Note

This documentation is not for the latest stable release version. The latest stable version is v5.3.1

Two-Wire Automotive Interface (TWAI)

Overview

The Two-Wire Automotive Interface (TWAI) is a real-time serial communication protocol suited for automotive and industrial applications. It is compatible with ISO11898-1 Classical frames, thus can support Standard Frame Format (11-bit ID) and Extended Frame Format (29-bit ID). The ESP32 contains 1 TWAI controller(s) that can be configured to communicate on a TWAI bus via an external transceiver.

Warning

The TWAI controller is not compatible with ISO11898-1 FD Format frames, and will interpret such frames as errors.

This programming guide is split into the following sections:

Sections

TWAI Protocol Summary

The TWAI is a multi-master, multi-cast, asynchronous, serial communication protocol. TWAI also supports error detection and signalling, and inbuilt message prioritization.

Multi-master: Any node on the bus can initiate the transfer of a message.

Multi-cast: When a node transmits a message, all nodes on the bus will receive the message (i.e., broadcast) thus ensuring data consistency across all nodes. However, some nodes can selectively choose which messages to accept via the use of acceptance filtering (multi-cast).

Asynchronous: The bus does not contain a clock signal. All nodes on the bus operate at the same bit rate and synchronize using the edges of the bits transmitted on the bus.

Error Detection and Signaling: Every node constantly monitors the bus. When any node detects an error, it signals the detection by transmitting an error frame. Other nodes will receive the error frame and transmit their own error frames in response. This results in an error detection being propagated to all nodes on the bus.

Message Priorities: Messages contain an ID field. If two or more nodes attempt to transmit simultaneously, the node transmitting the message with the lower ID value will win arbitration of the bus. All other nodes will become receivers ensuring that there is at most one transmitter at any time.

TWAI Messages

TWAI Messages are split into Data Frames and Remote Frames. Data Frames are used to deliver a data payload to other nodes, whereas a Remote Frame is used to request a Data Frame from other nodes (other nodes can optionally respond with a Data Frame). Data and Remote Frames have two frame formats known as **Extended Frame** and **Standard Frame** which contain a 29-bit ID and an 11-bit ID respectively. A TWAI message consists of the following fields:

- 29-bit or 11-bit ID: Determines the priority of the message (lower value has higher priority).
- Data Length Code (DLC) between 0 to 8: Indicates the size (in bytes) of the data payload for a Data Frame, or the amount of data to request for a Remote Frame.
- Up to 8 bytes of data for a Data Frame (should match DLC).

Error States and Counters

The TWAI protocol implements a feature known as "fault confinement" where a persistently erroneous node will eventually eliminate itself from the bus. This is implemented by requiring every node to maintain two internal error counters known as the **Transmit Error Counter (TEC)** and the **Receive Error Counter (REC)**. The two error counters are incremented and decremented

according to a set of rules (where the counters increase on an error, and decrease on a successful message transmission/reception). The values of the counters are used to determine a node's **error state**, namely **Error Active**, **Error Passive**, and **Bus-Off**.

Error Active: A node is Error Active when **both TEC and REC are less than 128** and indicates that the node is operating normally. Error Active nodes are allowed to participate in bus communications, and will actively signal the detection of any errors by automatically transmitting an **Active Error Flag** over the bus.

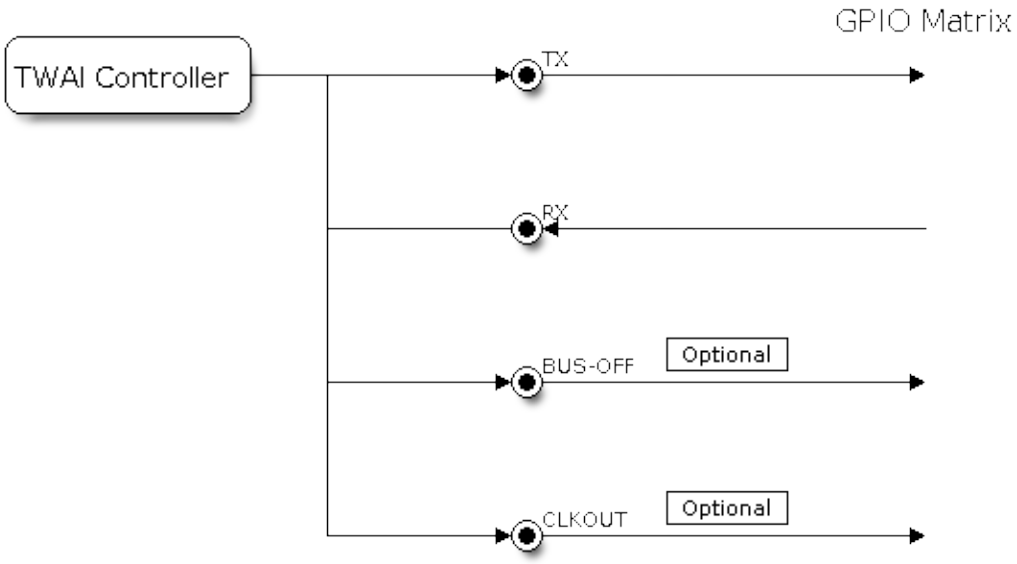
Error Passive: A node is Error Passive when **either the TEC or REC becomes greater than or equal to 128**. Error Passive nodes are still able to take part in bus communications, but will instead transmit a **Passive Error Flag** upon detection of an error.

Bus-Off: A node becomes Bus-Off when the **TEC becomes greater than or equal to 256**. A Bus-Off node is unable influence the bus in any manner (essentially disconnected from the bus) thus eliminating itself from the bus. A node will remain in the Bus-Off state until it undergoes bus-off recovery.

Signals Lines and Transceiver

The TWAI controller does not contain a integrated transceiver. Therefore, to connect the TWAI controller to a TWAI bus, **an external transceiver is required**. The type of external transceiver used should depend on the application's physical layer specification (e.g., using SN65HVD23x transceivers for ISO 11898-2 compatibility).

The TWAI controller's interface consists of 4 signal lines known as **TX, RX, BUS-OFF, and CLKOUT**. These four signal lines can be routed through the GPIO Matrix to the ESP32's GPIO pads.



Signal lines of the TWAI controller

TX and RX: The TX and RX signal lines are required to interface with an external transceiver. Both signal lines represent/interpret a dominant bit as a low logic level (0 V), and a recessive bit as a high logic level (3.3 V).

BUS-OFF: The BUS-OFF signal line is **optional** and is set to a low logic level (0 V) whenever the TWAI controller reaches a bus-off state. The BUS-OFF signal line is set to a high logic level (3.3 V) otherwise.

CLKOUT: The CLKOUT signal line is **optional** and outputs a prescaled version of the controller's source clock.

Note

An external transceiver **must internally loopback the TX to RX** such that a change in logic level to the TX signal line can be observed on the RX line. Failing to do so will cause the TWAI controller to interpret differences in logic levels between the two signal lines as a loss in arbitration or a bit error.

API Naming Conventions

Note

The TWAI driver provides two sets of API. One is handle-free and is widely used in IDF versions earlier than v5.2, but it can only support one TWAI hardware controller. The other set is with handles, and the function name is usually suffixed with "v2", which can support

any number of TWAI controllers. These two sets of API can be used at the same time, but it is recommended to use the "v2" version in your new projects.

Driver Configuration

This section covers how to configure the TWAI driver.

Operating Modes

The TWAI driver supports the following modes of operations:

Normal Mode: The normal operating mode allows the TWAI controller to take part in bus activities such as transmitting and receiving messages/error frames. Acknowledgement from another node is required when transmitting a message.

No Ack Mode: The No Acknowledgement mode is similar to normal mode, however acknowledgements are not required for a message transmission to be considered successful. This mode is useful when self testing the TWAI controller (loopback of transmissions).

Listen Only Mode: This mode prevents the TWAI controller from influencing the bus. Therefore, transmission of messages/acknowledgement/error frames will be disabled. However the TWAI controller is still able to receive messages but will not acknowledge the message. This mode is suited for bus monitor applications.

Alerts

The TWAI driver contains an alert feature that is used to notify the application layer of certain TWAI controller or TWAI bus events. Alerts are selectively enabled when the TWAI driver is installed, but can be reconfigured during runtime by calling `twai_reconfigure_alerts()`. The application can then wait for any enabled alerts to occur by calling `twai_read_alerts()`. The TWAI driver supports the following alerts:

TWAI Driver Alerts

Alert Flag	Description
`TWAI_ALERT_TX_IDLE`	No more messages queued for transmission
`TWAI_ALERT_TX_SUCCESS`	The previous transmission was successful
`TWAI_ALERT_RX_DATA`	A frame has been received and added to the RX queue
`TWAI_ALERT_BELOW_ERR_WARN`	Both error counters have dropped below error warning limit

Alert Flag	Description
`TWAI_ALERT_ERR_ACTIVE`	TWAI controller has become error active
`TWAI_ALERT_RECOVERY_IN_PROGRESS`	TWAI controller is undergoing bus recovery
`TWAI_ALERT_BUS_RECOVERED`	TWAI controller has successfully completed bus recovery
`TWAI_ALERT_ARB_LOST`	The previous transmission lost arbitration
`TWAI_ALERT_ABOVE_ERR_WARN`	One of the error counters have exceeded the error warning limit
`TWAI_ALERT_BUS_ERROR`	A (Bit, Stuff, CRC, Form, ACK) error has occurred on the bus
`TWAI_ALERT_TX_FAILED`	The previous transmission has failed
`TWAI_ALERT_RX_QUEUE_FULL`	The RX queue is full causing a received frame to be lost
`TWAI_ALERT_ERR_PASS`	TWAI controller has become error passive
`TWAI_ALERT_BUS_OFF`	Bus-off condition occurred. TWAI controller can no longer influence

Note

The TWAI controller's **error warning limit** is used to preemptively warn the application of bus errors before the error passive state is reached. By default, the TWAI driver sets the **error warning limit** to **96**. The `TWAI_ALERT_ABOVE_ERR_WARN` is raised when the TEC or REC becomes larger then or equal to the error warning limit. The `TWAI_ALERT_BELOW_ERR_WARN` is raised when both TEC and REC return back to values below **96**.

Note

When enabling alerts, the `TWAI_ALERT_AND_LOG` flag can be used to cause the TWAI driver to log any raised alerts to UART. However, alert logging is disabled and `TWAI_ALERT_AND_LOG` if the CONFIG_TWAI_ISR_IN_IRAM option is enabled (see Placing ISR into IRAM).

Note

The `TWAI_ALERT_ALL` and `TWAI_ALERT_NONE` macros can also be used to enable/disable all alerts during configuration/reconfiguration.

Bit Timing

The operating bit rate of the TWAI driver is configured using the `twai_timing_config_t` structure. The period of each bit is made up of multiple **time quanta**, and the period of a **time quantum** is determined by a pre-scaled version of the TWAI controller's source clock. A single bit contains

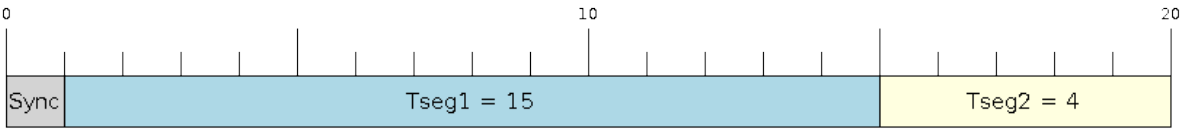
the following segments in the following order:

1. The **Synchronization Segment** consists of a single time quantum
2. **Timing Segment 1** consists of 1 to 16 time quanta before sample point
3. **Timing Segment 2** consists of 1 to 8 time quanta after sample point

The **Baudrate Prescaler** is used to determine the period of each time quantum by dividing the TWAI controller's source clock. On the ESP32, the `brp` can be **any even number from 2 to 128**. Alternatively, you can decide the resolution of each quantum, by setting `twai_timing_config_t::quanta_resolution_hz` to a non-zero value. In this way, the driver can calculate the underlying `brp` value for you. It is useful when you set different clock sources but want the bitrate to keep the same.

Supported clock source for a TWAI controller is listed in the `twai_clock_source_t` and can be specified in `twai_timing_config_t::clk_src`.

If the ESP32 is a revision 2 or later chip, the `brp` will **also support any multiple of 4 from 132 to 256**, and can be enabled by setting the CONFIG_ESP32_REV_MIN to revision 2 or higher.



Bit timing configuration for 500kbit/s given BRP = 8, clock source frequency is 80MHz

The sample point of a bit is located on the intersection of Timing Segment 1 and 2. Enabling **Triple Sampling** causes 3 time quanta to be sampled per bit instead of 1 (extra samples are located at the tail end of Timing Segment 1).

The **Synchronization Jump Width** is used to determine the maximum number of time quanta a single bit time can be lengthened/shortened for synchronization purposes. `sjw` can **range from 1 to 4**.

Note

Multiple combinations of `brp`, `tseg_1`, `tseg_2`, and `sjw` can achieve the same bit rate. Users should tune these values to the physical characteristics of their bus by taking into account factors such as **propagation delay, node information processing time, and phase errors**.

Bit timing **macro initializers** are also available for commonly used bit rates. The following macro initializers are provided by the TWAI driver.

- `TWAI_TIMING_CONFIG_1MBITS`
- `TWAI_TIMING_CONFIG_800KBITS`
- `TWAI_TIMING_CONFIG_500KBITS`
- `TWAI_TIMING_CONFIG_250KBITS`
- `TWAI_TIMING_CONFIG_125KBITS`
- `TWAI_TIMING_CONFIG_100KBITS`
- `TWAI_TIMING_CONFIG_50KBITS`
- `TWAI_TIMING_CONFIG_25KBITS`

Revision 2 or later of the ESP32 also supports the following bit rates:

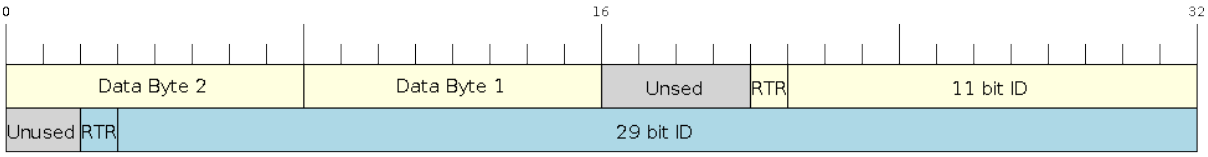
- `TWAI_TIMING_CONFIG_20KBITS`
- `TWAI_TIMING_CONFIG_16KBITS`
- `TWAI_TIMING_CONFIG_12_5KBITS`

Acceptance Filter

The TWAI controller contains a hardware acceptance filter which can be used to filter messages of a particular ID. A node that filters out a message **does not receive the message, but will still acknowledge it**. Acceptance filters can make a node more efficient by filtering out messages sent over the bus that are irrelevant to the node. The acceptance filter is configured using two 32-bit values within `twai_filter_config_t` known as the **acceptance code** and the **acceptance mask**.

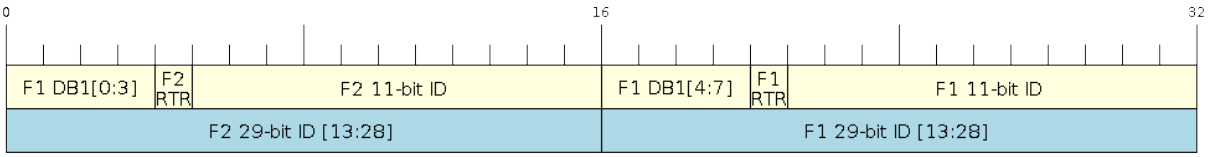
The **acceptance code** specifies the bit sequence which a message's ID, RTR, and data bytes must match in order for the message to be received by the TWAI controller. The **acceptance mask** is a bit sequence specifying which bits of the acceptance code can be ignored. This allows for a messages of different IDs to be accepted by a single acceptance code.

The acceptance filter can be used under **Single or Dual Filter Mode**. Single Filter Mode uses the acceptance code and mask to define a single filter. This allows for the first two data bytes of a standard frame to be filtered, or the entirety of an extended frame's 29-bit ID. The following diagram illustrates how the 32-bit acceptance code and mask are interpreted under Single Filter Mode (Note: The yellow and blue fields represent standard and extended frame formats respectively).



Bit layout of single filter mode (Right side MSBit)

Dual Filter Mode uses the acceptance code and mask to define two separate filters allowing for increased flexibility of ID's to accept, but does not allow for all 29-bits of an extended ID to be filtered. The following diagram illustrates how the 32-bit acceptance code and mask are interpreted under **Dual Filter Mode** (Note: The yellow and blue fields represent standard and extended frame formats respectively).



Bit layout of dual filter mode (Right side MSBit)

Disabling TX Queue

The TX queue can be disabled during configuration by setting the `tx_queue_len` member of `twai_general_config_t` to `0`. This allows applications that do not require message transmission to save a small amount of memory when using the TWAI driver.

Placing ISR into IRAM

The TWAI driver's ISR (Interrupt Service Routine) can be placed into IRAM so that the ISR can still run whilst the cache is disabled. Placing the ISR into IRAM may be necessary to maintain the TWAI driver's functionality during lengthy cache disabling operations (such as SPI Flash writes, OTA updates etc). Whilst the cache is disabled, the ISR continues to:

- Read received messages from the RX buffer and place them into the driver's RX queue.
- Load messages pending transmission from the driver's TX queue and write them into the TX buffer.

To place the TWAI driver's ISR, users must do the following:

- Enable the CONFIG_TWAI_ISR_IN_IRAM option using `idf.py menuconfig`.
- When calling `twai_driver_install()`, the `intr_flags` member of `twai_general_config_t` should

set the `ESP_INTR_FLAG_IRAM` set.

Note

When the CONFIG_TWAI_ISR_IN_IRAM option is enabled, the TWAI driver will no longer log any alerts (i.e., the `TWAI_ALERT_AND_LOG` flag will not have any effect).

ESP32 Errata Workarounds

The ESP32's TWAI controller contains multiple hardware errata (more details about the errata can be found in the ESP32's ECO document). Some of these errata are critical, and under specific circumstances, can place the TWAI controller into an unrecoverable state (i.e., the controller gets stuck until it is reset by the CPU).

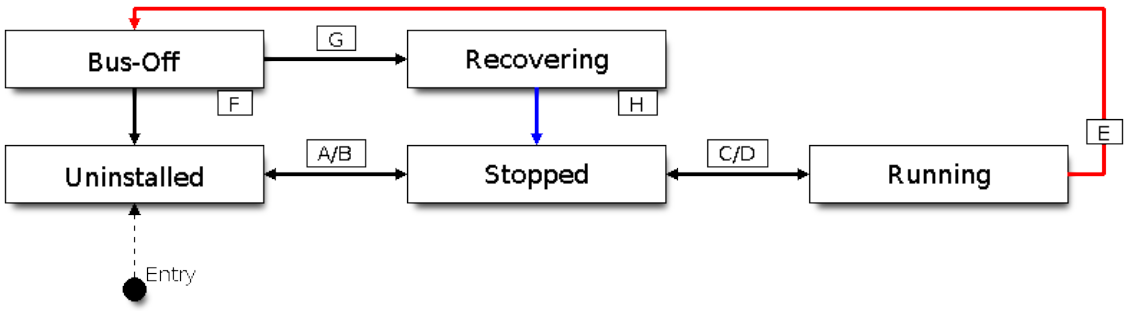
The TWAI driver contains software workarounds for these critical errata. With these workarounds, the ESP32 TWAI driver can operate normally, albeit with degraded performance. The degraded performance will affect users in the following ways depending on what particular errata conditions are encountered:

- The TWAI driver can occasionally drop some received messages.
- The TWAI driver can be unresponsive for a short period of time (i.e., will not transmit or ACK for 11 bit times or longer).
- If CONFIG_TWAI_ISR_IN_IRAM is enabled, the workarounds will increase IRAM usage by approximately 1 KB.

The software workarounds are enabled by default and it is recommended that users keep this workarounds enabled.

Driver Operation

The TWAI driver is designed with distinct states and strict rules regarding the functions or conditions that trigger a state transition. The following diagram illustrates the various states and their transitions.



State transition diagram of the TWAI driver (see table below)

Label	Transition	Action/Condition
A	Uninstalled > Stopped	`twai_driver_install()`
B	Stopped > Uninstalled	`twai_driver_uninstall()`
C	Stopped > Running	`twai_start()`
D	Running > Stopped	`twai_stop()`
E	Running > Bus-Off	Transmit Error Counter >= 256
F	Bus-Off > Uninstalled	`twai_driver_uninstall()`
G	Bus-Off > Recovering	`twai_initiate_recovery()`
H	Recovering > Stopped	128 occurrences of 11 consecutive recessive bits.

Driver States

Uninstalled: In the uninstalled state, no memory is allocated for the driver and the TWAI controller is powered OFF.

Stopped: In this state, the TWAI controller is powered ON and the TWAI driver has been installed. However the TWAI controller is unable to take part in any bus activities such as transmitting, receiving, or acknowledging messages.

Running: In the running state, the TWAI controller is able to take part in bus activities. Therefore messages can be transmitted/received/acknowledged. Furthermore, the TWAI controller is able to transmit error frames upon detection of errors on the bus.

Bus-Off: The bus-off state is automatically entered when the TWAI controller's Transmit Error Counter becomes greater than or equal to 256. The bus-off state indicates the occurrence of severe errors on the bus or in the TWAI controller. Whilst in the bus-off state, the TWAI controller is unable to take part in any bus activities. To exit the bus-off state, the TWAI controller must undergo the bus recovery process.

Recovering: The recovering state is entered when the TWAI controller undergoes bus recovery. The TWAI controller/TWAI driver remains in the recovering state until the 128 occurrences of 11 consecutive recessive bits is observed on the bus.

Message Fields and Flags

The TWAI driver distinguishes different types of messages by using the various bit field members of the `twai_message_t` structure. These bit field members determine whether a message is in standard or extended format, a remote frame, and the type of transmission to use when transmitting such a message.

These bit field members can also be toggled using the `flags` member of `twai_message_t` and the following message flags:

Message Flag	Description
`TWAI_MSG_FLAG_EXTD`	Message is in Extended Frame Format (29bit ID)
`TWAI_MSG_FLAG_RTR`	Message is a Remote Frame (Remote Transmission Request)
`TWAI_MSG_FLAG_SS`	Transmit message using Single Shot Transmission (Message will not be ret
`TWAI_MSG_FLAG_SELF`	Transmit message using Self Reception Request (Transmitted message wil
`TWAI_MSG_FLAG_DLC_NON_COMP`	Message's Data length code is larger than 8. This will break compliance wi
`TWAI_MSG_FLAG_NONE`	Clears all bit fields. Equivalent to a Standard Frame Format (11bit ID) Data

Examples

Configuration & Installation

The following code snippet demonstrates how to configure, install, and start the TWAI driver via the use of the various configuration structures, macro initializers, the `twai_driver_install()` function, and the `twai_start()` function.

```
#include "driver/gpio.h"
#include "driver/twai.h"

void app_main()
{
    //Initialize configuration structures using macro initializers
    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_21, GPIO_NUM_22,
TWAI_MODE_NORMAL);
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();

    //Install TWAI driver
    if (twai_driver_install(&g_config, &t_config, &f_config) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    //Start TWAI driver
    if (twai_start() == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    ...

}
```

The usage of macro initializers is not mandatory and each of the configuration structures can be manually.

Install Multiple TWAI Instances

The following code snippet demonstrates how to install multiple TWAI instances via the use of the `twai_driver_install_v2()` function.

```
#include "driver/gpio.h"
#include "driver/twai.h"

void app_main()
{
    twai_handle_t twai_bus_0;
    twai_handle_t twai_bus_1;
    //Initialize configuration structures using macro initializers
    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_0, GPIO_NUM_1,
TWAI_MODE_NORMAL);
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();

    //Install driver for TWAI bus 0
    g_config.controller_id = 0;
    if (twai_driver_install_v2(&g_config, &t_config, &f_config, &twai_bus_0) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }
    //Start TWAI driver
    if (twai_start_v2(twai_bus_0) == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    //Install driver for TWAI bus 1
    g_config.controller_id = 1;
    g_config.tx_io = GPIO_NUM_2;
    g_config.rx_io = GPIO_NUM_3;
    if (twai_driver_install_v2(&g_config, &t_config, &f_config, &twai_bus_1) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }
    //Start TWAI driver
    if (twai_start_v2(twai_bus_1) == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    //Other Driver operations must use version 2 API as well
    ...

}
```

Message Transmission

The following code snippet demonstrates how to transmit a message via the usage of the

`twai_message_t` type and `twai_transmit()` function.

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

...

//Configure message to transmit
twai_message_t message;
message.identifier = 0xAAAA;
message.extd = 1;
message.data_length_code = 4;
for (int i = 0; i < 4; i++) {
    message.data[i] = 0;
}

//Queue message for transmission
if (twai_transmit(&message, pdMS_TO_TICKS(1000)) == ESP_OK) {
    printf("Message queued for transmission\n");
} else {
    printf("Failed to queue message for transmission\n");
}
```

Message Reception

The following code snippet demonstrates how to receive a message via the usage of the `twai_message_t` type and `twai_receive()` function.

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

...

//Wait for message to be received
twai_message_t message;
if (twai_receive(&message, pdMS_TO_TICKS(10000)) == ESP_OK) {
    printf("Message received\n");
} else {
    printf("Failed to receive message\n");
    return;
}

//Process received message
if (message.extd) {
    printf("Message is in Extended Format\n");
} else {
    printf("Message is in Standard Format\n");
}
printf("ID is %d\n", message.identifier);
if (!(message.rtr)) {
    for (int i = 0; i < message.data_length_code; i++) {
        printf("Data byte %d = %d\n", i, message.data[i]);
    }
}
```

Reconfiguring and Reading Alerts

The following code snippet demonstrates how to reconfigure and read TWAI driver alerts via the use of the `twai_reconfigure_alerts()` and `twai_read_alerts()` functions.

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

...

//Reconfigure alerts to detect Error Passive and Bus-Off error states
uint32_t alerts_to_enable = TWAI_ALERT_ERR_PASS | TWAI_ALERT_BUS_OFF;
if (twai_reconfigure_alerts(alerts_to_enable, NULL) == ESP_OK) {
    printf("Alerts reconfigured\n");
} else {
    printf("Failed to reconfigure alerts");
}

//Block indefinitely until an alert occurs
uint32_t alerts_triggered;
twai_read_alerts(&alerts_triggered, portMAX_DELAY);
```

Stop and Uninstall

The following code demonstrates how to stop and uninstall the TWAI driver via the use of the `twai_stop()` and `twai_driver_uninstall()` functions.

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

...

//Stop the TWAI driver
if (twai_stop() == ESP_OK) {
    printf("Driver stopped\n");
} else {
    printf("Failed to stop driver\n");
    return;
}

//Uninstall the TWAI driver
if (twai_driver_uninstall() == ESP_OK) {
    printf("Driver uninstalled\n");
} else {
    printf("Failed to uninstall driver\n");
    return;
}
```

Multiple ID Filter Configuration

The acceptance mask in `twai_filter_config_t` can be configured such that two or more IDs are accepted for a single filter. For a particular filter to accept multiple IDs, the conflicting bit positions amongst the IDs must be set in the acceptance mask. The acceptance code can be set to any one of the IDs.

The following example shows how the calculate the acceptance mask given multiple IDs:

```
ID1 =  11'b101 1010 0000
ID2 =  11'b101 1010 0001
ID3 =  11'b101 1010 0100
ID4 =  11'b101 1010 1000
//Acceptance Mask
MASK = 11'b000 0000 1101
```

Application Examples

Network Example: The TWAI Network example demonstrates communication between two ESP32s using the TWAI driver API. One TWAI node acts as a network master that initiates and ceases the transfer of a data from another node acting as a network slave. The example can be found via peripherals/twai/twai_network.

Alert and Recovery Example: This example demonstrates how to use the TWAI driver's alert and bus-off recovery API. The example purposely introduces errors on the bus to put the TWAI controller into the Bus-Off state. An alert is used to detect the Bus-Off state and trigger the bus recovery process. The example can be found via peripherals/twai/twai_alert_and_recovery.

Self Test Example: This example uses the No Acknowledge Mode and Self Reception Request to cause the TWAI controller to send and simultaneously receive a series of messages. This example can be used to verify if the connections between the TWAI controller and the external transceiver are working correctly. The example can be found via peripherals/twai/twai_self_test.

API Reference

Header File

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

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

Structures

struct **twai_message_t**

Structure to store a TWAI message.

Note

The flags member is deprecated

Public Members

uint32_t extd

Extended Frame Format (29bit ID)

uint32_t rtr

Message is a Remote Frame

uint32_t ss

Transmit as a Single Shot Transmission. Unused for received.

uint32_t self

Transmit as a Self Reception Request. Unused for received.

uint32_t dlc_non_comp

Message's Data length code is larger than 8. This will break compliance with ISO 11898-1

uint32_t reserved

Reserved bits

uint32_t flags

Deprecated: Alternate way to set bits using message flags

uint32_t identifier

11 or 29 bit identifier

uint8_t data_length_code

Data length code

uint8_t data[TWAI_FRAME_MAX_DLC]

Data bytes (not relevant in RTR frame)

---

struct **twai_timing_config_t**

Structure for bit timing configuration of the TWAI driver.

Note

Macro initializers are available for this structure

Public Members

twai_clock_source_t **clk_src**

Clock source, set to 0 or TWAI_CLK_SRC_DEFAULT if you want a default clock source

uint32_t quanta_resolution_hz

The resolution of one timing quanta, in Hz. Note: the value of `brp` will reflected by this field if it's non-zero, otherwise, `brp` needs to be set manually

uint32_t brp

Baudrate prescale (i.e., clock divider). Any even number from 2 to 128 for ESP32, 2 to 32768 for non-ESP32 chip. Note: For ESP32 ECO 2 or later, multiples of 4 from 132 to 256 are also supported

uint8_t tseg_1

Timing segment 1 (Number of time quanta, between 1 to 16)

uint8_t tseg_2

Timing segment 2 (Number of time quanta, 1 to 8)

uint8_t sjw

Synchronization Jump Width (Max time quanta jump for synchronize from 1 to 4)

bool triple_sampling

Enables triple sampling when the TWAI controller samples a bit

struct **twai_filter_config_t**

Structure for acceptance filter configuration of the TWAI driver (see documentation)

Note

Macro initializers are available for this structure

Public Members

uint32_t acceptance_code

32-bit acceptance code

uint32_t acceptance_mask

32-bit acceptance mask

bool single_filter

Use Single Filter Mode (see documentation)

Macros

TWAI_EXTD_ID_MASK

TWAI Constants.

Bit mask for 29 bit Extended Frame Format ID

TWAI_STD_ID_MASK

Bit mask for 11 bit Standard Frame Format ID

TWAI_FRAME_MAX_DLC

Max data bytes allowed in TWAI

TWAI_FRAME_EXTD_ID_LEN_BYTES

EFF ID requires 4 bytes (29bit)

TWAI_FRAME_STD_ID_LEN_BYTES

SFF ID requires 2 bytes (11bit)

TWAI_ERR_PASS_THRESH

Error counter threshold for error passive

Type Definitions

typedef soc_periph_twai_clk_src_t **twai_clock_source_t**

RMT group clock source.

Note

User should select the clock source based on the power and resolution requirement

Enumerations

enum **twai_mode_t**

TWAI Controller operating modes.

Values:

***enumerator* TWAI_MODE_NORMAL**

Normal operating mode where TWAI controller can send/receive/acknowledge messages

***enumerator* TWAI_MODE_NO_ACK**

Transmission does not require acknowledgment. Use this mode for self testing

***enumerator* TWAI_MODE_LISTEN_ONLY**

The TWAI controller will not influence the bus (No transmissions or acknowledgments) but can receive messages

Header File

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

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

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

```
REQUIRES driver
```

or

```
PRIV_REQUIRES driver
```

Functions

esp_err_t **twai_driver_install**(*const* twai_general_config_t *g_config, *const* twai_timing_config_t *t_config, *const* twai_filter_config_t *f_config)

Install TWAI driver.

This function installs the TWAI driver using three configuration structures. The required memory is allocated and the TWAI driver is placed in the stopped state after running this function.

Note

Macro initializers are available for the configuration structures (see documentation)

Note

To reinstall the TWAI driver, call twai_driver_uninstall() first

Parameters:
- **g_config** -- **[in]** General configuration structure
- **t_config** -- **[in]** Timing configuration structure
- **f_config** -- **[in]** Filter configuration structure

Returns:
- ESP_OK: Successfully installed TWAI driver
- ESP_ERR_INVALID_ARG: Arguments are invalid, e.g. invalid clock source, invalid quanta resolution
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_INVALID_STATE: Driver is already installed

esp_err_t **twai_driver_install_v2**(*const* twai_general_config_t *g_config, *const* twai_timing_config_t *t_config, *const* twai_filter_config_t *f_config, twai_handle_t *ret_twai)

Install TWAI driver and return a handle.

Note

This is an advanced version of `twai_driver_install` that can return a driver handle, so that it allows you to install multiple TWAI drivers. Don't forget to set the proper controller_id in the `twai_general_config_t` Please refer to the documentation of `twai_driver_install` for more details.

Parameters:
- **g_config** -- **[in]** General configuration structure
- **t_config** -- **[in]** Timing configuration structure
- **f_config** -- **[in]** Filter configuration structure
- **ret_twai** -- **[out]** Pointer to a new created TWAI handle

Returns:
- ESP_OK: Successfully installed TWAI driver
- ESP_ERR_INVALID_ARG: Arguments are invalid, e.g. invalid clock source, invalid quanta resolution, invalid controller ID
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_INVALID_STATE: Driver is already installed

---

esp_err_t **twai_driver_uninstall**(void)

Uninstall the TWAI driver.

This function uninstalls the TWAI driver, freeing the memory utilized by the driver. This function can only be called when the driver is in the stopped state or the bus-off state.

Warning

The application must ensure that no tasks are blocked on TX/RX queues or alerts when this function is called.

Returns:
- ESP_OK: Successfully uninstalled TWAI driver
- ESP_ERR_INVALID_STATE: Driver is not in stopped/bus-off state, or is not installed

---

esp_err_t **twai_driver_uninstall_v2**(twai_handle_t handle)

Uninstall the TWAI driver with a given handle.

Note

This is an advanced version of `twai_driver_uninstall` that can uninstall a TWAI driver with a given handle. Please refer to the documentation of `twai_driver_uninstall` for more

details.

Parameters: **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`

Returns:

- ESP_OK: Successfully uninstalled TWAI driver
- ESP_ERR_INVALID_STATE: Driver is not in stopped/bus-off state, or is not installed

---

esp_err_t **twai_start**(void)

Start the TWAI driver.

This function starts the TWAI driver, putting the TWAI driver into the running state. This allows the TWAI driver to participate in TWAI bus activities such as transmitting/receiving messages. The TX and RX queue are reset in this function, clearing any messages that are unread or pending transmission. This function can only be called when the TWAI driver is in the stopped state.

Returns:

- ESP_OK: TWAI driver is now running
- ESP_ERR_INVALID_STATE: Driver is not in stopped state, or is not installed

---

esp_err_t **twai_start_v2**(twai_handle_t handle)

Start the TWAI driver with a given handle.

Note

This is an advanced version of `twai_start` that can start a TWAI driver with a given handle. Please refer to the documentation of `twai_start` for more details.

Parameters: **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`

Returns:

- ESP_OK: TWAI driver is now running
- ESP_ERR_INVALID_STATE: Driver is not in stopped state, or is not installed

---

esp_err_t **twai_stop**(void)

Stop the TWAI driver.

This function stops the TWAI driver, preventing any further message from being transmitted or received until twai_start() is called. Any messages in the TX queue are cleared. Any

messages in the RX queue should be read by the application after this function is called. This function can only be called when the TWAI driver is in the running state.

Warning

A message currently being transmitted/received on the TWAI bus will be ceased immediately. This may lead to other TWAI nodes interpreting the unfinished message as an error.

Returns:

- ESP_OK: TWAI driver is now Stopped
- ESP_ERR_INVALID_STATE: Driver is not in running state, or is not installed

---

esp_err_t **twai_stop_v2**(twai_handle_t handle)

Stop the TWAI driver with a given handle.

Note

This is an advanced version of `twai_stop` that can stop a TWAI driver with a given handle. Please refer to the documentation of `twai_stop` for more details.

Parameters: **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`

Returns:

- ESP_OK: TWAI driver is now Stopped
- ESP_ERR_INVALID_STATE: Driver is not in running state, or is not installed

---

esp_err_t **twai_transmit**(*const* twai_message_t *message, TickType_t ticks_to_wait)

Transmit a TWAI message.

This function queues a TWAI message for transmission. Transmission will start immediately if no other messages are queued for transmission. If the TX queue is full, this function will block until more space becomes available or until it times out. If the TX queue is disabled (TX queue length = 0 in configuration), this function will return immediately if another message is undergoing transmission. This function can only be called when the TWAI driver is in the running state and cannot be called under Listen Only Mode.

Note

This function does not guarantee that the transmission is successful. The TX_SUCCESS/TX_FAILED alert can be enabled to alert the application upon the success/failure of a transmission.

Note

The TX_IDLE alert can be used to alert the application when no other messages are awaiting transmission.

Parameters:
- **message** -- **[in]** Message to transmit
- **ticks_to_wait** -- **[in]** Number of FreeRTOS ticks to block on the TX queue

Returns:
- ESP_OK: Transmission successfully queued/initiated
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_TIMEOUT: Timed out waiting for space on TX queue
- ESP_FAIL: TX queue is disabled and another message is currently transmitting
- ESP_ERR_INVALID_STATE: TWAI driver is not in running state, or is not installed
- ESP_ERR_NOT_SUPPORTED: Listen Only Mode does not support transmissions

---

esp_err_t **twai_transmit_v2**(twai_handle_t handle, *const* twai_message_t *message, TickType_t ticks_to_wait)

Transmit a TWAI message via a given handle.

Note

This is an advanced version of `twai_transmit` that can transmit a TWAI message with a given handle. Please refer to the documentation of `twai_transmit` for more details.

Parameters:
- **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`
- **message** -- **[in]** Message to transmit
- **ticks_to_wait** -- **[in]** Number of FreeRTOS ticks to block on the TX queue

Returns:

- ESP_OK: Transmission successfully queued/initiated
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_TIMEOUT: Timed out waiting for space on TX queue
- ESP_FAIL: TX queue is disabled and another message is currently transmitting
- ESP_ERR_INVALID_STATE: TWAI driver is not in running state, or is not installed
- ESP_ERR_NOT_SUPPORTED: Listen Only Mode does not support transmissions

---

esp_err_t **twai_receive**(twai_message_t *message, TickType_t ticks_to_wait)

Receive a TWAI message.

This function receives a message from the RX queue. The flags field of the message structure will indicate the type of message received. This function will block if there are no messages in the RX queue

Warning

The flags field of the received message should be checked to determine if the received message contains any data bytes.

Parameters:

- **message** -- **[out]** Received message
- **ticks_to_wait** -- **[in]** Number of FreeRTOS ticks to block on RX queue

Returns:

- ESP_OK: Message successfully received from RX queue
- ESP_ERR_TIMEOUT: Timed out waiting for message
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_receive_v2**(twai_handle_t handle, twai_message_t *message, TickType_t ticks_to_wait)

Receive a TWAI message via a given handle.

Note

This is an advanced version of `twai_receive` that can receive a TWAI message with a given handle. Please refer to the documentation of `twai_receive` for more details.

Parameters:
- **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`
- **message** -- **[out]** Received message
- **ticks_to_wait** -- **[in]** Number of FreeRTOS ticks to block on RX queue

Returns:
- ESP_OK: Message successfully received from RX queue
- ESP_ERR_TIMEOUT: Timed out waiting for message
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_read_alerts**(uint32_t *alerts, TickType_t ticks_to_wait)

Read TWAI driver alerts.

This function will read the alerts raised by the TWAI driver. If no alert has been issued when this function is called, this function will block until an alert occurs or until it timeouts.

Note

Multiple alerts can be raised simultaneously. The application should check for all alerts that have been enabled.

Parameters:
- **alerts** -- **[out]** Bit field of raised alerts (see documentation for alert flags)
- **ticks_to_wait** -- **[in]** Number of FreeRTOS ticks to block for alert

Returns:
- ESP_OK: Alerts read
- ESP_ERR_TIMEOUT: Timed out waiting for alerts
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_read_alerts_v2**(twai_handle_t handle, uint32_t *alerts, TickType_t ticks_to_wait)

Read TWAI driver alerts with a given handle.

Note

This is an advanced version of `twai_read_alerts` that can read TWAI driver alerts with a given handle. Please refer to the documentation of `twai_read_alerts` for more details.

Parameters:
- **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`
- **alerts** -- **[out]** Bit field of raised alerts (see documentation for alert flags)
- **ticks_to_wait** -- **[in]** Number of FreeRTOS ticks to block for alert

Returns:
- ESP_OK: Alerts read
- ESP_ERR_TIMEOUT: Timed out waiting for alerts
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_reconfigure_alerts**(uint32_t alerts_enabled, uint32_t *current_alerts)

Reconfigure which alerts are enabled.

This function reconfigures which alerts are enabled. If there are alerts which have not been read whilst reconfiguring, this function can read those alerts.

Parameters:
- **alerts_enabled** -- **[in]** Bit field of alerts to enable (see documentation for alert flags)
- **current_alerts** -- **[out]** Bit field of currently raised alerts. Set to NULL if unused

Returns:
- ESP_OK: Alerts reconfigured
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_reconfigure_alerts_v2**(twai_handle_t handle, uint32_t alerts_enabled, uint32_t *current_alerts)

Reconfigure which alerts are enabled, with a given handle.

Note

This is an advanced version of `twai_reconfigure_alerts` that can reconfigure which alerts are enabled with a given handle. Please refer to the documentation of `twai_reconfigure_alerts` for more details.

Parameters:
- **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`
- **alerts_enabled** -- **[in]** Bit field of alerts to enable (see documentation for alert flags)
- **current_alerts** -- **[out]** Bit field of currently raised alerts. Set to NULL if unused

Returns:

- ESP_OK: Alerts reconfigured
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_initiate_recovery**(void)

Start the bus recovery process.

This function initiates the bus recovery process when the TWAI driver is in the bus-off state. Once initiated, the TWAI driver will enter the recovering state and wait for 128 occurrences of the bus-free signal on the TWAI bus before returning to the stopped state. This function will reset the TX queue, clearing any messages pending transmission.

Note

The BUS_RECOVERED alert can be enabled to alert the application when the bus recovery process completes.

Returns:

- ESP_OK: Bus recovery started
- ESP_ERR_INVALID_STATE: TWAI driver is not in the bus-off state, or is not installed

---

esp_err_t **twai_initiate_recovery_v2**(twai_handle_t handle)

Start the bus recovery process with a given handle.

Note

This is an advanced version of `twai_initiate_recovery` that can start the bus recovery process with a given handle. Please refer to the documentation of `twai_initiate_recovery` for more details.

Parameters: **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`

Returns:

- ESP_OK: Bus recovery started
- ESP_ERR_INVALID_STATE: TWAI driver is not in the bus-off state, or is not installed

---

esp_err_t **twai_get_status_info**(twai_status_info_t *status_info)

Get current status information of the TWAI driver.

Parameters: **status_info** -- **[out]** Status information

Returns:

- ESP_OK: Status information retrieved
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_get_status_info_v2**(twai_handle_t handle, twai_status_info_t *status_info)

Get current status information of a given TWAI driver handle.

Note

This is an advanced version of `twai_get_status_info` that can get current status information of a given TWAI driver handle. Please refer to the documentation of `twai_get_status_info` for more details.

Parameters:

- **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`
- **status_info** -- **[out]** Status information

Returns:

- ESP_OK: Status information retrieved
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_clear_transmit_queue**(void)

Clear the transmit queue.

This function will clear the transmit queue of all messages.

Note

The transmit queue is automatically cleared when twai_stop() or twai_initiate_recovery() is called.

Returns:

- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed or TX queue is disabled

---

esp_err_t **twai_clear_transmit_queue_v2**(twai_handle_t handle)

Clear the transmit queue of a given TWAI driver handle.

Note

This is an advanced version of `twai_clear_transmit_queue` that can clear the transmit queue of a given TWAI driver handle. Please refer to the documentation of `twai_clear_transmit_queue` for more details.

Parameters: **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`

Returns:

- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed or TX queue is disabled

---

esp_err_t **twai_clear_receive_queue**(void)

Clear the receive queue.

This function will clear the receive queue of all messages.

Note

The receive queue is automatically cleared when twai_start() is called.

Returns:

- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

---

esp_err_t **twai_clear_receive_queue_v2**(twai_handle_t handle)

Clear the receive queue of a given TWAI driver handle.

Note

This is an advanced version of `twai_clear_receive_queue` that can clear the receive queue of a given TWAI driver handle. Please refer to the documentation of `twai_clear_receive_queue` for more details.

Parameters: **handle** -- **[in]** TWAI driver handle returned by `twai_driver_install_v2`

Returns:

- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

Structures

struct **twai_general_config_t**

Structure for general configuration of the TWAI driver.

Note

Macro initializers are available for this structure

Public Members

int controller_id

TWAI controller ID, index from 0. If you want to install TWAI driver with a non-zero controller_id, please use `twai_driver_install_v2`

twai_mode_t **mode**

Mode of TWAI controller

gpio_num_t tx_io

Transmit GPIO number

gpio_num_t rx_io

Receive GPIO number

gpio_num_t clkout_io

CLKOUT GPIO number (optional, set to -1 if unused)

gpio_num_t bus_off_io

Bus off indicator GPIO number (optional, set to -1 if unused)

uint32_t tx_queue_len

Number of messages TX queue can hold (set to 0 to disable TX Queue)

uint32_t rx_queue_len

Number of messages RX queue can hold

uint32_t alerts_enabled

Bit field of alerts to enable (see documentation)

uint32_t clkout_divider

CLKOUT divider. Can be 1 or any even number from 2 to 14 (optional, set to 0 if unused)

int intr_flags

Interrupt flags to set the priority of the driver's ISR. Note that to use the ESP_INTR_FLAG_IRAM, the CONFIG_TWAI_ISR_IN_IRAM option should be enabled first.

struct twai_status_info_t

Structure to store status information of TWAI driver.

Public Members

twai_state_t state

Current state of TWAI controller (Stopped/Running/Bus-Off/Recovery)

uint32_t msgs_to_tx

Number of messages queued for transmission or awaiting transmission completion

uint32_t msgs_to_rx

Number of messages in RX queue waiting to be read

uint32_t tx_error_counter

Current value of Transmit Error Counter

uint32_t rx_error_counter

Current value of Receive Error Counter

uint32_t tx_failed_count

Number of messages that failed transmissions

uint32_t rx_missed_count

Number of messages that were lost due to a full RX queue (or errata workaround if enabled)

uint32_t rx_overrun_count

Number of messages that were lost due to a RX FIFO overrun

uint32_t arb_lost_count

Number of instances arbitration was lost

uint32_t bus_error_count

Number of instances a bus error has occurred

Macros

TWAI_IO_UNUSED

Marks GPIO as unused in TWAI configuration

Type Definitions

typedef struct twai_obj_t ***twai_handle_t**

TWAI controller handle.

Enumerations

enum **twai_state_t**

TWAI driver states.

Values:

***enumerator* TWAI_STATE_STOPPED**

Stopped state. The TWAI controller will not participate in any TWAI bus activities

***enumerator* TWAI_STATE_RUNNING**

Running state. The TWAI controller can transmit and receive messages

***enumerator* TWAI_STATE_BUS_OFF**

Bus-off state. The TWAI controller cannot participate in bus activities until it has recovered

***enumerator* TWAI_STATE_RECOVERING**

Recovering state. The TWAI controller is undergoing bus recovery