AN0101
Application Note

AT32 MCU DAC Application Note

Introduction

The digital-to-analog converter (DAC) uses a 12-bit/8-bit digital input to generate an analog output between 0 and reference voltage V_{REF+}. The digital part of the DAC can be configured in 8-bit or 12-bit mode and can be used in conjunction with the DMA. It supports left or right alignment in a single /dual DAC modes
The DAC of AT32 MCU product series listed below has two output channels, DAC1 and DAC2, with its own converter each. Each DAC1/DAC2 can be converted independently or simultaneously in dual DAC mode. The input reference voltage V_{REF+} makes conversion more accuracy.

Note: The corresponding code in this application note is developed on the basis of V2.x.x BSP provided by Artery. For other versions of BSP, please pay attention to the differences in usage.

Applicable products:

Part number	AT32F403
	AT32F403A
	AT32F407
	AT32F435
	AT32F437

Contents

List of tables

List of figures

1 DAC introduction

The DAC uses 12-bit/8-bit digital input to generate an analog output between 0 and reference voltage V_{REF+}.

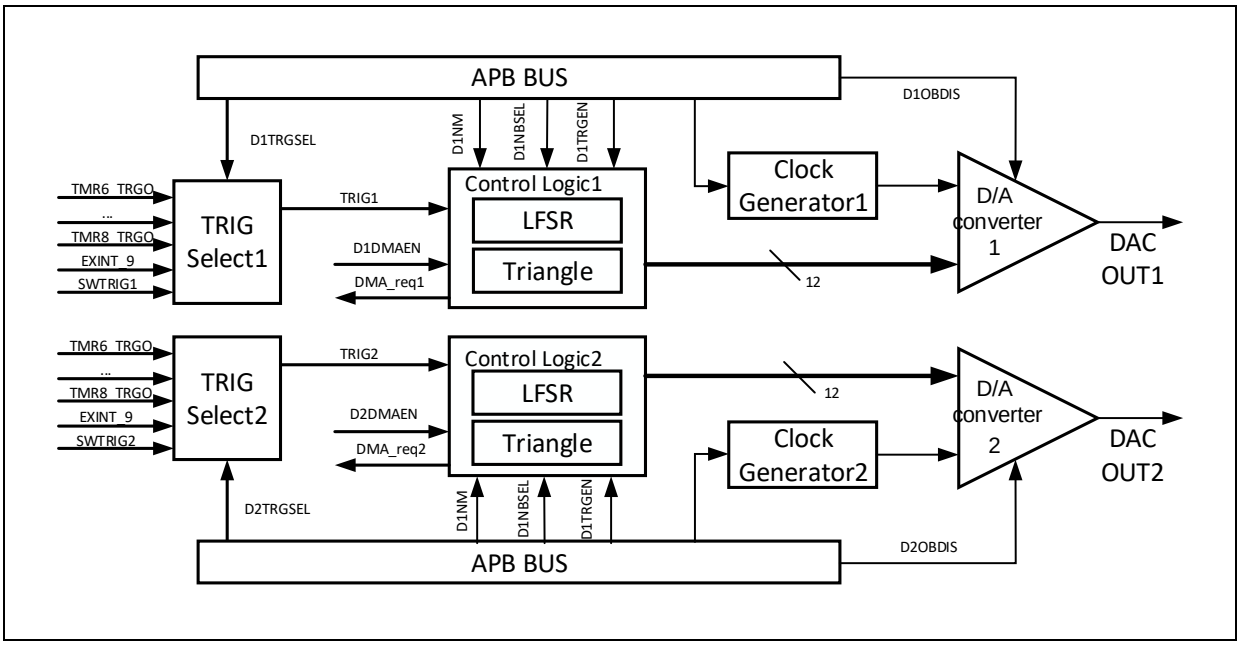
DAC main features:

- 2xDACs, corresponding to one output channel each
- A single/dual DAC 8-bit or 12-bit digital input
- 12-bit data left or right alignment
- Noise-wave/triangular-wave generation
- Dual DACs or single DAC1/DAC2 independent conversions
- DMA mode for DAC1/DAC2
- Software or external triggers for conversion
- Input reference voltage V_{REF+}
- Use in conjunction with DMA

2 DAC function overview

The block diagram of a single DAC channel is shown below.

Figure 1.DAC1/DAC2 block diagram



2.1 DAC pins

The digital inputs are linearly converted to analog voltage outputs by the DAC, and it is between 0 and V_{REF+}.

The analog DAC module is supplied by VDDA. The positive analog reference voltage input falls between 2.0 V and VDDA, and PA4/PA5 should be configured to analog input.

Generally, to guarantee high accuracy of DAC at low voltage, the digital circuit is powered by VDD and the analog circuit is powered by VDDA independently. For the following types in 64PIN package, the external reference voltage V_{REF+} is connected to VDDA pin.

The relation between output voltage and V_{REF+} is shown below:

$$\text{DAC output} = V_{REF+} \times \frac{\text{DAC\_DxODT}[11:0]}{4095}$$

The DAC pins are listed below.

Table 1. DAC pins

Pin	Type	Description
V_{REF+}	Input, positive analog reference voltage	Input reference voltage of DAC, $2.0V \le V_{REF+} \le V_{DDA}$
V_{DDA}	Input, analog supply	Analog supply
V_{SSA}	Input, analog supply ground	Analog supply ground wire
DAC_OUTx	Analog output signal	Analog output of DAC channel x

Note: Once the DAC channel x is enabled, the corresponding GPIO (PA4 or PA5) is automatically connected to DAC analog output (DACx_OUT). To avoid parasitic interruption and excessive consumption, the PA4/PA5 should be configured to analog mode.

2.2 Data alignment

It is not allowed to load data into the DAC_DxODT register directly. Any data output to DAC channel x should be loaded into the corresponding data holding registers (DAC_DxDTH8R, DAC_DxDTH12L, DAC_DxDTH12R, DAC_DDTH8R, DAC_DDTH12L or DAC_DDTH12R register).

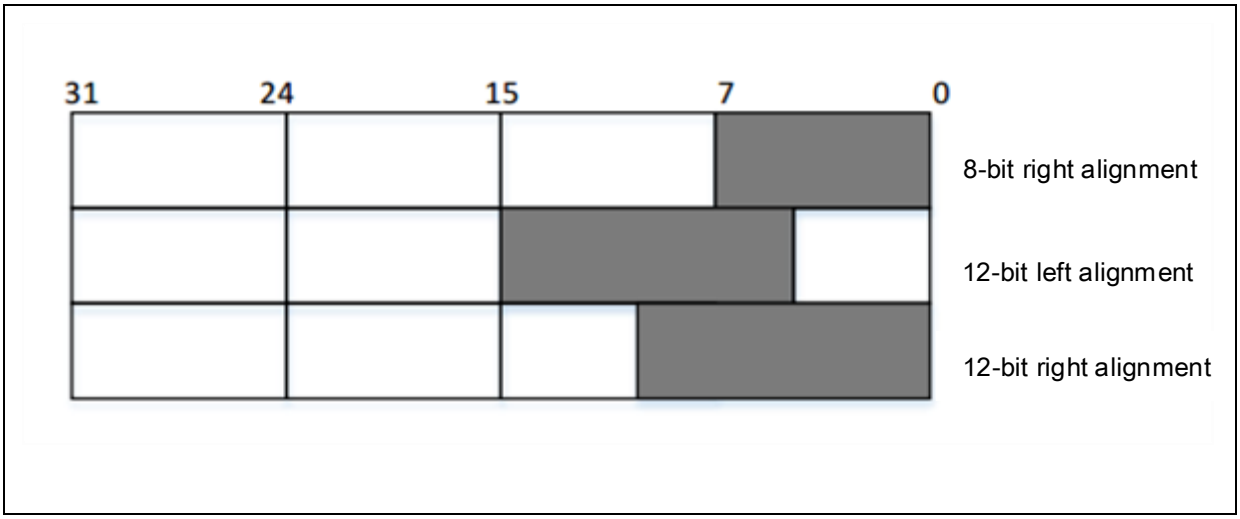
The DAC supports a single DAC and dual DAC mode. The data format is dependent on the selected configuration mode.

Single DAC data format:

- 8-bit right alignment: load data into the DAC_DxDTH8R[7:0]
- 12-bit left alignment: load data into the DAC_DxDTH12L[15:4]
- 12-bit right alignment: load data into the DAC_DxDTH12R[11:0]

Operate the DAC_DxDTHx register and after the corresponding bit shift, the loaded data is transferred into the DHRx register (DHRx is an internal data holding register; the loaded 8-bit data corresponds to the DHRx[11:4] and the loaded 12-bit data corresponds to the DHRx[11:0]). Then, the content in the DHRx register is transmitted to the DAC_DxODT register automatically or through software/external trigger.

Figure 2. Data alignment in single DAC mode

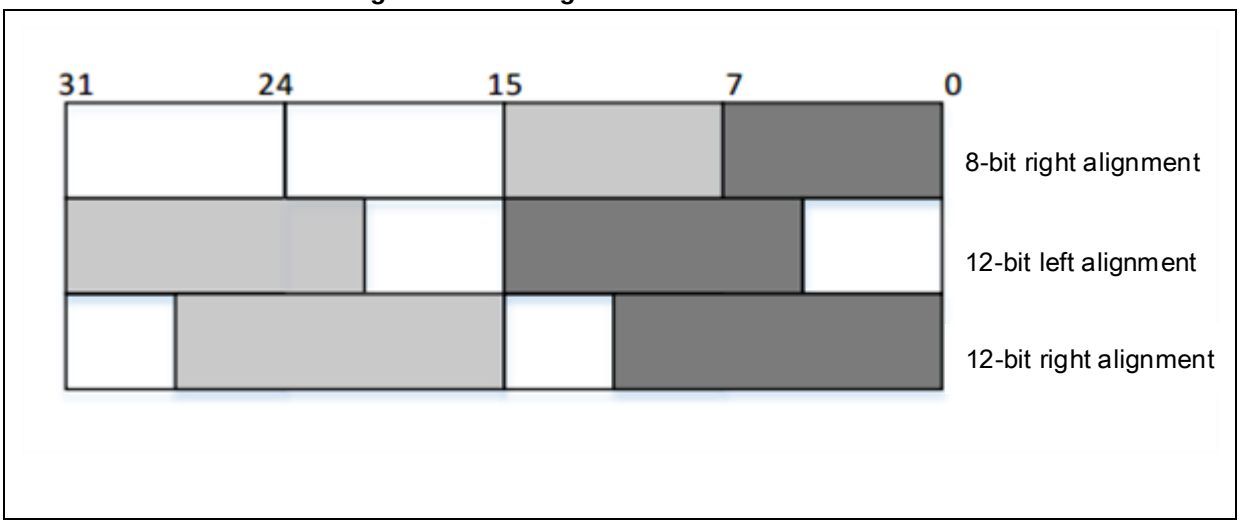


Dual DAC data format:

- 8-bit right alignment: load data into the DAC_DDTH8R[7:0] and DAC_DDTH8R[15:8]
- 12-bit left alignment: load data into the DAC_DDTH12L[15:4] and DAC_DDTH12L[31:20]
- 12-bit right alignment: load data into the DAC_DDTH12R[11:0] and DAC_DDTH12R[27:16]

Operate the DAC_DDTHx register and after the corresponding bit shift, the loaded data transferred into the DHRx register (DHRx is an internal data holding register; the loaded 8-bit data corresponds to the DHRx[11:4] and the loaded 12-bit data corresponds to the DHRx[11:0]). Then, the content in the DHR1 and DHR2 registers is transmitted to the DAC_DxODT register automatically or through software/external trigger.

Figure 3. Data alignment in dual DAC mode



2.3 Trigger events

If the DxTRGEN bit in the DAC_CTRL register is set, the DAC conversion can then be triggered by an external event (timer counter, external interrupt line) or by software. The DxTRGSEL[2:0] bit is used to select trigger sources.

Note: It is not allowed to alter the DxTRGSEL[2:0] bit when DxEN=1.

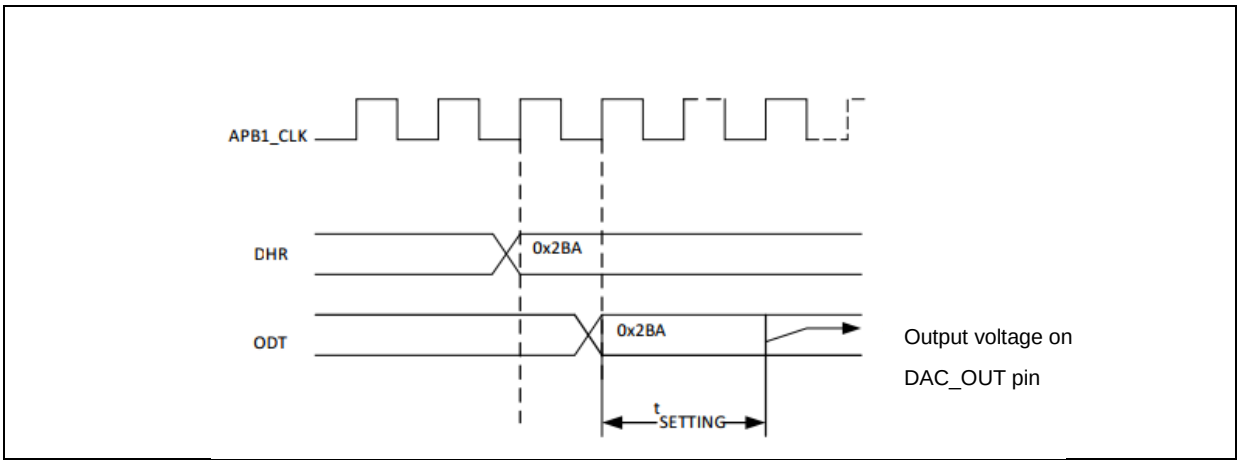
Figure 4. Trigger source selection

Source	DxTRGSEL [2:0]	Description
TMR6_TRGOUT	000	On-chip signals
TMR8_TRGOUT	001	
TMR7_TRGOUT	010	
TMR5_TRGOUT	011	
TMR2_TRGOUT	100	
TMR4_TRGOUT	101	
EXINT_9	110	External signals
DxSWTRG	111	Software trigger

- When the DxTRGEN bit is set, the data stored into the DHRx register is transferred into the DAC_DxODT register each time a DAC detects an active trigger event (timer TRGO output or rising edge on external interrupt line 9). After three APB1 clock cycles, the DAC_DxODT is updated to a new value.
- When the DxTRGEN bit is set and software trigger is selected, the conversion starts once the DxSWTRG is set by software, and the DAC_DxODT is updated to a new value after one APB1 clock cycle. After the data is transmitted to the DAC_DxODT register, the DxSWTRG is cleared by hardware automatically.
- When the DxTRGEN bit is cleared, the conversion starts immediately each time the data is written to the data holding register, without the need of a trigger event. After one APB1 clock cycle, the data is immediately transmitted to the DAC_DxODT register.

The DAC output becomes active after a period of time ($t_{SETTLING}$) once the data is loaded into the DAC_DxODT register from DHRx[11:0]. The $t_{SETTLING}$ is dependent on the power supply voltage and analog output load.

Figure 5. Time block diagram of conversion when DxTRGEN=0



2.4 DMA

The DAC1/DAC2 both have a DMA capability that can be enabled by setting the DxDMAEN bit in the DAC_CTRL register. One DMA request is generated when a trigger signal is active while the DxTRGEN bit is set. Then, the data in the DHRx register is transmitted to the DAC_DxODT register.

In dual DAC mode, two DMA requests are generated when the DxDMAEN bits of the two channels (DAC1/DAC2) are set. If only one DMA is required for transmission, set one DxDMAEN bit to 1, and then the application can handle two channels (DAC1/DAC2) by using only one DMA request and one DMA channel.

Note: The DAC DMA request is not added up, meaning the new DAM request will be ignored and no error is reported. If the second external trigger occurs before the response to the first external trigger, the second DMA request cannot be processed.

2.5 Noise/triangular wave generation

The DAC can output a variable-amplitude pseudo noise and a triangular wave, which is done by the LENinear Feedback Shift Register and triangle wave generator respectively. The DAC variable-amplitude pseudo noise generation is selected by setting DxNM[1:0]=01 in the DAC_CTRL register, and the DAC triangular-wave generation is selected by setting the DxNM[1:0]=1x.

LFSR logic

The preload value in the LFSR is 0xAAA. This register is updated after each trigger event based on a specific calculation algorithm. The DxNBSEL[3:0] bit in the DAC_CTRL register is set to mask partially or totally the LFSR data. The resulting value is then added up to the DHRx value without overflow, and this value is loaded into the DAC_DxODT register.

It is possible to disable LFSR function and reset LFSR wave generation algorithm by setting DxNM[1:0]=00.

Triangular wave logic

The triangular wave amplitude is configured through the DxNBSEL[3:0] bit in the DAC_CTRL register. An internal triangular-wave counter is incremented at each trigger event. Once the maximum amplitude programmed in the DxNBSEL[3:0] is reached, the value of this counter is decremented down to 0, then incremented again, and so on. Meanwhile, the value of this counter is then added up to the DHRx register without overflow, and the resulting value is loaded into the DAC_DxODT register.

It is possible to disable/reset the triangular-wave generation by setting DxNM[1:0]=00.

Note 1: To generate noise/triangular wave, the DAC trigger must be enabled by setting the DxTRGEN bit in the DAC_CTRL register.

Note 2: The DxNBSEL[3:0] bit must be set before enabling DAC; otherwise, it cannot be modified.

2.6 DAC design tips

- DACx trigger source
  - Set the trigger enable bit (DxTRGEN) of DACx channel
  - Set the trigger select bit (DxTRGSEL) of DACx channel to select the trigger source
- DACx noise/triangular wave generation
  - To generate noise, set DxNM[1:0]=01 in the DAC_CTRL register and set the DxNBSEL[3:0] to select the mask bit
  - To generate triangular wave, set DxNM[1:0]=1x in the DAC_CTRL register and set DxNBSEL[3:0] to select the amplitude
  - Set DxNM[1:0]=00 to disable this function
- Load the data to be output to DACx into the DAC_DxDTHx/DAC_DDTHx register
  - DACx 8-bit right-aligned data holding register (DAC_DxDTH8R)
  - DACx 12-bit left-aligned data holding register (DAC_DxDTH12L)
  - DACx 12-bit right-aligned data holding register (DAC_DxDTH12R)
  - Dual DAC 8-bit right-aligned data holding register (DAC_DDTH8R)
  - Dual DAC 12-bit left-aligned data holding register (DAC_DDTH12L)
  - Dual DAC 12-bit right-aligned data holding register (DAC_DDTH12R)
- DMA configuration
  - Configure DMA as required
- DACx output buffer
  - DAC integrates output gains to reduce output impedance and directly drive external loads without the need of an external operational amplifier. Each DAC output gain can be enabled and disabled through the DxOBDIS bit in the DAC_CTRL register.
- DACx enable
  - DACx analog part is enabled through the DxEN bit in the DAC_CTRL register, and the digital part is not controlled by this bit.

3 DAC application

Note: All projects are built around keil 5. If users want to use them in other compiling environments, please refer to AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates (such as IAR6/7, keil 4/5) for a simple change.

3.1 Example 1: Dual DAC sine wave output by using DMA

3.1.1 Function overview

Select TMR2 TRGO event to trigger DAC conversion, and use DMA to transfer the data in array to the DAC_HDR12RD register to realize sine wave output through PA4/PA5.

3.1.2 Resources

1) Hardware

   AT-START BOARD of the AT32F403A series

2) Software

   \project\at_start_f403a\examples\dac\double_mode_dma_sinewave\mdk_v5

3.1.3 Software design

1) Configuration process
   - Configure clocks and the GPIO corresponding to DAC
   - Configure TMR/DAC/DMA

2) Code
   - Clock configuration code

```
/* Enable DMA1/DAC/TMR2/GPIOA clock */
crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_DAC_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_TMR2_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
```

   - GPIO configuration code

```
/*Once the DACx channel is enabled, the corresponding GPIO pin (PA4/PA5) will be connected to
DACx_OUT automatically. To avoid parasitic interruption and excessive consumption, the PA4/PA5
should be configured to analog mode in advance. */
gpio_init_struct.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5; /*Select pin4 and pin5*/
gpio_init_struct.gpio_mode = GPIO_MODE_ANALOG; /*Configure GPIO to analog mode*/
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /*Configure GPIO to push-pull mode*/
gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /*Configure GPIO to no pull-up and no pull-down*/
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /*Configure strong
GPIO drive capability*/
gpio_init(GPIOA, &gpio_init_struct); /*Configure PA4/PA5 according to the above settings*/
```

- TMR configuration code

```
/* Get the system clock for TMR trigger frequency configuration */
crm_clocks_freq_get(&crm_clocks_freq_struct);
/* Initialize TMR2, upcounting mode, trigger frequency of 10kHz==
(systemclock/(systemclock/1000000))/100    */
tmr_base_init(TMR2, 100-1, (crm_clocks_freq_struct.sclk_freq/1000000 - 1));
tmr_cnt_dir_set(TMR2, TMR_COUNT_UP);
/* Master TMR2 output selection update OVERFLOW */
tmr_primary_mode_select(TMR2, TMR_PRIMARY_SEL_OVERFLOW);
/*Enable TMR2*/
tmr_counter_enable(TMR2, TRUE);
```

- DAC configuration code

```
/* Select TMR2 TRGOUT as the trigger source of DAC1/DAC2 */
dac_trigger_select(DAC1_SELECT, DAC_TMR2_TRGOUT_EVENT);
dac_trigger_select(DAC2_SELECT, DAC_TMR2_TRGOUT_EVENT);
/* DAC1/DAC2 trigger enable*/
dac_trigger_enable(DAC1_SELECT, TRUE);
dac_trigger_enable(DAC2_SELECT, TRUE);
/* DAC1/DAC2 noise/triangular wave generation disable */
dac_wave_generate(DAC1_SELECT, DAC_WAVE_GENERATE_NONE);
dac_wave_generate(DAC2_SELECT, DAC_WAVE_GENERATE_NONE);
/* DAC1/DAC2 output buffer disable */
dac_output_buffer_enable(DAC1_SELECT, FALSE);
dac_output_buffer_enable(DAC2_SELECT, FALSE);
/* DAC1/DAC2 DMA enable */
dac_dma_enable(DAC1_SELECT, TRUE);
dac_dma_enable(DAC2_SELECT, TRUE);
/* DAC1/DAC2 enable*/
dac_enable(DAC1_SELECT, TRUE);
dac_enable(DAC2_SELECT, TRUE);
```

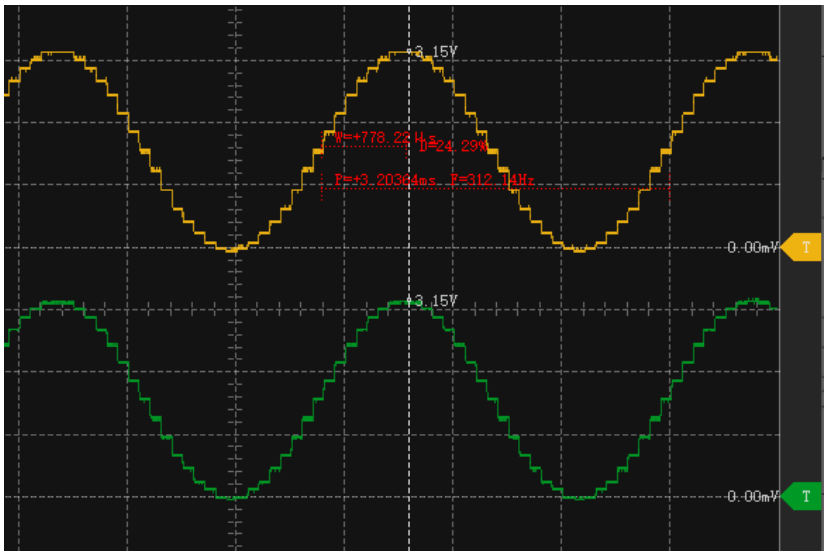
- DMA configuration code

```
/* Sine wave array data */
const uint16_t sine12bit[32] = {2047, 2447, 2831, 3185, 3498, 3750, 3939, 4056,
                                4095, 4056, 3939, 3750, 3495, 3185, 2831, 2447,
                                2047, 1647, 1263, 909,  599,  344,  155,  38,
                                0,    38,   155,  344,  599,  909,  1263, 1647};
uint32_t dualsine12bit[32];
```

```
uint32_t idx = 0;
/*Array data is aligned with the DAC_HDR12RD register*/
for(idx = 0; idx < 32; idx++)
{
  dualsine12bit[idx] = (sine12bit[idx] << 16) + (sine12bit[idx]);
}
/* DMA1 CHANNEL1 configuration */
dma_reset(DMA1_CHANNEL1); /*Reset DMA1 channel1 and keep channel1 at default*/
dma_init_struct.buffer_size = 32; /*Set DMA buffer length: to be the same as the array */
dma_init_struct.direction = DMA_DIR_MEMORY_TO_PERIPHERAL; /*Data transmission direction:
memory to peripheral */
dma_init_struct.memory_base_addr = (uint32_t)dualsine12bit; /*Memory address */
dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_WORD; /*Data width WORD*/
dma_init_struct.memory_inc_enable = TRUE; /*Memory address increment: the memory address is
 added up each time the data is transmitted */
dma_init_struct.peripheral_base_addr = (uint32_t)0x40007420; /*Peripheral DAC_HDR12RD register*/
dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_WORD; /*Data width*/
dma_init_struct.peripheral_inc_enable = FALSE; /*Peripheral address increment disabled,
DAC_HDR12RD register*/
dma_init_struct.priority = DMA_PRIORITY_MEDIUM; /*DMA priority: medium */
dma_init_struct.loop_mode_enable = TRUE; /*Loop mode enable*/
dma_init(DMA1_CHANNEL1, &dma_init_struct); /*Set DMA1 channel1 as abovementioned*/
/*DMA1 CHANNEL1 flexible mapping to DAC2 */
dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_DAC2);
/*Enable DMA1 CHANNEL1 */
dma_channel_enable(DMA1_CHANNEL1, TRUE);
```

3.1.4 Test result

Figure 6. Test result of double_mode_dma_sinewave



As shown in Figure 6, the sine wave amplitude is about 3.15 V and the period is about 3.2 ms (32*1/10kHz), basically meeting the program design expectations.

3.2 Example 2: Dual DAC square wave output by using DMA

3.2.1 Function overview

Trigger DAC conversion by software every 100us, and use DMA to transfer the data (0xfff and 0x0) in array to the DAC_HDR12RD) to realize square wave output through PA4/PA5.

3.2.2 Resources

1) Hardware

   AT-START BOARD of the AT32F403A series

2) Software

   \project\at_start_f403a\examples\dac\double_mode_dma_squarewave\mdk_v5

3.2.3 Software design

1) Configuration process
   - Configure clocks and the GPIO corresponding to DAC
   - Configure DAC/DMA

2) Code
   - Clock configuration code

```
/* Enable DMA1/DAC/ GPIOA clock */
crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_DAC_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
```

- GPIO configuration code

```
/* Once the DACx channel is enabled, the corresponding GPIO pin (PA4/PA5) will be connected to
DACx_OUT automatically. To avoid parasitic interruption and excessive consumption, the PA4/PA5
should be configured to analog mode in advance. */
gpio_init_struct.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5; /*Select pin4 and pin5*/
gpio_init_struct.gpio_mode = GPIO_MODE_ANALOG; /* Configure GPIO to analog mode */
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* Configure GPIO to push-pull mode */
gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* Configure GPIO to no pull-up and no pull-down */
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* Configure strong
GPIO drive capability*/
gpio_init(GPIOA, &gpio_init_struct); /* Configure PA4/PA5 according to the above settings */
```

- DAC configuration code

```
/* Select software to trigger DAC1/DAC2 */
dac_trigger_select(DAC1_SELECT, DAC_SOFTWARE_TRIGGER);
dac_trigger_select(DAC2_SELECT, DAC_SOFTWARE_TRIGGER);
/* DAC1/DAC2 trigger enable*/
dac_trigger_enable(DAC1_SELECT, TRUE);
dac_trigger_enable(DAC2_SELECT, TRUE);
/* DAC1/DAC2 noise/triangular wave generation disable */
dac_wave_generate(DAC1_SELECT, DAC_WAVE_GENERATE_NONE);
dac_wave_generate(DAC2_SELECT, DAC_WAVE_GENERATE_NONE);
/* DAC1/DAC2 output buffer disable */
dac_output_buffer_enable(DAC1_SELECT, FALSE);
dac_output_buffer_enable(DAC2_SELECT, FALSE);
/* DAC1/DAC2 DMA enable*/
dac_dma_enable(DAC1_SELECT, TRUE);
dac_dma_enable(DAC2_SELECT, TRUE);
/* DAC1/DAC2 enable*/
dac_enable(DAC1_SELECT, TRUE);
dac_enable(DAC2_SELECT, TRUE);
```

- DMA configuration code

```
/* The square wave array data is aligned with the DAC_HDR12RD register */
uint32_t dualsquare12bit[2] = {(0xfff | (0xfff << 16)), 0};
/* DMA1 CHANNEL1 configuration */
dma_reset(DMA1_CHANNEL1); /*Reset DMA1 channel1 and keep channel1 as default */
dma_init_struct.buffer_size = 2; /*Set DMA buffer length: to be the same as the array */
dma_init_struct.direction = DMA_DIR_MEMORY_TO_PERIPHERAL; /*Data transmission direction:
```

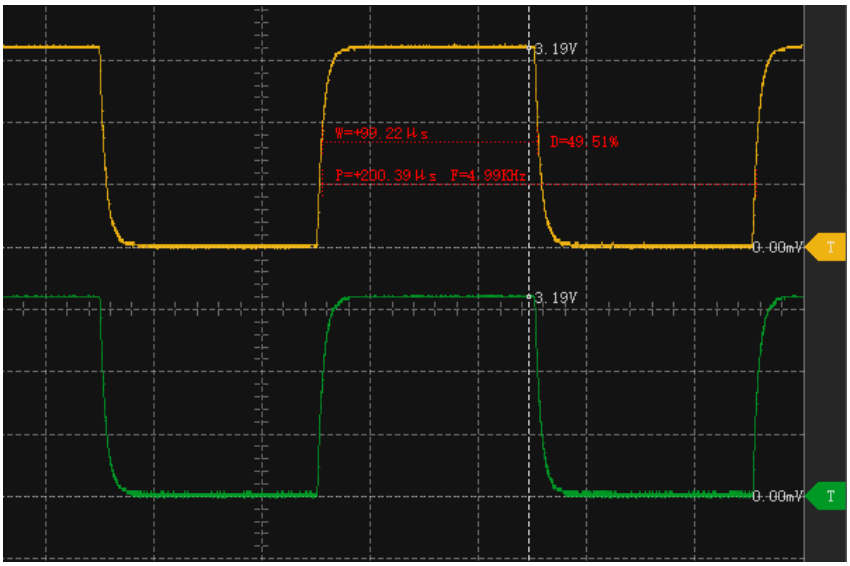
```
memory to peripheral */
    dma_init_struct.memory_base_addr = (uint32_t)dualsquare12bit; /* Memory address*/
    dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_WORD; /*Data width WORD*/
    dma_init_struct.memory_inc_enable = TRUE; /* Memory address increment: the memory address is added up each time the data is transmitted */
    dma_init_struct.peripheral_base_addr = (uint32_t)0x40007420; /* Peripheral DAC_HDR12RD register*/
    dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_WORD; /*Data width*/
    dma_init_struct.peripheral_inc_enable = FALSE; /* Peripheral address increment disabled, DAC_HDR12RD register*/
    dma_init_struct.priority = DMA_PRIORITY_MEDIUM; /*DMA priority: medium */
    dma_init_struct.loop_mode_enable = TRUE; /*Loop mode enable */
    dma_init(DMA1_CHANNEL1, &dma_init_struct); /* Set DMA1 channel1 as abovementioned */
    /*DMA1 CHANNEL1 flexible mapping to DAC2 */
    dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_DAC2);
    /*Enable DMA1 CHANNEL1 */
    dma_channel_enable(DMA1_CHANNEL1, TRUE);
```

- DAC software trigger enable code

```
    /* Trigger DAC1/DAC2 by software every 100us */
    while(1)
    {
      delay_us(100);
      dac_software_trigger_generate(DAC1_SELECT);
      dac_software_trigger_generate(DAC2_SELECT);
    }
```

3.2.4 Test result

Figure 7. Test result of double_mode_dma_squarewave



As shown in Figure 7, the square wave amplitude is about 3.19 V and the period is about 200us, basically meeting the program design expectations.

3.3 Example 3: DAC trapezoid wave output by using DMA

3.3.1 Function overview

Select TMR2 TRGO event to trigger DAC conversion, and use DMA to transfer the data in array to the DAC_D1DTH8R register to realize trapezoid wave output through PA4.

3.3.2 Resources

1) Hardware

   AT-START BOARD of the AT32F403A series

2) Software

   \project\at_start_f403a\examples\dac\one_dac_dma_escalator\mdk_v5

3.3.3 Software design

1) Configuration process
   - Configure clocks and the GPIO corresponding to DAC
   - Configure TMR/DAC/DMA

2) Code
   - Clock configuration code

```
/* Enable DMA1/DAC/TMR2/GPIOA clocks */
crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_DAC_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_TMR2_PERIPH_CLOCK, TRUE);
```

```
crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
```

- GPIO configuration code

```
/* Once the DACx channel is enabled, the corresponding GPIO pin (PA4/PA5) will be connected to
DACx_OUT automatically. To avoid parasitic interruption and excessive consumption, the PA4/PA5
should be configured to analog mode in advance.  */
gpio_init_struct.gpio_pins = GPIO_PINS_4; /*Select pin4*/
gpio_init_struct.gpio_mode = GPIO_MODE_ANALOG; /* Configure GPIO to analog mode */
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* Configure GPIO to push-pull mode */
gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* Configure GPIO to no pull-up and no pull-down */
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* Configure strong
GPIO drive capability*/
gpio_init(GPIOA, &gpio_init_struct); /* Configure PA4 according to the above settings */
```

- TMR configuration code

```
/* Get the system clock for TMR trigger frequency configuration */
crm_clocks_freq_get(&crm_clocks_freq_struct);
/* Initialize TMR2, upcounting mode, trigger frequency of 1kHz==
(systemclock/(systemclock/1000000))/1000  */
tmr_base_init(TMR2, 1000-1, (crm_clocks_freq_struct.sclk_freq/1000000 - 1));
tmr_cnt_dir_set(TMR2, TMR_COUNT_UP);
/* Master TMR2 output selection update OVERFLOW */
tmr_primary_mode_select(TMR2, TMR_PRIMARY_SEL_OVERFLOW);
/*EnableTMR2*/
tmr_counter_enable(TMR2, TRUE);
```

- DAC configuration code

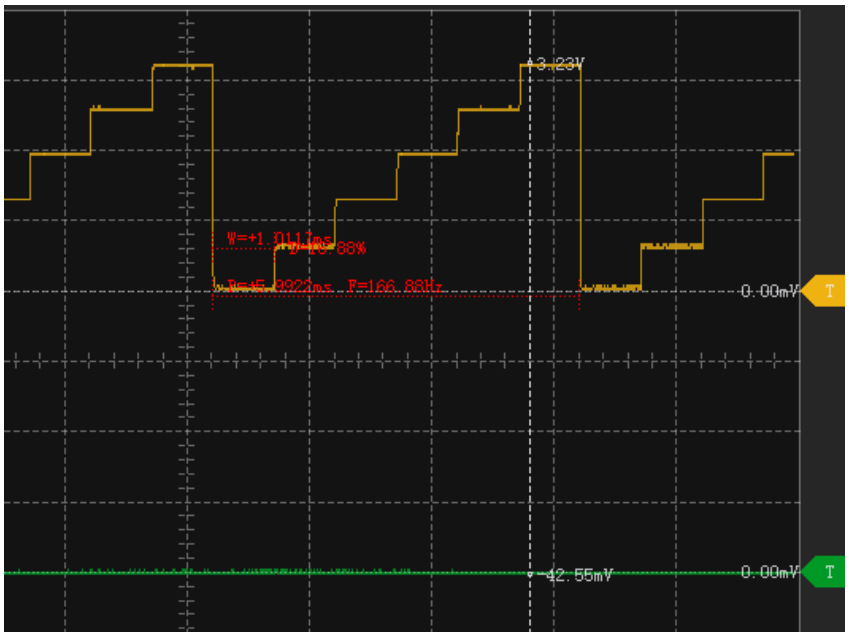
```
/* Select TMR2 TRGOUT as the trigger source of DAC1 */
dac_trigger_select(DAC1_SELECT, DAC_TMR2_TRGOUT_EVENT);
/* DAC1 trigger enable*/
dac_trigger_enable(DAC1_SELECT, TRUE);
/* DAC1 noise/triangular wave generation disable */
dac_wave_generate(DAC1_SELECT, DAC_WAVE_GENERATE_NONE);
/* DAC1 output buffer disable*/
dac_output_buffer_enable(DAC1_SELECT, FALSE);
/* DAC1 DMA enable*/
dac_dma_enable(DAC1_SELECT, TRUE);
/* DAC1 enable*/
dac_enable(DAC1_SELECT, TRUE);
```

- DMA configuration code

```
/* Trapezoid wave array data */
const uint8_t escalator8bit[6] = {0x0, 0x33, 0x66, 0x99, 0xCC, 0xFF};
/* DMA1 CHANNEL2 configuration */
dma_reset(DMA1_CHANNEL2); /*Reset DMA1 channel2 and keep channel2 at default*/
dma_init_struct.buffer_size = 6; /*Set DMA buffer length: to be the same as the array */
dma_init_struct.direction = DMA_DIR_MEMORY_TO_PERIPHERAL; /*Data transmission direction:
memory to peripheral */
dma_init_struct.memory_base_addr = (uint32_t) escalator8bit; /*Memory address*/
dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_ BYTE; /*Data width BYTE*/
dma_init_struct.memory_inc_enable = TRUE; /* Memory address increment: the memory address is
added up each time the data is transmitted */
dma_init_struct.peripheral_base_addr = (uint32_t)0x40007410; /* Peripheral DAC_D1DTH8R register*/
dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_ BYTE; /*Data width*/
dma_init_struct.peripheral_inc_enable = FALSE; /* Peripheral address increment disabled,
DAC_D1DTH8R register*/
dma_init_struct.priority = DMA_PRIORITY_MEDIUM; /*DMA priority: medium */
dma_init_struct.loop_mode_enable = TRUE; /*Loop mode enable*/
dma_init(DMA1_CHANNEL2, &dma_init_struct); /*Configure DMA1 channel2 as abovementioned*/
/*DMA1 CHANNEL2 flexible mapping to DAC1 */
dma_flexible_config(DMA1, FLEX_CHANNEL2, DMA_FLEXIBLE_DAC1);
/*Enable DMA1 CHANNEL1 */
dma_channel_enable(DMA1_CHANNEL2, TRUE);
```

3.3.4 Test result

Figure 8. Test result of one_dac_dma_escalator



As shown in Figure 8, the trapezoid wave amplitude is about 3.23 V and the period is about 6 ms (6*1/1kHz), basically meeting the program design expectations.

3.4 Example 4: DAC noise output

3.4.1 Function overview

Trigger DAC conversion by software, and DAN outputs a variable-amplitude pseudo noise through PA4 through the LENinear Feedback Shift Register (LFSR). In this example, to guarantee continuous output of noise, the DHRx is not configured and is kept at its default value (0x0, refer to 2.5 Noise/Triangular wave generation).

3.4.2 Resources

1) Hardware

   AT-START BOARD of the AT32F403A series

2) Software

   \project\at_start_f403a\examples\dac\one_dac_noisewave\mdk_v5

3.4.3 Software design

1) Configuration process
   - Configure clocks and the GPIO corresponding to DAC
   - Configure DAC
2) Code
   - Clock configuration code

```
/* Enable DAC/GPIOA clock */
crm_periph_clock_enable(CRM_DAC_PERIPH_CLOCK, TRUE);
```

```
crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
```

- GPIO configuration code

```
/* Once the DACx channel is enabled, the corresponding GPIO pin (PA4/PA5) will be connected to
DACx_OUT automatically. To avoid parasitic interruption and excessive consumption, the PA4/PA5
should be configured to analog mode in advance.  */
gpio_init_struct.gpio_pins = GPIO_PINS_4; /*Select pin4*/
gpio_init_struct.gpio_mode = GPIO_MODE_ANALOG; /* Configure GPIO to analog mode */
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* Configure GPIO to push-pull mode */
gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* Configure GPIO to no pull-up and no pull-down */
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* Configure strong
GPIO drive capability*/
gpio_init(GPIOA, &gpio_init_struct); /* Configure PA4 according to the above settings */
```

- DAC configuration code

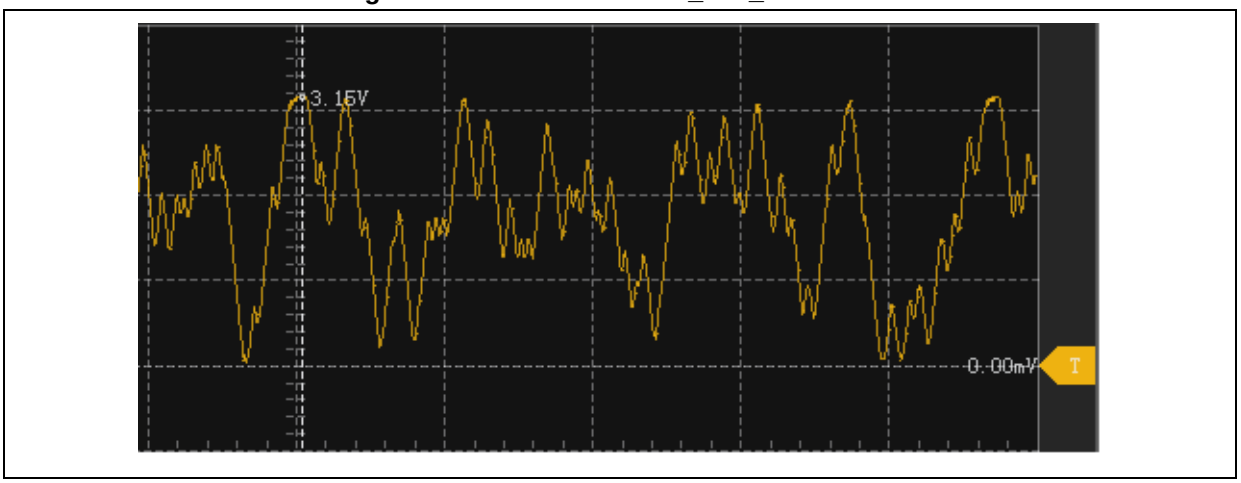
```
/* Select software to trigger DAC1 */
dac_trigger_select(DAC1_SELECT, DAC_SOFTWARE_TRIGGER);
/* DAC1 trigger enable*/
dac_trigger_enable(DAC1_SELECT, TRUE);
/* DAC1 noise generation enable*/
dac_wave_generate(DAC1_SELECT, DAC_WAVE_GENERATE_NOISE);
/* DAC1 noise bit select: Unmask LSFR bit [11:0] */
dac_mask_amplitude_select(DAC1_SELECT, DAC_LSFR_BITB0_AMPLITUDE_4095);
/* DAC1 output buffer enabled to respond the continuous noise in time */
dac_output_buffer_enable(DAC1_SELECT, TRUE);
/* DAC1 enable */
dac_enable(DAC1_SELECT, TRUE);
```

- DAC software trigger enable code

```
/* DAC1 software trigger generated continuously */
while(1)
{
  dac_software_trigger_generate(DAC1_SELECT);
}
```

3.4.4 Test result

Figure 9. Test result of one_dac_noisewave



As shown in Figure 9, the DAC1 randomly outputs pseudo noise with amplitude of 0-3.3V, basically meeting the program design expectations.

3.5 Example 5: Dual DAC triangular wave output

3.5.1 Function overview

Select TMR2 TRGO event to trigger DAC conversion, and use a triangle generator to generate triangular wave. In this example, the DHRx register value is set to 0x100, which is added up to the triangular-wave counter without overflow, and this value is then loaded into the DAC_D1ODT register (refer to 2.5 Noise/Triangular wave generation) and output through PA4/PA5.

3.5.2 Resources

1) Hardware

   AT-START BOARD of the AT32F403A series

2) Software

   \project\at_start_f403a\examples\dac\two_dac_trianglewave\mdk_v5

3.5.3 Software design

1) Configuration process
   - Configure clocks and the GPIO corresponding to DAC
   - Configure TMR/DAC

2) Code
   - Clock configuration code

```
/* Enable DMA1/DAC/TMR2/GPIOA clocks */
crm_periph_clock_enable(CRM_DAC_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_TMR2_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
```

- GPIO configuration code

```
/* Once the DACx channel is enabled, the corresponding GPIO pin (PA4/PA5) will be connected to
DACx_OUT automatically. To avoid parasitic interruption and excessive consumption, the PA4/PA5
should be configured to analog mode in advance. */
gpio_init_struct.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5; /*Select pin4 and pin5*/
gpio_init_struct.gpio_mode = GPIO_MODE_ANALOG; /* Configure GPIO to analog mode */
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* Configure GPIO to push-pull mode */
gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* Configure GPIO to no pull-up and no pull-down */
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* Configure strong
GPIO drive capability*/
gpio_init(GPIOA, &gpio_init_struct); /* Configure PA4/PA5 according to the above settings */
```

- TMR configuration code

```
/* Get the system clock for TMR trigger frequency configuration */
crm_clocks_freq_get(&crm_clocks_freq_struct);
/* Initialize TMR2, upcounting mode, trigger frequency of 10kHz==
(systemclock/(systemclock/1000000))/100 */
tmr_base_init(TMR2, 100-1, (crm_clocks_freq_struct.sclk_freq/1000000 - 1));
tmr_cnt_dir_set(TMR2, TMR_COUNT_UP);
/* Master TMR2 output selection update OVERFLOW */
tmr_primary_mode_select(TMR2, TMR_PRIMARY_SEL_OVERFLOW);
/*Enable TMR2*/
tmr_counter_enable(TMR2, TRUE);
```

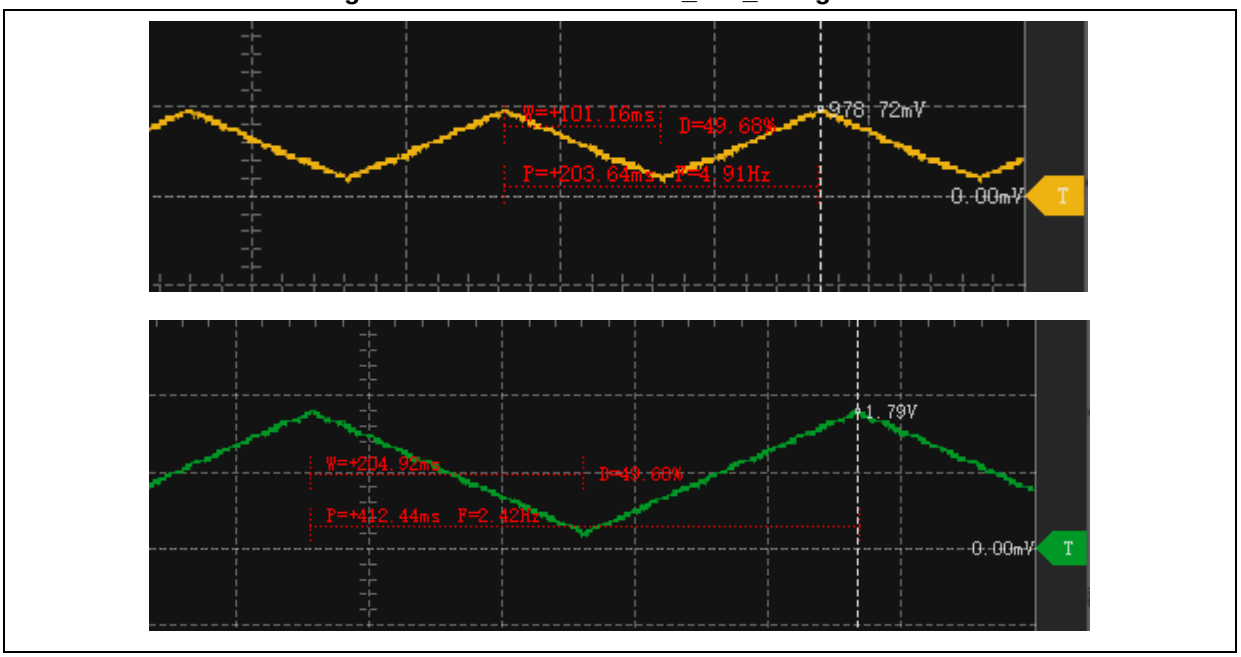
- DAC configuration code

```
/* Select TMR2 TRGOUT to trigger DAC1/DAC2 */
dac_trigger_select(DAC1_SELECT, DAC_TMR2_TRGOUT_EVENT);
dac_trigger_select(DAC2_SELECT, DAC_TMR2_TRGOUT_EVENT);
/* DAC1/DAC2 trigger enable*/
dac_trigger_enable(DAC1_SELECT, TRUE);
dac_trigger_enable(DAC2_SELECT, TRUE);
/* DAC1/DAC2 triangular wave generation enable*/
dac_wave_generate(DAC1_SELECT, DAC_WAVE_GENERATE_TRIANGLE);
dac_wave_generate(DAC2_SELECT, DAC_WAVE_GENERATE_TRIANGLE);
/* DAC1 noise bit select: triangle amplitude is equal to 1023, formula: 3.3V*1023/4095=824.4mV*/
dac_mask_amplitude_select(DAC1_SELECT, DAC_LSFR_BIT90_AMPLITUDE_1023);
/* DAC2 noise bit select: triangle amplitude is equal to 2047, formula: 3.3V*2047/4095=1649.6mV */
dac_mask_amplitude_select(DAC2_SELECT, DAC_LSFR_BITA0_AMPLITUDE_2047);
/* DAC1/DAC2 output buffer disable */
```

```
dac_output_buffer_enable(DAC1_SELECT, FALSE);
dac_output_buffer_enable(DAC2_SELECT, FALSE);
/* DAC1/DAC2 enable */
dac_enable(DAC1_SELECT, TRUE);
dac_enable(DAC2_SELECT, TRUE);
/* DHRx is set to 0x100, formula 3.3V*0x100/0xfff=206.3mV*/
dac_dual_data_set(DAC_DUAL_12BIT_RIGHT, 0x100, 0x100);
```

3.5.4 Test result

Figure 10. Test result of two_dac_trianglewave



As shown in Figure 10, the DAC1 (yellow) outputs triangular wave with amplitude of 978.72 mV (206.3mV+824.4mV) and period of 204.8 ms (1024*2*1/10kHz); the DAC2 (green) outputs triangular wave with amplitude of 1.79 V (206.3mV+1649.6mV) and period of 409.6 ms (2048*2*1/10kHz), basically meeting the program design expectations.

4 Revision history

Table 2. Document revision history

Date	Version	Revision note
2022.01.10	2.0.0	Initial release.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services, and ARTERY assumes no liability whatsoever relating to the choice, selection or use of the ARTERY products and services described herein.

No license, express or implied, to any intellectual property rights is granted under this document. If any part of this document deals with any third party products or services, it shall not be deemed a license grant by ARTERY for the use of such third party products or services, or any intellectual property contained therein, or considered as a warranty regarding the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

Unless otherwise specified in ARTERY's terms and conditions of sale, ARTERY provides no warranties, express or implied, regarding the use and/or sale of ARTERY products, including but not limited to any implied warranties of merchantability, fitness for a particular purpose (and their equivalents under the laws of any jurisdiction), or infringement of any patent, copyright or other intellectual property right.

Purchasers hereby agrees that ARTERY's products are not designed or authorized for use in: (A) any application with special requirements of safety such as life support and active implantable device, or system with functional safety requirements; (B) any air craft application; (C) any automotive application or environment; (D) any space application or environment, and/or (E) any weapon application. Purchasers' unauthorized use of them in the aforementioned applications, even if with a written notice, is solely at purchasers' risk, and is solely responsible for meeting all legal and regulatory requirement in such use.

Resale of ARTERY products with provisions different from the statements and/or technical features stated in this document shall immediately void any warranty grant by ARTERY for ARTERY products or services described herein and shall not create or expand in any manner whatsoever, any liability of ARTERY.