AN12282

Digital Signal Processing for NXP LPC5500 Using PowerQuad

Rev. 0 — 24 January 2019 Application Note

Contents

1 PowerQuad introduction

Mobile IoT and Context® awareness are growing tremendously and more local digital signal processing is required. Low power always-on systems are good options for Cortex M-based MCUs (for leakage reduction and overall low power considering limited computation).

Arm® Cortex-M architecture gears towards energy efficient control applications. Signal processing lags behind traditional DSP architectures, sometimes as much as 10x-20x in terms of performance due to the following factors:

- Narrow memory width (single 32-bit data bus) – DSPs typically have at least two data buses as well as local memory blocks.
- Limited simultaneous computational capability (for example, one multiplication + add per cycle).
- Not enough registers for intermediate keeping of necessary data.
- No dedicated built-in accelerators for functions such as FFT (large load of additions/subtractions), Biquad Filters.

Although Arm does not bring large scale DSP improvements to Cortex-M family of cores, it has standardized the DSP library (CMSIS DSP Lib). When users are using a common standard interface for DSP functions, there is an opportunity to provide a vendor supplied optimizations. User's code still uses CMSIS DSP, but NXP can 'improve the recipe under the hood'. A further key point to note is that accelerating computations cuts power not only by MCU being able to go to sleep earlier, but furthermore, through capability to run slower at a lower frequency, thus lower voltage (lowering energy further still). Then the PowerQuad comes.

Here are some typical mathematical requirements in DSP applications:

- Motion context
  - Matrix operations, Rotation via trigonometric functions, FFT, Filter (FIR/IIR) for calibration.
  - Convolution and correlation for motion feature extraction and matching.
- Voice recognition
  - FFT for spectral analysis, Logarithm and Mel-Frequency and other windowing (Matrix multiplication), Filter (FIR/IIR), DCT for Cepstrum extraction.
  - Statistical modeling for feature extraction and comparison.
- Neural networks architecture specific features
  - Matrix MAC
  - Logistic/Sigmoid function (using exponentiation) for perceptron evaluation (also very useful for statistical distribution analysis.
- Biometrics
  - FFT for Heartbeat monitoring, Arctan/other trig for Fingerprinting.

Now, the PowerQuad can support most of these mathematical requirements on the hardware, which accumulates the process and saves CPU time for other thread simultaneously.

2 PowerQuad hardware

2.1 PowerQuad computing features

As a hardware module integrated inside the chip, PowerQuad executes the calculation task all on the hardware. It involves various computing engines:

- Transform engine
- Transcendental function engine
- Trigonometry function engine
- Dual biquad IIR filter engine
- Matrix accelerator engine
- FIR filter engine
- CORDIC engine

Table 1. PowerQuad hardware function on page 2 lists the computing features that PowerQuad supports directly.

Table 1. PowerQuad hardware function

Class	Function	Comments
Math	1/x, ln(x), sqrt(x), 1/sqrt(x), e^(x), e^(-x), (x1) / (x2), sin(x), cos(x)	coprocessor instruction
	arctan(x), arctanh(x)	
Filter	• 2nd order IIR filter	coprocessor instruction
	• FIR filter • FIR filter incremental • Correlation • Convolution	
Matrix	• Scale • Addition • Subtraction • Invert • Profuct • Hadamard product (elementwise product) • Transpose • Dot product	-

Table continues on the next page...

Table 1. PowerQuad hardware function (continued)

Transform	• Complex FFT (complex-valued input sequence) • Real FFT (real-valued input sequence) • Inverse FFT • Complex DCT (complex-valued input sequence) • Real DCT (real-valued input sequence) • Inverse DCT	-

These functions form the foundation for the implementation of advanced algorithm.

2.2 PowerQuad bus interfaces

PowerQuad is integrated with the Arm Cortex-M33 co-processor Interface, so it can be accessed through the co-processor instructions (**MCR** and **MRC**). Also, there are programmable registers designed inside the PowerQuad to connect the AHB bus. That means user code running on the Cortex-M33 core can read and write its register as well like other normal programmable modules. See Figure 1. on page 3.

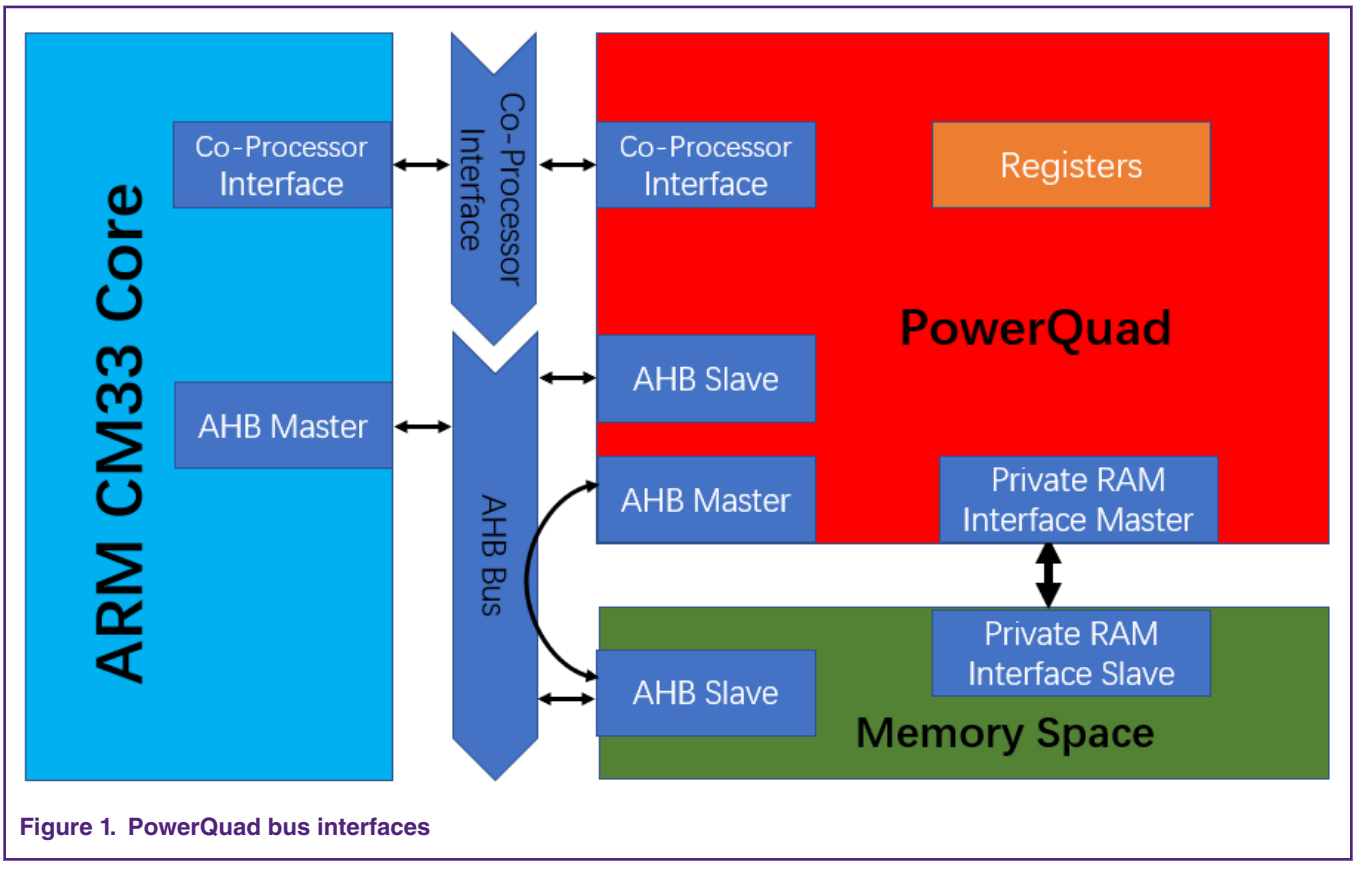


Figure 1. PowerQuad bus interfaces

However, specific access ways are for the specific usage. Generally, for PowerQuad, Arm Cortex-M co-processor interface and AHB slave interface are used to deliver the commands/configurations, while the AHB master interface and the private RAM master interface are used to operate the memory.

- Co-processor functions

When doing the calculation which accepts one number as input parameter and return one number as output result, they would mostly use the Cortex-M Co-processor Interface to pass in the input parameter and return the result. For example, the most math functions are implemented in this way. These functions are simple and running very soon.

- Streaming/DMA functions

  When doing the calculation that works on an array of data and the result is another array of data, the PowerQuad uses a DMA-like way to handle the input and output data. Examples of AHB access functions are the transform functions, matrix functions, and most filter functions. When using the PowerQuad for these functions, users need to set some base address registers of PowerQuad, like using DMA, then the PowerQuad hardware uses the memory indicated by these addresses automatically when the calculation is launched.

NXP MCUXpresso SDK already provides the driver for PowerQuad. It packs the operations with co-processor interface (co-operator instruments) and AHB bus (functional registers). So, if the users develop their applications with the SDK API, they do not need to care how to select the instructions or register settings.

2.3 PowerQuad memory handlers

When considered as an embedded mathematic computer, the PowerQuad needs a lot of data to be processed and produced.

Along with the powerful computing engines, there are four groups for memory handler, which indicate the four memory areas to support the data management requirement of PowerQuad functions.

- Input A. pointer to the input data array 1.
- Input B. pointer to the input data array 2 when necessary. For example, when making the matrix addition, the other matrix will be indicated by Input B handler.
- Temp. pointer to the temporary memory that keeps the intermediate computational results when necessary (for FFT and Matrix Inversion). The memory should be initialized before the current calculation and can be cleared later. PowerQuad writes values and reads them automatically during the calculation.

Each of the four memory areas can be configured for the customized format:

- Format of originating data (32-bit fixed, 16-bit fixed or 32-bit float)
- Format of data desired for PowerQuad (float for all except FFT, which is a fixed-point engine)
- Scale of result (PowerQuad can do scaling by power of 2 on the way in its out.)

Users can fill the address of prepared memory into the responding registers in the PowerQuad module. See Table 2. PowerQuad registers for memory handlers on page 4.

Table 2. PowerQuad registers for memory handlers

Address	Name	Description	Access	Reset value
0x000	OUTBASE	Base address register for output region	RW	0
0x004	OUTFORMAT	Data format for output region	RW	0
0x008	TMPBASE	Base address register for temp region	RW	0
0x00C	TMPFORMAT	Data format for region Temp	RW	0

Table continues on the next page...

Table 2. PowerQuad registers for memory handlers (continued)

0x010	INABASE	Base address register for input A region	RW	0
0x014	INAFORMAT	Data format for region input A	RW	0
0x018	INBBASE	Base address register for input B region	RW	0
0x01C	INBFORMAT	Data format for region input B	RW	0

PowerQuad can handle the general RAM memory (shared with other AHB masters, like Cortex-M core) and private RAM memory (start from 0xE000_0000, 16 KB). Specially, for private RAM memory, as it is reserved only for PowerQuad, PowerQuad can access it without any arbitration delay, saving a lot of time for PowerQuad to get data. Then, PowerQuad can access the private RAM four banks of memory in parallel, giving 128-bit wide. So, it performs some functions even much faster, like FFT, FIR, convolution, matrix etc.

Some notes for using the private RAM:

- FFT engine may only use the private memory as temp memory (not as input or output).
- All data in private memory must be floating point. (You can get data in and out of private memory by using the matrix scale operation with private memory being destination).
- The private memory does not provide any scaling. Scaling is only available for data which is being read/written to the system memory.

3 PowerQuad DSP examples

This section describes the basic usage of PowerQaud in application. During the explanation of demo case, the description for the PowerQuad APIs will be mentioned.

The demo runs on the LPCXpresso5500 (OM40011) board with an LCD screen module to show the GUI. In the demo project, a simple framework is designed to switch the separate task as a scheduler. Then the various simple tasks can be executed one by one, for FFT, matrix, and FIR. With the LCD screen module, the display function is also integrated into the framework.

The PowerQuad FFT, matrix, and the FIR filter are chosen in this demo, as these calculations are popular in most DSP application but usually cost a lot of time when implemented by pure software (Arm CMSIS-DSP Lib). In the end of the section, a comparison of performance for PowerQuad APIs and Arm CMSIS-DSP API is provided.

Note that the detail thing about the calculation process would not be discussed in this paper. For further information, refer to PowerQuad UM and SDK driver code.

A detailed illustration about using PowerQuad APIs is described for FFT cases. The same idea is applied to other cases.

3.1 Task schedule with display GUI

To involve the separate cases into one project, a scheduler is implemented in the demo project. Each case is implemented within a function as the task entry. All the task entries are collected into the task array cAppLcdDisplayPageFunc[]. Also, a hardware thread to capture the button is launched.

Then, the MCU will be in the sleep mode until waken up by the key interruption. The key value is changed in the ISR of key interruption. The main loop will check the change of key value and switch to the task with the index (using the key value) in the task list.

```
    /* List of lcd display with tasks.  */
    void (*cAppLcdDisplayPageFunc[])(void) =
```

```
    {
        task_pq_fft_128,
        task_pq_fft_256,
        task_pq_fft_512,
        task_pq_mat_add,
        task_pq_mat_inv,
        task_pq_mat_mul,
        task_pq_fir_lowpass,
        task_pq_fir_highpass,
        task_pq_records
    };

    int main(void)
    {
        ...
         while (1)
        {
            keyValue = App_GetUserKeyValue(); /* keyvalue is used as the index of task. */
            if (keyValue != keyValuePre) /* only switch task when keyvalue is changed. */
            {
                App_DeinitUserKey(); /* disable detecting key when changing lcd display. */
                (*cAppLcdDisplayPageFunc[keyValue])(); /* switch to new page with new task. */
                keyValuePre = keyValue;
                App_InitUserKey(); /* enable detecting key for next event. */
            }
            __WFI(); /* sleep when in idle. would wake up when the key interrupt happens caused by
the touch screen. */
        }
    }
```

In each task, it executes the PowerQuad computing to finish a simple task and measure the time for critical operations. Then it show the record to the LCD screen module.

3.2 Functions of measuring time

Considering that the functions are usually running fast, interrupt-based timing method is not suitable in the demo case. However, in some test projects specially for measuring, interrupt-based timing method is still available by measuring plenty times of the target function, then to get the average time for one execution.

In this demo, SysTick timer is chosen as the timer, so that the code here could be well portable for the other Arm Cortex-M MCU. Then use the 24-bit counter value directly for timing. For the LPC5500, which is running at 98 MHz for the SysTick timer's clock source, the max timing period could be 171 ms.

```
    /* Systick Start */
    #define TimerCount_Start()  do {                              \
        SysTick->LOAD  =  0xFFFFFF  ;   /* Set reload register */\
        SysTick->VAL  =  0  ;           /* Clear Counter */      \
        SysTick->CTRL  =  0x5 ;         /* Enable Counting*/     \
    } while(0)

    /* Systick Stop and retrieve CPU Clocks count */
    #define TimerCount_Stop(Value)  do {                           \
        SysTick->CTRL  =0;  /* Disable Counting */                 \
        Value = SysTick->VAL;/* Load the SysTick Counter Value */ \
        Value = 0xFFFFFF - Value;/* Capture Counts in CPU Cycles*/\
    } while(0)
```

The usage is:

```
uint32_t calcTime;

TimerCount_Start();
arm_cfft_q31(&instance, gPQFftQ31InOut, 0, 1); /* Calculation. */
TimerCount_Stop(calcTime);

printf("calcTime: %d", calcTime);
```

3.3 FFT demo cases

There are three FFT cases in the demo: 128 points, 256 points, and 512 points.

Tips for using PowerQuad FFT engine are:

- PowerQuad can support 16/32/64/128/256/512 points for FFT computing engine on the hardware.
- The PowerQuad FFT engine always scales the input data by 1/N when computing the FFT (and by extension DCT). If an unscaled result is necessary, the input data (in the INPUT A region) must first be multiplied by N manually. The inverse FFT is scaled by 1/N, but this is correct as per the iDFT formula, so no scaling treatment is needed.
- The FFT engine only looks at the bottom 27 bits of the input word, so no pre-scaling can exceed to avoid the saturation.
- The purely real (prefixed by ‘r’ in API name), and the complex flavors of the functions (prefixed by 'c' in API name) expect the input data sequences to be arranged in memory as follows.
- If the sequence x = x0, x1 ... xN-1 are real numbers, then the input array in memory must be organized as x[N] = {x0, x1, ... xN-1}.
- If the sequence x = x0, x1 ... xN-1 are complex numbers of the form of (x0_real + i*x0_im), (x1_real + i*x1_im), ... (xN-1_real + i*xN-1_im), then the input array in memory must be organized as x[N] = {x0_real, x0_im, x1_real, x1_im, ... xN-1_real, xN-1_im}.
- The output sequence is always stored in the memory organized as an array of complex numbers where the imaginary parts will be zero for real-valued output data.

When running the PowerQuad Transform engine (include the FFT), only the INPUT A memory handler is used for input, and the OUT memory handler is used for output. For the full information about the usage of memory handler for Transform engine, refer to Table 3. Usage of memory handlers for FFT engine on page 7.

Table 3. Usage of memory handlers for FFT engine

Operation	Driver function	Access type	Input/ Output data formats	Input A region usage	Input B region	Output region usage	Temp. region usage	Fixed point input/ output scalers	Engine	Uses GPREGs / COMPREGs?
Complex FFT	Pq_cfft	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes

Table continues on the next page...

Table 3. Usage of memory handlers for FFT engine (continued)

Real FFT	Pq_rfft	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
Inverse FFT	Pq_ifft	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
Complex DCT	Pq_cdct	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
Real DCT	Pq_rdct	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
Inverse DCT	Pq_idct	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes

The PowerQuad APIs used in the demo is designed to be compatible as the CMSIS-DSP API. So, for the CMSIS-DSP users, they do not need to change the existing codes but can run faster with PowerQuad's implementation.

Taking FFT of 128 points as examples:

```
extern q31_t     gPQFftQ31In[APP_PQ_FFT_SAMPLE_COUNT_MAX*2u];
extern q31_t     gPQFftQ31Out[APP_PQ_FFT_SAMPLE_COUNT_MAX*2u];
extern q31_t     gPQFftQ31InOut[APP_PQ_FFT_SAMPLE_COUNT_MAX*2u];
extern float32_t gPQFftF32In[APP_PQ_FFT_SAMPLE_COUNT_MAX*2u];
extern float32_t gPQFftF32Out[APP_PQ_FFT_SAMPLE_COUNT_MAX*2u];

void task_pq_fft_128(void)
{
    arm_cfft_instance_q31 instance;
    uint32_t i;
    uint32_t calcTime;

    /* Create the input signal. */
    for (i = 0; i < APP_PQ_FFT_SAMPLE_COUNT_128; i++)
    {
        /* real part. */
        gPQFftF32In[i*2] = 1.5f /* direct current. */
```

```
                            + 1.0f * arm_cos_f32( (       2.0f * PI / APP_PQ_FFT_PERIOD_BASE) *
i  ) /* low frequence */
                            + 0.5f * arm_cos_f32( (4.0f * 2.0f * PI / APP_PQ_FFT_PERIOD_BASE) *
i  ) /* high frequence */
                            ;
            gPQFftF32In[i*2] /= 3.0f; /* make sure the value in (0, 1) */

            /* imaginary part */
            gPQFftF32In[i*2+1] = 0.0f;

        }
    /* PowerQuad FFT can only operate fix-point number. */
        arm_float_to_q31(gPQFftF32In, gPQFftQ31In, APP_PQ_FFT_SAMPLE_COUNT_128*2u);
        for (i = 0u; i < APP_PQ_FFT_SAMPLE_COUNT_128 * 2u; i++)
        {
            gPQFftQ31InOut[i] = gPQFftQ31In[i] >> 5u; /* powerquad fft engine can only accept 27-bit
input data. */
        }

        instance.fftLen = APP_PQ_FFT_SAMPLE_COUNT_128;
        TimerCount_Start(); /* start timing. */
        arm_cfft_q31(&instance, gPQFftQ31InOut, 0, 1); /* computing. */
        TimerCount_Stop(calcTime);

        for (i = 0u; i < APP_PQ_FFT_SAMPLE_COUNT_128 * 2u; i++)
        {
            gPQFftQ31Out[i] = gPQFftQ31InOut[i] « 5u; /* restore the data from 27-bit to 32-bit. */
        }

        arm_q31_to_float(gPQFftQ31Out, gPQFftF32Out, APP_PQ_FFT_SAMPLE_COUNT_128*2u);
        arm_cmplx_mag_f32( gPQFftF32Out, gPQFftF32In, APP_PQ_FFT_SAMPLE_COUNT_128);

        /* Todo ...
         * - Record the time.
         * - Display the waveform.
         */
    }
```

arm_cfft_q31() calls the PowerQuad driver PQ_TransformCFFT() / PQ_TransformIFFT().

```
    void arm_cfft_q31(const arm_cfft_instance_q31 *S, q31_t *p1, uint8_t ifftFlag, uint8_t
bitReverseFlag)
    {
        assert(bitReverseFlag == 1);

        q31_t *pIn = p1;
        q31_t *pOut = p1;
        uint32_t length = S->fftLen;

        PQ_DECLARE_CONFIG;
        PQ_BACKUP_CONFIG;
        PQ_SET_FFT_Q31_CONFIG;

        if (ifftFlag == 1U)
        {
            PQ_TransformIFFT(POWERQUAD_NS, length, pIn, pOut);
        }
        else
        {
            PQ_TransformCFFT(POWERQUAD_NS, length, pIn, pOut);
```

```
    }

    PQ_WaitDone(POWERQUAD_NS);

    PQ_RESTORE_CONFIG;
}
```

Then the PQ_TransformCFFT() function configures the PowerQuad registers to setup the input/output and the length of memory, then launches the computing by enabling the PowerQuad as CFFT engine. After these operations, the PowerQuad can work.

```
void PQ_TransformCFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)
{
    assert(pData);
    assert(pResult);

    base->OUTBASE = (int32_t)pResult;
    base->INABASE = (int32_t)pData;
    base->LENGTH = length;
    base->CONTROL = (CP_FFT << 4) | PQ_TRANS_CFFT; /* Launch the computing task. */
}
```

When the computing is done, the INST_BUSY is asserted. Users can use the PQ_WaitDone() function to wait the PowerQuad done.

```
void PQ_WaitDone(POWERQUAD_Type *base)
{
    /* wait for the completion */
    while ((base->CONTROL & INST_BUSY) == INST_BUSY)
    {
        __WFE(); /* Enter to low power. */
    }
}
```

There are display pages on the LCD screen module for each of FFT demo cases when running the demo project, as shown in Figure 2. on page 10.

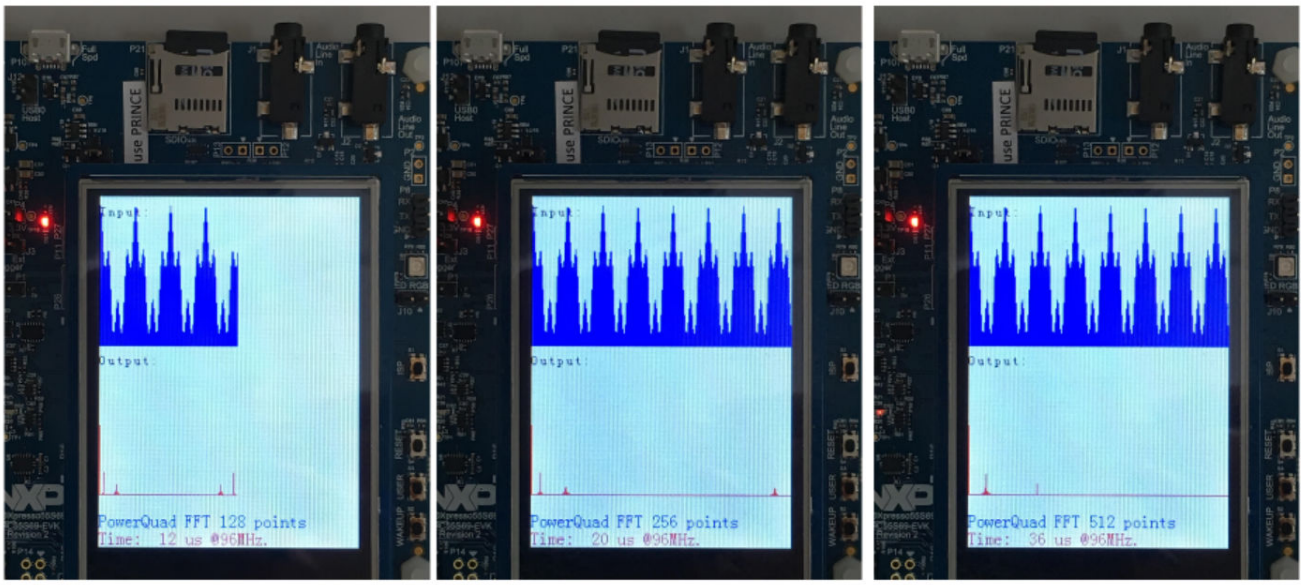


Figure 2. PowerQuad FFT 128/256/512 points

3.4 Matrix demo cases

The Matrix accelerator engine supports the eight operations, as listed in Table 4. PowerQuad matrix length range on page 11, given with their respective maximum supported dimensionalities.

Table 4. PowerQuad matrix length range

PowerQuad engine	Operation	Max. row
Matrix	Addition	16 × 16
	Subtraction	16 × 16
	Hadamard product	16 × 16
	Product	16 × 16
	Vector dot-product	256 elements
	Inversion	9 × 9
	Transpose	16 × 16
	Scaling	16 × 16

Matrix data are expected to be stored in memory row-by-row, arranged like standard C/C++ arrays. So, if two 2 × 2 integer matrices A and B are:

```
A = [1 2]        B = [5 6]
    [3 4]            [7 8]
```

Then the input data is expected to be stored in memory arrays as follows:

```
int MatA[4] = {1, 2, 3, 4};
int MatB[4] = {5, 6, 7, 8};
```

For the usage of memory handlers for PowerQuad Matrix engine, see Table 5. Usage of memory handlers for Matrix engine on page 11.

Table 5. Usage of memory handlers for Matrix engine

Operation	Driver function	Access type	Input/ Output data formats	Input A region usage	Input B region usage	Output region usage	Temp. region usage>	Engine
Matrix addition	Pq_mtx_add	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	Result matrix	N.A.	Matrix
Matrix substraction	Pq_mtx_sub	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	Result matrix	N.A.	Matrix
Matrix hadamard product	Pq_mtx_hadamard	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	Result matrix	N.A.	Matrix
Matrix product	Pq_mtx_prod	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	Result matrix	N.A.	Matrix

Table continues on the next page...

Table 5. Usage of memory handlers for Matrix engine (continued)

Matrix invert	Pq_mtx_inv	AHB	FP, Fix-16, Fix-32	Matrix M1	N.A.	Result matrix	Max. 1024 words	Matrix
Matrix transpose	Pq_mtx_tran	AHB	FP, Fix-16, Fix-32	Matrix M1	N.A.	Result matrix	N.A.	Matrix
Matrix scale	Pq_mtx_scale	AHB	FP, Fix-16, Fix-32	Matrix M1	N.A. (scale factor in MISC register)	Result matrix	N.A.	Matrix
Vector dot product	Pq_vec_dotp	AHB	FP, Fix-16, Fix-32	Vector A	Vector B	Scaler result	N.A.	Matrix

In the demo case, there are three calculations used for each task:

- task_pq_mat_add() for matrix addition
- task_pq_mat_mul() for matrix multiplication
- task_pq_mat_inv() for matrix inversion

Just like the FFT, the PowerQuad driver implements the CMSIS-DSP API as well. The usage is the same as CMSIS-DSP API. Taking the task_pq_mat_add() as an example,

```
#define PQ_MAT_ROW_COUNT_MAX      16u
  #define PQ_MAT_COL_COUNT_MAX        16u

  /* A + B = C. */
  void task_pq_mat_add(void)
  {
  arm_matrix_instance_f32 matrixA;
  arm_matrix_instance_f32 matrixB;
  arm_matrix_instance_f32 matrixC;
  float32_t mDataA[PQ_MAT_ROW_COUNT_MAX][PQ_MAT_COL_COUNT_MAX];
  float32_t mDataB[PQ_MAT_ROW_COUNT_MAX][PQ_MAT_COL_COUNT_MAX];
  float32_t mDataC[PQ_MAT_ROW_COUNT_MAX][PQ_MAT_COL_COUNT_MAX];
  uint32_t i, j;
  uint32_t  calcTime;

  /* Initialize the matrix. */
  for (i = 0u; i < PQ_MAT_ROW_COUNT_MAX; i++)
  {
  for (j = 0u; j < PQ_MAT_COL_COUNT_MAX; j++)
  {
  mDataA[i][j] = 1.0f * i * PQ_MAT_ROW_COUNT_MAX + j;
  mDataB[i][j] = 1.0f * i * PQ_MAT_ROW_COUNT_MAX + j;
  }
  }
  matrixA.numRows = PQ_MAT_ROW_COUNT_MAX;
  matrixA.numCols = PQ_MAT_COL_COUNT_MAX;
  matrixA.pData   = (float32_t *)mDataA;
  matrixB.numRows = PQ_MAT_ROW_COUNT_MAX;
  matrixB.numCols = PQ_MAT_COL_COUNT_MAX;
  matrixB.pData   = (float32_t *)mDataB;
  matrixC.numRows = PQ_MAT_ROW_COUNT_MAX;
  matrixC.numCols = PQ_MAT_COL_COUNT_MAX;
  matrixC.pData   = (float32_t *)mDataC;
```

```
    /* Calc & Measure. */
    TimerCount_Start();
    arm_mat_add_f32(&matrixA, &matrixB, &matrixC);
    TimerCount_Stop(calcTime);

    /* Todo ...
    * - Record the time.
    * - Display the waveform.
    */
    }
```

There are display pages on the LCD screen module for each of Matrix demo cases when running the demo project, as shown in Figure 3. on page 13.

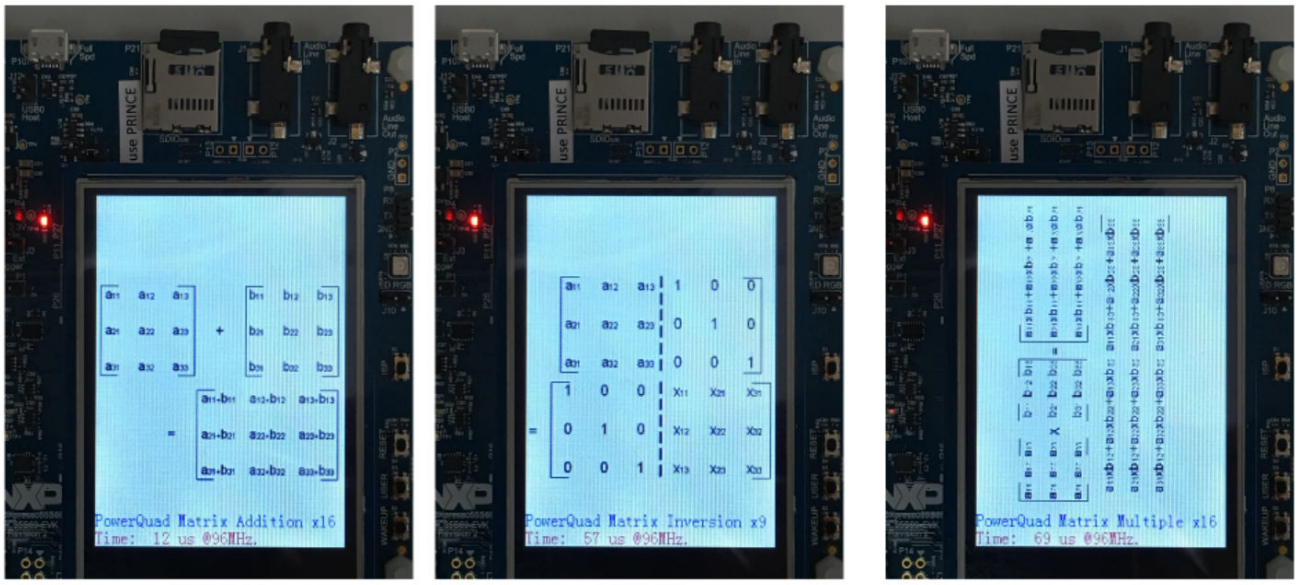


Figure 3. PowerQuad matric addtion/inversion/multiplication

3.5 FIR demo cases

The goal of this demonstration is to create a high-pass/low-pass FIR filter.

There are two demo cases to create different filters:

- task_pq_fir_lowpass() for low-pass filter, to remove the high frequency and get the low frequency from the mixed signal.
- task_pq_fir_highpass()for high-pass filter, to remove the low frequency and get the high frequency from the mixed signal.

In the demo cases, the taps (coefficients) for filters are calculated previously by the Matlab software. Then into the PowerQuad, and the hardware helps to do the filter process to signal automatically, so that time consuming mathematical calculation is avoided.

The original signal is mixed with a low frequency signal (a sine wave at 1 kHz) and a high frequency signal (a sin wave at 15 kHz). See Figure 4. on page 14 for waveform and Figure 5. on page 14 for frequency spectrum.

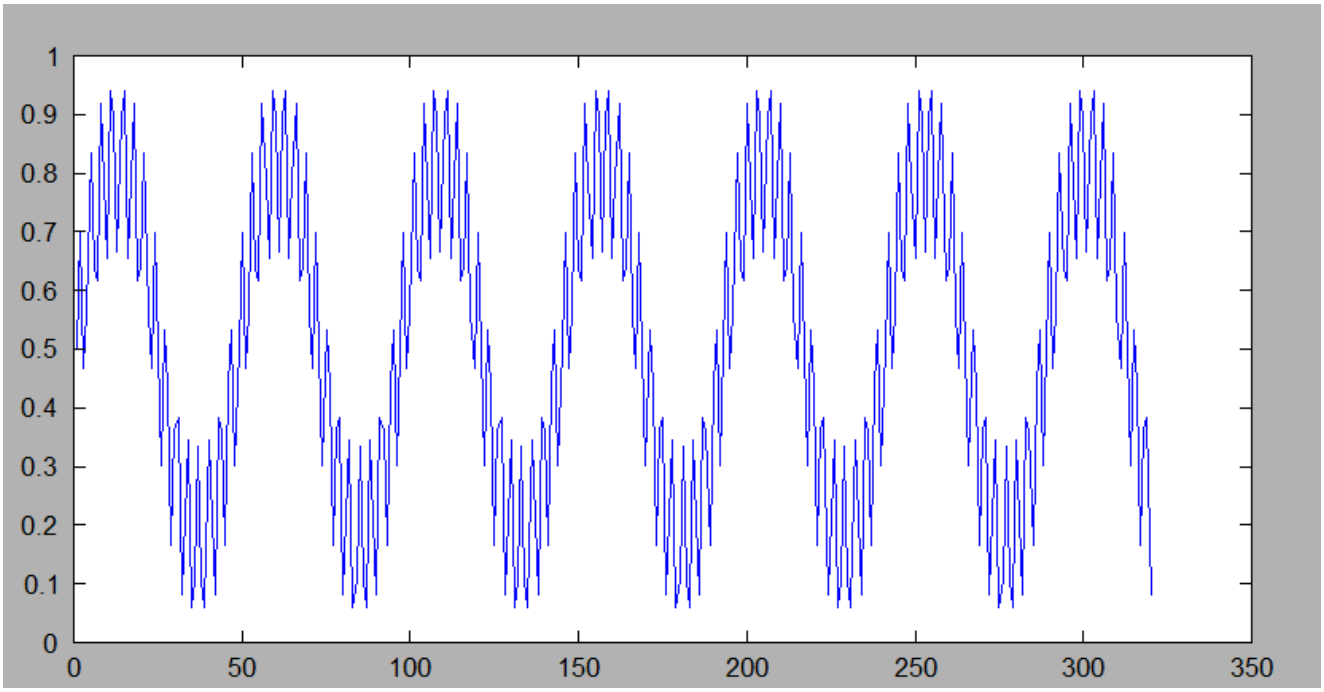


Figure 4. Waveform of mixed signal

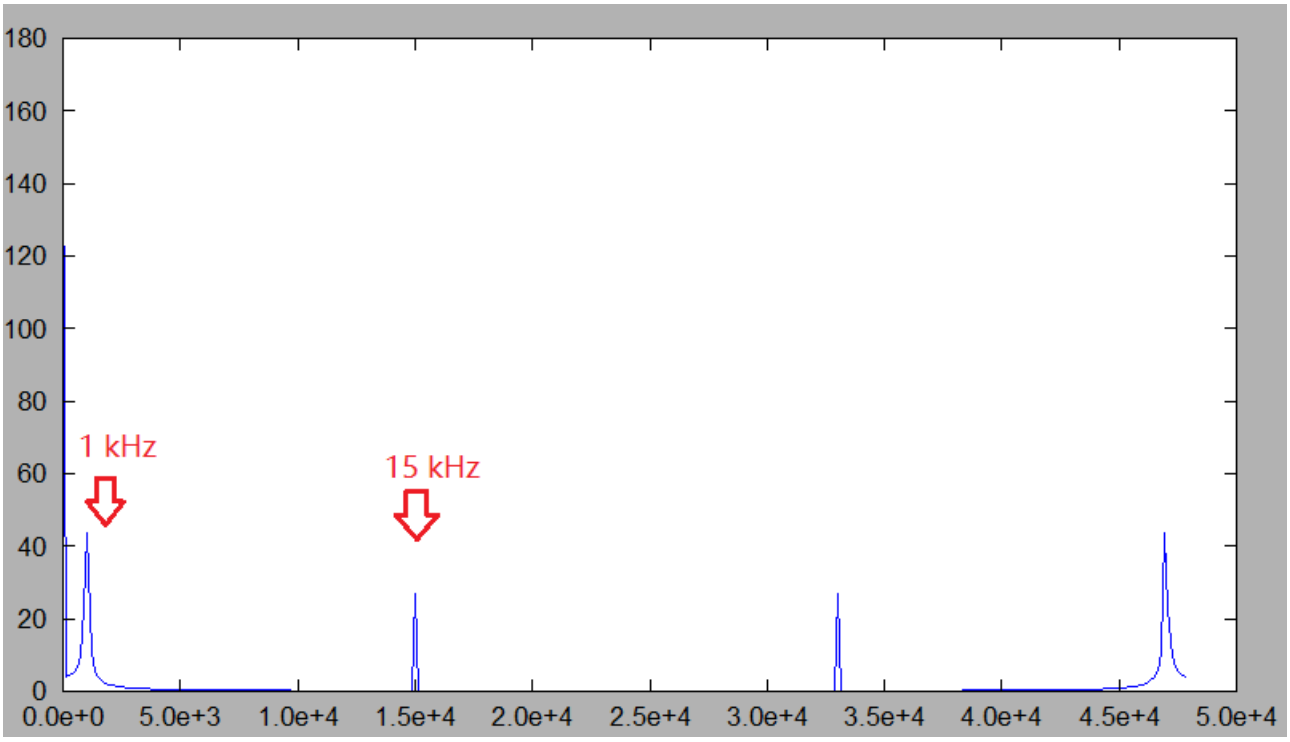


Figure 5. Frequency spectrum of mixed signal

Run the following codes in MatLab to create the coefficients.

```
clear all
close all
Fs=48000;
```

```
T=1/Fs;
Lenght=320;
t=(0:Lenght-1)*T;
Input_signal=(sin(2*pi*1000*t)+0.5*sin(2*pi*15000*t)+1.5)/3;
figure;
plot(Input_signal);

res=fft(Input_signal,Lenght);
figure;
f=((0:Lenght-1)/320*Fs);
plot(f,abs(res));
Cutoff_Freq=6000;
Nyq_Freq=Fs/2;
cutoff_norm=Cutoff_Freq/Nyq_Freq;
order=31;
FIR_Coeff=fir1(order,cutoff_norm,'high'); % for high-pass
%FIR_Coeff=fir1(order,cutoff_norm); % for low-pass
Filterd_signal=filter(FIR_Coeff,1,Input_signal);
figure;
plot(Filterd_signal);

fvtool(FIR_Coeff,'Fs',Fs); % generate the coeff and display the diagram
```

The filter features are:

- Type: high-pass/low-pass
- Order: 32
- Sampling frequency: 48 kHz
- Cut-off frequency:6 kHz

Response reports are shown in following figures.

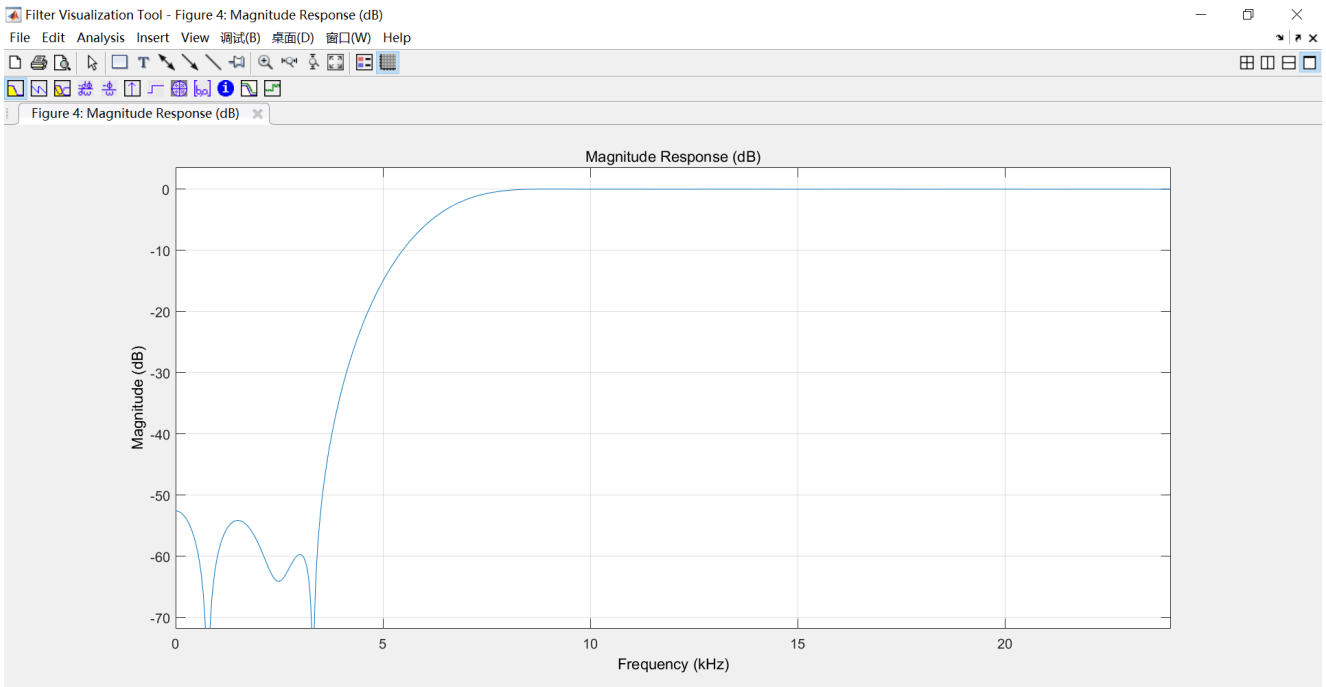


Figure 6. Magnitude response of FIR filter

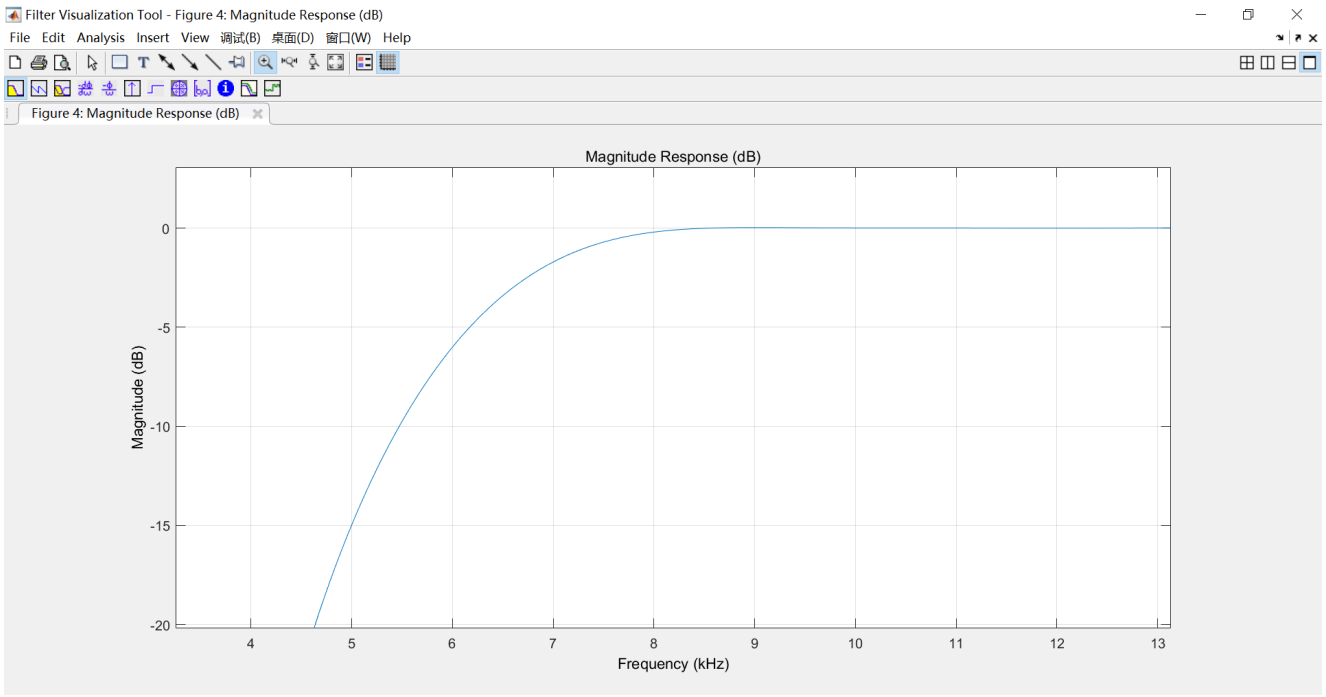


Figure 7. Magnitude response of FIR filter

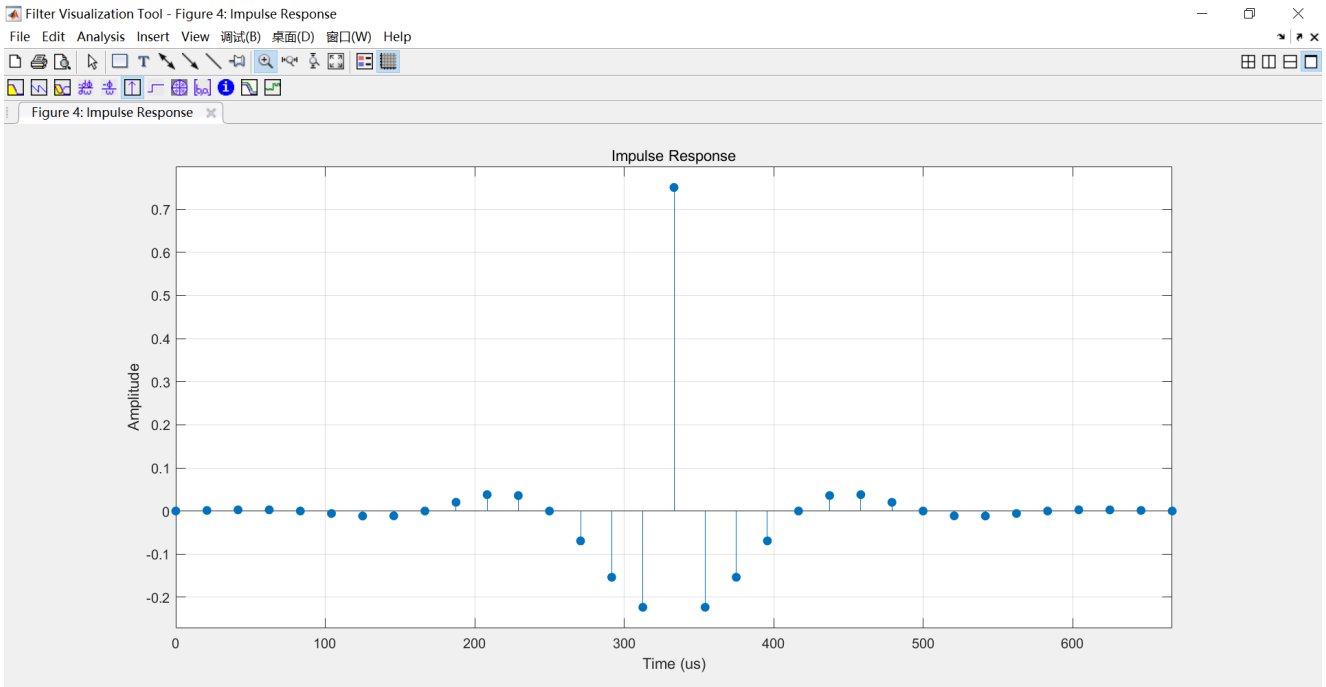


Figure 8. Impulse response of FIR filter

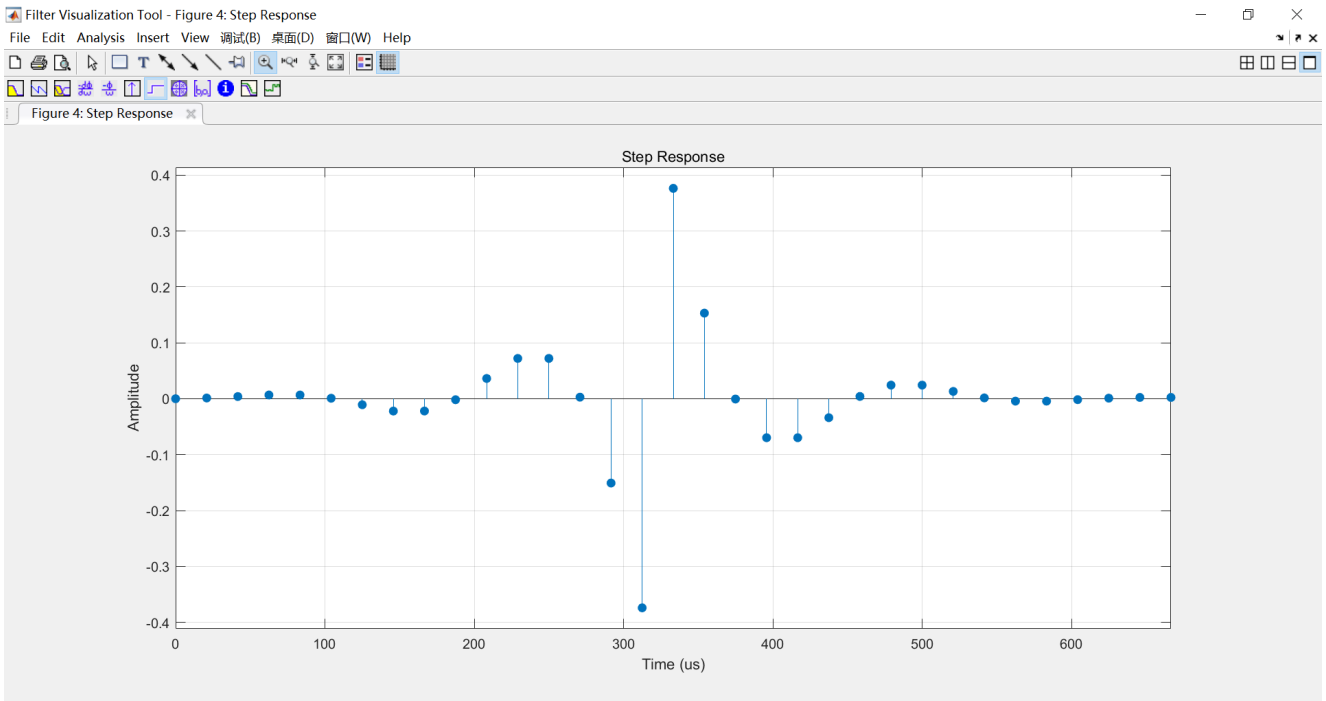


Figure 9. Step response of FIR filter

Then, setup the PowerQuad to execute the filter process on MCU, taking high-pass task as an example.

```
void task_pq_fir_highpass(void)
{
    uint32_t i;
    uint32_t Fs=48000;

    arm_fir_instance_f32 S;
    float32_t  *inputF32, *outputF32;
    uint32_t calcTime;

    inputF32 = &gPQFirF32In[0];
    outputF32 = &gPQFirF32Out[0];

    /* Generate the wave. */
    for (i = 0; i < FIR_INPUT_LEN; i++)
    {
        gPQFirF32In[i]  = 1.5
                        + 0.5 * arm_sin_f32(2*PI*15000*i/Fs)
                        +       arm_sin_f32(2*PI*1000*i/Fs) ;
        gPQFirF32In[i] /= 3.0f;
    }

    // ...

    /* Call FIR init function to initialize the instance structure. */
    arm_fir_init_f32(   &S,
                        NUM_TAPS,
                        (float32_t *)&firCoeffs32_highpass[0],
                        &firStateF32[0],
                        FIR_INPUT_LEN );
```

```
        PQ_Init(POWERQUAD_NS);
        pq_config_t pqConfig;

        pqConfig.inputAFormat = kPQ_Float;
        pqConfig.inputAPrescale = 0;
        pqConfig.inputBFormat = kPQ_Float;
        pqConfig.inputBPrescale = 0;
        pqConfig.outputFormat = kPQ_Float;
        pqConfig.outputPrescale = 0;
        pqConfig.tmpFormat = kPQ_Float;
        pqConfig.tmpPrescale = 0;
        pqConfig.machineFormat = kPQ_Float;
        pqConfig.tmpBase = (uint32_t *)0xE0000000;
        PQ_SetConfig(POWERQUAD_NS, &pqConfig);

        /* move the taps into private RAM to improve the performance of operating memory. */
        PQ_MatrixScale( POWERQUAD_NS,
                        POWERQUAD_MAKE_MATRIX_LEN(16, NUM_TAPS / 16, 0),
                        1.0,
                        firCoeffs32_highpass,
                        EXAMPLE_PRIVATE_RAM );
        PQ_WaitDone(POWERQUAD_NS);

        /* In the next calculation, data in private ram is used. */
        pqConfig.inputBFormat = kPQ_Float;
        pqConfig.outputFormat = kPQ_Float;
        PQ_SetConfig(POWERQUAD_NS, &pqConfig);

        TimerCount_Start();
        PQ_FIR(POWERQUAD_NS, inputF32, APP_PQ_FIR_SAMPLE_COUNT_240, EXAMPLE_PRIVATE_RAM, NUM_TAPS,
outputF32,PQ_FIR_FIR);
        PQ_WaitDone(POWERQUAD_NS);
        //arm_fir_f32(&S, inputF32, outputF32, FIR_INPUT_LEN);
        TimerCount_Stop(calcTime);

        /* Todo ...
         * - Record the time.
         * - Display the waveform.
         */
    }
```

When running the demo cases to execute the filter with PowerQuad hardware, the results are shown in the LCD Screen, as shown in Figure 10. on page 19.

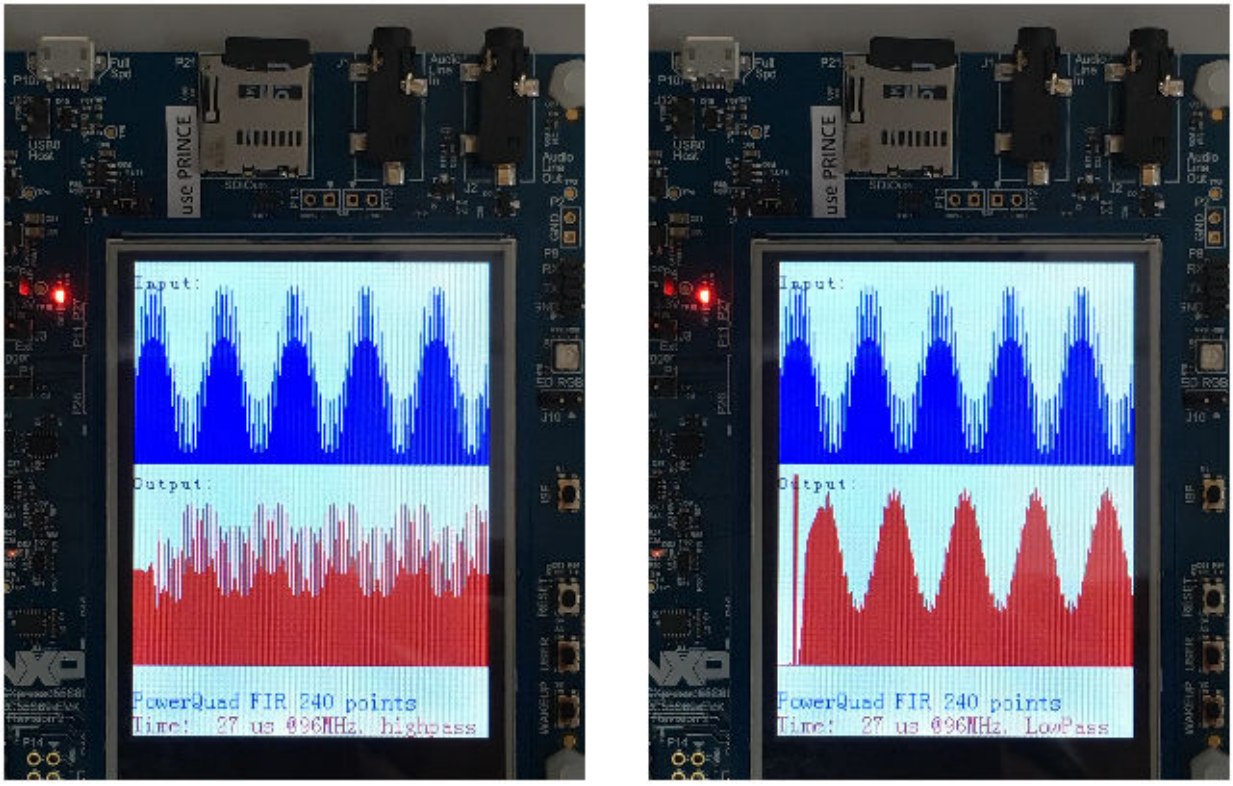


Figure 10. PowerQuad FIR High-Pass/Low-Pass filter

4 PowerQuad vs Arm CMSIS-DSP performance

Finally, in the demo project, a page is setup for the comparison between the PowerQuad and Arm CMSIS-DSP when they are running the same tasks. To make a fair comparison, when running the DSP task, the Arm CMSIS-DSP code is running in RAM while the PowerQaud is using the dedicated RAM (the private one), so that they can achieve the highest performance.

Figure 11. on page 20 shows the snapshot of the screen.

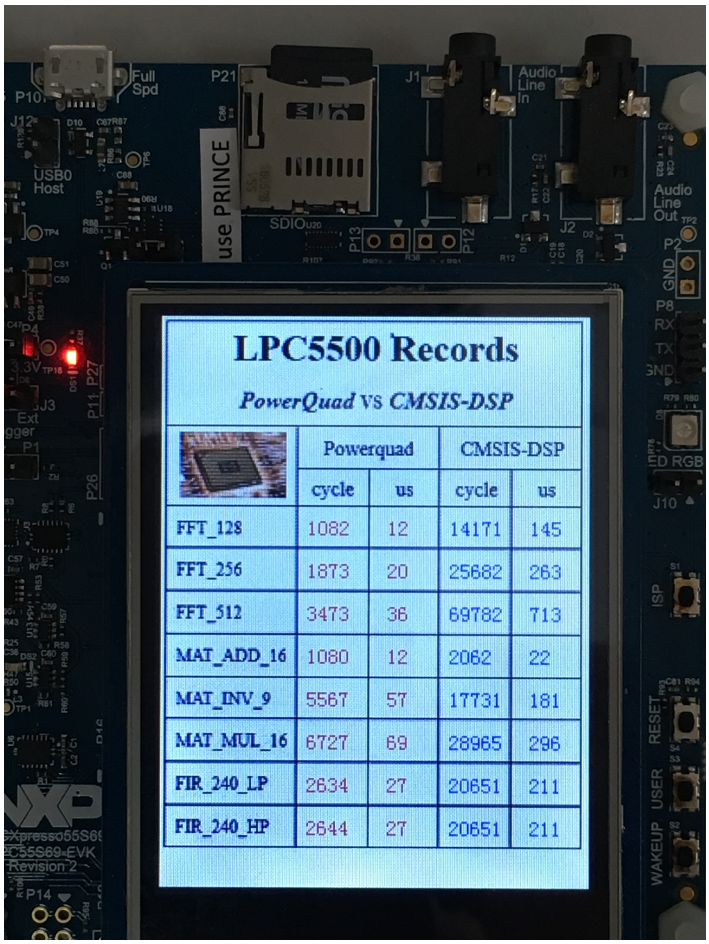
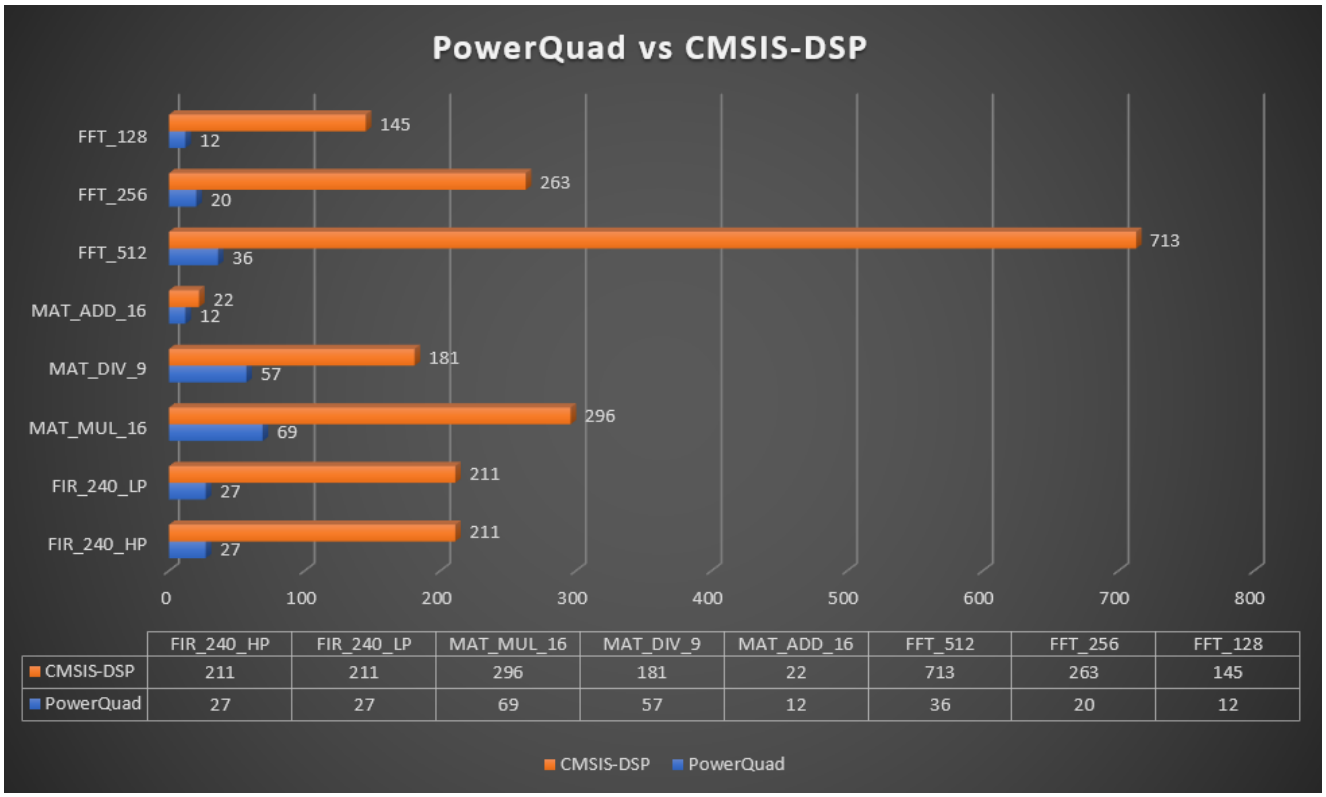


Figure 11. Running time for PowerQuad vs Arm CMSIS-DSP

Figure 12. on page 21 summarizes the data.



	FIR_240_HP	FIR_240_LP	MAT_MUL_16	MAT_DIV_9	MAT_ADD_16	FFT_512	FFT_256	FFT_128
CMSIS-DSP	211	211	296	181	22	713	263	145
PowerQuad	27	27	69	57	12	36	20	12

Figure 12. Running time data table for PowerQuad vs Arm CMSIS-DSP

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 24 January 2019
Document identifier: AN12282

arm