



# XCORE Peripheral IO Framework - Programming Guide

Release: 3.1.0

Publication Date: 2023/10/04

# Table of Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>API Reference</b>	<b>2</b>
2.1	UART Library	2
2.1.1	UART Tx	2
	UART Tx Usage	2
	UART Tx Usage ISR/Buffered	3
	UART Tx API	3
2.1.2	UART Rx	6
	UART Rx Usage	6
	UART Rx Usage ISR/Buffered	7
	UART Rx API	8
2.2	I <sup>2</sup> C Library	9
2.2.1	I <sup>2</sup> C Master	10
	I <sup>2</sup> C Master Usage	10
	I <sup>2</sup> C Master API	10
2.2.2	I <sup>2</sup> C Slave	12
	I <sup>2</sup> C Slave Usage	12
	I <sup>2</sup> C Slave API	13
2.2.3	I <sup>2</sup> C Registers	15
	I <sup>2</sup> C Register API	15
2.3	I <sup>2</sup> S Library	18
2.3.1	I <sup>2</sup> S Common API	19
	I <sup>2</sup> S Instances	19
	TDM Instances	22
2.3.2	I <sup>2</sup> S Master	22
	I <sup>2</sup> S Master Usage	22
	I <sup>2</sup> S Master API	23
2.3.3	I <sup>2</sup> S Slave	24
	I <sup>2</sup> S Slave Usage	24
	I <sup>2</sup> S Slave API	25
2.3.4	TDM Slave	25
	TDM Slave Tx Usage	25
	TDM Slave Tx API	26
2.4	SPI Library	27
2.4.1	SPI Master	27
	SPI Master Usage	27
	SPI Master API	28
2.4.2	SPI Slave	32
	SPI Slave Usage	32
	SPI Slave API	32
<b>3</b>	<b>Copyright &amp; Disclaimer</b>	<b>34</b>
<b>4</b>	<b>Licenses</b>	<b>35</b>
4.1	XMOS	35
	<b>Index</b>	<b>36</b>

# 1 Overview

---

The peripheral IO framework is a collection of IO libraries written in C for XCORE.AI. It includes software defined peripherals for:

- UART - transmit and receive
- I<sup>2</sup>C - master and slave
- I<sup>2</sup>S - master and slave and TDM slave Tx
- SPI - master and slave

## 2 API Reference

---

### 2.1 UART Library

This library provide a software defined UART (universal asynchronous receiver transmitter) allowing you to communicate with other UART enabled devices in your system. A UART is a single wire per direction communications interface allowing either half or full duplex communication. The components in this library are controlled via C and behave as a UART transmitter and/or receiver peripheral.

Various configuration options are available including baud rate (individually settable per direction), number of data bits (between 5 and 8), parity (EVEN, ODD or NONE) and number of stop bits (1 or 2). The UART does not support flow control signals. Only a single 1b IO port per UART direction is needed.

The Tx UART supports up to 1152000 baud unbuffered and 576000 baud buffered with a 75MHz logical core. The Rx UART supports up to 700000 baud unbuffered and 422400 baud buffered with a 75MHz logical core. Proportionally higher rates are achievable using a higher logical core MHz.

The UART receive supports standard error detection including START, PARITY and FRAMING errors. A callback mechanism is included to notify the user of these conditions.

The UART may be used in blocking mode, where the call to Tx/Rx does not return until the stop bit is complete. It may also be used in ISR/buffered mode where the UART Rx and/or Tx operates in background mode using a FIFO and callbacks to manage data-flow and error conditions. Cycles are stolen from the logical core which setup the interrupt. In ISR/buffered mode additional callbacks are supported indicating the UNDERRUN condition when the Tx buffer is empty and OVERRUN when the Rx buffer is full.

Table 2.1: UART data wires

Tx	Transmit line controlled by UART Tx
Rx	Receive line controlled by UART Rx

All UART functions can be accessed via the `uart.h` header:

```
#include "uart.h"
```

#### 2.1.1 UART Tx

##### UART Tx Usage

The following code snippet demonstrates the basic blocking usage of an UART Tx device.

```
#include <xs1.h>
#include "uart.h"

uart_tx_t uart;

port_t p_uart_tx = XS1_PORT_1A;
hwtimer_t tmr = hwtimer_alloc();

uint8_t tx_data[4] = {0x01, 0x02, 0x04, 0x08};

// Initialize the UART Tx
uart_tx_blocking_init(&uart, p_uart_tx, 115200, 8, UART_PARITY_NONE, 1, tmr);
```

(continues on next page)



(continued from previous page)

```
// Transfer some data
for(int i = 0; i < sizeof(tx_data); i++){
    uart_tx(&uart, tx_data[i]);
}
```

## UART Tx Usage ISR/Buffered

The following code snippet demonstrates the usage of an UART Tx device used in ISR/Buffered mode:

```
#include <xs1.h>
#include "uart.h"

HIL_UART_TX_CALLBACK_ATTR void tx_empty_callback(void *app_data){
    int *tx_empty = (int *)app_data;
    *tx_empty = 1;
}

void uart_tx(void){

    uart_tx_t uart;
    port_t p_uart_tx = XS1_PORT_1A;
    hwtimer_t tmr = hwtimer_alloc();
    uint8_t buffer[64 + 1] = {0}; // Note buffer size plus one

    uint8_t tx_data[4] = {0x01, 0x02, 0x04, 0x08};
    volatile int tx_empty = 0;

    // Initialize the UART Tx
    uart_tx_init(&uart, p_uart_tx, 115200, 8, UART_PARITY_NONE, 1, tmr, buffer,
↳sizeof(buffer), tx_empty_callback, &tx_empty);

    // Transfer some data
    for(int i = 0; i < sizeof(tx_data); i++){
        uart_tx(&uart, tx_data[i]);
    }

    // Wait for it to complete
    while(!tx_empty);
}
```

## UART Tx API

The following structures and functions are used to initialize and start an UART Tx instance.

enum uart\_parity

Enum type representing the different options parity types.

Values:

enumerator UART\_PARITY\_NONE

enumerator UART\_PARITY\_EVEN

enumerator `UART_PARITY_ODD`

enum `uart_callback_code_t`

Enum type representing the callback error codes.

*Values:*

enumerator `UART_RX_COMPLETE`

enumerator `UART_UNDERRUN_ERROR`

enumerator `UART_START_BIT_ERROR`

enumerator `UART_PARITY_ERROR`

enumerator `UART_FRAMING_ERROR`

enumerator `UART_OVERRUN_ERROR`

enum `uart_state_t`

Enum type representing the different states for the UART logic.

*Values:*

enumerator `UART_IDLE`

enumerator `UART_START`

enumerator `UART_DATA`

enumerator `UART_PARITY`

enumerator `UART_STOP`

typedef enum `uart_parity` `uart_parity_t`

Enum type representing the different options parity types.

```
void uart_tx_init(uart_tx_t *uart, port_t tx_port, uint32_t baud_rate, uint8_t data_bits, uart_parity_t parity,
                uint8_t stop_bits, hwtimer_t tmr, uint8_t *tx_buff, size_t buffer_size_plus_one, void
                (*uart_tx_empty_callback_fptr)(void *app_data), void *app_data)
```

Initializes a UART Tx I/O interface. Passing a valid buffer will enable buffered mode with ISR for use in bare-metal applications.

#### Parameters

- `uart` – The `uart_tx_t` context to initialise.
- `tx_port` – The port used transmit the UART frames.
- `baud_rate` – The baud rate of the UART in bits per second.
- `data_bits` – The number of data bits per frame sent.
- `parity` – The type of parity used. See `uart_parity_t` above.

- `stop_bits` – The number of stop bits asserted at the of the frame.
- `tmr` – The resource id of the timer to be used. Polling mode will be used if set to 0.
- `tx_buff` – Pointer to a buffer. Optional. If set to zero the UART will run in blocking mode. If initialised to a valid buffer, the UART will be interrupt driven.
- `buffer_size_plus_one` – Size of the buffer if enabled in `tx_buff`. Note that the buffer allocation and size argument must be one greater than needed. Eg. `buff[65]` for a 64 byte buffer.
- `uart_tx_empty_callback_fptr` – Callback function pointer for UART buffer empty in buffered mode.
- `app_data` – A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

```
void uart_tx_blocking_init(uart_tx_t *uart, port_t tx_port, uint32_t baud_rate, uint8_t data_bits,
                          uart_parity_t parity, uint8_t stop_bits, hwtimer_t tmr)
```

Initializes a UART Tx I/O interface. The API is hard wired to blocking mode where the call to `uart_tx` will return at the end of sending the stop bit.

#### Parameters

- `uart` – The *uart\_tx\_t* context to initialise.
- `tx_port` – The port used transmit the UART frames.
- `baud_rate` – The baud rate of the UART in bits per second.
- `data_bits` – The number of data bits per frame sent.
- `parity` – The type of parity used. See *uart\_parity\_t* above.
- `stop_bits` – The number of stop bits asserted at the of the frame.
- `tmr` – The resource id of the timer to be used. Polling mode will be used if set to 0.

```
void uart_tx(uart_tx_t *uart, uint8_t data)
```

Transmits a single UART frame with parameters as specified in *uart\_tx\_init()*

#### Parameters

- `uart` – The *uart\_tx\_t* context to initialise.
- `data` – The word to transmit.

```
void uart_tx_deinit(uart_tx_t *uart)
```

De-initializes the specified UART Tx interface. This disables the port also. The timer, if used, needs to be freed by the application.

#### Parameters

- `uart` – The *uart\_tx\_t* context to de-initialise.

```
UART_START_BIT_ERROR_VAL
```

Define which sets the enum start point of RX errors. This is relied upon by the RTOS drivers and allows optimisation of error handling.

```
HIL_UART_TX_CALLBACK_ATTR
```

This attribute must be specified on the UART TX UNDERRUN callback function provided by the application. It ensures the correct stack usage is calculated.

```
HIL_UART_RX_CALLBACK_ATTR
```

This attribute must be specified on the UART Rx callback functions (both ERROR and Rx complete callbacks) provided by the application. It ensures the correct stack usage is correctly calculated.

```
struct uart_tx_t
```

`#include <uart.h>` Struct to hold a UART Tx context.

The members in this struct should not be accessed directly. Use the API provided instead.

## 2.1.2 UART Rx

### UART Rx Usage

The following code snippet demonstrates the basic usage of an UART Rx device where the function call to Rx returns after the stop bit has been sampled. The function blocks until a complete byte has been received.

```
#include <xs1.h>
#include <print.h>
#include "uart.h"
```

```
HIL_UART_RX_CALLBACK_ATTR void rx_error_callback(uart_callback_code_t callback_code,
↳void *app_data){
    switch(callback_code){
        case UART_START_BIT_ERROR:
            printstrln("UART_START_BIT_ERROR");
            break;
        case UART_PARITY_ERROR:
            printstrln("UART_PARITY_ERROR");
            break;
        case UART_FRAMING_ERROR:
            printstrln("UART_FRAMING_ERROR");
            test_abort = 1;
            break;
        case UART_OVERRUN_ERROR:
            printstrln("UART_OVERRUN_ERROR");
            break;
        case UART_UNDERRUN_ERROR:
            printstrln("UART_UNDERRUN_ERROR");
            break;
        default:
            printstr("Unexpected callback code: ");
            printintln(callback_code);
    }
}
```

```
void uart_rx(void){

    uart_rx_t uart;

    port_t p_uart_rx = XS1_PORT_1B;
    hwtimer_t tmr = hwtimer_alloc();

    char test_rx[16];

    // Initialize the UART Rx
    uart_rx_blocking_init( &uart, p_uart_rx, 115200, 8, UART_PARITY_NONE, 1, tmr,
                          rx_error_callback, &uart);

    // Receive some data
```

(continues on next page)



(continued from previous page)

```

for(int i = 0; i < sizeof(rx_data); i++){
    test_rx[i] = uart_rx(&uart);
}

```

## UART Rx Usage ISR/Buffered

The following code snippet demonstrates the usage of an UART Rx device used in ISR/Buffered mode:

```

#include <xs1.h>
#include <print.h>
#include "uart.h"

```

```

HIL_UART_RX_CALLBACK_ATTR void rx_error_callback(uart_callback_code_t callback_code,
void *app_data){
    switch(callback_code){
        case UART_START_BIT_ERROR:
            printstrln("UART_START_BIT_ERROR");
            break;
        case UART_PARITY_ERROR:
            printstrln("UART_PARITY_ERROR");
            break;
        case UART_FRAMING_ERROR:
            printstrln("UART_FRAMING_ERROR");
            test_abort = 1;
            break;
        case UART_OVERRUN_ERROR:
            printstrln("UART_OVERRUN_ERROR");
            break;
        case UART_UNDERRUN_ERROR:
            printstrln("UART_UNDERRUN_ERROR");
            break;
        default:
            printstr("Unexpected callback code: ");
            printintln(callback_code);
    }
}

```

```

HIL_UART_RX_CALLBACK_ATTR void rx_callback(void *app_data){
    unsigned *bytes_received = (unsigned *)app_data;
    *bytes_received += 1;
}

```

```

void uart_rx(void){

    uart_rx_t uart;
    port_t p_uart_rx = XS1_PORT_1A;
    hwtimer_t tmr = hwtimer_alloc();
    uint8_t buffer[64 + 1] = {0}; // Note buffer size plus one

    volatile unsigned bytes_received = 0;

    // Initialize the UART Rx
    uart_rx_init(&uart, p_uart_rx, 115200, 8, UART_PARITY_NONE, 1, tmr,

```

(continues on next page)



(continued from previous page)

```

        buffer, sizeof(buffer), rx_callback, &bytes_received);

// Wait for 16b of data
while(bytes_received < 15);

// Get the data
uint8_t test_rx[NUM_RX_WORDS];
for(int i = 0; i < 16; i++){
    test_rx[i] = uart_rx(&uart);
}

```

## UART Rx API

The following structures and functions are used to initialize and start an UART Rx instance.

```

void uart_rx_init(uart_rx_t *uart, port_t rx_port, uint32_t baud_rate, uint8_t data_bits, uart_parity_t parity,
                uint8_t stop_bits, hwtimer_t tmr, uint8_t *rx_buff, size_t buffer_size_plus_one, void
                (*uart_rx_complete_callback_fptr)(void *app_data), void
                (*uart_rx_error_callback_fptr)(uart_callback_code_t callback_code, void *app_data), void
                *app_data)

```

Initializes a UART Rx I/O interface. Passing a valid buffer will enable buffered mode with ISR for use in bare-metal applications.

### Parameters

- `uart` – The *uart\_rx\_t* context to initialise.
- `rx_port` – The port used receive the UART frames.
- `baud_rate` – The baud rate of the UART in bits per second.
- `data_bits` – The number of data bits per frame sent.
- `parity` – The type of parity used. See *uart\_parity\_t* above.
- `stop_bits` – The number of stop bits asserted at the of the frame.
- `tmr` – The resource id of the timer to be used. Polling mode will be used if set to 0.
- `rx_buff` – Pointer to a buffer. Optional. If set to zero the UART will run in blocking mode. If initialised to a valid buffer, the UART will be interrupt driven.
- `buffer_size_plus_one` – Size of the buffer if enabled in `rx_buff`. Note that the buffer allocation and size argument must be one greater than needed. Eg. `buff[65]` for a 64 byte buffer.
- `uart_rx_complete_callback_fptr` – Callback function pointer for UART rx complete (one word) in buffered mode only. Optionally NULL.
- `uart_rx_error_callback_fptr` – Callback function pointer for UART rx errors The error is contained in `cb_code` in the *uart\_rx\_t* struct.
- `app_data` – A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

```

void uart_rx_blocking_init(uart_rx_t *uart, port_t rx_port, uint32_t baud_rate, uint8_t data_bits,
                          uart_parity_t parity, uint8_t stop_bits, hwtimer_t tmr, void
                          (*uart_rx_error_callback_fptr)(uart_callback_code_t callback_code, void
                          *app_data), void *app_data)

```

Initializes a UART Rx I/O interface. This API is fixed to blocking mode which is where the call to `uart_rx` returns as soon as the stop bit has been sampled.

### Parameters

- `uart` – The `uart_rx_t` context to initialise.
- `rx_port` – The port used receive the UART frames.
- `baud_rate` – The baud rate of the UART in bits per second.
- `data_bits` – The number of data bits per frame sent.
- `parity` – The type of parity used. See `uart_parity_t` above.
- `stop_bits` – The number of stop bits asserted at the of the frame.
- `tmr` – The resource id of the timer to be used. Polling mode will be used if set to 0.
- `uart_rx_error_callback_fptr` – Callback function pointer for UART rx errors The error is contained in `cb_code` in the `uart_rx_t` struct.
- `app_data` – A pointer to application specific data provided by the application. Used to share data between the error callback function and the application.

```
uint8_t uart_rx(uart_rx_t *uart)
```

Receives a single UART frame with parameters as specified in `uart_rx_init()`

#### Parameters

- `uart` – The `uart_rx_t` context to receive from.

#### Returns

The word received in the UART frame. In buffered mode it gets the oldest received word.

```
void uart_rx_deinit(uart_rx_t *uart)
```

De-initializes the specified UART Rx interface. This disables the port also. The timer, if used, needs to be freed by the application.

#### Parameters

- `uart` – The `uart_rx_t` context to de-initialise.

```
struct uart_rx_t
```

`#include <uart.h>` Struct to hold a UART Rx context.

The members in this struct should not be accessed directly. Use the API provided instead.

## 2.2 I<sup>2</sup>C Library

A software defined I<sup>2</sup>C library that allows you to control an I<sup>2</sup>C bus via xcore ports. I<sup>2</sup>C is a two-wire hardware serial interface, first developed by Philips. The components in the library are controlled via C and can either act as I<sup>2</sup>C master or slave.

The library is compatible with multiple slave devices existing on the same bus. The I<sup>2</sup>C master component can be used by multiple tasks within the xcore device (each addressing the same or different slave devices).

The library can also be used to implement multiple I<sup>2</sup>C physical interfaces on a single xcore device simultaneously.

All signals are designed to comply with the timings in the I<sup>2</sup>C specification.

Note that the following optional parts of the I<sup>2</sup>C specification are not supported:

- Multi-master arbitration
- 10-bit slave addressing
- General call addressing
- Software reset
- START byte

- Device ID
- Fast-mode Plus, High-speed mode, Ultra Fast-mode

I<sup>2</sup>C consists of two signals: a clock line (SCL) and a data line (SDA). Both these signals are open-drain and require external resistors to pull the line up if no device is driving the signal down. The correct value for the resistors can be found in the I<sup>2</sup>C specification.

All I<sup>2</sup>C functions can be accessed via the `i2c.h` header:

```
#include <i2c.h>
```

## 2.2.1 I<sup>2</sup>C Master

### I<sup>2</sup>C Master Usage

The following code snippet demonstrates the basic usage of an I<sup>2</sup>C master device.

```
#include <xs1.h>
#include "i2c.h"

i2c_master_t i2c_ctx;

port_t p_scl = XS1_PORT_1A;
port_t p_sda = XS1_PORT_1B;

uint8_t data[1] = {0x99};

// Initialize the master
i2c_master_init(
    &i2c_ctx,
    p_scl, 0, 0,
    p_sda, 0, 0,
    100);

// Write some data
i2c_master_write(&i2c_ctx, 0x33, data, 1, NULL, 1);

// Shutdown
i2c_master_shutdown(&i2c_ctx) ;
```

### I<sup>2</sup>C Master API

The following structures and functions are used to initialize and start an I<sup>2</sup>C master instance.

```
enum i2c_res_t
```

Status codes for I2C master operations

Values:

```
enumerator I2C_NACK
```

The slave has NACKed the last byte.

```
enumerator I2C_ACK
```

The slave has ACKed the last byte.

enumerator `I2C_STARTED`

The requested I2C transaction has started.

enumerator `I2C_NOT_STARTED`

The requested I2C transaction could not start.

```
typedef struct i2c_master_struct i2c_master_t
```

Type representing an I2C master context

```
i2c_res_t i2c_master_write(i2c_master_t *ctx, uint8_t device_addr, uint8_t buf[], size_t n, size_t
                        *num_bytes_sent, int send_stop_bit)
```

Writes data to an I2C bus as a master.

#### Parameters

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to write to.
- `buf` – The buffer containing data to write.
- `n` – The number of bytes to write.
- `num_bytes_sent` – The function will set this value to the number of bytes actually sent. On success, this will be equal to `n` but it will be less if the slave sends an early NACK on the bus and the transaction fails.
- `send_stop_bit` – If this is non-zero then a stop bit will be sent on the bus after the transaction. This is usually required for normal operation. If this parameter is zero then no stop bit will be omitted. In this case, no other task can use the component until a stop bit has been sent.

#### Returns

`I2C_ACK` if the write was acknowledged by the device, `I2C_NACK` otherwise.

```
i2c_res_t i2c_master_read(i2c_master_t *ctx, uint8_t device_addr, uint8_t buf[], size_t n, int send_stop_bit)
```

Reads data from an I2C bus as a master.

#### Parameters

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to read from.
- `buf` – The buffer to fill with data.
- `n` – The number of bytes to read.
- `send_stop_bit` – If this is non-zero then a stop bit will be sent on the bus after the transaction. This is usually required for normal operation. If this parameter is zero then no stop bit will be omitted. In this case, no other task can use the component until a stop bit has been sent.

#### Returns

`I2C_ACK` if the read was acknowledged by the device, `I2C_NACK` otherwise.

```
void i2c_master_stop_bit_send(i2c_master_t *ctx)
```

Send a stop bit to an I2C bus as a master.

This function will cause a stop bit to be sent on the bus. It should be used to complete/abort a transaction if the `send_stop_bit` argument was not set when calling the `i2c_master_read()` or `i2c_master_write()` functions.

#### Parameters

- `ctx` – A pointer to the I2C master context to use.

```
void i2c_master_init(i2c_master_t *ctx, const port_t p_scl, const uint32_t scl_bit_position, const uint32_t
                    scl_other_bits_mask, const port_t p_sda, const uint32_t sda_bit_position, const
                    uint32_t sda_other_bits_mask, const unsigned kbits_per_second)
```

Implements an I2C master device on one or two single or multi-bit ports.

#### Parameters

- `ctx` – A pointer to the I2C master context to initialize.
- `p_scl` – The port containing SCL. This may be either the same as or different than `p_sda`.
- `scl_bit_position` – The bit number of the SCL line on the port `p_scl`.
- `scl_other_bits_mask` – A value that is ORed into the port value driven to `p_scl` both when SCL is high and low. The bit representing SCL (as well as SDA if they share the same port) must be set to 0.
- `p_sda` – The port containing SDA. This may be either the same as or different than `p_scl`.
- `sda_bit_position` – The bit number of the SDA line on the port `p_sda`.
- `sda_other_bits_mask` – A value that is ORed into the port value driven to `p_sda` both when SDA is high and low. The bit representing SDA (as well as SCL if they share the same port) must be set to 0.
- `kbits_per_second` – The speed of the I2C bus. The maximum value allowed is 400.

```
void i2c_master_shutdown(i2c_master_t *ctx)
```

Shuts down the I2C master device.

This function disables the ports associated with the I2C master and deallocates its timer if it was not provided by the application.

If subsequent reads or writes need to be performed, then `i2c_master_init()` must be called again first.

#### Parameters

- `ctx` – A pointer to the I2C master context to shut down.

```
struct i2c_master_struct
```

`#include <i2c.h>` Struct to hold an I2C master context.

The members in this struct should not be accessed directly.

## 2.2.2 I<sup>2</sup>C Slave

### I<sup>2</sup>C Slave Usage

The following code snippet demonstrates the basic usage of an I<sup>2</sup>C slave device.

```
#include <xs1.h>
#include "i2c.h"

port_t p_scl = XS1_PORT_1A;
port_t p_sda = XS1_PORT_1B;

// Setup callbacks
// NOTE: See API or SDK examples for more on using the callbacks
i2c_callback_group_t i_i2c = {
```

(continues on next page)

(continued from previous page)

```

.ack_read_request = (ack_read_request_t) i2c_ack_read_req,
.ack_write_request = (ack_write_request_t) i2c_ack_write_req,
.master_requires_data = (master_requires_data_t) i2c_master_req_data,
.master_sent_data = (master_sent_data_t) i2c_master_sent_data,
.stop_bit = (stop_bit_t) i2c_stop_bit,
.shutdown = (shutdown_t) i2c_shutdown,
.app_data = NULL,
};

// Start the slave device in this thread
// NOTE: You may wish to launch the slave device in a different thread.
// See the XTC Tools documentation reference for lib_xcore.
i2c_slave(&i_i2c, p_scl, p_sda, 0x3c);

```

## I<sup>2</sup>C Slave API

The following structures and functions are used to initialize and start an I<sup>2</sup>C slave instance.

enum `i2c_slave_ack`

I2C Slave Response

This type is used to describe the I2C slave response.

Values:

enumerator `I2C_SLAVE_ACK`

ACK to accept request

enumerator `I2C_SLAVE_NACK`

NACK to ignore request

typedef enum `i2c_slave_ack` `i2c_slave_ack_t`

I2C Slave Response

This type is used to describe the I2C slave response.

typedef `i2c_slave_ack_t` (`*ack_read_request_t`)(void \*app\_data)

The bus master has requested a read.

This callback function is called if the bus master requests a read from this slave device.

At this point the slave can choose to accept the request (and drive an ACK signal back to the master) or not (and drive a NACK signal).

### Param `app_data`

A pointer to application specific data provided by the application. Used to share data between the callback functions and the application.

### Return

The callback must return either `I2C_SLAVE_ACK` or `I2C_SLAVE_NACK`.

typedef `i2c_slave_ack_t` (`*ack_write_request_t`)(void \*app\_data)

The bus master has requested a write.

This callback function is called if the bus master requests a write from this slave device.

At this point the slave can choose to accept the request (and drive an ACK signal back to the master) or not (and drive a NACK signal).

**Param app\_data**

A pointer to application specific data provided by the application. Used to share data between the callback functions and the application.

**Return**

The callback must return either *I2C\_SLAVE\_ACK* or *I2C\_SLAVE\_NACK*.

```
typedef uint8_t (*master_requires_data_t)(void *app_data)
```

The bus master requires data.

This callback function is called when the bus master requires data from this slave device.

**Param app\_data**

A pointer to application specific data provided by the application. Used to share data between the callback functions and the application.

**Return**

a byte of data to send to the master.

```
typedef i2c_slave_ack_t (*master_sent_data_t)(void *app_data, uint8_t data)
```

The bus master has sent some data.

This callback function is called when the bus master has transferred a byte of data this slave device.

**Param app\_data**

A pointer to application specific data provided by the application. Used to share data between the callback functions and the application.

**Param data**

The byte of data received from the bus master.

**Return**

The callback must return either *I2C\_SLAVE\_ACK* or *I2C\_SLAVE\_NACK*.

```
typedef void (*stop_bit_t)(void *app_data)
```

The bus master has sent a stop bit.

This callback function is called when a stop bit is sent by the bus master.

**Param app\_data**

A pointer to application specific data provided by the application. Used to share data between the callback functions and the application.

```
typedef int (*shutdown_t)(void *app_data)
```

Shuts down the I2C slave device.

This function can be used to stop the I2C slave task. It will disable the SCL and SDA ports and then return.

**Param app\_data**

A pointer to application specific data provided by the application. Used to share data between the callback functions and the application.

**Return**

- Non-zero if the I2C slave task should shut down.

- Zero if the I2C slave task should continue running.



```
void i2c_slave(const i2c_callback_group_t *const i2c_cbg, port_t p_scl, port_t p_sda, uint8_t device_addr)
    I2C slave task.
```

This function instantiates an I2C slave device.

#### Parameters

- *i2c\_cbg* – The I2C callback group pointing to the application’s functions to use for initialization and getting and receiving frames. Also points to application specific data which will be shared between the callbacks.
- *p\_scl* – The SCL port of the I2C bus. This should be a 1 bit port. If not, The SCL pin must be at bit 0 and the other bits unused.
- *p\_sda* – The SDA port of the I2C bus. This should be a 1 bit port. If not, The SDA pin must be at bit 0 and the other bits unused.
- *device\_addr* – The address of the slave device.

```
I2C_CALLBACK_ATTR
```

This attribute must be specified on all I2C callback functions provided by the application.

```
struct i2c_callback_group_t
```

*#include <i2c.h>* Callback group representing callback events that can occur during the operation of the I2C slave task. Must be initialized by the application prior to passing it to one of the I2C tasks.

## 2.2.3 I<sup>2</sup>C Registers

### I<sup>2</sup>C Register API

The following structures and functions are used to read and write I<sup>2</sup>C registers.

```
enum i2c_regop_res_t
```

This type is used by the supplementary I2C register read/write functions to report back on whether the operation was a success or not.

*Values:*

```
enumerator I2C_REGOP_SUCCESS
```

The operation was successful.

```
enumerator I2C_REGOP_DEVICE_NACK
```

The operation was NACKed when sending the device address, so either the device is missing or busy.

```
enumerator I2C_REGOP_INCOMPLETE
```

The operation was NACKed halfway through by the slave.

```
inline uint8_t read_reg(i2c_master_t *ctx, uint8_t device_addr, uint8_t reg, i2c_regop_res_t *result)
```

Read an 8-bit register on a slave device.

This function reads from an 8-bit addressed, 8-bit register in an I2C device. The function reads the data by sending the register address followed reading the register data from the device at the specified device address.

---

**Note:** No stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

---

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to read from.
- `reg` – The address of the register to read from.
- `result` – Indicates whether the read completed successfully. Will be set to [I2C\\_REGOP\\_DEVICE\\_NACK](#) if the slave NACKed, and [I2C\\_REGOP\\_SUCCESS](#) on successful completion of the read.

**Returns**

The value of the register.

```
inline uint8_t read_reg8_addr16(i2c_master_t *ctx, uint8_t device_addr, uint16_t reg, i2c_regop_res_t
                               *result)
```

Read an 8-bit register on a slave device.

This function reads from an 16-bit addressed, 8-bit register in an I2C device. The function reads the data by sending the register address followed reading the register data from the device at the specified device address.

---

**Note:** No stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

---

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to read from.
- `reg` – The address of the register to read from.
- `result` – Indicates whether the read completed successfully. Will be set to [I2C\\_REGOP\\_DEVICE\\_NACK](#) if the slave NACKed, and [I2C\\_REGOP\\_SUCCESS](#) on successful completion of the read.

**Returns**

The value of the register.

```
inline uint16_t read_reg16_addr8(i2c_master_t *ctx, uint8_t device_addr, uint8_t reg, i2c_regop_res_t
                                 *result)
```

Read an 16-bit register on a slave device.

This function reads from an 8-bit addressed, 16-bit register in an I2C device. The function reads the data by sending the register address followed reading the register data from the device at the specified device address.

---

**Note:** No stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

---

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to read from.
- `reg` – The address of the register to read from.
- `result` – Indicates whether the read completed successfully. Will be set to [I2C\\_REGOP\\_DEVICE\\_NACK](#) if the slave NACKed, and [I2C\\_REGOP\\_SUCCESS](#) on successful completion of the read.

**Returns**

The value of the register.

```
inline uint16_t read_reg16(i2c_master_t *ctx, uint8_t device_addr, uint16_t reg, i2c_regop_res_t *result)
```

Read an 16-bit register on a slave device.

This function reads from an 16-bit addressed, 16-bit register in an I2C device. The function reads the data by sending the register address followed reading the register data from the device at the specified device address.

---

**Note:** No stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

---

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to read from.
- `reg` – The address of the register to read from.
- `result` – Indicates whether the read completed successfully. Will be set to [I2C\\_REGOP\\_DEVICE\\_NACK](#) if the slave NACKed, and [I2C\\_REGOP\\_SUCCESS](#) on successful completion of the read.

**Returns**

The value of the register.

```
inline i2c_regop_res_t write_reg(i2c_master_t *ctx, uint8_t device_addr, uint8_t reg, uint8_t data)
```

Write to an 8-bit register on an I2C device.

This function writes to an 8-bit addressed, 8-bit register in an I2C device. The function writes the data by sending the register address followed by the register data to the device at the specified device address.

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to write to.
- `reg` – The address of the register to write to.
- `data` – The 8-bit value to write.

**Returns**

[I2C\\_REGOP\\_DEVICE\\_NACK](#) if the address is NACKed.

**Returns**

[I2C\\_REGOP\\_INCOMPLETE](#) if not all data was ACKed.

**Returns**

[I2C\\_REGOP\\_SUCCESS](#) on successful completion of the write.

```
inline i2c_regop_res_t write_reg8_addr16(i2c_master_t *ctx, uint8_t device_addr, uint16_t reg, uint8_t data)
```

Write to an 8-bit register on an I2C device.

This function writes to a 16-bit addressed, 8-bit register in an I2C device. The function writes the data by sending the register address followed by the register data to the device at the specified device address.

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to write to.
- `reg` – The address of the register to write to.

- `data` – The 8-bit value to write.

**Returns**

`I2C_REGOP_DEVICE_NACK` if the address is NACKed.

**Returns**

`I2C_REGOP_INCOMPLETE` if not all data was ACKed.

**Returns**

`I2C_REGOP_SUCCESS` on successful completion of the write.

```
inline i2c_regop_res_t write_reg16_addr8(i2c_master_t *ctx, uint8_t device_addr, uint8_t reg, uint16_t data)
```

Write to a 16-bit register on an I2C device.

This function writes to an 8-bit addressed, 16-bit register in an I2C device. The function writes the data by sending the register address followed by the register data to the device at the specified device address.

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to write to.
- `reg` – The address of the register to write to.
- `data` – The 16-bit value to write.

**Returns**

`I2C_REGOP_DEVICE_NACK` if the address is NACKed.

**Returns**

`I2C_REGOP_INCOMPLETE` if not all data was ACKed.

**Returns**

`I2C_REGOP_SUCCESS` on successful completion of the write.

```
inline i2c_regop_res_t write_reg16(i2c_master_t *ctx, uint8_t device_addr, uint16_t reg, uint16_t data)
```

Write to a 16-bit register on an I2C device.

This function writes to a 16-bit addressed, 16-bit register in an I2C device. The function writes the data by sending the register address followed by the register data to the device at the specified device address.

**Parameters**

- `ctx` – A pointer to the I2C master context to use.
- `device_addr` – The address of the device to write to.
- `reg` – The address of the register to write to.
- `data` – The 16-bit value to write.

**Returns**

`I2C_REGOP_DEVICE_NACK` if the address is NACKed.

**Returns**

`I2C_REGOP_INCOMPLETE` if not all data was ACKed.

**Returns**

`I2C_REGOP_SUCCESS` on successful completion of the write.

## 2.3 I<sup>2</sup>S Library

A software defined library that allows you to control an I<sup>2</sup>S (Inter-IC Sound) bus via xcore ports. I<sup>2</sup>S is a digital data streaming interfaces particularly appropriate for transmission of audio data. TDM is a special case of I<sup>2</sup>S which supports transport of more than two audio channels and is partially included in the library at this

time. The components in the library are controlled via C and can either act as I<sup>2</sup>S master, I<sup>2</sup>S slave or TDM slave.

**Note:** TDM is only currently supported as a TDM16 slave Tx component. Expansion of this library to support master or slave Rx is possible and can be done on request.

I<sup>2</sup>S is a protocol between two devices where one is the *master* and one is the *slave* which determines who drives the clock lines. The protocol is made up of four signals shown in [I<sup>2</sup>S data wires](#).

Table 2.2: I<sup>2</sup>S data wires

<i>MCLK</i>	Clock line, driven by external oscillator. This signal is optional.
<i>BCLK</i>	Bit clock. This is a fixed divide of the <i>MCLK</i> and is driven by the master.
<i>LRCLK</i> (or <i>WCLK</i> )	Word clock (or word select). This is driven by the master.
<i>DATA</i>	Data line, driven by one of the slave or master depending on the data direction. There may be several data lines in differing directions.

All I<sup>2</sup>S functions can be accessed via the `i2s.h` header:

```
#include "i2s.h"
```

TDM is a protocol between two devices similar to I<sup>2</sup>S where one is the *master* and one is the *slave* which determines who drives the clock lines. The protocol is made up of four signals shown in [TDM data wires](#).

Table 2.3: TDM data wires

<i>MCLK</i>	Clock line, driven by external oscillator. This signal is optional.
<i>BCLK</i>	Bit clock. This is a fixed divide of the <i>MCLK</i> and is driven by the master.
<i>FSY- CNH</i>	Frame synchronization. Toggles at the start of the TDM data frame. This is driven by the master.
<i>DATA</i>	Data line, driven by one of the slave or master depending on the data direction. There may be several data lines in differing directions.

Currently supported TDM functions can be accessed via the `i2s_tdm_slave.h` header:

```
#include "i2s_tdm_slave.h"
```

## 2.3.1 I<sup>2</sup>S Common API

### I<sup>2</sup>S Instances

The following structures and functions are used by an I<sup>2</sup>S master or slave instance.

```
enum i2s_mode
```

I2S mode.

This type is used to describe the I2S mode.

*Values:*

```
enumerator I2S_MODE_I2S
```

The LR clock transitions ahead of the data by one bit clock.

enumerator `I2S_MODE_LEFT_JUSTIFIED`

The LR clock and data are phase aligned.

enum `i2s_slave_bclk_polarity`

I2S slave bit clock polarity.

Standard I2S is positive, that is toggle data and LR clock on falling edge of bit clock and sample them on rising edge of bit clock. Some masters have it the other way around.

*Values:*

enumerator `I2S_SLAVE_SAMPLE_ON_BCLK_RISING`

Toggle falling, sample rising (default if not set)

enumerator `I2S_SLAVE_SAMPLE_ON_BCLK_FALLING`

Toggle rising, sample falling

enum `i2s_restart`

Restart command type.

Restart commands that can be signalled to the I2S or TDM component.

*Values:*

enumerator `I2S_NO_RESTART`

Do not restart.

enumerator `I2S_RESTART`

Restart the bus (causes the I2S/TDM to stop and a new init callback to occur allowing reconfiguration of the BUS).

enumerator `I2S_SHUTDOWN`

Shutdown. This will cause the I2S/TDM component to exit.

typedef enum `i2s_mode` `i2s_mode_t`

I2S mode.

This type is used to describe the I2S mode.

typedef enum `i2s_slave_bclk_polarity` `i2s_slave_bclk_polarity_t`

I2S slave bit clock polarity.

Standard I2S is positive, that is toggle data and LR clock on falling edge of bit clock and sample them on rising edge of bit clock. Some masters have it the other way around.

typedef struct `i2s_config` `i2s_config_t`

I2S configuration structure.

This structure describes the configuration of an I2S bus.

typedef enum `i2s_restart` `i2s_restart_t`

Restart command type.

Restart commands that can be signalled to the I2S or TDM component.

```
typedef void (*i2s_init_t)(void *app_data, i2s_config_t *i2s_config)
```

I2S initialization event callback.

The I2S component will call this when it first initializes on first run of after a restart.

This will contain the TDM context when in TDM mode.

**Param app\_data**

Points to application specific data supplied by the application. May be used for context data specific to each I2S task instance.

**Param i2s\_config**

This structure is provided if the connected component drives an I2S bus. The members of the structure should be set to the required configuration. This is ignored when used in TDM mode.

```
typedef i2s_restart_t (*i2s_restart_check_t)(void *app_data)
```

I2S restart check callback.

This callback is called once per frame. The application must return the required restart behavior.

**Param app\_data**

Points to application specific data supplied by the application. May be used for context data specific to each I2S task instance.

**Return**

The return value should be set to *I2S\_NO\_RESTART*, *I2S\_RESTART* or *I2S\_SHUTDOWN*.

```
typedef void (*i2s_receive_t)(void *app_data, size_t num_in, const int32_t *samples)
```

Receive an incoming frame of samples.

This callback will be called when a new frame of samples is read in by the I2S task.

**Param app\_data**

Points to application specific data supplied by the application. May be used for context data specific to each I2S task instance.

**Param num\_in**

The number of input channels contained within the array.

**Param samples**

The samples data array as signed 32-bit values. The component may not have 32-bits of accuracy (for example, many I2S codecs are 24-bit), in which case the bottom bits will be arbitrary values.

```
typedef void (*i2s_send_t)(void *app_data, size_t num_out, int32_t *samples)
```

Request an outgoing frame of samples.

This callback will be called when the I2S task needs a new frame of samples.

**Param app\_data**

Points to application specific data supplied by the application. May be used for context data specific to each I2S task instance.

**Param num\_out**

The number of output channels contained within the array.

**Param samples**

The samples data array as signed 32-bit values. The component may not have 32-bits of accuracy (for example, many I2S codecs are 24-bit), in which case the bottom bits will be arbitrary values.

`I2S_MAX_DATA_LINES`

`I2S_CHANS_PER_FRAME`

`I2S_CALLBACK_ATTR`

This attribute must be specified on all I2S callback functions provided by the application.

`struct i2s_config`

*#include <i2s.h>* I2S configuration structure.

This structure describes the configuration of an I2S bus.

`struct i2s_callback_group_t`

*#include <i2s.h>* Callback group representing callback events that can occur during the operation of the I2S task. Must be initialized by the application prior to passing it to one of the I2S tasks.

## TDM Instances

The following structures and functions are used by an TDM master or slave instance.

`typedef void (*tdm_post_port_init_t)(void *i2s_tdm_ctx)`

TDM post resource initialization event callback.

The TDM component will call this after it first initializes the ports. This gives the app the chance to make adjustments to port timing which are often needed when clocking above 15MHz.

### Param `i2s_tdm_ctx`

Points to *i2s\_tdm\_ctx\_t* struct allowing the resources to be modified after they have been enabled and initialised.

`I2S_TDM_MAX_POUT_CNT`

`I2S_TDM_MAX_PIN_CNT`

`I2S_TDM_MAX_CH_PER_FRAME`

`TDM_CALLBACK_ATTR`

This attribute must be specified on the TDM callback function provided by the application.

`struct i2s_tdm_ctx_t`

*#include <i2s\_tdm\_slave.h>* Struct to hold an I2S TDM context.

The members in this struct should not be accessed directly.

## 2.3.2 I<sup>2</sup>S Master

### I<sup>2</sup>S Master Usage

The following code snippet demonstrates the basic usage of an I<sup>2</sup>S master device.



```

#include <xs1.h>
#include "i2s.h"

port_t p_i2s_dout[1];
port_t p_bclk;
port_t p_lrclk;
port_t p_mclk;
xclock_t bclk;
i2s_callback_group_t i2s_cb_group;

// Setup ports and clocks
p_i2s_dout[0] = PORT_I2S_DAC_DATA;
p_bclk = PORT_I2S_BCLK;
p_lrclk = PORT_I2S_LRCLK;
p_mclk = PORT_MCLK_IN;
bclk = I2S_CLKBLK;

port_enable(p_mclk);
port_enable(p_bclk);
// NOTE: p_lrclk does not need to be enabled by the caller

// Setup callbacks
// NOTE: See API or SDK examples for more on using the callbacks
i2s_cb_group.init = (i2s_init_t) i2s_init;
i2s_cb_group.restart_check = (i2s_restart_check_t) i2s_restart_check;
i2s_cb_group.receive = (i2s_receive_t) i2s_receive;
i2s_cb_group.send = (i2s_send_t) i2s_send;
i2s_cb_group.app_data = NULL;

// Start the master device in this thread
// NOTE: You may wish to launch the slave device in a different thread.
//       See the XTC Tools documentation reference for lib_xcore.
i2s_master(&i2s_cb_group, p_i2s_dout, 1, NULL, 0, p_bclk, p_lrclk, p_mclk, bclk);

```

## I<sup>2</sup>S Master API

The following structures and functions are used to initialize and start an I<sup>2</sup>S master instance.

```
void i2s_master(const i2s\_callback\_group\_t *const i2s_cbg, const port_t p_dout[], const size_t num_out,
               const port_t p_din[], const size_t num_in, const port_t p_bclk, const port_t p_lrclk, const
               port_t p_mclk, const xclock_t bclk)
```

I2S master task

This task performs I2S on the provided pins. It will perform callbacks over the [i2s\\_callback\\_group\\_t](#) callback group to get/receive frames of data from the application using this component.

The task performs I2S master so will drive the word clock and bit clock lines.

### Parameters

- `i2s_cbg` – The I2S callback group pointing to the application's functions to use for initialization and getting and receiving frames. Also points to application specific data which will be shared between the callbacks.
- `p_dout` – An array of data output ports
- `num_out` – The number of output data ports
- `p_din` – An array of data input ports

- `num_in` – The number of input data ports
- `p_bclk` – The bit clock output port
- `p_lrclk` – The word clock output port
- `p_mclk` – Input port which supplies the master clock
- `bclk` – A clock that will get configured for use with the bit clock

```
void i2s_master_external_clock(const i2s_callback_group_t *const i2s_cbg, const port_t p_dout[], const
                             size_t num_out, const port_t p_din[], const size_t num_in, const port_t
                             p_bclk, const port_t p_lrclk, const xclock_t bclk)
```

I2S master task

This task differs from `i2s_master()` in that `bclk` must already be configured to the BCLK frequency. Other than that, it is identical.

This task performs I2S on the provided pins. It will perform callbacks over the `i2s_callback_group_t` callback group to get/receive frames of data from the application using this component.

The task performs I2S master so will drive the word clock and bit clock lines.

#### Parameters

- `i2s_cbg` – The I2S callback group pointing to the application's functions to use for initialization and getting and receiving frames. Also points to application specific data which will be shared between the callbacks.
- `p_dout` – An array of data output ports
- `num_out` – The number of output data ports
- `p_din` – An array of data input ports
- `num_in` – The number of input data ports
- `p_bclk` – The bit clock output port
- `p_lrclk` – The word clock output port
- `bclk` – A clock that is configured externally to be used as the bit clock

### 2.3.3 I<sup>2</sup>S Slave

#### I<sup>2</sup>S Slave Usage

The following code snippet demonstrates the basic usage of an I<sup>2</sup>S slave device.

```
#include <xs1.h>
#include "i2s.h"

// Setup ports and clocks
port_t p_bclk = XS1_PORT_1B;
port_t p_lrclk = XS1_PORT_1C;
port_t p_din [4] = {XS1_PORT_1D, XS1_PORT_1E, XS1_PORT_1F, XS1_PORT_1G};
port_t p_dout[4] = {XS1_PORT_1H, XS1_PORT_1I, XS1_PORT_1J, XS1_PORT_1K};
xclock_t bclk = XS1_CLKBLK_1;

port_enable(p_bclk);
// NOTE: p_lrclk does not need to be enabled by the caller

// Setup callbacks
// NOTE: See API or SDK examples for more on using the callbacks
```

(continues on next page)

(continued from previous page)

```

i2s_callback_group_t i_i2s = {
    .init = (i2s_init_t) i2s_init,
    .restart_check = (i2s_restart_check_t) i2s_restart_check,
    .receive = (i2s_receive_t) i2s_receive,
    .send = (i2s_send_t) i2s_send,
    .app_data = NULL,
};

// Start the slave device in this thread
// NOTE: You may wish to launch the slave device in a different thread.
// See the XTC Tools documentation reference for lib_xcore.
i2s_slave(&i_i2s, p_dout, 4, p_din, 4, p_bclk, p_lrclk, bclk);

```

## I<sup>2</sup>S Slave API

The following structures and functions are used to initialize and start an I<sup>2</sup>S slave instance.

```
void i2s_slave(const i2s\_callback\_group\_t *const i2s_cbg, port_t p_dout[], const size_t num_out, port_t
    p_din[], const size_t num_in, port_t p_bclk, port_t p_lrclk, xclock_t bclk)
```

I2S slave task

This task performs I2S on the provided pins. It will perform callbacks over the [i2s\\_callback\\_group\\_t](#) callback group to get/receive data from the application using this component.

The component performs I2S slave so will expect the word clock and bit clock to be driven externally.

### Parameters

- `i2s_cbg` – The I2S callback group pointing to the application's functions to use for initialization and getting and receiving frames. Also points to application specific data which will be shared between the callbacks.
- `p_dout` – An array of data output ports
- `num_out` – The number of output data ports
- `p_din` – An array of data input ports
- `num_in` – The number of input data ports
- `p_bclk` – The bit clock input port
- `p_lrclk` – The word clock input port
- `bclk` – A clock that will get configured for use with the bit clock

## 2.3.4 TDM Slave

### TDM Slave Tx Usage

The following code snippet demonstrates the basic usage of a TDM slave Tx device.

```

#include <xs1.h>
#include "i2s_tdm_slave.h"

// Setup ports and clocks
port_t p_bclk = XS1_PORT_1A;
port_t p_fsyc = XS1_PORT_1B;
port_t p_dout = XS1_PORT_1C;

```

(continues on next page)



(continued from previous page)

```

xclock_t clk_bclk = XS1_CLKBLK_1;

// Setup callbacks
// NOTE: See API or sln_voice examples for more on using the callbacks
i2s_tdm_ctx_t ctx;
i2s_callback_group_t i_i2s = {
    .init = (i2s_init_t) i2s_init,
    .restart_check = (i2s_restart_check_t) i2s_restart_check,
    .receive = NULL,
    .send = (i2s_send_t) i2s_send,
    .app_data = NULL,
};

// Initialize the TDM slave
i2s_tdm_slave_tx_16_init(
    &ctx,
    &i_i2s,
    p_dout,
    p_fsync,
    p_bclk,
    clk_bclk,
    0,
    I2S_SLAVE_SAMPLE_ON_BCLK_FALLING,
    NULL);

// Start the slave device in this thread
// NOTE: You may wish to launch the slave device in a different thread.
// See the XTC Tools documentation reference for lib_xcore.
i2s_tdm_slave_tx_16_thread(&ctx);

```

## TDM Slave Tx API

The following structures and functions are used to initialize and start a TDM slave Tx instance.

```

void i2s_tdm_slave_tx_16_init(i2s_tdm_ctx_t *ctx, i2s_callback_group_t *i2s_cbg, port_t p_dout, port_t
    p_fsync, port_t p_bclk, xclock_t bclk, uint32_t tx_offset,
    i2s_slave_bclk_polarity_t slave_bclk_polarity, tdm_post_port_init_t
    tdm_post_port_init)

```

I2S TDM slave context initialization for 16 channel TX only with 1 output port, 32b word length, 32b channel length, and 16 channels per frame.

This prepares a context for I2S TDM slave on the provided pins.

The resulting context can be used with [i2s\\_tdm\\_slave\\_tx\\_16\\_thread\(\)](#).

### Parameters

- `ctx` – A pointer to the I2S TDM context to use.
- `i2s_cbg` – The I2S callback group pointing to the application's functions to use for initialization and getting and receiving frames. For TDM the `app_data` variable within this struct is NOT used.
- `p_dout` – The data output port. MUST be a 1b port
- `p_fsync` – The fsync input port. MUST be a 1b port
- `p_bclk` – The bit clock input port. MUST be a 1b port



- `bclk` – A clock that will get configured for use with the bit clock
- `tx_offset` – The number of bclks from FSYNC transition to the MSB of Slot 0
- `slave_bclk_pol` – The polarity of bclk
- `tdm_post_port_init` – Callback to be called just after resource init. Allows for modification of port timing for >15MHz clocks. Set to NULL if not needed.

```
void i2s_tdm_slave_tx_16_thread(i2s_tdm_ctx_t *ctx)
```

I2S TDM TX 16 ch slave task

This task performs I2S TDM slave on the provided context which was initialized with `i2s_tdm_slave_tx_16_init()`. It will perform callbacks over the `i2s_callback_group_t` callback group to get data from the application using this component.

This thread assumes 1 data output port, 32b word length, 32b channel length, and 16 channels per frame.

The component performs I2S TDM slave so will expect the fsync and bit clock to be driven externally.

#### Parameters

- `ctx` – A pointer to the I2S TDM context to use.

## 2.4 SPI Library

A software defined SPI (serial peripheral interface) library that allows you to control a SPI bus via the xcore GPIO hardware-response ports. SPI is a four-wire hardware bi-directional serial interface. The components in the library are controlled via C and can either act as SPI master or slave.

The SPI bus can be used by multiple tasks within the xcore device and (each addressing the same or different slaves) and is compatible with other slave devices on the same bus.

The SPI protocol requires a clock, one or more slave selects and either one or two data wires.

Table 2.4: SPI data wires

<code>SCLK</code>	Clock line, driven by the master
<code>MOSI</code>	Master Output, Slave Input data line, driven by the master
<code>MISO</code>	Master Input, Slave Output data line, driven by the slave
<code>SS</code>	Slave select line, driven by the master

All SPI functions can be accessed via the `spi.h` header:

```
#include <spi.h>
```

### 2.4.1 SPI Master

#### SPI Master Usage

The following code snippet demonstrates the basic usage of an SPI master device.

```
#include <xs1.h>
#include "spi.h"
```

```
spi_master_t spi_ctx;
spi_master_device_t spi_dev;
```

(continues on next page)

(continued from previous page)

```

port_t p_miso = XS1_PORT_1A;
port_t p_ss[1] = {XS1_PORT_1B};
port_t p_sclk = XS1_PORT_1C;
port_t p_mosi = XS1_PORT_1D;
xclock_t cb = XS1_CLKBLK_1;

uint8_t tx[4] = {0x01, 0x02, 0x04, 0x08};
uint8_t rx[4];

// Initialize the master device
spi_master_init(&spi_ctx, cb, p_ss[0], p_sclk, p_mosi, p_miso);
spi_master_device_init(&spi_dev, &spi_ctx,
    1,
    SPI_MODE_0,
    spi_master_source_clock_ref,
    0,
    spi_master_sample_delay_0,
    0, 0 ,0 ,0 );

// Transfer some data
spi_master_start_transaction(&spi_ctx);
spi_master_transfer(&spi_ctx, (uint8_t *)tx, (uint8_t *)rx, 4);
spi_master_end_transaction(&spi_ctx);

```

## SPI Master API

The following structures and functions are used to initialize and start an SPI master instance.

```
enum spi_master_sample_delay_t
```

Enum type representing the different options for the SPI master sample delay.

*Values:*

```
enumerator spi_master_sample_delay_0
```

Samples 1/2 clock cycle after output from device

```
enumerator spi_master_sample_delay_1
```

Samples 3/4 clock cycle after output from device

```
enumerator spi_master_sample_delay_2
```

Samples 1 clock cycle after output from device

```
enumerator spi_master_sample_delay_3
```

Samples 1 and 1/4 clock cycle after output from device

```
enumerator spi_master_sample_delay_4
```

Samples 1 and 1/2 clock cycle after output from device

```
enum spi_master_source_clock_t
```

Enum type used to set which of the two clock sources SCLK is derived from.

*Values:*

enumerator `spi_master_source_clock_ref`

SCLK is derived from the 100 MHz reference clock

enumerator `spi_master_source_clock_xcore`

SCLK is derived from the core clock

```
typedef void (*slave_transaction_started_t)(void *app_data, uint8_t **out_buf, size_t *outbuf_len,
uint8_t **in_buf, size_t *inbuf_len)
```

Master has started a transaction

This callback function will be called when the SPI master has asserted this slave's chip select.

The input and output buffer may be the same; however, partial byte/incomplete reads will result in `out_buf` bits being masked off due to a partial bit output.

**Param `app_data`**

A pointer to application specific data provided by the application. Used to share data between

**Param `out_buf`**

The buffer to send to the master

**Param `outbuf_len`**

The length in bytes of `out_buf`

**Param `in_buf`**

The buffer to receive into from the master

**Param `inbuf_len`**

The length in bytes of `in_buf`

```
typedef void (*slave_transaction_ended_t)(void *app_data, uint8_t **out_buf, size_t bytes_written, uint8_t
**in_buf, size_t bytes_read, size_t read_bits)
```

Master has ended a transaction

This callback function will be called when the SPI master has de-asserted this slave's chip select.

The value of `bytes_read` contains the number of full bytes that are in `in_buf`. When `read_bits` is greater than 0, the byte after the last full byte contains the partial bits read.

**Param `app_data`**

A pointer to application specific data provided by the application. Used to share data between

**Param `out_buf`**

The buffer that had been provided to be sent to the master

**Param `bytes_written`**

The length in bytes of `out_buf` that had been written

**Param `in_buf`**

The buffer that had been provided to be received into from the master

**Param `bytes_read`**

The length in bytes of `in_buf` that has been read in to

**Param `read_bits`**

The length in bits of `in_buf`

```
void spi_master_init(spi_master_t *spi, xclock_t clock_block, port_t cs_port, port_t sclk_port, port_t
mosi_port, port_t miso_port)
```

Initializes a SPI master I/O interface.

Note: To guarantee timing in all situations, the SPI I/O interface implicitly sets the fast mode and high priority status register bits for the duration of SPI operations. This may reduce the MIPS of other threads based on overall system setup.

### Parameters

- `spi` – The `spi_master_t` context to initialize.
- `clock_block` – The clock block to use for the SPI master interface.
- `cs_port` – The SPI interface's chip select port. This may be a multi-bit port.
- `sclk_port` – The SPI interface's SCLK port. Must be a 1-bit port.
- `mosi_port` – The SPI interface's MOSI port. Must be a 1-bit port.
- `miso_port` – The SPI interface's MISO port. Must be a 1-bit port.

```
void spi_master_device_init(spi_master_device_t *dev, spi_master_t *spi, uint32_t cs_pin, int cpol, int
    cpha, spi_master_source_clock_t source_clock, uint32_t clock_divisor,
    spi_master_sample_delay_t miso_sample_delay, uint32_t miso_pad_delay,
    uint32_t cs_to_clk_delay_ticks, uint32_t clk_to_cs_delay_ticks, uint32_t
    cs_to_cs_delay_ticks)
```

Initialize a SPI device. Multiple SPI devices may be initialized per SPI interface. Each must be on a unique pin of the interface's chip select port.

### Parameters

- `dev` – The context representing the device to initialize.
- `spi` – The context representing the SPI master interface that the device is connected to.
- `cs_pin` – The bit number of the chip select port that is connected to the device's chip select pin.
- `cpol` – The clock polarity required by the device.
- `cpha` – The clock phase required by the device.
- `source_clock` – The source clock to derive SCLK from. See `spi_master_source_clock_t`.
- `clock_divisor` – The value to divide the source clock by. The frequency of SCLK will be set to:
  - $(F_{src}) / (4 * \text{clock\_divisor})$  when `clock_divisor > 0`
  - $(F_{src}) / (2)$  when `clock_divisor = 0` Where  $F_{src}$  is the frequency of the source clock.
- `miso_sample_delay` – When to sample MISO. See `spi_master_sample_delay_t`.
- `miso_pad_delay` – The number of core clock cycles to delay sampling the MISO pad during a transaction. This allows for more fine grained adjustment of sampling time. The value may be between 0 and 5.
- `cs_to_clk_delay_ticks` – The minimum number of reference clock ticks between assertion of chip select and the first clock edge.
- `clk_to_cs_delay_ticks` – The minimum number of reference clock ticks between the last clock edge and de-assertion of chip select.
- `cs_to_cs_delay_ticks` – The minimum number of reference clock ticks between transactions, which is between de-assertion of chip select and the end of one transaction, and its re-assertion at the beginning of the next.



```
void spi_master_start_transaction(spi_master_device_t *dev)
```

Starts a SPI transaction with the specified SPI device. This leaves chip select asserted.

#### Parameters

- `dev` – The SPI device with which to start a transaction.

```
void spi_master_transfer(spi_master_device_t *dev, uint8_t *data_out, uint8_t *data_in, size_t len)
```

Transfers data to/from the specified SPI device. This may be called multiple times during a single transaction.

#### Parameters

- `dev` – The SPI device with which to transfer data.
- `data_out` – Buffer containing the data to send to the device. May be NULL if no data needs to be sent.
- `data_in` – Buffer to save the data received from the device. May be NULL if the data received is not needed.
- `len` – The length in bytes of the data to transfer. Both buffers must be at least this large if not NULL.

```
inline void spi_master_delay_before_next_transfer(spi_master_device_t *dev, uint32_t delay_ticks)
```

Enforces a minimum delay between the time this is called and the next transfer. It must be called during a transaction. It returns immediately.

#### Parameters

- `dev` – The active SPI device.
- `delay_ticks` – The number of reference clock ticks to delay.

```
void spi_master_end_transaction(spi_master_device_t *dev)
```

Ends a SPI transaction with the specified SPI device. This leaves chip select de-asserted.

#### Parameters

- `dev` – The SPI device with which to end a transaction.

```
void spi_master_deinit(spi_master_t *spi)
```

De-initializes the specified SPI master interface. This disables the ports and clock block.

#### Parameters

- `spi` – The *spi\_master\_t* context to de-initialize.

`SPI_MODE_0`

Convenience macro that may be used to specify SPI Mode 0 to *spi\_master\_device\_init()* or *spi\_slave()* in place of `cpol` and `cpha`.

`SPI_MODE_1`

Convenience macro that may be used to specify SPI Mode 1 to *spi\_master\_device\_init()* or *spi\_slave()* in place of `cpol` and `cpha`.

`SPI_MODE_2`

Convenience macro that may be used to specify SPI Mode 2 to *spi\_master\_device\_init()* or *spi\_slave()* in place of `cpol` and `cpha`.

`SPI_MODE_3`

Convenience macro that may be used to specify SPI Mode 3 to *spi\_master\_device\_init()* or *spi\_slave()* in place of `cpol` and `cpha`.

**SPI\_CALLBACK\_ATTR**

This attribute must be specified on all SPI callback functions provided by the application.

**struct spi\_master\_t**

*#include <spi.h>* Struct to hold a SPI master context.

The members in this struct should not be accessed directly.

**struct spi\_master\_device\_t**

*#include <spi.h>* Struct type representing a SPI device connected to a SPI master interface.

The members in this struct should not be accessed directly.

## 2.4.2 SPI Slave

### SPI Slave Usage

The following code snippet demonstrates the basic usage of an SPI slave device.

```
#include <xs1.h>
#include "spi.h"

// Setup callbacks
// NOTE: See API or SDK examples for more on using the callbacks
spi_slave_callback_group_t spi_cbg = {
    .slave_transaction_started = (slave_transaction_started_t) start,
    .slave_transaction_ended = (slave_transaction_ended_t) end,
    .app_data = NULL
};

port_t p_miso = XSi_PORT_1A;
port_t p_cs   = XSi_PORT_1B;
port_t p_sclk = XSi_PORT_1C;
port_t p_mosi = XSi_PORT_1D;
xclock_t cb   = XSi_CLKBLK_1;

// Start the slave device in this thread
// NOTE: You may wish to launch the slave device in a different thread.
// See the XTC Tools documentation reference for lib_xcore.
spi_slave(&spi_cbg, p_sclk, p_mosi, p_miso, p_cs, cb, SPI_MODE_0);
```

### SPI Slave API

The following structures and functions are used to initialize and start an SPI slave instance.

```
void spi_slave(const spi\_slave\_callback\_group\_t *spi_cbg, port_t p_sclk, port_t p_mosi, port_t p_miso,
               port_t p_cs, xclock_t clk, int cpol, int cpha, uint32_t thread_mode)
```

Initializes a SPI slave.

The CS to first clock minimum delay, sometimes referred to as setup time, will vary based on the duration of the `slave_transaction_started` callback. This parameter will be application specific. To determine the typical value, time the duration of the `slave_transaction_started` callback, and add 2000ns as a safety factor. If `slave_transaction_started` has a non-deterministic runtime, perhaps due to waiting

on an XCORE resource, then the application developer must decide an appropriate CS to first SCLK specification.

The minimum delay between consecutive transactions varies based on SPI mode, and if MISO is used.

---

**Note:** Verified at 25000 kbps, with a 2000ns CS assertion to first clock in all modes.

---

struct `spi_slave_callback_group_t`

*#include <spi.h>* Callback group representing callback events that can occur during the operation of the SPI slave task. Must be initialized by the application prior to passing it to one of the SPI slaves.

### 3 Copyright & Disclaimer

---

Copyright © 2023, XMOS Ltd

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

## 4 Licenses

---

### 4.1 XMOS

All original source code is licensed under the [XMOS License](#).

## 4 Index

---

### A

`ack_read_request_t` (C type), 13  
`ack_write_request_t` (C type), 13

### H

`HIL_UART_RX_CALLBACK_ATTR` (C macro), 5  
`HIL_UART_TX_CALLBACK_ATTR` (C macro), 5

### I

`I2C_CALLBACK_ATTR` (C macro), 15  
`i2c_callback_group_t` (C struct), 15  
`i2c_master_init` (C function), 11  
`i2c_master_read` (C function), 11  
`i2c_master_shutdown` (C function), 12  
`i2c_master_stop_bit_send` (C function), 11  
`i2c_master_struct` (C struct), 12  
`i2c_master_t` (C type), 11  
`i2c_master_write` (C function), 11  
`i2c_regop_res_t` (C enum), 15  
`i2c_regop_res_t.I2C_REGOP_DEVICE_NACK` (C enumerator), 15  
`i2c_regop_res_t.I2C_REGOP_INCOMPLETE` (C enumerator), 15  
`i2c_regop_res_t.I2C_REGOP_SUCCESS` (C enumerator), 15  
`i2c_res_t` (C enum), 10  
`i2c_res_t.I2C_ACK` (C enumerator), 10  
`i2c_res_t.I2C_NACK` (C enumerator), 10  
`i2c_res_t.I2C_NOT_STARTED` (C enumerator), 11  
`i2c_res_t.I2C_STARTED` (C enumerator), 10  
`i2c_slave` (C function), 14  
`i2c_slave_ack` (C enum), 13  
`i2c_slave_ack.I2C_SLAVE_ACK` (C enumerator), 13  
`i2c_slave_ack.I2C_SLAVE_NACK` (C enumerator), 13  
`i2c_slave_ack_t` (C type), 13  
`I2S_CALLBACK_ATTR` (C macro), 22  
`i2s_callback_group_t` (C struct), 22  
`I2S_CHANS_PER_FRAME` (C macro), 22  
`i2s_config` (C struct), 22  
`i2s_config_t` (C type), 20  
`i2s_init_t` (C type), 20  
`i2s_master` (C function), 23  
`i2s_master_external_clock` (C function), 24  
`I2S_MAX_DATA_LINES` (C macro), 21  
`i2s_mode` (C enum), 19  
`i2s_mode.I2S_MODE_I2S` (C enumerator), 19  
`i2s_mode.I2S_MODE_LEFT_JUSTIFIED` (C enumerator), 19  
`i2s_mode_t` (C type), 20  
`i2s_receive_t` (C type), 21  
`i2s_restart` (C enum), 20  
`i2s_restart.I2S_NO_RESTART` (C enumerator), 20

`i2s_restart.I2S_RESTART` (C enumerator), 20  
`i2s_restart.I2S_SHUTDOWN` (C enumerator), 20  
`i2s_restart_check_t` (C type), 21  
`i2s_restart_t` (C type), 20  
`i2s_send_t` (C type), 21  
`i2s_slave` (C function), 25  
`i2s_slave_bclk_polarity` (C enum), 20  
`i2s_slave_bclk_polarity.I2S_SLAVE_SAMPLE_ON_BCLK_FALLING` (C enumerator), 20  
`i2s_slave_bclk_polarity.I2S_SLAVE_SAMPLE_ON_BCLK_RISING` (C enumerator), 20  
`i2s_slave_bclk_polarity_t` (C type), 20  
`i2s_tdm_ctx_t` (C struct), 22  
`I2S_TDM_MAX_CH_PER_FRAME` (C macro), 22  
`I2S_TDM_MAX_PIN_CNT` (C macro), 22  
`I2S_TDM_MAX_POUT_CNT` (C macro), 22  
`i2s_tdm_slave_tx_16_init` (C function), 26  
`i2s_tdm_slave_tx_16_thread` (C function), 27

### M

`master_requires_data_t` (C type), 14  
`master_sent_data_t` (C type), 14

### R

`read_reg` (C function), 15  
`read_reg16` (C function), 17  
`read_reg16_addr8` (C function), 16  
`read_reg8_addr16` (C function), 16

### S

`shutdown_t` (C type), 14  
`slave_transaction_ended_t` (C type), 29  
`slave_transaction_started_t` (C type), 29  
`SPI_CALLBACK_ATTR` (C macro), 31  
`spi_master_deinit` (C function), 31  
`spi_master_delay_before_next_transfer` (C function), 31  
`spi_master_device_init` (C function), 30  
`spi_master_device_t` (C struct), 32  
`spi_master_end_transaction` (C function), 31  
`spi_master_init` (C function), 29  
`spi_master_sample_delay_t` (C enum), 28  
`spi_master_sample_delay_t.spi_master_sample_delay_0` (C enumerator), 28  
`spi_master_sample_delay_t.spi_master_sample_delay_1` (C enumerator), 28  
`spi_master_sample_delay_t.spi_master_sample_delay_2` (C enumerator), 28  
`spi_master_sample_delay_t.spi_master_sample_delay_3` (C enumerator), 28  
`spi_master_sample_delay_t.spi_master_sample_delay_4` (C enumerator), 28



[spi\\_master\\_source\\_clock\\_t \(C enum\), 28](#)  
[spi\\_master\\_source\\_clock\\_t.spi\\_master\\_source\\_clock\\_ref \(C function\), 17](#)  
[spi\\_master\\_source\\_clock\\_t.spi\\_master\\_source\\_clock\\_ref \(C enumerator\), 28](#)  
[spi\\_master\\_source\\_clock\\_t.spi\\_master\\_source\\_clock\\_xcore \(C enumerator\), 29](#)  
[spi\\_master\\_start\\_transaction \(C function\), 30](#)  
[spi\\_master\\_t \(C struct\), 32](#)  
[spi\\_master\\_transfer \(C function\), 31](#)  
[SPI\\_MODE\\_0 \(C macro\), 31](#)  
[SPI\\_MODE\\_1 \(C macro\), 31](#)  
[SPI\\_MODE\\_2 \(C macro\), 31](#)  
[SPI\\_MODE\\_3 \(C macro\), 31](#)  
[spi\\_slave \(C function\), 32](#)  
[spi\\_slave\\_callback\\_group\\_t \(C struct\), 33](#)  
[stop\\_bit\\_t \(C type\), 14](#)

## T

[TDM\\_CALLBACK\\_ATTR \(C macro\), 22](#)  
[tdm\\_post\\_port\\_init\\_t \(C type\), 22](#)

## U

[uart\\_callback\\_code\\_t \(C enum\), 4](#)  
[uart\\_callback\\_code\\_t.UART\\_FRAMING\\_ERROR \(C enumerator\), 4](#)  
[uart\\_callback\\_code\\_t.UART\\_OVERRUN\\_ERROR \(C enumerator\), 4](#)  
[uart\\_callback\\_code\\_t.UART\\_PARITY\\_ERROR \(C enumerator\), 4](#)  
[uart\\_callback\\_code\\_t.UART\\_RX\\_COMPLETE \(C enumerator\), 4](#)  
[uart\\_callback\\_code\\_t.UART\\_START\\_BIT\\_ERROR \(C enumerator\), 4](#)  
[uart\\_callback\\_code\\_t.UART\\_UNDERRUN\\_ERROR \(C enumerator\), 4](#)  
[uart\\_parity \(C enum\), 3](#)  
[uart\\_parity.UART\\_PARITY\\_EVEN \(C enumerator\), 3](#)  
[uart\\_parity.UART\\_PARITY\\_NONE \(C enumerator\), 3](#)  
[uart\\_parity.UART\\_PARITY\\_ODD \(C enumerator\), 3](#)  
[uart\\_parity\\_t \(C type\), 4](#)  
[uart\\_rx \(C function\), 9](#)  
[uart\\_rx\\_blocking\\_init \(C function\), 8](#)  
[uart\\_rx\\_deinit \(C function\), 9](#)  
[uart\\_rx\\_init \(C function\), 8](#)  
[uart\\_rx\\_t \(C struct\), 9](#)  
[UART\\_START\\_BIT\\_ERROR\\_VAL \(C macro\), 5](#)  
[uart\\_state\\_t \(C enum\), 4](#)  
[uart\\_state\\_t.UART\\_DATA \(C enumerator\), 4](#)  
[uart\\_state\\_t.UART\\_IDLE \(C enumerator\), 4](#)  
[uart\\_state\\_t.UART\\_PARITY \(C enumerator\), 4](#)  
[uart\\_state\\_t.UART\\_START \(C enumerator\), 4](#)  
[uart\\_state\\_t.UART\\_STOP \(C enumerator\), 4](#)  
[uart\\_tx \(C function\), 5](#)  
[uart\\_tx\\_blocking\\_init \(C function\), 5](#)  
[uart\\_tx\\_deinit \(C function\), 5](#)  
[uart\\_tx\\_init \(C function\), 4](#)  
[uart\\_tx\\_t \(C struct\), 5](#)

## W

[write\\_reg16 \(C function\), 18](#)  
[write\\_reg16\\_addr8 \(C function\), 18](#)  
[write\\_reg16\\_addr16 \(C function\), 17](#)



Copyright © 2023, All Rights Reserved.

---

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

