# 16. Signal generation using DDS

A technique known as Direct Digital Synthesis (DDS) will be implemented to demonstrate the use of interrupts and DAC to generate a signal with the desired frequency.

Consider a 16-bit wide register. Its content is increased periodically (the period is given as $T_P$) by a factor *K*. Obviously the time *T* needed to overflow the register equals:

$$T = T_P \cdot \frac{2^{16}}{K}$$

If we keep increasing the content of the register then the overflows will repeat at regular time intervals, and the frequency of overflows *f* can be calculated as:

$$f = \frac{1}{T} = \frac{1}{T_P} \cdot \frac{K}{2^{16}} = f_P \cdot \frac{K}{65536}$$

where $f_P$ represents the number of increases per unit of time. Such structure is easy to implement in a microcontroller. One only needs a timer and associated interrupt function. The timer defines the period $T_P$, and periodically calls an interrupt function, where a variable declared as an unsigned integer gets increased by a factor *K*. The frequency of overflows is linearly dependant of factor K, and it changes for about 15.26E-6 times $f_P$. If we select the frequency of interrupt requests from the timer ($f_P$) at 100 kHz, then we will be able to define the frequency of overflows linearly in steps of 1.5 Hz.

If we consider the content of the register as the output signal, then we get a sawtooth generator. However, we can use the content of the register as a pointer to a table, which can be filled by any waveshape during the initialization process, like for instance one period of a sinewave. If we consider the entry from a table, pointed to by the content of the register, as the output signal, we have a sinewave generator. The frequency of the signal can still be adjusted in small steps, and the amplitude can be adjusted by a simple multiplication of the entry by a constant before it gets passed to the DAC.

It does not seem rational to prepare a huge table with 65536 entries, since neighboring entries will be very similar. We can prepare smaller table, and use only the bits from the upper part of the register as a pointer; we intend to generate signals within the audio range from 20 Hz on, and the factor *K* will be about 16 at least. The decision on the table length depends on the required precision, and we will use a table with 4096 entries in this example. It would be possible to reduce the size of this table to ¼ by exploiting the properties of a sinewave, but we will not do this for the reason of simplicity. The block diagram of the hardware that could be used as a DDS generator is shown in Fig. 1.

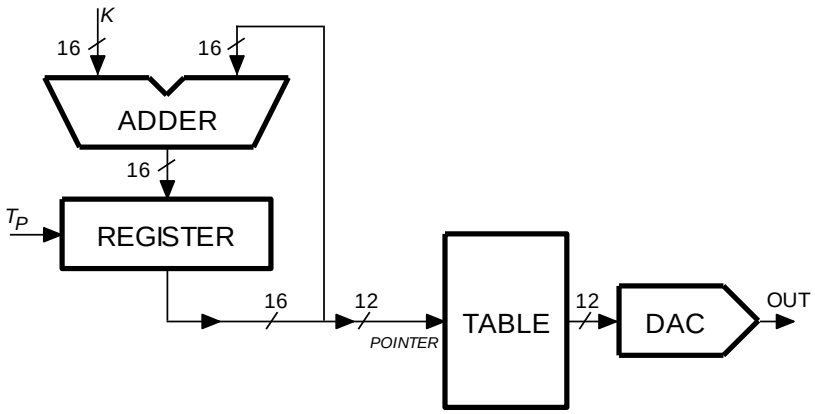


*Figure 1: A hardware implementation of the DDS technique*

The same technique is widely used in contemporary digital receiver units, and width of registers is increased to about 40 bits as is the frequency of increasing to some hundreds of MHz to achieve a stable output signal with a frequency of above 100 MHz that can be set within few mHz! By expanding this technique it is very simple to generate a modulated signal (AM, FM, PM, …), and also to define new shapes of signals as well as to generate multiple signals with different frequencies. There are specialized integrated circuits available to do this.

The complete listing of the program is given in Fig. 2. The program starts with the declaration of the table with 4096 integer elements, the pointer to this table, and some variables to define the amplitude Am and factor K. Those must be declared as global since they are used in the interrupt function. The program then proceeds to fill the table with entries following one period of a sinewave. The values range from -1850 to +1850, allowing for some space to accommodate the saturation voltage of the operational amplifier at the output from the DAC. Floating point arithmetic must be used to calculate sin function. Next is the initialization of clocks for peripherals, ports, LCD display, DAC, timer and the controller NVIC, as it was already explained in previous chapters. This initialization is simply copied from examples given before. The program then enters an endless loop where it periodically reads pushbuttons S372 to S375, increases or decreases the variables defining the amplitude and the frequency of the output signal, and writes on LCD.

The interrupt function is given next. It is properly named as required by the interrupt vector table as “TIM2_IRQHandler”. Both port accesses at the beginning and at the end are inserted to allow the determination of the execution time of this function using an oscilloscope, and could be removed. The important body starts by clearing the interrupt flag in the timer TIM2 and calculating the new pointer into the table. A factor K is added to the pointer, and its value is bound by 65536. Next the calculated pointer is used to retrieve correct element of the table, and only upper 12 bits of the pointer are used (this is the “ >> 4 ” operation). The retrieved element is multiplied by a variable Am (amplitude), and the resulting number divided by 256 to remain in the range of the DAC. The DAC can handle positive numbers only, so half of the DAC range (2048) is still added before sending the number to its destination.

In order to show the possibility of phase modulation (and also to allow further experiments with quadrature signals) the second DAC is used to generate a sinewave signal which is 90 degrees out of phase with the already generated signal. This is accomplished by retrieving the element from the table which is for ¼ of the table length away from the current pointer. Upper 12 bits of the pointer are taken, and 1024 is added to it. The sum might point outside of the table, so the sum is corrected

to remain within the bounds of the table by AND-ing it with the pointer to the last valid element (4095), the rest is the same as for the first DAC.

The interrupt function takes about 500ns to execute, and the quality of the generated signal could be much improved by increasing the frequency of interrupt requests.

```
#include "stm32f4xx.h"
#include "LCD2x16.c"
#include "math.h"

int Table[4096], TablePtr, Am = 255, K = 655;   // declare global variables

int main ()  {

  // Table init
  for (TablePtr = 0; TablePtr <= 4095; TablePtr++)
    Table[TablePtr] = (int)(1850.0 * sin((float)TablePtr / 2048.0 * 3.14159265));

  // GPIO clock enable, digital pin definitions
  RCC->AHB1ENR |= 0x00000001;   // Enable clock for GPIOA
  RCC->AHB1ENR |= 0x00000010;   // Enable clock for GPIOE
  GPIOE->MODER |= 0x00010000;   // output pin PE08: time mark for TIM2 IRQ

  // LCD init
  LCD_init(); LCD_string("f =", 0x01); LCD_string("Am=", 0x41);

  // DAC set-up
  RCC->APB1ENR |= 0x20000000;   // Enable clock for DAC
  DAC->CR      |= 0x00010001;   // DAC control reg, both channels ON
  GPIOA->MODER |= 0x00000f00;   // PA04, PA05 are analog outputs

  // Timer 2 set-up
  RCC->APB1ENR |= 0x0001;       // Enable clock for Timer 2
  TIM2->ARR     = 840;          // Auto Reload: 8400 == 100us -> 100kHz
  TIM2->DIER   |= 0x0001;       // DMA/IRQ Enable Register - enable IRQ on update
  TIM2->CR1    |= 0x0001;       // Enable Counting

  // NVIC IRQ enable
  NVIC_EnableIRQ(TIM2_IRQn);     // Enable IRQ for TIM2 in NVIC

  // waste time - check pushbuttons and display parameters
  while (1) {
    if ((GPIOE->IDR & 0x0004) && (K  < 32767))  K  += 1;        // IF S372...
    if ((GPIOE->IDR & 0x0008) && (K  > 2))      K  -= 1;        // IF S373...
    if ((GPIOE->IDR & 0x0010) && (Am < 255))    Am += 1;        // IF S374...
    if ((GPIOE->IDR & 0x0020) && (Am > 1))      Am -= 1;        // IF S375...
    int Fp = (int)(1.0e5 * (float)K / 65536.9);           // calculate frequency
    LCD_uInt16(Fp,0x08,1);  LCD_uInt16(Am,0x48,1);        // display Fp, Ap
  };
}

// IRQ function
void TIM2_IRQHandler(void)       // IRQ function takes approx 500ns of CPU time!
{
  GPIOE->ODR |=  0x0100;                    // PE08 up
  TIM2->SR &= ~0x00000001;                  // clear update event flag in TIM2
  TablePtr = (TablePtr + K) & 0xffff;       // increase pointer and limit to 16b
  DAC->DHR12R1 = (Am * Table[  TablePtr >> 4                ]) / 256 + 2048;
  DAC->DHR12R2 = (Am * Table[((TablePtr >> 4) + 1024) & 4095]) / 256 + 2048;
  GPIOE->ODR &= ~0x0100;                    // PE08 down
}
```

*Figure 1: A listing of the program for sinewave signal generation – DDS method*