AN12282

NXP LPC5500 使用 PowerQuad 做数字信号处理

版本 0 — 2019 年 1 月 24 日

应用笔记

目录

1 PowerQuad 简介

随着移动物联网和环境感知应用的迅速增长，设备需要做更多的本地数字信号处理。基于 Cortex M 的 MCU 对于低功耗常开系统是不错的选择（考虑到有限的计算量，可减少功耗泄漏并降低总体功耗）。

Arm Cortex-M 架构更适合节能控制应用场景。在信号处理应用中相对传统 DSP 架构的芯片并不占优势，由于以下因素有时在性能上差距可能高达 10 倍至 20 倍：

- 存储器带宽窄（单个 32 位数据总线）– DSP 通常至少具有两条数据总线以及本地存储器块。
- 有限的同时计算能力（例如，每个周期一个乘加运算）。
- 没有足够的寄存器用来在计算过程中间保存必要的数据。
- 没有专用的内置加速器来实现诸如傅里叶变换（大量加/减），Biquad 滤波器之类的功能。

尽管 Arm 并未为 Cortex-M 系列内核带来大规模的 DSP 改进，但它已经标准化了 DSP 库（CMSIS DSP 库）。当用户将通用标准接口用于 DSP 功能时，就有机会由芯片厂商为 DSP 计算提供优化方案。 用户的代码仍然使用 CMSIS DSP，但是 NXP 可以“从本质上提升”。要注意的另一个关键点是，加速计算不仅可以使 MCU 更早地进入睡眠状态而降低 MCU 的功耗，而且还可以通过在较低的频率下以较低的速度运行来降低电压（从而进一步降低能耗）。然后 PowerQuad 应运而生。

以下是 DSP 应用中的一些典型数学计算需求：

- 运动感知
  - 矩阵运算，通过三角函数的旋转，傅里叶变换，滤波器（FIR / IIR）。
  - 用于运动特征提取和匹配的卷积和相关。
- 语音识别
  - 傅里叶变换用于频谱分析、MFCC 以及其他加窗（矩阵乘法）、滤波器（FIR / IIR），离散余弦变换用于倒谱提取。
  - 用于特征提取和比较的统计建模。
- 神经网络架构的特定功能
  - 矩阵 MAC
  - Logistic/Sigmoid 函数（使用幂运算）用于感知器评估（对于统计分布分析也非常有用）。
- 生物识别
  - 傅里叶变换用于心跳监视，Arctan /其他触发用于指纹识别。

现在，PowerQuad 可以在硬件上支持多数的数学计算需求，从而可以累加进程并同时节省其他线程的 CPU 时间。

2 PowerQuad 硬件

2.1 PowerQuad 计算功能

作为集成在芯片中的硬件模块，PowerQuad 的计算任务全部在硬件上执行。它涉及多种计算引擎：

- 变换引擎
- 超越功能引擎
- 三角函数引擎
- 双二阶 IIR 滤波器引擎
- 矩阵加速器引擎
- FIR 滤波器引擎
- CORDIC 引擎

表 1 列出了 PowerQuad 直接支持的计算功能。

表 1. PowerQuad 硬件功能

类别	函数	注解
数学	1/x, ln(x), sqrt(x), 1/sqrt(x), e^(x), e^(-x), (x1)/(x2), sin(x), cos(x)	协处理器指令
	arctan(x), arctanh(x)	—
过滤	• 二阶 IIR 滤波器	协处理器指令
	• FIR 滤波器 • FIR 滤波器增量 • 相关性 • 卷积	—
矩阵	• 缩放 • 加法 • 减法 • 求逆 • 矩阵乘法 • 哈达玛积（元素积） • 转置 • 点积	—
转变	• 复数傅里叶变换（复数值输入序列） • 实数傅里叶变换（实数值输入序列） • 傅里叶逆变换 • 复数离散余弦变换（复数值输入序列） • 实数离散余弦变换（实数值输入序列） • 离散余弦逆变换	—

这些功能构成了实现高级算法的基础。

2.2 PowerQuad 总线接口

PowerQuad 与 Arm Cortex-M33 协处理器接口集成在一起，因此可以通过协处理器指令（MCR 和 MRC）进行访问。另外，在 PowerQuad 内部设计有可编程的寄存器来连接 AHB 总线。这意味着在 Cortex-M33 内核上运行的用户代码可以像其他普通可编程模块一样读写其寄存器。请参阅 图 1。

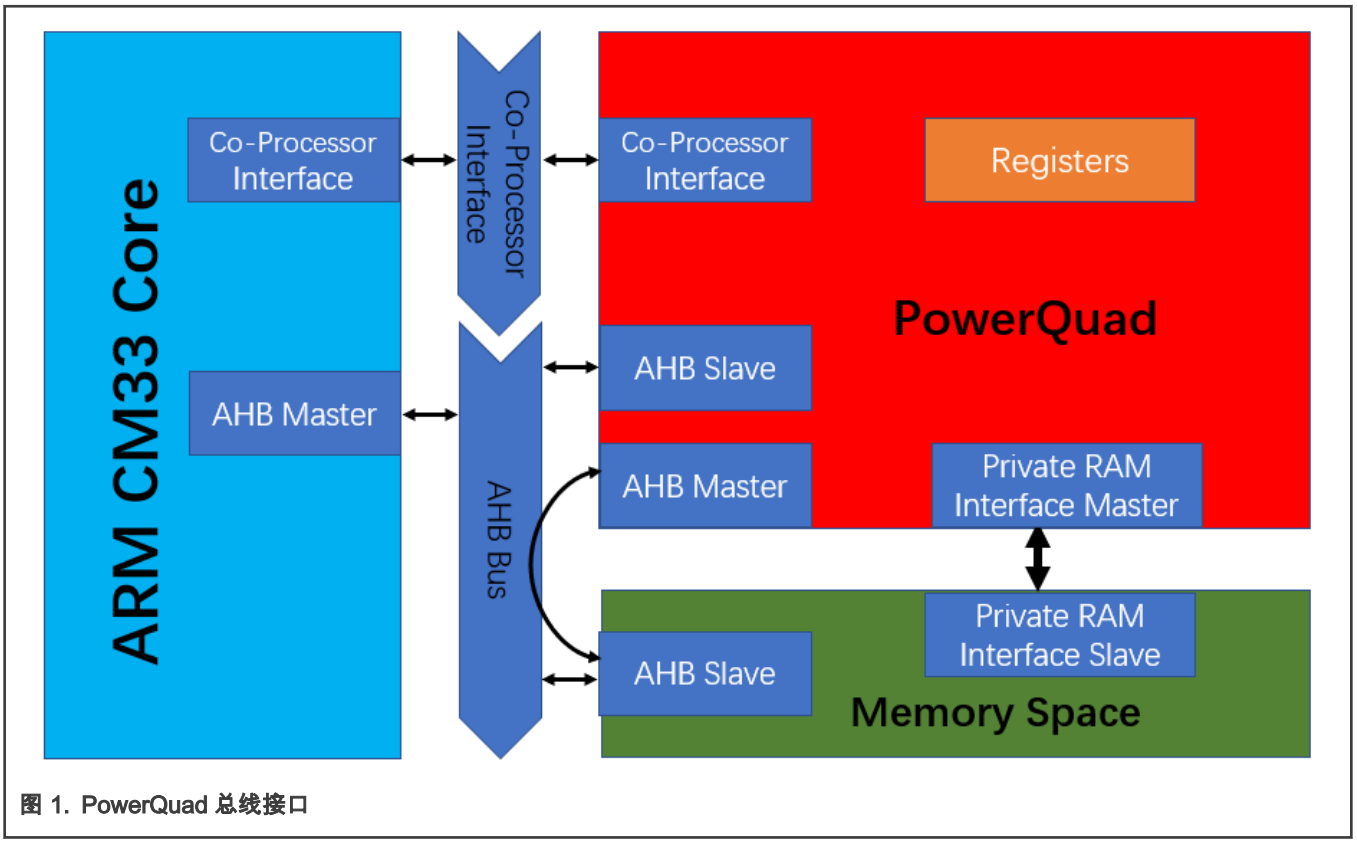


图 1. PowerQuad 总线接口

但是，特定的访问方式适用于特定的用途。通常，对于 PowerQuad，Arm Cortex-M 协处理器接口和 AHB 从属接口用于传递命令/配置，而 AHB 主接口和专用 RAM 主接口用于操作内存。

- 协处理器功能

  在进行接受一个数字作为输入参数并返回一个数字作为输出结果的计算时，他们通常会使用 Cortex-M 协处理器接口来传递输入参数并返回结果。例如，大多数数学函数都是以这种方式实现的。这些功能很简单，运行得很快。

- 流/DMA 功能

  在对一组数据进行计算，结果是另一组数据时，PowerQuad 使用类似于 DMA 的方式来处理输入和输出数据。AHB 访问功能的示例包括变换功能、矩阵功能和大多数过滤器功能。将 PowerQuad 用于这些功能时，用户需要设置一些 PowerQuad 的基址寄存器，例如使用 DMA，然后 PowerQuad 硬件在启动计算时会自动使用这些地址指示的内存。

NXP MCUXpresso SDK 已经提供了 PowerQuad 的驱动程序。它打包了协处理器接口操作（协处理器指令）和 AHB 总线操作（功能寄存器）。因此，如果用户使用 SDK API 开发应用程序，则无需关心如何选择指令或寄存器设置。

2.3 PowerQuad 内存处理程序

当被视为嵌入式数学计算机时，PowerQuad 需要处理和产生大量数据。除功能强大的计算引擎外，还有四个用于内存处理程序的组，它们指示四个内存区域来支持 PowerQuad 函数的数据管理要求。

- Input A.指向输入数据数组 1 的指针。
- Input B.必要时指向输入数据数组 2 的指针。例如，当进行矩阵加法时，另一个矩阵将由 Input B 处理程序指示。
- Temp.指向临时存储器的指针，该临时存储器在必要时保留中间计算结果（用于傅里叶变换和矩阵求逆）。内存应在当前计算之前初始化，并可以在以后清除。

可以将四个存储区中的每个存储区配置为自定义格式：

- 原始数据的格式（32 位定点，16 位定点或 32 位浮点型）
- PowerQuad 所需的数据格式（除傅里叶变换（定点引擎）以外的所有数据均为浮点数）
- 结果的缩放比例（PowerQuad 可以在输出的途中按 2 的幂进行缩放。）

用户可以将准备好的存储器的地址填充到 PowerQuad 模块的响应寄存器中。请参阅 表 2。

表 2. 用于内存处理程序的 PowerQuad 寄存器

地址	名字	描述	访问	重置值
0x000	OUTBASE	输出区域的基址寄存器	RW	0
0x004	OUTFORMAT	输出区域的数据格式	RW	0
0x008	TMPBASE	临时区域的基址寄存器	RW	0
0x00C	TMPFORMAT	临时区域的数据格式	RW	0
0x010	INABASE	输入 A 区域的基址寄存器	RW	0
0x014	INAFORMAT	区域输入 A 的数据格式	RW	0
0x018	INBBASE	输入 B 区域的基址寄存器	RW	0
0x01C	INBFORMAT	区域输入 B 的数据格式	RW	0

PowerQuad 可以处理常规 RAM 内存（与 Cortex-M 内核等其他 AHB 主设备共享）和专用 RAM 内存（从 0xE000_0000 开始，16 KB）。特别是对于私有 RAM 内存，由于仅为 PowerQuad 保留，因此 PowerQuad 可以无任何仲裁延迟地访问它，从而为 PowerQuad 节省了很多时间来获取数据。然后，PowerQuad 可以并行访问私有 RAM 的四个存储区，从而提供 128 比特位宽。因此，它甚至可以更快地执行某些功能，例如傅里叶变换，FIR，卷积，矩阵等。

有关使用专用 RAM 的一些注意事项：

- 傅里叶变换引擎只能将专用存储器用作临时存储器（不能用作输入或输出）。
- 私有内存中的所有数据必须为浮点数。您可以通过以私有内存为目标的矩阵缩放操作将数据移入和移出私有内存）。
- 私有内存不提供任何缩放。缩放仅适用于正在读取/写入系统内存的数据。

3 PowerQuad DSP 示例

本节介绍了 PowerQaud 在应用程序中的基本用法。 在演示案例的说明中，将提及 PowerQuad API 的描述。

该示例在带有 LCD 屏幕模块的 LPCXpresso5500（OM40011）板上运行，以显示 GUI。在示例工程中，设计了一个简单的框架作为调度程序来切换单独的任务。然后可以针对傅里叶变换、矩阵和 FIR 逐一执行。通过 LCD 屏，显示功能也集成到框架中。

在本示例中选择了 PowerQuad 傅里叶变换、矩阵和 FIR 滤波器，因为这些计算在大多数 DSP 应用中都很流行，但是当用纯软件（Arm CMSIS-DSP Lib）实施时通常会花费大量时间。在本节的最后，提供了 PowerQuad API 和 Arm CMSIS-DSP API 的性能比较。

注意，有关计算过程的详细内容将不在本文中讨论。有关更多信息，请参阅 PowerQuad UM 和 SDK 驱动程序代码。

针对傅里叶变换情况，本节描述了有关使用 PowerQuad API 的详细说明。同样的想法也适用于其他情况。

3.1 带有显示 GUI 的任务调度

为了将不同的案例合并到一个工程中，在示例工程中应用实现了一个调度器。每种案例都在一个函数内实现并将该函数作为任务入口。所有任务入口都集中到任务数组 cAppLcdDisplayPageFunc[]中。此外，还启动了一个用于捕获按钮的硬件线程。

然后，MCU 将处于睡眠模式，直到被按键中断唤醒。 在按键中断的中断服务程序中更改键值。 主循环将检查键值的更改，并使用任务列表中的索引（使用键值）切换到任务。

```
    /* 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. */
       }
     }
```

在每个任务中，它都会执行 PowerQuad 计算以完成一个简单的任务，并测量关键操作的时间。然后通过 LCD 屏幕显示测量结果。

3.2 计时功能

考虑到这些函数通常运行速度很快，因此在演示案例中不适合使用基于中断的计时方法。但是，在一些专门用于测量的测试项目中，仍然可以通过基于中断的时间测量方法来测量目标函数多次调用的总时间而获得一次执行的平均时间。

在此示例中，选择 SysTick 作为计时器，因此此处的代码可以很好地移植到其他 Arm Cortex-M MCU 中。然后直接使用 24 位计数器值进行计时。对于运行在 98 MHz 的 SysTick 定时器时钟源的 LPC5500，最大定时周期可能为 171 毫秒。

```
    /* 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)
```

用法是：

```
uint32_t calcTime;

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

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

3.3 傅里叶变换演示案例

示例中有三种傅里叶变换案例：128 点，256 点和 512 点。

使用 PowerQuad 傅里叶变换引擎的技巧是：

- PowerQuad 可以在硬件上支持傅里叶变换计算引擎的 16/32/64/128/256/512 点。
- PowerQuad 傅里叶变换引擎在计算傅里叶变换（以及扩展离散余弦变换）时，总是按 1/N 的比例缩放输入数据。如果需要非缩放结果，则必须首先手动将输入数据（在输入 A 区域）乘以 N。FFT 按 1/N 进行缩放，但根据 iDFT 公式这是正确的，因此不需要进行缩放处理。
- 傅里叶变换引擎仅查看输入字的低 27 比特位，因此预缩放不能超过该比特位以避免饱和。
- 纯实数函数（在 API 名称中以'r'作为前缀）和复数函数（在 API 名称中以 c 作为前缀）期望输入数据序列按以下方式排列在内存中。
- 如果序列 x = x0，x1 ... xN-1 是实数，则存储器中的输入数组必须以 x [N] = {x0，x1，... xN-1}的形式存放。
- 如果序列 x = x0，x1 ... xN-1 是（x0_real + i * x0_im），（x1_real + i * x1_im），...（xN-1_real + i * xN-1_im）形式的复数，则内存中的输入数组必须以 x [N] = {x0_real，x0_im，x1_real，x1_im，... xN-1_real，xN-1_im}的形式存放。
- 输出序列始终以复数数组形式存储在存储器中，其中实数输出数据的虚部为零。

当运行 PowerQuad 变换引擎（包括傅里叶变换）时，只有 INPUT A 用于输入，而 OUT 内存处理程序用于输出。有关变换引擎内存处理程序使用情况的完整信息，请参阅 表 3。

表 3. 傅里叶变换引擎的内存处理程序的用法

操作方式	驱动功能	访问类型	输入/输出数据格式	Input A 区域用法	Input B 区域用法	输出区域使用	临时区域使用	定点输入/输出定标器	引擎	是否使用 GPREG/ COMPREG？
复数傅里叶变换	Pq_cfft	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
实数傅里叶变换	Pq_rfft	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
傅里叶逆变换	Pq_ifft	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scale	Xform	Yes

下页继续...

表 3. 傅里叶变换引擎的内存处理程序的用法（续上页）

操作方式	驱动功能	访问类型	输入/输出数据格式	Input A 区域用法	Input B 区域用法	输出区域使用	临时区域使用	定点输入/输出定标器	引擎	是否使用 GPREG/ COMPREG？
								r/ Out_scaler		
复数离散余弦变换	Pq_cdct	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
实数离散余弦变换	Pq_rdct	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes
逆离散余弦变换	Pq_idct	AHB	Fix-16, Fix-32	Input data	N.A.	Output data	N.A.	Ina_scaler/ Inb_scaler/ Out_scaler	Xform	Yes

演示中使用的 PowerQuad API 设计为与 CMSIS-DSP API 兼容。 因此，对于 CMSIS-DSP 用户而言，他们不需要更改现有代码，就可以通过 PowerQuad 实现更快地运行。

以 128 点的傅里叶变换为例：

```
    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() 调用 PowerQuad 驱动程序 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;
    }
```

然后，PQ_TransformCFFT() 函数配置 PowerQuad 寄存器以设置输入/输出和内存长度，然后通过启用 PowerQuad 作为 CFFT 引擎来启动计算。 完成这些操作后，PowerQuad 即可工作。

```
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. */
 }
```

开始计算后，INST_BUSY 被置位。用户可以使用 PQ_WaitDone() 函数来等待 PowerQuad 完成。

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

运行示例工程时，LCD 屏上有每种傅里叶变换演示案例的显示页面，如 图 2 所示。

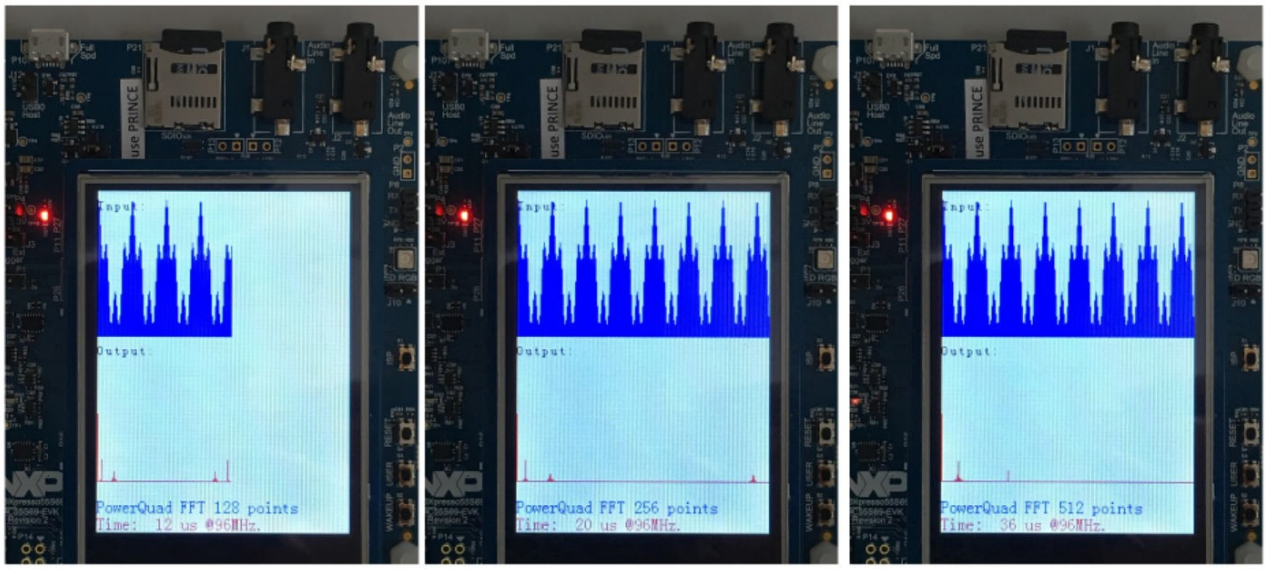


图 2. PowerQuad 傅里叶变换 128/256/512 点

3.4 矩阵演示案例

矩阵加速器引擎支持八种操作，如 表 4 所示，给出了各自的最大支持维度。

表 4. PowerQuad 矩阵长度范围

PowerQuad 引擎	运作方式	最大行
矩阵	加法	16 × 16
	减法	16 × 16
	哈达玛积	16 × 16
	矩阵乘法	16 × 16
	矢量点积	256 元素
	矩阵求逆	9 × 9
	转置	16 × 16
	缩放	16 × 16

矩阵数据应按标准 C / C ++数组的顺序逐行存储在内存中。 因此，如果两个 2×2 整数矩阵 A 和 B 为：

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

那么输入数据将预计存储在内存阵列中，如下所示：

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

有关 PowerQuad Matrix 引擎的内存处理程序的用法，请参阅 表 5。

表 5. 矩阵（ Matrix ）引擎的内存处理程序的用法

运作方式	驱动功能	访问类型	输入/输出数据格式	Input A 区域用法	Input B 区域用法	输出区域使用	临时区域使用情况	引擎
矩阵加法	Pq_mtx_add	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	结果矩阵	N.A.	矩阵（ Matrix ）
矩阵减法	Pq_mtx_sub	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	结果矩阵	N.A.	矩阵（ Matrix ）
矩阵哈达玛积	Pq_mtx_hadamard	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	结果矩阵	N.A.	矩阵（ Matrix ）
矩阵积	Pq_mtx_prod	AHB	FP, Fix-16, Fix-32	Matrix M1	Matrix M2	结果矩阵	N.A.	矩阵（ Matrix ）
矩阵逆	Pq_mtx_inv	AHB	FP, Fix-16, Fix-32	Matrix M1	N.A.	结果矩阵	Max. 1024 words	矩阵（ Matrix ）
矩阵转置	Pq_mtx_tran	AHB	FP, Fix-16, Fix-32	Matrix M1	N.A.	结果矩阵	N.A.	矩阵（ Matrix ）

下页继续..

表 5. 矩阵（ Matrix ）引擎的内存处理程序的用法（续上页）

运作方式	驱动功能	访问类型	输入/输出数据格式	Input A 区域用法	Input B 区域用法	输出区域使用	临时区域使用情况	引擎
矩阵缩放	Pq_mtx_scale	AHB	FP, Fix-16, Fix-32	Matrix M1	N.A. (scale factor in MISC register)	结果矩阵	N.A.	矩阵（ Matrix ）
矢量点积	Pq_vec_dotp	AHB	FP, Fix-16, Fix-32	Vector A	Vector B	标量结果	N.A.	矩阵（ Matrix ）

在演示案例中，每个任务使用三种计算：

- task_pq_mat_add（ ）用于矩阵加法
- task_pq_mat_mul（ ）用于矩阵乘法
- task_pq_mat_inv（ ）用于矩阵求逆

就像傅里叶变换一样，PowerQuad 驱动程序也实现了 CMSIS-DSP API。用法与 CMSIS-DSP API 相同。 以 task_pq_mat_add（ ）为例，

```
#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.
        */
        }
```

运行示例工程时，每个 Matrix 演示案例在 LCD 屏幕模块上都有显示页面，如 图 3 所示。

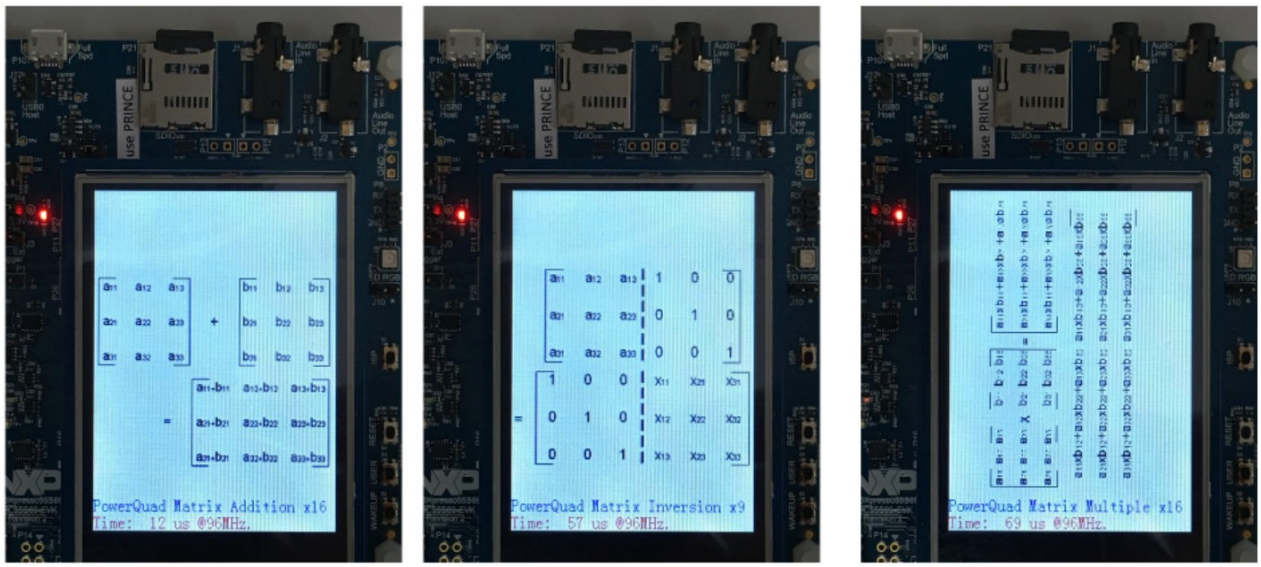


图 3. PowerQuad 矩阵加法/求逆/乘法

3.5 FIR 演示案例

该示例的目的是创建一个高通/低通 FIR 滤波器。

有两个演示案例可创建不同的滤波器：

- task_pq_fir_lowpass（ ）用于低通滤波器，以去除混合信号中的高频信号并获取低频信号。
- task_pq_fir_highpass（ ）用于高通滤波器，以去除混合信号中的低频信号并获取高频信号。

在演示案例中，滤波器的抽头（系数）是由 Matlab 软件预先计算的。 然后进入 PowerQuad，硬件实现对信号的滤波处理，从而避免了费时的数学计算。

原始信号是低频信号（1 kHz 的正弦波）和高频信号（15 kHz 的正弦波）的混合信号。波形请参见 图 4。频谱请参见 图 5。

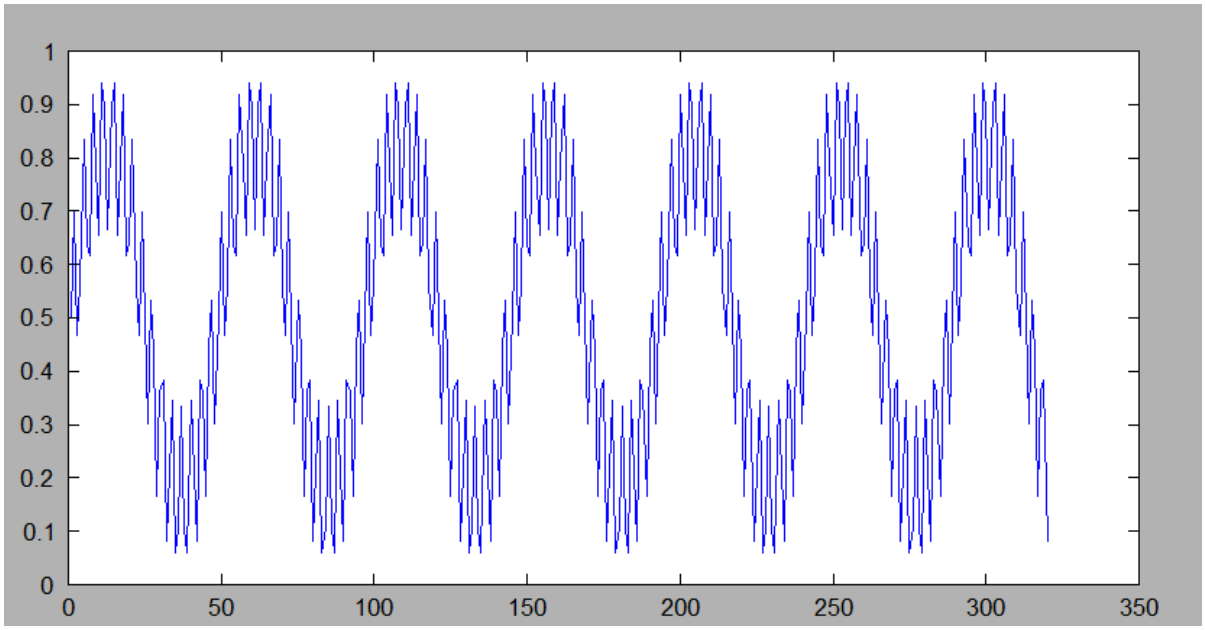


图 4. 混合信号波形

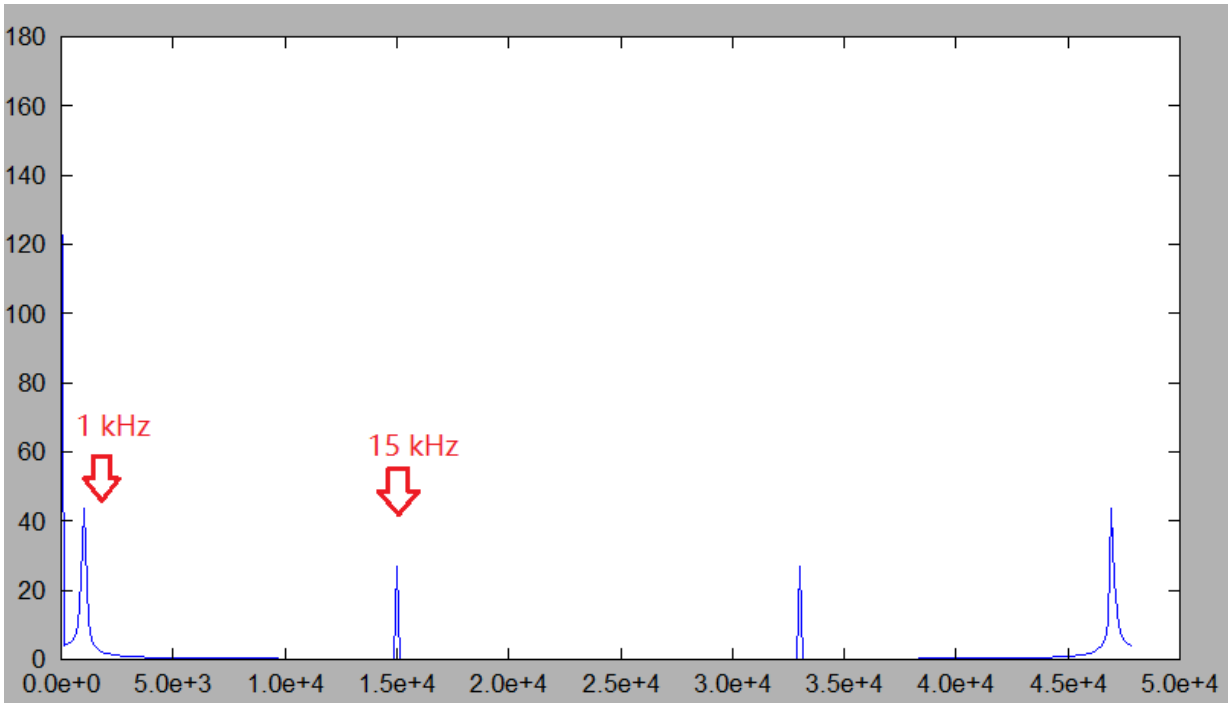


图 5. 混合信号频谱

在 MatLab 中运行以下代码以创建系数。

```
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
```

滤波器的特征是：

- 类型：高通/低通
- 阶数：32
- 采样频率：48 kHz
- 截止频率：6 kHz

响应报告如 图 6 至 图 9 所示。

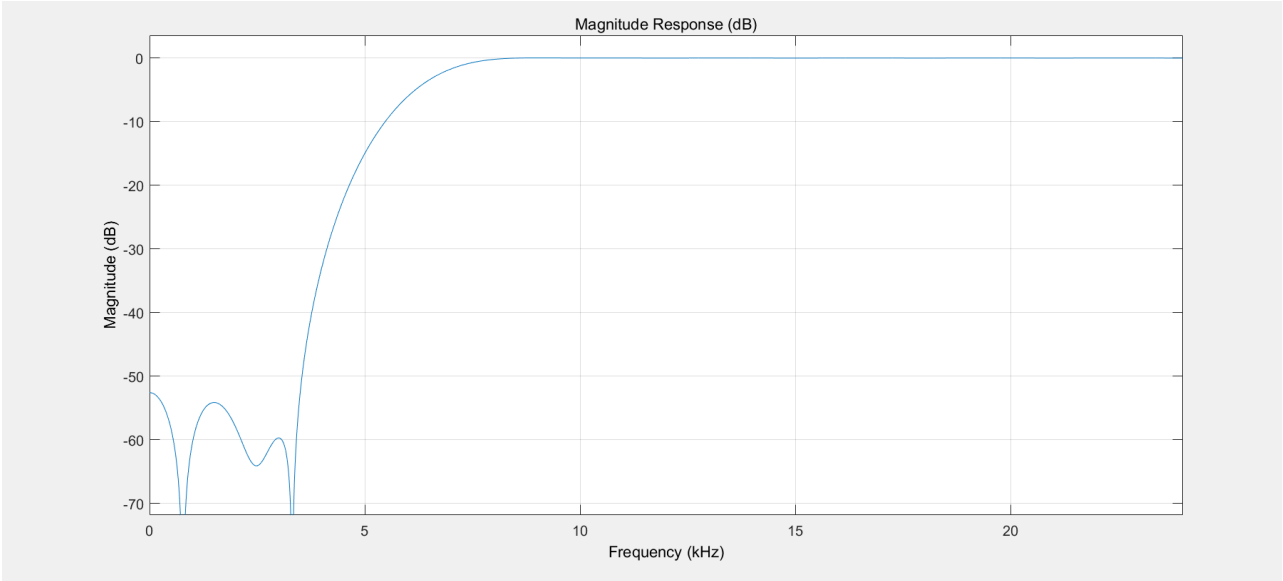


图 6. FIR 滤波器的幅度响应

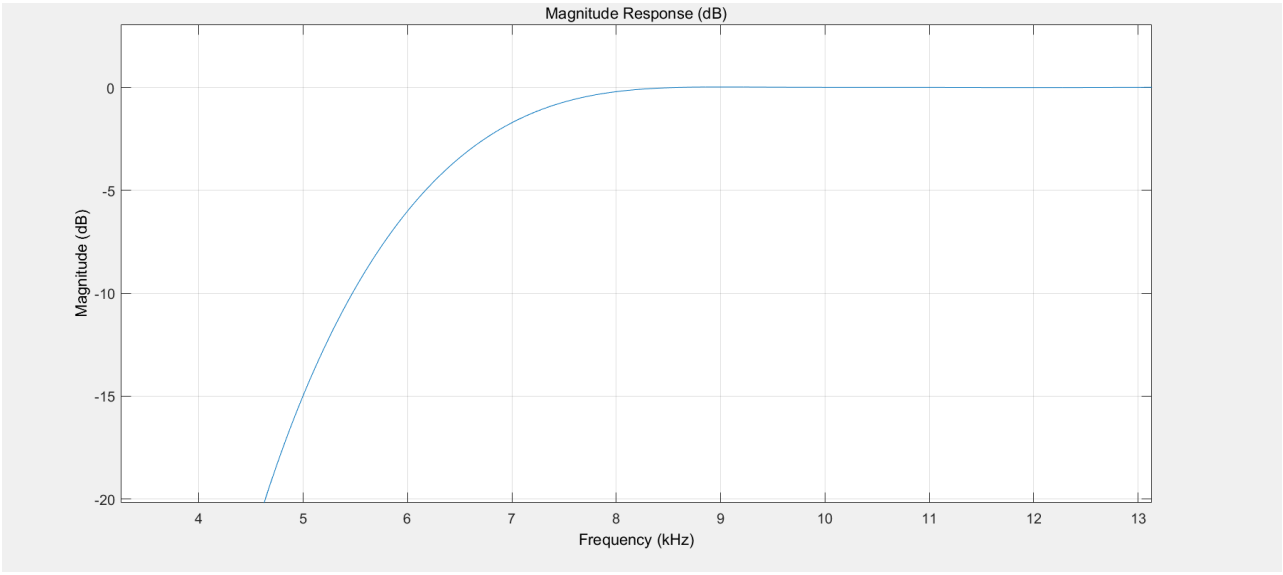


图 7. FIR 的幅度响应

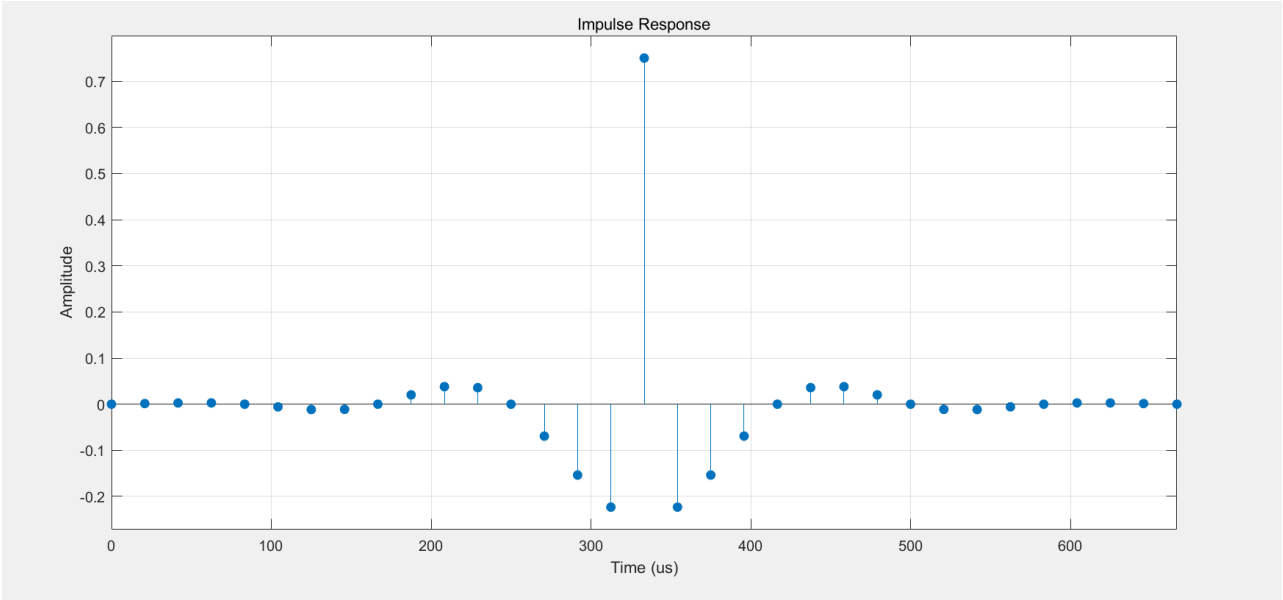


图 8. FIR 滤波器的脉冲响应

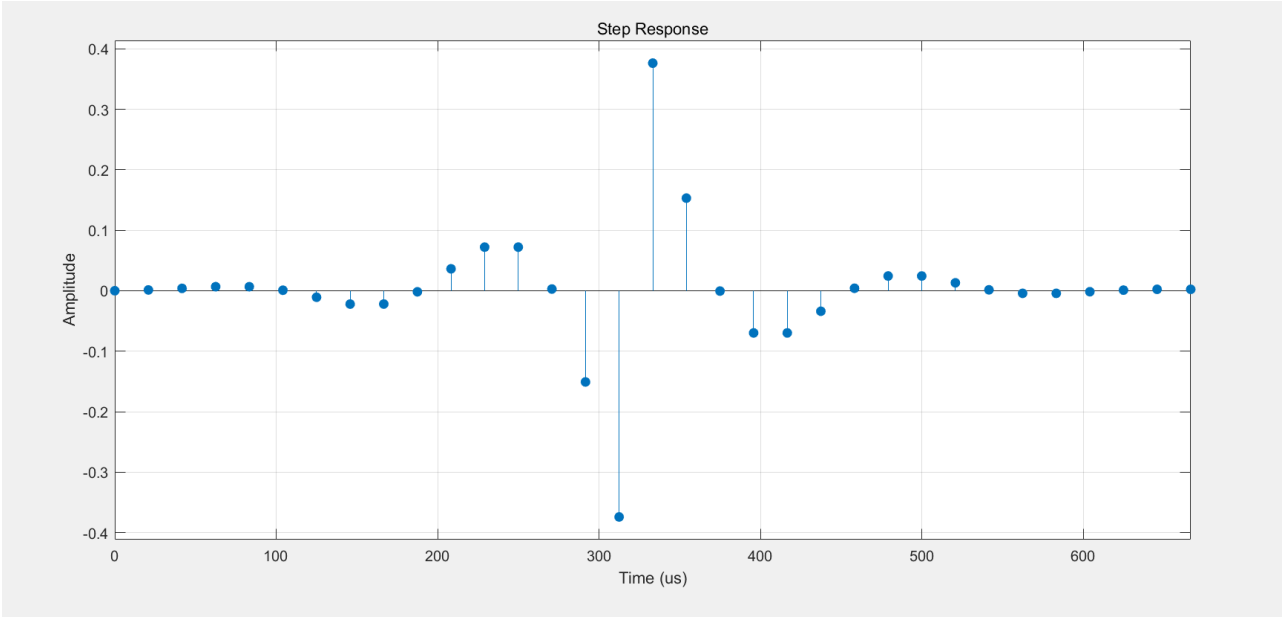


图 9. FIR 滤波器的阶跃响应

然后，设置 PowerQuad 从而在 MCU 上执行滤波过程，以高通任务为例。

```
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.
            */
    }
```

当运行由 PowerQuad 硬件执行的滤波器示例时，结果将显示在 LCD 屏幕上，如 图 10 所示。

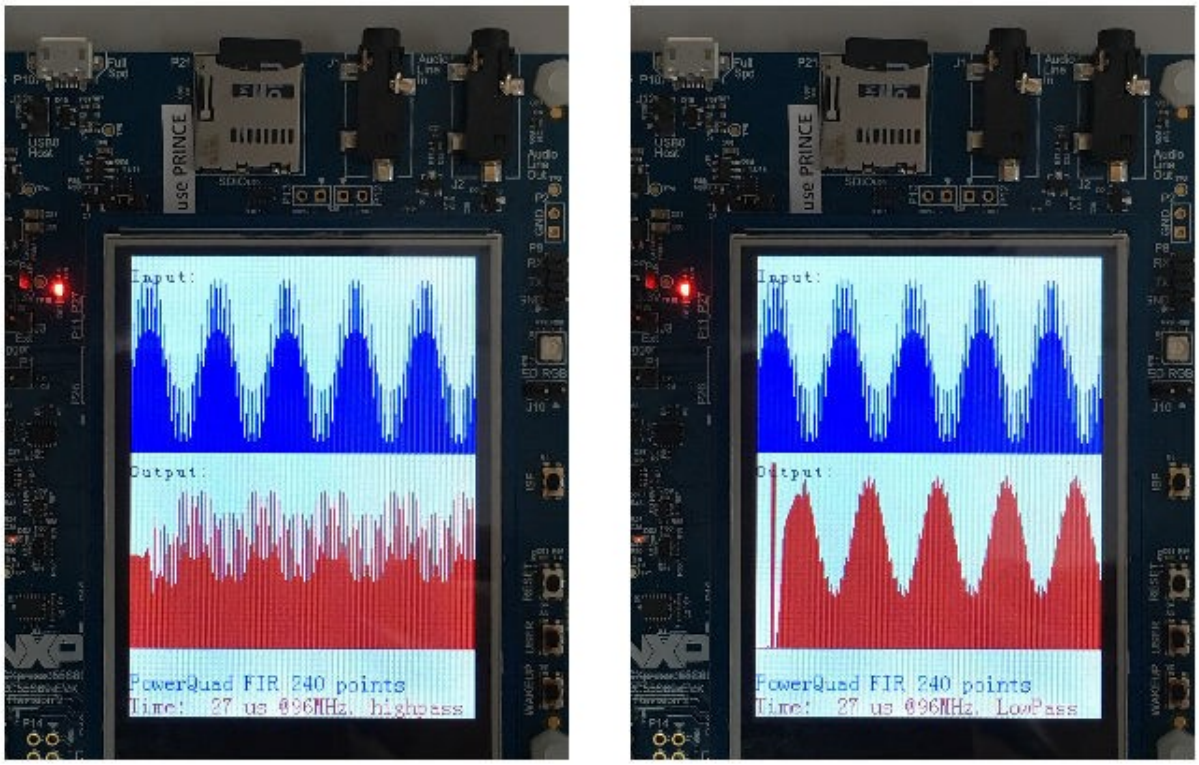


图 10. PowerQuad FIR 高通/低通滤波器

4 PowerQuad 与 Arm CMSIS-DSP 性能对比

最后，在示例工程中，设置了一个页面，用于在 PowerQuad 和 Arm CMSIS-DSP 运行相同任务时进行比较。为了进行公平的比较，在运行 DSP 任务时，Arm CMSIS-DSP 代码在 RAM 中运行，而 PowerQaud 使用专用 RAM（私有），因此它们可以实现最高性能。

图 11 显示了屏幕快照。

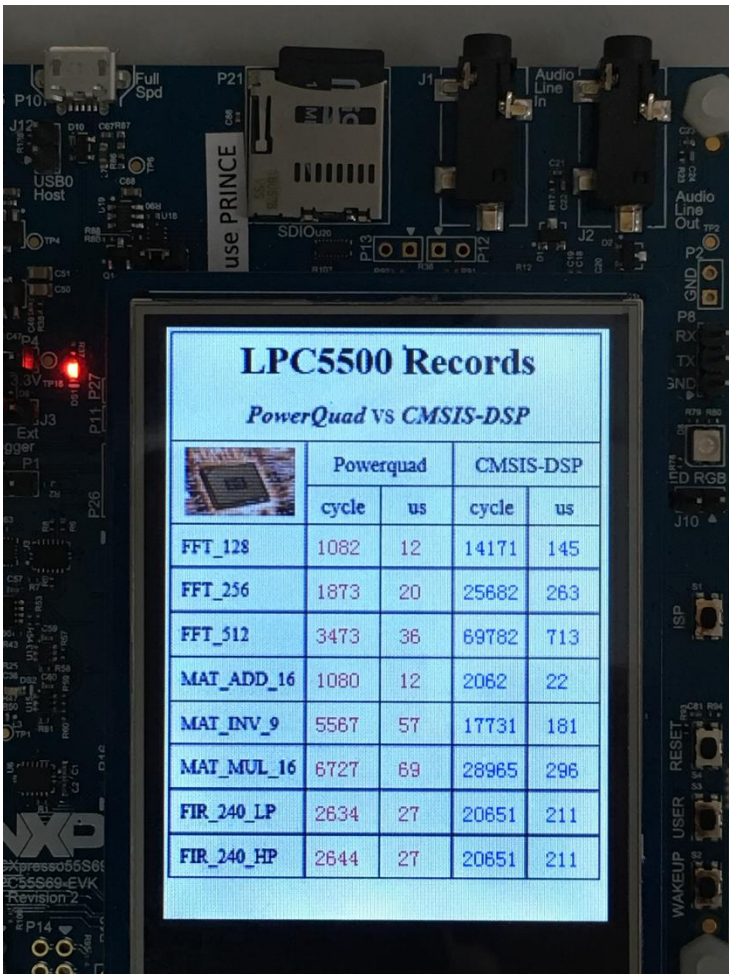
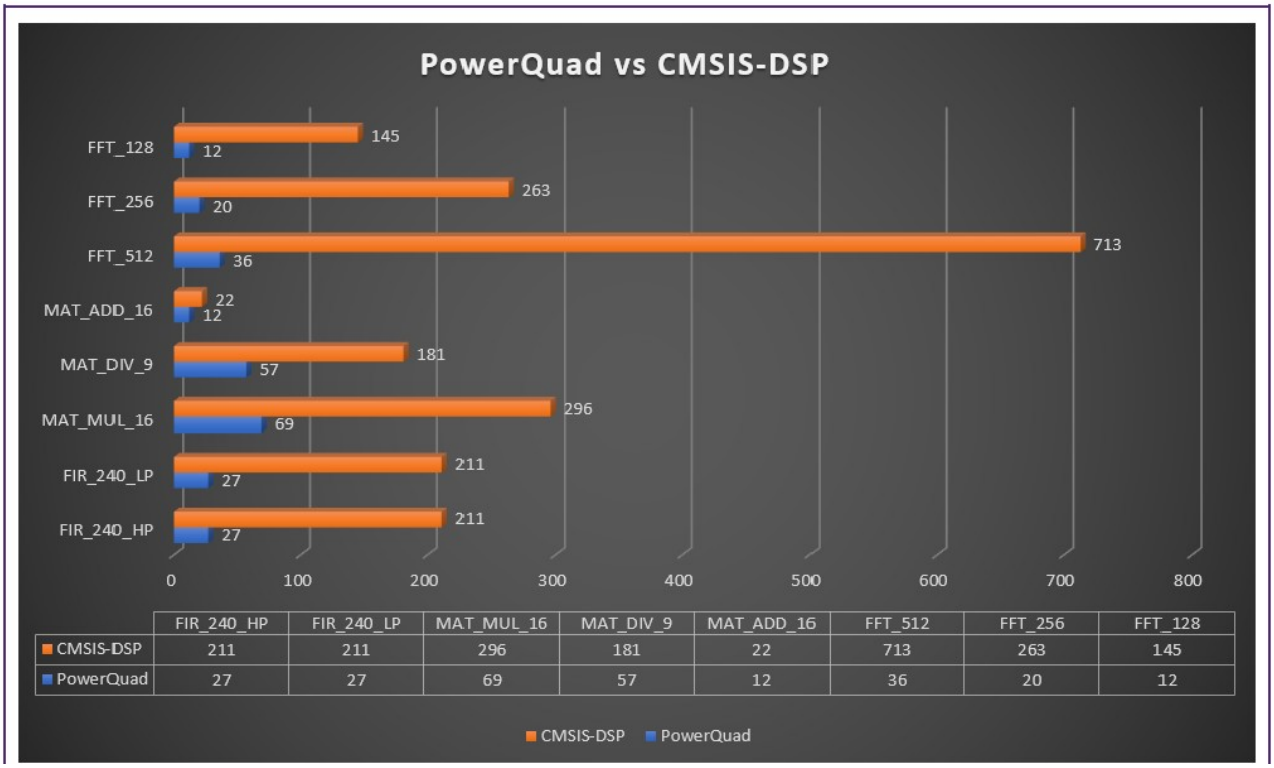


图 11. PowerQuad 与 Arm CMSIS-DSP 的运行时间

图 12 汇总了数据。



	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

图 12. PowerQuad 与 Arm CMSIS-DSP 的运行时间表

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 2019 年 1 月 24 日

Document identifier: AN12282

arm