I2C Library

A software defined, industry-standard, I^{2}C library that allows you to control an I^{2}C bus via xCORE ports. I^{2}C is a two-wire hardware serial interface, first developed by Philips. The components in the libary are controlled via C using the XMOS multicore extensions (xC) and can either act as I^{2}C master or slave.

The libary is compatible with multiple slave devices existing on the same bus. The I^{2}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^{2}C physical interfaces on a single xCORE device simultaneously.

Features

- I^{2}C master and I^{2}C slave modes.
- Supports speed up to 400 Kb/s (I^{2}C Fast-mode).
- Clock stretching support.
- Synchronous and asynchronous APIs for efficient usage of processing cores.

Typical Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
Master	2	2 (1-bit)	0	~1.4K	0
Master (single port)	2	1 (multi-bit)	0	~1.5K	1
Master (asynchronous)	2	2 (1-bit)	0	~3.4K	1
Master (asynchronous, combinable)	2	2 (1-bit)	0	~3.3K	≤ 1
Slave	2	2 (1-bit)	0	~1.3K	≤ 1

Software version and dependencies

This document pertains to version 5.0.0 of this library. It is known to work on version 14.3.3 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_xassert (>=3.0.0)
- lib_logging (>=2.1.0)

Related application notes

The following application notes use this library:

- AN00156: How to use the I2C master library
- AN00157: How to use the I2C slave library
- AN00181: xCORE-200 explorer accelerometer demo

1 External signal description

All signals are designed to comply with the timings in the I^2C specification found here:

http://www.nxp.com/documents/user_manual/UM10204.pdf

Note that the following optional parts of the I^2C 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^2C 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^2C specification.

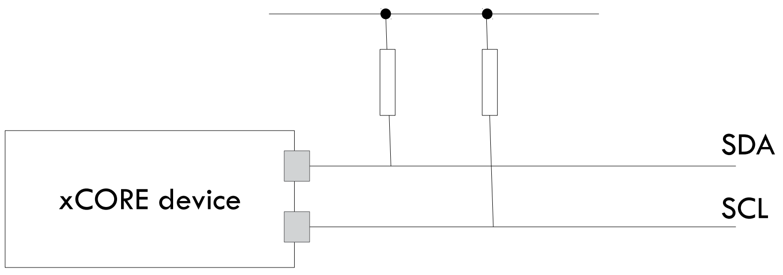

Figure 1: I^2C open-drain layout

Transactions on the line occur between a *master* and a *slave*. The master always drives the clock (though the slave can delay the transaction at any point by holding the clock line down). The master initiates a transaction with a start bit (consisting of driving the data line from high to low whilst the clock line is high). It will then clock out a seven-bit device address followed by a read/write bit. The master will then drive one more clock pulse during which the slave can either ACK (drive the line low), accepting the transaction or NACK (leave the line high). This sequence is shown in Figure 2.

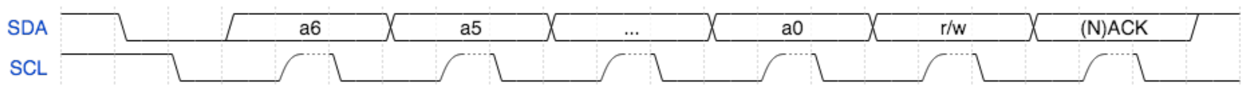

Figure 2: I^2C transaction start

If the read/write bit of the transaction start is 1 then the master will execute a sequence of reads. Each read consists of the master driving the clock whilst the slave drives the data for 8-bits (most significant bit first). At the end of each byte, the master drives another clock pulse and will either drive either an ACK (0) or NACK (1) signal on the data line. When the master drives a NACK signal, the sequence of reads is complete. A read byte sequence is show in Figure 3

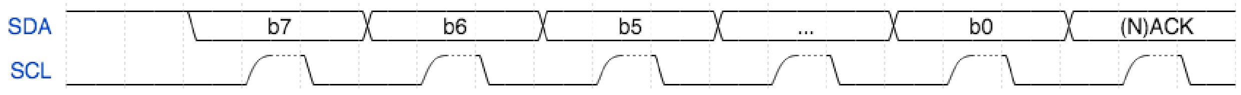

Figure 3: I^{2}C read byte

If the read/write bit of the transaction start is 0 then the master will execute a sequence of writes. Each write consists of the master driving the clock whilst and also driving the data for 8-bits (most significant bit first). At the end of each byte, the master drives another clock pulse and the slave will either drive either an ACK (0) (signalling that it can accept more data) or a NACK (1) (signalling that it cannot accept more data) on the data line. After the ACK/NACK signal, the master can complete the transaction with a stop bit or repeated start. A write byte sequence is show in Figure 4

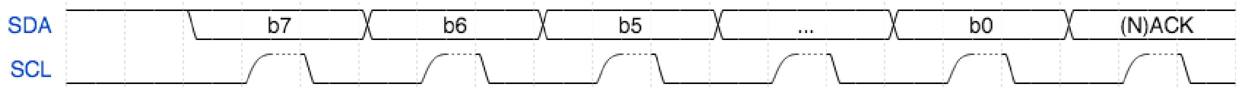

Figure 4: I^{2}C write byte

After a transaction is complete, the master may start a new transaction (a *repeated start*) or will send a stop bit consisting of releasing the data line so that it floats from low to high whilst the clock line is high (see Figure 5).

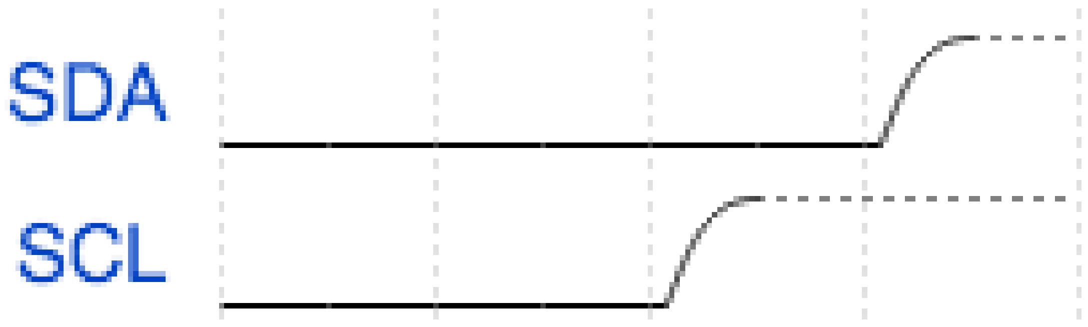

Figure 5: I^{2}C stop bit

1.1 Connecting to the xCORE device

When the xCORE is the I^{2}C master, the normal configuration is to connect the clock and data lines to different 1-bit ports as shown in Figure 6.

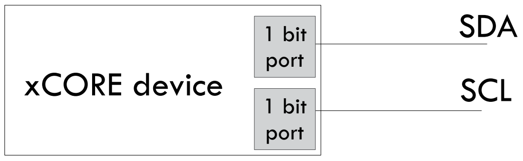

Figure 6: I^{2}C master (1-bit ports)

It is possible to connect both lines to different bits of a multi-bit port as shown in Figure 7. This is useful if other constraints limit the use of one bit ports. However the following should be taken into account:

- On L-series and U-series devices in this configuration, the xCORE can only perform write transactions to the I^{2}C bus.
- On L-series and U-series clock stretching is not supported in this configuration.
- The other bits of the multi-bit port cannot be used for any other function.

The restrictions on reading and clock stretching do not apply to xCORE-200 devices.

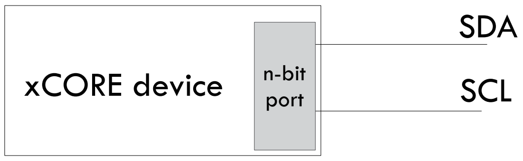

Figure 7: I^{2}C master (single n-bit port)

When the xCORE is acting as I^{2}C slave the two lines *must* be connected to two 1-bit ports (as shown in Figure 8).

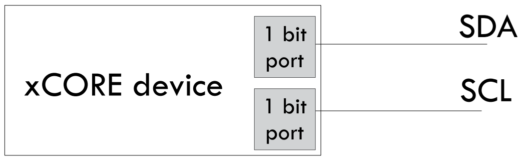

Figure 8: I^{2}C slave connection

2 Usage

2.1 I^{2}C master synchronous operation

There are two types of interface for I^{2}C masters: synchronous and asynchronous.

The synchronous API provides blocking operation. Whenever a client makes a read or write call the operation will complete before the client can move on - this will occupy the core that the client code is running on until the end of the operation. This method is easy to use, has low resource use and is very suitable for applications such as setup and configuration of attached peripherals.

I^{2}C masters are instantiated as parallel tasks that run in a par statement. For synchronous operation, the application can connect via an interface connection using the i2c_master_if interface type:

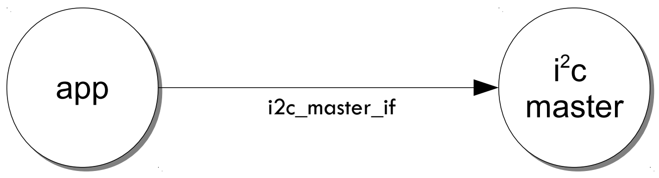

Figure 9: I^{2}C master task diagram

For example, the following code instantiates an I^{2}C master and connects to it

```
port p_scl = XS1_PORT_1E;
port p_sda = XS1_PORT_1F;

int main(void) {
  i2c_master_if i2c[1];
  static const uint8_t target_device_addr = 0x3c;

  par {
    i2c_master(i2c, 1, p_scl, p_sda, 100);
    my_application(i2c[0], target_device_addr);
  }
  return 0;
}
```

For the single multi-bit port version of I^{2}C the top level instantiation would look like

```
port p_i2c = XS1_PORT_4C;

int main(void) {
  i2c_master_if i2c[1];
  static const uint8_t target_device_addr = 0x3c;

  par {
    i2c_master_single_port(i2c, 1, p_i2c, 100, 1, 3, 0);
    my_application(i2c[0], target_device_addr);
  }
  return 0;
}
```

Note that the connection is an array of interfaces, so several tasks can connect to the same master.

The application can use the client end of the interface connection to perform I^{2}C bus operations e.g.

```
void my_application(client i2c_master_if i2c, uint8_t target_device_addr) {
    uint8_t data[2];
    i2c.read(target_device_addr, data, 2, 1);
    printf("Read data %d, %d from the bus.\n", data[0], data[1]);
}
```

Here the operations such as `i2c.read` will block until the operation is completed on the bus. More information on interfaces and tasks can be be found in the XMOS Programming Guide (see XM-004440-PC). By default the I^{2}C synchronous master mode component does not use any logical cores of its own. It is a *distributed* task which means it will perform its function on the logical core of the application task connected to it (provided the application task is on the same tile as the I^{2}C ports).

2.2 I^{2}C master asynchronous operation

The synchronous API will block your application until the bus operation is complete. In cases where the application cannot afford to wait for this long the asynchronous API can be used.

The asynchronous API offloads operations to another task. Calls are provided to initiate reads and writes. Notifications are provided when the operation completes. This API requires more management in the application but can provide much more efficient operation. It is particularly suitable for applications where the I^{2}C bus is being used for continuous data transfer.

Setting up an asynchronous I^{2}C master component is done in the same manner as the synchronous component.

```
port p_scl = XS1_PORT_1E;
port p_sda = XS1_PORT_1F;

#define BUFFER_BYTES 100

int main(void) {
  i2c_master_async_if i2c[1];
  static const uint8_t target_device_addr = 0x3c;

  par {
    i2c_master_async(i2c, 1, p_scl, p_sda, 100, BUFFER_BYTES);
    my_application(i2c[0], target_device_addr);
  }
  return 0;
}
```

The application can then use the asynchronous API to offload bus operations to the I^{2}C master. For example, the following code repeatedly calculates *BUFFER_BYTES* bytes to send over the bus.

```
void my_application(client i2c_master_async_if i2c, uint8_t target_device_addr) {
  uint8_t buffer[BUFFER_BYTES];

  // Create and send initial block of data
  my_application_fill_buffer(buffer);
  i2c.write(target_device_addr, buffer, BUFFER_BYTES, 1);

  // Start computing the next block of data
  my_application_fill_buffer(buffer);

  while (1) {
    select {
      case i2c.operation_complete():
        size_t num_bytes_sent;
        i2c_res_t result = i2c.get_write_result(num_bytes_sent);
        if (num_bytes_sent != BUFFER_BYTES) {
           my_application_handle_bus_error(result);
        }

        // Offload the next data bytes to be sent
        i2c.write(target_device_addr, buffer, BUFFER_BYTES, 1);

        // Compute the next block of data
        my_application_fill_buffer(buffer);

        break;
    }
  }
}
```

Here the calculation of `my_application_fill_buffer` will overlap with the sending of data by the other task.

2.3 Repeated start bits

The library supports repeated start bits. The `read` and `write` functions allow the application to specify whether to send a stop bit at the end of the transaction. If this is set to 0 then no stop bit is sent and the next transaction will begin with a repeated start bit e.g.

```
void my_application(client i2c_master_if i2c, uint8_t target_device_addr) {
  uint8_t data[2] = { 0x1, 0x2 };
  size_t num_bytes_sent = 0;

  // Do a write operation with no stop bit
  i2c.write(target_device_addr, data, 2, num_bytes_sent, 0);

  // This operation will begin with a repeated start bit
  i2c.read(target_device_addr, data, 2, 1);
  printf("Read data %d, %d from the bus.\n", data[0], data[1]);
}
```

Note that if no stop bit is sent then no other client using the same I^{2}C master can send or receive data. They will block until a stop bit is sent.

2.4 I^{2}C slave library usage

I^{2}C slaves are instantiated as parallel tasks that run in a par statement. The application can connect via an interface connection.

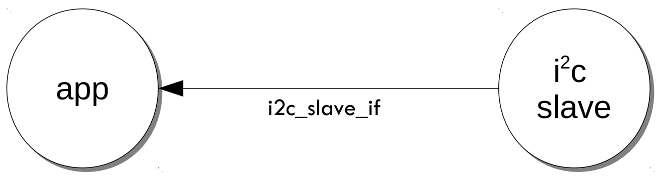

Figure 10: I^{2}C slave task diagram

For example, the following code instantiates an I^{2}C slave and connects to it.

```
port p_scl = XS1_PORT_1E;
port p_sda = XS1_PORT_1F;

int main(void) {
  static const uint8_t device_addr = 0x3c;
  i2c_slave_callback_if i2c;

  par {
    i2c_slave(i2c, p_scl, p_sda, device_addr);
    my_application(i2c);
  }

  return 0;
}
```

The slave acts as the client of the interface connection. This means it can “callback” to the application to respond to requests from the bus master. For example, the my_application function above needs to respond to the calls e.g.

```
void my_application(server i2c_slave_callback_if i2c) {
  while (1) {
    select {
    case i2c.ack_read_request() -> i2c_slave_ack_t response:
      response = I2C_SLAVE_ACK;
      break;
    case i2c.ack_write_request() -> i2c_slave_ack_t response:
      response = I2C_SLAVE_ACK;
      break;
    case i2c.master_sent_data(uint8_t data) -> i2c_slave_ack_t response:
      // handle write to device here, set response to NACK for the
      // last byte of data in the transaction.
      break;
    case i2c.master_requires_data() -> uint8_t data:
      // handle read from device here
      break;
    case i2c.stop_bit():
      break;
    }
  }
}
```

More information on interfaces and tasks can be be found in the XMOS Programming Guide (see XM-004440-PC).

3 Master API

All I^{2}C master functions can be accessed via the `i2c.h` header:

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

You will also have to add `lib_i2c` to the `USED_MODULES` field of your application Makefile.

3.1 Creating an I^{2}C master instance

Function	**i2c_master**
Description	Implements I2C on the i2c_master_if interface using two ports.

Type	`[[distributable]]`
	`void`
	`i2c_master(server interface i2c_master_if i[n],`
	`   size_t n,`
	`   port p_scl,`
	`   port p_sda,`
	`   static const unsigned kbits_per_second)`

Parameters	`i` an array of server interface connections for clients to connect to `n` the number of clients connected `p_scl` the SCL port of the I2C bus `p_sda` the SDA port of the I2C bus `kbits_per_second` the speed of the I2C bus

Function	**i2c_master_single_port**
Description	Implements I2C on a single multi-bit port. This function implements an I2C master bus using a single port. It is only supported on xCORE-200 devices.

Type	`[[distributable]]`
	`void`
	`i2c_master_single_port(server interface i2c_master_if c[n],`
	`   static const size_t n,`
	`   port p_i2c,`
	`   static const unsigned kbits_per_second,`
	`   static const unsigned scl_bit_position,`
	`   static const unsigned sda_bit_position,`
	`   static const unsigned other_bits_mask)`

Parameters	`c` an array of server interface connections for clients to connect to `n` the number of clients connected `p_i2c` the multi-bit port containing both SCL and SDA. the bit positions of SDA and SCL are configured using the `sda_bit_position` and `scl_bit_position` arguments. `kbits_per_second` the speed of the I2C bus `sda_bit_position` the bit of the SDA line on the port `scl_bit_position` the bit of the SCL line on the port `other_bits_mask` a value that is ORed into the port value driven to `p_i2c`. The SDA and SCL bit values for this variable must be set to 0. Note that `p_i2c` is configured with `set_port_drive_low()` and therefore external pullup resistors are required to produce a value 1 on a bit.

Function	**i2c_master_async**
Description	I2C master component (asynchronous API). This function implements I2C and allows clients to asynchronously perform operations on the bus.

Type	`void`
	`i2c_master_async(server interface i2c_master_async_if i[n],`
	`   size_t n,`
	`   port p_scl,`
	`   port p_sda,`
	`   static const unsigned kbits_per_second,`
	`   static const size_t max_transaction_size)`

Parameters	`i` the interfaces to connect the component to its clients `n` the number of clients connected to the component `p_scl` the SCL port of the I2C bus `p_sda` the SDA port of the I2C bus `kbits_per_second` the speed of the I2C bus `max_transaction_size` the size of the local buffer in bytes. Any transactions exceeding this size will cause a run-time exception.

3.2 I^{2}C master supporting typedefs

Type	**i2c_res_t**
Description	This type is used in I2C functions to report back on whether the slave performed an ACK or NACK on the last piece of data sent to it.
Values	`I2C_NACK` the slave has NACKed the last byte `I2C_ACK` the slave has ACKed the last byte

Type	**i2c_regop_res_t**
Description	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	`I2C_REGOP_SUCCESS` the operation was successful `I2C_REGOP_DEVICE_NACK` the operation was NACKed when sending the device address, so either the device is missing or busy `I2C_REGOP_INCOMPLETE` the operation was NACKed halfway through by the slave

3.3 I^{2}C master synchronous interface

Type	**i2c_master_if**
Description	This interface is used to communication with an I2C master component. It provides facilities for reading and writing to the bus.
Functions	

Function	**write**
Description	Write data to an I2C bus.

Type	`[[guarded]]`
	`i2c_res_t write(uint8_t device_addr,`
	`              uint8_t buf[n],`
	`              size_t n,`
	`              size_t &num_bytes_sent,`
	`              int send_stop_bit)`

Parameters	`device_addr` the address of the slave 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 slave device, otherwise `I2C_NACK`.

Continued on next page

Type	i2c_master_if (continued)

Function	read
Description	Read data from an I2C bus.

Type	`[[guarded]]`
	`i2c_res_t read(uint8_t device_addr,`
	`              uint8_t buf[n],`
	`              size_t n,`
	`              int send_stop_bit)`

Parameters	`device_addr` the address of the slave 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 slave device, otherwise I2C_NACK.

Function	send_stop_bit
Description	Send a stop bit. 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 `read()` or `write()` functions.
Type	`void send_stop_bit(void)`

Function	shutdown
Description	Shutdown the I2C component. This function will cause the I2C task to shutdown and return.
Type	`void shutdown()`

Continued on next page

Type	i2c_master_if (continued)
	Function: read_reg **Description:** Read an 8-bit register on a slave device. This function reads an 8-bit addressed, 8-bit register from the i2c bus. The function reads data by transmitting the register addr and then reading the data from the slave device. Note that no stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start. **Type:** `uint8_t read_reg(uint8_t device_addr, uint8_t reg, i2c_regop_res_t &result)` **Parameters:** `i` the interface to the I2C master `device_addr` the address of the slave device to read from `reg` the address of the register to read `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

Continued on next page

Type	i2c_master_if (continued)
	(nested table below)

Function	write_reg
Description	Write an 8-bit register on a slave device. This function writes an 8-bit addressed, 8-bit register from the i2c bus. The function writes data by transmitting the register addr and then transmitting the data to the slave device.
Type	`i2c_regop_res_t write_reg(uint8_t device_addr, uint8_t reg, uint8_t data)`
Parameters	`i` the interface to the I2C master `device_addr` the address of the slave device to write to `reg` the address of the register to write `data` the 8-bit value to write

Continued on next page

Type	i2c_master_if (continued)

Function	**read_reg8_addr16**
Description	Read an 8-bit register on a slave device from a 16-bit register address. This function reads a 16-bit addressed, 8-bit register from the i2c bus. The function reads data by transmitting the register addr and then reading the data from the slave device. Note that no stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

Type	`uint8_t`
	`read_reg8_addr16(uint8_t device_addr,`
	`                 uint16_t reg,`
	`                 i2c_regop_res_t &result)`

Parameters	`i` the interface to the I2C master `device_addr` the address of the slave device to read from `reg` the 16-bit address of the register to read (most significant byte first) `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

Continued on next page

Type	i2c_master_if (continued)

Function	write_reg8_addr16
Description	Write an 8-bit register on a slave device from a 16-bit register address. This function writes a 16-bit addressed, 8-bit register from the i2c bus. The function writes data by transmitting the register addr and then transmitting the data to the slave device.

Type	`i2c_regop_res_t`
	`write_reg8_addr16(uint8_t device_addr,`
	`                  uint16_t reg,`
	`                  uint8_t data)`

Parameters	`i` the interface to the I2C master `device_addr` the address of the slave device to write to `reg` the 16-bit address of the register to write (most significant byte first) `data` the 8-bit value to write

Continued on next page

Type	i2c_master_if (continued)

Function	read_reg16
Description	Read an 16-bit register on a slave device from a 16-bit register address. This function reads a 16-bit addressed, 16-bit register from the i2c bus. The function reads data by transmitting the register addr and then reading the data from the slave device. It is assumed the data is returned most significant byte first on the bus. Note that no stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.
Type	`uint16_t read_reg16(uint8_t device_addr, uint16_t reg, i2c_regop_res_t &result)`
Parameters	`i` the interface to the I2C master `device_addr` the address of the slave device to read from `reg` the address of the register to read (most significant byte first) `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 16-bit value of the register

Continued on next page

Type	i2c_master_if (continued)
	<table><tr><td>**Function**</td><td>**write_reg16**</td></tr><tr><td>**Description**</td><td>Write an 16-bit register on a slave device from a 16-bit register address. This function writes a 16-bit addressed, 16-bit register from the i2c bus. The function writes data by transmitting the register addr and then transmitting the data to the slave device.</td></tr><tr><td>**Type**</td><td>`i2c_regop_res_t write_reg16(uint8_t device_addr, uint16_t reg, uint16_t data)`</td></tr><tr><td>**Parameters**</td><td>`i` the interface to the I2C master `device_addr` the address of the slave device to write to `reg` the 16-bit address of the register to write (most significant byte first) `data` the 16-bit value to write (most significant byte first)</td></tr><tr><td>**Returns**</td><td>`I2C_REGOP_DEVICE_NACK` if the address is NACKed, `I2C_REGOP_INCOMPLETE` if not all data was ACKed and `I2C_REGOP_SUCCESS` on successful completion of the write with every byte being ACKed.</td></tr></table>

Continued on next page

Type	i2c_master_if (continued)

Function	read_reg16_addr8
Description	Read an 16-bit register on a slave device from a 8-bit register address. This function reads a 8-bit addressed, 16-bit register from the i2c bus. The function reads data by transmitting the register addr and then reading the data from the slave device. It is assumed that the data is return most significant byte first on the bus. Note that no stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

Type	`uint16_t`
	`read_reg16_addr8(uint8_t device_addr,`
	`                 uint8_t reg,`
	`                 i2c_regop_res_t &result)`

Parameters	`i` the interface to the I2C master `device_addr` the address of the slave device to read from `reg` the address of the register to read `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 16-bit value of the register

Continued on next page

Type	i2c_master_if (continued)

	Function: write_reg16_addr8
	Description: Write an 16-bit register on a slave device from a 8-bit register address.
	This function writes a 8-bit addressed, 16-bit register from the i2c bus. The function writes data by transmitting the register addr and then transmitting the data to the slave device.
	Type: `i2c_regop_res_t`
	`write_reg16_addr8(uint8_t device_addr,`
	`                   uint8_t reg,`
	`                   uint16_t data)`
	Parameters:
	`i` the interface to the I2C master
	`device_addr` the address of the slave device to write to
	`reg` the address of the register to write
	`data` the 16-bit value to write (most significant byte first)
	Returns: `I2C_REGOP_DEVICE_NACK` if the address is NACKed, `I2C_REGOP_INCOMPLETE` if not all data was ACKed and `I2C_REGOP_SUCCESS` on successful completion of the write with every byte being ACKed.

3.4 I^2C master asynchronous interface

Type	**i2c_master_async_if**
Description	This interface is used to communicate with an I2C master component asynchronously. It provides facilities for reading and writing to the bus.
Functions	

Function	**write**
Description	Initialize a write to an I2C bus.

Type	`[[guarded]]`
	`void write(uint8_t device_addr,`
	`           uint8_t buf[n],`
	`           size_t n,`
	`           int send_stop_bit)`

Parameters	`device_addr` the address of the slave device to write to `buf` the buffer containing data to write `n` the number of bytes to write `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.

Continued on next page

Type	i2c_master_async_if (continued)

Function	read
Description	Initialize a read to an I2C bus.

Type	`[[guarded]]`
	`void read(uint8_t device_addr,`
	`          size_t n,`
	`          int send_stop_bit)`

Parameters	`device_addr` the address of the slave device to read from. `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.

Function	operation_complete
Description	Completed operation notification. This notification will fire when a read or write is completed.

Type	`[[notification]]`
	`slave void operation_complete(void)`

Continued on next page

Type	i2c_master_async_if (continued)

Function	get_write_result
Description	Get write result. This function should be called after a write has completed.

Type	`[[clears_notification]]`
	`i2c_res_t`
	`get_write_result(size_t &num_bytes_sent)`

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

Function	get_read_data
Description	Get read result. This function should be called after a read has completed.

Type	`[[clears_notification]]`
	`i2c_res_t get_read_data(uint8_t buf[n], size_t n)`

Parameters	`buf` the buffer to fill with data. `n` the number of bytes to read, this should be the same as the number of bytes specified in `read()`, otherwise the behavior is undefined.
Returns	I2C_ACK if the write was acknowledged by the slave device, otherwise I2C_NACK.

Function	send_stop_bit
Description	Send a stop bit. 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 `read()` or `write()` functions.
Type	`void send_stop_bit(void)`

Continued on next page

Type	i2c_master_async_if (continued)
	Function: shutdown **Description**: Shutdown the I2C component. This function will cause the I2C task to shutdown and return. **Type**: `void shutdown()`

Function	shutdown
Description	Shutdown the I2C component. This function will cause the I2C task to shutdown and return.
Type	`void shutdown()`

4 Slave API

All I^{2}C slave functions can be accessed via the `i2c.h` header:

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

You will also have to add `lib_i2c` to the `USED_MODULES` field of your application Makefile.

4.1 Creating an I^{2}C slave instance

Function	i2c_slave
Description	I2C slave task. This function instantiates an i2c_slave component.

Type	`[[combinable]]`
	`void`
	`i2c_slave(client i2c_slave_callback_if i,`
	`          port p_scl,`
	`          port p_sda,`
	`          uint8_t device_addr)`

Parameters	`i` the client end of the i2c_slave_callback_if interface. The component takes the client end and will make calls on the interface when the master performs reads or writes. `p_scl` the SCL port of the I2C bus `p_sda` the SDA port of the I2C bus `device_addr` the address of the slave device

4.2 I^{2}C slave interface

Type	**i2c_slave_callback_if**
Description	This interface is used to communicate with an I2C slave component. It provides facilities for reading and writing to the bus. The I2C slave component acts a to this interface. So the application must respond to these calls (i.e. the members of the interface are callbacks to the application).
Functions	

Function	**ack_read_request**
Description	Master has requested a read. This callback function is called by the component 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).

Type	`[[guarded]]`
	`i2c_slave_ack_t ack_read_request(void)`

Returns	the callback must return either I2C_SLAVE_ACK or I2C_SLAVE_NACK.

Function	**ack_write_request**
Description	Master has requested a write. This callback function is called by the component 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).

Type	`[[guarded]]`
	`i2c_slave_ack_t ack_write_request(void)`

Returns	the callback must return either I2C_SLAVE_ACK or I2C_SLAVE_NACK.

Continued on next page

Type	i2c_slave_callback_if (continued)

Function	master_requires_data
Description	Master requires data. This callback function will be called when the I2C master requires data from the slave.

Type	`[[guarded]]`
	`uint8_t master_requires_data()`

Returns	the data to pass to the master.

Function	master_sent_data
Description	Master has sent some data. This callback function will be called when the I2C master has transferred a byte of data to the slave.

Type	`[[guarded]]`
	`i2c_slave_ack_t master_sent_data(uint8_t data)`

Function	stop_bit
Description	Stop bit. This callback function will be called by the component when a stop bit is sent by the master.
Type	`void stop_bit(void)`

Function	shutdown
Description	Shutdown the I2C component. This function will cause the I2C slave task to shutdown and return.

Type	`[[notification]]`
	`slave void shutdown()`

APPENDIX A - Known Issues

- The reg_ops_nack test fails on the XS1 architecture because it is unable to meet timing. This library is not recommended for use with the XS1 architecture.

APPENDIX B - I2C library change log

B.1 5.0.0

- CHANGE: i2c_master_single_port no longer supported on XS1.
- CHANGE: Removed the start_read_request() and start_write_request() functions from the i2c_slave_callback_if.
- CHANGE: Removed the start_master_read() and start_master_write() functions from the i2c_slave_callback_if.
- RESOLVED: Fixed timing of i2c master (both single port and multi-port).
- RESOLVED: Fixed bug with the master not coping with clock stretching on start bits.

B.2 4.0.2

- RESOLVED: Make use of Wavedrom in documentation generation offline (fixes automated build due to a known Wavevedrom issue where it would generate zero size PNG)

B.3 4.0.1

- RESOLVED: Suppressed warning “argument 1 of ‘i2c_master_async_aux’ slices interface preventing analysis of its parallel usage”.

B.4 4.0.0

- CHANGE: Register read/write functions are now all MSB first
- RESOLVED: i2c slave working properly (versions pre 4.0.0 not suitable for i2c slave)
- RESOLVED: Fixed byte ordering of write_reg16_addr8()
- RESOLVED: Fixed master transmitting on multi-bit port

B.5 3.1.6

- CHANGE: Change title to remove special characters

B.6 3.1.5

- CHANGE: Update app notes

B.7 3.1.4

- CHANGE: Remove invalid app notes

B.8 3.1.3

- CHANGE: Update to source code license and copyright

B.9 3.1.2

- RESOLVED: Fix incorrect reading of r/w bit in slave component

B.10 3.1.1

- CHANGE: Minor user guide updates

B.11 3.1.0

- ADDED: Add support for reading on i2c_master_single-port for xCORE-200 series.
- CHANGE: Document reg_read functions more clearly with respect to stop bit behavior.

B.12 3.0.0

- CHANGE: Consolidated version, major rework from previous I2C components.
- Changes to dependencies:
  - lib_logging: Added dependency 2.0.0
  - lib_xassert: Added dependency 2.0.0