Universal Asynchronous Receiver/Transmitter (UART)

[中文]

Introduction

A Universal Asynchronous Receiver/Transmitter (UART) is a hardware feature that handles communication (i.e., timing requirements and data framing) using widely-adopted asynchronous serial communication interfaces, such as RS232, RS422, and RS485. A UART provides a widely adopted and cheap method to realize full-duplex or half-duplex data exchange among different devices.

The ESP32-S3 chip has 3 UART controllers (also referred to as port), each featuring an identical set of registers to simplify programming and for more flexibility.

Each UART controller is independently configurable with parameters such as baud rate, data bit length, bit ordering, number of stop bits, parity bit, etc. All the regular UART controllers are compatible with UART-enabled devices from various manufacturers and can also support Infrared Data Association (IrDA) protocols.

The ESP32-S3 chip also supports using DMA with UART. For details, see to UART DMA (UHCI).

Functional Overview

The overview describes how to establish communication between an ESP32-S3 and other UART devices using the functions and data types of the UART driver. A typical programming workflow is broken down into the sections provided below:

1. Install Drivers - Allocating ESP32-S3's resources for the UART driver
2. Set Communication Parameters - Setting baud rate, data bits, stop bits, etc.
3. Set Communication Pins - Assigning pins for connection to a device
4. Run UART Communication - Sending/receiving data
5. Use Interrupts - Triggering interrupts on specific communication events
6. Deleting a Driver - Freeing allocated resources if a UART communication is no longer required

Steps 1 to 3 comprise the configuration stage. Step 4 is where the UART starts operating. Steps 5 and 6 are optional.

The UART driver's functions identify each of the UART controllers using `uart_port_t`. This identification is needed for all the following function calls.

Install Drivers

First of all, install the driver by calling `uart_driver_install()` and specify the following parameters:

- UART port number
- Size of RX ring buffer
- Size of TX ring buffer
- Event queue size
- Pointer to store the event queue handle
- Flags to allocate an interrupt

The function allocates the required internal resources for the UART driver.

```
// Setup UART buffered IO with event queue
const int uart_buffer_size = (1024 * 2);
QueueHandle_t uart_queue;
// Install UART driver using an event queue here
ESP_ERROR_CHECK(uart_driver_install(UART_NUM_2, uart_buffer_size, uart_buffer_size, 10,
&uart_queue, 0));
```

Set Communication Parameters

As the next step, UART communication parameters can be configured all in a single step or individually in multiple steps.

Single Step

Call the function `uart_param_config()` and pass to it a `uart_config_t` structure. The `uart_config_t` structure should contain all the required parameters. See the example below.

```
const uart_port_t uart_num = UART_NUM_2;
uart_config_t uart_config = {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_CTS_RTS,
    .rx_flow_ctrl_thresh = 122,
};
// Configure UART parameters
ESP_ERROR_CHECK(uart_param_config(uart_num, &uart_config));
```

For more information on how to configure the hardware flow control options, please refer to peripherals/uart/uart_echo.

Multiple Steps

Configure specific parameters individually by calling a dedicated function from the table given below. These functions are also useful if re-configuring a single parameter.

Functions for Configuring specific parameters individually

Parameter to Configure	Function
Baud rate	`uart_set_baudrate()`
Number of transmitted bits	`uart_set_word_length()` selected out of `uart_word_length_t`
Parity control	`uart_set_parity()` selected out of `uart_parity_t`
Number of stop bits	`uart_set_stop_bits()` selected out of `uart_stop_bits_t`
Hardware flow control mode	`uart_set_hw_flow_ctrl()` selected out of `uart_hw_flowcontrol_t`
Communication mode	`uart_set_mode()` selected out of `uart_mode_t`

Each of the above functions has a `_get_` counterpart to check the currently set value. For example, to check the current baud rate value, call `uart_get_baudrate()`.

Set Communication Pins

After setting communication parameters, configure the physical GPIO pins to which the other UART device will be connected. For this, call the function `uart_set_pin()` and specify the GPIO pin numbers to which the driver should route the TX, RX, RTS, and CTS signals. If you want to keep a currently allocated pin number for a specific signal, pass the macro `UART_PIN_NO_CHANGE`.

The same macro `UART_PIN_NO_CHANGE` should be specified for pins that will not be used.

```
// Set UART pins(TX: IO4, RX: IO5, RTS: IO18, CTS: IO19)
ESP_ERROR_CHECK(uart_set_pin(UART_NUM_2, 4, 5, 18, 19));
```

Run UART Communication

Serial communication is controlled by each UART controller's finite state machine (FSM).

The process of sending data involves the following steps:

1. Write data into TX FIFO buffer
2. FSM serializes the data
3. FSM sends the data out

The process of receiving data is similar, but the steps are reversed:

1. FSM processes an incoming serial stream and parallelizes it
2. FSM writes the data into RX FIFO buffer
3. Read the data from RX FIFO buffer

Therefore, an application only writes and reads data from a specific buffer using `uart_write_bytes()` and `uart_read_bytes()` respectively, and the FSM does the rest.

Transmit Data

After preparing the data for transmission, call the function `uart_write_bytes()` and pass the data buffer's address and data length to it. The function copies the data to the TX ring buffer (either immediately or after enough space is available), and then exit. When there is free space in the TX FIFO buffer, an interrupt service routine (ISR) moves the data from the TX ring buffer to the TX FIFO buffer in the background. The code below demonstrates the use of this function.

```
// Write data to UART.
char* test_str = "This is a test string.\n";
uart_write_bytes(uart_num, (const char*)test_str, strlen(test_str));
```

The function `uart_write_bytes_with_break()` is similar to `uart_write_bytes()` but adds a serial break signal at the end of the transmission. A 'serial break signal' means holding the TX line low for a period longer than one data frame.

```
// Write data to UART, end with a break signal.
uart_write_bytes_with_break(uart_num, "test break\n",strlen("test break\n"), 100);
```

Another function for writing data to the TX FIFO buffer is `uart_tx_chars()`. Unlike `uart_write_bytes()`, this function does not block until space is available. Instead, it writes all data which can immediately fit into the hardware TX FIFO, and then return the number of bytes that were written.

There is a 'companion' function `uart_wait_tx_done()` that monitors the status of the TX FIFO buffer and returns once it is empty.

```
// Wait for packet to be sent
const uart_port_t uart_num = UART_NUM_2;
ESP_ERROR_CHECK(uart_wait_tx_done(uart_num, 100)); // wait timeout is 100 RTOS ticks (TickType_t)
```

Receive Data

Once the data is received by the UART and saved in the RX FIFO buffer, it needs to be retrieved using the function `uart_read_bytes()`. Before reading data, you can check the number of bytes available in the RX FIFO buffer by calling `uart_get_buffered_data_len()`. An example of using these functions is given below.

```
// Read data from UART.
const uart_port_t uart_num = UART_NUM_2;
uint8_t data[128];
int length = 0;
ESP_ERROR_CHECK(uart_get_buffered_data_len(uart_num, (size_t*)&length));
length = uart_read_bytes(uart_num, data, length, 100);
```

If the data in the RX FIFO buffer is no longer needed, you can clear the buffer by calling `uart_flush()`.

Software Flow Control

If the hardware flow control is disabled, you can manually set the RTS and DTR signal levels by using the functions `uart_set_rts()` and `uart_set_dtr()` respectively.

Communication Mode Selection

The UART controller supports a number of communication modes. A mode can be selected using the function `uart_set_mode()`. Once a specific mode is selected, the UART driver handles the behavior of a connected UART device accordingly. As an example, it can control the RS485 driver chip using the RTS line to allow half-duplex RS485 communication.

```
// Setup UART in rs485 half duplex mode
ESP_ERROR_CHECK(uart_set_mode(uart_num, UART_MODE_RS485_HALF_DUPLEX));
```

Use Interrupts

There are many interrupts that can be generated depending on specific UART states or detected errors. The full list of available interrupts is provided in *ESP32-S3 Technical Reference Manual* > *UART Controller (UART)* > *UART Interrupts* and *UHCI Interrupts* [PDF]. You can enable or disable specific interrupts by calling `uart_enable_intr_mask()` or `uart_disable_intr_mask()` respectively.

The UART driver provides a convenient way to handle specific interrupts by wrapping them into corresponding events. Events defined in `uart_event_type_t` can be reported to a user application using the FreeRTOS queue functionality.

To receive the events that have happened, call `uart_driver_install()` and get the event queue handle returned from the function. Please see the above code snippet as an example.

The processed events include the following:

- **FIFO overflow** (`UART_FIFO_OVF`): The RX FIFO can trigger an interrupt when it receives more data than the FIFO can store.
    - (Optional) Configure the full threshold of the FIFO space by entering it in the structure `uart_intr_config_t` and call `uart_intr_config()` to set the configuration. This can help the data stored in the RX FIFO can be processed timely in the driver to avoid FIFO overflow.
    - Enable the interrupts using the functions `uart_enable_rx_intr()` .
    - Disable these interrupts using the corresponding functions `uart_disable_rx_intr()` .

```
const uart_port_t uart_num = UART_NUM_2;
// Configure a UART interrupt threshold and timeout
uart_intr_config_t uart_intr = {
    .intr_enable_mask = UART_INTR_RXFIFO_FULL | UART_INTR_RXFIFO_TOUT,
    .rxfifo_full_thresh = 100,
    .rx_timeout_thresh = 10,
};
ESP_ERROR_CHECK(uart_intr_config(uart_num, &uart_intr));

// Enable UART RX FIFO full threshold and timeout interrupts
ESP_ERROR_CHECK(uart_enable_rx_intr(uart_num));
```

- **Pattern detection** (`UART_PATTERN_DET`): An interrupt triggered on detecting a 'pattern' of the same character being received/sent repeatedly. It can be used, e.g., to detect a command string with a specific number of identical characters (the 'pattern') at the end. The following functions are available:
    - Configure and enable this interrupt using `uart_enable_pattern_det_baud_intr()`
    - Disable the interrupt using `uart_disable_pattern_det_intr()`

```
//Set UART pattern detect function
uart_enable_pattern_det_baud_intr(EX_UART_NUM, '+', PATTERN_CHR_NUM, 9, 0, 0);
```

- **Other events**: The UART driver can report other events such as data receiving (`UART_DATA`), ring buffer full (`UART_BUFFER_FULL`), detecting NULL after the stop bit (`UART_BREAK`), parity check error (`UART_PARITY_ERR`), and frame error (`UART_FRAME_ERR`).

The strings inside of brackets indicate corresponding event names. An example of how to handle various UART events can be found in peripherals/uart/uart_events.

Deleting a Driver

If the communication established with `uart_driver_install()` is no longer required, the driver can be removed to free allocated resources by calling `uart_driver_delete()` .

Macros

The API also defines several macros. For example, `UART_HW_FIFO_LEN` defines the length of hardware FIFO buffers; `UART_BITRATE_MAX` gives the maximum baud rate supported by the UART controllers, etc.

Overview of RS485 Specific Communication Options

Note

The following section uses `[UART_REGISTER_NAME].[UART_FIELD_BIT]` to refer to UART register fields/bits. For more information on a specific option bit, see **ESP32-S3 Technical Reference Manual** > **UART Controller (UART)** > **Register Summary** [PDF]. Use the register name to navigate to the register description and then find the field/bit.

- `UART_RS485_CONF_REG.UART_RS485_EN`: setting this bit enables RS485 communication mode support.
- `UART_RS485_CONF_REG.UART_RS485TX_RX_EN`: if this bit is set, the transmitter's output signal loops back to the receiver's input signal.
- `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN`: if this bit is set, the transmitter will still be sending data if the receiver is busy (remove collisions automatically by hardware).

The ESP32-S3's RS485 UART hardware can detect signal collisions during transmission of a datagram and generate the interrupt `UART_RS485_CLASH_INT` if this interrupt is enabled. The term collision means that a transmitted datagram is not equal to the one received on the other end. Data collisions are usually associated with the presence of other active devices on the bus or might occur due to bus errors.

The collision detection feature allows handling collisions when their interrupts are activated and triggered. The interrupts `UART_RS485_FRM_ERR_INT` and `UART_RS485_PARITY_ERR_INT` can be used with the collision detection feature to control frame errors and parity bit errors accordingly in RS485 mode. This functionality is supported in the UART driver and can be used by selecting the `UART_MODE_RS485_APP_CTRL` mode (see the function `uart_set_mode()`).

The collision detection feature can work with circuit A and circuit C (see Section Interface Connection Options). In the case of using circuit A or B, the RTS pin connected to the DE pin of the bus driver should be controlled by the user application. Use the function `uart_get_collision_flag()` to check if the collision detection flag has been raised.

The ESP32-S3 UART controllers themselves do not support half-duplex communication as they cannot provide automatic control of the RTS pin connected to the RE/DE input of RS485 bus driver. However, half-duplex communication can be achieved via software control of the RTS pin by the UART driver. This can be enabled by selecting the `UART_MODE_RS485_HALF_DUPLEX` mode when calling `uart_set_mode()`.

Once the host starts writing data to the TX FIFO buffer, the UART driver automatically asserts the RTS pin (logic 1); once the last bit of the data has been transmitted, the driver de-asserts the RTS pin (logic 0). To use this mode, the software would have to disable the hardware flow control function. This mode works with all the used circuits shown below.

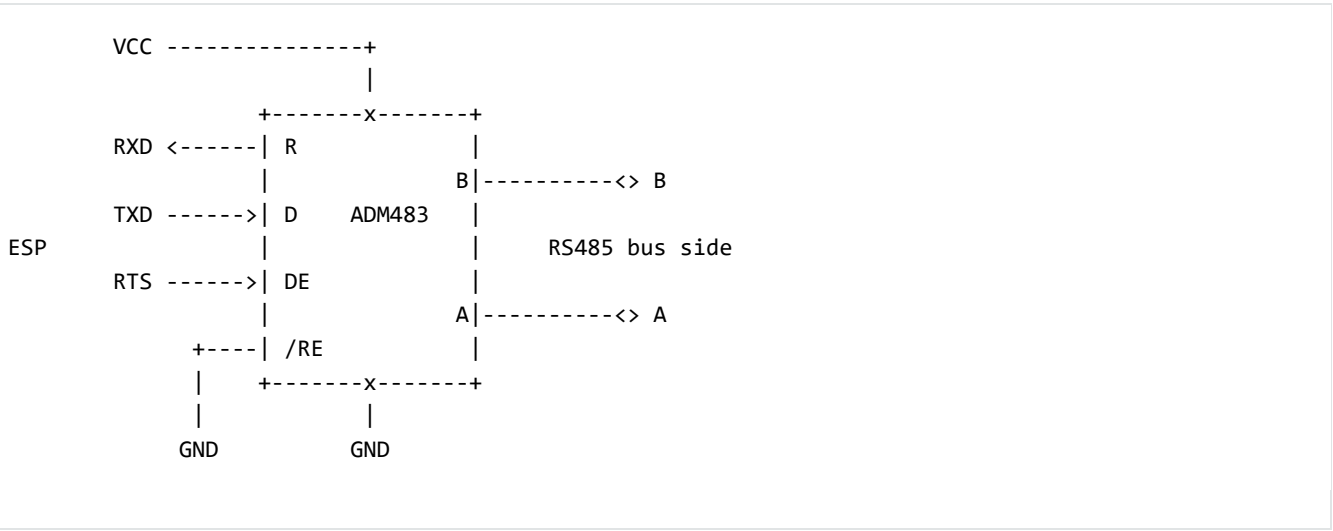
Interface Connection Options

This section provides example schematics to demonstrate the basic aspects of ESP32-S3's RS485 interface connection.

Note

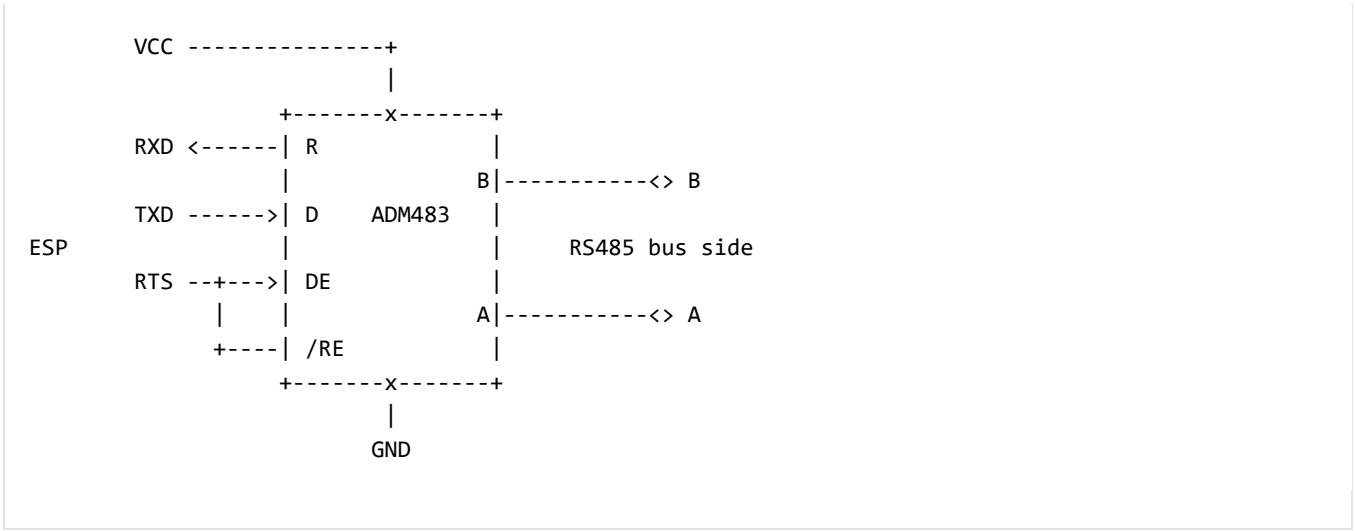
- The schematics below do **not** necessarily contain **all required elements**.
- The **analog devices** ADM483 & ADM2483 are examples of common RS485 transceivers and **can be replaced** with other similar transceivers.

Circuit A: Collision Detection Circuit



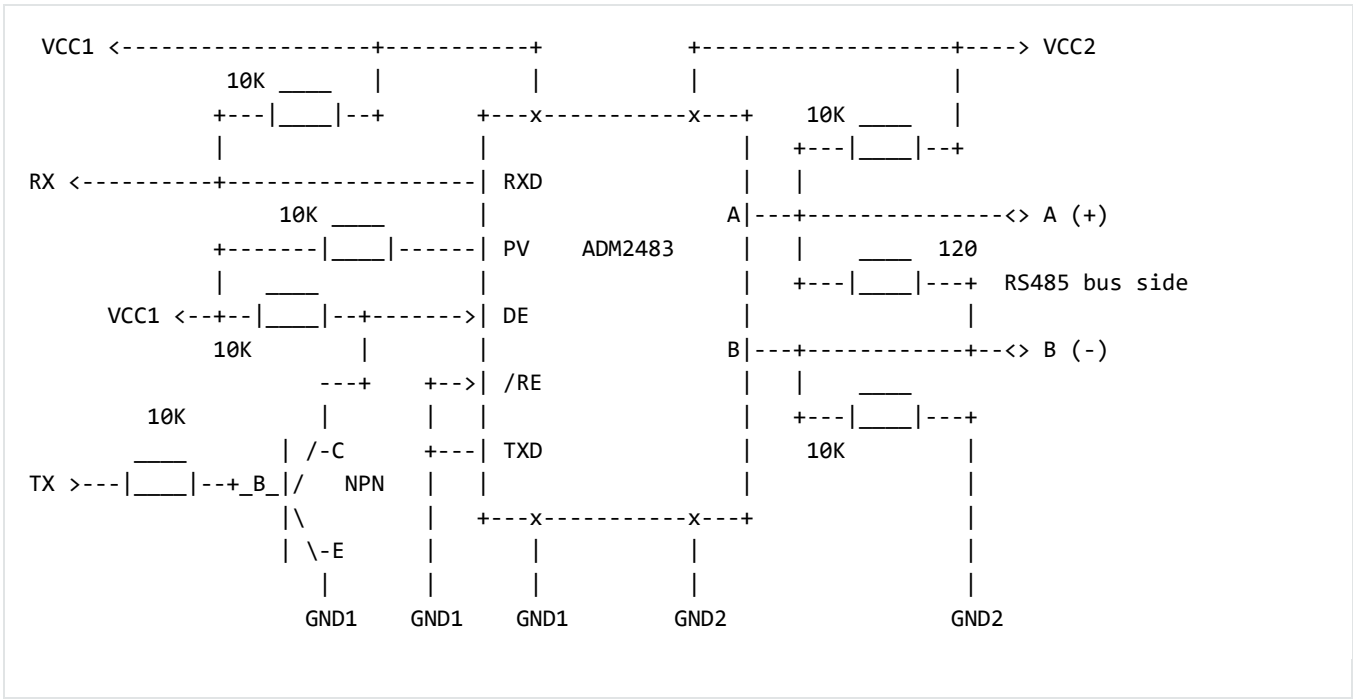
This circuit is preferable because it allows for collision detection and is quite simple at the same time. The receiver in the line driver is constantly enabled, which allows the UART to monitor the RS485 bus. Echo suppression is performed by the UART peripheral when the bit `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` is enabled.

Circuit B: Manual Switching Transmitter/Receiver Without Collision Detection



This circuit does not allow for collision detection. It suppresses the null bytes that the hardware receives when the bit `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` is set. The bit `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN` is not applicable in this case.

Circuit C: Auto Switching Transmitter/Receiver



This galvanically isolated circuit does not require RTS pin control by a software application or driver because it controls the transceiver direction automatically. However, it requires suppressing null bytes during transmission by setting `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN` to 1 and `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` to 0. This setup can work in any RS485 UART mode or even in `UART_MODE_UART`.

Application Examples

- peripherals/uart/uart_async_rxtxtasks demonstrates how to use two asynchronous tasks for communication via the same UART interface, with one task transmitting "Hello world" periodically and the other task receiving and printing data from the UART.
- peripherals/uart/uart_echo demonstrates how to use the UART interfaces to echo back any data received on the configured UART.
- peripherals/uart/uart_echo_rs485 demonstrates how to use the ESP32's UART software driver in RS485 half duplex transmission mode to echo any data it receives on UART port back to the sender in the RS485 network, requiring external connection of bus drivers.
- peripherals/uart/uart_events demonstrates how to use the UART driver to handle special UART events, read data from UART0, and echo it back to the monitoring console.
- peripherals/uart/uart_repl demonstrates how to use and connect two UARTs, allowing the UART used for stdout to send commands and receive replies from another console UART without human interaction.
- peripherals/uart/uart_select demonstrates the use of `select()` for synchronous I/O multiplexing on the UART interface, allowing for non-blocking read and write from/to various sources such as UART and sockets, where a ready resource can be served without being blocked by a busy resource.
- peripherals/uart/nmea0183_parser demonstrates how to parse NMEA-0183 data streams from GPS/BDS/GLONASS modules using the ESP UART Event driver and ESP event loop library, and output common information such as UTC time, latitude, longitude, altitude, and speed.

API Reference

Header File

- components/esp_driver_uart/include/driver/uart.h
- This header file can be included with:

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

- This header file is a part of the API provided by the `esp_driver_uart` component. To declare that your component depends on `esp_driver_uart`, add the following to your CMakeLists.txt:

```
REQUIRES esp_driver_uart
```

or

```
PRIV_REQUIRES esp_driver_uart
```

Functions

esp_err_t **uart_driver_install**(uart_port_t uart_num, int rx_buffer_size, int tx_buffer_size, int queue_size, QueueHandle_t *uart_queue, int intr_alloc_flags)

Install UART driver and set the UART to the default configuration.

UART ISR handler will be attached to the same CPU core that this function is running on.

Note

Rx_buffer_size should be greater than UART_HW_FIFO_LEN(uart_num). Tx_buffer_size should be either zero or greater than UART_HW_FIFO_LEN(uart_num).

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **rx_buffer_size** -- UART RX ring buffer size.
- **tx_buffer_size** -- UART TX ring buffer size. If set to zero, driver will not use TX buffer, TX function will block task until all data have been sent out.
- **queue_size** -- UART event queue size/depth.
- **uart_queue** -- UART event queue handle (out param). On success, a new queue handle is written here to provide access to UART events. If set to NULL, driver will not use an event queue.
- **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. Do not set ESP_INTR_FLAG_IRAM here (the driver's ISR handler is not located in IRAM)

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_driver_delete**(uart_port_t uart_num)

Uninstall UART driver.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

bool **uart_is_driver_installed**(uart_port_t uart_num)

Checks whether the driver is installed or not.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- true driver is installed
- false driver is not installed

---

esp_err_t **uart_set_word_length**(uart_port_t uart_num, uart_word_length_t data_bit)

Set UART data bits.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **data_bit** -- UART data bits

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_get_word_length**(uart_port_t uart_num, uart_word_length_t *data_bit)

Get the UART data bit configuration.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **data_bit** -- Pointer to accept value of UART data bits.

Returns:

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*data_bit)

---

esp_err_t **uart_set_stop_bits**(uart_port_t uart_num, uart_stop_bits_t stop_bits)

Set UART stop bits.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **stop_bits** -- UART stop bits

Returns:
- ESP_OK Success
- ESP_FAIL Fail

---

esp_err_t **uart_get_stop_bits**(uart_port_t uart_num, uart_stop_bits_t *stop_bits)

Get the UART stop bit configuration.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **stop_bits** -- Pointer to accept value of UART stop bits.

Returns:
- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*stop_bit)

---

esp_err_t **uart_set_parity**(uart_port_t uart_num, uart_parity_t parity_mode)

Set UART parity mode.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **parity_mode** -- the enum of uart parity configuration

Returns:
- ESP_FAIL Parameter error
- ESP_OK Success

---

esp_err_t **uart_get_parity**(uart_port_t uart_num, uart_parity_t *parity_mode)

Get the UART parity mode configuration.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **parity_mode** -- Pointer to accept value of UART parity mode.

Returns:
- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*parity_mode)

---

esp_err_t **uart_get_sclk_freq**(uart_sclk_t sclk, uint32_t *out_freq_hz)

Get the frequency of a clock source for the HP UART port.

Parameters:
- **sclk** -- Clock source
- **out_freq_hz** -- **[out]** Output of frequency, in Hz

Returns:

- ESP_ERR_INVALID_ARG: if the clock source is not supported
- otherwise ESP_OK

---

esp_err_t **uart_set_baudrate**(uart_port_t uart_num, uint32_t baudrate)

Set desired UART baud rate.

Note that the actual baud rate set could have a slight deviation from the user-configured value due to rounding error.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **baudrate** -- UART baud rate.

Returns:

- ESP_FAIL Parameter error, such as baud rate unachievable
- ESP_OK Success

---

esp_err_t **uart_get_baudrate**(uart_port_t uart_num, uint32_t *baudrate)

Get the actual UART baud rate.

It returns the real UART rate set in the hardware. It could have a slight deviation from the user-configured baud rate.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **baudrate** -- Pointer to accept value of UART baud rate

Returns:

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*baudrate)

---

esp_err_t **uart_set_line_inverse**(uart_port_t uart_num, uint32_t inverse_mask)

Set UART line inverse mode.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **inverse_mask** -- Choose the wires that need to be inverted. Using the ORred mask of `uart_signal_inv_t`

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_hw_flow_ctrl**(uart_port_t uart_num, uart_hw_flowcontrol_t flow_ctrl, uint8_t rx_thresh)

Set hardware flow control.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **flow_ctrl** -- Hardware flow control mode
- **rx_thresh** -- Threshold of Hardware RX flow control (0 ~ UART_HW_FIFO_LEN(uart_num)). Only when UART_HW_FLOWCTRL_RTS is set, will the rx_thresh value be set.

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_set_sw_flow_ctrl**(uart_port_t uart_num, bool enable, uint8_t rx_thresh_xon, uint8_t rx_thresh_xoff)

Set software flow control.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1)
- **enable** -- switch on or off
- **rx_thresh_xon** -- low water mark
- **rx_thresh_xoff** -- high water mark

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

esp_err_t **uart_get_hw_flow_ctrl**(uart_port_t uart_num, uart_hw_flowcontrol_t *flow_ctrl)

Get the UART hardware flow control configuration.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **flow_ctrl** -- Option for different flow control mode.

Returns:
- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*flow_ctrl)

esp_err_t **uart_clear_intr_status**(uart_port_t uart_num, uint32_t clr_mask)

Clear UART interrupt status.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **clr_mask** -- Bit mask of the interrupt status to be cleared.

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_enable_intr_mask**(uart_port_t uart_num, uint32_t enable_mask)

Set UART interrupt enable.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **enable_mask** -- Bit mask of the enable bits.

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_disable_intr_mask**(uart_port_t uart_num, uint32_t disable_mask)

Clear UART interrupt enable bits.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **disable_mask** -- Bit mask of the disable bits.

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_enable_rx_intr**(uart_port_t uart_num)

Enable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_disable_rx_intr**(uart_port_t uart_num)

Disable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_disable_tx_intr**(uart_port_t uart_num)

Disable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Parameters: **uart_num** -- UART port number

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_enable_tx_intr**(uart_port_t uart_num, int enable, int thresh)

Enable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **enable** -- 1: enable; 0: disable
- **thresh** -- Threshold of TX interrupt, 0 ~ UART_HW_FIFO_LEN(uart_num)

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_set_pin**(uart_port_t uart_num, int tx_io_num, int rx_io_num, int rts_io_num, int cts_io_num)

Assign signals of a UART peripheral to GPIO pins.

Note

If the GPIO number configured for a UART signal matches one of the IOMUX signals for that GPIO, the signal will be connected directly via the IOMUX. Otherwise the GPIO and signal will be connected via the GPIO Matrix. For example, if on an ESP32 the call `uart_set_pin(0, 1, 3, -1, -1)` is performed, as GPIO1 is UART0's default TX pin and GPIO3 is UART0's default RX pin, both will be connected to respectively U0TXD and U0RXD through the IOMUX, totally bypassing the GPIO matrix. The check is performed on a per-pin basis. Thus, it is possible to have RX pin binded to a GPIO through the GPIO matrix, whereas TX is binded to its GPIO through the IOMUX.

Note

It is possible to configure TX and RX to share the same IO (single wire mode), but please

be aware of output conflict, which could damage the pad. Apply open-drain and pull-up to the pad ahead of time as a protection, or the upper layer protocol must guarantee no output from two ends at the same time.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **tx_io_num** -- UART TX pin GPIO number.
- **rx_io_num** -- UART RX pin GPIO number.
- **rts_io_num** -- UART RTS pin GPIO number.
- **cts_io_num** -- UART CTS pin GPIO number.

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_set_rts**(uart_port_t uart_num, int level)

Manually set the UART RTS pin level.

Note

UART must be configured with hardware flow control disabled.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **level** -- 1: RTS output low (active); 0: RTS output high (block)

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_set_dtr**(uart_port_t uart_num, int level)

Manually set the UART DTR pin level.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **level** -- 1: DTR output low; 0: DTR output high

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_set_tx_idle_num**(uart_port_t uart_num, uint16_t idle_num)

Set UART idle interval after tx FIFO is empty.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **idle_num** -- idle interval after tx FIFO is empty(unit: the time it takes to send one bit under current baudrate)

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_param_config**(uart_port_t uart_num, *const* uart_config_t *uart_config)

Set UART configuration parameters.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **uart_config** -- UART parameter settings

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error, such as baud rate unachievable

---

esp_err_t **uart_intr_config**(uart_port_t uart_num, *const* uart_intr_config_t *intr_conf)

Configure UART interrupts.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **intr_conf** -- UART interrupt settings

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_wait_tx_done**(uart_port_t uart_num, TickType_t ticks_to_wait)

Wait until UART TX FIFO is empty.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **ticks_to_wait** -- Timeout, count in RTOS ticks

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error
- ESP_ERR_TIMEOUT Timeout

---

int **uart_tx_chars**(uart_port_t uart_num, *const* char *buffer, uint32_t len)

Send data to the UART port from a given buffer and length.

This function will not wait for enough space in TX FIFO. It will just fill the available TX FIFO and return when the FIFO is full.

Note

This function should only be used when UART TX buffer is not enabled.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **buffer** -- data buffer address
- **len** -- data length to send

Returns:
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

`int uart_write_bytes(uart_port_t uart_num, const void *src, size_t size)`

Send data to the UART port from a given buffer and length,.

If the UART driver's parameter 'tx_buffer_size' is set to zero: This function will not return until all the data have been sent out, or at least pushed into TX FIFO.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **src** -- data buffer address
- **size** -- data length to send

Returns:
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

`int uart_write_bytes_with_break(uart_port_t uart_num, const void *src, size_t size, int brk_len)`

Send data to the UART port from a given buffer and length,.

If the UART driver's parameter 'tx_buffer_size' is set to zero: This function will not return until all the data and the break signal have been sent out. After all data is sent out, send a break signal.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually. After all data sent out, send a break signal.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **src** -- data buffer address
- **size** -- data length to send
- **brk_len** -- break signal duration(unit: the time it takes to send one bit at current baudrate)

Returns:
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

---

int **uart_read_bytes**(uart_port_t uart_num, void *buf, uint32_t length, TickType_t ticks_to_wait)

UART read bytes from UART buffer.

Parameters:
- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **buf** -- pointer to the buffer.
- **length** -- data length
- **ticks_to_wait** -- sTimeout, count in RTOS ticks

Returns:
- (-1) Error
- OTHERS (>=0) The number of bytes read from UART buffer

---

esp_err_t **uart_flush**(uart_port_t uart_num)

Alias of uart_flush_input. UART ring buffer flush. This will discard all data in the UART RX buffer.

Note

Instead of waiting the data sent out, this function will clear UART rx buffer. In order to send all the data in tx FIFO, we can use uart_wait_tx_done function.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:
- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_flush_input**(uart_port_t uart_num)

Clear input buffer, discard all the data is in the ring-buffer.

Note

In order to send all the data in tx FIFO, we can use uart_wait_tx_done function.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_get_buffered_data_len**(uart_port_t uart_num, size_t *size)

UART get RX ring buffer cached data length.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **size** -- Pointer of size_t to accept cached data length

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_get_tx_buffer_free_size**(uart_port_t uart_num, size_t *size)

UART get TX ring buffer free space size.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **size** -- Pointer of size_t to accept the free space size

Returns:

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

---

esp_err_t **uart_disable_pattern_det_intr**(uart_port_t uart_num)

UART disable pattern detect function. Designed for applications like 'AT commands'. When the hardware detects a series of one same character, the interrupt will be triggered.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

---

esp_err_t **uart_enable_pattern_det_baud_intr**(uart_port_t uart_num, char pattern_chr, uint8_t chr_num, int chr_tout, int post_idle, int pre_idle)

UART enable pattern detect function. Designed for applications like 'AT commands'. When

the hardware detect a series of one same character, the interrupt will be triggered.

Parameters:

- **uart_num** -- UART port number.
- **pattern_chr** -- character of the pattern.
- **chr_num** -- number of the character, 8bit value.
- **chr_tout** -- timeout of the interval between each pattern characters, 16bit value, unit is the baud-rate cycle you configured. When the duration is more than this value, it will not take this data as at_cmd char.
- **post_idle** -- idle time after the last pattern character, 16bit value, unit is the baud-rate cycle you configured. When the duration is less than this value, it will not take the previous data as the last at_cmd char
- **pre_idle** -- idle time before the first pattern character, 16bit value, unit is the baud-rate cycle you configured. When the duration is less than this value, it will not take this data as the first at_cmd char.

Returns:

- ESP_OK Success
- ESP_FAIL Parameter error

`int uart_pattern_pop_pos(uart_port_t uart_num)`

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, this function will dequeue the first pattern position and move the pointer to next pattern position.

The following APIs will modify the pattern position info: uart_flush_input, uart_read_bytes, uart_driver_delete, uart_pop_pattern_pos It is the application's responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

Note

If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

`int uart_pattern_get_pos(uart_port_t uart_num)`

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, This function do nothing to the queue.

The following APIs will modify the pattern position info: uart_flush_input, uart_read_bytes, uart_driver_delete, uart_pop_pattern_pos It is the application's responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

Note

If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

Parameters: **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).

Returns:

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

---

esp_err_t **uart_pattern_queue_reset**(uart_port_t uart_num, int queue_length)

Allocate a new memory with the given length to save record the detected pattern position in rx buffer.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1).
- **queue_length** -- Max queue length for the detected pattern. If the queue length is not large enough, some pattern positions might be lost. Set this value to the maximum number of patterns that could be saved in data buffer at the same time.

Returns:

- ESP_ERR_NO_MEM No enough memory
- ESP_ERR_INVALID_STATE Driver not installed
- ESP_FAIL Parameter error
- ESP_OK Success

---

esp_err_t **uart_set_mode**(uart_port_t uart_num, uart_mode_t mode)

UART set communication mode.

Note

This function must be executed after uart_driver_install(), when the driver object is initialized.

Parameters:

- **uart_num** -- Uart number to configure, the max port number is (UART_NUM_MAX -1).
- **mode** -- UART UART mode to set

Returns:

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

---

esp_err_t **uart_set_rx_full_threshold**(uart_port_t uart_num, int threshold)

Set uart threshold value for RX fifo full.

Note

If application is using higher baudrate and it is observed that bytes in hardware RX fifo are overwritten then this threshold can be reduced

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1)
- **threshold** -- Threshold value above which RX fifo full interrupt is generated

Returns:

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

---

esp_err_t **uart_set_tx_empty_threshold**(uart_port_t uart_num, int threshold)

Set uart threshold values for TX fifo empty.

Parameters:

- **uart_num** -- UART port number, the max port number is (UART_NUM_MAX -1)
- **threshold** -- Threshold value below which TX fifo empty interrupt is generated

Returns:

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

---

esp_err_t **uart_set_rx_timeout**(uart_port_t uart_num, *const* uint8_t tout_thresh)

UART set threshold timeout for TOUT feature.

Parameters:
- **uart_num** -- Uart number to configure, the max port number is (UART_NUM_MAX -1).
- **tout_thresh** -- This parameter defines timeout threshold in uart symbol periods. The maximum value of threshold is 126. tout_thresh = 1, defines TOUT interrupt timeout equal to transmission time of one symbol (~11 bit) on current baudrate. If the time is expired the UART_RXFIFO_TOUT_INT interrupt is triggered. If tout_thresh == 0, the TOUT feature is disabled.

Returns:
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

---

esp_err_t **uart_get_collision_flag**(uart_port_t uart_num, bool *collision_flag)

Returns collision detection flag for RS485 mode Function returns the collision detection flag into variable pointed by collision_flag. *collision_flag = true, if collision detected else it is equal to false. This function should be executed when actual transmission is completed (after uart_write_bytes()).

Parameters:
- **uart_num** -- Uart number to configure the max port number is (UART_NUM_MAX -1).
- **collision_flag** -- Pointer to variable of type bool to return collision flag.

Returns:
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

---

esp_err_t **uart_set_wakeup_threshold**(uart_port_t uart_num, int wakeup_threshold)

Set the number of RX pin signal edges for light sleep wakeup.

UART can be used to wake up the system from light sleep. This feature works by counting the number of positive edges on RX pin and comparing the count to the threshold. When the count exceeds the threshold, system is woken up from light sleep. This function allows setting the threshold value.

Stop bit and parity bits (if enabled) also contribute to the number of edges. For example, letter 'a' with ASCII code 97 is encoded as 0100001101 on the wire (with 8n1 configuration), start and stop bits included. This sequence has 3 positive edges (transitions from 0 to 1). Therefore, to wake up the system when 'a' is sent, set wakeup_threshold=3.

The character that triggers wakeup is not received by UART (i.e. it can not be obtained from UART FIFO). Depending on the baud rate, a few characters after that will also not be received. Note that when the chip enters and exits light sleep mode, APB frequency will be changing. To ensure that UART has correct Baud rate all the time, it is necessary to select a

source clock which has a fixed frequency and remains active during sleep. For the supported clock sources of the chips, please refer to `uart_sclk_t` or `soc_periph_uart_clk_src_legacy_t`

Note

in ESP32, the wakeup signal can only be input via IO_MUX (i.e. GPIO3 should be configured as function_1 to wake up UART0, GPIO9 should be configured as function_5 to wake up UART1), UART2 does not support light sleep wakeup feature.

Parameters:
- **uart_num** -- UART number, the max port number is (UART_NUM_MAX -1).
- **wakeup_threshold** -- number of RX edges for light sleep wakeup, value is 3 .. 0x3ff.

Returns:
- ESP_OK on success
- ESP_ERR_INVALID_ARG if uart_num is incorrect or wakeup_threshold is outside of [3, 0x3ff] range.

---

esp_err_t **uart_get_wakeup_threshold**(uart_port_t uart_num, int *out_wakeup_threshold)

Get the number of RX pin signal edges for light sleep wakeup.

See description of uart_set_wakeup_threshold for the explanation of UART wakeup feature.

Parameters:
- **uart_num** -- UART number, the max port number is (UART_NUM_MAX -1).
- **out_wakeup_threshold** -- **[out]** output, set to the current value of wakeup threshold for the given UART.

Returns:
- ESP_OK on success
- ESP_ERR_INVALID_ARG if out_wakeup_threshold is NULL

---

esp_err_t **uart_wait_tx_idle_polling**(uart_port_t uart_num)

Wait until UART tx memory empty and the last char send ok (polling mode).

Returns:
- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters: **uart_num** -- UART number

---

esp_err_t **uart_set_loop_back**(uart_port_t uart_num, bool loop_back_en)

Configure TX signal loop back to RX module, just for the test usage.

- **Returns:**
  - ESP_OK on success
  - ESP_ERR_INVALID_ARG Parameter error
  - ESP_FAIL Driver not installed

Parameters:
- **uart_num** -- UART number
- **loop_back_en** -- Set true to enable the loop back function, else set it false.

---

void **uart_set_always_rx_timeout**(uart_port_t uart_num, bool always_rx_timeout_en)

Configure behavior of UART RX timeout interrupt.

When always_rx_timeout is true, timeout interrupt is triggered even if FIFO is full. This function can cause extra timeout interrupts triggered only to send the timeout event. Call this function only if you want to ensure timeout interrupt will always happen after a byte stream.

Parameters:
- **uart_num** -- UART number
- **always_rx_timeout_en** -- Set to false enable the default behavior of timeout interrupt, set it to true to always trigger timeout interrupt.

---

esp_err_t **uart_detect_bitrate_start**(uart_port_t uart_num, *const* uart_bitrate_detect_config_t *config)

Start to do a bitrate detection for an incoming data signal (auto baud rate detection)

This function can act as a standalone API. No need to install UART driver before calling this function.

It is recommended that the incoming data line contains alternating bit sequence, data bytes such as `0x55` or `0xAA`. Characters 'NULL', `0xCC` are not good for the measurement.

Parameters:
- **uart_num** -- The ID of the UART port to be used to do the measurement. Note that only HP UART ports have the capability.
- **config** -- Pointer to the configuration structure for the UART port. If the port has already been acquired, this parameter is ignored.

Returns:
- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL No free uart port or source_clk invalid

---

esp_err_t **uart_detect_bitrate_stop**(uart_port_t uart_num, bool deinit, uart_bitrate_res_t *ret_res)

Stop the bitrate detection.

The measurement period should last for at least one-byte long if detecting UART baud rate, then call this function to stop and get the measurement result.

Parameters:
- **uart_num** -- The ID of the UART port
- **deinit** -- Whether to release the UART port after finishing the measurement
- **ret_res** -- **[out]** Structure to store the measurement results

Returns:
- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Unknown tick frequency

Structures

struct **uart_config_t**

UART configuration parameters for uart_param_config function.

Public Members

int baud_rate

UART baud rate Note that the actual baud rate set could have a slight deviation from the user-configured value due to rounding error

uart_word_length_t **data_bits**

UART byte size

uart_parity_t **parity**

UART parity mode

uart_stop_bits_t **stop_bits**

UART stop bits

uart_hw_flowcontrol_t **flow_ctrl**

UART HW flow control mode (cts/rts)

uint8_t rx_flow_ctrl_thresh

UART HW RTS threshold

uart_sclk_t **source_clk**

UART source clock selection

uint32_t allow_pd

If set, driver allows the power domain to be powered off when system enters sleep mode. This can save power, but at the expense of more RAM being consumed to save register context.

uint32_t backup_before_sleep

Deprecated:

, same meaning as allow_pd

struct uart_config_t **flags**

Configuration flags

struct **uart_intr_config_t**

UART interrupt configuration parameters for uart_intr_config function.

Public Members

uint32_t intr_enable_mask

UART interrupt enable mask, choose from UART_XXXX_INT_ENA_M under UART_INT_ENA_REG(i), connect with bit-or operator

uint8_t rx_timeout_thresh

UART timeout interrupt threshold (unit: time of sending one byte)

uint8_t txfifo_empty_intr_thresh

UART TX empty interrupt threshold.

uint8_t rxfifo_full_thresh

UART RX full interrupt threshold.

struct **uart_event_t**

Event structure used in UART event queue.

Public Members

uart_event_type_t **type**

UART event type

size_t size

UART data size for UART_DATA event

bool timeout_flag

UART data read timeout flag for UART_DATA event (no new data received during configured RX TOUT) If the event is caused by FIFO-full interrupt, then there will be no event with the timeout flag before the next byte coming.

struct uart_bitrate_detect_config_t

UART bitrate detection configuration parameters for `uart_detect_bitrate_start` function to acquire a new uart handle.

Public Members

int rx_io_num

GPIO pin number for the incoming signal

uart_sclk_t **source_clk**

The higher the frequency of the clock source, the more accurate the detected bitrate value; The slower the frequency of the clock source, the slower the bitrate can be measured

struct uart_bitrate_res_t

Structure to store the measurement results for UART bitrate detection within the measurement period.

Formula to calculate the bitrate: If the signal is ideal, bitrate = clk_freq_hz * 2 / (low_period + high_period) If the signal is weak along falling edges, then you may use bitrate = clk_freq_hz * 2 / pos_period If the signal is weak along rising edges, then you may use bitrate = clk_freq_hz * 2 / neg_period

Public Members

uint32_t low_period

Stores the minimum tick count of a low-level pulse

uint32_t high_period

Stores the minimum tick count of a high-level pulse

uint32_t pos_period

Stores the minimum tick count between two positive edges

uint32_t neg_period

Stores the minimum tick count between two negative edges

uint32_t edge_cnt

Stores the count of RX edge changes (10-bit counter, be careful, it could overflow)

uint32_t clk_freq_hz

The frequency of the tick being used to count the measurement results, in Hz

Macros

UART_PIN_NO_CHANGE

UART_FIFO_LEN

Length of the HP UART HW FIFO.

UART_HW_FIFO_LEN(uart_num)

Length of the UART HW FIFO.

UART_BITRATE_MAX

Maximum configurable bitrate.

Type Definitions

typedef intr_handle_t **uart_isr_handle_t**

Enumerations

enum **uart_event_type_t**

UART event types used in the ring buffer.

Values:

enumerator **UART_DATA**

Triggered when the receiver either takes longer than rx_timeout_thresh to receive a byte, or when more data is received than what rxfifo_full_thresh specifies

enumerator **UART_BREAK**

Triggered when the receiver detects a NULL character

enumerator **UART_BUFFER_FULL**

Triggered when RX ring buffer is full

enumerator **UART_FIFO_OVF**

Triggered when the received data exceeds the capacity of the RX FIFO

enumerator **UART_FRAME_ERR**

Triggered when the receiver detects a data frame error

enumerator **UART_PARITY_ERR**

Triggered when a parity error is detected in the received data

enumerator **UART_DATA_BREAK**

Internal event triggered to signal a break afte data transmission

enumerator **UART_PATTERN_DET**

Triggered when a specified pattern is detected in the incoming data

enumerator **UART_WAKEUP**

Triggered when a wakeup signal is detected

enumerator **UART_EVENT_MAX**

Maximum index for UART events

Header File

- components/esp_driver_uart/include/driver/uart_wakeup.h
- This header file can be included with:

```
#include "driver/uart_wakeup.h"
```

- This header file is a part of the API provided by the `esp_driver_uart` component. To declare that your component depends on `esp_driver_uart`, add the following to your CMakeLists.txt:

```
REQUIRES esp_driver_uart
```

or

```
PRIV_REQUIRES esp_driver_uart
```

Functions

esp_err_t **uart_wakeup_setup**(uart_port_t uart_num, *const* uart_wakeup_cfg_t *cfg)

Initializes the UART wakeup functionality.

This function configures the wakeup behavior for a specified UART port based on the provided configuration. The behavior depends on the selected wakeup mode and additional parameters such as active threshold or character sequence, if applicable. It is important that the provided configuration matches the capabilities of the SOC to ensure proper operation.

Parameters:
- **uart_num** -- The UART port to initialize for wakeup (e.g., UART_NUM_0, UART_NUM_1, etc.).
- **cfg** -- Pointer to the `uart_wakeup_cfg_t` structure that contains the wakeup configuration settings.

Returns:
- `ESP_OK` if the wakeup configuration was successfully applied.
- `ESP_ERR_INVALID_ARG` if the provided configuration is invalid (e.g., threshold values out of range).

esp_err_t **uart_wakeup_clear**(uart_port_t uart_num, uart_wakeup_mode_t wakeup_mode)

Clear the UART wakeup configuration.

This function will clear the UART wakeup behavior and set to its default configuration.

Parameters:
- **uart_num** -- The UART port to initialize for wakeup (e.g., UART_NUM_0, UART_NUM_1, etc.).
- **wakeup_mode** -- The UART wakeup mode set in `uart_wakeup_setup`.

Returns:

- `ESP_OK` Clear wakeup configuration successfully.

Structures

struct **uart_wakeup_cfg_t**

Structure that holds configuration for UART wakeup.

This structure is used to configure the wakeup behavior for a UART port. The wakeup mode can be selected from several options, such as active threshold, FIFO threshold, start bit detection, and character sequence detection. The availability of different wakeup modes depends on the SOC capabilities.

Public Members

uart_wakeup_mode_t **wakeup_mode**

Wakeup mode selection

uint16_t rx_edge_threshold

Used in Active threshold wake-up; related: UART_WK_MODE_ACTIVE_THRESH; Configures the number of RXD edge changes to wake up the chip.

Header File

- components/hal/include/hal/uart_types.h
- This header file can be included with:

```
#include "hal/uart_types.h"
```

Structures

struct **uart_at_cmd_t**

UART AT cmd char configuration parameters Note that this function may different on different chip. Please refer to the TRM at confirguration.

Public Members

uint8_t cmd_char

UART AT cmd char

uint8_t char_num

AT cmd char repeat number

uint32_t gap_tout

gap time(in baud-rate) between AT cmd char

uint32_t pre_idle

the idle time(in baud-rate) between the non AT char and first AT char

uint32_t post_idle

the idle time(in baud-rate) between the last AT char and the none AT char

struct **uart_sw_flowctrl_t**

UART software flow control configuration parameters.

Public Members

uint8_t xon_char

Xon flow control char

uint8_t xoff_char

Xoff flow control char

uint8_t xon_thrd

If the software flow control is enabled and the data amount in rxfifo is less than xon_thrd, an xon_char will be sent

uint8_t xoff_thrd

If the software flow control is enabled and the data amount in rxfifo is more than xoff_thrd, an xoff_char will be sent

Type Definitions

typedef soc_periph_uart_clk_src_legacy_t **uart_sclk_t**

UART source clock.

Enumerations

enum **uart_port_t**

UART port number, can be UART_NUM_0 ~ (UART_NUM_MAX -1).

Values:

enumerator **UART_NUM_0**

UART port 0

enumerator **UART_NUM_1**

UART port 1

enumerator **UART_NUM_2**

UART port 2

enumerator **UART_NUM_MAX**

UART port max

enum **uart_mode_t**

UART mode selection.

Values:

enumerator **UART_MODE_UART**

mode: regular UART mode

enumerator **UART_MODE_RS485_HALF_DUPLEX**

mode: half duplex RS485 UART mode control by RTS pin

enumerator **UART_MODE_IRDA**

mode: IRDA UART mode

enumerator **UART_MODE_RS485_COLLISION_DETECT**

mode: RS485 collision detection UART mode (used for test purposes)

enumerator **UART_MODE_RS485_APP_CTRL**

mode: application control RS485 UART mode (used for test purposes)

enum **uart_word_length_t**

UART word length constants.

Values:

enumerator **UART_DATA_5_BITS**

word length: 5bits

enumerator **UART_DATA_6_BITS**

word length: 6bits

enumerator **UART_DATA_7_BITS**

word length: 7bits

enumerator **UART_DATA_8_BITS**

word length: 8bits

enumerator **UART_DATA_BITS_MAX**

enum **uart_stop_bits_t**

UART stop bits number.

Values:

enumerator **UART_STOP_BITS_1**

stop bit: 1bit

enumerator **UART_STOP_BITS_1_5**

stop bit: 1.5bits

enumerator **UART_STOP_BITS_2**

stop bit: 2bits

enumerator **UART_STOP_BITS_MAX**

enum **uart_parity_t**

UART parity constants.

Values:

enumerator **UART_PARITY_DISABLE**

Disable UART parity

enumerator **UART_PARITY_EVEN**

Enable UART even parity

enumerator **UART_PARITY_ODD**

Enable UART odd parity

enum **uart_hw_flowcontrol_t**

UART hardware flow control modes.

Values:

enumerator **UART_HW_FLOWCTRL_DISABLE**

disable hardware flow control

enumerator **UART_HW_FLOWCTRL_RTS**

enable RX hardware flow control (rts)

enumerator **UART_HW_FLOWCTRL_CTS**

enable TX hardware flow control (cts)

enumerator **UART_HW_FLOWCTRL_CTS_RTS**

enable hardware flow control

enumerator **UART_HW_FLOWCTRL_MAX**

enum **uart_signal_inv_t**

UART signal bit map.

Values:

enumerator **UART_SIGNAL_INV_DISABLE**

Disable UART signal inverse

enumerator **UART_SIGNAL_IRDA_TX_INV**

inverse the UART irda_tx signal

enumerator **UART_SIGNAL_IRDA_RX_INV**

inverse the UART irda_rx signal

enumerator **UART_SIGNAL_RXD_INV**

inverse the UART rxd signal

enumerator **UART_SIGNAL_CTS_INV**

inverse the UART cts signal

enumerator **UART_SIGNAL_DSR_INV**

inverse the UART dsr signal

enumerator **UART_SIGNAL_TXD_INV**

inverse the UART txd signal

enumerator **UART_SIGNAL_RTS_INV**

inverse the UART rts signal

enumerator **UART_SIGNAL_DTR_INV**

inverse the UART dtr signal

enum **uart_wakeup_mode_t**

Enumeration of UART wake-up modes.

Values:

enumerator **UART_WK_MODE_ACTIVE_THRESH**

Wake-up triggered by active edge threshold

GPIO Lookup Macros

The UART peripherals have dedicated IO_MUX pins to which they are connected directly. However, signals can also be routed to other pins using the less direct GPIO matrix. To use direct routes, you need to know which pin is a dedicated IO_MUX pin for a UART channel. GPIO Lookup Macros simplify the process of finding and assigning IO_MUX pins. You choose a macro based on either the IO_MUX pin number, or a required UART channel name, and the macro returns the matching counterpart for you. See some examples below.

Note

These macros are useful if you need very high UART baud rates (over 40 MHz), which means you will have to use IO_MUX pins only. In other cases, these macros can be ignored, and you can use the GPIO Matrix as it allows you to configure any GPIO pin for any UART function.

1. `UART_NUM_2_TXD_DIRECT_GPIO_NUM` returns the IO_MUX pin number of UART channel 2 TXD pin (pin 17)
2. `UART_GPIO19_DIRECT_CHANNEL` returns the UART number of GPIO 19 when connected to the UART peripheral via IO_MUX (this is UART_NUM_0)
3. `UART_CTS_GPIO19_DIRECT_CHANNEL` returns the UART number of GPIO 19 when used as the UART CTS pin via IO_MUX (this is UART_NUM_0). It is similar to the above macro but specifies the pin function which is also part of the IO_MUX assignment.

Header File

- components/soc/esp32s3/include/soc/uart_channel.h
- This header file can be included with:

```
#include "soc/uart_channel.h"
```

Macros

UART_GPIO43_DIRECT_CHANNEL

UART_NUM_0_TXD_DIRECT_GPIO_NUM

UART_GPIO44_DIRECT_CHANNEL

UART_NUM_0_RXD_DIRECT_GPIO_NUM

UART_GPIO16_DIRECT_CHANNEL

UART_NUM_0_CTS_DIRECT_GPIO_NUM

UART_GPIO15_DIRECT_CHANNEL

UART_NUM_0_RTS_DIRECT_GPIO_NUM

UART_TXD_GPIO43_DIRECT_CHANNEL

UART_RXD_GPIO44_DIRECT_CHANNEL

UART_CTS_GPIO16_DIRECT_CHANNEL

UART_RTS_GPIO15_DIRECT_CHANNEL

UART_GPIO17_DIRECT_CHANNEL

UART_NUM_1_TXD_DIRECT_GPIO_NUM

UART_GPIO18_DIRECT_CHANNEL

UART_NUM_1_RXD_DIRECT_GPIO_NUM

UART_GPIO20_DIRECT_CHANNEL

UART_NUM_1_CTS_DIRECT_GPIO_NUM

UART_GPIO19_DIRECT_CHANNEL

UART_NUM_1_RTS_DIRECT_GPIO_NUM

UART_TXD_GPIO17_DIRECT_CHANNEL

UART_RXD_GPIO18_DIRECT_CHANNEL

UART_CTS_GPIO20_DIRECT_CHANNEL

UART_RTS_GPIO19_DIRECT_CHANNEL