Inter-Integrated Circuit (I2C)

Introduction

I2C is a serial, synchronous, multi-device, half-duplex communication protocol that allows co-existence of multiple masters and slaves on the same bus. I2C uses two bidirectional open-drain lines: serial data line (SDA) and serial clock line (SCL), pulled up by resistors.

ESP32-S3 has 2 I2C controller (also called port), responsible for handling communication on the I2C bus. A single I2C controller can be a master or a slave.

Typically, an I2C slave device has a 7-bit address or 10-bit address. ESP32-S3 supports both I2C Standard-mode (Sm) and Fast-mode (Fm) which can go up to 100KHz and 400KHz respectively.

Warning

The clock frequency of SCL in master mode should not be larger than 400 KHz

Note

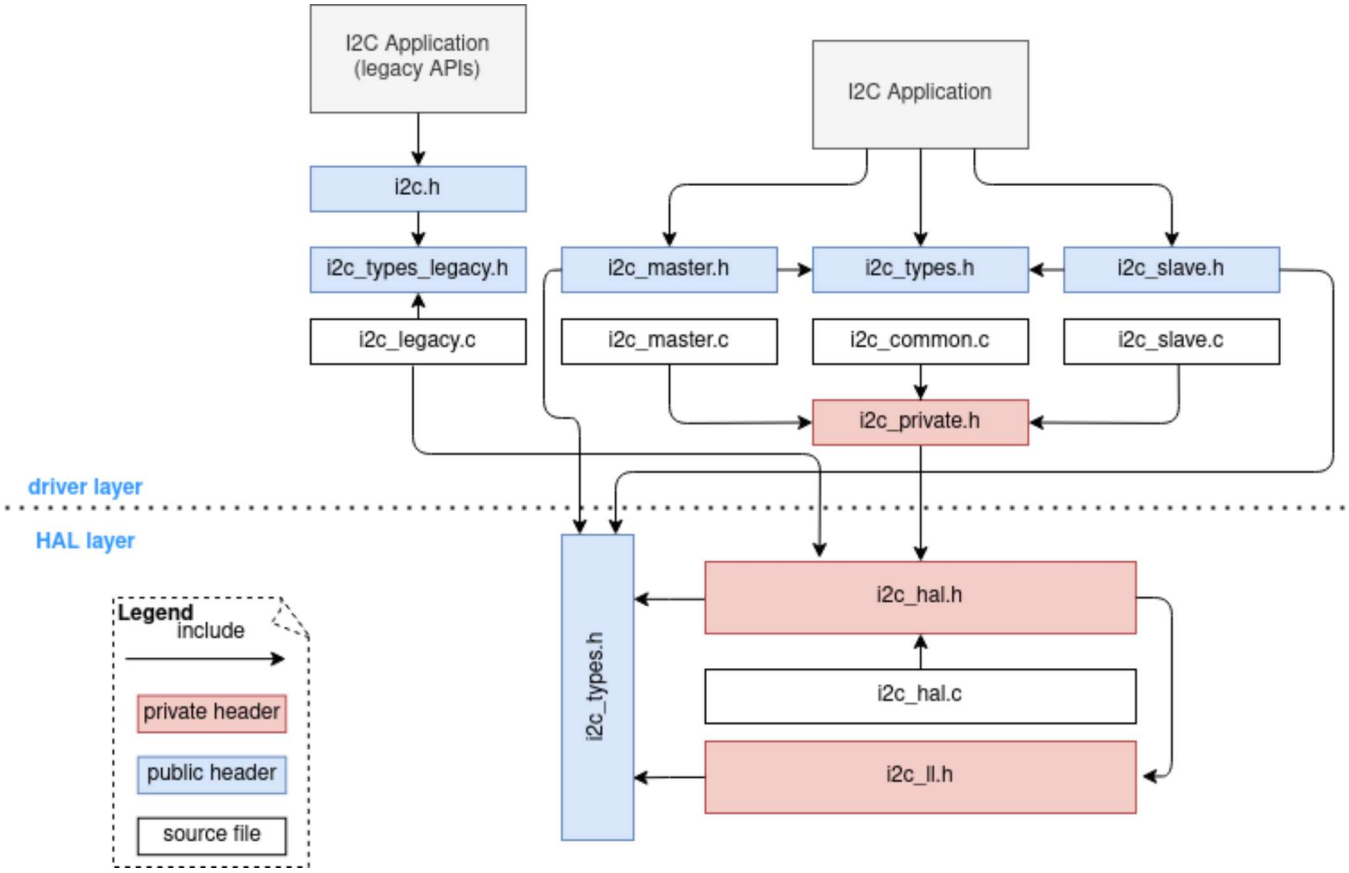
The frequency of SCL is influenced by both the pull-up resistor and the wire capacitance. Therefore, users are strongly recommended to choose appropriate pull-up resistors to make the frequency accurate. The recommended value for pull-up resistors usually ranges from 1K Ohms to 10K Ohms.

Keep in mind that the higher the frequency, the smaller the pull-up resistor should be (but not less than 1 KOhms). Indeed, large resistors will decline the current, which will increase the clock switching time and reduce the frequency. We usually recommend a range of 2 KOhms to 5 KOhms, but users may also need to make some adjustments depending on their current draw requirements.

I2C Clock Configuration

- `i2c_clock_source_t::I2C_CLK_SRC_DEFAULT`: Default I2C source clock.
- `i2c_clock_source_t::I2C_CLK_SRC_XTAL`: External crystal for I2C clock source.
- `i2c_clock_source_t::I2C_CLK_RC_FAST`: Internal 20MHz rc oscillator for I2C clock source.

I2C File Structure



I2C file structure

Public headers that need to be included in the I2C application

- `i2c.h`: The header file of legacy I2C APIs (for apps using legacy driver).
- `i2c_master.h`: The header file that provides standard communication mode specific APIs (for apps using new driver with master mode).
- `i2c_slave.h`: The header file that provides standard communication mode specific APIs (for apps using new driver with slave mode).

> **Note**
>
> The legacy driver can't coexist with the new driver. Include `i2c.h` to use the legacy driver or the other two headers to use the new driver. Please keep in mind that the legacy driver is now deprecated and will be removed in future.

Public headers that have been included in the headers above

- `i2c_types_legacy.h`: The legacy public types that only used in the legacy driver.
- `i2c_types.h`: The header file that provides public types.

Functional Overview

The I2C driver offers following services:

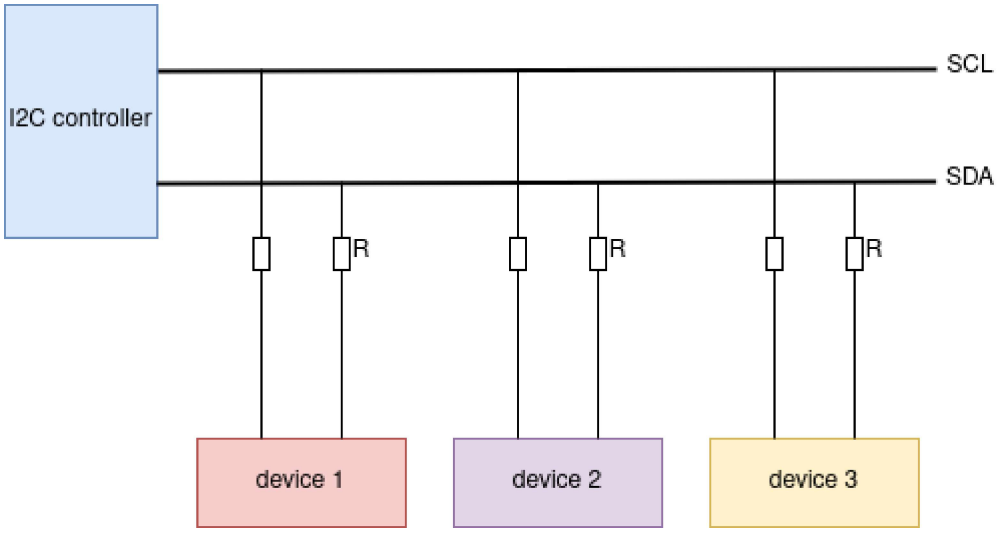
- Resource Allocation - covers how to allocate I2C bus with properly set of configurations. It also covers how to recycle the resources when they finished working.
- I2C Master Controller - covers behavior of I2C master controller. Introduce data transmit, data receive, and data transmit and receive.
- I2C Slave Controller - covers behavior of I2C slave controller. Involve data transmit and data receive.
- Power Management - describes how different source clock will affect power consumption.
- IRAM Safe - describes tips on how to make the I2C interrupt work better along with a disabled cache.
- Thread Safety - lists which APIs are guaranteed to be thread safe by the driver.
- Kconfig Options - lists the supported Kconfig options that can bring different effects to the driver.

Resource Allocation

Both I2C master bus and I2C slave bus, when supported, are represented by `i2c_bus_handle_t` in the driver. The available ports are managed in a resource pool that allocates a free port on request.

Install I2C master bus and device

The I2C master is designed based on bus-device model. So `i2c_master_bus_config_t` and `i2c_device_config_t` are required separately to allocate the I2C master bus instance and I2C device instance.



I2C master bus-device module

I2C master bus requires the configuration that specified by `i2c_master_bus_config_t`:

- `i2c_master_bus_config_t::i2c_port` sets the I2C port used by the controller.
- `i2c_master_bus_config_t::sda_io_num` sets the GPIO number for the serial data bus (SDA).
- `i2c_master_bus_config_t::scl_io_num` sets the GPIO number for the serial clock bus (SCL).
- `i2c_master_bus_config_t::clk_source` selects the source clock for I2C bus. The available clocks are listed in `i2c_clock_source_t`. For the effect on power consumption of different clock source, please refer to Power Management section.
- `i2c_master_bus_config_t::glitch_ignore_cnt` sets the glitch period of master bus, if the glitch period on the line is less than this value, it can be filtered out, typically value is 7.
- `i2c_master_bus_config_t::intr_priority` Set the priority of the interrupt. If set to `0`, then the driver will use a interrupt with low or medium priority (priority level may be one of 1,2 or 3), otherwise use the priority indicated by `i2c_master_bus_config_t::intr_priority` Please use the number form (1,2,3) , not the bitmask form ((1<<1),(1<<2),(1<<3)).
- `i2c_master_bus_config_t::trans_queue_depth` Depth of internal transfer queue. Only valid in asynchronous transaction.
- `i2c_master_bus_config_t::enable_internal_pullup` Enable internal pullups. Note: This is not strong enough to pullup buses under high-speed frequency. A suitable external pullup is recommended.

If the configurations in `i2c_master_bus_config_t` is specified, users can call `i2c_new_master_bus()` to allocate and initialize an I2C master bus. This function will return an I2C bus handle if it runs correctly. Specifically, when there are no more I2C port available, this function will return `ESP_ERR_NOT_FOUND` error.

I2C master device requires the configuration that specified by `i2c_device_config_t`:

- `i2c_device_config_t::dev_addr_length` configure the address bit length of the slave device. User can choose from enumerator `I2C_ADDR_BIT_LEN_7` or `I2C_ADDR_BIT_LEN_10` (if supported).
- `i2c_device_config_t::device_address` I2C device raw address. Please parse the device address to this member directly. For example, the device address is 0x28, then parse 0x28 to `i2c_device_config_t::device_address`, don't carry a write/read bit.
- `i2c_device_config_t::scl_speed_hz` set the scl line frequency of this device.
- `i2c_device_config_t::scl_wait_us`. SCL await time (in us). Usually this value should not be very small because slave stretch will happen in pretty long time. (It's possible even stretch for 12ms). Set 0 means use default reg value.

Once the `i2c_device_config_t` structure is populated with mandatory parameters, users can call `i2c_master_bus_add_device()` to allocate an I2C device instance and mounted to the master bus then. This function will return an I2C device handle if it runs correctly. Specifically, when the I2C bus is not initialized properly, calling this function will result in a `ESP_ERR_INVALID_ARG` error.

```c
#include "driver/i2c_master.h"

i2c_master_bus_config_t i2c_mst_config = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
    .flags.enable_internal_pullup = true,
};

i2c_master_bus_handle_t bus_handle;
ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));

i2c_device_config_t dev_cfg = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = 0x58,
    .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg, &dev_handle));
```

Uninstall I2C master bus and device

If a previously installed I2C bus or device is no longer needed, it's recommended to recycle the resource by calling `i2c_master_bus_rm_device()` or `i2c_del_master_bus()`, so that to release the underlying hardware.

Install I2C slave device

I2C slave requires the configuration that specified by `i2c_slave_config_t`:

- `i2c_slave_config_t::i2c_port` sets the I2C port used by the controller.
- `i2c_slave_config_t::sda_io_num` sets the GPIO number for serial data bus (SDA).
- `i2c_slave_config_t::scl_io_num` sets the GPIO number for serial clock bus (SCL).
- `i2c_slave_config_t::clk_source` selects the source clock for I2C bus. The available clocks are listed in `i2c_clock_source_t`. For the effect on power consumption of different clock source, please refer to Power Management section.
- `i2c_slave_config_t::send_buf_depth` sets the sending buffer length.
- `i2c_slave_config_t::slave_addr` sets the slave address
- `i2c_master_bus_config_t::intr_priority` Set the priority of the interrupt. If set to `0`, then the driver will use a interrupt with low or medium priority (priority level may be one of 1,2 or 3), otherwise use the priority indicated by `i2c_master_bus_config_t::intr_priority` Please use the number form (1,2,3) , not the bitmask form ((1<<1),(1<<2),(1<<3)). Please pay attention that once the interrupt priority is set, it cannot be changed until `i2c_del_master_bus()` is called.
- `i2c_slave_config_t::addr_bit_len` sets true if you need the slave to have a 10-bit address.

- `i2c_slave_config_t::access_ram_en` Set true to enable the non-fifo mode. Thus the I2C data fifo can be used as RAM, and double addressing will be synchronised opened.

Once the `i2c_slave_config_t` structure is populated with mandatory parameters, users can call `i2c_new_slave_device()` to allocate and initialize an I2C master bus. This function will return an I2C bus handle if it runs correctly. Specifically, when there are no more I2C port available, this function will return `ESP_ERR_NOT_FOUND` error.

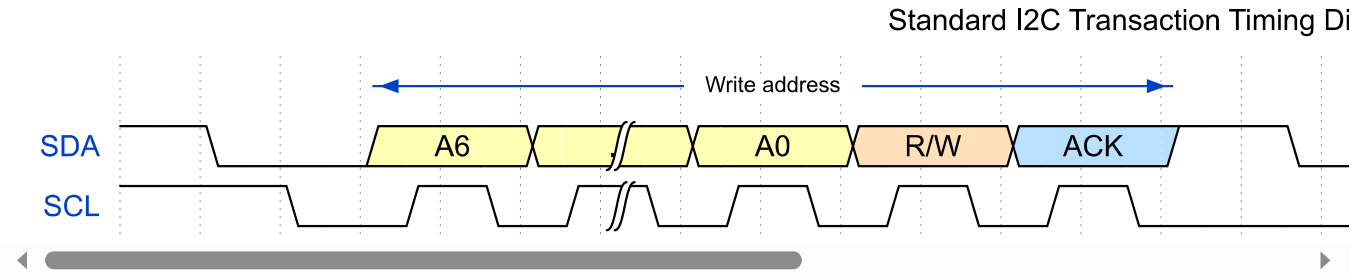
```
i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .send_buf_depth = 256,
    .scl_io_num = I2C_SLAVE_SCL_IO,
    .sda_io_num = I2C_SLAVE_SDA_IO,
    .slave_addr = 0x58,
};

i2c_slave_dev_handle_t slave_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &slave_handle));
```

Uninstall I2C slave device

If a previously installed I2C bus is no longer needed, it's recommended to recycle the resource by calling `i2c_del_slave_device()`, so that to release the underlying hardware.

I2C Master Controller

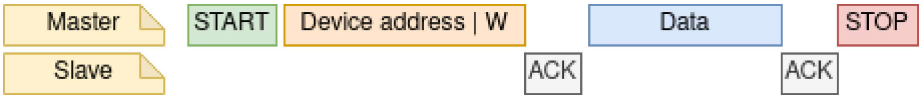
After installing the i2c master driver by `i2c_new_master_bus()`, ESP32-S3 is ready to communicate with other I2C devices. I2C APIs allow the standard transactions. Like the wave as follows:



I2C Master Write

After installing I2C master bus successfully, you can simply call `i2c_master_transmit()` to write data to the slave device. The principle of this function can be explained by following chart.

In order to organize the process, the driver uses a command link, that should be populated with a sequence of commands and then passed to I2C controller for execution.



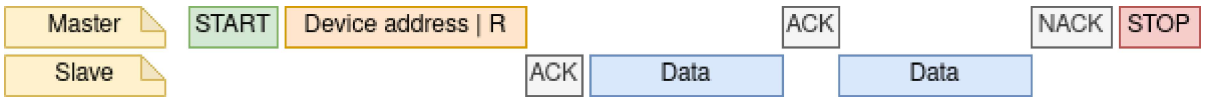
I2C master write to slave

Simple example for writing data to slave:

```
#define DATA_LENGTH 100
i2c_master_bus_config_t i2c_mst_config = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = I2C_PORT_NUM_0,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
};
i2c_master_bus_handle_t bus_handle;

ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));

i2c_device_config_t dev_cfg = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = 0x58,
    .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg, &dev_handle));

ESP_ERROR_CHECK(i2c_master_transmit(dev_handle, data_wr, DATA_LENGTH, -1));
```

I2C Master Read

After installing I2C master bus successfully, you can simply call `i2c_master_receive()` to read data from the slave device. The principle of this function can be explained by following chart.



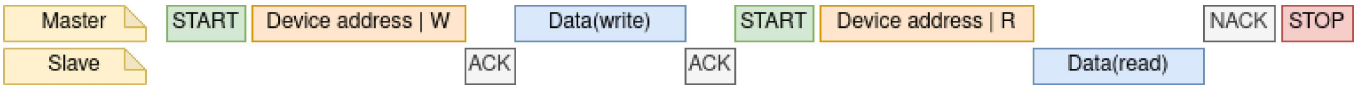
I2C master read from slave

Simple example for reading data from slave:

```
#define DATA_LENGTH 100
i2c_master_bus_config_t i2c_mst_config = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = I2C_PORT_NUM_0,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
};
i2c_master_bus_handle_t bus_handle;

ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));

i2c_device_config_t dev_cfg = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = 0x58,
    .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg, &dev_handle));

i2c_master_receive(dev_handle, data_rd, DATA_LENGTH, -1);
```

I2C Master Write and Read

Some I2C device needs write configurations before reading data from it, therefore, an interface called `i2c_master_transmit_receive()` can help. The principle of this function can be explained by following chart.



I2C master write to slave and read from slave

Simple example for writing and reading from slave:

```
i2c_device_config_t dev_cfg = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = 0x58,
    .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(I2C_PORT_NUM_0, &dev_cfg, &dev_handle));
uint8_t buf[20] = {0x20};
uint8_t buffer[2];
ESP_ERROR_CHECK(i2c_master_transmit_receive(dev_handle, buf, sizeof(buf), buffer, 2, -1));
```

I2C Master Probe

I2C driver can use `i2c_master_probe()` to detect whether the specific device has been connected on I2C bus. If this function return `ESP_OK`, that means the device has been detected.

> **Important**
>
> Pull-ups must be connected to the SCL and SDA pins when this function is called. If you get *ESP_ERR_TIMEOUT* while *xfer_timeout_ms* was parsed correctly, you should check the pull-up resistors. If you do not have proper resistors nearby, setting *flags.enable_internal_pullup* as true is also acceptable.

I2C master probe

Simple example for probing an I2C device:

```
i2c_master_bus_config_t i2c_mst_config_1 = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
    .flags.enable_internal_pullup = true,
};
i2c_master_bus_handle_t bus_handle;

ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config_1, &bus_handle));
ESP_ERROR_CHECK(i2c_master_probe(bus_handle, 0x22, -1));
ESP_ERROR_CHECK(i2c_del_master_bus(bus_handle));
```

I2C Slave Controller

After installing the i2c slave driver by `i2c_new_slave_device()`, ESP32-S3 is ready to communicate with other I2C master as a slave.

I2C Slave Write

The send buffer of the I2C slave is used as a FIFO to store the data to be sent. The data will queue up until the master requests them. You can call `i2c_slave_transmit()` to transfer data.

Simple example for writing data to FIFO:

```
uint8_t *data_wr = (uint8_t *) malloc(DATA_LENGTH);

i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,   // 7-bit address
    .clk_source = I2C_CLK_SRC_DEFAULT,    // set the clock source
    .i2c_port = 0,                        // set I2C port number
    .send_buf_depth = 256,                // set tx buffer length
    .scl_io_num = 2,                      // SCL gpio number
    .sda_io_num = 1,                      // SDA gpio number
    .slave_addr = 0x58,                   // slave address
};

i2c_bus_handle_t i2c_bus_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &i2c_bus_handle));
for (int i = 0; i < DATA_LENGTH; i++) {
    data_wr[i] = i;
}

ESP_ERROR_CHECK(i2c_slave_transmit(i2c_bus_handle, data_wr, DATA_LENGTH, 10000));
```

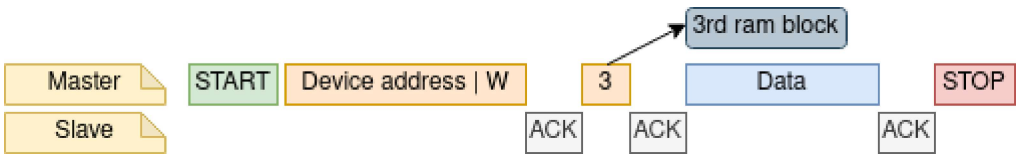
I2C Slave Read

Whenever the master writes data to the slave, the slave will automatically store data in the receive buffer. This allows the slave application to call the function `i2c_slave_receive()` as its own discretion. As `i2c_slave_receive()` is designed as a non-blocking interface. So the user needs to register callback `i2c_slave_register_event_callbacks()` to know when the receive has finished.

```c
static IRAM_ATTR bool i2c_slave_rx_done_callback(i2c_slave_dev_handle_t channel, const
i2c_slave_rx_done_event_data_t *edata, void *user_data)
{
    BaseType_t high_task_wakeup = pdFALSE;
    QueueHandle_t receive_queue = (QueueHandle_t)user_data;
    xQueueSendFromISR(receive_queue, edata, &high_task_wakeup);
    return high_task_wakeup == pdTRUE;
}

uint8_t *data_rd = (uint8_t *) malloc(DATA_LENGTH);
uint32_t size_rd = 0;

i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .send_buf_depth = 256,
    .scl_io_num = I2C_SLAVE_SCL_IO,
    .sda_io_num = I2C_SLAVE_SDA_IO,
    .slave_addr = 0x58,
};

i2c_slave_dev_handle_t slave_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &slave_handle));

s_receive_queue = xQueueCreate(1, sizeof(i2c_slave_rx_done_event_data_t));
i2c_slave_event_callbacks_t cbs = {
    .on_recv_done = i2c_slave_rx_done_callback,
};
ESP_ERROR_CHECK(i2c_slave_register_event_callbacks(slave_handle, &cbs, s_receive_queue));

i2c_slave_rx_done_event_data_t rx_data;
ESP_ERROR_CHECK(i2c_slave_receive(slave_handle, data_rd, DATA_LENGTH));
xQueueReceive(s_receive_queue, &rx_data, pdMS_TO_TICKS(10000));
// Receive done.
```

Put Data In I2C Slave RAM

I2C slave fifo mentioned above can be used as RAM, which means user can access the RAM directly via address fields. For example, writing data to the 3rd ram block with following graph. Before using this, please note that `i2c_slave_config_t::access_ram_en` needs to be set to true.

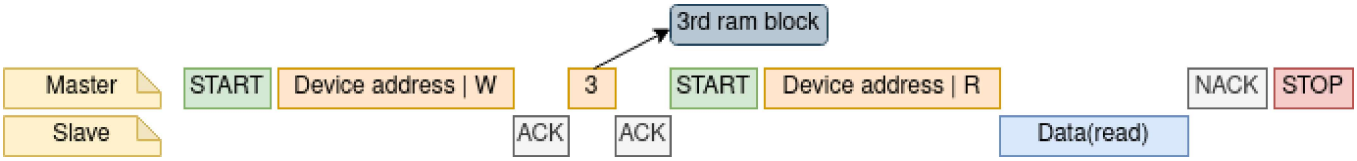


Put data in I2C slave RAM

```
uint8_t data_rd[DATA_LENGTH_RAM] = {0};

i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .send_buf_depth = 256,
    .scl_io_num = I2C_SLAVE_SCL_IO,
    .sda_io_num = I2C_SLAVE_SDA_IO,
    .slave_addr = 0x58,
    .flags.access_ram_en = true,
};

// Master write to slave.

i2c_slave_dev_handle_t slave_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &slave_handle));
ESP_ERROR_CHECK(i2c_slave_read_ram(slave_handle, 0x5, data_rd, DATA_LENGTH_RAM));
ESP_ERROR_CHECK(i2c_del_slave_device(slave_handle));
```

Get Data From I2C Slave RAM

Data can be stored in the RAM with a specific offset by the slave controller, and the master can read this data directly via the RAM address. For example, if the data is stored in 3rd ram block, master can read this data by following graph. Before using this, please note that `i2c_slave_config_t::access_ram_en` needs to be set to true.



Get data from I2C slave RAM

```
uint8_t data_wr[DATA_LENGTH_RAM] = {0};

i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .send_buf_depth = 256,
    .scl_io_num = I2C_SLAVE_SCL_IO,
    .sda_io_num = I2C_SLAVE_SDA_IO,
    .slave_addr = 0x58,
    .flags.access_ram_en = true,
};

i2c_slave_dev_handle_t slave_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &slave_handle));
ESP_ERROR_CHECK(i2c_slave_write_ram(slave_handle, 0x2, data_wr, DATA_LENGTH_RAM));
ESP_ERROR_CHECK(i2c_del_slave_device(slave_handle));
```

Register Event Callbacks

I2C master callbacks

When an I2C master bus triggers an interrupt, a specific event will be generated and notify the CPU. If you have some functions that need to be called when those events occurred, you can hook your functions to the ISR (Interrupt Service Routine) by calling `i2c_master_register_event_callbacks()`. Since the registered callback functions are called in the interrupt context, user should ensure the callback function doesn't attempt to block (e.g. by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The callback functions are required to return a boolean value, to tell the ISR whether a high priority task is woke up by it.

I2C master event callbacks are listed in the `i2c_master_event_callbacks_t`.

Although I2C is a synchronous communication protocol, we also support asynchronous behavior by registering above callback. In this way, I2C APIs will be non-blocking interface. But note that on the same bus, only one device can adopt asynchronous operation.

> **Important**
>
> I2C master asynchronous transaction is still an experimental feature. (The issue is when asynchronous transaction is very large, it will cause memory problem.)

- `i2c_master_event_callbacks_t::on_recv_done` sets a callback function for master "transaction-done" event. The function prototype is declared in `i2c_master_callback_t`.

I2C slave callbacks

When an I2C slave bus triggers an interrupt, a specific event will be generated and notify the CPU. If you have some function that needs to be called when those events occurred, you can hook your function to the ISR (Interrupt Service Routine) by calling `i2c_slave_register_event_callbacks()`. Since the registered callback functions are called in the interrupt context, user should ensure the callback function doesn't attempt to block (e.g. by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The callback function has a boolean return value, to tell the caller whether a high priority task is woke up by it.

I2C slave event callbacks are listed in the `i2c_slave_event_callbacks_t`.

- `i2c_slave_event_callbacks_t::on_recv_done` sets a callback function for "receive-done" event. The function prototype is declared in `i2c_slave_received_callback_t`.

Power Management

If the controller clock source is selected to `I2C_CLK_SRC_XTAL`, then the driver won't install power management lock for it, which is more suitable for a low power application as long as the source clock can still provide sufficient resolution.

IRAM Safe

By default, the I2C interrupt will be deferred when the Cache is disabled for reasons like writing/erasing Flash. Thus the event callback functions will not get executed in time, which is not expected in a real-time application.

There's a Kconfig option CONFIG_I2C_ISR_IRAM_SAFE that will:

1. Enable the interrupt being serviced even when cache is disabled
2. Place all functions that used by the ISR into IRAM
3. Place driver object into DRAM (in case it's mapped to PSRAM by accident)

This will allow the interrupt to run while the cache is disabled but will come at the cost of increased IRAM consumption.

Thread Safety

The factory function `i2c_new_master_bus()` and `i2c_new_slave_device()` are guaranteed to be thread safe by the driver, which means, user can call them from different RTOS tasks without protection by extra locks. Other public I2C APIs are not thread safe. which means the user should avoid calling them from multiple tasks, if user strongly needs to call them in multiple tasks, please add extra lock.

Kconfig Options

- CONFIG_I2C_ISR_IRAM_SAFE controls whether the default ISR handler can work when cache is disabled, see also IRAM Safe for more information.
- CONFIG_I2C_ENABLE_DEBUG_LOG is used to enable the debug log at the cost of increased firmware binary size.

API Reference

Header File

- components/esp_driver_i2c/include/driver/i2c_master.h
- This header file can be included with:

```
#include "driver/i2c_master.h"
```

- This header file is a part of the API provided by the `esp_driver_i2c` component. To declare that your component depends on `esp_driver_i2c`, add the following to your CMakeLists.txt:

```
REQUIRES esp_driver_i2c
```

or

```
PRIV_REQUIRES esp_driver_i2c
```

Functions

esp_err_t **i2c_new_master_bus**(*const* i2c_master_bus_config_t *bus_config, i2c_master_bus_handle_t *ret_bus_handle)

Allocate an I2C master bus.

Parameters:
- **bus_config** -- **[in]** I2C master bus configuration.
- **ret_bus_handle** -- **[out]** I2C bus handle

Returns:
- ESP_OK: I2C master bus initialized successfully.
- ESP_ERR_INVALID_ARG: I2C bus initialization failed because of invalid argument.
- ESP_ERR_NO_MEM: Create I2C bus failed because of out of memory.
- ESP_ERR_NOT_FOUND: No more free bus.

esp_err_t **i2c_master_bus_add_device**(i2c_master_bus_handle_t bus_handle, *const* i2c_device_config_t *dev_config, i2c_master_dev_handle_t *ret_handle)

Add I2C master BUS device.

Parameters:
- **bus_handle** -- [in] I2C bus handle.
- **dev_config** -- [in] device config.
- **ret_handle** -- [out] device handle.

Returns:
- ESP_OK: Create I2C master device successfully.
- ESP_ERR_INVALID_ARG: I2C bus initialization failed because of invalid argument.
- ESP_ERR_NO_MEM: Create I2C bus failed because of out of memory.

---

esp_err_t **i2c_del_master_bus**(i2c_master_bus_handle_t bus_handle)

Deinitialize the I2C master bus and delete the handle.

Parameters: **bus_handle** -- [in] I2C bus handle.

Returns:
- ESP_OK: Delete I2C bus success, otherwise, failed.
- Otherwise: Some module delete failed.

---

esp_err_t **i2c_master_bus_rm_device**(i2c_master_dev_handle_t handle)

I2C master bus delete device.

Parameters: **handle** -- i2c device handle

Returns:
- ESP_OK: If device is successfully deleted.

---

esp_err_t **i2c_master_transmit**(i2c_master_dev_handle_t i2c_dev, *const* uint8_t *write_buffer, size_t write_size, int xfer_timeout_ms)

Perform a write transaction on the I2C bus. The transaction will be undergoing until it finishes or it reaches the timeout provided.

Note

If a callback was registered with `i2c_master_register_event_callbacks`, the transaction will be asynchronous, and thus, this function will return directly, without blocking. You will get finish information from callback. Besides, data buffer should always be completely prepared when callback is registered, otherwise, the data will get corrupt.

Parameters:
- **i2c_dev** -- [in] I2C master device handle that created by `i2c_master_bus_add_device`.
- **write_buffer** -- [in] Data bytes to send on the I2C bus.
- **write_size** -- [in] Size, in bytes, of the write buffer.

- **xfer_timeout_ms** -- **[in]** Wait timeout, in ms. Note: -1 means wait forever.

Returns:

- ESP_OK: I2C master transmit success
- ESP_ERR_INVALID_ARG: I2C master transmit parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.

---

esp_err_t **i2c_master_transmit_receive**(i2c_master_dev_handle_t i2c_dev, *const* uint8_t *write_buffer, size_t write_size, uint8_t *read_buffer, size_t read_size, int xfer_timeout_ms)

Perform a write-read transaction on the I2C bus. The transaction will be undergoing until it finishes or it reaches the timeout provided.

Note

If a callback was registered with `i2c_master_register_event_callbacks`, the transaction will be asynchronous, and thus, this function will return directly, without blocking. You will get finish information from callback. Besides, data buffer should always be completely prepared when callback is registered, otherwise, the data will get corrupt.

Parameters:

- **i2c_dev** -- **[in]** I2C master device handle that created by `i2c_master_bus_add_device`.
- **write_buffer** -- **[in]** Data bytes to send on the I2C bus.
- **write_size** -- **[in]** Size, in bytes, of the write buffer.
- **read_buffer** -- **[out]** Data bytes received from i2c bus.
- **read_size** -- **[in]** Size, in bytes, of the read buffer.
- **xfer_timeout_ms** -- **[in]** Wait timeout, in ms. Note: -1 means wait forever.

Returns:

- ESP_OK: I2C master transmit-receive success
- ESP_ERR_INVALID_ARG: I2C master transmit parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.

---

esp_err_t **i2c_master_receive**(i2c_master_dev_handle_t i2c_dev, uint8_t *read_buffer, size_t read_size, int xfer_timeout_ms)

Perform a read transaction on the I2C bus. The transaction will be undergoing until it finishes or it reaches the timeout provided.

Note

If a callback was registered with `i2c_master_register_event_callbacks`, the transaction will be asynchronous, and thus, this function will return directly, without blocking. You will get finish information from callback. Besides, data buffer should always be completely prepared when callback is registered, otherwise, the data will get corrupt.

Parameters:

- **i2c_dev** -- **[in]** I2C master device handle that created by `i2c_master_bus_add_device`.
- **read_buffer** -- **[out]** Data bytes received from i2c bus.
- **read_size** -- **[in]** Size, in bytes, of the read buffer.
- **xfer_timeout_ms** -- **[in]** Wait timeout, in ms. Note: -1 means wait forever.

Returns:

- ESP_OK: I2C master receive success
- ESP_ERR_INVALID_ARG: I2C master receive parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.

---

esp_err_t **i2c_master_probe**(i2c_master_bus_handle_t bus_handle, uint16_t address, int xfer_timeout_ms)

Probe I2C address, if address is correct and ACK is received, this function will return ESP_OK.

> **Attention**
>
> Pull-ups must be connected to the SCL and SDA pins when this function is called. If you get `ESP_ERR_TIMEOUT while` xfer_timeout_ms `was parsed correctly, you should check the pull-up resistors. If you do not have proper resistors nearby.` flags.enable_internal_pullup` is also acceptable.

Note

The principle of this function is to sent device address with a write command. If the device on your I2C bus, there would be an ACK signal and function returns `ESP_OK`. If the device is not on your I2C bus, there would be a NACK signal and function returns `ESP_ERR_NOT_FOUND`. `ESP_ERR_TIMEOUT` is not an expected failure, which indicated that the i2c probe not works properly, usually caused by pull-up resistors not be connected properly. Suggestion check data on SDA/SCL line to see whether there is ACK/NACK signal is on line when i2c probe function fails.

Note

There are lots of I2C devices all over the world, we assume that not all I2C device support the behavior like `device_address+nack/ack`. So, if the on line data is strange and no ack/nack got respond. Please check the device datasheet.

Parameters:

- **bus_handle** -- **[in]** I2C master device handle that created by `i2c_master_bus_add_device`.
- **address** -- **[in]** I2C device address that you want to probe.
- **xfer_timeout_ms** -- **[in]** Wait timeout, in ms. Note: -1 means wait forever (Not recommended in this function).

Returns:

- ESP_OK: I2C device probe successfully
- ESP_ERR_NOT_FOUND: I2C probe failed, doesn't find the device with specific address you gave.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.

---

esp_err_t **i2c_master_register_event_callbacks**(i2c_master_dev_handle_t i2c_dev, *const* i2c_master_event_callbacks_t *cbs, void *user_data)

Register I2C transaction callbacks for a master device.

Note

User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Note

When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

Note

If the callback is used for helping asynchronous transaction. On the same bus, only one device can be used for performing asynchronous operation.

Parameters:

- **i2c_dev** -- **[in]** I2C master device handle that created by `i2c_master_bus_add_device`.
- **cbs** -- **[in]** Group of callback functions

- **user_data** -- **[in]** User data, which will be passed to callback functions directly

Returns:

- ESP_OK: Set I2C transaction callbacks successfully
- ESP_ERR_INVALID_ARG: Set I2C transaction callbacks failed because of invalid argument
- ESP_FAIL: Set I2C transaction callbacks failed because of other error

---

esp_err_t **i2c_master_bus_reset**(i2c_master_bus_handle_t bus_handle)

Reset the I2C master bus.

Parameters: **bus_handle** -- I2C bus handle.

Returns:

- ESP_OK: Reset succeed.
- ESP_ERR_INVALID_ARG: I2C master bus handle is not initialized.
- Otherwise: Reset failed.

---

esp_err_t **i2c_master_bus_wait_all_done**(i2c_master_bus_handle_t bus_handle, int timeout_ms)

Wait for all pending I2C transactions done.

Parameters:

- **bus_handle** -- **[in]** I2C bus handle
- **timeout_ms** -- **[in]** Wait timeout, in ms. Specially, -1 means to wait forever.

Returns:

- ESP_OK: Flush transactions successfully
- ESP_ERR_INVALID_ARG: Flush transactions failed because of invalid argument
- ESP_ERR_TIMEOUT: Flush transactions failed because of timeout
- ESP_FAIL: Flush transactions failed because of other error

Structures

---

struct **i2c_master_bus_config_t**

I2C master bus specific configurations.

Public Members

i2c_port_num_t **i2c_port**

I2C port number, `-1` for auto selecting, (not include LP I2C instance)

gpio_num_t **sda_io_num**

GPIO number of I2C SDA signal, pulled-up internally

gpio_num_t **scl_io_num**

GPIO number of I2C SCL signal, pulled-up internally

i2c_clock_source_t **clk_source**

Clock source of I2C master bus

uint8_t **glitch_ignore_cnt**

If the glitch period on the line is less than this value, it can be filtered out, typically value is 7 (unit: I2C module clock cycle)

int **intr_priority**

I2C interrupt priority, if set to 0, driver will select the default priority (1,2,3).

size_t **trans_queue_depth**

Depth of internal transfer queue, increase this value can support more transfers pending in the background, only valid in asynchronous transaction. (Typically max_device_num * per_transaction)

uint32_t **enable_internal_pullup**

Enable internal pullups. Note: This is not strong enough to pullup buses under high-speed frequency. Recommend proper external pull-up if possible

struct i2c_master_bus_config_t::[anonymous] **flags**

I2C master config flags

---

struct **i2c_device_config_t**

I2C device configuration.

Public Members

i2c_addr_bit_len_t **dev_addr_length**

Select the address length of the slave device.

uint16_t **device_address**

I2C device raw address. (The 7/10 bit address without read/write bit)

uint32_t **scl_speed_hz**

I2C SCL line frequency.

uint32_t **scl_wait_us**

Timeout value. (unit: us). Please note this value should not be so small that it can handle stretch/disturbance properly. If 0 is set, that means use the default reg value

uint32_t **disable_ack_check**

Disable ACK check. If this is set false, that means ack check is enabled, the transaction will be stopped and API returns error when nack is detected.

struct i2c_device_config_t::[anonymous] **flags**

I2C device config flags

struct **i2c_master_event_callbacks_t**

Group of I2C master callbacks, can be used to get status during transaction or doing other small things. But take care potential concurrency issues.

Note

The callbacks are all running under ISR context

Note

When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

i2c_master_callback_t **on_trans_done**

I2C master transaction finish callback

Header File

- components/esp_driver_i2c/include/driver/i2c_slave.h
- This header file can be included with:

```
#include "driver/i2c_slave.h"
```

- This header file is a part of the API provided by the `esp_driver_i2c` component. To declare that your component depends on `esp_driver_i2c`, add the following to your CMakeLists.txt:

```
REQUIRES esp_driver_i2c
```

or

```
PRIV_REQUIRES esp_driver_i2c
```

Functions

esp_err_t **i2c_new_slave_device**(*const* i2c_slave_config_t *slave_config, i2c_slave_dev_handle_t *ret_handle)

Initialize an I2C slave device.

Parameters:
- **slave_config** -- **[in]** I2C slave device configurations
- **ret_handle** -- **[out]** Return a generic I2C device handle

Returns:
- ESP_OK: I2C slave device initialized successfully
- ESP_ERR_INVALID_ARG: I2C device initialization failed because of invalid argument.
- ESP_ERR_NO_MEM: Create I2C device failed because of out of memory.

esp_err_t **i2c_del_slave_device**(i2c_slave_dev_handle_t i2c_slave)

Deinitialize the I2C slave device.

Parameters: **i2c_slave** -- **[in]** I2C slave device handle that created by `i2c_new_slave_device`.

Returns:
- ESP_OK: Delete I2C device successfully.
- ESP_ERR_INVALID_ARG: I2C device initialization failed because of invalid argument.

esp_err_t **i2c_slave_receive**(i2c_slave_dev_handle_t i2c_slave, uint8_t *data, size_t buffer_size)

Read bytes from I2C internal buffer. Start a job to receive I2C data.

Note

This function is non-blocking, it initiates a new receive job and then returns. User should check the received data from the `on_recv_done` callback that registered by `i2c_slave_register_event_callbacks()`.

Parameters:

- **i2c_slave** -- **[in]** I2C slave device handle that created by `i2c_new_slave_device`.
- **data** -- **[out]** Buffer to store data from I2C fifo. Should be valid until `on_recv_done` is triggered.
- **buffer_size** -- **[in]** Buffer size of data that provided by users.

Returns:

- ESP_OK: I2C slave receive success.
- ESP_ERR_INVALID_ARG: I2C slave receive parameter invalid.
- ESP_ERR_NOT_SUPPORTED: This function should be work in fifo mode, but I2C_SLAVE_NONFIFO mode is configured

---

esp_err_t **i2c_slave_transmit**(i2c_slave_dev_handle_t i2c_slave, *const* uint8_t *data, int size, int xfer_timeout_ms)

Write bytes to internal ringbuffer of the I2C slave data. When the TX fifo empty, the ISR will fill the hardware FIFO with the internal ringbuffer's data.

Note

If you connect this slave device to some master device, the data transaction direction is from slave device to master device.

Parameters:

- **i2c_slave** -- **[in]** I2C slave device handle that created by `i2c_new_slave_device`.
- **data** -- **[in]** Buffer to write to slave fifo, can pickup by master. Can be freed after this function returns. Equal or larger than `size`.
- **size** -- **[in]** In bytes, of `data` buffer.
- **xfer_timeout_ms** -- **[in]** Wait timeout, in ms. Note: -1 means wait forever.

Returns:

- ESP_OK: I2C slave transmit success.
- ESP_ERR_INVALID_ARG: I2C slave transmit parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the device is busy or hardware crash.
- ESP_ERR_NOT_SUPPORTED: This function should be work in fifo mode, but I2C_SLAVE_NONFIFO mode is configured

esp_err_t **i2c_slave_register_event_callbacks**(i2c_slave_dev_handle_t i2c_slave, *const* i2c_slave_event_callbacks_t *cbs, void *user_data)

Set I2C slave event callbacks for I2C slave channel.

Note

User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

Note

When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

Parameters:

- **i2c_slave** -- **[in]** I2C slave device handle that created by `i2c_new_slave_device`.
- **cbs** -- **[in]** Group of callback functions
- **user_data** -- **[in]** User data, which will be passed to callback functions directly

Returns:

- ESP_OK: Set I2C transaction callbacks successfully
- ESP_ERR_INVALID_ARG: Set I2C transaction callbacks failed because of invalid argument
- ESP_FAIL: Set I2C transaction callbacks failed because of other error

esp_err_t **i2c_slave_read_ram**(i2c_slave_dev_handle_t i2c_slave, uint8_t ram_address, uint8_t *data, size_t receive_size)

Read bytes from I2C internal ram. This can be only used when `access_ram_en` in configuration structure set to true.

Parameters:

- **i2c_slave** -- **[in]** I2C slave device handle that created by `i2c_new_slave_device`.
- **ram_address** -- **[in]** The offset of RAM (Cannot larger than I2C RAM memory)
- **data** -- **[out]** Buffer to store data read from I2C ram.
- **receive_size** -- **[in]** Received size from RAM.

Returns:

- ESP_OK: I2C slave transmit success.
- ESP_ERR_INVALID_ARG: I2C slave transmit parameter invalid.

- ESP_ERR_NOT_SUPPORTED: This function should be work in non-fifo mode, but I2C_SLAVE_FIFO mode is configured

---

esp_err_t **i2c_slave_write_ram**(i2c_slave_dev_handle_t i2c_slave, uint8_t ram_address, *const* uint8_t *data, size_t size)

Write bytes to I2C internal ram. This can be only used when `access_ram_en` in configuration structure set to true.

Parameters:

- **i2c_slave** -- **[in]** I2C slave device handle that created by `i2c_new_slave_device`.
- **ram_address** -- **[in]** The offset of RAM (Cannot larger than I2C RAM memory)
- **data** -- **[in]** Buffer to fill.
- **size** -- **[in]** Received size from RAM.

Returns:

- ESP_OK: I2C slave transmit success.
- ESP_ERR_INVALID_ARG: I2C slave transmit parameter invalid.
- ESP_ERR_INVALID_SIZE: Write size is larger than
- ESP_ERR_NOT_SUPPORTED: This function should be work in non-fifo mode, but I2C_SLAVE_FIFO mode is configured

Structures

---

struct **i2c_slave_config_t**

I2C slave specific configurations.

Public Members

i2c_port_num_t **i2c_port**

I2C port number, `-1` for auto selecting

gpio_num_t **sda_io_num**

SDA IO number used by I2C bus

gpio_num_t **scl_io_num**

SCL IO number used by I2C bus

i2c_clock_source_t **clk_source**

Clock source of I2C bus.

uint32_t **send_buf_depth**

Depth of internal transfer ringbuffer, increase this value can support more transfers pending in the background

uint16_t **slave_addr**

I2C slave address

i2c_addr_bit_len_t **addr_bit_len**

I2C slave address in bit length

int **intr_priority**

I2C interrupt priority, if set to 0, driver will select the default priority (1,2,3).

uint32_t **broadcast_en**

I2C slave enable broadcast

uint32_t **access_ram_en**

Can get access to I2C RAM directly

struct i2c_slave_config_t::[anonymous] **flags**

I2C slave config flags

struct `i2c_slave_event_callbacks_t`

Group of I2C slave callbacks (e.g. get i2c slave stretch cause). But take care of potential concurrency issues.

Note

The callbacks are all running under ISR context

Note

When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

i2c_slave_received_callback_t **on_recv_done**

I2C slave receive done callback

Header File

- components/esp_driver_i2c/include/driver/i2c_types.h
- This header file can be included with:

```
#include "driver/i2c_types.h"
```

- This header file is a part of the API provided by the `esp_driver_i2c` component. To declare that your component depends on `esp_driver_i2c`, add the following to your CMakeLists.txt:

```
REQUIRES esp_driver_i2c
```

or

```
PRIV_REQUIRES esp_driver_i2c
```

Structures

struct **i2c_master_event_data_t**

Data type used in I2C event callback.

Public Members

i2c_master_event_t **event**

The I2C hardware event that I2C callback is called.

struct **i2c_slave_rx_done_event_data_t**

Event structure used in I2C slave.

Public Members

uint8_t ***buffer**

Pointer for buffer received in callback.

Type Definitions

typedef int **i2c_port_num_t**

I2C port number.

typedef struct i2c_master_bus_t ***i2c_master_bus_handle_t**

Type of I2C master bus handle.

typedef struct i2c_master_dev_t ***i2c_master_dev_handle_t**

Type of I2C master bus device handle.

typedef struct i2c_slave_dev_t ***i2c_slave_dev_handle_t**

Type of I2C slave device handle.

typedef bool (***i2c_master_callback_t**)(i2c_master_dev_handle_t i2c_dev, *const* i2c_master_event_data_t *evt_data, void *arg)

An callback for I2C transaction.

Param i2c_dev: [in] Handle for I2C device.

Param evt_data: [out] I2C capture event data, fed by driver

Param arg: [in] User data, set in `i2c_master_register_event_callbacks()`

Return: Whether a high priority task has been waken up by this function

typedef bool (***i2c_slave_received_callback_t**)(i2c_slave_dev_handle_t i2c_slave, *const* i2c_slave_rx_done_event_data_t *evt_data, void *arg)

Callback signature for I2C slave.

Param i2c_slave: [in] Handle for I2C slave.

Param evt_data: [out] I2C capture event data, fed by driver

Param arg: [in] User data, set in `i2c_slave_register_event_callbacks()`

Return: Whether a high priority task has been waken up by this function

Enumerations

enum **i2c_master_status_t**

Enumeration for I2C fsm status.

Values:

enumerator **I2C_STATUS_READ**

read status for current master command

enumerator **I2C_STATUS_WRITE**

write status for current master command

enumerator **I2C_STATUS_START**

Start status for current master command

enumerator **I2C_STATUS_STOP**

stop status for current master command

enumerator **I2C_STATUS_IDLE**

idle status for current master command

enumerator **I2C_STATUS_ACK_ERROR**

ack error status for current master command

enumerator **I2C_STATUS_DONE**

I2C command done

enumerator **I2C_STATUS_TIMEOUT**

I2C bus status error, and operation timeout

enum **i2c_master_event_t**

Enumeration for I2C event.

Values:

enumerator **I2C_EVENT_ALIVE**

i2c bus in alive status.

enumerator **I2C_EVENT_DONE**

i2c bus transaction done

enumerator **I2C_EVENT_NACK**

i2c bus nack

enumerator **I2C_EVENT_TIMEOUT**

i2c bus timeout

Header File

- components/hal/include/hal/i2c_types.h
- This header file can be included with:

```
#include "hal/i2c_types.h"
```

Structures

struct **i2c_hal_clk_config_t**

Data structure for calculating I2C bus timing.

Public Members

uint16_t **clkm_div**

I2C core clock divider

uint16_t **scl_low**

I2C scl low period

uint16_t **scl_high**

I2C scl high period

uint16_t **scl_wait_high**

I2C scl wait_high period

uint16_t **sda_hold**

I2C scl low period

uint16_t **sda_sample**

I2C sda sample time

uint16_t **setup**

I2C start and stop condition setup period

uint16_t **hold**

I2C start and stop condition hold period

uint16_t **tout**

I2C bus timeout period

Type Definitions

typedef soc_periph_i2c_clk_src_t **i2c_clock_source_t**

I2C group clock source.

Enumerations

enum **i2c_port_t**

I2C port number, can be I2C_NUM_0 ~ (I2C_NUM_MAX-1).

Values:

enumerator **I2C_NUM_0**

I2C port 0

enumerator **I2C_NUM_1**

I2C port 1

enumerator **I2C_NUM_MAX**

I2C port max

enum **i2c_addr_bit_len_t**

Enumeration for I2C device address bit length.

Values:

enumerator **I2C_ADDR_BIT_LEN_7**

i2c address bit length 7

enumerator **I2C_ADDR_BIT_LEN_10**

i2c address bit length 10

enum **i2c_mode_t**

Values:

enumerator **I2C_MODE_SLAVE**

I2C slave mode

enumerator **I2C_MODE_MASTER**

I2C master mode

enumerator **I2C_MODE_MAX**

enum **i2c_rw_t**

Values:

enumerator **I2C_MASTER_WRITE**

I2C write data

enumerator **I2C_MASTER_READ**

I2C read data

enum **i2c_trans_mode_t**

Values:

enumerator **I2C_DATA_MODE_MSB_FIRST**

I2C data msb first

enumerator **I2C_DATA_MODE_LSB_FIRST**

I2C data lsb first

enumerator **I2C_DATA_MODE_MAX**

enum **i2c_addr_mode_t**

Values:

enumerator **I2C_ADDR_BIT_7**

I2C 7bit address for slave mode

enumerator **I2C_ADDR_BIT_10**

I2C 10bit address for slave mode

enumerator **I2C_ADDR_BIT_MAX**

enum **i2c_ack_type_t**

Values:

enumerator **I2C_MASTER_ACK**

I2C ack for each byte read

enumerator **I2C_MASTER_NACK**

I2C nack for each byte read

enumerator **I2C_MASTER_LAST_NACK**

I2C nack for the last byte

enumerator **I2C_MASTER_ACK_MAX**

enum **i2c_slave_stretch_cause_t**

Enum for I2C slave stretch causes.

Values:

enumerator **I2C_SLAVE_STRETCH_CAUSE_ADDRESS_MATCH**

Stretching SCL low when the slave is read by the master and the address just matched

enumerator **I2C_SLAVE_STRETCH_CAUSE_TX_EMPTY**

Stretching SCL low when TX FIFO is empty in slave mode

enumerator **I2C_SLAVE_STRETCH_CAUSE_RX_FULL**

Stretching SCL low when RX FIFO is full in slave mode

enumerator **I2C_SLAVE_STRETCH_CAUSE_SENDING_ACK**

Stretching SCL low when slave sending ACK