM_SDIR_isConnected(sensor_struct *sensor) {
00093     return sensor->connected && (sensor->errorCount < sensor->max_error_count);
00094 }
00095
00096 // Reset de I2C
00097 int8_t KM_SDIR_ResetI2C(sensor_struct *sensor) {
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 }
00122
00123 // Ajuste de offset
00124 void KM_SDIR_setCenterOffset(sensor_struct *sensor, uint16_t offset) {
00125     sensor->centerOffset = offset;
00126     ESP_LOGI(TAG, "AS5600 center offset set to: %d", offset);
00127 }
00128
00129 /***** FUNCIONES PRIVADAS *****/
00130
00131 // Leer registros I2C
00132 static int8_t KM_SDIR_ReadRegisters(sensor_struct *sensor, uint8_t reg, uint8_t* data, uint8_t len) {
00133     i2c_cmd_handle_t cmd = i2c_cmd_link_create();
00134     i2c_master_start(cmd);
00135     i2c_master_write_byte(cmd, (AS5600_ADDR << 1) | I2C_MASTER_WRITE, true);
00136     i2c_master_write_byte(cmd, reg, true);
00137     i2c_master_stop(cmd);
00138     esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
00139     i2c_cmd_link_delete(cmd);
00140
00141     if (ret != ESP_OK) {
00142         return 0;
00143     }
00144
00145     cmd = i2c_cmd_link_create();
00146     i2c_master_start(cmd);
00147     i2c_master_write_byte(cmd, (AS5600_ADDR << 1) | I2C_MASTER_READ, true);
00148     if (len > 1) {
00149         i2c_master_read(cmd, data, len - 1, I2C_MASTER_ACK);

```

```

00150     }
00151     i2c_master_read_byte(cmd, data + len - 1, I2C_MASTER_NACK);
00152     i2c_master_stop(cmd);
00153
00154     ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, pdMS_TO_TICKS(I2C_MASTER_TIMEOUT_MS));
00155     i2c_cmd_link_delete(cmd);
00156
00157     return (ret == ESP_OK) ? 1 : 0;
00158 }
00159
00160 // Devuelve ángulo en radianes (-PI a PI)
00161 static float KM_SDIR_ReadAngle(sensor_struct *sensor) {
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 }
00169
00170 /***** FIN DE ARCHIVO *****/

```

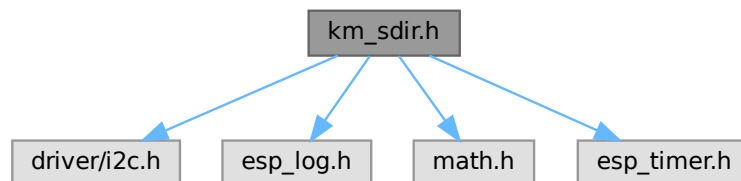
4.27 km_sdir.h File Reference

```

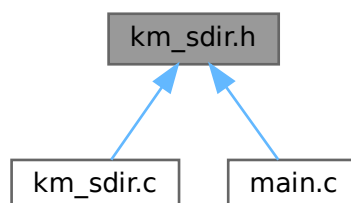
#include "driver/i2c.h"
#include "esp_log.h"
#include <math.h>
#include "esp_timer.h"

```

Include dependency graph for km_sdir.h:



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



Classes

- struct [sensor_struct](#)

Structure that represents 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`.

```
00040     {
00041         AS5600_ADDR = 0x36,
00042         AS5600_ANGLE_MSB = 0x0C,
00043         AS5600_ANGLE_LSB = 0x0D
00044     } connection_consts;
```

4.27.2.2 sensor_consts

enum `sensor_consts`

Enumerator

SENSOR_MIN	
SENSOR_MAX	
SENSOR_CENTER	

Definition at line 46 of file `km_sdir.h`.

```
00046     {
00047         SENSOR_MIN = 0,
00048         SENSOR_MAX = 4095,
00049         SENSOR_CENTER = 2048,
00050     } sensor_consts;
```

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`.

```
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         return sensor->connected;
00058     }
00059 }
```

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](#).

```
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 }
```

4.27.3.3 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 }
```

4.27.3.4 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 }
```

4.27.3.5 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 }
```

4.27.3.6 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
00027 typedef struct {
00028     uint8_t connected;
00029     uint8_t errorCount;
00030     uint8_t max_error_count;
00031     uint64_t lastReadTime;
00032     uint16_t lastRawValue;
00033     uint16_t centerOffset;
00034 } sensor_struct;
00035
00036 // Represents the connection constants
00037 typedef enum {
00038     AS5600_ADDR = 0x36,
00039     AS5600_ANGLE_MSB = 0x0C,
00040     AS5600_ANGLE_LSB = 0x0D
00041 } connection_consts;
00042
00043 typedef enum {
00044     SENSOR_MIN = 0,
00045     SENSOR_MAX = 4095,
00046     SENSOR_CENTER = 2048,
00047 } sensor_consts;
00048
00049 /***** VARIABLES PÚBLICAS *****/
00050 // Variables globales visibles (si realmente se necesitan)
00051
00052 const float MAX_RAD = PI;
00053
00054 // extern int ejemplo_variable_publica;
00055
00056 /***** FUNCIONES PÚBLICAS *****/
00057
00058 sensor_struct KM_SDIR_Init(uint8_t max_error_count);
00059
00060 // Initialize the sensor
00061 int8_t KM_SDIR_Begin(sensor_struct *sensor, gpio_num_t sdaPin, gpio_num_t sclPin);
00062
00063 // Read raw sensor value (0-4095)
00064 uint16_t KM_SDIR_ReadRaw(sensor_struct *sensor);
00065
00066 // Read angle in radians (-PI to PI)
```

```

00074 float KM_SDIR_ReadAngleRadians(sensor_struct *sensor);
00075
00076 // Read angle in degrees (-180 to 180)
00077 float KM_SDIR_ReadAngleDegrees(sensor_struct *sensor);
00078
00079 // Check if sensor is connected
00080 int8_t KM_SDIR_isConnected(sensor_struct *sensor);
00081
00082 // Reset I2C communication if errors occur
00083 int8_t KM_SDIR_ResetI2C(sensor_struct *sensor);
00084
00085 // Get center offset for calibration
00086 void KM_SDIR_setCenterOffset(sensor_struct *sensor, uint16_t offset);
00087
00088 #endif /* KM_SDIR_H */

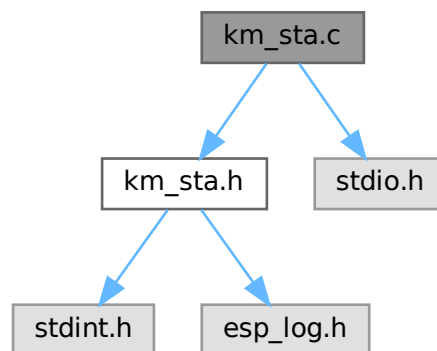
```

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.](#)

```

00001 /*****
00002  * @file    nombre_libreria.c
00003  * @brief   Implementación de la librería.
00004  *****/
00005
00006 #include "km_sta.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 // #define LIBRERIA_DEBUG 1
00015
00016 /***** VARIABLES PRIVADAS *****/
00017 // Variables globales internas (static)
00018
00019 /***** DECLARACION FUNCIONES PRIVADAS *****/
00020

```

```

00021
00022 /***** FUNCIONES PÚBLICAS *****/
00026 // int libreria_operacion(int valor) {
00027 //     ...
00028 // }
00029
00030 /***** FUNCIONES PRIVADAS *****/
00034 // static void funcion_privada(void);
00035
00036 /***** FIN DE ARCHIVO *****/
00037

```

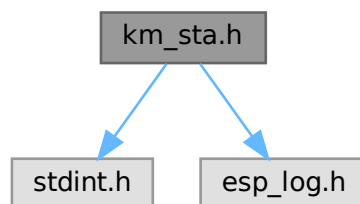
4.31 km_sta.h File Reference

```

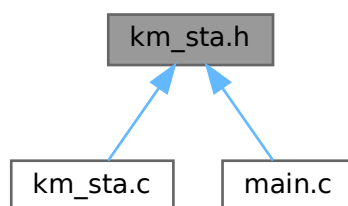
#include <stdint.h>
#include "esp_log.h"

```

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.](#)

```

00001 /*****
00002  * @file    nombre_libreria.h
00003  * @brief   Interfaz pública de la librería.
00004  * @author   Autor

```

```

00005  * @date    DD-MM-AAAA
00006  * @version 1.0
00007  *****/
00008
00009  #ifndef NOMBRE_LIBRERIA_H
00010  #define NOMBRE_LIBRERIA_H
00011
00012  /***** INCLUDES *****/
00013  // Includes necesarios para la API pública
00014  #include <stdint.h>
00015  #include "esp_log.h" // Para log
00016
00017  /***** DEFINES PÚBLICAS *****/
00018  // Constantes, flags o configuraciones visibles desde fuera de la librería
00019
00020  /***** TIPOS PÚBLICOS *****/
00021  // Estructuras, enums, typedefs públicos
00022
00023  /***** VARIABLES PÚBLICAS *****/
00024  // Variables globales visibles (si realmente se necesitan)
00025
00026  // extern int ejemplo_variable_publica;
00027
00028  /***** FUNCIONES PÚBLICAS *****/
00029  // int libreria_init(void);
00030
00031  // int libreria_operacion(int valor);
00032
00033  #endif /* NOMBRE_LIBRERIA_H */
00034
00035

```

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](#).

```
00121         {  
00122  
00123         // Habilitar logs por UART0 (USB) para debug  
00124         esp_log_level_set("*", ESP_LOG_INFO);  
00125  
00126         ESP_LOGI(TAG, "ESP iniciando...");  
00127  
00128         // Inicializa Bluetooth, Bluepad32, NVS, etc.  
00129         // init_bluetooth();  
00130  
00131         // Inicia todas las librerias que se necesitan  
00132         system_init();  
00133  
00134         // Creacion de toda las tareas de FreeRTOS  
00135         // Tareas de libreria de comunicaciones  
00136         // Tareas de libreria de maquina de estado  
00137  
00138         // Ejecutar loop de BTstack (bloquea dentro de app_main, pero otras tareas siguen)  
00139         //btstack_run_loop_execute();  
00140 }
```

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 communications 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_ProcessMsgs();
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
```

```

00010 // #include "esp_gatt_common_api.h" // Funciones GATT comunes
00011
00012 #include <btstack_port_esp32.h> // Puerto BTstack para ESP32
00013 #include <btstack_run_loop.h> // Loop de BTstack
00014 #include <btstack_stdio_esp32.h> // Consola BTstack
00015 #include <uni.h> // Core Bluepad32 (unijoysticle)
00016 // #include <platform/esp32/uni_platform_esp32.h> // Adaptación a ESP32
00017
00018 #include "esp_timer.h" // Temporizadores de alta resolución
00019 #include "esp_wifi.h" // Para WiFi
00020 #include "esp_event.h" // Event loop de ESP-IDF
00021 #include "esp_netif.h" // Configuración de interfaces de red
00022 #include "esp_sntp.h" // Para sincronizar tiempo
00023 #include "esp_system.h" // Información de sistema
00024
00025 // Librerías propias
00026 #include "km_act.h"
00027 #include "km_coms.h"
00028 #include "km_gamc.h"
00029 #include "km_pid.h"
00030 #include "km_rtos.h"
00031 #include "km_sdir.h"
00032 #include "km_sta.h"
00033 #include "km_gpio.h"
00034
00035 static const char *TAG = "MAIN";
00036
00037 // =====
00038 // Funciones auxiliares de Bluetooth
00039 // =====
00040 // void init_bluetooth(void) {
00041 //     esp_err_t ret;
00042 //
00043 //     // Inicializar NVS (necesario para BT/WiFi)
00044 //     ret = nvs_flash_init();
00045 //     if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00046 //         ESP_ERROR_CHECK(nvs_flash_erase());
00047 //         ret = nvs_flash_init();
00048 //     }
00049 //     ESP_ERROR_CHECK(ret);
00050 //
00051 //     // Inicializar controlador BT
00052 //     esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
00053 //     ESP_ERROR_CHECK(esp_bt_controller_init(&bt_cfg));
00054 //     ESP_ERROR_CHECK(esp_bt_controller_enable(ESP_BT_MODE_BTDM)); // BLE + Classic
00055 //
00056 //     // Inicializar Bluedroid
00057 //     ESP_ERROR_CHECK(esp_bluedroid_init());
00058 //     ESP_ERROR_CHECK(esp_bluedroid_enable());
00059 //
00060 //     // Inicializar consola BTstack
00061 //     btstack_stdio_init();
00062 //
00063 //     // Configurar plataforma ESP32 para Bluepad32
00064 //     uni_platform_set_custom(get_esp32_platform());
00065 //
00066 //     // Inicializar Bluepad32
00067 //     uni_init(0, NULL);
00068 //
00069 //     ESP_LOGI(TAG, "Bluetooth y Bluepad32 inicializados");
00070 // }
00071
00072 #define MAX_ERROR_COUNT_SDIR 10
00073
00074 void system_init(void) {
00075     // Initialize hardware
00076     if (KM_GPIO_Init() != ESP_OK)
00077         ESP_LOGE(TAG, "Error inicializando libreria gpio\n");
00078
00079     // Initialize I2C
00080     if (KM_GPIO_I2CInit() != ESP_OK)
00081         ESP_LOGE(TAG, "Error inicializando libreria i2c\n");
00082
00083     // Initialise tasks
00084     KM_RTOS_Init();
00085
00086     // Initialise communications on UART0 (USB to Orin)
00087     if (KM_COMS_Init(UART_NUM_0) != ESP_OK)
00088         ESP_LOGE(TAG, "Error inicializando libreria de comunicaciones");
00089
00090     // KM_SDIR_Init(MAX_ERROR_COUNT_SDIR);
00091
00092     // KM_ACT_Begin();
00093     // KM_PID_Init();
00094
00095     // 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_ProcessMsgs();
00112
00113         loop_count++;
00114         vTaskDelay(pdMS_TO_TICKS(100));
00115     }
00116 }
00117
00118 // =====
00119 // app_main - punto de entrada
00120 // =====
00121 void app_main(void) {
00122
00123     // Habilitar logs por UART0 (USB) para debug
00124     esp_log_level_set("*", ESP_LOG_INFO);
00125
00126     ESP_LOGI(TAG, "ESP iniciando...");
00127
00128     // Inicializa Bluetooth, Bluepad32, NVS, etc.
00129     // init_bluetooth();
00130
00131     // Inicia todas las librerías que se necesitan
00132     system_init();
00133
00134     // Creación de toda las tareas de FreeRTOS
00135     // Tareas de librería de comunicaciones
00136     // Tareas de librería de máquina de estado
00137
00138     // Ejecutar loop de BTstack (bloquea dentro de app_main, pero otras tareas siguen)
00139     //btstack_run_loop_execute();
00140 }
00141
00142 // ESP_LOGI(TAG, "Mensaje: %d", valor); → Información normal
00143
00144 // ESP_LOGW(TAG, "Mensaje: %s", cadena); → Advertencia
00145
00146 // ESP_LOGE(TAG, "Mensaje: %s", cadena); → Error
00147
00148 // ESP_LOGD(TAG, "Mensaje debug"); → Debug (solo si está habilitado)

```

Index

- active
 - RTOS_Task, [9](#)
- ACTUAL_ACCERELATION
 - km_objects.h, [71](#)
- ACTUAL_BRAKING
 - km_objects.h, [71](#)
- ACTUAL_SHUTDOWN
 - km_objects.h, [71](#)
- ACTUAL_SPEED
 - km_objects.h, [71](#)
- ACTUAL_STEERING
 - km_objects.h, [71](#)
- app_main
 - main.c, [121](#)
- AS5600_ADDR
 - km_sdir.h, [114](#)
- AS5600_ANGLE_LSB
 - km_sdir.h, [114](#)
- AS5600_ANGLE_MSB
 - km_sdir.h, [114](#)
- BUF_SIZE_RX
 - km_coms.h, [27](#)
- BUF_SIZE_TX
 - km_coms.h, [27](#)
- centerOffset
 - sensor_struct, [11](#)
- conection_constsans
 - km_sdir.h, [113](#)
- connected
 - sensor_struct, [11](#)
- context
 - RTOS_Task, [9](#)
- crc
 - km_coms_msg, [6](#)
- errorCount
 - sensor_struct, [11](#)
- ESP_ACT_BRAKING
 - km_coms.h, [28](#)
- ESP_ACT_SHUTDOWN
 - km_coms.h, [28](#)
- ESP_ACT_SPEED
 - km_coms.h, [28](#)
- ESP_ACT_STEERING
 - km_coms.h, [28](#)
- ESP_ACT_THROTTLE
 - km_coms.h, [28](#)
- ESP_COMPLETE
 - km_coms.h, [28](#)
- ESP_HEARTBEAT
 - km_coms.h, [28](#)
- ESP_MACHINE_STATE
 - km_coms.h, [28](#)
- ESP_MISION
 - km_coms.h, [28](#)
- handle
 - RTOS_Task, [9](#)
- I2C_MASTER_TIMEOUT_MS
 - km_sdir.c, [106](#)
- integral
 - PID_Controller, [7](#)
- integralMax
 - PID_Controller, [7](#)
- integralMin
 - PID_Controller, [7](#)
- kd
 - PID_Controller, [7](#)
- ki
 - PID_Controller, [7](#)
- km_coms.c, [13](#), [22](#)
 - KM_COMS_crc8, [14](#)
 - KM_COMS_Init, [15](#)
 - km_coms_mutex, [21](#)
 - KM_COMS_ProccessMsgs, [16](#)
 - KM_COMS_ProccessPayload, [17](#)
 - km_coms_ReceiveMsg, [19](#)
 - KM_COMS_SendMsg, [20](#)
 - km_coms_uart, [21](#)
 - KM_COMS_WAIT_SEM_AVAILABLE, [14](#)
 - rx_buffer, [21](#)
 - rx_buffer_len, [22](#)
- km_coms.h, [26](#), [33](#)
 - BUF_SIZE_RX, [27](#)
 - BUF_SIZE_TX, [27](#)
 - ESP_ACT_BRAKING, [28](#)
 - ESP_ACT_SHUTDOWN, [28](#)
 - ESP_ACT_SPEED, [28](#)
 - ESP_ACT_STEERING, [28](#)
 - ESP_ACT_THROTTLE, [28](#)
 - ESP_COMPLETE, [28](#)
 - ESP_HEARTBEAT, [28](#)
 - ESP_MACHINE_STATE, [28](#)
 - ESP_MISION, [28](#)
 - KM_COMS_Init, [29](#)
 - KM_COMS_MSG_MAX_LEN, [27](#)

- KM_COMS_ProcessMsgs, 30
- km_coms_ReceiveMsg, 31
- KM_COMS_RX_CHUNK, 28
- KM_COMS_SendMsg, 32
- KM_COMS_SOM, 28
- message_type_t, 28
- ORIN_COMPLETE, 28
- ORIN_HEARTBEAT, 28
- ORIN_MACHINE_STATE, 28
- ORIN_MISION, 28
- ORIN_SHUTDOWN, 28
- ORIN_TARG BRAKING, 28
- ORIN_TARG STEERING, 28
- ORIN_TARG_THROTTLE, 28
- KM_COMS_crc8
 - km_coms.c, 14
- KM_COMS_CreateTask
 - km_rtos.c, 85
 - km_rtos.h, 98
- KM_COMS_Init
 - km_coms.c, 15
 - km_coms.h, 29
- km_coms_msg, 5
 - crc, 6
 - len, 6
 - payload, 6
 - type, 6
- KM_COMS_MSG_MAX_LEN
 - km_coms.h, 27
- km_coms_mutex
 - km_coms.c, 21
- KM_COMS_ProcessMsgs
 - km_coms.c, 16
 - km_coms.h, 30
- KM_COMS_ProcessPayload
 - km_coms.c, 17
- km_coms_ReceiveMsg
 - km_coms.c, 19
 - km_coms.h, 31
- KM_COMS_RX_CHUNK
 - km_coms.h, 28
- KM_COMS_SendMsg
 - km_coms.c, 20
 - km_coms.h, 32
- KM_COMS_SOM
 - km_coms.h, 28
- km_coms_uart
 - km_coms.c, 21
- KM_COMS_WAIT_SEM_AVAILABLE
 - km_coms.c, 14
- km_gamc.c, 35
- km_gamc.h, 36, 37
- km_gpio.c, 37, 49
 - KM_GPIO_I2CInit, 39
 - KM_GPIO_Init, 39
 - KM_GPIO_ReadADC, 40
 - KM_GPIO_ReadPin, 41
 - KM_GPIO_WriteDAC, 41
- KM_GPIO_WritePinHigh, 41
- KM_GPIO_WritePinLow, 42
- KM_GPIO_WritePWM, 42
- pin_cmd_acc, 43
- pin_cmd_brake, 43
- pin_hydraulic_1, 43
- pin_hydraulic_2, 44
- pin_i2c_scl, 44
- pin_i2c_sda, 44
- pin_motor_hall_1, 45
- pin_motor_hall_2, 45
- pin_motor_hall_3, 45
- pin_pedal_acc, 46
- pin_pedal_brake, 46
- pin_pressure_1, 46
- pin_pressure_2, 47
- pin_pressure_3, 47
- pin_status_led, 47
- pin_steer_dir, 48
- pin_steer_pwm, 48
- uart_config, 48
- km_gpio.h, 52, 66
 - KM_GPIO_I2CInit, 58
 - KM_GPIO_Init, 58
 - KM_GPIO_ReadADC, 59
 - KM_GPIO_ReadPin, 60
 - KM_GPIO_WriteDAC, 60
 - KM_GPIO_WritePinHigh, 60
 - KM_GPIO_WritePinLow, 61
 - KM_GPIO_WritePWM, 61
 - LEDC_HIGH_SPEED_MODE, 54
 - LEDC_LOW_SPEED_MODE, 54
 - PIN_CMD_ACC, 55
 - pin_cmd_acc, 62
 - PIN_CMD_BRAKE, 55
 - pin_cmd_brake, 62
 - PIN_HYDRAULIC_1, 55
 - pin_hydraulic_1, 62
 - PIN_HYDRAULIC_2, 55
 - pin_hydraulic_2, 62
 - PIN_I2C_SCL, 55
 - pin_i2c_scl, 63
 - PIN_I2C_SDA, 55
 - pin_i2c_sda, 63
 - PIN_MOTOR_HALL_1, 55
 - pin_motor_hall_1, 63
 - PIN_MOTOR_HALL_2, 55
 - pin_motor_hall_2, 63
 - PIN_MOTOR_HALL_3, 56
 - pin_motor_hall_3, 63
 - PIN_ORIN_UART_RX, 56
 - PIN_ORIN_UART_TX, 56
 - PIN_PEDAL_ACC, 56
 - pin_pedal_acc, 64
 - PIN_PEDAL_BRAKE, 56
 - pin_pedal_brake, 64
 - PIN_PRESSURE_1, 56
 - pin_pressure_1, 64

- PIN_PRESSURE_2, 56
- pin_pressure_2, 64
- PIN_PRESSURE_3, 56
- pin_pressure_3, 64
- PIN_STATUS_LED, 57
- pin_status_led, 65
- PIN_STEER_DIR, 57
- pin_steer_dir, 65
- PIN_STEER_PWM, 57
- pin_steer_pwm, 65
- PIN_USB_UART_RX, 57
- PIN_USB_UART_TX, 57
- uart_config, 65
- KM_GPIO_I2CInit
 - km_gpio.c, 39
 - km_gpio.h, 58
- KM_GPIO_Init
 - km_gpio.c, 39
 - km_gpio.h, 58
- KM_GPIO_ReadADC
 - km_gpio.c, 40
 - km_gpio.h, 59
- KM_GPIO_ReadPin
 - km_gpio.c, 41
 - km_gpio.h, 60
- KM_GPIO_WriteDAC
 - km_gpio.c, 41
 - km_gpio.h, 60
- KM_GPIO_WritePinHigh
 - km_gpio.c, 41
 - km_gpio.h, 60
- KM_GPIO_WritePinLow
 - km_gpio.c, 42
 - km_gpio.h, 61
- KM_GPIO_WritePWM
 - km_gpio.c, 42
 - km_gpio.h, 61
- KM_OBJ_GetObjectValue
 - km_objects.c, 68
 - km_objects.h, 72
- KM_OBJ_LAST
 - km_objects.h, 71
- KM_OBJ_SetObjectValue
 - km_objects.c, 69
 - km_objects.h, 72
- km_objects.c, 67, 69
 - KM_OBJ_GetObjectValue, 68
 - KM_OBJ_SetObjectValue, 69
 - km_objects_values, 69
- km_objects.h, 70, 73
 - ACTUAL_ACCERELATION, 71
 - ACTUAL BRAKING, 71
 - ACTUAL_SHUTDOWN, 71
 - ACTUAL_SPEED, 71
 - ACTUAL_STEERING, 71
 - KM_OBJ_GetObjectValue, 72
 - KM_OBJ_LAST, 71
 - KM_OBJ_SetObjectValue, 72
- km_objects_t, 71
- MACHINE_STATE_ESP, 71
- MACHINE_STATE_ORIN, 71
- MISION_ESP, 71
- MISION_ORIN, 71
- OBJECT_VALUE_ERROR, 71
- SHUTDOWN_ORIN, 71
- TARGET BRAKING, 71
- TARGET_STEERING, 71
- TARGET_THROTTLE, 71
- km_objects_t
 - km_objects.h, 71
- km_objects_values
 - km_objects.c, 69
- km_pid.c, 74, 77
 - KM_PID_Calculate, 74
 - KM_PID_GetIntegral, 75
 - KM_PID_GetTunings, 75
 - KM_PID_Init, 75
 - KM_PID_Reset, 76
 - KM_PID_SetIntegralLimits, 76
 - KM_PID_SetOutputLimits, 76
 - KM_PID_SetTunings, 76
- km_pid.h, 78, 82
 - KM_PID_Calculate, 79
 - KM_PID_GetIntegral, 80
 - KM_PID_GetTunings, 80
 - KM_PID_Init, 80
 - KM_PID_Reset, 81
 - KM_PID_SetIntegralLimits, 81
 - KM_PID_SetOutputLimits, 81
 - KM_PID_SetTunings, 81
- KM_PID_Calculate
 - km_pid.c, 74
 - km_pid.h, 79
- KM_PID_GetIntegral
 - km_pid.c, 75
 - km_pid.h, 80
- KM_PID_GetTunings
 - km_pid.c, 75
 - km_pid.h, 80
- KM_PID_Init
 - km_pid.c, 75
 - km_pid.h, 80
- KM_PID_Reset
 - km_pid.c, 76
 - km_pid.h, 81
- KM_PID_SetIntegralLimits
 - km_pid.c, 76
 - km_pid.h, 81
- KM_PID_SetOutputLimits
 - km_pid.c, 76
 - km_pid.h, 81
- KM_PID_SetTunings
 - km_pid.c, 76
 - km_pid.h, 81
- km_rtos.c, 82, 93
 - KM_COMS_CreateTask, 85

- KM_RTOS_AddTask, [85](#)
- KM_RTOS_ChangePriority, [87](#)
- KM_RTOS_DeleteTask, [87](#)
- KM_RTOS_Destroy, [88](#)
- KM_RTOS_FindTask, [89](#)
- KM_RTOS_Init, [89](#)
- KM_RTOS_RestartTask, [90](#)
- KM_RTOS_ResumeTask, [90](#)
- KM_RTOS_SuspendTask, [91](#)
- KM_RTOS_TaskWrapper, [92](#)
- RTOS_DEFAULT_PERIOD_MS, [84](#)
- RTOS_DEFAULT_PRIORITY, [84](#)
- RTOS_DEFAULT_STACK_SIZE, [84](#)
- tasks, [93](#)
- km_rtos.h, [95](#), [104](#)
 - KM_COMS_CreateTask, [98](#)
 - KM_RTOS_AddTask, [98](#)
 - KM_RTOS_ChangePriority, [100](#)
 - KM_RTOS_DeleteTask, [100](#)
 - KM_RTOS_Destroy, [101](#)
 - KM_RTOS_Init, [102](#)
 - KM_RTOS_RestartTask, [102](#)
 - KM_RTOS_ResumeTask, [103](#)
 - KM_RTOS_SuspendTask, [104](#)
 - KM_RTOS_TaskFunction_t, [97](#)
 - RTOS_MAX_TASKS, [97](#)
- KM_RTOS_AddTask
 - km_rtos.c, [85](#)
 - km_rtos.h, [98](#)
- KM_RTOS_ChangePriority
 - km_rtos.c, [87](#)
 - km_rtos.h, [100](#)
- KM_RTOS_DeleteTask
 - km_rtos.c, [87](#)
 - km_rtos.h, [100](#)
- KM_RTOS_Destroy
 - km_rtos.c, [88](#)
 - km_rtos.h, [101](#)
- KM_RTOS_FindTask
 - km_rtos.c, [89](#)
- KM_RTOS_Init
 - km_rtos.c, [89](#)
 - km_rtos.h, [102](#)
- KM_RTOS_RestartTask
 - km_rtos.c, [90](#)
 - km_rtos.h, [102](#)
- KM_RTOS_ResumeTask
 - km_rtos.c, [90](#)
 - km_rtos.h, [103](#)
- KM_RTOS_SuspendTask
 - km_rtos.c, [91](#)
 - km_rtos.h, [104](#)
- KM_RTOS_TaskFunction_t
 - km_rtos.h, [97](#)
- KM_RTOS_TaskWrapper
 - km_rtos.c, [92](#)
- km_sdir.c, [106](#), [110](#)
 - I2C_MASTER_TIMEOUT_MS, [106](#)
- KM_SDIR_Begin, [107](#)
- KM_SDIR_Init, [107](#)
- KM_SDIR_isConnected, [107](#)
- KM_SDIR_ReadAngle, [108](#)
- KM_SDIR_ReadAngleDegrees, [108](#)
- KM_SDIR_ReadAngleRadians, [108](#)
- KM_SDIR_ReadRaw, [108](#)
- KM_SDIR_ReadRegisters, [108](#)
- KM_SDIR_ResetI2C, [109](#)
- KM_SDIR_setCenterOffset, [109](#)
- TAG, [106](#)
- km_sdir.h, [112](#), [117](#)
 - AS5600_ADDR, [114](#)
 - AS5600_ANGLE_LSB, [114](#)
 - AS5600_ANGLE_MSB, [114](#)
 - conection_constans, [113](#)
 - KM_SDIR_Begin, [114](#)
 - KM_SDIR_Init, [114](#)
 - KM_SDIR_isConnected, [115](#)
 - KM_SDIR_ReadAngleDegrees, [115](#)
 - KM_SDIR_ReadAngleRadians, [115](#)
 - KM_SDIR_ReadRaw, [115](#)
 - KM_SDIR_ResetI2C, [116](#)
 - KM_SDIR_setCenterOffset, [116](#)
 - MAX_RAD, [117](#)
 - PI, [113](#)
 - SENSOR_CENTER, [114](#)
 - sensor_constans, [114](#)
 - SENSOR_MAX, [114](#)
 - SENSOR_MIN, [114](#)
- KM_SDIR_Begin
 - km_sdir.c, [107](#)
 - km_sdir.h, [114](#)
- KM_SDIR_Init
 - km_sdir.c, [107](#)
 - km_sdir.h, [114](#)
- KM_SDIR_isConnected
 - km_sdir.c, [107](#)
 - km_sdir.h, [115](#)
- KM_SDIR_ReadAngle
 - km_sdir.c, [108](#)
- KM_SDIR_ReadAngleDegrees
 - km_sdir.c, [108](#)
 - km_sdir.h, [115](#)
- KM_SDIR_ReadAngleRadians
 - km_sdir.c, [108](#)
 - km_sdir.h, [115](#)
- KM_SDIR_ReadRaw
 - km_sdir.c, [108](#)
 - km_sdir.h, [115](#)
- KM_SDIR_ReadRegisters
 - km_sdir.c, [108](#)
- KM_SDIR_ResetI2C
 - km_sdir.c, [109](#)
 - km_sdir.h, [116](#)
- KM_SDIR_setCenterOffset
 - km_sdir.c, [109](#)
 - km_sdir.h, [116](#)

km_sta.c, [118](#)
km_sta.h, [119](#)
kp
 PID_Controller, [8](#)

lastError
 PID_Controller, [8](#)
lastRawValue
 sensor_struct, [11](#)
lastReadTime
 sensor_struct, [11](#)
lastTime
 PID_Controller, [8](#)
LEDC_HIGH_SPEED_MODE
 km_gpio.h, [54](#)
LEDC_LOW_SPEED_MODE
 km_gpio.h, [54](#)
len
 km_coms_msg, [6](#)

MACHINE_STATE_ESP
 km_objects.h, [71](#)
MACHINE_STATE_ORIN
 km_objects.h, [71](#)
main.c, [120](#), [122](#)
 app_main, [121](#)
 MAX_ERROR_COUNT_SDIR, [121](#)
 system_init, [121](#)
 TAG, [122](#)
max_error_count
 sensor_struct, [11](#)
MAX_ERROR_COUNT_SDIR
 main.c, [121](#)
MAX_RAD
 km_sdir.h, [117](#)
message_type_t
 km_coms.h, [28](#)
MISION_ESP
 km_objects.h, [71](#)
MISION_ORIN
 km_objects.h, [71](#)

name
 RTOS_Task, [9](#)

OBJECT_VALUE_ERROR
 km_objects.h, [71](#)
ORIN_COMPLETE
 km_coms.h, [28](#)
ORIN_HEARTBEAT
 km_coms.h, [28](#)
ORIN_MACHINE_STATE
 km_coms.h, [28](#)
ORIN_MISION
 km_coms.h, [28](#)
ORIN_SHUTDOWN
 km_coms.h, [28](#)
ORIN_TARG BRAKING
 km_coms.h, [28](#)

ORIN_TARG_STEERING
 km_coms.h, [28](#)
ORIN_TARG_THROTTLE
 km_coms.h, [28](#)
outputMax
 PID_Controller, [8](#)
outputMin
 PID_Controller, [8](#)

payload
 km_coms_msg, [6](#)
period_ms
 RTOS_Task, [10](#)
PI
 km_sdir.h, [113](#)
PID_Controller, [6](#)
 integral, [7](#)
 integralMax, [7](#)
 integralMin, [7](#)
 kd, [7](#)
 ki, [7](#)
 kp, [8](#)
 lastError, [8](#)
 lastTime, [8](#)
 outputMax, [8](#)
 outputMin, [8](#)
PIN_CMD_ACC
 km_gpio.h, [55](#)
pin_cmd_acc
 km_gpio.c, [43](#)
 km_gpio.h, [62](#)
PIN_CMD_BRAKE
 km_gpio.h, [55](#)
pin_cmd_brake
 km_gpio.c, [43](#)
 km_gpio.h, [62](#)
PIN_HYDRAULIC_1
 km_gpio.h, [55](#)
pin_hydraulic_1
 km_gpio.c, [43](#)
 km_gpio.h, [62](#)
PIN_HYDRAULIC_2
 km_gpio.h, [55](#)
pin_hydraulic_2
 km_gpio.c, [44](#)
 km_gpio.h, [62](#)
PIN_I2C_SCL
 km_gpio.h, [55](#)
pin_i2c_scl
 km_gpio.c, [44](#)
 km_gpio.h, [63](#)
PIN_I2C_SDA
 km_gpio.h, [55](#)
pin_i2c_sda
 km_gpio.c, [44](#)
 km_gpio.h, [63](#)
PIN_MOTOR_HALL_1
 km_gpio.h, [55](#)
pin_motor_hall_1

- km_gpio.c, [45](#)
- km_gpio.h, [63](#)
- PIN_MOTOR_HALL_2
 - km_gpio.h, [55](#)
- pin_motor_hall_2
 - km_gpio.c, [45](#)
 - km_gpio.h, [63](#)
- PIN_MOTOR_HALL_3
 - km_gpio.h, [56](#)
- pin_motor_hall_3
 - km_gpio.c, [45](#)
 - km_gpio.h, [63](#)
- PIN_ORIN_UART_RX
 - km_gpio.h, [56](#)
- PIN_ORIN_UART_TX
 - km_gpio.h, [56](#)
- PIN_PEDAL_ACC
 - km_gpio.h, [56](#)
- pin_pedal_acc
 - km_gpio.c, [46](#)
 - km_gpio.h, [64](#)
- PIN_PEDAL_BRAKE
 - km_gpio.h, [56](#)
- pin_pedal_brake
 - km_gpio.c, [46](#)
 - km_gpio.h, [64](#)
- PIN_PRESSURE_1
 - km_gpio.h, [56](#)
- pin_pressure_1
 - km_gpio.c, [46](#)
 - km_gpio.h, [64](#)
- PIN_PRESSURE_2
 - km_gpio.h, [56](#)
- pin_pressure_2
 - km_gpio.c, [47](#)
 - km_gpio.h, [64](#)
- PIN_PRESSURE_3
 - km_gpio.h, [56](#)
- pin_pressure_3
 - km_gpio.c, [47](#)
 - km_gpio.h, [64](#)
- PIN_STATUS_LED
 - km_gpio.h, [57](#)
- pin_status_led
 - km_gpio.c, [47](#)
 - km_gpio.h, [65](#)
- PIN_STEER_DIR
 - km_gpio.h, [57](#)
- pin_steer_dir
 - km_gpio.c, [48](#)
 - km_gpio.h, [65](#)
- PIN_STEER_PWM
 - km_gpio.h, [57](#)
- pin_steer_pwm
 - km_gpio.c, [48](#)
 - km_gpio.h, [65](#)
- PIN_USB_UART_RX
 - km_gpio.h, [57](#)
- PIN_USB_UART_TX
 - km_gpio.h, [57](#)
- priority
 - RTOS_Task, [10](#)
- RTOS_DEFAULT_PERIOD_MS
 - km_rtos.c, [84](#)
- RTOS_DEFAULT_PRIORITY
 - km_rtos.c, [84](#)
- RTOS_DEFAULT_STACK_SIZE
 - km_rtos.c, [84](#)
- RTOS_MAX_TASKS
 - km_rtos.h, [97](#)
- RTOS_Task, [9](#)
 - active, [9](#)
 - context, [9](#)
 - handle, [9](#)
 - name, [9](#)
 - period_ms, [10](#)
 - priority, [10](#)
 - stackSize, [10](#)
 - taskFn, [10](#)
- rx_buffer
 - km_coms.c, [21](#)
- rx_buffer_len
 - km_coms.c, [22](#)
- SENSOR_CENTER
 - km_sdir.h, [114](#)
- sensor_consts
 - km_sdir.h, [114](#)
- SENSOR_MAX
 - km_sdir.h, [114](#)
- SENSOR_MIN
 - km_sdir.h, [114](#)
- sensor_struct, [10](#)
 - centerOffset, [11](#)
 - connected, [11](#)
 - errorCount, [11](#)
 - lastRawValue, [11](#)
 - lastReadTime, [11](#)
 - max_error_count, [11](#)
- SHUTDOWN_ORIN
 - km_objects.h, [71](#)
- stackSize
 - RTOS_Task, [10](#)
- system_init
 - main.c, [121](#)
- TAG
 - km_sdir.c, [106](#)
 - main.c, [122](#)
- TARGET BRAKING
 - km_objects.h, [71](#)
- TARGET STEERING
 - km_objects.h, [71](#)
- TARGET_THROTTLE
 - km_objects.h, [71](#)
- taskFn

- RTOS_Task, [10](#)
- tasks
 - km_rtos.c, [93](#)
- type
 - km_coms_msg, [6](#)
- uart_config
 - km_gpio.c, [48](#)
 - km_gpio.h, [65](#)