**NXP Semiconductors**
Application Notes

Document Number: AN12202
Rev. 0, 11/2018

# Using Synchronous Audio Interface (SAI) on S32K148

by: NXP Semiconductors

## Contents



## 1. Introduction

The Synchronous Audio Interface (SAI) module found in S32K148 device is targeted to be used in different applications in which audio processing is required. This module is highly-configurable and allows users to process audio in different audio format such as I2S, Codec/DSP and TDM.

This application note is focused on providing an overview, module explanation and use-case implementation in different audio formats for SAI module.

## 2. Audio bus topology

The audio bus (either for I2S, Codec/DSP modes, TDM, etc.) was designed to minimize the number of pins required and to keep wiring simple. A 3-line serial bus is used consisting of one serial data line, one channel/word selection and one clock line. Since the transmitter and receiver have the same clock signal for data transmission, there are two different roles involved: Master and Slave.

Device either transmitter or receiver in charge to provide clock and word select lines is defined as Master, there must only be one master in the bus no matter if several transmitters and receivers are connected. The rest of devices connected to the bus are named as slaves. Following image depicts these master and slave roles on common audio bus connections.

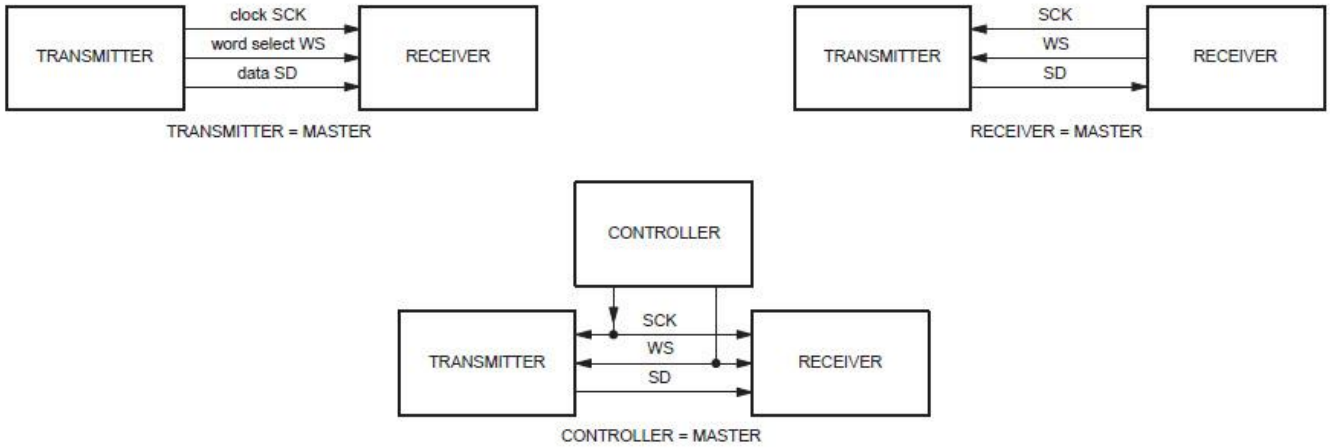



**Figure 1. Master and Slaves roles within an audio bus connection**

# 3. Audio formats

Before reviewing SAI module and its features, it is important to understand different audio formats and concepts related to digital audio signals. Although some audio ICs/Codec might have different names for some audio formats presented in this section, it is important to identify each format based on its characteristics rather than its name.

## 3.1. Inter-IC Sound (I2S)

A serial protocol (I2S) specially for digital audio applications was developed to standardized communication between IC manufacturer and audio processing units. The I2S bus has three lines:

- Continuous Serial clock (SCK), Bit Clock (BCLK)
- Word Select (WS), Frame Sync (FS), Word Clock (WCLK), Left-Right Clock (LRCLK)
- Serial Data (SD), Serial Data Out/In (SDOUT, SDIN)

Serial clock (SCK) also known as bit clock (BCLK) is the line used to provide clock reference for each audio bit. Word Select (WS), Frame Sync (FS) or Word clock (WCLK) indicates the channel being transmitted: when this line is set to ‘0’, channel 1 (left) is being transmitted, while when this line is set to ‘1’, channel 2 (right) is transmitted. WS line changes one clock period before the MSB is transmitted. The frequency of this line corresponds to the Audio sample rate frequency. Serial data (SD) is transmitted in two’s complement with MSB first always (as both transmitter and receiver may have different word lengths). Data range from 8 to 32-bit. The figure below shows the diagram for I2S format.

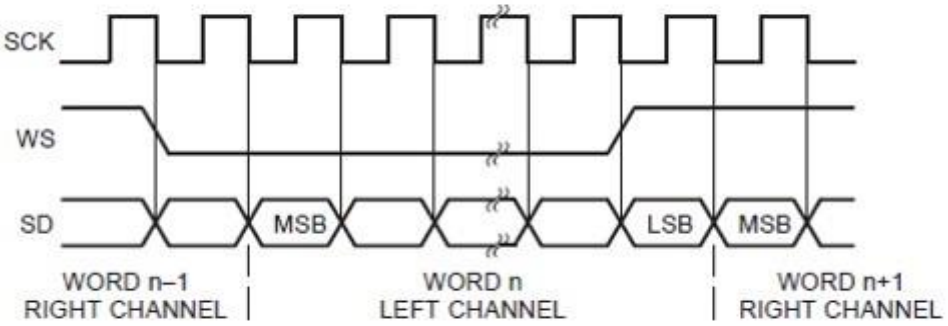



**Figure 2. I2S diagram**

## 3.2. Codec mode (Left/Right justified)

Codec mode can be compared with I2S protocol but Word Select is asserted at the same time that first bit is transmitted for current frame (it is not delayed one bit clock as in I2S). Also, channel selection by Word Select signal is inverted in comparison with I2S: when Word Select is set to '0', right channel is transmitted and when it is set to '1', left channel is transmitted.

There are two variants explained below.

### 3.2.1. Left-Justified (MSB justified)

For Left-Justified also known as MSB justified, the Word Select changes when the MSB bit for current frame is available. Serial data is justified to the left, which means that if Word Select's half period is 32-bit long and only 24 bits are used for audio data, the first 24 bits will be used for audio and the remaining 8 bits must be set to zero.

The figure below depicts Left-Justified format.

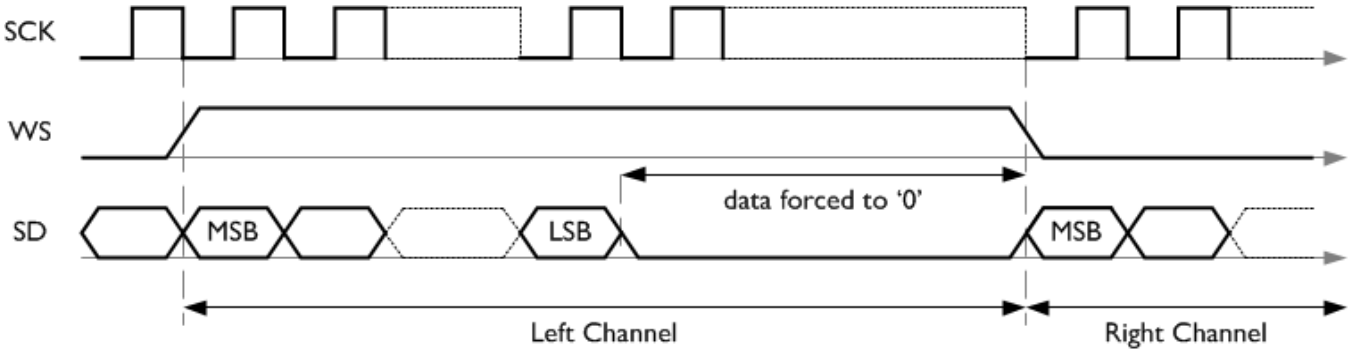



**Figure 3. Left-Justified format**

### 3.2.2. Right-Justified (LSB justified)

For Right-Justified also known as LSB justified, the Word Select changes when the first bit for current frame is available. Serial data is justified to the right, which means that if Word Select's half period is 32-bit long and only 24 bits are used for audio data, the first 8 bits must be set to zero while remaining

24 bits will be used for audio data. As serial data is transmitted in MSB format, LSB bit matches with last bit clock cycle before WS changes its state.

The figure below depicts Right-Justified format.

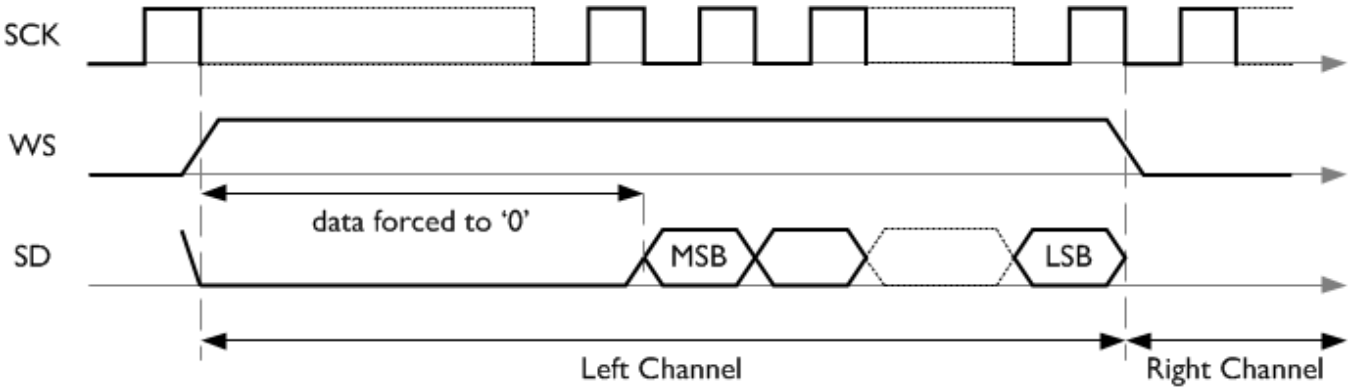



**Figure 4. Right-Justified format**

## 3.3. DSP mode

DSP mode is similar to Left-Justified Codec format but Word Select's width may vary depending on IC architecture (minimum allowed value is 1 bit clock). As Word Select signal is not a 50% duty cycle signal, the rising edge of the Word Select signals the beginning of audio data with the left channel data first followed by right channel data immediately. Word Select's frequency still defines the audio sample rate.

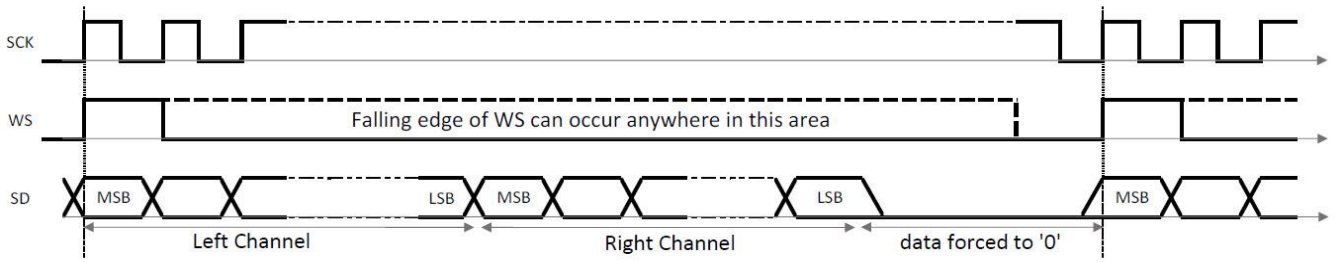



**Figure 5. DSP format**

There are some DSP mode connections in which, data is delayed one bit clock as in I2S format. The following figure illustrate this DSP connection where Word Select width is only 1 bit clock long.

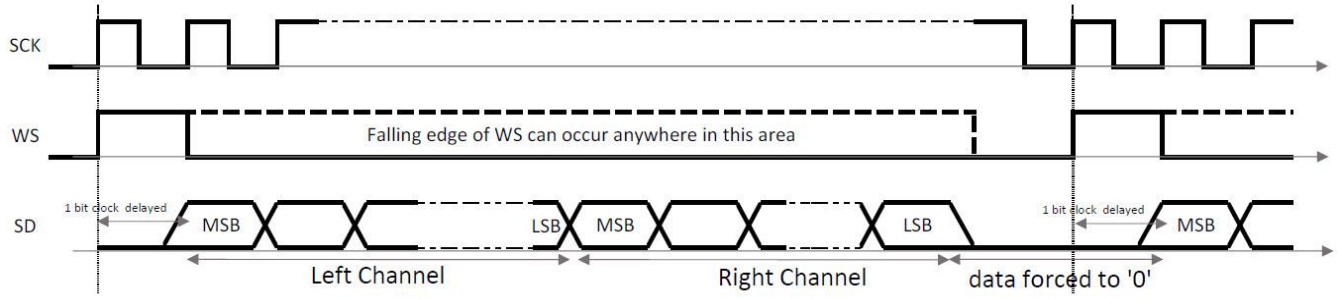



**Figure 6. DSP mode with serial data delayed one bit clock**

**NOTE**

Some ICs might require zero-forced data between each left and right channel instead of leaving this to the end. It is important to understand ICs' requirements from device's datasheet.

## 3.4. Time-Division Multiplexed (TDM)

For previous audio formats, only 2 channels can be sent in a single Word-Select period, however, for TDM format, sending more than 2 channels in same Word-Select period is possible. In TDM format, Word Select's width is only 1 bit clock long.

TDM mode is usually used when more than one slave is connected to the bus, in which, master sends data for all slaves using same synchronization (word-select) line. Every slave is configured to use their own channel by setting an "offset" since the Word-Select's assertion to the channel that corresponds to each slave.

## 3.5. PCM

In Pulse Code Modulation (PCM), only one channel is transferred. There are two types of synchronization modes: short frame and long frame. For ***short frame*** synchronization mode, the falling edge of the Word Select / Frame Sync indicates the start of the serial data. Word Select/Frame Sync is always one clock cycle long. For ***long frame*** synchronization mode, the rising edge of the Word Select/Frame Sync indicates the start of the serial data. Word Select / Frame Sync stays active high for 13 clock cycles. Figure below shows PCM formats for both synchronization modes.

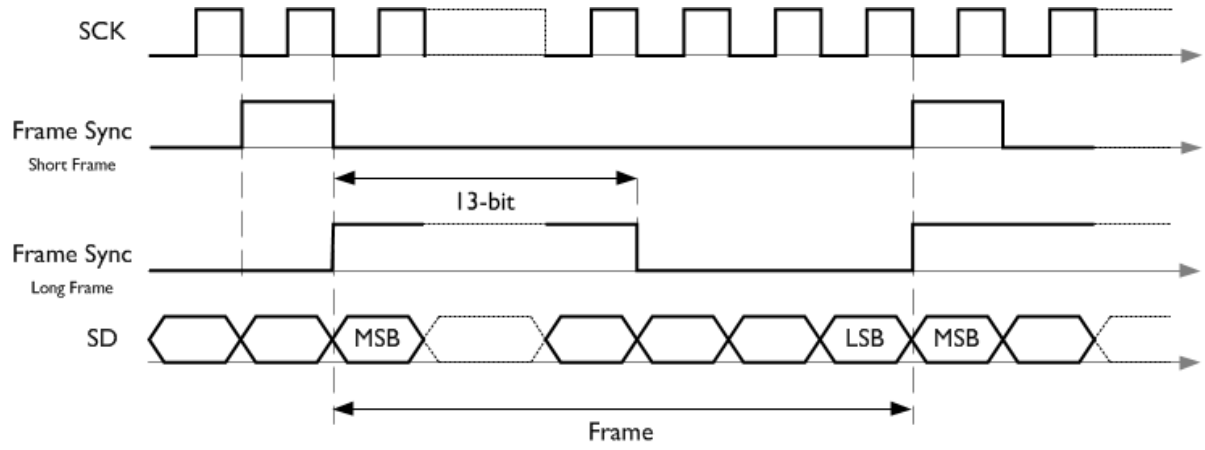



**Figure 7. PCM format for both short and long synchronization modes**

# 4. SAI module overview

As Synchronous Audio Interface (SAI) supports full-duplex serial interface and several audio protocols such as I2S, AC97, TDM and codec/DSP interfaces, this section is focused on providing a brief explanation on SAI features/components and how to configure them for different audio formats.

SAI's lines are identified as BCLK, SYNC and Dn for clock line, word select/frame sync line and data lines respectively.

For detailed information on SAI module architecture, see SAI's chapter in device's Reference Manual.

## 4.1. SAI architecture

SAI module in S32K148 includes logic for both transmitter and receiver independently. Each logic includes FIFO support of 8 words (32 bits each) with DMA support to efficiently improve audio processing performance. There are two SAI instances in S32K148: SAI0 and SAI1, SAI0 has support for 4 data lines while SAI1 has support for only one data line.

The figure shows SAI's block diagram.

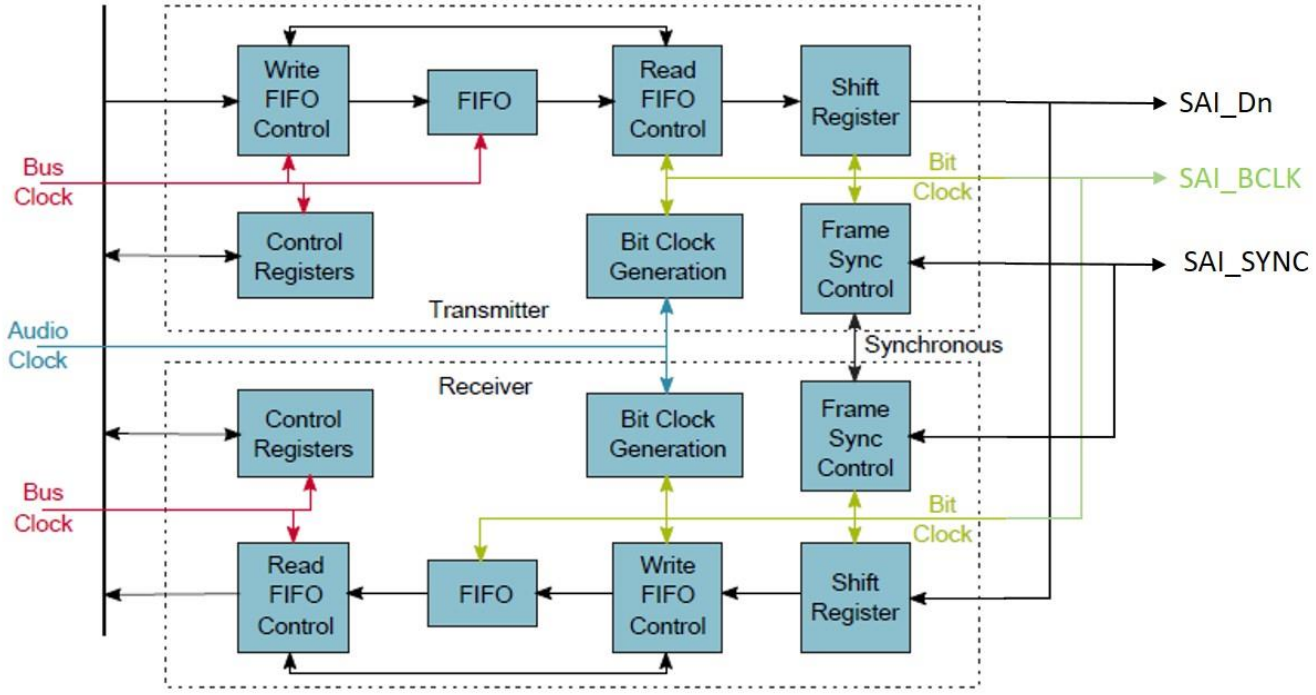



**Figure 8. SAI block diagram on S32K148**

As communication lines are connected together for both transmitter and receiver, only one can be enabled at a time. Be aware that registers for transmitter and receiver are independent between each other, so a register will be called as SAI_xCRn to refer to either SAI_TCRn or SAI_RCRn respectively.

## 4.2. SAI clocking

Before accessing to any SAI register, it is necessary to enable module clock in PCC_SAIx register for both master and slave configurations.

### 4.2.1. Clock configuration as master

When SAI is configured as Master (SAI_xCR4[FSD]=1 and SAI_xCR2[BCD]=1), BCLK and SYNC signals will be generated internally based on clock source reference and divider configuration fields. In S2K148, SAI can use up to four different clock sources: its module clock (Bus Clock), its external MCLK clock, frequency from System Oscillator Clock Divider 1 (SOSCDIV1) and the external MCLK clock from the opposite SAI instance. The figure below shows these clock options.

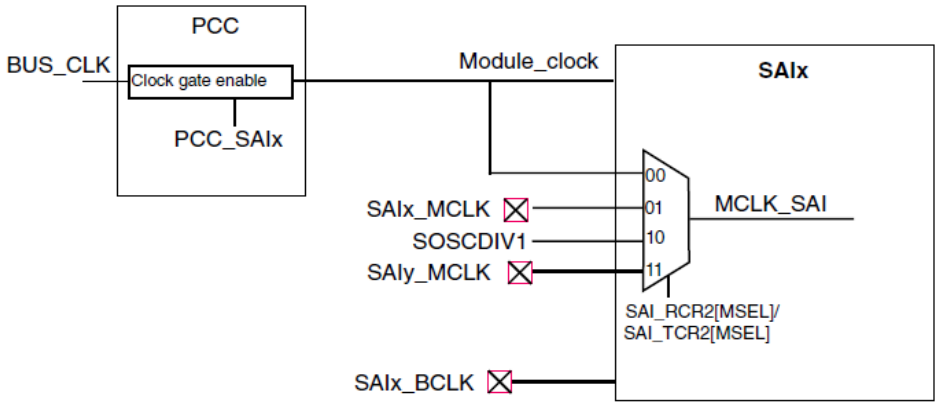



**Figure 9. SAI clock source options**

**NOTE**

SAIx_MCLK refers to MCLK pin for that specific instance while SAIy_MCLK refers to MCLK pin for the other instance.

Once MCLK_SAI is selected, this clock is used to calculate the Bit Clock frequency as shown below:

$$BCLK = \frac{MCLK\_SAI}{(DIV + 1) \cdot 2}$$

Where DIV is an 8-bit field from SAIx_TCR2 or SAIx_RCR2 depending if transmitter or receiver is configured.

As common audio frequency is based on frame Sync (sample rate) frequency, bit clock can be calculated from sample rate as follows:

$$BCLK = bits\ per\ channel \cdot number\ of\ channels \cdot sample\ rate\ freq$$

So, DIV field can be calculated from sample rate frequency as shown below:

$$bits\ per\ channel \cdot number\ of\ channels \cdot sample\ rate = \frac{MCLK\_SAI}{(DIV + 1) \cdot 2}$$

$$DIV + 1 = \frac{MCLK\_SAI}{bits\ per\ channel \cdot number\ of\ channels \cdot sample\ rate \cdot 2}$$

$$DIV = \frac{MCLK\_SAI}{bits\ per\ channel \cdot number\ of\ channels \cdot sample\ rate \cdot 2} - 1$$

For example, for I2S format, number of channels is 2 (left and right), bits per channel is 32 and sample rate can be: 8 kHz, 11.025 kHz, 16 kHz, 22.05 kHz, 32 kHz, 44.1 kHz, 48 kHz or 96 kHz. Assuming that MCLK_SAI clock is derived from bus clock (SAI_xCR2[MSEL] = 0) and bus clock has been configured to 40 MHz, then DIV value is:

$$DIV = \frac{40\ MHz}{32 \cdot 2 \cdot sample\ rate \cdot 2} - 1$$

Following table shows different DIV values for some of the main sample rate values.

**Table 1. Sample Rate and DIV values**

| Sample Rate (kHz) | Exact DIV value | Closet DIV value | Resulting BCLK frequency (MHz) (SAI clock as 40Mhz) | Real Sample Rate (BCLK / 64) in KHz |
|---|---|---|---|---|
| 8 | 38.06 | 38 | 0.512 | 8.012 |
| 11.025 | 27.34 | 27 | 0.714 | 11.16 |
| 16 | 18.53 | 18 | 1.052 | 16.447 |
| 22.05 | 13.17 | 13 | 1.428 | 22.321 |
| 32 | 8.76 | 9 | 2.222 | 34.722 |
| 44.1 | 6.08 | 6 | 2.857 | 44.642 |
| 48 | 5.51 | 5 | 3.333 | 52.08 |
| 96 | 2.25 | 2 | 6.666 | 104.166 |

In order to get more accurate sample rate frequencies, it is recommended to use MCLK clock with standard master clock values. For example, if MCLK is set to 24.576 MHz, more accurate DIV values are obtained for 8, 16, 32, 48 and 96 kHz as shown below:

**Table 2. Sample Rate and DIV values when MCLK is 24.576MHz**

| Sample Rate (kHz) | DIV value | Resulting BCLK frequency (MHz) |
|---|---|---|
| 8 | 23 | 0.512 |

| Sample Rate (kHz) | DIV value | Resulting BCLK frequency (MHz) |
|---|---|---|
| 11.025 | 16.41497 | 0.7056 |
| 16 | 11 | 1.024 |
| 22.05 | 7.707483 | 1.4112 |
| 32 | 5 | 2.048 |
| 44.1 | 3.353741 | 2.8224 |
| 48 | 3 | 3.072 |
| 96 | 1 | 6.144 |

### 4.2.2. Clock configuration as slave

When SAI is configured as Slave (SAI_xCR4[FSD]=0 and SAI_xCR2[BCD]=0), as BCLK and SYNC will be generated externally by master, there is no need to configure SAI_xCR2[MSEL] and SAI_xCR2[DIV] values.

## 4.3. Synchronous modes

SAI module can be configured to operate using two different synchronization modes:

- Synchronous Mode: SAI transmitter and receiver can be configured to operate with synchronous bit clock and frame sync and controlled by the same module enabled. (Both transmitter and receiver operate using same SAI lines)
- Asynchronous mode: SAI transmitter or receiver can be configured to operate asynchronously between each other.

Although SAI module implementation in S32K148 do not allow to operate both transmitter and receiver of same instance simultaneously (due most lines are already shared between transmitter and receiver), it is recommended that synchronous mode is set in either transmitter or receiver registers (TCR2[SYNC] or RCR2[SYNC] respectively) in order to correctly generate the SAI's internal bit clock when it is configured as master.

## 4.4. SAI configuration fields

SAI module allows to configure some audio settings as shown in figure below.

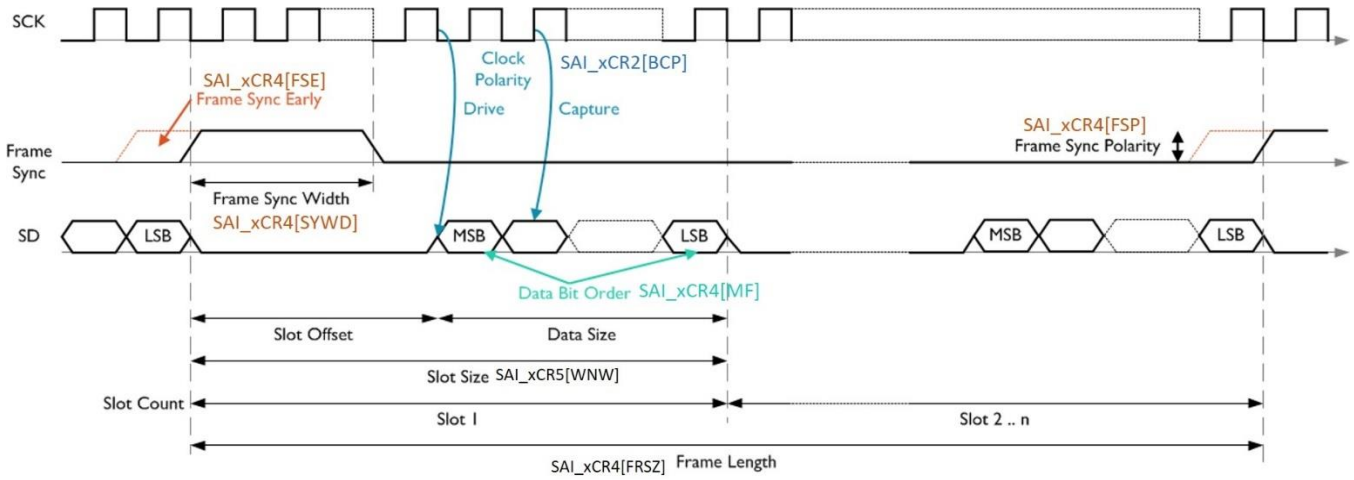



**Figure 10. Audio settings and SAI's fields involved**

- Clock polarity (SAI_xCR2[BCP]): Clock line can be configured in two different ways depending on which edge is used for driving the data and which is used for sampling/capturing the data.
- Frame sync early (SAI_xCR4[FSE]): The option to assert one bit-clock the frame sync line with respect data line is called Frame sync early feature. If Frame sync early features is enabled, then Frame Sync is asserted one bit clock before the first bit of the current frame. If this feature is disabled, then Frame sync line will be asserted at the same that first bit of current frame is available.
- Frame sync polarity (SAI_xCR4[FSP]): This feature configures whether Frame sync is active high (default) or low.
- Frame sync width (SAI_xCR4[SYWD]): Defined in ‘bit-clock’ numbers, this parameter establishes how many bit-clocks the frame sync will be active. The value written must be one less that the number of bit clocks.
- Data bit order (SAI_xCR4[MF]): Although most audio formats required data line to be configured as MSB, there is an option to change this feature to LSB if needed.
- Slot size (SAI_xCR5[WNW]): Defined in ‘bit-clock’ numbers. This parameter configures the number of bits in each word. The value written must be one less that the number of bits per word.
- Slot count (SAI_xCR4[FRSZ]): This feature configures the number of words (also known as channels) in each audio frame. The value written must be one less than the number of words in the frame.
- Slot offset: This feature is implemented by shifting data while writing into SAI_TDRn. Commonly, slot size is affected by shifting data “n” bits to the right/left to align into specific bit value.

## 4.5. SAI FIFO and DMA/Interrupt generation

SAI transmit and receive channel include a FIFO of 8 words (32-bit) each. The FIFO data is accessed using the SAI Transmit/Receive Data registers.

SAI_TCR1 and SAI_RCR1 registers configure FIFO watermark to be less than or equal to eight if needed. This FIFO watermark helps user to set a specific value in which, FIFO does not need to get empty before requesting more audio data to DMA.

There are two different flags used to trigger a DMA requests:

1. FIFO request flag: Signals if current FIFO entries is below/above the watermark configuration for the transmit FIFO and receive FIFO respectively. It is automatically cleared if number of entries are above/below the watermark value. The figures below show FIFO request flag values for transmitter and receiver under certain conditions.

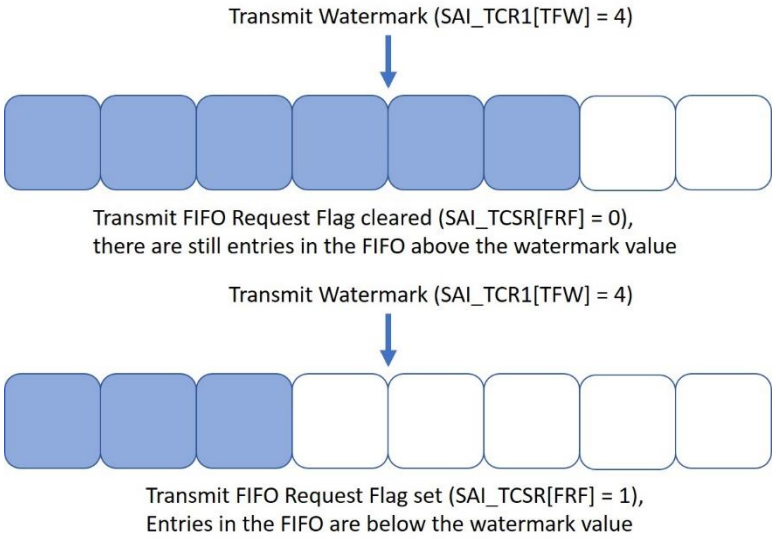



**Figure 11. FIFO request flag for transmitter**

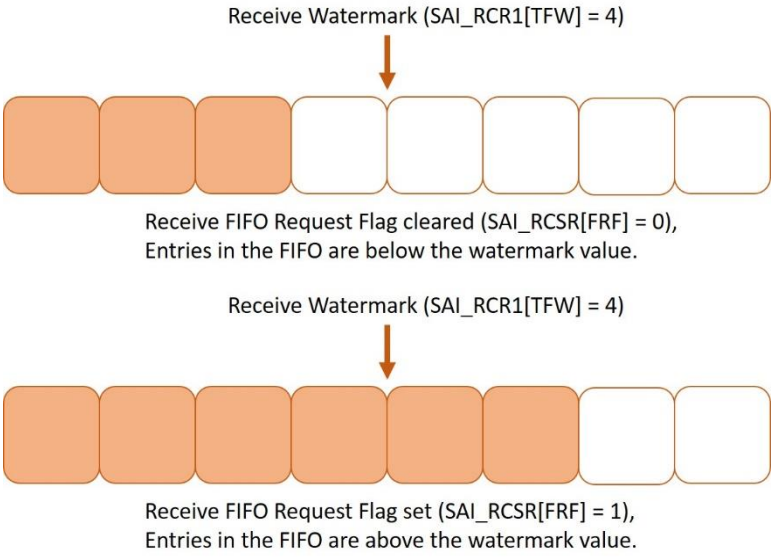



**Figure 12. FIFO request flag for receiver**

2. FIFO warning flag: Signals if FIFO is empty (for transmitter) or full (for receiver) and it is cleared automatically if the condition is removed (if FIFO is no longer empty for transmitter or full for receiver).

SAI transmitter and receiver generate separate interrupts and separate DMA requests, but support the same status flags. Section interrupts and DMA requests in device reference manual describes these flags in depth.

### 4.5.1. FIFO error handling

Whenever a FIFO error is detected, SAI module will stop transferring data unless SAI_xTCR4[FCONT] bit is set. In order to continue sending/receiving data, SAI_xCSR[FEF] flag must be cleared.

## 4.6. Masking SAI channels

SAI module allows user to mask/unmask channels by clearing/setting bits in Transmit Mask Register (SAIx_TMR) or Receive Mask Register (SAIx_RMR). Each bit from this 16-bit field represents a channel in the frame. By masking (clearing) the channel bit, the respective channel is sent/received from/within the SAI's FIFO. If unmasking (setting) the channel bit, the respective channel is not considered to be read/written from/to FIFO.

For example, if transmitting audio data in I2S format two channels are required, but in case that only one of these channels is used (for example left channel), bit 0 (channel 1 - left) should be cleared while bit 1 (channel 2 - right) can be set. After doing this, all data in the transmitter FIFO will contain data for channel 1(left). If channel 2 is not unmasked, then FIFO will need to stored data for both left and right channels so user should write zeros to all words corresponding to right channel, so user padding is needed if channel masking is not set properly.

## 4.7. SAI initialization procedure

Next section lists all steps needed when configuring SAI module.

1. Turn SAI module's clock on by setting PCC_SAIx[CGC] bit.
2. Configure SAI_BCLK, SAI_Dn and SAI_SYNC pins functionality in PORTx_PCR registers.
3. Disable Transmitter/Receiver.
4. Reset FIFOs for both transmitter and receiver.
5. Configure synchronous mode for SAI's transmitter and receiver by setting SAI_xCR2[SYNC] field.
6. Configure SAI_xCR2, SAI_xCR4, SAI_xCR5 register to meet desired audio format's characteristics.
7. Configure Master/Slave mode by setting/clearing SAI_xCR4[FSD] and SAI_xCR2[BCD] bits.
8. If master mode is required, adjust clock settings as explained in SAI clocking section.
9. Mask/Unmasked desired channels in SAI_xMR register.

10. Disable SAI data lines by clearing corresponding bits in SAI_xCR3[TCE].
11. In case DMA/interrupts are used, configure FIFOs watermark in SAI_xCR1[TWF].
12. Enable Transmitter/Receiver. After transmitter/receiver is enabled, bit clock and frame sync signals are generated (if configured as master).

### 4.7.1. Triggering a SAI transmission/reception operation

If user wants to start a SAI transmission/reception, after SAI module is initialized, user needs to follow next steps.:

- Using DMA
    1. Be sure DMAMUX module is already configured to use SAI's transmitter/receiver requests.
    2. If SAI is configured as transmitter, write to SAI transmitter FIFOs in order to fill the 8-entries FIFO. This will lead that transmitter buffer has already data when transmission starts, otherwise, an underrun error might happen.
    3. Be sure DMA's TCD is configured to move data to/from SAI transmitter/receiver from/to RAM locations. Depending on SAI configuration (bits per channel, watermark value) TCD configuration might vary. Consider that for transmitter, eight entries are already moved from RAM to SAI's FIFO buffer.
    4. Enable DMA requests on SAI's registers: SAI_xCSR[FWDE] and/or SAI_xCSR[FRDE].
    5. Enable used channels in SAI_xCR3[TCE].
- Using Interrupts.
    1. Be sure SAI interrupt is already enabled in NVIC.
    2. If SAI is configured as transmitter, write to SAI transmitter FIFOs in order to fill the eight entries FIFO. This will lead that transmitter buffer has already data when transmission starts, otherwise, an underrun error might happen.
    3. Enable interrupt request on SAI's registers: SAI_xCSR[FWIE] and/or SAI_xCSR[FRIE].
    4. Enable used channels in SAI_xCR3[TCE].

### 4.7.2. Pausing/Stopping SAI transfers

In case user is moving data by DMA/Interrupt and transfer is intended to be paused/stopped, user should perform steps listed below to avoid a FIFO error event.

1. Disable DMA request / Interrupt generation by clearing SAI_xCSR[FWDE] and/or SAI_xCSR[FRDE] or SAI_xCSR[FWIE] and/or SAI_xCSR[FRIE] respectively.
2. Wait until SAI_xCSR[FWF] is set signaling that FIFO is empty.
3. Disable all used channels by clearing corresponding bits in SAI_xCR3[TCE].
4. In case that transfer is intended to be stopped, reset FIFO pointers by setting SAI_xCSR[FR].

# 5. SAI configuration for different audio formats

Table below shows different configuration fields according some audio formats. Some other configuration fields not listed below might vary depending on codec architecture. Data bits represents how many bits represent an audio frame, while Slot Size indicates how many bits are included in each channel. Example consider that SAI clock is taken from Bus clock (40MHz) and desired sample rate is 48kHz.

**Table 3. SAI configuration fields for different audio formats**

| Audio Format | Data bits | Frame Sync Early *SAI_xCR4[FSE]* | Frame Sync Width *SAI_xCR4[SYWD]* | Slot Offset *(SAI_xDRn)]* | Slot Size *SAI_xCR5[WNW]* | Slot count *SAI_xCR4[FRSZ]* | Bit Clock divider *(SAI_xCR2[DIV])* | Channel Mask *SAI_xMR[xWM]* | Diagram |
|---|---|---|---|---|---|---|---|---|---|
| I2S | 32 | 1 | 31 | Data[31:0] | 31 | 1 | 6 | 0xFFFC | Figure A. I2S 32bit format |
| | 24 | 1 | 31 | (Data[23:0] << 8) & 0xFFFFFF00 | 31 | 1 | 6 | 0xFFFC | Figure B. I2S 24bit format |
| | 16 | 1 | 31 | (Data[15:0] << 16) & 0xFFFF0000 | 31 | 1 | 6 | 0xFFFC | Figure C. I2S 16bit format |
| Right-Justified | 32 | 0 | 31 | Data[31:0] | 31 | 1 | 6 | 0xFFFC | Figure D. Left-Justified 32bit / Right-Justified 32bit formats |
| | 24 | 0 | 31 | Data[23:0] & 0x00FFFFFF | 31 | 1 | 6 | 0xFFFC | Figure H. Right-Justified 24bit format |
| | 16 | 0 | 31 | Data[15:0] & 0x0000FFFF | 31 | 1 | 6 | 0xFFFC | Figure I. Right-Justified 16bit format |
| Left-Justified | 32 | 0 | 31 | Data[31:0] | 31 | 1 | 6 | 0xFFFC | Figure D. Left-Justified 32bit / Right-Justified 32bit formats |
| | 24 | 0 | 31 | (Data[23:0] << 8) & 0xFFFFFF00 | 31 | 1 | 6 | 0xFFFC | Figure E. Left-Justified 24bit format |
| | 16 | 0 | 31 | (Data[15:0] << 16) & 0xFFFF0000 | 31 | 1 | 6 | 0xFFFC | Figure F. Left-Justified 16bit format |
| | 16 | 0 | 15 | (Data[15:0] << 16) & 0xFFFF0000 | 15 | 1 | 12 | 0xFFFC | Figure G. Left-Justified 16bit format (Slot size is 16) |
| TDM | 32 | 0 | 0 | (Data[15:0] << 16) & 0xFFFF0000 | 31 | 7 | 1 | 0xFF00 | Figure J. TDM, 32bit format, 8 channels |

# 6. Use cases

When implementing audio processing in embedded systems, it is important to manage audio buffers properly so no glitches are heard. There are several techniques to do this, the most common is to implement ping pong buffers that, along with DMA usage, decrease CPU overhead and facilitate audio processing.

Next section implements this audio processing method using basic example codes for S32K148. I2S in 16-bit mode is used for a 48 kHz sample rate.

**NOTE**

Although it is specified that 32 bits are used for each channel in I2S mode, 16-bit provides the most accurate value for desired sample rate using the 40 MHz SAI reference clock.

## 6.1. Ping pong buffer channel processing (left and right channels in same buffer)

By using the DMA in scatter gather configuration, the system is able to store audio data in two separated buffers. The CPU can process the information of buffer 1 while the DMA is transferring data from the buffer 2. When the transfer is finished, then the DMA will change its source address to start sending data from the buffer 1 and the CPU will start processing the data from Buffer 2. The figure below shows the block diagram for this implementation.

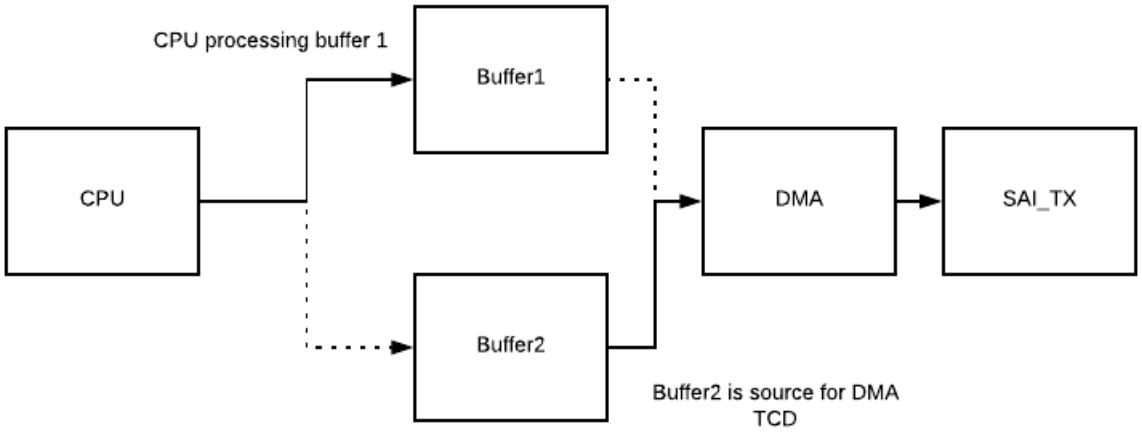



**Figure 13. Ping Pong buffer implementation**

CPU must be able to process all data before the DMA completes the transfer. For this specific use case, 8 words are sent on every DMA transfer, so time needed for DMA to complete the transfer is given by the following formula:

$$t = \frac{Total\ words}{sample\ rate \cdot words\ per\ frame} = \frac{8\ words}{48\ kHz \cdot 2\ words} = 83.33\mu s$$

In order to identify which buffer is free, application compares the source address for current DMA's TCD (DMA_TCD[n].SADDR) against the address saved on the first TCD array stored in SRAM.

Example code included in the application note shows the implementation done for this basic use case.

## 6.2. SAI receiver splitting Left and Right channels in separate buffers

In some audio applications, audio processing should be applied to each channel separately so, for this case, both receiver and transmitter have separated left and right buffers and the DMA is used to

send/receive the data from/to the proper buffer. In order to do this, the TCD is configured with especial offsets on every data transmission and at the end of every request.

DMA configuration plays a role to get ordered data properly. The following figure shows how the offsets are configured.

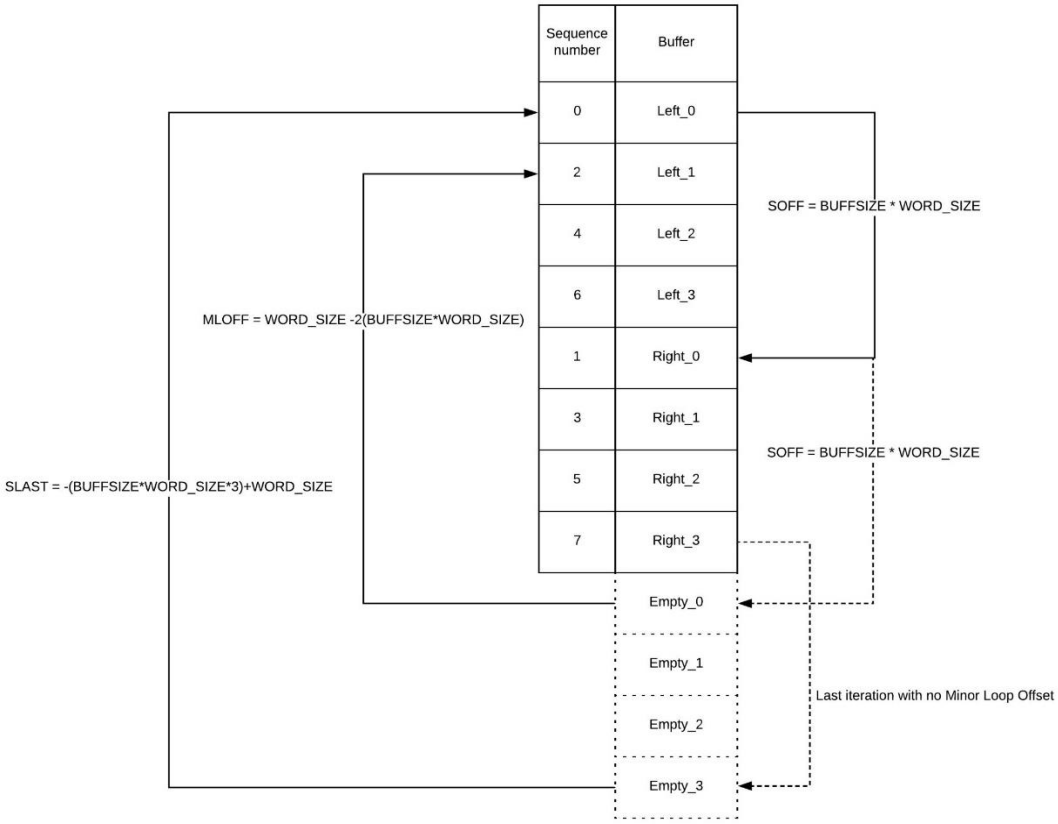



**Figure 14. DMA TCD offset configuration**

As TCD is configured to move two SAI words on every request (1, 2 or 4 bytes each depending on SAI configuration 8-.16- or 32-bits per word), DOFF is applied on every word write, so this offset is basically the address difference between buffer 1 and buffer 2 (or the buffer size multiplied by the word width). Minor loop offset is applied once the request is completed (once the two words are moved) so it returns to the next address for the previous buffer.

This process is repeated “citer” times to get both buffers filled with proper data. Once the MAJOR loop is completed, minor offset is not applied, so SADDR/DADDR are pointing to the last element on the “auxiliary buffer”. So, an adjust of (-3 * BUFFSIZE * WORDSIZE + WORDSIZE) is needed.

For this example code, SAI0 will be providing data to SAI1 which is in charge to receive and separate data.

For SAI1, reception buffers are including 4 more slots per channels as transmitter is sending eight additional data (four for right channel and four for left channel) as a part of the initialization procedure and to avoid underflow flag to be set in case the transmitter buffer is enabled with empty FIFO.

Both Transmitter (SAI0) and receiver (SAI1) uses the same DMA logic to move data to/from separate buffers. Following image shows the data seen on SAI lines and how they are split into 2 buffers when these is received.

# 7. SAI driver in SDK.

SAI driver is also included in SDK (0.8.6 version). There is a similar example code as the one shown in 6.2.

You can check SDK documentation for more information about the APIs used to configure SAI module.

# 8. References

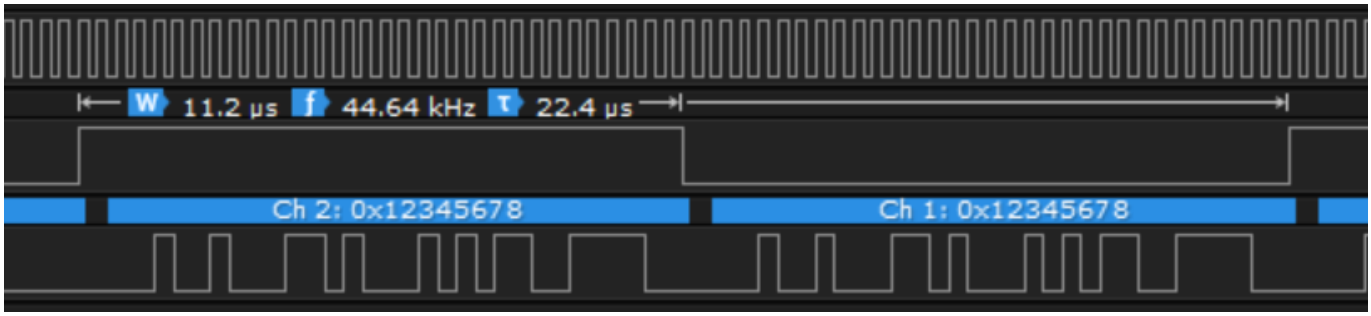
https://www.keil.com/pack/doc/CMSIS/Driver/html/group__sai__interface__gr.html

# 9. Revision History

| Version Number | Revision Date | Description of changes |
|---|---|---|
| Rev 0 | November 2018 | Initial release |

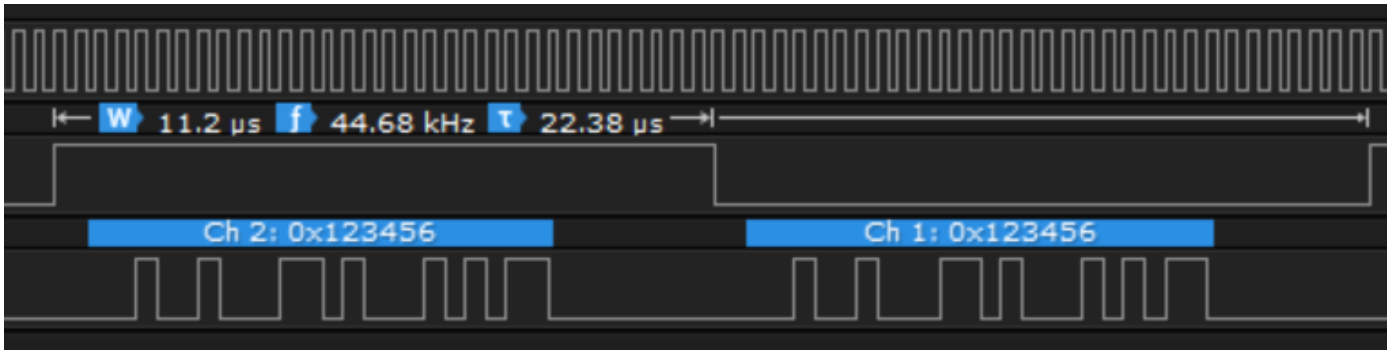
# Appendix A. Figures

## Figure A. I2S 32bit format



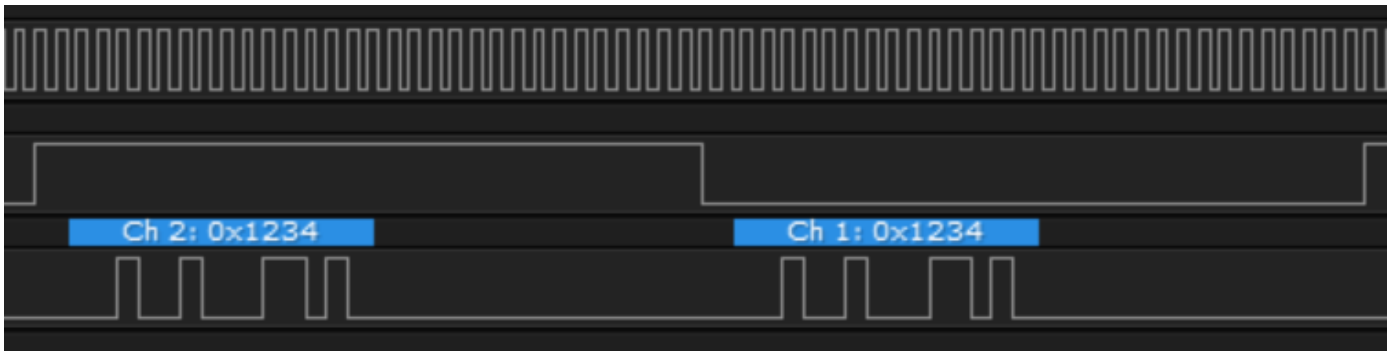

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex];
}
```

## Figure B. I2S 24bit format



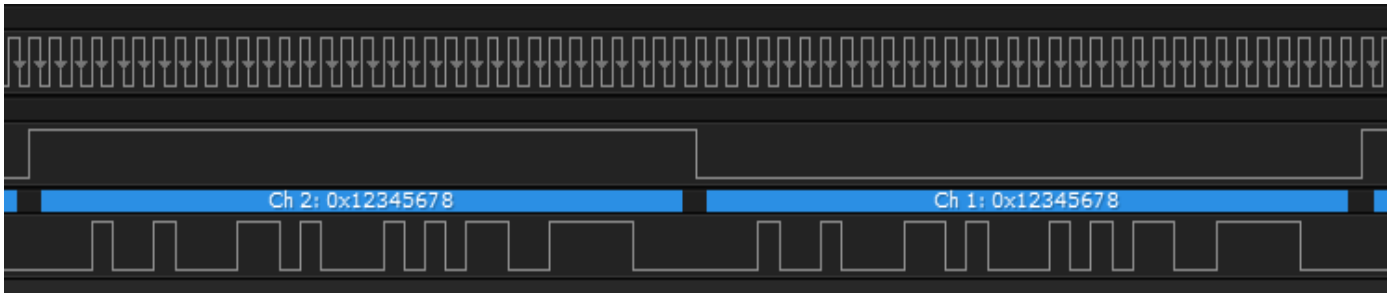

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex] & 0xFFFFFF00;
}
```

## Figure C. I2S 16bit format



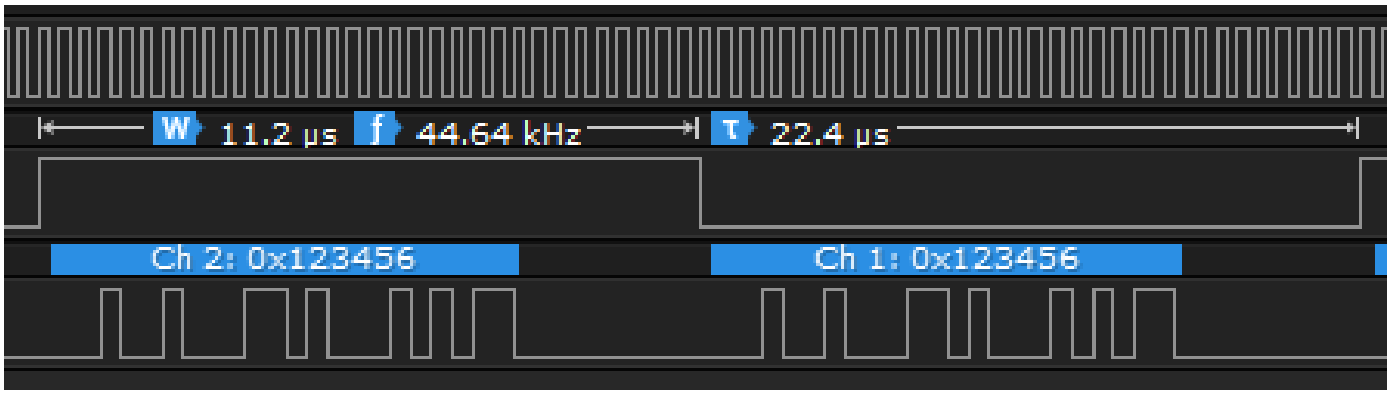

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex] & 0xFFFF0000;
}
```

## Figure D. Left-Justified 32bit / Right-Justified 32bit formats



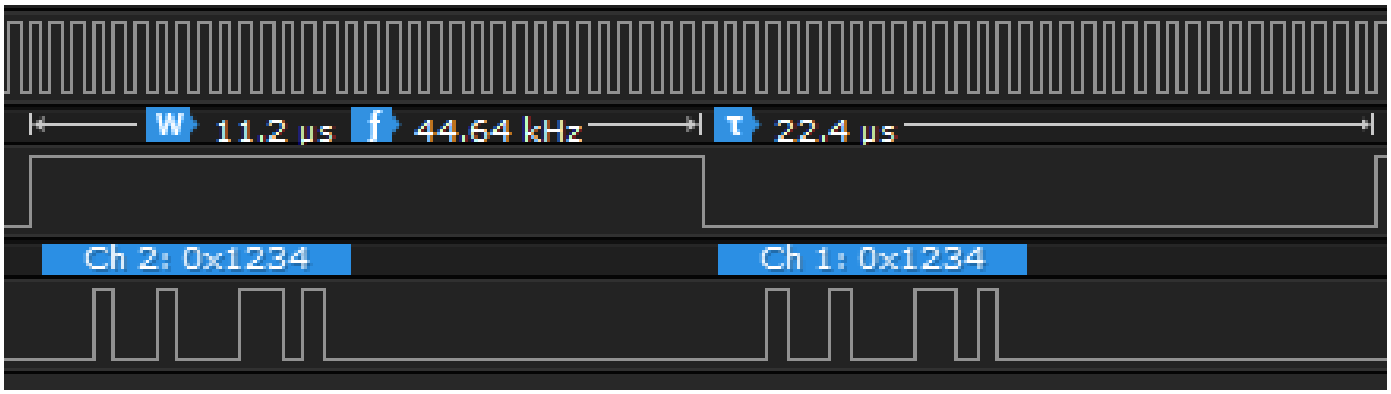

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex];
}
```

## Figure E. Left-Justified 24bit format



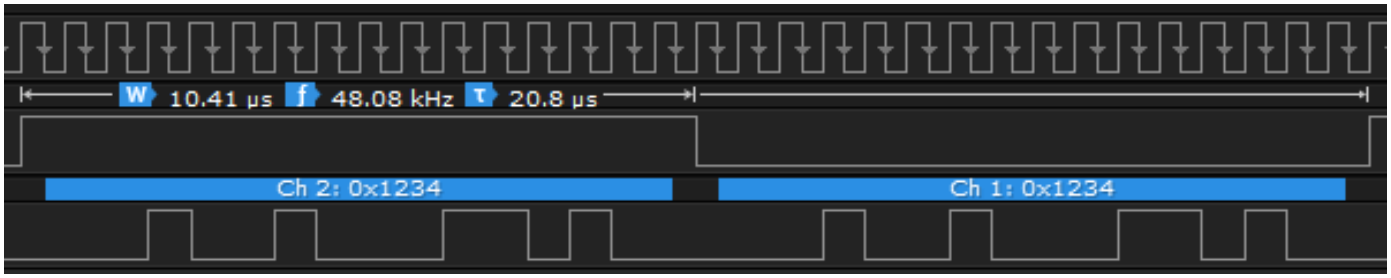
```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex] & 0xFFFFFF00;
}
```

## Figure F. Left-Justified 16bit format



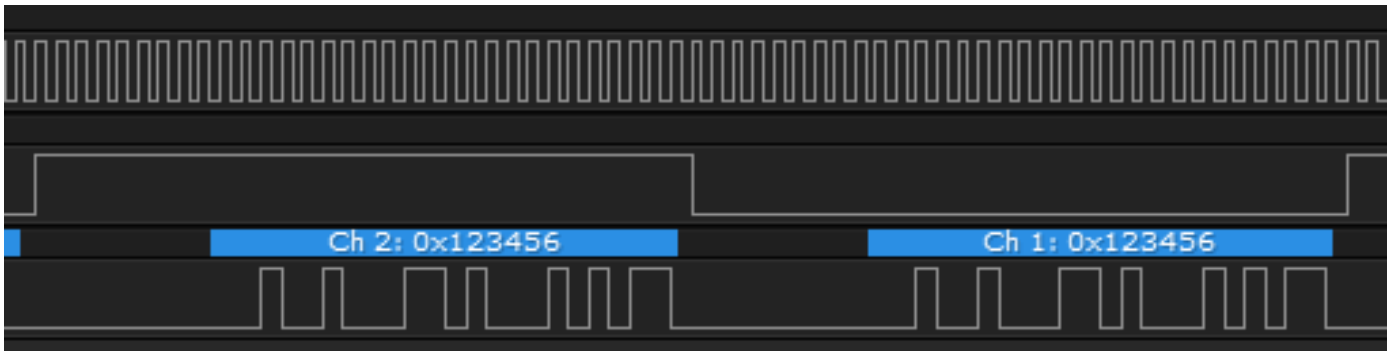

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex] & 0xFFFF0000;
}
```

## Figure G. Left-Justified 16bit format (Slot size is 16)



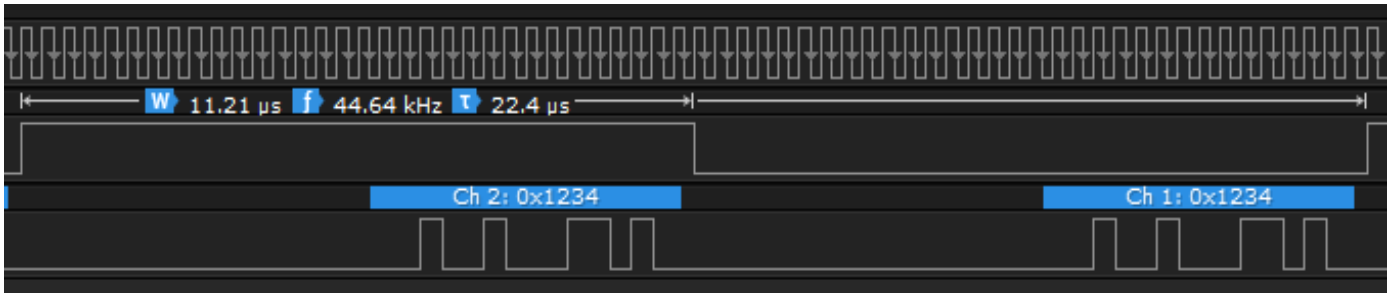

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex] & 0xFFFF0000;
}
```

## Figure H. Right-Justified 24bit format



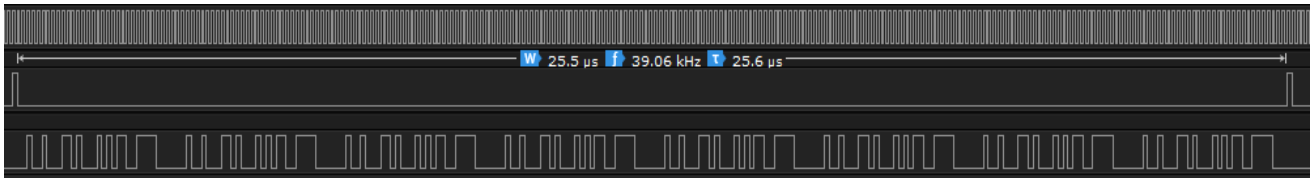

```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = (txBuffer[fifoIndex] >> 8) & 0x00FFFFFF;
}
```

## Figure I. Right-Justified 16bit format




```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = (txBuffer[fifoIndex] >> 16) & 0x0000FFFF;
}
```

## Figure J. TDM, 32bit format, 8 channels




```
uint8_t fifoIndex = 0;
uint32_t txBuffer[8] = {0x12345678, 0x12345678, 0x12345678, 0x12345678,
                        0x12345678, 0x12345678, 0x12345678, 0x12345678};
for (fifoIndex = 0; fifoIndex < 8; fifoIndex++)
{
    SAI0->TDR[0] = txBuffer[fifoIndex];
}
```

***How to Reach Us:***

**Home Page:**
nxp.com

**Web Support:**
nxp.com/support



Document Number: AN12202
Rev. 0
11/2018