AN13496

在 LPC5500 上使用 PowerQuad 和 CMSIS-DSP 进行快速傅里叶变换运算

第 0 版 — 2021 年 12 月 31 日 **应用笔记**

目录

1 介绍 1
2 PowerQuad 硬件 FFT 引擎 1
2.1 计算方程 2
2.2 输入和输出细节 3
2.3 专用 RAM 的使用 4
3 在演示项目中测量时间 4
4 演示项目中的计算实例 5
4.1 [输入] 5
4.2 [输出] 5
5 使用 CMSIS-DSP 软件进行快速傅里叶变换 6
5.1 复数快速傅里叶变换. 7
5.2 实数快速傅里叶变换 13
6 使用 PowerQuad 硬件进行快速傅里叶变换 18
6.1 定点复数快速傅里叶变换 18
6.2 定点实数快速傅里叶变换 22
6.3 浮点快速傅里叶变换 26
7 总结和结论 35
8 修订历史 39

1 介绍

快速傅里叶变换（FFT，Fast Frontier Transform）是数字信号处理（DSP）应用中使用最广泛的计算，它可以在时域和频域之间进行转换。当信号的样本数组从时域转换到频域时，会出现一些实用和有趣的属性，这些属性可以用于确定信号的模式。FFT 凭借这个特点，通过对定时采样信号的分析来提取在语音识别、信号检测和其他机器学习等应用中的特征。

Arm® CMSIS-DSP 软件库提供了一组 API，可满足在 Cortex®-M MCU 上进行 FFT 运算的要求，然而 CMSIS-DSP 中的功能是由软件来实现的。虽然该软件优化得很好，但计算时间取决于编译器的优化条件和 CPU 的性能。另外，单纯通过软件进行复杂运算（如 FFT）的计算时间会很长。在实时应用中应考虑这一点。

PowerQuad 硬件模块用于加速一些常规的 DSP 计算任务，包括数学函数、矩阵函数、滤波器函数和变换函数（包括 FFT）。而计算则由 Arm 内核之外的特定硬件执行，该特定硬件运行速度快且能节省 CPU 时间。PowerQuad 是一个简化的 DSP 硬件，具有更低的功耗。它被集成在 Arm 的生态系统中。基于 PowerQuad 的开发是非常友好的。

定点 FFT 和浮点 FFT 在不同领域有各自的实现和应用。定点 FFT 用于处理从硬件传感器模块（如 ADC）捕获的音频、视频和其他数据，且这些情况下的原始直接采样值是定点的。浮点 FFT 在导航系统中以高精度和高分辨率处理经度和纬度。本应用笔记将讲述定点 FFT 和浮点 FFT。

2 PowerQuad 硬件 FFT 引擎

PowerQuad 提供离散傅立叶变换（DFT）和离散余弦变换（DCT）。它们是用一个 Radix-8 蝶形结构的 FFT 引擎实现的。该引擎使用分辨率为 24 位的定点算法。

图 1 显示了该引擎的 Radix-8 蝶形结构。此种方式减少了内存访问的次数，并使用了 PowerQuad 中配备的四个乘法器。

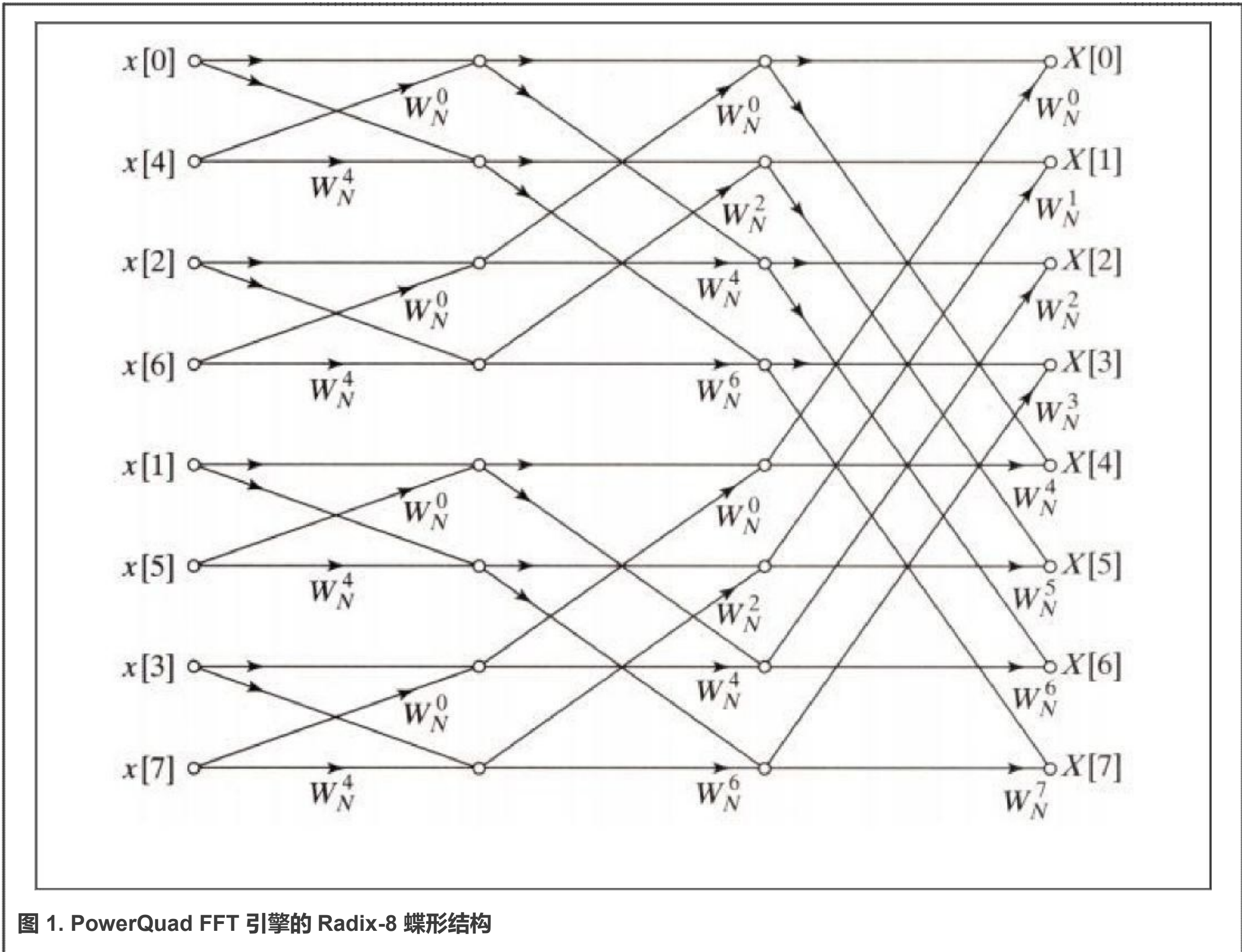

图 1. PowerQuad FFT 引擎的 Radix-8 蝶形结构

2.1 计算方程

离散傅立叶变换将一个含有 N 个复数的序列转换：

```
x0, x1, x2, ..., xN-1
```

为另一个含有 N 个复数的序列：

```
X0, X1, X2, ..., XN-1
```

这个序列的定义是：

它的逆变换由下式给出：

$$X_k = \sum_{n=0}^{N-1}(x_n \cdot e^{-\mathrm{i}\cdot\frac{2\pi}{N}\cdot k\cdot n})$$

$$= \sum_{n=1}^{N-1}(x_n \cdot \left[\cos\left(\frac{2\pi}{N}\cdot k\cdot n\right) - i\cdot\sin\left(\frac{2\pi}{N}\cdot k\cdot n\right)\right])$$

$$x_n = \frac{1}{N}\sum_{n=0}^{N-1}(X_n \cdot e^{\mathrm{i}\cdot\frac{2\pi}{\mathrm{N}}\cdot k\cdot n})$$

在大多数实际应用中，$x_0, x_1, x_2, ..., x_{N-1}$ 是纯实数。离散傅立叶变换（DFT）遵守对称性：

$$X_{N-k} = X_{-k} = X_k^*$$

因此，X_0 和 $X_{N/2}$ 是读取值，N/2-1 指定 DFT 的余数为复数。

注意

PowerQuad 的 FFT 计算引擎支持离散余弦变换（DCT），但它不如 FFT 常用。FFT 计算引擎是以矩阵方式计算的，这种方式更简单。PowerQuad 矩阵计算引擎也支持 DCT。与 FFT 计算引擎相比，使用矩阵计算引擎来计算 DCT 更容易且更灵活。由于 DCT 的用法与 FFT 用法基本相同，所以本应用笔记不详细讲述 DCT。

2.2 输入和输出细节

2.2.1 仅适用于 FFT 引擎的定点数

PowerQuad FFT 引擎使用定点数作为输入和输出，还可以将临时数据保留在 TEMP 区域中。

注意

FFT 引擎会查看输入字的底部 27 位。为了避免饱和，任何预缩放都不得过大。

如果需要浮点数的 FFT，则要将输入的浮点数转换为定点数。启动计算，然后再将输出的定点数转换为浮点数。幸而，PowerQuad 的矩阵引擎提供了矩阵缩放功能，它能通过混合格式计算来加快转换速度。

2.2.2 内存中的输入和输出序列

纯实数函数（以 r 为前缀）和复数函数（以 c 为前缀）要求输入数据序列在内存中按如下形式排列：

- 如果输入序列 $x_0, x_1, ... x_{N-1}$ 是复数形式，N 是数组长度：

  ```
  (X0_real + i * X0_im), (X1_real + i * X1_im), ...(XN-1_real + i * XN-1_im)
  ```

  则内存中的输入数组的组织形式为：

  ```
  {X0_real, X0_im, X1_real, X1_im, ..., XN-1_real, XN-1_im}
  ```

- 如果输入序列 $x_0, x_1, ... x_{N-1}$ 是实数，则内存中的输入数组的组织形式为：

  ```
  {X0, X1, ... XN-1
  ```

  输出序列将以复数数组的形式存储在内存中。实数值输出数据的虚部全为零。

  PowerQuad FFTs/DCTs 支持的数组长度为 N = 16、32、64、128、256、512。

2.2.3 默认硬件预缩放器

PowerQuad FFT 引擎在计算 FFT 之前，会通过默认硬件将输入数据的值缩放 1/N（除以 N）。这些值在离散傅里叶变换（DFT）和逆 DFT 计算期间都不会溢出。如果需要未缩放的结果，则在将输入数据放入 INPUT A 区域之前，要先将其乘以 N，或者为 INPUT A 区域设置一个硬件预缩放器。

逆 FFT 则按 1/N 缩放。按照逆 DFT 公式，这就可以了，不再需要缩放处理。

要替换当前项目中所使用的 CMSIS-DSP 的 FFT API，应手动添加预缩放器以保持输入和输出数据。但如果是新设计的应用，则不需要此步骤。输出之间的比例关系仍然保持不变，这是 FFT 计算中最重要的信息。

下面显示了使用和不使用手动预缩放器的不同结果。

2.3 专用 RAM 的使用

此专用 RAM 是一个专门用于 PowerQuad 的内存区域。PowerQuad 可以独自访问这部分内存，且不会产生任何仲裁延迟，从而尽可能快地加速整个计算过程。PowerQuad 以交错的方式同时访问带有 32 位总线的四个存储区。它可以达到等效的 128 位总线带宽。如果使用专用 RAM，PowerQuad 就可以更快地访问数据。PowerQuad 可以在从 RAM 访问一个操作数时，同时又访问系统的其他部分。这样 PowerQuad 的性能就提高了。

LPC5500 上专用 RAM 的空间为 16 KB，地址在 `0xE000_0000` 和 `0xE000_3FFF`.之间。此专用 RAM 仅支持 32 位寻址。它适用于浮点数（这是 PowerQuad 原本的形式）。此专用 RAM 中的所有地址空间都用于四个内存处理接口，即 INPUTA、INPUT B、TEMP、OUTPUT。当数据进出专用 RAM 时，无法选择内存处理接口的格式。

FFT 是一个特例，因为它的引擎是定点引擎，而所有其他函数原本就是浮点数。FFT 引擎被设计为通过 AHB 操作来作为输入（INPUT A）和最终输出（OUTPUT）。它的内存位于常规内存空间中。专用内存用作 TEMP 内存处理接口的临时存储。启动 FFT 引擎时，允许专用 RAM 进行中间（TEMP）存储。FFT 以定点运行，它将其临时数据以定点形式存储并以定点形式取回。

TEMP 区域仅用于 FFT（的中间计算）和矩阵求逆。对于其他函数，仅用到了内存处理接口 INPUT A、INPUT B 和 OUTPUT。

另一个重要的注意事项是内存处理接口的内存地址的对齐。由于 PowerQuad 一次用 4 个字（128 位）读取输入并写入输出，因此为 PowerQuad 内存处理接口分配的内存地址必须是 4 个字（或 16 字节）对齐的。FFT 是一个特例。对于 TEMP 内存处理接口，它需要与其空间大小匹配。例如，512 个点表示 512 个复数对，那么它必须与 1024 个字对齐。

由于 FFT 是唯一使用专用 RAM 的大操作，所以它是唯一具有如此大的对齐要求的操作。因此，将 `0xE000_0000` 用于其 TEMP 内存处理接口，让硬件 FFT 引擎能够将此空间用于 FFT。

3 在演示项目中测量时间

考虑到函数通常运行速度很快，基于中断的计时方法不适用于本演示实例。

注意

在一些专门进行测量的测试项目中，仍然可以使用基于中断的时间测量方法。该方法通过测量目标函数的多次运行时间，从而获得一次执行的平均时间。

在本文档的演示代码中，选择 SysTick 计时器作为硬件计时器，该代码可以移植到其他 Arm Cortex-M MCU 中。然后直接使用 24 位计数器值进行计时。SysTick 计时器的时钟源由运行在 96 MHz 的 LPC5500 提供，因此最大计时周期可达 174 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)
```

用法是：

```
uint32_t cycles;

TimerCount_Start();
arm_cfft_q31(&instance, inputF32, 0, 1); /* Computing Complex FFT. */
TimerCount_Stop(cycles);

printf("timing cycles: %d", cycles);
```

本文档将对每个功能实例在不同条件下的运行时间进行测量，并汇总测量时间以显示计算性能。

4 演示项目中的计算实例

本文档对所有演示计算实例都使用通用的计算过程。它运行 512 点 FFT 变换，将给定数组转换为预期的输出数组。

4.1 [输入]

输入数组包含长度为 512 的纯实数序列{1, 2, 1, 2, 1, 2, …, 1, 2}。

- 对于实定点数，输入是整数 1 或 2。
- 对于实浮点数，输入是浮点数 1.0f 或 2.0f。
- 对于复定点数，输入是复数（1，0）或（2，0）。
- 对于复浮点数，输入是复数（1.0f，0.0f）或（2.0f，0.0f）。

不同计算实例的输入值是相同的。

4.2 [输出]

输出数组的值全部为零，除了：

- 第 0 个数是 765。
- 第 256 个数是 -256。

此输出结果是有意义的。如原始输入数组所示，输入数的平均值为 1.5，而这个简单开关波形的幅值为 0.5，这代表原始输入可以表示为 1.5-0.5、1.5+0.5、1.5-0.5、1.5+0.5，...。开关周期为 2，频率为 1/2，相位为负，没有其他频率系数。

在频域中，用于 512 点 FFT 变换的步长为 1/512。只有第一项和 1/2 处（第 256 项）非零。第一项是直流因子，第 256 项对应于这个简单开关波形。非零位置的值是幅值：result [0] = 1.5, result [256] = -0.5。

当输出结果时，使用通用数学计算器（如 Matlab）来简化 1/N 的步长。这意味着直接输出是最终结果乘以 N。在本文档的实例中，实际结果为：结果 [0] = 768，结果 [256] = -256。

为了获得结果，还支持使用带有以下脚本的 FreeMat 软件（一种类似于 MabLab 的开源版本的数学计算器，FreeMat）进行计算。

```
--> for (i = 1:512); x(i) = mod(i-1,2) + 1; end    % create the input array in x.
--> y = fft(x)    % run the fft and keep result in y
--> plot([1:1:512], y)    % display the diagram of fft result
```

计算结果显示在终端。

```
y =
    1.0e+002 *
  Columns 1 to 6
  7.6800 +    0.0000i    0    0    0    0          0
  Columns 7 to 12
          0    0    0    0    0          0
  ...
  Columns 253 to 258
          0    0    0    0    -2.5600 +    0.0000i    0
  Columns 259 to 264
          0    0    0    0    0          0
  ...
  Columns 505 to 510
          0    0    0    0    0          0
  Columns 511 to 512
          0    0
```

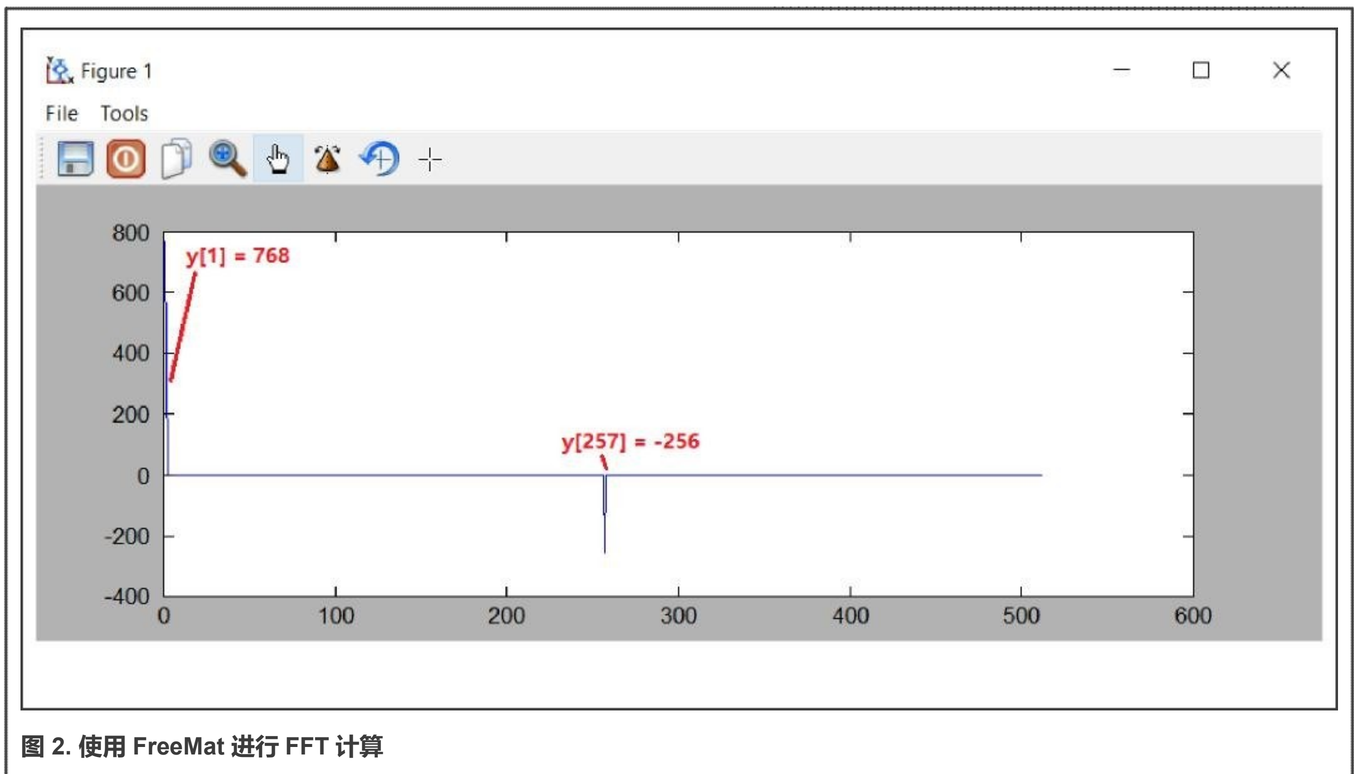

图 2. 使用 FreeMat 进行 FFT 计算

5 使用 CMSIS-DSP 软件进行快速傅里叶变换

本章在讲述 PowerQuad FFT 引擎的用法之前，先介绍 CMSIS-DSP FFT API 的用法，这是具备 MCU 知识的 DSP 开发人员所熟知的。这个优化后的软件实现了 CMSIS-DSP FFT API。

快速傅立叶变换（FFT）是计算离散傅立叶变换（DFT）的有效算法。FFT 的速度可以比 DFT 快几个数量级，尤其是对于较长长度的计算。FFT 有单独的算法可以处理浮点、Q15 和 Q31 等数据类型。

FFT 函数是就地运行的。也就是说，要保存相应的结果，就要使用保存输入数据的数组。输入数据是复数，包含 2*`fftLen` 交错值，如下所示。

```
{real[0], imag[0], real[1], imag[1]...}
```

FFT 的结果是包含在同一数组里的。频域值包含同一种交错。CMSIS-DSP 提供了一组用于计算 FFT 的 API：

- `arm_cfft_f32()`
- `arm_cfft_q31()`
- `arm_cfft_q15()`
- `arm_rfft_fast_f32_init()`和 `arm_rff_fast_f32()`(`arm_rfft_f32()`不再使用)
- `arm_rfft_q31()`
- `arm_rfft_q15()`

有关这些函数的详细信息，请参见 CMSIS-DSP。

下面介绍各种格式下 API 的用法。所有实例都可以在启用了 Arm Armtex-M33 内核、FPU 和 DSP 指令的 LPC5500 平台上良好运行。

5.1 复数快速傅里叶变换

5.1.1 F32 类型的复数 FFT 计算

浮点复数FFT 使用混合基（mixed-radix）算法。根据需要，多个8基阶段与单个2基或4基阶段可以一起执行。该算法支持`[16, 32, 64, ..., 4096]`的长度，并且每个长度都使用不同的旋转因子表。

该函数使用标准的 FFT 定义。在进行前向变换时，输出值可能会增加 `fftLen` 倍。作为计算的一部分，逆变换包括一个 `1/fftLen` 的缩放，这符合教科书中对逆 FFT 的定义。

源文件 `arm_const_structs.h` 提供并定义了预初始化数据结构。该结构包含旋转因子和位反转表。在函数中包含此头文件，并将其中一个常量结构作为参数传递给 `arm_cfft_f32`。例如：

```
arm_cfft_f32(arm_cfft_sR_f32_len64, pSrc, 1, 1)
```

该任务的代码为：

```
/* app_cmsisdsp_cfft_f32.c */
#include "app.h"
extern uint32_t timerCounter;
extern float32_t inputF32[APP_FFT_LEN_512*2];
extern float32_t outputF32[APP_FFT_LEN_512*2];

void App_CmsisDsp_CFFT_F32_Example(void)
{
     uint32_t i;

     PRINTF("%s\r\n",   func  );

     /* input. */
     for (i = 0u; i <APP_FFT_LEN_512; i++)
```

```
     {
inputF32[2*i    ]    =    (1.0f    +    i%2);    /* real part.    */
               inputF32[2*i+1]    =    0;    /*    complex part. */
      }

      TimerCount_Start();
      arm_cfft_f32(&arm_cfft_sR_f32_len512, inputF32, 0, 1);
      TimerCount_Stop(timerCounter);

      /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
      for (i = 0u; i < APP_FFT_LEN_512; i++)
      {
           PRINTF("%4d: %f, %f\r\n", i, inputF32[2*i], inputF32[2*i+1]);
       }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
       PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
       PRINTF("\r\n");
}
/* EOF. */
```

图 3 显示了计算结果。

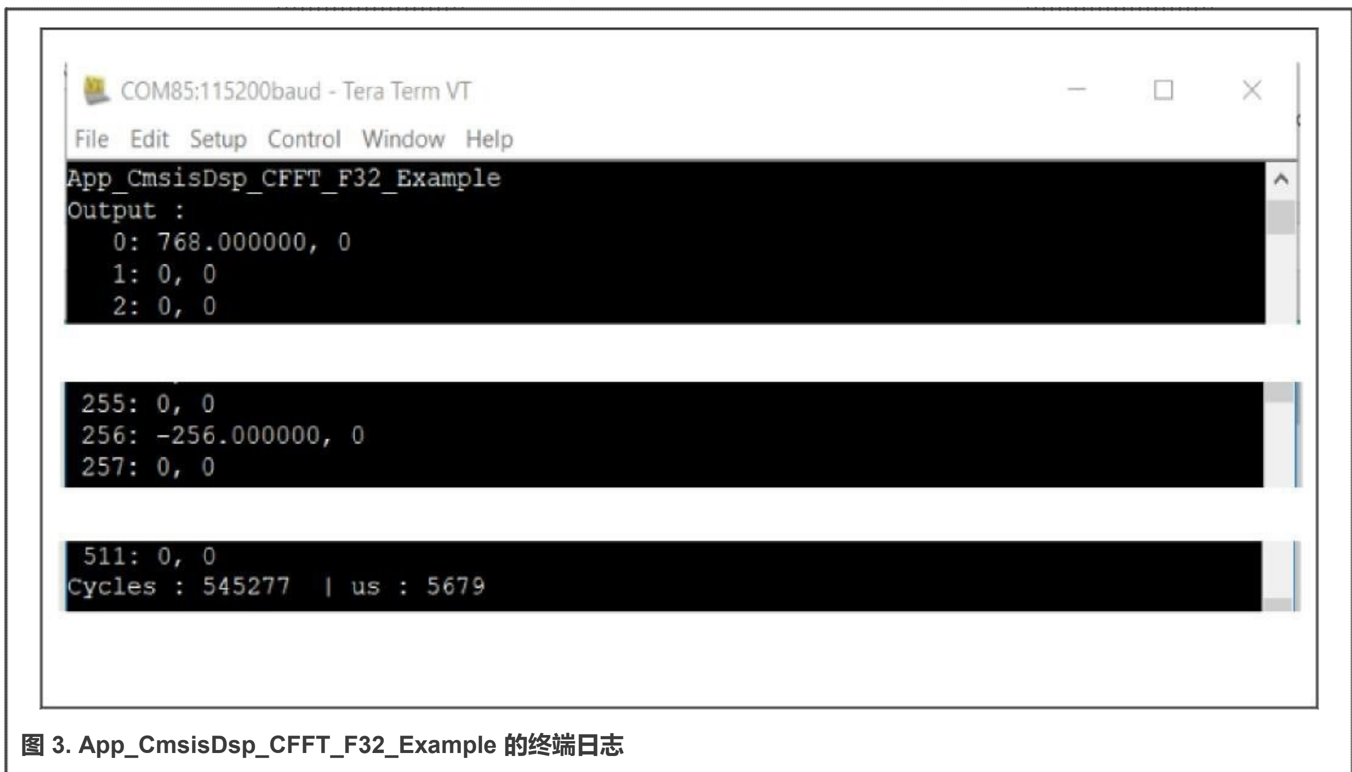

图 3. App_CmsisDsp_CFFT_F32_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看出：

- 计算函数修改了 `inputF32[]` 的内存。输出数覆盖了输入数。输出数使用两个部分分别作为复数的实部和虚部。
- CMSIS-DSP 函数会忽略结果的 `1/fftLen` 缩放。以下所有实例都将使用没有 `1/fftLen` 缩放的结果作为通用目标。
- 此运行时间是在没有编译优化的情况下跑的。表 5 汇总了不同优化条件下的所有计算时间。

5.1.2 Q31 类型的复数 FFT 计算

Q31 的 FFT 版本与浮点版本的 FFT 实现方式不同。由于 Q31 数在（-1，1）的范围内，因此定点数的范围是未知的。但是，在这个实例的应用层面上，它们被用作纯 32 位整数，或者可以将其视为定点格式的 Q0。这种考虑是有道理的。除非整个应用都设计成使用内存中所有特殊格式的定点数，否则 FFT 的输出将作为标准值来完成后续运算。

该任务的代码是：

```
/* app_cmsisdsp_cfft_q31.c */
#include "app.h"

extern uint32_t timerCounter;
extern q31_t    inputQ31[APP_FFT_LEN_512*2];
extern q31_t    outputQ31[APP_FFT_LEN_512*2];

void App_CmsisDsp_CFFT_Q31_Example(void)
{
    uint32_t i;
    PRINTF("%s\r\n",   func  );

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
        inputQ31[2*i ] = APP_FFT_LEN_512 * (1 + i%2); /* real part. */
        inputQ31[2*i+1] = 0; /* complex part. */
    }

    TimerCount_Start();
    arm_cfft_q31(&arm_cfft_sR_q31_len512, inputQ31, 0, 1);
    TimerCount_Stop(timerCounter);

     /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
     PRINTF("Output :\r\n");
     for (i = 0u; i < APP_FFT_LEN_512; i++)
     {
          PRINTF("%4d: %d, %d\r\n", i, inputQ31[2*i], inputQ31[2*i+1]);
      }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
      PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
      PRINTF("\r\n");
}

/* EOF. */
```

图 4 显示了计算结果。

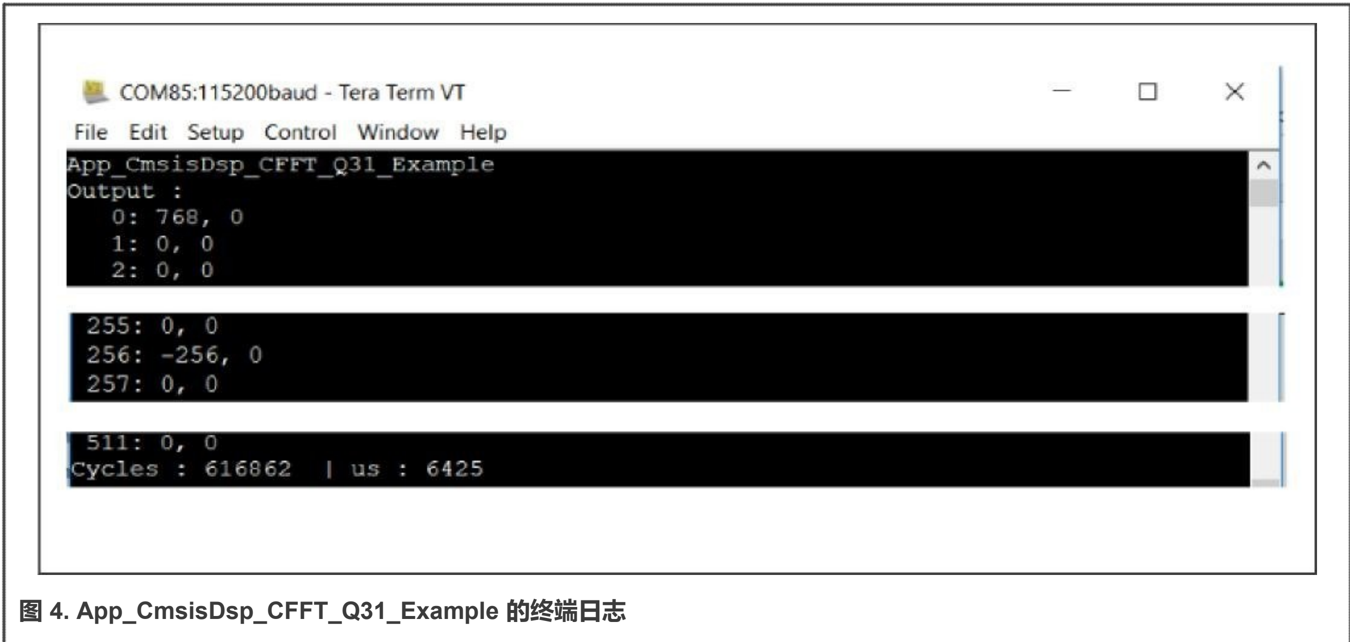

图 4. App_CmsisDsp_CFFT_Q31_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 定点版本的 FFT 在函数中按 `1/fftLen` 比例的进行缩放。这种方式可以保存更多的有效数字，并防止计算过程中的溢出。要实现通用目标，就要在代码中，手动使用一个软件中的预缩放器。

定点 FFT 函数会根据计算长度自动转换输入。为了避免在 CFFT/CIFFT 过程中出现饱和，每个阶段的输入均按比例 2 缩小。输出格式随 FFT 大小而不同。表 1 和 表 2 列出了对于不同 FFT 大小和放大位数的输入和输出格式。

表 1. CMSIS-DSP 中 Q31 CFFT 的输入/输出格式

CFFT 大小	输入格式	输出格式	放大位数
16	1.31	5.27	4
64	1.31	7.25	6
256	1.31	9.23	8
1024	1.31	11.21	10

表 2. CMSIS-DSP 中 Q31 CIFFT 的输入/输出格式

CFFT 大小	输入格式	输出格式	放大位数
16	1.31	5.27	0
64	1.31	7.25	0
256	1.31	9.23	0
1024	1.31	11.21	0

5.1.3 Q15 类型的复数 FFT 计算

CMSIS-DSP 中 Q15 版本的 FFT 虽然消耗的内存和时间更少，但是有效位数较少。它适合处理原始格式为 16 位的数据。其用法与 Q31 版本相同。也可以使用具有适当位移的纯 16 位整数，正如 Q31 版本的实例的操作。

该任务的代码是：

```
/* app_cmsisdsp_cfft_q15.c */
#include "app.h"

extern uint32_t timerCounter;
extern q15_t    inputQ15[APP_FFT_LEN_512*2];
extern q15_t    outputQ15[APP_FFT_LEN_512*2];

void App_CmsisDsp_CFFT_Q15_Example(void)
{
    uint32_t i;

    PRINTF("%s\r\n", __func__);

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
         inputQ15[2*i ] = APP_FFT_LEN_512 * (1 + i%2); /* real part. */
         inputQ15[2*i+1] = 0; /* complex part. */
     }

     TimerCount_Start();
     arm_cfft_q15(&arm_cfft_sR_q15_len512, inputQ15, 0, 1);
     TimerCount_Stop(timerCounter);

     /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
     PRINTF("Output :\r\n");
     for (i = 0u; i < APP_FFT_LEN_512; i++)
     {
           PRINTF("%4d: %d, %d\r\n", i, inputQ15[2*i], inputQ15[2*i+1]);
     }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
     PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
     PRINTF("\r\n");
}

/* EOF. */
```

图 5 显示了计算结果。

图 5. App_CmsisDsp_CFFT_Q15_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- Q15 版本的 FFT 与 Q31 版本一样，在函数中按 `1/fftLen` 比例的进行缩放。要实现通用目标，就要在代码中，手动使用一个软件中的预缩放器。

表 3 和表 4 列出了 Q15 FFT 的输入和输出格式。

表 3. CMSIS-DSP 中 Q15 CFFT 的输入和输出格式

CFFT 大小	输入格式	输出格式	放大位数
16	1.15	5.11	4
64	1.15	7.9	6
256	1.15	9.7	8
1024	1.151	11.5	10

表 4. CMSIS-DSP 中 Q15 CIFFT 的输入和输出格式

CFFT 大小	输入格式	输出格式	放大位数
16	1.15	5.11	0
64	1.15	7.9	0
256	1.15	9.8	0
1024	1.15	11.5	0

5.2 实数快速傅里叶变换

实数 N 点序列的快速傅里叶变换在频域中是偶对称的。数据的后半部分等于频率翻转的前半部分的共轭。其中仅有 N/2 个复数独立地表示结果。它们被打包成交替的实部和虚部放入输出数组中。

```
X = { real[0], imag[0], real[1],imag[1], real[2], imag[2] ... real[(N/2)-1], imag[(N/2)-1}
```

第一个复数（`real[0], imag[0]`）是纯实数。`real[0]`代表 DC 偏移，而 `imag[0]`一定为 0。使用 `imag[0]`的位置来存放 `real[N/2]`，这是另一个纯实数。（`real[1], imag[1]`）则是基频，（`real[2], imag[2]`）是第一谐波，依此类推。

实数 FFT 函数以这种方式打包频域数据。前向变换也以这种形式输出数据。逆变换则以这种形式输入数据。然后该函数执行所需的位反转，以便输入和输出数据按正常顺序排列。该函数支持长度为[`32, 64, 128, ..., 4096`]的采样值。

CMSIS DSP 库包含用于计算实数序列 FFT 的专用算法。FFT 是在复数上定义的，但在许多应用中，输入数据是实数。实数 FFT 算法利用了 FFT 的对称性，在速度上优于同长度的复数算法。

快速 RFFT 算法依赖于混合基数 CFFT，从而节省处理器的使用。图 6 显示了一个长度为 N 的实数序列的前向 FFT 运算的步骤。

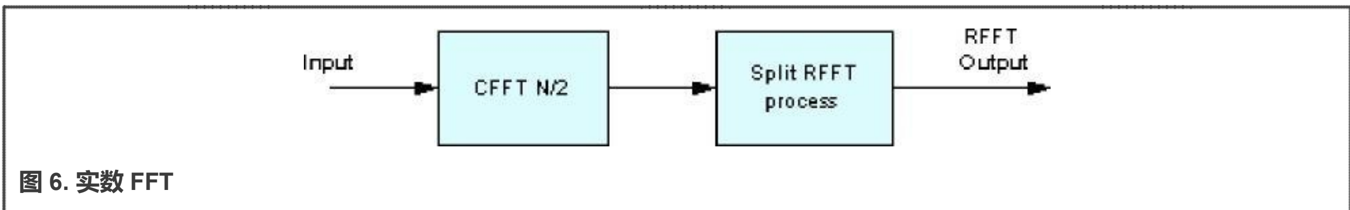

图 6. 实数 FFT

实数序列会被当作一个复数序列进行 CFFT 运算。随后，某个处理阶段会对数据进行重构以得到复数形式的一半频谱。除了包含两个实数 `X[0]`和 `X[N/2]`的第一个复数之外，所有数据都是复数。换句话说，第一个复数样本打包进了两个实数。

逆 RFFT 的输入要与前向 RFFT 的输出保持相同的格式。第一处理阶段会对数据进行预处理，以便稍后进行逆 CFFT 运算。

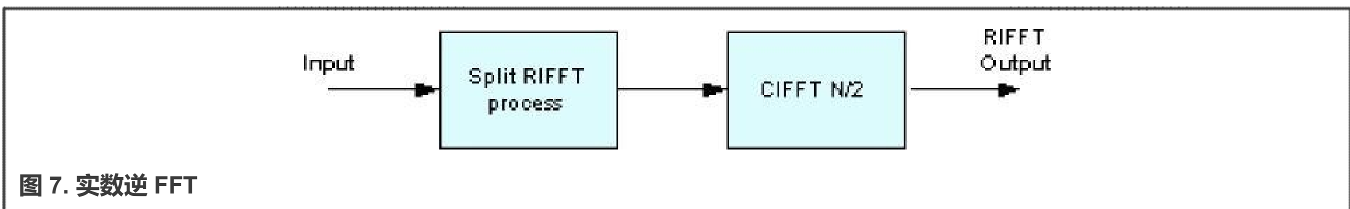

图 7. 实数逆 FFT

进行 N 点实数 FFT 运算的总结：

- 输入数组的长度为 N，有 N 个实数。
- 对应于频谱的前半部分，输出数组的长度也为 N，有 N/2 个复数。数据的后半部分则等于频率翻转的前半部分的共轭。
- 输出数组的第一个复数包含两个实数，即 `real[0]` 和 `real[N/2]`。

5.2.1 F32 类型的实数 FFT 计算

CMSIS-DSP 提供了一个新的快速 API 来计算实浮点数 FFT，它可以取代旧的 API。当前，只有 `arm_rfft_fast_init_f32()/arm_rfft_fast_f32` 的 API 是推荐的用于计算的 API。输入和输出内存的位置与复数 FFT 函数相同。输入和输出内存在用户代码中是分开的。输出数据的方式略有不同。

该任务的代码是：

```
/* app_cmsisdsp_rfft_fast_f32.c */
#include "app.h"

extern uint32_t timerCounter;
extern float32_t inputF32[APP_FFT_LEN_512*2];
extern float32_t outputF32[APP_FFT_LEN_512*2];

void App_CmsisDsp_RFFT_Fast_F32_Example(void)
{
   uint32_t i;
   arm_rfft_fast_instance_f32 rfft_fast_instance;

   PRINTF("%s\r\n",   func  );

   /* input. */
   for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
        inputF32[i] = (1.0f + i%2); /* only real part. */
   }

   arm_rfft_fast_init_f32(&rfft_fast_instance, APP_FFT_LEN_512);

   TimerCount_Start();
   arm_rfft_fast_f32(&rfft_fast_instance, inputF32, outputF32, 0);
   TimerCount_Stop(timerCounter);

   /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
   PRINTF("Output :\r\n");
   for (i = 0u; i < APP_FFT_LEN_512/2; i++)
   {
        PRINTF("%4d: %f, %f\r\n", i, outputF32[2*i], outputF32[2*i+1]);
    }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
    PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
    PRINTF("\r\n");
}

/* EOF. */
```

图 8 显示了计算结果。

图 8. App_CmsisDsp_RFFT_Fast_F32_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 输出数是复数，但是只有输入项长度的一半（输入在 512 个内存项中有 512 个实数，而输出在 512 个内存项中有 256 个复数）。输出数组的第一项与其他项不同。
  - — 第一个复数项（`real[0]`, `imag[0]`）全部是实数。
  - — `real[0]`代表直流偏置。
  - — `imag[0]`一定是 0。
  - — （`real[1]`, `imag[1]`）是基频，（`real[2]`, `imag[2]`）是第一谐波，以此类推。

5.2.2 Q31 类型的实数 FFT 计算

Q31 的实数 FFT 与快速的浮点版本不同。它使用类似于复数 FFT 函数的旧格式。输入数组包含所有实数。输出数组是长度不缩减的复数。这意味着输出数组的内存是输入数组内存的两倍。

该任务的代码是：

```
/* app_cmsisdsp_rfft_q31.c */
#include "app.h"

extern uint32_t timerCounter;
extern q31_t    inputQ31[APP_FFT_LEN_512*2];
extern q31_t    outputQ31[APP_FFT_LEN_512*2];

void App_CmsisDsp_RFFT_Q31_Example(void)
{
   uint32_t i;

   PRINTF("%s\r\n",   func  );

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
         inputQ31[i] = APP_FFT_LEN_512 * (1 + i%2); /* only real part. */
     }
```

```
    TimerCount_Start();
    arm_rfft_q31(arm_rfft_sR_q31_len512, inputQ31, outputQ31);
    TimerCount_Stop(timerCounter);

    /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
    PRINTF("Output :\r\n");
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
        PRINTF("%4d: %d, %d\r\n", i, outputQ31[2*i], outputQ31[2*i+1]);
     }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
     PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
     PRINTF("\r\n");
}

/* EOF. */
```

图 9 显示了计算结果。

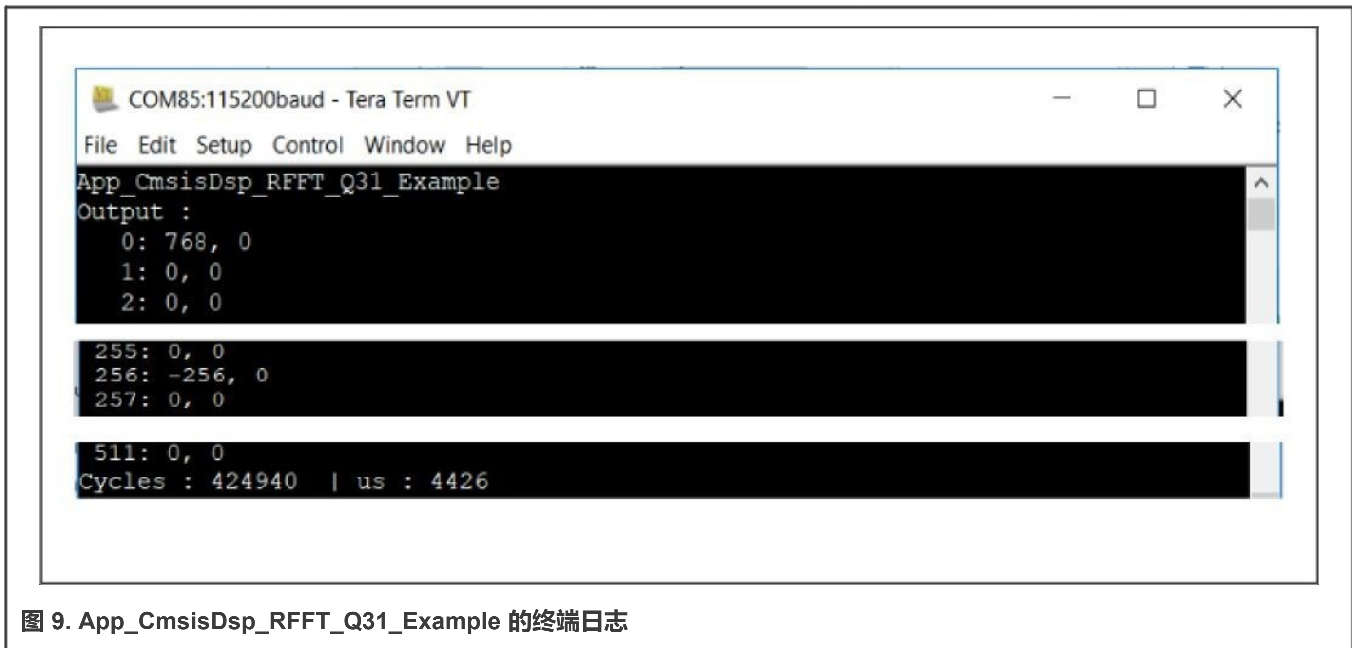

图 9. App_CmsisDsp_RFFT_Q31_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 预缩放器用来达到通用目标。
- 对于 512 个实数，可用输入数组的长度为 512。对于 512 个复数，可用输出数组的长度为 1024。
- 输出数组的格式与传统复数函数的格式相同。第一个数并不像快速浮点实数 FFT 那样特殊。

5.2.3 Q15 类型的实数 FFT 计算

Q15 版本的实数 FFT 运算继承了 Q31 版本的特点。

该任务的代码是：

```
/* app_cmsisdsp_rfft_q15.c */
#include "app.h"
```

```
extern uint32_t timerCounter;
extern q15_t    inputQ15[APP_FFT_LEN_512*2];
extern q15_t    outputQ15[APP_FFT_LEN_512*2];

void App_CmsisDsp_RFFT_Q15_Example(void)
{
    uint32_t i;
    PRINTF("%s\r\n",   func  );

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
         inputQ15[i] = APP_FFT_LEN_512 * (1 + i%2); /* only real part. */
    }

     TimerCount_Start();
     arm_rfft_q15(&arm_rfft_sR_q15_len512, inputQ15, outputQ15);
     TimerCount_Stop(timerCounter);

      /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
      PRINTF("Output :\r\n");
      for (i = 0u; i < APP_FFT_LEN_512; i++)
      {
           PRINTF("%4d: %d, %d\r\n", i, outputQ15[2*i], outputQ15[2*i+1]);
       }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
       PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
       PRINTF("\r\n");
}

/* EOF. */
```

图 10 显示了计算结果。

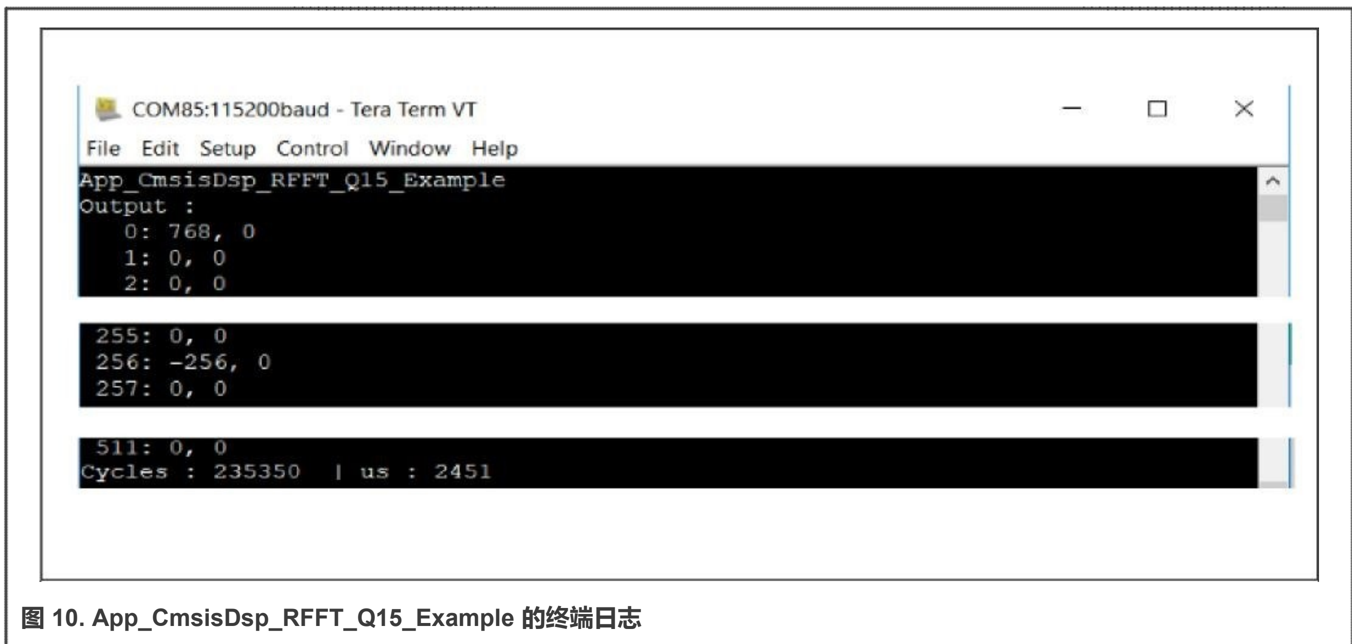

图 10. App_CmsisDsp_RFFT_Q15_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 它的代码和终端日志和 Q31 版本相同。
- 它的运行速度稍快于 Q31 版本。

6 使用 PowerQuad 硬件进行快速傅里叶变换

CMSIS-DSP API 的纯软件实现受到如下限制：

- Arm 内核的架构（狭窄的内存总线）
- 编译器的性能（不同级别的优化条件）

PowerQuad 的计算引擎（包括 FFT 引擎）是由硬件实现和优化的。与使用 CMSIS-DSP 相比，PowerQuad 硬件可以节省大量 CPU 负载和代码量，并显著提高性能。PowerQuad 被集成为协处理器，可与 Arm 内核并行运行，以满足实时系统的要求。

恩智浦 MCUXpresso SDK 软件库支持 PowerQuad 模块。在 PowerQuad 驱动程序中，有一组用于进行 FFT 运算的 API：

- `PQ_TransformCFFT()`
- `PQ_TransformRFFT()`
- `PQ_SetConfig ()`用于设置各种定点格式。

PowerQuad 硬件不支持浮点 FFT 运算。要解锁此功能，请使用基于现有 PowerQuad 硬件的软件解决方案。该解决方案涵盖了 CMSIS-DSP FFT API 适用的相同领域。

下面将讨论 API 的用法。

6.1 定点复数快速傅里叶变换

PowerQuad FFT 引擎硬件仅支持定点 FFT 运算，因此可直接在 PowerQuad 硬件上处理定点 FFT 任务。

6.1.1 Q31 类型的复数 FFT 计算

在以前的 CMSIS-DSP 实例中，为了实现通用目标输出，会使用软件预缩放器用于处理输入数据。PowerQuad 硬件提供了一个具有硬件预缩放器设置的新选项。输入数据和输出数据都有其对应的硬件预缩放器设置。在 512 点 FFT 的情况下，预缩放器数应为 512，`pq_cfg.inputAPrescale` 的相应设置值为 9。输入值左移 9 位作为与 512 相乘。

配置 PowerQuad 硬件的输入和输出格式时，请注意：

- FFT 引擎使用 Input A、Temp 和 Output 内存处理接口。
- 硬件仅支持定点 FFT。
- `pq_cfg.inputAFormat`、`pq_cfg.tmpFormat` 和 `pq_cfg.outputFormat` 中的这些内存处理接口的格式设置是定点的，例如 `kPQ_32Bit` 或 `kPQ_16Bit`。在这个实例中，它们是 `kPQ_32Bit`。
- 对于 FFT 引擎，可忽略输出存储器处理接口的设置。
- 输入和输出数组必须是 32 位字。

复数 FFT 的数据输入数组由实部和虚部组成。每个部分在内存中占用一个 32 位字。输出数据是复数。

临时内存处理接口使用从 `0xE000_0000` 开始的专用 RAM，以在计算过程中保存中间数据。对于 512 点 FFT，要使用 1 K 个 32 位字保存 512 个复数，就要在专用 RAM 中保留 4 KB 的内存空间。

在这个实例中，关键函数是 `PQ_TransformRFFT()`。使用 Q31 数据作为输入和输出，输入数是复数。

该任务的代码是：

```
/* app_powerquad_cfft_q31.c */
#include "app.h"

extern uint32_t timerCounter;
extern q31_t    inputQ31[APP_FFT_LEN_512*2];
extern q31_t    outputQ31[APP_FFT_LEN_512*2];

void App_PowerQuad_CFFT_Q31_Example(void)
{
    uint32_t i;

    PRINTF("%s\r\n",   func  );

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512;
    i++) {

#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
        inputQ31[2*i ] = (1 + i%2); /* real part. */
#else
        inputQ31[2*i ] = APP_FFT_LEN_512 * (1 + i%2); /* real part. */
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
        inputQ31[2*i+1] = 0; /* complex part. */
      }
      memset(outputQ31, 0, sizeof(outputQ31)); /* clear output. */

      /* computing by PowerQuad hardware. */
      {
          pq_config_t pq_cfg;

          PQ_Init(POWERQUAD); /* initialize the PowerQuad hardware. */

          pq_cfg.inputAFormat = kPQ_32Bit;
#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
          pq_cfg.inputAPrescale = 9; /* 2 ^9 for 512 len of input. */
#else
          pq_cfg.inputAPrescale = 0;
          #endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
          pq_cfg.inputBFormat =
          kPQ_32Bit;
          pq_cfg.inputBPrescale = 0;
          pq_cfg.tmpFormat = kPQ_32Bit;
          pq_cfg.tmpPrescale = 0;
          pq_cfg.outputFormat =
          kPQ_32Bit;
          pq_cfg.outputPrescale = 0;
          pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
          pq_cfg.machineFormat = kPQ_32Bit;
          PQ_SetConfig(POWERQUAD, &pq_cfg);

          TimerCount_Start();
          PQ_TransformCFFT(POWERQUAD, APP_FFT_LEN_512, inputQ31,
      }   outputQ31); PQ_WaitDone(POWERQUAD);
          TimerCount_Stop(timerCounter);

          /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
          PRINTF("Output :\r\n");
          for (i = 0u; i < APP_FFT_LEN_512; i++)
```

```
        {
            PRINTF("%4d: %d, %d\r\n", i, outputQ31[2*i], outputQ31[2*i+1]);
        }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
        PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
        PRINTF("\r\n");
}

/* EOF. */
```

图 11 显示了结果。

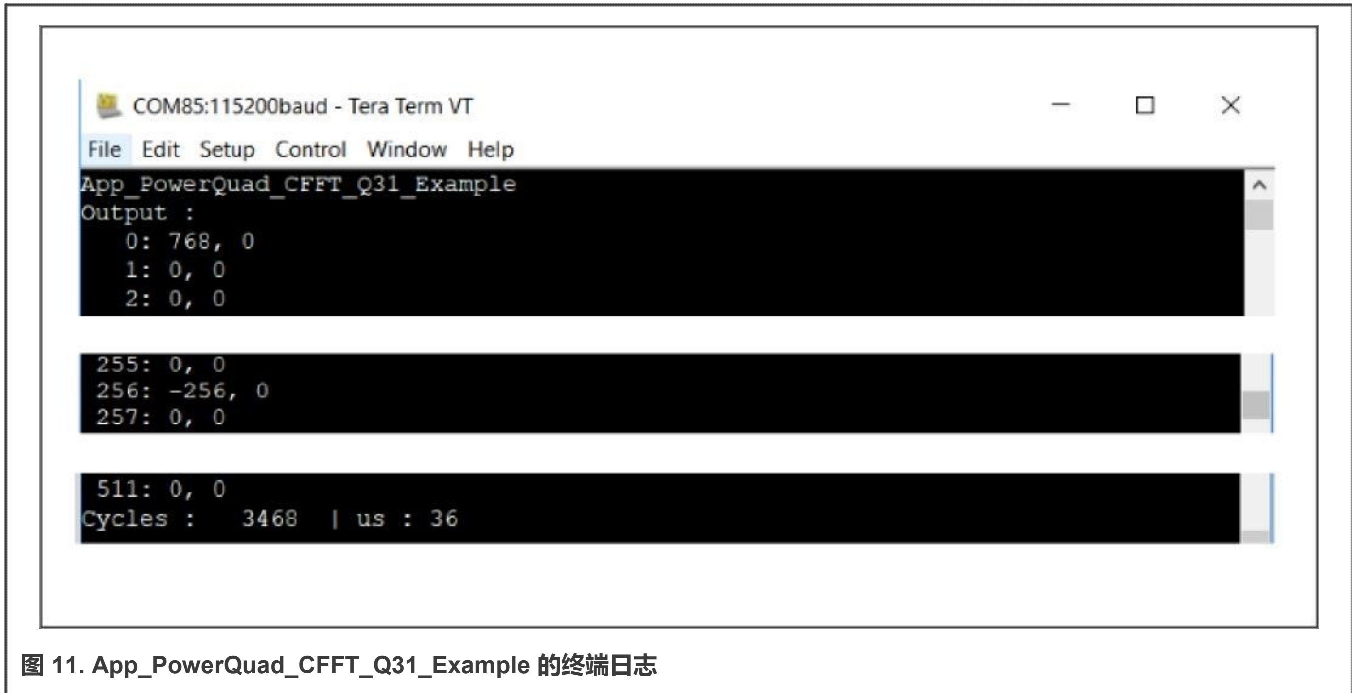

图 11. App_PowerQuad_CFFT_Q31_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 硬件预缩放器发挥了效用，就像软件缩放器一样。
- PowerQuad 硬件达到了预期结果（通用目标）。
- 它确实比 CMSIS-DSP 复数 Q31 定点 FFT 函数更快。

输出定点数的预缩放器可以复用 CMSIS-DSP 定点 FFT 输出的数据表。

6.1.2 Q15 类型的复数 FFT 计算

对于 PowerQuad FFT 引擎，复数 Q15 任务与复数 Q31 任务是相同的，不同之处在于：

- `pq_cfg.inputAFormat`、`pq_cfg.tmpFormat` 和 `pq_cfg.outputFormat` 的数据格式设置为 `kPQ_16Bit`。

该任务的代码是：

```
/* app_powerquad_cfft_q15.c */
#include "app.h"

extern uint32_t timerCounter;
extern q15_t    inputQ15[APP_FFT_LEN_512*2];
extern q15_t    outputQ15[APP_FFT_LEN_512*2];

void App_PowerQuad_CFFT_Q15_Example(void)
```

```
{
    uint16_t i;

    PRINTF("%s\r\n", __func__);

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {

#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
        inputQ15[2*i ] = (1 + i%2); /* real part. */
#else
        inputQ15[2*i ] = APP_FFT_LEN_512 * (1 + i%2); /* real part. */
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
        inputQ15[2*i+1] = 0; /* complex part. */
     }
     memset(outputQ15, 0, sizeof(outputQ15)); /* clear output. */

     /* computing by PowerQuad hardware. */
    {
         pq_config_t pq_cfg;

         PQ_Init(POWERQUAD); /* initialize the PowerQuad hardware. */

         pq_cfg.inputAFormat = kPQ_16Bit; /* for q15_t. */
#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
         pq_cfg.inputAPrescale = 9; /* 2 ^9 for 512 len of input. */
#else
         pq_cfg.inputAPrescale = 0;
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
         pq_cfg.inputBFormat = kPQ_16Bit; /* no use. for q15_t. */
         pq_cfg.inputBPrescale = 0;
         pq_cfg.tmpFormat = kPQ_16Bit; /* for q15_t. */
         pq_cfg.tmpPrescale = 0;
         pq_cfg.outputFormat = kPQ_16Bit; /* for q15_t. */
         pq_cfg.outputPrescale = 0;
         pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
         pq_cfg.machineFormat = kPQ_32Bit; /* even q15_t, they are used as 32-bit internally. */
         PQ_SetConfig(POWERQUAD, &pq_cfg);

         TimerCount_Start();
         PQ_TransformCFFT(POWERQUAD, APP_FFT_LEN_512, inputQ15, outputQ15);
         PQ_WaitDone(POWERQUAD);
         TimerCount_Stop(timerCounter);
    }
    /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
    PRINTF("Output :\r\n");
    for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
        PRINTF("%4d: %d, %d\r\n", i, outputQ15[2*i], outputQ15[2*i+1]);
    }
 #endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
    PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
    PRINTF("\r\n");
}

/* EOF. */
```

图 12 显示了结果。

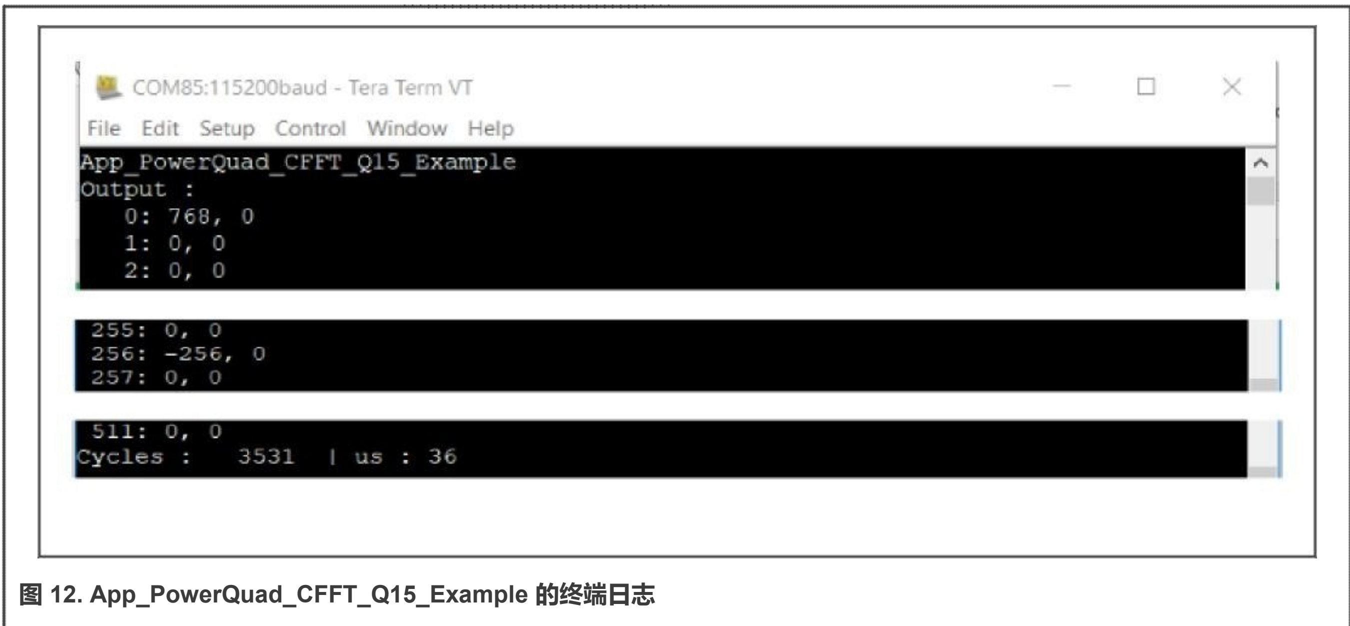

图 12. App_PowerQuad_CFFT_Q15_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 硬件预缩放器也生效了。
- PowerQuad 硬件达到了预期结果（通用目标）。
- Q15 类型的复数 FFT 不比 Q31 的复数 FFT 更快。在实际情况下，它的运行速度甚至更慢。位数更少并不会减少 PowerQuad 硬件的工作负荷。

6.2 定点实数快速傅里叶变换

PowerQuad 硬件执行的纯实数 FFT 将虚部打包，仅将实数部分保留在输入数组中。它比复数的 FFT 节省了一半的内存长度。PowerQuad 硬件可以识别这种方式。但是，PowerQuad 将输出保持为复数（CMSIS-DSP API 也使用相同的方式）。

6.2.1 Q31 类型的实数 FFT 计算

关键函数是 `PQ_TransformRFFT()`，使用 Q31 数据作为输入和输出。输入数据是纯实数。

该任务的代码是：

```
/* app_powerquad_rfft_q31.c */
#include "app.h"

extern uint32_t timerCounter;
extern q31_t    inputQ31[APP_FFT_LEN_512*2];
extern q31_t    outputQ31[APP_FFT_LEN_512*2];

void App_PowerQuad_RFFT_Q31_Example(void)
{
    uint32_t i;

    PRINTF("%s\r\n",   func  );

    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
```

```
#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
        inputQ31[i ] = (1 + i%2); /* only real part. */
#else
        inputQ31[i ] = APP_FFT_LEN_512 * (1 + i%2); /* real part. */
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
    }
    memset(outputQ31, 0, sizeof(outputQ31)); /* clear output. */

    /* computing by PowerQuad hardware. */
    {
         pq_config_t pq_cfg;
         PQ_Init(POWERQUAD); /* initialize the PowerQuad hardware. */
         pq_cfg.inputAFormat = kPQ_32Bit;
#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
         pq_cfg.inputAPrescale = 9; /* 2 ^9 for 512 len of input. */
#else
         pq_cfg.inputAPrescale = 0;
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
         //pq_cfg.inputBFormat = kPQ_32Bit; // no use.
         //pq_cfg.inputBPrescale = 0;
         pq_cfg.tmpFormat = kPQ_32Bit;
         pq_cfg.tmpPrescale = 0;
         pq_cfg.outputFormat = kPQ_32Bit;
         pq_cfg.outputPrescale = 0;
         pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
         pq_cfg.machineFormat = kPQ_32Bit;
         PQ_SetConfig(POWERQUAD, &pq_cfg);

         TimerCount_Start();
         PQ_TransformRFFT(POWERQUAD, APP_FFT_LEN_512, inputQ31, outputQ31);
         PQ_WaitDone(POWERQUAD);
         TimerCount_Stop(timerCounter);
    }

    /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
     PRINTF("Output :\r\n");
     for (i = 0u; i < APP_FFT_LEN_512; i++)
     {
          PRINTF("%4d: %d, %d\r\n", i, outputQ31[2*i], outputQ31[2*i+1]);
      }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
      PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
      PRINTF("\r\n");
}

/* EOF. */
```

图 13 显示了结果。

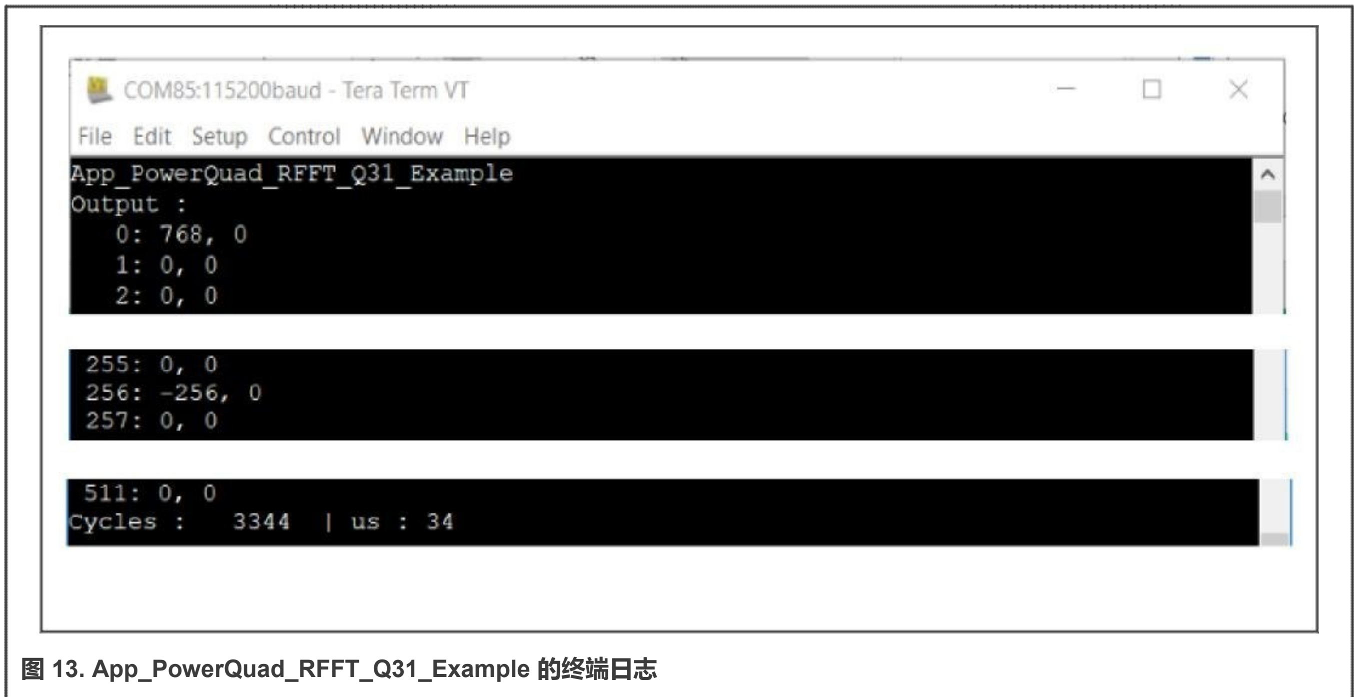

图 13. App_PowerQuad_RFFT_Q31_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 硬件预缩放器也生效了。
- PowerQuad 硬件达到了预期结果（通用目标）。
- 由于减少了内存操作，因此它稍快于 Q31 的复数 FFT。
- 输出数的长度不会像 CMSIS-DSP 函数那样减半。对于复数 FFT 计算，不需要特殊的格式。

6.2.2 Q15 类型的实数 FFT 计算

对于 PowerQuad FFT 引擎，读取的 Q15 任务与实际的 Q31 任务几乎相同，区别在于：

- `pq_cfg.inputAFormat`、`pq_cfg.tmpFormat` 和 `pq_cfg.outputFormat` 的数据结构应设置为 `kPQ_16Bit`。

该任务的代码是：

```
/* app_powerquad_rfft_q15.c */
#include "app.h"

extern uint32_t timerCounter;
extern q15_t    inputQ15[APP_FFT_LEN_512*2];
extern q15_t    outputQ15[APP_FFT_LEN_512*2];

void App_PowerQuad_RFFT_Q15_Example(void)
{
   uint16_t i;

   PRINTF("%s\r\n",   func  );

   /* input. */
   for (i = 0u; i < APP_FFT_LEN_512; i++)
   {

#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
        inputQ15[i ] = (1 + i%2); /* only real part. */
```

```
#else
        inputQ15[i ] = APP_FFT_LEN_512 * (1 + i%2); /* only real part. */
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
    }
    memset(outputQ15, 0, sizeof(outputQ15)); /* clear output. */

    /* computing by PowerQuad hardware. */
   {

         pq_config_t pq_cfg;

         PQ_Init(POWERQUAD); /* initialize the PowerQuad hardware. */

         pq_cfg.inputAFormat = kPQ_16Bit; /* for q15_t. */
#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
         pq_cfg.inputAPrescale = 9; /* 2 ^9 for 512 len of input. */
#else
         pq_cfg.inputAPrescale = 0;
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
         pq_cfg.inputBFormat = kPQ_16Bit; /* no use, for q15_t. */
         pq_cfg.inputBPrescale = 0;
         pq_cfg.tmpFormat = kPQ_16Bit; /* for q15_t. */
         pq_cfg.tmpPrescale = 0;
         pq_cfg.outputFormat = kPQ_16Bit; /* for q15_t. */
         pq_cfg.outputPrescale = 0;
         pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
         pq_cfg.machineFormat = kPQ_32Bit; /* even q15_t, they are used as 32-bit internally. */
         PQ_SetConfig(POWERQUAD, &pq_cfg);

         TimerCount_Start();
         PQ_TransformRFFT(POWERQUAD, APP_FFT_LEN_512, inputQ15, outputQ15);
         PQ_WaitDone(POWERQUAD);
         TimerCount_Stop(timerCounter);
     }

     /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) & (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
     PRINTF("Output :\r\n");&
     for (i = 0u; i < APP_FFT_LEN_512; i++)
     {
          PRINTF("%4d: %d, %d\r\n", i, outputQ15[2*i], outputQ15[2*i+1]);
     }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
     PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
     PRINTF("\r\n");
}

/* EOF. */
```

图 14 显示了结果。

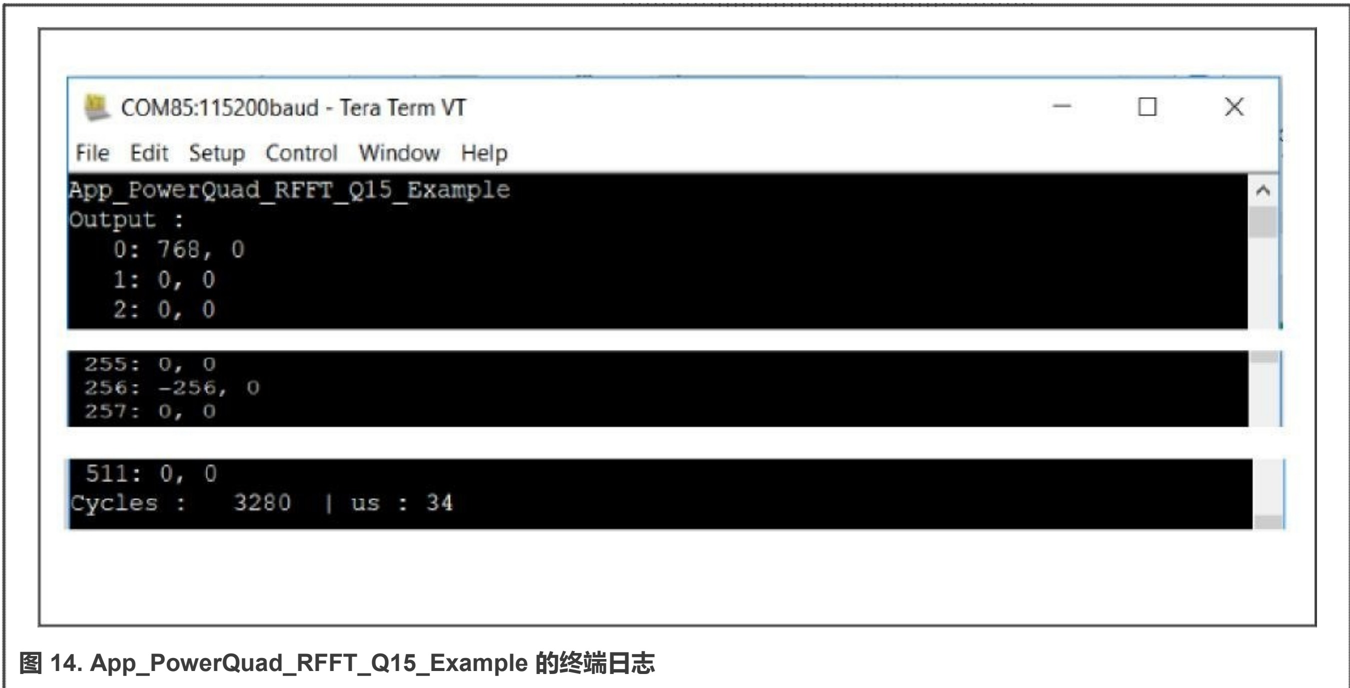

图 14. App_PowerQuad_RFFT_Q15_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 硬件预缩放器也生效了。
- PowerQuad 硬件达到了预期结果（通用目标）。
- 由于减少了内存操作，因此它稍快于 Q31 的复数 FFT。
- 输出数的长度不会像 CMSIS-DSP 函数那样减半。对于复数 FFT 计算，不需要特殊的格式。

6.3 浮点快速傅里叶变换

PowerQuad 硬件不直接支持浮点 FFT。在某些应用中，为了利用 PowerQuad 硬件计算引擎的强大加速能力，同时又能减少代码改动，可以用 PowerQuad 的实现来替换现有的用于浮点 FFT 的 CMSIS-DSP API。这需要数据格式在浮点和定点之间进行转换。

幸运的是，PowerQuad 的矩阵缩放函数能够用硬件实现格式转换。它比 ARM-CMSIS DSP API 的 `arm_float_to_q31()`/`arm_q31_to_float()`运行得更快。为了将下列的操作连接一起，包括将浮点输入数据转换为定点输入数据、定点 FFT 以及定点输出转换为浮点输出，创建了一个全部基于 PowerQuad 硬件的浮点 FFT 函数。

6.3.1 使用 PowerQuad 矩阵缩放函数进行格式转换

CMSIS-DSP 包含将浮点数转换为定点数的 API，例如：`arm_float_to_q31()`和 `arm_q31_to_float()`。在 PowerQuad 模块中，将输入和输出设置成不同的数据格式，并执行缩放比为 1.0f 的矩阵缩放操作。数值不能在输入和输出中更改。当把数值从输入缓冲区转移到输出缓冲区时，该转换会自动执行。

浮点值和定点值之间格式转换的示例代码为：

```
/* app_powerquad_format_switch.c */
#include "app.h"

extern uint32_t timerCounter;
```

```
extern float    inputF32[APP_FFT_LEN_512*2];
extern float    outputF32[APP_FFT_LEN_512*2];
extern q31_t    inputQ31[APP_FFT_LEN_512*2];
extern q31_t    outputQ31[APP_FFT_LEN_512*2];

/* input */
void App_PowerQuad_float_to_q31_Example(void)
{
   uint32_t i;
   pq_config_t pq_cfg;

   PRINTF("%s\r\n",   func  );

   /* input. */
   for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
        inputF32[i*2 ] = (1.0f + i%2); /* real part. */
        inputF32[i*2+1] = 0.0f;    /* imaginary part. */
        inputQ31[i*2 ] = 0; /* clear output. */
        inputQ31[i*2+1] = 0;
    }

    /* convert the data. */
    PQ_Init(POWERQUAD);
    pq_cfg.inputAFormat = kPQ_32Bit; /* input. */
    pq_cfg.inputAPrescale = 0;
    pq_cfg.outputFormat = kPQ_Float; /* output */
     pq_cfg.outputPrescale = 0;
     pq_cfg.machineFormat = kPQ_Float;
     PQ_SetConfig(POWERQUAD,    &pq_cfg);

     TimerCount_Start();
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32    , inputQ31    ); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32+256, inputQ31+256); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32+512, inputQ31+512); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32+768, inputQ31+768); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     TimerCount_Stop(timerCounter);

     /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
     PRINTF("Output :\r\n");
     for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
          PRINTF("%4d: 0x%x, 0x%x\r\n", i, inputQ31[2*i], inputQ31[2*i+1]);
    }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
    PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
    PRINTF("\r\n");
}

/* output */
void App_PowerQuad_q31_to_float_Example(void)
{
   uint32_t i;
   pq_config_t pq_cfg;
```

```
    PRINTF("%s\r\n",   func  );
    /* input. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
        outputQ31[2*i ] = (1 + i%2); /* real part. */
        outputQ31[2*i+1] = 0;    /* imaginary part. */
        outputF32[2*i ] = 0.0f;    /* clear output. */
        outputF32[2*i+1] = 0.0f;
    }

     /* convert the data. */
     PQ_Init(POWERQUAD);
     pq_cfg.inputAFormat = kPQ_32Bit;
     pq_cfg.inputAPrescale = 0;
     pq_cfg.outputFormat = kPQ_Float;
     pq_cfg.outputPrescale = 0;
     pq_cfg.machineFormat = kPQ_Float;
     PQ_SetConfig(POWERQUAD,    &pq_cfg);

     TimerCount_Start();
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31    , outputF32    ); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+256, outputF32+256); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+512, outputF32+512); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+768, outputF32+768); /* 256 items. */
     PQ_WaitDone(POWERQUAD);
     TimerCount_Stop(timerCounter);

     /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
     PRINTF("Output :\r\n");
     for (i = 0u; i < APP_FFT_LEN_512; i++)
     {
        PRINTF("%4d: %f, %f\r\n", i, outputF32[2*i], outputF32[2*i+1]);
      }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
      PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
      PRINTF("\r\n");
}

/* EOF. */
```

图 15 显示了结果。

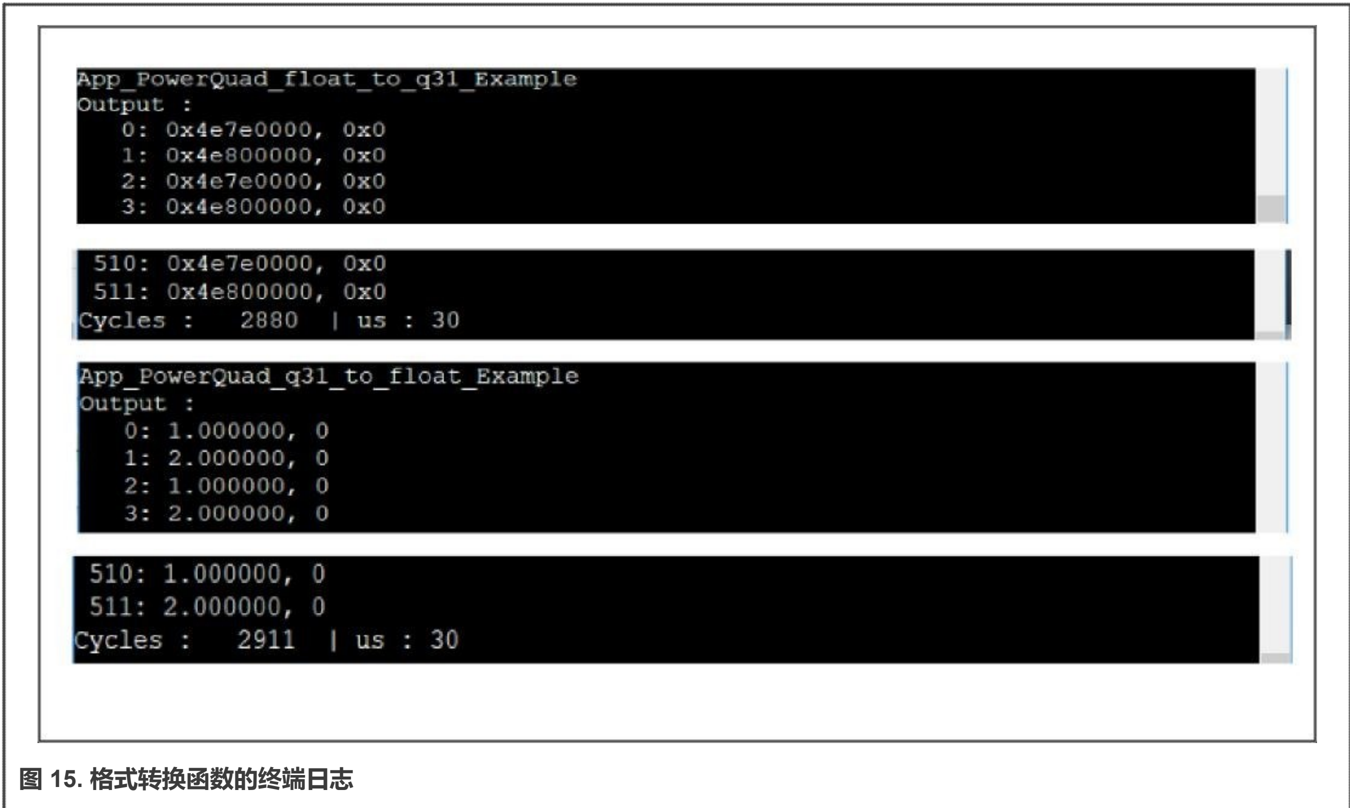

图 15. 格式转换函数的终端日志

使用 ARM-CMSIS DSP 的 API 运行相同的测试实例。如果不进行编译优化，则 `arm_float_to_q31()`和 `arm_q31_to_float()`的速度比 PowerQuad 的转换函数慢。使用转换函数时，有下列限制：

- 对于 CMSIS-DSP 的 API，定点数要遵循标准 q31 格式，其范围必须在（-1，1）之间。
- 对于 PowerQuad 的 API，数组的最大长度为 256。要处理更长的数组，则要多次调用矩阵缩放函数。

6.3.2 F32 类型的复数 FFT 计算

在这个实例中，操作步骤如下：

1. 调用 256 点矩阵缩放函数四次。
2. 将 512 个浮点输入复数（数组中的 1024 个数）转换为定点输入数。
3. 运行硬件 FFT。
4. 获得输出定点数。
5. 调用 256 点矩阵缩放函数四次。
6. 获得浮点输出数。

该任务的代码是：

```
/* app_powerquad_cfft_f32.c */
#include "app.h"

extern uint32_t   timerCounter;

extern float32_t  inputF32[APP_FFT_LEN_512*2];
extern float32_t outputF32[APP_FFT_LEN_512*2];
extern q31_t      inputQ31[APP_FFT_LEN_512*2];
```

```
extern q31_t     outputQ31[APP_FFT_LEN_512*2];

void App_PowerQuad_CFFT_F32_Example(void)
{
   uint32_t i;

   PRINTF("%s\r\n",   func  );

   /* input. */
   for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
 #if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
         inputF32[2*i  ] = (1.0f + i%2); /* real part. */
#else
         inputF32[2*i  ] = APP_FFT_LEN_512 * (1.0f + i%2); /* real part. */
  #endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
         inputF32[2*i+1] = 0; /* imaginary part. */
      }
      memset(inputQ31 , 0, sizeof(inputQ31 )); /* clear input. */
      memset(outputQ31, 0, sizeof(outputQ31)); /* clear output. */
      memset(outputF32, 0, sizeof(outputF32)); /* clear output. */

      /* initialize the PowerQuad hardware. */
      PQ_Init(POWERQUAD);

      TimerCount_Start();

      /* convert the floating numbers into q31 numbers with PowerQuad. */
      {
          pq_config_t pq_cfg;

          pq_cfg.inputAFormat = kPQ_Float; /* input. */
          pq_cfg.inputAPrescale = 0;
          pq_cfg.inputBFormat = kPQ_32Bit; /* no use. */
          pq_cfg.inputBPrescale = 0;
          pq_cfg.tmpFormat = kPQ_32Bit; /* no use. */
          pq_cfg.tmpPrescale = 0;
          pq_cfg.outputFormat = kPQ_32Bit; /* output. */
          pq_cfg.outputPrescale = 0;
          pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
          pq_cfg.machineFormat = kPQ_Float;
          PQ_SetConfig(POWERQUAD, &pq_cfg);

          /* total 1024 items for 512-point CFFT. */
          PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32    , inputQ31    ); /* 256
items. */
          PQ_WaitDone(POWERQUAD);
          PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32+256, inputQ31+256); /* 256
items. */
          PQ_WaitDone(POWERQUAD);
          PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32+512, inputQ31+512); /* 256
items. */
          PQ_WaitDone(POWERQUAD);
          PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, inputF32+768, inputQ31+768); /* 256
items. */
          PQ_WaitDone(POWERQUAD);
      }

      /* computing by PowerQuad hardware. */
      {
```

```
          pq_config_t pq_cfg;

          pq_cfg.inputAFormat = kPQ_32Bit;
 #if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
          pq_cfg.inputAPrescale = 9; /* 2 ^9 for 512 len of input. */
#else
          pq_cfg.inputAPrescale = 0;
 #endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
          //pq_cfg.inputBFormat = kPQ_32Bit;
          //pq_cfg.inputBPrescale = 0;
          pq_cfg.tmpFormat = kPQ_32Bit;
          pq_cfg.tmpPrescale = 0;
          pq_cfg.outputFormat = kPQ_32Bit;
          pq_cfg.outputPrescale = 0;
          pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
          pq_cfg.machineFormat = kPQ_32Bit;
          PQ_SetConfig(POWERQUAD, &pq_cfg);

          PQ_TransformCFFT(POWERQUAD, APP_FFT_LEN_512, inputQ31, outputQ31);
          PQ_WaitDone(POWERQUAD);
        }

        /* convert the q31 numbers into floating numbers. */
        {
           pq_config_t pq_cfg;

           pq_cfg.inputAFormat = kPQ_32Bit;
           pq_cfg.inputAPrescale = 0;
           pq_cfg.inputBFormat = kPQ_32Bit; /* no use. */
           pq_cfg.inputBPrescale = 0;
           pq_cfg.tmpFormat = kPQ_Float; /* no use. */
           pq_cfg.tmpPrescale = 0;
           pq_cfg.outputFormat = kPQ_Float;
           pq_cfg.outputPrescale = 0;
           pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
           pq_cfg.machineFormat = kPQ_Float;
           PQ_SetConfig(POWERQUAD, &pq_cfg);

           PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31    , outputF32    ); /* 256
items. */
           PQ_WaitDone(POWERQUAD);
           PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+256, outputF32+256); /* 256
items. */
           PQ_WaitDone(POWERQUAD);
           PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+512, outputF32+512); /* 256
items. */
           PQ_WaitDone(POWERQUAD);
           PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+768, outputF32+768); /* 256
items. */
           PQ_WaitDone(POWERQUAD);
        }

        TimerCount_Stop(timerCounter);

        /* output. */
 #if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
        PRINTF("Output :\r\n");
        for (i = 0u; i < APP_FFT_LEN_512; i++)
        {
            PRINTF("%4d: %f, %f\r\n", i, outputF32[2*i], outputF32[2*i+1]);
```

```
        }
 #endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
     PRINTF("Cycles : %6d  | us : %d\r\n", timerCounter, timerCounter/96u);
     PRINTF("\r\n");
}
/* EOF. */
```

图 16 显示了结果。

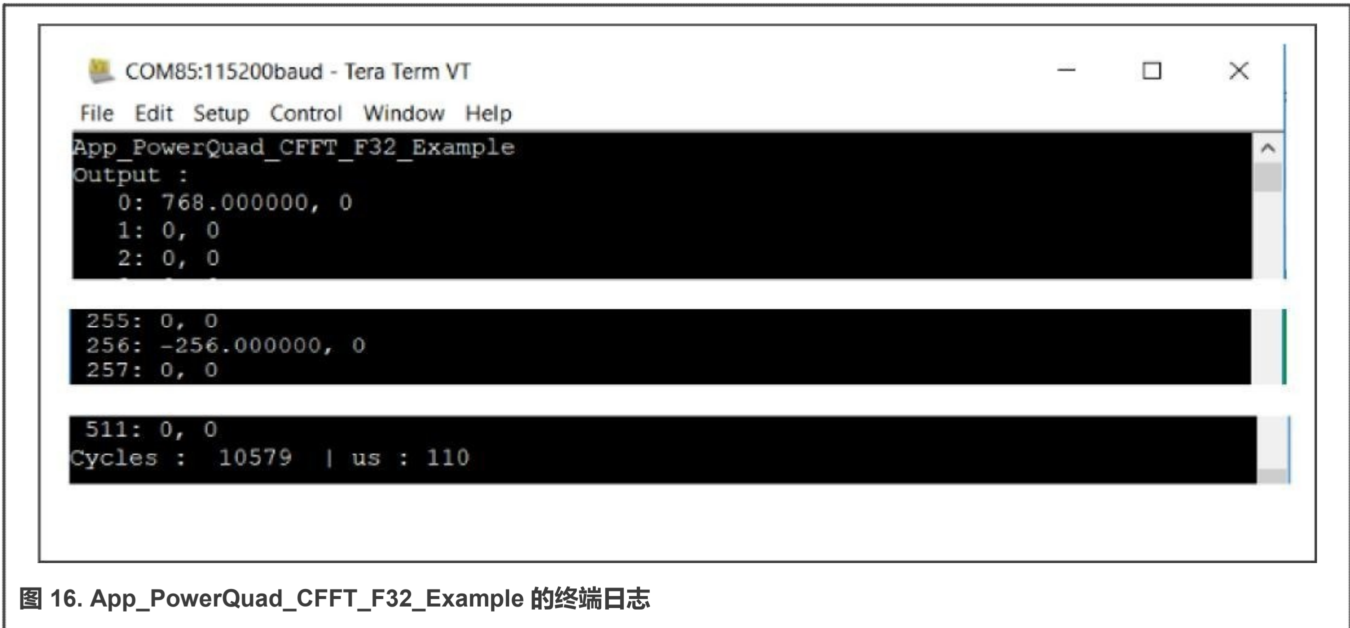

图 16. App_PowerQuad_CFFT_F32_Example 的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 结果是正确的，与 CMSIS-DSP 的浮点复数 FFT 结果相同。
- 硬件转换函数运行良好。
- 它运行的时间几乎和 2 × PowerQuad 矩阵缩放 + 1 × PowerQuad CFFT 的时间相等。且看起来比 CMSIS-DSP 中的 `arm_cfft_f32()`函数更快。

6.3.3 F32 类型的实数 FFT 计算

在这个实例中，操作步骤如下：

1. 将打包的实数浮点数数组转换为 Q31 数。
2. 使用 PowerQuad 的 FFT 引擎用 `PQ_TransformRFFT()`函数进行计算。
3. 获得 Q31 数的输出。
4. 使用 PowerQuad 矩阵缩放函数。
5. 转换为浮点格式。

该任务的代码是：

```
/* app_powerquad_rfft_f32.c */
#include "app.h"

extern uint32_t timerCounter;

extern float32_t inputF32[APP_FFT_LEN_512*2];
extern float32_t outputF32[APP_FFT_LEN_512*2];
extern q31_t    inputQ31[APP_FFT_LEN_512*2];
```

```
extern q31_t    outputQ31[APP_FFT_LEN_512*2];

void App_PowerQuad_RFFT_F32_Example(void)
{
   uint32_t i;

   PRINTF("%s\r\n",   func  );

   /* input. */
   for (i = 0u; i < APP_FFT_LEN_512; i++)
   {
#if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
        inputF32[i ] = (1.0f + i%2); /* only real part. */
#else
        inputF32[i ] = APP_FFT_LEN_512 * (1.0f + i%2); /* real part. */
#endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
    }
    memset(inputQ31 , 0, sizeof(inputQ31 )); /* clear input. */
    memset(outputQ31, 0, sizeof(outputQ31)); /* clear output. */
    memset(outputF32, 0, sizeof(outputF32)); /* clear output. */

    /* initialize the PowerQuad hardware. */
    PQ_Init(POWERQUAD);

    /* convert the floating numbers into q31 numbers. */
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {

        inputF32[i ] = inputF32[i ] / 512 / 8 / 512 / 1024; /* make all the input is in (-1, 1). */
        //PRINTF("[%4d]: %f\r\n", i, inputF32[i]);
     }
     //PRINTF("\r\n");

     TimerCount_Start();
     arm_float_to_q31(inputF32, inputQ31, APP_FFT_LEN_512); /* use arm converter function here. */

     /* computing by PowerQuad hardware. */
    {
         pq_config_t pq_cfg;
         pq_cfg.inputAFormat = kPQ_32Bit;
 #if defined(APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER) && (APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER==1)
         pq_cfg.inputAPrescale = 9; /* 2 ^9 for 512 len of input. */
#else
         pq_cfg.inputAPrescale = 0;
 #endif /* APP_CFG_POWERQUAD_ENABLE_HW_PRESCALER */
         pq_cfg.tmpFormat = kPQ_32Bit;
         pq_cfg.tmpPrescale = 0;
         pq_cfg.outputFormat = kPQ_32Bit;
         pq_cfg.outputPrescale = 0; /* restore the effect of pre-divider. */
         pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
         pq_cfg.machineFormat = kPQ_32Bit;
         PQ_SetConfig(POWERQUAD, &pq_cfg);
         PQ_TransformRFFT(POWERQUAD, APP_FFT_LEN_512, inputQ31, outputQ31);
         PQ_WaitDone(POWERQUAD);
     }

     /* convert the q31 numbers into floating numbers. */
    {
         pq_config_t pq_cfg;
```

```
        pq_cfg.inputAFormat = kPQ_32Bit;
        pq_cfg.inputAPrescale = 0;
        pq_cfg.tmpFormat = kPQ_Float;
        pq_cfg.tmpPrescale = 0;
        pq_cfg.outputFormat = kPQ_Float;
        pq_cfg.outputPrescale = 0;
        pq_cfg.tmpBase = (uint32_t *)0xE0000000; /* private ram. */
        pq_cfg.machineFormat = kPQ_Float;
        PQ_SetConfig(POWERQUAD, &pq_cfg);

        PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31    , outputF32    ); /* 256
items. */
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+256, outputF32+256); /* 256
items. */
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+512, outputF32+512); /* 256
items. */
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, (16u << 8u) | 16u, 1.0f, outputQ31+768, outputF32+768); /* 256
items. */
        PQ_WaitDone(POWERQUAD);
    }

    TimerCount_Stop(timerCounter);

    /* output. */
#if defined(APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS) && (APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS==1)
    PRINTF("Output :\r\n");
    for (i = 0u; i < APP_FFT_LEN_512; i++)
    {
        PRINTF("%4d: %f, %f\r\n", i, outputF32[2*i], outputF32[2*i+1]);
    }
#endif /* APP_CFG_ENABLE_SHOW_OUTPUT_NUMBERS */
    PRINTF("Cycles : %6d | us : %d\r\n", timerCounter, timerCounter/96u);
    PRINTF("\r\n");
}

/* EOF. */
```

图 17 显示了结果。

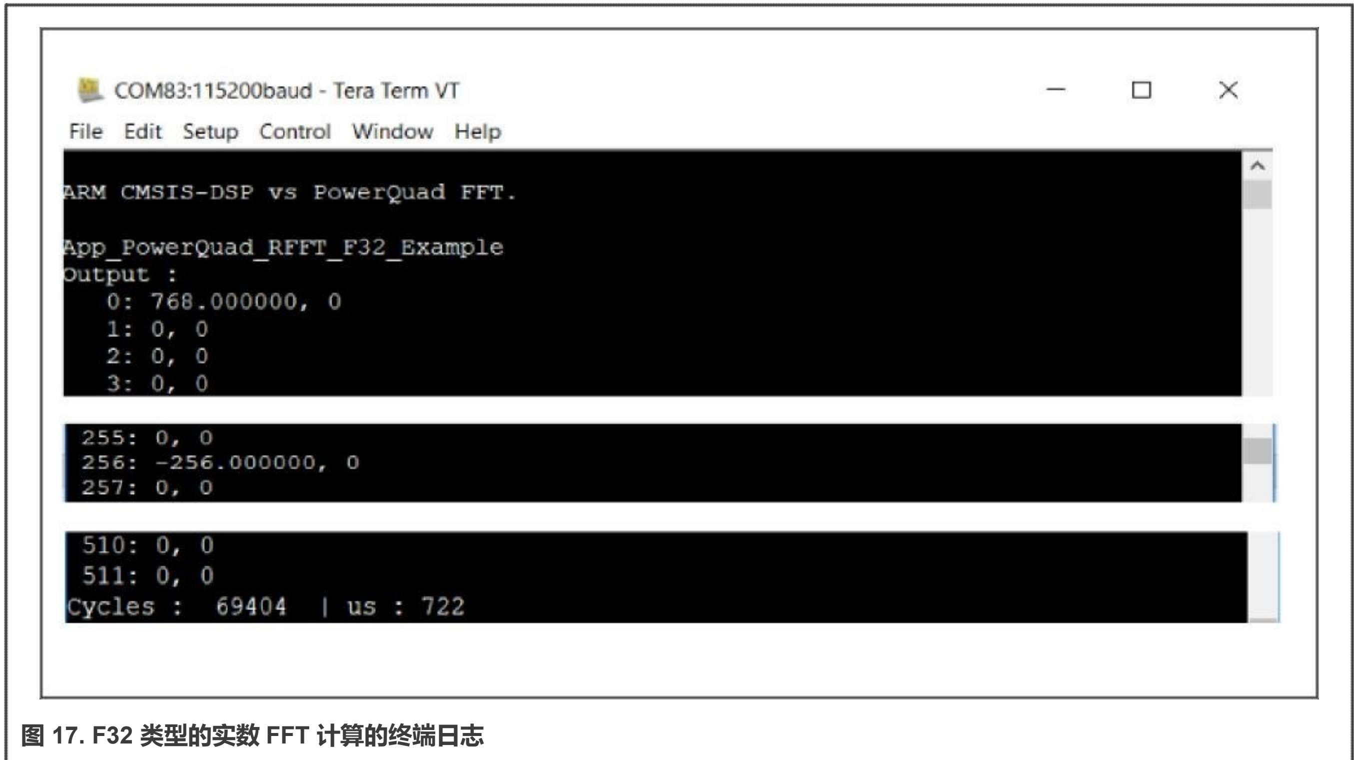

图 17. F32 类型的实数 FFT 计算的终端日志

根据这个实例的代码和终端日志，我们可以看到：

- 根据 Arm CMSIS-DSP 的数字转换，要将输入数缩小到（-1，1）的范围。数字转换的输出是严格的 Q31 数。使用了类整数的定点数（`q0` 格式），并对输入浮点数进行了额外的缩小。接着，就可以像其他演示实例一样达到通用目标。
- 由于采用了该变通方法，导致 `arm_flaot_to_q31()`函数占用了整个过程的大部分时间。即使如此，PowerQuad 的计算速度仍然比纯软件实现更快。关于运行时间的比较，请参见总结和结论。

7 总结和结论

本应用笔记讲述了在同一计算实例中使用 CMSIS-DSP 软件和 PowerQuad 硬件计算 FFT 的用法。对于相同格式的输入和输出计算 FFT 时，PowerQuad 硬件可以替代 CMSIS-DSP 软件。演示实例表明 PowerQuad 的运行速度比 CMSIS-DSP 快得多。

本节列出了这些演示实例的时间特性表。这些汇总表显示了 PowerQuad 的累加能力。可以在 IAR IDE 的项目选项对话框中设置不同的编译优化条件，如图 18 所示。

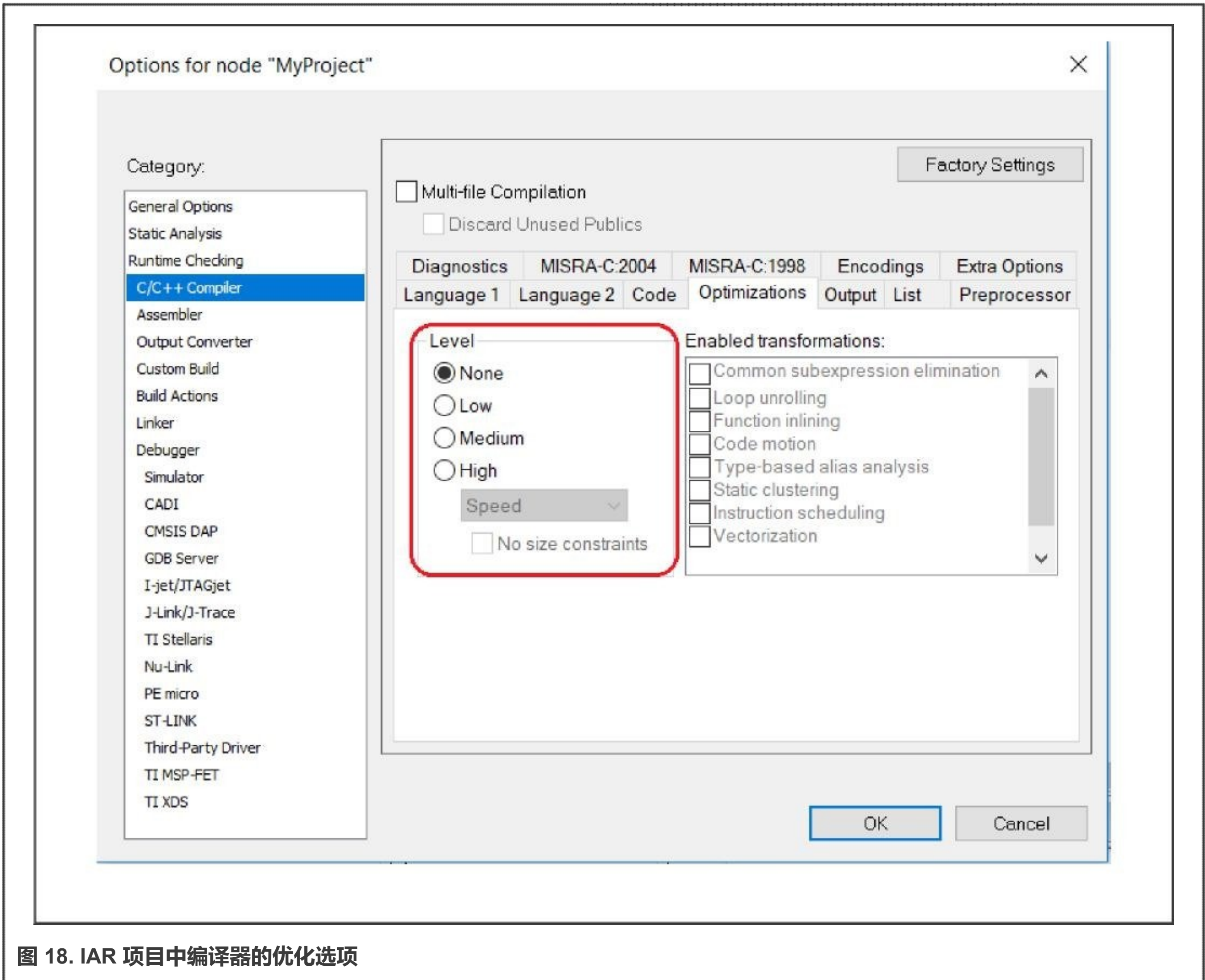

图 18. IAR 项目中编译器的优化选项

表 5 汇总了测量时间。

表 5. 不同优化条件下的测量时间

演示实例	无		低		中		高（速度）		无（禁用 FPU）	
	周期	μs	周期	μs	周期	μs	周期	μs	周期	μs
App_CmsisDsp_CFFT_F32_Example	545274	5679	392081	4084	310262	3231	291130	3032	3382749	35236
App_CmsisDsp_CFFT_Q31_Example	616859	6425	420576	4381	324477	3379	298884	3113	610091	6355
App_CmsisDsp_CFFT_Q15_Example	375995	3916	180156	1876	189941	1978	145103	1511	371291	3867

表格续下页......

表 5. 不同优化条件下的测量时间（续）

演示实例	无		低		中		高（速度）		无（禁用 FPU）	
	周期	μs	周期	μs	周期	μs	周期	μs	周期	μs
App_CmsisDsp_RFFT_Fast_F32_Example	331456	3452	232862	2425	165032	1719	155098	1615	2293419	23889
App_CmsisDsp_RFFT_Q31_Example	428229	4460	330874	3446	263057	2740	246746	2570	418553	4359
App_CmsisDsp_RFFT_Q15_Example	228254	2377	132360	1378	135290	1409	89941	936	240691	2507
App_PowerQuad_CFFT_Q31_Example	3469	36	3465	36	3465	36	3455	35	3468	36
App_PowerQuad_RFFT_Q31_Example	3308	34	3276	34	3174	33	3201	33	3338	34
App_PowerQuad_CFFT_Q15_Example	3500	36	3465	36	3464	36	3455	35	3500	36
App_PowerQuad_RFFT_Q15_Example	3307	34	3277	34	3205	33	3200	33	3338	34
App_PowerQuad_CFFT_F32_Example	10459	108	10698	111	10748	111	10626	110	10758	112
App_PowerQuad_RFFT_F32_Example	61641	642	58216	606	65702	684	35064	365	191849	1998
App_CmsisDsp_float_to_q31_Example	114621	1193	114988	1197	155050	1615	91759	955	417532	4349
App_CmsisDsp_q31_to_float_Example	39062	406	23400	243	10525	109	19175	199	333258	3471
App_PowerQuad_float_to_q31_Example	3005	31	3083	32	3060	31	2983	31	3051	31
App_PowerQuad_q31_to_float_Example	3002	31	3051	31	3028	31	3012	31	3019	31

从表 5 可以看出：

- PowerQuad 的计算速度比 CMSIS-DSP 函数快得多，测量值大约快 100 倍。
- 对于不同格式数据和不同编译优化条件的 FFT 计算，PowerQuad 的时间性能是稳定的。但是，CMSIS-DSP 软件的性能会因编译优化条件的不同而不同。在使用 CMSIS-DSP 软件来实现时，更高级别的优化并不会使代码运行更快（在 `App_CmsisDsp_CFFT_Q15_Example` 中，低级别的优化运行时间为 1876 μs，而中级别的优化运行时间为 1978 μs）。
- 定点计算并不比浮点计算更快。当硬件 FPU 被禁用时，用常规定点指令进行浮点计算需要更多的 CPU 周期。此时，定点算法将运行得更加流畅。当编译器启用 FPU 后，浮点计算指令将节省更多时间，并直接在一个计算指令中计算浮点数，而定点计算则需要更多的指令将大量的计算转换为多个步骤，从而花费更多的时间。因此，当编译器启用 FPU 时，演示实例 `App_CmsisDsp_CFFT_F32_Example` 的运行速度比 `App_CmsisDsp_CFFT_Q31_Example` 更快，但是在禁用 FPU 时，其运行则要慢得多。

- 浮点数和定点数之间的格式转换需要花费大量时间。CMSIS-DSP 软件和 PowerQuad 硬件都是如此，处于同一水平。
- 对于 `App_PowerQuad_RFFT_F32_Example` 演示实例，即使使用了有关格式转换问题的软件变通方法，并用一部分 ARMCMSIS-DSP 的实现进行了替代，它仍比纯软件方式快大约 3 倍。由于复数浮点 FFT 运行速度较快且需要更少的额外内存，因此更推荐使用。也可以在应用程序中将原始数据格式修改为定点数来达到最佳性能。

当以 150 MHz 内核时钟运行时，相应记录如表 6 所示。

表 6. 使用 150 MHz 内核时钟在各种条件下的测量时间

演示实例	无		低		中		高（速度）		无（禁用 FPU）	
	周期	µs	周期	µs	周期	µs	周期	µs	周期	µs
`App_CmsisDsp_CFFT_F32_Example`	239309	1595	169895	1132	136581	910	130355	869	434728	2898
`App_CmsisDsp_CFFT_Q31_Example`	279582	1863	161018	1307	160515	1070	140809	938	279516	1863
`App_CmsisDsp_CFFT_Q15_Example`	184759	1231	95802	638	96057	640	74689	497	74839	498
`App_CmsisDsp_RFFT_Fast_F32_Example`	146585	977	106645	710	78675	524	73689	491	272143	1814
`App_CmsisDsp_RFFT_Q31_Example`	174190	1161	135846	905	111712	744	108408	722	106262	708
`App_CmsisDsp_RFFT_Q15_Example`	110248	734	67754	451	64548	430	50829	338	50920	339
`App_PowerQuad_CFFT_Q31_Example`	3349	22	3356	22	3341	22	3335	22	3344	22
`App_PowerQuad_RFFT_Q31_Example`	3088	20	3072	20	3046	20	3039	20	3039	20
`App_PowerQuad_CFFT_Q15_Example`	3372	22	3345	22	3352	22	3334	22	3333	22
`App_PowerQuad_RFFT_Q15_Example`	3088	20	3073	20	3045	20	3039	20	3039	20
`App_PowerQuad_CFFT_F32_Example`	8819	58	8794	58	8910	59	8802	58	8677	57
`App_PowerQuad_RFFT_F32_Example`	36332	242	39369	262	36163	241	24399	162	33885	225
`App_CmsisDsp_float_to_q31_Example`	73151	487	72612	484	72870	485	42636	284	56703	378
`App_CmsisDsp_q31_to_float_Example`	28315	188	28288	188	10033	66	9577	63	38817	258

表格续下页......

表 6. 使用 150 MHz 内核时钟在各种条件下的测量时间（续）

演示实例	无		低		中		高（速度）		无（禁用 FPU）	
	周期	μs	周期	μs	周期	μs	周期	μs	周期	μs
App_PowerQuad_float_to_q31_Example	2505	16	2512	16	2521	16	2477	16	2422	16
App_PowerQuad_q31_to_float_Example	2502	16	2525	16	2520	16	2470	16	2423	16

8 修订历史

版本号	日期	说明
第 0 版	2021 年 12 月 31 日	初版发布

How To Reach Us

Home Page:

nxp.com.cn

Web Support:

nxp.com.cn/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.cn/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.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX,EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE, VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. M, M Mobileye and other Mobileye trademarks or logos appearing herein are trademarks of Mobileye Vision Technologies Ltd. in the United States, the EU and/or other jurisdictions.

© NXP B.V. 2021. All rights reserved.

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

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

Date of release: 31 December 2021

Document identifier: AN13496