Prática 2

GPIO

Criação de tarefas

Gerenciamento de Filas

GPIO

- The ESP32 chip features 34 physical GPIO pads.
- The I/O GPIO pads are 0-19, 21-23, 25-27, 32- 39. GPIO pads 34 - 39 are input Only.

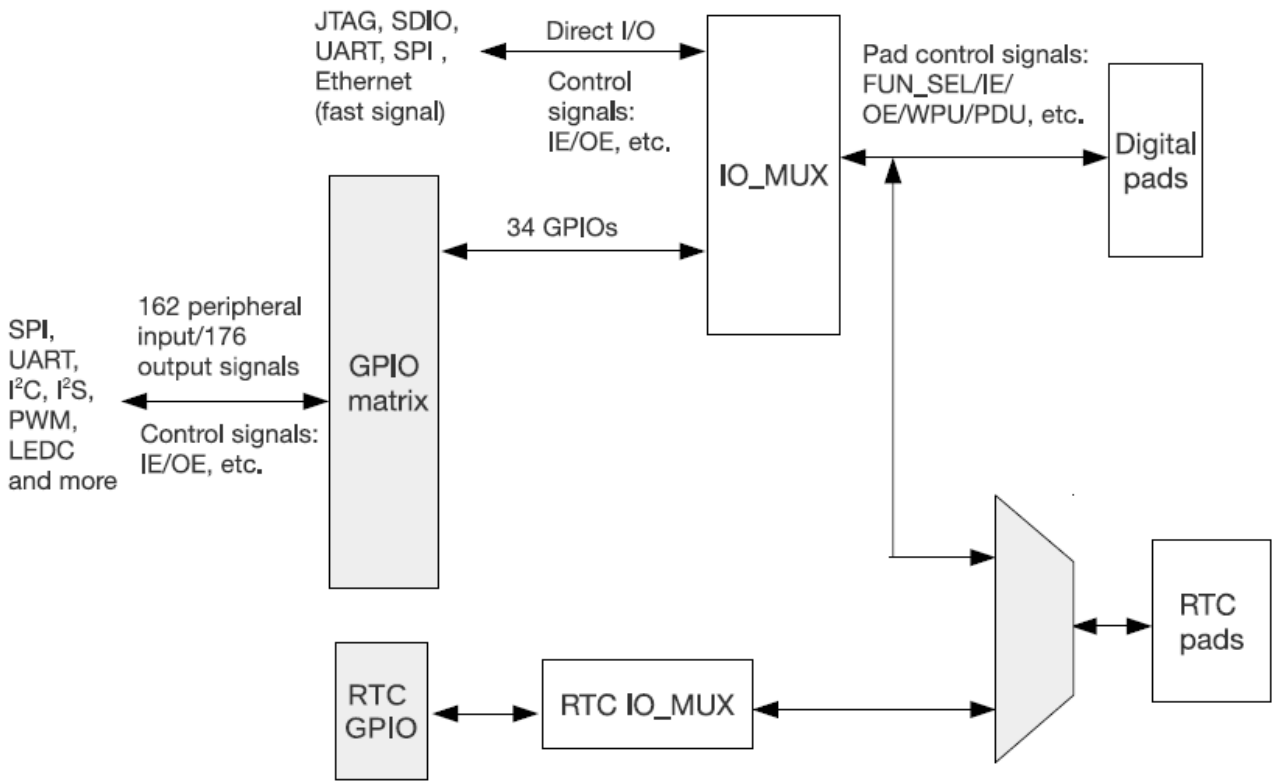


Table 4-3. IO_MUX Pad Summary

GPIO	Pad Name	Function 1	Function 2	Function 3	Function 4	Function 5	Function 6	Reset	Notes
0	GPIO0	GPIO0	CLK_OUT1	GPIO0	-	-	EMAC_TX_CLK	3	R
1	U0TXD	U0TXD	CLK_OUT3	GPIO1	-	-	EMAC_RXD2	3	-
2	GPIO2	GPIO2	HSPIWP	GPIO2	HS2_DATA0	SD_DATA0	-	2	R
3	U0RXD	U0RXD	CLK_OUT2	GPIO3	-	-	-	3	-
4	GPIO4	GPIO4	HSPIHD	GPIO4	HS2_DATA1	SD_DATA1	EMAC_TX_ER	2	R
5	GPIO5	GPIO5	VSPICS0	GPIO5	HS1_DATA6	-	EMAC_RX_CLK	3	-
6	SD_CLK	SD_CLK	SPICLK	GPIO6	HS1_CLK	U1CTS	-	3	-
7	SD_DATA_0	SD_DATA0	SPIQ	GPIO7	HS1_DATA0	U2RTS	-	3	-
8	SD_DATA_1	SD_DATA1	SPID	GPIO8	HS1_DATA1	U2CTS	-	3	-
9	SD_DATA_2	SD_DATA2	SPIHD	GPIO9	HS1_DATA2	U1RXD	-	3	-
10	SD_DATA_3	SD_DATA3	SPIWP	GPIO10	HS1_DATA3	U1TXD	-	3	-
11	SD_CMD	SD_CMD	SPICS0	GPIO11	HS1_CMD	U1RTS	-	3	-
12	MTDI	MTDI	HSPIQ	GPIO12	HS2_DATA2	SD_DATA2	EMAC_TXD3	2	R
13	MTCK	MTCK	HSPID	GPIO13	HS2_DATA3	SD_DATA3	EMAC_RX_ER	2	R
14	MTMS	MTMS	HSPICLK	GPIO14	HS2_CLK	SD_CLK	EMAC_TXD2	3	R
15	MTDO	MTDO	HSPICS0	GPIO15	HS2_CMD	SD_CMD	EMAC_RXD3	3	R
16	GPIO16	GPIO16	-	GPIO16	HS1_DATA4	U2RXD	EMAC_CLK_OUT	1	-
17	GPIO17	GPIO17	-	GPIO17	HS1_DATA5	U2TXD	EMAC_CLK_180	1	-
18	GPIO18	GPIO18	VSPICLK	GPIO18	HS1_DATA7	-	-	1	-
19	GPIO19	GPIO19	VSPIQ	GPIO19	U0CTS	-	EMAC_TXD0	1	-
21	GPIO21	GPIO21	VSPIHD	GPIO21	-	-	EMAC_TX_EN	1	-
22	GPIO22	GPIO22	VSPIWP	GPIO22	U0RTS	-	EMAC_TXD1	1	-
23	GPIO23	GPIO23	VSPID	GPIO23	HS1_STROBE	-	-	1	-
25	GPIO25	GPIO25	-	GPIO25	-	-	EMAC_RXD0	0	R
26	GPIO26	GPIO26	-	GPIO26	-	-	EMAC_RXD1	0	R
27	GPIO27	GPIO27	-	GPIO27	-	-	EMAC_RX_DV	0	R
32	32K_XP	GPIO32	-	GPIO32	-	-	-	0	R
33	32K_XN	GPIO33	-	GPIO33	-	-	-	0	R
34	VDET_1	GPIO34	-	GPIO34	-	-	-	0	R, I
35	VDET_2	GPIO35	-	GPIO35	-	-	-	0	R, I
36	SENSOR_VP	GPIO36	-	GPIO36	-	-	-	0	R, I
37	SENSOR_CAPP	GPIO37	-	GPIO37	-	-	-	0	R, I
38	SENSOR_CAPN	GPIO38	-	GPIO38	-	-	-	0	R, I
39	SENSOR_VN	GPIO39	-	GPIO39	-	-	-	0	R, I

Configurando GPIO

struct **gpio_config_t**

Configuration parameters of GPIO pad for gpio_config function.

Public Members

uint64_t **pin_bit_mask**

GPIO pin: set with bit mask, each bit maps to a GPIO

gpio_mode_t **mode**

GPIO mode: set input/output mode

gpio_pullup_t **pull_up_en**

GPIO pull-up

gpio_pulldown_t **pull_down_en**

GPIO pull-down

gpio_int_type_t **intr_type**

GPIO interrupt type

enum **gpio_mode_t**

Values:

enumerator **GPIO_MODE_DISABLE**

GPIO mode : disable input and output

enumerator **GPIO_MODE_INPUT**

GPIO mode : input only

enumerator **GPIO_MODE_OUTPUT**

GPIO mode : output only mode

enumerator **GPIO_MODE_OUTPUT_OD**

GPIO mode : output only with open-drain mode

enumerator **GPIO_MODE_INPUT_OUTPUT_OD**

GPIO mode : output and input with open-drain mode

enumerator **GPIO_MODE_INPUT_OUTPUT**

GPIO mode : output and input mode

enum **gpio_pullup_t**

Values:

enumerator **GPIO_PULLUP_DISABLE**

Disable GPIO pull-up resistor

enumerator **GPIO_PULLUP_ENABLE**

Enable GPIO pull-up resistor

enum **gpio_pulldown_t**

Values:

enumerator **GPIO_PULLDOWN_DISABLE**

Disable GPIO pull-down resistor

enumerator **GPIO_PULLDOWN_ENABLE**

Enable GPIO pull-down resistor

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/gpio.html

struct **gpio_config_t**

Configuration parameters of GPIO pad for gpio_config function.

Public Members

uint64_t **pin_bit_mask**

GPIO pin: set with bit mask, each bit maps to a GPIO

gpio_mode_t **mode**

GPIO mode: set input/output mode

gpio_pullup_t **pull_up_en**

GPIO pull-up

gpio_pulldown_t **pull_down_en**

GPIO pull-down

gpio_int_type_t **intr_type**

GPIO interrupt type

enum **gpio_int_type_t**

Values:

enumerator **GPIO_INTR_DISABLE**

Disable GPIO interrupt

enumerator **GPIO_INTR_POSEDGE**

GPIO interrupt type : rising edge

enumerator **GPIO_INTR_NEGEDGE**

GPIO interrupt type : falling edge

enumerator **GPIO_INTR_ANYEDGE**

GPIO interrupt type : both rising and falling edge

enumerator **GPIO_INTR_LOW_LEVEL**

GPIO interrupt type : input low level trigger

enumerator **GPIO_INTR_HIGH_LEVEL**

GPIO interrupt type : input high level trigger

enumerator **GPIO_INTR_MAX**

Exemplo

```
// zero-initialize the config structure.
gpio_config_t io_conf = {};
// disable interrupt
io_conf.intr_type = GPIO_INTR_DISABLE;
// set as output mode
io_conf.mode = GPIO_MODE_OUTPUT;
// bit mask of the pins that you want to set,e.g.GPIO18/19
io_conf.pin_bit_mask = GPIO_OUTPUT_PIN_SEL;
// disable pull-down mode
io_conf.pull_down_en = 0;
// disable pull-up mode
io_conf.pull_up_en = 0;
// configure GPIO with the given settings
gpio_config(&io_conf);
```

Tasks - exemplo

- Tasks are implemented as C functions. The prototype must return void and take a void pointer parameter.

```
static void gpio_task_example(void* arg)
```

- Each task is a small program in its own right. It has an entry point, will normally run forever within an infinite loop, and will not exit.
- A single task function definition can be used to create any number of tasks—each created task being a separate execution instance, with its own stack and its own copy of any automatic (stack) variables defined within the task itself.

Tasks

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

pvTaskCode	Tasks are simply C functions that never exit and, as such, are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function that implements the task (in effect, just the name of the function).

https://www.freertos.org/a00019.html

Tasks

pcName	A descriptive name for the task. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a task by a human readable name is much simpler than attempting to identify it by its handle.
usStackDepth	Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack.
pvParameters	Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task.

Tasks

uxPriority	Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority.
pxCreatedTask	pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

Tasks - Example

```
static void gpio_task_example(void *arg)
{
    uint32_t io_num;
    for (;;)
    {
        if (xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY))
        {
            printf("GPIO[%d] intr, val: %d\n", io_num, gpio_get_level(io_num));
        }
    }
}
```

```
xTaskCreate(gpio_task_example, "gpio_task_example", 2048, NULL, 10, NULL);
```

Queue

‘Queues’ provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism.

A queue can hold a finite number of fixed size data items.

Queues are normally used as First In First Out (FIFO) buffers

https://www.freertos.org/a00018.html

Using a Queue

A queue must be explicitly created before it can be used.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Parameter Name	Description
uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size in bytes of each data item that can be stored in the queue.
Return Value	If NULL is returned, then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area. A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

Queue

```
BaseType_t xQueueSendFromISR
          (
              QueueHandle_t xQueue,
              const void *pvItemToQueue,
              BaseType_t *pxHigherPriorityTaskWoken
          );
```

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

```
BaseType_t xQueueReceive(
                            QueueHandle_t xQueue,
                            void *pvBuffer,
                            TickType_t xTicksToWait
                          );
```

Parameters:

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. Setting xTicksToWait to 0 will cause the function to return immediately if the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Queue

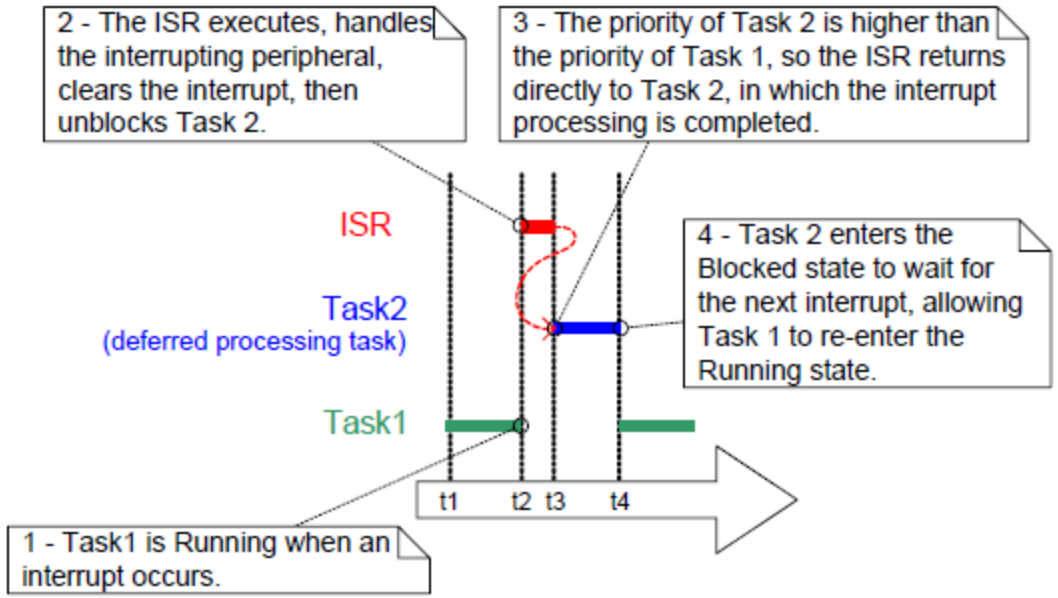
```
static xQueueHandle gpio_evt_queue = NULL; //variável global

gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t)); //main()

static void IRAM_ATTR gpio_isr_handle(void *arg)
{
    uint32_t gpio_num = (uint32_t)arg;
    xQueueSendFromISR(gpio_evt_queue, &gpio_num, NULL);
}
static void gpio_task_example(void *arg)
{
    uint32_t io_num;
    for (;;)
    {
        if (xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY))
        {
            printf("GPIO[%d] intr, val: %d\n", io_num, gpio_get_level(io_num));
        }
    }
}
```

Interrupção

```
static void IRAM_ATTR gpio_isr_handler(void *arg)
{
    uint32_t gpio_num = (uint32_t)arg;
    xQueueSendFromISR(gpio_evt_queue, &gpio_num, NULL);
}
```

```
// install gpio isr service
gpio_install_isr_service(ESP_INTR_FLAG_DEFAULT);
// hook isr handler for specific gpio pin
gpio_isr_handler_add(GPIO_INPUT_IO_0, gpio_isr_handler, (void *)GPIO_INPUT_IO_0);
// hook isr handler for specific gpio pin
gpio_isr_handler_add(GPIO_INPUT_IO_1, gpio_isr_handler, (void *)GPIO_INPUT_IO_1);
```

Interrupção



Referências

- https://freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

- https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf