AN5086
Application note

I²S protocol emulation on STM32L0 Series microcontrollers using a standard SPI peripheral

Introduction

The I²S protocol is widely used to transfer audio data from a microcontroller / DSP to an audio codec in order to play audio content (stored in a memory) or, to capture analog sound (from a microphone).

This application note describes how standard SPI (Serial Protocol Interface) and TIMER peripherals are able to emulate an I²S interface. This solution can be useful when the application does not allow the use of the I²S feature due to physical constraints (small package) or because it is already used (full-duplex audio exchange).

As an application example based on the STM32L0 Series products, we propose a common use case where data is received asynchronously from the UART peripheral and transferred synchronously to an external IC (integrated circuit) with the proposed emulated I²S master interface.

Contents

1 I²S peripheral emulation principle 6
1.1 I²S digital audio protocol 7
1.1.1 Serial Protocol Interface peripheral (SPI) 8
1.1.2 Timer peripheral 9
1.1.3 Results 10

2 Application description 11
2.1 48 kHz application: CD-audio quality playback 11
2.2 8 kHz application: radio AM quality 12

3 System implementation 13
3.1 System implementation with embedded memory 13
3.1.1 Presentation 13
3.1.2 Memory mechanism implementation 13
3.2 System implementation with UART 14
3.2.1 Asynchronous audio transfer with UART 14
3.2.2 Firmware implementation 15
3.2.3 UART peripheral parameters with flow control 16
3.2.4 Host side: computer interface using Tera Term software 16
3.2.5 WAV file header removal script 17
3.2.6 UART Interface from MCU 18

4 Audio results using I²S emulated peripheral 19
4.1 Audio usual measurements 20
4.1.1 Digital amplitude (full-scale) 20
4.1.2 Offset 20
4.1.3 Frequency response 20
4.1.4 Dynamic range 20
4.1.5 Total harmonic distortion 21
16-bit truncation 21
4.2 Constant audio content 21
4.3 Sinusoidal audio content 21
4.3.1 Amplitude and offset measurement 22
4.3.2 Frequency response 22

4.3.3 Dynamic range (THD+N) 23
4.3.4 Total harmonic distortion (THD) 23

5 Conclusion 24

Appendix A WAV format to UART Powershell script. 25

6 Revision history 27

List of tables

Table 1. STM32 peripheral constraints to emulate I²S protocol 7
Table 2. SPI peripheral signals 8
Table 3. I²S protocol frequency with HSE clock 15
Table 4. I²S protocol frequency with MSI clock 15
Table 5. Theoretical sine distortion versus data bit width (1 kHz 0 dBFS, FS = 48 kHz) 21
Table 6. DC offset and maximum amplitude measurement 22
Table 7. Document revision history 27

List of figures

Figure 1. Overview of the master transmitter I2S emulator . . . 6
Figure 2. Overview of the master receiver I2S emulator . . . 6
Figure 3. I²S protocol waveform . . . 7
Figure 4. I²S master/slave configurations . . . 8
Figure 5. System overview with timing diagram . . . 9
Figure 6. Oscilloscope acquisition with I2S signals . . . 10
Figure 7. Overview of the application note example . . . 11
Figure 8. 48 kHz audio system . . . 12
Figure 9. 8 kHz audio system . . . 12
Figure 10. I²S measurement bench . . . 13
Figure 11. SPI DMA configuration . . . 13
Figure 12. UART timing constraints and packet lost using flow control . . . 14
Figure 13. SPI/UART DMA update mechanism . . . 15
Figure 14. Host serial port configuration . . . 16
Figure 15. Sending formated wav file with the Tera term send file command . . . 17
Figure 16. WAV file header removal script execution window (FS=48kHz) . . . 17
Figure 17. Host/client UART exchange mechanism . . . 18
Figure 18. Client UART TX messages during host/client exchange . . . 18
Figure 19. I²S measurement bench . . . 19
Figure 20. Audio measurement interface (Audio Precision instrument) . . . 19
Figure 21. Maximum amplitude measurement (audio analyzer measure) . . . 22
Figure 22. Frequency variation relative to 1 kHz (audio analyzer measure) . . . 22
Figure 23. THD+N level variation (audio analyzer measure) . . . 23
Figure 24. THD level variation (audio analyzer measure) . . . 23

1 I²S peripheral emulation principle

When the I²S feature (included in the SPI peripheral) is not available or is already used (full-duplex or multiple audio codec system), it is possible to emulate the I²S protocol master or slave using standard SPI and timer peripherals. This application note describes the emulation of a master I²S interface.

Figure 1. Overview of the master transmitter I²S emulator

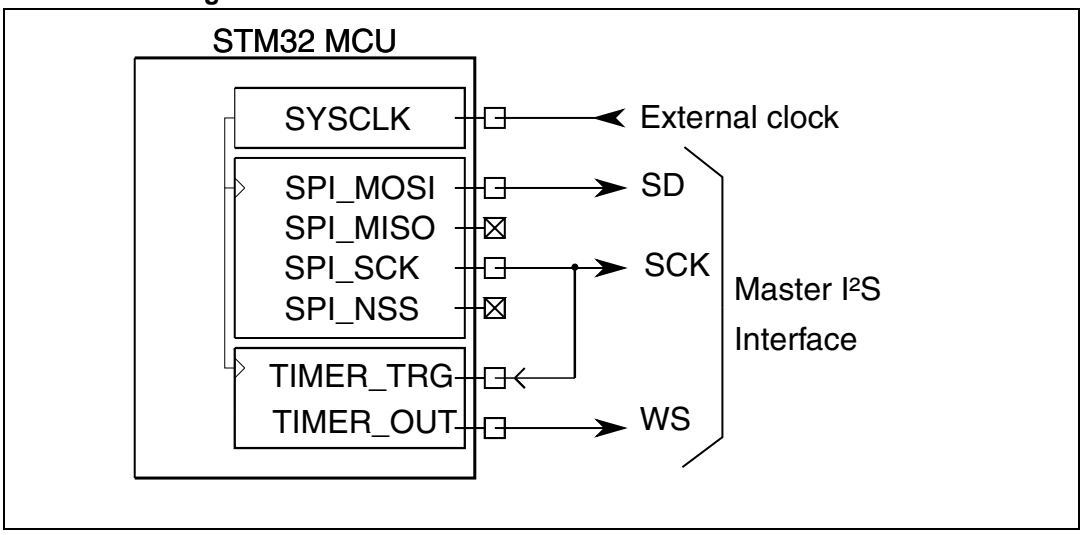


Figure 2. Overview of the master receiver I²S emulator

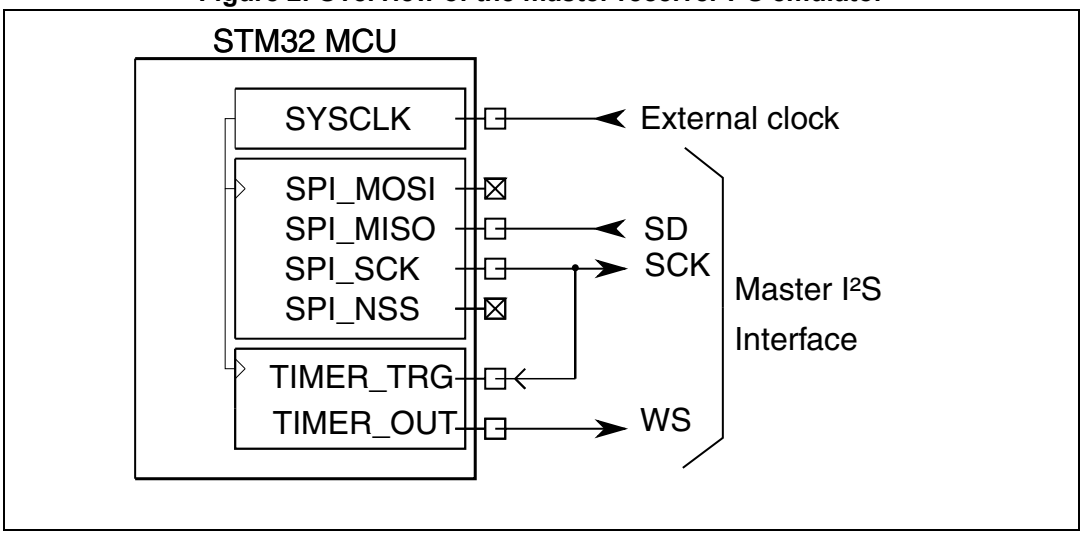


Table 1 describes the list of required peripherals in order to fully emulate an I²S interface:

Table 1. STM32 peripheral constraints to emulate I²S protocol

Purpose	Peripheral	Comments
System clock	HSI/MSI or HSE	Master clock source for I²S emulated peripheral. The I²S protocol frequency is derived from this clock
I²S serial data and clock	SPI (standard)	I²S protocol SD and SCK signal generation is based on SPI SCK and SPI MOSI signals. Please note that the I²S emulator could be also used in a slave configuration using the SPI MISO input pin.
I²S frame rate	TIMER	The timer peripheral should have the following features: – extern ETR input pin – 2 channels – slave configuration to correctly generate the I²S WS signal

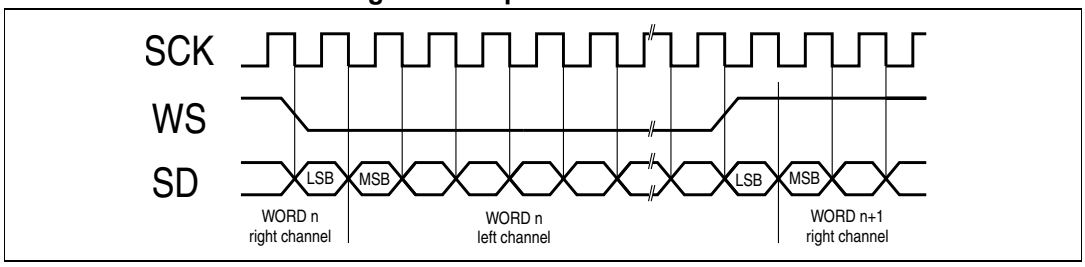
1.1 I²S digital audio protocol

The I²S interface (Inter-IC-Sound) is a synchronous serial bus interface standard which is used to interconnect digital audio devices. I²S can be used to transmit audio data in little-endian format (MSB first) and its rate is synchronized with the falling clock edge (SCK signal) for the transmitter, and with the rising edge for the receiver. Notice that separate clock and data result in low jitter.

The data (inherited from the Compact-Disc audio format) represents stereo digital sound, so each sample contains two words, the right-channel sample and the left-channel sample. Instead of using two data channels, multiplexing is performed by transmitting each word over half a sampling period, which doubles the sampling rate, and makes it possible to transmit two words per period.

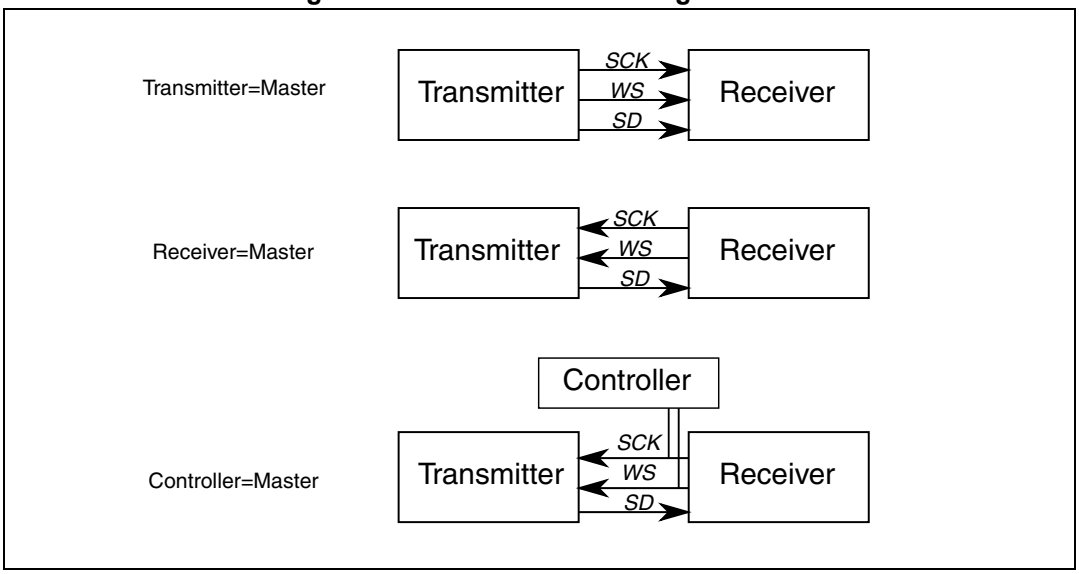
A control signal, WS (Word Select), is then used to determine if the word being sent is the right or left one. This signal also determines both the beginning and the end of the data; there is no need to fix the data length. Receiver and transmitter data lengths can therefore be different, as well as the right and left data lengths. WS is synchronized on the falling edge of SCK and precedes the MSB by one SCK period in order to allow enough time for store and shift operations.

Figure 3. I²S protocol waveform



As in most communication protocols, there must be a master and a slave. The master provides and controls the SCK clock and the WS signal, while the slave only sends or receives data. The master can be the receiver, the transmitter or a third element:

Figure 4. I²S master/slave configurations



1.1.1 Serial Protocol Interface peripheral (SPI)

The internal Standard Peripheral Interface, or SPI, provides a simple communication interface allowing the micro-controller to communicate with external devices. This interface is highly configurable to support many standard protocols. SCK(SPI) can be directly used as SCK(I²S), and MOSI(SPI) as SD (I²S).

The third signal, WS (Word Select), requires the use of the timer peripheral because it is synchronous and delayed with respect to the SCK signal.

Table 2. SPI peripheral signals

Signal name	SPI SCK	MOSI	SPI nSS	MISO
Signal description	SPI output serial clock	Master Output Slave Input (data)	SPI output Slave select	Master Input Slave Output (data)
I²S emulation	SCK	SD	-	-
Signal direction	output	output	output	input

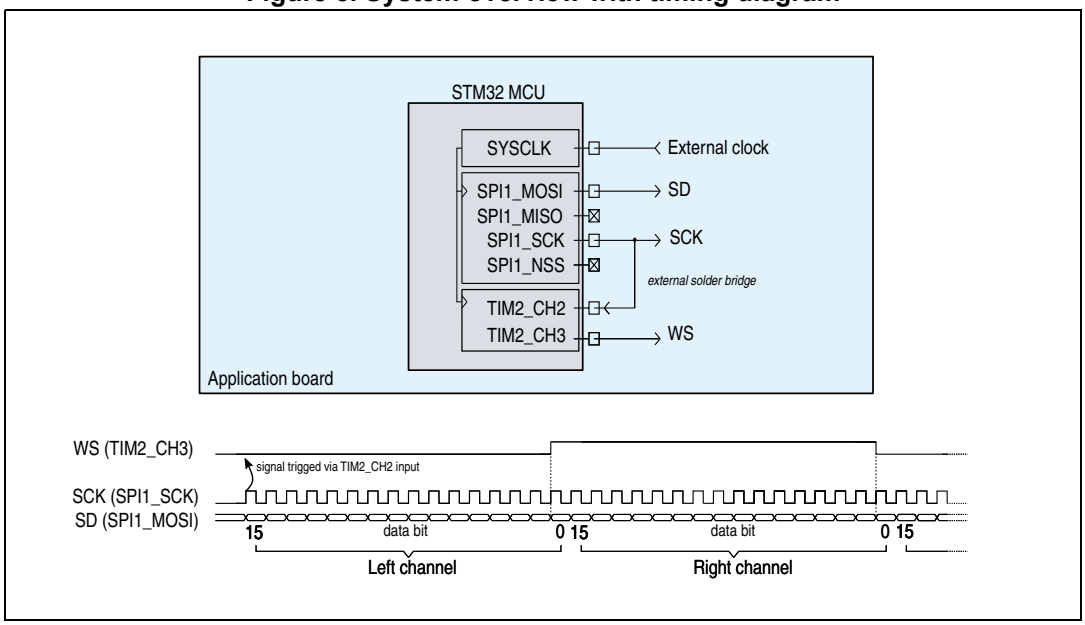
1.1.2 Timer peripheral

The STM32 embeds multiple timers providing timing resources for hardware tasks related to I/Os. The timers can generate waveforms on their outputs and react to external events on their inputs. The timers are very versatile and provide multiple operating modes to offload repetitive and time-critical tasks from the CPU, while minimizing interfacing circuitry needs. The timer kernel consists of a 16-bit up-counter, coupled with an auto-reload register to program the counting period, and a repetition counter to adjust the counter roll-over interrupt rate. The timer is an adequate resource to generate the WS (Timer CH3) signal which is synchronous and delayed with respect to the SCK signal. The timer is internally clocked by SYSCLK (the same clock source is used by the SPI peripheral), and its start trigger (timer CH2) is connected to the SCK signal through the STM32 GPIOs.

The polarity of the timer, the counter period and its initial values are set to match the I²S protocol configuration:

- period value = 31
- pulse value = 15
- trigger polarity = rising
- output comparator polarity = TIM_OCPOLARITY_HIGH

Figure 5. System overview with timing diagram

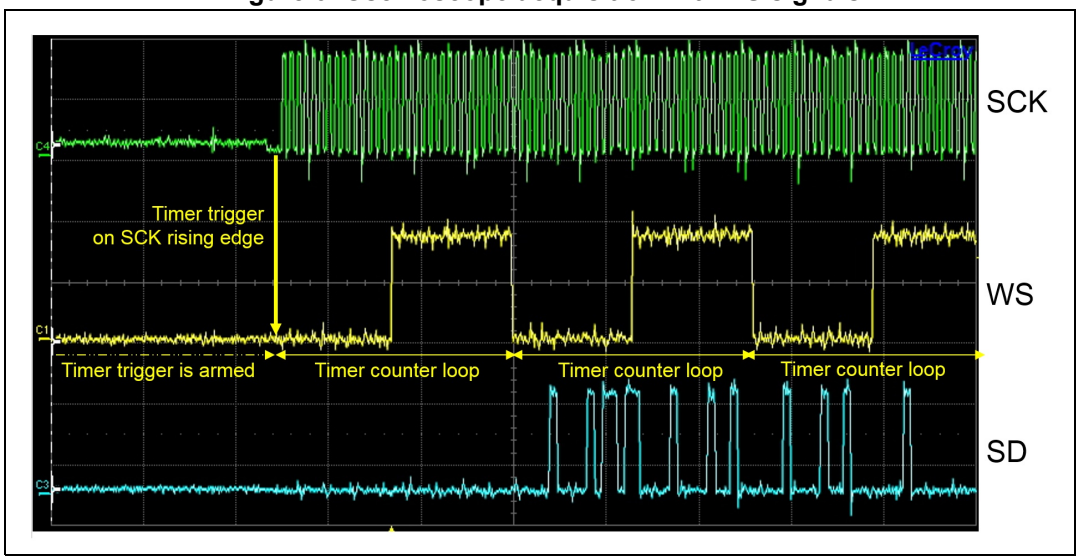


1.1.3 Results

Using a standard oscilloscope, we verify the signal waveforms and timings. With this methodology, we also verify that the peripheral timings are correct. We have also verify the firmware compatibility with the following list of IS modes:

- I²S standard - stereo channels
- I²S left justified – stereo channels
- I²S right justified – stereo channels

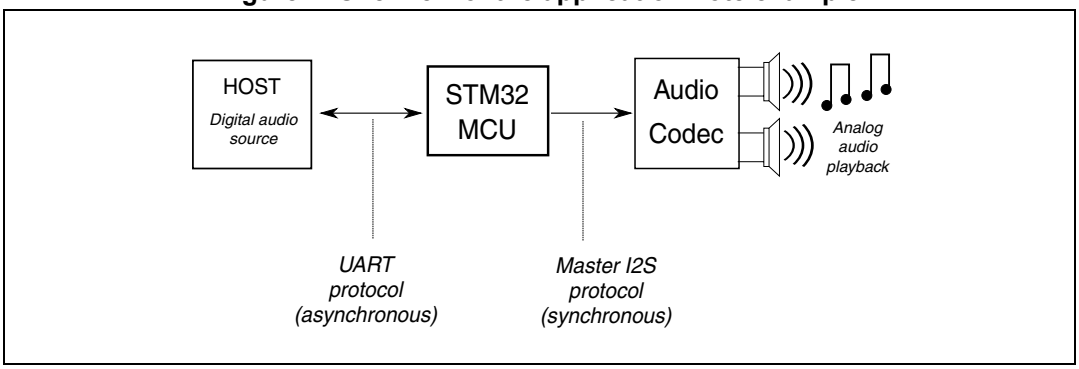
Figure 6. Oscilloscope acquisition with I²S signals



2 Application description

This section describes an application proposal which transfers stored digital audio content from a host to an audio codec through the UART and I²S protocols managed by the STM32 MCU.

Figure 7. Overview of the application note example



The following application examples are intended to be relevant to the targeted market.

2.1 48 kHz application: CD-audio quality playback

The use of an I²S interface at 48 kHz sample rate is a common case for high-end audio applications connected to an audio codec, to provide a excellent audio quality (equivalent to Audio Compact-Disc) such as MP3 players, portable speakers or wireless headsets.

- the system sample frequency is 48 kHz (SAMPLERATE)
- the system number of bits is 16 (BITWIDTH)
- the system number of channels is 2 (stereo)

From these numbers, we can calculate the frequency of the I²S protocol bit rate equal to :

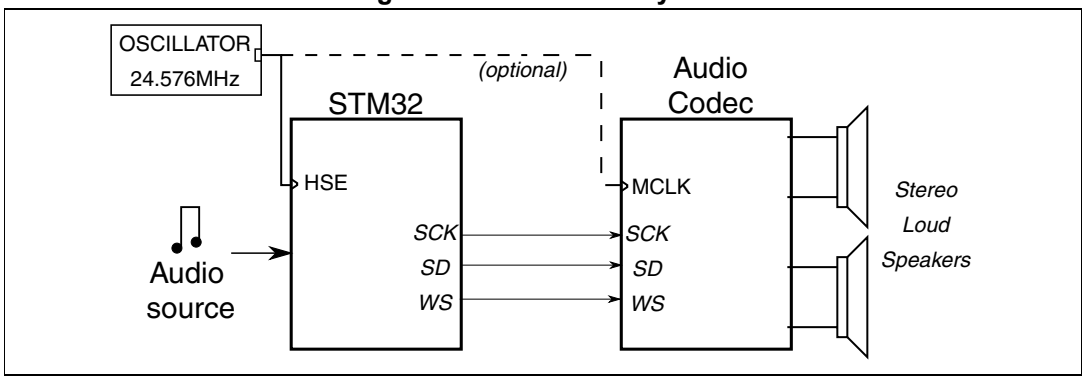
Equation 1- Calculation of the I²S interface bit rate

$$\text{I2S BITRATE} = \text{SAMPLE RATE} \times \text{WIDTH} \times \text{CHANNEL} = 48000 \times 16 \times 2 = 1.536 \text{ Mbit/s}$$

Audio systems use specific frequency values for compliance with existing standards (11 kHz, 44.1 kHz, 48 kHz and so on). Usually, an external oscillator is present in the system to provide an accurate clock with low jitter.

In this particular case (48 kHz), we could choose MCLK = 24.576 MHz from an external oscillator for an oversampling ratio of 16 with the I²S protocol bit rate.

Figure 8. 48 kHz audio system



2.2 8 kHz application: radio AM quality

In low-cost applications, we recommend the use of a lower I²S bit rate and the STM32 internal frequency oscillator to minimize the power consumption of the system and the cost of the bill of materials. This application drives a Class-D audio amplifier using its I²S protocol.

- system sample frequency is 8 kHz
- system number of bits is 16 (BITWIDTH)
- system number of channels = 2
- I²S bit rate = 8 kHz x 16 x 2 = 256 Kbits/s

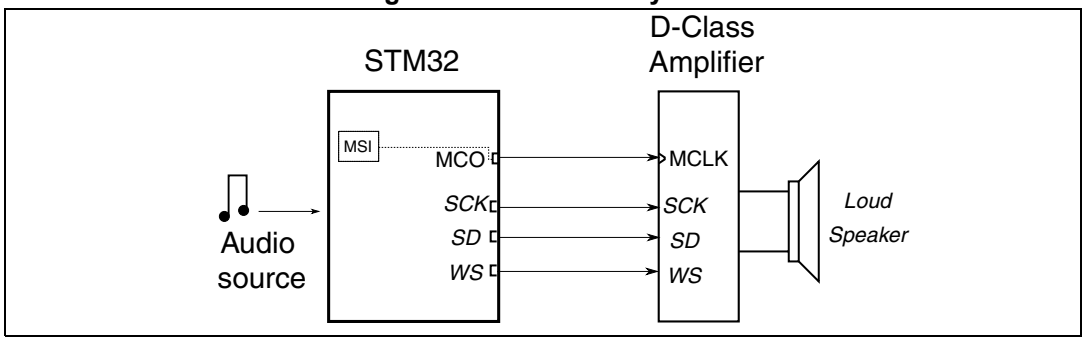
For this example, based on the STM32L0x, we could either use MSI (4.194 MHz) or HSI (16 MHz) oscillators, but neither would achieve the exact and correct system clock frequency to match a multiple of 256 kHz (the I²S bit rate). The alternative, with an extra cost, would be to add an external oscillator to the system.

Equation 2- Calculation of frequency mismatch using MSI as clock source

$$MCLK = \frac{4{,}194}{16} = 262.125 \text{ kHz (versus 256 kHz)}$$

MCLK = 4.194 MHz, giving SCK = 4.194/16 = 262.125 kHz compared to 256 kHz which is an absolute frequency error of 2.3%.

Figure 9. 8 kHz audio system



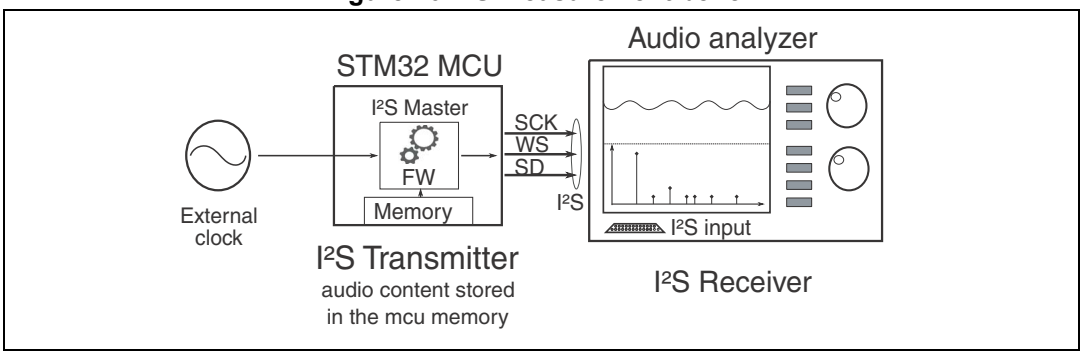
3 System implementation

3.1 System implementation with embedded memory

3.1.1 Presentation

In order to verify the correct behavior of the emulated I²S master interface, it is connected to a digital audio analyzer with a parametric I²S slave interface. As a first step, we analyze a fixed constant value and the stored sinusoidal waveform (16-bit stereo). Secondly, we analyze the same audio signals coming from the UART peripheral.

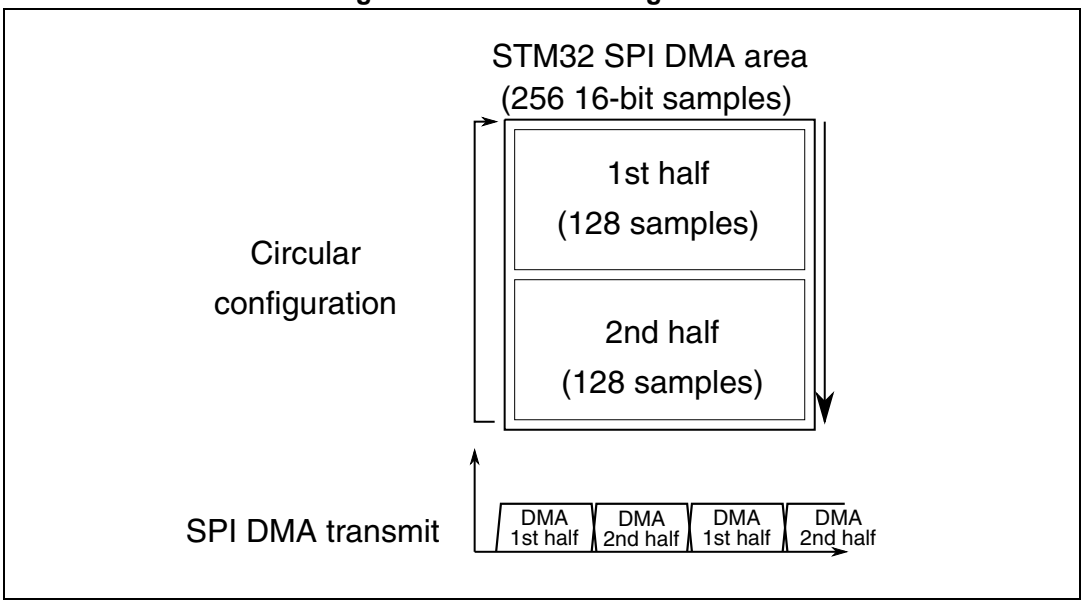
Figure 10. I²S measurement bench



3.1.2 Memory mechanism implementation

The DMA is configured in circular mode during the SPI transmit. The SPI peripheral continuously reads the buffered audio data:

Figure 11. SPI DMA configuration



3.2 System implementation with UART

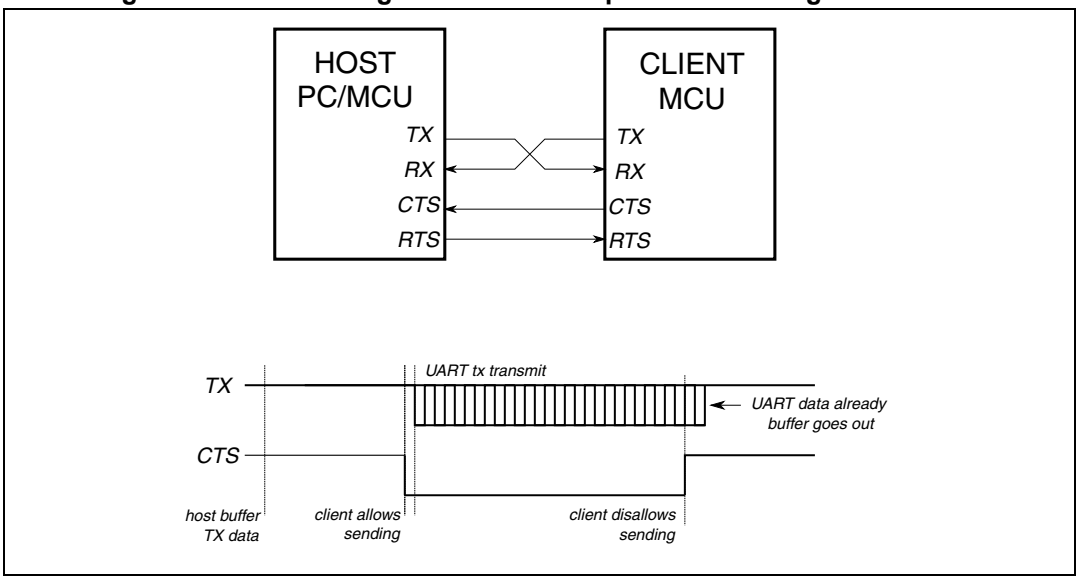
The previous simple test bench is useful for the measurement of audio performance with periodic sinusoidal signals. It also allows verification of the correct behavior of the I²S emulator. In a standard audio application, the playback content varies; an input interface should be added in our application to provide the audio stream. We have chosen a very common interface to transfer the audio content, the universal asynchronous receiver transmitter (UART). Most of the STM32 micro-controller series includes a UART peripheral that can be used as described in this application note.

3.2.1 Asynchronous audio transfer with UART

This application uses the UART interface to receive an audio digital flow asynchronously. The data exchange mechanism is managed with UART Flow control to avoid any packet loss.

The UART interface (Universal Asynchronous Receiver Transmitter) is a simple 2-wire standard interface. Two additional wires RTS (Request To Send) and CTS (Clear To Send) can be used in order to add the 'Flow Control' feature which manages data exchange with packets between devices to avoid any data loss (excluding data already buffered).

Figure 12. UART timing constraints and packet lost using flow control



This example transfers audio data received asynchronously from UART to SPI peripheral configured to transmit continuously I^2S protocol. A specific and precise mechanism should be implemented to avoid any data loss during the DMA exchange.

3.2.2 Firmware implementation

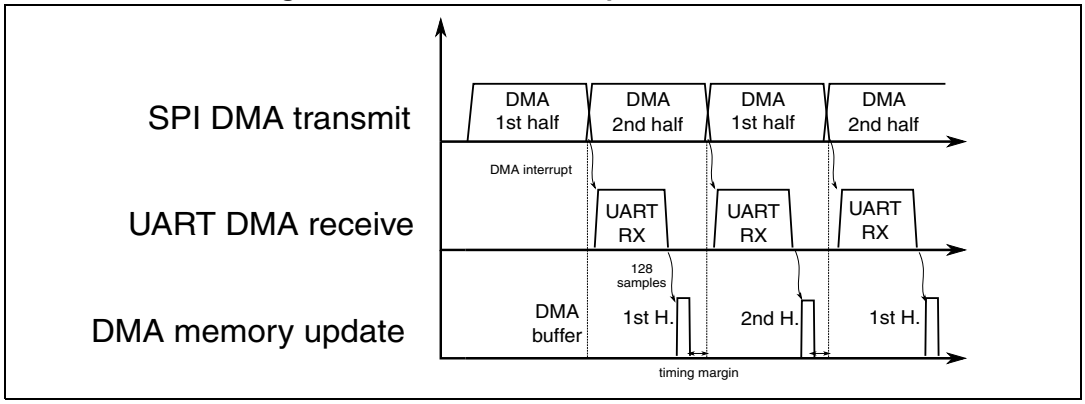
The UART receive mechanism should interleave each DMA half-block reading according to the following principle:

- The UART receive function is called when an SPI DMA interrupt occurs.

The system should be configured correctly to retrieve all data (from the UART) before the SPI DMA half-buffer transmit is completed.

This application uses a ratio of 1.76: the UART baud rate is 460800 and the SPI sample rate is 262.125 kHz

Figure 13. SPI/UART DMA update mechanism



This specific application requires a trade-off between the SPI and UART buffer sizes and their running frequencies.

A large buffer size is used: 256 16-bit stereo samples (equivalent to 1024 8-bit raw samples) to give enough time to call, execute and complete the UART receive call function. The drawback

Furthermore, this example, based on the STM32L031 allows a maximum system clock frequency of 32 MHz, and a minimum multiplication ratio of 2 for the SPI peripheral. *Table 3* presents the optimum system configuration.

Table 3. I²S protocol frequency with HSE clock

I²S SCK frequency (MHz)	SPI Clock Prescaler	HSE frequency (MHz)	Comments
1.536	x16	24.576	Optimal configuration

In this application, it is mandatory to use an external clock from the HSE input at the specific value 24.576 MHz to get an I²S protocol working at 48 kHz. This external oscillator can be used as the master clock source for the complete system

Table 4. I²S protocol frequency with MSI clock

MSI frequency (kHz)	SPI Clock Prescaler	I²S SCK frequency (kHz)	Comments
4194	x16	262.125	-

3.2.3 UART peripheral parameters with flow control

The STM32 UART peripheral has an optimized setup regarding the hardware clock SYSCLK. There is a minimum factor ratio of x8 between the microcontroller clock and the output baud rate of the UART peripheral.

```
Stop Bit = One Stop bit
Parity = No parity
BaudRate = 460800 baud
Hardware flow control enabled(RTS and CTS signals) */
UartHandle.Instance = USARTx;
UartHandle.Init.BaudRate = 460800;
UartHandle.Init.WordLength = UART_WORDLENGTH_8B;
UartHandle.Init.StopBits = UART_STOPBITS_1;
UartHandle.Init.Parity = UART_PARITY_NONE;
UartHandle.Init.HwFlowCtl = UART_HWCONTROL_RTS_CTS;
UartHandle.Init.Mode = UART_MODE_TX_RX;
```

3.2.4 Host side: computer interface using Tera Term software

The host configuration should be aligned with the STM32 UART peripheral setup.

Figure 14. Host serial port configuration

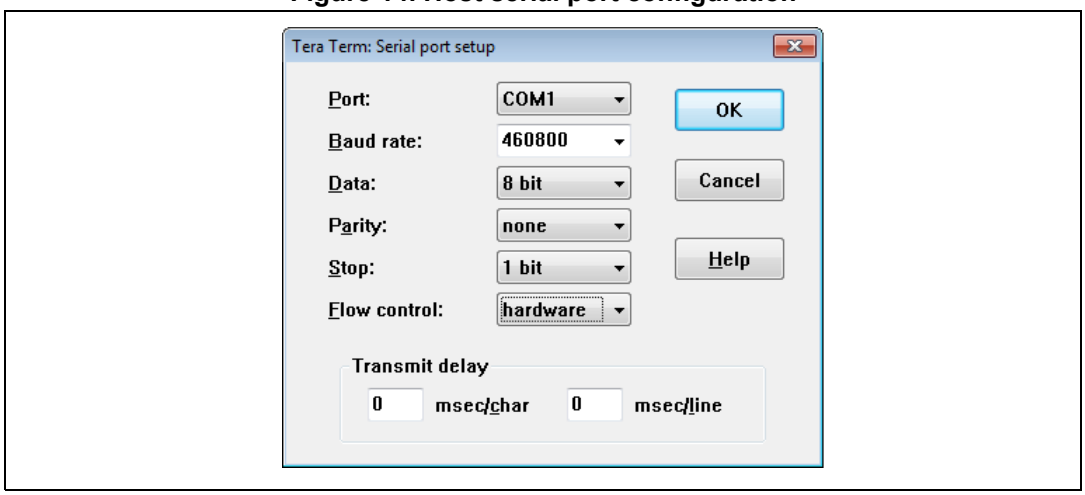
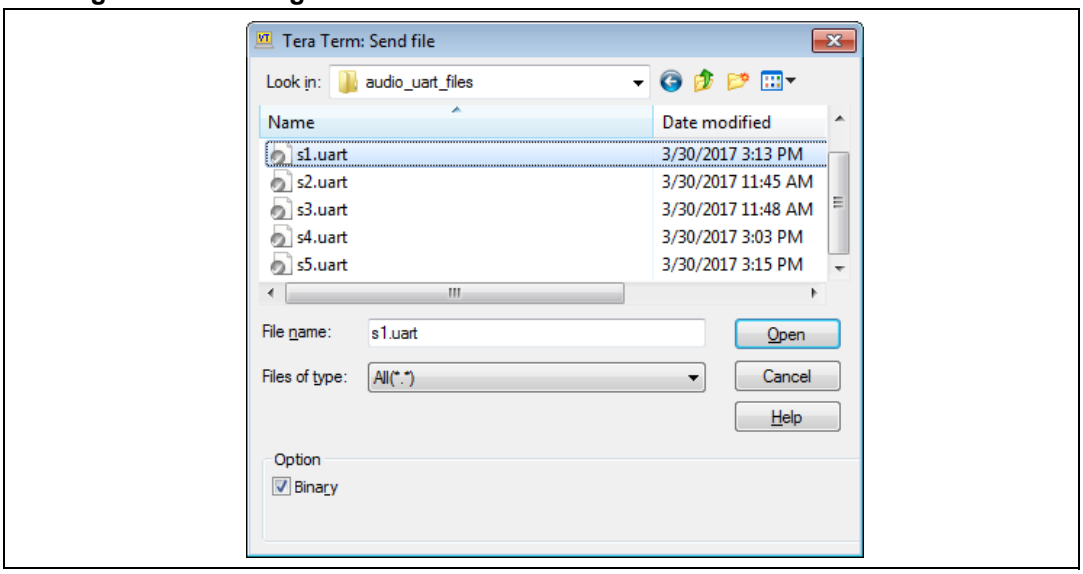


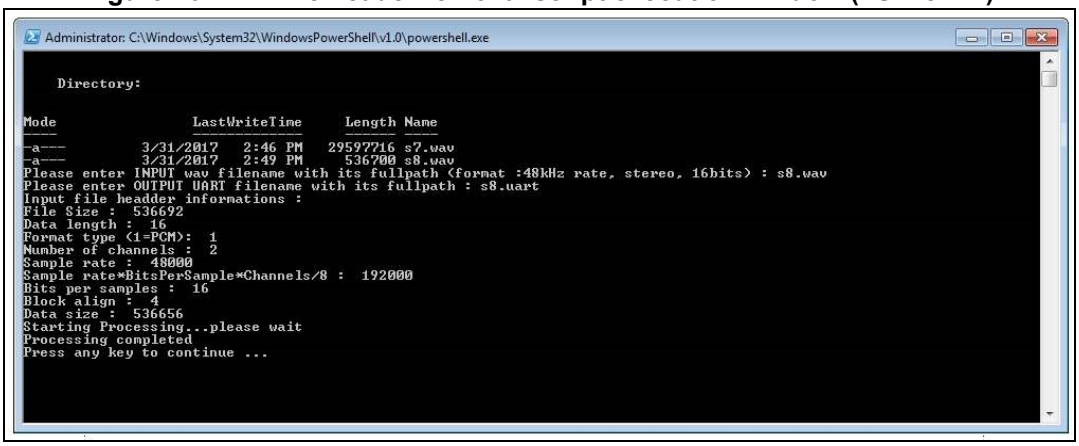
Figure 15. Sending formated wav file with the Tera term send file command



3.2.5 WAV file header removal script

The file standard Waveform Audio File Format (WAV) could be easily used to as an audio source. It is convenient either to test the audio system with sinusoidal tones or to playback the music. The WAV data format is identical to the I²S format, the only drawback is the need to remove the file header description that does not contain audio data. This preprocessing could be done using a simple script as shown in *Appendix A: WAV format to UART Powershell script*.

Figure 16. WAV file header removal script execution window (FS=48kHz)



3.2.6 UART Interface from MCU

Because the UART is a bidirectional interface, we would recommend integrating into the MCU firmware a simple Graphical User Interface (GUI) using ASCII characters, in order to manage the audio data exchange between the host and the client. *Figure 17* shows such a simple interface display. The firmware detects the UART input buffer state (idle, receiving on going, transfer completed) to manage the I²S protocol.

Figure 17. Host/client UART exchange mechanism

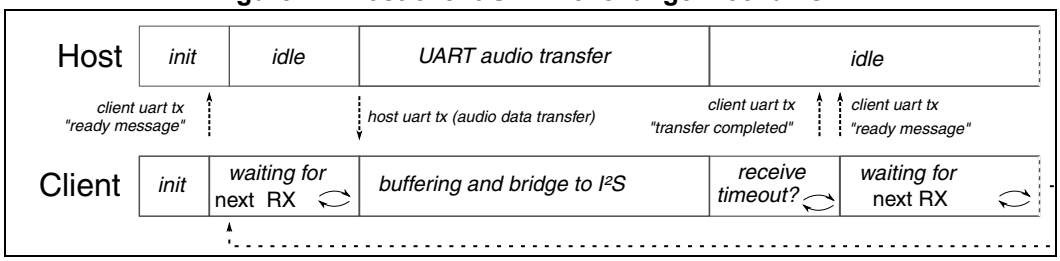
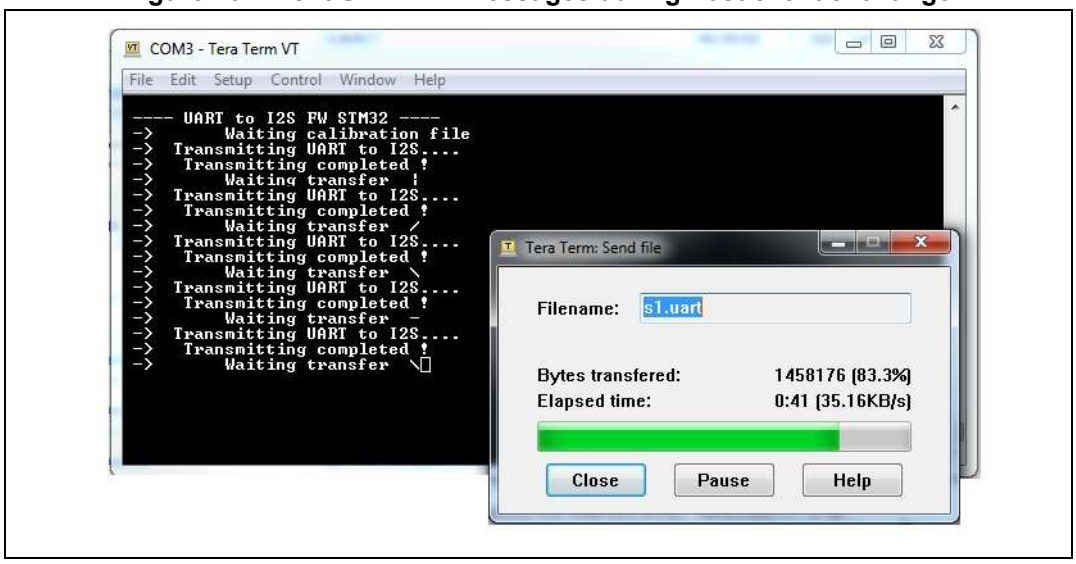


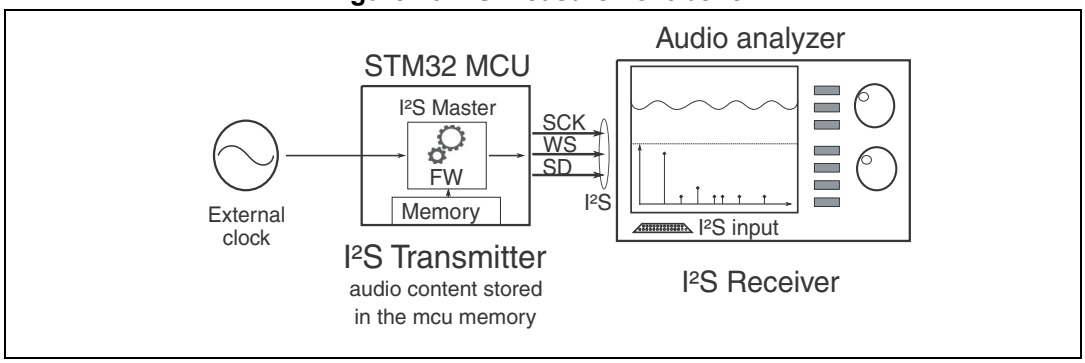
Figure 18. Client UART TX messages during host/client exchange



4 Audio results using I²S emulated peripheral

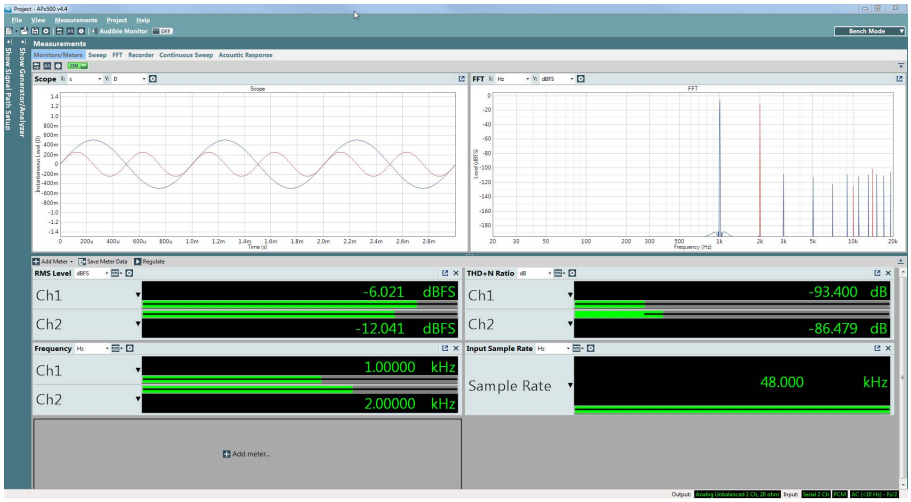
The analysis of the I²S audio content is performed using a commercial digital audio analyzer as shown in *Figure 19*.

Figure 19. I²S measurement bench



The audio measurements are performed using the audio analyzer and its interface as shown in *Figure 20*.

Figure 20. Audio measurement interface (Audio Precision instrument)



4.1 Audio usual measurements

4.1.1 Digital amplitude (full-scale)

In the audio digital domain, the amplitude of the signal is proportional to the Full-Scale (FS) which represents the smallest and largest signal amplitude: in our application 0x0000 (0 FS) and 0x7FFF (1 FS). If we calculate the signal amplitude in dBFS, it is common to use the peak amplitude and not the root-mean-square to compute the level in dBFS.

Example 1

- digital sinusoidal signal with a peak amplitude equal to 16384 lsb
- Its amplitude in FS is 16384/32767 = 0.5 FS (where 32767 is the maximum signal level)
- We can convert this value into decibels: 20 x $\log_{10}(0.5)$ = -6 dBFS

4.1.2 Offset

The offset measurement is the mean value of the output signal (numeric domain). Normally, the system should deliver an output signal around the mid-scale of the system. In our application, the mid-scale values for the I²S format is 0x0000 (signed 16-bit).

4.1.3 Frequency response

This measurement is defines the frequency range of the audio system. Most of the time, the amplitude of the output level at 1 kHz is used as a reference to be compared to the measured amplitude. We measure the amplitude of the output signal with a series of frequencies (from 20 Hz to 20 kHz with FS = 48 kHz, or from 20 Hz to 4 kHz with FS = 8kHz) and calculate the attenuation in dB versus the reference (1 kHz amplitude). An audio system with a flat frequency (+/- 1 dB) does not alternate the audio signal and its content, which guarantees good audio rendering.

4.1.4 Dynamic range

The dynamic range measurement is defined by the ratio in dB between the maximum output level and its noise level (minimum output signal).

4.1.5 Total harmonic distortion

The THD represents the linearity performance of the application. During this test, a single tone is applied to the input of audio system. Mathematically, we extract the harmonics of the input fundamental, and calculate the ratio in dB or percentage between the main tone and its distortion (harmonics).

16-bit truncation

Because of the use of 16-bit data width, the generated sinusoidal signal contains a non negligible level of distortion as shown in *Table 5*.

Table 5. Theoretical sine distortion versus data bit width (1 kHz 0 dBFS, FS = 48 kHz)

8-bit	16-bit	24-bit	32-bit
THD = -53 dB	THD = -99 dB	THD = -127 dB	THD = -200 dB

4.2 Constant audio content

We have chosen the following parameters to verify the integrity of the I²S emulated interface. The memory content is set with a constant/DC value to verify:

- signal timings versus I²S specificationg
- glitch detection: long-term acquisition to detect signal
- measurement of the offset and the full-scale

4.3 Sinusoidal audio content

We have chosen the following parameters to verify the integrity of the I²S emulated interface.

- level (dBFS)
- frequency response (Hz)
- distortion/THD (dB)

Before performing analysis of the audio signal transferred via the I²S protocol, the expected performance must be evaluated.

The sinusoidal waveform is computed with a build-in mathematical function. The result is an array of double-precision floating point numbers. We use a 16-bit format so we must perform a truncation operation to reduce the precision of our represented sinusoid. This operation introduces distortion in the resulting signal (visible in *Table 6*).

Using the mathematical analysis of the signal, the expected distortion level is around -93 dB for a level of -6 dBFS

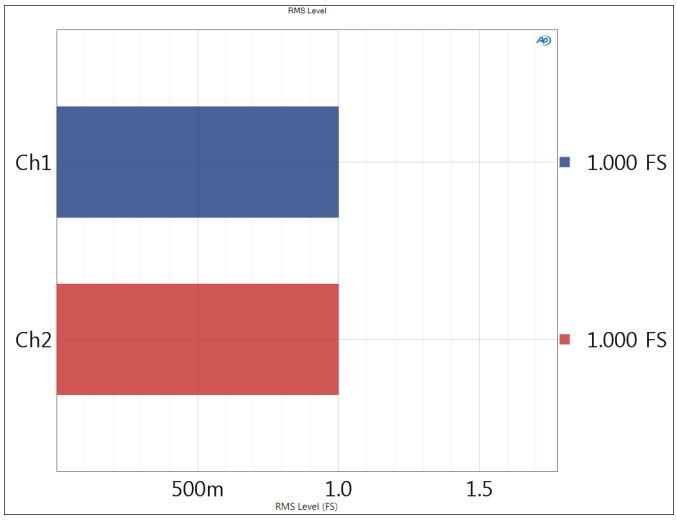
4.3.1 Amplitude and offset measurement

This section shows the results of the offset and maximum amplitude using a sinusoidal signal with 1 kHz frequency.

Table 6. DC offset and maximum amplitude measurement

DC Offset (FS) CH1 and CH2	Maximum amplitude (FS) CH1 and CH2
0FS	1FS

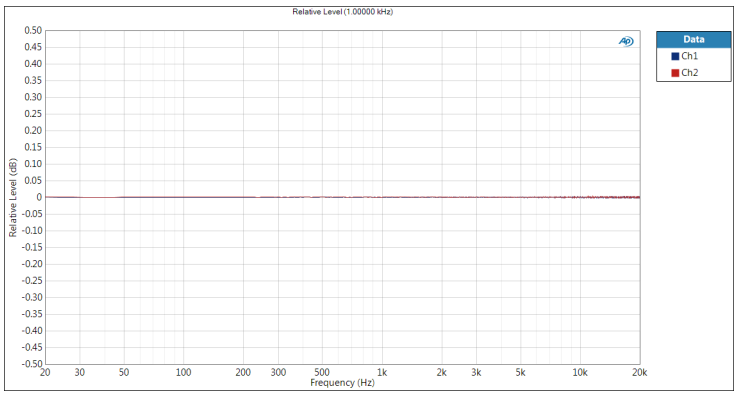
Figure 21. Maximum amplitude measurement (audio analyzer measure)



4.3.2 Frequency response

This section shows the results of the frequency response relative to 1 kHz frequency amplitude from 20 Hz to 20 kHz.

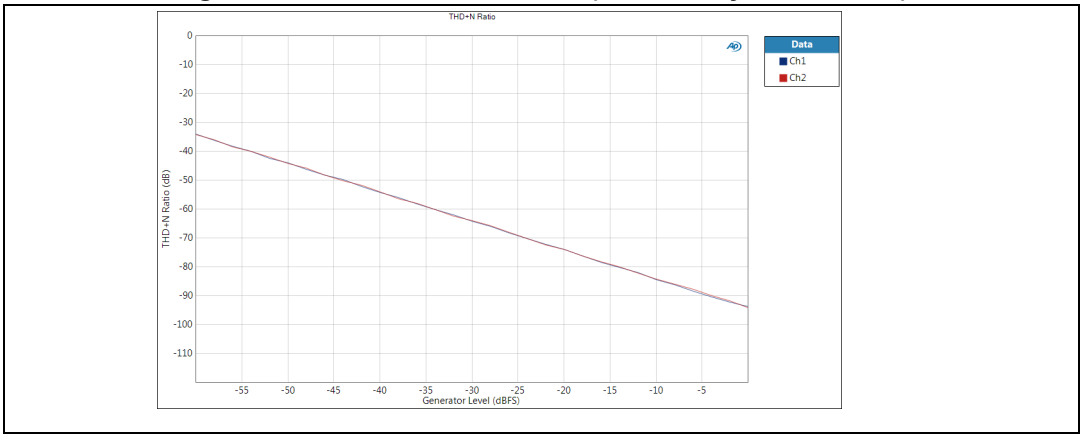
Figure 22. Frequency variation relative to 1 kHz (audio analyzer measure)



4.3.3 Dynamic range (THD+N)

This section shows the results of the total harmonic distortion + noise measurement using a 1 kHz sinusoidal signal with a level variation from -60 dBFS to 0 dBFS.

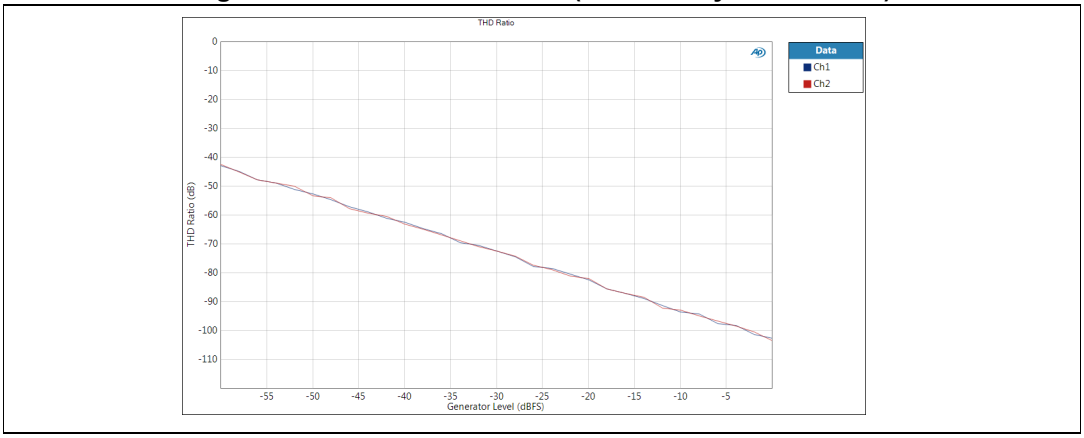
Figure 23. THD+N level variation (audio analyzer measure)



4.3.4 Total harmonic distortion (THD)

This section shows the results of the total harmonic distortion measurement using a 1 kHz sinusoidal signal with a level variation from -60 dBFS to 0 dBFS.

Figure 24. THD level variation (audio analyzer measure)



5 Conclusion

This application note presents ways in which the STM32 peripherals might be combined to create new or advanced features. In this way, it is simple to emulate an I²S peripheral using SPI and TIMER peripherals. Using the same method, it would be quite feasible to emulate a slave I^2S device.

Furthermore, the STM32 DMA peripheral is a very efficient resource for data exchange between peripherals, and easily creates a bridge in the application environment as demonstrated by our UART-to-I²S example.

The solution shared in this document can be used as a starting point for development or improvement your own solution based on the STM32 microcontroller family.

Appendix A WAV format to UART Powershell script

```
dir *.wav 'Displaying the wav file list inside the folder'
$input_file = Read-Host 'Please enter INPUT wav filename with its fullpath
(format: 48kHz rate, stereo, 16bits) '
$output_file = Read-Host 'Please enter OUTPUT UART filename with its
fullpath '
$input_buffer = [System.IO.File]::ReadAllBytes($input_file)

$input_buffer_length=[math]::floor($input_buffer.length/256)
$output_buffer_length=($input_buffer_length)*(256+2);
$input_buffer_length=($input_buffer_length-1)*256;
Write-Host "Input file headder informations:"
Write-Host "File Size: "
($input_buffer[7]*16777216+$input_buffer[6]*65536+$input_buffer[5]*256+$in
put_buffer[4])
Write-Host "Data length: "
($input_buffer[19]*16777216+$input_buffer[18]*65536+$input_buffer[17]*256+
$input_buffer[16])
Write-Host "Format type (1=PCM): "
($input_buffer[21]*256+$input_buffer[20])
Write-Host "Number of channels: " ($input_buffer[23]*256+$input_buffer[22])
Write-Host "Sample rate: "
($input_buffer[27]*16777216+$input_buffer[26]*65536+$input_buffer[25]*256+
$input_buffer[24])
Write-Host "Sample rate*BitsPerSample*Channels/8: "
($input_buffer[31]*16777216+$input_buffer[30]*65536+$input_buffer[29]*256+
$input_buffer[28])
Write-Host "Bits per samples: " ($input_buffer[35]*256+$input_buffer[34])
Write-Host "Block align: " ($input_buffer[33]*256+$input_buffer[32])
Write-Host "Data size: "
($input_buffer[43]*16777216+$input_buffer[42]*65536+$input_buffer[41]*256+
$input_buffer[40])
$output_buffer=@(0) * ($output_buffer_length)
$j=1;$uart_frame_cnt=511;
$riff_header=44;
Write-Host "Starting Processing...please wait"
for($i=0; $i -le ($input_buffer_length); $i++)
{
 $output_buffer[$i+$j]= $input_buffer[$i+$riff_header];
  if($uart_frame_cnt -eq 0)
 {
 $uart_frame_cnt=511;
 $j=$j+2;
 }
 else
 {$uart_frame_cnt=$uart_frame_cnt-1;}
```

```
}
Write-Host "Processing completed"
[io.file]::WriteAllBytes($output_file,$output_buffer) 'Output buffer
writing
Write-Host "Press any key to continue ..."
$x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")
```

6 Revision history

Table 7. Document revision history

Date	Revision	Changes
11-Dec-2017	1	Initial release.