13. The use of circular buffer

A circular buffer is a reserved part of memory which serves as a temporary buffer for data. It is organized in a special manner: the incoming data fills the buffer until it is completely full, and then starts to fill the buffer again from the beginning overwriting the previous content. It is the task of the software to make use of the stored data before it gets overwritten by new data.

It is customary to build a circular buffer using an array and a pointer into this array. However, the pointer must be bound to point into the array and never outside of it since accessing the area outside of the array might have disastrous effects to the execution of the program. For instance the operation of writing into the circular buffer using a pointer, where the pointer points outside of the array, might corrupt the content of important memory locations, like registers or system variables. This must be prevented.

Consider the use of an array with the length of 2^N, where N is a natural number. For the purpose of this example N equals 3, and we are dealing with an array with eight elements. Their indexes are from 0 to 7 in decimal notation, or from 000 to 111 in binary. If we take any integer number named *Ptr* written in binary, and use only the least significant three bits of this number as a pointer, these three bits point to one of the elements within the array. Therefore any integer number can be used as a valid pointer once we strip away all but the least significant three bits, and this can be done by a simple AND operation, where one of the arguments of the AND operation is the integer itself, and the other is the last available index within the array, 7 (111 binary) in our case.

Now consider incrementing the number *Ptr* by one, starting from 0, and its behavior as a pointer in the array. The pointer derived by AND-ing the number *Ptr* by 7 initially points to element 0 of the array, then element 1, then…. then element 7, and then element 0 again. The indexed elements form a circle, element 0 follows the element 7.

Consider also decrementing the number *Ptr* by one, starting from 2, and use 8-bit binary notation to ease the understanding. The number decrements from 00000010b to 0000001b, then to 00000000b, followed by 11111111b, 11111110b, and 11111101b… Recall that numbers are written in two's complement when written as signed integers! The number 11111111b represents -1_{10}, and 11111110b represents -2_{10}. Taking the least significant three bits of the number *Ptr* again keeps the derived pointer within the bounds of the array, and uses elements 2, 1, 0, 7, 6, … as before, the use of element 0 is followed by the use of element 7, then 6…, and the indexed elements again form a circle, hence the array used in this way can be called a circular buffer.

The trick relies on a simple AND logic function, and is effective for circular buffers with the length of 2^N elements. Other lengths need more complex bounding of the pointer, and are best avoided.

We implement a circular buffer in this example to delay the generation of a signal in one of the DACs. The program is based on the one last one derived in previous chapter, and the initialization of the hardware is identical and will not be re-commented here. One of the ADCs is used to periodically sample the input signal (actually both ADCs are initialized and run, but result from one of them is used), and these samples are stored in a circular buffer named *Result*. One of the DACs is filled directly with the ADC result, and the other is filled with whatever the result were 100 sampling intervals before. The complete listing is given in Fig. 1.

```
#include "stm32f4xx.h"

int Result[1024], Ptr = 0;

void main ()  {
  // 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
  GPIOA->MODER |= 0x00001000;   // output pin PA06: LED D390

  // 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

  // ADC set-up
  RCC->APB2ENR |= 0x00000100;   //  clock for ADC1
  RCC->APB2ENR |= 0x00000200;   //  clock for ADC2
  ADC->CCR      = 0x00000006;   // Regular simultaneous mode only
  ADC1->CR2     = 0x00000001;   // ADC1 ON
  ADC1->SQR3    = 0x00000002;   // use PA02 as input
  ADC2->CR2     = 0x00000001;   // ADC1 ON
  ADC2->SQR3    = 0x00000003;   // use PA03 as input
  GPIOA->MODER |= 0x000000f0;   // PA02, PA03 are analog inputs

  ADC1->CR2    |= 0x06000000;   // use TIM2, TRG0 as SC source
  ADC1->CR2    |= 0x10000000;   // Enable external SC, rising edge
  ADC1->CR1    |= 0x00000020;   // Enable ADC Interrupt for EOC

  // NVIC IRQ enable
  NVIC_EnableIRQ(ADC_IRQn);     // Enable IRQ for ADC in NVIC

  // Timer 2 set-up
  RCC->APB1ENR |= 0x0001;       // Enable clock for Timer 2
  TIM2->ARR     = 8400;         // Auto Reload value: 8400 == 100us
  TIM2->CR2    |= 0x0020;       // select TRGO to be update event (UE)
  TIM2->CR1    |= 0x0001;       // Enable Counting

  // waste time - indefinite
  while (1) {
    if (GPIOE->IDR & 0x0001) GPIOA->ODR |=  0x0040;   // LED on
    else                     GPIOA->ODR &= ~0x0040;   // else LED off
  };
}

// IRQ function
void ADC_IRQHandler(void)       // PASS takes approx 400ns of CPU time!
{
  GPIOE->ODR   |=  0x0100;      // PE08 up
  Ptr = (Ptr + 1) & 1023;       // Increment pointer and limit its value
  Result[Ptr] = ADC1->DR;       // save ADC in circular buffer & clear EOC flag
  DAC->DHR12R1  = Result[Ptr];               // pass current buffer -> DAC
  DAC->DHR12R2  = Result[(Ptr - 100) & 1023]; // pass past buffer -> DAC
  GPIOE->ODR   &= ~0x0100;      // PE08 down
}
```

Figure 1: A listing of the program to periodically sample input signals and pass them to the DAC

The changes in the program are:

- An array named *Result* is declared as global, it consists of 1024 elements, their indexes ranging from 0 to 1023. A pointer named *Ptr* is declared as global and initialized to 0. Both must be declared global since they are used in the interrupt function.
- The complete initialization section is the same as before.
- The interrupt function starts with a statement to calculate new value of the pointer in array. Variable *Ptr* is first incremented, and then bound to stay within the range from 0 to 1023 forming a current pointer. The result is stored into the array at current pointer. The content of this element of the array is copied into the first DAC. The content of the element which was renewed 100 sampling time intervals is copied into the second DAC. The pointer to the corresponding element of the array is calculated by subtracting 100 from a current pointer and bounding the value of the derived pointer by AND-ing it with the index of the last valid element (1023).