7. Digital to analog converter

The use of a DAC is demonstrated.

7.1. Hardware – a DAC block

There are two 12-bit digital to analog converters (DAC) available. A principal block diagram for one of the two identical DACs is given in Fig. 7.1 (Fig. 64, RM0090, pg. 431). A letter 'x' can be '1' or '2', depending on the DAC in question.

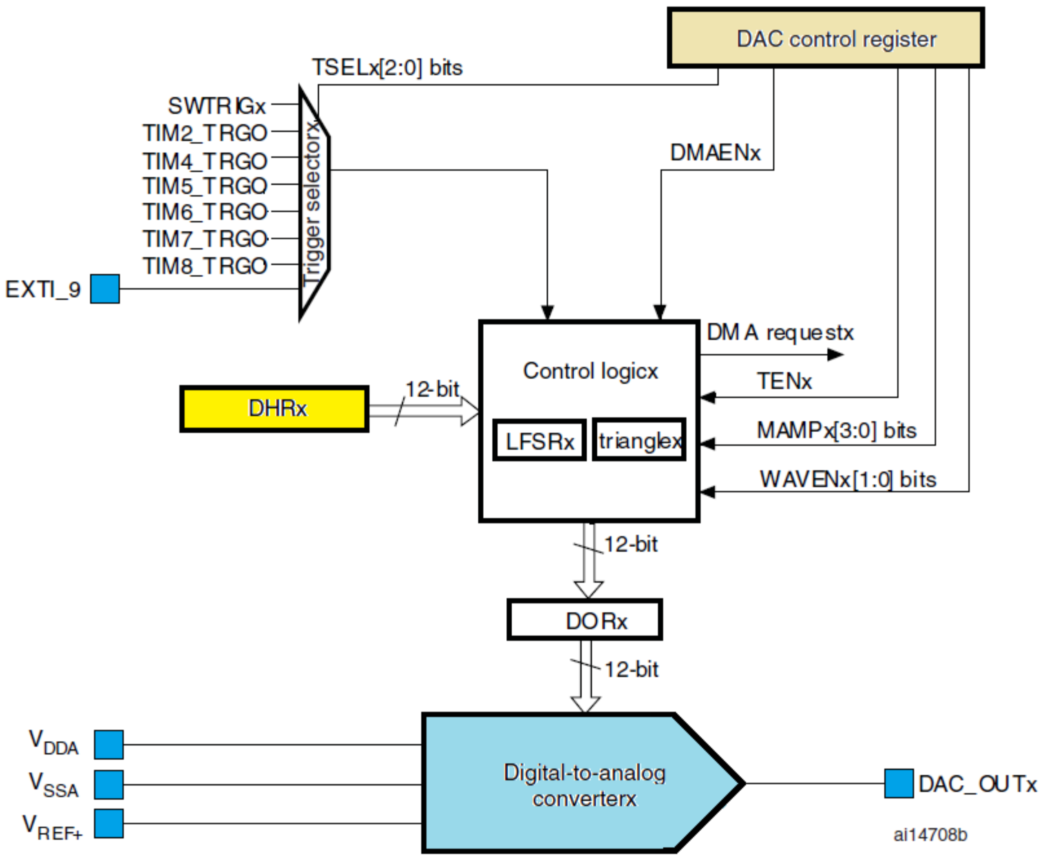

Figure 7.1: The block diagram of a single DAC

The actual DAC is shown at the bottom. Its operation is supported by power supply lines V_{DDA} and V_{SSA}, and the reference signal V_{REF+}, all coming from pins of the microcontroller on the left. The output of the DACx is available at a pin to the right DAC_OUTx. The user program loads data (to be converted to analog signal) into the data holding register DHRx from where it is copied into register DORx under a hardware control to be instantly converted into analog signal by the DAC. The hardware enclosed in block Control logic defines the moment of copying.

The register DHRx is a 32-bit wide register, but can be accessed as a whole (DAC_DHR12RD, 12 bits for each DAC, data right aligned in both halves of the 32-bit register) or each half separately (DAC_DHR12R1 and DAC_DHR12R2, 12 bits for each DAC1 or DAC2, data right aligned in a 32-bit register). Each half of the register DHR is responsible for one DAC, see the details in RM0090, page 445 to 451.

The moment of transferring the data from register DHR into DORx can be automatic, but can also be defined by software (SWTRIGx), hardware (TIMx_TRGO) or external signals (EXTI_9), as selected by the multiplexor on the left above the register DHRx. Such organization allows the synchronization of signals out of the two DAC with a selected event. There is also a possibility to transfer the data using a direct memory access.

Additional hardware is provided in Control logicx box to generate noise or triangle wave signal to be added to the content of the register DHR. This will not be used in this experiment.

7.2. Software to test the DAC

Several bits distributed along control registers for the DAC are responsible for the behavior of the two DACs, but the programmer should know their location and detailed function. To ease the programming the CMSIS library will be used. The functions to actually configure the operation of DACs are stored in the source file “stm32f4x_dac.c”, and the definitions and data structures to go along are stored in the header file “stm32f4x_dac.h”.

The recipe to use the DAC is given in the source file under section ‘How to use this driver’:

- Configure pins where the two DAC outputs are connected, these are port A, pins 4 and 5 for STM32F407VG microcontroller. First enable the clock for port A, then put pins 4 and 5 in analog mode, and disable terminating resistors.
- Enable the clock for the DAC block.
- Configure both DACs.
- Enable both DACs.

The DACs can be used afterwards by simply writing new digital value into the DHR register; corresponding analog value will appear at port A, pins 4 and 5.

The recipe is implemented in a function “DACinit()” listed below.

```
void DACinit (void)     {
GPIO_InitTypeDef        GPIO_InitStructure;                                                    // 2
DAC_InitTypeDef         DAC_InitStructure;                                                     // 3

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);                                        // 5
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4 | GPIO_Pin_5;                                     // 6
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;                                                // 7
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                            // 8
  GPIO_Init(GPIOA, &GPIO_InitStructure);                                                       // 9

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);                                          // 11
  DAC_InitStructure.DAC_Trigger        = DAC_Trigger_None;                                     // 12
  DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;                              // 13
  DAC_InitStructure.DAC_OutputBuffer   = DAC_OutputBuffer_Enable;                              // 14
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);                                                 // 15
  DAC_Init(DAC_Channel_2, &DAC_InitStructure);                                                 // 16

  DAC_Cmd(DAC_Channel_1, ENABLE);                                                              // 18
  DAC_Cmd(DAC_Channel_2, ENABLE);                                                              // 19
}
```

Two data structures are needed, and these are declared in lines 2 and 3. The first data structure is required to initialize port A with a function call in line 9. The second data structure is required to initialize DAC blocks with a function calls in lines 15 and 16.

Line 5 enables the clock for port A, and lines 6 to 8 fill the data structure that will serve to initialize the port. Note that the setting is applied to pins 4 and 5 only, and that the selected mode is analog (“GPIO_Mode_AN”). Terminating resistors at these pins are disabled (“GPIO_PuPd_NOPULL”).

Line 11 enables the clock for the DAC block. Next comes the initialization of members of the second data structure. There are four members altogether, as can be seen in a header file “stm32f4x_dac.h”, lines 54 to 69:

- Member ‘.DAC_Trigger’ tells the hardware enclosed in block Control logic when to copy data from the Data holding register to Data output register. In our case the hardware will copy data immediately after to software sends new value into register DHR, therefore option ‘DAC_Trigger_None’ is used. All available options are listed in the header file, lines 81 to 91. Basically, signals connected to the multiplexor, Fig. 7.1 top left, can be used to initiate the copying. These signals are coming either from timers inside the microcontroller, from external sources, or can be defined by software. In any case option other than the one used in the listing above assures an update of DAC outputs which is synchronous with the selected event.
- Member ‘.DAC_WaveGeneration’ allows the hardware enclosed in block Control logic to autonomously generate a triangular signal or noise and add it to the value specified in register DHR. Such option is convenient for special measurements, but will not be used in this example. To disable the option this member is initialized to ‘DAC_WaveGeneration_None’, other options are listed in the header file, lines 111 to 113.
- Member ‘.DAC_LFSRUnmask_TriangleAmplitude’ can be used to define properties of the added triangular or noise signal. Since in our case these were not used, we do not need to initialize this member.
- Member ‘.DAC_OutputBuffer’ determines the use of an additional buffer stage between the DAC and the pin of microprocessor. When this buffer is used the output impedance is reduced allowing stronger loading of the DAC, but increasing the power consumption of the microcontroller. In this example the output buffers are enabled by initializing the member with ‘DAC_OutputBuffer_Enable’.

This data structure is then used to configure both DACs in two consecutive function calls “DAC_Init()”, lines 15 and 16. Note that the first argument of the function specifies the DAC to be affected, and the second argument is a pointer to the data structure.

The next two lines are used to enable both DACs using the command “DAC_Cmd()” with two arguments again. The first specifies the DAC in question, and the second commands the enabling.

The demo program for the two DAC generates two sawtooth signals by writing consecutively increasing numbers to Data holding register for DAC1 (‘DAC->DHR12R1’) and consecutively decreasing numbers to Data holding register for DAC2 (‘DAC_DHR12R2’). These two writes could be replaced by two CMSIS function calls “DAC_SetChannel1Data()”, but the version used in this example is faster (and less portable). The listing of the program is given below. Note the additional include file “stm32f4x_dac.c”.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"

int main ()  {
```

```
unsigned int j;

  DACinit();

  while (1) {
    DAC->DHR12R1 = j;            // up ramp
    DAC->DHR12R2 = 0xfff - j;   // down ramp
    j = (j + 1) & 0x0fff;
  };
};
```