13. The use of a circular buffer

A digital processing algorithms rely heavily on present and past samples of signals. The standard approach to memorize past values is the use of a circular buffer. Such structure will be described and its use will be demonstrated to generate a delayed version of a signal.

13.1. The background on storing past samples

A circular buffer is a reserved part of memory which serves as a temporary storage for data. It is organized in a special manner: the incoming data fills the reserved space until completely full, and then starts to fill the reserved space 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, and the task of the programmer to reserve sufficient memory for the circular buffer.

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, under any circumstances, allow the access of data outside of the array 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, and can be accomplished by two 'if' statements, see the simplified listing bellow.

```
int X[100], ptrX
….
  if (ptrX > 99) ptrX = 99;
  if (ptrX <  0) ptrX = 0;
…
```

In simple words: if the array has 100 elements, then the pointer should never be less than 0 or more then 99. These two 'if'statements should be inserted just before any use of the pointer 'ptrX' to access the content of the array 'X[ptrX]'. In a real processing this bounding of a pointer may look even more complex. The inclusion of these lines will slow down the execution of the program, and make the code less readable, so a better way is searched for.

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 the array has eight elements. Their indexes are from 0 to 7 in decimal notation, or from 00000000 to 00000111 in binary (8 bits notation). 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 this 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 (00000111 binary) in our case. To be re-emphasized: this trick works only for arrays with the length of 2^N!

Now consider incrementing the variable 'Ptr' by one starting from 0, and its behavior as a pointer in the array. The pointer derived by AND-ing the variable '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 7 is followed by element 0.

Consider now decrementing the variable 'Ptr' by one, starting from, say, 2. The variable decrements from 00000010_2 to 0000001_2, then to 00000000_2, followed by 11111111_2, 11111110_2, and 11111101_2… Recall that values are written in two's complement when written as signed integers! The value 11111111_2 represents -1_{10}, and 11111110_2 represents -2_{10}. Taking the least significant three bits of the variable 'Ptr' again keeps the derived pointer within the bounds of the array, and selects elements 2, 1, 0, 7, 6, … and the element 0 is followed by the 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 logic operation AND, and is effective for buffers with the length of 2^N elements. Other lengths need more complex bounding of the pointer, and are best avoided.

13.2. The implementation of a circular buffer

We shall implement a circular buffer to delay the generation of a signal. The program is based on the one derived in chapter 12, and the initialization of the hardware is identical and will not be re-discussed here. Both ADCs are used to periodically sample two input signals, and results from both ADS are stored in a circular buffers named 'X1[]' and 'X2[]'. One of the DACs is filled with the current result of the conversion, and the other is filled with whatever the result was 10 sampling intervals before. The listing is given below, the initialization functions are to be copied from chapter 12.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"

short int X1[1024], X2[1024], ptrX = 0;

int main ()  {
  ADCinit_T5_CC1_IRQ();
  DACinit();
  TIM5init_TimeBase_CC1(840);           // 840 == 100us

  while (1) {                           // endless loop
  };
}

void ADC_IRQHandler(void)       //
{
  ptrX++; ptrX &= 1023;
  X1[ptrX] = ADC1->DR;
  X2[ptrX] = ADC2->DR;
  DAC->DHR12R1 = X1[ptrX];
  DAC->DHR12R2 = X1[(ptrX-10) & 1023];
}
```

The distinctive details are:

- Two arrays named ‘X1[1024]’ and ‘X2[1024]’ are declared as global, they consists of 1024 elements (2^{10}) each, their indexes ranging from 0 to 1023. A pointer named ‘ptrX’ is declared as global and initialized to 0. Note that all variables used in an interrupt function must be declared as global.
- The interrupt function starts with a statement to calculate new value of the pointer in array. Variable ‘ptrX’ is first incremented, and then bound to stay within the range from 0 to 1023 forming a current pointer. Two results from ADCs are stored into arrays at current pointer. The content of current element of the array is copied into the first DAC. The content of the element which was stored into the array 10 sampling time intervals before is to be copied into the second DAC. The pointer to this element is calculated by subtracting 10 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_{10} or 0x3ff in hex).