I2S/TDM Library

A software library that allows you to control an I^{2}S or TDM (time division multiplexed) bus via xCORE ports. I^{2}S and TDM are digital data streaming interfaces particularly appropriate for transmission of audio data. The components in the library are controlled via C using the XMOS multicore extensions (xC) and can either act as I^{2}S master, TDM master or I^{2}S slave.

Features

- I^{2}S master, TDM master and I^{2}S slave modes.
- Handles multiple input and output data lines.
- Support for standard I^{2}S, left justified or right justified data modes for I^{2}S.
- Support for multiple formats of TDM synchronization signal.
- Efficient "frame-based" versions of I^{2}S master and slave allowing use of processor cycles in between I2S signal handling.
- Sample rate support up to 192kHz or 768kHz for "frame-based" versions.
- Up to 32 channels in/32 channels out (depending on sample rate and protocol).

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
I2S Master (frame-based)	3 + data lines	3 x (1-bit) + data lines	2	~1.6K	1
I2S Master	3 + data lines	3 x (1-bit) + data lines	2	~2.1K	1
I2S Slave (frame-based)	2 + data lines	2 x (1-bit) + data lines	1	~1.6K	1
I2S Slave	2 + data lines	2 x (1-bit) + data lines	1	~1.6K	1
TDM Master	2 + data lines	2 x (1-bit) + data lines	1	~1.8K	1

Software version and dependencies

This document pertains to version 3.0.0 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)

Notes on "frame-based" I^{2}S implementations

The library supports both "sample-based" and "frame-based" versions of I^{2}S master and slave. The "frame-based" versions are recommended for new designs and support higher I^{2}S channel counts and rates. In

addition, the number of callbacks to pass data to and from the I^{2}S handler task are reduced. "Frame-based" I^{2}S pass an array of channels per sample period whereas "sample-based" versions make a callback per channel within a sample period. The "Frame-based" callbacks are all grouped together allowing the user side to make maximum use of the MIPS between I^{2}S frames. For example, a 48kHz (20.83us) I^{2}S interface supports a total of 19us processing per sample period, in any order, across the callbacks. The older "channel-based" versions are currently maintained to provide compatibility with existing code examples.

Related application notes

The following application notes use this library:

- AN00162 - Using the I^{2}S library

1 External signal description

1.1 I²S

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 Table 1.

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.

Table 1: I²S data wires

The configuration of an I²S signal depends on the parameters shown in Table 2.

MCLK_BCLK_RATIO	The fixed ratio between the master clock and the bit clock.
MODE	The mode - either I²S or left justified.

Table 2: I²S configuration parameters

The *MCLK_BCLK_RATIO* should be such that 64 bits can be output by the bit clock at the data rate of the I²S signal. For example, a 24.576MHz master clock with a ratio of 8 gives a bit clock at 3.072MHz. This bit clock can output 64 bits at a frequency of 48kHz - which is the underlying rate of the data.

The master signals data transfer should occur by a transition on the *LRCLK* wire. There are two supported modes for I²S. In *I2S mode* (shown in Figure 1) data is transferred on the second falling edge after the *LRCLK* transitions.

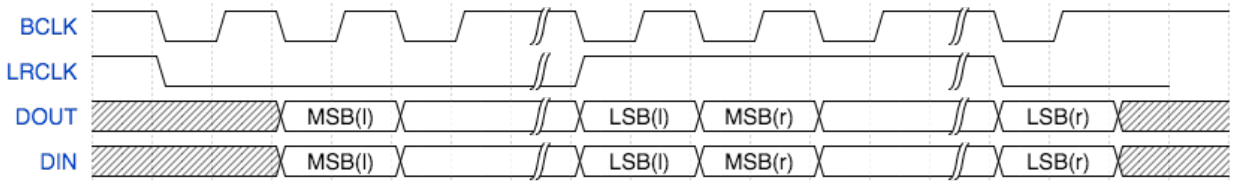

Figure 1: I²S Mode

In *Left Justified Mode* (shown in Figure 2) the data is transferred on the next falling edge after the *LRCLK* transition.

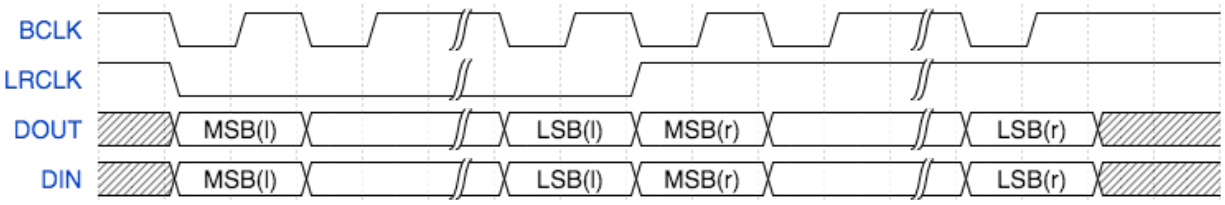

Figure 2: Left Justified Mode

In either case the signal multiplexes two channels of data onto one data line. When the *LRCLK* is low, the *left* channel is transmitted. When the *LRCLK* is high, the *right* channel is transmitted.

All data is transmitted most significant bit first. The xCORE I^{2}S library assumes 32 bits of data between *LRCLK* transitions. How the data is aligned is expected to be done in software by the application. For example, some audio CODECs have a *Right Justified* mode; to attain this mode the library should be set to *Left Justified* mode to align the *LRCLK* signal and then the data should be right shifted by the application before being passed to the library.

1.1.1 Connecting I^{2}S signals to the xCORE device

The I^{2}S wires need to be connected to the xCORE device as shown in Figure 3 and Figure 4. The signals can be connected to any one bit ports on the device provide they do not overlap any other used ports and are all on the same tile.

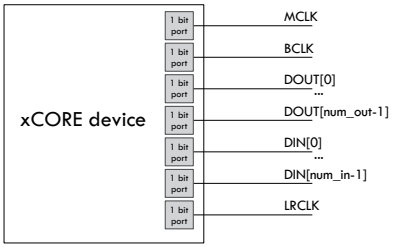

Figure 3: I^{2}S connection to the xCORE device (xCORE as I^{2}S master)

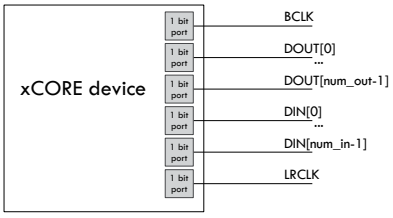

Figure 4: I^{2}S connection to the xCORE device (xCORE as I^{2}S slave)

If only one data direction is required then the *DOUT* or *DIN* lines need not be connected.

1.1.2 I²S master speeds and performance

The speed and number of data wires that can be driven by the I²S library running as I²S master depends on the speed of the logical core that runs the code and the amount of processing that occurs in the user callbacks for handling the data from the library. Table 3 and Table 4 show configurations that are known to work for small amounts of callback processing. Other speeds will be achievable depending on the amount of processing in the application and the logical core speed.

MCLK FREQ	MCLK/BCLK RATIO	SAMPLE FREQ	NUM IN (num channels)	NUM OUT (num channels)
24.576MHz	2	192000	1 (2)	1 (2)
24.576MHz	4	96000	2 (4)	2 (4)
24.576MHz	8	48000	4 (8)	4 (8)
12.288MHz	2	96000	2 (4)	2 (4)
12.288MHz	4	48000	4 (8)	4 (8)

Table 3: Known working I²S master configurations on a 62.5MHz core

MCLK FREQ	MCLK/BCLK RATIO	SAMPLE FREQ	NUM IN (num channels)	NUM OUT (num channels)
24.576MHz	2	192000	2 (4)	2 (4)
24.576MHz	4	96000	4 (8)	4 (8)
12.288MHz	2	96000	4 (8)	4 (8)

Table 4: Known working I²S master configurations on a 83.3MHz core

On the xCORE-200 the frame-based I²S master can be used. This uses hardware clock dividers only available in the the xCORE-200 and a more efficient callback interface to achieve much higher throughputs. Table 5 shows the known working configurations:

MCLK FREQ	MCLK/BCLK RATIO	SAMPLE FREQ	NUM IN (num channels)	NUM OUT (num channels)
49.152MHz	1	768000	4 (8)	4 (8)
24.576MHz	2	192000	4 (8)	4 (8)
24.576MHz	24	16000	4 (8)	4 (8)

Table 5: Known working I²S frame-based master configurations on a 62.5MHz core

1.1.3 I²S slave speeds and performance

The speed and number of data wires that can be driven by the I²S library running as slave depends on the speed of the logical core that runs the code and the amount of processing that occurs in the user callbacks for handling the data from the library. Table 6 shows configurations that are known to work for small amounts of callback processing. Other speeds will be achievable depending on the amount of processing in the application and the logical core speed. Note that the when acting as slave the performance of the library only depends on the bit clock frequency, not the underlying master clock frequency.

BCLK FREQ	SAMPLE FREQ	NUM IN (num channels)	NUM OUT (num channels)
12.288MHz	192000	4 (8)	4 (8)

Table 6: Known working I²S slave configurations on a 62.5MHz core

Frame-based I²S master can be used. This uses a more efficient callback interface to achieve much higher throughputs by transferring a frame (all channels in one sample period) at a time and re-ordered callbacks and I/O operations so that maximum back pressure tolerance is achieved. The table Table 7 shows the known working configurations. Other configurations may be possible depending on performance:

BCLK FREQ	SAMPLE FREQ	NUM IN (num channels)	NUM OUT (num channels)
24.576MHz	384000	4 (8)	4 (8)
12.288MHz	192000	4 (8)	4 (8)
3.072MHz	16000	4 (8)	4 (8)

Table 7: Known working I²S frame-based master configurations on a 62.5MHz core

I²S "frame-master" is capable of running at higher rates such as 768kHz within a 62.5MIPS logical core. However, it may be necessary to modify the port timing delays to ensure proper sampling of the data and LRCLK lines. There are methods for doing this using pad and/or sample delays however this is beyond the scope of this document. Please consult *I/O timings for xCORE200* available on xmos.com for further information.

1.2 TDM

TDM is a protocol that multiplexes several signals onto one wire. It is a protocol between two devices where one is the *master* and one is the *slave* . The protocol is made up of three signals shown in Table 8.

BCLK	Bit clock line, driven by external oscillator.
FSYNC	The frame sync line. 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.

Table 8: TDM data wires

Unlike I^2S, the bit clock is not a divide of an underlying master clock.

The configuration of a TDM signal depends on the parameters shown in Table 9.

CHANNELS_PER_FRAME	The number of channels multiplexed into a frame on the data line.
FSYNC_OFFSET	The number of bits between the frame sync signal transitioning and data being drive on the data line.
FSYNC_LENGTH	The number of bits that the frame sync signal stays high for when signaling frame start.

Table 9: TDM configuration parameters

Figure 5 and Figure 6 show example waveforms for TDM with different offset and sync length values.

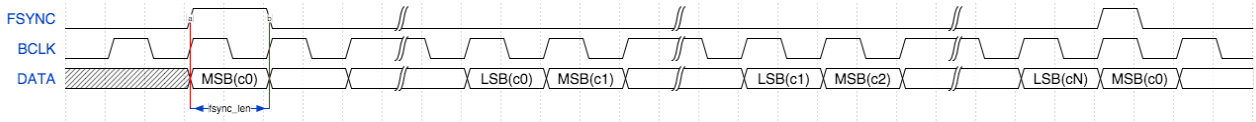

Figure 5: TDM signal (sync offset 0, sync length 1)

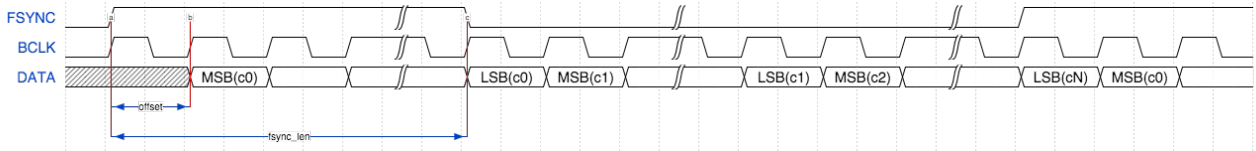

Figure 6: TDM signal (sync offset 1, sync length 32)

The master signals a frame by driving the *FSYNC* signal high. After a delay of *FSYNC_OFFSET* bits, data is driven. Data is driven most significant bit first. First, 32 bits of data from Channel 0 is driven, then 32 bits from channel 1 up to channel N (when N is the number of channels per frame). The next frame is then signaled (there is no padding between frames).

1.2.1 Connecting TDM signals to the xCORE device

The TDM wires need to be connected to the xCORE device as shown in Figure 7. The signals can be connected to any one bit ports on the device provide they do not overlap any other used ports and are all on the same tile.

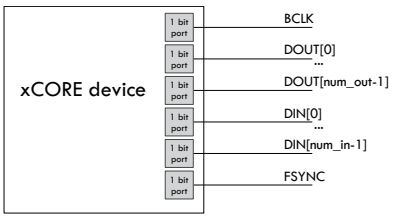

Figure 7: TDM connection to the xCORE device

If only one data direction is required then the *DOUT* or *DIN* lines need not be connected.

1.2.2 TDM speeds and performance

The speed and number of data wires that can be driven by the I^{2}S library running as TDM master depends on the speed of the logical core that runs the code and the amount of processing that occurs in the user callbacks for handling the data from the library. Table 10 show configurations that are known to work for small amounts of callback processing. Other speeds will be achievable depending on the amount of processing in the application and the logical core speed.

BCLK FREQ	CHANNELS PER FRAME	SAMPLE FREQ	NUM IN (num channels)	NUM OUT (num channels)
12.288MHz	8	48000	2 (16)	2 (16)
6.144MHz	4	48000	4 (16)	4 (16)

Table 10: Known working TDM configurations on a 62.5MHz core

2 Usage

All I²S functions can be accessed via the `i2s.h` header:

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

You will also have to add `lib_i2s` to the USED_MODULES field of your application Makefile.

2.1 The I²S callback interface

All major functions in the I²S library work by controlling the I²S or TDM bus on its own logical core on an xCORE device. The library will then make callbacks to the application when it receives a sample or needs to send a sample.

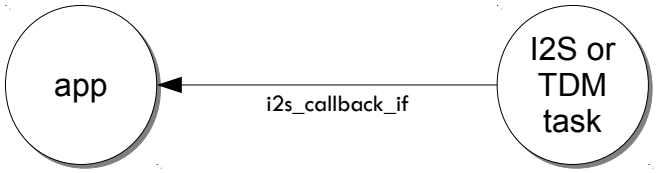

Figure 8: I²S callback usage

The callbacks are implemented by the application providing a task which receives requests on the `i2s_callback_if` xC interface. The application tasks can run the callbacks on the same logical core by implementing a *distributable* task. More information on interfaces and tasks can be be found in the XMOS Programming Guide (see XM-004440-PC).

A template application task is shown below. The specific contents of each callback will depend on the application:

```
[[distributable]]
void my_application(server i2s_callback_if i2s) {
while (1) {
  select {
  case i2s.init(i2s_config_t &?i2s_config, tdm_config_t &?tdm_config):
    i2s_config.mclk_to_bclk_ratio = 2;
    i2c_config.mode = I2S_MODE_LEFT_JUSTIFIED;
    ...
    break;

  case i2s.restart_check() -> i2s_restart_t restart:
    ...
    break;

  case i2s.receive(size_t index, int32_t sample):
    ...
    break;

  case i2s.send(size_t index) -> int32_t sample:
    ...
    break;
  }
}
```

The send/receive callbacks pass a channel index parameter to the application. This channel maps to the

data signals as shown in §2.7.

The initialization callback will provide configuration structures relevant to the communication bus being used. The application can set the parameters of the bus (*MCLK/BCLK* ratio, *LRCLK* alignment etc.) at this point.

2.2 I²S frame-based master usage

The I²S frame-based master task (only supported on xCORE-200) is instantiated as a parallel task that run in a `par` statement. The application can connect via the `i2s_frame_callback_if` interface connection. For example, the following code instantiates an I²S frame-based master component and connects to it:

```
out buffered port:32 p_dout[2] = {XS1_PORT_1D, XS1_PORT_1E};
in buffered port:32 p_din[2]  = {XS1_PORT_1I, XS1_PORT_1K};
port p_mclk  = XS1_PORT_1M;
out buffered port:32 p_bclk  = XS1_PORT_1A;
out buffered port:32 p_lrclk = XS1_PORT_1C;

clock bclk = XS1_CLKBLK_1;

int main(void) {
  i2s_frame_callback_if i_i2s;
  par {
    i2s_frame_master(i_i2s, p_dout, 4, p_din, 4,
             p_bclk, p_lrclk, p_mclk, bclk);
    my_application(i_i2s);
  }
  return 0;
}
```

2.3 I²S master usage

The I²S master task is instantiated as a parallel task that run in a `par` statement. The application can connect via the `i2s_callback_if` interface connection. For example, the following code instantiates an I²S master component and connects to it:

```
out buffered port:32 p_dout[2] = {XS1_PORT_1D, XS1_PORT_1E};
in buffered port:32 p_din[2]  = {XS1_PORT_1I, XS1_PORT_1K};
port p_mclk  = XS1_PORT_1M;
out buffered port:32 p_bclk  = XS1_PORT_1A;
out buffered port:32 p_lrclk = XS1_PORT_1C;

clock mclk = XS1_CLKBLK_1;
clock bclk = XS1_CLKBLK_2;

int main(void) {
  i2s_callback_if i_i2s;
  configure_clock_src(mclk, p_mclk);
  start_clock(mclk);
  par {
    i2s_master(i_i2s, p_dout, 2, p_din, 2,
             p_bclk, p_lrclk, bclk, mclk);
    my_application(i_i2s);
  }
  return 0;
}
```

2.4 I²S frame-based slave usage

The I²S frame slave task is instantiated as a parallel task that run in a par statement. The application can connect via the i2s_frame_callback_if interface connection. For example, the following code instantiates an I²S slave component and connects to it:

```
out buffered port:32 p_dout[2] = {XS1_PORT_1D, XS1_PORT_1E};
in buffered port:32 p_din[2]  = {XS1_PORT_1I, XS1_PORT_1K};
in port p_bclk  = XS1_PORT_1A;
in port p_lrclk = XS1_PORT_1C;

clock bclk = XS1_CLKBLK_1;

int main(void) {
  par {
    i2s_frame_slave(i2s_i, p_dout, 2, p_din, 2,
              p_bclk, p_lrclk, bclk);
    my_application(i_i2s);
  }
  return 0;
}
```

2.5 I²S slave usage

The I²S slave task is instantiated as a parallel task that run in a par statement. The application can connect via the i2s_callback_if interface connection. For example, the following code instantiates an I²S slave component and connects to it:

```
out buffered port:32 p_dout[2] = {XS1_PORT_1D, XS1_PORT_1E};
in buffered port:32 p_din[2]  = {XS1_PORT_1I, XS1_PORT_1K};
in port p_bclk  = XS1_PORT_1A;
in port p_lrclk = XS1_PORT_1C;

clock bclk = XS1_CLKBLK_1;

int main(void) {
  par {
    i2s_slave(i2s_i, p_dout, 2, p_din, 2,
              p_bclk, p_lrclk, bclk);
    my_application(i_i2s);
  }
  return 0;
}
```

Slave has an additional config option to sample data and word clock on falling edge of bit clock, instead of rising edge. Data is then output on rising edge instead of falling edge. This option is useful with non-standard masters that invert their bit clock.

2.6 TDM usage

The TDM master task is instantiated as a parallel task that run in a par statement. The application can connect via the i2s_callback_if interface connection. For example, the following code instantiates an TDM master component and connects to it:

```
out buffered port:32 p_dout[2] = {XS1_PORT_1D, XS1_PORT_1E};
in buffered port:32 p_din[2]  = {XS1_PORT_1I, XS1_PORT_1K};
in port p_bclk  = XS1_PORT_1A;
out buffered port:32 p_fsync = XS1_PORT_1C;

clock bclk = XS1_CLKBLK_1;

int main(void) {
  i2s_callback_if i_i2s;
  configure_clock_src(bclk, p_bclk);
  par {
    tdm_master(i2s_i, p_fsync, p_dout, 2, p_din, 2, bclk);
    my_application(i_i2s);
  }
  return 0;
}
```

2.7 Channel numbering

The callback interface numbers the channels being sent/received for the send and receive callbacks. There is a fixed mapping from these channel indices to the physical interface begin used.

2.7.1 I²S channel numbering

The data words within I²S frames have even channel numbers assigned to the left samples (first within the frame) and odd numbers assigned to the right (second within the frame) samples.

The actual sample number will be given with respect to the order that the ports are provided in the data in and data out array arguments to the component.

For example, in a system with 4 data out ports and 4 data in ports declared as:

```
out buffered port:32 p_dout[4] = {XS1_PORT_1A, XS1_PORT_1B, XS1_PORT_1C, XS1_PORT_1D};
in buffered port:32 p_din[4] = {XS1_PORT_1E, XS1_PORT_1F, XS1_PORT_1G, XS1_PORT_1H};
```

The channels wil be numbered as indicated in Figure 9:

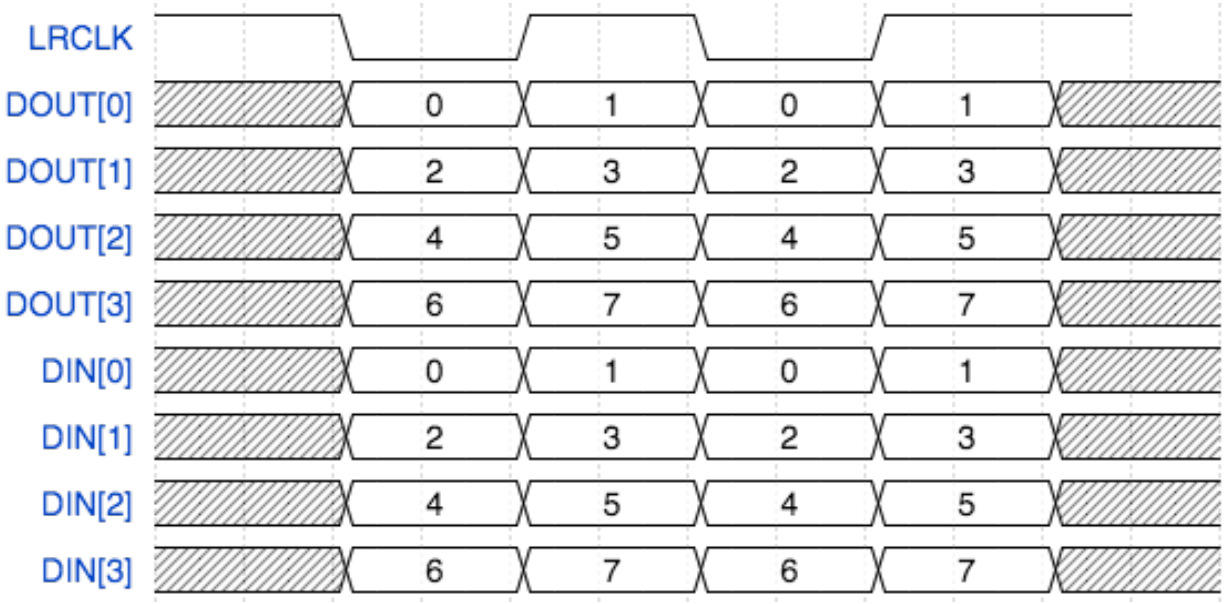

Figure 9: I²S channel numbering

2.7.2 TDM channel numbering

The data words within TDM frames are assigned sequentially from the start of the frame. Each data line will have its channel numbers assigned in the order that the ports are provided in the data in and data out array arguments to the component.

For example, in a system with 2 data out ports and 2 data in ports declared as:

```
out buffered port:32 p_dout[2] = {XS1_PORT_1A, XS1_PORT_1B};
in buffered port:32 p_din[2] = {XS1_PORT_1E, XS1_PORT_1F};
```

With the number of channels per frame as 4, the samples will be numbered as indicated in Figure 10:

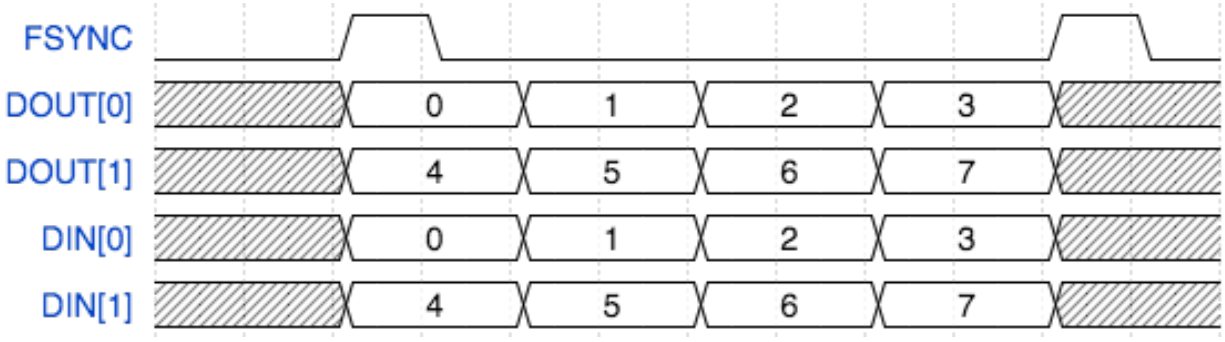

Figure 10: TDM channel numbering

2.8 Callback sequences

The send/receive callbacks of "sample-based" I²S callbacks occur in a pre-determined order. The sequence consists of receipt of all even channel, sending of all even channels, receipt of all odd channels and then sending of all odd channels.

Since the hardware port buffers within the xCORE device there is an initial sequences of sends after initialization. Similarly there is a final sequences of receives after a restart/shutdown request. Table 11 shows an example sequence of callbacks for two output lines and two input lines (four channels in and four channels out).

Initial send:	S0 S2 S1 S3
Frame:	R0 R2 S0 S2 R1 R3 S1 S3
Frame:	R0 R2 S0 S2 R1 R3 S1 S3
...	...
Frame:	R0 R2 S0 S2 R1 R3 S1 S3
Final receive:	R0 R2 R1 R3

Table 11: Sample-based I²S callback sequence

For "frame-based" I²S implementations the callback sequence is much simpler. Table 12 shows an example sequence.

Initial send:	Init, Send All
Frame:	Restart check, Send All, Receive All
Frame:	Restart check, Send All, Receive All
...	...
Frame:	Restart check, Send All, Receive All
Final receive:	Restart check (I2S_RESTART), Receive All

Table 12: Frame-based I²S callback sequence

When using TDM, the receive callbacks for a channel occur after the send callbacks. The receive callback for the last channel of the frame will occur after the send callback for the next frame. After a restart request a tail of receive callbacks for the last channel of the final frame will occur. Table 13 shows an example TDM callback sequence for two data lines in and out with four channels per frame.

S0 S4 S1 S5 R0 R4 S2 S6 R1 R5 S3 S7 R2 R6
S0 S4 R3 R7 S1 S5 R0 R4 S2 S6 R1 R5 S3 S7 R2 R6
...
S0 S4 R3 R7 S1 S5 R0 R4 S2 S6 R1 R5 S3 S7 R2 R6
S0 S4 R3 R7 S1 S5 R0 R4 S2 S6 R1 R5 S3 S7 R2 R6
R3 R7

Table 13: Sample TDM callback sequence

In both cases the components attempt to distribute the calling of the callbacks evenly within the frame to allow processing to occur throughout the frame evenly.

The `restart_check` callback is called once per frame to allow the application to request a restart/shutdown of the data bus.

2.9 Clock configuration

For the I^{2}S master and TDM components is it the application's responsibility to set up and start the internal clock used for the master clock before calling the component.

For example, the following code configures a clock to be based of an incoming data wire and starts the clock:

```
configure_clock_src(mclk, p_mclk);
start_clock(mclk);
```

For more information on configuring clocks see the XMOS tools user guide.

3 API

3.1 Supporting types

Type	i2s_mode_t
Description	I2S mode. This type is used to describe the I2S mode.
Values	I2S_MODE_I2S The LR clock transitions ahead of the data by one bit clock. I2S_MODE_LEFT_JUSTIFIED The LR clock and data are phase aligned.

Type	i2s_config_t
Description	I2S configuration structure. This structure describes the configuration of an I2S bus.
Fields	unsigned mclk_bclk_ratio The ratio between the master clock and bit clock signals. i2s_mode_t mode The mode of the LR clock. i2s_slave_bclk_polarity_t slave_bclk_polarity Slave bit clock polarity.

Type	tdm_config_t
Description	TDM configuration structure. This structure describes the configuration of a TDM bus.
Fields	int offset The number of bits that the FSYNC signal transitions before the data. Must be a value between -31 and 31. unsigned sync_len The length that the FSYNC signal stays high counted as ticks of the bit clock. unsigned channels_per_frame The number of channels in a TDM frame. This must be a power of 2.

Type	i2s_restart_t
Description	Restart command type. Restart commands that can be signalled to the I2S or TDM component.
Values	I2S_NO_RESTART Do not restart. I2S_RESTART Restart the bus (causes the I2S/TDM to stop and a new init callback to occur allowing reconfiguration of the BUS). I2S_SHUTDOWN Shutdown. This will cause the I2S/TDM component to exit.

3.2 Creating an I²S instance

Function	i2s_frame_master
Description	I2S frame-based master component This task performs I2S on the provided pins. It will perform callbacks over the i2s_frame_callback_if interface to get/receive frames of data from the application using this component. The component performs I2S master so will drive the word clock and bit clock lines. This is a more efficient version of i2s master which reduces callback frequency and allows useful processing to be done in distributable i2s handler tasks. It also uses xCORE200 specific features to remove the need for software BCLK generation which decreases processor overhead.

Type	`void`
	`i2s_frame_master(client i2s_frame_callback_if i2s_i,`
	`  out buffered port:32 (&?p_dout)[num_out],`
	`  static const size_t num_out,`
	`  in buffered port:32 (&?p_din)[num_in],`
	`  static const size_t num_in,`
	`  out port p_bclk,`
	`  out buffered port:32 p_lrclk,`
	`  in port p_mclk,`
	`  clock bclk)`

Parameters	`i2s_i` The I2S frame callback interface to connect to the application `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

Function	i2s_master
Description	I2S master component. This task performs I2S on the provided pins. It will perform callbacks over the i2s_callback_if interface to get/receive data from the application using this component. The component performs I2S master so will drive the word clock and bit clock lines.

Type	`void`
	`i2s_master(client i2s_callback_if i2s_i,`
	`  out buffered port:32 (&?p_dout)[num_out],`
	`  static const size_t num_out,`
	`  in buffered port:32 (&?p_din)[num_in],`
	`  static const size_t num_in,`
	`  out buffered port:32 p_bclk,`
	`  out buffered port:32 p_lrclk,`
	`  clock bclk,`
	`  const clock mclk)`

Parameters	`i2s_i` The I2S callback interface to connect to the application `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 will get configured for use with the bit clock `mclk` The clock connected to the master clock frequency. Usually this should be configured to be driven by an incoming master system clock.

Function	i2s_frame_slave
Description	I2S High Efficiency slave component. This task performs I2S on the provided pins. It will perform callbacks over the i2s_callback_if interface 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.

Type	`void`
	`i2s_frame_slave(client i2s_frame_callback_if i2s_i,`
	`  out buffered port:32 (&?p_dout)[num_out],`
	`  static const size_t num_out,`
	`  in buffered port:32 (&?p_din)[num_in],`
	`  static const size_t num_in,`
	`  in port p_bclk,`
	`  in buffered port:32 p_lrclk,`
	`  clock bclk)`

Parameters	`i2s_i` The I2S callback interface to connect to the application `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

Function	i2s_slave
Description	I2S slave component. This task performs I2S on the provided pins. It will perform callbacks over the i2s_callback_if interface 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.

Type	`void`
	`i2s_slave(client i2s_callback_if i2s_i,`
	`  out buffered port:32 (&?p_dout)[num_out],`
	`  static const size_t num_out,`
	`  in buffered port:32 (&?p_din)[num_in],`
	`  static const size_t num_in,`
	`  in port p_bclk,`
	`  in buffered port:32 p_lrclk,`
	`  clock bclk)`

Parameters	`i2s_i` The I2S callback interface to connect to the application `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

3.3 Creating an TDM instance

Function	tdm_master
Description	TDM master component. This task performs TDM on the provided pins. It will perform callbacks over the i2s_callback_if interface to get/receive data from the application using this component. The component performs as TDM master so will drive the fsync signal.

Type	`void`
	`tdm_master(client interface i2s_callback_if tdm_i,`
	`  out buffered port:32 p_fsync,`
	`  out buffered port:32 (&?p_dout)[num_out],`
	`  size_t num_out,`
	`  in buffered port:32 (&?p_din)[num_in],`
	`  size_t num_in,`
	`  clock clk)`

Parameters	`tdm_i` The TDM callback interface to connect to the application `p_fsync` The frame sync output port `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 `clk` The clock connected to the bit/master clock frequency. Usually this should be configured to be driven by an incoming master system clock.

3.4 The I²S frame-based callback interface

Type	i2s_frame_callback_if
Description	Interface representing callback events that can occur during the operation of the I2S task. This is a more efficient interface and reccomended for new designs.
Functions	

Function	init
Description	I2S frame-based initialization event callback. The I2S component will call this when it first initializes on first run of after a restart.
Type	`void init(i2s_config_t & ?i2s_config, tdm_config_t & ?tdm_config)`

Parameters	`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.
	
	`tdm_config`
	This structure is provided if the connected component drives an TDM bus. The members of the structure should be set to the required configuration.

Function	restart_check
Description	I2S frame-based restart check callback. This callback is called once per frame. The application must return the required restart behaviour.
Type	`i2s_restart_t restart_check()`
Returns	The return value should be set to I2S_NO_RESTART, I2S_RESTART or I2S_SHUTDOWN.

Continued on next page

Type	i2s_frame_callback_if (continued)

Function	receive	
Description	Receive an incoming frame of samples. This callback will be called when a new frame of samples is read in by the I2S frame-based component.	
Type	`void receive(size_t num_in, int32_t samples[num_in])`	
Parameters	`num_in`	The number of input channels contained within the array.
	`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.

Function	send	
Description	Request an outgoing frame of samples. This callback will be called when the I2S frame-based component needs a new frame of samples.	
Type	`void send(size_t num_out, int32_t samples[num_out])`	
Parameters	`num_out`	The number of output channels contained within the array.
	`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.

3.5 The I²S callback interface

Type	i2s_callback_if
Description	Interface representing callback events that can occur during the operation of the I2S task.
Functions	

Function | **init**

Description: I2S initialization event callback.
The I2S component will call this when it first initializes on first run of after a restart.

Type:
```
void init(i2s_config_t & ?i2s_config,
          tdm_config_t & ?tdm_config)
```

Parameters:

`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.

`tdm_config`
This structure is provided if the connected component drives an TDM bus. The members of the structure should be set to the required configuration.

Function | **restart_check**

Description: I2S restart check callback.
This callback is called once per frame. The application must return the required restart behaviour.

Type:
```
i2s_restart_t restart_check()
```

Returns: The return value should be set to I2S_NO_RESTART, I2S_RESTART or I2S_SHUTDOWN.

Continued on next page

Type	i2s_callback_if (continued)

Function	receive
Description	Receive an incoming sample. This callback will be called when a new sample is read in by the I2S component.
Type	`void receive(size_t index, int32_t sample)`
Parameters	`index` The index of the sample in the frame. `sample` The sample data as a signed 32-bit value. The component may not use all 32 bits of the value (for example, many I2S codecs are 24-bit), in which case the bottom bits are ignored.

Function	send
Description	Request an outgoing sample. This callback will be called when the I2S component needs a new sample.
Type	`int32_t send(size_t index)`
Parameters	`index` The index of the requested sample in the frame.
Returns	The sample data as a signed 32-bit value. 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.

APPENDIX A - Known Issues

No known issues.

APPENDIX B - I2S library change log

B.1 3.0.0

- REMOVED: Combined I2S and TDM master

B.2 2.4.0

- ADDED: Frame-based I2S slave implementation.
- CHANGE: AN00162 now uses frame-based I2S master component.

B.3 2.3.0

- ADDED: Configuration option for slave bit clock polarity. This allows supporting masters that toggle word clock and data on rising edge of bit clock.

B.4 2.2.0

- ADDED: Frame-based I2S master using the new i2s_frame_callback_if. This reduces the overhead of an interface call per sample.
- CHANGE: Reduce number of LR clock ticks needed to synchronise.
- RESOLVED: Documentation now correctly documents the valid values for FSYNC.
- RESOLVED: The I2S slave will now lock correctly in both I2S and LEFT_JUSTFIED modes. Previously there was a bug that meant LEFT_JUSTFIED would not work.

B.5 2.1.3

- CHANGE: Slave mode now includes sync error detection and correction e.g. when bit-clock is interrupted

B.6 2.1.2

- RESOLVED: .project file fixes such that example(s) import into xTIMEComposer correctly

B.7 2.1.1

- CHANGE: Update to source code license and copyright

B.8 2.1.0

- CHANGE: Input or output ports can now be null, for use when input or output-only is required
- CHANGE: Software license changed to new license

B.9 2.0.1

- CHANGE: Performance improvement to TDM to allow 32x32 operation
- RESOLVED: Bug fix to initialisation callback timing that could cause I2S lock up

B.10 2.0.0

- CHANGE: Major update to API from previous I2S components
- Changes to dependencies:

- lib_logging: Added dependency 2.0.0
- lib_xassert: Added dependency 2.0.0