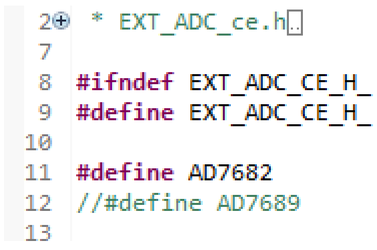
EXT_ADC Library Manual

This library is a microcontroller driver of the STM32F4 family for data exchange with AD7682 / 7689 ADC. To choose between AD7682 or AD7689, you need to use the #define directive in the file "EXT_ADC_ce.h":



```
 * EXT_ADC_ce.h

#ifndef EXT_ADC_CE_H_
#define EXT_ADC_CE_H_

#define AD7682
//#define AD7689

```

The library structure is shown in the following figure, along with libraries that are included using the #include directive for implementing circular buffers (**circular_buffers_lib**), filtering measured data (**data_processing_lib**), and SPI interface settings:

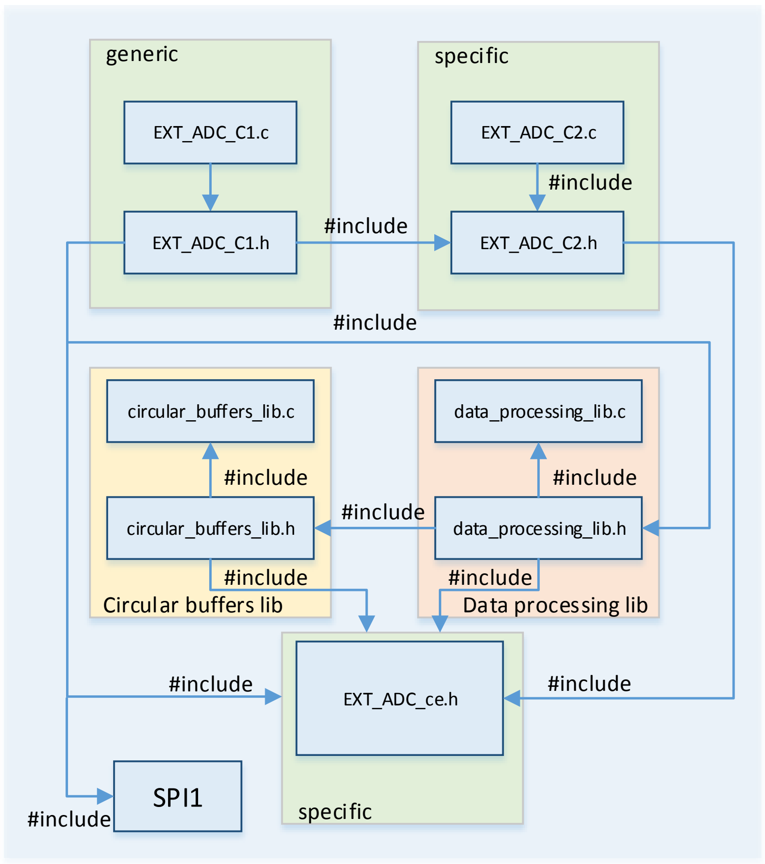


Figure 1 - File structure

Before continuing to review the library in more detail, you need to read the time diagrams from the AD7682 / 7689 ADC documentation:

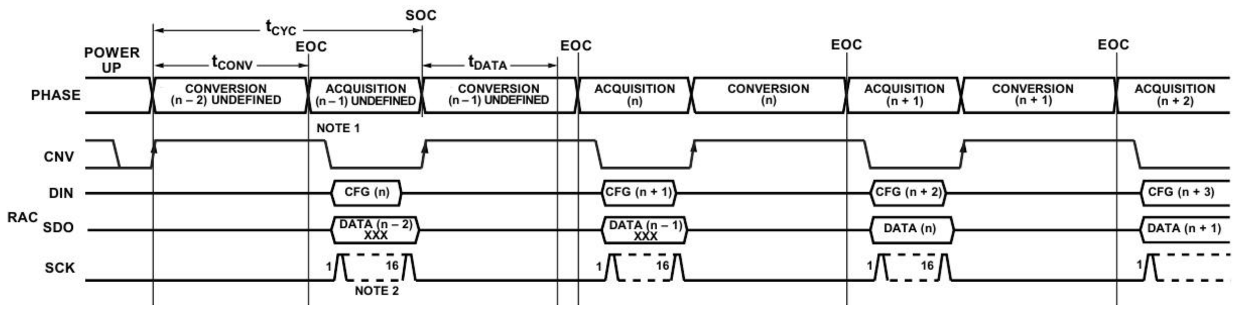


Figure 2 – Timing diagrams without a busy indicator

The **conversion** is triggered by the rising edge of the CNV line, the conversion is followed by the data **acquisition** phase, this is the time interval when receiving and transmitting via SPI occurs. As can be seen from the diagrams, the data to be sent is the value to be written to the Configuration Register, one of the fields of this register and determines from which input channel the measurement will occur.

Also, as can be seen from the diagrams, the data is provided with a shift of 2, since after switching on the power, the value of the Configuration Register register is not defined and requires two transactions before it becomes one.

There are two transmission modes: with and without a busy indicator. The transfer mode with the readiness indicator is when the ADC sets the SDO line to zero when the conversion is completed. In this mode, before starting a new conversion, an additional 17th clock pulse is needed; this is necessary in order for the ADC to transfer the SDO line to the Z-state.

In order for the ADC to work in the mode with busy indicator, it is necessary that at the moment of the conversion end, the CNV line is set to zero (Figure 3).

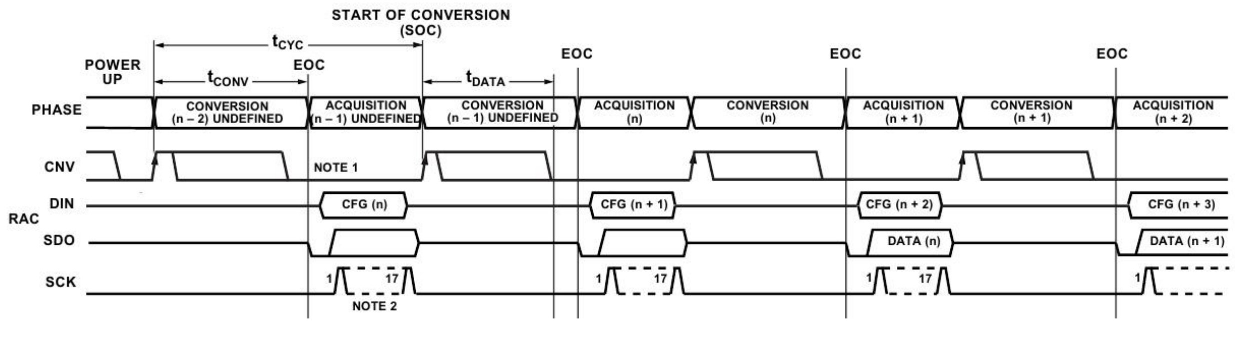
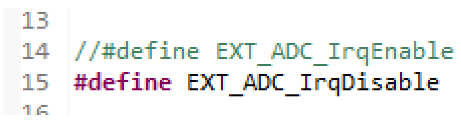


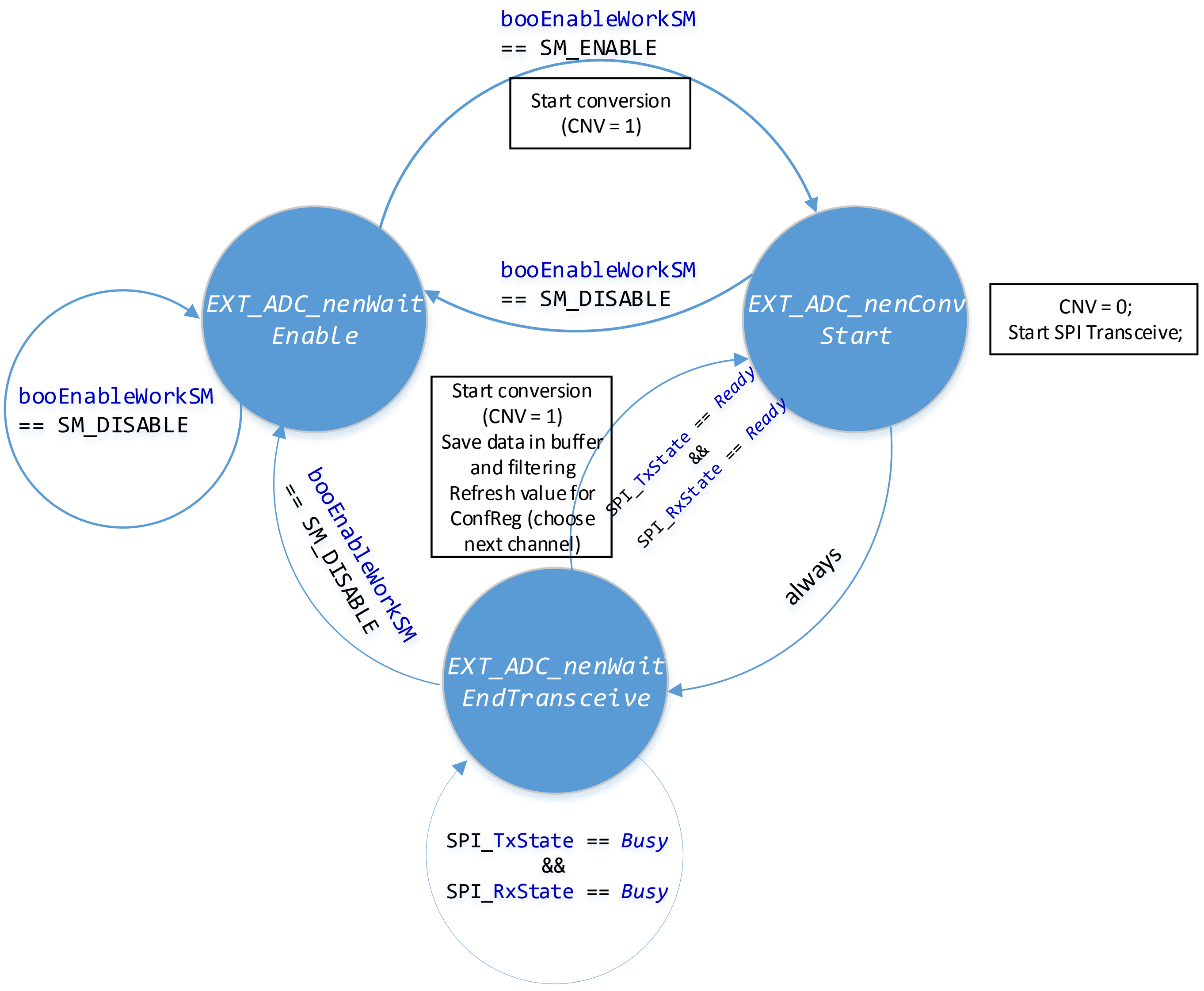
Figure 3 – Timing diagrams with a busy indicator

The **EXT_ADC_vMain()** function is an state machine and it is used to control the transfer algorithm. In the file "EXT_ADC_ce.h" you need to select the transfer mode using the directive #define:



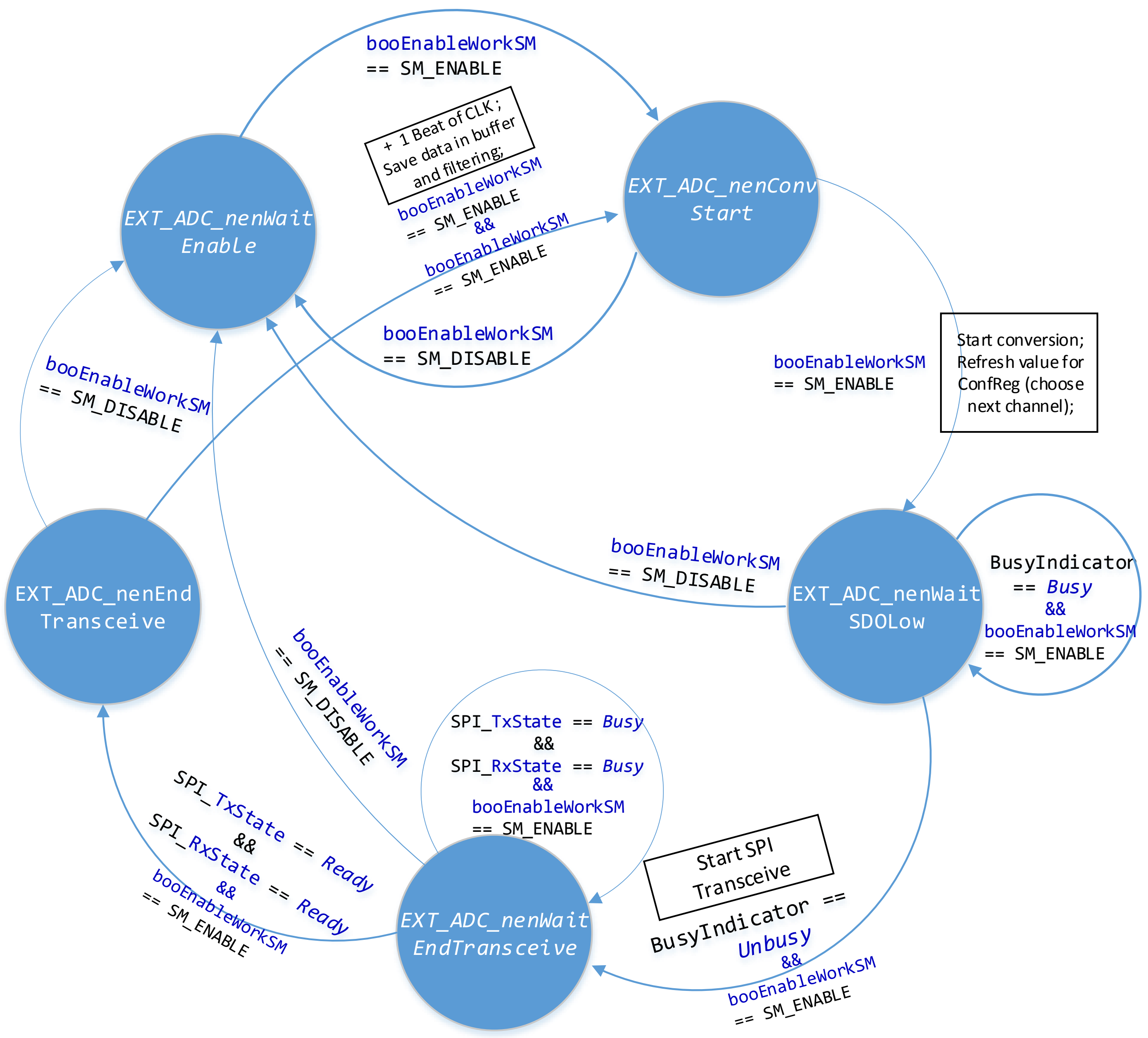
```
//#define EXT_ADC_IrqEnable
#define EXT_ADC_IrqDisable
```

Depending on which of these modes is selected using the #ifdef directive, there are two variants of the state machine **EXT_ADC_vMain():**



booEnableWorkSM == SM_ENABLE
Start conversion (CNV = 1)
booEnableWorkSM == SM_DISABLE
EXT_ADC_nenWait Enable
EXT_ADC_nenConv Start
CNV = 0; Start SPI Transceive;
booEnableWorkSM == SM_DISABLE
Start conversion (CNV = 1) Save data in buffer and filtering Refresh value for ConfReg (choose next channel)
SPI_TxState == Ready && SPI_RxState == Ready
booEnableWorkSM == SM_DISABLE
always
EXT_ADC_nenWait EndTransceive
SPI_TxState == Busy && SPI_RxState == Busy

Figure 4 – State machine for transmission **without** busy indicator

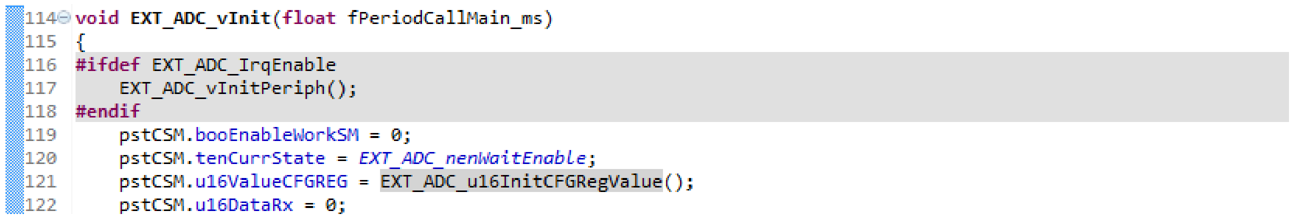


booEnableWorkSM == SM_ENABLE
+ 1 Beat of CLK; Save data in buffer and filtering;
booEnableWorkSM == SM_ENABLE && booEnableWorkSM == SM_ENABLE
EXT_ADC_nenWait Enable
EXT_ADC_nenConv Start
booEnableWorkSM == SM_DISABLE
Start conversion; Refresh value for ConfReg (choose next channel);
booEnableWorkSM == SM_ENABLE
booEnableWorkSM == SM_DISABLE
EXT_ADC_nenEnd Transceive
booEnableWorkSM == SM_DISABLE
EXT_ADC_nenWait SDOLow
BusyIndicator == Busy && booEnableWorkSM == SM_ENABLE
booEnableWorkSM == SM_DISABLE
SPI_TxState == Busy && SPI_RxState == Busy && booEnableWorkSM == SM_ENABLE
SPI_TxState == Ready && SPI_RxState == Ready && booEnableWorkSM == SM_ENABLE
Start SPI Transceive
BusyIndicator == Unbusy && booEnableWorkSM == SM_ENABLE
EXT_ADC_nenWait EndTransceive

Figure 5 – State machine for transmission **with** busy indicator

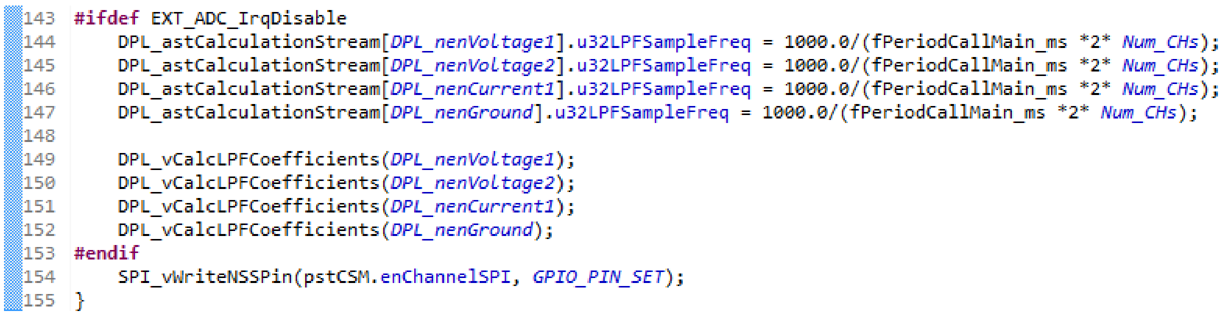
The mode without busy indicator is simpler because there is no need to form an additional clock pulse. Entry into the state machine determines the one of the ADC input channel measurement frequency. The low-pass filter coefficients from the data_processing_lib depend on this frequency; therefore, the initialization function takes as an argument the period of entering the state machine in milliseconds for to calculate the low-pass filter coefficients.

Automatic recalculation is not provided for the mode with the busy indicator, therefore, in this case, you must set the correct sampling rate (LPFilterSampleFreq) manually in the file “EXT_ADC_ce.h”.



```
void EXT_ADC_vInit(float fPeriodCallMain_ms)
{
#ifdef EXT_ADC_IrqEnable
    EXT_ADC_vInitPeriph();
#endif
    pstCSM.booEnableWorkSM = 0;
    pstCSM.tenCurrState = EXT_ADC_nenWaitEnable;
    pstCSM.u16ValueCFGREG = EXT_ADC_u16InitCFGRegValue();
    pstCSM.u16DataRx = 0;
```

…



```
#ifdef EXT_ADC_IrqDisable
    DPL_astCalculationStream[DPL_nenVoltage1].u32LPFSampleFreq = 1000.0/(fPeriodCallMain_ms *2* Num_CHs);
    DPL_astCalculationStream[DPL_nenVoltage2].u32LPFSampleFreq = 1000.0/(fPeriodCallMain_ms *2* Num_CHs);
    DPL_astCalculationStream[DPL_nenCurrent1].u32LPFSampleFreq = 1000.0/(fPeriodCallMain_ms *2* Num_CHs);
    DPL_astCalculationStream[DPL_nenGround].u32LPFSampleFreq = 1000.0/(fPeriodCallMain_ms *2* Num_CHs);

    DPL_vCalcLPFCoefficients(DPL_nenVoltage1);
    DPL_vCalcLPFCoefficients(DPL_nenVoltage2);
    DPL_vCalcLPFCoefficients(DPL_nenCurrent1);
    DPL_vCalcLPFCoefficients(DPL_nenGround);
#endif
    SPI_vWriteNSSPin(pstCSM.enChannelSPI, GPIO_PIN_SET);
}
```

The mode with busy indicator makes sense to use for fast transmission between the microcontroller and the ADC, since in this case the state machine triggered on the interrupt after the end of the transmission via SPI and on the readiness indicator. In this case, the **EXT_ADC_vMain()** function can be called without any delay, in the infinite loop (while (1)) of the **main()** functions. The maximum transfer rate in this case is limited only by the time parameters of the ADC.

This project uses a mode without a busy indicator because the OS has tasks with a minimum period of 1 ms. The maximum transfer rate in this case is limited by the period of entry into the state machine.

The state machine is written in such a way that for each interrupt after the end of the transmission via SPI, data is saved to buffers and filtered. The state machine links data exchange with ADC and data processing using **circular_buffers_lib** and **data_processing_lib**.

These two libraries are written so that they can be easily linked with each other, but at the same time, this does not prevent them from acting as libraries completely independent of each other. The following figure shows in more detail the connection between libraries:

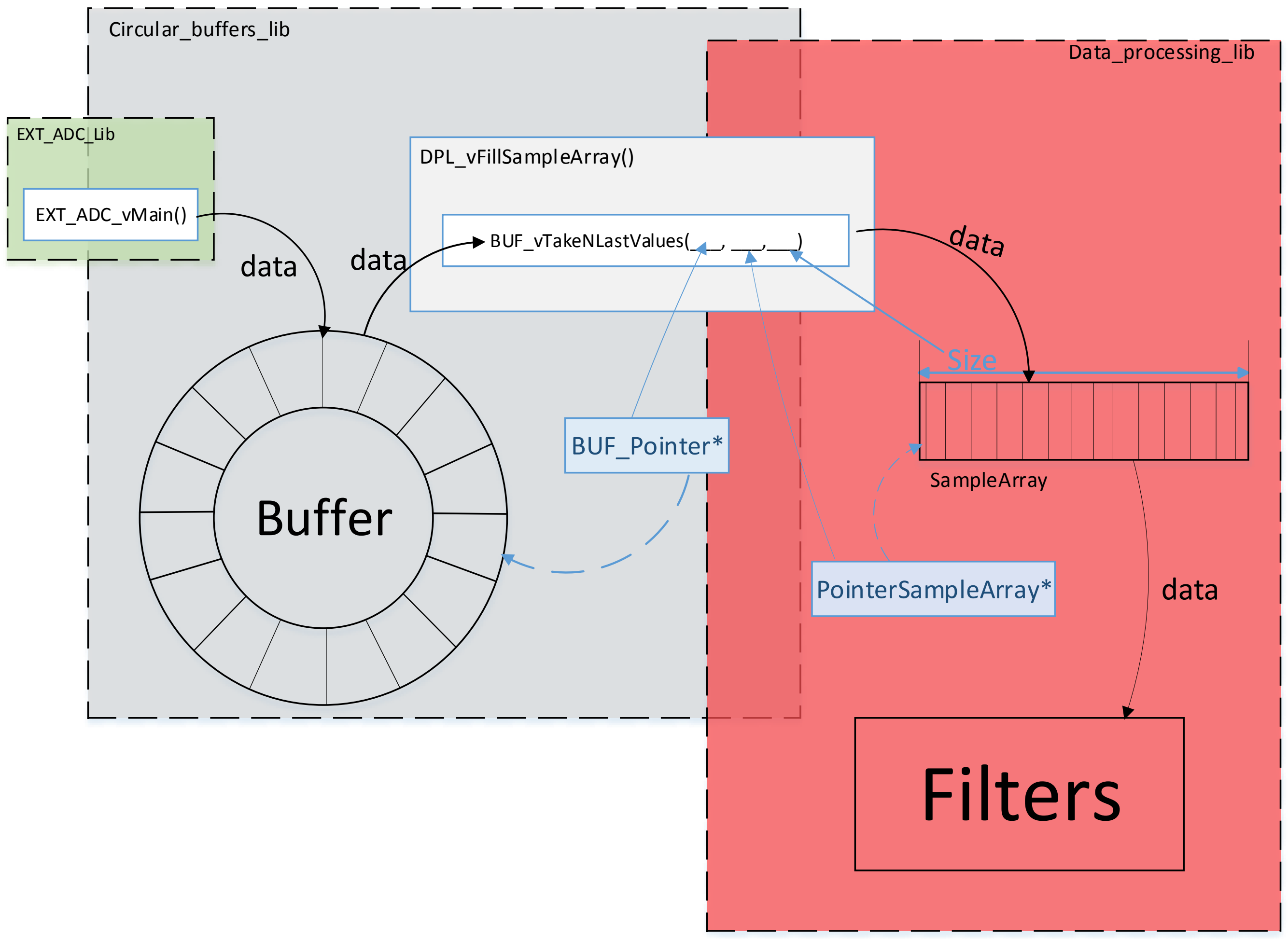


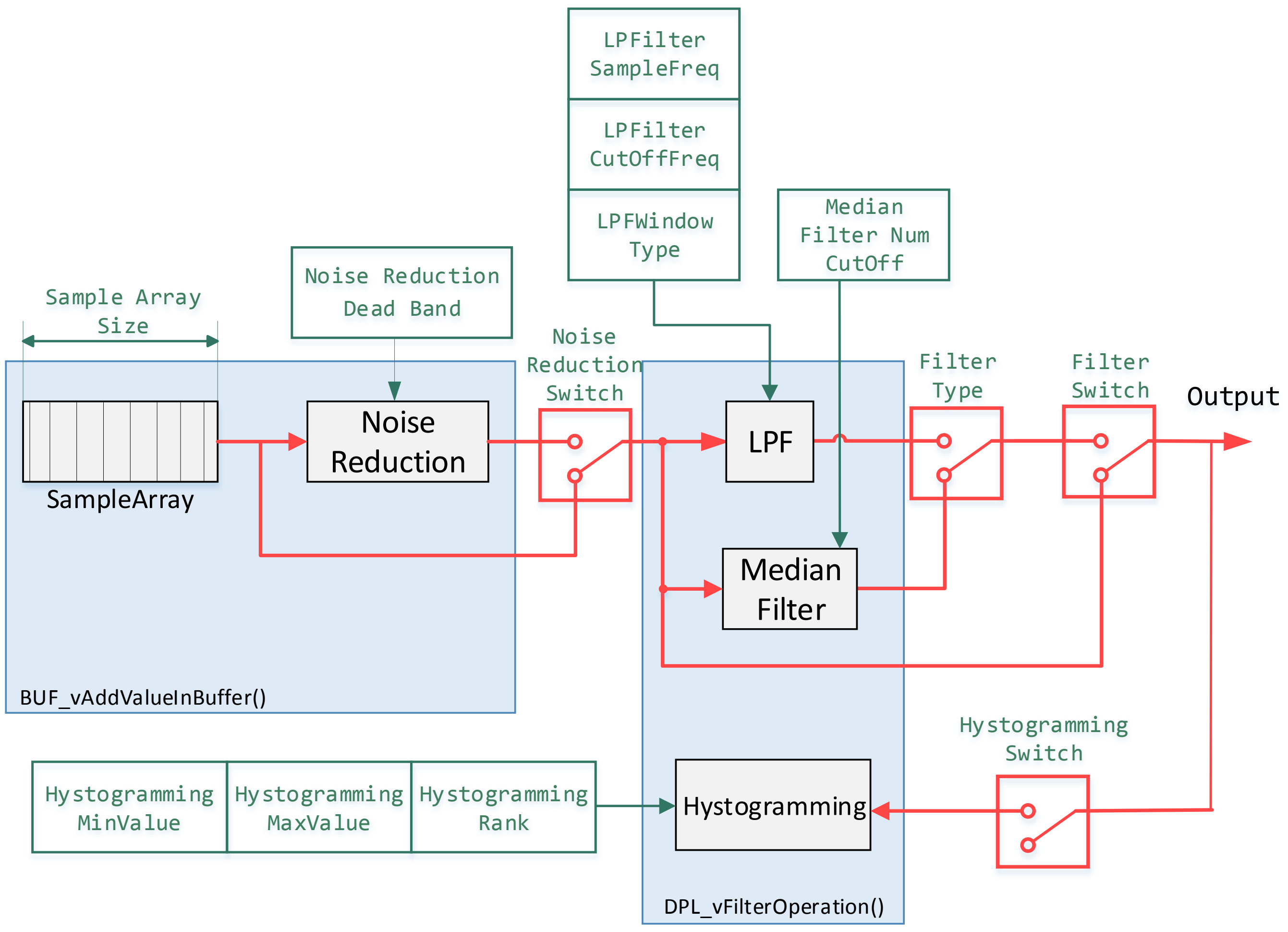
Figure 6 – Relationship between libraries

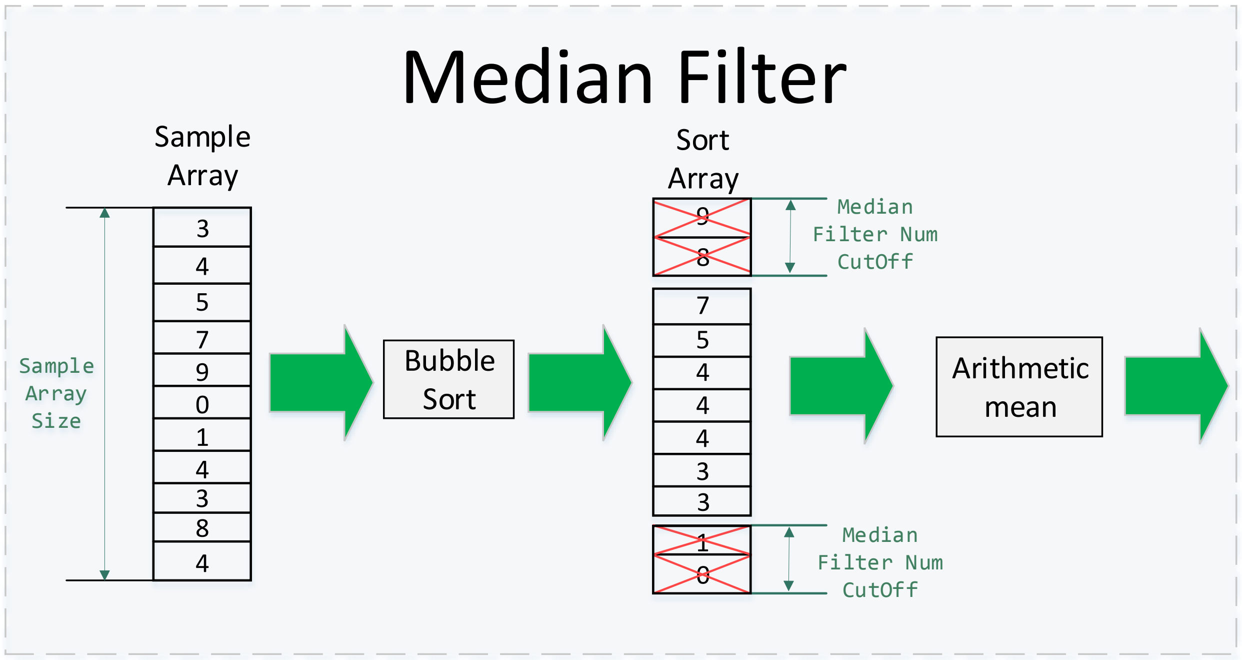
The libraries **circular_buffers_lib** and **data_processing_lib** are linked using the function **DPL_vFillSampleArray()**. This function belongs to the library **data_processing_lib**, but inside it is a call **BUF_vTakeNLastValues ()** function, which is owned by **circular_buffers_lib**. The function **BUF_vTakeNLastValues (BUF_tenBuffers enBuf, uint8_t u8NumLastValues, float * pfRecipientAddr)** takes as the arguments the number of the buffer, the number of elements that need to be taken and the pointer to the array. Using this data, the function simply copies the last values added to the buffer to the array, then this data array is used for the data filtering process.

In the **DPL_vFillSampleArray()** function, the NoiseReduction also occurs after copying data to the sample array. Subsequent filtering is performed by the **DPL_vFilterOperation()** function.

How the low pass filter is arranged can be found in this article: http://www.labbookpages.co.uk/audio/firWindowing.html

The way the setup of the filtering process is arranged is shown in the following figure:


LPFilter SampleFreq
LPFilter CutOffFreq
LPFWindow Type
Median Filter Num CutOff
Noise Reduction Dead Band
Sample Array Size
Noise Reduction Switch
Filter Type
Filter Switch
Output
SampleArray
Noise Reduction
LPF
Median Filter
BUF_vAddValueInBuffer()
Hystogramming Switch
Hystogramming MinValue
Hystogramming MaxValue
Hystogramming Rank
Hystogramming
DPL_vFilterOperation()


Median Filter
Sample Array
Sample Array Size
3
4
5
7
9
0
1
4
3
8
4
Bubble Sort
Sort Array
9
8
7
5
4
4
4
3
3
1
0
Median Filter Num CutOff
Median Filter Num CutOff
Arithmetic mean

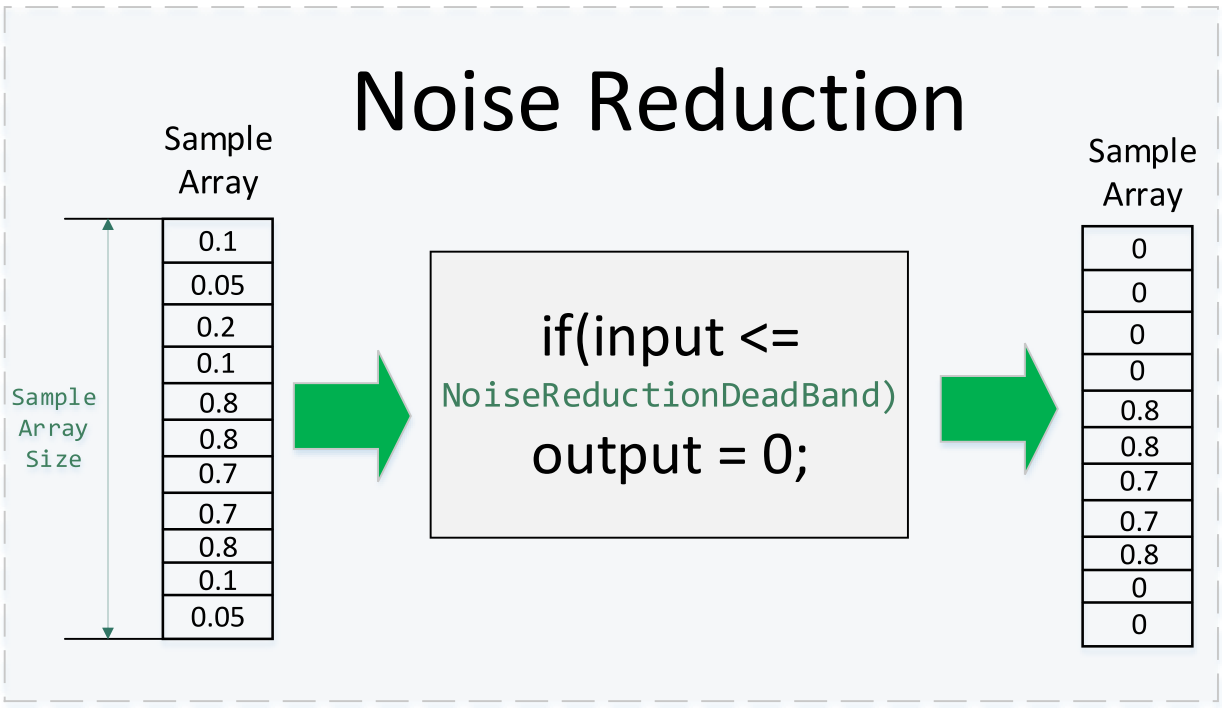


Figure 7 – Data processing library organization

Setup in the file "EXT_ADC_ce.h"

1) SPI is configured separately in the file "SPI1_ce.h", and in order to associate the desired SPI with the EXT_ADC library, you need to specify its name, port and CLK pin in the file "EXT_ADC_ce.h". The last thing is necessary for the formation of the 17th clocking pulse.

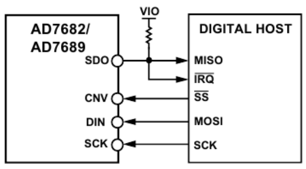
```
#define EXT_ADC_ConnectSPI   \
/*                 SPI_ChannelId,  CLK_Port,    CLK_Pin */ \
EXT_ADC_MakeChannel ( SPI1,          GPIOA,       GPIO_PIN_5)

```

2) The following macro concerns the setting of the Circular buffers library. Buffer creation is configured here. You need to specify the type, name, size, and coefficients for the offset and multiplication. Offset and multiplication occurs before adding a new element to the buffer (function **BUF_vAddValueInBuffer**()), according to the following formula:

Output = (Input + OffsetBeforeMultiply)*Scale + OffsetAfterMultiply

```
#define BUF_MakeConfig   \
/*                      Type,   Name,      Size,   OffsetBeforeMultiply,   Scale,     OffsetAfterMultiply*/ \
BUF_MakeCircularBuffer (float,  Voltage1,  20,     0,                      6.67,      0 )  \
BUF_MakeCircularBuffer (float,  Voltage2,  20,     0,                      6.67,      0 )  \
BUF_MakeCircularBuffer (float,  Current1,  20,     -0.46,                  2.5,       0 )  \
BUF_MakeCircularBuffer (float,  Ground,    20,     0,                      1,         0 )  \

```

3) The following macro sets the initial values for the fields of the Configuration Register register, defines the initial value of the reference voltage for the ADC and the port and the number of the pin that monitors the busy indicator.



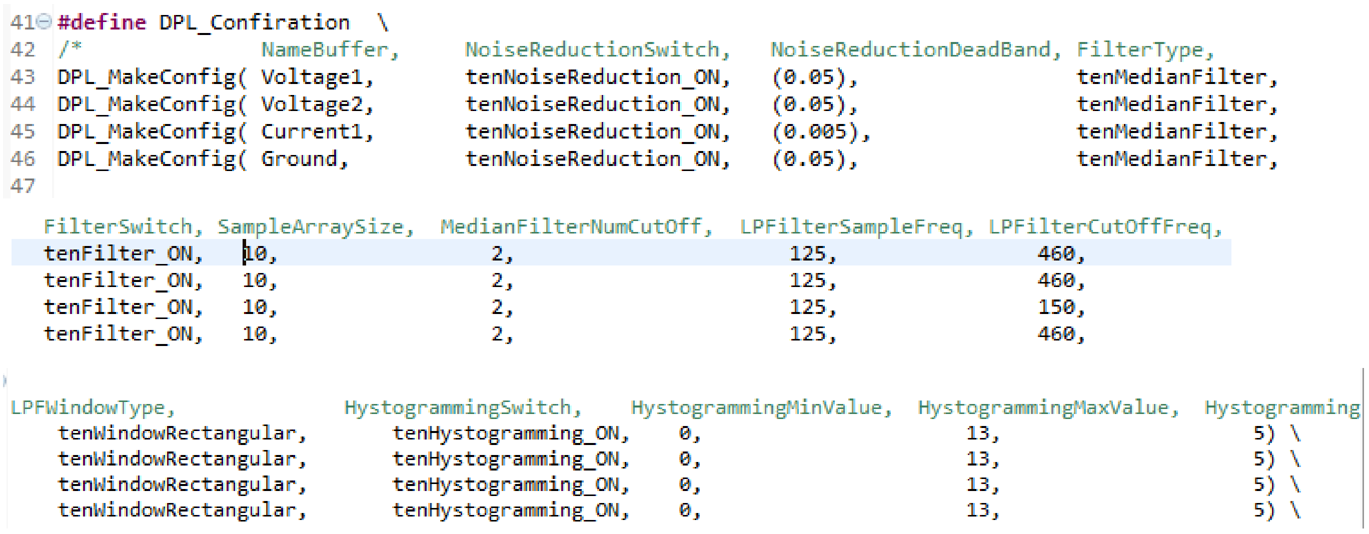
```
#define EXT_ADC_ConfigADC   \
/*                      RefreshCFG,              InputCHConf,                 InputCHSelection,
EXT_ADC_MakeConfig (    EXT_ADC_OVERWRITE_REG,   EXT_ADC_UNIPOLAR_REF_GND,    EXT_ADC_CH0,

```

```
 Bandwidth,             Reference,                         ExternRefVoltage,   Sequencer,
 EXT_ADC_FULL_BANDWIDTH, EXT_ADC_EXTREF__TEMP_DIS__INTBUFF_DIS,  5,              EXT_ADC_SEQUENCER_DIS,
```

```
 Readback,              IrqPort,     IrqPin*/ \
 EXT_ADC_READBACK_OFF,  GPIOC,       GPIO_PIN_4)
```

4) Configure the Data processing library. In order for the Data processing library and Circular buffers library libraries to work together, it is important to specify the buffer name correctly here.

The way the filtering process setup works is shown in Figure 7.



```
#define DPL_Confiration  \
/*                NameBuffer,      NoiseReductionSwitch,    NoiseReductionDeadBand, FilterType,
DPL_MakeConfig( Voltage1,       tenNoiseReduction_ON,    (0.05),                 tenMedianFilter,
DPL_MakeConfig( Voltage2,       tenNoiseReduction_ON,    (0.05),                 tenMedianFilter,
DPL_MakeConfig( Current1,       tenNoiseReduction_ON,    (0.005),                tenMedianFilter,
DPL_MakeConfig( Ground,         tenNoiseReduction_ON,    (0.05),                 tenMedianFilter,

  FilterSwitch, SampleArraySize,  MedianFilterNumCutOff,  LPFilterSampleFreq, LPFilterCutOffFreq,
  tenFilter_ON,   10,                  2,                      125,                460,
  tenFilter_ON,   10,                  2,                      125,                460,
  tenFilter_ON,   10,                  2,                      125,                150,
  tenFilter_ON,   10,                  2,                      125,                460,

LPFWindowType,            HystogrammingSwitch,     HystogrammingMinValue,  HystogrammingMaxValue,  Hystogramming
    tenWindowRectangular,     tenHystogramming_ON,     0,                     13,                     5) \
    tenWindowRectangular,     tenHystogramming_ON,     0,                     13,                     5) \
    tenWindowRectangular,     tenHystogramming_ON,     0,                     13,                     5) \
    tenWindowRectangular,     tenHystogramming_ON,     0,                     13,                     5) \
```

How to use

1) Make all settings in the file "EXT_ADC_ce.h";

2) Call the EXT_ADC_vInit(float fPeriod_ms) function and pass to it the period for calling the state machine EXT_ADC_vMain () in milliseconds;

3) Periodically call the EXT_ADC_vMain() function;

4) For to get output values use functions EXT_ADC_fGetVCC1(), EXT_ADC_fGetVCC2(), EXT_ADC_fGetCurrent(), EXT_ADC_fGetGND().

IMPORTANT: The library is not designed for data exchange with more than one ADC AD7682/7689!

Example

```
int main(void)
{
	HAL_Init();

	SystemClock_Config();

	__HAL_RCC_TIM10_CLK_ENABLE();
	__HAL_RCC_TIM3_CLK_ENABLE();

	timer_init(TIM10,TIM1_UP_TIM10_IRQn, 5000 , US);
	timer_init(TIM3,TIM3_IRQn, 1000 , US);

	init_button_context_struct(&user_button);

	SPI_vInit();
	EXT_ADC_vInit(1);
	EXT_ADC_vEnableWorkSM();


	while(1)
	{
		SPI_vMain();

		if(check_vMain == 1)
		{
			check_vMain = 0;

			EXT_ADC_vMain();
		}
```

```
				if(user_button.flag_check_button == 1)
				{
					user_button.flag_check_button = 0;
					user_button.flag_button = update_pin_state(&user_button);
				}

				if(user_button.flag_button == 1)
				{
					user_button.flag_button = 0;

					printf("vcc1 = %f\r\n", EXT_ADC_fGetVCC1());
					printf("vcc2 = %f\r\n", EXT_ADC_fGetVCC2());
					printf("current = %f\r\n", EXT_ADC_fGetCurrent());
					printf("gnd = %f\r\n", EXT_ADC_fGetGND());
				}
		}
		}

void TIM3_IRQHandler(void)
{
	TIM_HandleTypeDef htim;

	htim.Instance = TIM3;

	 if(__HAL_TIM_GET_FLAG(&htim, TIM_FLAG_UPDATE) != RESET)
		{
		  if(__HAL_TIM_GET_IT_SOURCE(&htim, TIM_IT_UPDATE) !=RESET)
		   {
		    __HAL_TIM_CLEAR_IT(&htim, TIM_IT_UPDATE);

		      check_vMain = 1;
		   }
		}
}

void TIM1_UP_TIM10_IRQHandler(void)
{
	TIM_HandleTypeDef htim;

	htim.Instance = TIM10;

	 if(__HAL_TIM_GET_FLAG(&htim, TIM_FLAG_UPDATE) != RESET)
	  {
	    if(__HAL_TIM_GET_IT_SOURCE(&htim, TIM_IT_UPDATE) !=RESET)
	    {
	      __HAL_TIM_CLEAR_IT(&htim, TIM_IT_UPDATE);

	        user_button.flag_check_button = 1;
	    }
	  }

}
```

Filter results

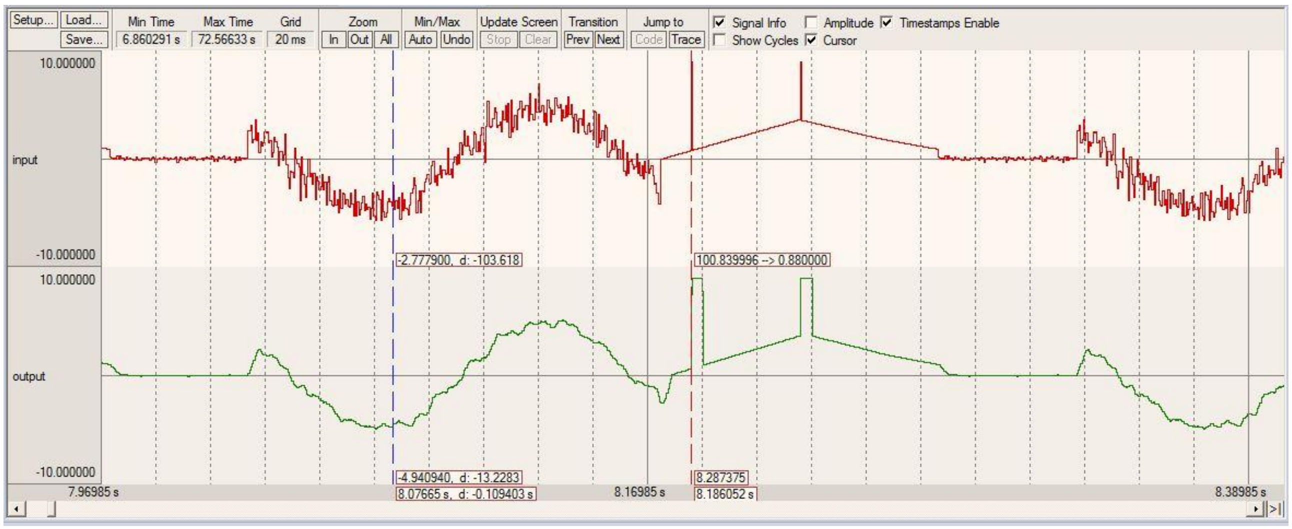


Figure 8 – LPF operation example

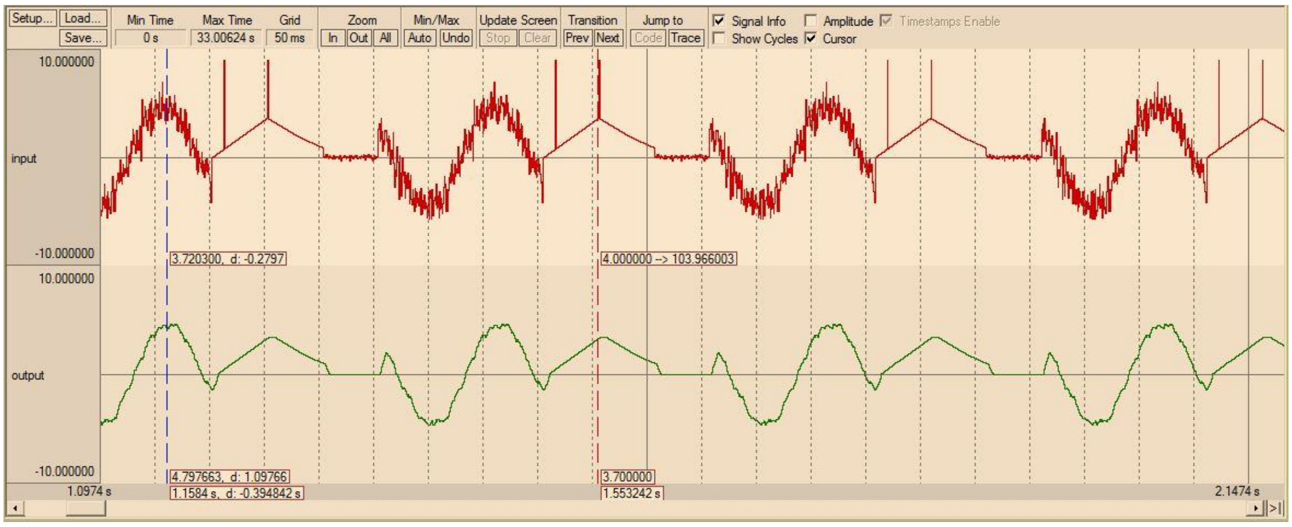


Figure 9 – Median filter operation example