fwk_io - Programming Guide

Release: 1.0.0

Publication Date: 2023/03/20

XMOS

Table of Contents

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^2C - master and slave
- I^2S - master and slave
- SPI - master and slave
- QSPI - IO and flash with Serial Flash Discoverable Parameter (SFDP) support

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};
```

(continues on next page)

(continued from previous page)

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

// 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_t`

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_t* `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:
```

(continues on next page)

(continued from previous page)

```
            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
    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;
```

(continues on next page)

(continued from previous page)

```
            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,
                 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 I²C Library

A software defined I²C library that allows you to control an I²C bus via xcore ports. I²C 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 I²C master or slave.

The library is compatible with multiple slave devices existing on the same bus. The I²C 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 I²C physical interfaces on a single xcore device simultaneously.

All signals are designed to comply with the timings in the I²C specification. Information on obtaining the I²C specification can be found here:

https://www.i2c-bus.org/specification/

Note that the following optional parts of the I²C 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

I²C 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 I²C specification.

All I²C functions can be accessed via the `i2c.h` header:

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

2.2.1 I²C Master

I²C Master Usage

The following code snippet demonstrates the basic usage of an I²C 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) ;
```

I²C Master API

The following structures and functions are used to initialize and start an I²C 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 I²C Slave

I²C Slave Usage

The following code snippet demonstrates the basic usage of an I²C slave device.

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

port_t p_scl = XS1_PORT_1A;
port_t p_sda = XS1_PORT_1B;

// Setup callbacks
```

(continues on next page)

(continued from previous page)

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

I²C Slave API

The following structures and functions are used to initialize and start an I²C 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 I²C Registers

I²C Register API

The following structures and functions are used to read and write I²C 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 I²S Library

A software defined library that allows you to control an I²S (Inter-IC Sound) bus via xcore ports. I²S is a digital data streaming interfaces particularly appropriate for transmission of audio data. The components in the library are controlled via C and can either act as I²S master or I²S slave.

Note: The TDM protocol is not yet supported by this library.

I²S is a protocol between two devices where one is the *master* and one is the *slave* . The protocol is made up of four signals shown in *I2S data wires*.

Table 2.2: I²S data wires

MCLK	Clock line, driven by external oscillator
BCLK	Bit clock. This is a fixed divide of the *MCLK* and is driven by the master.
LRCLK (or *WCLK*)	Word clock (or word select). This is driven by the master.
DATA	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 I²S functions can be accessed via the `i2s.h` header:

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

2.3.1 I²S Common API

The following structures and functions are used by an I²S 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.

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.

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_DATALINES`

`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.

2.3.2 I²S Master

I²S Master Usage

The following code snippet demonstrates the basic usage of an I²S 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;
```

(continues on next page)

(continued from previous page)

```
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);
```

I²S Master API

The following structures and functions are used to initialize and start an I²S 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 I²S Slave

I²S Slave Usage

The following code snippet demonstrates the basic usage of an I²S 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
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,
};
```

(continues on next page)

(continued from previous page)

```
// 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);
```

I²S Slave API

The following structures and functions are used to initialize and start an I²S 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

2.4 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 2.3: SPI data wires

SCLK	Clock line, driven by the master
MOSI	Master Output, Slave Input data line, driven by the master
MISO	Master Input, Slave Output data line, driven by the slave
SS	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;

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 * 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,
```

(continues on next page)

(continued from previous page)

```
    .app_data = NULL
};

port_t p_miso = XS1_PORT_1A;
port_t p_cs   = XS1_PORT_1B;
port_t p_sclk = XS1_PORT_1C;
port_t p_mosi = XS1_PORT_1D;
xclock_t cb   = XS1_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.

2.5 QSPI Library

A software defined QSPI (quad serial peripheral interface) library that allows you to read and write to a QSPI peripheral via the xcore ports.

All QSPI functions can be accessed via the `qspi_flash.h` or `qspi_io.h` header:

```
#include <qspi_flash.h>
#include <qspi_io.h>
```

2.5.1 QSPI Flash

QSPI Flash Usage

The following code snippet demonstrates the basic usage of an QSPI flash device.

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

qspi_flash_ctx_t qspi_flash_ctx;
qspi_io_ctx_t *qspi_io_ctx = &qspi_flash_ctx.qspi_io_ctx;

uint8_t data[4];

// Setup the flash device
qspi_flash_ctx.custom_clock_setup = 1;
qspi_flash_ctx.quad_page_program_cmd = qspi_flash_page_program_1_4_4;
qspi_flash_ctx.source_clock = qspi_io_source_clock_xcore;

qspi_io_ctx->clock_block = FLASH_CLKBLK;
qspi_io_ctx->cs_port = PORT_SQI_CS;
qspi_io_ctx->sclk_port = PORT_SQI_SCLK;
qspi_io_ctx->sio_port = PORT_SQI_SIO;

// Full speed clock configuration
qspi_io_ctx->full_speed_clk_divisor = 5; // 600 MHz / (2*5) -> 60 MHz
qspi_io_ctx->full_speed_sclk_sample_delay = 1;
qspi_io_ctx->full_speed_sclk_sample_edge = qspi_io_sample_edge_rising;

// SPI read clock configuration
qspi_io_ctx->spi_read_clk_divisor = 12;  // 600 MHz / (2*12) -> 25 MHz

qspi_io_ctx->spi_read_sclk_sample_delay = 0;
qspi_io_ctx->spi_read_sclk_sample_edge = qspi_io_sample_edge_falling;
qspi_io_ctx->full_speed_sio_pad_delay = 0;
qspi_io_ctx->spi_read_sio_pad_delay = 0;

// Initialize the flash device
qspi_flash_init(&qspi_flash_ctx);

// Read some data
qspi_flash_read(&qspi_flash_ctx, *data, 0x64, 4);
```

QSPI Flash API

The following structures and functions are used to QSPI flash I/O.

enum `qspi_flash_page_program_cmd_t`

Values:

enumerator `qspi_flash_page_program_1_1_1`

Programs pages using only MOSI. Most, if not all, QSPI flashes support this command. Use this if in doubt.

enumerator `qspi_flash_page_program_1_1_4`

Programs pages by sending the command and address over just SIO0, but the data over all four data lines. Many QSPI flashes support either this or qspi_flash_page_program_1_4_4, but not both. Check the datasheet. If the particular flash chip that will be used is unknown, then qspi_flash_page_program_1_1_1 should be used.

enumerator `qspi_flash_page_program_1_4_4`

Programs pages by sending the command over just SIO0, but the address and data over all four data lines. Many QSPI flashes support either this or qspi_flash_page_program_1_1_4, but not both. Check the datasheet. If the particular flash chip that will be used is unknown, then qspi_flash_page_program_1_1_1 should be used.

enum `qspi_flash_erase_length_t`

Most QSPI flashes allow data to be erased in 4k, 32k or 64k chunks, as well as the entire chip. However, these values are not always available on all chips. The erase info table in the qspi flash context structure defines the size of each erasable sector size. This is typically populated automatically by *qspi_flash_init()* from the SFDP data in the flash.

Values:

enumerator `qspi_flash_erase_1`

Erase the first available sector size. This should always be available and will be the smallest available erasable sector size.

enumerator `qspi_flash_erase_2`

Erase the second available sector size. This should be smaller than qspi_flash_erase_3 if both are available.

enumerator `qspi_flash_erase_3`

Erase the third available sector size. This should be smaller than qspi_flash_erase_4 if both are available.

enumerator `qspi_flash_erase_4`

Erase the fourth available sector size. This is typically not available, but will be the largest available erasable sector size if it is.

enumerator `qspi_flash_erase_chip`

Erase the entire chip

inline size_t `qspi_flash_erase_type_size`(*qspi_flash_ctx_t* *ctx, *qspi_flash_erase_length_t* erase_type)

inline uint32_t `qspi_flash_erase_type_size_log2`(*qspi_flash_ctx_t* *ctx, *qspi_flash_erase_length_t* erase_type)

bool `qspi_flash_quad_enable_write`(*qspi_flash_ctx_t* *ctx, bool set)

Sets or clears the quad enable bit in the flash.

Note: The quad enable bit is fixed to '1' in some QSPI flash chips, and cannot be cleared.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `set` – When true, the quad enable bit is set. When false, the quad enable bit is cleared if possible.

Return values

- `true` – if the QE bit was already at the requested value, or if the write was successful.
- `false` – if the write did not complete successfully. This can happen when trying to clear the QE bit on parts where it is fixed to '1'.

void `qspi_flash_write_enable`(*qspi_flash_ctx_t* *ctx)

Sets the write enable latch in the QSPI flash. This must be called prior to erasing, programming, or writing to a register.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.

void `qspi_flash_write_disable`(*qspi_flash_ctx_t* *ctx)

This clears the write enable latch in the QSPI flash. This is cleared automatically at the end of write operations, so normally does not need to be called.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.

bool `qspi_flash_write_in_progress`(*qspi_flash_ctx_t* *ctx)

This checks to see if the QSPI flash has a write operation in progress.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.

Return values

- `true` – if there is a flash write in progress.
- `false` – if the flash is not writing and is ready to accept another read or write command.

void `qspi_flash_wait_while_write_in_progress`(*qspi_flash_ctx_t* *ctx)

This waits while the QSPI flash has a write operation in progress. It returns when the write operation is complete, or immediately if there is not one in progress to begin with.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.

void `qspi_flash_erase`(*qspi_flash_ctx_t* *ctx, uint32_t address, *qspi_flash_erase_length_t* erase_length)

This performs an erase operation. *qspi_flash_write_enable()* must be called prior to this.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `address` – Any byte address within the data block to erase.

- `erase_length` – The data block size to erase. See qspi_flash_erase_length_t. If qspi_flash_erase_chip, then `address` is ignored.

void `qspi_flash_write_register`(*qspi_flash_ctx_t* *ctx, uint32_t cmd, const uint8_t *val, size_t len)

This writes to a register in the QSPI flash. This allows an application to write to non-standard registers specific to its flash chip.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `cmd` – The command required for writing to the desired register. Must be the value returned by *QSPI_IO_BYTE_TO_MOSI()*.
- `val` – Pointer to the data to write to the register.
- `len` – The number of bytes from `val` to write to the register.

void `qspi_flash_write_status_register`(*qspi_flash_ctx_t* *ctx, const uint8_t *val, size_t len)

This writes to the status register in the QSPI flash. *qspi_flash_write_enable()* must be called prior to this.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `val` – Pointer to the data to write to the register.
- `len` – The number of bytes from `val` to write to the register.

void `qspi_flash_read_register`(*qspi_flash_ctx_t* *ctx, uint32_t cmd, uint8_t *val, size_t len)

This reads from a register in the QSPI flash. This allows an application to read from non-standard registers specific to its flash chip.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `cmd` – The command required for reading from the desired register. Must be the value returned by *QSPI_IO_BYTE_TO_MOSI()*.
- `val` – Pointer to the buffer to store the data read from the register.
- `len` – The number of bytes to read from the register and save to `val`.

void `qspi_flash_read_status_register`(*qspi_flash_ctx_t* *ctx, uint8_t *val, size_t len)

This reads from the status register in the QSPI flash.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `val` – Pointer to the buffer to store the data read from the register.
- `len` – The number of bytes to read from the register and save to `val`.

void `qspi_flash_read_id`(*qspi_flash_ctx_t* *ctx, uint8_t *val, size_t len)

This reads the JEDEC ID of the QSPI flash.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `val` – Pointer to the buffer to write the ID to.
- `len` – The number of ID bytes to read and save to `val`.

void `qspi_flash_poll_register`(*qspi_flash_ctx_t* *ctx, uint32_t cmd, const uint8_t mask, const uint8_t val)

This polls a register in the QSPI flash. This allows an application to poll non-standard registers specific to its flash chip. The register must be one byte and repeatedly sent out over MISO following the read register command.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `cmd` – The command required for reading from the desired register. Must be the value returned by *QSPI_IO_BYTE_TO_MOSI()*.
- `mask` – The bitmask to apply to the received register value before comparing it to `val`;
- `val` – The value that the register value, masked with `mask`, must match before returning.

void `qspi_flash_poll_status_register`(*qspi_flash_ctx_t* *ctx, const uint8_t mask, const uint8_t val)

This polls the status register in the QSPI flash.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `mask` – The bitmask to apply to the received register value before comparing it to `val`;
- `val` – The value that the register value, masked with `mask`, must match before returning.

void `qspi_flash_fast_read`(*qspi_flash_ctx_t* *ctx, uint8_t *data, uint32_t address, size_t len)

This reads data from the flash in fast mode. This is a normal SPI read, using only SIO0 (MOSI) and SIO1 (MOSI), but includes eight dummy cycles between the address and data.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. The byte in bits 31:24 is not sent.
- `len` – The number of bytes to read and save to `data`.

void `qspi_flash_read`(*qspi_flash_ctx_t* *ctx, uint8_t *data, uint32_t address, size_t len)

This reads data from the flash in quad I/O mode. All four lines are used to send the address and to read the data.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are actually transmitted to the flash during the first two dummy cycles following the three address bytes. Some flashes read the SIO lines during these first two dummy cycles to enable certain features, so this might be useful for some applications.
- `len` – The number of bytes to read and save to `data`.

void `qspi_flash_read_nibble_swapped`(*qspi_flash_ctx_t* *ctx, uint8_t *data, uint32_t address, size_t len)

This is the same as *qspi_flash_read()* except that the nibbles in each byte of the data returned are swapped.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.

- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are actually transmitted to the flash during the first two dummy cycles following the three address bytes. Some flashes read the SIO lines during these first two dummy cycles to enable certain features, so this might be useful for some applications.
- `len` – The number of bytes to read and save to `data`.

void `qspi_flash_xip_read`(*qspi_flash_ctx_t* *ctx, uint8_t *data, uint32_t address, size_t len)

This reads data from the flash in quad I/O "eXecute In Place" mode. All four lines are used to send the address and to read the data. No command is sent. The flash must already have been put into "XIP" mode.

The method used to put the flash into XIP mode, as well as to take it out, is flash dependent. See your flash's datasheet.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are actually transmitted to the flash during the first two dummy cycles following the three address bytes. Some flashes read the SIO lines during these first two dummy cycles to enable certain features, so this might be useful for some applications.
- `len` – The number of bytes to read and save to `data`.

void `qspi_flash_xip_read_nibble_swapped`(*qspi_flash_ctx_t* *ctx, uint8_t *data, uint32_t address, size_t len)

This is the same as *qspi_flash_xip_read()* except that the nibbles in each byte of the data returned are swapped.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are actually transmitted to the flash during the first two dummy cycles following the three address bytes. Some flashes read the SIO lines during these first two dummy cycles to enable certain features, so this might be useful for some applications.
- `len` – The number of bytes to read and save to `data`.

void `qspi_flash_write`(*qspi_flash_ctx_t* *ctx, const uint8_t *data, uint32_t address, size_t len)

This writes data to a page in the QSPI flash. If ctx->quad_page_program_enable is true, then the command in ctx->quad_page_program_cmd is sent and all four SIO lines are used to send the address and data. Otherwise, the standard page program command is sent and only SIO0 (MOSI) is used to send the address and data.

qspi_flash_write_enable() must be called prior to this.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the data to write to the flash.
- `address` – The byte address in the flash to begin writing at. Only bits 23:0 contain the address. The byte in bits 31:24 is not sent.
- `len` – The number of bytes to write to the flash.

void `qspi_flash_write_nibble_swapped`(*qspi_flash_ctx_t* *ctx, const uint8_t *data, uint32_t address, size_t len)

This is the same as *qspi_flash_write()* except that the nibbles in each byte of the data written are swapped.

qspi_flash_write_enable() must be called prior to this.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the data to write to the flash.
- `address` – The byte address in the flash to begin writing at. Only bits 23:0 contain the address. The byte in bits 31:0 is not sent.
- `len` – The number of bytes to write to the flash.

void `qspi_flash_sfdp_read`(*qspi_flash_ctx_t* *ctx, uint8_t *data, uint32_t address, size_t len)

This reads the SFDP data from the flash in 1-1-1 mode.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the SFDP area to begin reading at. Only bits 23:0 contain the address. The byte in bits 31:24 is not sent.
- `len` – The number of bytes to read and save to `data`.

void `qspi_flash_deinit`(*qspi_flash_ctx_t* *ctx)

Deinitializes the QSPI I/O interface associated with the QSPI flash.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash. This should have been previously initialized with *qspi_flash_init()*.

void `qspi_flash_init`(*qspi_flash_ctx_t* *ctx)

Initializes the QSPI I/O interface associated with the QSPI flash. The ports and clock block in the ctx->qspi_io_ctx must be set prior to calling this.

If ctx->custom_clock_setup is false, then the QSPI I/O clock configuration is set to safe default values that should work for all QSPI flashes. Otherwise, the clock configuration values must be supplied by the application.

Parameters

- `ctx` – The QSPI flash context associated with the QSPI flash.

`QSPI_FLASH_SANITY_CHECKS`

When QSPI_FLASH_SANITY_CHECKS is true then some run-time sanity checks are made at the expense of some extra overhead.

`QSPI_FLASH_STATUS_REG_WIP_BM`

The bit mask for the status register's write in progress bit.

`QSPI_FLASH_STATUS_REG_WEL_BM`

The bit mask for the status register's write enable latch bit.

struct `qspi_flash_ctx_t`

#include <qspi_flash.h> The context structure that must be passed to each of the qspi_flash functions.

2.5.2 QSPI I/O

QSPI I/O API

The following structures and functions are used for low-level QSPI I/O.

enum `qspi_io_sample_edge_t`

Enum type used to set which SCLK edge SIO is sampled on.

Values:

enumerator `qspi_io_sample_edge_rising`

Sample SIO on the rising edge

enumerator `qspi_io_sample_edge_falling`

Sample SIO on the falling edge

enum `qspi_io_source_clock_t`

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

Values:

enumerator `qspi_io_source_clock_ref`

SCLK is derived from the 100 MHz reference clock

enumerator `qspi_io_source_clock_xcore`

SCLK is derived from the core clock

enum `qspi_io_transfer_mode_t`

Enum type used to specify whether or not nibbles should be swapped during transfers.

Values:

enumerator `qspi_io_transfer_normal`

Do not swap nibbles

enumerator `qspi_io_transfer_nibble_swap`

Swap nibbles

enum `qspi_io_transaction_type_t`

Enum type used to specify whether the next transaction will be a full speed QSPI transaction with dummy cycles, or a lower speed SPI read transaction without dummy cycles.

Values:

enumerator `qspi_io_full_speed`

The transaction will be full speed with dummy cycles

enumerator `qspi_io_spi_read`

The transaction will be low speed without dummy cycles

inline uint32_t `qspi_io_byte_to_mosi`(uint32_t x)

This function should only be called internally by *qspi_io_mosi_out()*. It performs the same transformation as *QSPI_IO_BYTE_TO_MOSI()* but also integrates the byte reversal and nibble swap performed by *qspi_io_words_out()*.

Parameters

- `x` – The byte to send out to MOSI.

Returns

the word to output to SIO.

inline uint32_t `qspi_io_miso_to_byte`(uint32_t x)

This function should only be called internally by *qspi_io_miso_in()*.

When reading in a single byte in SPI mode, the word that is received on SIO needs to be transformed such that bit one from each nibble (which corresponds to SIO1, or MISO) is shifted into the correct bit position.

Parameters

- `x` – The word received on SIO.

Returns

the byte received on MISO.

inline uint32_t `qspi_io_nibble_swap`(uint32_t word)

This function swaps the nibbles in each of the four bytes of `word`.

Parameters

- `word` – The word to nibble swap.

Returns

the nibble swapped word.

inline void `qspi_io_start_transaction`(*qspi_io_ctx_t* *ctx, uint32_t first_word, size_t len, *qspi_io_transaction_type_t* transaction_type)

Begins a new QSPI I/O transaction by starting the clock, asserting CS, and sending out the first word which is typically a command.

Note: If more words or bytes need to be sent or received as part of this transaction, then the appropriate functions will need to be called immediately following this one. For example, *qspi_io_bytes_out()* then *qspi_io_sio_direction_input()* then *qspi_io_bytes_in()*. The "out" or "in" instruction in each must be executed within eight SCLK cycles of the preceding one, including the OUT instruction in *qspi_io_start_transaction()*. Some analysis may be necessary depending on the frequency of SCLK. These functions are all marked as inline to help keep them closer together by removing the overhead associated with function calls and to allow better optimization.

Note: It is likely not possible to follow an input with an output within a single transaction unless the frequency of SCLK is sufficiently slow. Fortunately in practice this rarely, if ever, required.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `first_word` – The first word to output.
- `len` – The total number of clock cycles in the transaction. CS will at some point during the transaction be setup to deassert automatically after this number of cycles.
- `transaction_type` – Set to qspi_io_spi_read if the transaction will be a SPI read with no dummy cycles. This may run at a slower clock frequency in order to turn around SIO from output to input in time. Otherwise set to qspi_io_full_speed.

inline void `qspi_io_bytes_out`(const *qspi_io_ctx_t* *ctx, const *qspi_io_transfer_mode_t* transfer_mode, const uint8_t *data, size_t len)

Outputs a byte array to the QSPI interface. The byte array must start on a 4-byte boundary. This call must be made in time such that its OUT instruction executes within 8 SCLK cycles of the previous OUT instruction.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `transfer_mode` – Can be either qspi_io_transfer_normal or qspi_io_transfer_nibble_swap. When qspi_io_transfer_nibble_swap, each byte transferred out is nibble swapped. Because the data is inherently sent out nibble swapped over the port, setting this to qspi_io_transfer_nibble_swap actually removes a nibble swap operation.
- `data` – Pointer to the byte array to output. This MUST begin on a 4-byte boundary.
- `len` – The number of bytes in `data` to output.

inline void `qspi_io_words_out`(const *qspi_io_ctx_t* *ctx, const *qspi_io_transfer_mode_t* transfer_mode, const uint32_t *data, size_t len)

Outputs a word array to the QSPI interface. This call must be made in time such that its OUT instruction executes within 8 SCLK cycles of the previous OUT instruction.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `transfer_mode` – Can be either qspi_io_transfer_normal or qspi_io_transfer_nibble_swap. When qspi_io_transfer_nibble_swap, each byte transferred out is nibble swapped. Because the data is inherently sent out nibble swapped over the port, setting this to qspi_io_transfer_nibble_swap actually removes a nibble swap operation.
- `data` – Pointer to the word array to output.
- `len` – The number of words in `data` to output.

inline void `qspi_io_mosi_out`(const *qspi_io_ctx_t* *ctx, const *qspi_io_transfer_mode_t* transfer_mode, const uint8_t *data, size_t len)

Outputs a byte array to the QSPI interface over the single data line MOSI (SIO0). This call must be made in time such that its OUT instruction executes within 8 SCLK cycles of the previous OUT instruction.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `transfer_mode` – Can be either qspi_io_transfer_normal or qspi_io_transfer_nibble_swap. When qspi_io_transfer_nibble_swap, each byte transferred out is nibble swapped.

- `data` – Pointer to the byte array to output.
- `len` – The number of words in `data` to output.

inline void `qspi_io_sio_direction_input`(*qspi_io_ctx_t* *ctx)

This must be called to change the direction of SIO from output to input before calling either *qspi_io_bytes_in()* or *qspi_io_words_in()*. This call must be made in time such that the call to port_set_buffered() completes before the sample edge of SCLK shifts in the first nibble of the next data word to be read. This also will setup CS to deassert at the end of the transaction while waiting for the previous output to complete.

Note: This is probably the most fragile function. Ensure that the port direction is turned around on time, and that the subsequent read IN instruction executes on time, by inspecting a VCD trace with both ports and instructions.

Parameters

- `ctx` – Pointer to the QSPI I/O context.

inline void `qspi_io_bytes_in`(const *qspi_io_ctx_t* *ctx, const *qspi_io_transfer_mode_t* transfer_mode, uint8_t *data, uint32_t start_time, size_t len)

Inputs a byte array from the QSPI interface. The byte array must start on a 4-byte boundary. *qspi_io_sio_direction_input()* must have been called previously. This call must be made in time such that its IN instruction executes before `start_time`.

Note: The number of bytes input may be any number. However, if it is NOT a multiple of four, then this likely will need to be the last call in the transaction. This is because the shorter length of the last input chunk plus the extra overhead required to deal with the sub-word accesses will not allow subsequent I/O to keep up unless the SCLK frequency is significantly slower than the core clock.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `transfer_mode` – Can be either qspi_io_transfer_normal or qspi_io_transfer_nibble_swap. When qspi_io_transfer_nibble_swap, each byte transferred in is nibble swapped. Because the data is inherently received nibble swapped over the port, setting this to qspi_io_transfer_nibble_swap actually removes a nibble swap operation.
- `data` – Pointer to the byte array to save the received data to. This MUST begin on a 4-byte boundary.
- `start_time` – The port time, relative to the beginning of the transfer, at which to input the first group of four bytes. This must line up with the last nibble of the fourth byte. If `len` is less than four, then it must line up with the last nibble of the last byte.
- `len` – The number of bytes to input.

inline void `qspi_io_words_in`(const *qspi_io_ctx_t* *ctx, const *qspi_io_transfer_mode_t* transfer_mode, uint32_t *data, uint32_t start_time, size_t len)

Inputs a word array from the QSPI interface. *qspi_io_sio_direction_input()* must have been called previously. This call must be made in time such that its IN instruction executes before `start_time`.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `transfer_mode` – Can be either qspi_io_transfer_normal or qspi_io_transfer_nibble_swap. When qspi_io_transfer_nibble_swap, each byte transferred in is nibble swapped. Because the data is inherently received nibble swapped over the port, setting this to qspi_io_transfer_nibble_swap actually removes a nibble swap operation.
- `data` – Pointer to the word array to save the received data to.
- `start_time` – The time, relative to the beginning of the transfer, at which to input the first word. This must line up with the last nibble of the first word.
- `len` – The number of words to input.

inline void `qspi_io_miso_in`(const *qspi_io_ctx_t* *ctx, const *qspi_io_transfer_mode_t* transfer_mode, uint8_t *data, uint32_t start_time, size_t len)

Inputs a byte array from the QSPI interface over the single data line MISO (SIO1). *qspi_io_sio_direction_input()* must have been called previously. This call must be made in time such that its IN instruction executes before `start_time`.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `transfer_mode` – Can be either qspi_io_transfer_normal or qspi_io_transfer_nibble_swap. When qspi_io_transfer_nibble_swap, each byte transferred in is nibble swapped.
- `data` – Pointer to the byte array to save the received data to.
- `start_time` – The time, relative to the beginning of the transfer, at which to input the first byte. This must line up with the last bit of the first byte.
- `len` – The number of words to input.

inline void `qspi_io_miso_poll`(const *qspi_io_ctx_t* *ctx, const uint8_t mask, const uint8_t val, uint32_t start_time)

Polls the SPI interface by repeatedly receiving a byte over MISO until a specified condition is met. For each time the received byte does not meet the condition, the deassertion of CS is extended by eight SCLK cycles. *qspi_io_sio_direction_input()* must have been called previously. This call must be made in time such that its IN instruction executes before `start_time`.

Parameters

- `ctx` – Pointer to the QSPI I/O context.
- `mask` – The bitmask to apply to the received byte before comparing it to `val`;
- `val` – The value that the received byte, masked with `mask`, must match before returning.
- `start_time` – The time, relative to the beginning of the transfer, at which to input the first byte. This must line up with the last bit of the first byte.

inline void `qspi_io_end_transaction`(const *qspi_io_ctx_t* *ctx)

This sets up CS to deassert at the end of the transaction if it has not already, waits for the current QSPI transaction to complete, and then stops SCLK.

Parameters

- `ctx` – Pointer to the QSPI I/O context.

void `qspi_io_deinit`(const *qspi_io_ctx_t* *ctx)

This disables and frees the clock block and all the ports associated with the QSPI I/O interface.

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

Parameters

- `ctx` – Pointer to the QSPI I/O context. This should have been previously initialized with *qspi_io_init()*.

void `qspi_io_init`(const *qspi_io_ctx_t* *ctx, *qspi_io_source_clock_t* source_clock)

This sets up the clock block and all the ports associated with the QSPI I/O interface. This must be called first prior to any other QSPI I/O function.

Parameters

- `ctx` – Pointer to the QSPI I/O context. This must be initialized with the clock block and ports to use.
- `source_clock` – Set to qspi_io_source_clock_ref to use the 100 MHz reference clock as the source for SCLK. Set to qspi_io_source_clock_xcore to use the xcore clock.

`QSPI_IO_BYTE_TO_MOSI`(x)

This macro may be used when sending out bytes that are only transmitted over the single data line MOSI (SIO0). The returned word should be transmitted using either *qspi_io_start_transaction()* or *qspi_io_words_out()*. Typically the byte argument to this macro is a constant known at compile time, like commands, as the compiler can perform this computation at compile time. For arrays of data, it may be more appropriate to use *qspi_io_mosi_out()* which more efficiently computes this transformation at run time on the fly.

When writing a single byte out in SPI mode, the byte needs to be transformed such that each nibble in the word that is sent out on SIO contains one bit from the byte in bit 0 (which corresponds to SIO0, or MOSI).

Parameters

- `x` – The byte to send out to MOSI.

`QSPI_IO_SETC_PAD_DELAY`(n)

`QSPI_IO_RESOURCE_SETCI`(res, c)

`QSPI_IO_RESOURCE_SETC`(res, r)

struct `qspi_io_ctx_t`

#include <qspi_io.h> The context structure that must be passed to each of the qspi_io functions. Several of the members in this structure must be set by the application prior to calling either *qspi_io_init()* or *qspi_io_start_transaction()*.

3 Copyright & Disclaimer

4 Licenses

4.1 XMOS

All original source code is licensed under the XMOS License.

4 Index

R

S

U

W