Anexo 1 - ADC

Anexo ADC. API Reference > Analog To Digital Converter

Functions

esp_err_t **adc1_pad_get_io_num**(adc1_channel_t *channel*, gpio_num_t * *gpio_num*)

Get the GPIO number of a specific ADC1 channel.

Return

- ESP_OK if success
- ESP_ERR_INVALID_ARG if channel not valid

Parameters

- `channel` : Channel to get the GPIO number
- `gpio_num` : output buffer to hold the GPIO number

esp_err_t **adc1_config_width**(adc_bits_width_t *width_bit*)

Configure ADC1 capture width, meanwhile enable output invert for ADC1. The configuration is for all channels of ADC1.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `width_bit` : Bit capture width for ADC1

esp_err_t **adc_set_data_width**(adc_unit_t *adc_unit*, adc_bits_width_t *width_bit*)

Configure ADC capture width.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `adc_unit` : ADC unit index
- `width_bit` : Bit capture width for ADC unit.

esp_err_t **adc1_config_channel_atten**(adc1_channel_t *channel*, adc_atten_t *atten*)

Set the attenuation of a particular channel on ADC1, and configure its associated GPIO pin mux.

The default ADC full-scale voltage is 1.1 V. To read higher voltages (up to the pin maximum voltage, usually 3.3 V) requires setting >0 dB signal attenuation for that ADC channel.

Note

For any given channel, this function must be called before the first time adc1_get_raw() is called for that channel.

Note

This function can be called multiple times to configure multiple ADC channels simultaneously. adc1_get_raw() can then be called for any configured channel.

When VDD_A is 3.3 V:

- 0 dB attenuation (ADC_ATTEN_DB_0) gives full-scale voltage 1.1 V
- 2.5 dB attenuation (ADC_ATTEN_DB_2_5) gives full-scale voltage 1.5 V
- 6 dB attenuation (ADC_ATTEN_DB_6) gives full-scale voltage 2.2 V
- 11 dB attenuation (ADC_ATTEN_DB_11) gives full-scale voltage 3.9 V (see note below)

Due to ADC characteristics, most accurate results are obtained within the following approximate voltage ranges:

Note

The full-scale voltage is the voltage corresponding to a maximum reading (depending on ADC1 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

Note

At 11 dB attenuation the maximum voltage is limited by VDD_A, not the full scale voltage.

- 0 dB attenuation (ADC_ATTEN_DB_0) between 100 and 950 mV
- 2.5 dB attenuation (ADC_ATTEN_DB_2_5) between 100 and 1250 mV
- 6 dB attenuation (ADC_ATTEN_DB_6) between 150 to 1750 mV
- 11 dB attenuation (ADC_ATTEN_DB_11) between 150 to 2450 mV

For maximum accuracy, use the ADC calibration APIs and measure voltages within these recommended ranges.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel` : ADC1 channel to configure
- `atten` : Attenuation level

int **adc1_get_raw**(adc1_channel_t *channel*)

Take an ADC1 reading from a single channel.

Note

When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of 'ECO_and_Workarounds_for_Bugs_in_ESP32' for the description of this issue.

Note

Call adc1_config_width() before the first time this function is called.

Note

For any given channel, adc1_config_channel_atten(channel) must be called before the first time this function is called. Configuring a new channel does not prevent a previously configured channel from being read.

Return

- -1: Parameter error
- Other: ADC1 channel reading.

Parameters

- `channel` : ADC1 channel to read

Enumerations

enum `adc1_channel_t`

Values:

`ADC1_CHANNEL_0` = 0

ADC1 channel 0 is GPIO36 (ESP32), GPIO1 (ESP32-S2)

`ADC1_CHANNEL_1`

ADC1 channel 1 is GPIO37 (ESP32), GPIO2 (ESP32-S2)

`ADC1_CHANNEL_2`

ADC1 channel 2 is GPIO38 (ESP32), GPIO3 (ESP32-S2)

`ADC1_CHANNEL_3`

ADC1 channel 3 is GPIO39 (ESP32), GPIO4 (ESP32-S2)

`ADC1_CHANNEL_4`

ADC1 channel 4 is GPIO32 (ESP32), GPIO5 (ESP32-S2)

`ADC1_CHANNEL_5`

ADC1 channel 5 is GPIO33 (ESP32), GPIO6 (ESP32-S2)

`ADC1_CHANNEL_6`

ADC1 channel 6 is GPIO34 (ESP32), GPIO7 (ESP32-S2)

`ADC1_CHANNEL_7`

ADC1 channel 7 is GPIO35 (ESP32), GPIO8 (ESP32-S2)

Anexo 2_ API Reference Timers

Structures

Struct **timer_config_t**

Data structure with timer's configuration settings.

Public Members

timer_alarm_t **alarm_en**

Timer alarm enable

timer_start_t **counter_en**

Counter enable

timer_intr_mode_t **intr_type**

Interrupt mode

timer_count_dir_t **counter_dir**

Counter direction

timer_autoreload_t **auto_reload**

Timer auto-reload

uint32_t **divider**

Counter clock divider. The divider's range is from from 2 to 65536.

Enumerations

enum **timer_group_t**

Selects a Timer-Group out of 2 available groups.

Values:

TIMER_GROUP_0 = 0 Hw timer group 0

TIMER_GROUP_1 = 1 Hw timer group 1

TIMER_GROUP_MAX

enum **timer_idx_t**

Select a hardware timer from timer groups.

Values:

TIMER_0 = 0 Select timer0 of GROUPx

TIMER_1 = 1 Select timer1 of GROUPx

TIMER_MAX

enum **timer_count_dir_t**

Decides the direction of counter.

Values:

TIMER_COUNT_DOWN = 0 Descending Count from cnt.high|cnt.low

TIMER_COUNT_UP = 1 Ascending Count from Zero

TIMER_COUNT_MAX

enum **timer_start_t**

Decides whether timer is on or paused.

Values:

TIMER_PAUSE = 0 Pause timer counter

TIMER_START = 1 Start timer counter

enum **timer_intr_t**

Interrupt types of the timer.

Values:

enum **timer_intr_mode_t**

Select interrupt type if running in alarm mode.

Values:

TIMER_INTR_T0 = BIT(0) interrupt of timer 0 **TIMER_INTR_T1** = BIT(1) interrupt of timer 1 **TIMER_INTR_WDT** = BIT(2) interrupt of watchdog **TIMER_INTR_NONE** = 0	**TIMER_INTR_LEVEL** = 0 Interrupt mode: level mode **TIMER_INTR_MAX**
enum **timer_alarm_t** Decides whether to enable alarm mode. *Values:* **TIMER_ALARM_DIS** = 0 Disable timer alarm **TIMER_ALARM_EN** = 1 Enable timer alarm **TIMER_ALARM_MAX**	*enum* **timer_autoreload_t** Select if Alarm needs to be loaded by software or automatically reload by hardware. *Values:* **TIMER_AUTORELOAD_DIS** = 0 Disable auto-reload: hardware will not load counter value after an alarm event **TIMER_AUTORELOAD_EN** = 1 Enable auto-reload: hardware will load counter value after an alarm event **TIMER_AUTORELOAD_MAX**

Functions

esp_err_t **timer_set_counter_value**(timer_group_t *group_num*, timer_idx_t *timer_num*, uint64_t *load_val*)

Set counter value to hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- **group_num** : Timer group, 0 for TIMERG0 or 1 for TIMERG1
- **timer_num** : Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- **load_val** : Counter value to write to the hardware timer.

esp_err_t **timer_start**(timer_group_t *group_num*, timer_idx_t *timer_num*)

Start the counter of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- **group_num** : Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- **timer_num** : Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]

esp_err_t **timer_pause**(timer_group_t *group_num*, timer_idx_t *timer_num*)

Pause the counter of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- **group_num** : Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- **timer_num** : Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]

---

esp_err_t **timer_set_alarm_value**(timer_group_t *group_num*, timer_idx_t *timer_num*, uint64_t *alarm_value*)

Set timer alarm value.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- **group_num** : Timer group, 0 for TIMERG0 or 1 for TIMERG1
- **timer_num** : Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- **alarm_value** : A 64-bit value to set the alarm value.

---

esp_err_t **timer_isr_register**(timer_group_t *group_num*, timer_idx_t *timer_num*, void (**fn*)(void *), void **arg*, int *intr_alloc_flags*, timer_isr_handle_t **handle*,)

Register Timer interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

If the intr_alloc_flags value ESP_INTR_FLAG_IRAM is set, the handler function must be declared with IRAM_ATTR attribute and can only call functions in IRAM or ROM. It cannot call other timer APIs. Use direct register access to configure timers from inside the ISR in this case.

Note

If use this function to register ISR, you need to write the whole ISR. In the interrupt handler, you need to call timer_spinlock_take(..) before your handling, and call timer_spinlock_give(…) after your handling.

Parameters

- **group_num** : Timer group number
- **timer_num** : Timer index of timer group
- **fn** : Interrupt handler function.
- **arg** : Parameter for handler function
- **intr_alloc_flags** : Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- **handle** : Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **timer_init**(timer_group_t *group_num*, timer_idx_t *timer_num*, const timer_config_t **config*)

Initializes and configure the timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- **group_num** : Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- **timer_num** : Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- **config** : Pointer to timer initialization parameters.

esp_err_t **timer_enable_intr**(timer_group_t *group_num*, timer_idx_t *timer_num*)

Enable timer interrupt.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- **group_num** : Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- **timer_num** : Timer index.

Anexo 3 – UART

Home / Programming / Language Reference / Functions / serial / Serial.begin()

Serial.begin()

Syntax

- Serial.begin(speed)
- Serial.begin(speed, config)

Parameters

config: sets data, parity, and stop bits. Valid values are:

- SERIAL_5N1
- SERIAL_6N1
- SERIAL_7N1
- SERIAL_8N1 (the default)
- SERIAL_5N2
- SERIAL_6N2
- SERIAL_7N2
- SERIAL_8N2
- SERIAL_5E1: even parity
- SERIAL_6E1
- SERIAL_7E1
- SERIAL_8E1
- SERIAL_5E2
- SERIAL_6E2
- SERIAL_7E2
- SERIAL_8E2
- SERIAL_5O1: odd parity
- SERIAL_6O1
- SERIAL_7O1
- SERIAL_8O1
- SERIAL_5O2
- SERIAL_6O2
- SERIAL_7O2
- SERIAL_8O2

Returns

Nothing

Anexo 4_ Arduino Reference Wire Library

Functions

Function Wire.beginTransmission ()

Description
Begin a transmission to the I2C slave device with the given address. Subsequently, queue bytes for transmission with the write() function and transmit them by calling endTransmission().

Syntax
Wire.beginTransmission(address);

Parameters
address: the 7-bit slave address

Returns:
None.

Function Wire.write()

Description
Writes data from a slave device in response to a request from a master, or queues bytes for transmission from a master to slave device (in-between calls to beginTransmission() and endTransmission()).

Syntax
Wire.write(value)
Wire.write(string)
Wire.write(data, length)

Parameters

- **value:** a value to send as a single byte
- **string:** a string to send as a series of bytes
- **data:** an array of data to send as bytes
- **length:** the number of bytes to transmit

Returns: byte: write() will return the number of bytes written, though reading that number is optional

Function Wire.requestFrom()

Description
Used by the master to request bytes from a slave device. The bytes may then be retrieved with the available() and read() functions.
As of Arduino 1.0.1, requestFrom() accepts a boolean argument changing its behavior for compatibility with certain I2C devices.
If true, requestFrom() sends a stop message after the request, releasing the I2C bus.
If false, requestFrom() sends a restart message after the request. The bus will not be released, which prevents another master device from requesting between messages. This allows one master device to send multiple requests while in control.
The default value is true.

Syntax
Wire.requestFrom(address, quantity)
Wire.requestFrom(address, quantity, stop)

Parameters
address: the 7-bit address of the device to request bytes from

quantity: the number of bytes to request

stop : boolean. true will send a stop message after the request, releasing the bus. false will continually send a restart after the request, keeping the connection active.

Returns
byte : the number of bytes returned from the slave device

Function Wire.available()

Description
Returns the number of bytes available for retrieval with read(). This should be called on a master device after a call to requestFrom() or on a slave inside the onReceive() handler.

available() inherits from the Stream utility class.

Parameters

None

Returns
The number of bytes available for reading.

Function Wire.endTransmission()

Description
Ends a transmission to a slave device that was begun by beginTransmission() and transmits the bytes that were queued by write().

Syntax
Wire.endTransmission()
Wire.endTransmission(stop)

Parameters : stop : boolean. true will send a stop message, releasing the bus after transmission. false will send a restart, keeping the connection active.

Returns: byte, which indicates the status of the transmission:

0:success
1:data too long to fit in transmit buffer
2:received NACK on transmit of address
3:received NACK on transmit of data
4:other error

Function Wire.read()

Description
Reads a byte that was transmitted from a slave device to a master after a call to requestFrom() or was transmitted from a master to a slave. read() inherits from the Stream utility class.

Syntax
Wire.read()

Parameters
none

Returns
The next byte received

Function Wire.setClock()

Description
This function modifies the clock frequency for I2C communication. I2C slave devices have no minimum working clock frequency, however 100KHz is usually the baseline.

Syntax
Wire.setClock(clockFrequency)

Parameters
clockFrequency: the value (in Hertz) of desired communication clock. Accepted values are 100000 (standard mode) and 400000 (fast mode). Some processors also support 10000 (low speed mode), 1000000 (fast mode plus) and 3400000 (high speed mode). Please refer to the specific processor documentation to make sure the desired mode is supported.

Returns
None

Function Wire.onreceive()

Description
Registers a function to be called when a slave device receives a transmission from a master.

Parameters
handler: the function to be called when the slave receives data; this should take a single int parameter (the number of bytes read from the master) and return nothing, e.g.: void myHandler(int numBytes)

Returns
None

Function Wire.onRequest

Description
Register a function to be called when a master requests data from this slave device.

Parameters
handler: the function to be called, takes no parameters and returns nothing, e.g.: void myHandler()

Returns
None

Anexo 5- librería Arduino

pinMode()

Syntax

`pinMode(pin, mode)`

Parameters

- `pin`: the Arduino pin number to set the mode of.
- `mode`: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`. See the **Digital Pins** page for a more complete description of the functionality.

attachInterrupt()

Syntax

- `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)` (recommended)
- `attachInterrupt(interrupt, ISR, mode)` (not recommended)
- `attachInterrupt(pin, ISR, mode)` (Not recommended. Additionally, this syntax only works on Arduino SAMD Boards, UNO WiFi Rev2, Due, and 101.)

Parameters

- `interrupt`: the number of the interrupt. Allowed data types: `int`.
- `pin`: the Arduino pin number.
- `ISR`: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.
- `mode`: defines when the interrupt should be triggered. Four constants are predefined as valid values:
  - **LOW** to trigger the interrupt whenever the pin is low,
  - **CHANGE** to trigger the interrupt whenever the pin changes value
  - **RISING** to trigger when the pin goes from low to high,
  - **FALLING** for when the pin goes from high to low.

The Due, Zero and MKR1000 boards allow also:

- **HIGH** to trigger the interrupt whenever the pin is high.

Anexo 6 - Funciones FreeRTOS

API Reference » System API » FreeRTOS (IDF)

static inline BaseType_t **xTaskCreate**(TaskFunction_t pxTaskCode, *const* char **const* pcName, *const* configSTACK_DEPTH_TYPE usStackDepth, void **const* pvParameters, UBaseType_t uxPriority, TaskHandle_t **const* pxCreatedTask)

xSemaphoreCreateBinary()

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

Returns: Handle to the created semaphore, or NULL if the memory required to hold the semaphore's data structures could not be allocated.

- **xSemaphore** -- Handle to the created semaphore. Should be of type SemaphoreHandle_t.

xSemaphoreTake(xSemaphore, xBlockTime)

Parameters:
- **xSemaphore** -- A handle to the semaphore being taken - obtained when the semaphore was created.
- **xBlockTime** -- The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. A block time of portMAX_DELAY can be used to block indefinitely (provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h).

Returns: pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

xSemaphoreGive(xSemaphore)

Parameters:
- **xSemaphore** -- A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns: pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

xSemaphoreGiveFromISR(xSemaphore, pxHigherPriorityTaskWoken)

Parameters:

- **xSemaphore** -- A handle to the semaphore being released. This is the handle returned when the semaphore was created.
- **pxHigherPriorityTaskWoken** -- xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns: pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

xSemaphoreTakeFromISR(xSemaphore, pxHigherPriorityTaskWoken)

Parameters:

- **xSemaphore** -- A handle to the semaphore being taken. This is the handle returned when the semaphore was created.
- **pxHigherPriorityTaskWoken** -- xSemaphoreTakeFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if taking the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreTakeFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns: pdTRUE if the semaphore was successfully taken, otherwise pdFALSE

xQueueCreate(uxQueueLength, uxItemSize)

Parameters:

- **uxQueueLength** -- The maximum number of items that the queue can contain.
- **uxItemSize** -- The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns: If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

xQueueSend(xQueue, pvItemToQueue, xTicksToWait)

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.
- **xTicksToWait** -- The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. 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.

Returns: pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

BaseType_t xQueueReceive(QueueHandle_t xQueue, void *const 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. xQueueReceive() will return immediately if xTicksToWait is zero and 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.

Returns: pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

UBaseType_t uxQueueSpacesAvailable(const QueueHandle_t xQueue)

Parameters: **xQueue** -- A handle to the queue being queried.

Returns: The number of messages available in the queue.

UBaseType_t uxQueueMessagesWaiting(const QueueHandle_t xQueue)

Parameters: **xQueue** -- A handle to the queue being queried.

Returns: The number of messages available in the queue.