life.augmented

AN3126
Application note

Audio and waveform generation using the DAC in STM32 products

Introduction

This application note provides some examples for generating audio waveforms using the Digital to Analog Converter (DAC) peripheral embedded in STM32 products (see *Table 1*).

A digital to analog converter (DAC) is a device with a function opposite to that of an analog to digital converter, i.e. it converts a digital word to a corresponding analog voltage.

The STM32 DAC module is a 12-bit word converter, with up to three output channels to support audio functions.

The DAC can be used in many audio applications such as security alarms, Bluetooth® headsets, talking toys, answering machines, man-machine interfaces, and low-cost music players.

The STM32 DAC can also be used for many other purposes, such as analog waveform generation and control engineering.

This application note is organized in two main sections:

- *Section 1* describes the main features of the STM32 DAC module.
- *Section 2* presents two examples.
  - In the first example, the DAC is used to generate a sine waveform
  - In the second example, the DAC is used to generate audio from .WAV files

Table 1. Applicable products

Type	Product categories
Microcontrollers	STM32 32-bit Arm Cortex MCUs
Microprocessors	STM32 Arm Cortex MPUs

Contents

List of tables

List of figures

1 DAC main features

1.1 Overview

STM32 products, based on Arm®[a] Cortex® cores, integrate DACs with different configurations and features:

- one to three output channels
- noise waveform generation
- triangular waveform generation
- DMA under run flag
- dedicated analog clock

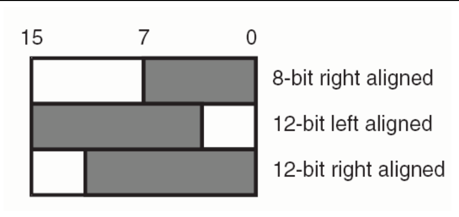
For the different DAC configurations refer to the product datasheets and to reference manuals. Additional info can be found in AN4566 "*Extending the DAC performance of STM32 microcontrollers*". All these documents are available on *www.st.com*.

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.2 Data format

As shown in *Figure 1*, the DAC accepts data in three integer formats: 8-bit (the LS byte of the data hold register), 12-bit right aligned (the twelve LS bits of the data hold register) and 12-bit left aligned (the twelve MS bits of the data hold register).

Figure 1. DAC data format



The analog output voltage on each DAC channel output is determined by the equation

$$DAC_{Output} = V_{REF} \times DOR / 4096$$

1.3 Dual channel mode

Note: This feature is supported only for products that embed at least two DACs.

The DAC has two output channels, each with its own converter. In dual DAC channel mode, conversions can be done independently or simultaneously.

When the DAC channels are triggered by the same source, both channels are grouped together for synchronous update operations and conversions are done simultaneously.

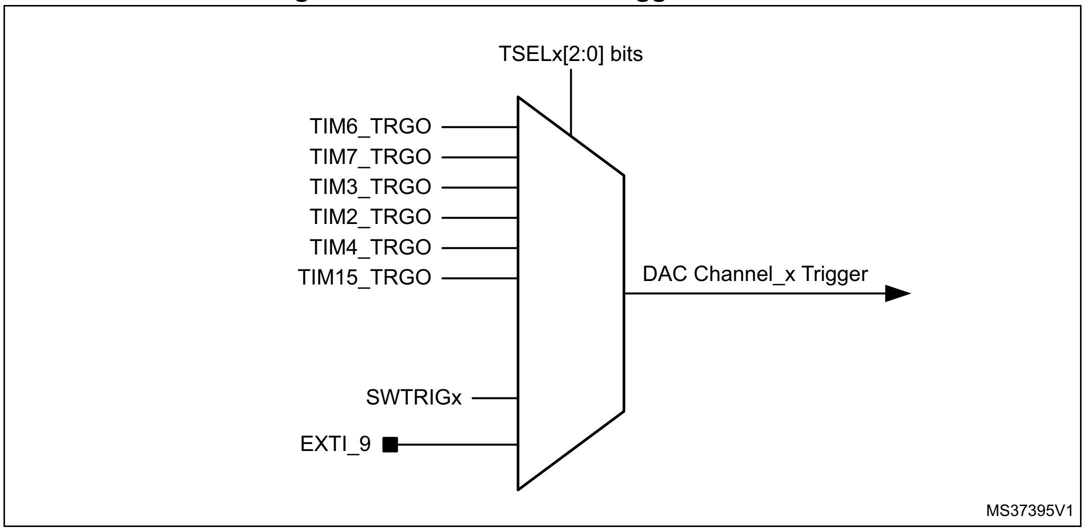
1.4 Dedicated timers

In addition to the software and external triggers, the DAC conversion can be triggered by different timers.

TIM6 and TIM7 are basic timers and are intended for DAC triggering.

Each time a DAC interface detects a rising edge on the selected timer trigger output (TIMx_TRGO), the last data stored in the DAC_DHRx register is transferred to the DAC_DORx register (an example for STM32F100x is given in *Figure 2*).

Figure 2. STM32F10x DAC trigger channels



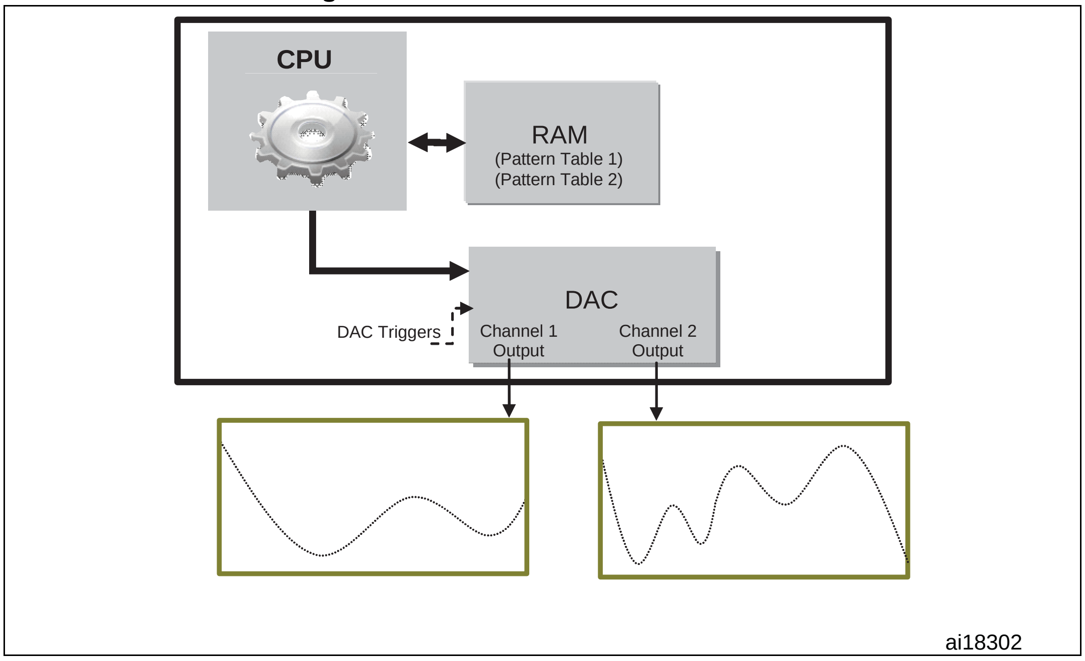
1.5 DMA capabilities

The STM32 products have at least one DMA module with multiple channels (streams).

Each DAC channel (stream) is connected to an independent DMA channel. As an example, for STM32F10x microcontrollers, DAC channel1 is connected to the DMA channel3 and DAC channel2 is connected to DMA channel4.

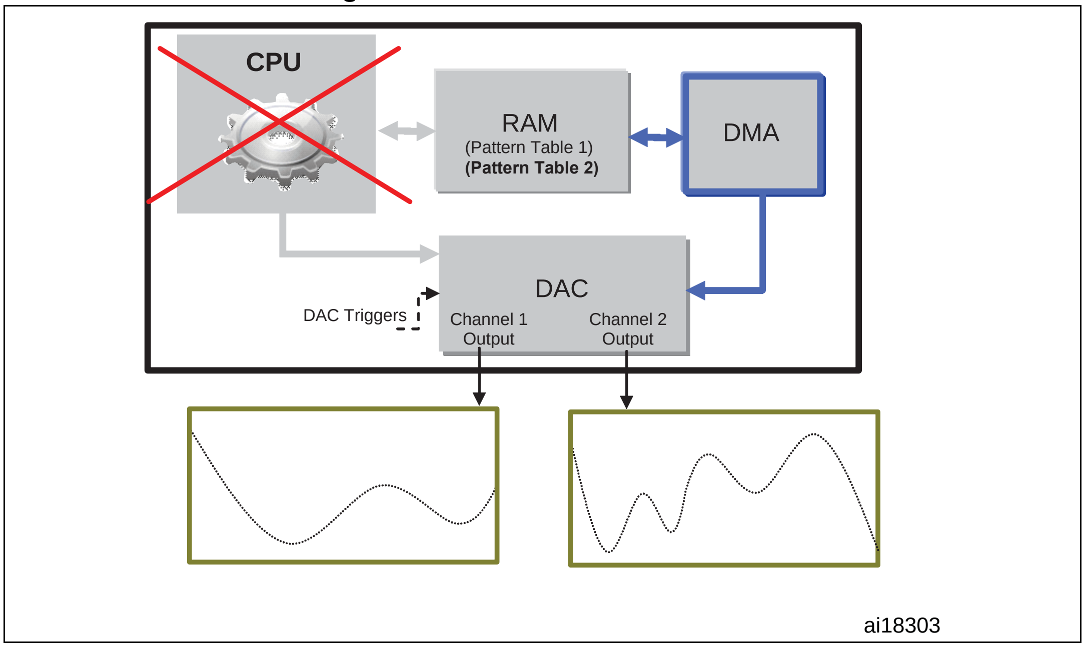
When DMA is not used, the CPU is used to provide DAC with the digital code relevant to the waveform to generate. This code is saved in a RAM, or in an embedded NV memory, and the CPU transfers the data from the memory to the DAC.

Figure 3. DAC interaction without DMA



When using the DMA, the overall performance of the system is increased by freeing up the CPU. This is because data is moved from memory to DAC by DMA, without need for any action by the CPU. This keeps CPU resources available for other operations.

Figure 4. DAC interaction with DMA



1.6 DMA under run error

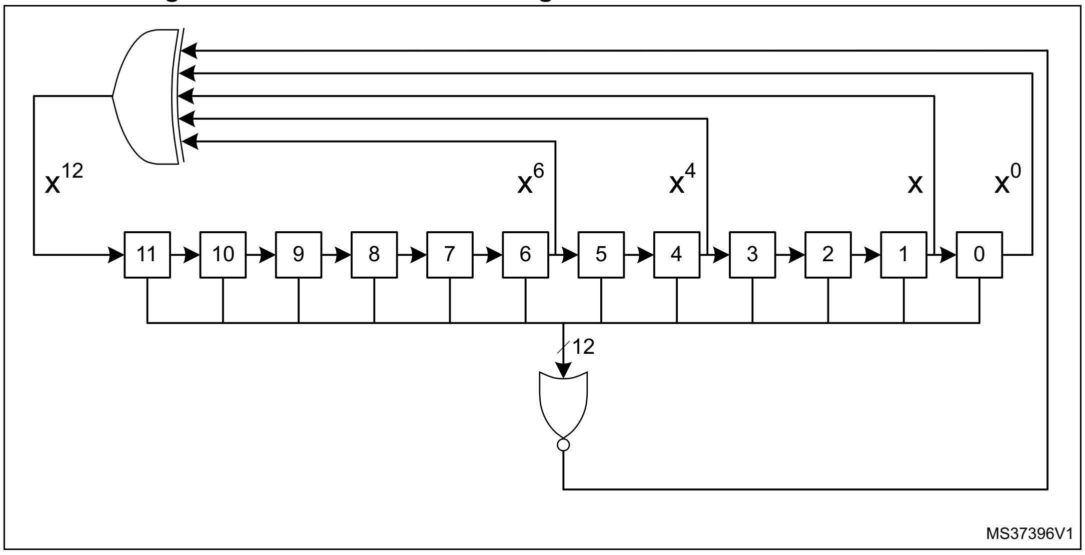
When the DMA is used to provide DAC with the waveform digital code, there are cases where the DMA transfer is slower than the DAC conversion. In these cases, the DAC detects that a part of the pattern waveform has not been received and cannot be converted, and then sets the “DMA under run error” flag.

1.7 White noise generator

1.7.1 Definition

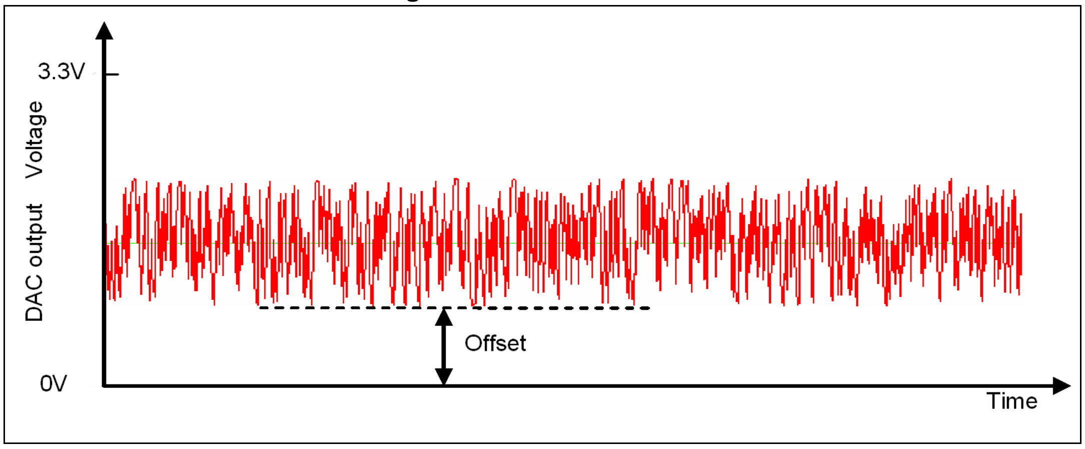
The STM32 DACs feature a pseudo random code generator, sketched in *Figure 5*. Depending on what taps are used on the shift register, a sequence of up to 2^{n-1} numbers can be generated before the sequence repeats.

Figure 5. Pseudo random code generator embedded in the DAC



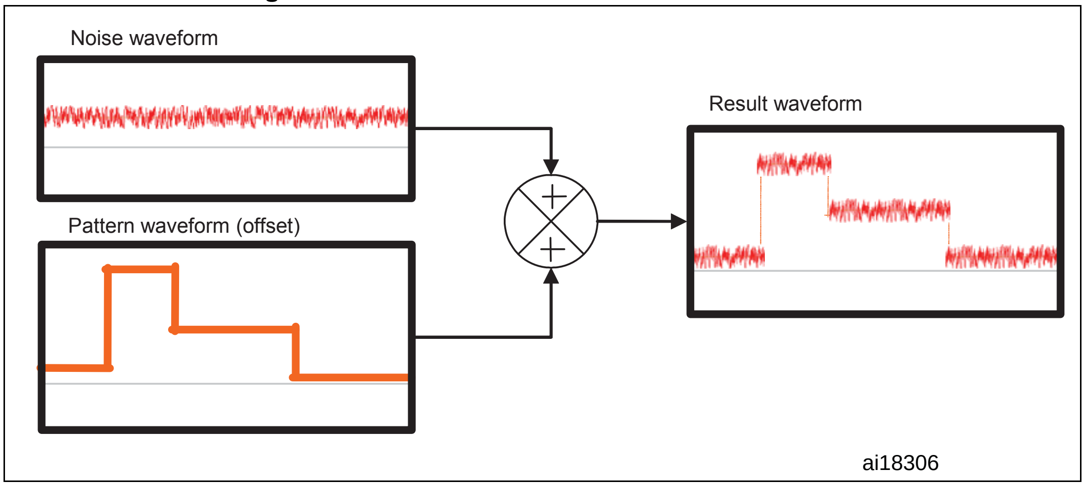
The noise produced by this generator has a flat spectral distribution and can be considered white noise. However, instead of having a Gaussian output characteristics, it is uniformly distributed, see *Figure 6*.

Figure 6. Noise waveform



The offset (or DC bias) of the noise waveform is programmable. By varying this offset with a preconfigured table of offsets (signal pattern), the user can obtain a waveform that corresponds to the sum of the signal pattern and the noise waveform.

Figure 7. Noise waveform with modifiable offset



1.7.2 Typical applications

STM32 products come with 12-bit enhanced ADCs with a sampling rate that can exceed 1 M samples/s. In most applications, this resolution is sufficient, when higher accuracy is required, the concept of oversampling and decimating the input signal can be implemented to save the use of an external ADC solution and to reduce the application power consumption. This noise waveform can be used to enhance the ADC accuracy with the oversampling method.

More details about these methods are explained in the application note AN2668, available on *www.st.com*, in the section titled “Oversampling using white noise”.

The white noise generator can be also used in the production of electronic music, either directly or as an input for a filter to create other types of noise signals. It is used extensively in audio synthesis, typically to recreate percussive instruments such as cymbals, which have high noise content in their frequency domain.

White noise generator can be used for control engineering purposes, e.g. for frequency response testing of amplifiers and electronic filters.

1.8 Triangular wave generator

1.8.1 Definition

The STM32 DAC provides the user with a triangular waveform generator with flexible offset, amplitude and frequency.

The amplitude of the triangular waveform can be fixed using the MAMPx bits in the DAC_CR register.

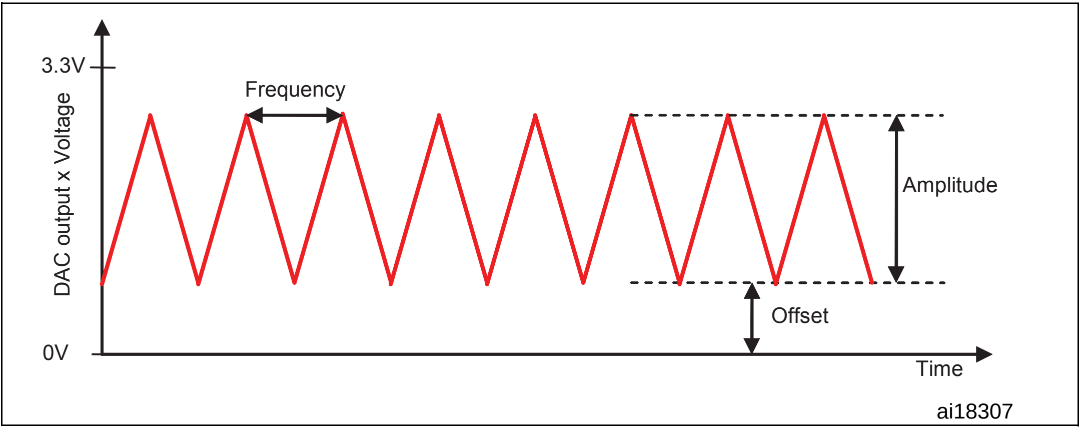
Table 2. Preprogrammable triangular waveform amplitude values

MAMPx[3:0] bits	Digital amplitude	Analog amplitude (Volt) (with V_{REF+} = 3.3 V)
0	1	0.0008
1	3	0.0024
2	7	0.0056
3	15	0.0121
4	31	0.0250
5	63	0.0508
6	127	0.1023
7	255	0.2054
8	511	0.4117
9	1023	0.8242
10	2047	1.6492
≥ 11	4095	3.2992

For more details about the triangular waveform refer to the dedicated sections in the reference manuals.

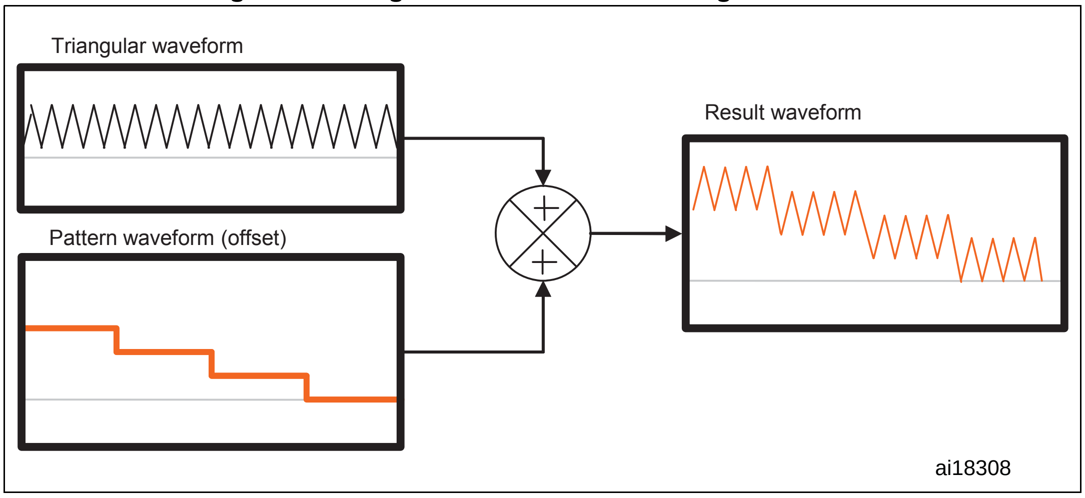
The triangular waveform frequency is related to the frequency of the trigger source.

Figure 8. Triangular waveform



The offset of the triangular waveform is programmable (see *Figure 9*). By varying the offset of the triangular waveform with a preconfigured table of offsets (signal pattern), user can obtain a waveform that corresponds to the sum of the signal pattern and the triangular waveform. As there is no hardware overflow protection, the sum of offsets and amplitude must not be higher than 4095.

Figure 9. Triangular waveform with changeable offset



1.8.2 Typical applications

Triangular wave generators are often used in sound synthesis as their timbre is less harsh than the square wave (the amplitude of its upper harmonics falls off more rapidly).

1.9 Buffered output

To drive external loads without using an external amplifier, DAC channels have embedded output buffers that can be enabled and disabled depending on the user application.

When the DAC output is not buffered, and there is a load in the user application circuit, the voltage output is lower than the desired voltage (*Figure 10*), because of the significant DAC output impedance. Refer to the relevant STM32 datasheet for the specification of DAC output impedance (for example, for STM32F4 products, the resistive load resistance must be higher than 1.5 MΩ to have an output voltage drop below 1% of the output signal voltage).

When enabling the buffer, the output and the desired voltages are similar (*Figure 11*).

Figure 10. Non buffered channel voltage (with and without load)

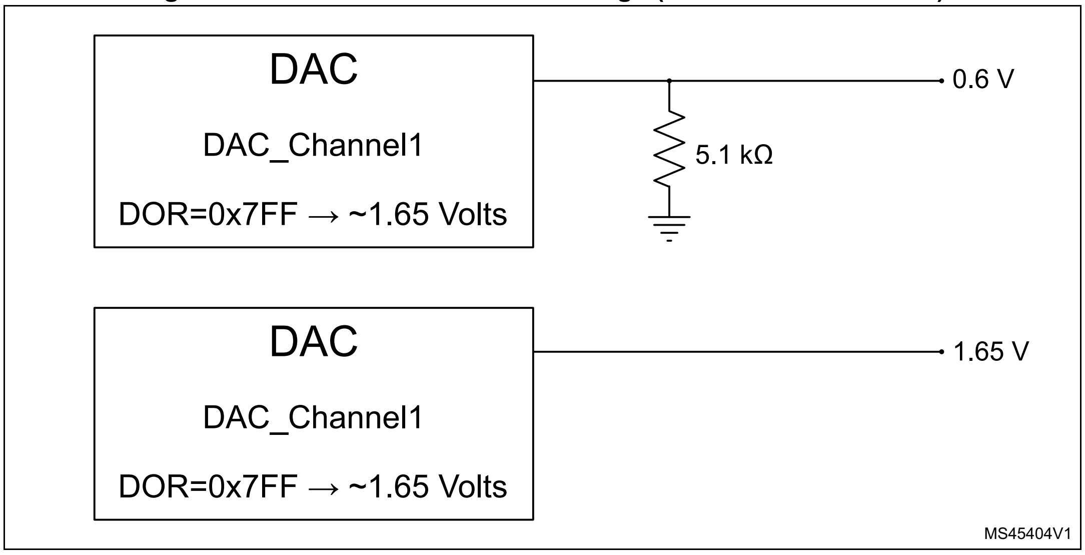
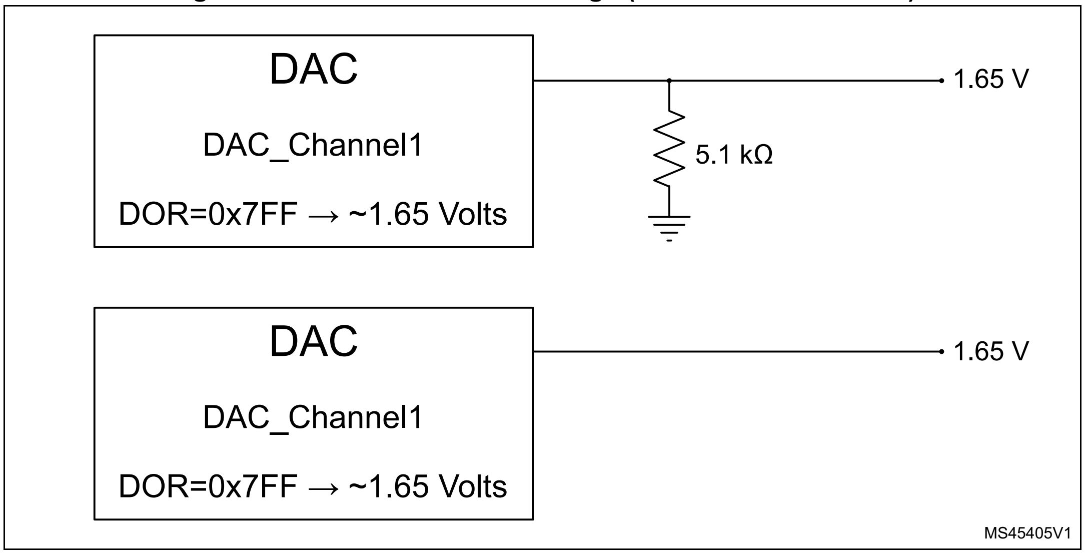


Figure 11. Buffered channel voltage (with and without load)



2 Application examples

2.1 Using the DAC to generate a sine waveform

2.1.1 Description

This example describes step by step how to generate a sine waveform.

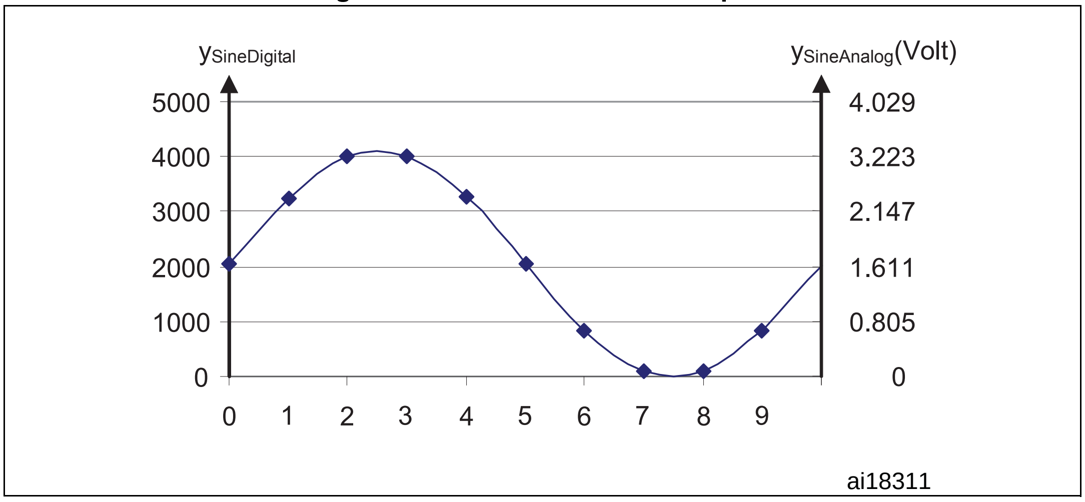
A sine waveform is also called a monotone signal, it is known as a pure (or sine) tone. The sine tones are traditionally used as stimuli to assess the response of the audio system.

2.1.2 Waveform preparation

To prepare the digital pattern of the waveform, we have to go through some mathematics.

Our objective is to have ten digital pattern data (samples) of a sine wave form that varies from 0 to 2π.

Figure 12. Sine wave model samples



The sampling step is 2π / n_s (number of samples).

The value of sin(x) varies between -1 and 1, we have to shift it up to have a positive sine wave with samples varying between 0 and 0xFFF (corresponding to the 0 to 3.3 V voltage range, where V_{REF} is set to 3.3 V).

$$y_{SineDigital}(x) = \left(\sin\left(x \cdot \frac{2\pi}{n_s}\right) + 1\right)\left(\frac{(0xFFF + 1)}{2}\right)$$

Digital inputs are converted to output voltages on a linear conversion between 0 and V_{REF+}.

The analog output voltages on each DAC channel pin are determined by the equation

$$DAC_{Output} = V_{REF}\frac{DOR}{DAC\_MaxDigitalValue + 1}$$

Note: For right-aligned 12-bit resolution: DAC_MaxDigitalValue = 0xFFF
For right-aligned 8-bit resolution: DAC_MaxDigitalValue = 0xFF

So the analog sine waveform $y_{SineAnalog}$ can be determined by the following equation

$$y_{SineAnalog}(x) = 3.3\text{Volt}\frac{y_{SineDigital}(x)}{0xFFF + 1}$$

Table 3. Digital and analog sample values of the sine wave

Sample (x)	Digital sample value $y_{SineDigital}(x)$	Analog sample value (Volt) $y_{SineAnalog}(x)$
0	2048	1.650
1	3251	2.620
2	3995	3.219
3	3996	3.220
4	3253	2.622
5	2051	1.653
6	847	0.682
7	101	0.081
8	98	0.079
9	839	0.676

The table is saved in the memory and transferred by the DMA, the transfer is triggered by the same timer that triggers the DAC.

2.1.3 Setting the sine wave frequency

To set the frequency of the sine wave signal, the user has to set the frequency ($f_{TimerTRGO}$) of the timer trigger output.

The frequency of the produced sine wave is

$$f_{Sinewave} = \frac{f_{TimerTRGO}}{n_s}$$

So, if the TIMx_TRGO output frequency is 1 MHz, the frequency of the sine wave generated by the DAC is 100 kHz.

Note: To get close to the targeted monotone waveform, it is recommended to use the highest possible number of samples n_s (the difference can be easily understood by comparing Figure 13 with Figure 14).

Figure 13. Sine wave generated with n_s = 10

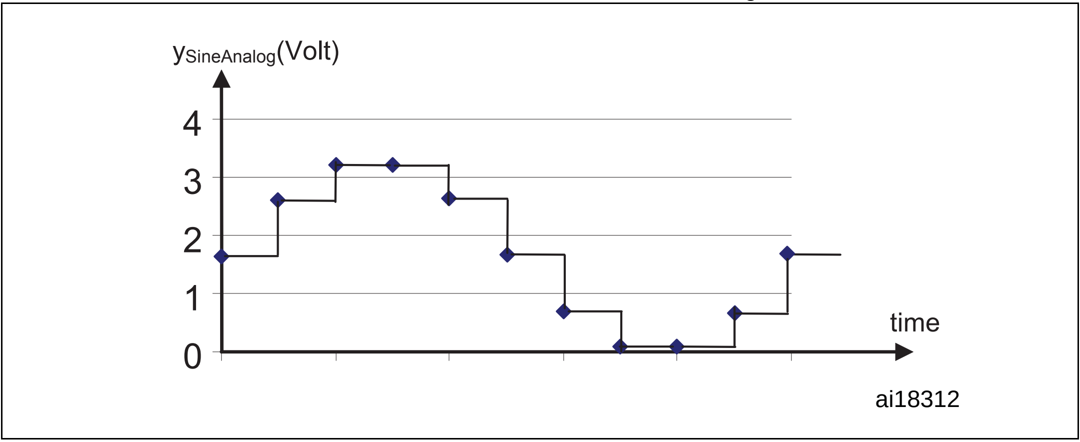
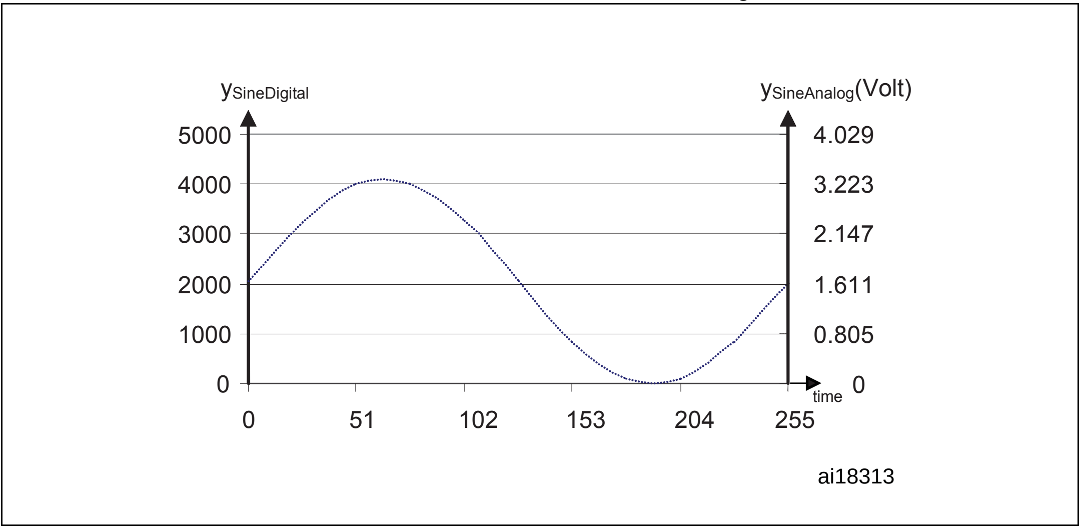


Figure 14. Sine wave generated with n_s = 255

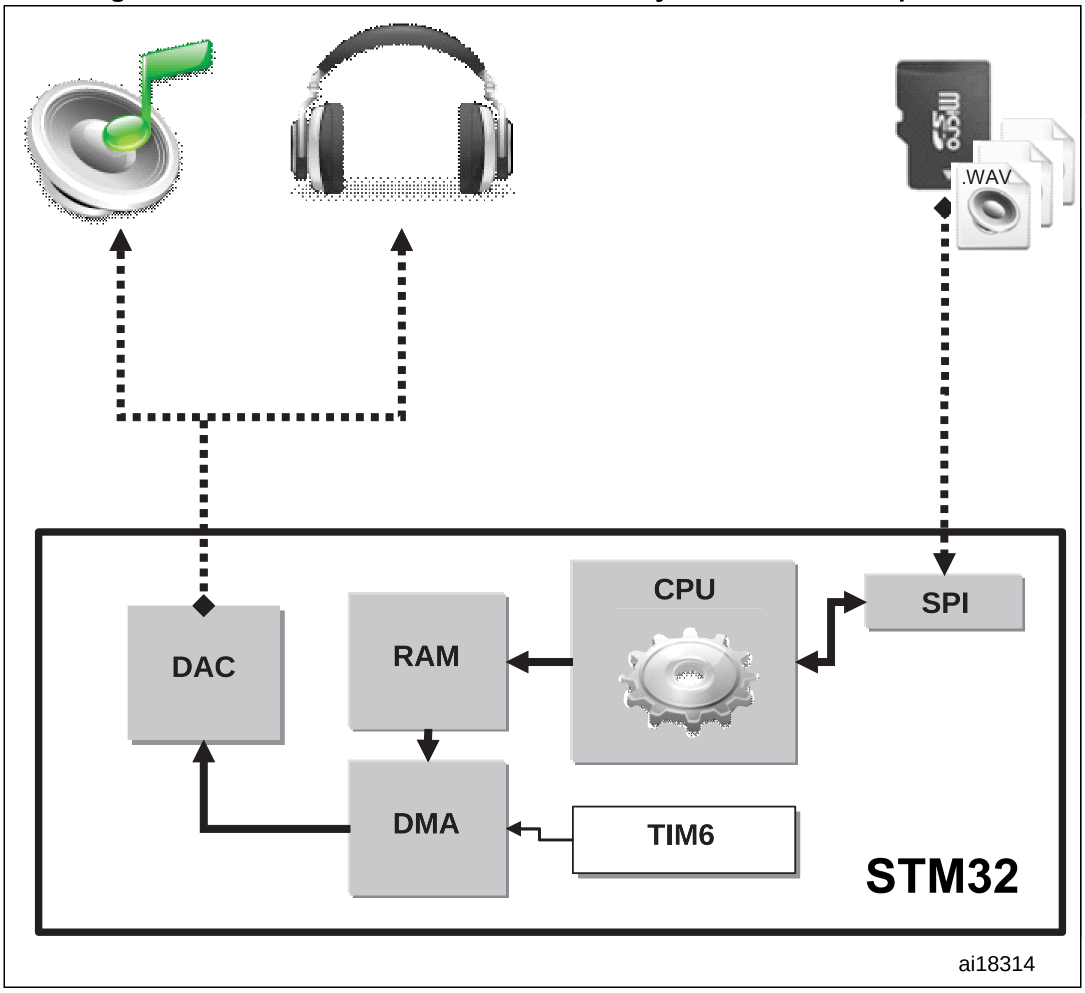


2.2 Using the DAC to implement an audio player

2.2.1 Description

The purpose of this demonstration is to provide an audio player solution based on STM32 products to play .WAV files. The approach is optimized to use a minimum number of external components, and offers to end-users the possibility of using their own .WAV files. The audio files are stored in a microSD memory card, accessible by the STM32 through the SPI bus.

Figure 15. Flow of data from microSD memory card to external speakers



The audio player demonstration described in this section is a part of the STM32100B-EVAL demonstration firmware, which can be downloaded, together with the associated user manual (UM0891), from the STMicroelectronics website *www.st.com*.

2.2.2 Audio wave file specifications

This application assumes that the .WAV file to be played has the following format:

- audio format: PCM (an uncompressed wave data format in which each value represents the amplitude of the signal at the time of sampling)
- sample rate: may be 8000, 11025, 22050 or 44100 Hz
- bits per sample: 8-bit (audio sample data values are in the range [0-255])
- number of channels: 1 (mono)

2.2.3 .WAV file format

The .WAV file format is a subset of the Resource Interchange File Format (RIFF) specification used for the storage of multimedia files. A RIFF file starts with a file header followed by a sequence of data chunks. A .WAV file is often just a RIFF file with a single "WAVE" chunk consisting of two sub-chunks:

1. a **fmt** chunk, specifying the data format
2. a **data** chunk, containing the actual sample data.

The WAVE file format starts with the RIFF header: it indicates the file length.

Next, the fmt chunk describes the sample format, it contains information about the format of the wave audio (PCM / ...), the number of channels (mono/stereo), the sample rate (number of samples per seconds, e.g. 22050), and the sample data size (e.g. 8 bit / 16 bit). Finally, the data chunk contains the sample data.

2.3 Audio wave player implementation

The Audio wave player application is based on the SPI, DMA, TIM6, and DAC peripherals.

At start up, the application first uses the SPI to interface with the microSD card and parses its content, using the DOSFS file system, looking for available .WAV files in the USER folder. Once a valid .WAV file is found, it is read back though the SPI, and the data are transferred using the CPU to a buffer array located in the RAM. The DMA is used to transfer data from RAM to DAC peripheral. TIM6 is used to trigger the DAC that will convert the audio digital data to an analog waveform.

Before the audio data can be played, the header of the .WAV file is parsed so that the sampling rate of the data and their length can be determined.

The task of reproducing audio is achieved by using sampled data (data contained in the .WAV file) to update the value of the DAC output, these data are coded in 8 bits (with values from 0 to 255).

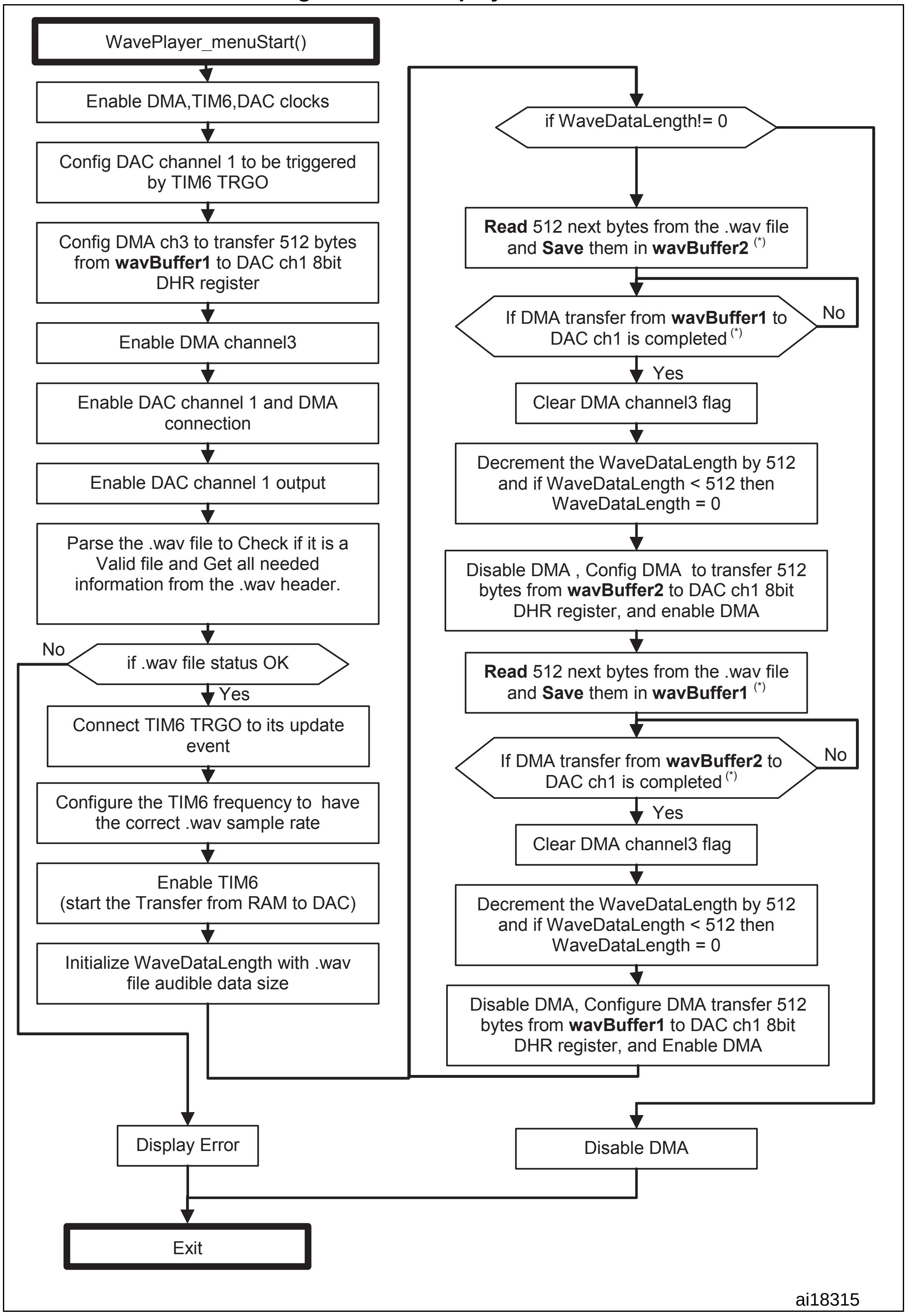
The DAC Channel1 is triggered by TIM6 at regular intervals, specified by the sample rate of the .WAV file header.

The .WAV files are read from the microSD™ card already formatted with a DOSFS file system.

In the demonstration source code project the audio player routines are included in the C-language files waveplayer.c and waveplayer.h.

The audio player task starts by invoking the `WavePlayerMenu_Start()` function described in *Figure 16*.

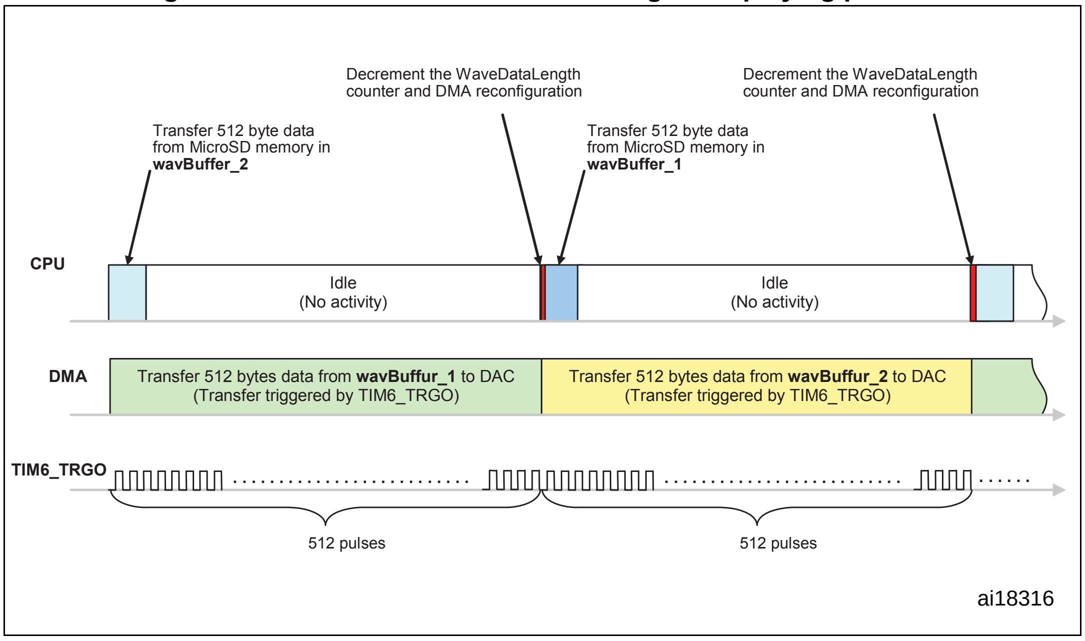
Figure 16. Audio player flowchart


WavePlayer_menuStart()
Enable DMA,TIM6,DAC clocks
Config DAC channel 1 to be triggered by TIM6 TRGO
Config DMA ch3 to transfer 512 bytes from wavBuffer1 to DAC ch1 8bit DHR register
Enable DMA channel3
Enable DAC channel 1 and DMA connection
Enable DAC channel 1 output
Parse the .wav file to Check if it is a Valid file and Get all needed information from the .wav header.
if .wav file status OK
No
Yes
Connect TIM6 TRGO to its update event
Configure the TIM6 frequency to have the correct .wav sample rate
Enable TIM6 (start the Transfer from RAM to DAC)
Initialize WaveDataLength with .wav file audible data size
if WaveDataLength!= 0
Read 512 next bytes from the .wav file and Save them in wavBuffer2 (*)
If DMA transfer from wavBuffer1 to DAC ch1 is completed (*)
No
Yes
Clear DMA channel3 flag
Decrement the WaveDataLength by 512 and if WaveDataLength < 512 then WaveDataLength = 0
Disable DMA , Config DMA to transfer 512 bytes from wavBuffer2 to DAC ch1 8bit DHR register, and enable DMA
Read 512 next bytes from the .wav file and Save them in wavBuffer1 (*)
If DMA transfer from wavBuffer2 to DAC ch1 is completed (*)
No
Yes
Clear DMA channel3 flag
Decrement the WaveDataLength by 512 and if WaveDataLength < 512 then WaveDataLength = 0
Disable DMA, Configure DMA transfer 512 bytes from wavBuffer1 to DAC ch1 8bit DHR register, and Enable DMA
Display Error
Disable DMA
Exit
ai18315

When DMA transfers data from one SRAM buffer, the CPU transfers data from the microSD™ Flash memory to the other SRAM buffer.

In this application, co-processing is mandatory to permit a simultaneous read (from the external memory) and write (in the DAC register) of the waveform digital code.

Figure 17. CPU and DMA activities during wave playing process


Decrement the WaveDataLength counter and DMA reconfiguration
Decrement the WaveDataLength counter and DMA reconfiguration
Transfer 512 byte data from MicroSD memory in **wavBuffer_2**
Transfer 512 byte data from MicroSD memory in **wavBuffer_1**
CPU
Idle (No activity)
Idle (No activity)
DMA
Transfer 512 bytes data from **wavBuffur_1** to DAC (Transfer triggered by TIM6_TRGO)
Transfer 512 bytes data from **wavBuffur_2** to DAC (Transfer triggered by TIM6_TRGO)
TIM6_TRGO
512 pulses
512 pulses
ai18316

3 Conclusion

This application note and in particular the examples given in *Section 2: Application examples* help the user to become familiar with the DACs main features.

The first example (*Section 2.1: Using the DAC to generate a sine waveform*) shows how to generate an analog waveform (a sine waveform generation source code is provided as reference). The second example (*Section 2.2: Using the DAC to implement an audio player*) offers a straightforward and flexible solution to use the STM32 to play .WAV files stored in an SPI microSD™ Flash memory.

These examples can be used as starting points to develop your own solution based on STM32 products.

4 Revision history

Table 4. Document revision history

Date	Revision	Changes
28-May-2010	1	Initial release.
16-Apr-2015	2	Updated *Introduction*, *Section 1.3: Dual channel mode* and *Section 3: Conclusion*. Updated formulas in *Section 2.1.2: Waveform preparation*. Updated *Figure 2: STM32F10x DAC trigger channels* and *Figure 5: Pseudo random code generator embedded in the DAC*. Added *Table 1: Applicable products* and *Table 2: DAC configuration for STM32 microcontrollers*. Added *Section 1.1: Overview* and Note in *Section 1.3: Dual channel mode*.
11-May-2017	3	Introduced STM32H7 Series, hence updated *Table 1: Applicable products* and *Table 2: DAC configuration for STM32 microcontrollers*. Updated *Introduction*, *Section 1.2: Data format*, *Section 1.5: DMA capabilities*, *Section 1.6: DMA under run error*, *Section 1.7.2: Typical applications*, *Section 1.8.1: Definition*, *Section 1.9: Buffered output*, *Section 2.1.1: Description*, *Section 2.1.2: Waveform preparation*, *Section 2.1.3: Setting the sine wave frequency*, *Section 2.2.1: Description*, *Section 2.3: Audio wave player implementation* and *Section 3: Conclusion*. Updated *Table 2: Preprogrammable triangular waveform amplitude values*. Updated *Figure 10: Non buffered channel voltage (with and without load)*, *Figure 11: Buffered channel voltage (with and without load)* and *Figure 15: Flow of data from microSD memory card to external speakers*. Updated second equation in *Section 2.1.2: Waveform preparation*.
03-Jul-2020	4	Document extended to all STM32 products, hence updated document title, *Introduction* and *Table 1: Applicable products* . Removed former *Table 2: DAC configuration for STM32 microcontrollers*. Minor text edits across the whole document.