SMART ARM-based Microcontrollers

AT07451: SAM D21/DA1 Inter-IC Sound Controller (I2S) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Inter-IC Sound Controller functionality.

The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- I^2S (Inter-IC Sound Controller)

The following devices can use this module:

- Atmel | SMART SAM D21
- Atmel | SMART SAM DA1

The outline of this documentation is as follows:

Table of Contents

1. Software License

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

The I²S provides bidirectional, synchronous, digital audio link with external audio devices through these signal pins:

- Serial Data (SDm)
- Frame Sync (FSn)
- Serial Clock (SCKn)
- Master Clock (MCKn)

The I²S consists of two Clock Units and two Serializers, which can be separately configured and enabled, to provide varies functionalities as follow:

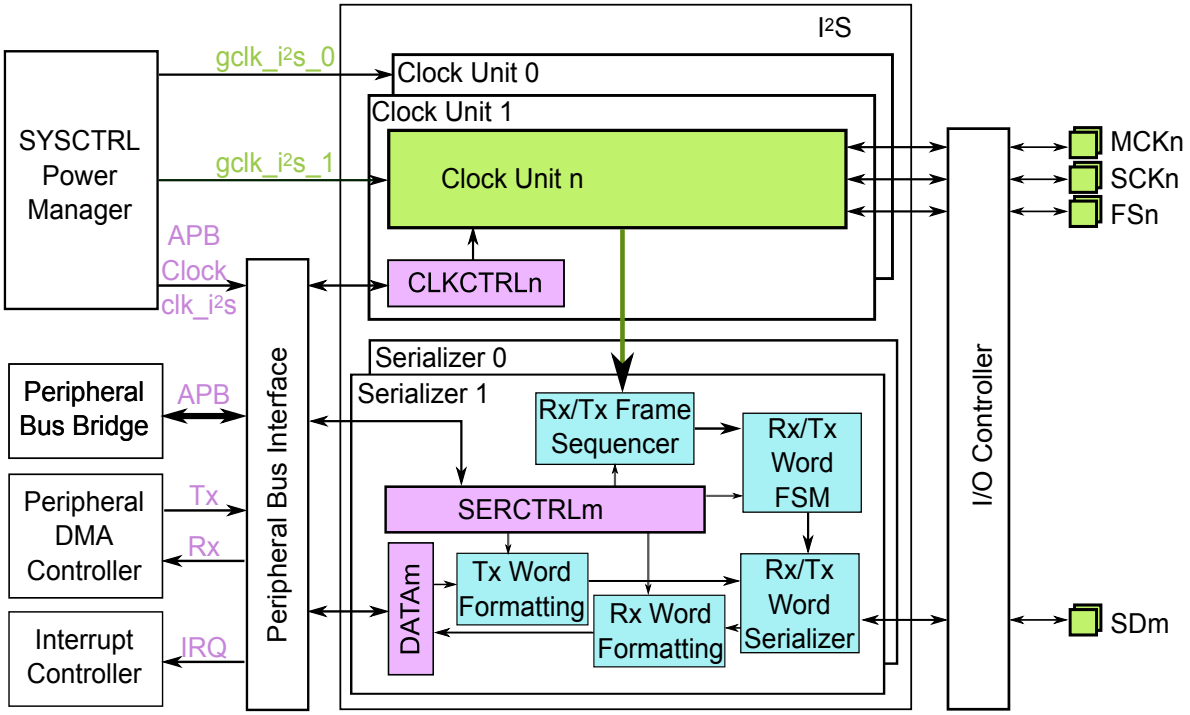
- Communicate to Audio CODECs as Master or Slave, or provides clock and frame sync signals as Controller
- Communicate to DAC or ADC through dedicated I²S serial interface
- Communicate to multi-slot or multiple stereo DACs or ADCs, via Time Division Multiplexed (TDM) format
- Reading mono or stereo MEMS microphones, using the Pulse Density Modulation (PDM) interface

The I²S supports compact stereo data word, where left channel data bits are in lower half and right channel data bits are in upper half. It reduces the number of data words for stereo audio data and the DMA bandwidth.

In master mode, the frame is configured by number of slots and slot size, and allows range covering 16fs to 1024fs MCK, to provide oversampling clock to an external audio CODEC or digital signal processor (DSP).

A block diagram of the I²S can be seen in Figure 3-1 I2S Block Diagram on page 7.

Figure 3-1. I²S Block Diagram



This driver for I²S module provides an interface to:

- Initialize and control I²S module
- Configure and control the I²S Clock Unit and Serializer
- Transmit/receive data through I²S Serializer

3.1. Clocks

To use I²S module, the I²S bus interface clock (clk_i2s) must be enabled via Power Manager.

For each I²S Clock Unit, a generic clock (gclk_i2s_n) is connnected. When I²S works in master mode the generic clock is used. It should be prepared before clock unit is used. In master mode the input generic clock will be used as MCK for SCKn and FSn generation, in addition, the MCK could be devided and output to I²S MCKn pin, as oversampling clock to external audio device.

The I²S Serializer uses clock and control signal from Clock Unit to handle transfer. Select different clock unit with different configurations allows the I²S to work as master or slave, to work on non-related clocks.

When using the driver with ASF, enabling the register interface is normally done by the `init` function. The Generic Clock Controller (GCLK) source for the asynchronous domain is normally configured and set through the _configuration struct_ / _init_ function. If GCLK source != 0 is used, this source has to be configured and enabled through invoking the system_gclk driver function when needed, or modifying conf_clock.h to enable it at the beginning.

3.2. Audio Frame Generation

Audio sample data for all channels are sent in frames, one frame can consist 1 - 8 slots where each slot can be configured to a size 8-bit, 16-bit, 24-bit, or 32-bit. The audio frame synch clock is generated by the I^2S Clock unit in the master/controller mode. The frame rate (or frame sync frequency) is calculated as follows:

FS = SCK / number_of_slots / number_of_bits_in_slot

The serial clock (SCK) source is either an external source (slave mode) or generated by the I^2S clock unit (controller or master mode) using the MCK as source.

SCK = MCK / sck_div
Note: SCK generation division value is MCKDIV in register.

MCK is either an external source or generated using the GCLK input from a generic clock generator.

3.3. Master, Controller, and Slave Modes

The I^2S module has three modes: master, controller, and slave.

3.3.1. Master

In master mode the module will control the data flow on the I^2S bus and can be responsible for clock generation. The Serializers are enabled and will transmit/receive data. On a bus with only master and slave the SCK, and FS clock signal will be outputted on the SCK and FS pin on the master module. MCK can optionally be outputted on the MCK pin, if there is a controller module on the bus the SCK, FS, and optionally the MCK clock is sourced from the same pins. Serial data will be trancieved on the SD pin in both scenarios.

3.3.2. Controller

In controller mode the module will generate the clock signals, but the Serializers are disabled and no data will be transmitted/received by the module in this mode. The clock signals is outputted on the SCK, FS and optionally the MCK pin.

3.3.3. Slave

In slave mode the module will use the SCK and FS clock source from the master or the controller which is received on the SCK and FS pin. The MCK can optionally be sourced externally on the MCK pin. The Serializers are enabled and will tranceive data on the SD pin. All data flow is controlled by the master.

3.3.4. Switch Modes

The mode switching between master, controller, and slave modes are actually done by modifying the source mode of I^2S pins. The source mode of I^2S pins are selected by writing corresponding bits in CLKCTRLn. Since source mode switching changes the direction of pin, the mode must be changed when the I^2S Clock Unit is stopped.

3.4. Data Stream Reception/Transmission

The I^2S module support several data stream formats:

- I²S format
- Time Division Multiplexed (TDM) format
- Pulse Density Modulation (PDM) format (reception only)

Basically the I²S module can send several words within each frame, it's more like TDM format. With adjust to the number of data words in a frame, the FS width, the FS to data bits delay, etc., the module is able to handle I²S compliant data stream.

Also the Serializer can receive PDM format data stream, which allows the I²S module receive 1 PDM data on each SCK edge.

3.4.1. I²S Stream Reception/Transmission

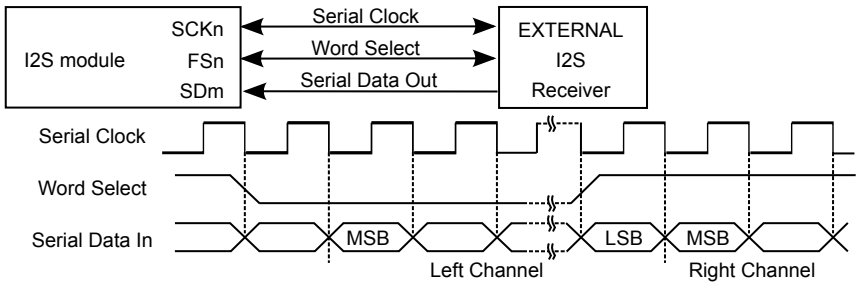
For 2-channel I²S compliant data stream format the I²S module uses the FS line as word select (WS) signal and will send left channel data word on low WS level and right channel data word on high WS level as specified in the I²S standard. The supported word sizes are 8-, 16-, 18-, 20-, 24-, and 32- bit.

Thus for I²S stream, the following settings should be applied to the module:

- Data starting delay after FS transition : one SCK period
- FS width : half of frame
- Data bits adjust in word : left-adjusted
- Bit transmitting order : MSB first

Following is an example for I²S application connections and waveforms. See the figure below.

Figure 3-2. I²S Example Diagram



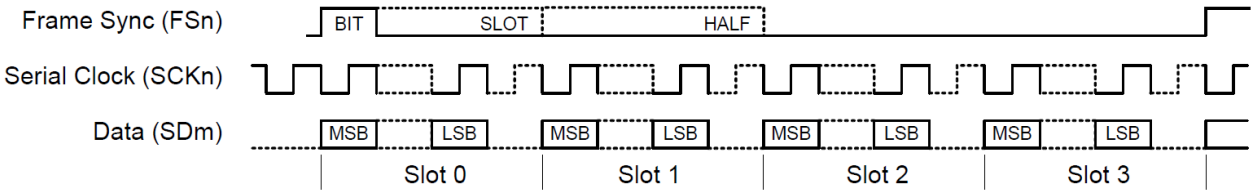
3.4.2. TDM Stream Reception/Transmission

In TDM format, the module sends several data words in each frame. For this data stream format most of the configurations could be adjusted:

- Main Frame related settings are as follow:
  - Frame Sync (FS) options:
    - The active edge of the FS (or if FS is inverted before use)
    - The width of the FS
  - The delay between FS to first data bit
- Data alignment in slot
- The number of slots and slot size can be adjusted, it has been mentioned in Audio Frame Generation
- The data word size is controlled by Serializer, it can be chosen among 8, 16, 18, 20, 24, and 32 bits.

The general TDM waveform generation is as follows:

Figure 3-3. TDM Waveform Generation

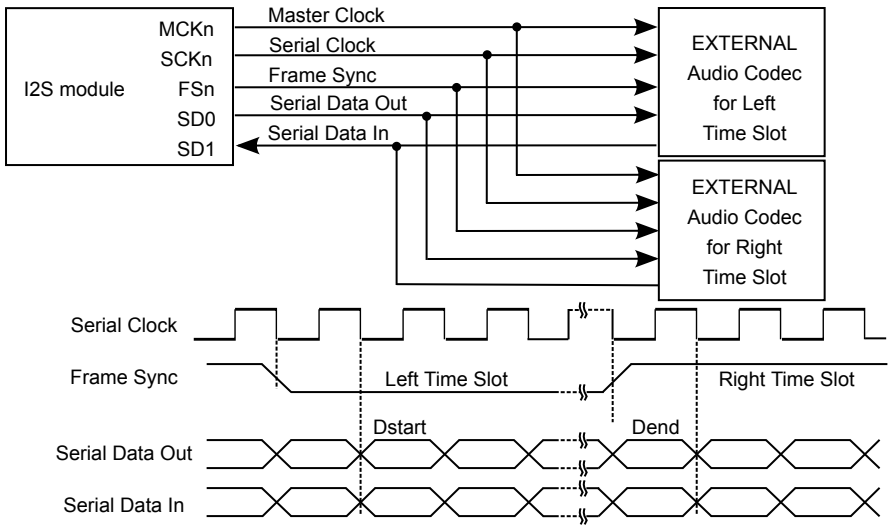


Some other settings could also be found to set up clock, data formatting and pin multiplexer (MUX). Refer to Clock Unit Configurations and Serializer Configurations for more details.

Following is examples for different application use cases.

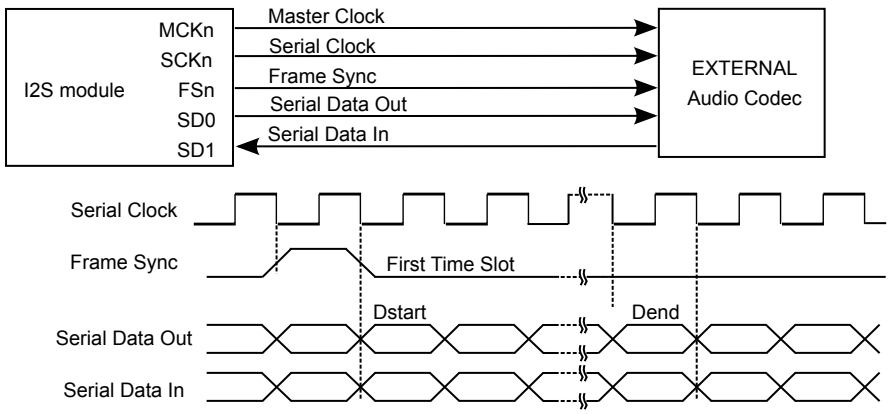
See Figure 3-4 Codec Example Diagram on page 10 for the Time Slot Application connection and waveform example.

Figure 3-4. Codec Example Diagram



See Figure 3-5 Time Slot Example Diagram on page 10 for the Codec Application connection and waveform example.

Figure 3-5. Time Slot Example Diagram



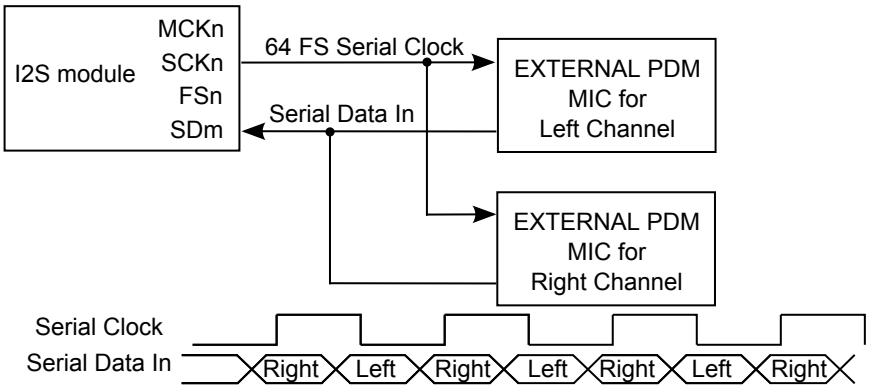
3.4.3. PDM Reception

The I²S Serializer integrates PDM reception feature, to use this feature, simply select PDM2 mode in Serializer configuration. In PDM2 mode, it assumes two microphones are input for stereo stream. The left

microphone bits will be stored in lower half and right microphone bits in upper half of the data word, like in compact stereo format.

See Figure 3-6 Time PDM2 Example Diagram on page 11 for an example of PDM Microphones Application with both left and right channel microphone connected.

Figure 3-6. Time PDM2 Example Diagram



3.4.4. MONO and Compact Data

The I²S Serializer can accept some pre-defined data format and generates the data stream in specified way.

When transmitting data, the Serializer can work in MONO mode: assum input is single channel mono data on left channel and copy it to right channel automatically.

Also the I²S Serializer can support compact stereo data word. The data word size of the Serializer can be set to 16-bit compact or 8-bit compact, with these option I²S Serializer will compact left channel data and right channel data together, the left channel data will take lower bytes and right channel data take higher bytes.

3.5. Loop-back Mode

The I²S can be configured to loop back the Transmitter to Receiver. In this mode Serializer's input will be connected to another Serializer's output internally.

3.6. Sleep Modes

The I²S will continue to operate in any sleep mode, where the selected source clocks are running.

4. Special Considerations

There is no special considerations for I^2S module.

5. Extra Information

For extra information see Extra Information for I2S Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

For a list of examples related to this driver, see Examples for I2S Driver.

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Type i2s_serializer_callback_t

```
typedef void(* i2s_serializer_callback_t )(struct i2s_module *const module)
```

Type of the callback functions.

7.2. Structure Definitions

7.2.1. Struct i2s_clock_config

Configure for I²S clock (SCK).

Table 7-1. Members

Type	Name	Description
enum gclk_generator	gclk_src	Clock source selection
uint8_t	mck_out_div	Divide generic clock to master clock output (1~32, 0,1 means no div)
bool	mck_out_enable	Generate MCK clock output
bool	mck_out_invert	Invert master clock output
enum i2s_master_clock_source	mck_src	Master clock source selection: generated or input from pin
uint8_t	sck_div	Divide generic clock to serial clock (1~32, 0,1 means no div)
bool	sck_out_invert	Invert serial clock output
enum i2s_serial_clock_source	sck_src	Serial clock source selection: generated or input from pin

7.2.2. Struct i2s_clock_unit_config

Configure for I²S clock unit.

Table 7-2. Members

Type	Name	Description
struct i2s_clock_config	clock	Configure clock generation
struct i2s_frame_config	frame	Configure frame generation

Type	Name	Description
struct i2s_pin_config	fs_pin	Configure frame sync pin
struct i2s_pin_config	mck_pin	Configure master clock pin
struct i2s_pin_config	sck_pin	Configure serial clock pin

7.2.3. Struct i2s_frame_config

Configure for I²S frame.

Table 7-3. Members

Type	Name	Description
enum i2s_data_delay	data_delay	Data delay from Frame Sync (FS) to first data bit
struct i2s_frame_sync_config	frame_sync	Frame sync (FS)
uint8_t	number_slots	Number of slots in a frame (1~8, 0,1 means minimum 1)
enum i2s_slot_size	slot_size	Size of each slot in frame

7.2.4. Struct i2s_frame_sync_config

Configure for I²S frame sync (FS).

Table 7-4. Members

Type	Name	Description
bool	invert_out	Invert Frame Sync (FS) signal before output
bool	invert_use	Invert Frame Sync (FS) signal before use
enum i2s_frame_sync_source	source	Frame Sync (FS) generated or input from pin
enum i2s_frame_sync_width	width	Frame Sync (FS) width

7.2.5. Struct i2s_module

Table 7-5. Members

Type	Name	Description
I2s *	hw	Module HW register access base
struct i2s_serializer_module	serializer[]	Module Serializer used

7.2.6. Struct i2s_pin_config

Configure for I²S pin.

Table 7-6. Members

Type	Name	Description
bool	enable	Enable this pin for I2S module
uint8_t	gpio	GPIO index to access the pin
uint8_t	mux	Pin function MUX

7.2.7. Struct i2s_serializer_config

Configure for I2S Serializer.

Table 7-7. Members

Type	Name	Description
enum i2s_bit_padding	bit_padding	Data Formatting Bit Extension
enum i2s_clock_unit	clock_unit	Clock unit selection
bool	data_adjust_left_in_slot	Data Slot Formatting Adjust, set to `true` to adjust words in slot to left
bool	data_adjust_left_in_word	Data Word Formatting Adjust, set to `true` to adjust bits in word to left
enum i2s_data_padding	data_padding	Data padding when under-run
struct i2s_pin_config	data_pin	Configure Serializer data pin
enum i2s_data_size	data_size	Data Word Size
bool	disable_data_slot[]	Disable data slot
enum i2s_dma_usage	dma_usage	DMA usage
enum i2s_line_default_state	line_default_state	Line default state where slot is disabled
bool	loop_back	Set to `true` to loop-back output to input pin for test
enum i2s_serializer_mode	mode	Serializer Mode
bool	mono_mode	Set to `true` to assumes mono input and duplicate it (left channel) to right channel
bool	transfer_lsb_first	Set to `true` to transfer LSB first, `false` to transfer MSB first

7.2.8. Struct i2s_serializer_module

Table 7-8. Members

Type	Name	Description
i2s_serializer_callback_t	callback[]	Callbacks list for Serializer
enum i2s_data_size	data_size	Serializer data word size
uint8_t	enabled_callback_mask	Callback mask for enabled callbacks
void *	job_buffer	Job buffer
enum status_code	job_status	Status of the ongoing or last transfer job
enum i2s_serializer_mode	mode	Serializer mode
uint8_t	registered_callback_mask	Callback mask for registered callbacks
uint32_t	requested_words	Requested data words to read/write
uint32_t	transferred_words	Transferred data words for read/write

7.3. Macro Definitions

7.3.1. Module Status Flags

I²S status flags, returned by i2s_get_status() and cleared by i2s_clear_status().

7.3.1.1. Macro I2S_STATUS_TRANSMIT_UNDERRUN

```
#define I2S_STATUS_TRANSMIT_UNDERRUN(x)
```

Module Serializer x (0~1) Transmit Underrun.

7.3.1.2. Macro I2S_STATUS_TRANSMIT_READY

```
#define I2S_STATUS_TRANSMIT_READY(x)
```

Module Serializer x (0~1) is ready to accept new data to be transmitted.

7.3.1.3. Macro I2S_STATUS_RECEIVE_OVERRUN

```
#define I2S_STATUS_RECEIVE_OVERRUN(x)
```

Module Serializer x (0~1) Receive Overrun.

7.3.1.4. Macro I2S_STATUS_RECEIVE_READY

```
#define I2S_STATUS_RECEIVE_READY(x)
```

Module Serializer x (0~1) has received a new data.

7.3.1.5. Macro I2S_STATUS_SYNC_BUSY

```
#define I2S_STATUS_SYNC_BUSY
```

Module is busy on synchronization.

7.4. Function Definitions

7.4.1. Driver Initialization

7.4.1.1. Function i2s_init()

Initializes a hardware I²S module instance.

```
enum status_code i2s_init(
        struct i2s_module *const module_inst,
        I2s * hw)
```

Enables the clock and initialize the I²S module.

Table 7-9. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	hw	Pointer to the TCC hardware module

Returns
Status of the initialization procedure.

Table 7-10. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_BUSY	Hardware module was busy when the initialization procedure was attempted
STATUS_ERR_DENIED	Hardware module was already enabled

7.4.2. Enable/Disable/Reset

7.4.2.1. Function i2s_enable()

Enable the I²S module.

```
void i2s_enable(
        const struct i2s_module *const module_inst)
```

Enables a I²S module that has been previously initialized.

Table 7-11. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

7.4.2.2. Function i2s_disable()

Disables the I²S module.

```
void i2s_disable(
        const struct i2s_module *const module_inst)
```

Disables a I²S module.

Table 7-12. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

7.4.2.3. Function i2s_reset()

Resets the I²S module.

```
void i2s_reset(
        const struct i2s_module *const module_inst)
```

Resets the I²S module, restoring all hardware module registers to their default values and disabling the module. The I²S module will not be accessible while the reset is being performed.

Table 7-13. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

7.4.3. Clock Unit Initialization and Configuration

7.4.3.1. Function i2s_clock_unit_get_config_defaults()

Initializes config with predefined default values for I²S clock unit.

```
void i2s_clock_unit_get_config_defaults(
        struct i2s_clock_unit_config *const config)
```

This function will initialize a given I²S Clock Unit configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- The clock unit does not generate output clocks (MCK, SCK, and FS)
- The pins (MCK, SCK, and FS) and MUX configurations are not set

Table 7-14. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to a I²S module clock unit configuration struct to set

7.4.3.2. Function i2s_clock_unit_set_config()

Configure specified I²S clock unit.

```
enum status_code i2s_clock_unit_set_config(
        struct i2s_module *const module_inst,
```

```
        const enum i2s_clock_unit clock_unit,
        const struct i2s_clock_unit_config * config)
```

Enables the clock and initialize the clock unit, based on the given configurations.

Table 7-15. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	clock_unit	I²S clock unit to initialize and configure
[in]	config	Pointer to the I²S clock unit configuration options struct

Returns

Status of the configuration procedure.

Table 7-16. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_BUSY	Hardware module was busy when the configuration procedure was attempted
STATUS_ERR_DENIED	Hardware module was already enabled
STATUS_ERR_INVALID_ARG	Invalid divider value or MCK direction setting conflict

7.4.4. Clock Unit Enable/Disable

7.4.4.1. Function i2s_clock_unit_enable()

Enable the Specified Clock Unit of I²S module.

```
void i2s_clock_unit_enable(
        const struct i2s_module *const module_inst,
        const enum i2s_clock_unit clock_unit)
```

Enables a Clock Unit in I²S module that has been previously initialized.

Table 7-17. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	clock_unit	I²S Clock Unit to enable

7.4.4.2. Function i2s_clock_unit_disable()

Disable the Specified Clock Unit of I²S module.

```
void i2s_clock_unit_disable(
        const struct i2s_module *const module_inst,
        const enum i2s_clock_unit clock_unit)
```

Disables a Clock Unit in I²S module that has been previously initialized.

Table 7-18. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	clock_unit	I²S Clock Unit to disable

7.4.5. Serializer Initialization and Configuration

7.4.5.1. Function i2s_serializer_get_config_defaults()

Initializes config with predefined default values for I²S Serializer.

```
void i2s_serializer_get_config_defaults(
        struct i2s_serializer_config *const config)
```

This function will initialize a given I²S Clock Unit configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- Output data does not internally loopback to input line
- Does not extend mono data (left channel) to right channel
- None of the data slot is disabled
- MSB of I²S data is transferred first
- In data word data is adjusted right
- In slot data word is adjusted left
- The data size is 16-bit width
- I²S will padd 0 to not defined bits
- I²S will padd 0 to not defined words
- I²S will use single DMA channel for all data channels
- I²S will use clock unit 0 to serve as clock
- The default data line state is 0, when there is no data
- I²S will transmit data to output line
- The data pin and MUX configuration are not set

Table 7-19. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to a I²S module Serializer configuration struct to set

7.4.5.2. Function i2s_serializer_set_config()

Configure specified I²S serializer.

```
enum status_code i2s_serializer_set_config(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const struct i2s_serializer_config * config)
```

Enables the clock and initialize the serializer, based on the given configurations.

Table 7-20. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	serializer	I^2S serializer to initialize and configure
[in]	config	Pointer to the I^2S serializer configuration options struct

Returns
Status of the configuration procedure.

Table 7-21. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_BUSY	Hardware module was busy when the configuration procedure was attempted
STATUS_ERR_DENIED	Hardware module was already enabled

7.4.6. Serializer Enable/Disable

7.4.6.1. Function i2s_serializer_enable()

Enable the Specified Serializer of I^2S module.

```
void i2s_serializer_enable(
        const struct i2s_module *const module_inst,
        const enum i2s_serializer serializer)
```

Enables a Serializer in I^2S module that has been previously initialized.

Table 7-22. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	I^2S Serializer to enable

7.4.6.2. Function i2s_serializer_disable()

Disable the Specified Serializer of I^2S module.

```
void i2s_serializer_disable(
        const struct i2s_module *const module_inst,
        const enum i2s_serializer serializer)
```

Disables a Serializer in I^2S module that has been previously initialized.

Table 7-23. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	I^2S Serializer to disable

7.4.7. Status Management

7.4.7.1. Function i2s_get_status()

Retrieves the current module status.

```
uint32_t i2s_get_status(
        const struct i2s_module *const module_inst)
```

Retrieves the status of the module, giving overall state information.

Table 7-24. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the I^2S software instance struct

Returns
Bitmask of `I2S_STATUS_*` flags.

Table 7-25. Return Values

Return value	Description
I2S_STATUS_SYNC_BUSY	Module is busy synchronization
I2S_STATUS_TRANSMIT_UNDERRUN(x)	Serializer x (0~1) is underrun
I2S_STATUS_TRANSMIT_READY(x)	Serializer x (0~1) is ready to transmit new data word
I2S_STATUS_RECEIVE_OVERRUN(x)	Serializer x (0~1) is overrun
I2S_STATUS_RECEIVE_READY(x)	Serializer x (0~1) has data ready to read

7.4.7.2. Function i2s_clear_status()

Clears a module status flags.

```
void i2s_clear_status(
        const struct i2s_module *const module_inst,
        uint32_t status)
```

Clears the given status flags of the module.

Table 7-26. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the I^2S software instance struct
[in]	status	Bitmask of `I2S_STATUS_*` flags to clear

7.4.7.3. Function i2s_enable_status_interrupt()

Enable interrupts on status set.

```
enum status_code i2s_enable_status_interrupt(
        struct i2s_module *const module_inst,
        uint32_t status)
```

Enable the given status interrupt request from the I²S module.

Table 7-27. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the I²S software instance struct
[in]	status	Status interrupts to enable

Returns
Status of enable procedure.

Table 7-28. Return Values

Return value	Description
STATUS_OK	Interrupt is enabled successfully
STATUS_ERR_INVALID_ARG	Status with no interrupt is passed

7.4.7.4. Function i2s_disable_status_interrupt()

Disable interrupts on status set.

```
void i2s_disable_status_interrupt(
        struct i2s_module *const module_inst,
        uint32_t status)
```

Disable the given status interrupt request from the I²S module.

Table 7-29. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the I²S software instance struct
[in]	status	Status interrupts to disable

7.4.8. Data Read/Write

7.4.8.1. Function i2s_serializer_write_wait()

Write a data word to the specified Serializer of I²S module.

```
void i2s_serializer_write_wait(
        const struct i2s_module *const module_inst,
        enum i2s_serializer serializer,
        uint32_t data)
```

Table 7-30. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The Serializer to write to
[in]	data	The data to write

7.4.8.2. Function i2s_serializer_read_wait()

Read a data word from the specified Serializer of I²S module.

```
uint32_t i2s_serializer_read_wait(
        const struct i2s_module *const module_inst,
        enum i2s_serializer serializer)
```

Table 7-31. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The Serializer to read

7.4.8.3. Function i2s_serializer_write_buffer_wait()

Write buffer to the specified Serializer of I²S module.

```
enum status_code i2s_serializer_write_buffer_wait(
        const struct i2s_module *const module_inst,
        enum i2s_serializer serializer,
        void * buffer,
        uint32_t size)
```

Table 7-32. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The serializer to write to
[in]	buffer	The data buffer to write
[in]	size	Number of data words to write

Returns

Status of the initialization procedure.

Table 7-33. Return Values

Return value	Description
STATUS_OK	The data was sent successfully
STATUS_ERR_DENIED	The module or serializer is disabled
STATUS_ERR_INVALID_ARG	An invalid buffer pointer was supplied

7.4.8.4. Function i2s_serializer_read_buffer_wait()

Read from the specified Serializer of I²S module to a buffer.

```
enum status_code i2s_serializer_read_buffer_wait(
        const struct i2s_module *const module_inst,
        enum i2s_serializer serializer,
        void * buffer,
        uint32_t size)
```

Table 7-34. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The serializer to write to
[in]	buffer	The buffer to fill read data (NULL to discard)
[in]	size	Number of data words to read

Returns
Status of the initialization procedure.

Table 7-35. Return Values

Return value	Description
STATUS_OK	The data was sent successfully
STATUS_ERR_DENIED	The module or serializer is disabled
STATUS_ERR_INVALID_ARG	An invalid buffer pointer was supplied

7.4.9. Callback Management

7.4.9.1. Function i2s_serializer_register_callback()

Registers a callback for serializer.

```
void i2s_serializer_register_callback(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const i2s_serializer_callback_t callback_func,
        const enum i2s_serializer_callback callback_type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled by for the interrupt handler to call it when the condition for the callback is met.

Table 7-36. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	serializer	The serializer that generates callback
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

7.4.9.2. Function i2s_serializer_unregister_callback()

Unregisters a callback for serializer.

```
void i2s_serializer_unregister_callback(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const enum i2s_serializer_callback callback_type)
```

Unregisters a callback function which is implemented by the user.

Table 7-37. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	serializer	The serializer that generates callback
[in]	callback_type	Callback type given by an enum

7.4.9.3. Function i2s_serializer_enable_callback()

Enables callback for serializer.

```
void i2s_serializer_enable_callback(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const enum i2s_serializer_callback callback_type)
```

Enables the callback function registered by i2s_serializer_register_callback. The callback function will be called from the interrupt handler when the conditions for the callback type are met.

Table 7-38. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	serializer	The serializer that generates callback
[in]	callback_type	Callback type given by an enum

7.4.9.4. Function i2s_serializer_disable_callback()

Disables callback for Serializer.

```
void i2s_serializer_disable_callback(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const enum i2s_serializer_callback callback_type)
```

Disables the callback function registered by the i2s_serializer_register_callback.

Table 7-39. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	serializer	The serializer that generates callback
[in]	callback_type	Callback type given by an enum

7.4.10. Job Management

7.4.10.1. Function i2s_serializer_write_buffer_job()

Write buffer to the specified Serializer of I^2S module.

```
enum status_code i2s_serializer_write_buffer_job(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const void * buffer,
        const uint32_t size)
```

Table 7-40. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The serializer to write to
[in]	buffer	The data buffer to write
[in]	size	Number of data words to write

Returns

Status of the initialization procedure.

Table 7-41. Return Values

Return value	Description
STATUS_OK	The data was sent successfully
STATUS_ERR_DENIED	The serializer is not in transmit mode
STATUS_ERR_INVALID_ARG	An invalid buffer pointer was supplied

7.4.10.2. Function i2s_serializer_read_buffer_job()

Read from the specified Serializer of I²S module to a buffer.

```
enum status_code i2s_serializer_read_buffer_job(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        void * buffer,
        const uint32_t size)
```

Table 7-42. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The serializer to write to
[out]	buffer	The buffer to fill read data
[in]	size	Number of data words to read

Returns

Status of the initialization procedure.

Table 7-43. Return Values

Return value	Description
STATUS_OK	The data was sent successfully
STATUS_ERR_DENIED	The serializer is not in receive mode
STATUS_ERR_INVALID_ARG	An invalid buffer pointer was supplied

7.4.10.3. Function i2s_serializer_abort_job()

Aborts an ongoing job running on serializer.

```
void i2s_serializer_abort_job(
        struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const enum i2s_job_type job_type)
```

Aborts an ongoing job.

Table 7-44. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The serializer which runs the job
[in]	job_type	Type of job to abort

7.4.10.4. Function i2s_serializer_get_job_status()

Gets the status of a job running on serializer.

```
enum status_code i2s_serializer_get_job_status(
        const struct i2s_module *const module_inst,
        const enum i2s_serializer serializer,
        const enum i2s_job_type job_type)
```

Gets the status of an ongoing or the last job.

Table 7-45. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	serializer	The serializer which runs the job
[in]	job_type	Type of job to abort

Returns
Status of the job.

7.4.11. Function i2s_is_syncing()

Determines if the hardware module(s) are currently synchronizing to the bus.

```
bool i2s_is_syncing(
        const struct i2s_module *const module_inst)
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus, This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Table 7-46. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Returns
Synchronization status of the underlying hardware module(s).

Table 7-47. Return Values

Return value	Description
false	If the module has completed synchronization
true	If the module synchronization is ongoing

7.5. Enumeration Definitions

7.5.1. Enum i2s_bit_order

I^2S data bit order.

Table 7-48. Members

Enum value	Description
I2S_BIT_ORDER_MSB_FIRST	Transfer Data Most Significant Bit first (Default for I²S protocol)
I2S_BIT_ORDER_LSB_FIRST	Transfer Data Least Significant Bit first

7.5.2. Enum i2s_bit_padding

I²S data bit padding.

Table 7-49. Members

Enum value	Description
I2S_BIT_PADDING_0	Padding with 0
I2S_BIT_PADDING_1	Padding with 1
I2S_BIT_PADDING_MSB	Padding with MSBit
I2S_BIT_PADDING_LSB	Padding with LSBit

7.5.3. Enum i2s_clock_unit

I²S clock unit selection.

Table 7-50. Members

Enum value	Description
I2S_CLOCK_UNIT_0	Clock Unit channel 0
I2S_CLOCK_UNIT_1	Clock Unit channel 1
I2S_CLOCK_UNIT_N	Number of Clock Unit channels

7.5.4. Enum i2s_data_adjust

I²S data word adjust.

Table 7-51. Members

Enum value	Description
I2S_DATA_ADJUST_RIGHT	Data is right adjusted in word
I2S_DATA_ADJUST_LEFT	Data is left adjusted in word

7.5.5. Enum i2s_data_delay

Data delay from Frame Sync (FS).

Table 7-52. Members

Enum value	Description
I2S_DATA_DELAY_0	Left Justified (no delay)
I2S_DATA_DELAY_1	I²S data delay (1-bit delay)
I2S_DATA_DELAY_LEFT_JUSTIFIED	Left Justified (no delay)
I2S_DATA_DELAY_I2S	I²S data delay (1-bit delay)

7.5.6. Enum i2s_data_format

I²S data format, to extend mono data to two channels.

Table 7-53. Members

Enum value	Description
I2S_DATA_FORMAT_STEREO	Normal mode, keep data to its right channel
I2S_DATA_FORMAT_MONO	Assume input is mono data for left channel, the data is duplicated to right channel

7.5.7. Enum i2s_data_padding

I²S data padding.

Table 7-54. Members

Enum value	Description
I2S_DATA_PADDING_0	Padding 0 in case of under-run
I2S_DATA_PADDING_SAME_AS_LAST	Padding last data in case of under-run
I2S_DATA_PADDING_LAST	Padding last data in case of under-run (abbr. `I2S_DATA_PADDING_SAME_AS_LAST`)
I2S_DATA_PADDING_SAME	Padding last data in case of under-run (abbr. `I2S_DATA_PADDING_SAME_AS_LAST`)

7.5.8. Enum i2s_data_size

I²S data word size.

Table 7-55. Members

Enum value	Description
I2S_DATA_SIZE_32BIT	32-bit
I2S_DATA_SIZE_24BIT	24-bit
I2S_DATA_SIZE_20BIT	20-bit
I2S_DATA_SIZE_18BIT	18-bit

Enum value	Description
I2S_DATA_SIZE_16BIT	16-bit
I2S_DATA_SIZE_16BIT_COMPACT	16-bit compact stereo
I2S_DATA_SIZE_8BIT	8-bit
I2S_DATA_SIZE_8BIT_COMPACT	8-bit compact stereo

7.5.9. Enum i2s_dma_usage

DMA channels usage for I^{2}S.

Table 7-56. Members

Enum value	Description
I2S_DMA_USE_SINGLE_CHANNEL_FOR_ALL	Single DMA channel for all I^{2}S channels
I2S_DMA_USE_ONE_CHANNEL_PER_DATA_CHANNEL	One DMA channel per data channel

7.5.10. Enum i2s_frame_sync_source

Frame Sync (FS) source.

Table 7-57. Members

Enum value	Description
I2S_FRAME_SYNC_SOURCE_SCKDIV	Frame Sync (FS) is divided from I^{2}S Serial Clock
I2S_FRAME_SYNC_SOURCE_FSPIN	Frame Sync (FS) is input from FS input pin

7.5.11. Enum i2s_frame_sync_width

Frame Sync (FS) output pulse width.

Table 7-58. Members

Enum value	Description
I2S_FRAME_SYNC_WIDTH_SLOT	Frame Sync (FS) Pulse is one slot width
I2S_FRAME_SYNC_WIDTH_HALF_FRAME	Frame Sync (FS) Pulse is half a frame width
I2S_FRAME_SYNC_WIDTH_BIT	Frame Sync (FS) Pulse is one bit width
I2S_FRAME_SYNC_WIDTH_BURST	1-bit wide Frame Sync (FS) per Data sample, only used when Data transfer is requested

7.5.12. Enum i2s_job_type

Enum for the possible types of I^{2}S asynchronous jobs that may be issued to the driver.

Table 7-59. Members

Enum value	Description
I2S_JOB_WRITE_BUFFER	Asynchronous I²S write from a user provided buffer
I2S_JOB_READ_BUFFER	Asynchronous I²S read into a user provided buffer

7.5.13. Enum i2s_line_default_state

I²S line default value when slot disabled.

Table 7-60. Members

Enum value	Description
I2S_LINE_DEFAULT_0	Output default value is 0
I2S_LINE_DEFAULT_1	Output default value is 1
I2S_LINE_DEFAULT_HIGH_IMPEDANCE	Output default value is high impedance
I2S_LINE_DEFAULT_HIZ	Output default value is high impedance (abbr. `I2S_LINE_DEFAULT_HIGH_IMPEDANCE`)

7.5.14. Enum i2s_master_clock_source

Master Clock (MCK) source selection.

Table 7-61. Members

Enum value	Description
I2S_MASTER_CLOCK_SOURCE_GCLK	Master Clock (MCK) is from general clock
I2S_MASTER_CLOCK_SOURCE_MCKPIN	Master Clock (MCK) is from MCK input pin

7.5.15. Enum i2s_serial_clock_source

Serial Clock (SCK) source selection.

Table 7-62. Members

Enum value	Description
I2S_SERIAL_CLOCK_SOURCE_MCKDIV	Serial Clock (SCK) is divided from Master Clock
I2S_SERIAL_CLOCK_SOURCE_SCKPIN	Serial Clock (SCK) is input from SCK input pin

7.5.16. Enum i2s_serializer

I²S Serializer selection.

Table 7-63. Members

Enum value	Description
I2S_SERIALIZER_0	Serializer channel 0
I2S_SERIALIZER_1	Serializer channel 1
I2S_SERIALIZER_N	Number of Serializer channels

7.5.17. Enum i2s_serializer_callback

Table 7-64. Members

Enum value	Description
I2S_SERIALIZER_CALLBACK_BUFFER_DONE	Callback for buffer read/write finished
I2S_SERIALIZER_CALLBACK_OVER_UNDER_RUN	Callback for Serializer overrun/underrun

7.5.18. Enum i2s_serializer_mode

I^2S Serializer mode.

Table 7-65. Members

Enum value	Description
I2S_SERIALIZER_RECEIVE	Serializer is used to receive data
I2S_SERIALIZER_TRANSMIT	Serializer is used to transmit data
I2S_SERIALIZER_PDM2	Serializer is used to receive PDM data on each clock edge

7.5.19. Enum i2s_slot_adjust

I^2S data slot adjust.

Table 7-66. Members

Enum value	Description
I2S_SLOT_ADJUST_RIGHT	Data is right adjusted in slot
I2S_SLOT_ADJUST_LEFT	Data is left adjusted in slot

7.5.20. Enum i2s_slot_size

Time Slot Size in number of I^2S serial clocks (bits).

Table 7-67. Members

Enum value	Description
I2S_SLOT_SIZE_8_BIT	8-bit slot
I2S_SLOT_SIZE_16_BIT	16-bit slot

Enum value	Description
I2S_SLOT_SIZE_24_BIT	24-bit slot
I2S_SLOT_SIZE_32_BIT	32-bit slot

8. Extra Information for I²S Driver

8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
I²S, IIS	Inter-IC Sound Controller
MCK	Master Clock
SCK	Serial Clock
FS	Frame Sync
SD	Serial Data
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
TDM	Time Division Multiplexed
PDM	Pulse Density Modulation
LSB	Least Significant Bit
MSB	Most Significant Bit
DSP	Digital Signal Processor

8.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

8.3. Errata

There are no errata related to this driver.

8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

9. Examples for I²S Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Inter-IC Sound Controller (I2S) Driver. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that QSGs can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for I2S - Basic
- Quick Start Guide for I2S - Callback
- Quick Start Guide for I2S - DMA

9.1. Quick Start Guide for I²S - Basic

In this use case, the I²S will be used to generate Master Clock (MCK), Serial Clock (SCK), Frame Sync (FS), and Serial Data (SD) signals.

Here MCK is set to the half of processor clock. SCK is set to a quarter of the frequency of processor. FS generates half-half square wave for left and right audio channel data. The output serial data of channels toggle from two values to generate square wave, if codec or DAC is connected.

The I²S module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- MCK, SCK, and FS clocks outputs are enabled
- MCK output divider set to 2
- SCK generation divider set to 4
- Each frame will contain two 32-bit slots
- Data will be left adjusted and start transmit without delay

9.1.1. Quick Start

9.1.1.1. Prerequisites

There are no prerequisites for this use case.

9.1.1.2. Code

Add to the main application source file, before any functions:

```
#define CONF_I2S_MODULE      I2S

#define CONF_I2S_MCK_PIN     PIN_PA09G_I2S_MCK0

#define CONF_I2S_MCK_MUX     MUX_PA09G_I2S_MCK0

#define CONF_I2S_SCK_PIN     PIN_PA10G_I2S_SCK0

#define CONF_I2S_SCK_MUX     MUX_PA10G_I2S_SCK0

#define CONF_I2S_FS_PIN      PIN_PA11G_I2S_FS0

#define CONF_I2S_FS_MUX      MUX_PA11G_I2S_FS0

#define CONF_I2S_SD_PIN      PIN_PA07G_I2S_SD0

#define CONF_I2S_SD_MUX      MUX_PA07G_I2S_SD0
```

Add to the main application source file, outside of any functions:

```
struct i2s_module i2s_instance;
```

Copy-paste the following setup code to your user application:

```
static void _configure_i2s(void)
{
    i2s_init(&i2s_instance, CONF_I2S_MODULE);

    struct i2s_clock_unit_config config_clock_unit;
    i2s_clock_unit_get_config_defaults(&config_clock_unit);

    config_clock_unit.clock.gclk_src = GCLK_GENERATOR_0;

    config_clock_unit.clock.mck_src = I2S_MASTER_CLOCK_SOURCE_GCLK;
    config_clock_unit.clock.mck_out_enable = true;
    config_clock_unit.clock.mck_out_div = 2;

    config_clock_unit.clock.sck_src = I2S_SERIAL_CLOCK_SOURCE_MCKDIV;
    config_clock_unit.clock.sck_div = 4;

    config_clock_unit.frame.number_slots = 2;
    config_clock_unit.frame.slot_size = I2S_SLOT_SIZE_32_BIT;
    config_clock_unit.frame.data_delay = I2S_DATA_DELAY_0;

    config_clock_unit.frame.frame_sync.source =
I2S_FRAME_SYNC_SOURCE_SCKDIV;
    config_clock_unit.frame.frame_sync.width =
I2S_FRAME_SYNC_WIDTH_HALF_FRAME;

    config_clock_unit.mck_pin.enable = true;
    config_clock_unit.mck_pin.gpio = CONF_I2S_MCK_PIN;
    config_clock_unit.mck_pin.mux = CONF_I2S_MCK_MUX;

    config_clock_unit.sck_pin.enable = true;
    config_clock_unit.sck_pin.gpio = CONF_I2S_SCK_PIN;
    config_clock_unit.sck_pin.mux = CONF_I2S_SCK_MUX;

    config_clock_unit.fs_pin.enable = true;
    config_clock_unit.fs_pin.gpio = CONF_I2S_FS_PIN;
    config_clock_unit.fs_pin.mux = CONF_I2S_FS_MUX;

    i2s_clock_unit_set_config(&i2s_instance, I2S_CLOCK_UNIT_0,
            &config_clock_unit);

    struct i2s_serializer_config config_serializer;
    i2s_serializer_get_config_defaults(&config_serializer);

    config_serializer.clock_unit = I2S_CLOCK_UNIT_0;
    config_serializer.mode = I2S_SERIALIZER_TRANSMIT;
    config_serializer.data_size = I2S_DATA_SIZE_16BIT;

    config_serializer.data_pin.enable = true;
    config_serializer.data_pin.gpio = CONF_I2S_SD_PIN;
    config_serializer.data_pin.mux = CONF_I2S_SD_MUX;

    i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_0,
            &config_serializer);

    i2s_enable(&i2s_instance);
    i2s_clock_unit_enable(&i2s_instance, I2S_CLOCK_UNIT_0);
```

```
    i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_0);
}
```

Add to user application initialization (typically the start of `main()`):

```
_configure_i2s();
```

9.1.1.3. Workflow

1. Create a module software instance structure for the I²S module to store the I²S driver state while it is in use.

   ```
   struct i2s_module i2s_instance;
   ```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.
2. Configure the I²S module.
   1. Initialize the I²S module.

      ```
      i2s_init(&i2s_instance, CONF_I2S_MODULE);
      ```
   2. Initialize the I²S Clock Unit.
      1. Create a I²S Clock Unit configuration struct, which can be filled out to adjust the configuration of a physical I²S Clock Unit.

         ```
         struct i2s_clock_unit_config config_clock_unit;
         ```
      2. Initialize the I²S Clock Unit configuration struct with the module's default values.

         ```
         i2s_clock_unit_get_config_defaults(&config_clock_unit);
         ```

         Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
      3. Alter the I²S Clock Unit settings to configure the general clock source, MCK, SCK, and FS generation.

         ```
         config_clock_unit.clock.gclk_src = GCLK_GENERATOR_0;

         config_clock_unit.clock.mck_src =
         I2S_MASTER_CLOCK_SOURCE_GCLK;
         config_clock_unit.clock.mck_out_enable = true;
         config_clock_unit.clock.mck_out_div = 2;

         config_clock_unit.clock.sck_src =
         I2S_SERIAL_CLOCK_SOURCE_MCKDIV;
         config_clock_unit.clock.sck_div = 4;

         config_clock_unit.frame.number_slots = 2;
         config_clock_unit.frame.slot_size = I2S_SLOT_SIZE_32_BIT;
         config_clock_unit.frame.data_delay = I2S_DATA_DELAY_0;

         config_clock_unit.frame.frame_sync.source =
         I2S_FRAME_SYNC_SOURCE_SCKDIV;
         config_clock_unit.frame.frame_sync.width =
         I2S_FRAME_SYNC_WIDTH_HALF_FRAME;
         ```
      4. Alter the I²S Clock Unit settings to configure the MCK, SCK, and FS output on physical device pins.

         ```
         config_clock_unit.mck_pin.enable = true;
         config_clock_unit.mck_pin.gpio = CONF_I2S_MCK_PIN;
         ```

```
config_clock_unit.mck_pin.mux = CONF_I2S_MCK_MUX;

config_clock_unit.sck_pin.enable = true;
config_clock_unit.sck_pin.gpio = CONF_I2S_SCK_PIN;
config_clock_unit.sck_pin.mux = CONF_I2S_SCK_MUX;

config_clock_unit.fs_pin.enable = true;
config_clock_unit.fs_pin.gpio = CONF_I2S_FS_PIN;
config_clock_unit.fs_pin.mux = CONF_I2S_FS_MUX;
```

5. Configure the I²S Clock Unit with the desired settings.

```
i2s_clock_unit_set_config(&i2s_instance, I2S_CLOCK_UNIT_0,
        &config_clock_unit);
```

3. Initialize the I²S Serializer.
   1. Create a I²S Serializer configuration struct, which can be filled out to adjust the configuration of a physical I²S Serializer.

```
struct i2s_serializer_config config_serializer;
```

   2. Initialize the I²S Serializer configuration struct with the module's default values.

```
i2s_serializer_get_config_defaults(&config_serializer);
```

      Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
   3. Alter the I²S Serializer settings to configure the serial data generation.

```
config_serializer.clock_unit = I2S_CLOCK_UNIT_0;
config_serializer.mode = I2S_SERIALIZER_TRANSMIT;
config_serializer.data_size = I2S_DATA_SIZE_16BIT;
```

   4. Alter the I²S Serializer settings to configure the SD on a physical device pin.

```
config_serializer.data_pin.enable = true;
config_serializer.data_pin.gpio = CONF_I2S_SD_PIN;
config_serializer.data_pin.mux = CONF_I2S_SD_MUX;
```

   5. Configure the I²S Serializer with the desired settings.

```
i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_0,
        &config_serializer);
```

4. Enable the I²S module, the Clock Unit, and Serializer to start the clocks and ready to transmit data.

```
i2s_enable(&i2s_instance);
i2s_clock_unit_enable(&i2s_instance, I2S_CLOCK_UNIT_0);
i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_0);
```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
while (true) {
    /* Infinite loop */
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0xF87F);
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0x901F);
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0);
```

```
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0);
}
```

9.1.2.2. Workflow

1. Enter an infinite loop to output data sequence via the I²S Serializer.

```
while (true) {
    /* Infinite loop */
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0xF87F);
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0x901F);
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0);
    i2s_serializer_write_wait(&i2s_instance, I2S_SERIALIZER_0, 0);
}
```

9.2. Quick Start Guide for I²S - Callback

In this use case, the I²S will be used to generate Master Clock (MCK), Serial Clock (SCK), Frame Sync (FS), and Serial Data (SD) signals.

Here MCK is set to the half of processor clock. SCK is set to a quarter of the frequency of processor. FS generates half-half square wave for left and right audio channel data. The output serial data of channels toggle from two values to generate square wave, if codec or DAC is connected.

The I²S module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- MCK, SCK, and FS clocks outputs are enabled
- MCK output divider set to 2
- SCK generation divider set to 4
- Each frame will contain two 32-bit slots
- Data will be left adjusted and start transmit without delay

9.2.1. Quick Start

9.2.1.1. Prerequisites

There are no prerequisites for this use case.

9.2.1.2. Code

Add to the main application source file, before any functions:

```
#define CONF_I2S_MODULE      I2S

#define CONF_I2S_MCK_PIN     PIN_PA09G_I2S_MCK0

#define CONF_I2S_MCK_MUX     MUX_PA09G_I2S_MCK0

#define CONF_I2S_SCK_PIN     PIN_PA10G_I2S_SCK0

#define CONF_I2S_SCK_MUX     MUX_PA10G_I2S_SCK0

#define CONF_I2S_FS_PIN      PIN_PA11G_I2S_FS0

#define CONF_I2S_FS_MUX      MUX_PA11G_I2S_FS0

#define CONF_I2S_SD_PIN      PIN_PA07G_I2S_SD0
```

```
#define CONF_I2S_SD_MUX      MUX_PA07G_I2S_SD0
```

Add to the main application source file, outside of any functions:

```
struct i2s_module i2s_instance;
```

Copy-paste the following data buffer code to your user application:

```
uint16_t data_buffer[4] = {0xF87F, 0x901F, 0, 0};
```

Copy-paste the following callback function code to your user application:

```
static void _i2s_callback_to_send_buffer(
        struct i2s_module *const module_inst)
{
    i2s_serializer_write_buffer_job(module_inst,
            I2S_SERIALIZER_0, data_buffer, 4);
}
```

Copy-paste the following setup code to your user application:

```
static void _configure_i2s(void)
{
    i2s_init(&i2s_instance, CONF_I2S_MODULE);

    struct i2s_clock_unit_config config_clock_unit;
    i2s_clock_unit_get_config_defaults(&config_clock_unit);

    config_clock_unit.clock.gclk_src = GCLK_GENERATOR_0;

    config_clock_unit.clock.mck_src = I2S_MASTER_CLOCK_SOURCE_GCLK;
    config_clock_unit.clock.mck_out_enable = true;
    config_clock_unit.clock.mck_out_div = 2;

    config_clock_unit.clock.sck_src = I2S_SERIAL_CLOCK_SOURCE_MCKDIV;
    config_clock_unit.clock.sck_div = 4;

    config_clock_unit.frame.number_slots = 2;
    config_clock_unit.frame.slot_size = I2S_SLOT_SIZE_32_BIT;
    config_clock_unit.frame.data_delay = I2S_DATA_DELAY_0;

    config_clock_unit.frame.frame_sync.source =
I2S_FRAME_SYNC_SOURCE_SCKDIV;
    config_clock_unit.frame.frame_sync.width =
I2S_FRAME_SYNC_WIDTH_HALF_FRAME;

    config_clock_unit.mck_pin.enable = true;
    config_clock_unit.mck_pin.gpio = CONF_I2S_MCK_PIN;
    config_clock_unit.mck_pin.mux = CONF_I2S_MCK_MUX;

    config_clock_unit.sck_pin.enable = true;
    config_clock_unit.sck_pin.gpio = CONF_I2S_SCK_PIN;
    config_clock_unit.sck_pin.mux = CONF_I2S_SCK_MUX;

    config_clock_unit.fs_pin.enable = true;
    config_clock_unit.fs_pin.gpio = CONF_I2S_FS_PIN;
    config_clock_unit.fs_pin.mux = CONF_I2S_FS_MUX;

    i2s_clock_unit_set_config(&i2s_instance, I2S_CLOCK_UNIT_0,
            &config_clock_unit);

    struct i2s_serializer_config config_serializer;
```

```
    i2s_serializer_get_config_defaults(&config_serializer);

    config_serializer.clock_unit = I2S_CLOCK_UNIT_0;
    config_serializer.mode = I2S_SERIALIZER_TRANSMIT;
    config_serializer.data_size = I2S_DATA_SIZE_16BIT;

    config_serializer.data_pin.enable = true;
    config_serializer.data_pin.gpio = CONF_I2S_SD_PIN;
    config_serializer.data_pin.mux = CONF_I2S_SD_MUX;

    i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_0,
            &config_serializer);

    i2s_enable(&i2s_instance);
    i2s_clock_unit_enable(&i2s_instance, I2S_CLOCK_UNIT_0);
    i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_0);
}

static void _configure_i2s_callbacks(void)
{
    i2s_serializer_register_callback(
            &i2s_instance,
            I2S_SERIALIZER_0,
            _i2s_callback_to_send_buffer,
            I2S_SERIALIZER_CALLBACK_BUFFER_DONE);

    i2s_serializer_enable_callback(&i2s_instance,
            I2S_SERIALIZER_0,
            I2S_SERIALIZER_CALLBACK_BUFFER_DONE);
}
```

Add to user application initialization (typically the start of `main()`):

```
_configure_i2s();
_configure_i2s_callbacks();
```

Add to user application start transmitting job (typically in `main()`, after initialization):

```
i2s_serializer_write_buffer_job(&i2s_instance,
        I2S_SERIALIZER_0, data_buffer, 4);
```

9.2.1.3. Workflow

1. Create a module software instance structure for the I²S module to store the I²S driver state while it is in use.

   ```
   struct i2s_module i2s_instance;
   ```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.
2. Configure the I²S module.
   1. Initialize the I²S module.

      ```
      i2s_init(&i2s_instance, CONF_I2S_MODULE);
      ```
   2. Initialize the I²S Clock Unit.
      1. Create a I²S module configuration struct, which can be filled out to adjust the configuration of a physical I²S Clock Unit.

         ```
         struct i2s_clock_unit_config config_clock_unit;
         ```

2. Initialize the I²S Clock Unit configuration struct with the module's default values.

```
i2s_clock_unit_get_config_defaults(&config_clock_unit);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the I²S Clock Unit settings to configure the general clock source, MCK, SCK, and FS generation.

```
config_clock_unit.clock.gclk_src = GCLK_GENERATOR_0;

config_clock_unit.clock.mck_src =
I2S_MASTER_CLOCK_SOURCE_GCLK;
config_clock_unit.clock.mck_out_enable = true;
config_clock_unit.clock.mck_out_div = 2;

config_clock_unit.clock.sck_src =
I2S_SERIAL_CLOCK_SOURCE_MCKDIV;
config_clock_unit.clock.sck_div = 4;

config_clock_unit.frame.number_slots = 2;
config_clock_unit.frame.slot_size = I2S_SLOT_SIZE_32_BIT;
config_clock_unit.frame.data_delay = I2S_DATA_DELAY_0;

config_clock_unit.frame.frame_sync.source =
I2S_FRAME_SYNC_SOURCE_SCKDIV;
config_clock_unit.frame.frame_sync.width =
I2S_FRAME_SYNC_WIDTH_HALF_FRAME;
```

4. Alter the I²S Clock Unit settings to configure the MCK, SCK, and FS output on physical device pins.

```
config_clock_unit.mck_pin.enable = true;
config_clock_unit.mck_pin.gpio = CONF_I2S_MCK_PIN;
config_clock_unit.mck_pin.mux = CONF_I2S_MCK_MUX;

config_clock_unit.sck_pin.enable = true;
config_clock_unit.sck_pin.gpio = CONF_I2S_SCK_PIN;
config_clock_unit.sck_pin.mux = CONF_I2S_SCK_MUX;

config_clock_unit.fs_pin.enable = true;
config_clock_unit.fs_pin.gpio = CONF_I2S_FS_PIN;
config_clock_unit.fs_pin.mux = CONF_I2S_FS_MUX;
```

5. Configure the I²S Clock Unit with the desired settings.

```
i2s_clock_unit_set_config(&i2s_instance, I2S_CLOCK_UNIT_0,
        &config_clock_unit);
```

3. Initialize the I²S Serializer.
    1. Create a I²S Serializer configuration struct, which can be filled out to adjust the configuration of a physical I²S Serializer.

```
struct i2s_serializer_config config_serializer;
```

    2. Initialize the I²S Serializer configuration struct with the module's default values.

```
i2s_serializer_get_config_defaults(&config_serializer);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the I²S Serializer settings to configure the serial data generation.

```
config_serializer.clock_unit = I2S_CLOCK_UNIT_0;
config_serializer.mode = I2S_SERIALIZER_TRANSMIT;
config_serializer.data_size = I2S_DATA_SIZE_16BIT;
```

4. Alter the I²S Serializer settings to configure the SD on a physical device pin.

```
config_serializer.data_pin.enable = true;
config_serializer.data_pin.gpio = CONF_I2S_SD_PIN;
config_serializer.data_pin.mux = CONF_I2S_SD_MUX;
```

5. Configure the I²S Serializer with the desired settings.

```
i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_0,
        &config_serializer);
```

4. Enable the I²S module, the Clock Unit, and Serializer to start the clocks and ready to transmit data.

```
i2s_enable(&i2s_instance);
i2s_clock_unit_enable(&i2s_instance, I2S_CLOCK_UNIT_0);
i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_0);
```

3. Configure the I²S callbacks.
   1. Register the Serializer 0 TX ready callback function with the driver.

```
i2s_serializer_register_callback(
        &i2s_instance,
        I2S_SERIALIZER_0,
        _i2s_callback_to_send_buffer,
        I2S_SERIALIZER_CALLBACK_BUFFER_DONE);
```

   2. Enable the Serializer 0 TX ready callback so that it will be called by the driver when appropriate.

```
i2s_serializer_enable_callback(&i2s_instance,
        I2S_SERIALIZER_0,
        I2S_SERIALIZER_CALLBACK_BUFFER_DONE);
```

4. Start a transmitting job.

```
i2s_serializer_write_buffer_job(&i2s_instance,
        I2S_SERIALIZER_0, data_buffer, 4);
```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

i2s_serializer_write_buffer_job(&i2s_instance,
        I2S_SERIALIZER_0, data_buffer, 4);

while (true) {
}
```

9.2.2.2. Workflow

1. Enter an infinite loop while the output is generated via the I²S module.

```
while (true) {
}
```

9.3. Quick Start Guide for I²S - DMA

In this use case, the I²S will be used to generate Master Clock (MCK), Serial Clock (SCK), Frame Sync (FS), and Serial Data (SD) signals.

Here MCK is set to the half of processor clock. SCK is set to a quarter of the frequency of processor. FS generates half-half square wave for left and right audio channel data. The output serial data of channels toggle from two values to generate square wave, if codec or DAC is connected.

The output SD is also fed back to another I²S channel by internal loop back, and transfer to values buffer by DMA.

The I²S module will be setup as follows:

- GCLK generator 0 (GCLK main) clock source
- MCK, SCK, and FS clocks outputs are enabled
- MCK output divider set to 2
- SCK generation divider set to 4
- Each frame will contain two 32-bit slots
- Data will be left adjusted and start transmit without delay

9.3.1. Quick Start

9.3.1.1. Prerequisites

There are no prerequisites for this use case.

9.3.1.2. Code

Add to the main application source file, before any functions:

```
#define CONF_I2S_MODULE      I2S

#define CONF_I2S_MCK_PIN     PIN_PA09G_I2S_MCK0

#define CONF_I2S_MCK_MUX     MUX_PA09G_I2S_MCK0

#define CONF_I2S_SCK_PIN     PIN_PA10G_I2S_SCK0

#define CONF_I2S_SCK_MUX     MUX_PA10G_I2S_SCK0

#define CONF_I2S_FS_PIN      PIN_PA11G_I2S_FS0

#define CONF_I2S_FS_MUX      MUX_PA11G_I2S_FS0

#define CONF_I2S_SD_PIN      PIN_PA07G_I2S_SD0
```

```
#define CONF_I2S_SD_MUX      MUX_PA07G_I2S_SD0
```

```
#define CONF_RX_TRIGGER 0x2A
```

```
#define CONF_TX_TRIGGER 0x2B
```

Add to the main application source file, outside of any functions:

```
struct i2s_module i2s_instance;
```

```
uint16_t rx_values[4] = {0xEEEE, 0xEEEE, 0xEEEE, 0xEEEE};
struct dma_resource rx_dma_resource;
COMPILER_ALIGNED(16) DmacDescriptor rx_dma_descriptor;
```

```
uint16_t tx_values[4] = {0xF87F, 0x901F, 0, 0};
struct dma_resource tx_dma_resource;
COMPILER_ALIGNED(16) DmacDescriptor tx_dma_descriptor;
```

Copy-paste the following setup code to your user application:

```
static void _config_dma_for_rx(void)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
    config.peripheral_trigger = CONF_RX_TRIGGER;

    dma_allocate(&rx_dma_resource, &config);

    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = 4;
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.step_selection = DMA_STEPSEL_SRC;
    descriptor_config.src_increment_enable = false;
    descriptor_config.destination_address =
            (uint32_t)rx_values + sizeof(rx_values);
    descriptor_config.source_address = (uint32_t)&CONF_I2S_MODULE->DATA[1];

    dma_descriptor_create(&rx_dma_descriptor, &descriptor_config);

    rx_dma_descriptor.DESCADDR.reg = (uint32_t)&rx_dma_descriptor;

    dma_add_descriptor(&rx_dma_resource, &rx_dma_descriptor);
    dma_start_transfer_job(&rx_dma_resource);
}
```

```
static void _config_dma_for_tx(void)
{
    struct dma_resource_config config;
    dma_get_config_defaults(&config);
    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
    config.peripheral_trigger = CONF_TX_TRIGGER;
    dma_allocate(&tx_dma_resource, &config);
```

```
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = 4;
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.source_address =
            (uint32_t)tx_values + sizeof(tx_values);
    descriptor_config.destination_address = (uint32_t)&CONF_I2S_MODULE-
>DATA[0];

    dma_descriptor_create(&tx_dma_descriptor, &descriptor_config);

    tx_dma_descriptor.DESCADDR.reg = (uint32_t)&tx_dma_descriptor;

    dma_add_descriptor(&tx_dma_resource, &tx_dma_descriptor);
    dma_start_transfer_job(&tx_dma_resource);
}
```

```
static void _configure_i2s(void)
{
    i2s_init(&i2s_instance, CONF_I2S_MODULE);

    struct i2s_clock_unit_config config_clock_unit;
    i2s_clock_unit_get_config_defaults(&config_clock_unit);

    config_clock_unit.clock.gclk_src = GCLK_GENERATOR_0;

    config_clock_unit.clock.mck_src = I2S_MASTER_CLOCK_SOURCE_GCLK;
    config_clock_unit.clock.mck_out_enable = true;
    config_clock_unit.clock.mck_out_div = 2;

    config_clock_unit.clock.sck_src = I2S_SERIAL_CLOCK_SOURCE_MCKDIV;
    config_clock_unit.clock.sck_div = 4;

    config_clock_unit.frame.number_slots = 2;
    config_clock_unit.frame.slot_size = I2S_SLOT_SIZE_32_BIT;
    config_clock_unit.frame.data_delay = I2S_DATA_DELAY_0;

    config_clock_unit.frame.frame_sync.source =
I2S_FRAME_SYNC_SOURCE_SCKDIV;
    config_clock_unit.frame.frame_sync.width =
I2S_FRAME_SYNC_WIDTH_HALF_FRAME;

    config_clock_unit.mck_pin.enable = true;
    config_clock_unit.mck_pin.gpio = CONF_I2S_MCK_PIN;
    config_clock_unit.mck_pin.mux = CONF_I2S_MCK_MUX;

    config_clock_unit.sck_pin.enable = true;
    config_clock_unit.sck_pin.gpio = CONF_I2S_SCK_PIN;
    config_clock_unit.sck_pin.mux = CONF_I2S_SCK_MUX;

    config_clock_unit.fs_pin.enable = true;
    config_clock_unit.fs_pin.gpio = CONF_I2S_FS_PIN;
    config_clock_unit.fs_pin.mux = CONF_I2S_FS_MUX;

    i2s_clock_unit_set_config(&i2s_instance, I2S_CLOCK_UNIT_0,
            &config_clock_unit);

    struct i2s_serializer_config config_serializer;
    i2s_serializer_get_config_defaults(&config_serializer);
```

```
    config_serializer.clock_unit = I2S_CLOCK_UNIT_0;
    config_serializer.mode = I2S_SERIALIZER_TRANSMIT;
    config_serializer.data_size = I2S_DATA_SIZE_16BIT;

    config_serializer.data_pin.enable = true;
    config_serializer.data_pin.gpio = CONF_I2S_SD_PIN;
    config_serializer.data_pin.mux = CONF_I2S_SD_MUX;

    i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_0,
            &config_serializer);

    config_serializer.loop_back = true;
    config_serializer.mode = I2S_SERIALIZER_RECEIVE;
    config_serializer.data_size = I2S_DATA_SIZE_16BIT;

    config_serializer.data_pin.enable = false;

    i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_1,
            &config_serializer);

    i2s_enable(&i2s_instance);
    i2s_clock_unit_enable(&i2s_instance, I2S_CLOCK_UNIT_0);
    i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_1);
    i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_0);
}
```

Add to user application initialization (typically the start of `main()`):

```
_config_dma_for_rx();
_config_dma_for_tx();
_configure_i2s();
```

9.3.1.3. Workflow

Configure the DMAC module to obtain received value from I²S Serializer 1.

1. Allocate and configure the DMA resource
    1. Create a DMA resource instance.

        ```
        struct dma_resource rx_dma_resource;
        ```

        Note: This should never go out of scope as long as the resource is in use. In most cases, this should be global.
    2. Create a DMA resource configuration struct.

        ```
        struct dma_resource_config config;
        ```
    3. Initialize the DMA resource configuration struct with default values.

        ```
        dma_get_config_defaults(&config);
        ```

        Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
    4. Adjust the DMA resource configurations.

        ```
        config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
        config.peripheral_trigger = CONF_RX_TRIGGER;
        ```
    5. Allocate a DMA resource with the configurations.

        ```
        dma_allocate(&rx_dma_resource, &config);
        ```

2. Prepare DMA transfer descriptor
    1. Create a DMA transfer descriptor.

```
COMPILER_ALIGNED(16) DmacDescriptor rx_dma_descriptor;
```

Note: When multiple descriptors are linked. The linked item should never go out of scope before it's loaded (to DMA Write-Back memory section). In most cases, if more than one descriptors are used, they should be global except the very first one.

    2. Create a DMA transfer descriptor configuration struct, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

    3. Initialize the DMA transfer descriptor configuration struct with default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

    4. Adjust the DMA transfer descriptor configurations.

```
descriptor_config.block_transfer_count = 4;
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.step_selection = DMA_STEPSEL_SRC;
descriptor_config.src_increment_enable = false;
descriptor_config.destination_address =
        (uint32_t)rx_values + sizeof(rx_values);
descriptor_config.source_address = (uint32_t)&CONF_I2S_MODULE-
>DATA[1];
```

    5. Create the DMA transfer descriptor with configuration.

```
dma_descriptor_create(&rx_dma_descriptor, &descriptor_config);
```

    6. Adjust the DMA transfer descriptor if multiple DMA transfer will be performed.

```
rx_dma_descriptor.DESCADDR.reg = (uint32_t)&rx_dma_descriptor;
```

3. Start DMA transfer job with prepared descriptor
    1. Add the DMA transfer descriptor to the allocated DMA resource.

```
dma_add_descriptor(&rx_dma_resource, &rx_dma_descriptor);
```

    2. Start the DMA transfer job with the allocated DMA resource and transfer descriptor.

```
dma_start_transfer_job(&rx_dma_resource);
```

Configure the DMAC module to transmit data through I^2S serializer 0.

The flow is similar to last DMA configure step for receive.

1. Allocate and configure the DMA resource

```
struct dma_resource tx_dma_resource;
```

```
struct dma_resource_config config;
dma_get_config_defaults(&config);
config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
config.peripheral_trigger = CONF_TX_TRIGGER;
dma_allocate(&tx_dma_resource, &config);
```

2. Prepare DMA transfer descriptor

```
COMPILER_ALIGNED(16) DmacDescriptor tx_dma_descriptor;
```

```
struct dma_descriptor_config descriptor_config;

dma_descriptor_get_config_defaults(&descriptor_config);

descriptor_config.block_transfer_count = 4;
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.source_address =
        (uint32_t)tx_values + sizeof(tx_values);
descriptor_config.destination_address = (uint32_t)&CONF_I2S_MODULE-
>DATA[0];

dma_descriptor_create(&tx_dma_descriptor, &descriptor_config);

tx_dma_descriptor.DESCADDR.reg = (uint32_t)&tx_dma_descriptor;
```

3. Start DMA transfer job with prepared descriptor

```
dma_add_descriptor(&tx_dma_resource, &tx_dma_descriptor);
dma_start_transfer_job(&tx_dma_resource);
```

Configure the I2S.

1. Create I²S module software instance structure for the I²S module to store the I²S driver state while it is in use.

```
struct i2s_module i2s_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the I²S module.
    1. Initialize the I²S module.

```
i2s_init(&i2s_instance, CONF_I2S_MODULE);
```

    2. Initialize the I²S Clock Unit.
        1. Create a I²S module configuration struct, which can be filled out to adjust the configuration of a physical I²S Clock Unit.

```
struct i2s_clock_unit_config config_clock_unit;
```

        2. Initialize the I²S Clock Unit configuration struct with the module's default values.

```
i2s_clock_unit_get_config_defaults(&config_clock_unit);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

        3. Alter the I²S Clock Unit settings to configure the general clock source, MCK, SCK, and FS generation.

```
config_clock_unit.clock.gclk_src = GCLK_GENERATOR_0;

config_clock_unit.clock.mck_src =
I2S_MASTER_CLOCK_SOURCE_GCLK;
config_clock_unit.clock.mck_out_enable = true;
config_clock_unit.clock.mck_out_div = 2;
```

```
config_clock_unit.clock.sck_src =
I2S_SERIAL_CLOCK_SOURCE_MCKDIV;
config_clock_unit.clock.sck_div = 4;

config_clock_unit.frame.number_slots = 2;
config_clock_unit.frame.slot_size = I2S_SLOT_SIZE_32_BIT;
config_clock_unit.frame.data_delay = I2S_DATA_DELAY_0;

config_clock_unit.frame.frame_sync.source =
I2S_FRAME_SYNC_SOURCE_SCKDIV;
config_clock_unit.frame.frame_sync.width =
I2S_FRAME_SYNC_WIDTH_HALF_FRAME;
```

4. Alter the I²S Clock Unit settings to configure the MCK, SCK, and FS output on physical device pins.

```
config_clock_unit.mck_pin.enable = true;
config_clock_unit.mck_pin.gpio = CONF_I2S_MCK_PIN;
config_clock_unit.mck_pin.mux = CONF_I2S_MCK_MUX;

config_clock_unit.sck_pin.enable = true;
config_clock_unit.sck_pin.gpio = CONF_I2S_SCK_PIN;
config_clock_unit.sck_pin.mux = CONF_I2S_SCK_MUX;

config_clock_unit.fs_pin.enable = true;
config_clock_unit.fs_pin.gpio = CONF_I2S_FS_PIN;
config_clock_unit.fs_pin.mux = CONF_I2S_FS_MUX;
```

5. Configure the I²S Clock Unit with the desired settings.

```
i2s_clock_unit_set_config(&i2s_instance, I2S_CLOCK_UNIT_0,
        &config_clock_unit);
```

3. Initialize the I²S Serializers.
   1. Create a I²S Serializer configuration struct, which can be filled out to adjust the configuration of a physical I²S Serializer.

```
struct i2s_serializer_config config_serializer;
```

   2. Initialize the I²S Serializer configuration struct with the module's default values.

```
i2s_serializer_get_config_defaults(&config_serializer);
```

      Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

   3. Alter the I²S Serializer settings to configure the SD transmit generation.

```
config_serializer.clock_unit = I2S_CLOCK_UNIT_0;
config_serializer.mode = I2S_SERIALIZER_TRANSMIT;
config_serializer.data_size = I2S_DATA_SIZE_16BIT;
```

   4. Alter the I²S Serializer settings to configure the SD transmit on a physical device pin.

```
config_serializer.data_pin.enable = true;
config_serializer.data_pin.gpio = CONF_I2S_SD_PIN;
config_serializer.data_pin.mux = CONF_I2S_SD_MUX;
```

   5. Configure the I²S Serializer 0 with the desired transmit settings.

```
i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_0,
        &config_serializer);
```

6. Alter the I²S Serializer settings to configure the SD receive.

```
config_serializer.loop_back = true;
config_serializer.mode = I2S_SERIALIZER_RECEIVE;
config_serializer.data_size = I2S_DATA_SIZE_16BIT;
```

7. Alter the I²S Serializer settings to configure the SD receive on a physical device pin (here it's disabled since we use internal loopback).

```
config_serializer.data_pin.enable = false;
```

8. Configure the I²S Serializer 1 with the desired transmit settings.

```
i2s_serializer_set_config(&i2s_instance, I2S_SERIALIZER_1,
        &config_serializer);
```

4. Enable the I²S module, the Clock Unit and Serializer to start the clocks and ready to transmit data.

```
i2s_enable(&i2s_instance);
i2s_clock_unit_enable(&i2s_instance, I2S_CLOCK_UNIT_0);
i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_1);
i2s_serializer_enable(&i2s_instance, I2S_SERIALIZER_0);
```

9.3.2. Use Case

9.3.2.1. Code

Copy-paste the following code to your user application:

```
while (true) {
    /* Infinite loop */
}
```

9.3.2.2. Workflow

1. Enter an infinite loop while the signals are generated via the I²S module.

```
while (true) {
    /* Infinite loop */
}
```

10. Document Revision History

Doc. Rev.	Date	Comments
42255B	12/2015	Added support for SAM DA1
42255A	01/2014	Initial release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**